Expériences en synthèse logique

Yves Durand

THESE

présentée par

Yves Durand

pour obtenir le titre de **DOCTEUR**

de **L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**
(arrêté ministériel du 5 Juillet 1984)

(spécialité Informatique)

Expériences en synthèse logique

Thèse soutenue le 21 Octobre 1988 devant la commission d'examen:

G. MAZARE	Président
G. CAMBON	Rapporteur
F. JUTTAND	Rapporteur
J. MERMET	Examinateur
A. DEBREIL	Examinateur

Thèse préparée au sein du laboratoire ARTEMIS

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	Herve	UFR PGP
CHOVET	Alain	ENSERG
COHEN	Joseph	ENSERG
COUMES	André	ENSERG
DARVE	Félix	ENSHMG
DELLA-DORA	Jean-François	ENSIMAG
DEPORTES	Jacques	ENSPG
DESRE	Pierre	ENSEEG
DOLMAZON	Jean-Marc	ENSERG
DURAND	Francis	ENSEEG
DURAND	Jean-Louis	ENSIEG
FOGGIA	Albert	ENSIEG
FONLUPT	Jean	ENSIMAG
FOULARD	Claude	ENSIEG
GANDINI	Alessandro	UFR PGP
GAUBERT	Claude	ENSPG
GENTIL	Pierre	ENSERG
GREVEN	Hélène	IUFA
GUERIN	Bernard	ENSERG
GUYOT	Pierre	ENSEEG
IVANES	Marcel	ENSIEG
JAUSSAUD	Pierre	ENSIEG
JOUBERT	Jean-Claude	ENSPG
JOURDAIN	Geneviève	ENSIEG
LACOUME	Jean-Louis	ENSIEG
LESIEUR	Marcel	ENSHMG
LESPINARD	Georges	ENSHMG
LONGEQUEUE	Jean-Pierre	ENSPG
LOUCHET	François	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
SERMET	Pierre	ENSERG
SILVY	Jacques	UFR PGP
SIRYES	Pierre	ENSHMG
SOHM	Jean-Claude	ENSEEG
SOLER	Jean-Louis	ENSIMAG
SOUQUET	Jean-Louis	ENSEEG
TROMPETTE	Philippe	ENSHMG
VEILLON	Gérard	ENSIMAG
ZADWORNY	François	ENSERG

Personnes ayant obtenu le diplôme

D'HABILITATION A DIRIGER DES RECHERCHES

BECKER	Monique
BINDER	Zdeneck
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	LANDAU	Ioan
FRUCHART	Robert	VACHAUD	Georges
HOPFINGER	Emile	VERJUS	Jean-Pierre
JORRAND	Philippe		

Directeurs de recherche 2ème Classe

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

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

ENSEEG

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

ENSERG

BOREL	Joseph

ENSIEG

DESCHIZEAUX	Pierre	PERARD	Jacques
GLANGEAUD	François	REINISCH	Raymond

ENSHMG

ROWE	Alain

ENSIMAG

COURTIN	Jacques

EFP

CHARUEL	Robert

C.E.N.G

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

Laboratoires extérieurs :

C.N.E.T

DEVINE	Rodericq	MERCKEL	Gérard
GERBER	Roland	PAULEAU	Yves

****

****

***

UNIVERSITE SCIENTIFIQUE TECHNOLOGIQUE ET MEDICALE
DE GRENOBLE

Président de l'Université :

M. PAYAN Jean Jacques

ANNEE UNIVERSITAIRE 1987 - 1988

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ERE CLASSE

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 I.S.N.
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
BOEHLER Jean Paul	Mécanique
BONNIER Jane Marie	Chimie Générale
BOUCHEZ Robert	Physique Nucléaire I.S.N.
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
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
FONTAINE Jean Marc	Mathématiques Pures
GAGNAIRE Didier	Chimie Physique
GERMAIN Jean Pierre	Mécanique
GIRAUD Pierre	Géologie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean René	Mathématiques Pure

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.
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du Solide
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PAYAN Jean Jacques	Mathématiques Pures
PEBAY PEYROULA Jean Claude	Physique
PERRIER Guy	Géophysique
PIERRARD Jean Marie	Mécanique
PIERRE Jean Louis	Chimie Organique
RENARD Michel	Thermodynamique
RINAUDO Marguerite	Chimie C.E.R.M.A.V.
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
VALENTIN Jacques	Physique Nucléaire I.S.N.
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS DE 2EME CLASSE

ADIBA Michel	Mathématiques Pures
ANTOINE Pierre	Géologie
ARMAND Gilbert	Géographie
BARET Paul	Chimie
BLANCHI Jean Pierre	S.T.A.P.S.
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BRUANDET Jean François	Physique
BRUGAL Gérard	Biologie
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
COURT Jean	Chimie
DUFRESNOY Alain	Mathématiques Pures
GASPARD François	Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GONZALEZ SPRINBERG Gérardo	Mathématiques Pures
GUIGO Maryse	Géographie
GUMUCHIAN Hervé	Géographie
GUITTON Jacques	Chimie
HACQUES Gérard	Mathématiques Appliquées
HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
JARDON Pierre	Chimie
JOSELEAU Jean Paul	Biochimie
KERCKHOVE Claude	Géologie
LONGEQUEUE Nicole	Sciences Nucléaires I.S.N
LUCAS Robert	Physique
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
NEMOZ Alain	Thermodynamique C.N.R.S. C.R.T.B.T.
OUDET Bruno	Mathématiques Appliquées
PECHER Arnaud	Géologie
PELMONT Jean	Biochimie
PERRIN Claude	Sciences Nucléaires I.S.N.
PFISTER Jean Claude	Physique du Solide
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
RICHARD Jean Marc	Physique
RIEDTMANN Christine	Mathématiques Pures
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
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L'I.U.T. 1

PROFESSEURS DE 1ERE CLASSE

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

PROFESSEURS DE 2EME CLASSE

BOUTHINON Michel	E.E.A. I.U.T. 1
CHAMBON René	Génie Mécanique I.U.T. 1.
CHEHIKIAN Alain	E.E.A. I.U.T. 1
CHENAVAS Jean	Physique I.U.T. 1
CHOUTEAU Gérard	Physique I.U.T. 1
CONTE René	Physique I.U.T. 1
GOSSE Jean Pierre	E.E.A. I.U.T. 1
GROS Yves	Physique I.U.T. 1
KUHN Gérard, détaché	Physique I.U.T. 1
MAZUER Jean	Physique I.U.T. 1
MICHOULIER Jean	Physique I.U.T. 1
MONLLOR Christian	E.E.A. I.U.T. 1
PEFFEN René	Métallurgie I.U.T. 1
PERRAUD Robert	Chimie I.U.T. 1
PIERRE Gérard	Chimie I.U.T. 1
TERRIEZ Jean Michel	Génie Mécanique I.U.T. 1
TOUZAIN Philippe	Chimie I.U.T. 1
VINCENDON Marc	Chimie I.U.T. 1

PROFESSEURS 2EME 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/Cardiovasculaire	C.H.R.G.
CORDONNIER Daniel	Nephrologie	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/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	Faculté la Merci
JUNIEN-LAVILLAUROY Claude	O.R.L.	C.H.R.G.
KOLODIE Lucien	Hématologie Biologique	C.H.R.G.
LETOUBLON Christian	Chirugie Générale	C.H.R.G.
MACHECOURT Jacques	Cardiologie et Maladies Vasculaires	C.H.R.G.
MAGNIN Robert	Hygiène	C.H.R.G.
MASSOT Christan	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.

PROFESSEURS DE PHARMACIE

AGNIUS DELORD Claudine	Physique	Faculté la Tr
ALARY Josette	Chimie Analytique	Faculté la Tr
BERIEL Hélène	Physiologie et Pharmacologie	Faculté la Tr
CUSSAC Max	Chimie Therapeutique	Faculté la Tr
DEMENGE Pierre	Pharmacodynamie	Faculté la Tr
FAVIER Alain	Biochimie	C.H.R.G.
JEANNIN Charles	Pharmacie Galenique	Faculté Meyla
LATURAZE Jean	Biochimie	Faculté la Tr
LUU DUC Cuong	Chimie Générale	Faculté la Tr
MARIOTTE Anne-Marie	Pharmacognosie	Faculté la Tr
MARZIN Daniel	Toxicologie	Faculté Meyla
RENAUDET Jacqueline	Bactériologie	Faculté la Tr
ROCHAT Jacques	Hygiène et Hydrologie	Faculté la Tr
SEIGLE MURANDI Françoise	Botanique et Cryptogamie	Faculté Meyla
VERAIN Alice (Mme Rapin)	Pharmacie galénique	Faculté Meyla

MEMBRES DU CORPS ENSEIGNANT DE MEDECINE

PROFESSEURS CLASSE EXCEPTIONNELLE ET 1ERE CLASSE

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatre - Puériculture	C.H.R.G.
BEZES Henri	Orthopédie - Traumatologie	Hopital Sud
BONNET Jean-Louis	Ophtalmologie	C.H.R.G.
BOUCHET Yves	Anatomie	Faculté la Mer
	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 Mer
HOLLARD Daniel	Hématologie	C.H.R.G.
LATREILLE René	Chirurgie Thoracique/Cardiovasculaire	C.H.R.G.
LE NOC Pierre	Bactériologie - Virologie	Faculté la Mer
MALINAS Yves	Gynécologie et Obstétrique	C.H.R.G.
MALLION Jean Michel	Médecine du Travail	C.H.R.G.
MICOUD Max	Clinique Médicale/Maladies Infectieuses	C.H.R.G.
MOURIQUAND Claude	Histologie	Faculté la Mer
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	Anesthésiologie	C.H.R.G.
TANCHE Maurice	Physiologie	Faculté la Mer
VIGNAIS Pierre	Biochimie	Faculté la Mer

Le lecteur attentif (et il faut l'être, pour lire même les remerciements) pourra discerner, derrière la relative diversité des expériences présentées, le cheminement troublé d'un thésard. Mes remerciements iront tout d'abord à Jean Mermet, qui est à l'origine de ce travail, et qui en a assuré l'encadrement et la nécessaire continuité. Je remercie également M. Jean-François Trichard, chef du département D.E.D.M. à BULL-SYSTEMES, pour la confiance qu'il m'a témoigné en m'acceptant dans son département, et M. Alain Debreil, chef du service "micrologiciel et vérification logique" pour son encadrement patient, et sa participation au jury.

Je voudrais également remercier M. Guy Mazaré, professeur à l'ENSIMAG, qui a accepté de présider ce jury, et MM. Gaston Cambon, professeur au LAMM, et Francis Juttand, professeur à l'ENST, qui ont assumé la tâche de rapporteur.

Je tiens également à remercier Jacky Guillon, François Bertrand et Yvon Gressus qui ont été les principales victimes de mes "interviews" de concepteurs. Au chapitre des bénévoles honteusement exploités, je remercie également mes lecteurs Dominique Borrione, Pierre Fontanille, Annick Montanvert et Patrice Uvietta.

Je ne pourrais pas citer ici tout ceux qui m'ont aidé, au laboratoire ARTEMIS et dans les autres laboratoires de l'IMAG. Qu'ils reconnaissent ici l'expression de ma gratitude...

Enfin, comme la recherche en informatique est quelquefois semée d'embûches matérielles, je rends grâce à Artemis, Athena et surtout Osiris d'avoir prété oreille à mes incantations, offrandes et compilations, et d'avoir souvent amené mes programmes au terme de leurs exécutions.

RÉSUMÉ

Le problème de la synthèse automatique de circuits est abordé ici à travers trois expériences de réalisation de compilateurs. La première concerne la traduction de la spécification fonctionnelle d'un circuit décrit en LASCAR (langage de la famille CASCADE). Le deuxième compilateur utilise un formalisme de règles de réécriture pour produire un circuit adapté à une bibliothèque spécifique de BULL-SYSTEMES à partir d'une description en langage LDS. La troisième expérience aborde le problème de la synthèse de parties opératives, dont les principales difficultés sont présentées en détail. La méthode utilisée met en œuvre un algorithme de séquencement fondé sur un formalisme potentiel-tâche, et une méthode de partage de registres et d'allocation d'opérateurs à partir d'un algorithme de coloriage de graphe.

MOTS-CLEFS

CAO de VLSI, synthèse automatique de matériel, compilation de silicium.

Chapitre 1

INTRODUCTION

1-INTRODUCTION

Le credo de la compilation de silicium ne manque jamais d'évoquer le temps où l'on faisait des circuits en dessinant "à la main" toutes les couches du circuit que l'on était en train de réaliser. Heureusement, maintenant[1], dans la panoplie des outils de CAO, parmi les simulateurs, extracteurs, routeurs, vérificateurs, etc., on trouve le compilateur de silicium qui dessine un circuit automatiquement à partir d'une description intelligible de son comportement. Le travail présenté ici a pour but de préciser cette image idyllique en décrivant des expériences de réalisation de compilateurs de silicium.

Nous ne présenterons pas toutes les formes de compilation de silicium, mais essentiellement la synthèse logique à partir de langages de description de haut niveau. La réalisation de deux systèmes est décrite en détail. On aborde également la synthèse de parties opératives de type micro-processeur, et la synthèse de parties contrôle.

La compilation de silicium est un sujet vaste, et beaucoup de littérature est déjà parue à ce sujet. Le **chapitre 2 (préliminaires)** est une tentative de classification des différents types de compilateurs que l'on rencontre. Cette délicate entreprise est compliquée par un problème de vocabulaire: les auteurs utilisent des vocables comme "fonctionnel", "langage de haut niveau", etc..., dans des sens différents.

Les trois chapitres suivants se limitent à un des types de compilation, la synthèse logique. Le **chapitre 3 (la synthèse logique)** spécifie le problème et décrit de manière générale les méthodes employées.

Le **chapitre 4 (synthèse logique à partir de LASCAR: COMPLO)** décrit en détail la première expérience de réalisation d'un compilateur descendant partant au niveau "Transfert de registres", utilisant un langage source non procédural , LASCAR, pour effectuer une synthèse de type "fonctionnel". Cette synthèse réalise de manière simple à la fois la synthèse booléenne, la synthèse de la partie opérative et la synthèse de la partie contrôle. Le résultat de cette compilation est une liste de modules et d'interconnexions sans architecture cible. Le masque du circuit est obtenu en utilisant un outil de placement/routage à partir de la "liste d'interconnexions".

---

[1] sous-entendu: avec la complexité croissante des VLSI, etc...

Le **chapitre 5 (synthèse logique à partir de LDS)** décrit un compilateur similaire qui utilise un langage source procédural (LDS). Cet outil effectue également une synthèse booléenne et une synthèse de partie opérative. Le résultat de la compilation est une liste d'interconnexions utilisant les éléments d'une bibliothèque modifiable. L'architecture cible est représentée par une "liste de règles" acquises auprès des concepteurs. Cette expérience est également complétée par l'utilisation d'assembleurs de silicium.

Dans ces deux chapitres, on trouve une description complète des méthodes utilisées pour la compilation. Si dans les grandes lignes les deux compilateurs se ressemblent, surtout au niveau de l'allocation, ils diffèrent sur beaucoup de points de détail. Nous avons donc préféré les décrire indépendamment.

Nous complétons ce travail sur la synthèse logique par deux études: la première est décrite dans le **chapitre 6 (synthèse de partie opérative)** et concerne la synthèse de partie opérative à partir d'un langage procédural. Cette étude assez générale est illustrée par une méthode de synthèse du langage LDS. Ce travail est en fait antérieur au compilateur logique du langage LDS présenté au chapitre 4.

Le **chapitre 7 (synthèse de partie contrôle)** récapitule brièvement les différentes façons de modéliser et de synthétiser les parties contrôles des circuits séquentiels.

Après la **conclusion** et la **bibliographie** traditionnelles, le lecteur trouvera dans les **annexes** les compléments plus formels aux différents chapitres.

Chapitre 2

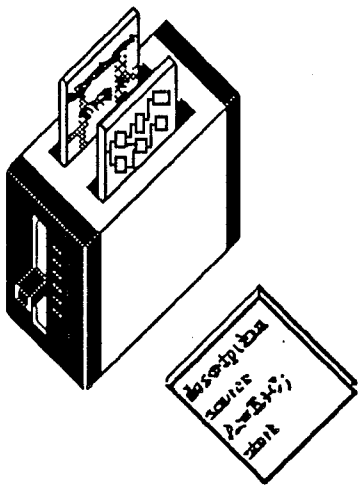
PRELIMINAIRES

2-PRÉLIMINAIRES

2.1. motivations

Un compilateur de silicium est un programme de transformation qui produit un circuit prêt à implanter à partir d'une description intelligible de son comportement. Le bénéfice attendu est essentiellement un gain de temps : on parle quelquefois de semaines pour la réalisation de circuits type micro-processeur. De plus, ce gain de temps peut amener un gain en qualité, puisqu'il devient possible d'explorer rapidement différentes possibilités au cours du développement.

L'utilisation d'un compilateur de silicium permet en outre de réaliser des ensembles qui auraient été trop complexes à réaliser à la main. C'est le cas par exemple de la plupart des compilateurs de structures régulières.



le compilateur de silicium idéal

Le compilateur est parfois défini comme un support et une sauvegarde du savoir-faire des concepteurs, dans la ligne des "systèmes expert". L'aspect sauvegarde n'est pas négligeable dans le milieu des circuits intégrés: il permet au équipes de conception de conserver un certain savoir-faire même quand l'ingénieur est parti.

En définitive, ce qu'on attend d'un compilateur de silicium est un compromis entre qualité du résultat, temps de réalisation et souplesse d'utilisation et de mise à jour. Les importances relatives de ces critères

varient suivant le type de compilation étudié. Le seul critère vraiment obligatoire est la "justesse par construction" du résultat.

2.2. terminologie des niveaux de modélisation

On peut imaginer beaucoup de choses différentes, sous "description intelligible". Nous nous limitons ici aux descriptions textuelles. On parlera de langage de description de matériel. Un langage de description de matériel est caractérisé par plusieurs critères : le premier est le niveau d'abstraction auquel il se place, qu'on appelle également niveau de modélisation. On adoptera ici la terminologie de [Bor 81]:

Architecture système : le système est décrit comme un ensemble de processus communiquants au sens de Hoare.

Architecture matérielle : on suppose que les ressources matérielles et leurs fonctions sont spécifiées.

Instructions : le jeu d'instruction de la machine est décrit.

Transferts de registres : le comportement de la machine est décrit comme une série de transferts de données de registre à registre.

Circuit séquentiel : le système est décrit comme une interconnexion entre éléments logiques au sens large, c'est-à-dire en incluant les éléments de mémorisation...

Circuit combinatoire : le système est décrit comme un réseau de portes logiques, du type ET, OU, NON, NON-ET ...

Circuit électrique : le circuit est décrit comme un assemblage de composants électroniques du type transistor, résistance, source de courant, de tension, ...

Masques : le système est "décrit" par ses masques. On ne peut plus vraiment parler d'"abstraction", même s'il s'agit encore d'une modélisation de la réalité physique du circuit.

La conception de circuit est trop complexe pour pouvoir être réduite à une série d'étapes aussi simples. Comme toutes les définitions, celles-ci sont très discutables, mais nous nous baserons sur ce vocabulaire.

2.3. synthèse remontante et synthèse descendante

Il y a deux démarches possibles pour passer de la "description intelligible" à la bande de génération de masques.

La première, dite démarche "descendante"[1], consiste à décomposer systématiquement le problème en sous-problèmes plus simples à résoudre. Le processus passe par les étapes intermédiaires typiques des concepteurs : conception des algorithmes, définition des fonctions, décomposition en modules puis en portes, implantation symbolique et dessin des masques. Le compilateur est alors un rassemblement d'outils correspondants à ces étapes : compilateur RTL[2], compilateur logique, assembleur de silicium, compacteur d'implantation. On trouve des exemples de ce type de compilateurs dans les principaux laboratoires de recherche industriels : IBM, ATT, NTT, ...

La deuxième démarche, dite "ascendante"[3], consiste à assembler des modules de matériel à mesure que les spécifications se précisent. Le compilateur comprend un certain nombre de générateurs paramétrables d'implantation, qui produisent des blocs particuliers : PLA, portes, UAL, etc... Cette démarche est celle prise par les compilateurs de silicium commercialisés (voir paragraphe suivant).

Ces deux types d'outils, descendant et ascendant, ont des champs d'application et des performances différents. Les compilateurs de type ascendant peuvent arriver à de bons résultats, comparables à ceux des concepteurs, dans le cas où le circuit utilise un nombre restreint de fonctions et d'opérateurs, et qu'il se prête à une implantation régulière. Les outils de type "descendant" n'ont pas un domaine d'application restreint de cette manière. Ce sont généralement des ensembles de traducteurs-optimiseurs entre différents niveaux de modélisation.

Typiquement, le système comprendra :

---

[1] quelquefois appellée "synthèse par décomposition"

[2] Register Transfer Level. Nous utiliserons cet acronyme pour "Transfert de registre".

[3] On trouve aussi "synthèse par raffinement", mais nous n'emploierons pas ces termes qui peuvent être facilement ambigüs.

- Un synthétiseur RTL qui extrait une description en blocs matériels à partir d'un algorithme.

- Un compilateur logique qui convertit ces blocs ou ces fonctions spécifiques en circuits logiques booléens ("net list").

-Un "synthétiseur électrique" qui construit un réseau de transistors à partir du schéma logique.

- Un implanteur qui dessine les masques ou qui assiste leur dessin.

Il est clair que ces définitions sont très simplificatrices. On ne peut pas entrer plus en détail dans la conception descendante sans prendre en compte les questions de langage de description. Nous abordons la question dans les trois chapitres suivant. Nous présentons d'abord les compilateurs commerciaux, qui constituent un exemple typique de compilateur ascendant.

2.4. les compilateurs du commerce

On ne peut guère parler de compilation de silicium sans parler des outils commercialisés. Mais ce commentaire ne peut être que superficiel. Comme la littérature qui paraît à ce sujet a une claire vocation commerciale, on ne peut pas trop lui faire confiance. La seule solution pour une étude en profondeur serait de les essayer tous!

le marché

Le terme "compilateur de silicium" est employé généralement soit pour un générateur de modules, soit pour la collection d'outils qui le contient (ou qui les contient). L'idée est apparue avec Bristle blocks [Joh 79] , mais le premier système commercialisé fut *Mac Pitts* ([Sis 82] actuellement disparu du marché). L'idée a été rapidement reprise et actuellement, quelques vendeurs de CAO et la plupart des fondeurs d'ASIC proposent leur(s) compilateur(s).

La différence entre ces outils et les outils universitaires équivalents est claire : elle apparaît dès que l'on s'assied devant l'écran. Les compilateurs commerciaux intègrent vraiment dans une interface utilisateur presque exclusivement graphique, la génération de tests, les différents simulateurs, avec vérification des règles de dessin à tous les étages...

Les compilateurs de silicium soulèvent une délicate question qui n'est pas encore résolue : à qui revient-il de les utiliser ? aux fondeurs ? aux clients ?

Pour les fondeurs, l'outil présente certains avantages : tout d'abord, il permet d'accepter des spécifications de plus haut niveau, ou des conceptions incomplètes. Par exemple, Gould ajoute des filtres aux circuits de ses clients en utilisant un outil automatique. D'autre part, un compilateur peut regénérer rapidement une bibliothèque, surtout dans les technologies proches du micron, où il ne suffit pas de réduire l'échelle de référence ("lambda reduction").

L'intérêt pour le client est clair également : s'il utilise des compilateurs de silicium, il pourra faire évoluer son produit, il pourra changer facilement de technologie, etc... En deux mots, il ne sera plus aussi dépendant de son fondeur. Mais la médaille a son revers : certains fondeurs ne facilitent pas l'utilisation des compilateurs (autres que les leurs) qu'ils accusent de briser la "relation sacrée" (sic) entre le fondeur et son client. Concrètement, cela peut se traduire par une moins bonne garantie des

performances, en particulier sur les caractéristiques du processus de fonderie.

En fait la question fait rapidement intervenir des arguments techniques : un outil indépendant du fondeur peut-il être optimal ? On est obligé de rentrer dans le détail des différentes technologies pour répondre. Mais les arguments commerciaux ne restent pas longtemps absents du débat.

applications et limitations

Chez VTI on distingue au moins 45 champs d'application différents. Sans aller jusque-là, nous en énumérons quelques-uns :

RAM (presque tous), ROM (presque tous), ALU (VTI,...), additionneurs(Macgen de LSI logic), multiplieurs (Macgen de LSI logic), multiplieurs-accumulateurs (Macgen de LSI logic), PLA (presque tous), SLA (Cirrus logic)

On trouve quelques générateurs de circuits analogiques :

amplificateurs opérationnels : (Concord de SST), circuits d'acquisition de données (Concord de SST)

...mais aussi des choses plus complexes :

chemins de données type 2901 (VTI), microprocesseur type 8051

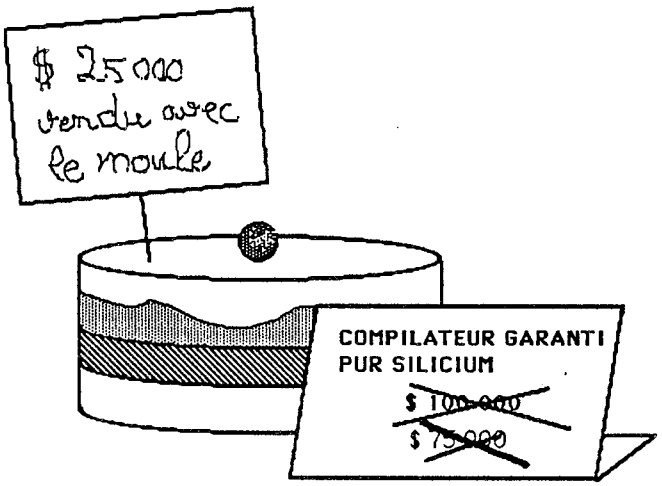
la synthèse logique

La synthèse logique (ici la conversion de comportemental en structurel puis en masques) fait une entrée timide dans les compilateurs de silicium. Genesil peut fabriquer des masques à partir d'équations booléennes, ainsi que "Logic compiler" de VTI. "Finesse" de Seattle Silicon et "Future designer" de Futurenet se réclament également de la synthèse logique. Gould propose un "compilateur fonctionnel" (functional compiler). VTI permet également la synthèse d'automates. Cet outil ("State machine Compiler", maintenant intégré à "Chipcompiler") permet de réaliser facilement de petits automates. Un essai a permis de réaliser une version de l'automate de "mult" (voir chapitres 4 et 5) en deux heures. Le langage d'entrée est plus restreint que ceux que nous utilisons, ce qui fait que la comparaison n'est pas possible avec nos résultats. Ce genre d'outil est surtout intéressant pour la synthèse de petits automates. Dans ce cas, la différence de performances par rapport à la conception manuelle ne se fait

pas sentir. De plus, le "state machine compiler" utilise certains algorithmes éprouvés de "ESPRESSO" pour optimiser ses résultats.

Metasyn (de Metalogic, aujourd'hui disparu du marché) a été le seul à proposer une synthèse complète à partir d'un langage de haut niveau : malheureusement, son architecture cible était très restreinte (voir chapitre 6 § 6) et sa technologie obsolète.

la "seconde génération"



Bien qu'il soit la "tarte à la crème" de l'électronique actuelle, le compilateur de silicium est peu utilisé, et peu vendu : malgré leurs titres ronflants, les articles adoptent un ton désabusé à ce sujet. On assiste en réaction à un re-ciblage des outils. Puisque les outils destinés aux concepteurs néophytes ne se vendent pas, les vendeurs proposent des outils plus puissants aux concepteurs expérimentés. Prenons en exemple l'évolution Genesil / Genesis. Genesil est défini comme un "ensemble de systèmes experts paramétrés pour l'analyse et la synthèse de circuits". Pour faire un circuit avec Genesil, il faut éditer une série de fiches décrivant les blocs que l'on veut compiler, puis interconnecter ces blocs, considérés comme des boîtes noires. Avec Genesis, on dispose en plus d'une série de langages spécialisés pour écrire ses propres compilateurs de blocs.

en bref

Malgré les proclamations des documentations commerciales, il est clair maintenant que l'outil qui transforme le concepteur de système en électronicien expérimenté n'existe pas encore. Le compilateur de silicium

est destiné surtout aux experts. Puisque c'est ainsi, les vendeurs leurs proposent des outils raffinés et puissants, utilisant des entrées textuelles complexes pour produire des outils personnalisés. Mais les experts résistent. Les vendeurs attribuent cette réticence à la traditionnelle méfiance des concepteurs de circuits devant les langages, et font confiance au temps. Certains comparent l'évolution actuelle à la période qui a précédé l'acceptation des compilateurs de langages de programmation tel FORTRAN ([And 88]).

2.5. le langage source

La manière dont on spécifie le circuit en entrée de l'outil détermine en grande partie le traitement ultérieur : le langage d'entrée fixe à la fois le niveau de modélisation et le style de la description. Le niveau de modélisation peut être très abstrait (algorithme, jeu d'instructions), ou de bas niveau (équations booléennes) en passant par diverses nuances (transferts de registres, équations fonctionnelles). Le style de la description est également important : le travail du compilateur n'est pas le même suivant que l'entrée est algorithmique, structurelle ou fonctionnelle. Outre les caractéristiques intrinsèques du langage, le style de la description englobe la manière dont les concepteurs l'utilisent. Puisque le compilateur de silicium est une sémantique du langage, il faut qu'il corresponde à l'usage de son langage source.

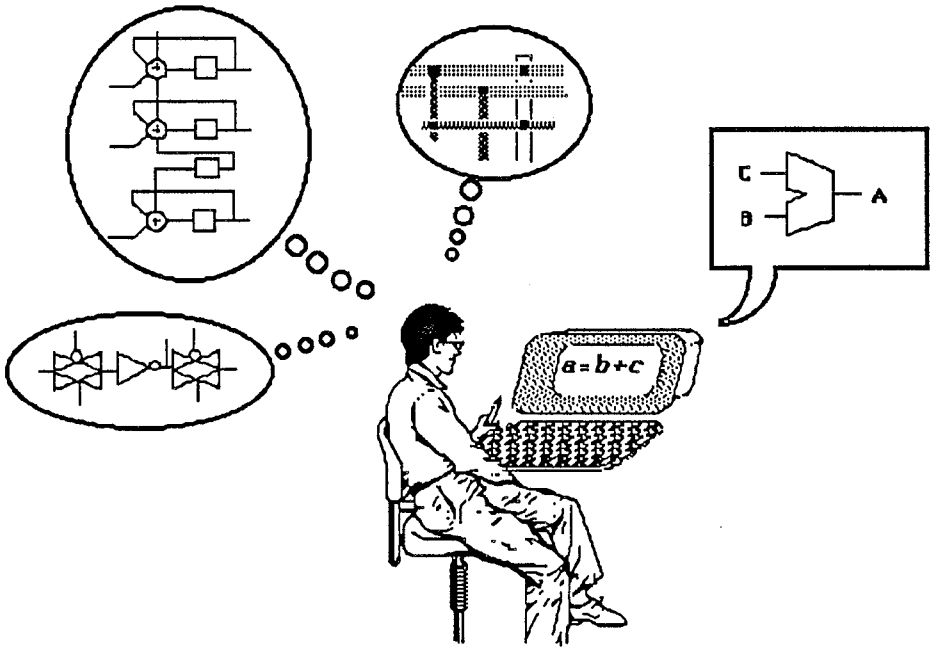


figure 2 : le problème des niveaux de modélisation

Les chapitres 4 et 5 présentent des systèmes de synthèse logique basés sur des langages syntaxiquement très proches. Mais leurs différences de sémantiques entraînent des différences importantes dans les algorithmes.

modélisation fonctionnelle et modélisation algorithmique

Quand on décrit un circuit de manière algorithmique, un bloc de la description modélise une partie de l'algorithme et non un bloc physique du circuit. Les blocs de la description correspondent plutôt à des instructions ou à des procédures du type "fetch", "add", etc... . Au contraire, dans le cas d'une modélisation fonctionnelle, chaque bloc de la description correspond à un bloc physique du circuit. Les appellations "algorithmiques" et "fonctionnelles" sont employées ici dans le sens de [Jer 86][1] En d'autres termes, la modélisation algorithmique décrit **ce que fait** le circuit, alors que la modélisation fonctionnelle précise **comment.**

Ces deux types de synthèse ont des langages sources différents : une modélisation algorithmique se fera naturellement avec un langage procédural. Par contre, le matériel étant par nature parallèle, la

---

[1] On trouve dans [Dud 83] une classification utilisant des termes voisins, mais dans un sens différent. Nos notations se transposent en "modélisation imperative" et "modélisation fonctionnelle" dans [Camp 86], dans la catégorie "langages fonctionnels" de [Gasj 86], sans parler de la nomenclature VHDL...

modélisation fonctionnelle s'exprimera de manière déclarative dans un langage non procédural.

Les deux synthèses posent des problèmes très différents, comme on le verra dans les chapitres suivants. La synthèse algorithmique est généralement considérée comme la plus difficile. Certains auteurs l'appellent "synthèse de haut niveau" pour l'opposer à la synthèse fonctionnelle.

2.6. synthèse combinatoire, synthèse de partie opérative et synthèse de partie contrôle

Qu'ils soient ascendants ou descendants, les outils peuvent également être classés selon le type de circuits auquel ils sont destinés : **les outils de synthèse combinatoire** effectuent la synthèse logique d'un ensemble de fonctions booléennes. Le principal problème est ici un problème d'optimisation multi-critères (place, longueur des chaînes, "routabilité") qui dépend fortement de la technologie cible (logique sauvage, "standard cells", PLA, réseaux prédiffusés...).

Les outils de synthèse de partie opérative partent de la description algorithmique des fonctionnalités du circuit, et élaborent un schéma de la partie opérative selon certains modèles architecturaux (voir Chapitre 6) La synthèse est effectuée en tenant compte des différents parallélismes entre actions, des ressources disponibles sur le circuit.

Enfin, **la synthèse de partie contrôle** consiste à générer un contrôleur à partir soit d'un jeu d'instructions et d'une partie opérative, soit à partir du graphe de contrôle du circuit (voir Chapitre 8).

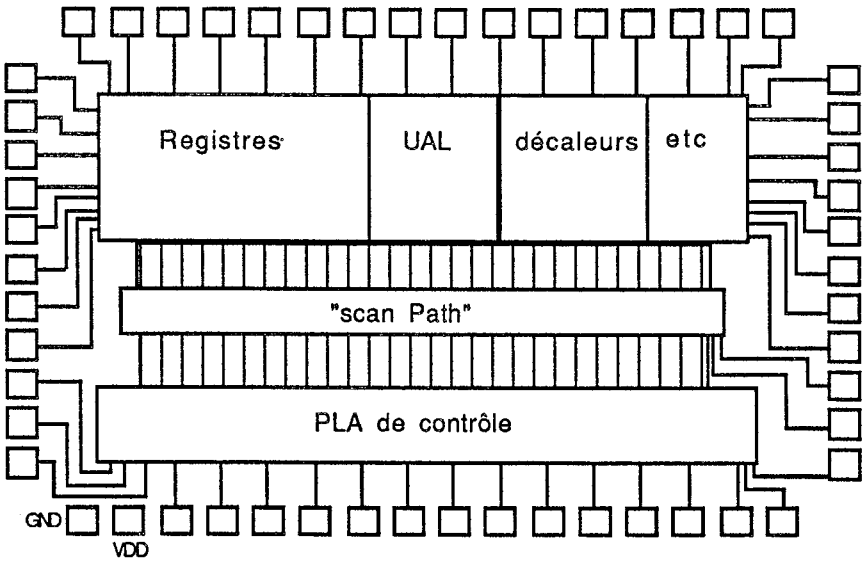
Pour éviter de multiplier les "classes" d'outils de synthèse, nous ne considèrerons dans la suite que les 3 types suivants :

Type	description	langage
synthèse combinatoire	Fonctionnelle	Non Procédural
synthèse de partie opérative	Fonctionnelle ou Algorithmique	Procédural
synthèse de partie contrôle	Algorithmique	Procédural[1]

[1] "procédural" s'applique ici à la modélisation d'un jeu d'instruction. Lorsqu'on décrit un système par son graphe de contrôle, la sémantique ne se réduit pas à des définitions comme "procédural" ou "non procédural".

2.7. l'architecture cible

L'"architecture cible" définit une "structure d'accueil" pour le circuit synthétisé, c'est-à-dire : son organisation en blocs pour le contrôle, ses registres de données, ses opérateurs, l'organisation de ses horloges, les interfaces de communication entre les différentes parties du circuit. A titre d'exemple, nous reproduisons ci-dessous l'architecture cible d'un des premiers compilateurs du genre: "Bristle blocks" [Joh 79].



architecture cible du compilateur "Bristle Blocks"

Le problème de l'architecture cible se pose surtout quand le compilateur part d'une description algorithmique. Dans ce cas le compilateur doit allouer lui-même les ressources aux opérations. Ceci est rendu plus facile si les ressources et les chemins de communication sont fixés a priori. De plus, la définition d'une architecture cible est le seul moyen d'obtenir automatiquement une implantation correcte, en résolvant une grande partie des problèmes de placement-routage.

Nous ne parlerons pas des systèmes de synthèse dédiés à une application très précise. Citons par exemple la synthèse d'architectures systoliques à partir d'un système d'équations récurrentes décrite dans [Gach 87]. Nous

nous limitons à la synthèse de circuits modélisés au moyen d'un langage source de haut niveau.

2.8. l'évaluation d'un compilateur de silicium

L'évaluation des performances d'un compilateur de silicium est une question délicate. Les critères qui entrent en ligne de compte sont ceux qui caractérisent la qualité du circuit obtenu : surface, vitesse et consommation. Nous distinguons les compilateurs de circuits combinatoires et les compilateurs dédiés à une architecture précise.

Dans le cas des compilateurs de circuits combinatoires, le problème semble simple : on est tenté de les comparer en donnant en entrée un jeu d'équations et de mesurer le nombre de portes ou la surface de la sortie. Mesurer le nombre de portes obtenues ne donne aucune indication : il dépend beaucoup du jeu de cellules, et des limitations d'entrance et de sortance, si le compilateur les prend en compte. De plus, rien ne prouve qu'il sera possible de router la liste de composants obtenue. Mesurer un compilateur d'après la surface des circuits obtenus est également très discutable, puisqu'ainsi on mesure surtout les performances du placement-routage des cellules. Certains auteurs effectuent des comparaisons en terme de "portes équivalentes" ([Par 79]). Par exemple ALERT refaisait un IBM 1800 avec 160% de portes supplémentaires par rapport au modèle originel. Quand le compilateur est dédié à une architecture cible précise, le problème se complique puisqu'on est obligé de comparer les compilateurs sur des exemples qui leur conviennent. En général, on trouve des comparaisons avec la conception manuelle originale du circuit qui sert de référence. On trouve ainsi différentes versions de PDP8 ([Dir 81] avec 30% de surface en moins, [Hua 83] avec 30% de portes équivalentes en plus).

Dans une récente réunion[1], un certain nombre d'universitaires, frustrés par cette incapacité à comparer leurs résultats, se sont mis d'accord sur une procédure d'évaluation des outils de synthèse, procédure basée sur des exemples communs, une bibliothèque de cellules communes, une définition des critères, etc...

Enfin, il faut distinguer la qualité du circuit à la sortie de la compilation, et les résultats que l'on peut obtenir en utilisant des

---

[1] "3rd High Level Synthesis Workshop", Orcas Island, Janvier 1988. Les principaux résultats sont reportés dans [Bor 88].

compilateurs pour explorer différentes alternatives de conception. Dans [Row 87], on rapporte la conception particulièrement réussie d'un circuit musical ("E-Chip" de E-mu Systems), mais le développement a coûté huit homme-mois.

Les vendeurs promettent en général un écart de 10 à 20 % par rapport à la conception manuelle. En fait, la surface du bloc compilé n'est qu'un des aspects du problème. Il faut considérer également l'interconnexion du nouveau bloc avec les autres blocs, générés automatiquement ou non. D'autres questions se posent : peut-on faire confiance à un circuit conçu automatiquement ?

2.9. prouver la correction des résultats

La nécessité de coupler étroitement les outils de compilation de silicium et les outils de preuve est claire. Une des nombreuses raisons est que l'on ne peut guère envisager d'utiliser un compilateur de silicium sans retoucher le résultat, ne serait-ce que pour l'intégrer aux parties dessinées à la main. Ceci implique qu'on ne sera jamais sûr à cent pour cent du résultat, et qu'il faudra donc recourir à des logiciels de preuve de matériel.

Le problème est de savoir si le circuit que l'on obtient par la compilation d'une spécification est correct ou non. Il soulève avec lui beaucoup de questions liées à la preuve de circuit : quels sont les critères pour estimer un circuit correct ? est-il possible de vérifier la correction ? En fait, dans le domaine de la compilation de silicium, peu d'auteurs s'en soucient... [Lipp 83]. Malheureusement, les outils de preuve ne sont pas encore assez au point pour pouvoir prouver n'importe quel circuit. On peut envisager d'autres méthodes pour prouver la correction des circuits obtenus par compilation. Ce qui paraît le plus simple est de prouver le compilateur. Une expérience est décrite dans [Mil 83], mais elle porte sur une compilation limitée aux portes NOR. Une autre méthode pourrait être de garder en cours de compilation certaines informations susceptibles de faciliter la preuve du circuit compilé.

Chapitre 3

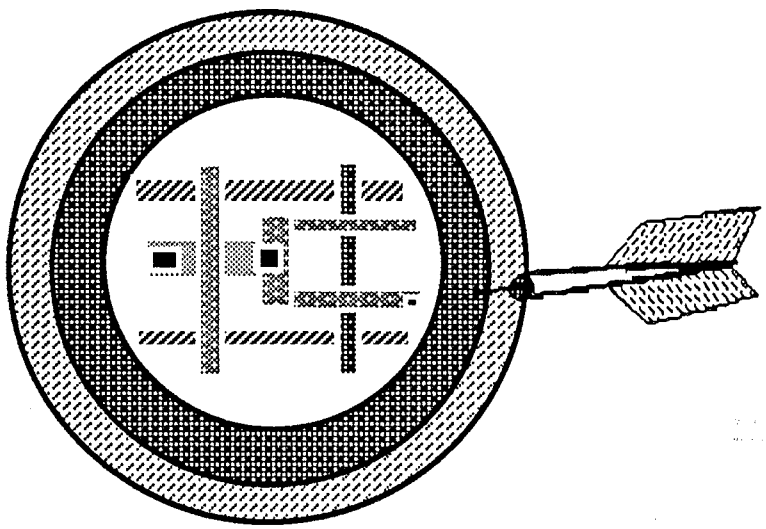
LA SYNTHESE LOGIQUE

3-LA SYNTHESE LOGIQUE

3.1. définition

Nous amalgamerons ici "synthèse logique" et "synthèse combinatoire" pour ne considérer que la première : ce type de synthèse est le plus répandu et consiste à traduire les fonctions décrites dans la description source en un circuit composé de cellules simples. Le point de départ de ce type de synthèse est une description au niveau "transferts de registres" (RTL) où les registres et les principaux points d'interconnexion sont spécifiés. La synthèse consiste alors à implémenter au mieux les fonctionnalités de la description en utilisant les composants d'une bibliothèque définie. Au mieux, c'est-à-dire en utilisant le moins possible de composants et en minimisant les longueurs de chaînes. D'autres contraintes interviennent également, comme les limitations en entrance et en sortance des éléments, les problèmes de testabilité ...

3.2. technologie cible



technologie cible (allégorie)

La synthèse logique est une synthèse très descendante : on augmente le circuit en lui ajoutant incrémentalement le matériel correspondant à chacune des fonctions décrites. En conséquence, on parlera de technologie cible plutôt que d'architecture cible pour la synthèse logique.

L'implémentation d'un circuit combinatoire peut se faire de différentes manières : PAL, PLA, réseaux prédiffusés, logique aléatoire ... Si les principes généraux sont à peu de choses près les mêmes pour la synthèse automatique dans ces technologies, les optimisations sont elles très différentes. Nous ne parlerons pas de la technologie "full custom" puisque, par nature, la synthèse automatique lui convient mal.

Dans la technologie "gate array", le circuit est implanté en personnalisant une grille de portes non connectées. Dans ce cas, la synthèse est contrainte par le type unique des portes (NAND, NOR,...) et doit également prendre en compte la routabilité de l'ensemble.

En conception "standard Cell" ou en "Poly-Cell", on utilise un nombre moins restreint de cellules de base. Le problème qui se pose est de bien distribuer les fonctionnalités sur les cellules disponibles.

Enfin, la conception "macro-cell" relie en quelque sorte la conception "custom" et la conception "standard cell". Les macro-cellules sont des cellules obtenues algorithmiquement en fournissant des paramètres à des générateurs de modules. Ce sont souvent des implémentations efficaces, car très régulières. Cette démarche est complémentaire de la synthèse logique.

3.3. méthode générale

Quand on parcourt la littérature qui a trait à la synthèse logique, on trouve certains points communs à tous les systèmes : tout d'abord un schéma général commun, la séquence allocation/ optimisation/ composition ([Par 84]).

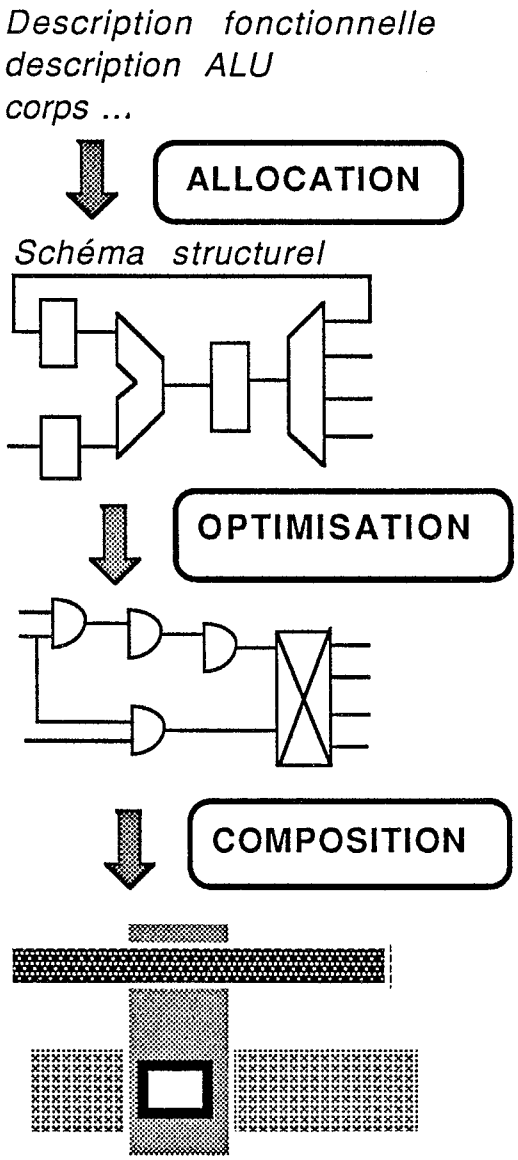


Schéma général pour la synthèse de matériel

L'allocation est la mise en correspondance des caractéristiques abstraites du circuit : opérations, opérandes... et des ressources matérielles : registres, unités fonctionnelles. En effet une description comportementale comprenant une addition n'explicite pas le type

d'additionneur utilisé, ni le nombre... ce qui la fait qualifier d'"ambiguë" par Marine [Mari 86]. Cette phase d'allocation est souvent programmée algorithmiquement ("implémentation naïve" chez LSS de IBM, COMPLO, etc...), parfois avec des règles (ASYL, DDL/SX de Fujitsu, le système de synthèse de NEC, etc...).

L'optimisation[1] est la transformation du résultat de l'allocation. Elle a pour but de minimiser ou de maximiser certains paramètres tout en préservant la fonctionnalité du circuit. Les méthodes employées sont très diverses et dépendent de la technologie cible et du type d'application de l'outil de synthèse (cf § "l'optimisation").

La composition est l'implantation du circuit sur le cristal (ou sur le circuit imprimé) en composants réels : c'est le gros problème du placement/routage, et il déborde largement du cadre de la synthèse logique. En fait la philosophie de la synthèse logique est de l'ignorer (à la différence des autres synthèses, qui, ayant une "architecture cible", l'abordent dès le début).

---

[1] le terme "optimisation" est ici un abus de langage puisque le résultat n'est jamais optimal. On devrait utiliser "amélioration". Mais "optimisation" est plus proche du terme anglais "optimization" utilisé habituellement.

3.4. La synthèse logique est une compilation classique...

Le terme souvent utilisé de "compilation" est tout à fait justifié quand il s'agit de synthèse logique, à tel point que J. Ullman s'y est essayé ([Flo 82]) : les langages RTL ressemblent la plupart du temps fortement aux langages de programmation comme Ada, Pascal ou Lisp... L'affectation de signaux ou de registres remplace l'affectation de variables, la modularité est essentielle, on trouve des structures de contrôle de type "if-then-else"... Même les délicats problèmes de parallélisme sont abordés de manière similaire. Ceci explique que l'on retrouve en synthèse logique les techniques de la compilation des langages de programmation : méthodes d'optimisation de la génération de code de Aho et Ullmann dans LSS ([Aho 77], [Tre 86]), allocation de registres par coloration de graphes ([Chai 82]) utilisé dans CMU-DA ([Chia 83]), techniques "data-flow" comme par exemple l'analyse des temps de vie...(voir chapitre 8). La phase d'optimisation du code se retrouve dans notre phase d'optimisation. Le compromis entre taille du code et rapidité d'exécution est transposé en compromis entre taille du circuit et rapidité des signaux.

3.5. l'allocation

Nous distinguerons l'allocation "algorithmique" qui est un processus relativement simple et proche de l'analyse syntaxique, des autres méthodes. Les autres méthodes recouvrent des techniques d'intelligence artificielle, des méthodes de programmation entière, etc...

Comme il est dit plus haut, la phase d'allocation transforme la description source en une description structurelle représentant un circuit non optimal, au fonctionnement équivalent.

L'allocation algorithmique est un processus de compilation assez classique. Le principal problème est de donner un équivalent matériel à chaque notion du langage source. Les questions qui se posent en général sont les suivantes : comment traduire une expression ? une instruction? une instruction conditionnée? si le langage exprime un séquencement, comment le traduire? que faire des variables dimensionnées? comment traduire les constantes? Les chapitres 4 et 5 décrivent deux méthodes proches de manière assez complète.

3.6. l'optimisation

Le problème de l'optimisation se pose toujours : comme les critères d' "optimalité" varient énormément d'un circuit à un autre, il n'est pas possible de définir une méthode d'allocation qui donne de bons résultats de manière générale. En conséquence, la plupart des systèmes de synthèse comportent deux phases. Le circuit qui a été tout d'abord créé par l'allocation doit être optimisé avant d'être implanté. Cette séparation du processus permet en outre d'utiliser la phase d'optimisation de manière indépendante (l'optimisation booléenne existait avant la synthèse logique). Enfin, la phase gagne à être isolée: ceci permet l'intervention du concepteur qui peut explorer différentes possibilités en modifiant les paramètres.

Nous divisons les programmes d'optimisation en trois catégories : les simplifications booléennes, les transformations locales et les méthodes dérivées des techniques d'intelligence artificielle. Comme toujours, cette classification est simplificatrice : l'optimisation à base de transformations locales utilise beaucoup de règles de simplifications booléennes, les systèmes de transformation locale à base de règles se réclament souvent de l'intelligence artificielle...

les méthodes de simplification booléenne

Les méthodes de simplification booléenne sont des méthodes globales, par opposition aux méthodes de transformations locales. Elles ont été étudiées avant la synthèse logique. La raison en est simple : les circuits des années 60 étaient très coûteux et essentiellement combinatoires. On peut citer la méthode de factorisation en NAND et en NOR de D. Dietmeyer qui date de 1963 ([Die 63]). En 1965, Quine et Mc Cluskey publient une technique automatisable. Dans ses grandes lignes, la méthode est simple, et toujours d'actualité: tous les impliquants premiers de la fonction sont tout d'abord calculés, ensuite on identifie un sous-ensemble de ces impliquants pour définir une couverture de la fonction. Tison utilise le consensus pour la génération des monômes premiers et l'extraction d'une couverture irrédondante pour un ensemble de fonctions. En 1971 Biswas introduit la notion de degré d'adjacence qui permet de déterminer les implicants essentiels de la fonction en évitant les coûteuses manipulations de tables ([Bis 84]). Il existe actuellement un certain nombre de programmes heuristiques fondés sur ces principes : MINI [Hon 74], PRESTO [Bro 81], ESPRESSO [Bra 85], Mc Boole [Dag 85], PHIPLA [Laa 85].

Comme nous avons préféré les méthodes par transformations locales aux méthodes globales de simplification booléennes, nous ne détaillerons pas plus ce sujet.

optimisation par transformations locales

Une transformation locale est la transformation d'une seule fonction indépendamment du reste du modèle. Plus de problème ni de forme canonique, ni de monômes premiers ... On renonce à l'optimalité pour favoriser la souplesse d'utilisation. John Darringer, qui l'introduit pour son système de synthèse LSS, qualifie la méthode de "nouveau regard sur la synthèse logique" ("A new look at logic synthesis" [Dar 80]).

Les transformations locales s'appliquent à différents formalismes :

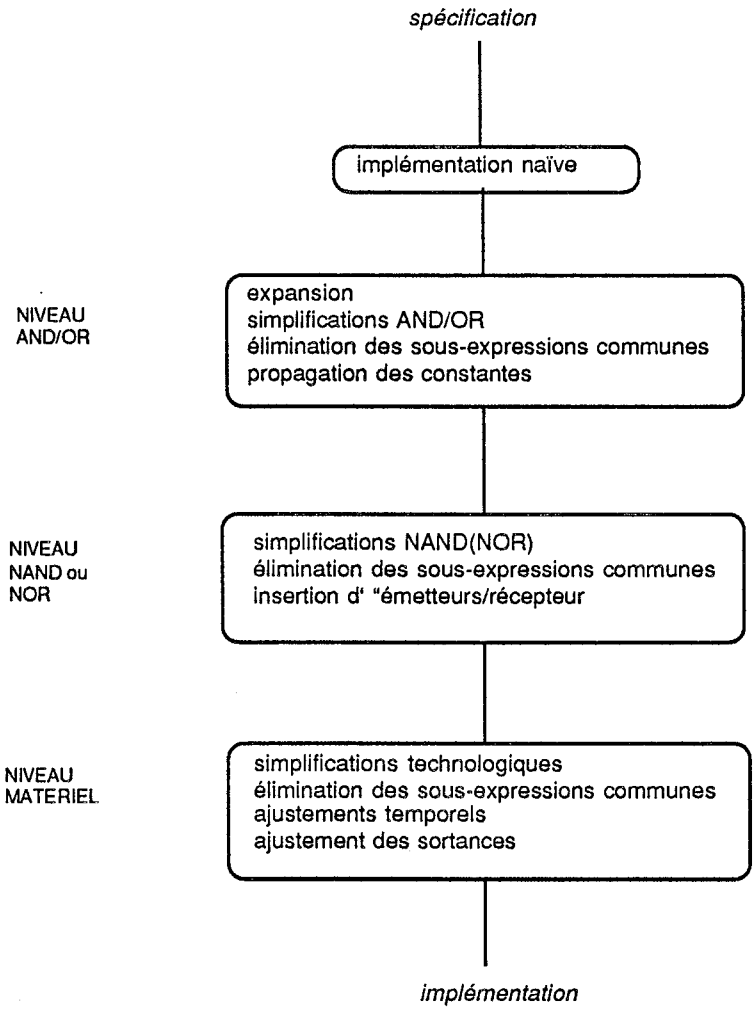
- aux tables de Karnaugh ([Han 84])

- à des combinaisons précises de portes logiques (le plus répandu)

- au niveau interrupteurs

Contrairement à l'optimisation de fonctions booléennes, la difficulté n'est plus de trouver une programmation efficace, mais de déterminer quelles transformations appliquer et surtout dans quel ordre les appliquer.

A titre d'exemple, nous reproduisons la structure générale du système LSS en 1980 :



Cet ordre global des transformations ne suffit pas pour garantir une bonne optimisation. Il faut également trouver des mécanismes pour assurer que les nombreuses transformations ne s'annulent pas les unes les autres. Les critères du type nombre de portes supprimées, longueur de chaîne, etc... qui introduisent des minimums locaux ne sont pas suffisants. En général, la solution adoptée est une hiérarchisation précise des transformations, mais il existe d'autres solutions (détaillées au chapitre 5).

3.7. le point de vue IA

La synthèse de circuits analogiques a attiré les concepteurs de systèmes experts en quête d'application. Le domaine s'y prête bien : grande complexité même pour la synthèse de petits circuits, grande importance de l'expertise... On trouve assez souvent des méthodes pour la synthèse de petits circuits passifs dans les exemples des articles d'intelligence artificielle (par exemple dans [Sus 78], [McDer 78]). Nous ne nous attarderons néanmoins pas sur ces exemples, car le problème de la synthèse de petits circuits analogiques diffère nettement du problème de la synthèse des (gros) circuits discrets.

De nombreux systèmes d'optimisation de circuits à base de règles sont baptisés "systèmes experts". Ces systèmes utilisent des mécanismes de contrôle relativement simples. Le côté expert est tout entier dans leur choix judicieux de règles d'optimisation et dans leur hiérarchisation. Nous développons le sujet dans les chapitres 4 et 5.

Ces systèmes sont à différencier de *DAA* [Kow 83] où toute la synthèse, c'est-à-dire l'allocation et l'optimisation, est faite dans un formalisme de règles de production (programmé en OPS5). Ici, le contrôle est plus sophistiqué. La sélection des règles se fait par chaînage avant sans retour en arrière. L'interprète cherche dans la mémoire de règles une règle dont toutes les préconditions sont vraies. Si plus d'une règle est applicable, la règle provenant de l'élément modifié le plus récemment est appliquée. S'il existe encore un choix, la règle la plus spécifique est sélectionnée. En ce sens, on considère que les règles de DAA sont temporairement ordonnées. Le processus est appliqué jusqu'à ce que plus aucune règle ne soit applicable. A ce moment, on considère que le circuit est réalisé.

On trouve également des environnements de synthèse intelligents : *Palladio* [Bro 83], *Redesign* ([Stei 84],[Stei 85]). Dans ces systèmes, il n'y a pas de synthèse automatique à proprement parler, mais un ensemble d'outils intelligents d'aide à la synthèse : outils de diagnostic, propagations de certaines propriétés... D'ailleurs, ces outils sont définis comme des "assistants intelligents".

On trouve dans [Per 78] un système d'optimisation booléenne fondé sur une méthode d'exploration d'états. Mais en général les systèmes rencontrés fonctionnent par raffinements successifs.

La notion d'apprentissage dépasse rarement l'enrichissement incrémental d'une base de données. Certains efforts sont faits : [Acos 86] décrit un système "raisonnant" par analogie. Le système travaille par génération de plans hiérarchiques. La synthèse d'un circuit utilise l'abstraction de synthèses déjà réalisées.

Dixon ([Dix 84]) divise les décisions d'un concepteur en trois catégories : décisions techniques, décisions de planification et décisions d'acceptabilité. Nous utiliserons cette classification pour caractériser nos systèmes d'optimisation dans les chapitres 4 et 5.

3.8. comment faire accepter l'outil aux concepteurs ?

La synthèse logique est un outil simple, très comparable à la compilation des langages de programmation de haut niveau. Pour que le compilateur logique entre dans les mœurs des concepteurs, il faut tout d'abord que le concepteur apprenne son langage source. De la même manière que les programmeurs qui ont dû se mettre à FORTRAN dans les années 60, puis à C et peut-être à ADA demain, les concepteurs connaîtront bientôt tous VHDL. Cette standardisation d'un langage favorisera certainement la pénétration de la synthèse logique, en évitant le rebutant apprentissage d'un nouveau langage pour chaque outil.

L'autre problème qui se pose à l'utilisation d'un outil de synthèse est la lecture du résultat. Il est très difficile de lire une liste de portes et d'interconnexions. Il suffit de regarder le résultat de la compilation d'un modèle simple comme dans le paragraphe 4.7.2 pour s'en convaincre. Le résultat doit être exprimé de manière graphique pour être lisible, et pouvoir être vérifié visuellement. Malheureusement les circuits logiques dessinés automatiquement sont rarement lisibles. Mais ce placement-routage ne constitue sans doute pas un problème insoluble[1]...

[1]Le problème du placement-routage d'un petit schéma n'est pas intrinsèquement difficile: un tel module a été développé pour Cascade. Toute la difficulté est de rendre lisible le résultat, ce qui contraint beaucoup le routage.

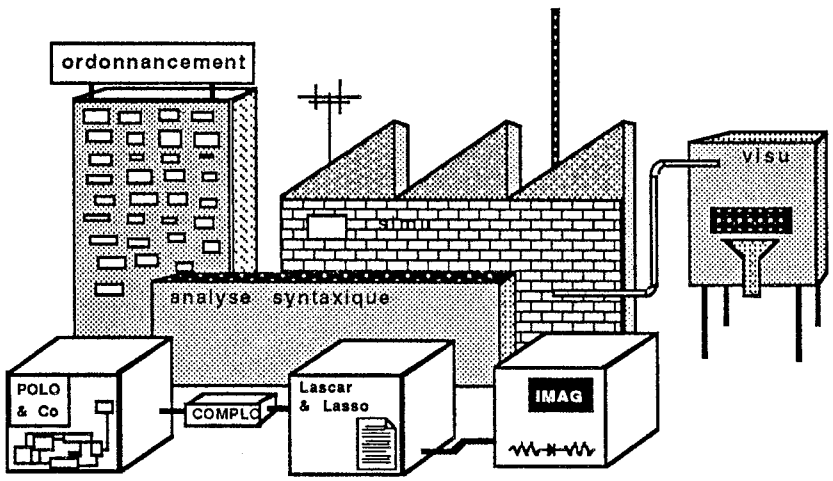
Chapitre 4

SYNTHESE LOGIQUE A PARTIR DE LASCAR: COMPLO

4-SYNTHESE LOGIQUE À PARTIR DE LASCAR: COMPLO

4.1. contexte et objectifs de l'outil

Le compilateur logique COMPLO est un des outils du système CASCADE. CASCADE comporte également des simulateurs, des outils graphiques... tous fondés sur une famille de langages. Tous ces langages ont des syntaxes très voisines: même forme générale, même expression de la structure, mêmes notions fondamentales des types, porteuses ...Ils permettent de décrire un système depuis le niveau "architecture" jusqu'au niveau "électrique fin". Nous ne rentrerons pas plus avant dans la description du système Cascade.



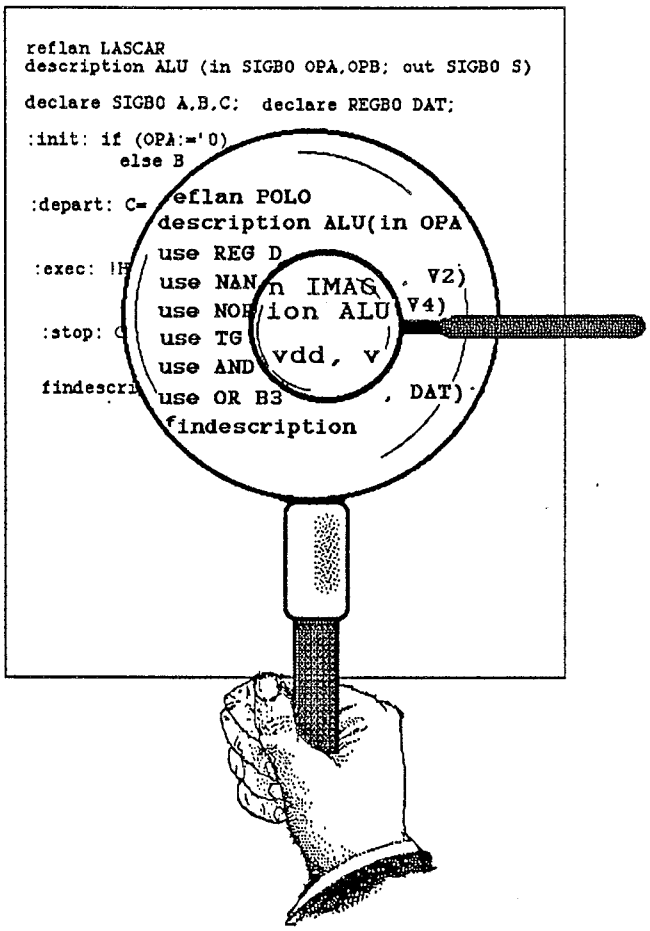
Le système CASCADE (vision symbolique)

Pour une vision plus complète du système et du langage Cascade, nous renvoyons le lecteur à ([Mer 85], [Dur 87]).

4.1.1. La place de complo dans Cascade

Les différents langages de Cascade matérialisent les étapes de la conception d'un circuit. On considère que la conception d'un ensemble digital commencera par sa description en LASSO. Une fois fixé le comportement algorithmique et les synchronisations entre sous-ensembles,

elle continuera au niveau LASCAR. A ce niveau, les fonctions des sous-ensembles seront fixées, ainsi que leurs interfaces. La synthèse continue alors par une traduction au niveau portes logiques POLO et/ou au niveau interrupteurs CASTOR. La conception finit au niveau électrique, où l'on distingue deux niveaux : le niveau découplé IMAG-D et le niveau fin IMAG-F.

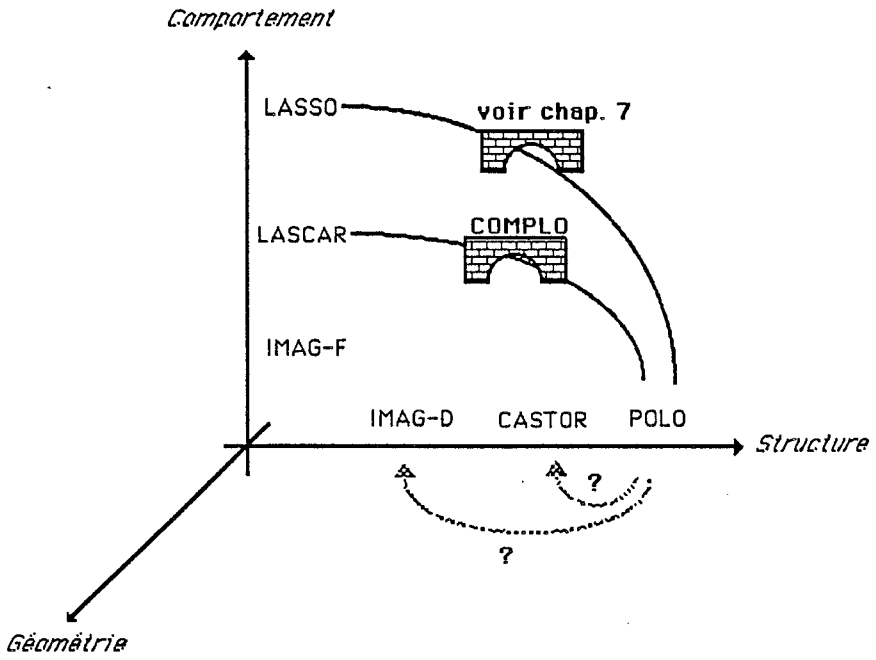


Conception strictement descendante en CASCADE

Tous ces langages ont en commun une base sémantique, ce qui permet de garder une certaine cohérence au cours des étapes de conception.

Dans Cascade, la conception est vue comme une série de traductions du modèle initial en LASSO, et passant par différents langages bien définis, alternativement utilisés comme langage cible puis source. Peut-on aider le concepteur en lui fournissant des traducteurs automatiques entre les différents niveaux ?

La réponse paraît simple quand il s'agit d'une traduction entre deux niveaux structurels (par exemple portes logiques -> interrupteurs ou interrupteurs -> électrique ou portes logiques -> électrique). Il "suffit" de donner les correspondances entre chacun des éléments utilisés dans les langages. C'est en effet une solution possible, mais qui donne rarement de bons résultats. Il vaut mieux choisir les éléments utilisés dans une bibliothèque en fonction de paramètres donnés au départ. On rejoint ici le problème des compilateurs structurels.



Cette figure utilise le schéma classique des trois axes de la conception de circuits: structure, comportement, géométrie.

Nous considérons ici la synthèse du langage LASCAR en POLO. Dans le chapitre 7 nous abordons la synthèse du langage LASSO. Bien que les deux langages aient des syntaxes quasiment identiques, les deux synthèses sont très différentes. La synthèse de LASSO est un problème de partitionnement, alors que la synthèse de LASCAR ressemble plus à la compilation d'un langage de programmation.

Le but du compilateur logique est de traduire la description fonctionnelle d'un circuit au niveau LASCAR en une description structurelle au niveau POLO, qui a une interface et un fonctionnement équivalent. L'application principale de cet outil est l'aide à la conception de circuit, mais ce n'est pas la seule : des programmes similaires sont utilisés

comme pré-compilateurs pour simuler au niveau portes des circuits décrits au niveau fonctionnel.

4.2. Le langage LASCAR

LASCAR est un des niveaux de langage de CASCADE. Il permet d'exprimer à la fois la structure et le comportement d'un circuit donné. Sans entrer dans le détail, nous dirons qu'un circuit est décrit par un modèle, qui est lui-même l'instanciation d'un ensemble de descriptions. Dans la suite, nous nous intéresserons uniquement à l'expression du comportement en LASCAR.

LASCAR décrit textuellement les fonctions effectuées par le matériel au niveau transferts de registres. Il permet d'exprimer des fonctions logiques ou entières, des synchronisations et des séquencements d'actions élémentaires.

LASCAR est un langage impératif : sa construction de base est l'affectation d'une variable ("signal", "registre") avec le résultat d'une opération. LASCAR est non procédural : l'ordre d'écriture des affectations n'influe pas sur le résultat. Mais on ne peut pas affecter plusieurs fois la même variable, à moins que ce ne soit sous des conditions disjointes.

Une description LASCAR combine deux notions : les notions correspondant à la partie opérative du circuit où sont effectuées les opérations et la partie contrôle où est calculé le séquencement des actions de la partie opérative. La partie opérative est décrite par des combinaisons de "signaux" et de "latches", par des transferts de registres et de mémoires. La partie contrôle est modélisée directement dans un formalisme d'états finis. Ces notions "conceptuellement" distinctes sont très liées dans l'écriture d'une description.

Au niveau de l'expression du parallélisme, on considère que tous les blocs d'un modèle sont actifs en même temps. La synchronisation et le contrôle des communications doivent être décrits explicitement par des signaux dans les interfaces des différentes descriptions. Ceci a comme conséquence qu'une description ne requiert aucune autre description pour être synthétisée.

4.3. Stratégie de la synthèse

4.3.1. deux phases

COMPLO reproduit le schéma de Alice Parker (voir chap. 3). La phase d'allocation est appelée ici implémentation naïve. L'optimisation et la composition se retrouvent dans la phase d'optimisation. Nous n'abordons pas ici le problème du placement/routage.

4.4. le module d'implémentation naïve

Pratiquement, l'entrée de COMPLO est une liste d'instructions de trois types : instructions simples, instructions conditionnées et instructions de séquencement. La sortie représente la description structurelle du circuit obtenu, c'est-à-dire une liste d'appels de descriptions génériques instanciées.

La compilation se fait en deux phases principales : compilation des instructions, puis regroupement des affectations. La compilation des instructions est faite par un mécanisme d'analyse syntaxique qui fait intervenir différents types de traitements sémantiques : compilation des expressions, compilation des transitions d'états, traitement des conditions.

4.4.1. compilation des expressions

le cas des instructions simples

Nous considérons ici toutes les instructions simples, sous portée d'automate ou pas. Les instructions simples sont les instructions d'affectation. Toutes ces instructions sont de la forme :

```
<porteuse>  :=  <expression>
```

L'affectation peut être soit une connexion de signaux, soit un chargement de registres. Le travail effectué pendant la compilation est le même dans les deux cas : on crée la circuiterie qui implémente l'expression. Le "raccordement" de cette circuiterie au signal ou au registre ne sera effectué que dans la phase de regroupement des affectations.

Les expressions traitées sont toutes les expressions possibles en LASCAR, c'est-à-dire parenthésées de manière quelconque, ou des expressions *si...alors...sinon* etc... Les opérateurs reconnus sont les opérateurs classiques : *et, ou, non-et, non-ou , ou exclusif* ... binaires ou n-aires, ainsi que les opérateurs spéciaux permettant d'appliquer un opérateur à un vecteur (ou à la première dimension d'un tableau quelconque) : "*reduc ou*", "*reduc et*" et "*reduc non*".

L'idée générale du traitement est la suivante. On parcourt en profondeur d'abord l'arbre représentant l'expression, en évaluant chaque nœud. Les nœuds terminaux sont des signaux (ou des sorties de registres, ce qui revient au même). Le résultat de leur évaluation sera le signal lui-même. Que faire si le nœud terminal est une constante ? (question abordée

dans [Fri 69]). Dans COMPLO la solution est simple : on crée un "registre constante" de la largeur désirée. Le problème de sa simplification et de son implémentation est ainsi reporté à la phase de composition. Les nœuds non terminaux sont des opérateurs. Leur évaluation se fait en trois temps :

- création dans la structure de données d'une boîte dont le type correspond à celui du nœud,

- connexion en entrée de cette boîte des évaluations des nœuds fils,

- création d'un signal de sortie qui sera la "valeur" du nœud.

Le résultat de ce traitement sera d'une part une liste de boîtes du type :

```
( <nom de la boîte> <type> <liste d'entrées> <liste de sorties>)
```

et d'autre part une liste de triplets

```
(<signal affecté> <condition d'affectation> <signal implémentant l'expression>)
```

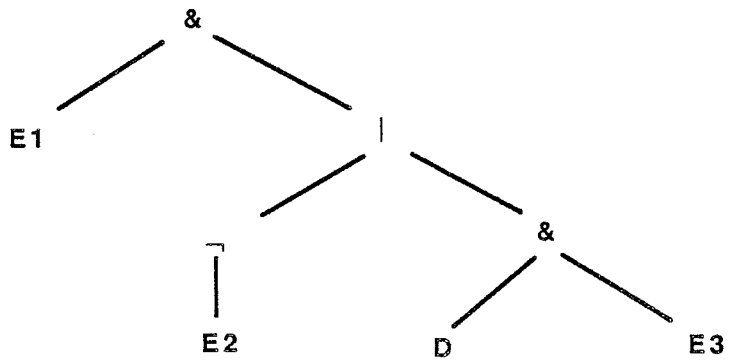
Le signal affecté est le signal apparaissant en partie gauche de l'affectation. La condition d'affectation sera "nil" si l'instruction n'est pas sous portée d'automate et sinon ce sera le signal correspondant à l'état courant de l'automate (signal sortie des fonctions de décodage). Le signal restant est le résultat de l'évaluation de l'expression. Les paragraphes ci-dessous illustrent le processus sur quelques expressions simples :

compilation d'une expression simple parenthésée

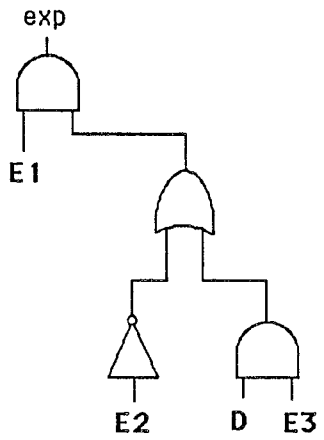
L'expression :

```
E1 & ( ¬ E2 | D & E3 )
```

qui est représentée par l'arbre syntaxique



sera implémentée par :



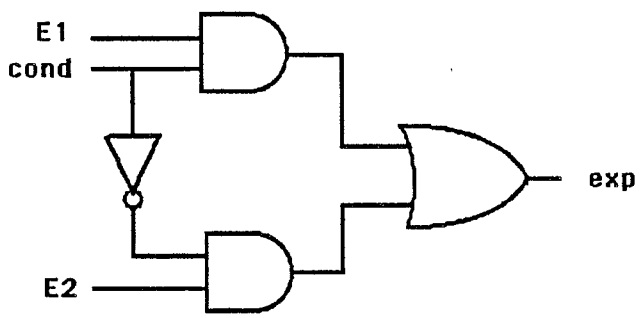
compilation d'une expression if...then...else

Cascade permet l'écriture d'expressions conditionnelles. Une expression conditionnelle est évaluée de deux manières différentes (partie *then* ou partie *else*) selon la valeur à Vrai ou à Faux de sa condition. Comme il faut toujours que cette expression ait une valeur, la partie *else* est obligatoire, comme dans le cas de la construction équivalente *...when...else* de VHDL . Il ne faut pas confondre l'*expression* conditionnelle avec *l'instruction* conditionnelle, même si la transposition de la première dans la deuxième est immédiate.

Dans COMPLO, l'expression :

```
if COND then E1 else E2 endif
```

sera traduite par :



On retrouve la même traduction dans le compilateur de Wislan ([Vai 82]).

note : Cette richesse d'expression est très utile pour COMPLO. Comme l'expression conditionnelle et l'instruction conditionnelle ne sont pas traduites de la même manière, le concepteur pourra mieux maîtriser la synthèse par un choix judicieux de l'une ou de l'autre.

compilation d'une expression "reduc"

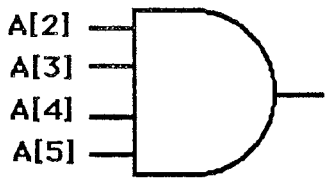
L'opérateur "reduc" est une facilité d'écriture qui permet d'appliquer un opérateur associatif à un vecteur. Plutôt que d'écrire :

```
A[2] & A[3] & A[4] & A[5]
```

On écrira :

```
reduc &! A[2:5]
```

... et ceci sera traduit par :



Ceci est généralisable aux expressions portant sur la première dimension d'un tableau quelconque.

expressions d'adressage

Bien que ne posant aucune difficulté conceptuelle, la traduction des opérations portant sur des variables concaténées ou sur des sous-vecteurs d'une variable est difficile à programmer. En effet au niveau du matériel, et donc au niveau du programme, on ne connecte que des variables de mêmes dimensions. Plusieurs solutions sont utilisées :

- synthèse logique ne prenant en compte que des scalaires : c'est la solution la plus simple à mettre en œuvre, mais aussi la plus lourde.

- concaténation avec des zéros. Solution adoptée par Liddell ([Lid 70],[Mer 73]).

- création de cellules de concaténation. C'est la solution prise dans COMPLO.

Nous avons abordé les opérations d'adressage portant sur des constantes. La traduction des expressions d'adressage plus complexes suppose que l'on ait défini des champs distincts pour les adresses et pour les données. Cette fonctionnalité n'est pas prévue dans le compilateur de septembre 1985. Néanmoins l'utilisateur peut contourner le problème en utilisant une variable intermédiaire d'adresse.

compilation des expressions dans les instructions conditionnées

Dans le paragraphe précédant on a vu que le résultat de la compilation d'une instruction simple est double : d'une part une liste de boîtes et d'autre part un triplet *(signal affecté, condition de chargement, signal résultat)*. Le résultat est le même pour une instruction conditionnée, mais :

1- La liste des boîtes créées n'implémente pas que l'expression mais également la *condition courante* ,

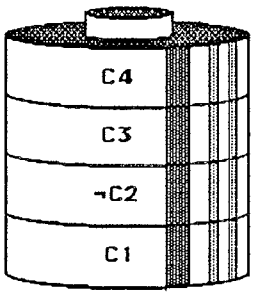
2- la condition de chargement n'est plus un signal d'état ou rien, mais un ET logique entre les conditions portant sur le chargement.

Ce mécanisme est implémenté en utilisant une pile . Cette pile *pile condition* reflète les différentes conditions rencontrées avant l'évaluation d'une expression.

Par exemple, l'évaluation de exp1 dans l'instruction suivante :

```
if C1 then D .= exp3;
       if C2 then B .= exp2;
            else      if C3 then
                           if C4 then A.= exp1;
                           endif
                      endif
       endif
endif
```

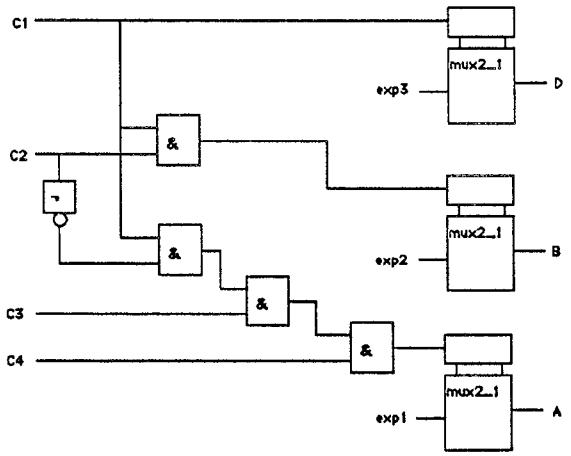
La pile sera dans l'état suivant lors de l'évaluation de exp1 :



pile condition

remarque : dans la pile ci-dessus, C2 apparaît complémenté : ceci correspond au *else* rencontré après *if C2*.

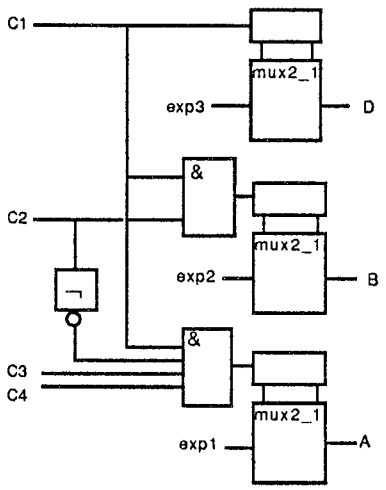
Il y a deux manières de créer le signal *condition courante* à partir de cette pile. La première consiste à évaluer incrémentalement la pile. Chaque évaluation de la pile est mémorisée et placée en haut de pile. L'avantage est qu'on ne duplique pas inutilement le matériel. L'inconvénient est que l'on risque de créer des chaînes longues. L'exemple ci-dessus donnera :



implémentation des conditions "en cascade"

Cette solution a été choisie dans [Mer 73].

L'autre solution consiste à faire un ET logique entre tous les éléments de la pile. Cette deuxième solution est celle que nous avons adoptée. Elle donne comme résultat des chaînes beaucoup plus courtes, mais qui peuvent devenir très larges. Autre inconvénient, cette méthode amène une duplication du matériel. Ces défauts peuvent être facilement corrigés dans la phase d'optimisation qui suit l'implémentation naïve. Dans notre exemple, la condition sera réalisée par le matériel suivant :



implémentation des conditions "en parallèle"

En pratique, les deux solutions donnent des résultats similaires pour des profondeurs d'imbrication de 1 ou 2. On ne rencontre guère d'imbrications plus complexes dans les exemples des concepteurs.

les instructions case

Dans COMPLO elles sont traitées comme une série de if...then...else.

4.4.2. compilation des parties contrôle

Le but de cette phase est de créer la circuiterie implémentant l'automate décrit dans la description source.

Dans le cas de COMPLO, on effectue un codage trivial des états : à chaque état un registre (remplacé dans COMPLO 2 par un codage canonique au hasard). Toutes les fonctions de codage et de décodage se trouvent simplifiées, puisque supprimées. La synthèse de l'automate est faite incrémentalement, à mesure que l'analyse rencontre les instructions "load <état>;". Les instructions sous portée de l'état sont traitées

normalement, à la seule différence que le signal d'état est inséré dans la pile conditions. Les instructions de transition d'état sont traitées comme des affectations de registre : au registre d'état est affecté le code de l'état suivant. La méthode garantit qu'en fonctionnement normal, il y a au plus un état actif. Néanmoins le type du codage (un '1' parmi (n-1) '0') implique que l'automate se trouvera dans un état indéterminé à la moindre anomalie de fonctionnement.

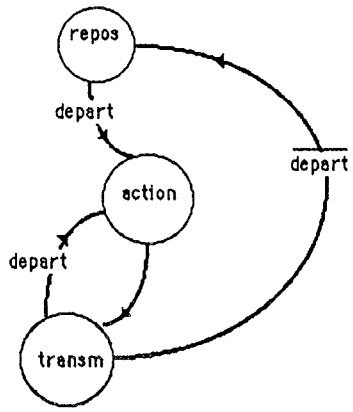
Le résultat de cette phase est le même que pour la précédente, c'est-à-dire :

- augmentation de la liste des boîtes, avec les registres d'état et la circuiterie de séquencement,

- augmentation de la liste des affectations.

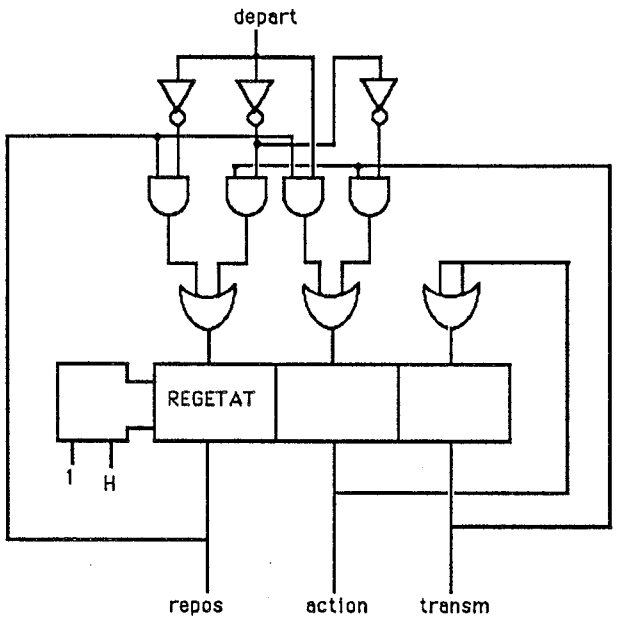
un exemple

les transitions suivantes :




```
:repos: if depart then !H! load action;
:action: !H! load transm;
:transm:  if ~depart then !H! load repos;
                          else !H! load action; endif
```

sera traduit par :



implémentation d'un automate dans COMPLO

4.4.3. regroupement des affectations

Lascar est un langage d'affectation, et ses instructions ont la forme :

```
<recepteur>  := <signal expression> ;
```

Comme il est dit plus haut, il y a deux sortes d'affectations : la connexion de signaux (.=) et le chargement de registres (<=).

Les deux premières phases de COMPLO créent des signaux <signal expression> correspondant à la valeur des expressions apparaissant en partie droite. Cette création se concrétise par une liste de triplets :

(<récepteur>, <condition d'affectation>,<signal expression>)

Il s'agit maintenant de connecter le récepteur au signal expression. Quatre cas peuvent se présenter :

- soit le récepteur est du type "SIGNAL", et il n'est affecté qu'une fois : affectation simple d'un signal.

- soit le récepteur est du type "SIGNAL", et il est affecté plusieurs fois, sous des conditions différentes : affectation multiple d'un signal.

- soit le récepteur est du type "REGISTRE", et il n'est affecté qu'une fois : affectation simple d'un registre.

- soit le récepteur est du type "REGISTRE", et il est affecté plusieurs fois, sous des conditions différentes : affectation multiple d'un registre.

affectation simple d'un signal.

C'est le cas le plus simple : un ET est créé. Ses entrées sont le signal expression et le signal condition d'affectation. Si la condition d'affectation est "Vrai", le ET n'est pas créé. Si le signal expression est d'une dimension > 1, il sera traduit à la composition par la cellule "ad-hoc"). La sortie du ET est substituée par le signal récepteur, ou par l'extrémité de son tampon pour les entrées/sorties.

affectation multiple d'un signal.

Dans une même description, il peut y avoir plusieurs affectations d'un même signal. On suppose que les conditions d'affectation sont disjointes. Il y a alors création d'un aiguilleur de données[1]. Cet aiguilleur est noté MUXn/m , où n est le nombre d'entrées et m la largeur de ces entrées. En fait, n ne prend que les valeurs 2, 4, 8, 12, 16, ... La sémantique de cet aiguilleur est la suivante :

- si l'entrée de contrôle C[i] (avec $1 \leq i \leq n$) est à l'état haut, la sortie sera connectée à l'entrée E[i],

- si aucune entrée de contrôle n'est à 1, la sortie sera dans l'état haute impédance. Si plusieurs entrées de contrôle C[i] et C[j] sont à 1, ce

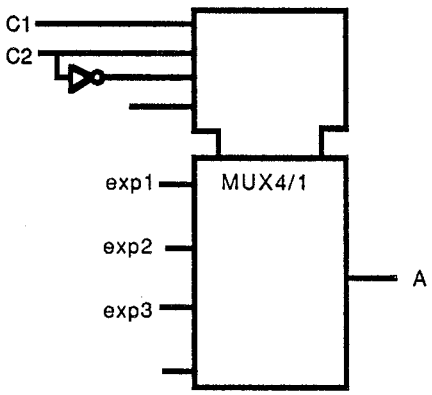
[1]Cet aiguilleur de données est le plus souvent appellé abusivement "multiplexeur" (on trouve également "bus", "link"...). Nous utiliserons cette dénomination, en précisant lorsque il s'agira d'un multiplexeur aux entrées codées.

qui en théorie ne devrait pas arriver, la sortie sera soit à l'état haut, soit connectée au "ou" bit à bit des entrées E[i] et E[j], suivant l'implémentation finale de l'aiguilleur.

Par exemple, les instructions suivantes :

```
if C1 then A.= exp1 ;
if C2 then A .= exp2; else A.= exp3; endif
```

...seront implémentées par :

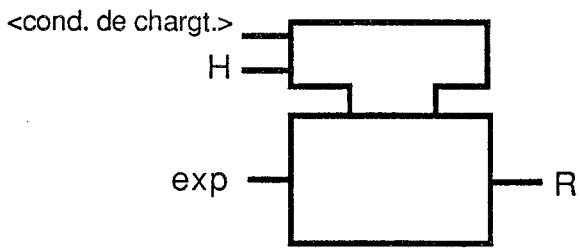


affectation simple d'un registre.

Les registres utilisés ont tous la même interface : entrée horloge (scalaire) , entrée validation (scalaire) , entrée données (largeur m) et sortie (de largeur m). L'affectation simple d'un registre, qui s'écrit :

```
!H! R  <= exp ;
```

se traduit d'une manière assez simple : une boîte de type REGm est créée. On connecte l'entrée "horloge" à H, l'entrée "validation" à la condition de chargement (qui dépend de l'automate, des éventuelles conditions ...) et l'entrée "données" au signal expression. La sortie du registre est connectée à un signal portant le même nom que le registre.



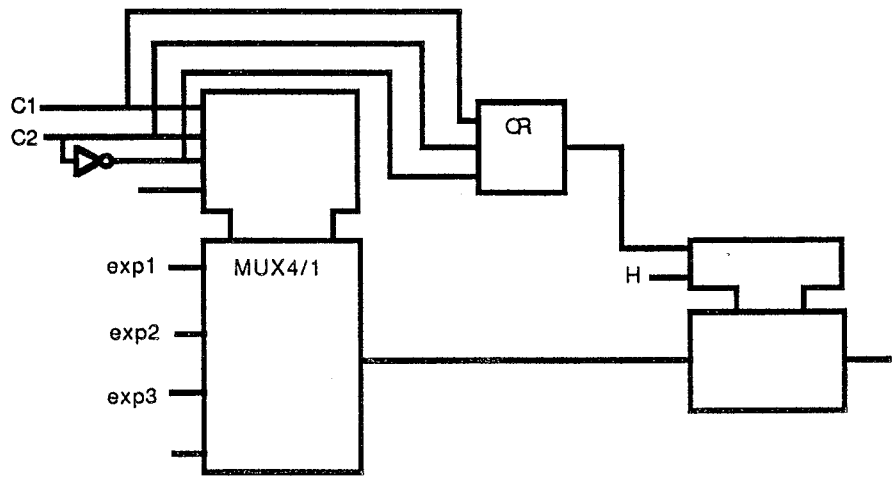
affectation multiple d'un registre.

Lorsqu'un registre est affecté par différentes expressions sous différentes conditions, on procède de la manière suivante : il y a création d'un multiplexeur exactement de la même manière que pour un signal. De plus, on crée un OR regroupant toutes les conditions de chargement. La sortie de ce OR est connectée à l'entrée "validation" du registre. L'entrée horloge est connectée également à un OR des horloges.

Par exemple :

```
if C1 then !H! R <= exp1 ;
!H! if C2 then R <= exp2 else R <=exp3 endif ;
```

Cette instruction sera traduite par :



4.4.4. récapitulatif des primitives de COMPLO

Les cellules qui constituent les circuits générés par COMPLO sont les suivantes :

opérateurs combinatoires :

AND n entrées . OR n entrées . NAND n entrées . NOR n entrées . XOR 2 entrées . XNOR 2 entrées . NOT 1 entrée.

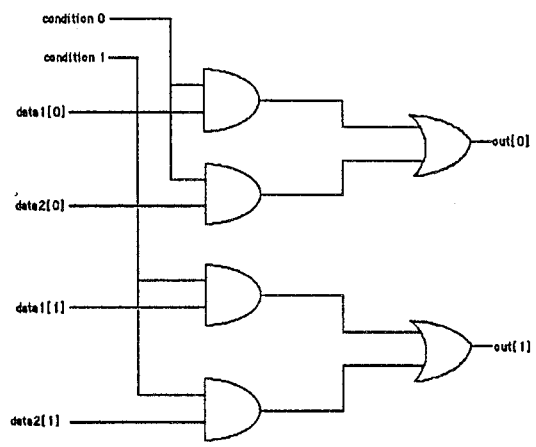
registres :

REGn où n est la largeur du champ de données . Ce sont des registres à 3 entrées : entrée horloge , entrée validation, entrée données de largeur n. Leur sémantique est celle de la bascule D.

multiplexeurs :

Les multiplexeurs utilisés sont notés MUXn/m. En fait ce sont des aiguilleurs de données. La sortie `out[i]` prend la valeur de l'entrée `dataj[i]` si l'entrée `conditionj` est à l'état haut. Ceci impose qu'à tout moment, il n'y ait qu'un seul signal `conditionj` à l'état haut. Que se passe-t-il si deux signaux condition sont à 1 ? Tout dépend de la réalisation du multiplexeur.

Nous donnons ci-dessous une réalisation de MUX2_2 :



Une réalisation possible de MUX2_2

Avec cette réalisation de MUX2_2, la sortie est un "OU" des deux entrées lorsque plusieurs signaux condition sont à 1.

4.4.5. le module d'optimisation

Le module d'optimisation a pour but de transformer le circuit composé d'éléments génériques de la phase d'implémentation naïve en un circuit équivalent composé d'éléments issus d'une bibliothèque.

Le module d'optimisation décrit ici peut servir à faire du "remapping", c'est-à-dire traduire un circuit conçu pour une technologie donnée en un circuit équivalent dans une autre technologie. Par exemple, on pourra traduire un circuit composé de portes NOR en un circuit équivalent en NAND.

Pour reprendre la trilogie décisions techniques / décisions de planification / décisions d'acceptabilité :

Les décisions techniques sont implémentées sous la forme d'opérateurs d'optimisation. Par exemple :

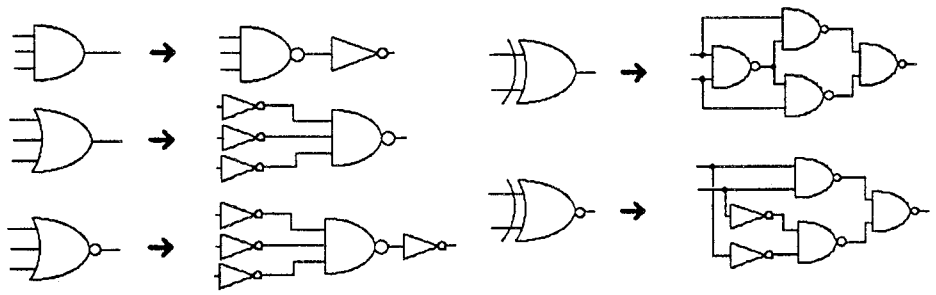
si il existe deux inverseurs A et B
tels que la sortie de A est reliée à l'entrée de B
alors supprimer B
et connecter la sortie de Y à l'entrée de X
fin

Les décisions de planification sont implémentées sous la forme de séquences d'opérateurs d'optimisation (ce qu'on appelle souvent "règles").

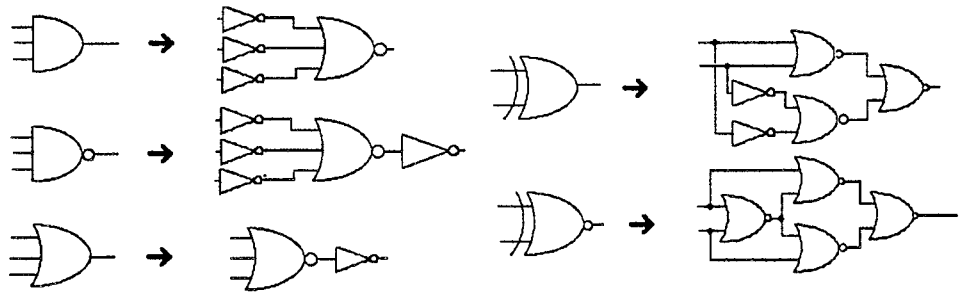
exemples de transformations locales

Nous donnons quelques-unes des transformations locales utilisées. La plupart de ces transformations ont été glanées dans [Bra 85], [Dar 80], [Dar 81], [Vai 82]. Comme le prototype était très facile à modifier, certaines transformations et les séquences de planification ont été modifiées au fur et à mesure de l'utilisation.

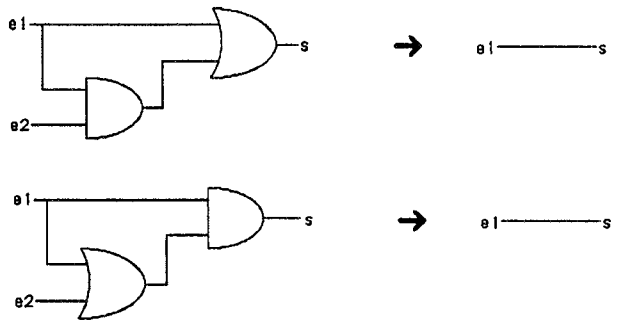
transformation en NAND



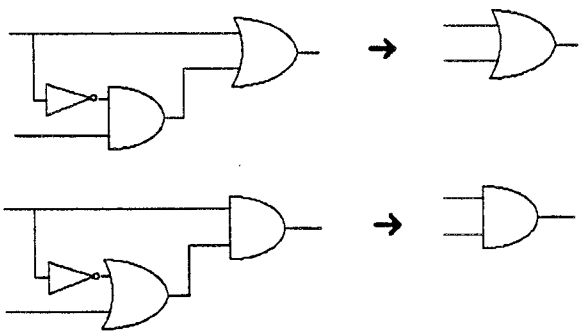
transformations en NOR



minimisation par absorption



minimisation par simplification



Il serait inutile de donner toutes les transformations utilisables. Le point le plus important est de pouvoir les modifier facilement. Si les premières modifications ont été effectuées en modifiant le code PROLOG,

nous avons rapidement utilisé un macro-générateur en Pascal. L'étape suivante consisterait à les écrire dans un langage simple, et de les compiler pour obtenir un code efficace : c'est ce que nous avons fait pour la phase d'optimisation décrite au chapitre suivant.

4.5. correction des circuits obtenus

4.5.1. correction du module d'implémentation naïve

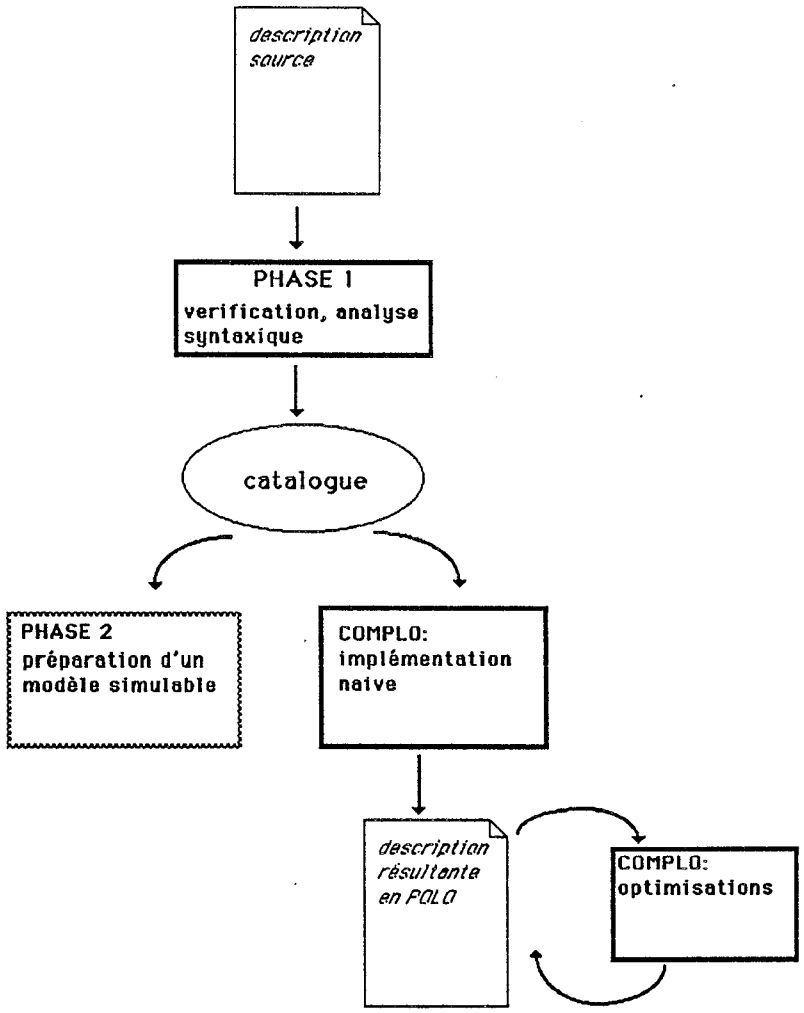
Le module d'implémentation traduit de la manière la plus simple les constructions de sa description source. Il n'y ajoute pas de sémantique : en LASCAR, le détail du séquencement est fixé, et les opérateurs ont tous une sémantique précise. Donc, mis à part les erreurs de programmation, on peut considérer que le module d'implémentation crée des circuits qui reproduisent fidèlement les fonctionnalités de la description source (en particulier les erreurs de conception).

4.5.2. correction du module d'optimisation

Le module d'optimisation n'effectue que des transformations locales, et remplace donc des combinaisons de composants par d'autres combinaisons. Si l'on ne considère que l'aspect fonctionnel du circuit, en ignorant les aspects longueur des chaines ou "timing"..., la correction de ces remplacements ne dépend que des règles utilisées. C'est donc du concepteur qui écrit ses règles que dépend la correction du module d'optimisation.

4.6. notes d'implémentation

COMPLO est intégré dans le système CASCADE. Le module d'implémentation naïve utilise comme format d'entrée une représentation codée appelée "catalogue"[1]. Ce "catalogue" est généré à partir de la description source après une analyse lexicographique, une analyse syntaxique et une série de vérifications sémantiques qui sont effectuées par le programme "PHASE1". La syntaxe des données du catalogue est décrite par une grammaire LL(1).



La place de COMPLO dans le système CASCADE

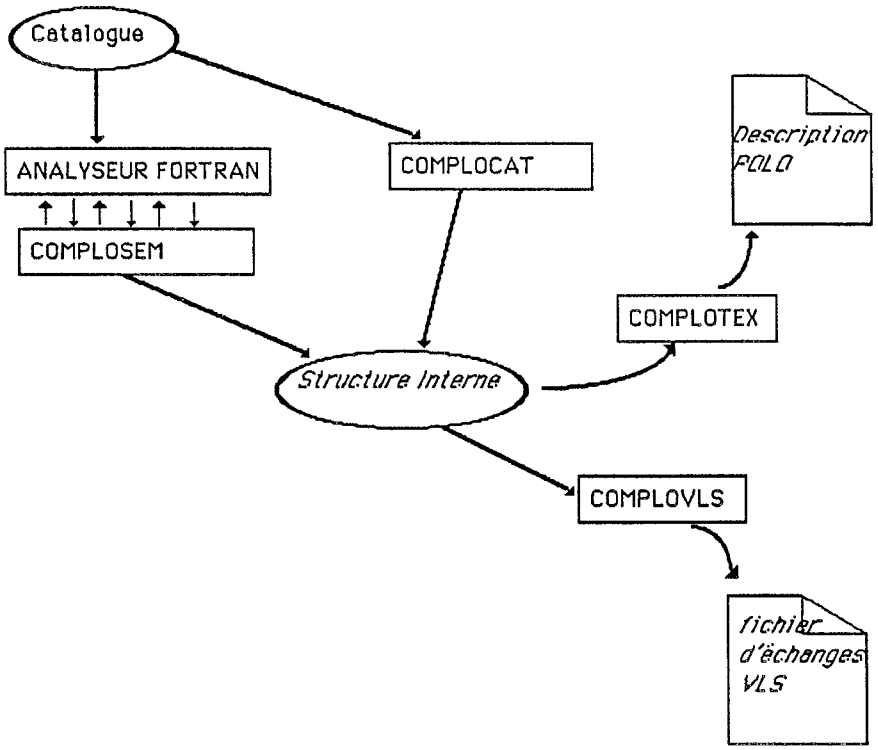
[1] dans son format de Septembre 85

4.6.1. implémentation du module naïf

Le programme COMPLO est constitué de deux parties :

- D'une part, l'analyseur syntaxique LL(1) qui analyse la structure du catalogue. Les tables de cet analyseur sont générées automatiquement par le transformateur TRGLL1 disponible au Centre Interuniversitaire de Calcul de Grenoble (CICG). Le programme utilisant ces tables est écrit en FORTRAN.

- D'autre part, la deuxième composante de COMPLO est l'ensemble des fonctions sémantiques qui génère réellement le circuit (MODULE COMPLOSEM). Ces fonctions sémantiques sont écrites en Pascal. D'autres modules écrits en Pascal viennent s'ajouter : module "COMPLOTEX" de traduction du résultat dans différents formats (langage POLO, format d'échanges graphiques VLS, format PROLOG), module "COMPLOCAT" de décodage des données du catalogue.



Organisation interne du module d'implémentation naïve

Dans la première version de COMPLO, la structure de données de travail est composée de deux listes : une liste de signaux, et une liste de

boîtes. Le mécanisme ci-dessus crée des boîtes (donc impose une mise à jour de la liste de boîtes) et crée des signaux (même chose pour les signaux) et enrichit la structure de pointeurs entre les deux listes. Comme ces pointeurs sont des pointeurs Pascal, le mécanisme perd beaucoup de temps en mises à jour de pointeurs etc... La mise au point du programme est rendue pénible.

Ce problème a également été rencontré dans la mise au point du prototype de compilation logique de Wislan (bien qu'il soit plus simple). Les auteurs ont abandonné les structures compliquées pour revenir à une simple table des composants. Les temps perdus de parcours des tables se retrouvent car il n'y a pas de structures de pointeurs à remettre continuellement à jour. Pour la réalisation de l'outil similaire décrit au chapitre suivant, la programmation en Lisp permettra d'utiliser des méthodes très simples. En particulier, les structures de données seront des listes identiques à celles que nous utilisons dans ce chapitre pour décrire les méthodes.

Bogue : Le langage LASCAR prévoit qu'en l'absence de chargement explicite, l'état courant est conservé. La version de COMPLO de Septembre 85 n'assure pas ce fonctionnement.

4.6.2. implémentation du module d'optimisation

Après avoir étudié différents "systèmes experts d'ordre 1", nous avons choisi de réaliser le module d'optimisation en langage PROLOG, pour les raisons suivantes :

* l'application de toute transformation locale commence par l'identification dans la description de la combinaison de portes que l'on veut remplacer. C'est un mécanisme de filtrage. Le filtrage est un cas particulier d'unification et PROLOG est fondé sur ce principe : la programmation se fait donc de manière "naturelle".

* PROLOG est également un langage de programmation, ce qui permet d'intégrer les règles avec des mécanisme autres : analyse syntaxique du langage POLO, mise en forme du résultat.

Le circuit à traiter est représenté par des clauses déclaratives du style

```
us(AND,first_gate,in1,in2,out).
```

Ces clauses sont générées par l'analyse syntaxique d'une description en POLO (l'analyse syntaxique est effectuée par un programme PROLOG).

Les règles sont implémentées sous la forme de clauses de Horn avec une partie droite. Ces règles sont programmées directement en PROLOG. Pour diminuer le nombre de règles activables et donc améliorer les temps de

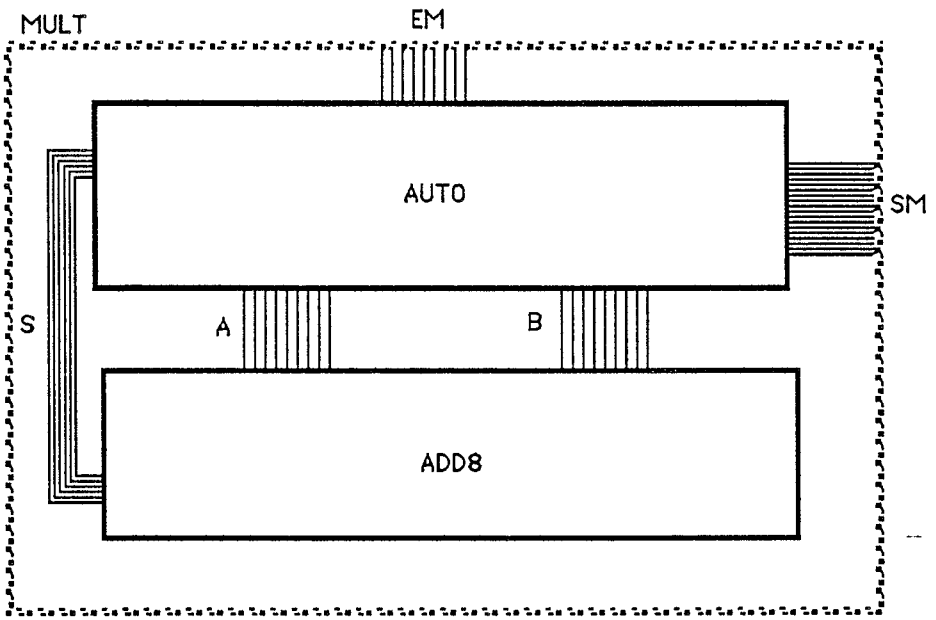
calcul, les règles sont organisées en "contextes" : contexte AND/OR, contexte NAND, contexte NOR...

4.7. un exemple : Synthèse d'un multiplieur

Nous avons choisi comme exemple la synthèse d'un multiplieur 8 bits. Il permet d'illustrer presque toutes les notions du langage, tout en restant d'une taille raisonnable. Nous utiliserons le même exemple au chapitre suivant.

4.7.1. La description source

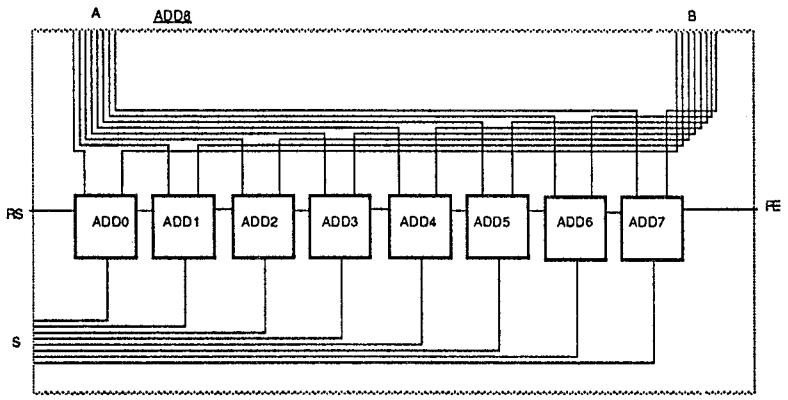
Bien que nous ne synthétisions ici que l'automate du multiplieur, nous allons décrire le modèle complètement. Le multiplieur est composé de deux parties, un additionneur ADD8 et un automate AUTO.



La description la plus englobante MULT est une description structurelle :

```
reflan LASCAR
description MULT
( in HORLOGE H ; in SIGB0 EM[0:7], DEPART, RE;
       out SIGB0 SM[0:15], OCCUPE)
body
declare
SIGB0 S[0:7], RS;
REGB0 A[0:7], B[0:7];
external FULLADD, AUTOMATE;
use    FULLADD ADD8 (A, B, S ,RS );
       AUTOMATE AUTO(H,EM,DEPART,S,RS,A,B,SM,OCCUPE);
enddescription
```

L'additionneur 8 bits FULLADD est composé de 8 additionneurs ADD de 1 bit. Les signaux A[i] et B[i] sont les opérandes des additionneurs. Les connections R*i* entre les additionneurs assurent la propagation de la retenue.



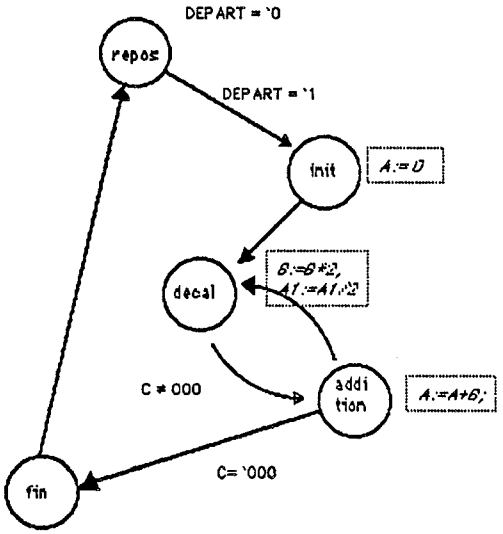
```
lanref LASCAR
description FULLADD (in REGB0 A[0:7], B[0:7];
       in SIGB0 RE;out SIGB0 S[0:7],RS)
body
relations
declare SIGB0 R1,R2,R3,R4,R5,R6,R7;
external ADD;
use ADD     ADD7 (A[7],B[7],RE,S[7],R7),
            ADD6 (A[6],B[6],R7,S[6],R6),
       ADD5 (A[5],B[5],R6,S[5],R5),
       ADD4 (A[4],B[4],R5,S[4],R4),
       ADD3 (A[3],B[3],R4,S[3],R3),
       ADD2 (A[2],B[2],R3,S[2],R2),
       ADD1 (A[1],B[1],R2,S[1],R1),
       ADD0 (A[0],B[0],R1,S[0],RS);
enddescription
```

la description de l'additionneur est la suivante :

```
reflan LASCAR
description ADD
( in REGB0 A,B; in SIGB0 RE; out SIGB0 ST, RS)
body
declare SIGB0 S1,R1,R2;
relations
       R1 := A & B,
       S1 := A xor B,
       R2 := RE & S1,
       ST := RE xor S1,
       RS := R1 | R2
enddescription
```

L'automate.

L'automate compte 4 états : on retrouve facilement l'algorithme d'origine.



Nous allons nous intéresser à la première étape, l'étape REPOS. Les instructions **sous portée de l'état** REPOS suivent l'étiquette :

```
: REPOS :
```

Sur le front montant de l'horloge H, on charge le registre B avec le signal d'entrée EM :

```
! H ! B <= EM
```

Si le signal DEPART est à l'état haut `1, on passe à l'état INIT et on charge le registre BUSY avec la valeur `0, sinon on reste dans l'état REPOS. Le point-virgule délimite la portée de l'horloge :

```
…if DEPART then load INIT, BUSY := `0 else load REPOS endif;
```

En fait, il est convenu implicitement qu'un automate reste dans l'état où il se trouve si on n'explicite pas de changement. La ligne précédente est donc équivalente à :

```
…if DEPART then load INIT, BUSY := `0  endif;
```

La description entière s'écrit :

```
lanref LASCAR
description AUTOMATE
 ( in HORLOGE H; in SIGB0 EM[0:7],DEPART,S[0:7],RS;
   out REGB0 A[0:7], B[0:7]; out SIGB0 SM[0:15], OCCUPE)
body
declare
       SIGB0 RC0,RC1,RC2;
       REGB0 BUSY, C[0:2],A1[8:15],R;
relations
 OCCUPE:=BUSY;
 RC2 := `1,
 RC1 := RC2 & C[2],
 RC0 := RC1 & C[1]

 :REPOS:!H! B <= EM,
       if DEPART then load INIT, BUSY := `0 endif;

 :INIT:!H! A <= `00000000, A1 <= B, B <= EM,
C <= `000,  BUSY <= `1, load ADDITION;

:ADDITION: if A1[15] then !H! A <= S, R <= RS;
else !H! R <= `0; endif,
!H! C <= (RC0 \ RC1 \ RC2) xor C, load DECAL;
 : DECAL:  !H! A <= R \ A[8:14],
A1 <= A[7]\A1[8:14],
if reduc| C then load ADDITION;
else  load FIN; endif;
:FIN: SM := A \ A1,
!H! load REPOS;
enddescription
```

4.7.2. Résultat de la synthèse de la partie automate

Le résultat de la compilation peut apparaître dans deux formats:

* d'une part le format VLS: ce format est commun à différents outils graphiques de CASCADE: floorplanner, éditeur structurel, placement/routage[1]. Il contient des données structurelles, ainsi que des données graphiques optionnelles. Dans le cas de COMPLO, les données graphiques sont laissées en blanc. L'intérêt de ce format est de permettre la visualisation des circuits compilés.

* d'autre part, dans le langage POLO, qui est décrit en annexe. Dans ce format, la description obtenue peut être simulée. C'est dans ce format que nous donnons le résultat de la compilation de l'automate du multiplieur.

description POLO du résultat

La version présentée ci-dessous est une version simulable[2]. La version visualisable est plus complexe:

- les concaténations en interface (opérateur \) sont remplacées par des boîtes "CONCAT"

- les constantes (`0) sont remplacées

- les signaux sont tous scalaires

```
lanref POLO
description AUTOMATV2 (in VARLOG H,EM[0:7],S[0:7],RS; out VARLOG
A[0:7],B[0:7],SM[0:15],OCCUPE)
corps
declare VARLOG RC0,RC1,RC2,ETAT[1:5],COUT[1:3],SIG_1;
externe MCELL_1, MCELL_3, MCELL_8, MCELL_5, MUX2_1, MUX3_1, MUX2_8,
MUX3_8, MUX2_16, SHIFT;
use MCELL_5 REG_ETAT (H, H, B6.OUT\B5.OUT\B8.OUT\ETAT[3]\B10.OUT,ETAT)
```

---

[1] le module de placement/routage existe à l'état de prototype. Il est destiné à la visualisation des descriptions structurelles. Il peut placer environ vingt éléments, et les router dans la plupart des cas.

[2] compatible avec une version du simulateur POLO-S qui accepte les entorses à la règle de non-référence avant.

```
use MCELL_3 C (H, B13.OUT, COUT);
use MCELL_1 BUSY (H, B12.OUT, OCCUPE),
      R (H, B23.OUT);
use MCELL_8 A1 (H, B27.OUT, , ),
      A (H, B25.OUT, , ),
      B (H, B26.OUT, , );
use MUX2_1 MUX1 (B11.OUT, ETAT[2], `0, `1, BUSY.IN),
      MUX6 (B21.OUT, B22.OUT, RS, `0, R.IN);
use MUX2_3 MUX3 (ETAT[2], ETAT[3], B19.OUT\B18.OUT\B17.OUT, C.IN);
use MUX2_8 MUX4 (ETAT[2], ETAT[4], SHIFT2.OUT[0:6]\A[7], B.OUT, A1);
use MUX2_16 MUX5 (ETAT[5], B21.OUT, `0, A\A1, SM );
use MUX3_8 MUX2 (ETART[2], ETAT[4], B24.OUT, `0, S, SHIFT1.OUT[0:6]\R,
A);
use SHIFT SHIFT1 (A, ),
      SHIFT2 (A1, );
use OR B1 (ETAT[1], ETAT[2], ETAT[3], ETAT[4], ETAT[5], ),
      B6 (ETAT[5], B2.OUT, B4.OUT, ),
      B8 (ETAT[2], B7.OUT, ),
      B12 (B11.OUT, ETAT[2], ),
      B13 (ETAT[2], ),
      B14 (C[0], C[1], C[3], SIG_1),
      B23 (B21.OUT, B22.OUT, ),
      B25 (ETAT[2], ETAT[4], B24.OUT, ),
      B27 (ETAT[2], ETAT[4], );
use NOT B2 (B1.OUT, ),
      B3 (DEPART, ),
      B9 (SIG_1, ),
      B20 (A1[15], ),
      B28 (ETAT[5], );
use AND B4 (ETAT[1], B3.OUT, ),
      B5 (ETAT[1], DEPART, ),
      B7 (SIG_1, ETAT[4], ),
      B10 (B9.OUT, ETAT[4], ),
      B11 (DEPART, ETAT[1], ),
      B15 (RC2, C[2], RC1, ),
      B16 (RC1, C[1], RC0, ),
      B21 (ETAT[3], B20.OUT, ),
      B22 (ETAT[3], A1[15], ),
      B24 (A1[15], ETAT[3], ),
      B26 (ETAT[1], ETAT[2], ),
use XOR B17 (C[2], RC2, ),
      B18 (C[1], RC1, ),
      B19 (C[0], RC0, );
findescription
```

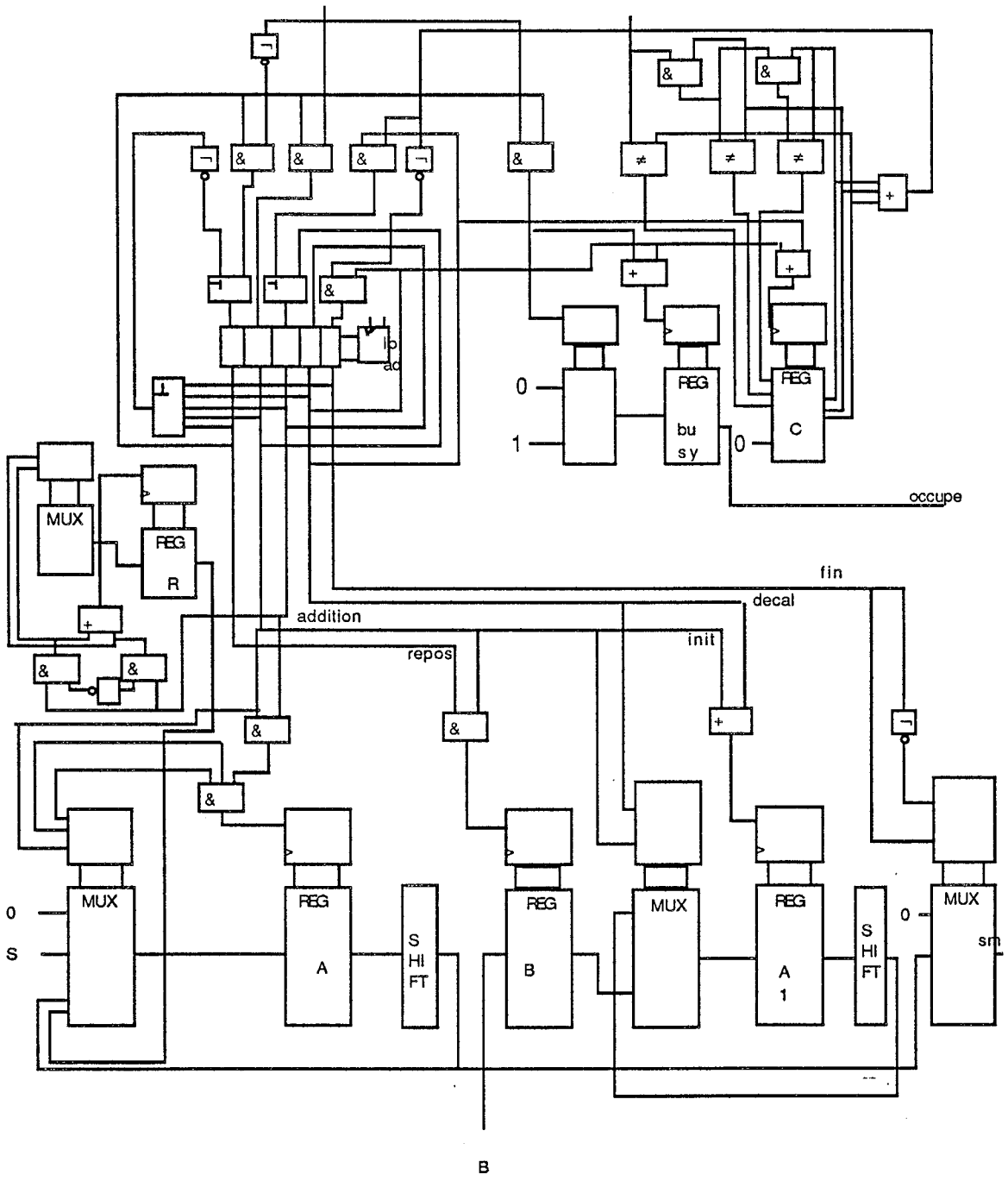
Cet exemple montre bien combien il est difficile de lire une liste d'interconnexions. Néanmoins on peut remarquer:

- le signal de type "HORLOGE" de la description source est devenu un signal de type VARLOG (variable logique) comme les autres.

- Dans les déclarations apparaissent deux signaux nouveaux: ETAT et SIG_1 . Les autres signaux créés par la compilation apparaissent en notation pointée.

- les boîtes OR, NOT et AND n'ont pas besoin d'être déclarées. Ce sont des primitives en POLO.

Le circuit est beaucoup plus lisible de manière graphique. Le dessin manuel de la page suivante donne une idée du circuit.



réalisation par COMPLO de l'automate de MULT

Chapitre 5

SYNTHESE LOGIQUE DE LDS

5 SYNTHESE LOGIQUE DE LDS

5.1 Objectifs de l'outil

Le travail présenté ici a été réalisé dans l'environnement du département DEDM[1] de BULL-SYSTEMES. La vocation de ce département est essentiellement le développement de la famille DPS7 de mini-ordinateurs. Cet environnement conditionne fortement certaines orientations. En particulier, les descriptions de références et les optimisations du paragraphe 5.5 correspondent au développement d'une machine précise. Le langage utilisé est le langage LDS, acronyme de Langage de Description de Systèmes. Mais notre machine de référence a été décrite avec son prédécesseur LDL, et certains choix doivent plus à la méthodologie LDL qu'aux méthodologies utilisées avec LDS.

L'objectif de l'outil est de synthétiser un ensemble de fonctions opérant sur une partie commune du matériel. Chaque fonction est décrite par un module LDS, et les variables (signaux et registres) qui apparaissent dans cette description sont supposées communes. L'outil est donc destiné à une utilisation locale et non à la synthèse globale d'un circuit VLSI entier, ce qui rendrait impossible certaines des optimisations globales, ainsi que le routage automatique final.

La phase de précompilation est assez proche des transformations "data-flow" utilisées ailleurs (voir chapitre 6).Les idées d'implémentation naïve sont classiques ([Dar 81],[Hach 86],[Sai 86]). L'optimisation par transformations locales l'est également, mais la manière d'implémenter les transformations diffère du format "règles de production" que l'on trouve souvent.

5.2 Le langage LDS, son utilisation et sa sémantique

LDS est un langage de haut niveau qui, selon la terminologie de Don Thomas [Tho81], décrit un système au niveau comportemental : le matériel est décrit dans un langage de programmation, et ni les variables ni les

[1] Direction des Etudes et Développements du Matériel.

opérateurs n'ont forcément de correspondance matérielle. Le langage est procédural, mais il permet l'expression d'actions parallèles. LDS supporte deux types de descriptions : la description structurelle,les "SMODULE", exprimant l'arrangement hiérarchique des composants et la description comportementale, avec les "CMODULES", qui décrit le fonctionnement du matériel au moyen d'instructions. Les deux coexistent à l'intérieur d'une structure englobante, le module "COMPONENT", et partagent les mêmes ressources déclarées dans un RMODULE. Dans la suite, nous nous concentrerons sur le format du CMODULE car c'est le format d'entrée du compilateur. Néanmoins, le format des SMODULES est un des formats de sortie possibles du compilateur.

Le langage LDS est utilisé pour différents outils : simulateurs fonctionnels et structurels, extracteurs, générateurs de micro-code. En particulier, LDS sert de langage d'entrée au compilateur de silicium *SYCO*. L'entrée de *SYCO* est le jeu d'instruction d'un microprocesseur, dont il donne une réalisation du type "68000".

LDS est également le langage de spécification pour les concepteurs de Bull-Systèmes. Il sert d'interface entre les concepteurs de micro-fonctions et les concepteurs de hardware. Le comportement de la machine en cours de conception est exprimé micro-fonction par micro-fonction. Les principaux registres ainsi que le codage des commandes sont spécifiés, mais ni les ressources matérielles ni la connectique de l'ensemble ne le sont.

La spécification du compilateur logique décrit ici s'est faite sur la base d'un modèle existant, de la bibliothèque correspondante et d'entretiens avec les logiciens qui ont fait la traduction matérielle de ce modèle. Certaines des orientations prises correspondent à ce choix : l'optique très "standard cell" de la synthèse, l'hypothèse d'un découpage en fonctions du modèle. Par contre, certaines fonctionnalités sont transposables à d'autres environnements : le traitement des instructions, l'optimisation et la composition par règles de réécriture.

5.2.1- Les hypothèses sur la sémantique du langage

Il faut bien distinguer les possibilités du langage et l'utilisation que nous allons en faire. LDS est un langage au large spectre d'application, et la synthèse nécessite une sémantique restreinte. Cette sémantique doit bien sûr correspondre à l'idée que s'en font les concepteurs.

La première question qui se pose est celle du séquencement. Comme LDS est procédural, le matériel sera modélisé par des instructions séquentielles. Il faut donc fixer une convention sur le comportement temporel de deux instructions consécutives. A l'examen du modèle de

référence, on ne trouve pas d'exemple d'instructions où intervienne la relation prédécesseur-successeur : en clair, on ne trouve pas, sous la portée d'une même étiquette, deux instructions où une variable V soit affectée par deux expressions différentes. Le séquencement est explicite dans le modèle : il est exprimé par les étiquettes, les branchements et les délais. On peut donc considérer que toutes les instructions sous portée d'une même étiquette sont exécutées en parallèle. Par contre, il existe un séquencement implicite sur le chargement de registres. Chaque registre est chargé sur une horloge particulière, fixée à la déclaration.

La sémantique des étiquettes est également un problème : on peut considérer que chaque étiquette représente un état d'un automate d'états finis qu'il faut générer, ou bien une phase de l'horloge, ou bien un signal de commande provenant d'un automate externe. Nous prévoyons deux cas : commande externe et automate d'états finis synchronisé sur une seule horloge.

Une autre question importante est la correspondance matérielle des variables utilisées. Les variables de type "registre" correspondent toutes dans l'esprit des concepteurs à un registre matériel. De même, les variables de type "signal" correspondent à une connexion réelle. De plus, une étiquette correspond à un signal portant le même nom.

La correspondance matérielle des opérateurs pose peu de problèmes, dans la mesure où chaque opérateur entraîne la création d'un composant générique qui lui correspond (néanmoins la dimension des signaux de sortie de ce composant dépendra de sa sémantique). Ce composant générique sera traduit à la composition en un composant réel, suivant les règles de réécriture accessibles au concepteur.

Encore une question très importante : quelle est la valeur d'une variable de type "SIGNAL" aux instants où elle n'est pas affectée ? Les signaux en LDS n'ont pas de valeur par défaut : nous avons donc effectué une série de choix correspondants à différents cas de figure.

Enfin, LDS permet d'exprimer un certain parallélisme. Nous ne traduisons que le lancement en parallèle de plusieurs fonctions.

5.3 Un exemple typique de description comportementale

Nous présentons ici un exemple simplifié représentant deux fonctions. Cet exemple montre le style des descriptions sources rencontrées.

```
CMODULE DU_PP1;

<read>
     ADR := DAD / EN_PT_AD;
<exec>
     case (IND_PT)
          when ('01') PT := PAD / ADR;
          when ('11') PT := ADR;
          when ('00') PT := '0';
     endcase;
<report>
     if ( C2 0:2 = CR 0:2 ) PAD := PAD . D;
          else if ( C2 0:2 > INDEX 0:2 )
               PAD := # MA / PAD . D;
               else ERR := MERR / PERR;
               endif
     endif;

end;
```

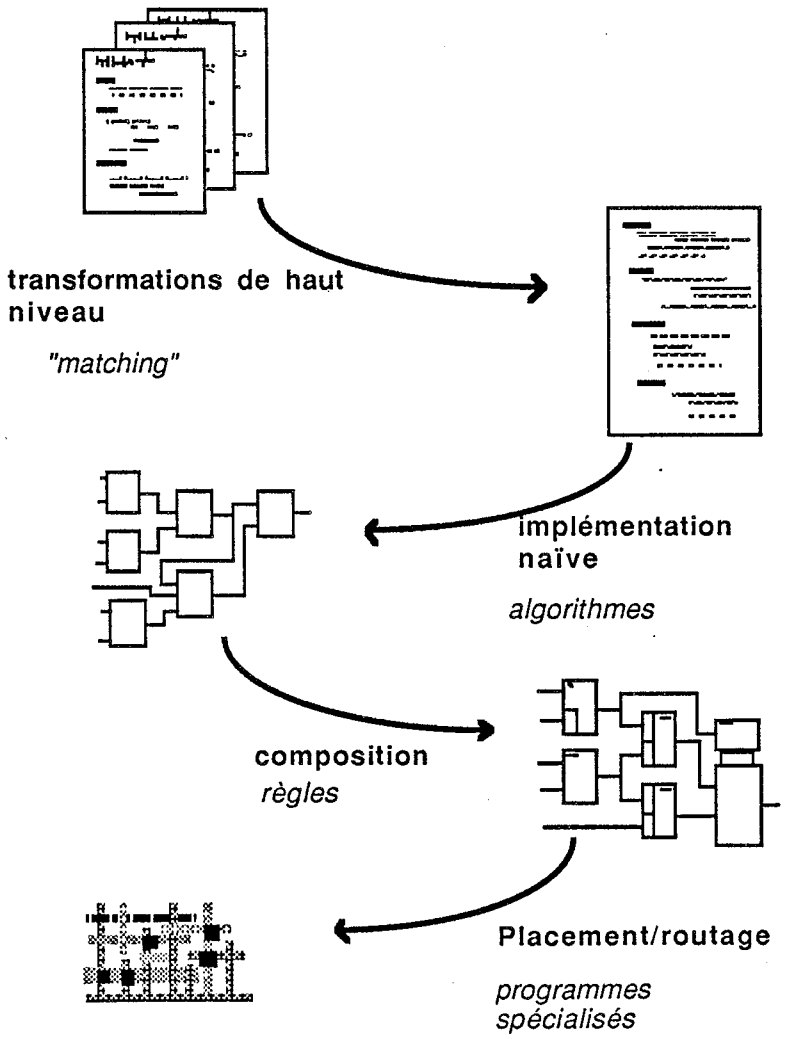
```
CMODULE DU_PP2;

<read>
    ADR :=  EN_PT_AD / DAD;
<exec>
    case (IND_PT)
         when ('10') PT := PAD . ADR;
         when ('01') PT := ADR;
         when ('00') PT := '0';
    endcase;
<report>
    if ( C2 0:2 = CR 0:2 ) PAD := PAD . D;
         else if ( C2 0:2 > INDEX 0:2 )
              PAD := # MA / PAD . D;
              else ERR := MERR / PERR;
              endif
    endif;

end;
```

en LDS, "#" est l'opérateur de négation, "/" le "ou" logique et "." le "et"" logique.

5.4 Stratégie de la synthèse

La conception de circuits peut être vue, d'une façon très descendante, comme une série de choix. Certains de ces choix comportent peu de possibilités et peuvent se traduire par des algorithmes. D'autres en comportent beaucoup, et on peut essayer de les implémenter par des méthodes dites "intelligentes". Notre système utilise alternativement chacune de ces techniques.

■



stratégie de synthèse de LDS

La première phase de la synthèse, "transformations de haut niveau" traite les problèmes dus à la fusion des fonctions et certains problèmes de modélisation : certaines fonctions qui s'écrivent de manière complexe correspondent en réalité à une seule cellule qu'il faut reconnaître très tôt dans la synthèse. Cette phase traite différents cas particuliers auxquels elle applique un traitement simple : elle est réalisée par une série de filtrages simples, appliqués sur une forme arborescente des instructions. La deuxième phase, "implémentation naïve", est la compilation directe de la série d'instructions issue de la première phase. Les termes "compilation directe" et "implémentation naïve" sont utilisés respectivement dans [Dar 81] et dans [Ros 86] pour qualifier cette phase . C'est un traitement très algorithmique . Le résultat est un circuit décrit en LDS structurel, composé d'éléments génériques. La dernière phase, la phase de "composition", traduit "le mieux possible" ces composants génériques en composants spécifiques, en utilisant une base de règles. Le circuit est modifié itérativement par des transformations locales, en respectant certaines contraintes globales. La sortie est une description LDS structurelle, compatible avec différents outils de placement/routage.

5.5 Transformations de haut niveau

Ce module effectue certaines optimisations globales. Tout d'abord les problèmes engendrés par le regroupement des modules comportementaux. On trouve dans un processus analogue dans le système BECOME ([Wei 88]). Le langage d'entrée est un langage procédural proche de C, que les concepteurs utilisent pour la simulation. Avant la synthèse proprement dite, il est nécessaire de transformer le modèle en déterminant les assignements toujours vrais, les "don't cares fonctionnels", ... en un modèle combinatoire.

Les instructions devront être rassemblées état par état, en supprimant celles qui sont dupliquées et en en regroupant certaines dans un case. Certaines séquences d'instructions peuvent également être reconnues à ce stade et être traduites dans le matériel spécifique qu'elles expriment. Enfin, pendant cette phase on peut partager les opérateurs complexes. *Ces méthodes s'appliquent également dans le cas d'un module unique.*

Concrètement, cette phase est une succession de "mises en correspondance"[1]:

---

[1] en anglais "matching".

5.5.1- reconnaissance des instructions dupliquées

Il s'agit de reconnaître les instructions comportant les mêmes variables et les mêmes opérateurs. En effet, un inconvénient de la modélisation avec un langage procédural est que la valeur d'une variable V doit être calculée chaque fois qu'elle est utilisée. En conséquence, on peut retrouver en plusieurs endroits la même instruction intervenant à des instants distincts, même si, au niveau du matériel, cette instruction correspond à la même circuiterie.

Pour ne pas avoir à générer autant de matériel qu'il y a d'exemplaires de l'instruction, on effectue dès le début de la compilation une recherche des instructions dupliquées. Bien que l'exploration combinatoire soit en n! (n étant le nombre total d'instructions considérées) ce processus est peu coûteux puisqu'il s'agit de la mise en correspondance d'arbres dans lesquels les variables sont toutes instanciées.

note : le processus pourrait être rendu plus efficace par une recherche limitée à des opérations ayant un même "genre", et mise sous une même forme canonique : on trouve ce traitement pour l'identification d'instructions semblables pour la synthèse de machines microprogrammées dans [Pol 73].

5.5.2- regroupement des instructions case

La qualité d'un circuit doit beaucoup au choix judicieux de ses multiplexeurs. En général, les concepteurs "expriment" un multiplexage par une ou plusieurs instructions *case* où toutes les sous-instructions sont les affectations d'une même variable. On trouve donc dans les différentes fonctions des instructions "*case*" qui se ressemblent, soit parce qu'elles affectent la même variable, soit qu'elles décrivent les mêmes fonctions, soit qu'elles sont complémentaires . Malheureusement, il est difficile de dégager une méthode simple et systématique donnant une bonne traduction matérielle de ce type d'instructions.

Dans la première phase, nous traitons quelques modèles typiques. Les instructions *case* qui ne correspondent à aucun de ces modèles seront traitées de manière naïve dans la phase suivante, comme une série de *if...then*. Les critères qui définissent nos modèles sont les suivants :

```
case (ctrl)
     when (val i) : Ei := expression i;
     ...
endcase;
```

- égalité de leurs variables de contrôle *ctrl*

- liste des valeurs (*val i*) : on s'intéresse essentiellement aux vecteurs binaires de largeur 2 où 3.

- égalité de la variable assignée *Ei*;

- superposabilité ou non des couples (*val i, expression i*).

- largeur des données traitées. On privilégie le multiplexage du contrôle lorsque les données ne sont pas scalaires.

Le traitement se concrétisera par une recherche des arbres unifiables entre eux, selon des modèles définis. Nous illustrons par exemple le regroupement de deux instructions répondant au schéma suivant :

```
ctrl 1 ≠ ctrl2 , E1 ≠ E2 , {(val1 i, exp2 i)}  ≈ {(val2 i, exp2 i)}
```

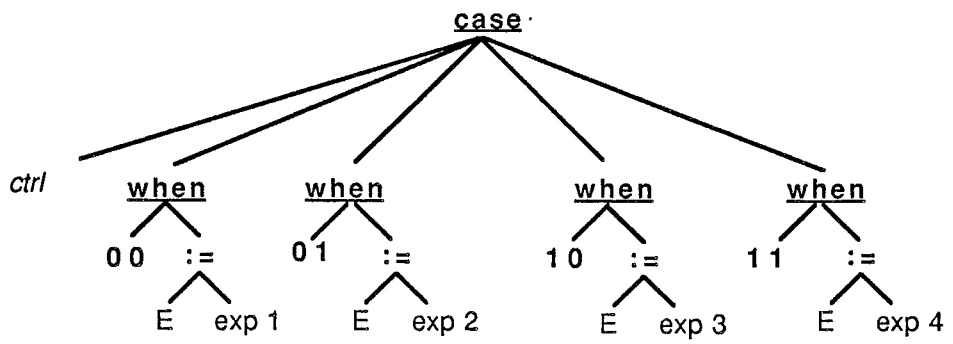
Ce cas est le plus complexe : les variables de contrôle sont différentes, les variables affectées sont différentes mais les couples (valeurs, instructions) sont superposables. Nous ne traitons pas encore le cas où ils ne le sont pas.

```
<mud>
case (arg)
     when ('01') IW := PAC 0:5;
     when ('11') IW := 1\#TLV\TLV\'00';
     when ('00') IW := 1\#TLV\RDBAL 37:2;
endcase;
```

```
<smd>
case (adcode)
    when ('01') IL := PAC 0:5;
    when ('10') IL := 1\#TLV\TLV\'00';
    when ('00') IL := 1\#TLV\RDBAL 37:2;
endcase;
```

(en LDS, \ est l'opérateur de concaténation)

Les deux instructions sont unifiables avec l'arbre correspondant au multiplexeur à quatre entrées génériques :



De plus elles sont unifiables entre elles en substituant (IW, IL) et (arg,adcode).

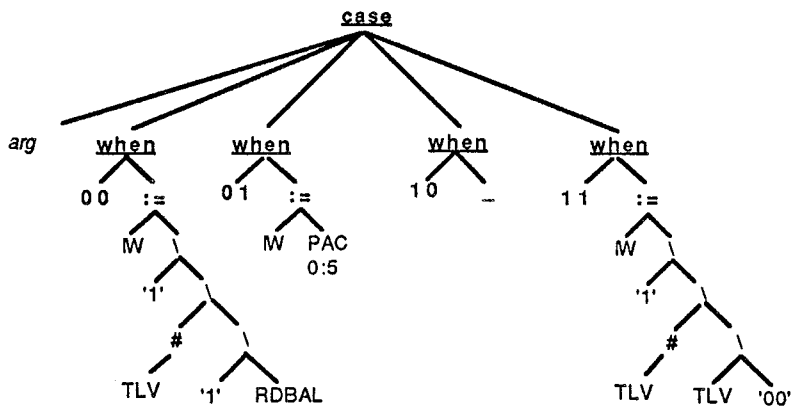


fig. arbre correspondant à <mud>

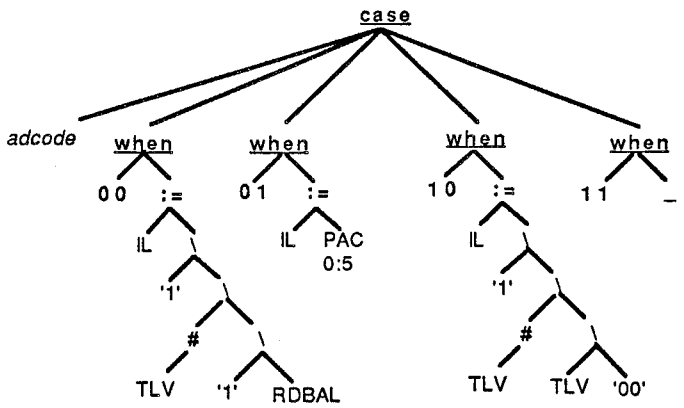
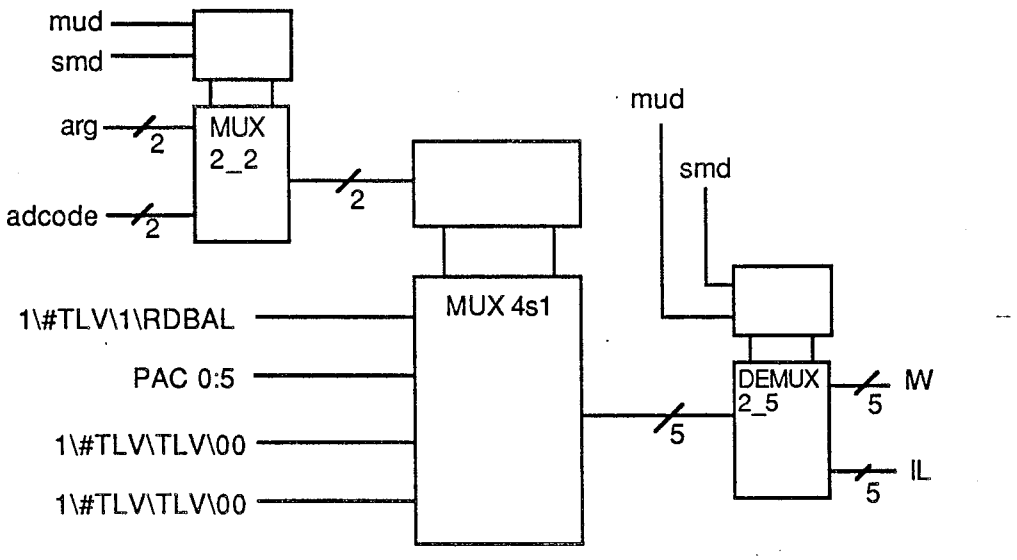


fig. arbre correspondant à <smd>

On traduira l'arbre par un multiplexeur à quatre entrées, dont le contrôle sera également multiplexé par un aiguilleur de données 2 entrées, et dont la sortie sera démultiplexée.

On crée donc le matériel suivant :



En réalité on remplace les deux arbres *case* par un arbre spécial *mux* qui sera traité dans la phase d'implémentation naïve.

5.5.3- reconnaissance de séquences d'instructions spécifiques

Le but de cette phase est de reconnaître certaines instructions typiques dans un projet donné. Il s'agit là d'une vraie "mise en correspondance", par opposition au filtrage simple du 1/, puisqu'il n'y a pas simplement mise en correspondance des opérateurs mais une substitution avec les variables du modèle. Un des problèmes est de tirer avantage de la commutativité et de l'associativité des opérateurs. Malheureusement, les problèmes de "mises en correspondance" commutatives et de "mises en correspondance" associatives sont NP-complets ([Ben 87]). Nous utilisons donc une méthode partielle pour réduire le nombre de possibilités à explorer : les arbres syntaxiques sont ordonnés en fonction de la longueur de leurs branches.

5.5.4- partage des opérateurs complexes

Certains opérateurs complexes, comme par exemple les additionneurs, coûtent cher en place. On a donc intérêt à regrouper les opérations similaires sur un même opérateur, en multiplexant entrées et sorties. Il faut d'abord déterminer si deux opérations peuvent se produire en même temps. Si ce n'est pas le cas, on les dit compatibles et elles peuvent partager un même opérateur. Au niveau de la description source, deux instructions seront compatibles si les opérations qu'elles modélisent sont compatibles. Ce sera le cas par exemple pour deux branches distinctes d'une même instruction "case", ou bien pour deux instructions apparaissant dans la même fonction sous deux étiquettes différentes.

On construit un graphe dont les nœuds sont les opérations et dont les arcs représentent la compatibilité de deux opérations qu'ils joignent. On partitionne en cliques disjointes ce graphe, et chaque clique représente les opérations qui peuvent partager un même opérateur. On peut également colorer le graphe dual, ce qui revient exactement au même. Nous avons utilisé l'algorithme décrit dans [Tse 83]. Cet algorithme permet d'affecter des poids aux arcs et de privilégier ainsi certains regroupements de nœuds. En affectant aux arcs des poids d'autant plus élevés que leurs opérations se ressemblent (1 ou plusieurs opérandes communs, mêmes dimensions des opérandes, même état de l'automate...), on arrive à des regroupements efficaces. Pour plus de détails, nous renvoyons le lecteur à [Tse 83]. La méthode a été testée sur des exemples indépendamment du système, il reste à l'intégrer.

5.6 Le module d'allocation

On traduit les instructions de la description source en un circuit composé d'éléments génériques. Cette opération est très algorithmique, elle est donc programmée de manière algorithmique. Le processus se passe en 5 phases :

a/ traduction des expressions,

b/ traduction des conditions,

c/ génération (optionnelle) de l'automate,

d/ regroupement des affectations et

e/ suppression des éléments dupliqués.

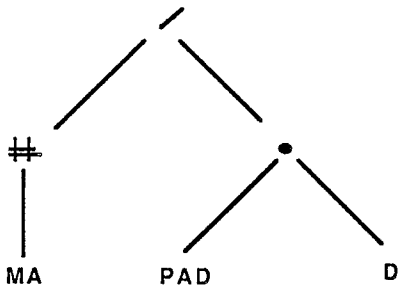
Nous allons illustrer cette phase sur une instruction :

```
if ( C2 0:2 = CR 0:2 )
                PAD := PAD . D;
   else if ( C2 0:2 > INDEX 0:2 ) PAD := # MA / PAD . D;
          else ERR := MERR / PERR;
        endif
endif
```

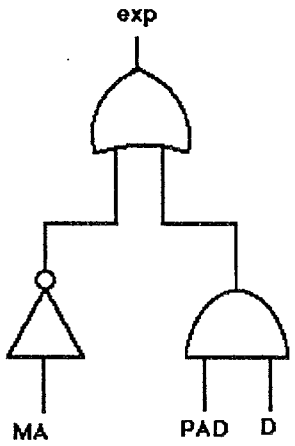
Les phases a/ et b/, i. e. la compilation des expressions et la compilation des conditions, sont traitées simultanément.

5.6.1- compilation des expressions

La compilation des expressions consiste à créer le matériel nécessaire pour réaliser les opérations apparaissant en partie droite des instructions d'affectation. A chacune de ces expressions correspondra un signal. Par exemple l'expression `# MA / PAD . D`, correspondant à l'arbre syntaxique suivant :



sera traduite d'une manière très simple



Les expressions traitées ne sont pas toutes aussi simples ! Les composants génériques utilisés sont : additionneurs de largeur n, multiplexeurs (aiguilleurs de données) n-en-1 de largeur m, décodeurs n en 2**n, comparateurs n bits, portes logiques de largeur n, tampon d'entrée/sortie génériques.

Si l'expression rencontrée est une constante, il y a deux cas :

- les constantes '1' et '0' sont remplacées par les signaux "vcc" et "vdd".

- les autres par une boîte spécifique de la constante rencontrée. Cette boîte générique sera traduite pendant l'optimisation.

5.6.2- compilation des conditions

La plupart des instructions sont des instructions conditionnées, c'est à dire des affectations sous portée de condition. Cette condition découle soit de l'état d'automate sous lequel l'instruction se place, soit de la combinaison des "if...then...else" ou des "case" qui précèdent l'instruction. La prise en compte se fait par un mécanisme de pile, dans laquelle sont stockées les conditions rencontrées avant l'instruction.

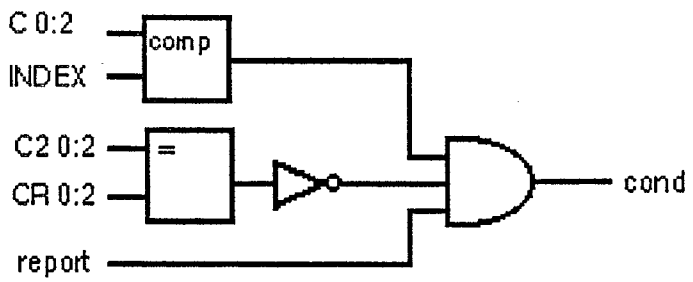
Par exemple, à l'évaluation de l'instruction

```
PAD  := # MA / PAD . D ;
```

la pile contient :

C 0:2 > INDEX 0:2
(C2 0:2 = CR 0:2)
report

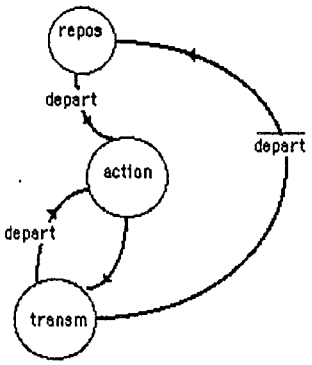
Cette pile est traduite par le matériel suivant



5.6.3- génération des automates

La phase c/ effectue la génération des automates décrits explicitement, c'est-à-dire transforme les instructions de branchement et les étiquettes en un automate d'états finis. La méthode est simple : les n états sont codés sur log2(n) registres de manière aléatoire. Les fonctions d'excitation des registres sont déduites des instructions de chargement et traduites par des affectations. Les fonctions de décodage sont créées afin qu'à chaque état de l'automate corresponde un signal portant le même nom. Les automates que nous générons sont synchronisés sur une horloge unique, qui peut être différente de celle des transferts de la partie opérative.

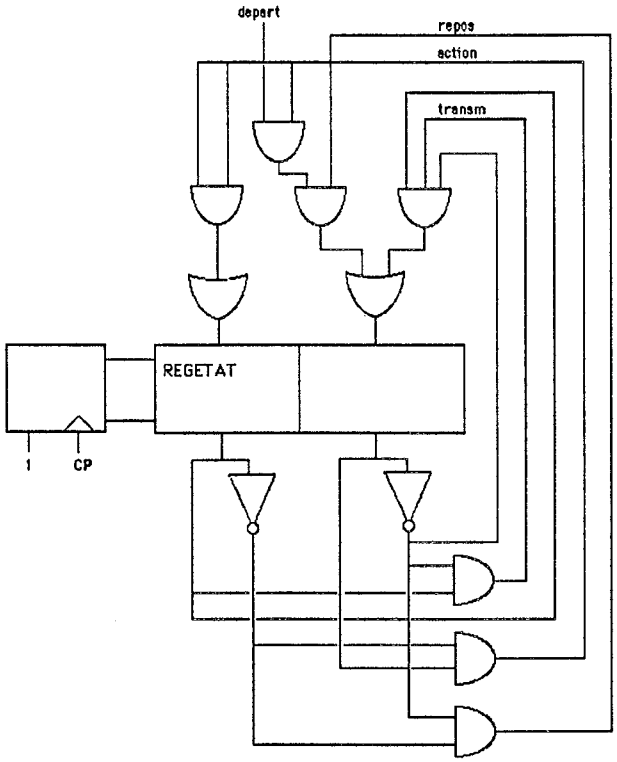
Nous illustrons la méthode sur l'exemple suivant :



En LDS on traduira cet exemple comme suit :

```
CMODULE auto;
<repos> if (depart)  goto action; end;
<action> goto transm;
<transm> if (#depart) goto repos ;
       else goto action; end;
end;
```

Ce code LDS sera traduit de la manière suivante :



exemple d'automate créé par le compilateur de LDS

Il y a beaucoup de maladresse dans cet automate : une porte OR à une seule entrée, une porte AND dont les deux entrées sont identiques, une porte AND redondante. Ces défauts seront corrigés pendant la phase d'optimisation.

traduction des instructions de synchronisation

L'instruction LDS :

```
waitclk <horloge> ;
```

provoque une attente du front montant de l'horloge *<horloge>*. Elle est traduite par l'introduction d'un état supplémentaire dans l'automate. La transition vers cet état est activée lorsque le signal *<horloge>* est à l'état haut.

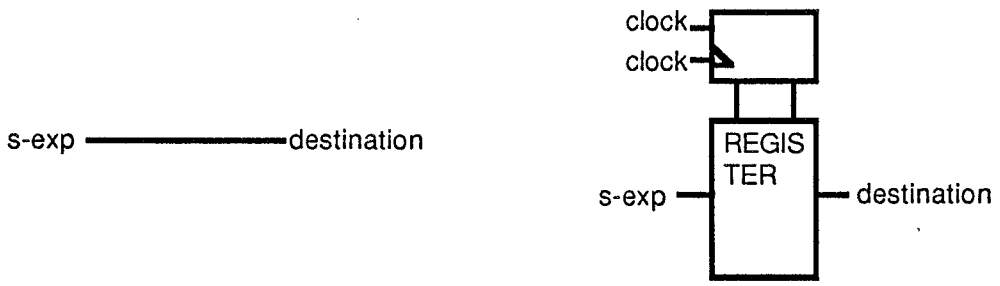
5.6.4- regroupement des affectations

La phase d/ est le regroupement des affectations[1]. LDS est un langage d'affectations et toutes les instructions sont de la forme :

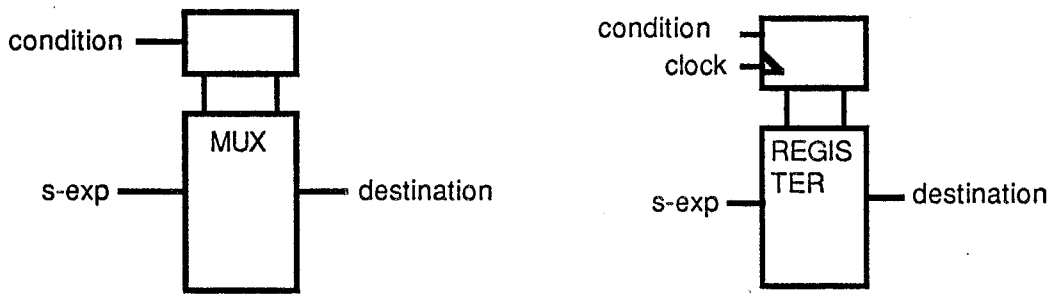
```
[ si <condition> alors ]  <recepteur> := <signal expression> ;
```

Les phases a/, b/ et c/ ont créé des signaux <condition> et <signal expression>, il reste à les connecter au récepteur. Il faut distinguer plusieurs cas :

Le cas des affectations non conditionnées est le plus simple :



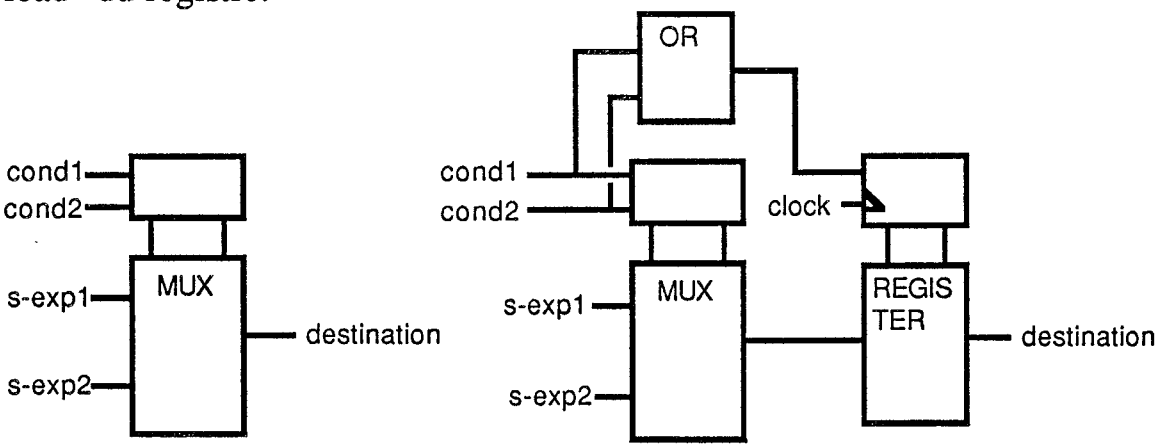
Dans le cas d'une affectation simple conditionnée on créera:



Le cas le plus fréquent est celui des affectations multiples : une même variable, représentant un signal ou bien un registre, est affectée par

[1] Cette phase est rigoureusement identique à la phase équivalente dans COMPLO.

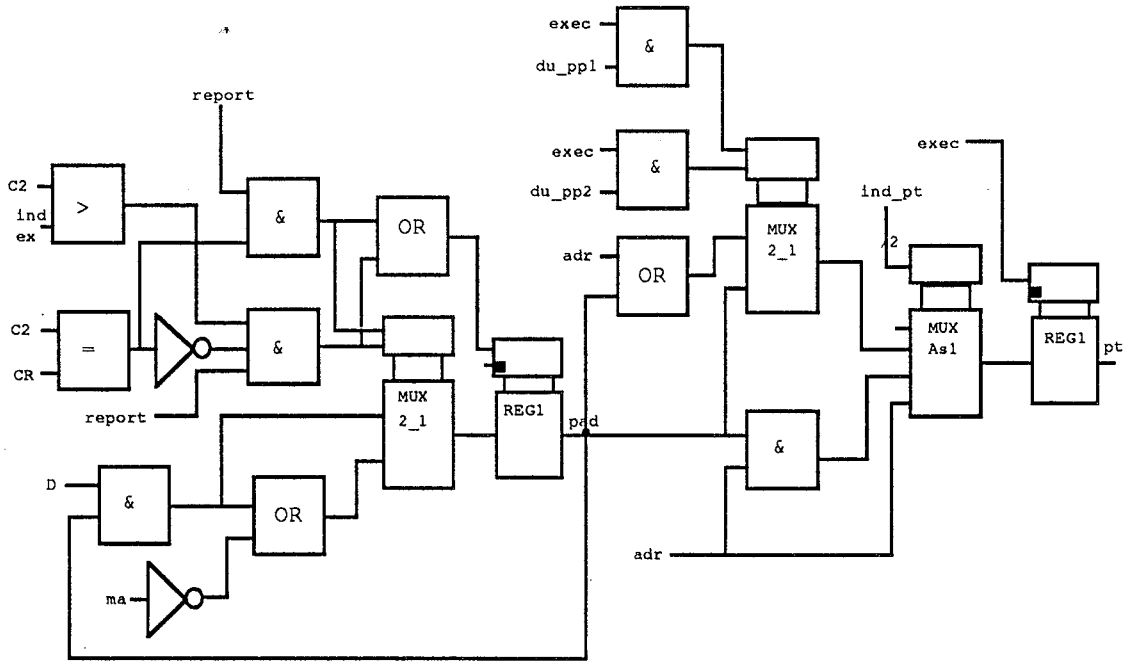
différentes expressions sous différentes conditions. Dans ce cas on utilise un aiguilleur de données, baptisé abusivement "multiplexeur" pour organiser le cheminement des bons signaux expressions à la destination quand les signaux conditions sont à l'état haut. Dans le cas du chargement d'un registre, on crée en plus un "ou" des différentes conditions pour l'entrée "load" du registre.



5.6.5- élimination des éléments dupliqués

Les phases a/, b/, c/ et d/ génèrent du matériel de manière très systématique et donc peu optimale. On trouve beaucoup de redondance dans le matériel. La phase e/ sert à éliminer tous les éléments dupliqués.

La compilation naïve des fonctions DU_PP1 et DU_PP2 décrites plus haut donne le résultat suivant :



Cette implémentation, simple et systématique, n'est valable que si la description source est correcte. Par exemple, toutes les conditions de chargement d'un même registre doivent être disjointes.

5.7 La composition

Nous regroupons dans cette phase l'optimisation du résultat de l'implémentation naïve et le choix des composants finals.

Dans les bibliothèques Standard Cell on trouve souvent des composants plus complexes que les simples portes combinatoires AND OR NAND ... C'est le cas chez Bull-Systèmes. C'est pourquoi nous avons choisi d'effectuer la composition au moyen de transformations locales spécifiques plutôt que d'utiliser les méthodes d'optimisation booléennes classiques. La synthèse par transformations locales consiste :

1/ à identifier des combinaisons données de composants et

2/ à les remplacer par d'autres combinaisons.

Mais il y beaucoup de combinaisons à identifier, et encore plus de manières de les remplacer. Il est donc nécessaire de définir une stratégie pour la composition.

5.7.1- Une solution optimale, la recherche exhaustive...

Nous avons utilisé un programme de recherche exhaustif pour explorer presque toutes les possibilités d'implémentation à partir de notre bibliothèque d'un circuit réduit (le multiplieur, voir § 5.11) en composants génériques. Cette recherche avait des contraintes : chaque combinaison qui dépasse la surface minimum atteinte jusque-là est écartée, les longueurs de chaînes sont limitées, etc... Même avec ces contraintes, le programme de recherche exhaustif n'est pas utilisable sur des exemples en vraie grandeur : le nombre de choix possibles est beaucoup trop important. Par exemple, rien que pour l'expression suivante :

```
[(t . u) / (f . g)] . [¬( r | u) . b]
```

on trouve 20 solutions acceptables avec la bibliothèque ARES.

note : Notre manière de réduire le nombre des combinaisons à examiner rappelle la méthode "branch and bound"[1]. Le critère d'évaluation qu'elle utilise est le calcul de la surface effective de la combinaison : aucune évaluation n'est suspendue. On peut imaginer un critère d'évaluation plus efficace au point de vue temps de calcul, mais ce critère dépendra forcément très étroitement de la bibliothèque considérée.

5.7.2- ...et la solution pratique : les règles de réécriture.

La recherche exhaustive a fait apparaître que pour notre bibliothèque de référence, les transformations "payantes" (qui économisent du silicium)

---

[1] en français "séparation et évaluation progressive".

se retrouvent souvent. En pratique, on a intérêt à garder des expressions booléennes en AND/NAND/NOT comportant 2 à 6 opérateurs. Nous avons traduit ces transformations en règles de réécriture, auxquelles sont affectées des poids. Les poids permettent au système de choisir quelle règle appliquer quand plusieurs transformations sont possibles. A la différence du système *SOCRATES* ([Hach 86]) où les règles sont ordonnées dynamiquement, les poids déterminent un ordre statique pour l'application des règles.

On ne trouvera pas d'optimisation booléenne au sens habituel du terme dans nos règles. Par contre, pour une même cellule, on retrouvera différentes écritures booléennes de sa fonction.

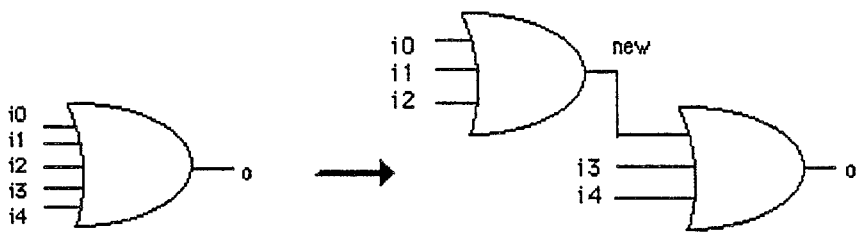
Le principe du traitement est le suivant : on applique les règles dans l'ordre des poids décroissants et dans la mesure où elles satisfont les contraintes suivantes :

- Les éléments dont la sortance est supérieure à 2 ne doivent pas être supprimés.

- La longueur des chaînes entre une interface d'entrée et une interface de sortie doit rester inférieure à une limite fixée.

Les règles de réécriture sont de deux types : règles d'expansion et règles de réduction.

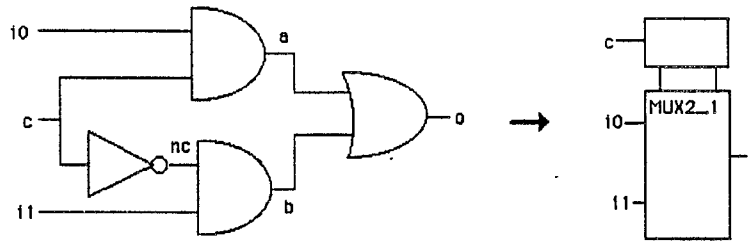
Les règles d'expansion augmentent localement le nombre de composants. Par exemple, la règle suivante éclate les OR à 5 entrées en OR à 3 entrées, plus avantageux pour les transformations ultérieures.



Dans notre syntaxe, cette transformation s'exprime par la règle suivante :

```
 poids 1
or([i0,i1,i2,i3,i4],o) -> or([i0,i1,i2],new) & or([new,i3,i4],o);
```

Les règles de réduction diminuent localement le nombre de composants. Par exemple :



Cette règle s'écrit :

```
mux2_1([c,i0,i1],o) <- or([a,b],o)
                     & and([c,i0],a)
                     & and([nc,i1],b)
                     & not([c],nc);
```

On peut définir des transformations locales de plus haut niveau, par exemple la suppression des inverseurs redondants :

```
_b([i;j],o) <- nr1([o2],o) & nr1([o1],o2) & _b([i;j],o1);
```

Grossièrement, les règles sont organisées dans l'ordre suivant (poids décroissants) :

- suppression des "effets de bord" dus aux algorithmes de l'implémentation naïve : signaux identiques en entrée des portes combinatoires, portes n'ayant qu'un signal en entrée...

- regroupement des AND/OR 2 entrées en AND/OR 3 entrées.(transformation recommandée par J. Darringer).

- éclatement des AND/OR de plus de 3 entrées en AND/OR 3 entrées.

- regroupements complexes,

- règles concernant les multiplexeurs,

- regroupement complexes de moindre intérêt,

- regroupements dans lesquels interviennent des cellules déjà définies,

- choix des registres et des tampons,

- regroupements simples,

- transformations "1 à 1" des AND/OR/... qui restent,

- élimination des inverseurs redondants.

Ces règles sont analysées et produisent un programme Prolog capable de transformer une description LDS structurelle, et de donner le circuit réécrit en LDS structurel. Certaines des règles sont néanmoins prédéfinies, comme par exemple la contrainte sur les longueurs de chaînes.

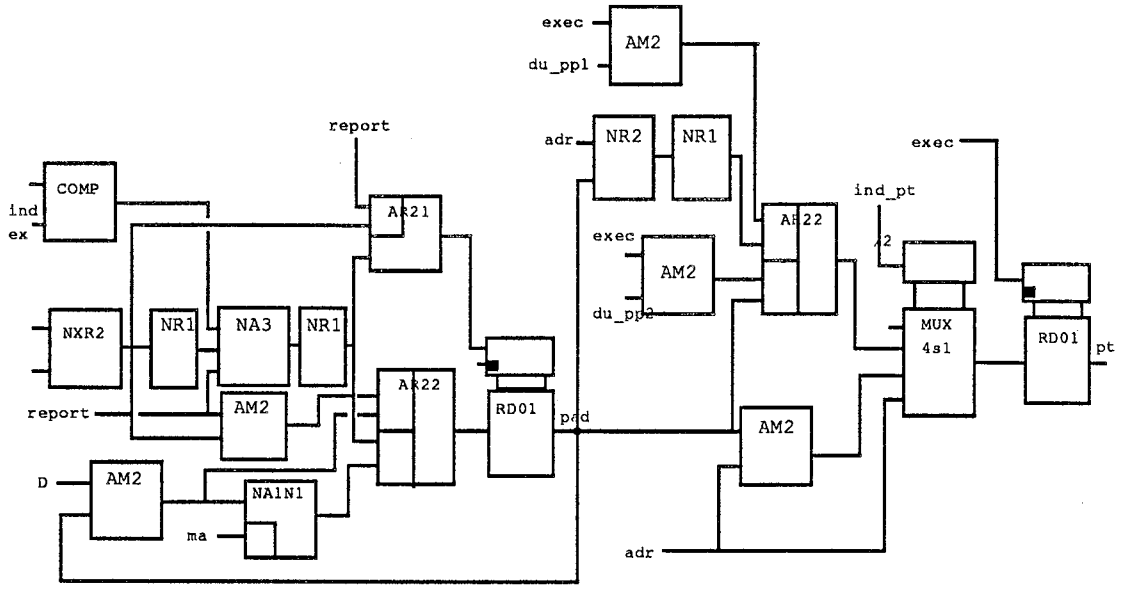
Pour reprendre la trilogie décisions techniques/ planification/ acceptabilité :

- Les décisions techniques sont implémentées par les règles de réécriture.

- La planification est représentée par les poids attachés aux règles.

- L'acceptabilité est contenue dans les contraintes sur les longueurs de chaînes, dans la méthode de suppression d'un composant.

La composition de notre exemple nous donne le résultat suivant :



résultat de l'optimisation de l'exemple

5.8 Connexion avec les outils de placement-routage

La sortie du module de composition est formatée en LDS structurel (comme d'ailleurs la sortie du module d'implémentation naïve). Ce format est commun à de nombreux outils, en particulier il peut servir d'entrée aux outils de placement-routage de Bull-Systèmes : BALI, THALES . De plus, il existe un pont entre LDS structurel et le format "hns" des outils VTI, ce qui permet d'utiliser les outils de placement-routage de VTI (Chipcompiler).

5.9 Correction des circuits obtenus

A la différence de COMPLO, le compilateur logique de LDS "crée" un peu de logique de séquencement :

-L'instruction de synchronisation "waitclk" est traduite par une synchronisation sur le premier front montant de l'horloge de l'automate qui suit le front montant de l'horloge spécifiée. En effet l'instruction "waitclk" entraîne la génération d'un séquencement, et

ce séquencement prend comme horloge de base l'horloge de l'automate.

- L'utilisation détournée des étiquettes comme séparateur entre états peut entraîner une confusion chez l'utilisateur.

- L'implémentation des conditions est juste dans tous les cas, mais assez lourde. Il vaut mieux éviter les expressions du type :

```
if (sig=1) ...
```

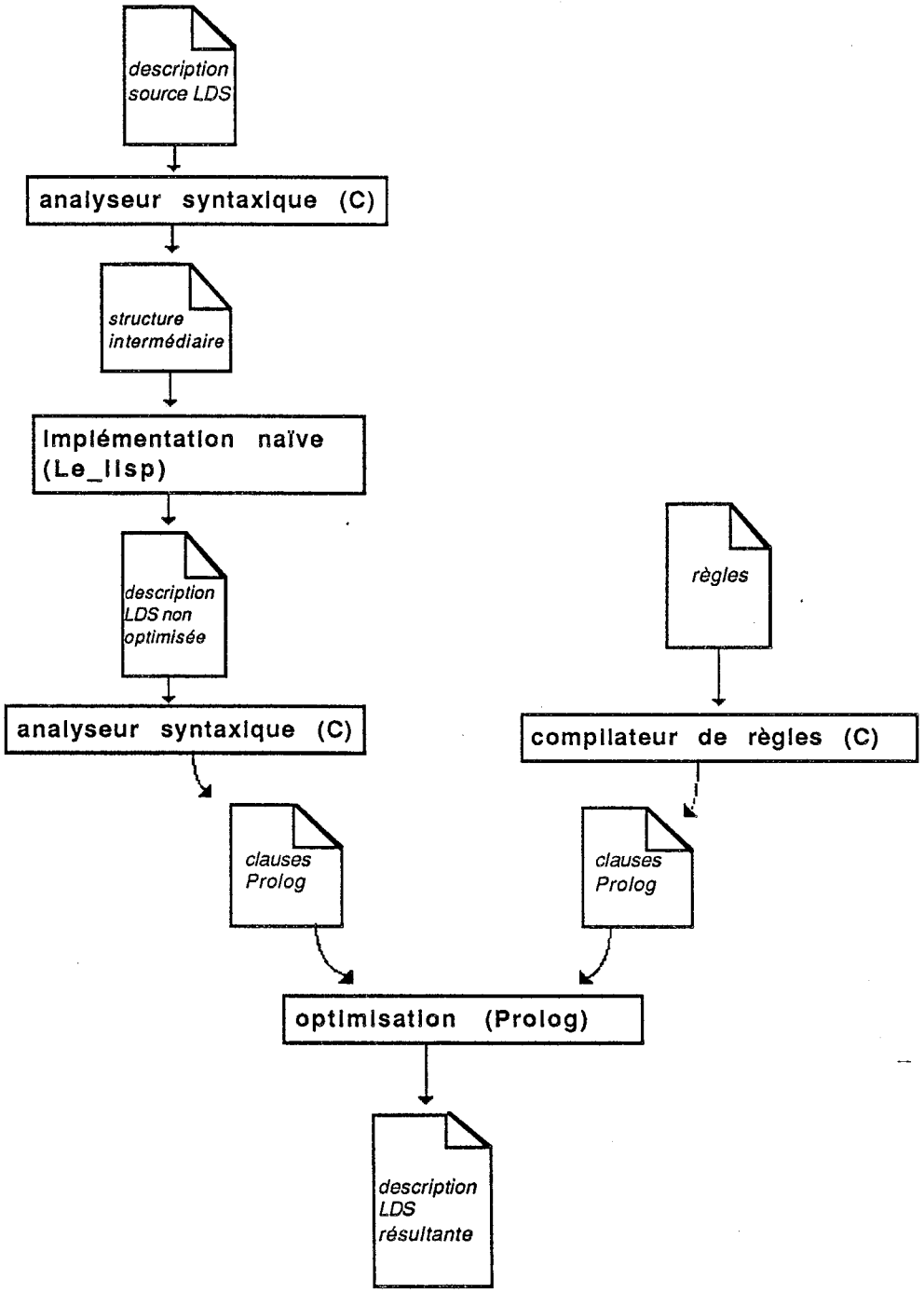
qui vont créer un opérateur de comparaison (ici, une porte xnor) et les remplacer par

```
if (sig) ...
```

En bref, les problèmes de correction proviennent du fait que nous avons été obligé de choisir des conventions d'utilisation de LDS. Si l'utilisateur du compilateur en utilise d'autres, il est normal que le résultat ne corresponde pas à son attente.

5.10 Notes d'implémentation

5.10.1- structure de la maquette



Organisation du compilateur de LDS

5.10.2- limitations de la maquette

Comme le remarque Kahrs dans [Kah 86], le "tree matching" est une forme restreinte du "graph matching" qui donne des solutions efficaces, mais restreint beaucoup l'exploration des différentes solutions.

Les dimensions sont considérées scalaires par défaut, l'optimisation ne se fait que sur des scalaires.

La maquette est programmée en Le_Lisp (dialecte de Lisp de l'INRIA). La compilation est effectuée par l'évaluation du circuit entré sous forme arborescente. Les structures utilisées pour les données sont très proches des listes décrites dans le chapitre précédent. Cette façon de programmer s'est révélée très confortable et facile à modifier. Les performances de l'interprète Le_Lisp en vitesse d'exécution ont toujours été très suffisantes pour les exemples traités.

Par contre, l'analyseur syntaxique écrit en Ceyx, trop difficile à modifier, a du être abandonné pour un analyseur généré par Lex et Yacc.

L'analyseur de règles de réécriture a également été programmé en C avec Lex et Yacc.

La maquette d'optimisation doit tourner en SP-Prolog (compilateur Prolog de Bull-MTS). Malheureusement, pour des raisons de versions, il tourne actuellement sous une forme réduite avec un interprète Prolog.

5.11 Un exemple : synthèse d'un multiplieur

Le multiplieur décrit ici est identique à celui du chapitre précédent. Il y a néanmoins deux différences notables :

- Pour simplifier le modèle, nous n'avons pas séparé l'additionneur du reste de la description. Il est ici modélisé par une instruction d'addition.

- Le modèle implique que certaines équations soient vraies en permanence : c'est le cas par exemple de l'instruction d'addition, de la circuiterie de comptage à 8. Comment les représenter dans un langage procédural ? Nous résolvons le problème en faisant évoluer en parallèle deux CMODULES : l'un représentant ces instructions, qui est réévalué à chaque coup d'horloge (CMODULE "toujours"), et l'autre représentant le reste de la description (CMODULE "mult").

5.11.1- la description source

```
? MULTIPLIEUR 8 x 8 BITS

GMODULE gmult ;
clock;
 H, OFFSET 0, UP, LAPSE 1, PERIOD 2;
end;
END;

GMODULE bedouin;
? variables globales
signal;
 sync=H;
 em 0:8; depart; re;
 sm 0:16; occupe;
 RC0; RC1; RC2;
 s 0:8; rs;
end;
END;


SMODULE smult (em,depart,re,sm,occupe);
signal;
 em 0:8, in; depart,in; re,in;
 sm 0:16,out; occupe,out;
 RC0; RC1; RC2;
 s 0:8; rs; ? sortie et retenue de l'additionneur
end;
 link cmult, rmult;
END;

RMODULE rmult;
? registres du multiplieur
register;
 sync = H;
 busyreg; areg 0:8; breg 0:8;
 creg 0:3;
 alreg 8:8; ?sert au calcul
 rreg ;
end;
END;
```

```
CMODULE toujours;
? module qui sert a calculer a chaque coup
? d'horloge ce qui est non-procedural
context rmodule rmult;
 <dummy>
 waitclk H;
 re:='0'; ? provisoire qui risque fort de durer (comme un turc)
 RC2:='1';
 RC1:=RC2 . creg 2;  ? circuiterie de
 RC0:=RC1 . creg 1;  ? comptage a 8
 S := areg + breg + re;
 rs:=carry(areg,breg,re);
 goto dummy;
END;
CMODULE cmult;
context rmodule rmult;
<dummy>
 waitclk H;     ? ca ne peut pas faire de mal
 cause toujours; ? very funny
<repos>
 waitclk H ; ?*********************************
 breg:=em;   ? chargt du 1er operande
 busyreg:='0';
 if (depart='1') goto init; else goto repos; end;
<init>
 waitclk H ; ?*********************************
 a1reg:=breg; ? 1er operande dans a1
 breg:=em;    ? 2eme operande dans breg
 areg:='0';
 busyreg:='1';
 goto addition;
<addition>
 waitclk H ; ?*********************************
 if (a1reg 15='1') areg :=s; rreg:=rs;
  else rreg:='0';
 end;
 creg 0:1 := RC0%creg 0:1;
 creg 1:1 := RC1%creg 1:1;
 creg 2:1 := RC2%creg 2:1;
 goto decal;
<decal>
 waitclk H ; ?*********************************
 areg:=rreg!areg 0:7;
 a1reg:=areg 7:1 ! a1reg 8:7;
 if (creg 0/creg 1/creg 2)
  goto addition;
  else goto fin ;
```

```
 end;
<fin>
 waitclk H ; ?*********************************
 sm:=areg!alreg;
 goto repos;

END;
```

5.11.2- Le résultat

La sortie de l'implémentation naïve se fait en LDS structurel.

fichier de sortie

```
component exnihilo
interface
  vcc 0:1 , in;
  vdd 0:1 , in;
  em 0:8 , in;
  depart 0:1 , in;
  re 0:1 , in;
  sm 0:16 , out;
  occupe 0:1 , out;
end;
end exnihilo;

smodule of exnihilo

signal
  b_depart 0:1;
  sig1 0:1;
  sig2 0:1;
  sig3 0:1;
  a1reg 15:1;
  sig4 0:1;
  sig5 0:1;
  sig6 0:1;
  rc0 0:1;
  sig7 0:1;
  rc1 0:1;
  sig8 0:1;
  rc2 0:1;
  sig9 0:1;
  rreg 0:1;
  sig10 0:8;
  areg 7:1;
  sig11 0:8;
  creg0 0:1;
  sig12 0:1;
  sig13 0:1;
  sig14 0:1;
  sig15 0:1;
  sig16 0:1;
  areg 0:8;
  b_sm 0:16;
```

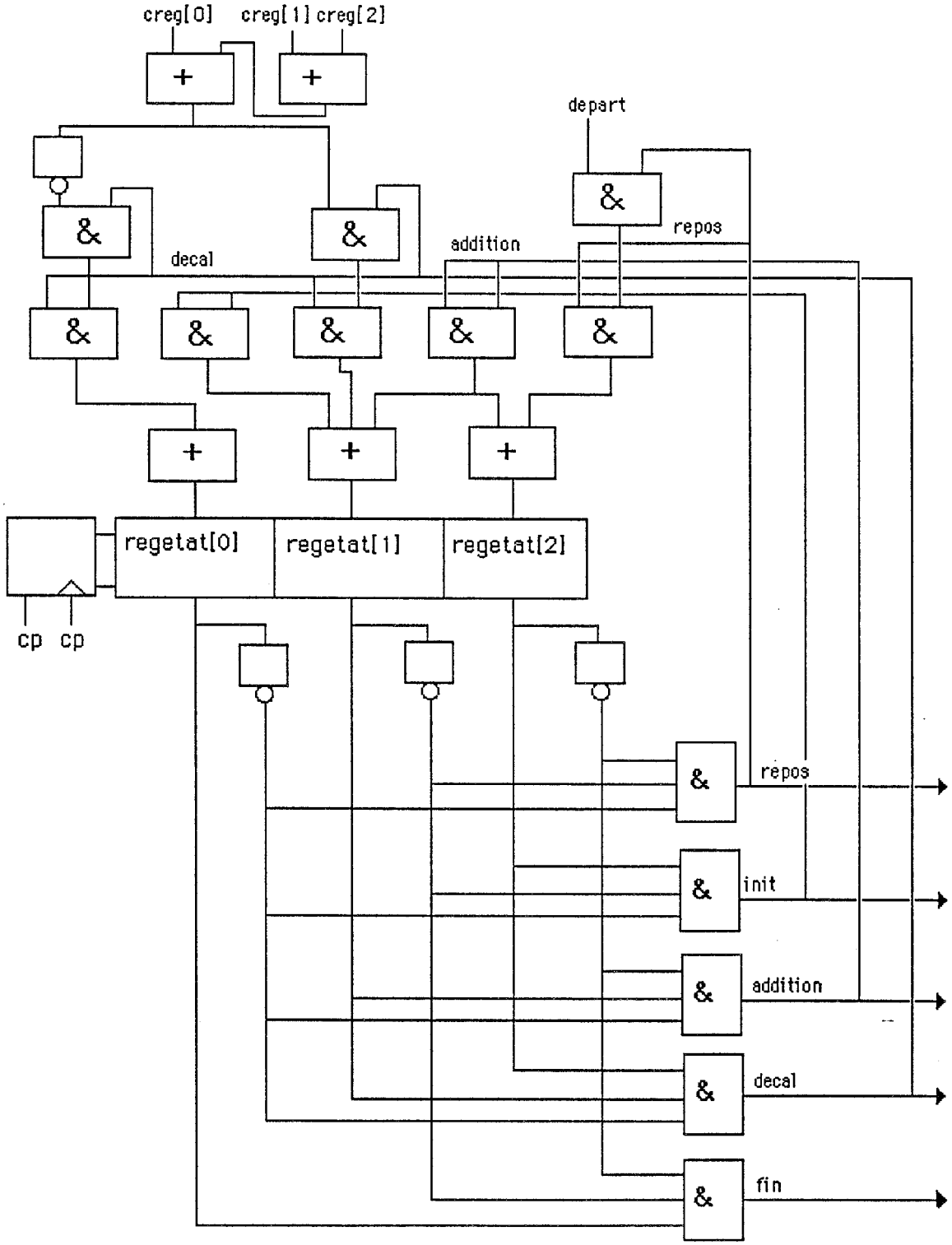
```
  regetat 0:1;
  sig18 0:1;
  regetat 1:1;
  sig19 0:1;
  regetat 2:1;
  sig20 0:1;
  dummy 0:1;
  repos 0:1;
  init 0:1;
  addition 0:1;
  decal 0:1;
  fin 0:1;
  sig35 0:0;
  sig36 0:0;
  sig37 0:0;
  sig39 0:0;
  sig40 0:0;
  sig42 0:0;
  sig44 0:0;
  sig45 0:1;
  sig46 0:1;
  sig47 0:1;
  sig48 0:8;
  sig49 0:1;
  cp 0:1;
  breg 0:8;
  sig50 0:1;
  busyreg 0:1;
  alreg 0:1;
  sig53 0:1;
  sig54 0:1;
  sig55 0:1;
  sig56 0:1;
  creg 0:1;
  creg 1:1;
  creg 2:1;
  b_vcc 0:1;
  b_vdd 0:1;
  b_em 0:8;
  b_re 0:1;
  b_occupe 0:1;
end;

connect

  box1:and (b_depart 0:1, repos 0:1, sig1 0:1);
  box2:not (b_depart 0:1, sig2 0:1);
```

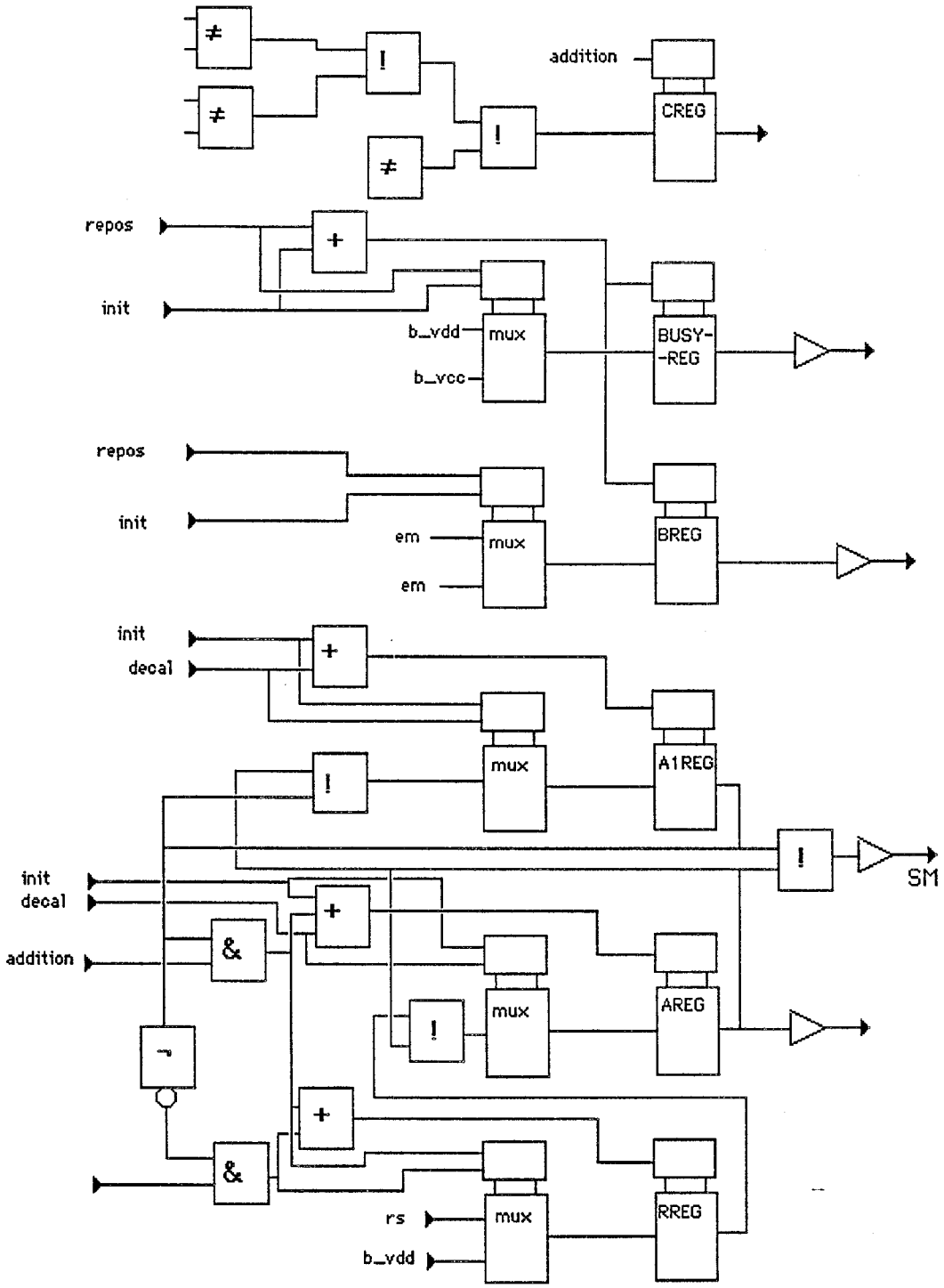
```
  box3:and (sig2 0:1, repos 0:1, sig3 0:1);
  box4:and (a1reg 15:1, addition 0:1, sig4 0:1);
  box5:not (a1reg 15:1, sig5 0:1);
  box6:and (sig5 0:1, addition 0:1, sig6 0:1);
  box7:xor (rc0 0:1, creg 0:1, sig7 0:1);
  box8:xor (rc1 0:1, creg 1:1, sig8 0:1);
  box9:xor (rc2 0:1, creg 2:1, sig9 0:1);
  box10:concat (rreg 0:1, areg 0:7, sig10 0:8);
  box11:concat (areg 7:1, a1reg 8:7, sig11 0:8);
  box12:or (creg0 0:1, creg1 0:1, sig12 0:1);
  box13:or (sig12 0:1, creg2 0:1, sig13 0:1);
  box14:and (sig13 0:1, decal 0:1, sig14 0:1);
  box15:not (sig13 0:1, sig15 0:1);
  box16:and (sig15 0:1, decal 0:1, sig16 0:1);
  box17:concat (areg 0:8, a1reg 0:1, b_sm 0:16);
  box18:not (regetat 0:1, sig18 0:1);
  box19:not (regetat 1:1, sig19 0:1);
  box20:not (regetat 2:1, sig20 0:1);
  box21:and (sig18 0:1, sig19 0:1, sig20 0:1, dummy 0:1);
  box24:and (sig18 0:1, sig19 0:1, regetat 2:1, repos 0:1);
  box27:and (sig18 0:1, regetat 1:1, sig20 0:1, init 0:1);
  box29:and (sig18 0:1, regetat 1:1, regetat 2:1, addition 0:1);
  box32:and (regetat 0:1, sig19 0:1, sig20 0:1, decal 0:1);
  box34:and (regetat 0:1, sig19 0:1, regetat 2:1, fin 0:1);
  box35:and (repos 0:1, sig1 0:1, sig35 0:0);
  box36:and (repos 0:1, sig3 0:1, sig36 0:0);
  box37:and (init 0:1, init 0:1, sig37 0:0);
  box39:and (addition 0:1, addition 0:1, sig39 0:0);
  box40:and (decal 0:1, sig14 0:1, sig40 0:0);
  box42:and (decal 0:1, sig16 0:1, sig42 0:0);
  box44:and (fin 0:1, fin 0:1, sig44 0:0);
  box45:or (sig39 0:0, sig42 0:0, sig45 0:1);
  box46:or (sig35 0:0, sig37 0:0, sig40 0:0, sig46 0:1);
   box47:or (sig36 0:0, sig37 0:0, sig40 0:0, sig42 0:0, sig44 0:0,
sig47 0:1);
  box48:mux2_8 (repos 0:1, init 0:1, b_em 0:8, b_em 0:8, sig48 0:8);
  box49:or (repos 0:1, init 0:1, sig49 0:1);
  box50:reg8 (cp 0:1, sig49 0:1, sig48 0:8, breg 0:8);
  box51:mux2_1 (repos 0:1, init 0:1, b_vdd 0:1, b_vcc 0:1, sig50 0:1);
  box53:reg1 (cp 0:1, sig49 0:1, sig50 0:1, busyreg 0:1);
  box54:mux2_8 (init 0:1, decal 0:1, breg 0:8, sig11 0:8, a1reg 0:1);
   box55:mux4_1 (init 0:1, sig4 0:1, decal 0:1, nul 0:1, b_vdd 0:1, s
0:8,
sig10 0:8, nul 0:1, sig53 0:1);
  box56:or (init 0:1, sig4 0:1, decal 0:1, sig54 0:1);
  box57:reg8 (cp 0:1, sig54 0:1, sig53 0:1, areg 0:8);
  box58:mux2_1 (sig4 0:1, sig6 0:1, rs 0:1, b_vdd 0:1, sig55 0:1);
```

```
  box59:or (sig4 0:1, sig6 0:1, sig56 0:1);
  box60:reg1 (cp 0:1, sig56 0:1, sig55 0:1, rreg 0:1);
  box61:reg1 (cp 0:1, addition 0:1, sig7 0:1, creg 0:1);
  box62:reg1 (cp 0:1, addition 0:1, sig8 0:1, creg 1:1);
  box63:reg1 (cp 0:1, addition 0:1, sig9 0:1, creg 2:1);
  box64:reg1 (cp 0:1, cp 0:1, sig45 0:1, regetat 0:1);
  box65:reg1 (cp 0:1, cp 0:1, sig46 0:1, regetat 1:1);
  box66:reg1 (cp 0:1, cp 0:1, sig47 0:1, regetat 2:1);
  box67:buffer_in (vcc 0:1, b_vcc 0:1);
  box68:buffer_in (vdd 0:1, b_vdd 0:1);
  box69:buffer_in (em 0:8, b_em 0:8);
  box70:buffer_in (depart 0:1, b_depart 0:1);
  box71:buffer_in (re 0:1, b_re 0:1);
  box72:buffer_out (sm 0:16, b_sm 0:16);
  box73:buffer_out (occupe 0:1, b_occupe 0:1);
endconnect;

end ;
```

visualisation du résultat

Comme dans le cas de COMPLO, le résultat de la compilation du multiplieur est quasiment illisible. Les pages suivantes présentent une visualisation dessinée "à la main" du circuit obtenu.


creg[0]
creg[1]
creg[2]
depart
repos
addition
decal
regetat[0]
regetat[1]
regetat[2]
cp
cp
repos
init
addition
decal
fin


≠
≠
!
≠
!
addition
CREG
repos
+
init
b_vdd
b_vcc
mux
BUSY-
-REG
repos
init
em
em
mux
BREG
init
decal
+
mux
A1REG
!
!
SM
init
decal
addition
+
&
mux
AREG
!
¬
+
&
rs
b_vdd
mux
RREG

5.12 Avancement du prototype et développements ultérieurs

En Septembre 88, le prototype est à l'état de maquette dite "académique". Le module d'implémentation naive est écrit en Lisp et le reste en Prolog. Cette maquette a permis de valider la démarche sur des exemples simples:

- synthèse d'un multiplieur 16 bits

- synthèse d'exemples de fonctions

La composition a été effectuée en utilisant 2 bibliothèques de cellules, une bibliothèque Bull-Systèmes (63 règles) et la bibliothèque VTI (47 règles).

La maquette doit être complétée au niveau des primitives qu'elle traite: par exemple, il faut lui ajouter un traitement des boucles "for".

La phase d'optimisation ("pré-compilation heuristique") ne correspond pas aux méthodes actuelles de modélisation. Elle doit donc être remaniée complètement.

L'étape suivante de validation sera de mettre l'outil dans les mains des concepteurs. Pour cela, il doit être reprogrammé et rendu convivial, avec des interfaces homogènes, etc....

Chapitre 6

SYNTHESE DE PARTIE OPERATIVE

6-SYNTHESE DE PARTIE OPÉRATIVE

Ce chapitre aborde la synthèse des parties opératives et de leur contrôle. Cette synthèse diffère de la synthèse de parties opératives telle qu'on la trouve dans les compilateurs commerciaux, qui est uniquement une synthèse structurelle. Elle diffère également de la synthèse logique, essentiellement par le fait qu'elle part d'une description source algorithmique, écrite dans un langage procédural. Ceci complique la synthèse car la description est alors "de plus haut niveau", c'est-à-dire qu'elle contient moins d'informations. Nous précisons ceci en différenciant les langages procéduraux des langages non procéduraux (cf. chapitre 2).

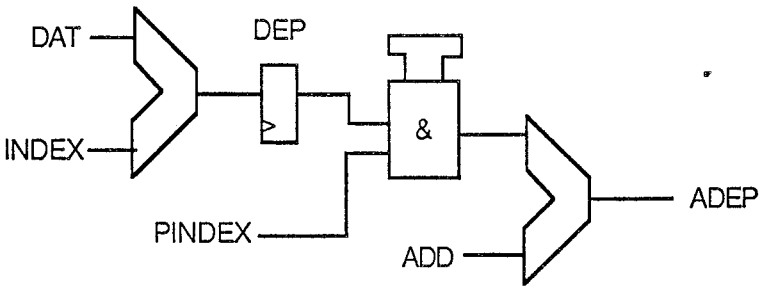
6.1. description fonctionnelle et description algorithmique

La plupart des langages de description de matériel dit "fonctionnels" se ressemblent, syntaxiquement parlant. Mais une même instruction peut avoir des significations très différentes : prenons par exemple deux instructions très simples :

```
ADEP : = ADD + (DEP & PINDEX) ;
DEP : = DAT  + INDEX ;
```

Dans le cas d'un langage non procédural (déclaratif) elles exprimeront le chargement du registre ADEP avec le résultat de l'opération ADD + (DAT + INDEX) & PINDEX ; si le langage est procédural elles expriment la remise à jour du registre DEP après le chargement de ADEP.

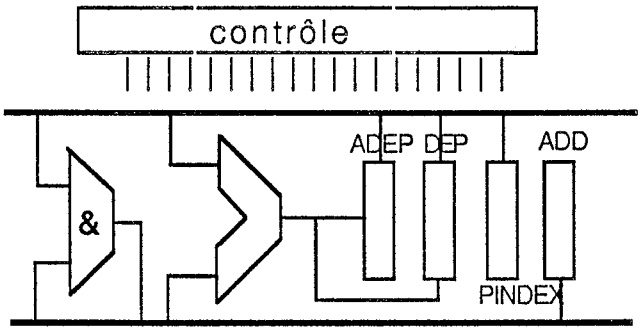
Ces deux versions nécessitent des traductions matérielles différentes. La version déclarative nécessite deux additionneurs :



réalisation "non procédurale"

Le travail de synthèse est fini : les modules réalisent complètement le fonctionnement requis.

La version procédurale nécessite un module de séquencement agissant sur une partie opérative, ici réduite à un additionneur et un opérateur &).



réalisation "procédurale"

La synthèse de parties opératives se fait en général à partir d'un langage procédural. C'est le cas par exemple des outils développés à Carnegie-Mellon dans le cadre du projet CMU-DA : tous ces outils utilisent le langage procédural ISPS.

La synthèse que nous définissons ici comprend à la fois le choix des modules et de leurs interconnexions, mais également la génération du contrôle qui agira sur ces modules.

6.2. les tâches de la synthèse de partie opérative

Nous donnons ici les tâches qu'il faut effectuer pour transformer une description procédurale en un circuit "prêt à fonctionner". L'ordre dans lequel sont effectuées ces tâches dépend beaucoup de la stratégie de la compilation.

* Génération du séquencement (ordonnancement des opérations) : dans le cas où le séquencement des opérations élémentaires n'est pas complètement spécifié, il faut ordonner les opérations les unes par rapport aux autres, et les organiser en cycles .

* Identification des boucles.

* Résolution des conflits de dépendance des données.

* Allocation / minimisation des registres : le concepteur a défini dans sa description des variables de type REGISTRE (capables de conserver leur valeur dans le temps). De plus les traitements antérieurs auront souvent créé des variables REGISTRE. Cette phase en diminue le nombre et alloue ces variables à des registres réels, en faisant éventuellement partager un registre réel par plusieurs variables.

* Optimisation des connexions (bus ou multiplexeurs).

* Assignement des opérations sur les différents blocs fonctionnels.

* Choix des composants réels ou sélection des modules.

* Assignation et minimisation d'états : la génération du séquencement a créé un automate d'états finis possédant autant d'états que de cycles d'opérations élémentaires. Cette phase en diminue le nombre et leur affecte un codage (ou bien choisit une partie contrôle micro-codée...). Cf "Synthèse de partie contrôle".

* Optimisations globales ou locales du matériel obtenu : on rejoint ici l'optimisation telle qu'elle est présentée au chapitre 2.

Cette liste n'est pas complète : par exemple, dans SYCO, comme les parties opératives sont divisées en sous-parties opérative, on passe par une étape supplémentaire pour déterminer le nombre de sous-parties

opératives. Par contre, comme il y a un opérateur par sous-partie opérative, notre tâche d'"assignation des opérateurs" n'apparaît plus.

6.3. illustration : synthèse procédurale à partir de LDS

6.3.1. Objectifs de l'expérience

Il s'agit ici d'aborder le problème de la génération du séquencement dans la synthèse automatique de circuits. L'expérience aborde ce problème comme un problème d'ordonnancement. On retrouve également des techniques proches des techniques "data-flow" de compilation de langages de programmation. Nous soulignons dans la suite des points communs entre les deux sortes de compilation et la manière dont nous transposons certaines techniques.

Pour présenter un travail complet, nous avons ajouté des algorithmes de partage de registres, de génération de partie contrôle et de partie opérative au module de génération du séquencement. Le partage de registre est un complément nécessaire de l'algorithme de séquencement, car celui-ci génère beaucoup de variables intermédiaires.

Les algorithmes utilisés sont donnés de manière complète. La méthode présentée ici a été implémentée et expérimentée avec des outils "académiques". Il comporte de grosses restrictions (additionneurs 2 entrées, largeur des données et fonctions d'entrée/sortie simplifiées).

6.3.2. Le contexte

Même s'il est orienté vers la synthèse de micro-processeurs, le travail présenté ici a beaucoup bénéficié des discussions avec les concepteurs de Bull-Systèmes.

Le langage LDS, qui nous sert ici de langage source, est présenté dans le chapitre précédent et en annexe. La sémantique qui lui est attribuée ici diffère nettement des hypothèses formulées dans le chapitre 5. Le langage représente ici un jeu d'instructions, et non la fonction d'une partie de matériel. Il est considéré de manière procédurale. Les étiquettes ne représentent plus les différents états d'un automate d'états finis, mais sont des points de branchement. Les opérations consécutives entre deux étiquettes ne sont pas considérées comme forcément parallèles.

6.3.3. Notre exemple

L'exemple présenté partiellement ici représente un micro-processeur simple et ses opérations d'entrée/sortie. Les instructions sont inspirées du jeu d'instruction de la famille DPS7 et surtout de celui du micro-processeur 6502. Il prend en compte 5 modes d'adressage. Il est modélisé par un automate d'états finis comportant 5 états. Nous ne présentons ici que le CMODULE :

```
CMODULE MAIN;

<rest> if (START) goto depart ;
            else goto rest;
       end;

<depart>    CO := 0;

<fetch>     RI := access(CO);
       LI := RI 0:2;       ? longueur de l'instruction
       I := RI 2:3         ? paramètre ou adresse GR
       MODAD := RI 5:3;          ? mode d'adressage
       OPC := RI 8:4;            ? code opération
       D := RI 12:20;            ? adresse
       OLDCO := CO;
       CO := CO + LI;

<calcad>    case (MODAD)          ? calcul d'adresse
            when ('001')  AD :=D;                  ? dep. direct
            when ('010')  AD :=ACCESS(D);          ? indirect
            when ('011')  AD :=D + R_INDEX;        ? indexé
            when ('100')  AD :=ACCESS(D+R_INDEX); ? indexé indirect
            when ('101')  AD :=ACCESS(D)+R_INDEX; ? indirect indexé
       end;

<decode>    case (OPC)
            when (1) execute lgr ;
            when (2) execute adgr;
            when (3) execute ldsum;
            when (4) execute movl;
            when (5) execute movs;
            when (6) execute mult;
       end;

<lgr>  ? load general register
```

```
        GR(I) := access(AD);
        if (GR(I)=0)   CC:=0;
             else if (GR(I) 0:1 = '0') CC :=2;
                       else CC := 1;
                  end;
        end;

? … nous ne donnerons pas toutes les fonctions

<movs> ? move short
        D1:=access(OLDCO+1+R_INDEX);
        GR(1):=access(AD+2J);
        GR(2):=access(AD+2J+1);
        send(GR(1),AD+2J+D1);
        send(GR(2),AD+2J+D1+1);
        end;

<movl> ? move long
        if (L1=1) D1 := access(OLDCO+1+R_INDEX);
             else D1 := DEFAULT; end;
        for J:=0 to I
             GR(1):=access(AD+2J);
             GR(2):=access(AD+2J+1);
             send(GR(1),AD+2J+D1);
             send(GR(2),AD+2J+D1+1);
        end;
end;
```

texte source partiel de notre exemple

Nous avons omis la partie déclaration de notre modèle : précisons que les variables *GR(1)*, *GR(2)*, *OLDCO*, etc... qui apparaissent dans le CMODULE sont globales et sont toutes déclarées comme des registres.–

6.4. analyse "data-flow" et analyse "control-flow"

La représentation data-flow[1] est chaudement préconisée par Aho et Ullmann dans [Aho 77] pour la compilation des langages de haut niveau. En général, la compilation utilise les arbres syntaxiques comme structure intermédiaire. En conséquence, la génération de code est effectuée instruction par instruction. L'analyse "data-flow" génère un graphe orienté appelé "graphe data-flow". Chaque nœud de ce graphe correspond à une variable ou un opérateur du programme analysé.

Le graphe "data-flow" ne représente pas toute la sémantique du programme. Par exemple il ne représente pas les boucles. Il doit être complété d'un "graphe control-flow". Ici chaque nœud représente une instruction et chaque arc le transfert du contrôle de l'instruction source à l'instruction cible.

L'utilisation des graphes "data-flow" comme structure intermédiaire est très synthétique. Elle permet de rassembler tout ce qui concerne une même variable, de manipuler les constructions de contrôle... Dans le cadre de la synthèse de matériel, l'analyse data-flow fait apparaître les dépendances de données et donc permet de résoudre les conflits d'accès. Comme elle représente les dépendances entre instructions, l'analyse "control-flow" est très utile pour identifier les parallélismes, comme par exemple dans MIMOLA ([Marw 84]), ou pour "déplier" les boucles ([DeMa 86]).

Les transformations ne sont pas toutes spécifiques de l'analyse "data-flow", en particulier le dépliage de boucles. Mais la technique en permet une vision systématique. D'autres, par contre, seraient difficiles sans cette analyse, comme par exemple les mises en correspondance sur le graphe de contrôle décrit par Kahrs dans [Kah 86].

Les graphes data-flow sont lourds à mettre en œuvre. Tous les projets de synthèse de Carnegie-Mellon utilisent la même forme, appelée *value trace*, obtenue à partir du langage *ISPS*, mais d'autres avouent rentrer leurs graphes à la main (par exemple [Raj 86]).

---

[1] Nous n'utiliserons pas la francisation "analyse des flux de données", à cause de sa lourdeur.

6.5. illustration : analyse "control-flow" et "data-flow" d'une description LDS

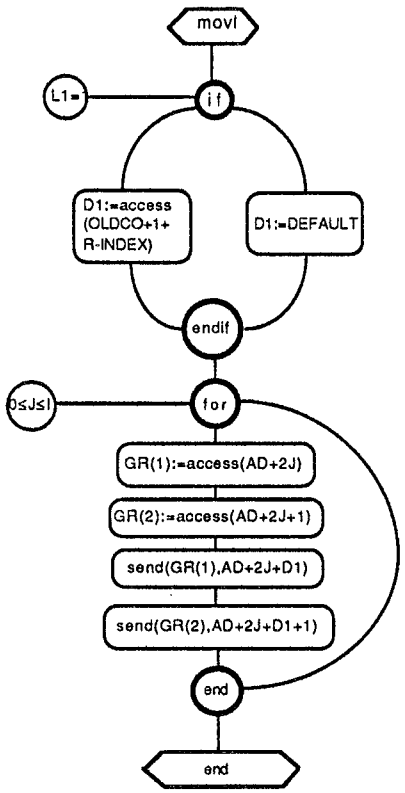
6.5.1. dépliage des boucles FOR par analyse control-flow

Le calcul du séquencement se fait pour des programmes "non répétitifs", c'est à dire ne comportant pas de boucles. Il faut donc "déplier" les boucles FOR (ce n'est pas la seule solution : dans le système CATHEDRAL [DeMa 86] il est prévu une synthèse spécifique des boucles de répétition).

Nous prenons comme exemple l'instruction *movl* :

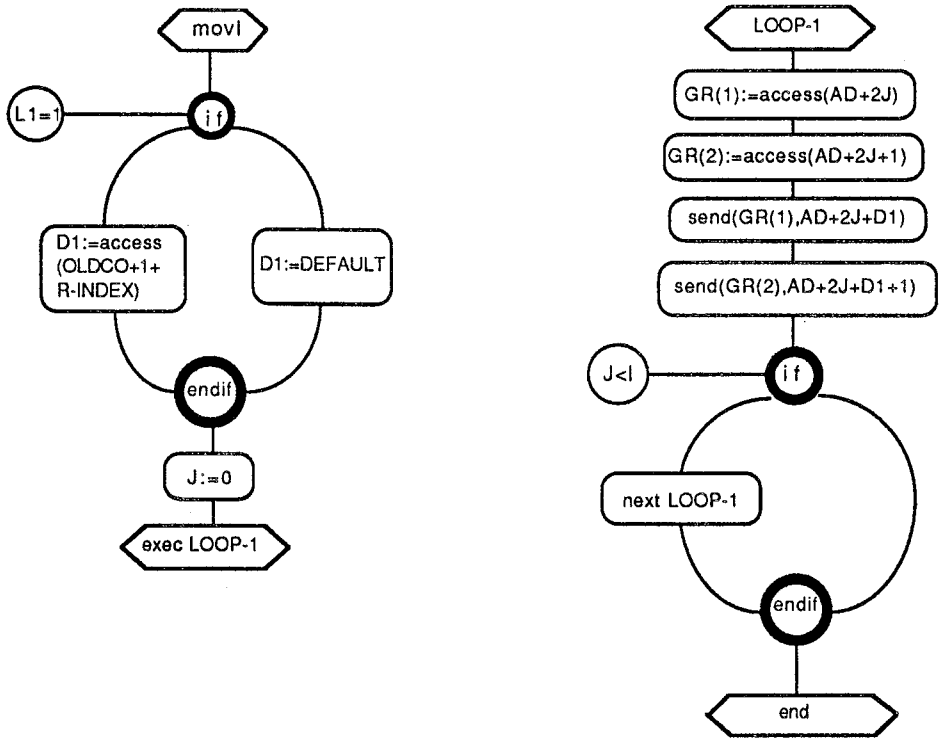
```
<movl> ? move long
       if (L1=1) D1 := access(OLDCO+1+R_INDEX);
            else D1 := DEFAULT; end;
       for J:=0 to I
            GR(1):=access(AD+2J);
            GR(2):=access(AD+2J+1);
            send(GR(1),AD+2J+D1);
            send(GR(2),AD+2J+D1+1);
       end;
end;
```

Cette séquence se traduit par le graphe control-flow suivant:



graphe control-flow de l'instruction movl

Le dépliage se traduit par l'introduction d'une routine : LOOP-1. Le graphe initial se traduit par deux graphes de contrôle :



graphe control-flow de movl après modification

Si on traduit ces graphes en code LDS, ceci devient :

```
<movl> ? move long
       if (L1=1) D1 := access(OLDCO+1+R_INDEX);
            else D1 := DEFAULT;
       end;
       J:=0;
       EXECUTE(LOOP_1);
       end;

<LOOP_1>    GR(1):=access(AD+2J);
       GR(2):=access(AD+2J+1);
       send(GR(1),AD+2J+D1);
       send(GR(2),AD+2J+D1+1);
       if (J<I) J:=J+1; NEXT LOOP_1; end;
       end;
```

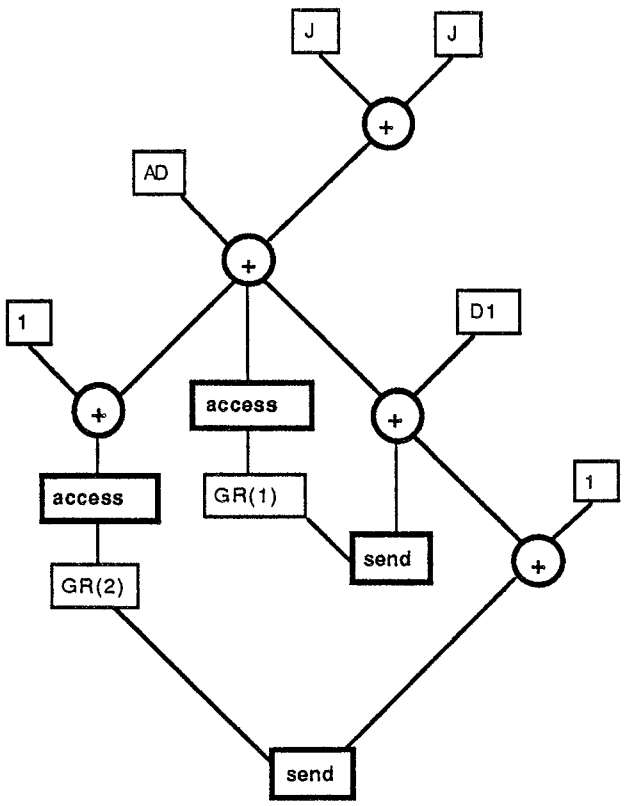
6.5.2. simplification des expressions par analyse "data-flow"

Notre système ne permet pas les additions de trois opérandes : une instruction telle que...

```
GR(2) := access(AD+2J+1);
```

...doit donc être simplifiée par l'introduction de variables intermédiaires. Ceci peut être fait par le parcours de l'arbre de l'instruction pendant l'analyse syntaxique. Mais dès qu'il y a plusieurs instructions, on court le risque de dupliquer les opérations, ou d'introduire les variables dans un ordre peu efficace. Par exemple, dans le cas de la séquence LOOP-1, il est plus efficace de calculer AD+2J que 2J+1. Le choix n'est pas évident quand on analyse la séquence instruction par instruction.

L'analyse data-flow prend en compte le problème pendant la construction du graphe (algorithme 12.2 de [Aho 77]). Pour LOOP-1 on obtient le graphe suivant :



graphe data-flow de l'instruction movs

Si on veut construire la séquence de code LDS modifié correspondant au graphe ci-dessus, la méthode est simple : on attribue à chaque nœud opérateur une instruction. Quand l'opération est une opération d'affectation, et qu'aucune variable affectée n'est explicitée, on crée une nouvelle variable *X_i*. Par exemple, LOOP-1 deviendra :

```
X_3:=J+J;
X_4:=AD+X_3;
X_5:=X_4+1;
GR(2):=access(X_5);
```

Résultat : à l'issue de cette analyse "data-flow" et "control flow", on obtient une description proche de la description originale, mais ne comportant plus de boucles, et dont toutes les opérations sont dyadiques. Cette description contient les variables globales initiales, et des variables "locales" dont les durées de vie ne dépassent pas la séquence où elles sont affectées.

6.6. la génération du séquencement

Nous considérons ici le cas d'une modélisation comportementale où le coté temporel n'est pas complètement spécifié. Dans le cas contraire, on a une modélisation fonctionnelle. Le concepteur a décrit les opérations à réaliser dans un ordre "a priori", comparable à l'ordre des instructions dans un programme Pascal. Le but de la phase de génération du séquencement est d'affecter toute opération élémentaire à un cycle de base de la machine. Pour cela il faut tenir compte de 3 contraintes :

1-contrainte d'antériorité (ou contrainte de précédence)
Une instruction utilisant une variable V en partie droite ne peut précéder une affectation de cette variable qui la devance dans la description originelle. En clair, le séquencement doit respecter l'ordre des affectations de la description source.

2-durée des opérations
L'utilisateur fixe une durée maximale pour chaque opération.Par exemple :
- addition : 2 cycles
- lecture mémoire : 3 cycles

- écriture mémoire : 2 cycles
- transfert registre/registre : 1 cycle

3-nombre maximum de ressources
L'utilisateur fixe le nombre maximum de chaque ressource. Par exemple :
- 1 seule opération d'addition à la fois
- 2 opérations d'entrée/sortie

La génération du séquencement peut être présenté comme un problème d'ordonnancement classique. Il s'agit de fixer un ordre pour l'exécution de tâches de durée non nulle qui ont entre elles des contraintes d'antériorité. On peut y rencontrer les problèmes classiques de l'ordonnancement : présence de boucles, contraintes non disjonctives, durées non fixes, ... Comme pour tous les problèmes d'ordonnancement, il est préférable d'utiliser des méthodes heuristiques bien adaptées aux problèmes spécifiques, plutôt que d'essayer d'appliquer une méthode générale qui n'existe pas.

Les méthodes que l'on trouve dans la littérature sont très diverses, mais on y rencontre deux grandes familles : les méthodes de séquencement par **construction itérative** et les méthodes **transformationnelles** [Ca& 88].

La méthode de construction itérative consiste à prendre en compte les opérations les unes après les autres, dans un ordre déterminé, et à leur assigner un séquencement. Dans *Facet* [Tse 81], on utilise la méthode "ASAP" (As Soon As Possible) : les opérations sont effectuées le plus tôt possible compte-tenu des contraintes d'antériorité. On ne tient pas compte des limitations de ressources. Cette méthode est également utilisée dans *Mimola* [Marw 84], [Marw 86].

La méthode transformationnelle consiste à partir d'un séquencement initial, généralement soit massivement parallèle soit massivement série, et à le modifier jusqu'à ce que les contraintes soient satisfaites. *Expl* en constitue un exemple particulier puisqu'il explore toutes les solutions possibles pour une séquence d'opérations donnée. Cette recherche exhaustive n'est malheureusement pas possible pour des exemples en vraie grandeur.

Dans *Hal* [Pau 86], on utilise un algorithme spécial pour distribuer régulièrement les opérations entre ressources similaires ("load balancing").

Dans *Maha* [Par 86], le séquencement se fait en deux temps : tout d'abord on fixe le déroulement des opérations du chemin critique. Dès qu'il n'y a plus de possibilité de choix dans les dates, les autres sont traitées dans

l'ordre de "liberté" croissant. L'ordre de liberté correspond à la marge (date au plus tard- date au plus tôt).

Dans *Cathedral* [DeMa 86], les contraintes du séquencement sont modélisées par un ensemble d'équations linéaires entières, dont les variables représentent les dates.

Toutes ces méthodes calculent des séquencements basés sur une seule horloge. [NPar 85] aborde le problème de l'optimisation du séquencement des transferts de registres "multi-phasés".

Sehwa ([NPar 86], [Par 88]) traite un cas plus complexe, le séquencement dans une machine "pipe-line". La méthode utilisée est basée sur l'"urgence" des opérations. L'"urgence" d'une opération rend compte de sa distance dans le temps avec soit l'entrée soit la sortie du graphe data-flow. On trouve une idée similaire dans [Gir 84] quoique le terme d'"urgence" soit pris dans un sens différent : comme les opérations sont considérées de durées égales, l'urgence représente le nombre d'opérations dans la séquence.

Nous proposons dans le chapitre suivant une méthode de séquencement du type transformationnelle basée sur le formalisme "potentiel-tâche". Notre méthode est proche de celle de *Slicer* décrite dans [Pau 86]. On retrouve des définitions voisines des dates au plus tôt, des dates au plus tard et des chemins critiques. Par contre, il n'y a pas de prise en compte des ressources critiques. Notre façon de gérer les conflits d'accès aux ressources est proche de la prise en compte des contraintes temporelles décrite dans [Nes 86].

6.7. illustration : génération du séquencement à partir d'un formalisme potentiel-tâche

6.7.1. nos contraintes

Nous nous fixons arbitrairement des contraintes pour le découpage en cycles simples des instructions de notre modèle.

1-contrainte d'antériorité (ou contrainte de précédence)
Cette contrainte n'est pas spécifique au modèle, elle traduit la règle de l'affectation unique : une instruction utilisant une variable V en partie droite ne peut précéder une affectation de cette variable qui la devance dans la description originelle.

2-modèle temporel pour les opérations
Nous utiliserons :
- addition : 2 cycles, on considère que les entrées sont tamponnées.
- lecture mémoire (fonction *access*) : 3 cycles,
- écriture mémoire (fonction *send*) : 2 cycles, on considère que l'entrée des données est tamponnée.
- transfert registre/registre : 1 cycle

3-nombre maximum de ressources
Ici nous nous fixons les limitations suivantes :
- 2 opérations d'addition à la fois
- 1 seule opération d'entrée/sortie

6.7.2. méthode générale

Notre méthode consiste à trouver un séquencement des opérations élémentaires séquence par séquence. Un premier séquencement est tout d'abord déterminé à partir des contraintes d'antériorité : c'est la construction du premier graphe potentiel/tâche. Ce graphe est ensuite incrémentalement modifié pour satisfaire les contraintes de partage des ressources.

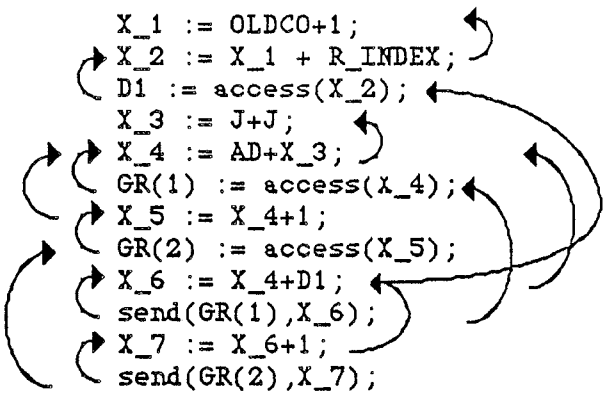
6.7.3. calcul des contraintes d'antériorité

Méthode : une instruction I1 comportant une variable V en partie droite doit être postérieure à la dernière instruction précédant I1 ou V apparait en partie gauche. Si on préfère les algorithmes, ceci se traduit :

```
début
   I= nombre d'instructions;
   tantque ( I≠1) faire
      pourtout ( V εpartie-droite (instruction[I]))
      faire
         J=I-1;
         tantque ( J≠0)&&(V ≠partie-gauche(instruction[J])) faire
            J=J-1;
         finfaire
         précédent(J,I)          /* J doit précéder I */
         J=0;
         I=I-1;
   finfaire
fin
```

note : Dans le cas des instructions complexes du type if...then...else, on considère l'ensemble des variables apparaissant en partie droite et l'ensemble des variables apparaissant en partie gauche.

Nous illustrons la méthode sur l'instruction *movs* :



La flèche Ij <- Ii exprime la contrainte de précédence "l'instruction Ij doit précéder l'instruction Ii".

6.7.4. formalisme et construction du graphe potentiel-tâche

La première étape de notre séquencement est de calculer un ordonnancement pour les instructions, basé sur les contraintes d'antériorité et en ignorant les contraintes de partage de ressources. Le problème à résoudre est de définir la meilleure "date au plus tôt" pour la fin de chaque séquence d'instructions élémentaires. Nous utiliserons une représentation voisine des réseaux PERT, mais plus adaptée à notre problème : le diagramme potentiel/tâche (ou MPM).

Dans le diagramme Potentiel/tâche, les sommets représentent les tâches alors que dans un PERT ils représentent les dates privilégiées. L'ensemble des tâches Ii que nous considérons est l'ensemble des instructions élémentaires d'une séquence, auquel on ajoute deux tâches fictives "début" et "fin" .

Les arcs du graphe représentent les contraintes d'antériorité :

- (0,i) : l'instruction Ii n'a aucune contrainte de précédence

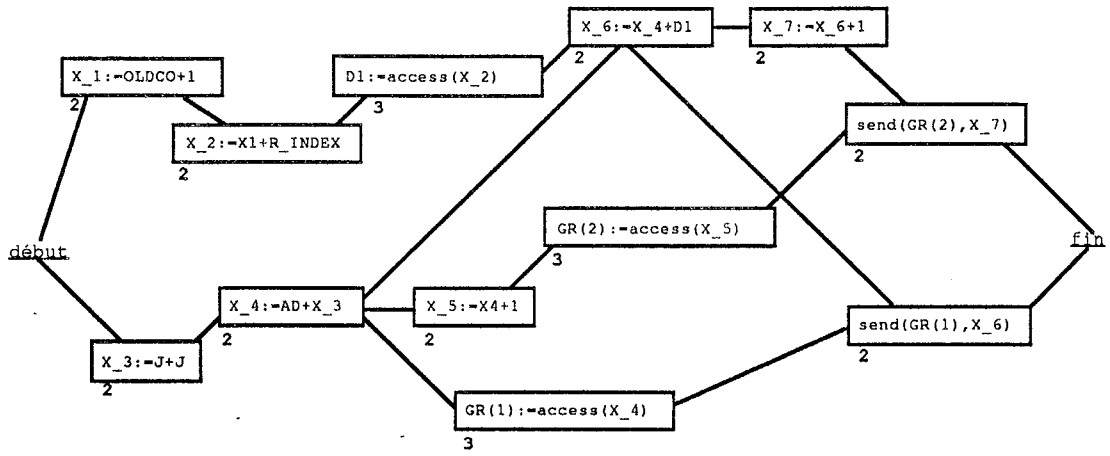
- (i,j) : Ij doit être précédée par Ii.

- (i,NbreInst+1) : aucune instruction ne doit être précédée par Ii (NbreInst+1 représente la pseudo-instruction "fin").

A chaque tâche est associée une durée (en cycles) . On calcule pour chaque tâche une "date au plus tôt", que nous noterons *dtot*, qui représente la date à laquelle on peut commencer l'exécution de cette tâche. La durée minimum de la séquence est donnée par la "date au plus tôt" de la pseudo-instruction "fin".

Le formalisme potentiel/tâche ne s'applique qu'à des graphes sans boucles. Nous montrons en annexe que c'est le cas ici. En effet, l'algorithme utilisé pour calculer les contraintes d'antériorité n'introduit pas de boucles.

L'instruction *movs* se traduit par le graphe suivant :

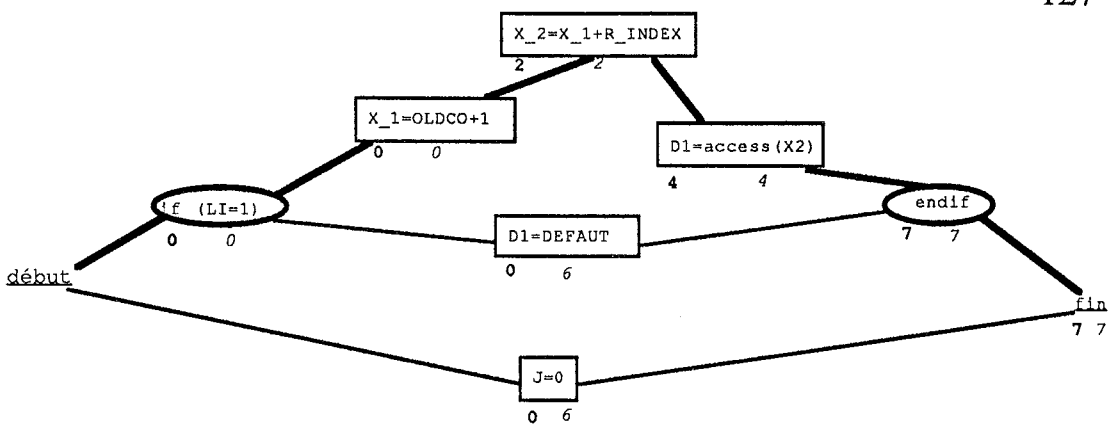


graphe de séquencement de l'instruction movs

Les instructions de type *if...then...else* et *case* donnent lieu à la construction d'un sous-graphe. Par exemple, dans le cas d'une instruction if...then...else, le sous-graphe commence par la pseudo-instruction if et finit par la pseudo-instruction endif. Le sous-graphe qui représente les instructions sous portée de la condition est construit comme un graphe ordinaire, pour lequel on pourra calculer une durée *d*. Dans le graphe principal, le sous-graphe sera considéré comme un tâche de durée *d*.[1]

Nous donnons ci-dessous le diagramme correspondant à l'instruction *movl* (les conventions du schéma sont décrites plus loin).

[1] Dans la réalité, l'algorithme précédent est toujours utilisable: les instructions sous portée de la condition sont considérées comme toutes activable en même temps. Il n'est pas nécessaire de calculer à part la durée du sous-graphe.



graphe de séquencement de l'instruction movl

note : Cette méthode qui a l'avantage de sa simplicité, est peu efficace au niveau du partage des registres : les deux branches *then* et *else* sont considérées comme parallèles mais non disjonctives : elles ne pourront pas partager de registres.

6.7.5. calcul des dates au plus tôt

On retrouve les dates au plus tôt et les dates au plus tard dans le formalisme PERT, et les algorithmes utilisés sont très proches des algorithmes utilisés pour les graphes PERT. La date au plus tôt d'une tâche T est la date minimum à laquelle on peut entreprendre cette tâche. Dans notre cas c'est la date minimum à laquelle on peut commencer une micro-instruction. Comme nous prenons comme principe de fixer notre séquencement sur la date au plus tôt, ce sera également la date de début de commencement effectif de la micro-instruction.

Nous présentons ci-dessous une forme simple de l'algorithme de calcul de date au plus tôt, en utilisant la forme exposée dans [Saka 84].

Chaque instruction est représentée par un entier, la tâche fictive "début" étant représentée par 0. Chaque relation de précédence, et donc chaque arc du graphe par le couple (i,j) exprimant que l'instruction j doit précéder l'instruction i. On note $\mathcal{P}$ l'ensemble des arcs, d(i) la durée de l'instruction i et dtôt(i) la date au plus tôt de l'instruction i.

```
début
   S={ 0 } ; dtôt(0) = 0;
   tantque il existe j tel que {j} ⊄S et dont tous les prédécesseurs sont dans S faire
      dtôt(j) = max (i,j)∈ℙ { dtôt(i) + d(i) }
      S = S∪{j}
   fintantque
fin
```

algorithme 1 : calcul de la date au plus tôt

remarque : Dans la réalité, cet algorithme est programmé de manière beaucoup plus efficace, en utilisant la propriété prec(i,j) ⇒ i > j

```
début
   pourtout j de 1 à Nbreinstructions faire
      dtot(j) = 0;
      pourtout i de 0 à j-1 faire
         si prec(i,j) alors dtot(j) = max[ dtot(i,j), dtot(i)+d(i) ];
      finpourtout
   finpourtout
fin
```

algorithme 2 : calcul modifié de la date au plus tôt

De la même manière, on calcule les dates au plus tard. La date au plus tard d'une tâche T est la date maximum à laquelle la tâche T doit commencer pour ne pas retarder les autres tâches. La date au plus tard des micro-instructions n'est pas utilisée directement pour le séquencement, mais elle servira à déterminer le chemin critique, et à résoudre les conflits d'accès aux ressources. Nous donnons ci-dessous l'algorithme de calcul de date au plus tard, avec les mêmes conventions que plus haut.

```
début
   S={ NbreInst +1} ; dtard(NbreInst+1) = dtôt(NbreInst+1) ;
   tantque il existe i tel que {i} ⊄S et dont tous les successeurs sont dans S faire
      dtard(i) = min (i,j)ε 𝒫 { dtard(j) - d(i) }
      S = S∪{i}
   fintantque
fin
```

algorithme 3 : calcul de la date au plus tard

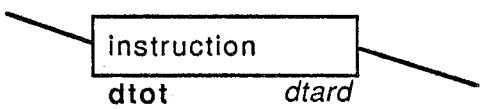
remarque : comme dans le cas du calcul de la date au plus tôt, cet algorithme sera traduit par l'algorithme suivant :

```
début
   dtard(NbreInst+1) = dtot(NbreInst+1) ;
   pourtout i de NbreInst à 1 faire
      dtard(i)=dtard(i+1);
      pourtout j de NbreInst à 0 faire
         si prec(i,j) alors dtard(i) = min(dtard(i),dtard(j)-d(i));
      finpourtout
   finpourtout
fin
```

algorithme 4 : calcul modifié de la date au plus tard

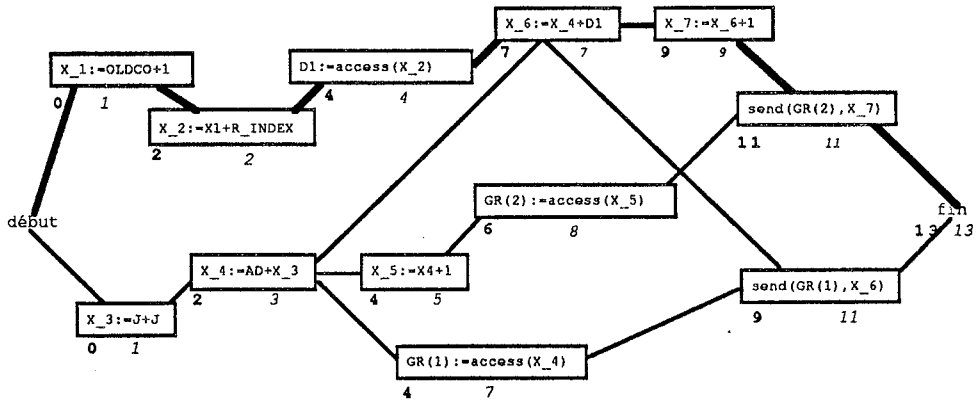
On appelle "tâche critique" une tâche pour laquelle dtot(i)=dtard(i), et chemin critique l'ensemble des tâches critiques.

Finalement, si on adopte les notations suivantes :



Les arcs représentés en gras sont les arcs du chemin critique.

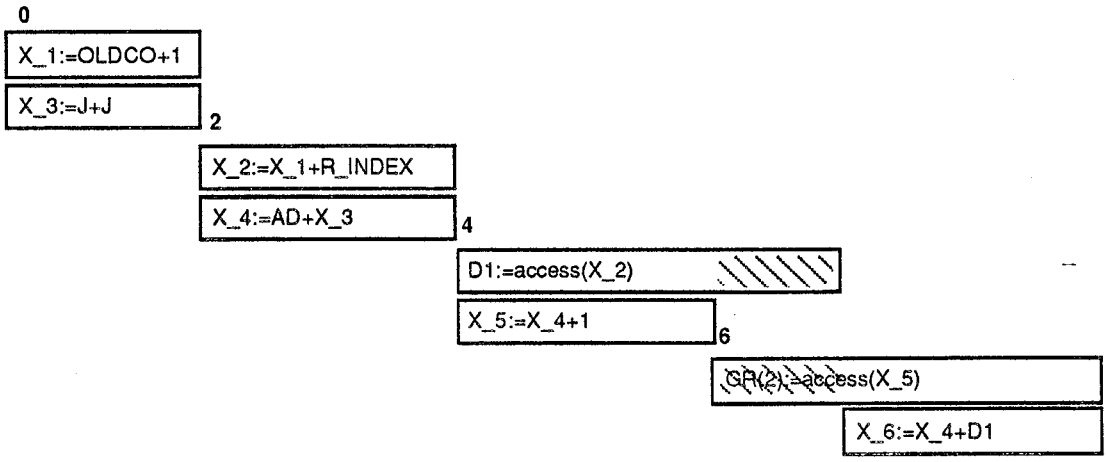
On obtient le graphe :



graphe initial pour movs

6.7.6. résolution des conflits

Le séquencement apparaît nettement sur un diagramme barre. Nous représentons ci-dessous une partie du diagramme barre de l'instruction movs, construit avec les dates au plus tôt.



Ce graphe construit au moyen des dates au plus tôt montre un séquencement qui serait valable s'il n'y avait aucune contrainte sur le partage des opérateurs. Si nous prenons en compte les contraintes définies précédemment, il apparaît une première violation à la date 6 : on trouve deux opérations d'entrée/sortie simultanées (zone hachurée).

Pour résoudre ces conflits d'accès aux ressources, on introduit des contraintes de précédence additionnelles. Si on appelle T1 et T2 deux tâches conflictuelles, on a la possibilité d'introduire deux contraintes : soit *précédent(T1,T2)*, soit *précédent(T2,T1)*. Pour pouvoir choisir entre les deux, on introduit la fonction *retard* définie comme suit :

retard(prec(T1,T2)) = dtôt(T1) + durée(T1) - dtard(T2)

Si la valeur est positive, *retard* représente le nombre de cycles perdus par l'introduction de la contrainte. Si elle est négative, la fonction donne le nombre de cycles dont on pourrait encore retarder T2 sans changer la durée de l'instruction. On choisit d'introduire la contrainte qui minimise *retard.*

Dans l'exemple plus haut, on a un conflit entre deux instructions, l'instruction [3]

```
[3]:      D1 :=  access(X_2);
```

et l'instruction [8]

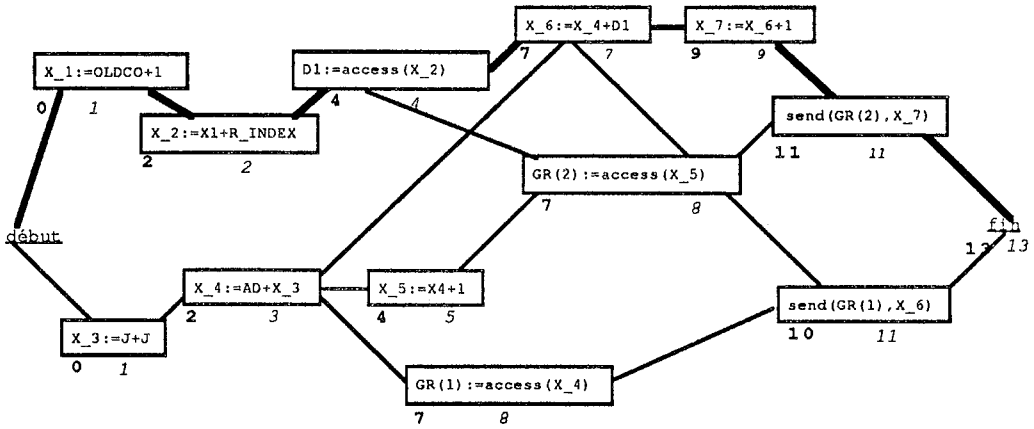
```
[8]:      GR(2) := access(X_5)  ;
```

Nous allons donc introduire une contrainte de précédence supplémentaire. On calcule les fonctions *retard* pour chacune des transitions :

retard(prec([3],[8])= dtot([3] + d([3]) - dtard([8]) = 4 + 3 - 8 = -1

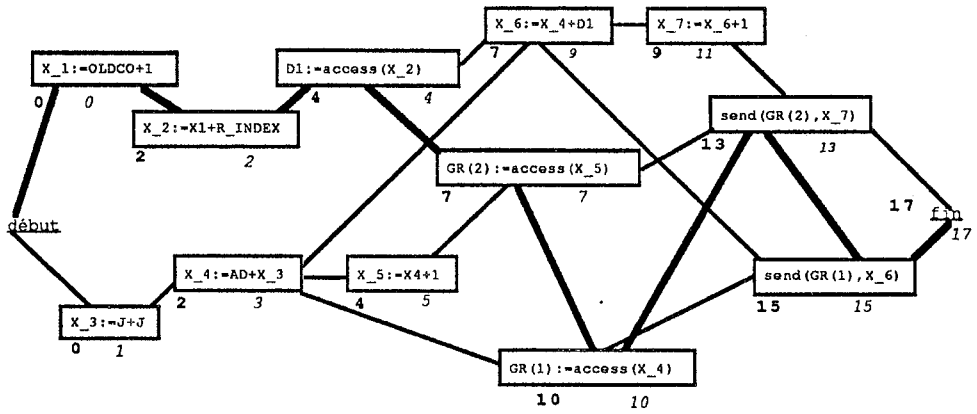
retard(prec([8],[3])= dtot([8] + d([8]) - dtard([3]) = 6´+ 3 - 4 = 5

Ceci signifie que si on introduit la contrainte prec([8],[3]), l'instruction *movs* s'allonge de 5 cycles, alors que si on introduit prec([3],[8]) la durée totale de *movs* ne change pas. L'ensemble des dates au plus tard et des dates au plus tôt doit être recalculé.



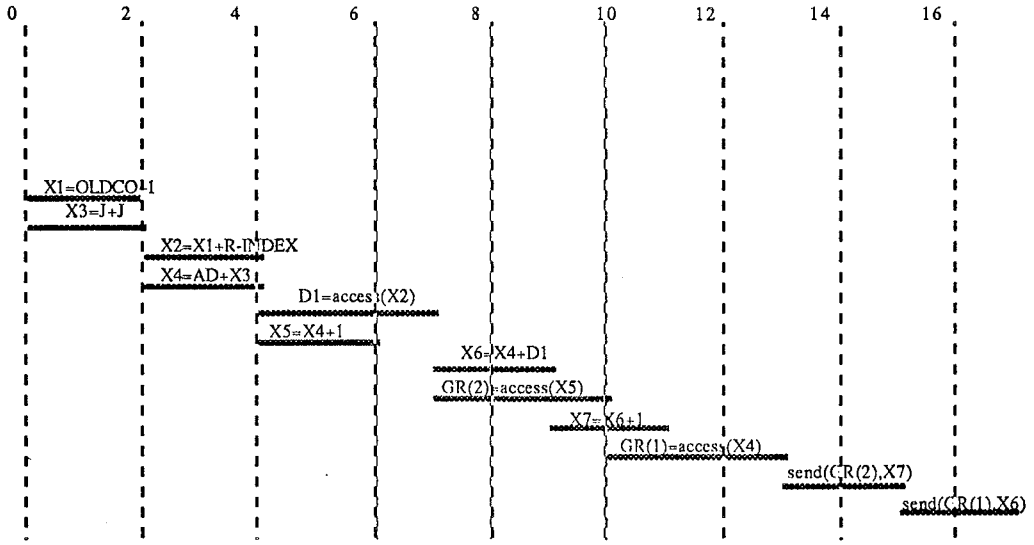
une étape du séquencement de movs

On itère le processus jusqu'à la pseudo-tâche "fin". On arrive finalement au graphe suivant :



séquencement final de movs

Comme ce diagramme est peu lisible, nous en donnons ci-dessous l'équivalent en diagramme barre.



représentation en "diagramme barre" de l'instruction movs

note : La méthode présentée ici est transformationnelle, puisqu'elle part d'un séquencement initial. Mais elle n'est pas globale : la prise en compte des contraintes se fait itérativement dans l'ordre des dates croissantes.

Résultat : A l'issue de cette phase de calcul du séquencement, pour chaque séquence d'instructions LDS, nous avons une liste d'opérations élémentaires datées à partir du début de la séquence.

6.8. partage des registres

Pour que deux variables puissent partager un élément de mémorisation, il faut qu'elles aient des temps de vie disjoints. Ici encore, on retrouve un problème de la compilation des langages de programmation.

Pour pouvoir effectuer le partage des registres, nous avons besoin de construire le tableau des temps de vie des variables, puisqu'elles ne peuvent partager un même registre que si elles ont des temps de vie disjoints.

6.9. illustration : partage des registres par coloration de graphes

On utilise souvent la coloration de graphe, ou le partitionnement en cliques, son cousin, dans les problèmes de partage de registre ([NPar 88], [Tse 83], [Chai 82]). La méthode que nous présentons est donc peu originale. Néanmoins, comme l'étape de partage des registres est une étape importante, nous présentons la méthode en détail.

On considère que chacun des registres définis par l'utilisateur correspond à un registre matériel. Le partage des registres ne sert qu'à assigner les variables intermédiaires introduites par programme à un registre matériel. Ces variables sont nombreuses : l'instruction *movs* en utilise déjà 7. Ce souci de partager les registres effectifs entre les variables correspond au problème d'allocation[1] de registre pour les compilateurs classiques. La seule différence est que, dans le cas de la compilation de langages de programmation, le nombre de registres utilisables est fixé, alors que dans le cas de la synthèse de matériel, on cherche à déterminer un nombre "quasi-minimum" de registres nécessaires.

Comme les instructions du modèle ne peuvent être activées simultanément, on peut effectuer le partage des registres séquence par séquence.

---

[1] En compilation on distingue l'allocation et l'assignement de registres: l'allocation de registres consiste à décider quelles variables il est judicieux de stocker dans les registres rapides de la machine, et l'assignement permet de préciser à quels registres précis sont affectées ces variables. Le terme exact serait donc "assignement".

6.9.1. analyse des temps de vie des variables

Une variable est dite active (ou vivante) entre le temps de son affectation et sa dernière utilisation avant l'affectation suivante. L'algorithme de remplissage du tableau est simple, et nous ne le reproduisons pas (algorithme 14.6 dans [Aho 77]).

Nous donnons en exemple les temps de vie des variables de *movs*. Néanmoins, ce travail doit être effectué pour toutes les variables intermédiaires générées par le séquencement de toutes les opérations.

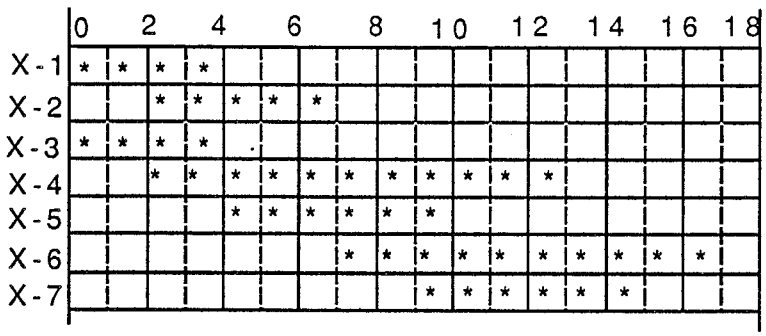
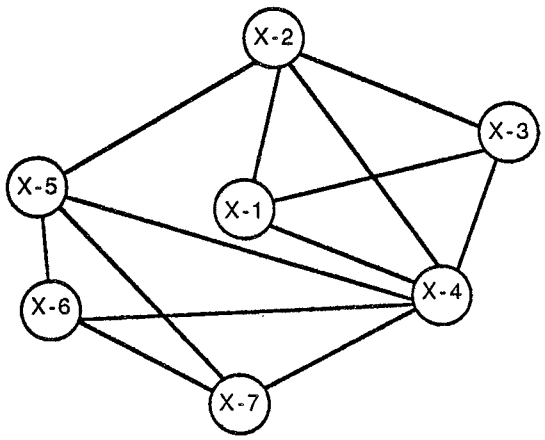


tableau des temps de vie pour l'instruction movs

A partir du tableau des temps de vie des variables, on peut construire le graphe de compatibilité des variables. Nous représentons ci-dessous le graphe correspondant à notre exemple. Les nœuds du graphe représentent les différentes variables et les arcs relient les variables qui ne peuvent pas partager un registre.



graphe de compatibilité des registres pour MOVS

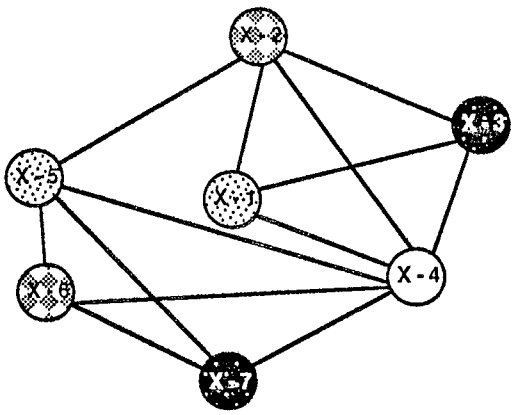
Pour chacune des instructions, on peut construire un graphe similaire. Comme les instructions ne peuvent pas être exécutées en parallèle, les différents graphes obtenus seront déconnectés.

6.9.2. partage des registres par coloration du graphe de compatibilité

Si à chaque registre on affecte une couleur, le problème se pose comme un problème de coloration de graphe : il faut colorier le graphe sans que deux sommets adjacents soient de la même couleur. Le nombre de registres nécessaires pour implémenter nos instructions est le nombre chromatique du graphe . On retrouve ici la méthode d'allocation de registres par coloration de graphes décrite dans [Chai 82].

L'algorithme que nous utilisons est en fait un algorithme de partitionnement en cliques disjointes d'un graphe, décrit plus loin. Nous utilisons le fait que le problème de coloriage d'un graphe G est équivalent au partitionnement en cliques disjointes du graphe dual G'.

Nous représentons ci-dessous une solution de coloration du graphe de compatibilité de *movs*.



graphe de compatibilité des registres pour MOVS

Une fois les variables d'une instruction regroupées en registres, on peut réécrire l'instruction en utilisant les noms des registres effectifs. L'instruction *movs* devient :

```
(1)    R1:=OLDCO+1;
(2)    R3:=J+J;
(3)    R2:=R1+R_INDEX;
(4)    R4:=AD+R3;
(5)    D1:=access(R2);
(6)    R1:=R4+1;
(7)    R2:=R4+D1;
(8)    GR(2):=access(R1);
(9)    R3:=R2+1;
(10)   GR(1):=access(R4);
(11)   send(GR(2),R3);
(12)   send(GR(1),R2);
```

séquence réécrite pour MOVS

Les chiffres entre parenthèses qui précèdent chaque instruction sont les étiquettes des instructions. Ces étiquettes seront utilisées dans la suite.

Résultat : A l'issue de la phase de partage de registres, on obtient une liste d'instructions, toutes datées par rapport au début de leur séquence, et dont les opérations sont des opérations dyadiques simples, et dont les variables sont soit les variables définies dans la description source, soit des registres réels.

6.10. la génération de la partie opérative

6.10.1. à partir de quoi générer la PO

Les phases d'ordonnancement et de simplifications de fonctions ont transformé la description initiale en une série de séquences d'opérations élémentaires. L'allocation de registres a transformé les variables initiales en une série restreinte de registres. On dispose donc d'une série de séquences d'opérations élémentaires. En compilation de langages de programmation

on appellerait cela le code trois adresses, ici nous appelerons ces opérations "transfert de registres", au sens premier du terme.

```
<séquence 1>
REG1 <- REGA + REGB
<séquence 2>
REGA <- REG6
REGB <- 1
```

Pour générer la partie opérative, il reste à effectuer les trois tâches suivantes :

1/ déterminer le nombre d'unités fonctionnelles nécessaires pour pouvoir effectuer les opérations des différentes séquences, et les choisir : sélection des modules.

2/ déterminer la manière dont seront acheminées les données des registres aux unités fonctionnelles, et vice-versa. Cette phase est très différente selon qu'on a choisi un style distribué ou un style bus.

3/ calculer pour chaque séquence d'opérations simples le vecteur de contrôle de la PO, c'est-à-dire l'ensemble des commandes à activer (interrupteurs, commandes d'UAL, ouverture de registres) pour que la PO réalise la séquence.

Comme pour le calcul du séquencement, on peut distinguer deux familles de méthodes pour calculer une partie opérative : les méthodes globales et les méthodes de construction itérative.

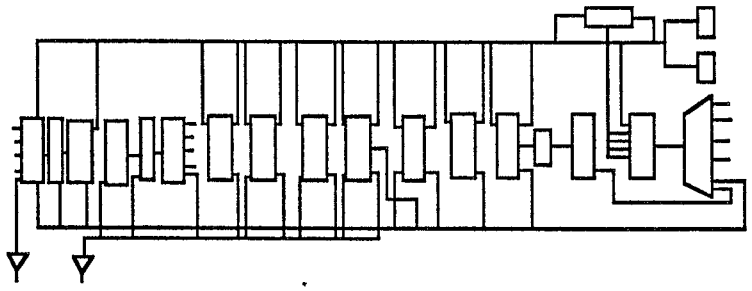
6.10.2. style distribué ou style bus ?

La question se pose si on veut pouvoir effectuer plus d'une opération en parallèle, sinon une simple ALU suffit. Le **style distribué** consiste à disperser des unités fonctionnelles parmi les registres, ce qui amène à dupliquer les opérateurs et les chemins de données. Ce style est obligatoire pour le haut parallélisme, par exemple pour le traitement du signal. Le **style bus** consiste à organiser les transferts de données en les regroupant en bus. Ceci permet d'économiser beaucoup de place car (1) la méthode permet une implantation régulière, (2) il y a moins d'opérateurs, moins de

place utilisée, moins de multiplexeurs et d'unités d'interconnexions. Mais cette méthode pose des problèmes de partage des bus et des opérateurs, la machine obtenue est moins rapide et son contrôle est plus complexe. C'est le style choisi pour la plupart des micro-processeurs génériques actuels (iAPX, 680xx, etc...).

6.10.3. synthèse de PO style Bus

Le style bus se retrouve principalement dans les microprocesseurs et dans les processeurs. L'archétype du genre est la PO à 2 bus du microprocesseur Motorola MC6800.



PO du MC6800 (d'après [Obr 82])

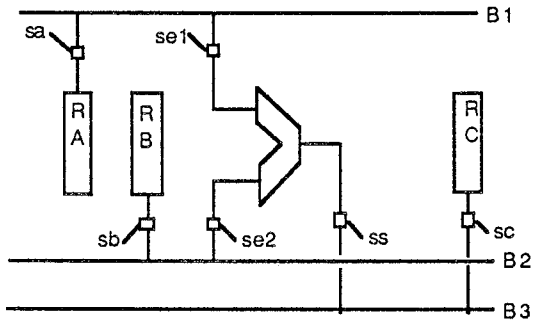
Dans un paragraphe suivant, nous décrivons la génération de partie opératives style bus, avec des hypothèses simples sur le comportement des bus. La suite de ce paragraphe en décrit simplement le fonctionnement.

Le principe du style bus est d'utiliser le moins possible de passages pour les données. Sur les 2 (ou 3 ou 4) chemins passeront les unes après les autres toutes les données. Les données seront dirigées au moyen d'interrupteurs vers les registres ou les opérateurs auxquels elles sont destinées.

La PO à bus non segmenté est le modèle le plus simple. Par exemple, on cherche à réaliser l'opération

```
RC <- RA + RB
```

sur la PO à bus non segmentés suivante :



PO style bus simplifiée

Les interrupteurs représentés sur la figure représentent en fait deux interrupteurs dans le cas des registres : l'un pour la lecture et l'autre pour l'écriture. Nous les distinguons en notant par exemple saR et saW les interrupteurs du registre RA.

- On "ouvre" les interrupteurs saR et sbR, ce qui a pour effet de donner au bus B1 la valeur de RA et au bus B2 la valeur de RB.

- On ouvre les interrupteurs se1 et se2 : les opérandes en entrée de l'additionneur seront les valeurs contenues dans B1 et B2, donc les valeurs de RA et RB.

- On ouvre l'interrupteur de sortie ss de l'additionneur. La valeur portée par le bus B3 sera la somme des opérandes en entrée de l'additionneur, soit RA + RB.

- Enfin, on ouvre l'interrupteur sc qui amène le contenu de B3 à RC, et on charge RC en ouvrant l'interrupteur scW.

Finalement, le vecteur de commandes pour cette opération sera :

```
saR = sbR= se1= se2= ss= scW= vRC= 1
```

Dans le cas d'un additionneur tamponné, c'est-à-dire que soit ses entrées, soit ses sorties sont gérées par un tampon, on décalera les

opérations saR= sbR= se1 = se2 = 1 et scW = vRC = ss = 1 d'une phase d'horloge.

Un grand nombre de bus peut être très coûteux en surface sur le cristal. Une manière d'augmenter le nombre de bus sur une surface donnée tout en autorisant beaucoup de transferts consiste à découper les bus en sous-bus indépendant au moyen d'interrupteurs. C'est le principe des parties opératives à bus segmentés. La segmentation des bus permet une implantation régulière. En fait, on peut considérer une PO à bus segmentés comme une juxtaposition de PO à bus non segmentés.

Le comportement électrique des bus ainsi que les parties opératives segmentées sont décrites dans [Anc 86a].

Sans entrer plus dans les détails, nous énumérons rapidement les problèmes qui se posent, et nous donnons certaines des solutions qui y ont été apportées.

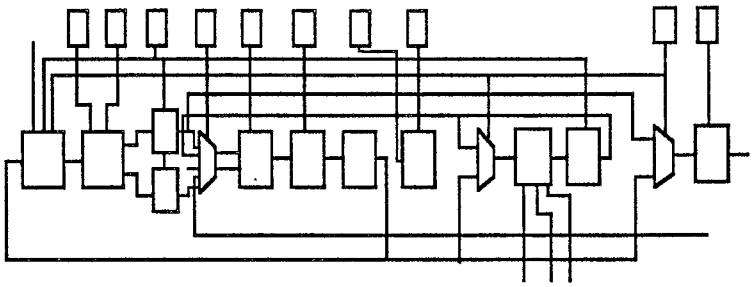
* déterminer le nombre de bus nécessaires
Torng et Wilhelm [Tor 77] utilisent la technique de génération et de test en programmation dynamique, mais cette technique est très coûteuse en temps de calcul. Tseng et Sieworek ([Tse 81]) créent les bus un par un en essayant d'affecter le plus de transferts possibles à un bus, sans introduire de délai ni réduire le parallélisme. A la différence de ces deux méthodes qui sont typiquement constructives, Facet utilise une méthode globale fondé sur une méthode de traitement de graphes. Nous présentons une méthode qui consiste à assigner et à regrouper les opérations de transferts en fonction de leur comptabilités et de leurs similitudes, en utilisant un algorithme de regroupements en cliques. Certains systèmes résolvent le problème en limitant le nombre de bus à deux. C'est le cas par exemple de *Bristle blocks*. *SYCO* génère également des parties opératives à deux bus segmentés.

* déterminer le nombre de registres (problème abordé indépendamment plus haut).

* transformer chaque transfert (registre vers élément fonctionnel et élément fonctionnel vers registre) en un mot de commande.

6.10.4. synthèse de PO distribuées

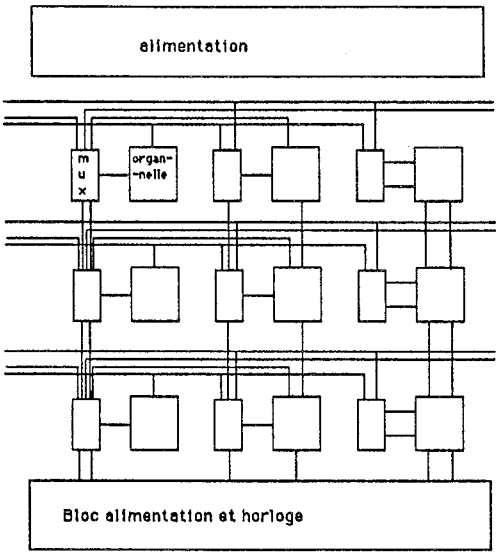
A titre de comparaison, nous montrons une vue simplifiée du microprocesseur Am2901, représentée ci-dessous au même niveau de détail que le 68000 précédent :



PO de l'AM2901 (d'après [Row 86])

C'est un style qui est obligatoire pour le haut parallélisme :

Le problème qui se pose n'est plus le partage des bus mais le regroupement des interconnexions. Certains systèmes ne s'en soucient pas, comme par exemple *Mac Pitts*. *Mac Pitts* ([Sis 82]) est destiné à la synthèse de circuits de traitement du signal. Il est orienté vers un haut degré de parallélisme. Son architecture cible est composée d'un grand nombre de registres et d'opérateurs distribués de manière "bit-slice". Ces composants sont appelés "Organelles". Les interconnexions sont assurées par des bus locaux : il en existe un par organelle.



organisation d' une partie opérative générée par Mac Pitts

D'autre systèmes créent des architectures distribuées. *EMUCS* ([Tse 81], [Tho 83]) est un système de la famille *CMU-DA*. Comme *FACET*, *EMUCS* synthétise des descriptions écrites en *ISPS* en utilisant la forme intermédiaire Value-Trace. Le programme affecte incrémentalement des blocs matériels aux nœuds du graphe data-flow, en se basant à chaque étape sur des estimations de coût. Le circuit obtenu est naturellement distribué, mais le concepteur peut introduire interactivement des bus s'il le désire.

6.10.5. Les autres styles de machines

La synthèse de machines à haut degré de parallélisme (systoliques, vectorielles ...) sort de notre cadre. En effet, la conception part rarement d'une description dans un langage de description de matériel de haut niveau.

6.10.6. la sélection des modules

3 méthodes sont possibles quand on veut choisir les modules qui composent effectivement la partie opérative.

1- utiliser des modules fonctionnels complètement spécifiés : c'est la solution adoptée par exemple par *Mac Pitts*.

2- dessiner des modules nouveaux.

3- instancier une structure prédéfinie : c'est la solution adoptée par exemple par *SYCO*. Certains auteurs utilisent une représentation par "schéma" ([Sai 86]) ou bien "orientée objet" (c'est un aspect important de *HAL* [Pau 86]) pour classifier leurs modules. Ceci permet un processus d'instanciation efficace.

6.11. illustration : synthèse de PO utilisant le partitionnement en cliques

A l'issue de la phase de partage des registres, on connait les éléments suivant :

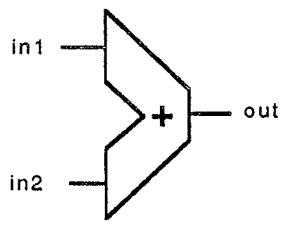
- une série de séquences d'instructions simples, datées par rapport au début de la séquence.

- le nombre de registres, généraux (ceux qui proviennent de la phase de partage des registres) ou bien déclarés par l'utilisateur, à implanter.

- le nombre d'opérateurs à implanter.

6.11.1. assignement des opérateurs

Pour pouvoir générer une partie opérative, il faut tout d'abord transformer toutes les opérations simples en tranferts de registre au sens strict du terme, c'est-à-dire soit en transferts de registre à registre, soit en transferts de registre à opérateur, soit en transferts d'opérateur à registre. Ceci ne peut être fait qu'après la phase de partage des registres. Par exemple, si on dispose de l'additionneur non tamponné représenté ci-dessous :



on traduira l'instruction suivante :

```
REG1 := REGA + REGB ;
```

...par la séquence de transferts suivante :

```
in1 <- REGA
in2 <- REGB
REG1 <- out
```

Cette traduction est assez simple : le principal problème est de choisir quel opérateur affecter à une instruction donnée. La réécriture se fait ensuite simplement, et dépend néanmoins de la façon dont sont tamponnées les entrées ou les sorties des opérateurs.

Nous cherchons à affecter des opérations comportant des variables communes à un même opérateur. Dans notre cas, nous n'avons qu'un opérateur d'entrée/sortie et deux opérateurs d'addition. La question ne se pose donc que pour les opérations d'addition.

Nous utilisons une méthode fondée sur le **partitionnement d'un graphe en cliques disjointes.**

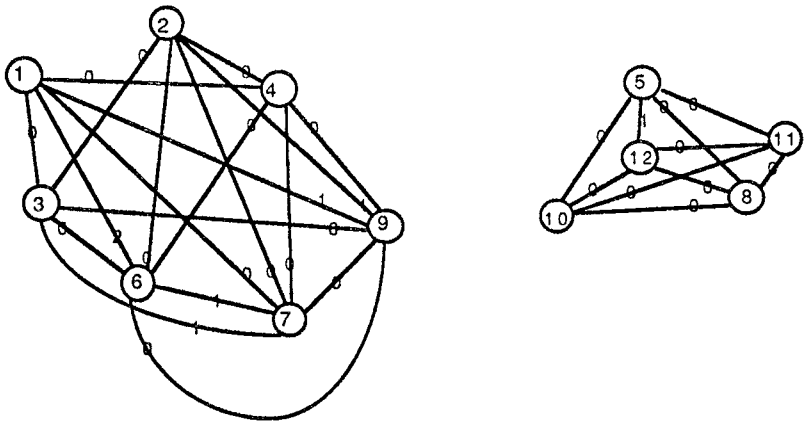
Cette méthode est très proche de la méthode utilisée dans *FACET*. C'est une méthode globale, et elle donne de meilleurs résultats que des programmes itératifs en PROLOG que nous avons expérimenté pour résoudre le même problème. Pratiquement, elle utilise le même programme de partitionnement en cliques que la phase de partage des registres et que la phase de regroupement des bus.

On retrouve une méthode très proche dans *Sehwa* ([NPar 88]) : ce système de synthèse de circuits pipe-line utilise la méthode de coloration des nœuds d'un graphe pour assigner ses opérateurs.

Une clique est un sous-graphe d'un graphe G dont tous les nœuds sont reliés deux à deux. Partitionner un graphe G en cliques, c'est le découper en sous-graphes disjoints dont tous les nœuds sont reliés.

On construit un graphe dont les nœuds sont les étiquettes des différentes opérations. On relie entre eux les nœuds représentant des opérations compatibles, c'est-à-dire non simultanées. Partitionner ce graphe en cliques

disjointes revient à regrouper les opérations compatibles, et qui peuvent donc partager un même opérateur. Néanmoins, si on effectue le partitionnement sans autres critères, le résultat a peu de chances d'être bon : les opérations seront regroupées quasiment au hasard. On cherche à regrouper au maximum les opérations qui se ressemblent. Pour cela nous affectons aux arcs du graphe un coefficient de ressemblance. Ce coefficient est calculé en sommant le nombre de variables communes en partie gauche (1 ou 0) et le nombre de variables communes en partie droite (0, 1 ou 2). Nous donnons ci-dessous le graphe correspondant à l'instruction movs. Les nœuds sont étiquetés conformément à la "réécriture de la séquence de MOVS".



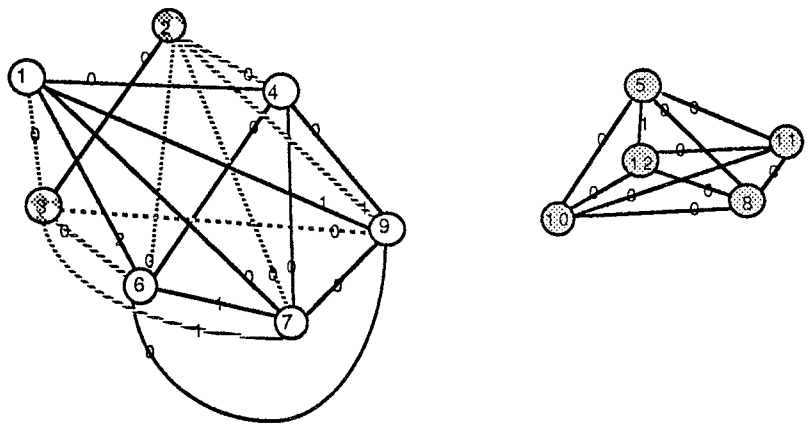
graphe de compatibilité des opérateurs de movs

L'algorithme de partitionnement en cliques disjointes doit pouvoir prendre en compte les coefficients affectés aux arcs et privilégier les regroupements entre arcs aux coefficients les plus élevés. Nous donnons ci-dessous l'algorithme utilisé. Cet algorithme diffère nettement de celui de FACET.

Notations : on note $(p,q)_k$ l'arc de catégorie k reliant les nœuds p et q.

```
début
   pourtout k de k_max à 1
      mettre à jour la liste des voisins communs dans G_k
      pourtout arc (p,q)_k qui a le maximum de voisins communs
         tantque p apparait dans G_k faire
            pourtout nœud r connecté à p oubien q faire
               supprimer (p,r) oubien (q,r) de G et de G_k
            finfaire
            pourtout nœud r connecté à p et r faire
               supprimer (q,r) de G
               si (q,r)_j ε G_k, supprimer (q,r) de G_k
               (p,r)i devient (p,r)_k, et (p,r)_k ⊂ G_k
            finfaire
            mettre à jour la liste des voisins communs dans G_k
         fintantque
      finfaire
   finfaire
fin
```

le résultat du partitionnement du graphe de movs est le suivant :



partitions du graphe de compatibilité des opérateurs de movs

Le fait de connaître l'opérateur à utiliser ne suffit pas pour traduire les opérations en transferts : il reste à déterminer quel opérande est attaché à quelle entrée de l'opérateur. Nous utilisons pour cela un algorithme très simple qui affecte une variable en priorité à l'entrée à laquelle elle a déjà été affectée.

On dispose à l'issue de cette phase d'une liste de séquences de transferts de registres datés par rapport au début de chaque séquence.

6.11.2. regroupement des bus

Nous connaissons les registres et les opérateurs que notre partie opérative doit comprendre. La sélection des modules effectifs n'est pas un problème dans le cas de notre synthèse simpliste : les seuls modules utilisés sont l'additionneur deux entrées, le module d'entrées/sorties et le registre.

Le seul problème qu'il reste à résoudre est de déterminer la connectique de ces différents éléments, c'est-à-dire :

- déterminer le nombre de bus nécessaires.

- fixer les connexions bus/registre.

- en déduire les séquences de "vecteurs de commandes" à utiliser pour chaque instruction.

Nous utilisons une méthode globale comportant trois étapes :

1- assigner un bus différent à chaque transfert,

2- regrouper les bus dont les temps d'occupation sont disjoints,

3- réécrire les transferts pour obtenir les vecteurs de commandes des séquences.

L'étape 1 ne pose pas de problèmes. A titre d'exemple, nous illustrons ci-dessous l'instruction movs après traitement par la phase 1. Les transferts sont regroupés en fonction de leur date. inA1, inA2, inB1 et inB2 symbolisent les entrées des additionneurs A et B. Leurs sorties sont nommées outA et outB. Les ports du module d'entrées/sorties sont appelées IODAT pour le port de données et IOADR pour l'entrée d'adresse.

```
date 0:
        J -> inB1
        J -> inB2
        OLDCO -> inA1
        1 -> inA2
date 1:
        outB -> R1
        outA -> R3
        etc ...
```

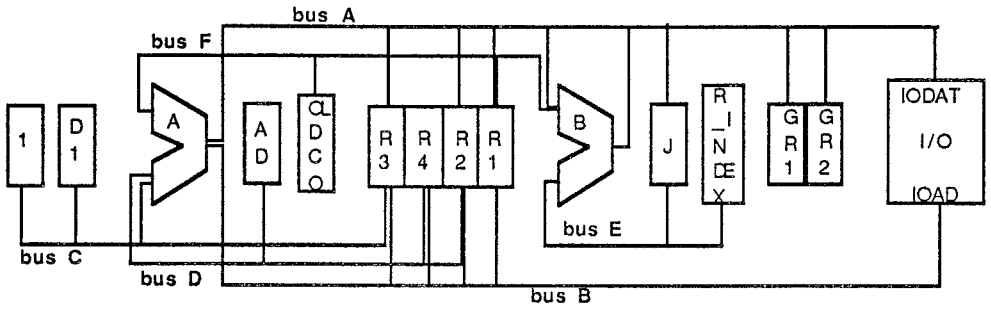
La séquence est reproduite "in extenso" dans les annexes.

Nous détaillons ci-dessous la phase de regroupement des bus. On peut remarquer que si on l'omet, on obtient une architecture distribuée, même si elle est peu efficace. Cette phase commence par la construction du graphe de compatibilité des bus. Dans notre cas, les données qui transitent sur un bus n'y restent qu'un cycle, sauf pour les données servant d'adresse pour les opérations d'entrée/sortie.

Comme pour le regroupement des opérateurs, le simple partitionnement du graphe de compatibilité des bus ne suffit pas. L'algorithme trouvera un nombre intéressant de bus, mais chacun des registres et des entrées ou sorties d'opérateurs sera connecté quasiment à tous les bus. Cela multipliera d'autant le nombre d'interrupteurs, et cela rendra la partie opérative difficile à implanter. Nous utilisons donc l'algorithme de partitionnement modifié.

Comme le but est de regrouper les bus concernant les mêmes variables, on cherche à favoriser les transferts qui ont le maximum de variables communes : pour cela on affecte à chaque arc le nombre de variables communes entre les deux transferts. En fait, à l'expérimentation, on s'aperçoit qu'il est plus intéressant de favoriser le regroupement des opérations d'entrées/sorties, pour lesquelles il n'existe qu'un module plutôt que les opérations d'additions. On affecte donc à ces opérations les poids 0, 2, 3 selon qu'elles ont 0, 1 ou 2 variables communes.

Le graphe que l'on construit **uniquement** à partir des transferts de *movs* contient 31 nœuds et 409 arcs. Son partitionnement prend **16 minutes** sur un Hewlett Packard Vectra ("compatible PC"). Le graphe est partitionné en 6 cliques, on obtient donc une partie opérative à 6 bus, reproduite ci-dessous. Les registres R1, R2, R3 et R4 sont les registres créés par la phase de partage des registres.



partie opérative à bus non segmenté

On obtient manuellement une partie opérative comparable, et même meilleure au niveau des entrées des additionneurs. Mais dans ce cas il faut vérifier que tous les transferts sont possibles.

6.11.3. quelques améliorations de la méthode

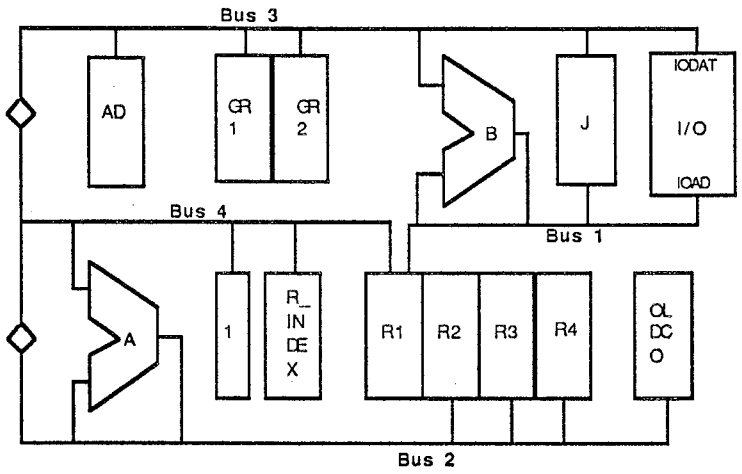
a) La partie opérative ci-dessus a été obtenue à partir du graphe des transferts de *movs* seulement. Le graphe complet de tous les transferts de toutes les séquences est plus important que celui de *movs*. En effet, comme chaque transfert d'une séquence donnée est compatible avec tous les transferts de toutes les autres séquences, on obtient un graphe contenant de l'ordre de n! arcs, n étant le nombre de transferts total.

Il existe une alternative à la construction de graphes importants : on peut construire la partie opérative à partir de l'instruction la plus complexe, et ensuite utiliser un autre programme, d'une part pour vérifier que la partie opérative obtenue peut exécuter toutes les séquences, et d'autre part pour calculer les vecteurs de commandes. Ce faisant, on réduit beaucoup les temps de traitement mais on perd l'aspect global du traitement. Une méthode et un programme sont décrits dans [Dur 86a]. L'idée a été pleinement développée depuis 1983 par F. Anceau et son équipe ([Anc 87b]).

b) la partie opérative obtenue ci-dessus sera certainement difficile à implanter : elle contient beaucoup de croisements de bus, un des opérateurs est connecté à cinq bus différents, etc... plutôt que d'essayer d'arranger encore notre méthode, nous cherchons à créer une partie opérative à bus segmentés.

Nous avons expérimenté cette méthode de manière "semi-manuelle", en nous servant des programmes déjà écrits : c'est pourquoi nous n'entrons pas dans le détail. Le résultat est tout de même intéressant puisqu'on arrive à une partie opérative beaucoup plus simple.

Pour cela on commence par affecter des bus aux bornes des opérateurs, par la méthode décrite dans le paragraphe 6-11.1. Ensuite on détermine les connections obligées en identifiant les instants critiques, c'est-à-dire les moments où tous les bus sont occupés. On utilise ensuite une méthode de construction itérative pour affecter les transferts restant (c'est-à-dire ceux qui ne sont ni critiques, ni déjà résolus) aux bus encore libres. Nous avons pu trouver une solution pour notre exemple :



On peut envisager d'autres améliorations que l'introduction d'interrupteurs, par exemple l'utilisation de registres transparents ...

6.12. notes d'implémentation

L'expérience décrite dans ce chapitre a été réalisée sur IBM-PC XT, pour des raisons de disponibilité. Les programmes ont été écrits en Turbo-Pascal et en Prolog-Criss.

Le format d'entrée est une simplification du langage LASCAR, et le format de sortie est le langage POLO.

Les programmes effectuent des traductions instructions/graphes, graphes/instructions ou graphe/graphe. Les échanges se font à travers deux formats très simples, l'un pour les instructions, l'autre pour les graphes. Cette simplification permet de construire très rapidement des programmes.

Si l'expérience avait été poursuivie, il aurait fallu transférer tous les programmes sur une machine beaucoup plus puissante. Les programmes de partitionnement en cliques amènent rapidement les "compatibles PC" à leurs limites, en mémoire et surtout en temps d'exécution (en fait, on peut supposer que la taille limitée de la mémoire dégrade beaucoup la vitesse d'exécution). Le regroupement des bus de l'instruction *movs* prend 16 minutes. Certains graphes ont parfois demandé quatre heures de calcul, ce qui excède quelquefois le temps moyen de bon fonctionnement de certaines machines.

6.13. conclusion de l'expérience

Le but de cette expérience était de définir une spécification pour un compilateur de silicium, dans le cadre d'une coopération avec Bull-Systèmes. L'orientation "synthèse de partie opérative" a été abandonnée, car convenant mal à l'environnement Bull-Systèmes, pour une approche plus "synthèse logique".

Avant de décliner les faiblesses, les qualités et les enseignements de l'expérience, on peut noter qu'elle met en valeur des ressemblances profondes qui existent entre de nombreux projets similaires.

6.13.1. Les faiblesses...

Notre expérience a de nombreux points faibles : certains sont dûs à son approche "exploratoire" qui nous a amené à faire beaucoup d'hypothèses simplificatrices afin de ne pas alourdir les programmes. A côté de cette faiblesse de réalisation, on trouve des défauts plus intéressants qui sont des faiblesses dans la méthode. Nous commencerons par détailler les premières

* Nous avons ignoré la compilation du séquenceur qui gère les branchement et génère les vecteurs de contrôle. C'est un problème indépendant, qui est abondamment traité dans la littérature (voir chapitre suivant).

*Nous n'avons pris en compte que quatre opérations différentes, qui comportent au maximum deux opérandes.

*Nous avons fixé les comportements temporels des opérateurs, et nous avons choisi un modèle simpliste. Notre modèle est beaucoup plus simple que celui de *SYCO*, qui comporte quatre phases : précharge, lecture, amplification, écriture.

*Le cas des instructions conditionnées n'est pas abordé pour la synthèse de parties opératives. On peut envisager deux solutions simples : soit implémenter le calcul de la condition dans la partie opérative, et donc modifier le séquencement et le contrôle de l'instruction, pour laisser le temps au séquenceur d'interagir avec la partie opérative, soit inclure le calcul de la condition dans le

contrôle, qui contiendra alors une sous-partie opérative de calcul des conditions.

*Le calcul de la partie opérative est un traitement global pour l'analyse des temps de vie et le partage des registres mais local pour ce qui est assignement des opérateurs et partage des bus : cette limitation est due aux tailles des graphes à considérer pour un traitement global.

On peut espérer résoudre ces points en effectuant une analyse plus complète du problème. Il existe par contre des points qui impliquent des changements profonds dans la méthode. Ces points posent des problèmes difficiles, et ils sont rarement résolus dans la littérature.

*Nous supposons que nos tâches ont une durée fixe, même les entrées-sorties. Les micro-processeurs complexes ont maintenant des mémoires caches intégrées, et cette hypothèse paraît maintenant simpliste. Mais prendre en compte des durées variables complique beaucoup le travail d'ordonnancement.

*Notre méthode de séquencement travaille instruction par instruction. Ceci la limite aux modèles sans parallélisme, ou aux modèles partitionnés explicitement en sous-parties opératives sans parallélisme, ce qui revient au même. Mais la généralisation du problème est un ordonnancement non-simple et donc beaucoup plus complexe.

Ces deux derniers points posent une question de fond : est-il possible de créer des machines raisonnablement parallèles sans aucune spécification du parallélisme par l'utilisateur ? Par exemple, la modélisation d'entrées-sorties tamponnées oblige à préciser les parallélismes entre instructions.

6.13.2. ...l'intérêt...

Cette expérience a fait apparaître une grande similitude entre différents projets de synthèse procédurale, surtout au niveau de la génération du séquencement. Elle a permis de révéler les difficultés de la synthèse de parties opératives de type bus, et les points où il faut particulariser sa méthode pour obtenir des résultats :

- génération du séquencement : tel que nous l'avons abordé, le problème est un problème de séquencement. Il devient très complexe dès que l'on introduit des possibilités de parallélisme et des incertitudes sur les durées des différentes tâches. C'est pourtant souvent le cas, la plupart des mémoires étant tamponnées.

- partage des registres : trouver un nombre optimal de registres n'est pas le problème crucial. Le problème principal est d'affecter les variables aux registres. Ce problème est indissociable du problème de l'affectation des opérateurs et des bus.

- partage des bus. Ce problème, déjà complexe avec des bus non segmentés, devient encore plus difficile avec des bus segmentés. Il faut donc utiliser dès le début un modèle efficace (sans être trop restrictif).

Finalement, l'expérience montre qu'il est possible de créer des parties opératives acceptables pour des opérations complexes et ceci sans trop de restrictions architecturales. Elle décrit un ensemble de méthodes de base pour chacun des problèmes qui se posent. Elle montre la faiblesse de la résolution point par point de ces problèmes et la nécessité d'une stratégie globale.

Chapitre 7

SYNTHESE DE PARTIE CONTROLE

7-SYNTHESE DE PARTIE CONTROLE

7.1. point de départ et représentation du problème

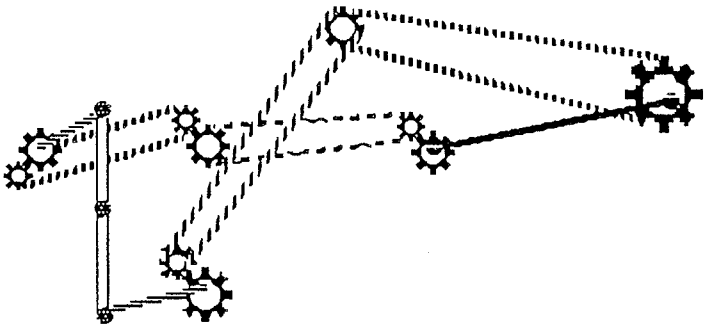


figure 7-1 : automate

Ce chapitre traite du problème particulier de la synthèse d'une partie contrôle décrite par un automate explicite. Ce problème est à distinguer de la synthèse de séquencement à partir d'un algorithme, ou du calcul de mots de contrôle pour une partie opérative. Ces deux problèmes ont par ailleurs été abordés dans les deux chapitres précédents. Il est à distinguer également de la synthèse d'automates d'états finis à partir des suites de valeurs (entrée,état,sorties) que nous n'aborderons pas.

Que reste-t-il ? Nous présenterons rapidement la synthèse d'**automates d'états finis** telle qu'elle est le plus souvent décrite dans la littérature. Nous ne l'illustrerons pas: on trouve dans les chapitres 3 et 4 deux exemples de synthèse d'automates simples. Nous aborderons également la synthèse de **graphes de contrôles** tels qu'ils sont définis en LASSO, dont les critères sont très différents.

La "description explicite" d'un automate est une notion vague. Il existe une définition formelle précise des automates d'états finis (automates de Moore, automates de Mealy) mais elle nous est de peu de secours ici. La synthèse nécessite une meilleure caractérisation de la description du contrôle dans les langages de description de matériel. Nous allons employer pour cela le vocabulaire Conlan [Conlan 85], sans restreindre la généralité du propos. Conlan est une base formelle qui permet la définition de langages de description de matériel. Notre "description explicite" rentre dans le cadre de la modélisation par transitions d'états controllées

("controlled state transitions"), qui est un des modes fondamentaux de description comportementale. Les variables, appelées "porteuses"[1], sont divisées en "porteuses de contrôle" et "porteuses de données". L'état des porteuses de contrôle détermine quand une instruction[2] doit être évaluée. Les instructions correspondant à une même porteuse de contrôle sont regroupées en blocs. En général, ces blocs sont indiqués par des notations du type étiquettes[3]. Le séquencement est exprimé par des instructions spéciales (par exemple GOTO, FORK, NEXT, EXECUTE, etc...).

On distingue la sémantique du comportement des instructions à l'intérieur des blocs, et la sémantique de l'activation des blocs. Nous définissons donc deux critères, 1/ le séquencement des opérations dans un même bloc et 2/ le séquencement des blocs. Il existe un troisième niveau, qui est la sémantique des ensembles de blocs, ce que nous appelons description. Mais comme nous considérons ici la synthèse des descriptions, nous ne le prenons pas en compte.

7.1.1. premier critère : séquencement des opérations dans un même bloc

langage procédural: les opérations élémentaires d'un même bloc sont séquentielles. Exemples: LDS, ISPS, LASSO.

langage non procédural: les opérations élémentaires d'un même bloc sont parallèles. Exemples: LASCAR, CASSANDRE.

7.1.2. second critère: activation des blocs

formalisme d'automate: un seul bloc actif à la fois. Chaque bloc contient l'activation de son successeur. Exemples: LDS, LASCAR.

formalisme parallèle: un ou plusieurs blocs actifs à la fois. Leurs activations sont déclenchées par des événements. Il existe de nombreuses variantes suivant qu'il y a retour ou non à l'instruction d'activation, si l'activation est bloquante ou non... Exemple: LASSO...

---

[1] la notion de porteuse est en fait plus complexe que cela. Comme le propos n'est pas de décrire Conlan ici, nous nous en tiendrons à cette vision reductrice...

[2] pour rester dans le vocabulaire Conlan, il serait plus juste d'employer le terme "activité"

[3] ce que nous décrivons ici est très proche de la notion d'"instruction gardée".

Dans tous les cas, concrètement, l'entrée est une suite d'opérations élémentaires affectées d'étiquettes. La sémantique des étiquettes dépend du formalisme choisi, et du langage utilisé.

7.2. différents types de synthèses d'automates d'états finis

Nous abordons ici la synthèse des automates décrits par des langages non procéduraux dans un formalisme d'automate. En général, un automate contient deux sortes d'actions: des actions qui agissent sur la partie opérative, que nous appelerons "opérations", et des actions qui agissent sur le contrôle, c'est-à-dire sur le fonctionnement de l'automate lui-même, et que nous appelerons "transitions".

Pour pouvoir synthétiser l'automate, il faut que les opérations soient définies de manière précise. Ceci implique que la partie opérative soit presque complètement définie. Pour chaque opération, nous pouvons donc calculer le "vecteur de commandes", qui devra être généré par la partie contrôle.

La difficulté de la compilation de parties contrôle se trouve dans la synthèse du matériel qui réalise les transitions, appelé *contrôleur*. Il existe principalement trois architectures de contrôleurs: les contrôleurs microprogrammés, les contrôleurs en PLA et les contrôleurs en logique sauvage.

Un **contrôleur en PLA** complexe comme celui du 8048 est composé de trois parties: un dispositif qui élabore les commandes à activer (PLA de propriétés), un autre qui délivre les instants définissant le temps (PLA de génération) et un troisième qui valide les commandes (PLA de validation) [Ber 85]. La synthèse pose surtout un problème de place sur le cristal. On cherche à diminuer le nombre d'états de l'algorithme par recherche des états semblables, à optimiser les équations booléennes et à répartir au mieux les sorties pour que la place consommée par le dispositif soit minimale.

La synthèse de **parties contrôle micro-programmées** recouvre à la fois le choix d'un contrôleur (taille des champs, type de séquencement) et la génération du micro-code pour ce contrôleur. On cherche à rassembler les micro-opérations pour en faire des micro-fonctions correspondant *grosso modo* à un même champ du mot d'instruction. Classiquement, un contrôleur microprogrammé fonctionne en deux phases: une pour les transitions du contrôleur, une pour les opérations de la PO [Dir 81].

Pour les automates de taille modeste, on utilise des **contrôleurs en logique sauvage.** La synthèse d'automates séquentiels d'états finis comporte 3 phases [Koh 70]:

- fixer un assignement d'états aux éléments de mémorisation

- en dériver les tables de transition ainsi que les tables de sortie des éléments de mémorisation

- en déduire la table d'excitation, puis les fonctions de sortie et d'excitation.

Il y a deux manières pour fixer un assignement d'états, c'est à dire de fixer comment seront codés les états sur les registres d'états: on peut soit effectuer un codage trivial, c'est-à-dire utiliser autant de registres qu'il y a d'états. On peut également effectuer un codage canonique, c'est à dire utilisant $Log_2(n)$ registres pour coder n états [Mer 73]. Nous avons utilisé l'une et l'autre méthodes: la première est décrite dans le chapitre 4, l'autre dans le chapitre 5. Tout n'est pas dit ici. En particulier, la qualité d'un codage canonique varie beaucoup suivant que l'on assigne les états au hasard ou si on les assigne de manière à minimiser ensuite la logique des fonctions d'excitation. Ce point est détaillé dans [Han 86].

Certaines optimisations sont intéressantes pour tous les types de synthèse. Par exemple, on a toujours intérêt à diminuer le nombre d'états de l'automate. En effet, la surface des automates grandit de manière non linéaire quand on ajoute de nouvelles transitions et de nouveaux états.

La plupart des contrôleurs que l'on compile sont des contrôleurs *synchrones*, c'est-à-dire que les transitions d'états sont cadencées (ou échantillonnées) par un signal d'horloge. La compilation d'automates *asynchrones* pose des problèmes supplémentaires à cause des possibilités d'états incorrects dûs aux non-simultanéités des changements des entrées.

Nous n'entrerons pas plus dans le détail de ce problème complexe, qui a fait l'objet de nombreux travaux. Deux exemples de synthèse d'automates ont été abordés dans les chapitres antérieurs.

7.3. synthèse d'automates parallèles

Le fonctionnement d'un système parallèle[1] est difficile à décrire en raison du nombre d'états potentiels de tels systèmes. De plus, la description des mécanismes de synchronisation et de gestion des conflits en rendent difficile la description sous forme d'automates d'états finis.

Un certain nombre de langages sont mieux adaptés à la description de systèmes parallèles : citons le formalisme CSP de Hoare, SIMULA, LDS dans une certaine mesure, etc... Nous nous limiterons aux langages possédant une sémantique proche des réseaux de Petri : réseaux de Petri colorés, GRAFCET, CAP/DSL (DACAPO), LASSO... car ce sont les plus utilisés pour la synthèse [Oli 83], [Brü 86], [Ova 87]. Néanmoins, certains systèmes de synthèse utilisent des notations de type algébrique du style de CSP [Mar 87].

Il est intéressant d'utiliser les réseaux de Petri quand on veut décrire un contrôle décentralisé, sans avoir à préciser explicitement tous les mécanismes de synchronisation.

On trouvera par exemple dans [Ova 87] la modélisation d'un protocole de communication à priorité cyclique, décrit dans le formalisme des réseaux de Petri et en LASSO. Plus proche de notre problème, R. R. Razouk décrit dans [Raz 88] la modélisation de processeurs pipe-line en réseaux de Petri.

7.3.1. synthèse à partir d'un réseau de PETRI

Nous parlons ici de réseaux de Petri au sens large : les réseaux de Petri proprement dits, et les réseaux de Petri augmentés. Nous englobons également certains langages de description de hardware possédant une sémantique très proche. C'est le cas par exemple de LASSO, qui est un des langages de la famille Cascade (voir annexes). En LASSO, une place est appelé "contrôle".

On peut effectuer certaines vérifications sur un réseau de Petri. La propriété de sûreté assure que deux opérations ne peuvent pas arriver en

---

[1] plus précisément un ensemble de parties évoluant en parallèle, comme un protocole de communication, le fonctionnement d'un cache, et pas uniquement une "machine parallèle" au sens actuel du terme.

même temps, ce qui est précieux pour la synthèse. La propriété de vivacité permet d'identifier les parties inutiles dans un réseau, donc dans un circuit. Mais la vérification de ces propriétés n'est possible que quand la sémantique des transitions est restreinte. Elle est possible avec CAMAD [Pen 86], mais non avec LASSO.

Une première idée pour la synthèse de réseaux de Petri peut être de les traduire en un automate d'états finis, quand la sémantique choisie le permet. Mais cette traduction devient rapidement compliquée si on cherche un résultat optimisé. De plus, les résultats sont rarement bons.

Diverses solutions pour la compilation directe de réseaux de Petri ont été proposées : certaines solutions centralisées proposent une synthèse à partir de PLA, mais la plupart sont basées sur une traduction locale des places et des transitions. En général, une place est traduite par un élément de mémorisation, et une transition par un élément matériel spécifique. Par exemple, on trouve dans [Oli 83] la réalisation suivante pour une transition simple d'une place P1 vers une place P2, conditionnée par le signal T3 :

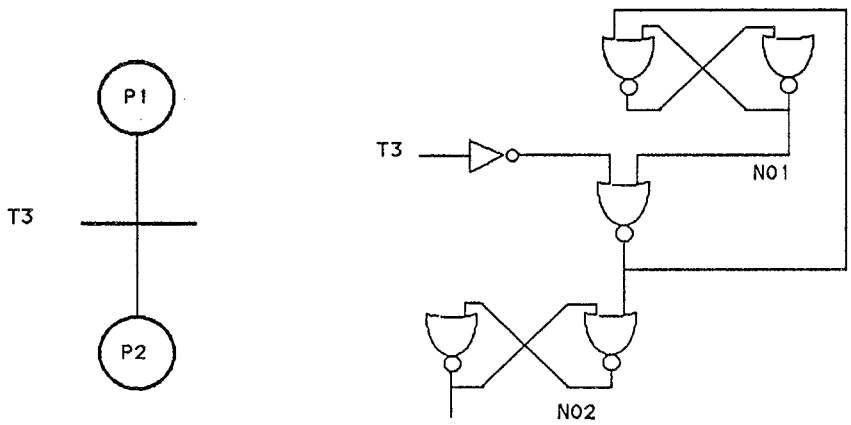


figure 7-2 : traduction d'une transition simple ([Oli 83])

Nous proposons une solution similaire pour la synthèse des parties contrôles décrites en LASSO. La transition **AND** est la plus simple : le ET logique est réalisé par une porte AND. La remise à zéro est assurée en connectant les entrées "RESET" des bascules sources à la sortie du AND.

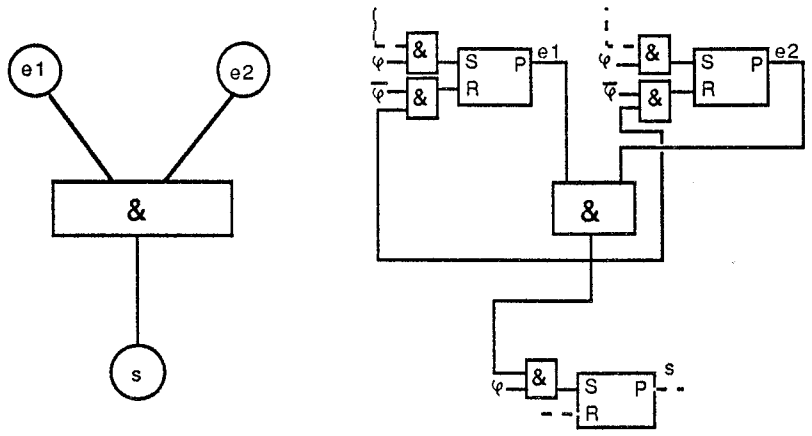


figure 7-3 : traduction de la transition AND de LASSO

La synthèse du type une place/une bascule est intéressante pour LASSO du fait de la complexité de certaines des transitions, qui sont traduites par des circuits combinatoires spécifiques. Elle a des inconvénients : en particulier, la boucle de retour pour la remise à zéro entraîne des difficultés pour la phase de routage, à moins de choisir une implémentation efficace. D'autres méthodes sont utilisées : un codage "NRZ" (Non-Return-to-Zero) qui permet d'éviter la boucle, ou une méthode basée sur la recherche d'invariants dans la matrice d'incidence des transitions[1].

Comme on peut le voir sur la figure, nous évitons les problèmes de stabilité en synchronisant les validations sur une phase de l'horloge et les remises à zéro sur l'autre phase. Ceci a pour principal inconvénient la non-prise en compte de deux transitions successives dans la même phase d'horloge.

Ce problème de synchronisation et de stabilité du circuit, face à des changements rapides de signaux d'entrée par exemple, est un des problèmes importants de la synthèse des réseaux de Petri. Les solutions sont différentes selon que l'on choisit une réalisation purement synchrone, purement asynchrone, ou bien comportant des contrôleurs synchrones communiquant de manière asynchrone.

La réalisation **asynchrone** quoique la plus simple a priori, pose des problèmes de stabilité. [Oli 83] résout le problème en jouant sur les capacitances et sur les résistances des sorties des bascules (cf. figure 7-2).Le problème est traduit en une série d'inégalités linéaires faisant intervenir des facteurs entiers déterminés expérimentalement.

---

[1]M. Currat, communication personnelle.

Pour une raison de simplicité, nous avons choisi une réalisation **synchrone** pour notre synthèse maximale de LASSO [Ova 87].

Les problèmes engendrés par la **communication asynchrone entre contrôleurs synchrones** sont abordés dans [Amb 84]. Le problème est abordé dans [Brü 86] : la synthèse passe par l'introduction dans le graphe de contrôle de transitions spéciales. Ce point est un point important pour la synthèse de circuits pipe-line à partir de LASSO. Nous le traitons en introduisant des places supplémentaires dans le graphe de contrôle.

7.3.2. spécifications pour une synthèse de circuits pipe-line à partir de LASSO

On trouver dans [Ova 87] les éléments pour une synthèse simple de LASSO. Cette méthode donne des circuits importants, puisqu'il n'y a pas de partage des ressources : chaque action associée à un contrôle entraîne la création d'un élément fonctionnel. On cherche donc à réduire cet inconvénient, en réduisant arbitrairement les transitions de manière à avoir à créer moins d'éléments fonctionnels et en essayant de les tenir occupés. En cela les principes de l'architecture pipe-line nous conviennent.

L'architecture pipe-line et sa conception posent des problèmes complexes et sont l'objet d'une littérature spécialisée. Comme **notre ambition ici n'est que d'utiliser le modèle pour améliorer les performances de la synthèse simple du langage LASSO**, nous n'entrerons pas dans le détail.

La synthèse d'un circuit pipe-line consiste à définir le contrôle et les différents modules fonctionnels d'une machine en partant d'une description de haut niveau mettant en évidence les parallélismes possibles entre actions. LASSO convient bien en tant que langage source.

La méthode générale pour la synthèse est la suivante : partitionnement du graphe de contrôle en sous-graphes indépendants, réalisation synchrone de ces sous-graphes suivant la méthode maximale, introduction de mécanismes de synchronisation entre les modules.

Les critères de qualité d'une machine pipe-line définissent les critères pour une bonne synthèse :

1) pour ne pas perdre le bénéfice du pipe-line, veiller à ce que les temps de synchronisation ne soient pas trop importants.

2) trouver une partition du graphe de contrôle qui amène le moins possible de duplication du matériel. En clair, essayer de regrouper dans un même module toutes les opérations qui concernent une même variable.

3) répartir les opérations dans les modules de manière que les temps de passages soient équivalents.

Le travail décrit dans [Ova 87] a permis de donner les principes de la synthèse simple à partir de LASSO et définir les principes pour une synthèse pipe-line. Plusieurs exemples ont été réalisés "à la main", et ont permis de valider les mécanismes de synchronisation entre modules. Ce travail a également mis en évidence les difficultés engendrées par les différents critères.

Chapitre 8

CONCLUSION

8-CONCLUSION

La synthèse logique est le sujet principal du travail que nous présentons ici, même si celui-ci se termine sur des incursions dans les domaines immédiatement voisins. Nous avons montré que le processus de synthèse se divise naturellement en deux phases d'inégale complexité :

Tout d'abord une phase d'allocation (ou implémentation naïve), qui est assez simple, même lorsqu'on prend en compte des langages riches comme LASCAR ou LDS. Nos deux réalisations montrent que si les principes généraux se retrouvent, il faut être très prudent avec les détails car ils dépendent beaucoup du langage source.

La deuxième phase est la phase d'optimisation. C'est d'elle que dépend la qualité du circuit. Contrairement à l'allocation, il n'existe pas de "méthode" répandue. Compte-tenu du côté peu contraint de notre synthèse, nous avons choisi d'utiliser les transformations locales. La première expérience ayant montré la nécessité d'un processus très interactif, nous avons créé un formalisme souple qui permet de les structurer.

Notre synthèse donne des résultats "acceptables" sur des petites descriptions. Elle donnera des résultats fonctionnellement corrects sur des exemples importants, mais peu optimaux : notre réalisation des automates engendrera des circuits combinatoires importants, les aiguilleurs de données seront multipliés inutilement, et surtout le placement-routage automatique créera des circuits peu denses.

Dans notre contexte, l'optimisation devient essentiellement un travail expérimental de mise au point d'un jeu de règles, d'une faible complexité tant que l'on ne cherche pas l'optimalité.

La synthèse de parties opératives soulève des problèmes beaucoup plus complexes. Comme notre expérience le montre, elle fait intervenir un certain nombre de sous-problèmes étroitement corrélés, et dont chacun est intrinsèquement difficile. Alors que la synthèse logique est suffisamment mûre pour être utilisée dans l'industrie, il reste d'importants problèmes dans la synthèse de parties opératives de haut niveau.

Nous avons voulu montrer que les systèmes développés actuellement dans les universités présentent de grandes ressemblances dans les méthodes qu'ils utilisent, sous des apparences très différentes. Même si elle comporte

d'importantes restrictions, la méthode que nous utilisons est à ce titre assez représentative. Néanmoins elle met aussi en évidence *par défaut* l'importance primordiale du modèle architectural dans ce genre de compilation.

La synthèse de partie contrôle est le complément nécessaire de la synthèse logique et de la synthèse de partie opérative : nous en avons donné deux solutions simplistes pour la synthèse logique, et nous l'avons ignoré pour la synthèse de partie opérative. A défaut de réalisation plus complète, nous en avons précisé les principales difficultés.

pour un développement à court terme...

Nous avons montré que la synthèse de parties opératives ne peut être rendue efficace qu'au prix d'importantes restrictions sur les applications traitées. C'est également le cas de la synthèse logique, quoique dans une moindre mesure. Il faut donc choisir un certain nombre de restrictions réalistes, qui rendent les compilations efficaces et permettent la comparaison avec les systèmes voisins : l'effort commencé récemment dans la communauté pour la définition de 'benchmarks" et de bibliothèques standard rend cette démarche possible. Le développement de notre système de synthèse logique passe donc par la compilation de ces exemples, qui permettra de fixer les problèmes dûs à la modélisation et au langage. Ensuite, l'évaluation critique du résultat permettra de valider le compilateur.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[Acos 86] R.D. Acosta, M.N. Huhns, S. Liuh
ANALOGICAL REASONNING FOR DIGITAL SYSTEM SYNTHESIS
Proc. of the Intern. Conf. on Comp. Aided Design,,1986, pp. 173-176.

[Agr 85] Agrawal, V.D.Agrawal, N.N. Biswas
MULTIPLE OUTPUT MINIMIZATION
Proc. of the 22nd Design Automation Conference, 1985.

[Aho 77] A.V. Aho, J.D. Ullman
PRINCIPLES OF COMPILER DESIGN
Addison Wesley 1977.

[Amb 84] P. Amblard
CONCEPTION TEMPORELLEMENT SURE DE CIRCUITS INTÉGRÉS COMPLEXES
Thèse 3ème cycle Informatique, INPG, Grenoble, 15 Juin 1984

[Anc 87a] F. Anceau
SILICON COMPILATION FOR MICROPROCESSOR-LIKE VLSI
in Algorithmics for VLSI, INTERNATIONAL LECTURE SERIES IN COMPUTER SCIENCE, 1987.

[Anc 87b] F. Anceau
FORCE: A FORMAL CHECKER OF EXECUTABILITY
From HDL to guaranteed correct circuit designs, Elsevier Science Publishers, 1987, pp. 295-302.

[And 88] W. Andrews
SILICON COMPILERS STILL STRUGGLING TOWARD WIDESPREAD ACCEPTANCE
Computer Design, February 15, 1988, pp .37-43.

[Ber 85] F. Bertrand
CONCEPTION DESCENDANTE APPLIQUÉE AUX MICROPROCESSEURS VLSI
Thèse de 3ème cycle INPG, Grenoble, Septembre 85

[Bis 84] N.N Biswas
COMPUTER AIDED MINIMIZATION PROCEDURE FOR BOOLEAN FUNCTIONS
Proc. of the 21st Design Automation Conference, 1984, paper 47.1, pp. 699-702.

[Bor 81] D. Borrione
LANGAGES DE DESCRIPTION DE SYSTEMES LOGIQUES
Thèse d'état, INPG , Grenoble, Juillet 1981.

[Bor 88] G. Boriello, E. Detjens
HIGH-LEVEL SYNTHESIS : CURRENT STATUS AND FUTURE DIRECTIONS
Proc. of the 25th Design Automation Conference, paper 32.1, pp. 477-482, 1988.

[Bra 83] D. Brand
REDUNDANCY AND DON'T CARES IN LOGIC SYNTHESIS
IEEE trans. on comp. vol C-32 Oct 83.

[Bra 85] R.K. Brayton, G.D. Hachtel, C.T. McMullen, A.L. Sangiovanni Vincentelli
LOGIC MINIMIZATION FOR VLSI SYNTHESIS
Kluwer Academic Publishers, 1985.

[Bra 86] D. Brand
LOGIC SYNTHESIS
Proc. of the Nato Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design, L'Aquila (Italy) July 7-18, 1986.

[Bro 81] D.W. Brown
A STATE MACHINE SYNTHESIZER
Proc. of the 18th Design Automation Conference, 1981, paper 15.1, pp. 301-305.

[Bro 83] H. Brown, C. Tong, G. Foyster
PALLADIO: AN EXPLORATORY ENVIRONMENT FOR CIRCUIT DESIGN
IEEE Computer, vol 16 n°12, December 1983, pp. 41-56.

[Brü 86] R. Brück, B. Kleinjohan, T. Kathöfer, F.J. Rammig
SYNTHESIS OF CONCURRENT MODULAR CONTROLLERS FROM ALGORITHMIC DESCRIPTIONS
Proc. of the 23rd Design Automation Conference, paper 15.4, pp. 285-292, 1986.

[Bur 86] D. Bursky
FAST SILICON COMPILERS OPTIMIZES MATH BLOCKS
Electronic Design, April 3, 1986.

[Ca& 88] R. Camposano, M. C. McFarland, A.C. Parker
TUTORIAL ON HIGH-LEVEL SYNTHESIS
Proc. of the 25th Design Automation Conference, 1988, paper 23.1, pp. 331-336.

[Chai 82] G.J. Chaitin
REGISTER ALLOCATION & SPILLING VIA GRAPH COLORING
Proc. of the SIGPLAN 82 Symposium on compiler construction, Boston, June 23-25 1982, pp. 98-105.

[Conlan 85] R. Piloty, D. Borrione
THE CONLAN PROJECT: CONCEPTS, IMPLEMENTATIONS, AND APPLICATIONS
Computer, February 85, pp. 81-92.

[Cri 85] Criss
PROLOG-CRISS: UNE EXTENSION DU LANGAGE PROLOG
Document technique, Grenoble, Février 85.

[Cut 87] R.B. Cutler, S. Muroga
DERIVATION OF MINIMAL SUMS FOR COMPLETELY SPECIFIED FUNCTIONS
IEEE trans. on computers vol C-36 n°3, March 87.

[Dag 85] M.R.Dagenais, V.K. Agrawal, N.C. Rumin
THE MC BOOLE LOGIC MINIMIZER
Proc. of the 22nd Design Automation Conference, 1985.

[Dar 80] J.A. Darringer, W.H. Joyner
A NEW LOOK AT LOGIC SYNTHESIS
Proc. of the 17th Design Automation Conference, 1980, pp. 543-548.

[Dar 81] J.A. Darringer, W.H. Joyner
LOGIC SYNTHESIS THROUGH LOCAL TRANSFORMATIONS
IBM journal of Research and Development vol 25 n°4 July 81.

[Dar 84] J.A. Darringer et al.
LSS: A SYSTEM FOR PRODUCTION LOGIC SYNTHESIS
IBM journal of Research and Development vol 28 ,n°5, Sept 84.

[DeMa 86] H. de Man
A SYNTHESIS AND MODULE GENERATION SYSTEM FOR MULTIPROCESSOR SYSTEMS ON A CHIP
Proc. of the Nato Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design, L'Aquila (Italy) July 7-18, 1986.

[Dir 81] S.W. Director, A.C. Parker, D. Sieworek, D.E. Thomas
DESIGN METHODOLOGY AND COMPUTER AIDS FOR DIGITAL VLSI SYSTEMS
IEEE Trans. on circuits and systems vol C-28 n°7 Jul 81.

[Dix 84] J.R. Dixon, M.K. Simmons, P.R. Cohen
AN ARCHITECTURE FOR APPLICATION OF ARTIFICIAL INTELLIGENCE TO DESIGN
Proc. of the 21st Design Automation Conference,1984, paper 40.2, pp. 634-640.

[Dud 83] S. Dudani, E. Stabler
TYPES OF HARDWARE DESCRIPTIONS
Proc. of the Computer Hardware Description Languages and their Applications Conference, 1983, pp. 127-136.

[Dur 86a] Y. Durand
RAPPORT N° 2
Rapport Interne Bull, 8 Août 1986

[Dur 87] Y. Durand
MANUEL D'INTRODUCTION A CASCADE
Rapport technique IMAG n° 30, 1987.

[Eva 85] St. Evanczuk
RESULTS OF A SILICON COMPILER DESIGN CHALLENGE
VLSI Design July 85, pp. 46-54.

[Fri 69] T.D. Friedman, S Yang
METHODS USED IN AN AUTOMATIC LOGIC DESIGN GENERATOR (ALERT)
IEEE Trans. on comp. vol C-18 n°7 Jul 69.

[Gach 87] P. Gachet, B. Joinnault
CONCEPTION D'ALGORITHMES ET D'ARCHITECTURES SYSTOLIQUES
Thèse de l'Université de Rennes I, Rennes, Septembre 87.

[Gir 84] E.F. Girczyc, J.P. Knight
AN ADA TO STANDARD CELL HARDWARE COMPILER BASED ON GRAPH GRAMMARS AND SCHEDULING
Proc. of the I.C.C.D. 84 , Oct 84.

[Goe 86] R. Goering
A NEW GENERATION OF SILICON COMPILERS STALK SYSTEM-LEVEL DESIGN AND SIMULATION
Computer Design, April 15, 1986.

[Goe 87] R. Goering
SILICON COMPILERS BRIDGE GAP BETWEEN CONCEPTS AND SILICON
Computer Design, November 1,1987, pp. 67-80.

[Hach 86] G.D. Hachtel
SOCRATES: A SYSTEM FOR AUTOMATICALLY SYNTHESIZING AND OPTIMIZING COMBINATIONAL LOGIC
Proc. of the 23rd Design Automation Conference, paper 6.1, pp. 79-85, 1986.

[Hit 83] C.Y. Hitchcock III, D. E. Thomas
A METHOD OF AUTOMATIC DATA-PATH SYNTHESIS
Proc. of the 20th Design Automation Conference, paper 31.3, 1983.

[Hon 74] S.J. Hong, R.G. Cain, D.L. Ostapko
MINI: A HEURISTIC APPROACH FOR LOGIC MINIMIZATION
IBM Journal for research and development ,vol 18 ,Sept 74.

[Hor 84] P.W. Horstmann, E.P. Stabler
COMPUTER-AIDED DESIGN USING LOGIC PROGRAMMING
Proc. of the 21st Design Automation Conference, paper 10.1, pp. 144-151, 1984.

[Hua 83] C. Huang, S. Y. H. Su
LOGIC DESIGN AUTOMATION BASED ON LALSD II
Proc. of the Computer Hardware Description Languages and their Applications Conference, 1983, pp. 165-178.

[Jam 85] R. Jamier, N. Bekkara, A. Jerraya
THE AUTOMATIC SYNTHESIS OF DATA PROCESSING SECTIONS
Proc. of the ICCD , 1985.

[Jer 86] A. Jerraya, P. Varinot, R. Jamier, B. Courtois
PRINCIPLES OF THE SYCO COMPILER
Proc. of the 23rd Design Automation Conference, paper 41.1, pp. 715-721, 1986.

[Joe 86] H. Joepen, M. Glesner
OPTIMAL STRUCTURING OF HIERARCHICAL CONTROL PATHS IN A SILICON COMPILER SYSTEM
Proc. of the Intern. Conf. on Comp. Aided Design,1986.

[Joh 79] D. Johansen
BRISTLE BLOCKS: AA SILICON COMPILER
Proc. of the Caltech Conference in VLSI, 1979, pp. 303-310.

[Joy 86] W. Joyner et al.
TECHNOLOGY ADAPTATION IN LOGIC SYNTHESIS
Proc. of the 23rd Design Automation Conference, 1986, paper 6.3, pp. 94-100.

[Kah 84] Mark William Kahrs
SILICON COMPILATION OF VERY HIGH LEVEL LANGUAGES
Ph.D. Thesis, University of Rochester, New York 1984.

[Kah 86] M. Kahrs
MATCHING A PARTS LIBRARY IN A SILICON COMPILER
Intern. Conf. on Comp. Aided Design,1986 .

[Koh 70] Zvi Kohavi
SWITCHING AND FINITE AUTOMATA THEORY
Mc Graw-Hill Book company, 1970.

[Kow 83] T.J. Kowalski, D.E. Thomas
THE VLSI DESIGN AUTOMATION ASSISTANT: PROTOTYPE SYSTEM
Proc. of the 20th Design Automation Conference, paper 31.2, pp. 479-483,1983.

[Laa 85] P.J.M. van Laarhoven, E.H.L. Aarts, M.Davio
PHIPLA: A NEW ALGORITHM FOR LOGIC MINIMIZATION
Proc. of the 22nd Design Automation Conference, 1985.

[Lei 81] G.W. Leive, D.E. Thomas
A TECHNOLOGY RELATIVE LOGIC SYNTHESIS AND MODULE SELECTION SYSTEM
Proc. of the 18th Design Automation Conference, 1981, paper 23.2, pp. 479-485.

[Lid 70] P. Liddell
PARTITION SYNTAXIQUE D'UN SYSTEME LOGIQUE DÉCRIT EN CASSANDRE
Thèse de 3ème cycle, Université de Grenoble, Mars 1970

[Lipp 83] H.M. Lipp
METHODIC ASPECTS OF LOGIC SYNTHESIS
Proc. of the IEEE vol 71 n°1 Jan 83.

[Mari 86] S. Marine
IRENE: UN LANGAGE POUR LA DESCRIPTION, SIMULATION ET SYNTHESE AUTOMATIQUE DU MATERIEL VLSI
Thèse INPG, Grenoble, 3 Fevrier 86.

[Mar 87] A.J. Martin
SELF-TIMED FIFO: AN EXERCISE IN COMPILING PROGRAMS INTO VLSI CIRCUITS
From HDL to guaranteed correct circuit designs, Elsevier Science Publishers, 1987, pp. 133-153.

[Marw 84] P. Marwedel
THE MIMOLA DESIGN SYSTEM: TOOLS FOR THE DESIGN OF DIGITAL PROCESSORS
Proc. of the 21st Design Automation Conference, 1984, paper 35.2, pp 587-593.

[Marw 86] P. Marwedel
A NEW SYNTHESIS ALGORITHM FOR THE MIMOLA SOFTWARE SYSTEM
Proc. of the 23rd Design Automation Conference, 1986, paper 15.2, pp. 271-277.

[Math 88] H.-J. Mathony, U.G. Baitinger
CARLOS: AN AUTOMATED MULTILEVEL LOGIC DESIGN SYSTEM FOR CMOS SEMI-CUSTOM INTEGRATED CIRCUITS
IEEE Trans. on C.A.D. vol 7 n°3 March 88.

[McDer 78] D. McDermott
CIRCUIT DESIGN AS PROBLEM SOLVING
Artificial Intelligence and Pattern Recognition in Computer Aided Design, Latombe ed., North-Holland Publishing Company, 1978.

[Mer 73] J. Mermet
ETUDE MÉTHODOLOGIQUE DE LA CONCEPTION ASSISTÉE PAR ORDINATEUR DES SYSTEMES LOGIQUES: CASSANDRE
Thèse d'état, U.S.M.G., Grenoble, Avril 1973

[Mer 85] J. Mermet
SEVERAL STEPS TOWARDS A CIRCUIT INTEGRATED CAD SYSTEM: CASCADE
Proc. of the 7th Conf. on Computer Hardware Description Languages, Tokyo Aug. 1985.

[Mich 87] G. de Micheli
PERFORMANCE-ORIENTED SYNTHESIS OF LARGE-SCALE DOMINO CMOS CIRCUITS
IEEE Trans. on C.A.D. vol 6 n°5, Sept 87.

[Mil 83] G. J. Milne
THE CORRECTNESS OF A SIMPLE SILICON COMPILER
Proc. of the 6th Conf. on Computer Hardware Description Languages, Tokyo Aug. 1985, pp. 1-12.

[Nes 86] J.A. Nestor, D.E. Thomas
BEHAVIORAL SYNTHESIS WITH INTERFACES
Proc. of the Intern. Conf. on Comp. Aided Design,1986, pp. 112-115.

[NPar 85] N. Park, A.C. Parker
SYNTHESIS OF OPTIMAL CLOCKING SCHEMES
Proc. of the 22nd Design Automation Conference, 1985, paper 29.3, pp. 489-495.

[NPar 86] N. Park, A.C. Parker
SEHWA: A PROGRAM FOR SYNTHESIS OF PIPE-LINES
Proc. of the 23rd Design Automation Conference, 1986, paper 27.1, pp. 454-460.

[NPar 88] N. Park, A. C. Parker
SEHWA: A SOFTWARE PACKAGE FOR SYNTHESIS OF PIPELINES FROM BEHAVIORAL SPECIFICATIONS
IEEE Trans. on Computer-Aided-Design, vol. 7, n° 3, Mars 1988, pp. 356-370.

[Obr 82] M. Obrebska
ETUDE COMPARATIVE DE DIFFÉRENTES MÉTHODES DE CONCEPTION DES PARTIES CONTROLE DES MICROPROCESSEURS
Thèse de Docteur-Ingénieur, INPG , Grenoble,1982.

[Oli 83] V. Olive, D. Rouquier
A SYSTEMATIC METHOD FOR THE SYNTHESIS OF CONTROL PARTS DEFINED BY GRAFCET
Information processing `83, Elsevier Science Publishers, 1983.

[Ova 87] D. A. Ovalle
SYNTHESE DE SYSTEMES PIPE-LINE A PARTIR D'UNE DESCRIPTION FORMELLE DE LEUR COMPORTEMENT
Mémoire de DEA, INPG, Grenoble, Septembre 1987.

[Pan 86] B.M. Pangrle, D.D. Gajski
STATE SYNTHESIS AND CONNECTIVITY BINDING FOR MICROARCHITECTURE COMPILATION
Proc. of the Intern. Conf. on Comp. Aided Design,, 1986 .

[Par 79] A. Parker et al.
THE CMU DESIGN AUTOMATION SYSTEM: AN EXAMPLE OF AUTOMATED DATA PATH DESIGN
Proc. of the 16th Design Automation Conference, 1979, pp. 73-80.

[Par 82] A.W. Nagle, R. Cloutier, A.C. Parker
SYNTHESIS OF HARDWARE FOR THE CONTROL OF DIGITAL SYSTEMS
IEEE Trans. on CAD of integrated circuits and systems vol CAD-I n°4 Oct 82.

[Par 84] A.C. Parker, F. Kurdahl, M. Mlinar
A GENERAL METHODOLOGY FOR SYNTHESIS AND VERIFICATION OF REGISTER-TRANSFER DESIGNS
Proc. of the 21st Design Automation Conference, 1984, paper 20.4, pp. 329-335.

[Par 86] A.C. Parker, J.T. Pizzaro, M.Mlinar
MAHA: A PROGRAM FOR DATA-PATH SYNTHESIS
Proc. of the 23rd Design Automation Conference, 1986, paper 27.2, pp. 461-466.

[Par 88] N. Park, A.C. Parker
SEHWA: A SOFTWARE PACKAGE FOR SYNTHESIS OF PIPE-LINES FROM BEHAVIORAL SPECIFICATIONS
IEEE Trans. on C.A.D. vol 7 n°3 March 88.

[Pau 86] P.G. Paulin, J.P. Knight, E.F. Girczyc
HAL: A MULTI-PARADIGM APPROACH TO AUTOMATIC DATA-PATH SYNTHESIS
Proc. of the 23rd Design Automation Conference, 1986, paper 15.2, pp. 263-270.

[Pen 86] Z. Peng
SYNTHESIS OF VLSI SYSTEMS WITH THE CAMAD DESIGN AID
Proc. of the 23rd Design Automation Conference,. paper 15.3, pp. 278-284.

[Per 78] M. Perkowski
THE STATE-SPACE APPROACH TO THE DESIGN OF A MULTIPURPOSE PROBLEM-SOLVER FOR LOGIC DESIGN
Artificial Intelligence and Pattern Recognition in Computer Aided Design, Latombe ed., North-Holland Publishing Company, 1978.

[Pol 73] K. de Polignac
UTILISATION DU LANGAGE CASSANDRE POUR LA CONCEPTION DE MACHINES MICROPROGRAMMÉES
Thèse 3ème cycle, U.S.M.G., Grenoble, Juin 1973

[Raj 86] V.K. Raj
ANOTHER AUTOMATED DATA PATH DESIGNER
Proc. of the Intern. Conf. on Comp. Aided Design,,1986 .

[Raz 88] R. R. Razouk
THE USE OF PETRI NETS FOR MODELING PIPELINED PROCESSORS
Proc. of the 25th Design Automation Conference, paper 36.4, pp. 548-553, 1988.

[Ros 85] W. Rosenstiel, R. Camposano
SYNTHESIZING CIRCUITS FROM BEHAVIORAL LEVEL SPECIFICATIONS
Proc. of the C.H.D.L. .1985.

[Ros 87] W. Rosenstiel, R. Camposano
THE KARLSRUHE DSL SYNTHESIS SYSTEM
From HDL to guaranteed correct circuit designs, Elsevier Science Publishers, 1987, pp. 155-170.

[Row 86] J. Rowson, B. Walker
INSIDE A 2901 DATAPATH COMPILER
VLSI Systems Design, May 1986.

[Row 87] J. Rowson, S. Trimberger
SECOND-GENERATION COMPILERS OPTIMIZE SEMI-CUSTOM CIRCUITS
Electronic Design, February 19, 1987, pp. 93-96.

[Sai 86] T. Saito, H. Sugimoto, M. Yamazaki, N. Kawato
A RULE-BASED LOGIC CIRCUIT SYNTHESIS SYSTEM FOR CMOS GATE ARRAYS
Proc. of the 23rd Design Automation Conference,, paper 34.1, pp. 594-600, 1986.

[Saka 84] M. Sakarovitch
TECHNIQUES MATHÉMATIQUES DE LA RECHERCHE OPÉRATIONNELLE . III - OPTIMISATION DANS LES RÉSEAUX
Hermann ,1984.

[Sis 82] J.H. Siskind, J.R. Southard, K.W. Crouch
GENERATING CUSTOM HIGH PERFORMANCE VLSI DESIGNS FROM SUCCINCT ALGORITHMIC DESCRIPTIONS
Conference on advanced research in VLSI, MIT, 1982, pp. 28-39.

[Stei 84] L.I. Steinberg, T.M. Mitchell
A KNOWLEDGE-BASED APPROACH TO VLSI CAD : THE REDESIGN SYSTEM
Proc. of the 21st Design Automation Conference, paper 26.2, pp. 412-418.

[Stei 85] L.I. Steinberg, T.M. Mitchell
THE REDESIGN SYSTEM: A KNOWLEDGE-BASED APPROACH TO VLSI CAD
IEEE Design and Test, Feb 1985, pp45-54.

[Sus 78] G.J. Sussman
SLICES: AT THE BOUNDARY BETWEEN ANALYSIS AND SYNTHESIS
Artificial Intelligence and Pattern Recognition in Computer Aided Design, Latombe ed., North-Holland Publishing Company, 1978.

[Tre 86] L. Trevillyan, W. Joyner, L. Berman
GLOBAL FLOW ANALYSIS IN AUTOMATIC LOGIC DESIGN
IEEE trans. on comp. vol C-35 ,n°1 ,Jan 86.

[Tor 77] H.C. Torng, N. C. Wilhelm
THE OPTIMAL INTERCONNECTION OF CIRCUIT MODULES IN MICROPROCESSOR AND DIGITAL SYSTEM DESIGN
IEEE Trans. on computers vol C-26 n°5 pp. 450-457 May 77.

[Tse 81] Chia-Jeng Tseng, D.P. Sieworek
THE MODELING AND SYNTHESIS OF BUS SYSTEMS
Proc. of the 18th Design Automation Conference, 1981, paper 23.1, pp. 471-478.

[Tse 83] Chia-jeng Tseng, D.P. Sieworek
FACET: A PROCEDURE FOR THE AUTOMATED SYNTHESIS OF DIGITAL SYSTEMS
Proc. of the 20th Design Automation Conference, 1983, paper 31.4, pp. 490-496.

[Tse 84] Chia-Jeng Tseng, D.P. Sieworek
EMERALD: A BUS STYLE DESIGNER
Proc. of the 21st Design Automation Conference, 1986, paper 20.2, pp. 315-321.

[Tse 86] C-J Tseng, A.M. Prabhu, C.Li, Z. Mehmood, M.M. Tong
A VERSATILE FINITE STATE MACHINE SYNTHESIZER
Proc. of the Intern. Conf. on Comp. Aided Design,1986.

[Vai 82] A.K. Vaidya
AUTOMATIC LOGIC SYNTHESIS FOR VLSI GATE-ARRAY IMPLEMENTATIONS OF A DIGITAL SYSTEM FROM A CONLAN DESCRIPTION
PH.D. Thesis at the University of Wisconsin-Madison, Madison, 1982.

[VLSI 84] VLSI DESIGN staff
SILICON COMPILERS
VLSI DESIGN Sept 84 vol V n°9.

[Wei 88] R. Wei, S. Rothweiler, J. Jou
BECOME: BEHAVIOR LEVEL CIRCUIT SYNTHESIS BASED ON STRUCTURE MAPPING
Proc. of the 25th Design Automation Conference, paper 27.2, pp. 409-414.

[Yos 86] T. Yoshimura, S. Goto
A RULE-BASED AND ALGORITHMIC APPROACH FOR LOGIC SYNTHESIS
Proc. of the Intern. Conf. on Comp. Aided Design,1986 pp. 162-165.

[You 85] J. L. Young
WHY SILICON COMPILERS HAD TO CHANGE ITS STRATEGY
Electronics, October 14, 1985.

ANNEXES

ANNEXES

1. généralités sur les langages de CASCADE

Le langage Cascade permet de modéliser les circuits depuis le niveau électrique jusqu'au niveau architecture. Néanmoins, tous les sous-langages de Cascade ont des caractéristiques communes:

* même forme: interface, corps, déclaration

* même expression de la structure

* notions fondamentales: porteuses, types de valeurs...

La grammaire est en grande partie identique pour les différents niveaux de langage : même forme générale, mêmes opérateurs, même expression des conditions, des expressions régulières, des constructions répétitives...

la notion de porteuse

La notion de porteuse est une généralisation de la notion de variable: une porteuse est définie comme un type générique auquel on associe un couple de paramètres: le type de la valeur portée et la valeur par défaut.

Le type générique caractérise le comportement de la variable: est-elle capable de conserver une valeur ou non, de quelle manière l'affecte-t-on ... le type de valeur caractérise l'ensemble des valeurs que peut prendre la porteuse, et les opérations qui peuvent lui être appliquées. Par exemple on pourra additionner des porteuses ayant des types de valeurs entières. La valeur par défaut est la valeur que prend la porteuse lorsqu'aucune valeur ne lui a encore été imposée.

expression de la structure

Chaque module peut être décomposé en plusieurs sous-modules interconnectés.

Tout module apparaissant dans le modèle est une instance d'une description, soit prédéfinie soit issue d'un catalogue de descriptions indépendantes. A chaque niveau de l'arbre des imbrications de modules, tout sous-arbre peut être considéré comme un modèle complet si nécessaire.

Une description se décrit par une partie "description". Une description est le modèle d'une boite qui communique avec l'extérieur par une interface composée de porteuses correspondant aux broches d'entrée/sortie d'un circuit. Dans le corps de la description, on trouve les déclarations d'objets typés, une partie fonctionnelle (relations) et/ou une partie structurelle. Cette dernière partie décrit la décomposition structurelle d'un modèle en modules: les unités. Une unité est une instance d'une description précédemment définie - laquelle peut elle-même être décrite sous la forme d'un assemblage d'unités plus élémentaires.

2. le langage LASCAR

niveau de modélisation

LASCAR décrit des systèmes au niveau transfert de registres. A ce niveau, on décrit textuellement les fonctions effectuées par le matériel. LASCAR permet d'exprimer des fonctions logiques ou entières, des synchronisations et des séquencements d'actions élémentaires.

Comme la plupart des langages de programmation, LASCAR est un langage d'affectation de variables: cela signifie que comme FORTRAN, PASCAL, C, etc ... sa construction de base est l'affectation d'une variable avec le résultat d'une opération. Mais attention ! à la différence de ces derniers, LASCAR est non procédural: l'ordre d'écriture des affectations n'influe pas sur le résultat. On ne peut affecter plusieurs fois la même variable, à moins que ce soit sous des conditions disjointes.

Une description LASCAR combine deux notions: les notions correspondant à la partie opérative du circuit où sont effectuées les opérations et la partie contrôle où est calculé le séquencement des actions de la partie opérative. La partie opérative est décrite par des combinaisons et des connexions de "signaux" et de "latches", par des transferts de registres ou de mémoires. La partie contrôle peut être modélisée directement dans un formalisme d'automates d'états finis.

Les types prédéfinis

Il y a 3 types de valeurs en LASCAR:

type BOOL = {`0,`1}

type IMPULSION = {`0,`1}

type ENT (entiers relatifs) = Z

Il y a 4 types de porteuses.

type SIGNAL: ce type représente les fils de connexion. Après exécution d'une opération de connexion, le nouveau contenu du signal est immédiatement disponible , mais n'est pas mémorisé.

type LATCH: Le contenu d'un latch est mémorisé, et immédiatement disponible.

type REGISTRE: Une porteuse de type registre est chargée sur le front montant d'une porteuse de type impulsion. La valeur chargée est mémorisée, mais n'est disponible que une unité de temps plus tard.

type COMPTEUR :Ce type est défini à partir du type REGISTRE mais ses opérateurs associés sont différents. Il n'admet que des valeurs entières. La valeur chargée à l'instant T est disponible à l'instant T+1.

Certains types courants de porteuses sont prédéfinis, ce qui évite leur redéfinition:

COMPT0 = COMPTEUR (INT,0), REGENT0 = REGISTER (INT,0),

SIGENT0 = SIGNAL (INT,0), LATENT0 = LATCH (INT,0),

REGB0 = REGISTER (BOOL, `0), SIGB0 = SIGNAL (BOOL,`0),

CLOCK = SIGNAL (IMPULSION,`0), LATB0 = LATCH (BOOL,`0)

connexions et chargement de registres.

Les instructions de base en LASCAR sont les instructions d'affectation. Il y a deux sortes d'affectation: les connexions et les chargements de registres.

Modélisation du contrôle.

LASCAR permet de modéliser la partie contrôle d'un circuit en utilisant le formalisme d'automates. Un automate c'est:

* un ensemble d'états représentés par des étiquettes.

* des instructions de séquencement (instructions load), conditionnées ou non, qui permettent de "passer" d'un état à un autre.

Dans un automate, un seul état est actif à un instant donné: ceci signifie que seules les instructions sous portée de l'état actif seront activables. Dans une description LASCAR, il y a au plus un automate.

L'environnement de LASCAR

Comme tous les outils du système CASCADE, le langage LASCAR bénéficie d'outils de simulation et de vérification. Il existe également un outil de structuration des descriptions écrites en LASCAR.

3. le langage LASSO

niveau de modélisation

Le langage LASSO sert à modéliser les interactions entre composants d'un système. Les instructions et les notions primitives du langage sont assez abstraites: elles sont basées sur un modèle de graphe de contrôle pour la synchronisation des actions concurrentes. De ce fait, le langage sépare très nettement:

* la partie opérative, exprimée par des actions,

* la partie contrôle, modélisée par des signaux de contrôle et des transitions.

Les types prédéfinis.

On retrouve les trois types de LASCAR BOOL, IMPULSION et ENT. On ajoute le type chaine de caractère STRING.

En LASSO, on rencontre plus de types de porteuses qu'en LASCAR:
De LASCAR on retrouve le type SIGNAL. Les autres types sont:

type VARIABLE: ce type représente la variable algorithmique habituelle. Il n'a pas de signification au niveau du matériel.

type D-VARIABLE: ce type représente un élément de mémorisation présentant un délai entre sa modification et la disponibilité de sa nouvelle valeur.

type CONTROL: ce type est utilisé pour modéliser les mises à feu des transitions du modèle. La valeur d'une porteuse de type CONTROL ne peut être modifiée que par les transitions primitives du graphe de contrôle, et ne peut prendre que deux valeurs: 1 ("valide") ou 0 ("invalide").

Types prédéfinis

On retrouve les SIGB0, SIGENT0, HORLOGE de LASCAR. De plus:

SIGCAR = SIGNAL (STRING, ' ') ,

CTRLE0 = CONTROL(INT,0),

CTRLE1 = CONTROL (INT,1),

VARB0 = VARIABLE (BOOL,0),

IVAR0 = VARIABLE (INTEGER,0),

etc ...

Modélisation de la partie opérative.

Les actions sont des groupes d'"instructions opératives". Ces instructions modifient les valeurs des porteuses du modèle. Assez classiquement, ce sont des affectations de variables , éventuellement conditionnées (formes "if" et "case"). Ces opérations sont éxécutées lors de la mise à feu d'une transition.

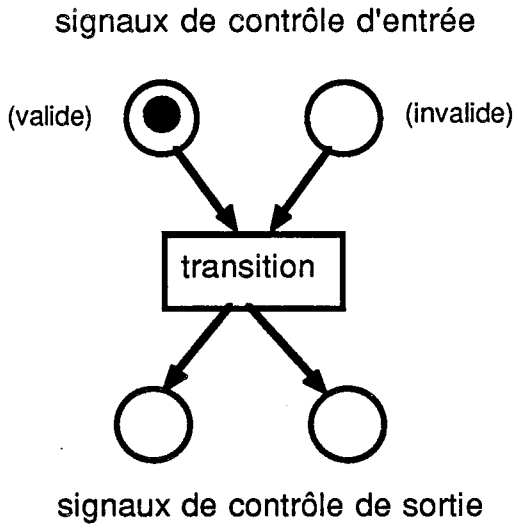
Dans tous les cas, la modification simultanée d'une même variable est considérée comme une erreur. Si elle se produit pendant la simulation, elle sera signalée comme telle.

Comme LASCAR, LASSO est un langage impératif, c'est à dire un langage d'affectation. On dispose ici de deux types d'affectations: la connexion (comme en LASCAR) et l'affectation de variables (comme en PASCAL). L'affectation de variables est plus délicate en LASSO qu'en PASCAL: comme LASSO est un langage parallèle, si on n'est pas prudent on arrive très vite à des modèles non déterministes.

Il y a deux types d'affectations en LASSO: la connexion (.=) qui concerne les signaux (type SIGNAL), et l'affectation (:=) qui concerne les variables (types VARIABLE et D_VARIABLE).

modélisation du contrôle.

Le graphe de contrôle est constitué de l'interconnexion de deux types de nœuds: les signaux de contrôle et les transitions. Les signaux de contrôle représentent des évènements et les transitions modifient les signaux de contrôle auxquelles elles sont connectées ("mise à feu").



Les signaux de contrôle: un signal de contrôle peut être valide ou invalide. Il est valide si l'évènement qu'il représente a eu lieu, et que ses conséquences n'ont pas encore été prises en compte. Il est invalide dans les autres cas.

Les signaux de contrôle sont nommés et déclarés dans la partie déclaration.

Les transitions: Les transitions font évoluer l'état du graphe de contrôle: elles changent l'état de ses nœuds. Une transition a un ou plusieurs signaux de contrôle en entrée, et un ou plusieurs signaux de contrôle en sortie. La transition est mise à feu dans le cas d'une certaine configuration de ses entrées, dépendante de sa sémantique. Lorsqu'une transition est mise à feu, certains de ses signaux d'entrée sont modifiés (invalidés) , ainsi que certains de ses signaux de sortie (qui eux sont validés).

Il y a 6 types de transitions: AND, INDEX, SELECT, SELECTP, SELECTR et TEST. On peut associer un délai (exprimé par un nombre entier d'unités de temps) à chaque transition. Par défaut, le délai vaut 0.

L'environnement de LASSO

Comme tous les outils du système CASCADE, le langage LASSO bénéficie d'outils de simulation et de vérification. Il existe également un

éditeur graphique de LASSO, qui permet de visualiser graphiquement l'évolution d'un modèle au cours de sa simulation.

4. le langage LDS

Le langage LDS, comme Cascade, permet à la fois la description structurelle et la description comportementale.

Une description structurelle en LDS ressemble beaucoup à une description structurelle en Cascade: on retrouve la même expression de la hiérarchie, au moyen d'instanciations de modules, d'interfaces, etc...

Nous ne nous intéresserons dans la suite que à la description comportementale au moyen de LDS *en nous restreignant à ce que nous avons utilisé.*

niveau de modélisation

LDS décrit des systèmes au niveau algorithme. LDS est procédural, et il décrit les fonctions du matériel par des instructions. Les opérations considérées par LDS sont les fonctions logiques ou entières.

types des variables

Les déclarations de variables ne se trouvent pas avec les instructions dans les CMODULEs, mais dans des modules indépendants. La déclaration des variables d'interface se fait dans le module englobant COMPONENT, les autres variables se trouvent dans un GMODULE si leur déclaration est globale pour le modèle, ou dans un RMODULE sinon.

Ils y a plusieurs "types de porteuses" en LDS: clock, latch, signal, ...

Dans la déclaration d'une porteuse de type "latch" on spécifie l'horloge de type "clock" qui en synchronise le chargement.

Ces porteuses n'ont qu'un type de valeur, le type "bin" mais peuvent avoir des longueurs de champ différents. Des attributs comme "alim", "domain" permettent de préciser la nature des informations qui transitent par son intermédiaire. Cette information permet des contrôles sur le modèle.

modélisation du séquencement à l'intérieur d'un CMODULE

les fonctions d'un modèle sont décrite par des instructions évaluées procéduralement comme en Pascal.

L'utilisateur dispose de constructions du type while, for. Il peut de plus utiliser des étiquettes et l'instruction de branchement GOTO.

synchronisation entre CMODULES

Il existe deux instructions permettant l'activation d'un CMODULE depuis un autre CMODULE:

L'instruction "CAUSE" indique un branchement parallèle à un bloc comportemental. L'évaluation de ce bloc débute au temps courant, concurrement à l'éxécution du bloc appellant.

L'instruction "EXECUTE" indique un branchement séquentiel à un bloc comportemental: le bloc appelant est mis en attente jusqu'à la fin du traitement du bloc appelé.

l'environnement de LDS

L'environnement du langage LDS est riche: des outils remontants du type extracteurs (FAON, PAON,GEMO), un simulateur (SCALP) et un environnement de modélisation (LEDS).

5. preuve de l'algorithme de séquencement

Le formalisme potentiel tâche ne s'applique qu'à des graphes non bouclés. Il faut donc prouver que les contraintes d'antériorités calculées par l'algorithme du chapitre 7 ne contiennent pas de boucles[1].

Nous reproduisons ci-dessous l'algorithme. Pour rester cohérent avec les notations de [BerBiz 83], nous traduisons la boucle *pourtout* en une boucle *tantque*. Pour simplifier l'écriture, nous notons PaDr[I] l'ensemble des variables apparaissant en partie droite de l'instruction I, et paga[I] la partie gauche de cette instruction.

```
début
  I= nombre d'instructions;
  tantque ( I>1) faire
    tantque  (PaDr[I]≠ ∅) & ( V εPaDr[I]) faire
      J=I-1;
      tantque ( J≠0)&(V ≠paga[J])  faire
        J=J-1;
      finfaire
      précédent(J,I)       /* J doit précéder I */
    finfaire
    J=0;
    I=I-1;
  finfaire
fin
```

Nous prouvons d'abord la propriété suivante *précédent(J,I) => I>J* , où I et J représentent les I*e* et J*e* instructions de la description source. La propriété apparait presque évidente à la lecture de l'algorithme. Pour les sceptiques, nous utiliserons une méthode de preuve de correction partielle due à Hoare telle qu'elle est exposée dans [BerBiz 83].

Si nous appelons P notre algorithme, nous allons d'abord montrer:

$$(I>J)\{P\}(I>J)$$

---

[1]Ce travail est issu de la recherche d'une erreur.

Nous commençons par montrer que I>J est invariant de la boucle P3 définie comme suit:

$$P3 \Leftrightarrow \{ \text{tantque } (J\neq 0)\&(V\neq paga[I]) \text{ faire } J=J-1;\}$$

en utilisant l'axiome de l'affectation, on écrit:

$$(I>J-1)\ \{ J = J-1 \}\ (I>J)$$

comme:

$$(I>J) \Rightarrow (I>J-1)$$

en utilisant la règle de la précondition, on déduit:

$$\textbf{a:}\ (I>J)\ \{ J = J-1; \}\ (I>J)$$

comme d'autre part

$$\textbf{b:}\ (I>J)\&(J\neq 0)\&(V\neq paga[I]) \Rightarrow (I>J)$$

en utilisant la règle de la composition avec a: et b: , il vient:

$$(J\neq 0)\&(V\neq paga[I])\ \{J = J-1 \}\ (I>J)$$

ce qui, d'après la règle de la boucle *tantque*, est équivalent à:

$$(I>J)\ \{ \text{tantque } (J\neq 0)\&(V\neq paga[I]) \text{ faire } J=J-1; \text{fin}\}\ (I>J)$$

ce que nous écrivons

$$\textbf{c:}\ (I>J)\ \{ P3 \}\ (I>J)$$

montrons ensuite que (I>J) est un invariant de la boucle P2, qui englobe P3

$$P2 \Leftrightarrow \{\text{tantque } (PaDr[I]\neq \varnothing)\&(V \varepsilon PaDr[I]) \text{ faire } J=I-1;\ P3;\ prec(J,I);\ \text{fin}\}$$

en utilisant l'axiome de l'affectation, il vient:

$$(I>I-1)\ \{ J = I-1 \}\ (I>J)$$

comme

$$(I>J)\ \&(PaDr[I]\neq \varnothing)\ \&\ (V \varepsilon PaDr[I]) \Rightarrow (I>I-1)$$

en utilisant la règle de la précondition, on déduit:

d: *(I>J) & (PaDr[I]≠ ∅) & (V∈PaDr[I]) { J = I-1; } (I>J)*

en utilisant la règle de la composition avec c: et d: , il vient:

e: *(I>J) &(PaDr[I]≠ ∅) & (V∈PaDr[I]) { J =I-1; P3; } (I>J)*

d'autre part

f: *(I>J) { prec(J,I); } (I>J)*

en utilisant la règle de la composition avec e: et f: , il vient:

(I>J) &(PaDr[I]≠ ∅) & (V∈PaDr[I]) { J = I-1; P3; prec(J,I); } (I>J)

ce qui, d'après la règle de la boucle *tantque*, est équivalent à:

(I>J) {tantque(PaDr[I]≠∅) & (V∈PaDr[I]) faire J=I-1; P3; prec(I,J);fin}(I>J)

ce que nous écrivons

g: *(I>J) { P2 } (I>J)*

montrons enfin que (I>J) est un invariant de la boucle P1, qui englobe P2

P1 ⇔ {tantque (I≠1) faire P2; J=0; I=I-1; fin}

comme P2 ne contient aucune affectation de I, on a

h: *(I>1) { P2 } (I>1)*

de g: et h: on déduit

i: *(I>J) & (I>1) { P2 } (I>J) & (I>1)*

d'autre part, en utilisant l'axiome de l'affectation, il vient

(I-1>J) { I=I-1 ; } (I>J)

en utilisant la composition

(I-1>0) { J=0; I=I-1; } (I>J)

comme

$(I>J)\ \&\ (I>1) \Rightarrow (I-1>0)$

on obtient

$j: (I>J)\ \&\ (I>1)\ \{ J=0; I=I-1; \}\ (I>J)$

en combinant i: et j:, on obtient

$(I>J)\ \&\ (I>1)\ \{ P2 ; J=0; I=I-1; \}\ (I>J)$

ce qui est équivalent à

$(I>J)\ \{tantque\ (I\neq 1)faire\ P2; J=0; I=I-1; fin\}\ (I>J)$

comme (I>J) est un invariant de notre algorithme, et que

$(I>J) \Rightarrow \neg(prec(J,I))\ \vee\ (I>J)$

nous en déduisons la propriété recherchée

$précédent(J,I) => I>J$

A partir de cette propriété, nous montrons que les contraintes d'antériorité obtenues par cet algorithme ne créent pas de boucle.

SZupposons que le graphe contienne une boucle. Ceci se traduit par:

$\exists(H, K_1,K_2...K_n,L)\ \varepsilon \mathbb{N}^{n+2}$ *tel que*

$prec(L,H)\ \&\ prec(H,K_1)\ \&\ prec(K_1,K_2)\ \&...\&\ prec(K_n,L)$

ceci implique

$(L > H)\ \&\ (H > K_1)\ \&\ (K_1>K_2)\ \&...\&\ (K_{n-1}>K_n)\ \&\ (K_n>L)$

soit (L>H) & (H>L) ce qui est impossible.

référence: les propriétés utilisées sont détaillées aux pages 5,6,7 de [BerBiz 83]

[BerBiz 83] P.Berlioux, Ph. Bizard
ALGORITHMIQUE: CONSTRUCTION, PREUVE ET ÉVALUATION DE PROGRAMMES
DUNOD informatique, 1983

6. séquence de transferts de l'instruction movs

les transferts

```
date 0:
       J -> inB1      b0
       J -> inB2      b1
       OLDCO -> inA1  b2
       1 -> inA2      b3
date 1:
       outB -> R1     b4
       outA -> R3     b5
date 2:
       R1 -> inB1     b6
       R_INDEX -> inB2     b7
       AD -> inA1     b8
       R3 -> inA2     b9
date 3:
       outB -> R2     b10
       outA -> R4     b11
date 4:
       R2 -> IOAD     b12
       R4 -> inA1     b13
       1 -> inA2      b14
date 5:
       outA -> R1     b15
date 6:
       IODAT -> D1    b16
date 7:
       R4 -> inA1     b17
       D1 -> inA2     b18
       R1 -> IOAD     b19
date 8:
       outA -> R2     b20
date 9:
       IODAT -> GR(2)      b21
       R2 -> inA1     b22
       1 -> inA2      b23
date 10:
       outA -> R3     b24
       R4 -> IOAD     b25
date 11:
date 12:
       IODAT -> GR(1)      b26
```

```
date 13:
       R3 -> IOAD     b27
       GR(2) -> IODAT      b28
date 14:
date 15:
       R2 -> IOAD     b29
       GR(1) -> IODAT      b30
```

regroupements obtenus

Ces transferts se traduisent par un graphe de compatibilités comportant 31 nœuds et 409 arcs. L'algorithme du chapitre 6 les regroupe en 6 cliques:

clique A: { b26,b30,b28,b16,b24,b21,b20,b15,b0,b10 }

clique B: { b12,b29,b27,b25,b19,b5,b11 }

clique C: { b14,b23,b18,b9,b4 }

clique D: { b13,b17,b22,b8,b3 }

clique E: { b1,b7 }

clique F: { b2,b6 }

On en déduit la connectivité des six bus:

bus A: { IODAT,GR(1),GR(2),D1,outA,R3,R2,R1,inB1,J,outB }

bus B: { IOAD,R2,R3,R4,R1,outA }

bus C: { inA2,1,D1,R3 }

bus D: {inA1, R4,R2,AD,inA2 }

bus E: {inB2,J,R_INDEX }

bus F: { inB1, inA1, OLDCO, R1 }

7. Index

RÉSUMÉ

Le problème de la synthèse automatique de circuits est abordé ici à travers trois expériences de réalisation de compilateurs. La première concerne la traduction de la spécification fonctionnelle d'un circuit décrit en LASCAR (langage de la famille CASCADE). Le deuxième compilateur utilise un formalisme de règles de réécriture pour produire un circuit adapté à une bibliothèque spécifique de BULL-SYSTEMES à partir d'une description en langage LDS. La troisième expérience aborde le problème de la synthèse de parties opératives, dont les principales difficultés sont présentées en détail. La méthode utilisée met en œuvre un algorithme de séquencement fondé sur un formalisme potentiel-tâche, et une méthode de partage de registres et d'allocation d'opérateurs à partir d'un algorithme de coloriage de graphe.

MOTS-CLEFS

B.7

CAO de VLSI, synthèse automatique de matériel, compilation de silicium.