



Compilateur de parties contrôle de microprocesseurs

Noureddine Mhaya

► To cite this version:

Noureddine Mhaya. Compilateur de parties contrôle de microprocesseurs. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 1988. Français. NNT: . tel-00329951

HAL Id: tel-00329951

<https://theses.hal.science/tel-00329951>

Submitted on 13 Oct 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée par

Noureddine MHAYA

pour obtenir le titre de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(arrêté ministériel du 5 Juillet 1984)

Spécialité: **Informatique**

COMPILATEUR DE PARTIES CONTROLE

DE MICROPROCESSEURS

Date de soutenance: 24 Juin 1988

Composition du jury:

Messieurs	Gérard NOGUEZ	Président et rapporteur
	Bernard COURTOIS	
	Ahmed Amine JERRAYA	
	Christian LANDRAULT	Rapporteur
	Guy MAZARE	
	Jean Pierre MOREAU	

Thèse préparée au sein du laboratoire TIM3.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD

Année 1988

Professeurs des Universités

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

**Professeur Université des Sciences
Sociales
(Grenoble II)**

BOLLIET Louis

**Personnes ayant obtenu le diplôme
d'HABILITATION A DIRIGER DES
RECHERCHES**

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

**Chercheurs du C.N.R.S
Directeurs de recherche 1ère Classe**

CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
LANDAU Ioan
VACHAUD Georges
VERJUS Jean-Pierre

**Directeurs de recherche
2ème Classe**

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques

COURTOIS Bernard
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GUELIN Pierre
JODD Jean-Charles
KLEITZ Michel
KOFMAN Walter
KAMARINOS Georges
LEJEUNE Gérard
LE PROVOST Christian
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MUNIER Jacques
PIAU Monique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
WACK Bernard

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

E.N.S.E.E.G

CHATILLON Christian
HAMMOU Abdelkader
MARTIN GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G

BOREL Joseph

E.N.S.I.E.G

DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G

ROWE Alain

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

COURTIN Jacques

E.F.P.

CHARUEL Robert

C.E.N.G

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

Laboratoires extérieurs

C.N.E.T

DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

Je tiens à remercier

M. Gérard NOGUEZ, Professeur à l'université PARIS VI, de m'avoir fait l'honneur de présider le jury et d'être rapporteur de cette thèse.

M. Bernard COURTOIS, Directeur de recherche au C.N.R.S et directeur-adjoint du laboratoire TIM3, de m'avoir accueilli dans l'équipe d'architecture des ordinateurs et pour l'intérêt qu'il a porté à mon travail.

M. Ahmed Amine JERRAYA, chargé de recherche au C.N.R.S, pour l'intérêt permanent qu'il a porté à mon travail tout au long de la période que j'ai passée dans l'équipe qu'il dirige.

M. Christian LANDRAULT, directeur de recherche au C.N.R.S, qui a accepté d'être rapporteur et membre du jury de cette thèse.

M. Guy MAZARE, professeur à l'ENSIMAG, d'avoir accepté d'être membre du jury de cette thèse.

M. Jean Pierre MOREAU, chef du département aide à la conception de circuits intégrés de SGS Thomson, d'avoir accepté d'être membre du jury de cette thèse.

Tous les membres de l'équipe d'architecture des ordinateurs pour leur sympathie et toutes les discussions qui ont contribué à mener à bien ce travail.

Le personnel du service reprographie de l'IMAG pour le tirage de cette thèse.

**à ma mère,
mes frères et sœurs**



**L'homme, sans aucun appui
et sans aucun secours, est
condamné à chaque instant à
inventer l'homme.**

J.P. S

Résumé:

Cette thèse consiste à présenter le compilateur de parties contrôle CPC. Cet outil a été développé dans le cadre du développement du compilateur de silicium SYCO au laboratoire TIM3-IMAG-INPG.

SYCO est un compilateur qui génère le dessin des masques de circuits du type microprocesseur à partir d'une description structurelle et comportementale. Les circuits générés sont composés d'une partie opérative et d'une partie contrôle. La partie contrôle est composée d'une pile d'étages de contrôle. Chaque étage est réalisé par un PLA unique.

La CPC reçoit une description hiérarchique du circuit. Sa tâche consiste à générer, à partir de la description de chaque étage de contrôle, une description logique permettant la génération du dessin des masques du PLA réalisant l'étage. Pour chaque étage de contrôle, le processus de compilation est composé de trois étapes: génération de la table d'états, optimisation logique et ajout des branchements implicites, génération de la description logique du PLA réalisant l'étage. Le CPC génère aussi la description de la partie opérative, permettant au compilateur de partie opérative APOLLON de générer le dessin des masques de la partie opérative, et un fichier de communication permettant l'assemblage des étages de contrôle et de la partie opérative.

La génération automatique d'autres structures d'étages de contrôle est présentée dans cette thèse, notamment des étages microprogrammés.

Mots clés:

Compilateur de silicium, partie contrôle, PLA, table d'états, optimisation, automate, hiérarchie.

ABSTRACT

This thesis presents the control part compiler CPC. This tool was developed as part of the SYCO silicon compiler in TIM3-IMAG-INPG laboratory.

SYCO is a compiler which generates the layout of microprocessor-like circuits from a structural and behavioral description. Circuits generated are composed of a data path and a control part. The control part is made up of a stack of control slices. Each one is achieved by a single PLA.

The CPC receives a hierarchical description of the circuit. Its task consists in generating, from the description of each control slice, the logic description which allows the layout generation of the PLA achieving the slice.

For each slice, the compilation process is composed of three steps : state table generation, logic optimization and completion of the implicit jumps, generation of logic description of the PLA.

The CPC generates also the data path description, which allows the data path compiler APOLLON to generate the data path layout, and the communication file which allows to assemble the control slices and the data path.

Automatic generation of other control slice structures is presented in this thesis, especially a microprogrammed control slices.

Keywords:

Silicon compilation, control part, PLA, synthesis, optimization.

INTRODUCTION

Les progrès enregistrés dans la technologie de fabrication des circuits intégrés se sont concrétisés par l'avènement des circuits VLSI. En effet, depuis les années 60, le nombre de transistors qu'on peut intégrer sur une seule puce ne cesse de croître. Il a doublé tous les ans au cours des années 60. Vers 1972, ce rythme de croissance a connu un ralentissement (on avait atteint 10^4 composants par puce en 1972). Néanmoins, cette croissance reste spectaculaire et on croit pouvoir intégrer jusqu'à 10^7 composants par puce en l'an 1990. Par conséquent l'intégration à très grande échelle GSI (Giga Scale Integration), qui permettra d'intégrer un milliard de composants par puce, devrait voir le jour en l'an 2000.

Le fait de pouvoir intégrer plusieurs millions de transistors par puce entraîne une croissance continue de la complexité des circuits et une diversification des fonctions qu'ils réalisent. Par conséquent, la conception de tels circuits va demander des dizaines d'hommes/année, et autant pour les vérifier. Cependant, pour pouvoir exploiter les potentialités de l'évolution technologique, il est nécessaire de définir de nouvelles méthodes de conception adaptées à la complexité des circuits à produire et tenant compte des contraintes économiques propres à l'industrie des semi-conducteurs.

Une approche permettant de réduire le temps et le coût de conception des circuits intégrés est la compilation de silicium.

La compilation de silicium est un procédé permettant la génération automatique du dessin des masques d'un système à partir de sa description qui peut être exprimée à différents niveaux d'abstraction.

Le travail présenté dans cette thèse entre dans le cadre du développement du compilateur de silicium SYCO et consiste à développer le compilateur de partie contrôle qui lui est associé.

Le premier chapitre comporte une introduction à la compilation de silicium ainsi qu'un l'historique des compilateurs de silicium. Notre but n'est pas de faire une étude exhaustive sur les compilateurs de silicium, mais de présenter leurs principales caractéristiques et propriétés.

Le deuxième chapitre présente le compilateur de silicium SYCO et situe

le compilateur de partie contrôle CPC dans l'ensemble d'outils qui composent SYCO.

Le chapitre 3 présente les caractéristiques générales d'un compilateur de partie contrôle et rappelle les principales tâches que doit accomplir un tel outil. On y trouvera également un aperçu sur la forme intermédiaire des compilateurs de partie contrôle et une présentation des différentes organisations des parties contrôle.

Le chapitre 4 sera consacré à la présentation détaillée du compilateur de partie contrôle CPC, son langage d'entrée, son principe, sa forme intermédiaire et les différentes transformations appliquées à la forme intermédiaire pour atteindre la description logique permettant la génération du dessin des masques de la partie contrôle.

Enfin, le chapitre 5 propose d'autres structures de partie contrôle qui peuvent être générées automatiquement et qui permettent d'augmenter la vitesse des circuits conçus.

CHAPITRE 1

Introduction à la compilation de Silicium

I - Historique

Le terme "Compilation de Silicium" a été introduit, pour la première fois, par David JOHANNSEN en 1979 [JHO79]. Il l'a utilisé pour désigner un processus permettant l'assemblage paramétré de cellules d'une bibliothèque de cellules pré-caractérisées.

Ce terme a été largement repris par la suite notamment par WALLISH [WAL83] qui définit un compilateur de silicium comme étant un programme capable de produire une description physique d'un circuit à partir d'une spécification de haut niveau. Cette définition inclut un large spectre d'outils de traduction présenté dans [GRO83]. Ce terme n'a pas cessé d'évoluer en fonction du type des circuits générés par le compilateur, de la spécification de départ et de la stratégie de compilation [THO81, GAJ86, GAJ87]. De nos jours, on parle de compilateurs de silicium structurels, comportementaux et intelligents [GAJ86, GAJ86a].

Dans la suite de ce chapitre, on présentera les différents outils que GROSS situait dans le spectre d'outils de traduction et les compilateurs de silicium tels qu'ils sont définis par GAJSKI.

GROSS [GRO83] distingue deux classes de traducteurs :

1^{ère} classe: Les traducteurs à spectre limité :

Ces outils peuvent franchir une étendue limitée du spectre. Ils sont divisés en plusieurs sous ensembles dont:

1- Les systèmes de cellules standard : Ces outils sont les premiers apparus. Un tel système retrouve la description des masques de chaque cellule spécifiée à partir d'une bibliothèque, sélectionne la place de cette description dans le circuit à réaliser, oriente la description des masques dans cette place et assure le routage entre les différentes cellules.

2- Les compilateurs de modules : Ces outils produisent des structures régulières et relativement compactes d'une architecture fixe. Ils sont de quatre types selon la structure régulière qu'ils génèrent.

- a. Systèmes de micro-cellules: Ils automatisent partiellement le routage.
- b. Générateurs de PLA: Les PLAs sont des structures régulières très connues et largement utilisées. Ces structures permettent d'implémenter des fonctions booléennes arbitraires. Les PLAs sont spécialement utilisés

dans la réalisation des parties contrôle de circuits intégrés.

c. Générateurs de SLA (Storage Logic Array): Un SLA est une extension du PLA qui introduit des éléments de mémorisation dans un PLA brisé. Ces outils permettent au concepteur de spécifier la fonction réalisée par le SLA, et dans un niveau de transfert de registres une topologie limitée.

d. Les compilateurs d'expressions régulières: Ce sont des systèmes qui génèrent le layout d'un circuit à partir d'expressions régulières. Ces outils ont été utilisés par FLOYD [FLO80] pour générer des machines d'états finis séquentielles. En effet, FLOYD et ULLMAN ont démontré qu'un circuit implémentant un automate d'états finis non déterministe, construit à partir d'expressions régulières, peut être généré automatiquement et de façon efficace.

3- Les assembleurs de silicium : A la différence des compilateurs de modules, qui ont une structure cible spécifiée et qui sont souvent utilisés pour générer des cellules de contrôle, ces outils produisent des circuits généraux.

Les assembleurs de silicium sont des outils de génération automatique de dessin des masques de circuits décrits par une hiérarchie de blocs. Un bloc peut être:

- i) une cellule de base appartenant à une bibliothèque de cellules propre à la technologie,
- ii) une structure régulière (PLA, ROM, ...),
- iii) un assemblage d'autres blocs.

Les assembleurs permettent au concepteur de décrire et de manipuler les blocs de son circuit. Une description consiste à spécifier les blocs qui composent le circuit et leur interconnexion.

4 - Les synthétiseurs logiques : Ces outils se situent au début du spectre. A partir d'une description de haut niveau, ils permettent d'extraire des équations booléennes. Ces équations peuvent être sous forme canonique permettant leur implémentation par des portes logiques. L'extraction des équations tient compte de la technologie d'implémentation.

2eme classe: Les traducteurs à large spectre:

Comme leur nom l'indique, ces outils couvrent la totalité du spectre,

c'est-à-dire qu'ils partent d'une description de haut niveau pour aboutir au dessin géométrique du circuit à réaliser. Ces outils doivent vérifier certaines propriétés; ainsi les questions suivantes doivent se poser :

- 1) préservent-ils les informations exprimées par leur langage source dans une forme qui peut être exploitée plus tard dans le processus de traduction ?
- 2) présentent-ils une interface de communication avec le concepteur et avec d'autres outils d'aide à la conception tels que les simulateurs ?
- 3) Sont-ils indépendants de la technologie ?

On verra que pour répondre à ces questions, ces outils devront respecter certaines contraintes. On peut d'ores et déjà avancer que le problème de l'indépendance vis-à-vis de la technologie n'est pas aussi simple à résoudre qu'il n'y paraît. En effet, cette indépendance ne pourra être assurée tout au long du processus de traduction puisqu'on doit toujours passer par une description géométrique du circuit, qui dépend nécessairement de la technologie. Ainsi un générateur de PLA en technologie NMOS sera certainement différent d'un générateur de PLA en technologie CMOS.

Pour que ces outils puissent réaliser leur tâche, malgré la complexité du circuit, il est nécessaire de restreindre l'architecture cible de ces traducteurs, ce qui simplifie considérablement le processus de traduction. GROSS sépare ces outils en deux groupes.

1^{er} groupe : Les compilateurs de plan de masse :

Ces outils produisent un dessin géométrique (layout) paramétrable similaire à un plan d'architecte classique. Une caractéristique des compilateurs de plan de masse est qu'ils partent d'une spécification structurelle. Leur architecture cible n'est pas obligatoirement du type microprocesseur. Le premier compilateur de plan de masse est Bristle Blocks qui a été développé par D. JOHANNSEN [JOH79].

On peut aussi citer CAPRI [ANC83] qui a été développé au sein du laboratoire TIM3. Avec CAPRI, la structure du circuit est spécifiée en langage IRENE [MARI86]. Les parties opérative et contrôle sont extraites à partir de cette structure. Des cellules optimisées et paramétrées sont sélectionnées et déformées dans les deux dimensions pour s'insérer efficacement dans la topologie du plan.

L'efficacité de CAPRI vient de la paramétrisation de solutions pré-existantes qui sont améliorées pour être efficaces des points de vue architecture et implantation.

2^e groupe : Les traducteurs de spécifications comportementales :

La plus grande restriction de l'utilisation des compilateurs de plan de masse est que le concepteur est amené à définir lui-même l'architecture de son circuit. La tâche des traducteurs de spécification comportementale consiste à transformer une description comportementale, et non pas structurelle, du circuit en une description géométrique des masques.

Généralement, tous les circuits générés par un tel outil ont la même architecture interne puisque ces outils se fixent une architecture cible. On reprendra plus en détail ce point en présentant le compilateur de silicium SYCO.

II - Les compilateurs de silicium

De nos jours, les compilateurs de silicium sont les outils de CAO les plus généraux. Ils sont utilisés dans la conception de circuits intégrés à application spécifique (ASIC) et permettent aux concepteurs de décrire leurs circuits à un haut niveau d'abstraction, ce qui permet aux utilisateurs non qualifiés dans le domaine de la conception des circuits intégrés de décrire et de concevoir des circuits [GAJ87]. GAJSKI définit un compilateur de silicium comme étant un ensemble de programmes permettant de passer d'une description comportementale ou structurelle d'un système au dessin des masques permettant la fabrication de ce système. Ce processus de traduction est bien représenté par le diagramme en Y de GAJSKI (figure 1.1) [GAJ86, GAJ87].

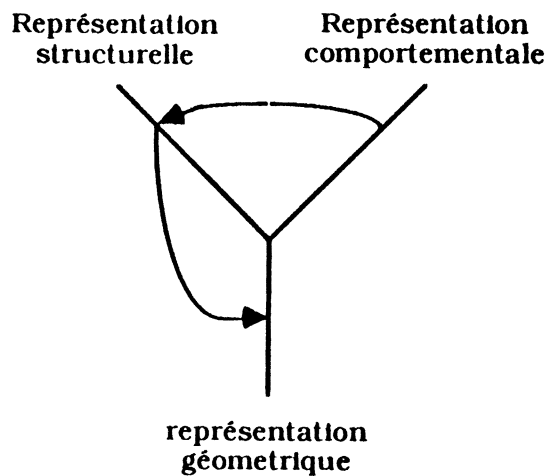


Figure 1.1 Diagramme en Y de GAJSKI

Relativement à la classification donnée par GROSS dans [GRO83], les compilateurs de silicium représentent les traducteurs à large spectre.

Les compilateurs de silicium peuvent être classés selon leurs langages d'entrée. THOMAS [THO81] distingue trois niveaux de description de système:

- description logique,
- description structurelle,
- description comportementale.

Le point commun de ces niveaux est que le système peut être divisé en deux parties (partie opérative et partie contrôle).

a) Le niveau logique : Ce niveau de représentation permet la spécification d'interconnexion des éléments logiques. Les éléments logiques représentent des modules physiques et par conséquent il y a une correspondance directe entre éléments logiques et modules physiques implémentant le système.

b) Le niveau structurel : Ce niveau permet l'abstraction de l'information spécifiée par le niveau logique. Un module dans la représentation structurelle spécifie une primitive de haut niveau tel qu'un additionneur ou un registre. La limite entre ces deux niveaux est floue car les modules du niveau logique peuvent correspondre directement à des opérations spécifiées par le niveau structurel. La différence est que le niveau structurel spécifie une opération et non pas une implémentation spécifique

de l'opération.

c) Le niveau comportemental : Ce niveau permet la spécification de la fonction réalisée par le système sans donner la moindre information explicite sur l'implémentation structurelle du système à réaliser.

Pour développer un compilateur de silicium, l'une des décisions les plus importantes à prendre est le choix d'un langage d'entrée approprié permettant de décrire de façon concise les algorithmes à réaliser par le circuit à fabriquer. Le langage d'entrée doit être suffisamment général pour permettre de décrire la plus large classe de systèmes. [JER86] donne une classification des compilateurs de silicium tenant compte de leurs langages d'entrée.

De nos jours, on parle de trois grandes classes de compilateurs:

- les compilateurs comportementaux,
- les compilateurs structurels,
- les compilateurs intelligents [ORA86].

A) Les compilateurs comportementaux : Les caractéristiques d'un tel compilateur sont [GAJ86]:

- a- le modèle architectural,
- b- la topologie,
- c- la synchronisation,
- d- le langage d'entrée et les outils de support.

Pour un compilateur comportemental, la décision importante est, outre le choix du langage d'entrée, le choix d'une architecture cible permettant d'implémenter correctement les spécifications exprimées dans le langage d'entrée [SIS82]. Ainsi l'architecture cible doit être suffisamment générale pour permettre l'implémentation de tous les systèmes que le langage d'entrée permet de décrire.

La majorité de ces compilateurs adopte l'un des modèles architecturaux suivants:

- 1) chemin de données (DP),
- 2) contrôleur de machine d'états finis(contrôleur de MEF),
- 3) unité de contrôle et un chemin de données (UC+DP),

4) UC + DP et des mémoires.

1) Le premier modèle (DP) est caractérisé par le type d'unités de base utilisées (additionneur, décaleur, ALU...), le type d'éléments de mémorisation (registre, mémoire...) et le type de connexions (bus unidirectionnels, bidirectionnels...). Ce modèle a été adopté par GENESIL et CONCORDE.

2) Le deuxième modèle (contrôleur de MEF) est valable pour la réalisation des parties contrôle de machines séquentielles. La machine d'états finis peut être du type Moore (les sorties ne dépendent que de l'état courant de la machine) ou du type Mealy (les sorties dépendent des entrées et de l'état courant de la machine). Une machine d'états finis peut être implémentée par un PLA, un ROM ou même en logique aléatoire.

3) Le troisième modèle (UC+DP) est surtout utilisé pour la conception de circuits de type microprocesseur, composés d'une partie opérative et d'une partie contrôle. Dans ce modèle, la partie opérative exécute les instructions et la partie contrôle dirige l'exécution de la partie opérative [ANC86]. La communication entre les deux unités est synchrone. Ce modèle a été adopté par MacPitts [SOU83, SIS82], SILC [FOX85, BLA85, FUN86], ALGIC [JOE85, JOE86], et SYCO [JER 86].

4) Le quatrième modèle (DP + CU + mémoires) est une extension du modèle précédent. L'ajout de mémoire aux éléments de mémorisation (registres, bascules ...) rend plus difficile l'optimisation de la structure. CMUDA [HAF82, BLA85] est l'un des rares systèmes basé sur ce modèle architectural.

Le modèle architectural d'un compilateur de silicium peut être basé sur l'un des deux types de topologie suivants:

- la topologie linéaire,
- la topologie aléatoire.

La topologie linéaire est basée sur la structure "bit-slice" où le layout est généré en joignant les "bits-slices" des cellules, ce qui réduit le problème

de routage à un problème unidimensionnel.

Quant à la topologie aléatoire, elle permet d'avoir une plus grande flexibilité du layout mais elle pose un problème bidimensionnel de placement et de routage [GAJ86].

Avec les compilateurs comportementaux, le processus de compilation commence par une spécification comportementale du système à concevoir. Une telle spécification définit avec précision les entrées/sorties du système et la fonction qu'il doit réaliser sans donner aucune information sur sa structure interne. La première tâche du compilateur consiste alors à extraire la structure interne du circuit, en terme de modules physiques qui permettront d'implémenter la fonction réalisée par le circuit, à partir de la spécification comportementale. Pour les compilateurs adoptant le 3^e modèle architectural, cette étape peut être décomposée en deux problèmes distincts:

- la synthèse de la partie opérative,
- La synthèse de la partie contrôle [STE81].

B) Les compilateurs structurels : Avec ces compilateurs, c'est le concepteur qui définit explicitement la micro-architecture du circuit. La spécification de départ consiste à définir les entrées/sorties du circuit global, les entrées/sorties et la fonction réalisée par chaque bloc fonctionnel composant le circuit et tous les liens entre les différents blocs. Dans le cas général, il y a une correspondance directe entre blocs fonctionnels spécifiés par le concepteur et blocs physiques dans le circuit après fabrication. La tâche du compilateur se réduit au placement et au routage des blocs fonctionnels.

C) Les compilateurs intelligents [GAJ86]: Un compilateur intelligent utilise des connaissances qui lui permettent de faire des transformations itératives jusqu'à ce que les contraintes spécifiées par le concepteur soient satisfaites. Un tel système est composé éventuellement de trois tâches :

- le raffinement;
- l'optimisation;
- la stratégie de sélection.

1) **Le raffinement** : C'est un processus de traduction d'une description

comportementale ou fonctionnelle en une structure à base de composants prédéfinis.

2) L'optimisation : C'est une étape qui améliore l'utilisation des ressources et peut être exécutée après ou en parallèle avec le raffinement. On trouve plusieurs types d'optimisation (logique, topologique...).

3) La stratégie de sélection : C'est une séquence de différentes étapes de raffinement et d'optimisation. L'ordre et le type de l'étape dans la séquence caractérisent la stratégie.

III- Conclusion

Différents compilateurs de silicium ont été décrits par leurs auteurs [JOH79, SOU83, KRE86, CAM87, Bla85, PEN86, JOE86, SOL86, FUN86, MAR86]. Les compilateurs intelligents se distinguent des compilateurs classiques par le pouvoir de retour arrière (backtracking). En effet ce retour arrière permet au concepteur de reprendre le processus de conception à partir d'une certaine étape si le résultat obtenu n'est pas satisfaisant ou s'il veut obtenir un autre résultat en effectuant des choix différents.

CHAPITRE 2

Le Compilateur de Silicium

SYCO

SYCO [JER86] est un compilateur de silicium développé par l'équipe de recherche en architecture des ordinateurs du laboratoire TIM3-IMAG-INPG. Cet outil a été développé dans le cadre du projet national SYCOMORE qui regroupe :

- des outils de dessin de masques,
- des outils de compilation de silicium (SYCO),
- des outils de mise au point.

SYCOMORE a réuni au lancement du projet quatre partenaires qui sont BULL, THOMSON, l'INRIA, et l'INPG.

I- Présentation de SYCO

SYCO est un compilateur de silicium destiné à la génération automatique du dessin des masques de machines séquentielles à haute densité. Ces machines sont composées d'une partie opérative et d'une partie contrôle. SYCO génère les spécifications des masques de chaque circuit à partir d'une description comportementale de très haut niveau. Le langage d'entrée de SYCO est LDS (Langage de Description de Systèmes) [LAU85], développé par la compagnie BULL.

II- Modèle d'interprétation

Le modèle d'interprétation de SYCO consiste à produire un ensemble d'interpréteurs qui, juxtaposés, réalisent la même fonction que l'interpréteur spécifié par la description de départ. Ce modèle est illustré par la figure 2.1 où le langage L_n correspond au langage de départ et L_0 est le langage des primitives de base de la machine. L'interpréteur I_n décompose les primitives du langage L_n en primitives du langage L_{n-1} etc... [OBR82, ANC80]. Dans ce modèle, tel qu'il a été présenté par F. ANCEAU, chaque langage correspond à un niveau d'abstraction de la machine. Les langages utilisés appartiennent donc à des niveaux différents. Pour le cas du compilateur SYCO, on verra que les différents interpréteurs sont exprimés dans le même langage, mais le principe d'interprétation reste conforme à ce modèle étant donné qu'une instruction d'un niveau donné peut être exprimée en plusieurs instructions du niveau inférieur. Un appel de procédure dans un niveau correspond à l'exécution de plusieurs instructions du niveau inférieur. Pour l'étage le plus bas, chaque étape

opératives (ensemble d'instructions à exécuter en parallèle) s'exécute en un ou deux cycles. Un cycle correspond à une microcommande et s'exécute en quatre phases d'horloge.

Chacun de ces interpréteurs sera réalisé par un automate appartenant à l'un des deux types suivants :

1) automates de contrôle : Ils transforment une commande d'un niveau d'interprétation en une séquence de commandes du niveau d'interprétation immédiatement inférieur,

2) automate d'exécution : seul l'interpréteur du niveau le plus bas (I_0) est réalisé par ce type d'automate qui permet l'exécution des opérations de base. Ce schéma a été utilisé pour la conception du micro-processeur 8085 d'INTEL où un premier interpréteur décompose une instruction en cycles machine, un deuxième décompose un cycle machine en étapes et un troisième décompose une étape en deux phases d'horloge [ANC80].

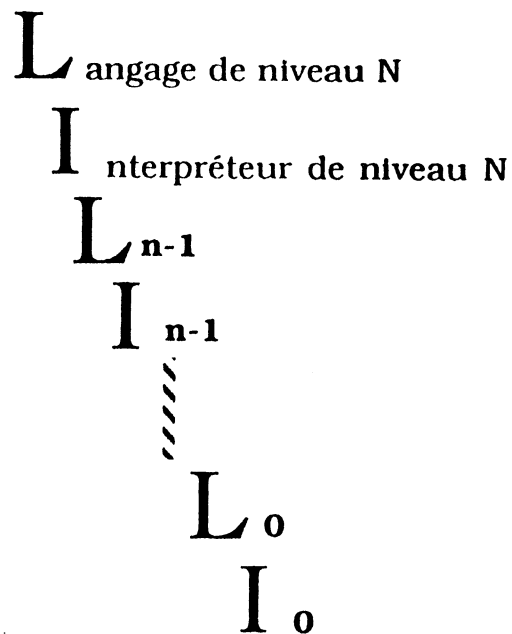


Figure 2.1: Modèle d'interprétation

Ce modèle a aussi été adopté par [EZZ86] qui représente la description du circuit par un GRAFCET. Chaque nœud du GRAFCET est décrit par un autre GRAFCET de niveau d'interprétation inférieur. Cette décomposition permet de hiérarchiser la description du circuit. On remarque aussi que tous les niveaux d'interprétation sont exprimés dans un même niveau

d'abstraction par des GRAFCETs.

III- Modèle architectural

Les circuits générés par SYCO sont composés d'une partie opérative et d'une pile de tranches de contrôle (figure 2.2). Le modèle architectural est directement déduit du modèle d'interprétation précédent. En effet, la partie opérative correspond au dernier interpréteur qui exécute donc les opérations de base de la machine. Les tranches de contrôle correspondent alors aux autres interpréteurs. Chaque tranche reçoit les commandes de l'étage supérieur et génère une séquence de commandes à l'étage inférieur. L'étage de contrôle le plus bas décompose chaque étape opérative en une ou deux microinstructions.

Dans ce modèle, un étage ne peut communiquer qu'avec les étages immédiatement inférieur et supérieur. Donc l'interprétation d'une instruction du langage du niveau le plus haut doit traverser tous les étages de contrôle. De plus, à un instant donné, un seul étage de contrôle est actif à la fois. Chaque étage de contrôle est connecté aux deux bus de contrôle et de compte rendu.

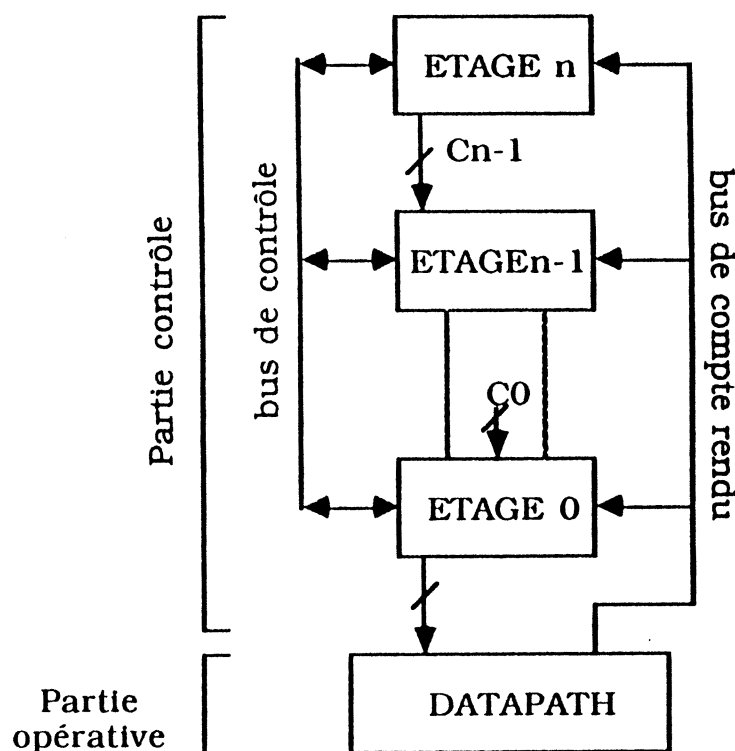


Figure 2.2: Modèle architectural de SYCO

IV- Historique de SYCO

Depuis le lancement de ce projet en 1983, l'équipe chargée de sa réalisation a développé un prototype autour du langage de description **IRENE** [MARI86]. Mais il s'est avéré que ce langage n'était pas assez adapté à l'application et présentait certaines restrictions et insuffisances qu'on verra plus loin. De ce fait, et au début de l'année 1986, le développement d'une nouvelle version de SYCO autour du langage LDS a été décidé. Cette nouvelle version sera présentée au paragraphe V de ce chapitre .

Le langage IRENE [MARI86]:

IRENE est un langage de description et de génération automatique des circuits intégrés VLSI. Il permet la description comportementale à différents niveaux d'abstraction et, à des étapes très avancées du processus de conception, la description structurelle de ces circuits. On commence par la description du comportement à un niveau très haut pour finir avec une description du niveau correspondant à l'implémentation VLSI. La description est une hiérarchie de modules. Un module est une entité qui décrit le comportement d'une partie délimitée du circuit. La description du circuit représente le fonctionnement d'un automate dont chaque état est identifié par une étiquette suivie d'une séquence. Cette dernière peut elle-même représenter un automate comportant des sous-états. Chaque état s'écrit sous la forme :

< etiquette > : < inst parallèle >

Dans un module, on peut avoir une liste de spécifications de séquences imbriquées. Chaque spécification de séquence définit son nom, le nombre de ses états, la liste de ses identificateurs (préfixés par \$) et la règle qui concerne leur évolution (exprimée par GENERAL ou RIGID).

Exemple de définition de séquencement :

Cet exemple définit une machine à deux niveaux de séquencement :

```

séquence machine<3> : $m1, $m2, $m3; général
    sequence tempo<3> : $t1, $t2, $t3; général
    endseq
endseq

```

le séquençement "tempo" est de niveau inférieur au séquençement "machine".

Un état est identifié par une ou plusieurs étiquettes appartenant à la déclaration du séquençement, chacune de ces étiquettes ayant été définie à un niveau de séquençement différent.

Un état est repéré par au moins une étiquette qui indique le niveau de séquençement dans lequel il agit. Les autres étiquettes placées avant celle-ci appartiennent à des niveaux de séquençement supérieurs.

Exemple d'écriture des états :

En considérant la définition de séquençement donnée plus haut, on peut avoir:

\$m1 : \$t1 : instructions parallèles ;
 \$t2 : instructions parallèles ;

\$m2 : \$t1 : instructions parallèles ;
 \$t2 : instructions parallèles ;
 \$t3 : instructions parallèles ;

De manière générale, chaque niveau de séquençement est spécifié à partir des états du niveau inférieur. Cette cascade d'imbrications fait que l'état du niveau le plus bas sert d'unité de temps au niveau qui lui est immédiatement supérieur.

Avec le langage IRENE, c'est le concepteur qui détermine l'architecture de la partie contrôle (le nombre d'étages) de son circuit.

Un programme décrivant le circuit à réaliser, écrit dans le langage IRENE, constitue la source du compilateur CIRENE [MART85]. Ce dernier a pour rôle d'éclater ce programme en plusieurs, chacun décrivant un niveau d'interprétation du circuit. Ces programmes sont de deux types :

Un programme représentant la description de la partie opérative (POP), il est destiné au compilateur APOLLON [JAM86b] qui se charge de générer le dessin des masques associé à la POP.

Un ou plusieurs programmes (selon le nombre de niveaux d'interprétation déclarés par le concepteur dans sa description) représentant les descriptions des différents étages de contrôle associés aux différents niveaux d'interprétation. Chacun de ces programmes est

généré dans un fichier et constitue la source du compilateur de partie contrôle (CPC). Ce dernier prend ces programmes, un à un, pour générer la description logique de l'étage correspondant en tenant compte de son organisation. Ces résultats sont fournis au générateur topologique qui les transforme en dessin des masques.

SYCO (version IRENE) est un prototype qui a présenté certains avantages dont:

- la démarche suivie a permis la mise au point des différents outils constituant SYCO indépendamment les uns des autres,
- la communication entre les différents outils se fait via des fichiers sous forme textuelle, ce qui offre la possibilité de produire manuellement des exemples d'entrée...

Mais cette version a aussi présenté quelques inconvénients qui se résument en:

- le langage IRENE lui-même présente une ambiguïté au niveau du branchement. En effet, à un niveau d'interprétation quelconque on peut faire un branchement à n'importe quel état de n'importe quel niveau d'interprétation. Ceci pose un problème au niveau de l'extraction des étages de contrôle,
- les fichiers de communication sont nombreux et peu lisibles, ce qui rend leur traitement assez délicat,
- les formes intermédiaires entre la description de départ et le dessin des masques ne sont pas simulables. Le fait de pouvoir simuler les formes intermédiaires permet de vérifier, lors du développement des outils, que les transformations appliquées par chacun ne changent pas la sémantique du circuit, ce qui permet de valider les outils,
- les différents outils constituant SYCO reçoivent des textes de syntaxes différentes, ce qui nécessite différents analyseurs lexicaux et syntaxiques.

C'est pourquoi cette première version a été abandonnée. Cependant cette expérience autour d'IRENE n'était pas sans intérêt car elle a permis de fixer un certain nombre de solutions et surtout de valider les principaux choix sur lesquels est basé le compilateur. De plus, le compilateur a été opérationnel et a permis à trois stagiaires de l'ENSERG de décrire le microprocesseur 68000 de Motorola et d'obtenir une partie du dessin des masques associé [RIN85].

V- La version autour du langage LDS (version actuelle)**V-1-Le langage d'entrée de SYCO**

LDS [LAU85] est un langage qui permet la description structurelle et comportementale de circuits séquentiels.

V-1-1- La description structurelle

La description structurelle est faite par une entité appelée Smodule. Un Smodule est composé de trois parties :

a- l'en-tête:

Cette partie permet de donner le nom du circuit et de définir l'interface du circuit avec l'environnement externe (bus de données, bus d'adresse, signaux d'interruption...).

b- la partie déclarative:

La partie déclarative est divisée en plusieurs blocs qui permettent de spécifier, en détail, les ressources constituant le circuit.

Le bloc SIGNAL permet de décrire l'interface du circuit défini en en-tête. Pour chaque signal, le concepteur détermine sa taille (pour les bus), son type (contrôle ou donnée), et son sens (entrée, sortie ou sortie/entrée).

Le bloc VARIABLE permet de déclarer des variables de contrôle qui sont inaccessibles de l'extérieur du circuit. Ces variables servent généralement pour paramètres aux procédures de la description comportementale.

Le bloc REGISTRE permet de déclarer tous les registres qui seront implémentés dans la partie opérative du circuit. Le concepteur détermine la taille de chaque registre.

c- Le lien:

Le lien permet de préciser la procédure principale de l'algorithme décrivant la fonction que le circuit doit réaliser (primitive LINK).

V-1-2- Définition d'opérateurs complexes et d'environnement de simulation

La définition d'opérateurs complexes se fait à l'aide des Fmodules alors que l'environnement de simulation est décrit par un Xmodule.

Les Fmodules sont des modules de description fonctionnelle banalisée.

Ils se situent au même niveau que les équations et ont la sémantique d'un opérateur complexe.

Exemple

```
Fmodule alustd ( , reg1, reg2, reg3, oper) ;
SIGNAL ;
    reg1 0:8, OUT;
    reg2 0:8, IN;
    reg3 0:8, IN;
END ;
VARIABLE ;
    oper INT, IN;
END ;
CASE (oper)
    WHEN ( 1 ) reg1 := reg2 + reg3;
    WHEN ( 2 ) reg1 := reg2 - reg3;
    WHEN ( 3 ) reg1 := reg2 * reg3;
    WHEN ( 4 ) reg1 := reg2 / reg3;
END ;
END ;
```

L'appel d'un Fmodule se fait par l'instruction CALL (CALL alustd(, A,B,C,2) pour faire la soustraction).

Les Xmodules servent à réaliser des moniteurs de simulation. L'utilisateur peut y spécifier l'environnement et les conditions dans lesquels la simulation sera effectuée, i. e. :

- Contrôle de la simulation : activation du Smodule racine, arrêt.
- Accès aux variables de tous les modules de la hiérarchie.
- Forçage de vecteurs d'entrée et visualisation de l'état des variables.
- Demande de trace de modules.

Un Xmodule possède une structure permettant de décrire un automate de simulation sous forme d'un enchaînement de séquences.

V-1-3- La description comportementale

Elle définit la fonction réalisée par le circuit. Elle est composée d'un ensemble d'entités appelées Cmodule. Un Cmodule est analogue à une procédure du langage PASCAL. Une description comporte alors un Cmodule principal (défini par la primitive LINK du Smodule) et une

hiérarchie d'appel de Cmodules secondaires. Chaque Cmodule peut faire appel à un autre (excepté le Cmodule principal), cependant le langage LDS ne permet pas la récursivité alors qu'il offre au concepteur la possibilité d'exprimer le parallélisme de son circuit. Un Cmodule décrit un automate d'états finis. Les instructions sont interprétées séquentiellement, chacune correspondant à un état de l'automate. Un ensemble d'instructions entre parenthèses correspond aussi à un état de l'automate et permet d'exprimer le parallélisme du circuit.

Exemple

```
Cmodule m1n1 ;
  < état 1 >  NEXT état 5 ;
  < état 2 >  A:=B;
  ( RESET X; SET Y; EXECUTE Cmod2;)
  ( SET X; RESET Y; EXECUTE Cmod3;)
  < état 5 > ( A := 1 ; NEXT état 2 ; )
END;
```

L'étiquette de l'état n'est nécessaire que si on fait un branchement explicite à cet état. Dans l'exemple, les états 3 et 4 n'ont pas été étiquetés. Les variables de contrôles déclarées dans le Smodule servent de paramètres d'appel aux Cmodules.

Le flux de contrôle est exprimé en utilisant trois types d'instructions:

- If Condition Action1 else Action2 endif :

Action1 et Action2 peuvent être un ensemble d'instructions conditionnelles ou inconditionnelles qui seront exécutées en parallèle. Action2 peut être vide c'est à dire qu'il n'y a pas de partie ELSE.

- CASE (selecteur)

```
When Val1 Action1 ;
When Val2 Action2 ;
. . . .
When valn Actionn ;
END;
```

Cette instruction permet d'exécuter une action en fonction de la valeur d'un selecteur.

- **NEXT état** : Cette instruction permet d'exprimer un branchement explicite à un état de l'automate. Pour un état final, la fin du cycle de l'automate s'exprime par l'instruction EXIT.

Le flux de données est exprimé par trois instructions :

- l'affectation ($A := B$) pour le transfert de registre,
- EXECUTE pour l'appel de Cmodule,
- CALL pour l'appel d'un opérateur complexe qui a été défini au préalable par l'entité Fmodule.

Les variables de contrôle sont affectées par les instructions SET (mise à 1) et RESET (mise à 0). Si la variable est structurée, l'opérateur d'affectation ($:=$) permettra de l'affecter.

La figure 2.3 donne un exemple de description qui correspond à un simple microprocesseur dont l'algorithme a été défini par F. ANCEAU et qui a été modifié par R. JAMIER. Le langage d'assemblage de ce microprocesseur est composé de cinq instructions (sta, ada, lda, bru, brug). L'interprétation de chaque instruction a été définie par un Cmodule indépendant.

Les lignes 1 et 2 définissent le nom du Smodule et l'interface du circuit avec le monde externe. La ligne 4 correspond à la déclaration d'une variable de contrôle interne de deux bits. Le bloc SIGNAL' (lignes 7 à 15) décrit avec précision l'interface du circuit. Pour chaque signal ou groupe de signaux on définit le sens dans lequel les informations sont véhiculées sur le signal (IN, OUT ou INOUT) et le type du signal (contrôle ou donnée). Par défaut, un signal est considéré comme un signal de donnée (lignes 8 et 9).

Le bloc REGISTRE (lignes 16 à 21) permet de déclarer les registres qui seront utilisés par la partie opérative du circuit. La ligne 22 détermine le Cmodule principal de la description comportementale. On remarque que ce Cmodule porte ici le même nom que le Smodule. Le reste de la description (lignes 25 à 73) correspond aux différents Cmodules décrivant la fonction remplie par le circuit. Le schéma d'appel hiérarchique correspondant à cette description est donnée par la figure 2.4. Cet exemple nous servira de support pour illustrer les différentes étapes de la compilation.

```

1 SMODULE mini ( adbus , dtbus , dtack , restart ,
2     adstrobe , dstrobe , read_write ) ; ? I/O SIGNALs
3 VARIABLE ;
4 r 0:2 , CTRL ; ? Control internal carrier , used as a
5     ? parameter for The fetch procedure
6 END ;
7 SIGNAL ; ? detail of the I/O signals
8 adbus 0:8 , OUT ; ?address bus
9 dtbus 0:8 , INOUT ; ?data bus
10 dtack , IN , CTRL ; ?data acknowledgement
11 restart , IN , CTRL ; Logical reset
12 adstrobe , OUT , CTRL ; ?address strobe
13 dstrobe , OUT , CTRL ; ?data strobe
14 read_write , OUT , CTRL ; ?selection read=1 / write=0
15 END ;
16 REGISTER ; ? data internal carrier
17 a 0:8 ; ?accumulator
18 b 0:8 ; ?special purpose register
19 ir 0:8 ; ?instruction register
20 pc 0:8 ; ?program counter
21 END ;
22 LINK mini ; ? A link to the main CMODULE (program)
23     ? of the behavioural description
24 END ;
25 CMODULE fetch ;
26 <send> ( adbus := pc ; read_write := 1 ; )
27 <> ( adbus := pc ; adstrobe := 1 ; NEXT receive ; )
28 <receive> IF ( dtack = 0 ) adbus := pc ; NEXT receive ;
29     ELSE
30         CASE ( r )
31             WHEN ( 0 ) ir := dtbus ;
32             WHEN ( 1 ) a := dtbus ;
33             WHEN ( 2 ) b := dtbus ;
34         END ;
35         pc := pc + 1 ; adstrobe := 0 ;
36     END ;
37 END ;
38 CMODULE sta ;
39 <> ( r := 2 ; EXECUTE fetch ; ) ? read (b) the adress
40 ( adbus := b ; read_write := 0 ; dtbus := a ; )
41 ( adbus := b ; adstrobe := 1 ; dtbus := a ; dstrobe := 1 ; )
42 <ack> IF ( dtack = 0 ) adbus := b ; dtbus := a ; NEXT ack ;
43     ELSE adstrobe := 0 ; dstrobe := 0 ;
44     END ;
45 END ;
46 CMODULE ada ;
47 <> ( r := 2 ; EXECUTE fetch ; ) ? read (b)
48     a := a + b ;
49 END ;
50 CMODULE kda ;
51 <> ( r := 1 ; EXECUTE fetch ; ) ?read (a)
52 END ;
53 CMODULE bru ;
54 <> ( r := 2 ; EXECUTE fetch ; ) ? read (b) the adress
55     pc := b ;
56 END ;
57 CMODULE bneg ;
58 <> ( r := 2 ; EXECUTE fetch ; ) ? read (b) the adress
59     IF ( a 7:1 = '1' ) pc := b ; END ;
60 END ;
61 CMODULE mnti ;
62 <rest> IF ( restart = 0 ) NEXT rest ; ELSE NEXT start ; END ;
63 <start> pc := 0 ;
64 <newinstr>( r := 0 ; EXECUTE fetch ; )
65 <decode> ( CASE ( ir 0:4 )
66     WHEN ( '0000' ) EXECUTE ada ; ? a <- a + dtbus
67     WHEN ( '0001' ) EXECUTE kda ; ? a <- dtbus
68     WHEN ( '0010' ) EXECUTE sta ; ? memory <- a
69     WHEN ( '0101' ) EXECUTE bneg ; ? jump if a < 0
70     WHEN ( '0110' ) EXECUTE bru ; ? unconditional jump
71     END ;
72     IF ( restart = 0 ) NEXT rest ; ELSE NEXT newinstr ; END ; )
73 END ;

```

Figure 2.3: Description d'un microprocesseur simple

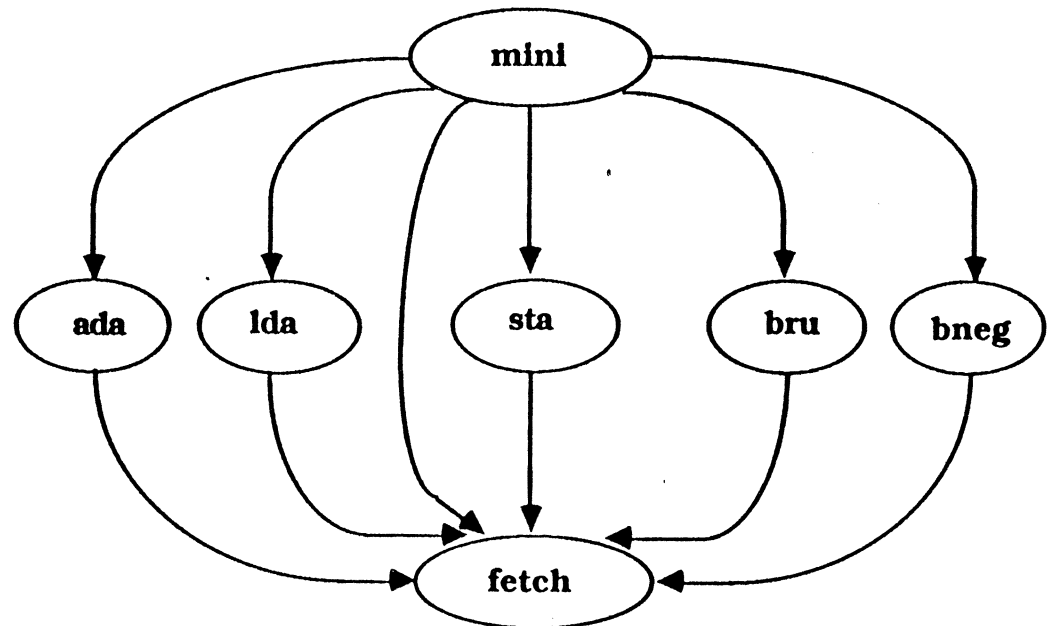


Figure 2.4: Schéma d'appel hiérarchique

V-2-Organisation de SYCO

SYCO est composé d'un ensemble d'outils qui permettent de générer le dessin des masques d'un circuit à partir de sa description de départ. Cette organisation est illustrée par la figure 2.5.

- Pré-compilateur: à partir de la description de départ, le pré-compilateur génère une structure de données LDS sur laquelle travailleront les autres outils. A chaque module déclaré par le concepteur correspond une entité dans la structure de données interne.

- EXTRACT : A partir de la hiérarchie d'appel définie dans la description de départ, cet outil extrait les différents étages de contrôle du circuit. Tous les Cmodules appartenant au même niveau hiérarchique sont regroupés dans un même Cmodule avec différents points d'entrée. Le nombre d'étages de contrôle va donc dépendre de la hiérarchie définie par le concepteur lors de la description de son circuit.

Pour être conforme au modèle architectural, un Cmodule généré par EXTRACT ne pourra faire appel qu'au Cmodule du niveau immédiatement inférieur. Pour assurer cette liaison, EXTRACT est amené à rajouter des points d'entrée intermédiaires dans les différents Cmodules générés.

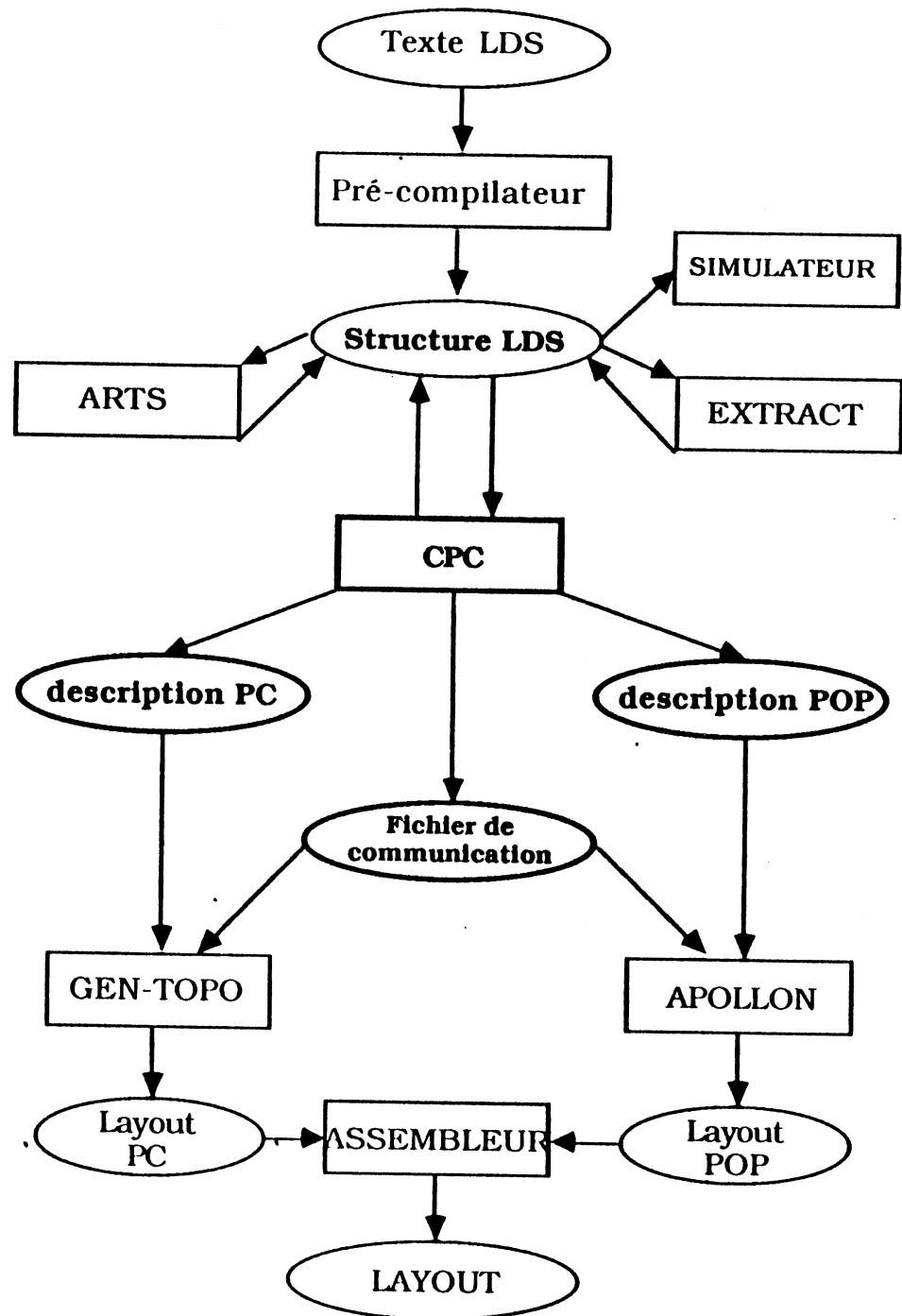


Figure 2.5: Organisation de SYCO

Dans la figure 2.4 on voit que le Cmodule principal fait appel au Cmodule "fetch" qui est au niveau le plus bas de la hiérarchie. EXTRACT rajoute dans le Cmodule du milieu (mini_1) un point d'entrée (mini_newinstr_1) pour assurer la transmission de l'appel. La figure 2.6 donne la hiérarchie d'appel obtenue par EXTRACT à partir de la hiérarchie donnée par la figure 2.4 alors que la figure 2.7 donne les Cmodules générés par EXTRACT.

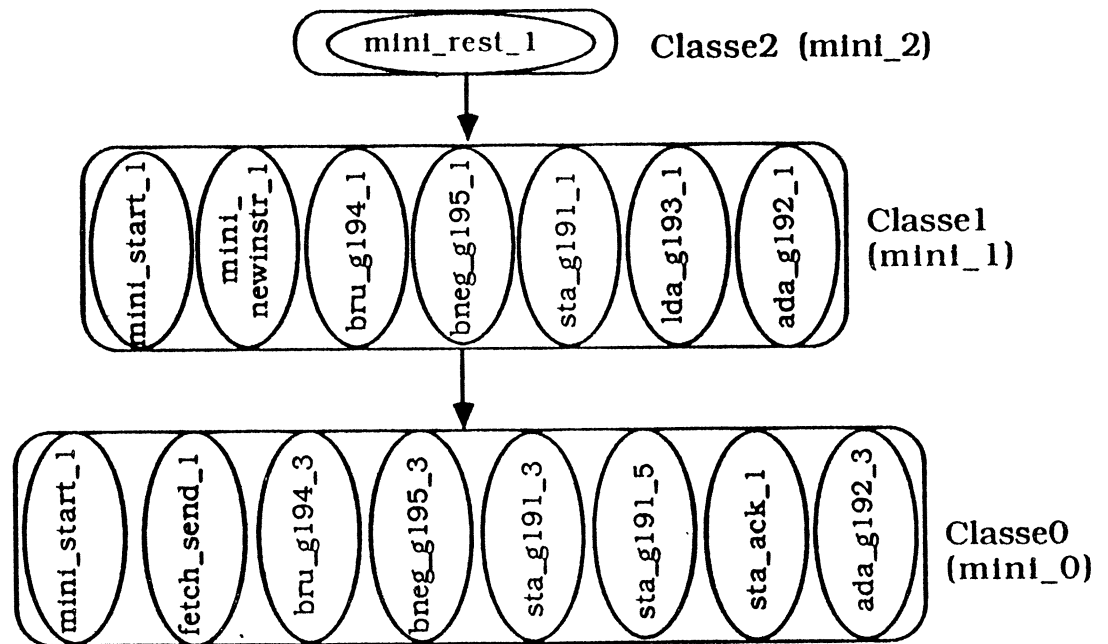


Figure 2.6: Différentes classes de Cmodules extraites par EXTRACT

Remarques :

- 1) Les instructions CASE sont remplacées par des instructions If en parallèle. La sémantique du circuit ne change pas.
- 2) Tous les états sont étiquetés.
- 3) L'action d'une instruction If est composée d'une seule instruction.
- 4) L'instruction If n'admet pas de ELSE.
- 5) Seul le Cmodule du plus bas niveau d'interprétation contient des instructions opératives (transfert de registres).

- CPC [MHA87, MHA88] est le compilateur de partie contrôle. À partir de la structure générée par EXTRACT, il génère:

- la table d'état de chaque étage de contrôle qui n'est autre que la table de transition de l'ensemble d'automates implémentés par l'étage de contrôle,
- la description logique de chaque étage de contrôle (Cmodule),
- la description de la partie opérative,
- un fichier de communication qui permettra l'assemblage des étages de contrôle et de la partie opérative.

Cet outil fait l'objet essentiel de ce travail et sera présenté plus en détail dans le chapitre 4.

```
? ----- Module mint_2
CMODULE mint_2 ;
< mint_rest_1 >
( IF (restart 0:1 = (0)) NEXT mint_rest_1 ; ENDIF ;
  IF ( # (restart 0:1 = (0))) NEXT mint_start_1 ; ENDIF ; )
< mint_start_1 >
(EXECUTE mint_1 (mint_start_1) ; )
< mint_newinstr_1 >
(r 0:2 := 0 ; EXECUTE mint_1 (mint_newinstr_1) ; )
< mint_decode_1 >
( IF (ir 0:4 = (BIN'0110')) EXECUTE mint_1 (bru_g194_1) ; ENDIF ;
  IF (ir 0:4 = (BIN'0101')) EXECUTE mint_1 (bneg_g195_1) ; ENDIF ;
  IF (ir 0:4 = (BIN'0010')) EXECUTE mint_1 (sta_g191_1) ; ENDIF ;
  IF (ir 0:4 = (BIN'0001')) EXECUTE mint_1 (lda_g193_1) ; ENDIF ;
  IF (ir 0:4 = (BIN'0000')) EXECUTE mint_1 (ada_g192_1) ; ENDIF ;
  IF (restart 0:1 = (0)) NEXT mint_rest_1 ; ENDIF ;
  IF ( # (restart 0:1 = (0))) NEXT mint_newinstr_1 ; ENDIF ; )
END ; ? ----- End of Module mint_2

? ----- Module mint_1
CMODULE mint_1 ;
< mint_start_1 >
(EXECUTE mint_0 (mint_start_1) ; )
< mint_newinstr_1 >
(EXECUTE mint_0 (fetch_send_1) ; )
< bru_g194_1 >
(r 0:2 := 2 ; EXECUTE mint_0 (fetch_send_1) ; )
< bru_g194_3 >
(EXECUTE mint_0 (bru_g194_3) ; )
< bneg_g195_1 >
(r 0:2 := 2 ; EXECUTE mint_0 (fetch_send_1) ; )
< bneg_g195_3 >
(EXECUTE mint_0 (bneg_g195_3) ; )
< sta_g191_1 >
(r 0:2 := 2 ; EXECUTE mint_0 (fetch_send_1) ; )
< sta_g191_3 >
(read_write 0:1 := 0 ; EXECUTE mint_0 (sta_g191_3) ; )
< sta_g191_5 >
(adstrobe 0:1 := 1 ; dstrobe 0:1 := 1 ; EXECUTE mint_0 (sta_g191_5) ; )
< sta_ack_1 >
( IF ( # (dtack 0:1 = (0))) adstrobe 0:1 := 0 ; ENDIF ;
  IF ( # (dtack 0:1 = (0))) dstrobe 0:1 := 0 ; ENDIF ;
  IF (dtack 0:1 = (0)) NEXT sta_ack_1 ; ENDIF ;
  EXECUTE mint_0 (sta_ack_1) ; )
< lda_g193_1 >
(r 0:2 := 1 ; EXECUTE mint_0 (fetch_send_1) ; )
< ada_g192_1 >
(r 0:2 := 2 ; EXECUTE mint_0 (fetch_send_1) ; )
< ada_g192_3 >
(EXECUTE mint_0 (ada_g192_3) ; )
END ; ? ----- End of Module mint_1

? ----- Module mint_0
CMODULE mint_0 ;
< mint_start_1 >
(pc 0:8 := 0 ; )
< bru_g194_3 >
(pc 0:8 := b 0:8 ; )
< bneg_g195_3 >
( IF (a 7:1 = (BIN'1')) pc 0:8 := b 0:8 ; ENDIF ; )
< sta_g191_3 >
(adbus 0:8 := b 0:8 ; dtbus 0:8 := a 0:8 ; )
< sta_g191_5 >
(adbus 0:8 := b 0:8 ; dtbus 0:8 := a 0:8 ; )
< sta_ack_1 >
( IF (dtack 0:1 = (0)) adbus 0:8 := b 0:8 ; ENDIF ;
  IF (dtack 0:1 = (0)) dtbus 0:8 := a 0:8 ; ENDIF ; )
< ada_g192_3 >
(a 0:8 := a 0:8 + (b 0:8) ; )
< fetch_send_1 >
(adbus 0:8 := pc 0:8 ; read_write 0:1 := 1 ; )
< fetch_g190_1 >
(adbus 0:8 := pc 0:8 ; adstrobe 0:1 := 1 ; NEXT fetch_receive_1 ; )
< fetch_receive_1 >
( IF (dtack 0:1 = (0)) adbus 0:8 := pc 0:8 ; ENDIF ;
  IF (dtack 0:1 = (0)) NEXT fetch_receive_1 ; ENDIF ;
  IF (r 0:2 = (2)) . ( # (dtack 0:1 = (0))) b 0:8 := dtbus 0:8 ; ENDIF ;
  IF (r 0:2 = (1)) . ( # (dtack 0:1 = (0))) a 0:8 := dtbus 0:8 ; ENDIF ;
  IF (r 0:2 = (0)) . ( # (dtack 0:1 = (0))) ir 0:8 := dtbus 0:8 ; ENDIF ;
  IF ( # (dtack 0:1 = (0))) pc 0:8 := pc 0:8 + (1) ; ENDIF ;
  IF ( # (dtack 0:1 = (0))) adstrobe 0:1 := 0 ; ENDIF ; )
END ; ? ----- End of Module mint_0
```

Figure 2.7: Cmodules générés par EXTRACT

- ARTS [BEK87] est un outil d'évaluation et d'optimisation topologique du circuit décrit. Généralement le circuit obtenu à partir de la description de départ a une forme pyramidale. Le CPC fait appel à cet outil pour mieux équilibrer le circuit et par conséquent améliorer le compromis surface/vitesse. Etant donné que l'exécution de chaque instruction doit traverser tous les étages de contrôle, le temps d'exécution va dépendre du nombre d'étages de contrôle. Quant à la surface du circuit, elle dépend de la largeur de la partie opérative (donc du degré de parallélisme spécifié par le concepteur) et de la hauteur de la partie contrôle (du nombre d'étages du contrôle).

Les principales actions réalisées par ARTS sont :

1- déplacement d'un automate d'un étage vers l'étage supérieur ou inférieur : quand la partie contrôle n'est pas très régulière, ce mouvement peut équilibrer les différents étages de contrôle,

2- fusion de deux étages de contrôle voisins : comme la vitesse d'exécution du circuit dépend du nombre d'étages de contrôle, la fusion de deux étages de contrôle voisins permet de réduire leur nombre et par conséquent augmenter la rapidité du circuit,

3- éclatement d'un étage de contrôle en deux : ce traitement est appliqué quand la largeur d'un étage de contrôle dépasse celle de la partie opérative. Le fait de l'éclater en deux permet d'augmenter la régularité du circuit au détriment de la vitesse d'exécution du circuit.

- APOLLON [JAM86b] génère des parties opératives (POP) selon un modèle architectural bien défini. Ce modèle est inspiré de celui du microprocesseur MC 68000. Chaque élément de la POP peut être connecté à deux bus qui traversent la partie opérative. Ces deux bus peuvent être segmentés pour obtenir des sous-parties opératives pouvant fonctionner en parallèle (voir figure 2.8). Une sous-partie opérative (figure 2.9) est composée:

- de registres à simple ou double accès, certains registres pouvant être accessibles à la partie contrôle (compte rendu de la POP, registre d'instructions),
- de constantes à simple accès.

- de buffers d'entrée/sortie,
- d'opérateurs.

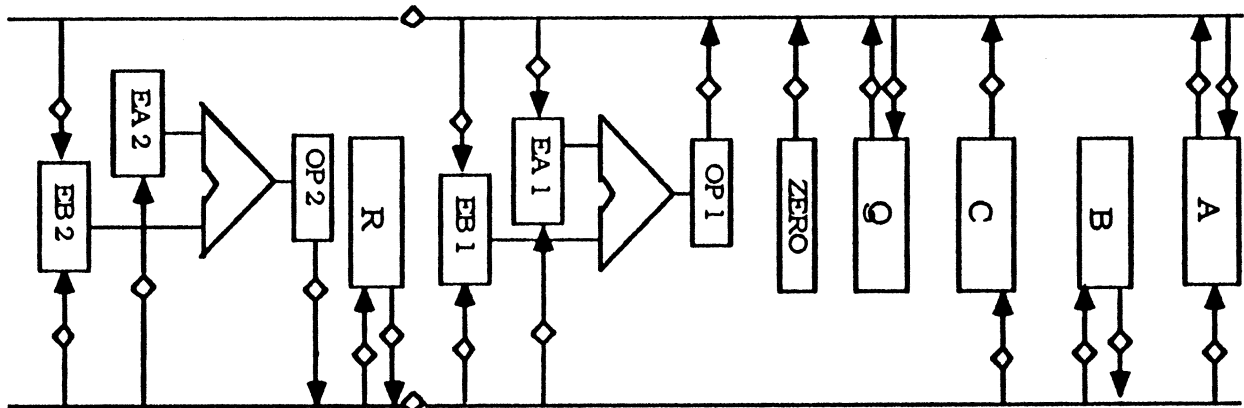


Figure 2.8: Modèle architectural de la POP

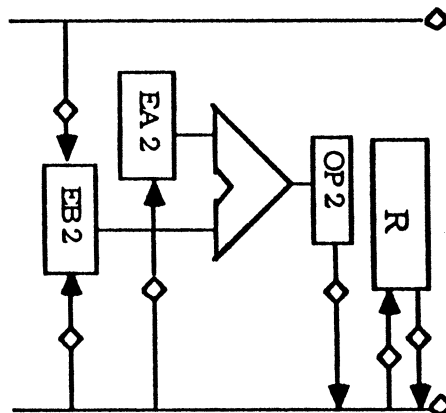


Figure 2.9: Schéma d'une sous-POP

A partir de la description comportementale de la POP, générée par le CPC, APOLLON génère une description de blocs fonctionnels à partir de laquelle le dessin des masques est généré sous forme de pile de bit slices. APOLLON utilise une librairie de cellules standard NMOS. La description structurelle de la partie opérative est obtenue en trois étapes :

1) Placement des registres:

La tâche principale d'APOLLON est le partitionnement des registres en sous-ensembles disjoints correspondant aux sous parties opératives. Ces ressources sont définies explicitement par le concepteur dans le Smodule. On verra plus loin que la description comportementale de la partie

opérative, générée par le CPC, spécifie tous les registres utilisés effectivement dans cette description. Cette redondance permet, le cas échéant, d'éviter d'implémenter un registre déclaré par le concepteur dans le Smodule mais qui n'a jamais été utilisé dans la description comportementale.

Tous les registres implémentés dans la POP sont lus avant d'être écrits ce qui permet d'interchanger deux registres appartenant à la même sous partie opérative, en un seul cycle ($A := B ; B := A$).

2) Allocation des opérateurs:

La liste des opérateurs utilisés dans la description comportementale est fournie par le CPC dans la partie déclarative de la description de la partie opérative.

Différentes occurrences d'un même opérateur peuvent aboutir à une implémentation unique de cet opérateur ou à autant d'implémentations que d'occurrences. Chaque opérateur peut être réalisé en utilisant des composants spécialisés ou des composants universels (ALUs). Pour chaque sous partie opérative, APOLLON génère la liste d'opérateurs qui appartiendront à cette sous-POP. Lors de la génération du layout, chaque liste est transformée en une unité d'exécution dépendant de la librairie de primitives matérielles disponible.

3) Allocation d'interconnexion d'unités:

Ce traitement consiste à déterminer le nombre de segments des deux bus autour desquels la partie opérative est construite. Il tient compte du fait qu'un registre peut être connecté à un ou deux bus de la sous-POP à laquelle il appartient. Pour éviter l'utilisation de multiplexeurs, tout en permettant la communication entre les différentes sous-parties opératives, APOLLON peut interconnecter deux segments d'un même bus.

Une étape APOLLON s'exécute au maximum en deux cycles. Chaque cycle est composé de 4 phases d'horloge, T1, T2, T3, et T4 non recouvrantes.

- GEN-TOPO [VAR86, GER87] permet la génération de PLA à partir de la description logique et du fichier de communication générés par le CPC. Les PLAs générés sont des PLAs classiques organisés en structure

NOR-NOR en technologie CMOS à précharge (figure 2.10).

Les PLAs générés sont disposés horizontalement. Les bus de contrôle et de compte rendu sont à gauche des PLAs. Une fois tous les étages de contrôle et la partie opérative générés, on passe à l'assemblage.

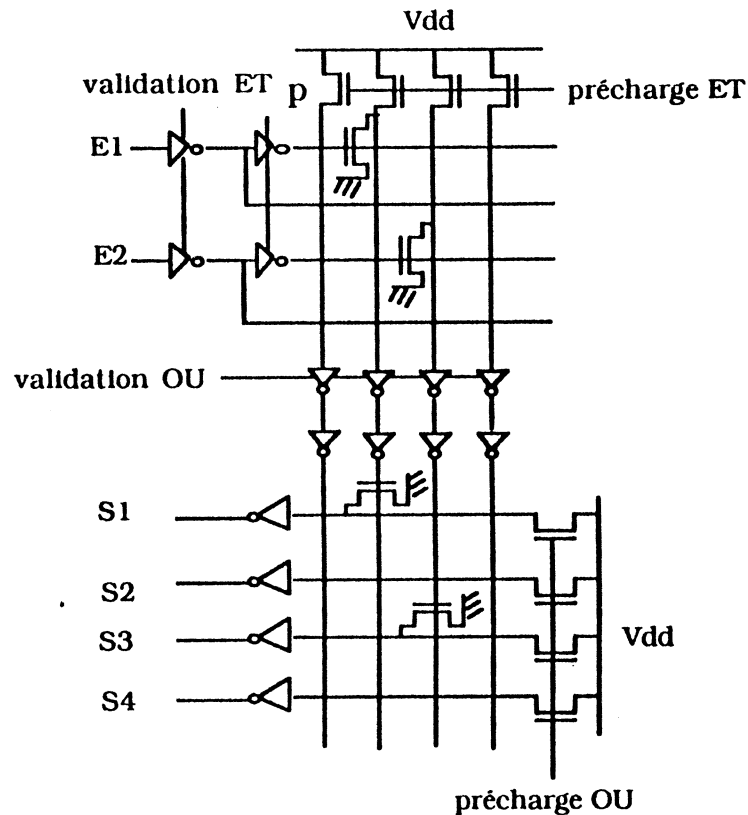


Figure 2.10: Schéma d'un PLA CMOS à précharge

Dans la version actuelle, la synchronisation des étages de contrôle est assurée par l'intermédiaire de deux variables de synchronisation (WAIT et WORK) à chaque étage [GER87].

Pour l'étage i , la variable $WAIT_i$ est à "1" lorsque l'étage en question attend des comptes rendus. Ceci correspond au cas où l'étage a envoyé une commande à l'étage inférieur et qu'il doit attendre la fin d'exécution du cycle de l'étage inférieur avant de pouvoir continuer l'exécution de son cycle.

La deuxième variable $WORK_i$ est à "1" lorsque l'étage de contrôle est actif (en cours d'exécution d'une instruction).

Un étage de contrôle i peut:

- ```
mettre à "0" (RESET) sa variable WORKi pour indiquer aux autres
étages qu'il est au repos et susceptible de recevoir une nouvelle
```



- commande. C'est le cas de fin d'exécution d'un cycle,
- # mettre à "1" (SET) la variable  $WAIT_i$  quand il doit attendre la fin d'exécution de la commande envoyée à l'étage inférieur,
  - # mettre à "1" la variable  $WORK_{i-1}$  de l'étage inférieur à l'issue d'une commande qu'il vient de lui envoyer. C'est un signal de réveil pour l'étage inférieur,
  - # mettre à "0" la variable  $WAIT_{i+1}$  de l'étage supérieur pour lui indiquer qu'il lui est possible de continuer l'exécution de son cycle et éventuellement d'envoyer une commande à l'étage i. C'est un signal d'invitation.

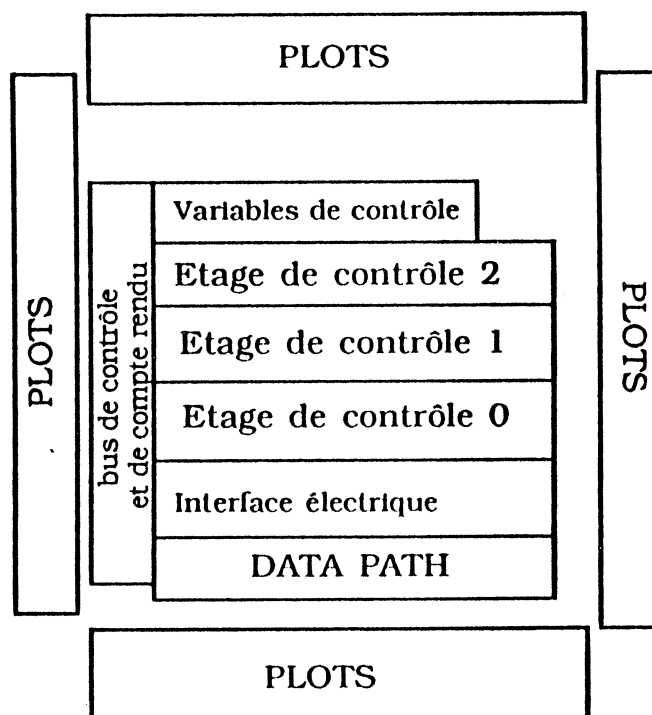


Figure 2.11: Plan de masse de circuits générés par SYCO

- ASSEMBLEUR [GER87]: Un programme d'assemblage prototype a été réalisé en Nautille [JER88]. Il permet d'évaluer les différentes cellules du circuit et de réaliser le bus de connexion entre les étages de contrôle. Ce programme reçoit la description logique des PLAs réalisant les différents étages de contrôle, et le fichier de communication générés par le CPC. La tâche de l'Assembleur consiste à:

- 1) Connecter les entrées/sorties des différents PLAs aux bus de contrôle, de compte rendu et de synchronisation.

2) Connecter le dernier étage de contrôle à la partie opérative. Un exemple de plan de masse est illustré par la figure 2.11. Le plan de masse est composé d'une pile de tranches. La tranche la plus basse correspond à la partie opérative. Les autres correspondent à la partie contrôle. Les variables de contrôle sont implémentées au dessus de la dernière tranche de contrôle alors que les bus de compte rendu et de contrôle passent à gauche de la pile de tranches. Les plots sont placés tout autour du circuit.

### **V-3-Schéma d'utilisation de SYCO**

L'algorithme de SYCO est donné par l'organigramme de la figure 2.12. Dans cet algorithme on remarquera qu'après l'application de ARTS à la description on repasse par le CPC car s'il y a eu une transformation de la structure des étages de contrôle, il faut évidemment recalculer la table d'états de chaque étage.

## **VI- Conclusion**

Le compilateur de partie contrôle CPC joue un rôle important dans le compilateur SYCO étant donné qu'il constitue une interface entre la description générée par EXTRACT et tous les autres outils (ARTS, APOLLON, GEN-TOPO). Le choix des interfaces de communication entre le CPC et les autres outils est donc important. Le CPC sera présenté dans le chapitre 4.

SYCO est un compilateur comportemental. Tous les outils le composant évoluent autour de la même structure de données. ARTS est un outil qui permet d'améliorer les performances du circuit en étant guidé par le concepteur qui précise certains choix. Avec SYCO, le concepteur a la possibilité de revenir en arrière pour modifier ses choix et influencer le résultat obtenu par ARTS. Par conséquent, SYCO peut être classé parmi les compilateurs intelligents.

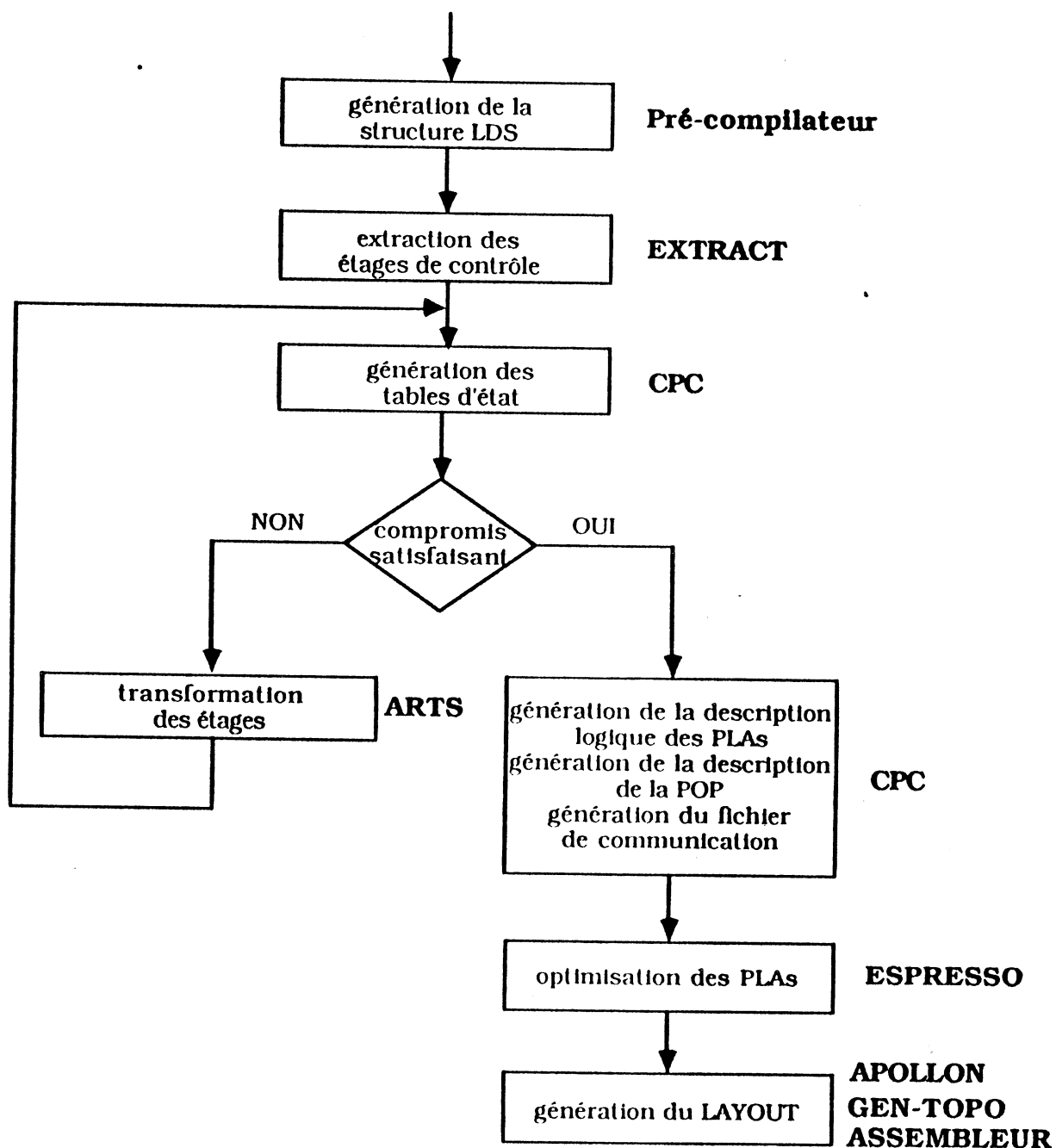


Figure 2.12: Algorithme général de SYCO.

# **CHAPITRE 3**

## **Les Compilateurs de Partie**

### **Contrôle**



## **I- Introduction**

Dans ce chapitre, on parlera surtout des compilateurs qui partent d'une description de haut niveau. La tâche principale de ces outils est de traduire la description de départ en une description logique des blocs physiques permettant de réaliser la fonction que doit remplir le circuit.

Le passage direct d'une description de haut niveau à une description de bas niveau (logique) s'avère généralement assez difficile. Les différents compilateurs de partie contrôle passent donc par une forme intermédiaire pour simplifier cette tâche.

Le processus de traduction se compose généralement de trois étapes:

- 1) la traduction de la description de départ en une forme intermédiaire,
- 2) l'application de règles d'optimisation logique et topologique à la forme intermédiaire,
- 3) la génération de la description logique des blocs physiques réalisant la fonction spécifiée par la description de départ.

Il est important de s'assurer que la sémantique de la fonction ne change pas lors des passages d'une étape à une autre.

Cependant, ce schéma n'est pas totalement figé et peut varier d'un compilateur à un autre. Comme exemple, on peut citer le compilateur YASC [KRE85, KRE86] qui transforme le graphe de flux de données, reçu en entrée, en un ensemble d'équations booléennes. Ensuite, ces équations sont optimisées. Le layout est directement généré à partir de ces équations booléennes optimisées. Cet exemple montre que le processus de compilation peut dépendre de la forme intermédiaire choisie par le compilateur. Pour YASC, la forme intermédiaire est une description logique, donc l'étape 3 ne doit pas exister.

Les deux principales stratégies d'implémentation d'une partie contrôle sont:

- 1) la logique aléatoire ,
- 2) les structures régulières.

La première est caractérisée par des réseaux complexes de portes et d'éléments de mémorisation et présente des inconvénients qui font qu'elle est peu adoptée.

La seconde consiste à implémenter la partie contrôle à l'aide de PLAs et de ROMs, qui sont des structures connues par leur régularité. Ces structures peuvent être programmées de sorte qu'elles réalisent un automate d'états finis.

La surface occupée par la partie contrôle d'un circuit croît avec la complexité de celui-ci [ANC86]. Par conséquent, il va falloir choisir la structure de la partie contrôle en tenant compte de la complexité du circuit à concevoir.

## II- Les différentes organisations de parties contrôle

Une partie contrôle représente une machine séquentielle qui reçoit en entrée le code opération de l'instruction à exécuter et les conditions d'exécution, et génère en sortie un ensemble de commandes (destinées généralement à la partie opérative) et met à jour les conditions. Cette machine représente l'implémentation de deux fonctions combinatoires:

- (1) la fonction de transition:

$F(\text{état courant, instruction, conditions}) \rightarrow \text{état suivant}$

- (2) la fonction de sortie

$G(\text{état courant, instruction, conditions}) \rightarrow (\text{commandes, conditions})$

La fonction G correspond au cas où l'automate réalisé est du type Mealy. Quand la machine réalise un automate du type Moore, la fonction de sortie G ne dépend que de l'état courant de l'automate, on aura:

$G(\text{état courant}) \rightarrow (\text{commandes, conditions})$

Donc une partie contrôle peut être divisée en deux parties fonctionnelles distinctes (figure 3.1):

- # le mécanisme de séquencement interne de la machine (il réalise la fonction de transition F);
- # le mécanisme de génération des commandes (il réalise la fonction de sortie G).

Le choix entre les deux modèles de Mealy et de Moore est critique. Le passage d'un modèle à l'autre peut influencer la complexité de la partie contrôle [ANC86]. En effet, la taille de la partie contrôle dépend énormément du modèle choisi.

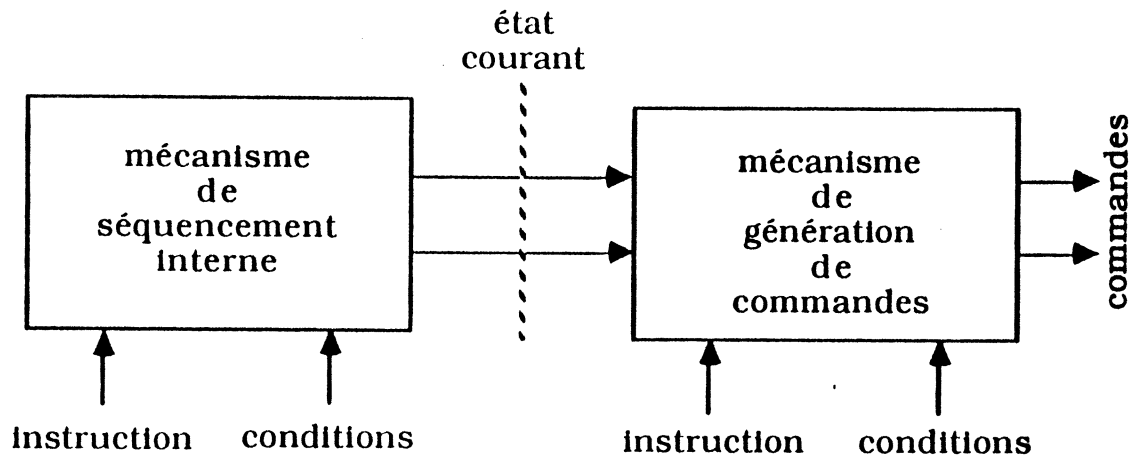


Figure 3.1: Schéma fonctionnel d'une partie contrôle [ANC86]

Pour le modèle de Moore, à chaque état de l'automate, seul le séquençage peut être conditionnel, toutes les autres actions étant liées à l'état et exécutées inconditionnellement, alors que pour le modèle de Mealy toutes les actions, y compris le séquençage, peuvent être conditionnelles ou inconditionnelles.

### II-1- Le séquençage conditionnel (modèle de Moore)

Avec le modèle de Moore, l'état suivant de l'automate est sélectionné en fonction des conditions de branchement (ou de transition). Ce mécanisme est généralement composé de deux parties:

(1) le mécanisme de sélection de la condition à tester, c'est la propriété,

(2) le mécanisme de séquençage lui-même qui détermine l'état suivant en fonction de la propriété et de l'état courant de l'automate.

Pour un automate du type Moore, le nombre d'états est maximum et le mécanisme de séquençage est complexe [ANC86].

### II-2- La génération de commandes conditionnelles (modèle de Mealy)

Ce mécanisme peut aussi être décomposé en deux parties analogues à celles du mécanisme de séquençage conditionnel.

Pour un automate de Mealy, le nombre d'états est réduit mais le mécanisme de génération de commandes est assez complexe et lent.

Les mécanismes de séquençage occupent plus de surface et sont moins performants que les mécanismes de génération de commandes.



L'efficacité d'une partie contrôle s'améliore avec la décroissance du nombre d'états de l'automate qu'elle implémente. Par conséquent, l'approche de Mealy s'avère plus intéressante que celle de Moore [ANC86].

### **II-3- Calcul de l'état suivant:**

Généralement, le calcul de l'état suivant de l'automate dépend de la façon de mémoriser l'état. On distingue deux façons:

#### **1) L'état est mémorisé par un registre:**

A chaque instant, le registre contient le numéro de l'état courant (ou la microadresse de la microinstruction à exécuter pour les ROMs). Le mécanisme de séquençement interne génère l'état suivant que ce soit pour un branchement qui rompt la séquence ou un branchement à l'état suivant (dans la séquence de l'automate). Le mécanisme de séquençement est obtenu à partir d'une description de l'automate dans laquelle tous les branchements sont spécifiés explicitement.

#### **2) L'état courant est mémorisé dans un compteur:**

Le mécanisme de séquençement ne calcule l'état suivant que dans le cas d'un branchement qui rompt la séquence de l'automate, quand le branchement se fait à l'état suivant (syntaxiquement), il est obtenu par incrémentation du compteur. Cette solution est efficace pour l'implémentation d'automates ayant de longues séquences et quand l'implémentation est assurée par une ROM où les microadresses se suivent. Pour le cas des PLAs, cette solution n'est possible que si les états successifs de l'automate ont des numéros successifs.

Les différentes organisations de parties contrôle diffèrent des techniques d'implémentation des deux fonctions (de transition et de sortie) et du modèle de l'automate réalisé. Dans la suite de ce chapitre, on donnera un aperçu des différentes structures de parties contrôle.

### **II-4- Parties contrôle câblées**

Elles sont obtenues par une traduction directe de l'algorithme d'interprétation en matériel. A chaque état de l'algorithme on fait correspondre une bascule D ou un registre maître-esclave d'un bit. L'architecture de la partie contrôle réalisée avec cette technique se caractérise par la présence d'un décodeur du code opération, le seul

élément régulier de la structure, et d'un réseau combinatoire anarchique occupant le reste de la surface de contrôle.

Quand l'algorithme d'interprétation est assez complexe, une partie contrôle de ce type devient assez encombrante, donc déconseillée. Ce type de PC permet l'implémentation d'automates de Moore.

### II-5- Parties contrôle microprogrammées

L'algorithme d'interprétation est traduit sous forme de microprogramme enregistré dans une ROM. A chaque état de l'algorithme correspond une ou plusieurs micro-instructions qui doivent délivrer les commandes et des informations décrivant l'adresse ou la façon d'obtenir l'adresse de la micro-instruction suivante.

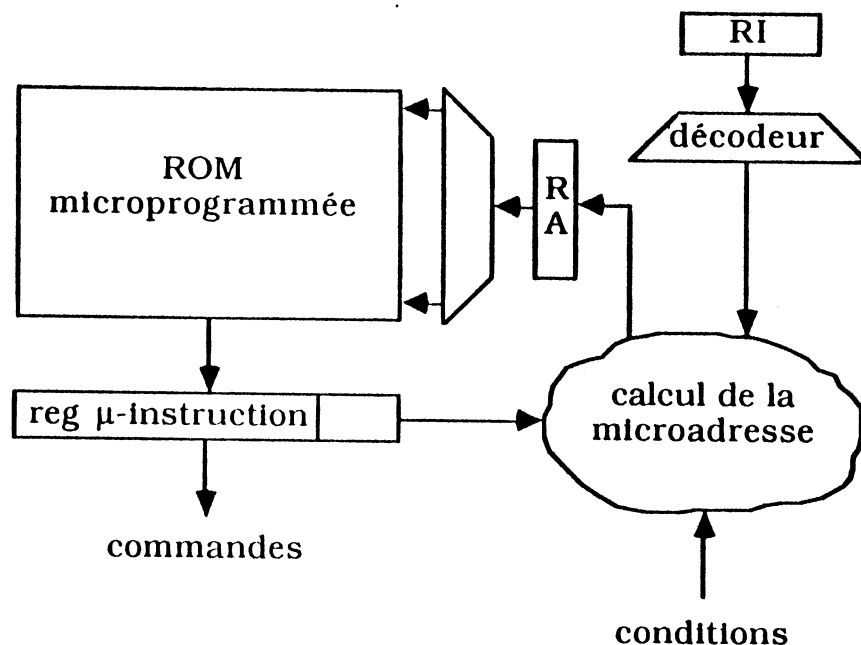


Figure 3.2: Schéma général d'une PC microprogrammée

Une PC microprogrammée comprend essentiellement (figure 3.2):

- (1) la ROM microprogrammée,
- (2) un micro-compteur ordinal contenant l'adresse du mot sélectionné de la ROM: c'est le registre d'adresse (RA),
- (3) le registre de micro-Instruction (RI),
- (4) et le circuit de calcul d'adresse de la micro-Instruction suivante.

#### II-5-1- La microprogrammation horizontale

Chaque état de l'algorithme d'interprétation est décrit par une seule

micro-instruction qui fournit toutes les informations concernant l'état. Généralement, la micro-instruction est composée de deux champs:

- # le champ de séquençement,
- # le champ de commande.

Pour pouvoir faire un branchement conditionnel, on rajoute un champ T à la micro-instruction. Dans ce cas, le champ de séquençement est composé de deux parties:

- (1) le champ adresse: il contient, dans le cas du branchement inconditionnel, l'adresse effective de la micro-instruction suivante,
- (2) le champ test: il indique que dans le cas du branchement conditionnel, le champ adresse contient non pas l'adresse effective mais une adresse de référence qui va être modifiée en fonction des valeurs des conditions conduisant ainsi aux différentes adresses de branchement.

L'adresse de référence pourrait être l'adresse de la micro-instruction courante qui permettrait effectivement le calcul de l'adresse de la micro-instruction suivante selon les conditions d'entrée.

Les micro-instructions définies de cette façon sont dites longues et la taille de la ROM de contrôle peut devenir importante.

#### II-5-2: La microprogrammation verticale:

Cette solution est un remède au problème de surface occupée par la ROM dans le cas des micro-instructions longues. En effet, l'intérêt de cette organisation est de minimiser la surface occupée par la ROM utilisant plusieurs techniques de réduction de la surface occupée par le champ de commande dans les micro-instructions longues. Une des techniques consiste à définir une partition de la micro-instruction en champs reliés aux blocs fonctionnels composant la partie opérative. Chaque champ permet de coder les commandes destinées à un bloc fonctionnel bien déterminé [OBR82].

#### **II-6- Parties contrôle utilisant des PLAs**

Le principe est de réaliser une PC à partir d'équations booléennes extraites de l'algorithme d'interprétation. Chaque commande est exprimée sous forme de somme de produits des variables d'entrées qui correspondent:

- 1) aux bits du code opération de l'instruction,

- 2) aux signaux de contrôle externes,
- 3) aux comptes rendus émanant de la partie opérative,
- 4) aux bits codant l'état de l'automate de contrôle.

Une PC peut être composée d'un ou de plusieurs PLAs. Ce type de PC est mieux adapté aux automates de Mealy.

#### II-6-1- PC à PLA unique

Un seul PLA réalise les deux mécanismes de génération de commandes et de séquençage interne (figure 3.3). Le PLA est obtenu par un codage direct des équations booléennes extraites de l'algorithme de contrôle de départ.

En entrée, la matrice ET du PLA décode le registre d'instruction, les conditions de transition (variables de contrôle et de compte rendu) et l'état courant de l'automate. La matrice OU génère l'état suivant de l'automate et les commandes.

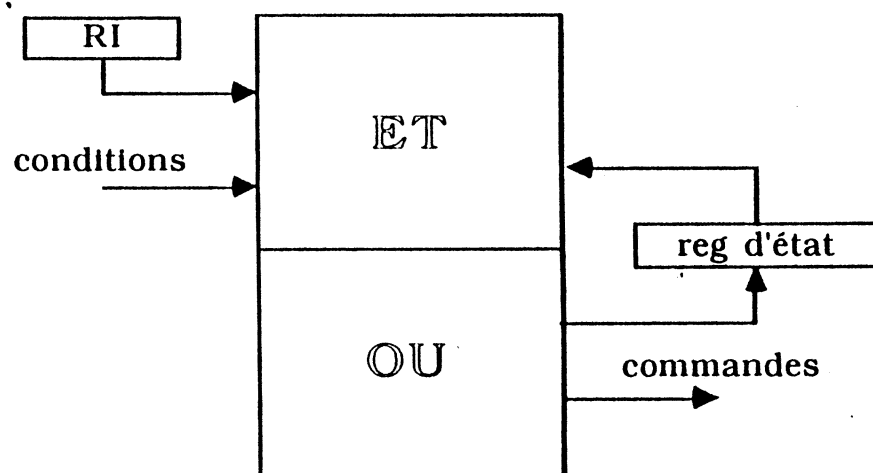


Figure 3.3: PC mono-PLA

#### II-6-2- PC à PLAs multiple

Quand l'algorithme d'interprétation du circuit à réaliser est assez complexe, une PC mono-PLA occupe une surface considérable et devient assez lente. Pour réduire la surface occupée par la partie contrôle, une des solutions consiste à découper le PLA en plusieurs. La solution la plus courante est de réaliser la PC par deux PLAs, chacun implémentant l'un des mécanismes de séquençage et de génération de commandes (figure 3.4):

- # le PLA de séquençement: la matrice ET décode le registre d'instruction, les variables de condition et l'état courant de la machine. La matrice OU génère l'état suivant de la machine,
- # le PLA de commande: la matrice ET décode le registre d'instruction, les variables de condition et l'état courant de la machine. La matrice OU génère les commandes destinées à la partie opérative.

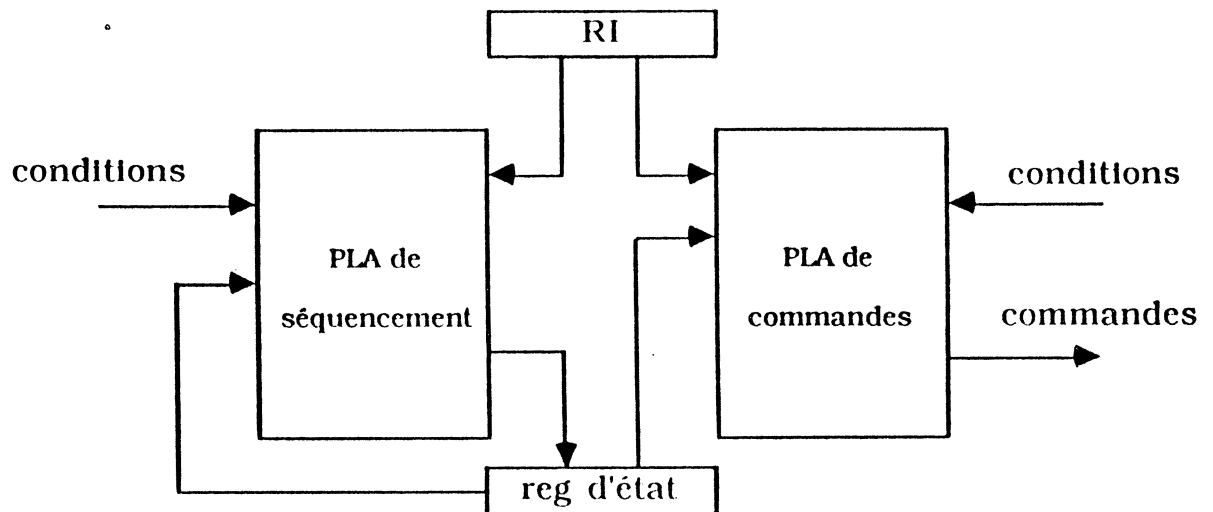


Figure 3.4: PC à deux PLAs

### II-7- Parties contrôle utilisant un générateur de temps

Pour ce type de PC, chaque commande est caractérisée par le ou les instants de son activation au cours de l'exécution d'une instruction de l'algorithme d'interprétation. Le déroulement d'une instruction peut donc être décrit par deux ensembles:

- 1) l'ensemble de toutes les commandes à activer pour cette instruction,
- 2) l'ensemble des instants d'activation de chaque commande.

Il en découle que trois dispositifs sont nécessaires pour réaliser une PC à générateur de temps (figure 3.5):

- a- un dispositif qui élabore les commandes à activer,
- b- un autre qui délivrera les instants d'activation des commandes: c'est le générateur de temps,
- c- et un troisième qui validera les commandes au bon moment.

Le générateur de temps est un automate qui assure la progression dans le temps de l'interprétation de l'instruction ainsi que l'identification de chaque instant d'exécution grâce aux signaux de séquençement qu'il

délivre. Il tient compte:

- du registre d'instruction,
- des signaux de contrôle,
- et de l'état de la machine.

Quant au mécanisme de validation des commandes, il peut être réalisé de façon câblée ou à l'aide d'un PLA [OBR82].

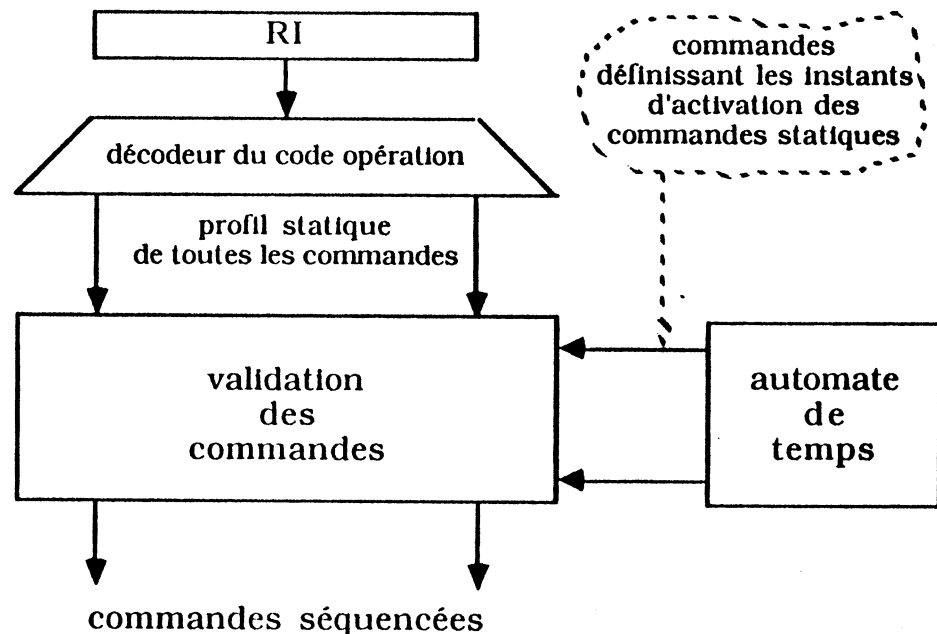


Figure 3.5: PC à générateur de temps

## II-8- Parties contrôle à plusieurs niveaux d'interprétation

Le principe consiste à empiler plusieurs étages de contrôle de sorte que les commandes d'un niveau servent d'instructions pour le niveau immédiatement inférieur. Chaque niveau (excepté le plus haut) doit indiquer au niveau immédiatement supérieur la fin d'interprétation de la commande en cours. Le niveau supérieur peut alors envoyer une autre commande au niveau inférieur. Chaque étage de contrôle peut être réalisé par n'importe quelle autre structure présentée plus haut.

Ce modèle de PC est adopté par le compilateur de silicium SYCO. [JER86] présente trois structures de contrôle qu'on peut générer automatiquement. Il s'agit:

- d'étages de contrôle mono-PLA,
- d'étages de contrôle microprogrammés,
- d'étages de contrôle à générateur de temps.

Dans le chapitre suivant, on montrera comment le CPC génère

automatiquement des étages de contrôle mono-PLA. Dans le dernier chapitre, on démontrera la faisabilité de la génération d'étages de contrôle microprogrammés automatiquement par le CPC.

### **III- La forme intermédiaire des compilateurs des parties contrôle**

Pour un compilateur de partie contrôle, le choix de la forme intermédiaire est très important car le résultat final du processus de synthèse peut dépendre de cette forme. L'inconvénient majeur de certaines formes intermédiaires est que certaines transformations (logiques ou topologiques) qui leur sont appliquées aboutissent à une perte d'informations concernant la description comportementale à partir de laquelle la forme intermédiaire a été générée.

La forme intermédiaire peut être un graphe de précedence, un réseau de Petri, un ensemble d'équations logiques, etc.

Dans le système CAMAD [PEN86], la description comportementale est transformée en un réseau de Petri étendu temporisé (extended timed Petri net: ETPN). Ce réseau ETPN permet une représentation graphique du comportement du circuit. Le processus de synthèse est alors accompli par une série de transformations de l'ETPN tout en conservant la sémantique du circuit. Par contre, le système ALGIC [JOE85, JOE86] transforme la description de départ en un ensemble de graphes de précedence. A partir de ces graphes, un ensemble hiérarchique de machines d'états finis (MEF) est évalué. Chaque MEF est représentée par une table d'états permettant la génération de la description logique du bloc physique réalisant l'automate.

Le système CMU-DA [Bla85] illustre bien l'inconvénient que peuvent présenter certaines formes intermédiaires. En effet, la description comportementale d'entrée est transformée en une base de données globale (GDB) à partir de laquelle un fichier de "value trace" (VT) est généré. Des transformations topologiques sont appliquées à ce fichier de VT et consistent à ajouter, éliminer ou déplacer des opérateurs dans cette représentation. Après ces transformations, on perd certaines informations de la description d'entrée, surtout dans le cas d'élimination d'opérateurs. [SNO78] donne des exemples de transformations du fichier VT.

Pour remédier à ce problème, [TSU86] suggère de développer des outils qui soient capables de vérifier l'équivalence entre les spécifications de bas niveau et celles de haut niveau.

On donne encore plus d'importance à la forme intermédiaire car la

force et les performances d'un compilateur de partie contrôle lui sont étroitement liées. Le choix de la forme intermédiaire dépend du langage d'entrée du compilateur, ce qui prouve encore l'importance du choix du langage d'entrée d'un compilateur de silicium.

### Exemple:

Comme exemple concret, on prendra le système Mac-Pitts [SIS82, SOU83]. Les circuits générés par ce système sont composés d'une partie opérative et d'une partie contrôle. La partie contrôle est implémentée sous forme de machines d'états finis. Ces MEFs sont composées d'une logique combinatoire attachée à un registre d'état qui mémorise l'état courant de l'automate. Cette logique doit générer l'état suivant de l'automate seulement quand il y a un branchement explicite. Cette logique combinatoire est réalisée par une structure "Weinberger". La description d'entrée est donnée dans un langage proche du langage LISP. Elle est transformée en une forme intermédiaire en terme de formules booléennes. Soit la description de l'état "Execute":

```
Execute (Cond (exch (setq A B)
 (setq B A)
 (go Fetch))
 (incr (setq A (+ A 1))
 (setq B (+ B 1))
 (go Fetch))
 (t (setq A (- A 1))
 (setq B (- B 1))
 (go Fetch)))
```

On remarque que le flux de contrôle est exprimé par la forme **Cond**. Cependant, les conditions des différentes alternatives de cette forme sont mutuellement exclusives. Cette contrainte risque de poser des problèmes puisque l'exclusion mutuelle est assurée par interprétation de la forme Cond. En effet, pour interpréter la description de l'état "Execute", on évalue la première condition. Si elle est vraie, on exécute les actions qui lui sont associées et on se branche à l'état indiqué. L'interprétation de cet état s'arrête à ce niveau même si une autre condition est vraie en parallèle avec la première. Si la condition est fausse, on passe à la suivante.



La forme intermédiaire correspondant à cette description est la suivante:

exch / A<--B , B<--A / Fetch

( $\neg$  exch ) And incr / A<--A+1 , B<--B+1 / Fetch

( $\neg$  exch ) And ( $\neg$  incr ) / A<--A-1 , B<--B-1 / Fetch

Chaque phrase de la forme intermédiaire est composée de trois champs:

- la condition de transition,
- les actions à exécuter en parallèle, séparées par des virgules,
- l'état suivant.

Cette forme décrit des automates de Mealy.

L'exemple donné nous montre clairement la dépendance de la forme intermédiaire du langage d'entrée, surtout au niveau des conditions qui doivent être générées de sorte à respecter leur exclusion mutuelle. De plus, on peut voir la répercussion du choix du langage d'entrée sur la façon d'exprimer un algorithme.

Supposons que l'on veuille affecter aux variables A, B et C des valeurs selon les valeurs des variables booléennes X, Y et Z suivant l'algorithme suivant:

Si X alors A<-- A+D;

Si Y alors B<-- B+V;

Si Z alors C<-- C+H;

En utilisant le langage d'entrée de Mac-Pitts, on aura tendance à écrire cet algorithme comme suit:

```
Choix (Cond (X (setq A (+ A D)))
 (Y (setq B (+ B V)))
 (Z (setq C (+ C H))))
```

Les formules booléennes correspondant à cet état sont:

X / A<--A+D / ?

( $\neg$  X ) And Y / B<--B+V / ?

( $\neg$  X ) And ( $\neg$  Y ) And Z / C<--C+H / ?

Par conséquent, on ne peut jamais affecter plus d'une variable à la fois. Or l'algorithme de départ ne correspond pas exactement à ces formules. Donc, si le concepteur désire décrire cet algorithme, il doit énumérer tous les cas possibles, autrement dit, il doit donner la description suivante:

```
Choix (Cond ((X and ¬ Y and ¬ Z) (setq A (+ A D)))
 ((X and Y and ¬ Z) (setq A (+ A D))
 (setq B (+ B V)))
 ((X and Y and Z) (setq A (+ A D))
 (setq B (+ B V))
 (setq C (+ C H)))

)
```

On voit qu'un état aussi simple que celui-là occupera relativement beaucoup de place dans le circuit.

#### IV- Transformations de la forme intermédiaire

Pour tous les compilateurs de partie contrôle, cette étape consiste à appliquer un traitement à la forme intermédiaire dans le seul but de réduire la surface de silicium nécessaire à l'implémentation de la partie contrôle. Ce traitement est généralement composé de deux parties:

**Optimisation logique:** elle consiste à simplifier l'algorithme réalisé par la partie contrôle. Les règles appliquées dépendent de la forme intermédiaire et du niveau d'abstraction dans lequel elle est exprimée. Par exemple, dans le cas des équations booléennes, on applique des règles qui permettent de simplifier ces équations. Dans [HAN86] se trouve un ensemble de règles applicables aux équations booléennes. Par contre dans le cas d'un graphe de précedence, ces transformations consistent à réduire le nombre d'états et d'arcs du graphe.

**Optimisation topologique:** elle consiste à faire des transformations pour minimiser les opérateurs utilisés par le circuit et mettre en commun des blocs fonctionnels. Ces transformations dépendent aussi de la forme intermédiaire utilisée. [SNO78] présente les transformations appliquées au graphe du flux des données pour réduire

le nombre d'opérateurs implémentés dans le circuit. On verra plus loin les transformations que peut faire le système ARTS pour améliorer la régularité des circuits générés par SYCO et obtenir un meilleur compromis surface/vitesse.

Généralement, le résultat de ces transformations dépend de l'ordre dans lequel elles sont appliquées [PEN86].

### **V- Génération de la description logique**

Cette étape concerne les compilateurs dont la forme intermédiaire décrit un automate d'états finis. La partie contrôle peut être réalisée par un ou plusieurs automates éventuellement interconnectés. Chaque automate sera implémenté par un bloc physique. Cette étape consiste alors à générer la table de vérité de l'ensemble des fonctions réalisées par chaque automate. Il s'agit donc de coder les entrées et les sorties de chaque automate. Les entrées sont composées de deux champs:

- 1) l'état courant de l'automate,
- 2) les conditions de transitions de l'automate.

Quant aux sorties, elles sont aussi composées de deux champs:

- 1) l'état suivant de l'automate,
- 2) les sorties de l'automate.

Le codage est généralement choisi de sorte à pouvoir minimiser la surface de silicium nécessaire pour implémenter l'automate, autrement dit de sorte à pouvoir appliquer d'autres règles d'optimisation logique sur la table de vérité pour simplifier davantage les fonctions logiques exprimées par la table.

Le codage des conditions de transitions et des sorties de l'automate ne pose pratiquement pas de problèmes. Par contre tous les compilateurs se heurtent au problème de codage des états de l'automate. Ce problème est spécifique des parties contrôle réalisées par des PLAs et peut être divisé en deux sous-problèmes:

- 1- le choix du code utilisé pour le codage des états,
- 2- l'assignation d'un code à chaque état.

#### **V-1- Le choix du code utilisé**

Ce choix est important car il détermine le nombre de bits nécessaires pour le codage. Les codes les plus utilisés sont:

# le code **CBP** ( code binaire pur ): le nombre de bits nécessaires pour coder  $2^N$  états est N. Le code de l'état i est donné par:

$$CBP(i) = \log_2(i);$$

# le code **GRAY**: le nombre de bits nécessaires pour coder  $2^N$  états est N. Le code de l'état i est déduit du code CBP(i) par la formule:

$$GRAY(i) = (CBP(i) \oplus 2(CBP(i)))/2;$$

# le code **1 parmi N**: Pour N états, on aura besoin de N bits. Le code de l'état i est  $1N(i) = \langle 00...1...00 \rangle$  avec 1 à la position i.

COPPOLA, dans [COP86], démontre qu'il est difficile de choisir un type de code qui soit le meilleur dans tous les cas. En effet, il présente une table comparative d'utilisation des différents codes et démontre que le résultat final du codage dépend non seulement du code utilisé mais aussi de l'automate codé. Pour décider du choix du code utilisé, [COP86] essaie les différents codes et retient la solution qui donne le meilleur résultat après optimisation logique faite par ESPRESSO [RUD85]. Cependant, il est souhaitable de réduire le nombre de bits nécessaires pour coder les états, ce qui permettrait de réduire le nombre d'entrées et de sorties du PLA.

### V-2- L'assignation du code aux états

Ce problème est plus crucial que le précédent. Etant donné un ensemble d'états désignés par leurs étiquettes, le problème revient à choisir une représentation binaire pour chaque état de sorte à réduire le nombre de monômes du PLA réalisant l'automate.

Ce problème est de nature combinatoire et les techniques mises en œuvre dans les années 60 par [ARM62, KAR64] s'avèrent limitées à des exemples très simples qui ne montrent pas réellement la complexité du problème pour le cas de contrôleurs réalistes. Des méthodes heuristiques ont été développées afin de résoudre ce problème dans un temps raisonnable. Ces heuristiques sont évidemment applicables dans le cas de contrôleurs de taille importante tels que les parties contrôle de microprocesseurs.

[SAU87] présente un certain nombre de règles pour l'assignation des états d'un automate tout en différenciant les règles applicables pour le codage d'un état successeur de celles applicables pour le codage d'un état prédécesseur. On peut cependant remarquer que ces règles sont assez contraignantes puisqu'elles ne peuvent être appliquées que si certaines

conditions sont vérifiées. En effet, pour un état ayant  $P$  prédécesseurs, il faut que tous ces prédécesseurs aient la même condition sur les arcs les reliant à l'état à coder. Par contre si un état possède  $S$  successeurs, il faut que la somme de toutes les conditions sur les arcs entre l'état en question et ses successeurs forme une tautologie. Donc cet ensemble de règles n'est applicable qu'à des automates particuliers. En plus, il n'est pas toujours évident de vérifier la tautologie surtout quand le nombre de variables de condition dépasse 10.

Les deux problèmes du choix du code et de l'assignation des états ne peuvent pas être traités séparément. La solution optimale, qui minimise le nombre d'entrées/sorties et le nombre de monômes du PLA, est un compromis des solutions à ces deux problèmes.

De MICHELI [DeM84a, DeM84b] traite ces deux problèmes simultanément de façon astucieuse. Son idée consiste à coder les champs correspondant aux conditions de transitions et aux sorties de l'automate, et garder les états (courants et suivants) sous forme de mnémoniques. Le problème revient alors à trouver une représentation de l'automate avec un nombre minimum de symboles. Ceci conduit à une minimisation symbolique qui n'est autre que la minimisation de fonctions logiques multi-valeurs [RUD87, RUD86, BRA86] où chaque symbole représentant un état courant correspond à différentes valeurs logiques d'entrée et chaque symbole représentant un état suivant correspond aux différentes fonctions de sortie.

L'algorithme part d'une matrice qui correspond au codage de départ de l'automate où les états sont gardés sous forme symbolique.

Solent: **Ns** le nombre de mnémoniques à coder,

**Nl** le nombre de groupes, un groupe étant un ensemble de lignes de la matrice de départ correspondant au même état courant,

**Nb** le nombre de bits nécessaires pour le codage des mnémoniques.

L'algorithme utilise une matrice de contraintes  $A \in \{0,1\}^{Nl \times Ns}$ . Un mnémonique  $m_j$  appartient au groupe  $i$  si  $a_{ij}=1$  dans la matrice  $A$ .

Le résultat de l'algorithme est une matrice  $S \in \{0,1\}^{Nl \times Nb}$ . Le traitement consiste alors à déterminer la matrice  $S$  partant de la matrice de départ tout en respectant les contraintes de matrice  $A$ .  $S$  est construite ligne par ligne et colonne par colonne. Initialement,  $S$  est une matrice à une seule

colonne. L'algorithme est le suivant:

- (1) sélectionner un mnémonique non codé,
- (2) déterminer le code de ce mnémonique satisfaisant aux conditions de la matrice A,
- (3) si aucun code ne peut lui être affecté augmenter le nombre de bits pour le codage ( $Nb \leftarrow Nb + 1$ ); aller à (2),
- (4) affecter le code au mnémonique,
- (5) s'il reste encore des mnémoniques non codés aller à (1).

A première vue, cet algorithme semble assez performant, pourtant le résultat dépend quand même de l'ordre d'arrivée des états à coder. Donc, pour trouver la meilleure solution, il est nécessaire d'essayer toutes les combinaisons possibles de l'ordre d'arrivée des états. Ceci n'est possible que si tous les branchements sont spécifiés explicitement, or pour un automate de N états, on aura  $N!$  combinaisons possibles, par conséquent le temps de calcul risque d'être prohibitif pour les automates complexes. En outre, le nombre de bits nécessaires pour le codage n'est pas minimal. En effet, dans l'exemple présenté dans [DeM84b], pour neuf mnémoniques, neuf bits ont été utilisés pour le codage des états. Cet exemple nous montre qu'on ne peut pas trouver une solution qui permette de réduire le nombre d'entrées/sorties et le nombre de monômes du PLA.

## VI- Conclusion

Dans ce chapitre, on a présenté les principales tâches qu'un compilateur de partie contrôle est censé assurer. On a aussi évoqué les principaux problèmes et difficultés auxquels se heurtent ces outils. Plusieurs compilateurs de partie contrôle ont été, ou sont en train d'être, développés. Dans le chapitre suivant, la présentation du compilateur CPC nous permettra de voir comment aborder réellement les problèmes qui se posent.



# ***CHAPITRE 4***

**CPC: Le Compilateur de Partie**

**Contrôle**





## **I- Introduction**

Le compilateur de partie contrôle CPC est un outil dont le rôle est de transformer la description comportementale de chaque étage de contrôle générée par EXTRACT, en une forme qui permet l'implémentation de la partie contrôle.

Comme on peut le voir sur la figure 2.12 (chapitre 2), le CPC joue un rôle important dans le compilateur de silicium SYCO. A partir de la description hiérarchique, obtenue par EXTRACT, le CPC se charge de générer les données d'entrée de tous les autres outils constituant SYCO (ARTS, APOLLON, GEN-TOPO).

Le CPC a été défini compte tenu de son interface avec tous les autres outils et des transformations topologiques que ARTS peut appliquer sur la description du circuit. En particulier, on a choisi de garder la forme intermédiaire en LDS pour assurer l'indépendance de ARTS par rapport au CPC. Ainsi on permet au concepteur, qui retrouve toutes les informations qu'il a introduites dans la description de départ, d'intervenir à toute étape du processus de synthèse du circuit.

Le CPC communique avec APOLLON et GEN-TOPO via des structures de données (description des étages de contrôle et fichier de communication), ce qui évite le traitement de données textuelles et assure la cohérence du système global.

Etant donné que la partie contrôle générée par le CPC est hiérarchique, un problème de synchronisation des différents étages de contrôle se pose. Le CPC génère, automatiquement, deux variables de contrôle par étage pour permettre au GEN-TOPO d'installer un protocole de communication pour les synchroniser.

Dans la présente version du CPC, chaque étage de contrôle est implémenté par un PLA unique. Pour chaque étage, le processus de compilation est composé de trois étapes :

- 1) transformation de la description d'entrée en une table d'états (table de transition),
- 2) optimisation logique et ajout des branchements implicites.
- 3) génération de la description logique du PLA qui doit réaliser l'étage de contrôle.

A partir de la table d'états du dernier étage, le CPC génère la description de la partie opérative.

Le CPC se charge également de la génération d'un fichier de communication qui permettra l'assemblage des étages de contrôle et de la partie opérative.

## II- Le langage d'entrée du CPC

Le CPC reçoit une description comportementale et structurelle du circuit à générer. Cette description est composée de trois types d'entités:

1) Le Smodule : (voir langage d'entrée de SYCO au chapitre 2) : Pour le CPC, le Smodule sert surtout à connaître les différentes variables de contrôle internes, les variables de contrôle externes (Interface du circuit avec le monde extérieur) et les registres implémentés dans la partie opérative.

2) Le Fmodule : Un Fmodule est une entité permettant de définir des opérateurs complexes réalisés par la POP. Le CPC s'en sert pour déterminer les opérateurs élémentaires ( + , - .... ) qui servent à la définition de cet opérateur complexe (chapitre 2).

3) Le Cmodule : Le CPC reçoit un ensemble de Cmodules, un pour chaque étage de contrôle. Un Cmodule est composé d'un ensemble (non vide) d'automates indépendants avec différents points d'entrée. Un point d'entrée correspond à l'état initial d'un automate du Cmodule. Cette description correspond à une hiérarchie où un Cmodule peut activer un automate de l'étage inférieur en désignant son état initial (point d'entrée) comme paramètre d'appel. La figure 2.7 donne un exemple de Cmodules d'entrée du CPC.

Dans un Cmodule, tous les états des automates sont étiquetés (par EXTRACT). Un état est donc désigné par un nom et composé d'un ensemble (non vide) d'instructions à exécuter en parallèle. Une instruction peut être:

1) Une instruction inconditionnelle du type:

- \* NEXT étiqu où étiqu appartient au même automate du Cmodule que l'instruction puisque les automates sont indépendants,
- \* EXECUTE étiqu : étiqu est un état initial d'un automate appartenant au Cmodule inférieur,
- \* SET et RESET pour la mise à "1" et "0" d'une variable de contrôle booléenne. Si la variable est structurée l'opérateur d'affectation "!=" sert pour l'affectation d'une valeur à cette variable,
- \* CALL : cette instruction ne peut figurer que dans l'étage le plus bas et consiste à appeler un opérateur complexe (défini par un Fmodule),
- \* " : =" l'opérateur d'affectation sert à désigner le transfert entre registres ou entre constante et registre. De même, seul l'étage de contrôle le plus bas peut contenir des instructions de transfert de registre.

## 2) Une instruction conditionnelle du type :

IF Cond ACTION ; ENDIF;

**ACTION** est un ensemble non vide d'instructions inconditionnelles. On remarque que la description a été mise à plat par **EXTRACT** et qu'il n'existe plus d'imbrication de IF.

Pour un état donné, les conditions des différentes instructions IF ne sont pas obligatoirement mutuellement exclusives. Par contre, toutes les conditions peuvent être fausses à la fois.

Dans une condition, les seuls opérateurs permis sont

- les opérateurs d'égalité '=', d'inégalité "<>"
- les opérateurs logiques : NOT, OR, AND, XOR.

Les autres opérateurs de comparaison ( >= , <= , > , < ) sont interdits car sinon il y aurait implantation d'opérateurs dans la partie contrôle, qui perdrait donc de sa régularité. Cependant, le concepteur a toujours la possibilité d'utiliser ces opérateurs par l'intermédiaire de l'entité Fmodule qui peut avoir 5 paramètres

Fmodule ALU-STD ( Res, op1, op2, flag, oper )

- Res correspond au registre qui contiendra le résultat de l'opération (il

peut être omis)

- op1 et op2 sont les opérandes
- flag : est le registre qui contiendra les comptes rendus correspondant à l'opération ( carry, zéro, overflow )
- oper est l'opérateur qui peut être un numéro si le Fmodule définit un opérateur complexe.

### Exemple 1

Fmodule BOP1 ( A, B, C, D, OP )

CASE (OP)

when (0) A := ( B + C ) \* ( B - C );

when (1) A := ( B + C ) \* ( B + C );

when (2) A := ( B - C ) \* ( B - C );

end;

end;

Dans cet exemple on récupère dans le registre A un résultat dépendant du paramètre OP et dans le registre D les comptes rendus de l'opération.

### Exemple 2

Fmodule COMP ( , B, C, D, OP )

CASE (OP)

when ( '<=' ) B <= C;

when ( '>=' ) B >= C;

when ( '>' ) B > C;

when ( '<' ) B < C;

end;

end;

Dans ce deuxième exemple, le paramètre OP est une chaîne de caractères qui désigne l'opérateur de comparaison qu'on veut appliquer, néanmoins, on aurait pu procéder de la même façon que pour l'exemple 1, c'est-à-dire donner un numéro à chaque opérateur au lieu d'utiliser des chaînes de caractères. Le registre résultat est omis.

L'appel se fait par CALL BOP1 ( X, Y, Z, T, 0 ) ou CALL COMP( , Y, Z, T, '<=' ).

Dans la description d'un circuit, on peut distinguer trois classes de Cmodules:

**classe1** : Le Cmodule est composé d'un seul automate. Généralement on n'a qu'un seul Cmodule de cette classe. Il correspond à l'étage le plus haut.

**classe2** : Le Cmodule est composé de plusieurs automates indépendants et ne contenant aucune instruction opérative (transfert de registre...). Cette classe correspond aux étages de contrôle intermédiaires dans la hiérarchie.

**classe3** : Le Cmodule est composé de plusieurs automates indépendant et ne contenant pas d'instruction d'appel de point d'entrée mais contenant des instructions opératives. Cette classe correspond à l'étage le plus bas dans la hiérarchie. Une description contient un seul Cmodule de cette classe.

Un Cmodule peut appartenir aux classes 1 et 3 en même temps, c'est le cas d'une partie contrôle à un seul étage de contrôle.

La description d'un circuit contient:

- un Cmodule de la classe1,
- un Cmodule de la classe3,
- un nombre quelconque ( $\geq 0$ ) de Cmodules de classe2.

### III- Structure d'un étage de contrôle

La structure d'un étage de contrôle est donnée par la figure 4.1.

Les entrées du PLA sont composées de quatre champs:

- 1) les commandes émanant de l'étage supérieur (INST<sub>i</sub>),
- 2) l'entrée du séquençement interne (seq),
- 3) les variables de contrôle internes et externes (bus de contrôle),
- 4) les variables de compte rendu émanant de la POP (bus de compte rendu).

En sortie le PLA génère :

- 1) les sorties du séquençement interne (seq),
- 2) les commandes destinées à l'étage inférieur (INST<sub>i-1</sub>)
- 3) les commandes de manipulation de variables de contrôle. A chaque variable, on associe deux fils de commande (un pour le SET et un pour le RESET),
- 4) les fils de synchronisation des différents étages de contrôle.

Remarquons par ailleurs qu'une variable de contrôle peut être représentée par une boîte à trois ports (une sortie et deux entrées). La sortie permet de lire la valeur de la variable alors que les deux entrées servent à modifier la valeur de cette variable. Les trois valeurs des entrées utilisées sont:

- 00: variable non modifiée,
- 01: variable mise à 1,
- 10: variable mise à 0.

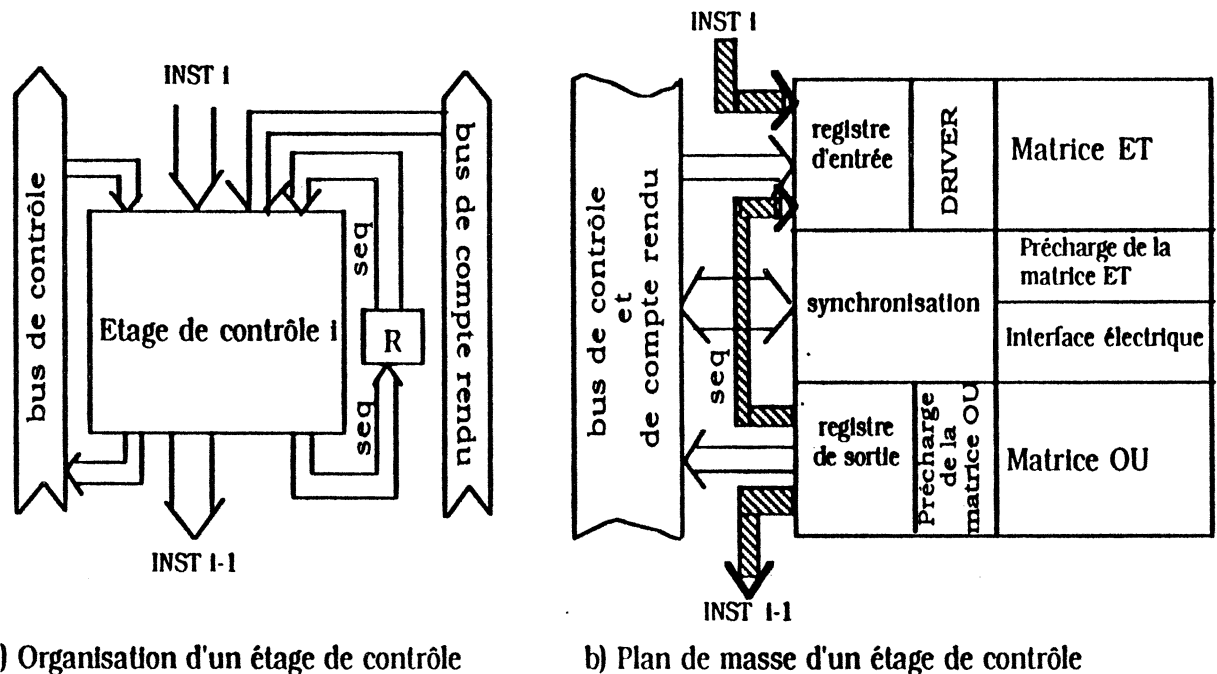


Figure 4.1: Structure d'un étage de contrôle [GER87]

#### IV- Génération de la table d'états:

Ce traitement consiste à transformer la description de chaque étage de contrôle de sorte que les conditions ne contiennent que les opérateurs élémentaires "=" et "AND". Cette transformation est nécessaire car on a choisi de ne pas implémenter d'opérateurs dans la partie contrôle. Ces deux opérateurs sont les seuls qui peuvent être réalisés par des fils d'entrée d'un PLA. Ces transformations sont nécessaires indépendamment de la structure de l'étage de contrôle (structure microprogrammée ou à PLAs). Dans le cas d'étages de contrôle microprogrammés, seul le séquençement est conditionnel. Comme le mécanisme de séquençement sera réalisé par un PLA, on reste toujours dans les mêmes hypothèses de transformations.

Dans la description transformée, chaque instruction correspond à une

transition d'un automate de l'étage de contrôle. Le codage direct de la table d'états aboutit à la description logique du PLA réalisant l'étage. Chaque instruction de la table d'états correspond à un monôme du PLA. On verra dans le paragraphe V par quels moyens le CPC réduit le nombre de monômes du PLA.

Ces transformations ont pour but de générer une table d'états qui vérifie les conditions suivantes:

- 1) chaque instruction conditionnelle possède un branchement (NEXT ou EXIT),
- 2) si un état<sub>i</sub> possède un branchement inconditionnel à l'état<sub>j</sub>, tous les branchements conditionnels mènent à cet état<sub>j</sub>,
- 3) aucun automate ne se bloque,
- 4) aucun branchement à deux états différents d'un automate au même instant n'est possible,
- 5) aucune activation de deux points d'entrée de l'étage inférieur au même instant n'est possible.

Outre la transformation des opérateurs, cette étape effectue le recensement, qui consiste à préparer certaines informations utiles à la suite de la compilation.

Pour le CPC, la table d'états, qui sera maintenue en LDS, représente la forme intermédiaire sur laquelle toutes les transformations logiques et topologiques seront appliquées.

#### IV-1- Transformation des opérateurs complexes

Dans la version actuelle, le seul opérateur logique complexe traité par le CPC est le Ou exclusif "EXOR" (représenté par le symbole %).

Le traitement consiste à remplacer une expression contenant cet opérateur par une expression équivalente ne contenant que les opérateurs élémentaires OR( / ), AND ( • ), NOT (#), =, et <>. Il s'agit d'appliquer la formule de l'algèbre de Boole  $A \oplus B = \neg A \cdot B / A \cdot \neg B$

#### Exemple:

(A 0:3 ='001') % (B 0:2 ='01')

"A 0:3" désigne le champ du registre "A" commençant au bit "0" et de taille 3 bits.

En appliquant la formule donnée plus haut, on obtient:



$$(A\ 0:3 = '001') \cdot \#(B\ 0:2 = '01') / \#(A\ 0:3 = '001') \cdot (B\ 0:2 = '01')$$

Cette transformation fait apparaître l'opérateur NOT (#) qui sera à son tour transformé.

#### IV-2- Transformation de l'opérateur NOT

Cet opérateur peut être suivi d'une expression simple ou complexe. Dans le premier cas, il s'agit de remplacer l'opérateur "=" par "<>" et vice-versa. Pour le cas d'une expression complexe, il s'agit d'appliquer le théorème de DE MORGAN:

$$\neg(\sum a_i) = \prod \neg a_i$$

$$\neg(\prod a_i) = \sum \neg a_i$$

Si par contre l'expression est une variable booléenne (#A), le traitement consiste à introduire l'opérateur d'égalité (A 0:1 = '0').

En poursuivant le traitement sur l'exemple précédent, on obtient l'expression suivante:

$$(A\ 0:3 = '001') \cdot (B\ 0:2 <> '01') / (A\ 0:3 <> '001') \cdot (B\ 0:2 = '01')$$

On voit qu'on doit introduire l'opérateur d'inégalité "<>" qui ne doit pas figurer non plus dans la table d'états finale.

#### IV-3- Transformation de l'opérateur d'inégalité <>

L'opérande droit de cet opérateur ne peut être qu'une valeur. Une expression simple contenant cet opérateur est transformée en une expression complexe. Celle-ci est obtenue en introduisant l'opérateur logique OR et en utilisant la valeur indéfinie "x" dans les opérandes droits des expressions simples composant cette expression complexe. L'utilisation de la valeur indéfinie doit assurer la couverture de toutes les valeurs prises par la variable de départ.

##### Exemple:

Soit l'expression simple (A 0:4 <> 9) qui équivaut à (A 0:4 <> '1001'). Elle est transformée en:

$$(A\ 0:4 = 'xxx0') / (A\ 0:4 = 'xx1x') / (A\ 0:4 = 'x1xx') / (A\ 0:4 = '0xxx')$$

L'intérêt d'utiliser la valeur indéfinie "x" est de réduire le nombre d'expressions simples liées par les opérateurs OR. En effet, dans cet exemple, le champ utilisé de la variable A est de taille 4 bits, donc on peut avoir  $2^4=16$  valeurs. L'expression (A 0:4 <> 9) correspond à 15 valeurs

différentes de ce champ, elle est alors équivalente à:

$((A_{0:4}=0) \text{ OR } \dots \text{ OR } (A_{0:4}=8) \text{ OR } (A_{0:4}=10) \text{ OR } \dots \text{ OR } (A_{0:4}=15))$ .

Or, en utilisant la valeur indéfinie "x", on n'a eu que quatre expressions simples liées par des opérateurs OR. Dans le cas général, pour une variable de N bits, on génère N expressions simples au lieu de  $2^N - 1$ .

Pour le CPC, une valeur entière est transformée en une représentation binaire. Cependant, si la taille de la variable n'est pas suffisante pour contenir la valeur, le CPC fait une troncature à gauche et ne garde que les bits de poids faible (  $(A_{0:3}=9)$  devient  $(A_{0:3}='001')$ ).

Inversement, si la taille de la variable est supérieure au nombre de bits nécessaires pour représenter la valeur en binaire, le CPC complète les bits de poids fort par des zéros ( $(A_{0:5}=3)$  devient  $(A_{0:5}='00011')$ ).

Si on reprend l'expression traitée par les précédentes transformations, on obtient:

$((A_{0:3}='001') \cdot ((B_{0:2}='x0') / (B_{0:2}='1x'))$   
 $/ ((A_{0:3}='xx0') / (A_{0:3}='x1x') / (A_{0:3}='1xx')) \cdot (B_{0:2}='01'))$

qui devient à la fin:

$((A_{0:3}='001') \cdot (B_{0:2}='x0') / (A_{0:3}='001') \cdot (B_{0:2}='1x')$   
 $/ (A_{0:3}='xx0') \cdot (B_{0:2}='01') / (A_{0:3}='x1x') \cdot (B_{0:2}='01')$   
 $/ (A_{0:3}='1xx') \cdot (B_{0:2}='01'))$

#### IV-4- Transformation de l'opérateur OR

Dans le cas des autres transformations, on ne s'est intéressé qu'à l'expression elle-même. Pour cet opérateur, on va s'intéresser plutôt à l'instruction conditionnelle dont la condition contient cet opérateur.

Une telle instruction sera éclatée en autant d'instructions conditionnelles que de termes liés par des opérateurs OR. Ces instructions auront la même action que celle de départ.

#### Exemple:

If  $(A_{0:4}='1001' / B_{0:2}='01')$  NEXT Etat<sub>i</sub>; Endif;

cette instruction est éclatée en:

If  $(A_{0:4}='1001')$  NEXT Etat<sub>i</sub>; Endif;

If  $(B_{0:2}='01')$  NEXT Etat<sub>i</sub>; Endif;

Comme chaque instruction conditionnelle correspond à un monôme dans le PLA, l'utilisation de la valeur indéfinie "x" permet alors de réduire la

surface occupée par le PLA réalisant l'étage de contrôle.

#### IV-5- Le recensement

Le recensement permet de collecter certaines informations nécessaires pour la suite de la compilation. Il consiste à déterminer:

1) les points d'entrée de l'étage inférieur. Ces points d'entrée permettent de délimiter les automates indépendants de l'étage inférieur et de définir le nombre de bits nécessaires pour le codage des commandes destinées à l'étage inférieur.

2) Le nombre d'états du plus grand automate (en terme de nombre d'états) de l'étage courant. Ceci permet de définir le nombre de bits pour le codage du séquençement interne de l'étage courant.

3) La liste des variables de contrôle et de compte rendu utilisées dans les conditions des différents automates de l'étage et la liste des variables de contrôle modifiées par l'étage en cours.

Par souci de minimiser le nombre d'entrées et de sorties du PLA réalisant l'étage de contrôle, le CPC ne mémorise que les bits de registres (ou de variables de contrôle) effectivement utilisés.

#### Exemple:

Si on utilise les champs RI 0:4 et RI 7:1, le CPC ne mémorise que les numéros des bits utilisés (0 1 2 3 7) et non pas les bornes inférieure et supérieure de chaque registre.

A l'issue du recensement, le CPC génère deux vecteurs:

1) le vecteur de condition: chaque élément de ce vecteur correspond à un bit d'un registre ou d'une variable de contrôle testé dans les conditions. La position de chaque bit dans ce vecteur est fixée une fois pour toutes, ce qui permettra le codage de toutes les conditions de façon unique.

Dans ce vecteur, les bits des variables de contrôle occupent la partie gauche, alors que les bits des variables de compte rendu (registres) occupent la partie droite.

#### Exemple:

Supposons avoir utilisé les bits (0 3 5) d'une variable de contrôle C, et les bits (0 1 2 3 7) d'un registre RI de la partie opérative. Le CPC, génère un vecteur de condition V où:

$V[0]=C(0)$ ,  $V[1]=C(3)$ ,  $V[2]=C(5)$ ,

$V[3]=RI(0)$ ,  $V[4]=RI(1)$ ,  $V[5]=RI(2)$ ,  $V[6]=RI(3)$ ,  $V[7]=RI(7)$ .

L'ordre des variables dans ce vecteur est fonction de leur apparition dans la description.

2) Le vecteur de contrôle: Il est construit de la même façon que le vecteur de condition mais il correspond aux variables de contrôle modifiées par l'étage. Ce vecteur permettra le codage des commandes de manipulation des variables de contrôle.

### Conclusion

Etant donné que l'outil EXTRACT ne génère qu'une seule instruction inconditionnelle comme action d'une instruction conditionnelle et vu les différentes transformations appliquées par le CPC à la forme intermédiaire (table d'états), le nombre de monômes du PLA réalisant un étage de contrôle va être, inutilement, important. Il est alors impératif de faire une optimisation logique de cette forme intermédiaire avant de passer à l'optimisation topologique du circuit.

## V- Optimisation logique et ajout du branchement implicite

### V-1- Optimisation logique

Le but de ce traitement est de simplifier la description de chaque étage de contrôle, donc de réduire la surface du PLA qui réalise cet étage. Cette optimisation est locale à chaque état des différents automates.

Soit un état de N instructions:

ETAT ( If Cond<sub>1</sub> Action<sub>1</sub> Endif;

• • • • •

If Cond<sub>n</sub> Action<sub>n</sub> Endif;)

Une instruction inconditionnelle est représentée par une instruction conditionnelle dont la condition est toujours vraie.

Le principe de l'optimisation repose sur la comparaison des N conditions deux à deux dans le but de détecter les cas où il est possible d'éliminer l'une d'entre elles ou de les fusionner.

Un des plus grands problèmes rencontrés à ce niveau est la comparaison des conditions. En effet, pour pouvoir comparer deux conditions en LDS (sous forme textuelle), il est impératif de les trier.

**Exemple:**

```
Etat (If ((A 0:2='11') • (B 0:1= '0')) Action1; Endif;
 If ((B 0:1= '0') • (A 0:2='11')) Action2; Endif;)
```

Dans cet exemple, vu la commutativité de l'opérateur logique "•"(AND), les deux conditions sont logiquement identiques. Par conséquent, les deux instructions peuvent être remplacées par une seule dont l'action est l'union des actions Action1 et Action2.

Une première solution consiste à trier les conditions en tenant compte seulement des identificateurs des registres et variables utilisés. Mais on s'aperçoit tout de suite que l'identificateur seul ne suffit pas pour trier les conditions, étant donné que plusieurs champs du même registre peuvent figurer dans la même condition.

**Exemple:**

```
Etat (If ((RI 0:2= '00') • (A 0:2='11') • (RI 5:2= '1x')) Action1; Endif;
 • • •)
```

Pour trier cette condition, il est nécessaire de tenir compte des champs des registres. Ce problème devient plus complexe quand les champs d'un même registre se chevauchent ou sont imbriqués.

**Exemple:**

```
Etat (If ((RI 0:2= '00') • (A 0:2='11') • (RI 1:2= '01')) Action1; Endif;
 • • •)
```

Il est vrai que le troisième terme de cette condition peut se réduire à (RI 2:1='1') mais rien n'empêche le concepteur d'écrire cette condition sous cette forme là.

La solution adoptée par le CPC repose sur le codage des conditions. En effet, on a vu que pendant le recensement le CPC génère un vecteur de condition dans lequel chaque variable occupe une position bien déterminée. Donc l'idée est de coder dès cette étape les différentes conditions puis de comparer les vecteurs. Cette solution est plus simple à mettre en œuvre et permet de réduire le temps de compilation.

**Exemple:**

Supposons avoir les champs des variables suivantes: A 0:2, RI 1:3, RI 5:2. On suppose aussi que les bits 0 et 1 de la variable A occupent les bits

(0 1) du vecteur de condition et que les bits 1, 2, 3, 5 et 6 du registre RI occupent respectivement les bits (2 3 4 5 6) de ce même vecteur de condition. Si on a les deux instructions suivantes:

```
(If ((RI 1:3= '001') • (A 0:2='00')) • (RI 5:2= '11')) Action1; Endif;
```

```
 If ((RI 1:3= '111') • (A 0:2='01')) Action2; Endif;)
```

les deux vecteurs de codage correspondant à ces deux conditions seront respectivement: [00 001 11] et [01 111 xx]. Donc la comparaison ne pose plus aucun problème.

Pour chaque couple d'instructions (dont les conditions sont Condi et Condj), le CPC applique trois règles d'optimisation:

### Règle1 Règle d'identité:

Si Condi et Condj sont égales alors les deux instructions en question sont remplacées par une seule ayant la même condition (Condi ou Condj) et dont l'action est l'union des actions des instructions comparées.

#### Exemple:

```
Etat (If (RI 1:3= '001') RESET r; Endif;
```

```
 If (RI 1:3= '001') NEXT Etat2; Endif;
```

```
 . . .)
```

Ces deux instructions sont remplacées par une seule. Cet état devient alors:

```
Etat (If (RI 1:3= '001') RESET r; NEXT Etat2; Endif;
```

```
 . . .)
```

Ce cas de figure peut provenir soit directement du concepteur, soit de l'extracteur des niveaux d'interprétation EXTRACT, qui éclate une instruction ayant plusieurs actions en plusieurs instructions de même conditions et ayant chacune une seule action.

### Règle2 Règle de l'inclusion:

Ici, il s'agit de vérifier si une condition est incluse dans une autre. En effet, si  $\text{Condi} \subset \text{Condj}$  (respectivement  $\text{Condj} \subset \text{Condi}$ ) et Actioni est égale à Actionj alors l'instruction dont la condition est Condj (resp Condi) est éliminée.

#### Exemple:

```
Etat (If ((RI 1:3= '001') • (B 0:1='1')) RESET r; Endif;
```

```

 If (RI 1:3= '001') RESET r; Endif;
 . . .)

```

Il est clair, dans cet exemple, que la condition (RI 1:3= '001') suffit pour mettre la variable de contrôle r à zéro. Donc, le CPC élimine la première instruction.

Dans le cas où Actioni n'est pas égale à Actionj, on élimine de Actionj (resp Actioni) tous les éléments appartenant à Actioni (resp Actionj), excepté les branchements.

#### Exemple:

```

Etat (If ((RI 1:3= '001') • (B 0:1='1')) RESET r; NEXT Etat2; Endif;
 If (RI 1:3= '001') RESET r; EXECUTE Point1; Endif;
 . . .)

```

Cet état se transforme en:

```

Etat (If ((RI 1:3= '001') • (B 0:1='1')) NEXT Etat2; Endif;
 If (RI 1:3= '001') RESET r; EXECUTE Point1; Endif;
 . . .)

```

#### Règle3 Règle d'exclusion d'un terme nul:

Si, dans une condition quelconque, une variable x et son complémentaire  $\neg x$  apparaissent, l'instruction dont la condition est en question est éliminée.

#### Exemple:

```

Etat (If ((RI 0:2= '01') • (RI 1:2= '01')) RESET r; NEXT Etat2; Endif;
 . . .)

```

Dans cette condition, le bit 1 du registre RI est testé à 0 et à 1, ce qui est toujours faux. Donc cette instruction est éliminée de l'état.

Durant cette phase d'optimisation, le CPC fait deux types de vérifications sémantiques. Il vérifie que:

# il n'y a pas de branchement à deux états différents de l'automate,

# il n'y a pas d'activation de deux automates différents de l'étage inférieur.

La première vérification est faite parce qu'on traite des automates séquentiels dans lesquels on ne peut pas être dans deux états différents au même instant. Cette vérification est faite pendant l'application des règles 1 et 2 d'optimisation. En effet, si on a égalité des conditions et que les

deux instructions possèdent chacune un branchement, le CPC vérifie que ces branchements se font au même état, autrement il signale une erreur au concepteur et élimine l'une des instructions.

De même, quand on a une inclusion et qu'il n'est pas possible de fusionner les deux instructions, le CPC fait la même vérification. Cependant, le CPC ne sait pas détecter cette erreur en dehors de ces deux cas. En effet, si le concepteur donne l'état suivant:

```
Etat (If (A 0:1='0') NEXT EtatI, SET Y, Endif,
 If (B 0:1='1') NEXT EtatJ, RESET Y, Endif,)
```

le CPC ne sait pas décider statiquement si ces deux conditions peuvent être vraies en même temps ou non. Donc c'est au concepteur d'essayer d'éviter ces cas.

Quant à la deuxième vérification, elle garantit la conformité du modèle architectural choisi dès le départ.

Cette vérification se fait également lors de l'application des règles 1 et 2. Pour l'étage le plus bas, cette vérification change d'aspect et permet, au lieu de détecter des erreurs, de regrouper les instructions exécutable en parallèle.

#### **Cas d'égalité:**

Quand les actions opératives des deux instructions comparées sont différentes, le CPC fusionne quand même les deux instructions conditionnelles en une seule dont l'action opérative est l'union des deux actions opératives de départ. Si les instructions comparées sont inconditionnelles, le CPC ne fait rien et suppose qu'elles sont fusionnées.

#### **Cas d'inclusion:**

Dans ce cas, le traitement est plus complexe. Supposons avoir comparé des instructions conditionnelles dont les conditions sont Cond1 et Cond2, et que Cond1 est incluse dans Cond2. Si les actions opératives de ces deux instructions sont différentes, le CPC ne se contente pas de fusionner les deux instructions mais doit aussi assurer que chaque action opérative ne s'exécute que lorsque la condition qui lui est associée est vraie. Ceci revient à exécuter l'action associée à Cond1 quand Cond1 est vraie et Cond2 ne l'est pas, et à exécuter les deux actions en parallèle quand Cond2 est vraie ( Cond1 le sera évidemment).



**Exemple:**

Etat1 ( If (A=1) • (B=0) A := X+Y; Endif;

    If (A=1) T:= U+Z; Endif;)

cet état est transformé en:

Etat1 ( If (A=1) • (B=0) • (A=0) A := X+Y; Endif;

    If (A=1) • (B=1 / A=0) T:= U+Z; Endif;

    If (A=1) • (B=0) A := X+Y; T:= U+Z; Endif;)

qui, après transformation et optimisation, devient:

Etat1 ( If (A=1) • (B=1) T:= U+Z; Endif;

    If (A=1) • (B=0) A := X+Y; T:= U+Z; Endif;)

Pour un état de N instructions notées ST<sub>i</sub> (i=1..N), l'algorithme d'optimisation est le suivant:

Pour i=1 à N-1 faire

debut

        appliquer la règle3 d'optimisation à ST<sub>i</sub>

Si ST<sub>i</sub> n'est pas éliminée alors

pour j=i+1 à N faire

debut

                    appliquer la règle3 à ST<sub>j</sub>

Si ST<sub>j</sub> n'est pas éliminée alors

debut

                            comparer Cond(ST<sub>i</sub>) et Cond(ST<sub>j</sub>)

                            appliquer la règle1 ou la règle2

fin

fin

fin.

Au pire des cas ( quand aucune instruction n'est éliminée), la complexité de l'algorithme est de l'ordre de  $N(N-1)/2$ .

**V-2- L'ajout du branchement implicite**

Deux problèmes essentiels nous ont conduit à rajouter ce traitement au CPC.

**Problème1:**

Pour qu'un automate soit synthétisable de façon déterministe et non ambiguë, sans tenir compte de la méthode de synthèse, il est impératif de s'assurer qu'il ne présente aucune condition de blocage dans son évolution

au cours du temps.

Un blocage peut survenir lorsqu'aucune transition en aval d'un état n'est franchissable à l'instant où toutes les variables de condition associées aux différentes transitions sont évaluées.

Pour qu'un automate ne se bloque pas, chaque état doit satisfaire au moins l'une des conditions suivantes:

- 1-il admet un branchement inconditionnel,
- 2-la somme des conditions pour lesquelles il a un branchement forme une tautologie.

```

Cmodule typique;
<E1> (If Cond NEXT E1 Endif;)
<E2> (NEXT E4 ...)
<E3> (SET A)
<E4> (If Cond NEXT E3 Endif;)
<E5> (If Cond1 NEXT E2 ; RESET A ; Endif;
 If Cond2 SET A Endif;)
<E6> (NEXT E1;)
END;

```

Figure 4.2: Description typique d'un Cmodule

Etant donné qu'on fait une interprétation procédurale de la description du circuit, pour un état donné, quand le branchement n'est pas spécifié explicitement, l'interprétation peut être faite de la façon suivante:

On se branche implicitement à l'état suivant dans la séquence syntaxique des états ou bien l'état est final. Le branchement peut être omis quand on a un branchement inconditionnel. En effet, dans ce cas il n'est pas indispensable de spécifier explicitement les branchements conditionnels. De même, si on a déjà spécifié, explicitement, des branchements conditionnels dont la somme des conditions forme une tautologie, il n'est plus nécessaire de spécifier le reste des branchements conditionnels explicitement.

On peut remarquer que si on a les conditions de blocage d'un automate

dans un état, l'interprétation ne peut être faite correctement.

Une description typique d'un automate en LDS est illustrée en figure 4.2. Le graphe de précedence correspondant à cette description est donné en figure 4.3.

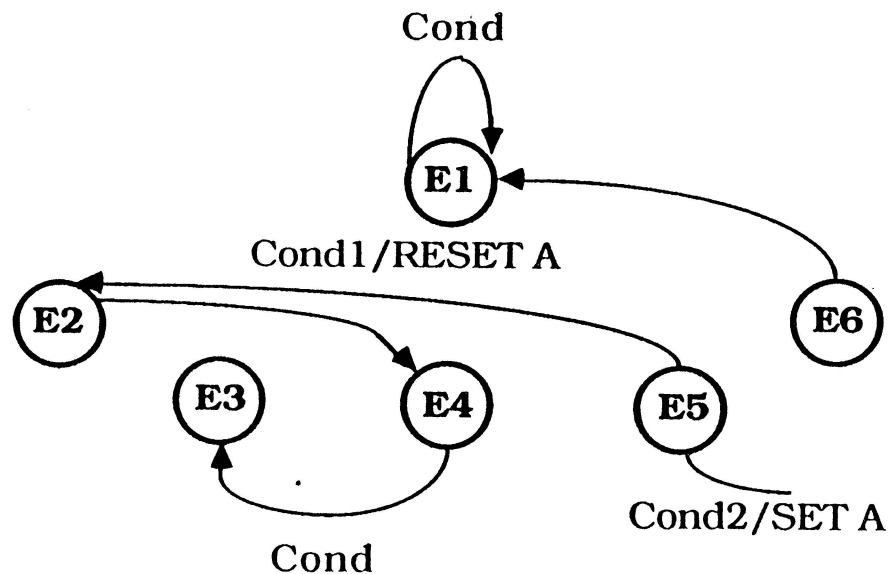


Figure 4.3: Graphe de l'automate de la figure 4.2

Du point de vue algorithmique, la description ne pose aucun problème. Par contre, dans le graphe de précedence, on voit bien que pour l'état E1, si la condition **Cond** n'est pas vérifiée, l'automate sera bloqué dans cet état. De même, l'automate se bloquera toujours à l'état E3. Pour éviter ce blocage, il faut se ramener au graphe de la figure 4.4.

Pour un état de N instructions on définit les entités suivantes:

- S1={ensemble d'instructions conditionnelles ayant un branchement},  
Card(S1)=C1,
- S2={instruction de branchement inconditionnel}, Card(S2)=C2,  $C2 \leq 1$ ,
- S3={ensemble d'instructions conditionnelles n'ayant pas de branchement},  
Card(S3)=C3,
- S4={ensemble d'instructions inconditionnelles différentes du branchement},  
Card(S4)=C4.  $C1+C2+C3+C4=N$ .

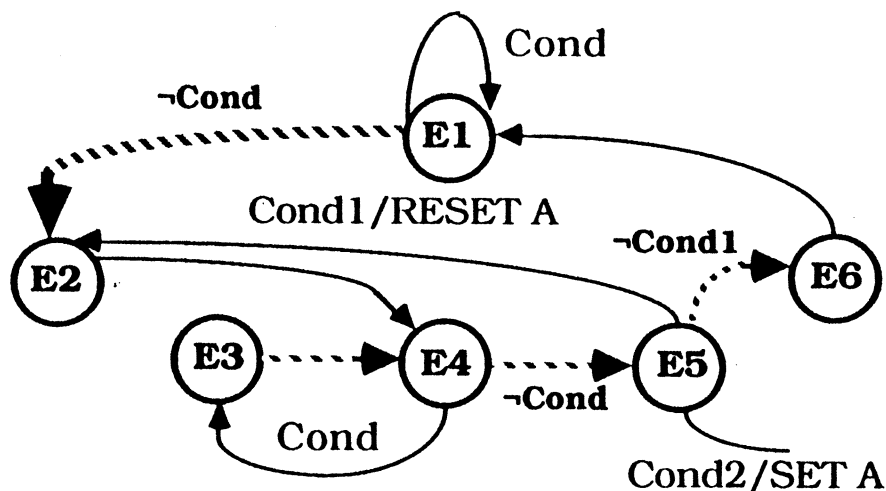


Figure 4.4: Automate non bloquant

Pour résoudre le problème1, le CPC distingue les cas suivants:

#### Cas1: (C2=0) et (C1≠0)

On n'a que des branchements conditionnels. Pour vérifier que l'automate ne se bloquera pas à cet état, le CPC vérifie si la somme des conditions de ces instructions conditionnelles forme une tautologie. Si c'est le cas, on est sûr que l'automate ne se bloquera jamais à cet état. Autrement, le CPC rajoute, sur demande du concepteur, des instructions conditionnelles de façon à obtenir la tautologie des conditions de transitions. Les actions associées aux instructions rajoutées sont des branchements à l'état suivant dans la séquence des états (NEXT état ou un EXIT si l'état est final).

Remarquons que l'ajout d'instructions pour obtenir une tautologie se fait à la demande du concepteur pour éviter l'éclatement inutile de l'étage. En effet, si le concepteur a vérifié, par simulation ou par un autre moyen, qu'il a toujours au moins une transition franchissable à un instant donné, le CPC suppose que la somme des conditions pour lesquelles on a un branchement forme une tautologie, par conséquent il ne rajoute pas d'instruction.

Une fois les instructions rajoutées, le CPC applique les règles d'optimisation à l'état courant.

#### Exemple:

```
Etat1 (If (A 0:2= '00') RESET r; NEXT Etat1; Endif;
 If (A 0:2= '10') SET r; NEXT Etat2; Endif;)
```

Comme la somme des conditions ne forme pas une tautologie, le CPC

rajoute une instruction dont la condition est:

# (A 0:2= '00') • # (A 0:2= '01')

En développant cette expression on obtient:

$((A\ 0:2 = 'x1') \bullet (A\ 0:2 = 'x1')) / ((A\ 0:2 = 'x1') \bullet (A\ 0:2 = '0x')) /$

$((A\ 0:2 = '1x') \bullet (A\ 0:2 = 'x1')) / ((A\ 0:2 = '1x') \bullet (A\ 0:2 = '0x'))$

Donc l'état résultant sera:

```
Etati (If (A 0:2= '00') RESET r; NEXT Etat1; Endif;
 If (A 0:2= '10') SET r; NEXT Etat2; Endif;
 If (A 0:2= 'x1') NEXT Etati+1; Endif;
 If ((A 0:2= 'x1') • (A 0:2= '0x')) NEXT Etati+1; Endif;
 If ((A 0:2= '1x') • (A 0:2= 'x1')) NEXT Etati+1; Endif;
 If ((A 0:2= '1x') • (A 0:2= '0x')) NEXT Etati+1; Endif;)
```

On suppose dans cet exemple que l'état n'est pas un état final, donc on rajoute des NEXT.

En appliquant la règle2 d'optimisation, on va pouvoir éliminer les instructions 4 et 5 par comparaison à l'instruction 3. Quant à l'instruction 6, elle est éliminée en appliquant la règle3 puisque le bit 1 de la variable A est testé à 0 et à 1. D'où la forme finale de cet état:

```
Etati (If (A 0:2= '00') RESET r; NEXT Etat1; Endif;
 If (A 0:2= '10') SET r; NEXT Etat2; Endif;
 If (A 0:2= 'x1') NEXT Etati+1; Endif;)
```

### Cas2: (C2=0) et (C1=0)

On n'a aucun branchement, ni conditionnel ni inconditionnel. Le CPC rajoute tout simplement un branchement inconditionnel (NEXT ou EXIT) à l'état courant.

### Exemple:

```
Etati (If A SET X; RESET Y; Endif;
 If B RESET X; SET Y; Endif;
 EXECUTE Point1;)
```

Cet état est transformé en:

```
Etati (If A SET X; RESET Y; Endif;
 If B RESET X; SET Y; Endif;
 EXECUTE Point1;
 NEXT Etati+1;
```

Après résolution de ce problème, un état de N instructions vérifie les contraintes suivantes:

- (1) La somme des conditions des instructions conditionnelles ayant un branchement est supposée former une tautologie. Cette supposition vient du fait que si le CPC n'a pas rajouté des instructions pour avoir la tautologie c'est que le concepteur est sûr d'avoir au moins une transition franchissable pour chaque état,
- (2) s'il y a un branchement inconditionnel, tous les branchements conditionnels lui sont identiques,
- (3) si l'état n'a pas de branchement conditionnel, il possède un branchement inconditionnel.

#### **Problème2:**

Au chapitre 3 (§II-3), on a vu qu'il y a deux façons de mémoriser l'état courant de l'automate pour la génération du séquençement interne. La première façon consiste à utiliser un registre. Or dans ce cas, le mécanisme de séquençement ne peut être obtenu qu'à partir d'une description où tous les branchements sont spécifiés explicitement. Dans notre cas, c'est cette façon de faire qui est adoptée pour l'implémentation de chaque étage de contrôle. Le CPC doit donc rajouter des branchements dans la table d'états de façon à obtenir un graphe complet de l'automate (voir figure 4.5).

En outre, pour pouvoir réaliser des optimisations topologiques (par ARTS) touchant plusieurs états à la fois, telles que l'élimination des états inutiles ou la fusion d'états successifs, il est impératif d'avoir une table d'états où chaque automate est décrit par un graphe complet.

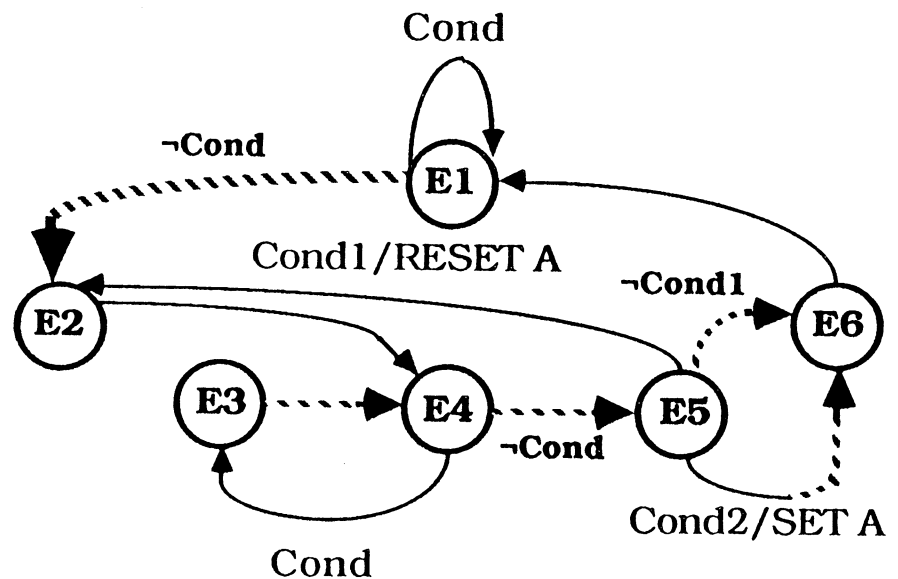


Figure 4.5: Graphe de précedence complet correspondant à la figure 4.3

Vu les contraintes vérifiées par la table d'états de chaque étage après résolution du problème1, le problème2 est essentiellement localisé au niveau des instructions conditionnelles n'ayant pas de branchement. Le CPC distingue deux cas:

**Cas1: (C2=1) et (C3≠0)**

On a un branchement inconditionnel et des instructions conditionnelles sans branchement. Ce cas peut provenir du traitement du problème1 (cas2). Pour être conforme à la contrainte (1) citée à la fin du traitement du problème1, le CPC commence par vérifier la tautologie des conditions des instructions n'ayant pas de branchement (appartiennent à S3). S'il n'y a pas tautologie, il rajoute des instructions dont les actions sont le branchement inconditionnel. Ce même branchement est inséré par la suite dans les actions des instructions conditionnelles de départ.

**Exemple:**

```
Etati (If A SET X; RESET Y; Endif;
 If B RESET X; SET Y; Endif;
 EXECUTE Point1;
 NEXT Etatj;
```

Cet état se transforme en:

```
Etati (If (A=1) SET X; RESET Y; NEXT Etatj; Endif;
 If (A=0) NEXT Etatj; Endif;
 If (B=1) RESET X; SET Y; NEXT Etatj; Endif;
 If (B=0) NEXT Etatj; Endif;
 EXECUTE Point1;
 NEXT Etatj;)

```

**Cas2: (C2=0) et (C3≠0) et (C1 ≠0)**

Dans ce cas on n'a pas de branchement inconditionnel, par contre on a des branchements conditionnels (C1 ≠ 0) et des instructions conditionnelles sans branchement (C3 ≠ 0). Pour résoudre le problème2, le CPC commence par la vérification de la tautologie des conditions des instructions de l'ensemble S3. Si la tautologie n'est pas vérifiée, il rajoute des instructions conditionnelles pour la vérifier. Les instructions rajoutées auront une action vide. On verra plus loin l'intérêt de cette vérification.

**Exemple:**

```
Etati (If (A=0) SET X; NEXT Etatj; Endif;
 If (A=1) RESET X; NEXT Etatk; Endif;
 If (B=0) EXECUTE Point1; Endif;)

```

Pour cet état, le CPC rajoute une instruction dont la condition sera (B=1). On aura alors:

```
Etati (If (A=0) SET X; NEXT Etatj; Endif;
 If (A=1) RESET X; NEXT Etatk; Endif;
 If (B=0) EXECUTE Point1; Endif;
 If (B=1) Endif;)

```

Après ce pré-traitement, le CPC combine chaque instruction de l'ensemble S1 avec celles de l'ensemble S3 selon l'algorithme suivant:



Pour chaque instruction ST1 de S1 faire

debut

Pour chaque instruction ST3 de S3 faire

debut

créer une nouvelle instruction conditionnelle ST,

$\text{Cond}(\text{ST}) \leftarrow \text{Cond}(\text{ST1}) \cdot \text{Cond}(\text{ST3}),$

$\text{Action}(\text{ST}) \leftarrow \text{Action}(\text{ST1}) \cup \text{Action}(\text{ST3}),$

remplacer ST3 par ST,

fin

éliminer ST1 de l'état courant,

fin

Donc les instructions de S1 et S3 sont remplacées par  $C1 \cdot C3$  nouvelles instructions.

Si on prend l'exemple précédent, en appliquant cet algorithme on obtient l'état suivant:

```
Etati (If ((A=0) • (B=0)) EXECUTE Point1; SET X; NEXT Etatj; Endif;
 If ((A=0) • (B=1)) SET X; NEXT Etatj; Endif;
 If ((A=1) • (B=0)) EXECUTE Point1; RESET X; NEXT Etatk; Endif;
 If ((A=1) • (B=1)) RESET X; NEXT Etatk; Endif;)
```

L'intérêt de vérifier la tautologie des conditions des instructions appartenant à S3 est d'éviter problème du blocage de l'automate. En effet, pour cet exemple, si on n'avait pas rajouté l'instruction dont la condition est  $(B=1)$ , les instructions dont les conditions sont  $(A=0).(B=1)$  et  $(A=1).(B=1)$  n'auraient pas existé après l'application de l'algorithme. Par conséquent, si  $(B=0)$  est vraie on se branche vers l'état j ou l'état k selon la valeur de A, sinon l'automate se bloquera dans cet état.

A la fin de ce traitement, la table d'états vérifie les conditions citées au début du paragraphe IV.

Dans la description d'un automate, chaque instruction conditionnelle correspond à un arc entre deux états dans le graphe de précedence (c'est une transition conditionnelle), et toutes les instructions inconditionnelles correspondent à un seul arc (c'est une transition inconditionnelle). On remarque qu'une description de ce type correspond à un automate de Mealy.

Pour réaliser un automate de Mealy par un PLA, chaque arc de l'automate va correspondre à un monôme du PLA. Etant donné que dans les automates décrits par la table d'états on a toujours au moins une transition franchissable, le CPC applique à la table d'états une autre étape de minimisation qui consiste à réduire le nombre de monômes du PLA qui réalisera l'étage de contrôle courant. Le principe de ce traitement repose sur l'idée d'insérer toutes les instructions inconditionnelles dans l'action de chaque instruction conditionnelle.

**Exemple:**

```
étati (If (A=0) RESET x; NEXT etatj; Endif;
 If (A=1) SET x; NEXT etatk; Endif;
 EXECUTE point1;
 RESET y;)
```

L'implémentation de cet état nécessite trois monômes dans le PLA réalisant l'étage de contrôle en question. Or les deux instructions inconditionnelles sont exécutées dans tous les cas. Comme on a toujours une condition de transition qui est vraie à un instant donné, on peut insérer les instructions inconditionnelles dans les actions des instructions conditionnelles. On obtient alors:

```
étati (If (A=0) RESET x; NEXT etatj; EXECUTE point1; RESET y;Endif;
 If (A=1) SET x; NEXT etatk; EXECUTE point1; RESET y; Endif;)
```

Ce type d'optimisation permet de gagner au plus un monôme par état de chaque automate, mais permet en même temps d'augmenter le taux de remplissage de la matrice OU du PLA. La figure 4.6 illustre les tables d'états correspondant aux Cmodules de la figure 2.7.

```

? ----- Module mini_1
CMODULE mini_1 ;
< mini_start_1 >
(EXECUTE mini_0 (mini_start_1) ; EXIT ;)
< mini_newinstr_1 >
(EXECUTE mini_0 (fetch_send_1) ; EXIT ;)
< bru_g194_1 >
(r 0:2 := 2 ; EXECUTE mini_0 (fetch_send_1) ;
NEXT bru_g194_3 ;)
< bru_g194_3 >
(EXECUTE mini_0 (bru_g194_3) ; EXIT ;)
< bneg_g195_1 >
(r 0:2 := 2 ; EXECUTE mini_0 (fetch_send_1) ;
NEXT bneg_g195_3 ;)
< bneg_g195_3 >
(EXECUTE mini_0 (bneg_g195_3) ; EXIT ;)
< sta_g191_1 >
(r 0:2 := 2 ; EXECUTE mini_0 (fetch_send_1) ;
NEXT sta_g191_3 ;)
< sta_g191_3 >
(read_write 0:1 := 0 ; EXECUTE mini_0 (sta_g191_3) ;
NEXT sta_g191_5 ;)
< sta_g191_5 >
(adstrobe 0:1 := 1 ; dstrobe 0:1 := 1 ;
EXECUTE mini_0 (sta_g191_5) ; NEXT sta_ack_1 ;)
< sta_ack_1 >
(IF (dack 0:1 = (BIN'1))
adstrobe 0:1 := 0 ; dstrobe 0:1 := 0 ;
EXECUTE mini_0 (sta_ack_1) ; EXIT ; ENDIF ;
IF (dack 0:1 = (0))
NEXT sta_ack_1 ;
EXECUTE mini_0 (sta_ack_1) ; ENDIF ;)
< lda_g193_1 >
(r 0:2 := 1 ; EXECUTE mini_0 (fetch_send_1) ; EXIT ;)
< ada_g192_1 >
(r 0:2 := 2 ; EXECUTE mini_0 (fetch_send_1) ;
NEXT ada_g192_3 ;)
< ada_g192_3 >
(EXECUTE mini_0 (ada_g192_3) ; EXIT ;)
END ;
? ----- End of Module mini_1

? ----- Module mini_2
CMODULE mini_2 ;
< mini_rest_1 >
(IF (restart 0:1 = (0)) NEXT mini_rest_1 ; ENDIF ;
IF (restart 0:1 = (BIN'1)) NEXT mini_start_1 ; ENDIF ;)
< mini_start_1 >
(EXECUTE mini_1 (mini_start_1) ; NEXT mini_newinstr_1 ;)
< mini_newinstr_1 >
(r 0:2 := 0 ; EXECUTE mini_1 (mini_newinstr_1) ;
NEXT mini_decode_1 ;)
< mini_decode_1 >
(IF (restart 0:1 = (0)) . (tr 0:4 = (BIN'0110)))
EXECUTE mini_1 (bru_g194_1) ;
NEXT mini_rest_1 ; ENDIF ;
IF (restart 0:1 = (0)) . (tr 0:4 = (BIN'0101)))
EXECUTE mini_1 (bneg_g195_1) ;
NEXT mini_rest_1 ; ENDIF ;
IF (restart 0:1 = (0)) . (tr 0:4 = (BIN'0010)))
EXECUTE mini_1 (sta_g191_1) ;
NEXT mini_rest_1 ; ENDIF ;
IF (restart 0:1 = (0)) . (tr 0:4 = (BIN'0001)))
EXECUTE mini_1 (lda_g193_1) ;
NEXT mini_rest_1 ; ENDIF ;
IF (restart 0:1 = (0)) . (tr 0:4 = (BIN'0000)))
EXECUTE mini_1 (ada_g192_1) ;
NEXT mini_rest_1 ; ENDIF ;
IF (restart 0:1 = (BIN'1)) . (tr 0:4 = (BIN'0110)))
EXECUTE mini_1 (bru_g194_1) ;
NEXT mini_newinstr_1 ; ENDIF ;
IF (restart 0:1 = (BIN'1)) . (tr 0:4 = (BIN'0101)))
EXECUTE mini_1 (bneg_g195_1) ;
NEXT mini_newinstr_1 ; ENDIF ;
IF (restart 0:1 = (BIN'1)) . (tr 0:4 = (BIN'0010)))
EXECUTE mini_1 (sta_g191_1) ;
NEXT mini_newinstr_1 ; ENDIF ;
IF (restart 0:1 = (BIN'1)) . (tr 0:4 = (BIN'0001)))
EXECUTE mini_1 (lda_g193_1) ;
NEXT mini_newinstr_1 ; ENDIF ;
IF (restart 0:1 = (BIN'1)) . (tr 0:4 = (BIN'0000)))
EXECUTE mini_1 (ada_g192_1) ;
NEXT mini_newinstr_1 ; ENDIF ;)
END ;
? ----- End of Module mini_2

? ----- Module mini_0
CMODULE mini_0 ;
< mini_start_1 >
(pc 0:8 := 0 ; EXIT ;)
< bru_g194_3 >
(pc 0:8 := b 0:8 ; EXIT ;)
< bneg_g195_3 >
(IF (a 7:1 = (BIN'1)) pc 0:8 := b 0:8 ; EXIT ; ENDIF ;
IF (a 7:1 = (BIN'0)) EXIT ; ENDIF ;)
< sta_g191_3 >
(adbus 0:8 := b 0:8 ; dbus 0:8 := a 0:8 ; EXIT ;)
< sta_g191_5 >
(adbus 0:8 := b 0:8 ; dbus 0:8 := a 0:8 ; EXIT ;)
< sta_ack_1 >
(IF (dack 0:1 = (0)) adbus 0:8 := b 0:8 ;
dbus 0:8 := a 0:8 ; EXIT ; ENDIF ;
IF (dack 0:1 = (BIN'1)) EXIT ; ENDIF ;)
< ada_g192_3 >
(a 0:8 := a 0:8 + (b 0:8) ; EXIT ;)
< fetch_send_1 >
(adbus 0:8 := pc 0:8 ; read_write 0:1 := 1 ; NEXT fetch_g190_1 ;)
< fetch_g190_1 >
(adbus 0:8 := pc 0:8 ; adstrobe 0:1 := 1 ; NEXT fetch_receive_1 ;)
< fetch_receive_1 >
(IF (r 0:2 = (2)) . (dack 0:1 = (BIN'1)))
b 0:8 := dbus 0:8 ; pc 0:8 := pc 0:8 + (1) ; EXIT ; ENDIF ;
IF (r 0:2 = (1)) . (dack 0:1 = (BIN'1)))
a 0:8 := dbus 0:8 ; pc 0:8 := pc 0:8 + (1) ; EXIT ; ENDIF ;
IF (r 0:2 = (0)) . (dack 0:1 = (BIN'1)))
tr 0:8 := dbus 0:8 ; pc 0:8 := pc 0:8 + (1) ; EXIT ; ENDIF ;
IF (dack 0:1 = (BIN'1))
pc 0:8 := pc 0:8 + (1) ; adstrobe 0:1 := 0 ; EXIT ; ENDIF ;
IF (dack 0:1 = (0))
adbus 0:8 := pc 0:8 ; NEXT fetch_receive_1 ; ENDIF ;)
END ; ? ----- End of Module mini_0

```

Figure 4.6: Tables d'états correspondant à la figure 2.7

### V-3-Vérification de la tautologie

Pour vérifier la tautologie d'un ensemble de conditions, le CPC utilise le principe des tableaux de Karnaugh.

En partant des vecteurs de conditions associés aux différentes conditions, il crée et remplit un tableau de Karnaugh. Pour une fonction logique, une case du tableau de Karnaugh prend la valeur "1" si le monôme correspondant à la case figure dans l'expression de la fonction. Une fonction est une tautologie si toutes les cases du tableau de Karnaugh obtenu à partir de son expression prennent la valeur "1".

#### Exemple:

Supposons avoir quatre variables booléennes A, B, C et D. Pour vérifier si la somme des conditions suivantes  $((A=0), (B=1 \text{ and } C=0))$  forme une tautologie, le CPC considère les vecteurs de conditions associés à ces conditions (ces vecteurs sont  $[0 \ x \ x \ x]$  et  $[x \ 1 \ 0 \ x]$ ) et remplit un tableau de Karnaugh qui aura quatre lignes et quatre colonnes.

Pour le remplissage du tableau, le CPC procède de la façon suivante pour chaque vecteur de condition:

- 1) il considère la première moitié des variables (A, B),
- 2) il cherche toutes les valeurs prises par ces variables d'après le vecteur de conditions en question, si on considère le premier vecteur de condition, la valeur des variables A et B est "0x" ce qui correspond aux valeurs (00, 01),
- 3) il fait la même chose pour le reste des variables. Dans cet exemple, les variables CD prennent la valeur "xx" ce qui donnent les valeurs binaires suivantes (00, 01, 10, 11).

On remarque que chaque valeur de AB correspondra à une ligne du tableau et chaque valeur de CD correspondra à une colonne du tableau.

- 4) il met "1" dans les cases qui correspondent aux lignes et colonnes indicées par les valeurs de AB et CD. Dans ce cas on a ((00,00) (00,01) (00,10) (00,11) (01,00) (01,01) (01,10) (01,11)).

Pour le deuxième vecteur de condition, la valeur "x1" de AB donne les lignes 01 et 11 alors que la valeur "0x" de CD donne les colonnes 01 et 00 d'où le tableau donné en table1.

| AB \ CD | CD |    |    |    |
|---------|----|----|----|----|
|         | 00 | 01 | 10 | 11 |
| 00      | 1  | 1  | 1  | 1  |
| 01      | 1  | 1  | 1  | 1  |
| 10      |    |    |    |    |
| 11      | 1  | 1  |    |    |

Table1: Tableau de Karnaugh

D'après ce tableau on peut voir tout de suite que la somme des deux conditions données ne forme pas une tautologie. Toujours d'après ce tableau, pour avoir la tautologie, on peut rajouter les produits  $((A=1 \text{ and } B=0), (A=1 \text{ and } C=1))$ . Or le CPC rajoute la condition  $(\text{Not } (A=0 \text{ or } (B=1 \text{ and } C=0)))$  ce qui équivaut à  $(A=1 \text{ and } (B=0 \text{ or } C=1))$ . On retrouve les produits rajoutés d'après le tableau de Karnaugh.

Etant donné que le tableau de Karnaugh n'est pas utilisé par le CPC pour la simplification des fonctions logiques, les indices des lignes et des colonnes sont pris dans l'ordre croissant sans assurer que deux indices ne diffèrent que par une seule position.

Le CPC n'impose au concepteur aucune restriction sur le nombre de variables de condition qu'il utilise. Dans le cas des circuits complexes, ce nombre dépasse généralement 10. Dans le cas du microprocesseur 6502, les deux étages les plus hauts ont utilisé 14 variables chacun alors que l'étage le plus bas en a utilisé 30.

A titre indicatif, pour 50 variables de condition et en supposant que le traitement de chaque case du tableau de Karnaugh prend 1  $\mu$ s, pour vérifier la tautologie, le temps de calcul peut atteindre 30 ans.

Quand le nombre de variables utilisées dans les conditions est important, la vérification de la tautologie de cette façon pose un problème à deux niveaux:

- 1) au niveau espace mémoire, en effet pour N variables il faut un tableau de  $2^N$  cases,
- 2) au niveau temps d'exécution, le remplissage du tableau et la vérification de la tautologie demandent beaucoup de temps.

Le problème du temps d'exécution ne peut être surmonté par le CPC et

il faut chercher d'autres algorithmes ou heuristiques qui permettent la vérification de la tautologie sans nécessité de test de toutes les combinaisons possibles des variables de condition.

SASAO [SAS85] propose un dispositif matériel (figure 4.7) composé de trois éléments:

- 1-un générateur de monômes,
- 2-un ensemble de latches (un pour chaque monôme),
- 3-des portes AND.

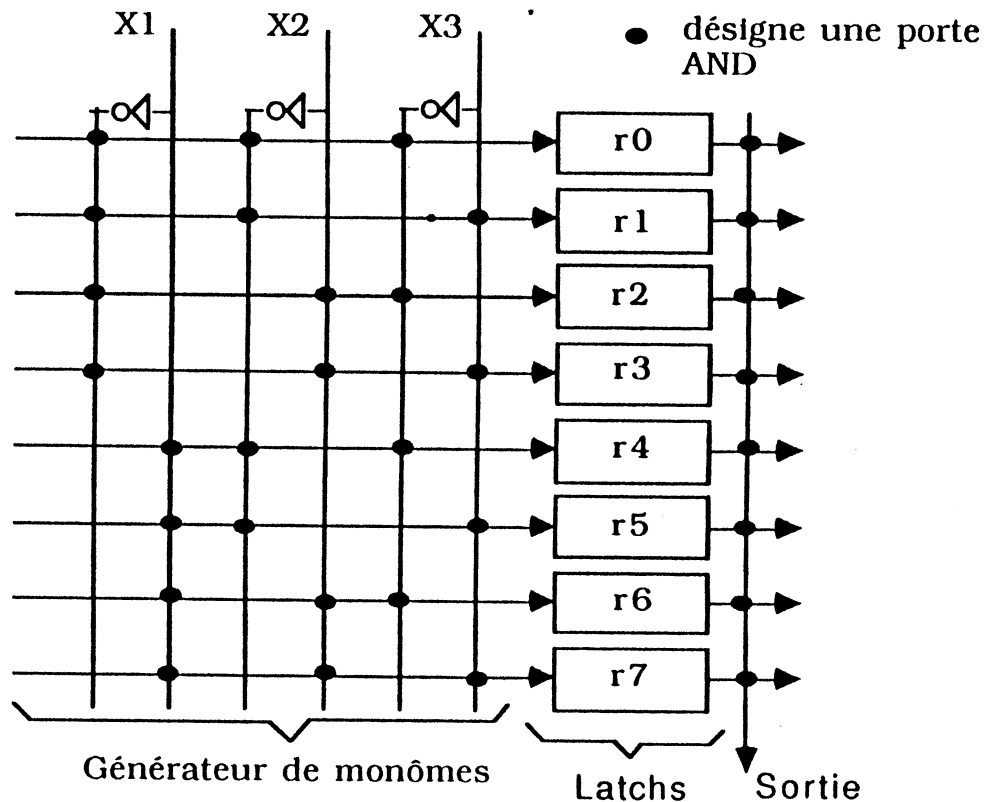


Figure 4.7: Vérificateur matériel de tautologie [SAS85]

Le générateur de monômes possède  $N$  entrées et  $2^N$  sorties. Chaque sortie correspond à un monôme de la fonction dont on veut vérifier la tautologie. Chaque fois qu'on applique un produit (ou un cube), le monôme qui lui correspond passe à "1". Cette valeur est mémorisée dans le latch correspondant. Pour une fonction (qui est la somme de produits de variables booléennes), on applique tous les produits séquentiellement dans un ordre quelconque. A la fin, si tous les latches sont à 1, la sortie (qui est le produit des valeurs des latches) sera à 1. Dans ce cas, la fonction est une tautologie, sinon elle ne l'est pas.

Ce dispositif est certes efficace en temps d'exécution mais pose un problème d'interfaçage avec les compilateurs de partie contrôle.

Quant au problème d'espace mémoire, il peut être surmonté par le CPC lui-même. En effet, au lieu de construire tout le tableau de Karnaugh à la fois, le CPC construit des tranches de tableau ( des matrices  $2 \times 2$ ). Si la tautologie locale à une tranche de tableau est vérifiée, on passe à la tranche suivante. L'algorithme est le suivant:

séparer l'ensemble des variables de condition en deux ensembles LI et CO,

TOTO <-- Vrai,

tantqu'il reste des variables non traitées dans LI et TOTO faire

    prendre une variable VI de l'ensemble LI,

    tantqu'il reste des variables non traitées dans CO et TOTO faire

        prendre une variable Vc de CO et créer une matrice  $2 \times 2$ ,

        pour chaque vecteur de condition faire

            déterminer les indices des lignes et des colonnes qui correspondent aux valeurs de VI et Vc à partir du vecteur de condition courant,

        finpour

        mettre 1 dans les cases qui correspondent aux lignes et colonnes déterminées,

        TOTO <-- (il y a 1 partout dans la matrice)

    fintantque

fintantque

Avec cet algorithme, la vérification de la tautologie ne nécessite pas beaucoup d'espace mémoire. Le temps d'exécution peut être très court si on n'a pas la tautologie. Par contre, dans le cas contraire, on perd en temps d'exécution.

Dans l'exemple précédent, et en considérant les vecteurs de condition pour lesquels on n'a pas la tautologie, on va prendre la première variable A et la troisième variable C. Pour le premier vecteur de condition, la valeur de A est 0, ce qui donne la ligne 0 de la matrice et la valeur de la variable C donne les colonnes 0 et 1, d'où on aura 1 aux cases (0,0) et (0,1). La valeur de A dans le deuxième vecteur de condition donne les lignes 0 et 1 et la valeur de C donne la colonne 0. Donc on aura 1 dans les cases (0,0) et

(1,0). Enfin, on peut voir que la tautologie locale n'est pas vérifiée, puisque la case (1,1) contient un zéro, et l'algorithme s'arrête à ce stade.

## **VI- Génération de la description de la POP**

Le Cmodule correspondant à l'étage le plus bas est le seul ayant des instructions exécutables par la partie opérative. Donc à partir de ce Cmodule, le CPC génère la description de la POP permettant au compilateur APOLLON de générer le layout de celle-ci. Une telle description est composée de deux parties: une partie déclaration et une partie action.

### **VI-1: La partie déclaration:**

Elle consiste à informer APOLLON des ressources qui doivent être implémentées dans la partie opérative. Cette partie fournit:

- a) la liste des opérateurs qui doivent figurer dans la POP,
- b) la liste des registres implémentés dans la POP,
- c) les bits des registres accessibles à la partie contrôle. Ces bits constituent le bus de compte rendu qui sera généré par APOLLON,
- d) les constantes utilisées dans la description et qui doivent donc figurer dans la POP,
- e) le nombre maximum de sous-parties opératives,
- f) le nombre de bits des registres de données du circuit à concevoir,
- g) et le nom du circuit.

### **VI-2- La partie action**

C'est la description proprement dite de la POP. Elle correspond au flux de données de la partie opérative et est composée d'une suite d'étapes. Une étape est un ensemble non vide d'instructions exécutables en parallèle (les instructions sont séparées par "/"). A partir de cette description, APOLLON génère le layout de la partie opérative et les micro-commandes associées à chaque étape de la description. On verra au paragraphe VII comment ces micro-commandes seront traitées par le CPC.

Les instructions à exécuter en parallèle sont regroupées en étapes durant l'optimisation de la table d'états de l'étage le plus bas. Actuellement, le regroupement se fait dans deux situations (voir § V-1):

- 1-quand on a égalité des conditions de deux instructions,
- 2-quand on a inclusion des deux conditions.



Comme cela a été signalé au paragraphe V-1, en dehors de ces deux cas, le CPC ne peut pas décider statiquement si on peut exécuter deux actions opératives en parallèle ou non.

**Exemple:**

Soit l'état:

```
Etat (. . .
 If (A=1) X:=Y; NEXT Etatj; Endif;
 If (B=1) Z:=W; NEXT Etatj; Endif;
 . . .)
```

Si on est sûr que les deux conditions ne sont jamais vraies en même temps alors les deux affectations ne doivent jamais s'exécuter en parallèle. Donc le CPC n'a pas à regrouper ces actions. Autrement, il faut faire le regroupement tout en gardant la sémantique de la description. Les deux instructions se transforment en:

```
Etat (. . .
 If (A=1)• (B=0) X:=Y; NEXT Etatj; Endif;
 If (A=1)• (B=1) X:=Y; Z:=W; NEXT Etatj; Endif;
 If (B=1)• (A=0) Z:=W; NEXT Etatj; Endif;
 . . .)
```

Dans le cas général, si les deux variables A et B sont structurées et de tailles respectives N et M, les deux instructions sont remplacées par N+M+1 instructions.

Comme on n'est pas sûr que les conditions soient vraies en même temps, on risque d'éclater, inutilement, la table d'états de l'étage de contrôle le plus bas.

Dans la version actuelle du CPC, on garde les instructions telles quelles. Cette option risque de poser un problème dans le cas où les conditions sont vraies simultanément puisque la POP va recevoir deux micro-commandes différentes à la fois. Ce problème ne peut être surmonté que si les microcommandes générées par APOLLON pour chaque action sont compatibles, ce qui n'est pas forcément vrai.

La figure 4.8 donne un exemple de description de la partie opérative.

```

operators : + ;
registers : a b ir pc ;
Obuffers : adbus<0:8> ;
IObuffers : dtbus<0:8> ;
control accessible registers : ir<0:3> a<7:7> ;
constants : 0 1 ;
maximum subsection number : 3 ;
bit number : 8 ;
name : mini ;

pc <- 0 ;
pc <- b ;
pc <- b ;
adbus <- b / dtbus <- a ;
adbus <- b / dtbus <- a ;
adbus <- b / dtbus <- a ;
a <- a + b ;
adbus <- pc ;
adbus <- pc ;
b <- dtbus / pc <- pc + 1 ;
a <- dtbus / pc <- pc + 1 ;
ir <- dtbus / pc <- pc + 1 ;
pc <- pc + 1 ;
adbus <- pc .

```

Figure 4.8: Description de la POP

## VII- Codage des tables d'états

Le codage consiste à générer la description logique de chaque PLA réalisant un étage de contrôle. Pour chaque automate d'un étage donné, le codage concerne:

- 1) les états de l'automate,
- 2) les conditions de transition associées à chaque état,
- 3) les actions destinées à l'étage inférieur (EXECUTE),
- 4) les commandes de manipulation des variables de contrôle (SET et RESET),
- 5) les branchements (NEXT ou EXIT),
- 6) les commandes de synchronisation des différents étages de contrôle.

Comme un étage peut être composé de plusieurs automates indépendants, le code d'un état se compose de deux champs:

a- le numéro de l'automate dans l'étage: ce champ correspond aux commandes émanant de l'étage supérieur,

b- le numéro de l'état dans l'automate.

Pour chaque étage de contrôle, la matrice ET du PLA reçoit donc en entrée trois champs (figure 4.1):

- 1- les commandes émanant de l'étage supérieur (INST<sub>i</sub>). Pour l'étage le plus haut, ce champ est ôté,

- 2- les entrées du séquençement interne (l'état courant de l'automate),
- 3- les fils de conditions. Ces fils correspondent aux bus de contrôle et de compte rendu.

En sortie, chaque PLA génère:

- a- les commandes destinées à l'étage inférieur ( $INST_{i-1}$ ),
- b- les sorties de séquençement interne,
- c- les commandes de manipulation des variables de contrôle (deux fils pour chaque variable),
- d- les fils de synchronisation.

Dans la version actuelle du CPC, on utilise deux variables de synchronisation. La première est mise à "1" quand on a une instruction EXECUTE (ou une instruction opérative pour l'étage le plus bas). Elle est mise à "0" dans le cas contraire, ce qui correspond à l'instruction NOP. Quant à la deuxième, elle est mise à "1" quand on a un NEXT comme branchement et mise à "0" quand le branchement est un EXIT (état final de l'automate).

#### **VII-1- Choix du code pour chaque champ**

Dans ce paragraphe, le choix du code pour chaque champ du PLA implémentant un étage de contrôle sera analysé.

1) Champs de séquençement interne et de commandes (destinées à ou émanant de l'étage voisin): On a vu au chapitre 3 qu'il est difficile, voire impossible, de trouver un code qui soit le meilleur dans tous les cas. Le code CBP, qui minimise le nombre d'entrées du PLA, a été choisi. La minimisation du nombre de monômes va dépendre des automates codés. Si un étage comprend  $2^N$  automates dont le plus grand possède  $2^K$  états, on aura besoin donc de N fils pour le codage des points d'entrée et de K fils pour le codage du séquençement interne.

2) Champ des conditions de transition: A chaque variable booléenne correspond un fil en entrée du PLA. Si on a T variables de contrôle et/ou de compte rendu, on aura donc T entrées de condition dans le PLA.

3) Champ des commandes de manipulation des variables de contrôle: A

chaque variable de contrôle modifiée par l'étage, le CPC fait correspondre deux fils en sortie du PLA (le SET et le RESET). Si un étage modifie  $V$  variables de contrôle, le PLA aura donc  $2V$  sorties de commande.

Le PLA aura  $(N+K+T)$  entrées (sans compter les entrées complémentées) et  $(K+2V+N'+4)$  sorties où  $N'$  correspond au nombre de fils pour les commandes destinées à l'étage inférieur et 4 correspond au fils de synchronisation.

### VII-2- L'assignation des états

Ce problème a été évoqué dans le cas général des compilateurs de parties contrôle. Pour le CPC, il se pose à deux niveaux puisqu'un étage de contrôle peut avoir plusieurs automates indépendants. Le problème d'assignation des états de chaque automate et celui de l'assignation des points d'entrée de l'étage en cours se posent donc au CPC.

L'assignation des états est directe pour le CPC, c'est à dire qu'il considère la liste des étiquettes et il donne un numéro à chacune. Ce numéro commence à 0 et dépend de l'ordre des états dans l'automate.

Le CPC n'applique aucun algorithme ou heuristique pour l'assignation des états du moment que les automates ne possèdent pas beaucoup d'états. A titre indicatif, la description du microprocesseur 6502 [REI86] a été faite avec 67 automates dont 34 n'avaient qu'un seul état, 17 n'en avaient que 2 et 6 n'en avaient que 3.

Quant à l'assignation des points d'entrée, elle pourrait contribuer à réduire le nombre de monômes du PLA lors de l'optimisation faite par ESPRESSO [RUD85]. Cependant la question s'impose des critères à considérer pour affecter un code à chaque point d'entrée se pose.

En fait, pour l'assignation des états d'un même automate on est guidé par le séquençement de l'automate, donc on connaît les états suivants de chaque état; par contre, pour l'assignation des points d'entrée, cette notion de séquençement n'est pas valable et on doit trouver une autre notion pour mieux affecter les codes aux points d'entrée.

Le CPC ne résout pas ce problème, mais contourne la difficulté. En effet, on va considérer le champ correspondant aux points d'entrée comme faisant partie des conditions de transition. La solution revient alors à donner un code à chaque point d'entrée de manière à permettre de fusionner le maximum de monômes du PLA. Donc, on aura une autre phase

d'optimisation logique appliquée à chaque étage de contrôle. Par conséquent, on aura en tout trois phases d'optimisation logique:

(1) Une optimisation locale à chaque état des différents automates composant l'étage de contrôle. Cette phase est appliquée directement à la table d'états (§ V-1).

(2) Une optimisation globale à l'étage de contrôle, elle est appliquée aussi à la table d'états (§ VIII).

(3) Une optimisation, faite par ESPRESSO, appliquée à la description logique du PLA (§ VIII).

En considérant le champ des points d'entrée comme conditions de transition, l'optimisation locale appliquée par le CPC peut être généralisée à tout l'étage de contrôle.

Solent deux états  $E_1$  et  $e_1$  appartenant à deux automates différents du même étage de contrôle:

$E_1$ ( If (A=0) RESET x; EXECUTE point1; NEXT  $E_1$ ; Endif;

If (A=1) SET x; EXECUTE point1; NEXT  $E_2$ ; Endif;)

$e_1$ ( If (A=0) RESET x; EXECUTE point1; NEXT  $e_1$ ; Endif;

If (A=1) SET x; EXECUTE point1; NEXT  $e_2$ ; Endif;)

On peut remarquer tout de suite que ces états ne diffèrent que par les étiquettes de branchement. Donc ils sont presque identiques. A priori, il serait possible de fusionner ces deux états si on donnait deux codes compatibles aux deux points d'entrée auxquels appartiennent ces états.

Supposons que le point d'entrée du premier automate ait le code 0010 et que les deux états en question aient le même numéro dans leurs automates respectifs (on va prendre  $\text{code}(E_1)=\text{code}(e_1)=01$ ).

Le codage de l'état  $E_1$  va donner les deux monômes suivants:

0010 01 0 ACTION1  $E_1$  (1)

0010 01 1 ACTION2  $E_2$  (2)

avec ACTION1= ( RESET x; EXECUTE point1 )

ACTION2= ( SET x; EXECUTE point1)

Le codage de l'état  $e_1$  va donner deux monômes (3) et (4) qui ne diffèrent des monômes (1) et (2) que par le premier champ. Donc pour pouvoir fusionner les monômes (1) et (3), et (2) et (4) il faut donner au

point d'entrée auquel appartient  $e_1$  un code qui ne diffère de celui du premier point d'entrée que d'une position (on prend par exemple le code 0011) et on aura:

0011 01 0 ACTION1  $e_1$  (3)

0011 01 1 ACTION2  $e_2$  (4)

La fusion va aboutir aux deux monômes suivants:

001x 01 0 ACTION1  $E_1$  (5)

001x 01 1 ACTION2  $E_2$  (6)

Cette fusion a été possible parce qu'on a supposé que les états ( $E_1$ ,  $e_1$ ), ( $E_2$ ,  $e_2$ ) ont le même code.

En conclusion, pour que deux monômes de deux états appartenant à deux automates différents d'un même étage de contrôle puissent être fusionnés, il faut qu'il y ait compatibilité:

(1) des conditions de transition (égalité ou inclusion),

(2) des actions (les mêmes actions),

(3) des branchements ( les états de branchement ont le même code),

(4) des états ( les deux états doivent avoir le même code ou des codes qui ne diffèrent que d'une position).

Si ces quatre conditions étaient vérifiées, on pourrait affecter aux points d'entrée des deux automates deux codes compatibles. On entend par codes compatibles, deux codes qui ne diffèrent que d'une position.

L'algorithme général peut se résumer en:

debut

    affecter le code 1 au premier point d'entrée,

    tantqu'il reste des points d'entrée non consultés faire

        prendre un point d'entrée,

        pour chaque point d'entrée non consulté faire

            vérifier la compatibilité de fusion des différents états,

            s'il y a possibilité de fusion, affecter le code de GRAY suivant au point d'entrée courant,

            faire la fusion nécessaire,

        finpour

    fintantque

    s'il reste des points d'entrée non codés, leur donner un code aléatoire.

fin

Cet algorithme n'est pas encore implémenté dans le CPC car les conditions de compatibilité sont difficilement réalisables.

Donc pour la présente version du CPC, l'assignation des points d'entrée est aussi directe. Le code dépend de l'ordre d'activation des automates par l'étage supérieur.

Etant donné qu'une description peut comporter plusieurs classes de Cmodules (voir § II), leur codage ne sera pas parfaitement identique.

Pour une description hiérarchique d'un circuit, la compilation commence par l'étage le plus haut et se termine par l'étage le plus bas. Cette stratégie a été choisie car en compilant un étage on peut collecter les points d'entrée de l'étage inférieur. De cette façon, la compilation de chaque étage ne nécessite qu'un seul passage du compilateur.

#### 1- Codage du Cmodule de la classe1: (l'étage le plus haut)

Ce Cmodule est composé d'un seul automate, donc il comprend un seul point d'entrée. Dans le PLA réalisant cet étage, le champ correspondant aux commandes émanant de l'étage supérieur est vide.

a. Codage de l'état courant: Lors de la génération de la table d'états, le CPC détermine la liste d'états de l'automate. Chaque étiquette est remplacée par son code (  $\text{code}(\text{etat}_i) = \log_2(i)$  ) où  $i$  est l'ordre de l'état dans l'automate.

b. Codage des conditions: On a vu aussi que pendant le recensement, le CPC génère la liste des bits des registres testés par l'étage de contrôle et détermine leur ordre dans le vecteur de condition. Le codage consiste à générer un vecteur où chaque position prend la valeur du bit qui lui correspond.

#### **Exemple:**

Supposons avoir les bits (0 1 3) du registre RI et les bits (0 1) de la variable de contrôle r. Le vecteur de condition sera composé par (r1 r0 RI3 RI1 RI0). Le codage de la condition (RI 0:2 = '01') conduit au vecteur [xxx01] puisque l'expression "RI 0:2" désigne le champ du registre RI commençant au bit 0 et de taille 2. Il s'agit donc des bits 0 et 1.

Le codage des conditions est identique pour toutes les classes de Cmodules.

c. Codage des commandes destinées à l'étage inférieur: Le CPC génère la liste des points d'entrée du niveau inférieur lors de la génération de la table d'états du Cmodule courant. Le nombre de points d'entrée définit le nombre de bits nécessaires pour le codage de ce champ. Les codes commencent à 1 (et non pas à 0) pour la simple raison que le CPC rajoute un automate à un seul état qui correspond au NOP (No OPeration). C'est le point d'entrée de cet automate qui aura le code 0.

Le code donné à un point d'entrée dépend de son ordre d'activation par l'étage supérieur. Le codage de ce champ est le même pour les Cmodules de la classe2 mais il sera différent pour le Cmodule de la classe3 (l'étage le plus bas).

d. Codage des commandes de manipulation des variables de contrôle: Pendant le recensement, le CPC génère la liste des variables de contrôle (internes et externes) modifiées par l'étage courant. Comme pour les variables de condition, il génère un vecteur de contrôle où à chaque variable de contrôle correspondent deux positions figées. La position gauche correspond à la commande RESET et la position droite à la commande SET.

### Exemple:

Supposons avoir quatre variables de contrôle "r 0:2 et b 0:2". On suppose que leur ordre dans le vecteur de contrôle est (r1 r0 b1 b0).

Si dans une action on a les instructions (r 0:2 := '01', b 0:2 := '10'), ces instructions sont équivalentes à

( SET r0:1, SET b1:1, RESET r1:1, RESET b0:1)

le codage de ces actions conduit au vecteur [10 01 01 10].

Le codage de ce champ est le même pour toutes les classes de Cmodules.

## 2- Codage des Cmodules de la classe2: (Les Cmodules intermédiaires)

La seule différence avec le codage du Cmodule de la classe1 est l'existence du champ correspondant aux commandes émanant de l'étage supérieur. Or ce champ coïncide avec le champ des commandes envoyées par l'étage supérieur à l'étage courant. Ce champ a été codé lors du codage



de l'étage supérieur. Donc il suffit de remplacer chaque étiquette désignant un point d'entrée par son code préalablement préparé. Le codage de ce champ est identique pour le Cmodule le plus bas (classe3).

### 3- Codage du Cmodule de la classe3: (l'étage le plus bas)

La différence avec le codage des Cmodules de la classe2 réside au niveau du codage des commandes destinées au niveau inférieur, qui est ici la partie opérative.

Le CPC commence par donner un code CBP à chaque étape de la description de la partie opérative. Ensuite on a le choix entre deux stratégies:

**A) Garder ce code:** dans ce cas APOLLON doit générer un étage de contrôle intermédiaire qui recevra le code de l'étape à exécuter et générera les microcommandes nécessaires pour l'exécution effective de cette étape.

**B) Remplacer le code de chaque étape par les microcommandes qui lui sont associées:** dans ce cas, APOLLON génère la partie opérative et fournit au CPC les microcommandes associées à chaque étape. Remarquons qu'une étape peut avoir une ou deux microcommandes selon le parallélisme et les opérateurs exécutés par l'étape.

#### 1. L'étape possède une seule microcommande:

Le code de l'étape est remplacé tout simplement par la microcommande.

#### **Exemple:**

Soit le monôme

0010 10 --0- 11 **1110** 0000 100101

où le champ en caractères gras correspond au code d'une étape de la partie opérative. Supposons que cette étape s'exécute en un seul cycle (c'est le cas d'un transfert simple) correspondant à la microcommande:

**100110010110**

en remplaçant le code de l'étape par la micro-commande on obtient:

0010 10 --0- 11 **100110010110** 0000 100101

#### 2. L'étape possède deux microcommandes:

Dans ce cas le code de l'étape est remplacé par la première

microcommande et le CPC rajoute un monôme pour la deuxième tout en assurant le séquençement entre le monôme existant et celui rajouté.

**Exemple:**

On va prendre le même monôme que dans l'exemple précédent et supposer que l'étape s'exécute en deux cycles, donc elle nécessite deux microcommandes qui sont:

**100110010110 et 011010011001**

Le monôme d'origine est remplacé par le monôme suivant:

**0010 010 --0- 110 100110010110 0000 100101**

On remarque ici que le codage du séquençement interne tient sur trois bits et non plus sur deux puisque le nombre d'état de l'automate va, au pire des cas, doubler. Pour l'état courant, on rajoute un "0" à gauche. Le code de l'état suivant est obtenu à partir de celui de l'état courant en remplaçant le bit de gauche par "1" (010 --> 110).

Le monôme rajouté par le CPC sera alors:

**0010 110 --0- 011 011010011001 0000 100101**

Pour ce monôme, le point d'entrée ne change pas par rapport à celui du monôme d'origine, l'état courant est donné par l'état suivant du premier monôme, la condition de transition est toujours la même, l'état suivant est obtenu à partir de l'état suivant du monôme d'origine "11" auquel on rajoute "0" à gauche. Les autres champs sont les mêmes que pour le monôme d'origine.

La figure 4.9 donne les PLAs correspondant aux tables d'états de la figure 4.6. Ces PLAs sont donnés sous la forme acceptée par ESPRESSO dans le but de faire une optimisation logique de la description de chaque PLA.

Dans cette description, on donne:

# le nombre d'entrées (non complémentées), le nombre de sorties et le nombre de monômes du PLA,

# la description logique du PLA. Dans l'ordre, les champs correspondent:

1. aux points d'entrées (s'ils existent),
2. à l'état courant de l'automate,
3. aux conditions de transition,
4. à l'état suivant de l'automate,
5. aux commandes destinées à l'étage inférieur,

6. aux commandes de manipulation des variables de contrôle,
7. et aux fils de synchronisation.

```
.name mini_2
.i 7
.o 13
.p 14
00 0---- 00 000 0000 0110
00 1---- 01 000 0000 0110
01 ---- 10 001 0000 0101
10 ---- 11 010 1010 0101
11 00110 00 011 0000 0101
11 00101 00 100 0000 0101
11 00010 00 101 0000 0101
11 00001 00 110 0000 0101
11 00000 00 111 0000 0101
11 10110 10 011 0000 0101
11 10101 10 100 0000 0101
11 10010 10 101 0000 0101
11 10001 10 110 0000 0101
11 10000 10 111 0000 0101
.e
```

```
.name mini_0
.i 10
.o 14
.p 17
0000 00 ---- 00 0000 0000 1010
0001 00 ---- 00 0001 0000 1001
0011 00 ---- 00 0010 0000 1001
0100 00 ---1 00 0011 0000 1001
0100 00 ---0 00 0000 0000 1010
0101 00 ---- 00 0100 0000 1001
0110 00 ---- 00 0101 0000 1001
0111 00 --0- 00 0110 0000 1001
0111 00 --1- 00 0000 0000 1010
1000 00 ---- 00 0111 0000 1001
0010 00 ---- 01 1000 0001 0101
0010 01 ---- 10 1001 0100 0101
0010 10 101- 00 1010 0000 1001
0010 10 011- 00 1011 0000 1001
0010 10 001- 00 1100 0000 1001
0010 10 --1- 00 1101 1000 1001
0010 10 --0- 10 1110 0000 0101
.e
```

```
.name mini_1
.i 6
.o 20
.p 15
000 00 - 00 0000 0000000000 1010
001 00 - 00 0001 0000000000 1001
010 00 - 00 0010 0000000000 1001
011 00 - 01 0010 0000000110 0101
011 01 - 00 0011 0000000000 1001
100 00 - 01 0010 0000000110 0101
100 01 - 00 0100 0000000000 1001
101 00 - 01 0010 0000000110 0101
101 01 - 10 0101 0000100000 0101
101 10 - 11 0110 0101000000 0101
101 11 1 00 0111 1010000000 1001
101 11 0 11 0111 0000000000 0101
110 00 - 00 0010 0000001001 1001
111 00 - 01 0010 0000000110 0101
111 01 - 00 1000 0000000000 1001
.e
```

Figure 4.9: PLAs correspondants à la figure 4.6

### VIII- Génération du fichier de communication

Ce fichier contient toutes les informations nécessaires à la génération du layout de la partie contrôle et à l'assemblage des différents étages de contrôle et de la partie opérative. Ce fichier est composé de deux parties:

- 1) informations associées au circuit global,

## 2) informations associées à chaque étage de contrôle.

Dans la première partie, le CPC fournit:

1. le nom du circuit, .
2. le nombre d'étages de contrôle,
3. les bits de registres constituant le bus de compte rendu. L'ordre de ces bits est important et doit être pris en compte par GEN-TOPO et APOLLON pour assurer la cohérence dans le circuit,
4. les variables de contrôle en entrée, en sortie, en entrée/sortie (bidirectionnelles) et internes. Ces informations permettent la construction du bus de contrôle par le générateur de la partie contrôle.

Quant à la deuxième partie de ce fichier, elle permet de fournir:

1. le niveau d'interprétation associé à l'étage de contrôle,
2. la liste des fils du séquençement interne. Le CPC donne un nom à chaque fil, ce qui permet leur permutation éventuellement par des outils d'optimisation topologique,
3. la liste des fils des points d'entrée. Cette liste est vide pour l'étage le plus haut. De même, les fils sont nommés,
4. la liste des noms des fils correspondant aux commandes du niveau inférieur,
5. la position (verticale ou horizontale) du PLA. Actuellement, la position est toujours horizontale,
6. les bits des registres de données utilisées en entrées (en tant que conditions de transition) par l'étage courant. Là aussi, l'ordre défini par le CPC est important puisqu'il donne l'ordre des fils à l'entrée du PLA. Cet ordre doit être pris en compte par le générateur de PLAs pour la connexion du PLA au bus de compte rendu,
7. les variables de contrôle en entrée (utilisées en tant que conditions de transition) et en sortie (modifiées par l'étage). De même, l'ordre de ces variables est important et doit être pris en compte lors de la génération de la partie contrôle.

La figure 4.10 donne le fichier de communication correspondant au circuit décrit par la figure 2.3.

Dans cet exemple, si on prend l'étage de contrôle PC2, on voit que les variables de condition sont restart0 ( le bit 0 de la variable de contrôle restart) et ir3, ir2, ir1, ir0 (les bits 3 2 1 0 du registre ir). Dans le champ de condition du PLA correspondant à cet étage, la 1<sup>ère</sup> colonne

correspond à restart0 (puisque l'on a dit que les variables de contrôle sont placées à gauche dans le vecteur de condition), les quatre autres colonnes correspondent respectivement à ir3, ir2, ir1, ir0.

```

NAME mini
PCN 3
CRN (ir3 ir2 ir1 ir0 a7)

CTN (dtack0 restart0)
 (adstrobe0 dtstrobe0 read_write0)
 0
 (r0 r1)
PC 2 (seq11 seq12) () (PT21 PT22 PT23)
rot 0
cr (ir3 ir2 ir1 ir0)
ct
 en (restart0)
 so (r1 r0)
PC 1 (seq21 seq22) (PT21 PT22 PT23) (PT31 PT32 PT33 PT34)

rot 0
cr ()
ct
 en (dtack0)
 so (dtstrobe0 adstrobe0 read_write0 r1 r0)

PC 0 (seq31 seq32) (PT31 PT32 PT33 PT34) (PT41 PT42 PT43 PT44)
rot 0
cr (a7)
ct
 en (r1 r0 dtack0)
 so (adstrobe0 read_write0)

```

Figure 4.10: Fichier de communication

## IX- L'optimisation dans le CPC

A part l'optimisation appliquée par le CPC lui-même sur les tables d'états des différents étages de contrôle, on peut faire appel à ARTS pour une optimisation topologique appliquée sur les tables d'états. On peut aussi faire appel à l'outil ESPRESSO pour une optimisation logique appliquée à la description logique de chaque PLA.

### IX-1- L'optimisation topologique

ARTS [BEK87] est un système qui a été développé au sein de l'équipe d'architecture des ordinateurs du laboratoire TIM3 dans le cadre du projet de développement du compilateur de silicium SYCO. Il permet de

chercher un meilleur compromis surface/vitesse du circuit en modifiant la hiérarchie de la partie contrôle. Les transformations faites par ARTS ne changent évidemment pas la fonction réalisée par le circuit décrit par le concepteur.

En entrée, ARTS reçoit les tables d'états générées par le CPC et la liste des points d'entrée de chaque étage de contrôle. Il permet:

1. de fusionner deux étages de contrôle voisins,
2. d'éclater un étage de contrôle en deux,
3. de déplacer la description d'un automate d'un étage à un étage voisin.

Pour augmenter la vitesse du circuit, on peut réduire le nombre d'étages de contrôle par fusion. L'éclatement d'un étage de contrôle en deux ralentit, certes, la vitesse du circuit mais évite en revanche que la largeur de la partie contrôle ne dépasse celle de la partie opérative. Par contre, le déplacement d'un automate d'un étage à l'autre permet d'avoir une partie de contrôle plus régulière.

#### **Exemple d'utilisation de ARTS:**

On va poursuivre le raisonnement sur la description du circuit donnée par la figure 2.3. Cette description mène à un circuit à trois étages. Le plan de masse de ce circuit est donné par la figure 4.11.a. Etant donné que le circuit est très simple, l'effet pyramidal de la partie contrôle n'apparaît pas. Par contre on peut remarquer que la partie contrôle est assez longue (sens vertical).

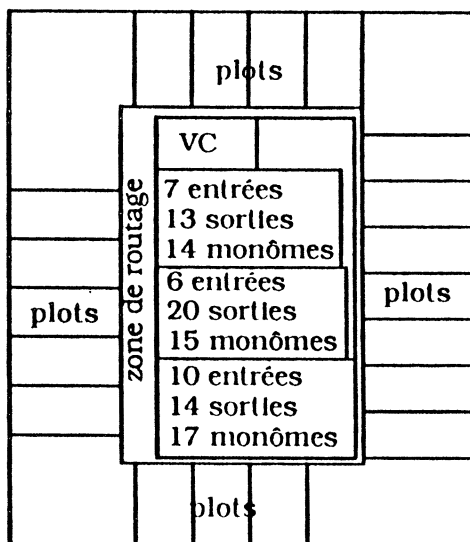
A partir des tables d'états données par la figure 4.6, on va faire trois manipulations:

(1) La fusion des deux étages les plus bas (mini\_0 et mini\_1): L'étage résultant est très large relativement à l'étage le plus haut. Le plan de masse du circuit après cette fusion est donné par la figure 4.11.b. On peut conclure tout de suite que cette solution ne peut être retenue étant donné que la partie contrôle n'est pas régulière et qu'on perd une surface importante de silicium.

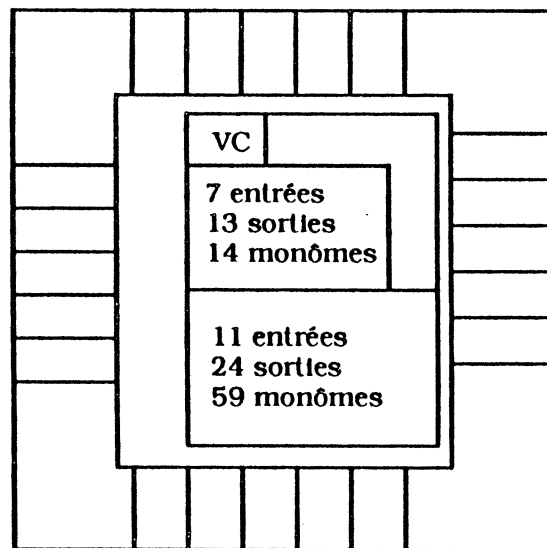
(2) La fusion de deux étages les plus hauts (mini\_2 et mini\_1): Le plan de masse du circuit après la fusion est donné par la figure 4.11.c. On remarque aussi que l'étage résultant est assez large, relativement à l'étage le plus bas. Donc le circuit ne peut être retenu.

(3) Déplacement d'un automate et fusion des deux étages les plus hauts: En effet, pour avoir un circuit régulier, il est impératif, dans ce cas, de déplacer un ou plusieurs automates d'un étage à un autre puis de fusionner deux étages.

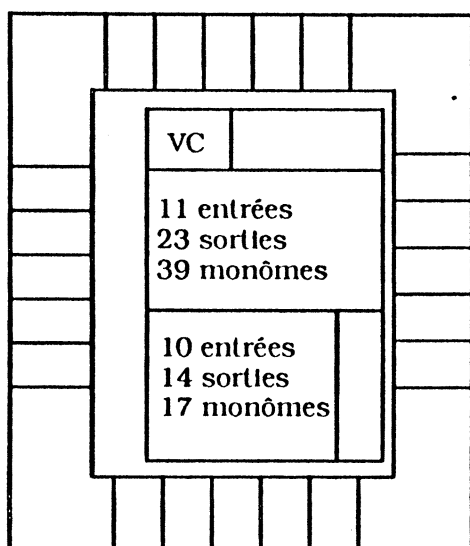
VC= variables de contrôle



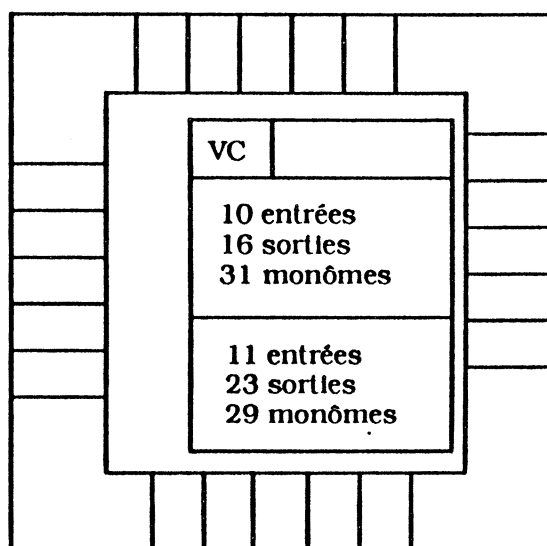
a)



b)



c)



d)

Figure 4.11: Résultats de ARTS

Dans notre cas, il s'agit de déplacer l'automate, dont l'état initial est "sta\_g191\_1" dans la figure 4.6, de l'étage du milieu (mini\_1) vers l'étage

inférieur (mini\_0), puis de fusionner les deux étages les plus hauts. Les deux étages obtenus ont pratiquement la même largeur et le circuit est assez régulier. Cette solution peut être retenue. Le plan de masse du circuit est donné par la figure 4.11.d.

### IX-2- L'optimisation logique

```
#mini_2
.i 7
.o 13
.p 11
1-1000- 10000000000000
1-10-01 10000000000000
-110-10 10000000000000
1--000- 00010000000000
-01---- 01000000000000
11-0110 0001100000101
11-0-01 0010000000101
11-00-0 0010100000101
00----- 0000000000110
01----- 1000100000101
10----- 1101010100101
.e

#mini_1
.i 6
.o 20
.p 13
11000- 00000000000010010000
00000- 00000000000000001010
11101- 00100000000000001001
101111 00011110100000001001
10001- 00010000000000001001
00100- 00000100000000001001
101110 11011100000000000101
01101- 00001100000000001001
-1000- 00001000000000001001
10101- 10010100001000000101
10110- 11011001010000000101
-1100- 01001000000001100101
10-00- 01001000000001100101
.e

#mini_0
.i 10
.o 14
.p 16
00101010-- 00001000000000
00101001-- 00001000000000
011100--1- 00000000001010
0-0000---0 00000000001010
000000---- 00000000001010
011100--0- 00011000001001
010000---1 00001100001001
001010--0- 10111000000101
010100---- 00010000001001
001001---- 10100101000101
000100---- 00000100001001
001010--1- 00110110001001
001000---- 01100000010101
001100---- 00001000001001
100000---- 00011100001001
011000---- 00010100001001
.e
```

Figure 4.12: Résultat de ESPRESSO à partir des PLAs de la figure 4.9

ESPRESSO [RUD85] est un outil d'optimisation logique de PLAs. On a vu que la description logique des PLAs générés par le CPC est donnée sous



une forme acceptée par ESPRESSO.

L'application de cet outil à une telle description permet de réduire le nombre de monômes du PLA. La figure 4.12 illustre le résultat de ESPRESSO à partir des PLAs de la figure 4.9. On remarque qu'il y a un gain de trois monômes pour l'étage mini\_2, de deux monômes pour l'étage mini\_1 et d'un seul monôme pour l'étage mini\_0. ESPRESSO a aussi été utilisé pour vérifier que le résultat de l'optimisation logique ne dépend pas seulement du type de code utilisé pour le codage des états. En effet, on génère les mêmes PLAs en utilisant le code GRAY pour coder les états. Après passage de ESPRESSO il n'y a qu'un gain de deux monômes pour l'étage mini\_2, d'un seul monôme pour l'étage mini\_1, par contre il y a un gain de deux monômes pour l'étage mini\_0.

#### **X- Autres résultats**

Tout au long de cette thèse, on a essayé de guider l'explication du fonctionnement du CPC par des exemples simples et en se référant à la description d'un microprocesseur simplifié. Le CPC a été également utilisé pour la synthèse de la partie contrôle de microprocesseurs du marché. L'un de ces microprocesseurs est le 6502 qui est un microprocesseur 8 bits. La description de départ, dont on a parlé au §VII-2, a conduit à un circuit dont la partie contrôle est composée de trois étages. Les caractéristiques de ces étages ont été:

Etage2: - 18 entrées (non complémentées),  
- 33 sorties,  
- 55 monômes.

Etage1: - 23 entrées (non complémentées),  
- 28 sorties,  
- 166 monômes.

Etage0: - 39 entrées (non complémentées),  
- 32 sorties,  
- 170 monômes.

Les PLAs réalisant les différents étages ont été générés en codant les états

en code CBP et en code GRAY. On a appliqué ESPRESSO aux deux ensembles de descriptions de PLAs, les résultats ont donné:

Code CBP: - étage2 52 monômes (gain de 3 monômes),  
 - étage1 146 monômes (gain de 20 monômes),  
 - étage0 133 monômes (gain de 37 monômes).

Code GRAY: - étage2 55 monômes (pas de gain),  
 - étage1 141 monômes (gain de 25 monômes),  
 - étage0 123 monômes (gain de 47 monômes).

On remarque ainsi que le code GRAY a été meilleur pour les étages 1 et 0, alors que le code CBP l'a été pour l'étage 2. Mais dans l'ensemble, pour ce circuit, le code GRAY est meilleur.

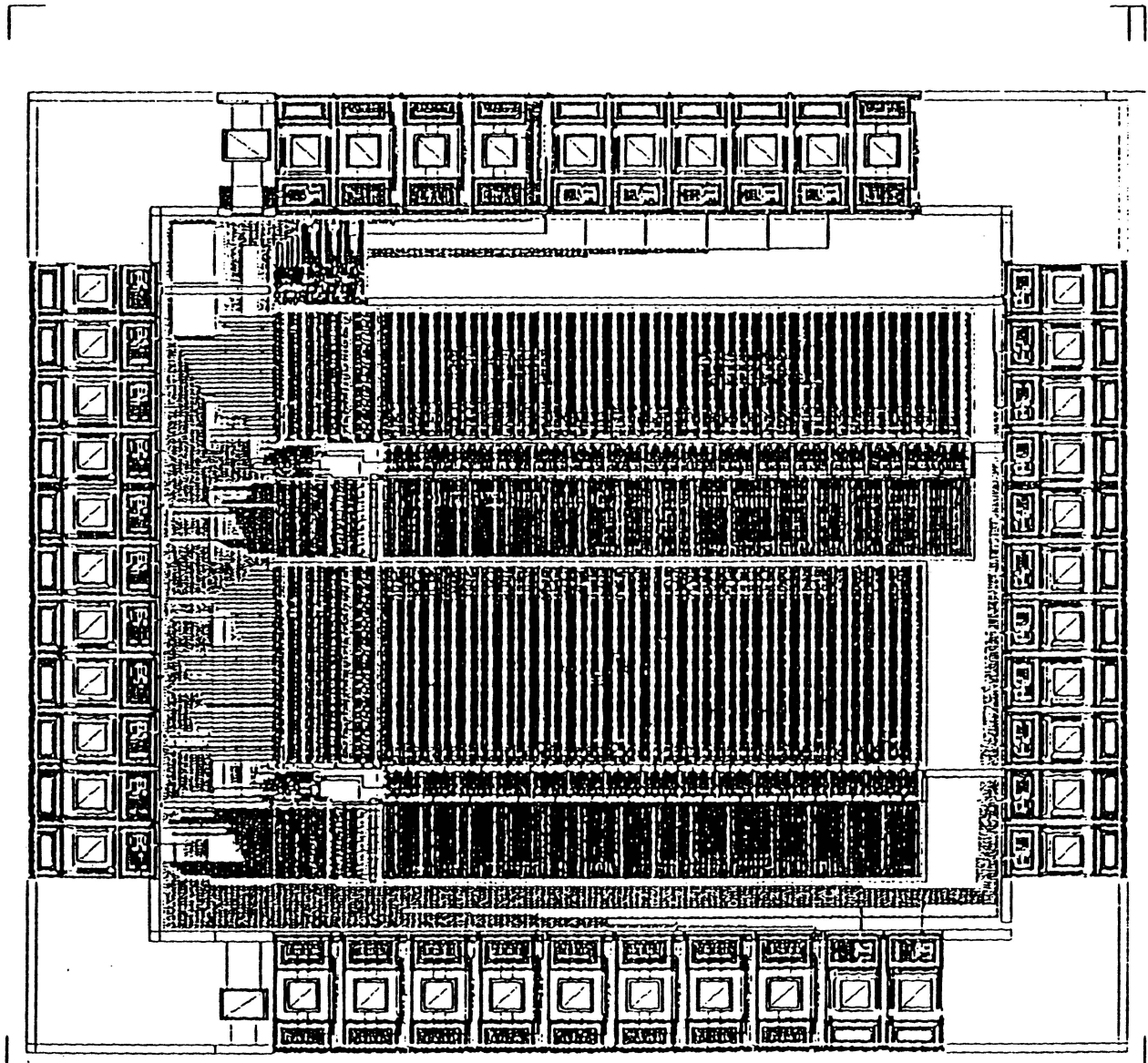


Figure 4.13: Le layout de la partie contrôle du 6502 [GER87].

ARTS a été utilisé pour améliorer les performances du circuit. En effet on a obtenu un circuit à deux étages de contrôle. Le dessin des masques de la partie contrôle de ce microprocesseur est donné par la figure 4.13.

## **XI- Conclusion**

L'avantage le plus important du CPC est le fait de maintenir sa forme intermédiaire en langage LDS, ce qui permet au concepteur de donner la description de son circuit sous cette forme intermédiaire. En outre, les tables d'états sont simulables par le même environnement que la description de départ. Donc même à une étape très avancée de la synthèse, on peut toujours simuler le fonctionnement du circuit. De plus, les tables d'états peuvent être récupérées par le concepteur qui pourrait, éventuellement, améliorer lui-même les performances de son circuit.

Les tables d'états servent aussi de point de départ au CPC pour la génération d'autres structures de contrôle. En effet, si pour cette version les étages de contrôle sont implémentés par des PLAs, on verra dans le chapitre suivant qu'à partir de ces tables, il sera possible de générer des étages de contrôle microprogrammés.

En revanche, dans la présentation du CPC, on a démontré que les compilateurs de parties contrôle ne doivent pas obligatoirement travailler sur des formes intermédiaires qui sont exprimées dans un langage différent du langage d'entrée du compilateur. Par conséquent, le choix du langage de départ ne doit pas tenir compte seulement de la façon par laquelle on peut décrire un circuit, mais aussi des transformations, logiques et topologiques, susceptibles d'être appliquées à la description pour obtenir un meilleur résultat final.

La description d'entrée du CPC est hiérarchique, le nombre d'étages de contrôle du circuit dépend de la description donnée par le concepteur. Pour simplifier la compréhension de sa description, le concepteur a tendance à augmenter la hiérarchie du circuit. Pour les gros circuits, ce phénomène se traduit par un effet pyramidal dans la partie contrôle du circuit. L'appel à ARTS permet d'équilibrer la partie contrôle et d'améliorer les performances du circuit généré.

# **CHAPITRE 5**

## **Perspectives**



## I- Introduction

Les PLAs sont des structures très régulières et faciles à générer automatiquement. Cependant, quand l'algorithme implémenté par un PLA est assez complexe, le PLA occupe une surface considérable de silicium. De plus, la rapidité du PLA se dégrade. Pour remédier à ce problème de rapidité, il faut utiliser d'autres structures pour implémenter chaque étage de contrôle. Dans la suite de ce chapitre, on va montrer que la génération de structures microprogrammées est facilement automatisable dans le cas du compilateur CPC. On présentera aussi une autre structure d'étages de contrôle multi-PLAs.

## II- Etage de contrôle microprogrammé

Rappelons que les tables d'états de chaque étage de contrôle se présente sous forme d'automates indépendants. Ces automates sont du type Mealy.

Pour pouvoir implémenter un étage de contrôle par une structure microprogrammée, il est nécessaire tout d'abord de transformer chaque automate de Mealy en un automate de Moore équivalent.

La table d'états de l'étage "mini\_2" de la figure 4.6 est composée d'un seul automate de Mealy de quatre états. Le diagramme de transitions de cet automate est illustré par la figure 5.3 où les états E1, E2, E3, et E4 correspondent respectivement aux états mini\_rest\_1, mini\_start\_1, mini\_newinstr\_1, et mini\_decode\_1 dans la description de la figure 4.6. Un automate de Moore qui lui équivaut est donné par la figure 5.4.

Dans le cadre de la préparation de la génération des étages de contrôle microprogrammés, on a développé un programme prototype qui permet effectivement de transformer une table d'états sous forme de Mealy en son équivalent sous forme de Moore. En appliquant ce programme à la description de la figure 4.6, on obtient la table d'états (sous forme de Moore) illustrée par la figure 5.5.

Ce programme réalise un algorithme décrit dans [BAR79] et dont le principe est le suivant:

Soit un automate de Mealy  $S_A = (A_A, Z_A, W_A, \delta_A, \lambda_A, a_{1A})$  où

$A_A = \{a_1, \dots, a_m\}$  : ensemble d'états de  $S_A$ .

$Z_A = \{z_1, \dots, z_f\}$  : vocabulaire d'entrée de  $S_A$ .

$W_A = \{w_1, \dots, w_g\}$  : vocabulaire de sortie de  $S_A$ .

$\delta_A : A_A * Z_A \dashrightarrow A_A$  : fonction de transition,

$\lambda_A : A_A * Z_A \dashrightarrow W_A$  : fonction des sorties,

$a_{1A} = a_1$  c'est l'état initial de  $S_A$ .

Construire un automate de Moore  $S_B = (A_B, Z_B, W_B, \delta_B, \lambda_B, a_{1B})$  revient à définir  $A_B$ ,  $\delta_B$ ,  $\lambda_B$  et  $a_{1B}$ .  $Z_B$  et  $W_B$  étant égaux respectivement à  $Z_A$  et  $W_A$ .

Pour définir  $A_B$ , considérant un état  $a_s$  de  $A_A$ , on lui fait correspondre l'ensemble  $A_S$  défini comme suit:

$$A_S = \{ (a_s, w_g) / \delta_A(a_m, z_f) = a_s \text{ et } \lambda_A(a_m, z_f) = w_g \}$$

Le nombre d'éléments de  $A_S$  est égal au nombre des différents signaux de sortie des transitions amenant à l'état  $a_s$  à partir de l'automate  $S_A$  (voir figure 5.1).

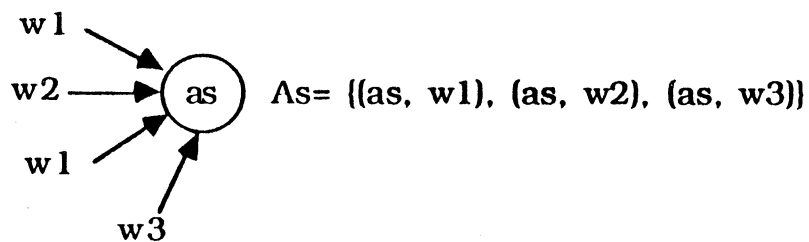


Figure 5.1 Construction de  $A_S$  [BAR79]

Le nombre d'états de l'automate  $S_B$  est la réunion des ensembles  $A_S$  ( $S=1..m$ ).

Les fonctions des sorties  $\lambda_B$  et de transitions  $\delta_B$  sont définies de la façon suivante:

A chaque état de  $A_B$ , qui représente un couple du type  $(a_s, w_g)$ , on associe un signal de sortie  $w_g$ .

Si l'automate de Mealy  $S_A$  admet une transition  $\delta_A(a_m, z_f) = a_s$  suivie de l'élaboration d'un signal de sortie  $\lambda_A(a_m, z_f) = w_g$ , l'automate de Moore

$S_B$  doit admettre une transition à partir de l'ensemble  $A_m$  vers l'état  $(a_s, w_g)$  sous l'action du même signal  $z_f$  (voir figure 5.2).

Comme état initial  $a_{1B}$ , on peut retenir n'importe lequel des états de  $A_1$  engendrés par l'état initial  $a_1$  de l'automate  $S_A$ .

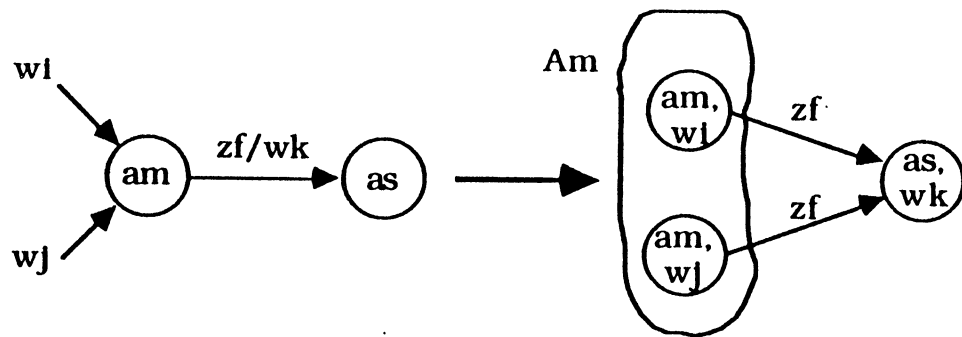


Figure 5.2: Illustration du passage du modèle de Mealy à celui de Moore [BAR79]

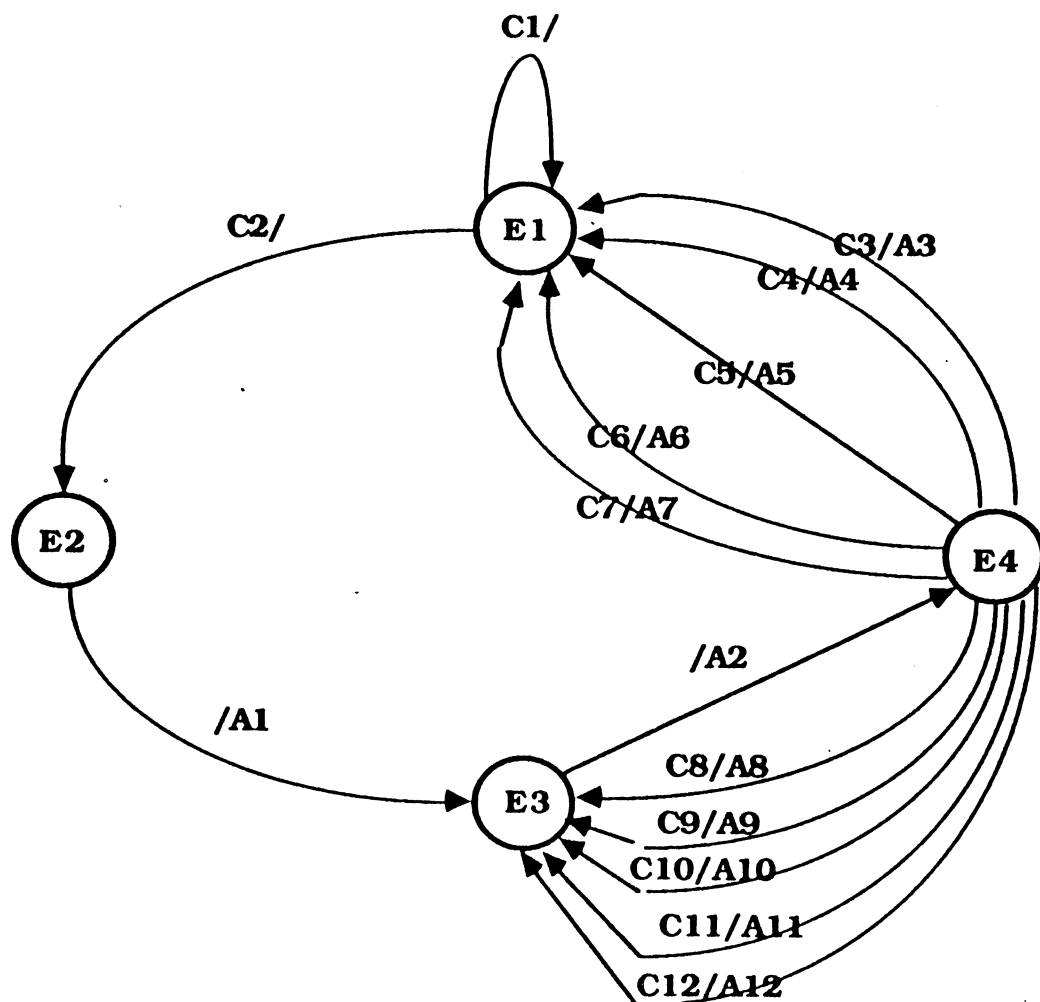


Figure 5.3: Automate de Mealy



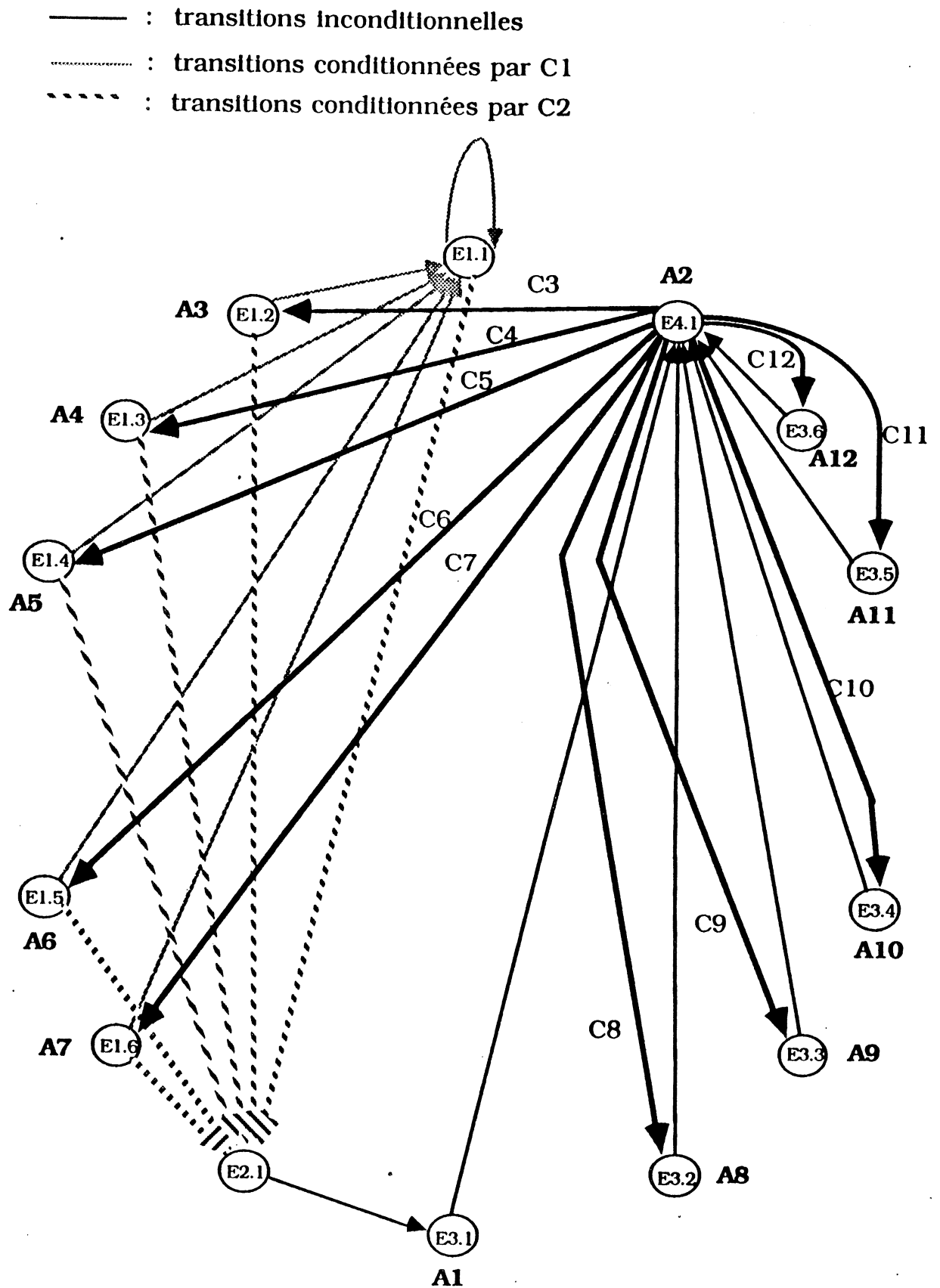


Figure 5.4: Automate de Moore équivalent à l'automate de Mealy de la figure 5.3

```

?----- Module mini_2
CMODULE mini_2
<mini_rest_1.1>
 (IF (restart 0:1 = (0) NEXT mini_rest_1.1; ENDIF;
 IF (restart 0:1 = (BIN'1') NEXT mini_start_1.1; ENDIF;)
<mini_rest_1.2>
 (IF (restart 0:1 = (0) NEXT mini_rest_1.1; ENDIF;
 IF (restart 0:1 = (BIN'1') NEXT mini_start_1.1; ENDIF;
 EXECUTE mini_1 (bru_g194_1.1);)
<mini_rest_1.3>
 (IF (restart 0:1 = (0) NEXT mini_rest_1.1; ENDIF;
 IF (restart 0:1 = (BIN'1') NEXT mini_start_1.1; ENDIF;
 EXECUTE mini_1 (bru_g195_1.1);)
<mini_rest_1.4>
 (IF (restart 0:1 = (0) NEXT mini_rest_1.1; ENDIF;
 IF (restart 0:1 = (BIN'1') NEXT mini_start_1.1; ENDIF;
 EXECUTE mini_1 (bru_g191_1.1);)
<mini_rest_1.5>
 (IF (restart 0:1 = (0) NEXT mini_rest_1.1; ENDIF;
 IF (restart 0:1 = (BIN'1') NEXT mini_start_1.1; ENDIF;
 EXECUTE mini_1 (bru_g193_1.1);)
<mini_rest_1.6>
 (IF (restart 0:1 = (0) NEXT mini_rest_1.1; ENDIF;
 IF (restart 0:1 = (BIN'1') NEXT mini_start_1.1; ENDIF;
 EXECUTE mini_1 (bru_g192_1.1);)
< mini_start_1.1>
 (NEXT mini_newinstr_1.1;)
< mini_newinstr_1.1>
 (NEXT mini_decode_1.1; EXECUTE mini_1 (mini_start_1.1);)
< mini_newinstr_1.2>
 (NEXT mini_decode_1.1; EXECUTE mini_1 (bru_g194_1.1);)
< mini_newinstr_1.3>
 (NEXT mini_decode_1.1; EXECUTE mini_1 (bru_g195_1.1);)
< mini_newinstr_1.4>
 (NEXT mini_decode_1.1; EXECUTE mini_1 (bru_g191_1.1);)
< mini_newinstr_1.5>
 (NEXT mini_decode_1.1; EXECUTE mini_1 (bru_g193_1.1);)
< mini_newinstr_1.6>
 (NEXT mini_decode_1.1; EXECUTE mini_1 (bru_g192_1.1);)
< mini_decode_1.1>
 (r 0:2 := 0;
 EXECUTE mini_1 (mini_newinstr_1.1);
 IF (restart 0:1 =(0) . (lr 0:4 =(BIN'0110'))) NEXT mini_rest_1.2; ENDIF;
 IF (restart 0:1 =(0) . (lr 0:4 =(BIN'0101'))) NEXT mini_rest_1.3; ENDIF;
 IF (restart 0:1 =(0) . (lr 0:4 =(BIN'0010'))) NEXT mini_rest_1.4; ENDIF;
 IF (restart 0:1 =(0) . (lr 0:4 =(BIN'0001'))) NEXT mini_rest_1.5; ENDIF;
 IF (restart 0:1 =(0) . (lr 0:4 =(BIN'0000'))) NEXT mini_rest_1.6; ENDIF;
 IF (restart 0:1 =(BIN'1') . (lr 0:4 =(BIN'0110')))
 NEXT mini_newinstr_1.2; ENDIF;
 IF (restart 0:1 =(BIN'1') . (lr 0:4 =(BIN'0101')))
 NEXT mini_newinstr_1.3; ENDIF;
 IF (restart 0:1 =(BIN'1') . (lr 0:4 =(BIN'0010')))
 NEXT mini_newinstr_1.4; ENDIF;
 IF (restart 0:1 =(BIN'1') . (lr 0:4 =(BIN'0001')))
 NEXT mini_newinstr_1.5; ENDIF;
 IF (restart 0:1 =(BIN'1') . (lr 0:4 =(BIN'0000')))
 NEXT mini_newinstr_1.6; ENDIF;)
END;
?----- End of module mini_2

```

Figure 5.5: Table d'états (forme de Moore)

Pour l'exemple de la figure 4.6, l'étage de contrôle est composé d'un seul automate de Mealy qui n'a pas d'état final. Donc l'automate de Moore généré par ce programme n'admet pas non plus d'état final. Si l'automate de Mealy admet un état final, le programme rajoute des états finaux selon le nombre d'EXIT qu'on a dans l'automate de Mealy et selon les actions associées.

**Exemple:**

```
Etat1(If (A=0) EXECUTE point1; EXIT; Endif;
```

```
 If (A=1) EXECUTE point2; EXIT; Endif;)
```

cet état se comporte comme état final pour les deux conditions, dans ce cas on va rajouter deux états finaux dont on associe l'action " EXECUTE point1" au premier et l'action " EXECUTE point2" au deuxième.

On peut cependant remarquer que ce programme n'optimise pas l'automate de Moore généré. En effet, l'état "mini\_start\_1.1" n'assure qu'un branchement et aucune action ne lui est associée. Donc il peut être éliminé en remplaçant le branchement à cet état par le branchement qu'il assure.

La table d'états (forme de Moore) se caractérise par le fait que, pour chaque état, seul le séquençement peut être conditionnel. Toutes les actions sont associées à l'état et sont donc inconditionnelles.

## **II-1:- Structure d'un étage de contrôle microprogrammé**

La structure d'un étage de contrôle microprogrammé est illustrée par la figure 5.6.

La ROM constitue le mécanisme de génération des commandes alors que le PLA réalise le mécanisme de séquençement interne.

A chaque état de la table d'états va correspondre une microinstruction dans la ROM. A partir de la table d'états, le codage des actions constituera le contenu de la ROM et le codage du branchement générera la description logique du PLA de séquençement.

Dans chaque étage de contrôle microprogrammé, il y aura un registre de microadresse qui donne l'adresse de la microinstruction à exécuter. Si dans la structure mono-PLA un numéro d'état est composé de deux champs (le numéro du point d'entrée et le numéro de l'état dans l'automate désigné par le point d'entrée), dans le cas de la structure microprogrammée chaque état aura un numéro absolu dans l'étage de contrôle. Cette solution sera retenue pour ne pas avoir d'adresses

inutilisées dans la ROM, ce qui serait le cas si on maintenait l'adresse en deux champs.

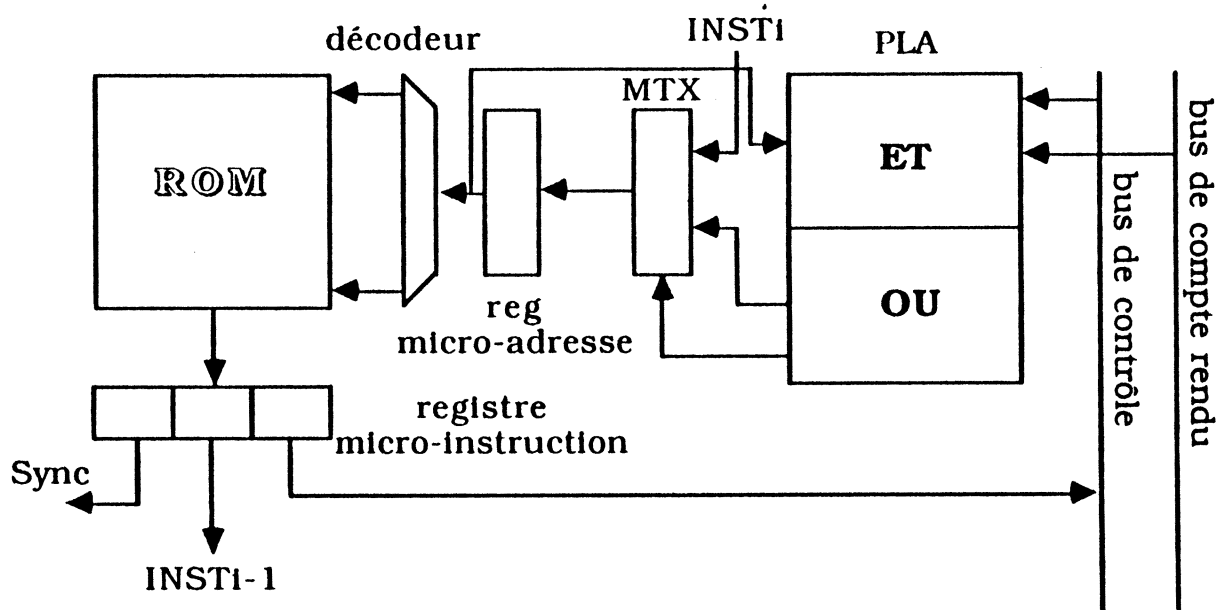


Figure 5.6: Etage de contrôle microprogrammé

### Exemple:

Supposons avoir un étage de contrôle à trois automates, dont le plus long possède 16 états. On aura besoin de 6 bits pour l'adressage (2 bits pour le point d'entrée et 4 bits pour les états). Si l'automate de numéro '10' ne possède que 4 états, seules les adresses comprises entre '100000' et '100011' seront utilisées. Les adresses comprises entre '100100' et '101111' ne le seront pas, ce qui conduira à des "trous" dans la ROM.

Le nombre de bits du registre de microadresse sera donc défini par le nombre total d'états de l'étage de contrôle.

### II-2:- Structure de la ROM

A chaque état de la table d'états va correspondre une micro-instruction dans la ROM. Chaque micro-instruction sera composée de trois champs:

- 1) la commande vers l'étage inférieur,
- 2) les commandes de manipulation des variables de contrôle,
- 3) les commandes de synchronisation.

### II-3- Le PLA d'adressage

La description logique du PLA d'adressage est obtenue par codage des

instructions conditionnelles, qui définissent le séquençement, de la table d'états ou de l'instruction de branchement inconditionnelle (NEXT ou EXIT).

En entrée, le PLA reçoit:

- # le bus de contrôle,
- # le bus de compte rendu,
- # le registre de micro-adresse.

En sortie, le PLA génère:

- # la micro-adresse suivante,
- # un fil destiné au multiplexeur du registre de micro-adresse.

Le registre de micro-adresse contient l'état courant et permet de calculer l'état suivant (la micro-adresse suivante). Ce registre va être chargé par l'adresse de base donnée par la commande émanant de l'étage supérieur au début de chaque cycle d'un étage. Par la suite, il est chargé par la micro-adresse fournie par le PLA. Le multiplexeur sert alors à faire cet aiguillage.

Ce choix a été fait pour ne pas être amené à utiliser la commande de l'étage supérieur comme entrée du PLA de séquençement, ce qui permet de réduire la surface occupée par le PLA d'adressage.

Pour les étages microprogrammés, le codage de la table d'états doit commencer par l'étage le plus bas pour qu'on puisse propager le code des points d'entrée vers l'étage supérieur.

Remarquons par ailleurs que pour l'étage le plus haut, on n'aura pas besoin du multiplexeur puisqu'il n'y a pas de commande émanant de l'étage supérieur.

La structure microprogrammée présentée a été définie sans tenir compte des problèmes topologiques qui peuvent se poser, par conséquent, les choix présentés ici ne sont pas définitifs.

### **III- Etage de contrôle multi-PLAs**

Dans ce paragraphe, on va proposer une structure de contrôle à trois PLAs. Cette proposition est fondée sur des constatations faites tout au long du développement et de l'utilisation du CPC.

En analysant la description logique des PLAs pour des circuits complexes, tels que le 6502, on a remarqué que les matrices ET et OU sont assez creuses surtout au niveau des variables de condition (entrées de la matrice ET) et des commandes des variables de contrôle (sorties de la

matrice OU). On va illustrer ces constatations en présentant le résultat du 6502.

1) Etage supérieur:

- 13 variables de condition (seulement 43 produits ont été utilisés),
- 18 sorties pour les commandes des variables de contrôle (seulement 16 produits ont été utilisés).

2) Etage du milieu:

- 14 variables de condition (seulement 67 produits ont été utilisés),
- 12 sorties pour les commandes des variables de contrôle (seulement 75 produits ont été utilisés).

3) Etage le plus bas:

- 30 variables de condition (seulement 113 produits ont été utilisés),
- 16 sorties pour les commandes des variables de contrôle (seulement 13 produits ont été utilisés).

En limitant notre raisonnement à ce dernier étage et sachant qu'il possède 170 monômes, il serait possible de réduire la surface du PLA réalisant cet étage en utilisant un encodeur de condition en entrée et un décodeur de commandes en sortie. Ces deux dispositifs peuvent être réalisés par des PLAs.

Le premier sera caractérisé par:

- 30 entrées,
- 7 sorties,
- 113 monômes.

Sa surface sera alors  $S1 = \beta \cdot (2 \cdot 30 + 7) \cdot 113$  où  $\beta$  ( $\beta = 11 \lambda$ ) est un facteur qui tient compte du pas d'entrée, de sortie et de monômes.

Le deuxième PLA aura:

- 4 entrées,
- 16 sorties,
- 13 monômes.

Sa surface sera  $S2 = \beta \cdot (2 \cdot 4 + 16) \cdot 13$ .

La surface occupée par ces deux PLAs est de l'ordre de 7883  $\beta$  unités de surface. Or la surface du PLA de départ est de:

$\beta \cdot (2 \cdot 23 + 12) \cdot 170 = 9860 \beta$ , d'où un gain de 2000  $\beta$  unités.

Cette illustration montre qu'il est possible de réduire la surface occupée par un étage de contrôle. Mais, il faut aussi tenir compte des interfaces des PLAs et de la régularité de l'étage obtenu.

Cette décomposition ne permet pas de réduire la surface dans tous les cas, par conséquent une évaluation préalable de la surface et de la régularité des deux structures (mono-PLA et multi-PLAs) est nécessaire pour pouvoir prendre une décision sans risque. La structure d'un étage de contrôle multi-PLAs est illustré par la figure 5.7.

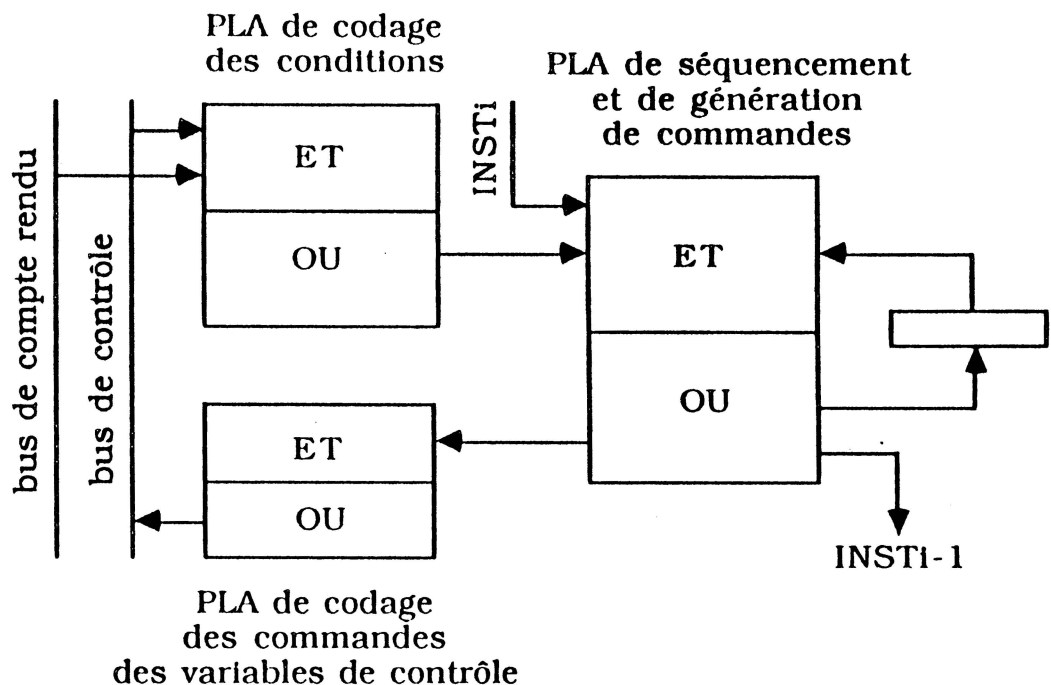


Figure 5.7: Etage de contrôle multi-PLAs

#### IV- Conclusion

Dans ce chapitre, on a présenté le profil de l'évolution du compilateur de silicium SYCO. Pour les étages de contrôle microprogrammés et multi-PLAs, une étude topologique de chaque structure reste à faire pour la validation de ces solutions. Pour la sûreté du fonctionnement des circuits générés, une autre évolution de SYCO consiste à générer des circuits autotestables en utilisant un concept récent nommé UBIST (Unified Built-In Self Test) [NIC86] unifiant le test intégré en ligne (circuits autotestables: "Self-Checking") et le test intégré hors ligne de type BIST (Built-In Self Test). Une étude a été effectuée pour définir une stratégie de génération automatique de partie contrôle mono-PLA UBIST [TOR88].

## **CONCLUSION**





Dans le premier chapitre de cette thèse, un rappel de l'histoire et des différents types de compilateurs de silicium. Un compilateur de silicium comportemental ne peut être général et doit se limiter à générer une catégorie de circuits bien définie dès le début de son développement. Cette limitation réside au niveau de la sémantique du langage d'entrée du compilateur et de l'architecture cible des circuits générés.

Par conséquent, le fait de se restreindre à une catégorie de circuits et à une architecture cible permet de simplifier la tâche de traduction qu'un compilateur de silicium doit accomplir et d'avoir un mécanisme facile à comprendre. Ça permet aussi au compilateur de chercher un meilleur compromis du circuit final en modifiant l'organisation du circuit tout en restant fidèle à l'architecture cible.

Les deux décisions à prendre pour développer un compilateur de silicium comportemental sont:

- 1) le choix du langage d'entrée qui doit permettre de décrire de façon concise les fonctions que le circuit doit réaliser,

- 2) le choix de l'architecture cible permettant de bien implémenter les spécifications faites dans le langage d'entrée du compilateur.

La présentation du compilateur de silicium SYCO a permis de mettre en évidence ces deux décisions et de démontrer que le circuit généré peut dépendre des spécifications données par le concepteur.

Dans le cas du compilateur de parties contrôle CPC, on a montré que le nombre d'étages de contrôle dépend de la hiérarchie donnée par le concepteur. Le système ARTS est un exemple d'outil qui illustre la façon par laquelle on peut modifier la description du circuit pour aboutir à un résultat qui satisfait le concepteur.

Pour simplifier la synthèse de la partie contrôle, les compilateurs de parties contrôle passent par une forme intermédiaire. Le choix de celle-ci est d'une très grande importance étant donné que le résultat de la synthèse est étroitement influencé par ce choix. Le choix du langage d'entrée d'un compilateur de silicium doit tenir compte de la forme

intermédiaire par laquelle le compilateur de partie contrôle passe et des différentes transformations qui lui sont appliquées.

Dans le cadre de la présentation du CPC, on a démontré que le fait de maintenir sa forme intermédiaire en un sous-ensemble du langage LDS offre une très grande souplesse d'utilisation au concepteur. Celui-ci a la possibilité d'intervenir à n'importe quelle étape de la synthèse pour améliorer les performances de son circuit.

Toujours dans le cadre du CPC et de SYCO, on a présenté la façon par laquelle on génère des étages de contrôle mono-PLA. Etant donné que les PLAs, pour des circuits complexes, risquent d'occuper une surface considérable dans le silicium et d'être lents, on a vu la nécessité d'étudier la génération d'autres structures d'étages de contrôle, notamment les étages microprogrammés. Là aussi, on a démontré l'importance de maintenir la forme intermédiaire du CPC en LDS. En effet, la génération d'étages de contrôle microprogrammés se résume alors à la transformation des automates de Mealy en automates de Moore équivalents, puis le codage direct permet de générer la ROM et le PLA de séquençement. Avant de passer au codage, une optimisation logique de la description de chaque étage de contrôle est indispensable pour réduire la surface occupée par la partie contrôle.

Pour un compilateur de silicium, il est important d'assurer une certaine indépendance entre les différents outils qui le composent et leur intégrité. C'est le cas de SYCO où tous les outils évoluent autour de la même structure de données en LDS. Effectivement, cette structure assure la cohérence de l'ensemble et a permis de développer les différents outils en parallèle puisqu'on peut toujours entrer la description d'entrée de chaque outil manuellement.

Une version du CPC générant des étages de contrôle mono-PLA a été développée en Le\_lisp (de l'INRIA version 15.2). Il a été utilisé pour la synthèse de parties contrôle de circuits complexes. Le temps de compilation était raisonnable (700s pour générer les trois étages de contrôle). La version du CPC générant des étages microprogrammés n'est pas entièrement réalisée étant donné que les problèmes topologiques ne sont pas encore résolus et que l'interface du CPC avec un générateur

topologique de ROM n'est pas encore définie.

Le CPC forme un ensemble cohérent avec les autres outils constituant le compilateur SYCO. Il s'interface avec ARTS et GEN-TOPO via des structures de données qui simplifient la communication entre ces outils. L'interfaçage avec le compilateur APOLLON doit aussi se faire par l'intermédiaire de structure de données qui permettra au CPC de récupérer les microcommandes associées à chaque étape de la partie opérative.



***REFERENCES ET***  
***BIBLIOGRAPHIE***



- [ANC80] **F. ANCEAU.**  
Architecture and Design of Van Newmann Microprocessor.  
NATO advanced summer institute, July 1980.
- [ANC83] **F. ANCEAU.**  
CAPRI: A Design Methodology and a Silicon Compiler for VLSI  
Circuits specified by Algorithms.  
3rd Caltech Conference on VLSI, March 1983.
- [ANC86] **F. ANCEAU.**  
The Architecture of Microprocessors.  
Addison-Wesley Publishing Company. 1986.
- [ARM62] **B. D. ARMSTRONG.**  
A Programmed Algorithm for Assigning internal codes to  
Sequential Machines.  
IRE Trans. on Elec. Comp., Vol. EC11, Aug. 62, pp466-472.
- [AYR79] **R. AYRES.**  
Silicon Compilation- a Hierarchical Use Of PLAs.  
DAC'79, pp314-326.
- [BAR79] **S. BARANOV.**  
Synthèse des Automates microprogrammés.  
Edition de MOSCOU. 1979.
- [BER87] **N. BEKKARA.**  
Optimisation et Compromis Surface/Vitesse dans le Compilateur  
de Silicium SYCO.  
Thèse de docteur ingénieur de l'INPG. Octobre 1987.
- [BLA85] **T. BLACKMAN, J. FOX, C. ROSEBRUGH.**  
The SILC Silicon Compiler: Language And Features.  
DAC'85, pp232-237.
- [Bla85] **R.L. BLACKBURN. D.E. THOMAS.**  
Linking The Behavioral And Structural Domains Of  
Representation In a Synthesis System.  
DAC'85, pp374-380.
- [BRA84] **R.K. BRAYTON, C. McMULLEN.**  
Synthesis and Optimization Of Multistage Logic.  
ICCD'84, pp23-28.
- [BRA85] **R.K. BRAYTON, C.L. CHEN, G. De MICHELI, R.H.J.M. OTTEN.**  
A Microprocessor Design Using The YORKTOWN Silicon  
Compiler. ICCD'85, pp225-231.



- [BRA85a] R.K. BRAYTON and Al.**  
Logic Minimization algorithms For VLSI Synthesis.  
Fluwer Academic Publishers, 1985.
- [BRA86] R.K. BRAYTON & Al.**  
Multiple-Level Logic Optimization System.  
ICCAD'86. pp356-359.
- [BRO81] D.W. BROWN.**  
A State Machine Synthesizer.  
DAC'81, pp301-305.
- [CAM85] R. CAMPOSANO.**  
Synthesis Techniques for Digital Systems design.  
DAC'85. pp475-481.
- [CAM86] R. CAMPOSANO, A. KUNZMANN.**  
Considering Timing Constraints In Synthesis From A  
Behavioural Description.  
ICCD'86, pp6-9.
- [CAM87] R. CAMPOSANO.**  
Structural Synthesis In The YORKTOWN Silicon compiler.  
VLSI'87.
- [CHA86] H.H. CHAO, S. ONG, K. LEWIS, J.Y. TANG.**  
Design Automation for Control Circuits Implemented with PLAs.  
ICCD'86. pp526-529.
- [CHU86] T. CHU.**  
Synthesis Of Self-timed Control Circuits From Graphs: An  
Example.  
ICCD'86, pp565-571.
- [COP86] A.J. COPPOLA.**  
An Implementation of a State Assignment Heuristic.  
DAC'86. pp643-647.
- [DeM83] G. De MICHELI, A. SANGIOVANNI-VINCENTELLI.**  
Computer-aided Synthesis Of PLA-based Finite State Machines.  
ICCAD'83, pp154-156.
- [DeM84a] G. De MICHELI, A.S. VINCENTELLI, R. BRAYTON.**  
KISS: A Program For Optimal State Assignment Of Finite state  
Machines.  
ICCAD'84 Santa Clara. pp209-211.
- [DeM84b] G. De MICHELI.**  
Optimal Encoding Of Control Logic.  
IEEE, ICCD'84, New York, pp16-22.

- [DeM86] **G. De MICHELI.**  
Synthesis of Control Systems.  
Nota advanced study institute on Logic Synthesis and Silicon  
Compilation for VLSI Design. L'Aquila (Italy) July 7-18, 1986.
- [EZZ86] **T. EZZEDINE, G. SAGNES.**  
Hierarchical Algorithmic Description for Complex Control Parts  
Design.  
ICTC, Liemerick, Ireland, Sept 1986.
- [FLA84] **E. FLAMAND.**  
A Complete And Automatic System For Sequencer Design.  
DAC'84. pp324-329.
- [FLO80] **R.W. FLOYD, J.D. ULLMAN.**  
The Compilation of Regular Expressions into Integrated  
Circuits.  
Proceedings of the 21st annual symposium on the foundations of  
computer science (1980), pp260-269.
- [FOX85] **J.R. FOX, J.A. FRIED.**  
Telecommunications Circuit Design Using The Silicon  
Compiler.  
ICCD'85, pp213-219.
- [FUN86] **H.S. FUNG, S. HIRSCHHORN.**  
An Automatic DFT System For The SILC Silicon Compiler.  
IEEE design & test of computers, February 86, pp45-57.
- [GAJ86] **D.D. GAJSKI, N.D. DUTT, B.M. PANGRLE.**  
Silicon Compilation (Tutorial)  
IEEE 1986 Custom Integrated Circuits Conference, pp102-110.
- [GAJ86a] **D.D. GAJSKI, F.D. BREWER.**  
Towards Intelligent Silicon Compilation.  
Nota advanced study institute on Logic Synthesis and Silicon  
Compilation for VLSI Design. July 7-18 1986.
- [GAJ87] **D.D. GAJSKI.**  
Silicon Compilation (chapter 1).  
Addison Wesley 1987.
- [GER87] **J.P. GERONIMI, A.A. JERRAYA.**  
Architecture des parties contrôle générées par SYCO:  
GEN-TOPO.  
Rapport interne, TIM3/IMAG 1987.
- [GRE86] **D. GREGORY, K. BARTLETT, A. De GUES, G. HACHTEL.**  
SOCRATES: A System For Automatically Synthesising and  
Optimizing Combinational Logic.  
DAC'86. pp79-85.

- [GRO83] **R.R. GROSS.**  
Silicon Compilers: A Critical Survey.  
Dep of computer science, Univ. of north carolina at chapel hill,  
May 1983.
- [GRA79] **J.P. GRAY.**  
Introduction To Silicon Compilation.  
DAC'79, pp305-306.
- [HAF82] **L.J. HAFFER, A.C. PARKER.**  
Automated Synthesis Of Digital Hardware.  
IEEE transactions on computer. February 1992, VOL C-31, No  
2, pp93-109.
- [HAN86] **S. HANRIAT.**  
Synthèse Logique a Base De Regles Pour Les Compilateurs De  
Silicium.  
Thèse de L'INPG. Sep86.
- [HEA86] **S.T. HEALEY, D.D. GAJSKI.**  
Decomposition Of logic Networks Into Silicon.  
22 DAC'86, pp162-168.
- [HIR87] **M. HIRAYAMA.**  
A Silicon Compiler System Based on Asynchronous Architecture.  
IEEE Tran. on CAD, Vol. CAD-6, No. 3, May 1987. pp297-304.
- [JAI87] **R. JAIN, R.W. BRODERSON.**  
Integration of Algorithms In Silicon.  
ESSIRC'87. pp239-244.
- [JAM86] **R. JAMIER.**  
Génération Automatique De Parties Operatives De Circuits VLSI  
De Type Microprocesseur.  
Thèse de docteur Ingenieur de l'INPG. Nov 1986.
- [JAM86b] **R. JAMIER, N. BEKKARA, A. JERRAYA.**  
The Automatic Synthesis of Data Processing Sections.  
ICCD'86. pp64-67.
- [JER86] **A.A. JERRAYA, P. VARINOT, R. JAMIER, B. COURTOIS.**  
Principles of the SYCO Silicon Compiler.  
DAC'86.
- [JER88] **A.A. JERRAYA. & AL.**  
NAUTILE: A physical design Environment based on an object  
oriented data manager.  
1988 IEEE Design Automation Workshop, Arizona.
- [JOE85] **H. JOEPEN, M. GLESNER.**  
Architecture Constructions For A General Silicon Compiler  
System.  
ICCD'85. pp312-315.

- [JOE86] **H. JOEPEN, M. GLESNER.**  
Optimal Structuring of Hierarchical Control-Paths in a Silicon Compiler System.  
ICCAD'86. pp264-267.
- [JOH79] **D. JOHANNSEN.**  
BRISTE BLOCKS: A Silicon Compiler.  
DAC'79, pp310-313.
- [KAR64] **R. KARP.**  
Some Techniques for state Assignment for Synchronous Sequential Machines.  
IEEE Trans. on Elec. Comp., Vol. EC13, Oct. 64, pp507-518.
- [KRE85] **D.E. KREBELBERG, G.E. SOBELMAN, C.S. JHON.**  
Yet Another Silicon Compiler.  
DAC'85, pp176-182.
- [KRE86] **D.E. KREBELBERG, E. SHRAGOWITZ, G.E. SOBELMAN, L. LIN.**  
Automated Layout Synthesis in the YASC Silicon Compiler.  
DAC'86. pp447-453.
- [LAR87] **T. LARSSON.**  
Semantics of a Hardware Specification Language and Related Transformation Rules.  
North Holland, Integration, The VLSI Journal 5 (1987) pp145-158.
- [LAU85] **D. LAURENT, H. NGUYEN.**  
LDS: Langage de Description de Systèmes.  
Internal report, Sycomore project., June 1985.
- [LIP83] **H.M. LIPP.**  
Methodical Aspects of Logical Synthesis.  
Proceedings of the IEEE, Vol.71, No.1, January 1983. pp88-97.
- [MAL85] **Y.K. MALAIYA.**  
Options In Control Implementation.  
ICCD'85, pp267-272.
- [MAR86] **P. MARWEDEL.**  
A New Synthesis Algorithm For The MIMOLA Software System.  
DAC'86, pp271-277.
- [MARI86] **S. MARINE.**  
IRENE: un Langage de description, simulation et synthèse automatique du matériel VLSI.  
Thèse de docteur de l'INPG, Février 1986.
- [MART85] **F. MARTINEZ.**  
Le Compilateur CIRENE.  
Mémoire d'ingénieur CNAM. 1985.

- [MEA80] **C. MEAD, L. CONWAY.**  
Introduction to VLSI Systems.  
Addison-Wesley, 1980.
- [MHA87] **N. MHAYA, A.A. JERRAYA.**  
The SYCO Control Section Compiler.  
ESSIRC'87. pp249-252.
- [MHA88] **N. MHAYA, A.A. JERRAYA, B. COURTOIS.**  
Control Section Synthesis using CPC.  
Rapport de recherche N° RR692 -I-, TIM3-IMAG, Janvier 1988.
- [NAG81] **A.W. NAGLE, A.C. PARKER.**  
Algorithms For Multiple-criterion Design Of Microprogramme  
Control Hardware.  
DAC'81, pp486-493.
- [NAK87] **Y. NAKAMURA.**  
An Integrated Logic Design Environment Based On Behavioral  
Description.  
IEEE trans on CAD. Vol. CAD-6, No. 3, May 1987.
- [NIC86] **M. NICOLAIDIS.**  
An Unified BIST Approach Using Specific Strongly Code  
Disjoint Checker's Design.  
Rapport de recherche N° RR599, Mars 1986.
- [OBR82] **M. OBREBSKA.**  
Efficiency And Performance Comparalson Of Different design  
Methodologies For Control Parts Of Microprocessors.  
North Holland, Microprocessing and Microprogramming 10  
(1982) pp163-178.
- [ORA86] **A. ORAILOGLU, D.D. GAJSKI.**  
Flow Graph Representation.  
DAC'86, pp503-509.
- [PAP85] **C.A. PAPACHRISTOU, D. CORNETT.**  
Generation And Implementation Of State Machine Controllers: A  
VLSI Approach.  
North Holland, Microprocessing and Microprogramming 16  
(1985) pp73-82.
- [PAR86] **A.C. PARKER And Al.**  
MAHA: A Program For Datapath Synthesis.  
DAC'86. pp461-466.
- [PEN86] **Z. PENG.**  
Synthesis Of VLSI Systems With The CAMAD Design Aid.  
DAC'86, pp278-284.
- [RAJ85] **J.V. RAJAN, D.E. THOMAS.**  
Synthesis By Delayed Binding Of Decisions.  
DAC'85, pp367-373.

- [REI86] **R. REIS, A.A. JERRAYA, R. JAMIER.**  
Design of the SYCO6502 using the SYCO silicon compiler.  
ICCD'86.
- [RIN85] **S. RINGO, D. SAVART, J.P. GERONIMI.**  
Conception d'un microprocesseur du marché à l'aide d'un compilateur de silicium.  
Rapport de stage d'ingénieur ENSERG (Juin 1985).
- [RUD85] **R. RUDELL.**  
ESPRESSO. Berkeley CAD Tools User's Manuel. Report No. UCB/CSD 86/272. Dec 1985.
- [RUD86] **R. RUDELL, A. SANGIOVANNI-VINCENTELLI.**  
Exact Minimization of Multiple-Valued Functions for PLA Optimization.  
ICCAD'86. pp352-355.
- [RUD87] **R. RUDELL, A. SANGIOVANNI-VINCENTELLI.**  
Multiple-Valued Minimization for PLA Optimization.  
IEEE Tran. on CAD, Vol. CAD-6, No. 5, September 1987.
- [SAS85] **T. SASAO.**  
HART: A Hardware for Logic Minimization and Verification.  
ICCD'85 pp713-717.
- [SAU86] **G. SAUCIER, S. HANRIAT.**  
Rule Based Logical Synthesis For Silicon Compilers.  
ICCAD'86. pp166-168.
- [SAU87] **G. SAUCIER, M.C. PAULET, P. SICARD.**  
ASYL: A Rule-Based System For Controller Synthesis.  
IEEE trans on CAD. Vol. CAD-6, No.6, Nov 1987. pp1088-1097.
- [SHI83] **S.G. SHIVA**  
Automatic Hardware Synthesis.  
Proceedings Of The IEEE, Vol. 71, No. 1, January 1983.
- [SIS82] **J.M. SISKIND, J.R. SOUTHARD, K.W. CROUCH.**  
Generating Custom High Performance VLSI Designs From Succint Algorithmic Descriptions.  
1982 conference on advanced research in VLSI, M.I.T. JANUARY, pp28-39.
- [SNO78] **F.A. SNOW, D.P. SIEWIOREK, D.E. THOMAS.**  
A Technology-relative Computer-aided Design System: Abstract Representations, Transformations, And Design Tradeoffs.  
DAC'78, pp220-226.
- [SOU83] **J.R. SOUTHARD.**  
MAC-PITTS: An Approach To Silicon Compilation.  
COMPUTER Dec83, pp74-82.

- [SOL86] **J.A. SOLWORTH.**  
GENERIC: A Silicon Compiler Support Language.  
DAC'86, pp524-530.
- [STR86] **C.E. STROUD, R.R. MUNOZ, D.A. PIERCE.**  
CONES: A System for Automated Synthesis of VLSI and Programmable Logic From Behavioral Models.  
ICCAD'86. pp428-431.
- [THO81] **D.E. THOMAS.**  
The Automatic Synthesis Of Digital Systems.  
Proceedings of the IEEE, Vol. 69, No. 10, October 1981.
- [TOR88] **K. TORKI, M. NICOLAIDIS, A.A. JERRAYA, B. COURTOIS.**  
UBIST Version of the SYCO's Control Section Compiler.  
A paraitre à ICCD'88, New York, Octobre 1988.
- [TSE83] **C. TSENG, D.P. SIEWIOREK.**  
FACET: A Procedure For The Automated Synthesis Of Digital Systems.  
DAC'83, pp490-496.
- [TSE84] **C. TSENG, D.P. SIEWIOREK.**  
EMERALD: A Bus Style Designer.  
DAC'84, pp315-321.
- [TSU86] **Y. TSUCHITA and Al.**  
Establishment of Higher Level Logic Design for Very Large Scale Computer.  
DAC'86. pp366-371.
- [VAR87] **P. VARINOT.**  
Compilation De Silicium: Application A La Compilation De Partie Contrôle. Thèse de docteur de l'INPG. Fev 1987.
- [WAL83] **P. WALLISH.**  
Technology'83: Automation (Design/Manufacturing).  
IEEE Spectrum, January 1983. pp55-58.
- [YOS86] **T. YOSHIMURA, S. GOTO.**  
A Rule-Based Algorithmic Approach for Logic Synthesis.  
ICCAD'86. pp162-165.

**TABLE DES MATIERES**  
**ET FIGURES**





## Table des matières

|                                                                                   |    |
|-----------------------------------------------------------------------------------|----|
| Introduction.....                                                                 | 3  |
| <br>Chapitre 1- Introduction à la compilation de silicium                         |    |
| I- Historique.....                                                                | 7  |
| II- Les compilateurs de silicium.....                                             | 10 |
| A- Les compilateurs comportementaux.....                                          | 12 |
| B- Les compilateurs structurels.....                                              | 14 |
| C- Les compilateurs intelligents.....                                             | 14 |
| III- Conclusion.....                                                              | 15 |
| <br>Chapitre 2- Le compilateur de silicium SYCO                                   |    |
| I- Présentation de SYCO.....                                                      | 19 |
| II- Modèle d'interprétation.....                                                  | 19 |
| III- Modèle architectural.....                                                    | 21 |
| IV- Historique de SYCO.....                                                       | 22 |
| V- La version autour du langage LDS (version actuelle).....                       | 25 |
| V-1- Le langage d'entrée de SYCO.....                                             | 25 |
| V-1-1- La description structurelle.....                                           | 25 |
| V-1-2- Définition d'opérateurs complexes et d'environnement<br>de simulation..... | 25 |
| V-1-3- La description comportementale.....                                        | 26 |
| V-2- Organisation de SYCO.....                                                    | 30 |
| V-3- Schéma d'utilisation de SYCO.....                                            | 39 |
| VI- Conclusion.....                                                               | 39 |
| <br>Chapitre 3- Les compilateurs de partie contrôle                               |    |
| I- Introduction.....                                                              | 43 |
| II- Les différentes organisations de parties contrôle.....                        | 44 |
| II-1- Le séquençement conditionnel.....                                           | 45 |
| II-2- La génération des commandes conditionnelles.....                            | 45 |
| II-3- Calcul de l'état suivant.....                                               | 46 |
| II-4- Parties contrôle câblées.....                                               | 46 |
| II-5- Parties contrôle microprogrammées.....                                      | 47 |
| II-5-1- La microprogrammation horizontale.....                                    | 47 |

|                                                                  |     |
|------------------------------------------------------------------|-----|
| II-5-2- La microprogrammation verticale.....                     | 48  |
| II-6- Parties contrôle utilisant des PLAs.....                   | 48  |
| II-6-1- PC à PLA unique.....                                     | 49  |
| II-6-2- PC à PLAs multiple.....                                  | 49  |
| II-7- Parties contrôle utilisant un générateur de temps.....     | 50  |
| II-8- Parties contrôle à plusieurs niveaux d'interprétation..... | 51  |
| III- La forme Intermédiaire des compilateurs des parties         |     |
| contrôle.....                                                    | 52  |
| IV- Transformations de la forme Intermédiaire.....               | 55  |
| V- Génération de la description logique.....                     | 56  |
| V-1- Le choix du code utilisé.....                               | 56  |
| V-2- L'assignation du code aux états.....                        | 57  |
| VI- Conclusion.....                                              | 59  |
| Chapitre 4- CPC: Le compilateur de partie contrôle               |     |
| I- Introduction.....                                             | 63  |
| II- Le langage d'entrée du CPC.....                              | 64  |
| III- Structure d'un étage de contrôle.....                       | 67  |
| IV- Génération de la table d'état.....                           | 68  |
| IV-1- Transformation des opérateurs complexes.....               | 69  |
| IV-2- Transformation de l'opérateur NOT.....                     | 70  |
| IV-3- Transformation de l'opérateur d'inégalité <>.....          | 70  |
| IV-4- Transformation de l'opérateur OR.....                      | 71  |
| IV-5- Le recencement.....                                        | 72  |
| V- Optimisation logique et ajout du branchement implicite.....   | 73  |
| V-1- Optimisation logique.....                                   | 73  |
| V-2- Ajout du branchement implicite.....                         | 78  |
| V-3- Vérification de la tautologie.....                          | 89  |
| VI- Génération de la description de la POP.....                  | 93  |
| VI-1- La partie déclaration.....                                 | 93  |
| VI-2- La partie action.....                                      | 93  |
| VII- Codage des tables d'états.....                              | 95  |
| VII-1- Le choix du code pour chaque champ.....                   | 96  |
| VII-2- L'assignation des états.....                              | 97  |
| VIII- Génération du fichier de communication.....                | 104 |
| IX- L'optimisation dans le CPC.....                              | 106 |
| IX-1- L'optimisation topologique.....                            | 106 |

|                                                             |         |
|-------------------------------------------------------------|---------|
| IX-2-L'optimisation logique.....                            | 109     |
| X- Autres résultats.....                                    | 110     |
| XI- Conclusion.....                                         | 112     |
| <br>Chapitre 5- Perspectives                                |         |
| I- Introduction.....                                        | 115     |
| II-Etage de contrôle microprogrammé.....                    | 115     |
| II-1- Structure de'un étage de contrôle microprogrammé..... | 120     |
| II-2- Structure de la ROM.....                              | 121     |
| II-3- Le PLA d'adressage.....                               | 121     |
| III- Etage de contrôle multi-PLAs.....                      | 122     |
| IV- Conclusion.....                                         | 124     |
| <br>Conclusion.....                                         | <br>127 |
| <br>Références et bibliographie.....                        | <br>133 |

## Figures

|             |                                                                    |     |
|-------------|--------------------------------------------------------------------|-----|
| Figure 1.1  | Diagramme en Y de GAJSKI.....                                      | 11  |
| Figure 2.1  | Modèle d'interprétation.....                                       | 20  |
| Figure 2.2  | Modèle architectural de SYCO.....                                  | 21  |
| Figure 2.3  | Description d'un microprocesseur simple.....                       | 29  |
| Figure 2.4  | Schéma d'appel hiérarchique.....                                   | 30  |
| Figure 2.5  | Organisation de SYCO.....                                          | 31  |
| Figure 2.6  | Différentes classes de Cmodules extraites par EXTRACT.....         | 32  |
| Figure 2.7  | Cmodules générés par EXTRACT.....                                  | 33  |
| Figure 2.8  | Modèle architectural de la POP.....                                | 35  |
| Figure 2.9  | Schéma d'une sous-POP.....                                         | 35  |
| Figure 2.10 | Schéma d'un PLA CMOS à précharge.....                              | 37  |
| Figure 2.11 | Plan de masse de circuits générés par SYCO.....                    | 38  |
| Figure 2.12 | Algorithme général de SYCO.....                                    | 40  |
| Figure 3.1  | Schéma fonctionnel d'une partie contrôle.....                      | 45  |
| Figure 3.2  | Schéma général d'une PC microprogrammée.....                       | 47  |
| Figure 3.3  | PC mono-PLA.....                                                   | 49  |
| Figure 3.4  | PC à deux PLAs.....                                                | 50  |
| Figure 3.5  | PC à générateur de temps.....                                      | 51  |
| Figure 4.1  | Structure d'un étage de contrôle.....                              | 68  |
| Figure 4.2  | Description typique d'un Cmodule.....                              | 79  |
| Figure 4.3  | Graphe de l'automate de la figure 4.2.....                         | 80  |
| Figure 4.4  | Automate non bloquant.....                                         | 81  |
| Figure 4.5  | Graphe de précedence complet correspondant<br>à la figure 4.3..... | 84  |
| Figure 4.6  | Tables d'états correspondant à la figure 2.7.....                  | 88  |
| Figure 4.7  | Vérificateur matériel de tautologie.....                           | 91  |
| Figure 4.8  | Description de la POP.....                                         | 95  |
| Figure 4.9  | PLAs correspondants à la figure 4.6.....                           | 104 |
| Figure 4.10 | Fichier de communication.....                                      | 106 |
| Figure 4.11 | Résultats de ARTS.....                                             | 108 |

|             |                                                                             |     |
|-------------|-----------------------------------------------------------------------------|-----|
| Figure 4.12 | Résultat de ESPRESSO à partir des PLAs<br>de la figure 4.9.....             | 109 |
| Figure 4.13 | Le layout de la partie contrôle du 6502.....                                | 111 |
| Figure 5.1  | Construction de $A_S$ .....                                                 | 116 |
| Figure 5.2  | Illustration du passage du modèle de Mealy à celui<br>de Moore.....         | 117 |
| Figure 5.3  | Automate de Mealy.....                                                      | 117 |
| Figure 5.4  | Automate de Moore équivalent à l'automate de Mealy<br>de la figure 5.3..... | 118 |
| Figure 5.5  | Table d'états (forme de Moore).....                                         | 119 |
| Figure 5.6  | Etage de contrôle microprogrammé.....                                       | 121 |
| Figure 5.7  | Etage de contrôle multi-PLAs.....                                           | 124 |



**A U T O R I S A T I O N de S O U T E N A N C E**

VU les dispositions de l'article 15 Titre III de l'arrêté du 5 juillet 1984 relatif aux études doctorales

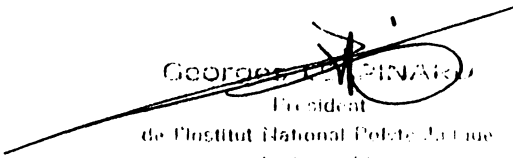
VU les rapports de présentation de Messieurs

- . C. LANDRAULT, Directeur de recherche
- . G. NOGUEZ, Professeur

**Monsieur MHAYA Nouredine**

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme  
de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité  
"Informatique"

Fait à Grenoble, le 16 juin 1988

  
Georges LEVIGNAR  
Président  
de l'Institut National Polytechnique  
de Grenoble



