

Année 1991

Numéro d'ordre: 713

THESE

présentée à

l'Université des Sciences et Techniques de LILLE FLANDRES ARTOIS

Pour Obtenir le titre de

Docteur en Informatique



L'abstraction fonctionnelle des parties contrôles des circuits pour l'accélération de simulateurs générés

Une contribution au développement d'outils de C.A.O. de l'architecture matérielle.

Par Jean-Luc DUBOIS

soutenue le 18 Avril 1991 devant la commission d'Examen

Membres du Jury:

Vincent CORDONNIER
Jean-Paul CALVEZ
Bernard TOURSEL
Przemyslaw BAKOWSKI
Adam PAWLAK
Ronald WAXMAN

Président
Rapporteur
Rapporteur
Directeur de Thèse
Examineur
Examineur

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT
ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PAREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES
DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CONSTANT Eugène	Electronique
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie Appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean-Pierre	Chimie-Physique
M. BOSCOQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BREZINSKI Claude	Analyse Numérique

M. BRIDOUX Michel
 M. CELET Paul
 M. CHAMLEY Hervé
 M. COEURE Gérard
 M. CORDONNIER Vincent
 M. DAUCHET Max
 M. DEBOURSE Jean-Pierre
 M. DHAINAUT André
 M. DOUKHAN Jean-Claude
 M. DYMENT Arthur
 M. ESCAIG Bertrand
 M. FAURE Robert
 M. FOCT Jacques
 M. FRONTIER Serge
 M. GRANELLE Jean-Jacques
 M. GRUSON Laurent
 M. GUILLAUME Jean
 M. HECTOR Joseph
 M. LABLACHE-COMBIER Alain
 M. LACOSTE Louis
 M. LAVEINE Jean-Pierre
 M. LEHMANN Daniel
 Mme LENOBLE Jacqueline
 M. LEROY Jean-Marie
 M. LHOMME Jean
 M. LOMBARD Jacques
 M. LOUCHEUX Claude
 M. LUCQUIN Michel
 M. MACKE Bruno
 M. MIGEON Michel
 M. PAQUET Jacques
 M. PETIT Francis
 M. POUZET Pierre
 M. PROUVOST Jean
 M. RACZY Ladislav
 M. SALMER Georges
 M. SCHAMPS Joel
 M. SEGUIER Guy
 M. SIMON Michel
 Melle SPIK Geneviève
 M. STANKIEWICZ François
 M. TILLIEU Jacques
 M. TOULOTTE Jean-Marc
 M. VIDAL Pierre
 M. ZEYTOUNIAN Radyadour

Chimie-Physique
 Géologie Générale
 Géotechnique
 Analyse
 Informatique
 Informatique
 Gestion des Entreprises
 Biologie Animale
 Physique du Solide
 Mécanique
 Physique du Solide
 Mécanique
 Métallurgie
 Ecologie Numérique
 Sciences Economiques
 Algèbre
 Microbiologie
 Géométrie
 Chimie Organique
 Biologie Végétale
 Paléontologie
 Géométrie
 Physique Atomique et Moléculaire
 Spectrochimie
 Chimie Organique Biologique
 Sociologie
 Chimie Physique
 Chimie Physique
 Physique Moléculaire et Rayonnements Atmosph.
 E.U.D.I.L.
 Géologie Générale
 Chimie Organique
 Modélisation - calcul Scientifique
 Minéralogie
 Electronique
 Electronique
 Spectroscopie Moléculaire
 Electrotechnique
 Sociologie
 Biochimie
 Sciences Economiques
 Physique Théorique
 Automatique
 Automatique
 Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne
 M. ANDRIES Jean-Claude
 M. ANTOINE Philippe
 M. BART André
 M. BASSERY Louis

Composants Electroniques
 Biologie des organismes
 Analyse
 Biologie animale
 Génie des Procédés et Réactions Chimiques

Mme BATTIAU Yvonne
 M. BEGUIN Paul
 M. BELLET Jean
 M. BERTRAND Hugues
 M. BERZIN Robert
 M. BKOUCHE Rudolphe
 M. BODARD Marcel
 M. BOIS Pierre
 M. BOISSIER Daniel
 M. BOIVIN Jean-Claude
 M. BOUQUELET Stéphane
 M. BOUQUIN Henri
 M. BRASSELET Jean-Paul
 M. BRUYELLE Pierre
 M. CAPURON Alfred
 M. CATTEAU Jean-Pierre
 M. CAYATTE Jean-Louis
 M. CHAPOTON Alain
 M. CHARET Pierre
 M. CHIVE Maurice
 M. COMYN Gérard
 M. COQUERY Jean-Marie
 M. CORIAT Benjamin
 Mme CORSIN Paule
 M. CORTOIS Jean
 M. COUTURIER Daniel
 M. CRAMPON Norbert
 M. CROSNIER Yves
 M. CURGY Jean-Jacques
 Mlle DACHARRY Monique
 M. DEBRABANT Pierre
 M. DEGAUQUE Pierre
 M. DEJAEGER Roger
 M. DELAHAYE Jean-Paul
 M. DELORME Pierre
 M. DELORME Robert
 M. DEMUNTER Paul
 M. DENEL Jacques
 M. DE PARIS Jean Claude
 M. DEPREZ Gilbert
 M. DERIEUX Jean-Claude
 Mlle DESSAUX Odile
 M. DEVRAINNE Pierre
 Mme DHAINAUT Nicole
 M. DHAMELIN COURT Paul
 M. DORMARD Serge
 M. DUBOIS Henri
 M. DUBRULLE Alain
 M. DUBUS Jean-Paul
 M. DUPONT Christophe
 Mme EVRARD Micheline
 M. FAKIR Sabah
 M. FAUQUAMBERGUE Renaud

3

Géographie
 Mécanique
 Physique Atomique et Moléculaire
 Sciences Economiques et Sociales
 Analyse
 Algèbre
 Biologie Végétale
 Mécanique
 Génie Civil
 Spectroscopie
 Biologie Appliquée aux enzymes
 Gestion
 Géométrie et Topologie
 Géographie
 Biologie Animale
 Chimie Organique
 Sciences Economiques
 Electronique
 Biochimie Structurale
 Composants Electroniques Optiques
 Informatique Théorique
 Psychophysiologie
 Sciences Economiques et Sociales
 Paléontologie
 Physique Nucléaire et Corpusculaire
 Chimie Organique
 Tectonique Géodynamique
 Electronique
 Biologie
 Géographie
 Géologie Appliquée
 Electronique
 Electrochimie et Cinétique
 Informatique
 Physiologie Animale
 Sciences Economiques
 Sociologie
 Informatique
 Analyse
 Physique du Solide - Cristallographie
 Microbiologie
 Spectroscopie de la réactivité Chimique
 Chimie Minérale
 Biologie Animale
 Chimie Physique
 Sciences Economiques
 Spectroscopie Hertzienne
 Spectroscopie Hertzienne
 Spectrométrie des Solides
 Vie de la firme (I.A.E.)
 Génie des procédés et réactions chimiques
 Algèbre
 Composants électroniques

M. FONTAINE Hubert
 M. FOUQUART Yves
 M. FOURNET Bernard
 M. GAMBLIN André
 M. GLORIEUX Pierre
 M. GOBLOT Rémi
 M. GOSSELIN Gabriel
 M. GOUDMAND Pierre
 M. GOURIEROUX Christian
 M. GREGORY Pierre
 M. GREMY Jean-Paul
 M. GREVET Patrice
 M. GRIMBLLOT Jean
 M. GUILBAULT Pierre
 M. HENRY Jean-Pierre
 M. HERMAN Maurice
 M. HOUDART René
 M. JACOB Gérard
 M. JACOB Pierre
 M. Jean Raymond
 M. JOFFRE Patrick
 M. JOURNAL Gérard
 M. KREMBEL Jean
 M. LANGRAND Claude
 M. LATTEUX Michel
 Mme LECLERCQ Ginette
 M. LEFEBVRE Jacques
 M. LEFEBVRE Christian
 Mlle LEGRAND Denise
 Mlle LEGRAND Solange
 M. LEGRAND Pierre
 Mme LEHMANN Josiane
 M. LEMAIRE Jean
 M. LE MAROIS Henri
 M. LEROY Yves
 M. LESENNE Jacques
 M. LHENAFF René
 M. LOCQUENEUX Robert
 M. LOSFELD Joseph
 M. LOUAGE Francis
 M. MAHIEU Jean-Marie
 M. MAIZIERES Christian
 M. MAURISSON Patrick
 M. MESMACQUE Gérard
 M. MESSELYN Jean
 M. MONTEL Marc
 M. MORCELLET Michel
 M. MORTREUX André
 Mme MOUNIER Yvonne
 Mme MOUYART-TASSIN Annie Françoise
 M. NICOLE Jacques
 M. NOTELET François
 M. PARSY Fernand

4

Dynamique des cristaux
 Optique atmosphérique
 Biochimie Structurale
 Géographie urbaine, industrielle et démog.
 Physique moléculaire et rayonnements Atmos.
 Algèbre
 Sociologie
 Chimie Physique
 Probabilités et Statistiques
 I.A.E.
 Sociologie
 Sciences Economiques
 Chimie Organique
 Physiologie animale
 Génie Mécanique
 Physique spatiale
 Physique atomique
 Informatique
 Probabilités et Statistiques
 Biologie des populations végétales
 Vie de la firme (I.A.E.)
 Spectroscopie hertzienne
 Biochimie
 Probabilités et statistiques
 Informatique
 Catalyse
 Physique
 Pétrologie
 Algèbre
 Algèbre
 Chimie
 Analyse
 Spectroscopie hertzienne
 Vie de la firme (I.A.E.)
 Composants électroniques
 Systèmes électroniques
 Géographie
 Physique théorique
 Informatique
 Electronique
 Optique-Physique atomique
 Automatique
 Sciences Economiques et Sociales
 Génie Mécanique
 Physique atomique et moléculaire
 Physique du solide
 Chimie Organique
 Chimie Organique
 Physiologie des structures contractiles
 Informatique
 Spectrochimie
 Systèmes électroniques
 Mécanique

M. PECQUE Marcel
M. PERROT Pierre
M. STEEN Jean-Pierre

5
Chimie organique
Chimie appliquée
Informatique

Je voudrais tout d'abord remercier Monsieur Vincent Cordonnier, Professeur à l'université de Lille I, qui a bien voulu accepter la présidence du Jury.

Mes plus vifs remerciements vont à Monsieur Bernard Toursel, Professeur à l'E.U.D.I.L.¹ et à l'université de Lille I, qui a accepté la tâche de rapporteur. J'aimerais aussi le remercier de m'avoir accueilli dans son équipe lorsque, mon directeur de Thèse étant parti à Nantes, je me suis retrouvé en quelque sorte orphelin administratif.

Mes plus sincères remerciements vont à Monsieur Jean-Paul Calvez, Professeur à l'I.R.E.S.T.E.² de Nantes, qui a bien voulu être mon second³ rapporteur, et dont j'ai apprécié la pertinence des remarques.

J'aimerais ensuite témoigner ma sincère reconnaissance à Monsieur Przemyslaw Bakowski, aujourd'hui professeur à l'I.R.E.S.T.E. et directeur de cette Thèse, pour l'aide et les conseils qu'il m'a prodigués tout au long de ce travail. Il a su donner une dimension à mes recherches en me poussant à effectuer des stages aux états-unis et en Allemagne Fédérale.

Je voudrais remercier vivement Monsieur Adam Pawlak, chercheur au G.M.D.⁴ de Sank Augustin en Allemagne Fédérale, avec qui la collaboration à la rédaction de rapports et articles fut non seulement enrichissante mais accompagnée de rapports humains chaleureux.

C'est non sans une pointe d'émotion que je voudrais témoigner ma réelle gratitude et ma profonde admiration à Monsieur Ronald Waxman, *Principal Scientist* au C.S.I.S.⁵ de l'université de Charlottesville en Virginie. Outre l'aide et l'encadrement qu'il m'a apportés au cours de mon stage, j'ai énormément apprécié l'accueil chaleureux qu'il m'a réservé dans sa demeure ainsi que la visite guidée de Washington D.C. et de ses environs.

Je voudrais aussi saluer Madame Lisel Borns, qui m'a hébergé au cours de différents stages en Allemagne, gommant notamment les difficultés dues à la vie dans un pays dont on ne parle que très mal la langue.

1. Ecole Universitaire D'Ingénieurs de Lille.

2. Institut de Recherche et d'Enseignement Supérieur aux Techniques de l'Electronique.

3. Ce terme fait référence à l'ordre alphabétique.

4. Gesellschaft für Mathematik und Datenverarbeitung MBH, centre allemand de recherche pour l'informatique.

5. Center for Semicustom Integrated Circuits.

Chapitre 1

<i>Introduction.....</i>	<i>7</i>
--------------------------	----------

Chapitre 2

<i>Le domaine de la CAO</i>	<i>11</i>
-----------------------------------	-----------

2.1 Les étapes de la CAO.....	11
2.1.1 La modélisation.....	11
a. Le niveau d'abstraction	13
b. Le domaine de description.....	14
c. Le modèle temporel	17
2.1.2 La simulation.....	20
a. La modélisation du temps.....	20
b. Le mode de fonctionnement du simulateur	21
c. Le système logique utilisé	23
2.1.3 Le placement et le routage.....	24
a. Le partitionnement.....	25
b. Le placement	26
c. Le routage.....	27
d. Le test	28
2.2 La tendance actuelle	29
2.2.1 Introduction.....	29
2.2.2 L'approche incrémentale	29
a. Les niveaux d'abstraction.....	30
b. L'approche progressive	33
2.2.3 L'intégration.....	37
a. Le problème	37
b. L'intégration des langages.....	39
c. Les outils	41
2.2.4 La communication	45
a. VHDL	45
b. EDIF	45
2.3 Les simulateurs.....	46
2.3.1 L'interprétation directe.....	46
2.3.2 L'interprétation indirecte.....	47
2.3.3 La compilation directe.....	48
2.3.4 La compilation indirecte.....	49
2.4 Conclusion.....	51

Chapitre 3

Vers des simulateurs efficaces 53

3.1 Les solutions actuelles	53
3.1.1 Les solutions matérielles	53
3.1.2 Les solutions logicielles	54
3.2 Objectif du travail	55
3.3 Choix généraux	56
3.4 Présentation théorique de la solution	60
3.4.1 Naissance de l'idée	60
3.4.2 Définition de la synthèse	61
3.4.3 Le fonctionnement de la synthèse	63
a. Bases relatives à la conception de parties contrôles	63
b. La décomposition des parties contrôles	64
c. Les étapes de la synthèse	65
3.5 Emploi de la synthèse lors de l'exécution	72
3.6 Conclusion	73

Chapitre 4

L'implémentation..... 75

4.1 L'environnement.....	75
4.1.1 Le langage LIDO.....	75
a. Les types de données	76
b. Les facilités de déclarations.....	78
c. Opérations et conditions	79
d. Les unités de descriptions.....	80
e. Synchronisation et activation	82
4.1.2 Le système LIDO	84
a. L'interface utilisateur	84
b. Réalisation de l'exécution	86
4.2 L'évolution de l'outil.....	86
4.2.1 Version prospective initiale	86
4.2.2 La version actuelle	87
4.3 Le code du logiciel	89
4.3.1 La simulation.....	90
a. Le code d'un simulateur	91
b. Le simulateur exécutable	102
c. L'exécution.....	104
4.3.2 La génération	104

a. L'analyse	105
b. Les transformations	110
c. La production du code C	113
4.3.3 Quelques chiffres	114
4.4 Conclusion	114

Chapitre 5

<i>Evaluations de l'outil</i>	<i>115</i>
-------------------------------------	------------

5.1 Contraintes d'évaluation	115
5.2 Les critères d'évaluation	117
5.3 Estimation des ressources nécessaires	119
5.3.1 L'analyse de régression	119
5.3.2 De la théorie à la pratique	120
a. Prédiction de l'espace mémoire occupé	121
b. Prédiction du temps de simulation	125
5.4 Les résultats	127
5.4.1 Les circuits simulés	127
5.4.2 Résultats généraux	128
a. Importance de la partie contrôle	128
b. Avantage de la microprogrammation	129
c. Avantage des graphes	130
d. Limites de la traduction algorithmique	131
5.4.3 Efficacité de la synthèse	131
a. Résultats locaux	131
b. Résultats globaux	133
5.4.4 Efficacité du logiciel	134
5.5 Conclusion	136

Chapitre 6

<i>Conclusion</i>	<i>139</i>
-------------------------	------------

A Annexe	141
A.1 Description en LIDO du processeur Riscp	141
A.2 Modèle C du processeur Riscp	144

Bibliographie	151
----------------------------	------------

Fig. 2.1	Les étapes de la CAO	12
Fig. 2.2	Différents niveaux de représentation graphique	13
Fig. 2.3	Caractérisation d'une description par l'abstraction	14
Fig. 2.4	Représentation arborescente d'une description hiérarchisée	15
Fig. 2.5	Caractérisation des descriptions par le domaine et l'abstraction	16
Fig. 2.6	Temps de propagation fréquemment utilisés	18
Fig. 2.7	Temps impératif et indicatif	19
Fig. 2.8	Connexions verticales et horizontales	27
Fig. 2.9	Représentations et niveaux au cours de la CAO	31
Fig. 2.10	Modélisation/simulation incrémentale	34
Fig. 2.11	Extraction pour la simulation temporelle fine	36
Fig. 2.12	Représentations et traducteurs	39
Fig. 2.13	Compilateurs de silicium pour un processus minimal	43
Fig. 2.14	Les simulateurs compilés	49
Fig. 2.15	Méthodes de simulation	50
Fig. 3.1	Structure du générateur	56
Fig. 3.2	Adaptation de la machine cible à la machine hôte	59
Fig. 3.3	Importance de la partie contrôle	61
Fig. 3.4	Différentes compositions d'un simulateur	62
Fig. 3.5	Algorithme d'interprétation d'une instruction	64
Fig. 3.6	Séparation du contrôle et des données dans un graphe	67
Fig. 3.7	Transformation d'un graphe de contrôle en PLACS	69
Fig. 3.8	Compression des données avant la génération	71
Fig. 3.9	Emploi des données à l'exécution	73
Fig. 4.1	Ordre d'activation des Unités de Descriptions	83
Fig. 4.2	Organisation du système LIDO	84
Fig. 4.3	Menu principal du système LIDO	85
Fig. 4.4	Minimisation du nombre de fonctions run-time	89
Fig. 4.5	Sélection optimisée d'une partie des bits d'une variable	92
Fig. 4.6	Exemples de SDs utilisées pendant la simulation	93
Fig. 4.7	Complexité du chargement des opérandes longs	95
Fig. 5.1	Courbes de la taille d'une structure de contrôle	124
Fig. 5.2	Courbes du temps de simulation d'une structure de contrôle	127
Fig. 5.3	Part relative de la partie contrôle d'un modèle C	128
Fig. 5.4	Intérêt des versions microprogrammées	129
Fig. 5.5	Intérêt des versions à base de graphes	130
Tab. 5.1	Prédiction de la taille d'une structure de contrôle	123
Tab. 5.2	Prédiction de la taille des données	124
Tab. 5.3	Prédiction du temps de simulation d'une structure de contrôle	126
Tab. 5.4	Comparaison des résultats pour les structures de contrôle	132
Tab. 5.5	Comparaison des résultats pour les parties contrôles	133
Tab. 5.6	Temps de simulation du compteur 16-bit	135
Tab. 5.7	Temps de simulation de la partie contrôle de Riscp	136

Chapitre 1

Introduction

La fin des années cinquante a marqué les premières utilisations de l'informatique en tant qu'aide à la conception d'ordinateurs [CGK58, ADW59]. Si la Conception Assistée par Ordinateur (CAO) couvre de nos jours d'autres domaines tels que l'automobile et le bâtiment par exemple, cette Thèse se limite aux problèmes inhérents à la conception d'ordinateurs. Ainsi, le terme CAO sera utilisé tout au long de ce texte dans le sens de *l'application des ordinateurs d'aujourd'hui à la conception des ordinateurs de demain*.

L'objectif général de la CAO est de réduire le temps total de conception des circuits, et par la même occasion leur coût. Les outils de CAO visent à:

- a. gérer les différentes représentations d'un circuit au cours des phases successives du processus,
- b. aider les concepteurs à prendre des décisions en leur permettant d'évaluer l'impact de chacun des choix qui s'offrent à eux,
- c. les remplacer dans des tâches qui ne nécessitent pas de décisions,
- d. les aider à vérifier leur design.

L'évolution de la technologie des circuits (la surface utile de la puce double tous les cinq ans, la résolution des motifs élémentaires double tous les cinq ans, le nombre de niveaux de métallisation augmente) provoque un accroissement de la densité des composants au millimètre carré et donc une augmentation de la complexité (qui double tous les ans).

Des circuits intégrés "préhistoriques" SSI¹ (des centaines de composants) aux VLSI¹ (des centaines de milliers de composants) d'hier et d'aujourd'hui, et aux premiers ULSI¹ (des millions de composants), le processus de conception a évolué vers un ensemble d'outils intégrés dans un même environnement permettant d'obtenir les masques d'un circuit à partir d'une description spécifiée à l'aide d'un langage de description de matériel.

Toutes ces évolutions ne sont pas sans conséquences sur le processus de CAO qui doit s'adapter à l'accroissement de complexité et aux besoins

1. SSI: Small Scale Integrated circuits,
VLSI: Very Large Scale Integrated circuits,
ULSI: Ultra Large Scale Integrated circuits.

nouveaux des concepteurs. Des outils toujours plus puissants, notamment de simulation, sont requis.

La simulation, qui représente environ soixante à soixante-dix pour cent du temps total du processus de conception, en est un rouage primordial. A mesure que s'accroît la complexité des circuits intégrés, il devient essentiel de réduire le temps de simulation dans l'optique de la diminution du temps total de conception, et donc de son coût.

Les outils proposés jusqu'alors présentent encore des inconvénients. En effet, soit les solutions proposées limitent à un petit nombre les utilisateurs potentiels de ces outils du fait de leur coût, soit elles restreignent à des circuits de faible taille ceux qui peuvent être traités.

L'objectif de cette Thèse est donc de proposer une solution efficace pour la simulation de circuits complexes sur les stations de travail courantes et qui ne soit pas dédiée à une technologie particulière ou à un type d'architecture de circuit.

Le deuxième chapitre de cette Thèse est consacré de façon générale à la CAO. Après la présentation de ses différentes étapes, quelques exemples significatifs de systèmes de CAO donneront un aperçu de la tendance actuelle. Le reste de ce chapitre sera consacré à la simulation logique de circuits, aux modes de fonctionnement des simulateurs, aux langages de description de matériel et aux niveaux d'abstraction. Les avantages et inconvénients des différentes approches de la simulation seront présentés, tant du point de vue théorique que sur le plan pratique.

Le troisième chapitre présente les objectifs de cette Thèse, basé sur l'analyse effectuée au chapitre précédent, en tant que contribution à l'amélioration des outils de simulation. La génération de simulateurs est la solution proposée pour atteindre les objectifs fixés et les choix effectués sont discutés dans ce chapitre. Cette solution utilise en outre une méthode de transformation de description qui en fait son originalité. La transformation de parties contrôles en une forme adaptée pour la simulation permet d'accroître la vitesse de simulation et surtout de réduire la taille des simulateurs générés, qui est l'inconvénient majeur de ce type de simulateur.

Le chapitre suivant traite de l'implémentation. Après le système et le langage LIDO, ce sont les algorithmes utilisés qui sont présentés. L'influence des choix effectués est à nouveau discutée ainsi que certains aspects du

compromis taille/vitesse des simulateurs.

Le chapitre cinq compare les résultats obtenus avec ceux d'autres simulateurs. Une partie importante des résultats est constituée de la prévision à l'aide d'équations de l'espace mémoire et du temps qui seront nécessaires à la simulation d'une architecture. Les architectures simulées ainsi que les résultats des simulations effectuées sont présentés et analysés.

La dernière partie constitue la conclusion de cette Thèse. Les objectifs, choix et caractéristiques principaux sont résumés.

Chapitre 2

Le domaine de la CAO

Ce chapitre constitue l'état de l'art de cette Thèse. Afin de situer le domaine de travail, la CAO et ses étapes sont présentées d'abord de façon générale. Celles-ci sont par la suite détaillées au fil de la description de la tendance actuelle en matière de conception de circuits intégrés.

Cette tendance est caractérisée par la prise en compte dans une description de plus en plus de détails dans un processus incrémental, par l'intégration des langages de description de matériel, des outils et, à un degré moindre, par la communication des circuits sous forme de modèles.

Enfin, les différentes manières de réaliser un outil de simulation sont présentées (section 2.3). L'analyse de leurs avantages et de leurs inconvénients respectifs servira de base à la définition de la solution proposée au chapitre suivant.

2.1 Les étapes de la CAO

La CAO est un processus qui consiste à produire une description "fabrifiable" d'un circuit à partir de sa spécification initiale (cahier des charges). Derrière cette définition très générale, se cachent plusieurs étapes distinctes. Si l'on excepte la documentation qui ne concerne que l'utilisation du circuit, les quatre principales phases, dont l'enchaînement est décrit dans la figure 2.1, sont les suivantes:

- a. la modélisation,
- b. la simulation,
- c. le placement et le routage,
- d. le test.

La suite de cette section donne les définitions et un aperçu général de ces étapes. Les notions élémentaires nécessaires à leur compréhension sont définies. Il faudra néanmoins y revenir plus en détail dans la section suivante afin de présenter la tendance actuelle en CAO.

2.1.1 La modélisation

Au cours de cette étape, le concepteur définit son circuit à l'aide d'un langage de description de matériel (noté HDL pour Hardware Description Language) ou de primitives graphiques. L'existence d'un éditeur graphique

permet de prendre en compte les contraintes d'espace dès le début du processus, ce qui est particulièrement important avec les circuits ULSI ou les ASIC¹s d'aujourd'hui. Idéalement, un environnement de modélisation autorise à la fois les modes d'entrée graphique et texte, et maintient à jour chacune des vues en les modifiant de façon automatique à toute nouvelle édition de l'une ou de l'autre forme.

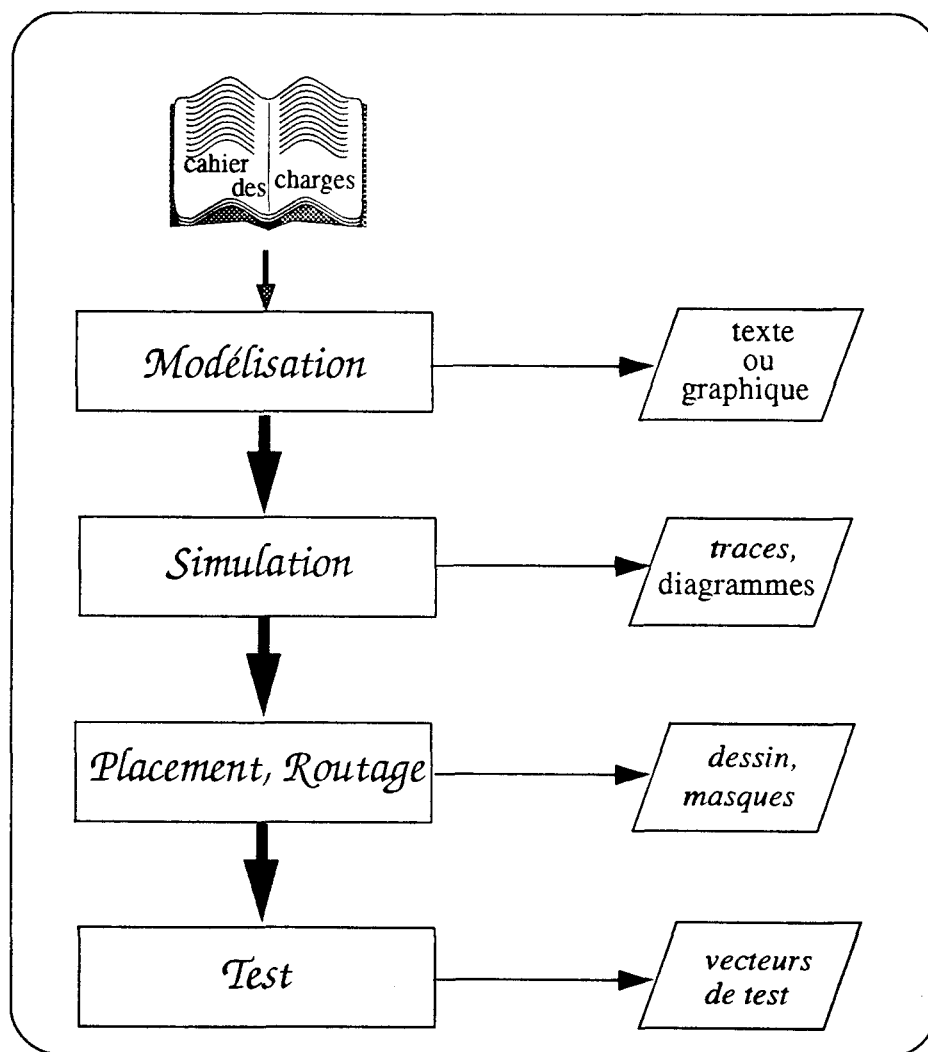


Figure 2.1 Les étapes de la CAO

Quel que soit le mode d'entrée disponible ou utilisé, le choix de la technologie ou de la logique d'implantation incombe au concepteur. Certains choix seront effectués très tôt alors que d'autres seront repoussés jusqu'aux phases finales de la conception. Dans tous les cas, la conception d'un circuit se fait de façon hiérarchisée dans un processus descendant.

1. Application Specific Integrated Circuits ou circuits intégrés dédiés à une application.

L'approche descendante proprement dite (*top-down*) consiste à représenter un circuit sous forme d'objets interconnectés, eux-même étant par la suite décomposés en sous-objets. De cette façon, un problème est divisé en problèmes moins complexes. C'est l'application de la célèbre stratégie "diviser pour régner".

L'approche remontante (*bottom-up*) consiste quant à elle, à construire des blocs de plus en plus complexes à partir des primitives existantes (portes, bascules, etc...). Les blocs ainsi produits (UALs, registres, etc...) sont ensuite utilisés comme primitives dans une approche descendante.

Une description textuelle peut être comportementale, structurelle ou un mélange des deux alors qu'une vue graphique est essentiellement structurelle. En effet, les éléments d'une telle vue sont des blocs et des connexions entre ces blocs. Un bloc peut être une primitive, une boîte noire associée à un certain comportement (fonction) ou une interconnexion de sous-blocs. Le terme primitive recouvre ici des entités aussi différentes qu'une porte logique ou une unité centrale.

Ceci introduit la notion de niveau d'abstraction que nous allons maintenant détailler ainsi que celles de domaine de description et de modèle temporel.

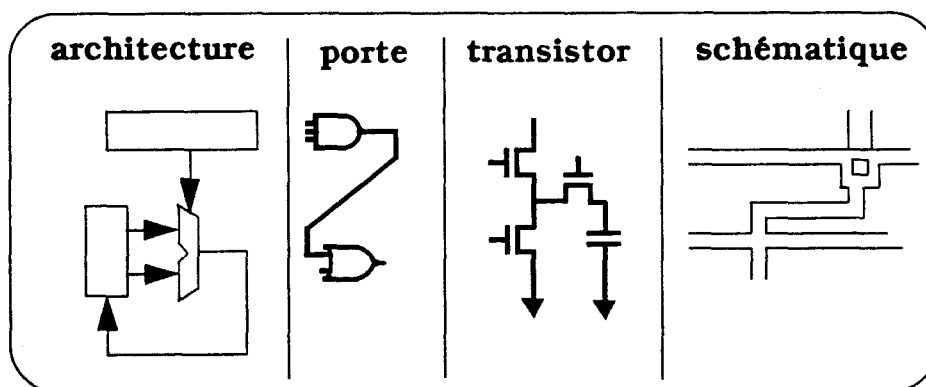


Figure 2.2 Différents niveaux de représentation graphique

a. Le niveau d'abstraction

L'abstraction est définie dans le micro Robert comme *le fait de considérer à part dans son esprit une qualité, une relation indépendamment de l'objet, des objets qu'on perçoit ou qu'on imagine.*

L'abstraction, appliquée à la description et à la simulation de circuits, permet de définir, par l'intermédiaire des niveaux d'abstraction, les objets

“visibles” que l'on peut décrire et manipuler lors d'une simulation.

Un niveau d'abstraction correspond donc à la précision avec laquelle on s'intéresse à un circuit. Ainsi, un niveau d'abstraction est souvent défini comme le niveau de détail avec lequel on décrit, ce qui est paradoxal si l'on considère que le mécanisme d'abstraction consiste à occulter des détails.

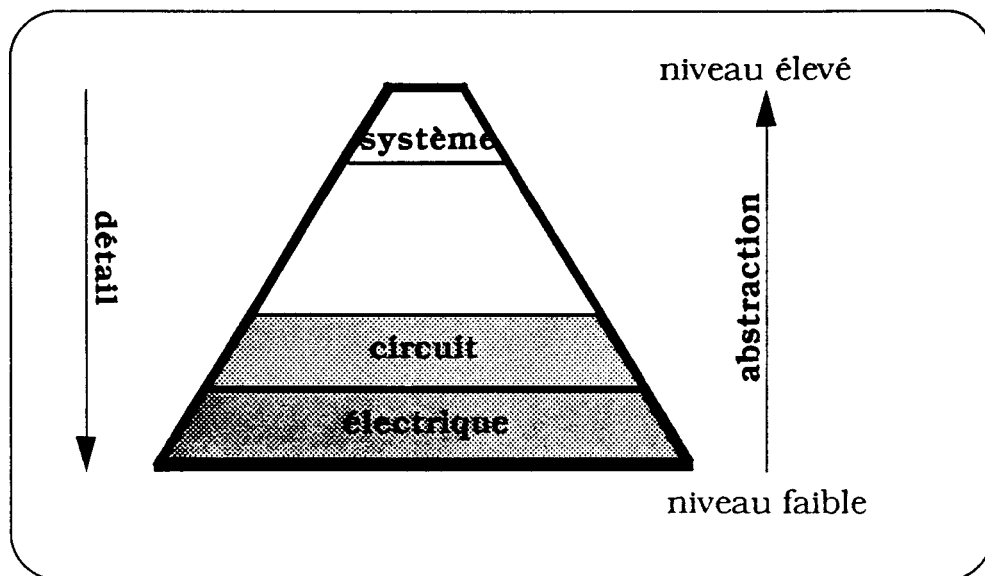


Figure 2.3 Caractérisation d'une description par l'abstraction

L'emploi de cette définition est dû au fait que l'abstraction est une notion abstraite alors que le détail est lui plus concret. Il suffit de lire la définition du Robert pour s'en convaincre. L'abstraction et l'approche "toujours plus de détails" vont dans deux directions opposées comme le montre la figure précédente. La représentation pyramidale de cette figure indique (de façon non quantitative) l'accroissement du nombre des objets à manipuler à mesure que l'on descend vers les bas niveaux d'abstraction.

Un langage qui couvre plusieurs niveaux d'abstraction est appelé langage multiniveau. De la même façon, on appelle simulateur multiniveau, un simulateur qui permet de simuler un modèle multiniveau, c'est-à-dire un modèle dans lequel cohabitent plusieurs niveaux. On parle dans ce cas de simulation multiniveau.

b. Le domaine de description

L'approche hiérarchisée utilisée par les concepteurs de circuits et qui consiste à découper un objet en plusieurs parties, chacune d'elles ayant une fonctionnalité bien précise, fait apparaître deux domaines de description:

- Le domaine structurel dans lequel un élément est décomposé en sous-éléments connectés entre eux.
- Le domaine comportemental où la fonctionnalité d'un élément est décrite en terme d'effet des entrées sur les sorties.

Un autre domaine est constitué par le domaine géométrique. Il intervient essentiellement au cours des phases de conception physique d'un circuit et liées à la simulation analogique.

Une description peut être représentée par un arbre dont les feuilles sont des éléments décrits de façon comportementale et dont les noeuds sont décrits de façon structurelle ou mixte.

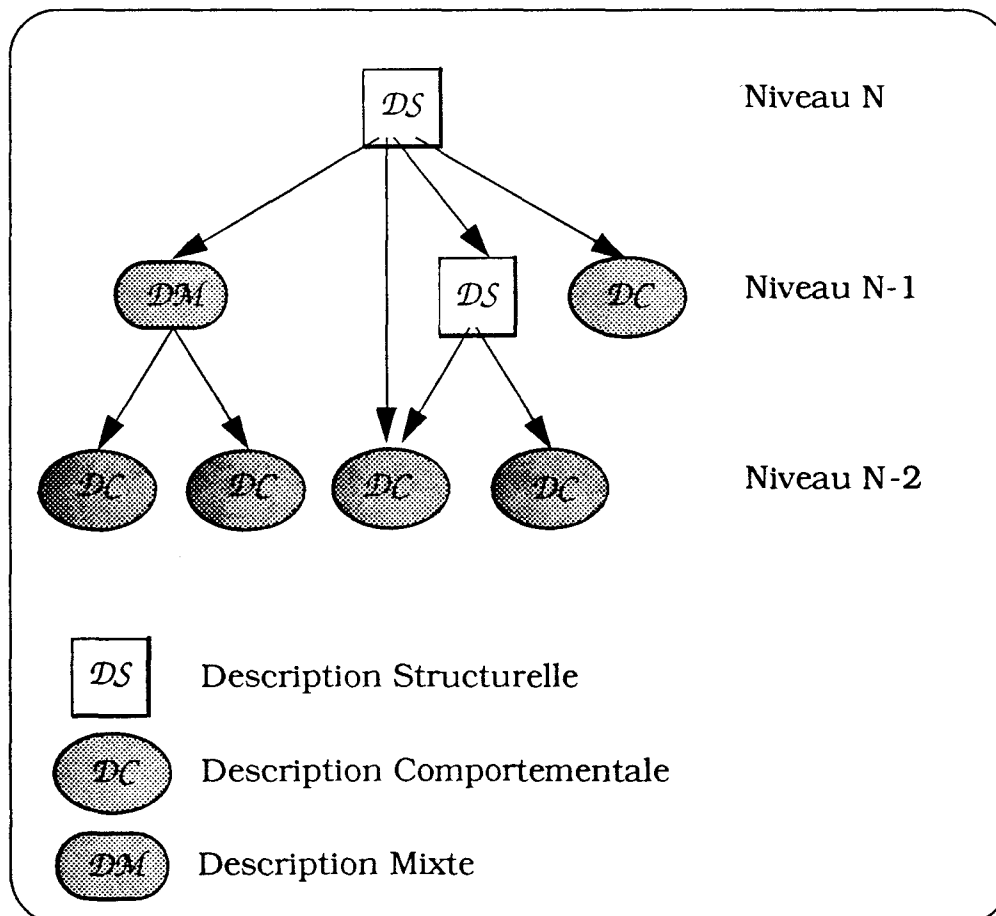


Figure 2.4 Représentation arborescente d'une description hiérarchisée

Une description mixte est caractérisée par la cohabitation d'éléments structurels (généralement sous forme d'instantiation d'objets) et d'éléments

comportementaux dont la nature dépend du niveau d'abstraction. Les objets instantiés dans un module décrit de façon structurée peuvent être modélisés à différents niveaux d'abstraction comme le montre la figure précédente. D'autre part, un même objet peut être instantié dans des descriptions à différents niveaux.

Le type de domaine permet d'ajouter une "dimension" à la pyramide présentée par la figure 2.3.

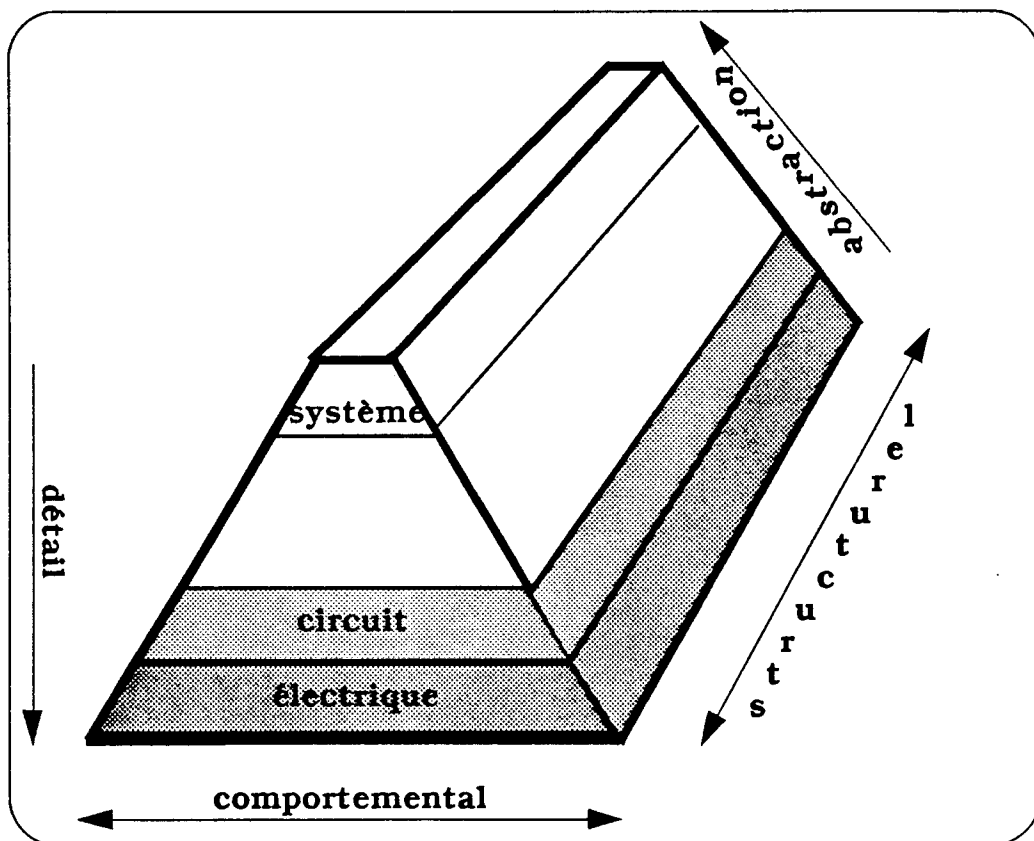


Figure 2.5 Caractérisation des descriptions par le domaine et l'abstraction

Les descriptions structurées prennent en compte plus de détails que les descriptions comportementales, c'est pourquoi la pyramide apparaît beaucoup plus longue que large.

Paradoxalement, à mesure que l'on décrit à des niveaux d'abstraction moins élevés, on perd de l'information comme nous le verrons à la section 2.2.

c. Le modèle temporel

Il ne faut pas confondre le modèle temporel, défini par le concepteur et associé à un module de description, avec la façon de modéliser le temps à l'intérieur d'un simulateur, vue au paragraphe 2.1.2.a. Le modèle temporel détermine la précision avec laquelle les temps de propagation notamment sont modélisés et simulés à l'intérieur d'une description. En règle générale, cette précision varie avec le niveau d'abstraction auquel on modélise. Ainsi, dans le cas d'une simulation purement fonctionnelle, le concepteur peut laisser de côté les temps de propagation. Les plus employés sont les suivants:

- *aucun*: dans le cas d'une simulation fonctionnelle. La valeur d'un signal n'est modifiée qu'un cycle de simulation plus tard si la simulation est dirigée par événement (voir figure 2.6 a).
- *fixes*: les temps de propagation à travers un objet peuvent être différents pour chaque sortie, mais ils sont identiques quelle que soit la nouvelle valeur (voir figure 2.6 b).
- *minimal, typique et maximal*: ces valeurs permettent d'effectuer une simulation plus précise du matériel (voir figure 2.6 c). Elles sont notamment nécessaires pour simuler le pire (respectivement le meilleur) des cas qui est la combinaison du temps de propagation maximal (respectivement minimal) et des caractéristiques temporelles (*setup* et *hold*) minimales (respectivement maximales) d'un circuit.
- *front montant et front descendant*: ces valeurs permettent d'associer des temps de propagation différents suivant la valeur transmise à une sortie (voir figure 2.6 d).

Dans une simulation au niveau porte, les deux derniers types de valeurs sont souvent combinés pour obtenir une simulation précise. L'analyse des "cours de signaux" ou hasards, qui est essentielle à ce niveau, par exemple dans le cas de bascules, peut ainsi être effectuée.



cs : cycle de simulation
 tP : temps de propagation
 HL: front descendant
 LH: front montant

a) retards nuls:

t	T	T+1	T+1 ⁺	T+2	T+3	T+3 ⁺
e_i	0	1	1	1	0	0
s_j	1	1	0	0	0	1

Diagram showing signal transitions for zero delay. Arrows indicate the propagation of a signal change from e_i to s_j . The first transition occurs between $T+1$ and $T+1^+$, and the second between $T+3$ and $T+3^+$. Each transition is labeled with a horizontal bar and 'cs' (simulation cycle).

b) retards fixes:

t	T	T+1	T+2	T+3	T+4	T+5	T+6
e_i	0	1	1	1	0	0	0
s_j	1	1	1	0	0	0	1

Diagram showing signal transitions for fixed delay. Arrows indicate the propagation of a signal change from e_i to s_j . The first transition occurs between $T+2$ and $T+3$, and the second between $T+5$ and $T+6$. Each transition is labeled with a horizontal bar and 'tP' (propagation time).

c) retards minimal, typique ou maximal:

Ce cas est une variante du précédent. Après sélection du type de retard, on se trouve dans le cas d'un retard fixe.

d) retards à front montant ou descendant:

t	T	T+1	T+2	T+3	T+4	T+5
e_i	0	1	1	0	0	0
s_j	1	1	0	0	0	1

Diagram showing signal transitions for rising or falling front delays. Arrows indicate the propagation of a signal change from e_i to s_j . The first transition occurs between $T+2$ and $T+3$, and the second between $T+4$ and $T+5$. The first transition is labeled with a horizontal bar and 'tPHL' (propagation time for high-to-low), and the second is labeled with a horizontal bar and 'tPLH' (propagation time for low-to-high).

Figure 2.6 Temps de propagation fréquemment utilisés

Une autre distinction relative aux temps de propagation peut être faite. Elle est liée à la sémantique du langage de description concernant les retards. On distingue:

• Le retard "impératif" (*transport delay*). Il permet de spécifier un retard qui interviendra quelle que soit la durée de la valeur appliquée à une sortie (voir figure 2.7 a).

• Le retard "indicatif" (*inertial delay*) est essentiel pour la simulation au niveau porte. Il permet de spécifier un retard qui interviendra si et seulement si la valeur appliquée à une sortie a une durée suffisante, c'est-à-dire supérieure au temps indiqué (voir figure 2.7 b, c).

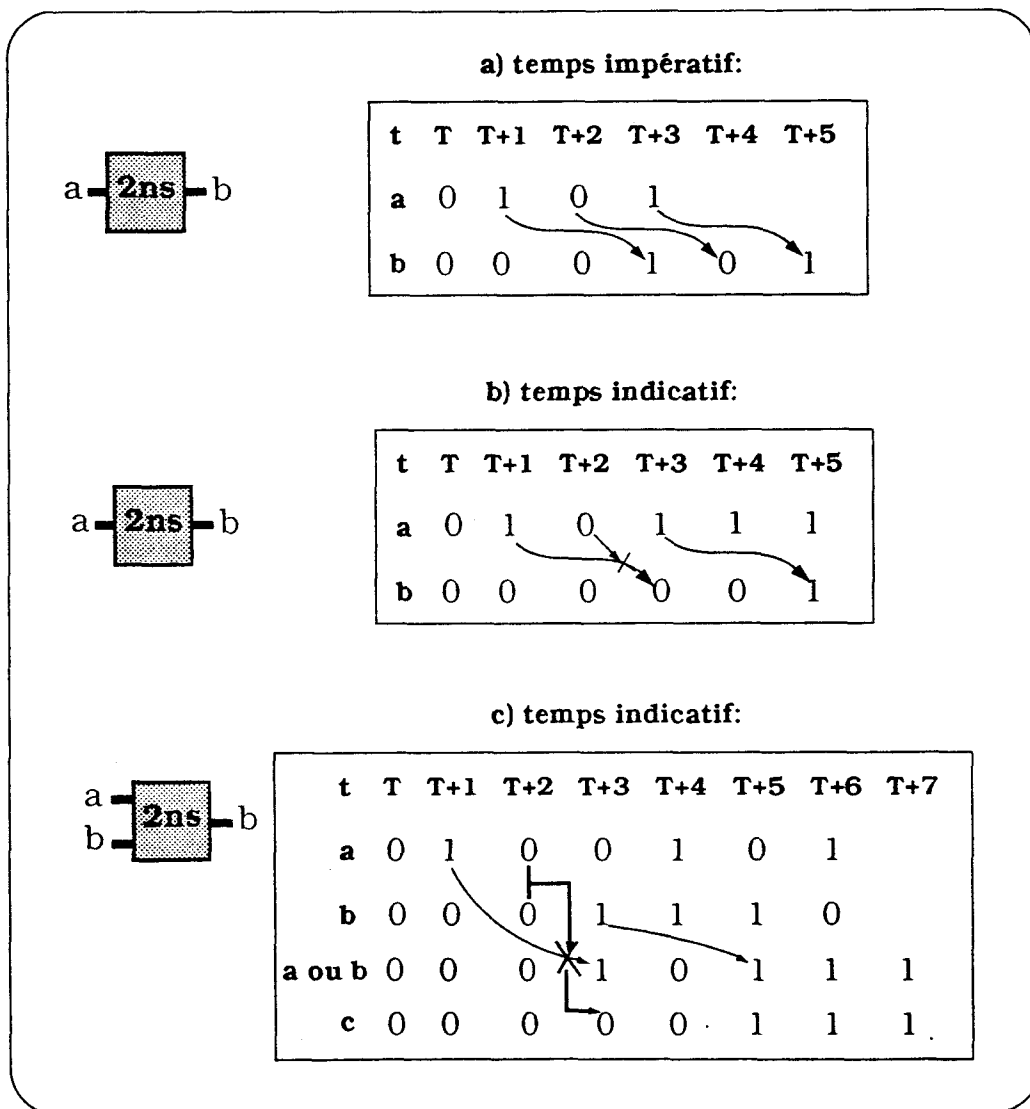


Figure 2.7 Temps impératif et indicatif

Les temps de propagation énumérés ci-dessus sont propres à un module et ne tiennent compte ni des temps de traversée des connexions entre plusieurs modules ni des temps de charge. Ces différentes valeurs et leur utilisation au niveau porte seront détaillées à la section 2.2.2.b car elle

requiert l'usage d'outils d'analyse.

2.1.2 La simulation

On peut faire une première distinction entre différents types de simulation.

1) La *simulation de pannes* ou *simulation de fautes*, liée à l'environnement de test, sera traitée ultérieurement.

2) La simulation de valeurs, appelée plus communément simulation est un procédé de vérification.

L'objectif de la simulation est de vérifier si un modèle est correct avant de passer à l'étape suivante. Un modèle est une représentation du circuit à concevoir. La simulation consiste donc à vérifier si un modèle a le comportement requis par le cahier des charges. La simulation est aussi appelée *vérification dynamique* par opposition aux outils d'analyse statique qui composent, avec les simulateurs, l'ensemble des outils de vérification.

De nombreux critères permettent de distinguer différents types de simulation. Ces critères sont liés au(x) simulateur(s) utilisé(s), au(x) langage(s) de description disponible(s) ainsi qu'au type de modèle défini par le concepteur. Une liste des critères majeurs est présentée ci-dessous.

a. La modélisation du temps

C'est de la modélisation du temps à l'intérieur d'un simulateur qu'il s'agit ici et non de sa représentation dans une description. Le temps dont il est question dans ce paragraphe est transparent pour l'utilisateur. Les différents types de modèles temporels qui sont utilisés dans des descriptions ont été décrits plus haut (voir paragraphe 2.1.1.c).

La modélisation du temps concerne la précision avec laquelle celui-ci est représenté dans un simulateur. Ceci permet de faire une première distinction entre simulation logique et simulation analogique. La simulation analogique est associée au niveau circuit et utilise les équations différentielles des transistors, capacités et résistances pour en déterminer par exemple la vitesse de commutation ou la consommation de courant d'un circuit. Dans ce cas, le temps est représenté de façon continu et les valeurs des signaux en fonction du temps définissent des courbes continues et dérivables. Si ce type de simulation est très précis, il est par contre très

coûteux tant en terme de mémoire nécessaire qu'en terme de temps requis pour la simulation. Cela limite considérablement le nombre d'éléments simulables en même temps, à moins que des machines spécialisées et très coûteuses ne soient employées.

La simulation logique est certes moins précise mais beaucoup plus efficace en terme de temps CPU, la différence d'efficacité dépendant notamment du niveau d'abstraction. L'échelle du temps est découpée en unités discrètes et fines telles que la seconde ou la milliseconde, en unités liées à l'architecture telles que le cycle ou la phase d'horloge ou encore en "tranches" dont la longueur varie au cours de la simulation. Dans ce dernier cas, on fait référence aux événements consécutifs qui délimitent les tranches de temps plutôt qu'aux laps de temps eux-mêmes. La finesse de modélisation du temps influe grandement sur l'efficacité de la simulation et est fortement liée au mode d'exécution des simulateurs.

b. Le mode de fonctionnement du simulateur

Il est défini par le mode de séquençement et d'exécution des opérations. Un simulateur "exécute" un circuit de façon relativement indépendante de son modèle en ce qui concerne le temps, ce qui peut sembler paradoxal. En réalité, le temps doit être représenté dans une description en fonction du ou des modes de simulation possibles et du ou des langages de description disponibles.

Un simulateur logique est un algorithme de simulation qui examine l'état du circuit à des intervalles de temps déterminés. Le type de ces intervalles et ce qui est exécuté par l'algorithme à ces instants caractérisent le mode de fonctionnement d'un simulateur.

Les instants qui définissent ces intervalles de temps correspondent pour la plupart des modes de simulation à des moments qui ont une correspondance dans le circuit réel.

Un premier mode de séquençement des opérations est la synchronisation sur top d'horloge. Dans ce cas, toutes les actions se produisant pendant un cycle horloge ou plus généralement une phase horloge, sont "alignées" sur le top/la phase horloge et exécutées "en même temps". Elles sont bien entendu exécutées dans un certain ordre mais les modifications apparaissent au même temps simulé. L'exécution des opérations est ordonné statiquement et certaines des opérations peuvent être exécutées en parallèle. Ce type de simulation s'adresse aux circuits synchrones et est associé à un niveau d'abstraction élevé tel que les niveaux

transfert de registre ou architecture pour lesquels l'horloge est un élément de base.

Un second mode de séquençement peut être dérivé du précédent. Il s'agit de découper le temps en tranches plus fines et régulières afin d'obtenir une simulation plus précise. Ce gain en précision va de pair avec plusieurs défauts considérables ce qui déprécie considérablement ce mode de simulation. Tout d'abord cette précision est toute relative car contrairement au cas précédent les temps obtenus ne sont que des approximations et nécessitent plus de précautions dans l'analyse des résultats. Ceci est de surcroît accompagné d'une perte de performance car c'est la totalité du circuit qui doit être évalué beaucoup plus souvent (de l'ordre de 50 à 100 fois plus). Enfin, ce mode de simulation n'a pas de correspondance physique dans le circuit, tant en ce qui concerne les intervalles de temps qu'en terme de niveau d'abstraction.

La simulation dirigée par événements est un mode de simulation radicalement différent. En effet, la durée des intervalles de temps séparant les instants où l'algorithme examine l'état du circuit dépend de sa description et des vecteurs de test fournis lors de la simulation. Mais avant d'aller plus loin il est nécessaire de donner quelques définitions:

- Une *transaction* est une action qui correspond à l'affectation d'une valeur à un élément du modèle. L'élément concerné n'est pas modifié ce qui ne provoque aucune modification de l'état du circuit. Ceci ne permet pas de distinguer la modification d'un élément et le simple "rafraîchissement" de son contenu.

- Un *événement* est une transaction qui apporte un changement au contenu d'un élément. L'état du circuit change et l'événement déclenche l'exécution des actions qui en dépendent.

Ainsi, la simulation d'un modèle peut être vue comme l'exécution d'une suite d'actions, chacune donnant lieu soit à une transaction soit à un événement. Ce mode de simulation consiste donc à limiter au maximum le nombre d'opérations à effectuer au cours d'une simulation.

La partie essentielle d'un simulateur dirigé par événements est un noyau (ordonnanceur) qui coordonne les actions à exécuter et gère les files d'attente associées à chaque élément.

Une telle simulation consiste en la répétition du cycle de simulation:

- 1) exécution des transactions en suspend,

- 2) identification des événements,
- 3) sélection des processus et actions qui en dépendent,
- 4) mise à jour des transactions en suspend.

Le cycle de simulation (CDS) est un concept sans relation avec le temps simulé et à un instant simulé quelconque, il peut (théoriquement) y avoir une infinité de CDSs. Ainsi, des événements se produisant à des CDSs consécutifs peuvent en réalité avoir lieu au même temps simulé.

C'est le plus répandu des modes de simulation car, outre son efficacité, il permet de simuler à la fois les circuits synchrones et asynchrones. Il est utilisé dans la plupart des simulateurs au niveau porte pour lequel il est particulièrement adapté.

Lorsqu'un simulateur autorise plusieurs modes lors de la simulation d'un même modèle, ce simulateur est appelé simulateur multimode. Il permet dans ce cas, de simuler les circuits combinatoires ainsi que les circuits séquentiels (synchrones et asynchrones).

c. Le système logique utilisé

Malgré les nombreux efforts de recherche pour implanter la logique multivaluée [Smi81, Hur84, KT81], seules les valeurs zéro et un "existent" actuellement en matériel. Pourtant la logique booléenne est insuffisante pour décrire de façon précise les circuits ou pour différencier les technologies d'implantation.

Des problèmes liés à l'initialisation de variables par les simulateurs ont amené à considérer tout d'abord une troisième valeur logique 'U' qui indique qu'une variable ou un signal n'est pas initialisé (U pour *uninitialized*). Ceci permet de simuler précisément lors de l'initialisation, des éléments tels que des bascules, dont l'état n'est pas connu; l'initialisation manuelle des sorties par le concepteur est une solution beaucoup trop longue et, qui plus est, source d'erreurs.

Pour différencier l'état non initialisé de l'état indéterminé, la valeur logique 'X' est à son tour introduite. Celle-ci permet de traiter les cas d'erreurs (par exemple lorsque les entrées J et K d'une bascule JK ont toutes deux la valeur un) et éventuellement de les propager. Enfin, pour permettre de simuler les circuits trois états, la valeur 'Z' est utilisée pour représenter la haute impédance. Ainsi, le système ('0', '1', 'U', 'X', 'Z') est le système logique de base pour la simulation. Parfois les valeurs 'X' et 'U' sont

fusionnées en une seule. Ce système de base ne permet pas de distinguer certaines technologies; c'est ainsi que la valeur zéro résistant (respectivement un résistant) doit être ajoutée pour la technologie Open Emitter (respectivement Open Collector). Ces valeurs sont représentées par 'L' (*Low*) et 'H' (*High*) respectivement. De plus, un système logique peut être constitué à la fois de valeurs logiques et de forces. Comme son nom l'indique, une force détermine la force d'un signal. Elle est utilisée pour résoudre les conflits entre plusieurs signaux.

Un simulateur est généralement associé à un système logique à l'aide duquel le concepteur doit décrire ses circuits. Le système logique dépend aussi du langage utilisé pour la simulation. Certains langages (VHDL par exemple) ne proposent pas de système logique particulier et c'est au concepteur de le définir. Enfin, certains vendeurs proposent des simulateurs basé sur des systèmes comprenant 46 ou même 128 valeurs [Coe89]! Ceci n'est d'ailleurs pas sans poser des problèmes de compatibilité entre des modèles écrits par plusieurs concepteurs ou par différentes compagnies. Si la normalisation des langages de description de matériel a abouti en décembre 1987 à la version IEEE 1076-1987 de VHDL, l'obtention d'un consensus sur la façon d'écrire les modèles est encore un problème ouvert [Dub90] sur lequel se penche la communauté scientifique depuis plusieurs années déjà. Mais le survol détaillé des systèmes logiques utilisés en CAO n'est pas l'objet de ce paragraphe.

2.1.3 Le placement et le routage

Cette étape conduit à une description physique du circuit à concevoir, partant d'une description logique. Elle est parfois appelée synthèse, mais il faut maintenant définir clairement ce que recouvre cette notion. La synthèse est la transformation d'une description d'un objet effectuée à un certain niveau en une description de ce même objet à un niveau plus bas. Cette définition est très générale. C'est pourquoi ce terme prête à confusion lorsqu'il est utilisé seul. En effet, il a souvent des sens différents suivant l'adjectif qualificatif qui lui est, ou qui devrait lui être associé. Ceci est d'autant plus vrai qu'apparaît depuis quelques années la notion de "synthèse de haut niveau" (voir section 2.2.3.c). Le type de synthèse dont il est question ici est la synthèse logique. La description physique qu'elle produit est constituée des masques du circuit.

C'est la première étape du processus de conception qui soit

véritablement dépendante des considérations technologiques, même si certaines données utilisées en simulation par exemple (c'est en particulier le cas des temps de propagation) varient sensiblement avec la technologie. C'est au cours de cette étape que sont identifiées les différences résultant des choix technologiques. Ces différences découlent de choix majeurs et sont atténuées ou accentuées par les caractéristiques du circuit. Les choix majeurs sont les suivants:

- la logique: positive ou inversée, cette dernière conduisant à des circuits plus compacts,
- la technologie MOS, ECL, TTL (compromis vitesse/densité),
- la technique de partitionnement, qui influence grandement les temps de propagation et les connexions
- le type d'implantation: standard cell (plus sûre), gate array et PLA (plus régulières, qui se prêtent à l'automatisation).

La conception physique est traditionnellement divisée en trois phases étroitement liées mais intervenant séparément. Elles sont présentées ci-dessous.

a. Le partitionnement

Cette phase consiste à "découper" un circuit en sous-éléments et à les répartir sur des unités physiques telles que puces, modules ou cartes en tenant compte des contraintes d'espace et de temps des parties critiques notamment. Deux types de techniques sont actuellement utilisées:

- Le partitionnement fonctionnel a pour objectif de rassembler tous les éléments d'un bloc ayant une fonction bien déterminée sur la même unité physique, aussi petite que possible. Ce type de partitionnement est encore, à l'heure actuelle, réalisé de façon manuelle car les outils automatiques qui utilisent cette méthode présentent de moins bons résultats.

- Le partitionnement basé sur les connexions a pour objectif premier de minimiser le nombre d'unités physiques en affectant un maximum de blocs à chaque unité dans la limite des broches disponibles.

Dans les deux cas, des tests préliminaires vérifient que le résultat du partitionnement ne dépasse pas les ressources disponibles et n'enfreint pas les règles élémentaires, de dissipation de courant par exemple.

b. Le placement

Le placement consiste à affecter une place physique à chacun des éléments du circuit. Son objectif est d'assurer un routage optimal de manière à diminuer les pertes de courant et à minimiser les temps de propagation à travers les connexions. Pour cela, la routabilité du circuit dans sa version courante est évaluée périodiquement au cours de cette phase.

Les éléments du circuit sont d'abord placés sur les puces, puis les puces sont disposées sur les modules et enfin les modules sont répartis sur les cartes. Le placement sur la puce est le plus complexe car il est le moins régulier et le nombre d'éléments à placer est plus élevé.

La puce est physiquement divisée en colonnes et lignes dont les intersections sont des cellules pouvant recevoir les éléments du circuit. L'ensemble est appelée une grille. Le placement est fondamentalement différent selon la manière de diviser la puce. On distingue ainsi deux types de placement:

- à grille inerte; c'est en quelque sorte un moule. Les éléments doivent impérativement être casés dans les cellules. Le placement doit s'adapter à la grille dont la nature, fixée et généralement régulière, se prête à l'automatisation de cette phase.

- à grille malléable; le nombre des colonnes est variable; des cellules plus grandes sont formées par regroupement de cellules pour placer des éléments fonctionnellement liés tels que les portes d'un compteur. Ce type de placement adapte la grille au circuit. Il laisse plus de liberté au concepteur mais est plus complexe. Il n'est pas totalement automatisé car certaines parties de la puce sont parfois réservées à des éléments mémoire devant être placés par le concepteur. C'est néanmoins ce type de placement qui est utilisé dans la plupart des systèmes dédiés à la technologie MOS.

Dans les deux cas, le placement est conceptuellement un problème simple. Il s'agit d'affecter les $\mathcal{N}$ éléments d'un circuit aux $\mathcal{N}$ cellules d'une grille. L'examen exhaustif de toutes les solutions ($\mathcal{N}!$) devient très rapidement impossible à mesure que $\mathcal{N}$ augmente. On opère alors un nouveau partitionnement jusqu'à obtenir des groupes d'éléments dont la complexité peut être traitée.

Une étape de transition avant le routage est l'affectation des ports d'entrée-sortie. Si certains d'entre-eux sont logiquement équivalents,

comme par exemple les entrées d'une porte ET, certaines associations entre l'ensemble des entrées d'une porte et celui des ports que l'on doit y connecter sont plus intéressantes que d'autres en terme de longueur de connexion.

c. Le routage

Le routage consiste à connecter entre-eux les éléments physiques du circuit, c'est-à-dire à déterminer les chemins physiques des ports de sortie d'éléments vers les ports d'entrée d'autres éléments, auxquels ils doivent être reliés.

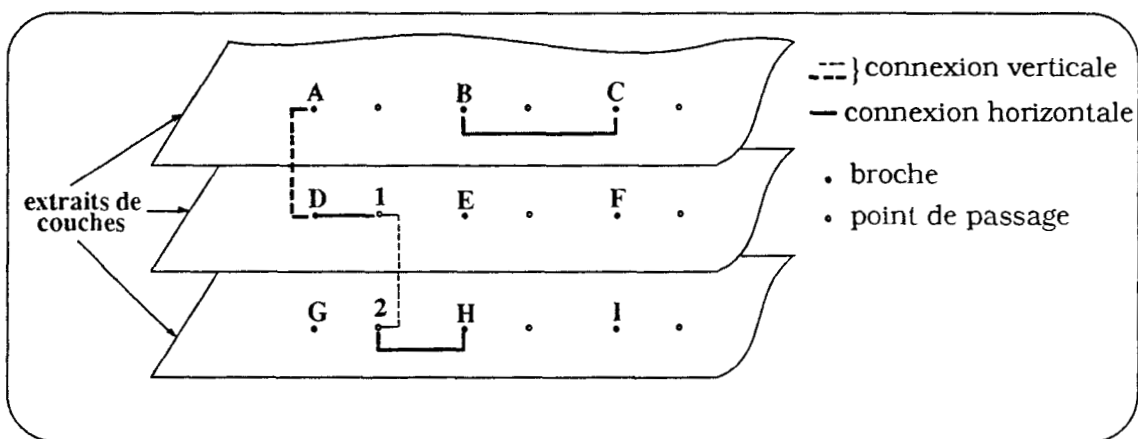


Figure 2.8 Connexions verticales et horizontales

Les connexions doivent être établies sur chaque grille, d'une grille à une autre et entre différentes couches. Les connexions entre couches se font au moyen de "points de passage" qui sont des points de conduction répartis équitablement sur une grille. Le nombre de ces points varie avec la technologie et la nature de l'unité physique. L'espace laissé entre les broches et les points de passage est utilisé pour implanter les canaux de conduction. Ceux-ci permettent de réaliser les connexions sur une même couche. On distingue donc:

- Les connexions horizontales qui relient une broche à une autre (liaison B-C) ou une broche à un point de passage (liaisons D-1 et 2-H) sur une même couche.

- Les connexions verticales qui relient deux broches entre-elles (liaison A-D) ou deux points de passage entre-eux (liaison 1-2) sur des couches différentes, séparées éventuellement par plus d'un niveau. Les connexions verticales sont le plus souvent orthogonales aux couches bien qu'on en

trouve qui soient inclinées à quarante-cinq degrés, de rares systèmes autorisant même des connexions verticales inclinées différemment.

Lorsque le routage est terminé, le circuit est presque réalisé. Pour cela, il ne reste plus qu'à en générer les masques et à envoyer le résultat à la fonderie afin d'obtenir une première réalisation physique (prototype). La génération de masques utilise les informations produites par le placement-routage et celles propres aux cellules élémentaires, stockées dans une bibliothèque.

d. Le test

Le test a pour but d'éliminer les circuits mauvais, c'est-à-dire qui ne correspondent pas à ce que l'on attend d'eux d'après le cahier des charges. Le test s'étend donc au-delà de l'environnement traditionnel de la CAO puisqu'il consiste à faire l'essai d'un produit fini réalisé par une fonderie. Néanmoins, le test commence avant la réalisation physique par la préparation des séquences de test; celle-ci est liée à la simulation de fautes qui intervient après la modélisation du circuit au niveau logique.

Cette approche du problème en terme de faute permet de rejeter les circuits défectueux sans avoir recours à la liste exhaustive des tests. D'ailleurs leur nombre est souvent tel que cela ne serait pas possible, en particulier lorsque l'on tient compte des contraintes temporelles.

On distingue deux types de test selon qu'il concerne le produit fini ou son modèle avant fabrication.

Le test du modèle:

Celui-ci a lieu avant la fabrication et prend beaucoup de temps. Mais il n'a qu'une influence limitée sur le coût final de production du fait qu'il intervient lors de la conception. Par contre, sa qualité est essentielle dans la mesure où il permet d'écarter plus tôt des circuits erronés. Il consiste à simuler des fautes intervenant à certains points d'entrée d'un circuit et à répertorier son comportement. Deux phases essentielles sont les suivantes:

- La génération de vecteurs de test. Ceux-ci sont destinés à être utilisés comme stimuli appliqués au circuit. Le but est de déterminer le plus petit nombre de ces vecteurs qui couvrent le maximum de types de faute. Les vecteurs ainsi construits forment, avec les masques du circuit, les données

nécessaires à la fabrication.

- La simulation de fautes. Elle permet de vérifier si l'application au produit fini des vecteurs de test précédemment construits, détectera ou non, les fautes pour lesquelles les séquences de test ont été définies. C'est en quelque sorte une simulation du test qui sera effectué sur le circuit final.

Le test du produit fini:

Ce test est une opération de contrôle qui doit être effectuée rapidement car son coût a un impact direct sur celui de la production. La découverte d'un défaut, dont la nature n'est pas distinguée lors du test, conduit à l'élimination du mauvais circuit.

2.2 La tendance actuelle

2.2.1 Introduction

Les étapes qui constituent l'ossature du processus de CAO ont été présentées de façon générale, sur le plan théorique en quelque sorte. Elles sont en fait beaucoup plus complexes dans la réalité et le concepteur doit procéder par phases successives au sein même d'une étape. De plus, des étapes telles que la modélisation et la simulation sont étroitement liées.

Cette section présente la tendance actuelle en CAO pour faire face à l'augmentation constante de la complexité des circuits. La tendance concerne essentiellement l'intégration des outils dans un même environnement et l'emploi d'une méthode de conception incrémentale.

2.2.2 L'approche incrémentale

Le processus de conception doit s'adapter à l'évolution de la technologie, de l'intégration et de la complexité des circuits. Il est clair que décrire d'emblée un circuit au niveau des portes sans modélisation ni simulation préalable aux niveaux d'abstraction supérieurs est une tâche que ne peut appréhender le cerveau humain.

Dans cette optique, l'approche incrémentale consiste à tenir compte de plus en plus de détails pour modéliser progressivement de façon plus fine le circuit.

a. Les niveaux d'abstraction

Un niveau d'abstraction détermine les éléments et les fonctions dont on dispose pour décrire un circuit à ce niveau. Comme l'a montré la figure 2.5, la nature du domaine de description introduit une distinction supplémentaire. Ainsi, une description structurelle à un niveau d'abstraction donné est effectuée à l'aide des primitives associées à ce niveau, alors qu'une description comportementale décrit le fonctionnement du type d'élément associé à ce niveau de façon algorithmique.

Une description comportementale est parfois appelée fonctionnelle. Une légère différence existe pourtant qui concerne le temps. Lorsque le temps est introduit dans un modèle, on parle de description comportementale alors que dans le cas contraire on parle de description purement fonctionnelle.

Il n'y a pas de classification normalisée des niveaux d'abstraction ni surtout de leur signification. La figure 2.9 est une extension du célèbre diagramme Y de Gajski-Kuhn [GK83]. Elle introduit le domaine géométrique qui intervient lors du placement-routage. La liste donnée ci-dessous comprend les niveaux les plus couramment mentionnés. Elle détaille pour chacun d'eux le type d'objets décrits, les primitives employées dans les descriptions structurelles ainsi que les méthodes de représentation utilisées dans les descriptions comportementales.

Niveau 0, électrique:

A ce niveau, on considère un circuit comme un réseau électrique. Les courants et les tensions sont exprimés en fonction du temps de façon continue à l'aide des lois de la physique. Il donne lieu à une simulation analogique. SPICE [VL80] est un langage célèbre dédié à la simulation à ce niveau.

Primitives structurelles: transistors, résistances, capacités,...

Méthodes: équations différentielles.

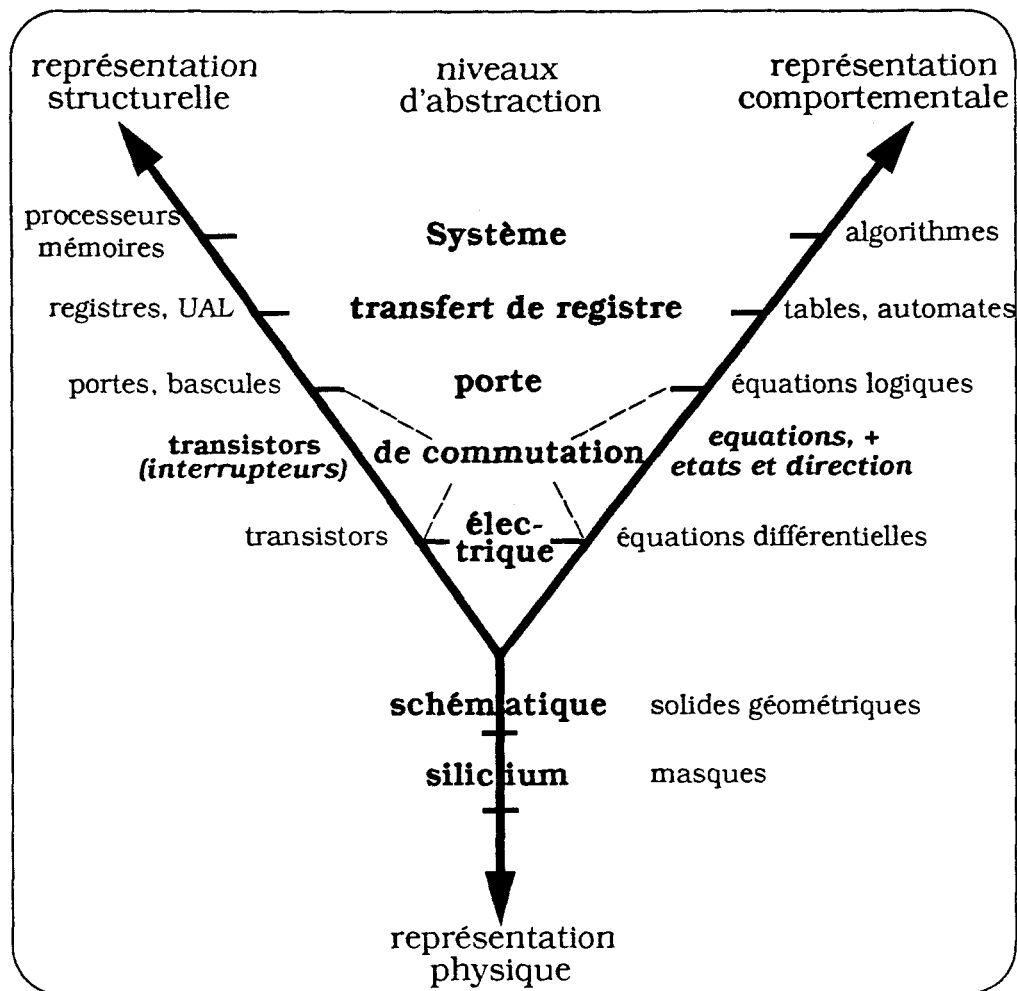


Figure 2.9 Représentations et niveaux au cours de la CAO

Niveau 1, de commutation:

Un circuit est considéré comme un réseau de commutation en ce sens qu'un transistor est vu comme un interrupteur (non orienté). Ce niveau ne correspond pas à une réalité physique mais à une manière de représenter les transistors, c'est pourquoi il apparaît sur la figure 2.9 comme un niveau optionnel supplémentaire. Ceci permet de manipuler dans une simulation logique des primitives plus élémentaires que les portes. On obtient ainsi une simulation plus fine tout en évitant une simulation analogique lourde. MOSSIM [Bry81] ou CASTOR du système CASCADE sont deux langages dédiés à ce niveau.

Primitives structurelles: transistors, résistances, capacités.

Méthodes: équations logiques, état des transistors.

Niveau 2, porte:

Un circuit est vu au niveau des portes et de modules élémentaires tels que les bascules. A ce niveau, on peut distinguer les parties combinatoires des parties séquentielles. La différence essentielle est que tous les éléments d'un bloc combinatoire sont potentiellement actifs en même temps alors que les éléments d'un bloc séquentiel sont activés tour à tour pour exécuter une séquence d'opérations. Le langage POLO du système CASCADE est typiquement dédié à ce niveau.

Primitives structurelles: portes et bascules.

Méthodes: équations logiques, diagrammes de transitions et tables de vérité.

Niveau 3, transfert de registre:

Alors que les niveaux précédents sont communs à tous les circuits numériques, celui-ci est le premier niveau où apparaît le concept de programme et d'instruction. Un circuit est décrit par les opérations qu'il exécute telles que la capture, le décodage et l'exécution d'une instruction. Ce niveau permet de distinguer la partie contrôle de la partie opérative. Certaines primitives utilisées à ce niveau sont aussi appelées blocs fonctionnels; c'est pourquoi il est parfois nommé niveau fonctionnel. De nombreux langages sont destinés à la simulation à ce niveau. Citons par exemple KARL, CASSANDRE, LIDO, RTS ainsi que ISPS [Bar81].

Primitives structurelles: registres, verrous, compteurs, UAL, etc...

Méthodes: tables de vérité, tables d'états, opérateurs, automates. Le type de donnée typiquement utilisé à ce niveau est le bit ou la chaîne de bits.

Niveau 4, algorithmique:

Ce niveau introduit des notions autres que matérielles telles que celles de système d'exploitation, de pagination, de segmentation, de mémoire cache et de langage de programmation. Les éléments de ce niveau sont considérés comme des modules semi-autonomes. Leur fonction est définie à l'aide d'algorithmes spécifiques. L'algorithme global peut être concurrent lorsque le circuit fait appel au pipelining. Ce niveau est parfois appelé niveau architecture comme pour le niveau LASCAR du système CASCADE.

Primitives structurelles: processeurs, mémoires et protocoles de communication.

Méthodes: algorithmes, types de donnée abstraits, réseaux de Petri interprétés.

Niveau 5, système:

Les primitives sont ici les mêmes que celles du niveau précédent mais on y décrit un circuit en termes radicalement différents. Ce niveau permet de décrire et de vérifier, lors de la simulation, les spécifications initiales concernant les performances, la fiabilité et le coût. Il sert surtout à évaluer et à classer les architectures. Il est aussi appelé PMS d'après la notation développée à l'université de Carnegie Mellon par Bell et Newell [BN70]. Celle-ci a donné lieu au système PMSL utilisé pour l'analyse de la performance d'un système.

Primitives structurelles: processeurs, mémoires et protocoles de communication.

Méthodes: les entrées-sorties sont décrites à l'aide d'algorithmes, théorie des files d'attente pour décrire les performances.

Les deux niveaux de la représentation physique de la figure 2.9 n'ont pas de représentation comportementale. On ne peut pas dire non plus qu'ils aient une représentation structurelle même s'ils représentent la structure finale du circuit. A ces deux niveaux, le circuit est représenté par le dessin. Le niveau silicium introduit avec les masques les dépendances technologiques telles que la largeur de fil et l'espace minimum entre eux. Il utilise les dimensions réelles des cellules et des connexions pour définir le circuit. La production de modèles à ces niveaux est automatisée même si le concepteur a parfois la possibilité de modifier la schématique.

Si le fait de décrire un circuit à un bas niveau d'abstraction donne accès à plus de détails qu'à un niveau supérieur, on peut néanmoins constater une perte d'information. Ainsi, la fonction logique n'est plus visible en-dessous du niveau porte, l'exécution des instructions définie au niveau transfert de registre est cachée au niveau porte, l'algorithme à exécuter a disparu au niveau transfert de registre et l'aspect performance n'apparaît qu'au niveau système.

b. L'approche progressive

Le nombre élevé de portes des circuits d'aujourd'hui nécessite que l'on ait recours à plusieurs étapes de modélisation/simulation afin de limiter les risques d'erreur. Cette méthode consiste à modéliser et à simuler un circuit d'abord à un haut niveau d'abstraction avant de passer à la modélisation/simulation aux niveaux inférieurs, dans un processus itératif.

Bien qu'on ait alors des risques d'erreur à chaque niveau, on obtient

une fiabilité globale plus importante puisque les spécifications du circuit sont vérifiées avant de passer à un autre niveau. De cette manière, la détection d'un mauvais fonctionnement du modèle à un niveau donné ne remet en cause que ce modèle et non pas les modèles des niveaux précédents.

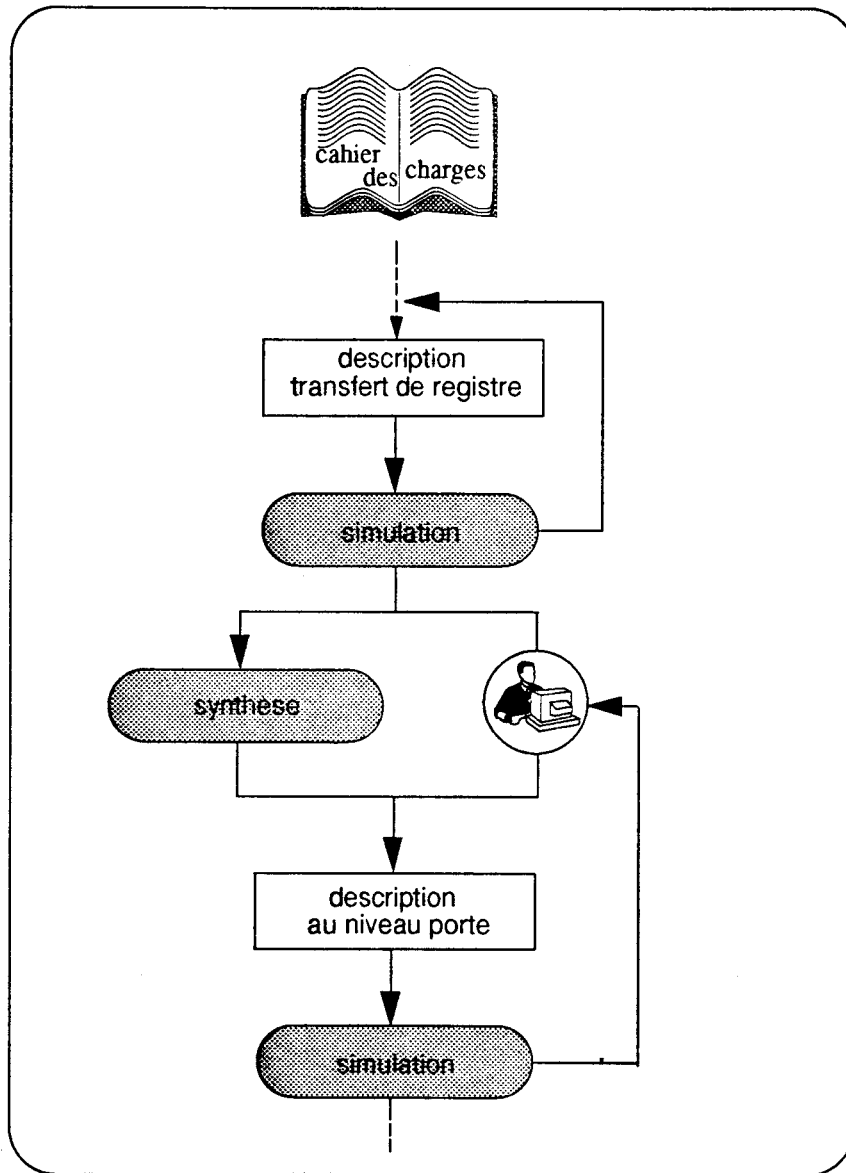


Figure 2.10 Modélisation/simulation incrémentale

Partant des spécifications du cahier des charges écrites en langage naturel, le concepteur doit donc écrire une première description du circuit. Dans l'approche qui nous intéresse ici, ce modèle est généralement défini au niveau transfert de registre ou algorithmique. En effet, le niveau système est

surtout utilisé pour évaluer le potentiel de la machine à concevoir et les choix architecturaux effectués de manière à modifier éventuellement les spécifications originelles. On procède ensuite à une simulation du modèle qui détermine s'il est fonctionnellement correct. Ce procédé est ensuite réitéré aux niveaux inférieurs jusqu'à obtenir un modèle au niveau porte ou au niveau de commutation. Le niveau final dépend essentiellement des caractéristiques du langage de description.

Le passage d'un niveau à un autre peut se faire à l'aide d'outils de synthèse (voir paragraphe 2.2.3) ou nécessiter l'écriture d'un nouveau modèle. Cette transition peut être partielle lorsque seulement une partie du modèle est réécrite à un/des niveau(x) inférieur(s), ce qui se produit quand les autres parties du modèle ont une correspondance connue en matériel. Sachant de quelle manière seront implantés ces blocs, il n'est pas utile dans leur cas de poursuivre plus loin le processus. Ceci est remis en question si l'on veut procéder à une simulation temporelle précise au niveau des portes.

L'approche progressive peut aussi concerner la description à un même niveau d'abstraction. Ce peut être le cas lorsque par exemple, on modélise les temps de propagation de plus en plus finement au niveau des portes. Ceci nécessite alors l'emploi d'analyseurs de descriptions ou d'extracteurs de la schématique afin de fournir les valeurs temporelles dépendantes de l'organisation physique du circuit et non pas seulement des cellules.

Cette tendance a subi un regain d'intérêt ces dernières années avec la venue sur le marché de VHDL. Depuis sa normalisation en décembre 1987, les grandes compagnies de CAO se sont efforcées de l'intégrer dans leur environnement de conception. Les facultés de description de ce langage et sa souplesse d'utilisation autorisent des simulations temporelles très précises au niveau des portes, après l'obtention et l'analyse de la schématique.

Mais revenons à l'analyse des différents temps de propagation d'un circuit pour expliquer cette approche. Une simplification classique fait apparaître trois types de retards. Ils sont détaillés ci-dessous et présentés dans la figure 2.11.

- Les retards liés aux cellules: ce sont les temps de propagation (notés **tp**) propres aux cellules du circuit. Ces caractéristiques sont fournies par les constructeurs. Six valeurs, obtenues en combinant MIN, MAX, TYP avec HL et LH, sont généralement données pour chaque sortie. Ces valeurs sont attachées aux sorties des portes.

• Les retards liés au layout: ce sont les temps de traversée des connexions. Ils doivent être fournis par un extracteur après analyse de la schématique. Ils sont directement liés à la longueur des fils (notée $\overline{se_i}$) et à la technologie. Ces retards sont attachés aux entrées des portes auxquelles sont connectés ces fils car une sortie est souvent reliée à plusieurs entrées alors qu'une entrée est rarement connectée à plusieurs sorties.

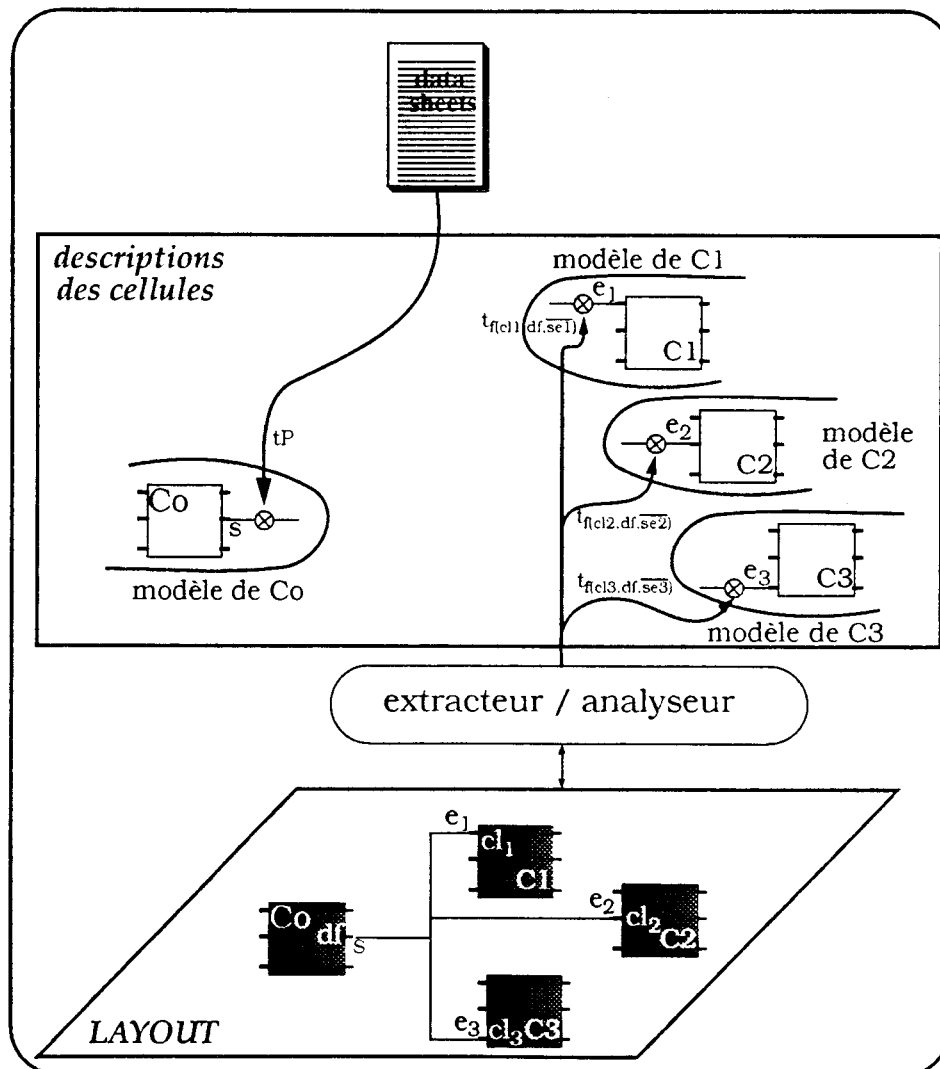


Figure 2.11 Extraction pour la simulation temporelle fine

• Les retards liés au fan-out: ce sont les temps liés aux effets de charge des entrées et sorties de cellules. D'une sortie s à une entrée e , ces temps dépendent de la force de s (notée df pour *driving force*), de la capacité de charge de e (notée cl pour *capacitive load*) mais aussi des autres entrées connectées à s . Les caractéristiques des cellules (df et cl) sont fournies par

les constructeurs tandis qu'un analyseur produit les informations relatives à l'identification des cellules connectées à **s**. Ces retards sont attachés aux entrées des cellules bien qu'ils peuvent aussi l'être aux sorties si l'on considère une capacité de charge moyenne par cellule; dans ce dernier cas, l'analyseur détermine simplement le nombre d'entrée(s) connectée(s) à la sortie **s** sans tenir compte de leurs caractéristiques propres.

2.2.3 L'intégration

L'intégration dont il est ici question est celle des outils de CAO dans un même environnement, que l'on qualifie alors d'intégré. Ces outils, organisés autour d'une base de données, forment ainsi un environnement de seconde génération. Les environnements de la première génération, dont les outils étaient créés au fur et à mesure des besoins, sans véritables liens entre eux, ne peuvent plus faire face à la complexité des circuits actuels.

Un environnement intégré doit:

- ▶ assurer le suivi d'un modèle tout au long du processus de conception. Pour cela, il doit maintenir la correspondance entre les vues graphiques et textuelles ainsi qu'entre les représentations aux différents niveaux d'abstraction. La modification d'une description à un niveau doit entraîner la mise à jour des descriptions aux niveaux supérieurs (approche remontante). La couverture d'un maximum de niveaux ainsi que la gestion des données sont des facteurs essentiels d'un tel environnement.

- ▶ assurer l'enchaînement automatique des différentes étapes ou en faciliter la commande interactive par un outil de dialogue.

a. Le problème

Les impératifs présentés ci-dessus ont pour objectifs la simulation multiniveau d'un circuit de façon efficace (ressources de temps et d'espace utilisées) et transparente (aspect utilisateur). Il est clair que la manière de représenter un circuit est un point clé quant à son évolution au cours du processus. A la fois les représentations internes (celles utilisées par les outils) et surtout externes (celles sous forme de langage de description) doivent être prises en compte pour traiter de cette évolution.

Les premiers langages de description de matériel étaient destinés à la modélisation à un niveau d'abstraction type, bien que pouvant parfois être étendus à d'autres niveaux. L'utilisation de ces langages dans un même

environnement présente des défauts qui ont nécessité une autre solution. Les défauts majeurs sont énumérés ci-dessous:

- Le concepteur doit connaître plusieurs langages et bien souvent plusieurs simulateurs (un par langage), ce qu'il rebute à faire.

- L'harmonisation des vues internes, parfois à des niveaux différents, n'est pas sans poser des problèmes liés à la sémantique des objets manipulés. Il faut au minimum choisir les langages de façon qu'ils soient compatibles avec une unique vue interne ou de manière à ce qu'elles puissent être facilement traduites les unes dans les autres.

- Les représentations externes aux différents niveaux, c'est-à-dire les descriptions sous forme de texte HDL, ainsi que les représentations internes des divers outils requièrent l'usage de traducteurs pour passer d'une représentation à une autre. Là aussi, cela pose de sérieux problèmes, notamment d'efficacité, même lorsque les langages sont correctement choisis. Ainsi, dans un environnement de N outils, le passage d'une représentation i à une représentation j nécessite:

- a) la définition d'au plus $n(n-1)$ traducteurs et l'emploi d'un d'entre-eux, ou
- b) le développement de $2(n-1)$ et l'usage de $j-i$ (si $j>i$) traducteurs.

L'usage de plusieurs langages est donc une solution peu pratique, lourde et coûteuse. C'est pourquoi les environnements actuels utilisent un langage unique.

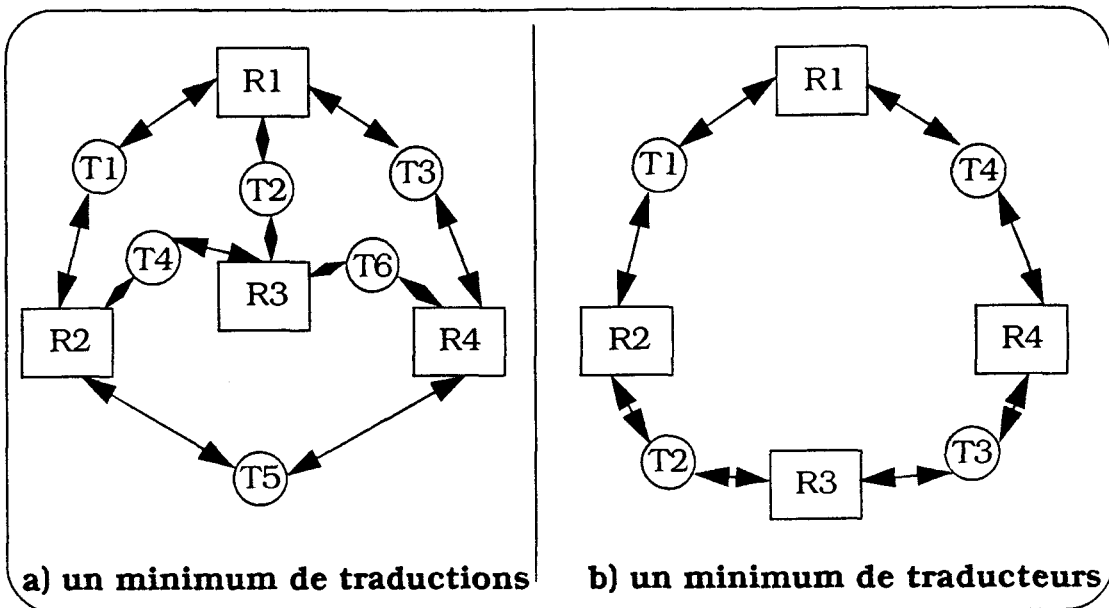


Figure 2.12 Représentations et traducteurs

b. L'intégration des langages

L'utilisation d'un seul langage de description est donc la solution admise pour obtenir un environnement intégré et efficace de CAO. Deux types de langage unique peuvent être utilisés. Il s'agit soit d'un langage qui couvre tous les niveaux d'abstraction soit d'un ensemble de langages définis à partir d'une même syntaxe.

Une famille de langages:

L'exemple le plus significatif de cette approche est sans doute celui du système CASCADE (Circuits And Systems Computer Aided Design and Engineering). Il a été développé au laboratoire ARTEMIS de l'université de Grenoble autour du langage CASCADE qui s'inspire de la philosophie du langage CONLAN (CONsensus LANguage), dont l'origine remonte à la première conférence sur les HDLs tenue en 1973. L'unanimité de la communauté scientifique à reconnaître le besoin d'un HDL couvrant tous les niveaux d'abstraction, a conduit à la formation d'un groupe international de recherche sur ce sujet. L'appellation de langage est relativement trompeuse car CONLAN est aussi un mécanisme de construction de HDLs. Ainsi, tous les langages dérivés de CONLAN ont un noyau syntaxique commun, ce qui permet de résoudre certains des problèmes cités plus haut.

Par ce mécanisme, un nouveau langage est construit à partir d'un

ensemble initial de primitives (objets, règles et concepts) associées à un niveau, cet ensemble pouvant être réduit ou augmenté de nouvelles primitives.

Ainsi, CASCADE est plus une famille de langages qu'un langage à proprement parler. Il couvre les niveaux de CONLAN (du niveau système au niveau de commutation) ainsi que le niveau électrique qui a été ajouté. CASCADE est donc constitué des sous-langages LASSO (système), LASCAR (architecture), CASSANDRE (Transfert de registre), POLO (porte), CASTOR (de commutation) et IMAG (électrique).

L'environnement de CASCADE est constitué d'un simulateur hiérarchique, multiniveau et multimode utilisant une unique structure interne, ce qui est rendu possible grâce au noyau syntaxique commun. Dans le cas d'un modèle multiniveau, l'emploi de fonctions de conversions reste néanmoins nécessaire mais est compensé par l'usage de fonctions spécifiques associées à chaque niveau ([BF85]).

Un langage unique:

L'autre solution consiste à définir un langage qui couvre tous les niveaux d'abstraction et un simulateur unique pour ce langage, quel que soit le niveau de chaque module d'une description. A chaque niveau, on a donc accès à toutes les primitives du langage et aux concepts de tous les niveaux réunis. On peut donc écrire une description comportementale à l'aide notamment des structures de données et des constructions du langage "de tous les niveaux", en particulier des niveaux plus élevés. Les concepts des langages de programmation, habituellement associés au niveau algorithmique auquel l'utilisateur est familier, sont alors disponibles dans toute la hiérarchie d'une description.

Ainsi, la syntaxe d'un tel langage est souvent assez proche des langages de programmation du type Pascal même si la sémantique des opérations est très différente. En effet, un HDL doit permettre de modéliser le comportement d'un circuit ce qui implique des notions de simultanéité d'opérations et une sémantique radicalement différente pour l'affectation par exemple. Ce type de langage offre la notion de généricité permettant d'obtenir des descriptions modulaires. Les niveaux couverts sont tous les niveaux logiques (du niveau système au niveau de commutation). Ces simulateurs sont donc multiniveau mais pas forcément multimode.

Le langage CAP/DSDL (Concurrent Algorithmic Programming/Digital Systems Description Language) [Ram80] est un premier exemple de cette catégorie dont deux simulateurs maintenant largement connus ont été

réalisés. Il s'agit du système SMILE [GEF84] de Siemens d'une part et du système DACAPO [KS*85] de Dosis d'autre part. Le premier utilise un algorithme d'interprétation de réseau de Pétri tandis que le second, dans sa nouvelle version remplace cet algorithme par la génération directe de code exécutable.

Un second exemple est constitué du langage VHDL [VHD87]. Celui-ci ne possède pas réellement de niveau de commutation mais la description à ce niveau est possible [Leu89] bien que tout ou presque doive alors être défini par le concepteur. Par contre, VHDL présente l'avantage de permettre une simulation temporelle fine au niveau porte grâce à l'emploi d'un extracteur/analyseur (voir figure 2.11). La plupart des simulateurs de ce langage utilisent C comme "langage intermédiaire". Les plus grandes compagnies proposent de tels produits; citons par exemple, Intermetrics, Mentor Graphics, Zycad, Vantage Analysis Systems et MCC¹ pour ceux qui traitent le langage dans sa totalité. D'autres développent des logiciels pour leurs propres besoins mais ne les commercialisent pas (c'est le cas d'IBM) ou ne traitent qu'une partie du langage (Cadence Design Systems). Enfin, tous les autres vendeurs intègrent au minimum une passerelle entre VHDL et leur langage de description initial.

c. Les outils

Les outils qui réalisent l'intégration d'un système de CAO peuvent être définis comme des outils de transformation d'un niveau/domaine de description vers un autre. Une telle transformation peut prendre diverses formes mais elle s'effectue toujours d'un niveau intermédiaire entre les spécifications initiales et les masques du circuit vers un niveau plus bas ou vers un autre domaine.

Lorsqu'un outil permet d'obtenir les masques d'un circuit à partir d'une description de haut niveau, il est appelé compilateur de silicium. Dans le cas contraire, c'est-à-dire lorsqu'il ne permet que de changer de domaine (comportemental vers structurel), il s'agit d'un outil de synthèse. On dénombre plusieurs types de synthèse. Il faut notamment distinguer la synthèse comportementale ou synthèse de haut niveau de la synthèse logique. La première consiste à traduire une description comportementale à un haut niveau d'abstraction en une description structurelle à un bas niveau (niveau porte ou inférieur). La seconde est une transformation d'une description au niveau porte jusqu'à obtenir les masques; ce niveau est

1. Microelectronics and Computer Technology Corporation

parfois appelé niveau logique (terme néanmoins impropre), ce qui explique l'appellation de synthèse logique.

Cette partie traite essentiellement des compilateurs de silicium, outils "idéaux" d'un environnement intégré puisque l'enchaînement des étapes se fait de façon automatique. Un compilateur de silicium intègre la connaissance des experts en conception, notamment en synthèse, et met la conception de circuits à la portée des utilisateurs non experts en intégration. Ainsi, la majeure partie des tâches de bas niveau (placement et routage) sont transparentes pour l'utilisateur, même s'il doit parfois modifier quelque peu la schématique (parties critiques). De plus, les étapes intermédiaires de vérification peuvent être éliminées car les phases de génération internes et automatiques sont à-priori sans erreur.

Un compilateur de silicium agit comme un compilateur au sens habituel. Il offre une passerelle entre une forme de haut niveau (langage de programmation ou HDL au niveau transfert de registre ou plus élevé) et une forme de bas niveau (code machine ou masques d'un circuit). De nombreuses étapes se trouvent ainsi encapsulées dans un compilateur de silicium, ce qui permet de réduire le processus de conception. Ce celui-ci est présenté dans la figure 2.13. Les masques du circuit sont obtenus à partir d'une description écrite au niveau transfert de registre ou au niveau architecture.

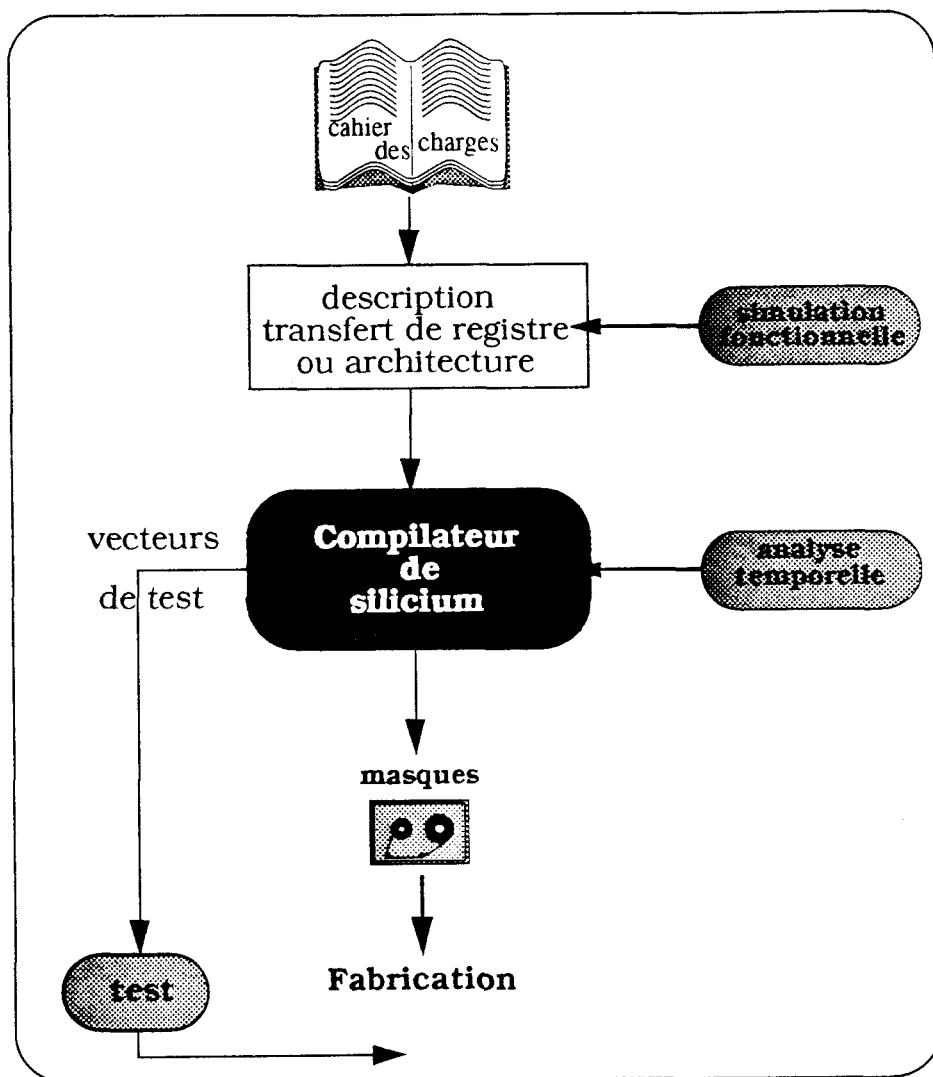


Figure 2.13 Compilateurs de silicium pour un processus minimal

Après la simulation fonctionnelle, nécessaire à la vérification du modèle, le compilateur de silicium génère les masques du circuit, les vecteurs de test et permet une analyse temporelle fine du circuit. L'intervention de l'utilisateur est limitée au maximum pour diminuer le temps de conception mais reste nécessaire pour certaines décisions critiques.

On peut aussi faire une similitude en comparant les performances. Un compilateur est plus sûr que l'humain et permet un gain de temps considérable en produisant toutefois un résultat sensiblement moins bon. D'un côté le code machine serait plus efficace s'il était généré par un assembleur et d'un autre côté la densité obtenue sur le silicium est souvent moins bonne (la surface nécessaire serait moins importante si la

schématique était produite "à la main"). Ainsi, en 1985 la densité d'un circuit obtenu à l'aide d'un compilateur de silicium est inférieure de dix pour cent à celle d'un circuit produit "manuellement" mais reste meilleure de vingt-cinq à quarante pour cent par rapport à celle d'un circuit construit manuellement à l'aide de *standard cells* ou de *gate array* [Bon85].

La tendance en matière de compilation de silicium est à l'abstraction, c'est-à-dire que les outils tendent de plus en plus à accepter des descriptions écrites à des niveaux élevés pour faire face à l'accroissement de la complexité des circuits. Si la synthèse logique est une technique maîtrisée depuis plusieurs années, il n'en est pas de même en ce qui concerne la synthèse de haut niveau. C'est une tâche très complexe qui introduit plusieurs problèmes NP-complets [CW85] pour lesquels il n'y a pas encore d'approche standard.

De nombreux produits sont apparus sur le marché depuis le premier compilateur de silicium FIRST [DRB82], dédié aux environnements de télécommunication et à la technologie CMOS. Si ces produits sont très divers par leur approche, on peut néanmoins les classer en deux grandes catégories selon qu'ils sont articulés autour de l'axe structurel ou comportemental du diagramme Y de Gajski-Kuhn (voir figure 2.9).

Un exemple bien connu à suivre l'approche structurelle, qui dénote une orientation matérielle très forte, est le système *Genesil* de Silicon Compilers. Les premières versions ont permis de réaliser des circuits complexes; par exemple le DEC 32-bits Micro-VAX I a été conçu en sept mois par une équipe de trois personnes au lieu de vingt personnes pendant deux années avec le processus classique [Joh84]. C'est le premier environnement complet destiné à la conception de systèmes et non plus à celle de "petits" circuits. Il permet la définition d'un circuit, sa simulation fonctionnelle, la synthèse et l'analyse temporelle à partir d'une description au niveau architecture. Les différentes étapes sont indépendantes de la technologie et de la fonderie jusqu'à la génération des masques.

L'approche comportementale est suivie par MetaLogic et son système *MetaSyn*. Celui-ci est écrit en Lisp et très orienté logiciel. C'est un descendant du système MacPitts [SSG82] dont il reprend l'avantage principal, c'est-à-dire l'analyse et la prévision de performances. Ainsi, après l'obtention de la schématique, l'utilisateur peut revenir à la description initiale, la modifier et obtenir automatiquement une nouvelle schématique. Même si le temps de conception s'en trouve légèrement accru, cela permet d'atteindre les performances spécifiées initialement.

2.2.4 La communication

A mesure que les machines à construire sont de plus en plus constituées de l'assemblage de processeurs, le temps de conception a une influence considérable sur le coût d'un tel système. De plus la diversité des environnements rend difficile l'échange de modèles de circuits, ce qui permettrait un gain de temps appréciable. C'est l'absence de standard qui fait défaut dans ce domaine. Il y a quelques années, deux langages standards sont apparus pour pallier ce défaut.

a. VHDL

Ce langage de description est un standard depuis 1987. En conséquence, un modèle écrit en VHDL peut être utilisé dans tout environnement le supportant. VHDL permet donc de communiquer des modèles de circuits sous forme de description. Un tel modèle peut alors servir dans la description d'un autre circuit, dont il est une partie. Malheureusement, la qualité de standard d'un HDL n'est qu'une condition nécessaire à la communication de modèles. Tout comme des programmes écrits par des équipes différentes dans un langage de programmation standard (par exemple C, norme ANSI) ne peuvent fonctionner ensemble que sous certaines conditions (emploi des mêmes types de données ou définition d'interfaces), l'usage de modèles écrits par différents concepteurs posent des problèmes de compatibilité. Ceux-ci sont bien plus importants dans le domaine des HDLs car ils se conjuguent aux considérations de domaine de description, de niveau de détail et de type de modèle temporel utilisés. La communauté scientifique travaille depuis plusieurs années déjà à la définition de conventions pour l'écriture de descriptions compatibles. Ce problème et l'état actuel des propositions sont présentés plus en détail dans [Dub90, BD*90]. Une solution pour l'obtention de modèles *consistants*, la consistance constituant avec l'*interopérabilité* les deux caractéristiques nécessaires à la compatibilité, est présentée dans [Dub91].

b. EDIF

EDIF (Exchange Design Interchange Format) est un format industriel standard qui permet de communiquer les modèles sous forme de netlist. Ce format est donc obtenu après mise au point et synthèse du circuit. Les effets

de l'explosion du nombre de systèmes de conception et du nombre de stations de travail ont été anéantis avec la venue d'EDIF sur le marché. Dans la perspective des compilateurs de silicium comme ultimes outils de conception, ce langage permet de répondre au manque actuel de méthode normalisée.

2.3 Les simulateurs

Cette partie est consacrée plus en détail aux différentes manières de réaliser un simulateur. Ceci permet de présenter leurs avantages et inconvénients et de justifier le besoin d'une solution nouvelle, qui sera expliquée dans les chapitres suivants.

La simulation est une étape importante de la CAO et reste nécessaire même dans le cas des compilateurs de silicium. Il est en effet impératif de vérifier la correction d'une description avant de la fournir à un tel outil afin d'éviter qu'il ne produise un circuit "mutant". Plus les erreurs de conception sont découvertes tôt et moins elles ont une influence importante sur le coût de production d'un circuit. Cette étape de modélisation/simulation ne peut être évitée car il n'existe pas à l'heure actuelle de logiciel qui traduise les spécifications du cahier des charges, données en langage naturel, en une description écrite en HDL.

Il existe essentiellement quatre manières d'effectuer la simulation. Celles-ci sont obtenues en combinant le type de réalisation de l'exécution (interprétation ou compilation) et son mode (direct ou indirect). Les caractéristiques de ces solutions sont présentées ci-dessous. Elles sont comparées selon les critères de performance, de portabilité, d'espace mémoire nécessaire et de convivialité. Les solutions extrêmes seront traitées plus en détail car elles offrent les avantages les plus marqués.

2.3.1 L'interprétation directe

Dans ce cas, une description est interprétée, c'est-à-dire qu'elle est analysée en même temps que se poursuit l'exécution (et réciproquement). Pour une simulation et un stimuli d'entrée donnés, il se peut que certaines parties du circuit ne soient pas vérifiées; par exemple, la branche alors ou sinon d'une condition peut ne pas être évaluée tout au long d'une même exécution.

Ce type de simulation est lent (de cinq à dix instructions par seconde

sur une machine classique) car l'analyse doit être effectuée à chaque fois. De plus, les erreurs ne sont découvertes qu'une à une et provoquent l'arrêt de l'exécution ce qui ralentit d'autant plus la mise au point de la description.

D'un autre côté, ce genre de solution permet la modification interactive du modèle et la poursuite immédiate de l'exécution où elle s'était arrêtée. L'avantage principal de cette méthode réside donc dans la qualité de l'interface utilisateur. Celui-ci permet le déroulement de l'exécution pas à pas et donne accès à chaque instant à la description, aux représentations graphiques des signaux à l'aide de diagrammes et à la visualisation des chemins actifs de l'architecture simulée. De plus, la représentation des résultats sous forme numérique tout au long de la simulation, permet de poursuivre la simulation à tout moment après avoir éventuellement modifié la valeur de certains signaux.

L'espace mémoire nécessaire à ce type de simulation est important, dû essentiellement aux diverses représentations qui doivent être gérées au cours de l'exécution.

2.3.2 L'interprétation indirecte

Celle-ci diffère assez peu de l'interprétation directe. De ce fait, ces deux méthodes sont souvent nommées par le seul terme d'interprétation. La caractéristique principale qui les distingue concerne l'analyse d'une description. Alors qu'elle est effectuée au fur et à mesure dans le cas de l'interprétation directe, elle constitue une étape propédeutique à l'interprétation indirecte. Une description est ainsi analysée syntaxiquement et sémantiquement avant que ne soit produit un code intermédiaire sur lequel s'exécute l'algorithme de simulation.

Ceci a des répercussions sur les critères énoncés plus haut, de comparaisons entre les différentes stratégies de réalisation des simulateurs. Les erreurs d'une description sont donc découvertes plus rapidement, comme dans un compilateur classique, et l'exécution est plus efficace car elle utilise une représentation intermédiaire "précompilée" des variables et de la description, adaptée à l'algorithme de simulation. L'amélioration des performances est donc l'avantage majeur de cette solution par rapport à la précédente, auquel il faut toutefois ajouter la réduction sensible de l'espace mémoire nécessaire à la simulation.

Ces avantages sont accompagnés d'une restriction sur les fonctions de l'interface proposé à l'utilisateur. Pour celui-ci, la représentation interne est inaccessible. En conséquence, toute modification de la description implique

une nouvelle analyse et la production d'un nouveau code intermédiaire. C'est le processus dans sa totalité qui doit être recommencé, ce qui déprécie quelque peu l'avantage de cette méthode sur la précédente.

2.3.3 La compilation directe

Cette solution consiste à traduire directement la description d'un circuit, donnée sous forme de HDL, en code machine. Un simulateur est donc destiné à effectuer la simulation d'une *machine cible* (celle que l'on veut concevoir) sur une *machine hôte* donnée (celle sur laquelle s'exécutera la simulation).

Suivant qu'un simulateur obtenu de cette manière incorpore ou non des paramètres d'exécution et suivant leur nature, on distingue quatre types de simulateurs compilés, présentés par la figure 2.14.

Un simulateur peut être dédié à une architecture (lorsque la description initiale ne décrit que son fonctionnement) ou destiné à simuler l'exécution d'un programme sur cette architecture. Il s'agit dans ce cas d'un simulateur *dédié à un programme*. Un simulateur peut encore être plus spécifique lorsqu'il simule l'exécution d'un programme sur une architecture avec un jeu de données particulier. Les données sont fournies dans ce cas à la compilation et non à l'exécution. On parle alors d'un simulateur *dédié à une exécution*. Le cas le plus extrême a lieu lorsque les paramètres d'une simulation fournis par l'utilisateur sont inclus lors de la compilation. Il s'agit alors d'un simulateur *complètement spécifié*.

Cette approche de l'intégration des paramètres d'exécution lors de la compilation vise à accroître l'efficacité mais se fait au détriment de la souplesse. Les deux derniers cas sont d'ailleurs d'un intérêt très restreint. En effet, le concepteur doit avoir recours à de nombreuses simulations pour vérifier la fonctionnalité d'un circuit. Pour chacune d'elles, un nouveau simulateur doit être généré, ce qui ajoute à la durée de chaque simulation celle de la préparation du simulateur (génération du modèle HLL et sa compilation).

L'avantage majeur de la compilation directe réside dans l'efficacité du simulateur obtenu, rendue possible grâce à la prise en compte des caractéristiques de l'architecture hôte. Un simulateur ainsi produit est donc rapide et peu gourmand en espace mémoire. Par contre, l'objectif de cette approche (l'efficacité) se marie mal avec la réalisation d'un interface utilisateur évolué. De plus, les simulateurs obtenus ne sont pas portables, pas plus que ne l'est l'outil qui les produit.

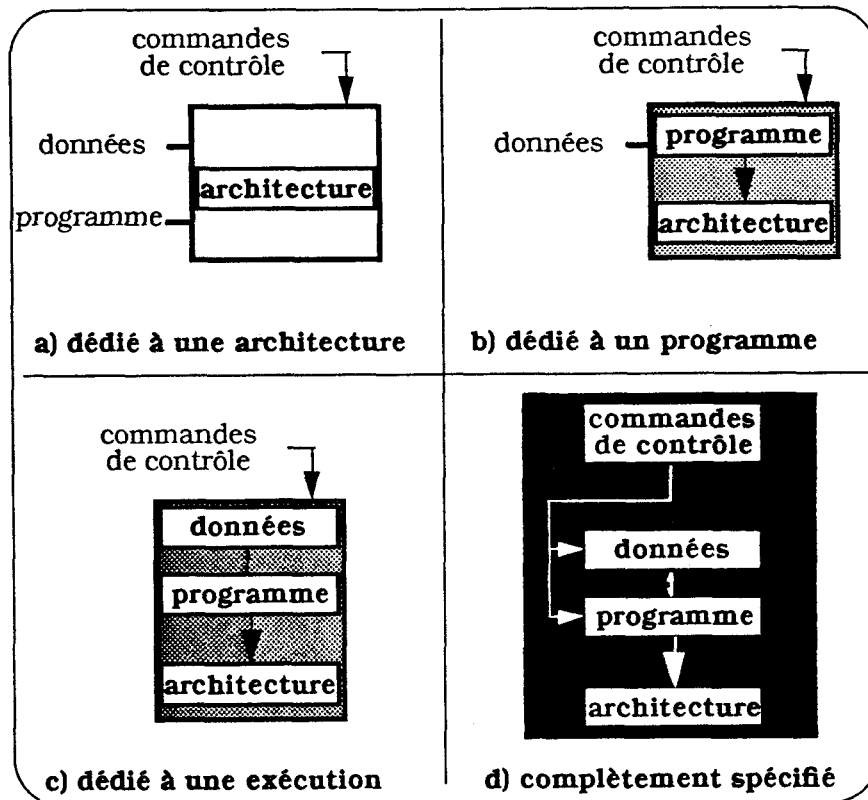


Figure 2.14 Les simulateurs compilés

2.3.4 La compilation indirecte

L'objectif de ce type de solution est de remédier aux inconvénients de la compilation directe. Une description est traduite dans un langage de programmation (HLL) tel que Pascal, Ada ou C par exemple. Le simulateur est donc obtenu sous forme de source et non plus sous forme de code exécutable. Le terme de *modèle* s'applique aussi au source du simulateur alors qu'une description est toujours spécifiée à l'aide d'un HDL. Le terme de simulateur est peut être abusif dans la mesure où l'exécution du programme généré est en fait souvent gérée par un programme maître de contrôle. Tout comme dans le cas précédent, un simulateur généré de cette façon peut être dédié à une architecture, à un programme, à une exécution ou être complètement spécifié.

Un simulateur obtenu de la sorte peut être transporté puis compilé et exécuté sur une machine hôte quelconque. L'idéal est que le générateur produise des simulateurs adaptables qui prennent en compte les

caractéristiques des machines sur lesquelles ils seront exécutés (taille d'un mot par exemple).

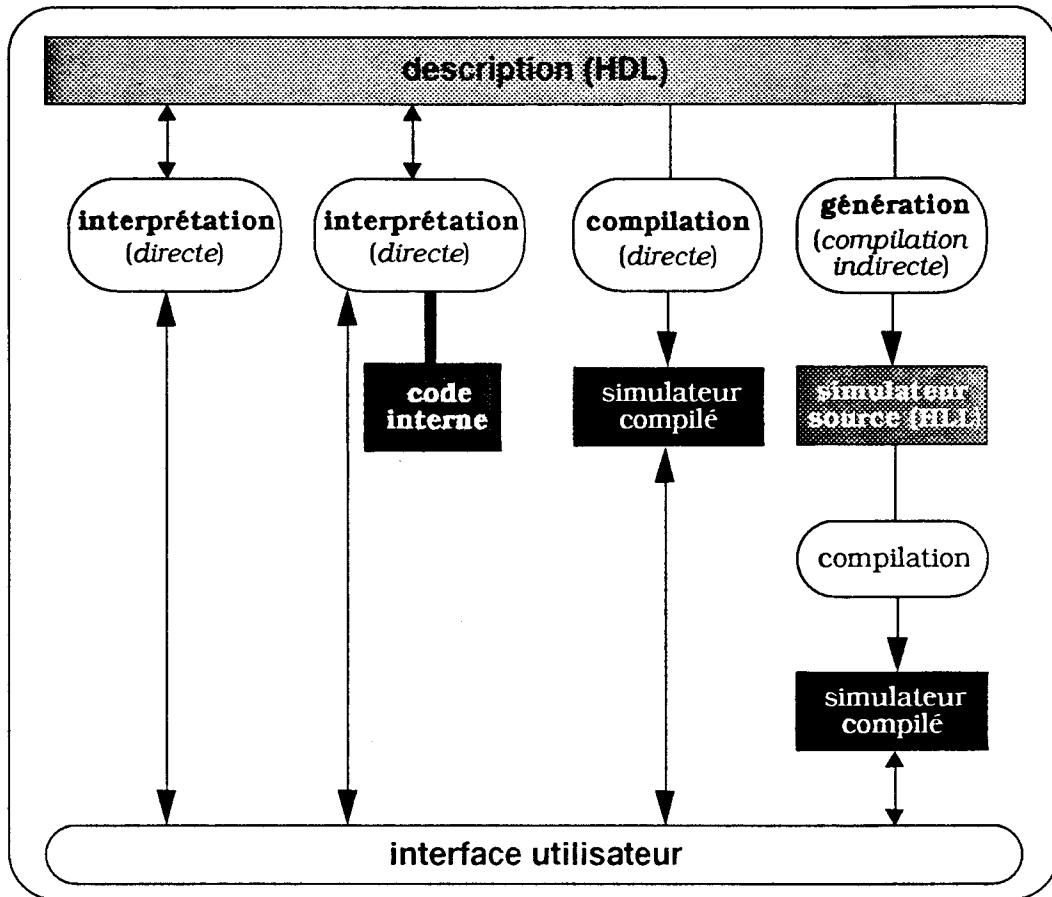


Figure 2.15 Méthodes de simulation

De plus, la mise au point d'une description est grandement facilitée par l'emploi d'un débogueur symbolique, outil disponible sur toute station de travail digne de ce nom. Un tel outil offre un contrôle total sur l'exécution du simulateur sans requérir l'usage de types de données spécialement adaptés à un interface particulier, ce qui constitue généralement un goulot d'étranglement pour l'efficacité de la simulation.

L'inconvénient principal de la génération réside dans la taille des simulateurs obtenus, c'est-à-dire qu'ils ne sont pas compilables en un temps acceptable pour l'utilisateur. La décomposition d'un module en unités de description permet de réduire ce défaut même si la taille d'un modèle reste plus importante que celle obtenue par les autres solutions. Un simulateur est alors constitué de sous-programmes compilés séparément ce qui n'est pas sans offrir d'autres avantages tels que par exemple la réutilisabilité

grâce à l'emploi de bibliothèques.

L'efficacité des simulateurs générés n'est que très légèrement inférieure à celle des simulateurs compilés. La solution idéale en la matière serait l'écriture du modèle dans un langage de programmation par des concepteurs-programmeurs mais elle nécessite beaucoup trop de temps pour être envisageable, surtout pour des circuits complexes.

A la lumière des différentes solutions présentées ci-dessus, il apparaît que la qualité de l'interface homme/outil et le souci des performances sont deux objectifs difficilement conciliables. L'emploi d'un débogueur symbolique permet certes de mettre facilement au point une description mais n'atteint pas le niveau de convivialité qu'offre un environnement interactif d'interprétation. Il est possible de parvenir à une telle qualité d'interface avec un simulateur compilé, mais les performances s'en trouvent alors diminuées et cela nécessite des machines hôtes rapides. De plus, l'interface proposé dans ce cas est dépendant d'un type de machine hôte donné.

L'interprétation et la compilation (directe ou indirecte) sont deux solutions complémentaires, nécessaires pour les stations de travail courantes. Dans le cas de machines spécialisées ou tout simplement plus rapides (station de travail SPARC), la compilation peut être envisagée comme unique outil de simulation logique dans la mesure où l'intégration des fonctions d'un interface évolué permet de conserver une vitesse de simulation suffisante.

2.4 Conclusion

La présentation des étapes de la CAO et de la tendance actuelle en cette matière a montré l'importance de la modélisation-simulation. Ces étapes, étroitement liées dans l'approche incrémentale, restent nécessaires au moins jusqu'au niveau porte dans la mesure où les outils de synthèse de haut niveau n'en sont qu'à leur premiers pas.

L'amélioration des outils de simulation est donc nécessaire pour faire face à l'accroissement de complexité des circuits. Dans cette optique, les différentes manières de réaliser un simulateur ont été présentées afin d'en dégager les avantages et inconvénients.

La contribution de cette Thèse, qui sera détaillée dans les chapitres suivants, consistera donc à trouver un remède aux inconvénients d'une de

ces solutions tout en conservant ses avantages.

Chapitre 3

Vers des simulateurs efficaces

L'analyse des différentes solutions de simulation effectuée dans la partie précédente a montré que l'interprétation offre un interface utilisateur très convivial au détriment des performances. D'autre part, les divers outils de simulation utilisant une méthode de compilation apportent une réponse à ce problème mais présentent encore des inconvénients. Ceux-ci concernent la taille des simulateurs (traduction) ou la difficulté à les échanger entre des équipes ou compagnies diverses (compilation).

Les solutions proposées jusque-là par des laboratoires ou compagnies et décrites dans la suite de ce chapitre, présentent elles aussi des défauts liés soit à leur manque d'efficacité soit à leur coût. Il y a donc besoin d'une solution nouvelle.

C'est la traduction qui a été choisie comme technique de simulation et ce chapitre présente, outre les objectifs du travail et les choix généraux d'implémentation relatifs à cette technique, la synthèse de parties contrôles. Cette transformation de la description initiale est une alternative au schéma trop classique de traduction d'un algorithme en un autre.

3.1 Les solutions actuelles

Un certain nombre de solutions à ces problèmes, proposées et appliquées dans des environnements divers, sont présentées dans cette partie. Celles-ci se divisent en deux catégories.

3.1.1 Les solutions matérielles

Il est bien sûr possible d'utiliser une machine parallèle, hautement spécialisée, dédiée uniquement à la simulation. Depuis la première machine spécialisée créée à cet effet [McK69] pour les besoins de Boeing Aircraft, de nombreuses machines parallèles ont été réalisées, généralement par les plus grandes compagnies commerciales. Citons par exemple, celles de chez IBM¹ YSE [Den82] et EVE [Dun84], HAL [Sas83] de chez NEC², Realfast Simulation Machine de VALID Logic ou encore Zycad Logic Evaluator de Zycad.

Ces machines multiprocesseurs cablent les algorithmes de simulation et permettent d'atteindre, dans le meilleur des cas, un facteur d'amélioration

1. International Business Machines

2. Nippon Electric Corporation

de la vitesse de simulation (noté FA par la suite) d'un million par rapport à une simulation effectuée sur une machine un MIPS¹. Les machines spécialisées plus récentes et plus simples à mettre en œuvre utilisent les transputers comme support de réalisation d'une architecture dédiée à la simulation [RSV88].

Un FA aussi extraordinaire est limité à des simulations très spécifiques, ne pouvant être effectuées bien souvent qu'à un unique (bas) niveau d'abstraction. Il s'agit dans la plupart des cas du niveau porte, parfois du niveau de commutation [DB85], car ce type de simulation est lent et permet une meilleure amélioration. Dans certains cas, la simulation peut être effectuée à plusieurs niveaux; il s'agit alors des niveaux portes et RTL. De plus un tel FA ne peut être obtenu que pour de longues exécutions (de l'ordre de un million de cycles) de manière à diminuer la part relative des entrées/sorties. Les gains de temps sont certes considérables mais les machines, qui sont en fait des simulateurs matériels, ne peuvent être utilisées pour d'autres tâches. Enfin, une telle solution nécessite un investissement préalable dont le coût limite de façon radicale l'éventail de ses utilisateurs potentiels. Il y a grand besoin d'une solution efficace dédiée aux environnements de travail de petite importance (PCs et stations de travail).

3.1.2 Les solutions logicielles

Une gamme de solutions moins coûteuses est constituée des méthodes et algorithmes logiciels. Si de telles solutions présentent l'avantage d'être souples et utilisables sur une machine traditionnelle, elles offrent une réduction beaucoup moins importante du temps de simulation. Ces solutions utilisent toutes, à des degrés divers, le mécanisme d'abstraction pour diminuer le temps de simulation.

En dehors des solutions consistant à écrire une description à un niveau d'abstraction élevé, insuffisantes tant que les compilateurs de silicium n'incluent pas d'outils de synthèse de haut niveau, une première technique consiste à appliquer un algorithme de simulation utilisant la notion de processus, plus abstraite que celle d'événements. Ceci implique des contraintes sur la manière de modéliser un circuit et permet d'obtenir une accélération très relative [RA87].

Une seconde méthode consiste à transformer, de façon transparente pour l'utilisateur, la description initiale afin d'utiliser des types de données plus abstraits lors de l'exécution et donc plus adaptés à la simulation rapide.

1. Million d'Instructions Par Seconde

Cette transformation peut aussi produire un programme de simulation en langage de haut niveau [KW86]. Mais cette méthode de génération se heurte à la taille trop importante des simulateurs produits. [BS*89] est un exemple typique car il est caractérisé par un FA variant de dix à cent mais, du fait même qu'il n'utilise pas de procédure, la taille d'un modèle peut croître de façon dramatique, notamment dans le cas de la simulation d'architectures parallèles. Les exemples donnés sont d'ailleurs d'une taille limitée.

Un autre moyen d'obtenir un FA consiste à utiliser des marqueurs. Un algorithme de simulation manipule alors ces marqueurs plutôt que d'interpréter des valeurs logiques ([AA86]). Le temps de simulation devient dans ce cas proportionnel au nombre de changements d'états du réseau (le circuit est vu comme un réseau de blocs fonctionnels traversés par des marqueurs) et non plus à la taille du circuit. Le FA obtenu peut être important par rapport à une simulation dirigée par événements (FA de trois à huit allant jusqu'à vingt dans les meilleurs cas). Cela requiert une modélisation radicalement différente de celle habituellement employée des concepteurs. A ce titre, une méthode de définition de telles descriptions est donnée dans [HA*89], qui pourrait être utilisée par un outil de génération automatique.

3.2 Objectif du travail

L'objectif de cette Thèse est de proposer une solution qui réduise les défauts présentés ci-dessus tout en conservant les avantages des solutions basées sur la compilation. Ainsi, le but est de proposer une méthode permettant d'obtenir des simulateurs **efficaces, compacts, portables** et **souples**.

L'*efficacité* et la *compacité* des simulateurs devront permettre la simulation logique de circuits complexes sur une station de travail traditionnelle (ayant une vitesse de un à cinq MIPS).

La *portabilité* vise à pouvoir communiquer des circuits par l'intermédiaire de leur(s) simulateur(s) plutôt que par celui de leur(s) description(s). Ceci permet l'échange de modèles entre des environnements utilisant des HDLs différents, ce qui est d'un intérêt croissant étant donnée l'expansion du nombre des systèmes de CAO.

La *souplesse* s'applique à la fois aux simulateurs et à leur générateur. Pour un simulateur, la souplesse signifie généralité et adaptabilité à l'architecture hôte. Dans le cas du générateur, il s'agit de l'adaptabilité à l'architecture cible, de modularité et de paramétrisation.

3.3 Choix généraux

Afin d'atteindre les objectifs fixés, un certain nombre de choix ont été effectués dès le début du projet qui n'ont pas été remis en cause par la suite. Des modifications ont été introduites au fur et à mesure des expériences. Ce sont les choix initiaux qui sont présentés dans cette section. Les modifications apportées en cours d'implémentation relèvent surtout de l'optimisation et seront détaillées dans la section 4.3.

La contribution des différents choix à chacune des caractéristiques désirées du/des simulateur(s) est donnée ci-après. Ces choix définissent l'organisation du système créé, présentée par la figure 3.1. Celle-ci ne fait pas apparaître la relation avec l'environnement initial. Les section et figure 4.1 en font l'objet.

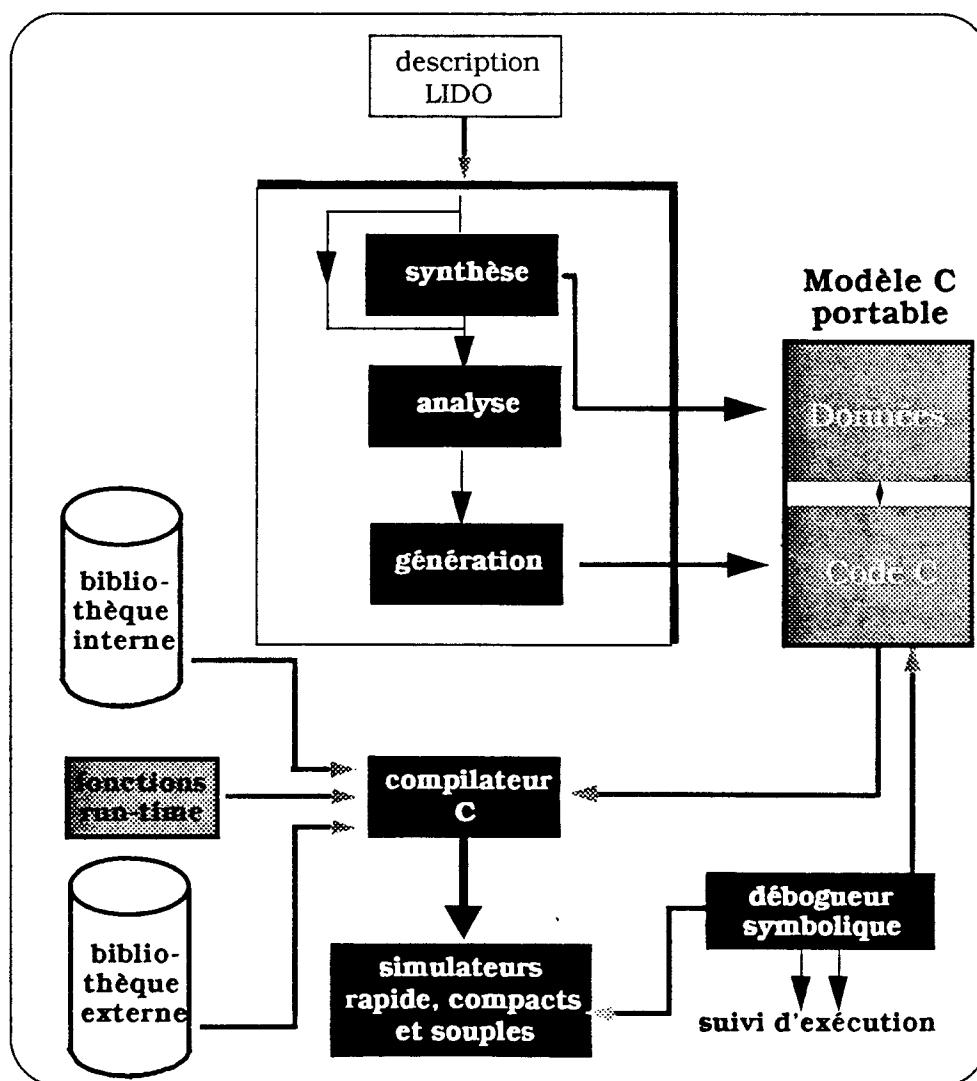


Figure 3.1 Structure du générateur

La compilation indirecte est la méthode de simulation choisie. Comme il a été énoncé au paragraphe 2.3.4, la génération contribue à obtenir des simulateurs efficaces. De plus, la production automatique de programmes de simulation en langage de haut niveau est compatible avec les objectifs de souplesse et de portabilité, pourvu que le langage cible soit convenablement choisi. Des solutions devront être apportées afin de résoudre l'inconvénient majeur de ce type de solution, à savoir la taille trop importante des simulateurs ainsi générés.

Le langage C a été choisi comme langage cible. Les programmes de simulation (simulateurs) utilisent le C norme ANSI et sont de ce fait portables. Ce langage est de plus, largement répandu et toute station de travail digne de cette appellation dispose d'un compilateur C. C est aussi reconnu pour sa faculté à exprimer des opérations de façon compacte et surtout pour son efficacité. A ce titre, il est utilisé par la plupart des logiciels de ce type [Lem88, MM*86, SG86, Tho86, VUM88]. Un exemple significatif est celui du simulateur d'Intermetrics dont les descriptions en VHDL étaient initialement traduites en ADA et le sont maintenant en C, dans un souci d'efficacité. Enfin, outre les types de données abstraits, C dispose aussi de pointeurs, d'adresses et d'opérateurs spécifiques au matériel; il s'agit des opérateurs de décalage, des opérateurs binaires (qui manipulent des bits) et de l'opérateur de complémentation.

L'usage de bibliothèques pour l'organisation des descriptions et des modèles HLL est un choix "structurel" important quant à la limitation de la taille des simulateurs. Une description doit être divisée en plusieurs unités (appelées unités de description, notées UD) au moment de la création de la description initiale du circuit par le concepteur. Les UD sont compilées séparément en respectant la sémantique des inclusions. Le modèle C d'une UD est un sous-programme, archivé dans une bibliothèque. Le modèle C d'un circuit est alors constitué des appels de ces sous-modèles. Cette solution permet d'éviter l'expansion du code C, en ayant au plus une fois le code C d'une UD dans un simulateur et dans l'environnement. De plus, les UD peuvent être réutilisées, ce qui simplifie la tâche du concepteur et permet de réduire le temps de production des modèles C utilisés par la suite.

L'adaptation modulable du simulateur d'une machine cible à une machine hôte permet d'améliorer les performances. Une des caractéristiques d'une machine est la taille de ses bus d'adresses et de données. Celle du circuit à simuler peut être différente de celle de la machine sur laquelle a lieu la simulation et ne pas en tenir compte conduirait à une

perte de performance. Il est évident qu'utiliser une procédure pouvant traiter des paramètres ayant jusqu'à 64 bits pour effectuer une opération d'addition avec des opérandes de 32 bits n'est pas optimal. Ainsi, l'objectif est d'utiliser dans chaque simulateur des structures de données adaptées aux caractéristiques des deux machines. Avant d'expliquer en quoi cela consiste, il convient d'énoncer deux alternatives qui en découlent.

1) Les caractéristiques de la machine hôte ne sont pas connues au moment de la génération. Dans ce cas, si l'on veut produire le modèle C sur une machine, l'échanger pour le compiler et l'exécuter sur une autre machine en tenant compte de ses caractéristiques, il est nécessaire que le code C inclue la sélection des structures de données et des procédures les mieux adaptées à la simulation sur cette machine hôte.

Cette solution conduit à l'expansion de la taille des simulateurs et à la réduction de la vitesse d'exécution. Elle ne va pas dans le sens de la philosophie adoptée et décrite plus loin.

2) L'ensemble des machines hôtes est réduit aux machines 32 bits de manière à éviter les inconvénients de l'alternative précédente. On obtient ainsi un simulateur spécifique à un type de machine hôte. Ceci permet de sélectionner les structures de données (notées SDs) et procédures adéquates de façon statique au moment de la génération.

C'est la seconde solution qui a été finalement choisie après des essais d'implémentation de simulateurs plus généraux (voir chapitre 4). Les simulateurs sont donc destinés aux machines 32 bits mais peuvent être compilés et exécutés sur des machines de mots plus longs, la vitesse de simulation pouvant alors ne plus être optimale. L'ensemble des fonctions disponibles à l'exécution travaillant sur des mots de 32 bits, il n'est pas possible de procéder à une simulation sur une machine 8 ou 16 bits par exemple. Cela n'est pas une restriction importante, car les machines 32 bits sont les plus répandues.

Il s'agit donc d'établir la correspondance entre la machine cible et une architecture 32 bits. Ainsi, un mot est divisé en multiples de 32 et les opérations sont exécutées, lors de la simulation, sur des objets de 32 bits à l'aide de procédures spécifiques et optimisées. Ca n'est donc pas vraiment la machine cible qui est simulée mais en quelque sorte une machine "virtuelle" équivalente (voir figure 3.2).

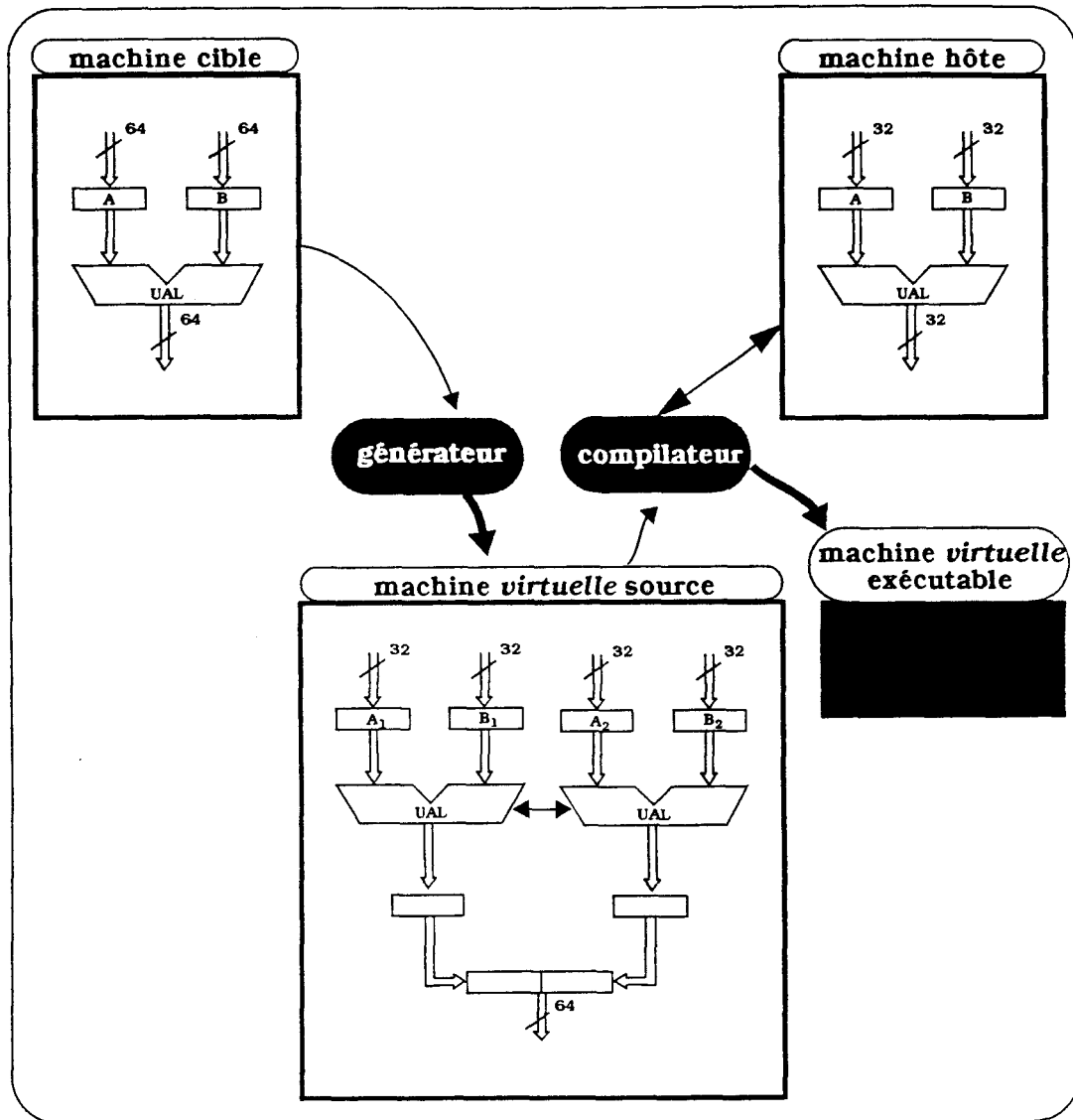


Figure 3.2 Adaptation de la machine cible à la machine hôte

La philosophie de génération est de **ne jamais remettre à l'exécution ce que l'on peut faire lors de la production** du code C. Un simulateur est destiné à être exécuté de nombreuses fois alors que la génération n'a lieu qu'une seule fois. C'est donc le temps de simulation qu'il faut réduire au maximum, même si cela augmente le temps d'obtention du code exécutable (génération plus compilation). Les expériences ont d'ailleurs montré que celui-ci restait dans des proportions très acceptables.

Enfin, "intelligence" et souplesse doivent être des qualités du générateur. Ainsi, il doit être capable de prendre des décisions (il détermine par exemple si les données seront compactées ou non en fonction des

caractéristiques de la machine cible) et doit permettre de produire des modèles dédiés à un programme ou simplement dédiés à une architecture. De plus, la description est analysée afin de sélectionner sur quels types de données seront exécutées les opérations et à l'aide de quelles primitives, choisies dans le jeu des fonctions du run-time.

La contribution majeure de cette Thèse dans le domaine de la génération, bien que relevant d'une certaine philosophie, n'est pas présentée dans cette partie car elle concerne plus des algorithmes que des choix (voir section suivante).

Les premiers résultats de l'implémentation des choix présentés se sont avérés prometteurs, c'est à dire comparables (sinon meilleurs) en terme d'efficacité aux résultats obtenus et publiés par les laboratoires et groupes commerciaux. Les expériences ont toutefois révélé la persistance d'un défaut; il s'agit de la taille des simulateurs qui restait trop importante pour espérer pouvoir procéder à la simulation des architectures complexes actuelles (Motorola 68xxx, Intel 80x86,...). Une réponse à ce problème devait être apportée.

3.4 Présentation théorique de la solution

Les caractéristiques présentées jusqu'à maintenant sont relativement communes à de nombreux systèmes de simulation développés ailleurs. La solution présentée ici constitue l'essence de la contribution de cette Thèse en matière de génération et en fait son originalité.

3.4.1 Naissance de l'idée

L'analyse des premières expériences a montré deux choses. D'une part, la partie contrôle des machines simulées représente une part importante (voir figure 5.3 et paragraphe 5.4.2.a) de la consommation d'espace et de temps CPU. Ceci est d'ailleurs le reflet de la réalité (voir figure 3.3). D'autre part ce sont les versions microprogrammées de ces machines qui présentaient les simulateurs les plus compactes et les plus rapides (voir figure 5.4 et paragraphe 5.4.2.b).

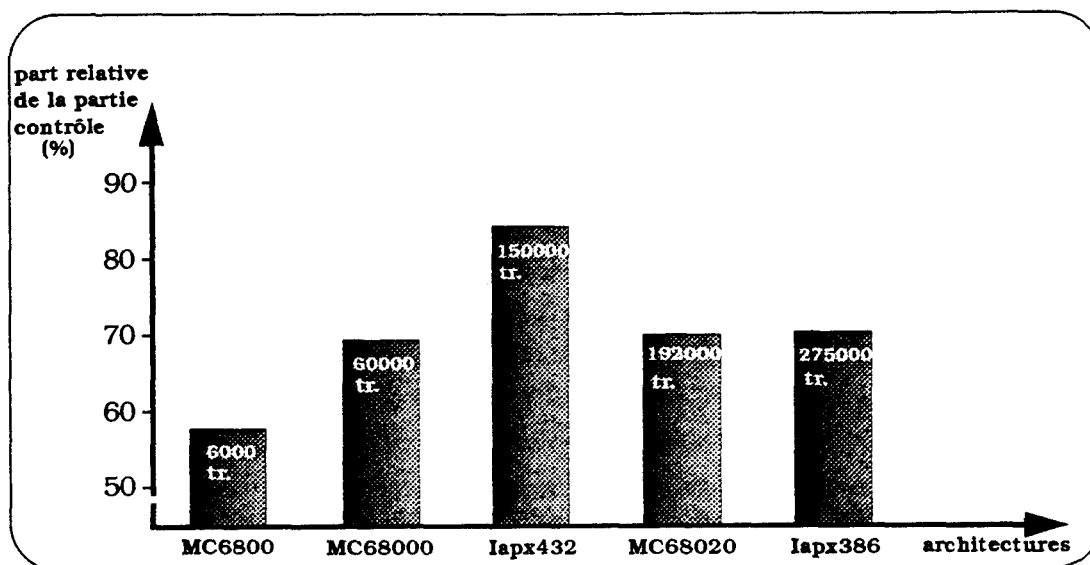


Figure 3.3 Importance de la partie contrôle

Il apparaissait donc intéressant de s'attacher à réduire la part relative des parties contrôles, tant en espace qu'en temps consommé, plutôt que de vouloir réduire celle des parties opératives.

Le défaut majeur des logiciels de génération est de reposer sur un schéma trop classique de traduction d'un algorithme en un autre. Ainsi, le code généré est bien souvent le reflet d'algorithmes d'interprétation et les outils de compilation indirecte n'utilisent pas le potentiel de transformations intermédiaires de cette méthode.

De plus, la partie contrôle est généralement plus régulière que la partie opérative et se prête mieux à l'abstraction et aux algorithmes de transformation. L'avantage des versions microprogrammées sur les versions "glue-logic" a fait germer l'idée de transformer les parties contrôles afin de réduire la taille des simulateurs et d'accroître leur vitesse.

3.4.2 Définition de la synthèse

Cette solution au problème de la taille des simulateurs est appelée **synthèse** dans cette ouvrage. Certains pourront considérer cette dénomination comme un abus de langage dans la mesure où le terme synthèse est, en principe, défini en CAO comme la transformation d'une description fonctionnelle en une description structurale avec l'idée de produire une représentation physique d'un circuit. Il est nécessaire de présenter plus en détail ce qu'est la synthèse dans le cadre de cette Thèse

avant de justifier l'emploi de ce terme.

La **synthèse** est ici la transformation des parties contrôles sous une forme adaptée à la simulation rapide [DBP90]. C'est une synthèse orientée pour la simulation.

La fonctionnalité d'une partie contrôle est synthétisée sous forme de données, utilisées au moment de l'exécution (*run-time*). Ce principe de représentation d'une fonction sous forme de données, alors appelé *ROM approach*, a été utilisé dans [Arm89] pour la description de fonctions logiques simples.

Dans le cas présent, le code C d'un simulateur est composé d'une partie purement algorithmique (instructions en langage C) et d'une partie sous forme de données sur lesquelles sera exécuté l'algorithme de simulation (voir figure 3.4). Celui-ci est relativement identique quelque soit l'architecture cible alors que les données sont bien sûr totalement dépendantes de celle-ci.

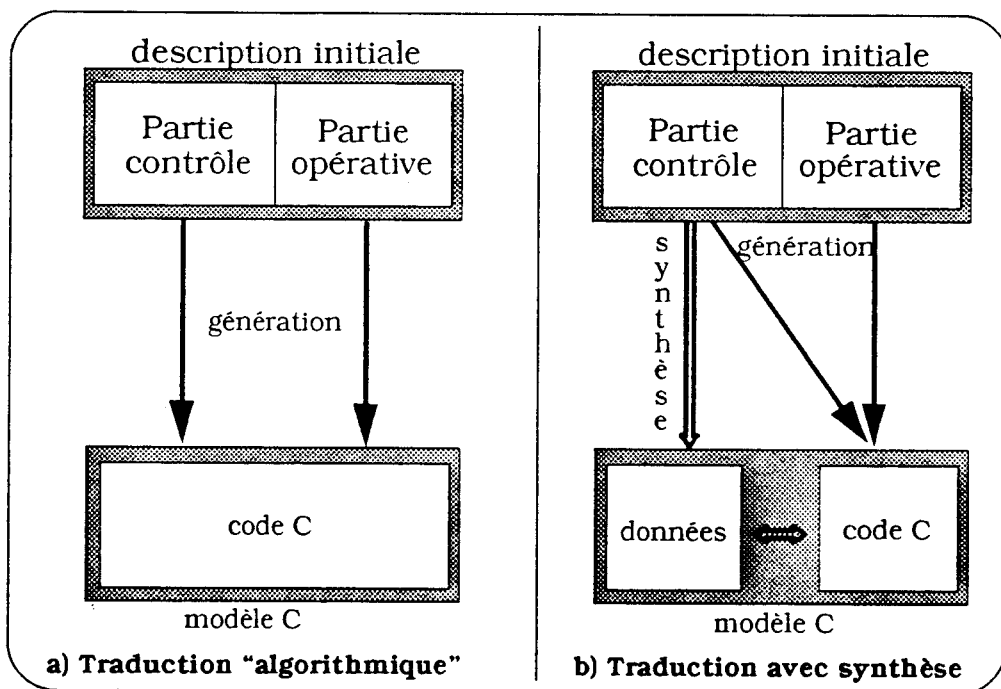


Figure 3.4 Différentes compositions d'un simulateur

Cette séparation des données et des opérations de contrôle, permet avec un peu d'expérience, de paramétrer une architecture cible en autorisant la modification des données, ce qui procure une souplesse supplémentaire au système. Les données peuvent d'ailleurs être écrites à l'extérieur du simulateur, selon le choix du concepteur/utilisateur. Comme

l'indique la figure 3.1, la possibilité a été laissée à l'utilisateur d'utiliser ou non la synthèse.

3.4.3 Le fonctionnement de la synthèse

Si le principe de la méthode est très simple, elle n'en nécessite pas moins plusieurs étapes de transformation des parties contrôles et requiert une bonne décomposition fonctionnelle et/ou structurelle de l'architecture cible. En effet, dans le cas de décompositions de mauvaise qualité, l'espace mémoire nécessaire au stockage des données peut s'avérer gigantesque; une partie contrôle non décomposée comportant 32 entrées et 128 sorties logiques occupe, sous forme de données, un espace de 64 giga-octets!

Avant de présenter comment obtenir une bonne décomposition, il est nécessaire de faire un bref rappel sur les architectures et leur partie contrôle.

a. Bases relatives à la conception de parties contrôles

Le rôle de la partie contrôle d'un processeur consiste d'une part à lire, décoder et à déclencher l'exécution des instructions et à assurer d'autre part, la communication avec l'extérieur à travers les broches de contrôle. C'est surtout du cycle d'exécution d'une instruction dont il est ici question car les broches de contrôle, instruments de prise en compte d'informations extérieures de nature différente du code opération, impliquent l'exécution d'algorithmes spéciaux, ce qui sort du contexte de cette présentation.

Les registres (d'état, d'instruction,...) sont des objets élémentaires typiques de la partie contrôle. Celle-ci active les différentes unités en leur envoyant des signaux de commandes.

Quelle que soit la manière de réaliser la partie contrôle (câblée, microprogrammée horizontale ou verticale, à l'aide de PLD¹s ou de cellules normalisées), l'algorithme d'interprétation des instructions suit l'organigramme général de la figure suivante, qu'il faudrait d'ailleurs compléter par la description du traitement des signaux externes (initialisation, interruptions, synchronisation,...) afin de décrire complètement le fonctionnement d'un processeur.

1. Programmable Logic Device

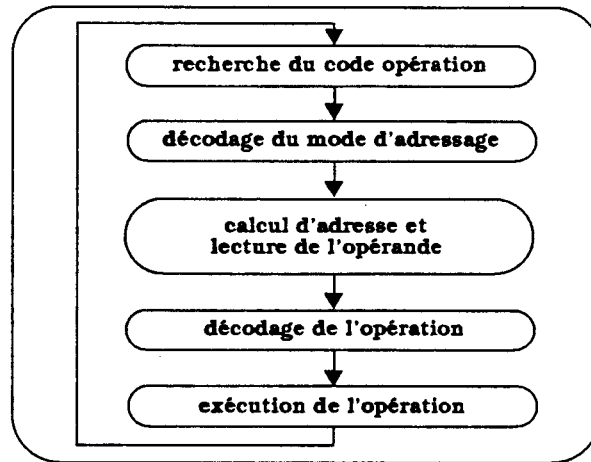


Figure 3.5 Algorithme d'interprétation d'une instruction

Cet algorithme d'interprétation peut agir à un ou plusieurs niveaux et détermine les choix d'implantation possibles. Excepté dans le cas de circuits simples, l'interprétation est divisée en plusieurs niveaux afin de réduire l'occupation sur le support de silicium d'une part, et dans le but de diminuer la complexité de la tâche de conception. Ainsi, un premier niveau d'interprétation consiste à reconnaître et à classer les instructions par genre (arithmétiques et logiques, branchement,...). De cette façon, il est possible de travailler sur des mots plus petits en utilisant des mécanismes de codage et de décodage. Les instructions de chaque genre sont ensuite ventilées vers des unités physiques qui traitent toutes les données recues de la même manière.

b. La décomposition des parties contrôles

La décomposition fonctionnelle doit être effectuée par le concepteur et implique l'identification et la séparation:

- des fonctions de séquençement
- des opérations de décodage
- des fonctions de synchronisation

A cet effet, le concepteur utilisera principalement les deux techniques suivantes:

- *L'extraction de propriétés:* Elle implique la sélection d'un sous-

ensemble de conditions suffisantes pour prendre une décision. Il s'agit de dégager en une seule fois, lors du décodage du code opération, les caractéristiques de l'instruction qui restent fixes tout au long de son interprétation (lecture ou écriture, opérande = registre ou mémoire).

• *Le mécanisme de paramétrage*: il s'agit d'extraire les paramètres qui correspondent à des commandes provoquant l'exécution d'une opération spécifique par un bloc fonctionnel. Cette technique concerne surtout des blocs tels que l'UAL ou le circuit de positionnement du registre d'instruction. Ces blocs exécutent une seule fonction spécifique au cours de l'interprétation d'une instruction et une opération par défaut est exécutée pendant les autres cycles. Il est donc possible de positionner les commandes spécifiques dès la lecture du code de l'instruction. La technique consiste donc à générer de façon statique des commandes en fonction des paramètres propres aux unités fonctionnelles.

La décomposition structurelle est totalement dépendante du langage de description utilisé. Dans le cas de ce projet, il s'agit du langage LIDO¹. Celui-ci sera présenté au chapitre 4 en même temps que le système LIDO. Le lecteur impatient de connaître les caractéristiques des éléments du langage se reportera donc immédiatement à la section correspondante afin d'assouvir sa curiosité et d'être ainsi en mesure de poursuivre sereinement sa lecture, que l'auteur espère enrichissante.

Le concepteur devra utiliser, pour décrire la partie contrôle de son circuit, des decodeurs, registres variés, des mémoires ROM (versions micro-programmées) et des PLA²s (à un ou plusieurs niveaux). Il peut aussi, dans le cas de circuits peu complexes ou s'il ne désire pas utiliser la synthèse, employer la structure de graphe qui donne de meilleurs résultats que les PLAs dans le cas de la traduction algorithmique.

c. Les étapes de la synthèse

Le principe général de la synthèse consiste à transformer les éléments structurels de la partie contrôle sous forme de PLAs complètement spécifiés (PLACS), c'est-à-dire comportant les matrices ET et OU, dont les fonctions propres sont ensuite transformées sous forme de données, utilisées lors de l'exécution par le simulateur. Pour ce faire, on procède en deux phases, chacune divisées en plusieurs étapes. Ces phases sont présentées ci-dessous, accompagnées éventuellement de figures expliquant leur

1. Langage Interprétable pour la Description des Ordinateurs

2. Programmable Logic Array

fonctionnement. Des exemples donnés au chapitre 4 sont associés à chacune, auxquels le lecteur pourra se reporter de suite afin d'améliorer ou de vérifier sa compréhension des étapes et figures.

Lors de la première, les modules complexes tels que séquenceurs, PLAs et décodeurs sont transformés en PLACSs optimisés. Ceci s'effectue en plusieurs étapes:

• **séparation des chemins de données et des chemins de contrôle**

Cette étape extrêmement simple, présentée par la figure 3.6, consiste en l'itération des actions suivantes:

- reconnaissance d'une action sur les données dans un graphe ou un PLA,
- remplacement de ce chemin par un drapeau,
- recopie de l'action dans une unité fonctionnelle créée à cet effet. Le drapeau est chargé de valider/d'invalider l'activation de ce chemin.

La figure 3.6 présente l'étape dans le cas de l'identification d'un chemin de données dans un graphe. Si la figure et certains des termes employés (noeud, condition) ne sont plus valables dans le cas d'un PLA, le principe reste le même.

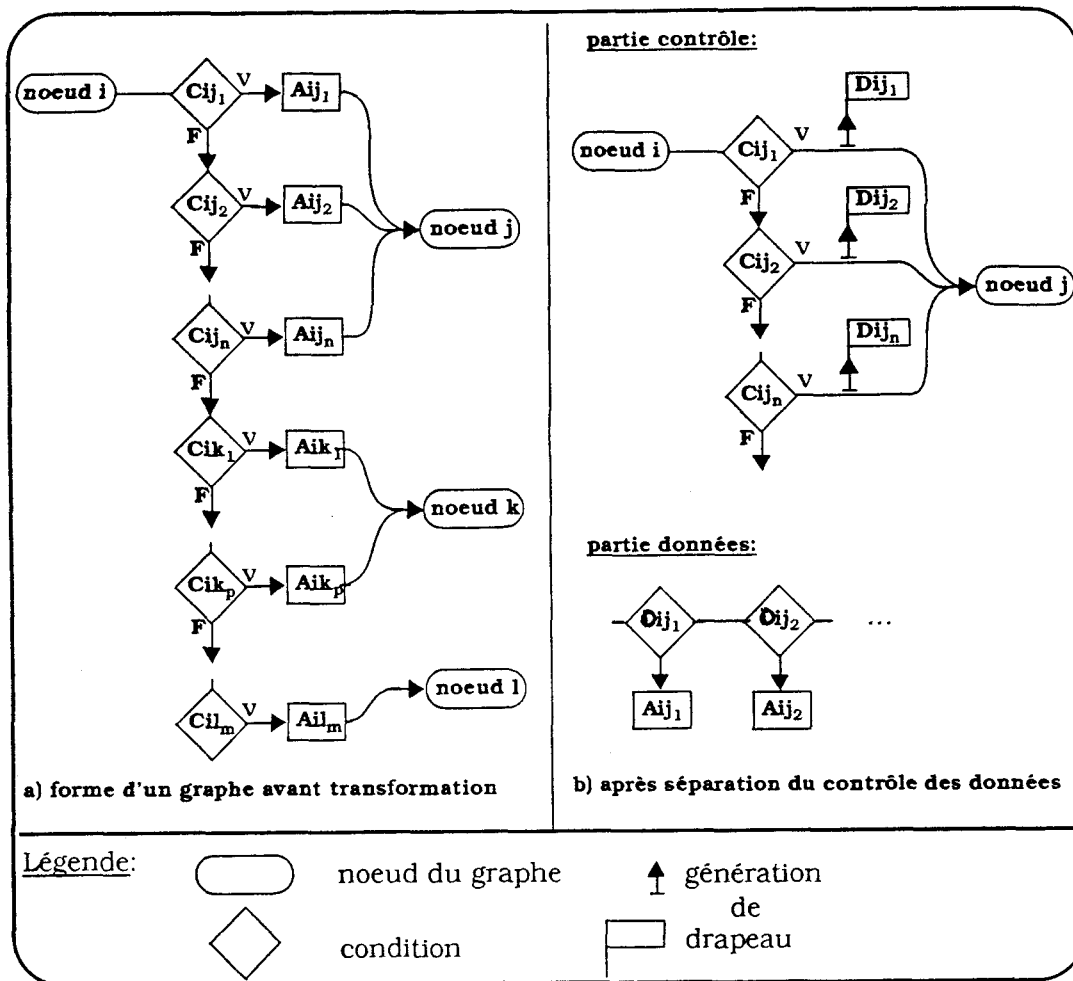


Figure 3.6 Séparation du contrôle et des données dans un graphe

• **transformation des structures de contrôle obtenues en PLAs optimisés**

C'est l'étape la plus complexe de cette phase. Elle comprend deux parties non triviales:

- optimisation des graphes obtenus

La saisie sous forme de graphe offre un niveau d'abstraction élevé, facile d'utilisation pour le concepteur. Malheureusement, cette forme d'écriture n'est pas compatible avec l'obtention de parties contrôles compactes. Ainsi, les PLACS obtenus seraient à l'image des graphes, c'est-à-dire surdimensionnés. Il est donc nécessaire d'optimiser les graphes de contrôle avant de les transformer en PLAs. Ceci est particulièrement

important dans l'optique de la synthèse car la simple économie de deux entrées par exemple permet de diviser par quatre l'espace occupé par les données.

L'optimisation dont il est ici question ne concerne pas la réduction de l'occupation physique d'un circuit ni l'amélioration des performances de ce circuit car ces opérations sont très dépendantes de la technologie employée au moment de l'implantation. Néanmoins, l'objectif visé peut parfois conduire à l'une ou l'autre de ces améliorations.

L'objectif est de parvenir à une réduction de la taille des PLAs (de leurs modèles) et à un accroissement de la vitesse de simulation. Les articles concernant l'optimisation de PLAs publiés à ce jour, et Dieu sait¹ s'ils sont nombreux [WC90, CM88, ADN90, DeM84, De88, HQ88, PP89, De89, WC88, De_88, SK88, AB89], sont à la connaissance de l'auteur relatifs à l'implantation des PLAs et non à leur simulation. L'adaptation pour la simulation de certaines des méthodes proposées comme par exemple [YC89], amènerait une réduction (faible) de la taille des PLAs, mais conduirait à une réduction des performances non négligeable compte tenu du faible gain en espace obtenu en contrepartie.

L'algorithme d'optimisation a dû être spécialement défini pour la simulation. Celui-ci est relativement simple à comprendre quoique non trivial à écrire. Il consiste simplement à réunir ensemble les états de même niveau, c'est-à-dire ceux se trouvant à égale distance du noeud initial et n'étant pas reliés entre eux par une relation de descendance (du noeud initial à chacun d'eux, on ne passe par aucun des autres). Ceci conduira à réunir les conditions de ces noeuds lors de la transformation en PLA mais cela n'a pas de conséquence sur sa taille car c'est la réunion de toutes les conditions attachées aux noeuds du graphe (voir figure 3.7) qui intervient dans la taille du PLA final.

C'est le parcours du graphe et la reconnaissance de ces noeuds qui constitue la partie délicate de cette étape. Il faut en effet éviter de propager les conditions héritées d'un noeud à ses descendants pour écarter les problèmes de boucles. Afin d'éviter de nombreux retours en arrière coûteux lors de cette identification, une procédure préalable à la reconnaissance initialise une structure de donnée indiquant pour chaque noeud l'ensemble de ses descendants.

1. L'emploi de cette expression, empruntée au langage parlé, n'implique en aucune façon l'adhésion de l'auteur à une quelconque idéologie religieuse.

- transformation des graphes et PLAs divers en PLACS

C'est la transformation d'un graphe en un PLACS qui requiert le plus d'efforts. Elle est présentée par la figure 3.7. Tout d'abord un PLA est créé de telle sorte qu'il soit activé au même top horloge que le graphe dont il devra réaliser la fonction. Il peut alors être nécessaire de renommer les unités afin de conserver l'ordre implicite d'activation (voir section 4.1). Un graphe issu de l'étape précédente peut être défini par $G=(\mathcal{N}, C, \mathcal{F}, \mathcal{T})$ où

$\mathcal{N}$ est l'ensemble des noeuds de G ,

C est l'ensemble des conditions C_{ij_k} associées aux arcs de $\mathcal{N}_i$ vers $\mathcal{N}_j$,

$\mathcal{F}$ est l'ensemble des fonctions attachées aux arcs qui associent une valeur (0 ou 1) à un drapeau,

$\mathcal{A}$ est l'ensemble des arcs de G ,

Un registre d'état est créé pour coder l'ensemble $\mathcal{N}$. Sa taille est fonction du cardinal de $\mathcal{N}$ ($t_r = E(\log_2(\text{Card}(\mathcal{N})))$). Ceci permet de réduire le nombre de lignes d'état nécessaires et de minimiser la taille du nouveau PLA. En effet, le registre, divisé en t_r variables, compose avec la réunion des variables de C (respectivement de la réunion des drapeaux) l'ensemble des entrées (respectivement des sorties) de ce PLA.

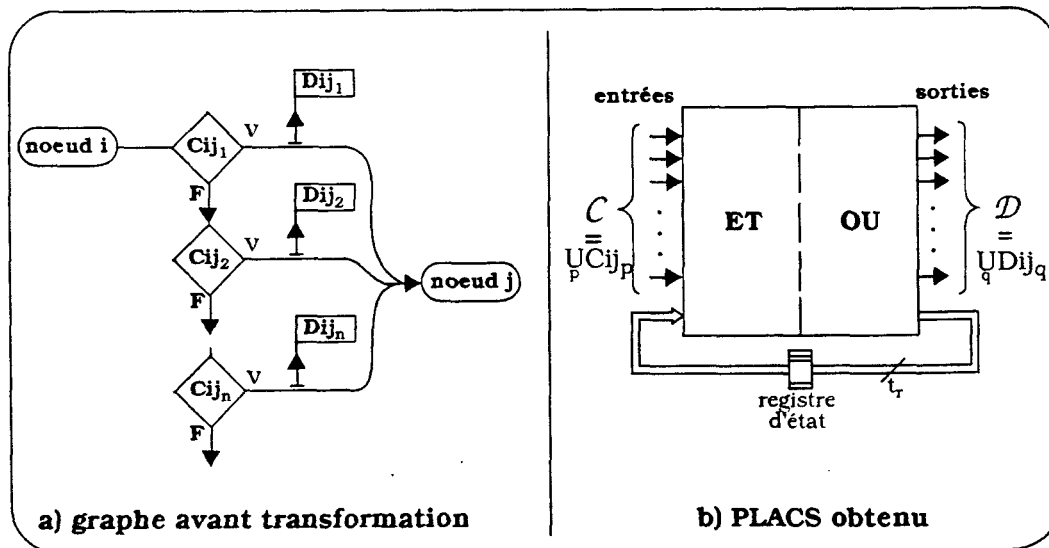


Figure 3.7 Transformation d'un graphe de contrôle en PLACS

Le contenu des matrices ET et OU en découle logiquement. Pour chaque arc de G , il existe une (unique) condition vraie (C_{ij_p}) qui détermine le passage

du noeud de départ (N_i) vers celui d'arrivée (N_j). Le numéro du noeud de départ ainsi que les variables intervenant dans cette condition déterminent les valeurs d'une ligne de la matrice ET. Les autres variables (celles de l'ensemble $C-Q_j$) n'interviennent pas dans la ligne correspondante et produisent la valeur 'x' (*don't care*). On procède de la même manière pour tous les arcs, ce qui détermine la totalité de la matrice ET.

De la même façon, le contenu de la matrice OU est déduit du code du noeud N_j et de la valeur des drapeaux ($\mathcal{D}$). Sachant qu'au plus un drapeau est présent par arc, il y a au plus une valeur à un par ligne de la matrice OU (si l'on excepte le code du noeud).

Le cas de la transformation d'un PLA en un PLACS est plus simple. D'une part, le renommage des unités n'a pas lieu car l'ordre implicite d'activation n'est pas modifié. D'autre part, il n'y a pas de codage des noeuds et la matrice ET existe déjà.

La seconde phase fait partie du back-end. Au cours de celle-ci, les structures obtenues sont à nouveau transformées et générées sous forme de tableau de données en langage C. Cette phase est, elle aussi, divisée en plusieurs étapes:

- **expansion des PLAs**

C'est lors de cette étape que la fonctionnalité de la partie contrôle d'un circuit est transformée sous forme de données pour être projetée sur la mémoire de la machine hôte.

Les valeurs non significatives (*don't care*) sont éliminées et remplacées par zéro et un, ce qui produit deux lignes pour chaque 'x' de la matrice ET. La partie OU est pour sa part, simplement dupliquée dans les deux cas. Ainsi, les mintermes sont produits pour toute combinaison des entrées. Il est possible que l'expansion de plusieurs lignes, dont une au moins contient une valeur 'x' et ayant des parties OU différentes, produise la même partie ET. Il est dans ce cas nécessaire d'effectuer un OU logique bit à bit entre les parties OU. Ainsi, l'expansion réalise l'opération OU au moment de la génération, qui n'a plus à être exécutée lors de la simulation. Enfin, la partie OU est codée sous forme d'entiers pour être utilisée à l'exécution de façon efficace.

Le terme expansion est relatif à la multiplication des lignes du PLA. Celles-ci peuvent être représentées par une somme logique de produits (forme disjonctive simplifiée) ne faisant intervenir que certaines variables pour chaque produit. Selon ce principe, l'expansion peut être vue comme la

mise sous forme normale disjonctive (toutes les variables apparaissent dans chacun des termes).

• **compression éventuelle des données**

Cette étape est optionnelle. Son usage dépend du choix du concepteur et/ou des caractéristiques du circuit.

Les données obtenues peuvent occuper dans certains cas un espace important (voir table 5.2). En dehors de la taille de l'espace mémoire qui doit leur être réservé, le temps de compilation peut croître de façon dramatique jusqu'à ébranler la patience de l'utilisateur¹. Deux types de solution sont envisageables.

La première consiste simplement à générer les données dans un fichier séparé, ce qui résout le problème du temps de compilation (Pb1) mais pas celui de l'espace mémoire (Pb2). L'emploi de cette solution est au choix de l'utilisateur. Pour être utilisable en pratique, elle requiert la simulation du circuit pendant de nombreux cycles de manière à minimiser la part consacrée à la lecture des données lors de l'initialisation.

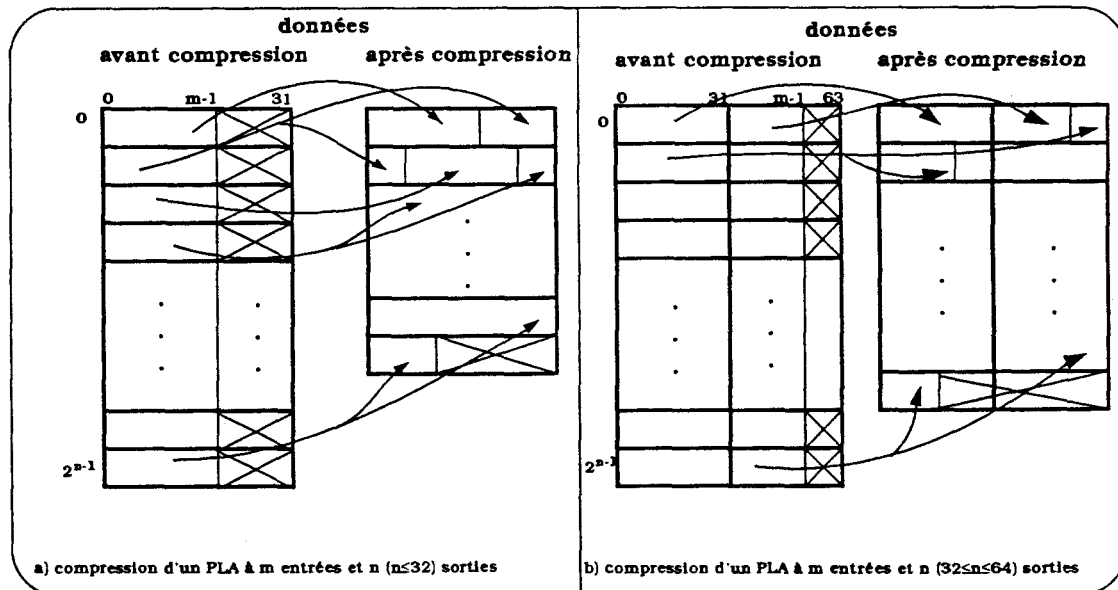


Figure 3.8 Compression des données avant la génération

La seconde solution consiste à compresser les données avant leur génération (voir figure 3.8). Cette solution est dédiée aux deux problèmes et

1. Ne vous souvenez vous pas d'avoir, un jour ou l'autre, pressé la touche CTRL-C pour mettre fin à l'insoutenable attente du résultat?

donne des résultats variables suivant les caractéristiques du circuit traité. Son usage dépend d'un choix effectué par le générateur mais il peut être forcé dans tous les cas par le concepteur. Cette décision du générateur est effectuée en fonction du rapport estimé entre:

- le gain en espace mémoire qui sera obtenu. Celui-ci est calculé de façon précise en fonction des caractéristiques de la partie contrôle.
- la perte d'efficacité due à un accès aux données plus complexe. Celle-ci est estimée en se basant d'une part sur des évaluations effectuées au cours d'expériences antérieures et d'autre part, sur la taille d'un mot du circuit. Lors de la simulation, l'accès aux données consiste à recomposer les mots initiaux. Cette "décompression" pourrait être représentée par les flèches de la figure 3.8 inversées, l'opération de compression étant heureusement bijective.

• génération des tables de données en langage C

Cette étape est triviale et consiste simplement à générer le contenu des tables de données soit dans le simulateur, soit dans un fichier séparé.

3.5 Emploi de la synthèse lors de l'exécution

L'usage des données produites par les différentes étapes de transformation est très simple, ce qui explique l'efficacité de cette méthode. Les différentes opérations sont visualisées par la figure 3.9 dans le cas de données non compactées et d'un PLA ayant moins de 32 entrées et sorties logiques.

Lors de l'exécution, les valeurs des entrées d'un PLA sont utilisées pour calculer l'adresse (F1) à laquelle la fonction sélectionnée (F2) pour la phase d'horloge en cours se trouve mémorisée. Il ne reste plus qu'à exécuter cette fonction (F3), c'est-à-dire à affecter aux sorties chacun des bits de la donnée.

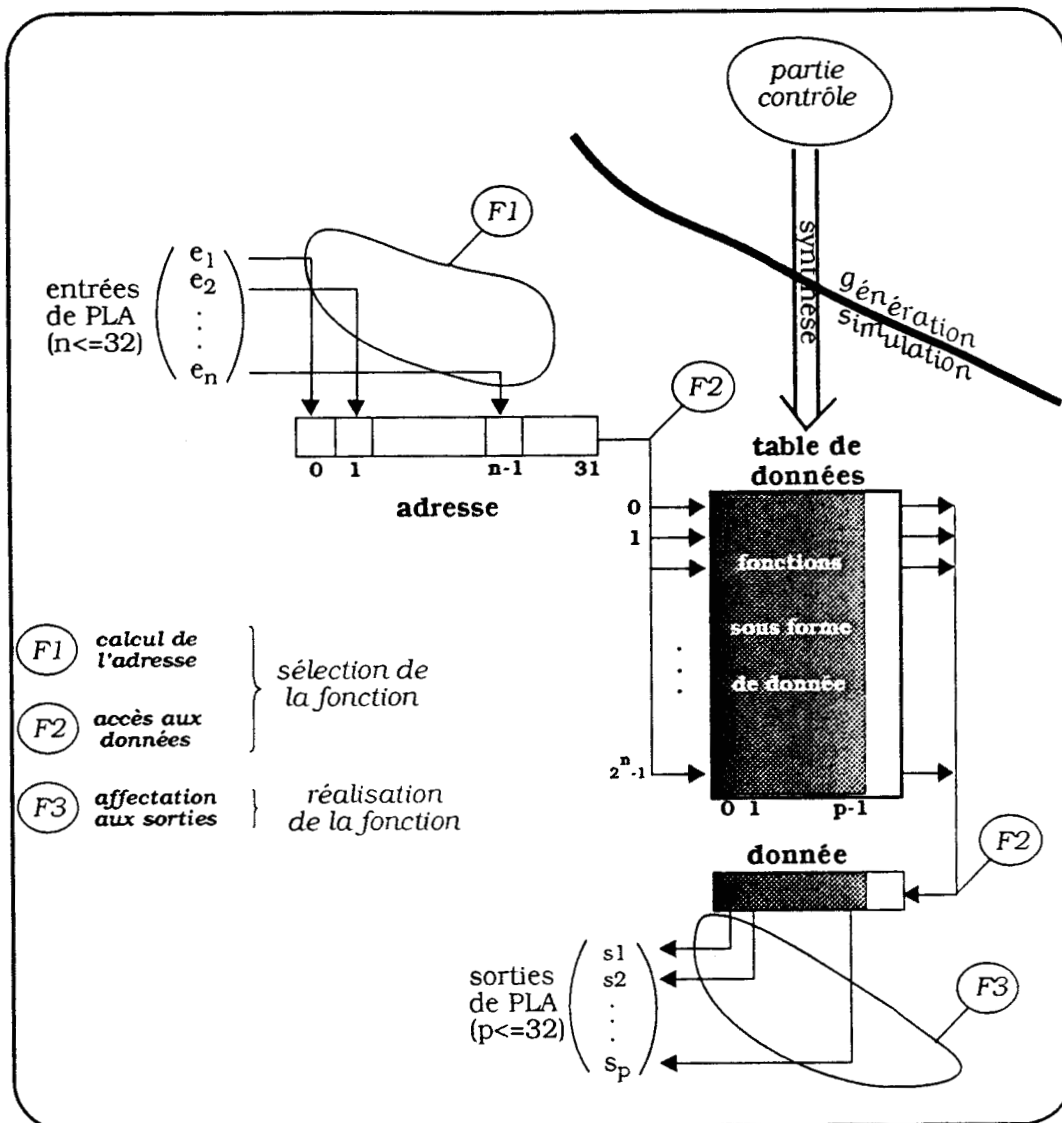


Figure 3.9 Emploi des données à l'exécution

3.6 Conclusion

L'humanité [...] est un puits à deux seaux. Pendant que l'un descend pour être rempli, l'autre monte pour être vidé.

Samuel Beckett¹

Un outil, quel que soit son domaine d'application, est caractérisé par plusieurs aspects liés les uns aux autres. Comme le laissent entendre ces

1. Tiré de *Murphy*, éditions de minuit

mots de Samuel Beckett, l'amélioration d'un aspect donné va de pair avec la perte de performance concernant un autre critère, selon le principe des vases communicants. Les choix effectués qui contribuent à chacun des objectifs visés ont donc une influence (bonne ou mauvaise) sur les autres aspects de l'outil.

Ainsi, un certain nombre de compromis ont été mis à jour:

1) Compilation/Interprétation: la compilation permet d'obtenir de meilleures performances que l'interprétation mais se prête mal à la réalisation d'un interface de haut niveau.

2) Vitesse/Taille: l'emploi de macros et de fonctions *run-time* très spécifiques contribue à accroître la vitesse de simulation mais implique une augmentation de la taille des simulateurs et du modèle compilé. De même, la réduction de la taille des données par leur compactage va de pair avec la perte de performance.

3) Généralité/Spécificité: l'usage de simulateurs très spécifiques (section 2.3.3) permet d'obtenir les meilleurs résultats en terme de performance mais l'usage de ces simulateurs est limité et nécessite la génération et l'exécution de nombreux simulateurs afin de vérifier la fonctionnalité d'une architecture.

Certains des objectifs fixés (voir section 3.2) apparaissent donc difficilement conciliables. Les choix généraux concernant le générateur et les simulateurs (voir section 3.3) ont été effectués en conséquence. Ils devraient permettre d'obtenir les résultats escomptés grâce à la synthèse de parties contrôles qui augmente la vitesse de simulation tout en réduisant la taille des simulateurs.

Chapitre 4

L'implémentation

L'objectif de ce chapitre est la description des structures de données et algorithmes utilisés par les simulateurs, par le générateur et par les fonctions run-time.

Si la solution proposée comme moyen de simulation est indépendante du langage de description employé, son implémentation est fortement liée à l'environnement et aux moyens disponibles. C'est pourquoi celui-ci est présenté en premier lieu. Il est constitué de deux parties dont la plus intéressante est sans doute celle du langage LIDO car ses objets conditionnent les structures de données utilisées par les simulateurs et, à un degré moindre, l'efficacité des algorithmes.

La partie suivante du chapitre présente d'abord ce qui est exécuté lors de la simulation (ce qui est généré, c'est-à-dire les algorithmes et les SDs), puis les étapes de la traduction (analyse, transformations et génération du code C).

4.1 L'environnement

C'est dans l'environnement LIDO que la solution présentée au chapitre précédent a été implémentée. Celui-ci est composé du système LIDO et de son langage. Ce système a été développé en C et existe sous UNIX ainsi que sous MS-DOS. La solution a été réalisée sous UNIX sur station de travail SUN et HP et peut être facilement portée à d'autres systèmes d'exploitation, comme cela a été le cas sur DEC-VAX 8xxx (système VMS).

4.1.1 Le langage LIDO

LIDO est l'unique langage de description du système bien qu'il soit possible d'appeler des sous-programmes écrits dans un autre langage. C'est aussi, à l'heure actuelle, l'unique mode de spécification des circuits; des outils de saisie sous forme graphique et de transformation sous forme de langage ont certes été réalisés mais aucun d'eux n'a encore été intégré au système.

LIDO est un langage de description non procédural qui permet de décrire des circuits séquentiels synchrones au niveau transfert de registre (comportemental ou structurel) et au niveau algorithmique (comportemental). Le contrôle des opérations est effectué au moyen de *guards* et de synchronisation sur horloge.

Les seules unités temporelles en LIDO sont le cycle et les phases horloge. De ce fait, LIDO est un langage purement fonctionnel dont le seul

but est la vérification (fonctionnelle) des architectures, classiques ou multi-processeurs.

La syntaxe et la sémantique du langage sont présentées plus en détail dans [Bak89], où le lecteur pourra aussi trouver de nombreux exemples de descriptions d'architectures comme des circuits plus petits.

a. Les types de données

Le langage n'offre pas à l'utilisateur la possibilité de définir ses propres types de données et seuls, sept types de données sont définis en LIDO. Ceux-ci sont présentés ci-après, accompagnés de quelques exemples de déclarations. La syntaxe du langage est donc très simple et permet de faire référence à un type d'objet simplement à l'aide d'une lettre capitale.

Une déclaration commence par une lettre capitale et se termine par un point, le point-virgule étant le séparateur d'identificateurs. Il n'y a pas d'espace réservé aux déclarations et celles-ci peuvent apparaître n'importe où, excepté à l'intérieur des unités de traitements.

Les terminaux (T):

Ce sont des porteurs d'information sans mémoire ou à mémoire transitoire (dans ce cas, l'objet retrouve sa valeur par défaut après une durée spécifiée dans la déclaration). Les terminaux correspondent aux connexions à l'intérieur d'un circuit.

Syntaxe:

T...; <i>nom.</i> ou T...0; <i>nom.</i>	<i>nom</i> est un terminal dont la valeur par défaut est 0.
T...1; <i>nom.</i>	<i>nom</i> est un terminal dont la valeur par défaut est 1.
T...#; <i>nom.</i>	<i>nom</i> est un terminal normalement en haute impédance.
T...[2]; <i>nom.</i>	<i>nom</i> retient sa valeur pendant deux cycles horloge (défaut = 0).

Exemple:

Term; abus(20).	bus d'adresses (20 bits).
------------------------	---------------------------

Les terminaux d'entrée (I):

Ce sont des porteurs d'information utilisés comme signaux d'entrée d'un circuit qui ne peuvent pas être modifiés en cours d'exécution. Ils correspondent aux broches d'entrée et véhiculent les stimuli et vecteurs de test.

Exemple:

Input; reset;	signal de réinitialisation.
data(32);	32 bits de donnée.
int(3).	trois lignes d'interruptions.

Les verrous (V | L):

Ce sont des porteurs d'information avec mémoire simple de type tampon. Quatre objets de ce type sont prédéfinis et servent à la génération de valeurs aléatoires pour les tests. Appelés *random* (respectivement *random2*, *random4* et *random8*) ils sont calculés tous les cycles (respectivement tous les deux, quatre et huit cycles) horloge.

Exemple:

Latches; int; ligne d'interruption avec son verrou.
 abus(20); 20 lignes du bus d'adresses avec le tampon de sortie.

Les registres (R):

Ce sont des porteurs d'information de type maître-esclave. L'information est écrite dans la partie maître et lue dans la partie esclave. A chaque top horloge, la partie maître devient active et "écrase" la partie esclave.

Exemple:

Reg; pc(16); pointeur de programme.
 flags(9); indicateurs.
 acc(32). accumulateur.

Les multiplexeurs (X):

Les multiplexeurs et démultiplexeurs sont, comme les terminaux, des porteurs d'information sans mémoire. Par contre, ce sont des objets à deux dimensions dont la valeur par défaut ne peut être que 0 ou 1. Ils permettent de sélectionner des données¹ diverses.

Exemple:

Xer; mul(8,16). multiplexeur à 8 voies de 16 bits.
Xer1; demul(32,1). démultiplexeur sur 32 voies de 1 bit, sortie normalement à 1.

Les mémoires (M):

Ce sont des porteurs d'information bidimensionnels qui retiennent l'information.

Exemple:

Mem; ram(64K, 8). 64 K octets de mémoire ram.

Les mémoires associatives (A):

A la différence des mémoires, l'accès aux mémoires associatives nécessite une clé pour sélectionner le type d'accès (voir paragraphe c).

1. "données" est ici utilisé au sens large; il peut s'agir d'adresses par exemple.

Exemple:

Amem; ass(64, 32, 8). 64 mots de 32 bits chacun avec une clé de 8 bits.

b. Les facilités de déclarations

Outre la simplicité de sa syntaxe, LIDO offre des facilités de déclarations visant d'une part à obtenir des parties déclaratives compactes, et d'autre part, à limiter les tâches fastidieuses pour l'utilisateur.

Ces possibilités se divisent en trois catégories et sont présentées ci-dessous.

Les déclarations multiples (#):

Surtout utiles pour la description de systèmes multi-processeurs, elles permettent de déclarer des éléments de même type et de mêmes caractéristiques (taille, valeur par défaut,...) en une seule déclaration. Ceci correspond à l'instantiation automatique d'objets génériques qu'offrent certains langages. De tels objets comportent alors un numéro utilisé lors de leur utilisation au sein d'une unité de description, elle aussi générique et associée au même numéro.

Exemple:

Reg; R(16)#4. déclaration de quatre registres de 16 bits: R0, R1, R2, et R3.
Mem; Rom(1K, 8)#3. trois mémoires d'un K-octets chacune: Rom0, Rom1, Rom2.

Les sous-éléments (S):

Ils permettent de définir et d'associer des objets à des parties d'éléments déjà créés. Les sous-éléments sont l'équivalent des alias de certains langages. Ainsi, les différentes parties d'une instruction peuvent par la suite être référencées chacune par un nom différent sans avoir à spécifier à chaque fois les bits concernés.

Exemple:

Reg; ri(16).
Subr; ri(15,12)=cop; code opération
 ri(11,0)=cai. champ adresse de l'instruction

Des alias peuvent aussi être créés conjointement aux déclarations de génériques comme le montrent les exemples suivants:

Ter; dbus(8)=db#2. db0=db1=dbus(8) : deux alias génériques
Sub; dbus#; dbus0=dbus(0)...., dbus7=dbus(7)
 ri(15,12)=cop#2. cop0=cop1=ri(15,12)

Les équivalences (=):

Elles permettent de déclarer des éléments ayant la même valeur tout au long d'une simulation ou des éléments contenant les mêmes valeurs mais organisées de façon différente (cas des mémoires et multiplexeurs). En définissant l'équivalence d'objets d'un circuit, les "copies" d'un objet source sont mises à jour automatiquement.

Exemple:

Term; a=b=c.

trois signaux issus d'un même fil.

Mem; Rm(8, 32)=r16(16, 16)=r32(32, 8).

trois adressages de la mémoire

c. Opérations et conditions

Les conditions logiques et les opérations arithmétiques et logiques définissent les types d'expression que l'on peut construire en LIDO. Les opérateurs permettent d'exprimer le fonctionnement des circuits tandis que les conditions constituent le moyen de contrôle du langage.

Les opérations:

Les opérateurs disponibles sont ceux habituellement rencontrés dans les langages classiques enrichis d'un certain nombre plus spécifiques au traitement des circuits. Ces opérateurs sont les suivants:

<i>logiques:</i>	<i>arithmétiques:</i>	<i>logiques en largeur:</i>	<i>relationnels:</i>
& et	+ addition	/& et	> supérieur
ou	- soustraction	/ ou	> supérieur
^ ou exclusif	* multiplication	/^ ou exclusif	>= supérieur ou égal
! complémentation			<= inférieur ou égal

les transferts:

-> transfert de contrôle (graphe de transition)
 => transfert de valeur (affectation immédiate)
 \$ transfert d'exécution (appel d'unité)

l'adressage:

@ accès à une mémoire

A l'exception du transfert de contrôle et des opérations en largeur (qui s'appliquent à un vecteur), tous les opérateurs sont définis sur un ou plusieurs bits.

En ce qui concerne l'adressage, il faut distinguer celui des mémoires associatives de celui des autres éléments, qu'il s'agisse de mémoires ou de registres. L'accès aux mémoires associatives nécessite une clé d'adressage et peut se faire selon trois modes différents dans le cas de la lecture.

Les conditions:

On distingue en LIDO deux types de conditions. D'une part, les conditions logiques ont la forme disjonctive c'est-à-dire composées d'une somme de termes. Ce sont des gardes qui apparaissent dans le texte entre slashes. D'autre part, les conditions d'horloge permettent de réaliser la synchronisation des circuits. Elles apparaissent entre crochets dans une description. Ces deux types de conditions peuvent être combinées pour effectuer la synchronisation des unités.

Exemple:

/a&b ri(8,5)/&/	condition logique, vraie si a&b ou si les bits de ri sélectionnés sont à 1.
[1 3]	condition d'horloge, vraie aux phases d'horloge 1 et 3.
[2]/state=1/	condition vraie à la phase 2 si state a la valeur 1.

d. Les unités de descriptions

Une description LIDO est composée de parties décrivant le fonctionnement d'un circuit et de déclarations d'objets globaux situées à l'extérieur de ces parties. Ce sont ces parties qui constituent les unités de description. Elles permettent de grouper les informations relatives aux opérations effectuées soit par la partie contrôle, soit par la partie opérative, soit encore par l'ensemble du circuit. Trois types d'unités de description sont disponibles en LIDO. Comme les autres objets du langage, elles sont déclarées par une lettre capitale. Chacune de ces unités est présentée ci-dessous, accompagnée d'un exemple simple; il s'agit d'un compteur/décompteur sur seize bits.

Les graphes de transitions (G):

Ils correspondent au concept d'automates d'états finis et permettent de décrire au niveau algorithmique la partie contrôle d'un circuit (graphe de contrôle, noté CG¹) ou sa totalité (graphe de contrôle et de donnée, noté CDFG¹).

Le contrôle dans un graphe est effectué à l'aide de conditions, une condition par défaut étant indiquée par deux slashes consécutifs ("//"). Le transfert de contrôle, c'est-à-dire le branchement à un noeud donné s'effectue à l'aide de l'opérateur "->".

1. CG: Control Graph, CDFG: Control Data Flow Graph

Exemple (Eg1):

```

Gra 1,1;;
n0: /up/ : count/& => rstmin -> n1,
    // : count! => rstmax -> n2;
n1: /rstmin/ : 0 => count -> n0,      n1 : compteur
    // : count +1 => count -> n0;
n2: /rstmax/ : %xffff => count -> n0,  n2 : décompteur
    // : count -1 => count -> n0.

```

Les PLAs (P):

Ces unités, ainsi que les décodeurs qui entrent dans la même catégorie, permettent de décrire la partie contrôle des circuits. Ce sont des constructions structurelles qui expriment une forme d'implantation des fonctions combinatoires et séquentielles très utilisée dans la conception des systèmes VLSI.

Partant du principe que toute fonction logique peut être représentée par sa table de vérité (n entrées e_i , p sorties s_j) et réalisée avec de la logique à deux niveaux, un PLA consiste à décoder les lignes de la table de vérité qui ont pour valeurs celles des n entrées e_i et à affecter aux sorties s_j les p valeurs ainsi sélectionnées.

Un PLA est donc constitué de deux matrices. Celle de gauche, qui réalise le décodage, est appelée matrice ET car toutes les entrées doivent correspondre aux valeurs d'une ligne de la table de vérité. Celle de droite, qui réalise finalement la fonction, est appelée matrice OU car lorsque plusieurs lignes ont été sélectionnées, c'est le OU logique des sorties de tous les décodeurs qui doit être affecté aux sorties du PLA.

Il existe trois formes de spécification des PLAs en LIDO. La première, appelée PLA code double (voir PLA 1,1), est constituée des matrices ET et OU alors que dans la deuxième, appelée PLA code simple (voir PLA 1,2), seules les valeurs des sorties sélectionnées sont indiquées, dans un souci de concision et de clarté. La matrice OU d'un PLA code double peut aussi comporter la valeur '*' (*don't care*). Celle-ci indique que la sortie correspondante n'est pas modifiée. Enfin, dans la troisième forme (voir PLA 1,3), la matrice OU est remplacée par des instructions exécutées lorsque la ligne correspondante est choisie.

Exemples:

```

PLA 1,1; up, count/&, count! - /* entrées */
rstmin, rstmax, plus, moins; /* sorties */
/*ET - OU */
(11x) - (1000); /* signal de remise à zéro du compteur */
(10x) - (0010); /* signal d'incréméntation */
(0x0) - (0001); /* signal de décrémentation */
(0x1) - (0100). /* signal de remise à un de tous les bits */

```

PLA 1,2; up, count/&, count! ; /* entrées */	PLA 1,3; up, count/&, count!;
/*ET - sorties */	/*ET - opérations */
(11x) - rstmin;	(11x) : 0 => count;
(10x) - plus;	(10x) : count +1 => count;
(0x0) - moins;	(0x0) : count -1 => count;
(0x1) - rstmax.	(0x1) : %xffff => count.

Les unités fonctionnelles (F):

Celles-ci (notées FUNs) servent à décrire la partie opérative d'un circuit. Cette description peut être purement fonctionnelle ou structurelle. Dans le premier cas, seul le comportement du circuit est décrit alors que dans le second les registres utilisés et les bus empruntés sont spécifiés.

L'unité fonctionnelle donnée en exemple pour le compteur doit être associée avec le PLA 1,1 ou le PLA 1,2 donnés en exemple car les opérations effectuées sont conditionnées par les sorties de ces PLA.

Exemple:

```

Fun 1,2;;
/rstmin/ 0 => count; /* remise à zéro du compteur*/
/plus/ count +1 => count; /* incréméntation */
/rstmax/ %xffff => count; /* remise à un de tous les bits */

```

e. Synchronisation et activation

Un circuit peut être décomposé de façon fonctionnelle ou structurelle à l'aide des objets complexes définis au paragraphe précédent. Les composants physiques auxquels ils sont associés sont, dans la réalité, activés à des moments différents soit du fait de leur position topologique bi(tri)dimensionnelle, soit du fait de signaux de synchronisation. LIDO étant un HDL non-procédural, l'ordre d'activation ne peut être décrit à l'aide de structures de contrôle.

La synchronisation des objets est spécifiée en LIDO à l'aide des

conditions d'horloge et des conditions logiques définies au paragraphe c.

L'ordre topologique intervient à un second niveau, c'est-à-dire après la synchronisation. Il est réalisé à l'aide de la matrice d'activation dont le mécanisme est présenté par la figure 4.1. Celle-ci est une structure bi-dimensionnelle composée d'emplacements où aucune, une ou plusieurs unités de descriptions peut être présente. Les unités présentes sont activées par niveau croissant, puis pour chaque niveau, par rang croissant, le niveau (respectivement le rang) définissant une ligne (respectivement une colonne) de la matrice.

Des unités se trouvant à un même emplacement sont activées dans un ordre implicite (troisième dimension) donnant la priorité aux objets de la partie contrôle sur ceux de la partie opérative. Pour des unités de même niveau et rang, un graphe sera activé avant un PLA, lui-même activé avant une unité fonctionnelle.

Mais il est des unités qui ne sont pas activées conformément à cette méthode. Il s'agit de procédures écrites en langage C par l'utilisateur et d'unités appelées par l'opérateur \$ à l'intérieur d'un autre objet complexe. Ces unités (PLA ou unité fonctionnelle uniquement) peuvent avoir été déclarées dans une autre description et sont alors recherchées dans une bibliothèque où peuvent avoir été définies pour la première fois dans la description où elles sont appelées.

Ces unités et procédures sont donc dépendantes des objets dans lesquels elles apparaissent pour ce qui est des conditions d'horloge et des conditions logiques mais elles sont exécutées de façon séquentielle comme les autres instructions.

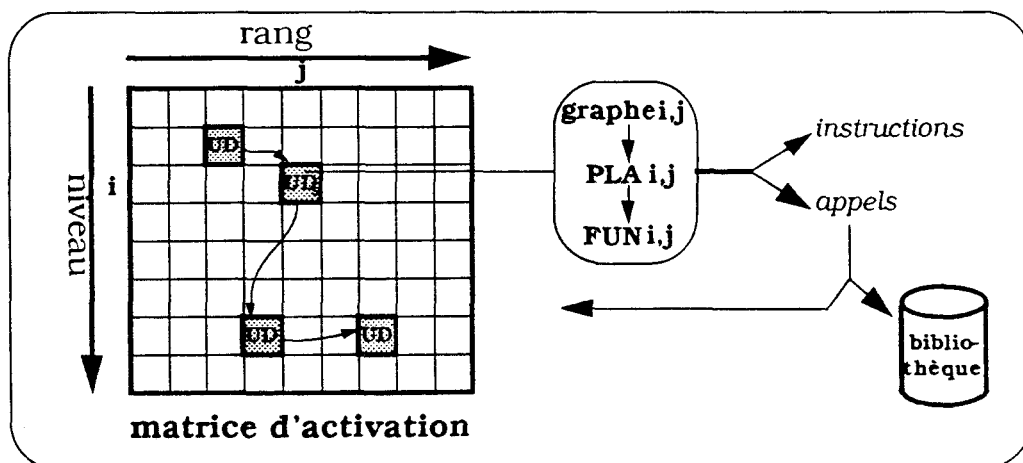


Figure 4.1 Ordre d'activation des Unités de Descriptions

4.1.2 Le système LIDO

Le système LIDO, dont le but est la vérification fonctionnelle de systèmes VLSI au niveau transfert de registre ([BP87]), était initialement composé d'un interpréteur du langage. C'est la méthode de l'interprétation directe, dont les avantages et inconvénients sont maintenant connus, qui a été choisie pour réaliser la simulation.

Ce système, présenté dans la figure 4.2, peut être caractérisé essentiellement par son interface utilisateur et par la partie relative à la réalisation effective de la simulation.

a. L'interface utilisateur

Celui-ci est très convivial et permet d'animer, d'analyser et de tester une description de façon semi-automatique. L'utilisateur peut notamment charger la description d'un circuit, charger des vecteurs de test ou le contenu de porteurs d'information, sélectionner le type d'exécution désiré, choisir le type de visualisation des résultats, procéder à l'exécution, consulter les résultats, modifier éventuellement certaines valeurs avant de continuer le processus ou de le reprendre au début.

Dans la plupart des cas, les entrées peuvent se faire au clavier ou par l'intermédiaire d'un fichier de données et les sorties sont possibles sur écran ou dans un fichier résultat.

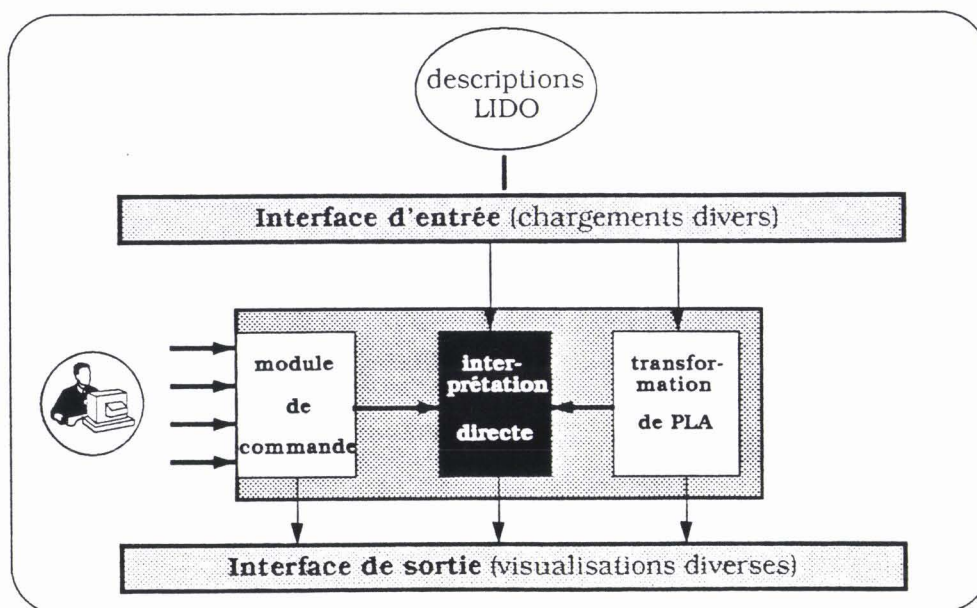


Figure 4.2 Organisation du système LIDO

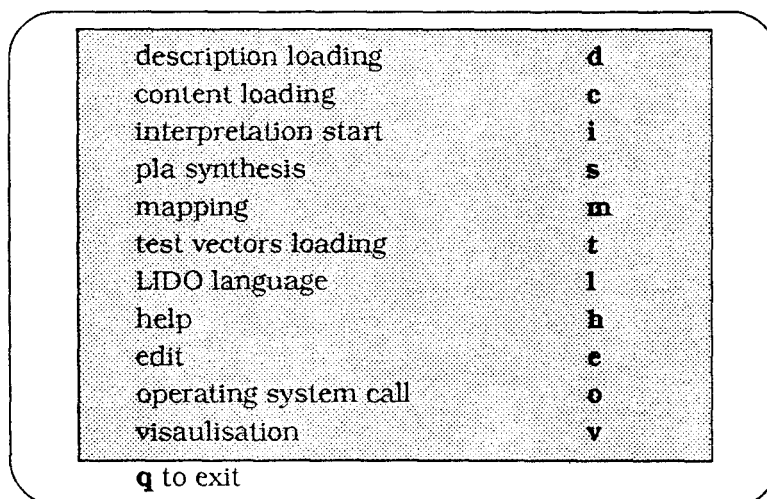
Ce sont surtout le choix du type de simulation et celui des types de trace qui contribuent à la convivialité de l'interface utilisateur et seuls ces choix seront détaillés par la suite. Des informations plus précises sur l'ensemble des commandes du système, dont le menu principal est décrit dans la figure 4.3, peuvent être trouvées dans [Bak89].

Choix du type de simulation:

L'utilisateur a le choix entre l'interprétation continue ou interactive. L'interprétation continue (IC) permet de simuler un système pendant un nombre de phases ou de cycles horloge donné. Les entrées se font dans ce cas en mode batch.

L'interprétation interactive (II) permet de localiser plus précisément les erreurs d'une description en spécifiant des points d'arrêt constitués d'un élément et d'une valeur, l'exécution s'arrêtant lorsque l'élément a la valeur spécifiée. Ainsi, après avoir estimé "temporellement" un mauvais fonctionnement, il est possible de localiser "physiquement" les erreurs qui en sont la cause. Dans le cas de l'II, il est aussi possible d'arrêter la simulation après un temps réel (de simulation) déterminé.

Pour chacun de ses types d'interprétation, tous les types de trace sont autorisés. La modification du contenu de certains éléments est plus facile dans le cas de l'II, car elle peut être effectuée par saisie au clavier alors qu'elle doit l'être par l'intermédiaire de fichiers dans le cas de l'IC.

The image shows a rectangular box with rounded corners containing a list of menu items and their corresponding single-letter shortcuts. The items are listed in two columns, with the shortcuts aligned to the right of each item. At the bottom of the list, there is a separate line for the 'q' shortcut to exit.

description loading	d
content loading	c
interpretation start	i
pla synthesis	s
mapping	m
test vectors loading	t
LIDO language	l
help	h
edit	e
operating system call	o
visaulisation	v
q to exit	

Figure 4.3 Menu principal du système LIDO

Choix du type de trace:

Trois types de trace sont offerts à l'utilisateur, chacun permettant de sélectionner les porteurs et les unités de description concernés. Après la simulation, l'usager peut consulter les données stockées dans des tampons associés à chaque trace choisie s'il a opté pour l'II ou dans un fichier s'il a choisi l'IC. De plus, dans le cas de l'II, les traces peuvent aussi être écrites dans un fichier de sortie.

- La trace numérique mémorise au cours de l'exécution le contenu des éléments sélectionnés. Les valeurs peuvent être données sous forme binaire, octale, décimale ou encore hexadécimale.

- La trace graphique mémorise les informations nécessaires à l'affichage sous forme graphique simplifiée (courbes carrées).

- La trace symbolique mémorise sous forme de texte les opérations exécutées au cours de la simulation. Cette trace est très proche de la description ([Bak88]) et permet de visualiser les chemins actifs de l'architecture.

Les traces numériques et graphiques sont relatives à des éléments tandis que la trace symbolique est relative aux unités d'une description.

b. Réalisation de l'exécution

La simulation d'une description se fait de façon séquentielle et respecte l'ordre d'exécution des unités défini par la matrice d'activation. Les opérations de chaque unité sont exécutées suivant l'ordre d'apparition dans sa description.

En plus de la valeur des variables associées aux différents porteurs d'information d'une description, il est nécessaire de maintenir à jour les informations liées aux traces sélectionnées (cas de l'II) et/ou d'écrire ces renseignements dans un fichier de sortie (cas de l'IC et de l'II).

4.2 L'évolution de l'outil

4.2.1 Version prospective initiale

Cette version est celle obtenue à l'issue du D.E.A. Son but était de vérifier la faisabilité de la méthode choisie et d'évaluer si les espoirs concernant les performances pouvaient être réalisés. Ceci a permis de mettre

en relief des points sensibles à optimiser en priorité et de dégager une philosophie de génération.

La compilation indirecte nécessitait cette phase préliminaire pour se familiariser à cette technique. La principale difficulté de cette méthode réside dans la cohabitation dans un même module du logiciel des variables propres au générateur, de celles liées à la description LIDO, de celles générées dans un simulateur et issues de la description initiale ainsi que de celles communes à tous les simulateurs et aux fonctions run-time.

Ainsi, les caractéristiques principales de cette première version étaient les suivantes:

- Le traitement d'un sous-ensemble du langage LIDO et non de sa totalité.

- L'emploi des mêmes SDs pour les variables de la description LIDO dans le générateur et dans les simulateurs. Toutes les informations requises pour l'analyse étaient donc présentes lors de la simulation ce qui était bien sûr loin d'être optimal.

- L'absence d'identification des architectures cibles traitées. Dans tous les cas, les SDs étaient adaptées aux architectures de 128 bits sans tenir compte de caractéristiques différentes.

- La représentation des types de porteurs d'information sous forme de tableaux, l'accès à un objet se faisant par un indice. Dès lors, le texte d'un simulateur était peu lisible et il devenait très rapidement difficile de savoir quel objet était associé à tel indice d'un tableau¹.

Cette version ayant permis d'obtenir des résultats encourageants, l'outil a évolué jusqu'à sa version actuelle comprenant notamment l'emploi d'une méthode de synthèse de PLA et l'usage de bibliothèques.

4.2.2 La version actuelle

La philosophie de l'outil consiste, de façon générale, à utiliser au maximum les étapes d'analyse et de génération pour réaliser des transformations sur les éléments, unités et représentations d'une description afin de minimiser le temps de simulation. Cela peut se faire au détriment de l'augmentation du temps de production d'un modèle C, la première version ayant conduit à des temps de génération très raisonnables (ne dépassant pas la minute).

1. Sauf pour un véritable bien entendu.

Ainsi, les améliorations apportées à la première version ont permis d'atteindre les caractéristiques suivantes:

- Prise en compte de la presque totalité de LIDO (seules les mémoires associatives ne sont pas traitées).
- Prise en compte des unités structurellement multipliées pour la simulation d'architectures parallèles.
- Prise en compte des caractéristiques de l'architecture cible et adaptation du générateur par la production d'un simulateur d'une "machine virtuelle" équivalente à la machine cible.
- Création et gestion de bibliothèques pour la modularité des descriptions LIDO et des modèles C.
- Définition de la méthode de transformation de partie contrôle pour la réduction de la taille et l'augmentation de la vitesse des modèles C produits.
- Réduction au minimum de la taille des objets manipulés pendant la simulation. De plus, les informations utiles sont codées chaque fois que cela est possible et dans la mesure où cela n'augmente pas de manière significative le temps d'exécution à cause d'un accès trop complexe aux informations.
- Génération des valeurs constantes d'une description plutôt que la mémorisation et l'accès à ces informations lors de la simulation. Les valeurs constantes concernent par exemple, la valeur par défaut ou le nombre de phase(s) de mémorisation(s) d'un terminal, la taille d'un élément ou encore celle d'un mot mémoire.
- Emploi des mêmes noms pour les variables d'un modèle C que ceux spécifiés pour les porteurs d'informations de la description initiale. Ces variables sont donc déclarées en tête d'un programme de simulation et toutes les procédures y font référence par nom, ce qui mène à des simulateurs compréhensibles par tous, dès lors que l'on dispose de la description initiale comme référence.
- Les éventuels conflits de noms entre les variables propres aux simulateurs et celles associées aux porteurs d'information sont évités par l'emploi du caractère souligné comme suffixe des noms de variables d'un simulateur. A l'heure actuelle, les porteurs d'information de la description LIDO sont supposés ne pas comporter le caractère '_' dans leur nom, mais cette vérification est facile à ajouter au générateur.
- Génération automatique de commentaires dans le code C des simulateurs. Ceux-ci concernent le type des variables déclarées ou la phase d'horloge par exemple.

- Usage de macros chaque fois que cela est possible afin d'éviter un appel (coûteux) de procédure. Le corps d'une macro venant remplacer son appel au moment de la compilation du modèle C, ceci n'est possible que pour les fonctions de petite taille car il faut aussi éviter l'expansion de la taille des simulateurs.

- La prévision des ressources (taille et temps CPU) nécessaires pour une simulation. Celle-ci est limitée au cas des parties contrôles et se fait à l'aide d'équations.

- La limitation du nombre de fonctions run-time nécessaires. Les opérations sont effectuées en LIDO sur des objets de types différents ce qui, avec un langage comme C, requiert une procédure par type d'opération (14), par "type d'architecture"¹ cible (3) et par combinaison des types d'opérandes (6x6). Afin de réduire ce nombre, les opérations sont exécutées sur des tampons de même type comme le montre la figure 4.4.a. L'exécution d'une séquence d'opérations est donc la répétition du "cycle d'exécution" (voir figure 4.4.b).

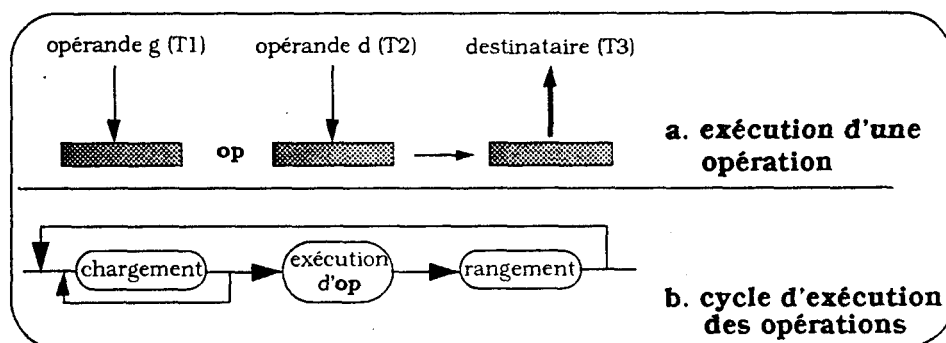


Figure 4.4 Minimisation du nombre de fonctions run-time

4.3 Le code du logiciel

Cette section décrit les SDs et algorithmes utilisés soit par l'outil (partie génération), soit par les simulateurs (partie simulation). L'objectif n'étant pas de détailler le code C employé, seules certaines SDs seront présentées et les algorithmes resteront généraux. Néanmoins, le travail de développement n'est pas négligeable (voir 4.3.3) même s'il demande essentiellement de la méthode et du temps.

La partie relative à la simulation est présentée en premier lieu car la génération est directement dépendante de la manière de simuler une

1. La notion de type n'est ici relative qu'aux différentes tailles de mots machines considérées.

architecture. Le lecteur à l'esprit critique acéré pourra bien sûr arguer de la réciprocité de cette dépendance (la simulation n'est que le reflet de ce qui est généré) mais il est clair qu'en génération, ce qui est important c'est de savoir comment on va simuler un circuit et le générateur produira ensuite le code C qui le simulera.

Enfin, le langage C, langage d'implémentation du générateur et des simulateurs est supposé connu. Il n'est certes pas indispensable de le connaître dans les moindres détails pour suivre cette section mais des rudiments de langages de programmation structurée ainsi que la connaissance des opérateurs de C peuvent aider.

Dans les exemples de code C présentés, les commentaires générés automatiquement par l'outil apparaissent entre `/*` et `*/` alors que ceux ajoutés pour aider à leur compréhension sont encadrés de `--`.

4.3.1 La simulation

Un simulateur exécutable est le résultat de la compilation du modèle C du circuit et de l'édition de liens avec les fonctions run-time et éventuellement des unités déjà compilées et stockées dans une bibliothèque.

Dans l'état actuel de l'outil, l'interface utilisateur d'un simulateur est réduit au minimum, la réalisation de cette partie n'offrant pas vraiment d'intérêt théorique. Les entrées se font donc par l'intermédiaire d'un fichier de données alors que les sorties se font sur l'écran et consistent simplement en l'affichage des variables et de leur contenu après chaque cycle horloge ou à la fin de la simulation. Toutefois, le nombre de transactions intervenant dans une simulation est calculé au fur et à mesure et affiché à la fin de l'exécution. L'emploi d'un débogueur symbolique est une solution pratique pour visualiser le contenu d'éléments pendant un cycle horloge, ce qui est nécessaire dans le cas d'une mauvaise description. De plus, ceci offre l'avantage d'être possible quelque soit l'environnement dans lequel est utilisé un simulateur, ce qui est particulièrement intéressant dans l'optique de l'échange de modèles C.

Chaque simulateur accepte un ou deux paramètres. Le premier est le nombre de cycles horloge pendant lequel durera la simulation. Le second, qui est optionnel, est le fichier de données qui initialise le contenu de certains éléments. Le nombre de cycles horloge doit impérativement être spécifié, que ce soit par l'un ou dans l'autre des paramètres.

Les opérations réalisées lors de la simulation peuvent être classées en

trois catégories. Elles consistent à:

- 1) Initialiser les variables lorsqu'un fichier de données est spécifié.
- 2) a. A chaque cycle et phase d'horloge, exécuter les procédures associées aux UD_s conformément à l'ordre défini par la matrice d'activation.
b. Réinitialiser, si nécessaire, les variables ayant une valeur par défaut après chaque cycle d'horloge.
- 3) Afficher le contenu des variables.

L'affichage des variables peut avoir lieu après chaque cycle, à la fin de la simulation ou ne pas avoir lieu du tout (lorsque l'utilisateur veut simplement mesurer le temps d'exécution). Ceci est choisi au moment de la génération. Ainsi le code correspondant n'est pas toujours généré. De la même manière, lorsqu'aucun registre maître-esclave ou élément ayant une valeur par défaut n'est présent dans une description et que les objets à contenu aléatoire ne sont pas utilisés, le code de la procédure de réinitialisation n'est pas généré.

a. Le code d'un simulateur

Celui-ci est constitué des différentes parties nécessaires à la réalisation des opérations décrites plus haut. Ainsi, un simulateur est typiquement composé des parties suivantes:

- les déclarations,
- le programme principal,
- le corps des procédures des UD_s,
- le corps des procédures d'entrée/sortie.

Les déclarations:

Il faut distinguer les déclarations des variables communes à tous les simulateurs (VC_s) de celles spécifiques à chacun (VS_s). Les premières sont incluses à l'aide d'une instruction *#include* alors que les secondes sont générées pour chaque simulateur.

Les VC_s concernent un certain nombre de variables de travail nécessaires à tout programme C, les variables liées au contrôle de l'exécution (nombre de cycle de simulation), celles des tampons sur lesquels sont effectués les opérations ainsi que la déclaration d'une constante utilisée pour optimiser la réalisation des opérations logiques.

Afin de pouvoir stocker 32, 64 ou 128 bits, un tampon est composé d'un, de deux ou de quatre entiers non-signés de 32 bits chacun. Deux variables de ce type sont utilisées pour charger les opérandes droit et gauche d'une opération (voir figure 4.4).

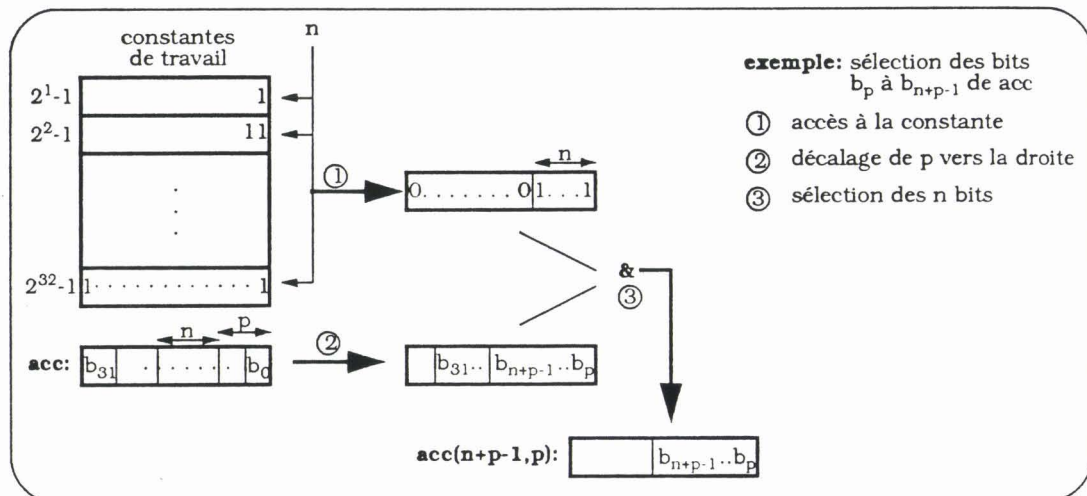


Figure 4.5 Sélection optimisée d'une partie des bits d'une variable

Les opérations logiques ainsi que celles de chargement et de rangement requièrent la sélection de plusieurs bits consécutifs du contenu d'une variable. Une telle sélection est techniquement très facile en langage C et se fait à l'aide de l'opérateur $\&$ (ET logique binaire) et d'un masque dont les bits à un sélectionnent les bits correspondants de la variable. C'est la construction du masque qui nécessite d'être optimisée. La valeur de n bits consécutifs à un étant égale à deux puissance n moins un ($2^n - 1$), il suffit d'obtenir cette valeur pour produire le masque (voir figure 4.5). Il n'existe malheureusement pas d'opérateur puissance en C et calculer cette valeur à l'aide d'une fonction serait extrêmement coûteux. C'est pourquoi toutes les valeurs possibles (de $2^1 - 1$ à $2^{32} - 1$, puisque un simulateur travaille en réalité sur des mots de 32 bits) sont stockées dans un tableau défini comme constante. Dès lors, la connaissance du nombre de bits (n) suffit à construire le masque en accédant simplement à la constante d'indice $n-1$ ¹. Ainsi, toutes les opérations intervenant dans l'accès à une variable sont élémentaires et donc très rapides (cas du chargement d'une variable dans un tampon par exemple, voir figure 4.5).

Les VSs sont directement liées aux porteurs d'information d'une

1. Les tableaux, en C, commencent à l'adresse zéro, est-il besoin de le rappeler?

par les données mais occasionnerait des opérations d'accès (chargement et rangement) plus complexes.

- Emploi de tableaux de un, deux ou quatre ENS pour la déclaration des mémoires et multiplexeurs.

- Définition pour les autres porteurs d'information de deux catégories de VSs. Il est en effet intéressant de distinguer les porteurs à un seul bit (appelés opérandes simples) et les opérandes à plusieurs bits (opérandes longs). L'accès à un opérande long est une opération complexe (voir figure 4.7) ce qui n'est pas le cas avec un opérande simple. De plus, la définition des procédures d'accès pour un opérande long ne dépend pas de sa taille (celle-ci influe uniquement sur le temps d'accès) ce qui permet de ne distinguer qu'une catégorie d'opérande long au lieu de trois (32, 64 et 128 bits).

- Emploi d'un type unique pour les porteurs simples. Un ENS est utilisé de façon différente suivant le type du porteur. En effet, un verrou simple stocke un seul bit alors qu'un registre doit en mémoriser deux (parties maître et esclave). Le terminal est un cas particulier car l'information qu'il mémorise est perdue au bout d'un certain temps. Cette durée est certes une constante mais la date de "réinitialisation" varie au cours de l'exécution, c'est pourquoi il est aussi nécessaire de mémoriser cette information. A cet effet, 16 bits sont utilisés, ce qui permet de simuler un circuit pendant $2^{16}-1$ cycles horloge. Il est aussi possible d'utiliser la totalité des 31 bits restants afin de procéder à de plus longues simulation.

- Une SD de 128 bits est utilisée pour déclarer les opérandes longs quelle que soit leur taille. L'emploi de SDs plus adaptées en terme de taille (comme pour les tampons) conduirait à l'expansion du nombre de procédures d'accès dont l'écriture s'est avérée complexe car il faut tenir compte, en plus de la taille d'une tranche à traiter (de 2 à 128 bits), du chevauchement de cette tranche sur les ENS (voir figure 4.7).

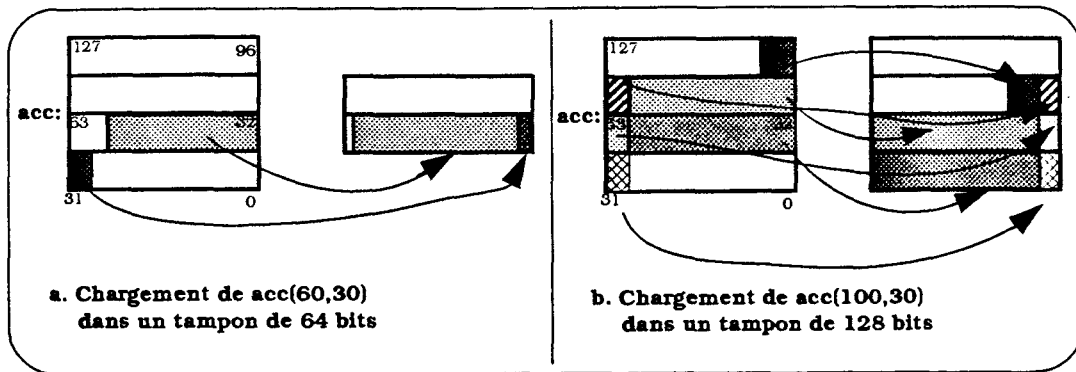


Figure 4.7 Complexité du chargement des opérandes longs

Le programme principal:

Celui-ci dirige l'exécution des trois parties d'un simulateur dégagées au paragraphe 4.3.1. La partie qui réalise effectivement la simulation est décrite dans ce paragraphe, les autres (opérations d'entrée/sortie) le seront ultérieurement.

Une simulation consiste en l'exécution du modèle pendant un nombre de cycles horloge donné (choisi par l'utilisateur). La partie traitement du programme principal est donc une boucle à l'intérieur de laquelle sont appelées les UD's conformément à l'ordre de la matrice d'activation. Les appels de ces unités sont produits suivant l'ordre topologique et implicite pour chaque phase d'horloge. La taille du code C correspondant est donc proportionnelle au nombre de phases d'horloge utilisées dans la description du circuit. Cette taille pourrait être réduite en utilisant une boucle et des tests sur les phases d'horloge mais cela augmenterait le temps de simulation (de deux à quatre pour cent pour les exemples traités). C'est pourquoi cette solution n'a pas été retenue, l'augmentation de la taille du code C étant par ailleurs négligeable par rapport à la part du code des UD's.

Algo 4.1 Structure du programme principal

- lecture éventuelle du fichier de données
- pour chaque cycle d'horloge faire
 - /* appel des UD's actives à la phase 1: */
 - ... appel des UD's dans l'ordre topologique ...
 - ... même chose pour toutes les phases ...
 - réinitialisation des éléments "spéciaux"
- fait
- affichage éventuel des résultats

Exemple: code C pour le cas d'Eg1

```
if (lire) lire_donnees(fichier);
for (i_=0; i_!=nbcycles_; i_++)
```



```
/* phase 1: */
```

```
G1_1(&sup, &count, &rstmin, &rstmax);
```

```
-- une seule unité dans ce cas --
```

```
-- une seule phase dans ce cas --
```

```
afficher_valeurs(); -- l'utilisateur a choisi l'affichage à la fin --
```

Le corps des procédures des UDS:

Le code C des UDS est radicalement différent suivant qu'il s'agit d'un graphe, d'un PLA ou d'une unité fonctionnelle. C'est pourquoi il est nécessaire de les examiner séparément. De plus, il existe plusieurs formes de PLA et un PLA peut être traduit de différentes manières. Enfin, un graphe peut être traduit directement ou transformé en PLA avant la génération.

Le concept de graphe en LIDO a été présenté par les figures 3.6 et 3.7 et aux paragraphes 3.4.3.c et 4.1.1.d. L'exécution d'un graphe consiste à le parcourir en fonction des conditions associées à chaque noeud et à exécuter les actions associées à l'arc ainsi déterminé. La simulation d'un graphe est donc la répétition sans fin d'une suite d'opérations qui mènent d'un noeud vers un autre. Ces opérations sont les suivantes:

- a. évaluer les conditions associées au noeud courant jusqu'à en trouver une qui soit vérifiée (il en existe forcément une par définition),
- b. exécuter les opérations associées à la condition sélectionnée,
- c. mémoriser le nouveau noeud courant (transfert de contrôle).

A chaque fois qu'un graphe est activé, cette suite d'opérations est effectuée puis les autres UDS sont exécutées suivant l'ordre d'activation. L'exécution suspendue d'un graphe reprend au plus tard au cycle horloge suivant et au plus tôt à la phase d'horloge suivante.

La partie b du traitement d'un graphe est en tout point semblable à celle intervenant dans les unités fonctionnelles. Une structure *switch* est employée pour sélectionner les actions (a, b et c) associées à chaque noeud. L'instruction *break* est utilisée pour sortir du *switch* et suspendre ainsi l'exécution du graphe lorsque les actions associées à la condition sélectionnée ont été exécutées.

Dans le programme C d'un graphe, une seule variable est nécessaire pour le parcourir. Il s'agit du numéro du noeud courant, utilisé de préférence à son nom. Comme il peut y avoir plusieurs graphes dans une description, un tableau est utilisé pour les numéros des noeuds courants.

Algo 4.2 Structure employée pour un graphe

suivant noeud courant du graphe faire
 cas noeud 0: -- évaluer les conditions associées au noeud 0 --
 · évaluer la première condition
 · si condition vraie alors
 - exécuter les actions associées
 - suspendre l'exécution du graphe
 fsi
 ... même chose pour les autres conditions ...
 · exécuter les actions associées à la condition par défaut
 ... même chose pour les autres noeuds ...
fait

Un PLA, quelle que soit sa forme (voir 4.1.1.d), peut être traduit de façon algorithmique ou être transformé avant que ne soit produit le code C (cas de la synthèse). La manière de spécifier un PLA en LIDO importe peu et c'est surtout le type de génération employée qui influe sur l'aspect du code C produit. Bien que la partie OU d'un PLA soit très différente d'une forme à l'autre, c'est la manière d'évaluer une matrice ET qui différencie les deux types de traduction de cette structure.

Dans le cas de la traduction algorithmique, toutes les lignes d'une matrice ET doivent être successivement évaluées. Cette évaluation consiste à comparer les entrées d'un PLA aux valeurs de la ligne ET. Lorsque toutes les valeurs correspondent, la ligne OU associée doit être exécutée.

Pour les deux premières formes de PLA (code double et code simple), les sorties reçoivent les valeurs un ou zéro, cette valeur étant obtenue par OU logique sur les bits de toutes les lignes sélectionnées. Dans la troisième forme, des actions doivent être exécutées. Celles-ci sont les mêmes que celles pouvant apparaître dans les FUNs (voir plus loin). Le code C résultant réalise en quelque sorte l'interprétation de la matrice ET et l'exécution de la partie OU.

Afin de stopper l'évaluation d'une ligne ET dès qu'une entrée ne correspond pas à la valeur associée, une boucle et un *break* sont utilisés. De plus, il est nécessaire d'employer un tampon pour chaque sortie sur lequel est réalisé le OU logique. A cause des valeurs *don't care*, il est possible que des sorties ne soient pas affectées par l'exécution d'un PLA. De ce fait, le tampon doit pouvoir contenir une troisième valeur (autre que zéro et un) afin de tester si la sortie doit être affectée ou non. Dès lors, le OU logique ne peut plus utiliser l'opérateur `|` du C car le résultat entre zéro et cette troisième valeur ne donnerait pas zéro. Ainsi, les opérations effectuées suivant les

valeurs de la matrice OU sont les suivantes:

- 1: le tampon reçoit la valeur un (résultat du OU logique),
- 0: le tampon reçoit zéro seulement lorsque sa valeur actuelle est *don't care* (seul cas où sa valeur doit être modifiée),
- *don't care*: aucune action (rien n'est généré).

Algo 4.3 Structure d'un PLA double code (traduction algorithmique)

```
-- évaluation de la première ligne de la matrice ET --
• mettre eval à vraie
• tant que eval vraie faire
  - si entrée un ≠ bit numéro un de la ligne alors
    • mettre eval à faux
    • sortir de l'évaluation de la ligne (break)
  fsi
  ... même chose pour les autres entrées de la ligne ...
fait
• si eval vraie faire      -- exécution de la ligne OU sélectionnée --
  pour toutes les sorties si faire
    effectuer le OU entre le ième bit de la ligne OU et le tampon ti
  fait
fsi
  ... même chose pour les autres lignes de la matrice ET ...
• pour toutes les sorties si faire
  affecter le tampon ti à la sortie si
fait
```

Exemple: C "light"¹ du PLA1,1 (paragraphe 4.1.1.d)

-- évaluation de la première ligne de la matrice ET --

cond_=1;

while (cond_) {

if (up≠1) {cond_=0; **break**;}

if (et_log_large(count)≠1) {cond_=0; **break**;}

break;-- pour sortir quand la ligne est sélectionnée --

}

if (cond_) { -- exécution de la partie OU --

 tampon_rstmin=1;

if (tampon_plus = *don't care*) tampon_plus=0;

if (tampon_moins = *don't care*) tampon_moins=0;

}

 ... même chose pour les autres lignes ...

-- affectation des tampons aux sorties --

if (tampon_rstmin ≠ *don't care*) rstmin=tampon_rstmin;

if (tampon_rstmax ≠ *don't care*) rstmax=tampon_rstmax;

1. Destiné à clarifier les esprits, le code C a été allégé et le détail des accès aux données a été occulté.



```
if (tampon_plus ≠ don't care) plus=tampon_plus;
if (tampon_moins ≠ don't care) moins=tampon_moins;
```

Dans le cas de la traduction après transformation (synthèse), il suffit de décrire la génération des PLAs code double. En effet, la transformation des graphes en PLA code double relève de la génération et les Unités Fonctionnelles sont traduites de la même manière que dans le cas de la traduction algorithmique.

Après l'expansion (voir 3.4.3.c), la matrice ET est complètement spécifiée et il existe une ligne et une seule dont les valeurs correspondent à celles des entrées. Il n'y a donc plus de OU logique à effectuer lors de la simulation et l'usage de tampons devient inutile. De plus, l'ensemble des valeurs des lignes ET correspond à l'ensemble des entiers compris entre 0 et $2^n - 1$ où n est le nombre d'entrées du PLA. Dès lors, il ne reste plus qu'à calculer un entier à partir des entrées pour sélectionner la ligne OU dont les bits doivent être affectés aux sorties.

Il faut toutefois signaler que l'emploi de la synthèse a été restreint aux PLAs dont le nombre d'entrées est inférieur ou égal à trente-deux. Il est nécessaire de modifier les SDs qui mémorisent les lignes OU si l'on veut traiter des cas plus généraux. Néanmoins, l'espace mémoire alors occupé (au minimum un Giga-octets) pourra forcer l'emploi de machines hôtes dont les caractéristiques dépassent celles fixées au paragraphe 3.2.

Algo 4.4 Structure employée pour un PLA double code dans la cas de la synthèse

```
-- calcul de l'adresse ou est stockée la ligne OU: --
· adresse = 0;
· pour toutes les entrées de  $n$  à 1 faire
  - décaler l'adresse à gauche de un bit
  - ajouter la valeur de l'entrée
fait
  -- accès à la ligne OU: --
  · accéder à la donnée stockée à l'adresse calculée
  -- modifier les sorties: --
  · pour chaque sortie  $s_i$  faire
    - accéder au bit  $i$  de la donnée
    - l'affecter à la sortie (traitement différent suivant le type)
fait
```

exemple: C light du PLA1,1

```
-- calcul de l'adresse --
adresse_ <<= 1; adresse_ |= valeur(up);
adresse_ <<= 1; adresse_ |= et_log_large(count);
adresse_ <<= 1; adresse_ |= (count == 0);
-- accès à la donnée --
```

```

donnee = donnees_pla11[adresse];
-- modification des sorties --
rstmin = donnee&1;
rstmax = (donnee >> 1) &1;
plus = (donnee >> 2) &1;
moins = (donnee >>3) &1;

```

Les FUNs sont des UDs beaucoup moins régulières que celles vues jusqu'à maintenant. Elles mettent en jeu toutes les opérations du langage LIDO si l'on excepte le cas du transfert de contrôle, spécifique aux graphes de transitions.

Une FUN est une suite de groupes d'opérations éventuellement gardées par des conditions. L'évaluation de conditions et l'exécution d'opérations obéissent à un même algorithme et seule la taille du résultat (1 bit pour une condition et de 1 à 128 pour une opération) et l'utilisation qui en est faite (test ou affectation) les distingue dans le code C d'une FUN.

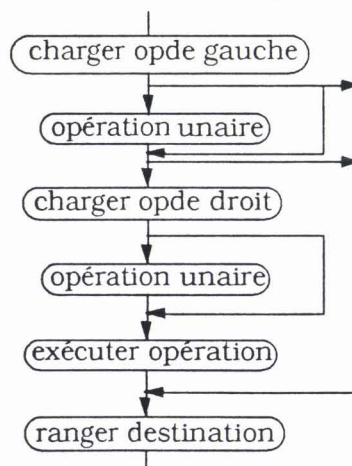
Algo 4.5 Structure utilisée pour les FUNs

```

-- exécution du premier groupe d'instructions --
· [évaluer la condition]
· [si condition vraie alors] exécuter les opérations associées
[fsi]
... même chose pour les autres groupes d'opérations ...

```

évaluation de conditions/opérations:



La majeure partie des fonctions du module précompilé sont utilisées par les FUNs. Il s'agit des procédures de rangement, de chargement et des opérations logiques et arithmétiques. Les opérations sont effectuées à l'aide de deux tampons dans lesquels sont chargés les contenus respectifs des

opérandes droit et gauche. Le résultat d'une opération est toujours affecté au tampon gauche afin d'évaluer une suite d'opérations de façon récursive. Dans le cas des opérations unaires (en largeur ou complémentarité), dont la priorité est supérieure à celle des autres opérations, un même tampon sert comme source et destination. Les procédures employées seront définies plus tard (voir paragraphe b de cette section).

Les procédures d'entrée/sortie:

La lecture du fichier de données se fait essentiellement à l'aide de deux procédures. Le problème posé par cette partie réside dans le fait que les informations sémantiques concernant les variables et nécessaires à la lecture et à la mémorisation des données ne sont pas disponibles lors de la simulation. En effet, seul le générateur possède ces renseignements et peut reconnaître l'appartenance d'une variable à un certain type d'élément et y attacher les caractéristiques du porteur d'information associé. Les données sont spécifiées dans le fichier suivant différents formats selon le type des variables et peuvent être données en base deux, huit, dix ou seize. Pour toutes ces raisons, une première procédure, qui fait partie du module compilé, lit des noms de variables alors qu'une seconde est chargée de les identifier et de lire des valeurs selon le format requis et de les mémoriser. Ce second sous-programme doit être généré pour chaque simulateur car il dépend des éléments de la description LIDO.

L'algorithme de cette seconde procédure recherche chaque variable dans une table (générée elle aussi pour chaque simulateur), puis effectue la lecture et la mémorisation en conséquence. Une structure de sélection (switch-case) est employée afin d'associer un nom au traitement adapté à son type et à ses caractéristiques. La sélection utilise l'indice où se trouve le nom dans la table. Cet indice, connu du générateur, sert à sélectionner le traitement à l'aide d'un case.

Algo 4.6 Structure des procédures de lecture

p1: lecture du fichier	p2: traitement d'un nom
· ouverture du fichier	· chercher le nom dans la table
tg fichier non vide faire	· <u>sui</u>vant indice faire
- lire un nom	- <u>cas</u> celui de var1: lire et stocker données dans var1
- traiter ce nom (p2)	- <u>cas</u> celui de var2: lire et stocker données dans var2
fait	... autres variables de la description ...
	fait

L'affichage des résultats est généré pour chaque simulateur pour les

mêmes raisons que dans le cas de la lecture des données. Il consiste à visualiser pour chaque variable, son nom et son contenu. Le code de cette procédure est toujours généré de la même façon, l'appel de sous-programme étant placé à la fin d'un cycle ou à la fin du programme. Ni l'appel ni le sous-programme ne sont produits lorsque l'utilisateur ne souhaite pas l'affichage des résultats.

Exemple: affichage dans le cas d'Eg1.

```
printf("%s %d \n", "resetmin ", resetmin&1);  
printf("%s %d \n", "resetmax ", resetmax&1);  
printf("%s %d \n", "up ", up&1);  
aff_val(count.valeur, "count", 1, 16);-- affichage en hexadécimal de la valeur du compteur (1 mot de 16 bits) --
```

b. Le simulateur exécutable

L'obtention d'un simulateur exécutable nécessite la compilation du modèle C et l'édition de liens avec le module précompilé. Celui-ci, ainsi que le fichier des macros inclus dans chaque simulateur, définit l'ensemble des fonctions et procédures qui réalisent les opérations arithmétiques et logiques, de rangement et de chargement.

Le nom des procédures comporte trois types d'information nécessaires à la compréhension du code C d'un simulateur. Il s'agit du nom de l'opération, de la taille du(des) tampons(s) sur le(s)quel(s) s'effectue l'opération, et du type d'élément concerné dans le cas des opérations d'accès. Chacune de ces informations est bien sûr réduite à quelques lettres, voir une seule de manière à limiter la taille du code C. Des exemples seront donnés plus loin.

Le module compilé:

Ce sont les procédures d'accès qui sont les plus complexes car elles doivent sélectionner ou ranger de 1 à 32, de 1 à 64 ou de 1 à 128 bits suivant la taille du tampon utilisé. Le problème posé par ces opérations réside dans les chevauchements potentiels des valeurs concernées sur plusieurs ENS (voir figure 4.7). Les procédures de traitement (celles qui réalisent les autres opérations du langage LIDO) sont plus simples car elles agissent sur des tampons dont les valeurs utiles ont été cadrées à droite. Il ne s'agit donc plus que d'exécuter une opération sur un, deux ou quatre ENS.

Algo 4.7 Rangement du contenu d'un tampon de 32 bits

```

-- rangement d'au plus 32 bits du tampon dans un élément long (au plus 128 bits) --
-- stockage du contenu aux adresses debut à fin (par ex. => acc(debut, fin)) --
· si fin ≤ 31 alors -- ( les bits d'un ENS sont numérotés de 0 à 31) --
  si debut ≤ 31 alors - ranger dans le premier ENS (pas de chevauchement)
  sinon -- rangement dans les deux premiers ENS --
    - ranger une partie du contenu dans le premier ENS
    - ranger l'autre partie dans le deuxième ENS
  fsi
sinon
  si fin ≤ 63 alors
    si debut ≤ 63 alors - ranger dans le deuxième ENS (pas de chevauchement)
    sinon -- rangement dans les deuxième et troisième ENS --
      - ranger une partie du contenu dans le deuxième ENS
      - ranger l'autre partie dans le troisième ENS
    fsi
  ... même chose pour les deux autres ENS --

```

Les actions élémentaires de l'algorithme précédent, telles que le rangement dans un ENS ou la sélection d'une partie d'un ENS du tampon, découlent immédiatement de la figure 4.5 et ne seront donc pas présentées plus en détail. D'autre part, étant donné le nombre de procédures du module compilé (76), seul l'algorithme précédent sera donné (voir néanmoins les macros).

Les macros:

Il n'y a pas de différence fondamentale entre les opérations réalisées à l'aide de macros et celles définies dans le module objet. Les macros réalisent les opérations les moins complexes et qui peuvent être écrites de façon compacte. Ainsi, toutes les macros définies occupent une seule ligne de code C et 29 des 33 macros opèrent sur des tampons de 32 bits.

Trente-trois macros ont été définies qui concernent des opérations effectuées sur au plus trente-deux bits. Elles s'appliquent donc aux tampons de trente-deux bits, sauf dans le cas du rangement d'un élément simple (un seul bit) qui peut être effectué sur un, deux ou quatre ENS (voir exemples).

Exemples: code C de quelques macros

```

#define et1_() (*infog1_ &= *infod1_)          -- ET logique sur un ENS --
#define sup1_() (( *infod1_ < infog1_)? 1 : 0)  -- supérieur à entre deux ENS --
#define rg_rs_4(var) (var= (var&2) | (*infog4_&1))  -- rangement d'un bit dans un registre simple --

```

c. L'exécution

La génération a produit un modèle C et un fichier des données de(s) PLA(s) lorsque la synthèse a été utilisée et que l'utilisateur a choisi de générer les données à l'extérieur du simulateur.

La compilation a produit un simulateur exécutable. Une commande a été définie afin de rendre transparente la phase de compilation pour l'utilisateur. Elle effectue notamment l'édition de liens avec les deux bibliothèques (celle des UD appelée *bibi* et celle des sous-programmes écrits en C appelée *biblio*).

Exemple: Code de la commande de compilation

```
cc -Lgenlib $1.c -lbibi -lbiblio -o $1.s -O -s  
chmod 751 $1.s
```

Dans cette commande, le nom du simulateur est représenté par \$1.c, \$1 étant le paramètre passé à la commande, qui est le nom du fichier contenant la description du circuit. Pour un circuit dont la description est stockée dans le fichier risc.l, le code C du simulateur est mémorisé dans le fichier risc.l.c, et le simulateur exécutable est contenu dans le fichier risc.l.s.

Si le générateur a produit un simulateur dédié à un programme, celui-ci requiert un seul paramètre qui est le nombre de cycles de simulation. Dans le cas d'un simulateur dédié à une architecture, deux paramètres sont autorisés dont un au moins est obligatoire. Il s'agit du nombre de cycles de simulation. Celui-ci peut être passé directement en paramètre au simulateur ou spécifié dans le second paramètre qui est le fichier des données.

exemple: lancement de la simulation de l'architecture "risc".

```
risc.l.s -10000          -- 10000 cycles de simulation --  
risc.l.s -risc.m -10000  -- données contenues dans risc.m, 10000 cycles de simulation --
```

4.3.2 La génération

Tout outil de traduction peut être théoriquement divisé en deux parties:

- Une partie indépendante du langage cible appelée *front end* qui construit un modèle interne du circuit. Cette partie comporte notamment les phases d'analyses lexicale, syntaxique et sémantique.
- Une partie indépendante du HDL utilisé appelée *back end* et qui produit le modèle cible.

Notre générateur n'échappe pas à cette décomposition théorique à

laquelle il faut toutefois apporter quelques précisions liées à ce cas particulier.

Tout d'abord, il serait illusoire de vouloir considérer en pratique ces deux parties comme totalement indépendantes l'une de l'autre sous peine de réduire l'efficacité du générateur et des modèles C produits. Le modèle interne, qui dépend nécessairement du langage source (HDL), doit donc tenir compte des caractéristiques du langage cible choisi. En fait, l'intérêt de considérer ces deux parties de façon indépendante, qui réside dans le fait de n'avoir à redéfinir qu'une seule partie si l'on veut traiter le cas d'autres langages sources ou cibles, trouve surtout son importance dans le cas d'une famille de langages, ce qui n'est pas le cas avec LIDO.

Ensuite, bien que la construction du modèle comporte toujours des étapes de transformations des SDs internes, il faut considérer la synthèse comme une partie à part entière étant donnée son influence sur la production du modèle C. C'est pourquoi la génération sera présentée en trois phases en relation avec la figure 3.1.

- la construction du modèle interne (analyse),
- les transformations (synthèse),
- la production du code C (génération).

a. L'analyse

L'analyse qui est effectuée n'est pas une analyse classique en ce sens qu'elle ne recherche pas toutes les erreurs d'une description en un seul passage comme c'est le cas dans un compilateur. En fait, une des hypothèses émises au début du projet supposait que toute description fournie au générateur soit préalablement vérifiée à l'aide de l'interpréteur. Dans la première version de l'outil, un certain nombre de vérifications indispensables à son fonctionnement étaient néanmoins effectuées et un message, certes peu détaillé, était envoyé à l'utilisateur avant d'interrompre l'analyse ou la production du code C.

Dans la mesure où des parties d'une description peuvent ne pas être analysées par l'interpréteur au cours d'une simulation, il n'est pas possible de garantir qu'une description soit correcte, l'interpréteur ne permettant pas de vérifier si tous les chemins ont été parcourus. C'est pourquoi, au fur et à mesure du développement de l'outil, des messages d'erreurs plus précis ont été générés et tous les cas d'erreurs potentiels sont maintenant examinés.

Ni le générateur d'analyseur syntaxique LEX, ni le compilateur de

compilateur YACC n'ont été utilisés pour réaliser le générateur car ils sont apparus mal adaptés aux multiples étapes d'analyse et de transformations nécessaires avant la production d'un modèle C.

A cause de la particularité de LIDO qui fait que les déclarations peuvent être placées n'importe où dans une description, un premier passage est nécessaire pour reconnaître les identificateurs avant de pouvoir analyser les UD's. L'usage de retours en arrière serait très complexe, surtout si l'on considère qu'un sous-élément peut être utilisé avant d'être déclaré et que sa déclaration peut elle aussi apparaître avant celle de l'élément dont il est une partie (dorénavant appelé élément père).

En fait, le nombre de passages importe peu car une description LIDO est toujours de faible taille (quelques Koctets). De plus, chaque bloc d'une description étant repérés par une lettre capitale et terminé par un point, il suffit de scruter les premiers caractères des lignes pour sélectionner un bloc ou le laisser de côté. Ainsi, l'analyse comporte quatre phases, présentées par l'algorithme 4.8.

Algo 4.8 Les phases de l'analyse

- 1) lecture et mémorisation de la description dans une SD
(avec élimination des commentaires, espace, tabulations,...)
- 2) analyse des déclarations (sauf des sous-éléments)
- 3) analyse des sous-éléments
- 4) analyse des UD's

L'étape 2 de l'algorithme précédent traite aussi le cas des équivalences bien que celles-ci soient utilisées et mémorisées comme des sous-éléments. Ceci est possible puisqu'elles apparaissent toujours après la déclaration de l'élément père.

L'analyse des déclarations:

Les SDs utilisées pour stocker les informations relatives aux déclarations doivent permettre par la suite d'associer un type et des caractéristiques à un identificateur. Plutôt qu'une unique table des symboles (notée TdS), c'est quatorze qui ont été définies afin d'économiser l'espace mémoire occupé, chaque TdS mémorisant le minimum d'informations requises par son type. Ceci offre les avantages supplémentaires de ne pas avoir à mémoriser le type d'un élément (implicitement associé à la TdS) et surtout de réduire le temps de recherche d'un élément en commençant par les types statistiquement les plus utilisés.

Ces derniers sont les signaux de contrôle des graphes et PLAs. C'est pourquoi il est apparu judicieux de distinguer aussi lors de la génération les éléments simples (un seul bit) des autres porteurs d'information. Ainsi, les TdS employées sont les suivantes:

- quatre TdSs pour chaque type d'élément simple (registres, verrous, terminaux et terminaux d'entrée),
- quatre TdSs pour chaque type d'élément long (les mêmes),
- deux TdSs pour les multiplexeurs et mémoires,
- une TdS pour les mémoires réorganisées (différents types d'adressage),
- une TdS pour les sous-éléments,
- deux TdSs pour les appels d'UDs internes (déclarées dans la description) et externes (stockées dans une bibliothèque).

De plus, toujours dans le souci de minimiser l'espace mémoire occupé, les TdSs sont allouées de façon dynamique. A cet effet, un passage supplémentaire a été ajouté entre les phases un et deux de l'algorithme 4.8. Celui-ci consiste simplement à répertorier le nombre d'éléments de chaque type et à réserver physiquement l'espace mémoire pour chacune des TdSs.

Les SDs utilisées pour les terminaux longs, pour les sous-éléments et pour les appels d'unités sont présentées ci-après afin de donner un aperçu des informations à gérer.

Exemple: quelques structures de données.

```
-- terminal long --
struct { unsigned duree: 16;      /* duree d'activation */
        unsigned vald: 1;        /* valeur par default */
        unsigned card: 8;        /* taille de l'element */
        char nom[lgmax];         /* nom de l'element */
    }

-- sous-élément --
struct { char nomelt[lgmax];      /* nom de l'element pere */
        char nomsselt[lgmax];    /* nom du sous-element */
        char type;               /* type du pere, (type reel du sous-element) */
        char debut;              /* debut de la tranche de l'element */
        char fin;                /* fin de la tranche, -1 lorsque tranche = un seul bit */
    }

-- unité appelée --
struct { char nomfonct[lgmax];    /* nom de l'unite */
        int cond: 1;             /* indique s'il existe une condition logique d'activation */
        int multiple: 10;        /* facteur de multiplication structurelle de l'unite */
        int avant: 1;            /* indique si la multiplication concerne la condition */
    }
```

```

    int nbvar: 8;           /* nombre de parametres de l'unite */
    int nbvar_con: 8;       /* nombre de parametres de la condition */
    un_var lvar[maxvar];    /* liste des parametres de l'unite */
    un_var lvarc[maxvar];   /* liste des parametres de la condition associee */
}
struct { char nom[lgmax];   /* nom du parametre */
        char mode;         /* passage d'adresse ou de valeur (appels imbriqués) */
        char type;         /* type de la variable, (verification de type) */
        char nbbits;       /* taille du parametre (a verifier) */
    } un_var;

```

Algo 4.9 L'analyse des déclarations

```

· etat=aucun, ligne=première ligne de la description
· tg reste une ligne à traiter faire
  suivant etat faire
    cas aucun: suivant premier caractère de la ligne faire
      cas 'T': · identifier la valeur par défaut et la durée de mémorisation
                · lire et mémoriser les identificateurs et leur taille
                · si dernier caractère de la ligne ≠ '.' alors etat='T'; fsi
      cas 'L': lire et mémoriser les verrous et leur taille
                · si dernier caractère ≠ '.' alors etat='L'; fsi
                ... même chose pour 'R', 'I', 'X', 'M' ...
      cas 'S' ou 'P' ou 'F' ou 'G' ou 'C': mettre à jour etat
    fait
  cas 'T': · lire et mémoriser les terminaux et leur taille
            · si dernier caractère de la ligne = '.' alors etat=aucun; fsi
            ... même chose pour 'L', 'I', 'R', 'X', 'M' ...
  cas 'S' ou 'P' ou 'F' ou 'G' ou 'C': mettre à jour etat
fait
fait

```

La mémorisation des identificateurs (algorithme précédent) sous-entend la vérification préalable qu'il ne s'agit pas d'une double déclaration. Une procédure de recherche dans les TdSs est utilisée à cet effet.

Analyse des sous-éléments:

L'algorithme employé pour ce traitement parcourt le fichier, et mémorise les informations relatives aux sous-éléments dans la SD présentée dans l'exemple antérieur.

Algo 4.10 L'analyse des sous-éléments

```

• etat=aucun, ligne=première ligne de la description
• tg reste une ligne à traiter faire
  suivant etat faire
    cas aucun: suivant premier caractère de la ligne faire
      cas 'S': • lire un élément père (nom et tranche)
        • lire le nom du sous-élément
        • rechercher le type de l'élément père
        • mémoriser les informations
        • si dernier caractère de la ligne ≠ '.' alors etat='S'; fsi
      cas autre: mettre à jour etat
    fait
  cas 'S': • même traitement que 'S' précédent
    • si dernier caractère de la ligne = '.' alors etat=aucun; fsi
  fait
fait

```

La mémorisation des sous-éléments suit le même principe que pour les autres types et ce sont simplement les informations stockées qui diffèrent. Le cas des sous-éléments multiples conduit à introduire plusieurs éléments identiques, ce qui est aussi le cas pour les éléments multiples des autres types.

L'analyse des UD's:

Elle consiste à mémoriser les informations relatives aux UD's mais le nombre de ces informations rend cette partie de l'analyse beaucoup plus complexe que dans le cas des déclarations.

L'analyse d'une UD comporte deux parties distinctes qui sont l'analyse de son en-tête et celle de son corps. L'analyse de l'en-tête est commune aux graphes, PLAs et FUNs. Elle comporte en plus, dans le cas des PLAs, la lecture des entrées et éventuellement celle des sorties (PLA code double).

Les informations associées à un en-tête d'UD sont les suivantes:

- le niveau et le rang (pour l'ordre d'activation)
- le nom de l'UD (pour le nom de la procédure C)
- les phases d'horloge auxquelles l'UD sera activée
- la condition logique d'activation (si présente, pour l'activation)
- le facteur de multiplication structurel (si présent)
- le champ d'application de ce facteur (la condition est elle concernée?)
- les entrées et les sorties dans le cas d'un PLA/décodeur

L'analyse du corps d'une UD est spécifique à chacune d'entre elles mais diffère relativement peu d'une UD à une autre. En fait, les informations mémorisées pour chaque type d'UD sont très proches mais l'analyse diffère à cause de la différence de syntaxe. Les seules différences concernant les informations effectivement mémorisées sont relatives aux entrées et sorties des PLAs et aux noeuds des graphes.

Les informations du corps d'une UD et mémorisées sont les suivantes:

- le corps (utilisé lors de la production du code C)
- les éléments utilisés (paramètres de la procédure C)
- les noeuds (dans le cas d'un graphe)

b. Les transformations

Les algorithmes intervenant dans les étapes de la synthèse, présentés au chapitre trois de façon très générale, sont détaillés dans cette partie. Se reporter à ce chapitre avant d'aller plus loin, pourra être bénéfique au lecteur dont la quantité de travail est telle qu'il a dû interrompre sa lecture passionnée avant d'en arriver à la présente page.

Chacunes des étapes transforme une description en une autre. Des fichiers intermédiaires sont donc générés au cours de la synthèse et détruits à la fin.

Paradoxalement, la synthèse intervient avant l'analyse alors qu'aucune SD n'est encore initialisée. Il est apparu plus simple d'analyser une description lorsqu'elle est définitive que de la modifier après l'analyse en créant de nouveaux éléments et de nouvelles unités et un nouvel ordre d'activation. Cette solution est vraisemblablement plus coûteuse en temps de génération, mais l'optimisation de la génération n'est pas un problème majeur (voir chapitre cinq pour les temps de génération).

La séparation du contrôle et des données:

La partie contrôle d'un graphe est constituée des noeuds, des conditions, des affectations de signaux et de transferts de contrôle. La partie donnée est pour sa part, constituée d'opérations telles que celles apparaissant dans les FUNs. Elles sont précédées, dans un graphe, du caractère ':'.

Algo 4.11 Séparation du contrôle et des données dans un graphe

```
• car = premier caractère du graphe
• tq car ≠ '.' faire
  - lire et recopier car jusqu'à car = '.' ou car = ':'
  - si car = ':' alors
    • lire la partie donnée (jusqu'à ">")
    • la mémoriser dans un tampon
    • créer un drapeau, y associer le tampon
    • copier le drapeau à la place de la partie donnée
  fsi
fait
• si une partie donnée a été trouvée alors
  - créer une unité fonctionnelle, indiquer s'il faudra renommer les unités
  - pour chaque drapeau créé faire
    • générer la condition ( /drapeau/ )
    • copier la donnée associée
  fait
fsi
```

L'unité fonctionnelle créée doit être activée aux mêmes conditions que le graphe dont elle est issue. Elle aura donc même rang et même niveau et sera associée aux mêmes phases d'horloge et à la même condition logique d'activation. Dans la mesure où la création d'une FUN peut modifier l'ordre implicite d'activation, il est parfois nécessaire de renommer les unités qui existent déjà.

Transformation des structures de contrôle en PLACS:

La première étape de cette transformation est l'optimisation des graphes. L'algorithme utilisé est tiré de [Fou87].

L'étape suivante est la transformation d'un graphe en un PLACS. L'algorithme qui réalise cette étape est donné ci-dessous. Le lecteur pourra se reporter au chapitre 3 pour se remémorer les grandes lignes de cet algorithme.

Algo 4.12 Transformation d'un graphe en un PLACS

- déclarer un registre d'état et les tr sous-éléments de ce registre
- renommer les unités s'il existe déjà un PLA de même niveau et rang que le graphe
- créer un nouveau PLA de même niveau et rang que le graphe
- ajouter à l'en-tête de ce PLA:
 - les conditions du graphe et les sous-éléments du registre en entrée du PLA
 - les drapeaux du graphe et les sous-éléments du registre en sortie du PLA
- remplir les matrices du PLA de la façon suivante:
 - **pour** chaque arc du graphe **faire**
 - matrice ET:
 - ajouter les tr bits du code du noeud de départ
 - **pour** toutes les variables de la réunion des conditions du graphe **faire**
 - si** la variable n'intervient pas dans la condition de l'arc **alors** ajouter 'x'
 - sinon si** la variable est complémentée dans la condition **alors** ajouter '0'
 - sinon** ajouter '1'
 - fsi**
 - fait**
 - matrice OU:
 - ajouter les tr bits du code du noeud d'arrivée
 - **pour** toutes les variables de la réunion des drapeaux du graphe **faire**
 - **si** la variable est complémentée ou n'apparaît pas **alors** ajouter '0'
 - sinon** ajouter '1'
 - fsi**
 - fait**

Expansion des PLAs:

Cette étape consiste simplement à remplacer les valeurs 'x' de chaque ligne d'un PLA par '1' et '0' successivement. Chaque remplacement produit donc deux lignes, le OU logique étant effectué dès maintenant. Appliqué de façon récursive à toutes les valeurs de la matrice ET, ce processus conduit à la détermination de toutes les fonctions du PLA.

Exemple:

avant expansion:					après expansion:				
e_1	e_2	s_1	s_2	s_3	e_1	e_2	s_1	s_2	s_3
0	x	1	0	1	0	0	1	0	1
x	1	0	1	1	0	1	1	1	1
					1	1	0	1	1

Les deux dernières étapes, qui ne présentent pas de difficultés du point de vue algorithmique, ne seront pas détaillées et seul l'algorithme relatif à l'expansion est présenté ci-dessous.

Algo 4.13 Expansion des PLAs

```

.initialiser à zéro la nouvelle matrice OU NM_ou
• pour chaque ligne  $l_i$  du PLA faire
  - no_car = 0
  - eclater_la_ligne( $l_i$ , no_car):
    si le caractère no_car vaut '0' ou '1' alors eclater_la_ligne( $l_i$ , no_car+1)
    sinon si vaut 'x' alors
      • le remplacer par '0' dans  $l_{i0}$ 
      • eclater_la_ligne( $l_{i0}$ , no_car+1)
      • le remplacer par '1' dans  $l_{i1}$ 
      • eclater_la_ligne( $l_{i1}$ , no_car+1)
    sinon /* fin de ligne: effectuer le OU logique */
      • extraire la partie ET de  $l_i$  et calculer sa valeur v_et
      • extraire la partie OU de  $l_i$  et calculer sa valeur v_ou
      • OU logique:  $NM\_ou[v\_et] \leq v\_ou \mid NM\_ou[v\_et]$ 
    fsi
  fait

```

c. La production du code C

Cette étape ne pose pas de problème majeur. Elle consiste simplement à produire le code C qui, après compilation, réalisera l'exécution des algorithmes présentés à la section 4.3.1.a.

Les seules parties délicates concernent:

- la génération du code pour l'évaluation d'une suite d'opérations logiques ou arithmétiques,
- la création et la gestion des noms de variables, notamment dans le cas de blocs structurellement multipliés,
- la production d'un code, si possible indenté, dont la taille des lignes respecte les limites fixées par le compilateur.

La première partie relève de solutions algorithmiques tandis que les deux autres nécessitent surtout une bonne méthode.

L'évaluation d'une suite d'opérations arithmétiques est un problème classique dont la solution réside dans l'utilisation d'une pile d'opérateurs et une pile d'opérandes. Dans le cas de la génération, la difficulté se trouve accrue car il faut produire le code et non l'exécuter, ce qui requiert parfois un enchaînement différent des opérations.

Cette partie pourrait être fortement optimisée dans le cas de l'évaluation de conditions booléennes en utilisant deux parcours pour

chaque expression:

- le premier détermine le type de l'expression (forme disjonctive ou conjonctive),
- le second produit le code C en stoppant l'exécution dès que possible, c'est-à-dire lorsque la valeur courante d'un monôme d'une expression disjonctive est 1 ou lorsque la valeur courante d'un monôme d'une expression conjonctive est 0.

4.3.3 Quelques chiffres

Le développement de l'outil a nécessité de nombreuses heures de travail comme en témoignent les quelques chiffres suivant:

	nombre de lignes de code C	taille des compilés
- synthèse:	2680	106K
- analyse:	2000	100K
- génération:	3640	210K
- run-time:	1800	50K

4.4 Conclusion

Les principaux algorithmes et SDs utilisés dans le générateur et dans les simulateurs ont été présentés. Ceux-ci ont été dictés à la fois par les objectifs fixés, par les caractéristiques du langage LIDO et par la nécessité de gérer les compromis dégagés au chapitre précédent.

Bien qu'il existe des optimisations importantes à effectuer (notamment celle de l'évaluation des conditions booléennes), l'outil est proche de sa maturité comme vont le montrer les résultats du chapitre 5.

Chapitre 5

Evaluations de l'outil

Ce chapitre a pour objectif l'évaluation de l'outil, c'est-à-dire la comparaison de ses performances avec celles d'autres logiciels du même type, ainsi que l'évaluation comparative des différentes méthodes employées, notamment la traduction algorithmique avec la synthèse. Une part importante des résultats sera constituée de la prévision de ceux-ci à l'aide d'équations mises au point grâce à une technique inspirée de l'analyse de régression.

Un consommateur Américain achètera-t-il une Toyota Subaru¹ simplement parce qu'une publicité comparative vante les mérites de sa suspension toutes les vingt minutes pendant le film de 20h sur C.B.S.²? Ce serait en tout cas une erreur que d'occulter les autres paramètres qui font la qualité d'une automobile. Il en est de même concernant les logiciels, notamment les outils de simulation.

Comment comparer des outils et méthodes de simulation dont les origines et les objectifs sont différents? Quelles sont les caractéristiques des simulateurs utiles à leur comparaison? Des réponses seront apportées à ces questions dans les sections suivantes, la prise en compte de toutes les caractéristiques d'un outil étant nécessaire à sa comparaison avec d'autres.

5.1 Contraintes d'évaluation

Les lecteurs d'Auto Magazine auront sans doute remarquer que les essais comparatifs effectués par un spécialiste sur un ensemble de véhicules concernent des automobiles de même catégorie. Une Renault 16 TL ne sera donc pas comparée à une Peugeot 205 GTI. De même, il serait stupide de comparer une Formule 1 à une Renault 4 sur une spéciale du Paris-Dakar, sous peine de conclure à la supériorité de cette dernière³.

Ces remarques s'appliquent aussi aux simulateurs et les comparaisons devront utiliser des descriptions semblables, c'est-à-dire caractérisées par:

- le même niveau d'abstraction,
- le même domaine (structurel/comportemental),
- le même modèle temporel.

1. Prononcer "Soubewrou"

2. Sauf pendant la guerre du Golfe, période durant laquelle il regarde C.N.N.

3. La F1 s'enliserait au bout de quelques mètres alors que la Renault a déjà remporté cette épreuve!

Chacune de ces caractéristiques influe de façon notable sur les résultats des comparaisons.

Cela n'est malheureusement pas toujours facile car les HDLs associés aux simulateurs que l'on veut comparer peuvent être très différents. Ainsi, les descriptions doivent être écrites à des niveaux d'abstraction couverts par les deux langages, ce qui peut avantager un des simulateurs. Afin d'éviter ce type de problème, il est nécessaire d'effectuer des comparaisons à plusieurs niveaux et dans chaque domaine.

Un facteur à ne pas négliger dans la comparaison de plusieurs simulateurs est la performance de leur machine hôte. Celle-ci est mesurée en MIPS et en MFLOPS¹. Afin de pouvoir comparer des simulateurs disponibles uniquement sur des machines différentes, il faut ramener les temps de simulation obtenus à une machine de un MIPS. On parle de normalisation du temps d'exécution et de temps normalisé. Il n'a pas été tenu compte du MFLOPS d'une machine car les simulations logiques ne font pas intervenir de nombres réels, sauf peut-être dans des cas particuliers comme par exemple la description fonctionnelle d'un multiplieur.

La comparaison des temps normalisés permet un classement fiable des simulateurs. Pourtant, des écarts faibles mais néanmoins sensibles peuvent être enregistrés entre les temps prévisibles (grâce à la donnée des MIPS) d'exécution sur une machine hôte A à partir de la simulation sur une machine B et les temps réellement obtenus sur la machine A. Ceci est dû à un certain nombre de facteurs dont les deux plus significatifs sont:

- le degré d'optimisation du compilateur utilisé (dans le cas de simulateurs générés),
- la dépendance des MIPS vis-à-vis du jeu d'instructions qui rend délicate la comparaison entre des programmes exécutés sur des machines ayant des jeux d'instructions différents (même à MIPS égaux). Une discussion plus détaillée de ce problème peut être trouvée dans [PH90].

Il est donc préférable de comparer des simulateurs sur la même machine lorsque que cela est possible.

1. Million d'opérations en virgule flottante par seconde.

5.2 Les critères d'évaluation

Tout comme une voiture de tourisme peut être caractérisée par ses performances, sa consommation et son équipement, un simulateur est caractérisé par:

- ses performances (vitesse de simulation),
- sa consommation (espace mémoire nécessaire),
- le temps d'obtention d'un simulateur dans le cas de générateurs,
- son interface utilisateur,
- son étendue d'utilisation (niveaux couverts).

Il est clair que le dernier paramètre est directement lié au langage de description utilisé plutôt qu'au simulateur, même s'il a une influence sur son efficacité.

La qualité de l'interface utilisateur est une notion plus subjective. De plus, les simulateurs générés par l'outil ne disposent que de fonctions rudimentaires et l'emploi d'un débogueur symbolique a été suggéré.

Ainsi, seules les trois premières caractéristiques, objectives et mesurables, interviendront dans les comparaisons effectuées.

La mesure de l'espace mémoire nécessaire à un simulateur ne pose pas de problèmes particuliers. Il n'en est pas de même en ce qui concerne les performances. A l'inverse d'autres disciplines, il n'existe pas une unique mesure des performances. En effet, certains utilisent des unités aussi diverses que:

- le nombre de transactions par seconde,
- le nombre d'événements par seconde,
- le temps d'exécution.

D'une part, cela rend difficile la comparaison de l'outil avec d'autres à partir de résultats publiés dans la littérature et d'autre part, les deux premières unités sont criticables comme nous l'allons voir.

Une transaction n'est pas forcément représentative de ce qui se passe dans le circuit. En particulier dans le cas des simulateurs générés, plusieurs variables peuvent être utilisées pour un élément du circuit, conduisant à plusieurs transactions alors qu'une seule aurait lieu dans le circuit.

Plusieurs exemples ont révélé clairement que les NTs par seconde, fortement liés au HDL utilisé, pouvaient conduire à classer des simulateurs dans un ordre inverse au temps d'exécution.

Facile à calculer, le NT d'une simulation est donné par de nombreux simulateurs. C'est le cas du simulateur VHDL d'Intermetrics Inc. et des simulateurs générés par l'outil proposé.

Un événement dans un simulateur est lié de façon plus forte à ceux des circuits simulés que ne l'est une transaction, dans la mesure où tout événement implique une modification d'un élément qui peut être visible par l'utilisateur. Néanmoins, deux problèmes se posent quant au choix du NE par seconde pour la mesure des performances des simulateurs.

Le premier problème est lié au calcul effectif du NE d'une simulation. Déterminer cette quantité doit se faire lors de la simulation et implique le test de chaque élément qui voit une transaction afin de savoir si celle-ci produit un changement de l'élément ou pas. Ce calcul allongerait le temps de simulation du circuit de façon importante, c'est pourquoi le NE d'une exécution n'est pas fourni par les logiciels de simulation. L'estimation statique du NE d'une simulation serait donc une meilleure solution, d'autant plus qu'elle permettrait de ne prendre en compte que les événements réels d'un circuit. Si cela est possible pour des circuits assez simples et à des niveaux d'abstraction élevés (par exemple dans le cas d'une description fonctionnelle d'un compteur 16 bits), Il devient extrêmement complexe d'estimer le NE pour de plus larges circuits tels qu'une UAL ou un processeur. Ce problème se trouve encore accru lorsque le NE dépend des vecteurs de test fournis au simulateur et une estimation statique devient alors impossible.

Le deuxième problème est dû au fait que les événements lors d'une simulation sont liés à la description du circuit et donc au HDL utilisé. Ce problème est inévitable quel que soit l'unité prise en compte pour l'évaluation des simulateurs et le premier problème est donc la raison majeure pour laquelle le NE n'a pas été utilisé.

C'est donc le temps d'exécution qui a été retenu comme unité de comparaison des performances. Plus exactement, c'est le temps CPU d'exécution utilisé par le simulateur qui est répertorié pour chaque exécution. En effet, le temps total d'exécution, c'est-à-dire le temps écoulé entre le début et la fin d'une simulation, n'est pas le reflet du temps réellement passé à simuler un modèle. Le temps écoulé comprend plusieurs

parties:

- le temps CPU d'exécution, lui-même divisé en deux parties:
 - le temps CPU utilisé par le simulateur,
 - le temps CPU utilisé par le système d'exploitation
- le temps d'attente en entrée/sortie pendant lequel les ressources CPU sont utilisées par d'autres programmes.

C'est donc la partie *user* renvoyée par la commande *time* des systèmes UNIX qui a été répertoriée lors des expériences.

Signalons toutefois qu'une autre unité est parfois utilisée pour l'évaluation d'un simulateur. Il s'agit du rapport entre le nombre d'instructions par seconde exécutées par le circuit réel et celui du nombre d'instructions par seconde exécutées par le simulateur.

Ce rapport est surtout usité dans le cas de processeurs mais pourrait l'être aussi pour d'autres circuits si l'on ne considère plus le nombre d'instructions exécutées par seconde, mais le temps nécessaire à un traitement donné (par exemple, le temps pour compter de 0 à 2^{16}). Encore faudrait il, pour que cette méthode soit employée, que les résultats publiés mentionnent clairement le nom du circuit simulé.

5.3 Estimation des ressources nécessaires

L'objectif de cette partie est d'évaluer de façon statique les ressources nécessaires à la simulation d'un circuit, c'est-à-dire:

- l'espace mémoire occupé,
- le temps de simulation d'un circuit.

L'établissement d'équations pour chacune de ces ressources permet de prouver, raisonnement mathématique à l'appui, l'avantage de la synthèse sur la traduction algorithmique et de le quantifier en fonction de paramètres.

La mise au point de ces équations a été inspirée d'une technique appelée "analyse de régression".

5.3.1 L'analyse de régression

Cette technique permet de déterminer la relation existant entre

plusieurs variables indépendantes $X_1, X_2, \dots, X_n$ et une unique variable Y (espace mémoire occupé ou temps de simulation dans notre cas).

Elle comprend deux phases distinctes qui doivent apporter des réponses aux questions suivantes:

1) quel est le modèle mathématique le plus approprié? Est-ce une droite, un polynôme, une fonction logarithmique, etc...

2) Etant donné un modèle spécifique, quels sont les paramètres et coefficients qui déterminent l'équation décrivant la relation recherchée?

Plusieurs stratégies peuvent être appliquées dans le cadre de l'analyse de régression. Les trois plus fréquemment rencontrées, décrites plus en détail dans [KK78], sont les suivantes:

1) *la stratégie en avant* consiste à partir d'un modèle mathématique simple (droite) que l'on complique par étapes successives jusqu'à obtenir la précision désirée.

2) *la stratégie en arrière* consiste à partir d'un modèle complexe (il s'agit en général d'un polynôme de degré élevé) que l'on simplifie par étapes successives en éliminant les monômes inutiles ou en réduisant leur degré.

3) *la stratégie expérimentale* consiste à utiliser un modèle dicté par l'expérience ou par la théorie, ce modèle étant par la suite modifié en avant ou en arrière en fonction des paramètres.

C'est cette dernière stratégie qui a été employée, dans la mesure où elle permet de tirer avantage du fait que l'outil a été écrit par l'auteur pour lequel il est en effet assez simple de savoir ce qui sera généré pour une description donnée.

5.3.2 De la théorie à la pratique

Etant donnée la stratégie choisie, c'est surtout la détermination des paramètres qui pose un problème. Il est en effet souhaitable d'obtenir les équations les plus simples possibles tout en offrant la plus grande précision possible. A cet égard, il convient de distinguer les parties contrôles (PCs) des parties opératives (POs).

Les PCs sont régulières et peuvent être caractérisées par leurs entrées et leurs sorties si l'on transforme toutes les UD's de la PC en PLA. Dans la mesure où seul le mode (entrée ou sortie) des variables détermine le

traitement qui leur est réservé, le nombre de paramètres est limité.

Le cas des POs est différent car toutes les variables y intervenant sont traitées à priori de façon très différente les unes des autres. Dès lors de nombreux paramètres seraient nécessaires à la mise au point d'équations de prédiction des ressources consommées par les POs et c'est la raison pour laquelle les équations obtenues ne concernent que les PCs. Pour les mêmes raisons, seuls les PLAs et décodeurs sont concernés par ces prédictions, ce qui n'est pas une restriction importante, si l'on considère que la fonctionnalité d'un graphe peut être exprimée à l'aide d'un PLA.

De plus, il a été nécessaire de distinguer l'emploi de la synthèse de celui de la traduction algorithmique car le code généré est très différent d'un cas à l'autre.

a. Prédiction de l'espace mémoire occupé

L'espace mémoire occupé par un modèle C se divise en deux parties:

- 1) la partie occupée par le simulateur proprement dit,
- 2) la partie occupée par les données.

La partie relative au simulateur est constituée de code C et est facile à prédire. Il suffit de se baser sur les algorithmes présentés aux sections 4.3.1. et 4.3.2. Ceux-ci permettent de déterminer les traitements effectués sur les variables de la partie contrôle et d'estimer en conséquence la taille du code C généré.

La taille du code C est mesurée en nombre de lignes plutôt qu'en octets. Ceci permet d'occulter les différences dues aux types de variable (qui produisent un code différent pour un même traitement) et donc de limiter le nombre de paramètres. Cette imprécision est toute relative et l'erreur dans la prédiction reste très faible.

La partie relative aux données n'intervient que dans le cas de la synthèse. C'est la taille des données présentes en mémoire lors de la simulation qui est évaluée et non celle des déclarations et initialisations de la SD correspondante. En effet, d'une part la SD peut être générée à l'extérieur du modèle C dans un fichier séparé et d'autre part, celle-ci serait beaucoup trop dépendante de son contenu (l'écriture d'un entier sur trente-deux bits peut nécessiter de un à dix caractères).

Taille d'un simulateur:

Comme il a été énoncé plus haut, la prédiction de la taille d'un simulateur ne concerne que les structures de la partie contrôle. Les équations résultantes s'appliquent à une seule structure, la taille du modèle C d'une partie contrôle pouvant être aisément obtenue par la somme des tailles de chacune de ses structures.

Le code C d'une procédure de la partie contrôle est constitué:

- de l'en-tête de la procédure,
- de la déclaration des paramètres,
- de la déclarations des variables internes,
- du corps de la procédure,

Le nombre de lignes pour les trois premières parties a été supposé constant afin de limiter le nombre de paramètres des équations. Ceci n'est pas tout à fait juste car le nombre de lignes de la première partie dépend du nombre de paramètres et de la taille de chacun de leur nom (pas plus de quatre-vingts caractères par ligne) et il dépend, dans la deuxième partie, du nombre de types différents des paramètres (une ligne par type) et de la taille de leur nom. Néanmoins cette approximation ne provoque que de faibles erreurs.

Le corps de la procédure peut être lui aussi décomposé en plusieurs parties distinctes dont le code C dépend du type de traduction employée:

- évaluation de la matrice ET,
- affectation des sorties.

Ces parties sont de tailles très différentes suivant qu'on utilise la synthèse ou la traduction algorithmique.

Dans le cas de la synthèse, la taille relative à l'évaluation de la matrice ET est directement proportionnelle au nombre d'entrées de la structure de contrôle et la taille relative à l'affectation des sorties est elle directement proportionnelle au nombre de sorties de la structure de contrôle. Il est à remarquer de plus que les équations obtenues ne dépendent pas du type de structure de contrôle (PLA ou décodeur).

Table 5.1 Prédiction de la taille d'une structure de contrôle

méthode employée		taille (ligne)	erreur (%)
Synthèse		$nbe + nbs + 6$	0.18
Traduction Algorithmique	PLAs	$nbl \times (nbs + 4 + nbe \times cr) + nbs + 5$	0.3
	décodeurs	$nbl \times (4 + nbe \times cr) + 2.5 \times nbs + 5$	5.33

Légende:

nbe: nombre d'entrées

nbs: nombre de sorties

nbl: nombre de lignes de la matrice ET

cr: coefficient de remplissage de la matrice ET (pourcentage de 1 et 0)

Dans le cas de la traduction algorithmique, deux paramètres supplémentaires sont nécessaires à la mise en forme des équations. Il s'agit du nombre de lignes de la matrice ET d'une part (voir algorithme 4.3) et du pourcentage de valeurs zéro ou un (autre que *don't care*) de la matrice ET. En effet, la comparaison d'une entrée à une valeur de la matrice ET n'est pas générée dans le cas d'une valeur *don't care*.

Les pourcentages d'erreur de la table 5.1 sont des moyennes effectuées sur de nombreuses expériences.

La figure 5.1 montre, à l'aide de courbes, l'avantage de la synthèse sur la traduction algorithmique dans le cas de la taille de la partie contrôle d'un modèle C. Ces courbes sont valides pour des PLAs ayant huit entrées (nombre le plus souvent rencontré dans les exemples traités) et dont le coefficient de remplissage est de 0.75 (coefficient moyen là aussi).

Si l'on suppose que le nombre de lignes d'une matrice ET (nbl) est au moins égal au nombre d'entrée (nbe) de la structure de contrôle, on obtient une croissance en nbe^2 pour la traduction algorithmique et en nbe pour la synthèse.

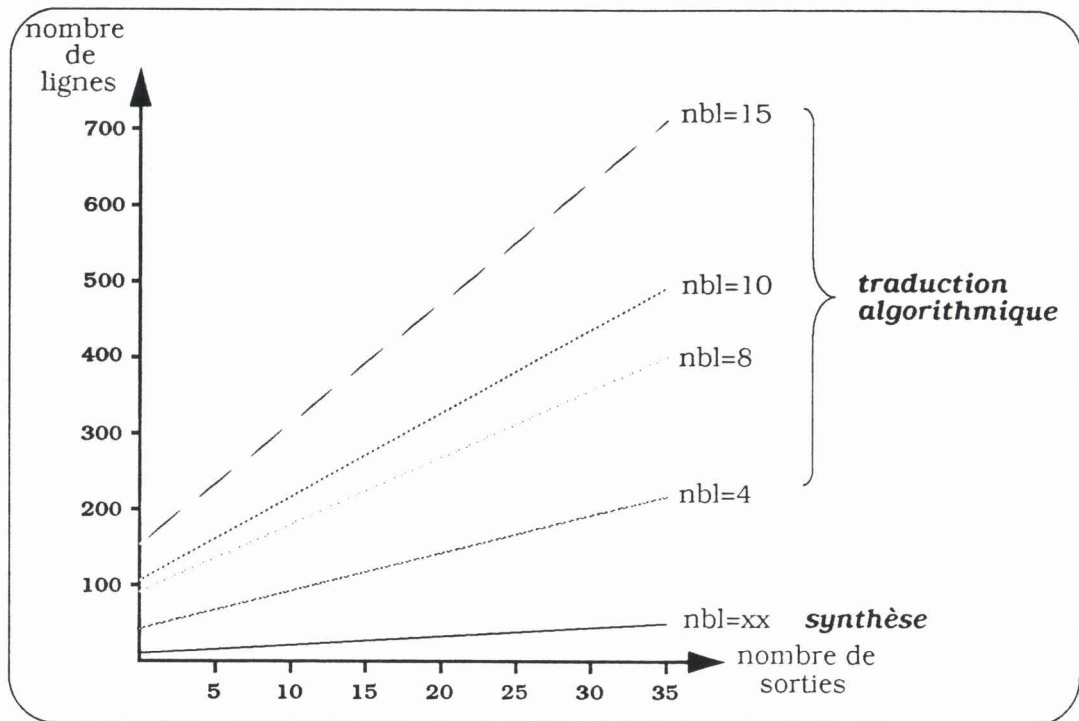


Figure 5.1 Courbes de la taille d'une structure de contrôle

Taille des données:

Deux équations sont nécessaires à l'évaluation de la taille des données suivant que celles-ci sont compactées ou non. Ces équations découlent de la figure 3.8.

La taille des données non compactées (voir équation a.) correspond au nombre de lignes de la matrice OU après expansion multiplié par le nombre d'octets nécessaires au stockage d'une ligne OU (multiple de quatre car le codage des lignes utilise des entiers de quatre octets).

La taille des données compactées (voir équation b.) correspond au nombre d'octets nécessaires pour stocker les bits de l'ensemble des lignes OU en les mettant les uns à la suite des autres.

Table 5.2 Prédiction de la taille des données

a. $4 \times 2^{nbe} \times E \left(\frac{nbs - 1}{32} + 1 \right)$

b. $4 \times E \left(\frac{2^{nbe} \times nbs - 1}{32} + 1 \right)$

b. Prédiction du temps de simulation

Celle-ci est beaucoup plus difficile à effectuer car elle nécessite, outre l'identification des traitements appliqués aux variables d'un PLA, l'évaluation préalable du temps d'exécution de chaque type de traitement. Cette évaluation s'avère délicate car elle ne peut tenir compte de l'optimisation de l'enchaînement des traitements qui peut être faite par le compilateur. Ceci est une première raison pour laquelle les équations obtenues permettent des prévisions moins précises que dans le cas de la taille des simulateurs.

D'autre part, les équations (voir table 5.3) font intervenir de nombreux paramètres car un même traitement peut nécessiter un temps qui va du simple au sextuple suivant le type de la variable à laquelle il s'applique.

Dans le cas de la synthèse, l'équation obtenue présente une faible marge d'erreur, due uniquement au problème énoncé plus haut. L'équation est une somme de trois termes, chacun d'entre-eux correspondant aux opérations décrites dans la figure 3.9 et au paragraphe 3.5.

La cas de la traduction algorithmique donne de moins bons résultats. Le problème est dû au fait que l'évaluation de la matrice ET dépend de la valeur des entrées au moment de la simulation. En effet, dès qu'une variable n'a pas la valeur indiquée par la matrice ET, l'évaluation de la ligne en cours est interrompue. Ainsi, l'évaluation d'une ligne dont la structure de contrôle comporte nbe entrées peut être stoppée après une seule comparaison ou après nbe comparaisons. La valeur des variables lors de l'exécution n'étant pas prévisible de façon statique, seule une approche probabiliste permet d'estimer le temps d'évaluation d'une matrice ET. Ce temps dépend:

- du nombre de lignes (nbl) de la matrice,
- du pourcentage (cr) de valeurs de la matrice autre que 'x',
- du nombre de comparaisons par ligne,
- du temps moyen d'une comparaison (y compris le temps passé dans les boucles).

Le nombre de comparaisons par ligne est obtenu en exprimant la probabilité d'aller jusqu'à n tests pour n compris entre 1 et nbe . On obtient

ainsi la somme suivante:

$$1 \times \frac{1}{2} + 2 \times \frac{1}{4} + 3 \times \frac{1}{8} + \dots + nbe \times \frac{1}{2^{nbe}} = \sum_{p=1}^{nbe} \frac{p}{2^p}$$

qui est utilisée dans l'équation de la table 5.3.

Les coefficients K_{1j} , K_{3j} et K_{4j} sont les temps calculés des traitements appliqués aux entrées (K_{1j} pour la synthèse) ou aux sorties (K_{3j} et K_{4j} pour la traduction algorithmique) suivant leur type. Tous les coefficients K_{1j} d'une structure de contrôle auront par exemple la même valeur si toutes ses entrées sont du même type.

Ces coefficients ont été déterminés pour une machine un MIPS et donnent des résultats en microseconde.

Table 5.3 Prédiction du temps de simulation d'une structure de contrôle

Méthode utilisée		temps de simulation (10^{-6} s)	erreur (%)
Synthèse		$\sum_{j=1}^{nbe} K_{1j} + 7 \times E\left(\frac{nbs-1}{32} + 1\right) + \sum_{j=1}^{nbs} K_{3j}$	2.33
traduction algorithmique	PLAs	$6 \times nbl \times cr \times \sum_{p=1}^{nbe} \frac{p}{2^p} + 2 \times nbs + \sum_{j=1}^{nbs} K_{3j}$	17.1
	décodeurs	$2.8 \times nbl \times cr \times \sum_{p=1}^{nbe} \frac{p}{2^p} + 2 \times nbs + \sum_{j=1}^{nbs} K_{4j}$	13.9

Les équations obtenues sont présentées par la table 5.3. La figure 5.2 montre, quant à elle, l'avantage de la synthèse sur la traduction algorithmique. Dans cette figure, les droites ont été tracées en utilisant dans les équations les mêmes valeurs de cr et nbe que pour la figure 5.a. Les points de la figure représentent des valeurs réelles obtenues lors de simulations; ils permettent de visualiser l'erreur de prévision.

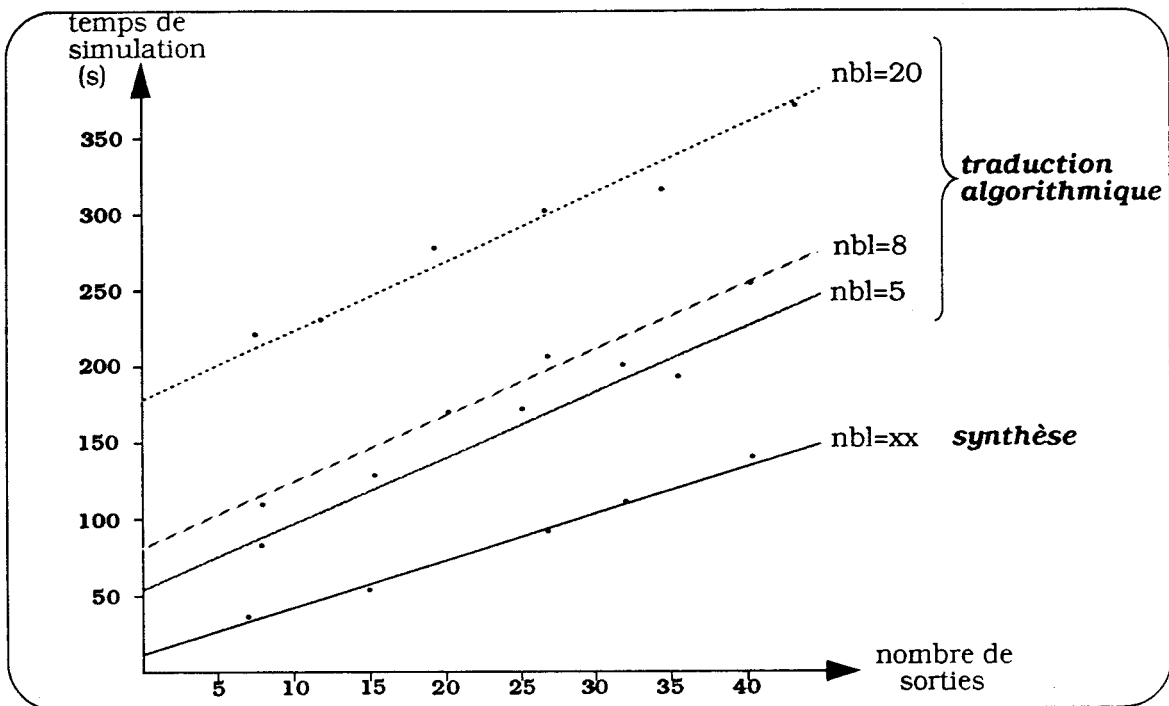


Figure 5.2 Courbes du temps de simulation d'une structure de contrôle

L'échelle du temps a été étendue à la seconde, ainsi les droites sont valides pour la simulation d'une structure de contrôle pendant un million de cycles.

5.4 Les résultats

5.4.1 Les circuits simulés

De nombreux circuits ont été simulés à l'aide de leur modèle C. Ceux-ci concernent des circuits tels que des additionneurs, compteurs ou UALs et des circuits plus complexes, en général des processeurs.

Les plus connus d'entre eux sont le 8085 d'Intel et un processeur RISC¹ comparable au processeur RISC I développé à l'université de Berkeley [PS82]. Des circuits multi-processeurs, construits à partir de ce même processeur, ont aussi été simulés.

Les descriptions LIDO du 8085 et de ce processeur RISC sont données en annexe.

1. Reduced Instruction Set Processor

5.4.2 Résultats généraux

Dans cette partie, sont rassemblés des résultats qui concernent la modélisation en général et l'intérêt d'employer telle structure LIDO plutôt que telle autre.

a. Importance de la partie contrôle

Ainsi que le laissait supposer la figure 3.3, la partie contrôle d'un simulateur en représente une part importante, y compris dans le cas des modèles C. Cette remarque fut à l'origine de la recherche d'une solution dédiée principalement aux parties contrôles.

Ceci apparaît clairement dans la figure 5.3 qui montre les parts relatives des parties contrôles de processeurs dans le cas de la traduction algorithmique. La partie contrôle d'un modèle C représente, dans les exemples traités, de soixante à quatre-vingts pour cent de la taille totale du modèle et de cinquante à quatre-vingts pour cent du temps de simulation d'un modèle.

L'analyse des résultats relatifs aux trois architectures RISC (*Riscp*, *Riscp4* et *Riscp8*) montre que le pourcentage du temps d'exécution augmente avec le nombre de processeurs. Cette augmentation est nette de un à quatre processeurs à cause du circuit de séquençement supplémentaire nécessaire à l'arbitrage des processeurs. L'augmentation est beaucoup moins importante de quatre à huit processeurs car le circuit d'arbitrage existe déjà.

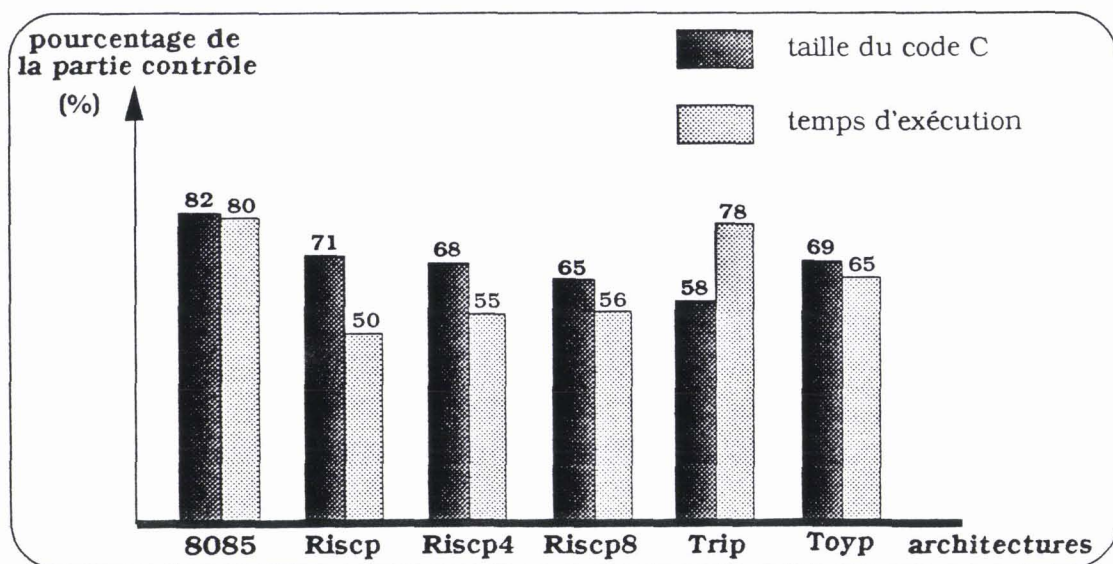


Figure 5.3 Part relative de la partie contrôle d'un modèle C

Par contre, l'augmentation du nombre de processeurs va de pair avec la diminution de la part relative de la partie contrôle si l'on considère la taille d'un modèle C. Ceci est dû au fait que l'accroissement du nombre d'appels de procédure est contre-balançé par l'augmentation des déclarations du modèle C, dont la taille est directement proportionnelle au nombre de processeurs.

b. Avantage de la microprogrammation

Wilkes [W53, WT53] fut le premier à identifier les avantages de la microprogrammation au début des années cinquante. Il fallut pourtant attendre les développements de la technologie des mémoires pour que ce principe soit réellement appliqué en 1964 avec la famille IBM/360 [Tuc67].

De nos jours, l'emploi de la microprogrammation dans les ordinateurs est chose courante afin de réduire l'espace nécessaire au système de commande; citons par exemple le cas du micro VAX-I [LE89, Pat85], qui implémente 80% des instructions VAX et atteint des performances égales à 90% d'une implantation VLSI pour une réduction de la taille du système de commande d'un facteur 8.

La microprogrammation a aussi rendu possible la mise au point d'architectures RISC [Pat83].

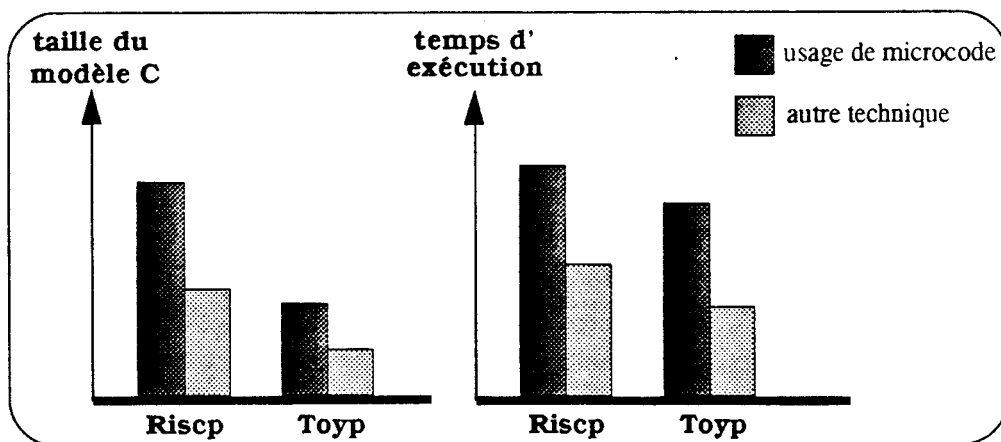


Figure 5.4 Intérêt des versions microprogrammées

Les expériences de simulation effectuées ont montré l'intérêt des versions programmées pour les modèles C tant en ce qui concerne leur taille qu'en ce qui concerne leur temps d'exécution.

La figure 5.4 donne les résultats de deux des comparaisons effectuées.

Ces résultats s'appliquent uniquement aux parties contrôles de circuits dans la mesure où seules celles-ci sont concernées par la microprogrammation. Ni les valeurs réelles des résultats, ni les unités n'apparaissent dans cette figure car seule importe la comparaison du type de modélisation. Il n'est en effet pas encore question de comparer les performances de l'outil à d'autres simulateurs.

Ainsi, les modèles C issus des versions microprogrammées sont deux fois plus compacts et deux fois plus rapides que ceux issus des autres versions.

Cette remarque fut, elle aussi, à l'origine de l'idée de transformer les parties contrôles avant la génération.

c. Avantage des graphes

L'utilisation du générateur dans sa version élémentaire (traduction algorithmique) a montré l'avantage d'utiliser la structure de graphe plutôt que des PLAs ou décodeurs pour la description des parties contrôles. L'avantage des graphes concerne surtout le temps de simulation. En effet, le temps d'exécution des exemples traités est réduit d'un facteur 5 (voir figure 5.5) alors que la taille n'est que légèrement inférieure.

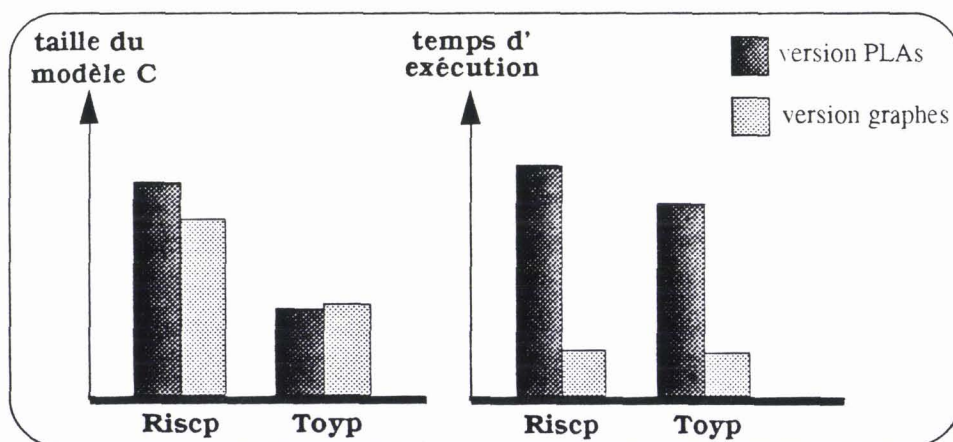


Figure 5.5 Intérêt des versions à base de graphes

Le graphe permet la description d'un circuit au niveau algorithmique, ce qui explique sa meilleure adaptation à la traduction algorithmique (traduction d'un algorithme en un autre) alors que les PLAs et décodeurs, qui expriment la fonctionnalité d'une partie contrôle sous forme de données, s'y prêtent moins bien.

d. Limites de la traduction algorithmique

La génération d'un simulateur du 8085 en utilisant la traduction algorithmique a révélé l'importance de la taille de son modèle C, qui atteint près de 4000 lignes et 128 Kilo-octets de code C.

Si l'on considère le rapport entre la taille du 8085 et celles de processeurs plus complexes tels que le 8086 ou le Motorola 68000 (qui contiennent respectivement quatre et dix fois plus de transistors), on peut estimer à environ 500 K-octets et 1,25 M-octets la taille des modèles C qui seraient générés.

La taille du simulateur du 68000 serait sans doute moins grande à cause de sa nature microprogrammée [Gro84]. Néanmoins, ces valeurs sont beaucoup trop élevées, surtout si l'on veut pouvoir traiter des circuits actuels (plus complexes).

5.4.3 Efficacité de la synthèse

Cette partie montre l'efficacité de la synthèse en comparant dans un premier temps (a.) les tailles et performances des modèles C de parties contrôles suivant le type de traduction. Ce sont ensuite les caractéristiques des modèles C de structures de contrôles (b.) qui servent à la comparaison afin d'écarter les déclarations, le programme principal et les procédures non directement liées à une unité de description.

La séparation du contrôle et des données, effectuée en début de synthèse, recopie une partie de la description dans une unité fonctionnelle. Or, les FUNs ont été décrites au chapitre quatre comme des structures servant à la description des parties opératives. Le lecteur critique pourrait donc s'interroger sur la valeur des résultats présentés ci-dessous.

En fait, les modèles C des parties contrôles simulées comprennent aussi ces unités fonctionnelles dans le cas de la synthèse. Ainsi, aucune partie d'un modèle n'est écartée et l'avantage de la synthèse est bien dû à la méthode employée et non à la réduction du modèle pris en compte.

a. Résultats locaux

Les résultats donnés dans cette partie concernent des comparaisons prenant en compte, une à une, les principales structures de contrôle des circuits simulés.

La taille et le temps de simulation des modèles C de ces structures sont rassemblés dans la table 5.4. Le facteur de réduction de la taille des modèles C (noté FR) ainsi que le facteur d'accélération de leur simulation (noté FA) ont été calculés pour chacune d'elles.

Les simulations ont été effectuées au LIFL¹ sur le serveur *lifl1* (3 MIPS) alors que le réseau n'était pas chargé pour une plus grande fiabilité des résultats (des écarts allant jusqu'à 30% ont été enregistrés pour différentes simulations d'une même structure lorsque le réseau est chargé).

La taille d'une structure est celle du sous-programme qui lui correspond dans le modèle C. Cette taille ne comprend donc ni les déclarations globales, ni la boucle du programme principal nécessaires à la simulation de toute structure.

Chaque structure a été simulée pendant 10^5 cycles de simulation. Le temps d'exécution en seconde et sur une machine d'un MIPS d'une structure de la table peut donc être obtenu en multipliant la valeur donnée dans la table par $3 \cdot 10^{-5}$.

Les noms des structures de la table 5.4 sont composés du nom de l'architecture (par exemple 8085) et de celui de la structure dans la description (par exemple P1_2, où 1 et 2 sont respectivement les niveau et rang). La présence éventuelle de la lettre 'm' indique qu'il s'agit d'une version microprogrammée.

Table 5.4 Comparaison des résultats pour les structures de contrôle

structures \ résultats	Taille (Koets)			Temps d'exécution (s)			
	technique	T. algorithmique	synthèse	FR	T. algorithmique	synthèse	FA
8085P1_1		11.5	1	11.5	20	3.7	5.4
8085P2_1		75	4.3	17.5	73	18	4.1
8085P3_4		4.3	0.5	8.5	8	1.3	6.15
RiscpP1_1		23.8	1.4	17	26	5	5.2
RiscpP2_1		2.7	0.5	5.5	3	1.4	2.14
RiscmP1_1		1	0.4	2.5	3	1.3	6.15
RiscmP3_1		5.8	0.9	6.5	11.2	3.6	3.1
ToypP2_1		5.7	1	5.5	15.7	4.3	3.65

1. Laboratoire d'Informatique Fondamentale de Lille.

Les résultats de ce tableau montrent des FR variant de 2.5 à 17.5.

Il apparaît que le FR croît avec la taille des structures issues de la traduction algorithmique. Il suffit de faire la différence des polynômes présentés dans la table 5.1 pour en avoir une justification mathématique. Le résultat est en effet très proche de l'équation obtenue pour la traduction algorithmique.

Il est par contre beaucoup plus difficile d'établir une règle générale en ce qui concerne le FA, dont l'évolution dépend des caractéristiques des structures. Celui-ci varie de 2.14 à 6.15 dans les exemples traités.

b. Résultats globaux

Les résultats donnés dans cette partie concernent des comparaisons de la totalité des modèles C de parties contrôles.

Les simulations ont été effectuées dans les mêmes conditions que pour les résultats locaux. Les tailles de la table 5.5 correspondent à celles de l'ensemble des modèles C de parties contrôles, non compris les procédures d'entrée/sortie. Il faudra donc tenir compte de chacune des parties qui composent un modèle C (déclarations, programme principal, les corps de unités et celui de la procédure de réinitialisation) lors de l'analyse des résultats.

Les structures ont été simulées pendant 10^4 cycles de simulation afin de conserver la même unité de mesure dans toute la table. C'est donc par $3 \cdot 10^{-4}$ qu'il faut multiplier les temps d'exécution si l'on veut obtenir le temps de simulation de la partie contrôle correspondante sur une machine 1 MIPS.

Table 5.5 Comparaison des résultats pour les parties contrôles

structures \ résultats	technique	Taille (Koctets)			Temps d'exécution (s)		
		T. algorithmique	synthèse	FR	T. algorithmique	synthèse	FA
8085		106	17.3	6.1	12.15	4.8	2.5
Riscp		28	3.6	7.7	2.9	0.7	4.1
Riscp4		37	7	5.3	12.0	2.9	4.1
Riscp8		41	10.4	3.9	23.4	5.2	4.5
Riscm		14.6	6.3	2.3	2.0	1.4	1.4
Toyp		12.9	5.1	2.5	2.8	1.25	2.2
Toym		6.7	3.4	2	1.4	1.1	1.3
Trip		4.6	2.2	2.1	1.0	0.48	2.1

Les FA et FR sont moins élevés que ceux de la table 5.4 à cause de la prise en compte dans les mesures de toutes les parties d'un modèle C (excepté les procédures d'entrée/sortie).

Pour la même raison, il n'est pas possible de donner une règle simple de l'évolution du FR ou du FA.

Il apparaît nettement que les versions microprogrammées, avantageuses lorsque la traduction algorithmique est utilisée, donnent de moins bon résultats lorsque la synthèse est employée. Ceci est d'autant plus vrai que le circuit traité est important.

A mesure que le nombre de processeurs des architectures RISC traitées augmente, le FR décroît. L'explication est toute simple; l'augmentation du nombre de processeurs se traduit dans un modèle C par l'accroissement de la taille des déclarations, du nombre d'appel de procédures (unités) et de la taille du sous-programme de réinitialisation. Seules les unités sont affectées par la synthèse et leur taille ne change pas d'une architecture à l'autre. Ainsi, le FR de chaque structure ne change pas et le FR global diminue avec l'augmentation de la taille des autres parties du modèle C.

5.4.4 Efficacité du logiciel

Les performances de l'outil ont été comparées au simulateur VHDL d'Intermetrics afin de le situer par rapport aux autres outils du même type.

Le langage VHDL a été choisi pour sa couverture des niveaux d'abstraction (il est nécessaire d'effectuer les comparaisons aux mêmes niveaux d'abstraction) et à cause de la similitude de sa méthode (traduction en langage C avant la production de code objet). De plus, VHDL est le premier HDL standard depuis Décembre 1987.

La comparaison à l'aide de résultats publiés dans la littérature scientifique s'avère difficile car:

- les simulations ne sont pas toujours effectuées au même niveau d'abstraction,
- les simulations concernent souvent des circuits différents (excepté le compteur 16-bits qui semble faire l'unanimité),
- des informations nécessaires aux comparaisons font souvent défaut (sur quelle machine hôte par exemple).

Deux circuits ont été décrits et simulés en VHDL. Il s'agit d'un

compteur 16-bits et de la partie contrôle du processeur RISC mentionné plus haut sous le nom de *Riscp*. Seule la partie contrôle de ce processeur a été décrite afin de comparer efficacement l'emploi de la synthèse dans notre générateur au simulateur VHDL.

La partie contrôle du 8085 d'Intel a aussi été décrite en VHDL mais n'a pu être simulée faute de simulateur disponible¹.

Les simulations ont été effectuées sur VAX-8810, système VMS dont les performances sont de 9.5 MIPS. Les temps d'exécution des tables 5.6 et 5.7 n'ont pas été normalisés car les simulations des modèles C produits par notre générateur ont eu lieu sur la même machine afin d'obtenir une comparaison fiable.

Pour la même raison, les opérations d'entrée/sortie ont été écartées de chaque simulateur. Ainsi, dans le cas du simulateur VHDL, les options de génération /NOLINE_TRACE et /NO_CHECK ainsi que les options de compilation /NO_TRACE et NO_DEBUG ont été utilisées.

Table 5.6 Temps de simulation du compteur 16-bit

logiciel \ domaine	fonctionnel	structurel
simulateurs C (LIDO)	0s 82c	7s 54c
simulateurs C (VHDL)	40s 35c	2 mn 9s 76c

Le compteur a été simulé pour un comptage de 0 à 2^{16} (voir table 5.6) alors que la partie contrôle de *Riscp* (table 5.7) l'a été pendant 10^5 cycles de simulation (cycles d'horloge).

Les résultats de la table 5.6 montrent que l'outil est plus performant que le simulateur VHDL, cela alors que la traduction algorithmique a été employée. D'autre part, il apparaît que les performances de l'outil sont d'autant plus intéressantes que le niveau d'abstraction est élevé. Ceci s'explique par le fait que LIDO est un langage de description fonctionnelle au niveau transfert de registre alors que VHDL est adapté à tous les niveaux d'abstraction, donc moins adapté à chacun d'entre-eux.

1. Les simulations ont été réalisées à G.M.D. (Gesellschaft für Mathematik und Datenverarbeitung MBH, centre allemand de recherche pour l'informatique). Des problèmes administratifs n'ont pas permis le renouvellement de la licence d'exploitation du simulateur pour la période de présence au centre de Bonn.

Table 5.7 Temps de simulation de la partie contrôle de Riscp

logiciel \ mesures	technique employée	taille de la description	taille du modèle C	temps de production du compilé	temps d'exécution
simulateurs C (LIDO)	traduction algorithmique	4K	27.5K	18.93s	18.9s
	synthèse		4.5K	5.96s	6.18s
simulateurs C (VHDL)	description fonctionnelle	12K	123K	43.5s	22mn45s
	approche ROM	9K	102K	34.83s	14mn23s

Si l'on compare les résultats donnés ici et ceux de la table 5.5, on peut constater que le FA de la synthèse par rapport à la traduction algorithmique a changé (environ 3 dans ce cas). Ceci est dû au fait que les résultats de chacune des tables ont été obtenus sur des machines hôtes différentes. Les temps d'exécution sont bien sûr différents, mais on aurait pu s'attendre à retrouver le même FA. Ceci souligne encore, si besoin était, la nécessité d'effectuer les expériences dont on veut comparer les résultats sur la même machine.

L'emploi de l'approche ROM lors de la modélisation en VHDL permet d'obtenir des résultats sensiblement meilleurs lors de la simulation (réduction du temps d'exécution de plus de 35%). La taille du modèle C produit n'est par contre que très peu réduite.

Enfin, si l'on compare les meilleurs résultats obtenus par chacun des outils (ceux de la synthèse d'un côté et ceux de l'approche ROM de l'autre), on obtient un net avantage en faveur de notre outil puisque le temps d'exécution est près de 140 fois plus long dans le cas de VHDL. De plus, tous les autres critères pris en compte conduisent à la même conclusion.

5.5 Conclusion

La réelle portabilité de l'outil a été démontrée en générant des simulateurs sur des machines différentes (HP 9000-Unix, SUN-Unix, VAX8810-VMS). La portabilité peut donc réellement être une qualité offerte par la compilation indirecte.

Les résultats obtenus ont clairement montré la supériorité de la synthèse sur la représentation classique sous forme d'algorithme. Ceci a été prouvé à l'aide d'expériences ainsi que d'équations mathématiques, qui permettent de surcroît la prévision des ressources nécessaires à une

simulation.

De plus, la comparaison de l'outil avec un simulateur VHDL a permis d'évaluer l'efficacité du générateur et des simulateurs produits, qui se sont avérés très performants.

Les processeurs simulés sont assez anciens. En se basant sur les caractéristiques du 8085 (nombre de transistors, pourcentage de la partie contrôle) et de son simulateur (taille respective des PC et PO) et sur celles du MC68020 (192000 transistors, partie contrôle de 70%), la taille du simulateur du MC68020 qui serait obtenu a été évaluée à 1.88 Moctets, dont 0.78 Moctets pour la partie contrôle. Bien que ces chiffres tiennent compte de la ROM de ce processeur, ils sont à prendre sous réserve car assujettis à la vérification de la forte hypothèse selon laquelle le rapport entre la taille du MC68020 et celle de son simulateur serait le même que celui du 8085. Cette évaluation n'est destinée qu'à estimer le champ d'application du générateur.

Chapitre 6

Conclusion

Le travail effectué au cours de cette Thèse se situe dans le cadre de la recherche de solutions nouvelles pour la réalisation d'outils de Conception Assistée par Ordinateur, dont les performances doivent évoluer afin de faire face à l'accroissement de complexité des circuits intégrés.

Ainsi, la présentation des différentes étapes de la CAO a permis de souligner l'importance de la simulation. C'est pourquoi les efforts ont été portés sur la recherche d'une solution de simulation qui réponde aux besoins croissants de performance. Constatant de plus la nécessité de communiquer et d'échanger des modèles de circuits (projet EUROCHIP par exemple), l'objectif retenu a été de proposer une solution permettant d'obtenir des simulateurs **efficaces, compacts, portables et souples** pouvant être utilisés sur des machines courantes (stations de travail ou PCs).

Les différentes techniques de simulation ont donc été analysées afin d'en dégager les avantages et inconvénients. Ainsi, les techniques d'interprétation indirecte, d'interprétation directe et de compilation directe ont été écartées pour retenir celle de la compilation indirecte. Cette technique consiste à générer un programme de simulation à partir de la description, dans un langage de description de matériel, de l'architecture à simuler. Le langage C a été choisi comme langage cible pour son efficacité, sa compacité et sa portabilité.

La contribution essentielle de cette Thèse est une solution qui vient se greffer à cette technique. Elle permet de tirer parti du potentiel de transformations diverses qu'offre la compilation indirecte avant la génération du code C, en portant un maximum d'efforts sur la génération afin de réduire le temps de simulation. Partant de la constatation de la part importante que représente la partie contrôle des machines simulées en ressources consommées (espace et temps CPU), une méthode de transformation de parties contrôles a été définie. Ainsi, la fonctionnalité d'une partie contrôle est exprimée sous forme de données afin d'éviter le schéma classique de traduction d'un algorithme en un autre. C'est donc un algorithme très réduit qui réalise la simulation d'une partie contrôle, ce qui permet d'accélérer la simulation.

L'outil de simulation proposé a été défini dans l'environnement LIDO sur système Unix. Il a été transporté avec succès sur des machines

supportant le système VMS, démontrant ainsi sa portabilité et celles des simulateurs produits. De plus, la méthode de synthèse des parties contrôles proposée est indépendante du HDL utilisé, ce qui ne fait qu'accroître son intérêt.

Les résultats présentés au chapitre cinq ont montré le gain appréciable obtenu grâce à l'emploi de cette méthode. D'autre part, ces résultats ont été confirmés par des équations qui permettent la prédiction des ressources consommées lors de la simulation. Par ailleurs, les expériences comparatives effectuées avec un logiciel du commerce ont permis d'évaluer l'efficacité du générateur et des simulateurs produits, qui se sont avérés très performants.

Il serait intéressant de décrire un processeur de la taille du MC68020 afin d'en générer un simulateur et de pousser plus avant l'évaluation de l'étendue d'utilisation du logiciel proposé. L'emploi de stations de travail très performantes (en particulier de machines RISC) serait alors un recours possible s'il s'avérait que la simulation d'un tel processeur soit gourmande en ressources. Ceci peut d'ailleurs être envisagé dans le cas contraire dans la mesure où le prix de ce type de machines est en baisse constante.

La taille des données produites par la synthèse dépend pour beaucoup de la description de la partie contrôle d'un circuit dont la qualité de décomposition repose entièrement sur le concepteur. C'est pourquoi, il serait judicieux de définir une méthode de décomposition automatique d'une partie contrôle.

La représentation sous forme de données des parties contrôles se prête à leur description à l'aide de boîtiers ROM/RAM. Ainsi, la définition d'une interface EDIF pourrait permettre la production à faible coût de prototypes de parties contrôles dans le cadre d'une première implantation physique.

Le langage LIDO ne couvre que des niveaux élevés d'abstraction. L'utilisation de la solution proposée dans l'environnement LIDO trouve tout son intérêt avec les compilateurs de silicium et la venue d'outils de synthèse de haut niveau. L'usage de cette méthode pour la vérification fonctionnelle d'architectures avant l'emploi d'un tel compilateur nécessite l'ajout de caractéristiques temporelles au langage LIDO. Celles-ci pourraient être constituées de nouvelles règles de grammaire ou simplement avoir la forme d'annotations.

A Annexe

A.1 Description en LIDO du processeur Riscp

/* Ce processeur 32 bits, base sur les concepts RISC, realise 23 instructions:

- 5 instructions de saut,
- 16 instructions arithmetiques et logiques,
- instructions LOAD & STORE.

Comme dans toute machine RISC, les instructions ont un format unique:

- les bits 28 et 29 du registre d'instruction (*ir(29, 28)*) donnent le type de l'instruction (LOAD-STORE, ALU, JUMP),
- les bits 24 a 27 (*ir(27, 24)*) donnent le code de l'instruction (par exemple, 0000 est le code de l'addition).

Le transit des donnees se fait a l'aide de deux bus (*bus1* et *bus2*) et d'une memoire locale (*rmem(16, 32)*).

La partie controle est decrite a l'aide d'un PLA (*Pla 1, 1*). Le second PLA (*Pla 2, 1*) a une fonction de decodeur et decide si un saut conditionnel a lieu ou non. Les operations sont animees par deux phases d'horloge; la premiere synchronise la partie controle tandis que la seconde synchronise la partie operative.*/

/* Partie Controle: */

Clock; 2. /* horloge a deux phases */

/* description structurelle du sequenceur: PLA + registre d'etat *re* */

Pla 1,1:[1] *ir(29), ir(28), ir(27), ir(26), ir(25), ir(24), re(1), re(0)* -/*entrees */

re(1), re(0), sa, sual, dtzs, dtr, dtd, dadd, dadr, dsub, dsur, dou, det, dex, ddif,

dded, ddeg, ddar, dcom, dinc, ddec, dpas, dzer, schr, scon, ses1, ses2; /* sorties */

/* matrice ET -	matrice OU	/*
(xx xxxx 00) - (01 10 000 0000 0000 0000 0000 0000);		/* "fetch" */
(00 0000 01) - (10 01 111 1000 0000 0000 0000 0000);		/* addition */
(00 0001 01) - (10 01 111 0100 0000 0000 0000 0000);		/* add + carry */
(00 0010 01) - (10 01 111 0010 0000 0000 0000 0000);		/* subtract */
(00 0011 01) - (10 01 111 0001 0000 0000 0000 0000);		/* sub - carry */
(00 0100 01) - (10 01 100 0000 1000 0000 0000 0000);		/* OR */
(00 0101 01) - (10 01 100 0000 0100 0000 0000 0000);		/* AND */
(00 0110 01) - (10 01 100 0000 0010 0000 0000 0000);		/* EXOR */
(00 0111 01) - (10 01 111 0000 0001 0000 0000 0000);		/* difference */
(00 1000 01) - (10 01 100 0000 0000 1000 0000 0000);		/* shift right */
(00 1001 01) - (10 01 100 0000 0000 0100 0000 0000);		/* shift left */
(00 1010 01) - (10 01 100 0000 0000 0010 0000 0000);		/* shift arit.*/
(00 1011 01) - (10 01 100 0000 0000 0001 0000 0000);		/* complement. */
(00 1100 01) - (10 01 111 0000 0000 0000 1000 0000);		/* incrementation */
(00 1101 01) - (10 01 111 0000 0000 0000 0100 0000);		/* decrementation */
(00 1110 01) - (10 01 000 0000 0000 0000 0010 0000);		/* transfer */
(00 1111 01) - (10 01 000 0000 0000 0000 0001 0000);		/* 0=>flags */
(00 xxxx 10) - (01 10 000 0000 0000 0000 0000 1000);		/* overlapping */
(01 xxxx 01) - (00 00 000 0000 0000 0000 0000 0100);		/* jumps cond. */
(10 xxxx 01) - (11 00 000 0000 0000 0000 0000 0010);		/* I/O 1st step */
(10 xxxx 11) - (00 00 000 0000 0000 0000 0000 0001);		/* I/O 2nd step */

/* decodage des sauts conditionnels */

Pla 2,1:[2] scon, ir(26), ir(25), ir(24), r, s, d, z - dcp;

```
(10xxxxxx) - (1); /* unconditional jump */
(11001xxx) - (1); /* jump if carry = 1 */
(11000xxx) - (0);
(1101x1xx) - (1); /* jump if sign = 1 */
(1101x0xx) - (0);
(1110xx1x) - (1); /* jump if overflow = 1 */
(1110xx0x) - (0);
(1111xxx1) - (1); /* jump if zero = 1 */
(1111xxx0) - (0);
```

/* Partie operative: */

Fun 3,1; [2]; /* unite de transit de bus, placee avant l'UAL */

/sa/ prc => eat(19,0), emem(@eat) => ir, 1 + prc => prc;

/sual/ ireg1 => rat1, ireg2 => rat2,

rmem(@rat1) => bus1, rmem(@rat2) => bus2;

/schr & ddif/ act => bus1, bus1 => rmem(@rat1);

/dcp/ ir(19,0) => prc; /* adresse de saut */

/ses1 & ir(24)! / 1 => erd, ir(19,0) => eat(19,0), emem(@eat) => bus1;

/ses1 & ir(24) / ir(23,20) => rat1, rmem(@rat1) => bus1;

/ses2 & ir(24)! / ir(23,20) => rat1, bus1 => rmem(@rat1);

/ses2 & ir(24) / 1 => ewr, ir(19,0) => eat(19,0), bus1 => emem(@eat).

Fun 3,2; [2]; /* Unite Arithmetique et Logique - description fonctionnelle */

/dadd/ bus1 + bus2 => cact;

/dadr/ bus1 + bus2 + r => cact;

/dsub/ bus1 - bus2 => cact;

/dsur/ bus1 - bus2 - r => cact;

/dou/ bus1 | bus2 => cact(31,0);

/det/ bus1 & bus2 => cact(31,0);

/dex/ bus1 ^ bus2 => cact(31,0);

/ddif/ bus1 - bus2 => cact;

/dded/ r => cact(31), bus1(31,1) => cact(30,0), bus1(0) => r;

/ddeg/ r => cact(0), bus1(30,0) => cact(31,1), bus1(31) => r;

/ddar/ bus1(31) => cact(31), bus1(31,1) => cact(30,0), bus1(0) => r;

/dcom/ bus1(31,0)! => cact(31,0);

/dinc/ bus1 + 1 => cact;

/ddec/ bus1 - 1 => cact;

/dpas/ bus1 => cact(31,0).

Fun 3,3; [2]; /* unite de tests arithmetiques et logiques */

/dzer/ 0 => r, 0 => s, 0 => d, 0 => z;

/dtzs/ cact(31) => s, cact / 1 => z!;

/dtd/ bus1(31) & bus2(31) & cact(31)! => cdn, /* negative overflow */

bus1(31)! & bus2(31)! & cact(31) => cdp, /* positive overflow */

cdn | cdp => d;

/dtr/ cact(31) => r.

Fun 3,4; [2]; /* unite de chargement de resultat, placee apres l'UAL */

/sual/ cact(31,0) => act.

/* declarations: */

Memories: rmem(16,32); /* memoire locale */
emem(4K,32). /* memoire externe de programme */

Registers: ir(32); /* registre d'instruction */
re(2); /* registre d'etat (sequenceur) */
fr(4); /* registre des drapeaux */
prc(20). /* compteur de programme */

Latches: act(32); /* accumulateur temporaire */
rat1(4); rat2(4); /* adresses registre */
eat(20); /* sortie d'adresse externe */
sa; sual; scon; ses1; ses2; schr; dadd; dadr; dsub; dsur; dou; det; dex; ddif;
dded; ddeg; ddar; dcom; dinc; ddec; dpas; dzer; dtzs; dtr; dtd.

Terminals|2|;bus1(32);bus2(32). /* terminaux dynamiques avec memoire transitoire de deux phases d'horloge */

Subregisters:ir(31,24)=icode;ir(23,20)=ireg1;ir(19,16)=ireg2.

Subreg;fr(3)=r;fr(2)=s;fr(1)=d;fr(0)=z.

Terminals: dcp; /* signal de saut */
erd; /* signal de lecture de la memoire externe */
ewr. /* signal d'ecriture de la memoire externe */

Terminals: cact(32); /* sortie de l'UAL */
cdn;cdp. /* debordement (negatif ou positif) */

End.

[illegible]

```
P1_1(ir, re, sa, sual, dtzs, dtr, dtd, dadd, dadr, dsub, dsur, dou, det, dex, ddif, dded, ddeg, ddar, dcom, dinc, ddec,
dpas, dzer, schr, scon, ses1, ses2)
```

```
es_ *sa, *sual, *dtzs, *dtr, *dtd, *dadd, *dadr, *dsub, *dsur, *dou, *det, *dex, *ddif, *dded, *ddeg, *ddar, *dcom,
*dinc, *ddec, *dpas, *dzer, *schr, *scon, *ses1, *ses2;
```

```
rm_ *ir, *re;
```

```
{register unsigned entree_=0, sortie0_;
```

```
entree_<<=1; entree_ |= (ir->esclave[0]>>29)&1; /* calcul */
```

```
entree_<<=1; entree_ |= (ir->esclave[0]>>28)&1;
```

```
entree_<<=1; entree_ |= (ir->esclave[0]>>27)&1;
```

```
entree_<<=1; entree_ |= (ir->esclave[0]>>26)&1;
```

```
entree_<<=1; entree_ |= (ir->esclave[0]>>25)&1; /* de */
```

```
entree_<<=1; entree_ |= (ir->esclave[0]>>24)&1;
```

```
entree_<<=1; entree_ |= (re->esclave[0]>>1)&1;
```

```
entree_<<=1; entree_ |= (re->esclave[0]>>0)&1; /* l'adresse */
```

```
sortie0_ = mouP1_1[entree_]; /* acces a la fonction a realiser */
```

```
re->maitre[0]&=-2; re->maitre[0] |= (sortie0_&1)<<1; /* realisation */
```

```
re->maitre[0]&=-1; re->maitre[0] |= ((sortie0_>>1)&1);
```

```
*sa=(sortie0_>>2)&1;
```

```
*sual=(sortie0_>>3)&1;
```

```
*dtzs=(sortie0_>>4)&1
```

```
*dtr=(sortie0_>>5)&1;
```

```
*dtd=(sortie0_>>6)&1;
```

```
*dadd=(sortie0_>>7)&1; /* de */
```

```
*dadr=(sortie0_>>8)&1;
```

```
*dsub=(sortie0_>>9)&1;
```

```
*dsur=(sortie0_>>10)&1;
```

```
*dou=(sortie0_>>11)&1;
```

```
*det=(sortie0_>>12)&1;
```

```
*dex=(sortie0_>>13)&1;
```

```
*ddif=(sortie0_>>14)&1;
```

```
*dded=(sortie0_>>15)&1;
```

```
*ddeg=(sortie0_>>16)&1; /* la */
```

```
*ddar=(sortie0_>>17)&1;
```

```
*dcom=(sortie0_>>18)&1;
```

```
*dinc=(sortie0_>>19)&1;
```

```
*ddec=(sortie0_>>20)&1;
```

```
*dpas=(sortie0_>>21)&1;
```

```
*dzer=(sortie0_>>22)&1;
```

```
*schr=(sortie0_>>23)&1;
```

```
*scon=(sortie0_>>24)&1;
```

```
*ses1=(sortie0_>>25)&1;
```

```
*ses2=(sortie0_>>26)&1; /* fonction */
```

```
nbtrans_+=27;
```

```
|
```

```
P2_1(scon, ir, fr, dcp)
```

```
es_ *scon, *dcp;
```

```
rm_ *ir, *fr;
```

```
{register unsigned entree_=0, sortie0_;
```

```
entree_<<=1; entree_ |= *scon;
```

```
entree_<<=1; entree_ |= (ir->esclave[0]>>26)&1;
```

```
entree_<<=1; entree_ |= (ir->esclave[0]>>25)&1;
```

```
entree_<<=1; entree_ |= (ir->esclave[0]>>24)&1;
```

```
entree_<<=1; entree_ |= (fr->esclave[0]>>3)&1;
```

```
entree_<<=1; entree_ |= (fr->esclave[0]>>2)&1;
```

```
entree_<<=1; entree_ |= (fr->esclave[0]>>1)&1;
```

```
entree_<<=1; entree_ |= (fr->esclave[0]>>0)&1;
```

```
sortie0_ = mouP2_1[entree_];
```

```
*dcp=(clock_+1)<<1 | sortie0_&1;
```

```
nbtrans_+=1;
```

```
|
```



```

F3_1(sa, prc, eat, emem, ir, sual, rat1, rat2, rmem, bus1, bus2, schr, ddif, act, dcp, ses1, erd, ses2, ewr)
es_ *sa, *sual, *schr, *ddif, *dcp, *ses1, *erd, *ses2, *ewr;
em_ *eat, *rat1, *rat2, *act;
rm_ *prc, *ir;
tm_ *bus1, *bus2;
unsigned emem[], rmem[];
ch_sl(*sa, infog1_);
cond_ = *(infog1_+0)!=0; /* /sa/ */
if (cond_) {
    ch_mtrl(prc, infog1_);
    rg_em_1(eat->valeur, 19, 0, 20); /* prc => eat(19, 0) */
    nbtrans_++;
    bg_ = adr_m(eat);
    ch_mem1(emem, bg_, infog1_);
    rg_em_t1(32, ir->maitre); /* emem(@eat) => ir */
    nbtrans_++;
    *infog1_ = 1;
    ch_mtrl(prc, infod1_);
    add1_();
    rg_em_t1(20, prc->maitre); /* 1+prc => prc */
    nbtrans_++;
}

ch_sl(*sual, infog1_); /* /sual/ */
cond_ = *(infog1_+0)!=0;
if (cond_) {
    ch_m1(ir->esclave, 23, 20, 4, infog1_);
    rg_em_t1(4, rat1->valeur); /* ireg1 => rat1 */
    nbtrans_++;
    ch_m1(ir->esclave, 19, 16, 4, infog1_);
    rg_em_t1(4, rat2->valeur); /* ireg2 => rat2 */
    nbtrans_++;
    bg_ = adr_m(rat1);
    ch_mem1(rmem, bg_, infog1_);
    rg_em_t1(32, bus1->valeur); /* rmem(@rat1) => bus1 */
    bus1->fin=clock_+3; nbtrans_++;
    bg_ = adr_m(rat2);
    ch_mem1(rmem, bg_, infog1_);
    rg_em_t1(32, bus2->valeur); /* rmem(@rat2) => bus2 */
    bus2->fin=clock_+3; nbtrans_++;
}

/* le reste de la procedure utilise les memes sous-programmes */

}

F3_2(dadd, bus1, bus2, cact, dadr, fr, dsub, dsur, dou, det, dex, ddif, dded, ddeg, ddar, dcom, dinc, ddec, dpas)
es_ *dadd, *dadr, *dsub, *dsur, *dou, *det, *dex, *ddif, *dded, *ddeg, *ddar, *dcom, *dinc, *ddec, *dpas;
rm_ *fr;
tm_ *bus1, *bus2, *cact;
ch_sl(*dadd, infog1_);
cond_ = *(infog1_+0)!=0; /* /dadd/ */
if (cond_) {
    ch_mt1(bus1, infog1_);
    ch_mt1(bus2, infod1_);
    add1_();
    rg_em_t1(32, cact->valeur); /* bus1 + bus2 => cact */
    cact->fin=clock_+1; nbtrans_++;
}

ch_sl(*dadr, infog1_);
cond_ = *(infog1_+0)!=0; /* /dadr/ */
if (cond_) {

```

```

ch_mt1(bus1,infog1_);
ch_mt1(bus2,infod1_);
add1_();
ch_m1(fr->esclave,3,3,1,infod1_);
add1_();
rg_em_t1(32,cact->valeur);          /* bus1 + bus2 + r => cact */
cact->fin=clock_+1;nbtrans_++;
}

/* le reste de la procedure utilise le meme principe */

}
F3_3(dzer, fr, dtzs, cact, dtd, bus1, bus2, cdn, cdp, dtr)
es_ *dzer, *dtzs, *dtd, *cdn, *cdp, *dtr;
rm_ *fr;
tm_ *cact, *bus1, *bus2;
{ch_sl(*dzer, infog1_);
cond_ = *(infog1_+0)!=0; /* /dzer/ */
if (cond_) {
    *infog1_ = 0;
    rg_em_1(fr->maitre, 3, 3, 1); /* 0 => r */
    nbtrans_++;
    *infog1_ = 0;
    rg_em_1(fr->maitre, 2, 2, 1); /* 0 => s */
    nbtrans_++;
    *infog1_ = 0;
    rg_em_1(fr->maitre, 1, 1, 1); /* 0 => d */
    nbtrans_++;
    *infog1_ = 0;
    rg_em_1(fr->maitre, 0, 0, 1); /* 0 => z */
    nbtrans_++;
}
ch_sl(*dtzs, infog1_);
cond_ = *(infog1_+0)!=0; /* /dzer/ */
if (cond_) {
    ch_m1(cact->valeur, 31, 31, 1, infog1_);
    rg_em_1(fr->maitre, 2, 2, 1); /* cact(31) => z */
    nbtrans_++;
    ch_mt1(cact, infog1_);
    oul1_();
    negl_1();
    rg_em_1(fr->maitre, 0, 0, 1); /* cact/1 => z! */
    nbtrans_++;
}

/* le reste de la procedure utilise le meme principe */

}

F3_4(sual, cact, act)
es_ *sual;
em_ *act;
tm_ *cact;
{ch_sl(*sual, infog1_);
cond_ = *(infog1_+0)!=0; /* /sual/ */
if (cond_) {
    ch_m1(cact->valeur, 31, 0, 32, infog1_);
    rg_em_t1(32, act->valeur); /* cact(31, 0) => act */
    nbtrans_++;
}
}

```



```

retour_defaut(clock)
register int clock;
|register int i,j;
  if (((dcp&131070)>>1) == clock) dcp&= 131070; /* reinitialisation */
  if (((erd&131070)>>1) == clock) erd&= 131070;
  if (((ewr&131070)>>1) == clock) ewr&= 131070;
  if (((cdn&131070)>>1) == clock) cdn&= 131070;
  if (((cdp&131070)>>1) == clock) cdp&= 131070; /* de la valeur */
  if (bus1.fin == clock) bus1.valeur[0] = 0;
  if (bus2.fin == clock) bus2.valeur[0] = 0;
  if (cact.fin == clock) cact.valeur[0] = 0; /* des terminaux */
  ir.esclave[0] = ir.maitre[0]; /* traitement des */
  re.esclave[0] = re.maitre[0];
  fr.esclave[0] = fr.maitre[0];
  prc.esclave[0] = prc.maitre[0]; /* registres maitre-esclave */
|

/* affichage du contenu de chaque element de la description: */

afficher_valeurs()
|printf("%s %s %d \n","dcp"," ",dcp&1);
  printf("%s %s %d \n","erd"," ",erd&1);
  printf("%s %s %d \n","ewr"," ",ewr&1);
  printf("%s %s %d \n","cdn"," ",cdn&1);
  printf("%s %s %d \n","cdp"," ",cdp&1);
  printf("%s %s %d \n","sa"," ",sa);
  printf("%s %s %d \n","sual"," ",sual);
  printf("%s %s %d \n","scon"," ",scon);

/* meme principe pour les autres elements a un seul bit */

aff_val(bus1.valeur, "bus1", 1, 32);
aff_val(bus2.valeur, "bus2", 1, 32);
aff_val(cact.valeur, "cact", 1, 32);
aff_val(ir.esclave, "ir", 1, 32);
aff_val(re.esclave, "re", 1, 2);
aff_val(fr.esclave, "fr", 1, 4);
aff_val(prc.esclave, "prc", 1, 20);
aff_val(act.valeur, "act", 1, 32);
aff_val(rat1.valeur, "rat1", 1, 4);
aff_val(rat2.valeur, "rat2", 1, 4);
aff_val(eat.valeur, "eat", 1, 20);
aff_val(rmem, "rmem", 16, 32);
aff_val(emem, "emem", 4096, 32);
|

```

/* procedure de lecture de la valeur d'un element, utilisee lors de la lecture du fichier de donnees: */

```

traiter_1nom(c, nom, fp)
FILE *fp;
char c, nom[];
{char s1[maxlong_nom], car, base;
register int nomot, k, i, nb;
int puit;
base=lire_base(c);
switch(nocas_(nom)) { /* cette fonction du module compile renvoie l'indice du nom dans la table des symboles */
    case 0 :if (nbphases_)
        if (fp->_cnt) return;
        else fscanf(fp, "%d", &puir);
        else fscanf(fp, "%d", &nbphases_);
        break;
    case 1:fscanf(fp, "%c", &car);
        dcp|=car-48;
        break;
    case 2:fscanf(fp, "%c", &car);
        erd|=car-48;
        break;

```

/* meme principe pour les autres elements a un seul bit */

```

case 31: switch(base) {
    case 'd':fscanf(fp, "%d", bus1.valeur[0]);break;
    case 'u':fscanf(fp, "%u", bus1.valeur[0]);break;
    case 'b':
    case 'o':
    case 'x':
    case 'a': fscanf(fp, "%|'0"1"2"3"4"5"6"7"8"9"A"B"C"D"E"F|", tmp_);
        k=strlen(tmp_);
        switch(base) {
            case 'b':aff_b(bus1.valeur, k);break;
            case 'o':aff_o(bus1.valeur, k);break;
            case 'x':aff_x(bus1.valeur, k);break;
            case 'a':aff_a(bus1.valeur, k);break;
        }
    }
    break;

```

/* meme principe pour les autres elements */

|

- [AA86] Auletta R., Aylor J.
"A Non-Interpreted Model for Digital System Design"
 Proc. IEEE Southeastcon 86, pp 296-299. 55
- [AB89] Amann R., Baitinger U.
"Optimal state chains and state codes in finite state machines"
 IEEE Transactions on CAD, Vol. 8, Feb. 1989, pp 153-170. 68
- [ADN90] Ashar P., Devadas S., Newton A.
"A unified approach to the decomposition and re-decomposition of sequential machines",
 Proc. 27th ACM/IEEE Design Automation Conference, 1990, pp 601-606. 68
- [ADW59] Altman G., De Campo L., Warburton C.
"Automation of computer panel wiring"
 IBM Corp., Technical Report no TR 00.1100.689, july 1959. 7
- [Arm89] Armstrong J.
"Modeling with VHDL"
 Prentice Hall, Englewood Cliffs, NJ 07632, 1989. 62
- [Bak88] Bakowski P.
"Design analysis at RT level using correlated representation methods"
 CHDL '88, pp259-265, 1988. 86
- [Bak89] Bakowski P.
"Mode d'emploi du LIDO"
 polycopié d'A.M.I., 1989. 76
- [Bar81] Barbacci M.
"Instruction Set Processor Specifications (ISPS) : The notation and its applications"
 IEEE Transactions on Computers, Jan. 1981, pp 24-40. 32
- [BD*90] Beck K., Dubois J-L., Liu F., Pawlak A.
"VHDL model of a SUPRENUM supercomputer arbiter"
 15ème ESPRIT cave workshop, CAO for VLSI in Europe, mai 90. 45
- [BF85] Borrione D., Le Faou C.
"Overview of the cascade multi-level description language and its mixed-mode simulation mechanisms"
 Proc. CHDLs and their applications, Tokyo 1985, pp 239-260. 40
- [BN70] Bell C., Newell A.
"The PMS and ISP descriptive systems for computer structures"
 Proc. AFIPS, Vol. 36 pp 351-374, AFIPS Press, Reston, Virginia, 1970. 33

- [Bon85] **Bond J.**
"Design tool pace custom/semicustom IC development"
 Computer Design, march 1985, pp 91-101. 44
- [BP87] **Bakowski P., Pawlak A.**
"LIDO - a silicon compiler preprocessor"
 Euromicro '87, pp 167-171, 1987. 84
- [Bry81] **Bryant R.**
"MOSSIM : a switch level simulator for MOS-LSI"
 Proc. 18th ACM/IEEE Design Automation Conference, 1981. 31
- [BS*89] **D. Blaauw, D. Saab, R. Mueller-Thuns, J. Abraham, J. Rahmeh**
"automatic generation of behavioral models from switch level descriptions"
 Proc. 26th ACM/IEEE Design Automation Conference, June 1989. 55
- [CGK58] **Case P., Graff H., Kloomok M.**
"The recording, checking and printing of logic diagrams"
 Proc. Eastern Joint Conf., 1958, pp 108-118. 7
- [CM88] **Chen K-C., Muroga S.**
"Input assignment algorithm for decoded PLA's with multi-input decoders"
 Proc. ICCAD '88, pp 474-477. 68
- [Coe89] **Coelho D.**
"The VHDL handbook"
 Kluwer Academic Publishers, Norwell, MA 02061, 1989. 24
- [CW85] **Compasano R., Rosenstiel W.**
"A design environment for the synthesis of integrated circuits"
 Proc. of euromicro, 1985, pp 211-215. 44
- [DB85] **Dally W., Bryant R.**
"a hardware architecture for switch level simulation"
 IEEE Transactions on Computer Aided Design, pp 239-250, July 1985. 54
- [DBP90] **Dubois J-L., Bakowski P., Pawlak A.**
"Synthesis of Programmable Control Structures for a Simulation Speed-Up"
 Euromicro '90, pp 223-230, 1990. 62
- [De_88] **Devadas S.**
"Decomposition and factorization of sequential finite state machines"
 Proc. ICCAD '88, pp 148-151. 68

- [De88] Devadas S.
"Boolean decomposition in multi-level logic optimization"
 Proc. ICCAD'88, pp 290-293. 68
- [De89] Devadas S.
"General decomposition of sequential machines : relationships to state assignment"
 Proc. 26th ACM/IEEE Design Automation Conference, 1989, pp 314-320. 68
- [DeM84] De Micheli G.
"Optimal encoding of control logic"
 Proc. ICCAD'84, pp 16-19. 68
- [Den82] Denneau M.
"The Yorktown Simulation Engine"
 Proc. of 19th Design Automation Conference, June 1982, pp 55-59. 53
- [DRB82] Denyer P., Renshaw D., Bergman N.
"A silicon compiler for VLSI signal processors"
 Proc. of the European Solid State Circuits conference, 1982. 44
- [Dub90] Dubois J-L.
"Standard component modelling in VHDL - requirements, conventions and coding practices"
 Rapport interne à GMD, 5205 Sankt Augustin 1, R.F.A., 1990. 45
- [Dub91] Dubois J-L., Pawlak A.
"A methodology for standard component models definition"
 Proc. VHDL Forum for CAD in Europe, Avril 1991, Marseille. 45
- [Dun84] Dunn L.
"IBM's engineering design system support for VLSI design and verification"
 IEEE Design & Test of Computers, Feb. 1984, pp 30-40. 53
- [GEF84] Gonauser M., Egger F., Frantz D.
"SMILE - A multilevel simulation system"
 Proc. of ICCD'84, 1984. 41
- [GK83] Gaijski D., Kuhn R.
"Guest Editor's Introduction : New VLSI Tools"
 Computer, Vol. 16, N. 12, pp11-14, 1983. 30
- [Gro84] Groves S.
"A comparison of the internal architectures of the Motorola MC68000, 68881 and 68020"
 Proc. ICCD'84, pp 211-216. 131

- [HA*89] Hady F., Aylor J., Williams R., Waxman R.
"Uninterpreted modeling using the VHSIC Hardware Description Language (VHDL)"
 Proc. ICCAD 89, pp 172-175. 55
- [HQ88] Huertas J., Quintana J.
"A new method for the efficient state-assignment of PLA-based sequential machines"
 Proc. ICCAD '88, pp 156-160. 68
- [Hur84] Hurst S.
"Multiple-Valued Logic - Its Status and Its Future"
 IEEE transactions on computers, Vol. c-33, No 12, Dec. 1984. 23
- [Joh84] Johnson S.
"VLSI circuit design reaches the level of architectural description"
 Electronics, may 1984. 44
- [KK78] Kleinbaum D., Kupper L.
"Applied Regression Analysis and Other Multivariable Methods"
 Duxbury Press, North Scituate, Massachussetts, 1978. 120
- [KS*85] Kroening K., Schulz P., Reusch B., Bonnenberg H., Bormann R., Paul W.
"DACAPO III : a new approach to hardware simulation"
 Proc. of ICCD '85. 41
- [KT81] Kerkhoff H., Tervoert M.
"Multiple-Valued Logic Charge-Couple Devices"
 IEEE transactions on computers, Vol. C-30, No 9, Sept. 1981." 23
- [KW86] J. Katzenelson, E. Wietz
"VLSI simulation and data abstractions"
 IEEE Transactions on Computer Aided Design, pp 371-378, July 1986. 55
- [LE89] Levy H., Eckhouse R.
"Computer programming and Architecture : The VAX, 2nd ed."
 Digital Press, Bedford, Massachussetts, pp 358-372, 1989. 129
- [Lem88] Lemkin P.
"PSAIL: a portable Sail to C compiler - description and tutorial"
 SIGPLAN notices, Vol 23, No 11, pp 149-169, April 88. 57
- [Leu89] Leung S.
"Behavioral modeling of transmission gates in VHDL"
 26th ACM/IEEE Design Automation Conference, 1989. 41

- [McK69] McKey A.
"comment on Computer Aided Design : simulation of digital design logic"
 IEEE transactions on computer, Sept. 1969, pp 862. 53
- [MM*86] Miczo A., Mohapatra D., Perkins S., Kaufman K., Huang K.
"The effect of modeling on simulation performance" (Schlumberger BEHAVE System)
 IEEE Design & Test of Computers, April 86, pp 46-54. 57
- [Pat83] Patterson D.
"La microprogrammation"
 Pour la Science, Mai 1983, pp 50-59. 129
- [PH90] Patterson D., Hennessy J.
"Computer Architecture : A Quantitative Approach"
 Morgan Kaufmann Publishers Inc., San Mateo, California, \$2.2, 1990. 116
- [PP89] Paulin P.
"Horizontal partitioning of PLA-based finite state machines"
 Proc. 26th ACM/IEEE Design Automation Conference, 1989, pp 333-338. 68
- [PS82] Patterson D., Séquin C.
"A VLSI RISC"
 Computer, Sept. 82, pp 8-22. 127
- [RA87] M. Roumeliotis, J. Armstrong
"HDL modeling for process oriented simulation"
 Proc. CHDL and their applications 1987, IFIP, pp 1-8. 54
- [Ram80] Rammig F.
"preliminary CAP/DSDL Language Reference Manual"
 Université de Dortmund, 1980. 40
- [RSV88] Rammig F., Schrewe M., Vorloepen G.
"a transputer-based accelerator for multilevel digital simulation"
 Proc. Euromicro, North-Holland, 1988, pp 299-305. 54
- [Sas83] Sasaki T. et al.
"HAL: a block level hardware logic simulator"
 Proc. 20th Design Automation Conference, June 1983, pp 150-156. 53
- [SG86] Siegel B., Gross T.
"Program-specific and architecture-specific simulators"
 Department of computer science, Carnegie Mellon University, PA 15213, 1986.
 57

- [SK88] Saldanka A., Katz R.
"PLA optimization using output encoding"
 Proc. ICCAD'88, pp 478-481. 68
- [Smi81] Smith K.
"The Prospects for Multivalued Logic: A Technology and Applications View"
 IEEE transactions on computers, Vol. C-30, No 9, Sept. 1981. 23
- [SSG82] Siskind J., Southand J., Crouch K.
"Generating custom high performance VLSI designs form succinct algorithm descriptions"
 Proc. Advanced Research in VLSI, 1982, pp 28-40. 44
- [Tho86] Thomas D.,
"The RTL simulation"
 Computer Design, Feb. 86. 57
- [Tuc67] Tucker S.
"Microprogram control for the System/360"
 IBM Systems Journal no 6, pp 222-241, 1967. 129
- [VHD87] Intermetrics
"Standard VHDL language reference manual"
 Std. 1076-1987. 41
- [VL80] Vladimirescu A., Liu S.
"The simulation of MOS integrated circuits using SPICE 2"
 Memorandum VCB/ERLM 80/7, University of California, Berkeley, 1980. 30
- [VUM88] Intermetrics
User's manual for the standard VHDL 1076 support environment
 pp 1, Sep. 88. 57
- [W53] Wilkes M.
"The best way to design an automatic calculating machine"
 Manchester University Computer Inaugural Conference, 1951,
 (publiée en 1953). 129
- [WC88] Wey C-L., Chang T-Y.
"PLAYGROUNG : minimization of PLAs with mixed ground true outputs"
 Proc. 25th ACM/IEEE Design Automation Conference, 1988, pp 421-426. 68
- [WC90] Wey C-L., Chang T-Y.
"An efficient output phase assignment for PLA minimization"
 IEEE Transactions on Computer-Aided Design, Vol 9, Jan. 1990, pp 1-7. 68

[WT53] Wilkes M., Struger J.

"Microprogramming and the design of the control circuits in an electronic digital computer"

Proc. Cambridge Philosophical Society, no 49, pp 230-238, 1953. 129

[YC89] Yang S., Ciesielski M.

"PLA decomposition with generalized decoders"

Proc. ICCAD'89, pp 312-315. 68

[DuP91] Dubois J-L., Pawlak A.

"A technique for generating efficient simulators"

ICCD'91 (VLSI in computers and processors), paper n° 174.



Résumé

L'accroissement incessant de complexité des circuits nécessite la définition de méthodes et d'outils, en particulier de simulation, toujours plus puissants.

La génération de simulateurs, c'est-à-dire la traduction de la description d'un circuit en un programme écrit dans un langage de haut niveau, permet d'ajouter la portabilité à l'avantage majeur des méthodes de compilation qui est l'efficacité.

L'inconvénient de cette technique réside dans l'importance de la taille des simulateurs produits et est dû à l'emploi du schéma trop classique de traduction d'un algorithme en un autre. Ce problème est évité grâce à la transformation des parties contrôles de circuits avant la génération. Le modèle résultant est alors constitué d'une partie contrôle essentiellement sous forme de données et d'une partie opérative sous forme d'algorithme.

L'emploi de cette méthode, qui ne dépend pas du langage de description utilisé, permet à la fois la réduction de la taille des simulateurs et l'accélération de leur exécution.

Abstract

As the complexity of today's chips steadily increases, more and more powerful methods and tools are required, especially simulation tools.

The automatic generation of simulators, i.e. the translation of a circuit's description into a program written using a high level language, allows for the combining of portability and efficiency, the last topic being a main feature of compilation methods.

The problem with this technique consists in a too large size of generated simulators. This drawback is mainly due to the traditional translation scheme *algorithm to algorithm*. This problem is avoided using a transformation method of circuits' control part before the production of any simulator. Thus, a resulting high level model is composed of code and data tables implementing the control part and algorithms implementing the operative one.

This method, which is not dedicated to a particular Hardware Description Language, allows us to reduce a simulator's size and to speed-up its execution.

Mots clés:

SIMULATION
ACCELERATION
MODELISATION
CAO
ABSTRACTION FONCTIONNELLE
PARTIES CONTROLES
GRAPHES
PLAS