Contribution à la compilation de silicium et au compilateur SYCO

A.A. Jerraya

THESE

PRESENTEE A

L'UNIVERSITE
JOSEPH FOURIER

L'INSTITUT
NATIONAL POLYTECHNIQUE

DE GRENOBLE

pour obtenir

LE TITRE DE DOCTEUR

ES SCIENCES "Informatique"

par:

Ahmed Amine JERRAYA

*
* *

CONTRIBUTION A LA COMPILATION DE SILICIUM ET AU COMPILATEUR SYCO

*
* *

Soutenue le 19 Décembre 1989 devant la commission d'Examen

JURY

Monsieur	J.P. Verjus,	PRESIDENT
Messieurs	F. Anceau B. Courtois M. Glesner F. Juttand C. Landrault C. Trullemans	EXAMINATEURS

Thèse préparée au sein du laboratoire TIM3, URA au CNRS N° 397

UNIVERSITE Joseph FOURIER (GRENOBLE I)

Président de l'Université :
M. NEMOZ Alain

Année Universitaire 1988 - 1989

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ère Classe

ADIBA Michel	Informatique
ANTOINE Pierre	Géologie I.R.I.G.M.
ARNAUD Paul	Chimie Organique
ARVIEU Robert	Physique Nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S
AURIAULT Jean-Louis	Mécanique
AYANT Yves	Physique Approfondie
BARBIER Marie-Jeanne	Electrochimie
BARJON Robert	Physique Nucléaire ISN
BARNOUD Fernand	Biochimie Macromoléculaire Végétale
BARRA Jean-René	Statistiques-Mathématiques Appliquées
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BELORISKY Elie	Physique
BENZAKEN Claude	Mathématiques Pures
BERARD Pierre	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean-Paul	Mathématiques Pures
BILLET Jean	Géographie
BOELHER Jean-Paul	Mécanique
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
CASTAING Bernard	Physique
CAUQUIS Georges	Chimie Organique
CHARDON Michel	Géographie
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean-Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEGRANGE Charles	Zoologie
DEMAILLY Jean-Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPORTES Charles	Chimie Minérale
DOLIQUE Jean-Michel	Physique des Plasmas
DOUCE Roland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FINKE Gerde	Informatique
GAGNAIRE Didier	Chimie Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GERMAIN Jean-Pierre	Mécanique,
GIDON Maurice	Géologie
GUITTON Jacques	Chimie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean René	Mathématiques Pures

JOSELEAU Jean Paul	Biochimie
KAHANE André, détaché	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées
LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre-Jean	Mathématiques Appliquées
LEBRETON Alain	Mathématiques Appliquées
DE LEIRIS Joël	Biologie
LHOMME Jean	Chimie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences Nucléaires I.S.N.
LONGEQUEUE Nicole	Physique
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du Solide
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PANNETIER Jean	Chimie
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRIER Guy	Géophysique
PIERRE Jean Louis	Chimie Organique
RENARD Michel	Thermodynamique
RIEDTMANN Christine	Mathématiques
RINAUDO Marguerite	Chimie CERMAV
ROSSI André	Biologie
SAXOD Raymond	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
SOUTIF Michel	Physique
STUTZ Pierre	Mécanique
TRILLING Laurent	Mathématiques Appliquées
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS de 2ème Classe

ARMAND Gilbert	Géographie
ATTANE Pierre	Mécanique
BARET Paul	Chimie
BERTIN José	Mathématiques
BLANCHI J.Pierre	STAPS
BLOCK Marc	Biologie
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BORRIONE Dominique	Automatique informatique
BOUVET Jean	Biologie
BROSSARD Jean	Mathématiques
BRUANDET J.François	Physique
BRUGAL Gérard	Biologie
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
CHOLLET Jean Pierre	Mécanique
COLOMBEAU Jean François	Mathématiques (ENSL)
COURT Jean	Chimie
CUNIN Pierre Yves	Informatique
DAVID Jean	Géographie

DHOUAILLY Danielle	Biologie
DUFRESNOY Alain	Mathématiques Pures
GASPARD François	Physique
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GONZALEZ SPRINBERG Gérardo	Mathématiques Pures
GUIGO Maryse	Géographie
GUMUCHAIN Hervé	Géographie
HACQUES Gérard	Mathématiques Appliquées
HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
HERINO Roland	Physique
JARDON Pierre	Chimie
KERCKHOVE Claude	Géologie
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
MOREL Alain	Géographie
NEMOZ Alain	Thermodymamique CNRS - CRTBT
NGUYEN HUY Xuong	Informatique
OUDET Bruno	Mathématiques Appliquées
PAUTOU Guy	Biologie
PECHER Arnaud	Géologie
PELMONT Jean	Biochimie
PELLETIER Guy	Astrophysique
PERRIN Claude	Sciences Nucléaires I.S.N.
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
REGNARD Jean René	Physique
RICHARD Jean-Marc	Physique
RIEDTMANN Christine	Mathématiques Pures
ROBERT Danielle	Chimie
ROBERT Gilles	Mathématiques Pures
ROBERT Jean-Bernard	Chimie Physique
SARROT-REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SCHOLL Pierre-Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
VALLADE Marcel	Physique
VIDAL Michel	Chimie Organique
VINCENT Gilbert	Physique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L' IUT 1

PROFESSEURS de 1ère Classe

BUISSON Roger	Physique IUT 1
CHEHIKIAN Alain	E.E.A. I.U.T.1
DODU Jacques	Mécanique Appliquée IUT 1
NEGRE Robert	Génie Civil IUT 1
NOUGARET Marcel	Automatique IUT 1
PERARD Jacques	EEA. IUT 1

PROFESSEURS de 2ème classe

BEE Marc	Physique IUT 1
BOUTHINON Michel	EEA. IUT 1
CHAMBON René	Génie Mécanique IUT 1
CHENAVAS Jean	Physique IUT 1

CHILO Jean	Physique IUT 1
CHOUTEAU Gérard	Physique IUT 1
CONTE René	Physique IUT 1
FOSTER Panayotis	Chimie IUT 1
GOSSE Jean-Pierre	EEA.IUT 1
GROS Yves	Physique IUT 1
HAMAR Roger	Chimie IUT 1
KUHN Gérard, (Détaché)	Physique IUT 1
LEVIEL Jean Louis	Physique IUT 1
MAZUER Jean	Physique IUT 1
MICHOULIER Jean	Physique IUT 1
MONLLOR Christian	EEA.IUT 1
PERRAUD Robert	Chimie IUT 1
PIERRE Gérard	Chimie IUT 1
TERRIEZ Jean-Michel	Génie Mécanique IUT 1
TOUZAIN Philippe	Chimie IUT 1
TURGEMAN Sylvain	Génie civil
VINCENDON Marc	Chimie IUT 1
ZIGONE Michel	Physique IUT 1

PROFESSEURS DE PHARMACIE

AGNIUS-DELORD Claudine	Physique	Faculté La Tronche
ALARY Josette	Chimie Analytique	Faculté La Tronche
BERIEL Hélène	Physiologie et Pharmacologie	Faculté La Tronche
CUSSAC Max	Chimie Therapeutique	Faculté La Tronche
DEMENGE Pierre	Pharmacodynamie	Faculté La Tronche
FAVIER Alain	Biochimie	C.H.R.G.
JEANNIN Charles	Pharmacie Galénique	Faculté Meylan
LATURAZE Jean	Biochimie	Faculté La Tronche
LUU DUC Cuong	Chimie Générale	Faculté La Tronche
MARIOTTE Anne-Marie	Pharmacognosie	Faculté La Tronche
MARZIN Daniel	Toxicologie	Faculté Meylan
RENAUDET Jacqueline	Bactériologie	Faculté La Tronche
ROCHAT Jacques	Hygiéne et Hydrologie	Faculté La Tronche
SEIGLE-MURANDI Françoise	Botanique et Cryptogamie	Faculté Meylan
VERAIN Alice	Pharmacie Galénique	Faculté Meylan

MEMBRES DU CORPS ENSEIGNANT DE MEDECINE

PROFESSEURS CLASSE EXEPTIONNELLE ET 1ère CLASSE

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatrie-Puericulture	C.H.R.G.
BEZEZ Henri	Orthopédie-Traumatologie	Hopital SUD
BONNET Jean-Louis	Ophtalmologie	C.H.R.G.
BOUCHET Yves	Anatomie	Faculté La Merci
	Chirurgie Générale et Digestive	C.H.R.G.
BUTEL Jean	Orthopédie-Traumatologie	C.H.R.G.
CHAMBAZ Edmond	Biochimie	C.H.R.G.
CHAMPETIER Jean	Anatomie-Topographique et Appliquée	C.H.R.G.
CHARACHON Robert	O.R.L.	C.H.R.G.
COLOMB Maurice	Immunologie	Hopital sud
COUDERC Pierre	Anatomie-Pathologique	C.H.R.G.
DELORMAS Pierre	Pneumophtisiologie	C.H.R.G.
DENIS Bernard	Cardiologie	C.H.R.G.
GAVEND Michel	Pharmacologie	Faculté La Merci

HOLLARD Daniel	Hématologie	C.H.R.G.
LATREILLE René	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
LE NOC Pierre	Bactériologie-Virologie	C.H.R.G.
MALINAS Yves	Gynécologie et Obstétrique	C.H.R.G.
MALLION Jean-Michel	Médecine du Travail	C.H.R.G.
MICOUD Max	Clinique Médicale et Maladies Infectieuses	C.H.R.G.
MOURIQUAND Claude	Histologie	Faculté La Merci
PARAMELLE Bernard	Pneumologie	C.H.R.G.
PERRET Jean	Neurologie	C.H.R.G.
RACHAIL Michel	Hépato-Gastro-Entérologie	C.H.R.G.
DE ROUGEMONT Jacques	Neurochirurgie	C.H.R.G.
SARRAZIN Roger	Clinique Chirurgicale	C.H.R.G.
STIEGLITZ Paul	Anestésiologie	C.H.R.G.
TANCHE Maurice	Physiologie	Faculté La Merci
VIGNAIS Pierre	Biochimie	Faculté La Merci

PROFESSEURS 2ème CLASSE

BACHELOT Yvan	Endocrinologie	C.H.R.G.
BARGE Michel	Neurochirurgie	C.H.R.G.
BENABID Alim Louis	Biophysique	Faculté La Merci
BENSA Jean-Claude	Immunologie	Hopital Sud
BERNARD Pierre	Gynécologie-Obstétrique	C.H.R.G.
BESSARD Germain	Pharmacologie	ABIDJAN
BOLLA Michel	Radiothérapie	C.H.R.G.
BOST Michel	Pédiatrie	C.H.R.G.
BOUCHARLAT Jacques	Psychiatrie Adultes	Hopital Sud
BRAMBILLA Christian	Pneumologie	C.H.R.G.
CHIROSSEL Jean-Paul	Anatomie-Neurochirurgie	C.H.R.G.
COMET Michel	Biophysique	Faculté La Merci
CONTAMIN Charles	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
CORDONNIER Daniel	Néphrologie	C.H.R.G.
COULOMB Max	Radiologie	C.H.R.G.
CROUZET Guy	Radiologie	C.H.R.G.
DEBRU Jean-Luc	Médecine Interne et Toxicologie	C.H.R.G.
DEMONGEOT Jacques	Biostatistiques et Informatique Médicale	Faculté La Merci
DUPRE Alain	Chirurgie Générale	C.H.R.G.
DYON Jean-François	Chirurgie Infantile	C.H.R.G.
ETERRADOSSI Jacqueline	Physiologie	Faculté La Merci
FAURE Claude	Anatomie et Organogénèse	C.H.R.G.
FAURE Gilbert	Urologie	C.H.R.G.
FOURNET Jacques	Hépato-Gastro-Entérologie	C.H.R.G.
FRANCO Alain	Médecine Interne	C.H.R.G.
GIRARDET Pierre	Anesthésiologie	C.H.R.G.
GUIDICELLI Henri	Chirurgie Générale et Vasculaire	C.H.R.G.
GUIGNIER Michel	Thérapeutique et Réanimation Médicale	C.H.R.G.
HADJIAN Arthur	Biochimie	Faculté La Merci
HALIMI Serge	Endocrinologie et Maladies Métaboliques	C.H.R.G.
HOSTEIN Jean	Hépato-Gastro-Entérologie	C.H.R.G.
HUGONOT Robert	Médecine Interne	C.H.R.G.
JALBERT Pierre	Histologie-Cytogénétique	C.H.R.G.
JUNIEN-LAVILLAUROY Claude	O.R.L.	C.H.R.G.
KOLODIE Lucien	Hématologie Biologique	C.H.R.G.
LETOUBLON Christian	Chirurgie Générale	C.H.R.G.
MACHECOURT Jacques	Cardiologie et Maladies Vasculaires	C.H.R.G.
MAGNIN Robert	Hygiène	C.H.R.G.
MASSOT Christian	Médecine Interne	C.H.R.G.

MOUILLON Michel	Ophtalmologie	C.H.R.G.
PELLAT Jacques	Neurologie	C.H.R.G.
PHELIP Xavier	Rhumatologie	C.H.R.G.
RACINET Claude	Gynécologie-Obstétrique	Hopital Sud
RAMBAUD Pierre	Pédiatrie	C.H.R.G.
RAPHAEL Bernard	Stomatologie	C.H.R.G.
SCHAERER René	Cancérologie	C.H.R.G.
SEIGNEURIN Jean-Marie	Bactériologie-Virologie	Faculté La Merci
SELE Bernard	Cytogénétique	Faculté La Merci
SOTTO Jean-Jacques	Hématologie	C.H.R.G.
STOEBNER Pierre	Anatomie Pathologique	C.H.R.G.
VROUSOS Constantin	Radiothérapie	C.H.R.G.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD **Année 1988**

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Marc	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FOGGIA Albert	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 Jean-Claude	ENSPG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSIEG
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
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
RAMEAU Jean-Jacques	ENSEEG
RENAUD Maurice	UFR PGP
ROBERT André	UFR PGP
ROBERT François	ENSIMAG
SABONNADIERE Jean-Claude	ENSIEG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
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
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
PASTUREL Alain
PLA Fernand
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri

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

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

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
COURTOIS Bernard
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GUELIN Pierre
JOUD Jean-Charles
KLEITZ Michel
KOFMAN Walter
KAMARINOS Georges
LEJEUNE Gérard
LE PROVOST Christian
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MUNIER Jacques
PIAU Monique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
WACK Bernard

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

E.N.S.E.E.G
CHATILLON Christian
HAMMOU Abdelkader
MARTIN GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G
BOREL Joseph

E.N.S.I.E.G
DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G
ROWE Alain

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

E.F.P.
CHARUEL Robert

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

Laboratoires extérieurs

C.N.E.T
DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

Je voudrais exprimer toute ma reconnaissance aux membres du jury:

Monsieur Jean Pierre Verjus, professeur à l'INPG et directeur de l'IMAG qui m'a fait l'honneur de présider le jury.

Monsieur François Anceau, responsable des programmes avancées au centre de recherches de Bull Louveciennes, qui est à l'origine de ce travail. Je le remercie pour la confiance qu'il m'a accordée en dirigeant mes premiers travaux de recherches et la première année des travaux de cette thèse.

Monsieur Bernard Courtois, directeur de recherche au CNRS, qui dirige l'équipe d'architecture des ordinateurs de TIM3 et qui a dirigé en grande partie le travaux de cette thèse en assumant la succession de François Anceau après son départ à Bull. Il n'a cessé de travailler pour procurer aux membres de cette équipe des conditions de travail excellentes.

Messieurs Christian Landrault, Directeur de recherche au CNRS, et Francis Juttand, Professeur à l'ENST de Paris, d'avoir accepté d'être rapporteurs, et dont les critiques m'ont permis d'améliorer la version finale de cette thèse.

Messieurs Charles Trullemens, professeur à l'université catholique de Louvain la Neuve, et Manfred Glesner, professeur au technische hochschule à Darmstadt, qui ont acceptés de faire le déplacement pour participer au jury de cette thèse.

Je tiens à remercier aussi

- Tous ceux qui ont participé de près ou de loin au développement du projet SYCO et plus particulièrement, et par ordre alphabétique, C. Arzounian, N. Bekkara, Ph. Bondono, E. Bourcier, J.P. Geronimi, L. Harivel, A. Hornick, R. Jamier, F. Martinez, D. Macq, N. Mhaya, K. O'brien, I. Park, F.R. Rougeaux, P.Varinot et j'en oublie certainement.

- Mes amis et collègues Jacques Laurent et Meryem Marzouki pour la patience dont ils ont fait preuve pour la relecture du texte de cette thèse. J'admire le courage avec lequel ils ont affrontés la première version du manuscrit.

- Tous mes camarades de l'équipe d'architecture des ordinateur, et plus particulièrement Alain Guyot, qui ont participés, par de nombreuses discussions, à l'enrichissement de ce travail.

- Et enfin, Mlle Lydie Sanz pour sa participation à la frappe du manuscrit et le service de reprographie de l'IMAG pour l'exellente qualité de son travail.

Nul être ne peut tous ses espoirs réaliser,
La course des vents contrarie les désirs des voiliers.

(Al Moutanabbi
Poète Arabe 915-965)

ماكل مايتمنى المرء يدركه

تجري الرياح بما لا تشتهي السفن

Contribution à la compilation de silicium et au compilateur Syco

Résumé:

L'objet de cette thèse est la conception d'outils pour l'automatisation du processus de conception des circuits intégrés. Ces outils sont appelés compilateurs de silicium. Le premier chapitre décrit brièvement le contexte de cette thèse. Le second chapitre est une introduction générale à la compilation de silicium. La deuxième partie de la thèse est consacrée aux compilateurs de comportements, ils permettent de générer l'architecture d'un circuit en partant de sa description comportementale. Les techniques mises en œuvre par ces compilateurs sont discutées dans le troisième chapitre. Le quatrième chapitre présente le compilateur de silicium Syco. Syco permet de générer la description physique d'un circuit en partant de sa description comportementale. Le processus de compilation a été simplifié par l'utilisation d'un certain nombre de modèles pré-définis. Les circuits générés sont composés d'une partie opérative parallèle et d'une partie contrôle hiérarchique.

Mots clefs:

compilation de silicium, description comportementale, architecture, circuits intégrés.

Silicon compilation and the Syco Silicon compiler

Abstract:

The present work deals with silicon compilers which are tools that perform the automatic synthesis of integrated circuits. The first chapter describe the content of this work. Silicon compilation is introduced by the second one. Several compilation schemes and compilation levels are stand out. The third chapter deals with behavioural silicon compilation techniques. Finally, the Syco silicon compiler is detailed within the fourth chapter. The translation scheme used by Syco (from behavior to layout) is easy to understand thanks to the correspondence between the target architecture and the behavioral description.

Keywords:

silicon compilation, behavioral description, architecture, integrated circuits.

TABLE DES MATIERES

TABLE DES FIGURES:

Figures **Page**

CHAPITRE I:

INTRODUCTION

Cette thèse traite de la compilation de silicium, c'est à dire l'automatisation du processus de conception des circuits intégrés. Elle est avant tout le résultat d'un travail collectif effectué dans le cadre de la réalisation du compilateur de silicium SYCO.

Le processus général de conception d'un circuit intégré, allant d'une spécification de haut niveau au dessin des masques, est difficilement maîtrisable dans sa totalité. D'autre part, le nombre de réalisations physiques (dessin des masques) possibles correspondant à une spécification donnée est très grand. Ces deux facteurs sont à conjuguer à l'accroissement de la complexité des circuits intégrés.

Il y a quelques années, les informaticiens étaient confrontés au même problème pour concevoir des programmes. Cette situation fut contournée par une double évolution :

• L'utilisation de langages évolués a permis d'élever le niveau d'abstraction des primitives manipulées par un programme. Ainsi, la réalisation d'un programme peut se faire indépendamment de certains détails relatifs à la machine cible. On peut remarquer aussi que cette évolution a entraîné une meilleure compréhension des concepts manipulés par les programmes et, par la suite, une standardisation des constructions utilisées par les langages de programmation.

• La définition de méthodologies pour structurer le processus de conception a permis de maîtriser la réalisation de gros programmes.

La conjugaison de ces évolutions et du développement des machines a accéléré le développement des compilateurs de logiciel. En fait, un compilateur n'est rien d'autre qu'un programme permettant de traduire un programme-source, spécifié dans un langage de haut niveau, en un programme-objet, directement exécutable sur une machine donnée.

L'évolution des techniques de conception de circuits intégrés semble profiter de cette expérience, et suivre le même chemin. Les essais de développement de méthodologies de conception et de définition de langages de spécification de haut niveau ont entraîné le développement actuel des compilateurs de silicium. Voici donc la genèse de cette thèse. Elle est la suite logique des travaux réalisés par F. Anceau, pour la définition de CAPRI, une méthodologie pour la conception de circuits de type microprocesseurs [ANC83], et pour la définition de IRENE, un langage pour la spécification de la structure et du comportement des circuits intégrés [MAR86].

L'utilisation des compilateurs de logiciel a entraîné une séparation entre les techniques de réalisation des compilateurs et de programmes d'application. Cette séparation permet au programmeur de se concentrer sur la spécification de son application dans un langage de haut

niveau, tout en se libérant de la rude besogne consistant à traduire cette spécification dans un langage-machine. Malheureusement, les compilateurs de silicium actuels n'ont pas encore atteint le degré de maturité nécessaire pour mériter la confiance des concepteurs, comme c'est le cas des compilateurs de langages de programmation. Une meilleure formalisation des méthodes de conception,et surtout une meilleure définition des niveaux d'abstraction, pour la représentation des circuits intégrés, restent à trouver.

Dans le cas des circuits intégrés, plusieurs niveaux d'abstraction sont actuellement des candidats potentiels pour la spécification des circuits. Trois niveaux, en plus du niveau physique qui permet de spécifier un circuit avec tous les détails nécessaires pour sa fabrication, peuvent être considérés. Le niveau structurel permet de spécifier un circuit dans le cadre d'une famille technologique donnée et indépendamment du processus de fabrication. Le niveau architectural permet de représenter le circuit indépendamment de la technologie. Le niveau comportemental décrit la fonction du circuit indépendamment de son architecture. Ces trois niveaux définissent trois types de compilateurs de silicium : les compilateurs de structure, les compilateurs d'architecture et les compilateurs de comportement.

Le deuxième chapitre de cette thèse est une introduction à la compilation de silicium. Une étude des méthodologies de conception de circuits intégrés permettra de définir le contexte actuel dans lequel s'inscrivent les outils de compilation de silicium. Deux classes de méthodologies de conception seront considérées, la première concernant les circuits de commande qui réalisent un traitement complexe sur un flux d'informations faible et la seconde se rapportant aux circuits de communication qui réalisent un traitement peu complexe sur un flux d'information important. Le choix des niveaux d'abstraction, cités ci-dessus, sera justifié par l'étude des étapes qui constituent généralement le processus de conception d'un circuit intégré. Ce chapitre contient également une présentation de l'état de l'art de la compilation de silicium. Cette présentation est un essai de classification des compilateurs de silicium actuels, à la fois selon leur langage d'entrée et selon leur organisation interne.

Le troisième chapitre est consacré à la compilation de comportement,c'est à dire, la génération de l'architecture d'un circuit à partir de sa description comportementale. Les techniques utilisées pour la compilation de comportement sont décrites. Elles sont introduites de manière informelle à l'aide de deux exemples de compilation. Le premier concerne un circuit de contrôle et le second un circuit de communication. Bien que les deux processus de compilation mettent en œuvre les mêmes techniques de base, ils utilisent des schémas de compilation différents. Dans le cas de la compilation d'un circuit de commande, l'accent est mis sur la compilation de la partie contrôle, tandis que dans le cas de la compilation d'un circuit de communication la partie contrôle est sacrifiée au profit de la partie opérative. Même si plusieurs des techniques utilisées sont bien maîtrisées, l'inadéquation des langages de description de comportement utilisés semble être le principal frein au développement des compilateurs de

comportement.

Le quatrième chapitre présente le compilateur de silicium SYCO développé depuis 1984 au sein de l'Equipe d'Architecture des Ordinateurs du laboratoire TIM3 de l'IMAG. SYCO permet de générer la description physique d'un circuit en partant de sa description comportementale. En plus de l'étape de compilation de comportement, qui permet de générer l'architecture, il réalise les étapes de compilation d'architecture et de compilation de structure. Le processus de compilation a été simplifié par l'utilisation d'un certain nombre de modèles prédéfinis. Les circuits générés sont composés d'une partie opérative parallèle et d'une partie contrôle hiérarchique. Le choix de cette architecture provient du projet CAPRI, dont le but était de définir une méthodologie pour la conception des circuits de type microprocesseurs [ANC83]. La réalisation du compilateur SYCO est un travail collectif auquel plus de vingt personnes ont participé directement. Détailler les participations de chacun nécessiterait un chapitre supplémentaire dans cette thèse. L'auteur remercie tous ceux qui ont participé à la réalisation du compilateur SYCO : C. Arzounian, N. Bekkara, Ph. Bondono, E. Bourcier, J.P. Geronimi, L. Harivel, A. Hornick, R. Jamier, F. Martinez, N. Mhaya, F.R. Rougeaux, P.Varinot, et j'en oublie certainement.

CHAPITRE II:

INTRODUCTION A LA COMPILATION DE SILICIUM

Le terme compilation de silicium a été introduit pour la première fois par D. JOHANSSEN [JOH79] pour désigner l'assemblage de cellules paramétrées. Depuis, le terme s'est popularisé et a été utilisé dans plusieurs contextes différents.

Le plus souvent, un compilateur de silicium est défini comme étant un système permettant de générer le dessin des masques d'un circuit intégré en partant d'une description de haut niveau [GAJ87]. Cette description peut être un algorithme, une architecture, un dessin symbolique...

Pour la suite, on retiendra une définition plus large : un compilateur de silicium est un système permettant de transformer la représentation d'un circuit ou d'une partie de circuit en une autre représentation de ce circuit, d'un niveau d'abstraction inférieur.

La notion de niveau d'abstraction sera introduite par la première section de ce chapitre. Cette section traite des méthodologies de conception de circuits intégrés. La présence de cette section se justifie par le fait qu'un compilateur de silicium permet d'automatiser une partie d'un processus de conception qui est généralement basé sur une méthodologie de conception.

La définition d'un compilateur de silicium, telle que donnée ci dessus, couvre une large gamme d'outils. La bibliographie sur le sujet est donc abondante et variée. La suite de ce chapitre contient une tentative de classification des compilateurs de silicium. Deux critères de classification seront retenus : le langage d'entrée du compilateur et l'organisation de ce dernier. La deuxième section définit les différents niveaux d'abstraction retenus et les compilateurs associés à ces niveaux. Elle distingue les compilateurs de comportement, les compilateurs d'architecture et les compilateurs de structure. La troisième section décrit trois organisations possibles pour un compilateur de silicium. Pour chaque niveau d'abstraction, plusieurs organisations seront considérées pour la réalisation d'un compilateur de silicium. Une classification des compilateurs de silicium existants est présentée dans la quatrième section.

II.1 Méthodologies de conception

Le processus de conception d'un circuit intégré, allant d'une description comportementale de haut niveau au dessin des masques est, dans le cas général, difficilement maîtrisable dans sa totalité. Il nécessite des connaissances dans plusieurs domaines très différents qui vont de la physique des semi-conducteurs aux techniques de programmation [SOU87]. D'autre part, le nombre de réalisations physiques (dessins des masques) possibles et correspondant à la description comportementale d'un circuit donné est très grand.

La définition de démarches standards pour la conception de circuits intégrés permet de

mieux maîtriser la complexité du processus de conception. Ces démarches sont généralement appelées méthodologies de conception. Elles permettent de limiter les domaines de connaissances requis en définissant des primitives permettant de cacher les détails de la conception. L'utilisation d'une bibliothèque de cellules standards, par exemple, permet de concevoir un circuit sans se préoccuper des détails de fabrication. Le choix des primitives de base va définir ce qu'on appelle un style de conception [NEW87]. On peut aussi limiter le choix des réalisations en fixant une architecture standard, le plus souvent appelée architecture cible. Ceci limite le domaine du processus de conception à des applications bien particulières.

II.1.1 Méthodologie et style de conception

Un style de conception correspond à un mode d'organisation des C.I. On distingue quatre types d'organisation [FEY86], [SAN87] :

- Les circuits organisés en réseaux programmables,
- Les circuits organisés en cellules pré-caractérisées,
- Les circuits organisés en macro-blocs,
- Les circuits organisés à la demande.

Un réseau programmable est un ensemble de cellules répétées dans une structure de tableau à deux dimensions. La personnalisation d'un tel réseau se fait par l'ajout ou la suppression d'éléments d'interconnexion. Comme exemple de réseaux programmables, on peut citer les PLAs, les réseaux pré-diffusés, les réseaux pré-dessinés ("sea-of-gates") [HUI85] et les ROMs. Un circuit est alors un réseau programmable personnalisé.

Un circuit pré-caractérisé est un assemblage de cellules prédéfinies (appelées aussi cellules standards ou pré-caractérisées). La conception de ces cellules est indépendante de celle des circuits qu'elles peuvent constituer. Elles sont généralement peu complexes, rarement plus qu'une porte de base ou qu'une bascule. Les cellules sont le plus souvent organisées selon une topologie standard afin de faciliter leur assemblage.

Les méthodologies de conception de C.I. qui utilisent les réseaux programmables ou les cellules standards ne permettent pas des réalisations efficaces de certaines classes de circuits [SAN87] telles que les mémoires (RAM) par exemple. Elles consistent généralement à transformer, dans un premier temps, la spécification du circuit en une description qui utilise uniquement des primitives correspondant aux cellules de base (portes et bascules). Dans un deuxième temps, une phase d'allocation permet d'associer à chaque élément de la description une cellule parmi l'ensemble des cellules prédéfinies. Dans le cas des réseaux programmables, une cellule est désignée par son emplacement. Dans le cas des cellules standards, une étape de placement des cellules est nécessaire avant de réaliser les connexions physiques entre les cellules. Le fait que la même méthodologie soit utilisée pour ces deux styles de circuits fait que,

dans plusieurs cas, on commence par réaliser le circuit en réseau programmable. C'est la méthode la plus rapide ; par la suite on réalise une version en pré-caractérisé. En fait, la première étape permet de réaliser le circuit au plus vite et à moindre frais, puis une seconde version permet d'améliorer les performances du circuit [JON86].

Le troisième style de conception utilise des macro-blocs (ou macro-cellules) qui réalisent des fonctions complexes telles que des mémoires ou des opérateurs. Un macro-bloc peut être un réseau programmable tel qu'une ROM ou un PLA. Il peut aussi réaliser une simple cellule. Aucune restriction n'est faite quant à l'organisation et à la taille de ces macro-blocs.

La conception d'un circuit à l'aide de macro-blocs consiste à partager la description du circuit en fonctions réalisables par des éléments pris dans une bibliothèque de macro-blocs.

Le fait de paramétrer les macro-blocs permet de rendre leur utilisation plus souple. Un macro-bloc peut avoir quatre types de paramètres :

- Des paramètres pour personnaliser la fonction (table de vérité pour un PLA),
- Des paramètres pour personnaliser la structure (nombre de bits pour un registre),
- Des paramètres pour personnaliser la topologie (facteurs de formes, emplacement des connecteurs),
- Des paramètres pour personnaliser des caractéristiques physiques (consommation, vitesse, surface).

De plus en plus, les méthodes de conception de circuits intégrés permettent de mélanger , dans un même circuit, les trois styles cités plus haut.

Le quatrième style est appelé organisation à la demande. Le processus de conception n'utilise pas nécessairement des modèles prédéfinis. Les éléments de base qui constituent le circuit sont définis durant le processus de conception. Ce type d'organisation permet d'obtenir de meilleures performances (en terme de compromis surface/vitesse/consommation) et par la suite de concevoir des circuits plus complexes. Par contre, le coût de conception est plus élevé que pour les autres styles.

La table de la figure II.1, tirée de [FEY87], montre la productivité moyenne d'un concepteur selon le style de conception. Les auteurs de [FEY86] et [FEY87] donnent de plus amples détails sur ces quatre styles de conception

On peut trouver étonnant que l'utilisation des macro-blocs entraîne une productivité très proche de celle obtenue pour des circuits organisés à la demande. Ceci vient du fait que les chiffres de la figure II.1 tiennent compte des outils de CAO existants : en raison de l'absence d'outils d'aide à la génération d'architecture, le goulet d'étranglement des processus de conception actuels se situe au niveau de la définition de l'architecture.

Style de conception	Productivité moyenne porte/jour
Réseaux prédiffusés	25
Cellules précaractérisées	13
Macro-bloc	4
Organisation à la demande	2.7

Figure II.1 Productivité moyenne d'un concepteur [FEY87]

II.1.2 Méthodologies de conception et types d'applications

Le fait de limiter les champs d'application d'une méthodologie à une classe de circuit permet à la fois de réduire le coût du processus de conception et de réaliser des circuits performants. En général, on distingue deux classes de circuits : les circuits dédiés à la communication et ceux dédiés à la commande. Les premiers réalisent un traitement simple sur un grand flux d'informations. Ils nécessitent généralement une organisation parallèle pour pouvoir respecter la vitesse des flux d'informations. Ils sont souvent appelés circuits de traitement du signal. Les circuits dédiés à la commande sont dits de type microprocesseurs ou circuits de contrôle. Ils réalisent un traitement complexe sur un faible flux d'informations et sont généralement programmables.

Plusieurs méthodologies ont été développées pour ces deux classes de circuits. Ces méthodologies représentent souvent une première étape vers l'automatisation du processus de conception.

Pour les méthodologies de conception de circuits de communication on peut citer celles qui ont abouti aux systèmes CATHEDRAL [DeM87], [RAB86] et FIRST [DEN85]. Ces deux méthodes partent d'une description des flux de données dans le circuit, utilisée pour générer une architecture avec un maximum de parallélisme. Dans le cas du système CATHEDRAL, pour tenir compte de certaines contraintes fixées par l'utilisateur, cette première solution peut être améliorée dans un deuxième temps. La méthodologie utilisée par FIRST est beaucoup plus restrictive, elle a recours à des opérateurs, travaillant sur un seul bit, qui sont assemblés de manière standard.

Pour les méthodologies de conception de systèmes de commande, on peut citer CAPRI [ANC83], à l'origine du compilateur SYCO, et la méthodologie des diagrammes [TRE87]. Ces

méthodologies considèrent séparément la partie opérative et la partie contrôle d'un circuit, et partent d'une description algorithmique. La première étape du processus de conception fixe l'architecture de la partie opérative. Cette étape consiste essentiellement à déterminer le degré de parallélisme de la partie opérative. La seconde étape génère la description de la partie contrôle en partant de la description initiale et de la partie opérative. La méthode des diagrammes se limite à des parties contrôles microprogrammées et permet une évolution de l'architecture de la partie opérative durant la conception de la partie contrôle. CAPRI considère plusieurs architectures possibles pour la partie contrôle [OBR82]. La partie opérative est fixée une fois pour toutes dans le cas de la méthodologie CAPRI. Cette rigidité doit disparaître dans la prochaine version du compilateur SYCO. Il sera donc possible de modifier la partie opérative et la partie contrôle, afin de pouvoir réaliser des améliorations de l'architecture [BEK87].

Ces méthodologies permettent d'obtenir de manière quasi systématique l'organisation détaillée du circuit en partant d'une spécification de haut niveau. Chacune de ces méthodologies est optimisée pour une classe d'applications.

II.1.3 Niveaux d'abstraction et domaines de description

Durant le processus de conception d'un circuit, plusieurs langages de description de circuits intégrés peuvent être utilisés. Ces langages peuvent être classés selon les niveaux d'abstraction qu'ils manipulent, et selon les domaines de description qu'ils couvrent. L'état d'avancement du processus de conception définit le niveau d'abstraction du langage à utiliser, tandis que les aspects du circuit pris en compte définissent les domaines de description auquel appartient le langage.

II.1.3.1 Niveaux d'abstraction

Les niveaux d'abstractions correspondent aux étapes de la conception, leur nombre et leur définition vont donc dépendre de la méthodologie de conception utilisée.

Un niveau d'abstraction est caractérisé par les objets de base qu'il manipule. Un objet de base peut être un rectangle, un transistor, un opérateur complexe, etc. A chaque niveau d'abstraction correspond un langage de description de matériels. La plupart des travaux sur les niveaux d'abstraction [BAR75], [THO83], [ULL84] et [GAJ87] distinguent les niveaux suivants

- Niveau géométrique : c'est le dessin des masques. Cette description est suffisante pour la fabrication d'un circuit. L'élément de base à ce niveau est le polygone [MEA80].
- Niveau circuit : à ce niveau, on considère des objets primitifs, ayant une sémantique électrique, tels que le transistor, la résistance, la diode. Les langages de niveau interrupteur ("switch level") appartiennent également à cette catégorie ; ils permettent de décrire des circuits

plus grands avec moins de détails [CAI87].

Les essais de regroupement des niveaux géométrique et circuit ont donné naissance aux langages de description symbolique. On peut trouver une bibliographie sur ces langages dans [ROU87].

• Niveau logique : ici, un circuit peut être décrit comme un ensemble de portes logiques et d'éléments de mémorisation interconnectés. Ce niveau a fait l'objet d'une attention particulière dans le passé (durant les années 70) pour plusieurs raisons :

- Le niveau logique correspond au niveau d'abstraction le plus bas qui soit commun aux méthodes de conception de circuits intégrés et aux méthodes de conception de circuits imprimés à base de composants discrets.

- Ce niveau correspond le mieux au niveau de la saisie graphique appelé schématique.

- Les langages de niveau logique ont été les seuls à être utilisés comme point d'entrée des premiers systèmes de génération automatique de C.I., tels que les systèmes utilisant des cellules pré-caractérisées ou des réseaux prédiffusés [TUC85].

- Ce type de langages se prête bien aux méthodes formelles de synthèse et de vérification d'où des travaux plus récents concernant la synthèse logique [DEG86], [BRA87], [RUD87].

• Niveau transfert de registres : les langages de ce niveau manipulent des éléments de mémorisation et des opérateurs [BAR75]. La littérature est riche d'exemples de langages de niveau transfert de registres, [BAR75] donne une étude comparative de plusieurs de ces langages. On peut trouver aussi des références plus récentes dans [MAR86] et [CRA85]. Ce niveau est parfois désigné sous le nom de niveau micro-architecture [GAJ87]. Les langages de ce niveau permettent en général de mélanger des descriptions comportementales (essentiellement du contrôle) et des descriptions structurelles (essentiellement des chemins de données). Ceci correspond à une étape de la conception où l'on a fixé la structure d'une partie du circuit (par exemple la partie opérative) et où la partie contrôle est décrite sous forme d'un "programme" qui active la partie opérative. MARINE [MAR86] appelle ce genre de spécifications la description mixte (voir aussi III.3).

• Niveau programme : les objets manipulés à ce niveau sont aussi des éléments de mémorisation et des opérateurs. Mais, contrairement au niveau transfert de registres, les éléments de mémorisation et les opérateurs utilisés par une description ne sont pas liés à des réalisations physiques. Une description de niveau programme définit des séquences d'opérations manipulant des structures de données [GAJ87]. La séparation entre le niveau programme et le niveau transfert de registres n'est pas très nette. La plupart des langages de haut niveau pour la description du matériel permettent de décrire à la fois des réalisations au niveau transfert de registres et au niveau programme. Ce niveau de description sera plus détaillé dans le reste de cette thèse. En fait, le langage de description de SYCO se situe au niveau

programme.

• Niveau système : le niveau système est utilisé pour spécifier des systèmes entiers, comprenant des parties logicielles et des parties matérielles. Le but d'une telle spécification est en général de fixer les performances et le coût du système. Les éléments manipulés à ce niveau sont des sous-systèmes (processus, mémoire...). Ce niveau peut être vu comme une description complexe du niveau programme.

II.1.3.2 Domaines de description

La notion de domaines de description découle des travaux sur la compilation de silicium [GAJ82]. Les premières recherches dans ce domaine ont tenté de concevoir une hiérarchie d'outils de synthèse qui mime la hiérarchie des niveaux d'abstraction. Avec un tel modèle, un compilateur de silicium est un ensemble de traducteurs permettant de passer d'un niveau à un autre dans la hiérarchie. Ce passage est généralement réalisé par des techniques de macro-génération. En fait, ce procédé est trop rigide car certains des niveaux d'abstraction présentés ci-dessus sont plus complémentaires que hiérarchiques. Par exemple, les niveaux topologique et électrique manipulent une réalisation de circuit au même niveau d'abstraction, mais chacun considère des aspects différents du circuit.

La notion de domaines d'abstraction permet de considérer plusieurs aspects d'un circuit à chaque niveau d'abstraction. La figure II.2 retrace le diagramme de GAJSKI (dit aussi diagramme en "Y"), souvent utilisé pour représenter la notion de domaines de description. Ce diagramme représente trois domaines de description : le domaine comportemental, le domaine structurel et le domaine physique. La fonction du circuit appartient au domaine comportemental, la réalisation du circuit au domaine physique, et le domaine structurel est un pont, entre les deux autres domaines, qui décrit une réalisation comme étant une hiérarchie de blocs interconnectés. Dans chaque domaine, on trouve plusieurs niveaux d'abstractions.

La description complète d'un circuit spécifie la réalisation dans les trois domaines. Cette description peut être représentée par un cercle dont le centre est celui du diagramme en "Y" [WAL85]. Le rayon du cercle définit le niveau d'abstraction de la description. En traçant tous les cercles possibles, on obtient les différents niveaux d'abstraction. Dans le cas de la figure II.2, on obtient cinq niveaux.

Partant du diagramme de la figure II.2, il ressort clairement que le compilateur de silicium idéal peut être défini comme étant un système capable de transformer une description du niveau du cercle le plus grand (niveau système dans la figure II.2) en une description du niveau du cercle le plus petit (niveau circuit dans la figure II.2). Les auteurs de [GAJ87] et [WAL85] donnent plus de détails sur l'interprétation du diagramme en "Y". Ils donnent aussi

une présentation plus détaillée des niveaux d'abstraction en précisant les objets manipulés pour chaque niveau et pour chaque domaine de description.

Pour être complète, la spécification d'un circuit doit couvrir les trois domaines de description. Une telle description nécessite en général plusieurs représentations et chacune d'elles peut contenir des informations d'un ou plusieurs domaines de description. Une représentation peut être externe (description textuelle ou graphique lisible par l'utilisateur) ou interne (structure de données).

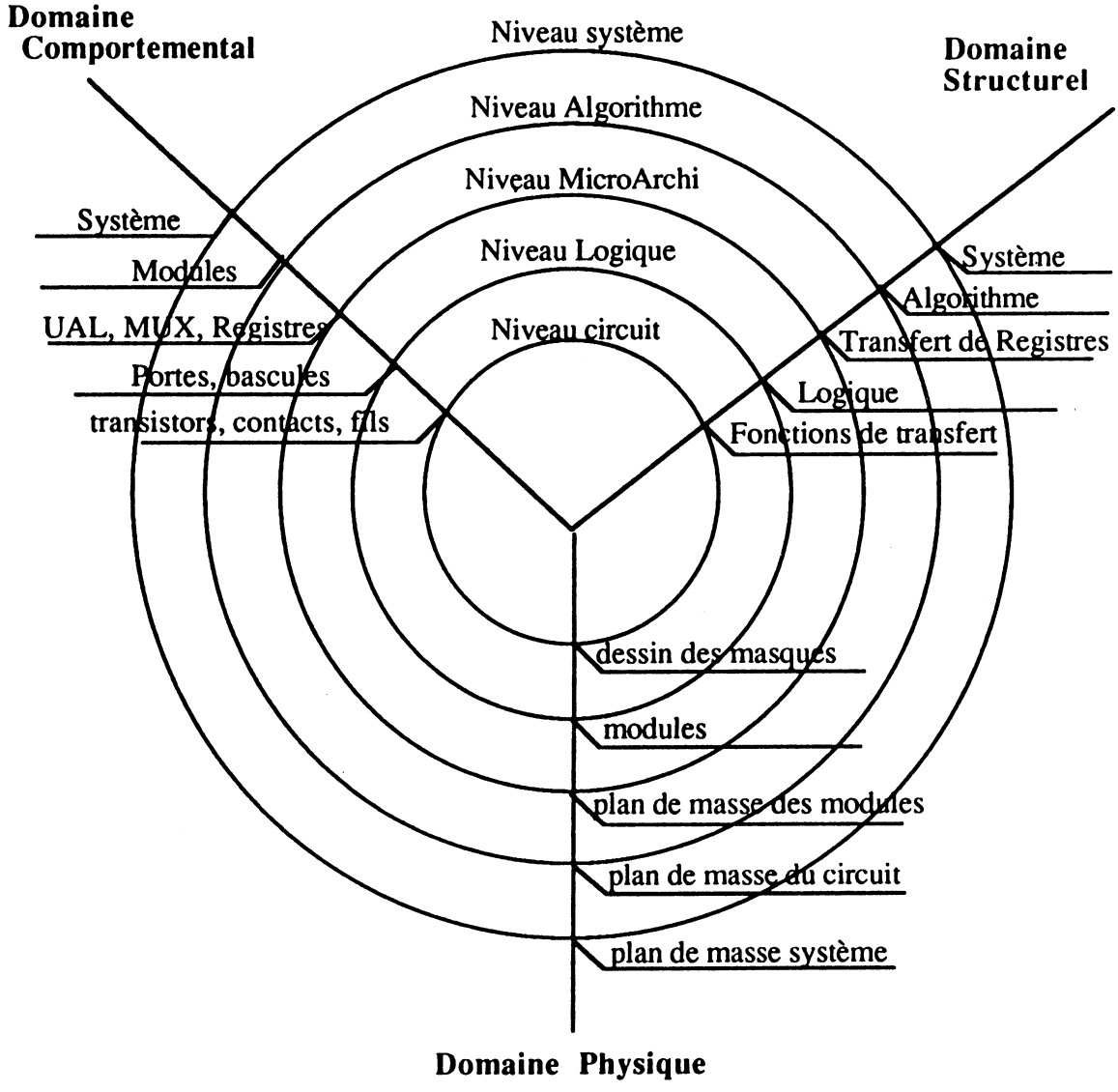


Figure II.2 Niveaux d'abstraction et domaines de description [WAL85]

La vérification de la cohérence entre les différentes représentations constitue l'élément central des processus manuels de conception. Cette vérification est en général compliquée par la présence de représentations hiérarchiques, car l'organisation hiérarchique d'un objet peut être différente dans des représentations différentes [KNA85].

II.1.4 Outils de CAO et méthodologies de conception

Chaque méthode de conception définit un ensemble d'outils permettant d'automatiser certaines parties ou la totalité du processus de conception. Malheureusement, en pratique, c'est l'inverse qui se passe : on est souvent amené à définir une méthodologie de travail en fonction des outils de CAO existants. Ceci vient du fait qu'on ne dispose pas toujours des moyens nécessaires à la réalisation d'outils "sur mesure".

Chacun de ces outils peut être concerné par plusieurs niveaux d'abstraction et/ou plusieurs domaines de description. La représentation interne du circuit, utilisée par chacun des outils peut correspondre à un ou plusieurs langages de description. Les termes représentation et langage de description seront utilisés de manière non différenciée dans la suite.

II.1.4.1 Outils de CAO

Un système de CAO peut contenir des outils pour aider à :

- créer et modifier des représentations,
- archiver et gérer les différentes représentations,
- transformer des représentations,
- vérifier des représentations,
- tester un circuit.

• Les outils de création et modification : vu le nombre de représentations possibles d'un circuit intégré, on peut avoir besoin de plusieurs outils de ce type pour couvrir tous les niveaux d'abstraction et tous les domaines de description. La création et la modification de descriptions nécessitent des éditeurs et des langages de description qui peuvent être graphiques ou textuels. Un langage de description peut couvrir plusieurs aspects d'un C.I. Par exemple, un langage symbolique permet de décrire à la fois le dessin des masques, le schéma logique et le réseau électrique. Mais la couverture de tous les niveaux d'abstraction et domaines de description utilisés par le processus de conception nécessite plusieurs langages de description et éventuellement plusieurs éditeurs.

• Les outils d'archivage et de gestion : il est préférable que l'archivage et la gestion des différentes représentations soient réalisés par un système unique. La fonction d'archivage peut être réalisée par des outils classiques de gestion de fichier. La gestion des représentations doit assurer la cohérence de toutes les informations concernant le circuit. Elle nécessite des outils spécifiques car même les systèmes de gestion de bases de données se sont avérés insuffisants pour les applications de CAO [KAT82], [BON89]. Etant donnée la quantité d'informations

contenue dans une description, ces outils sont cruciaux pour les premiers niveaux d'abstraction.

• Les outils de transformation : cette classe d'outils regroupe les outils d'abstraction, les outils d'optimisation et les outils de compilation. Ils permettent de générer une nouvelle représentation d'un objet en partant d'une représentation initiale.

Les outils d'abstraction sont en général utilisés pour la vérification. Ils sont souvent appelés extracteurs. Il permettent d'extraire une description d'un objet en partant d'une représentation interne de cet objet. Les deux représentations peuvent appartenir au même niveau d'abstraction et représenter des aspects différents d'un même objet. Par exemple, on peut extraire un schéma électrique en partant du dessin des masques [JER83].

L'optimisation d'un C.I. peut se faire à tous les niveaux d'abstraction. Plus le niveau d'abstraction est élevé, plus les effets des actions d'optimisation sont significatifs. Ceci vient du fait que les primitives utilisées dans une représentation de haut niveau, en général, concernent une partie plus grande du circuit que les primitives de plus bas niveau. Un outil d'optimisation peut modifier un ou plusieurs aspects d'un circuit. Ceci vient du fait que l'amélioration des performances d'un circuit dans un domaine donné peut entraîner une dégradation des performances du circuit dans les autres domaines. Par exemple, l'augmentation de la vitesse d'un circuit peut entraîner une augmentation de sa consommation et/ou de sa surface. Dans le cas où le processus d'optimisation peut modifier plusieurs aspects du circuit, il s'agit plutôt de rechercher un bon compromis et non d'optimiser. Pour mesurer les effets des actions d'optimisation, on utilise généralement des outils d'évaluation.

Les outils de compilation transforment une représentation appartenant à un niveau d'abstraction donné en une représentation d'un niveau inférieur ou égal, et peuvent mettre en œuvre des outils d'optimisation. L'étude des différents outils de compilation sera détaillée dans la suite de ce chapitre.

• Les outils de vérification : il existe deux types de vérification. La vérification d'une spécification initiale permet d'assurer qu'elle correspond à ce qu'on veut réaliser, tandis que la vérification d'une spécification intermédiaire (résultat d'une transformation de la spécification initiale) permet d'assurer le maintien de la cohérence entre les spécifications. La simulation a été longtemps le seul moyen de vérification. Récemment, des techniques de vérifications formelles commencent à être utilisées [STA88]. Elles consistent à prouver formellement que deux représentations différentes d'un même objet sont équivalentes. La simulation consiste à prouver qu'une représentation correspond bien à un objet produisant des séquences de vecteurs de sortie prédéfinies en présence de séquences de vecteurs d'entrée données. Les résultats de la simulation restent partiels tant que l'on n'a pas simulé le circuit de façon exhaustive.

• Les outils de test et de diagnostic : ils sont utiles après la phase de fabrication du circuit. Le test de fin de fabrication permet d'identifier les circuits utilisables à la sortie du processus de fabrication, tandis que les tests périodiques visent à détecter les pannes au cours de la vie du circuit (test), et éventuellement à les localiser (diagnostic). Des méthodes de conception adaptée au test ("DFT : Design For Testability") peuvent être utilisées pour prévoir des dispositifs particuliers facilitant le test.

Les méthodes de test se divisent en deux catégories. Le test en-ligne ("on-line testing"), qui s'effectue pendant le fonctionnement normal du circuit, et qui utilise comme vecteurs de test les données d'entrée du circuit, et le test hors-ligne ("off-line testing"), qui s'effectue en dehors du fonctionnement normal du circuit, et qui utilise des vecteurs de test spécifiques, nécessitant des méthodes de génération automatique ("ATPG : Automatic Test Patterns Generation") extrêmement coûteuses, et n'offrant pas toujours une couverture de pannes ("fault coverage") suffisante.

On peut également opérer une distinction des méthodes de test en test externe ("external testing"), nécessitant un équipement de test particulier, et le test intégré ("built-in testing") pour lequel le dispositif de test est prévu et intégré au circuit à sa conception.
Bien entendu, le test en-ligne est un test intégré, alors que le test hors-ligne peut être externe ou intégré.

Les méthodes de test en-ligne sont généralement basées sur la redondance d'informations (utilisation de codes détecteurs et/ou correcteurs d'erreurs) ou de matériels (utilisation de voteurs). Ces techniques entraînent une augmentation de la surface du circuit pouvant atteindre 50% [NIC84], mais permettent en revanche une très bonne couverture de pannes.

II.1.4.2 Organisation des outils de CAO et méthodologies de conception

Le processus de conception d'un circuit intégré peut mettre en œuvre plusieurs outils travaillant sur plusieurs niveaux d'abstraction. A part les outils de test, tous les types d'outils cités plus haut peuvent exister à plusieurs niveaux. Un système de CAO complet doit donc permettre la représentation de tous les niveaux d'abstraction et domaines de description. Un tel système comporte des outils à chaque niveau d'abstraction.

La figure II.3 montre un système de CAO générique travaillant sur plusieurs niveaux d'abstraction. Le système est organisé autour d'une pile de représentations. Chaque étage de cette pile correspond à un niveau d'abstraction et contient les différents aspects du circuit. Des outils de création, de validation et d'optimisation de la description doivent aussi exister à chaque niveau. Dans ce schéma, toutes les représentations sont accessibles au concepteur. Les outils

d'extraction et les outils de vérification permettent de valider les niveaux inférieurs d'abstractions (autres que le niveau système). Les outils de compilation permettent un passage rapide et sans erreur entre les étages. Les outils de compilation et d'extraction permettent de passer d'un niveau à l'autre. Les outils de vérification permettent de vérifier la cohérence entre les différents niveaux.

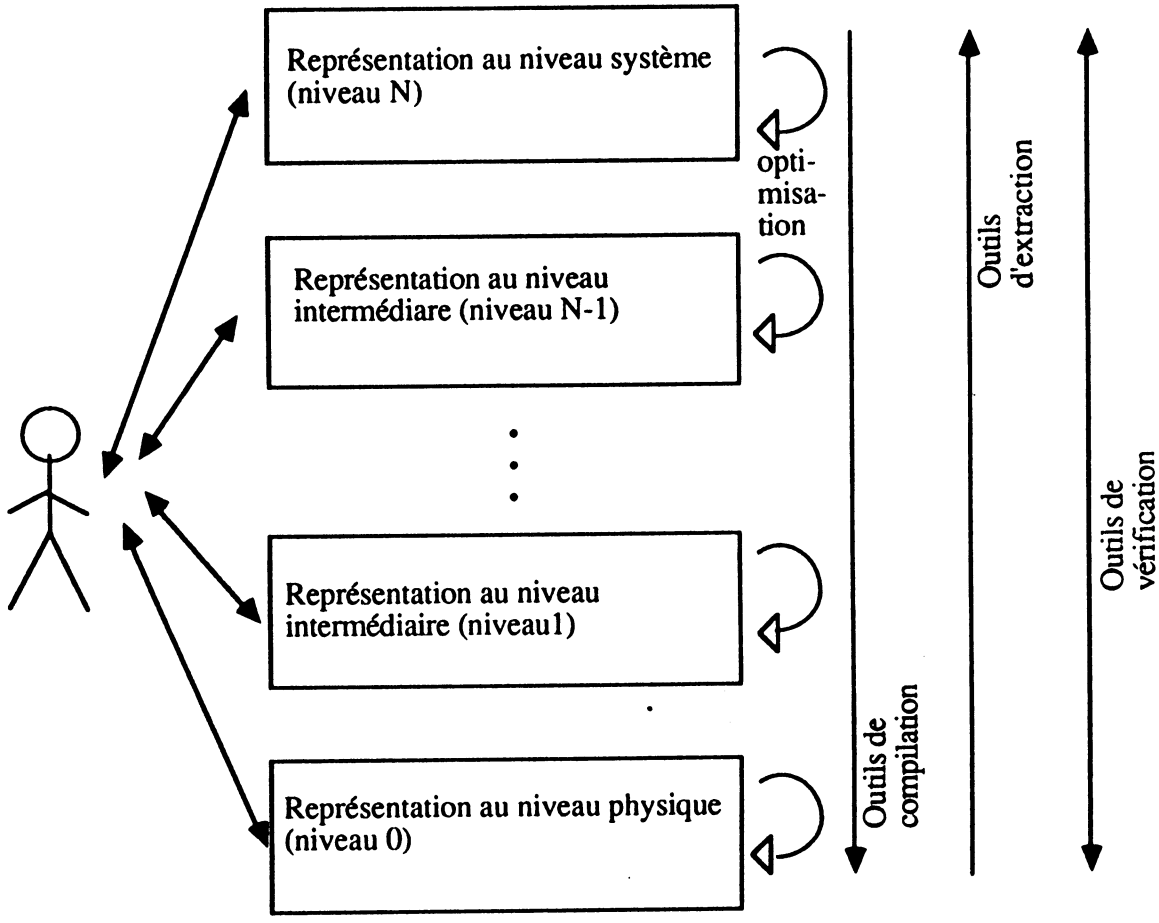


Figure II.3 Environnement complet pour l'aide à la conception de circuits intégrés

Le passage d'un niveau d'abstraction à un niveau inférieur peut se faire selon trois schémas qui définissent les organisations possibles d'un compilateur de silicium :

- Passage manuel : les outils du niveau inférieur sont utilisés comme un environnement pour créer la nouvelle description. Le système n'a aucun contrôle sur les décisions de l'utilisateur. Il faut alors des outils pour valider la nouvelle description et prouver qu'elle correspond à la description initiale.

- Passage assisté : la transformation est réalisée par l'utilisateur sous le contrôle du système. Le système, appelé assistant, vérifie donc la validité des actions du concepteur.
- Passage automatique : ce passage se fait à l'aide d'un compilateur qui produit la

représentation du niveau inférieur en partant de la représentation initiale. Dans ce cas, l'utilisateur n'a aucun contrôle sur le déroulement du processus de compilation lui-même. Si le compilateur est bien construit, la nouvelle description créée doit être cohérente avec la description initiale.

Ces trois schémas constituent trois cheminements possibles pour organiser un compilateur de silicium. Ils sont étudiés plus loin et nommés respectivement : environnement pour la compilation de silicium, assistant pour la compilation de silicium et compilateur de silicium.

Plusieurs méthodologies existantes sont basées sur des schémas pouvant dériver de celui donné par la figure II.3. Par exemple, le système CATHEDRAL II utilise deux niveaux d'abstraction : le niveau système et le niveau physique. La méthodologie de conception utilisée est appelée "rencontre au milieu" ("meet in the middle") car elle met en œuvre deux types de concepteurs : les concepteurs systèmes et les concepteurs circuits [DeM87]. Chaque type de concepteurs intervient à un seul niveau de la hiérarchie.

II.2 Compilateurs de silicium et processus de conception

Le schéma du compilateur idéal tel qu'il a été défini précédemment (II.1.3.2) part d'une représentation système et utilise tous les niveaux d'abstraction comme formes intermédiaires. En pratique, le passage par tous les niveaux d'abstraction n'est pas toujours utile.

Comme il a été dit plus haut, un compilateur de silicium est un système permettant d'automatiser une partie voire la totalité du processus de conception dans le cadre d'une méthodologie donnée. Il faut donc définir la représentation de départ qui sera le langage d'entrée du compilateur, puis cerner les différentes étapes du processus de conception, ce qui va permettre de caractériser les étapes du compilateur et les différentes représentations intermédiaires utilisées par le processus de compilation.

Pour la compilation de silicium, il faut distinguer quatre niveaux de représentations :

- La représentation de l'architecture externe, également nommée description comportementale.
- La représentation de l'architecture interne, indépendante de la technologie.
- La représentation de la structure, définie pour une technologie donnée,
- La représentation physique, définie pour un processus de fabrication donné.

Ces représentations correspondent à des états d'avancement du système à concevoir. Les choix et les délimitations de ces représentations sont donc fait en fonction des étapes importantes du processus de conception. Les quatre niveaux de représentations choisis correspondent aux étapes d'enrichissement de la description initiale par de nouvelles informations. Le processus de compilation, ou de conception, commence avec une description comportementale. La description architecturale nécessite l'introduction du découpage du circuits en sous-systèmes. La description structurelle introduit les informations technologiques. Finalement, la représentation physique introduit les informations concernant le processus de fabrication.

Cette hiérarchie, utilisant quatre niveaux de représentations, a été définie pour la première fois dans le cadre du projet CMU-DA [THO83], [KOW85]. Néanmoins, la présentation qui suit introduit quelques différences dans la définition des diverses représentations. Pour le projet CMU-DA [THO83], chaque représentation correspond à un seul langage de description. Dans ce qui suit, chaque représentation peut décrire plusieurs aspects du système et donc peut correspondre à plusieurs langages de description.

II.2.1 Représentations utilisées pour la compilation de silicium

Pour la compilation de silicium, il n'est pas nécessaire d'avoir une représentation complète du circuit à chaque niveau. Pour qu'une représentation soit complète, au sens de la classification des niveaux de description et des domaines de spécification définis par le diagramme de GAJSKI, elle doit spécifier le circuit dans les trois domaines de description.

I.2.1.1 Représentation de l'architecture externe : la description comportementale

La description comportementale d'un circuit décrit son architecture externe, c'est-à-dire la manière dont il peut être utilisé [ANC86]. Dans le cas d'un microprocesseur, par exemple, cette description donne le jeu d'instructions exécutées par le circuit et les facilités (registres, opérateurs...) [ANC86].

Dans le cas général, l'architecture externe d'un circuit décrit:

- L'interface du circuit : entrées/sorties accessibles à l'utilisateur.
- La fonction réalisée par le circuit. Elle est donnée sous une forme indépendante de l'organisation interne de la machine. Elle peut être exprimée à l'aide d'un langage fonctionnel ou d'un langage procédural. Les avantages et les inconvénients de chaque type de langages seront analysés plus loin en III.3.

L'interface et la fonction décrivent le comportement du système. Cette représentation peut contenir quelques informations structurelles ou topologiques. Ces informations peuvent être des indications, tel que l'utilisation d'une architecture cible, pour simplifier et/ou guider le processus de conception. Elles peuvent aussi spécifier des contraintes qui doivent être respectées par le processus de compilation (ou de conception) : on peut par exemple fixer une durée maximum du temps de cycle et/ou une surface maximum.

II.2.1.2 Représentation de l'architecture interne

Cette représentation décrit l'organisation globale du circuit. A ce niveau, un circuit est composé d'un ensemble de sous-systèmes interconnectés. Un sous-système peut être un module standard (contrôleur, partie opérative, opérateur complexe...) ou un système complexe. La représentation d'un sous-système peut être donnée sous une forme comportementale. Elle sera, par exemple, décrite dans le langage de description utilisé par le niveau précédent, ou un langage spécialisé dans le cas d'un module standard.

Cette représentation est indépendante de la technologie cible (MOS, Bipolaire...). Comme dans le cas de la représentation du comportement, ce niveau peut contenir des informations pour simplifier le reste du processus de compilation et des contraintes qui doivent être respectées. Toutes les contraintes exprimées au niveau comportemental et qui concernent la

structure, telles que le temps de cycle, doivent être conservées à ce niveau.

II.2.1.3 Représentation de la structure

Cette représentation décrit la structure détaillée du circuit. Elle correspond au niveau des descriptions logique et transfert de registres. Un circuit est représenté par un ensemble de blocs interconnectés. Un bloc peut-être un module standard (mémoire, registre...) ou constitué par un ensemble de blocs interconnectés. Cette représentation donne tous les détails concernant la réalisation des signaux d'interconnexion entre les blocs. Le découpage en modules standards tient compte d'une technologie cible.

Les informations topologiques peuvent exprimer un plan de masse détaillé du circuit, ou une série de contraintes à respecter pour l'étape de génération de dessin des masques. Les contraintes exprimées par les niveaux supérieurs doivent aussi être répercutées au niveau physique.

II.2.1.4 Représentation physique

Cette représentation contient tous les détails de la réalisation physique, y compris sur les dessins des masques. Elle dépend d'un processus de fabrication donné.

La représentation physique couvre généralement plusieurs aspects du circuit. Elle doit respecter toutes les contraintes énoncées par les représentations du comportement, de l'architecture et de la structure.

II.2.2 Les compilateurs de silicium

Les quatre niveaux de description décrits plus haut définissent trois classes de compilateurs. Chaque niveau de représentation, autre que le niveau physique, définit une classe de compilateurs couvrant une étape du processus de conception.

La figure II.4 donne un schéma de compilation complet utilisant les quatre niveaux des représentations définis plus haut. Ce schéma comporte les trois étapes suivantes :

- La compilation de comportement, qui produit l'architecture à partir de la représentation du comportement,
- La compilation d'architecture, qui produit la structure à partir de la représentation de l'architecture,
- La compilation de structure, qui produit une représentation physique en partant de la structure.

En pratique, certains compilateurs existants permettent de réaliser plusieurs étapes de compilation. Pour des raisons de clarté, on considérera dans la suite trois classes distinctes de compilateurs de silicium.

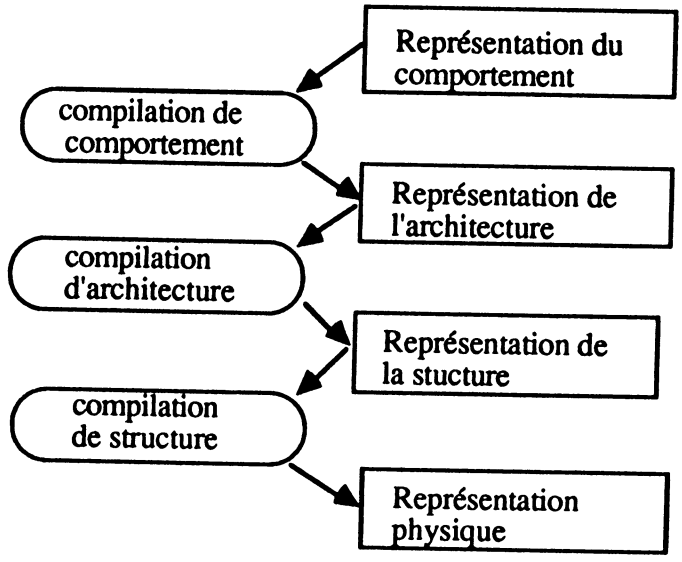


Figure II.4 Compilateurs de silicium et processus de conception

II.2.2.1 Les compilateurs de comportement

Un compilateur de comportement produit l'architecture d'un circuit en partant de sa description comportementale.

Dans une description de comportement, un circuit est décrit par ses entrées, ses sorties et sa fonction. La description de la fonction est indépendante de la réalisation future du circuit. La génération de l'architecture implique un découpage de la description comportementale en sous-systèmes. Cette opération est appelée partition (voir III.5). Elle doit résoudre deux difficultés

- Trouver un protocole de communication efficace entre les différents sous-systèmes générés.
- Trouver un moyen rapide pour évaluer les performances d'une éventuelle réalisation correspondant à un découpage donné.

Apporter une solution générale à ces deux problèmes est une tâche complexe, pour simplifier, les compilateurs de comportement utilisent fréquemment une architecture cible. Ainsi, ces compilateurs se trouvent limités à une classe réduite d'applications.

Un compilateur de comportement ne nécessite pas d'utilisateurs experts dans le domaine de la conception des circuits intégrés. Par contre, dans le cas général, il n'est pas simple de

générer des circuits efficaces avec ce genre de compilateur. Ceci vient du fait qu'il est très difficile d'optimiser certains aspects du circuit tels que la surface ou la consommation en agissant sur la description du comportement. L'optimisation, dans le cas d'un compilateur de comportement, consiste à améliorer le découpage. L'utilisation de modèles prédéfinis pour les compilations de l'architecture et de la structure permet de réduire l'importance de ce problème.

Le compilateur SYCO part d'une description du comportement et utilise une architecture cible permettant de générer des circuits ayant une partie contrôle hiérarchique et une partie opérative parallèle. L'utilisation de modèles prédéfinis pour la synchronisation et pour l'organisation du dessin des masques permet de développer des outils d'évaluation et d'optimisation efficaces. Les modèles utilisés par le compilateur SYCO sont détaillés dans le quatrième chapitre.

II.2.2.2 Les compilateurs d'architecture

Un compilateur d'architecture transpose une description de l'architecture en fonction d'une technologie donnée. Ce type de compilateurs ne réalise pas de découpage.

Au départ, le circuit est décrit par un ensemble de sous-systèmes interconnectés. Chaque bloc peut être décrit par un langage spécialisé, adapté au type du bloc. La compilation des sous-systèmes est généralement réalisée par des compilateurs spécialisés (compilateurs de parties opératives, compilateurs de contrôleurs, compilateurs d'opérateurs ...). Ces compilateurs spécialisés fonctionnent comme des modules générateurs (programmes pour générer des structures particulières). Ils fournissent des informations sur les caractéristiques des blocs générés. Ces compilateurs sont souvent associés à des bibliothèques de cellules (jeu de plots, amplificateurs, portes...) pour réaliser des blocs particuliers, ou pour interfacer le circuit avec l'extérieur.

Un compilateur d'architecture doit assurer la synchronisation entre les différents sous-systèmes. Il peut être amené à utiliser des modèles prédéfinis pour faciliter le processus de compilation. Comme il a été dit plus haut, l'usage de modèles prédéfinis limite l'espace des solutions possibles. L'utilisation d'un modèle d'horloge prédéfini, par exemple, peut entraîner des pertes de performances.

Ce type de compilateur peut être utilisé par des ingénieurs peu familiarisés avec les détails de la conception des circuits intégrés. Il est évident que plus le concepteur est expérimenté plus les circuits réalisés sont performants.

II.2.2.3 Les compilateurs de structure

Un compilateur de structure transpose une représentation structurelle en une représentation physique en tenant compte d'un processus de fabrication donné. Selon le style de conception adopté par le compilateur, il peut avoir besoin de :

- une bibliothèque de cellules prédéfinies,
- une bibliothèque de modules générateurs,
- une bibliothèque de fonctions ou de commandes pour la construction de la représentation physique (placement, routage...).

L'interaction d'un compilateur de structure avec l'utilisateur peut se faire par l'intermédiaire d'une interface graphique ou textuelle. Dans le dernier cas le compilateur est généralement organisé autour d'un langage de description procédurale des dessins des masques. La conception d'un circuit revient alors à écrire un programme qui, en s'exécutant, génère le circuit. Ce programme peut faire appel à bibliothèque de cellules et/ou à des programmes générateurs de structures particulières. Il est souvent appelé module générateur [NEW87].

Le placement des cellules peut être donné par la description structurelle ou réalisé automatiquement. Dans ce dernier cas, le compilateur doit déterminer le plan de masse du circuit, qui peut être fixe. La plupart des compilateurs de parties opératives, par exemple, utilisent un des modèles topologiques prédéfinis pour assembler le circuit [JAM85], [SHR85].

Un compilateur de structure peut être conçu pour permettre la réalisation de circuits aussi performants que des circuits générés manuellement [ROU87].

L'utilisation de ce type de compilateur nécessite d'être expert en conception de circuits intégrés. Dans le cas où l'interface est une description procédurale, le concepteur doit aussi être un expert en programmation. Ainsi, les pertes de performances dues à l'utilisation d'un tel système sont très liées à l'expérience des utilisateurs.

II.3 Organisation des compilateurs de silicium

Indépendamment du niveau de représentation considéré, comportement, architecture ou structure, il existe trois schémas pour passer d'un niveau donné au niveau inférieur. Comme il a été dit plus haut (II.1.4.2), chacun de ces schémas définit une classe de compilateurs de silicium. Les trois types de compilateurs sont :

- Les compilateurs simples (désignés par compilateur tout court),
- Les assistants pour la compilation,
- Les environnements de compilation.

Indépendamment de son organisation, chaque compilateur utilise une bibliothèque de cellules ou de modules prédéfinis et une bibliothèque d'outils. Le processus de compilation consiste à enchaîner des outils de la bibliothèque d'outils et à choisir des éléments dans la bibliothèque de modules. Chaque type d'organisation définit des modalités pour partager le contrôle du processus de conception entre l'utilisateur et le compilateur.

II.3.1 Les compilateurs

Comme il a été défini plus haut, un compilateur est un outil capable de transformer une représentation de départ en une autre représentation de niveau inférieur. Un tel processus peut être modélisé par la figure II.5. Il utilise une bibliothèque de sous-circuits et une bibliothèque d'outils prédéfinis. La nature des éléments des bibliothèques dépend du niveau d'abstraction traité par le compilateur.

Avec un tel système, l'utilisateur n'a aucun contrôle sur le processus de compilation. Le résultat de la compilation est déterminé uniquement par la description d'entrée. Les règles utilisées pour le choix des modules et les règles d'enchaînement des outils sont prédéfinies dans le compilateur.

Ce type de compilateur est simple à utiliser. L'utilisateur n'est pas obligé de connaître tous les détails de fonctionnement du compilateur ni les détails du résultat pour pouvoir l'utiliser, même s'il est vrai qu'une telle connaissance permet, en général, une utilisation plus efficace du compilateur. La réalisation de ce genre d'outils nécessite, pour des raisons d'efficacité, l'utilisation de modèles fixes, ce qui limite le domaine d'application du compilateur. D'autre part, l'ensemble des outils de la bibliothèque utilisables par le compilateur est généralement fixe, et il est généralement difficile d'ajouter un nouvel outil au compilateur.

En pratique, ce mode d'organisation est surtout utilisé pour les compilateurs de comportement. Ceci vient essentiellement du fait qu'il y a une grande ressemblance entre les

langages de programmation classiques et les langages d'entrée de ces compilateurs. Par ailleurs, l'utilisation de modèles prédéfinis permet l'utilisation de plusieurs techniques classiques de compilation [AHO86].

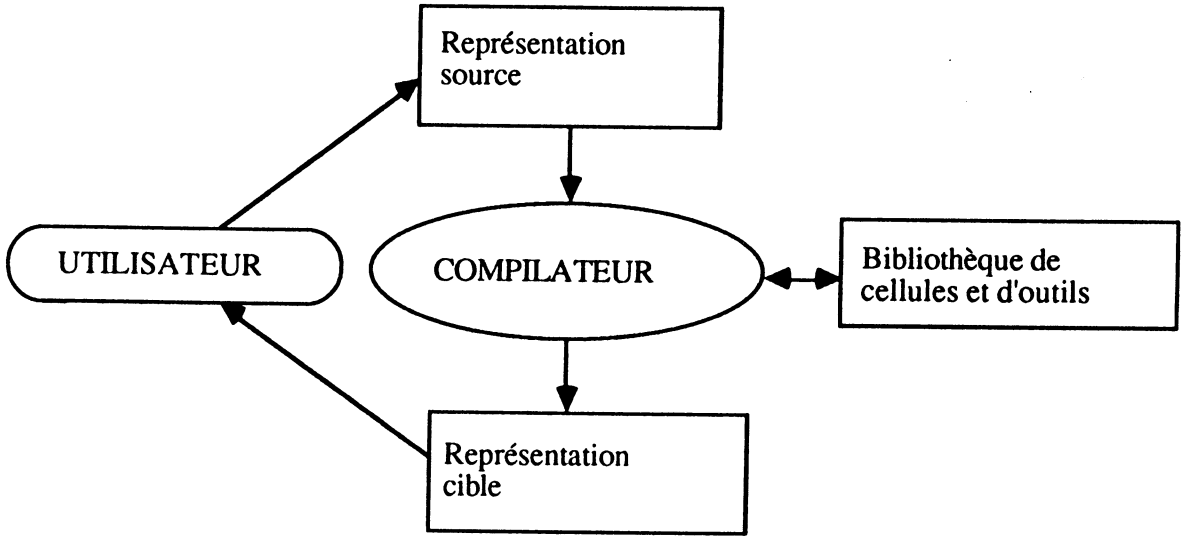


Fig II.5 Organisation d'un compilateur

II.3.2 Les assistants pour la compilation

Un assistant est un système dans lequel le processus de compilation est contrôlé en partie par l'utilisateur. Un assistant (figure II.6) utilise une base de données où sont archivés les résultats des différentes étapes du processus de compilation déjà effectuées, une bibliothèque de cellules et d'outils, et un ensemble de règles pour l'enchaînement des outils et le choix des cellules de la bibliothèque. Dans le cas où l'ensemble des règles est extensible, le système est dit "intelligent".

Dans le cas d'un assistant, l'utilisateur peut avoir accès au contrôle de l'enchaînement des outils et au choix des éléments de la bibliothèque à travers l'assistant. Avec un tel système, le processus de compilation comporte des itérations et des retours arrière. Les décisions peuvent être prises par l'utilisateur, ou par l'assistant. Dans ce dernier cas, il faut que l'assistant dispose d'outils d'évaluation.

L'assistant doit être capable de "comprendre" l'évolution du processus de compilation pour pouvoir critiquer les décisions prises par l'utilisateur. Pour cela il faut que l'assistant ait une connaissance bien précise des outils et des éléments de la bibliothèque mis en jeu. La nature de ces outils et des éléments de la bibliothèque dépend du niveau de la représentation d'entrée du compilateur.

Les assistants sont d'une utilisation souple, et permettent d'essayer plusieurs solutions pour en choisir la meilleure. Le degré de souplesse augmente avec la taille de l'ensemble des

règles dont dispose l'assistant. Paradoxalement, la définition de la base de règles nécessite de limiter le domaine des connaissances et de limiter les objets et les outils manipulés. Ceci réduit la souplesse de l'assistant et conduit généralement à une spécialisation de l'assistant pour une classe restreinte de circuits ou de sous-circuits.

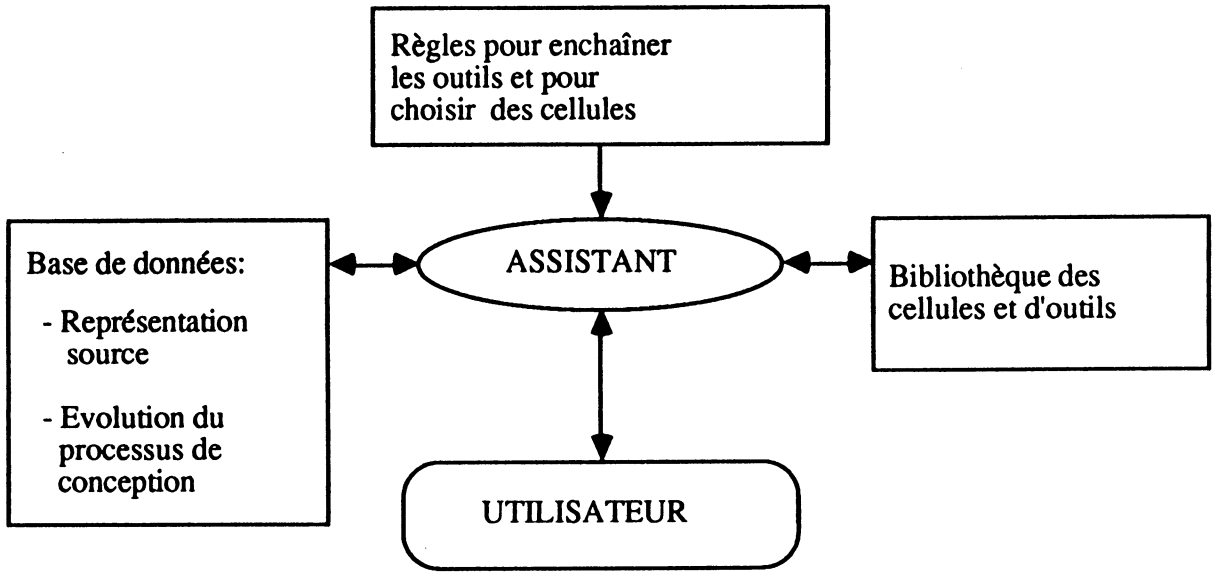


Fig II.6 Organisation d'un assistant pour la compilation

II.3.3 Les environnements pour la compilation de silicium

Un environnement de compilation est composé d'une base de données, qui contient l'état d'avancement du processus de conception, et d'un ensemble de bibliothèques d'outils et de cellules, tous deux accessibles à travers un gestionnaire (figure II.7). Un tel système offre des facilités de conception sans pour autant contrôler le processus de compilation.

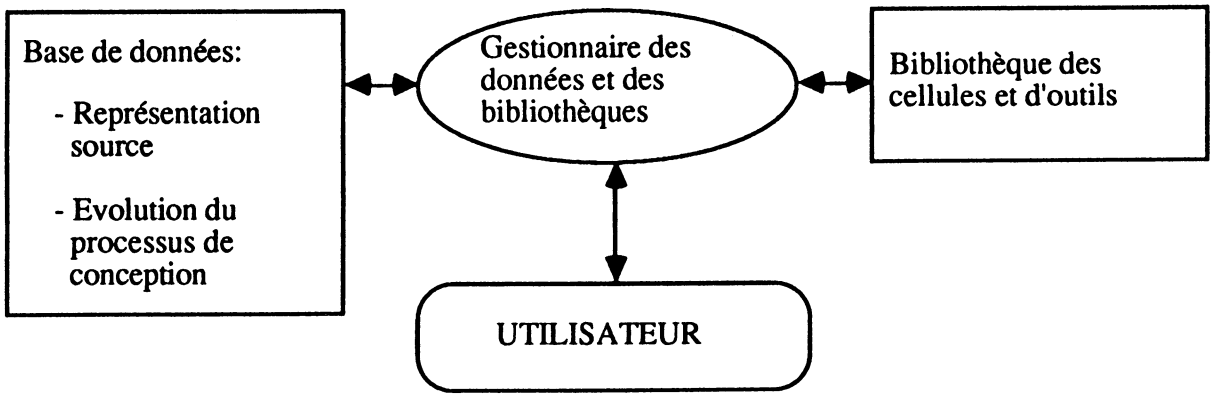


Fig II.7 Organisation d'un environnement de compilation

Dans le cas d'un environnement pour la compilation de silicium, c'est l'utilisateur qui contrôle complètement le processus de compilation. Avec un tel système, l'utilisateur est libre de choisir l'enchaînement des outils et de sélectionner les cellules de la bibliothèque. Par contre

il est difficile de contrôler la validité des décisions prises par l'utilisateur et, par conséquent la validité du résultat du processus de conception.

L'utilisation d'un environnement de compilation nécessite une bonne connaissance du système lui-même et des concepts manipulés, ce qui explique le succès de ce type d'organisation pour la manipulation des représentations structurelles et physiques. En fait les concepteurs expérimentés refusent les pertes de performances dues à l'utilisation d'outils de compilation de haut niveau. Avec un environnement de compilation de structure, le concepteur peut contrôler tout le processus de conception.

II.4 Compilation de silicium : état de l'art

Bien que récent, le domaine de la compilation de silicium a déjà fait l'objet d'une littérature abondante, qui reflète un grand besoin en outils d'aide à la conception.

La littérature contient aussi plusieurs essais récents de classification des compilateurs de silicium. GAJSKI, dans [GAJ86], considère quatre critères pour classer les compilateurs de silicium : modèle architectural utilisé, organisation des dessins des masques, synchronisation et langage d'entrée du compilateur. Ces critères mènent à une classification des compilateurs selon la complexité des circuits traités. Le même auteur, dans [GAJ87], distingue quatre types de compilateurs : les compilateurs de cellules, les compilateurs de blocs, les compilateurs de processeurs et les compilateurs de systèmes. DENYER, dans [DEN88], donne une autre classification des compilateurs de silicium, distinguant les générateurs ou assembleurs de silicium, les compilateurs utilisant une architecture fixe et les compilateurs intelligents. Ces derniers sont capables de choisir l'architecture en fonction de l'application.

Ces deux essais de classification ne tiennent compte ni des étapes de la conception que le compilateur est supposé réaliser, ni de la façon dont ces étapes s'intègrent dans une méthodologie de conception donnée. Cette section donne une classification des compilateurs de silicium qui tient compte à la fois des niveaux d'abstraction qu'ils manipulent et de leur organisation interne.

II.4.1 Les outils pour la compilation de comportement

La compilation de comportement consiste à produire l'architecture d'un circuit en partant de sa description comportementale. La plupart des outils de cette catégorie sont du type compilateur. Ceci fait qu'ils sont généralement spécialisés pour une classe restreinte d'applications. Les techniques de compilation de comportement feront l'objet du troisième chapitre.

II.4.1.1 Les compilateurs de comportement

La plupart de ces systèmes génèrent des circuits composés d'une partie opérative et d'une partie contrôle. Certains de ces compilateurs sont orientés vers la génération de circuits de communication. Dans ce dernier cas, le processus de compilation avantage le parallélisme dans la partie opérative par rapport à la partie contrôle.

Les systèmes ALERT [FRI67] et MIMOLA [MAR79, ZIM879] sont des précurseurs dans ce domaine. Mac PITTS [SIS82], [SOU83], [SOU87], a été le premier compilateur de

comportement commercialement disponible. Il a été le premier à avoir généré un dessin de masques en partant d'une description comportementale. Peu de compilateurs de comportement ont pris en compte les contraintes dues aux étapes de compilation de bas niveau. Parmi les compilateurs de comportement, on peut citer par exemple, et la liste n'est pas exhaustive, les outils : FLAMEL [TRI85], YSC [CAM87], ASP [BAL86], ELF [GIR84], CATHEDRAL II [DEM86], CMU-DA [THO83] ...

II.4.1.2 Les Assistants pour la compilation de comportement

Un assistant pour la compilation de comportement aide le concepteur à générer l'architecture d'un circuit en partant d'une description comportementale. Il faut donc des outils permettant de découper la description d'un circuit en sous-circuits, et des outils pour améliorer un découpage donné. Le processus de compilation peut être contrôlé par l'utilisateur ou par l'assistant. Dans ce dernier cas, l'assistant doit disposer de règles d'enchaînement d'outils et d'outils d'évaluation.

Dans la catégorie des assistants pour la compilation du comportement peu de systèmes existent. Le système ArchitectWorkBench [THO88] peut être classé dans cette catégorie. Le fait qu'il n'y ait pas eu plus tôt d'assistants pour la compilation de comportement peut s'expliquer par le manque d'outils de CAO travaillant au niveau du comportement. Il n'est donc pas étonnant que le premier assistant soit développé par les équipes de CMU, qui disposent du plus grand nombre d'outils de compilation de comportement. Ces outils ont été développés dans le cadre du système CMU-DA [THO83]. L'outil ARTS [BEK87], qui sera présenté dans le chapitre IV, peut aussi être classé dans cette catégorie d'outils.

Le principal obstacle au développement d'assistants pour la compilation de comportement reste l'existence d'outils d'évaluation dont les prédictions et les performances soient acceptables. Ces outils sont nécessaires pour l'évaluation des décisions prises par l'assistant et la critique des décisions prises par l'utilisateur. Il est difficile de réaliser des outils d'évaluation dans le cas général. Ceci est par contre possible dans le cas où l'on utilise des modèles de compilation prédéfinis. Paradoxalement, l'utilisation de modèles prédéfinis limite le domaine de "compétences" de l'assistant et le rapproche du simple compilateur.

II.4.1.3 Les environnements pour la compilation de comportement

Un environnement pour la compilation de comportement met à la disposition des concepteurs un ensemble d'outils et de bibliothèques pour la compilation de comportement. L'interface avec l'utilisateur permet d'accéder à ces outils et bibliothèques, et permet aussi la gestion d'une base de données contenant la description comportementale et les résultats des différents outils et commandes.

La conception d'un circuit va donc consister à définir un scénario pour enchaîner :

- les activations des outils de compilation,
- les commandes d'accès aux bibliothèques de cellules,
- les ordres de mise à jour de la base de données.

Ce scénario est, évidemment, défini par l'utilisateur.

Le système ULYSSE [BUS86] est le seul outil, à notre connaissance, qui peut être classé en tant qu'environnement pour la compilation de comportement. Ce système peut travailler sur plusieurs niveaux d'abstraction, y compris le niveau comportemental. Il fournit un formalisme pour décrire les interfaces avec les outils (fonctions, entrées, sorties...) permettant de générer des interfaces standards avec les outils. Il offre également un langage pour décrire des scénarios d'enchaînement d'outils. Bien que le système ULYSSE ait été utilisé uniquement aux niveaux architectural et structurel, il constitue une bonne approche pour la définition d'un environnement pour la compilation de comportement. Ce système est à l'origine de toute une nouvelle génération de systèmes pour la compilation de silicium, organisée sous forme d'environnement. Les systèmes ADAM [PAR85] et NAUTILE [BON89], par exemple, font partie de ces outils.

II.4.2 Les outils pour la compilation d'architecture

La compilation d'architecture consiste à transposer une architecture en une représentation structurelle tout en tenant compte d'une technologie donnée, décrite par un ensemble de cellules prédéfinies. Une cellule peut réaliser une fonction simple (une porte ou bascule), ou constituer un système entier.

L'étape de compilation d'architecture met en œuvre des compilateurs spécialisés pour générer des parties du circuit. Ces outils peuvent être classés en deux groupes : les systèmes de synthèses logiques et les compilateurs de structures particulières. La synthèse logique consiste à transposer une description logique sous forme de blocs combinatoires et de blocs de mémorisation, en une nouvelle description logique où tous les blocs combinatoires et les blocs de mémorisation correspondent à des éléments d'une bibliothèque de cellules. La nouvelle description doit respecter certaines contraintes de performances spécifiées au préalable. Dans ce groupe, on trouve des systèmes comme MIS [BRA86], basés sur des méthodes algorithmiques, et des systèmes, comme SOCRATES [DEG86] et HADES [FRE85], basés sur des techniques d'intelligence artificielle. Les compilateurs de structures particulières permettent de transposer une description, généralement donnée dans un langage spécialisé, en une structure prédéfinie. Les compilateurs de parties opératives [JAM86a], les compilateurs de parties contrôles [MHA88a] et les compilateurs d'opérateurs spécialisés font partie de ce groupe.

II.4.2.2.1 Les compilateurs d'architecture

SILC [FOX85] et HERCULE [DEM88] sont des compilateurs d'architecture. Dans les deux cas, la description de départ donne l'architecture du circuit sous forme d'un ensemble de blocs interconnectés. Chaque bloc est décrit de manière comportementale. SILC considère trois types de blocs : les blocs de mémorisation, les contrôleurs et les opérateurs. Ces derniers réalisent une fonction combinatoire. Le système génère automatiquement la synchronisation entre les différents blocs.

II.4.2.2 Les assistants pour la compilation d'architecture

La représentation des circuits au niveau de l'architecture se prête bien à la réalisation d'assistants pour la compilation. La transposition de l'architecture se fait par une série de raffinements commandés par l'utilisateur ou par le système. Dans ce dernier cas, les raffinements sont sélectionnés automatiquement selon des règles dont dispose l'assistant. Dans tous les cas, ces actions sont contrôlées conjointement par l'utilisateur et par le système. La part des décisions prises par l'utilisateur et de celles prises par l'assistant est variable. Avec le système DAA (Design Automation Assistant) [KOW85], par exemple, ce partage peut être maîtrisé par l'utilisateur, et si le processus de compilation est commandé entièrement par l'utilisateur, le rôle du DAA se limitera à la critique et à l'exécution des décisions prises par l'utilisateur. Le système DAA est capable de mener par lui même tout le processus de compilation, tout en donnant des explications sur sa démarche, le rôle de l'utilisateur se limitant alors à accepter ou refuser les résultats.

Le système SEHWA [PAR87] peut également être classé parmi les assistants pour la compilation d'architecture. SEHWA est spécialisé dans les architectures pipe-line. Partant d'une architecture initiale, SEHWA permet de générer une structure en fonction de contraintes imposées par l'utilisateur.

D'autres assistants tels que CHIP-COMPILER (de VTI) ou le CHIP-ASSISTANT (d'ES2) sont plus généraux, et ne fixent pas de limitation à la nature des circuits générés. Par contre, ils sont beaucoup moins "intelligents" et ne prennent pas de décisions : ils se contentent d'assister l'utilisateur à l'aide d'outils d'évaluation et de vérification. Ces deux systèmes utilisent des techniques algorithmiques simples. A titre de comparaison, DAA est organisé comme un système expert où la base de connaissances, la base de données et le moteur d'inférences sont séparés.

II.4.2.3 Les environnements pour la compilation d'architecture

Un environnement pour la compilation d'architecture utilise aussi une bibliothèque de cellules prédéfinies, qui peuvent être des sous-systèmes complexes paramétrés, et un ensemble de compilateurs spécialisés. Une base de données contient les parties du circuit déjà réalisées. Le processus de compilation consiste à définir des cellules complexes par assemblage de cellules existantes ou par utilisation de compilateurs spécialisés.

Il existe plusieurs environnements pour la compilation d'architecture commercialisés. Le succès de ces outils peut s'expliquer par deux faits :

- Ces compilateurs travaillent au même niveau que celui que les concepteurs systèmes sont habitués à manipuler. Ils sont donc bien acceptés par les ingénieurs systèmes.
- L'existence de plusieurs systèmes de qualité industrielle et commercialisés, tels que GENESIL [CHE87], CONCORDE [GAJ86], VTI [LIP87], etc.

Ces systèmes souffrent d'un manque d'ouverture quand il s'agit d'ajouter un nouvel outil ou un nouvel élément dans la bibliothèque. L'introduction d'une nouvelle cellule dans le système GENESIL, par exemple, est un processus long et fastidieux. Ainsi, la nouvelle cellule doit respecter un certain nombre de contraintes comme l'utilisation d'une horloge à deux temps. D'autre part, les différents aspects de la cellule (modèle de simulation, modèle de consommation, représentation symbolique...) doivent être décrits dans plusieurs langages différents.

II.4.3 Les outils pour la compilation de structure

Ces outils sont aussi appelés outils pour la génération de dessins des masques. La compilation de structure consiste à générer la représentation physique d'un circuit ou d'une partie de circuit en partant de sa description structurelle. Ce processus doit tenir compte d'un procédé de fabrication donné. Ce dernier point est crucial car la durée de vie des procédés de fabrication est assez courte. Dans certains cas, il arrive que cette durée de vie soit inférieure au temps exigé par le processus de conception. Pour s'affranchir de ce problème, plusieurs solutions ont été proposées, elles sont basées sur deux techniques :

- L'utilisation de motifs de base, dont la réalisation est redéfinie pour chaque procédé de fabrication, et l'utilisation d'outils de composition paramétrés, eux aussi, par le procédé de fabrication.
- L'utilisation d'un procédé de fabrication générique, définit par des règles de dessins et des modèles électriques. Le passage au procédé de fabrication réel se fera par une adaptation des formes géométriques ("resizing") appliquée au dessin des masques. Les modèles électriques sont utilisés pour l'estimation des performances durant le processus de conception.

L'utilisation de ces techniques entraîne une diminution des performances des circuits générés. Les deux techniques citées ci-dessus peuvent être combinées dans un même système.

Dans le cas où l'on utilise des motifs de base, durant le processus de conception, seuls les aspects physiques externes et les modèles de fonctionnement de la cellule sont considérés. On peut distinguer quatre degrés de complexité pour les motifs de base : les segments et articulations [JER85], [ROU87], [CHA88], les symboles électriques de base (transistors, fils, contacts) [BUR86], les portes logiques et les macro-blocs. Les systèmes utilisant les deux premiers sont dits symboliques.

La définition des motifs de base peut être manuelle ou automatique. Dans ce dernier cas, on utilise des techniques de génération automatique de cellules paramétrées par la technologie. Il s'agit d'écrire des programmes qui, en s'exécutant, utilisent les caractéristiques de la technologie comme paramètres externes pour générer les cellules de base. Dans la cas des systèmes symboliques, l'adaptation d'une description à un procédé de fabrication donné peut se faire à l'aide d'outils de compactage [WOL87].

La paramétrisation des outils peut être facilitée par l'adoption de restrictions via les règles de composition, comme l'interdiction de chevauchement de cellules par exemple.

La complexité et la diversité des motifs de base est un facteur important pour la mesure de dégradation de performances occasionnée par l'utilisation de cette technique. La modélisation d'un procédé de fabrication par des portes logiques entraîne plus de pertes de surface et de vitesse que dans les cas où la technologie est modélisée par des cellules élémentaires telles que les fils, transistors et contacts, comme motifs. Inversement, l'utilisation de cellules élémentaires entraîne plus de complications pour réaliser des outils paramétrés par le procédé de fabrication.

Dans le cas où l'on utilise un procédé de fabrication générique, le passage au procédé de fabrication réel se fait par une adaptation purement géométrique, qui ne prend pas en considération les problèmes électriques. Cette technique est plus facile à réaliser mais peut donner des résultats peu efficaces si le procédé de fabrication cible est très différent du procédé de fabrication générique. En fait, tout dépend de la définition du procédé générique et donc du nombre et des similitudes des procédés de fabrication que l'on veut couvrir. Les expériences du projet "circuit multi-projet français ont déjà prouvé que cette technique peut être efficace dans le cas où les procédés de fabrication sont voisins. Par contre, les modèles électriques se sont avérés peu utiles pour cause d'imprécision. Le problème de l'indépendance des compilateurs de silicium vis à vis de la technologie sera discuté en IV.10.

II.4.3.1 Les compilateurs de structure

Les outils organisés sous forme de compilateurs de structure sont plus connus sous la dénomination de systèmes de placement et de routage. [SAN87] présente en détail le principe de fonctionnement de ces outils, en fournissant une bibliographie récente. La plupart de ces outils utilisent des cellules standards ou des circuits prédiffusés. Pour être indépendant du procédé de fabrication, les outils de routage, et de placement dans le cas de l'utilisation des cellules standards, doivent être paramétrés, ce qui est relativement facile à réaliser.

II.4.3.2 Les assistants pour la compilation de structure

La compilation de structure est un domaine où les techniques algorithmiques sont encore très largement utilisées. Ceci peut s'expliquer par des raisons de coût de calcul. Vue la complexité des traitements et la quantité d'informations manipulées, les machines actuelles restent trop lentes pour envisager des assistants pour la compilation de structure. La complexité des traitements provient essentiellement du fait qu'il faut manipuler à ce niveau la représentation physique. Cette représentation est souvent composée de plusieurs facettes, ce qui complique la tâche de l'assistant qui doit contrôler la consistance de la base de données durant le processus de conception.

JOOBANI [JOO86] présente un système expert pour le routage, appelé WEAVER, dont l'organisation est voisine de celle de l'assistant défini plus haut. Les routages obtenus par WEAVER sont meilleurs que ceux obtenus par des routeurs classiques. Ces résultats concernent de petits exemples, mais le système tient compte d'un problème particulier de routage : le routage dans un rectangle ("switch-box routing"). Le routage complet d'un grand circuit à l'aide de WEAVER nécessiterait un temps de calcul exorbitant.

II.4.3.3 Les environnements pour la compilation de structure

Un environnement pour la compilation de structure offre une interface qui permet à l'utilisateur d'accéder à :

- Une base de données, qui contient la représentation physique du circuit durant le processus de conception,
- Une bibliothèque de cellules décrites de manière physique,
- Un ensemble d'outils pour la génération, la construction, la modification et la vérification des représentations physiques.

L'interface peut être graphique ou textuelle. La plupart des systèmes classiques pour la conception de circuits au niveau physique peuvent être vus comme des environnements pour la compilation de silicium [ROU87]. Bien entendu, ces environnements sont plus ou moins

complets. ROUGEAUX donne une bibliographie complète sur ces systèmes dans [ROU87]. Plusieurs de ces systèmes offrent à la fois une interface graphique et textuelle.

Le compilateur de silicium SYCO utilise un environnement pour la compilation de structure appelé NAUTILE. Le quatrième chapitre donne d'autres éléments bibliographiques concernant les environnements pour la compilation de silicium.

II.5 Conclusion

La conjugaison de l'évolution des ordinateurs, de l'évolution des techniques de représentation et de gestion des données et de l'évolution des techniques de conception des circuits intégrés a entraîné une grande évolution de la CAO de VLSI. Il y a dix ans, les outils de CAO étaient spécialisés et ne concernaient que des étapes très limitées du processus de conception telles que la vérification des règles de dessin, la saisie du dessin des masques et la simulation électrique.

Aujourd'hui, même si l'on est encore loin du système complet de CAO de VLSI tel qu'il est présenté en figure II.3, on trouve déjà des systèmes intégrant plusieurs outils et travaillant sur plusieurs niveaux d'abstraction et dans plusieurs domaines de description.

L'absence d'une meilleure formalisation des méthodes de conception et d'une meilleure définition des différentes représentations des C.I. constitue un handicap qui nous sépare du système complet de CAO de VLSI. La compilation de silicium et la preuve formelle de circuits semblent être les deux démarches les plus prometteuses pour surmonter ce handicap.

L'étude des compilateurs de silicium éxistants présentée a été placée dans un contexte plus large qui couvre les méthodologies de conception des circuits intégrés. L'utilisation d'une hiérarchie de niveaux d'abstraction adaptée à la compilation de silicium a permis une classification des compilateurs de silicium en fonction des étapes de conception qu'ils effectuent. Ceci fait apparaître les différentes techniques utilisées pour la compilation de silicium. Les techniques utilisées à chaque niveau dépendent des connaissances et des concepts manipulés.

Au niveau comportemental, il faut réussir un bon découpage du circuit en sous-circuits. Un découpage est réussi dans le cas où il produit une architecture pouvant être réalisée de manière efficace. Le problème général est difficile à résoudre, car il est difficile de réaliser des outils d'évaluation travaillant à ce niveau pour estimer l'efficacité d'une solution donnée. L'utilisation d'une architecture fixe rend ce genre de compilateurs possible mais le limite à une classe restreinte d'applications.

Le niveau architectural met en œuvre des compilateurs spécialisés pour la compilation séparée des différents sous-systèmes. Ce jeu de compilateurs spécialisés limite l'espace des solutions envisageables pour un problème donné. D'autre part, l'assemblage des sous-systèmes entraîne, généralement pour des raisons d'efficacité, l'utilisation d'une stratégie globale fixe pour la synchronisation et pour le plan de masse, ce qui peut aussi limiter le champ d'application du compilateur.

Au niveau des compilateurs de structure, il s'agit de gérer une grande quantité d'information pouvant contenir des redondances. La relation avec les processus de fabrication rend ce niveau sensible aux variations technologiques.

L'étude de l'organisation des outils de compilation de silicium a permis une classification fine des compilateurs de silicium existants.

Un compilateur est un système dont l'utilisation est simple, mais comme il renferme un schéma de compilation fixe, il offre une solution unique pour une description donnée. Des outils d'optimisation peuvent atténuer cette rigidité.

Un assistant permet d'explorer un espace de solutions plus riche que celui offert par un compilateur. Il nécessite une plus grande interaction avec le concepteur, donc une plus grande qualification de ce dernier. Ce type de systèmes est extensible. Les connaissances de l'assistant peuvent être enrichies par de nouvelles règles. L'acquisition, la représentation et l'utilisation des connaissances de manière efficace nécessitent une meilleure formalisation des techniques et des méthodes de conception de circuits intégrés [KOW85]. Ce handicap fait que les assistants existants sont spécialisés dans un domaine restreint d'applications.

Un environnement offre des facilités pour la conception sans pour autant contrôler le processus de conception. L'utilisateur d'un tel outil doit avoir une bonne connaissance des concepts qu'il manipule et une bonne connaissance du système lui même. En fait un environnement correspond à "l'automatisation du stylo et du cahier du concepteur". Il permet donc de réaliser des circuits aussi efficaces que ceux réalisés manuellement.

Le compilateur complet, tel qu'il est décrit dans la figure II.4, n'existe pas encore. Ce schéma constitue une évolution naturelle des compilateurs de silicium vers des environnements complets pour la compilation de silicium. Un environnement complet pour la compilation de silicium est un système hiérarchique avec plusieurs points d'entrées.

CHAPITRE III

TECHNIQUES DE COMPILATION DE COMPORTEMENT

Ce chapitre constitue une étude des techniques de compilation de comportement. Comme il a été défini au chapitre précédent un compilateur de comportement génère l'architecture d'un circuit en partant d'une description comportementale. Indépendamment de son organisation, un tel système est composé de:

- un langage de description de comportement,
- une ou des procédures pour la génération de l'architecture,
- un ensemble de sous-systèmes et de compilateurs spécialisés prédéfinis,
- éventuellement, des procédures d'optimisation.

Les compilateurs de comportement et les compilateurs de langages de programmation ont en commun plusieurs techniques. Les techniques communes utilisées par les deux types de compilateurs se trouvent au niveau du langage de description, de la représentation interne (appelée aussi forme intermédiaire) utilisée par les compilateurs et de certaines techniques d'optimisation. En plus, la compilation de silicium impose des contraintes spécifiques à la nature des circuits intégrés. La première section de ce chapitre analyse quelques différences entre les deux types de compilateurs. Les autres sections seront consacrées aux techniques utilisées pour la compilation de comportement. Ces techniques sont introduites de manière informelle dans la section 2 par deux exemples.
Après une présentation des langages de description et des formes intermédiaires, les problèmes de l'optimisation, de l'allocation et de l'ordonnancement seront étudiés. L'avant dernière section est consacrée à l'application de ces techniques dans le cas de la compilation orientée vers les circuits de commande et de la compilation orientée vers les circuits de communication. La dernière section examine les problèmes posés par la définition d'un compilateur de silicium.

III.1 Compilateurs de langages de programmation et compilateurs de silicium

Le rapprochement entre compilateurs de comportement et compilateurs de langages de programmation vient de la similitude entre les langages de description de comportement et les langages de programmation. Notons que plusieurs langages de programmation ont été utilisés pour spécifier le comportement des circuits intégrés.

Pour une comparaison entre la compilation des langages de programmation et la compilation de silicium, on peut se reporter à [AYR83], [JOH81] et [ULL84]. Comme il a été remarqué, à juste titre, dans [DUR88], le fait que des auteurs tels que Ullman et Ayres se soient intéressés à la compilation de silicium montre bien que cette discipline est proche de la compilation classique. Il faut remarquer aussi que le fait qu'ils n'y sont pas restés montre que la compilation de silicium présente des particularités. Le but de cette remarque est de souligner les enseignements qu'il faut tirer de l'évolution des compilateurs de langages de programmation d'une part et d'analyser les contraintes spécifiques aux compilateurs de silicium d'autre part.

Après 30 ans d'expérience, les techniques actuelles de compilation ne permettent pas la réalisation de compilateurs efficaces pour toutes les catégories de langages de programmation. L'efficacité d'un compilateur est étroitement liée à la correspondance entre le langage source et la machine cible du compilateur. On considérera que l'efficacité est fonction de deux paramètres : la taille de la mémoire nécessaire pour l'exécution du code généré par le compilateur et la vitesse d'exécution du code compilé. Un recensement rapide des langages de programmation actuels et des compilateurs associés à ces langages, sur les ordinateurs classiques (machines séquentielles), peut montrer que seuls les langages dit impératifs (ou procéduraux) possèdent des compilateurs performants. Ceci est dû au fait qu'un langage impératif, tel que Fortran, Pascal, C ..., peut être vu comme l'abstraction d'une machine séquentielle. Parmi les autres langages, on peut distinguer les langages dits fonctionnels tel que Lisp et les langages dits déclaratifs tel que Prolog, qui sont l'abstraction d'autres types de machines [BAC78]. La compilation de ces langages sur une machine séquentielle nécessite donc le génération d'un environnement d'exécution permettant d'adapter le code généré à la machine cible. Si on compile un programme Prolog sur un ordinateur classique, on obtient un code dont l'exécution nécessitera un environnement Prolog.

Dans le cas des langages impératifs, le mécanisme de compilation est direct (à chaque construction du langage d'entrée est associé une suite d'instructions de la machine cible), donc simple à comprendre. Ceci fait que l'on peut contrôler le résultat de la compilation en modifiant la description d'entrée. Ce contrôle permet de réaliser des optimisations en fonction des contraintes de performances citées plus haut. On peut aussi réaliser des choix dans les compromis possibles entre plusieurs critères d'efficacité. Le dernier type d'optimisation n'est pas toujours possible à réaliser dans le cas des langages non impératifs compilés sur des machines séquentielles.

Dans le cas des compilateurs de comportement, la machine cible est définie par le compilateur lui même. Le problème général, qui consiste à générer la meilleure architecture pour une description donnée, est difficile à résoudre à cause de la grande étendue de l'espace des solutions [PAR88] [JAM86a]. Comme il a été dit au chapitre II, l'utilisation d'une architecture cible permet de limiter l'ensemble des solutions possibles. Par contre, un tel choix limite l'efficacité du compilateur à une classe restreinte d'applications. On considère que l'efficacité d'un compilateur de silicium peut être mesuré par trois facteurs : la vitesse, la consommation et la surface du circuit généré.

D'autre part, en regard des techniques utilisées par les compilateurs de comportement, il est plus difficile d'établir une correspondance entre le langage de description et l'architecture cible. Dans le cas des langages de programmation, la transposition de la description source dans un langage objet se fait selon des règles qui font correspondre à chaque construction du langage source (instruction simple ou complexe) une ou plusieurs séquences d'instructions dans le langage objet. Cette technique de transposition (appelée aussi compilation directe [CAM85] [McF88]) fait que les deux descriptions ont la même structure : à un bloc (ensemble

d'instructions) de la description d'entrée correspond un bloc de la description objet. Dans le cas des compilateurs de comportement, les deux descriptions (description du comportement et description de l'architecture) ont, par définition, des organisations différentes (voir la section 2 du deuxième chapitre). Il faut rappeler que le but du compilateur de comportement est de générer l'architecture, donc de réaliser un "découpage" du circuit en sous-systèmes. On peut donc trouver des blocs de la description d'entrée auxquels ne correspond aucun bloc dans l'architecture, ainsi que des éléments de l'architecture qui n'ont pas d'équivalents directs dans la description du comportement. Par exemple, la description d'un microprocesseur peut contenir une procédure (ou un sous-programme) pour la lecture de l'instruction suivante, sans que l'architecture du microprocesseur correspondant ne contienne de bloc réalisant cette lecture. Par ailleurs, le découpage d'un microprocesseur en une partie opérative et une partie contrôle ne doit rien à l'organisation de la description comportementale du microprocesseur.

Cette technique de compilation directe, en tant que telle, est donc inadaptée pour la réalisation du découpage. Par contre, l'organisation de la description comportementale est souvent utilisée pour guider ce découpage. Le compilateur SYCO, par exemple, utilise la hiérarchie de la description comportementale pour définir des contrôleurs organisés en niveaux d'interprétation.

L'expérience avec les langages de programmation montre aussi que le code généré automatiquement par un compilateur est, dans la plupart des cas, moins performant que le code produit manuellement.

Dans le cas des langages de programmation, le développement des techniques de compilation a permis de réduire progressivement cette perte de performances. D'autre part l'évolution de la technologie a permis, elle, de réduire notablement le coût occasionné par une telle perte de performances, au niveau de la vitesse et au niveau de la mémoire occupée. Cette évolution des compilateurs et de la technologie a permis la réalisation de systèmes plus complexes que ceux réalisables manuellement.

Dans le cas des compilateurs de silicium, on se heurte à la fois à la jeunesse des compilateurs et au coût du silicium. Le manque d'expérience dans le domaine de la compilation de comportement fait que les circuits générés automatiquement par ces compilateurs sont souvent plus gros et plus lents. Par exemple, l'estimation du résultat de la compilation d'un circuit tel que le MC68000™ par la première version du compilateur SYCO avait donné un circuit deux fois plus grand et huit fois plus lent [GER85]. Ce résultat, qui parait aberrant aujourd'hui, était jugé très satisfaisant. Il faut rappeler qu'en 1985, époque où a été faite cette estimation, aucun compilateur de silicium ne pouvait prétendre compiler un circuit aussi complexe. Les coûts d'une dégradation de la vitesse et/ou de la consommation est difficile à estimer, car il est étroitement lié au type d'applications auquel est destiné le circuit. Il faut noter qu'il existe des applications peu sensibles à la perte de vitesse, on peut citer comme exemple les circuits pour la commande de mouvements mécaniques. Par contre, le coût d'une augmentation de surface peut être facilement estimé. En fait le coût de fabrication d'un circuit est très sensible

à sa surface. Denyer [DEN88] estime qu'une augmentation de 15% de la surface d'un circuit entraîne une multiplication de son coût de fabrication par un facteur 1.75, et qu'une augmentation de 50% de la surface entraîne une multiplication du coût par un facteur 15. Il faut rappeler aussi que la taille du circuit ne doit pas dépasser les limites fixées par le processus de fabrication. Donc, pour qu'une augmentation de surface soit acceptable, il faut que le circuit vérifie les deux contraintes suivantes:

- La réduction du coût de conception du circuit doit être supérieure ou égale à l'augmentation du coût de production total (il faut que le nombre de circuits à fabriquer soit connu au moment de la conception).
- La surface du circuit ne doit pas dépasser les limites autorisées par la technologie.

On peut remarquer que les deux contraintes citées plus haut sont contradictoires dans le cas des circuits complexes. La réduction du coût de conception augmente avec l'accroissement de la complexité du circuit. Donc plus un circuit est complexe, plus il est intéressant de le générer automatiquement à l'aide d'un compilateur. Paradoxalement la perte de surface occasionnée par l'utilisation d'un compilateur de silicium augmente avec la complexité du circuit. Donc plus un circuit est complexe, plus l'utilisation d'un compilateur peut entraîner le dépassement des limites de la technologie.

Le test et la mise au point posent des problèmes spécifiques dans le cas des circuits intégrés [ULL84]. Le schéma classique de mise au point, qui consiste à exécuter un programme pour vérifier s'il est correct, n'est pas applicable dans le cas des circuits intégrés. La fabrication d'un circuit, l'équivalent de l'exécution dans le cas d'un langage de programmation, coûte trop cher pour permettre de nombreux essais. Les circuits doivent donc être vérifiés avant l'étape de fabrication. Dans le cas des compilateurs de silicium cette vérification est d'autant plus facile que le niveau du langage d'entrée du compilateur est élevé. D'autre part le processus de fabrication lui même n'étant pas fiable à 100%, les erreurs de fabrication peuvent empêcher le bon fonctionnement d'un circuit bien conçu. Le processus de conception d'un circuit doit donc contenir la définition d'une procédure de test. Celle-ci consiste généralement à définir une séquence de vecteurs de tests permettant de détecter les erreurs de fabrication. La longueur de la séquence de vecteurs de test définissant le coût de l'étape de test, il faut donc réduire au maximum la longueur de cette séquence. Dans le cas d'une conception manuelle, la réduction est faite en tenant compte de la fonction et de la structure du circuit. Dans le cas d'un compilateur de silicium, il est difficile, voire impossible, de générer automatiquement des vecteurs de test en partant des circuits générés par le compilateur. Une solution à ce problème consiste à intégrer des procédures de test dans le circuit généré pour le rendre autotestable. Cette dernière solution a été retenue dans le cadre du compilateur SYCO [TOR88].

III.2 Techniques de compilation

Pour maîtriser la complexité du processus de conception d'un circuit intégré, il a fallu définir des méthodologies de conception dont le champ d'applications soit, lui aussi, limité à une classe de circuits. Deux méthodologies ont été discutées au premier chapitre. La première concerne les circuits de communication, tandis que la seconde est orientée vers les circuits de contrôle. Ces deux méthodologies ont engendré deux catégories de compilateurs de silicium spécialisés dans les mêmes classes de circuits. Ils seront désormais appelés compilateurs de circuits de contrôle et compilateurs de circuits de communication. La grande majorité des compilateurs de comportement actuels appartient à l'une ou l'autre de ces deux catégories.

Le processus de compilation d'un circuit nécessite généralement plusieurs étapes qui emploient des techniques différentes. Les compilateurs de circuits de commande sont essentiellement basés sur des techniques d'ordonnancement et partent d'une description procédurale de la fonction du circuit, qui décrit les flux de contrôle dans le circuit. Les compilateurs de circuits de communication acceptent, eux, en entrée, une description fonctionnelle (ou parallèle) qui décrit les flux de données dans le circuit. Ils utilisent des algorithmes essentiellement basés sur des techniques d'analyse de flux de données. Ces deux types de compilateurs seront illustrés par deux exemples.

III.2.1 Compilation d'un circuit de commande

Un circuit de commande réalise un traitement complexe sur un faible flux d'informations. On suppose que le langage de départ est de type impératif. Le comportement du circuit est décrit en terme d'actions à exécuter dans un ordre donné. Les langages de description de comportements impératifs sont similaires aux langages de programmation de même type. La figure III.1 représente une description en Pascal de la fonction PGCD (elle calcule le Plus Grand Commun Diviseur de deux entiers). Cet exemple a également été utilisé par [CAM87].

```
PROCEDURE pgcd(x, y: ENTIER; VAR  r : ENTIER);
    DEBUT
      TANTQUE x<>y
        FAIRE
          SI x<y ALORS y:=y-x SINON x:=x-y FINSI
        FINFAIRE
        r:=x
    FIN
```

Figure III.1 Description en Pascal de la fonction PGCD

La description donnée par la figure III.1 est insuffisante pour générer un circuit réalisant la fonction PGCD. Il lui manque les informations concernant la communication du circuit avec l'extérieur, soit :

- le protocole de communication avec le monde extérieur (protocole pour la lecture des

données en entrée et l'écriture des résultats en sortie),

- les conditions de démarrage du circuit,
- le format des entiers traités.

Certaines de ces informations sont difficiles à exprimer dans un langage de programmation classique tel que Pascal ou Ada. La figure III.2 présente la spécification d'un circuit réalisant la fonction PGCD écrite dans un langage de description de matériel appelé LDS [LAU85]. Le chapitre IV contient une présentation générale du langage LDS.

```
SMODULE pgcd( ex, ey, rst,r)
SIGNAL;
   xi 0:8, IN; ? Bus d'entrée de la donnée x
   yi 0:8, IN; ? Bus d'entrée de la donnée y
   r 0:8 , OUT; ? Bus de sortie du résultat r
   rst 0:1, IN, CTRL; ? Signal d'initialisation (reset)
END;
REGISTERS;
   x 0:8;
   y 0:8;
END;
LINK pgcd; ? lien vers la description de la fonction interne.
END;
CMODULE pgcd;
<start>
   WHILE (rst=0) DO; ENDDO;
   x:=xi; y:=yi;
   WHILE NOT(x=y)
      DO IF (x<y ) y:=y-x; ELSE x:=x-y; ENDIF;
      ENDDO
   r:=x;
   NEXT start;
END
```

Figure III.2 Description LDS du circuit PGCD.

La description LDS est composée de deux parties ; une partie déclaration (SMODULE) qui décrit l'interface du circuit PGCD avec l'extérieur et une partie qui décrit l'algorithme d'interprétation réalisé par le circuit. En plus des variables (x et y) utilisées par l'algorithme, la description contient des signaux d'entrées/sorties et un signal d'initialisation. Dans notre cas, la description fixe le format des données manipulées. Les variables x et y sont déclarées en tant que registres de 8 bits. Le protocole de communication du circuit avec l'extérieur est simple. Cette description suppose qu'à chaque fois que le signal d'initialisation (rst) est à "1" (non égal à "0") les données x et y sont présentes sur les bus externes xi et yi. Le résultat est transmis via le bus externe (r) à chaque fois qu'il est calculé. La description proprement dite du comportement, c'est à dire l'algorithme d'interprétation réalisé par le circuit, est similaire à la description de la figure III.1. Elle est composée d'une boucle infinie qui consiste à lire les données, calculer le PGCD, pour, enfin, écrire le résultat sur le bus externe.

Cette description peut constituer un point de départ pour la génération automatique de l'architecture du circuit PGCD.

La partie contrôle peut être obtenue directement à partir de la description d'entrée. Il suffit de transposer cette dernière en une machine d'états finis où chaque état renferme l'exécution d'une opération de l'algorithme de départ.

La génération de la partie opérative peut être réalisée par une méthode directe qui consiste à :

- associer un registre à chaque variable de la description,
- associer une unité de calcul à chaque opération de calcul réalisée par la description,
- pour chaque opération de transfert, générer les connexions nécessaires entre la source et la destination. En cas de conflit (plusieurs sources pour une même destination) il faut générer des multiplexeurs. Cette étape est souvent appelée étape d'allocation du matériel.

Dans le cas où la description contient des expressions de condition qui font appel à des opérateurs, il faut créer des registres de un bit pour mémoriser la valeur des expressions.

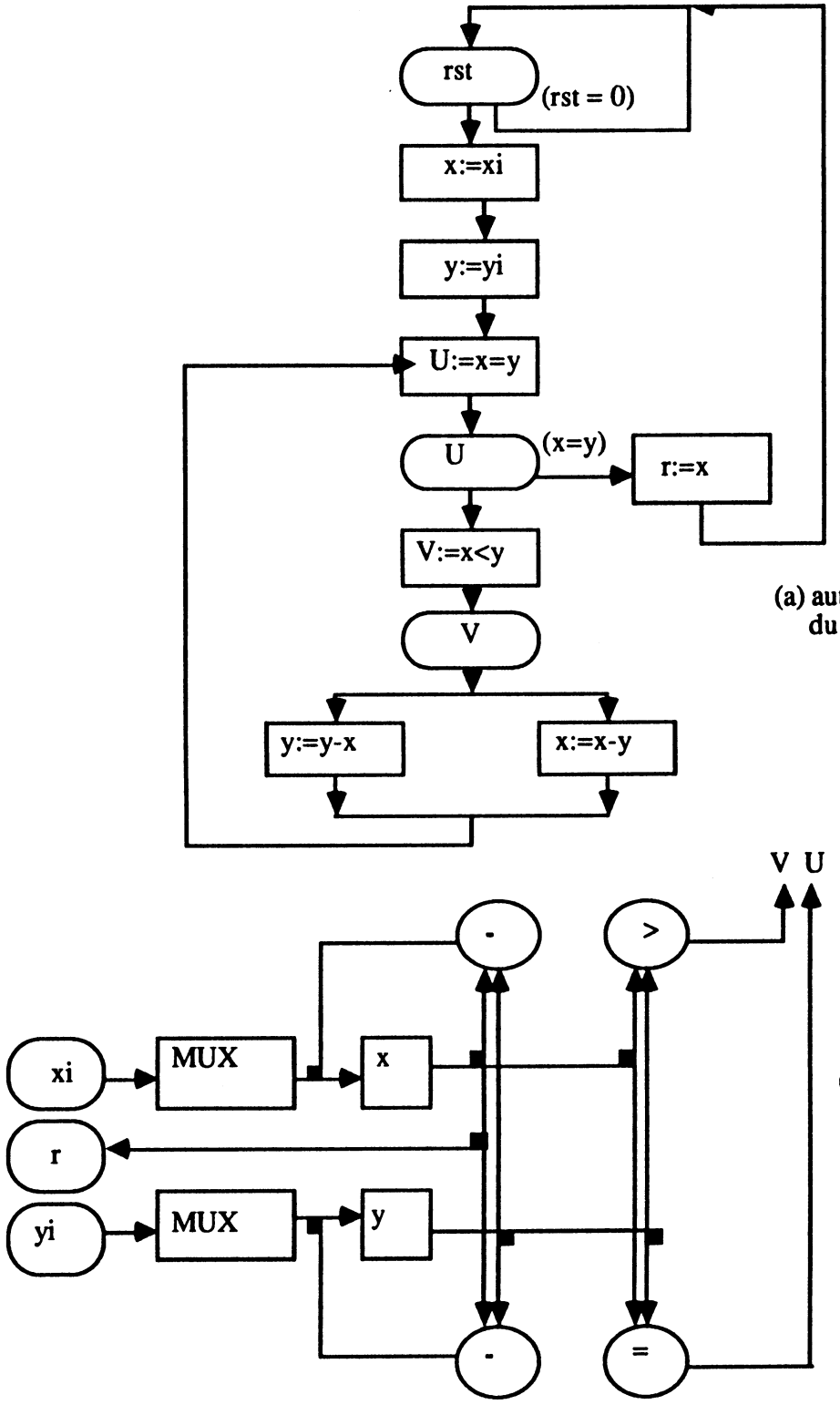
La figure III.3 montre l'architecture obtenue par les algorithmes cités plus haut. Notons que le terme architecture est utilisé de manière abusive dans ce cas. Cet abus de langage est courant dans la littérature de la compilation de silicium.

Pour compléter la définition de l'architecture et pour que la partie contrôle (figure III.3.a) et la partie opérative (figure III.3.b) puissent fonctionner ensemble, il faut donner un modèle de synchronisation qui soit à la fois compatible avec les registres et les opérateurs de la partie opérative, et compatible avec la réalisation du contrôleur. Les registres U et V ont été rajoutés dans la partie opérative pour mémoriser les résultats des expressions conditionnelles.

L'architecture obtenue peut constituer un point de départ pour les autres étapes de la compilation qui vont générer la structure, puis le dessin des masques du circuit correspondant. Elle peut aussi subir des modifications en vue d'améliorer ses performances ou en vue d'atteindre des contraintes de performances. Améliorer la partie contrôle consiste essentiellement à réduire le nombre d'états. Dans le cas de la partie opérative il s'agit essentiellement de partager les opérateurs et les chemins de données.

Dans le cas de l'exemple décrit plus haut, la partie opérative contient un opérateur pour chaque opération de la description d'entrée et un chemin de données pour chaque transfert. Dans ce cas précis, il est facile de remarquer que plusieurs opérations peuvent partager le même opérateur et que certains chemins de données peuvent être partagés. La figure III.4 montre deux nouvelles solutions architecturales obtenues par deux modifications successives. La première (figure III.4.a) consiste à remplacer toutes les opérations par une UAL. La seconde modification (figure III.4.b), remplace les multiplexeurs et les chemins de communication par deux bus. Ces modifications peuvent entraîner une réduction de la surface du circuit.

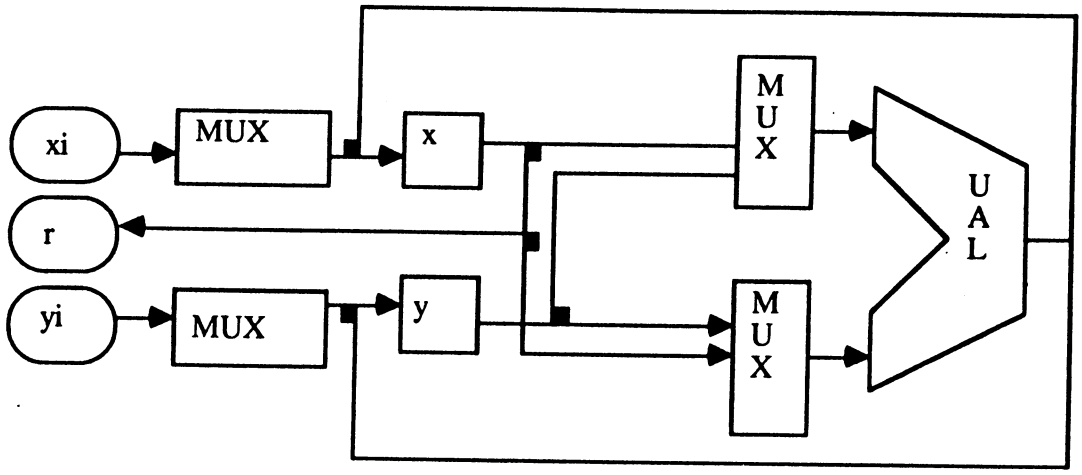
Cependant, l'utilisation d'une UAL unique interdit l'exécution d'opérations parallèles.



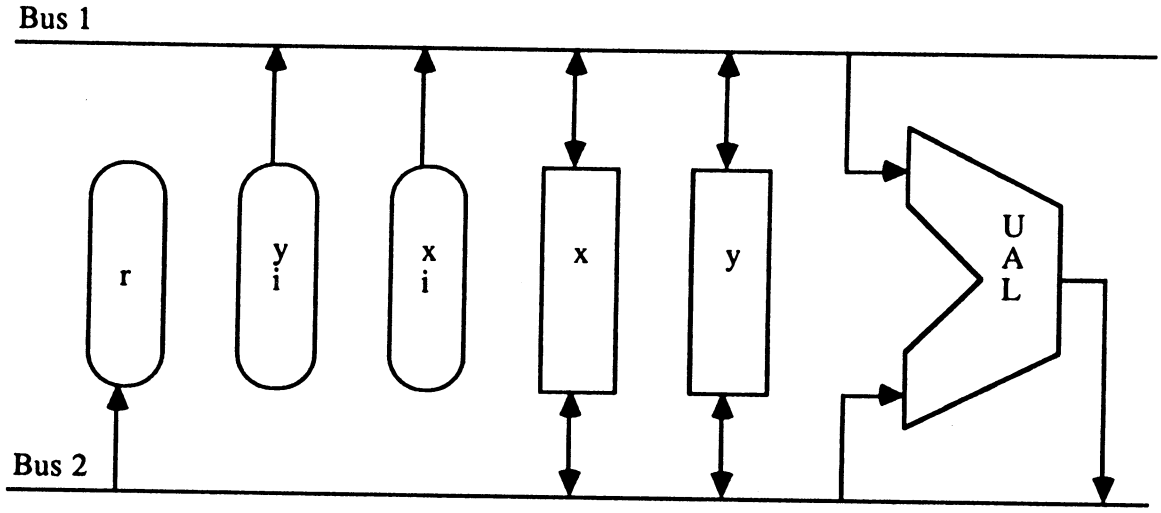
(a) automate de contrôle du circuit pgcd.

(b) Partie opérative de la fonction pgcd: Solution naïve

Figure III.3 Solution résultat d'une compilation directe de la fonction PGCD



(a) Partie opérative après regroupement des opérateurs



(b) Partie opérative organisée avec des bus

Figure III.4 Exemples d'améliorations possibles de la partie opérative

La partie contrôle générée plus haut considère que chaque opération nécessite une étape de contrôle et donc un cycle d'exécution. Plusieurs de ces étapes de contrôle peuvent être regroupées. Un tel regroupement entraîne une réduction du nombre d'étapes de contrôle, une diminution de la surface nécessaire pour la réalisation du contrôleur (cette réduction dépend de l'organisation choisie pour réaliser le contrôleur), et une amélioration de la vitesse. Chaque état du nouveau graphe peut commander plusieurs opérations en parallèle. Les décisions de regroupement des opérations dans une même étape de contrôle sont basées sur une analyse des flux de données [FIS81] [AHO86]. Ces modifications supposent donc que la partie opérative a préalablement été construite.

La figure III.5 donne un nouveau graphe de contrôle compacté en utilisant la partie opérative initiale (figure III.3.a).

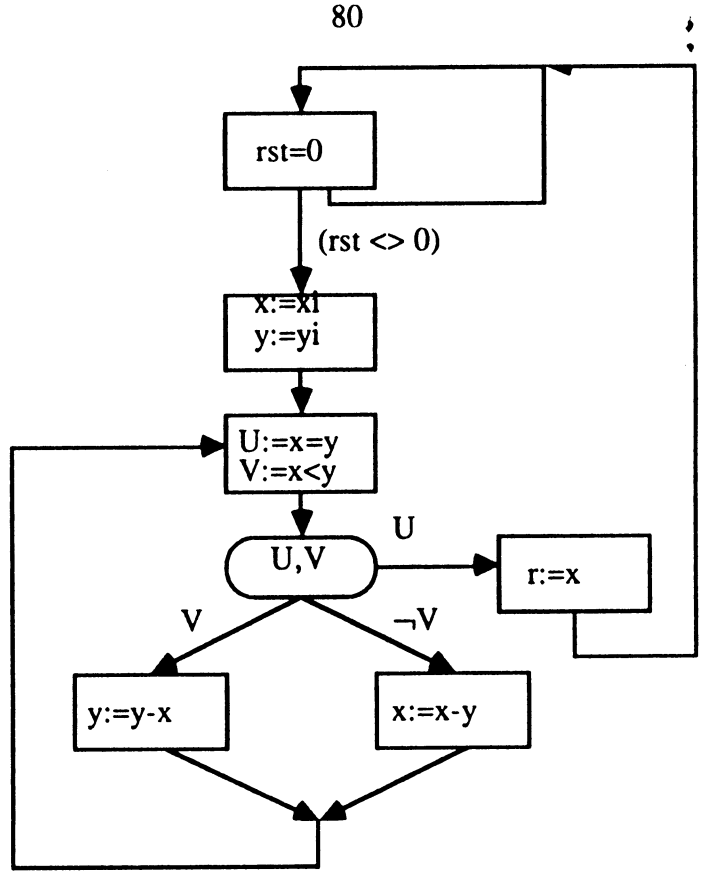


Figure III.5 Automate de contrôle après compactage.

III.2.2 Compilation d'un circuit de communication

Un circuit de communication réalise un traitement peu complexe pour un grand débit d'informations. Il a le plus souvent une architecture parallèle et sa spécification est basée sur les flux de données du circuit. La spécification peut être faite à l'aide d'un langage fonctionnel en décrivant les équations qui calculent les sorties en fonction des entrées. On verra plus loin (III.4 et III.9) que la plupart des compilateurs qui génèrent des circuits de communication partent plutôt d'un représentation de la description sous forme de graphes de flux de données. Les techniques d'extraction de graphes de flux de données à partir d'une description comportementale seront examinées plus tard (voir III.10).

On prendra comme exemple l'équation "$y = b^2 - 4ac$" [NEW86]. Le processus de compilation de sa description sera commenté.

La figure III.6 montre le graphe de flux de données correspondant à la fonction exemple. Les sommets du graphe représentent les fonctions des transformations que doivent subir les données. Les arcs du graphe représentent des chemins de données qui véhiculent des valeurs entre les fonctions. L'opérateur DUP permet de dupliquer une valeur, il est introduit pour des raisons de synchronisation.

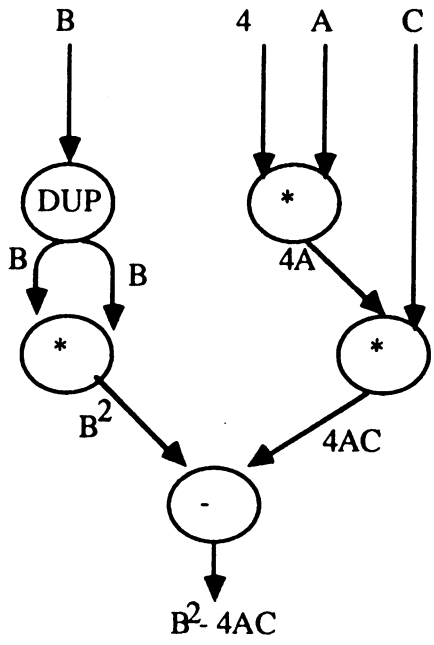
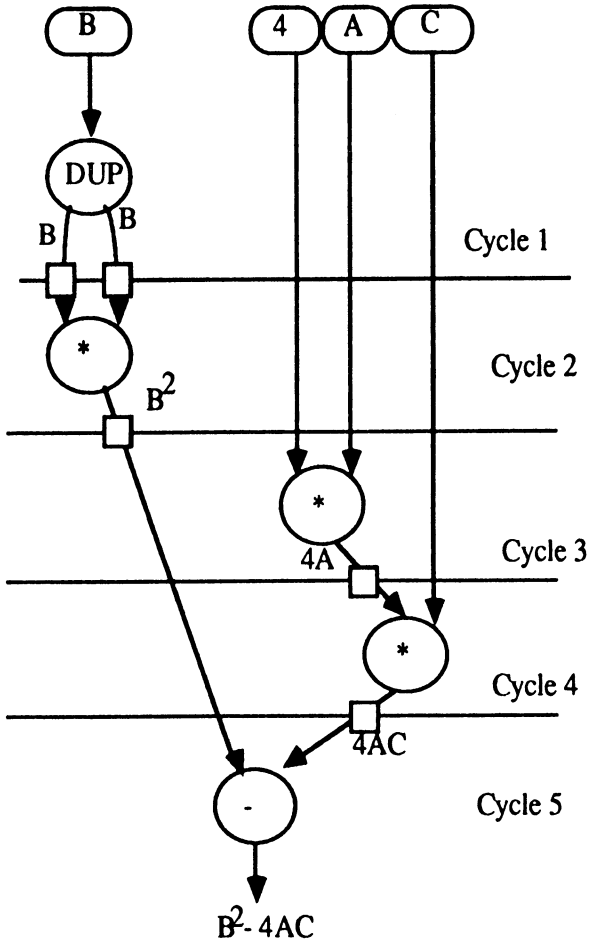


Figure III.6 Graphe de flux de données pour la fonctions $y=b^2-4ac$[NEW86]

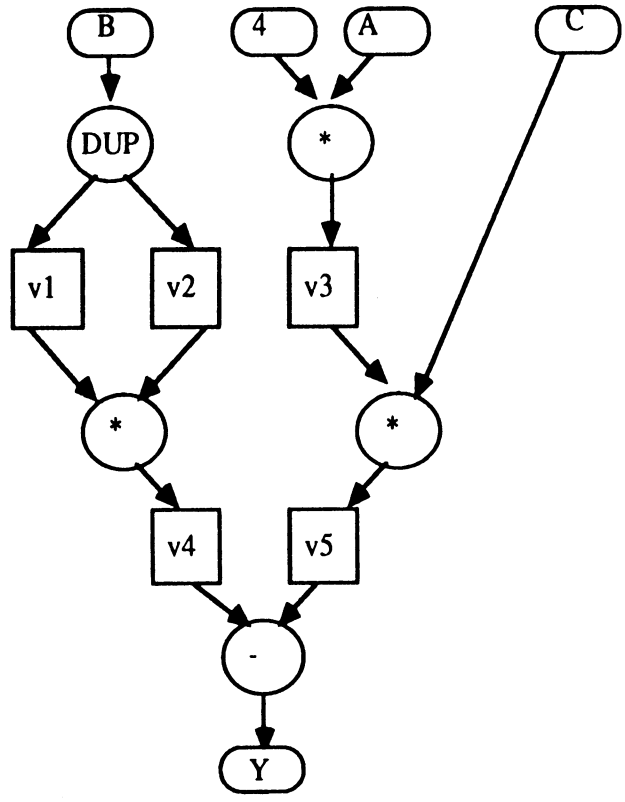
L'algorithme de génération de l'architecture va dépendre du type d'opérateurs utilisés. Dans le cas où l'on utilise des opérateurs auto-synchronisés, la génération de l'architecture peut se faire de manière directe en partant du graphe de flux de données. Par exemple, en remplaçant les fonctions par les unités matérielles correspondantes et les arcs du graphe par des bus. Cette solution ne nécessite pas la génération d'un contrôleur, le circuit étant contrôlé par les flux de données externes. L'utilisation d'opérateurs synchrones implique l'introduction de notions temporelles. Il faut donc affecter chaque opération à une tranche de temps. Cette étape peut être réalisée de manière graphique par le découpage du graphe en tranches correspondant à des tranches de temps (cycle d'horloge ou étape de contrôle). Cette étape, appelée ordonnancement, va permettre la génération d'un automate pour contrôler l'activation des opérateurs et les transferts entre opérateurs. L'exécution séquentielle des opérations va nécessiter la mémorisation dans des registres de toutes les valeurs dont la durée de vie dépasse un cycle. La figure III.7 montre découpage possible de la fonction exemple (figure III.7.a) ainsi qu'une solution pour la réalisation de la partie opérative (figure III.7.b) avec l'automate réalisé par la partie contrôle (figure III.7.c). Les registres v1-v5 ont été introduits pour mémoriser les valeurs intermédiaires (résultat d'évaluation de sous-expressions). Cette solution introduit un cycle 0 pour la lecture des données et suppose que le circuit réalise une boucle infinie de calculs de la fonction y.

Une fois l'ordonnancement fixé on peut réaliser des améliorations de l'architecture, comme dans le cas de la compilation de description impérative. Dans le cas de l'ordonnancement de la figure III.7 le circuit résultat peut ne contenir qu'une seule unité fonctionnelle capable de réaliser la multiplication (opération *). En fait, aucun cycle ne nécessite plus qu'un multiplieur.

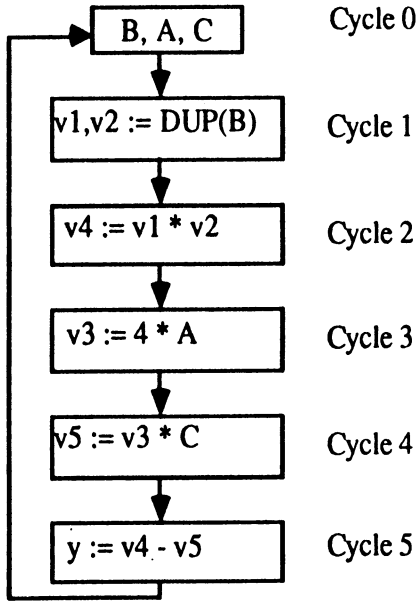


(a) Découpage du calcul de la fonction exemple en tranches de temps (cycles)

□ : Résultat intermédaire



(b) Une solution pour la partie opérative

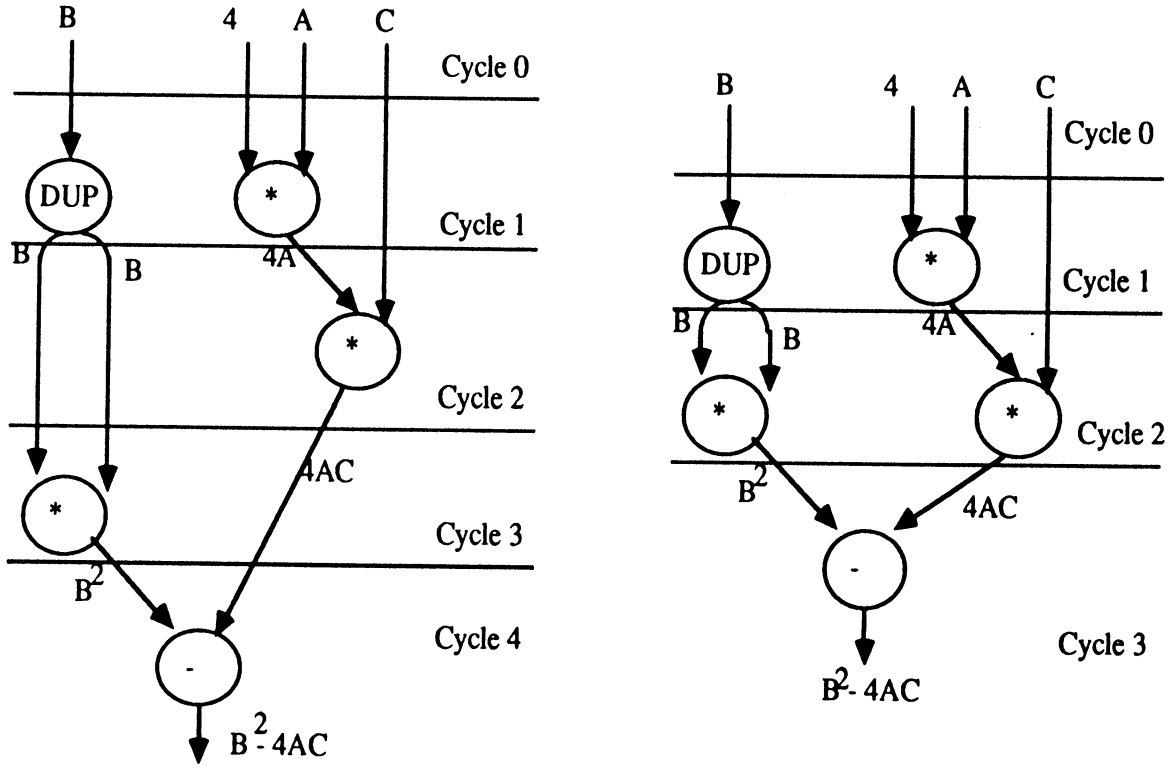


(c) Automate de contrôle

Figure III.7 Compilation de la fonction $y= b^2- 4ac$

L'optimisation peut aussi porter sur l'allocation des registres intermédiaires utilisés. On peut remarquer que dans le cas de la solution donnée par la figure III.7, les deux registres v3 et v5 ne sont jamais "vivants" en même temps, on peut donc les remplacer par un seul registre. Un registre est dit vivant entre le moment de son affectation et celui de sa dernière utilisation [AHO86].

L'ordonnancement peut être réalisé en fonction d'un certain nombre de contraintes. Par exemple, la figure III.8 montre les résultats de deux découpages réalisés en fonction du nombre de multiplieurs. Dans le cas où l'on autorise l'utilisation d'un seul multiplieur (gain de surface), le calcul de la fonction nécessite au minimum 5 cycles (figure III.8.a). Si l'on autorise l'utilisation de deux multiplieurs, le calcul de la fonction peut être ramené à quatre cycles seulement (gain de temps, figure III.8.b). On peut remarquer que quatre cycles est le temps minimum nécessaire au calcul de la fonction dans le cas où chaque opération nécessite un cycle entier. Autrement dit si l'on n'autorise pas deux opérations en séquence dans un cycle, et même si l'on introduit de nouveaux multiplieurs, on ne pourra pas aller plus vite. On peut noter que l'utilisation d'une architecture pipeline permet d'exécuter la fonction en moins de quatre cycles au prix d'une augmentation du nombre de registres.



(a) Découpage pour un seul multiplieur (b) Découpage pour deux multiplieurs

Figure III.8: Optimisation du découpage en fonction du nombre d'opérateurs

Les exemples de compilation décrits plus haut ont pour base plusieurs hypothèses simplificatrices. Pour des raisons d'efficacité et/ou de complexité, la plupart des algorithmes

vus dans cette section ne peuvent être utilisés sur des exemples complexes.

La description du circuit PGCD ne contient ni hiérarchie ni parallélisme. On verra (III.4) que ces deux concepts compliquent la tâche d'un compilateur de silicium.

Par ailleurs, dans le cas de la compilation du circuit de commande, la description ne nécessitait pas de protocole de communication complexe et ne contenait pas de structure de contrôle. Les protocoles de communication sont difficiles à décrire dans les langages fonctionnels car généralement il s'agit de protocoles asynchrones. D'autre part, les structures de contrôle (boucles et instructions de branchement) introduisent des "effets de bords" qui sont difficiles à traiter avec les seules techniques basées sur les graphes de flux de données.

III.3 Description comportementale des circuits

Une description comportementale décrit de manière précise la fonctionnalité d'un système ou d'un circuit sans faire d'hypothèse sur la manière dont il sera réalisé. On oppose souvent ce type de description à la description structurelle, où un circuit (ou un système) est décrit en terme de blocs interconnectés [SOU88]. Il se trouve qu'en pratique on a souvent besoin de mélanger des aspects structurels et des aspects du comportement dans une même description. En général les langages de description du comportement permettent aussi la description de structure.

III.3.1 Description comportementale pour la compilation de silicium

L'utilité de la description comportementale a evolué avec le développement des méthodes de conception et des outils de CAO associés. Au départ, ce type de description a été utilisé pour constituer la documentation de circuits et de systèmes. Le langage ISP [BEL71] par exemple, a été défini dans ce but. Avec le développement des outils d'aide à la conception et en particulier les simulateurs, de nouveaux langages de description ont fait leur apparition [BAR75]. Cette évolution fait que le domaine est assez mûr pour la reconnaissance de standards tel que VHDL [VHD87] et EDIF [MAR87]. Ces deux langages sont, par ailleurs, représentatifs de l'état de l'art dans le domaine des langages de description de matériel.

Le développement des compilateurs de comportement s'est heurté aux sémantiques peu précises des langages de description de matériel [NEW87]. En fait ce qui était suffisant pour décrire et simuler des systèmes et des circuits, ne l'est plus dans le cas de la génération automatique.

Un langage est défini par sa syntaxe (ce qu'on écrit) et sa sémantique (ce qu'on exprime). Faute de description formelle, la sémantique des langages de description de matériel est généralement définie par la manière dont une description sera simulée. La description d'un circuit revient donc à exprimer son comportement (ou sa structure) en fonction du simulateur et non en utilisant des concepts standards. Il en sera de même pour les compilateurs de silicium, sauf que ces derniers sont beaucoup moins souples que les simulateurs et que leur langage d'entrée est beaucoup plus limité que celui accepté par les outils de simulation. La souplesse des simulateurs vient du fait que:

- Ils fonctionnent généralement comme des interpréteurs. Ils sont donc capables de traiter des informations dynamiques. Ils permettent aussi de mélanger des instructions de description de circuits et des instructions de commande du simulateur.
- Ils sont capables de manipuler des structures de données complexes telles que celles manipulées par les langages de programmation (les listes et les nombres flottants par exemple).

Les compilateurs de comportement actuels ne peuvent donc traiter qu'un sous ensemble d'un langage de description de matériel accepté par les simulateurs.

III.3.2 Eléments d'une description comportementale

Un compilateur de comportement réalise le découpage d'un système en sous-systèmes. Au plus haut niveau, le système d'entrée est vu de l'extérieur comme un bloc indivisible, défini par une interface et une fonction. La communication avec l'extérieur se fait via l'interface et selon des protocoles définis dans la fonction. Toute modification de l'interface ou de la fonction constitue une redéfinition du circuit.

Un circuit peut être décrit de manière hiérarchique. Dans le cas d'une hiérarchie structurelle, chaque bloc correspond à une portion du circuit dont la réalisation est connue ou "prédictible" [MAR86]. Dans le cas de la description comportementale, un bloc ou un module décrit une sous-fonction du circuit, qui ne correspond pas forcément à une portion de matériel. En réalité on a aussi besoin d'utiliser des blocs dont la structure a déjà été définie telle que celle d'un opérateur non standard du système. La communication entre les blocs se fait à travers des variables globales et des interfaces qui décrivent les canaux de communication, dans le cas d'un module comportemental et les signaux d'interconnexion dans le cas d'un bloc structurel. L'utilisation de variables globales pour la communication entre les différents modules n'est pas autorisée par tous les langages. Ceci vient du fait que l'utilisation des variables globales se fait par "effet de bord". Certains langages, comme VHDL, interdisent tout "effet de bord". Dans ce cas, le découpage de la description comportementale correspond à un découpage structurel. Dans le cas du langage VHDL, chaque module du comportement (appelé processus) définit un sous-système indépendant.

III.3.3 Les langages de description du comportement

Le comportement d'un circuit peut être décrit en termes d'actions ou de fonctions. Dans le premier cas, un circuit réalise un certain nombre d'actions dans un ordre donné. Dans le second cas, un circuit réalise une fonction qui produit des sorties dépendantes des entrées. Cette fonction peut être décomposée en fonctions plus élémentaires. Cette description fonctionnelle appelée aussi parallèle (ou description de flux de données) spécifie les flux de données dans le circuit.

Ces deux types de descriptions définissent deux grandes familles de langages de description qui sont les langages impératifs et les langages fonctionnels ou parallèles. Une classification plus complète et plus détaillée des langages utilisés pour la description des circuits intégrés se trouve dans [BAR75], [BOR81], [CRA85], [MAR86].

On s'intéressera uniquement aux premiers. En fait, la plupart des compilateurs de comportement partent d'une description impérative, même dans le cas où le compilateur génère des circuits parallèles.

D'un point de vue syntaxique, les langages impératifs pour la description de comportement sont, à peu de choses près, similaires aux langages de programmation du même type. Les premiers imposent généralement des restrictions sur la nature et la complexité des structures de données manipulées. D'autre part, ils permettent de décrire des actions parallèles et des protocoles de communication entre processeurs. D'un point de vue sémantique, les langages de description de comportement présentent plusieurs particularités qui seront détaillées ci-après.

III.3.4 L'expression des flux de contrôle

Dans le cas d'une description impérative, les flux de contrôle définissent l'ordre d'exécution des opérations. Ces flux peuvent être exprimés par :

- l'ordre lexical des opérations (ordre d'apparition dans la description),
- des structures de contrôle permettant l'exécution en série ou en parallèle d'une liste d'opérations.
- les constructions conditionnelles et les constructions d'itérations,
- les instructions de branchement simple et appels de "sous- programmes"

L'ordre lexical n'est pas toujours utilisé pour définir des séquences d'opérations : dans le cas du langage ISPS [BAR81], les instructions consécutives sont, par défaut, supposées s'exécuter en parallèle, par contre, les séquences d'opérations sont déclarées de manière explicite.

La plupart des langages de description du type impératif utilisent des constructions spéciales pour décrire des blocs d'opérations pouvant s'exécuter en parallèle et/ou des blocs d'opérations devant s'exécuter en séquence. Il faut remarquer qu'il est important de pouvoir décrire des blocs d'opérations parallèles, même dans le cas d'une description impérative. Comme la description d'entrée est généralement utilisée pour produire une première solution architecturale, une déclaration implicite du parallélisme permet d'influencer cette solution. D'autre part, il est souvent difficile d'extraire le parallélisme d'une description purement séquentielle (voir III.9)

Pour que la description reste "compilable", on est généralement amené à imposer des restrictions sur l'utilisation de ces constructions.
Exemple : dans le cas où le but est de générer une machine séquentielle, comment ordonner les opérations de deux séquences pouvant s'exécuter en parallèle. Un tel cas peut se décrire de la manière suivante:

```
debut-parallèle
   debut-série
      I1, I2 ... In
   fin
   debut-série
      J1, J2, Jk
   fin
fin
```

où I1, I2, ... In et J1-Jk sont des instructions dont la durée peut être variable.

Cette description doit être transformée en une série de cycles (ou de micro-instructions), chacun pouvant contenir plusieurs instructions appartenant aux deux séquences. Le problème consiste à éviter une explosion du nombre de cycles. En fait la transformation revient a remplacer un réseaux de deux machines d'états finis par une machine d'état fini équivalente. Cette transformation est triviale dans le cas où les deux blocs ne contiennent ni instructions de branchement ni instructions conditionnelles.

La sémantique des constructions conditionnelles doit aussi être définie avec précaution car les différentes alternatives d'une instruction conditionnelle peuvent être évaluées en séquence ou en parallèle.

Exemple: L'expression

```
si X=0 alors  A:=B sinon A:=C finsi;
```

équivalente à:

```
debut
   si X=0 alors A:=B finsi ;
   si non (X=0) alors A:=C finsi ;
fin
```

peut être interprétée soit comme un bloc de deux instructions parallèles, tel que c'est le cas dans le sous ensemble de LDS [LAU85] utilisé par SYCO (voir chapitre IV) ou encore une séquence de deux instructions telle que c'est le cas dans le langage HardwareC [KU88]. La même remarque peut s'appliquer aux instructions de choix (du type 'case of'). Selon la manière dont elles sont interprétées, les instructions conditionnelles vont constituer des blocs d'opérations parallèles ou séquentielles. Les restrictions sur les imbrications des blocs séries/parallèles doivent être appliquées pour l'imbrication des instructions conditionnelles.

Les constructions d'itérations (boucles) ne posent pas de problème particulier dès lors qu'elles peuvent être décomposées en une série d'instructions conditionnelles utilisant des branchements. Pour faciliter une analyse des flux de données, en vue de l'optimisation de la description comportementale, plusieurs langages de description de comportement réglementent l'utilisation des boucles. La principale restriction, souvent utilisée aussi par les langages comme Pascal [AHO87], consiste à ne permettre que des boucles à une seule entrée et une seule sortie.

Pour les mêmes raisons, les langages de description de comportement limitent aussi l'utilisation des branchements (voir III.9).

III.3.5 Description hiérarchique

Une description hiérarchique fait appel à des constructions équivalentes aux sous-programmes utilisés dans les langages de programmation. Elle permet de décrire le contrôle de manière hiérarchique. Vu leur caractère dynamique, les variables locales et les paramètres formels des sous-programmes sont difficiles à gérer. Dans le cas où le but est de générer un circuit qui contient une seule partie opérative, les variables locales et les paramètres des sous-programmes doivent être transformés en variables globales. Les décisions de partage d'une variable globale entre plusieurs variables locales sont prises à la suite d'une analyse des flux de contrôle de la description. La sémantique des appels de sous-programmes dépend du choix de l'architecture qui sera générée par le compilateur.

Dans le cas où l'architecture générée par le compilateur ne supporte pas la hiérarchie, la description est mise à plat. D'autre part, la hiérarchie initiale est généralement modifiée durant le processus de compilation pour optimiser le résultat de la compilation (voir III.5).

III.3.6 Description des protocoles de communication

Le langage de description du comportement doit permettre la spécification des protocoles de communications. Ces derniers permettent au circuit de réaliser des opérations d'entrée/sortie et plus généralement de communiquer avec d'autres circuits. Chaque protocole doit être réalisé d'une manière bien définie afin de permettre au circuit de dialoguer avec l'extérieur. Dans ce cas précis, le compilateur de silicium peut ne pas avoir le choix de la manière dont ces protocoles seront réalisés. Ce cas est à distinguer des autres fonctions du circuit, pour lesquelles c'est le compilateur qui est supposé choisir la manière dont seront réalisées les différentes fonctions.

Un protocole de communication est décrit par un ensemble de procédures qu'on appellera procédures de communication. Chaque procédure décrit un ensemble d'opérations à réaliser dans un ordre donné. Ces procédures peuvent être immergées dans le reste de la description ou structurées dans des primitives de communication [PAR81].

Dans le premier cas, les opérations de communication sont synchronisées par l'intermédiaire des variables externes (signaux et variables directement accessibles de l'extérieur). Ces variables externes sont généralement séparées en deux catégories, les variables de donnée et les variables de contrôle. Une procédure de communication est une séquence d'opérations qui utilise ces variables externes. Dans le second cas, les procédures de communication de base sont prédéfinies et encapsulées dans des primitives. La lecture d'une

donnée dans une mémoire externe, par exemple, peut constituer une primitive.

L'utilisation de primitives spéciales pour décrire les protocoles de communication permet d'isoler facilement les procédures de communication du reste de la description comportementale, ce qui peut par la suite, simplifier le traitement du compilateur de silicium. Par contre, l'utilisation des primitives limite les possibilités de communication car, dans ce cas, on ne peut décrire que les protocoles pouvant être spécifiés en termes de ces primitives.

L'utilisation directe des variables externes pour la communication permet de décrire des procédures de communication "sur mesure". Par contre, si le langage d'entrée n'impose pas certaines restrictions, il sera possible de décrire des protocoles de communication qui ne soient pas réalisables. D'autre part le compilateur de silicium doit être capable de localiser les procédures de communication pour éviter de les modifier durant les différentes étapes de compilation.

Les deux styles de description des protocoles de communication peuvent être mélangés dans un même langage [VHD87] [KU88]. Il existe des systèmes qui permettent la description des protocoles de communication à l'aide de diagrammes de temps [BOR88]. Ces diagrammes sont difficiles à intégrer dans la description comportementale durant le processus de compilation.

Une procédure de communication peut être synchrone ou asynchrone. Dans les deux cas, elle est constituée par un ensemble d'opérations ordonnées. Une procédure pour la communication synchrone ne contient pas d'opération ou de boucle d'attente d'événement. Par contre la durée des opérations et les délais entre les opérations sont connus à l'avance. La description d'un tel protocole oblige l'utilisateur à faire des hypothèses sur le processus de compilation. Par exemple, pour décrire une procédure de communication qui nécessite un délai entre deux opérations, il faut pouvoir exprimer ce délai. La plupart des langages de description de comportement permettent de décrire des délais. Mais, quelle est la signification d'un délai de 175ns, par exemple, pour le compilateur de silicium? Les 175ns peuvent être converties en K cycles (K= 175ns /temps_de_cycle) dès que le temps de cycle devient connu [CAM88]. Dans ce cas il faut que le temps de cycle soit inférieur à 175ns, et que l'erreur d'arrondi due à la division soit non significative.

Dans le cas d'un protocole de communication asynchrone, les délais entre les opérations peuvent être variables et non connus à l'avance. Les opérations sont synchronisées sur des événements tels que le changement de la valeur d'une variable. Les délais peuvent être exprimés en terme de boucles d'attente. Dans le cas où les procédures de communication sont réalisées par des primitives prédéfinies il faut aussi prévoir des primitives d'attente.

Enfin il faut noter que les protocoles de communication peuvent mettre en œuvre des organes de communication complexes. La compilation de processus concurrents décrits en

HardwareC [KU88], par exemple, entraîne la génération d'unités spéciales pour la communication [DEM88].

III.3.7 Utilisation des fonctions externes

Les langages de description de comportement sont aussi composés d'un environnement de base, qui est fixe, et d'une partie modifiable comme dans le cas des langages de programmation [BAC78]. La partie changeable est constituée par l'ensemble des fonctions externes qui peuvent être utilisées par le langage sans que le comportement de ces fonctions soit décrit dans le langage. Ces fonctions sont généralement définies après le langage lui-même. Elle sont à distinguer des fonctions spécifiées à l'aide du langage de description de comportement, ces dernières étant destinées à être compilées comme le reste de la description.

Une fonction externe correspond à une unité capable d'exécuter un ensemble donnée d'opérations. Dans le cas où l'ensemble est réduit à une seule opération, l'unité est dite mono-fonction, sinon elle est dite multi- fonctions. Les caractéristiques matérielles de cette unité doivent être connues au moment de la compilation.

La généralisation de l'utilisation des fonctions externes permet aussi au compilateur d'être indépendant de la technologie. Le système DAA [KOW85], généralise la notion d'unité fonctionnelle aux autres éléments matériels. Il considère que tous les éléments de mémorisation (registres, constantes, mémoires) et les éléments de communication (multiplexeurs et bus) peuvent être décrits par des boîtes génériques. Dans le cas du système DAA, la bibliothèque de fonctions externes définit la base de données technologiques. Cette approche permet d'écrire des compilateurs de comportement qui sont complètement indépendants de la technologie.

Les fonctions externes constituent en quelque sorte une extension du langage. Un compilateur de comportement peut être vu comme un système qui produit une architecture en partant d'une description comportementale et d'une bibliothèque d'unités fonctionnelles prédéfinies (voir figure III.9). Ce schéma a aussi été proposé par [SAK86] et [DEN88].

Chaque fonction externe correspond à une unité appellée aussi boîte à opérations [JAM86], [JAM86a]. Une unité arithmétique et logique (UAL), par exemple, est une boîte à opérations capable d'effectuer des opérations arithmétiques (-, +, +1, ...) et des opérations logiques (et, ou, ouex, ...).

La figure III.10 montre une boîte à opérations génériques. Cette boîte communique avec l'extérieur via trois types de signaux, les signaux de type données permettant de véhiculer les opérandes et les résultats de l'opération. Cette boîte à opérations est capable de réaliser des opérations n-aire et peut délivrer plusieurs résultats (k). Le signal de commande permet de

sélectionner l'opération à exécuter. Le troisième signal de communication est de type paramètre, il permet de spécifier des informations de contrôle. Dans le cas d'une boîte à opérations capable d'effectuer des opérations sur des données de longueurs variables, le champ paramètres peut être utilisé pour spécifier la longueur des données.

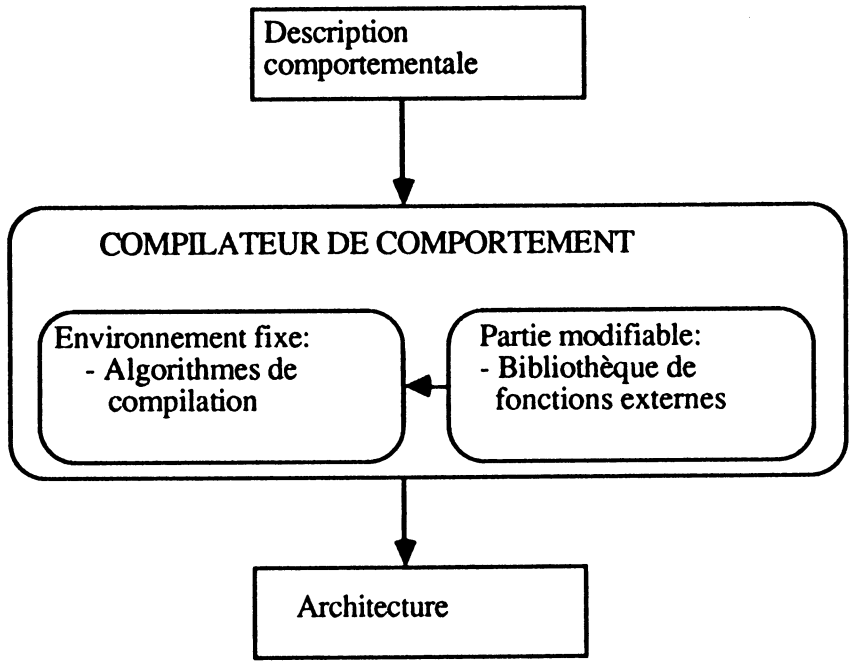


Figure III.9: Les fonctions externes dans un compilateur de silicium.

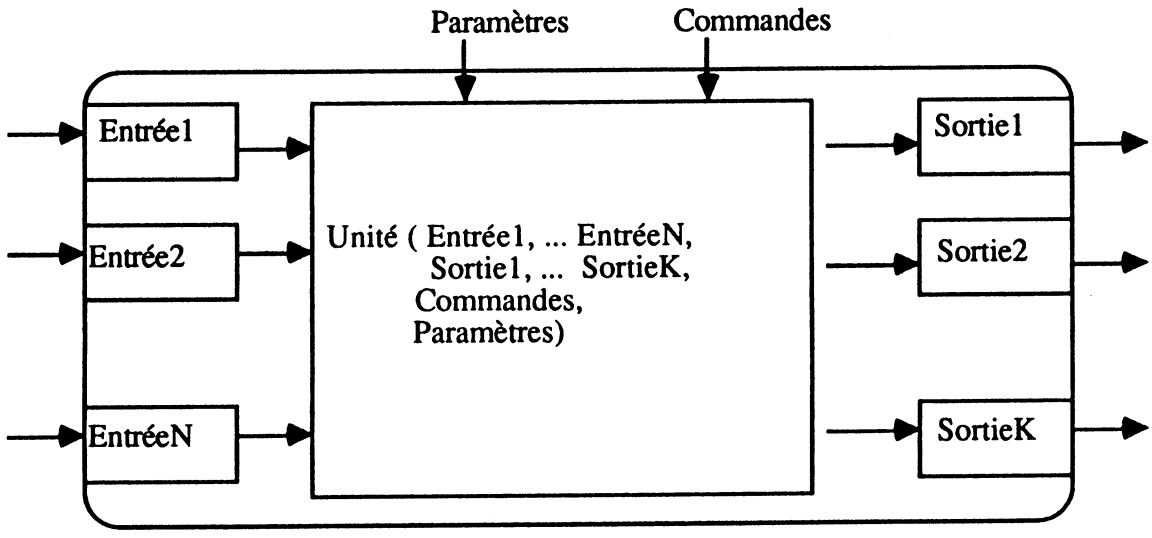


Figure III.10 Unité fonctionnelle générique

La notion d'unité fonctionnelle générique est similaire à la notion d'opérateur générique utilisée par Denyer [DEN85] pour décrire des opérateurs bit-série. Le schéma de Denyer contient un quatrième type de signaux utilisés pour la synchronisation. Avec ce modèle

générique, une UAL peut être représentée par une boîte à opérations à deux entrées (les deux opérandes) et deux sorties (le résultat de l'opération et les indicateurs arithmétiques). Le code opération de l'UAL est spécifié par le champ de commandes. L'UAL est montrée en la figure III.11, elle contient aussi un champ paramètres permettant de spécifier la structure des opérandes (un bit, 8 bits, 16 bits).

Cette représentation de l'UAL standard permet de résoudre le problème de la récupération des indicateurs arithmétiques (flags: zéro, carry, overflow et négatif). Ce problème est détaillé dans [JAM86] et [JAM86a]. Il résulte du fait que:

- la compilation d'une description peut générer un circuit qui contient plusieurs opérateurs,
- les indicateurs sont utiles dans la description comportementale pour contrôler le séquencement de certaines opérations,
- étant donné qu'au moment de la description, donc avant l'allocation de matériel, on ne connaît ni le nombre d'unités ni l'unité qui va exécuter chacune des opérations, on ne sait donc pas adresser (ou désigner) les indicateurs.

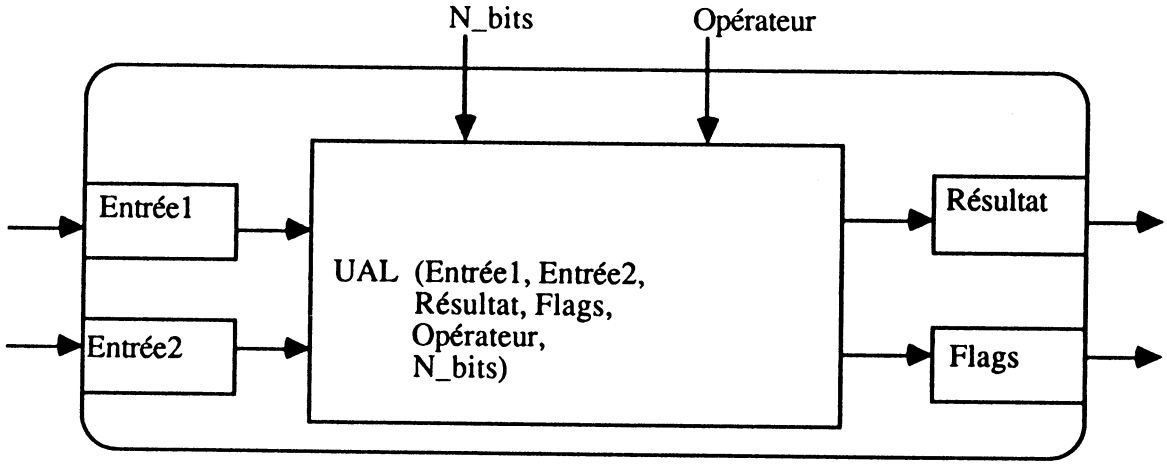


Figure III.11: Représentation de l'unité arithmétique et logique standard

L'utilisation des fonctions externes dans une description peut se faire soit en utilisant des appels de fonctions, soit en utilisant les opérateurs standards. Par exemple l'utilisation de la boîte à opérations de la figure III.11 peut se faire par des instructions du genre:

```
ALU(Opérande1, Opérande2, Résultat, Flags, Opération, N_bits)
```

Si on ignore les paramètres Flags (indicateurs arithmétiques) et N_bits (la longueur des données), cet appel est équivalent à l'écriture infixée suivante:

```
Résultat := Opérande1 Opération Opérande2
```

Dans le cas de l'appel fonctionnel, en plus de l'opération, les indicateurs sont stockés dans le registre Flags.

La représentation et l'utilisation des fonctions seront traitées plus en détail en IV.8. Dans le cas de la compilation de circuits séquentiels, les boîtes à opérations seront limitées à des unités exécutant des opérations en un nombre de cycles fini et préalablement connu. En fait comme la compilation consiste essentiellement à découper la description en cycles de base, il faut pouvoir découper de manière statique chaque opération qui nécessite plusieurs cycles en micro-opérations de un cycle chacune. Ce découpage suppose que l'on connaît à l'avance le nombre de cycles nécessaires à chaque opération.

III.4 Formes intermédiaires pour la compilation de comportement

Le processus de compilation comporte plusieurs étapes, chacune permettant de fixer de nouveaux aspects du circuit. Certaines de ces étapes sont inter-dépendantes et nécessitent donc plusieurs passages. Pour manipuler cet aspect évolutif, les compilateurs de comportement utilisent généralement une forme intermédiaire pour représenter de manière interne le circuit durant les étapes de compilation.

L'utilisation d'une forme intermédiaire permet aussi :

• d'éliminer les dépendances dues à la syntaxe du langage. Par exemple, la forme intermédiaire peut ne pas contenir l'ordre dans lequel ont été écrites les instructions qui doivent se dérouler en parallèle,

• de faciliter les étapes de compilation. La représentation interne peut être adaptée aux algorithmes de compilation. En pratique, les formes intermédiaires sont fortement biaisées par les algorithmes de compilation.

• de représenter plusieurs solutions architecturales possibles pour une seule description. Elle permet aussi des optimisations indépendantes de l'architecture du circuit.

La forme intermédiaire est une représentation qui doit enregistrer l'évolution du processus de compilation. Ainsi, la forme intermédiaire idéale doit représenter aussi bien la description comportementale que l'architecture correspondante. Au départ, elle représente un codage simple d'une description de circuit dans le langage d'entrée du compilateur. Elle est ensuite enrichie et transformée par les différentes étapes de la compilation. A la fin de la compilation, la forme intermédiaire représente l'architecture du circuit. Or une telle forme intermédiaire ne peut être réalisée qu'au prix de plusieurs restrictions au niveau de l'architecture et/ou du langage de description.

En pratique, on distingue deux types de formes intermédiaires. Les premières utilisent les arbres syntaxiques enrichis. On les appellera désormais forme intermédiaire syntaxique. Elles sont mieux adaptées à la représentation d'entrée qu'à la représentation de l'architecture. Les formes du second type utilisent des graphes. Elle seront appelées formes intermédiaires graphiques. Dans ce cas, le circuit est représenté par un ou plusieurs graphes décrivant les flux de données et les flux de contrôle dans le circuit. Elle sont bien adaptées à la représentation de l'architecture. Mais la conversion d'une description comportementale en graphes, entraîne généralement une perte d'informations.

Les formes intermédiaires graphiques et syntaxiques constituent les principales structures de données utilisées par les compilateurs de comportement. Il en existe d'autres: les GRAFCETs par exemple, ont aussi été utilisés comme formes intermédiaires pour la compilation de silicium [EZZ86]

III.4.1 Formes intermédiaires syntaxiques

Une forme intermédiaire syntaxique peut être construite facilement à partir de la description comportementale [AHO87]. A chaque action de la description (opération sur les données ou opération de contrôle) on fait correspondre une structure dans la forme intermédiaire. L'avantage principal de ce type de représentation est la lisibilité. En fait il est généralement facile d'obtenir une forme lisible par "décompilation". Pour sauvegarder cet avantage jusqu'à un stade avancé du processus de compilation, les transformations effectuées par les étapes de compilation ne doivent pas introduire d'opérations non "décompilables". Ceci est possible dans le cas des étapes de partition et d'ordonnancement, puisque la partition ne fait que modifier la hiérarchie initiale de la description, et l'ordonnancement ne fait que réarranger l'ordre des opérations. Par contre, le résultat de l'allocation, la partie opérative, peut difficilement être intégré dans une telle structure. D'autre part, les algorithmes d'allocation nécessitent des structures plus complexes pour être efficaces.

Le fait de pouvoir garder une structure lisible, à un stade avancé de la compilation, permet à l'utilisateur d'influencer le processus de compilation, soit par l'injection de nouvelles contraintes entre les étapes de compilation, soit par la modification manuelle de la description initiale. Par exemple, l'utilisateur peut décider de réécrire de manière plus fine et plus efficace la partie de la description qu'il juge critique et que le compilateur n'a pas pu traiter de manière efficace. Ce schéma est le même que celui utilisé dans le cas de l'écriture d'un programme : on peut être amené à optimiser manuellement la partie critique d'un programme.

La correspondance entre la description comportementale et la forme intermédiaire constitue un point déterminant pour la génération de contrôleurs complexes. Ainsi, les compilateurs qui permettent la génération de contrôleurs complexes tels que MIMOLA [ZIM79], [MAR79] ont choisis la forme intermédiaire syntaxique.

III.4.2 Formes intermédiaires graphiques

Les formes intermédiaires graphiques utilisent trois types de graphes pour représenter les flux de données, les flux de contrôle, un mélange des flux de données et des flux de contrôle.

Dans un graphe de flux de données, les sommets représentent des opérateurs et les arcs représentent des valeurs. L'ordre d'exécution des opérations est défini par le flux de données. Un opérateur est mis à feu (activé) dès que ses entrées sont prêtes. La figure III.6 donne l'exemple d'un graphe de flux de données.

Un graphe de flux de contrôle est un graphe de précédence. Chaque sommet de ce

graphe renferme un ensemble d'opérations pouvant s'exécuter en parallèle. Les arcs du graphe de contrôle définissent l'ordre d'exécution des opérations : l'arc du sommet s1 vers le sommet s2 indique que les opérations du sommet s1 doivent être exécutées avant celles du sommet s2. La figure III.5 montre un graphe de flux de contrôle.

Le troisième type de graphes utilisé, est un graphe mixte permettant de représenter à la fois les flux de données et les flux de contrôle.

Trois techniques ont été proposées pour représenter ce type de graphes mixtes, appelés aussi graphes de flux de contrôle et de données. La première utilise des graphes de flux similaires à ceux utilisés pour l'optimisation des langages de programmation [AHO87]. La seconde est basée sur un graphe avec deux types de sommets, pour représenter les opérateurs et les variables, et deux types d'arcs, pour représenter les valeurs et les relations de précédence [CAM85]. La dernière utilise un graphe mixte avec des sous-graphes de contrôle et des sous-graphes de flux de données, ainsi que des sommets relais permettant de passer d'une partie à l'autre [GAJ86].

La représentation basée sur les graphes de flux a été de loin la plus étudiée. Dans un graphe de flux, les sommets représentent des séquences d'opérations linéaires appelées aussi blocs de base. Les arcs du graphe décrivent les flux de contrôle. La figure III.12 montre la représentation de la fonction PGCD par un graphe de flux.

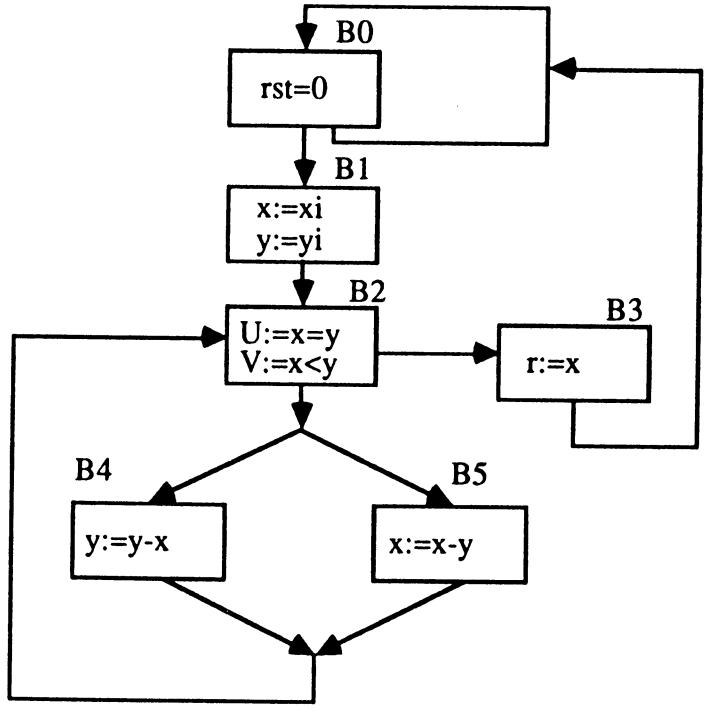


Figure III.12 Exemple de Graphe de flux (la fonction PGCD)

Les blocs de base peuvent être représentés par un graphe orienté acyclique, qu'on appellera DAG ("Directed Acyclic Graph") qui décrit les flux de données dans ce bloc de base. Cette représentation est intrinsèquement limitée à des descriptions non hiérarchiques, elle a été utilisée pour la compilation de silicium par FLAMEL [TRI85]. La représentation VT ("Value Trace") du système CMU-DA [THO81], [WAL83], [SNO78] étend cette représentation aux descriptions hiérarchiques. Elle utilise des opérateurs de contrôle dans les blocs de base. Parmi les opérateurs autorisés dans un bloc de base (appelé VT-bloc) on trouve l'opérateur SELECT (instruction choix), l'opérateur CALL (appel de procédure) et des opérateurs pour décrire les entrées/sorties (read/write). Un VT-bloc peut regrouper plusieurs blocs de base de FLAMEL.

La génération de la forme intermédiaire graphique entraîne généralement une perte des informations contenues dans la description originale. Il est donc difficile de maintenir une correspondance entre la description d'entrée et l'architecture. D'autre part, ce type de représentation est illisible. Par contre, la forme intermédiaire graphique est bien adaptée pour la représentation de l'architecture. Elle permet de représenter des contrôleurs, mais elle est surtout bien adaptée pour la représentation du parallélisme. Ce dernier point fait que la plupart des compilateurs de comportement orientés vers les circuits parallèles tels que DAA [KOW85a], FLAMEL [TRI86], CMU-DA (deuxième génération) [THO83], ... , utilisent des formes intermédiaires graphiques.

III.5 Optimisation de la description comportementale

L'optimisation de la description comportementale se fait par l'application de transformations permettant d'améliorer le résultat éventuel de la compilation. Le terme optimisation sera utilisé bien qu'il s'agisse plutôt d'amélioration.

Les transformations appliquées sont similaires à celles utilisées pour le compactage de code par les compilateurs de langages de programmation [AHO87]. Cette section sera limitée à l'étude des critères d'optimisation spécifiques à la compilation de silicium. Pour ce qui est des transformations de la description comportementale elles ont été largement étudiées dans la littérature [WAL83] [SNO83] [THO83] [TRI85] [BEK87 [WAL88].

L'optimisation de la description comportementale peut être indépendante de l'architecture qui sera générée. Elle doit être distinguée des autres actions d'optimisation liées aux différentes étapes de la compilation qui, elles, tiennent compte de l'architecture et effectuent plutôt des transformations architecturales. Par contre, dans le cas où il existe une certaine correspondance entre la description comportementale et le modèle architectural utilisé par le compilateur, il est possible de définir des transformations de la description comportementale qui engendrent des modifications prévisibles de l'architecture.

III.5.1 Critères d'optimisation

Dans le cas des langages de programmation, les transformations de programmes sont définies en fonction de trois critères [AHO87] :

- une transformation doit préserver la fonction réalisée par le programme,
- l'effet d'une transformation doit être mesurable,
- l'effort nécessaire pour la réalisation de la transformation doit se justifier par le gain qu'elle doit entraîner.

Ces trois critères restent valables dans le cas des transformations d'une description comportementale. Mais l'application des deux premiers critères est plus difficile.

Pour vérifier qu'une transformation n'altère pas la fonction d'une description comportementale, on a souvent besoin du contexte d'utilisation de cette description. Par exemple : les procédures décrivant les protocoles de communication, contiennent souvent des instructions d'attente. Ces instructions, inutiles en apparence, correspondent à ce que doivent éliminer certaines transformations. Pour éviter que les transformations ne changent les protocoles de communication, ou d'autres parties critiques de la description, il faut pouvoir spécifier des zones de la description interdites aux transformations.

Le second critère concerne le gain moyen en performances associé à une transformation. Dans le cas d'un programme, la vitesse est généralement le seul critère de

performance pris en compte. Dans le cas des circuits intégrés, on distingue au moins trois facteurs importants pour calculer les performances : la vitesse, la consommation et la surface. Ces trois facteurs sont difficiles à estimer en partant d'une description comportementale. La consommation est étroitement liée à l'organisation interne d'un circuit, et ne peut donc être prise en compte à ce stade. La vitesse et la surface sont également difficiles à estimer dans le cas général. La vitesse, par exemple, est le produit du nombre de cycles par la durée d'un cycle. Seul le nombre de cycles peut être pris en compte au niveau de la description comportementale. La durée du cycle de base dépend de la réalisation matérielle du circuit. Or une transformation qui diminue le nombre de cycles peut entraîner une augmentation du temps de cycle et par la suite dégrader la vitesse.

III.5.2 Transformations de la description comportementale

Les transformations sont appliquées sur la forme intermédiaire. Plusieurs réalisations ont été décrites pour des formes intermédiaires graphiques [WAL83] [TRI85] [WAL88] et pour des formes intermédiaires syntaxiques [BHA87] [BEK87].

Généralement, on distingue les transformations locales et les transformations globales. Les premières sont basées sur une analyse de flux de données qui limite leur portée à l'équivalent d'un bloc de base dans la forme intermédiaire. Elles réalisent des transformations d'opérations, telles que décomposer une opération complexe en une série d'opérations plus simples. Les transformations locales permettent de réorganiser les opérations contenues dans les instructions de choix, telles que isoler les opérations communes à toutes les branches d'une instruction de sélection (du type "case of" ou "select") et les placer en amont ou en aval de cette instruction [WAL83] [WAL88].

Les transformations globales nécessitent en plus une analyse des flux de contrôle. Elles permettent de réaliser des opérations inter-blocs comme, par exemple, le déroulement des boucles ou l'expansion de l'appel des procédures. Elles permettent aussi la détection des opérations inutiles et la détermination de la durée de vie des variables [AHO87].

Comme il a été dit plus haut, la portée de ces transformations en tant que telles se trouve réduite du fait qu'elles ne peuvent être appliquées à toute la description et par le manque des techniques permettant d'estimer avec précision leurs effets. Le fait que ces optimisations sont indépendantes de l'architecture limite également leur utilisation.

III.5.3 Optimisation de l'architecture via la description comportementale

Dans ce cas, il s'agit de modifier la description comportementale pour influencer le résultat de la compilation. Ce genre d'optimisation n'est possible que dans le cas où il existe une correspondance entre la description comportementale et l'architecture générée. Cette

correspondance ne peut être directe, auquel cas il s'agirait plutôt de description structurelle. Les systèmes ArchitectWorkBench et SYCO utilisent ce type d'optimisation. Dans ce cas aussi le terme optimisation est utilisé de manière abusive, car il s'agit plutôt d'une recherche de compromis surface/vitesse. Comme il a été remarqué au chapitre précédent, les facteurs de performances sont généralement interdépendants. L'amélioration de l'un des deux facteurs entraîne le plus souvent la dégradation de l'autre. Là aussi on se limitera aux facteurs de vitesse et de surface.

Le système ArchitectWorkBench [WAL88] utilise une architecture qui met en œuvre des processus pouvant fonctionner en parallèle et/ou organisés en pipe-line. Ce système fournit principalement deux commandes pour modifier la description en vue d'une transformation de l'architecture. La première autorise la création de nouveaux processus, et la seconde permet d'introduire un nouvel étage dans un tuyau ("pipe"). Le système SHEWA [PAR87] permet aussi de modifier une architecture pipe-line. Les transformations qu'il propose ne peuvent pas être modélisées par des modifications de la description comportementale car il travaille sur une description fine de l'architecture.

Dans le cas du compilateur Syco, par exemple, la hiérarchie et le parallélisme de la description comportementale définissent la hiérarchie et le parallélisme de l'architecture générée [JER86]. La recherche de compromis surface/vitesse est réalisée par le système ARTS [BEK87], qui permet de modifier le parallélisme et la hiérarchie. En règle générale, pour aller plus vite il faut augmenter le parallélisme et/ou diminuer la hiérarchie, pour diminuer la surface il faut diminuer le parallélisme et/ou augmenter la hiérarchie. Ce modèle d'optimisation sera commenté avec plus de détails plus loin (IV.9).

III.6 Partition ou découpage de description comportementale

Dans les exemples de la section III.2, on supposait que la fonction à compiler était réalisable sur un seul circuit, composé d'une partie opérative et d'une partie contrôle. En pratique, l'architecture externe d'un circuit résulte généralement d'un découpage fonctionnel de tout un système. Ce découpage sera aussi appelé partition ("partitionning"). D'autre part, un même circuit peut être composé de plusieurs sous-systèmes eux même composés de parties opératives et/ou de parties contrôle. Les différents cas de partition emploient des techniques différentes. On distinguera le cas de la partition d'une description en plusieurs sous-systèmes destinés à être réalisés sur un ou plusieurs circuits de celui de la partition en une partie opérative et une partie contrôle.

III.61 Partition en plusieurs sous-systèmes

Comme il a été exposé au second chapitre, le problème de partition est difficile à résoudre dans le cas général. Il n'existe pas de méthodologie de découpage qui soit générale et suffisamment bien formalisée pour être automatisable [TRE87]. Le processus de découpage d'un système en sous-systèmes doit tenir compte de plusieurs facteurs difficiles à mesurer. Dans [COX84], l'auteur distingue six facteurs pour guider le processus de partition (ou de découpage) :

-1- La flexibilité : chaque décision doit prévoir la possibilité de modifications ultérieures. Il faut donc retarder au maximum les décisions bloquantes telles que la définition précise de protocoles de communication.

-2- "Qui contrôle quoi" : tenir compte des relations entre les différentes fonctions du système.

-3- Utilisation du système : tenir compte de la future utilisation du système et avantager le cas d'utilisation le plus important.

-4- Vitesse : essayer de produire un système qui soit le plus rapide possible tout en tenant compte des autres contraintes. La vitesse doit être calculée de manière globale et non au niveau de chaque sous-système.

-5- Communication : le choix des protocoles doit tenir compte de l'importance des flux d'informations entre les différents sous-systèmes.

-6- Coût en termes de composants de base : le coût des interfaces entre les différents sous-systèmes est généralement difficile à estimer.

Seuls les critères de vitesse et de coût peuvent être mesurés, il n'est pas étonnant que ces deux facteurs soient les deux seuls pris en compte par les compilateurs de silicium. Les autres critères sont difficilement chiffrables. Pour évaluer la flexibilité d'une architecture, par exemple, les architectes utilisent leur expérience et leur intuition, ce qui est difficile à formaliser.

En fait, la plupart des compilateurs ne contiennent pas d'étapes de découpage. Cependant quand ils en contiennent une, elle est basée sur des algorithmes simples.

Les algorithmes de découpage utilisés par les compilateurs de silicium sont généralement guidés par l'organisation de la description d'entrée. Ces algorithmes utilisent l'organisation hiérarchique de la description comportementale pour générer les sous-systèmes. Le découpage en sous-systèmes peut être fait de manière directe. Dans ce cas, chaque module de description comportementale est réalisé par un sous-système. Cette technique est utilisée par la plupart des compilateurs qui acceptent des descriptions hiérarchiques [FOX85] [CAM87].

La partition doit aussi définir les interfaces entre les sous-systèmes ainsi que les protocoles de communication. Dans la description comportementale, la communication entre les modules se fait à travers des événements véhiculés par les paramètres de ces modules et via les variables globales. Après le découpage, chaque sous-système communiquera avec les autres uniquement via ses entrées/sorties. Il faut donc distribuer les accès aux variables globales et les accès aux entrées/sorties globales entre les différents sous-systèmes. De plus il faut transformer les paramètres en signaux d'entrées/sorties.

Dans le cas où le langage de description interdit l'utilisation de variables globales (autres que celles définies comme paramètres de modules) les interfaces des sous-systèmes seront déduites des paramètres des modules correspondants. Dans ce cas, tous les protocoles de communication sont contenus dans la description d'entrée. Cette technique est utilisée dans le YSC [CAM87]. Il faut aussi noter que dans ce cas la hiérarchie de la description comportementale est une hiérarchie structurelle, ou plutôt mixte [MAR86]. Si le langage d'entrée permet l'utilisation des variables globales, comme c'est le cas dans Hercule [DEM88], le compilateur doit générer des unités spéciales accessibles à plusieurs sous-systèmes. Il doit aussi prévoir des entrées/sorties dans les sous-systèmes pour accéder à ces unités et insérer de nouveaux protocoles de communication dans les fonctions des sous-systèmes correspondants. Dans le cas du système Hercule [DEM88], ceci est fait à travers les primitives de synchronisation (send, receive, ...). Les diverses utilisations des variables globales sont transformées en requêtes d'utilisation asynchrones.

Les systèmes de découpage supposent aussi d'autres hypothèses simplificatrices telles que l'utilisation d'une seule horloge pour synchroniser tous les sous-systèmes. Enfin, il faut remarquer que dans le cas où le résultat du découpage doit être réalisé par plusieurs circuits, il faut résoudre en plus les problèmes dus aux interfaces inter-circuits telles que la longueur des connexions [GLA85].

Les compilateurs SYCO [JER86] et ArchitectWorkBench [WAL88] utilisent des algorithmes de partition plus sophistiqués.

SYCO découpe la description comportementale en plusieurs contrôleurs et une partie opérative. La partie contrôle peut être composée d'une hiérarchie de contrôleurs pouvant fonctionner en pipe-line. La hiérarchie est obtenue par un réarrangement de la hiérarchie contenue dans la description comportementale. Cet algorithme de partition sera détaillé plus loin (IV.6).

Le système ArchitectWorkBench (AWB) permet de générer des architectures de type pipe-line. L'algorithme utilisé consiste à décomposer une description en processus pouvant fonctionner en pipe-line. L'algorithme utilise une forme intermédiaire graphique appelé VT [THO81] [WAL83] [SNO78]. Un processus réalise un sous-graphe de la forme intermédiaire ayant un seul point d'entrée.

Le fonctionnement de l'AWB sera illustré sur la fonction "$y=b^2 - 4ac$" (voir III.2.1). Supposons comme point de départ l'architecture donnée dans la figure III.7. Le découpage de cette fonction en processus de 1 cycle est donné par la figure III.13.

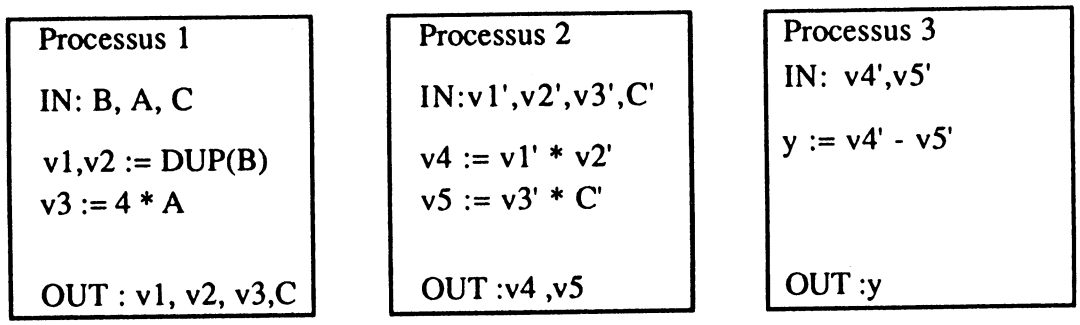


Figure III.13: Partitionnement de la fonction "Y" en processus pouvant fonctionner en pipe-line

Dans le système AWB la communication entre les processus se fait à travers des primitives de communication. La figure III.14 montre une description complète des trois processus réalisant la fonction "y". Les primitives "send" et "recv" réalisent respectivement l'émission et la réception de signaux externes pour assurer la synchronisation des processus. La primitive "restart" provoque un redémarrage du processus.

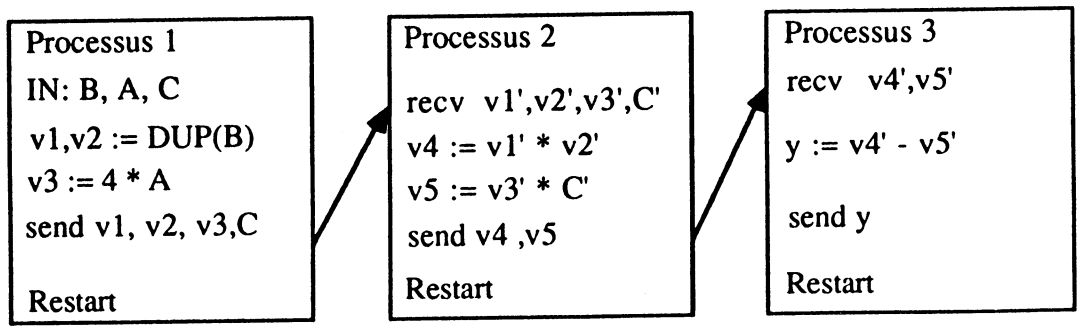


Figure III.14: Synchronisation des trois processus pour fonctionner en pipeline

Le temps de calcul de la fonction "y" peut être réduit à une moyenne de un seul cycle dans le cas où les primitives de communication ne prennent pas de temps.

L'algorithme de partition de l'AWB n'est pas incompatible avec celui utilisé par SYCO. Il serait possible de les intégrer dans le même système.

III.6.2 Découpage en partie opérative et partie contrôle

Dans ce cas, la partition consiste à générer un circuit composé d'une partie opérative et d'une partie contrôle. La section III.2 a montré deux exemples de ce type de partition. Il est généralement réalisé en deux étapes : l'allocation du matériel et l'ordonnancement des actions réalisées par le circuit. Or, comme il a été notifié dans [McF88], il se trouve que pour réaliser l'allocation, on a besoin de l'ordre d'exécution des opérations (le résultat de l'ordonnancement) et pour réaliser un ordonnancement des étapes, on a besoin de connaître le matériel disponible (résultat de l'allocation). Pour sortir de ce cercle, on utilise la description d'entrée pour obtenir une première solution. Dans le cas des circuits de contrôle, le partition commence par réaliser l'ordonnancement en se basant sur l'ordre des actions donné par la description. Dans le cas des circuits de communication, c'est l'étape d'allocation qui est réalisée en premier. Dans les deux cas, l'allocation et l'ordonnancement peuvent être appliqués de manière itérative pour optimiser une solution initiale. Dans les deux cas également, l'optimisation peut entraîner des changements parmi les choix établis. L'optimisation de l'allocation peut aussi modifier l'ordonnancement et vice versa. Les techniques d'allocation et d'ordonnancement dites constructives permettent de réaliser l'une et l'autre de manière simultanée.

Les techniques de partition en parties opératives et parties de contrôle seront détaillées dans la section III.9 qui est consacrée aux différents schémas de compilation, tandis que les techniques d'ordonnancement et d'allocation font l'objet des deux prochaines sections.

III.7 Ordonnancement (allocation du temps)

Dans le cas de la génération d'une machine séquentielle, le temps est découpé en tranches (cycles), et le traitement en opérations élémentaires (dont la durée d'exécution ne dépasse pas un cycle). L'ordonnancement, dit aussi allocation du temps, permet de lier chaque opération à une tranche de temps. L'ordonnancement détermine donc les opérations qui doivent s'exécuter en parallèle (durant le même cycle). Le problème de l'ordonnancement dans la compilation de silicium est similaire à celui de l'optimisation de micro-code [TOK81][FIS81][DAV81][ISO83], il constitue un cas particulier des problèmes d'ordonnancement en général.

III.7.1 Ordonnancement et compactage de micro-programmes

Les techniques d'ordonnancement sont souvent comparées aux techniques de compactage de micro-programmes et aux techniques de compactage d'un code intermédiaire, utilisées par les compilateurs de programmes. La différence entre l'ordonnancement, des opérations, effectué par un compilateur de silicium et le compactage de micro-programmes est que le compilateur de silicium permet aussi de générer les ressources nécessaires à l'exécution des opérations, tandis que dans le cas du compactage de micro-code, les ressources sont figées. De même, l'ordonnancement est différent du compactage des codes intermédiaires : en fait, le premier réorganise les opérations qui pourront être exécutées en parallèle, tandis que le second réordonne des opérations qui seront exécutées séquentiellement.

La terminologie utilisée pour les techniques de micro-programmation sera retenue pour introduire les méthodes d'ordonnancement. On appelle micro-opération une primitive de base de la machine. Une micro-opération correspond à une action opérative (transfert, opération) ou une action de contrôle pouvant s'exécuter en un seul cycle. L'ordre d'exécution des micro-opérations peut être défini par des flux de contrôle et/ou par des flux de données. Une micro-instruction est un ensemble de micro-opérations devant s'exécuter en parallèle durant un cycle donné. Le but de l'ordonnancement est de regrouper les micro-opérations en micro-instructions. Un micro-programme est constitué par un ensemble de micro-instructions. L'ordonnancement combine les micro-opérations dans des micro-instructions tout en respectant l'ordre d'exécution des opérations défini par les flux de contrôle et les flux de données. Ce processus peut être guidé par des contraintes de performances (surface et/ou vitesse). Le problème général est NP-complet [TOK81]. Donc toutes les méthodes qui ne prennent pas un temps exponentiel ne peuvent prétendre produire des solutions optimales [FIS81]. La plupart des algorithmes d'ordonnancement utilisent des heuristiques pour produire des solutions acceptables. Dans le cas des micro-programmes, on utilise le terme de compactage plutôt que celui d'optimisation.

Pour pouvoir s'exécuter en parallèle, les micro-opérations d'un même cycle ne doivent pas être en conflit. Il existe trois types de dépendance pouvant créer des conflits entre deux opérations :

- La dépendance des données : une opération ne peut pas utiliser des données calculées par une autre opération durant le même cycle.
- La dépendance des ressources : deux opérations différentes qui appartiennent à un même cycle ne peuvent pas utiliser la même ressource(un opérateur ou un registre en écriture).
- La dépendance de contrôle : un cycle ne peut pas contenir plus d'une instruction de branchement active à la fois. La notion de branchement actif sera définie plus loin (IV.7). Dans le cas des programmes de compactage de micro-programmes, cette dépendance est souvent appelée dépendance de format [TOK81]. Le terme vient du fait que chaque micro-instruction ne peut contenir qu'un seul branchement. Dans le cas général, où la partie contrôle n'est pas réalisée par une simple ROM, un cycle peut contenir plusieurs opérations de branchement conditionnel si, pour tout couple de branchement qui mènent à deux adresses différentes, les conditions de branchement sont exclusives.

Pour la compilation de silicium, l'étape d'ordonnancement peut être réalisée avant ou après l'étape d'allocation. Dans le premier cas, le but de l'ordonnancement est de fixer les paramètres de l'étape d'allocation et, par la suite, les caractéristiques de la partie opérative (résultat de l'allocation des ressources). Dans le second cas, l'ordonnancement a pour but d'optimiser l'utilisation de la partie opérative, et chaque micro-instruction correspond à un vecteur de commande pour la partie opérative.

III.7.2 Ordonnancement et formes intermédiaires

Les algorithmes d'ordonnancement utilisent la forme intermédiaire comme point de départ. Il est donc important de distinguer les techniques adaptées à chaque type de forme intermédiaire.

Dans le cas d'une forme intermédiaire graphique, les opérations sont représentées dans une structure à deux niveaux : des "blocs", contenant des opérations dont l'ordre d'exécution est déterminé uniquement par les flux de données et un graphe de contrôle qui donne l'ordre d'exécution des différents blocs. La figure III.12 représente la description de la fonction PGCD sous cette forme. Les blocs de base sont généralement représentés par un graphe de flux de données.

En partant d'une forme intermédiaire graphique, on distingue les algorithmes d'ordonnancement local et les algorithmes d'ordonnancement global. Les premiers effectuent des traitements locaux sur un bloc de base. Ces algorithmes sont basés sur une analyse des flux de données. L'ordonnancement global permet, lui, de réaliser des transferts d'opérations

inter-blocs. Il tient compte à la fois des flux de données et des flux de contrôle.

Dans le cas d'une forme intermédiaire syntaxique, les opérations sont généralement regroupées en cycles dès le départ. La notion de bloc de base n'existe pas explicitement, mais on peut retrouver ces blocs par une analyse de la forme intermédiaire. Les algorithmes d'ordonnancement local (voir ci dessous) ne sont pas applicables directement sur une forme intermédiaire syntaxique. Par contre ce type de représentation est bien adapté à la plupart des algorithmes d'ordonnancement global.

III.7.3 Ordonnancement local

Une étude comparative des quatre techniques principales utilisées pour le compactage local de micro-codes est présentée dans [DAV81]. Trois de ces techniques ont été largement utilisées pour l'ordonnancement dans la compilation de silicium.

La première de ces techniques est basée sur le principe "premier arrivé premier servi". Chaque opération est placée dans un cycle le plus tôt possible. Cette technique a été utilisée par plusieurs des premiers systèmes tels que CMU-DA (première génération) [HIT83], FLAMEL [TRI85]. Elle consiste à construire les micro-instructions une à une en parcourant les opérations à ordonner. L'algorithme d'ordonnancement est donné dans la figure III.15.

```
Algorithme d'ordonnancement local :
Entrée O : liste de micro-opérations
Sortie I : liste de micro-instructions

    Debut
        Tantque (O non vide) Faire
            Créer un nouveau cycle C (micro-instruction)
            Tantque (O non vide et la première opération de O n'est pas en conflit avec
                    toutes les opérations de C) Faire
                Retirer la première opération de O et l'insérer en queue de C
            Fin
            Insérer C en queue de I
        Fin
    Fin
```

Figure III.15 Exemple d'algorithme d'ordonnancement local ("premier arrivé premier servi")

Cette technique est la plus simple à programmer, mais les résultats produits dépendent de l'ordre initial des opérations.

La seconde technique est une variante de la première, elle utilise une liste d'ordonnancement qui lui permet de différer les décisions. La liste d'ordonnancement contient à tout instant toutes les opérations pouvant être placées dans la micro-instruction courante. A

chaque étape on place la "meilleure" candidate et on remet à jour la liste d'ordonnancement. Il faut définir une fonction coût qui permet de sélectionner la meilleure candidate.

Cette technique est aussi facile à réaliser que la première, par contre elle est plus performante [DAV81], même en utilisant une fonction coût simple. Elle a été utilisée par le compilateur de silicium ELF [GIR84].

La troisième technique est basée sur la notion de chemin critique. Le problème d'ordonnancement est, ici, ramené à un problème "potentiel-tâche". Cette technique donne des résultats qui sont comparables aux résultats de la précédente mais elle nécessite plus de temps de calcul. [DUR88] détaille un algorithme d'ordonnancement basé sur le formalisme "potentiel-tâche".

La quatrième technique d'ordonnancement utilise un arbre de décision ("branch and bound"). Théoriquement cet algorithme permet d'obtenir une solution optimale. Cependant, pour des raisons de temps de calcul, on est souvent amené à ne pas effectuer une recherche exhaustive de toutes les solutions. Les résultats obtenus par ce type d'algorithme va donc dépendre de l'heuristique utilisée pour limiter l'espace des solutions parcourues. En pratique, cette technique nécessiste plus de temps de calcul et donne des résultats qui restent comparables à ceux obtenus avec les deux techniques précédentes.

En plus des algorithmes d'optimisation locale développés initialement pour le compactage de micro-programmes, il existe des algorithmes spécifiques à la compilation de silicium. Le système HAL [PAU86] [PAU87] utilise un algorithme d'ordonnancement local basé sur un modèle de "force". Contrairement aux autres algorithmes, HAL ne traite pas les opérations une à une, mais les considère toutes simultanément. Le point de départ du système HAL est un graphe de flux de données. Après avoir déterminé le nombre de cycles, ou de micro-instructions, nécessaires pour ordonner le graphe de départ, l'algorithme calcule toutes les possibilités de placement pour chacune des opérations. Le reste du traitement est itératif, à chaque étape il place l'opération choisie de manière à minimiser une fonction coût. Le calcul du coût est basé sur un modèle de "graphe élastique" [PAU87].

III.7.4 Ordonnancement global

L'ordonnancement global doit tenir compte d'un nouveau facteur, par rapport à l'ordonnancement local, qui est la fréquence d'exécution de chaque opération. Dans le cas de l'ordonnancement local, le nombre de micro-instructions définit le nombre de cycles nécessaires pour l'exécution d'un bloc de base. Dans le cas de l'optimisation globale, le nombre de cycles nécessaires à l'exécution d'une description peut être difficile voire impossible à définir de manière statique. Reprenons l'exemple du PGCD de la figure III.12, si on suppose que chaque bloc est exécutable en un cycle, comment déterminer le nombre de cycles nécessaires pour

calculer un PGCD dans le cas général (indépendamment de la valeur des entrées)? Pour cela, il faut connaître le nombre de fois que la boucle contenant les blocs (B2, B4, B5) est exécutée. Ce nombre dépend des deux entrées (x et y).

Deux techniques ont été utilisées pour estimer la fréquence d'exécution des blocs. La première utilise les fréquences de l'exécution réelle. On utilise la trace d'une simulation pour déterminer les blocs les plus fréquemment utilisés [ZIM79]. Pour que le résultat de cette technique soit significatif il faut que le jeu de stimuli soit représentatif de l'environnement réel du circuit. La seconde technique utilise les fréquences de l'exécution relative. Elle procède de manière statique pour comparer la fréquence d'exécution de chacun des blocs avec ses prédécesseurs et ses successeurs. Dans le cas de la figure III.12, par exemple, on peut affirmer que la fréquence d'exécution du bloc B2 est supérieure ou égale à celle des blocs B4 et B5, car pour exécuter l'un de ces deux derniers blocs il faut passer par le premier. Les auteurs de cette technique [ISO83] remarquent que cet ordre local entre les blocs est suffisant car, dans le cas général, l'ordonnancement consiste à déplacer une micro-opération d'un bloc à un autre bloc voisin.

Les algorithmes d'ordonnancement restent essentiellement des algorithmes de compactage de micro-programmes. Peu de compilateurs de silicium contiennent une étape d'optimisation globale de l'ordonnancement. Trois algorithmes de compactage pouvant être utilisés par des compilateurs de comportement seront cités. Ces algorithmes sont itératifs, ils agissent sur une description déjà organisée en micro-instructions (ou cycles). L'ordonnancement initial est souvent donné par la description d'entrée.

La première technique [FIS81] utilise des chemins d'exécution appelés "trace" [FIS81]. Une trace est une séquence de cycles pouvant être exécutée de manière contiguë pour une certaine configuration des données. La configuration est déterminée par les valeurs des expressions de condition. Une trace ne doit pas contenir de boucles. Cette décomposition en trace ne tient pas compte du découpage en blocs de base. Une trace peut donc contenir plusieurs blocs de base. Chaque trace est compactée indépendamment des autres. Comme le compactage d'une trace peut entraîner le rallongement d'autres traces, les traces les plus fréquemment utilisées sont compactées en premier. Les transformations consistent à réordonner les opérations de la trace.

La seconde technique [ISO83] utilise une représentation appelée graphe de dépendance des données ("Data Dependancy Graph"). Ce dernier est un graphe de flux étendu qui permet de représenter la dépendance des données entre les micro-instructions pouvant appartenir à des blocs de base différents. Il permet donc de détecter les micro-instructions pouvant être placées dans plusieurs blocs de base différents. Pour chacune de ces micro-instructions, l'algorithme de compactage va choisir le meilleur placement. Les

micro-instructions sont placées de manière à ce que les blocs de base les plus fréquemment utilisés contiennent le moins possible de micro-instructions. Cette méthode utilise les fréquences relatives des exécutions (voir ci-dessus).

La troisième méthode de compactage [TOK81] utilise un graphe de précédence des micro-instructions. Chaque sommet du graphe représente une micro-instruction, les arcs décrivant l'ordre d'exécution des micro-instructions. L'algorithme examine les possibilités de fusionner chaque sommet avec ses prédécesseurs ou avec ses successeurs (dans le graphe de précédence). On peut remarquer que cette technique permet aussi de réaliser l'ordonnancement local. Cette technique de compactage a été utilisée partiellement dans SYCO [TLE87], [BEK87]. Seuls les successeurs ont été considérés. Cette limitation vient du fait que SYCO utilise une forme intermédiaire syntaxique où il est difficile de retrouver le prédécesseur d'un cycle sans un parcours complet de la description.

L'importance de l'ordonnancement local et de l'ordonnancement global pour un compilateur de silicium dépend du type de circuits compilés.

Les techniques d'ordonnancement local sont limitées à des blocs de micro-opérations ne contenant pas de coupure de séquence. Un bloc ne contient pas d'opérations de branchement autre que l'opération de fin de bloc. Dans le cas de la compilation de silicium, ces techniques ont été surtout utilisées pour la compilation de circuits de communication. En fait la première étape de ce type de compilateurs consiste, généralement, à transformer la description en un graphe de flux de données pouvant être représenté par un seul bloc de micro-opérations.

Mais dans le cas d'un compilateur de circuits de contrôle, l'étape d'ordonnancement global est plus importante. La description d'un circuit de contrôle contient plusieurs blocs de micro-opérations. Un ordonnancement global est nécessaire pour réaliser un compactage inter-blocs. D'autre part, la non restriction de l'utilisation des instructions de branchement fait que la taille moyenne d'un segment est très faible [TOK81], ce qui limite l'intérêt de l'optimisation locale.

III.8 Allocation des ressources

L'allocation permet de lier (ou d'associer) les objets manipulés par la description à des ressources matérielles. Chaque élément de mémorisation est lié à une unité de mémorisation matérielle (registre, constante, bloc mémoire ...). Chaque opération est liée à une unité capable de l'exécuter. Chaque transfert de données est lié à un chemin de communication (bus et multiplexeur). Ces éléments peuvent être déclarés explicitement dans la description d'entrée et/ou rajoutés par d'autres étapes du processus de compilation. L'allocation permet de générer la partie opérative.

L'allocation est généralement décomposée en trois catégories, reflets de trois problèmes différent : l'allocation des registres, l'allocation des opérateurs et celle des connexions. La complexité de ces trois problèmes rend la recherche d'une solution optimale difficile. La plupart des systèmes utilisent des heuristiques pour réaliser l'allocation en vue d'atteindre (resp. de ne pas dépasser) des objectifs (resp. des limites) de surface et de vitesse. Les trois types d'allocation peuvent être réalisés un à un ou en parallèle. Pour une étude détaillée de l'étape d'allocation de matériel, on peut se reporter à [THO83] et [JAM86a], ce dernier contient aussi une bibliographie plus complète sur cette étape. Le reste de cette section ne donne qu'un bref aperçu des différentes techniques utilisées pour l'allocation de matériel.

Il existe trois classes d'algorithmes d'allocation : les algorithmes dits itératifs, les algorithmes basés sur la recherche des "cliques" (voir plus loin), et ceux qui procèdent de manière constructive.

Un algorithme itératif utilise une solution initiale pour l'améliorer de manière itérative. La première solution peut être obtenue par un algorithme d'allocation directe, comme ceux utilisés en III.2. La suite du traitement procède de manière itérative pour réaliser des améliorations locales de cette solution initiale. Ces améliorations sont réalisées via des transformations dont l'objectif est de réduire la surface et/ou augmenter la vitesse du circuit. Il faut remarquer que les mesures de performances à ce niveau sont généralement le résultat d'une estimation et non de mesures exactes.

Pour l'allocation des opérateurs, la réduction de la surface peut se faire par le choix d'unités matérielles plus réduites en surface (généralement plus lentes) ou par le partage des unités. En fait, le coût de la surface se compose de la surface des unités matérielles allouées et de la surface des zones de routage. Si on ignore le coût des zones de routage, tout partage de matériel constitue une réduction de la surface. L'amélioration de la vitesse peut se faire par le choix d'unités plus rapides (généralement plus encombrantes) ou par l'augmentation du parallélisme. La modification du parallélisme peut entraîner une modification de l'ordonnancement.

Pour l'allocation des registres, les transformations consistent essentiellement à fusionner des registres et remplacer un ensemble de registres par un banc de registres. La fusion des registres nécessite une analyse des durées de vie des registres [AHO86]. Si on ignore le coût des interconnexions (bus et multiplexeurs) ces transformations permettent de réduire la surface de la partie opérative. Dans le cas de la fusion de registres, il est évident qu'un registre en moins ne peut qu'entraîner une diminution de la surface. Dans le cas de la formation de bancs de registres, le gain provient de la diminution du nombre de commandes nécessaires pour piloter les registres. A noter que dans le cas où la partie opérative est à base de multiplexeurs (voir ci-dessous), le gain dû à l'allocation des registres doit être modéré par la modification des structures d'interconnexions [JAM86a]. Par exemple, dans le cas où la fusion de deux registres en un seul contraint à la création d'un nouveau multiplexeur, et que le coût du multiplexeur est supérieur à celui du registre, la fusion des registres entraîne plutôt une augmentation de la surface [KOW85].

Pour l'allocation des éléments d'interconnexion, JAMIER distingue les parties opératives à mutiplexeurs et les parties opératives à bus [JAM86a]. Dans le premier cas, les unités sont connectées point à point. Dans le second cas, elle sont connectées via des bus pouvant être partagés. Les deux stratégies peuvent être mélangées au sein d'une même partie opérative [TSE84]. Les transformations consistent essentiellement à réduire les interconnexions inutiles et partager les unités d'interconnexion. Elles peuvent aussi remplacer des multiplexeurs par des bus et inversement. Il n'existe pas de cas général pour ces transformations : pour étudier les effets de chaque transformation il faut voir le cas particulier où elle est appliquée [JAM86a].

L'efficacité des méthodes itératives dépend énormément de la solution initiale. Dans le cas de la compilation d'un circuit de commande, une solution initiale obtenue par allocation directe, en partant du graphe de flux de données, donne une architecture avec le maximum de parallélisme, elle respecte donc les contraintes de vitesse. Le traitement itératif a pour but de satisfaire les contraintes de surface. L'amélioration consiste essentiellement à partager les unités matérielles (registres, opérateurs, connexions). La diminution du parallélisme réel, celui permis par le graphe de flux de données, permet aussi de diminuer la surface. La plupart des transformations utilisées par la méthode itérative découlent de l'analyse des flux de données, facile à réaliser dans le cas d'un circuit de communication. Par contre, dans le cas de la compilation de circuits de commande, une analyse des flux de données est difficile à réaliser sur tout le circuit. La méthode itérative ne permet donc de réaliser que des améliorations locales.

Les algorithmes basés sur la recherche des "cliques" [TSE83] procèdent de manière globale pour réaliser l'allocation. Une première étape détermine les éléments de la description qui peuvent partager la même unité. Elle consiste à trouver des "cliques". Une "clique" est un

ensemble d'éléments, qui pris deux à deux, peuvent se partager une ressource donnée. Cette méthode peut s'appliquer aux trois problèmes d'allocation.

La génération d'une solution optimale revient à trouver le nombre minimal de "cliques" disjointes. Là encore le problème est NP-complet. Les essais de réalisations par cette méthode [DUR88][McF88] ont montrés que cette technique est très coûteuse en temps de calcul.

Les algorithmes d'allocation de la troisième catégorie procèdent de manière constructive. La technique consiste à sélectionner un élément (registre, opérateur ou transfert), effectuer la liaison et réitérer tant qu'il existe des éléments à lier. Le résultat de ces algorithmes dépend de deux facteurs :

- l'ordre dans lequel les éléments sont pris,
- la possibilité de déterminer les choix bloquants à l'avance. Dans le cas où le système permet des retours arrières lors de la construction, cette possibilité n'est pas nécessaire mais elle permet d'économiser du temps de calcul.

Les compilateurs de parties opératives DAA [KOW85] et APOLLON [JAM85] utilisent des méthodes constructives pour l'allocation. Le premier est organisé comme un système expert. Il contient des règles d'allocation qui sont exécutées par un moteur d'inférence. L'ordre d'application des règles est basé sur le schéma de chaînage avant. Cette organisation fait que le système est facilement extensible, elle permet l'introduction de nouvelles règles. Les résultats obtenus, par ce système, montrent que ce type de réalisations reste peu efficace pour pouvoir affronter le monde réel. Par exemple la compilation du µ370 (l'unité centrale de la série 370 IBM), a nécessité 47 heures de calcul (sur un VAX 11/780), la partie opérative générée respecte les contraintes de vitesse, mais elle est irréalisable d'un point de vue surface et du point de vue électrique. La solution proposée ne tient pas compte des problèmes de placement-routage et des temps de transferts.

Le compilateur APOLLON utilise, lui aussi, une méthode constructive. Pour tenir compte des problèmes réels, posés par la conception d'un circuit VLSI, il utilise une architecture cible et une stratégie d'implantation des dessins des masques. Comme il a été dit plus haut, ce type de choix limite le compilateur à une classe réduite d'applications. Mais en contre partie il permet de générer des solutions efficaces. L'allocation est précédée d'un pré-traitement qui donne l'ordre dans lequel les opérations de liaison doivent se faire. Ce pré-traitement permet d'éviter les solutions bloquantes [JAM86a].

III.9 Schémas de compilation

Les différentes étapes de compilation citées plus haut se retrouvent dans la plupart des compilateurs de comportements existants. Par contre, l'importance de chacune de ces étapes dépend de leur langage d'entrée, et de la méthodologie de conception qu'ils réalisent. Par exemple, dans le cas des compilateurs de circuits de communication, l'allocation des registres constitue une étape critique du processus de compilation tandis que l'ordonnancement est généralement facile à réaliser. Par contre l'ordonnancement constitue une étape critique dans le cas des compilateurs de descriptions impératives alors que l'allocation des registres est généralement réalisée de manière directe.

Cette section présente l'application des techniques de compilation vues plus haut pour les compilateurs de circuits de contrôle et les compilateurs de circuits de communication. Les deux types de compilateurs utilisent deux schémas de compilation différents. Les compilateurs de circuits de contrôle commencent par générer une première solution, la moins parallèle possible, pour réaliser la description comportementale. Cette solution doit réaliser le parallélisme de la description initiale, tandis que les compilateurs de circuits de communication commencent par générer une solution initiale où le parallélisme n'est limité que par les flux de données de la description comportementale. Dans les deux cas le processus de compilation consiste à transformer cette description en tenant compte des contraintes de performances.

Les deux schémas de compilation utilisés par ces deux types de compilateurs seront présentés dans la suite. Cependant, la présentation sera limitée à la génération de circuits séquentiels composés d'une partie contrôle et d'une partie opérative.

III.9.1 Compilation de circuits de commande

Un circuit de contrôle ou de commande a été défini, au premier chapitre, comme étant un circuit qui réalise un traitement complexe sur un faible flux d'informations. Ces circuits sont souvent appelés circuits de type micro-processeur. La compilation de ce type de circuits est essentiellement basée sur des techniques d'analyse de flux de contrôle. Le processus de compilation commence par générer une partie opérative qui réalise le parallélisme contenu dans la description. L'effort d'optimisation est consacré en grande partie à la génération de la partie contrôle.

III.9.1.1 Principe

La figure III.16 montre le schéma de fonctionnement d'un système-type pour la compilation de descriptions impératives pour des circuits de contrôle.

Ce processus démarre avec une description algorithmique du comportement. L'étape de transformation de cette description en une forme intermédiaire est généralement suivie par une étape d'optimisation de la description comportementale. Le reste du processus de compilation réalise l'ordonnancement et l'allocation. L'ordonnancement est réalisé en deux étapes séparées par l'étape d'allocation. La première produit un ordonnancement initial. L'étape d'allocation détermine les ressources et l'organisation de la partie opérative. Les spécifications de la partie contrôle sont générées par la seconde étape d'ordonnancement qui réalise une optimisation de l'ordonnancement initial.

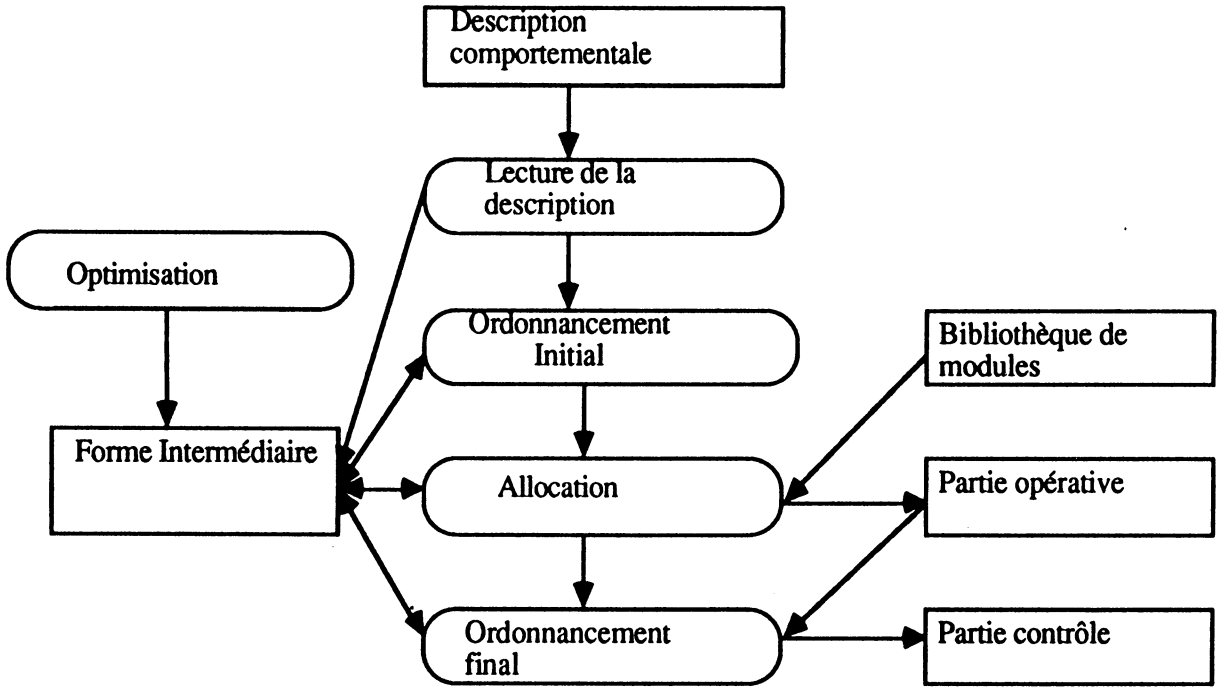


Figure III.16 Schéma de compilation d'un circuit de commande

Ce schéma est celui de la plupart des compilateurs orientés vers les circuits de commande tels que CMU-DA (première version) [THO81][HAF82], SFL [NAK85][NAK87], ELF [GIR84], HERCULE [DEM88], MacPitts [SIS82][SOU83][SOU88], OCCAM [MAY87], ... Il peut paraître paradoxal qu'un compilateur d'OCCAM génère des circuits séquentiels. En fait le compilateur cité [MAY87], compile des processeurs (processus) élémentaires qui sont réalisés sous forme de circuits composés d'une partie opérative simple (la partie opérative contient au plus une UAL) et d'une partie contrôle. La première génération de compilateurs de comportement, tels que ALERT [FRI69], avait une organisation plus simple. Ils utilisaient un schéma de compilation basé sur la macro-génération de descriptions au niveau transfert de registres en partant d'une description algorithmique du comportement.

III.9.1.2 Langage de description

Les langages du type impératif sont bien adaptés pour ce type de circuits. Le comportement du circuit est décrit par un ensemble d'opérations dont le séquencement est défini par des constructions de contrôle. Ces constructions (instruction de choix, branchement et appel de procédures) permettent de spécifier des contrôleurs complexes. Le parallélisme décrit explicitement dans la description de départ va déterminer le parallélisme de la solution initiale. Trois types de langages ont été utilisés par ce type de compilateurs, MacPitts et HERCULE utilisent des langages ad-hoc, Elf utilise un sous-ensemble d'un langage de programmation (ADA), d'autres utilisent des sous-ensembles d'un langage de description de matériel existant.

III.9.1.3 Ordonnancement initial

L'étape d'ordonnancement initial permet de décomposer les opérations complexes en opérations de base (pouvant se dérouler en un seul cycle de contrôle) et de calculer les branchements implicites. Elle spécifie aussi les données de l'étape d'allocation. La complexité de cette étape dépend des limitations imposées par la description d'entrée et de la stratégie adoptée pour l'allocation.

Cet ordonnancement est nécessaire pour la suite des opérations. Il permet de transposer la description d'entrée en une machine d'états finis. Chaque état correspond à une étape de contrôle. Ces étapes sont aussi appelées cycles de contrôle. Chacun de ces cycles pourra être décomposé par la suite en plusieurs cycles physiques (cycles de base). Les cycles de contôle peuvent être constitués par une lecture simple de la description. Par contre le calcul de l'état suivant peut nécessiter des calculs complexes dans le cas où le langage de description permet de décrire des branchements implicites [MHA88]. Il faut noter que le but de cette étape n'est pas de minimiser le nombre d'états de la machine, mais de trouver un ordonnancement initial pour pouvoir réaliser l'allocation.

III.9.1.4 Allocation

Cette étape doit générer la partie opérative. Dans le cas d'une description procédurale, l'allocation des registres est limitée aux variables de boucles et aux variables locales, dans le cas où le langage permet une description hiérarchique. L'allocation des variables globales est immédiate car ces variables représentent des éléments de mémorisation et non des valeurs, comme c'est le cas dans une description de flux de données. Il reste donc à faire l'allocation des connexions et celle des opérateurs.

Dans le cas où les cycles de contrôle peuvent utiliser des opérateurs nécessitant plusieurs cycles machine (appelés aussi sous-cycles), l'étape d'allocation doit aussi réaliser le découpage des cycles de contrôle en cycles de base : ce découpage est un problème

d'ordonnancement. Les algorithmes d'allocation qui utilisent des méthodes constructives [KOW85][JAM85], permettent de réaliser cet ordonnancement en même temps que l'allocation.

Cette étape produit la partie opérative et l'ordonnancement à l'intérieur des cycles de contrôle. Le nombre de sous-cycles de contrôle peut être fixe ou variable. Dans le cas d'APOLLON [JAM85] par exemple, le nombre de sous-cycles est fixé à 2. Dans le cas où le cycle de départ nécessite un seul sous-cycle pour s'exécuter, on peut avoir un sous-cycle vide. Dans le cas où le nombre de sous-cycles est variable, la marge de manœuvre de l'étape d'allocation devient plus grande. Le nombre de sous-cycles sera fixé en fonction de contraintes de surface et de vitesse. Ce cas sera détaillé plus loin (voir IV.8).

L'étape d'allocation peut avoir besoin de modifier l'ordonnancement initial soit pour respecter des contraintes de performances, soit pour des raisons algorithmiques. Avec le modèle à deux bus utilisé par APOLLON [JAM85], on peut trouver des combinaisons de cycles qui sont irréalisables à cause du modèle, dans ce cas il faut découper certains cycles.

La dernière étape de compilation, montrée par la figure III.16, permet de générer les spécifications du contrôleur. Cette étape peut réaliser une optimisation globale de l'ordonnancement. Elle doit également générer les informations d'interface entre la partie opérative et la partie contrôle pour les compilateurs d'architecture. Elle constitue le cœur des compilateurs de circuits de commande. En fait, c'est à ce niveau que l'on peut effectuer une optimisation architecturale pour réaliser des compromis surface/vitesse.

III.9.2 Compilation de circuits de communication

Un circuit de communication réalise un traitement peu complexe sur un grand flux d'informations. L'architecture d'un tel circuit contient généralement plusieurs opérateurs pouvant travailler en parallèle. La contrainte principale que doit respecter le circuit est la vitesse.

Il faut distinguer deux types de circuits de communication. Les premiers sont réalisés à base d'opérateurs auto-synchrones, les seconds sont composés d'une partie opérative et d'une partie contrôle. Dans un système auto-synchrone, dit aussi endochrone [MEA83], le contrôle est distribué entre les opérateurs. Le modèle de synchronisation dans ce type de systèmes est plus complexe que dans le cas des circuits synchrones où le contrôle est centralisé par un seul contrôleur [MEA80]. Dans la suite, on s'intéressera uniquement au cas des circuits de communication synchrones.

La compilation des circuits de communication est basée sur l'analyse des flux de données du circuit. La plus grande partie de l'effort de compilation est consacrée à la génération de la partie opérative. La partie contrôle est généralement réalisée par un simple contrôleur.

III.9.2.1 Principe

Dans un schéma de fonctionnement idéal pour un compilateur de circuits de communication, le comportement est décrit par un langage fonctionnel. La lecture de cette description permet de générer un graphe de flux de données qui constitue une forme intermédiaire pour les autres étapes de compilation. Le reste du processus de compilation permet de générer une architecture parallèle dans laquelle le contrôle est décentralisé. En pratique, ce schéma est doublement compliqué par l'utilisation d'un langage séquentiel pour la description du comportement et la génération d'une architecture séquentielle. Les raisons de la non utilisation des langages fonctionnels pour la description des circuits peuvent être trouvées dans [GAJ82]. L'utilisation d'une architecture dans laquelle le contrôle est centralisé vient de la difficulté de générer des circuits composés de sous-systèmes endochrones (auto-synchronisés).

La figure III.17 montre le schéma de fonctionnement d'un système type pour la compilation de circuits de communication.

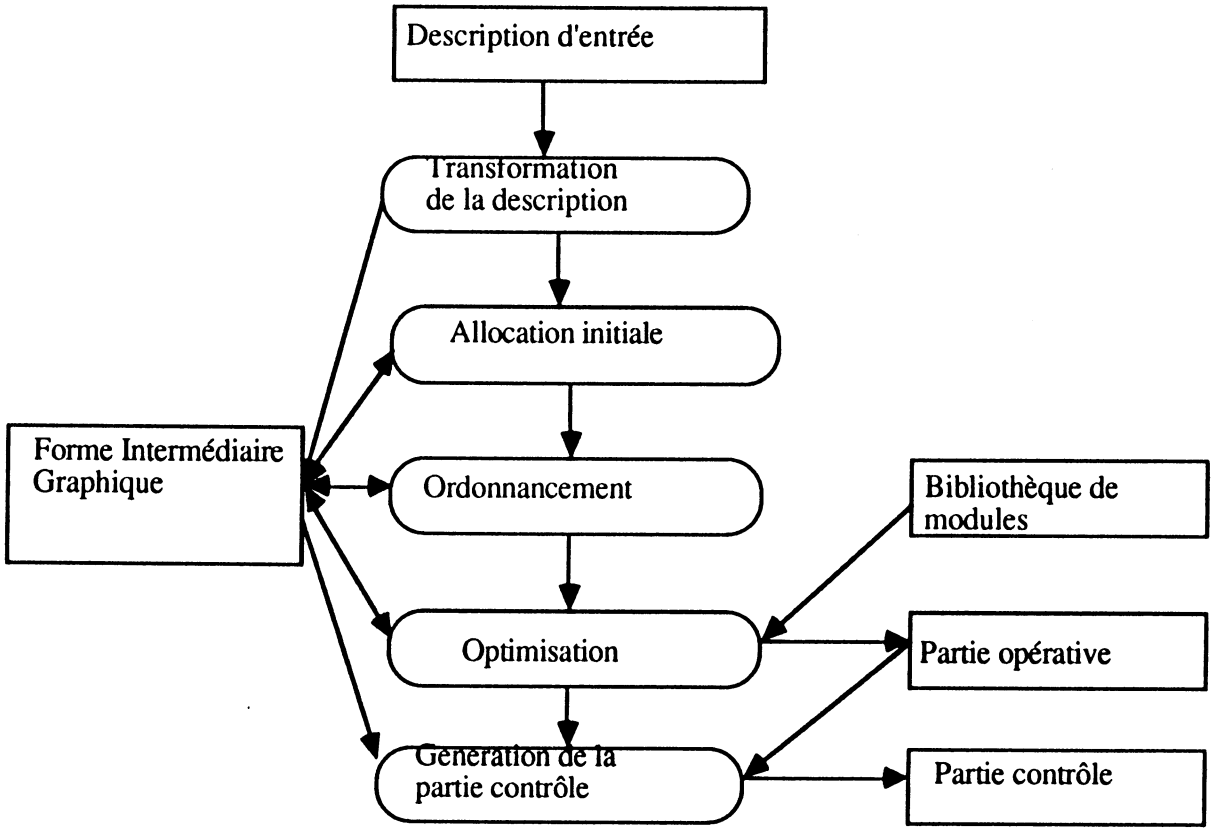


Figure III.17 Schéma de compilation de circuits de communication

Le processus de compilation commence par extraire un graphe de flux de données de la description comportementale. Les autres étapes de ce processus sont orientées de façon à générer une partie opérative parallèle permettant de respecter les contraintes de surface et de

vitesse.

Le schéma de la figure III.17 est aussi celui de la plupart des compilateurs orientés vers les circuits de communication tels que FLAMEL [TRI85], les systèmes de synthèse d'Eindhoven [JES88], DAA [KOW85], ASP [BAL86], YSC [CAM87], CMU-DA (deuxième génération) [THO83], Cathedral [DEM86], ...

III.9.2.2 Transformation de la description comportementale

La première étape du processus de compilation - la transformation de la description - consiste à paralléliser la description comportementale pour la transformer en un graphe de flux de données. Elle est souvent appelée étape de synthèse de l'architecture [CAM85][McF88]. Le résultat de cette étape est une forme intermédiaire graphique.

Cette étape réalise principalement une analyse des flux de données pour éliminer les dépendances entre les opérations. Les actions principales sont l'éclatement des boucles, l'expansion des appels de procédures et la décomposition des variables ("variable unfolding"). Pour faciliter l'analyse des flux de données, la plupart de ces systèmes imposent des restrictions au niveau du langage d'entrée. En fait la plupart de ces restrictions sont aussi imposées par les langages fonctionnels. Les restrictions les plus importantes sont :

- La restriction de l'utilisation des branchements : les instructions de branchement introduisent des dépendances de contrôle qui compliquent l'analyse des flux de données.
- L'utilisation des variables globales : dans une description de flux de données une variable n'est affectée qu'une seule fois. Une variable affectée plusieurs fois est décomposée en plusieurs variables. Cette opération est réalisée par une analyse des durées de vie des variables [AHO87]. Dans le cas d'une variable, cette décomposition nécessite une analyse complète de description.
- La hiérarchie : la hiérarchie de la description comportementale est généralement mise à plat pour réaliser l'analyse des flux de données. L'utilisation de paramètres pour les appels de procédure peut introduire des problèmes similaires à ceux posés par les variables globales.

III.9.2.3 Allocation initiale

La seconde étape permet de générer une solution initiale. Cette solution est générée directement à partir de la description des flux de données, produite par la première étape et représentée par la forme intermédiaire graphique. Le but de cette étape n'est pas de réaliser une allocation optimisée mais simplement de trouver une solution initiale afin de pouvoir réaliser l'ordonnancement. Cette étape peut être réalisée par une transposition directe du graphe des flux

de données en une architecture, on peut par exemple utiliser la méthode directe décrite en III.2.2.

III.9.2.4 Ordonnancement

Dans le cas où la première étape a permis de transformer la description comportementale en un graphe de flux de données, l'ordonnancement peut être réalisé par l'étape d'optimisation de l'allocation. Dans ce cas on considère que tout le graphe de flux de données constitue un seul cycle de contrôle. Ce dernier sera découpé en sous-cycles par l'étape d'optimisation de l'allocation. Dans le cas général la description initiale est transformée en un graphe de contrôle et plusieurs graphes de flux de données pouvant être réalisés en un seul cycle. On distingue quatre types de cycles [CAM87] :

- Les cycles dus aux boucles de la description comportementale, dans le cas où la première étape n'a pas éliminé les boucles. Chaque boucle introduit au minimum un cycle contenant les opérations du corps de la boucle. Ce traitement se trouve simplifié si le langage de description ne permet que l'utilisation des boucles structurées (instructions de boucles). Dans le cas où l'on peut réaliser des boucles à l'aide des instructions de branchement, il faut rajouter un nouveau type de cycles de contrôle [THO81].
- Les cycles dus aux appels de sous-modules. L'appel d'un sous-circuit nécessite généralement un cycle de contrôle. Ce dernier est utilisé pour initialiser le processus de communication avec le sous-circuit.
- Les cycles dus aux contraintes de performances. Par exemple, dans le cas où le temps de cycle maximum est fixé à l'avance, on peut être amené à limiter le nombre d'opérations séquentielles dans un cycle de contrôle pour respecter la longueur du temps de cycle. Ces cycles ne sont créés que dans le cas où l'étape d'allocation ne permet pas le découpage des cycles de contrôle en sous-cycles.
- Les cycles dus à l'utilisation des variables d'entrées/sorties : il sont introduits pour respecter les protocoles de communication.

Les cycles de contrôles constituent chacun un bloc de base dans le cas d'une forme intermédiaire graphique. Ils pourront être découpés par l'étape d'optimisation de l'allocation.

III.9.2.5 Optimisation de l'allocation

L'allocation initiale correspond à une architecture très coûteuse en matériel, mais qui respecte les contraintes de vitesse. Le but de cette étape est de réduire la surface tout en respectant les contraintes de vitesse. Cette étape nécessite les trois types d'allocation, l'allocation des registres qui permet généralement une réduction importante du nombre de registres, l'allocation des éléments d'interconnexion et l'allocation des opérateurs.

Cette étape permet aussi de découper les cycles de contrôle en cycles-machine. Dans

le cas des circuits de communication, l'étape d'ordonnancement génère le minimum de cycles de contrôle sans tenir compte de leur complexité. Ces cycles vont être découpés en fonction des ressources allouées. La marge de manœuvre de cette étape est généralement plus grande que dans le cas de la compilation des circuits de commande.

L'optimisation de l'allocation est l'étape principale des compilateurs de circuits de communication. En fait, c'est à ce niveau que l'on peut réaliser des modifications de l'architecture en vue d'améliorer le résultat du compilateur. Cette étape produit la partie opérative et complète l'ordonnancement. Elle peut être suivie d'une étape d'optimisation de l'ordonnancement pour générer les spécifications de la partie contrôle.

III.10 Conclusion

La séparation entre compilateurs de circuits de contrôle et de circuits de communication est due à l'état de l'art dans la compilation de silicium. Les premiers utilisent un schéma qui avantage la partie contrôle. Les caractéristiques de la partie opérative sont en grande partie fixées par la description comportementale ou fixées à l'avance par le choix d'une architecture cible. L'optimisation consiste à trouver des compromis architecturaux essentiellement au niveau de la partie contrôle. Les compilateurs de circuits de communication utilisent un schéma qui avantage la partie opérative. La partie contrôle est généralement constituée d'un contrôleur simple. Dans ce cas, l'optimisation consiste à modifier l'architecture de la partie opérative en vue de trouver un bon compromis surface/vitesse. La fusion de ces deux schémas doit conduire à la réalisation d'un compilateur de comportement permettant d'effectuer à la fois des optimisations de la partie opérative et de la partie contrôle. Pour que cette double optimisation soit possible, la définition d'un compilateur de comportement doit en tenir compte pour définir le langage de description, l'architecture des circuits générés et les algorithmes de compilation.

• La description du comportement : elle doit permettre une description précise des circuits. Le choix du langage de description doit tenir compte de l'architecture des circuits générés afin de permettre la réalisation d'algorithmes de compilation efficaces quant au temps de compilation et aux performances des circuits générés. Cette description doit permettre la description de plusieurs types de circuits.

• L'architecture : l'espace des solutions architecturales manipulées par le compilateur doit être suffisamment limité pour permettre une génération automatique et efficace des dessins de masques, et pour permettre des temps de compilation raisonnables. Par contre, il faut que l'utilisateur puisse choisir entre plusieurs solutions architecturales pour une description de comportement donnée. On peut par exemple fixer l'architecture globale du circuit, telle que celle basée sur le modèle partie opérative/partie contrôle, et laisser à l'utilisateur la définition du nombre d'opérateurs dans la partie opérative.

• Les algorithmes de compilation : les choix du langage de description et de l'architecture cible doivent tenir compte des algorithmes de compilation. Dans le cas où il existe une bonne correspondance entre le langage de description et l'architecture cible, il devient possible d'influencer le résultat du compilateur en modifiant la description d'entrée. Par contre, plus cette correspondance est forte, plus on se rapproche d'un compilateur d'architecture où l'architecture est complètement fixée par la description d'entrée.

La définition d'un compilateur de comportement doit aussi tenir compte des autres étapes de la compilation de silicium à savoir la compilation de l'architecture et la génération de

dessins de masques. Il est difficile d'optimiser certains aspects du circuit tels que la surface ou la consommation en agissant uniquement sur la description du comportement sans tenir compte de la structure ni de la génération des dessins de masques.

CHAPITRE IV
LE COMPILATEUR DE SILICIUM SYCO

Le projet SYCO a démarré en 1983 avec le projet national de CAO de VLSI SYCOMORE. Actuellement SYCO en est à sa deuxième version. La première version a été réalisée autour d'un langage de description de matériel, appelé IRENE. La seconde version est plus ou moins intégrée dans les outils de SYCOMORE. La troisième est en cours de préparation, elle inclura l'utilisation d'une architecture cible plus souple, ce qui permettra d'étendre le domaine d'application du compilateur de silicium SYCO.

La première section donne un bref aperçu du projet SYCOMORE et de la première version du projet SYCO. La deuxième version sera le sujet essentiel de ce chapitre. Elle sera souvent désignée par "version actuelle".

Le compilateur de silicium SYCO est le fruit d'un travail collectif. La plupart des travaux décrits par ce chapitre ont déjà fait l'objet de présentations détaillées qui seront référencées dans la suite. Le but de ce chapitre est de donner une vue d'ensemble du projet SYCO et d'introduire les extensions possibles pour la réalisation de la troisième version.

IV.1 Introduction

IV.1.1 Le projet CAPRI

SYCO est une suite logique du projet CAPRI dont le but était de définir une méthodologie de conception de circuits de type microprocesseur [ANC83], [ANC86]. Cette méthodologie préconisait l'utilisation de modèles de parties contrôle hiérarchiques et de parties opératives parallèles autour de deux bus.

Ces modèles architecturaux devaient être complétés, pour les besoins de SYCO, par un modèle de synchronisation, un modèle topologique et surtout un modèle de description comportementale.

Le processus de conception préconisé par CAPRI se composait essentiellement de quatre étapes [ANC83] :

- l'étape de conception architecturale permet de déterminer les spécifications de la partie opératives et celles de la partie contrôle,
- l'étape de conception de la partie opérative,
- l'étape de conception de la partie contrôle,
- l'étape de conception des parties restantes.

La première et la dernière étape ont été peu formalisées, car elles devaient se faire manuellement. L'étape de conception architecturale produit une spécification fine du circuit où l'architecture de la partie opérative est définie. Tandis que la dernière étape permet de finir

manuellement la personnalisation du circuit dans le cas des circuits non standards. Elle permet par exemple de réaliser les mécanismes d'interruption, l'ajout de plots d'entrées/sorties spéciaux ou la réalisation du système d'horloge.

Plusieurs outils ont été développés dans le cadre de cette méthodologie. On peut citer par exemple les outils d'assemblage de parties opératives [SCH85] [SCH83] et les outils destinés à l'automatisation de la génération de parties contrôles [CHU82] [DER84].

Le compilateur de silicium SYCO, suite logique du projet CAPRI, a pour but l'automatisation complète du processus de conception.

IV.1.2 Le projet SYCOMORE

Le projet SYCOMORE avait réuni deux industriels, THOMSON et BULL, et deux laboratoires publics, l'INRIA et l'INPG, pour la réalisation d'un système d'aide à la conception de VLSI [SYC84]. Les travaux ont commencé en 1983 et devaient durer 5 ans, avec le soutien de l'Etat.

Le but de SYCOMORE était de réaliser un système informatique permettant de concevoir un VLSI complexe dans un temps d'une semaine à six mois. Le projet se basait sur deux choix qui étaient ambitieux à l'époque, mais qui se sont avérés fondés. Ces choix concernaient la méthodologie de conception et les domaines de représentation.

Concernant les démarches de conception, SYCOMORE devait permettre à la fois :

- Une démarche descendante, qui est nécessaire lors des étapes d'analyse et de synthèse de la conception d'un circuit (ou d'un système). Cette démarche consiste à raffiner la définition des différents blocs qui composent le circuit (ou le système). Elle est en général à l'usage de l'ingénieur système.
- Une démarche ascendante, qui correspond aux étapes de la réalisation. Cette démarche consiste à construire des éléments complexes à partir d'éléments plus fins.

Concernant les domaines de représentation, les trois domaines domaines de description devaient être considérés : le domaine comportemental, le domaine structurel et le domaine géométrique.

Un dernier choix, et non des moindres, portait sur l'utilisation du langage de programmation Le_Lisp avec l'extension CEYX comme environnement logiciel.

Les principaux outils de SYCOMORE ayant fait l'objet de développements sont rapidement décrits ci-après.

Le système HADES [FRE87] est l'assistant du concepteur de circuits aux niveaux logique et électrique. Il est constitué d'un langage de description au niveau structurel et d'un ensemble d'outils intégrés. Le langage permet de décrire un circuit en terme de boîtes interconnectées. Les principaux outils construits autour du langage sont :

- des processeurs de synthèses logique et électrique (passage d'un schéma logique à un schéma électrique),
- un processeur de dessin automatique de structures,
- un noyau de simulation (HERMES) permettant la simulation multi-mode, et une analyse logique des circuits,
- un analyseur temporel hiérarchique, ANATEM,
- un extracteur de schéma logique à partir d'un schéma électrique,
- une interface avec le simulateur électrique SPICE.

Le système STYX [JER85], [ROU87], [CHA88] est un environnement pour la conception de dessins de masques. Il est constitué d'un langage et d'un ensemble d'outils. Le langage permet la description procédurale des dessins de masques. Les objets de base manipulés sont les articulations et les segments. Ils permettent de décrire les éléments de la technologie (transistors, contacts, fils,...). La description d'une cellule est un programme STYX. Le langage STYX est immergé dans Le_Lisp. L'ensemble d'outils qui devaient être développés autour de STYX comporte essentiellement :

- un éditeur graphique,
- un compacteur,
- un extracteur permettant de générer des descriptions HADES,
- une interface "masqueur" (GDS II).

Les deux systèmes HADES et STYX devaient travailler dans un même environnement système et utiliser les mêmes mécanismes de gestion de données. D'autre part, un outil permettant de vérifier la correspondance entre une description STYX et une description HADES devait exister.

Le langage LDS permet la description et la simulation au niveau système. Le noyau du système, construit autour de LDS, comprend un pré-compilateur du langage permettant la génération et la gestion de structures de données LDS et un ensemble d'outils qui sont essentiellement :

- un éditeur de texte orienté LDS,
- un outil d'aide au déverminage (debug) de descriptions LDS,
- un simulateur LDS.

SYCO, l'objet principal de ce chapitre, devait permettre de générer la description STYX d'un circuit en partant de sa description donnée dans un sous-ensemble du langage LDS. Les circuits générés par SYCO peuvent être des composants d'un système plus large décrit en LDS,

ou des composants d'un circuit dont d'autres parties sont générées par d'autres outils.

Le système GEODE intègre un certain nombre d'outils de routage, et est accessible à partir de STYX. Les outils associés à ce système sont essentiellement :

- RIVIERA : Routage mono-couche,
- CORYNTH : Routage bi-couches,
- un éditeur de placement et de routage.

Des outils de test étaient prévus mais n'ont pas fait l'objet de développement.

Malgré l'efficacité de chacun de ces outils pris individuellement, le tout ne forme pas un système intégré pour l'aide à la conception de VLSI. Ce manque d'intégration vient de l'insuffisance de l'environnement logiciel de base utilisé dans SYCOMORE.

En fait, le choix d'un langage de programmation unique ne pouvait, à lui seul, assurer la cohérence des outils développés. Il manquait un environnement système capable d'assurer l'unicité de l'interface homme- machine et l'unicité des mécanismes de gestion de données utilisées par les outils d'aide à la conception.

Il faut rappeler que les choix de base du projet ont été faits en 1983. A cette époque, il n'existait pas de véritable standard graphique, et les stations de travail étaient réalisées sur mesure pour les outils système d'aide à la conception [ROH88]. Il était alors difficile de prévoir l'essor actuel des stations de travail. Il faut aussi rappeler que l'absence d'un système unique pour la gestion des données peut s'expliquer par le fait qu'aucun des SGBD existants ne répondait aux besoins du projet SYCOMORE. Et l'état de l'art (ni les moyen alloués au projet) ne permettait pas d'en définir un.

En fait, si le but unique était de réaliser un système intégré, il aurait mieux valu laisser libre le choix du langage de programmation et imposer un environnement graphique et un gestionnaire de données uniques.

IV.1.3 La première version du compilateur SYCO

La première version de SYCO utilisait comme langage de description le langage IRENE [MAR86]. Ce langage a été conçu pour décrire avec précision la structure et le comportement des circuits. Dans le langage IRENE, la hiérarchie du contrôle est modélisée par une hiérarchie d'horloges. Chaque action est alors rattachée à une horloge et donc à un niveau d'interprétation. En réalité, la description des niveaux d'interprétation est réalisée à plat en un seul module.

On s'était fixé comme contrainte de réaliser un processus de compilation complet

c'est-à-dire d'aller jusqu'au dessin des masques. Cet objectif s'avéra à la fois coûteux mais rentable (voir IV.10). La quatrième section de ce chapitre présente un modèle architectural complet.

Plusieurs outils ont constitué la première version de SYCO. Ils ont surtout permis de valider les principaux choix et de montrer les limites de certains choix techniques. Les principaux outils de cette version sont :

- CIRENE [MAR85] : un compilateur du langage IRENE. Il réalise aussi la partition de la description en niveau d'interprétation.
- CPC : première version [MHA 86] : un compilateur de partie contrôle.
- APOLLON [JAM85], [JAM86a] : un compilateur de partie opérative. APOLLON est également utilisé par la version actuelle de SYCO, il utilise un langage de description propre [JAM86a].

Cette première version du compilateur n'utilise pas une forme intermédiaire comme structure de données commune à tous les outils. Les différents outils communiquent à travers un certain nombre de "fichiers texte" (ASCII), et donc lisibles. Cette version a permis la compilation d'un certain nombre d'exemples de circuits dont le plus significatif fut le MC68000 [GER85].

L'expérience du MC68000 a permis de valider les principaux choix de SYCO. Par contre, elle a montré des faiblesses, et à plusieurs niveaux, du compilateur. La description comportementale a fait apparaître quelques faiblesses dans le sous ensemble du langage IRENE utilisé :

- des instructions de branchement dans le langage ont une sémantique difficile a réaliser. L'instruction "GOTO" permet de spécifier des branchements de n'importe quel état, appartenant à n'importe quel niveau d'interprétation, à n'importe quel autre état [MHA88a].
- La description ne prévoit pas la récupération des comptes rendus d'opérations (voir IV.3.7).
- Le langage ne permet pas la séparation données/contrôle des signaux et des variables (voir IV.3.2.2).
- La description hiérarchique et explicite des horloges est difficile à manipuler.

Le fait que les outils communiquent par l'intermédiaire de structures externes (fichiers), alourdit le traitement. D'autre part, ces fichiers volumineux qui représentent les étapes intermédiaires ne sont pas simulables, et ne permettent donc pas de valider chaque outil séparément. Ces points ont mené à l'utilisation d'une forme intermédiaire comme structure de données unique pour la deuxième version. La forme intermédiaire choisie est de type syntaxique, ce qui lui permet d'être à la fois lisible et simulable.

Le modèle topologique des parties contrôle [VAR87], défini pour une technologie NMOS, était basé sur l'utilisation du système PAOLA [CHU82]. La non disponibilité de ce dernier et donc l'utilisation de générateurs de PLAs classiques ont donné des résultats peu intéressants [VAR87].

L'absence du système PAOLA [CHU82], [CHU84], [PER85] qui devait aussi permettre de modifier la formes des PLAs pour les ramener à la même largeur, a fait apparaître un autre problème. Il s'agit de l'effet pyramide" dans la partie contrôle et de la correspondance entre la largeur de la partie opérative et celle de la partie contrôle [JAM86a].

Dans le cas du MC68000, par exemple, la partie contrôle est composée de 5 niveaux d'interprétation [GER85]. La réalisation des tranches de contrôle par des PLAs, la seule solution offerte par la première version de SYCO, entraîne la génération d'une partie contrôle non régulière. La figure IV.1 montre la complexité et la taille des différents étages de contrôle. Les tailles relatives des différents blocs sont montrées par la figure IV.2.

Les chiffres de la figure IV.1 résultent d'une évaluation basée sur un modèle défini dans [GER85]. Les chiffres concernant la partie opérative sont exacts car les dessins des masques ont pu être générés. Ces évaluations ont été suffisantes pour montrer l'irrégularité de la forme des différents étages qui constituent le circuit (voir figure IV.2).

ETAGE	ENTREES	SORTIES	MONOMES	LARGEUR	HAUTEUR
5	7	6	20	182	181
4	16	10	76	613	347
3	16	10	269	2099	347
2	23	13	205	1606	475
1	34	138	602	3969	1715
Partie opérative				2255	864

Figure IV.1 : Complexité et dimensions (en Lambda pour une technologie NMOS) des différents étages qui composent le MC68000 [GER85]

La taille des blocs peut être ajustée par des modifications de la description comportementale ; étant donnée la taille de la description (50 pages dans le cas du MC68000) et les risques d'erreurs, il a été nécessaire de définir un outil d'aide à la modification de la description. Cet outil, appelé ARTS, est détaillé dans [BEK87] (voir aussi IV.9).

Pour finir, il faut noter que l'expérience du MC68000 a permis de briser deux mythes contradictoires. Il concernent la faisabilité des compilateurs de comportement et les miracles qui devaient être réalisés par ces derniers. Cette expérience a montré la faisabilité de ce type d'outils. Mais elle a montré aussi que la conception d'un circuit, même au niveau

comportemental, nécessite une définition bien précise du circuit, ce qui entraîne une description volumineuse et difficile à mettre au point.

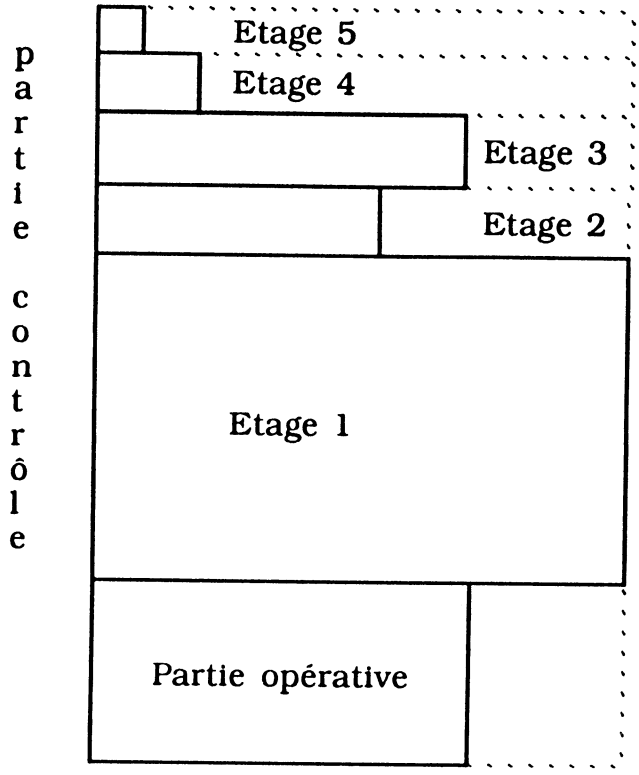


Figure IV.2 Plan de masse du MC68000

IV.2 Les choix de base du compilateur SYCO

Comme il a été dit à plusieurs reprises dans les trois premiers chapitres, un compilateur de silicium ne peut explorer tout l'espace des solutions en partant d'une description comportementale. Il faut donc limiter l'espace des solutions en imposant des restrictions à tous les niveaux du processus de compilation, de la description comportementale à la description physique.

Dans le cas du compilateur SYCO le choix des restrictions s'est fait en deux temps. Le choix de la méthodologie a imposé le modèle général de compilation, à savoir le découpage en niveaux d'interprétation [ANC 83]. Les choix des autres modèles ont été mis au point avec la réalisation de la première version de SYCO.

Les modèles utilisés par le compilateur SYCO ont été choisis pour réaliser un but principal qui est la compilation de circuits de commande complexes. Deux contraintes ont aussi guidées le choix de ces modèles :

- Pour valider les différents modèles, il faut aboutir à la génération des dessins des masques.
- Pour générer des circuits efficaces, il faut se donner les moyens pour pouvoir optimiser les résultats du compilateur SYCO.

Le reste de cette section présente brièvement les différents modèles utilisés par SYCO. La plupart de ces modèles seront détaillés dans des sections ultérieures.

IV.2.1 Modèle de compilation

Le processus de compilation est basé sur le modèle des niveaux d'interprétation [OBR82], [ANC83]. Un circuit de commande réalise un algorithme d'interprétation. Le principe de la compilation consiste à produire un ensemble d'interpréteurs qui, assemblés, réalisent la même fonction que l'interpréteur spécifié par la description de départ, une description algorithmique.

Ce principe peut être formalisé de la manière suivante : l'interpréteur d'un langage (Ln) peut être décomposé en (n) interpréteurs travaillant sur (n) langages (L0, L1, ..., Ln). L'interpréteur de niveau (i) accepte la langage (Li) et génère un langage de niveau (Li - 1). Les primitives du langage de niveau (0) constituent les opérations de base de la machine.

Chacun de ces interpréteurs sera réalisé par un automate. On distingue deux types d'automates :

- les automates de contrôle (tranches de parties contrôle) : ils transforment une commande de haut niveau en une séquence de commandes de niveau inférieur,

- les automates d'exécution (parties opératives) : ils réalisent les opérations de base.

Pour aller d'une description comportementale au dessin des masques, d'autres modèles sont nécessaires. L'utilisation de modèles prédéfinis pour l'organisation architecturale et pour l'organisation du dessin des masques permet la réalisation d'un processus de compilation efficace. Par contre, ces modèles prédéfinis limitent les performances des circuits générés par SYCO et limitent aussi son domaine d'application. Néanmoins il existe des domaines, comme la robotique par exemple, où le coût de conception des circuits est déterminant. Le système de commande d'un robot mobile, par exemple, nécessite l'embarquement d'un système informatique complexe. Des contraintes de consommation, de coût, de fiabilité, voire de rapidité impliquent le recours à l'intégration sur silicium. L'exemple type de fonctions intégrables sont les post-processeurs pour les capteurs, et les pré-processeurs pour les actionneurs. En général un capteur (par exemple une caméra) retourne une grande masse d'informations brutes (par exemple une image), l'utilisation d'un processeur spécialisé permettrait de réduire la quantité d'informations en ne communiquant que les informations utiles (contours, histogrammes,...).

IV.2.2 Description du comportement

Le compilateur SYCO utilise un sous-ensemble d'un langage, de description de matériel, existant appelé LDS. Ce langage a été défini et est utilisé par BULL [LAU85]. Le sous-ensemble utilisé est bien adapté à l'architecture cible utilisée par SYCO. La spécification d'un circuit contient :

- l'interface du circuit : les signaux externes d'entrées/sorties du circuit.
- la fonction du circuit : elle décrit l'algorithme d'interprétation que doit réaliser le circuit.

Cette description définit le flux de contrôle de manière hiérarchique et les opérations parallèles. Le langage d'entrée du compilateur SYCO sera plus détaillé dans la suite.

IV.2.3 Architecture cible

Les circuits générés par SYCO réalisent un algorithme d'interprétation. Comme il a été dit plus haut, le principe de la compilation consiste à générer une pile d'interpréteurs qui, assemblés, réalisent l'algorithme d'interprétation original [ANC86]. Chacun des interpréteurs de la pile décompose les commandes du voisin supérieur en primitives exécutables par son voisin inférieur. Le premier et le dernier étage de cette pile jouent des rôles particuliers. Le premier étage constitue la partie opérative, il exécute les opérations de base réalisées par le circuit. L'étage le plus haut assure le séquencement global du circuit. La partie contrôle du circuit est constituée par les étages de la pile autres que le premier.

Les étages de la partie contrôle communiquent à travers un bus de contrôle et utilisent un bus de comptes rendus alimenté par la partie opérative (figure IV.3). Le bus de contrôle permet aussi de communiquer avec l'extérieur. La synchronisation entre les différents étages, ainsi que leur organisation interne seront détaillées plus loin (IV.4)

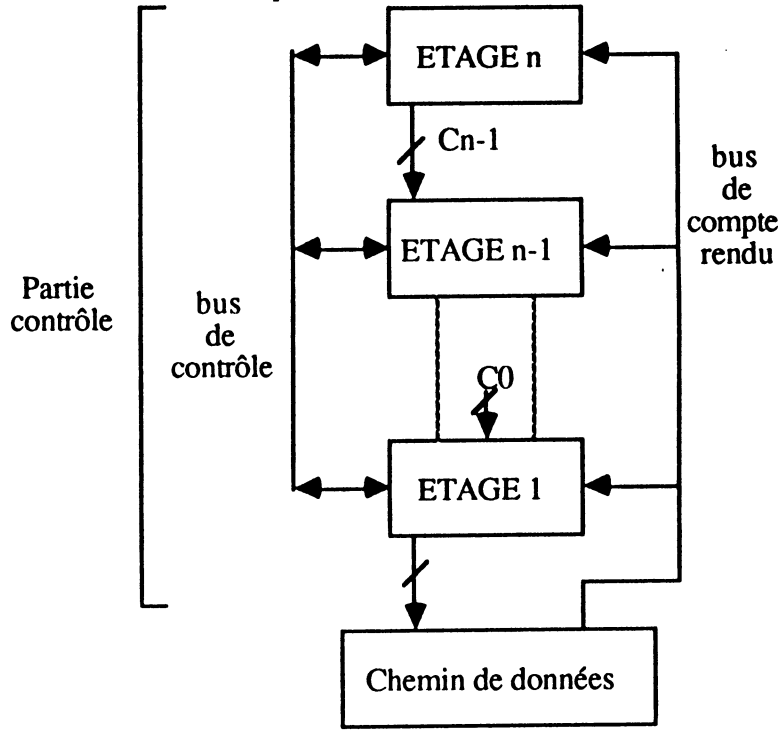


Figure IV.3 : Modèle architectural

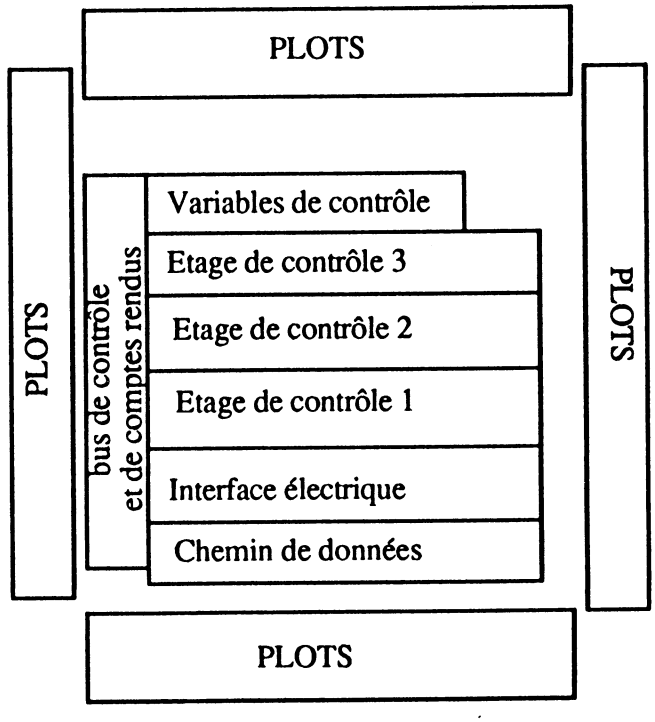


Figure IV.4 : Modèle topologique

IV.2.4 Stratégie d'implantation

Le dessin des masques du circuit est aussi composé d'une pile de blocs, constituée par une partie opérative et plusieurs tranches de contrôle (figure IV.4). Ces tranches sont générées par deux compilateurs spécialisés : APOLLON [JAM85] pour la tranche de la partie opérative et CPC [MHA87] pour les tranches de la partie contrôle. Les différentes tranches sont conçues pour être assemblées de manière simple.

L'utilisation d'une stratégie d'implantation permet la définition d'un modèle pour l'évaluation de la surface d'un circuit en partant de la description comportementale [BEK87] (voir aussi IV.9).

IV.2.5 Processus de compilation

Le processus de compilation se décompose en trois étapes. La première génère l'architecture globale du circuit (figure IV.3) en partant de la description comportementale. Les descriptions des différentes tranches sont données dans le langage LDS. Cette étape est réalisée par un outil appelé EXTRACT qui utilise la hiérarchie contenue dans la description originale pour fournir une première solution de découpage en niveaux d'interprétation. L'algorithme de découpage sera détaillé plus loin (voir IV.6). Ce découpage peut être modifié en vue d'améliorer les performances du circuit généré à l'aide des outils d'optimisation architecturale fournis par ARTS [BEK87].

La seconde étape met en œuvre deux compilateurs spécialisés pour générer l'architecture détaillée des différentes tranches de contrôle et l'architecture de la partie opérative.

La troisième étape génère le dessin des masques du circuit à l'aide de modules générateurs. Ces modules générateurs prennent en compte la stratégie d'implantation globale. Le schéma de fonctionnement du compilateur SYCO est donné en la figure IV.5.

Les deux premières étapes utilisent la même représentation, une forme intermédiaire syntaxique (voir IV.4), cette dernière peut être simulée au niveau comportemental.

L'algorithme de traduction de SYCO est simple. En effet, l'architecture cible facilite grandement la traduction. Le modèle adopté fournit à la fois des modèles architecturaux et des modèles topologiques, ce qui entraîne une simplification des trois étapes de la compilation. L'étape de génération de l'architecture globale est simplifiée par l'utilisation d'une architecture cible. L'utilisation de modèles prédéfinis, pour l'architecture des tranches de contrôle et celle de la partie opérative, réduit la complexité des compilateurs spécialisés. Finalement, l'adoption

d'une stratégie d'implantation (modèle topologique) simplifie l'étape de génération de dessin des masques.

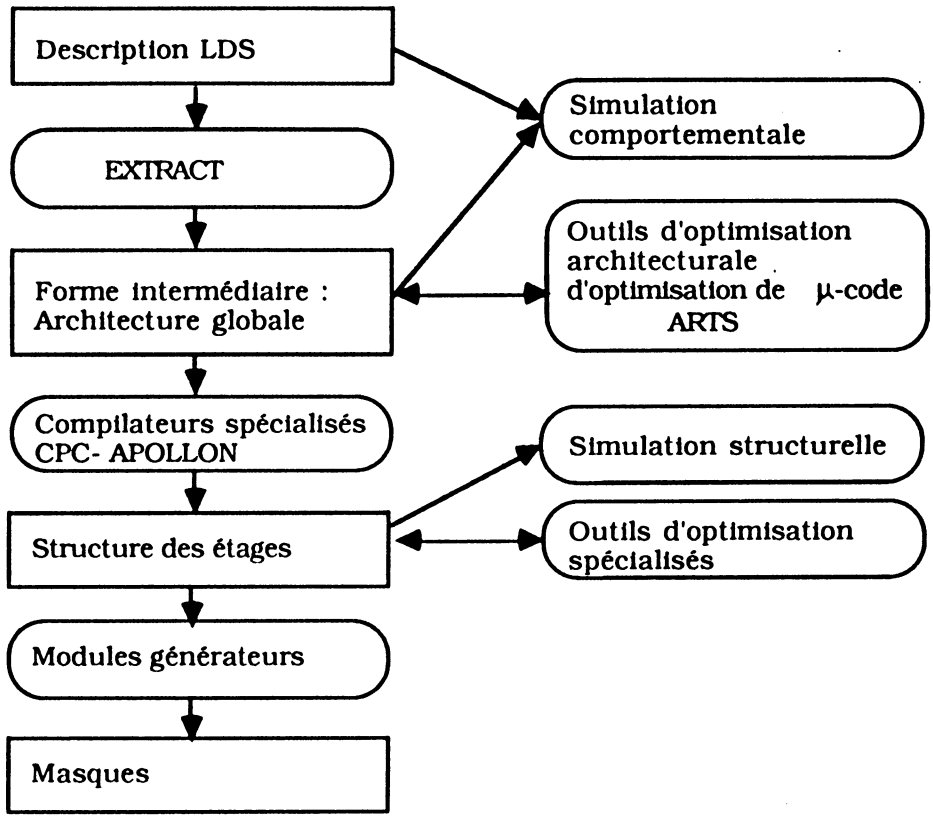


Figure IV.5 : Schéma de fonctionnement du compilateur de silicium SYCO

L'une des originalités du compilateur SYCO vient de la correspondance entre le modèle architectural et la description comportementale. Le processus de compilation utilise la hiérarchie de la description pour générer les niveaux d'interprétation et le parallélisme spécifié dans la description pour générer le parallélisme de la partie opérative. Ainsi, le concepteur peut contrôler les performances des circuits générés, à travers la description d'entrée. De plus, plusieurs solutions architecturales peuvent être essayée grâce à l'outil d'optimisation architecturale ARTS [BEK87] qui permet de modifier le parallélisme et/ou la hiérarchie.

La correspondance entre la description comportementale et l'architecture des circuits générés par SYCO peut rappeler les compilateurs d'architectures, où la description d'entrée du compilateur définit l'architecture du circuit à compiler. Cette similitude n'est qu'apparente car, dans le cas d'un compilateur d'architecture, le découpage du circuit (en sous-systèmes ou blocs) qui est défini par la description d'entrée ne peut être modifié par le compilateur. D'autre part, la description d'architecture fixe également les interfaces entre les différents blocs qui constituent le circuit. Dans le cas du compilateur SYCO, la description d'entrée définit les grands choix architecturaux (hiérarchie de la partie contrôle et parallélisme de la partie opérative)

mais elle ne définit pas complètement l'architecture du circuit. Par exemple, dans le cas de la partie opérative, la description comportementale manipule des opérations et des registres, et c'est le compilateur qui va déterminer les opérateurs ou boîtes à opérations (unités capables d'exécuter plusieurs opérations comme une Unité Arithmétique et Logique) ainsi que les interconnexions entre ces éléments. Le compilateur génère aussi les éléments de communication entre différents blocs du circuit. D'autre part, l'architecture initiale, générée à partir de la description comportementale, n'est pas définitive car elle peut être modifiée à l'aide des outils d'optimisation architecturale

IV.3 Description comportementale

Le langage d'entrée du compilateur doit permettre de décrire l'algorithme de fonctionnement d'un circuit. Le concepteur d'un microprocesseur, par exemple, 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.

Comme il a été dit plus haut, SYCO utilise un sous-ensemble du langage de description de matériel LDS, défini en fonction de l'architecture cible et des algorithmes de compilation.

IV.3.1 Le langage LDS

LDS (Langage de Description de Systèmes) a été développé par BULL [LAU85], [NGU86], [BEL87] dans le cadre du projet SYCOMORE. Comme pour la plupart des langages de description de matériel, la définition du langage LDS a été surtout guidée par les outils de vérification. Plusieurs aspects de ce type de langages ne sont pas compilables (voir III.3). Cette présentation du langage LDS a pour but de justifier le choix du sous-ensemble du langage utilisé par SYCO. Il faut noter que certaines caractéristiques de LDS, telles que le typage données/contrôle des variables, ont été introduites spécialement pour les besoins de SYCO.

La description d'un système en LDS est structurée en plusieurs sous-systèmes qui à leur tour peuvent être décrits de manière comportementale et/ou structurelle.
Un sous-système correspond à un "modèle" dans LDS. Un modèle est décrit par :

- la définition de l'interface entre le sous-système et le monde extérieur.
- la description de la fonction interne du modèle. Elle peut être structurelle ou comportementale.

Dans le cas d'une description structurelle, le sous-système est décrit comme un emboîtement hiérarchique d'autres modèles. La description définit aussi les liaisons entre les différents modèles à l'aide des signaux d'interface.

Dans le cas d'une description comportementale, la fonction du sous-système est exprimée par un programme composé d'un ensemble d'instructions qui sont interprétées de manière procédurale.

Les ressources d'un modèle peuvent être soit des éléments de mémorisation, soit des signaux.

Pour la simulation d'un modèle, il est indispensable de représenter le temps de manière explicite. Pour cela, le langage permet la définition d'horloges qui sont globales à tout le système. Le temps est modélisé par une succession chronologique de tous les fronts montants

des différentes horloges. Les différents modèles du système sont synchronisés par ces horloges. Dans le cas d'une description structurelle, les changements d'états des ressources sont synchronisés sur des fronts d'horloges. Dans le cas d'une description comportementale, l'exécution de toutes les instructions est instantanée, le séquencement étant défini explicitement par des instructions de contrôle.

L'ensemble des instructions du langage contient :

- des instructions itératives : WHILE, FOR,
- des instructions conditionnelles : IF, CASE,
- l'instruction d'attente, WAIT, provoque l'attente d'une expression ou l'attente d'un front d'horloge.
- des instructions de branchement : CAUSE, EXECUTE, NEXT, EXIT, CALL.

L'instruction CAUSE réalise un branchement parallèle à un bloc d'une description comportementale. L'exécution de ce bloc est déclenchée sans bloquer l'exécution du bloc courant. L'instruction EXECUTE réalise un branchement séquentiel à un bloc comportemental. Le bloc appelant est mis en attente jusqu'à la fin du traitement du bloc appelé. Elle est équivalente à l'instruction d'appel de sous-programmes dans les langages de programmation procéduraux. L'instruction EXIT redonne le contrôle au bloc appelant. L'instruction NEXT indique un branchement à une autre instruction du bloc courant (GOTO). L'instruction CALL réalise un branchement à une fonction (voir plus loin).

Dans le langage LDS, une fonction peut être un opérateur particulier ou un sous-système ou même un programme écrit dans un langage externe tel que C. Dans ce dernier cas la communication avec la fonction se fait par des variables et non par des signaux.

Cette présentation du langage LDS ne tient pas compte des évolutions récentes du langage qui devaient le rapprocher du langage VHDL [VHD87]. Ces évolutions ne présentent pas beaucoup d'importance pour le présent travail car le sous-ensemble du langage LDS utilisé par SYCO est aussi un sous-ensemble du langage VHDL, si on ne tient pas compte des détails syntaxiques.

IV.3.2 Les sous-ensembles de LDS utilisés par SYCO

SYCO utilise un sous-ensemble du langage LDS. Ce sous-ensemble correspond à ce que l'on sait compiler. Pour être compilable une description doit obéir à trois types de restrictions imposées par SYCO :

• Restrictions concernant l'organisation de la description : l'unité de compilation pour SYCO est un modèle défini par son comportement. SYCO ne permet donc pas de compiler une description structurelle.

• Restrictions concernant les instructions utilisées : SYCO ne permet de traiter qu'une partie des instructions de description du comportement. Le sous-ensemble est défini plus loin, il exclut tout ce qui peut introduire un comportement dynamique.

• Restrictions concernant l'interprétation des instructions : elles seront expliquées dans les deux sections concernant l'architecture (IV.4) et la forme intermédiaire (IV.5). Elles se résument en deux points :

•• l'interprétation parallèle des instructions conditionnelles : dans le cas du simulateur LDS, les branches d'une instruction conditionnelle sont évaluées séquentiellement. Elles peuvent contenir des boucles d'attentes. Pour SYCO, chaque instruction de contrôle est exécutée en un seul cycle.

•• le séquencement : pour SYCO le séquencement peut être défini soit explicitement par les instructions de branchement, soit implicitement par l'ordre d'écriture des instructions. Dans le cas du simulateur LDS, le séquencement est défini explicitement par les instructions WAIT et CAUSE.

Une description compilable, au sens de SYCO, ne contient pas de définition explicite d'horloge. SYCO utilise un modèle d'horloge standard qui est défini avec l'architecture. On peut remarquer qu'une description compilable n'est pas simulable en tant que telle. Par contre, SYCO peut générer automatiquement une description où le séquencement est défini par les horloges, et qui est donc simulable. Le même problème a été reporté plus récemment par CAMPOSANO et al. [CAM88] dans le cas de la compilation de VHDL.

IV.3.2.1 Organisation d'une description

Une description est composée de deux parties : une partie déclaration et une description de comportement. Le bloc déclaration est appelé SMODULE. Il contient la définition de l'interface et des ressources. La description comportementale décrit l'algorithme d'interprétation réalisé par le circuit. Elle peut être hiérarchique. Chaque bloc de cette description est appelé CMODULE. La description algorithmique peut utiliser des fonctions externes appelées FMODULEs.

La figure IV.6 montre la description d'un microprocesseur. Il s'agit d'une version simplifiée du microprocesseur décrit dans [ANC86]. Une autre version est donnée par [JAM86]. Le circuit décrit par la figure IV.6 permet de réaliser cinq instructions (lda, sta, ada, bru, bneg).

```
1 SMODULE mini ( adbus , dtbus , dtack , restart ,
2                adstrobe , dtstrobe , read_write ) ;
3 VARIABLE ;
4 r 0:2 , CTRL ; ? Variable de contrôle interne
5       ? paramètre de la procédure fetch
6 END ;
7 SIGNAL ;  ? signaux d'entrées/sorties
8  adbus 0:8 , OUT ; ? bus d'adresse
9  dtbus 0:8 , INOUT ; ? bus de données
10  dtack , IN , CTRL ; ? signal d'acquitement
11  restart , IN , CTRL ; ? Reset logique
12  adstrobe , OUT , CTRL ;
13  dtstrobe , OUT , CTRL ;
14  read_write , OUT , CTRL ;
15 END ;
16 REGISTER ;  ?Variables de données internes
17  a 0:8 ; ?accumulateur
18  b 0:8 ; ?
19  ir 0:8 ; ? registre instruction
20  pc 0:8 ; ?compteur ordinal
21 END ;
22 LINK mini ; ? Liaison sur le CMODULE principal
23            ? de la description comportementale
24 END ;
25 CMODULE fetch ;
26 <send> ( adbus := pc ; read_write := 1 ; )
27 <>     ( adbus := pc ; adstrobe := 1 ; NEXT receive ; )
28 <receive> IF ( dtack = 0 )  adbus := pc ; NEXT receive ;
29         ELSE
30           CASE ( r )
31         WHEN ( 0 ) ir := dtbus ;
32         WHEN ( 1 ) a := dtbus ;
33         WHEN ( 2 ) b := dtbus ;
34           END ;
35           pc := pc + 1 ; adstrobe := 0 ;
36         END ;
37 END ;
38 CMODULE sta ;
39 <> ( r := 2 ; EXECUTE fetch ; )? Lire l'adresse (b)
40    ( adbus := b ; read_write := 0 ; dtbus := a ; )
41    ( adbus := b ; adstrobe := 1 ; dtbus := a ; dtstrobe := 1 ; )
42 <ack> IF ( dtack = 0 )  adbus := b ; dtbus := a ; NEXT ack ;
43       ELSE  adstrobe := 0 ; dtstrobe := 0 ;
44    END ;
45 END ;
46 CMODULE ada ;
47 <> ( r := 2 ; EXECUTE fetch ; )? lire (b)
48      a := a + b ;
49 END ;
50 CMODULE lda ;
51 <> ( r := 1 ; EXECUTE fetch ; ) ?lire (a)
52 END ;
53 CMODULE bru ;
54 <>  ( r := 2 ; EXECUTE fetch ; )?Lire l'adresse (b)
55      pc := b ;
56 END ;
57 CMODULE bneg ;
58 <>  ( r := 2 ; EXECUTE fetch ; ) ?Lire l'adresse (b)
59    IF ( a 7:1 = '1' ) pc := b ; END ;
60 END ;
61 CMODULE mini ;
62 <rest> IF ( restart = 0 ) NEXT rest ; ELSE NEXT start ; END ;
63 <start> pc := 0 ;
64 <newinstr>( r := 0 ; EXECUTE fetch ; )
65 <decode> ( CASE ( ir 0:4 )
66          WHEN ( '0000' ) EXECUTE ada ; ? a <- a + dtbus
67          WHEN ( '0001' ) EXECUTE lda ; ? a <- dtbus
68          WHEN ( '0010' ) EXECUTE sta ; ? memory <- a
69          WHEN ( '0101' ) EXECUTE bneg ; ? branch si a < 0
70          WHEN ( '0110' ) EXECUTE bru ; ? branch inconditionnel
71        END ;
72        IF ( restart = 0 ) NEXT rest ; ELSE NEXT newinstr ; END;)
73 END ;
```

Figure IV.6 - Description d'un microprocesseur simplifié

IV.3.2.2 Description de l'interface et des ressources

Le bloc de déclaration (SMODULE) contient la description de l'interface externe du circuit, c'est-à-dire des signaux d'entrées/sorties. Ces signaux correspondent à une partie des plots du circuit. De plus, SYCO rajoute automatiquement quelques plots (voir IV.4). La partie déclaration contient aussi la description des variables utilisées par l'algorithme d'interprétation. Parmi les variables utilisées, on distingue 2 types de variables :

- les variables de type données : les registres de la partie opérative.
- les variables de type contrôle : ils sont situés dans la partie contrôle et servent à mémoriser les requêtes de contrôle externes et certains signaux de contrôle interne.

Les signaux d'entrées/sorties sont aussi classés en deux types. Les signaux de données décrivent en général les bus d'adresses et les bus de données, ils seront connectés aux bus de la partie opérative via des tampons. Les signaux de contrôle seront connectés au bus de contrôle

de la même façon.

Cette distinction entre variables de contrôle et variables de données a été introduite dans LDS pour les besoins du compilateur SYCO. En fait, parmi les variables et signaux manipulés par les circuits séquentiels on distingue trois types :

- les éléments manipulés uniquement par la partie opérative, pour les entrées/sorties et le calcul interne.
- les éléments manipulés uniquement par la partie contrôle.
- les éléments manipulés à la fois par la partie opérative et la partie contrôle, tel qu'un registre instruction.

Cette distinction est importante car elle permet de placer les premiers dans la partie opérative, les seconds dans la partie contrôle et de partager les troisièmes. D'autre part, les variables de contrôle portant généralement sur un faible nombre de bits, leur insertion dans la partie opérative peut affecter la régularité de celle-ci. Le langage VHDL, par exemple, ne fait pas cette distinction. Des travaux récents [LIS88] sur la compilation de comportement, en partant de VHDL, proposent d'introduire des distinctions entre les signaux de données et les signaux de contrôle.

Dans le cas de la figure IV.6, la partie déclarations (lignes 1 à 24) est composée d'un en-tête (lignes 1 à 2), spécifiant que le circuit s'appelle "mini", qu'il utilise 7 signaux externes : adbus, dtbus, dtack, restart, adstrobe, dtstrobe, read_write (lignes 8 à 14), une variable de contrôle interne (ligne 4) et 4 registres de 8 bits (lignes 17 à 20).

IV.3.2.3 Description de l'algorithme d'interprétation

L'algorithme d'interprétation est donné sous forme d'une hiérarchie de CMODULEs. Chacun des CMODULEs décrit une procédure de l'algorithme d'interprétation. Un CMODULE décrit une machine d'états finis, il est constitué par une suite d'états. Un état correspond à un cycle de contrôle, il contient une suite d'instructions (ou actions) devant se dérouler en parallèle. L'instruction EXECUTE du langage LDS permet à un CMODULE d'en appeler un autre. Les appels récursifs sont interdits. Un CMODULE est généralement décrit sous la forme :

```
CMODULE <NOM_MODULE>
<NOM> (<liste d'actions>)
   ......
<NOM> (<liste d'actions>)
END.
```

En pratique, le format d'entrée est plus libre. Les états ne sont pas obligatoirement délimités. Les labels <NOM> permettent d'identifier les différents états. Ils peuvent être omis dans le cas des états auxquels il n'existe pas de branchement explicite.

Les actions peuvent être conditionnelles (IF, CASE) ou inconditionnelles. Parmi les actions inconditionnelles, on distinguera les affectations : actions opératives (transferts, opérations) et actions de contrôle qui permettent d'affecter des valeurs aux variables de contrôle et les actions de séquencement : le branchement à l'intérieur d'un CMODULE (NEXT) et le retour au CMODULE appelant (EXIT).

Le sous-ensemble utilisé par SYCO ne comporte pas les instructions de boucles, l'instruction de branchement parallèle CAUSE ni l'instruction WAIT. D'autre part, une description ne contient pas d'horloge.

Les instructions de boucles peuvent introduire des ambiguïtés pour l'interprétation parallèle des instructions conditionnelles car une boucle peut nécessiter plus d'une étape de contrôle pour s'exécuter. Ce problème existe uniquement en cas d'imbrication des instructions conditionnelles et des instructions de boucles. Ceci vient du fait que, pour SYCO, les branches des instructions conditionnelles sont interprétées de manière parallèle.

Exemple : l'instruction

```
CASE (r)
    WHEN (0) ir : = dtbus ;
    WHEN (1) a : = dtbus ;
    WHEN (2) b : = dtbus ;
END;
```

est équivalente à trois instructions IF parallèles (en LDS, des instructions parallèles sont écrites entre parenthèses) :

```
(IF (r = 0) ir : = dtbus ; ENDIF;
 IF (r = 1) a : = dtbus ; ENDIF;
 IF (r = 2) b : = dtbus ; ENDIF;)
```

et l'instruction :

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

est équivalente à deux instructions IF parallèles :

```
(IF (restart = 0) NEXT rest ; ENDIF
IF (restart <> 0) NEXT start ; ENDIF)
```

Le sous-ensemble de LDS utilisé par SYCO permet la définition de boucles de contrôle à l'aide de l'instruction de branchement NEXT. Pour réaliser des boucles à l'intérieur d'une instruction conditionnelle il faut les claustrer dans des CMODULEs. Cette claustration pourrait être réalisée de manière automatique.

L'instruction de branchement CAUSE a été interdite car elle crée des situations équivalentes à l'imbrication des blocs série/parallèle (voir III-3).

Les actions qui se déroulent en parallèle forment une étape (ou un cycle) de contrôle. Le déroulement complet d'une étape peut nécessiter plusieurs cycles de base, en raison de l'organisation hiérarchique de l'algorithme d'interprétation. L'utilisation d'une fonction externe peut entraîner le découpage des cycles de contrôle en plusieurs cycles de base.

L'ordre d'exécution des étapes est fixé par les instructions de séquencement contenues dans la description. Dans le cas où tous les branchements ne sont pas spécifiés explicitement, les branchements implicites sont calculés automatiquement et rajoutés à la description.

Dans le cas de la figure IV.6, l'algorithme d'interprétation du circuit "mini" est constitué de sept CMODULEs. Le CMODULE "fetch", par exemple, (lignes 25 à 37), décrit un accès en lecture à une mémoire. Les deux premières étapes (lignes 26 à 27) transfèrent l'adresse (contenu du registre "pc") au bus d'adresse ("adbus") et positionnent les signaux de contrôle nécessaires au protocole de communication avec la mémoire. La troisième étape (lignes 28 à 36) est une boucle qui attend que la donnée soit disponible (cf. signal "dtack"). La donnée est lue et transférée dans un registre choisi selon la valeur de la variable "r". La même étape incrémente le registre "r".

L'exemple de la procédure fetch montre les capacités du language pour la description des protocoles de communication d'un circuit avec le monde extérieur. La lecture de la mémoire est décrite avec une précision qui peut être celle d'une description architecturale. Cette précision est nécessaire pour la description des protocoles de communication "sur-mesure". D'autres techniques peuvent être utilisées pour la description des protocoles de communication (voir III.3.6).

IV.3.2.4 Les fonctions externes

Le langage LDS permet l'utilisation de deux types de fonctions. Les premières peuvent être décrites dans un langage de programmation quelconque. Elles retournent une valeur. Les secondes sont décrites à l'aide des FMODULEs. Un FMODULE décrit une unité matérielle qui communique avec l'extérieur via des signaux.

L'origine de cette notion de FMODULE vient de la notion de fonction du langage IRENE [MAR86]. Elle permet l'utilisation d'une unité matérielle dans une description comportementale. Cette notion est utilisée dans SYCO pour définir les boîtes à opérations, les opérateurs externes.

Dans la version actuelle il existe un seul FMODULE reconnu, il est appelé ALU-STD. Il peut être invoqué par l'ordre "call". L'ordre :

Call ALU-STD (R1, R2, R3, R4, +);

est équivalent aux deux actions :

- R3 := R1 + R2
- R4 := Indicateurs arithmétiques de l'UAL.

La figure IV.7 montre une description LDS d'une unité capable d'exécuter une addition et une soustraction.

```
FMODULE plus_moins (result, flags, reg1, reg2, oper);
REGISTER;
        result 0:8 ,OUT; flags 0:8 ,OUT;
        reg1 0:8 ,IN; reg2 0:8 ,IN;
END;
VARIABLE;
        oper CHAR,IN;
END;
CASE (oper)
        WHEN ("+")
          flags 0:1:= carry(reg1 + reg2) ;
          flags 1:1:= overflow(reg1 + reg2) ;
          flags 2:1:= zero(reg1 + reg 2) ;
          flags 3:1:= neg(reg1 + reg2) ;
          result := reg1 + reg2;
          END;
        WHEN ("-")
          flags 0:1:= carry(reg1 - reg2) ;
          flags 1:1:= overflow(reg1 - reg2) ;
          flags 2:1:= zero(reg1 - reg2) ;
          flags 3:1:= neg(reg1 - reg2) ;
          result := reg1 - reg2;
          END;

END
END;
```

Figure IV.7 : Exemple de description d'une fonction

Les fonctions "carry", "overflow", "zéro" et "neg" sont prédéfinies en LDS. Elles permettent de calculer les effets de bord des opérations arithmétiques.

Comme il a été dit en III.3, l'utilisation des fonctions externes permet d'introduire de nouveaux opérateurs dans le langage. Pour qu'un opérateur externe puisse être utilisé dans SYCO, il faut, en plus du FMODULE, définir les autres aspects de la boîte à opérations correspondante. Le FMODULE décrit la manière dont l'opérateur peut être invoqué, et donne un modèle de simulation.
La description complète d'un opérateur contient aussi :

- La description architecturale de l'opérateur, pour pouvoir le connecter aux autres blocs du circuit. Cette description contient des protocoles de communication de la boîte à opérations : liste des opérateurs, code opération pour chaque opérateur, nombre de cycles

machine pour chaque opération, ...

• Les caractéristiques physiques de l'opérateur. Elles seront utilisées par les outils d'évaluation de performance. Elle peuvent être aussi utilisées par les outils de compilation et de vérification des autres niveaux de compilation (architecture, structure).

La section IV.8 donne d'autres détails sur l'utilisation des fonctions externes dans le compilateur SYCO.

Indépendamment de la syntaxe du langage LDS, le sous-ensemble utilisé par SYCO peut être trouvé dans tout langage de description. Mais, dans tous les cas, la description ne sera pas directement simulable, à cause de l'interprétation faite des instructions conditionnelles et du séquencement. Par contre, la génération d'une description simulable ne présente aucune difficulté

IV.4 Architecture des circuits générés par SYCO

L'architecture d'un circuit détermine sa structure interne et les mécanismes de communications entre les différentes parties du circuit, et entre le circuit et le monde extérieur. Vu que le compilateur SYCO part d'une description comportementale, il doit générer lui-même l'architecture du circuit. Comme il a été dit plus haut, le problème de génération automatique d'architecture est difficile à résoudre. Il faut donc utiliser des architectures cibles. L'architecture cible du compilateur SYCO est souple, elle permet de générer des circuits avec des parties contrôle hiérarchiques.

IV.4.1 Architecture globale

Les circuits générés par SYCO sont composés d'une partie contrôle et d'une partie opérative. La figure IV.8 donne l'organisation globale d'un circuit généré par SYCO.

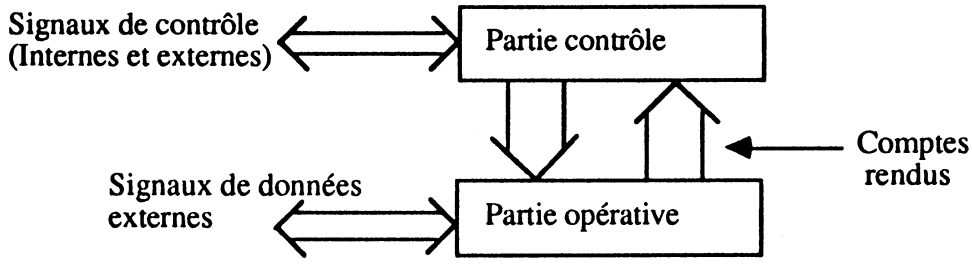


Figure IV.8 Organisation globale d'un circuit

Les circuits communiquent avec l'extérieur à travers trois types de signaux :

- des signaux d'entrée (IN),
- des signaux de sortie (OUT),
- des signaux bi-directionnels (INOUT).

Ces signaux seront rattachés aux plots d'entrées/sorties dans le cas de la compilation d'un circuit entier, et/ou à d'autres parties du circuit dans le cas de la compilation d'une partie du circuit.

En plus des signaux de contrôle déclarés par l'utilisateur, SYCO génère automatiquement les signaux nécessaires à la réinitialisation et à la synchronisation des différents blocs du circuit. Ces signaux (horloges et reset) seront distribués dans tout le circuit.

IV.4.2 Les signaux de communication et de synchronisation

IV.4.2.1 Les horloges

Les circuits générés utilisent une seule horloge externe pour synchroniser et contrôler le séquencement des différents éléments du circuit. Cette horloge de base est décomposée en quatre phases (T1, T2, T3, T4) qui forment le cycle de base de la machine.
Un cycle de base correspond au temps d'exécution d'un transfert dans la partie opérative. La période de l'horloge dépend de la bibliothèque de cellules utilisée et de la complexité du circuit compilé.

IV.4.2.2 Le "reset"

Ce signal est propagé dans tous les blocs du circuit. Quand il est actif, il remet à zéro toutes les variables internes, et initialise les différentes tranches du circuit. Après un "reset", le circuit reprend l'exécution de l'algorithme d'interprétation au départ (premier état de la procédure principale).

Les signaux d'horloge et de "reset" ne sont pas accessibles à l'intérieur de l'algorithme d'interprétation. Ils sont générés automatiquement par le compilateur SYCO.

IV.4.2.3 Les plots

A chaque bit des signaux de communication (contrôle et données) le compilateur génère une connexion à un plot. Comme dans le cas des signaux de communication, on distingue trois types de plots (IN, OUT, INOUT).

IV.4.3 Architecture de la partie contrôle

La partie contrôle est constituée d'une pile de tranches de contrôle (figure IV.9). La tranche (i) exécute les instructions générées par la tranche (i+1). Elle découpe une instruction en "sous-instructions" qui seront exécutées par la tranche (i-1). Une instruction correspond à un cycle de contrôle.

Le choix d'une partie contrôle hiérarchique résulte de l'ensemble des travaux de l'Equipe d'Architecture des Ordinateurs de l'IMAG [OBR82], [ANC86]. Ces travaux comparent plusieurs architectures de partie contrôle. Ils montrent que dans le cas du microprocesseur MC6800™, par exemple, le meilleur résultat est obtenu dans le cas d'une architecture à deux niveaux d'interprétation. La réalisation de circuits plus complexes peut nécessiter plus que deux niveaux d'interprétation. Il faut remarquer que l'augmentation du nombre des niveaux

d'interprétation pour un circuit donné permet généralement de diminuer sa surface. Le microprocesseur MC68000 (et tous les circuits du même type : µ370, µVAX, ...) contient une partie contrôle organisée en trois niveaux. Le premier réalise un décodage des instructions, le second déroule les séquences de microprogrammes relatives aux instructions, le dernier réalise l'interface avec la partie opérative. Il faut remarquer que le modèle de synchronisation utilisé par ces circuits est plus efficace que celui utilisé par SYCO.

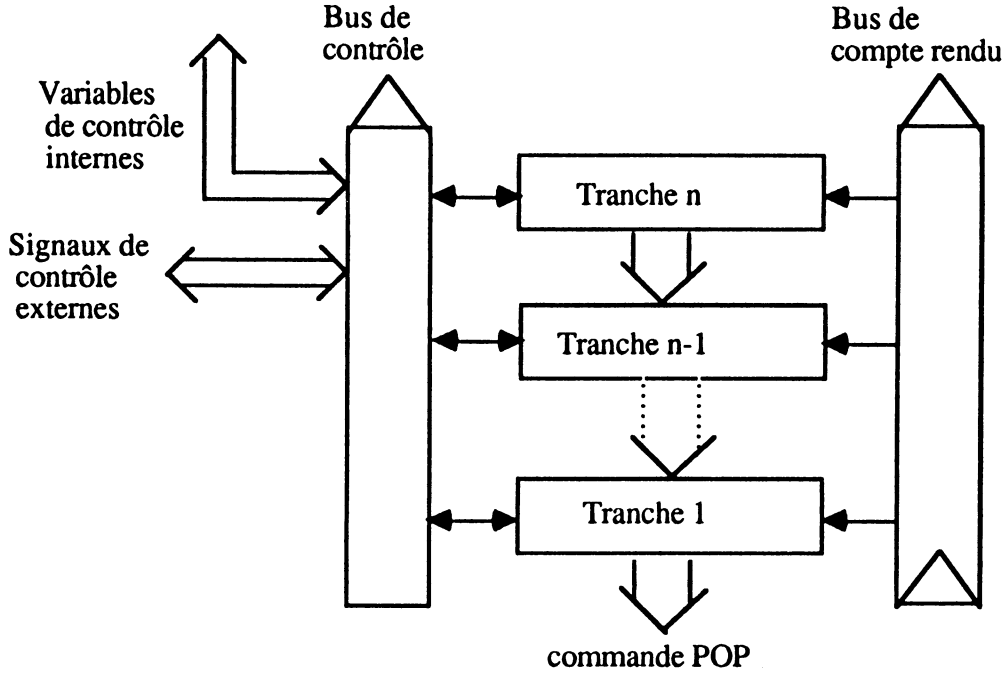


Figure IV.9 : organisation globale de la partie contrôle

Les tranches de contrôle générées par SYCO sont organisées autour de deux bus : le bus de contrôle et le bus de comptes rendus. Chaque tranche mémorise ses entrées et ses sorties. Les entrées sont mémorisées pendant la phase T1. Ces valeurs sont utilisées pendant les phases T2 et T3 pour le calcul des sorties. Les sorties sont mémorisées pendant la phase T4.

Le bus de comptes rendus est utilisé comme entrée par les tranches de contrôle et affecté par la partie opérative. Les tranches de contrôle ne peuvent donc pas y accéder durant les cycles où la partie opérative risque de le modifier. Ce bus, défini à partir de la description comportementale, permet de partager des registres entre la partie opérative et la partie contrôle. Cette dernière ne peut utiliser ces registres qu'en lecture. Il contient un fil pour chaque bit de données utilisé dans les expressions de condition de la description d'entrée. Notons que même si un bit est utilisé plusieurs fois, il ne coûtera qu'un seul fil.

Le bus de contrôle véhicule les valeurs et commandes (affectation) des variables de contrôle, les signaux d'horloge et les signaux de synchronisation interne. Il est partagé par

toutes les tranches en lecture/écriture, chaque tranche pouvant l'utiliser pour contrôler le séquencement (action conditionnelle), pour synchroniser le fonctionnement de la tranche par rapport aux autres et pour mettre à jour des variables de contrôle.

Une variable de contrôle interne est matérialisée par un registre au niveau de l'entrée de l'étage de contrôle qui l'utilise comme condition (figure IV.10). Un ou plusieurs étages, suivant la description, vont pouvoir affecter cette variable de contrôle par l'intermédiaire du bus de contrôle et d'un signal ("set" pour la mise à 1 et "reset" pour la mise à 0).

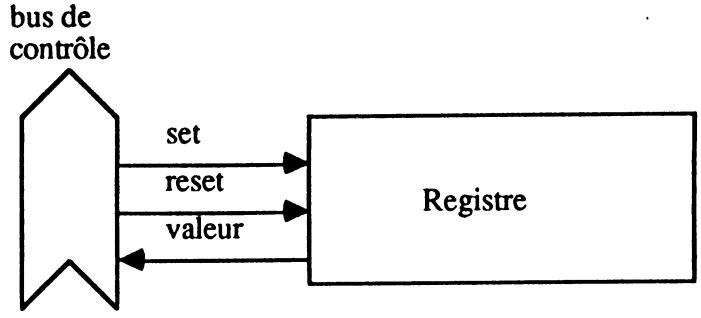


Figure IV.10 : Schéma de communication d'une variable interne

Le bus de contrôle, déterminé à partir de la description comportementale, est aussi utilisé par les signaux externes. Un protocole d'utilisation permet d'assurer la cohérence des informations véhiculées par ce bus. A chaque bit de contrôle (élément d'une variable ou d'un signal structuré) va correspondre un ou plusieurs fils (figure IV.4) :

- un fil pour un signal en entrée (valeur),
- deux fils pour un signal en sortie (set, reset),
- trois fils pour une variable de contrôle ou un signal bidirectionnel (set, reset, valeur).

Du point de vue temporel, il est nécessaire de garder une cohérence entre les signaux internes et les signaux externes ; pour cela les sorties sont valides pendant T4 et les entrées sont prises en compte pendant T1. [GER87] donne de plus amples détails sur le modèle de fonctionnement des variables de contrôle.

IV.4.4 Architecture d'une tranche de la partie contrôle

La figure IV.11 donne l'organisation générale d'une tranche de contrôle. Les entrées d'un étage sont composées des commandes émanant de l'étage supérieur (INSTi), de l'état interne de l'étage (seq), des variables de contrôle internes et externes (bus de contrôle) et des variables retournées par la partie opérative (bus de comptes rendus).

En sortie, chaque étage génère le nouvel état interne (seq), les commandes destinées à l'étage inférieur (INSTi-1) et les commandes de manipulation de variables de contrôle (les fils de commandes SET et RESET) et les fils de synchronisation des différents étages de contrôle

IV.4.5 Architecture de la partie opérative

SYCO utilise le compilateur APOLLON [JAM85], [JAM86a] pour générer les parties opératives. La partie opérative reçoit des signaux de commandes générés par la partie contrôle, qui sont amplifiés et synchronisés. Elle génère des signaux de comptes rendus.

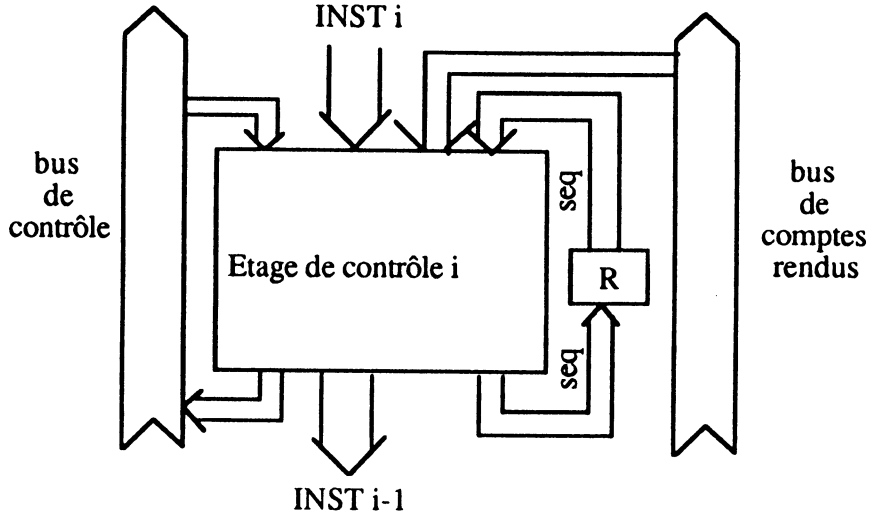
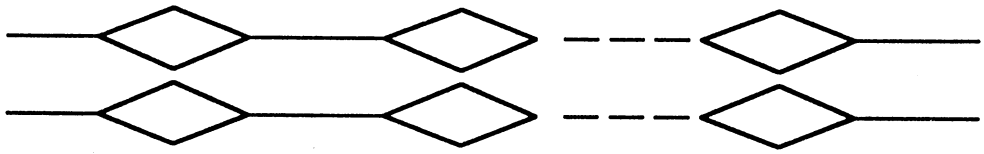
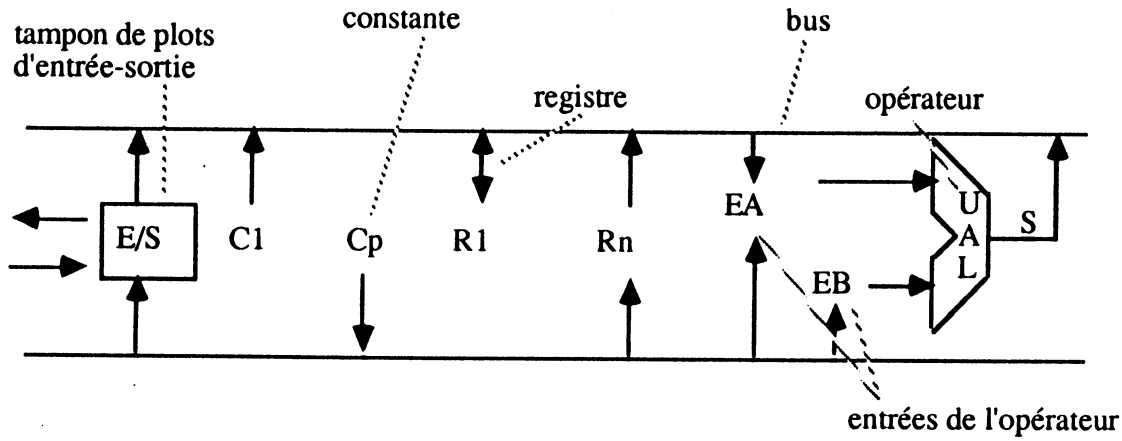


Figure IV.11 : Organisation d'un étage de contrôle

Le modèle global actuel de la partie opérative dans SYCO est dérivé de l'architecture de la partie opérative du microprocesseur MC68000. Il est organisé autour de deux bus. Ces deux bus peuvent être morcelés, ce qui permet de définir des sous-parties opératives (fig.IV.12).



(a) Architecture globale



(b) organisation d'une sous partie opérative

Figure IV.12 : Architecture cible des parties opératives générées par APOLLON [JAM86a]

La partie opérative est constituée d'un ensemble de sous-parties opératives à deux bus placées côte à côte et pouvant fonctionner en parallèle. Chaque sous-partie opérative peut fonctionner de manière indépendante (exemple : une sous-partie opérative pour les adresses et une autre pour les données) ou être connectée à l'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'entrées/sorties

et d'un ensemble d'opérateurs connectés à un ou aux deux bus.

[JAM86a] donne de plus amples détails sur l'organisation de ce modèle de partie opérative. Un nouveau modèle de partie opérative est en cours d'étude (voir IV.8).

IV.4.5 Modèle de fonctionnement : la synchronisation.

Le compilateur doit générer lui-même le modèle des procédures de communication entre les tranches du circuit. Il doit générer aussi le matériel nécessaire pour réaliser les protocoles de communication. Il faut donc définir des protocoles pour l'utilisation des signaux de synchronisation (horloge, "reset"), et des protocoles pour permettre les échanges de signaux entre les différentes tranches de contrôle.

Comme il a été dit plus haut, le circuit utilise une seule horloge externe. Toutes les tranches du circuit utilisent la même unité de temps ou cycle de base. Un cycle de base correspond à une période de l'horloge externe. Il est décomposé en quatre phases (T1, T2, T3, T4). Ces quatre phases peuvent être utilisées de manière différente d'un étage à l'autre.

L'initialisation du circuit est un autre point important pour la synchronisation du circuit. En fait, pour décrire les protocoles de communication, on suppose que les différentes tranches du circuit sont dans un état cohérent. Le circuit contient un signal d'initialisation physique rajouté automatiquement par le compilateur. Ce signal (appelé "reset") est distribué dans toutes les tranches de contrôle. Chaque tranche de contrôle mémorise ses entrées et la plupart de ses sorties. Seules les sorties qui commandent des variables de contrôle et celles qui commandent la partie opérative n'ont pas besoin d'être mémorisées par des registres statiques. Les autres entrées/sorties sont mémorisées par des registres statiques. Le signal d'initialisation permet de mettre ces registres dans un état initial cohérent. Cet état entraîne la reprise de l'exécution de

l'algorithme d'interprétation à l'état initial

Le modèle de tranche de contrôle de la figure IV.11 fait que chaque tranche reçoit des instructions (ou des commandes dans le cas de la partie opérative) de la tranche immédiatement supérieure. Avant de voir les avantages et les inconvénients de ce modèle qui est retenu pour SYCO, analysons deux autres modèles

Le premier utilise des connexions point à point permettant à chaque étage de passer des instructions à tous les étages qui lui sont inférieurs. Ce modèle est montré par la figure IV.13. Un étage choisit l'instruction qu'il doit exécuter à travers un multiplexeur

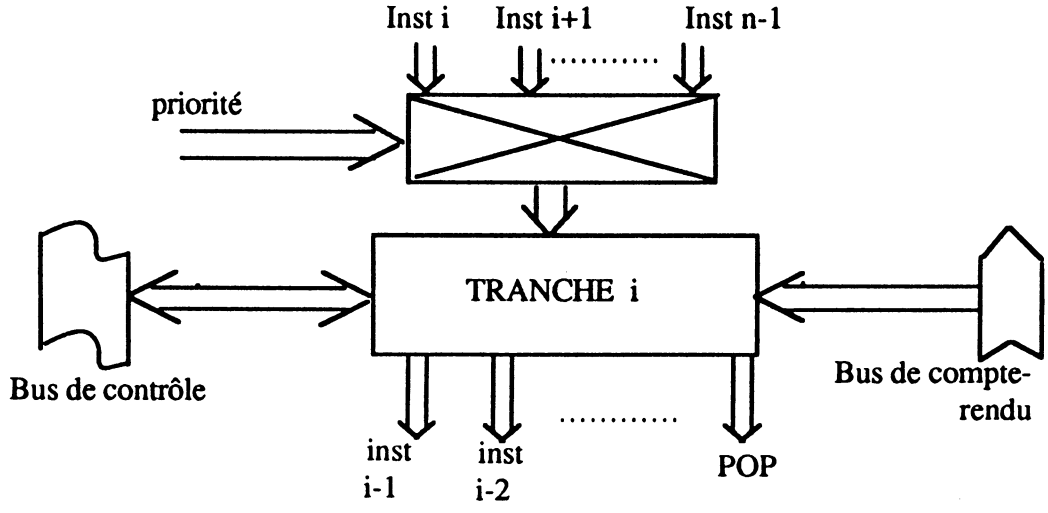


Figure IV.13 Organisation d'une tranche de contrôle pour le modèle de communication point à point

Cette solution présente deux inconvénients : elle nécessite une gestion de conflits sophistiquée et elle consomme trop de matériel. Les conflits arrivent quand deux tranches différentes veulent commander, en même temps, une troisième tranche. C'est un problème de synchronisation bien connu. Chaque tranche est une ressource utilisable par toutes les tranches qui lui sont supérieures. La gestion de ces conflits peut se faire de manière centralisée ou distribuée. La première solution entraîne un système lent tandis que la seconde peut être coûteuse. La duplication des fils de connexion fait aussi que ce modèle de tranche de contrôle est coûteux.

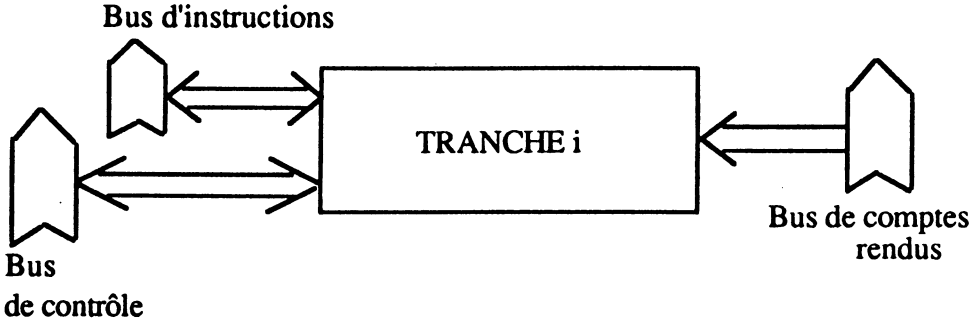


Figure IV.14 Organisation d'une tranche pour le modèle de communication via un bus

Dans le second modèle de communication, un bus d'instruction permet à chaque tranche de commander toutes les tranches qui lui sont inférieures. Ce modèle est montré par la figure IV.14. Il est plus économique que le premier, mais il ne permet qu'un seul appel à la fois, c'est-à-dire que les étages ne pourront pas travailler en parallèle.-

Le modèle retenu par SYCO est celui de la figure IV.11. Chaque étage communique avec son voisin supérieur et son voisin inférieur. Ce modèle peut entraîner une perte de cycles. En fait, dans le cas où un étage veut commander un étage autre que son voisin inférieur, il est obligé de passer par ce dernier pour transmettre la commande. Par contre, ce modèle permet de faire fonctionner les tranches de contrôle en mode pipeline. En plus ce modèle est économique d'un point de vue matériel

Chaque tranche de contrôle mémorise ses entrées/sorties, et contient un bloc de synchronisation. Ce dernier contient deux variables pour gérer le fonctionnement et la synchronisation des tranches de la partie contrôle. Elles sont utilisées comme des variables de contrôle. Les entrées sont mémorisées pendant la phase T1 dans des registres. Les phases T2 et T3 sont réservées au calcul des sorties. Les sorties sont mémorisées après ce calcul, durant la phase T4. Les deux registres d'entrées/sorties forment un système maître-esclave qui permet d'éviter les rebouclages et de séparer les cycles de fonctionnement. Ils appartiennent au même étage pour le séquencement interne, à des étages différents pour les instructions.

Lorsqu'une tranche de partie contrôle génère une instruction (INSTi-1) non nulle, elle va faire fonctionner une ou plusieurs des tranches inférieures et parfois la partie opérative. Les actions causées par cette instruction se termineront au bout de k cycles machine. Dans certains cas, k peut même être infini si on boucle sur un état.

Chaque tranche de partie contrôle calcule son état suivant à partir de l'instruction fournie par la tranche supérieure, de son état courant, et des conditions. L'instruction et l'état courant sont connus et fixes. Pour passer à l'état suivant et activer la tranche inférieure, il faudra que les conditions soient disponibles et que la tranche inférieure soit disponible. Dans le cas où l'état suivant nécessite un compte rendu pouvant être produit par l'état courant, la tranche de contrôle doit attendre que tous les étages inférieurs aient terminé l'exécution des commandes générées par l'état courant pour passer à l'état suivant. Ce modèle de synchronisation est détaillé dans [GER87]. Il a été mis à jour dans [MAC89] pour intégrer des étages organisés autour d'une ROM. On peut noter qu'un autre modèle de synchronisation a été utilisé par la première version de SYCO : il s'agit d'un modèle procédural où un seul étage peut être actif à la fois [VAR87]. Dans les deux cas, la partie opérative est esclave de la dernière tranche de la partie contrôle (un cycle de la dernière tranche fournit un cycle de travail de la partie opérative).

IV.5 La forme intermédiaire

L'utilisation d'une forme intermédiaire permet de simplifier les étapes de compilation. Dans le cas du compilateur SYCO, la forme intermédiaire est une forme canonique de la description d'entrée. La mise sous forme canonique permet aussi de détecter certaines incohérences contenues dans la description d'entrée.

La forme intermédiaire utilisée par SYCO est de type syntaxique. Au départ, elle constitue l'arbre syntaxique de la description du circuit en LDS. La forme intermédiaire utilisée par les différents outils de SYCO est aussi constituée par un sous-ensemble restreint du langage d'entrée. Le fait que la forme intermédiaire est à la fois une forme intermédiaire syntaxique et un sous-ensemble du langage d'entrée du compilateur a deux conséquences intéressantes :

- La forme intermédiaire peut être "décompilée". Le résultat de cette décompilation est aussi une description comportementale pouvant être réutilisée en entrée du compilateur. Ce point est important, car il rend la forme intermédiaire lisible. L'utilisateur peut récupérer la forme intermédiaire à un stade avancé de la compilation et la transformer manuellement puis la recompiler.
- Le sous-ensemble utilisé par la forme intermédiaire étant proche de l'architecture, l'utilisateur peut réaliser des optimisations fines au niveau de la description comportementale. Ainsi, les parties critiques des circuits peuvent être décrites de manière fine en utilisant les primitives de la forme intermédiaire.

IV.5.1 Organisation de la forme intermédiaire

Comme la description de circuits, la forme intermédiaire est composée d'un SMODULE qui contient les déclarations, et d'une hiérarchie de CMODULEs. Au début du processus de compilation, la forme intermédiaire a exactement la même structure que la description originale. La forme intermédiaire est générée par un pré-compilateur de LDS. Elle subit plusieurs transformations durant le processus de compilation. Ces transformations réorganisent chaque CMODULE pour regrouper les instructions en cycles de contrôle. Elle réorganisent également la hiérarchie des CMODULEs pour effectuer le découpage de la fonction réalisée par le circuit en niveaux d'interprétation. L'algorithme de découpage fera l'objet de la section IV.6.

La première étape de la compilation consiste à transformer chaque CMODULE en une machine d'états finis. Elle réalise un ordonnancement initial. Les instructions du CMODULE sont regroupées en cycles de contrôle. La formation des cycles de contrôle est faite selon un algorithme simple :

- Chaque PINST (groupe d'instructions parallèles) forme un cycle.
- Chaque instruction qui n'est pas imbriquée dans une autre instruction forme un cycle.

Comme il a été dit plus haut, les instructions conditionnelles sont exécutées en un seul cycle.

Cette étape transforme chaque CMODULE en une série de blocs d'instructions parallèles, appelés PINST en LDS. Une PINST correspond à un cycle de contrôle. Exemple : le CMODULE "ada" (cf. la description du circuit "mini" figure IV.6),

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

est réorganisé en deux cycles, et devient

```
CMODULE ada ;
<ada_xx> (r : = 2, EXECUTE fetch ;)
<ada_yy> (a : = a + b ;)
end ;
```

ada_xx et ada_yy sont des noms donnés aux deux cycles pour pouvoir y accéder par les instructions de branchement.

L'ordonnancement initial généré par cette étape sera complété, durant la compilation des parties contrôles, par le calcul des branchements implicites. Ce calcul et le découpage de la description en niveaux d'interprétation nécessitent la mise sous forme canonique des instructions de contrôle.

IV.5.2 Transformation de la forme intermédiaire sous une forme canonique

La mise sous forme canonique est une mise à plat des instructions conditionnelles. Chaque instruction conditionnelle (IF ou CASE) est transformée en un ensemble d'instructions conditionnelles de la forme :

IF <condition> <Action inconditionnelle> END ;

L'action inconditionnelle peut être une action opérative (instruction d'affectation) ou une instruction de contrôle (instruction de branchement ou instruction d'affectation d'une variable de contrôle). Cette forme interdit l'imbrication d'instructions conditionnelles. La mise sous forme canonique peut être réalisée par quatre type de transformation.

Chaque construction CASE est transformée en une suite de constructions IF. La forme générale de l'instruction CASE est :

```
CASE (<variable>)
      WHEN (<valeur 1>) <bloc-Instruction 1>
         • • •
      WHEN (<valeur n>) <bloc-Instruction n>
   end
```

Elle est transformée en

(IF (<variable> = <valeur 1>) <bloc-Instruction 1> END
• • •
IF (<variable> = <valeur n>) <bloc-Instruction n> END;)

Exemple: l'instruction

```
<modad> CASE (RI 4:2)
            WHEN ('00') NEXT decode ; END;
            WHEN ('01') EXECUTE fetch ; AD : = RI ; END;
            WHEN ('10') EXECUTE fetch ; AD : = C + RI ; END;
            WHEN ('11') EXECUTE fetch ; AD : = C  - RI ; END;
        END;
```

est transformée en :

```
<modad> (IF (RI  4 : 2 = (BIN '00')) NEXT decode ; END;
            IF (RI  4 : 2 = (BIN '01')) EXECUTE fetch ; AD : = RI ; END;
            IF (RI  4 : 2 = (BIN '10')) EXECUTE fetch ; AD : = C + RI ; END;
            IF (RI  4 : 2 = (BIN '11')) EXECUTE fetch ; AD : = C - RI ;END;)
```

On peut remarquer que, dans l'exemple de l'instruction CASE donné ci-dessus, la somme (booléenne) des expressions de condition ne forment pas une tautologie. D'autre part, ces expressions peuvent ne pas être mutuellement exclusives.

La deuxième transformation est semblable à la précédente, elle permet d'éliminer les "ELSE" dans les constructions IF. Exemple : l'instruction

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

est transformée en :

(IF (restart = 0) NEXT rest ; END;
IF (non (restart = 0)) NEXT start ; END;)

De manière plus générale, la construction

IF (<condition>) <Bloc-then> ELSE <Bloc-else> END

est transformée en :

(IF (<condition>) <Bloc-then> END ;
IF (non (<condition>)) <Bloc-else> END ;)

La troisième transformation réalise une expansion d'instructions conditionnelles de la forme

IF <cond> <Instruction 1>....<Instruction n> END

en une suite de n instructions conditionnelles du type

if <cond> <instruction i> end /1 ≤ i ≤ n.

Exemple : l'instruction

IF (RESTART) NEXT FETCH0 ; A : = B ; END ;

est transformée en :

(IF (RESTART) NEXT FETCH0 ; END ;

IF (RESTART) A : = B ; END ;

La quatrième et dernière transformation permet de mettre à plat les instructions imbriquées. La forme du type :

IF <cond 1> If <cond 2> <bloc-Instruction END ;

est transformée en

IF (<cond 1> AND <cond 2>) <bloc-Instruction END ;

Les transformations de la forme intermédiaire et la formation des cycles sont faites simultanément par un seul processeur qui travaille en une seule passe (NB : ce processeur a été réalisé en collaboration avec H.N. NGUYEN de la compagnie BULL). La forme canonique sera recompactée dans la suite du processus de compilation.

IV.5.3 Restrictions sémantiques

La syntaxe de la forme intermédiaire utilisée permet des descriptions qui sont incompatibles avec le modèle architectural défini par la section précédente (IV.4). Dans ce modèle, chaque étage de contrôle possède un seul registre d'état pour mémoriser l'état suivant et un seul registre pour mémoriser les appels à l'étage inférieur. Ces deux contraintes vont impliquer qu'un cycle ne peut avoir qu'un seul successeur actif à un instant donné. D'autre part, un cycle ne peut pas contenir plus d'une seule instruction EXECUTE active à la fois. Ceci revient à dire que, si un cycle contient deux instructions de branchement (NEXT ou EXIT) différentes, alors elles ne doivent jamais s'exécuter en même temps. Autrement dit, les deux instructions doivent être contrôlées par deux conditions mutuellement exclusives. La restriction est la même pour l'instruction EXECUTE. Le même problème se pose pour les instructions d'affectation, affectation de contrôle et affectation de données.

De manière plus générale, deux actions différentes peuvent appartenir au même cycle si l'une des deux conditions suivantes est vérifiée :

1) Les deux actions sont mutuellement exclusives, c'est-à-dire qu'elles ne sont jamais exécutées en même temps.

2) Les deux actions sont compatibles, c'est-à-dire qu'elles ne sont pas en conflit. Comme il a été défini en III.7.1, il existe trois types de conflits : les conflits dus aux données, les conflits dus aux opérateurs et les conflits dus au format (instruction de contrôle).
Les conflits dus aux données et aux ressources sont résolus par le compilateur de partie opérative, tandis que les conflits dus au format sont résolus par le compilateur de partie contrôle (voir aussi IV.9.1).

IV.6 Génération de l'architecture globale : partition

La génération de l'architecture globale consiste à découper la description du circuit en niveaux d'interprétation. Une première solution de découpage est extraite à partir de la hiérarchie d'appel des CMODULEs (procédures) contenue dans la description d'entrée. Cette solution initiale pourra être modifiée par les outils d'optimisation architecturale. Ce découpage est simplifié par l'utilisation d'un modèle de synchronisation prédéfini.

IV.6.1 Principe de la partition

La partition utilise l'arbre d'appel des procédures (ou CMODULEs) de la description initiale pour former des classes de telle sorte que :

- la procédure principale forme la classe n. Elle correspond à la tranche de contrôle du sommet de la pile.
- les procédures appelées par la classe i forment la classe i-1.

Les procédures de chaque classe sont regroupées pour former une nouvelle procédure avec plusieurs points d'entrée.

Pour que ce découpage donne une description directement transposable dans l'architecture décrite plus haut, deux contraintes doivent être respectées :

- une procédure donnée ne peut pas appartenir à deux classes différentes,
- seules les procédures constituant la dernière classe peuvent contenir des actions opératives (opérations sur les variables de type données)

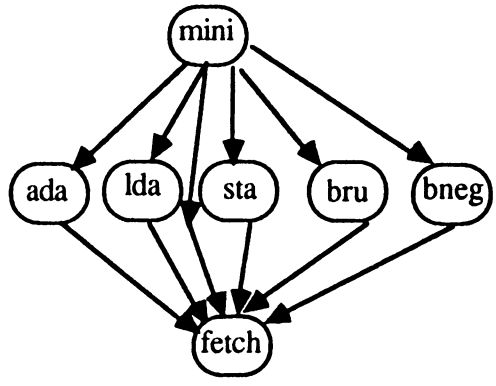


Figure IV.15 Hiérarchie de contrôle originale

Afin de respecter ces deux contraintes, on peut être amené à créer des "procédures relai". Les figures IV.15 et IV.16 illustrent le fonctionnement de l'extracteur. La figure IV.15 montre l'arbre d'appel des procédures extrait de la description du processeur "mini" (figure IV.6). Cet

arbre est composé de trois niveaux. Le terme "arbre d'appel" est un abus de langage. En réalité, il s'agit plutôt d'un graphe sans cycle. Le découpage produit 3 classes de procédures (figure IV.16). Dans cet exemple la procédure principale fait appel à la procédure (CMODULE) "fetch" qui est au niveau le plus bas de la hiérarchie. Un point d'entrée (mini_newinstr_1) est rajouté dans la classe 1 pour assurer la transmission de l'appel.

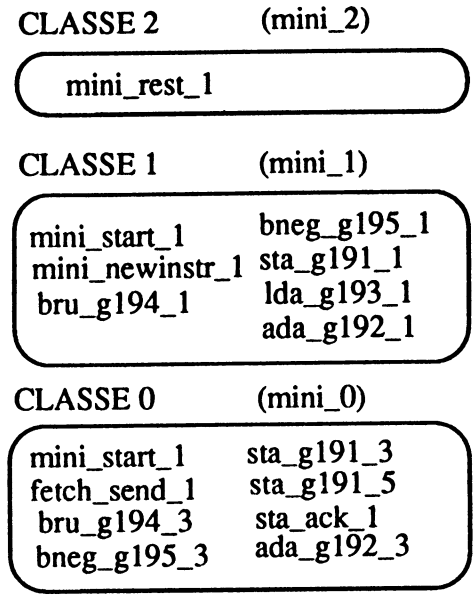


Figure IV.16 Classes de CMODULEs extraites

Le regroupement des instructions au niveau 0 a entraîné la création de plusieurs nouvelles procédures dans le niveau 1 (mini_start_1) et dans le niveau 0 (toutes les procédures autres que fetch_send_1).

IV.6.2 Algorithme de partition

L'algorithme de partition utilise la forme intermédiaire mise sous forme canonique et procède en 2 étapes, la première affectant un niveau d'interprétation à chaque instruction de la forme intermédiaire, la seconde réalisant le découpage proprement dit.

L'affectation des niveaux d'interprétation fonctionne de la manière suivante :

- Les instructions opératives sont de niveau 0.
- Les instructions de branchement (autres que EXECUTE) ont le niveau du CMODULE qui les contient.
- Le niveau d'un bloc d'instructions parallèles (PINST) est égal au niveau de l'instruction du bloc qui a le niveau le plus élevé.
- Le niveau d'un CMODULE est égal au niveau de la PINST du CMODULE, qui a le niveau le plus élevé.
- Le niveau d'une instruction EXECUTE est égal à "1 + le niveau du CMODULE appelé".

• Le niveau des autres actions (affectation de contrôle) n'est pas pris en compte par le découpage.

L'algorithme de partition en niveaux d'interprétation fonctionne de manière récursive. Les procédures de la classe zéro sont construites en premier, puis on construit la classe 1, etc.

La construction de la classe i peut être résumée de la manière suivante

```
Pour tout CMODULE C de la forme Intermédiaire tel que Niveau (C) >= i faire
   si Niveau (C) = i
      Alors Insérer C dans la classe i
      Sinon (* Niveau(C) > i *)
         Pour toute PINST P du CMODULE C faire
            Choix : 1 parmi trois cas.
            cas1) Niveau (P) > i et P ne contient pas d'instruction de niveau i
                  "ne rien faire"
            cas2) Niveau (P) = i
               Avec les instructions I de P tel que Niveau(I) = i où I est une affectation de
                     contrôle, Créer une nouvelle PINST P1
               Créer 1 CMODULE C1 qui contient P1, insérer C1 dans la classe i.
               Remplacer dans P, les instructions de P1 par un appel au CMODULE C1.
            cas3) Niveau(P) > i et P contient des instructions I telles que Niveau(I) = i
               P est remplacé par une séquence de deux PINSTs P1 et P2.
               P1 contient les affectations de contrôle et les instructions de niveau i.
               P2 contient les autres instructions.
               P1 est traitée comme le CAS 2.
            finchoix
         finpour
      finsinon
   finsi
finpour
```

L'algorithme décrit ci-dessus peut entraîner le découpage d'un cycle en plusieurs cycles. Ce découpage intervient dans les cas où un cycle contient des actions appartenant à plusieurs niveaux différents. La partition peut donc entraîner la modification du nombre de cycles nécessaires pour exécuter une fonction donnée. Une telle modification peut altérer le fonctionnement du circuit dans le cas où le cycle découpé appartient à des protocoles de communication.

On peut remarquer aussi que l'algorithme de construction des niveaux d'interprétation décrit ci-dessus ne considère pas le cas où les cycles sont exécutés en séquence. La prise en compte de ce genre de situation peut entraîner une amélioration du découpage.

Une fois la partition en niveaux d'interprétation terminé, les CMODULEs de chaque classe sont regroupés en un seul CMODULE avec plusieurs points d'entrée. Les trois CMODULEs produits par le découpage de la description exemple de la figure IV.6 sont donnés par la figure IV.17.

Cette description va remplacer la forme intermédiaire initiale pour les autres étapes de compilation. Chaque CMODULE constitue la description d'une tranche de contrôle.

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


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

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

Figure IV.17 Description des tranches de contrôle extraites de la description de la figure IV.6

IV.7 Compilation de partie contrôle

Cette section est en grande partie basée sur les travaux de MHAYA [MHA88a] concernant le compilateur de parties contrôle CPC. Ce dernier est mis en œuvre par SYCO pour transformer la description comportementale de chaque étage de contrôle en une forme directement transposable dans l'architecture d'une tranche de contrôle.

De manière générale, la compilation d'une partie contrôle est décomposée en trois étapes [MHA88a] : la première définit une table de transitions à partir de la description du contrôleur. La seconde étape effectue des modifications de la table des transitions pour réaliser des optimisations logiques ou des transformations topologiques. La dernière étape effectue un codage de la table de transitions en vue de sa transposition dans une structure donnée. La structure peut être associée à une topologie, comme c'est le cas pour les PLA.

Dans le cas du compilateur de partie contrôle CPC, tous les efforts ont été concentrés sur les deux premières étapes. L'étape de codage est réalisée de manière directe dans la version actuelle de CPC. Il faut dire que cette étape a été amplement traitée dans la littérature [DEM84]. Elle met en œuvre des outils de codage proprement dit et des outils d'optimisation logique [RUD87].

On peut se reporter à [BRA85a], [DEM84] et [DEV88] pour une bibliographie plus complète sur les outils d'optimisation et de codage.

IV.7.1 Le compilateur de partie contrôle CPC

Le compilateur CPC utilise la forme intermédiaire comme structure de données. Il intervient après l'étape de partition. La forme intermédiaire est alors organisée en un ensemble de CMODULEs. Chaque CMODULE décrit un étage de contrôle et peut contenir plusieurs points d'entrée. On peut distinguer trois types de tranches de contrôle :

- la tranche supérieure : elle correspond au programme principal de la description comportementale. Le CMODULE correspondant a un seul point d'entrée qui constitue, par ailleurs, l'état initial du circuit.
- les tranches intermédiaires : chacune de ces tranches est décrite par un CMODULE qui peut avoir plusieurs points d'entrée.
- la tranche inférieure : elle est décrite par un CMODULE qui peut avoir plusieurs points d'entrée. Elle est la seule à contenir des actions opératives. La spécification de la partie opérative est générée à partir de la description de cette tranche. Le schéma de compilation de cette tranche est différent de celui des autres tranches, puisqu'il inclut une étape de génération de la partie opérative.

La description de chaque tranche de contrôle spécifie une (dans le cas de la tranche supérieure) ou plusieurs machines d'états finis disjointes. L'état initial de chaque machine d'état constitue un point d'entrée dans le CMODULE qui décrit l'étage de contrôle. Chaque état est un cycle de contrôle. Dans le cas de la tranche inférieure, un cycle de contrôle peut être décomposé en plusieurs cycles de base par le compilateur de parties opératives.

IV.7.2. Organisation interne d'une tranche de contrôle

L'organisation générale d'une tranche de contrôle est montrée par la figure IV.11. Plusieurs solutions sont possibles pour l'organisation interne d'une tranche de contrôle. Le choix d'une organisation donnée doit tenir compte du processus de compilation. Pour que la génération automatique d'une tranche de contrôle soit possible, il faut :

- trouver une bonne solution topologique pour pouvoir assembler la tranche de contrôle avec les autres blocs du circuit sans grande perte de surface,
- respecter les protocoles de synchronisation,
- trouver des algorithmes efficaces pour la génération automatique de la tranche de contrôle,
- permettre l'établissement de fonctions simples et rapides pour l'évaluation des performances afin d'offrir au compilateur le choix entre plusieurs solutions.

Ces exigences font que certaines bonnes solutions connues sont difficilement automatisables. Par exemple les essai d'automatisation du modèle de partie contrôle avec générateur de temps [OBR82] faute d'un bon modèle topologique. Par contre, les modèles basés sur les structures programmables (PLA, ROM) sont faciles à automatiser. La figure IV.18 montre l'organisation d'une tranche de contrôle à base d'un PLA.

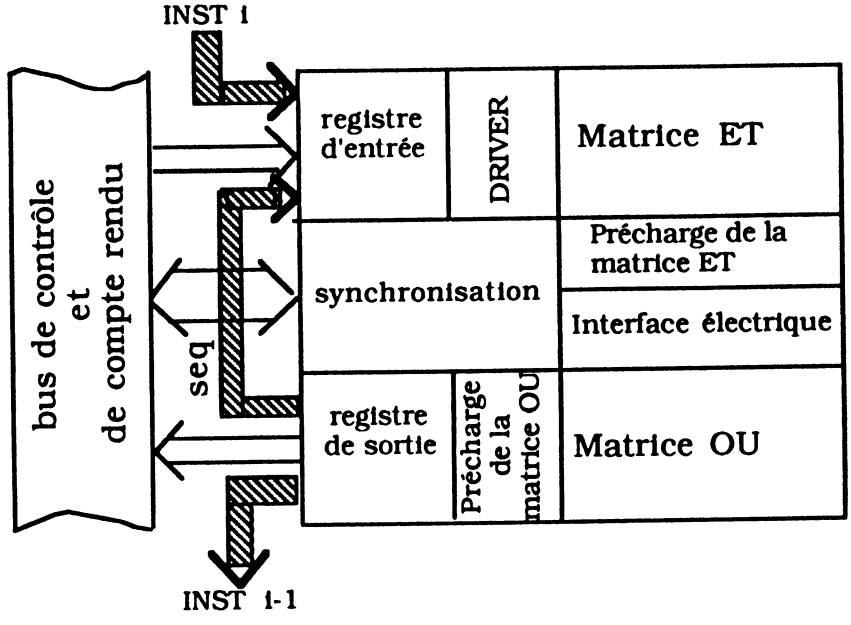


Figure IV.18 organisation d'une tranche de contrôle à base d'un PLA [GER87]

Cette figure montre aussi l'organisation topologique de la tranche de contrôle qui a été étudiée pour faciliter l'assemblage de plusieurs tranches de contrôle.

Un modèle de tranche de contrôle basé sur une ROM est en cours de développement. L'utilisation d'une ROM permet de réaliser des tranches de contrôle plus rapides. La figure IV.19 résume les mesures effectuées sur ce modèle. Ces chiffres sont le résultats de simulation Spice détaillées dans [MAC89]. La figure IV.19 montre qu'une tranche de contrôle basée sur une ROM peut travailler à une fréquence de 20 MHz même dans le cas où la ROM est de grande taille (256 mots de 256 bits), alors que les tranches à base de PLA sont limitées à 10 MHz [GER87].

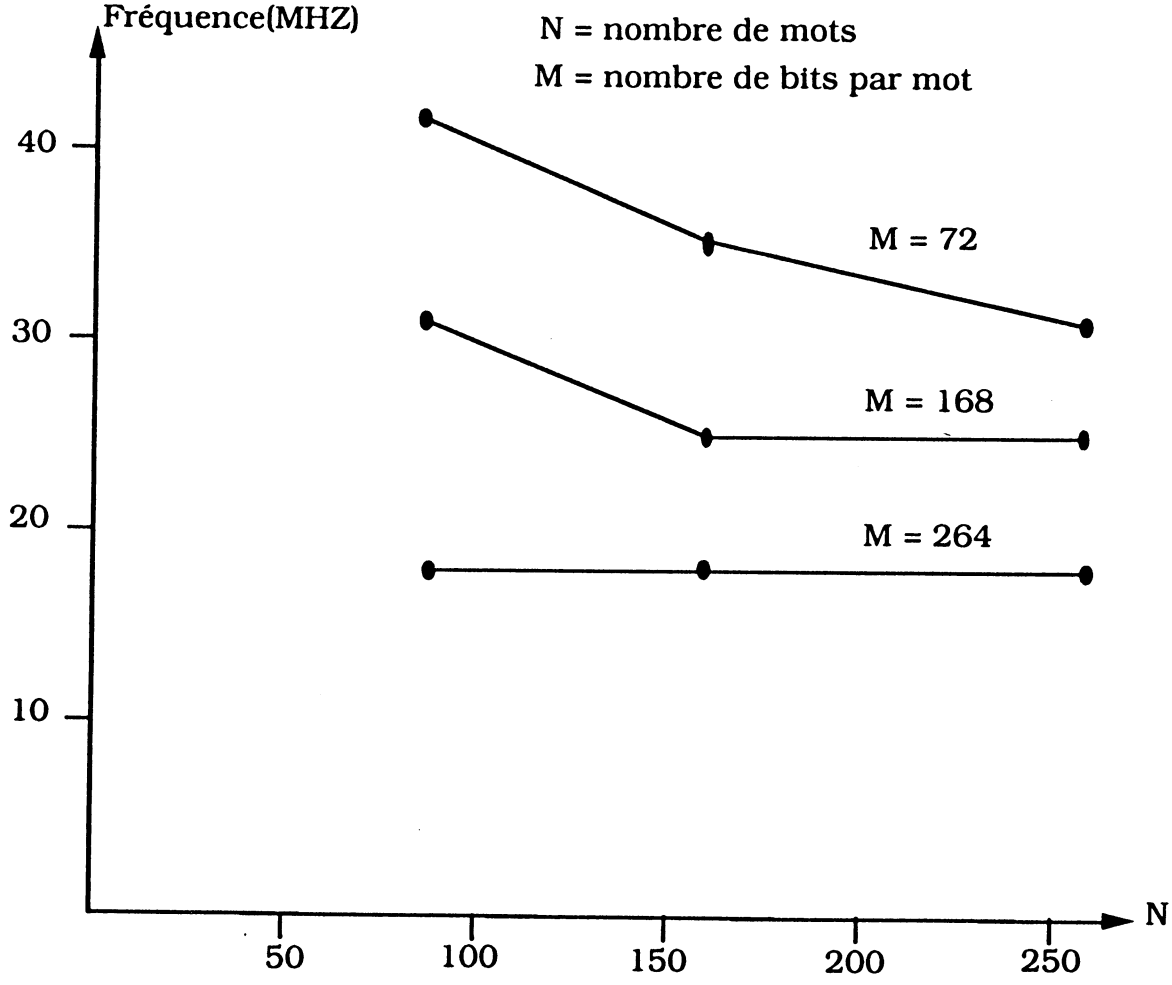


Figure IV.19 Fréquence de fonctionnement d'une tranche de contrôle basée sur une ROM [MAC89]

D'un autre côté, le modèle basé sur une ROM ne permet pas la réalisation de séquencement complexe. Les deux modèles (PLA et ROM) pourront être utilisés dans une même tranche de contrôle. Ce nouveau modèle de tranche de contrôle est étudié dans [MAC89] pour réaliser la tranche de contrôle inférieure. Cette tranche sera donc composée d'un PLA et d'une ROM. Le PLA réalise le séquencement des cycles de contrôle et la communication avec

l'étage supérieur. La ROM réalise le séquencement à l'intérieur des cycles de contrôle et génère les commandes de la partie opérative. La synchronisation entre le PLA et la ROM se fait de la même manière qu'entre deux étages de contrôle

Dans la suite, on supposera que les tranches de contrôle sont réalisées par des PLAs et implantées selon le plan de masse donné par la figure IV.18.

IV.7.3. Algorithmes de compilation

Le compilateur CPC traite les étages de contrôle de manière séparée. Pour chaque étage, le processus de compilation est composé de trois étapes :

- transformation de la description de l'étage en une table d'états, appelée aussi table de transitions,
- optimisation logique et résolution des branchements implicites,
- génération de la description logique du PLA.

La description de la partie opérative est générée à partir de la table d'états du dernier étage de contrôle. Le compilateur de parties contrôle génère aussi un fichier d'interconnexions qui permettra l'assemblage des tranches de contrôle et de la partie opérative. D'autre part, il intègre dans la description de chaque étage les commandes pour les blocs de synchronisation associés à chaque tranche de contrôle.

IV.7.3.1 Génération de la table d'états

La forme intermédiaire de départ permet d'obtenir une table d'états initiale. Cette dernière doit être complétée par le calcul des branchements implicites et optimisés. D'autre part, il faut l'adapter à l'architecture cible. Dans le cas où la tranche de contrôle est réalisée par un PLA, il faut mettre les expressions de condition sous une forme canonique. Ce traitement consiste à transformer la description de chaque étage de contrôle de sorte que les conditions ne contiennent que les opérateurs élémentaires "=" et "AND" (opérateur logique). La table d'états ainsi déterminée peut être directement transposée dans un PLA.

IV.7.3.2 Optimisation et résolution des branchements implicites.

Le but de l'optimisation est de réduire le nombre d'entrées dans la table d'états, et donc de réduire par la suite la taille du PLA réalisant l'étage de contrôle. Cette optimisation utilise des techniques classiques d'optimisation de microprogrammes. L'ajout des branchements implicites a pour but de générer une table d'états où chaque état possède un et un seul successeur à la fois. Les algorithmes mis en œuvre durant cette phase utilisent des heuristiques. Les solutions algorithmiques pour l'ajout des branchements implicites peuvent mener à une explosion

combinatoire de la table d'états. La vérification des règles de branchement, par exemple, nécessite une preuve de tautologie Les algorithmes d'optimisation sont détaillés dans [MHA88a] et ceux de calcul des branchements implicites dans [MHA88].

IV.7.3.3 Codage des tables d'états

Le codage consiste à générer pour chaque tranche de contrôle la description logique du PLA correspondant. Le codage concerne :

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

Un étage peut être formé à partir de plusieurs procédures. Dans ce cas, la table d'états correspondante décrit plusieurs automates indépendants et le code d'un état se compose de deux champs : le numéro de l'automate dans l'étage (ce champ correspond aux commandes émanant de l'étage supérieur) et le numéro de l'état dans l'automate.

Le compilateur de parties contrôle calcule aussi la valeur des variables de synchronisation en fonction des branchements et des actions générées pour l'étage inférieur.

L'algorithme d'assignation d'état utilisé par CPC est simple. Il consiste à numéroter les états de l'automate. Cet algorithme est insuffisant. Par contre, une étape d'optimisation des PLAs peut être réalisée à l'aide du système d'optimisation logique ESPRESSO [RUD85]. L'étape d'assignation d'états est souvent considérée comme une étape critique pour les compilateurs de parties contrôle. Cette étape consiste à affecter un code à chacune des étiquettes de la description. Un bon codage peut entraîner une réduction importante dans la taille du PLA généré au cours de l'étape d'optimisation [DEV88].

Outre les PLAs, le compilateur de parties contrôle génère aussi un fichier d'interconnexions (figure IV.20). Ce fichier contient toutes les informations nécessaires à la génération du dessin des masques de la partie contrôle et à l'assemblage des différents étages de contrôle et de la partie opérative.

Actuellement, la compilation de la tranche inférieure est incomplète. Elle n'intègre pas les résultats de la compilation de la partie opérative. Le compilateur génère la description de la partie opérative sous forme de cycles de contrôle. La figure IV.21 montre les spécifications de la partie opérative générée par CPC à partir de la description exemple.

```
NAME  mini
PCN   3     ? Nombre d'étages
CRN   (ir3 ir2 ir1 ir0 a7) ? Compte Rendu
CTN   (dtack0 restart0)  ? Variables de Contrôle Lues
     (adstrobe0 dtstrobe0 read_write0)  ?Variables de Contrôle écrites
     (r0 r1)  ?  variables de contrôle internes
PC    2 (seq11 seq12) () (PT21 PT22 PT23) ? Description connectique de l'étage 2
rot   0
cr    (ir3 ir2 ir1 ir0)  ? Compte rendu utilisé  par l'étage
ct
  en (restart0)  ? Variables de contrôles lues par l'étage
  so (r1 r0)    ? Variables de contrôle modifiées par l'étage
PC    1 (seq21 seq22) (PT21 PT22 PT23) (PT31 PT32 PT33 PT34) ? étage 1

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

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

Figure IV.20 Fichier d'interconnexions

```
operators :  + ;  ? operators
registers : a b ir pc ; ? registers
Obuffers : adbus<0:8> ; ? the output buffer
IObuffers : dtbus<0:8> ; ? the input/output buffer
control accessible registers : ir<0:3> a<7:7> ;
     ? data registers tested by the control section
constants : 0 1 ; ? constants used by the data path
maximum subsection number : 3 ;
bit number : 8 ;
name : mini ;

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

Figure IV.21 Description de la partie opérative

Dans la version actuelle, CPC se comporte comme si les étapes de contrôle de la tranche inférieure devaient être réalisées par un autre étage de contrôle. En fait cette solution correspond à une architecture où le dernier étage est composé d'un PLA et d'une ROM [MAC89]. Comme il a été dit plus haut, dans un tel modèle, le PLA réalise le séquencement des cycles de contrôle. La ROM, dont le contenu est généré par le compilateur de parties opératives, contient le découpage des cycles de contrôle en cycles de base.

IV.8 Compilation de parties opératives

Cette section est limitée à la compilation des parties opératives pour le compilateur SYCO. Elle est en grande partie basée sur les travaux de JAMIER [JAM86a], effectués dans le cadre de SYCO. Une étude plus complète sur la compilation de parties opératives se trouve dans [JAM86a].

Dans le cas du compilateur SYCO, le compilateur de parties opératives doit s'intégrer au reste du processus de compilation pour permettre la génération de circuits entiers. Le compilateur de parties opératives définit les détails d'exécution des cycles de contrôle. Il doit donc réaliser l'allocation des ressources et découper les cycles de contrôle en cycles de base. La partie opérative est constituée de l'ensemble des ressources. Son fonctionnement doit être compatible avec les autres parties du circuit. La compilation des parties opératives, dans SYCO, met en œuvre le compilateur APOLLON. Les principes de fonctionnement de ce dernier sont détaillés dans [JAM85] et [JAM86a]. La suite de cette section contient un rappel des modèles utilisés par APOLLON ; des propositions d'extension de ces modèles seront ensuite formulées.

IV.8.1. Le compilateur de parties opératives APOLLON

Pour la compilation de parties opératives, le choix de modèles prédéfinis permet également de faciliter la réalisation d'algorithmes de compilation efficaces. Les modèles et les algorithmes utilisés par APOLLON sont décrits brièvement dans la suite.

IV.8.1.1 Les modèles utilisés par APOLLON

Comme il a été dit auparavant (voir IV.4.5), le modèle général des parties opératives générées par APOLLON est dérivé de la partie opérative du MC68000™. En plus de ce modèle général qui utilise deux bus segmentés (voir figure IV.12), APOLLON utilise quatre autres modèles pour la description des cycles de contrôle, l'exécution des actions opératives, le fonctionnement électrique et l'organisation topologique.

La spécification initiale de la partie opérative est donnée sous forme d'une liste de cycles de contrôle (appelés étapes de contrôle ou instructions opératives). Chaque étape est composée d'un ensemble d'actions opératives devant se dérouler en parallèle. APOLLON distingue quatre types d'actions opératives : le transfert simple, l'opération binaire, l'opération unaire, et l'opération d'entrée/sortie. La spécification initiale d'une partie opérative (voir figure IV.21) contient aussi une partie déclarative qui spécifie les registres et les opérations utilisées par les cycles de contrôle, les registres de la partie opérative qui seront utilisés par la partie contrôle et le nombre maximum de sous parties opératives pouvant être générées.

Chaque cycle de contrôle est exécuté par la partie opérative en deux cycles de base. Le cycle de base est le même que celui utilisé par la partie contrôle (IV.4.2.1). Les actions opératives sont aussi exécutées selon des modèles prédéfinis. Un transfert de base (ou transfert simple) s'écrit sous la forme "R1 <--- R2" (R1 et R2 sont des registres) et est exécuté pendant un cycle de base. Durant ce cycle, les ressources utilisées par ce transfert (les bus) ne sont pas disponibles pour l'exécution des autres actions opératives. L'utilisation d'un même bus par plusieurs transferts est autorisée dans le cas où tous les transferts ont la même source. Ces transferts sont appelés multi-transferts [ANC85]. Durant un cycle de base chaque sous-partie opérative est donc capable d'exécuter quatre transferts. Un transfert peut mettre en œuvre des registres appartenant à des sous-parties opératives différentes.

Une opération binaire est découpée en trois transferts de base. Deux de ces transferts sont exécutés durant le premier cycle, ils permettent d'acheminer les opérandes vers les deux entrées de l'opérateur. Le troisième transfert est exécuté durant le second cycle, il permet de récupérer le résultat. La récupération des indicateurs (compte rendu) nécessite un autre transfert durant le deuxième cycle.

Une opération unaire est découpée en deux transferts. Le modèle d'exécution est similaire à celui de l'opération binaire sauf que durant le premier cycle, un seul transfert suffit pour acheminer l'opérande vers l'opérateur.

Les opérations d'entrées/sorties nécessitent aussi deux cycles pour s'exécuter. L'accès aux bus de données externes se fait à travers des tampons d'entrées/sorties connectés aux bus externes. Chaque opération est donc découpée en deux transferts. Le premier concerne un registre interne et le tampon. Le deuxième relie le tampon et le bus externe. L'ordre d'exécution des deux transferts dépend de la nature de l'opération, lecture ou écriture.

Les modèles d'exécution des opérations ont été définis en accord avec la bibliothèque de cellules utilisée par le compilateur APOLLON. Les éléments de cette bibliothèque respectent aussi les modèles électriques et topologiques.

IV.8.1.2 Construction de la partie opérative

La génération de la partie opérative consiste à déterminer :

- le nombre de sous-parties opératives et leurs connexions, 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 ; elles 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'entrées/sorties de plots à placer dans chaque sous-partie opérative.

Malgré tous les modèles utilisés par APOLLON l'espace des solutions possibles pour réaliser une description donnée reste trop grand pour envisager des solutions algorithmiques. Par exemple, pour placer N registres dans P sous-parties opératives, il existe $P^N/2$ solutions dans le cas où P est pair. Le nombre de solution est $(P^N + 1)/2$ dans le cas où P est impair. La démonstration est donnée dans [JAM86a].

Afin d'éviter cette complexité, l'algorithme du compilateur de parties opératives APOLLON 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 cycles de contrôle un à un. Pour limiter l'espace des solutions exploré, on impose au compilateur de ne pas remettre en cause le placement des registres placés au cours de la compilation des instructions précédentes.

L'efficacité d'un tel algorithme va donc dépendre de l'ordre dans lequel sont traités les cycles de contrôle. APOLLON commence par examiner les instructions comprenant le plus de transferts en parallèle et termine par les instructions les plus simples. Un pré-traitement permet de trier les cycles de contrôle. Ce pré-traitement permet aussi de calculer des contraintes pour le placement des registres. Il existe deux types de contraintes : les premières décrivent des cas de placement de registres à éviter. Le respect de ces contraintes est une condition nécessaire pour pouvoir implanter tous les cycles de contrôle [ARZ84]. Les contraintes du second type sont moins fortes, elles décrivent les registres qu'il est souhaitable de placer dans la même sous-partie opérative [JAM86a].

Tous les algorithmes du compilateur APOLLON sont décrits dans [JAM86a]. Il existe des cas où le compilateur ne trouve pas de solution, bien qu'il en existe une. Pour le placement des registres d'un même cycle de contrôle, le compilateur utilise des heuristiques. Il place, dans la mesure du possible à l'intérieur d'une même sous-partie opérative, les registres qui apparaissent souvent ensemble dans des transferts ou des opérations, 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.

IV.8.1.3 Les limites du modèle

Le modèle utilisé par APOLLON présente plusieurs avantages, dont le principal est la possibilité de réaliser de manière efficace le compilateur APOLLON lui-même. Ce dernier a été utilisé pour générer les parties opératives de plusieurs circuits exemples [GER85], [REI86]. Par contre, le modèle limite le compilateur à une classe très restreinte de circuits :

- L'organisation de la partie opérative autour de deux bus limite la complexité des

cycles de contrôle qu'on peut traiter. On peut trouver des combinaisons d'actions opératives qui ne peuvent pas être exécutées avec ce modèle.

• Le modèle d'exécution des cycles de contrôle en un nombre fixe de deux cycles, limite les possibilités d'optimisation architecturale. Avec la version actuelle, si un cycle de contrôle n'est pas exécutable en deux cycles de base, il faut le découper en deux cycles de contrôle. Il utilisera donc quatre cycles de base pour s'exécuter, même si trois suffisent. D'autre part, même dans le cas où un cycle de contrôle ne nécessite qu'un seul cycle de base, il sera découpé en deux cycles de base. L'algorithme du compilateur APOLLON peut être adapté pour lever cette limitation.

• L'utilisation d'une boîte à opérations unique constitue probablement la limitation la plus forte du compilateur APOLLON, car elle interdit toute extension possible de ce dernier.

Le paragraphe suivant contient des propositions d'extension du compilateur APOLLON pour dépasser toutes ces limitations.

IV.2.2. Extension du compilateur APOLLON

Au lieu d'avoir des types d'actions opératives fixes, des modèles fixes pour exécuter ces opérations et des éléments fonctionnels prédéfinis, on va utiliser des éléments fonctionnels externes (voir aussi III.3.7). Chaque fonction externe définit :

- les opérations qu'elle sait exécuter,
- le modèle d'exécution de ces opérations,
- le modèle d'interconnexion de la fonction avec les autres éléments fonctionnels,
- la réalisation physique de la fonction, pour les autres étapes de la compilation.

Pour réaliser un compilateur de parties opératives capable de traiter des fonctions externes, il faut commencer par résoudre trois problèmes :

• Trouver un modèle architectural pouvant accueillir les fonctions externes. La définition de ce modèle peut introduire des restrictions sur le type d'opérateurs utilisés.

• Définir un format pour décrire les éléments externes. Ce format doit permettre la description de tous les aspects de la fonction.

• Trouver un algorithme de compilation efficace.

Ces trois problèmes sont analysés ci-dessous. Des propositions de solutions pour l'extension d'APOLLON sont décrites.

IV.8.2.1 Le modèle architectural

Le nouveau modèle utilisera aussi un système de bus. Ce choix permet de simplifier les étapes avancées de la compilation. En fait, les parties opératives à base de bus sont les seules

qu'on sait générer automatiquement de manière efficace. Par contre, le nombre de bus ne sera pas limité. Pour simplifier, on considérera que tous les bus traversent tous les éléments de la partie opérative. Cette simplification nous permet d'avoir une bonne solution topologique. L'organisation de la partie opérative en tranches de 1 bit, avec un modèle topologique similaire à celui d'APOLLON, sera aussi retenue. Ainsi, on pourra réaliser des outils d'évaluation de surface et de vitesse qui travaillent au niveau de l'architecture.

La partie opérative doit aussi utiliser le cycle de base défini pour les parties contrôle. Les seules opérations dont le modèle d'exécution sera prédéfini sont les transferts entre registres. En fait, l'exécution des opérations définies par les fonctions externes sera décomposée en transferts.

On distingue trois types de registres :

• Les registres de données : ils sont déclarés dans la description d'entrée. Ils sont les seuls qui soient directement accessibles à l'utilisateur. Les registres peuvent être connectés aux bus en lecture et en écriture.

• Les entrées d'opérateurs : ils sont accessibles uniquement pour la description du modèle d'exécution des opérations des fonctions externes. Ils sont connectés aux bus en écriture seulement. Ils sont conçus pour que le cycle d'écriture de ce type de registres soit plus court que celui d'un registre de données.

• Les sorties des opérateurs : ils sont, eux aussi, accessibles uniquement pour la description des fonctions externes. Ils sont connectés aux bus en lecture seulement. Leur cycle de lecture est plus court que celui des registres de données.

Les entrées/sorties d'opérateurs peuvent aussi être réalisées par des registres de données. Les trois types de registres permettent la définition de trois types de transferts :

• transfert de type (TDD) : Registre de données <-- Registre de données

• transfert de type (TDS) : Registre de données <-- Sortie d'opérateur

• transfert de type (TDE) : Entrée opérateur <-- Registre de données.

Le modèle d'exécution de ces trois types de transferts sera prédéfini. Le transfert de type TDD nécessite un cycle de base comme c'était le cas pour la première version d'APOLLON. Les autres types de transferts (TDE, TDS) nécessitent seulement la moitié d'un cycle de base pour s'exécuter. Pour cela, il faut que les registres d'entrées soient capables de lire un bus sans que ce dernier soit amplifié. Les registres "sorties d'opérateurs" doivent être capables d'écrire un bus et de l'amplifier de manière à ce que ce bus puisse être lu par un registre de données. Le modèle d'exécution des transferts sera donc : un transfert de type TDD nécessite les 4 temps d'un cycle de base (T1, T2, T3, T4), un transfert de type TDE doit se dérouler durant T1 T2 et un transfert de type TDS doit se dérouler durant T3 T4. Une réalisation possible des trois types de registres est donnée dans [GER88].

Il faut remarquer que les registres d'entrées/sorties d'opérateur sont externes au système, ils sont définis avec les fonctions externes, et doivent respecter les contraintes des modèles de transfert décrits ci-dessus. Ces modèles de transferts permettent la spécification d'opérateurs qui exécutent des opérations en un ou plusieurs cycles.

L'ordonnancement des actions à l'intérieur d'un cycle de base sera ignoré par le compilateur. Il sera réalisé par l'interface électrique entre la partie opérative et la partie contrôle. Dans le cas de la première version d'APOLLON, les commandes de la partie contrôle vers la partie opérative étaient gérées par des amplificateurs qui, en plus de l'amplification des signaux de commandes, réalisaient l'ordonnancement des commandes à l'intérieur d'un cycle de base. Par exemple, dans le modèle utilisé par APOLLON, les transferts entre un registre et un bus sont contrôlés par une seule commande. Le sens du transfert est déterminé par l'instant d'activation de la commande. Si la commande est activée en T2, le contenu du registre est transféré dans le bus, si elle est activée en T4, le bus impose sa valeur au registre. La commande est générée par l'interface électrique à partir de commandes de la partie contrôle. Analysons le cas d'un registre R connecté à deux bus A et B. Cette configuration est montrée par la figure IV.22. Les transferts entre le registre et les bus peuvent se dérouler dans les deux sens.

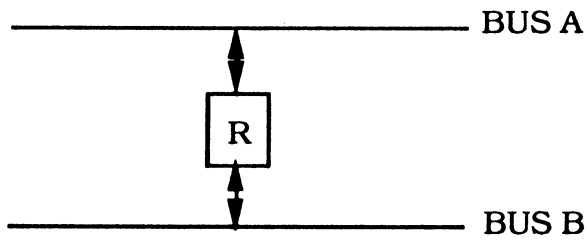


Figure IV.22 Connexion du Registre R

La fonction de l'interface électrique est schématisée dans la figure IV.23. Pour commander un transfert bus-registre, la partie contrôle utilise deux signaux, S (sélection de la connexion bus-registre) et L/E (lecture/écriture). Chaque couple de commandes (S , L/E) est transformé, par l'interface électrique, en une seule commande pouvant être délivrée en T2 ou en T4, selon la valeur de L/E. Ce schéma de fonctionnement est détaillé dans [SCH85].

Dans le nouveau modèle, chaque unité fonctionnelle peut posséder une interface électrique. Cette interface peut réaliser des fonctions plus complexes que celle décrite plus haut.

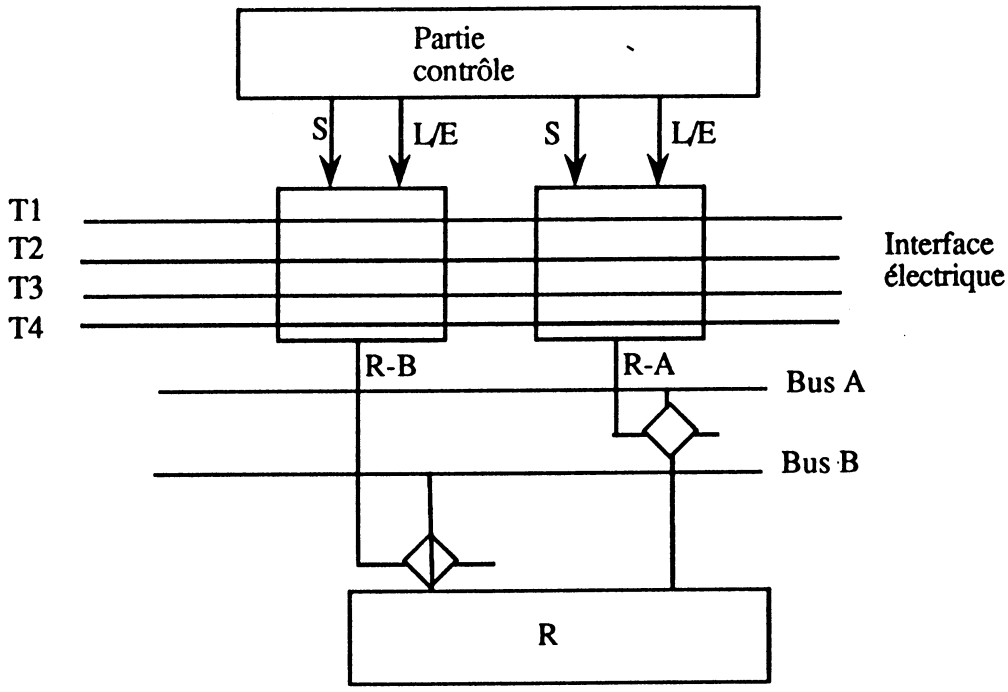


Figure IV.23 Commande du registre R à travers l'interface électrique

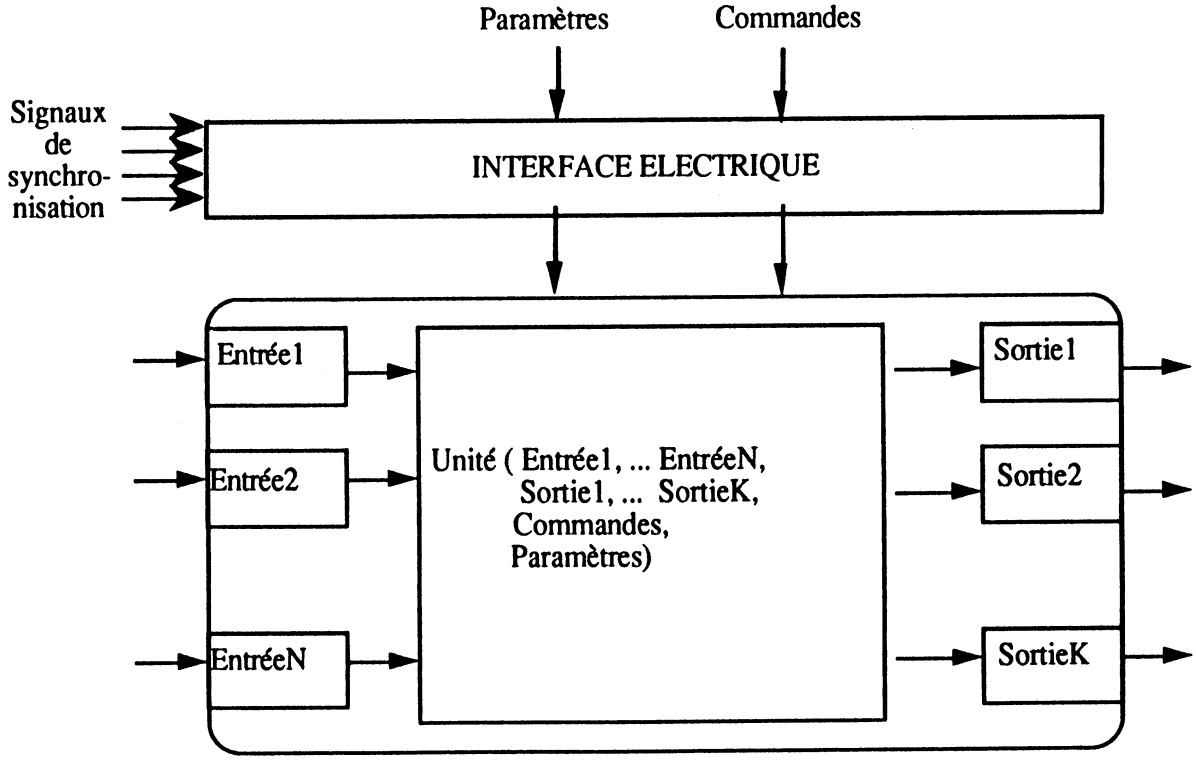


Figure IV.24 Modèle général d'une unité fonctionnelle

Le modèle général d'une unité fonctionnelle est montré en figure IV.24. Ce modèle décrit l'unité fonctionnelle générique du chapitre précédent (III.3.7). L'interface électrique reçoit les commandes et les paramètres de la partie contrôle, elle reçoit aussi des signaux de

synchronisation. Cette unité fonctionnelle peut décrire tous les éléments de la partie opérative. Mais pour les besoins de la description comportementale et des algorithmes de compilation, il faut considérer deux types d'unités fonctionnelles [KU88] :

• Les unités qui ne contiennent pas d'états internes statiques. Elles ne contiennent pas de registres internes autres que leurs entrées/sorties. Elles ne mémorisent pas d'informations entre deux opérations différentes, chaque opération peut durer plusieurs cycles de base. Ces unités correspondent aux opérateurs simples ou multi-fonctions.

• Les unités à mémoire : ces unités peuvent aussi être représentées par l'unité fonctionnelle générique. Elles peuvent contenir des valeurs statiques entre deux opérations. Elles correspondent aux blocs de mémorisation tels que les blocs de registres, les RAMs, les ROMs. Dans le cas de ces unités, la fonction d'adressage peut être réalisée par l'interface électrique.

Dans la suite de cette étude, on se limitera aux registres et aux unités fonctionnelles sans état interne statique.

IV.8.2.2 Description des unités fonctionnelles

Rappelons que le but est de réaliser un système extensible par des fonctions externes. La description de ces fonctions sera appelée description structurelle, et devra donc fournir les modèles d'utilisation de la fonction :

- dans la description comportementale,
- par le compilateur de partie opérative,
- pour la génération du dessin des masques.

Le modèle d'utilisation de la fonction dans la description comportementale peut être celui défini plus haut (voir IV.3). On s'intéressera uniquement aux deux derniers modèles. D'autre part, on ne considérera que les éléments de la bibliothèque accessibles à l'utilisateur, à savoir : les registres, les constantes et les opérateurs. Les bus, par exemple, font partie du modèle global. Le nombre de bus est connu au moment de la compilation. Les bus seront désignés par les noms bus1, bus2... bus n.

Dans le cas d'un registre seul, la description structurelle est nécessaire. Le modèle d'utilisation du registre est prédéfini (voir ci-dessus, IV.8.2.1)). Un registre est défini par :

- un nom, qui définit le type du registre,
- des connexions aux bus (couple bus/commande) :
  - connexions en lecture
  - connexions en écriture

Un format de liste sera utilisé par la description des éléments fonctionnels. Le modèle

général de description d'un registre est :

(Registre <type> (lecture <connexion en lecture>
(écriture <connexion en écriture>))

Exemple : l'expression

(registre R1 (lecture (bus1 c1) (bus2 c2)) (écriture (bus1 c3)))

décrit un registre de type "R1" connecté en lecture aux bus "bus1" et "bus2" par les commandes "c1" et "c2", et connecté en écriture au bus "bus1" par la commande "c3".

Dans le cas du registre "R1" (figure 25), le compilateur ne fait pas d'hypothèse sur la manière dont les connexions aux bus sont réalisées. Aucune hypothèse n'est faite non plus sur le fonctionnement de l'interface électrique.

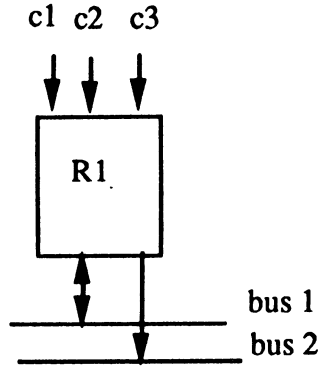


Figure IV.25 Schéma du registre R1

Une constante est décrite par une valeur et des connexions en lecture aux bus (couple bus/commande). La forme générale pour décrire une constante est :

(constante <type> <valeur> <connexion en lecture>)

Exemple : l'expression (constante t 0 (lecture (bus1 c))) déclare une constante de type t de valeur "0" et qui est connectée au bus "bus1" par la commande "c".

La description d'une unité fonctionnelle nécessite à la fois une description structurelle et un modèle d'utilisation. Elle contient :

- un nom,
- des paramètres de contrôle,
- des paramètres de données,
- des registres d'entrées/sorties
- commande (opération)
- modèle d'exécution des opérations.

Le modèle d'exécution des opérations est décrit sous forme de cycles. Chaque cycle

décrit une collection de transferts et d'opérations. Le modèle général pour décrire un transfert est (<typetransfert> <source> <destination>). Le type de transfert peut être TDD, TDE ou TDS pour désigner l'un des trois types de transferts prédéfinis.
Exemple : l'expression (TDS S R) décrit un transfert de type TDS entre la sortie d'opérateur "S" et le registre de données "R". Le modèle général pour décrire une opération est : (opération <unité> <opérateur>). L'expression (opération ALU +) décrit une commande "+" sur l'unité fonctionnelle "ALU".

Le modèle général pour décrire une unité fonctionnelle est :

```
(Unité <nom>
        <paramètres de données>
        <paramètres de contrôle>
        (entrées <liste de registres>)
        (sorties <liste de registres>)
        (opérateurs <symbole>          <vecteur de commande>
                                       <liste de cycles>)
)
```

Exemple : l'expression suivante,

```
(Unité ALU
    (a b c)
    (entrées    (registre E1 (lecture (bus1 c1)))
                (registre E2 (lecture (bus2 c2))))
    (sortie (registre S (écriture (bus1 c3))))
    (opérateurs
       (+ "01000" (cycle (TDD a E1)          (TDD b E2) (opération ALU +))
                  (cycle (TDD S c))
       (- "01001" (cycle (TDD a E1)          (TDD b E2) (opération ALU -))
                  (cycle (TDD S c)))))
```

décrit un opérateur de nom "ALU" capable d'effectuer les opérations "+" et "-". L'opérateur ALU possède deux entrées nommées "E1" et "E2" et une sortie "S". L'opération "+" nécessite deux cycles pour s'exécuter, le premier cycle est décrit par (cycle (TDD a E1) (TDD b E2) (opération ALU +)) et le second cycle par (cycle (TDD S c)).

L'expression décrite ci-dessus donne aussi la description structurelle de l'unité ALU (figure IV.26). La description d'une unité réalisant les mêmes opérations que ALU en un seul cycle doit utiliser des transferts de type TDS et TDE. La description du modèle d'exécution de l'opérateur +, par exemple, devient (+"01000" (cycle (TDE a E1) (TDE b E2) (TDS S c) (opération ALU +))).

La structure de l'unité ALU reste inchangée. On a supposé qu'il était possible de réaliser 2 transferts en parallèle sur le bus1 en un seul cycle de base. Si les contraintes électriques interdisent ce genre de parallélisme, il faudra alors trois bus pour exécuter l'opération "+" en un seul cycle.

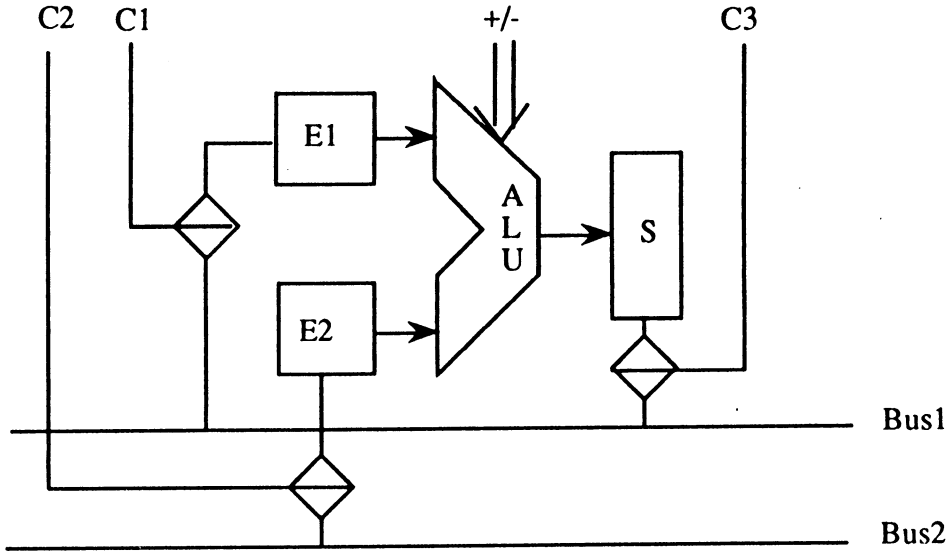


Figure IV.26 Schéma de l'unité ALU

IV.8.2.3. Algorithmes de compilation

L'algorithme de construction de la partie opérative utilisé par APOLLON peut être étendu pour intégrer les extensions du modèle architectural et l'utilisation des fonctions externes. Par contre tous les pré-traitements utilisés par APOLLON doivent être remplacés par de nouveaux processeurs plus adaptés au nouveau modèle.

L'extension portant sur le nombre de bus rend difficile le calcul des contraintes, de placement des registres, utilisées par APOLLON [ARZ84], [JAM86a]. Il faut donc trouver de nouveaux artifices pour limiter l'espace de recherche.

Le modèle de description de fonctions externes, décrit ci-dessus, permet qu'une opération soit réalisable par plusieurs unités différentes et donc être exécutée selon plusieurs modèles différents. Par exemple, on peut avoir le choix entre l'exécution d'une opération en deux cycles de base et en utilisant deux bus ou exécuter la même opération en un seul cycle mais en utilisant trois bus. Le compilateur doit donc tenir compte des différentes possibilités pour l'exécution de chaque opération. Ceci a pour effet d'augmenter l'espace des solutions, mais entraîne également une augmentation du temps de calcul. Plusieurs compilateurs de parties opératives existants utilisent des modèles [KOW85] [DEM86] pour représenter les opérateurs. L'originalité du modèle décrit ci-dessus vient de l'utilisation d'opérateurs pouvant avoir plusieurs sorties. D'autre part l'utilisation d'une interface entre l'opérateur et la partie contrôle permet la synchronisation conjointe de la partie opérative et de la partie contrôle.

Deux tentatives de réalisation d'une nouvelle version du compilateur APOLLON ont déjà

eu lieu. La première est décrite dans [GER88a], elle a permis de mettre au point le modèle architectural décrit ci-dessus. La seconde est en cours [PAR89].

Une autre extension du modèle serait d'autoriser des boîtes à opérations quelconques, càd, qui peuvent contenir une mémoire interne. Ainsi, n'importe quels circuits, même ceux compilés à l'aide de Syco, pourront être utilisés comme boîte à opérations.

IV.9 Optimisation du comportement et de l'architecture

Le succès des outils de compilation va dépendre de l'efficacité des outils d'optimisation associés. Dans le cas des compilateurs de comportement on distingue deux types d'optimisation : l'optimisation de la description comportementale et l'optimisation du découpage, ou optimisation de l'architecture.

Dans le cas où il n'existe pas de correspondance entre la description comportementale et l'architecture des circuits générés, les deux types d'optimisation sont indépendants. Il faut noter que, dans ce cas, l'intérêt de l'optimisation de la description comportementale se trouve réduit faute de modèle pour évaluer l'effet des modifications réalisées par les actions d'optimisation.

IV.9.1 Optimisation de la description comportementale

Comme il a été dit au chapitre précédent (III.5), les techniques utilisées pour l'optimisation de la description comportementale sont similaires aux techniques utilisées par les compilateurs de langages de programmation. Elle permettent surtout d'améliorer la vitesse d'exécution du circuit compilé. Dans le cas du compilateur Syco, ce type d'optimisation est possible en plusieurs points du processus de conception. En fait, l'optimisation de la description comportementale est effectuée sur la forme intermédiaire. Les mêmes outils d'optimisation peuvent être utilisés pour optimiser la description comportementale et l'ordonnancement. Dans ce dernier cas, les outils sont appliqués sur les tables de transitions produites par le compilateur de partie contrôle. D'autre part, vue la correspondance entre la description comportementale et l'architecture, il est toujours possible d'estimer l'effet de chaque action d'optimisation.

Les techniques d'optimisation utilisées par SYCO sont détaillées dans [BEK87] [TLE87]. L'intégration de ces techniques dans un outil d'optimisation, appelé OPC, est décrite dans [TLE87]. Ces travaux ont surtout permis une meilleure formalisation et une meilleure compréhension de la forme intermédiaire. L'optimisation met en œuvre des techniques d'optimisation de microprogrammes. Néanmoins, il faut signaler que les règles d'optimisation utilisées sont différentes (voir III.7).

Tout système d'optimisation de microprogrammes est basé sur deux modèles. Le premier définit le format des micro-instructions et le second définit le modèle de performances qui sera utilisé pour mesurer l'effet des actions de compactage. Dans le cas du compilateur SYCO, la notion de micro-instruction est remplacée par celle de cycle de contrôle. Un cycle de contrôle peut correspondre à plusieurs micro-instructions. Les règles de formation des cycles de contrôle ont été présentées plus haut de manière informelle (IV.5.), et sont énoncées de manière plus précise dans [BEK87] et [TLE87]. Elle sont une adaptation des règles de formation des

micro-instructions [FIS81]. Cette adaptation tient compte des instructions conditionnelles dans un cycle de contrôle.

Rappelons qu'un cycle de contrôle est constitué par un ensemble d'instructions. Une instruction peut être une action simple (action opérative, action de contrôle, branchement, appel de CMODULE) ou une instruction conditionnelle du type:

"IF <condition> <Liste d'actions simples> ENDIF.

Deux instructions peuvent appartenir au même cycle si elles sont mutuellement exclusives ou si elle sont compatibles.

Deux actions sont mutuellement exclusives si elles ne sont jamais exécutées en même temps. Elles doivent donc appartenir à deux instructions conditionnelles différentes dont les expressions de condition sont mutuellement exclusives. Deux expressions de condition sont dites mutuellement exclusives si et seulement si l'expression résultat de leur produit n'est jamais vérifiée.

Soient Ai et Aj deux actions différentes et mutuellement exclusives. Soient aussi RM et RU deux fonctions qui prennent comme paramètre une action et qui retournent respectivement l'ensemble des registres modifiés et l'ensemble des registres utilisés par cette action. Ai et Aj sont dites compatibles si et seulement si aucune des conditions suivantes n'est vérifiée:

- Ai et Aj sont deux actions de branchement (NEXT ou EXIT),
- Ai et Aj sont deux instructions d'appel de CMODULEs (EXECUTE)
- Les ensembles RM(Ai) et RU(Aj) ne sont pas disjoints.
- Les ensembles RU(Ai) et RM(Aj) ne sont pas disjoints.

Les condition exprimées ci-dessus correspondent à des situations de conflit. La description initiale peut contenir des conflits, certains pouvant être détectés durant l'étape de formation de cycles, les autres l'étant durant l'étape d'optimisation.

L'optimisation a pour but la réduction du nombre de cycles de contrôle nécessaires. Elle utilise trois types de traitements:

- élimination des cycles de contrôle non référencés,
- élimination des cycles de contrôle inutiles,
- fusion de cycles de contrôle.

Les deux premiers sont évident à réaliser en partant du graphe de précédence des cycles de contrôle. Ils sont similaires aux traitements réalisés par les compilateurs classiques [AHO87]. L'algorithme utilisé pour la fusion des cycles est similaire à l'algorithme d'optimisation de microprogrammes décrit dans [TOK81].

IV.9.2 Modèle de performance

Tout processus d'optimisation nécessite l'existence de modèles pour estimer l'effet des transformations mises en œuvre durant son exécution. Dans le cas de l'optimisation de la description comportementale ou de l'architecture il est difficile de trouver des modèles universels. Dans le cas du compilateur SYCO, l'utilisation de modèles prédéfinis et de la correspondance entre ces modèles et la description comportementale permet d'estimer la surface et la vitesse. L'utilisation d'une architecture cible permet la définition d'un modèle pour estimer la vitesse d'exécution d'un circuit en partant de sa description comportementale. De même, l'utilisation d'une stratégie globale pour l'organisation des dessins des masques permet la définition d'un modèle pour l'estimation de la surface. Ces modèles sont détaillés dans [BEK87].

Le modèle d'estimation de la surface considère la pile de tranches qui forme le circuit comme un rectangle. La largeur du circuit est déterminée par la tranche la plus large. Dans le cas de la partie opérative, la largeur est déterminée par celle des éléments fonctionnels qu'elle contient. Tous ces éléments, les opérateurs mis à part, peuvent être calculés à partir de la description comportementale. Le nombre d'opérateurs peut être estimé en fonction du parallélisme contenu dans la description comportementale. La largeur d'une tranche de contrôle dépend de son organisation. Dans le cas des tranches de contrôle réalisées par un PLA (figure IV.18), la largeur est déterminée essentiellement par le nombre de monômes du PLA. Elle dépend aussi de la taille des cellules d'interface, qui sont de tailles fixes, et de la taille des bus. La hauteur du circuit résulte de la somme des hauteurs des différentes tranches. La hauteur de la partie opérative est définie par la structure (nombre de bits) de ses éléments. La hauteur d'une tranche de contrôle est définie par le nombre d'entrées/sorties qu'elle contient et par un certain nombre de constantes (hauteurs des cellules prédéfinies). Il faut remarquer que la taille de l'interface entre la partie opérative et la partie contrôle est difficile à estimer.

Pour l'estimation de la vitesse, il faut tenir compte du temps de cycle et du nombre de cycles de base nécessaires pour exécuter une instruction donnée. Comme toutes les tranches utilisent le même cycle de base, c'est la tranche la plus lente qui va imposer la durée du cycle de base. Pour une technologie donnée, le temps de cycle peut être modifié par action sur les cellules d'interfaces (amplificateurs d'entrées/sorties et cellules de précharge). La version actuelle de SYCO utilise un seul jeu de cellules d'interface. Dans ce cas, la durée du cycle de base est déterminée par la complexité des tranches. Dans le cas d'une partie contrôle basée sur une ROM, on peut calculer la fréquence maximale de fonctionnement d'une tranche en partant de la complexité de la ROM, du nombre de mots et de la structure d'un mot (figure IV.19). L'estimation du nombre de cycles nécessaires à l'exécution d'une fonction donnée est plus complexe. Dans le cas où la description contient des boucles ou des instructions conditionnelles, cette estimation nécessite une estimation de la fréquence d'exécution de chaque

instruction. D'autre part, il faut différencier les cycles d'exécution et les cycles dus au modèle, causés par la hiérarchie de la partie contrôle. Cette distinction permet une estimation plus précise de l'effet des transformations qui concernent plusieurs étages de la hiérarchie [JEN80], [STA81]

IV.9.3 Principe de l'optimisation architecturale

L'optimisation architecturale permet d'améliorer un découpage donné. Elle consiste à trouver un bon compromis entre plusieurs critères de performance différents (surface, vitesse, consommation) et non à optimiser selon un critère donné. Cette optimisation correspond à la phase de conception architecturale dans le cas d'une conception manuelle. La recherche de compromis entre les différents facteurs de performance permet au concepteur de trouver la meilleure organisation de son circuit, qui réponde aux contraintes de performance imposées par le cahier des charges. Cette phase de la conception nécessite un concepteur expérimenté qui va utiliser son intuition et son expérience pour évaluer une solution donnée. Dans le cas d'un compilateur de comportement, l'architecture est générée automatiquement par le compilateur. Dans le cas où cette architecture n'est pas acceptable, il faut donc donner au concepteur les moyens de l'améliorer. L'exploration des différentes solutions peut se faire de manière automatique ou alors de manière interactive sous le contrôle de l'utilisateur.

Dans le cas du compilateur SYCO, l'optimisation de l'architecture est possible à travers la description d'entrée ou à l'aide d'un outil d'aide à la recherche de compromis surface-vitesse appelé ARTS [BEK87].

L'utilisateur peut influencer l'architecture d'un circuit à travers sa description en agissant sur plusieurs paramètres. Le nombre de sous parties opératives est défini par le degré de parallélisme contenu dans la description comportementale. Si une description comporte une étape d'interprétation contenant deux opérations (des additions par exemple) devant s'exécuter en parallèle, la partie opérative résultante contiendra forcément au moins deux opérateurs pouvant travailler en parallèle. Pour les parties contrôle, plusieurs paramètres peuvent être fixés par la description comportementale. Le nombre de tranches de contrôle est implicitement fixé par la hiérarchie d'appel des CMODULEs de la description comportementale (cf. IV.6). La taille des bus de contrôle et de comptes rendus est aussi directement fixée par cette description (cf. IV.4.3).

La recherche d'un compromis surface/vitesse est guidée par deux principes généraux :

- pour augmenter la vitesse, il faut diminuer la hiérarchie dans la partie contrôle et/ou augmenter le parallélisme dans la partie opérative.
- pour diminuer la surface, il faut diminuer le parallélisme et/ou augmenter la hiérarchie.

Les deux principes précédents peuvent être combinés localement afin d'ajuster la taille des tranches. La table de la figure IV.27 [BEK87] résume les différentes modifications basées sur ces deux principes. Chacune de ces modifications peut entraîner des "effets de bord" indésirables. Il faut signaler que les modifications données par cette table ne tiennent pas compte de certains paramètres. Les effets sur la surface n'incluent pas la taille des interfaces des étages. Les effets sur la vitesse ne tiennent pas compte des modifications possible du temps de cycle.

	Transformation	But	Effets secondaires possibles
Actions sur la Partie contrôle	Fusion de 2 étages de contrôle	Réduire le nombre d'étages de contrôle pour augmenter la vitesse du cicuit	Augmentation de la surface
	Eclatement d'un étage de contrôle	Réduire la surface occupée par un étage de contrôle	Dégradation de la vitesse
	Fusion partielle de 2 étages de contrôle	Ajuster les largeurs de 2 étages de contrôle pour réduire la surface des interconnexions	- Augmentation locale de la surface - Dégradation de la vitesse
Actions sur la Partie opérative	Augmentation du parallélisme	Augmentation de la vitesse du circuit	Augmentation de la surface
	Réduction du parallélisme	Réduction de la largeur du circuit	Dégradation de la vitesse

Figure IV.27 Transformations d'une description
en vue de l'amélioration du compromis surface vitesse [BEK87]

L'outil d'optimisation architecturale ARTS a été défini pour éviter que des modifications manuelles de la description comportementale introduisent des erreurs au niveau des spécifications du circuit compilé. ARTS permet de réaliser des modifications de la description comportementale sans altérer la description initiale. Il peut être utilisé pour améliorer le découpage de manière interactive. Il faut remarquer que dans le cas de SYCO on distingue deux types de découpage, le découpage en niveaux d'interprétation et le découpage partie opérative/partie contrôle réalisé par le compilateur de partie opérative. Ce dernier découpage n'est pas modifiable par la version actuelle de ARTS. Ceci est dû à la rigidité du modèle utilisé par la version actuelle du compilateur APOLLON. La nouvelle version, définie ci-dessus (IV.8), permettra une plus grande marge de manœuvre pour la modification du découpage en partie opérative/partie contrôle.

IV.9.4 Utilisation de ARTS pour la modification architecturale

La version actuelle de ARTS est limitée aux modifications concernant la partie contrôle. Pour plus de détails sur le système ARTS, on peut se reporter à [BEK87]. Le fonctionnement de ce système sera illustré par un exemple.

Le point de départ de cet exemple sera la description du micro-processeur simplifié (figure IV.6). La compilation de cette description produit un circuit dont la partie contrôle est composée de trois étages. Trois autres solutions architecturales ont été obtenues en utilisant le système ARTS.

	solution 1			solution 2		solution 3		solution 4	
tranche de CTRL:	2	1	0	1	0	1	0	1	0
entrées	7	6	10	7	11	11	10	10	11
sorties	15	22	16	15	26	25	16	18	25
monômes	13	15	17	13	59	39	17	31	29
surface mm2	2.53			2.95		2.58		2.40	

Figure IV.28: Résumé des quatres solutions architecturales du circuit "mini"

Les quatre solutions sont résumées dans la table de la figure IV.28. La seconde solution est obtenue à partir de la solution initiale en fusionnant les deux tranches de contrôle inférieures. La tranche de contrôle résultat de la fusion est jugée trop large. Toujours en partant de la solution initiale, on réalise la fusion des deux tranches supérieures de contrôle. On obtient la troisième solution. Elle est plus régulière que la première, mais les deux tranches de contrôle qui la constituent sont de largeur différente. Dans la quatrième solution, la partie contrôle est composée de deux tranches de largeurs respectives quasi égales. Elle a été obtenue à partir de la solution initiale, après application de la séquence de transformations suivante:

- Déplacer le CMODULE "sta" (renommé sta_g191_1, voir figures IV.16 et IV.17) de la tranche de contrôle du milieu vers la tranche inférieure. Cette modification entraîne un élargissement de l'étage inférieur et un rétrécissement de l'étage du milieu.
- Fusionner les deux étage supérieurs.

Ces modifications n'entraînent pas de modification de la partie opérative. La solution obtenue est plus rapide par rapport à la situation initiale. L'amélioration de la vitesse est due à l'élimination des cycles utilisés pour propager les appels de CMODULEs dans la solution initiale

ARTS a aussi été utilisé pour la compilation du microprocesseur 6502 [REI86]. Dans ce cas, la solution initiale obtenue à partir de la description comportementale a produit un circuit avec trois tranches de contrôle. La surface de ce circuit a été évaluée à 28.9 mm^2 [GER87]. L'utilisation de ARTS a permis d'obtenir un circuit plus petit, de surface 24.27 mm^2, et qui ne contient que deux tranches de contrôle.

IV.10 Génération des dessins de masques

La génération du dessin des masques est une étape nécessaire pour valider les résultats d'un compilateur de comportement. Cette constatation a été prise en compte dès le démarrage du projet SYCO. La génération des dessins de masques a permis de valider les choix du compilateur SYCO. D'un autre coté, elle a constitué un frein au développement du compilateur (voir plus loin).

Dans la première version de SYCO, la génération des dessins de masques est réalisée par un ensemble de modules générateurs, écrits en LUBRICK [SCH83]. Elle met en œuvre un ensemble de fonctions et d'outils pour la construction de cellules complexes en partant des éléments de la bibliothèque de base. Les modules générateurs, dans cette version, étaient basés sur une technologie NMOS. La stratégie globale d'implantation utilisée pour cette version s'était avérée peu efficace dans le cas des parties contrôle [VAR87]. Par contre, elle a donné des résultats satisfaisants dans le cas des parties opératives. La figure IV.29 montre la partie opérative du microprocesseur 6502, générée automatiquement par le compilateur APOLLON. Le dessin des masques est organisé en tranches de bits. La partie opérative résulte de l'empilement de n tranches de un bit. Cette stratégie d'organisation de la partie opérative est détaillée dans [JAM86a]. Elle a été retenue pour la 2ème version du compilateur SYCO qui devait être réalisée pour une technologie CMOS. Ce changement de technologie a mis en évidence les problèmes de l'indépendance de SYCO vis-à-vis de la technologie. Avant de présenter les choix retenus pour la réalisation de la version CMOS des outils de génération des dessins de masques, analysons les solutions possibles pour résoudre ce problème.

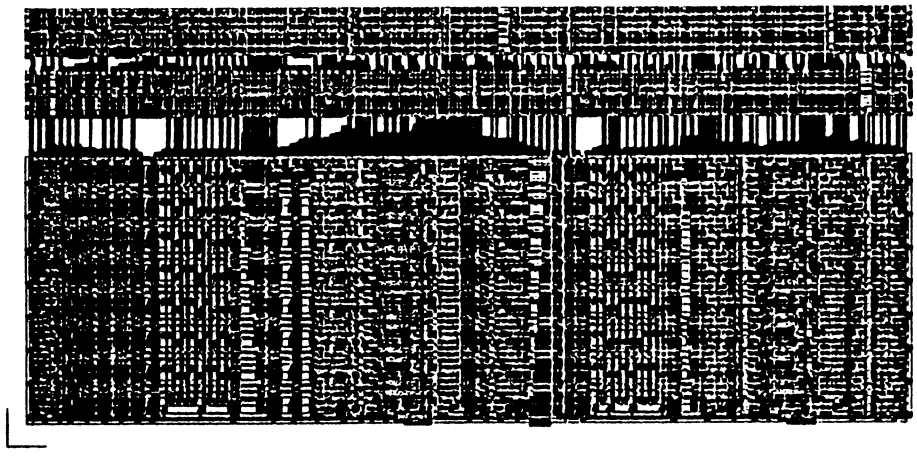


Figure IV.29 Dessin des masques de la partie opérative du 6502 [JAM86a]

IV.10.1 Génération des dessins de masques pour les compilateurs de comportement

Bien qu'elle ne mette en œuvre que des algorithmes bien maîtrisés, la génération des dessins de masques constitue à la fois l'épine dorsale et le goulet d'étranglement de tout compilateur de comportement. C'est l'épine dorsale car, sans elle, on ne peut valider les autres outils de compilation ; c'est un goulet d'étranglement car elle nécessite des efforts de développement très grands, qui sont anéantis à chaque changement de technologie. Dans le cas du compilateur SYCO, cette situation est aggravée par l'utilisation d'outils de génération de dessins des masques développés spécialement à cette fin. Avec le développement actuel des compilateurs d'architecture et des environnements pour la compilation de silicium, plusieurs solutions sont possibles pour maîtriser la génération des dessins des masques.

Une première solution consiste à utiliser un compilateur d'architecture tel que GENESIL [CHE88] comme environnement pour le compilateur de comportement. Un tel environnement fournit des compilateurs spécialisés (compilateur de contrôleurs, compilateur de parties opératives) et une bibliothèque de cellules standards. Dans ce cas, la tâche du compilateur sera réduite à la génération de l'architecture. Dans le cas où le compilateur d'architecture utilisé ne réalise pas de placement automatique, il faut générer aussi un placement initial des blocs. Cette solution résout complètement les problèmes de génération des dessins des masques, puisqu'elle permet des changements importants de technologie ; mais elle rend difficile l'utilisation d'une stratégie de plan de masse, car la topologie des cellules standards et celle des blocs générés par les compilateurs spécialisés sont fixes et peuvent ne pas être bien adaptées à la stratégie d'implantation, or, sans cette stratégie, il est difficile de réaliser des outils d'évaluation qui sont à leur tour nécessaires pour réaliser des outils d'optimisation architecturale (voir IV.9).

Une seconde solution consiste à utiliser un système de dessin symbolique tel que GDT [BUR86] ou STYX [JER85], [ROU87], [CHA88]. Dans ce cas, les éléments de la bibliothèque seront décrits sous forme symbolique, et les compilateurs spécialisés généreront des dessins symboliques. La transposition dans la technologie finale se fera par des outils d'expansion et de compactage de la description symbolique. Cette solution ne couvre que des changements technologiques mineurs, tels que l'évolution d'un procédé de fabrication. Par contre, elle ne permet pas l'introduction d'un nouveau niveau d'interconnexion. Cette solution est bien adaptée dans le cas d'un processus de conception manuel, par contre elle est difficile à mettre en œuvre dans le cadre d'un compilateur de silicium. Les systèmes symboliques existants mettent en œuvre des algorithmes qui supportent mal les descriptions hiérarchiques [WOL87] et dont le résultat dépend énormément de l'organisation initiale du dessin. Ces algorithmes demandent, généralement, à être guidés lors du processus de compactage. D'autre part, les systèmes symboliques sont très peu adaptés à la génération, l'expansion, et le compactage des structures régulières. Analysons ce fait dans le cas de la génération d'une ROM par assemblage de motifs de base. L'assemblage d'une ROM de 100 mots de 100 bits chacun,

dans le cas du système RG [MAC89], contient 14000 appels de cellules. En plus des points mémoire, 10 000 occurrences dans notre cas, l'assemblage d'une ROM utilise plusieurs autres cellules telles que les cellules de rappel de masse. Si on suppose que chaque cellule utilisée a en moyenne 4 points de connexion (connecteurs) et que chaque connecteur occupe 100 octets dans la structure de données, la représentation de la ROM va nécessiter 5,6 million d'octets, uniquement pour les connecteurs. Ces remarques font que les systèmes symboliques actuels sont peu pratiques pour servir d' environnement pour la compilation de silicium. Ce point est détaillé dans [WOL87].

Une troisième solution consiste à utiliser des générateurs de cellules et de structures régulières, paramétrés par la technologie. Cette solution est difficile à mettre en œuvre en tant que telle, mais si l'on sépare la génération des motifs de base de l'assemblage de ces motifs et si on considère que les outils de génération de structures régulières produisent des motifs, elle devient viable et donne de bons résultats. Cette simplification nous conduit à un système symbolique où les motifs de base sont définis non plus comme les éléments de base de la technologie, mais en fonction des besoins du compilateur de silicium. Les développements actuels du compilateur SYCO sont orientés vers cette solution. Un prototype d'environnement pour la génération des dessins de masques appelé NAUTILE [HOR89][BON89] a été réalisé pour montrer l'efficacité de cette solution. La suite de cette section donne une présentation brève du système NAUTILE, cette présentation est basée sur les travaux de BONDONO [BON89] et HORNIK [HOR89].

IV.10.2 Le système NAUTILE [BON89a]

NAUTILE est un environnement pour la génération des dessins de masques. Il est adapté au besoin des compilateurs de comportement. Il permet la réalisation d'outils d'assemblage et de générateurs paramétrés par la technologie. La présentation qui suit est extraite de [BON89a]

NAUTILE permet la manipulation de plusieurs vues d'un même circuit. Une cellule possède deux types de vues, des vues physiques et une "vue de construction". Cette dernière permet de concevoir des circuits "corrects par construction", et met en œuvre des règles de composition paramétrées par la technologie. NAUTILE combine les principes de deux systèmes existants, SKILL de Cadence [LAW86], qui autorise la manipulation d'un nombre de vues non fixé, et PALLADIO [BRO83], qui est un environnement de conception permettant de modifier à la fois les outils et les langages de description des circuits. PALLADIO considère que les différentes vues du circuit, appelées ici "perspectives" (comme dans [KAT86]), décomposent les cellules en différentes hiérarchies (une par vue), les éléments de base étant assemblés à l'aide de différentes règles de composition.

IV.10.2.1 Spécifications initiales

Les spécifications de NAUTILE ont été définies pour répondre aux besoins du compilateur de silicium SYCO afin de permettre de décrire des circuits indépendants de la technologie, et de développer des générateurs de cellules. Il ne s'agit donc pas d'un système général pour la conception de circuits. Cette limitation va permettre de combler un certain nombre de lacunes, et tenter de résoudre des problèmes qui ne l'étaient pas forcément par les systèmes déjà existants. Le cahier des charges devait notamment permettre :

- des changements minimes de la technologie à moindre frais,
- une description des circuits suivant de multiples niveaux d'abstraction et de manière hiérarchique,
- un traitement unifié de ces différents niveaux avec maintien de la cohérence entre eux,
- la possibilité de produire rapidement et aisément un dessin correct,
- la possibilité d'intégration dans un environnement ou une méthodologie de conception déjà existant, incluant notamment la faculté de lire et d'écrire des parties de circuit dans des systèmes externes.

IV.10.2.2 Organisation générale de NAUTILE

Le système NAUTILE gère une structure de données orientée objet pluraliste (c'est-à-dire pouvant intégrer des données externes à NAUTILE). Elle contient les différents objets composant les diverses représentations d'un circuit. Les primitives nécessaires à la manipulation de ces objets sont communes à tous les outils, fournis par le système et les outils et générateurs réalisés par l'utilisateur. Ces outils n'accèdent à la structure de données qu'au travers de primitives de manipulation de la base de données. La figure IV.30 montre l'organisation générale du système NAUTILE.

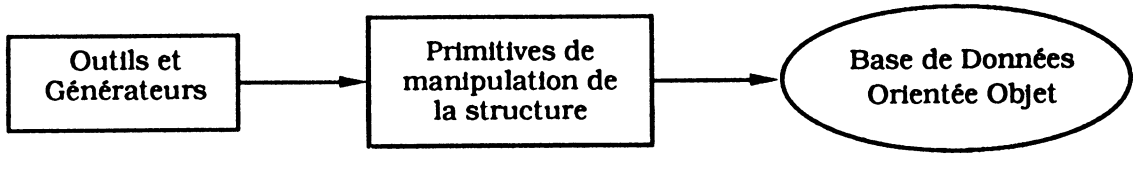


Figure IV.30 Organisation générale du système NAUTILE

IV.10.2.3 Notions de cellules et de motif

L'entité de base manipulée est la cellule, chaque cellule réalisant une fonction particulière. Une cellule est composée soit d'éléments de base (décrivant la technologie), soit d'autres cellules de composition. La description du circuit est donc hiérarchique, chaque cellule pouvant se décomposer en d'autres cellules, elles-mêmes à leur tour décomposables. Au niveau le plus bas de la hiérarchie (les "feuilles" de l'arborescence), les cellules sont appelées en NAUTILE des "motifs". Un motif est donc un élément terminal, qui ne peut être lui-même

décomposé en d'autres cellules ou motifs.

Une cellule est définie par : un ensemble de paramètres, un ensemble de définitions locales, la vue de construction de la cellule et éventuellement des représentations physiques de la cellule.

Nous distinguons différents types de motifs :

• Les motifs prédéfinis dans le système (on parle alors de "motifs système"). Il s'agit d'éléments de base de la technologie, tels que les transistors, les fils et les contacts.

• Les motifs définis manuellement par l'utilisateur, appelés "motifs utilisateur". Ce sont par exemple des portes élémentaires, des cellules simples, etc. Ces motifs peuvent être soit importés d'une bibliothèque déjà existante, soit créés pour les besoins immédiats. En général, les motifs utilisateur constituent une bibliothèque de cellules prédéfinies adaptées à une application donnée.

• Les motifs créés par des outils, tels que routeurs, générateurs de plan de masse, compacteurs, ... Ils peuvent aussi résulter de l'évaluation d'un générateur (ex: générateur de PLA).

Un circuit est donc un assemblage de motifs, ceux-ci pouvant être décrits suivant plusieurs types de représentations. Le système NAUTILE effectue un traitement unifié, avec un langage unique, sur les différentes représentations d'un motif.

IV.10.2.4 Notion de vues : représentations multiples d'un objet

NAUTILE permet de prendre en compte les différentes représentations d'un circuit. Ces différentes représentations s'appellent des "vues". En plus des deux types de vues cités plus haut, les vues physiques et la vue de construction, chaque cellule ou motif possède une vue externe qui est commune à toutes les représentations.

Les vues physiques (vue topologique, électrique, logique, temporelle, ...) décrivent les cellules de façon particulière à chaque type de représentation. Il peut y avoir plusieurs vues d'un même type : par exemple deux vues topologiques. Cette possibilité de diversifier les vues, et d'accepter pour celles-ci n'importe quel format (pourvu que l'on ait défini les primitives de manipulation de la base de données), constitue une des grandes originalités de NAUTILE : celui-ci n'impose aucune restriction au départ, et permet, suivant l'application considérée, de choisir les représentations qui doivent être manipulées.

La vue externe est "l'interface" du circuit, telle que définie en EDIF [MAR87] (en revanche, EDIF regroupe toutes les autres vues sous le terme "implémentation", alors que nous distinguons la vue de construction des vues physiques).

Cette vue externe exprime les relations de la cellule avec son environnement (elle représente la cellule "telle qu'elle est vue pour son utilisation") : paramètres (tels que paramètres de taille, facteurs de répétition, ...), boîte enveloppante (encombrement de la cellule, représentée pour des raisons d'efficacité par un rectangle), connecteurs (moyens de communication de la cellule avec l'extérieur), etc. Ainsi, dans un souci de simplification, les cellules sont considérées de l'extérieur comme des boîtes, avec leurs connecteurs positionnés au bord de ces boîtes, ce qui permet de les manipuler sans tenir compte de ce qu'elles contiennent.

Il est important de remarquer que, contrairement à d'autres systèmes (dont OCT [MOO86]), NAUTILE considère que la plupart des paramètres et des attributs (notamment les connecteurs), sont communs à toutes les vues. Ceci permet, outre une simplification du problème, d'assurer des liens entre les vues, et surtout de maintenir la cohérence entre elles. De plus, cela permet aux différentes primitives et outils NAUTILE de posséder des sémantiques multiples : ils agissent simultanément sur toutes les vues (nous reviendrons sur ce point très important, dans le paragraphe IV.10.2.7 consacré à l'environnement de travail). Une telle restriction est inacceptable dans le cas d'un système destiné à la conception manuelle de circuits. Par contre, dans le cas du système NAUTILE, cette restriction n'en est pas une car les circuits sont réalisés à partir d'une seule représentation, qui est la vue de construction.

La vue de construction décrit la cellule de façon hiérarchique. Elle est donc formée d'une hiérarchie de cellules de base ou de composition, ces dernières n'ayant pas de fonctionnalité propre, mais n'étant qu'un répertoire des cellules qui les composent (appelées "instances") et d'interconnexions. Par exemple, l'assemblage montré par la figure IV.31 peut être décrit, en NAUTILE, par la suite d'actions suivante:

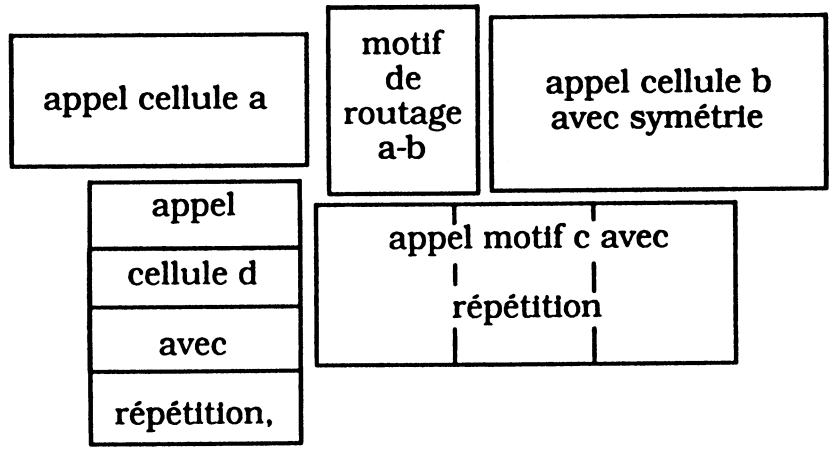


Fig IV.31 Exemple de de construction d'une cellule [BON89a]

```
(defcell abcd)     ;définition de la cellule de nom "abcd"
(instance d (0,0) (rep 4 nord))          ;appel de d en (0,0) répété 4 fois
(direct c est (0,10) (rep 3 est))        ;aboutement de c avec décalage de (0,10)
(direct a nord (-10,0))                  ;aboutement de c avec décalage de (-10,0)
(rout b est (0,0) (sym x))               ;appel du routeur pour ajouter b
(fin-accès)                              ;fin de définition de la cellule
```

Cet exemple illustre bien le mode de fonctionnement de NAUTILE : on peut noter que la vue de construction de la cellule représentée utilise aussi bien des appels de cellules (a, b et d), que de motifs (c) ou même d'outils (le motif de routage entre les cellules a et b).

En effet, NAUTILE considère que chaque outil construit des motifs intermédiaires, utilisables comme n'importe quels autres motifs.

Chaque cellule peut avoir plusieurs vues. Seules les cellules de composition possèdent une vue de construction et des vues physiques, les motifs possédant uniquement la vue externe. Aussi est-il indispensable, lors de la construction d'un circuit, de s'assurer de la disponibilité de tous les motifs nécessaires à la définition de ce circuit, faute de quoi il ne sera pas possible d'extraire toutes les vues du circuit.

IV.10.2.5 Maintien de la cohérence

Pour assurer ce maintien de la cohérence entre les différentes vues, le système NAUTILE utilise des règles simples :

si les motifs sont corrects par définition, et
si le circuit est décrit hiérarchiquement avec la vue de construction, et
si les règles de construction ont été correctement formalisées, et
si le concepteur n'a modifié aucune vue physique du circuit,
alors le circuit est correct par construction.

Dans la plupart des autres cas, des outils de vérification classiques sont nécessaires, comme ils le sont pour vérifier la cohérence au niveau des motifs.

De ces différentes contraintes, la première est relativement faible : en effet, il est relativement aisé de prouver sur n'importe quelle station de travail un peu évoluée, la correspondance entre les vues physiques d'une cellule simple.

En revanche, la contrainte de ne jamais modifier une vue physique est une contrainte très forte pour les concepteurs qui l'acceptent difficilement. Encore une fois, cette contrainte n'en est pas une, dans le cas système NAUTILE, car les vues physiques des cellules générées automatiquement ne sont pas destinées à être modifiées.

Les mécanismes de gestion de la cohérence sont basés sur la chronologie des appels, méthode déjà utilisée par d'autres systèmes [ROU87].

IV.10.2.6 L'indépendance technologique

L'indépendance vis-à-vis de la technologie est essentiellement due à l'existence d'un fichier technologique, constitué :

• d'un ensemble de constantes,

• de règles de construction définissant les contraintes d'assemblage des cellules entre elles (un jeu de règles par vue physique),

• d'un ensemble de motifs.

L'ensemble des motifs de base d'une technologie minimale comporte le transistor, le fil et le contact, mais en réalité les motifs de base nécessaires à la conception de circuits sont plus nombreux et plus complexes. Dans le cas du système NAUTILE la notion de motif est encore plus libre.

Les trois composantes du fichier technologique doivent être spécifiées pour toutes les vues physiques qu'on veut utiliser (IV.10.2.7).

Les règles d'assemblage doivent assurer que le circuit sera correct. Elles sont vérifiées lors de chaque utilisation d'une primitive de construction. Elles permettent de paramétrer la technologie pour les outils d'assemblage, et sont pour l'instant limitées à des règles électriques et topologiques.

IV.10.2.7 Interfaçage avec les systèmes externes

L'un des points forts du système NAUTILE est de pouvoir manipuler des données issues de systèmes externes. Une cellule peut avoir ses différentes vues décrites dans n'importe quel système (y compris le système NAUTILE!). La seule limitation est que tous les motifs utilisés pour sa construction soient définis dans toutes les vues qu'elle possède. Chaque vue peut alors être expansée dans une représentation externe, en utilisant à chaque fois un processeur d'expansion spécialisé

La figure IV.32 illustre le mode d'interfaçage de NAUTILE avec les systèmes externes. Le nombre de vues gérées par le système n'est pas limité. Les objets à redéfinir lors de la création d'une nouvelle vue (par exemple une vue spéciale utilisée pour une simulation particulière) sont :

• les motifs devant être utilisés pour cette vue particulière,

• les règles d'assemblage définies dans le fichier technologique,

• une fonction "charge" permettant d'importer une cellule décrite dans cette vue,

• une fonction "génère" qui produit à partir de la vue de construction d'une cellule sa représentation dans la nouvelle vue.

En outre, les primitives de construction doivent être enrichies pour réaliser des assemblages dans la nouvelle vue.

La structure de données décrivant les objets NAUTILE est orientée objet [KAT86]. Ainsi les objets répondent à des messages qui leurs sont adressés, sans que les messages en question

connaissent la nature des objets auxquels on les destine : la plupart des primitives définies dans NAUTILE possèdent des sémantiques multiples, ce qui signifie qu'elles s'appliquent de façon identique à différents types d'objets. Ces sémantiques sont en fait multiples à plusieurs niveaux : elles s'appliquent aussi bien aux différentes vues qu'aux différents environnements (graphique ou textuel), ou aux différents types de cellules (cellules composées ou motifs).

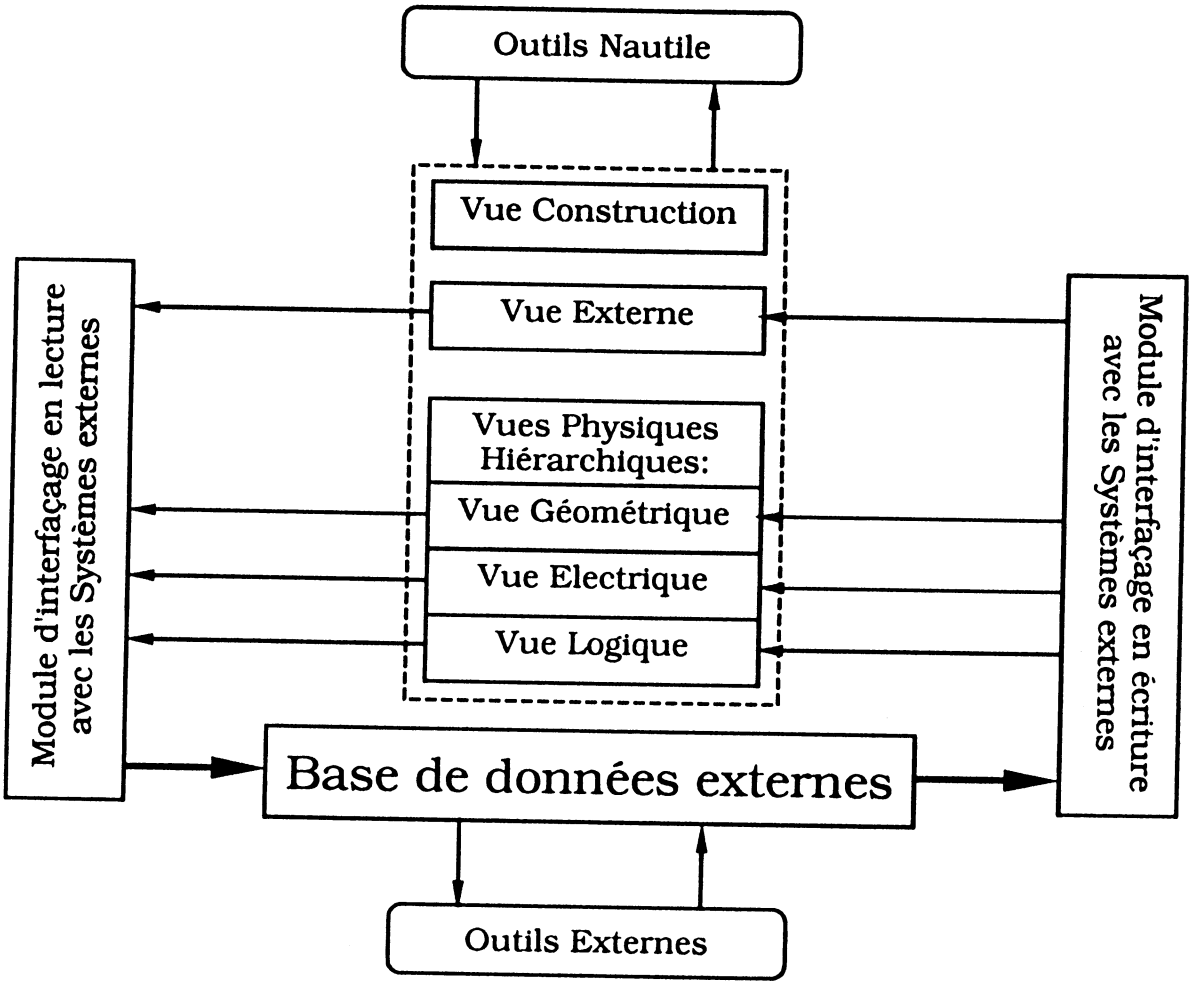


Figure IV.32: Interfaçage de NAUTILE avec un système externe [BON89a]

Plus clairement, cela signifie qu'il n'y aura, par exemple, qu'une seule fonction de routage, applicable aussi bien aux cellules qu'aux motifs, dans n'importe quelle vue, et utilisable de façon similaire en modes graphique et textuel. Par exemple, la primitive "rout" permet la définition d'une cellule de routage reliant deux cellules. L'appel suivant :

(rout Est AB A B <ensemble-de-connexions>)

va générer une cellule appelée AB, qui contient les cellules A et B. L'interprétation logique de ce résultat sera l'ensemble des connecteurs générés, alors qu'une interprétation géométrique en serait le placement de B à l'est de A, et la génération du canal de routage avec les différents fils joignant les différents connecteurs. Si on rajoute une nouvelle vue dans le système, il faut définir

une nouvelle sémantique pour préciser l'interprétation de la primitive "rout" dans la nouvelle vue.

IV.10.2.8 Le système NAUTILE : applications

L'usage initial de NAUTILE est la construction de générateurs pour un compilateur de comportement. La description des cellules est procédurale, une cellule étant décrite par une fonction dont l'évaluation crée dans la structure de données la cellule proprement dite. Les primitives de construction de NAUTILE ont une sémantique graphique, ce qui permet de visualiser le résultat de l'assemblage.

La définition d'une cellule peut donc être décrite comme suit : on crée la cellule, on l'ouvre, on y ajoute des éléments, on la ferme pour examiner une autre cellule, on y revient ultérieurement pour la modifier, fixer un certain nombre de paramètres, etc. Ce mode de construction est aussi adapté à la construction graphique des cellules.

La figure IV.33 décrit la façon dont NAUTILE est utilisé : partant des deux motifs "A" et "B" définis dans deux vues physiques externes (RNL et LUCIE), le concepteur peut construire une cellule de composition résultant d'un assemblage de ces deux cellules. Il décrit la vue de construction dans le fichier "AB.til", et le système génère alors automatiquement les vues physiques de la cellule résultante (dans les fichiers "AB.net" pour RNL, et "AB.luc" pour LUCIE).

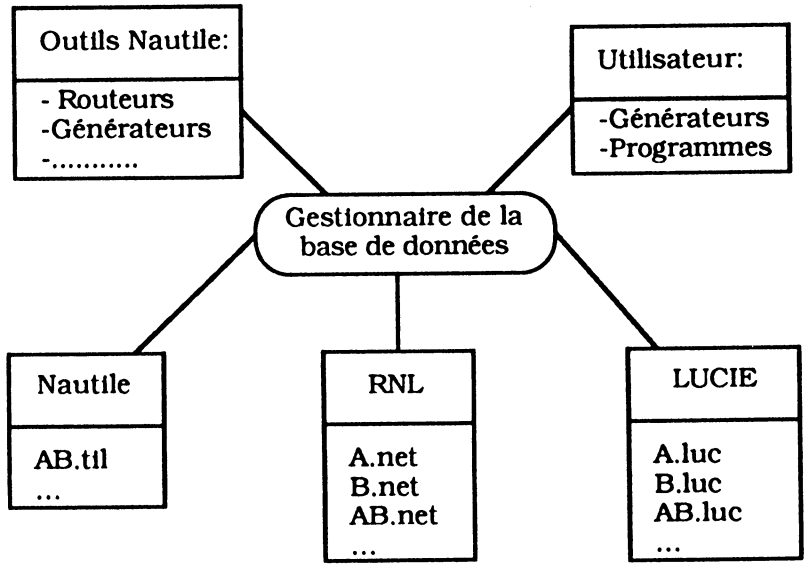


Figure IV.33 : Exemple d'utilisation de NAUTILE [GER88]

NAUTILE a été utilisé pour la génération de la partie contrôle du microprocesseur 6502 [GER88]. Ce circuit, dessiné en technologie CMOS 2µm, deux niveaux de métal, utilise:

- environ 30 motifs différents (incluant les plots d'entrées/sorties),
- 3 sortes de routeurs, deux provenant d'un environnement externe, en l'occurrence la

société BULL (routeurs de type rivière et canal), et un routeur développé à l'aide de NAUTILE (routeur de plots) [GER88].

Le plan de masse du circuit est montré en figure IV.34. La figure IV.35 montre le dessin des masques complet. Les structures régulières (ici, des PLAs) sont construites par un ensemble de générateurs spécialisés décrits en NAUTILE. Ces générateurs fonctionnent à deux niveaux :

- génération de la vue externe avec les connexions,
- génération du dessin des masques.

Ceci permet d'assembler le circuit et de le tester, la génération du dessin des masques n'étant effectuée qu'en phase finale de la conception. Ces deux modes correspondent aux dessins montrés en figures IV.34 et IV.35. La partie contrôle de la figure IV.35 est composée de deux tranches de contrôle produites par le même générateur. Ce générateur inclut des appels à un routeur, à des fonctions de placement, et à des motifs.

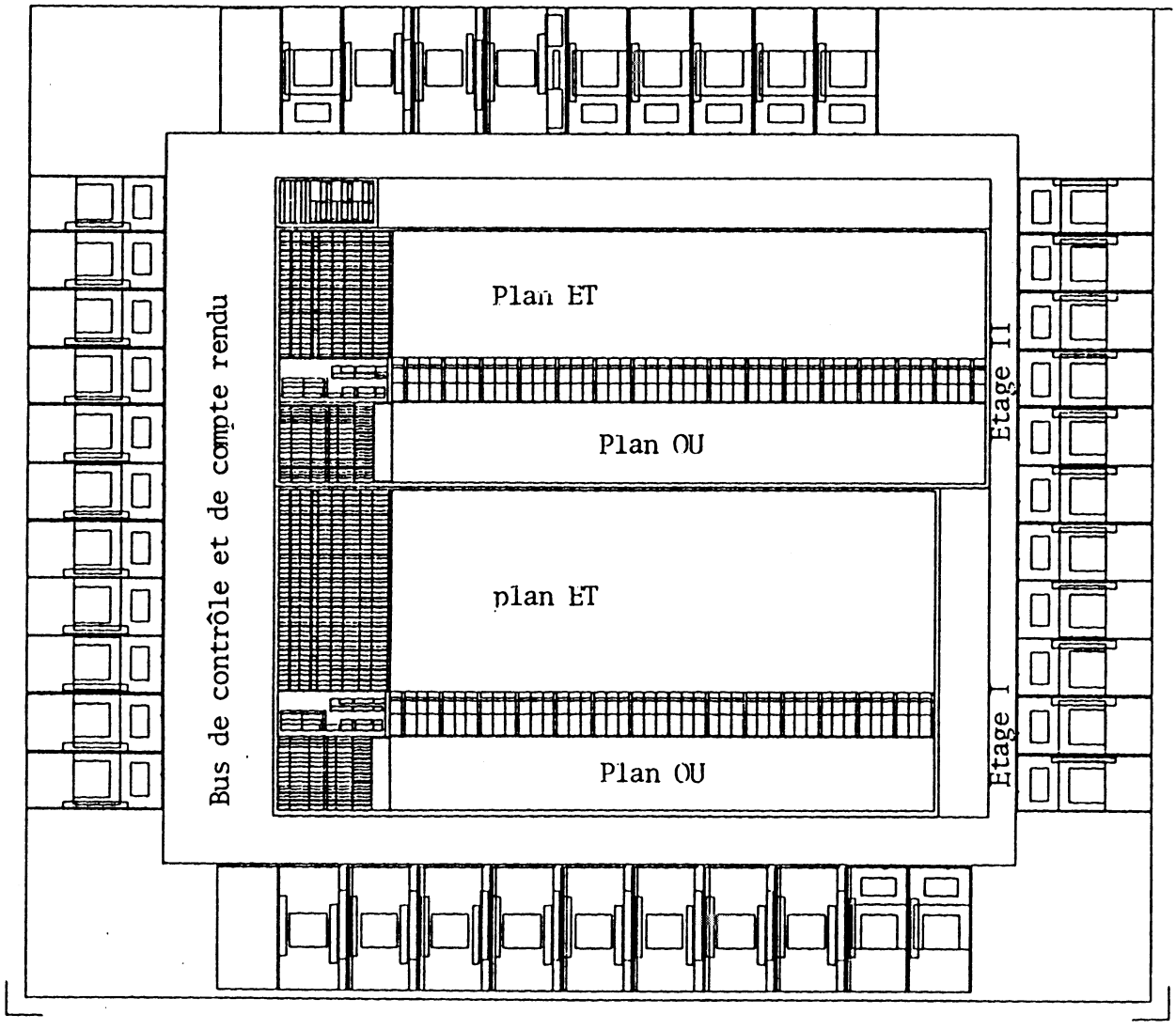


Figure IV.34 Plan de masse de partie contrôle du 6502 [HOR89]

Actuellement le système est encore limité par une interface graphique très pauvre, seul l'affichage des cellules et quelques primitives d'édition étant disponibles dans la version courante. L'expérience menée avec le compilateur SYCO a confirmé l'intérêt des principes énoncés pour la définition de NAUTILE. Bien que le système impose des contraintes relativement lourdes dans le cas d'une utilisation pour la conception manuelle de circuits, il est bien adapté à la construction de générateurs pour la compilation de comportement.

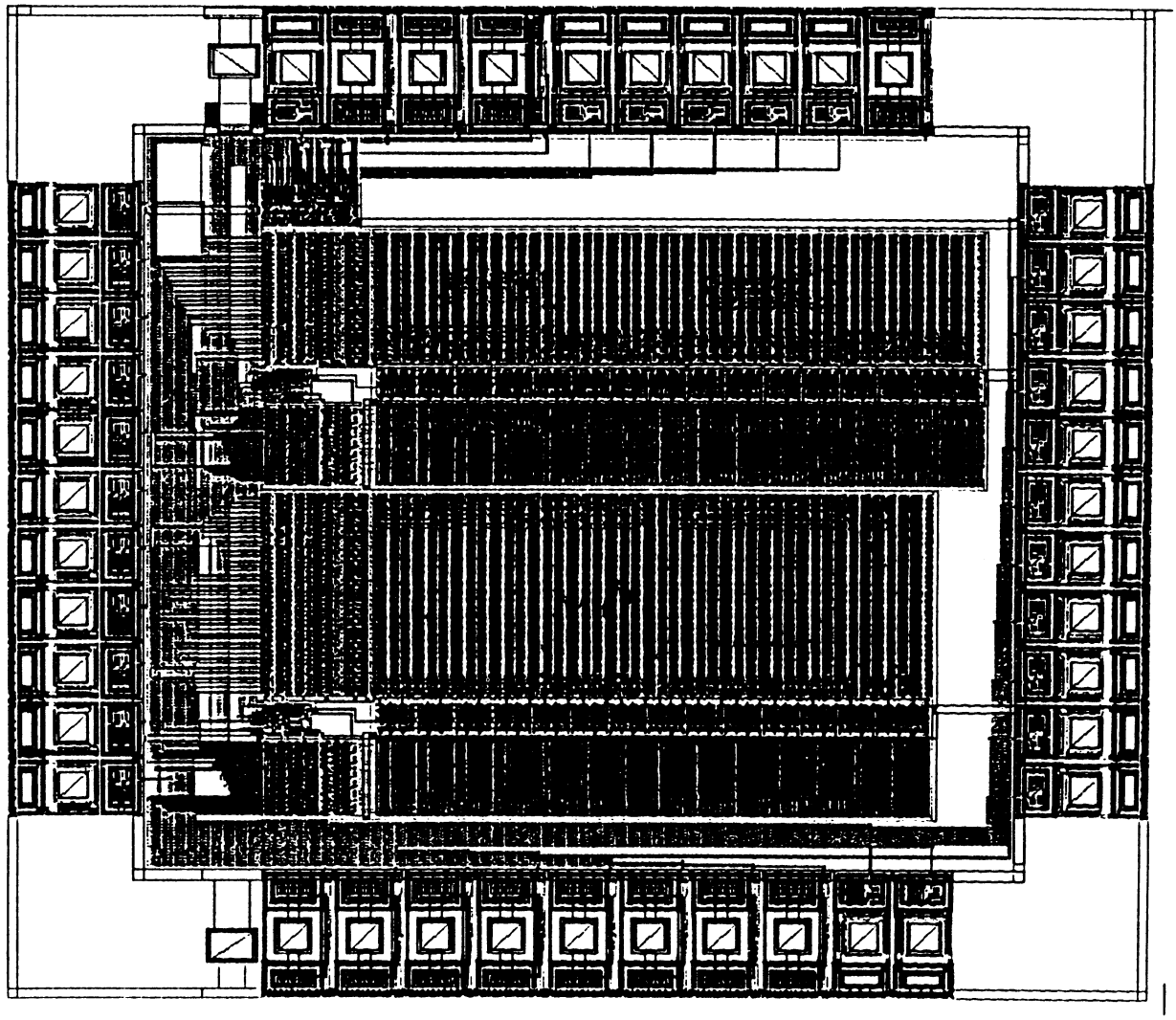


Figure IV.35 Dessin complet des masques de la partie contrôle du 6502 [GER87]

IV.11 CONCLUSION

Le projet SYCO a démarré en 1984. Plus de 20 chercheurs et étudiants ont participé de près à son développement. Aujourd'hui, SYCO est constitué de quelques 30000 lignes de programmes en Le_Lisp.

L'évolution rapide des technologies de fabrication a empêché la réalisation d'une version complètement opérationnelle de SYCO. La vitesse avec laquelle les bibliothèques de cellules étaient réalisées était inférieure à la vitesse d'évolution des technologies : dès qu'une bibliothèque de cellules était prête, le processus de fabrication correspondant devenait obsolète. L'utilisation de NAUTILE doit aider à surmonter ce problème. NAUTILE pourra être combiné avec un environnement de CAO tel que SOLO2000™ d'ES2 par exemple.

D'autres développements sont en cours pour préparer la prochaine version de SYCO. Ils portent essentiellement sur l'assouplissement de l'architecture cible utilisée par SYCO [PAR89] et sur les méthodes de test des circuits générés [TOR88].

Bien que paramétrable, l'architecture cible actuelle de SYCO limite le domaine d'application de ce dernier. Il faut prévoir une double action pour assouplir à la fois l'architecture des parties opératives et celle des parties contrôle. Ces extensions permettront d'étendre l'utilisation de SYCO aux applications nécessitant des circuits rapides, tels que certains circuits de communication.

CHAPITRE V:

CONCLUSION

L'étude des compilateurs de silicium existants présentée au chapitre premier a été placée dans un contexte plus large qui couvre les méthodologies de conception de circuits intégrés. En fait, un compilateur de silicium est un outil permettant d'automatiser une partie ou la totalité du processus de conception dans le cadre d'une méthodologie donnée. Cette étude a permis de dégager des niveaux d'abstraction utilisables pour la compilation de silicium. Il s'agit des niveaux comportemental, architectural, structurel et physique. Le premier permet de décrire un circuit indépendamment de son architecture interne. Le second spécifie l'architecture d'un circuit en termes de sous-systèmes indépendamment de la technologie. Le niveau structurel décrit un circuit en fonction d'une technologie donnée. Le dernier niveau donne tous les détails nécessaires à la fabrication du circuit, il dépend donc du processus de fabrication. Le passage d'un niveau de description donné à un niveau inférieur peut se réaliser à travers trois schémas possibles, selon les outils de compilation utilisés. Ces derniers peuvent être organisés en compilateur, en assistant ou en environnement pour la compilation.

L'organisation du type environnement est la plus générale, elle peut inclure les deux autres. Par exemple, on peut concevoir un environnement qui contient des compilateurs et des assistants spécialisés. Ce genre d'organisation est de plus en plus retenu pour la compilation de structure, ce qui reflète une certaine maîtrise des techniques de compilation de structure. La tendance actuelle est à l'intégration des environnements de compilation de structures et d'architecture. Dans ce cadre, on peut citer l'exemple de la fusion des systèmes GDT [BUR86] et GENESIL [CHE88]. Il est donc normal de s'attendre à ce que les systèmes du futur soient des environnements qui intègrent des outils travaillant à tous les niveaux d'abstraction, y compris le niveau comportemental. Le problème principal à résoudre pour la réalisation d'un tel système est l'assemblage, dans un même circuit, de parties conçues à des niveaux d'abstraction différents. Le terme "assemblage" est pris au sens composition et concerne plusieurs aspects du circuit. Au niveau physique, il faut vérifier que l'assemblage des blocs ne transgresse pas les règles technologiques. Au niveau structurel il faut vérifier la cohérence des blocs assemblés sur le plan logique. Au niveau architectural il faut tenir compte des problèmes de synchronisation.

Les techniques de compilation de comportement présentées au troisième chapitre proviennent en grande partie des domaines de la compilation de logiciel et du compactage de micro-programmes. Elles ont été adaptées à la compilation de comportement pour tenir compte du fait que l'architecture est générée par le compilateur lui même. L'optimisation de l'architecture semble être l'épine dorsale de la compilation de comportement. Le succès des compilateurs de comportement va dépendre des outils d'optimisation associés.

Pour que cette optimisation soit possible, la définition d'un compilateur de comportement doit en tenir compte pour définir le langage de description, l'architecture des circuits générés et les algorithmes de compilation. La définition d'un compilateur de comportement doit aussi tenir

compte des autres étapes de la compilation de silicium, à savoir, la compilation d'architecture et la génération des dessins de masques. Il est difficile d'optimiser certains aspects du circuit tels que la surface ou la consommation en agissant uniquement sur la description du comportement sans tenir compte de la structure ni de la génération des dessins de masques.

Ces différents points ont été pris en compte pour la définition du compilateur de silicium SYCO, présenté au quatrième chapitre. Le compilateur de silicium SYCO part d'une description du comportement et utilise une architecture cible permettant de générer des circuits avec une partie contrôle hiérarchique et une partie opérative parallèle. L'utilisation de modèles prédéfinis pour la synchronisation et pour l'organisation du dessin des masques a permis la définition des outils d'optimisation efficaces.

L'utilisation de modèles prédéfinis a également permis de simplifier les algorithmes de compilation. En effet, l'architecture cible facilite grandement la traduction. Le modèle adopté fournit à la fois des modèles architecturaux et des modèles topologiques. D'autre part l'utilisation de modèles prédéfinis pour l'architecture de tranches de contrôle et celle de la partie opérative réduit la complexité des compilateurs spécialisés. Finalement, l'adoption d'une stratégie d'implantation (modèle topologique) simplifie l'étape de génération des dessins des masques.

L'algorithme de traduction a été encore simplifié par le fait que le choix du sous-ensemble du langage d'entrée a été fait en intégrant les problèmes de la compilation. 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.

Cependant, l'utilisation de tous ces modèles limite le domaine d'application du compilateur SYCO. Une double action, pour assouplir à la fois l'architecture des parties opérative set celle des parties contrôle, est prévue. Le nouveau modèle des parties opératives introduit une souplesse quant au nombre de bus utilisé (fixé à 2 actuellement), et au nombre de cycles nécessaire pour exécuter une opération (fixé à 2 dans la présente version). L'utilisation d'opérateurs externes, définis par l'utilisateur, permettra d'étendre le champ d'application du compilateur SYCO. Pour la partie contrôle, d'autres architectures, plus rapides que celles basées sur les PLAs, doivent être évaluées.

Ces extensions permettront plusieurs choix architecturaux pour la réalisation d'un circuit donné et, par la suite, d'étendre l'utilisation de SYCO aux applications nécessitant des circuits rapides, tels que certains circuits de communication.

L'un des points le plus important de cette thèse est qu'elle s'appuie sur une réalisation d'un compilateur de silicium. Même si le compilateur SYCO n'a pu être à ce jour entièrement terminé, sa

réalisation a permis d'avoir une vue plus claire et plus précise des problèmes posés par la réalisation d'un compilateur de silicium. Il faut rappeler que le compilateur SYCO est le fruit d'un travail collectif auquel plus de vingt personnes ont participé.

En l'état actuel SYCO est encore loin de couvrir tout l'espace des solutions architecturales. L'étude a été limitée à des circuits composés d'une partie opérative et d'une partie contrôle. La prise en compte d'autres types d'architectures, telle que les architecture parallèles [ROB86], nécessite la définition de nouveaux schémas de partition d'un circuit en sous-systèmes.

L'état d'avancement de la compilation de silicium en général fait que l'extension des recherches actuelles ne doit pas se limiter uniquement à l'amélioration des modèles utilisés par les compilateurs ou à la recherche de nouveaux modèles. Mais il faut aussi aborder d'autres aspects de la conception des circuits intégrés tels que la vérification et le test des circuits.

En effet l'un des problèmes cruciaux posé par la conception de circuits intégrés est celui de la vérification. Dans le cas où le circuit est généré automatiquement par un compilateur, on peut envisager la vérification du résultat de la compilation et/ou vérification préalable du compilateur. Bien que plusieurs techniques de vérification permettent de vérifier certaines classes de circuits [MAD88], les recherches actuelles dans le domaine de la vérification de circuit ne permettent pas de répondre à tous les problèmes posés par les grands circuits. Par exemple, peu de résultats ont été obtenus pour la vérification hiérarchique de circuit [CLA89]. Pour ce qui est de la vérification des outils de compilation, peu de recherches ont été menées dans ce domaine. Cette vérification risque d'être difficile à réaliser car non seulement il faut prouver le compilateur, lui-même en tant que programme, mais il faut aussi prouver que les circuits générés par ce compilateur sont corrects. Il faut rappeler que l'ensemble des circuits pouvant être générés par un compilateur donnée est généralement infini.

Il est reconnu qu'il est particulièrement difficile, voire impossible, de tester un grand circuit si rien n'a été prévu au moment de sa conception. Chaque compilateur de silicium doit donc intégrer une stratégie de test. Dans le cas de SYCO, on se dirige vers la génération de circuits à tests intégrés [TOR88]. Une version autotestable en ligne doit également être prévue pour des applications critiques (nécessitant l'utilisation de circuits détectant l'occurrence de défauts au cours de leur fonctionnement). Cette version pourra utiliser les techniques du test unifié pour le test en ligne et le test hors ligne [NIC89]. L'introduction du test nécessitera donc la transformation du modèle architectural pour le rendre autotestable.

BIBLIOGRAPHIE

[AGE76] T. AGERWALA ; Microprogram Optimisation Survey ; IEEE Transaction on Computers;Vol C-25 Octobre 76.

[AHO86] A. AHO & R. SETHI & J. ULLMAN ; Compilers : Principles, Techniques, and Tools; Addison-Wesley 1986.

[ANC80] F. ANCEAU; Architecture and Design of Van Newmann Microprocessor; NATO advanced summer institute, Juillet 1980.

[ANC82] F. ANCEAU; Architecture des ordinateurs: exemple de conception d'un petit microprocesseur; Cours ENSIMAG, Juin 1982.

[ANC83] F. ANCEAU; CAPRI: A Design Methodology and a Silicon Compiler for VLSI Circuits specified by Algorithms; 3rd Caltech Conference on VLSI, Mars 1983.

[ANC84] F. ANCEAU; Silicon compilation for microprocessor-like VLSI; NATO advanced study institute on microarchitecture of VLSI comuters, Urbino, Italie, Juillet 84.

[ANC86] F. ANCEAU; The architecture of microprocessors; Addison-Wesley publication, 1986.

[AND86] J. ANDREWS ; CAD Software users adress database problems ; VLSI systems design, Sep. 1986.

[ARZ84] C. ARZOUNIAN & L. HARIVEL ; Génération automatique de parties opératives ; Rapport de fin d'étude ENSIMAG, IMAG/ TIM3, Juin 84.

[AYR83] R.F. AYRES ; Silicon compilation and the art of automatic microchip design ; Prentice-Hall, 1983.

[BAC78] J. BACKUS ; Can Programming be liberated from the Von Newman Style? A functionnal Style and its algebra of programs ; CACM, Vol 21, n° 8 Juillet 1978.

[BAE88] J.L. BAER al; A Notation For Describing Multiple Views Of VLSI Circuits; Proceedings of the 25th Design Automation Conference, pp. 102-107, juin 1988.

[BAL86] D. BALDWIN ; A Model for Automatic Design Of Digital Circuits ; Technical Report 188, Université de rochester, Juil 86.

[BAR75] M.R. BARBACCI ; A comparison of register tranfers languages ; IEEE Transaction on computers vol C24 N°2, Fevrier 1975.

[BAR81] M.R. BARBACCI ; ISPS: The notation and its Applications ; IEEE Transaction on computers vol C30 N°1, Jan 1981.

[BAT80] J. BATALI & A. HARTHEIMER ; The Design Procedure Language Manual ; Technical Report 598, Massachussets Institute of Technology,

sept. 1980.

[BEK87] N. BEKKARA ; Optimisation et compromis surface/vitesse dans le compilateur SYCO ; Thèse de l'INPG Oct 1987.

[BEL71] C.G. BELL & A. NEWELL ; Computer Structure: Reading and examples ; Ed. Mc Graw Hill 1971.

[BEL87] M.J. BELLOSTA & D. JAILLET & M. MERTENS ; Description Structurelle LDS, Evolution Fondamentale pour LEDS-30 (LDS V4) ; Rapport Interne Bull Système (DEDM), Octobre 1987.

[BER85] N.BERGMAN ; A Case Study of the FIRST silicon compiler ; Dept of computer sciences, U. of Edinburgh, Internal report, Feb 85.

[BER86] BERGE al ; LOF ; ESSCIRC '86.

[BER88] V. BERSTIS ; The V Compiler: Automatic Hardware design ; IEEE Design & Test of Computers, Avril 1989.

[BER88a] R. BERGAMASCHI & D.J. ALLERTON ; A Graph-based silicon compiler for concurrent VLSI Systems; COMPEURO, Avril 1988.

[BHA87] J. BHASKER ; An Algorithm for microcode compaction of VHDL behavioral description ; IEEE VLSI Technical Bulletin, Vol 2, n° 2, Juin 1987.

[BIL88] J.P. BILLON & J.C. MADRE ; Original concept of PRIAM, an industrial tool for efficient formal verification of combinational circuits ; International Working Conference on The Fusion of hardware design and Verification, Glasgow, UK, 1988.

[BLA85] T.BLACKMAN al ; The SILC silicon compiler : language and features ; 22nd DAC, 1985.

[BLA 88] R.L. BLACKBURN & D.E. Thomas & P. Koenig ; CORAL II: Linking Behaviour and Structure in an IC Design System ; 25th DAC ; 1988.

[BON87] Ph. BONDONNO al ; Définition Préléminaire d'un système d'aide à la conception automatique de VLSI ; Rapport Interne IMAG/TIM3 1988.

[BON89] Ph. BONDONNO ; Participation au système Nautile, Thèse de l'INPG, à paraitre 1989.

[BON89a] Ph BONDONNO & A HORNICK & AA JERRAYA & B. COURTOIS ; NAUTILE : Un environnement de conception de circuits en milieu hétérogène ; Papier soumis à TSI.

[BOR88] G. BORRIELLO ; Combining Event and DATA_Flow Graphs in behavioral Synthesis ; ICCAD 1988.

[BOR88a] G. BORRIELLO & E. DETJENS ; High Level synthesis: Current Status and Future Direction ; 25th DAC, 1988

[BOR81] D. BORRIONE ; Langage de description de systèmes logiques : Proposition

pour une méthode formelle de définition ; Thèse d'état, INPG, Juillet 1981.

[BOR88] D. BORRIONE & J.L. PAILLET & L. PIERRE ; Formal Verification of CASCADE Descriptions ; International Working Conference on The Fusion of hardware design and Verification, Glasgow, UK, 1988.

[BRA84] R.K. BRAYTON & C. McMULLEN ; Synthesis and Optimization Of Multistage Logic ; ICCD'84 , pp23-28.

[BRA85] R.K BRAYTON al, Logic minimization algorithms for VLSI synthesis, Kluer academic press Publishers, 1985

[BRA85a] R.K BRAYTON al ; The Yorktown Compiler ; ISKAS, KYOTO, 1985.

[BRA85b] R.K BRAYTON al ; A microprocessor design using the yorktown silicon compiler ; ICCD, 1985.

[BRA85c] R.K. BRAYTON al ; Logic Minimization algorithms For VLSI Synthesis ; Kluwer Academic Publishers, 1985.

[BRA86] R.K. BRAYTON al ; Multiple-Level Logic Optimization System ; ICCAD'86. pp356-359.

[BRA87] R.K. BRAYTON al ; The yorktown silicon compiler ; dans silicon compilation de D.D. GAJSKI, Addison wesley, 87.

[BRE86] F.D. BREWER & D.D. GAJSKI ; An expert System paradigm for design ; 23rd DAC, 1986.

[BRO81] D.W. BROWN ; A State Machine Synthesizer ; DAC'81, pp301-305.

[BRO83] H. BROWN & C. Tong & G. Foyster ; Palladio: An Exploratory Environment for Circuit Design ; IEEE Computer, décembre 1983

[BUC80] I. BUCHANAN: Modelling And Verification In Structured integrated Circuit Design ; PhD Thesis Department of Computer Science,University of Edimburgh, 1980.

[BUR86] M.R. BURICH ; Design of module generators an silicon compilers ; NATO summer course on logic synthesis and silicon compilation for VLSI design, Juillet 86.

[BUR86a] J.L. BURNS & A.R. NEWTON ; Sparcs: A New Constraint-Based IC Symbolic Layout spacer ; IEEE Custom Integrated Circuits Conference, 1986

[BUS86] M.L. BUSHNELL & S.W. DIRECTOR ; VLSI CAD Tool Integration Using the Ulysses Environment ; Proceedings of the 23rd Design Automation Conference, pp. 55-61, juin 1986

[CAD89] CAD For System Design : Is it Practical ; IEEE Design & Test of Computers, Avril 1989.

[CAG85] A. CAGNOLA & M. CORTI & G. VIGNATI ; LAPLACE : Another

Second generation PLA Tool, Microprecessing and microprogramming 16, 1986 .

[CAI87] J.P. CAISSO ; Contribution à la vérification des circuits integrés dans un environnement multivalué ; Thèse de doctorat INPG, Novembre 1987.

[CAM84] R. CAMPOSANO ; Automatic datapath Synthesis from DSL specification ; ICCD'84.

[CAM85] R. CAMPOSANO ; Synthesis techniques for digital systems design ; 22nd DAC, 1985.

[CAM86] R.CAMPOSANO & A. KUNZMANN ; Considering Timing Constraints In Synthesis From A Behavioural Description ; ICCD'86.

[CAM87] R. CAMPOSANO ; Structural synthesis in the YORKTOWN silicon compiler ; VLSI 1987

[CAM87a] R. CAMPOSANO & J.T.J. VAN EIJNDHOVEN ; Combined synthesis of control logic and datapath ; ICCAD 1987.

[CAM88] R. CAMPOSANO & L.F. SAUNDERS & R.M. TABET ; High-Level Synthesis from VHDL ; Rapport de recherche 62789, IBM Research division, T.J. Watson Research Center, Yorktown, 1988.

[CAM88a] R. CAMPOSANO ; Design process model in the YORKTOWN silicon compiler ; 25 DAC, 1988.

[CAS89] A.E. CASAVANT al ; A synthesis environment for Designing DSP Systems; IEEE Design & Test Of Computers, Avril 89.

[CAT87] F. CATTHOOR ; Architectural Design strategies for complex DSP Systems in an automated synthesis environment ; PhD, Université Catholique de louvain, Belgique, 1987.

[CAT89] F. CATTHOOR al ; A testatbility Strategy for multiprocessor architecture ; IEEE Design & Test Of Computers, Avril 89.

[CHA86] H.H. CHAO & S. ONG & K. LEWIS & J.Y. TANG ; Design Automation for Control Circuits Implemented with PLAs ; ICCD'86. pp526-529, 1986.

[CHA88] J. CHAMBON ; STYX, Un système de CAO pour l'édition et la programmation des masques des circuits VLSI ; Thèse de l'ENST, 1988.

[CHE84] E.K. CHENG ; Verifying compiled silicon ; VLSI Design, Oct. 1984.

[CHE87] E.K. CHENG & S. MAZOR ; The Genesil Silicon Compiler ; dans Silicon Compilation, édité par D. Gajski, Addison-Wesley, 1987.

[CHE85]: J. CHERRY al ; NS: An Integrated Symbolic Design System ; VLSI 1985, Elsevier Science Publishers B.V ; IFIP 1986.

[CHU82] S. CHUQUILLANQUI BERNAOLA & T PEREZ SEGOVIA ; PAOLA a tool for topological optimisation of large PLAs ; 19 DAC, 1982.

[CHU84] S. CHUQUILLANQUI BERNAOLA ; Une nouvelle approche pour l'optimisation et l'automatisation du dessin des masques de PLA complexes ; Thèse docteur ingénieur, INPG, 1985.

[CHU85] Y. CHU & Z.S. WANG ; VLSI Design: from CDL to Chip Layouts ; Tech. Rep.-1541, University of Maryland, 1985.

[CLA89] E. CLARKE , Compositional Model Checking, présenté au workshop on automatic verification methods for finite state systems ; Grenoble, 1989.

[COP86] A.J. COPPOLA ; An Implementation of a State Assignement Heuristic ; DAC'86. pp643-647, 1986.

[COX84] G.W. COX ; From macroarchitecture to microarchitecture ; NATO advanced study institute on microarchitecture of VLSI comuters, Urbino, Italie, Juillet 84.

[CRA85] M. CRASTES de PAULET ; Spécification et simulation de circuits complexe: Le système CADOC ; Thèse Docteur Ingénieur, INPG, 1985.

[DAN87] J.D. DANIELL & A.M. DEWEY & S.W. DIRECTOR ; Expanding VLSI Automation Technology ; rapport interne, CMU, 1987.

[DAR81] J.A. DARRINGER al ; Logic synthesis through local transformations ; IBM J. of Res. vol 25 No 4, July 1981.

[DAS87] S. DASGUPTA & U. AGUERO ; On The plausibility of architectural Design ; CHDL 87.

[DAV80] M. DAVIO & A.THAYSE ; Implementation of algorithms based on automata ; Philips J. Res.35, 1980.

[DAV81] S. DAVIDSON ; Some experiment in local micro-code compaction for horizontal machines ; IEEE Transaction on Computers, Vol C-30, N07, Juillet 81.

[DEG86] A.J. DEGEUS & DJ GREGORY ; The SOCRATES logic synthesis and optimisation system ; NATO summer course on logic synthesis and silicon compilation for VLSI design, July 86.

[DEM83] G. De MICHELI & A. SANGIOVANNI-VINCENTELLI ; Computer-aided Synthesis Of PLA-based Finite State Machines ; ICCAD, pp154-156, 1983.

[DEM84] G. De MICHELI & A.S. VINCENTELLI & R. BRAYTON. KISS: A Program For Optimal State Assignement Of Finite state Machines. ICCAD Santa Clara. pp209-211, 1984.

[DEM84a] G. De MICHELI ; Optimal Encoding Of Control Logic ; IEEE, ICCD, New York, pp16-22, 1984.

[DeM86] H. De MAN al ; CATHEDRAL II A synthesis and module generation system for multiprocessor systems on a chip ; NATO summer course on

logic synthesis and silicon compilation for VLSI design, Juillet 86.

[DEM86] G. De MICHELLI ; Synthesis of control systems ; NATO summer course on logic synthesis and silicon compilation for VLSI design, Juillet 86.

[DeM87] H. DeMAN al ; CATHEDRAL II A synthesis and module generation system for multiprocessor systems on a chip ; dans Design Systems for VLSI circuits synthesis and silicon compilation, Ed. NIJHOF 1987.

[DEM88] G. DeMICHELLI & D. KU ; HERCULES : A system for High-level synthesis ; 25th DAC, 1988.

[DEM89] G. DeMICHELLI ; Synchronous Logic Synthesis ; 2nd WorkShop On Logic Synthesis, MCNC, 1989.

[DEN85] P. DENYER & D. RENSHAW ; VLSI Signal processing : a bit-serial approach ; Addison Wesley publication, 1985.

[DEN88] P. DENYER ; The SARI Project ; ICMC, Londre, 1988.

[DeP85] G.F De PALMA ; Architecture experimentation ; VLSI System design, Nov 85.

[DER84] H. DERANTONIAN ; Génération automatique de parties contrôles de microprocesseurs sous formes de PLAs spécialisés ; thèse DI, INPG, 1984.

[DEV88] S. DEVADAS al ; MUSTANG: State Assignement of Finite State Machines Targetting Multilevel Logic implementations ; IEEE Transactions on CAD, vol 7, n°12, DEC 88.

[DIR81] S.W. DIRECTOR al ; A design methodology and computer aids for digital VLSI systems ; IEEE transaction on circuits and systems, VOL C-28, Juil. 81.

[DUR88] Y. DURAND ; Expériences en Synthèses logiques ; Thèse de l'INPG 88.

[EDIF 87] Electronic Design Interface Format, Version 2.0.0 ; Edif Steering Committee, Mai 1987

[EVA85] S.EVANKZUK ; Results of a silicon compiler design challenge ; VLSI design, Juill. 1985

[EVA85a] S. EVANCZUK ; Silicon Compilers: No Automatic route to Acceptance ; VLSIdesign, , Nov. 1985.

[EZZ86] T. EZZEDDINE & J. LASSALE & G. SAGNES, Hierarchical algorithmic description for complex control parts design ; Integrated Circuit Technology Conference, Limerick, IRELAND, 1986.

[FEY87] C.F FEY & D. E. PARASKEVOPOULOS ; Integrated circuits: Current Status And Future trends ; Proceedings Of The IEE 75, No 6, Juin 1987 .

[FEY86] C.F. FEY & D.E. PARASKEVOPOULOS ; A Model of design schedules for ASIC ; CICC, IEEE (ed.), 1986.

[FIS81] J. FISHER ; Trace Scheduling techniques for global micro-code compaction ; IEEE Transaction on Computers, Vol C-30, n°7, Juillet 81.

[FLA84] E. FLAMMAND ; A complete and automatic system for sequencer design ; DAC, 1984.

[FOX85] J. FOX & J.A. FRIED ; Telecommunication circuit design using the SILC Silicon Compiler ; ICCD, IEEE, 1985.

[FRE87] J. FREHEL & J.C. LONGCHAMBON & P. MALARDIER ; A Unique formalism for VLSI Logical and Electrical Synthesis ; Revue de Physique appliquée, Tome 22, n° 1, Janvier 1987.

[FRI69] T.D. FRIEDMAN & SIHCHIN ; Methods Used in an automatic logic design generator (ALERT) ; IEEE Transaction On Computers, C18, n° 7, Juil 1969.

[FUN85] H.S FUNG ; The SILC Silicon Compiler, ICCD ; 1985.

[FUN85a] H.S FUNG & S HIRSCHORN & R KULKARNI ; Design for testability in a silicon compilation environment ; 22nd DAC, 1985.

[FUN85b] H.S FUNG ; Testable-by-construction strategy for the SILC Silicon Compiler ; ICCD, 1985.

[GAJ82] D.D. GAJSKI al ; A second Opignion on Data Flow Machines and languages ; Computer, Fevrier 1982.

[GAJ83] D.D. GAJSKI ; Silicon compilation ; VLSI design, Nov. 1983.

[GAJ83a] D.D. GAJSKI & RH KHUN ; new VLSI Tools ; IEEE Computers, Dec 1983.

[GAJ86] D.D. GAJSKI ; Silicon compilation : a tutorial ; ICCAD, 1986.

[GAJ86a] D.D. GAJSKI & N.D. DUTT & B.M. PANGRLE ; Silicon Compilation (Tutorial) ; IEEE Custom Integrated Circuits Conference, pp102-110, 1986.

[GAJ86b] D.D. GAJSKI & F.D. BREWER ; Towards Intelligent Silicon Compilation; NATO advanced study institute on Logic Synthesis and Silicon Compilation for VLSI Design. Juillet 1986.

[GAJ 87] D.D. GAJSKI ; Silicon compilation ; Addison-Wesley, 1987.

[GLA85] LA GLASSER & DW DOBBERPUHL ; The design and analysis of VLSI circuits ; Addison Wesley 1985.

[GRA85]J. GRANACKI & D. KNAPP & A. PARKER ; Overview, Planner and natural language Interface ; 22nd DAC , 1985.

[GRE85] D. GREEN ; Modern Logic Design ; Electronic systems engineering Series, Addison Wesley, 1985.

[GER85] J.P. GERONIMI & S. RINGO & D. SAVART ; Conception d'un micro-processeur du marché à l'aide d'un compilateur de silicium ; Rapport de

stage d'ingénieur ENSERG, Juin 1985.

[GER87] J.P. GERONIMI & A.A. JERRAYA ; Architecture des parties contrôles générées par SYCO : GENTOPO ; Rapport Interne , IMAG/TIM3 1987.

[GER88] JP GEROMINI ; Génération automatique des parties contrôles ; Rapport interne TIM3, 1988.

[GER88a] J.P. GEROMINI ; Compilation de parties opératives ; DEA informatique, ENSIMAG, 1988.

[GIR84] E.F. GIRCZYC & J.P. KNIGHT ; An ADA to standard cell hardware compiler based on graph grammars and scheduling ; ICCD, 1984.

[GRE86] D. GREGORY & K. BARTLETT & A. De GUES & G. HACHTEL ; SOCRATES : A System For Automatically Synthesising and Optimizing Combinational Logic ; DAC, pp79-85, 1986.

[GRO83] R.GROSS ; Silicon compilers : a critical survey ; Dept. of computer science, Univ. of North Carolina at Chapel Hill, Mai 1983.

[GRA79] J.P. GRAY ; Introduction To Silicon compilation ; DAC, pp305-306, 1979.

[GUY75] A. GUYOT ; Un Compilateur de Microprogrammes ; thèse 3ème cycle USMG/INPG, 1975.

[HAF81] L.J. HAFER & A.C. PARKER ; A Formal method for the specification, analysis and design of RTL digital Logic ; 18th DAC 81.

[HAF82] L.J. HAFER & A.C. PARKER ; Automated Synthesis Of Digital Hardware ; IEEE transactions on computer ; VOL C-31, No 2, pp93-109, February 1892.

[HAN86] S. HANRIAT ; Synthèse Logique a Base De Regles Pour Les Compilateurs De Silicium ; Thèse de L'INPG. Sep 86.

[HEA85] S.T. HEALY & D.D. GAJSKI ; Decomposition of logic networks into silicon ; 22nd DAC, 1985.

[HEI87] S. HEILER & U. DAYAL & J. ORENSTEIN & S RADKE-SPROULL ; why design databases need it ; 24th DAC, 1987.

[HIT83] C.Y HITCHCOCK & DE THOMAS ; a method of automatic datapath synthesis ; 20th DAC, 1983.

[HOR89] A. HORNICK ; Contribution à la définition et à la mise en œuvre du système Nautile ; Thèse de l'INPG, Juin 1989.

[HSU87] D. HSU al ; The Chip Compiler, an automated standard cell/macrocell Physical Design Tool ; CIC Conference P448-491, Juillet 87.

[HUI85] A.HUI al ; A 4.1K gates Double Metal HCMOS Sea Of Gates Array ; CICC 85, pp 15-17

[ISO83] S. ISODA & Y. KOBAYASHI & T. ISHIDA ; generalized data dependancy graph ; Transaction on computers c-32, Oct 1983.

[JAM85] R.JAMIER & A.JERRAYA ; APOLLON : a datapath compiler ; ICCD'85

[JAM85a] R.JAMIER & A.JERRAYA ; APOLLON : a datapath silicon compiler, IEEE circuits and devices magazine, Mai 85

[JAM85b] R. JAMIER & A. JERRAYA & Y. ROBERT ; Using a Silicon compiler for computer algebra ; Computer and computing Informatique et calcul, Edité par P. CHENIN al ; Editions MASSON, 1985.

[JAM86] R.JAMIER & A.JERRAYA & N. BEKKARA ; The automatic synthesis of data processing sections ; ICCD, 1986.

[JAM86a] R. JAMIER ; Génération automatique de parties opératives de circuits VLSI de type microprocesseur ; Thèse docteur ingénieur, INPG Nov. 1986.

[JEN80] J.H. JENKINS & J.A. HOWARDS ; Control overhead : Aperformance metric for evaluating control unit designs ; IEEE transaction on computers, Vol C29, n° 4, Avril 1980.

[JER83] A.A. JERRAYA ; une nouvelle approche pour la vérification des masques de circuits intégrés ; Thèse de Docteur Ingénieur, INPG, 1983.

[JER85] JERRAYA & ROSIER & ROUGEAUX & COURTOIS ; A Hierarchical Symbolic Layout Tool: Styx ; VLSI 1985, Elsevier Science Publishers B.V., IFIP 1986.

[JER85a] A. JERRAYA ; COMFOR: A universal Extractor ; European Solid State Circuit Conference, Toulouse, Octobre 1985.

[JER86] A. JERRAYA & R. JAMIER & P. VARINOT & B. COURTOIS ; The principle of the SYCO silicon compiler ; 23d DAC, Las Vegas July 86.

[JER88] A. JERRAYA & Ph. BONDONO & A. HORNICK & J.P. GERONIMI & B. COURTOIS ; NAUTILE : A physical design Environment based on an object oriented data manager ; IEEE Deisign Automation Workshop, Arizona, 1988

[JER88b] A. JERRAYA & N. MHAYA & J.P. GERONIMI & B. COURTOIS ; SYCO A Silicon compiler for VLSI ASICS specified by Algorithms ; IEE Computer-Aided Engineering Journal, juin 1988.

[JER89] A. JERRAYA & B. COURTOIS ; Architecture experimentation with the behavioral computer SYCO ; MELECON'89, Lisbone, PORTUGAL, 1989.

[JER89a] A. JERRAYA al ; Le compilateur de silicium SYCO ; numéro spécial de TSI sur la CAO de VLSI, à paraitre dernier trimestre 1989 .

[JES88] J.A.G. JESS & R. BORN & L. STOCK ; Synthesis of concurrent Hardware Structure, ISCAS'88.

[JOE86] H. JOEPEN & M. GLESNER ; Optimal Structuring of Hierarchical Control-

Paths in a Silicon Compiler System ; ICCAD pp264-267, 1986.

[JOE85] H. JOEPEN & M. GLESNER ; Architecture construction for general silicon compiler system ; ICCD, 1985.

[JOH79] D.L. JOHANNSEN ; Bristle blocks : a silicon compiler ; 16th DAC, 1979.

[JOH81] D.L. JOHANNSEN ; Silicon Compilation ; PhD, California Institute of Technology, 1981.

[JON86] H.S. JONES Jr & P.R. NAGLE & H.T. NGUYEN ; A comparison of standard cell and gate array implémentations in common CAD system ; CICC, 88.

[JOO86] R. JOOBANI ; WEAVER: A Knowledge-based routing expert ; Kluer press 1986.

[JUL 86] C. JULLIEN & A. LEBLOND & J. LECOURVOISIER ; A Database Interface for an Integrated CAD System ; Proceedings of the 23rd Design Automation Conference, juin 1986.

[KAT82] R.H. KATZ ; A database approach for managing VLSI design data ; 19th DAC, 1982.

[KAT86] R.H. KATZ & M. ANWARRUDIN & E. CHANG ; A Version Server for Computer Aided design Data, Proceedings of the 23rd Design Automation Conference, juin 1986.

[KNA85]: D.W. KNAPP & A. PARKER ; A Unified Representation for Design Information ; Computer Hardware Description Languages and their Applications, Elsevier Science Publishers B.V., IFIP 1985

[KNA86] D.W. KNAPP & A.C. PARKER ; A design utility manager: the ADAM planning engine ; 23rd DAC, 1986.

[KOW84] T.J. KOWALSKI ; The VLSI Design Automation Assistant: A Knowledge-based Expert System ; CMUCAD-84-29, Pittsburg, Avril 1984.

[KOW85] T.J. KOWALSKI & D.E. THOMAS, The VLSI Design Automation Assistant: What's in a knowledge Base ; 22nd Design Automation Conference, 1985.

[KOW85a] T.J. KOWALSKI ; An Artificial Intelligence Approach to VLSI Design ; Kluwer Academic press, Boston, 1985.

[KRE85] D.E. KREKELBERG al ; Yet another silicon compiler ; 22nd DAC, 1985

[KRE86] D.E. KREBELBERG al ; Automated Layout Synthesis in tne YASC Silicon Compiler ; DAC pp447453, 1986.

[KU88] D. KU & G. DeMICHELLI ; Hardware C : A language for hardware Design ; Rapport Technique, CSL-TR-88-362, université de Stanford, Août 1988.

[LAN80] D. LANDSKOV al ; local micro-code compaction techniques ; ACM Comp.

Surveys, V1.12, Sept 80.

[LAR87] T. LARSSON ; Semantics of a Hardware Specification Language and Related Transformation Rules ; North Holland, Integration, The VLSI Journal N°5, 1987.

[LAU85] D.LAURENT & H.NGUYEN ; LDS : langage de description de système ; Rapport interne du projet SYCOMORE, Juin 1985

[LAW86] S. LAW ; A Mixed Media approach to module generator design ; NATO summer course on logic synthesis and silicon compilation for VLSI design, July 86.

[LIP83] H.M. LIPP ; mthodical aspect of logic synthesis ; Proc. IEEE No1, Jan. 83.

[LIP87] J. LIPMANN & D. McMILLAN ; Megacells: Augmenting silicon compilation for ASIC Chip Design ; CICC, 87.

[LIS88] J.S. LIS & D.D. GAJSKI ; Synthesis from VHDL ; ICCAD, 88.

[LUR84] C. LURSINSAP & D.D. GAJSKI, Cell compilation with constraints ; 21ème DAC, 1984.

[MAC89] D. MACQ, Génération automatique d'unité de contrôle à base de ROMs ; Mémoire de stage de fin d'étude, INPG/UC Louvain la neuve, Juin 1989.

[MAD88] J.C. MADRE & J.P. BILLON ; Prouving circuit correctness using formal comparison between expected and extracted behaviour ; 25th DAC, 88.

[MAL85] Y.K. MALAIYA ; Options In Control Implementation ; ICCD, 1985.

[MAR79] P. MARWEDEL ; The MIMOLA design system: detailed description of the software system ; 16th DAC, 1979.

[MAR85] F. MARTINEZ ; Le Compilateur CIRENE ; Mémoire d'ingénieur CNAM. 1985.

[MAR86] S. MARINE ; IRENE: un Langage de description, simulation et synthèse automatique du materiel VLSI ; Thèse de docteur de l'INPG, Fevrier 1986.

[MAR86a] P. MARWEDEL ; A New Synthesis Algorithm For The MIMOLA Software System ; DAC'86, pp271-277.

[MAR87] E.R. MARX ; EDIF: standard Support version control in CAD systems ; CHDL 87, M. Barbacci, Ed. North Holland, 1987.

[MAT85] T.G. MATHESON & C. CHRISTENSEN & M.R. BURIC ; A Software Environment for building core-Microcomputer compiler ; ICCD, IEEE (ed.), NYY, Oct. 1985.

[MAY87] D. MAY & C KEANE ; Compiling OCCAM into Silicon ; Technical note 72 TCH 023 00, INMOS, Fevrier 1987.

[McF83] M.C. McFARLAND ; Computer aided partitionning of behavioral hardware descriptions ; 20th DAC, 1983.

[McF86] M.C. McFARLAND ; Using Bottom Up Design techniques in the synthesis of digital hardware from abstract behavioral descriptions ; 23rd DAC, 1986.

[McF88] M.C. McFARLAND & A. PARKER & R. CAMPOSANO, Tutorial On High-Level Synthesis, 25th DAC, 88.

[MEA80] C.MEAD & L.CONWAY ; Introduction to VLSI systems ; Addison Wesley, 1980.

[MEA83] C.MEAD & L.CONWAY ; Introduction aux systèmes VLSI ; Addison Wesley, 1983.

[MER85] J. MERMET ; Several Steps toward an Integrated Circuits CAD System ; CHDL conférence, 1985.

[MHA86] N. MHAYA ; Compilation de silicium, Compilateur de PC ; Rapport de DEA Informatique , ENSIMAG, 1986.

[MHA87] N.MHAYA & A. JERRAYA ; CPC : The SYCO control section compiler ; ESSIRC, 87.

[MHA88] N.MHAYA & AA JERRAYA ; CPC: a Control Section Synthetizer, ICMC, Londre, 1988.

[MHA88a] N. MHAYA ; Compilateurs de parties contrôle de microprocesseurs ; Thèse de l'INPG, 1988.

[MOO86]: P.P. MOORE ; Oct: Database Programmer's Manual ; University of California Berkeley, janvier 1986

[MOR85] J.D. MORISON & N.E. PEELING & T.L. THORP ; The design Rationale of ELLA a hardware design and description language ; CHDL 85.

[MUL89] R.A. MUELLER & J. VARGHESE ; Retargettable Microcode Synthesis ; ACM Transaction on Computers, Vol 9, n°2, Avril 1987.

[NAK87] Y. NAKAMURA ; An integrated logic design Environment based on behavioral description ; IEEE transactions on computer-aided design, vol CAD-6 No 3, Mai 1987.

[NEW86] A.R. NEWTON ; Behavioral description synthesis ; NATO summer course on logic synthesis and silicon compilation for VLSI design, July 86.

[NEW87] A.R. NEWTON & AL. SANGIAVANNI-VINCENTELLI ; CAD Tools for ASIC Design ; PROCEEDINGS of The IEEE 75, No 6, June 1987.

[NEW87a] A.R. NEWTON ; Symbolic Layout and procedural Design ; dans Design systems for VLSI circuits, logic Synthesis and silicon compilation, edition NIJHOFF 87

[NGU86] H.N. NGUYEN ; Manuel de reference du LDS V0.1 ; Rapport interne Bull Système, Février 86.

[NIC84] M. NICOLAIDIS ; Conception de circuits integrés autotestables pour des

hypothèses de pannes analytiques ; Thèse Docteur Ingénieur, INPG, 1984

[NIC89] M. NICOLAIDIS ; A Unified Built-in Self-Test Scheme: UBIST ; IEEE Trans. on CAD, Vol. 8, n°3, Mars 89.

[NOM85] H. NOMUA ; Current status, Future trends and impact of VLSI ; VLSI'85, Tokyo Japon 1985.

[OBR82] M. OBREBSKA ; Efficiency and performance Comparison of different design methodologies for control parts of Microprocessors ; Microprocessing and Microprogramming, Oct 82.

[ORA86] A. ORAILOGLU & D.D. GAJSKI ; Flow graph representation ; 23rd DAC, 1986.

[OUS84] J. OUSTERHOUT & al ; Magic: A VLSI Layout System ; Proceedings of the 21st Design Automation Conference, pp. 152-159, juin 1984.

[PAP85] C.A. PAPACHRISTOU & D. CORNETT ; Generation And Implementation Of State Machine Controllers: A VLSI Approach ; North Holland, Microprocessing and Microprogramming 16, 1985.

[PAR79] A.C. PARKER al ; An example of automated Data path design ; 16th DAC, 1979.

[PAR81] A.C. PARKER & J.J. WALLACE ; SLIDE: An I/O Hardware Descriptive Language ; IEEE Transaction On Computers, V C30, n° 6 Juin 1981.

[PAR85] N. PARK & A.C. PARKER ; Synthesis of optimal clocking schemes ; 22nd DAC, 1985

[PAR86] A.C. PARKER & Al ; MAHA: A Program For Datapath Synthesis ; 23rd DAC 1986.

[PAR87] A.C. PARKER & S. HAYATI, Automating The VLSI design process using expert System and silicon compilation.

[PAR88] N. PARK & A.C. PARKER ; Shewa: A software Package for Synthesis of Pipelines from behavioral specifications ; IEEE transaction on CAD Vol. 7 No. 3, March 88.

[PAR89] I. PARK ; Compilation de Parties opératives ; DEA de micro-électronique, Grenoble, 1989.

[PAU86] P.G. PAULIN & J.P. KNIGHT & E.F. GIRCZYC ; A multi paradigm approach to Automatic data path Synthesis ; 23rd DAC 1986.

[PAU87] P.G. PAULIN & J.P. KNIGHT ; Force Directed Scheduling in Automatic data path Synthesis ; 24th DAC 1987.

[PEE85] N.E. PEELING, J.D. MORISON ; A Data base approach to design data management and programming support For ELLA, a high level HDL, CHDL 1985.

[PEN86] Z. PENG ; Synthesis Of VLSI Systems With The CAMAD Design Aid ;

DAC, 1986.

[PER85] T.PEREZ SEGOVIA ; PAOLA: Un système d'optimisation topologique de PLA ; Thèse 3ème cycle, INPG, 86.

[RAB86] J.RABAEY al ; CATHEDRAL II: A synthesis system for multiprocessor DSP Systems ; Dans Silicon compilation de DD Gajski, Ed. Addison Wesley, 1986.

[RAJ85] J.V. RAJAN & D.E. THOMAS ; Synthesis By Delayed Binding Of Decisions ; DAC'85, pp367-373.

[REI86] R. REIS & A. JERRAYA & R. JAMIER ; Design of the syco 6502 using the SYCO silicon compiler, 1986.

[RIS72] E.M. RISEMAN & C. AXTON & C. FOSTER ; The inhibition of Potential parallélism by conditionnal jump ; IEEE Transaction on computer Dec. 1972.

[RNL 87] R.W. NOTROTT & H. KOEMAN ; NETLIST & RNL tutorial for beginners ; University of Washington, Northwest LIS Release 3.1, février 1987

[ROB86] Y. ROBERT ; Algorithmes parallèles: reseaux d'automates, architectures systoliques, machines SIMD et MIMD ; Thèse d'etat, INPG, 1986.

[ROS85] W. ROSENSTIEL & R. CAMPOSANO ; Synthetising circuits from behavioral level specification ; CHDL 1985.

[ROH88] R.A. ROHER ; Evolution Of The Electronic Design Automation Industry ; IEEE Design & TEST and Computers, Dec 88.

[ROU87]: F.R. ROUGEAUX ; Outils de CAO Et Conception Structurée De Systèmes Intégrés Sur Silicium ; Thèse de Docteur de l'INPG, Février 1987.

[RUD85] R. RUDELL ; ESPRESSO & Berkeley CAD Tools User's Manuel ; Report No. UCB/CSD 86/272. Dec 1985.

[RUD86] R. RUDELL & A. SANGIOVANNI-VINCENTELLI ; Exact Minimization of Multiple-Valued Functions for PLA Optimization, ICCAD, 1986.

[RUD87] R.RUDELL & A. SANGIOVANNI-VINCENTELLI ; Multiple-Valued Minimization for PLA Optimization ; IEEE Tran. on CAD, Vol. CAD-6, No. 5, Sept. 1987.

[SAB86] D.G. SABO & D. JOHANSON & R. YAU ; GENESIL Silicon compilation and design for testability ; CICC, 1986.

[SAK86] H. SAKUMA; A silicon pre-compiler ; Internal report CSR-207-86 Department of computer science, University of Edinburgh.

[SAN87] A.L. SANGIOVANNI VINCENTELLI ; Synthesis of LSI Circuits ; dans Design systems for VLSI circuits, logic Synthesis and silicon compilation, edition NIJHOFF 87

[SAS85] T. SASAO ; HART: A Hardware for Logic Minimization and Verification ;
ICCD 1985.

[SAU86] G. SAUCIER & S. HANRIAT ; Rule Based Logical Synthesis For Silicon Compilers ; ICCAD, 1986.

[SAU87] G. SAUCIER & M.C. PAULET & P. SICARD ; ASYL: A Rule-Based System For Controller Synthesis ; IEEE trans on CAD. Vol. CAD-6, No.6, Nov 1987.

[SCH83]: J.P. SCHOELLKOPF ; Lubrick, a Silicon Assembler and its Application to Data-path Design for FISC ; VLSI'83, 1983.

[SCH85] J.P. SCHOELLKOPF ; SILICIEL : Contribution à l'architecture des circuits intégrés et à la compilation de silicium ; Thèse d'etat, I.N.P. Grenoble, 1985.

[SEQ83] C.H. SEQUIN ; Managing VLSI Complexity: An outlook ; Proceeding of the IEEE, V 71, n° 11, Janvier 83.

[SHI83] S.G.SHIVA ; Automatic Hardware Synthesis ; Proc. IEEE Vol 71, No 1, Jan. 1983.

[SHR85]H.E. SHROBE ; The Data Path Generator ; conference on advanced research in VLSI, MIT, Janv. 1985.

[SIS82] J.M.SISKIND al ; Generating custom high performance VLSI designs from succinct algorithmic descriptions ; Conference on advanced Res. in VLSI, MIT, 1982

[SIX86] P. SIX al ; An intelligent module générator environment ; 23rd DAC 1986.

[SNO78] F.A. SNOW & D.P. SIEWIOREK & D.E. THOMAS ; A Technology-relative Computer-aided Design System: Abstract Representations, Transformations, And Design Tradeoffs ; DAC, 1978.

[SOL86] J.A. SOLWORTH ; GENERIC: A Silicon Compiler Support Language ; DAC'86.

[SOU83] J.R. SOUTHARD ; MacPitts : an approach to silicon compilation ; Computer, proc IEEE, Dec 84.

[SOU87] J.R. SOUTHARD ; Algorithmic system compilation ; dans Silicon compilation, De D.D. Gajski, Ed. Addison Wesley, 1987.

[SPR 80] R.F. SPROULL & R.F. LYON ; The Caltech Intermediate Form For LSI Layout Description ; in C. Mead & L. Conway, Introduction to VLSI Systems, Addison Wesley, 1980, pp 115-127

[STA81] J.A. STANKOVIC ; The types and interactions of vertical migration of functions in a multilevel interpretive system ; IEEE Transaction on Computers, Vol C30, n° 7 , Juil 81.

[STA88] V. STAVRIDOU & H BARRINGER & D.A. EDWARDS ; Formal spécification and vérification of hardware: a comparative case Study ; 25th DAC, 1988.

[STR88] C.E. STROUD & R.R. MUNOZ & D.A. PIERCE ; Behavioural model synthesis with CONES ; Design & TEST of Computers, JUIN 1988.

[SYC84] Annexe technique du projet SYCOMORE ; THOMSON & BULL & INRIA & INPG, 1984.

[TAK86] R. TAKAHASHI & T. YOSHIMURA & S. GOTO ; A VLSI Architecture evaluation system ; ICCD, Oct. 86.

[TER86] H. TERRY ; A transaction model for object-oriented VLSI CAD ; Dep. of computer sciences, Univ. of Texas at Austin, 1986.

[THO81] D.E THOMAS ; The automatic synthesis of digital system ; Proc IEEE, vol 69, No 10, Oct. 1981

[THO83] D.E THOMAS al ; Automatic data path synthesis ; Computer, Decembre 1983.

[THO83a] D.E THOMAS al ; Methods for automatic data path synthesis ; RR No CMUCAD-8313, 1983.

[THO87] D.E. THOMAS al ; Linking the behavioral and structural domains of représentation for digital system design ; IEEE transaction on CAD, Vol CAD-6, n°1, Jannuary 1987.

[THO 88] D.E THOMAS al ; The System Architect' Workbench ; 25th DAC 88.

[TLE87] H TLEMCANI ; Optimisation pour la compilation de silicium ; Rapport de stage de fin d'étude, TIM3, 87.

[TOK81] M. TOKORO al ; Optimisation of Microprograms ; IEEE Transaction on Computers, Vol C-30, N07, Juillet 81.

[TOM67] R.M. TOMASULO ; An efficient algorithm for exploiting multiple arithmetic units ; IBM Journal, Janvier 1967.

[TOR88] K. TORKI & M. NICOLAIDIS & A.A. JERRAYA & B. COURTOIS ; UBIST Version of the SYCO's Control Section Compiler ; ICCD, Octobre 1988.

[TOR88a] J. TORRELLAS al ; Introductory User's Guide to the Architect's Workbench Tools ; CSL-TR-88-355, Université de Stanford, May 1988.

[TRE87] N. TREDENNICK ; Microprocessor Logic Design ; Addison Wesley 1987.

[TRE87a] L. TREVILLIAN ; An Overview of logic Synthesis systems ; 24th DAC, 1987

[TRI83] S.M. TRIMBERGER ; Automated performance optimisation of custom integrated circuits ; PhD, California Institute Of Technology, 1983.

[TRI 84] S.M. TRIMBERGER ; VTIcompose, A Powerful Graphical Chip Assembly

Tool ; Proceedings of the 21st Design Automation Conference, pp. 697-698, juin 1984.

[TRI85] H. TRICKEY ; Compiling Pascal programs into Silicon ; Phd, U. Stanford, 85.

[TRI86] S. TRIMBERGER & F. PROLI ; A Microprocessor datapath synthesized with a translator from scematics to silicon ; VLSI Design, Avril 1986.

[TSE83] C. TSENG & D.P. SIEWIOREK ; FACET: A Procedure For The Automated Synthesis Of Digital Systems ; DAC, 1983.

[TSE84] C. TSENG & D.P. SIEWIOREK ; EMERALD: A Bus Style Designer. DAC'84, pp315-321.

[TSE88] C. TSENG al ; Bridge a versatile behavioral synthesis system, 25th DAC, 1988.

[TSU86] Y. TSUCHITA al ; Establishment of Higher Level Logic Design for Very Large Scale Computer. DAC'86. pp366-371.

[TUC85] B.W. TUCKER ; Electronic CAD/CAM is it evolution or revolution ; 22nd DAC 1985.

[ULL84] J.D.ULLMAN ; Computational aspect of VLSI, Computer science press ; 1984

[VAN88] N. VANDER ZANDEN & D.D. GAJSKI ; MILO: A Microarchitecture and Logic Optimizer ; 25th DAC 88.

[VAR87] P. VARINOT ; Compilation des parties contrôles ; Thèse de l'INPG 87.

[VHD87] VHDL Tutorial for IEEE Standard ; 1076 VHDL. Mai 1987

[WAL83] R.A. WALKER & D.E. THOMAS ; Behavioral level transformation in the CMU-DA system ; 20th DAC , 1983.

[WAL85] R.A. WALKER & D.E. THOMAS ; A model for Design representation and synthesis ; 22ème DAC 85.

[WAL87] R.A. WALKER & D.E. THOMAS, Design Representation and transformation in the System Architect's Workbench ; CMUCAD-87-34, August 1987.

[WAL88] R.A. WALKER & D.E. THOMAS ; The Architect's Workbench ; 25th DAC, 1988.

[WEI88] R.S.WEI & S. ROTHWEILER & J.Y. JOU ; BECOME: behavior level circuit Synthesis Based on structure mapping ; 25th DAC, 1988.

[WES85] N. WESTE & K ESHRAGHIAN ; Principle of CMOS VLSI design ; Addison wesley, 1985.

[WIL84] J.A. WILMORE ; artwork editing and verification system ; ICCAD, 1984.

[WOL87] W. WOLF ; Symbolic layout and compaction for silicon compilation ; NATO summer course on logic synthesis and silicon compilation for

VLSI design, July 86.

[YOS86] T. YOSHIMURA & S. GOTO ; A Rule-Based Algorithmic Approach for Logic Synthesis ; ICCAD'86. pp162-165.

[ZAR85] J. ZARRELLA ; How silicon compilation will affect engineers ; COMPCON spring, 1985.

[ZIM79] G. ZIMMERMANN ; The MIMOLA Design system: A computer aided digital processor design method ; 16th DAC, 1979.

[ZYZ88] E. ZYSMAN ; Conception de parties contrôles de circuits VLSI, Application au coprocesseur FELIN ; Thèse de doctorat INPG, 1988.

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 5 de l'arrêté du 16 avril 1974
VU l'Arrêté du 23 novembre 1988
VU les rapports de

- Monsieur B. COURTOIS
- Monsieur F. JUTAND
- Monsieur C. LANDRAULT

Monsieur JERRAYA Ahmed Amine

est autorisé à présenter une thèse en soutenance pour l'obtention du grade de **DOCTEUR D'ETAT ES-SCIENCES. (" Informatique ")**

Fait à Grenoble, le 4 décembre 1989

Le Président
de l'Université Joseph Fourier

13 DEC. 1989

Le Président

A. NEMOZ

Pour le Président de l'I.N.P.G.
et par délégation,
le Vice-Président
P. VENNEREAU