Génération automatique de parties opératives de circuits VLSI de type microprocesseur

Robert Jamier

THESE

présentée à

l'Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR-INGENIEUR
en informatique

par

Robert JAMIER

Génération automatique de parties opératives de circuits VLSI de type microprocesseur

Date de soutenance : 28 novembre 1986

Jury :

Mr. G. Mazaré	Président
Mr. F. Anceau	Examinateurs
Mr. B. Courtois	
Mr. A. Jerraya	
Mr. J-P. Moreau	
Mr. R. Plouhinec	

Thèse préparée au sein du laboratoire IMAG/TIM3

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Daniel BLOCH

Année 1987

Vice - Présidents : René CARRE
Jean-Marie PIERRARD

Professeurs des Universités

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

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

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

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

Chercheurs du C.N.R.S

Directeurs de recherche 1ère Classe

CAILLET Marcel
CARRE René
FRUCHART Robert
JORRAND Philippe
LANDAU Ioan
MARTIN

Directeurs de recherche 2ème Classe

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

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

E.N.S.E.E.G

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

E.N.S.E.R.G

BOREL Joseph
CHOVET Alain
DOLMAZON Jean-Marc
HERAULT Jeanny

E.N.S.I.E.G

DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G

BOIS Daniel
DARVE Félix
MICHEL Jean-Marie
ROWE Alain
VAUCLIN Michel

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

BERT Didier
COURTIN Jacques
COURTOIS Bernard
DELLA DORA Jean
FONLUPT Jean
SIFAKIS Joseph

E.F.P.G

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

DEMOULIN Eric
DEVINE
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

Directeur : Monsieur M.MERMET
Directeur des Etudes et de la formation: Monsieur J. LEVASSEUR
Directeur des recherches : Monsieur J. LEVY
Secrétaire Général : Mademoiselle M. CLERGUE

PROFESSEURS DE 1ère CATEGORIE

COINDE Alexandre	Gestion
GOUX Claude	Métallurgie
LEVY Jacques	Métallurgie
LOWYS Jean-Pierre	Physique
MATHON Albert	Gestion
RIEU Jean	Mécanique-Résistance des matériaux
SOUSTELLE Michel	Chimie
FORMERY Philippe	Mathématiques Appliquées

PROFESSEURS DE 2ème CATEGORIE

HABIB Michel	Informatique
PERRIN Michel	Géologie
VERCHERY Georges	Matériaux
TOUCHARD Bernard	Physique Industrielle

DIRECTEUR DE RECHERCHE

LESBATS Pierre	Métallurgie

MAITRE DE RECHERCHE

BISCONDI Michel	Métallurgie
DAVOINE Philippe	Géologie
FOURDEUX Angeline	Métallurgie
KOBYLANSKI André	Métallurgie
LALAUZE René	Chimie
LANCELOT Francis	Chimie
LE COZE Jean	Métallurgie
THEVENOT François	Chimie
TRAN MINH Canh	Chimie

Personalités habilitées à diriger des travaux de recherche

DRIVER Julian	Métallurgie
GUILHOT Bernard	Chimie
THOMAS Gérard	Chimie

Professeurs à l'UER de Sciences de Saint-Etienne

VERGNAUD Jean-Maurice	Chimie des Matériaux et Chimie Industrielle

Je tiens à remercier :

Mr. le Professeur François Anceau, Chef de la division architecture pour l'intelligence artificielle à Bull, pour m'avoir accueilli dans son laboratoire et pour avoir bien voulu me faire l'honneur de participer au jury de cette thèse. Monsieur Anceau est aussi à l'origine de l'idée principale de ce document.

Mr. Bernard Courtois, Directeur du laboratoire IMAG/TIM3 qui m'a permis à la fois de continuer le travail que j'avais commencé et de m'inscrire en thèse.

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

Mr. Guy Mazaré, professeur à l'ENSIMAG, de me faire l'honneur de présider le jury de cette thèse.

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

Mr. René Plouhinec, chargé des problèmes de la CAO d'avenir, de circuits intégrés à Electronique Serge Dassault, pour avoir accepté d'être membre de ce jury.

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

Anne-Marie Molle et Sandrine Bonelli pour leur participation efficace à la frappe de ce document et à la saisie des dessins.

Le service de reprographie de l'IMAG pour avoir assuré le tirage de cette thèse.

A mes parents,
A mes amis.

Résumé

Le compilateur de parties opératives Apollon qui est présenté dans cette thèse, génère automatiquement le dessin des masques de parties opératives de circuits VLSI de type microprocesseur à partir d'une description comportementale de niveau transfert de registres constituée d'un ensemble non ordonné d'instructions opératives. Une instruction opérative est formée d'un ensemble d'actions opératives dont le format est prédéfini (transferts - opérations unaires ou binaires et entrées-sorties) devant se dérouler en parallèle en au plus deux cycles opératifs. Un cycle opératif comprend 4 phases qui correspondent aux 4 phases d'exécution d'un transfert entre 2 registres.

Apollon est basé sur un modèle dérivé de la partie opérative du MC68000. Ce modèle fournit à la fois :

- un modèle architectural : la partie opérative est formée d'un ensemble de sous parties opératives alignées à deux bus qui traversent tous les éléments d'une sous partie opérative.
- un modèle temporel : une opération prend 2 cycles, un transfert un seul.
- un modèle électrique : les bus sont complémentés et à précharge.
- un modèle topologique : le plan de masse est basé sur la structure en tranches appelée communément "bit slice".

Le compilateur génère d'abord l'architecture de la partie opérative, puis les spécifications des masques à partir de cette architecture. Pour générer l'architecture de la partie opérative en un temps raisonnable, le compilateur doit recourir à des heuristiques. Pour générer le dessin des masques, le compilateur utilise l'assembleur de silicium Lubrick qui permet d'assembler et de connecter automatiquement les cellules de base des éléments fonctionnels de la partie opérative. Les spécifications des masques sont générées à partir des spécifications des cellules prédéfinies d'une bibliothèque NMOS.

Mots-clés

Synthèse automatique du matériel, partie opérative, microprocesseur, VLSI, architecture, parallélisme, dessin des masques, compilateur de silicium.

Summary

The Apollon data processing section compiler, which is presented in this thesis, produces automatically the layout of data processing sections of VLSI microprocessor-like circuits. Apollon starts from an RTL behavioral description which is made of a set of unordered operative instructions. An operative instruction consists of a set of operative actions, the format of which is predefined (transfers, unary or binary operations, data input or output), which are to be executed simultaneously within at most two operative cycles. An operative cycle is divided into 4 phases which correspond to the 4 phases of a transfer between two registers.

Apollon is based on a model which is derived from the data processing section of the MC68000. This model provides simultaneously for :

- an architectural model : the data processing section is partitioned in a set of aligned subsections. Two buses run through all the elements of a subsection.
- a temporal model : an operation takes two cycles, a transfer one.
- an electric model : the 2 buses are complemented and precharged.
- a topological model : the floorplan is based on the bit slice structure.

The compiler produces first the architecture of the data processing section, then the layout from this architecture. To synthesize the architecture in a reasonable period of time, the compiler has to use heuristics. The compiler makes use of the Lubrick silicon assembler to assemble and to connect automatically the basic cells of the functional elements of the datapath. The layout results from the assemblage of the layout of predefined cells, which belong to an NMOS library.

Keywords

Automatic hardware synthesis, datapath, microprocessor, VLSI, architecture, parallelism, layout, silicon compiler.

INTRODUCTION

Introduction

Les performances des ordinateurs sont dues en bonne partie aux progrès de la technologie des circuits intégrés. A l'heure actuelle ce sont les ordinateurs qui à leur tour sont utilisés pour créer des circuits plus performants. Le vieux rêve des ordinateurs s'"autoreproduisant" pourrait devenir réalité avec les compilateurs de silicium.

Les compilateurs de silicium sont nés à la suite d'une double évolution : d'une part la surface des puces fabriquées augmente exponentiellement et d'autre part les technologies de fabrication continuent à s'affiner de sorte que le nombre de transistors que l'on peut intégrer sur un seul circuit est actuellement de l'ordre du million. La complexité et le coût de la conception d'un circuit rendent difficile la production de circuits spécialisés. La seule solution viable consiste à automatiser la conception. Il existe actuellement de nombreux outils de CAO qui permettent d'automatiser la plupart des étapes de la conception. Les compilateurs de silicium ont pour but d'automatiser l'ensemble des étapes de la conception.

Dans cette thèse, nous présentons la réalisation d'un système qui permet de générer les spécifications physiques de la partie opérative d'un circuit à partir d'une description de haut niveau des instructions que doit exécuter le circuit.
Cette thèse se compose de 3 chapitres.

Dans le premier chapitre, nous établissons une classification des compilateurs de silicium et nous présentons le compilateur de silicium SYCO réalisé dans l'équipe d'architecture des ordinateurs d'IMAG/TIM3. SYCO génère le dessin des masques des circuits de type microprocesseur et utilise le compilateur de parties opératives dont nous parlons en détail au chapitre 3 pour générer le dessin des masques de la partie opérative. Le travail présenté au chapitre 1 est le travail d'une équipe. Je tiens à remercier ici tous les membres de l'équipe : A. Jerraya, N. Bekkara, E. Bourcier, F. Martinez, N. Mhaya, K. Torki et P. Varinot qui m'ont fourni le matériel nécessaire à l'écriture de ce chapitre.

Le chapitre 2 est consacré à la compilation de parties opératives. Nous présentons d'abord les modèles de parties opératives utilisées dans les outils de synthèse de parties opératives existants. Puis les problèmes de synthèse de matériel en ce qui concerne la partie opérative sont abordés.

Le chapitre 3 est consacré à la présentation du compilateur Apollon. Apollon génère le dessin des masques de la partie opérative à partir d'une description comportementale des instructions opératives. Une instruction opérative est composée d'une ou plusieurs actions opératives de base à exécuter en parallèle. Le compilateur est basé sur modèle dérivé de la partie opérative du MC68000 : architecture 2 bus et structure "bit slice". La compilation s'effectue en 2 étapes. Nous présentons d'abord le système globalement, nous détaillons ensuite la génération de l'architecture puis la génération des masques. Enfin nous présentons quelques exemples de compilation. Le compilateur n'a pu être réalisé que grâce à l'acquis de toute l'équipe. En particulier j'ai utilisé :

- le langage de conception de circuits intégrés LUCIE,
- l'assembleur de silicium du système LUCIE, LUBRICK,

- les cellules de la bibliothèque NMOS de l'équipe.

Je remercie en outre tous ceux qui ont participé au développement d'Apollon : C. Arzounian, L. Harivel, N. Bekkara et A. Jerraya.

CHAPITRE II

LE COMPILATEUR SYCO

1. Classification des compilateurs existants

Le terme compilation de silicium a été introduit pour la première fois par Dave Johannsen au California Institute of Technology en 1979. Il l'utilisait pour désigner l'assemblage paramétré de cellules d'une bibliothèque de cellules précaractérisées. Depuis ce terme a été très largement repris. Les compilateurs de silicium sont de plus en plus populaires.

Au lieu de transformer un programme écrit dans un langage de haut niveau en un langage directement compréhensible par la machine sur laquelle doit être exécuté le programme comme le fait un compilateur de logiciels, un compilateur de silicium transforme les spécifications de haut niveau d'un circuit intégré en les spécifications des masques de fabrication du circuit.
Si tous les compilateurs de silicium existant sur le marché ou dans les laboratoires de recherche génèrent des spécifications de même niveau (masques), ils ne partent pas du même niveau ni ne génèrent le même type de circuits.
On trouve actuellement une grande variété de compilateurs de silicium. Ces compilateurs peuvent être classés selon le type des circuits générés ou selon le niveau du langage d'entrée du compilateur.
Les circuits intégrés peuvent être classés en trois groupes selon le style de conception adopté :

- les circuits prédiffusés,
- les circuits précaractérisés,
- et les circuits organisés à la demande.

Ces trois groupes correspondent à trois catégories de compilateurs. On peut dire que le problème est partiellement résolu pour les deux premiers groupes. Par contre on commence juste à trouver des compilateurs qui génèrent des circuits du troisième groupe. Nous nous intéresserons dans le cadre de cette thèse uniquement aux circuits de ce dernier groupe.

De nombreux travaux sur la compilation de silicium de circuits organisés à la demande ont été publiés. Notre but n'est pas ici de faire une revue exhaustive [GROS 83] et [GAJS 86] recensent la plupart des compilateurs existants et établissent une classification.
[JERR 86] distingue 2 types de compilation selon le niveau de description du circuit:

- la compilation structurelle
- la compilation comportementale

Un compilateur de type structurel génère les spécifications des masques à partir de la structure du circuit, c'est à dire à partir de la décomposition du circuit sous forme de blocs interconnectés [BRAY 85], [JOHA 79], [SHRO 82], [BERG 84].
Au contraire un compilateur de type comportemental permet de générer les masques directement à partir de la description comportementale, c'est à dire de l'algorithme d'interprétation des instructions du circuit [BLAC 85], [SISK 82], [JERR 86].

La plupart des compilateurs développés actuellement sont basés sur un modèle. Ce modèle peut être un modèle

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

Les compilateurs de type comportemental font tous appel à un modèle architectural cible. Ce modèle cible restreint la généralité du compilateur mais en contrepartie en assure la faisabilité. Les compilateurs de type structurel peuvent eux aussi aussi être basés sur un modèle mais sur un modèle topologique, par exemple la structure bit slice [SHRO 82]. L'originalité du système SYCO [JERR 86] réside dans le fait qu'il est basé sur un modèle à la fois architectural et topologique. Ce choix restreint la portée du compilateur puisque l'organisation fonctionnelle et topologique du circuit est prédéfinie mais il permet en échange d'utiliser des algorithmes simples et efficaces pour la génération des masques.
Les compilateurs de circuits dont le plan de masse n'est pas prédéfini doivent utiliser des outils de placement et de routage sophistiqués. Une partie importante du circuit peut être occupée par les connexions entre les différents blocs.

Une nouvelle génération de compilateurs de silicium semble émerger [BREW 86], [KNAP 86]. Il s'agit de systèmes beaucoup plus généraux que les précédents. Ces systèmes sont constitués d'un grand nombre d'outils de type divers (synthèse - évaluation - simulation - optimisation etc...) qui sont pilotés par un système expert. C'est ce système qui remplacerait le concepteur lorsqu'il doit faire des choix critiques et qu'il ne dispose que d'un poste de CAO intégré.

2. Présentation du compilateur SYCO

2.1. Caractéristiques générales

Le système SYCO, [JERR 86] développé dans le groupe architecture des ordinateurs du laboratoire IMAG/TIM3 est un compilateur de silicium pour des circuits intégrés à très haute densité de type microprocesseur. SYCO génère les spécifications des masques d'un circuit constitué d'une partie contrôle et d'une partie opérative à partir de sa description comportementale dans un langage de description de circuits intégrés: LDS [LAUR 85].

Le compilateur utilise une **architecture cible**.

L'architecture du circuit est basée sur un modèle d'interprétation à plusieurs niveaux. Le circuit est constitué d'un ensemble de machines emboîtées les unes dans les autres dont seule la dernière possède une partie opérative. La machine M1 interprète les instructions du microprocesseur à l'aide d'une machine M2, qui utilise elle même une machine M3 qui interprète les nanoinstructions de la machine M2. La machine M3 possède éventuellement une partie opérative qui exécute les picoinstructions provenant de la décomposition des nanoinstructions.

La description de l'algorithme d'interprétation des instructions de la machine peut être représentée par un arbre d'appels de procédures. Chaque appel de procédure est interprété comme l'exécution d'une séquence d'instructions par la partie contrôle de niveau immédiatement inférieur. Le nombre de parties contrôle imbriquées est donc égal au niveau maximum d'imbrication des procédures de la description augmenté de 1.

La partie opérative est constituée d'un ensemble de sous parties opératives alignées. Tous les éléments de la partie opérative sont traversés par 2 bus qui peuvent être segmentés. Les éléments qui sont traversés par les mêmes segments de bus forment une sous partie opérative. Le plan de masse de la partie opérative est basé sur la structure en tranches appelée couramment "bit-slice".

Chaque niveau d'interprétation de la partie contrôle de la machine globale est constitué d'un bloc rectangulaire appelé tranche de contrôle. Le circuit est constitué d'un bloc rectangulaire résultant de l'empilement de plusieurs tranches de contrôle et de la tranche de traitement. SYCO fait appel à des modules générateurs de spécifications de masques spécialisés pour générer les spécifications des masques des différentes tranches. Les tranches sont ensuite assemblées automatiquement.

Avant de présenter les différents modules de SYCO, étudions plus en détail le

modèle de circuit généré par SYCO.

2.2. Modèle de compilation

2.2.1. Architecture du circuit

2.2.1.1. Architecture de la partie contrôle

Architecture globale

La partie contrôle d'un circuit généré par SYCO est constituée d'une hiérarchie d'automates en cascade : les automates de niveau i reçoivent en entrée les sorties d'automates de niveau i-1 et les comptes rendus de la partie opérative; les sorties des automates de niveau i constituent les entrées des automates de niveau i+1. A chaque procédure de niveau i est associée un automate de niveau i.

Le premier et le dernier étages de contrôle jouent un rôle spécial. Le premier étage n'est constitué que d'un seul automate qui assure le séquencement global du circuit. En outre le premier étage ne reçoit pas en entrée les sorties d'un autre étage. Les sorties du dernier étage constituent les commandes de la partie opérative.

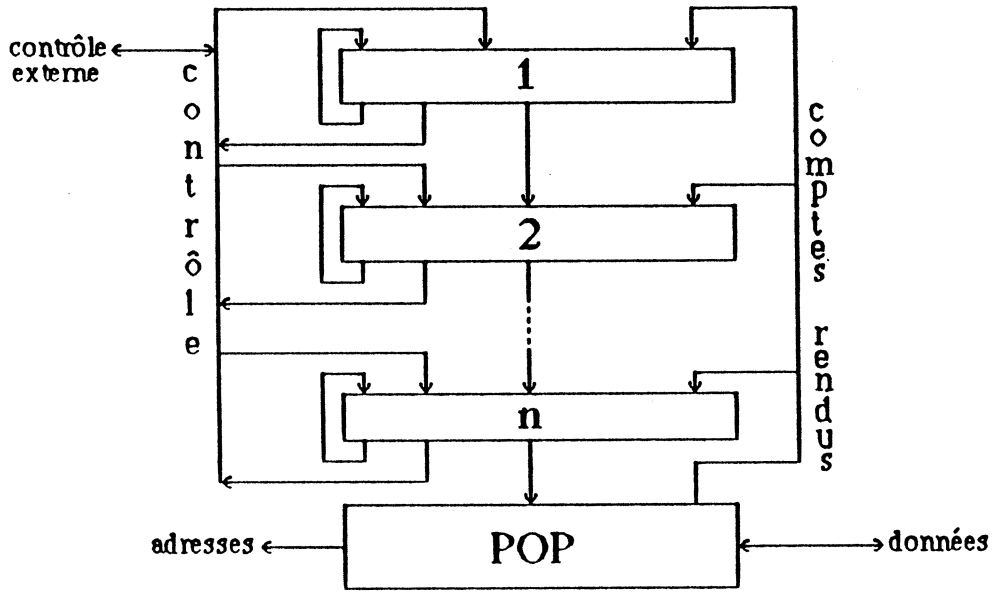


figure 1.1 : modèle architectural de SYCO

Les entrées d'un étage de contrôle sont constituées :

- des commandes de niveau supérieur,
- des commandes de séquencement interne,
- des comptes rendus de la partie opérative,
- de certains champs du registre instruction,
- de requêtes de contrôle externes,

- et enfin de variables de contrôle internes.

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

En effet SYCO permet l'emploi de procédures paramétrées, ce qui est d'ailleurs très pratique car cela permet d'éviter d'écrire plusieurs versions légèrement différentes d'une même procédure. A chaque paramètre, on associe une variable de contrôle. Ces variables sont réalisées à l'aide de bascules ou de registres selon que les paramètres sont booléens ou non, et sont placées dans la partie contrôle. Elles peuvent éventuellement être intégrées dans un chemin de données de type bit-slice à l'intérieur de la partie contrôle au dessus des étages de contrôle.

Les registres contenant les valeurs des paramètres des procédures sont placés dans la partie contrôle pour 2 raisons :

- les paramètres des procédures ont souvent des valeurs booléennes ou des valeurs qui ne nécessitent qu'un faible nombre de chiffres binaires. Le placement des registres de paramètres dans la partie opérative qui traite des mots de longueur très supérieure n'est pas souhaitable car cela conduirait à l'apparition de trous dans la structure bit-slice et donc à une mauvaise utilisation de la surface de silicium.
- d'autre part le placement de ces registres dans la partie contrôle permet d'affecter directement les paramètres. En effet dans une partie contrôle multiniveau de type SYCO, une commande de transfert dans la partie opérative doit traverser tous les étages de la hiérarchie avant que la commande suivante de même niveau ne puisse être exécutée. Par contre un étage de contrôle peut agir directement sur les variables de contrôle.

Les sorties d'un automate de contrôle sont constituées:

- du nouvel état de la procédure en cours d'exécution,
- des commandes de l'étage immédiatement inférieur,
- de signaux de synchronisation des tranches de contrôle,
- ainsi que des nouvelles valeurs et des commandes d'écriture pour toutes les variables de contrôle modifiées à ce niveau.

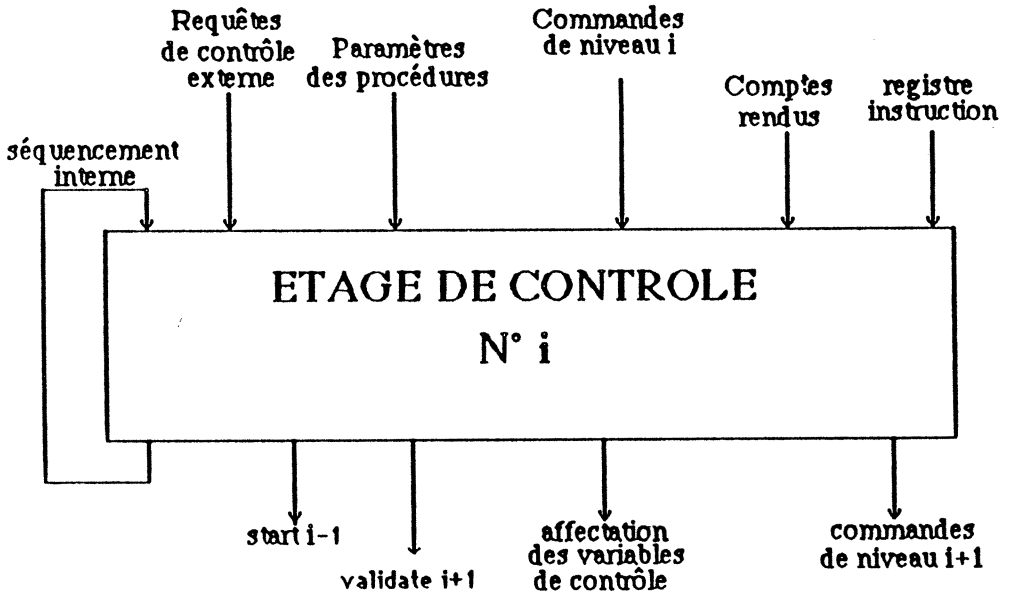


figure 1.2 : entrées et sorties d'un étage de contrôle

L'approche multiniveau permet d'obtenir une partie contrôle dense. Les instructions d'une procédure ne sont mémorisées qu'une seule fois. L'utilisation de procédures paramétrées permet de diminuer davantage la surface de la partie contrôle. Toutefois ce n'est pas la principale raison pour laquelle ce modèle a été choisi. On peut par exemple utiliser une partie contrôle microprogrammée à un seul niveau avec une pile permettant d'empiler les adresses de retour après l'exécution des procédures [OBRE 82], [NAGL 82].

Une partie contrôle multiniveau a été choisie essentiellement pour des raisons topologiques. En effet, un des gros problèmes de la génération des spécifications des masques de la partie contrôle a pour origine l'irrégularité (d'un point de vue géométrique) des parties contrôle. Le découpage d'une partie contrôle en tranches de contrôle permet d'obtenir des masques uniquement par empilement des différentes tranches de la même façon que l'on obtient les maques de la partie opérative en empilant les différentes tranches d'un bit. Il y a cependant une différence importante. Les tranches de la partie opérative ont toutes la même largeur, ce qui n'est pas le cas des tranches de contrôle en général.

La largeur d'une tranche de contrôle augmente avec le niveau de la tranche [GERO 85], [MHAY 86]. Plus la tranche de contrôle est proche de la partie opérative, plus elle a tendance à s'élargir.

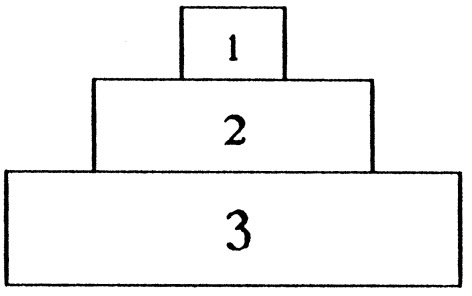


figure 1.3 : effet pyramide

Cet effet "pyramide" peut s'expliquer de la manière suivante. Si chaque procédure appelle plusieurs autres procédures et si les procédures de niveau i+1 appelées par les procédures de niveau i sont toutes différentes entre elles, alors le nombre de branches à chaque niveau de l'arbre des appels de procédures ne peut qu'augmenter. En effet une commande de niveau i ne peut actionner que la machine de niveau i+1, les différentes tranches traversées n'étant pas transparentes aux commandes. De toute façon, même si l'on diposait de blocs transparents, ces commandes directes contribueraient à l'élargissement des tranches traversées.
Le nombre de branches de l'arbre des appels n'augmente pas avec la profondeur si aucune procédure ne fait appel à plus d'une procédure de niveau inférieur ou alors si les mêmes procédures de niveau inférieur sont appelées plusieurs fois.
En général une procédure est appelée plusieurs fois, (c'est d'ailleurs même la principale raison pour laquelle on utilise des procédures : éviter de réécrire plusieurs fois la même séquence d'instructions) et les mêmes instructions opératives sont souvent utilisées plusieurs fois, mais il n'en reste pas moins qu'un programmeur adopte le plus souvent une approche hiérachique, ce qui conduit d'une part à l'effet pyramide et d'autre part à un nombre élevé de tranches de contrôle.
C'est pour cette raison qu'il est nécessaire de transformer [BEKK 86] la description d'origine, afin de diminuer le nombre de tranches de contrôle et de limiter cet effet pyramide. Cet effet ne peut être que limité car la largeur de la dernière tranche de contrôle est fixe, elle est imposée par le nombre des commandes de la partie opérative.

Architecture d'un étage de contrôle

Le schéma d'un étage de contrôle de niveau i est donné figure 1.2.
Il existe de nombreuses architectures possibles pour réaliser une partie contrôle [OBRE 82]. On peut classer ces architectures selon plusieurs critères [MALA 85] :

- type de l'automate de contrôle : Moore-Mealy.
- structure d'accueil de la partie contrôle : logique aléatoire - structures régulières (ROMs, PLAs ou SLAs).
- organisation fonctionnelle : on peut distinguer schématiquement 2 types d'approche :

- l'approche la plus utilisée consiste à générer les commandes microinstruction par microinstruction : il existe plusieurs façons de générer les commandes microinstruction par microinstruction :

. soit on associe à chaque état de l'automate une bascule. Chaque commande de la partie opérative est calculée par un circuit combinatoire dont les entrées sont constituées par les sorties des bascules correspondant aux états dans lesquels cette commande doit être activée et par des conditions d'activation si on réalise un automate de Mealy. Le passage d'un état à un autre se fait en reliant la sortie d'une bascule correspondant à un état aux entrées des bascules correspondant à ses successseurs. En cas de successeurs multiples, la liaison est validée par les conditions d'activation. Cette organisation fonctionnelle est utilisée généralement pour réaliser des parties contrôle câblées.

. soit on adopte une approche de type microprogrammé. L'algorithme est décrit sous la forme d'un microprogramme qui est enregistré typiquement dans une mémoire morte. L'adresse de la microinstruction à exécuter est fournie par un organe de séquencement qui doit prendre en compte les requêtes externes, les comptes rendus de la partie opérative et bien sûr la nature de l'instruction en cours d'exécution. Un automate d'états finis microprogrammé est par principe de type Moore.

. soit on utilise des PLAs : on peut distinguer 2 cas. On peut générer simultanément les commandes de la partie opérative et le nouvel état de l'automate et on n'utilise alors qu'un PLA ou alors on génère séparément le nouvel état et les commandes et on a besoin de 2 PLAs. Dans les 2 cas, il est possible de réaliser des automates de Moore et de Mealy. Toutefois la première solution se prête mieux à la réalisation d'un automate de Mealy.

- une deuxième approche consiste à générer en bloc les microinstructions d'une instruction ou d'une séquence de microinstructions ou du moins des informations communes à toutes les microinstructions d'une instruction.

- dans le premier cas on utilise un générateur de temps : on associe à une macroinstruction ou simplement à une séquence de microinstructions selon que la partie contrôle est à un seul ou à 2 niveaux, les commandes de toutes les microinstructions. Les commandes d'une microinstruction sont validées par un générateur de temps qui peut être un PLA ou un simple compteur si toutes les macroinstructions ou toutes les séquences ont le même nombre de microinstructions.

- dans le deuxième cas on cherche à extraire :

. soit des propriétés communes à toutes les microinstructions d'une instruction, qui ont une influence sur le séquencement,

. soit des propriétés qui ont une influence sur la génération des commandes : on parle alors de paramétrisation et on doit se ramener impérativement à un automate de Mealy.

La recherche de propriétés ou de paramètres peut d'ailleurs être combinée avec les autres approches citées si dessus.

Il faudrait en fait citer toutes les architectures existantes. Une dizaine d'architectures différentes sont comparées dans [OBRE 82]. Il n'y a pas de solution universelle. Le choix d'une solution dépend de la nature de l'algorithme et des performances requises.
La solution qui a été programmée dans SYCO est l'architecture mono-PLA. D'autres architectures sont également proposées dans [JERR 86] et [VARI 86] : partie contrôle microprogrammée et partie contrôle à générateur de temps.
La solution retenue actuellement a le mérite d'être simple et de pouvoir, être implantée très facilement si l'on dispose d'un générateur de PLAs optimisés. [CHUQ 84].
La matrice ET décode l'instruction, les comptes rendus de la partie opérative, les requêtes de contrôle externes (reset, interruptions), les paramètres des procédures et l'état de l'automate. La matrice OU génère les sorties et en particulier les commandes de la partie opérative ou les commandes de la partie contrôle de niveau inférieur et le nouvel état de l'automate simultanément. L'architecture mono PLA permet donc de réaliser naturellement des automates de Mealy. On peut toutefois réaliser des automates de Moore mais à l'instant t on ne génère pas les commandes de la partie opérative à l'instant t mais à l'instant t-1 puisqu'il faut déjà avoir généré l'état.

2.2.1.2. Architecture de la partie opérative

Nous ne présentons pas ici en détail l'architecture de la partie opérative puisqu'elle sera longuement étudiée dans le reste de la thèse. Précisons toutefois que les éléments de la partie opérative ne peuvent être connectés entre eux que par l'intermédiaire de 2 bus parallèles qui traversent toute la partie opérative. Ces 2 bus peuvent cependant être morcelés, ce qui permet d'exécuter des opérations en parallèle si c'est nécessaire. L'utilisation d'un nombre limité de bus restreint les possibilités de parallélisme mais permet en contrepartie d'obtenir une partie opérative très dense.

2.2.2. Séquencement

L'horloge de base du circuit est décomposée en 4 phases désignées respectivement par T1, T2, T3 et T4. Ces phases correspondent aux phases d'exécution d'un transfert dans la partie opérative.
Le temps d'exécution d'une tranche de contrôle est d'un cycle. En T1 le registre d'entrée est transparent, en T4 le registre de sortie est transparent.

Nous étudierons 2 types de modèle de séquencement : un modèle de base où un seul étage est actif à la fois et un modèle accéléré où tous les étages peuvent fonctionner en même temps.

2.2.2.1. Modèle de base

A chaque cycle, une seule tranche est active. Lorsqu'une tranche de contrôle lance l'exécution d'une procédure, elle doit attendre que l'exécution de cette procédure par la tranche inférieure soit terminée avant de lancer l'exécution d'une autre procédure ou de repasser le contrôle à la tranche supérieure.
Ce mécanisme de synchronisation est évidemment le plus simple. Pour le mettre en œuvre, il suffit de disposer de n-1 signaux d'activation de l'étage supérieur ("start"), de n-1 signaux d'activation de l'étage inférieur ("validate") et d'un reset pour une machine à n étages.
Décrivons le fonctionnement d'une machine à 3 étages.
Nous donnerons d'abord les caractéristiques générales de fonctionnement avant d'entrer dans les détails.
Nous supposons que lors d'une phase d'initialisation (Reset), les entrées et les sorties des différentes tranches ont été remises à zéro.
Au premier cycle la tranche 1 est activée. En T4 le nouvel état et les sorties de l'étage 1 sont disponibles.
Au cycle 2, pendant la phase T1 l'entrée de l'étage 2 prend la valeur de la sortie de l'étage 1.
Au cycle 3, c'est l'étage 3 qui travaille. A la fin de ce cycle, les commandes de la partie opérative sont disponibles en sortie. Ce n'est qu'au cycle 4 que la partie opérative commence à travailler. En même temps l'étage 3 génère de nouvelles commandes pour le cycle suivant et si l'exécution de l'instruction s'accomplit en 2 cycles, la dernière tranche de contrôle envoie un signal de fin d'exécution à la tranche supérieure (start 2) ainsi qu'un signal de remise à zéro des entrées de la tranche 3 de façon à ce que le registre de séquencement interne soit remis à zéro ainsi que le registre de "séquencement externe", ce qui permet de générer au cycle suivant l'instruction NOP : aucune opération.
Ce mécanisme de synchronisation est simple, mais n'est pas très efficace car on doit souvent générer des instructions NOPs.

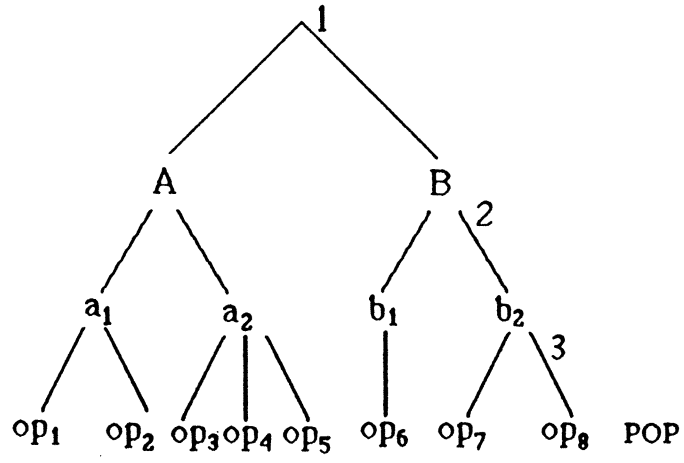


figure 1.4 : arbre d'appels de procédures

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
étage	1	2	3	3	2	3	3	3	2	1	2	3	2	3	3	
actions de contrôle	A	a_1	op_1	op_2 start 2	a_2	op_3	op_4	op_5 start 2	start 1	B	b_1	op_6 start 2	b_2	op_7	op_8	
action opérative				op_1	op_2		op_3	op_4	op_5				op_6		op_7	op_8

figure 1.5 : modèle temporel de base

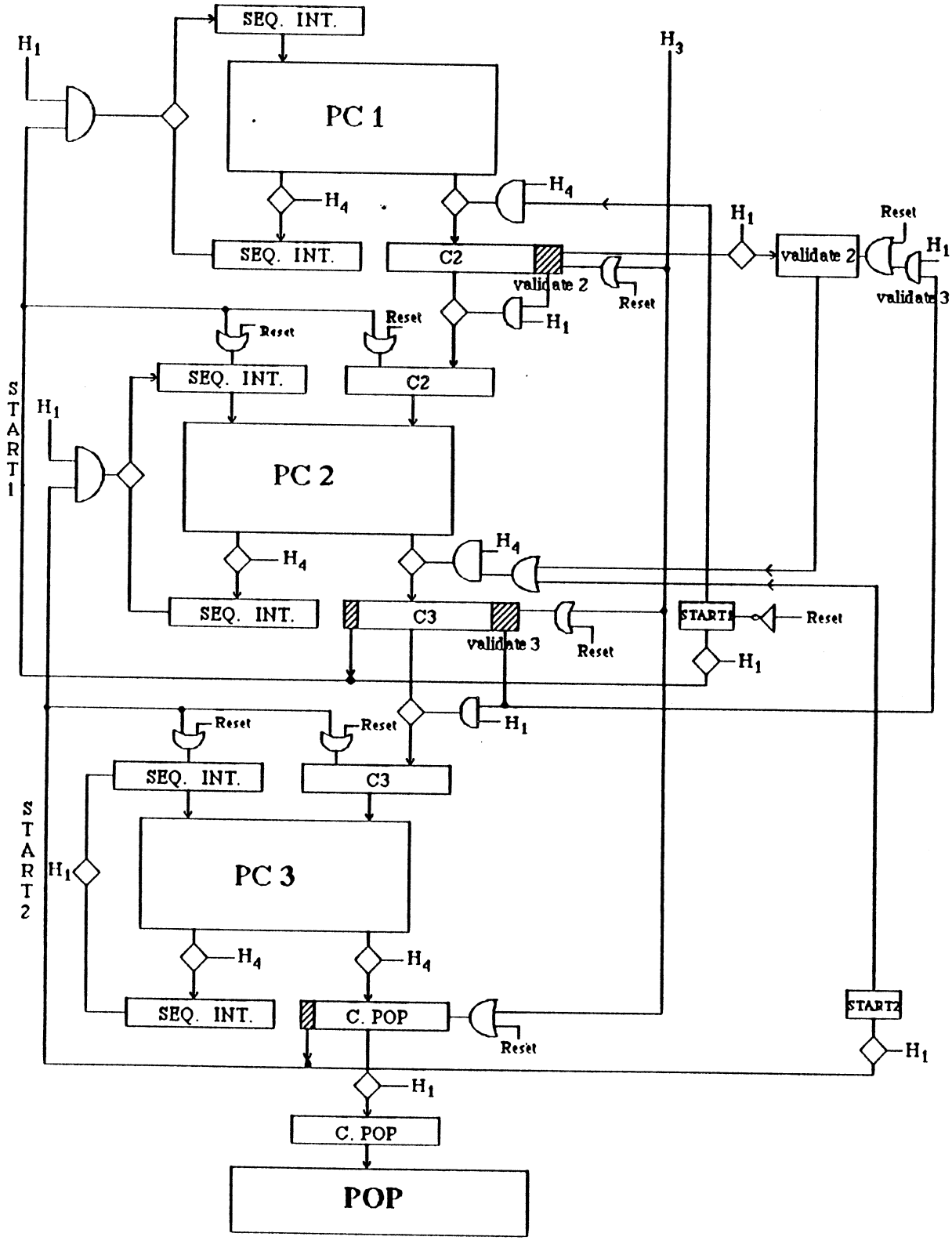


figure 1.6 : réalisation du modèle temporel de base

Détaillons maintenant le fonctionnement de cette machine à 3 étages.

Initialisation :

On suppose au départ que tous les registres sont remis à zéro sauf start1 qui est remis à 1.

1° cycle :

Au premier cycle, la tranche 1 est activée. En T4 le nouvel état et les sorties de l'étage 1 sont disponibles. Le registre de sortie est transparent si start1 vaut 1.

2° cycle :

En T1 du deuxième cycle, le registre d'entrée du deuxième étage est transparent car validate2 a été positionné à 1 au cycle précédent. En même temps validate2 est stocké dans une bascule. En T3 les registres de sortie des étages de contrôle sont remis à zéro, seul le registre de sortie de PC1 a alors un contenu non nul. Cette remise à zéro sera effectuée systématiquement à chaque cycle. En T4 du deuxième cycle, les résultats du deuxième étage sont mémorisés dans le tampon de sortie car validate2 a été positionné à 1 au début du cycle.

3° cycle :

En T1 du troisième cycle, la bascule contenant validate2 est remis à zéro par validate3 de façon à ce que la sortie de PC2 ne soit mémorisée que si PC1 a été effectivement redéclenchée. En même temps les sorties de PC2 sont disponibles en entrée de PC3. En T4 du troisième cycle les commandes de la partie opérative sont disponibles en sortie de PC3.

4° cycle :

En T1 du quatrième cycle la partie opérative peut commencer à travailler.

Remarque : la remise à zéro des registres de sortie en T3 n'est nécessaire que parce que le modèle permet d'affecter directement les variables de contrôle. En effet si l'on n'efface pas les ordres d'affectation des variables de contrôle, ils resteront toujours valables et risqueront d'annuler les ordres d'affectation générés par les autres étages. Il faut non seulement effacer mais ne valider la mémorisation des résultats en sortie que si $start_i$ ou $validate_i$ est positionné à 1. En ce qui concerne le premier étage, la mémorisation n'est validée que si start1 vaut 1. C'est d'ailleurs pour cette raison qu'au départ start1 doit être remis à 1.

Nous n'avons décrit jusqu'ici que la transmission des ordres de haut en bas. Décrivons maintenant le passage de contrôle du troisième au deuxième étage après l'exécution d'une séquence de 2 instructions de niveau 3. Reprenons l'exemple précédent.

4°cycle suite :

En T1 du quatrième cycle, le nouvel état interne est transféré en entrée de PC3. Le contenu de C3 (en entrée de PC3) reste inchangé puisque l'on ne change pas de procédure et la deuxième instruction de la séquence est exécutée. En T4, les résultats sont disponibles en sortie de PC3 et start2 est mis à 1 puisque la séquence ne contient

que 2 instructions. Les tampons d'entrée de PC3 sont alors remis à zéro.

5°cycle :

La deuxième instruction de la séquence de niveau 3 est exécutée par la partie opérative et en même temps la deuxième instruction de la séquence de niveau 2 est activée.

2.2.2.2. Modèle accéléré

Pour accélérer le déroulement des instructions, on maintient plusieurs tranches actives au même cycle. Le but est de "pipeliner" l'exécution de façon à ce que la partie opérative soit utilisée au maximum. Si les différents automates de contrôle n'attendaient pas le résultat de l'exécution des actions opératives on pourrait facilement maintenir la partie opérative constamment occupée. Ce n'est hélas pas le cas, et plus l'automate de contrôle qui attend un compte rendu de la partie opérative pour déterminer son nouvel état, est "éloigné" de la partie opérative, plus le retard introduit par cette attente est long. S'il y a n niveaux de contrôle, si le test d'un compte rendu de la partie opérative est requis au niveau i, le retard introduit est n-i+1.

On constate que l'on peut facilement diminuer le nombre de NOPs en générant le signal de fin d'exécution d'une procédure non pas à la fin de l'exécution mais un cycle avant. Cette avance n'a de sens que pour le dernier étage de contrôle. Pour l'exemple simple traité ici, ce séquencement permet d'économiser 2 NOPs.

tranche \ cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	A						↑	B						
2		a_1	↑	a_2		↑	RAZ		b_1	↑	b_2			
3			op_1	op_2 RAZi	op_3	op_4	op_5 RAZ			op_6 RAZ		op_7	op_8	
POP				op_1	op_2	op_3	op_4	op_5			op_6		op_7	op_8

figure 1.7 : avance d'un cycle du signal de fin d'exécution

Si l'on veut réduire davantage le nombre de NOPs il faut pouvoir générer le signal de fin d'exécution d'une procédure plusieurs cycles en avance. Le plus simple semble être de construire une table de séquencement telle qu'un nouvel état n'est

généré à un niveau et à un cycle donné que si un retard d'un cycle entraîne l'apparition d'une instruction NOP au niveau de la partie opérative.

tranche \ cycle	1	2	3	4	5	6	7	8	9	10	11
1	A				↑	B					
2		a_1	↑	a_2 ↑			b_1 ↑	b_2			
3			op_1	op_2	op_3	op_4	op_5	op_6	op_7	op_8	
POP				op_1	op_2	op_3	op_4	op_5	op_6	op_7	op_8

figure 1.8 : avance d'un nombre quelconque de cycles du signal de fin d'exécution

Cet algorithme permet de minimiser le nombre de NOPs et de le réduire à zéro une fois le démarrage effectué si aucun compte-rendu de la partie opérative n'est nécessaire. L'inconvénient du choix d'une partie contrôle multiniveau est sa relative lenteur de fonctionnement. En effet si le premier étage de contrôle attend un compte rendu de la partie opérative, cette dernière reste inactive pendant n cycles, n étant le nombre de tranches de contrôle alors que cette période d'inactivité n'est que d'un cycle pour une partie contrôle à un seul niveau.

2.2.3. Topologie du circuit

Nous ne parlerons ici que de la topologie de la partie contrôle. Le modèle topologique de la partie opérative, du reste très classique puisqu'il s'agit de l'architecture en tranches (bit slice) sera présenté plus longuement dans le chapitre 3 section 4.
La partie contrôle est constituée par l'empilement de couches de contrôle. Chaque couche de contrôle est réalisée à l'aide d'un réseau programmable à 2 plans (PLA).
Nous présentons d'abord la topologie d'un PLA implémentant une tranche de contrôle puis la topologie globale du circuit.

Les réseaux logiques programmables sont des structures régulières qui permettent de réaliser des fonctions logiques combinatoires du type somme de produits. Les PLAs peuvent être réalisés à l'aide de 2 matrices de transistors en parallèle (structure NOR-NOR), [PERE 85]. La matrice ET calcule à partir des entrées du PLA les monômes dont l'équation générale s'écrit :

$$m = \overline{e1}\ \overline{e2}\ \overline{e3} \ldots$$

Les monômes de la matrice ET constituent les entrées de la matrice OU qui calcule les sorties du PLA dont l'équation générale s'écrit :

$$\overline{s} = \overline{m1 + m2 + \ldots}$$

On génère en fait la sortie complémentée à partir des entrées complémentées.
La topologie adoptée pour les PLAs est la topologie classique. Un PLA est constitué de 2 matrices ET et OU qui sont placées l'une à côté de l'autre avec au milieu les transistors de charge des monômes, en haut les amplificateurs d'entrée de la matrice et les transistors de charge des sorties, en bas les amplificateurs de sortie.
De plus des blocs de mémorisation sont systématiquement utilisés en entrée et en sortie à la fois pour mémoriser les informations nécessaires au séquencement et pour assurer la synchronisation entre les différentes tranches de contrôle.

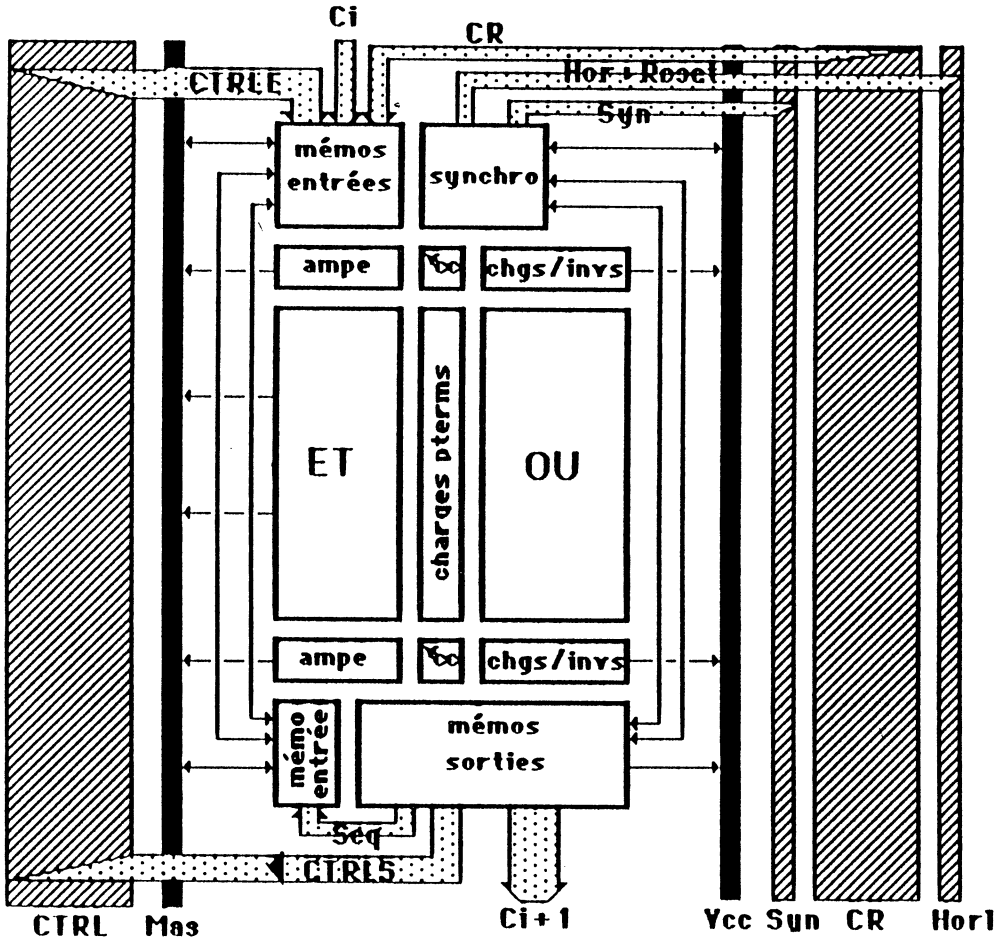


figure 1.9 : topologie d'un étage de contrôle

Pour ce qui est de la topologie globale, on précisera simplement que les fils de

contrôle (requêtes externes, variables de contrôle interne) sont situés à gauche ainsi que la masse alors que les fils de compte rendu de la partie opérative, de synchronisation (horloges, reset, activation des étages de contrôle) et l'alimentation VCC sont à droite [VARI 86].

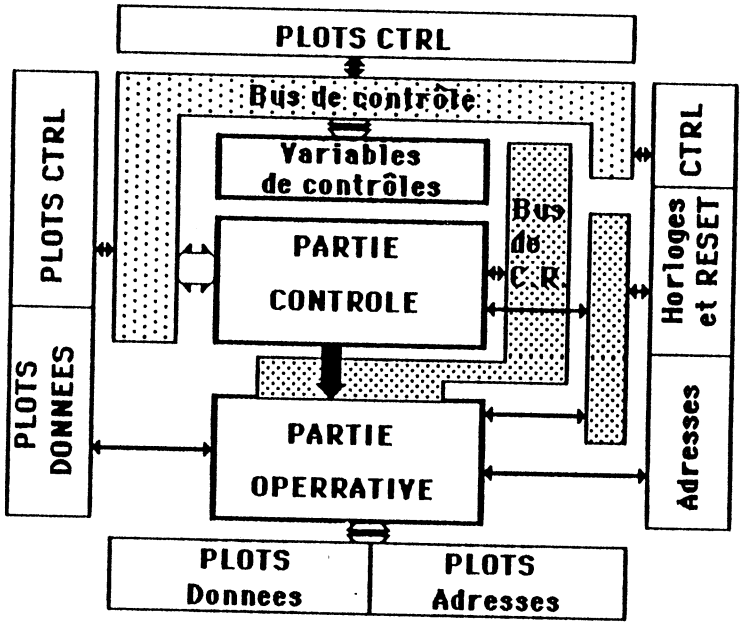


figure 1.10 : topologie du circuit

2.2.4. Circuiterie

Pour générer les masques, SYCO utilise un générateur de PLAs optimisés topologiquement PAOLA [CHUQ 84] et les cellules d'une bibliothèque de cellules prédéfinies. Actuellement, les circuits sont réalisés en technologie NMOS. Les différents blocs sont assemblés à l'aide d'un assembleur de silicium [SCHO 85].
PAOLA permet de placer les entrées et les sorties sur chacun des côtés libres des matrices ET pour les entrées et OU pour les sorties.
D'autre part PAOLA permet de déformer les PLAs générés. Cette déformabilité est un atout essentiel pour l'obtention de masques compacts pour le circuit. Cela permet en effet d'adapter dans une certaine mesure la largeur des différents PLAs de contrôle à la largeur de la partie opérative qui fixe la largeur du circuit.

2.3 Langage d'entrée du compilateur

2.3.1. Généralités

Le langage d'entrée du compilateur doit permettre de décrire l'algorithme de fonctionnement d'un circuit. Le concepteur d'un microprocesseur doit pouvoir spécifier l'algorithme d'interprétation des instructions. Pour cela, il doit disposer d'un langage algorithmique permettant d'une part de spécifier des transferts, des actions conditionnelles et de séquencement et d'autre part d'utiliser des fonctions et procédures, ce qui permet de hiérarchiser et de raccourcir la description de l'algorithme. SYCO utilise actuellement le langage de description de circuit LDS [LAUR 85]. La première version de SYCO était basée sur le langage IRENE [MARI 85]. Ces 2 langages procéduraux permettent de décrire le comportement et la structure de systèmes digitaux. La description qu'elle soit comportementale ou structurelle est de niveau transfert de registres [BARB 81], c'est à dire qu'elle fait intervenir des composants de base de type registres ou mémoires et non pas des portes ou des bascules pour descendre d'un niveau (niveau logique) ou des processeurs pour monter d'un niveau (niveau PMS processeur - mémoire - "switch"). D'autre part les 2 systèmes utilisés comprennent un compilateur et un simulateur. Le compilateur génère une structure intermédiaire sur laquelle travaille le simulateur. Ce simulateur est de type fonctionnel, c'est à dire qu'il permet de simuler le comportement du circuit décrit et donc de vérifier la cohérence de la description du concepteur.
Seule une partie des concepts de base de ces 2 langages a été utilisée.
La description structurelle consiste en la décomposition du circuit en modules interconnectés reliés entre eux par des signaux.
Cette description structurelle est hiérarchique : un module peut en appeler un autre.
Afin de distinguer la description comportementale de la description structurelle, il est nécessaire de définir 2 types de modules.

- les modules structurels (en LDS SMODULE)
- les modules comportementaux (en LDS CMODULE)

Le circuit peut donc être vu comme un ensemble de composants interconnectés. Chacun de ces composants peut être décomposé en composants de base. A chacun des modules structurels de base est associé un module comportemental qui permet de définir son comportement. L'évolution de la sortie en fonction des entrées doit être spécifiée.
En ce qui nous concerne, nous n'avons qu'un seul module structurel qui décrit le circuit tout entier puisque la description est de niveau comportemental. A ce module structurel est associé un module comportemental qui peut appeler d'autres modules comportementaux.
Cette organisation sous la forme d'une hiérarchie de modules comportementaux permet de hiérarchiser et de raccourcir la description. Mais c'est aussi la pierre angulaire sur laquelle est bâti le compilateur. Les appels des modules de même

niveau constituent les instructions d'un langage qui est interprété non pas directement par une machine physique mais par un interpréteur qui est écrit dans un langage de niveau inférieur, celui des instructions qui constituent ces modules (voir 2.1).

2.3.2. Description d'un circuit en LDS

La description comportementale d'un circuit est constituée d'une part d'un SMODULE, d'autre part d'un CMODULE qui lui est associé par l'instruction LINK.

2.3.2.1. Description d'un Smodule

Pour ce qui nous intéresse, un Smodule est caractérisé essentiellement par :
- ses bornes ou variables d'entrée-sortie
- les variables qu'il utilise.

Parmi les variables utilisées, on ditingue 2 types de variables :
- les variables de type données : ce sont les registres de la partie opérative. Ces registres sont de type point mémoire avec mémorisation sur un état et non sur un front.
- les variables de type contrôle : ce sont des variables qui sont réalisées par des bascules ou des registres selon que ces variables sont booléennes ou non. Ces bascules ou registres sont situés dans la partie contrôle et servent à mémoriser les requêtes de contrôle externes et les paramètres des procédures.

Remarques
1°) Cette distinction entre variables de contrôle et variables de données a été introduite pour les besoins du compilateur de silicium. En effet, il est peu intéressant d'intégrer les variables de contrôle booléennes ou qui portent sur un faible nombre de bits dans une partie opérative dont le nombre de bits est bien supérieur.
2°) la description n'est pas de nature purement comportementale puisque la description fait intervenir des registres et le parallélisme est spécifié par l'utilisateur.

2.3.2.2. Description d'un CMODULE

Un CMODULE est constitué d'une suite de "séquences". Une séquence est une suite d'instructions étiquetée par le nom de la séquence. Les instructions dans une séquence sont exécutées séquentiellement sauf si une action de branchement est exécutée.
L'appel d'un CMODULE (par l'instruction EXECUTE) provoque le déroulement

des séquences constituant ce CMODULE. A la fin de l'exécution du CMODULE appelé, l'exécution de la séquence appelante reprend.
Le nombre de séquences d'un CMODULE est égal au nombre de points de branchement différents augmenté de 1 ; en effet, le point d'entrée d'un CMODULE doit être étiqueté.
Une instruction (PINST en LDS) est constituée d'un ensemble d'actions se déroulant de façon concourante.
Les actions d'une PINST peuvent être conditionnelles (IF, CASE) ou inconditionnelles. Parmi les actions inconditionnelles, on distinguera les affectations: actions opératives (transferts, opérations) et actions de contrôle (SET, RESET) qui permettent d'affecter des valeurs aux variables de contrôle (d'entrée-sortie ou internes) et les actions de séquencement : le branchement à l'intérieur d'un CMODULE (NEXT) et le retour au niveau supérieur ou échappement d'un CMODULE appelé pour revenir au niveau de l'appel (EXIT).
Seules les actions opératives d'une part et les affectations des variables de contrôle d'autre part peuvent se dérouler en parallèle.
Les actions qui se déroulent en parallèle sont regroupées et sont délimitées par des parenthèses.

Exemple de description d'un Cmodule

```
CMODULE     MICROP;

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

<start>     pc :=0;

<newinstr>  r :=0;
            EXECUTE fetch;

<admode>    CASE (ir 4:2)
            WHEN ('00')  NEXT decode;
            WHEN ('01')  EXECUTE fetch; ad :=ir;
            WHEN ('10')  EXECUTE fetch; ad :=pc+ir;
            WHEN ('11')  EXECUTE fetch; ad :=pc-ir;
            END;

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

END;

2.4. Algorithme de traduction

La compilation de silicium n'est pas une tâche réputée facile, ni même tout simplement réalisable.

L'algorithme de traduction de SYCO est pourtant simple. En effet, SYCO est basé sur une architecture cible, ce qui facilite grandement la traduction. Le modèle adopté fournit à la fois un modèle architectural et un modèle topologique. Le processus de conception d'un circuit intégré peut être décrit par un diagramme en Y [GAJS 83] comportant 3 branches : les domaines comportemental, structurel et géométrique. Plusieurs niveaux d'abstraction sont représentés le long de chaque branche. Les niveaux d'abstraction augmentent au fur et à mesure que l'on s'éloigne du centre.

La donnée d'un modèle architectural et topologique facilite donc doublement la traduction. Les 2 phases de la compilation : traduction d'une description comportementale en une description structurelle et traduction d'une description structurelle en une description géométrique sont donc toutes les 2 simplifiées. L'algorithme de traduction peut être encore simplifié si le langage d'entrée du compilateur a été conçu en intégrant les problèmes de la traduction. Ainsi un typage données ou contrôle a été défini. La simplicité de l'algorithme est cependant essentiellement basée d'une part sur la correspondance entre la hiérarchie des appels de procédures et la hiérarchie des étages de contrôle et d'autre part sur la correspondance entre le nombre maximum d'opérations en parallèle et le nombre de sous parties opératives. Cette correspondance permet bien sûr au concepteur de contrôler le fonctionnement du compilateur puisque le mécanisme de traduction est évident. Toutefois, une phase d'optimisation peut être déclenchée afin d'aboutir automatiquement à un meilleur compromis surface vitesse. Un organigramme du système SYCO est présenté ci-dessous.

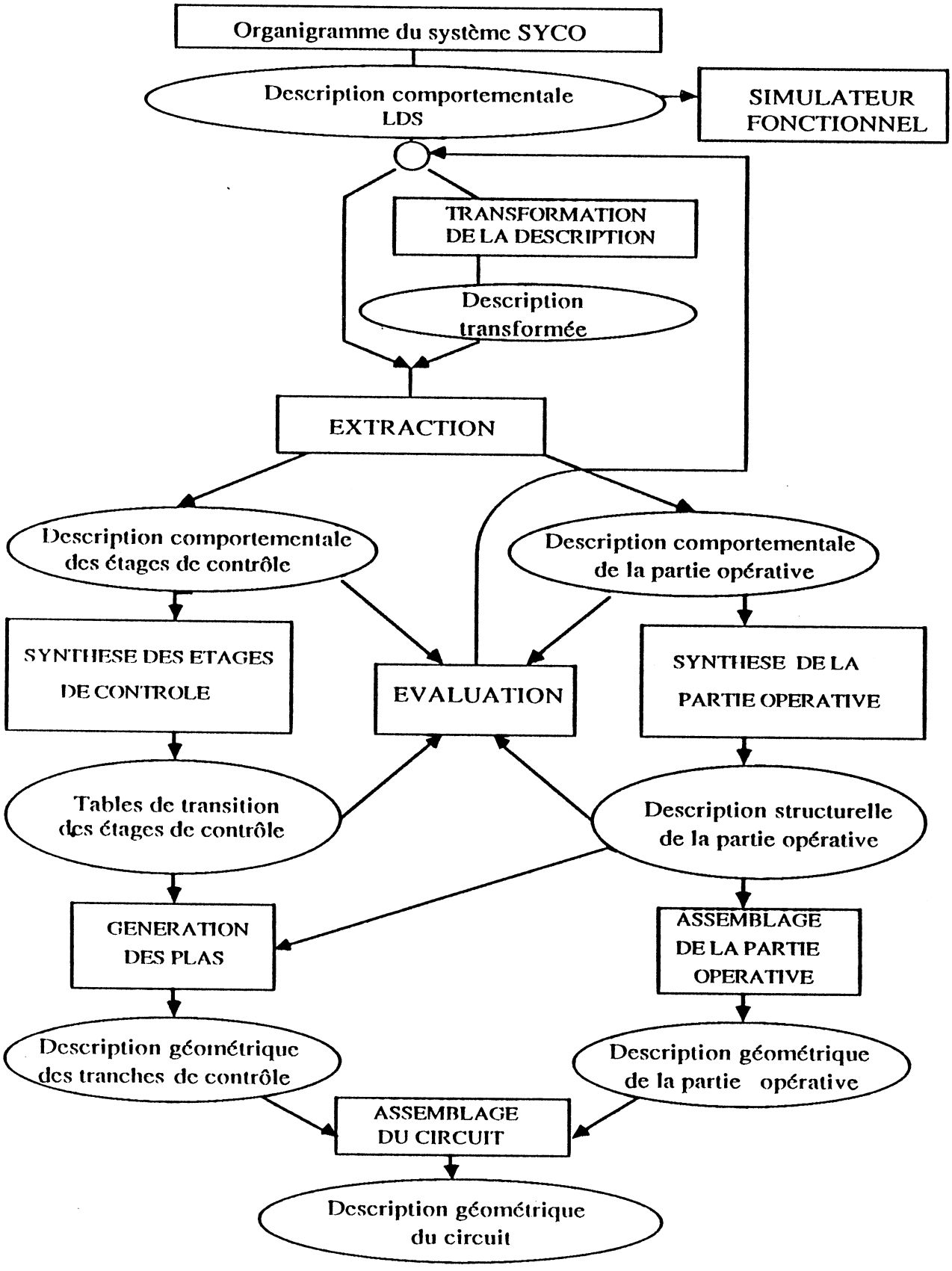


figure 1.11 : le système SYCO

Les différentes étapes de la compilation sont les suivantes:

2.4.1. Extraction des descriptions comportementales des tranches de contrôle et de la partie opérative [MART 85], [BOUR 86]

Cette étape de la compilation est actuellement la première étape. Elle sera précédée dans un proche avenir par une étape d'optimisation de la description originale.

Le but de l'extraction est de séparer les instructions des différents niveaux (opératif et niveaux de contrôle). La procédure la plus simple consiste à effectuer un traitement récursif. On extrait d'abord la description de la partie opérative, puis la description de l'étage de contrôle de plus bas niveau, puis celle de niveau immédiatement supérieur et on continue ainsi jusqu'à ce que l'on ait obtenu la description de toutes les étages de contrôle.

Avant de procéder à cette extraction, les instructions comportant des actions conditionnelles sont mises sous une forme canonique.

Toute instruction conditionnelle, c'est à dire comportant une ou plusieurs actions conditionnelles if ou case (qui sont alors en parallèle), ces actions conditionnelles pouvant elles mêmes comporter des actions conditionnelles imbriquées, est transformée en une suite de couples (condition instruction_inconditionnelle) où les différentes conditions sont exclusives et où les instructions sont constituées d'une suite d'actions inconditionnelles à exécuter en parallèle. Toute instruction conditionnelle est donc transformée en une instruction de type COND (de l'instruction lisp COND) où toutes les alternatives sont exclusives.

(COND

(cond 1 action 11 action 12... action $1p_1$)

(cond 2 action 21 ..... action $2p_2$)

(cond n action n1..... action np_n))

Cette décomposition peut être faite en 2 temps :

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

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

$n_1 * n_2 * \ldots * n_p$

avec : n_i le nombre d'alternatives du ième IF ou CASE selon le cas,

p le nombre de if ou de case en parallèle.

Remarque

Il est possible de minimiser le nombre de microinstructions en réduisant le nombre des alternatives ainsi générées. L'utilisation d'un réducteur logique permet d'une part de regrouper les instructions qui ont la même condition et d'autre part d'éliminer les instructions dont les conditions ne sont jamais respectées.

Ex : (IF e_1 a; ELSE b; END; IF (e_1 AND e_2) c; ELSE d; END;)

La condition d'exécution de l'instruction (IF (NOT e_1) AND (e_1 AND e_2) b; c; END;) n'est jamais vérifiée.

2.4.1.1. Génération de la description comportementale de la partie opérative

La description comportementale de la partie opérative est constituée d'un ensemble non ordonné d'instructions opératives. Une instruction opérative est constituée d'un ensemble d'actions opératives se déroulant en parallèle. Une action opérative est caractérisée d'une part par le fait qu'elle se trouve dans la partie exécution d'une action conditionnelle ou dans une action inconditionnelle et d'autre part par le fait qu'elle fait intervenir au moins un registre de la partie opérative.

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

Remarque

Si la génération des commandes est à 2 niveaux (génération d'un numéro d'instruction opérative puis génération des commandes : cf chapitre 3 section 5), il est souhaitable de ne garder qu'un exemplaire de chaque instruction opérative puis de procéder à une rénumérotation qui doit être répercutée dans la description traitée par l'extracteur des niveaux de contrôle.

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

Les instructions de contrôle de même niveau doivent être regroupées. La description comportementale peut être représentée par un arbre d'appels de CMODULEs. On suppose que les instructions opératives ont été remplacées par des CMODULEs opératifs.

Exemple :

```
CMODULE ada;
<e1>   (r:= 2; EXECUTE fetch;)
       a := a+b;
END;
```

est remplacé par

```
CMODULE ada;
<e1>   (r := 2; EXECUTE fetch;)
       EXECUTE instr1;
END;
```

avec

```
<instr1>   a := a+b;
```

On suppose aussi qu'il n'y a pas de cycle dans les appels de CMODULEs. Le cas suivant n'est pas compilable.

```
CMODULE a;
EXECUTE  b;
END;

CMODULE  b;
EXECUTE  a;
END;
```

Une instruction de contrôle est un ensemble d'actions de contrôle pouvant se dérouler en parallèle . De par ce fait, elle ne peut être constituée que d'un appel à un CMODULE, d'un ordre de branchement et d'affectations de variables de contrôle. En effet, 2 CMODULEs ne peuvent être exécutés simultanément. Le niveau d'une instruction de contrôle est donc donné par le niveau du CMODULE appelé s'il s'agit d'une instruction inconditionnelle et sinon par le niveau du CMODULE appelé de plus haut niveau parmi les CMODULEs appelés dans les différentes alternatives d'une instruction conditionnelle.

Si une instruction de contrôle ne fait pas appel à un CMODULE, 2 cas peuvent se présenter. Soit l'instruction comporte un ordre de branchement et le niveau de l'instruction est égal au niveau du CMODULE qui l'englobe, soit l'instruction ne comprend que des affectations de variables de contrôle, elle peut alors figurer dans n'importe quel niveau de contrôle puisque l'affectation d'une variable de contrôle peut se faire à n'importe quel niveau; on donne alors à cette instruction le niveau maximal qu'elle peut avoir, c'est à dire le niveau du CMODULE qui l'englobe afin de ne pas créer d'instructions intermédiaires inutiles.

L'algorithme d'extraction est le suivant. On remplace dans la description chaque instruction de contrôle de niveau directement supérieur (en fait si l'on compare les numéros des étages de contrôle, il faudrait dire inférieur d'une unité puisque la

partie contrôle de plus haut niveau a pour numéro 1) au dernier niveau extrait (c'est à dire une instruction appelant un CMODULE de niveau le dernier niveau extrait) par l'appel d'un CMODULE dont le nom est le point d'entrée dans la table des instructions de contrôle du niveau en cours d'extraction.

Les différentes instructions d'un CMODULE ne sont pas forcément de même niveau. On est alors obligé de créer des CMODULEs intermédiaires afin que toutes les instructions aient même niveau.

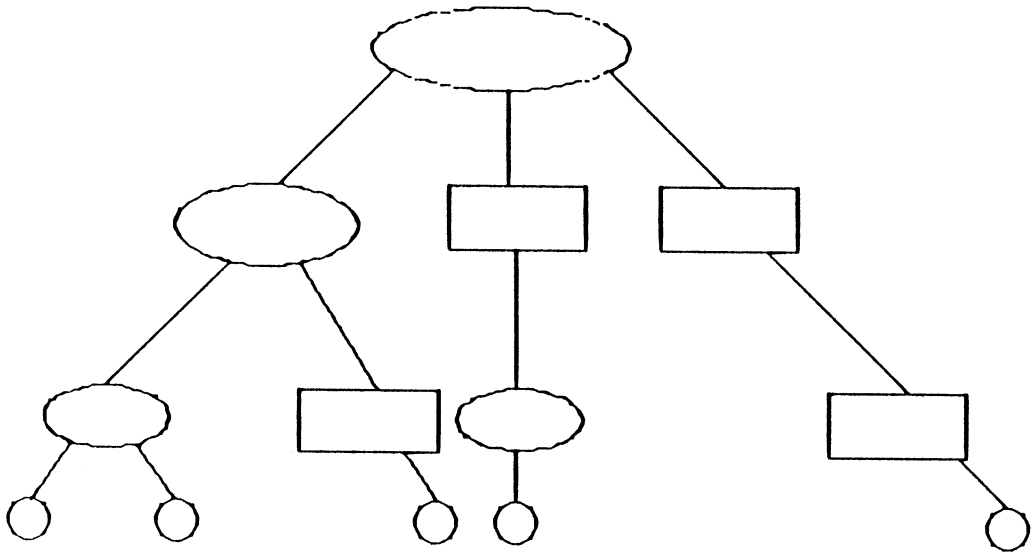


figure 1.12 : création de Cmodules intermédiaires

Exemple

Reprenons l'exemple précédent : on suppose que fetch n'est pas un CMODULE opératif.

```
CMODULE ada;
<e1>  (r := 2; EXECUTE fetch;)
        EXECUTE instr1. niv1;
END;

CMODULE instr1. niv1;
<instr1.niv1> EXECUTE  instr1;
END;
```

On introduit le CMODULE instr1.niv1 afin d'amener au même niveau les 2 instructions de contrôle.

Seule la tranche de contrôle qui est juste au dessus de la partie opérative peut actionner directement celle-ci. Cette restriction ralentit la machine. Il serait nécessaire de disposer de zones de transparence à l'intérieur des tranches de contrôle pour que n'importe quelle tranche de contrôle puisse venir activer directement la partie opérative.

2.4.2. Evaluation, optimisation et compromis surface-performance: [BEKK 86]

2.4.2.1. Evaluation

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

- la description comportementale des tranches de contrôle et de la partie opérative après la phase d'extraction; il s'agit alors d'une évaluation grossière qui permet de déterminer le nombre d'étages de contrôle et une estimation du nombre de sous parties opératives,
- la description structurelle des PLAs de contrôle et de la partie opérative après la phase de synthèse. Il s'agit d'une évaluation plus fine qui permet de déterminer le nombre et la taille des différents PLAs de contrôle ainsi que le nombre de sous parties opératives et la taille de la partie opérative.

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

- des optimisations,
- des transformations visant à la recherche d'un meilleur compromis surface performance.

On entend ici par optimisation toute transformation de la description du circuit qui a pour effet de réduire la surface du circuit sans diminuer les performances ou d'augmenter les performances mais sans provoquer une augmentation de la surface.

La recherche d'un meilleur compromis surface-performance consiste par contre à échanger une perte de performance contre un gain de surface ou vice versa.

2.4.2.2. Optimisation

Nous présentons dans cette section 2 types d'optimisation :

- l'optimisation de code qui est une technique bien connue des concepteurs de compilateurs [AHO 77].
- le compactage de microcode dont le but est de combiner, les microopérations en le plus petit nombre de microinstructions afin de réduire la taille et le temps d'exécution de l'algorithme d'interprétation des instructions.

Optimisation de code [AHO 77], [WALK 83]

Les transformations que l'on peut appliquer à la description originelle sont les suivantes :

- propagation des constantes
- élimination des expressions redondantes
- déplacement des actions invariantes à l'extérieur des boucles
- expansion de procédures appelées une seule fois.

Plus de détails sont donnés au chapitre II.

Compactage de microcode [DAVI 81], [AGER 76], [FISH 81], [ISOD 83], [TOKO 81].

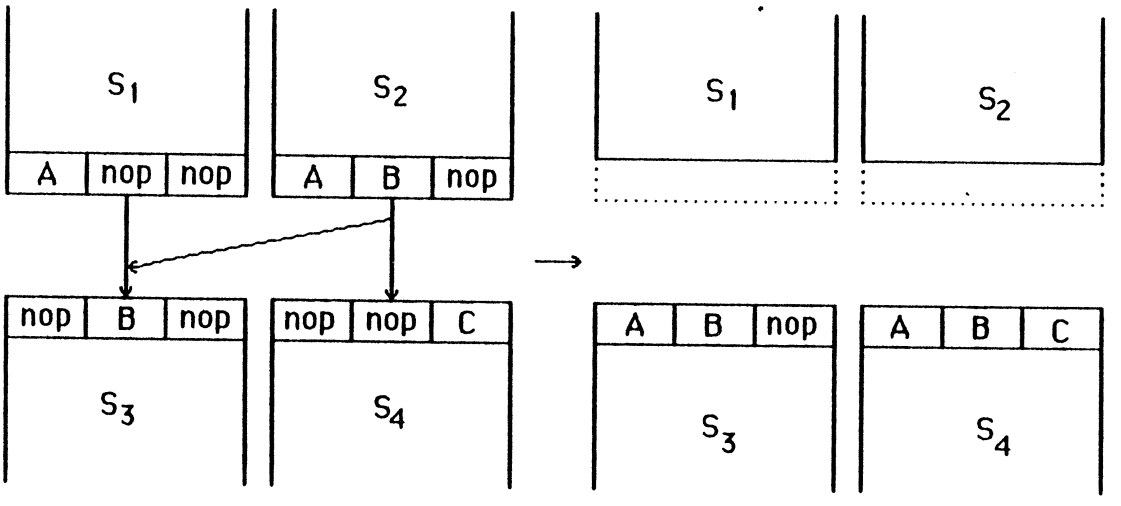
Les techniques de compactage de microcode peuvent être classées dans la rubrique optimisation ou dans la rubrique compromis surface-vitesse selon qu'elles sont appliquées après la construction de la partie opérative ou dès le début de la compilation.

Dans le premier cas, la mise en parallèle des microopérations dépend à la fois des relations de dépendance des données et des possibilités de parallélisme de la partie opérative alors que dans le second cas seules les relations de dépendance entre les données sont à prendre en compte. Le compactage de microcode peut être effectué à la fois avant et après la synthèse de la partie opérative. Dans le premier cas, il s'agit de paralléliser au maximum la description, dans le second il s'agit d'exploiter au maximum les ressources de la partie opérative.

La description comportementale est divisée en segments ne comportant qu'un seul point d'entrée et un seul point de sortie qui sont situés aux extrêmités des segments.

On distingue 2 types de compactage :

- le compactage local qui optimise la description à l'intérieur des blocs de base définis précédemment.
- le compactage global qui optimise la description dans son ensemble. Les segments successeurs et prédécesseurs de chaque segment doivent être déterminés afin de pouvoir effectuer une optimisation globale. Un des cas de figures qui peuvent se présenter est le suivant [TOKO 81]:



Compactage global

figure 1.13 : compactage global

Compactage local

Si les microinstructions d'un même bloc de base dans la description de départ contiennent déjà des microopérations en parallèle, il est nécessaire de décomposer ces microinstructions en microopérations afin d'obtenir le minimum absolu du nombre des microinstructions.

Davidson et al comparent 4 méthodes de compactage local dans [DAVI 81].

- la méthode du premier venu, premier servi : les microopérations sont traitées dans l'ordre de départ. Chaque microopération est placée dans la première des microinstructions (en cours de formation) possible.
- la méthode du chemin critique : c'est la méthode inverse de la précédente. Il s'agit de déterminer le nombre maximum de microinstructions que l'on peut exécuter avant une microopération sans provoquer d'augmentation du nombre de microinstructions.
- la méthode "Branch and Bound" : on construit un arbre dont les nœuds sont les microinstructions. On part des microopérations qui n'ont pas de prédécesseur dans le graphe de précédence. Des branches sont créées chaque fois qu'il y a plus d'une microinstruction qui peut être constituée à ce niveau. Toutes les branches de l'arbre qui peuvent conduire à une solution optimale sont parcourues. Seul cet algorithme dont le temps d'exécution est d'ailleurs une fonction exponentielle du nombre de microopérations garantit l'obtention de la solution optimale.
- la méthode "List scheduling" : seule la meilleure branche de l'arbre de la méthode Branch and Bound est formée à chaque niveau. La meilleure peut être par exemple celle dont les microopérations qui constituent la

microinstruction ont le plus de descendants (directs ou indirects) dans le graphe de dépendance.

Compactage global

En particulier si l'on compacte non pas la description d'entrée de SYCO mais les descriptions des tranches de contrôle après extraction, une forme de compactage global consiste à éliminer les instructions qui ne comportent qu'un ordre de branchement NEXT. Les instructions de contrôle peuvent en effet être constituées d'un appel de CMODULE : EXECUTE, d'affectations de variables de contrôle : SET et RESET et d'un ordre de branchement : NEXT ou EXIT.

Une instruction ne comportant qu'un NEXT peut être remplacée par la première instruction de la séquence commençant par l'étiquette de branchement du NEXT.

Il est d'autant plus important qu'il n'y ait pas d'instructions NOP (no opération) que le nombre d'étages de contrôle est plus élevé. Il est donc souhaitable dans la mesure du possible que chaque instruction de contrôle comporte une partie EXECUTE.

2.4.2.3. Compromis surface-vitesse [BEKK 86]

Parmi les transformations que l'on peut appliquer à un algorithme, un certain nombre peuvent conduire à un compromis : augmentation des performances mais aussi augmentation de la surface occupée par le circuit ou vice-versa.

- l'expansion de procédures ou la formation de procédures.
- transformation d'une instruction comportant des IF en parallèle en une suite de IF, factorisation d'un CASE .

Expansion/formation de procédures

L'expansion de procédures consiste à remplacer les appels des procédures par des copies des procédures. Cette transformation conduit à une augmentation de la surface nécessaire à la mémorisation des commandes de la partie opérative mais aussi à une augmentation des performances car il n'y a pas de perte de temps pour l'appel des procédures (empilement et dépilement de l'adresse de retour). En ce qui concerne SYCO, le temps gagné se traduit en fait par une diminution du nombre des étages de contrôle, ce qui est équivalent à la diminution du temps de traversée de la structure de contrôle quelle que soit d'ailleurs le mode de fonctionnement (une seule tranche active ou pipeline).

La formation de procédures consiste à remplacer des portions de code identiques par des appels à des procédures. Les effets de cette transformation sont inverses de ceux d'une expansion.

Dans le cas de SYCO [BEKK 86] , le compromis surface-performance se pose en les termes suivants : on peut ne remplacer que les appels de certaines procédures par des copies afin de contrôler le nombre d'étages dans la partie contrôle. Pour

diminuer d'un le nombre d'étages, seuls les appels des Cmodules de l'étage que l'on veut faire disparaître seront remplacés par des copies des Cmodules.

Diminution du parallélisme

1) Transformation d'une instruction comportant des IF en parallèle en une suite de IF

Cette transformation permet de ne mémoriser que 2n microinstructions au lieu de 2^n si n est le nombre de IF en parallèle.

La factorisation des alternatives d'un case permet de réaliser cette transformation.

```
CASE e1:3

    WHEN  ('000')   (b; d; f;)
          ('001')   (a; d;  f;)
          ('010')   (b; c; f;)        IF e1 a; else b; end;
          ('011')   (a; c; f;)        IF e2 c; else d; end;
          ('100')   (b; d; e;)        IF e3 e; else f ; end;
          ('101')   (a; d; e;)
          ('110')   (b; c; e;)
          ('111')   (a; c; e;)
END;
```

2) Décomposition des instructions opératives

En ce qui concerne la partie opérative, les étapes de compactage ont pour effet de paralléliser au maximum la description comportementale, donc d'exiger le maximum de ressources opératives. Si l'on ne veut pas dépasser une limite donnée pour la partie opérative, il peut être nécessaire de décomposer certaines instructions opératives.

Il existe aussi un autre type de compromis dont la recherche est intéressante. C'est le compromis largeur/hauteur. La déformabilité des blocs est une des conditions nécessaires à la minimisation de la surface des zones de connexion entre les blocs.

Plusieurs solutions fonctionnelles sont présentées dans [BEKK 86].

2.4.3. Synthèse des étages de contrôle [MHAY 86]

2.4.3.1. Généralités

Le but de la synthèse est de générer les spécifications des matrices ET et OU des différents PLAs de contrôle puisque l'architecture cible est constituée d'un empilement de niveaux de contrôle réalisés par des PLAs.

Les colonnes de la matrice ET sont constituées des entrées de l'étage de contrôle. Ce sont :

- les comptes rendus de la partie opérative,
- les champs du registre instruction,
- les requêtes de contrôle externes et internes,
- l'état constitué de 2 champs :
  - le séquencement "externe" : l'appel d'un CMODULE par le niveau supérieur,
  - le séquencement interne nécessaire au déroulement du CMODULE appelé.

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

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

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

L'étape d'extraction fournit pour chaque niveau de contrôle une description comportementale qui est constituée d'une suite de séquences d'instructions de contrôle de même niveau.

2.4.3.2. Les différentes étapes de la synthèse

La synthèse commence par l'étage de contrôle immédiatement supérieur à la partie opérative.

Avant de pouvoir générer les spécifications du PLA de génération des commandes de la partie opérative que ce PLA assure le séquencement des cycles opératifs ou non, le compilateur doit attendre le résultat de la compilation de la partie opérative. L'architecture de la partie opérative ainsi que les ressources utilisées pour l'exécution de chaque instruction opérative doivent être connues.

Avant de générer le contenu des matrices du PLA, un certain nombre de transformations et d'optimisations peuvent être effectuées. Nous ne parlerons ici que de la minimisation du nombre d'états de l'automate de contrôle. L'idée est de partitionner l'ensemble des états en classes d'équivalence disjointes. Deux états

appartiennent à une même classe d'équivalence si l'application d'une même séquence en entrée produit toujours la même séquence en sortie que l'automate parte de l'un ou l'autre de ces 2 états. Le nombre minimum d'états d'un automate équivalent est égal au nombre de classes de cette partition. Pour plus de détails se rapporter à [BEKK 86].

Les principales étapes de la synthèse de parties contrôle sont les suivantes [MHAY 86].

Mise sous forme canonique des expressions conditionnelles

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

2°) Les opérateurs NOT sont éliminés, puis les opérateurs <>. Par exemple, l'expression NOT (A=1) est d'abord transformée en (A<>1), puis si A est un nombre à 2 bits A_0 et A_1 en (A_0 = 0) OR (A_1 = 1). Cette étape, comme on le voit peut générer de nouveaux OR.

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

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

Recensement des entrées et sorties du PLA

4°) Cette étape consiste à recenser d'une part le nombre et la nature des entrées (comptes rendus de la partie opérative, champs du registre instruction, requêtes de contrôle internes et externes) et des sorties (ordres d'affectation des variables de contrôle internes et externes, signaux de synchronisation des tranches de contrôle, appels de CMODULEs de niveau inférieur ou commandes de la partie opérative) que l'on ne peut coder librement et d'autre part les entrées et les sorties que l'on peut coder librement.

Seuls les états de l'automate peuvent en fait être codés librement avec la restriction qu'un état doit être constitué de 2 champs :

- 1 champ nécessaire au séquencement interne,
- 1 champ codant le CMODULE en cours d'exécution.

Seul le nouvel état interne doit être généré puisque c'est l'étage supérieur qui détermine le nouveau CMODULE à exécuter.

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

Codage des états

5°) G. de Michelli propose 2 algorithmes de codage des états d'un automate [DEMI 84], [DEMI 86].

Les monômes qui ont les mêmes sorties et les mêmes entrées, les entrées correspondant aux états de l'automate non comprises bien sûr, sont regroupés.

Deux cas peuvent se présenter : le nombre de bits utilisés pour le codage n'est pas limité. On cherche alors un codage qui permette de réduire le nombre des monômes au nombre de lignes de la table de transition après le regroupement effectué précédemment. Le problème revient alors à trouver un codage tel que les sous espaces vectoriels de dimension minimum contenant les codes des mnémoniques des états d'entrée appartenant à une même ligne de la table de transition après regroupement, soient disjoints. Par exemple : si les états e_1 et e_2 ont respectivement les codes 0111 et 1110, le sous espace vectoriel de dimension minimum qui contient ces 2 codes est *11* ; il est de dimension 2. Le symbole * indique que la variable peut prendre indifféremment la valeur 1 ou 0.

Si le nombre de bits de codage est limité, on ne peut en général pas trouver de codage qui permette de réduire le nombre des monômes au nombre de lignes après regroupement.

Dans le cas de SYCO, il faut tenir compte du fait que l'état est constitué de 2 champs :

- le séquencement interne,
- et le nom du Cmodule en cours d'exécution. Ce champ est bien sûr le même pour tous les états du Cmodule.

En conséquence en général on ne peut coder librement que le champ de séquencement. D'autre part le premier état d'un Cmodule devra toujours avoir son champ de séquencement interne codé par la valeur 0 puisque le tampon de séquencement interne est remis à 0 au début de chaque Cmodule.

Seules les étapes 1, 2, 3 et 4 sont réalisées actuellement.

2.4.4. Génération des PLAs [CHUQ 84] [PERE 85]

La génération des PLAs est effectuée à l'aide du générateur de PLAs optimisés PAOLA. Le système PAOLA est basé sur la méthode des lignes brisées qui consiste à placer autant de segments (entrées ou sorties) que possible dans une même colonne alors que classiquement chaque entrée ou sortie occupe une colonne de la matrice ET (respectivement de la matrice OU).

2.4.5. Assemblage global [VARI 86]

L'assemblage global du circuit est réalisé à l'aide de l'assembleur de silicium Lubrick. Deux fonctions de routage (par dévoiement ou par croisement) sont

utilisées. Les problèmes topologiques de l'assemblage de la partie opérative, bien que moins complexes, sont de même nature que ceux de l'assemblage global du circuit. Ils seront abordés dans le chapitre 2.

Chapitre 2

Compilation de parties opératives

1. Origine de la notion de partie opérative

Il est bien connu [GLUS 65] que l'on peut décomposer une machine séquentielle en une partie contrôle et une partie opérative. Le comportement de la partie contrôle peut être modélisé à l'aide d'un automate d'état fini B de type Mealy alors que le comportement de la partie opérative peut être modélisé à l'aide d'un automate de Moore A dont le nombre d'états n'est pas forcément fini. Cette décomposition a été introduite pour des raisons pratiques. Les machines séquentielles réalisées étaient de plus en plus complexes et le nombre des états qui était lié aux informations mémorisées devenait de plus en plus grand. La décomposition d'une machine séquentielle en 2 automates qui coopèrent permet de réduire le nombre des états de l'automate de contrôle, ce qui permet d'appliquer les techniques de synthèse classiques. Les états de l'automate opératif sont constitués par l'ensemble des valeurs pris par les registres et autres éléments de mémorisation de la partie opérative.

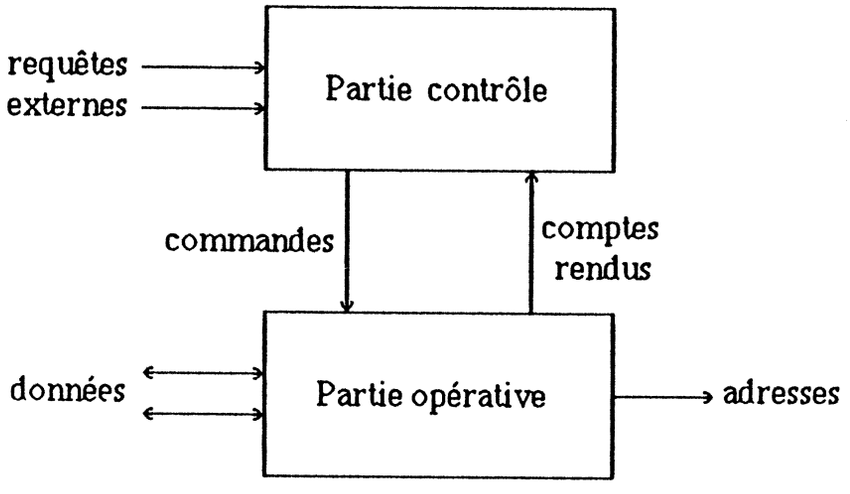


figure 2.1 : décomposition d'un microprocesseur

La partie opérative [SIFA 82] est constituée d'un ensemble de variables et d'opérateurs interconnectés. Les variables représentent l'état de la partie opérative qui peut être modifié par l'activation d'opérateurs. Les ordres d'activation sont générées par la partie contrôle. La partie opérative retourne des comptes rendus d'exécution qui sont essentiellement le contenu du registre instruction dans la mesure où il est placé dans la partie opérative et l'état de la partie opérative. En outre elle communique avec une zone de mémoire où sont stockés le programme et les résultats de l'exécution des instructions du programme.

La partie contrôle gère le séquencement des opérations dans la partie opérative.

Elle peut être représentée par un automate dont les sorties (associées aux états ou aux transitions) correspondent à des ordres d'activation des opérateurs. Les entrées de la partie contrôle sont les comptes rendus de la partie opérative. Il faut ajouter cependant en entrée un certain nombre de commandes externes par rapport au schéma de Glushkov. Ces commandes sont de trois types :

- l'horloge qui donne une référence temporelle,
- les signaux venus des unités périphériques,
- les signaux de commande venant d'un niveau supérieur : commande de mise en route, d'arrêt ou d'attente.

2. Styles de conception de parties opératives

Le choix d'un style de conception pour la partie opérative dépend :

* de la nature de l'algorithme :
  - exécution en pipeline
  - exécution d'opérations en parallèle

* des performances désirées : densité d'intégration et coût, rapidité, puissance dissipée

* et des technologies disponibles (composants TTLs pour les circuits LSI et MSI, MOS : prédiffusé, précaractérisé ou circuit à la demande).

En dehors de la technologie et de la méthodologie employées, une partie opérative peut être caractérisée par :

* sa microarchitecture : le nombre et la nature de ses composants de base
  - registres (bascules ou points mémoire)
  - opérateurs (opérateur spécifique ou universel (UAL))
  - interconnexions (multiplexeurs ou bus)

* sa topologie : structure bit slice ou structure irrégulière nécessitant l'utilisation d'outils de placement et de routage

* le séquencement des transferts et des opérations en phases de l'horloge de base

2.1. Classification basée sur l'architecture

Thomas définit plusieurs styles de conception de parties opératives dans sa thèse [THOM 77]. Sa classification, qui est basée sur des critères divers, n'est pas systématique. Nous retiendrons 2 styles de base de conception des parties opératives qui sont basés sur leur architecture :

- style distribué :

Ce style de conception est caractérisé par l'utilisation de nombreux éléments (non structurés) de mémorisation éparpillés et d'opérateurs interconnectés de façon irrégulière et d'autre part par un faible taux d'optimisation d'utilisation des ressources. Les composants de base d'une partie opérative sont les registres, les opérations monofonctions, les connexions directes (simples fils de connexion) et les multiplexeurs.

Il n'y a pas de bus. Les opérateurs et les chemins de données sont souvent dupliqués au lieu d'être combinés et regroupés autour d'une structure à bus. Les unités de traitement sont situées au voisinage des registres contenant les opérandes des opérations qui y sont effectuées.

Ce style permet d'obtenir des circuits rapides puisqu'aucune limite matérielle n'est imposée. Cependant les circuits générés de cette façon ne sont pas très denses à cause de l'utilisation de nombreux opérateurs et de multiplexeurs.

Les composants de base typiques de ce style sont :

- soit les cellules de bibliothèques précaractérisées LSI ou VLSI, la partie opérative est alors intégrée sur un seul circuit.
- soit les composants TTLs, SSI et MSI, la partie opérative est constituée d'une carte de composants SSI et MSI.

- style centralisé :

Ce style de conception est caractérisé par l'utilisation d'un faible nombre d'unités de traitement multifonctions (UALs par exemple) et d'éléments d'interconnexion. Les chemins de données non parcourus simultanément sont regroupés pour former des bus. Les registres et les opérateurs sont donc reliés entre eux par des bus. Mais le nombre de ces bus est en général limité. [TSEN 81] distingue plusieurs modèles de ce type selon le nombre de bus.

2.2. Classification basée sur la topologie

On peut établir une autre classification qui est basée non pas sur l'architecture mais sur la topologie des parties opératives.

Dans les premiers microprocesseurs, les parties opératives n'avaient pas une forme très régulière. On pouvait distinguer des blocs de type ROM ou RAM et une ALU. La tendance actuelle est de concevoir des parties opératives bit slice (en tranches). La partie opérative est constituée d'un bloc rectangulaire résultant de l'empilement de tranches. Chaque tranche correspond à un bit de la partie opérative. Pour que cette méthode donne de bons résultats, il faut que toutes les cellules d'une tranche aient la même hauteur. Cette contrainte permet d'obtenir des masques denses puisqu'il n'y a pas de trous dans les tranches.

3. Modèles de parties opératives pour la compilation

La plupart des publications traitant de la génération automatique de parties opératives décrivent soit la synthèse de l'architecture à partir d'une spécification comportementale, soit la génération des spécifications des masques à partir de la décomposition du circuit en blocs interconnectés. S'il est souhaitable de découper le processus de génération des spécifications de masques en 2 étapes indépendantes, il est aussi souhaitable de prendre en compte dans la phase de synthèse de l'architecture les problèmes topologiques de la conception de circuits intégrés. La plupart des travaux effectués récemment abordent le problème de la synthèse en toute généralité sans se préoccuper des problèmes topologiques. Cette approche même si elle peut permettre d'optimiser le nombre des composants de la partie opérative, ne donne pas forcément de bons résultats en termes de surface de silicium. Une partie importante du circuit peut être occupée par les connexions entre ces différents composants. Une solution à ce problème consiste en l'adoption d'un modèle à la fois architectural et topologique.

Nous nous proposons dans ce paragraphe d'établir une classification des modèles de parties opératives générées automatiquement. Ces modèles peuvent être caractérisés par le mode d'interconnexion des différents composants de la partie opérative ou par l'organisation topologique de celle-ci. [CAMP 85] classe les générateurs de parties opératives non pas non pas en fonction du modèle architectural de la partie opérative mais en fonction des techniques de synthèse utilisées :

- compilation directe sans optimisation,
- formalisation algébrique [HAFE 81], [HAFE 82]
- systèmes experts [KOWA 85],
- analyse du flot de données [ORAI 86],
- grammaires de graphes [GIRC 84].

3.1. Classification basée sur le mode d'interconnexion des composants de la partie opérative

On peut répartir en 2 classes les systèmes de synthèse existant actuellement selon qu'ils génèrent des parties opératives à multiplexeurs : [A. PARK 79], EMUCS [HITC 83], le système de Girczyc [GIRC 84], CADDY [CAMP 84], [W. ROSE 85], MAHA [A. PARK 86], MACPITTS [SOUT 83] ou des parties opératives à bus : DAA [KOWA 83], [KOWA 85a], [KOWA 85b], EMERALD [TSEN 84], DE [TAKA 84], APOLLON [JAMI 85], CATHEDRAL2 [DEMA 86].
L'utilisation de bus permet de diminuer le nombre de multiplexeurs mais pas de les

supprimer complètement. En effet, un registre ne possède qu'une entrée et qu'une sortie (qui peuvent d'ailleurs être confondues dans le cas des bus complémentés à précharge) et on est obligé de placer un multiplexeur devant les entrées des registres qui peuvent être accédés par plusieurs registres (parties opératives à multiplexeurs) ou par plusieurs bus (parties opératives à bus).

3.2. Classification basée sur l'organisation topologique de la partie opérative

On peut aussi distinguer 2 classes selon que les systèmes de synthèse génèrent des parties opératives dont le plan de masse est prédéfini :

Apollon [JAMI 85],
Cathedral2 [DEMA 86],
Macpitts [SOUT 83],

ou des parties opératives dont la structure topologique n'est pas régulière et qui nécessitent des outils de placement et de routage.La plupart des systèmes de synthèse sont de ce type.

Seuls Apollon, Cathedral2 et Macpitts ont un modèle topologique sous-jacent basé sur une structure bit slice. Il existe d'autres systèmes possédant un modèle topologique similaire (Bristle Blocks [JOHA 79], Datapath Generator [SHRO 82], Data path [MARS 86]) mais ces systèmes ont pour entrée une description structurelle. Ce sont en fait des assembleurs de silicium pour des parties opératives de type bit slice. Parmi tous ces sytèmes, seul Macpitts adopte une approche basée sur l'utilisation de multiplexeurs pour interconnecter les différents éléments de la partie opérative. Tous les autres utilisent des bus parallèles dont le nombre est limité par le pas d'une tranche de la partie opérative (2 pour Apollon, Bristle Blocks et Datapath Generator; 3 pour Data Path avec la possibilité d'augmenter le nombre de bus : les cellules de la bibliothèque sont traversées par 10 pistes de transparence, les bus d'alimentation non comptés; un nombre variable mais limité pour Cathedral2).

Cette approche permet de prendre en compte les problèmes topologiques dès la phase de génération de l'architecture. Ce n'est pas le cas de la plupart des autres systèmes. Toutefois quelques systèmes commencent à intégrer les problèmes topologiques dès la phase de l'allocation des ressources. BUD [MCFA 86] n'a pas de modèle topologique fixe mais il utilise des informations géométriques et électriques pour grouper les composants de la partie opérative .

L'avantage des systèmes à plan de masse prédéfini réside dans la possibilité de contrôler la surface de la partie opérative générée. La structure bit slice donne de bons résultats d'un point de vue surface.

Parmi les systèmes basés sur un modèle topologique, nous comparerons Macpitts et Apollon qui sont basées sur 2 modes d'interconnexion différents : le premier utilise des multiplexeurs alors que l'autre utilise des bus . Cette différence se traduit de la

manière suivante :

1) un canal de routage est réservé aux connexions entre les composants de la partie opérative au niveau de chaque tranche (Macpitts), alors que les bus sont situés au-dessus de la logique dans le cas d'Apollon. Dans ce cas aucune place n'est perdue à cause des connexions.
Par contre l'utilisation de canaux de routage permet de tracer les connexions de n'importe quel schéma d'interconnexion alors qu'Apollon est limité par le nombre de pistes de transparence de ses cellules de base.

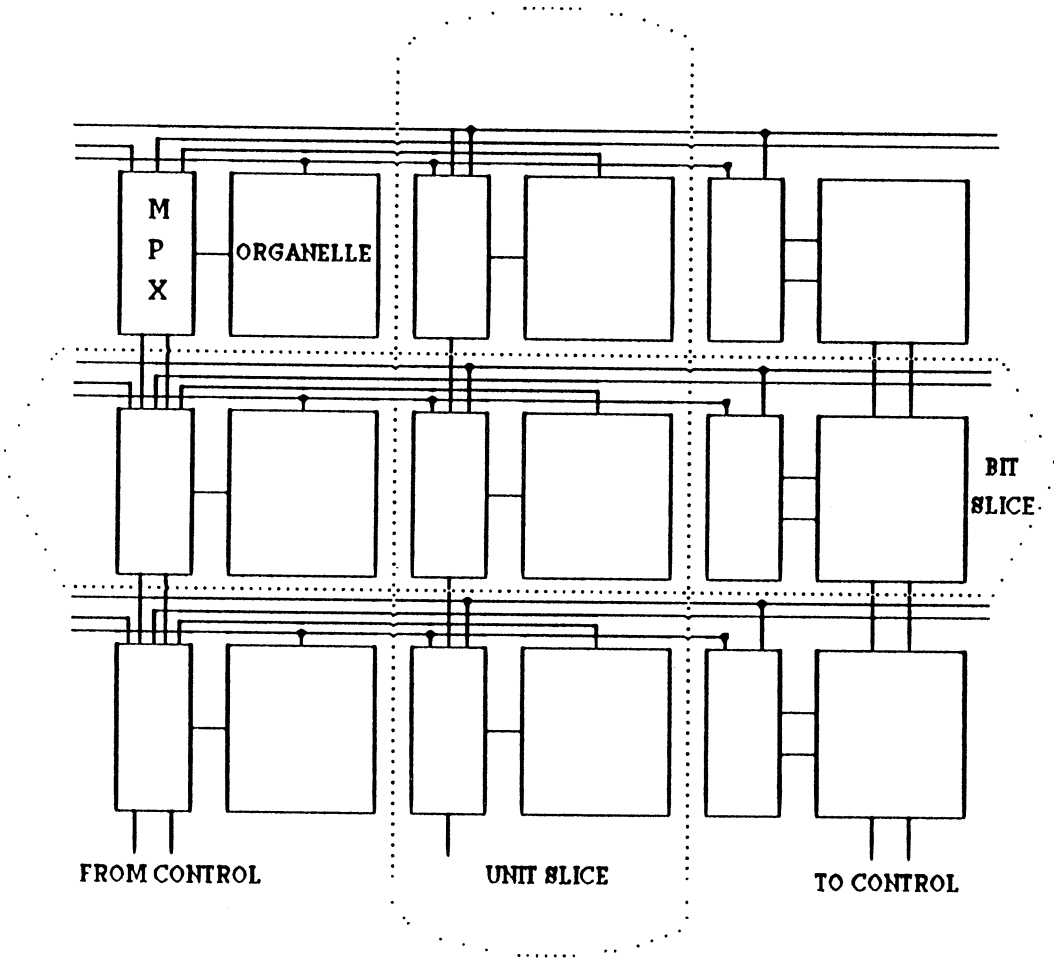
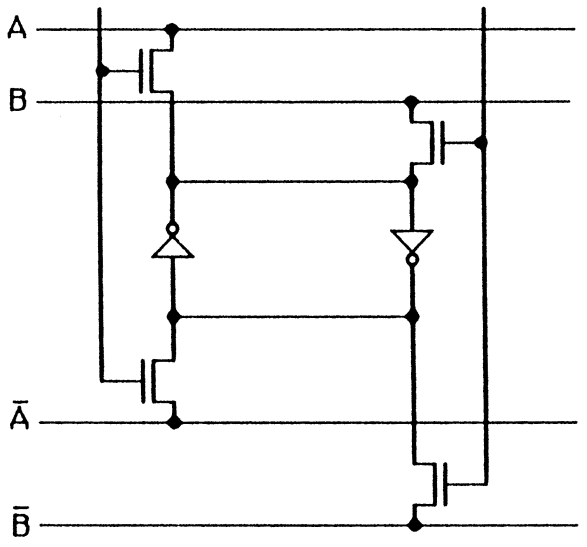


figure 2.2 : modèle de la partie opérative de Macpitts

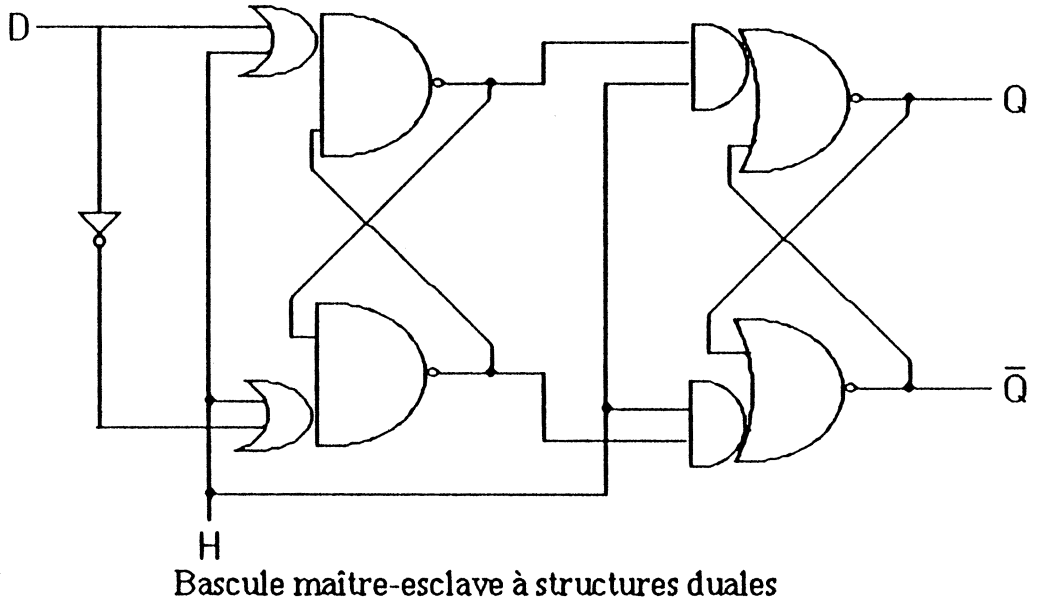
2) Macpitts utilise des bascules de type maître-esclave pour réaliser les registres alors qu'Apollon utilise de simples points mémoire beaucoup moins coûteux en surface; l'utilisation de points mémoire peu coûteux en surface n'est possible que parce que les transferts sont effectués à l'aide de bus complémentés préchargés et amplifiés.



Point mémoire double accès

figure 2.3

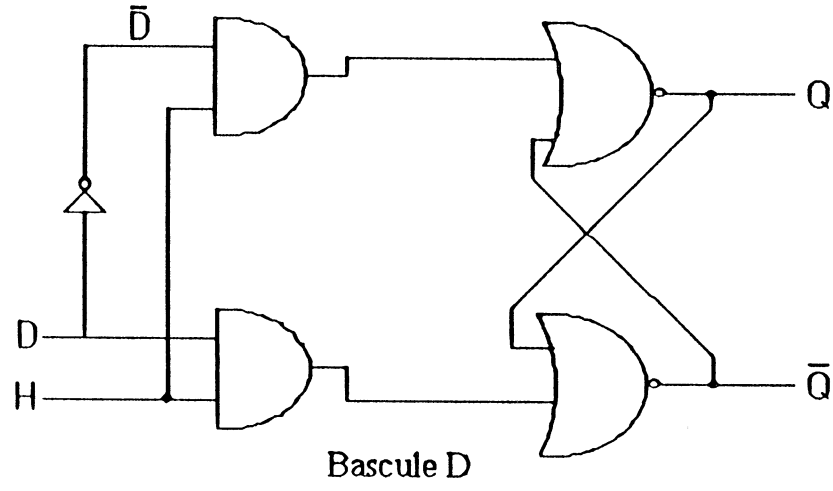
Un point mémoire double accès nécessite 8 transistors alors qu'un maître esclave constitué de 2 cellules RS avec horloges duales comprend 17 transistors en technologie NMOS enrichie/déplétée.



Bascule maître-esclave à structures duales

figure 2.4

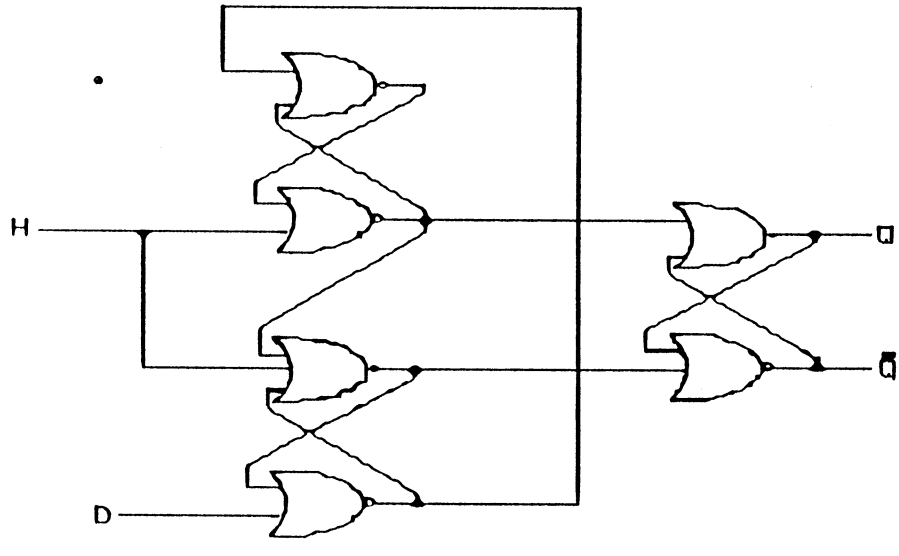
Une simple bascule D est plus économique qu'un maître esclave. Elle ne comprend que 9 transistors.



Bascule D

figure 2.5

Toutefois l'utilisation d'une simple bascule ne permet pas d'effectuer des transferts où un registre est à la fois source et destination. Pour cela on peut utiliser une bascule D synchronisée par front



Bascule D synchronisée sur front

figure 2.6

Une telle bascule nécessite 6 parties NOR soit 18 transistors ce qui est moins économique qu'un maître-esclave.

L'utilisation d'un point mémoire connecté à 2 paires de bus complémentés permet de diminuer le nombre de transistors des éléments de mémorisation :

- en contrepartie le dimensionnement des transistors des éléments de

mémorisation et des préchargeurs et amplificateurs des bus est beaucoup plus ardu. Les connexions des points mémoire aux bus sont bidirectionnels, il est donc important de s'assurer que lorsqu'on connecte un point mémoire en écriture, ce n'est pas le registre qui impose sa valeur au bus mais bien le bus qui impose sa valeur au registre.
- d'autre part le temps nécessaire à l'exécution d'un transfert se trouve nettement augmenté du fait de l'existence de plusieurs phases :

précharge
lecture
amplification
écriture

En conclusion de cette comparaison, nous retiendrons que :
- l'adoption d'un modèle topologique de type bit slice semble être la clé d'un circuit dense,
- 2 approches sont possibles en ce qui concerne les interconnexions :

.l'utilisation de multiplexeurs : une grande place est perdue à cause des interconnexions puisque chaque transfert a son propre chemin de données. Les connexions doivent être tracées dans un canal de routage au dessus de chaque tranche de la structure "bit slice". En contrepartie, d'une part cette approche est d'une très grande souplesse : elle permet d'interconnecter n'importe quels composants de la partie opérative (il n'y a donc pas de limitation de performances ni obligation d'avoir des cellules extensibles pour adapter le nombre des bus aux besoins de l'algorithme), d'autre part le temps d'exécution des transferts peut être diminué par rapport aux parties opératives à bus, chaque transfert ayant son propre chemin de données.

.l'utilisation de bus parallèles : cette approche est la plus efficace du point de vue surface. Le circuit est très dense puisqu'aucune place n'est perdue à cause des interconnexions. D'une part les bus sont au dessus de la logique, c'est à dire qu'ils sont réalisés en aluminium; la couche métallique est déposée après les couches qui permettent de réaliser les composants actifs. Les connexions font donc partie intégrante des cellules et ne nécessitent pas la création de canaux de routage. D'autre part 1°) il y a beaucoup moins de multiplexeurs : des multiplexeurs ne sont utilisés que dans le cas où un registre est relié à plusieurs bus, 2°) il est possible d'utiliser de simples points mémoire comme éléments de mémorisation, ce qui permet d'économiser de la place; il est nécessaire cependant de rajouter des préchargeurs et amplificateurs de bus. Cette approche présente toutefois 2 inconvénients par rapport à l'approche précédente :
- soit les possibilités de parallélisme sont limitées (modèle à 2 bus) soit le nombre de bus n'est pas fixé mais les cellules doivent être extensibles ou alors on doit disposer de plusieurs bibliothèques de cellules,
- la vitesse de transfert des données ne peut pas être aussi grande que dans le cas

où les chemins de données ne sont pas partagés.

4. Synthèse de la partie opérative

Le terme synthèse (synthèse automatique de systèmes digitaux) est utilisé dans la littérature [THOM 81], [THOM 83a], [SHIV 83] pour désigner le processus de création détaillée d'un circuit intégré à partir d'une spécification abstraite.

Nous essaierons dans ce paragraphe de répertorier les différentes approches de synthèse des parties opératives, qui à partir d'une spécification comportementale génèrent un ensemble de primitives matérielles de base interconnectées. Un des aspects essentiels de la synthèse de systèmes digitaux est que la plupart des problèmes rencontrés sont NP complets. Il est donc hors de question d'étudier toutes les alternatives. La plupart des programmes de synthèse utilisent des heuristiques.

Nous présenterons dans cette partie les différentes étapes de la transformation d'une description comportementale en une description structurelle de niveau transfert de registres.

Le nombre des étapes de la synthèse dépend du niveau de la description originelle. La description d'entrée de SYCO n'est pas purement comportementale, car l'allocation des microinstructions, c'est à dire le séquencement des opérations et l'allocation des registres sont effectuées par le concepteur.

Nous considérons dans ce chapitre que la description est de nature purement comportementale, c'est à dire que le concepteur décrit son circuit exactement comme s'il écrivait un programme dans un langage de haut niveau, la seule restriction dont il doit tenir compte est qu'il doit n'utiliser que les opérateurs qui sont disponibles dans la bibliothèque de cellules ou que l'on sait intégrer. Dans ce cas, c'est lui même qui devra fournir les masques de l'opérateur.

Si l'on part d'une description purement comportementale (ce qui n'est pas le cas d'Apollon), on doit effectuer un certain nombre de traitements avant de procéder à l'allocation des ressources de la partie opérative. D'une part il peut être utile de transformer la description de départ pour des raisons d'optimisation. D'autre part il peut être nécessaire de séquencer les opérations décrites par le concepteur.

Le processus de synthèse peut donc être décomposé en 3 tâches essentielles :

- optimisation et transformation de la description originelle [SNOW 78], [WALK 83],
- séquencement des opérations,
- allocation des ressources.

[DIRE 81] distingue en outre une phase de la synthèse dépendante de la technologie en aval des tâches citées ci-dessus : la sélection de modules [THOM 83b].

4.1. Optimisation et transformation de la description comportementale

Nous distinguerons 2 types de modifications :

- les optimisations : ces modifications permettent de diminuer la surface

du circuit et/ou d'augmenter la vitesse d'exécution des instructions du circuit sans aucun coût supplémentaire,

- les transformations qui diminuent la surface aux dépens de la vitesse ou qui augmentent la vitesse mais en provoquant un accroissement de la surface du circuit.

4.1.1. Optimisation : [SNOW 78], [WALK 83], [RAJA 85]

1) Propagation des constantes

Les expressions ne contenant que des constantes sont remplacées par leurs valeurs. Exemple: la séquence suivante :

```
B := 1;
A := B+2;
C := A+B;
```

devient :

```
B := 1;
A := 3;
C := 4;
```

2) Mise sous forme canonique des opérations

Les opérandes des opérations commutatives sont réordonnés de façon à se ramener à un minimum d'opérations différentes et à mettre les opérandes toujours à la même place (1° ou 2° entrée de l'opérateur) dans la mesure du possible. Cette transformation peut permettre d'économiser des multiplexeurs [RAJA 85]. Cette mise sous forme canonique n'est pas nécessaire dans le cas d'Apollon; la commutativité des opérations est prise en compte lors de la phase d'allocation .

3) Elimination des expressions redondantes

On détecte les opérations de même type qui ont les mêmes opérandes. Les opérandes doivent être situés dans les mêmes registres et aucun des 2 registres sources ne doit avoir changé de valeur entre temps. On remplace les opérations redondantes par la lecture du registre où a été conservé le résultat de la première occurrence de cette opération. Si la première occurrence apparaît dans une expression complexe, on est alors amené à scinder cette expression en au moins 2 expressions pour faire apparaître le résultat de la 1° occurrence de l'opération redondante.

4) Détection des opérateurs unaires

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

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

Cette transformation permet par exemple d'utiliser un incrémenteur plutôt qu'un

additionneur pour faire des incrémentations, ce qui est intéressant car un incrémenteur occupe une surface moins importante qu'un additionneur.

5) Déplacement des actions invariantes à l'extérieur des boucles
On déplace les actions invariantes qui sont situées à l'intérieur des boucles afin de minimiser le nombre de fois où elles sont exécutées [AHO 77].

6) Expansion des procédures appelées une seule fois.

4.1.2. Transformation de la description

1) Factorisation d'un CASE
Il peut être utile de factoriser les expressions communes à différentes alternatives dans les expressions conditionnelles (IF, CASE) ou de transformer un CASE en un produit de IFs en parallèle soit afin de pouvoir reloger ces expressions dans d'autres parties de la description soit afin de décomposer un CASE en un certain nombre de IFs qui pourront être exécutés séquentiellement.

Inversement il peut être utile de transformer une séquence de IFs en une instruction de type CASE.

2) Expansion de procédures
Si le langage de description de circuits utilisé pour spécifier l'algorithme de fonctionnnement permet d'utiliser des procédures, il peut être utile de remplacer les appels de ces procédures par des copies des procédures appelées. Cette transformation est d'ailleurs effectuée systématiquement dans le cas où il n'y a pas de mécanisme spécifique pour exécuter des procédures. Mais dans le cas où les appels de procédures sont interprétés de manière spécifique par la machine (empilement des niveaux d'interprétation [JERR 86], saut à des microsubroutines dans la "mémoire" microprogrammée lié à une pile à une pile d'adresses de retour et d'un pointeur de pile [RAJA 85], il peut être néanmoins utile de remplacer l'appel d'une procédure par sa copie, spécialement dans le cas où une procédure n'est appelée qu'une seule fois (le concepteur n'a introduit une procédure que pour mieux structurer la description du circuit) ou dans le cas où une procédure est très courte car son expansion ne conduira pas à une augmentation significative de la taille de la mémoire de contrôle
Cette transformation augmente la vitesse de fonctionnement de la partie contrôle. En effet les appels de procédures nécessitent soit un délai supplémentaire pour la manipulation de la pile d'adresses de retour, soit la création d'un niveau supplémentaire d'interprétation [JERR 85] ce qui se traduit par un temps d'attente plus long des comptes rendus de la partie opérative et en général un ralentissement du fonctionnement de la partie contrôle. Si la partie contrôle doit être

resynchronisée à la suite d'un branchement nécessitant l'évaluation des comptes rendus de la partie opérative au niveau de l'automate de contrôle global, le temps d'attente des comptes rendus est rallongé d'un cycle.

Inversement il peut être utile de remplacer des séquences d'instructions communes par des appels de procédures pour diminuer le nombre de microinstructions mémorisées.

3) Déroulement des boucles :

Le déroulement des boucles consiste à recopier le corps d'une boucle autant de fois que la boucle doit être exécutée afin d'éliminer le compteur de boucle et de simplifier les expressions ne contenant que des constantes. Cette transformation aboutit à une augmentation de la taille de la partie contrôle puisque certaines microinstructions sont dupliquées. En contrepartie la taille de la partie opérative peut être réduite car un certain nombre d'opérations peuvent être remplacées par de simples transferts.

4.2, Séquencement des opérations : [DIRE 81], [N.PARK 85], [A.PARK 86], [N.PARK 86]

Il existe en gros 2 types de description :

- soit le concepteur établit le séquencement des actions opératives
- soit c'est le compilateur qui établit et optimise le séquencement à partir du séquencement établi par le concepteur.

Dans ce dernier cas, le concepteur est alors libre de décrire des expressions complexes. C'est le compilateur qui décompose ces expressions complexes en une séquence d'opérations tout en assignant les résultats intermédiaires à des registres. Si le concepteur a déjà décomposé une expression complexe en une séquence d'opérations élémentaires, on peut cependant facilement optimiser ce séquencement en transformant la description de départ en un graphe orienté acyclique qui permet de représenter le flot des données où l'on ne représente pas les éléments de mémorisation mais seulement les opérateurs. Cette forme de représentation permet de travailler directement sur les valeurs des données et non pas sur les éléments de mémorisation où ces valeurs sont conservées. Les arcs du graphe représentent les valeurs et les sommets représentent les opérateurs. C'est cette approche qui est utilisée par le CMUDA [SNOW 78], le graphe de rerésentation du flot de données est désigné par le terme "value trace".

De nombreuses techniques de compactage global de microcode ont été recensées [AGER 76], [ISOD 85], [FISH 81]. Ces techniques ont pour but de paralléliser au maximum la description de départ, c'est à dire de minimiser le nombre de microinstructions et en particulier le nombre d'instructions opératives.
[TSEN 83] propose d'utiliser l'algorithme de compactage local de microcode du premier venu premier servi : on parcourt séquentiellement la liste des opérations et

on place une opération dans la première instruction après celle où tous ses opérandes sont définis.

Les techniques de compactage local permettent de séquencer les actions d'un bloc de base de façon à diminuer le nombre d'instructions tout en tenant compte des relations de précédence et du nombre de processeurs disponibles à chaque cycle.
Un bloc de base est défini comme une séquence linéaire d'instructions n'ayant qu'un point d'entrée, la 1° instruction exécutée et qu'un point de sortie , la dernière exécutée. Les techniques de compactage local sont efficaces [LAND 80]. Le compactage global est beaucoup plus complexe. Un algorithme simple consiste à diviser un programme en blocs de base et à appliquer les techniques de compactage local à chacun de ces blocs. La grande majorité des possibilités de parallélisme réside en dehors des blocs. En effet, les blocs de microcode sont souvent courts et le microcode obtenu à l'aide de cet algorithme simple est souvent plein de trous.

Plusieurs types de contraintes doivent être prises en compte pour optimiser le séquencement :

- les contraintes de précédence : une opération ne pourra être effectuée avant une opération qui définit un de ses opérandes.

- les contraintes matérielles : le nombre des opérateurs disponibles est généralement limité. En outre, on peut souhaiter effectuer une optimisation du séquencement après la génération de l'architecture de la partie opérative. Dans ce cas on doit alors tenir compte de l'architecture existante et ne pas regrouper dans une même instruction des opérations qui utilisent les mêmes ressources (bus, opérateurs).

- les contraintes imposées par le fonctionnement en pipeline du contrôleur [DEMA 86]. Il faut un certain délai entre l'évaluation des comptes rendus et l'exécution de l'instruction conditionnelle qui dépend du résultat de cette évaluation.

D'autre part l'optimisation du séquencement peut être étendue à la définition des horloges et au fonctionnement en pipeline : le programme MAHA [A.PARK 86] permet d'optimiser le temps d'exécution d'une séquence d'opérations. Un graphe orienté acyclique représentant le flot des données dont les noeuds sont les opérateurs et les arcs représentent des valeurs, est décomposé en un certain nombre de sous-graphes. Le nombre de sous-graphes est égal au nombre de cycles nécessaires à l'exécution de cette séquence d'opérations ; il est calculé en fonction des contraintes matérielles : nombre d'opérateurs disponibles et temps d'exécution des opérateurs. Les opérations d'un sous-graphe doivent être exécutées en un temps donné, ce temps de cycle est le même pour chaque sous-graphe.

Plusieurs opérations dont l'une utilise les résultats de l'autre peuvent être exécutées au même cycle du moment que le temps nécessaire pour exécuter ces

opérations ne dépasse pas la durée du cycle. Ainsi si la longueur du chemin critique est n, où n est le nombre d'opérateurs du graphe, le nombre de cycles sera inférieur ou égal à n mais pas forcément égal. MAHA utilise un algorithme du type premier venu premier servi en commençant par l'allocation des opérations du chemin critique, puis en continuant par l'allocation des opérateurs des chemins restants les plus longs.

SEHWA [N.PARK 86] est un programme de séquencement pour la synthèse des parties opératives fonctionnant en pipeline. Le but de SEHWA est de trouver un séquencement qui permette un recouvrement des tâches tout en tenant compte des contraintes matérielles.

Le fonctionnement en pipeline est caractérisé par l'exécution en parallèle de sous tâches de tâches consécutives sur différentes parties du circuit. Les tâches consécutives sont démarrées à des intervalles fixes (la longueur de l'intervalle est un multiple entier du cycle d'horloge de base) et à chaque cycle des sous tâches différentes des différentes tâches en cours sont exécutées en parallèle.

Pour une séquence donnée, la performance optimale en mode pipeline est déterminée par la période minimum du cycle de base, par la longueur de l'intervalle qui sépare 2 tâches consécutives et par le nombre de cycles nécessaires à l'exécution de la tâche.

$T = [n-1]ltcy + Ptcy$ l : temps de latence

n : nombre de tâches à exécuter

P : nombre de cycles nécessaires à l'exécution d'une tâche.

tcy : durée du cycle d'horloge de base.

Cette formule s'applique si aucune tâche ne nécessite une resynchronisation. Soit nb le nombre de tâches nécessitant une resynchronisation (s'il y a branchement par exemple) le temps d'exécution de n tâches devient :

$T = (n-nb-1)ltcy + P(nb+1)tcy$.

Un certain nombre d'algorithmes dont le temps d'exécution est une fonction polynôme du nombre de noeuds du graphe du flot des données sont proposés dans [N. PARK 86].

Les contraintes à respecter sont les mêmes que celles énoncées pour l'optimisation du séquencement sans pipeline si ce n'est que les conflits d'utilisation des ressources ne sont plus locaux à un cycle mais qu'ils concernent tous les cycles dont le reste du quotient du numéro par l est le même, où l est le nombre de cycles séparant le démarrage de 2 tâches successives.

Pour représenter ces contraintes, [N.PARK 86] utilise un tableau d'allocation à l colonnes et n lignes, où l est le nombre de cycles de latence et n est le nombre d'opérateurs disponibles.

4.3. Allocation des ressources

Une fois que la description originelle a été optimisée et séquencée, le processus d'allocation proprement dit peut commencer. Le but de l'étape d'allocation des ressources [DEMA 86] est de plaquer la description algorithmique sur une architecture qui minimise la surface et qui permette d'exécuter l'algorithme en un certain temps spécifié au départ. Le résultat de cette opération est la définition de la structure de la partie opérative du processeur et la description de la partie contrôle sous la forme d'une boîte noire devant exécuter une liste non ordonnée d'opérations sur la partie opérative que l'on vient de définir.
L'allocation consiste à établir une correspondance entre les primitives du langage de description comportementale et les primitives matérielles dont on dispose. Le processus d'allocation dépend des primitives matérielles que l'on utilise et du style de conception adoptée. C'est pourquoi nous présenterons successivement l'allocation des différentes primitives matérielles et pour chacune de ces primitives nous aborderons les problèmes de l'allocation en fonction des différents styles de conception définis précédemment [JAMI 86].
Les primitives de description comportementale des actions opératives sont les opérateurs et les variables de mémorisation.
Les primitives matérielles sont les registres, les constantes, les unités de traitement mono- ou multifonctions et les éléments d'interconnexion (multiplexeurs ou bus selon le style de conception adopté).
On peut remarquer que les différentes tâches de la synthèse optimisation, séquencement et allocation ne sont pas indépendantes l'une de l'autre.
Certaines optimisations modifient le séquencement. Le séquencement peut être considéré comme une première phase de l'allocation.
En effet, lors de l'assignation des opérations aux différentes instructions opératives, le nombre maximal d'unités de traitement nécessaires est déterminé (que ce nombre soit borné ou non par le concepteur) et un certain nombre de variables intermédiaires auxquelles il faudra assigner des registres sont générées.
Nous allons successivement examiner les problèmes d'allocation des différents types de composants de la partie opérative.

4.3.1. Allocation des registres

L'allocation des registres consiste à établir une correspondance entre les variables algorithmiques utilisées dans la description comportementale et les registres de la partie opérative. Le concepteur peut déclarer un certain nombre de registres si bien que l'allocateur de registres ne cherchera pas à optimiser l'utilisation de ces registres.
Par contre, les variables de mémorisation qu'elles soient introduites par le concepteur mais non déclarées comme registres ou qu'elles soient générées durant la phase d'allocation des microinstructions doivent être assignées à un certain nombre de registres que l'on veut minimiser. Certaines des variables de mémorisation peuvent d'ailleurs être supprimées lors de la phase de séquencement des opérations.
Peu de systèmes de synthèse décrits dans la littérature minimisent le nombre de

registres de la partie opérative. Seul Facet [TSEN 83] donne un algorithme de minimisation du nombre des éléments de mémorisation de la partie opérative.
Le problème est le suivant :
Il est en général intéressant d'assigner plus d'une variable au même élément physique de mémorisation.
Pour spécifier les conditions suffisantes pour pouvoir combiner 2 variables, il est nécessaire de rappeler la définition de la durée de vie d'une variable.

Une variable est dite active ou vivante entre l'instant où elle est définie et l'instant où elle est utilisée pour la dernière fois.
Une variable est dite inactive ou morte entre l'instant de sa dernière utilisation et l'instant de sa nouvelle définition.
Deux variables peuvent être combinées si elles ne sont jamais actives en même temps.
Cette contrainte n'est pas nécessaire. Deux variables peuvent être combinées si le seul moment où elles sont actives en même temps, une des variables est utilisée comme source et l'autre comme destination dans une même action opérative.
On construit le graphe de compatibilité de durée de vie des variables, où les nœuds sont les variables et les arcs relient 2 variables qui ne sont jamais actives en même temps. Les variables qui peuvent être combinées (s'il y a n variables il doit y avoir n(n-1)/2 relations de compatibilité) forment une clique. Le problème de minimisation du nombre d'éléments de mémorisation peut être ramené au problème de minimisation du nombre des cliques disjointes d'un graphe. La recherche des cliques d'un graphe est un problème NP-complet. [TSEN 83] décrit une procédure qui décompose un graphe en un nombre pratiquement minimum de cliques disjointes en un temps qui est une fonction polynômiale du nombre de nœuds et d'arcs du graphe.
On commence par grouper les nœuds qui ont le plus de voisins. Cet algorithme permet aussi de grouper en priorité les nœuds d'un sous graphe quelconque. On peut ainsi grouper en priorité des nœuds pour lesquels on sait que le gain résultant de ce groupement est plus important que le gain résultant du groupement d'autres ensembles de nœuds du graphe.
En particulier, si on combine 2 variables entre lesquelles existent des transferts simples (1 ou 2), on pourra supprimer ces transferts de la description et ainsi diminuer le nombre de microinstructions ou diminuer le nombre des composants de la partie opérative. Pour grouper en priorité ces variables, il suffit en utilisant la procédure de [TSEN 83] de définir un sous graphe de compatibilité, où 2 variables sont reliées par un arc si elles ont des durées de vie compatibles et s'il existe au moins un transfert entre ces 2 variables.
Il est possible que l'on puisse diminuer davantage le nombre des microinstructions si on réapplique la procédure de compactage des microinstructions après avoir renommé les variables et supprimé les transferts d'une variable dans elle même.
La détermination des durées de vie des variables est un problème classique, bien connu des réalisateurs des compilateurs [AHO 77].

Le problème de la formation des bancs de registres peut aussi être ramené au problème de décomposition en cliques. Les registres qui ne sont pas accédés en même temps peuvent être groupés dans un banc de registres. Le problème dépend du mode de connexion des registres du banc au reste de la partie opérative.
Nous distinguerons 2 cas :

1) Le banc de registres possède un chemin de lecture (sortie) et un chemin d'écriture (entrée) qui sont distincts. Deux registres d'un même banc ne peuvent être accédés simultanément en lecture ou en écriture. Par contre un registre peut être accédé en lecture et l'autre en écriture.
2) Le banc de registres possède 2 chemins d'accès, chacun des chemins peut être utilisé pour la lecture ou pour l'écriture. (Ce qui est le cas d'Apollon)
   Si l'on suppose que tout élément de la partie opérative peut être connecté à 2 bus, au plus 2 registres d'un même banc peuvent être accédés simultanément; peu importe que ce soit en lecture ou en écriture.

A partir de la description comportementale de la partie opérative, on construit un graphe de compatibilité d'accès, puis on extrait les cliques. Mais cette fois, le but n'est pas d'obtenir le minimum de cliques disjointes, mais d'effectuer une partition de l'ensemble des nœuds en cliques, de façon à minimiser le nombre des commandes de la partie opérative. Seuls les bancs de plus de 3 registres présentent un intérêt, car les bancs de 2 registres ne permettent pas de diminuer le nombre de commandes. Il y a 3 états à coder (R_1, R_2, ni R_1 ni R_2) et il faut dont 2 commandes.

Ce n'est pas le nombre de cliques de plus de 3 nœuds qui est important non plus, mais le nombre global de commandes gagnées.

Exemple
Considérons 15 registres que l'on cherche à regrouper en bancs.
La formation de bancs de 3 registres permet de gagner $2(3-\log_2(3+1)) = 2$ commandes.
La formation d'un banc de 10 registres pemet de gagner $2(10-(E(\log_2(10+1))+1) =$ 12 commandes; E(x) désigne la partie entière de x.
Supposons que l'on ait le choix entre une partition en 5 bancs de 3 registres chacun et une partition en un banc de 10 registres et 5 registres qui ne rentrent dans aucun banc.
La première partition permet d'économiser 10 commandes alors que la seconde permet d'en économiser 12.

Le principal effet de l'optimisation du nombre d'éléments de mémorisation est bien sûr de diminuer le nombre de composants de la partie opérative. Il y a cependant un effet secondaire qui peut ne pas être bénéfique. Les connexions sont modifiées.
On peut d'ailleurs distinguer 2 cas :

- Parties opératives à multiplexeurs
Soient A et B, 2 variables combinables,
3 cas peuvent être distingués :

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

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

3) Autres cas. La fusion de A et B peut entraîner selon les cas la création d'un multiplexeur ou la création d'un démultiplexeur.

En ce qui concerne les parties opératives à bus, dont le nombre et la structure sont régis par un modèle architectural, les effets secondaires de l'optimisation du nombre de registres sont généralement bénéfiques. Des connexions peuvent être créées ou suprimées selon le type d'instruction et la structure des connexions de la partie opérative. Ce raisonnement ne peut être fait qu'une fois que la partie opérative est construite car une correspondance du comportemental au structurel ne peut être faite aussi facilement que dans le cas des parties opératives à multiplexeurs où aucune contrainte architecturale n'est imposée.

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

Le principal problème d'Apollon est que l'on ne peut pas toujours connecter directement (c'est à dire au moyen d'un seul segment) 2 composants de la partie opérative et donc que l'on peut rencontrer des situations de verrou.
Plusieurs solutions peuvent être adoptées pour contourner ce problème :

- soit on augmente le nombre de bus,
- soit on augmente le nombre de cycles imparti à l'exécution de la ou des instructions créant le verrou, ce qui revient à ralentir l'exécution,

- soit on procède à l'opération inverse de la minimisation du nombre des registres : on les duplique de façon à diminuer le nombre des contraintes d'utilisation des bus.

Exemple :

soit l'instruction suivante comportant 3 opérations binaires en parallèle :
(A+B; B+C; C+A;)
peu importent les destinations.
Les opérandes ne peuvent être chargés en un cycle sur une partie opérative à 2 bus segmentés.
Dupliquons un des registres, mettons A

soit
D := A;
(A+B; B+C; C+D;)

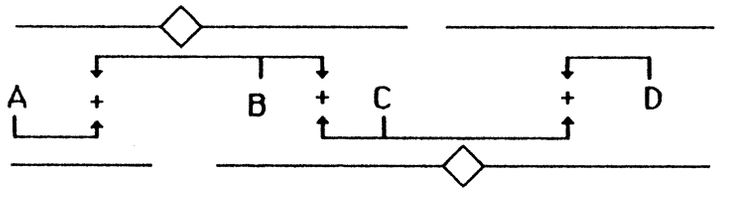


figure 2.7 : chargement des opérandes en un cycle

On insère dans la description comportementale avant l'instruction qui provoque le verrou une instruction qui recopie le contenu du registre source dans D.

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

Ces modifications peuvent d'ailleurs être gratuites. S'il existe une variable qui est morte (du point de vue de la durée de vie d'une variable), il suffit de recopier le registre à dupliquer dans ce registre, il n'est pas nécessaire de créer un nouveau registre.

D'autre part il peut être possible d'insérer l'opération de recopie dans une instruction existante, ce qui permet de ne pas rajouter d'instructions.

Nous donnerons un algorithme détaillé de l'allocation des ressources dans le chapitre 3.

Les registres qui sont utilisés à la fois comme sources et destinations de transferts simples au même instant doivent être réalisés à l'aide de maîtres-esclaves ou à l'aide de bascules synchronisées par le front d'une horloge (par front) de façon à ce que les registres soient lus avant que leurs valeurs ne soient modifiées.
Ainsi Macpitts [SISK 82] utilise systématiquemet des maîtres esclaves pour réaliser des registres.

Ce n'est pas le cas d'Apollon qui permet pourtant d'exécuter des permutations de registres ou d'autres opérations où à la fois un registre est consulté et sa valeur

modifiée. En effet, le problème est complètement différent : Apollon utilise des points mémoire (2 inverseurs rebouclés) qui peuvent être reliés aux 2 bus du système de connexion de la partie opérative.Il suffit alors que les phases de lecture et d'écriture soient non recouvrantes. On utilise un bus pour la lecture et l'autre bus pour l'écriture.

4.3.2. Allocation des constantes

Les constantes par définition ne sont que lues. On peut donc les dupliquer sans problème à l'inverse des registres. Le problème d'allocation des constantes se pose en des termes différents selon la nature du modèle architectural de la partie opérative

- s'il n'y a pas de contraintes architecturales (limites du nombre d'éléments d'interconnexion), seule une instance d'une constante est nécessaire,
- si le nombre de bus est limité, plusieurs instances d'une même constante peuvent être requises.

Exemple :
soit à exécuter l'instruction suivante en 2 cycles sur une partie opérative à 2 bus segmentés :
(A+3; A+B; B+C; C+D; D+3;)
peu importent les destinations.

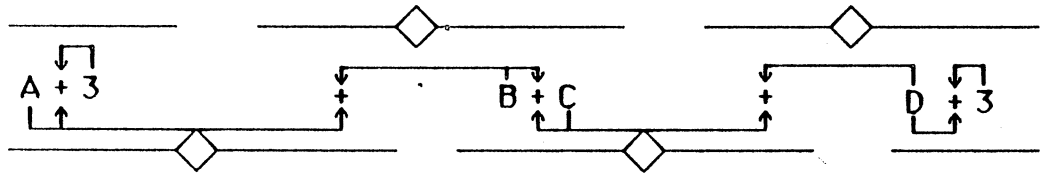


figure 2.8 : duplication de la constante 3

Pour que les opérandes puissent être transférés en un cycle, il faut dupliquer la constante 3 (cf chapitre 3).
Si un nombre important de constantes sont utilisées dans la description, on peut les regrouper dans une ROM, on gagne ainsi de la place mais seule une constante par cycle peut être accédée.
En ce qui concerne le modèle à 2 bus d'Apollon, on peut multiplexer les commandes des constantes reliées à un même segment ; en effet on ne peut lire qu'une constante par segment de bus.
Nous avons vu que lors de l'optimisation, on remplace les expressions ne contenant que des constantes par leurs valeurs et l'on propage les constantes. Cette optimisation qui a pour effet principal de réduire le nombre des opérations peut aboutir à une augmentation du nombre des constantes.

4.3.3. Allocation des opérateurs

Les outils de synthèse automatique génèrent autant d'opérateurs qu'il y a

d'opérations simultanées dans la description comportementale. Différentes occurrences (non simultanées) d'une même opération peuvent donner lieu à la création d'autant d'opérateurs identiques qu'il y a d'opérations identiques, ou à la création d'un seul opérateur. Différentes occurrences de différentes opérations non simultanées peuvent donner lieu à la création d'opérateurs spécialisés ou d'opérateurs universels (UALs).
Un programme de minimisation du nombre des opérateurs se doit de tenir compte des points suivants :

- est ce que les opérations sont simultanées ?
- est ce que l'on cherche à combiner plusieurs opérateurs et à introduire des opérateurs universels ? Si oui : est ce que l'on a intérêt à regrouper les opérations qui ne portent pas sur le même nombre de bits ?
- quelles sont les opérations que l'on peut effectuer sur un opérateur universel, comme une UAL ?
- Et finalement est ce qu'il existe des contraintes :
  * matérielles : le nombre et la nature des opérateurs disponibles sont-ils fixés ?
  * temporelles : la durée des opérations est-elle limitée ?
  * architecturales : quels sont les éléments d'interconnexion utilisés ? Le nombre de ces éléments est-il limité ?

Comme pour les autres composants, nous distinguerons 2 cas principaux selon que la puissance du modèle architectural adopté est limitée ou non :
Si la puissance du modèle est limitée, la minimisation du nombre des opérateurs n'est pas le problème majeur (cf chapitre 3). D'ailleurs certains opérateurs peuvent être dupliqués. Nous donnerons 2 exemples :

1) Soient les 3 instructions opératives suivantes :
(A+B ; A or C;)
(B+C ; B or A;)
(C+A ; C or B;)

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

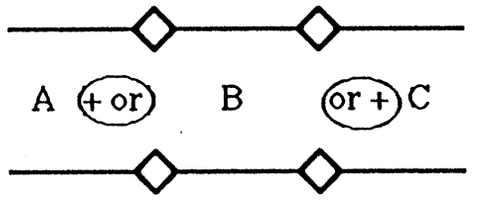


figure 2.9 : duplication des opérateurs + et or

2) Soient les 3 instructions opératives suivantes:
($A:=X_1+X_2$; $B:=Y_1+Y_2$; $C:=Z_1+Z_2$;)

($B:=X_1+X_2$; $C:=Y_1+Y_2$; $A:=Z_1+Z_2$;)
($C:=X_1+X_2$; $A:=Y_1+Y_2$; $B:=Z_1+Z_2$;)

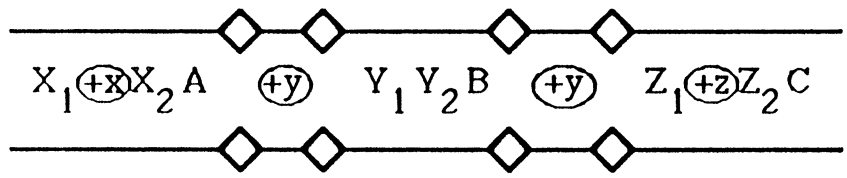


figure 2.10 : ajout d'un quatrième additionneur

4 additioneurs sont nécessaires alors que l'on n'effectue jamais plus de 3 additions en même temps.
Comme on ne peut pas toujours connecter directement un opérateur à ses opérandes ni dupliquer les registres opérandes, on est conduit à dupliquer les opérateurs pour pouvoir exécuter certaines instructions. L'allocation des opérateurs d'une partie opérative à 2 bus segmentés consiste à déterminer les opérations qui sont effectuées sur chaque sous partie opérative. C'est lors de la phase de génération des masques, que certains des opérateurs d'une sous-partie opérative seront combinés pour former des UALs, ou d'autres opérateurs universels s'il en existe dans la bibliothèque de composants de base utilisée.
Apollon cherche avant tout à minimiser le nombre de segments des bus utilisés pour effectuer les opérations.

Si la puissance du modèle architectural n'est pas limitée, c'est-à-dire si on peut toujours connecter 2 composants de la partie opérative, le problème de minimisation du nombre des opérateurs est essentiel car ce sont les opérateurs qui sont en général les composants les plus coûteux en surface.
Différentes approches sont possibles.
[TSEN 83] formalise à nouveau le problème de minimisation en un problème de décomposition en un nombre minimal de cliques disjointes. L'allocation des opérateurs consiste en 2 tâches :
l'une est de combiner les opérateurs de même type, l'autre est de grouper les opérateurs de différents types pour former des opérateurs universels.
[TSEN 83] tient compte de la structure d'interconnexion des opérateurs.
Par exemple si on regroupe 2 opérateurs dont les sources et les destinations sont différentes, on doit créer 2 multiplexeurs et un démultiplexeur alors que si on regroupe 2 opérateurs qui ont mêmes sources et même destination, on économise 2 démultiplexeurs et un multiplexeur.
8 types de relation de groupement sont ainsi définis, selon qu'une paire, les 2 paires d'opérandes, les destinations et les opérateurs sont identiques ou non.
Tseng cherche à grouper en priorité les opérateurs des opérations identiques, puis les opérateurs des opérations dont un seul des éléments (opérande 1, destination, opérande 2, opérateur) est différent.
Mc Farland [MCFA 83], [MCFA 86] partitionne les opérateurs en un ensemble de classes. Une classe est constituée d'un ensemble d'opérateurs que l'on a intérêt à

regrouper car
- ils peuvent utiliser les mêmes composants matériels,
- on peut ainsi réduire le nombre de connexions entre les classes de la partition.

La méthode de partition est basée sur la définition d'une distance qui exprime l'intérêt relatif que l'on a à regrouper les opérateurs dans une même classe.
Cette distance est fonction du type des opérateurs, de leur degré d'interconnexion et de leur parallélisme potentiel. Le résultat de cette partition est ensuite utilisé lors de la phase d'allocation proprement dite. Cette méthode [MCFA 86] permet en outre d'obtenir facilement une évaluation de la surface et de la vitesse du circuit.

Remarque
L'allocation des ressources est en général effectuée après une phase d'allocation des microinstructions. Cette phase de séquencement peut tenir compte des regroupements hiérarchiques du type de ceux effectués par McFarland pour minimiser le nombre des opérateurs.

4.3.4. Allocation des éléments d'interconnexion

Dans un premier temps nous définirons les différents éléments d'interconnexion.
Ensuite nous donnerons les principales caractéristiques des parties opératives à multiplexeurs et des parties opératives à bus avant de recenser les différentes approches d'allocation des éléments d'interconnexion.

4.3.4.1. Définitions

[TSEN 81] distingue 6 types de primitives d'interconnexion :

- un ensemble de fils simples : c'est l'élément d'interconnexion le plus simple. Il permet de relier directement 2 composants de la partie opérative.

- un multiplexeur : un multiplexeur peut être défini comme une boîte noire à entrées multiples et à une seule sortie. La sortie est égale à l'entrée qui est sélectionnée par les fils d'adressage du multiplexeur

- un démultiplexeur : un démultiplexeur peut être défini comme une boîte noire à une entrée et à sorties multiples. Seule une des sorties est active à un moment donné (basse impédance).

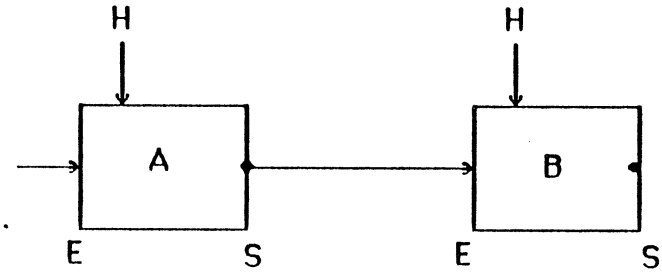
- un bus général ou bus est une primitive d'interconnexion à entrées et à sorties multiples. Chaque entrée et chaque sortie est connectée au bus par l'intermédiaire d'une porte de transfert. Quand la porte de transfert est désactivée, la connexion est coupée. Deux autres types d'éléments d'interconnexion peuvent être définis. Ce sont des formes dégénérées d'un bus

général.

- un arbre de multiplexage : c'est un bus à plusieurs entrées et une seule sortie. Seule une entrée peut être active. A la différence d'un multiplexeur, les portes de transferts sont contrôlées de façon indépendante.

- un arbre de transmission (ou de démultiplexage) : c'est un bus à plusieurs sorties et une seule entrée. A la différence d'un démultiplexeur, plusieurs sorties peuvent être actives et donc les portes de transfert sont contrôlées de façon indépendante alors que les commandes d'un démultiplexeur sont codées. Par définition les multiplexeurs et les démultiplexeurs sont des éléments d'interconnexion unidirectionnels.

De simples fils d'interconnexion sont en général unidirectionnels. Ils ne peuvent être bidirectionnels que : si les points d'entrée et de sortie sont identiques (point mémoire connecté à 2 bus complémentés) et s'il y a 2 portes de transfert sur ces fils, synchronisées par des horloges de lecture et d'écriture non recouvrantes.



Fils de connexion unidirectionnels

figure 2.11

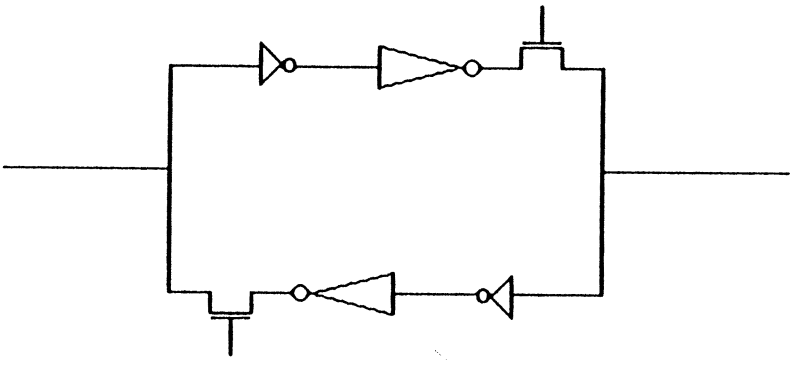
Un bus peut être unidirectionnel ou bidirectionnel.

Un élément d'interconnexion est dit bidirectionnel s'il est symétrique vis à vis de la direction du transfert.

Un bus est unidirectionnel dans le cas contraire.

L'avantage d'utiliser des bus unidirectionnels réside dans le fait que l'on peut insérer sans problème des portes logiques au milieu des bus. Avec les bus bidirectionnels, on doit dupliquer les portes logiques.

Exemple : amplification de bus bidirectionnel



Amplificateur bidirectionnel

figure 2.12

Si l'on utilise des bus unidirectionnels, on a besoin de davantage de bus pour connecter les composants de la partie opérative. L'ordre des registres connectés à un bus unidirectionnel détermine les possibilités de transfert. Un générateur automatique de parties opératives à bus unidirectionnels devra donc ordonner les registres de façon à minimiser le nombre de bus.
Dans la suite nous considèrerons que les bus sont bidirectionnels.

4.3.4.2. Parties opératives à multiplexeurs

L'utilisation de multiplexeurs permet une approche simple et systématique des problèmes d'interconnexion.
On place devant chaque entrée d'élément de mémorisation ou d'élément de traitement un multiplexeur à n entrées où n est le nombre des composants de la partie opérative qui peuvent accéder à ce registre. On considère bien sûr que les éléments d'interconnexion sont unidirectionnels. Comme aucun chemin de données n'est partagé, cette approche est assez coûteuse : le nombre d'éléments d'interconnexion est élevé.

Par exemple : si on veut pouvoir effectuer les transferts suivants :

```
D1:=A;    D1:=C;
D2:=A;    D2:=C;
D3:=A;    D3:=C;
D1:=B;
D2:=B;
D3:=B;
```

on doit utiliser 3 multiplexeurs.

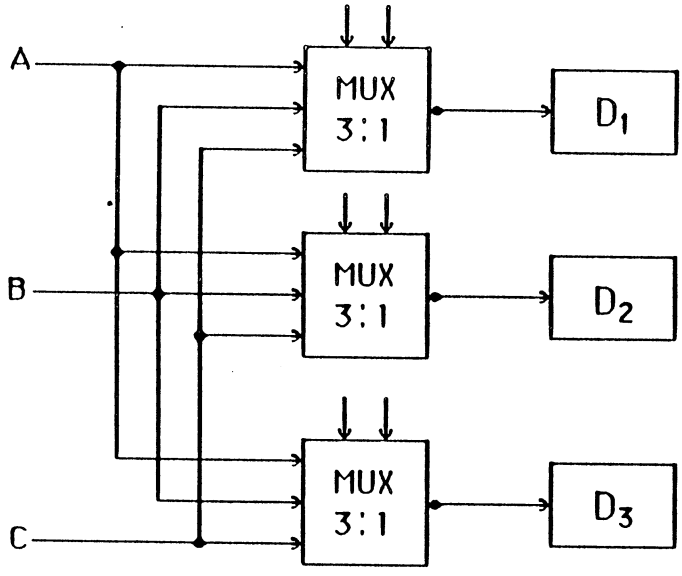


figure 2.13 : parties opératives à multiplexeurs

Alors que l'utilisation de bus permet de réduire le nombre de portes de transferts de 6, le nombre de lignes de connexions de 8 et le nombre de commandes de 3.

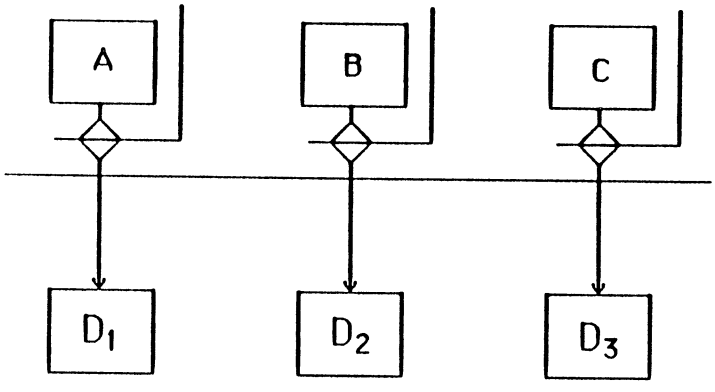


figure 2.14 : parties opératives à bus

On suppose que A, B, C, D1, D2 et D3 sont réalisés à l'aide de bascules. La porte de transfert d'accès en écriture d'une bascule n'est pas représentée ; on suppose qu'elle est interne à la bascule.

Une autre approche des problèmes d'interconnexion consiste à utiliser des arbres de transmission (nous emploierons le terme démultiplexeur même si plusieurs sorties peuvent être actives). On place derrière la sortie de chaque composant qui est à l'origine de plusieurs transferts un démultiplexeur qui a autant de branches qu'il y a de destinations qui peuvent être accédées par ce composant.

4.3.4.3. Parties opératives à bus

La structure d'interconnexion des composants d'une partie opérative est caractérisée d'une part par le nombre de portes de transfert et d'autre part par le nombre de lignes de connexion.

Nous avons vu que l'utilisation de multiplexeurs permet une approche simple et systématique. L'utilisation de bus permet de diminuer le nombre des lignes de connexion et peut permettre de diminuer le nombre de portes de transfert.

Ainsi la partie opérative de la figure 13 a 12 portes de transferts et 9 lignes de connexion alors que la partie opérative de la figure 14 a 6 portes de transferts et 1 ligne de connexion.

Le nombre minimum de lignes de connexion (ou de bus) est donné par le nombre maximum de transferts simultanés.

Le nombre minimum de portes de transferts ne peut être déterminé qu'à la suite d'une recherche combinatoire des solutions du problème d'interconnexion.

Ce nombre de portes de transferts est toutefois borné.

Ainsi s'il y a n registres à interconnecter il y a au plus n(n-1) transferts possibles entre ces différents registres. Et si l'on utilise une partie opérative à multiplexeurs, il faudra au maximum n(n-1)+ n portes de transfert : (n(n-1) : portes de transfert des multiplexeurs ; n : portes de transfert d'accès en écriture des n registres).

Si l'on utilise une partie opérative à bus, il faudra au maximum n bus (il ne peut y avoir plus de n transferts simultanés). Si chaque bus est connecté aux n registres à la fois en lecture et en écriture, on obtient un maximum de $2n^2+n$ portes de transfert.

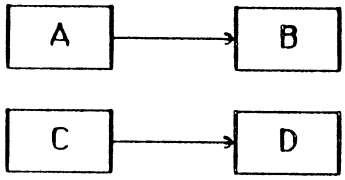
Le problème qui se pose est comment minimiser le nombre des portes de transfert sans rechercher toutes les solutions du problème d'interconnexion.

Les transferts non simultanés peuvent utiliser la même ligne de connexion. Si l'utilisation de bus permet de réduire le nombre de lignes d'interconnexion, il n'est pas du tout sûr que cela permette de diminuer le nombre de portes de transfert par rapport à l'approche multiplexeur ou démultiplexeur. En effet l'utilisation de bus ne permet pas de supprimer les multiplexeurs : si un registre est connecté en écriture à plusieurs bus, il faut placer un multiplexeur devant l'entrée de ce registre.

Même si aucun registre n'est lié à plus d'un bus, on ne peut non plus affirmer que le nombre de portes de transfert n'a pas augmenté.

Exemple :

A:=B;
C:=D;

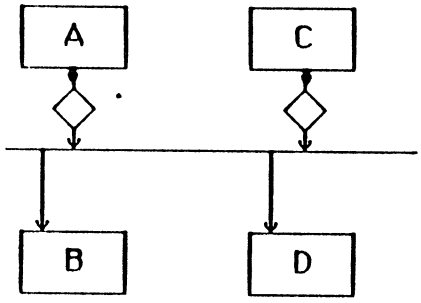


Approche multiplexeur

figure 2.15

L'approche multiplexeur (ou démultiplexeur; dans ce cas là ces deux approches sont équivalentes) nécessite en ce qui concerne les 2 transferts précédents:

- 2 portes de transfert,
- 2 lignes de connexion.



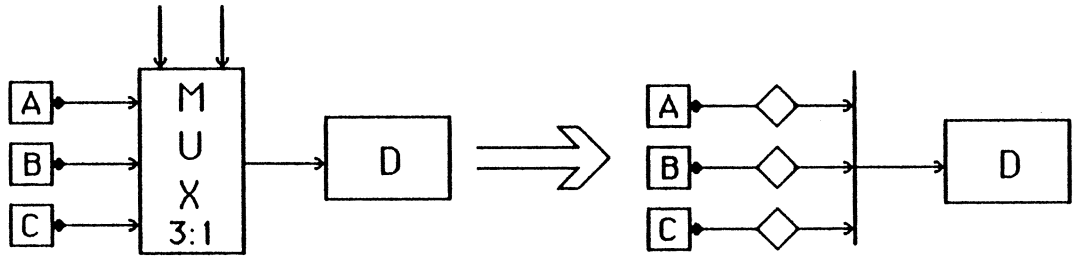
Approche bus

figure 2.16

Si l'utilisation d'un bus permet de diminuer d'un le nombre des lignes de connexion, le nombre des portes de transferts est multiplié par 2.
Il n'y a donc aucun intérêt à utiliser le même bus pour effectuer 2 transferts qui ont des sources différentes et des destinations différentes. La différence essentielle entre l'approche bus et l'approche multiplexeur-démultiplexeur est que l'on doit placer des portes de transfert à la fois à l'entrée et à la sortie des éléments à connecter quand on choisit l'approche bus car une même ligne de connexion peut être utilisée pour effectuer, de façon non simultanée bien sûr, des transferts différents.

L'approche bus peut permettre de diminuer le nombre des multiplexeurs (on l'a vu dans le premier exemple) dans le cas où plusieurs sources ont les mêmes destinations et où les transferts entre ces composants ne sont pas simultanés.

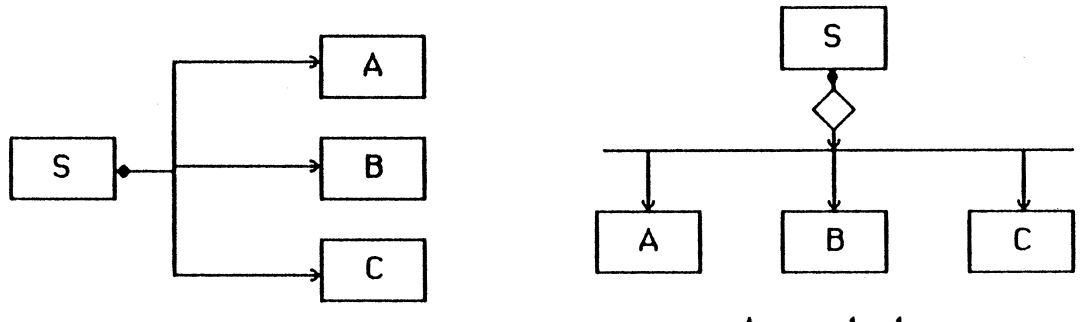
Si plusieurs sources n'ont qu'une destination commune, on ne gagne ni ne perd en remplaçant le multiplexeur par un bus



Approche multiplexeur Approche bus

figure 2.17

On augmente d'un le nombre de portes de transfert en connectant par un bus la source aux destinations des transferts qui ont cette source en commun.



Approche démultiplexeur Approche bus

figure 2.18

Cependant on ne peut pas conclure que l'approche démultiplexeur permet d'économiser des portes de transfert car on ne tient pas compte des transferts ayant pour destination A, B et C et dont la source est différente de S.
S'il n'y a qu'une seule source sur le bus, on peut remplacer le bus par un arbre de démultiplexage, ce qui permet de ne pas rajouter de portes de transfert.

4.3.4.4. Présentation de différentes approches d'allocation des éléments d'interconnexion

Le problème d'allocation des éléments d'interconnexion a été abordé d'abord par Torng et Wilhelm [TORN 77]. Torng et Wilhelm remarquent que le nombre minimum de bus est donné par le nombre maximum de transferts en parallèle. Ils utilisent alors les méthodes de la programmation dynamique pour chercher la solution au problème d'interconnexion qui minimise le nombre de portes de transferts pour un nombre de bus fixé à l'avance.

L'algorithme de Mathialagan et Biswas [MATH 80] permet de réduire le temps de calcul par rapport à l'algorithme proposé par Torng et Wilhelm.

Tseng et Siewiorek donnent 2 algorithmes pour minimiser le nombre des bus et le nombre des portes de transfert, [TSEN 81] et [TSEN 83], qui sont basés sur l'utilisation d'heuristiques.

Dans [TSEN 81], Tseng et Siewiorek proposent d'allouer les éléments d'interconnexion de la façon suivante : ils cherchent à effectuer le plus de transferts possible à l'aide d'un seul bus tout en essayant de minimiser le nombre des portes de transferts. Les bus sont créés un à un et on attribue à chaque bus nouvellement créé le groupe des transferts qui ont même source ou qui ont même destination mais qui ne sont pas simultanés, qui contient le plus de transferts, tous ces transferts devant être compatibles avec les transferts précédemment attribués au bus.

Dans [TSEN 83], Tseng et Siewiorek ramènent à nouveau le problème de minimisation des éléments d'interconnexion au problème de décomposition d'un graphe non orienté en un nombre minimal de cliques disjointes.

Auparavant, les opérandes des opérations commutatives sont réordonnés afin de minimiser le nombre de connexions. Tseng et Siewiorek partent de la constatation qu'il n'y a que peu d'intérêt à exécuter 2 transferts dont à la fois les sources et les destinations diffèrent (car on augmente alors le nombre des portes de transfert) alors qu'il peut être avantageux d'utiliser le même bus pour effectuer des transferts ayant même source et même destination.

Ils construisent un graphe dont les sommets sont des transferts et dont les arcs expriment que ces transferts ne sont pas simultanés ou ont les mêmes sources. Le sous-graphe des transferts ayant même source ou ayant même destination et n'étant pas simultanés est utilisé pour allouer d'abord des bus aux transferts de ce type. Comme les registres ne peuvent avoir qu'une entrée, des multiplexeurs sont placés devant les registres qui doivent être connectés en écriture à plusieurs bus.

Le problème d'allocation des éléments d'interconnexion est différent dans le cas d'Apollon. En effet un registre ne peut être relié directement qu'aux 2 bus qui le traversent. D'autre part les segments des 2 bus globaux parallèles, qui constituent autant de bus, peuvent être reconnectés.

Le nombre de bus est fixé à 2. Ces bus peuvent être morcelés. L'allocation des éléments d'interconnexion peut être effectuée en 2 temps :

- le nombre minimum de segments de chacun des 2 bus est d'abord déterminé en supposant que tous les composants sont à double accès,
- le nombre de portes de transmission et des commandes les contrôlant est alors minimisé.

En fait dans le système Apollon, cette allocation se fait au fur et à mesure que les instructions opératives sont analysées. Afin de minimiser le temps de traitement l'allocation des éléments d'interconnexion n'est pas décomposée en 2 phases.

Remarque

Les portes de transfert d'un registre à double accès sont constituées de 2 multiplexeurs (plus exactement 2 arbres de multiplexage).

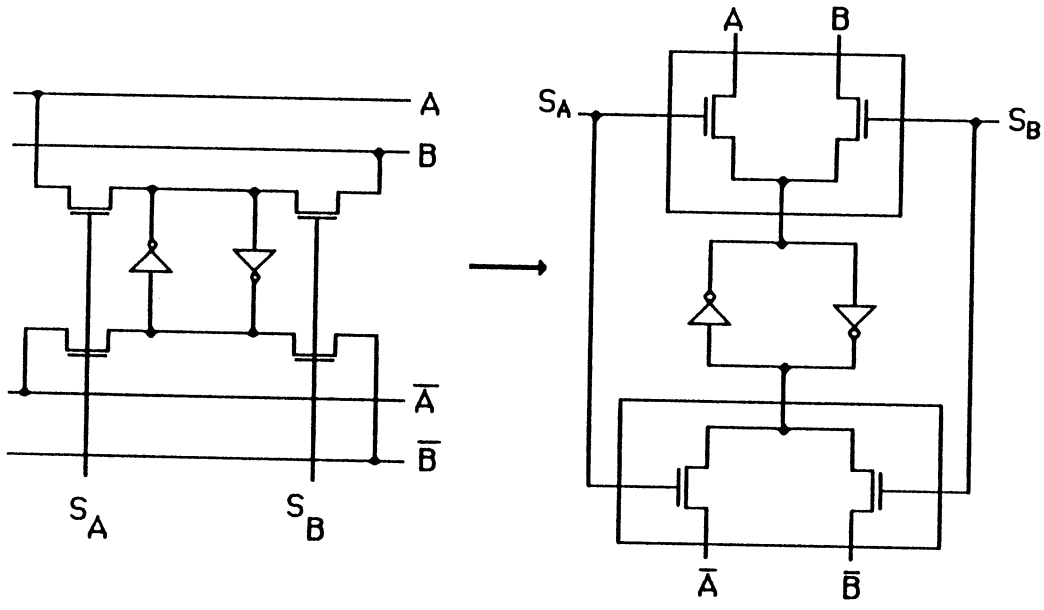


figure 2.19 : registre à double accès et multiplexeurs

5. Génération des masques de la partie opérative

Le concepteur définit d'abord l'architecture globale avant de dessiner les cellules de base de son circuit. De la même façon la génération automatique des spécifications des masques de la partie opérative peut être effectuée en 2 étapes.

1° étape : l'architecture de la partie opérative est extraite de la description comportementale initiale.

2° étape : les spécifications des masques sont générées à partir de la structure de la partie opérative définie précédemment.

Des modules générateurs de masques sont utilisés à cette fin. On fait correspondre à chacun des composants de base de la partie opérative (registre, constante, UAL, etc...) un module générateur spécifique.

La description structurelle qui est constituée d'un ensemble de composants interconnectés est transformée en une description d'assemblage constituée d'une suite d'instructions de placement et de routage portant sur les instances des modules générateurs de masques correspondants.

5.1. Module générateur de masques

Un module de génération de masques est un programme qui permet de générer les spécifications des masques d'une partie d'un circuit intégré à partir des spécifications des cellules constitutives de cette partie. Ces cellules peuvent être elles-mêmes paramétrées de façon à ce qu'on puisse contrôler leurs caractéristiques électriques (temps de propagation, puissance consommée) en fonction de leurs sortances.

La conception d'un module de génération comporte plusieurs aspects :

- structurel : quelles sont les cellules de base ? Quels sont les paramètres de l'assemblage ? Les modules de génération des éléments de la partie opérative utilisés par le système Apollon sont décrits au chapitre 3.
- topologique : quelle est l'organisation topologique de l'assemblage : comment les cellules sont elles placées les unes par rapport aux autres ? Comment sont réalisées les connexions (aboutement - routage par dévoiement - routage par croisement) ? Les problèmes topologiques sont abordés de façon générale au paragraphe 5.3.
- électrique : comment dimensionner les transistors, les bus d'alimentation ? Les problèmes électriques sont abordés au paragraphe 5.4. Dans le paragraphe 5.5, nous étudierons comment on peut adapter les modules générateurs à des contraintes de fonctionnement variables.

Pour construire un module générateur de masques, le concepteur doit avoir à sa

disposition un minimum d'outils et de ressources :

- un langage de programmation évolué qui permette d'utiliser les structures classiques de contrôle (if then else, for, while, etc...) et les mécanismes de passage des arguments des fonctions ou procédures,
- un assembleur de silicium : un certain nombre de primitives de placement et de routage doivent être accessibles à l'utilisateur. Ces primitives peuvent être données sous la forme d'un sur-ensemble d'un langage de programmation. Les spécifications d'un assembleur de silicium pour l'assemblage d'une partie opérative de type bit-slice sont données au chapitre 3,
- d'une bibliothèque de cellules de base.

Nous donnerons au paragraphe 5.6 la liste de tous les outils nécessaires à l'écriture de modules générateurs.

Le concept de module générateur de masques [BURI 86], [DEMA 86], [SCHO 83], [SHRO 82] a été introduit dans la littérature pour plusieurs raisons :

1) La notion de module générateur de masques permet de définir une interface propre entre l'architecture et la circuiterie. Les modules générateurs sont écrits par des spécialistes de la conception de circuits intégrés. La première phase de la compilation peut donc être pratiquement complètement indépendante de la phase de génération des masques.

2) Cette approche rentre parfaitement dans le cadre de la méthodologie hiérarchique descendante de la conception des circuits intégrés.
Le circuit de la partie opérative est découpé en blocs, chaque bloc étant un des composants de la partie opérative générés lors de la phase de synthèse de l'architecture de la partie opérative. Un bloc peut être lui-même décomposé en un certain nombre de cellules de base. Les modules générateurs sont utilisés pour assembler les différents blocs de la partie opérative à partir de leurs constituants de base.

3) Un module générateur est paramétré.
Cette propriété est essentielle. En effet, la paramétrisation des modules générateurs permet de réutiliser les mêmes procédures d'assemblage en changeant seulement la valeur des paramètres lorsque l'on veut générer un circuit différent. Par exemple le même programme d'assemblage peut être utilisé pour assembler un registre de 8 bits ou 16 bits, il suffit de définir le nombre de bits comme paramètre.

On peut distinguer plusieurs types de modules générateurs selon l'étendue de la paramétrisation. Les cellules de base peuvent être soit des cellules prédéfinies soit des cellules paramétrées ou compilées.

5.2. Les différents types de cellules de base

5.2.1. Cellules prédéfinies

Les cellules prédéfinies sont dimensionnées et testées pour fonctionner à une fréquence donnée. C'est l'approche que nous avons utilisée. Notre approche se différencie de l'approche de la conception dite "précaractérisée" dans la mesure où les cellules ont été conçues pour pouvoir s'interconnecter par aboutement. L'avantage de l'utilisation de cellules prédéfinies réside dans le fait que les masques de ces cellules standard sont très denses et que l'on sait parfaitement dans quelles conditions ces cellules peuvent fonctionner. Cette approche présente cependant un certain nombre d'inconvénients :

- La conception d'une bibliothèque est une tâche très coûteuse. La rentabilité d'une telle opération dépend :

  * de la durée de vie de la bibliothèque. La bibliothèque ne risque-t-elle pas d'être obsolète dès qu'elle sera opérationnelle soit à la suite d'une évolution des règles de conception (augmentation de la densité d'intégration) soit à la suite de l'introduction d'une technologie plus performante (NMOS → CMOS).

  * de la souplesse d'utilisation de la bibliothèque. Pour que la conception d'une bibliothèque soit rentable, il faut d'une part, nous l'avons vu que la technologie qu'elle utilise soit disponible pendant une période suffisamment longue mais d'autre part que les cellules puissent être réutilisées pour une autre application que celles pour lesquelles elles ont été conçues.

L'utilisation d'une bibliothèque standard pour la compilation de silicium ne permet pas d'adapter les cellules de base aux contraintes spécifiées par l'utilisateur (surface - rapidité - consommation). Les cellules ont en effet été conçues pour fonctionner à une fréquence donnée et leurs tailles ont été optimisées pour cette utilisation. Si l'on veut fabriquer un circuit moins rapide, on se retrouve avec un circuit surdimensionné qui consomme plus que nécessaire. On ne peut évidemment obtenir un circuit fonctionnant à une fréquence plus élevée, le seul moyen à la disposition du concepteur de pallier à cet inconvénient, est de modifier sa description pour introduire plus de parallélisme au niveau architectural (cf chapitre 3 section 6).

5.2.2. Cellules paramétrées, cellules compilées.

C'est pour cette raison que l'on utilise de plus en plus de cellules paramétrées. On peut distinguer 2 types de cellules paramétrées selon la nature de la paramétrisation.

1) soit les cellules sont effectivement paramétrées. C'est-à-dire que pour chacune des cellules de base un dessin a été fait à la main et saisi à l'aide d'un éditeur graphique. A partir de ce dessin, la cellule a été paramétrée soit en introduisant des fractures (ou points d'étirement), soit en remplaçant certaines des dimensions des polygones ou des transistors de la cellule par des

paramètres.

2) soit les cellules sont générées automatiquement [LURS 84], [GAJS 86] à l'aide de structures cibles : réseau logique programmable ou logique aléatoire.

Dans ces 2 cas, que les cellules soient simplement paramétrées ou générées complètement automatiquement, les dimensions des transistors et des bus d'alimentation peuvent être paramétrées, ce qui permet d'adapter les cellules à l'environnement électrique dans lequel elles sont utilisées et d'assurer que le circuit pourra fonctionner à la fréquence spécifiée par l'utilisateur à condition qu'elle reste comprise dans un intervalle donné qui dépend de la technologie et de la longueur des mots de données traités par la partie opérative.

La compilation de cellules est certainement l'approche la plus générale. Pour qu'elle soit intéressante, il faut qu'un certain nombre de contraintes soient prises en compte :

<u>électriques</u>

- temps de propagation
- puissance consommée

<u>géométriques</u>

- position des entrées-sorties
- hauteur

Cette approche peut permettre de générer automatiquement les cellules de base d'une bibliothèque. Cela représente un gain de temps appréciable. Cependant les cellules générées automatiquement sont encore très peu denses. C'est pourquoi la plupart des compilateurs de silicium utilisent actuellement des cellules paramétrées, ce qui représente un compromis entre densité d'intégration et automatisation de la conception.

On peut résumer brièvement la situation :

- cellules prédéfinies : densité d'intégration maximum mais caractéristiques électriques fixes, cellules conçues à la main.
- cellules paramétrées : densité d'intégration acceptable, contrôle des caractéristiques électriques, cellules conçues à la main.
- cellules compilées : faible densité d'intégration mais automatisation complète de la conception et contrôle des caractéristiques électriques.

Dans la suite nous n'envisagerons que le cas des cellules prédéfinies et des cellules paramétrées.

5.3. Problèmes topologiques

Les problèmes topologiques dépendent du mode de génération des spécifications des masques. La génération des masques peut être basée sur une structure cible :

- SLAs [HILL 84],

- prédiffusé [GRAY 82],

ou sur l'utilisation d'une bibliothèque de cellules. Nous ne considèrerons par la suite que les problèmes liés à l'assemblage des cellules d'une bibliothèque. On peut distinguer 2 types d'approche selon que l'on adopte un modèle topologique ou non. L'assemblage des cellules d'une bibliothèque est beaucoup plus simple lorsque le plan de masse est prédéfini. Les problèmes de placement sont ainsi éliminés.

Un certain nombre de compilateurs de silicium utilisent un modèle topologique cible pour générer la partie opérative. Ce modèle est celui des parties opératives de circuits de type microprocesseur conçus ces dernières années : elles ont une structure dite "bit slice" (en tranches). La partie opérative est un bloc rectangulaire qui résulte de l'empilement de n tranches d'un bit. Cette structure très régulière présente 2 avantages:

- d'une part les masques obtenues sont très denses. Il n'y a pas de trou : toutes les cellules doivent avoir la même hauteur.
- d'autre part la génération automatique d'une partie opérative de ce type est grandement facilitée.

Dans certains microprocesseurs moins récents, les registres et les constantes étaient regroupés au sein de RAMs et de ROMs, ce qui permettait de réduire la surface occupée par les éléments de mémorisation aux dépens de la régularité du circuit car les différents blocs étaient de hauteur inégale. Cependant le gain de surface résultant d'une optimisation locale est en général moindre que le gain de surface résultant d'une optimisation globale et les parties opératives de type bit-slice sont de plus en plus courantes.

Nous avons supposé implicitement que nous adoptions une approche de type précaractérisée, à savoir que l'on disposait pour chacun des composants de la partie opérative d'une ou d'un ensemble de cellules à partir desquelles on obtient les masques par assemblage. Cette approche se distingue cependant de l'approche précaractérisée dans la mesure où le plan de masse est prédéfini. On restreint de ce fait la portée du générateur automatique de parties opératives, mais on peut ainsi complètement éviter les problèmes de placement. Les composants de la partie opérative sont simplement placés côte à côte.

D'autre part l'approche adoptée pour dessiner les cellules de base de la bibliothèque de base d'Apollon permet d'éviter en grande partie les problèmes de routage. Les cellules ont été dessinées pour que les connexions se fassent par aboutement. Lorsque 2 cellules sont assemblées, les connexions correspondant aux connecteurs en vis à vis sont tracées automatiquement. Les connexions sont réalisées à l'aide de bus qui sont situés au-dessus de la logique si bien qu'aucune place n'est perdue. On obtient ainsi une partie opérative très dense. Cette approche est cependant assez contraignante. En effet non seulement toutes les cellules doivent avoir la même hauteur mais aussi la même structure de bus. De plus le nombre de pistes que l'on peut utiliser pour tracer les bus est limité et ce mode d'interconnexion ne permet pas de résoudre tous les problèmes d'interconnexion.

Malgré ces contraintes, cette approche est de plus en plus employée car elle permet de minimiser la surface de la partie opérative puisqu'aucune place n'est perdue à cause des connexions [JOHA 79], [MEAD 80], [SHRO 82], [MARS 86], [DEMA 86].

Macptitts [SOUT 83] a une approche légèrement différente. Macpitts utilise une architecture bit-slice mais les connexions entre les différents composants sont situées dans des canaux. Il y a un canal de connexion au-dessus de chaque tranche si bien qu'une tranche de bit est en fait constituée de 2 parties :

- une tranche de cellules,
- un canal de routage.

Une certaine place est perdue car les connexions ne sont pas situées au-dessus de la logique. Mais en contrepartie cette approche permet d'implanter n'importe quel schéma d'interconnexion puisque le nombre de pistes disponibles pour les connexions n'est pas limité (cf 4.3.4).

La structure "bit slice" n'est cependant pas la panacée. En particulier elle est assez restrictive car elle nécessite que tous les composants aient la même hauteur. Il existe des applications pour lesquelles la structure bit slice n'est pas adaptée. Ce sont toutes les applications pour lesquelles les composants de la partie opérative n'ont pas la même hauteur.

Exemples :

- les données traitées n'ont pas le même format. Une solution consiste cependant à concevoir une partie opérative qui traite les données de format maximum et à l'utiliser pour traiter les mots de données de longueur moindre. Une autre solution consiste à réaliser plusieurs parties opératives de format différent.
- les circuits de traitement du signal qui doivent en général être très rapides nécessitent souvent une architecture systolique.

Silc [ROSE 85] procède par placement et routage pour générer la partie opérative, ce qui lui permet de traiter des données de formats différents. Les différents éléments de la partie opérative sont placés dans une rangée à l'aide d'un programme de placement unidimensionnel, basé sur l'ensemble des connexions de ces éléments. Les éléments sont connectés par aboutement dans la mesure du possible, sinon des canaux de routage horizontaux sont utilisés comme dans le cas de Macpitts. Silc ne semble pas utiliser un plan de masse prédéfini.

5.4. Problèmes électriques

5.4.1. Généralités

Parmi les problèmes électriques à prendre en compte, on peut citer [MEAD 80]:

1) l'électromigration.

Les atomes d'aluminium qui est un métal mou sont entraînés par les électrons. Si la densité de courant est trop élevée, la section du fil diminue et une coupure apparaît. Les bus d'alimentation doivent être dimensionnés suffisamment larges pour éviter que des coupures se forment. Afin de pouvoir dimensionner les bus d'alimentation, il faut connaître les besoins de chacune des cellules connectées aux bus.

Le problème se pose différemment si les bus d'alimentation sont parallèles aux bus de données [MARS 86], [JAMI 85] ou perpendiculaires [MATH 83], [SISK 82].

Dans le premier cas, la largeur des bus d'alimentation dépend de la largeur de la partie opérative, c'est à dire du nombre de ses éléments alors que dans le second cas la largeur des bus d'alimentation dépend du nombre de bits des données traitées.

2) La vitesse de transmission des signaux électriques

Le temps de réponse d'un inverseur est inversement proportionnel au rapport de la largeur à la longueur de ses transistors constitutifs (pour le temps de montée, il s'agit du rapport Wc/Lc du transistor de charge, alors que pour le temps de descente on prend en compte le rapport Wd/Ld du transistor de décharge). Il est aussi proportionnel à la capacité de charge à la sortie de l'inverseur. Les autres paramètres qui interviennent dans le calcul du temps de propagation à travers un inverseur ne dépendent que de la technologie utilisée. Par conséquent la démarche du concepteur est simple : il doit estimer la charge en sortie : la capacité de charge est obtenue en faisant la somme de la capacité de sortie propre à l'inverseur, de la capacité des fils de connexion et des capacités des grilles attaquées par la sortie. En effet les capacités rapportées en chaque noeud du circuit ne sont pas uniquement celles des grilles connectées à ce nœud. Il faut ajouter les capacités réparties entre les liaisons et la masse, ainsi que toutes les autres capacités de couplage. Ces capacités réparties ne sont pas négligeables. Si la capacité par unité de surface des grilles est d'un ordre de grandeur supérieure à celle des liaisons, les liaisons ont une superficie très largement supérieure à celle des grilles. Connaissant la capacité de sortie, le concepteur dimensionne son inverseur en fonction du temps de propagation désiré. Le problème du dimensionnement vient du fait que s'il est relativement aisé de dimensionner un inverseur, il devient impossible de dimensionner avec précision les transistors d'un circuit un peu plus complexe ; il faut recourir à un simulateur électrique.

D'autre part on ne peut jouer indéfiniment sur le dimensionnement des transistors pour diminuer le temps de propagation des transistors et ceci pour 2 raisons :

* lorsque la résistance équivalente des transistors d'un inverseur devient du même ordre de grandeur que la résistance attaquée, on arrive à la limite théorique du temps de propagation.

*(T≈(Rt+Rf)C, on aura forcément T>Rf.C) où T est le temps de propagation, Rt la résistance du transistor et Rf la résistance du fil de connexion, C la capacité de charge.

* d'autre part si l'on augmente la taille d'un inverseur, il faudra aussi augmenter la taille de l'inverseur qui l'attaque ; on a ainsi une réaction en chaîne. C'est pour cette raison que l'on utilise une suite d'inverseurs de plus en plus gros pour attaquer une charge importante.

Mais d'un autre côté, plus on augmente le nombre d'inverseurs en série, plus le temps de propagation est long puisque le temps de propagation du signal est proportionnel au nombre d'inverseurs qu'il doit traverser. On peut d'ailleurs calculer le temps minimum et le nombre optimum d'inverseurs qu'il faut pour attaquer une grosse charge capacitive à partir d'une capacité de départ donnée [MEAD 80].

3) la puissance consommée

Pour un inverseur NMOS, la puissance consommée est la somme de la puissance statique et dynamique

$P = 1/2\ IcVdd + C\rho(Vdd)^2 f$

f : représente la fréquence de fonctionnement
$C\rho$: la capacité de sortie globale
Ic : le courant débité par le transistor de charge.
Vdd = 5 Volts. On peut donc jouer sur la consommation statique car Ic est proportionnel au rapport (W/L)c. Pour avoir un courant statique faible et par conséquent une puissance statique faible, il faut réduire (W/L)c.

4) le niveau de sortie des signaux

Le niveau bas en sortie d'un inverseur pour un inverseur NMOS, c'est à dire la tension en sortie équivalente au zéro logique doit être nettement inférieure à la tension de seuil d'un transistor enrichi. (Pour le CMP [DELO 86], on donne un rapport de forme $\frac{Ws/Ls}{Wc/Lc} > 2{,}25$).

5.4.2. Problèmes électriques de la conception d'une partie opérative

En ce qui concerne la partie opérative, les problèmes de dimensionnement et de timing dépendent de la nature des actions opératives.

- transfert : dans notre cas (bus complémentés préchargés), il s'agit de déterminer la taille des transistors des registres et des préchargeurs et amplificateurs de bus ainsi que la longueur de chacune des phases du cycle opératif pour qu'un transfert puisse s'exécuter dans le temps imparti.
Si le temps de transfert minimum que l'on obtient est supérieur au temps imparti, il peut être nécessaire de reconsidérer le modèle architectural et électrique (par

exemple utilisation de connexions directes qui ne nécessitent aucune phase de précharge).

- opération : dans un opérateur de type UAL, le chemin critique est constitué par la chaîne de propagation de la retenue.
Si l'on désire faire des additions sur un nombre de bits élevé rapidement, il faut recourir à des additionneurs plus performants : chaîne de propagation à précharge et retenue anticipée.

- entrée-sortie : il faut dimensionner les amplificateurs en fonction de la longueur maximum des fils de connexion des points mémoire du tampon d'entrée sortie aux plots correspondants.

Les problèmes de dimensionnement cruciaux sont les suivants :

1) Le dimensionnement des bus d'alimentation :
Les bus d'alimentation sont reliés à un nombre important de transistors, et le courant qui circule dans un bus d'alimentation est la somme des courants qui traversent les transistors actifs reliés à ce bus. De ce fait l'intensité du courant circulant dans les bus d'alimentation est de loin supérieur au courant qui circule dans les autres conducteurs de la partie opérative.

2) Le dimensionement des amplificateurs des commandes :
Plus le nombre de bits de la partie opérative est élevé, plus le nombre de grilles attaquées par les commandes est important. De ce fait la capacité d'une ligne de commande est assez élevée et elle dépend du nombre de bits de la partie opérative. Il faut donc pouvoir dimensionner correctement les amplificateurs des commandes. De plus en NMOS mono-métal, on utilise souvent le polysilicium pour réaliser les lignes de commande; la résistance de ces fils est donc relativement élevée.
L'amplificateur de la bibliothèque NMOS que nous utilisons est constitué d'un push-pull attaqué par un inverseur de faible dimension. Les transistors du push-pull qui ont une grille de largeur importante n'ont pas de consommation statique.

3) Le dimensionnement des points mémoire et des préchargeurs et amplificateurs de bus.
Plus le nombre d'éléments reliés au bus est important, plus le bus est long et donc plus sa capacité est importante. Le morcellement des bus conduit bien sûr à une diminution de la capacité vue par les points mémoire connectés au même morceau. Mais si l'on veut effectuer un transfert entre 2 registres situés dans des sous parties opératives séparées par n-1 sous parties opératives, le signal devra traverser n portes de transfert. Le temps de propagation sera d'autant plus grand qu'il y a de portes de transfert à traverser en plus, car ces portes ont une résistance importante. Le nombre des segments du bus doit donc rester limité. D'autre part, le temps de propagation d'un signal le long d'une ligne constituée de n sections RC est de l'ordre

de $\frac{n(n+1)RC}{2}$.

Si on a n portes de transfert, le temps de propagation est proportionnel à n^2. Une des solutions pour diminuer le temps de propagation consiste à placer des amplificateurs espacés de façon régulière qui regénèrent le signal. L'utilisation de préchargeurs et d'amplificateurs différentiels dans le cas de bus complémentés et préchargés permet d'utiliser les mêmes points mémoire pour attaquer des bus de charge variable.

On rencontre un problème analogue pour la réalisation de la chaîne de propagation d'un additionneur. La longueur de la chaîne est proportionnelle au nombre de bits. Le chemin critique d'un additionneur de base comporte un tronçon constitué de portes de transfert en série. Pour augmenter la vitesse de propagation, on introduit des inverseurs par exemple tous les 4 portes pour régénérer le signal.

5.5. Paramétrisation des modules générateurs en fonction des contraintes électriques

5.5.1. Détermination des caractéristiques électriques du circuit

Pour pouvoir paramétrer les modules générateurs en fonction des contraintes électriques, il faut que l'on dispose d'une représentation électrique du circuit afin :

- de déterminer les besoins en courant des différentes cellules (dimensionnement des bus d'alimentation),
- de calculer la capacité et la résistance de chaque noeud (dimensionnement des transistors) et en particulier des lignes de commandes (dimensionement des amplificateurs des commandes).

Pour extraire les caractéristiques électriques en général il est souhaitable de disposer d'un outil hiérarchique. On doit d'abord extraire les paramètres électriques des cellules de base à l'aide d'un extracteur électrique [JERR 83]. La description électrique extraite automatiquement peut d'ailleurs être utilisée pour effectuer des simulations électriques.

La détermination des caractéristiques électriques après assemblage peut être faite de façon relativement simple.

On peut utiliser un assembleur du type Lubrick [SCHO 85] (cf chapitre 3), où on associe à chacun des connecteurs une capacité et une résistance (celles extraites automatiquement) et on additionne les capacités et les résistances lorsque l'on établit les connexions entre connecteurs se correspondant, les capacités étant en parallèle et les résistances en série.

Cette solution ne permet pas de calculer les caractéristiques électriques avec précision. En effet on a supposé dans le calcul précédent que les capacités et les résistances sont regroupées en un point alors qu'elles sont réparties sur toute la longueur des connexions. On peut montrer [WEST 85] qu'on obtient ainsi un pire cas. Une solution plus rigoureuse consiste à générer non pas directement les

caractéristiques électriques du circuit assemblé mais la liste des connexions tracées par l'assembleur. Ces informations peuvent alors être utilisées pour calculer avec précision les capacités et les résistances aux différents noeuds du circuit à partir des caractéristiques électriques des cellules de base.

On souhaite bien entendu déterminer les caractéristiques électriques sans devoir procéder à l'assemblage géométrique des cellules (calcul des gardes, tracé des connexions); c'est possible si les connexions sont de longeur nulle ou pratiquement nulle (connexion par aboutement ou connexion rectiligne entre 2 cellules voisines) ou si l'on se contente d'une valeur moyenne pour la longueur des connexions à l'intérieur des canaux de routage.

On a supposé implicitement d'une part que les portions des bus qui traversent les cellules font partie des cellules (et ne sont pas tracées automatiquement lors de l'assemblage) et d'autre part que les connexions se font par aboutement et ont une longueur pratiquement nulle puisque les connexions ne font intervenir qu'une cellule et sa voisine puisqu'il n'y a pas de transparence. Si l'on veut introduire des transparences, on complique le calcul des capacités et des résistances.

Si les longueurs des connexions ne sont pas négligeables (routage par dévoiement ou routage avec croisement des connexions), on peut adopter 2 stratégies pour le calcul des capacités et des résistances :

- soit on utilise des valeurs moyennes pour les longueurs des connexions basées sur les résultats d'une analyse statistique des longueurs des connexions dans des circuits représentatifs.
- soit on détermine la longueur exacte des connexions en assemblant les cellules pour les valeurs par défaut des paramètres. On peut d'ailleurs se contenter de calculer la longueur maximum d'une connexion : c'est la somme de la hauteur du canal de routage et la différence maximum des abscisses de 2 connecteurs se correspondant.

Remarque

Nous avons supposé implicitement que les cellules avaient déjà été dimensionnées pour calculer les capacités et les résistances aux différents noeuds du circuit. Les paramètres des cellules doivent avoir des valeurs par défaut que l'on affine à partir des caractéristiques électriques déterminées après assemblage de tout le circuit.

Une fois que les capacités et les résistances de tous les noeuds (ou équipotentielles) du circuit ont été calculées, la largeur des bus d'alimentation, la taille des amplificateurs des commandes et la taille des points mémoire et des préchargeurs et amplificateurs de bus peuvent être déterminées.

5.5.2. Mise en oeuvre de la paramétrisation

5.5.2.1. Paramétrage global des cellules

De nombeux auteurs préconisent la paramétrisation des cellules [JOHA 79], [SHRO 82], [SIX 86], [SISK 82], [MATH 83]. Le paramétrage permet :

- d'intégrer des contraintes géométriques : si l'on peut déformer les cellules, on peut dessiner les cellules de façon indépendante. Une fois les cellules dessinées et paramétrées, on les déforme simultanément pour que leurs points de connexion soient alignés.

Si l'on utilise une approche de type précaractérisée [JOHA 79], on doit dessiner des cellules qui ont toutes la même hauteur. Si l'on veut que la hauteur soit minimum, il faut donc commencer par la cellule qui a la hauteur maximum, c'est-à-dire celle qui est la plus difficile à compacter en hauteur. En pratique on se donne au départ la structure d'interconnexion, c'est-à-dire le nombre de bus, leur largeur, la distance entre 2 bus voisins ainsi que l'ordre des bus ce qui fixe la hauteur des cellules et on joue sur la largeur.

- d'intégrer les contraintes électriques : en particulier il faut pouvoir jouer sur la largeur des bus d'alimentation. Il est souhaitable aussi de pouvoir jouer sur la taille des transistors.

La paramétrisation des cellules ne peut pas être effectuée de façon simple en général. La modification des tailles des transistors est un problème difficile. En effet l'augmentation de la largeur ou de la longueur d'un seul rectangle peut nécessiter une refonte complète de toute la cellule si la cellule est au départ très dense.

Pour pouvoir réaliser cette modification automatiquement, il faut disposer d'un compacteur qui à partir d'un schéma squelettisé mou (non métrique - on donne cependant la taille des transistors) génère les masques de la cellule.

Mais comme on cherche à connecter les cellules par aboutement, il faut que le compacteur puisse traiter toutes les cellules d'une tranche à la fois de sorte que les connecteurs des cellules compactées qui se correspondent soient alignés et bien sûr que toutes les cellules aient la même hauteur.

L'alignement automatique des points de connexion de plusieurs cellules peut être effectué à l'aide d'un éditeur de masques symboliques (Caméléon [SIX 86]) qui permet de minimiser la taille d'une cellule après une phase de compactage basée sur un graphe de contraintes et un algorithme de plus long chemin.

Silc [ROSE 85] génère les spécifications des masques à l'aide de cellules paramétrées. Le langage de description utilisé SHAWL est procédural et permet de répercuter le changement de taille d'un rectangle, en effet les cellules sont décrites à l'aide de rectangles dont on donne le placement relatif. Les contraintes géométriques que doivent respecter les cellules de la partie opérative ne sont pas cependant aussi fortes que celles prises en compte par Caméléon. Les connexions à l'intérieur de la partie opértive ne sont pas réalisées systématiquement par

aboutement.

Matheson et al. [MATH 83] utilisent un langage de description hybride ic qui permet d'inclure des programmes en langage i (langage de description hiérarchique de masques) dans des programmes C pour paramétrer les cellules. Le concepteur n'a pas besoin de calculer la position exacte de chaque élément, car i permet de résoudre les contraintes géométriques.

5.5.2.2. Paramétrage local des cellules

Certains auteurs se contentent de dimensionner uniquement les bus d'alimentation.

Ainsi le système Bristle Blocks utilise des cellules extensibles [JOHA 79]. De même le Datapath Generator de Shrobe [SHRO 82] et Macpitts [SISK 82] utilisent des lignes d'étirement (lignes de "fracture") ou points d'étirement qui permettent entre autres de dimensionner correctement les bus d'alimentation. Le principe est simple. Les rectangles qui sont coupés par la ligne d'étirement sont étirés dans le sens perpendiculaire à la ligne de la valeur déterminée par les besoins en puissance des différents éléments d'une tranche de la partie opérative. Pour que cette procédure triviale puisse être appliquée, il est nécessaire qu'il n'y ait pas de grilles de transistors en dessous des bus d'alimentation. Datapath [MARS 86] permet à la fois de dimensionner les bus d'alimentation et les amplificateurs des lignes de commande. Le langage de description procédural de circuits intégrés ICPL est utilisé à cet effet. D'autre part des amplificateurs de bus sont placés à intervalle fixe le long des bus.

5.5.2.3. Cellules prédéfinies

Nous proposons une méthode moins générale mais plus simple basée sur l'utilisation de cellules prédéfinies. Au lieu de paramétrer les cellules, on met au point plusieurs versions des cellules dont le dimensionnement est critique : amplificateurs des commandes, points mémoire et préchargeurs et amplificateurs des bus. De plus on conçoit des additionneurs de type différent selon que l'on veut faire une addition 16 bits ou 32 bits ou davantage. Au lieu d'adapter automatiquement la taille d'une cellule aux besoins électriques, on se contentera de choisir la version de la cellule dont les caractéristiques électriques sont les plus proches tout en restant compatibles avec les caractéristiques requises. Un problème demeure cependant, c'est le dimensionnement des bus d'alimentation. Les bus d'alimentation des cellules de la bibliothèque d'Apollon sont situés au-dessus des transistors. En particulier le bus de masse est situé au milieu des cellules, et il y a de nombreux transistors à ce niveau. La méthode d'extension triviale utilisée par Shrobe ne peut être appliquée. Pour cela il faudrait prévoir dès le départ de ne pas faire passer de transistors sous les bus d'alimentation, ce qui impliquerait bien sûr une perte de place. Une autre solution plus facile à mettre en oeuvre consiste à surdimensionner les bus d'alimentation dès la conception des cellules. Cette

approche ne permet pas d'obtenir des masques aussi denses mais évite au concepteur de dessiner plusieurs fois la même cellule seulement pour redimensionner les bus d'alimentation. Il existe une autre solution : on limite la longueur des peignes d'alimentation. Ainsi on peut garder la largeur des bus d'alimentation constante puisque si l'on dépasse une certaine consommation, on utilise de nouveaux peignes d'alimentation.

Dans le cas d'Apollon, on ne garantit le bon fonctionnement électrique, que si la partie opérative ne dépasse pas une certaine largeur, ou ne contient pas plus d'un certain nombre d'éléments (registres, opérateurs). On fait l'hypothèse simplificatrice que la consommation est proportionnelle à la largeur de la partie opérative.

5.6. Les outils de vérification des masques

Nous avons étudié jusqu'ici les problèmes géométriques et électriques de la conception des cellules en ce qui concerne l'aspect génération. On a vu que l'on a besoin d'un certain nombre d'outils pour la génération automatique.

- éditeur de masques
- extracteur électrique
- assembleur multi-niveau (géométrique et électrique)
- compacteur avec alignement automatique des connexions (pour l'approche cellules paramétrées)

Nous avons supposé jusqu'ici que la partie opérative est correcte par construction. Cependant il est clair que dans le cadre de la compilation de circuits intégrés, on ne peut se tenir à cette hypothèse et qu'il faut s'entourer du maximum de précautions.

Il est donc nécessaire de disposer (surtout si l'on utilise une approche qui n'est pas complètement automatisée, c'est-à-dire l'approche cellules prédéfinies et aussi dans une moindre mesure l'approche cellules paramétrées) des outils de vérification suivants :

- d'un simulateur électrique : on ne cherchera à faire des simulations électriques fines que pour les cellules de base. La simulation électrique fine d'un gros circuit n'étant pas possible actuellement.

- d'un vérificateur de règles de dessin : la vérification est faite au niveau de chaque cellule. On n'a pas besoin de faire une vérification globale du circuit (on aurait intérêt alors à faire une vérification hiérarchique) car on suppose que l'assemblage est correcte par construction. Lubrick calcule automatiquement la garde entre 2 cellules.

- d'un analyseur temporel : un analyseur temporel [OUST 83] est nécessaire car on ne peut pas se servir d'un simulateur électrique pour simuler de gros circuits, une simulation électrique étant très coûteuse en temps CPU. Une autre possibilité

serait d'utiliser un simulateur logique. Au lieu de résoudre un système d'équations différentielles, le simulateur logique associe un délai à chaque composant de base. Selon le simulateur, la prise en compte des temps de propagation est plus ou moins fine. Le gros inconvénient de la simulation logique vient du fait que la vérification du bon fonctionnement est incomplète car elle ne peut être effectuée exhaustivement, le nombre des combinaisons en entrée à tester pouvant être très élevé. C'est pourquoi de nombreux analyseurs temporels ont été développés. Un analyseur temporel hiérarchique permet de déterminer l'avance ou le retard des signaux aux différents points d'un circuit en utilisant la décomposition hiérarchique du circuit. Seuls les chemins logiques à l'intérieur des blocs du circuit doivent être tracés. Les temps de propagation entre 2 points du circuit sont calculés à partir des temps de propagation à travers les blocs constitutifs et des temps de propagation le long des connexions entre ces blocs. Selon le niveau de conception, les temps de propagation le long des connexions peuvent être calculés de façon plus ou moins fine.

Il est bien entendu qu'avant d'utiliser des outils de vérification hiérarchique, il faut que la vérification des cellules élémentaires ait déjà été effectuée.

De Man [DEMA 86] préconise d'utiliser 2 autres types d'outils :

- un vérificateur de règles de conception de circuits : Cathédral2 permet de vérifier si le circuit conçu appartient à la classe des configurations de circuits valides et si le circuit est synchronisé de façon adéquate.
- un vérificateur de règles électriques : les configurations illégales (coupures, etc...) sont détectées. Pour les circuits acceptables, le système vérifie que les niveaux logiques sont correctes.

Cette approche permet de réduire le temps de simulation électrique dans la mesure où le nombre des configurations des entrées pour lesquelles on effectue des simulations peut être diminué.

6. Interface entre la partie opérative et son environnement

6.1. Transferts de données entre la partie opérative d'un microprocesseur et une mémoire externe

6.1.1. Introduction

On se limite ici à l'étude des échanges d'information entre la partie opérative et une mémoire externe. Le compilateur SYCO ne génère qu'une unité centrale composée d'une partie opérative et d'une partie contrôle. Les problèmes d'échange d'information avec une mémoire interne sont différents:

1) dans le cas d'une mémoire externe, les données transitent par des plots, qui sont les ports d'entrée-sortie du microprocesseur et assurent un ajustement électrique entre les signaux internes et externes.

2) le protocole de communication peut être synchrone ou asynchrone. Mais il semble clair que l'on choisira plutôt un protocole synchrone dans le cas d'une mémoire interne et un protocole asynchrone dans l'autre cas. Par exemple si l'on désire réaliser un microprocesseur réduit à une unité centrale, pouvant s'interfacer avec n'importe quelle mémoire, on doit prévoir un mécanisme de lecture-écriture asynchrone. Si l'on dispose d'une mémoire interne, il est beaucoup plus facile de synchroniser les échanges d'information entre mémoire et partie opérative.

6.1.2. Primitives matérielles nécessaires à l'acheminement des données vers une mémoire externe

Pour transférer le contenu d'un registre vers l'extérieur ou vice-versa, on doit introduire entre un registre et un bus externe, un tampon relié à des plots d'entrée-sortie.
Ce tampon peut être placé dans la partie opérative ou être incorporé dans les plots d'entrée-sortie (plots avec mémorisation). Un tampon doit être utilisé pour garder disponible suffisamment longtemps l'information échangée et permettre ainsi de synchroniser le microprocesseur avec la mémoire.

Si l'on dispose d'une mémoire interne, un tampon est utilisé pour stocker le mot lu ou écrit; on peut placer ce tampon soit au niveau de la mémoire, soit au niveau de la partie opérative.

Par ailleurs, afin d'économiser de la place, on établit une correspondance biunivoque entre les plots et leur tampon correspondant. L'utilisation de

multiplexeurs coûteux en surface entre les plots et le tampon est prohibée. Si l'on dispose de 16 plots pour une partie opérative de 32 bits, on transfèrera les bits de poids faibles et de poids forts dans le même tampon 16 bits.

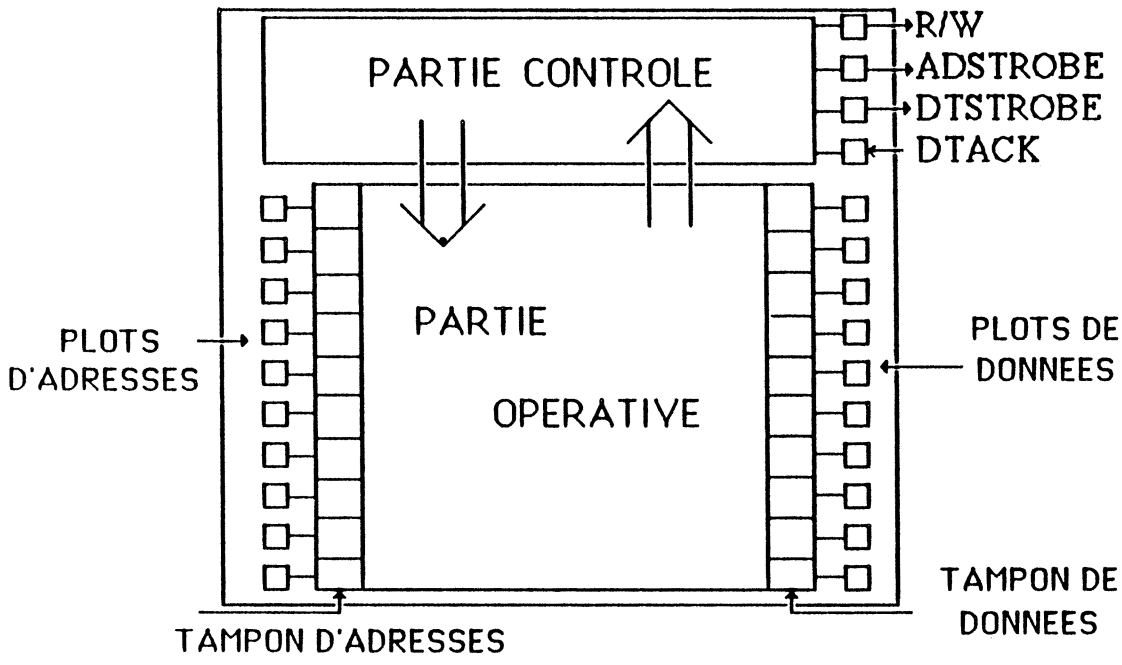


figure 2.20 : plots et tampons d'entrée-sortie

6.1.2.1. Ecriture en mémoire

Plusieurs commandes doivent être activées pour envoyer le contenu d'un registre sur les plots; dans le cas d'un plot sans mémorisation du type de celui de la figure 2.21, il faut 4 commandes :

1) établir la connexion entre le registre et un des bus de la partie opérative,
2) établir la connexion entre ce bus et le tampon de sortie,
3) établir la connexion entre le tampon de sortie et les plots correspondants,
4) mettre les plots à l'état passant.

Lorsque la valeur présente sur le plot a été lue, le plot doit être remis à l'état haute impédance.

En fait les commandes 3 et 4 peuvent être remplacées par une seule commande : il suffit de relier la commande de connexion du tampon de sortie aux plots et de s'en servir en même temps pour mettre les plots à l'état passant. Il sera sans doute nécessaire de réamplifier la commande de connexion de sortie, car elle devra encore attaquer des grilles à chaque plot. Il faudra donc trois commandes pour envoyer une donnée ou une adresse sur les plots :

1) établir la connexion entre le registre et un des bus de la partie opérative,
2) établir la connexion entre ce bus et le tampon de sortie,
3) établir la connexion entre le tampon de sortie et les plots correspondants et

mettre les plots à l'état passant.
Dans la figure 2.21, seuls un bit du tampon d'entrée-sortie et un plot ont été représentés.

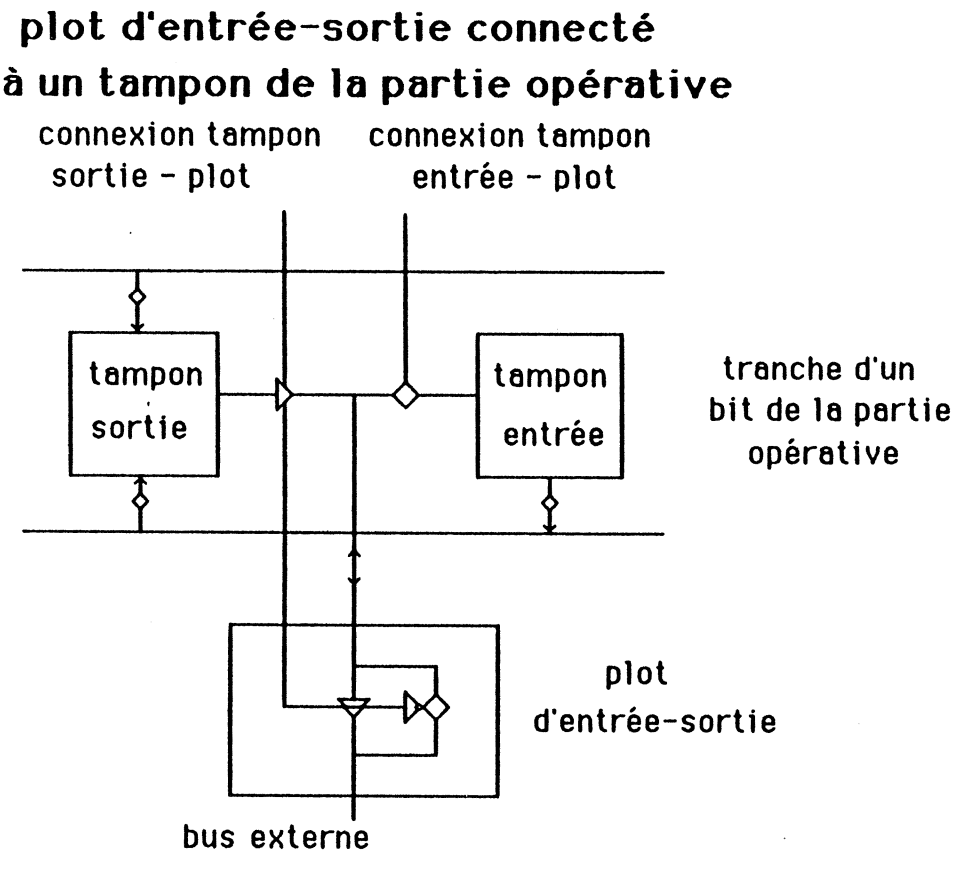


figure 2.21

6.1.2.2. Lecture de la mémoire

Pour lire un mot mémoire, il faut activer quatre commandes; il faut:
1) mettre le plot à l'état haute impédance et par la même connecter le plot à la ligne reliant le plot au tampon,
2) valider la connexion entre cette ligne et le tampon d'entrée,
3) valider la connexion entre le tampon d'entrée et un bus de la partie opérative,
4) et finalement valider la connexion entre ce bus et le registre destinataire.

6.1.3. Description de protocoles de communication à l'aide de LDS

Toute la difficulté de l'écriture d'un protocole de communication réside dans le fait qu'il existe un organe fonctionnel (tampon plus plots) entre le registre et la mémoire externe qui n'est pas décrit au niveau de LDS (cf chapitre I section2.3). Le paragraphe précédent a permis de préciser les commandes que l'on doit activer pour

acheminer des données entre un registre de la partie opérative et une mémoire externe. LDS doit permettre à l'utilisateur de décrire le séquencement de ces commandes. Pour ne pas introduire de tampon dans la description LDS, on est obligé d'imposer au compilateur de parties opératives un certain nombre de contraintes (voir chapitre III).

6.1.3.1. Exemple de protocole asynchrone

Deux procédures d'entrée-sortie asynchrones sont présentées dans l'exemple suivant:

```
SMODULE MICROP (adbus, dtbus, dtack,adstrobe, dtstrobe, read_write);

SIGNAL;
 adbus 0:8, OUT; ?bus d'adresses
 dtbus 0:8, INOUT; ?bus de données
 dtack, IN, CTRL; ?data acknowledgement
 adstrobe, OUT, CTRL; ?address strobe
 dtstrobe, OUT, CTRL; ?data strobe
 read_write, OUT, CTRL; ?sélection lecture=1 / écriture=0
END;

REGISTER;
 a 0:8; ?accumulateur
 ad 0:8; ?registre d'adresses
 ir 0:8; ?registre instruction
 pc 0:8; ?compteur ordinal
END;

LINK MICROP;

END;

CMODULE sta;
<e1> (adbus := ad; RESET read_write; dtbus := a;)
     (adbus := ad; SET adstrobe; dtbus := a; SET dtstrobe;)
<e2> IF (dtack = 0) (adbus := ad; dtbus := a; NEXT e2;) END;
     (RESET adstrobe; RESET dtstrobe;)
END;

CMODULE fetch;
<send> (adbus := pc; SET read_write;)
     (adbus := pc; SET adstrobe;)
<receive> IF (dtack = 0) (adbus := pc; NEXT receive;) END;
```

```
    (ir := dtbus; pc := pc + 1; RESET adstrobe;)
END;
```

Le programme LDS ci-dessus est constitué de deux parties:
1) d'une part, d'une partie déclaration de ressources, incorporée dans un module structurel "smodule",
2) et d'autre part d'un ensemble de modules comportementaux "cmodules" inclus dans un Cmodule global. Le cmodule "sta" décrit le séquencement des actions opératives et des actions de contrôle nécessaires à l'écriture du contenu d'un registre en mémoire externe. Le cmodule fetch est une procédure de lecture d'un mot-mémoire. L'utilisateur doit déclarer les ports d'entrée-sortie du système; ces ports peuvent être de deux types:
1) données : ce sont les bus de données et d'adresses du microprocesseur; l'utilisateur ne déclare pas les tampons d'entrée-sortie et les plots, c'est le compilateur qui gère les transferts entre les registres de la partie opérative et la mémoire externe.
2) contrôle : ce sont les signaux de contrôle et de synchronisation; pour échanger des données avec la mémoire, plusieurs signaux sont nécessaires:
- un signal read_write indiquant à la mémoire que l'on désire effectuer une lecture ou une écriture.
- un signal adstrobe indiquant à la mémoire qu'une adresse est disponible sur le bus d'adresses.
- un signal dtstrobe indiquant à la mémoire que des données sont disponibles sur le bus de données.
- un signal dtack indiquant au microprocesseur que des données sont disponibles sur le bus de données (lecture d'un mot-mémoire) ou que les données envoyées par le microprocesseur ont été écrites en mémoire.

6.1.3.2. Exemple de protocole synchrone

```
SMODULE MICROP (adbus, dtbus, adstrobe, read_write);

SIGNAL;
 adbus 0:8, OUT; ?bus d'adresses
 dtbus 0:8, INOUT; ?bus de données
 adstrobe, OUT, CTRL; ?address strobe
 read_write, OUT, CTRL; ?sélection lecture=1 / écriture=0
END;

REGISTER;
 a 0:8; ?accumulateur
 ad 0:8; ?registre d'adresses
 ir 0:8; ?registre instruction
 pc 0:8; ?compteur ordinal
```

```
END;

LINK MICROP;

END;

CMODULE sta;
<c1> (adbus := ad; RESET read_write; dtbus := a;)
     (adbus := ad; SET adstrobe; dtbus := a;)
     (adbus := ad; dtbus := a;)
     RESET adstrobe;
END;

CMODULE fetch;
<c1> (adbus := pc; SET read_write;)
     (adbus := pc; SET adstrobe;)
     adbus := pc;
     (ir := dtbus; pc := pc + 1; RESET adstrobe;)
END;
```

La description de transferts synchrones est plus simple que la description de transferts asynchrones. Seuls deux signaux doivent être envoyés à la mémoire externe:
1) read_write pour choisir le type de l'opération (lecture ou écriture),
2) et adstrobe pour démarrer l'opération.
Le microprocesseur n'est pas notifié du déroulement de l'opération car il s'agit d'une opération synchrone.

6.1.4. Fonctionnement détaillé du microprocesseur pendant une lecture ou une écriture en mémoire

6.1.4.1. Exécution d'actions d'entrée-sortie au niveau de la partie opérative

Les actions d'entrée-sortie sont de la forme:
registre := bus_externe;
ou
bus_externe := registre;
Pour passer d'un registre au bus_externe, l'information doit transiter par un tampon relié à un ensemble de plots, qui sont eux mêmes connectés au bus externe. Le tampon est placé dans la partie opérative alors que les plots sont situés sur le pourtour de la partie opérative. Le plot est commandé par un seul signal de commande : mise à l'état haute impédance. Comme ce signal est aussi le signal de connexion en sortie du tampon de sortie, les commandes des plots peuvent être

facilement gérées au niveau de la partie opérative.

Opération de sortie de la partie opérative

On désire effectuer l'opération suivante:
bus_externe := registre;
Une action de la partie opérative se déroule sur deux cycles (précharge - lecture - amplification - écriture). Logiquement l'action de sortie devrait se dérouler sur deux cycles de la façon suivante:
<cycle1> tampon_bus_externe := registre;
<cycle2> bus_externe := tampon_bus_externe;
Pendant le premier cycle, le contenu du registre est transféré dans le tampon de sortie de "bus_externe"; pendant le deuxième cycle, le contenu du tampon est transféré jusqu'aux plots.
Ce séquencement sur deux cycles n'est satisfaisant que si l'on ne désire pas maintenir le contenu du tampon sur les plots pendant plus d'un cycle opératif. Si pour des problèmes de synchronisation, on doit prolonger l'opération de sortie pendant plusieurs instructions de 2 cycles, on doit modifier ce séquencement sinon les données ne seraient disponibles qu'un cycle sur deux. Le séquencement suivant permet de résoudre ce problème:
<cyle1> (tampon_bus_externe := registre; bus_externe := tampon_bus_externe;)
<cyle2> bus_externe := tampon_bus_externe;
Bien sûr, pendant le premier cycle de la première occurrence de cette action de sortie, le contenu du registre ne sera pas disponible sur les plots; mais il sera disponible lors des occurrences suivantes.

Comme on vient de le voir, il est parfaitement possible de laisser le compilateur gérer lui-même le tampon d'entrée-sortie. Cela a le grand avantage de simplifier la vie de l'utilisateur.
Cependant comme l'utilisateur ne peut pas accéder au tampon, il ne peut en aucun cas s'en servir comme registre supplémentaire. En outre un bus de la partie opérative sera utilisé pour réaliser tous les premiers cycles le transfert :
tampon_bus_externe := registre;
qui n'a de raison d'être que pendant la première occurrence de l'action de sortie.

Opération d'entrée dans la partie opérative

On désire réaliser le transfert suivant:
registre := bus_externe;
On effectuera cette opération en deux cycles:
<cycle1> tampon_bus_externe := bus_externe;
<cycle2> registre := tampon_bus_externe;
Ce séquencement est adéquat car une telle action n'a besoin d'être réalisée qu'une seule fois.

6.1.4.2. Exécution d'opérations d'entrée-sortie au niveau de la partie contrôle

Fonctionnement détaillé de la procédure fetch asynchrone:

```
CMODULE fetch;
<send> (adbus := pc; SET read_write;)
    (adbus := pc; SET adstrobe;)
<receive> IF (dtack = 0) (adbus := pc; NEXT receive;) END;
    (ir := dtbus; pc := pc + 1; RESET adstrobe;)
END;
```

Première instruction :
Le contenu du compteur ordinal "pc" est disponible sur les plots associés au bus d'adresses "adbus" à la fin du deuxième cycle; au cours du premier cycle, read_write est positionné à 1; c'est à dire que la mémoire est mise en mode lecture.
Deuxième instruction :
Au cours du premier cycle, adstrobe est positionné à 1; la mémoire sait alors qu'il y a une adresse valide sur le bus d'adresses. Le contenu de "pc" est disponible sur les plots d'adresses pendant les deux cycles.
Troisième instruction :
Tant que "dtack" est nul, c'est à dire tant que la mémoire n'a pas notifié au microprocesseur que des données sont valides sur le bus de données, le microprocesseur attend tout en maintenant valide l'adresse sur le bus d'adresses.
Quatrième instruction :
Le registre d'instruction est chargé en fin du deuxième cycle; le compteur ordinal est incrémenté; "adstrobe" est positionné à 0 au cours du premier cycle.

6.2. Problèmes d'interface avec la partie contrôle : la récupération des comptes rendus de la partie opérative

6.2.1. Présentation du problème

Un certain nombre de tests (comparaison - test de la retenue - etc...) sont effectués par la partie opérative, et les résultats de ces tests sont exploités par la partie contrôle afin d'assurer le bon déroulement des instructions.

La récupération de ces résultats par la partie contrôle n'est cependant pas chose aisée et ceci pour plusieurs raisons.

6.2.1.1. Raisons liées au langage de description comportementale du circuit

Les indicateurs arithmétiques sont des sous-produits des opérations arithmétiques et logiques. La plupart des langages de description de circuit ne prennent pas en compte ce phénomène si bien que le concepteur ne peut récupérer que le résultat d'une opération et qu'il doit faire recalculer séparément les différents indicateurs arithmétiques pour pouvoir les utiliser.

Donnons un exemple :

$S_l := A_l + B_l$;

if carry $S_h := A_h + B_h + 1$; else $S_h := A_h + B_h$; end;

Le langage de description doit permettre au concepteur de décrire cette instruction conditionnelle sans qu'il doive spécifier que la retenue a été calculée lors de l'opération précédente. Un certain nombre de mots clés doivent donc être introduits dans le langage d'entrée du compilateur. Par exemple : carry, negative, overflow et zero pour les indicateurs de retenue, signe, débordement et de comparaison à zéro respectivement.

Ces mots clés doivent être reconnus par le compilateur bien sûr mais aussi par le simulateur fonctionnel de façon à ce que l'on puisse effectuer une simulation complète du circuit.

6.2.1.2. Raisons liées au modèle architectural

Récupération des indicateurs arithmétiques

Puisque le compilateur permet de décrire des actions opératives en parallèle, il est possible que la partie opérative possède plusieurs unités de traitement. Comme au départ le concepteur ne sait pas comment vont s'effectuer les différentes opérations, puisqu'il se contente de donner une description comportementale, il ne saura certainement pas comment récupérer les indicateurs arithmétiques s'il ne spécifie pas l'endroit où ils doivent être stockés.

Distinction entre registre d'état et indicateurs arithmétiques

La plupart des circuits complexes possèdent un registre d'état dont le rôle est de sauvegarder un certain nombre d'informations concernant l'état de la machine. Ces informations peuvent être utilisées lors de l'exécution d'une nouvelle instruction ou à la suite de l'exécution d'un sous-programme ou d'une interruption. S'il existe plusieurs unités de traitement, il devient nécessaire de dissocier le registre d'état du tampon de l'unité de traitement où sont stockés les indicateurs arithmétiques. En effet, s'il n'y a qu'une unité de traitement, rien ne s'oppose à ce que le tampon de l'unité de traitement contenant les indicateurs arithmétiques fasse partie du registre d'état. S'il y a plusieurs opérateurs, il faut donc pouvoir transférer les indicateurs arithmétiques dans le registre d'état. Afin de réduire le nombre de cycles nécessaires à ce transfert, il est souhaitable d'une part de pouvoir transférer ces indicateurs simultanément et d'autre part de pouvoir les transférer en même temps que le résultat de l'opération c'est-à-dire dès qu'ils ont été modifiés.

6.2.2. Solutions proposées :

On peut résoudre les problèmes précédents de 2 façons différentes. En effet soit on transfert directement les indicateurs arithmétiques vers la partie contrôle : on mémorise les indicateurs arithmétiques dans des tampons qui sont solidaires des opérateurs (UALs par exemple) et qui sont directement accessibles par la partie contrôle, soit on transfert d'abord les indicateurs arithmétiques dans des registres spécifiés par l'utilisateur, qui eux seulement sont accessibles par la partie contrôle. Nous allons voir que cette dernière solution est de loin la plus élégante.

6.2.2.1. Mémorisation des indicateurs arithmétiques dans des tampons solidaires des opérateurs

Cette solution semble à première vue être la plus logique. En effet les indicateurs arithmétiques sont calculés au niveau des opérateurs. Il est intéressant de mémoriser ces indicateurs là où ils sont calculés, car alors il n'y a pas besoin d'utiliser les bus globaux de la partie opérative pour effectuer le transfert des indicateurs arithmétiques dans leur registre de mémorisation.

Si la destination des indicateurs arithmétiques est solidaire de l'opérateur où ils ont été calculés, il devient nécessaire de donner au concepteur les moyens de lier la consultation de ces indicateurs à l'opérateur où ils ont été calculés ou à l'opération qui les a positionnés.

Introduisons donc la fonction :

BOP(A, B, +, C, 1)

où BOP est l'abréviation de BOîte à OPérations.

Cette fonction permet de spécifier :

1°) que l'on veut exécuter l'opération A:=B+C,

2°) que l'opération est indicée et que son indice est 1.

La décomposition d'une addition 2n bits en 2 opérations n bits donnée en exemple plus haut devient alors :

BOP(Sl, Al, +, Bl, 1) ;
if carry (1) Sh:=Ah+Bh+1 ; else Sh:=Ah+Bh ; end ;

L'indice permet donc de rattacher les indicateurs arithmétiques aux opérations qui les ont positionnés. Cet indice est nécessaire car une même opération peut être exécutée de façon différente si elle doit s'exécuter en parallèle avec des instructions différentes du fait que l'architecture de la partie opérative doit satisfaire aux contraintes du modèle à 2 bus (cf chapitre 3 section 3.1.2.3). Autrement on pourrait se contenter de récrire l'exemple précédent de la façon suivante :

Sρ:=Aρ+Bρ
if carry (Sρ:=Aρ+Bρ) Sh:=Ah+Bh+1 ; else Sh:=Ah+Bh ; end ;

Si l'on désire simplifier la spécification de la fonction carry comme suit :

Sρ:=Aρ+Bρ ;
if carry (Aρ+Bρ) Sh:=Ah+Bh+1 ; else Sh:=Ah+Bh ; end ;

il est nécessaire de supposer en outre que le choix de l'opérateur (dans le cas où il y a plusieurs additionneurs) dépend uniquement des opérandes, ce qui n'est bien sûr pas le cas en général. On peut donc conclure que si l'on veut omettre l'indice de liaison il faut implanter les instructions en vérifiant les contraintes précédentes. Dans le cas où le modèle ne le permet pas, il est alors nécessaire de modifier le séquencement et d'augmenter le nombre de cycles nécessaires à l'exécution de l'instruction.

Donc si l'on ne tient pas compte de ces contraintes d'implantation des instructions, il est nécessaire d'utiliser un paramètre de liaison. Ce paramètre peut être utilisé de plusieurs façons :

- on peut se contenter d'indicer les opérations : cette solution présente les 2 inconvénients suivants :

  * le concepteur doit tester la valeur des indicateurs arithmétiques avant d'effectuer d'autres opérations. En effet, si d'autres opérations sont effectuées entre temps, elles peuvent venir modifier la valeur des indicateurs arithmétiques puisque le concepteur ne peut pas spécifier où doivent s'effectuer les opérations.
  * d'autre part même si l'on n'effectue jamais de tests en parallèle, il se peut que les tests spécifiés par le concepteur ne soient pas tous exécutés par le même opérateur non pas parce que les tests nécessitent des opérateurs différents mais parce qu'il se peut que des opérateurs soient dupliqués :
    . soit parce que le modèle à 2 bus est trop contraignant,
    . soit parce que 2 opérations identiques doivent être exécutées en

parallèle. Le nombre des fils de compte rendu augmente alors inutilement.

- une deuxième façon de procéder est d'utiliser le paramètre de liaison pour lier la consultation des indicateurs arithmétiques à l'opération où ils ont été calculés.

On spécifie que 2 opérations se font sur le même opérateur en donnant la même valeur aux paramètres de liaison des 2 appels de la fonction BOP. On pallie ainsi aux inconvénients mentionnés plus haut, mais on augmente les contraintes imposées au générateur d'architecture de la partie opérative tout en forçant le concepteur à prendre lui-même les décisions quant à l'allocation des opérateurs.

L'introduction de la fonction BOP permet de résoudre le problème de récupération des comptes-rendus.

Si l'on veut mémoriser les indicateurs arithmétiques dans le registre d'état, il est nécessaire d'ajouter un autre paramètre à la fonction BOP, le nom du registre d'état. L'ajout de ce paramètre permet de spécifier que le transfert des indicateurs arithmétiques doit être effectué immédiatement après leur calcul.
exemple BOP (A, B, +, C, RE,)
Les indicateurs arithmétiques sont transférés dans le registre d'état RE en même temps que le résultat de l'addition dans A. On remarque que dans ce cas, il est inutile de recourir au paramètre de liaison.

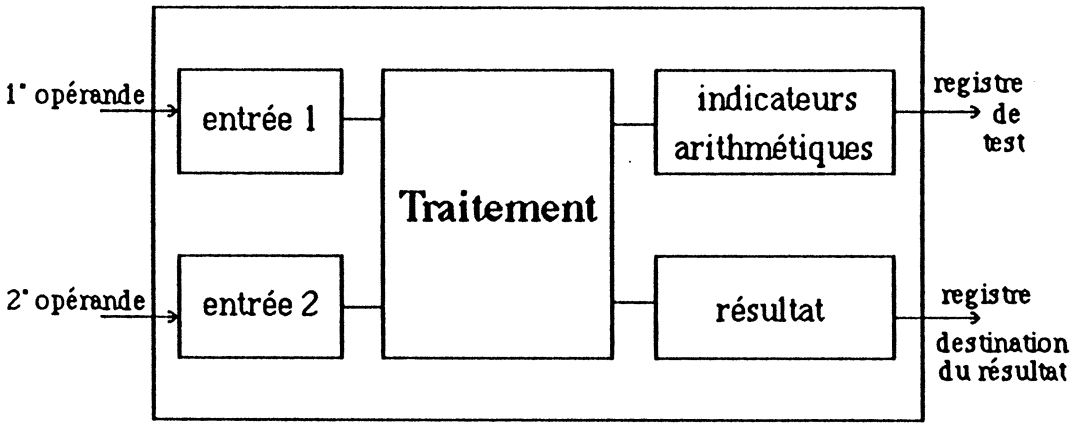
Ceci nous amène à la seconde solution pour la réalisation de l'interface partie opérative - partie contrôle.

6.2.2.2. Dissociation des éléments de mémorisation des indicateurs arithmétiques des opérateurs qui les ont positionnés

Si l'on considère que l'on peut mémoriser les comptes-rendus dans un registre quelconque, la récupération des résultats ne pose plus aucun problème. Il suffit de spécifier le registre destination des indicateurs arithmétiques comme on spécifie le registre destination du résultat de l'opération. On utilise la fonction BOP de la façon suivante :

BOP (A, B, +, C, RE)

pour signifier que l'on effectue l'opération B+C, que l'on transfère le résultat dans A et les indicateurs arithmétiques dans RE.



Unité de traitement généralisée à 2 entrées et 2 sorties

figure 2.22

Cette solution présente les avantages suivants :

- rien n'oblige le concepteur à tester la valeur des indicateurs arithmétiques immédiatement après leur calcul.
- le concepteur maîtrise le nombre de fils de compte rendu; il en existe autant que le concepteur en déclare.
- elle n'oblige pas le concepteur à effectuer une préallocation des opérateurs.

Le seul inconvénient est bien sûr que 2 bus au lieu d'un seront utilisés au second cycle d'une opération avec mémorisation des comptes rendus.

Une solution analogue est adoptée dans [MARS 86] (Datapath : a cmos datapath silicon assembler). La partie opérative est assemblée automatiquement à partir de 3 composants de base :

- des bancs de registre
- des unités de traitement
- des registres d'état ou de mémorisation des comptes rendus.

Le registre d'état ne fait pas partie des opérateurs et ceci pour plusieurs raisons d'après [MARS 86] :

- plus d'un opérateur et les registres peuvent ainsi accéder aux indicateurs arithmétiques, ce qui facilite l'exécution en pipeline.
- de plus c'est une bonne solution topologique qui cadre bien avec l'architecture en piles linéaires utilisées par DATAPATH.

Pour pouvoir tester la valeur des indicateurs arithmétiques, le concepteur doit savoir quel est le bit du registre de mémorisation des comptes-rendus qui correspond à tel indicateur arithmétique et ceci pour chaque indicateur

arithmétique.

Soit c'est le concepteur qui établit cette correspondance, soit c'est le compilateur qui fixe cette correspondance et le concepteur doit consulter le tableau de correspondance dans le manuel d'utilisation du compilateur.

Le fait de permettre au concepteur d'établir lui-même la correspondance entre indicateurs arithmétiques et bits du registre de mémorisation présente l'avantage suivant : les indicateurs arithmétiques de 2 opérations en parallèle peuvent être placés les uns au dessous des autres dans un même registre, ce qui permet une meilleure utilisation de la surface de silicium.

Les registres d'indicateurs arithmétiques sont souvent à l'origine des trous que l'on peut observer dans les parties opératives tout à fait régulières par ailleurs. Le nombre de bits du registre d'état et le nombre d'indicateurs arithmétiques sont en général inférieurs au nombre de bits des mots traités par la partie opérative.

Cependant, la sortie d'un indicateur arithmétique et le bit du registre destination qui lui correspond ne sont pas forcément situés dans la même tranche de bit. La mise en place d'une commande et d'un chemin d'accès propre à chaque indicateur arithmétique se traduit par une perte de place, ce qui en fait diminue les possibilités d'optimisation topologique.

Cette solution peut être mise en oeuvre de 2 façons différentes :

- soit c'est au niveau de l'opérateur que les indicateurs arithmétiques sont amenés à la hauteur des bus correspondant aux bits du registre destination spécifiés par le concepteur.
- soit l'organisation topologique des indicateurs arithmétiques des opérateurs (additionneurs, UALs) est figée et c'est au niveau du registre destination que l'on génèrera les commandes et les chemins d'accès nécessaires au transfert des indicateurs arithmétiques. C'est de cette façon qu'il est le plus facile de réaliser les chemins d'accès aux bits du registre destination. Les indicateurs arithmétiques d'un même opérateur ne sont en effet pas forcément connectés aux mêmes bits des registres destinations selon les opérations effectuées.

Une solution intermédiaire consiste à ne donner la possibilité au concepteur de transférer les indicateurs arithmétiques que dans un champ de bits contigus d'un registre ; la spécification du transfert devient alors plus aisée.

Exemple :
BOP(A0:16, B0:16, +, C0:16, RE 12:4)
On conviendra alors que les indicateurs arithmétiques carry, negative, overflow et zero seront transférés respectivement dans RE 12:1, RE 13:1, RE 14:1 et RE 15:1.

Chapitre IIII

Le compilateur Apolllon

1. Présentation générale du compilateur

Le système Apollon est un générateur automatique de parties opératives de circuits intégrés VLSI de type microprocesseur. Il fait partie du compilateur de silicium SYCO décrit au chapitre I. Apollon est basé sur un modèle architectural et topologique. L'entrée d'Apollon est constituée d'un ensemble non ordonné d'instructions opératives extraites de la description algorithmique du circuit. Chaque instruction opérative est constituée d'un ensemble d'actions opératives dont le format est prédéfini (transferts, opérations unaires et binaires, échanges avec l'extérieur) à exécuter en parallèle. La description d'entrée est obtenue en éliminant les primitives de contrôle (if, case, next, execute, exit) de la description algorithmique et en combinant les actions opératives correspondant à des actions conditionnelles en parallèle aux actions opératives inconditionnelles. Apollon génère les spécifications des masques de la partie opérative à partir de cette description de niveau comportemental de type transfert de registres en deux étapes.
Le compilateur génère d'abord l'architecture de la partie opérative, puis les spécifications des masques à partir de cette architecture et d'une bibliothèque de cellules prédéfinies. Le compilateur utilise une architecture cible pour générer l'architecture de la partie opérative. Cette architecture est dérivée de celle de la partie opérative du MC68000 [SUZI 81], [ANCE 83], [ANCE 84]. La partie opérative ne peut être constituée que par un ensemble de sous parties opératives alignées à deux bus.
Deux sous parties opératives voisines peuvent communiquer entre elles par l'intermédiaire de ces deux bus. Une instruction opérative doit s'exécuter en au plus deux cycles opératifs, un cycle étant décomposé en 4 phases correspondant aux phases d'exécution d'un transfert.
Apollon utilise une bibliothèque de cellules standard NMOS pour générer les masques. La partie opérative est réalisée à l'aide d'une structure "bit-slice". Elle est formée d'un bloc rectangulaire qui résulte de l'empilement de tranches horizontales d'un bit. Chaque élément fonctionnel de n bits de la partie opérative est formé d'une tranche verticale de n bits.

2. Spécifications d'entrée et de sortie du compilateur

Apollon génère les spécifications des masques de la partie opérative et toutes les informations nécessaires à la génération de la partie contrôle à partir d'une description comportementale de la partie opérative de niveau transfert de registres extraite automatiquement de l'algorithme de spécification du circuit.
Nous donnerons ici les spécifications d'entrée et les spécifications de sortie, à la fois intermédiaires et finales.
Les résultats de la première étape de la compilation sont constitués d'une part de la structure fonctionnelle de la partie opérative (ensemble de blocs fonctionnels - registres, constantes, tampons des plots d'entrée-sortie, boîtes à opérations -

interconnectés par des bus) et d'autre part des chemins de données utilisés pour l'exécution des instructions.
Les résultats de la compilation consistent:
- des spécifications des masques de la partie opérative,
- des spécifications des ports d'entrée-sortie et des segments frontières de la partie opérative pour l'assemblage global du circuit,
- et des informations nécessaires à l'interfaçage de la partie opérative avec la partie contrôle: spécifications des commandes de la partie opérative nécessaires à l'exécution de chaque instruction et de l'ordre des fils de compte rendu pour la génération de la partie contrôle.

2.1. Entrée d'Apollon

L'entrée d'Apollon est constituée de 2 parties : une partie de déclaration des ressources utilisées dans la partie opérative et une partie de spécification des instructions opératives. Une instruction opérative est constituée d'un ensemble d'actions opératives de types prédéfinis devant s'exécuter en parallèle en au plus deux cycles opératifs.

L'entrée du compilateur SYCO n'est pas tout à fait une entrée de niveau comportemental dans la mesure où le compilateur alloue un registre à chaque variable sans chercher à minimiser le nombre de registres et où le compilateur ne cherche pas à minimiser le nombre d'instructions données par l'utilisateur en essayant par exemple d'augmenter le taux de parallélisme dans la partie opérative. L'utilisateur doit donc déclarer les registres qu'il utilise et spécifier les actions opératives qui sont en parallèle.
L'utilisateur ne donne pas pour autant la structure de la partie opérative. A partir des ressources utilisées, le compilateur construit la partie opérative; il place registres, constantes et opérateurs dans les différentes sous-parties opératives tout en générant les connexions de ces différents éléments aux deux bus du modèle d'Apollon.
Donnons un exemple simple de description Apollon : le calcul de la division euclidienne.

Version optimisée de la division euclidienne

```
procedure division-euclidienne (A, B : integer; var Q, R : integer);

(* Cette procédure effectue la division euclidienne de A par B et fournit le
quotient dans Q et le reste dans R *)

var C : integer;

begin
```

```
    C := B;
    while C <= A do C := C * 2;
    R := A; Q:= 0;
    while C > B do
    begin
        Q := Q * 2; C := C div 2;
        if R >= C then
        begin
            Q := Q + 1; R := R - C;
        end;
    end;
end;
```

Description Apollon de cette procédure de division euclidienne

```
registres : A B C Q R RE ;
entrées-sorties : IOR 0:8 ;
sorties : ;
champs de contrôle : RE 4:4 ;
sous-pops: 5 ;
bits : 8 ;
figure : sleuc ;

C <- B / BOP(, B, <=, A, RE 4:4) ;
C <- mult2 C ;
BOP(, C, <=, A, RE 4:4) ;
R <- A / Q <- 0 / BOP(, C, >, B, RE 4:4) ;
Q <- mult2 Q / C <- div2 C ;
BOP(, R, >=, C, RE 0:4) / BOP(, C, >, B, RE 4:4) ;
Q <- incr Q / R <- R - C ;
BOP(, C, >, B, RE 4:4) ;
A <- IOR ;
B <- IOR ;
IOR <- Q ;
IOR <- R .
```

2.1.1. Déclaration des ressources

2.1.1.1. Registres

Les registres sont donc déclarés par l'utilisateur. Chaque registre est caractérisé par les informations suivantes :
- son nom
- sa taille : le nombre de bits du registre
- sa structure d'accès par la partie contrôle : elle est constituée du nombre de fils de compte-rendu et de leurs numéros
- sa structure d'accès par la partie opérative : elle est constituée du nombre et de la nature (origine et longueur) des champs accédés dans la partie opérative.

Les champs de bits des registres accédés par la partie contrôle peuvent être extraits facilement de la description algorithmique initiale : ils sont utilisés dans les parties condition des actions conditionnelles (IF, CASE). Le compilateur a besoin de cette information pour ne remonter vers la partie contrôle que les fils correspondant aux bits effectivement testés par la partie contrôle. Cette information n'est prise en compte que dans la phase de génération des masques car elle ne rentre pas en jeu dans la détermination des chemins de données.
Pour chaque registre, on doit fournir au compilateur la liste des champs (un champ est défini par son origine et sa longueur) de bits accédés simultanément par la partie opérative, c'est à dire en d'autres termes les groupes de bits transférés en même temps dans la partie opérative. La structure d'accès des registres est consultée à la fois pendant la phase de génération de la structure fonctionnelle et pendant la phase de génération des fichiers de masques. En effet, le compilateur doit s'assurer que les bus sont disponibles pour transférer tel champ de bits; on peut effectuer simultanément les transferts suivants:
(A 0:8 := R 0:8; B 8:8 := R 8:8;)
De plus il doit générer des commandes distinctes pour l'accès des champs 0:8 et 8:8.
Actuellement seuls le nom du registre, sa taille et sa structure d'accès par la partie contrôle sont pris en compte par Apollon.

2.1.1.2. Bus d'entrée-sortie

Les bus d'entrée-sortie sont déclarés dans le bloc "signal" du "smodule" de la description du circuit en LDS. A chaque bus de données (mais pas aux bus de contrôle bien sûr) est associé un tampon de la partie opérative qui a le même nombre de bits et le même type (entrée-sortie ou sortie ou entrée).

2.1.1.3. Nombre maximum de sous-parties opératives

L'utilisateur doit donner un nombre maximum de sous-parties opératives à ne pas dépasser; il fixe ainsi une limite, ce qui permet au compilateur d'une part d'arrêter la compilation dès qu'il constate un dépassement de ce nombre et d'autre part de diminuer la taille des structures de données utilisées. Dans la prochaine version de SYCO, ce nombre sera déterminé par l'évaluateur.

2.1.1.4. Préplacement des registres

L'utilisateur peut placer certains registres lui même. Cette possibilité peut permettre à l'utilisateur de trouver une solution ou d'améliorer la solution donnée par le compilateur. En effet l'allocation des ressources est un problème NP-complet et on est obligé de recourir à des heuristiques pour trouver une solution en un temps raisonnable.

2.1.2. Spécification des instructions opératives

La deuxième partie de la description est constituée d'un ensemble non ordonné d'instructions opératives à exécuter en au plus deux cycles opératifs. Chaque instruction opérative est constituée d'un ensemble d'actions opératives à exécuter en parallèle.
Notons que c'est le concepteur qui spécifie le parallélisme à mettre en oeuvre dans la partie opérative. Le rôle de l'extracteur est uniquement de séparer le flot de données du flot de contrôle. Les actions opératives ont un format prédéfini. Il existe trois types d'actions opératives :

- le transfert de données à l'intérieur de la partie opérative: le registre destination prend la valeur du registre source.
exemple de transfert : A := B;

- le traitement de données : une opération ne peut avoir qu'un ou deux opérandes. L'opérateur peut être représenté par une boîte noire à une ou deux entrées, les opérandes et deux sorties :
- le résultat (la somme pour l'addition)
- le compte-rendu d'exécution: un certain nombre d'indicateurs arithmétiques sont positionnés automatiquement lors de l'opération (indicateurs de retenue et de débordement pour l'addition)
exemple d'opération : A := B + C;

- l'échange de données avec une mémoire externe
exemple de lecture-mémoire: A := busadresses;

2.1.2.1. Transfert de données

Les horloges de lecture et d'écriture sont non recouvrantes (cf 4.1.3).
L'instruction (A:=B; B:=C;) sera exécutée de la façon suivante :
A prend l'ancienne valeur de B
B prend la valeur de C.
Le transfert peut porter sur tous les bits du registre ou sur un nombre quelconque de bits.
On appelle multitransfert un ensemble de transferts qui ont la même source et qui se déroulent en même temps.

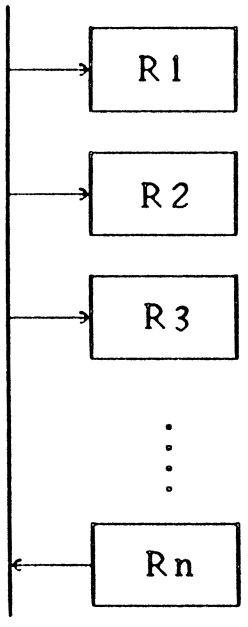


figure 3.1 : multitransfert

2.1.2.2. Traitement de données

Apollon ne permet que l'utilisation d'opérateurs unaires ou binaires. Une opération est une expression simple pouvant prendre les trois formes suivantes:
opération unaire : A:=-B;
opération binaire : A:=B+C;
opération avec positionnement d'indicateurs arithmétiques :
BOP(A,B,+,C,RE);
Cette action opérative est équivalente à:
(A:=B+C; RE:=CR(B + C);)
où l'on note CR(B+C) le compte rendu d'exécution de l'opération B+C; les indicateurs arithmétiques sont transférés dans le registre RE.
Si A est omis, le résultat n'est pas conservé; si C est omis, l'opération est unaire.
La fonction BOP (pour BOîte à OPérations) permet de décrire tout opérateur comme une boîte noire à une ou deux entrées et une ou deux sorties : le résultat et/ou le compte rendu d'exécution.
L'utilisation de la fonction BOP permet de conserver les valeurs des indicateurs arithmétiques et de les regrouper à l'intérieur d'un même registre (ceci pour des raisons d'optimisation topologique). Par souci de simplification, nous n'avons pas pour le moment précisé la structure d'accès aux bits d'un registre.
Notons comme dans le langage LDS:
R 0:8 les bits 0 à 7 du registre R.
Une opération peut donc prendre la forme générale suivante:
BOP(A m:n, B m:n, +, C m:n, RE m1:n1);

L'opération décrite porte sur n bits, plus précisément des bits m à m+n-1 des registres A, B et C.
Une boîte à opérations a un nombre fixé de fils de compte rendu. Les fils de compte rendu de la boîte à opérations d'Apollon correspondent aux indicateurs arithmétiques classiques d'une UAL.
Ce sont:
- la retenue
- le débordement
- le signe
- l'égalité à zéro

Pour simplifier l'établissement d'une correspondance entre les bits du registre destination et les indicateurs arithmétiques, on impose d'une part n1=4 et d'autre part que les bits m1 à m1+3 correspondent aux indicateurs suivants:
m1: retenue
m1+1: débordement
m1+2: signe
m1+3: égalité a zéro

Dans un premier temps nous limiterons l'usage des champs de bits dans la fonction BOP. Une opération portera sur tous les bits de la partie opérative.
BOP(A 0:n, B 0:n, +, C 0:n, RE m-4:4);
où m <= n.
L'utilisation d'opérations portant sur des champs de bits quelconques pose des problèmes de circuiterie. La génération des indicateurs arithmétiques est localisée au niveau du bit de poids le plus fort. D'autre part la retenue entrante s'il y en a une doit arriver au niveau du bit de poids le plus faible.
Si l'on veut effectuer une addition portant sur les bits 3 à 8 à l'aide d'un additionneur 16 bits, on doit modifier les cellules de l'additionneur au niveau des bits 3 et 8.
Notons que l'utilisateur ne peut décrire que des opérations simples unaires ou binaires. Il est en cela très dépendant du modèle architectural d'Apollon. D'une part il ne peut utiliser d'opérateurs ternaires ou plus généralement n-aires avec n>= 3. D'autre part il doit lui même transformer une opération complexe en une séquence d'opérations simples. C'est donc lui qui gère l'utilisation des registres et non le compilateur.
D:=A+B+C;
peut être transformé en par exemple:
D:=A+B;
D:=C+D;
Dans la version actuelle, les opérateurs de la bibliothèque NMOS sont les suivants:
- opérateurs binaires:
  .binaires: +, -
  .logiques: et, ou, ouex, non-et, non-ou, non-ouex
- unaires:
  .arithmétiques: incrémentation, décrémentation, opposé
  .logique: complémentation bit à bit

.décalage: à gauche, à droite d'une position avec mise à un ou mise à zéro du bit le plus à droite ou le plus à gauche, rotation à gauche ou à droite d'une position.
Toute opération plus complexe comme la multiplication doit donc être transformée en une séquence d'additions et de décalages.

2.1.2.3. Echange de données avec une mémoire externe

Le compilateur de silicium SYCO ne génère que la partie opérative et la partie contrôle d'un circuit. Toutes les mémoires sont donc externes. Pour des raisons de synchronisation et de séquencement, une variable d'entrée-sortie ne peut figurer que dans un transfert simple et non dans une opération unaire ou binaire.
Exemple:
A:=busext;

2.2. Résultats de la compilation

Rappelons qu'Apollon génère les masques de la partie opérative en 2 étapes.
La description comportementale est d'abord transformée en une description structurelle, puis en une description géométrique.

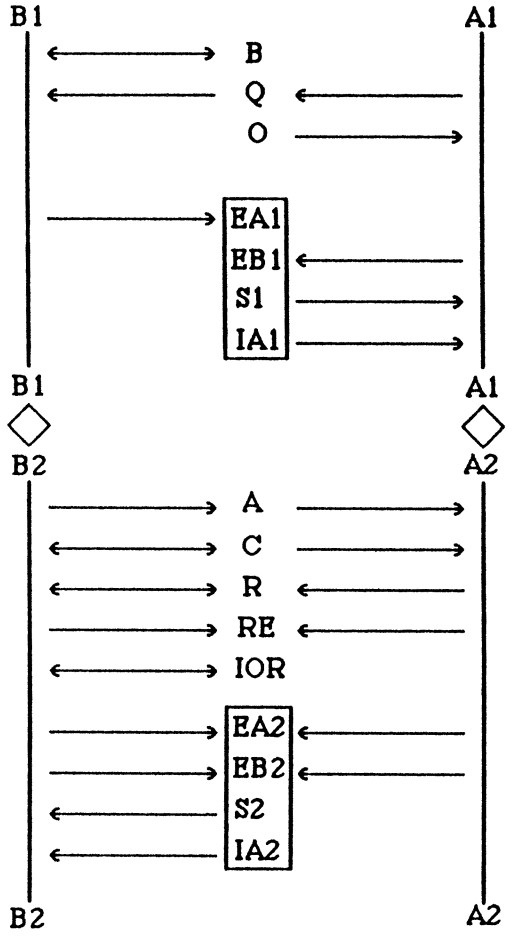
2.2.1. Résultats de la première étape de la compilation

Apollon génère la description fonctionnelle de la partie opérative. Cette description se compose de deux parties :
- une description structurelle sous la forme de blocs interconnectés: cette description peut être faite au moyen d'un langage de description de circuits comme LDS. Actuellement, le compilateur génère le dessin des différents blocs fonctionnels de la partie opérative et les structures de données nécessaires à la génération des masques.
- une description comportementale plus fine des instructions opératives.

Apollon génère le séquencement des instructions opératives en cycles puis en phases pour chaque cycle.
Cette description fait intervenir des transferts de types bus-registre et registre-bus ainsi que des sélections des opérations des boîtes à opérations chaque cycle de la partie opérative (une opération unaire ou binaire prend 2 cycles). Cette description est constituée des chemins de données empruntés pour exécuter les instructions opératives à chaque cycle.

2.2.1.1. Description structurelle

La partie opérative résulte de la juxtaposition de plusieurs sous-parties opératives. Apollon doit donc fournir une liste ordonnée de sous-parties opératives et les connexions éventuelles entre sous-parties opératives voisines (pas de connexion - connexion d'un bus - connexion des deux bus).



sous-pop N°1 : > incr mult2
sous-pop N°2 : >= - > <= div2 mult2

Partie opérative pour le calcul de la division euclidienne
figure 3.2

Chaque sous-partie est constituée d'un ensemble d'éléments de mémorisation, de traitement et de tampons de plots d'entrée-sortie interconnectés par des éléments de transfert.
Une sous-partie opérative est donc déterminée par l'ensemble de ses éléments constitutifs ainsi que par l'ensemble des connexions entre ces éléments et les éléments de transfert. Le nombre et la nature des éléments de transfert d'une sous-partie oprative sont fixés par le modèle architectural d'Apollon : il n'y a que deux bus.
Le compilateur génère la liste des éléments de chaque sous-partie opérative :
- registres
- constantes
- tampons d'entrée-sortie
- opérateurs

(seuls les registres et les tampons d'entrée-sortie ne peuvent pas être dupliqués - cf 3.1)
ainsi que les connexions en lecture et en écriture de chacun de ces éléments aux deux bus de la partie opérative.

2.2.1.2. Description de niveau comportemental

La description comportementale de départ est transformée en une description de niveau plus fin: chaque opération est d'abord décomposée en transferts élémentaires, chacun s'exécutant en un cycle.
Ce niveau de séquencement généré par Apollon est réalisé par la partie contrôle.
Le compilateur donne ensuite pour chaque transfert le bus utilisé et pour chaque opération la sous partie opérative où elle s'effectue.

```
==========================================================
instruction No 1 :   C <-- B . BOP(, B, <=, A, RE 4:4)
sous-pop 2 : <=
  phi1    B     --->  B1
  phi1    A     --->  A2 ---> EA2
  phi1    B1    --->  B2 ---> EB2   C
  phi2    IA2   --->  B2 ---> RE 4:4
==========================================================
instruction No 2 :   C <-- mult2 C
sous-pop 2 : mult2
  phi1    C     --->  B2 ---> EA2
  phi2    S2    --->  B2 ---> C
==========================================================
instruction No 3 :   BOP(, C, <=, A, RE 4:4)
sous-pop 2 : <=
  phi1    A     --->  A2 ---> EA2
  phi1    C     --->  B2 ---> EB2
  phi2    IA2   --->  B2 ---> RE 4:4
==========================================================
instruction No 4 :   R <-- A    Q <-- 0    BOP(, C, >, B, RE 4:4)
sous-pop 2 : >
  phi1    B     --->  B1
  phi1    C     --->  A2 ---> EB2
  phi1    B1    --->  B2 ---> EA2
  phi2    0     --->  A1 ---> Q
  phi2    A     --->  A2 ---> R
  phi2    IA2   --->  B2 ---> RE 4:4
==========================================================
```

Exécution détaillée de quelques instructions de la procédure de division euclidienne

figure 3.3

Cette description introduit un nouveau niveau de séquencement : celui des phases. Le séquencement en phases des cycles est réalisé au moyen d'amplificateurs de validation des commandes dans une tranche située juste au dessus de la partie opérative.
Les transferts registre-bus se font à la phase T2, les transferts bus-registre à la phase T4, les opérations se déroulent sur 2 cycles, de la fin du premier cycle au début du second.
C'est de cette description qu'on extrait les commandes de sélection des registres sur les bus et des opérations à exécuter sur les boîtes à opérations.

2.2.2. Résultats de la seconde étape de la compilation

2.2.2.1. Spécifications des masques

Les masques sont décrits à l'aide du langage de description géométrique de circuits LUCIE. Les spécifications des masques sont obtenues à l'aide de l'assembleur de silicium du système LUCIE, LUBRICK. Le circuit de la partie opérative est défini par la donnée de la figure principale et de toutes les figures externes appelées dans la figure principale et récursivement de celles appelées dans ces dernières figures, que ces figures soient générées par le compilateur ou qu'elles appartiennent à la bibliothèque utilisée. En exemple nous donnons figure 3.4 le dessin des masques de la partie opérative exécutant la division euclidienne.

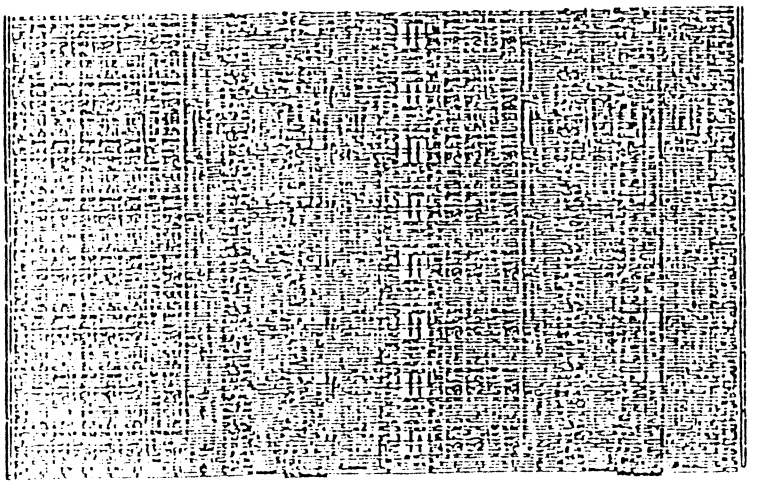


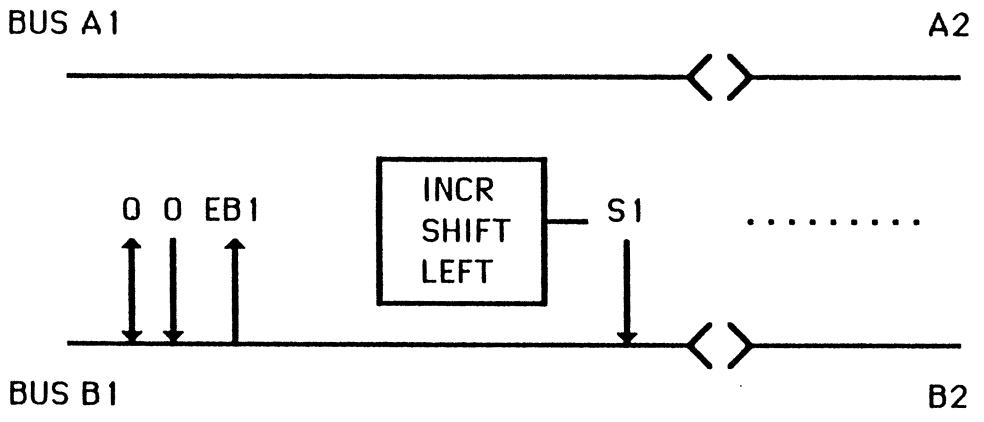
figure 3.4 : dessin des masques de la partie opérative résultant de la compilation de l'algorithme de la division euclidienne

2.2.2.2. Spécifications d'assemblage

Lubrick génère à partir des spécifications d'assemblage des figures de base les spécifications d'assemblage de la figure englobante. L'assemblage sera détaillé dans la section 4.3.

2.2.2.3. Spécifications des commandes de la partie opérative et de l'ordre des comptes rendus

La position des comptes rendus est obtenue à partir de l'ordre des registres de la partie opérative.
Pour obtenir les commandes pour chaque cycle, on numérote les commandes à partir de la place des éléments de la partie opérative et du nombre des commandes de chaque élément, ce nombre dépendant de la bibliothèque utilisée.



1. lecture / écriture : Q <--> B1
2. validation : Q <--> B1
3. lecture : 0 --> B1
4. écriture : B1 --> EB1
5. retenue entrante
6. décalage à gauche
7. pas de décalage
8. écriture : S1 --> B1
9. connexion : A1 <--> A2
10. connexion : B1 <--> B2

figure 3.5 : numérotation des commandes

Pour chaque cycle, on génère alors la liste des commandes ordonnées.

3. Génération de l'architecture de la partie opérative

La compilation de la partie opérative à partir de la description algorithmique du circuit se déroule en trois étapes.
La génération de la description d'entrée d'Apollon de niveau transfert de registres à partir de l'algorithme de spécification du circuit a été présentée au chapitre I.
Nous allons maintenant présenter la génération de l'architecture de la partie opérative, c'est à dire la décomposition de la partie opérative en blocs fonctionnels: registres, constantes, bus et opérateurs et leurs interconnexions.
Nous présenterons d'abord le modèle architectural et temporel de la partie opérative, puis l'algorithme d'allocation des ressources.

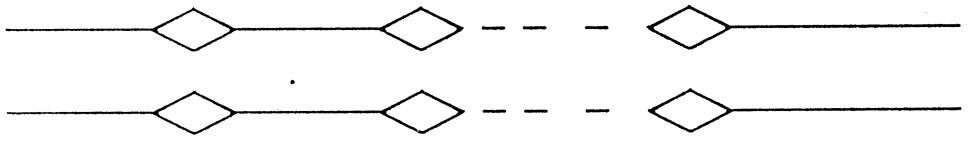
3.1. Modèles architectural et temporel de la partie opérative

Dans ce paragraphe, nous présentons d'abord le modèle architectural et temporel

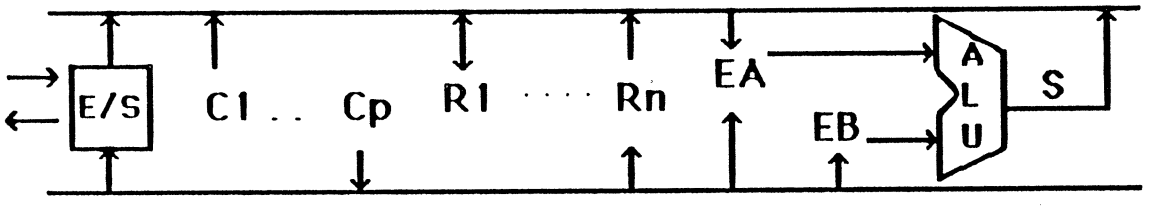
utilisé par Apollon, puis nous mettrons en évidence un certain nombre de limites du modèle avant de présenter d'autres modèles possibles.

3.1.1. Description des modèles architectural et temporel

Architecture



Modèle architectural de la partie opérative
figure 3.6



Modèle architectural d'une sous partie opérative
figure 3.7

Le modèle architectural de la partie opérative d'Apollon est dérivé de l'architecture de la partie opérative du MC68000. Le choix d'un modèle à 2 bus pour la compilation des parties opératives est préconisé dans [ANCE 83] et [ANCE 84]. Mead et Conway [MEAD 80] préconisent aussi l'utilisation de parties opératives à deux bus. Le système de Johannsen [JOHA 79], Bristle Blocks permet d'assembler des parties opératives dont l'architecture est pratiquement identique à celle des parties opératives générées par Apollon. Les deux bus peuvent couvrir toute la partie opérative ou être morcelés alors que le générateur de parties opératives de Shrobe [SHRO 82], DPG utilise des bus dissymétriques: l'un est global et ne peut être morcelé, l'autre est local.
La partie opérative d'Apollon est donc basée sur une architecture à deux bus.
Ces deux bus peuvent être morcelés, ce qui permet de définir des sous parties opératives. La partie opérative est constituée d'un ensemble de sous parties opératives à deux bus placées côte à côte et fonctionnant en parallèle.
Chaque sous partie opérative peut fonctionner de manière indépendante (exemple: une sous partie opérative pour les adresses et une autre pour les données) ou être connectée à une de ses voisines par l'intermédiaire d'un ou des deux bus.
Chaque sous partie opérative est constituée d'un ensemble d'éléments de mémorisation:

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

Modèle temporel

Chacune des actions opératives définies précédemment peut prendre un ou deux cycles opératifs. Un transfert prend un cycle alors qu'une opération en prend deux.

Un cycle opératif est constitué d'une succession de phases opératives. Les phases opératives correspondent aux différentes étapes du transfert et du traitement des données dans la partie opérative.

Le cycle opératif d'Apollon comprend 4 phases:
1) précharge des bus
2) lecture des registres sources
3) amplification
4) écriture dans les registres destinations

Modalités d'exécution des actions opératives

Un transfert prend donc un cycle opératif.

Une opération se déroule sur deux cycles opératifs:
premier cycle: transfert du ou des opérandes dans les tampons d'entrée de l'opérateur et début du traitement
deuxième cycle: fin du traitement et transfert du résultat et du compte rendu d'exécution dans les registres destination et d'indicateurs arithmétiques.

Un échange de données avec l'extérieur se déroule aussi sur deux cycles:
lecture d'un mot de la mémoire:
premier cycle: le mot mémoire est transféré dans le tampon d'entrée correspondant de la partie opérative,
deuxième cycle: transfert du tampon dans le registre destination.
écriture d'un mot en mémoire:
premier cycle: transfert du registre source dans le tampon de sortie correspondant à la mémoire choisie,
deuxième cycle: transfert du tampon dans la mémoire.

Nous avons décrit l'exécution des actions opératives en termes de cycles opératifs. Recensons maintenant les besoins de ces différentes actions en bus. Nous considérons que les registres sont déja placés. Pour exprimer le nombre de bus nécessaires pour chaque action, introduisons la notion de distance entre deux registres.

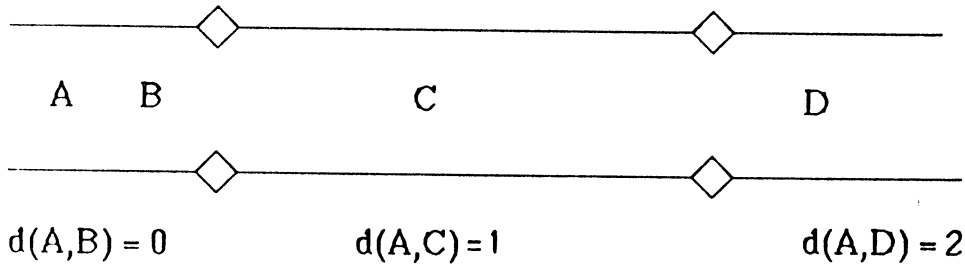


figure 3.8 : distance entre 2 registres

La distance entre deux éléments de la partie opérative est définie comme le nombre de sous parties opératives augmenté d'un qui séparent ces deux éléments. La distance entre deux éléments d'une même sous partie opérative est nulle. Pour un transfert: A := B, le nombre de bus nécessaires pour exécuter ce transfert est:
n=1+d(A,B)
d'où n≥1
Ce transfert peut être exécuté au premier cycle ou au second cycle si l'instruction où figure ce transfert doit se dérouler sur 2 cycles. Pour une opération plusieurs cas sont à distinguer:
opération unaire, - par exemple: A := - B
n1=1+d(B,-)
n2=1+d(-,A)
où n1 est n2 désignent le nombre de bus au premier et au second cycle de l'opération,
soit en tout n=2+d(B,-)+d(-,A)

Pour une opération binaire, + par exemple: A := B + C
n1=2+d(B,+)+d(C,+)
n2=1+d(+,A)
n=3+d(B,+)+d(C,+)+d(+,A)

Pour une opération unaire avec positionnement d'indicateurs: BOP(A,B,-,,RE)
n1=1+d(B,-)
n2=2+d(-,A)+d(-,RE)
n=3+d(B,-)+d(-,A)+d(-,RE)

Pour un test unaire: BOP(,B,-,,RE)
n=2+d(B,-)+d(-,RE)

Pour une opération binaire avec positionnement d'indicateurs: BOP(A,B,+,C,RE)
n=4+d(B,+)+d(C,+)+d(+,A)+d(+,RE)

enfin pour un test binaire: BOP(,B,>,C,RE) on a:
n=3+d(B,>)+d(C,>)+d(>,RE)

Le nombre de bus nécessaires à l'exécution d'une opération est la somme d'un nombre fixe (le nombre des opérandes et des destinations) et d'un nombre variable qui dépend de la distance de l'opérateur aux registres sources et destinations en termes de nombre de sous parties opératives. Le programme de placement cherche à minimiser le nombre de bus nécessaires à l'exécution et par conséquent à minimiser la distance entre les différents registres de l'opération ainsi que la distance de l'opérateur aux registres sources et destinations.

Pour un échange de données avec l'extérieur, deux cas sont à distinguer:

Lecture d'un mot de la mémoire: A:=busext
Notons par B le tampon associé à busext
n2=1+d(B,A)
le transfert s'exécute au deuxième cycle

Ecriture d'un mot en mémoire: busext:=A
n1=1+d(A,B)

3.1.2. Limites du modèle

Le parallèlisme d'exécution n'est possible que dans certaines limites; le modèle marche bien tant que les tâches effectuées simultanément sont indépendantes les unes des autres, par exemple dans le cas du traitement des adresses et du traitement des données. Mais dès que l'on veut effectuer simultanément des calculs sur les mêmes données, on rencontre des problèmes.
Exemples:
L'instruction:
(D1 := A op1 B; D2 := B op2 C; D3 := C op3 A;)
ne peut être exécutée par une partie opérative à deux bus en 2 cycles et ceci quelles que soient les destinations D1, D2 et D3 et les opérateurs op1, op2 et op3.
Les permutations circulaires de trois registres:
(A := B; B := C; C := A;)
ne sont pas exécutables en un cycle.
Remarquons que dans les 2 cas, les actions opératives portent sur les mêmes registres. Une analyse détaillée de ces 2 instructions sera présentée plus loin.
Dans la suite de ce paragraphe, nous dégagerons d'abord des conditions nécessaires ou nécessaires et suffisantes pour l'exécution d'une seule instruction opérative sur une partie opérative à 2 bus segmentables en au plus 2 cycles. Nous nous placerons dans le cas suivant:
- seuls les registres ne peuvent être dupliqués; en particulier les opérateurs pourront être librement dupliqués,
- nous nous intéressons à l'exécution d'une seule instruction : aucun des registres n'est déjà placé.

Ces conditions peuvent être utilisées à la fois pour réduire l'espace de recherche en éliminant dès le début les configurations impossibles et pour détecter les instructions impossibles à exécuter sur une partie opérative à 2 bus en 2 cycles.
Nous montrerons ensuite que même si plusieurs instructions sont exécutables seules, l'exécution de cet ensemble d'instructions en séquence par une même partie opérative de type Apollon n'est pas toujours possible.

3.1.2.1. Conditions nécessaires et suffisantes à l'exécution d'instructions isolées

Nous examinerons successivement le cas des instructions ne comportant que:
- des transferts,
- des opérations binaires.

3.1.2.1.1. Instructions comportant des transferts

Le modèle d'Apollon spécifie que les instructions doivent être exécutées en au plus 2 cycles. Une instruction ne comportant que des transferts peut être exécutée en un

seul cycle.

Exécution de transferts concurrents en un cycle

On peut toujours exécuter 2 transferts en un cycle sur une partie opérative à 2 bus.
Considérons donc une instruction comportant trois transferts:
(A2:=A1; B2:=B1; C2:=C1;)
On suppose bien sûr que la source et la destination d'un même transfert sont distinctes et que toutes les destinations sont distinctes.
Une condition nécessaire et suffisante pour que ces trois transferts soient exécutables en un cycle est:
$[A1\ A2] \cap [B1\ B2] \cap [C1\ C2] = 0$
où on désigne par [A1 A2] l'ensemble des sous parties opératives comprises entre A1 et A2, celle(s) contenant A1 et A2 compris.
Cette condition exprime que les trois chemins utilisés pour effectuer ces transferts ne passent pas sur une même sous partie opérative : en effet comme une sous partie opérative n'a que deux bus, elle ne peut supporter que deux transferts.
Le placement des registres ci-dessous ne respecte pas la condition nécessaire et suffisante énoncée, ce qui conduit à un conflit d'utilisation des bus.

figure 3.9 : transferts conflictuels

Les seules instructions comportant trois transferts non exécutables en un cycle sont les permutations circulaires de trois registres:
(A := B; B := C; C := A;)
On peut facilement généraliser et montrer qu'une permutation de n registres:
(A1 := A2; A2 := A3; ...; An := A1;)
n'est pas exécutable en un cycle.

Ce n'est pas le seul type d'instructions non exécutables en un cycle. Examinons l'exemple suivant:
(B2:=B1; B1:=B; B:=A; C2:=C1; C1:=C; C:=A; D2:=D1; D1:=D; D:=A;)

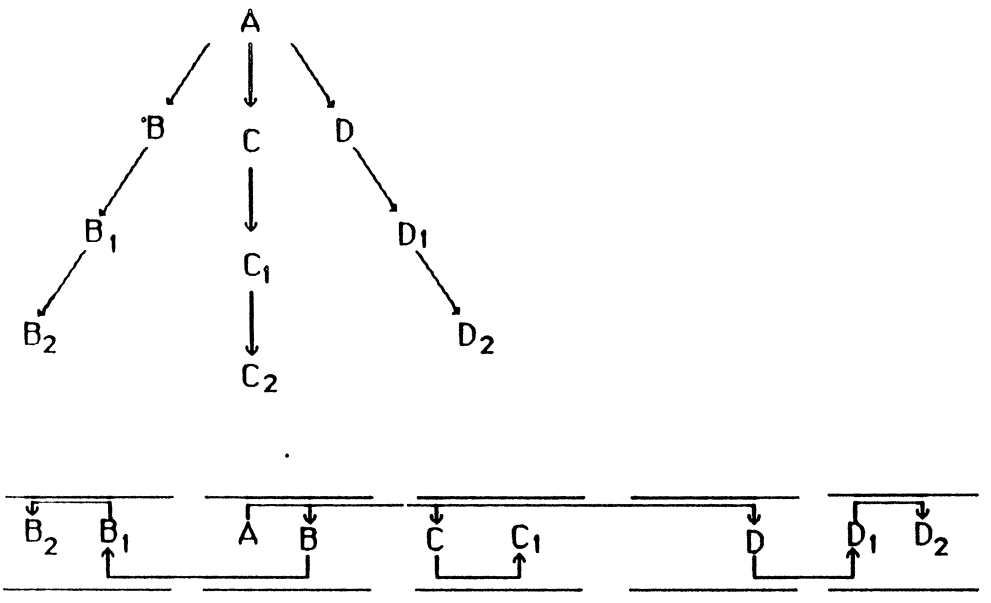
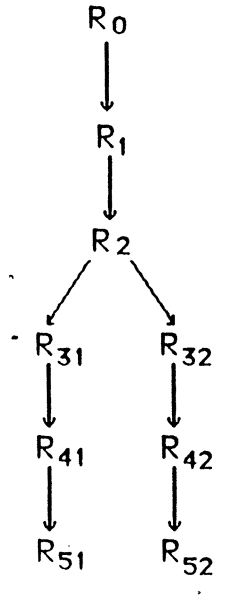


figure 3.10 : exemple de 9 transferts non exécutables en un cycle

Il s'agit d'un multitransfert à 3 destinations différentes et où chacune de ces destinations est source d'un transfert dont la destination est elle même source d'un autre transfert. Le problème est symétrique en ce qui concerne les destinations du multitransfert. Il est clair que par exemple si C est placé entre les deux autres destinations du multitransfert, le transfert C2:=C1 ne pourra pas être exécuté.
On peut donner un autre exemple de transferts non exécutables en un cycle:
(R2:=R1; R1:=R0;
R51:=R41; R41:=R31; R31:=R2;
R52:=R42; R42:=R32; R32:=R2;)



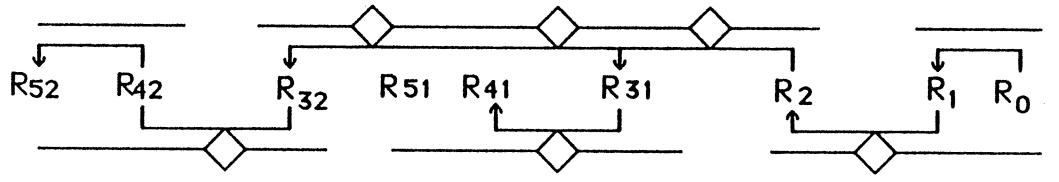


figure 3.11 : exemple de 8 transferts non exécutables en un cycle

On ne peut exécuter le transfert R51:=R41.

Essayons donc de déterminer un sous ensemble d'instructions exécutables en un cycle et qui ne comportent que des transferts. Pour cela considérons le graphe des transferts où tout transfert est représenté par un arc orienté de la source vers la destination du transfert.

Montrons que s'il n'existe pas de chemin de longueur supérieure ou égale à 2 dans le graphe, on peut exécuter ces transferts en un cycle.
Les transferts d'une composante connexe du graphe des transferts peuvent être exécutés indépendamment des transferts des autres composantes connexes.
Deux cas peuvent se présenter :

- soit les transferts d'une composante connexe forment un cycle d'ordre 2. On place alors les 2 registres dans la même sous partie opérative.
- soit il n'y a pas de cycle. Il existe alors un sommet de la composante connexe qui n'est destination d'aucun transfert. Si ce sommet est source de plusieurs transferts, il suffit de placer les registres des différents chemins qui partent de ce registre dans des sous parties opératives distinctes; sinon une seule sous partie opérative par composante connexe suffit.

On démontre ainsi que toute instruction dont le graphe des transferts ne comprend pas de chemin de longueur d'au moins 3 est exécutable en un cycle. On peut montrer de la même façon que toute instruction ne comportant ni multitransfert ni cycles de longueur supérieure ou égale à 3 est exécutable en un cycle.
En effet s'il n'y a pas de multitransfert, une composante connexe du graphe des transferts n'est formée que d'un chemin :

- s'il n'y a pas de cycle,

A --> B --> C --> D --> E --> F --> G --> H --> ...

on dispose les registres de la façon suivante :

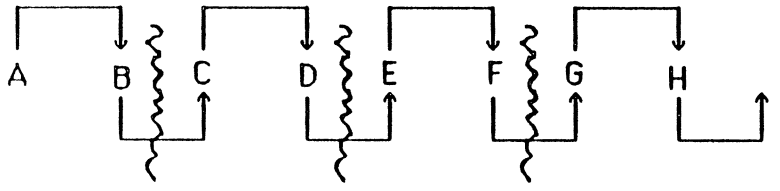


figure 3.12 : exécution d'une "chaîne" de transferts

- si le chemin est cyclique et que le cycle est d'ordre 2, il suffit de placer les 2 registres dans la même sous partie opérative; sinon on ne peut trouver aucune configuration qui permette d'exécuter les transferts du cycle en un cycle opératif. Si l'on coupe par exemple le cycle en un point, on peut se ramener au cas précédent. Mais la condition nécessaire et suffisante pour exécuter 3 transferts n'est pas respectée si on prend 2 transferts entre 3 registres consécutifs et comme troisième transfert le transfert entre les 2 extrémités du chemin créées par la coupure.

Exécution de transferts concurrents en 2 cycles

Le problème est le suivant. Si on ne peut exécuter un ensemble de transferts en un cycle, c'est qu'il y a au moins un chemin de longueur supérieure ou égal à 3. Si l'on décide d'exécuter ces transferts en 2 cycles, il faut soit exécuter tous les transferts d'un même chemin au même cycle, soit modifier la description sinon l'exécution de l'instruction en 2 cycles ne correspond pas aux spécifications données par l'utilisateur.
(A:=B; B:=C; C:=A;)
n'est pas équivalent à :
(A:=B; B:=C;)
C:=A;
mais à :
(D:=A; A:=B; B:=C;)
C:=D;

3.1.2.1.2. Instructions comportant des opérations binaires

Nous examinerons successivement les instructions comportant 2, 3 et un nombre quelconque d'instructions binaires.

Instructions comportant 2 opérations binaires

Montrons que 2 opérations binaires seules peuvent toujours être exécutées avec le modèle choisi. Distinguons 2 cas:
- les opérandes sont distincts:
(D1 := A op1 B; D2 := C op2 D;)
Une condition nécessaire et suffisante pour qu'un couple d'opérations binaires dont

les opérandes sont distincts soit exécutable en 2 cycles sur une partie opérative à 2 bus segmentables est que les opérandes vérifient les relations:
$[A\ B] \cap [C\ D] \neq [A\ B]$
$[A\ B] \cap [C\ D] \neq [C\ D]$
On note par [A B] l'ensemble des sous parties opératives entre A et B, celle(s) contenant A et B y compris.
On le montre en examinant les différents cas: on ne peut exécuter le chargement des opérandes de 2 opérations en un cycle que si les 2 opérandes d'une des opérations sont situés entre les 2 opérandes de l'autre opération.

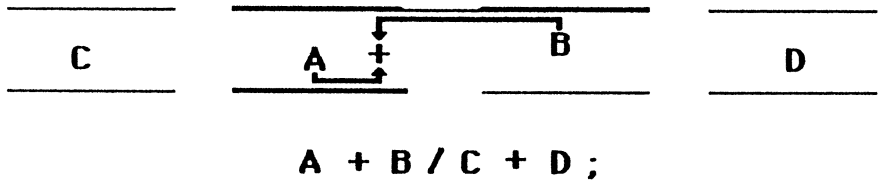


figure 3.13 : conflit d'utilisation des bus

En effet dans ce cas, tous les segments d'un bus doivent être utilisés pour transférer un des opérandes distants au niveau de l'autre alors que certains segments des 2 bus sont déjà utilisés pour charger les 2 opérandes qui sont au milieu, dans les tampons d'entrée de l'opérateur +. Le problème ne change d'ailleurs pas avec la position de cet opérateur.
- les deux opérations ont un opérande commun A:
(D1 := A op1 B1; D2 := A op2 B2;)
La condition nécessaire et suffisante devient : les deux autres opérandes B1 et B2 doivent être placés dans des sous parties opératives distinctes.
On peut remarquer que le placement des destinations D1 et D2 importe peu puisqu'elles n'interviennent qu'au second cycle et que l'on peut toujours effectuer 2 transferts en un cycle avec 2 bus.

Instructions comportant 3 opérations binaires

Considérons maintenant le cas des instructions à 3 opérations binaires.
Nous nous intéresserons d'abord au chargement des opérandes, puis au transfert des résultats.

Chargement des opérandes

Nous distinguerons 4 cas selon le nombre des opérandes différents:
- les 6 opérandes sont distincts: on peut montrer (en recensant tous les cas par exemple) que le chargement des opérandes de trois opérations binaires faisant intervenir respectivement les trois couples d'opérandes (A,B), (C,D) et (E,F) où tous les registres sont distincts, peut être effectué en un cycle si et seulement si les

conditions suivantes sont respectées:
$[A\ B] \cap [C\ D] \cap [E\ F] = 0$
et pour chaque combinaison de deux couples d'opérandes par exemple (A,B) et (C,D) on a
$[A\ B] \cap [C\ D] \neq [A\ B]$
$[A\ B] \cap [C\ D] \neq [C\ D]$

- 5 opérandes sont distincts; 2 opérations ont un opérande commun.

Soient les 3 couples d'opérandes: (A B1) (A B2) (C D).
Une condition nécessaire et suffisante pour que les opérandes puissent être chargés en un cycle est que:
1. B1 et B2 soient sur des sous parties opératives distinctes
$[A\ B1] \cap [C\ D] \neq [A\ B1] \neq [C\ D]$
$[A\ B2] \cap [C\ D] \neq [A\ B2] \neq [C\ D]$
2. et d'autre part $[B1\ B2] \cap [C\ D] \neq [B1\ B2]$ et $[B1\ B2] \cap [C\ D] \neq [C\ D]$
Remarque: on peut avoir $[A\ B1] \cap [A\ B2] \cap [C\ D] \neq 0$
Exemple:

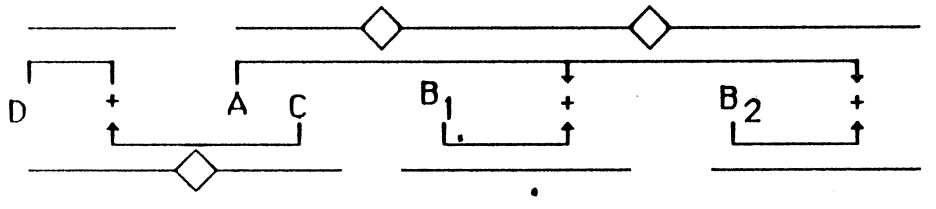


figure 3.14 : exécution de (C+D; A+B1; A+B2;)

- 4 opérandes sont distincts: 2 cas sont à considérer:

1. les 3 opérations ont un opérande en commun: (A B) (A C) (A D); une condition nécessaire et suffisante pour exécuter le chargement des opérandes est que B, C et D se trouvent sur 3 sous parties opératives distinctes.

2. 2 couples d'opérations ont un opérande commun mais qui est différent pour chacun d'eux: (A B) (B C) (C D); une condition nécessaire et suffisante pour exécuter le chargement des opérandes est que:
1. A et C soient sur des sous parties opératives distinctes ainsi que B et D; d'autre part $[A\ B] \cap [C\ D] \neq [A\ B]$ et $[A\ B] \cap [C\ D] \neq [C\ D]$
2. ni A, ni D n'appartient à $[A\ B] \cap [C\ D]$
Remarque: $[A\ B] \cap [B\ C] \cap [C\ D] = 0$ n'est pas nécessaire:

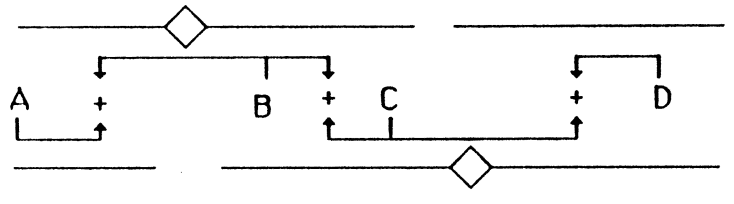


figure 3.15 : exécution de (A+B; B+C; C+D;)

- 3 opérandes sont distincts: (A B) (B C) (C A); nous avons vu qu'une telle

instruction ne peut être exécutée. Démontrons le. Si le chargement des opérandes peut être effectué en un cycle, alors forcément les registres A, B et C doivent être placés sur des sous parties opératives distinctes. Peu importe l'ordre des registres, puisque le problème est symétrique en A, B et C.
Considérons donc le cas où A, B et C sont dans 3 sous parties opératives successives. On ne peut effectuer l'opération C op3 A sur la sous partie opérative contenant B car on ne pourrait pas effectuer les deux autres opérations. Soit on effectue C op3 A dans une sous partie opérative à gauche de B, soit dans une sous partie opérative à droite de B, dans les deux cas on ne modifie pas le problème en supposant que l'on effectue l'opération sur la sous partie opérative contenant A (respectivement C) et non sur une sous partie opérative créée entre A (respectivement C) et B. Comme le problème est symétrique en A et C, on ne considèrera qu'un seul cas. Supposons donc que l'on place l'opérateur op3 dans la sous partie opérative contenant A.

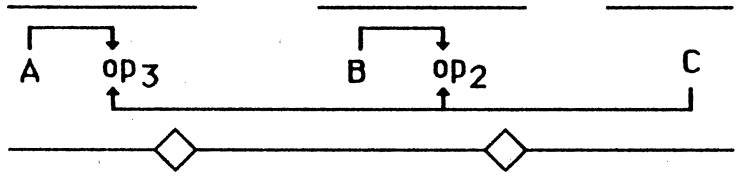


figure 3.16 : exécution partielle de (A+B; B+C; C+A;)

Il est alors impossible d'effectuer l'opération A op1 B, car les trois segments du bus inférieur sont connectés pour transférer le contenu du registre C dans un des tampons d'entrée de l'opérateur op3; seule l'opération B op2 C peut être effectuée simultanément.
On démontre ainsi que ces trois opérations ne peuvent être exécutées simultanément en 2 cycles sur une partie opérative à deux bus segmentables.

Transfert des résultats

Considérons maintenant le problème du transfert des résultats de ces trois opérations binaires. Nous avons vu que les permutations circulaires de trois registres sont les seules instructions à 3 transferts non exécutables en un cycle. Le problème est différent ici car d'une part les registres opérandes doivent respecter certaines contraintes de placement et d'autre part il ne peut y avoir de permutation circulaire puisque les sources qui sont les tampons de sortie des opérateurs sont différentes des destinations.
Si aucune destination n'est aussi opérande d'une des opérations binaires, il n'y a aucun problème. Il suffit de placer les destinations dans les sous parties opératives où se trouvent les tampons de sortie des opérateurs correspondants. S'il n'y a qu'un registre qui est à la fois source et destination, on met les 2 autres destinations dans les sous parties opératives où se trouvent les opérateurs correspondants. On utilise un des deux bus pour effectuer le transfert dans le registre qui est à la fois source et destination et l'autre bus pour effectuer les deux autres transferts, ce qui est toujours

possible car ces 2 derniers transferts se font sur des sous parties opératives distinctes. Le problème apparaît lorsque 2 registres au moins sont à la fois source et destination. Le transfert des 3 résultats n'est pas toujours possible en un cycle.
Exemple:
Soit l'instruction:
(D := A + B; E := B + C; A := C + D;)
Montrons qu'on ne peut pas exécuter cette instruction en 2 cycles et plus exactement que l'on ne peut pas transférer les résultats des opérations au second cycle. Pour cela examinons tous les cas de placement des opérandes. Sur les 4! permutations, seule la moitié est à considérer soit 12 cas; le problème est symétrique dans le sens que si on peut exécuter l'instruction pour un placement donné des registres, on pourra aussi l'exécuter en prenant le placement inverse, c'est à dire si l'on retourne la partie opérative. Sur ces 12 permutations, 4 peuvent être éliminées à cause de la condition nécessaire et suffisante portant sur les couples d'opérandes (A B) et (C D).
A C D B
A D C B
C A B D
C B A D
3 autres cas peuvent être éliminés à cause de la condition suivante: ni A ni D ne doivent appartenir à [A B] ∩ [C D].
A D B C
B C A D
C A D B
Il ne reste donc plus que 5 cas possibles pour le placement des opérandes à savoir:
A B C D
A B D C
B A C D
B A D C
A C B D
Pour ce qui est des 4 premiers cas, on a d'un coté (A B) et de l'autre côté (C D), donc forcément l'opération A + B se fera du côté où se trouvent A et B alors que C + D se fera du côté où se trouvent C et D, et par conséquent l'opération B + C se fera au milieu. Pour effectuer le transfert de A + B dans D et celui de C + D dans A, on devra utiliser les 2 bus et comme l'opération B + C est faite au milieu des 2 autres, on est sûr que l'on ne pourra pas transférer le résultat de B + C, quelle que soit d'ailleurs la destination.
On démontre de la même façon que l'opération B + C doit s'effectuer au milieu des 2 autres pour le dernier cas et donc que l'on ne peut pas exécuter les 3 transferts en un cycle.
Dans tous les cas, la condition:
[+1 D] ∩ [+2 E] ∩ [+3 A] ≠ 0
ne pourra être réalisée car +2 appartient forcément à [+1 D] ∩ [+3 A].
Les opérateurs +1, +2 et +3 désignent respectivement les opérateurs des opérations

A + B, B + C et C + D.
On peut donc affirmer plus généralement que si la condition nécessaire et suffisante pour exécuter le chargement des opérandes de n opérations binaires est vérifiée et si au plus une destination est aussi opérande d'une de ces opérations binaires, alors ces n opérations binaires peuvent être exécutées en 2 cycles.

Instructions comportant n opérations binaires

Plus généralement on peut énoncer le résultat suivant : une condition nécessaire pour que l'on puisse charger simultanément les opérandes de n ($n \geq 3$) opérations binaires dont les 2n opérandes sont tous distincts est que les conditions nécessaires et suffisantes pour tous les triplets distincts d'opérations binaires de ces n opérations soient respectées.
On peut facilement généraliser et montrer que:
(D1 := A1 op1 A2; D2 := A2 op2 A3; ...; Dn := An opn A1;)
ne peut s'exécuter en deux cycles sur une partie opérative à deux bus segmentables.

Nous savons que tout couple d'opérations binaires est exécutable en 2 cycles et nous avons énoncé une condition nécessaire pour exécuter des n-uplets d'opérations binaires mais uniquement en ce qui concerne le transfert des opérandes et dans le cas où tous les opérandes sont distincts.

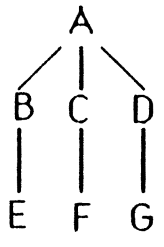
On a vu que si on trouve un placement satisfaisant pour le chargement des opérandes, on ne peut assurer le transfert des résultats que si au plus une destination est aussi opérande d'une des opérations binaires.
Raisonnons maintenant sur les couples d'opérandes des opérations binaires sans s'occuper des destinations.
Déterminons une condition nécessaire pour que l'on puisse charger les opérandes de n opérations binaires. Pour cela considérons le graphe non orienté des opérandes d'opérations binaires où l'on relie les 2 opérandes d'une même opération binaire. On ne considère que les opérations dont les opérandes appartiennent à la même composante connexe puisque les opérations d'une même composante peuvent être effectuées indépendamment des autres.
On peut alors effectuer les hypothèses suivantes :
Soient n couples différents d'opérandes d'opérations binaires:
(A1 A'1) (A2 A'2) ... (An A'n)
On suppose que:
1. {Ai, A'i} ≠ {Aj, A'j} sinon on est ramené à n-1 couples différents,
2. Ai ≠ A'i : on verra tout de suite après pourquoi,
3. Ai ou A'i est opérande d'au moins une autre opération binaire, sinon on peut implanter cette opération de façon indépendante et on est ramené à un problème à n-1 opérations binaires.

C'est pour une raison similaire que l'on impose Ai ≠ A'i, en effet si Ai=A'i on est ramené à une opération unaire dans la mesure où les 2 tampons d'entrée d'un

opérateur binaire peuvent être reliés au même bus. Puisque l'on a imposé la condition 3, il existe forcément une autre opération binaire dont Ai=A'i est opérande; le même bus peut alors être utilisé pour charger Ai dans les tampons des deux opérateurs et on est ramené à un problème à n-1 opérations binaires car il suffit de pouvoir exécuter l'autre opération binaire dont Ai est un des opérandes pour pouvoir exécuter l'opération binaire dont les 2 opérandes sont égaux à Ai.
On suppose par la suite que $n \geq 3$ car on a vu que pour $n \leq 2$, une instruction de ce type est toujours exécutable.
Sur 2n registres; il y a au plus $(2n-2)/2 + 2 = n+1$ registres différents et au moins x avec x vérifiant:

$C^2_x \geq n$ soit $x(x-1)/2 \geq n$
on trouve $x \geq (1+(1+8n)^{1/2})/2$.
soit pour $n=3$ $x \geq 3$ et pour $n=6$ $x \geq 4$.
Montrons que s'il n'y a pas de cycles dans le graphe obtenu en reliant les opérandes d'une même opération binaire, pour toutes les opérations binaires de l'instruction alors on doit avoir au moins n+1 registres différents.
Supposons donc qu'il n'existe pas de cycle; alors il existe forcément un registre A1 qui n'est opérande que d'une seule opération, soit A2 l'opérande auquel il est associé. On a donc un couple (A1 A2) d'opérandes d'une même opération binaire avec 2 registres différents (hypothèse 2).
Supposons donc que l'on ait k couples d'opérandes d'opérations binaires, avec $k<n$ constitués de k+1 registres différents.
Soit Ck = {(A1,A2), (A2,A3), ..., (Ai,Ak+1)} l'ensemble de ces k couples; Ai appartient à Rk.
Soit Rk+1 = {A1,A2, ...,Ak+1} l'ensemble de ces k+1 registres.
Puisque $k<n$ et d'après l'hypothèse 3, il existe forcément un registre Ak+2 qui est opérande d'une opération binaire dont l'autre opérande Aj appartient à Rk+1 et tel que (Aj,Ak+2) n'appartient pas à Ck.
Si Ak+2 appartient à Rk+1, alors il existe une suite de couples de Ck tel que:
(A2,Ab1) ... (Abq,Ak+2)
De même il existe une suite de couples de Ck telle que:
(A2,Ac1) ... (Acr,Aj)
Il existe alors un cycle d'ordre au plus égal à q+r+1 et au moins égal à 3.
Donc s'il n'y a pas de cycle, c'est que Ak+2 n'appartient pas à Rk+1; on démontre ainsi par récurrence qu'il faut au minimum n+1 registres pour qu'il n'y ait pas de cycle dans le graphe des opérandes.
Comme on a montré au départ que le nombre de registres est $\leq$ à n+1, on en déduit qu'il y a exactement n+1 registres. S'il y en a moins, certains opérandes forment un cycle et on ne peut pas exécuter l'instruction.
Cherchons maintenant à dégager une condition nécessaire et suffisante d'exécution de n opérations binaires à n+1 registres différents dans le cas où les hypothèses 1, 2 et 3 sont respectées.
On peut distinguer 2 cas:

- soit aucun registre n'apparaît plus de 2 fois comme opérande; on a alors une suite de couples d'opérandes de la forme suivante:
(A1,A2) (A2,A3) ... (An-1,An) (An,An+1)
Il suffit alors de placer les registres dans l'ordre résultant de l'ordre des registres dans la suite précédente; on met un registre par sous partie opérative; on obtient ainsi n+1 sous parties opératives. Comme l'on ne veut exécuter que n opérations, on peut regrouper 2 sous parties opératives voisines pour diminuer le nombre de sous parties opératives d'une unité.
- soit certains registres peuvent apparaître plus de 2 fois comme opérandes, c'est à dire qu'il existe des groupes de plus de 3 opérations qui ont un opérande commun. Il est nécessaire alors que pour chacun de ces registres qui apparaît au moins trois fois, au plus 2 des registres qui sont opérandes des opérations binaires qui ont en commun cet opérande multiple, figurent dans d'autres opérations binaires.
Contre-exemple:
A + B / A + C / A + D / B + E / C + F / D + G



On peut remarquer que B, C et D jouent des rôles symétriques dans la mesure où E, F et G jouent le même rôle.

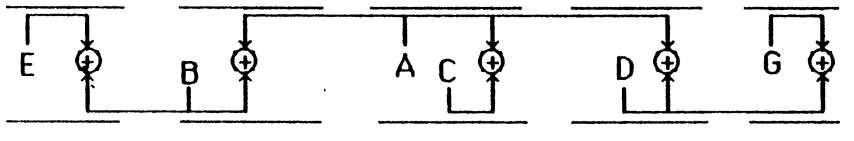


figure 3.17 : exécution partielle de (A+B; A+C; A+D; B+E; C+F; D+G;)

L'opération C+F ne peut être exécutée; peu importe que A et C soient ou non sur la même sous partie opérative. Il est clair que le registre qui sera au milieu du triplet B C D ne pourra jamais être connecté à un autre opérateur que celui où A est connecté.
On peut donc conclure que:
- s'il existe au moins un cycle d'ordre supérieur ou égal à 3 dans le graphe des opérandes,
- ou si un registre est opérande d'au moins 3 opérations et plus de 2 des opérandes associés à ce registre dans ces opérations apparaissent dans d'autres opérations binaires (cas de l'exemple précédent),
alors les opérandes ne peuvent être transférés en un cycle.

3.1.2.2. Séquences d'instructions, non exécutables avec le modèle 2 bus - 2 cycles

Nous avons recensé un certain nombre d'instructions qui seules ne sont pas exécutables en un cycle. Il existe cependant un beaucoup plus grand nombre de cas où une suite d'instructions ne peut être exécutée sur une partie opérative à 2 bus morcelables.
Exemple:
(D1 := A + B; D2 := C + D;)
(D3 := A + C; D4 := D + B;)
(D5 := A + D; D6 := B + C;)

Ces 3 instructions ne peuvent être exécutées successivement sur une partie opérative à 2 bus. En effet, le problème est symétrique en ce qui concerne les opérandes. On a pris toutes les combinaisons de 2 opérations binaires faisant intervenir les 4 registres A, B, C et D comme opérandes.
Comme le problème est symétrique, quel que soit l'ordre des registres opérandes, il existe une instruction pour laquelle la condition nécessaire et suffisante pour exécuter 2 opérations binaires en parallèle n'est pas respectée. Si l'on place les registres opérandes dans l'ordre alphabétique, on ne peut exécuter la dernière instruction puisque B et C appartiennent à l'intervalle [A D].
On peut facilement généraliser à 5 registres.
(D1 := A + B; D2 := C + D;)
(D3 := A + C; D4 := B + E;)
(D5 := A + D; D6 := C + E;)
(D7 := A + E; D8 := B + D;)
(D9 := B + C; D10 := D + E;)

Ces 5 instructions ne peuvent être exécutées successivement pour la même raison que précédemment. En effet, il y a 10 combinaisons de 2 registres différents parmi les 5 registres. Pour que chaque couple apparaisse au moins une fois comme couple d'opérandes d'une opération binaire dans une instruction comportant 2 opérations binaires, il faut qu'il y ait 5 instructions.
On peut généraliser à n registres: il faut n(n-1)/4 instructions si n(n-1)/2 est pair et (n(n-1)/2+1)/2 dans le cas contraire.

3.1.2.3. Effets secondaires du choix du modèle 2 bus - 2 cycles

3.1.2.3.1. Duplication des éléments de la partie opérative

Duplication des constantes

Il peut être nécessaire de dupliquer une constante pour pouvoir exécuter une instruction en 2 cycles avec le modèle choisi.
Exemple:

(A+3; A+B; B+C; C+D; D+3;)
Dans ce cas, la constante 3 doit être dupliquée.

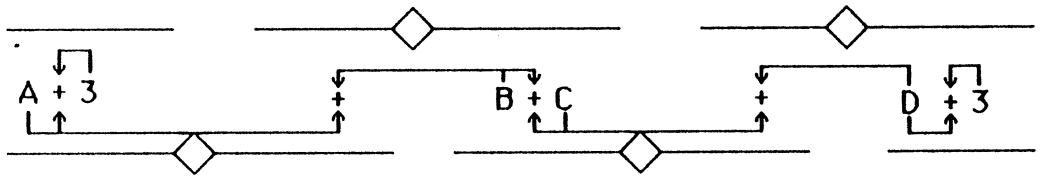
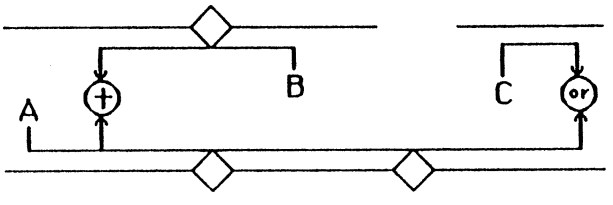


figure 3.18 : duplication des constantes

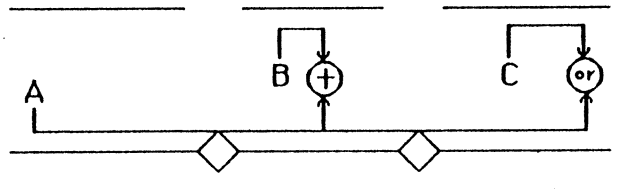
Duplication des opérateurs

Le modèle de partie opérative ne permet pas toujours de connecter directement les opérandes d'une opération aux entrées de l'opérateur où sera exécutée l'opération; il peut être nécessaire de passer par plusieurs segments de bus.
Il se peut qu'il y ait plusieurs opérateurs de même type dans des sous parties opératives distinctes sans que l'on cherche à effectuer les opérations correspondantes en parallèle.
Premier exemple:
Soit le triplet d'instructions suivant:
(D1 := A + B; D2 := A or C;)
(D3 := B + C; D4 := B or A;)
(D5 := C + A; D6 := C or B;)
Chaque instruction comprend 2 opérations binaires en parallèle: + et "or" portant sur le même groupe de registres A, B et C. Dans chacune d'elles l'opérande commun est différent. Les registres A, B et C doivent donc être sur des sous parties opératives distinctes. A, B et C jouent le même rôle dans la mesure où les destinations sont toutes distinctes et différentes des opérandes.
Il suffit de placer A, B et C dans un ordre quelconque, prenons l'ordre alphabétique par exemple.
Recensons les différentes positions possibles des opérateurs + et "or" pour chaque instruction.

1.1



1.2



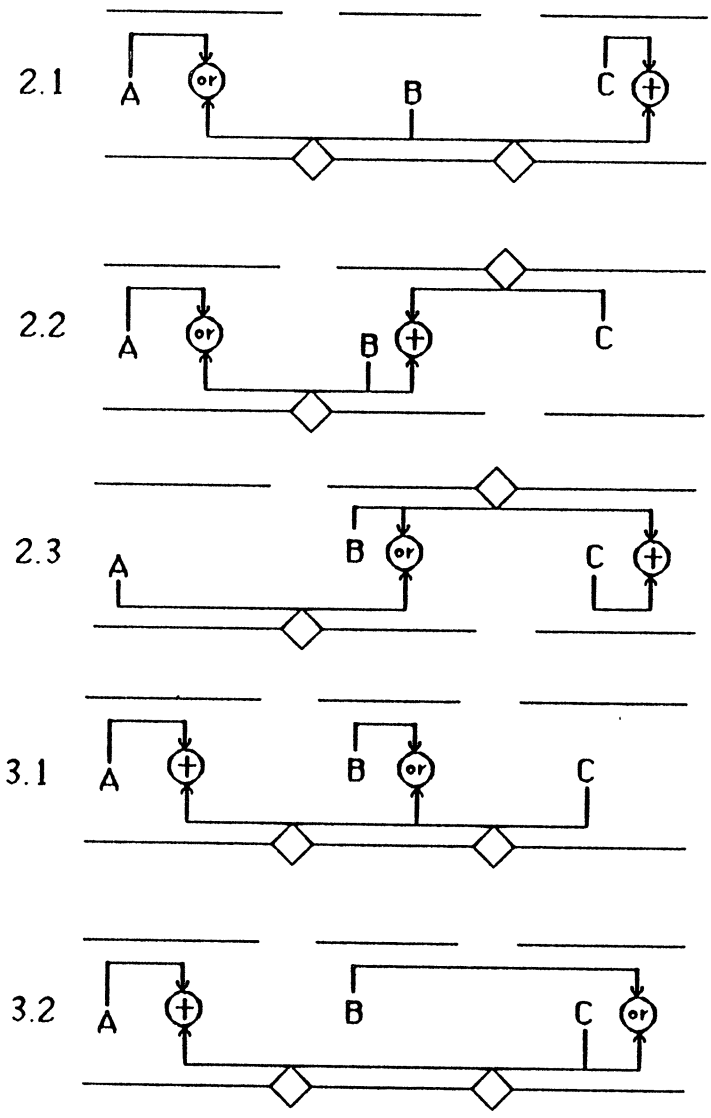


figure 3.19 : différentes alternatives de chargement des opérandes de
(A+B; A or C;)
(B+C; B or A;)
(C+A; C or B;)

On peut combiner ces différents placements en ce qui concerne les opérateurs et obtenir 2*3*2=12 placements satisfaisant aux 3 instructions de départ.
Chacune de ces combinaisons requiert au moins 2 opérateurs "+" et 2 opérateurs "or". La meilleure solution est de choisir la combinaison des différents placements qui minimise le nombre de sous parties opératives où il y a des opérateurs. En effet + et or peuvent être regroupés à l'intérieur d'une ual et partager un certain nombre de transistors.

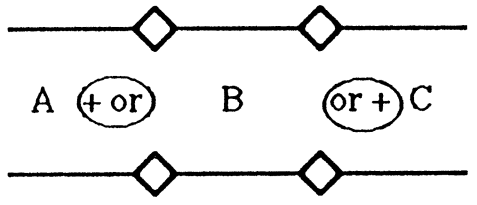


figure 3.20 : duplication des opérateurs + et or

Deuxième exemple:
(A:=X1+X2; B:=Y1+Y2; C:=Z1+Z2;)
(B:=X1+X2; C:=Y1+Y2; A:=Z1+Z2;)
(C:=X1+X2; A:=Y1+Y2; B:=Z1+Z2;)
Cet exemple est constitué de 3 instructions comportant chacune les 3 mêmes opérations binaires mais avec des destinations permutées.
Il est logique de regouper les opérandes d'une même opération binaire puisque la même opération est effectuée 3 fois. Plaçons A, B et C chacun dans une sous partie opérative de telle façon que pour chaque instruction il y ait une opération pour laquelle les opérandes et la destination soient regroupés.

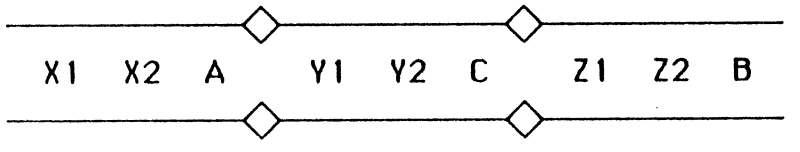
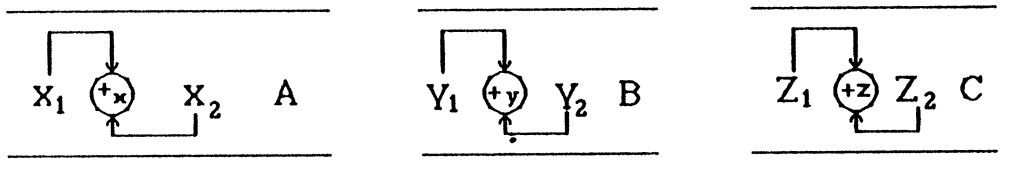


figure 3.21 : exemple de duplication des opérateurs

Ce placement permet de réaliser les instructions 1 et 3 sans problème; par contre l'instruction 2 ne peut être exécutée ainsi. Le placement suivant permet d'exécuter les 3 instructions précédentes.



(A : = X_1 + X_2 ; B : = Y_1 + Y_2 ; C : = Z_1 + Z_2 ;)

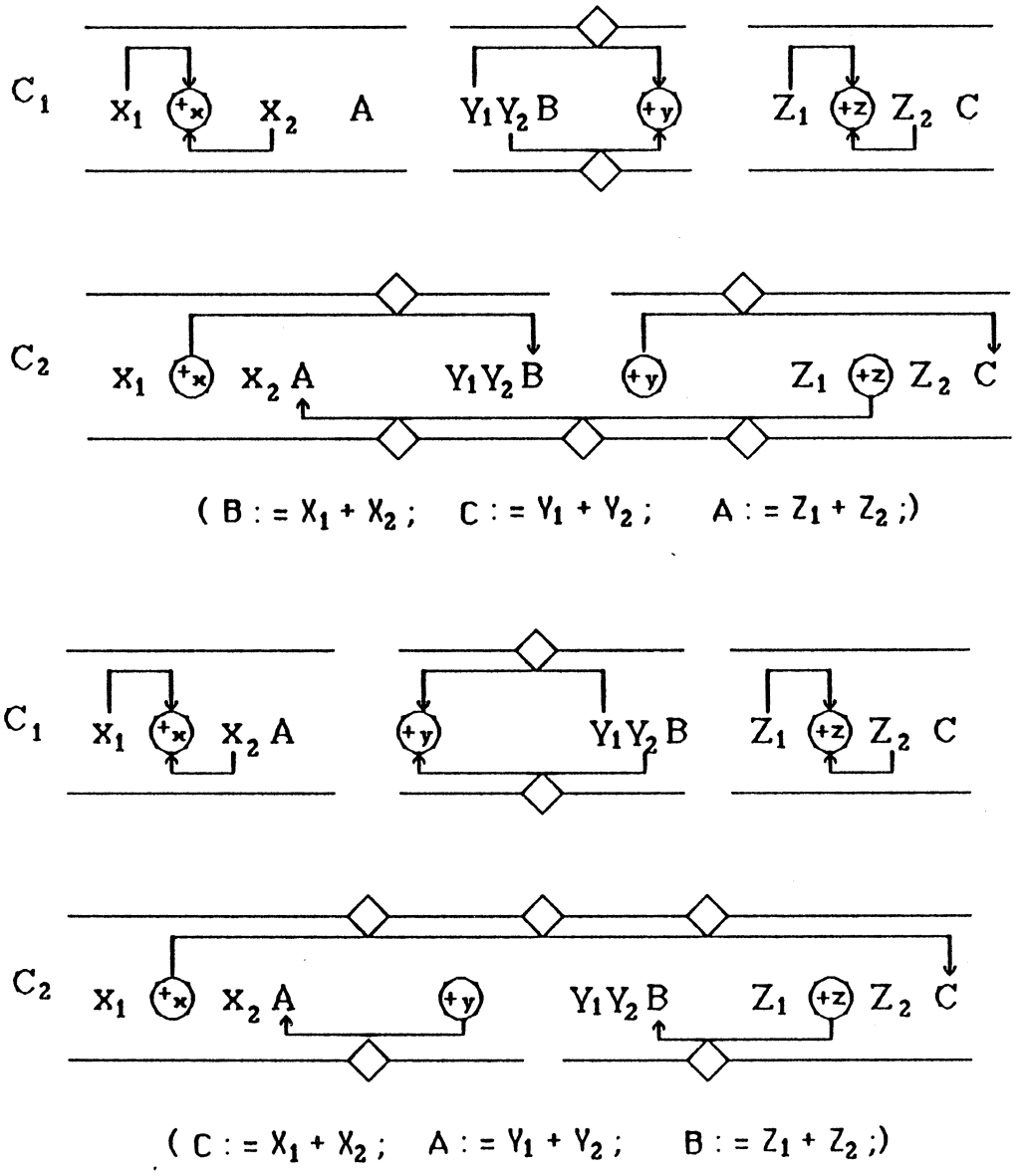


figure 3.22 : exécution de 3 instructions comportant les 3 mêmes opérations binaires mais dont les destinations sont permutées

Les symboles +x, +y et +z désignent les opérateurs utilisés pour effectuer les opéations X1+X2, Y1+Y2 et Z1+Z2.
On peut combiner les placements des opérateurs pour minimiser le nombre d'additionneurs:

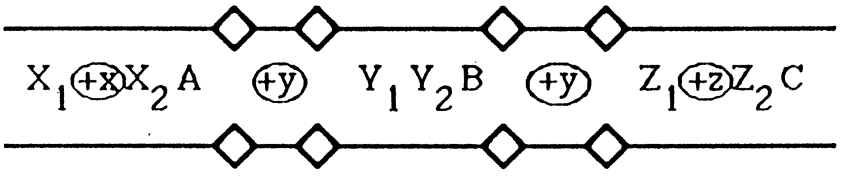
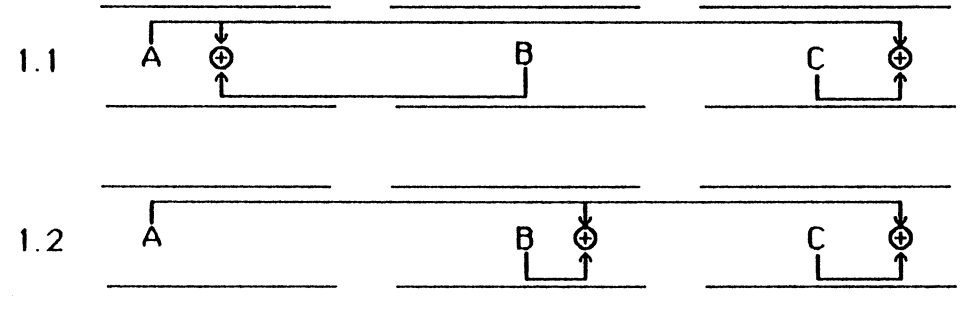


figure 3.23 : ajout d'un quatrième additionneur

On doit utiliser 5 sous parties opératives et 4 additionneurs alors que l'on n'effectue jamais plus de 3 additions en parallèle.
Remarque:
Si on rend le problème complètement symétrique en A, B et C, il devient impossible à résoudre en 2 cycles avec 2 bus.
Pour symétriser le problème on est obligé de rajouter 3 autres instructions.
(A:=X1+X2; B:=Y1+Y2; C:=Z1+Z2;)
(A:=X1+X2; C:=Y1+Y2; B:=Z1+Z2;)
(B:=X1+X2; C:=Y1+Y2; A:=Z1+Z2;)
(B:=X1+X2; A:=Y1+Y2; C:=Z1+Z2;)
(C:=X1+X2; A:=Y1+Y2; B:=Z1+Z2;)
(C:=X1+X2; B:=Y1+Y2; A:=Z1+Z2;)

3.1.2.3.2. Variation d'exécution d'une action opérative

Dans certains cas, une même action opérative doit être exécutée de façon différente selon l'instruction où elle apparaît.
Exemple:
Soient les 3 instructions suivantes:
(D1 := A + B; D2 := A + C;)
(D2 := A + C; D3 := B + C;)
(D3 := B + C; D1 := A + B;)
Remarquons tout d'abord que A, B et C doivent se trouver sur des sous parties opératives distinctes et que l'ordre importe peu puisque le problème est symétrique en A, B et C. On suppose que les destinations sont différentes de A, B et C : le placement de ces destinations ne pose alors pas de problème.
L'instruction 1 peut s'exécuter de 2 façons différentes en ce qui concerne la place des opérateurs.



De même l'instruction 2 peut s'exécuter de 2 façons différentes en ce qui concerne la place des opérateurs.

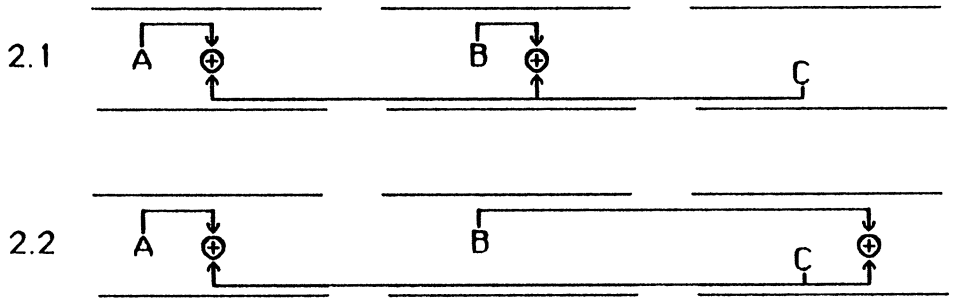


figure 3.24 : l'exécution d'une action dépend du contexte

On peut remarquer que l'opération D2:=A+C doit se faire dans la troisième sous partie opérative pour l'instruction 1 et dans la première pour l'instruction 2.
En conclusion on peut dire que l'exécution d'une action opérative dépend du contexte dans lequel elle est exécutée. On ne peut donc pas parler d'action opérative en général mais de l'action opérative d'une certaine instruction.

3.1.2.3.3. Augmentation du nombre de segments des bus

Le nombre de sous parties opératives n'est pas forcément égal au nombre maximum d'opérations en parallèle ou à la moitié du nombre maximum de transferts en parallèle dans le cas d'instructions ne comportant que des transferts. Il peut être nécessaire d'augmenter le nombre de segments des bus.
Exemple:
Soient les 2 instructions suivantes:
(D11 := A op11 B; D12 := C op12 D;)
(D21 := B op21 C; D22 := D op22 E;)
Les registres B, C et D sont opérandes des opérations binaires des deux instructions. Dans la première instruction, C et D sont opérandes de la même opération binaire alors que dans la deuxième, ce sont B et C qui sont opérandes d'une même opération.
Montrons qu'il faut au moins 3 sous parties opératives pour exécuter ces 2 instructions.
S'il n'y a que 2 sous parties opératives, il faut mettre 3 des registres opérandes dans une même sous partie opérative et les 2 autres opérandes dans l'autre sous partie opérative.
Parmi les 3 registres qui sont ensemble, il doit y avoir A et E sinon 3 opérandes d'une même instruction seraient réunis.
Examinons les 3 partitions possibles:
A B E - C D
A C E - D B
A D E - B C
Aucune de ces partitions ne convient car à chaque fois une des deux conditions nécessaires et suffisantes pour exécuter ces instructions n'est pas satisfaite.
Il faut donc 3 sous parties opératives, comme A et B d'une part, et D et E d'autre

part jouent un rôle symétrique par rapport à C, on place C au milieu.

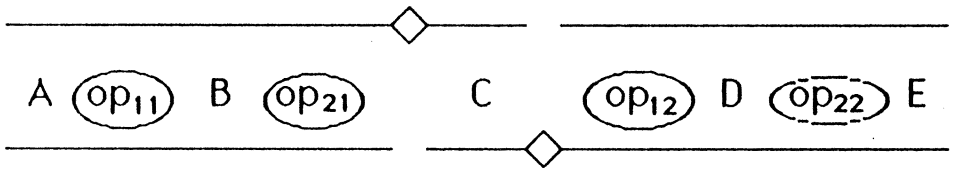


figure 3.25 : augmentation du nombre de segments

3.1.3. Autres modèles de parties opératives

Considérons maintenant d'autres modèles dérivant du modèle de partie opérative à 2 bus adopté par le compilateur Apollon.
On peut modifier ce modèle:
- soit en modifiant la structure de bus:
.1 en introduisant des coupures sur un bus mais pas forcément sur les 2 au même niveau; ce modèle n'est en fait qu'une variante du modèle des 2 bus segmentés d'Apollon,
.2 en diminuant d'un le nombre des bus ou en ajoutant un ou plusieurs bus aux 2 bus du modèle de base,
.3 en utilisant des bus non parallèles,
- soit en modifiant le séquencement des instructions:
.1 en augmentant le nombre de cycles d'exécution des instructions opératives,
.2 en permettant au concepteur de décrire les instructions opératives à un niveau plus fin.

3.1.3.1. Autres structures de bus

3.1.3.1.1. Segmentation non uniforme des 2 bus

Dans le cadre du modèle présenté précédemment, les 2 bus peuvent être morcelés mais au même endroit pour les 2 bus. Il peut être intéressant de découper les 2 bus de façon différente.
Par exemple, les 2 instructions présentées précédemment:
(D11 := A op11 B; D12 := C op12 D;)
(D21 := B op21 C; D22 := D op22 E;)
peuvent être exécutées par une partie opérative ne comportant pas d'interrupteur de bus.

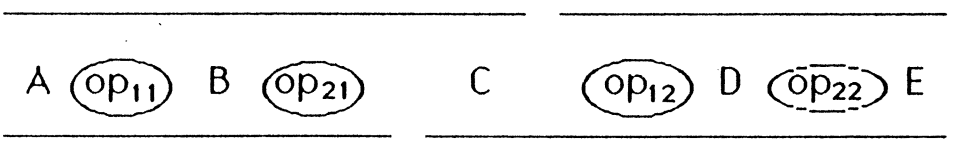
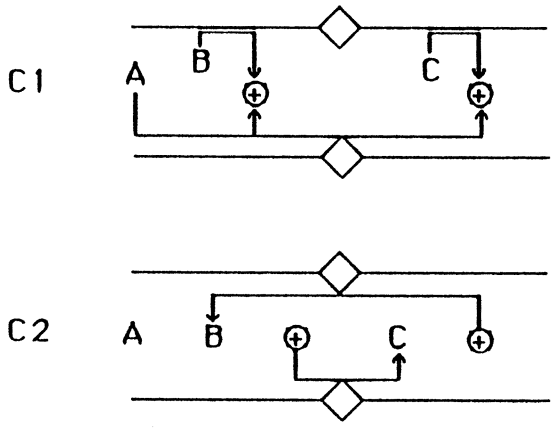


figure 3.26 : segmentation non uniforme des 2 bus

Autre exemple:
(C := A + B; B := A + C;)
Bus identiques:
Découpage uniforme des bus



Bus segmentés de façon différente:
Bus dissymétriques

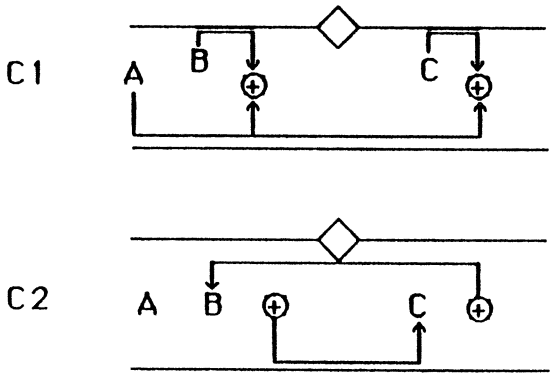
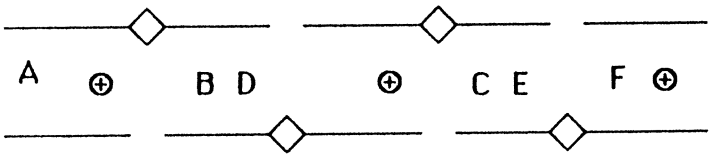


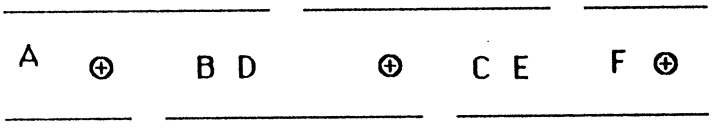
figure 3.27 : gain d'un interrupteur

Si l'on segmente les 2 bus de façon différente, on diminue le nombre des interrupteurs.
Deux segments peuvent être court-circuités s'ils ne sont jamais utilisés simultanément pour effectuer 2 transferts ayant des sources différentes, qu'un seul de ces segments soit utilisé ou qu'ils soient connectés pour exécuter un même transfert.
Donnons un dernier exemple où l'on peut diminuer le nombre des interrupteurs de 4.
(D1 := A + B; D2 := C + D; D3 := E + F;)

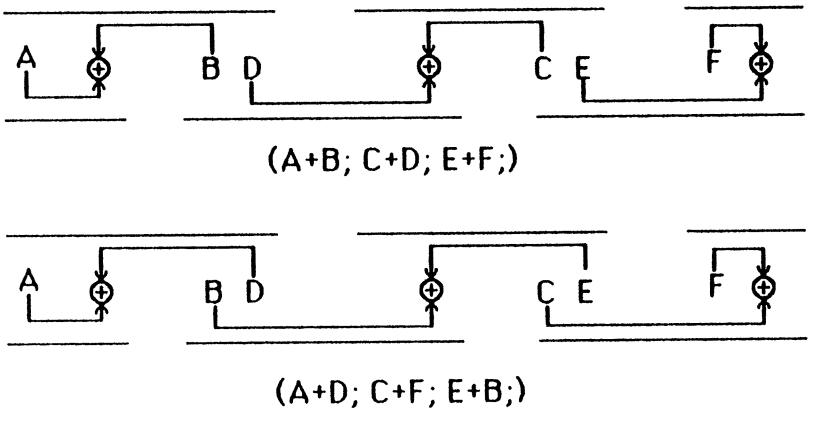
(D4 := A + D; D5 := C + F; D6 := E + B;)



segmentation identique des 2 bus



segmentation non uniforme des bus



chargement des opérandes

figure 3.28 : gain de 4 interrupteurs

3.1.3.1.2. Parties opératives à bus parallèles

Il peut être intéressant de générer des parties opératives avec un nombre variable de bus; on peut ainsi adapter la structure de la partie opérative aux besoins du concepteur.

Parties opératives à un bus morcelable

La partie opérative ne possède qu'un seul bus qui peut être segmenté de telle façon que l'on puisse exécuter des transferts en parallèle sur les différents segments.
Analysons les possibilités de parallélisme offertes par ce modèle.
Il n'est pas nécessaire de supposer que l'on se limite aux transferts et aux opérations

unaires. On peut aussi effectuer des opérations binaires en 2 cycles à condition de placer l'opérateur correspondant entre 2 segments.
On peut ainsi charger les 2 opérandes en un cycle. Le transfert du résultat ne pose pas de problème puisque l'on n'a besoin que d'un bus.

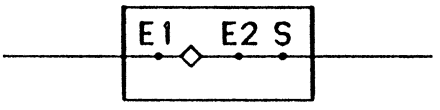


figure 3.29 : opérateur connecté à deux segments du bus

Ce modèle ne permet pas en général d'exécuter d'instructions où un même registre est à la fois source et destination. En effet pour cela il faut à la fois que les horloges de lecture et d'écriture soient non recouvrantes mais aussi que le registre soit à double accès ce qui n'est possible dans ce cas que si on place un registre entre 2 segments.

Déterminons le ou les conditions nécessaires et suffisantes pour exécuter simultanément 2 transferts:
Par hypothèse ces 2 transferts ne peuvent avoir la même destination. Deux cas sont à distinguer:
- soit ces 2 transferts ont même source: il n'y en fait qu'un seul transfert donc un seul bus suffit,
- soit les 2 sources sont distinctes: il faut et il suffit que l'intersection des segments occupés par les 2 transferts soit vide.

C'est à dire que si l'on veut exécuter (A:=B; C:=D;), il faut et il suffit que [A B] $\wedge$ [C D] = 0.
Il est donc facile de trouver des combinaisons d'instructions que l'on ne peut exécuter en un cycle même si chacune des instructions l'est, par exemple:
(A:=B; C:=D;)
(A:=D; C:=B);

Modèle à 3 bus

Si l'on n'effectue que des opérations unaires ou binaires, il ne semble pas à première vue utile d'augmenter le nombre de bus par sous partie opérative. Pour augmenter le parallélisme d'exécution, il suffit d'augmenter le nombre de sous parties opératives. On augmentera ainsi le nombre de bus.
On a vu précédemment que certaines instructions ou certaines combinaisons d'instructions ne peuvent être exécutées sur une partie opérative à 2 bus segmentables en 2 cycles opératifs. Dans ce cas, il est nécessaire d'augmenter le nombre de bus par partie opérative si l'on veut pouvoir exécuter ces instructions en 2 cycles.
D'autre part, il est nécessaire d'introduire un bus supplémentaire si l'on veut pipeliner des opérations binaires où si l'on désire effectuer rapidement des opérations ternaires (n-aires) sans les décomposer en une suite d'instructions (log2 n arrondi à l'unité supérieur).

Nous nous intéresserons ici uniquement aux parties opératives à 3 bus pouvant être segmentés.
Considérons d'abord les instructions ne comportant que des transferts ou des opérations unaires ou binaires.
L'ajout d'un troisième bus permet d'exécuter l'instruction du paragraphe 3.1.2 en 2 cycles :
(D1 := A + B; D2 := B + C; D3 := C + A;)

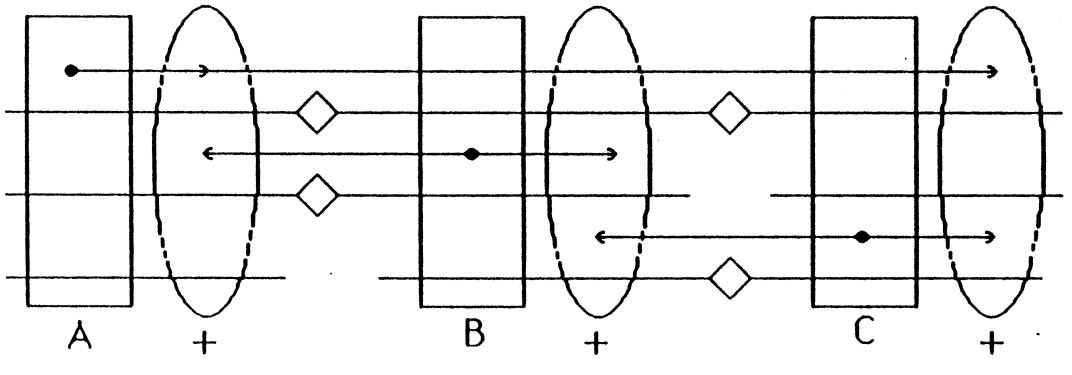
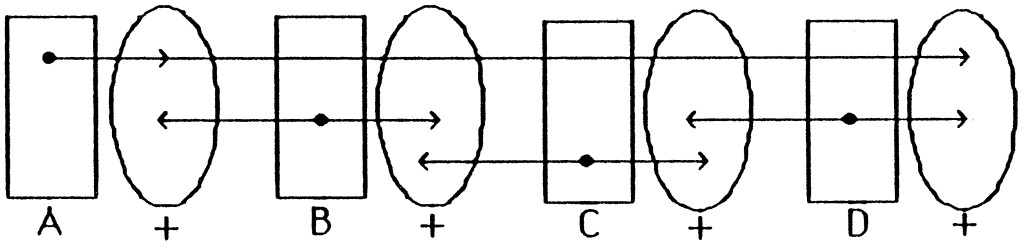


figure 3.30 : exécution de (A+B; B+C; C+A;) sur une partie opérative à 3 bus

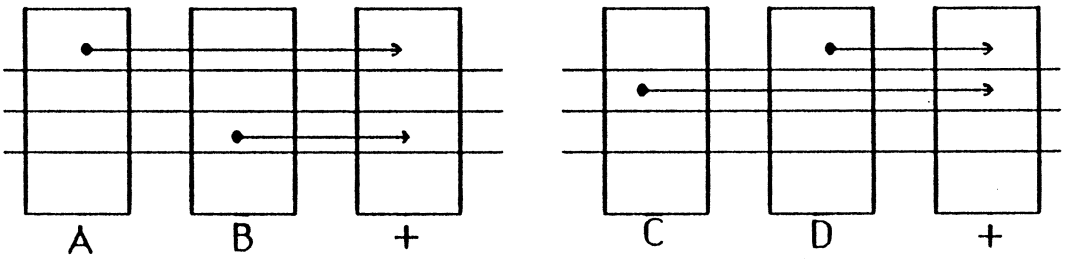
Les destinations ne sont pas placées car leur placement ne pose pas de problème.
On peut de la même façon exécuter une instruction comportant n opérations binaires dont les opérandes forment un cycle d'ordre n dans le graphe des opérandes.



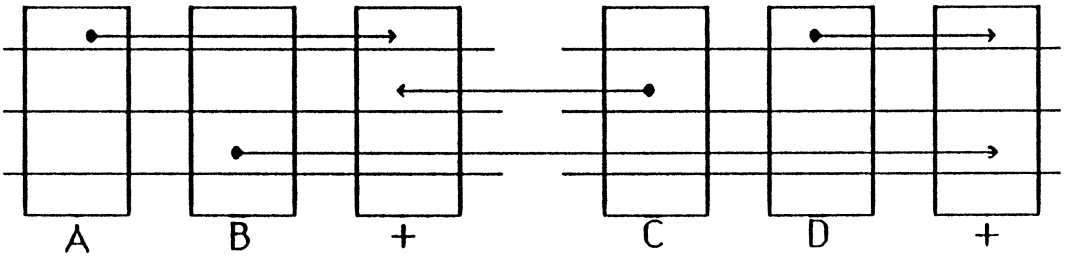
(A+B ; B+C ; C+D ; D+A ;)

figure 3.31 : exécution de (A+B; B+C; C+D; D+A;) sur une partie opérative à 3 bus

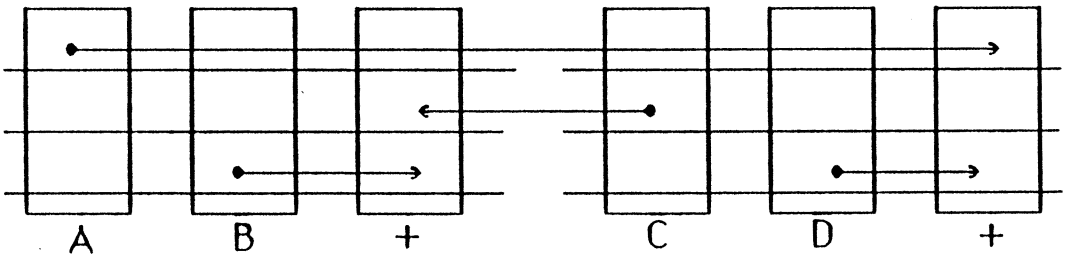
Autre exemple:
(D1 := A + B; D2 := C + D;)
(D3 := A + C; D4 := D + B;)
(D5 := A + D; D6 := B + C;)



(A+B; C+D;)



(A+C; D+B;)



(A+D; B+C;)

figure 3.32 : exécution de (A+B;C+D) (A+C;B+D) (A+D;B+C) sur une partie opérative à 3 bus

On peut aussi exécuter une permutation circulaire sur A, B et C.
(A := B; B := C; C := A;)

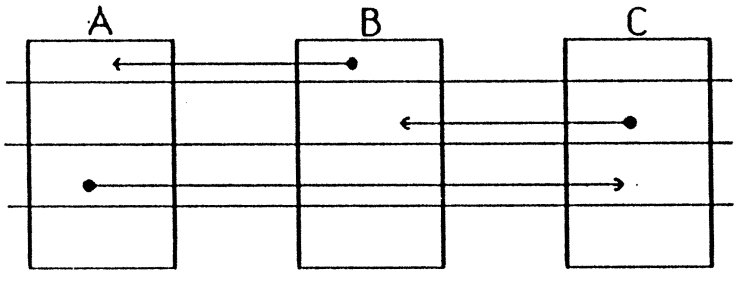
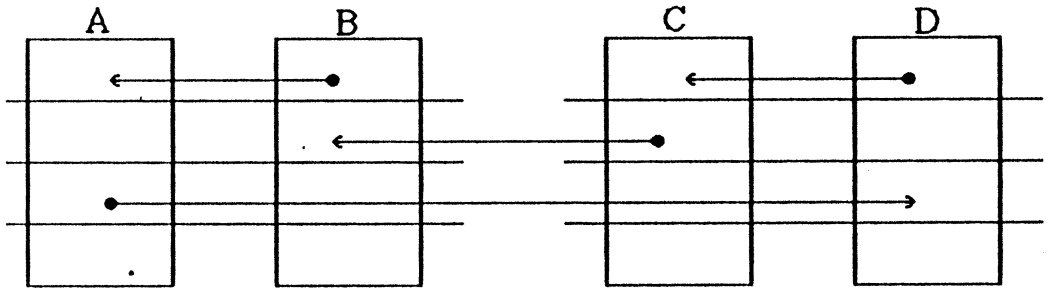


figure 3.33 : permutation de 3 registres

On peut généraliser et on montre facilement que l'on peut effectuer une permutation circulaire quelconque.



Permutation circulaire de 4 registres

figure 3.34

D'autre part, l'ajout d'un troisième bus permet de pipeliner les opérations binaires: on dispose d'un troisième bus pour transférer le résultat en même temps que les opérandes de la prochaine opération à exécuter.

Il existe cependant des combinaisons d'instructions non exécutables sur une partie opérative à 3 bus. Prenons des opérations binaires et intéressons nous aux opérandes de ces opérations.
Soient 6 registres A, B, C, D, E et F.
Constituons tous les triplets de paires de registres différents. Il y en a $(C^2_6.C^2_4.C^2_2)/3!=15$.

(A + B; C + D; E + F;)
(A + B; C + E; F + D;)
(A + B; C + F; D + E;)

(A + C; B + D; E + F;)
(A + C; B + E; F + D;)
(A + C; B + F; D + E;)
(A + D; B + C; E + F;)
(A + D; B + E; F + C;)
(A + D; B + F; C + E;)
(A + E; B + C; D + F;)
(A + E; B + D; F + C;)
(A + E; B + F; C + D;)
(A + F; B + C; D + E;)
(A + F; B + D; E + C;)
(A + F; B + E; C + D;)

Puisque l'on a pris soin de rendre symétrique le problème en A, B, C, D, E et F, il suffit de ne considérer qu'un placement pour ces 6 registres, par exemple:
A B C D E F
L'instruction: (A + F; B + E; C + D;) n'est pas exécutable en 2 cycles.

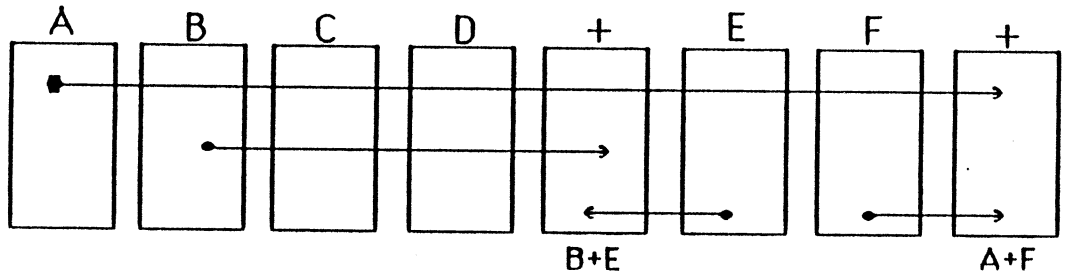


figure 3.35 : chargement des opérandes de (A+F; B + E; C + D;)

Dans l'exemple présenté ici, il manque encore un bus pour effectuer l'opération C + D.
Si le modèle à 3 bus présente des possibilités de parallélisme bien supérieures, il existe des groupes d'instructions non exécutables à l'aide d'une partie opérative de ce type.
On peut montrer que quel que soit n, il existe des groupes d'instructions à n opérations binaires non exécutables sur une partie opérative à n bus morcelables.
Si l'on utilise des opérateurs ternaires, on peut trouver des instructions qui bloquent le modèle à 3 bus.
Exemple:
(A + B + D; B + C + D; C + A + D;)
D est commun aux trois opérations et les trois autres opérandes forment un cycle dans le graphe des opérandes obtenu en reliant les opérandes d'une même opération.
Cette instruction a été obtenue en rajoutant un troisième opérande aux opérations de l'instruction à 3 instructions binaires dont les opérandes forment un cycle, qui n'est pas exécutable en 2 cycles sur une partie opérative à 2 bus.

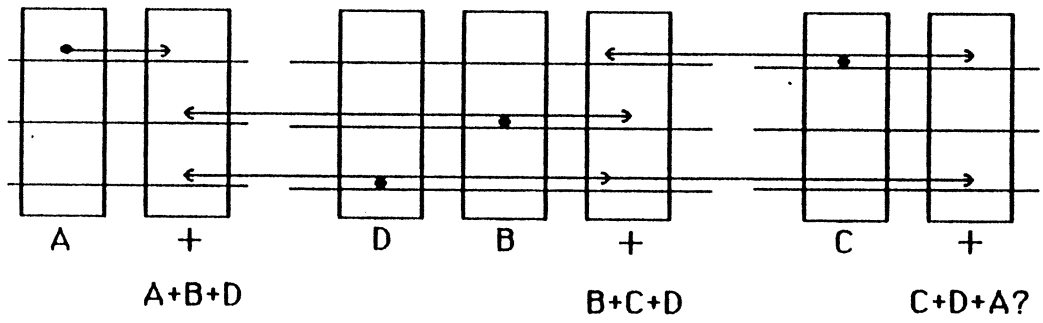


figure 3.36 : chargement des opérandes de (A+B+D;B+C+D;C+A+D;)

Dans l'exemple ci-dessus, il manque un bus pour amener A pour effectuer l'opération C + A + D.
On généralise facilement pour un modèle à n bus. Il suffit de prendre 3 opérations n-aires faisant intervenir n+1 opérandes distincts. n-2 opérandes sont communs aux 3 opérations. Les 3 autres forment un cycle d'ordre 3.
Exemple:
(A+B+D1+D2+ ... +Dn-2; B+C+D1+D2+ ... +Dn-2; C+A+D1+D2+ ... +Dn-2;)

Généralement parlant, l'addition de bus supplémentaires augmente le nombre de transferts entre registres que l'on peut effectuer simultanément. Cependant l'introduction de nouveaux bus ne peut conduire à une augmentation significative des performances car il y a peu d'opérateurs n-aires disponibles pour n supérieur à 3 et on ne peut exploiter au mieux les possibilités ouvertes par l'ajout de bus supplémentaires.

3.1.3.1.3. Modèle à bus non parallèles

Les sous parties opératives ne sont plus alignées, mais empilées les unes au dessus des autres [RAMA 85]. Un canal vertical placé sur un des côtés de ces sous parties opératives permet de les connecter entre elles. On dispose pour cela d'un nombre quelconque de bus. Les sous parties opératives peuvent avoir un nombre variable de bus.
Dans la suite, on supposera que toutes les sous parties opératives ont le même nombre de bus et que celui est égal à 2.

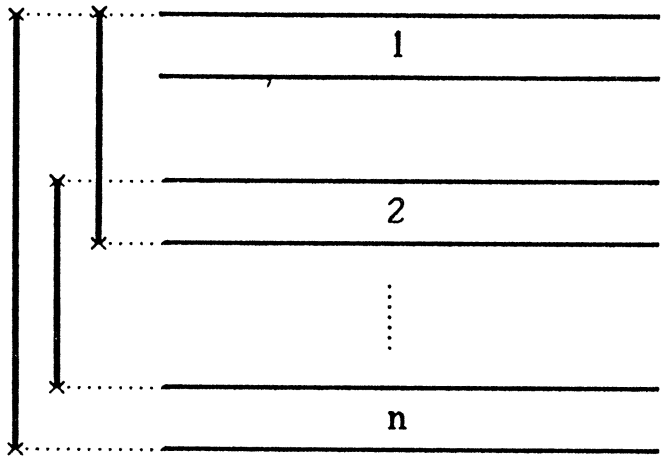


figure 3.37 : sous parties opératives empilées

Comme on dispose d'un nombre non limité de bus d'interconnexion, on peut relier directement n'importe quel couple de sous parties opératives.
On montre facilement que l'on peut exécuter n'importe quelle instruction sur une partie opérative constituée de cette façon.
Il suffit de prendre une sous partie opérative par registre. Rappelons qu'un registre est accessible en lecture et en écriture simultanément. On peut donc avoir au maximum pour n registres différents $A^2_n=n(n-1)$ transferts possibles (et non $C^2_n=n(n-1)/2$ car un transfert a un sens).
De ces A^2_n transferts, seul un certain nombre sont possibles simultanément : il faut en effet que les destinations soient différentes. On a donc forcément au plus n transferts en parallèle si un registre peut être lu et écrit au cours du même cycle. Il est donc nécessaire d'avoir n bus. Or si on a n sous parties opératives à 2 bus, on dispose de 2n bus et si on a le pire cas (une permutation portant sur n registres), il faut connecter les 2n bus 2 par 2 de sorte que l'on obtient finalement n bus, ce qui permet d'exécuter les n transferts en parallèle.
Pour que la démonstration soit complète, il faut traiter le cas du chargement des opérandes d'une opération binaire. En effet les tampons d'entrée d'un opérateur sont en général regroupés; il s'agit alors d'effectuer 2 transferts dont les destinations sont sur la même sous partie opérative. Il suffit de placer chacun des opérateurs seul sur une sous partie opérative.
On démontre ainsi que l'on peut exécuter n'importe quelle groupe d'instructions comportant des transferts et des opérations unaires, ..., n-aires où n est le nombre de bus d'une sous partie opérative.

3.1.3.2. Autres modèles de séquencement

3.1.3.2.1. Exécution des instructions opératives en un nombre variable de cycles

Nous avons considéré jusqu'à maintenant que les instructions s'exécutaient en 2 cycles ou en un cycle s'il n'y avait que des transferts.
Or ce nombre de cycles n'est pas toujours suffisant. Une permutation de plus de 2 registres nécessite 2 cycles. Certaines instructions nécessitent 3 cycles.
Exemple:
L'instruction
(D1 := A + B; D2 := B + C; D3 := C + A;)
peut être exécutée en 3 cycles sur une partie opérative à 2 additionneurs.

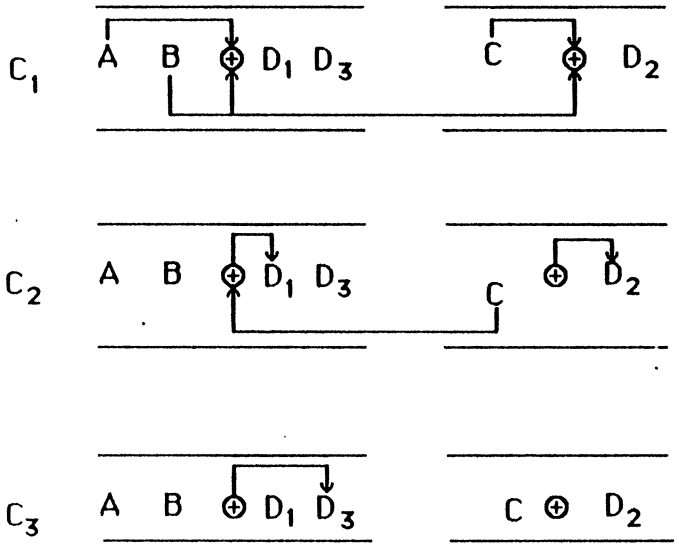


figure 3.38 : exécution de (D1:=A+B;D2:=B+C;D3:=C+A;) en 3 cycles

Au premier cycle les opérandes des opérations A+B et B+C sont chargés. Au deuxième cycle les résultats de ces 2 opérations sont transférés dans leurs destinations respectives tandis que C est chargé dans le tampon d'entrée du premier additionneur qui contenait B, l'autre tampon contenant déjà A.
Enfin au troisième cycle, on transfert le résultat de A+C dans D3.
Cet exemple montre qu'il peut être utile d'avoir un séquencement des instructions en un nombre de cycles qui varie avec la complexité des instructions. Une autre solution aurait pu être de découper une instruction non exécutable en 2 cycles, en 2 morceaux et d'essayer d'implanter chacun des morceaux de l'instruction séparément. Dans ce cas on a besoin de 4 cycles pour exécuter l'instruction précédente. Le modèle de séquencement en un nombre variable de cycles est plus avantageux qu'un simple découpage des instructions dans le cas d'instuctions non exécutables en 2 cycles.

3.1.3.2.2. Autre niveau de description des instructions

Si l'on désire pipeliner les opérations, il peut être utile de décrire les actions opératives de façon plus fine.
Pour pouvoir faire fonctionner les UALs sans arrêt, il est nécessaire de donner la possibilité au concepteur de spécifier séparément le chargement des opérandes et la récupération des résultats.
Pour cela on introduit 2 nouveaux types d'actions opératives:
- le chargement des opérandes dans les tampons d'entrée de l'opérateur,
- le transfert du résultat dans le registre destination.

Soient in-alu(op1,+,op2,d) et out-alu(op1,+,op2,d).
In-alu(op1,+,op2,d) signifie charger les opérandes op1 et op2 dans les tampons de l'ual utilisée pour effectuer l'opération d:=op1+op2. Out-alu(op1,+,op2,d) signifie transférer le résultat de l'opération op1+op2 dans d. Les fonctions in-alu et out-alu ne peuvent être spécifiées sous cette forme que si une opération est exécutée de façon identique quelle que soit l'instruction où elle apparaît.
Si ce n'est pas le cas, il est nécessaire d'introduire un paramètre de liaison.
Par exemple:
in-alu(op1,+,op2,d,1);
out-alu(d,1);
Le paramètre de liaison "1" permet d'exprimer qu'il s'agit des entrées et de la sortie de la même UAL. La fonction in-alu a aussi "d" comme paramètre pour permettre au compilateur de choisir l'additionneur utilisé pour l'opération non pas seulement en fonction des opérandes mais aussi en fonction de la destination. Comme les instructions opératives extraites de la description algorithmique de départ ne sont pas ordonnées, il faut que le compilateur puisse consulter une table de correspondance entre paramètres de liaison et actions opératives pour que l'on puisse omettre opérandes et opérateur dans la fonction out-alu.
Exemple:
(D1:=S11+S12; in-alu(S21,+,S22,D2,1);)
(out-alu(D2,1); D3:=S31+S32; in-alu(S41,+,S42,D4,2);)
(out-alu(D4,2); D5:=S51+S52; in-alu(S61,+,S62,D6,3);)
Il est possible d'exécuter 2n-1 opérations binaires en 2n cycles opératifs sur une sous partie opérative à 3 bus ou sur une sous partie opérative à 2 bus dont les entrées de l'opérateur sont connectées à 3 segments différents alors que si l'on n'introduit pas les actions in-alu et out-alu on ne peut faire que n opérations binaires en 2n cycles opératifs. On peut ainsi doubler la fréquence d'utilisation des UALs de chaque sous partie opérative.
On est en fait obligé d'utiliser un paramètre de liaison si l'on veut profiter au maximum des possibilités de parallélisme du modèle.
En effet l'exécution d'une instruction dépend du contexte dans lequel elle doit être exécutée (cf 3.1.2).
Une autre solution consiste à implanter les instructions de façon unique et à découper une instruction en au moins 2 morceaux si nécessaire.
En ce qui concerne le traitement du paramètre de liaison, 2 stratégies sont possibles:
- soit c'est le concepteur qui spécifie que 2 opérations doivent s'effectuer sur la

même sous partie opérative en donnant des valeurs identiques au paramètre de liaison,
- soit c'est le compilateur qui détermine la sous partie opérative où s'effectue l'opération et les valeurs données au paramètre de liaison doivent être toutes distinctes.

3.1.3.2.3. Fonctionnement en pipe-line

Il s'agit de maintenir constamment utilisées les uals de la partie opérative. Le modèle à 2 bus ne permet pas en général un tel fonctionnement. En effet au premier cycle, les opérandes sont chargés dans les registres tampons de l'ual. Au deuxième cycle, le résultat est transféré dans le registre destination. Il manque donc un bus pour transférer un second opérande au deuxième cycle sauf s'il y a au moins un opérande commun.
Un fonctionnement en pipe-line est par contre possible si l'on dispose de 3 bus. Au deuxième cycle, on pourra effectivement transférer le résultat dans le registre destination et charger les nouveaux opérandes.

Il est possible d'augmenter la fréquence d'utilisation des uals pour une partie opérative à 2 bus, soit en utilisant un tampon maître-esclave en sortie de chaque UAL, soit en plaçant entrées et sortie de chaque UAL sur des sous parties opératives voisines.

[HANS 84] propose de mettre un tampon maître-esclave à la sortie de chaque UAL et de modifier le séquencement d'une opération binaire.
Exemple: soit l'instruction :
(d1:=a1 op1 b1; d2:=a2 op2 b2;)
Pour exécuter ces 2 opérations binaires sur une seule sous partie opérative à 2 bus, il faut au moins 3 cycles puisque l'on ne peut pas effectuer les 2 opérations en même temps si (a1,b1) $\neq$ (a2,b2).
L'utilisation d'un tampon maître-esclave en sortie permet de retarder le transfert du premier résultat jusqu'au troisième cycle et ainsi de pouvoir utiliser les 2 bus de libre pendant le deuxième cycle pour charger les opérandes de la seconde opération.
Donnons le séquencement de ces 2 opérations. Chaque cycle est divisé en 4 phases:
1.précharge
2.lecture
3.amplification
4.écriture

cycle1.4: écriture de a1 et b1 dans les tampons d'entrée de l'UAL
cycle2.1: chargement du résultat r1=a1 op1 b1 dans le maître du tampon de sortie
cycle2.4: écriture de a2 et b2 dans les tampons d'entrée de l'UAL; d'autre part le premier résultat doit être transféré dans l'esclave entre les phases cycle2.1 et cycle3.1

cycle3.1: chargement du résultat r2=a2 op2 b2 dans le maître du tampon de sortie
cycle3.2 à cycle3.4: transfert de r1 et r2 dans leurs destinations respectives.
On peut donc conclure qu'en allongeant le temps d'exécution d'une instruction à 3 cycles opératifs et en utilisant des maîtres-esclaves en sortie des UALs on peut augmenter d'un tiers la fréquence d'utilisation des UALs.
En effet en 6 cycles, on peut effectuer sur chaque UAL 3 opérations binaires avec un tampon de sortie de type bistable et 4 avec un maître-esclave en sortie des UALs.

Une façon plus astucieuse d'augmenter la fréquence de fonctionnement des UALs est de placer les entrées et la sortie de chaque UAL sur des sous parties opératives voisines.
Les entrées et la sortie sont connectées à l'UAL elle même par des bus de service et non pas par les 2 bus servant au transfert des données de registre à registre (cf section 4.2). On peut donc placer entrées et sortie sur des sous parties opératives voisines, il suffit alors de prolonger un des bus de service pour connecter le tampon à l'UAL. Comme il y a 3 points de connexion aux bus, 2 répartitions sont possibles:
- soit les 2 entrées sont ensemble,
- soit la deuxième entrée et la sortie sont ensemble.

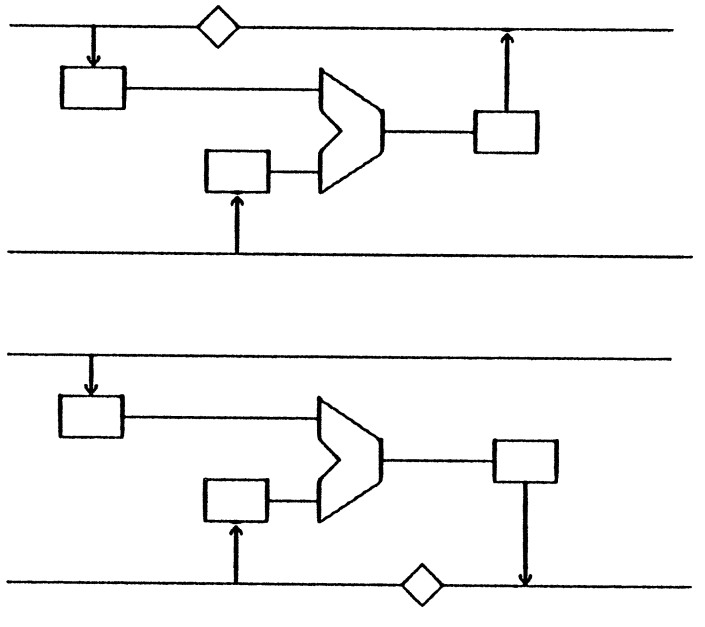


figure 3.39 : opérateurs connectés à 3 segments

Il n'y a d'ailleurs pas besoin de morceler les 2 bus, il suffit de morceler un seul des 2 bus. On obtient ainsi les 3 segments qui sont nécessaires à l'exécution du transfert du résultat et du chargement des nouveaux opérandes.
Comme pour le modèle à 3 bus, on peut doubler la fréquence d'utilisation des UALs.

Conclusion: on peut augmenter la fréquence d'utilisation des UALs soit en allongeant le nombre de cycles d'exécution des instructions, soit en permettant au

concepteur de spécifier les instructions opératives à un niveau plus fin. Seule cette dernière solution permet de faire fonctionner les opérateurs à leur fréquence maximale d'utilisation.

3.1.4. Avantages du modèle à 2 bus

L'utilisation de ce modèle présente bien sûr à la fois des avantages et des inconvénients. On a vu précédemment que les possibilités de parallélisme sont limitées. On peut d'ailleurs étendre ce modèle à 3 bus ou plus. Les possibilités de parallélisme sont accrues mais restent toujours bornées. On peut toujours trouver des instructions non implantables sur une partie opérative à n bus. Le grand avantage de ce modèle est qu'il fournit en même temps qu'un modèle architectural un modèle topologique.
L'utilisation de bus parallèles qui traversent toute la largeur de la partie opérative permet d'avoir des cellules de hauteur constante et de disposer de façon judicieuse les flux de données et de contrôle. Les bus de données étant au dessus de la logique et le flux de contrôle perpendiculaire au flux de données, on obtient un bloc très dense pour la partie opérative avec une zone de connexion réduite avec la partie contrôle.
Le choix de bus morcelés permet à la fois le parallélisme d'exécution et une réduction du nombre des connexions aux bus.
Un registre ne peut être connecté qu'aux 2 bus de la même sous partie opérative. Si l'on veut effectuer un transfert d'un registre dans un registre d'une autre sous partie opérative, au lieu d'ajouter une nouvelle connexion entre le premier registre et un des bus de la sous partie opérative où se trouve le second registre, on connecte les 2 bus inférieurs ou les 2 bus supérieurs des sous parties opératives contenant ces 2 registres ainsi que les bus intermédiaires si ces sous parties opératives ne sont pas voisines.
On économise ainsi des éléments d'interconnexion puisque l'on ne connecte un registre qu'à au plus 2 bus et que les éléments d'interconnexion entre bus peuvent servir eux à plusieurs transferts.
C'est d'ailleurs cette organisation qui limite la puissance du modèle. On a vu qu'en plaçant les sous parties opératives les unes au dessus des autres et en reliant les bus de ces sous parties opératives à l'aide de bus d'interconnexion, on supprime toute limitation de puissance. On aboutit hélas à un mauvais résultat en ce qui concerne l'utilisation de la surface. Les commandes des sous parties opératives inférieures ne peuvent être amenées directement que si les sous parties opératives supérieures sont transparentes à ces commandes, ce qui conduirait bien sûr à un élargissement des sous parties opératives mais diminuerait la surface occupée par les zones de routage nécessaires à l'acheminement des données.
Le choix de 2 bus est un compromis:
- le modèle à un bus est très limité: on trouve des combinaisons d'instructions très simples qui ne peuvent pas être exécutées à l'aide de ce modèle. D'autre part si l'on peut effectuer des opérations binaires en 2 cycles en plaçant les opérateurs entre 2 segments, il semble difficile d'exécuter des opérations binaires pour lesquelles on

veut récupérer à la fois le résultat et les indicateurs arithmétiques. Pour avoir 2 bus disponibles à la fois pour les entrées et pour les sorties de l'opérateur, il faut trouver une organisation topologique de la cellule qui permette de placer sur chaque segment une entrée et une sortie. En outre on ne peut pas effectuer des opérations ternaires en 2 cycles.

- le choix de 2 bus est un bon compromis si les instructions opératives comprennent de nombreuses opérations binaires. Si l'on peut effectuer des opérations binaires en 2 cycles à l'aide d'un seul bus segmenté, il est beaucoup plus aisé de disposer de 2 bus. Il n'y a aucun problème pour transférer à la fois le résultat et les indicateurs arithmétiques. D'autre part si l'on place les opérateurs entre 2 sous parties opératives, c'est à dire en fait si on ne met pas les entrées et la (ou les) sorties des opérateurs sur la même sous partie opérative, on peut pipeliner les opérations binaires ou exécuter des opérations ternaires ou quaternaires et même pipeliner les opérations ternaires sans mémorisation de compte rendu en 2 cycles. La puissance du modèle est bien supérieure à celle du modèle à un bus.
- l'utilisation d'un troisième bus permet d'augmenter la puissance du modèle: il existe beaucoup moins d'instructions non exécutables en 2 cycles. On peut pipeliner les opérations binaires sans mettre les opérateurs entre 2 sous parties opératives voisines, on peut pipeliner les opérations ternaires en mettant les opérateurs entre 2 sous parties opératives voisines. Toutefois en rajoutant un bus, on augmente aussi la surface dans la mesure où on augmente la hauteur des tranches. Mais il se peut que la largeur des cellules puisse être diminuée puisque leur hauteur augmente. Si une architecture à 3 bus ou plus est intéressante, l'utilisation d'un tel modèle pose les problèmes suivants:
- augmentation de la combinatoire,
- disponibilité d'une bibliothèque de cellules à 3 bus.

3.2. Algorithme d'allocation des ressources

3.2.1. Présentation du problème

Nous considérons ici le modèle de partie opérative à 2 bus segmentables pour des instructions exécutables en 2 cycles opératifs. Il s'agit de déterminer l'architecture d'une partie opérative qui permette d'exécuter un ensemble d'instructions, chacune de ces instructions pouvant comporter plusieurs actions exécutables en 2 cycles. Il faut donc déterminer:

- le nombre de sous parties opératives et leurs connexions, ou autrement dit le nombre de segments de chacun des 2 bus et les connexions entre segments adjacents,
- la répartition des registres entre ces différentes sous parties opératives et leurs connexions aux 2 bus, ou autrement dit les connexions des registres aux différents segments des 2 bus,
- les constantes à placer dans chaque sous partie opérative; les constantes peuvent être dupliquées à l'inverse des registres,
- les opérations effectuées sur chaque sous partie opérative; comme les constantes, les opérateurs peuvent être dupliqués,
- les tampons d'entrée-sortie de plots à placer dans chaque sous partie opérative.

Le problème de la génération de l'architecture de la partie opérative peut être décomposé en 2 sous problèmes: trouver une solution et optimiser ensuite. On peut considérer dans un premier temps que tous les éléments de la partie opérative sont reliés aux 2 bus, que toutes les constantes ainsi que tous les opérateurs sont disponibles directement dans chaque sous partie opérative et que toutes les sous parties opératives voisines sont connectées entre elles par les 2 bus. C'est à dire que l'on duplique tous les éléments duplicables et que l'on connecte tous les éléments au maximum des possibilités de connexion. Le seul problème critique est que l'on ne peut dupliquer les registres. Si l'on procède comme indiqué ci dessus, on peut se concentrer sur le problème de répartition des registres entre les différentes sous parties opératives. On cherchera à optimiser ensuite.
Le problème est complexe. Supposons que l'on ait n registres et p sous parties opératives; n est une donnée du problème.
Un minorant de p peut être déterminé à partir du nombre maximal d'opérations à faire en parallèle et du nombre maximal de transferts dans une instruction.
$p \geq \max$ (maxopérations[I], maxtransferts[I]/4, maxtransferts_simples[J]/2)
où I est une instruction quelconque et J une instruction ne comportant que des transferts.
p doit être supérieur au nombre maximum d'opérations à exécuter en parallèle, au quart arrondi à l'unité supérieure du nombre maximal de transferts élémentaires après réduction des opérations en transferts et à la moitié arrondie à l'unité supérieure du nombre maximum de transferts pour les instructions ne comprenant que des transferts.
On peut placer les registres dans n'importe quel ordre à l'intérieur d'une sous partie

opérative.
Le nombre de façons de placer ces n registres sur ces p sous parties opératives est égal au nombre d'applications d'un ensemble à n éléments dans un ensemble à p éléments, soit p^n. En fait on peut réduire de moitié le nombre de cas à envisager en remarquant que si l'on inverse l'ordre des sous parties opératives, le problème ne change pas.
Si p est pair, il y a $p^n/2$ cas à considérer; si p est impair, il y a $(p^n+1)/2$ cas à considérer.
En effet le cas où tous les registres sont placés sur la sous partie opérative centrale n'est à compter qu'une seule fois. On peut remarquer d'ailleurs qu'il n'est guère intéressant d'envisager ce cas puisque l'on ne peut alors exécuter qu'au plus 4 transferts en 2 cycles.
Mais il n'est pas aberrant de considérer le cas d'une sous partie opérative sans registre car une sous partie opérative peut avoir uniquement une UAL.
Exemple:
(A + B; B + C; C + D;)

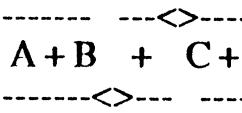


figure 3.40 : sous partie opérative sans registre

Donnons un exemple numérique représentatif : si on a 10 registres, 3 sous parties opératives et 100 instructions, on aura:
$(3^{10}+1)/2$ soit 29 525 cas à considérer et on devra vérifier pour chaque cas que les 100 instructions peuvent être exécutées; on devra donc effectuer 2 952 500 vérifications.
Cet exemple simple montre que le traitement combinatoire de ce problème est irréaliste.
On a supposé que l'on connaissait à l'avance le nombre de sous parties opératives. On n'est cependant jamais certain que ce nombre ne risque pas d'être supérieur au minimum calculé. S'il est facile de déterminer un minorant du nombre de sous parties opératives, il est plus difficile de trouver un majorant dans la mesure où on peut dupliquer constantes et opérateurs. Si l'on ne considère que les registres et si on met un registre par sous partie opérative, il y a n!/2 cas à considérer. Le problème revient alors à trouver l'ordre des registres.

On remarquera cependant que seules les instructions comportant plus de 4 transferts élémentaires après décomposition des opérations ou plus de 2 transferts si l'instruction ne comporte que des transferts sont critiques pour le placement des registres. En effet on peut toujours effectuer 2 transferts en un cycle opératif et par conséquent 4 transferts en 2 cycles opératifs en connectant tous les segments d'un même bus entre eux.

3.2.2. Présentation d'un algorithme de construction de la partie

opérative

L'algorithme d'Apollon [ARZO 84] est basé sur une construction progressive de la partie opérative. Le compilateur crée des sous parties opératives et y place les registres, constantes et opérateurs au fur et à mesure des besoins. Il implante les instructions une à une. Il commence par examiner les instructions comprenant le plus de transferts en parallèle et termine par les instructions les plus simples. Pour limiter la combinatoire, on impose au compilateur de ne pas remettre en cause le placement des registres placés au cours de la compilation des instructions précédentes. De plus au cours d'un premier traitement des instructions, le compilateur détermine les contraintes les plus fortes portant sur le placement des registres, résultant de l'exécution séquentielle de toutes les instructions.
Ces contraintes sont prises en compte lors de la compilation de chaque instruction. C'est ce qui permet de limiter les retours en arrière au placement des registres d'une même instruction. Il n'est hélas pas possible de tenir compte de toutes les contraintes, on se heurte à un problème combinatoire similaire à celui de départ. Il se peut donc que le compilateur ne trouve pas la solution. Dans ce cas ou dans le cas où il n'y a pas de solution, le compilateur essaie de recompiler cette instruction en augmentant le nombre de cycles d'exécution. Pour le moment, on se contente de découper l'instruction en deux ce qui revient à exécuter cette instruction en 4 cycles.
En ce qui concerne le placement des registres d'une même instruction, le compilateur utilise des heuristiques. Il place les registres, qui apparaissent souvent ensemble dans des transferts ou des opérations, à l'intérieur d'une même sous partie opérative dans la mesure du possible tout en cherchant à minimiser le nombre de bus utilisés pour chaque action opérative.
A la différence des registres les constantes et les opérateurs peuvent être dupliqués si nécessaire. Les connexions sont ajoutées au fur et à mesure des besoins. La recherche d'une solution au problème de partitionnement des registres prime la minimisation du nombre de constantes, d'opérateurs et de connexions car le problème est déja soumis à de fortes contraintes, le nombre de bus étant limité.

3.2.2.1. Prétraitement des instructions

Le prétraitement des instructions est destiné à réduire le nombre de cas à explorer tout en n'éliminant pas trop de solutions.

3.2.2.1.1. Extraction des contraintes de placement des registres

Nous avons vu dans le paragraphe décrivant les limites du modèle que certaines instructions ou combinaisons d'instructions ne peuvent être exécutées sur une partie opérative à 2 bus segmentés. Même si une instruction peut être exécutée, il se peut que le placement des registres résultant de la compilation des instructions précédentes ne permette pas de l'exécuter. Pour pouvoir le faire, il faudrait replacer certains registres. C'est pour cette raison que nous avons cherché à dégager pour

l'exécution d'une instruction des conditions nécessaires portant sur le placement des registres.
Les actions les plus contraignantes sont évidemment les opérations binaires qui monopolisent 2 bus pour le chargement des opérandes.

1.couple d'opérations binaires:

(D1 := A op1 B; D2 := C op2 D;)
Les destinations et les opérations importent peu; seuls les opérandes nous intéressent.
On suppose qu'au moins 3 des opérandes sont distincts sinon on peut toujours exécuter cette instruction.
- soit ces 2 opérations ont un opérande commun, mettons B=D, il faut et il suffit alors que les 2 autres opérandes A et C ne soient pas placés sur la même sous partie opérative,
- soit les 4 opérandes sont distincts: une condition nécessaire et suffisante pour que cette instruction soit exécutable est que:

$[A\ B] \cap [C\ D] \neq [A\ B]$ et

$[A\ B] \cap [C\ D] \neq [C\ D]$

où [A B] désigne l'ensemble des sous parties opératives entre A et B, celle(s) contenant A et B y compris.

2.triplet d'opérations binaires

(D1 := A op1 B; D2 := C op2 D; D3 := E op3 F;)
On suppose que les 2 opérandes d'une opération sont distincts et que 2 opérations ont au plus un registre en commun: il y a donc au moins 3 opérandes distincts.
Les différents cas ont déjà été examinés au paragraphe 3.1.2 : limites du modèle.
On rappelle que si les opérandes sont tous distincts, on doit s'assurer que les conditions nécessaires et suffisantes pour exécuter les 3 couples d'opérations sont respectées et en plus que:

$[A\ B] \cap [C\ D] \cap [E\ F] = 0$

pour pouvoir affirmer que le chargement des opérandes est possible en un cycle.
Il faut de plus que les 3 destinations n'appartiennent pas à la même sous partie opérative.

3.triplet d'opérations unaires

(A:= -B; C:= -D; E:= -F;)
Il faut que ni les 3 sources ni les 3 destinations ne soient placées dans la même sous partie opérative.

4.une opération binaire et une opération unaire

(A:=B+C; D:=-E;)
On suppose les 3 opérandes distincts sinon il n'y a pas de problème.
Il faut et il suffit que les 3 opérandes B, C et E ne soient pas placés dans une même sous partie opérative.

5.un couple d'opérations binaires et une opération unaire
(D1:=A+B; D2:=C+D; D3:=-E;)
On suppose que l'opérande de l'opération unaire n'est pas opérande des opérations binaires sinon on est ramené à 2 opérations binaires.
Deux cas peuvent se présenter en ce qui concerne les opérandes:
- soit les opérations binaires ont un opérande commun : pour pouvoir charger les opérandes en un cycle, il faut et il suffit que les conditions nécessaires et suffisantes portant sur tous les couples d'opérations soient respectées. De plus, il faut que les 3 destinations n'appartiennent pas à la même sous partie opérative,
- si tous les opérandes sont distincts, il faut aussi vérifier que:
E n'appartient pas à [A B] $\cap$ [C D]

6.triplet de transferts à exécuter en un cycle
On suppose que les 3 sources sont distinctes sinon il n'y a pas de condition.
Plusieurs cas peuvent se présenter:
- il n'y a que 3 registres distincts: on a une permutation circulaire et on ne peut exécuter cette instruction en un cycle,
- il n'y a que 4 registres distincts: puisque les sources et les destinations sont distinctes, c'est donc que 2 registres sont à la fois source et destination, soit:
(A:=B; B:=C; C:=D;)
Il faut et il suffit que [A B] $\cap$ [C D] = 0 pour pouvoir exécuter cette instruction en un cycle,
- il y a au moins 5 registres distincts: un registre peut être à la fois source et destination.
(A:=B; C:=D; E:=F;)
Il faut et il suffit que [A B] $\cap$ [C D] $\cap$ [E F] = 0. La condition nécessaire et suffisante trouvée pour 4 registres distincts peut aussi s'exprimer à l'aide de cette égalité.

On pourrait examiner d'autres contraintes portant sur des instructions différentes. On peut distinguer en fait 2 types de contraintes:
- des contraintes ponctuelles qui expriment que l'on ne doit pas mettre ensemble dans la même sous partie opérative certains registres:
1. il ne faut pas que 3 registres qui apparaissent comme sources ou destinations de 3 transferts différents que l'on doit exécuter au même cycle (instruction ne comportant que des transferts - chargement des opérandes - récupération des résultats) soient placés dans la même sous partie opérative.
2. il ne faut pas que 5 registres qui apparaissent comme sources ou destinations de 5 transferts élémentaires différents (c'est à dire que certains transferts peuvent provenir de la décomposition d'opérations unaires ou binaires) soient réunis dans la même sous partie opérative.

Pour ne pas avoir de conditions redondantes, il n'est pas utile d'inclure dans ces 5 registres 3 registres qui proviennent de transferts s'exécutant au même cycle

(chargement des opérandes ou récupération des résultats). En effet des conditions plus fortes ont déjà été dégagées pour ces triplets de registres.

- des contraintes d'intervalles qui expriment qu'on ne doit pas mettre tels registres entre tels autres:

nous avons dégagé des contraintes de ce type pour le chargement des opérandes d'opérations binaires en parallèle ou pour des transferts en nombre au moins égal à 3 à exécuter en un cycle. Dans les 2 cas, les transferts doivent être exécutés en un cycle.

De façon plus générale, on obtient des conditions nécessaires et suffisantes portant sur le placement des registres uniquement pour les instructions ne comportant que des transferts.

En effet lorsqu'il y a une opération à exécuter, on introduit en quelque sorte un degré de liberté en plus, qui est la place de l'opérateur.

On peut jouer en effet sur la place des opérateurs pour exécuter une instruction car on peut les dupliquer à l'inverse des registres.

Ces conditions nécessaires sont en grand nombre. Pour ne pas ralentir trop la compilation, nous avons décidé de ne tenir compte que des conditions les plus fortes: les conditions portant sur les transferts s'exécutant en un cycle. Dans le cas des transferts se déroulant en 2 cycles, il y a un degré de liberté en plus, et l'exécution de ces instructions pose moins de problèmes.

On retient donc les conditions ponctuelles suivantes:

une opération binaire et une opération unaire en parallèle: les 3 opérandes ne doivent pas être ensemble

2 opérations binaires en parallèle ayant un opérande commun: les 2 autres opérandes ne doivent pas être ensemble

3 opérations unaires en parallèle: les 3 sources ne doivent pas être ensemble

3 opérations en parallèle: les 3 destinations ne doivent pas être ensemble

Il faut aussi rajouter les combinaisons d'opérations et d'actions d'entrée-sortie de la partie opérative, car ces dernières se déroulent soit au premier cycle (sorties similaires en cela au chargement des opérandes), soit au second cycle (entrées similaires en cela à la récupération des résultats).

On retient les conditions d'intervalles suivantes:

<u>2 opérations binaires en parallèle:</u> conditions sur le chargement des opérandes; les opérandes sont tous distincts

(D1:=A+B; D2:=C+D;)

$[A\ B] \cap [C\ D] \neq [A\ B] \neq [C\ D]$

<u>3 opérations binaires en parallèle:</u> conditions sur le chargement des opérandes

(D1:=A+B; D2:=C+D; D3:=E+F;)

en plus des conditions portant sur les couples d'opérations, on a la condition suivante lorsque les opérandes sont distincts:

$[A\ B] \cap [C\ D] \cap [E\ F] = 0$

<u>2 opérations binaires et une opération unaire, tous les opérandes étant distincts:</u>

(D1:=A+B; D2:=C+D; D3:=-E;)

E n'appartient pas à [A B] ∩ [C D]
3 transferts à exécuter en un cycle:
(A:=B; C:=D; E:=F;)
[A B] ∩ [C D] ∩ [E F] = 0
Dans les autres cas (pour plus de 3 opérations ou pour plus de 3 transferts par exemple), on combine ces différentes conditions.
Remarque: on a le même type de contraintes pour le transfert des résultats que pour le transfert des opérandes pour les opérations dont on veut récupérer à la fois le résultat proprement dit et les indicateurs arithmétiques, si ce n'est que les destinations doivent être forcément distinctes.

3.2.2.1.2. Construction d'un graphe de proximité

Le graphe de proximité permet d'exprimer qu'il existe un ou plusieurs transferts entre 2 registres de la partie opérative. Ce graphe est utilisé pour définir une heuristique de recherche des solutions qui permet de réduire le nombre de cas explorés. Ce graphe peut aussi être utilisé pour simplifier le problème de partition des registres dans le cas où il existe plusieurs composantes connexes.
Alors que les conditions de placement de registres expriment que tels ou tels registres ne doivent pas être ensemble sur une même sous partie opérative, nous cherchons à exprimer ici que des registres doivent être placés dans la même sous partie opérative ou si c'est impossible qu'ils doivent être placés dans des sous parties opératives voisines. En effet il semble avantageux de grouper les registres qui apparaissent souvent ensemble comme source(s) ou destination(s) d'actions opératives.
Pour cela on construit un graphe dont les sommets sont les registres et dont les arcs relient les registres qui apparaissent au moins une fois dans une même action opérative. Un arc entre 2 registres porte un poids d'autant plus grand que les registres apparaissent plus de fois ensemble dans des actions opératives.
Pour les échanges avec une mémoire externe, seul le transfert entre le registre source ou destination et le tampon d'entrée-sortie des plots est pris en compte.
La construction de ce graphe se fait au fur et à mesure de l'analyse des actions opératives de chaque instruction. On relie par des arcs les registres d'une même action opérative qui ne sont pas encore reliés et on incrémente le poids de chacun des arcs reliant 2 registres de cette action:
- de 2 si les registres reliés par cet arc sont soit les 2 sources, soit les 2 destinations d'une opération binaire,
- de 1 si l'un des registres est source et l'autre destination.

Le poids des arcs entre 2 sources ou entre 2 destinations est augmenté d'une valeur double, car les 2 registres interviennent dans 2 transferts qui doivent se dérouler le même cycle.
Remarque importante: on ne tiendra compte que des instructions qui ont plus de 4 transferts après décomposition des opérations en transferts. En effet on peut toujours exécuter 4 transferts en 2 cycles sur une partie opérative à 2 bus segmentés.

Si l'on exécute les instructions ne comportant que des transferts en un cycle, on gardera aussi les instructions de ce type à 3 ou 4 transferts.
Il n'est guère utile d'introduire les opérateurs dans le graphe de proximité car les opérateurs peuvent être dupliqués à la différence des registres.
Ce graphe est utilisé pour orienter le choix de la place des registres de façon heuristique, mais il peut aussi être utilisé dans certains cas pour simplifier considérablement le problème de partition des registres.
Les registres d'une composante connexe peuvent être placés indépendamment des autres registres à condition qu'on les place sur des sous parties opératives distinctes.
S'il existe plusieurs composantes connexes, on peut de plus décomposer les instructions en sous-instructions portant sur les registres d'une même composante.
Pour les registres de chaque composante, on construit un nouveau graphe en ne considérant que les sous-instructions de plus de 4 transferts élémentaires ou de plus de 2 transferts pour une instruction ne comportant que des transferts. On continue le processus de décomposition en composantes connexes.
Si l'on obtient après un certain nombre d'itérations des graphes qui sont tous vides, toutes les instructions portant sur les registres d'une même composante peuvent s'exécuter en 2 cycles ou en un cycle selon le cas. On démontre ainsi que l'on peut compiler toutes les instructions et on obtient une décomposition en sous parties opératives qui n'est pas forcément optimum.
Il n'y a hélas pas en général plusieurs composantes connexes et lorsqu'il y en a plusieurs les sous-instructions qui ne portent que sur les registres d'une composante n'ont pas forcément moins de 4 transferts (ou 2 s'il s'agit d'une sous instruction ne comportant que des transferts); cette méthode ne peut donc être appliquée seule.
On peut aussi tenir compte dans le graphe du degré de parallélisme de chaque instruction. On incrémente le poids de chaque arc du produit de la valeur ajoutée précédemment par le nombre de transferts élémentaires de l'instruction ou par le double de ce nombre si l'instruction ne comporte que des transferts.

Il est possible de tenir compte dans une certaine mesure des conditions d'exécutabilité d'instructions à l'aide d'un graphe.
On relie chacun des registres d'une action opérative aux registres apparaissant dans les autres actions opératives d'une même instruction. On associe à chaque arc un poids qui exprime que les 2 registres reliés sont des source(s) ou destination(s) de transferts concurrents et donc que l'on n'a pas intérêt à les grouper dans une même sous partie opérative. Le seul problème est qu'un graphe ne permet d'exprimer que des relations binaires alors que les relations d'incompatibilité de placement des registres sont souvent ternaires ou n-aires en général.
De plus un tel graphe ne permet pas de tenir compte des contraintes d'intervalles.
Néanmoins un graphe d'incompatibilité permet d'exprimer à peu de frais que des registres apparaissent dans des transferts concurrents et peut être utilisé en même temps que le graphe de proximité pour sélectionner les combinaisons de placement de registres les plus plausibles.

3.2.2.1.3. Tri des instructions

Le tri des instructions consiste à ordonner les instructions dans un ordre décroissant de complexité. Les instructions sont compilées dans cet ordre.
Comme aucun retour en arrière remettant en cause l'implantation d'une instruction n'est effectué, c'est ce tri qui guide la recherche des solutions.
Les instructions comportant le plus de transferts élémentaires (après réduction des opérations en transferts) sont les instructions qui peuvent être le plus difficile à compiler car ce sont elles qui utilisent le plus les ressources de la partie opérative et donc celles qui ont le plus de chance de provoquer un conflit d'utilisation des bus.
On trie donc les instructions de sorte que les instructions les plus complexes soient compilées en priorité. On tient compte en outre:
- des multitransferts (il y a multitransfert lorsque plusieurs actions opératives ont une source commune),
- du nombre de conditions d'implantation des instructions.

Plus il y a de conditions à respecter, plus il sera facile d'implanter cette instruction.
On effectue dans un premier temps une partition de l'ensemble des instructions en classes. Dans chaque classe, on regroupe les instructions qui ont le même nombre de transferts ayant des sources différentes. Par exemple, une opération binaire avec mémorisation des indicateurs arithmétiques compte pour 4 transferts.
On ordonne les instructions de chaque classe en tenant compte:
- d'abord du nombre de conditions d'implantation. On met en premier les instructions qui ont le moins de conditions d'implantation.
- et à nombre égal du nombre de transferts, un multitransfert à n destinations comptant cette fois pour n et non pour 1.

Le problème est différent si l'on cherche à implanter les instructions ne comportant que des transferts en un cycle. Pour pouvoir comparer les instructions à exécuter en un cycle et celles devant s'exécuter en 2 cycles, on multiplie par 2 le nombre des transferts d'une instruction ne comportant que des transferts. A nombre de transferts égal, on pourrait alors tenir compte du type des instructions si ce n'est que l'on en tient déjà compte d'une certaine manière en comptant le nombre de conditions d'exécution.

3.2.2.1.4. Tri statique des actions d'une instruction

Si l'on ne fait pas de retour en arrière, une fois que l'on a implanté une instruction, il n'en est pas de même lorsque l'on a implanté une action opérative. Afin de minimiser le nombre de retours en arrière, on compile en premier les actions les plus complexes, c'est à dire celles comportant le plus de transferts et pour les transferts (actions d'entrée-sortie ou transferts de registres) on compile en premier les actions d'entrée-sortie car celles-ci doivent être exécutées à un cycle donné dans un laps de temps de 2 cycles.
Dans le cas où l'on a un ensemble d'actions d'entrée et de sortie, on implantera d'abord les actions du type qui est le plus représenté dans l'instruction car elles

doivent être exécutées pendant le cycle qui est le plus chargé.
Lorsqu'il y a des multitransferts, on les regroupe car on a intérêt à les implanter ensemble puisqu'ils ont même source.

3.2.2.2. Construction de la partie opérative

Les instructions sont implantées une à une dans l'ordre défini au premier passage et en tenant compte des conditions d'implantation, des relations de proximité extraites auparavant. Si une instruction ne peut être exécutée en le nombre de cycles fixé par le modèle, on augmente d'un le nombre de cycles et on essaie à nouveau d'implanter l'instruction. On peut exécuter t transferts en t/2 (arrondi à l'unité supérieure) cycles car l'on dipose toujours de 2 bus au moins, ce qui permet d'effectuer 2 transferts en un cycle. Actuellement on découpe l'instruction non exécutable en 2 et on reprend la compilation.

3.2.2.2.1. Compilation d'une action opérative sans création de sous partie opérative

On considère qu'il y a au départ 2 sous parties opératives (2 dès qu'on est sûr que la solution si elle existe a au moins 2 sous parties opératives).
Pour chacun des registres non encore placés de l'action, on génère tous les cas de placement possibles sans création de nouvelle sous partie opérative.
Il y a donc autant de cas que de sous parties opératives.
On génère ensuite toutes les combinaisons de ces cas de placement pour les registres non placés de l'action opérative et on élimine les combinaisons qui ne satisfont pas aux conditions extraites lors du premier traitement des instructions. Si l'action est une opération, on génère pour chacune des hypothèses de placement des registres restantes tous les cas de placement de l'opérateur.
Pour une opération binaire avec mémorisation des comptes rendus dont aucun des registres n'est placé, on envisage donc p^4 cas de placement différents pour les registres où p est le nombre de sous parties opératives existantes.
Si chacun de ces p^4 cas satisfait aux conditions de placement des registres, on doit envisager en tout p^5 pour le placement des registres et de l'opérateur.
On trie toutes ces hypothèses en tenant compte des critères suivants dans cet ordre d'importance:

1) en ce qui concerne le placement des registres, on essaie de placer chacun des registres dans la sous partie opérative qui maximise la somme des poids de proximité de ce registre aux registres déjà placés de cette sous partie opérative et on cherche à minimiser les distances 2 à 2 des différents registres de l'instruction (cf paragraphe 3.1.1.) afin de minimiser le nombre de segments utilisés pour exécuter l'action opérative tout en tenant compte des autres instructions.

2) en ce qui concerne le placement de l'opérateur, on essaie de minimiser d'abord la distance entre l'opérateur et ses opérandes et ensuite la distance entre l'opérateur et sa destination.

On ordonne les hypothèses dans un premier temps en ne tenant compte que du placement des registres; ensuite on ordonne les hypothèses pour un même placement des registres.

Une fois que les différentes possibilités d'implanter l'action en cours d'implantation ont été ordonnées, on commence à examiner le premier placement des registres et éventuellement de l'opérateur de l'action.
Dans le cas où l'action opérative est une opération, on ne génère les différentes possibilités de connexion que si l'on ne fait pas d'autre opération sur la sous partie opérative où doit s'exécuter l'opération sinon on passe au second cas de placement des éléments de l'opération. On génère toutes les possibilités de connexion des registres aux bus mais on ne génère que les possibilités de connexion des tampons de l'opérateur qui correspondent aux connexions des registres de l'action opérative.
Pour une opération binaire, il y a 4 façons de connecter les registres aux 2 bus. Il y a 2 choix possibles pour la destination (bus haut - bus bas) et 2 choix possibles pour le couple des opérandes.
opération commutative: C:=A+B

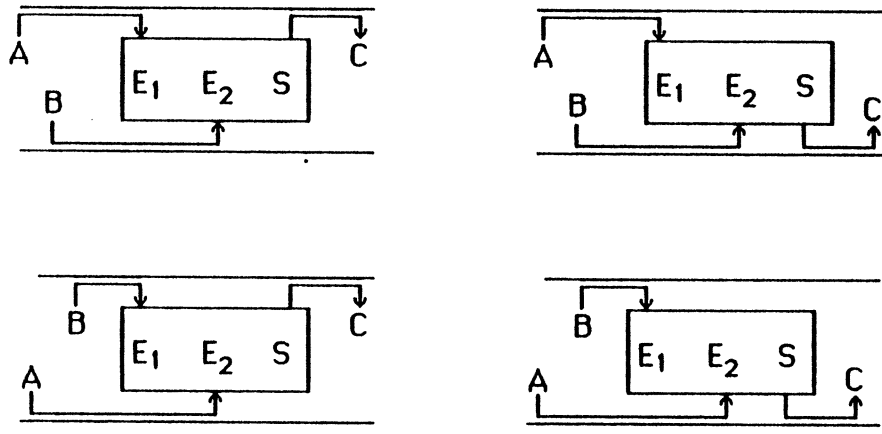


figure 3.41 : connexions pour l'exécution d'une opération binaire commutative

Si l'opérateur est commutatif, on connecte toujours le bus haut à l'entrée 1 de l'opérateur et le bus bas à l'entrée 2.
opération non commutative: C:=A-B

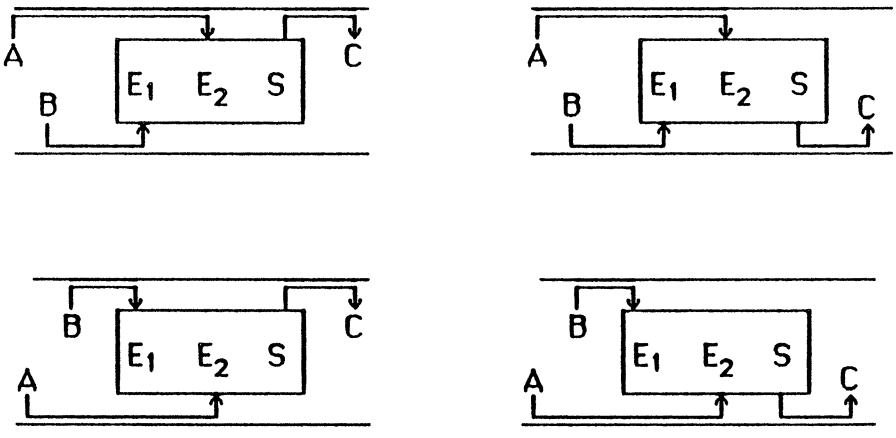


figure 3.42 : connexions pour l'exécution d'une opération binaire non commutative

Le second opérande d'une opération non commutative comme la soustraction doit être transféré dans le premier tampon de l'additionneur car la structure de bus des cellules de la bibliothèque actuelle d'Apollon ne permet pas de complémenter les 2 entrées de l'additionneur.
Pour une opération binaire avec mémorisation des indicateurs arithmétiques, il existe aussi 4 possibilités de connexion. Il y a 2 choix possibles pour le couple des opérandes et 2 choix pour le couple des destinations (résultat et indicateurs arithmétiques).
Pour une opération unaire, il existe aussi 4 possibilités de connexion.
B:=-A;

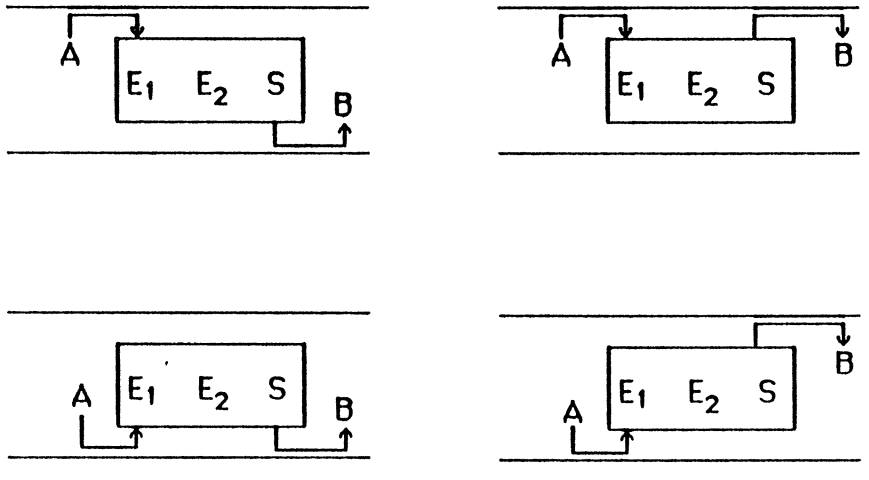


figure 3.43 : connexions pour l'exécution d'une opération unaire

L'opérande est transféré dans l'entrée 1. L'autre entrée peut être forcée à zéro.
Pour un transfert B:=A, il n'y a que 2 possibilités pour chaque cycle.
Si le transfert doit s'exécuter à un cycle fixé, on génère 2 possibilités sinon 4.
Dans tous les schémas précédents, on a regroupé les registres et les entrées-sorties

de l'opérateur sur une même sous partie opérative, mais le problème est le même s'ils sont placés sur plusieurs sous parties opératives, car ce qui importe c'est le choix des connexions aux 2 bus de la partie opérative.
Remarque:
Les tampons d'entrée et de sortie d'un opérateur peuvent être connectés aux 2 bus. On espère ainsi diminuer le nombre global de connexions en ajoutant en priorité des connexions aux tampons car les tampons doivent être en moyenne plus utilisés que les registres déclarés par l'utilisateur.
Ces possibilités de connexion sont triées de façon à ce que les cas de connexion placés en premier minimisent le nombre de nouvelles connexions physiques à créer. En cas d'égalité, on choisit plutôt de créer une connexion sur l'un des tampons de l'opérateur (qui aura plus de chance de servir pour une autre instruction) plutôt que de créer une nouvelle connexion sur un registre.
Une fois que toutes les possibilités de connexion pour les registres (et les tampons de l'opérateur) de l'action ont été générées et ordonnées, on examine le premier cas d'implantation de l'action opérative.
Si les bus utilisés par cette action sont tous libres ou peuvent être partagés, on implante effectivement l'instruction. On place les registres, les tampons des plots d'entrée-sortie, les constantes et des opérateurs dans les sous parties opératives correspondantes et on crée de nouvelles connexions si besoin est.
En même temps, on note pour chaque cycle les bus qui sont utilisés et les éléments de mémorisation auxquels ils sont connectés pendant ce cycle ainsi que les opérations effectuées et les sous parties opératives où elles ont été effectuées. D'autre part on garde les possibilités de connexion non encore examinées de façon à ce qu'en cas d'échec à la suite de l'implantation d'une des actions restantes de l'instruction, on réexamine d'abord les autres hypothèses de placement avant les hypothèses de connexion correspondant au placement des registres choisi initialement pour cette instruction.
Si la première hypothèse de connexion ne convient pas, on examine la suivante. Si aucune des hypothèses ne convient, on envisage la création d'une nouvelle sous partie opérative.

3.2.2.2.2. Création d'une sous partie opérative

Comme au départ, le nombre de sous parties opératives est initialisé à 2 (s'il en faut au moins 2 bien sûr), il peut être nécessaire de créer de nouvelles sous parties opératives.
On procède de la même façon que précédemment sauf que l'on génère pour chaque registre non encore placé de l'instruction tous les placements dans les sous parties opératives déjà existantes mais aussi dans la nouvelle sous partie opérative. On ne retient ensuite que les combinaisons de ces placements qui font intervenir la création d'une et d'une seule sous partie opérative. Comme précédemment on vérifie les conditions de placement des registres et on trie les hypothèses de placement d'une manière analogue. On définit la proximité d'un registre et d'une sous partie

opérative à créer comme le tiers de la somme des proximités du registre aux 2 sous parties opératives voisines si l'on crée la nouvelle sous partie opérative entre 2 sous parties opératives et sinon comme la proximité du registre et de la sous partie opérative qui se trouve à l'extrémité où on veut créer la nouvelle sous partie opérative.
Cette définition et en particulier la division par 3 et non par 2 de la somme des proximités d'un registre aux sous parties opératives entre lesquelles on crée la nouvelle sous partie opérative permet d'avantager la création de nouvelles sous parties opératives à l'une des 2 extrémités de la partie opérative déjà existante, ce qui est plus intuitif.
Pour ce qui est des hypothèses de connexion, on procède comme précédemment. Si aucune des hypothèses de connexion ne peut être satisfaite, alors soit il y a échec si l'action en cours d'implantation est la première action de l'instruction, soit on remet en cause l'implantation de l'action précédente.
Remarque:
Au départ, on crée 2 sous parties opératives vides si on est sûr que la partie opérative a au moins 2 sous parties opératives. Si l'on envisage la création d'une seule sous partie opérative par action, on est sûr de pouvoir implanter la première action si elle comporte moins de 4 registres dans la mesure où on suppose que dans le pire cas il faut une sous partie opérative par registre.

3.2.2.3. Contraintes imposées par la bibliothèque de cellules utilisée pour générer les masques de la partie opérative

Nous avons déjà vu en ce qui concerne les opérations que :
- on ne peut complémenter que la première entrée de l'additionneur ou de l'ual.
- que les opérandes d'opérations unaires doivent être chargés dans le premier tampon des boîtes à opérations.

D'autre part, le tampon d'entrée des informations venant d'une mémoire externe ne peut être connecté aux 2 bus de la partie opérative lorsqu'il est inclus dans un bloc d'interface d'entrée-sortie.

3.2.3. Evaluation des performances de l'algorithme d'allocation des ressources

L'algorithme a été écrit en Lelisp et tourne sur SM90 sous le système d'exploitation dérivé d'UNIX : SMX. Le programme comprend environ 2000 lignes de lisp. Le temps d'exécution ne dépend guère du nombre d'instructions opératives, il dépend surtout de la complexité des instructions, c'est à dire du nombre d'actions opératives à exécuter en parallèle dans chaque instruction. Par exemple la description du 6502 qui comporte une centaine d'instructions est compilée en 2 minutes (temps utilisateur). La partie opérative a deux sous parties opératives. Les instructions opératives comportent donc au plus 2 opérations en parallèle.
La raison pour laquelle le temps d'exécution ne dépend que faiblement du nombre

d'instructions de la description est simple : aucun retour en arrière n'est effectué pour recompiler une instruction. Les premières instructions compilées nécessitent un temps d'exécution plus élevé car au départ aucun registre n'est placé et le nombre d'alternatives de placement des registres peut être important. Ce nombre dépend du nombre de sous parties opératives et donc de la complexité des instructions. Lorsque tous les registres ont été placés, la compilation d'une instruction est simplifiée : le compilateur vérifie uniquement que l'instruction est exécutable dans la configuration actuelle et alloue de nouveaux opérateurs, constantes et connexions aux bus si besoin est.
Apollon donne de bons résultats pour de longues descriptions si les instructions ne comportent pas un nombre élevé d'actions opératives en parallèle. Si les instructions comportent de nombreux transferts simultanés, le compilateur peut être amené à découper certaines instructions soit parce que deux bus sont insuffisants, soit parce que certains cas de placement n'ont pas été pris en compte.
En conclusion, on peut dire que le système Apollon permet de compiler très rapidement de longues descriptions. En contrepartie le système ne donne pas forcément la meilleure solution. Si l'utilisateur est insatisfait il peut soit préplacer certains registres et relancer le compilateur afin d'obtenir une meilleure solution, soit modifier sa description. Une modification automatique de cette description est d'ailleurs prévue dans le cadre de l'évaluation [BEKK 86].

3.2.4. Optimisations

3.2.4.1. Optimisation du nombre de constantes

Une constante qui ne figure que comme opérande d'une opération binaire peut être placée juste à l'entrée de l'unité de traitement et n'être connectée qu'à un des bus de service (cf 4.2) de la partie opérative. On transforme les opérations binaires dont un des opérandes est une constante en des opérations unaires. On gagne ainsi un bus. D'autre part on peut placer n'importe quelle constante en entrée d'une unité de traitement et on peut regénérer cette constante en l'additionnant à zéro. Mais dans ce cas d'une part on bloque une unité de traitement et d'autre part on impose que le transfert se fasse au deuxième cycle donc de prendre 2 cycles pour le transfert d'une constante alors que la plupart des instructions ne comportant que des transferts peuvent s'exécuter en un cycle.
Donc si on veut augmenter les performances de la machine, on a intérêt à connecter les constantes sources de transferts simples à un ou l'autre des deux bus de la partie opérative et à connecter les constantes sources d'opérations binaires à un des deux bus de service qui relient les entrées de l'opérateur correspondant à ses tampons d'entrée.
Comme les constantes peuvent être dupliquées, les 2 approches peuvent être combinées et dans une même sous partie opérative, on pourrait avoir une instance d'une constante connectée à un des bus globaux et une autre instance de la même constante connectée à un bus local. Si on veut diminuer la surface de la partie

opérative, on ne dupliquera une constante que si c'est strictement nécessaire.
Actuellement les constantes ne sont connectées qu'aux bus globaux et elles peuvent être dupliquées. Une instance d'une constante opérande d'une opération binaire sera automatiquement placée dans la sous partie opérative où s'effectuera l'opération.
S'il s'agit d'un transfert simple, une instance de la constante sera placée dans la sous partie opérative où se trouve la destination. On peut remarquer qu'il est inutile de dupliquer des constantes lors de l'implantation d'instructions de moins de 3 transferts ou de moins de 5 transferts et comportant au moins une opération ou une action d'entrée-sortie car on est sûr que ces instructions sont exécutables.

3.2.4.2. Optimisation du nombre des opérateurs

Il serait souhaitable d'une part de tenir compte dès la phase d'allocation des ressources, des opérateurs universels (UALs) qui sont disponibles et d'autre part d'éviter de dupliquer inutilement certains opérateurs. En effet même si un autre choix pour le placement de l'opérateur que celui qui minimise le nombre de segments utilisés pourrait conduire à une situation de verrou, on peut remarquer que pour les instructions de moins de 5 tranferts (après réduction des opérations en transferts) comportant une opération il ne peut y avoir de conflit d'utilisation des bus puisqu'une partie opérative à 2 bus permet toujours d'exécuter 4 transferts en 2 cycles.
On peut donc modifier l'heuristique de tri des alternatives d'allocation des ressources d'une action opérative afin :

- d'éviter de dupliquer inutilement des opérateurs (c'est à dire quand il n'y a pas 2 opérations identiques à exécuter en même temps),
- d'essayer de regrouper les opérations qui ont des fonctionnalités voisines (addition, soustraction) ou qui peuvent être effectuées par le même opérateur universel (UAL, etc...) dans la même sous-partie opérative, lorsque le nombre de transferts élémentaires est strictement inférieur à 5.

3.2.5. Extension du modèle

3.2.5.1. Extension de la notion de sous partie opérative

Il semble nécessaire de revenir sur la notion de sous partie opérative.
On peut adopter la définition donnée précédemment, à savoir : une sous partie opérative est constituée d'un ensemble d'éléments de mémorisation, de traitement et de connexion avec l'extérieur reliés à une ou aux 2 portions de bus résultant de la segmentation des 2 bus de la partie opérative.
Le modèle à 2 bus segmentables est plus riche; en effet on a supposé:
- d'une part que les 2 bus étaient morcelés au même endroit, ce qui n'est pas nécessaire,

- d'autre part que les entrées et la (ou les) sortie(s) d'un opérateur devaient être situées dans la même sous partie opérative.

Nous avons vu que l'on peut économiser des interrupteurs entre les segments en découpant différemment les 2 bus et que l'on peut pipeliner des opérations binaires ou exécuter des opérations ternaires en mettant les opérateurs à cheval sur 2 sous parties opératives.

Pour tenir compte de ces extensions, on devra considérer dans l'algorithme d'allocation des ressources:

- que les entrées et la (ou les) sortie(s) d'un opérateur peuvent appartenir à 2 sous parties opératives adjacentes,
- d'autre part pour tenir compte de la segmentation différente des bus, on supposera au départ que les bus sont segmentés de la même façon et si pour les instructions déjà implantées, 2 segments adjacents peuvent être ressoudés, on vérifiera pour l'instruction en train d'être implantée si ces 2 segments sont encore soudables ou non.

Si lorsque toutes les instructions ont pu être implantées, certains segments sont ressoudables, on les ressoude à ce moment. La notion de sous partie opérative n'est alors plus significative.

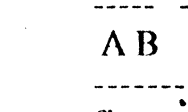


figure 3.44 : segmentation non uniforme des bus

En effet dans l'exemple précédent où les 2 bus n'ont pas le même nombre de segments, doit on considérer qu'il y a une ou deux sous parties opératives. Il est plus rigoureux de parler des segments de chacun des 2 bus et des registres, des constantes, des entrées-sorties d'opérateurs, des tampons de plots d'entrée-sortie connectés à chaque segment.

3.2.5.2. Modèle à 3 bus

L'ajout d'un troisième bus offre la possibilité d'exécuter des opérations ternaires. Les possibilités de parallélisme pour les autres types d'actions opératives sont augmentées considérablement. Cependant ce n'est que si l'on utilise des opérations ternaires que l'on exploite au maximum les possibilités de parallélisme.

Au niveau algorithmique, peu de modifications sont nécessaires pour adapter Apollon à une architecture 3 bus. Le problème primordial ne change pas : le système doit effectuer une partition des registres en un nombre minimal de sous parties opératives.

Deux procédures devront être essentiellement modifiées :

- l'allocation des connexions : un élément de la partie opérative pourra désormais être connecté à 3 bus,
- la recherche des conditions d'exécution des instructions opératives : les

possibilités de parallélisme offertes par l'ajout d'un bus remettent en cause l'utilisation des conditions d'exécution trouvées pour 2 bus. Par exemple les instructions non exécutables en 2 cycles que nous avons données au 3.1.2 sont maintenant exécutables en 2 cycles. Les conditions d'exécution pour une architecture 3 bus portent sur davantage d'actions opératives puisque les possibilités de parallélisme sont augmentées, mais ces conditions restent de même type comme nous allons le voir pour quelques instructions :

. exécution de 4 transferts : pusique l'on peut toujours effectuer 3 transferts en un cycle, on cherche une condition nécessaire et suffisante pour l'exécution de 4 transferts.
L'instruction (A:=B; C:=D; E:=F; G:=H;) n'est exécutable en un cycle que si et seulement si [A B] ∩ [C D] ∩ [E F] ∩ [G H] = 0.

. exécution de 2 opérations binaires : il faut et il suffit que les 4 opérandes ne soient pas sur la même sous partie opérative.

. exécution de 3 opérations binaires dont les opérandes sont tous différents: soit l'instruction
(X1 := A op1 B; X2 := C op2 D; X3 := E op3 F;)
Il faut et il suffit que les conditions nécessaires et suffisantes pour les trois couples d'opérations binaires soient satisfaites et d'autre part que :
[A B] ∩ [C D] ∩ [E F] ≠ [A B]
[A B] ∩ [C D] ∩ [E F] ≠ [C D]
[A B] ∩ [C D] ∩ [E F] ≠ [E F]

. exécution de 4 opérations binaires dont les opérandes sont tous différents : une condition nécessaire pour exécuter l'instruction
(X1 := A op1 B; X2 := C op2 D; X3 := E op3 F; X4 := G op4 H;)
est que les conditions portant sur tout triplet et donc sur tout couple d'opérandes soient satisfaites et d'autre part que
[A B] ∩ [C D] ∩ [E F] ∩ [G H] = 0.

.exécution de 2 opérations ternaires :
L'instruction (X1 := op1 A B C; X2 := op2 D E F;) n'est exécutable en 2 cycles que si et seulement si:
1° cas : les opérandes sont tous différents :
[A B C] ∩ [D E F] ≠ [A B C]
[A B C] ∩ [D E F] ≠ [D E F]
où [A B C] représente l'union des ensembles de sous parties opératives comprises entre A et B, B et C et finalement C et A.
[A B C] = [A B] U [B C] U [C A]

2° cas : un opérande est commun
(X1 := op1 A1 B1 C; X2 := op2 A2 B2 C;)
[A1 B1] ∩ [A2 B2] ≠ [A1 B1]
[A1 B1] ∩ [A2 B2] ≠ [A2 B2]

3° cas : les opérations ont 2 opérateurs en commun
(X1 := op1 A1 B C; X2 := op2 A2 B C;)
A1 et A2 ne doivent pas être sur la même sous partie opérative.

4. Génération des fichiers de masques de la partie opérative

Le compilateur Apollon génère les masques de la partie opérative à partir de l'architecture obtenue après la première phase de la compilation, c'est à dire à partir de sa décomposition en sous parties opératives alignées, chacune de ces sous parties opératives étant caractérisée par ses éléments de mémorisation et de traitement connectés à la structure 2 bus.
La génération des masques est doublement simplifiée:
1) d'une part au modèle architectural est associé un modèle topologique,
2) d'autre part on dispose d'une bibliothèque de cellules précaractérisées [SCHO 85], [GALL 84], [SUZI 81].
La partie opérative est organisée selon le principe du "bit-slice" [SUZI 81] ou tranche de bit. La partie opérative résulte de l'empilement de n tranches d'un bit. A chaque élément constitutif de la partie opérative, on fait correspondre une cellule ou un assemblage de cellules de la bibliothèque.

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

4.1. Modèles topologique et électrique de la partie opérative

4.1.1. Modèle topologique

La partie opérative est formée d'un bloc rectangulaire, constitué par l'empilement d'autant de tranches horizontales qu'il y a de bits. La tranche de numéro i résulte de la juxtaposition des cellules du ième bit des différents blocs fonctionnels. Chaque bloc fonctionnel de n bits est constitué par l'empilement vertical de n cellules identiques. Le bloc opératif peut être considéré comme une matrice dont les lignes correspondent aux différents bits et les colonnes correspondent aux différents éléments fonctionnels.
Les tranches ont la même hauteur. De plus d'une part les cellules d'une tranche ont la même hauteur et les cellules d'un élément ont la même largeur et d'autre part les éléments de la partie opérative ont en principe le même nombre de bits. Il en résulte que le circuit est dense puisqu'il n'y a pas de trous dus à la différence de hauteur des cellules d'une même tranche ou à la différence de largeur des cellules d'un même élément ou surtout à la différence de hauteur des éléments de la partie opérative.

MODELE TOPOLOGIQUE

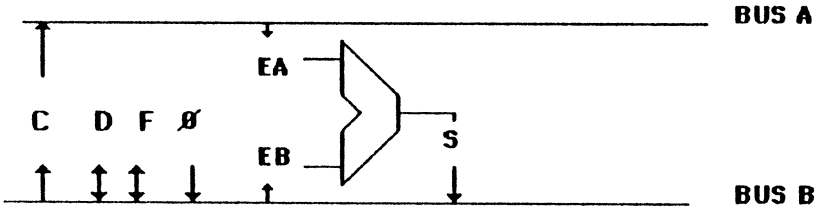
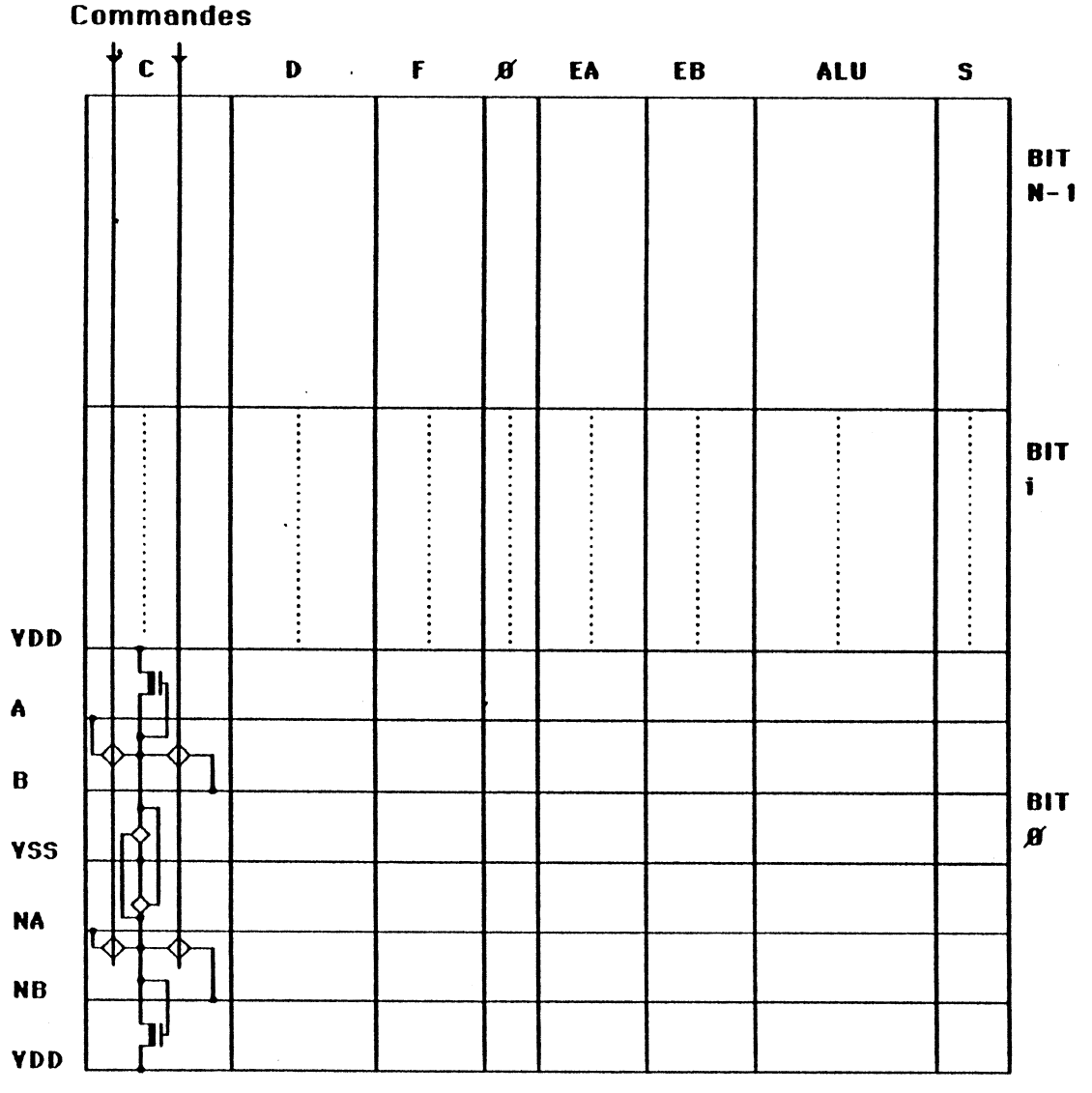


Schéma architectural



PLAN DE MASSE DE LA PARTIE OPERATIVE

figure 3.45

En pratique toutefois l'utilisateur peut être amené à introduire des registres qui portent sur un nombre de bits inférieur à celui des mots de longueur maximum traités par la partie opérative. D'autre part le nombre des indicateurs arithmétiques est en général inférieur au nombre de bits de la partie opérative et il y a une certaine place perdue au niveau du registre d'état.
En outre les cellules des différents bits d'un élément fonctionnel ne sont pas forcément identiques. Souvent la première et la dernière sont différentes. Elles peuvent être presque toutes différentes dans le cas d'éléments fonctionnels complexes (chaîne de propagation de la retenue).
Au lieu de générer à chaque fois les instructions d'assemblage d'un élément fonctionnel, on préfère écrire une fois pour toutes des fonctions d'assemblage pour les éléments fonctionnels complexes comme l'UAL. On se contentera alors de générer l'appel à cette ou ces fonctions avec les paramètres adéquats.

Il n'y a pas de problème de placement puisque tous les éléments d'une sous partie opérative ne sont connectés entre eux que par l'intermédiaire de 2 bus qui les traversent et que d'une part toutes les cellules d'une même tranche ont la même hauteur et d'autre part toutes les cellules d'un élément fonctionnel ont la même largeur. Les éléments d'une sous partie opérative sont donc placés côte à côte dans n'importe quel ordre. Le découpage en sous parties opératives est effectué au cours de la génération de l'architecture.

Le seul problème de placement que l'on ait rencontré concerne non pas l'assemblage de la partie opérative mais l'assemblage de l'interface électrique entre la partie opérative et la partie contrôle. Nous avons constaté que l'interface électrique est plus large que la partie opérative elle même si on met bout à bout les différentes cellules de l'interface électrique.
Plusieurs solutions peuvent être adoptées pour résoudre ce problème:

1) on redessine des cellules moins larges.

2) on minimise le nombre des multiplexeurs et on met dos à dos les cellules de même type afin de partager les contacts aux bus d'horloge et d'alimentation.

3) on dispose les cellules sur 2 étages [JIN 86] : on minimise ainsi l'espace de routage car les connecteurs qui se correspondent se trouvent rapprochés et le nombre maximal de coudes nécessaire au routage est diminué. On arrive ainsi à diminuer la surface globale. On obtient aussi un bloc rectangulaire plus facile à intégrer dans le circuit global qu'un bloc irrégulier. C'est cette dernière solution qui a été choisie.

4.1.2. stratégie d'utilisation des couches

Comme on dispose de plusieurs couches de conducteurs isolées les unes des autres, et pour occuper une surface minimale, on dessine les cellules de façon à ce que les fils de connexion soient placés au dessus des transistors. Ainsi on ne perd pas de place pour connecter les cellules entre elles.
Pour cela on adopte la stratégie de dessin suivante: les cellules doivent être

transparentes dans les 2 directions aux fils de connexion de 2 couches différentes. En NMOS la couche inférieure, le polysilicium, est utilisée pour canaliser le flux de contrôle, ce qui permet aux fils de commande d'attaquer directement les grilles des transistors qui contrôlent le flux des données.
La couche supérieure, l'aluminium est utilisée pour canaliser le flux des données. Les bus de la partie opérative sont donc réalisés en aluminium.

Il n'y a pas de problème de routage puisque d'une part les connexions entre les éléments de la partie opérative sont assurées par 2 bus parallèles qui traversent toutes les cellules et d'autre part les cellules ont été dessinées de telle façon que toutes les autres connexions entre cellules (par exemple les connexions horizontales entre les cellules d'un même élément fonctionnel et les connexions verticales entre les cellules de tranches voisines) se fassent par aboutement. C'est d'ailleurs la principale raison pour laquelle un tel modèle a été choisi : il n'y a aucune place perdue à cause des connexions. Toutes les connexions entre les éléments se font par l'intermédiaire de bus qui sont situés au dessus de la logique; toutes les autres connexions se font par aboutement.

4.1.3. Modèle électrique

Passons en revue les différents composants de la partie opérative

4.1.3.1. Les bus

Les bus sont les moyens de communication entre les éléments de la partie opérative. Ils sont en général longs par rapport aux éléments constitutifs de la partie opérative. C'est pourquoi ils ont une capacité de charge relativement importante. Cette charge est la caractéristique la plus importante des bus vis à vis du comportement électrique. En effet on choisit le conducteur le moins résistif, le métal pour réaliser les bus puisqu'ils représentent les liaisons les plus longues. La résistivité du métal est de plusieurs ordres de grandeur inférieure à celle des autres conducteurs et des éléments actifs. C'est pourquoi, on peut négliger en général la résistance des bus lorsqu'ils sont réalisés en aluminium. On obtient la capacité du bus en faisant la somme de toutes les capacités liées au bus et de sa propre capacité.

Il existe différentes solutions pour réaliser des bus bidirectionnels. On peut :
- soit utiliser un seul fil :
  - s'il n'y a pas de temps mort, on peut utiliser soit des portes 3 états soit un système et-ou-non réparti avec pull-up unique,
  - si l'on accepte qu'il y ait un temps mort, on peut utiliser soit le système de précharge-décharge ou le système de mise à seuil.
- soit on utilise 2 fils: on utilise alors le système à décharge-précharge.

L'utilisation de bus bifilaires à précharge présente évidemment le désavantage

d'exiger une phase supplémentaire à chaque cycle, ce qui entraîne un temps mort pour le calcul. Les avantages sont toutefois plus nombreux :

- il n'est pas nécessaire d'introduire un amplificateur de sortie pour chaque élément qui attaque le bus : on peut utiliser un seul amplificateur par bus. Si l'on ne dispose que d'un bus unifilaire, il est nécessaire d'adapter la taille du dispositif de sortie du registre à la longueur du bus. Le système à bus différentiels permet d'utiliser les mêmes cellules pour des applications relativement différentes grâce à l'emploi d'amplificateurs de données sur les bus.
- un registre peut utiliser le même chemin pour la lecture et l'écriture. Si on veut n'utiliser qu'un seul bus, on est obligé de rendre le point mémoire dissymétrique en introduisant une résistance ou un interrupteur synchronisé par l'inverse de l'horloge d'écriture entre le point de lecture et le point d'écriture.

L'emploi de connexions bidirectionnelles permet donc d'utiliser le même chemin pour la lecture et l'écriture, ce qui constitue un gain de place. Ce gain se traduit uniquement par un gain de place au niveau de la partie opérative et non au niveau de la partie contrôle car le nombre de commandes nécessaires est le même. Il peut même y avoir une augmentation de la surface de l'interface entre la partie opérative et la partie contrôle dans le cas où le multiplexeur lecture-écriture est plus large que l'amplificateur de lecture et l'amplificateur d'écriture réunis, (ce qui est le cas de la bibliothèque NMOS d'Apollon) et où l'on n'utilise pas le même multiplexeur pour la lecture-écriture de plusieurs points-mémoire.

En conclusion on peut dire que l'utilisation de bus complémentés permet de réduire la taille des registres et de diminuer le nombre de connexions.

C'est pourquoi le système Apollon est basé sur l'utilisation de bus bidirectionnels bifilaires à précharge.

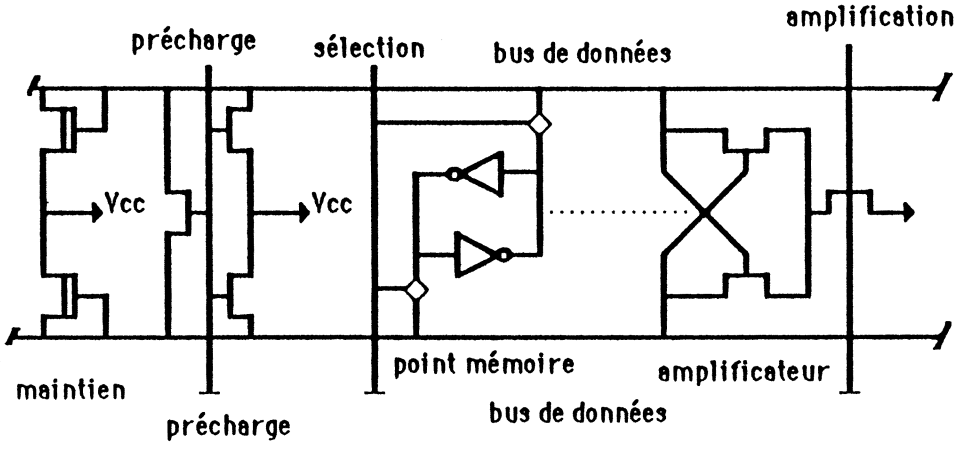


figure 46 : structure d'un bus d'Apollon

Un transfert entre 2 registres se déroule en 4 phases:
T1: phase de précharge et d'égalisation
T2: phase de sélection en lecture

T3: phase d'amplification
T4: phase de sélection en écriture

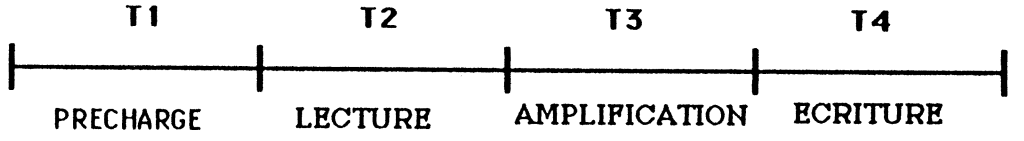


figure 3.47 : les 4 phases d'un transfert

T1
La précharge est faite à l'aide de 2 transistors enrichis. A la fin de la précharge, la tension des 2 bus est de 3,8V pour un signal de précharge de 5V. Le transistor se bloque pour VGS=1,2V à cause de l'effet substrat. La dimension des transistors de précharge dépend de la valeur de la capacité du bus et du temps de précharge désiré. Si le bus n'est pas utilisé immédiatement après la précharge, des transistors de maintien très résistifs qui fournissent juste un courant pour compenser les fuites seront ajoutés.
Les 2 bus sont aussi reliés entre eux par un transistor enrichi qui permet d'égaliser les valeurs de précharge.

T2
Les registres sont connectés aux 2 bus complémentés si bien que lorsque l'on connecte le registre aux 2 bus pendant la phase suivant la phase de précharge, l'un des 2 bus est déchargé. La tension de l'autre n'est pas modifiée puisqu'elle a déjà la bonne valeur.

T3
Après la phase de lecture, on amplifie le déchargement d'un des 2 bus.

T4
Lorsqu'un registre est connecté aux bus après la phase T3, les bus sont complémentés et c'est le bus qui impose sa valeur au registre.

4.1.3.2. Les registres

Il existe deux types de mémorisation:
- la mémorisation dynamique: la valeur n'est mémorisée que pendant un certain temps. Il est nécessaire de rafraîchir périodiquement l'information pour ne pas la perdre.

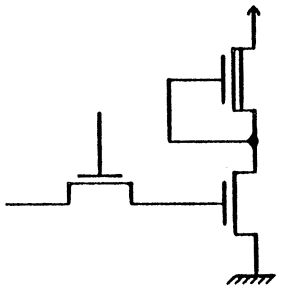


figure 3.48 : mémorisation dynamique

La charge mémorisée est stockée au niveau de la grille du transistor.
- la mémorisation statique : deux inverseurs sont rebouclés; l'information est regénérée automatiquement.
Nous considèrerons dans la suite des registres statiques uniquement. Lorsqu'on ne dispose que d'un bus, il est nécessaire d'affaiblir la boucle dissymétriquement pour pouvoir écrire dans le registre. La boucle peut être affaiblie en permanence à l'aide d'un transistor déplété dont la grille est reliée à la source

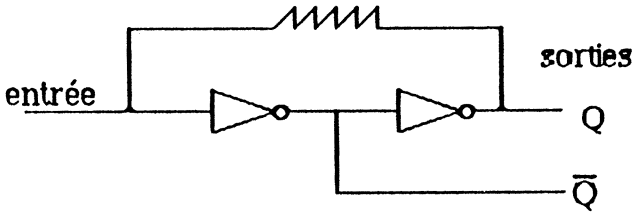


figure 3.49 : registre statique avec résistance

ou on peut introduire dans la boucle un transistor enrichi dont la grille est reliée au complément du signal d'écriture.

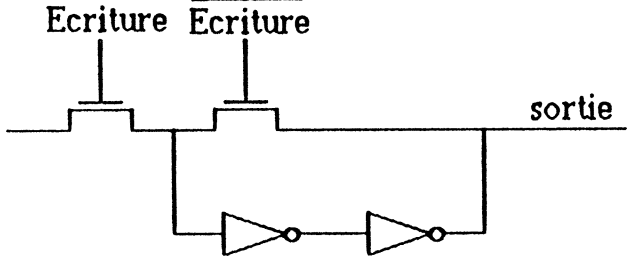


figure 3.50 : registre statique avec interrupteur

Si on utilise un bus complémenté, il n'y a besoin que d'une seule connexion physique pour la lecture et l'écriture et il n'est pas nécessaire d'affaiblir la boucle dissymétriquement. Il y a encore d'autres avantages dans le système des bus complémentés:
- la consommation: des inverseurs à très faible consommation peuvent être utilisés,
- la valeur d'une variable et son complément seront disponibles sur les bus.

4.1.3.3. Les constantes

On convertit les entiers en base 2. Pour chaque chiffre binaire de la constante en base 2, on met un interrupteur relié à la masse sur le bus non complémenté si la valeur du bit est 1 sinon on met un interrupteur relié à la masse sur le bus complémenté si la valeur du bit est 0.

4.1.3.4. Les opérateurs

Les opérateurs sont des fonctions combinatoires qui exécutent des transformations sur les données. Les opérateurs sont donc tous des fonctions logiques et on les réalise avec des portes logiques ou des montages électriques équivalents.

4.2. Présentation de la bibliothèque NMOS

Toutes les cellules de la bibliothèque ont la même hauteur qui est de 54 lambdas et ont la même structure de bus. Dans une tranche il y a 2 bus complémentés, soit 4 bus unifilaires, 2 bus de service (il en faut un pour pour chacune des entrées de l'UAL) et 2 bus d'alimentation (masse et 5V).
La masse est située au milieu alors que le "5 volts" est situé en haut. En fait chaque tranche utilise aussi le bus à 5 volts d'en dessous. C'est pourquoi on doit rajouter un fil d'aluminium en dessous de la première tranche.

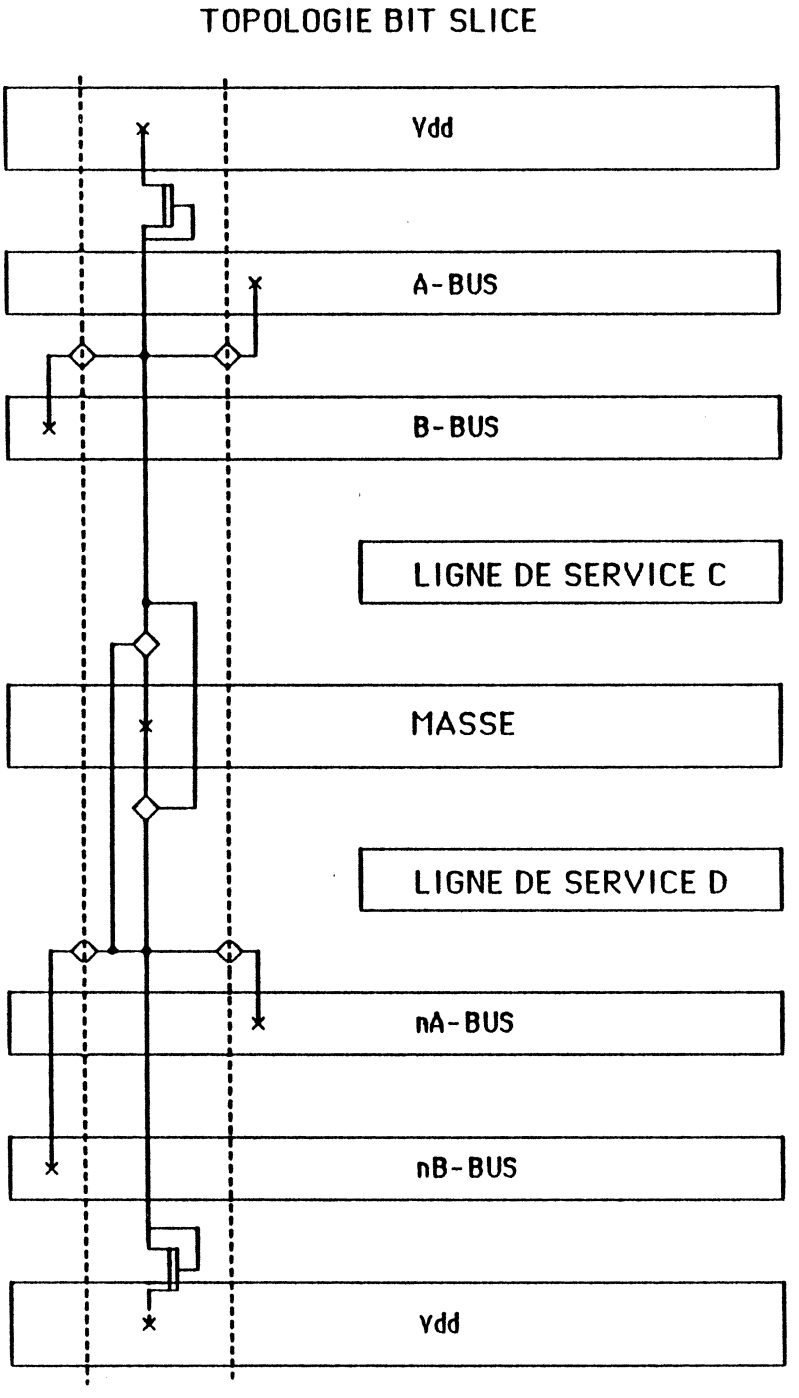


figure 3.51 : structure des bus des cellules de la bibliothèque d'Apollon

Il n'existe pas de cellules spécifiques pour les bus en effet les bus sont placés au dessus de la logique. Les segments de bus ne sont pas inclus dans chacune des cellules de la partie opérative. Des peignes portant des connecteurs à l'emplacement des bus sont placés de chaque côté de la partie opérative et indiquent à l'assembleur de silicium qu'il faut connecter les connecteurs en vis à vis.

Il existe plus d'une centaine de cellules différentes, la plupart ne contenant que quelques rectangles. Il y a en fait une trentaine de cellules de base:

- les cellules assurant le fonctionnement électrique des bus:
  .précharge
  .amplification
  .connexion interbus
- les éléments de mémorisation
  .registre simple accès
  .registre double accès

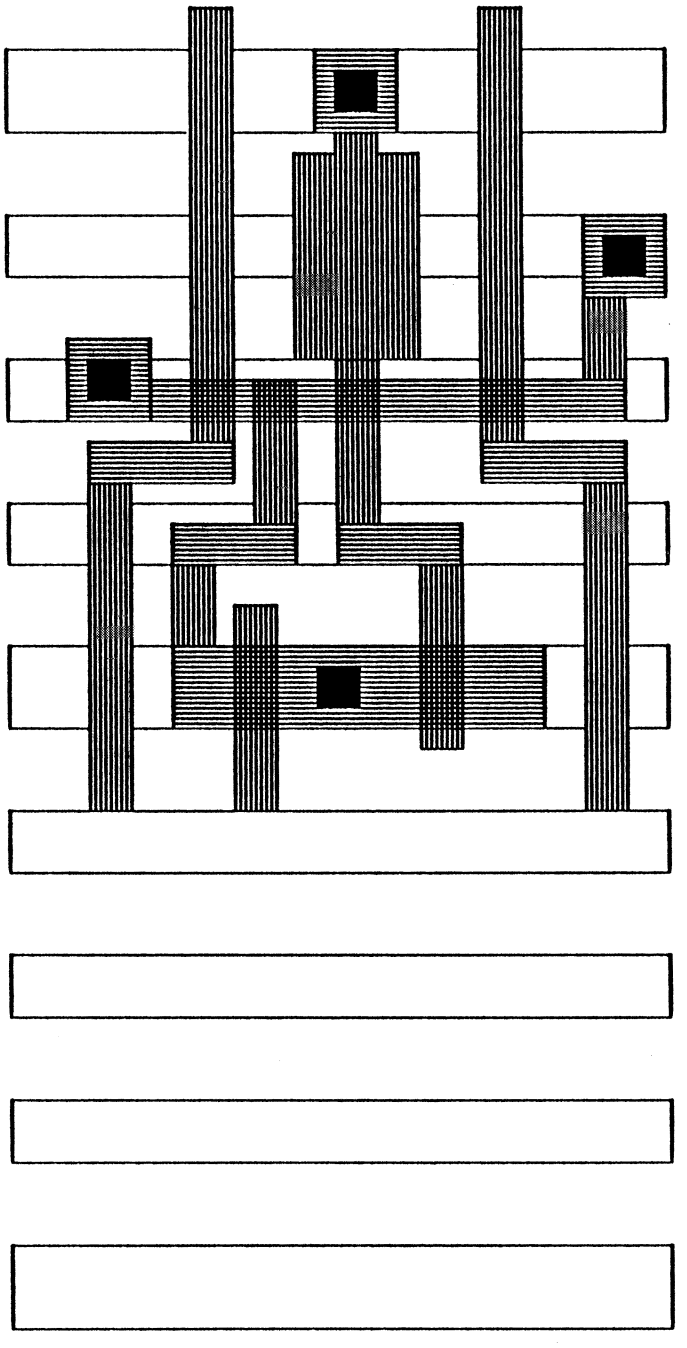


figure 3.52 : dessin des masques d'un demi point mémoire double accès

(le point mémoire complet peut être obtenu par symétrie)

.registre écrit à partir d'un bus unifilaire (tampon de sortie de l'UAL)
.constante (à simple accès)
- les tampons des plots d'entrée-sortie

.tampon d'entrée
.tampon d'entrée-sortie
- les opérateurs
.additionneur et chaîne de propagation de la retenue
.UAL
.décaleur
- les amplificateurs des commandes
.simple amplificateur
.amplificateur de validation
.multiplexeur lecture-écriture

Les cellules de la bibliothèque sont présentées dans le mémoire de R. Gallais [GALL 84] et dans la thèse de A.A. Suzim [SUZI 81].

4.3. Présentation de l'assembleur de silicium Lubrick

On donne ici les spécifications d'un assembleur de parties opératives pour Apollon. Ces spécifications ont été établies à partir de l'assembleur de silicium de l'équipe d'architecture des ordinateurs, Lubrick [SCHO 83] et de [JERR 86]. Pour une description du système Lubrick on se rapportera à [SCHO 83], [SCHO 85] et [GALL 84].
Un assembleur de silicium se doit de répondre aux besoins des concepteurs de circuits intégrés complexes fortement hiérarchisés.
L'assembleur doit donc permettre de construire de nouvelles figures à partir des figures de base dessinées à la main et des figures déjà construites par assemblage.
De cette façon, le concepteur pourra hiérarchiser son circuit, construire des figures plus complexes à partir des figures de base, les répéter ou leur appliquer un traitement géométrique quelconque.

4.3.1. Assemblage de la partie opérative

Les masques de la partie opérative résultent de l'assemblage de cellules fonctionnelles.
Chaque cellule fonctionnelle doit être conçue pour pouvoir être assemblée facilement avec ses voisines. A cette fin, les problèmes de connexion sont résolus au niveau des cellules de base et non au niveau de l'assemblage.
On doit réserver aux bus des canaux de passage à l'intérieur de chaque cellule. On utilise les couches inférieures pour réaliser les fonctions logiques alors que l'on utilise les couches supérieures pour réaliser les connexions si bien que les bus sont au dessus des portes.
Une cellule doit donc respecter un certain nombre de contraintes [SCHO 83]:
- avoir une structure de bus prédéfinie donc une hauteur fixe.
- avoir des connexions directes uniquement.

Les connexions par bus se font au niveau de l'assemblage global et non au niveau de chaque cellule puisqu'un passage a été réservé pour chaque bus.

Il ne reste plus au concepteur qu'à gérer les connexions locales entre 2 cellules. Les problèmes de connexion ayant été résolus par le concepteur, l'assemblage de la partie opérative peut être fait au moyen d'un assembleur rudimentaire.

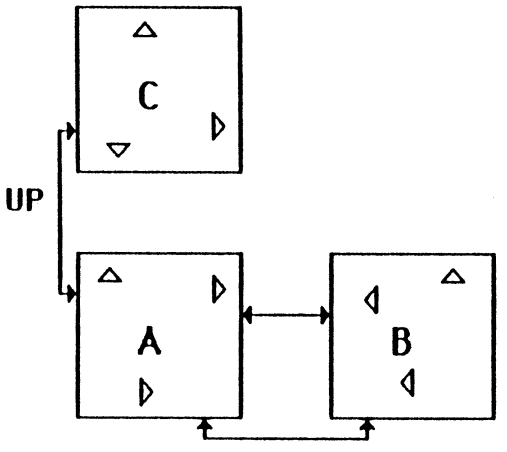
4.3.2. Spécifications d'un assembleur de parties opératives

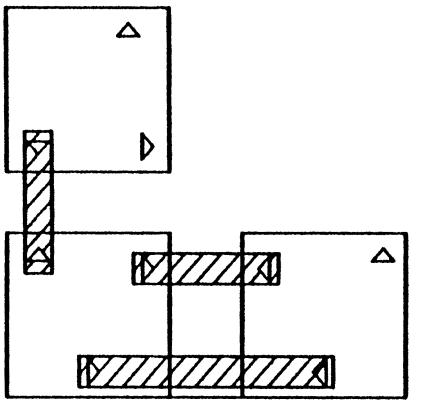
L'assembleur de parties opératives doit avant tout permettre de placer et de tasser des cellules les unes à côté des autres, horizontalement ou verticalement mais alors dans ce cas sans les tasser tout en s'assurant que les règles de dessin sont vérifiées. Les connnexions directes entre cellules sont tracées. Pour effectuer des connexions plus complexes, des fonctions spécifiques de routage doivent être utilisées.

LUBRICK

ASSEMBLEUR DE SILICIUM

DE BRIQUES LUCIE

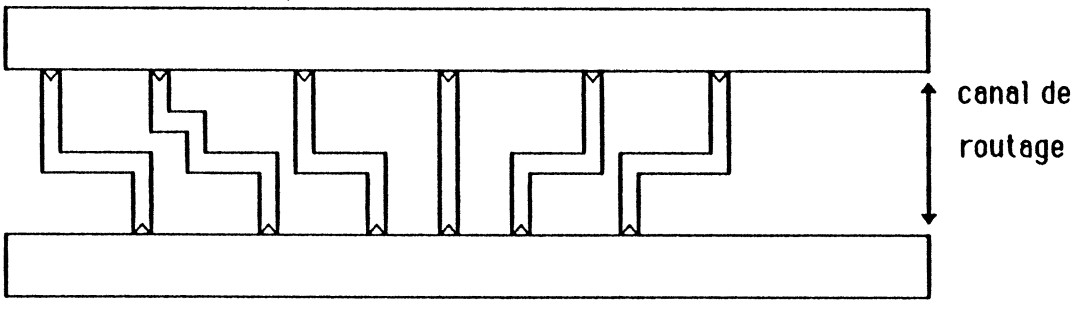




CONNEXIONS DIRECTES

PLACEMENT

```
i = openfig ('example')
i = right (right (i, getfig ('A'), 1, 0), getfig ('B'), 1, 0) ;
i = up (i, getfig ('C'), 1, 0) ;
i = closefig (i) ;
```



ROUTAGE PAR DEVOIEMENT

figure 3.53 : le système Lubrick

4.3.2.1. Définition des cellules de base

Chaque cellule peut être définie de 2 façons:
- par sa structure interne: sa décomposition en figures géométriques de base, (en Lucie on ne peut décrire que des rectangles), figures composées internes et figures composées externes
- par son interaction avec l'environnement : c'est ce seul aspect de la définition d'une cellule qui nous intéresse pour la définition d'un assembleur de silicium. Pour l'assembleur, une cellule est définie par la donnée de :

1. son encombrement qui est un rectangle (de hauteur fixe pour une cellule de la partie opérative).
2. ses frontières:

Pour chaque frontière (nord - sud - est - ouest) et pour chaque niveau technologique (aluminium - polysilicium - etc ...), on extrait de la description géométrique de la cellule la liste des segments correspondant aux côtés des rectangles les plus à l'extérieur pour chaque position le long de la frontière considérée.

3. ses connecteurs :

Un connecteur est défini par:
- son orientation (nord - sud - est - ouest)
- sa position
- sa largeur
- son niveau technologique

On peut rajouter dans cette définition un type et/ou un nom.
Les types peuvent être utilisés pour sélectionner les connecteurs actifs lors de l'assemblage. Les noms peuvent être utilisés pour générer un schéma des connexions résultant de l'assemblage.

4.3.2.2. Mise en oeuvre de la hiérarchie

L'assembleur de silicium doit permettre de hiérarchiser la description du circuit et donc donner au concepteur les moyens de définir des figures composées et d'imbriquer des définitions de figures composées.
Cette hiérarchisation peut être mise en oeuvre de 2 façons différentes:
- soit on dipose de fonctions n-aires qui permettent d'assembler n figures simultanément de telle sorte que l'on puisse assembler les cellules d'une même figure composée en une seule fois.

A chaque appel d'une fonction d'assemblage correspond la création d'une nouvelle figure composée.
L'imbrication des figures se traduit par l'imbrication des appels de fonctions d'assemblage.
- soit on ne dispose que de fonctions binaires qui permettent d'assembler les cellules 2 à 2, ou du moins on ne dispose pas de fonction permettant l'assemblage des cellules

d'une même figure en une seule fois, ce qui est le cas de Lubrick.
On doit alors introduire des fonctions de début et de fin de construction de cellules (openfig et closefig).
Les fonctions openfig et closefig servent en quelque sorte de délimiteurs.
L'appel de la fonction openfig provoque la création d'une figure vide dont le nom est l'argument de la fonction. Seuls les opérateurs de construction qui lui sont appliqués peuvent la faire passer dans un état non vide.
L'appel de la fonction closefig termine la construction. Il est aussi utile d'introduire une fonction qui permet de conserver la figure construite pour une réutilisation sans redéroulement du même programme d'assemblage.
Les opérateurs de base de construction de Lubrick sont binaires, c'est à dire qu'ils ne permettent d'assembler que 2 cellules en même temps.
L'assemblage de n cellules est décomposé en n-1 assemblages binaires successifs. On commence à assembler les 2 premières cellules, puis on assemble la troisième au résultat de l'assemblage précédent et on répète cette opération jusqu'à ce que les n cellules soient assemblées. Le premier argument d'un opérateur de construction doit toujours être le résultat de la dernière fonction de construction appliquée, et si aucun opérateur n'a encore été appliqué, le premier argument doit être le résultat de la fonction openfig. De même, l'argument de la fonction closefig doit être le résultat de la dernière fonction de construction appliquée.
A chaque appel de fonctions géométriques (répétition, symétrie, rotation) ou de fonctions de routage (routage par dévoiement, routage par croisement) correspond la création d'une figure composée interne ainsi bien sûr qu'à chaque définition de nouvelles figures par la fonction openfig.
L'imbrication de 2 figures se traduit cette fois par l'utilisation du résultat de la fonction closefig pour la figure imbriquée, dans la construction de la figure englobante. Les fonctions de base de construction de figures sont des fonctions de placement.

4.3.2.3. Fonctions de construction de figures

La fonction de base de l'assembleur est:

place(fig1, fig2, dir, offset, ta)

La fonction "place" place la figure "fig2" à la droite de la figure "fig1" si dir=right et en haut si dir=up tout en les tassant l'une contre l'autre en tenant compte de la garde.
Le paramètre "offset" permet de décaler la figure "fig2" dans le sens perpendiculaire à la direction de l'assemblage. Ce décalage n'est en principe pas nécessaire si toutes les cellules ont la même hauteur.
Le paramètre "ta" (type array) est un vecteur de types qui permet de spécifier les connecteurs actifs lors de l'assemblage. Les connecteurs non actifs ne sont pas pris en compte et sont éliminés de la figure résultant de l'assemblage.

Une autre façon de spécifier les connexions est d'introduire la notion de transparence. Si deux figures ont des connecteurs de même niveau en vis à vis sur une largeur supérieure ou égale à la largeur minimum du niveau, on trace la connexion. Si à un connecteur de la première figure correspond une transparence de même niveau et de largeur suffisante, on conserve le connecteur jusqu'à sa résolution lors d'un prochain assemblage.
Sinon, soit on élimine le connecteur de la description, soit on déclare une erreur.
Il est aussi intéressant de disposer d'une fonction qui permet de placer une figure par rapport à une autre sans calcul de garde, soit:

offset(fig1,fig2,dx,dy,ta)

où fig1, fig2 et ta ont la même signification que pour la fonction place.
La garde n'est pas calculée, la figure fig2 est décalée de dx dans la direction parallèle à l'axe des x et de dy dans la direction parallèle à l'axe des y.

4.3.2.4. Opérations géométriques

Pour construire un bloc fonctionnel de n bits à partir d'une cellule fonctionnelle de 1 bit, on utilise une fonction de placement à répétition.
La fonction

rep(fig, dir, repfactor, ta)

permet de placer "repfactor" figures "fig" côte à côte dans le sens spécifié par le paramètre "dir" (horizontal ou vertical).
On peut en outre utiliser des transformations géométriques simples comme la symétrie ou la rotation.

4.3.2.5. Fonctions de routage

Deux fonctions de routage sont très utiles pour connecter les amplificateurs de validation des commandes à la partie opérative: la fonction river_route et la fonction channel_route.

La fonction

river_route(fig1, fig2, dir, offset, ta)

connecte les figures "fig1" et "fig2" par dévoiement. La figure "fig2" est placée à droite de "fig1" si dir=right et en haut si dir=up. Le paramètre "offset" permet de décaler la figure "fig2" par rapport à "fig1" dans le sens perpendiculaire à "dir".
La largeur minimum du canal de routage est calculée automatiquement. Le niveau des connexions est fixé par le niveau des connecteurs des figures "fig1" et "fig2". Le

paramètre "ta" permet de spécifier les connecteurs actifs.
Pour chacun des niveaux des connecteurs de la première figure, le nième connecteur actif de "fig1" est relié au nième connecteur actif de "fig2" de même niveau. Si pour chaque niveau, les 2 figures n'ont pas le même nombre de connecteurs en vis à vis, l'assembleur signale une erreur.

La fonction

channel_route(fig1, fig2, dir, offset, level1, level2, ca, ta)

permet de réaliser un routage croisé bicouche. "level1" spécifie le niveau de la couche de base des connexions sortant des connecteurs de "fig1" et aboutissant aux connecteurs de "fig2". "level2" spécifie le niveau de la couche qui permet de faire des ponts entre les fils de niveau "level1". Le paramètre ca (connector array) est un tableau de correspondance entre les connecteurs de "fig1" et "fig2".
Exemple: ca[1]:=3;
Le premier connecteur de "fig1" est relié au troisième connecteur de "fig2". Les autres paramètres ont le même rôle que dans la fonction river_route. Toutes les connexions sont générées à l'aide de rectangles dont les côtés sont parallèles aux axes du repère de base.

4.3.2.6. Fonctions d'accès aux caractéristiques des cellules de base

Il est utile de disposer de fonctions d'accès aux caractéristiques des cellules pour permettre la programmation de fonctions spécifiques aux besoins de l'utilisateur.
On définit les fonctions suivantes:
- la fonction delta(fig, dir)

où dir spécifie l'axe des "x" ou l'axe des "y", qui permet d'accéder à l'encombrement d'une figure.
- la fonction getconn(fig,dir)

qui permet d'accéder aux connecteurs dont la direction est dir.
- la fonction getbound(fig,dir)

qui permet d'accéder aux segments de la frontière de direction dir.
En outre il peut être intéressant de mettre à la disposition de l'utilisateur la fonction de calcul de la garde entre 2 cellules.

4.3.2.7. Fonctions d'assemblage de l'interface électrique entre partie opérative et partie contrôle

Pour assembler les amplificateurs de validation de façon automatique, il est intéressant d'introduire la fonction "build".

build(fig, dir, mode, extend, fa)

Le paramètre "fa" (figure array) est un tableau de correspondance entre les types des connecteurs de direction "dir" (nord - sud - est - ouest) de la figure "fig" et les figures que l'on veut assembler automatiquement à partir de ces connecteurs.
A chaque connecteur de direction "dir" de la figure "fig", on associe une figure par l'intermédiaire de son type.
Le paramètre "mode" permet de spécifier si l'assemblage est relatif (mode=true), c'est à dire que l'on place côte à côte les figures correspondant aux connecteurs de direction "dir", dans le sens perpendiculaire à "dir", ou absolu (mode=false): les figures sont placées au niveau de chaque connecteur.
Enfin "extend" est un paramètre d'extension. Si extend≠-1, on écarte les figures de façon à ce que la largeur totale de la figure construite soit égale à "extend".
Toujours pour l'assemblage et la connexion des amplificateurs de validation, il est nécessaire de disposer d'une fonction permettant de répartir les amplificateurs sur plusieurs étages et de les connecter car la largeur de l'étage d'amplification-validation est souvent supérieure à la largeur de la partie opérative.

split_river_route(fig, dir, leftcomb, rightcomb, pass1, pass2, fa)

"dir" spécifie l'orientation des connecteurs actifs de la figure "fig".
"fa" est le tableau de correspondance type-figure.
"leftcomb", "rightcomb", "pass1" et "pass2" sont des cellules de l'étage d'amplification.
"leftcomb" est le peigne gauche de lignes d'alimentation et d'horloge; "rightcomb" est le peigne droit correspondant. "pass1" et "pass2" sont des cellules de passage qui permettent de faire remonter des fils de connexion de 2 niveaux différents à travers le ou les étages d'amplification.
La fonction split_river_route répartit les cellules de l'interface électrique sur un ou plusieurs étages qui ont tous la même largeur qui est celle de la partie opérative.
On peut encore définir une autre fonction analogue à la précédente mais qui permet en outre de faire passer à droite les fils de connexion correspondant à un type donné de cellules. On l'utilise pour faire sortir à droite les fils de compte-rendu de façon à bien séparer les flux de contrôle et de compte-rendu au niveau de la partie contrôle.

4.3.3. Mise à la disposition de l'utilisateur d'un langage de programmation

Le système Lubrick est constitué de plusieurs modules pascal. De cette façon, le concepteur peut utiliser tout pascal, ce qui peut lui permettre par exemple de générer des cellules paramétrables ou de se définir des fonctions d'assemblage pour la génération de blocs fonctionnels complexes (UALs).

4.4. Algorithme de génération des fichiers de masques

4.4.1. Principe de l'assemblage

La partie opérative est formée d'un bloc rectangulaire résultant de l'empilement de tranches d'un bit.
Les sous parties opératives sont placées côte à côte. Les éléments de chaque sous partie opérative peuvent être placés dans n'importe quel ordre.

4.4.1.1. Spécification des modules de génération de la partie opérative

Nous avons déja présenté au chapitre 2 la notion de module de génération. Nous présentons dans cette partie les spécifications structurelles, c'est à dire que nous donnerons les paramètres des différents modules qui permettent de spécifier quelles sont les cellules de base que l'on doit assembler. Il n'y a pas besoin de donner des informations topologiques ou électriques puisque d'une part l'organisation topologique est figée et d'autre part on utilise des cellules précaractérisées.

En ce qui concerne la partie opérative de façon générale, les modules générateurs devront être paramétrés par le nombre de bits. Si l'utilisateur peut définir des registres portant sur un nombre de bits variable, il est clair que le modèle topologique adopté (bit slice) ne donne de bons résultats que si tous les composants ont le même format. D'autre part les cellules placées dans la première tranche et la dernière tranche de la partie opérative sont en général légèrement différentes des cellules placées dans les tranches intermédiaires.

Exemple :
Ainsi la construction d'un décaleur à l'aide de Lubrick peut être programmée de la façon suivante (cf 4.3) :

```
begin
    i:=openfig ('shift ');
    i:=up (i,getfig ('shif ø '),1,0);
    j:=getfig ('shifi ');
    for j:=1 to n-2 do i:=up(i,j,1,0);
    i:=up(i,getfig('shifn'),1,0);
    i:=closefig (i);
end;
```

Les cellules de la bibliothèque NMOS dont nous disposons utilisent 2 bus d'alimentation reliés au 5 Volts, celui de la cellule qui se trouve à la limite supérieure et celui de la cellule d'en-dessous.
Il est donc nécessaire de rajouter un bus d'alimentation à la limite inférieure de la première tranche du bit slice.

Les bus sont situés au-dessus de la logique. Une place leur est réservée lors du dessin des cellules. Mais les bus ne sont pas dessinés car ils traversent tous les composants d'une sous-partie opérative. Les bus sont générés lors de l'assemblage de la partie opérative. Des peignes d'alimentation sont placés de part et d'autre de la partie opérative. Lors de l'assemblage des peignes les bus d'alimentation sont tracés; ces peignes portent aussi des connecteurs pour le tracé des bus de données. On intercale entre 2 sous parties opératives voisines un bloc d'interface constitué d'un empilement de cellules d'interconnexion portant sur les côtés des connecteurs pour le tracé des bus de données.

Une cellule d'interconnexion peut contenir soit aucun, soit 1 ou 2 interrupteurs selon que les sous parties opératives fonctionnent indépendamment l'une de l'autre ou qu'elles peuvent échanger des données par l'intermédiaire d'un ou des 2 bus. Lors de l'assemblage d'une sous partie opérative et du bloc d'interconnexion placé à sa droite, les bus de données sont tracés.

L'utilisation de cellules d'interface pour le tracé des bus est nécessaire du fait que d'une part les cellules ne sont pas simplement aboutées mais aussi tassées les unes contre les autres dans le sens du flot de données et d'autre part que la notion de transparence n'existe pas en Lubrick.

L'assembleur de silicium calcule la garde minimum entre 2 cellules lorsque l'on fait appel à la fonction de placement latéral (dans le sens du flot de données).

Module générateur de registres

Les paramètres de la génération des spécifications des masques d'un registre sont les suivants :

- le nombre de bits
- le ou les bus auxquels le registre est connecté (les connexions dépendent du modèle architectural. Les registres dont l'élément de base est de type bascule ont une entrée et une sortie distinctes. Si les registres sont simplement connectés entre eux par des multiplexeurs sans passer par des bus, on doit préciser pour chaque registre le nombre de sources différentes à partir desquelles on peut écrire dans ce registre. Des multiplexeurs sont utilisés pour sélectionner les chemins d'accès des bus aux registres).
- les bits qui sont accessibles à la partie contrôle
- les champs de bits qui sont accédés simultanément par la partie opérative.

Par exemple si on veut remettre à zéro les 8 bits de poids forts d'un registre 16 bits, 2 commandes sont nécessaires : l'une pour sélectionner les bits de poids faibles l'autre pour sélectionner les bits de poids forts.

Remarque

On a toujours besoin de 2 commandes pour accéder séparément à 2 champs d'un registre. Mais il existe une autre solution. Si plusieurs registres ont la même structure d'accès (mettons accès poids faibles et accès poids forts séparés), on peut se

servir d'un "fool it" (commme dans le MC68000) pour invalider les données sur les bits qui ne sont pas accédés.
Exemple : R 8:8:=0;
Pendant la phase de lecture, les bits 0 à 7 du bus utilisé pour la remise à zéro sont invalidés. Pour cela on connecte les fils du bus complémenté à un transistor de charge afin de maintenir les 2 fils préchargés à 3,8 Volts si bien que lorsque l'on connecte le registre R en écriture seuls les bits 8 à 15 seront modifiés car les bits 0 à 7 du bus ne sont pas complémentés et le bus ne peut imposer sa valeur au registre.

Cette solution permet de diminuer le nombre de commandes dont on a besoin pour accéder par champ aux registres d'une même sous partie opérative. Si plusieurs registres sont accédés pour un même champ de bits par un même bus, il suffit de placer un "fool it" sur ce bus pour les bits n'appartenant pas à ce champ et on n'a besoin que d'une seule commande.

Cette solution n'est cependant pas d'une portée très générale : elle n'est applicable qu'aux parties opératives à bus préchargés et complémentés.

Module générateur de constantes

Les paramètres du module de génération des constantes sont :
- le nombre de bits,
- la valeur de la constante,
- le bus auquel elle est connectée (une constante ne peut être que simple accès par construction).

Module générateur d'opérateurs

Les modules générateurs d'opérateurs sont certainement les plus complexes car les opérateurs sont constitués d'un assez grand nombre de cellules différentes. Il y a autant de modules différents qu'il y de types d'opérateurs.

Nous n'avons à notre disposition qu'une UAL qui peut être dégradée en un additionneur ou un incrémenteur, et un décaleur qui permet de décaler les bits d'un mot d'une position vers la droite ou vers la gauche. Ces opérateurs permettent d'effectuer n'importe quelle opération car on peut toujours décomposer une opération complexe en une suite d'opérations élémentaires (opération logique, addition et décalage). Toutefois cette décomposition ralentit l'exécution des opérations complexes. Il peut être intéressant de disposer d'un décaleur généralisé qui permet de décaler les bits d'un mot de plusieurs positions vers la droite ou vers la gauche en un seul cycle.

De Man [DeMa 86] utilise 5 types d'unités de traitement pour la réalisation de systèmes de traitement du signal multiprocesseurs.
- un multiplieur - accumulateur
- un diviseur / multiplieur parallèle série (une division ou une multiplication portant sur n bits est effectuée en n cycles)

- un décaleur / additionneur / comparateur
- une UAL
- une unité de calcul d'adresses

D'autre part il peut être intéressant d'effectuer un certain nombre d'opérations en série sans passer par l'intermédiaire de tampons dans la mesure où ces opérations peuvent s'effectuer dans le laps de temps imparti.

Il est possible de connecter directement la sortie d'un opérateur à l'entrée d'un autre opérateur sans passer par l'intermédiaire des 2 bus de la partie opérative. Par exemple on peut placer des décaleurs aux entrées ou à la sortie de l'ual.

Spécifions les paramètres d'un module générateur d'opérateurs. Un certain nombre de paramètres vont dépendre du type d'opérateurs générés.

- le nombre de bits : il est nécessaire de préciser non seulement le nombre de bits maximum sur lequel portent les opérations mais aussi tous les champs de bits différents sur lesquels portent les opérations à effectuer. En effet on peut très bien envisager d'effectuer une addition 8 bits et une addition 16 bits sur un même additionneur 16 bits. La génération des comptes rendus ainsi que l'injection de la retenue entrante posent des problèmes de réalisation. En effet, les comptes rendus (retenue, débordement et signe) sont générés au niveau du bit de poids le plus fort et le test d'égalité à zéro est réalisé par une porte NOR à n entrées où n est le nombre de bits. La retenue entrante nécessaire à la soustraction :

$$x - y = x + \bar{y} + 1$$

est introduite au niveau du bit de poids le plus faible.
On doit donc préciser si l'on effectue des opérations portant sur un nombre restreint de bits afin de pouvoir générer les comptes-rendus et injecter la retenue entrante. Toute combinaison d'opérations portant sur des nombres de bits différents n'est d'ailleurs pas souhaitable.

- le nombre des entrées et des sorties ainsi que leurs connexions aux bus

- le type d'opérateur : par exemple dans le cas d'un décaleur généralisé, il faut préciser de combien de positions on décale les données. Pour un type composite (décaleur / additionneur / comparateur) on devra donner les opérations qui sont utilisées. Si on ne fait pas de décalage, il est inutile d'inclure un décaleur. Une autre question importante à se poser est : est-ce que l'on veut mettre en série 2 opérateurs? ou a-t-on besoin de tous les comptes-rendus? Si on ne fait pas de comparaison, il est inutile de générer une porte NOR à n entrées.

On peut aussi avoir besoin d'un opérateur de manipulation de bits qui permette d'exécuter des manipulations arbitraires sur les bits individuels des mots donnés [BLAC 85]. On peut se contenter d'utiliser un décaleur généralisé qui permet d'effectuer des manipulations de ce type en un nombre réduit de cycles.

Nous avons vu que lors de la transformation de la description de la partie opérative en une description d'assemblage, on fait correspondre à chacun des composants de la partie opérative un module de génération de masques spécifique. Cependant on ne fait pas correspondre à chaque opérateur un module différent. Les opérations peuvent être regroupées à l'intérieur de boîtes à opérations du type UAL/décaleur (Apollon) ou décaleur / additionneur / comparateur (Cathedral2 [DEMA 86]). Lors de l'allocation des opérateurs il est souhaitable de regrouper dans la même sous-partie opérative les opérateurs qui ont certains composants en commun ou qui font partie de la même boîte à opérations. Les opérateurs d'une sous partie opérative peuvent nécessiter plusieurs boîtes à opérations si bien qu'on peut faire correspondre aux opérateurs d'une sous partie opérative plusieurs modules de génération de boîtes à opérations.

Le terme boîte à opérations se veut plus général que le terme UAL : une boîte à opération est un opérateur universel qui permet d'effectuer plusieurs opérations (arithmétiques, logiques, décalages...).

Module générateur de connexions aux plots d'entrée-sortie

Les paramètres de ce module sont les suivants :
- nombre de bits
- direction de la connexion (entrée, sortie ou connexion bidirectionnelle)
- le ou les bus connectés aux plots en lecture et/ou en écriture

Module générateur de connexions entre sous parties opératives voisines

Les paramètres de ce module sont les suivants :
- nombre de bits
- et pour chaque bus le type de connexion :

1) aucune : les 2 segments sont toujours utilisés de façon indépendante,

2) interrupteur : les 2 sous parties opératives peuvent échanger des données par l'intermédiaire de ce bus,

3) pas de coupure : les 2 bus ne sont pas segmentés de façon uniforme.

Module générateur de bus

Il n'y a pas en fait de module pour générer des bus au sens propre du terme puisque les bus font partie des cellules. Par contre il faut rajouter dans chaque sous partie opérative les composants nécessaires au fonctionnement des bus : préchargeur et amplificateur.

4.4.1.2. Réalisation des modules générateurs des éléments de la partie opérative

Les éléments d'une sous partie opérative peuvent être placés dans n'importe quel ordre mais en pratique l'ordre de placement des éléments de la partie opérative dépend :

1) des contraintes imposées par le langage de description de masques utilisé (Lucie). En particulier les nombres de figures externes et internes appelées dans une figure sont limités [PAIL 85].

2) des possibilités de minimisation de la surface occupée par ces éléments.

On peut gagner de la place en plaçant judicieusement les éléments les uns par rapport aux autres en ce qui concerne:

- les tampons des plots: les tampons des plots d'entrée-sortie doivent être reliés aux plots qui sont situés à la périphérie du circuit. Si le tampon est placé au milieu de la partie opérative, il est possible de relier les différents bits du tampon aux plots correspondants par l'intermédiaire d'une nappe de fils verticaux. Ces fils doivent être en polysilicium puisqu'ils croisent les bus de la partie opérative qui sont en aluminium.

Soit n le nombre de bits du tampon, l la largeur minimale d'un fil de polysilicium, d la distance minimale entre 2 fils de polysilicium. On augmente donc la largeur de la partie opérative de n(l+d) lambdas soit de 4n lambdas avec les règles du CMP [DELO 86] si on utilise des tampons de plots à sorties verticales.

Par conséquent on cherchera à placer les tampons aux extrémités de la partie opérative lorsque cela est possible. On peut alors utiliser des fils de liaison horizontaux puisque ces fils sortent de la partie opérative sans passer par dessus d'autres éléments puisqu'ils sont situés aux extrémités de la partie opérative. Comme il n'y a que 2 extrémités, 2 tampons seulement pourront avoir des sorties horizontales à condition bien sûr qu'il y en ait un dans chacune des sous parties opératives situées aux extrémités.

- les registres: il est possible de placer les cellules de sélection des registres de façon à partager les contacts aux bus et ainsi de minimiser la garde entre les registres.

- les constantes: on peut aussi minimiser la garde entre les constantes en retournant dans le sens de la largeur une cellule sur deux de façon à minimiser la garde.

- les amplificateurs de validation: 2 amplificateurs de même type peuvent partager leurs contacts sur les bus d'horloge et d'alimentation si on en retourne un des deux dans le sens de la largeur.

5. Interface de la partie opérative avec la partie contrôle

5.1. Minimisation du nombre des commandes remontant vers la partie contrôle

Toutes les commandes de la partie opérative ne remontent pas vers la partie contrôle. Certaines sont uniquement synchronisées par une horloge comme pour la

précharge : à chaque cycle les bus sont préchargés automatiquement pendant la phase de précharge.
Les commandes de certains des tampons d'entrée-sortie des opérateurs peuvent aussi être déclenchées automatiquement.
Puisque les tampons d'entrée et de sortie d'un opérateur sont uniquement utilisés pour charger les opérandes et stocker le résultat et les indicateurs arithmétiques avant de les transférer dans leurs destinations respectives, les commandes d'accès en écriture à ces tampons peuvent être uniquement synchronisées par une horloge d'écriture à condition que ces tampons ne puissent pas être accédés de plusieurs façons différentes.
C'est le cas des tampons de sortie qui sont connectés chacun à une des 2 sorties de l'opérateur. Les tampons d'entrée ne peuvent être synchronisés par l'horloge d'écriture sans validation de la partie contrôle que s'ils ne sont connectés qu'à un seul des 2 bus de la partie opérative.
Autrement une commande est nécessaire pour déterminer le bus à partir duquel on écrit dans le tampon.
Des commandes peuvent être constantes dans la mesure où toutes les fonctionnalités des unités de traitement de la bibliothèque ne sont pas utilisées. En effet il n'existe pas forcément dans la bibliothèque d'unité de traitement qui ne permette d'exécuter que les opérations à effectuer dans une même sous partie opérative.

La minimisation du nombre des commandes passe avant tout par la minimisation du nombre des éléments de la partie opérative que l'on duplique: constantes, opérateurs, multiplexeurs lecture-écriture et par la minimisation du nombre des connexions des registres aux bus et des connexions entre segments de bus adjacents.
Le problème de la minimisation du nombre des constantes, opérateurs et des connexions a déjà été abordé dans les paragraphes sur la synthèse de la partie opérative. Le problème est complexe.

5.1.1. Minimisation du nombre de multiplexeurs lecture-écriture

Le modèle électrique utilisé (bus complémentés à précharge) permet d'utiliser le même chemin électrique pour la lecture et l'écriture des points mémoire.
Le point mémoire est lu s'il est connecté au bus juste après la précharge des bus et il prend la valeur du bus s'il est connecté après que les 2 bus ont pris des valeurs opposées.
On voit donc que s'il ne faut qu'un interrupteur pour connecter un registre au bus en lecture ou en écriture, il faut 2 commandes de contrôle car l'interrupteur peut être:
- soit fermé en lecture
- soit fermé en écriture
- soit ouvert.

La commande de l'interrupteur est validée par une cellule de l'interface électrique entre la partie opérative et la partie contrôle qui a 4 entrées:
- les horloges de lecture et d'écriture

- la commande du multiplexeur lecture-écriture
- la commande de connexion du registre au bus

Cette cellule est constituée par un multiplexeur des horloges de lecture et d'écriture accolé à un amplificateur de validation. La sortie du multiplexeur est validée par la commande de connexion registre-bus.

Il est possible d'utiliser le même multiplexeur lecture-écriture pour plusieurs interrupteurs à condition que si des interrupteurs de ce groupe sont actifs au même moment, alors ils soient tous ouverts en lecture ou tous ouverts en écriture.

Il y a toutefois une limite à la mise en commun des multiplexeurs. Il faut que les cellules de l'interface électrique soient suffisamment transparentes pour que l'on puisse faire passer des bus supplémentaires pour connecter les amplificateurs des commandes de connexion des interrupteurs d'un même groupe au même multiplexeur lecture-écriture.

Or les cellules de la bibliothèque dont nous disposons ne peuvent laisser passer qu'un bus supplémentaire. Par conséquent si les groupes d'interrupteurs qui peuvent partager un même multiplexeur lecture-écriture ne forment pas des intervalles disjoints, il y aura un recouvrement ce qui nécessitera soit la création de bus supplémentaires, donc l'étirement des cellules en hauteur, soit l'utilisation d'un routeur bicouche de façon à ce que l'on puisse grouper les amplificateurs des commandes de connexion des interrupteurs d'un même groupe, soit simplement une permutation des registres si les registres de l'intervalle de recouvrement appartiennent à une même sous partie opérative.

La minimisation du nombre des multiplexeurs lecture-écriture est doublement intéressante:

- on diminue la largeur de l'interface électrique entre partie opérative et partie contrôle,
- on diminue en même temps le nombre de commandes venant de la partie contrôle.

Si on raisonne au niveau d'une sous partie opérative, deux registres ne peuvent être connectés au même segment, l'un en lecture et l'autre en écriture que s'il existe un transfert entre ces 2 registres.

Les opérations unaires ou binaires n'entrent pas en ligne de compte car elles font intervenir les tampons d'entrée et de sortie des opérateurs qui ne sont accédés par définition qu'en écriture, au premier cycle pour les entrées et qu'en lecture, au second cycle pour la ou les sorties.

On peut donc utiliser la méthode de décomposition en cliques maximales de [TSEN 83] pour déterminer les interrupteurs reliés au même bus qui peuvent partager un même multiplexeur. On construit pour chaque sous partie opérative et pour chaque bus un graphe de "non-transferts" où tous les registres qui sont connectés en lecture-écriture au bus considéré sont reliés entre eux sauf ceux pour lesquels il existe un transfert qui utilise le bus considéré.

Les groupes de registres provenant de la partition du graphe en cliques disjointes peuvent utiliser le même multiplexeur pour leurs connexions au bus.

Il se peut que l'intersection de 2 ensembles de registres correspondant chacun à la mise en commun d'un mutiplexeur pour des connexions au bus supérieur pour l'un

et au bus inférieur pour l'autre ne soit pas vide.
Rien n'empêche qu'un registre soit connecté en lecture-écriture aux 2 bus. Si l'intersection contient plus d'un élément, on ne pourra mettre en commun un multiplexeur pour chacun des 2 groupes que si l'on utilise un routeur bicouche. En effet s'il n'y a qu'un élément dans l'intersection, il suffit de réordonner convenablement les registres. Par contre si on considère les interrupteurs sur l'un ou l'autre des 2 bus d'une sous partie opérative, il ne peut y avoir de registre qui ait une connexion dans un groupe et l'autre connexion dans un autre groupe que si un registre peut être à la fois lu et écrit au cours du même cycle.

Si l'on désire minimiser le nombre de multiplexeurs lecture-écriture en mettant en commun les multiplexeurs pour les connexions aux segments des 2 bus, le complément du graphe des transferts ne peut plus être utilisé comme précédemment pour effectuer une partition en cliques.
Pour construire le graphe de compatibilité de fonctionnement des interrupteurs entre les registres et les 2 bus segmentés, on procèdera de la manière suivante: on ne reliera 2 interrupteurs i1 et i2 que si l'ensemble des cycles où l'un des interrupteurs est ouvert en lecture et l'ensemble des cycles où l'autre est ouvert en écriture sont disjoints et vice-versa.
On ne considèrera bien sûr que les interrupteurs qui sont utilisés à la fois en lecture et en écriture. A partir du graphe de compatibilité obtenu, on appliquera comme précédemment les algorithmes de [TSEN 83].

5.1.2. Paramétrisation des commandes

Il existe une autre façon de diminuer le nombre des commandes remontant vers la partie contrôle. On peut chercher à diminuer le nombre des éléments de la partie opérative. On peut aussi essayer de coder les commandes. En effet si n commandes sont nécessaires pour contrôler la partie opérative, seul en fait un petit nombre des 2^n combinaisons de commandes sont utilisées.
On peut coder les commandes de plusieurs façons:
1) on regroupe les commandes qui prennent simultanément la même valeur.
Exemple: minimisation du nombre des multiplexeurs lecture-écriture
2) on recode les codes de sélection des opérations des unités de traitement s'ils ne sont pas optimum. L'UAL de notre bibliothèque a 5 commandes qui permettent d'exécuter 10 fonctions: 4 commandes suffisent.
3) on cherche à mettre en évidence des groupes de commandes tels qu'à chaque cycle au plus une de ces commandes est active. On peut alors ne faire remonter vers la partie contrôle que log2(n+1) (arrondi à l'unité supérieure) fils au lieu de n, car le déclenchement de la commande active s'il y en a une au cycle considéré revient à la donnée du numéro du fil de commande correspondant et à zéro si aucune commande n'est à activer.
Comme on ne peut lire qu'une constante sur chaque bus, 2 constantes connectées au même bus ne peuvent jamais être lues simultanément. On en déduit que l'on peut

coder les commandes de ces constantes comme on l'a indiqué précédemment.
De même on ne peut lire qu'un seul registre par segment de bus, on peut donc inclure les connexions en lecture uniquement des registres dans l'ensemble des connexions des éléments de mémorisation connectés au même segment de bus dont les commandes peuvent être codées puisqu'un seul de ces éléments de mémorisation (constante ou registre connecté en lecture uniquement) peut être lu en un cycle.
On a vu qu'un tampon d'entrée connecté aux 2 bus doit être commandé de la partie contrôle, car si le tampon peut toujours être connecté en écriture même si on ne fait pas d'opération, il faut spécifier le bus que l'on veut connecter au tampon d'entrée. On peut donc coder les 2 commandes correspondant aux 2 connexions aux bus à l'aide d'un seul fil car il n'y a que 2 cas possibles de connexion :
- le tampon est connecté au bus inférieur,
- le tampon est connecté au bus supérieur.

On convient alors que si on ne fait pas d'opération, on connecte arbitrairement le tampon au bus supérieur. On gagne ainsi une commande.

5.2. Stockage des commandes de la partie opérative

Les commandes doivent être stockées dans un organe de mémorisation dans la partie contrôle. Plus on a réussi à paramétriser les commandes, moins il y a de commandes à stocker. Les commandes sont alors décodées puis amplifiées et validées au niveau de l'interface entre partie contrôle et partie opérative avant d'attaquer les grilles des transistors de commande.

Il existe plusieurs méthodes de stocker ces commandes:

- soit on sépare le séquencement de la génération des commandes,
- soit on génère simultanément le nouvel état de l'automate de contrôle et les commandes de la partie opérative.

Dans le premier cas, il est possible de ne stocker qu'une fois une même instruction à la différence du second cas où l'on doit stocker les commandes d'une instruction autant de fois qu'elle apparaît dans l'algorithme de départ.
Si l'on veut ne stocker qu'une fois chaque instruction opérative, on peut générer non pas directement les commandes de la partie opérative, mais un numéro qui correspond au numéro de l'instruction à exécuter. Au lieu de générer plusieurs fois les mêmes commandes, on génèrera plusieurs fois le même numéro ce qui est moins coutêux. On peut d'ailleurs envisager d'autres solutions pour réduire le nombre de commandes stockées. Par exemple on ne génère pas le numéro d'une instruction mais les numéros des actions qui la composent.

Etudions plus en détail comment les commandes sont générées selon que la génération est directe ou non.

1°) Génération directe des commandes

Le compilateur Apollon introduit un nouveau niveau de séquencement : celui des cycles opératifs. L'appel d'une instruction opérative par le dernier niveau de contrôle doit être remplacé par les commandes de la partie opérative pour tous les

cycles nécessaires à l'exécution de cette instruction. Actuellement le nombre de cycles d'une instruction est limitée à 2. Le séquencement des instructions opératives en un ou 2 cycles doit être généré à ce niveau.

2°) Génération indirecte des commandes

Plusieurs solutions peuvent être adoptées :

2.1°) On génère un numéro d'instruction opérative. Le numéro est ensuite décodé dans un PLA qui doit aussi assurer le séquencement de l'instruction en 1 ou 2 cycles. Cette solution permet de ne garder qu'une copie des différentes instructions opératives.

2.2°) On génère non plus un numéro d'instruction mais un numéro de cycle. Ce numéro est décodé dans le PLA de génération des commandes. Le séquencement des cycles de l'instruction est assuré par l'étage supérieur à l'étage de génération des commandes. Cette 2ème solution est du même type que la première. Elle ne présente un intérêt que si les cycles qui constituent les instructions opératives sont souvent identiques alors que les instructions elles-mêmes ne le sont pas.

2.3°) On génère non pas un numéro, mais les numéros des actions qui constituent une instruction opérative. Cette solution donne de bons résultats si les instructions sont constituées de combinaisons d'un faible nombre d'actions de base. Il est nécessaire en outre qu'une action soit toujours exécutée de la même façon quelles que soient les autres actions qui sont exécutées en même temps, ce qui n'est pas toujours le cas avec Apollon (cf 3.1.2.2).

Cette solution donne le meilleur résultat pour celles des instructions LDS (PINST) (cf chapitre 1) qui sont constituées de CASE en parallèle. Les différentes alternatives de chaque CASE peuvent être réalisées par autant de monômes dans le PLA de stockage des commandes de la partie opérative. Les commandes de la partie opérative sont obtenues en "sommant" les commandes correspondant aux actions sélectionnées par les conditions de sélection des différents CASE.

Si chaque CASE a ni alternatives,

on ne mémorise que $(\sum_{i=1}^{p} ni)$ monômes au lieu de $n1*n2* \ldots *np$.

Actuellement SYCO adopte la première solution : les commandes sont générées directement. Les appels d'instructions opératives sont remplacés directement par les commandes de la partie opérative.

6. Evaluation des performances

6.1. Evaluation de la surface de la partie opérative

La partie opérative désigne ici à la fois les n tranches d'un bit du chemin de données du microprocesseur et la ou les tranches des amplificateurs des commandes de la

partie contrôle.
La surface de la partie opérative dépend:

1) du parallélisme spécifié au niveau de l'algorithme de description du microprocesseur,

2) de la librairie de cellules utilisée pour générer le fichier des masques.

Nous nous intéresserons ici uniquement à l'influence du choix de la librairie de cellules sur la taille de la partie opérative.

6.1.1. Evaluation de la surface de la partie opérative

6.1.1.1. Evaluation de la surface du chemin de données

6.1.1.1.1. Evaluation de la hauteur du chemin de données

La partie opérative résulte de l'empilement de n tranches fonctionnelles d'un bit. Chaque tranche a une hauteur fixe. Cette hauteur dépend de la structure de bus de la partie opérative.
Prenons comme exemple la structure de bus de la librairie Nmos utilisée actuellement par le compilateur (voir figure 3.51).
La hauteur est de 54 lambdas. La hauteur d'une tranche est fonction :

1) du nombre de bus (bus fonctionnels : les bus bifilaires du modèle d'Apollon et les bus de service, bus d'alimentation),

2) de la largeur de chaque bus : largeur minimum imposée par la technologie (ici 3) ou largeur adaptée à la charge électrique des bus (ici 4 pour les bus d'alimentation),

3) de l'espace entre chaque pair de bus voisins : distance minimum entre deux fils de métal ou distance majorée pour tenir compte des contacts.

La hauteur du chemin de données est donc égale au produit du nombre de bits par la hauteur d'une tranche.

6.1.1.1.2. Evaluation de la largeur du chemin de données

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

1) du nombre de registres déclarés par l'utilisateur,

2) du nombre de constantes utilisées (certaines constantes peuvent être dupliquées),

3) du nombre de sous parties opératives nécessaires pour exécuter l'algorithme du microprocesseur et de la nature des opérations effectuées sur chaque sous partie opérative,

4) du nombre de fils remontant vers la partie contrôle : le nombre des fils de compte-rendu est fixé par l'utilisateur,

5) du nombre de fils descendant vers les plots. Ce nombre dépend:

- du nombre de bus d'entrée-sortie,
- du nombre de bits de chacun de ces bus,

- de la position des tampons d'entrée-sortie à l'intérieur de la partie opérative. S'ils sont placés sur les côtés, les sorties des tampons peuvent être horizontales et dans ce cas, aucune place n'a besoin d'être réservée pour faire passer des fils vers le bas.

6) du nombre de connexions des différents éléments aux 2 bus du chemin de données et des connexions entre sous parties opératives.

Un registre peut être à simple accès ou à double accès.

Les tampons d'entrée et de sortie des opérateurs ainsi que les tampons d'entrée-sortie des plots peuvent être eux aussi connectés à un ou deux bus.

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

Il est hors de question de calculer la largeur de la partie opérative de façon exacte. Cette largeur dépend d'un grand nombre de paramètres qui ne seront finalement connus que lorsque l'assemblage de la partie opérative sera achevée. En particulier il faut tenir compte non seulement de la taille des cellules fonctionnelles mais aussi de l'espacement entre cellules voisines, imposé par les règles technologiques. Toutefois, il est possible de fournir une estimation de cette largeur à partir d'une part du nombre et de la nature des différents éléments fonctionnels, de leurs connexions aux bus et des connexions entre bus et d'autre part de la décomposition des éléments fonctionnels en cellules de base et de la taille de ces cellules.

Une UAL est composée de plusieurs blocs fonctionnels :

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

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

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

En Nmos les gardes peuvent prendre les valeurs 1, 2 ou 3. On prendra comme valeur de la garde soit la valeur par excès 3 soit la valeur moyenne 2.

Evaluation fine

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

Evaluation grossière :

On ne dispose que d'une description comportementale : l'ensemble des actions à exécuter par la partie opérative. Pour calculer la largeur de la partie opérative, on doit au moins connaître le nombre de boîtes à opérations car ce sont les éléments de la partie opérative qui sont le plus coûteux en surface. Sans information particulière, on prendra ce nombre égal au nombre maximum d'opérations en parallèle dans une instruction opérative. A partir de là, on peut obtenir une première approximation.
Pour obtenir une première approximation, on considèrera:

- que chaque registre est connecté aux deux bus,
- qu'il y a 2n constantes où n est le nombre de constantes utilisées dans la description. Comme on ne dispose pas de constantes double accès, on duplique les constantes : on en connecte une sur un bus et l'autre sur le second bus. Remarquons que le nombre de constantes est borné par 2s.n où s le nombre de sous parties opératives,
- que toutes les sous parties opératives voisines sont connectées entre elles par les 2 bus,
- qu'il y a sur chaque sous partie opérative une boîte à opérations de taille fixe. On prendra pour valeur de la largeur d'une boîte à opérations une valeur moyenne.
- que tous les tampons d'entrée-sortie des plots ont des sorties verticales.

6.1.1.2. Evaluation de la surface occupée par les amplificateurs des commandes

On suppose que la largeur de l'interface électrique est la même que celle du chemin de données.
On ne cherche plus qu'à évaluer la hauteur.
La hauteur dépend :

- de la hauteur du ou des canaux de routage,
- de la hauteur d'une tranche; comme pour le chemin de données, on utilise des cellules de même hauteur que l'on assemble en tranches.
- du nombre de tranches.

Si les amplificateurs sont assemblés côte à côte, la largeur de la tranche d'amplification est en général supérieure à la largeur du chemin de données. Si on répartit les amplificateurs sur plusieurs étages de façon à ne pas dépasser la largeur du chemin de données, on obtient une forme rectangulaire. Par contre la hauteur de l'interface électrique entre partie opérative et partie contrôle est augmentée. La surface de cette interface est augmentée mais on obtient un bloc plus régulier qui sera connecté plus facilement au reste du circuit.

La hauteur d'une tranche est fixe et dépend de la technologie et de la stratégie d'implémentation. Le nombre de tranches est dans la pratique égal à 2.
Il ne reste alors qu'à estimer la hauteur des canaux de routage. Si l'on utilise un

routage monocouche, la hauteur est égale au produit du pas de la couche choisie (le pas est égal à la somme de la largeur minimum d'un fil et de la distance minimum entre 2 fils - on se place dans le cas où tous les fils ont la même largeur) par le nombre maximum de lignes horizontales. Ce nombre dépend de la position relative des connecteurs nord et sud.

6.1.2. Largeur des cellules de la bibliothèque Nmos

Les largeurs sont données en lambdas.

6.1.2.1. Eléments simples du chemin de données

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

6.1.2.2. Boîte à opérations

La boîte à opérations se compose de 3 parties :
1) les entrées : il y en a une ou deux; une entrée peut être multiplexée : la première s'il y en a deux; la seconde entrée peut incorporer les constantes 0 et 1 : on utilise un interrupteur pour sélectionner l'entrée ou une des deux constantes.
- entrée : 24
- multiplexeur : 16
- interrupteur : 10
- constante : 10

2) le corps de la boîte à opérations : cela peut être un incrémenteur, un additionneur ou une ual. Chacun de ces opérateurs comprend 2 parties :
- une partie qui lui est propre :
  - incrémenteur : 12
  - additionneur : 41
  - ual : 56

et une partie commune calculant la retenue : 54 (64 si on ajoute l'indicateur arithmétique zéro : c'est un nor distribué).
De plus on peut ajouter un décaleur :
- qui décale à droite : 23

- qui décale à gauche : 19
- qui décale dans les deux sens : 32

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

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

6.1.2.3. Interface électrique

amplificateur standard : 17
amplificateur avec multiplexeur incorporé : 53
cellule de passage : 2
contact : 4
Toutes les cellules de la tranche des amplificateurs ont une hauteur de 79 lambdas.

6.2 Compromis surface-vitesse

Apollon génère des parties opératives permettant d'exécuter des actions opératives en parallèle. Une question que l'on peut se poser est dans quelle mesure la description du circuit établie par le concepteur exploite les possibilités de la partie opérative. Il est souhaitable pour cela de connaître la fréquence d'utilisation des différentes instructions. Mais même si on connaît la fréquence d'utilisation des instructions de niveau machine, on ne connaîtra pas pour cela la fréquence des instructions opératives à l'intérieur de chaque instruction machine. On peut effectuer alors l'hypothèse simplificatrice que toutes les instructions opératives en entrée d'Apollon ont la même fréquence. Dans ce contexte, il est intéressant de déterminer le rapport du nombre moyen de bus utilisés au nombre total de bus ainsi que le rapport du nombre moyen d'opérations au nombre total d'opérateurs utilisables simultanément. On obtient ainsi une mesure de l'utilisation des ressources de la partie opérative, ce qui permet de mieux cerner le compromis surface-vitesse. Une approche plus simple consiste à analyser directement la description et non pas le résultat produit par le compilateur. On peut calculer le rapport du nombre moyen de transferts au nombre maximal de transferts ainsi que le rapport du nombre moyen d'opérations au nombre maximal d'opérations en parallèle spécifiées par le concepteur. On peut ainsi tracer les courbes de distribution du nombre de transferts et du nombre d'opérations et diminuer le parallélisme des instructions demandant beaucoup plus de matériel que la moyenne.

7. Exemples de parties opératives compilées

7.1 Division euclidienne

Nous donnons ici une version légèrement différente de la procédure de division euclidienne donnée à la section 2. Cette procédure utilise une variable de plus mais permet de diminuer le nombre d'instructions opératives en augmentant le parallélisme à l'intérieur des instructions.

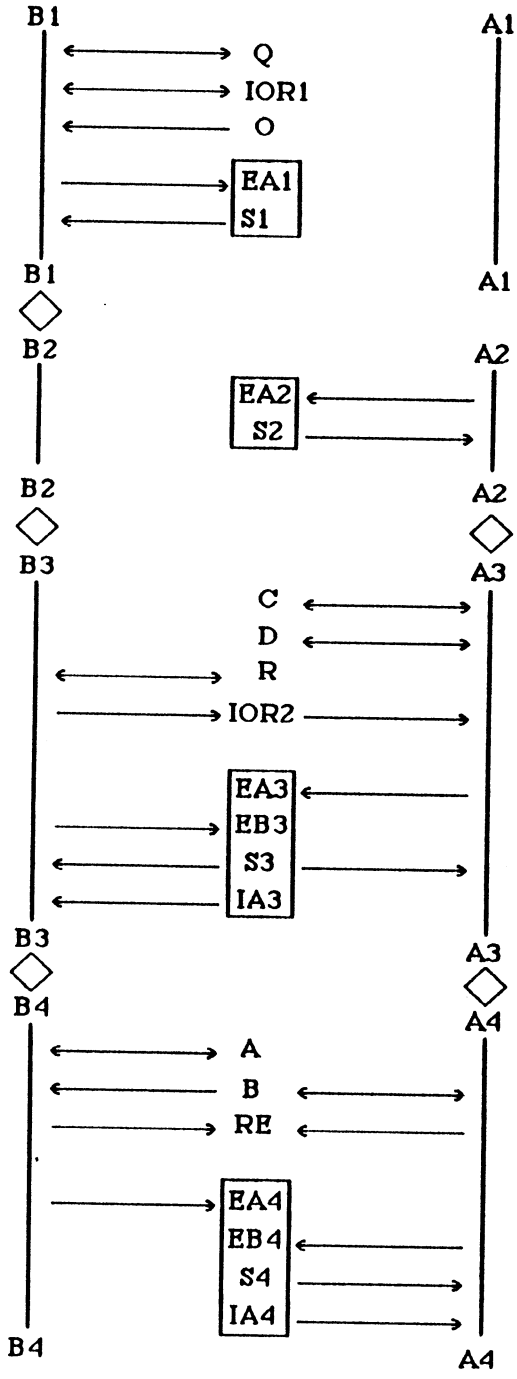
Deuxième version de la division euclidienne

```
procedure division-rapide (A, B : integer; var R, Q : integer);

(* Une autre version de la division euclidienne utilisant une variable de
plus *)

var C, D : integer;

begin

    C := B; D := B * 2;
    while C <= A do
    begin
        C := D; D := D * 2;
    end;
    R := A; Q := 0; D := C div 2;
    while C > B do
    begin
        Q := Q * 2; C := D; D := D div 2;
        if R >= C then
        begin
            Q := Q + 1; R := R - C;
        end;
    end;
end;
```

Description Apollon de cet algorithme

registres : A B C D Q R RE ;
entrées-sorties : IOR1 0:8 IOR2 0:8 ;
sorties : ;
champs de contrôle : RE 0:8 ;
sous-pops : 5;
nbits : 8 ;
fig : queuc ;

A <- IOR1 / B <- IOR2 ;
IOR1 <- Q / IOR2 <- R ;
C <- B / D <- mult2 B / BOP(, B, <=, A, RE 4:4) ;
C <- D phi1 / D <- mult2 D / BOP(, D, <=, A, RE 4:4) ;
Q <- 0 / R <- A / D <- div2 C / BOP(, C, >, B, RE 4:4) ;
Q <- mult2 Q / C <- D phi1 / D <- div2 D / BOP(, R, >=, D, RE 0:4) / BOP(, D, >, B, RE 4:4) ;
Q <- incr Q / R <- R - C / BOP(, C, >, B, RE 4:4) .



sous-pop N°1 : mult2 incr
sous-pop N°2 : div2
sous-pop N°3 : >= - div2 mult2
sous-pop N°4 : > <=

Partie opérative exécutant la division euclidienne deuxième version
figure 3.54

Ai et Bi sont les bus de la sous partie opérative numéro i, EAi et EBi les entrées, Si et IAi la sortie et les indicateurs arithmétiques.
Remarque : la partie opérative a maintenant 4 sous parties opératives puisqu'une instruction comporte 4 opérations en parallèle.

Exécution détaillée des instructions sur la partie opérative générée par Apollon

```
instruction No 1 :   A <-- IOR1    B <-- IOR2
  phi1    IOR1 --->  tampon-IOR1
  phi1    IOR2 --->  tampon-IOR2
  phi2    tampon-IOR1  --->  B1
  phi2    B1    --->  B2
  phi2    tampon-IOR2  --->  A3
  phi2    B2    --->  B3
  phi2    A3    --->  A4 ---> B
  phi2    B3    --->  B4 ---> A
=========================================================
instruction No 2 :   IOR1 <-- Q    IOR2 <-- R
  phi1    Q     --->  B1 ---> tampon-IOR1
  phi1    R     --->  B3 ---> tampon-IOR2
  phi1    tampon-IOR1 ---> IOR1
  phi1    tampon-IOR2 ---> IOR2
  phi2    tampon-IOR1 ---> IOR1
  phi2    tampon-IOR2 ---> IOR2
=========================================================
instruction No 3 :   C <-- B    D <-- mult2 B    BOP(, B, <=, A, RE 4:4)
sous-pop 3 : mult2     sous-pop 4 : <=
  phi1    A4    --->  A3 ---> EA3   C
  phi1    B     --->  A4 ---> EB4
  phi1    A     --->  B4 ---> EA4
  phi2    S3    --->  A3 ---> D
  phi2    IA4    --->  A4 ---> RE 4:4
=========================================================
instruction No 4 :   C <-- D phi1    D <-- mult2 D    BOP(, D, <=, A, RE 4:4)
sous-pop 3 : mult2     sous-pop 4 : <=
  phi1    D     --->  A3 ---> EA3   C
  phi1    A3    --->  A4 ---> EB4
  phi1    A     --->  B4 ---> EA4
  phi2    S3    --->  A3 ---> D
  phi2    IA4    --->  A4 ---> RE 4:4
=========================================================
```

```
instruction No 5 :   Q <-- 0    R <-- A    D <-- div2 C    BOP(, C, >, B, RE 4:4)
sous-pop 3 : div2     sous-pop 4 : >
  phi1    C      --->  A3 ---> EA3
  phi1    A3     --->  A4 ---> EB4
  phi1    B      --->  B4 ---> EA4
  phi2    0      --->  B1 ---> Q
  phi2    S3     --->  A3 ---> D
  phi2    B4     --->  B3 ---> R
  phi2    IA4    --->  A4 ---> RE 4:4
  phi2    A      --->  B4
=========================================================
instruction No 6 :   Q <-- mult2 Q    C <-- D phi1    D <-- div2 D    BOP(, R,
>=, D, RE 0:4)   BOP(, D, >, B, RE 4:4)
sous-pop 1 : mult2     sous-pop 2 : div2     sous-pop 3 : >=     sous-pop 4 : >

  phi1    Q      --->  B1 ---> EA1
  phi1    A3     --->  A2 ---> EA2
  phi1    D      --->  A3 ---> C     EA3
  phi1    R      --->  B3 ---> EB3
  phi1    A3     --->  A4 ---> EB4
  phi1    B      --->  B4 ---> EA4
  phi2    S1     --->  B1 ---> Q
  phi2    S2     --->  A2
  phi2    A2     --->  A3 ---> D
  phi2    IA3    --->  B3
  phi2    IA4    --->  A4 ---> RE 4:4
  phi2    B3     --->  B4 ---> RE 0:4
=========================================================
instruction No 7 :   Q <-- incr Q    R <-- R - C    BOP(, C, >, B, RE 4:4)
sous-pop 1 : incr     sous-pop 3 : -     sous-pop 4 : >
  phi1    Q      --->  B1 ---> EA1
  phi1    C      --->  A3 ---> EA3
  phi1    R      --->  B3 ---> EB3
  phi1    A3     --->  A4 ---> EB4
  phi1    B      --->  B4 ---> EA4
  phi2    S1     --->  B1 ---> Q
  phi2    S3     --->  B3 ---> R
  phi2    IA4    --->  A4 ---> RE 4:4
```

figure 3.55 : exécution détaillée des instructions de la division

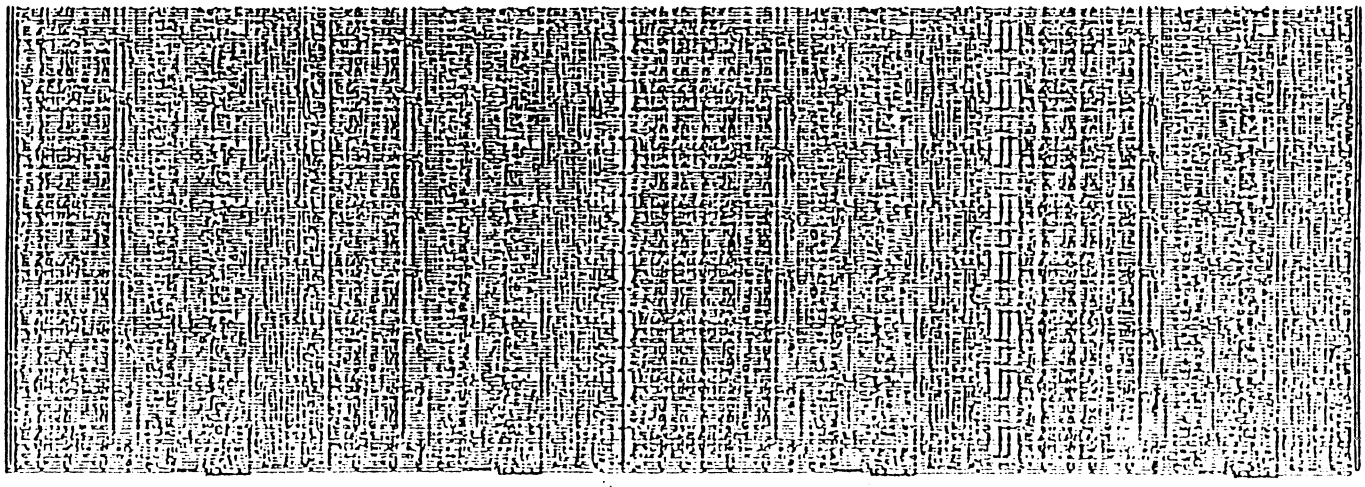
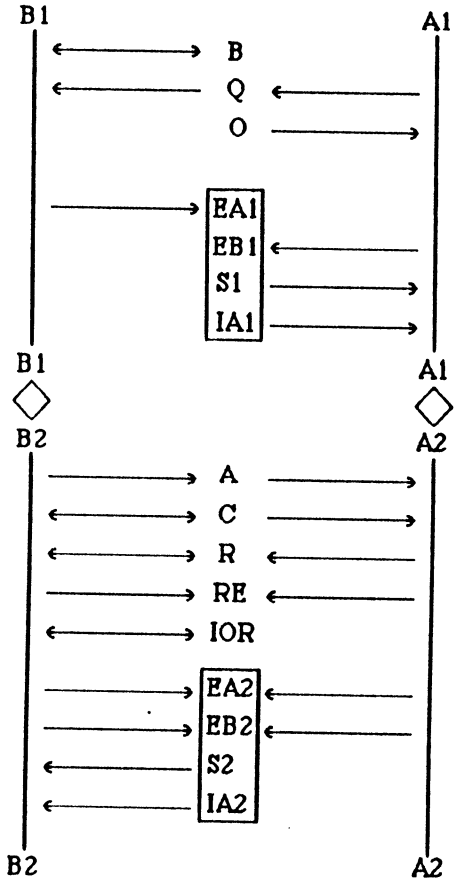
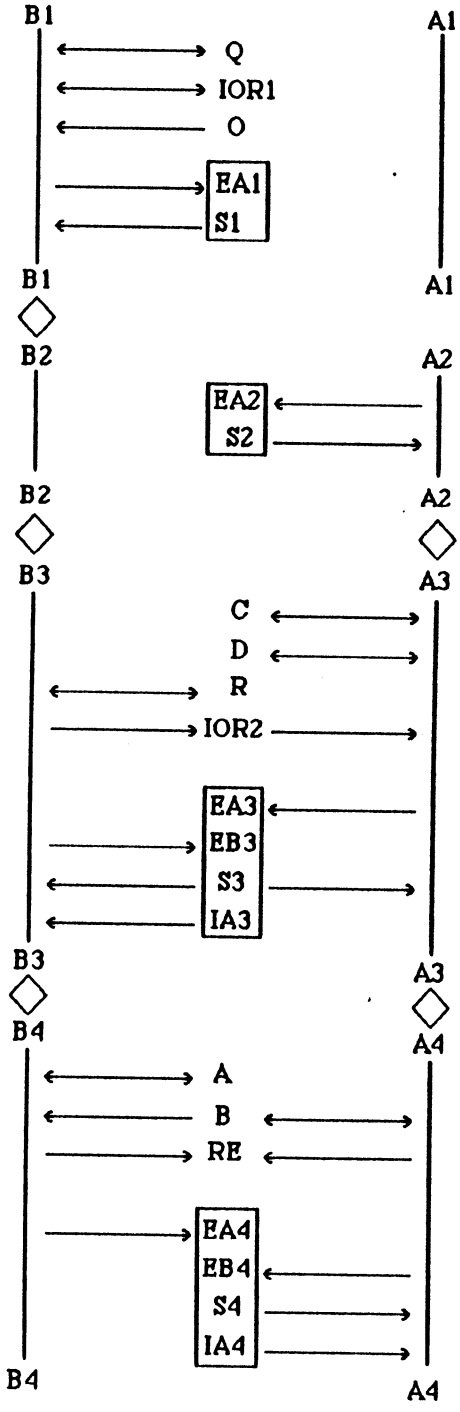


figure 3.56 : dessin des masques de la partie opérative exécutant la division euclidienne deuxième version





Comparaison des parties opératives exécutant les 2 versions de la division euclidienne

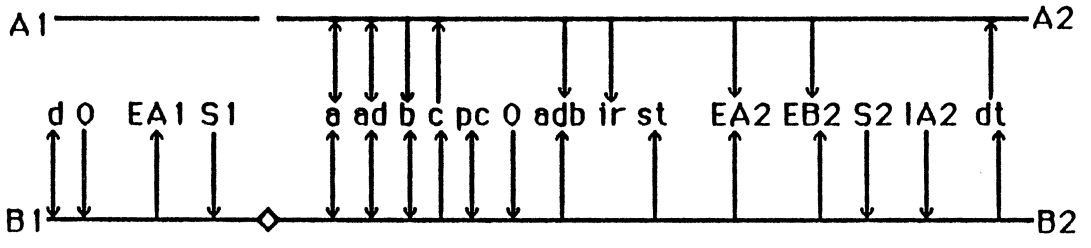
figure 3.57

7.2 Microprocesseur simplifié

Nous donnons en annexe les résultats de la compilation d'un microprocesseur simplifié exécutant un jeu très réduit d'instructions. La description de ce microprocesseur a été établie à partir d'un exemple donné dans [ANCE 82]. Nous avons repris cet exemple pour des raisons didactiques.
Nous ne présentons ici que le schéma de la partie opérative.

Schéma fonctionnel de la partie opérative du microprocesseur simplifié

EAi et EBi sont les entrées de la boîte à opérations numéro i, Si la sortie et IAi les indicateurs arithmétiques.
Ai et Bi sont les bus de la sous-partie opérative numéro i.



sous-pop N°1 : incr mult2
sous-pop N°2 : - div2 incr + mult2

figure 3.58 : partie opérative d'un microprocesseur simplifié

7.3 Le 6502

Dans ce paragraphe, nous présentons la compilation de la partie opérative du 6502. La compilation du 6502 est présentée dans [REIS 86]. Le dessin des masques de la partie opérative est généré automatiquement à partir d'une description algorithmique. La description a été effectuée en Irène [MARI 85] puis en LDS [LAUR 85] à partir des spécifications du W65C02 [WDC 83].

Le W65C02 est un microprocesseur 8 bits fabriqué en CMOS totalement compatible avec le 6502 NMOS. L'utilisation de la technologie CMOS a permis d'augmenter les fonctionnalités des microprocesseurs de la série 6500. Le jeu d'instructions du W65C02 comporte 66 instructions soit 10 de plus que celui du 6502 NMOS. Il existe 180 au lieu de 151 codes opérations et 15 au lieu de 13 modes d'adressage. Les registres du W65C02 sont identiques à ceux du 6502 NMOS. Ces registres sont constitués d'un ensemble de registres 8 bits - un accumulateur A, 2 registres index X et Y, un pointeur de pile S et un registre d'indicateurs d'état - et d'un registre 16 bits, le compteur ordinal qui est décomposé en 2 registres 8 bits : PCL et PCH. Toutes les opérations arithmétiques sont effectuées par l'UAL y

compris les incrémentations et les décrémentations sauf celles du compteur ordinal qui sont effectuées par un incrémenteur-décrémenteur lié au compteur ordinal.

Nous présentons ici le schéma de la partie opérative compilée et le dessin des masques. La description d'entrée d'Apollon est donnée en annexe. Elle est constituée d'une centaine d'instructions comportant au plus deux opérations en parallèle.

La façon dont le concepteur décrit l'algorithme d'interprétation des instructions a une grande influence sur le résultat de la compilation. En ce qui concerne la partie opérative, le parallélisme est spécifié par l'utilisateur. En particulier le nombre d'UALs et de bus est déterminé directement à partir du nombre d'opérations et de transferts à exécuter en parallèle. Les registres sont aussi spécifiés par l'utilisateur. La partie opérative du SYCO6502 possède 2 registres généraux de plus que le W65C02 ou le 6502. Elle possède 3 registres généraux A, B et C. Ces registres ont été rajoutés afin d'accélérer l'exécution des instructions. La partie opérative possède en outre 2 UALs. La première exécute les opérations suivantes :

- incrémentation, décrémentation
- opérations logiques OU, ET, OUX
- addition
- soustraction
- décalage à gauche et à droite

La deuxième peut effectuer les opérations :

- opérations logiques ET, OU
- incrémentation
- complémentation
- décalage à gauche et à droite.

Actuellement le compilateur ne permet pas d'effectuer des opérations sur un nombre restreint de bits d'un registre. C'est pourquoi les opérations de ce type sont effectuées comme des opérations logiques dont les opérandes sont un registre et une constante de masquage. C'est pour cette raison qu'un nombre assez important de constantes ont dû être rajoutées.

Plus généralement, on peut classer les opérations portant sur nombre restreint de bits d'un registre ou de plusieurs registres en 2 types :

- 1° type : les données sont transférées au moyen des 2 bus de la partie opérative.

exemple : A0:2 := B0:2;

L'exécution d'opérations de ce type portant sur un nombre restreint de bits nécessite la duplication des commandes de sélection des registres (ou bien l'introduction d'un "fool it" - cf chapitre 3, section 4.4.1.1) ainsi que la duplication du bloc de génération des comptes rendus.

- 2° type : les données ne peuvent pas être transférées par les bus. On veut par exemple écrire le 2° bit du registre A dans le 5° du registre B. L'exécution d'opérations de ce type nécessite soit la création au coup par coup de chemins perpendiculaires aux bus de données soit l'utilisation d'un décaleur généralisé qui permet d'accélérer l'exécution de ces opérations sans toutefois les rendre possible en un cycle.

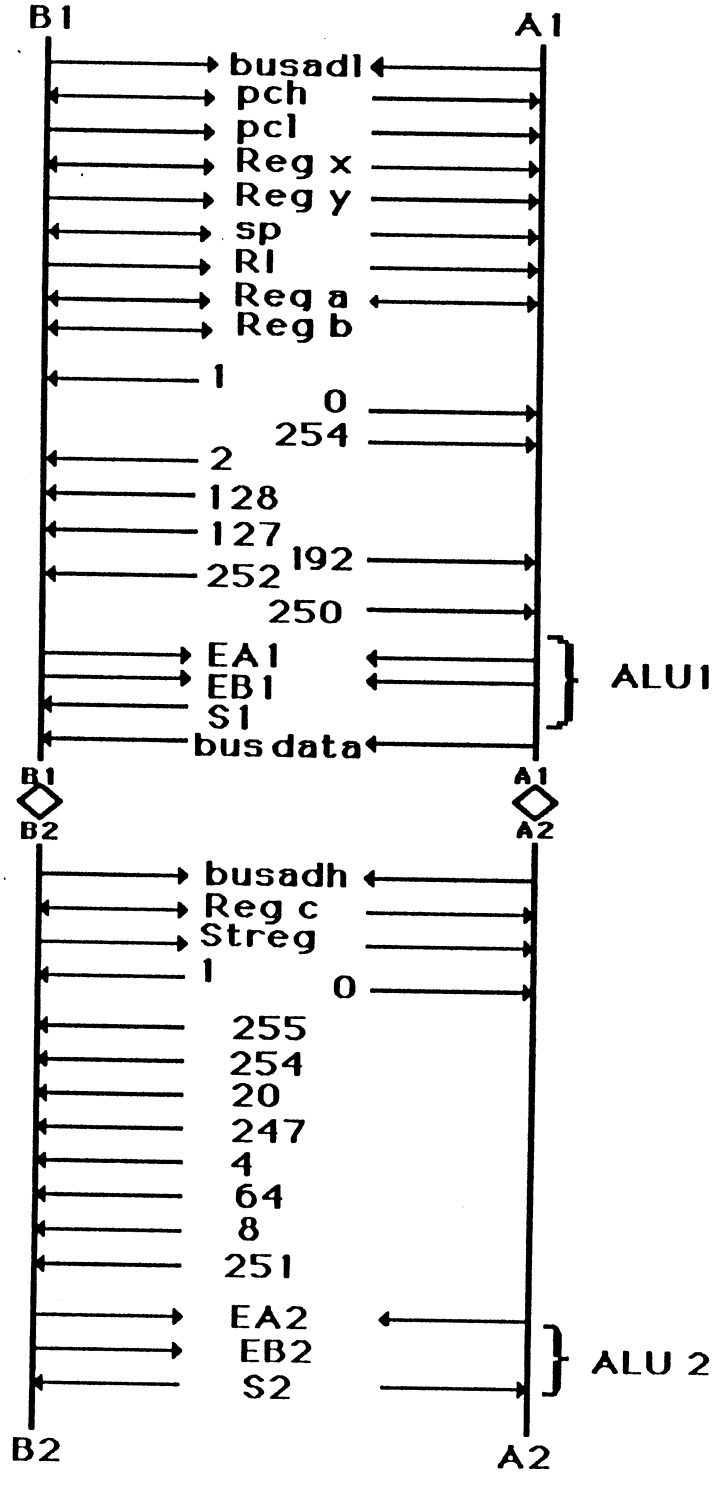
L'introduction d'opérations portant sur un nombre quelconque de bits permettra

d'une part de diminuer le nombre des constantes utilisées dans la partie opérative tout en provoquant cependant une augmentation de surface due aux modifications de la structure d'accès des registres et d'autre part d'augmenter la vitesse d'exécution de ces opérations.

Le 6502 a été choisi parce que d'une part c'est un exemple de circuit ayant la complexité d'un microprocesseur et d'autre part parce qu'il a été adopté comme référence pour comparer plusieurs compilateurs [EVAN 85]. [EVAN 85] donne les résultats de la compilation du 6502 par plusieurs compilateurs de silicium. Les résultats sont toutefois difficilement exploitables pour la compilation de la partie opérative puisque seule la surface globale du circuit est donnée.

La partie opérative du SYCO6502 est un rectangle de 1427 lambdas sur 680 lambdas soit 3.2 mm sur 1.5 mm pour une technologie NMOS 4.5 microns (lambda = 2.25), ce qui fait une surface de 4.9 mm^2. Par partie opérative, on entend le chemin de données de 8 bits et les amplificateurs de validation répartis sur 2 étages sans les plots d'entrée-sortie. La densité moyenne est de 530 transistors par mm^2 ou exprimée dans une autre unité 300 $lambda^2$ par transistor. La densité maximum est de 200 $lambda^2$ par transistor ou environ 800 transistors par mm^2 si l'on ne considère que les 8 tranches d'un bit sans l'interface partie opérative - partie contrôle. Si l'on rajoute 200 lambdas de chaque côté du bit slice pour les plots, on obtient une largeur de 4.3 mm alors que la largeur du W65C02 fabriqué par Western Design Center avec une technologie CMOS 3 microns est de 3.5 mm.

Remarque : d'autres parties opératives ont pu être compilées. La compilation d'une description reprenant les principales instructions du MC68000 est présentée dans [GERO 85]. [JAMI 85] présente la compilation d'un algorithme de calcul formel : le calcul du pgcd de polynômes.



sous-pop N°1 : incr or - and + decr mult2 div2 exor
sous-pop N°2 : and or incr not mult2 div2

figure 3.59 : partie opérative du 6502 compilé par Apollon

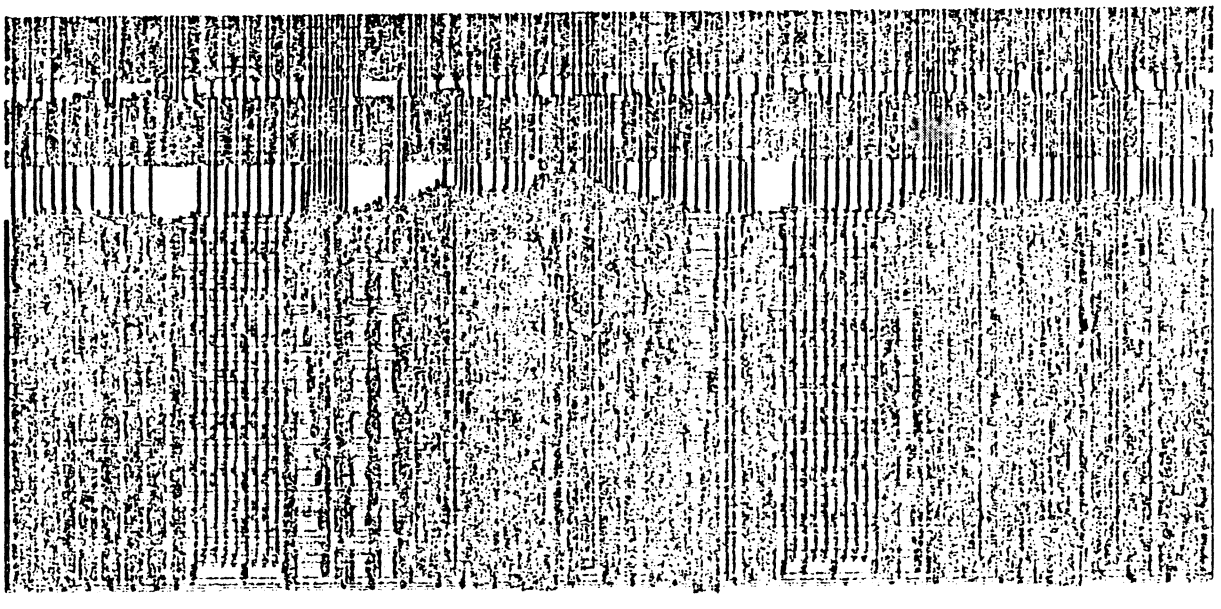


figure 3.60 : dessin des masques du 6502 compilé par Apollon

Conclusion

La réalisation du compilateur de parties opératives Apollon a démontré la faisabilité de l'approche proposée par [ANCE 84]. L'adoption d'un modèle à la fois architectural, temporel, électrique et topologique ainsi que l'utilisation de cellules précaractérisées permet d'obtenir un circuit dense tout en offrant des possibilités de parallélisme importantes. En outre l'algorithme de traduction est grandement simplifié du fait de l'adoption de ce modèle.

Toutefois la portée du compilateur en est diminuée. Plus particulièrement l'utilisation d'une structure d'interconnexion à 2 bus (éventuellement segmentés) et de la structure topologique en tranches ("bit slice") diminue la généralité de l'approche. En effet il peut être nécessaire d'utiliser un troisième bus, voire plus pour exécuter les instructions opératives dans le temps imparti. Il semble donc souhaitable que le compilateur puisse générer des parties opératives à 2 bus aussi bien qu'à 3 bus. La généralisation du modèle à plus de 3 bus n'est guère utile puisque la plupart des opérateurs utilisés sont unaires ou binaires.

La structure "bit slice" est utilisée dans la plupart des microprocesseurs actuels. C'est une structure régulière qui permet d'obtenir une partie opérative dense. Elle nécessite que toutes les cellules d'une même tranche aient la même hauteur. L'approche que nous avons adoptée est basée sur cette structure en tranches. Le choix du modèle bit slice à bus parallèles permet de résoudre les problèmes topologiques dès le début de la conception. L'adoption d'une approche topologique plus générale nécessiterait des outils de routage et de placement performants et remettrait en cause le choix de la structure d'interconnexion. Si la structure d'interconnexion à un nombre limité de bus parallèles contribue à la régularité de la structure "bit slice", elle restreint les possibilités de parallélisme et semble moins intéressante dans une perspective plus générale.

L'utilisation de cellules prédéfinies permet d'obtenir un circuit dense. Par contre il n'est pas possible d'adapter les caractéristiques électriques de ces cellules à leur environnement électrique ou aux spécifications du concepteur que si l'on dispose de plusieurs jeux de mêmes cellules fonctionnelles. La prise en compte automatique des problèmes électriques nécessite une modification des outils de génération des spécifications des masques.

REFERENCES

REFERENCES

[AGER 76] **T. Agerwala**, "Microprogram optimization : a survey", IEEE transactions on computers, Vol C.25, n°10, October 1976.

[AHO 77] **A.V. Aho and J.D. Ullman**, "Principles of compiler design", Addison Wesley, 1977.

[ANCE 82] **F. Anceau**, "Architecture des ordinateurs : exemple de conception d'un petit microprocesseur", cours ENSIMAG, Juin 1982.

[ANCE 83] **F. Anceau**, "Capri : a design methodology and a silicon compiler for VLSI circuits specified by algorithms", 3rd Caltech Conference on VLSI, March 1983.

[ANCE 84] **F. Anceau**, "Silicon compilation for microprocessor-like VLSI", NATO advanced study institute on microarchitecture of VLSI Computers, Urbino, Italy, July 1984.

[ARZO 84] **C. Arzounian, L. Harivel**, "Génération automatique de parties opératives", rapport de fin d'études ENSIMAG, IMAG/TIM3, 46 av. Félix Viallet, 38031 Grenoble Cédex, juin 1984.

[BARB 81] **M.R. Barbacci**, "Syntax and semantics of CHDLs", Computer hardware description languages and their applications, IFIP, 1981.

[BEKK 86] **N. Bekkara**, "Optimisation et compromis surface-performance dans le compilateur SYCO", rapport interne, IMAG/TIM3, 46 av. Félix Viallet, 38031 Grenoble Cédex, juin 1986.

[BERÇ 84] **N. Bergman**, "A case study of the FIRST silicon compiler", Dep of Computer Science, U. of Edinburgh, Internal report, February 1984.

[BLAC 85] **T. Blackman, J. Fox, C. Rosebrugh**, "The Silc silicon compiler language and features", 22nd DAC, 1985.

[BOUR 86] **E. Bourcier**, "LDS : langage d'entrée du compilateur de silicium SYCO", rapport interne, IMAG/TIM3, 46 av. Félix Viallet, 38031 Grenoble Cédex, avril 1986.

[BRAY 85] **R.K. Brayton et al.**, "The Yorktown silicon compiler", ISCAS'85, Kyoto, Japan, June 1985.

[BRAY 85] **R.K. Brayton et al.**, "A microprocessor design using the Yorktown silicon compiler", ICCD'85.

[BREW 86] **F.D. Brewer, D.D. Gajski**, "An expert system paradigm for design", 23rd DAC, 1986.

[BURI 86] **M.R. Burich**, "Design of module genrators and silicon compilers", Nato advanced study institute on logic synthesis and silicon compilation for VLSI design, L'Aquila, Italy, July 7-18, 1986.

[CAMP 84] **R. Camposano**, "Automatic data path synthesis from DSL specifications", ICCD'84.

[CAMP 85] **R. Camposano**, "Synthesis techniques for digital systems design", 22nd DAC, 1985.

[CHUQ 84] **S. Chuquillanqui Bernaola**, "Une nouvelle approche pour l'optimisation topologique et l'automatisation du dessin des masques de PLA complexes", thèse de docteur-ingénieur INPG, octobre 1984.

[DAVI 81] **S.Davidson**, "Some experiments in local microcode compaction for horizontal machines", IEEE transactions on computers, Vol C-30, n°7, July 1981.

[DELO 86] **H. Delori et al.**, "French MPC 1984-1985", IMAG/TIM3 research report n° 602, February 1986.

[DEMA 86] **H. De Man**, "A synthesis and module generation system for multiprocessor systems on a chip", Nato advanced study institute on logic synthesis and silicon compilation for VLSI design, L'Aquila, Italy, July 7-18, 1986.

[DEMI 84] **G. De Micheli**, "Optimal encoding of control logic", ICCD'84.

[DEMI 86] **G. De Micheli**, "Synthesis of control systems", Nato advanced study institute on logic synthesis and silicon compilation for VLSI design, L'Aquila, Italy, July 7-18, 1986.

[DIRE 81] **S.W. Director**, "A design methodology and computer aids for

digital VLSI systems", IEEE transactions on circuits and systems, VOL. CAS-28, N°7, July 1981.

[EVAN 85] **S. Evanczuk**, "Results of a silicon compiler design challenge", VLSI design, Jyly 1985.

[FISH 81] **J.A. Fisher**, "Trace scheduling : a technique for global microcode compaction", IEEE transactions on computers, Vol C-30, n°7, July 1981.

[GAJS 83] **D.D. Gajski, R.H. Kuhn**, "New VLSI Tools", IEEE Computer, December 1983.

[GAJS 86] **D.D. Gajski et al**, "Silicon compilation (tutorial)", IEEE 1986 CICC.

[GAJS 86] **D. D. Gajski et al**, "Silicon cell compilers", Nato advanced study institute on logic synthesis and silicon compilation for VLSI design, L'Aquila, Italy, July 7-18, 1986.

[GALL 84] **R. Gallais**, "Participation à la réalisation d'un microprocesseur 16 bits interprétant le Pcode", Thèse CNAM, Mars 1984.

[GERO 85] **J.P. Geronimi, S. Ringot, D. Savart**, "Conception d'un microprocesseur à l'aide d'outils de CAO de haut niveau", rapport de stage ENSERG, IMAG/TIM3, 46 Av. Félix Viallet, 38031 Grenoble Cédex, 1985.

[GIRC 84] **E.F. Girczyc, J.P. Knight**, "An ada to standard cell hardware compiler based on graph grammars and scheduling", ICCD'84.

[GLUS 65] **V.M. Glushkov**, "Automata theory and structural design problems of digital machines", Kibernetika, Vol 1 n°1 pp 3-11, 1965.

[GRAY 82] **J.P. Gray et al**, "Designing gate arrays using a silicon compiler", 19th DAC, 1982.

[GROS 83] **R.R. Gross**, "Silicon compilers : a critical survey", Dep. of Computer Science, University of North Carolina at Chapel Hill, May 1983.

[HAFE 81] **L. Hafer, A.C. Parker**, "A formal method for the specification, analysis and design of RTL digital logic", 18th

DAC, 1981.

[HAFE 82] **L.J. Hafer, A.C. Parker**, "Automated synthesis of digital hardware", IEEE transactions on computers, Vol C-31, n°2, February 1982.

[HANS 84] **J.E. Hansen** , "Cmos cell library for the Apollon silicon compiler", IMAG/TIM3 research report n° 490, 46, av. Félix Viallet, 38031 Grenoble Cédex, Décembre 1984.

[HITC 83] **C.Y. Hitchcock, D.E. Thomas**, "A method of automatic datapath synthesis", 20th DAC, 1983.

[HILL 84] **F.J. Hill**, "Hardware compilation from an RTL to a storage logic array target", IEEE transactions on computer-aided design, Vol. CAD-3, N°3, July 1984.

[ISOD 83] **S. Isoda**, "Global compaction of horizontal microprograms based on the generalized data dependency graph", IEEE transactions on computers, Vol C-32, n°10, October 1983.

[JAMI 85a] **R. Jamier, A. Jerraya**, "Apollon, a datapath silicon compiler", ICCD'85.

[JAMI 85b] **R.Jamier, A. Jerraya, Y. Robert**, "Using a silicon compiler for computer algebra", Proceedings of Future Trends of Computing, Grenoble, December 2-6, 1985.

[JAMI 86] **R. Jamier, N. Bekkara, A. Jerraya**, "The automatic synthesis of data processing sections", ICCD'86.

[JERR 83] **A. Jerraya**, "Une nouvelle approche pour la vérification des masques des circuits intégrés", Thèse INPG, novembre 1983.

[JERR 85] **A. Jerraya, F.R. Rougeaux, E. Rosier, B. Courtois**, "A hierarchical symbolic layout system : STYX", VLSI'85, Tokyo, Japan, August 1985.

[JERR 86] **A. Jerraya, P. Varinot, R. Jamier, B. Courtois**, "Principles of the SYCO compiler", DAC 86.

[JIN 86] **JIN Jianming**, "Une procédure d'assemblage automatique des cellules de l'interface électrique entre la partie opérative et la partie contrôle", rapport interne, IMAG/TIM3, 46, av. Félix

Viallet, 38031 Grenoble Cédex, juin 1986.

[JOHA 79] **D. Johannsen**, "Bristle blocks : a silicon compiler", 16th DAC, 1979.

[KNAP 86] **D. W. Knapp, A.C. Parker**, "A design utility manager : the ADAM planning engine", 23rd DAC, 1986.

[KOWA 83] **T.J. Kowalski, D.E. Thomas**, "The VLSI design automation assistant : prototype system", 20th DAC, 1983.

[KOWA 85a] **T.J. Kowalski et al**, "The VLSI design automation assistant : from algorithms to silicon", IEEE design and test, august 1985.

[KOWA 85b] **T.J. Kowalski**, "An artificial intelligence approach to VLSI design", Kluwer academic publishers, 1985.

[LAND 80] **D. Landskov et al**, "Local microcode compaction techniques", ACM Comput. Surveys, Vol.12, pp. 261-294, September 1980.

[LAUR 85] **D. Laurent, H.N. Nguyen**, "LDS", rapport Bull, Juin 1985.

[LURS 84] **C.Lursinsap, D. Gajski**, "Cell compilation with constraints", 21st DAC, 1984.

[MALA 85] **Y.K. Malaiya**, "Options in control implementation", ICCD'85.

[MARI 86] **S. Marine**, "Irène : un langage de description, simulation et synthèse automatique du matériel VLSI", thèse de docteur INPG, février 1986.

[MARS 86] **T. Marshburn et al**, "Datapath : a CMOS Data Path silicon assembler", 23rd DAC, 1986.

[MART 85] **F. Martinez**, "Compilateur du langage Irène", thèse CNAM, Novembre 1985.

[MATH 83] **T.G. Matheson et al**, "Embedding electrical and geometric constraints in hierarchical circuit-layout generators", ICCAD'83.

[MATH 80] **A. Mathialagan, N. Biswas**, "Optimal interconnections in the design of microprocessors and digital systems", IEEE journal of solid-state circuits, VOL. SC-15, NO. 1, february 1980.

[MCFA 83] **M.C. McFarland**, "Computer-aided partitioning of behavioral hardware descriptions", 20th DAC, 1983

[MCFA 86] **M.C. McFarland**, "Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions", 23rd DAC, 1986.

[MEAD 80] **C. Mead, L. Conway**, "Introduction to VLSI systems", Addison-Wesley, 1980.

[MHAY 86a] **N. Mhaya**, "Traitement d'un exemple simple", journées SYCO du 10/11 Avril 1986, IMAG/TIM3, 46 av. Félix Viallet, 38031 Grenoble Cédex, avril 1986.

[MHAY 86b] **N. Mhaya**, "Compilation de parties contrôle", rapport de DEA, IMAG/TIM3, 46 av. Félix Viallet, 38031 Grenoble Cédex, juillet 1986.

[NAGL 82] **A. Nagle et al**, "Synthesis of hardware for the control of digital systems", IEEE transactions on CAD, Vol CAD-1, N°4, october 1982.

[OBRE 82] **M. Obrebska**, "Etude comparative de différentes méthodes de conception des parties contrôle de microprocesseurs", Thèse INPG, Juin 1982.

[ORAI 86] **A. Orailoglu, D.J. Gajski**, "Flow graph representation", 23rd DAC, 1986.

[OUST 83] **J.K. Ousterhout**, "Crystal : a timing analyzer for nMOS VLSI circuits", Proceedings of the 3rd Caltech VLSI conference, 1983.

[PAIL 85] **J.F. Paillotin**, "Le système LUCIE", juillet 1985.

[N.PARK 85] **N. Park, A. Parker**, "Synthesis of optimal clocking schemes", 22nd DAC, 1985.

[N.PARK 86] **N. Park, A. Parker**, "Sehwa : a program for synthesis of pipelines", 23rd DAC, 1986.

[A.PARK 79] **A. C. Parker et al**, "An example of automated Data Path design", 16th DAC, 1979.

[A.PARK 86] **A.C. Parker et al**, "Maha : a program for datapath synthesis", 23rd DAC, 1986.

[PERE 85] **T. Perez Segovia**, "Paola : un système d'optimisation topologique de PLA", thèse 3° cycle INPG, octobre 1985.

[RAJA 85] **J.V. Rajan, D.E. Thomas**, "Synthesis by delayed binding of decisions", 22nd DAC, 1985.

[RAMA 85] **K. Ramayya**, "An automated data path synthesizer for a canonic structure, implementable in VLSI", 22nd DAC, 1985.

[REIS 86] **R.Reis, A.Jerraya, R. Jamier**, "Design of the SYCO 6502 using the Syco compiler", ICCD'86.

[ROSE 85] **C.P. Rosebrugh, J.H. Vellenga**, "Circuit synthesis for the Silc silicon compiler", IEEE CICC 1985.

[W. ROSE 85] **W. Rosenstiel, R. Camposano**, "Synthesizing circuits from behavioral level specifications", CHDL 85.

[SCHO 83] **J.P. Schoellkopf**, "Lubrick : a silicon assembler and its application to data-path design for FISC", VLSI'83.

[SCHO 85] **J.P. Schoellkopf**, "Contributions à l'architecture des circuits intégrés et à la compilation de silicium" Thèse d'Etat INPG, Mars 1985.

[SHIV 83] **S.G. Shiva**, "Automatic hardware synthesis", proceedings of the IEEE, Vol 71, n°1, January 1983.

[SHRO 82] **H.E. Shrobe**, " The Data Path Generator", 1982 Conference on advanced research in VLSI, MIT.

[SIFA 82] **J. Sifakis**, "Notes et compléments de cours sur les réseaux de Pétri", ENSIMAG, février 1982.

[SISK 82] **J.M. Siskind, J.R. Southard, K.W. Crouch**, "Generating custom high performance VLSI designs from succinct algorithmic descriptions", Conference on advanced research in VLSI, M.I.T., 1982.

[SIX 86] **P. Six et al**, "An intelligent module generator environment", 23rd DAC, 1986.

[SNOW 78] **E.A. Snow, D.P. Siewiorek, D.E. Thomas**, "A technology-relative computer-aided design system : abstract representations, transformations and design tradeoffs", 15th DAC, Las Vegas, 1978.

[SOUT 83] **J.R. Southard**, "Macpitts : an approach to silicon compilation", IEEE computer, December 1983.

[SUZI 81] **A.A. Suzim**, "Etude des parties opératives à éléments modulaires pour processeurs monolithiques", Thèse Docteur Ingénieur INPG, novembre 1981.

[TAKA 84] **S. Takagi**, "Rule based synthesis, verification and compensation of data paths", ICCD'84.

[THOM 77] **D.E. Thomas**, "The design and analysis of an automated design style selector", Ph. D. dissertation, Dep. Electrical Engineering, Carnegie-Mellon Univ., Pittsburgh, PA, June 1977.

[THOM 81] **D.E. Thomas**, "The automatic synthesis of digital systems", Proceedings of the IEEE, Vol.69, N°10, october 1981.

[THOM 83a] **D.E. Thomas et al**, "Automatic datapath synthesis", IEEE computer, December 1983.

[THOM 83b] **D.E. Thomas, G.W. Leive**, "Automating technology relative logic synthesis and module selection", IEEE Trans. on CAD, Vol. CAD-2, N°2, April 1983.

[TOKO 81] **M. Tokoro**, "Optimization of microprograms", IEEE transactions on computers, Vol C-30, n°7, July 1981.

[TORK 86] **K. Torki**, "Etude de l'architecture de parties contrôle autotestables pour le compilateur de silicium SYCO, compilation de circuits autotestables", rapport de DEA, 1986.

[TORN 77] **H.C. Torng, N.C.Wilhelm**, "The optimal interconnection of circuit modules in microprocessor and digital system design", IEEE Transactions on Computers, Vol C-26, N°5, may 1977.

[TSEN 81] **C.J.Tseng, D.P.Siewiorek**, "The modeling and synthesis of

bus systems", 18th DAC, 1981.

[TSEN 83] **C.J. Tseng, D.P. Siewiorek**, "Facet : a procedure for the automated synthesis of digital systems", 20th DAC, 1983.

[TSEN 84] **C.J. Tseng, D.P. Siewiorek**, "Emerald : a bus style designer", 21st DAC, 1984.

[VARI 86] **P. Varinot**, "Topologie générale d'un étage de partie contrôle" rapport interne, IMAG/TIM3, 46 av. Félix Viallet, 38031 Grenoble Cédex, 1986.

[VARI 86] **P. Varinot**, "Compilation de silicium : application à la génération automatique de parties contrôle", thèse à paraître.

[WALK 83] **R.A. Walker, D.E. Thomas**, "Behavioral level transformation in the CMUDA system", 20th DAC, 1983.

[WDC 83] **Western Design Center**, the W65C02 8-bit microprocessor data sheet, november 1983.

[WEST 85] **N. Weste, K. Eshraghian**, "Principles of CMOS VLSI design", Addison-Wesley 1985.

TABLE DES MATIERES

Génération automatique de parties opératives de circuits intégrés VLSI de type microprocesseur

Chapitre 2
La compilation de parties opératives p.43

Chapitre 3
Le compilateur APOLLON p.107

Liste des annexes

Liste des figures

Chapitre 1 :

Chapitre 2 :

Chapitre 3 :

ANNEXES

COMPILATION D'UN MICROPROCESSEUR SIMPLIFIE

1.Description LDS d'un microprocesseur simplifié

```
SMODULE MICROP(adbus,dtbus,dtack,restart,adstrobe,dtstrobe,read_write);

VARIABLE;
r 0:2, CTRL;
END;

SIGNAL;
 adbus 0:8, OUT; ?address bus
 dtbus 0:8, INOUT; ?data bus
 dtack, IN, CTRL; ?data acknowledgement
 restart, IN, CTRL;
 adstrobe, OUT, CTRL; ?address strobe
 dtstrobe, OUT, CTRL; ?data strobe
 read_write, OUT, CTRL; ?selection read=1 / write=0
END;

REGISTER;
 a 0:8; ?accumulator
 ad 0:8; ?address register
 b 0:8; ?special purpose register
 c 0:8; ?special purpose register
 d 0:8; ?special purpose register
 ir 0:8; ?instruction register
 pc 0:8; ?program counter
 streg 4:4; ?status register
 car, LIKE streg 1 ?carry bit
 neg, LIKE streg 2 ?negative bit
 ovf, LIKE streg 3 ?overflow bit
 zer, LIKE streg 4 ?zero bit
END;

LINK MICROP;

END;

?*********************************************
```

```
CMODULE ada;
<e1>
    (r := 2; EXECUTE fetch;)
    a := a + b;
END;

CMODULE lda;
<e1> (r := 1; EXECUTE fetch;)
END;

CMODULE sta;
<e1>
    (adbus := ad; RESET read_write; dtbus := a;)
    (adbus := ad; SET adstrobe; dtbus := a; SET dtstrobe; NEXT e2;)
<e2>
    IF (dtack = 0)
    (adbus := ad; dtbus := a; NEXT e2;)
    ELSE (RESET adstrobe; RESET dtstrobe;) END;
END;

CMODULE cma;
<e1>
    (r := 2; EXECUTE fetch;)
    BOP(, b, -, a, streg 4:4);
END;

CMODULE beq;
<e1> IF (zer = 1) pc := ad; END;
END;

CMODULE bgt;
<e1> IF ((neg = 0) AND (zer = 0)) pc := ad; END;
END;

CMODULE bra;
<e1> pc := ad;
END;

CMODULE div;
<e1>
    (r := 2; EXECUTE fetch;)
    (c := b; NEXT e2;)
<e2>
```

```
    BOP(, c, >, a, streg 4:4);
    IF ((neg=0) AND (zer=0)) NEXT e3; END;
    (c := c * 2; NEXT e2;)
<e3>
    (d := 0; NEXT e4;)
<e4>
    BOP(, c, <=, b, streg 4:4);
    IF ((neg=1) OR (zer=1)) NEXT e5; END;
     (d := d * 2; c := c / 2;)
    BOP(, a, <, c, streg 4:4);
    IF (neg=1) NEXT e4; END;
    (d := d + 1; a := a - c;)
    NEXT e4;
<e5>
    b := d;
    ? quotient in b, remainder in a
END;

CMODULE mva;
<e1> a := b;
END;

CMODULE fetch;
<send>
    (adbus := pc; SET read_write;)
    (adbus := pc; SET adstrobe; NEXT receive;)
<receive>
    IF (dtack = 0) (adbus := pc; NEXT receive;)
    ELSE
    (CASE (r)
          WHEN (0) ir := dtbus;
          WHEN (1) a := dtbus;
          WHEN (2) b := dtbus;
          WHEN (3) ad := dtbus;
    END;
    pc := pc + 1;
    RESET adstrobe;)
    END;
END;

?*********************************************

CMODULE MICROP;
```

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

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

<newinstr>
    (r := 0; EXECUTE fetch;)
    IF (ir 4:2 = '00') NEXT decode; ELSE (r :=3; EXECUTE fetch;) END;
    (CASE (ir 4:2
          WHEN ('01') ;
          WHEN ('10') ad := pc + ad;
          WHEN ('11') ad := pc - ad;
    END;
    NEXT decode;)

<decode>
    (CASE (ir 0:4)
          WHEN ('0000') EXECUTE ada; ? a <- a + dtbus
          WHEN ('0001') EXECUTE lda; ? a <- dtbus
          WHEN ('0010') EXECUTE sta; ? memory <- a
          WHEN ('0011') EXECUTE cma; ? compare dtbus to a
          WHEN ('0100') EXECUTE beq; ? jump if dtbus=A
          WHEN ('0101') EXECUTE bgt; ? jump if dtbus>A
          WHEN ('0110') EXECUTE bra; ? unconditional jump
          WHEN ('0111') EXECUTE div; ? division of "a" by dtbus
          WHEN ('1xxx') EXECUTE mva; ? a <- b
    END;
    IF (restart = 0) NEXT rest; ELSE NEXT newinstr; END;)

END;
```

2.Entree du compilateur de parties opératives

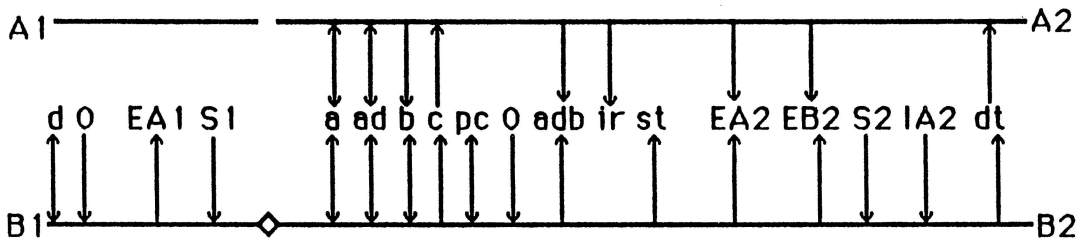
L'entrée est constituée de 2 parties :
- une partie déclaration de ressources
- une partie de spécification des actions opératives à exécuter en parallèle

micro.etp

```
registres : a ad b c d ir pc streg;
entrées-sorties : dtbus 0:8;
sorties : adbus 0:8;
champs de contrôle : ir 0:6 streg 4:4;
sous-pops: 3;
bits : 8;
nom : micro;

a <- a + b;
adbus <- ad / dtbus <- a;
BOP(, b, -, a, streg 4:4);
pc <- ad;
c <- b;
BOP(, c, -, a, streg 4:4);
c <- mult2 c;
d <- 0;
BOP(, c, -, b, streg 4:4);
d <- mult2 d / c <- div2 c;
BOP(, a, -, c, streg 4:4);
d <- incr d / a <- a - c;
b <- d;
a <- b;
adbus <- pc;
ir <- dtbus / pc <- incr pc;
a <- dtbus / pc <- incr pc;
b <- dtbus / pc <- incr pc;
ad <- dtbus / pc <- incr pc;
pc <- 0;
ad <- pc + ad;
ad <- pc - ad.
```

3.Schéma fonctionnel de la partie opérative

EAi et EBi sont les entrées de la boîte à opérations numéro i, Si la sortie et IΛi les indicateurs arithmétiques.
Ai et Bi sont les bus de la sous-partie opérative numéro i.



sous-pop No 1 : incr mult2
sous-pop No 2 : - div2 incr + mult2

4.Description des actions opératives précédentes incluant les bus et les opérateurs utilisés.

```
==========================================================
instruction No 1 :  a <-- a + b
sous-pop 2 : +
 phi1   a    ---> A2 ---> EA2
 phi1   b    ---> B2 ---> EB2
 phi2   S2   ---> B2 ---> a
==========================================================
instruction No 2 :  adbus <-- ad   dtbus <-- a
 phi1   ad   ---> A2 ---> tampon-adbus
 phi1   a    ---> B2 ---> tampon-dtbus
 phi1   tampon-adbus ---> adbus
 phi1   tampon-dtbus ---> dtbus
 phi2   tampon-adbus ---> adbus
 phi2   tampon-dtbus ---> dtbus
==========================================================
instruction No 3 :  BOP(, b, -, a, streg 4:4)
sous-pop 2 : -
 phi1   a    ---> A2 ---> EA2
 phi1   b    ---> B2 ---> EB2
 phi2   IA2  ---> B2 ---> streg 4:4
==========================================================
instruction No 4 :  pc <-- ad
 phi1   ad   ---> B2 ---> pc
==========================================================
instruction No 5 :  c <-- b
 phi1   b    ---> B2 ---> c
==========================================================
instruction No 6 :  BOP(, c, -, a, streg 4:4)
sous-pop 2 : -
 phi1   c    ---> A2 ---> EB2
 phi1   a    ---> B2 ---> EA2
 phi2   IA2  ---> B2 ---> streg 4:4
==========================================================
instruction No 7 :  c <-- mult2 c
sous-pop 2 : mult2
 phi1   c    ---> A2 ---> EA2
 phi2   S2   ---> B2 ---> c
==========================================================
instruction No 8 :  d <-- 0
 phi1   0    ---> B1 ---> d
==========================================================
```

```
instruction No 9 :  BOP(, c, -, b, streg 4:4)
sous-pop 2 : -
 phi1   c    ---> A2 ---> EB2
 phi1   b    ---> B2 ---> EA2
 phi2   IA2   ---> B2 ---> streg 4:4
==========================================================
instruction No 10 :  d <-- mult2 d    c <-- div2 c
sous-pop 1 : mult2    sous-pop 2 : div2
 phi1   d    ---> B1 ---> EA1
 phi1   c    ---> A2 ---> EA2
 phi2   S1   ---> B1 ---> d
 phi2   S2   ---> B2 ---> c
==========================================================
instruction No 11 :  BOP(, a, -, c, streg 4:4)
sous-pop 2 : -
 phi1   c    ---> A2 ---> EA2
 phi1   a    ---> B2 ---> EB2
 phi2   IA2   ---> B2 ---> streg 4:4
==========================================================
instruction No 12 :  d <-- incr d    a <-- a - c
sous-pop 1 : incr    sous-pop 2 : -
 phi1   d    ---> B1 ---> EA1
 phi1   c    ---> A2 ---> EA2
 phi1   a    ---> B2 ---> EB2
 phi2   S1   ---> B1 ---> d
 phi2   S2   ---> B2 ---> a
==========================================================
instruction No 13 :  b <-- d
 phi1   d    ---> B1
 phi1   B1   ---> B2 ---> b
==========================================================
instruction No 14 :  a <-- b
 phi1   b    ---> B2 ---> a
==========================================================
instruction No 15 :  adbus <-- pc
 phi1   pc   ---> B2 ---> tampon-adbus
 phi1   tampon-adbus ---> adbus
 phi2   tampon-adbus ---> adbus
==========================================================
instruction No 16 :  ir <-- dtbus    pc <-- incr pc
sous-pop 2 : incr
 phi1   pc   ---> B2 ---> EA2
 phi1   dtbus ---> tampon-dtbus
 phi2   tampon-dtbus ---> A2 ---> ir
```

```
 phi2   S2   ---> B2 ---> pc
==========================================================
instruction No 17 :  a <-- dtbus   pc <-- incr pc
sous-pop 2 : incr
 phi1   pc   ---> B2 ---> EA2
 phi1   dtbus ---> tampon-dtbus
 phi2   tampon-dtbus ---> A2 ---> a
 phi2   S2   ---> B2 ---> pc
==========================================================
instruction No 18 :  b <-- dtbus   pc <-- incr pc
sous-pop 2 : incr
 phi1   pc   ---> B2 ---> EA2
 phi1   dtbus ---> tampon-dtbus
 phi2   tampon-dtbus ---> A2 ---> b
 phi2   S2   ---> B2 ---> pc
==========================================================
instruction No 19 :  ad <-- dtbus   pc <-- incr pc
sous-pop 2 : incr
 phi1   pc   ---> B2 ---> EA2
 phi1   dtbus ---> tampon-dtbus
 phi2   tampon-dtbus ---> A2 ---> ad
 phi2   S2   ---> B2 ---> pc
==========================================================
instruction No 20 :  pc <-- 0
 phi1   0    ---> B2 ---> pc
==========================================================
instruction No 21 :  ad <-- pc + ad
sous-pop 2 : +
 phi1   ad   ---> A2 ---> EA2
 phi1   pc   ---> B2 ---> EB2
 phi2   S2   ---> B2 ---> ad
==========================================================
instruction No 22 :  ad <-- pc - ad
sous-pop 2 : -
 phi1   ad   ---> A2 ---> EA2
 phi1   pc   ---> B2 ---> EB2
 phi2   S2   ---> B2 ---> ad
```

5.Liste des fils de commande et de compte rendu sortant de la partie opérative

1 precharge_bus
2 lecture_ecriture:d<->busB
3 lecture:0->busB
4 ecriture:busB->EA1
5 isolement_precharge_incr1
6 precharge_retenue_incr1
7 retenue_entrante_incr1
8 entree_shift_up_spop1 = 0
9 shift_up_spop1
10 no_shift_spop1
11 validation:resultat_operation_spop1->tampon_de_sortie_shifter
12 lecture:tampon_sortie_shifter1->busB
13 communication:busB1<->busB2
14 precharge_bus
15 lecture_ecriture:a<->busA
16 lecture_ecriture:a<->busB
17 lecture_ecriture:ad<->busA
18 lecture_ecriture:ad<->busB
19 ecriture:busA->b
20 lecture_ecriture:b<->busB
21 lecture:c->busA
22 ecriture:busB->c
23 lecture_ecriture:pc<->busB
24 lecture:0->busB
25 validation:adbus->plot_de_sortie
26 ecriture:busA->adbus
27 ecriture:busB->adbus
28 ecriture:busA->ir
29 bit 6 du registre ir
30 bit 5 du registre ir
31 bit 4 du registre ir
32 bit 3 du registre ir
33 bit 2 du registre ir
34 bit 1 du registre ir
35 ecriture:busB->streg
36 bit 8 du registre streg
37 bit 7 du registre streg
38 bit 6 du registre streg
39 bit 5 du registre streg
40 ecriture:busA->EA2
41 ecriture:busB->EA2

42 multiplexeur:EB2->entree_C_adder
43 multiplexeur:EB2_complementee->entree_C_adder
44 ecriture:busA->EB2
45 ecriture:busB->EB2
46 validation:EB2->entree_D_adder
47 generation_de_0_sur_entree_D_adder2
48 isolement_precharge_retenue_adder2
49 precharge_retenue_adder2
50 retenue_entrante_complementee_adder2
51 validation:indicateurs_arithmétiques->IA2
52 lecture:IA2->busB
53 entree_shift_up_spop2 = 0
54 shift_up_spop2
55 entree_shift_down_spop2 = 0
56 no_shift_spop2
57 shift_down_spop2
58 validation:resultat_operation_spop2->tampon_de_sortie_shifter
59 lecture:tampon_sortie_shifter2->busB
60 ecriture:busB->Rout(dtbus)
61 validation:Rout(dtbus)->plot_de_donnees
62 validation:plot_de_donnees->Rin(dtbus)
63 lecture:Rin(dtbus)->busA

6.Liste des fils de commande sortant de l'interface électrique entre partie opérative et partie contrôle

1 lecture_ecriture:d<->busB
2 validation:d<->busB
3 lecture:0->busB
4 ecriture:busB->EA1
5 retenue_entrante_incr1
6 shift_up_spop1
7 no_shift_spop1
8 lecture:tampon_sortie_shifter1->busB
9 communication:busB1<->busB2
10 lecture_ecriture:a<->busA
11 validation:a<->busA
12 lecture_ecriture:a<->busB
13 validation:a<->busB
14 lecture_ecriture:ad<->busA
15 validation:ad<->busA
16 lecture_ecriture:ad<->busB
17 validation:ad<->busB
18 ecriture:busA->b
19 lecture_ecriture:b<->busB
20 validation:b<->busB
21 lecture:c->busA
22 ecriture:busB->c
23 lecture_ecriture:pc<->busB
24 validation:pc<->busB
25 lecture:0->busB
26 validation:adbus->plot_de_sortie
27 ecriture:busA->adbus
28 ecriture:busB->adbus
29 ecriture:busA->ir
30 ecriture:busB->streg
31 ecriture:busA->EA2
32 ecriture:busB->EA2
33 multiplexeur:EA2->entree_C_adder
34 multiplexeur:EA2_complementee->entree_C_adder
35 ecriture:busA->EB2
36 ecriture:busB->EB2
37 validation:EB2->entree_D_adder
38 generation_de_0_sur_entree_D_adder2
39 retenue_entrante_complementee_adder2
40 lecture:IA2->busB
41 shift_up_spop2

42 no_shift_spop2
43 shift_down_spop2
44 lecture:tampon_sortie_shifter2->busB
45 ecriture:busB->Rout(dtbus)
46 validation:Rout(dtbus)->plot_de_donnees
47 validation:plot_de_donnees->Rin(dtbus)
48 lecture:Rin(dtbus)->busA

7.Liste des comptes rendus sortant de la partie opérative

1 bit 6 du registre ir
2 bit 5 du registre ir
3 bit 4 du registre ir
4 bit 3 du registre ir
5 bit 2 du registre ir
6 bit 1 du registre ir
7 bit 8 du registre streg
8 bit 7 du registre streg
9 bit 6 du registre streg
10 bit 5 du registre streg

8.Programmes d'assemblage de la partie opérative

po1.p : programme d'assemblage de la partie opérative, amplificateurs non compris.

```
#include "lubinclude.p"

begin

    i:=openfig('po1  ');
    i:=right(i,repy(getfig('gpeig '),8),1,10);
    i:=right(i,getfig('spo1  '),1,0);
    i:=right(i,repy(getfig('bcon  '),8),1,10);
    i:=right(i,getfig('spo2  '),1,0);
    callrec(0,6,deltax(i),4,'mm');
    i:=right(i,repy(getfig('dpeio '),8),1,10);
    i:=closefig(i);
    putfig(i);

end.
```

spo1.p : programme d'assemblage de la sous partie opérative numéro 1.

```
#include "lubinclude.p"

begin

    i:=openfig('spo1  ');
    i:=right(i,repy(getfig('sgp   '),8),1,10);
    i:=right(i,getfig('rp1   '),1,6);
    i:=right(i,getfig('in1   '),1,6);
    i:=right(i,getfig('al1   '),1,0);
    callrec(3,6,deltax(i)-3,4,'mm');
    i:=right(i,repy(getfig('sdp   '),8),1,10);
    i:=closefig(i);
    putfig(i);

end.
```

rp1.p : programme d'assemblage des registres et des constantes de la sous partie opérative numéro 1.

```
program rp1 (output,fich,lucie);
```

```
#include "lubinclude.p"

begin

    i:=openfig('rp1   ');
    k:=openfig('r1    ');
    k:=right(k,getfig('preab '),1,0);
    k:=right(k,getfig('rg4   '),1,0);
    k:=closefig(k);
    for j:=0 to nbtf do tf[j]:=-1;
    tf[0]:=getfig('cond  ');
    i:=right(i,cup(-1,repy(k,8),-1,-1,1,tf),1,0);
    i:=right(i,const(2,0,8,'ct1   '),1,4);
    i:=closefig(i);
    putfig(i);

end.
```

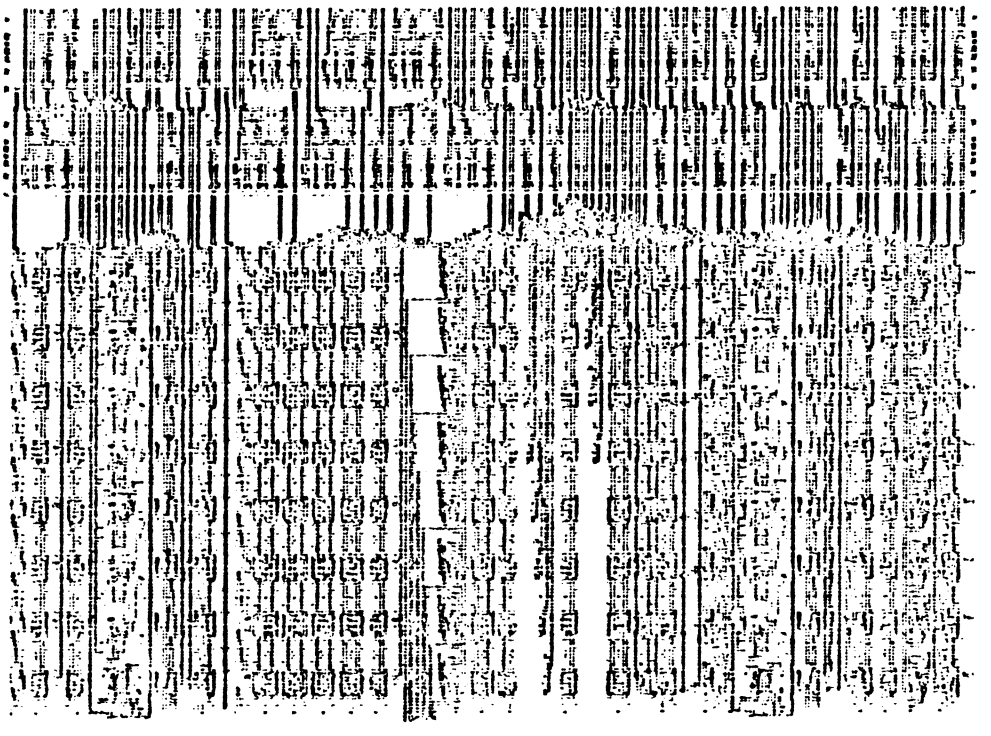
9.Fichier Lubrick des connecteurs de la partie opérative sans les amplificateurs

```
fig po1
  917  442
ce    8
    0  40   2   3  11
    0  94   2   3  11
    0 148   2   3  11
    0 202   2   3  11
    0 256   2   3  11
    0 310   2   3  11
    0 364   2   3  11
    0 418   2   3  11
cs    8
    0 379   2   3  10
    0 383   2   3  10
    0 387   2   3  10
    0 391   2   3  10
    0 395   2   3  10
    0 399   2   3  10
    0 403   2   3  10
    0 407   2   3  10
cn   63
    0  12   2   3   1
    0  37   2   3  20
  402  54   2   3   5
    0  64   2   3   3
    0  82   2   3  14
    0  86   2   3   2
    0 138   2   3   8
    0 154   2   3  15
    0 158   2   3   8
    0 168   2   3   8
    0 176   2   3   1
    0 194   2   3   5
    0 209   2   3   5
    0 224   2   3   1
    0 241   2   3  20
    0 253   2   3  20
    0 268   2   3  20
    0 280   2   3  20
    0 295   2   3   3
    0 307   2   3  20
```

```
  0  322   2   3   5
  0  334   2   3   3
  0  357   2   3  20
402  374   2   3   5
  0  434   2   3   8
  0  446   2   3   3
  0  458   2   3   3
  0  473   2   3   3
  0  498   2   3  10
  0  502   2   3  10
  0  506   2   3  10
  0  510   2   3  10
  0  514   2   3  10
  0  518   2   3  10
  0  537   2   3   3
  0  552   2   3  10
  0  556   2   3  10
  0  560   2   3  10
  0  564   2   3  10
  0  575   2   3   3
  0  587   2   3   3
  0  597   2   3   8
  0  609   2   3   8
  0  617   2   3   3
  0  631   2   3   3
  0  643   2   3   8
  0  658   2   3   8
  0  690   2   3  14
  0  694   2   3   2
  0  742   2   3   8
  0  746   2   3   1
  0  755   2   3   5
  0  762   2   3  15
  0  766   2   3   8
  0  776   2   3  15
  0  781   2   3   8
  0  785   2   3   8
  0  798   2   3   1
  0  816   2   3   5
  0  839   2   3   3
  0  851   2   3   8
  0  879   2   3   6
  0  897   2   3   5
bw 1
```

0 0 0 4 0
be 1
0 0 0 4 0
ffig

10. Dessin des masques de la partie opérative du microprocesseur simplifié



Description Apollon du 6502

registres : RI pch pcl rega regb regc regx regy sp streg;
entrées-sorties : busdata 0:8;
sorties : busadl 0:8 busadh 0:8;
champs de contrôle : RI 0:8 rega 7:1 regb 7:1 streg 0:3 , 6:2;
sous-pops : 3;
bits : 8;
figure : 6502;

busadl <- pcl / busadh <- pch / regb <- busdata;
busadl <- pcl / busadh <- pch / regc <- busdata;
busadl <- pcl / busadh <- pch;
alu_std(pcl 1 pcl + 1);
pch <- incr pch;
regb <- busdata / busadl <- regb / busadh <- regc;
busadl <- regb / busadh <- regc;
regb <- busdata / busadl <- regb / busadh <- 0;
busadl <- regb / busadh <- 0;
regb <- busdata / busadh <- 0 / busadl <- regc;
busadh <- 0 / busadl <- regc;
regc <- busdata / busadh <- 0 / busadl <- regc;
busadl <- sp / busadh <- 1 / rega <- busdata;
busadl <- sp / busadh <- 1;
busadl <- sp / busadh <- 1 / regx <- busdata;
busadl <- sp / busadh <- 1 / regy <- busdata;
busadl <- sp / busadh <- 1 / streg <- busdata;
busadl <- pcl / busadh <- pch / pch <- busdata;
busadl <- sp / busadh <- 1 / pch <- busdata;
busadl <- regb / pch <- busdata / busadh <- 255;
busadl <- regb / busadh <- 255;
busadl <- sp / busadh <- 1 / pcl <- busdata;
busadh <- 255 / pcl <- busdata / busadl <- rega;
busadh <- 255 / busadl <- rega;
busadl <- sp / busadh <- 1 / busdata <- rega;
busadl <- sp / busadh <- 1 / busdata <- regx;
busadl <- sp / busadh <- 1 / busdata <- regy;
busadl <- sp / busadh <- 1 / busdata <- streg;
busadl <- sp / busadh <- 1 / busdata <- pch;
busadl <- sp / busadh <- 1 / busdata <- pcl;
busdata <- rega ;

```
busdata <- regy ;
busdata <- regx ;
busdata <- 0;
busdata <- regc;
regc <- regb + regx;
regc <- incr regc;
regc <- incr regc / alu_std(regb regb regy + 2);
regb <- regc + regx;
regb <- regc + regy;
alu_std(regb regb regx + 3);
alu_std(regb regb regy + 4);
pcl <- regb / pch <- regc;
pcl <- regb;
alu_std(pcl pcl regx + 5);
sp <- decr sp;
sp <- incr sp;
streg <- 1 or streg / rega <- mult2 rega;
streg <- 254 and streg / rega <- mult2 rega;
regc <- 1 and rega;
rega <- div2 rega / streg <- regc or streg;
rega <- mult2 rega / regc <- 128 and rega;
streg <- 1 or streg / rega <- 1 or rega;
streg <- 254 and streg / rega <- 1 or rega;
streg <- 1 or streg;
streg <- 254 and streg;
regc <- 1 and rega / rega <- div2 rega;
streg <- regc or streg / rega <- 128 or rega;
streg <- regc or streg / rega <- 127 and rega;
rega <- decr rega;
rega <- incr rega;
alu_std(regb rega regb and 6) / regc <- not rega;
regb <- 192 and regb;
alu_std(regc regb regc and 6);
alu_std(regc rega regb or 6);
streg <- regb or streg;
alu_std(pcl pcl regb + 9);
streg <- 20 or streg / alu_std(pcl 2 pcl + 10);
rega <- regb;
regx <- regb;
regy <- regb;
alu_std(regc regx regb - 11);
alu_std(regc regy regb - 12);
rega <- rega + regb / regc <- 1 and streg;
regc <- 1 and streg / rega <- rega - regb;
```

```
rega <- rega or regb;
rega <- rega and regb;
rega <- rega exor regb;
rega <- rega + regc;
alu_std(regc rega regb - 13);
rega <- rega - regc;
streg <- 247 and streg / rega <- 252 / regb <- incr 252;
streg <- 4 or streg;
streg <- 64 or streg;
sp <- decr sp / streg <- 4 or streg;
sp <- decr sp / rega <- 250;
sp <- decr sp / rega <- 254;
regb <- incr rega;
busadl <- pcl / busadh <- pch / RI <- busdata;
alu_std(pcl 1 pcl + 14);
rega <- regx;
sp <- regx;
regx <- rega;
regx <- sp;
rega <- regy;
regy <- rega;
streg <- 1 or streg;
streg <- 254 and streg;
streg <- 247 and streg;
streg <- 8 or streg;
streg <- 251 and streg;
alu_std(regx regx 1 - 14);
alu_std(regy regy 1 - 14);
alu_std(regx 1 regx + 14);
alu_std(regy 1 regy + 14).
```

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974

VU les rapports de présentation de Messieurs

. B. COURTOIS, Chargé de recherche
. J.P MOREAU

Monsieur JAMIER Robert

est autorisé à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR-INGENIEUR, spécialité "Informatique".

Fait à Grenoble, le 19 novembre 1986

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,