FIDEL : un langage de description et de simulation des circuits VLSI

Hazem El Tahawy

THESE

présentée par

EL TAHAWY HAZEM

pour obtenir le titre de **DOCTEUR**

de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**
(arrêté ministériel du 5 juillet 1984)

(Spécialité : **Informatique**)

FIDEL
Un Langage de Description et de Simulation des Circuits VLSI

Date de soutenance : Le 23 Novembre 1987.

Composition du Jury :

Mr. J. MOSSIERE	**Président**
Mme. D. BORRIONE	**Examinateurs**
Mr. R. GERBER	
Mr. C. LANDRAULT	
Mr. G. MAZARE	
Mr. A. PUISSOCHET	

Thèse préparée au sein du laboratoire : IMAG/LGI.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année 1987

Président : Georges LESPINARD

Vice-Présidents : Jean-Marie PIERRARD
Jean-Pierre VERJUS

Professeurs des Universités

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

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET	Louis

Personnes ayant obtenu le diplôme

d'HABILITATION A DIRIGER DES RECHERCHES

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

Chercheurs du C.N.R.S

Directeurs de recherche 1ère Classe

CAILLET	Marcel	JORRAND	Philippe
CARRE	René	LANDAU	Ioan
FRUCHART	Robert	MARTIN	

Directeurs de recherche 2ème Classe

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

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

E.N.S.E.E.G

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

E.N.S.E.R.G

BOREL	Joseph

E.N.S.I.E.G

DESCHIZEAUX	Pierre	PERARD	Jacques
GLANGEAUD	François	REINISCH	Raymond

E.N.S.H.G

BOIS	Daniel	ROWE	Alain
		VAUCLIN	Michel
MICHEL	Jean-Marie		

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

BERT	Didier		
COURTIN	Jacques		
COURTOIS	Bernard	STEAKIS	Joseph

I.U.P. G

CHARUEL	Robert

C.E.N.G

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

Laboratoires extérieurs

C.N.E.T

DEMOULIN	Eric	MERCKEL	Gérard
DEVINE		PAULEAU	Yves
GERBER	Roland		

****

A mes parents,
A mes beaux parents,
A ma femme Mervat,
A ma fille Omnia.

Je tiens à exprimer toute ma reconnaissance

à Mr. Guy MAZARE, Professeur à l'ENSIMAG, pour m'avoir accueilli dans son groupe de recherche ainsi que pour ses remarques tout au long de cette thèse.

Je tiens à remercier

Mr. le professeur Jacques MOSSIERE , directeur du laboratoire Génie Informatique de l'IMAG, de me faire l'honneur de présider le jury de cette thèse,

Mme Dominique BORRIONE, Professeur à l'université de Provence pour avoir accepté d'être rapporteur de ce travail et m'avoir fait profiter de ses connaissances dans le domaine des langages HDL au cours des discussions que nous avons pu avoir,

Mr. le professeur Roland GERBER , directeur de division Conception de Circuits Intégrés au Centre National d'Etudes des Télécommunications de Grenoble , pour m'avoir acueilli dans sa division ainsi que pour avoir accepté d'être rapporteur de cette thèse.

Je tiens à remercier aussi

Mr. Christian LANDRAULT du Laboratoire d'Automatique et de Microélectronique de Montpellier,

Mr. Alain PUISSOCHET , Ingénieur en CAO du Centre National d'Etudes des Télécommunications de Grenoble

pour avoir accepté d'être membres de ce jury.

Je ne saurais oublier dans ces remerciements Jacques LECOURVOISIER, responsable de departement de Recherche en Conception Assistée , qui m'a assuré les meilleures conditions et m'a encouragé pendant mon travail , Jean-louis LARDY, responsable de departement de Méthodologie de Conception des Circuits ,pour ses remarques concernant la révision de ma thèse, Michel POIZE ,Ingénieur en CAO , pour m'avoir aussi beaucoup aider pendant la révision de cette thèse. Ces derniers travaillent tous au Centre National d'Etudes des Télécommunication de Grenoble , qu'ils trouvent ici l'expression de ma profonde sympathie.

Je n'oublie pas de remercier Mr. Michel DELAUNAY, Ingénieur à l'IMAG, qui m'a beaucoup aidé pour la construction de la grammaire de FIDEL et l'utilisation de son transformateur de grammaire.

Il m'est impossible de citer ici tous les membres du Centre National d'Etudes des Télécommunications qui m'ont aidé pendant ma thèse pourtant je tiens à leur exprimer ma profonde sympathie.

Je tiens enfin à remercier le service de reprographie de l'IMAG pour avoir assuré le tirage de cette thèse.

FIDEL : LANGAGE DE DESCRIPTION ET DE SIMULATION DES CIRCUITS VLSI

RESUME

Cette thèse discute dans un premier temps des propriétés et des concepts des langages de description de matériel HDL, ceci nous a permis de mettre l'accent sur des points importants que nous avons pris en considération lors de la définition et l'implémentation de notre travail.

Ensuite, le langage FIDEL pour la description (fonctionnelle et structurelle) et la simulation de circuits intégrés VLSI est présenté, en insistant sur les différentes caractéristiques de ce langage qui sont adaptées à une simulation hiérarchique et multi niveaux.

Deux outils de simulation, logico-fonctionnelle et électrico-fonctionnelle, sont présentés. Ces deux outils présentent une avancée dans le domaine de la simulation, dans le but de garder la précision tout en diminuant le coût de simulation des circuits VLSI.

Une évaluation des différents langages de description selon leurs domaine d'application et propriétés est présentée. Au vue de cette évaluation, FIDEL s'insére en bonne place, tant au niveau des concepts que l'utilisation pratique.

MOTS-CLEFS

Langages de description du matériel, HDL, description fonctionnelle, description structurelle, description hiérarchique, simulation, simulation fonctionnelle, simulation logique, simulation électrique, simulation multi niveaux, simulation de mode mixte, système de CAO, dispositif MOS, VLSI.

ABSTRACT

This thesis discusses some concepts and properties of Hardware Description Language (HDL) ; this discussion allowed us to precise some important points which we have been taken into consideration within the definition and implementation of our work.

The HDL FIDEL is then presented. This language allows the description (functional and structural) and the simulation of the VLSI circuits. The basic characteristics of the language are conformed to a hierarchical multi level simulation.

Two simulation tools built around FIDEL are presented. These tools present an advance in the domain with the goal to keep the precision while decreasing the cost of the simulation of VLSI circuits.

An evaluation of the different description languages according to their properties and appplications is presented. As a result of this evaluation, FIDEL is a good challenge through its concepts and practical usages.

KEY-WORDS

Hardware Description Lanuages, HDL, functional description, structural description, hierarchical description, simulation, functional simulation, logic simulation, circuit simulation , multi level simulation, mixed mode simulation, CAD systems, MOS devices , VLSI

TABLE DES MATIERES

TABLES DES MATIERES

INTRODUCTION

Le travail présenté dans cette thèse traite à la fois la définition et l'implémentation d'un langage de description de matériel, FIDEL, ainsi que les applications de simulation réalisées autour de ce langage.

Le langage a été réalisé dans le but de servir à la fois la description et la simulation dans le système CASSIOPEE [LEC 82] de la conception assistée (CAO) de circuits intégrés au CNET. Une description fonctionnelle FIDEL est présentée comme une boite noire et peut être interfacer avec les différentes types de simulateurs dans CASSIOPEE (logique et électrique).

Dans le chapitre I, nous allons d'abord présenter le processus de la conception dans la méthodologie dite "descendante". Ensuite, nous allons discuter la nécessité du système de la conception assistée, en présentant le système CASSIOPEE comme exemple.

Les trois applications principales d'un tel système sont : la documentation, la vérification et la synthèse. Ces trois applications sont basées sur l'utilisation de langages de description de matériel.

Dans les deux dernières parties de ce chapitr, le langage de description sera le thème principale. D'abord, nous allons discuter en détail certaines propriétés qui doivent être exister d'un tel langage de description. Ces propriétés seront prise en compte tout au long de cette thèse , lors de la synthèse de deux approches de langages de description : CONLAN et VHDL et lors la définition du langage FIDEL.

Le chapitre II sera consacré à la présentation du langage FIDEL (aspect fonctionnel). Cet aspect est basé sur la définition d'une description fonctionnelle comme intermédiaire entre une description comportementale et une description structurelle. En ce sens les composants matériels d'un système logique sont représentés, mais que les opérateurs peuvent ou non correspondre à des opérateurs matériels.

Par contre, le parallélisme et les aspects temporels sont prise en considération. Donc, FIDEL est un langage permettant la description du comportement et/ou de la structure d'un circuit intégré.

Dans ce chapitre, la présentation d'une description fonctionnelle FIDEL sera basée sur un ensemble d'éléments de base pour décrire le système logique. Cet ensemble peut être utilisé comme mesure d'efficacité d'un langage de description.

Le chapitre 3 est consacré à la discussion des outils réalisés autour du langage fonctionnel FIDEL. Il s'agit notamment des applications de simulation : Simulation logici-fonctionnelle et électrico-fonctionnelle.

La première application concernant l'intégration du modèle fonctionnel FIDEL dans le simulateur logique EPILOG qui aboutit au type de simulation de niveau mixte. Ce type de simulation est fait pour répondre au besoin aux concepteurs au CNET et THOMSON-EFCIS.

La deuxième application concernant l'intégration du modèle fonctionnel FIDEL dans le simulateur électrique ELDO qui aboutit au type de simulation de mode mixte. Ce type de simulation réponde bien au besoin pour simuler des grosses circuits en général et les circuits de télécommunication spécialement. Ces deux applications sont une avancée dans le domaine de simulation dans le but de garder la précision tout en diminuant le coût de simulation des circuits VLSI.

La description hiérarchique est une propriété importante pour le langage de description. Cette hiérarchie (interne et externe) facilite la tâche de description et la construction d'un simulateur multi niveau. La discussion de cette propriété avec la réalisation d'un simulateur hiérarchique et multi niveau seront présentées au chapitre IV. Nous allons aussi présenter dans ce chapitre la définition du modèle structural et l'environnement de scénarios de simulation en FIDEL (stand-alone).

Nous allons terminer ce rapport par le chapitre V concernant une étude d'évaluation de langages de description. Le but de cette étude est :

- de dresser un bilan concernant l'état de l'art des langages de description de matériel.
- d'évaluer FIDEL vis à vis des autres langages.

Le langage FIDEL a été réaliser dans le but de fournir aux concepteurs un outil de description et de simulation efficace et facile à manipuler.

CHAPITRE I

DESCRIPTION ET CONCEPTION DES SYSTEMES LOGIQUES

INTRODUCTION

Aujourd'hui, à cause de la complexité croissante des circuits intégrés, tout un potentiel de recherche se mobilise dans le domaine de la Conception Assistée par Ordinateur ou de la CAO (Computer Aided Design - CAD). En effet, cette complexité, qui s'accompagne d'une diminution de la surface des circuits, ne peut plus être maîtrisée qu' avec la puissance de l'informatique.

Toutes les étapes du processus de la conception sont concernées et utilisent d'ores et déjà des outils informatiques : définition architecturale, évaluation et synthèse logique, prise en compte des contraintes d'implantation, définition du schéma électrique, jusqu' à la génération des masques.

Dans ce chapitre, nous allons essayer de mettre l'accent sur les sujets suivants : la conception, la conception assistée par ordinateur, les outils de la conception assistée par ordinateur et les langages de description de matériel.

Dans la première partie de ce chapitre, nous présenterons les étapes du processus de la conception et les trois types de circuits (prédiffusé, précaractérisé et circuit à la demande) avec les outils de CAO qu' ils utilisent. A la fin de cette partie, nous allons mentionner la synthèse logique et la compilation de silicium comme un système complet de conception automatisée.

Dans la seconde partie nous présenterons la méthode de conception dite "descendante", en expliquant pourquoi cette méthode est plus utilisable que la méthode dite "ascendante". Dans cette partie, nous introduirons également les trois domaines dé la description de circuits (comportementale, structurelle et physique) et les différents niveaux de description associés.

La présentation du système d'aide à la conception de circuits VLSI sera faite dans la troisième partie. La nécessité d'un tel système intégré d'aide à la conception sera discutée. Nous présenterons les trois classes principales d'outils qui doivent être intégrées dans un système de CAO (les outils de synthèse, les outils d'analyse et les outils de gestion de données). Enfin, nous présenterons le système CASSIOPEE [LEC 82], qui est en développement au CNET Grenoble, comme un exemple de système d'aide à la conception.

Après la présentation du système de CAO, nous pouvons dire que les trois applications principales d'un tel système sont : la synthèse, la vérification et la documentation. Ces applications sont basés sur l'utilisation de langages de description de matériel (Hardware Description Language - HDL), qui sera le thème principal de notre présentation à partir de la quatrième partie jusqu' à la fin de ce chapitre.

Dans la quatrième partie, nous discuterons de l'importance de ces langages de description, nous présenterons les propriétés principales qui doivent exister pour effectuer de façon propre et sûre la description du système logique. Aussi, dans cette partie nous présenterons les différentes applications des langages de description. Enfin, nous discuterons les trois approches pour la présentation d'un langage de description (l'approche de langages dédiés, l'approche de famille de langages et l'approche d'un langage commun), en discutant les projets CONLAN [BOR 81], [PIL 85] et VHDL [SHA 85] , [ROG 86] comme exemples.

1. LE PROCESSUS DE CONCEPTION:

La conception assistée par ordinateur de circuits intégrés complexes a pris un essor considérable depuis quelques années. Ceci n'est pas dû au hasard, mais correspond bien à la nécessité de diminuer les coûts de production en concevant vite et bien, de façon sûre et systématique. L'augmentation de la fiabilité dans la conception d'un circuit intégré passe forcément par la réduction de l'intervention humaine, et donc par l'utilisation d'outils informatiques offrant aux concepteurs une assistance efficace. Le facteur de rapidité est également primordial.

L'aboutissement de la conception d'un circuit intégré consiste en la définition et la description complètes du circuit en terme d'éléments géométriques à partir de ses spécifications externes. Cette description permet la réalisation des différents masques intervenant dans la fabrication du circuit.

Définition :

Une primitive de description est un objet élémentaire bien défini (fonctionnement, structure) à un niveau de description donné qui ne peut pas et ne doit pas être redéfini.

DOMAINES

NIVEAUX	Comportemental	Structurel	Physique
	Système	Mémoires, processeurs bus.....	Partition physique
	Algorithme	Modules matériels	Assemblage de blocs
	Fonctionnel	registres, UAL, MUXs, ...	Plan masse
	Logique	Portes , bascules, latches	Cellule
	Switch	Transistors	Rectangle
	Eléctrique	Transistors	Rectangle

FIG. 1 : PLAN DE CONCEPTION

Définition :

Un niveau de description définit une étape ou un degré de détail en fonction d'un ensemble de primitives (si elles existent), des représentations de données et des informations temporelles.

Chaque niveau individuel peut être caractérisé par un langage de description ou par des éléments descriptifs. Nous allons discuter dans la deuxième partie la définition des niveaux de description et les éléments primitifs associés.

Définition :

Un domaine de description définit un type ou une vue de la description (comportemental, structurel et physique).

La discussion détaillée de chaque domaine de description sera présentée dans la deuxième partie.

L'intégration de domaines de description de manière horizontale et de niveaux de description de manière verticale donne le plan de conception, comme le montre la figure (1).

En général, le processus de conception consiste en une suite de transformations (horizontale et verticale) de la description du circuit dans le plan de conception. La figure (2) présente deux approches de transformations dans la méthode "DESCENDANTE" de conception (top-down method), que l'on va présenter dans la deuxième partie : soit on passe d'un domaine à l'autre en gardant le même niveau de hiérarchie ; soit on passe d'un domaine à l'autre mais en allant du niveau le plus haut jusqu'au niveau le plus bas de la hiérarchie de description. La conception dite "ASCENDANTE" s'effectue dans le sens inverse.

En réalité le concepteur utilise un mélange entre les deux approches avec la possibilité de retour arrière "backtracking".

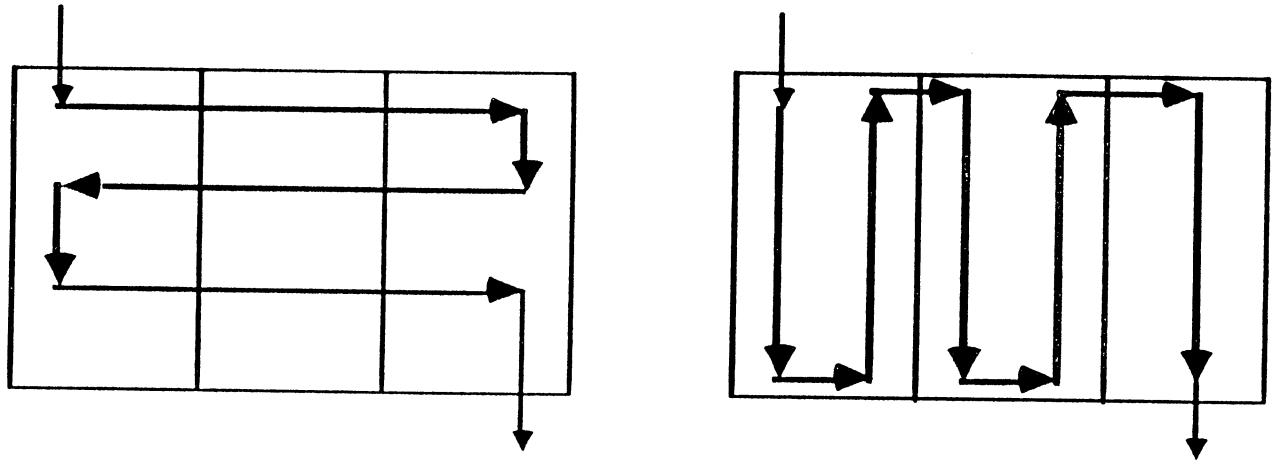


FIG . 2 : DEUX APPROCHES DE PASSAGE A TRAVERS LE PLAN DE CONCEPTION

La complexité des circuits VLSI demande une méthodologie de conception hiérarchique (c'est à dire la décomposition du circuit en sous circuits qui sont eux même décomposables et faciles à gérer). Cette méthodologie elle même nécessite une description hiérarchique du circuit. L'aspect description hiérarchique sera présenté en détail dans la quatrième partie.

Aujourd'hui, on remarque l'apparition des nouveaux types de produits que sont les circuits à application spécifique (ASIC), dont au moins une partie de la conception est spécifique de l'application. On distingue trois catégories de ces

circuits : les circuits prédiffusés (gate-array), les circuits précaractérisés (standard-cell), et les circuits entièrement conçus en fonction de l'application, couramment appelés circuits <à la demande> (full-custom). Chaque catégorie est disponible avec ses logiciels d'aide à la conception correspondants.

Avant de présenter en résumé la description de chaque catégorie de conception et les outils de CAO associés, on préfère d'abord présenter les phases principales du processus de conception qui sont les suivantes : phase d'initialisation, phase de création, phase d'implantation et phase de mise sur le marché. Eventuellement, les phases un et quatre ne sont pas effectuées par le concepteur du circuit, mais elles sont quand même liées au processus de conception du circuit.

- Phase d'initialisation :

  Dans cette phase, on commence par déterminer pourquoi on a besoin d'un nouveau circuit et quelles sont les ressources dont on a besoin pour développer ce circuit (exemple : identification de la demande commerciale, définition des nouvelles caractéristiques et combien d'"homme* année" pour la conception et la réalisation de ce circuit).

- Phase de la création :

  On commence dans cette phase par étudier les spécifications du circuit et définir les fonctions à accomplir. Ensuite on passe à la conception architecturale où l'on découpe le circuit à générer en ses parties architecturales et on associe chaque partie ou groupe de parties aux différents concepteurs de l'équipe. Après ce partage des tâches, on commence la conception détaillée en déterminant la méthode d'implantation et la méthodologie de conception à utiliser pour générer la description au niveau le plus bas. Pendant cette phase on fait les vérifications propres à chaque étape dans la hiérarchie de la description (par exemple: simulation comportementale, simulation fonctionnelle et logique).

- Phase d'implantation

  Cette phase représente la création et la réalisation du prototype du circuit (exemple : on génère les masques à partir de représentations symboliques).

Dans cette phase, on utilise aussi des outils de vérification (vérification des règles de dessin, vérification des caractéristiques électriques par la simulation électrique). La fin de cette phase est la fabrication d'un prototype dans une technologie donné et la vérification en utilisant les outils de test pour assurer la cohérence entre les spécifications initiales et la réalisation finale.

- Phase de mise sur le marché :

C'est la phase finale dans le processus de conception. Après la vérification d'un prototype on passe à sa production, sa commercialisation, sa documentation et on doit assurer la maintenance du circuit dans des environnements différents, la correction des erreurs si elles existent, et les adaptations nécessaires à certaines applications.

Il faut noter que dans certains projets le séquencement des étapes n'est pas strict. Quelques étapes peuvent être omises, d'autres peuvent être inclues et parfois, certaines peuvent être répétées.

1.1 Circuits Prédiffusés :

Un circuit prédiffusé ou réseau prédiffusé est une matrice de transistors déjà présents sur le silicium, mais non interconnectés. Seul le ou les masque(s) d'interconnexion ou de métallisation sont réalisés en fonction des besoins spécifiques de l'utilisateur. Il en résulte un délai de réalisation relativement court, un coût de développement relativement faible et une grande sûreté de conception. Par contre, c'est la catégorie de circuit qui utilise le moins bien la surface du silicium. La plupart du temps, le concepteur dispose d'importantes aides logicielles pour automatiser et rendre plus sûre la conception de son circuit. Il s'agit essentiellement d'une bibliothèque de fonctions élémentaires précaractérisées (aujourd'hui des fonctions logiques et fonctions complexes type registre, demain peut être des ROM, RAM et des processeurs), d'un logiciel de routage et de placement automatiques et d'un programme spécialisé d'édition et de génération des masques. La vérification globale du circuit se fait par le simulateur logique.

1.2 Circuits Précaractérisés :

Dans le cas d'un circuit précaractérisé, aucun élément n'est présent sur le silicium au départ, la conception se fait à partir de cellules mémorisées dans une base de données (bibliothèque). Si la bibliothèque est bien étoffée, cette technique doit permettre au concepteur de réaliser un circuit à haute intégration comportant aujourd'hui jusqu'à 60 000 portes. Les différents types de cellules peuvent être : des transistors, des portes logiques (inverseurs, ou-non, et-non ...), des fonctions (multiplexeurs, compteurs,....), des coeurs de microprocesseurs et des points mémoires. Comme pour le prédiffusé, le concepteur peut faire un assemblage et une simulation, mais la simulation dans ce cas peut être plus rapide et effectuée au niveau fonctionnel. Il dispose cependant de plus de degrés de liberté. Deux techniques de dessin prédominent : la première consiste en l'assemblage de cellules en bandes entourées de canaux de routage (standard-cell). Là encore, une grande aide est apportée par des logiciels de routage et de placement automatiques. La deuxième est fondée sur l'assemblage hiérarchique de cellules élémentaires avec un tracé automatique des connexions (opérateurs flexibles). L'efficacité de ces méthodes est liée à la richesse de la bibliothèque de cellules élémentaires, chacune étant définie pour une technologie déterminée.

1.3 Circuits à la demande :

La conception d'un circuit à la demande n'est entreprise, en principe, que lorsque le besoin correspondant est au moins de 100 000 circuits. Concevoir un circuit à la demande revient à intégrer un maximum de fonctions sur une surface donnée, le coût d'un circuit étant fortement fonction de sa surface. Dans cette méthodologie et pour un circuit logique complexe, les étapes de conception peuvent être les suivantes : après l'analyse de la fonction à intégrer et le choix de la technologie à utiliser, le concepteur commence la description du circuit au niveau algorithmique ou comportemental, la vérification du circuit à ce niveau se fait par la simulation comportementale. Ensuite, le concepteur découpe son circuit en structures de base, en recherchant la répétitivité maximale et en utilisant l'approche descendante de conception. Il effectue ensuite la simulation fonctionnelle. Une fois le schéma logique mis au point, la découpe structurelle et le plan de masse affinés, il procède à la simulation logique. Puis, en se servant d' un dessin au micron ou d'un dessin symbolique, il trace les blocs et les cellules qui ne sont pas disponibles en bibliothèque. Une extraction automatisée ou manuelle du

schéma des blocs et des éléments parasites lui permet de vérifier la connectivité et d'effectuer la simulation électrique des chemins critiques. Les masques sont alors générés après assemblage des blocs et vérification des contraintes temporelles.

On peut trouver dans le Tableau (1) [LAR 85] la comparaison entre les points principaux de ces catégories de conception .

TABLEAU 1

	Pourcentage des opérations déjà effectuées	Coût de développement	Délai (mois)	Quantinté minimale réaliste	Optimisation de place	Risque de reprise
Prédiffusé	80 à 90 %	50 à 600 KF	1 à 3	1 000 à 5 000	moyenne	en principe nul
Précaractérisé	0 %	300 à 900 KF	3 à 6	20 000	bonne	faible
A la demande	0 %	généralement plusieurs MF	6 à 30	75 000	maximale	toujours 1 à 3 reprise

1.4 Synthèse logique et compilateur de silicium :

Il est très intéressant de mentionner ici la synthèse logique et la compilation de silicium, non pas comme une nouvelle catégorie de conception mais comme un système complet de conception automatisée. La synthèse logique signifie la transformation automatique d'une description d'un circuit au niveau comportemental à une description structurelle ou logique. La combinaison de la synthèse logique avec les générateurs de blocs fonctionnels, et les méthodes de routage et placement automatiques mènent à ce que l'on peut vraiment appeler la compilation de silicium.

En général la compilation de silicium peut être classée selon le langage d'entrée sous deux formes [AND 85], [JAM 86] et [PAR 87] : fonctionnelle et structurelle. La compilation fonctionnelle est classée en deux formes : architecturale et

comportementale selon que le langage fonctionnel a une sémantique de structure implicite ou non.

Afin de terminer cette partie, il est important de noter que l'utilisation des techniques de l'Intelligence Artificielle (IA) à chaque niveau de conception pourrait rendre le développement des circuits VLSI accessible aux non-experts. L'utilisation de bases de connaissances et de systèmes experts comme composants dans un système de conception avancée, permettra un jour de prendre certaines décisions importantes pendant le processus de la conception à la place des concepteurs (mais toujours avec l'intervention de concepteur en début de phase d'initialisation et en fin de phase d'implantation).

Le futur système de conception pour les circuits VLSI est caractérisé par une combinaison efficace du processus algorithmique traditionnel, des techniques d'IA et de base de connaissances.

2. LA METHODE DESCENDANTE DE CONCEPTION

La méthode de conception décrit l'interaction entre le concepteur et les outils de la conception. Aussi, elle décrit la stratégie d'application de différentes étapes de la conception dans les différents domaines et aux différents niveaux de description (plan de conception, cf figure 1).

On peut dire que la méthode de la conception descendante est une démarche qui permet de décomposer une description complexe du comportement en sous-algorithmes décrivant des comportements plus simples. La méthode de la conception ascendante est une démarche qui permet, à partir d'éléments physiques prédéfinis, de réaliser des structures plus complexes.

En général, la méthode descendante est préférable à la méthode ascendante pour les raisons suivantes :

- La méthode descendante est susceptible de réaliser une solution globalement optimale, tandis que la méthode ascendante tend à optimiser le niveau le plus bas et au contraire à relâcher les contraintes au niveau le plus haut.

- Dans la démarche descendante, c'est tôt dans le processus de conception que l'on peut estimer si la conception peut satisfaire les spécifications.

- Dans la plupart des étapes de conception, la méthode descendante est indépendante de la technologie.

- La méthode descendante permet la vérification de la conception et la cohérence de données à tous les niveaux.

On peut noter que la méthode descendante de conception est appliquée avec succès dans la programmation structurée et dans la conception architecturale.

2.1. Domaines de description :

La description d'un circuit peut être divisée en trois domaines principaux :

- Un domaine de description de comportement durant lequel le concepteur décompose les spécifications initiales en une série d'algorithmes de complexité décroissante.

- Un domaine de description structurelle durant lequel le concepteur essaie de réaliser un module par les interconnexions d'autres modules primitifs.

- Un domaine de description physique pendant lequel le concepteur doit décrire son circuit en tenant compte d'une technologie donnée.

Dans la majorité des cas, la description comportementale et la description structurelle sont réalisées conjointement. Par contre, la description physique est effectuée indépendamment. Toutefois, il est clair que la description physique peut avoir des influences sur la description comportementale ou structurelle en nécessitant de nombreuses itérations pendant la conception.

Il est important de noter que dans tous les cas une approche hiérarchique est utilisée. Le partitionnement induit, physique, structurel ou algorithmique, permet de réduire localement les complexités et d'assurer ainsi une conception structurée.

2.1.1 Description du comportement :

Le terme description comportementale signifie la description du comportement d'un circuit par rapport à l'ensemble des fonctions qu' il réalise. C'est la phase qui

est la plus créatrice et qui a la plus grande influence sur les résultats. La liberté de la conception dépend seulement des éléments suivants : l'ensemble de spécifications et de limites qui sont proposées pour le circuit, les applications ou par le demandeur du circuit.

Les limites typiques peuvent être : le coût, la vitesse, la dissipation d'énergie ou la surface. Les spécifications comportementales peuvent être représentées par : une fonction spécifique, un algorithme ou un ensemble d'instructions.

Les descriptions comportementales d'un microprocesseur peuvent être décomposées hiérarchiquement comme le montre la figure (3). Cette décomposition commence par l'interprétation de l'ensemble des commandes à exécuter. Ces dernières se présentent sous la forme d'instructions de format variable mais généralement composées des champs suivants : un champ code opération spécifiant le type d'action à générer, un champ adresse spécifiant le type d'opération à effectuer pour localiser les opérandes, et un champ opérande constitué d'un ou deux opérandes sur lesquels sont effectuées certaines opérations.

Il est à noter que cette décomposition comportementale est indépendante de la décomposition physique du circuit. Elle peut donc être entreprise avant que toute contrainte d'ordre physique ou technologique ne soit prise en compte.

Maintenant le processus de la conception comportementale peut être décrit formellement par des langages comportementaux ; les langages de programmation C ou PASCAL peuvent être utilisés ainsi que n'importe quel langage comportemental de description de matériel. Un langage pour la simulation et la synthèse est préférable, exemples : ISPS [BAR 81], MIMOLA [ZIM79], et DACAPO [BRU 85]. Le système SARA [EST 78] à UCLA est un des premiers systèmes à avoir adopté cette méthode de conception comportementale et hiérarchique.

Les outils de conception assistée qui peuvent être utilisés pendant cette phase de conception sont : description du comportement, vérification formelle de

spécifications comportementales et simulation de description à chaque niveau dans la hiérarchie. Aujourd'hui, la recherche se dirige vers la synthèse ou la compilation de silicium au niveau comportemental qui est connu sous le nom de compilation architecturale.

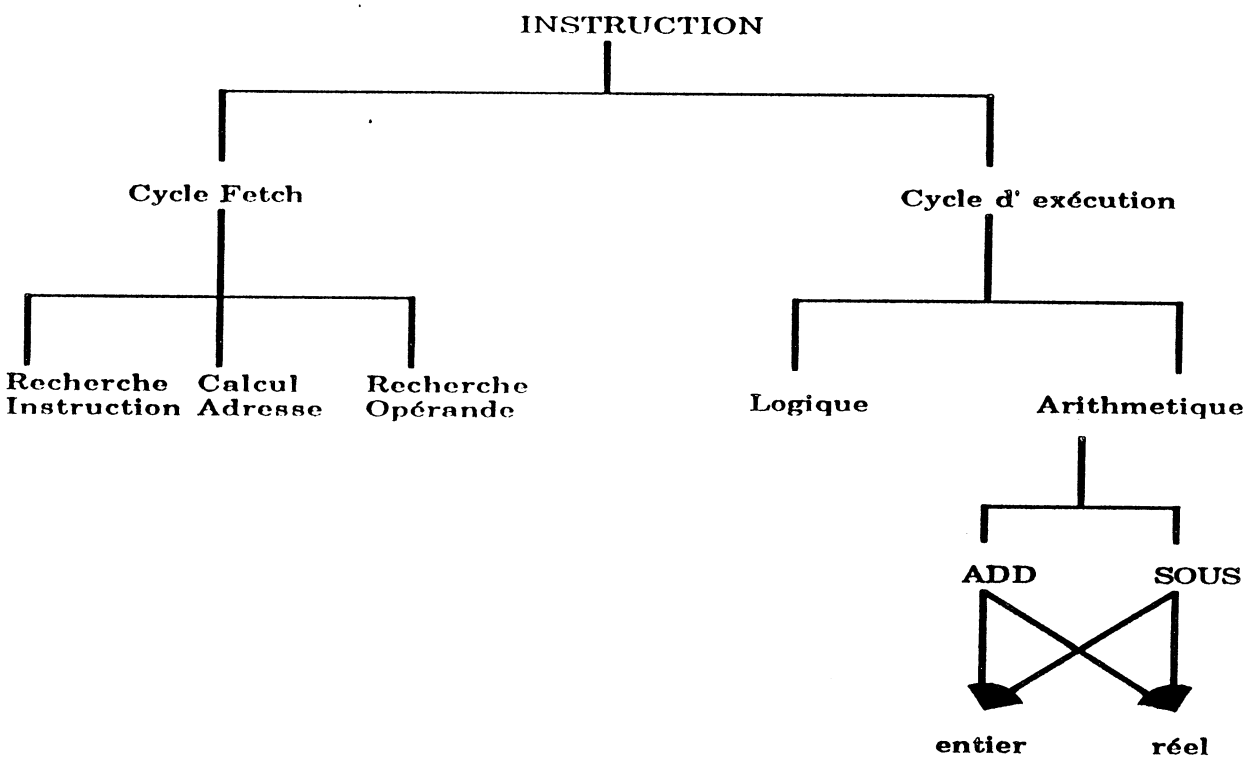


FIG . 3 : DECOMPOSITION COMPORTEMENTALE

2.1.2. Description structurelle :

Le terme description structurelle signifie la description de la structure d'un circuit en terme de modules, de blocs interconnectés. Ces descriptions structurelles sont elles aussi décomposables de manière hiérarchique, de la description de haut niveau jusqu' à la description utilisant des modules primitifs ou élémentaires.

Il y a différentes méthodes qui peuvent être utilisées pour la description structurelle, ces méthodes dépendent du niveau de hiérarchie. A tous les niveaux, la conception manuelle est encore dominante. Tant que la structure de chaque fonction au niveau le plus bas peut être réalisée, la méthodologie descendante de conception peut être appliquée. Mais, même si les spécifications fonctionnelles peuvent être

satisfaites, la question reste : à quel prix et à quelle vitesse. Les prédictions sont nécessaires.

En général, la description structurelle et la description comportementale d'un circuit sont réalisées conjointement. Ces deux descriptions sont étroitement liées car elles correspondent à deux aspects d'un même problème : comment décrire une fonction et comment décrire le support physique permettant de réaliser cette fonction.

La description structurelle d'un microprocesseur comme le montre la figure (4) est, elle aussi, réalisée de manière hiérarchique et est, par nature, modulaire. En effet, dans la mesure où des segmentations fonctionnelles sont réalisées pour réduire la complexité des algorithmes à implanter, leurs structures d'implantation sont, elles aussi, décomposées en structures plus simples.

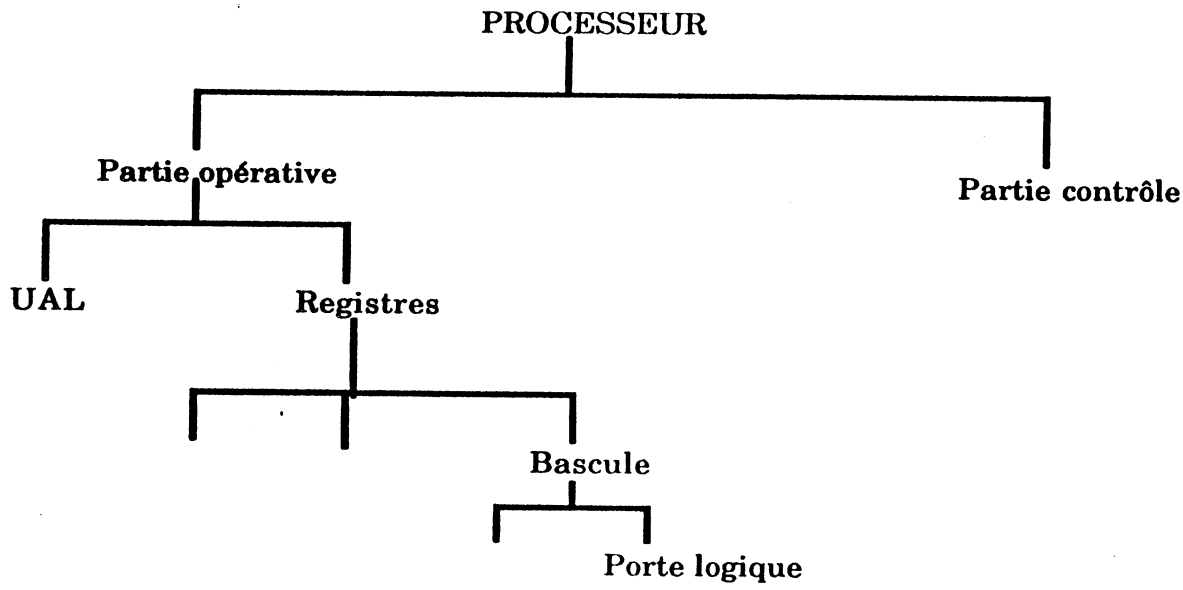


FIG. 4 : DECOMPOSITION STRUCTURELLE

Les avantages principaux d'une description structurelle hiérarchisée et modulaire sont :

- Tout d'abord, le circuit à réaliser est décrit sous la forme de modules dont les interfaces sont clairement définies. Donc, le temps de la conception peut être réduit de manière considérable et la lisibilité des descriptions en est par conséquent accrue.

- Pendant cette phase de la description on peut vérifier le comportement de la structure que l'on a proposée, cette vérification est conçue sous le terme "vérification de flux de données" (data flow verification).

- De plus, la mise en évidence de fonctions élémentaires (et donc de structures élémentaires d'implantation) favorise la standardisation de ces modules. Leur génération automatique est aujourd'hui courante. Elle permet de réduire considérablement le temps de la conception en augmentant l'aspect répétitif des descriptions structurelles.

Il y a beaucoup de systèmes fournissant tous les avantages que l'on a mentionnés ci-dessus. Par exemple, CMU-DA système [HIT 83], DAA [KOW 83], et SCALD [MCW 78]. Au CNET, le système CASSIOPÉE [LEC 82] avec le système SCHUSS [SCH 85], tient compte de tous les aspects de la description structurelle.

2.1.3. Description Physique :

Le terme description physique regroupe ici à la fois des descriptions géométriques du niveau le plus bas (rectangle) et le comportement électrique du circuit au niveau le plus fin.

La phase de la description physique est la plus consommatrice de temps pendant le processus de la conception. Par conséquent, l'utilisation des outils de CAO est d'importance primordiale afin de gérer la complexité de circuits VLSI.

La description physique d'un circuit est de manière générale réalisée par assemblage progressif d'objets physiques communément nommés CELLULES. Elle est aussi de nature hiérarchique car le concepteur construit à partir de ces cellules élémentaires des structures physiques plus complexes jusqu' à générer la description physique totale du circuit, comme le montre la figure (5). L'élément de base à partir duquel le circuit est décrit est le transistor (en forme de rectangle). Sa description physique varie selon la technologie utilisée pour réaliser le circuit.

La transformation d'une description structurelle en description physique est conçue comme le processus de partitionnement. Pour générer les masques, il y a deux méthodes : la méthode descendante de plan de masse, et la méthode ascendante d'assemblage de cellules. En général, le concepteur utilise un mélange

des deux méthodes.

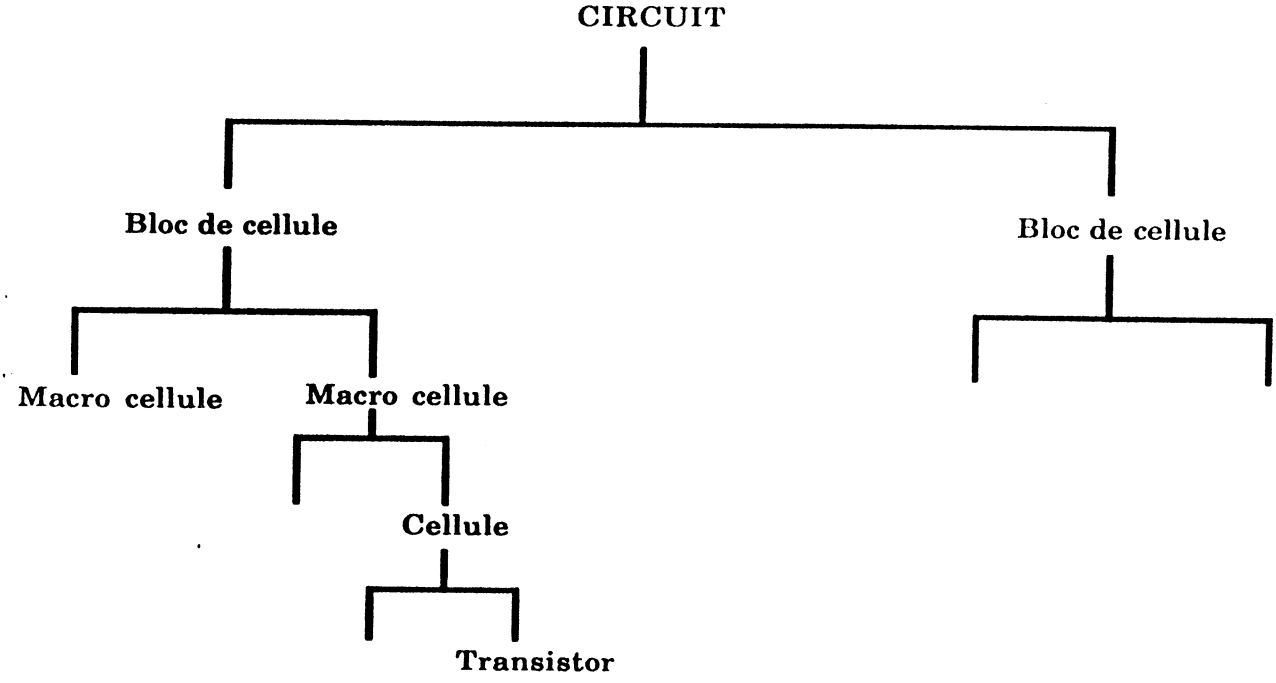


FIG. 5 : DECOMPOSITION PHYSIQUE

Pendant la phase de description physique, le concepteur peut avoir suffisamment d'informations détaillées qui lui permettent de prédire le comportement du système. Par exemple à partir d'une description des masques on peut extraire le graphe des connexions entre transistors. Cette liste peut être comparée au graphe généré à partir du niveau logique pour vérifier la cohérence de la description. En plus si nécessaire on peut générer les spécification électriques du circuit à partie des masques pour effectuer une simulation électrique.

2.2 Niveaux de description :

Il faut noter qu' il n'existe pas une définition standard des niveaux de description; par la suite en tenant compte de la présentation en [BAR 75], [BOR 81] et [RAM86], on va essayer de présenter ces niveaux en détail comme le montre la figure (6) (présentation détaillée de la figure (1)).

Niveau	Modèle	Eléments primitves	Valeurs à traiter	Modèle temporel
Système	Réseau de processeurs	processeur mémoire..	Données abstraites	Temps de commun.
Algorithme	Algorithmes paralléles	registre, bus, UAL,	Chaine de bits,entiers	Discréte
RTL	Fonctionnel	registre, bus,MUX,	Chaine de bits	Discréte
Logique	Equations booléens	Portes	Bits	Discréte
Switch	Equations algébrique	Switch, resistance	Noeuds (val,force)	Discréte
Eléctrique	Equs. Diffé rentielles	Transistor capacitor,	Valeurs réels	Continu

FIG. 6 : NIVEAUX DE DESCRIPTION

- Niveau Système :

C'est le niveau le plus haut de la description, ses modules ou éléments primitifs sont des processeurs, des mémoires, des interrupteurs et des unités périphériques (entrée/ sortie); ses paramètres sont le coût, la capacité de la mémoire, le débit et la quantité d'information manipulée, la puissance consommée .... D'un point de vue plus théorique, ces modules sont modélisés comme des types abstraits de données (abstract data type). Ces types sont des objets qui consistent en un porteur (carrier) structurel interne (registre, mémoire, fils de connexion) qui est invisible de l'extérieur et un ensemble d'opérations permises sur ce porteur. L'implantation de ces opérations est aussi invisible de l'extérieur.

- Niveau Algorithmique :

Pour chaque module au niveau système on décrit son algorithme d'interprétation pour l'ensemble des instructions. Cet algorithme inclue la structure de contrôle et les données à manipuler. Les composantes primitives sont le cycle d'interprétation, les instructions machine et les opérations à effectuer.

- Niveau transfert de registre :

Le niveau transfert de registre est obtenu du niveau algorithmique par une sorte d'inversion. Un algorithme est impératif par définition, le niveau transfert de registre est aussi impératif mais il est généralement non procédural. Au niveau transfert de registre le système est composé d'un ensemble (non ordonné) d'objets primitifs qui effectuent une action (normalement un transfert de registre) toutes les fois qu' une condition devient vraie. L'action à effectuer inclue une modification de l'espace global des conditions, par cette modification d'autres objets peut être activés. Les éléments primitifs à ce niveau sont les registres, les mémoires, les circuits combinatoires et les bus de données.

- Niveau Logique :

Pour que les modules au niveau transfert de registre puissent être implantés, on les développe en terme de fonctions ou circuits logiques. En principe, la description au niveau logique est purement structurelle, le comportement est caché et peut être construit à partir du comportement d'objets primitifs et de la structure d'interconnexion. Pour un niveau purement logique, les portes logiques sont les primitives de base. Habituellement, des fonctions complexes (bascules, additioneurs, multiplexeurs.....) peuvent exister dans une bibliothèque à ce niveau. La prise en compte du temps est faite de manière plus détaillée qu' aux niveaux précédents.

- Niveau Interrupteur (switch) :

Le point de vue de base à ce niveau est le même qu'au niveau logique. Le niveau d'interrupteurs est construit par le développement de portes logiques en terme de transistors interconnectés, où le transistor est présenté de manière simplifiée. A ce niveau les forces différentes du signal, et la transmission bidirectionnelle doivent être considérées. La différence principale entre ce niveau et le niveau logique est introduite par la porte de transmission bidirectionnelle alors que le niveau logique est unidirectionnel par nature.

- Niveau Electrique :

A ce niveau de description le comportement analogique détaillé du circuit est décrit. Habituellement c'est fait par un système d'équations différentielles. La majorité des propriétés discrètes des niveaux précédents est perdue, et le temps est traité au degré le plus fin, où le comportement du transitoire est une considération importante. Les objets primitifs à ce niveau sont des résistances, des capacités, des transistors et des sources de courant et de tension.

En principe, différents langages existent pour décrire un circuit aux différents niveaux de description. Dans la majorité des cas, ces langages sont consacrés à un seul niveau. Ainsi, un langage comme OCCAM [TAY82] peut être utilisé au niveau système, le langage ISPS [BAR 81] pour le niveau algorithmique, le langage CDL [CHU 72] au niveau transfert de registre, le langage TEGAS [THO 80] pour le niveau logique, le langage de MOSSIM [BRY 80] au niveau d'interrupteurs et le langage d'entrée SPICE [NAG 75] pour le niveau électrique.

Aujourd'hui, on peut trouver des langages qui décrivent le circuit aux différents niveaux de description "multi-level" dans le même environnement de description et de simulation. Des exemples de ces langages sont : MODLAN [PAW 81], DACAPO [BRU 85] et CASCADE [BOR 85].

Afin de terminer cette partie, il faut noter que notre objectif n'est pas de présenter en détail les différentes méthodes de description, mais d'attirer l'attention sur la nécessité et l'importance de l'utilisation d'outils logiciels automatisant les différentes étapes du processus de conception. Malgré la nécessité d'améliorer encore à la fois le résultat produit et les performances de tous ces outils, le problème essentiel est celui de leur intégration dans un système unique d'aide à la conception.

3. LES SYSTEMES INTEGRES D'AIDE A LA CONCEPTION

L'augmentation constante de la complexité des circuits intégrés et en plus l'importance qu' ils prennent dans de nombreux domaines, rendent indispensable la création d'outils de Conception Assistée par Ordinateur (CAO) répondant à cette complexité et en particulier permettant de suivre le processus global d'élaboration des ensembles électroniques.

Il est intéressant ici de distinguer les deux termes : conception automatisée et conception assistée. La conception automatisée dans un sens est plus ambitieuse. A partir d'une spécification abstraite du système à réaliser, la conception automatisée le génère de manière automatique. Un avantage de cette approche est que le système complet n' a pas besoin d'être vérifié. Un vrai système de conception automatisée ne génère que des objets corrects , une vérification du comportement au niveau le plus haut (algorithmique ou fonctionnel) peut être effectuée dans la phase initiale.

Dans l'état de l'art actuel, un algorithme complet pour la synthèse n'existe que pour de petites parties de la conception. D'autre part, les systèmes de conception assistée sont des collections d'outils qui impliquent des participations plus actives du concepteur. Quelques outils peuvent être des générateurs automatiques de parties spécifiques de l'objet à réaliser. D'autres outils sont utilisés par le concepteur de manière explicite pour effectuer la tâche de la conception.

Faut-il remplacer la conception humaine (où la plupart des décisions sont prises par le concepteur pendant le processus de la conception) par une compilation automatique ? Faut-il garder la conception humaine assistée par un ensemble d'outils? Il n'y a pas contradiction entre les deux, la première solution est simplement plus futuriste.

La majorité des systèmes de conception existant sont en réalité jusqu' à maintenant des systèmes de conception assistée; c' est le concepteur qui prend les décisions importantes pendant le processus de conception. Comme on l'a mentionné dans la première partie, le futur système de conception pour les circuits VLSI, va être caractérisé par la combinaison efficace du processus algorithmique traditionnel et des techniques d'IA et des bases de connaissance.

Un système de conception assistée doit s'appuyer sur un ensemble d'outils couvrant les différentes phases de la conception. Il doit également gérer des contraintes à propos de l'enchaînement de ces phases (limitant ainsi le risque d'erreurs humaines), et posséder des mécanismes d'interaction homme-machine évolués pour manipuler aisément les grandes quantités d'informations mises en jeu.

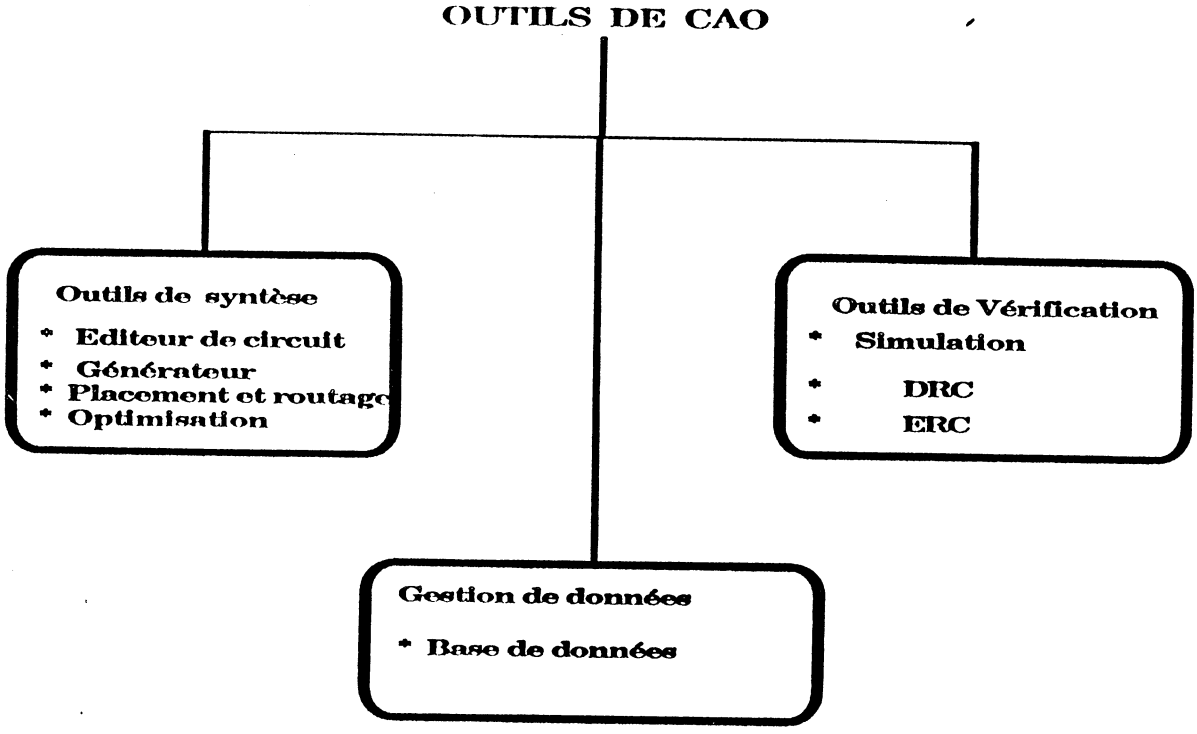


FIG . 7 : OUTILS DE SYSTEME DE CAO

On peut classer en trois catégories, comme le montre la figure (7) les différents types d'outils nécessaires pour assurer une conception correcte de circuits intégrés :

- Les outils de synthèse qui comprennent des outils de génération de description, de partitionnement, de génération de structures, d'optimisation , de placement, de routage et d'interconnexion entre structures d'implantation.

- Les outils d'analyse qui incluent des outils de simulation normale, de simulation de fautes, de génération de vecteurs du test, de vérification de règles de dessin (DRC) et de règles électriques (ERC), de comparaison de graphes (électrique-logique) et d'extraction de paramètres électriques.

- Les outils de gestion des données relatives au circuit. L'accès à ces dernières, leur modification, ajout ou suppression doivent être gérés et contrôlés de manière à assurer leur homogénéité et cohérence.

Le nombre de systèmes d'aide à la conception de circuits VLSI a augmenté de manière considérable dans les dernières années. La plupart de ces systèmes suivent une méthodologie de conception descendante hiérarchique. La notion de hiérarchie a

donné lieu à des représentations arborescentes de structures de données qui semblent être adéquates pour ce type de problèmes. Cette approche hiérarchique permet au concepteur d'aborder de manière progressive et modulaire la description des circuits et d'appréhender des complexités de plus en plus importantes grâce à une segmentation fonctionnelle.

Par rapport à l'utilisation de nombreux programmes séparés et indépendants, un système intégré de CAO présente les avantages suivants:

- Réduire le travail de description du circuit en évitant les doubles définitions (notion de conception simple).

- Eviter au maximum les vérifications à posteriori, toujours très lourdes, en remplaçant par des générations automatiques ou par des vérifications effectuées au moment de l'entrée conversationnelle des informations (notion de conception rapide).

- Permettre la réalisation de programmes travaillant sur plusieurs représentations habituellement distinctes du circuit. Cet avantage sera en particulier utilisé pour la mise en place d'outils permettant d'assurer la cohérence des informations entre les niveaux de description (notion de conception sûre).

La figure (8), présente la structure globale du système "CASSIOPEE" d'aide à la conception en cours de développement au CNET Grenoble. Dans la suite on va présenter en résumé les caractéristiques principales et les composantes de ce système.

Les logiciels de CASSIOPEE couvrent trois étapes successives, qui sont :

1) L'étude de l'architecture, c'est à dire le découpage en blocs fonctionnels tels que des mémoires, des unités de calcul, des portes d'entrée/sortie et le séquencement correspondant de l'ensemble.

2) L'étude du schéma logique à base de portes (ou du schéma analogique à base d'amplificateurs).

3) L'implantation physique de tous les transistors sur le silicium.

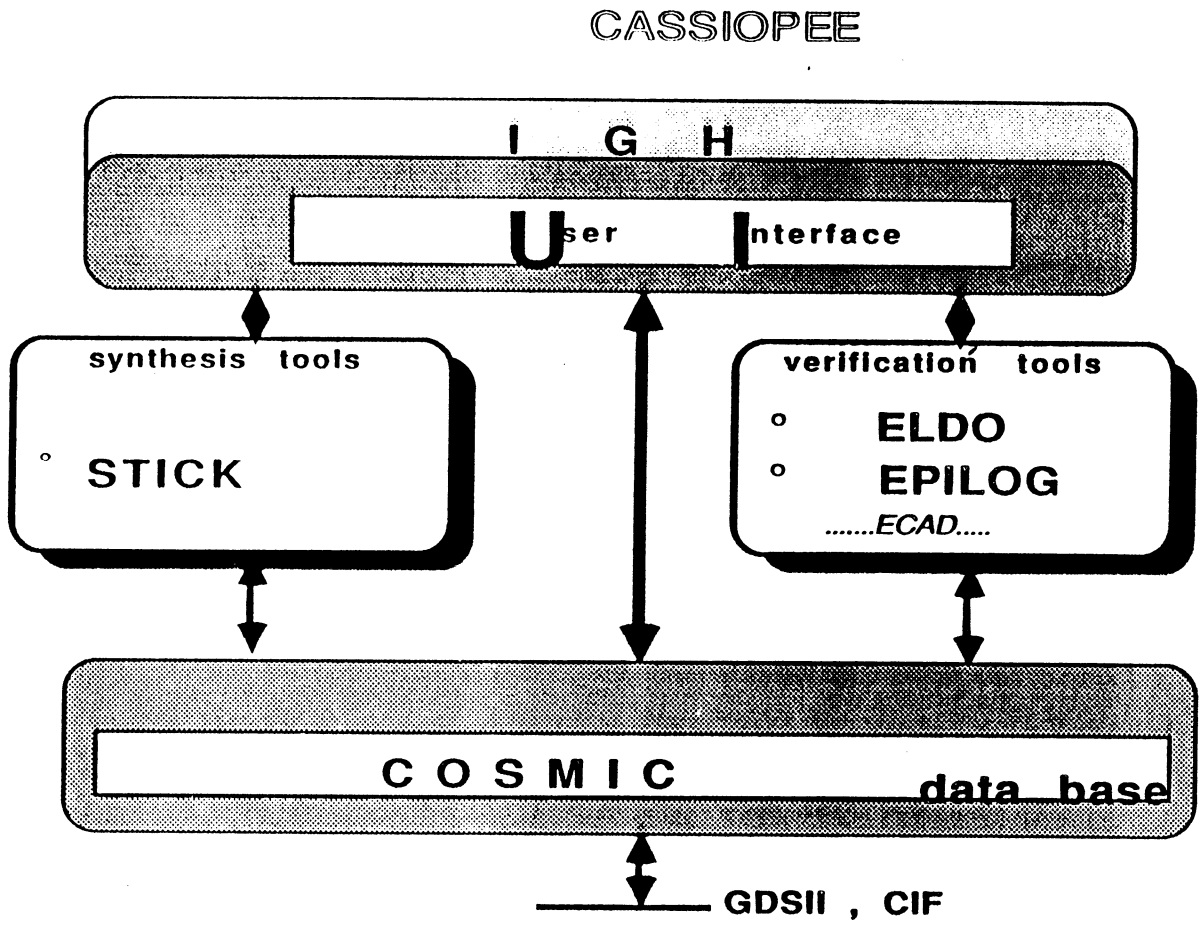


FIG. 8 : STRUCTURE GENERALE DE CASSIOPEE

L'infrastructure du système CASSIOPEE repose sur la base de données COSMIC [JUL 86], l'éditeur graphique généralisé IGH/UI [POI 86], et les outils de CAO .

3.1 La base de données COSMIC:

Ce gestionnaire de données fournit :

* La gestion d'un schéma unique de données qui supporte l'ensemble du processus de conception d'un circuit intégré en permettant :

  - la décomposition hiérarchique,
  - la manipulation de versions,
  - les descriptions graphiques,

- les descriptions procédurales.

* Un contrôle de cohérence sur les informations enregistrées (intégrité référentielle : par exemple vérification qu' une tension appartient à un intervalle valide).

* La gestion de vues << privées >> qui permet de ne faire voir qu'une partie du schéma à un outil ou au contraire d'étendre le schéma pour supporter de nouveaux outils ou intégrer de nouvelles méthodologies.

* La gestion physique des données et les méthodes d'accès à ces dernières.

* La cohérence et le partage des données (notion de bibliothèques).

Les informations enregistrées dans COSMIC sont manipulées à travers un interface standard appelé << FACILE >> [HEL 86]. COSMIC représente également le point de départ du projet européen CVT (Conception assistée de circuits VLSI pour les Télécommunications).

3.2 L'interface utilisateur (UI) et l'interface graphique (IGH) :

L'ensemble UI/IGH représente une boîte à outils de logiciels qui permet :

* D'offrir aux concepteurs de circuits intégrés un interface convenable et adaptable sur les différents outils. Cet objectif est réalisé par :

1) la définition de l'interface à l'aide d'un fichier interprété au lancement de l'application et modifiable par l'utilisateur.

2) une grande variété de formes de dialogues : menus fixés, clés de fonction (clavier), formulaires.

3) l'élimination des dialogues questions/réponses pour l'acquisition des données (utilisation de contextes).

4) le multi-fenêtrage.

* De décharger au maximum le programmeur d'application des tâches de gestion des dialogues et de l'écran.

3.3 Les outils de CASSIOPEE :

Dans l'environnement COSMIC-IGH/UI de CASSIOPEE, un ensemble d'outils est en cours de réalisation ou d'intégration :

* Un éditeur généralisé qui manipule les objets de COSMIC. Cet éditeur permet la manipulation des cellules contenues dans COSMIC au niveau de leurs représentations logiques (synthèse architecturale) ou implantées (génération de modules).

* Un générateur paramètrable de liste de connexion qui permet d'établir le lien entre les objets contenus dans COSMIC et les principaux simulateurs <<standard>> du marché.

* Une chaîne de conception de filtres à capacités commutées qui, à partir du gabarit d'un filtre, génère automatiquement les masques [ASS 87].

* Une chaîne de conception de cellules élémentaires basée sur une représentation symbolique du masque. Cette chaîne qui inclut un outil d'assemblage de cellules, LOF [BER 84], génère automatiquement des modules flexibles (ROM, RAM, PLA....).

Ces derniers outils sont communément applelés <assembleurs de silicium>.

Dans cette partie on a présenté les principes des systèmes de CAO, et on a montré que parmi les principaux outils intégrés dans ce système on peut trouver : des outils de description, des outils de synthèse et des outils de vérification. Ces trois catégories d'outils utilisent des langages de description de matériel. Dans la partie suivante on va discuter en détail l'aspect "**langage de description de matériel**".

4. LES LANGAGES DE DESCRIPTION DE MATERIEL

En 1974, un éditeur en chef pour une édition spéciale de "COMPUTER", Y. Chu [CHU 74] a posé la question "pourquoi a-t-on besoin de langages de description de matériel (Hardware Description Languages-HDL) ?". Des experts ont répondus à la question, en discutant les similitudes et les différences entre la programmation et la conception de matériel (circuits). Ils ont affirmé la nécessité d'avoir un langage de haut niveau par lequel le concepteur peut exprimer les opérations asynchrones, le contrôle parallèle, et la structure d'un système.

En 1977, G.Jack.Lipovski [LIP 77] a averti que "on a l'impression d'être isolé dans notre tour de Babel, chaque personne parle un langage différent mais personne ne communique". Stephen Su, l'éditeur en chef pour cette deuxième édition spéciale de"COMPUTER", a proposé des directions possibles pour des recherches dans l'avenir. En particulier, l'utilisation d'un langage commun entre les concepteurs et un langage de conception qui reflète la philosophie du modèle commun pour le logiciel et le matériel.

Enfin, en 1985, Barbacci [BAR 85] a mentionné que les langages de description de matériel aident à établir les liens entre le logiciel et le matériel. Ce lien est important puisque les problèmes rencontrés par les concepteurs de logiciel et de matériel sont de nature similaire.

A partir de ces remarques importantes on peut dire que le but est **de trouver un langage qui est descriptif aussi bien qu'efficace à utiliser pendant le processus de conception.**

L'utilisation de langages pour décrire le comportement d'un système, ses relations avec son environnement, et la manière dont il est réalisé est le support de base des différentes étapes de la conception de circuits VLSI.

L'importance de ces langages de description est liée au fait que, pour comprendre la structure d'un objet ou la fonction qu' il réalise, ce dernier doit être décrit selon un ensemble de règles. Le respect de ces règles (syntaxiques et sémantiques) est vérifié à chaque phase de description.

Tout système peut être représenté par la fonction qu' il réalise ; les algorithmes décrivant son comportement sont traduits successivement d'une représentation comportementale de haut niveau jusqu' à une représentation physique (réalisée sur silicium). Ces étapes de représentation sont importantes aussi bien que le processus de leur traduction.

4.1 Domaines d'applications :

Citons les domaines différents d'applications du langage de description:

- Documentation :
  Une application importante des langages de description de matériel est la description de comportement et/ou de structure de système à propos d'une communication précise entre les concepteurs et les utilisateurs. Une application particulière de ce langage comme un outil descriptif est la description de la structure de l'ordinateur.

- Simulation :
  La simulation est l'outil le plus utilisé pour la validation partielle ou totale d'un circuit à réaliser à chaque niveau d'abstraction pendant le processus de la conception. L'efficacité d'un tel simulateur dépend de la puissance et de la précision du langage de description qui lui est associé.

- Génération automatique :
  Maintenant, une application importante du langage de description de matériel est d'être utilisé en entrée de compilateur de silicium qui génère les masques à partir de la description ou spécification au niveau le plus haut (compilateur du type comportemental).

- Vérification formelle :
  Bien que des efforts soient fait pour vérifier la correction de programmes et de circuits, cette approche a abouti jusqu' à présent à peu de résultats à cause de la complexité de procédures de vérification.

- Application au test :
  Maintenant, les langages de description sont utilisés dans le système du test pour aider la génération automatique de séquence du test (Automatic Test

Sequence Generation - ATSG). Les meilleurs exemples de ces systèmes sont : TEGAS [SZY 72], et HILO [FLA 81].

Toutes les applications doivent être activées à l'aide d'un langage de commande.

4.2 Propriétés des langages :

Dans un système logique beaucoup d'activités peuvent arriver concurremment. Il est alors important d'avoir une méthode naturelle pour exprimer le parallélisme. D'autres caractéristiques du système logique sont la nature non-récursive de ses opérations, et la présence de contraintes de temps qui doivent être prises en compte.

Comme la construction de systèmes logiques est faite à partir d'unités fonctionnelles décomposées de manière logique, on a besoin de moyens pour décrire l'interaction entre ces unités, à la fois comme "subroutine" (relation hiérarchique), et comme "coroutines" (relation symétrique).

Le comportement du système logique est décrit par des actions ou des activités en séquence, on a besoin de mécanismes pour garantir la coopération harmonieuse (synchronisation) entre les activités.

Van Cleemput [VAN 79] a noté qu'il est difficile de décrire le processus de la conception de matériel formellement, puisque cela dépend de l'opinion de chaque concepteur, et des problèmes spécifiques de la conception qui doivent être résolus de manière spéciale. A partir d'un ensemble de spécifications qui sont quelquefois vagues et incomplètes, le concepteur applique une série de transformations successives (améliorations itératives) jusqu' à ce que le système puisse être réalisé (construit) dans un environnement technologique donné, ou jusqu' à ce que l'on trouve que les spécifications ne sont pas réalisables.

Bien que des méthodes formelles existent pour résoudre quelques problèmes, tels que la minimisation de circuits combinatoires ou l'assignation des états pour les circuits séquentiels, les concepteurs fondent leur conception sur une "bibliothèque" d'exemples. Ces exemples peuvent être créés depuis des expériences précédentes, de la littérature ou des exposés académiques.

Pour présenter le système logique de manière précise, les langages de description de matériel (HDL) doivent être capables de décrire le parallèlisme, la nature non-récursive et les contraintes de temps. Ils sont alors différents de la nature purement séquentielle de langages de programmation (HHL).

Pour que le langage de description de matériel soit un outil utilisable, il doit inclure différentes propriétés ([FIG 73],[BAR 75] et [SHI 79]). Nous allons en discuter quelques unes en détail.

4.2.1: Domaine du langage :

Le langage doit permettre la description du système logique aux différents niveaux (système, porte logique, et transistor). Le langage doit permettre la création de modèles de simulation correspondant à chaque niveau de la conception. Par exemple, au niveau système, des modèles algorithmiques sont demandés. Aussi, le langage doit permettre au concepteur de spécifier la valeur du retard pour les circuits combinatoires ; sans ces caractéristiques, le langage ne peut pas présenter le matériel de manière satisfaisante.

Le langage ne doit pas être limité à un niveau donné de description, à un style de conception, à une technologie donnée ou à une application spéciale. Ceci nécessite l'utilisation du langage dans les différentes applications (cf. 4.1) et qu' il doit être extensible et général (les notions du langage doivent décrire les éléments à chaque niveau de description et pour une majorité d'applications).

4.2.2: Description hiérarchique :

La hiérarchie de la description est une propriété importante pour un langage de description de matériel (HDL). La description hiérarchique est présentée comme la base pour la méthode descendante de la conception ; en utilisant le raffinement étape par étape on fournit des informations détaillées jusqu' au niveau le plus bas. L'abstraction des données de conception facilite aussi la conception hiérarchique.

Le langage doit supporter la description modulaire, et doit aussi être structuré afin que la recompilation globale d'un circuit à chaque changement interne dans un sous circuit ne soit pas nécessaire. La partition du circuit et le regroupement de sous circuits sont nécessaires pour créer les modèles pour la simulation ou pour les

autres applications dans le processus de la conception.

Avec la modélisation, les concepteurs peuvent réutiliser les sous circuits ou blocs du circuit qui sont déjà crées, la création d'une bibliothèque sera facile et assurée à chaque niveau de la hiérarchie.

La description hiérarchique est basée sur la hiérarchie structurelle et l'abstraction comportementale.

La hiérarchie structurelle signifie la décomposition du circuit en fonction des primitives du langage et de leurs interconnexions. Par exemple, on considère le cas où les fonctions logiques (AND, OR, NAND et NOR) sont les seules primitives matérielles. Par conséquent, les composantes structurelles au niveau haut comme les additionneurs, les multiplicateurs et les multiplexeurs peuvent avoir leurs fonctions décrites seulement en fonction de primitives de niveau le plus bas.

Plusieurs langages de description structurelle ont été proposés pour décrire la hiérarchie structurelle, on peut citer parmi eux : SDL [VAN77], HISDL [WIL 82], et BDL [SLU 84].

L'abstraction comportementale permet l'existence de la description de comportement à n' importe quel niveau et peut fournir une structure implicite pour les connexions entre les primitives au niveau le plus bas. L'abstraction comportementale permet que les fonctions d'additionneur, de multiplicateur et de multiplexeur puissent être décrites de manière algorithmique plutôt que seulement en terme de fonctions primitives. Parmi les langages qui peuvent être utilisés pour décrire l'abstraction comportementale, on peut citer ISPS [BAR 81], DACAPO [BRU85], CADOC [BEL 85], et ELLA [MOR 85].

Le support fourni par le langage pour la hiérarchie structurelle et l'abstraction comportementale permet que les descriptions structurelle et comportementale soient spécifiées à n'importe quel niveau de la hiérarchie. Dans ce cas, les modules du haut niveau peuvent être construits sans la restriction d'utiliser les primitives définies par le langage, et par conséquent on peut trouver des améliorations significatives dans la capacité de la simulation du circuit (simulation multi-niveau). Il y a plusieurs langages qui présentent cette hiérarchie, parmi eux :

MODLAN [PAW81], CASSANDRE [MER 73], HILO [FLA81], TEXSIM [CAL 85] et HELIX [SIL 83].

En plus la description hiérarchique est capable de réduire les ressources consacrées à la conception (réduction de taille mémoire pour stocker les informations et du temps de vérification).

4.2.3 : La prise en compte du temps :

La notion de temps qui est inclue dans le langage de description de matériel doit être orientée vers l'application de la simulation et la vérification temporelle. Pour assurer que le concepteur peut effectuer la description de tous les systèmes logiques, le langage doit supporter la notion de temps à tous les niveaux de la hiérarchie de la conception.

Pour satisfaire les différentes catégories de la conception, un modèle général du temps doit être fourni pour supporter la conception. Le modèle du temps doit permettre au concepteur de définir l'intervalle de temps en unités de temps.

Le langage doit permettre les différentes spécifications temporelles [BOR 81] : modèle asynchrone, modèle synchrone et modèle du retard.

Au niveau système, le concepteur peut définir un processus et ses entrées ; chaque fois qu'une entrée change sa valeur, le processus va être exécuté de manière asynchrone. Le temps à ce niveau apparait sous la forme de durées d'exécutions de processus.

Au niveau transfert de registre, un ensemble d'actions peut être exécuté de manière synchrone en utilisant une condition de contrôle. Quand cette condition est satisfaite, toute les actions s'exécutent en parallèle. La synchronisation est assurée par un ou plusieurs signaux de contrôle périodiques, appelés horloges, ou par des conditions gardées temporisées (qui déclenchent les actions).

Au niveau le plus fins (niveau logique, switch) le langage doit permettre les spécifications de différents modèles de retard de propagation : le retard de transport et le retard inertiel.

Le retard de propagation modélise une ligne de transmission (purement résistive), c'est à dire que chaque changement à l'entrée va apparaitre à la sortie après une certaine période du temps. Le retard de propagation peut être plus long que l'impulsion d'entrée.

Le premier type de retard de propagation est le retard de transport, et qui modélise l'effet que chaque changement à l'entrée (court ou long) va générer un changement retardé à la sortie. La figure (9), implique que la sortie d'une fonction logique va réagir au bout d'un temps constant aux signaux d'entrées.

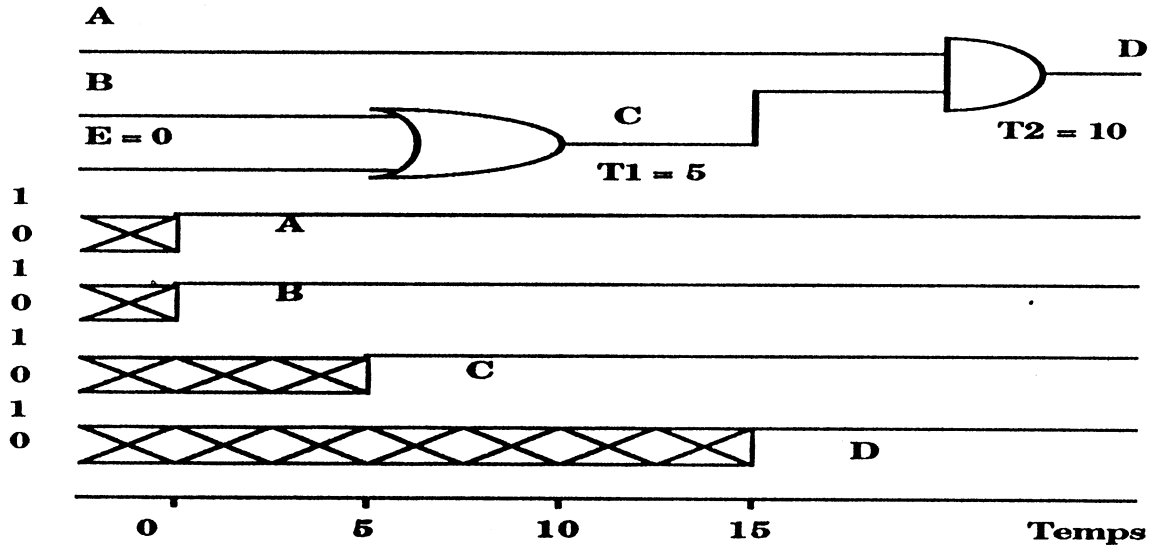


FIG . 9 : RETARD DE TRANSPORT

Dans certain cas, l'entrée peut changer sa valeur de manière parasite ; pour éviter l'apparition de ce changement à la sortie, on utilise le retard inertiel. C'est à dire, si l'entrée change sa valeur dans un intervalle plus court que la valeur du retard inertiel, ce changement n'est pas pris en compte (filtré) à la sortie.

Le retard inertiel donc détermine la période minimale de stabilité pour propager le changement de l'entrée à la sortie. La figure (10), représente le retard de transport avec (et sans) le retard inertiel.

En plus le langage doit permettre la description de caractéristiques et contraintes temporelles dans le modèle de la conception. Les caractéristiques temporelles incluent : le temps montant (rise time), le temps descendant (fall time) et le délai de propagation. Les contraintes temporelles incluent : le temps

d'établissement (setup time), le temps de maintien (hold time), et la période d'impulsion (pulse width).

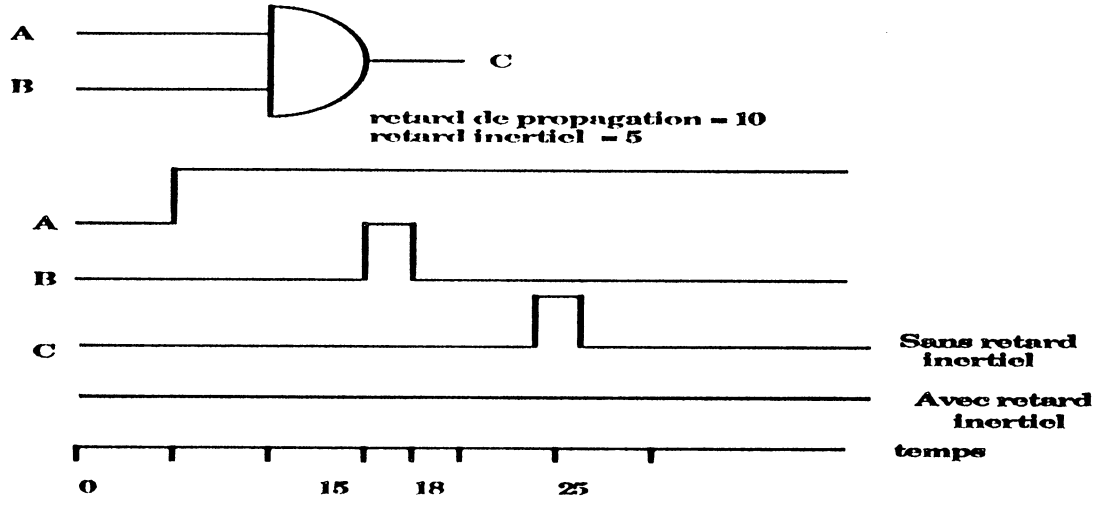


FIG. 10 : RETARD INERTIEL

4.2.4. Parallélisme et séquencement :

Il est important que le langage de description de matériel ait une manière simple et naturelle d'exprimer le parallélisme. En général tous les langages de description de matériel offrent des primitives pour exprimer le parallélisme.

Pour étudier la notion de parallélisme dans le langage de description , il faut d'abord distinguer, selon les définitions qui sont introduites par Barbacci [BAR 75] et mentionnées par Borrione [BOR 81], entre les langages procéduraux et non-procéduraux.

Un langage est dit procédural si l'ordre d'écriture des instructions influe sur l'ordre dans lequel elles sont exécutées. Un tel langage fournit des primitives pour décrire l'enchaînement des instructions, ou modifier l'enchaînement pris par défaut. En ce qui concerne la description du parallélisme dans ce langage il y a deux approches qui sont particulièrement typiques :

- Le parallélisme et la séquentialité des instructions sont spécifiés par des séparateurs différents et un parenthésage adéquat (comme ISPS [BAR 81]).

- Une action est décrite par une liste d'instructions, élémentaires ou composées, s'exécutant séquentiellement selon l'ordre dans lequel elles ont été écrites, le

parallélisme et la séquentialité entre les actions étant exprimés par un graphe de contrôle distinct (exemple : DACAPO [BRU 85], CADOC [BEL 85], LASSO [BOR 79]).

Un langage est dit non-procédural si l'ordre d'exécution des instructions est indépendant de leur ordre d'écriture. Dans un langage de description non-procédural, toutes les instructions, élémentaire ou composées, décrivent des actions parallèles. Ainsi, l'ordre d'écriture des instructions n' a aucune signification autre que de convenance ou de lisibilité. Des groupes d'instructions peuvent être préfixés par une condition sous forme d'expression de contrôle (comme CDL [CHU74]), ou de nom d'état (CASSANDRE [MER 73], DDL [DIE 74]). Les validations et invalidations successives de ces préfixes assurent le séquencement des actions qui en dépendent.

Borrione a ajouté que les langages de description procéduraux sont plus proches des langages de programmation. Toutefois, une différence essentielle existe entre une majorité de langages de description procéduraux et les langages de programmation comportant des primitives d'expression du parallélisme (on parle plus souvent dans ce deuxième cas de processus concurrents) malgré leurs similitudes syntaxiques. Cette différence tient précisément à la sémantique que l'on exprime.

La différence entre les langages procéduraux et non-procéduraux tient aux notions sémantiques de parallélisme et de séquentialité offertes par le langage, comme le montre la figure (11).

L'interprétation non procédurale implique que le modèle peut être activé dans deux cas : changement d'entrées et changement de variables internes ; tandis que l'interprétation procédurale implique qu' on active le modèle seulement dans le cas de changement d'entrées.

Dans cet exemple l'interprétation non procédurale est : on active le modèle au temps t (changement de la variable A) et on exécute les instructions en tenant compte de retards comme des événements futurs (on les stocke dans l'échéancier interne et on demande le réveil du modèle aux valeurs du temps correspondant). Chaque fois on réactive le modèle et on exécute toutes les instructions avec les nouvelles valeurs des variables internes (si elles existent). Donc, la variable C change au temps t+3

comme résultat d'évaluation au temps t et au temps t+5 à cause du changement de B au temps t+2.

Avec l'interprétation procédurale de cet exemple,les transferts B = A , C =B auront lieu au temps t+2 et t+5 respectivement. Les transferts conditionnels Z=0, Y=1 auront lieu au temps t+7 et t+6 respectivement.

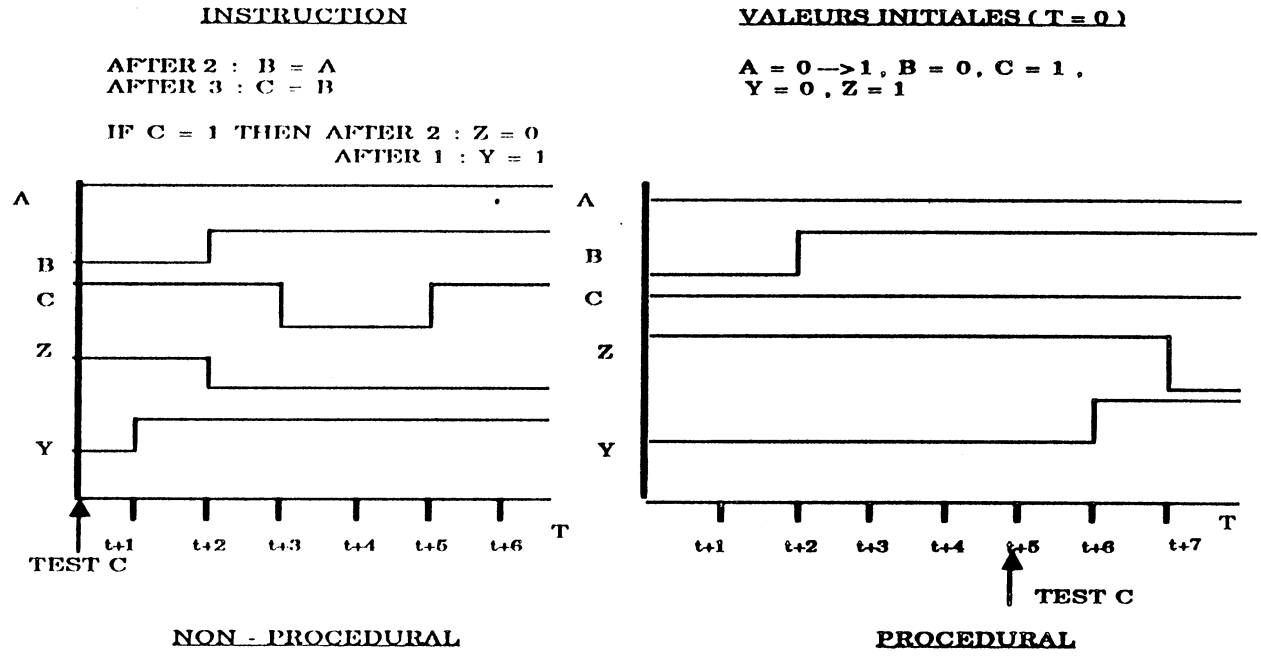


FIG . 11

Il faut remarquer que la variable C reste à la valeur 1 comme le transfert C=B est exécuté après le transfert B=A.

Les langages procéduraux sont mieux situés pour les applications dans le domaine de synthèse de partie opérative (comme ISPS [BAR 81]) et dans la vérification au niveau haut, tandis que les langages non-procéduraux sont utilisables aux niveaux détaillés pour les applications de vérification (simulation, test ...) et pour la synthèse de partie de contrôle (comme IRENE [MAR 86]).

4.2.5. Description d'architecture :

Dans la description de système logique, deux types essentiels et différents d'information peuvent être captés : le comportement projeté du système et les détails d'implantations actuelles.

Le premier demande un langage de description comportementale, tandis que le deuxième peut être exprimé par un langage de description structurelle.

La description comportementale doit supporter la description procédurale ou algorithmique du circuit ou système. L'ordre linéaire des instructions dans la description du comportement définit la séquence normale (par définition) de fonctionnement . Les structures de contrôle et de branchement doivent être fournies pour permettre aux concepteurs de contrôler cette séquence. Par définition, il n' y a pas de structure implicite dans une description purement comportementale.

Du point vue de l'architecture, les concepteurs décrivent leur système de matériel en utilisant des langages non-procéduraux. La structure peut être implicite par la combinaison d'instructions et de conditions logiques (instructions gardées) pour contrôler l'exécution de chaque instruction, et aussi par le transfert de données entre les différents porteurs (registres). La sémantique doit permettre les expressions du parallélisme, et du contrôle dans la description de l'architecture.

Avec cette forme de description le concepteur peut séparer la partie contrôle de la partie opérative. La logique de la partie contrôle est représentée par les gardes contrôlant l'exécution de chaque instruction. Les instructions à exécuter représentent la partie opérative.

Le langage doit permettre les descriptions structurelles de façon explicite. Ces descriptions ne doivent pas être limitées à un seul niveau d'abstraction. Une description de structure explicite doit fournir seulement les informations de connexion des composants inclus. Il ne doit pas exister d'information sur le comportement des composants dans une description de structure explicite. Pour que la simulation procède, la description du comportement des composants doit exister au niveau bas de la hiérarchie de description.

La séparation entres les deux descriptions (de structure et de comportement) aide à vérifier la cohérence de la description (en utilisant la simulation de différentes descriptions).

Enfin, le langage doit permettre la description de composants génériques. Un composant générique est un composant avec un ensemble de paramètres fourni par le concepteur. Ces paramètres vont permettre au concepteur pendant la phase de

compilation de générer les composants désirés. Le langage doit permettre la description de structures régulières.

La séparation entre description de structure et du comportement a été systématisée dans certains systèmes d'aide à la conception. Par exemple, le système SABLE [HIL 79], traite des descriptions exprimées à l'aide des deux langages distincts : ADLIB [HIL 79] pour le comportement de chaque type de module, SDL [VAN 77] pour l'interconnexion des modules.

De manière idéale, les informations comportementales et structurelles doivent être exprimées en utilisant un seul langage. En pratique, ce langage unique est possible parce qu'un appel simple d'une fonction dans une description comportementale, et dans lequel tous les arguments sont des signaux, correspond à une description structurelle aussi bien qu' à une description fonctionnelle.

Notre approche sera d'utiliser un seul langage pour exprimer les deux descriptions de comportement et de structure. Ceci facilite la tâche de description en laissant au concepteur toute la liberté de décider la finesse de décomposition de son circuit et de définir les modules qu'il considère comme primitifs pour son application.

5. LES APPROCHES DES LANGAGES DE DESCRIPTION

Pendant le processus de conception on doit procéder à des descriptions différentes aux niveaux différents d'abstraction (au moins si le processus de conception est fait de manière systématique en conception descendante). Pour cela, trois approches [RAM 86], peuvent être suivies :

- L'approche des langages dédiés :

  Cette approche est la solution classique. Elle a l'avantage que des langages relativement simples peuvent être utilisés, faciles à étudier, faciles à comprendre, et efficacement exécutés. Par contre, cette approche est complètement chaotique avec une grande perte aux interfaces entre les langages différents. En pratique, cette approche est seulement utilisable pendant le processus de conception pour des applications limitées.

- L'approche de famille de langages :

Une façon possible de résoudre les problèmes d'approche des langages dédiés, est de construire une famille de langages (langages génériques). Chaque langage est dédié à certain niveau d'abstraction, mais les membres de la famille sont liés en ce qui concerne les syntaxes et les sémantiques. Par cette solution, il n' y a pas beaucoup de pertes à attendre aux interfaces, mais elles sont pourtant présentes. L'approche CONLAN [BOR 81], [PIL 85] est un exemple typique de cette approche.

- L'approche du langage commun :

Un langage commun est tout à fait à l'opposé d'un ensemble de langages dédiés. Comme le concepteur a la possibilité de travailler avec un seul langage, il est très facile de décrire un circuit aux différents niveaux d'abstraction et le transformer d'un niveau à l'autre. Pour la même raison, il est plus facile de réaliser une simulation multi - niveau. D'autre part, il existe bien sûr le danger qu' un tel langage soit trop complexe. Pour résoudre ce problème les concepts de modélisation doivent être examinés en détail et réduits à un ensemble minimal de concepts nécessaires. Les exemples de cette approche sont DACAPO [BRU 85] que l'on va présenter en résumé dans le chapitre V, et VHDL [SHA 85], [ROG86] que l'on présente à la suite de cette partie.

Nous allons présenter les deux approches CONLAN et VHDL de manière générale (objectifs, caractéristiques générales) et nous allons aussi essayer d'analyser les deux approches selon les différentes propriétés de langages de description présentées dans la partie précédente (cf. 4.2).

5.1 L'approche CONLAN :

CONLAN (CONsensus LANguage) est un mécanisme de construction formelle pour définir des langages de description de système logique et décrire des machines digitales à différents niveaux d'abstraction.

Le projet CONLAN a commencé en 1973 dans le but de réunir les langages de description de matériel existant dans un langage standard.

En 1975 une nouvelle approche a commencé par la formation d'un groupe de travail international. Les objectifs les plus importants dans cette nouvelle approche sont : la syntaxe et la sémantique de CONLAN qui doivent être raisonnablement uniformes pour tous les niveaux de description ; elles doivent reposer sur une définition formelle.

Pour satisfaire tous les objectives de CONLAN [PIL 83] une famille de langages est défini. Cette famille de langages repose sur un coeur syntaxique commun et une définition sémantique commune.

Chaque élément de la famille sera particularisé pour un niveau de description et sera construit à l'aide d'un procédé uniforme d'extension à partir d'un ensemble commun de notions primitives.

Un langage élément de la famille CONLAN est défini par dérivation à partir d'un langage existant, son langage de référence (REFLAN). Tous les langages sont définis à partir d'un langage de base : BCL (Base CONLAN), qui est lui même défini (pour montrer le mécanisme de construction du langage) à partir d'un langage plus élémentaire : PSCL (Primitive Set CONLAN), comme le montre la figure (12). Tous les futurs langages de CONLAN vont être dérivés à partir de BCL.

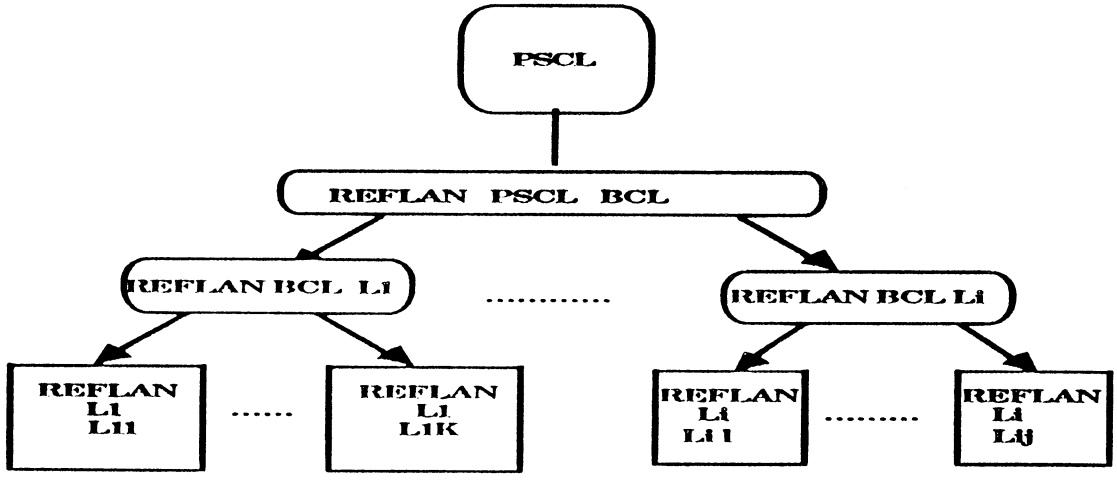


FIG. 12 : DEFINITION UN LANGAGE EN CONLAN

Le langage BCL est défini par un ensemble d'objets primitifs qui sont : des valeurs (entiers, booléens, caractères, et chaîne de caractères), et des porteurs (carriers) qui représentent des éléments matériels d'un système logique (registres, mémoires, fils de connexion).

Toutes les invocations d'opérations en BCL sont exécutées en parallèle. Les invocations conditionnelles d'opérations (IF, CASE) définissent les conditions de contrôle sous lesquelles une opération va être exécutée.

Le concept de modularité dans CONLAN est basé sur la notion de segment. Un module (un circuit ou partie d'un circuit) peut être décrit en fonction de sous modules (description structurelle) ou en fonction d'invocations d'opérations (description comportementale).

Le temps est discrétisé en BCL par des instants, chaque instant est lui même discrétisé par des étapes de calcul, le fonction de retard (delay) peut être appliqué au signal, comme le montre l'exemple suivant :

Exemple

```
FUNCTION delay ( x : signal(v) ; d : pint ) : signal(v) ;
```

Le signal(v) retourné par cette fonction est décalé de "d" unités de temps par rapport au signal(x).

BCL permet les formes générales de description de comportement de matériel et fournit les deux modes suivants : flux de signal et transition d'état conditionné.

Les deux exemples suivants expriment les modes précédents, le premier exemple est une description structurelle d'un bascule "RS" en termes de portes logiques NOR (avec retard unitaire). Le deuxième exemple sera une description fonctionnelle d'une bascule "D" à partir de son graphe d'états.

Exemple 1 flux de signal

```
DESCRIPTION rsff ( IN r,s : signal(bool); OUT q , nq : btm0 ) ;
   BODY
            nq .= nor ( r % 1 , q % 1 ) ;
             q .= nor ( s % 1 , nq % 1 )
END rsff
```

Le type btm0 est un sous_type de terminal avec des valeurs de type booléen et une

valeur par défaut 0 ; le signe % est la syntaxe simplifiée de l'appel de fonction DELAY .

Exemple 2 transition d'états condionné

```
DESCRIPTION dff ( tsu, th , tp  : int )
                        ( IN d , ck : signal ( bool ) ;
                          OUT q , nq : variable ( bool , 0 ) ) ;
ASSERT tp > th  ENDASSERT
BODY
  ASSERT
     IF (ck%th) &  not ( ck% (th+1)) THEN
           stable(d, tsu+th) & stable0(ck%th, tsu) & stable1(ck, tu)
     ELSE 1

     ENDIF
  ENDASSERT
  IF  (ck%(tp-1)) & not(ck%tp)  THEN  q := d , nq := not d ;
  ENDIF
END dff
```

Remarques :

- Cette description est paramètrée par "tsu, th, tp" (trois paramètres temporels). Ces paramètres doivent être fixés lors d'une instantiation.

- Deux assertions ont été données dans cette description. La première se situe en dehors du corps de segment. C'est une assertion concernant les paramètres de description qui sera vérifiée lors de l'instantiation du modèle. La seconde concerne la stabilité des entrées de la bascule, à savoir l'horloge "ck" et l'entrée "d". Cette assertion se trouve dans le corps du segment et sera vérifiée pendant la simulation.

- La dernière instruction "if" spécifie la condition de changement d'état de la bascule. Cette condition étant un front montant de ck : ck vaut 1 à (tp-1) et 0 à tp (tp = temps de propagation).

- Les descriptions fonctionnelles utilisent des objets de type VARIABLE. Les

instructions sont exécutées si les conditions associées sont vérifiées (sémantique des langages de descriptions fonctionnelles).

5.2 L'approche VHDL:

Le langage VHDL (VHSIC Hardware Description Language) [SHA85],[ROG 86] a été défini, depuis 1981, comme le langage de spécification de matériel dans le cadre du projet VHSIC (Very High Speed Integrated Circuit), qui est financé par le département de la défense américaine. Ce langage représente une progression dans l'état de l'art en matière de langages de description de matériel (HDL).

Comme CONLAN, VHDL fournit un mécanisme pour permettre les alternatives différentes des représentations en conception. Ce mécanisme est basé sur l'utilisation de l'unité de conception.

Une unité de conception consiste en une interface et un corps de description (séparés comme en ADA). L'interface définit les portes de communication avec l'environnement extérieur. Le corps de description décrit les détails internes d'une unité en utilisant trois catégories d'expression : structurelle, comportementale ou un mélange des deux catégories. L'exemple suivant exprime le concept d'une unité de conception avec deux descriptions d'une bascule "RS".

Exemple

```
-- description d' interface
ENTITY  RS_Latc ( R , S  : in BIT  ;     -- input ports
                           Q , QB : out BIT   -- output ports ) is
 end RS_Latch ;

-- description de corps
  architecture  Body1 of RS_Latch is
   B : block
        component  NAND_Gate  port ( A,B : in BIT;  C : out  BIT ) ;
       begin
            G1 : NAND_Gate port ( S, QB, Q ) ;
            G2 : NAND_Gate port ( R, Q, QB ) ;
       end block ;
    end  Body1 ;
```

```
-- une description alternative
   architecture Body2 of RS_Latch is
     B : block
           begin
              Q <= S nand QB after 10 ns;
              QB <= R nand Q after 10 ns;
           end block;
   end Body2;
```

VHDL est le premier langage utilisant la notion de package (provenant du langage ADA). Le package permet de spécifier : des types de variables, des constantes, des attributs , et un ensemble de fonctions. Pour forcer une description à spécifier explicitement ses portes de communication avec les autres composantes, on ne définit pas de signaux ou variables dans les packages. Une fois que le package est défini, il peut être utilisé par plusieurs unités, la clause "WITH" définit le contexte dans lequel l'unité sera compilée. Un ensemble de packages prédéfinis est fourni avec chaque installation de VHDL. La notion de package représente les primitives du langage, la figure (13) présente un exemple d'utilisation de cette notion.

```
WITH PACKAGE measure ; USE measure ;
PACKAGE static_cmos IS
TYPE cmos_logic_levels IS ( ' strong0 ' ,' strong1', 'weak0','weak1','x','z' ) ;
CONSTANT input_capacitance : capacitance := 0.16 pf ;
CONSTANT metal_1_width : distance := 3.0 um ;
CONSTANT metal_2_width : distance := 4.0 um ;
.....
END static_cmos ;
```

FIG. 13 : NOTION DU PACKAGE EN VHDL

Dans cet exemple l'instruction "WITH" réutilise le package "measure" qui est déjà stocké dans la bibliothèque et contient les différentes unités de mesure de capacité (pf) et de distance (um). Le nouveau package "static_cmos" contient la déclaration du type (les différentes forces en technologie CMOS) et des constantes.

Au lieu de prédéfinir des attributs d'objets dans le langage, VHDL possède des mécanismes permettant aux utilisateurs de les définir. Ils pourront ainsi ajouter dans leurs descriptions des informations qui seront exploitées par des outils

spécifiques. Les attributs sont typés en vue d'une vérification. Ils sont aussi rattachés à une classe de ports d'entrée/sortie. Par exemple l'attribut " fan-out" est rattaché à tous les signaux de sortie.

Exemple :

```
-- Déclaration d'un attribut
 TYPE positive IS integer RANGE 1 TO 30 ;
.....
  ATTRIBUT pin_no : positive ;
.....
-- Spécification
   ATTRIBUT pin_no OF cout : PORT IS 10 ;
```

Dans cet exemple on déclare l'attribut "pin_no" de type entier (de 1 à 30) que l'on associe au signal "cout" du type "port" avec la valeur "10".

VHDL peut créer une seule description de comportement qui inclut les deux codes : séquentiel et parallèle. Quelques langages demandent une séparation stricte entre les deux constructions, d'autres fournissent seulement une parmi les deux. Les instructions de processus (description séquentielle) et de bloc (description parallèle) peuvent être imbriquées.

Comme CONLAN, VHDL utilise des paramètres génériques pour définir une classe de composants : c'est à dire un modèle. Ces paramètres peuvent être liés à une technologie, à des caractéristiques temporelles, à des dimensions, ou à des attribut de type fan-in/fan-out. La valeur de ces paramètres doit être fixée lorsque l'unité est instantiée.

Le langage offre aux utilisateurs beaucoup de souplesse et de liberté par rapport à d'autres langages. Par exemple, le langage permet à l'utilisateur de définir et contrôler la fonction de détection de conflit (dans tous les langages cette fonction est prédéfinie). Aussi, le langage permet à l'utilisateur de définir des types qui sont propres à ses applications et non fournis par le langage. Enfin, le langage offre la possibilité de décrire le système à réaliser dans les différents niveaux d'abstraction dans la même description (un exemple complet qui résume les différents aspects de ce langage est présenté en [SHA 85]).

5.3 Analyse des deux approches :

Nous allons essayer d'analyser les deux approches selon les différentes propriétés : domaine de langage, description hiérarchique, prise en compte du temps, parallélisme, séquencement et description d'architecture.

5.3.1 Domaine de langage :

Les deux approches permettent la description du système logique aux différents niveaux d'abstraction (à partir du niveau système jusqu' au niveau logique). Ceci est permis en CONLAN par la définition d'un langage à chaque niveau et en VHDL par l'utilisation de constructions du langage.

A ce stade, on peut remarquer que le système CASCADE [MER 85] qui est basé sur CONLAN permet la description au niveau switch et électrique (par la redéfinition de langages à partir de BCL).

Les deux approches aussi permettent les différents types de description : combinatoire, synchrone et asynchrone. En plus, VHDL permet le mélange de tous dans la même description.

Les deux approches permettent aux utilisateurs de définir leurs propres types de données. Ils ne sont limités ni à une méthodologie donnée ni à une technologie spéciale ou à un style de conception. Ils peuvent fournir les différents supports (description,vérification et synthèse) pendant le processus de la conception.

5.3.2 Description hiérarchique :

La description hiérarchique (décomposition de structure et abstraction de données) est assurée dans les deux approches. VHDL garde la hiérarchie au niveau interne, pour CONLAN à notre connaissance il est difficile de déterminer cette facilité.

Les deux approches donnent la possibilité de présenter les différentes alternatives de description. VHDL est le seul langage qui permette la séparation de l'interface et du corps de description (comme en ADA).

La création d'une bibliothèque et la réutilisation d'unités sont assurées dans les deux approches.

5.3.3 Prise en compte du temps :

Les deux approches permettent la modélisation temporelle à tous les niveaux de description. En plus VHDL permet l'affectation de chronogrammes et la modélisation du retard inertiel (type du retard par défaut).

5.3.4 Parallélisme et séquencement :

En CONLAN, les deux langages PSCL et BCL sont non procéduraux et les instructions conditionnelles (IF,CASE) peuvent rendre actifs ou inactifs des groupes d'instructions. Par l'extension syntaxique, il est possible de définir des langages d'applications à exécution séquentielle.

VHDL est aussi non procédural et permet la description parallèle et séquentielle. En plus elle permet leur imbrication.

5.3.5 Description d'architecture :

Du point vue de l'architecture les deux approches permettent la description comportementale,fonctionnel et structurelle.

En plus CONLAN permet la description de structure explicite et récursive.

5.4 Conclusion :

Avant de terminer cette partie on doit remarquer les faits suivants : l'analyse de ces deux approches est basée sur un aspect uniquement théorique, car nous n'avons pas pratiqué ou utilisé CONLAN et VHDL ; par cette analyse nous n'essayons pas de favoriser une approche par rapport à l'autre car elles sont différentes à la base et l'ensemble des deux présente une continuité dans le domaine des langages de description de matériel.

L'approche CONLAN offre une méthode de définition formelle pour les langages de description de matériel. Cette méthode repose sur une base syntaxique et

sémantique commune. CONLAN est extensible (définition du nouvel langage à partir de BCL) mais l'utilisation de différents langages aux différents niveaux peut introduire certaine perte dans la continuité de description.

L'approche VHDL offre aux utilisateurs beaucoup de souplesse et de liberté par rapport aux autres langages. VHDL n'est pas extensible, mais la continuité de description est plus assurée. VHDL représente une avance dans l'état de l'art dans le domaine des langages de description de matériel, et est le premier candidat pour être le langage standard de description de système logique.

CONCLUSION

Dans la première partie de ce chapitre on a essayé de mettre l'accent sur les définitions différentes de la conception, la conception assistée et automatisée, et les outils d'un système d'aide à la conception.

Les trois applications principales dans un système d'aide à la conception sont : la synthèse, l'analyse et la documentation . Ces applications se basent sur l'utilisation des langages de description de matériel.

Notre démarche est de construire un langage de description (FIDEL) qui est descriptif aussi bien qu' efficace à utiliser dans le système de la conception assistée. Cette démarche a été basée sur les principes et les idées qu' on a présenté dans la deuxième partie de ce chapitre.

CHAPITRE II

FIDEL : LANGAGE DE DESCRIPTION FONCTIONNELLE

INTRODUCTION

Dans ce chapitre, on va présenter la définition et l'implémentation de **FIDEL** (**F**unctional **I**ntegrated circuits **DE**scription **L**anguage) qui est un langage de description fonctionnelle de circuits intégrés.

D'après la définition de Barbacci [BAR 75], la description fonctionnelle est un intermédiaire entre une description structurelle et une description de comportement, en ce sens que les composants matériels d'un système logique sont représentés mais que les opérateurs peuvent ou non correspondre à des opérateurs matériels. Par contre, le parallélisme et les aspects temporels sont pris en considération. Donc, FIDEL est un langage permettant la description du comportement et/ou la description de structure d'un circuit intégré logique.

Les circuits décrits en FIDEL sont sous forme compilable et exécutable, donc leurs descriptions peuvent être utilisées dans la simulation fonctionnelle et la vérification de composants aux différentes étapes dans le processus de la conception de systèmes logiques.

FIDEL est défini pour décrire les circuits asynchrones, mais la description de circuits synchrones se réalise facilement. Dans la définition et la réalisation du langage FIDEL, nous avons essayé de respecter les objectifs suivants :

- Indépendance vis à vis des technologies et des catégories de la conception.
- Facilité d'utilisation.
- Description fonctionnelle de haut niveau.
- Prise en compte du parallélisme et de l'aspect temporel.

Ce chapitre sera consacré à la discussion de ces objectifs en étudiant la description du comportement en FIDEL, la description structurelle sera présentée dans le chapitre IV.

La première partie de ce chapitre sera consacrée à la présentation des concepts et primitives de base de FIDEL. Dans cette partie, nous allons présenter les raisons de réaliser FIDEL, nous allons montrer les concepts principaux pris en compte pendant sa définition et son implémentation. Nous allons également présenter les primitives concernant les différents types de variables et constantes, les différents opérateurs et fonctions, les instructions de contrôle et les instructions de flux de données.

FIDEL est basé sur la notion de modularité : il représente le circuit en terme de modules différents qui sont conçus comme MODELES dans l'environnement de FIDEL ; cet aspect de modélisation sera le thème de la deuxième partie. Nous allons présenter les trois constructions principales d'un modèle FIDEL (entête du modèle, définition d'éléments et partie description).

La troisième partie sera consacrée à l'aspect temporel en FIDEL. Nous allons représenter les façons différentes de décrire le temps qui sont basées sur les critères du chapitre I.

Enfin, l'aspect informatique de la réalisation de FIDEL (compilateur et interpréteur) sera discuté dans la quatrième partie.

1. CONCEPTS ET PRIMITIVES DU LANGAGE

Le travail concernant la définition et la réalisation de FIDEL a commencé à la fin de l'année 1982. A cette époque le besoin des concepteurs au CNET et à EFCIS était d'intégrer un langage fonctionnel au simulateur logique EPILOG (EPISODE [CHI 76]) pour créer un simulateur logico-fonctionnel et améliorer les performances de la simulation de circuits intégrés (ce thème sera discuté en détail au chapitre III), et en même temps pour disposer d'un langage ouvert en vue de l'interfacer avec le système d'aide à la conception.

Pour répondre à ce besoin, on a étudié certains langages de description de matériel qui existaient à cette époque : ISPS [BAR 81], ADLIB [HIL79], KARL [HAR79] et MODLAN [PAW81]. En conclusion, cette étude montre que deux cas

sont possibles : soit le langage ne répond pas de manière globale aux caractéristiques désirées, soit il est difficile de l'intégrer avec notre système.

Pour ces raisons, il a été décidé de définir et réaliser un nouveau langage pour répondre au besoin des concepteurs et en même temps tenir compte des différents avantages proposés par les autres langages. Dans la suite nous allons présenter les différents concepts de base dans la construction du langage FIDEL.

1.1 Concepts de FIDEL:

En utilisant l'approche descendante de la conception, le premier objectif est de développer les propositions de conception. A partir de cette description générale de la conception, les descriptions fonctionnelles de sous modules ou différentes fonctions de conception sont développées. Il y a une demande pour un langage de spécification de conception qui va permettre au "concepteur" de spécifier la fonction de son système à réaliser et de vérifier de façon fonctionnelle que ce système l'accomplit bien comme désiré. Donc, le premier concept est que **FIDEL est un langage adressé au concepteur (non programmeur)**.

Le constructeur d'un langage de description doit permettre les descriptions comportementale et structurelle. Bien que FIDEL ait été développé avec des accents favorisant la description de comportement, la nécessité de présenter la structure a été prise en compte. Donc, un deuxième concept de FIDEL est de **permettre au concepteur de décrire son système au niveau du comportement aussi bien qu'au niveau de la structure** (cf chapitre IV).

Au lieu d'être un langage procédural ou nonprocédural, FIDEL effectue un compromis en utilisant la notion de modèle. Le concepteur peut définir des modèles séparés. Chaque modèle consiste en des instructions qui s'exécutent de manière séquentielle. De ce fait, un modèle fonctionne comme un programme dans un langage procédural. Cependant, plusieurs modèles peuvent être définis et tous ces modèles peuvent être exécutés en parallèle dans l'environnement de la simulation. Plusieurs applications utilisent cette description de façon naturelle. Donc, un concept primitif dans la définition de FIDEL est **l'implémentation du parallélisme.**

Enfin, FIDEL est défini pour décrire les circuits asynchrones mais la description de circuits synchrones s'effectue facilement. Tous ces concepts seront discutés dans la présentation détaillée du langage.

1.2 Primitives du langage :

Avant de présenter en détail les primitives du langage FIDEL, on décrit les règles syntaxiques comme suit :

- Les symboles non-terminaux (règles) sont présentés en minuscules.
- Les symboles terminaux (mots réservés) sont présentés en majuscules et entre deux ' " ' .
- {.....} : indique que les éléments situés entre les crochets peuvent apparaître 0 ou n fois.
- [.....] : indique que les éléments situés entre les intervalles peuvent être optionnels.
- Le symbole "," sépare les différentes alternatives d'une règle, et le symbole "*" termine chaque règle.
- Les caractères spéciaux sont mis entre ' " '.

1.2.1 Les variables et les constantes :

Toutes les variables en FIDEL sont des variables à mémorisation. Au niveau interne, chaque variable mémorise deux valeurs : l'ancienne et la nouvelle. A partir de cette représentation on peut détecter les différents types de changements (ils seront discutés dans la partie description).

Dans le langage FIDEL on distingue deux classes de variables : les variables d'interface et les variables locales.

Les premières représentent les connexions avec l'extérieur (les autres modèles ou l'environnement de la simulation), ces variables existent dans la liste des arguments d'une entête du modèle (cf. 2.1). Les types de ces variables sont "INPUT" (entrée), "OUTPUT" (sortie), et "INPOUT" (entrée/sortie ou

bidirectionnel). Ces variables de connexions peuvent être présentées sous la forme d'un bit (correspond à un objet matériel type bascule), d'un vecteur (correspond à un objet matériel type registre) ou d'une matrice (correspond à un objet matériel type ROM,RAM ou PLA) ; la syntaxe et la sémantique seront présentées en (2.2) avec des exemples. Il faut remarquer qu'il n' y a pas de limitation de taille de vecteurs indépendamment de la machine hôte.

Les deuxièmes représentent des variables internes du modèle et on les utilise pour exprimer l'état interne de ce modèle. Les types de ces variables sont : "**STATE**" et "**INTEGER**". Le premier peut se présenter comme les variables de la première classe (c'est à dire dans un seul bit, vecteur ou matrice), le deuxième est un scalaire et on l'utilise comme un compteur, un indicateur de boucle ou un intervalle d'un vecteur. **Il faut noter que chaque variable en FIDEL peut correspondre à un objet matériel ou non (définition de description fonctionnelle).**

Il faut noter qu'en FIDEL la dimension d'une variable (vecteur ou matrice) peut être décrite sous deux formes : ascendante ou descendante. Dans les deux représentations le poids fort est toujours à droite dans la déclaration (cf. 2.2).

En général, chaque variable est un nom (identificateur) d'un signal, la règle syntaxique dépend du contexte d'utilisation de ce signal (dans la partie de déclaration ou dans la partie algorithmique) (cf la carte syntaxique de FIDEL en annexe A).

Les constantes en FIDEL existent sous les formes suivantes : décimal, binaire, octal et hexadécimal avec des préfixes correspondants. Par défaut le mode de représentation est non-signé, les autres modes peuvent être présentés en utilisant l'instruction "**MODE**" (cf 2.4). La syntaxe d'une constante est la suivante :

constante ::= " **DEC_NUMBER**" , " **BIN_NUMBER**" ,
"OCT_NUMBER" , " **HEXA_NUMBER**" *

EXEMPLE

la valeur 10 peut être présentée comme suit :

en décimal : 10
en binaire : #B 1010
en octal : #O 12
en hexa-décimal : #H A

Remarque

Dans tous les formats de représentation de constantes, le bit de poids fort est à gauche. Dans la représentation binaire, l'ensemble de valeurs est : 0,1,Ø (transition montante ou descendante), X (indéterminé) et Z (haute impédance).

1.2.2 Opérateurs et fonctions :

Par définition, les opérateurs de FIDEL supposent la représentation en complément à deux (two's complement) pour les valeurs contenues dans les porteurs. Le mécanisme qui est utilisé pour sélectionner les modes différents sera discuté en (1.2.3). Toutes les opérations s'effectuent sur la représentation entière des opérandes, sauf dans le cas où on a besoin de la représentation en chaîne de bits (opérations logiques ou opérations de comparaison sur une représentation supérieure à 32 bits).

Le tableau (2) présente tous les opérateurs de FIDEL dans l'ordre de leur priorité.

Opérateurs arithmétiques :

Les opérateurs arithmétiques "*", "/" et "MOD" effectuent la multiplication, le quotient et le reste de leurs deux opérandes. La longueur du résultat de l'opérateur "*" est la somme des longueurs de ses opérandes. La longueur du résultat de l'opérateur "/" est égale à celle de l'opérande gauche, pour l'opérateur "MOD " c'est celle de l'opérande droit. Le signe (en mode signé) de l'opérateur "*" et "/" est calculé selon les règles arithmétiques normales. Pour l'opérateur "MOD" le signe du résultat est le même que celui de l'opérande droit.

TABLEAU 2

	OPERATEUR	Signification
1	*	Multiplication
2	/ , MOD	Division et le reste
3	+ , -	Addition et Soustraction
4	=,^=,<,<=,>, >=	Opération de comparaison

	OPERATEUR	Signification
5	SHL,SHR, SCL,SCR	Opérations de décalage
6	NOT , ^	non logique
7	AND , &	et logique
8	OR , XOR	Union et disjonction logique

Les opérateurs binaires "+" et "-" calculent la somme et la différence arithmétiques de leurs deux opérandes. Ces opérations demandent que les longueurs des deux opérandes soient égales, si ce n'est pas le cas on fait l'extension de signe pour le plus court jusqu' à ce qu'il corresponde à l'autre opérande. La longueur du résultat dans les deux cas a un bit de plus que la longueur du plus grand opérande.

Les opérateurs de comparaison (=, ^=, <, <=, >, >=) effectuent la comparaison arithmétique entre les deux opérandes. Le résultat sur un bit indique si la relation est vraie (1) ou fausse (0). Comme les opérateurs (+, -), ces opérateurs demandent que leurs opérandes soient de la même longueur ; on fait l'extension de signe si c'est nécessaire. Il faut noter qu'en FIDEL les valeurs booleenes (VRAI et FAUX) se présentent en valeurs binaires (0 et 1).

Opérateurs logiques :

L'opérateur "**NOT**" effectue le complément logique de son opérande (complément à un ou bit à bit). La longueur du résultat est la même que celle de l'opérande.

L'opérateur "**AND**" effectue la conjonction de ses opérandes. La longueur du résultat est égale à la longueur de ses opérandes, si les opérandes ne sont pas de la même longueur, on fait l'extension de l'opérateur qui est le plus court par l'insertion de "0" dans les poids forts (le plus à droite).

Les opérateurs "**OR**" et "**XOR**" effectuent les opérations d'union et de disjonction sur leurs opérandes. La longueur du résultat est la même que les longueurs des opérandes ; si les deux opérandes ne sont pas de la même longueur l'extension se fait par l'insertion de "0" dans les poids forts.

L'opérateur de concaténation et les opérateurs de décalage :

L'opérateur "@" concatène ses deux opérandes, la longueur du résultat est la somme des deux longueurs. On remarque que l'opérande à gauche sera dans les poids forts et l'opérande à droite sera dans les poids faibles, comme le montre la figure (14).

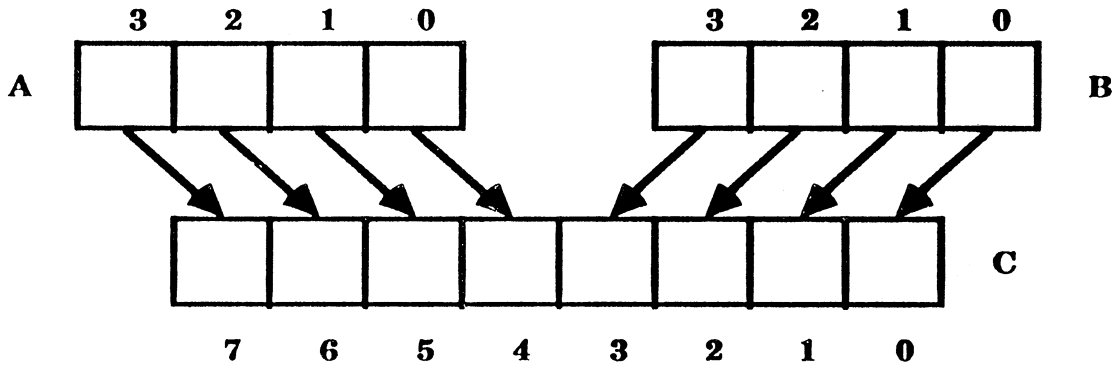


FIG. 14 : C = A @ B

Les opérateurs de décalage sont "**SHL**", "**SHR**", "**SCL**" et "**SCR**" pour le décalage à gauche, le décalage à droite, le décalage circulaire (rotation) à gauche et le décalage circulaire à droite respectivement. L'opérande à droite donne la valeur du décalage, c'est une quantité non signée. La longueur du résultat est la même que celle de l'opérande à gauche. En principe les opérations de décalage en FIDEL s'effectuent de manière arithmétique (division par 2 ou multiplication par 2), c'est à dire que pendant l'opération de décalage à droite, on fait le décalage et chaque fois on répète le bit de signe à la place du bit sortant. Pendant le décalage à gauche, chaque fois qu'on fait le décalage on insère "0" dans le poids faible (la partie la plus à droite). L'exemple suivant présente différents cas de décalage et de rotation.

EXEMPLE	valeur d'opérande	opération	résultat
	1001011	SHR 2	11110010
	01001011	SHR 2	00010010
	01001011	SHL 3	01011000

Les fonctions :

En plus de ces opérateurs, FIDEL offre au concepteur des fonctions prédéfinies. Les fonctions sont les suivantes : fonction logique, fonction de décalage et autres fonctions. Les fonctions logiques et les fonctions de décalage effectuent les mêmes opérations que les opérateurs logiques et de décalage mais avec un nombre d'opérandes supérieur ou égal à deux, les fonctions logiques sont : **FAND , FLOR , FXOR.**

EXEMPLE Fonction " et " logique

si A vaut 1100
B 1110
C 1000

le résultat de **FAND** (A , B , C) est 1000

Remarque :

Si les opérandes ne sont pas tous de même longueur on fait l'extension par rapport à la longueur la plus grande, cette extension est faite par l'insertion de "0" aux poids forts. La fonction "**FLOR**" est pour l'opération "ou" logique et la fonction "**FXOR**" est pour l'opération "ou exclusive".

Les fonctions de décalage sont : "**FSHL**", "**FSHR**", "**FSCL**" et "**FSCR**" pour le décalage à gauche, le décalage à droite, le décalage circulaire à gauche et le décalage circulaire à droite respectivement. Dans ces fonctions le nombre de décalage par définition est d'un seul bit à la fois.

EXEMPLE

si A vaut 1100 et B vaut 1010

FSCR (A,B) représente le décalage circulaire à droite sur la concaténation de A et B et donne comme résultat 01100101.

Les fonctions diverses sont :

- **ODD** : sur un vecteur booléen, vaut "1" si le nombre de bits à "1" est impair.
- **EVEN** : sur un vecteur booléen, vaut "1" si le nombre de bits à "1" est pair.
- **POSL** : poids du premier bit à gauche qui contient "1" dans un vecteur.
- **POSR** : poids du premier bit à droite qui contient "1" dans un vecteur.
- **COUNT** : nombre de bits à "1" du vecteur (résultat entier).
- **DECODE** : fonction de multiplexage.

EXEMPLE

si A (0 : 2) et vaut 101

ODD (A) ====> 0
EVEN (A) ====> 1
POSL (A) ====> 2 (c'est à dire le bit numéro 2)
POSR (A) ====> 0 (c'est à dire le bit numéro 0)
COUNT (A) ====> 2
DECODE(A) ====> 00010000 (sur 2 ** 3 bits).

1.2.3 Instructions de contrôle :

Les instructions de contrôle sont utilisées pour gérer le flux des informations, contrôler sur quelle condition on va exécuter une ou plusieurs fois telle action ou un ensemble d'actions . Ces instructions ont une syntaxe proche de la syntaxe des langages de programmation de type PASCAL. Dans la suite on va présenter cette syntaxe avec des exemples différents.

Instruction IF-THEN-ELSE :

Les interrupteurs sont des éléments de matériel fondamentaux qui sont exprimés en utilisant l'expression IF-THEN-ELSE. La syntaxe est la suivante :

instruction-if ::= "**IF** " condition " **THEN** " bloc [" **ELSE** " bloc] "**ENDIF** " *

Il faut noter que la condition ici est de type statique, et se construit en utilisant les opérateurs de comparaison. La condition peut être une expression simple ou complexe. Le bloc en FIDEL peut être un ensemble d'instructions de contrôle ou d'instructions de flux de données, donc l'imbrication de IF_THEN_ELSE est permise.

EXEMPLE :

```
IF  A > B  THEN   MAKE  X = Y  ENDIF

IF  A > B  THEN   MAKE X = Y
                  ELSE   MAKE  Z = Y
ENDIF

IF A(0) = 1 THEN
            IF A(1) = 1 THEN
                        IF A(2) = 1 THEN
                              .....
                        ENDIF
            ENDIF
ENDIF
```

Le premier exemple signifie qu' il y a un flux d'information de "Y" vers "X" si la condition est vraie, si non les deux porteurs sont déconnectés. Le deuxième exemple présente le cas général en utilisant l'option "ELSE" ; si la condition est vraie (c' est à dire "A" est supérieur à "B") on connecte "Y" avec "X"; si la condition est fausse (c' est à dire "A" est inférieur ou égal à "B") on connecte "Y" avec "Z". On peut dire que l'instruction IF_THEN_ELSE représente le composant matériel

type multiplexeur à deux branches. Le troisième exemple représente l'imbrication d'instructions IF_THEN_ELSE ; cet exemple peut représenter un sous ensemble de conditions pour la connexion entre les différents porteurs (c'est un cas spéciale d'un multiplexeur à n branches).

L'instruction SELECT:

Si un signal est affecté par plusieurs variables, alors toutes les affectations doivent être conditionnées. Dans ce cas on peut utiliser plusieurs instructions IF_THEN_ELSE, ou bien l'instruction **SELECT** qui présente le composant matériel type multiplexeur mais à "n" branches ; la valeur de "n" est déterminée par la taille de variable de contrôle en **SELECT**, comme le montre la syntaxe :

instruction-select ::= "**SELECT**" signal_name **bloc** { "**,,**" **bloc** } "**ENDSELECT**" *

EXEMPLE :

```
SELECT  A ( 0 : 1 )
        MAKE   B  =  A + 1  ,, (* premier cas *)
        MAKE   C =  A  -  1   (* deuxième cas *)
        IF  condition THEN
           .....................
        ENDIF  ,,
        MAKE    B  =  A  (troisième cas *)
        SELECT M (2 : 3 )
        ..................................
        ENDSELECT ,,
        NOOP  (* quatrième cas *)
ENDSELECT
```

La dimension de la variable de contrôle détermine le nombre de branches dans le **SELECT**, dans notre exemple la variable "A" a deux bits donc elle peut exprimer quatre valeurs possibles (0, 1, 2, 3). Chaque branche présente un bloc (bloc de contrôle ou de flux de données) associé à la valeur de la variable de contrôle. Pour la

séparation entre chaque bloc, on utilise le caractère ",,". Comme le montre la syntaxe, il n'y a pas d'étiquettes pour séparer chaque cas. L'ordre d'écriture implique l'ordre d'exécution selon la valeur de la variable de contrôle ; c'est à dire que dans notre exemple, si la valeur de "A" est "0" on exécute la première branche, si elle est "1" on exécute la deuxième ..., si le concepteur ne veut pas faire d'actions pendant certaines branches on peut utiliser l'opérateur "**NOOP**" ou no opération (signifie dans le cas de matériel la connexion d'une branche du multiplexeur à la masse).

L'instruction de boucle :

Pour exprimer des actions régulières ou répétitives, FIDEL offre deux alternatives de boucle, l'instruction **FOR** et l'instruction **WHILE**, la syntaxe de **FOR** est la suivante :

instruction-for ::= "**FOR**" "ID" "=" for_var ("**UPTO**" , "**DOWNTO**")
for_var ["**STEP**" for_var] "**DO**" loop_block "**ENDFOR**" *

La variable de contrôle dans cette instruction est derrière le mot réservé "**FOR**", la valeur de cette variable peut être incrémentée (**UPTO**) ou décrémentée (**DOWNTO**) jusqu' à une certaine limite. Par définition la valeur du pas d'incrémentation (ou de decrémentation) est "1" ou bien on peut utiliser l'option "**STEP**" pour déterminer la valeur de ce pas. Le bloc de "**FOR**" (loop_block) peut être n'importe quel bloc de FIDEL (sauf le bloc "**PAR**", pour les actions en parallèle). Donc l'imbrication de l'instruction "**FOR**" est possible. La variable "for_var" peut être une constante de type entier, une variable de type entier ou une expression dont le résultat est de type entier.

EXEMPLE :

```
FOR I=1 UPTO N DO
        MAKE A(I) = B(I+1)
ENDFOR
```

Cet exemple signifie connecter A(1) à B(2), A(2) à B(3) et ainsi de suite.

Le bloc "**WHILE**" présente les actions répétitives et conditionnées. La différence entre "**WHILE**" et "**FOR**" est que en bloc "**FOR**" on exécute les actions dans le corps de "**FOR**" de manière uniforme selon les valeurs (initiale et finale) que l'on associe à la variable de contrôle, mais pour le bloc "**WHILE**" l'exécution dépend de la validation d'une condition associée à ce bloc ; la syntaxe est la suivante :

instruction-while ::= "**WHILE**" condition "**DO**" loop_block "**ENDWHILE**" *

EXEMPLE :

```
WHILE I <= N DO
        MAKE A(I) = B(I+1)
        MAKE I = I + 1
ENDWHILE
```

Cet exemple effectue le même traitement que dans l'exemple donné en utilisant l'instruction "**FOR**", mais il faut noter que la condition de contrôle est faite de manière explicite en utilisant les opérateurs de relation (comme le cas de IF_THEN_ELSE) et le changement de cette condition est fait aussi de manière explicite (dans l'exemple l'incrémentation de I).

L'instruction de séquencement :

Par définition dans FIDEL l'ordre d'exécution des actions dans les blocs est l'ordre d'écriture (c'est à dire séquentiel) ; mais pour exprimer les actions en parallèle, FIDEL offre aux concepteurs l'instruction "**PAR**", la syntaxe est la suivante :

instruction-par ::= "**PAR**" block_par "**ENDPAR**" *

Le corps de ce bloc peut être n'importe quel bloc FIDEL (sauf les blocs de boucle "**FOR**" et "**WHILE**"). En principe, on exécute toutes les actions en parallèle et les résultats sont validés à la fin ("**ENDPAR**"). Deux cas sont possibles : si on est dans un contexte séquentiel (c'est à dire l'imbrication par un bloc "**SEQ**") la validation est tout de suite ; si on est dans un contexte parallèle (c'est à dire l'imbrication par

un bloc "**PAR**") on continue avec la sémantique de parallélisme. Lorsqu'une même variable est affectée plusieurs fois dans un bloc "**PAR**", on détecte le conflit, on affecte la variable par la valeur "X" et on génère un message d'avertissement.

EXEMPLE :

```
PAR
        MAKE A = B
        MAKE B = A
ENDPAR
```

A la fin d'exécution de cet exemple,"A" reçoit l'ancienne valeur de "B" et "B" reçoit l'ancienne valeur de "A" à l'entrée de bloc "**PAR**".

Quelque fois dans un bloc "**PAR**" on veut exprimer de manière séquentielle certaines actions, pour ça FIDEL offre l'instruction "**SEQ**" qui signifie l'exécution en séquentiel. La syntaxe est la suivante :

instruction-seq ::= "**SEQ**" block "**ENDSEQ**" *

Tous les changements à l'interieur d'un bloc "**SEQ**" sont locaux. Lorsqu'on rencontre "**ENDSEQ**" deux cas sont possibles : si on est dans un contexte séquentiel (l'imbrication par un autre bloc "**SEQ**") on tient compte de changements ; si on est dans un contexte parallèle (imbrication par un bloc "**PAR**") on applique la sémantique de parallélisme.

EXEMPLE :

```
MAKE B = 0 MAKE C = 1
PAR
    MAKE A = C
    SEQ
        MAKE B = 1
        MAKE D = B
    ENDSEQ
    MAKE C = B
ENDPAR
```

Le résultat à la fin de cet exemple est : A = 1, B = 1, C = 0 et D = 1. C'est à dire qu'à la fin de bloc "**SEQ**" on n'a pas pris en compte le changement de B dans l'affectation de C (le cas où le bloc "**SEQ**" imbrique par un bloc "**PAR**").

EXEMPLE :

```
MAKE B = 0 MAKE C = 1
PAR
    MAKE A = C
    PAR
        MAKE B = 1
        MAKE D = B
    ENDPAR
    MAKE C = B
ENDPAR
```

Le résultat à la fin de cet exemple est : A = 1, B = 1, C = 0 et D = 0 : on affecte les deux variables C et D par l'ancienne valeur de B (imbrication de deux blocs "**PAR**").

L'instruction de changement de mode d'opération :

Par définition, le mode utilisé pour les différentes opérations en FIDEL est le complément à deux. On peut changer ce mode par l'utilisation de "**OC**" (pour le complément à un - One's Complement), de "**SM**" (pour la représentation signée - Signe Magnitude) et "**NS**" (pour la présentation non signée - No Signe). Ces modes sont imbriqués. Lorsque l'on trouve un "**ENDMODE**", on retourne au mode précédent. A l'intérieur d'un mode, on peut également retourner au mode complément à deux en utilisant "**TC**" (Two's Complement).

EXEMPLE :

```
MAKE A = B + C   (* ici le mode est TC par définition *)
OC      (* on commence le mode OC *)
    MAKE C = Y - Z
    NS   (* on commence le mode NS *)
```

```
MAKE X = A * B
ENDMODE   (* retour au mode OC *)
.....
ENDMODE (* retour au mode TC)
```

1.2.4 Instructions de flux de données :

Dans la partie précédente on a vu les différents types d'instructions de contrôle : conditionnelles (**IF_THEN_ELSE** et **SELECT**), présentation de boucle (**FOR** et **WHILE**), changement de séquencement (**PAR** et **SEQ**) et changement de mode d'opération (**OC,TC,NS,SM**). Pour exprimer l'algorithme ou la fonction d'un modèle FIDEL, toutes ces instructions contrôlent l'exécution d'autres instructions qu'on appelle opérations de flux de données ; elles incluent : opération d'affectation d'une variable (instruction **MAKE**), commande de trace d'une variable pendant la simulation (instruction **TRACE**), commande de génération de messages pendant l'exécution du modèle (instruction **REPORT**), commande d'examen du contenu d'une variable (instruction **EXAMINE**), et l'opération de génération de macros (instruction **GENMACRO**). Le détail et la syntaxe de chaque opération sont présentés dans la suite de cette partie.

Instruction d'affectation :

L'opérateur d'affectation est le caractère "=" mais il est guidé par le mot réservé "**MAKE**" (en général FIDEL est guidé par les mots réservés, c'est à dire que chaque instruction commence par un mot réservé). L'instruction d'affectation présente le changement d'état d'une ou plusieurs variables. L'opérateur "=" effectue un transfert arithmétique, c'est à dire que le résultat de la partie droite de l'affectation doit avoir la même longueur que la partie gauche en utilisant l'extension de bit de signe ou la troncation (associée à un message d'avertissement). Ce résultat de transfert est le résultat de l'évaluation de la partie droite (avant que la troncation ou l'extension aient eu lieu). La syntaxe est la suivante :

instruction-make ::= "**MAKE**" assignement_var "=" expression *

L'expression peut être une expression simple (une seule variable) ou une expression complexe qui contient plusieurs opérations (logiques ou arithmétiques). La variable d'affectation elle aussi peut être une variable simple (un seule destinataire) ou plusieurs variables concaténées entre elles ou séparées par des virgules. La syntaxe de cette variable est la suivante :

assignement_var ::= signal_name { (" , " , " @ ") signal_name } *

EXEMPLE :

MAKE A = B
MAKE A @ B = C
MAKE A , B , C = 0

Le premier cas présente une affectation simple de la variable "A" par la valeur de "B", le deuxième présente l'affectation de deux variables "A" et "B" concaténées ensemble (par l'opérateur "@" de concaténation) par la valeur de "C". Il faut noter que le poids fort de "C" sera dans "A" et le poids faible sera en "B". Le troisième cas présente une affectation multiple de "A", "B" et "C" par la constante "0".

Bien que l'opérateur de transfert joue un rôle similaire à l'opérateur d'affectation dans le langage de programmation, il faut signaler que l'opérateur de transfert représente une connexion entre les porteurs. C'est la nature du porteur qui détermine la nature de cette connexion. Syntaxiquement il n' y a pas de différence entre le transfert d'un porteur logique (écriture d'une valeur dans une mémoire) et le chargement d'un bus (affecter une valeur pour certaines lignes de connexion).

L'instruction GENMACRO :

FIDEL offre au concepteur le bloc "**MACRO**" pour faciliter la tâche d'écriture dans la partie algorithmique du modèle. Le bloc "**MACRO**" (paramétré ou non) est un ensemble d'instructions FIDEL (instructions de contrôle ou instructions de flux de données) que l'on peut appeler avec son nom et ses paramètres (s'ils existent) en utilisant l'instruction "**GENMACRO**" pour éviter la répétition fastidieuse de cet

ensemble. La syntaxe de bloc "**MACRO**" et l'instruction "**GENMACRO**" est la suivante :

instruction_macro ::= "**MACRO**" "ID" ["(" macro_parameter_list ")"]
bloc "**ENDMACRO**" *

macro_statement ::= "**GENMACRO**" "ID" ["(" macro_parametr_list")"] *

macro_parameter_list ::= "ID" { "," "ID" } *

EXEMPLE :

```
MACRO TOTO
      MAKE A = B + C
ENDMACRO
```

Supposons que pendant la description algorithmique on écrive l'instruction suivante :

IF A > B THEN GENMACRO TOTO **ENDIF**

Cette instruction sera équivalente à l'instruction :

IF A > B THEN MAKE A = B + C ENDIF

Comme on l'a mentionné, on peut associer des paramètres au bloc **MACRO** de la façon suivante :

EXEMPLE :

```
MACRO TOTO (A,B,C)
      MAKE A=B + C
ENDMACRO
```

donc si on écrit l'instruction :

IF X > Y THEN GENMACRO TOTO(X , Y , Z) ENDIF

elle est équivalente à l'instruction :

IF X > Y THEN MAKE X = Y + Z ENDIF

L'instruction de trace :

L'instruction de "**TRACE**" permet la sortie de la trace de variables d'interface ou internes du modèle lors d'une exécution (simulation). On a plusieurs possibilités :

- Trace de toutes les variables :

Syntaxe : **TRACE ON**

Chaque fois que l'on exécute le modèle, on trace les variables qui se modifient jusqu' à la fin de la simulation sauf si on a rencontré "**TRACE OFF**".

- Trace de certaines variables :

syntaxe : **TRACE** A , SUM, CARYOUT

Chaque fois que l'on exécute le modèle, on trace seulement les variables citées ci-dessus qui sont modifiées jusqu' à la fin de la simulation sauf si on a rencontré "**TRACE OFF**".

- Fin de trace :

syntaxe : **TRACE OFF** ou **TRACE OFF** A,SUM

Ceci annule de façon globale ou spéciale les variables à tracer.

- Examen ponctuel :

Syntaxe : **EXAMINE** SUM

On effectue la trace de la variable SUM seulement quand on exécute cette instruction.

REMARQUE :

Dès la prise en compte d'une commande de "TRACE", sur la liste de simulation est indiqué également la liste des heures auxquelles le modèle est appelé (le temps courant de simulation).

L'instruction REPORT :

L'instruction de message peut être utilisé comme assertion statique dans la description du modèle. Ce message peut être associé à une condition sur certaine variable, ou peut être présenté seul comme indication de flux d'exécution d'un certain bloc. La syntaxe est la suivante:

message_statement::="**REPORT**" "**THAT**" """caracter_string """ [signal_name] *

EXEMPLE :

```
IF SUM < 0 THEN
    REPORT THAT ' ADDITION RESULT IS NEGATIVE ' SUM
ENDIF
```

EXEMPLE :

```
SELECT INSTR(0:3)
    MAKE ACC = A + B
    REPORT THAT ' CURRENT INSTRUCTION IS : ADD ' ,,
    MAKE ACC = A - B
    REPORT THAT ' CURRENT INSTRUCTION IS : SUB ' ,,
    .....
ENDSELEC
```

2. LE MODELE FONCTIONNEL

FIDEL étant basé sûr un concept de modularisation, il permet au concepteur de présenter un système en fonction de différents modules conçus comme modèles dans l'environnement de description. Il n' y a pas de restriction sur la taille d'un modèle, un modèle peut représenter un système complet (ça peut être le cas au niveau le plus haut dans la hiérarchie) ou bien une partie d'un système. Il y a trois types de modèle en FIDEL : le modèle fonctionnel, le modèle structurel et le modèle global. Dans cette partie on va discuter les principes du modèle fonctionnel, les deux autres types seront présentés au chapitre IV.

Chaque modèle FIDEL est constitué de trois parties : l'entête du modèle, la définition et déclaration d'éléments et la description principale.

2.1 L'entête du modèle :

L'entête du modèle a la syntaxe suivante

partie_entête ::= "**FMODEL**" model_name "(" argument_list ")" " ; " *

On commence donc l'entête du modèle par le mot réservé "**FMODEL**" , suivi du nom par lequel le modèle est identifié dans le système. La liste de paramètres formels représente la communication de ce modèle avec le monde externe. L'ordre d'écriture des paramètres dans cette liste n'a pas d'importance, mis à part le fait qu'il doit être le même que les paramètres effectifs au moment où l'on fait l'interface avec le modèle. L'activation d'un modèle fonctionnel est dû au changement de valeur d'un ou plusieurs éléments d'interface (évidement de type "**INPUT**" ou "**INPOUT**") ou au changement interne (effet d'élément temporel). Au point de vue syntaxique, l'entête se termine par le caractère " ; ".

EXEMPLE :

FMODEL ADD (CARYIN , A , B , SUM , CARYOUT) ;

Dans cet exemple on crée un modèle fonctionnel de nom "ADD" et cinq paramètres d'interface (jusqu' à maintenant le type de chaque élément n'a pas encore été

défini). La définition complète de chaque élément sera faite dans la partie suivante (partie de définition et déclaration).

2.2 Définition des éléments :

Dans cette partie on spécifie les attributs de toutes les variables utilisées dans la description du modèle. Ces attributs incluent le type (**INPUT, OUTPUT, INPOUT, STATE et INTEGER**), la dimension (un seul bit, vecteur de bits ou une matrice) et en option la valeur initiale.

La déclaration de type de chaque variable impose des vérifications et restrictions quant à son utilisation dans le modèle. Par exemple, la variable de type "**INPUT**" ne doit ni être initialisée dans le modèle, ni apparaître dans la partie gauche d'une opération d'affectation. De même, une variable interne (de type "**INTEGER**" ou "**STATE**") ne doit pas apparaître dans l'entête du modèle. Toutes ces vérifications provoquent des messages en cas d'erreurs pendant la phase de compilation.

EXEMPLE :

```
DECLARE INPUT CARYIN , A ( 0 : 7 ) , B ( 0 : 7 ) ;
DECLARE OUTPUT CARYOUT , SUM ( 0 : 7 ) ;
DECLARE STATE MEM ( 0 : 255 , 0 : 7 ) ;
DECLARE INTEGER INDEX ;
DECLARE INPOUT BUS ( 0 : 7 ) ;
```

Dans cet exemple, on présente les différents types de déclaration en FIDEL. Dans la première ligne on déclare trois variables de type "**INPUT**", la première (CARYIN) est définie sur un seul bit, les deux autres variables (A , B) sont de type vecteur défini sur 8 bits. Comme on l'a mentionné avant il faut noter que le poids fort est toujours à droite dans la déclaration.

La deuxième ligne représente la déclaration de deux variables de type "**OUTPUT**", la première (CARYOUT) est définie sur un seul bit, et la deuxième (SUM) est un vecteur de longueur 8 bits. Dans la troisième et quatrième lignes on trouve la déclaration de variables internes de type "**STATE**" et "**INTEGER**". La

variable de type "STATE" est une mémoire de 256 mots de longueur 8 bits. La dernière ligne représente la déclaration d'une variable de type "INPOUT" et de longueur de 8 bits.

REMARQUE :

En FIDEL l'ordre des déclarations n'a pas d'importance.

Dans certaines applications il est très intéressant d'avoir la possibilité d'initialiser certaines variables avant l'exécution du modèle. Cette possibilité existe en FIDEL par l'instruction "INITIALIZE". La syntaxe est la suivante :

instruct_initialise ::= "INITIALIZE" signal_name_init { " , "
signal_name_init "TO" init_value " ; " *

L'exemple suivant montre les différentes façons d'utiliser cette instruction.

EXEMPLE :

```
INITIALIZE  CARYOUT , SUM  TO  0 ;
INITIALIZE  INDEX TO  1 ;
INITIALIZE   MEM  TO #B X ;
INITIALIZE   MEM ( 1 , 0 : 7 ) TO #B10101010 ;
INITIALIZE   MEM ( 2 , 0 : 7 ) TO #H AF ;
INITIALIZE   BUS  TO #B  Z ;
```

La première et la deuxième lignes représentent l'initialisation de variables CARYOUT, SUM et INDEX en utilisant les constantes de type décimal (0 et 1 respectivement). Dans la troisième ligne, on initialise la mémoire MEM à la valeur binaire "X" (c'est à dire la valeur inconnue). Il faut noter qu'on peut initialiser une mémoire de façon globale soit à la valeur "0" soit à la valeur "X" ; si on a besoin d'initialiser certains mots on fait comme le montre la quatrième et la cinquième lignes (la quatrième ligne représente un vecteur binaire et la cinquième un vecteur hexadécimal). La dernière ligne représente l'initialisation d'une variable BUS de type INPOUT à la valeur binaire "Z" ou haute-impédance.

FIDEL offre aussi l'initialisation dynamique, mais le type de variable à initialiser doit être booléen (type **STATE , OUTPUT** et **INPOUT**)et défini sur un seul bit. La valeur dynamique est soit le front montant "**RISES**" soit le front descendant "**FALLS**". L'exemple suivant montre les deux types d'initialisation :

EXEMPLE :

```
DECLARE STATE PH1, PH2;
INITIALIZE PH1 TO RISES ;
INITIALIZE PH2 TO FALLS ;
```

La signification de la valeur "**RISES**" est de mettre "0" dans la valeur ancienne et "1" dans la valeur nouvelle. La signification de la valeur "**FALLS**" est l'inverse de cette affectation.

REMARQUE :

La déclaration d'un bloc "**MACRO**" (s' il existe) doit être faite après la partie de définition et avant la partie de description.

2.3 Partie description :

La partie description est spécifiée comme un algorithme qui représente la relation entre un changement d'une entrée et les événements à générer à la sortie (relation de boite noire). L'activation d'un modèle fonctionnel est contrôlée par les événements d'entrée et les événements internes (présentés dans la partie suivante) qui sont pris en compte par le mécanisme de simulation (simulation dirigée par les événements).

Dans la conception de tous les systèmes logiques, il y a un ensemble d'éléments de base qui peuvent être utilisés pour présenter la description fonctionnelle. Ils sont :

- une cause : expression d'événements .
- un effet : liste d'actions.
- un ensemble de conditions.

- un intervalle du temps.
- une action nulle ou no opération.

La cause représente le signal de déclenchement pour que le système commence à fonctionner ; en général c'est un évènement externe au système (changement d'entrée) qui représente cette cause. Pour un système combinatoire, un changement d'entrée est suffisant pour que le système fonctionne. Par contre, pour un système séquentiel, on doit tenir compte de certaines conditions internes (les différents états du système). Donc, un ensemble de conditions doit exister en parallèle avec la cause pour représenter un ensemble complet (externe et interne) pour le déclenchement du système.

La liste d'actions représente l'ensemble des opérations effectuées par le système pour générer les événements de sortie. Cette liste est exécutée quand la cause est active et que l'ensemble des conditions est vérifié ; cette exécution est faite après un certain intervalle de temps.

A notre avis cet ensemble de base est suffisant et nécessaire pour représenter un système logique ; **la présence de cet ensemble peut être utilisé comme mesure d'efficacité d'un langage de description.**

En FIDEL, pendant la phase de définition et d'implantation du langage, on a essayé de respecter cet ensemble de base. Le bloc principal en FIDEL est le bloc "**WHEN**" qui contient cet ensemble de base. La syntaxe de bloc "**WHEN**" est la suivante :

bloc_when ::= "**WHEN**" event_clause **block** " ; " *

La description fonctionnelle en FIDEL peut être représentée par un ou plusieurs blocs "**WHEN**". Les événements représentent les conditions primaires qui doivent être satisfaites pour déclencher la liste d'actions.

Les différents types d'événement en FIDEL sont :

- Evènement dynamique :

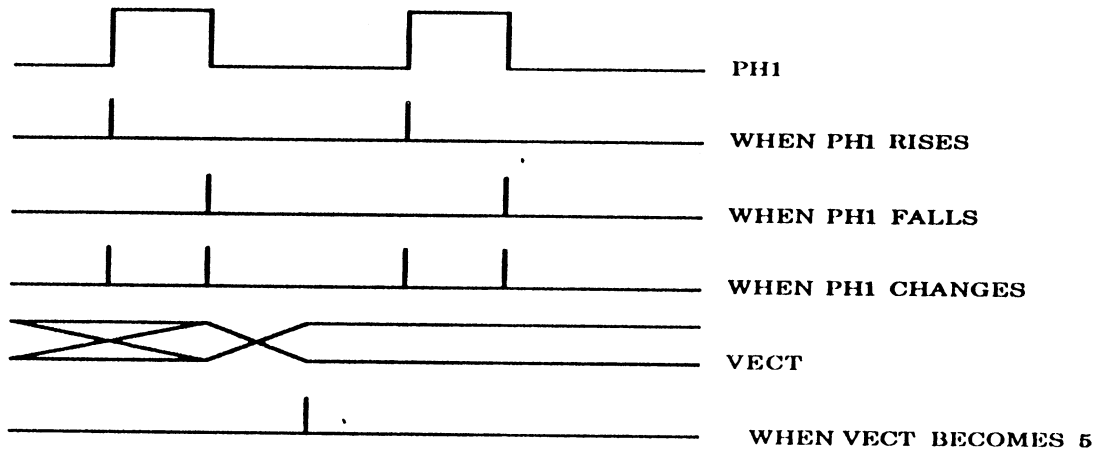
qui exprime une transition dynamique d'état d'une ou plusieurs variables :

RISES PASSAGE à 1)
) pour variable sur un bit
FALLS PASSAGE à 0)

CHANGES changement d' état)
) sur un bit ou vecteur
BECOMES changement spécifique)

EXEMPLE :

WHEN PH1 RISES .....
WHEN PH1 FALLS .....
WHEN PH1 CHANGES .....
WHEN VECT BECOMES 5 .....

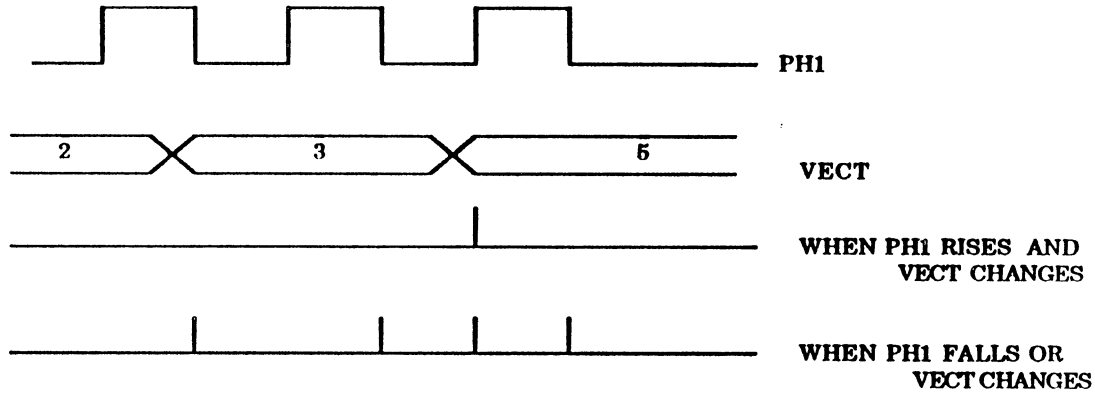


REMARQUE:

Tous ces types peuvent être combinés dans une seule expression en utilisant les opérateurs logiques (**AND, OR, XOR**).

EXEMPLE :

WHEN PH1 RISES AND VECT CHANGES .....
WHEN PH1 FALLS OR VECT CHANGES .....



EXEMPLE :

```
WHEN CLOCK CHANGES
    WITHIN 10 TO 10 MAKE CLOCK = NOT CLOCK;
```

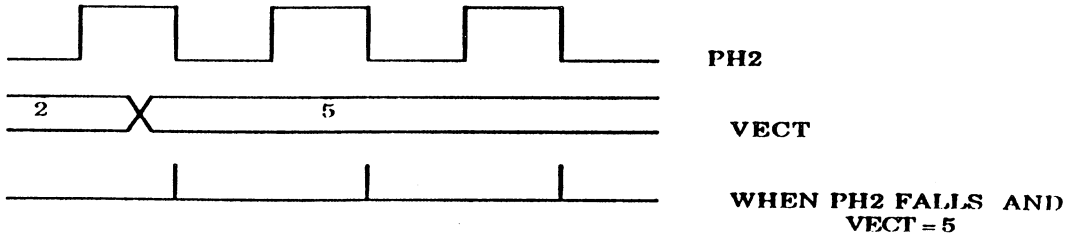
Cet exemple représente un **générateur d'horloge (CLOCK)** en utilisant l'événement "**CHANGES**" et l'**affectation avec retard (on va discuter les** principes du retard dans la troisième partie).

- Evènement filtré :

C'est un évènement dynamique qui est **filtré par une condition statique** en utilisant seulement l'opérateur logique "**AND**" ; **la condition statique** est constituée par des opérations de **comparaison.**

EXEMPLE :

WHEN PH2 FALLS AND VECT = 5 .....

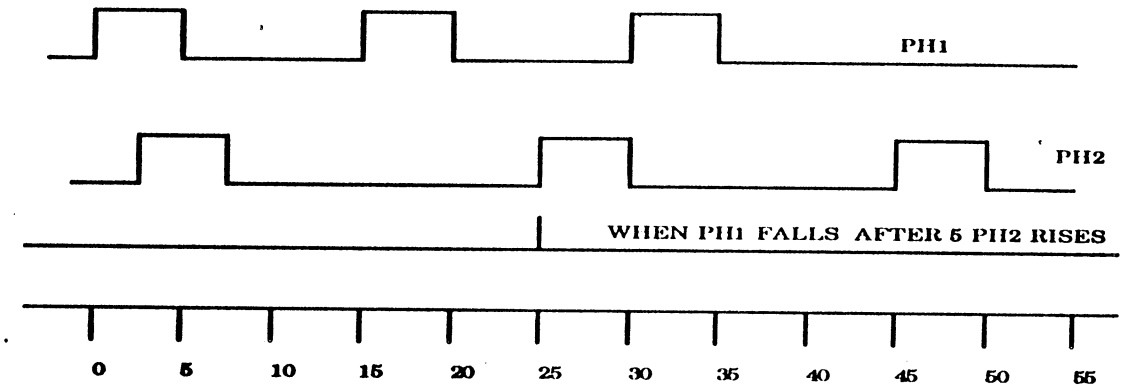


- **Evènement en cascade :**

qui représente une séquence d'événements qui sont séparés par une période de temps ; il est spécifié en utilisant l'opérateur "AFTER".

EXEMPLE :

WHEN PH1 FALLS AFTER 5 PH2 RISES.....



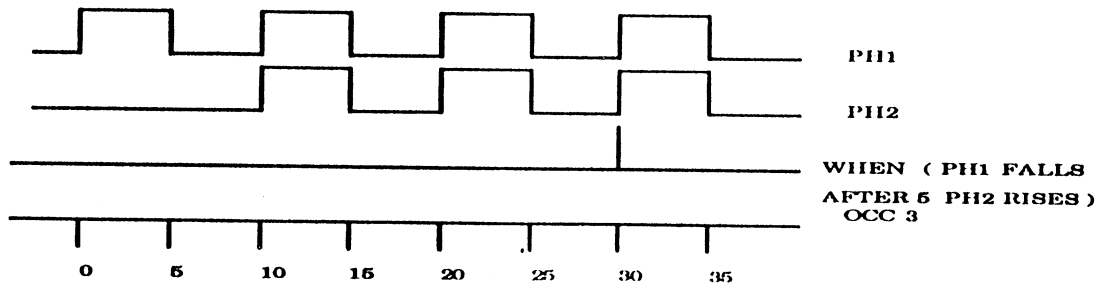
- **Evènement répétitif :**

qui représente une occurrence multiple de l'expression d'évènement en utilisant l'opérateur "OCC".

EXEMPLE :

WHEN (PH1 FALLS AFTER 5 PH2 RISES) OCC 3 ...

Cet exemple représente la troisième occurrence d'un évènement dynamique en cascade (cet évènement signifie l'occurrence de front montant de PH2 5 unité de temps après le front descendant de PH1).



- Evènement nommé :

c'est un évènement qui est référencé par nom à la place d'une expression, l'événement nommé est créé dans la liste d'actions en utilisant l'instruction "**GENEVENT**". Il faut noter que ce type d'événement n'est valide que pendant l'exécution courante du modèle.

EXEMPLE :

```
WHEN PH RISES
.....
IF A > B THEN GENEVENT EV1 ENDIF
..... ;

WHEN EV1
..... ;
```

La liste d'actions est représentée par les instructions de contrôle et de flux de données (cf. 1.2.3 et 1.2.4).

Les conditions statiques internes sont représentées par les instructions "**IF_THEN_ELSE**" et "**SELECT**". On termine cette partie par un exemple complet d'un aditionneur de 8 bits.

EXEMPLE :

```
FMODEL ADD ( CARYIN, A , B , CARYOUT , SUM ) ;
DECLARE INPUT CARYIN , A( 0 : 7 ) , B ( 0 : 7 ) ;
DECLARE OUTPUT CARYOUT , SUM ( 0 : 7 ) ;
INITIALIZE CARYOUT , SUM TO 0 ;
FUNCTIONAL
WHEN CARYIN CHANGES OR A CHANGES OR
        B CHANGES
          MAKE CARYOUT @ SUM = A + B + CARYIN ;
ENDFUNCTIONAL
ENDMODEL
```

3. PRISE EN COMPTE DU TEMPS

La plupart des langages de description de systèmes logiques ayant été défini en vue de permettre la simulation des circuits décrits, ils incluent des notions temporelles. En général le temps intervient sous trois formes essentielles [BOR 81], qui caractérisent assez bien le niveau de modélisation :

- Aux niveaux les plus abstraits sous la forme de durées d'exécution.
- Aux niveaux intermédiaires sous la forme d'horloges pour la synchronisation.
- Aux niveaux les plus fins sous la forme de retards de transmission.

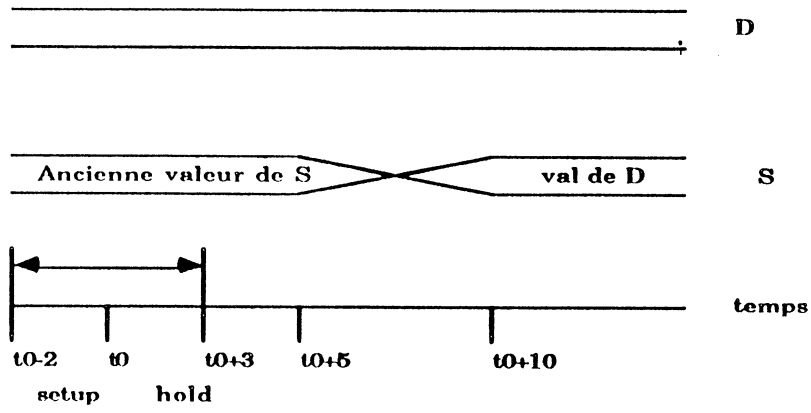
En FIDEL on définit une seule forme permettant d'exprimer la majorité des cas, c'est le bloc "**WITHIN**". Il y a deux types de retard en FIDEL : retard normal et retard à priorité. De plus FIDEL permet la vérification du temps d'établissement (set up), du temps de maintien (hold) et de la largeur de l'impulsion. Le bloc "**WITHIN**" peut être associé à une seule action (niveau le plus fin) ou à un ensemble d'actions (niveau abstrait). La syntaxe est la suivante :

bloc_within ::= ("**WITHIN**" , "**WITHINP**") trimmer "**TO**"
trimme block "**ENDWITHIN**" *

Dans cette règle le terme "trimmer" représente les valeurs associées à chaque instruction dans le bloc (ces valeurs peuvent être des constantes ou des paramètres).

EXEMPLE : niveau plus fin

```
WITHIN 5 TO 10
    MAKE S = D [2,3]
ENDWITHIN
```



L'heure t0 étant l'heure courante, l'affectation indiquée se produira entre les heures t0+5 et t0+10 (la valeur de S est X dans cet intervalle, mais stabilisée à l'heure t0+10). L'intervalle (5 à 10) représente la durée de changement de S. Si pendant cette période S est accédé en lecture un message d'avertissement est envoyé.

De plus le temps (t0-2) est le temps d'établissement (set up) et le temps (t0 +3) est le temps de maintien (hold) pour la variable D. Donc la période (t0-2 à t0 + 3) est la durée de stabilisation de D ; si la valeur de cette variable change dans cette durée, un message d'avertissement va être généré.

EXEMPLE : plus abstrait

```
WITHIN 5 TO 10
    action 1...
    action 2...
    .....
    action n...
ENDWITHIN
```

Dans cet exemple toute les actions s'effectuent au même temps : T+10 où T est le temps d'activer le bloc **WITHIN** (une autre possibilité pour décrire le parallélisme en FIDEL). Le temps d'exécution global d'un modèle ne peut être exprimé de manière directe (c'est à dire que l'attribut "DELAY" n'existe pas au niveau du modèle). Pour l'exprimer, soit le temps d'exécution est obtenu comme la somme de temps élémentaires (plusieurs instructions "**WITHIN**"), soit il est associé à l'affectation de la sortie en utilisant "**WITHIN**" (s'il n'existe pas le bloc "**WITHIN**"

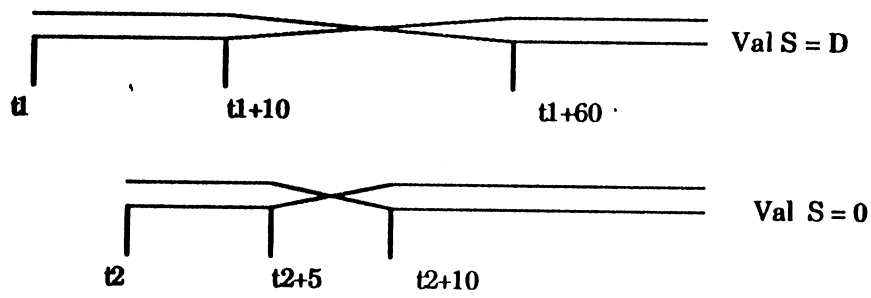
dans la description FIDEL, l'exécution se fait en temps nul).

Le deuxième type de retard est le retard à priorité, cette notion est introduite pour exprimer une priorité entre deux affectations sur la même variable ; son fonctionnement est mis en évidence dans l'exemple suivant :

EXEMPLE :

```
WHEN CLOCK RISES AND RESET = 0
    WITHIN 10 TO 60
        MAKE S = D
    ENDWITHIN ;
WHEN RESET BECOMES 1
    WITHINP 5 TO 10
        MAKE S = 0
    ENDWITHIN ;
```

On suppose en outre que les temps de déclenchements t1 de l'action **WITHIN**, et t2 de l'action **WITHINP** sont ainsi .



Dans cet exemple, il existe une portion de recouvrement des zones de changement d'état (zone X). La règle suivante s'applique alors : si la borne supérieure t2+10 de l'action **WITHINP** est inférieure à celle de l'action **WITHIN** (ici t1+60), alors l'action **WITHINP** étant prioritaire elle annulera l'affectation de S à t1+60 ; et dans ce cas à partir de t1+20 on aura S = 0.

REMARQUE :

En FIDEL, il n'y a pas de choix d'unité de temps (c'est à dire en nano-seconde par exemple). Chaque valeur de temps (5 , 10 ou 20) est relative par rapport au

temps courant de simulation. Il faut noter qu'on détecte le conflit sur l'affectation d'une variable en même temps (même valeur du retard) avec deux valeurs différents (si il n'existe pas de priorité).

L'existence de retard dans la description génère des événements internes qui sont pris en compte par la simulation (géré par les événements).

4. OUTILS D'EXPLOITATION DE FIDEL

Dans cette partie on va présenter les outils informatiques autour de la première version de FIDEL qui ne comprend que la représentation fonctionnelle ; l'organisation générale comme le montre la figure (15) est constituée par le compilateur et l'interpréteur.

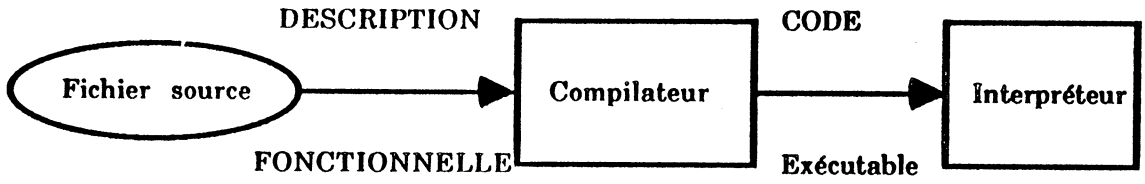


FIG. 15 : ORGANISATION GENERALE DE FIDEL

Le compilateur FIDEL est écrit en langage PASCAL ; il contient 15000 lignes de source découpées en 20 modules. A partir d'un fichier source contenant la description du circuit en FIDEL, le compilateur génère un fichier code interprétable pour l'interpréteur. Le compilateur réalise les fonctions suivantes : vérification lexicale et détection d'erreurs syntaxiques, détection d'erreurs sémantiques, et génération de code interprétable.

La première fonction est effectuée par l'analyseur du langage qui interprète des tables contenant la grammaire du langage. Cet analyseur est construit à partir d'un transformateur de grammaire [DEL 80] type LL1, c'est à dire de grammaire où la connaissance d'un seul symbole permet de décider quelle est la règle de production qui doit être choisie (analyse descendante de gauche à droite sans retour arrière déterministe sur un symbole unique). Ce transformateur génère les tables à interpréter à partir des règles de production de la grammaire du langage.

L'utilisation de cette technique permet d'avoir une implémentation rapide du langage et en même temps permet l'insertion des actions de compilation en tout point des règles de la grammaire, afin d'effectuer les contrôles sémantiques et la génération de code exécutable. L'inconvénient principal du transformateur utilisé est que dans le cas où une erreur grave de syntaxe est détectée, l'analyse s'arrête.

Un schéma de fonctionnement du compilateur est présenté en figure (16). A partir d'un fichier source, l'analyseur lexical découpe la description en unités lexicales. Après la vérification lexicographique, l'analyseur lexical fournit cette unité à l'analyseur syntaxique pour déterminer la règle de production ; la vérification syntaxique se fait en tenant compte de la relation entre l'unité lexicale courante, l'unité lexicale précédente et la règle de la grammaire. Chaque fois que l'analyseur détecte une erreur il génère un message d'avertissement.

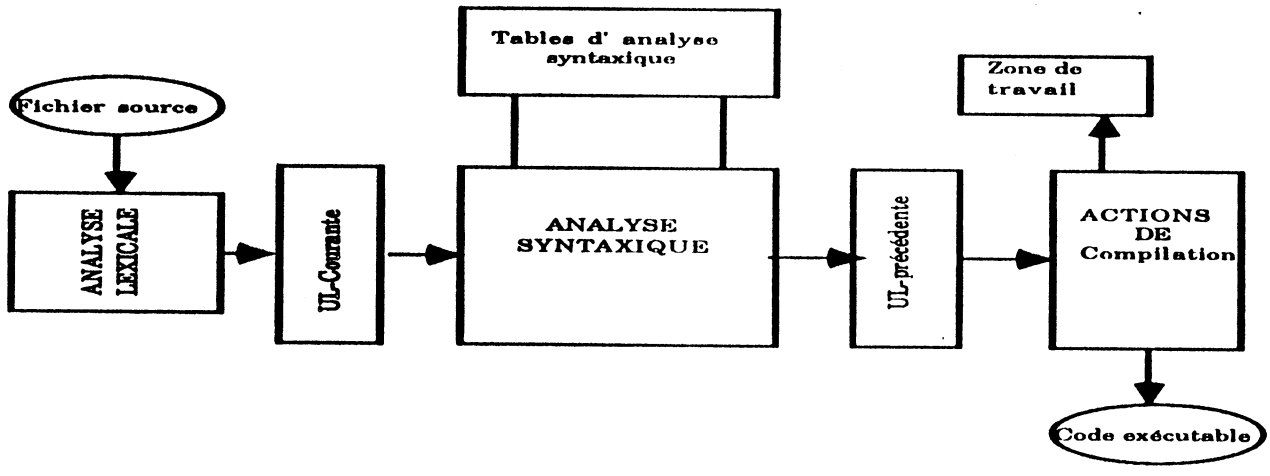


FIG . 16 : SCHEMA DE FONCTIONNEMENT

L'analyseur sémantique en utilisant les actions de compilation effectue la vérification sémantique et génère le code interprétable. Ce code est dans la forme postfixée et est stocké dans un fichier spécial pour l'interpréteur du langage. Ce fichier contient les informations suivantes :

- la table de description du modèle : contient toutes les informations concernant le modèle comme : type, nombre d'entrées, nombre de sorties,

nombre d'entrée/sorties, différents pointeurs vers la zone des éléments, la zone des variables et la zone de code exécutable.

- la table des éléments : représente la liste d'arguments déclaré dans le modèle, cette table contient un pointeur vers la zone des variables.

- la table des variables : contient toutes les informations de chaque variable comme son type, ses valeurs (ancienne et nouvelle), un pointeur vers la table de dimension, sa date de dernière modification et des champs spéciaux pour la vérification temporelle.

- l'échéancier interne : est une table dynamique générée seulement s'il existe le bloc "**WITHIN**" dans la description. Dans cet échéancier sont stockées des informations comme la date de début de changement, la date de fin de changement, l'adresse de variable et la valeur à affecter.

- la table de code interprétable : contient deux champs, le champ d'opérateur et le champ d'opérande. (voir l'exemple après).

En plus de ces informations, il y a des tables spéciales pour stocker les différents modes d'opération, le parallélisme et l'extension de variables.

L'interpréteur FIDEL est décrit en FORTRAN en 9 modules. Chaque module contient des sous programmes associés à chaque opération en FIDEL.

EXEMPLE : on suppose que A, B et C sont des vecteurs

MAKE A = B + C

après la compilation cette instruction est transformée comme suit

champ_opération	champ_opérande
LOAD	adresse_A
LOAD	adresse_B
LOAD	adresse_C
ADD	identification
STORE	identification

Le codage du champ_opération s'effectue sous forme d'entier ; par exemple : LOAD (soit 2 soit 3 si la variable est sur un seul bit ou un vecteur, dans notre exemple sera 3) ADD est (6), STORE est (17). Le deuxième champ contient des informations complémentaires : l'adresse de la variable, le type d'opération (entre des vecteurs ou des variables simples .....).

Chaque fois que l'interpréteur est activé pour l'exécution d'un modèle, il passe par les étapes suivantes :

- mettre à jour les valeurs d'entrées, et effectuer les événements internes (c'est à dire affectation des variables associées aux retards).

- aller chercher l'action à effectuer répérée par le compteur d'action.

- analyser l'action pour déterminer quelle opération à exécuter et avec quels opérandes.

- exécuter l'opération par l'appel au sous programme correspondant à cette opération.

- faire progresser le compteur d'action et aller à l'étape 2.

L'exécution se continue jusqu' à ce que l'on rencontre l'instruction "ENDMODEL" codé en entier (99). Chaque fois qu'il y a des changements d'entrées ou des événements internes, on appelle l'interpréteur et on répète les mêmes étapes.

CONCLUSION

La représentation de la description FIDEL comme une boite noire assure l'indépendance du langage vis à vis des technologies et des catégories de la conception. Comme chaque instruction de FIDEL est guidée par un mot réservé, cela facilite l'utilisation du langage. En plus la structure de base du modèle fonctionnel comme un ensemble de blocs "**WHEN**" est bien adaptée à l'ensemble de base (défini en 2.3) pour décrire un système logique. Le temps et le parallélisme sont définis de manière simple et efficace et peuvent être utilisés à tous les niveaux de la description.

FIDEL est un langage fonctionnel nonprocédural. L'instruction "**WHEN**" avec les différents types d'événements présente bien cette propriété qui est convenable pour décrire les différents formes de contrôle (par exemple **RESET** et **INTERRUPT**). Les différentes instructions de contrôle et de flux de données assurent la possibilité de décrire les actions procédurales dans le langage. On peut facilement décrire les circuits asynchrones, mais la description de circuits synchrones est aussi assurée.

Enfin, on peut dire que la définition et l'implantation de FIDEL dans l'état actuel sont basées sur la discussion présentée dans le chapitre I sur les propriétés des langages de description (la majorité de ces propriétés est respectée) et l'utilisation réelle de concepteurs du langage.

CHAPITRE III

APPLICATIONS DU MODELE FONCTIONNEL DANS LA SIMULATION DE CIRCUITS VLSI

INTRODUCTION

La complexité des circuits VLSI nécessite la vérification de leur conception avant la fabrication ; la simulation semble rester l'outil le plus pratique pour cette vérification. L'objectif de la simulation est de vérifier que le système va fonctionner correctement dans un environnement opérationnel. La simulation facilite la modélisation et la préparation du test pour le circuit à réaliser.

Les différentes formes de simulation peuvent être catégorisées soit en fonction du niveau de la description (fonctionnel, logique et électrique), soit en fonction du mode de manipulation du temps (discret ou continu).

Dans ce chapitre on va discuter en détail le premier objectif de la simulation comme outil de vérification de conception. Le deuxième objectif concernant le test est en dehors du sujet de cette thèse.

La première partie sera consacrée à la discussion des différents concepts et techniques de simulation, des différents types de simulation (soit en fonction du niveau de la description, soit en fonction de la manipulation du temps), des différents types de retards, et des mécanismes de flux du temps.

La deuxième partie sera consacrée à l'intégration du modèle fonctionnel FIDEL dans le simulateur logique EPILOG [CHI 76]. Dans cette partie on va discuter en détail le concept et le rôle de la simulation logique, les problèmes de ce type de simulation et les différentes solutions. La notion de simulation logico-fonctionnelle (simulation multi niveau) sera le thème principal de cette partie. On va présenter l'interface entre FIDEL et EPILOG en discutant tous les points principaux de cette approche. On termine cette partie par un exemple complet pour montrer l'efficacité et l'importance de cette approche.

La troisième partie sera consacrée à la présentation d'une autre application du modèle fonctionnel FIDEL en discutant l'approche de la simulation en mode mixte (mixed-mode simulation). Cette approche est basée sur l'interface entre le modèle fonctionnel FIDEL et le simulateur électrique ELDO [HEN 85a], [HEN 85b]. Dans

cette partie on va présenter les différents mécanismes d'implantation de cette approche, l'importance de cet outil pour vérifier les circuits mixtes (analogique/ numérique).

1. LA SIMULATION DE CIRCUITS VLSI

La simulation est une méthode pour calculer le comportement d'un système dans un environnement spécifié par le concepteur. La simulation permet en outre des vérifications de performance, de bon fonctionnement logique (ou analogique), de bonne synchronisation, d'absence d'aléas dus à des retards trop importants pendant la transmission des signaux dans le circuit, et de détection de défauts.

En général, selon l'objectif, on peut classer la simulation en deux catégories principales : la simulation de détection de défauts, et la simulation de vérification normale.

La simulation de détection de défauts est utilisée pour examiner le comportement d'un circuit qui a un ou plusieurs éléments défectueux, et elle est associée seulement à la génération du test. Les simulateurs de défauts sont utilisés pour mesurer l'efficacité du modèle du test, c'est à dire pour mesurer si les éléments défectueux peuvent être détectés ou non par le testeur. Ce type de simulation est en dehors du sujet de cette thèse.

En général, la simulation de la vérification normale est la catégorie de simulation la plus utilisée dans le processus de conception. Pour mieux discuter les différents types de cette simulation, on regarde d'abord l'expression générale d'un système logique (analogique) qui peut être présentée sous la forme suivante :

$$S = F (E , V , T)$$

Cette forme présente le système comme une boite noire, c'est à dire que l'ensemble des sorties (S) s'obtient par la fonction (F) en fonction des paramètres : E (l'ensemble d'entrées), V (l'ensemble de variables internes) et T (l'ensemble d'éléments temporels).

En général la simulation de vérification peut être classée selon les caractéristiques suivantes:

- Les différents niveaux de représentation de la fonction F dans la hiérarchie de la conception.
- Les méthodes ou techniques de manipulation du temps T, et le type de retard associé à chaque partie du circuit.
- Le nombre de valeurs considérées pour la représentation de E, S, et V.
- Les différents algorithmes pour manipuler la fonction F.

Il faut noter que ces caractéristiques ne sont pas indépendantes, c'est à dire que le choix d'une caractéristique donnée peut affecter les choix des autres.

1.1 Les niveaux de simulation :

Il existe un lien fort entre la simulation et la modélisation (c'est à dire la description de circuits). Les niveaux de simulation sont donc les mêmes que les niveaux de description. Le fonctionnement de la simulation varie selon le niveau de présentation, donc on peut trouver les simulateurs suivants :

- Au niveau système :
  le rôle de la simulation est de vérifier le fonctionnement en étudiant le dialogue de communication, la synchronisation de messages entre les différents processus, la création dynamique de processus et la notion de file d'attende. A ce niveau on utilise des simulateurs de type général "general purpose" comme GPSS [GOR69], Simscript [KIV69] et Simula [DAH66].

- Au niveau fonctionnel :
  le rôle de la simulation est de vérifier le flux de données entre les modules, le comportement de chaque module, la synchronisation de dialogue entre eux et l'aspect temporel associé à chaque module. Les exemples de simulateurs à ce niveau sont : LASSO [BOR 79], et DACAPO [BRU 85]. Dans le chapitre IV, nous allons discuter le simulateur fonctionnel FIDEL.

- Au niveau transfert de registres :
  L'architecture du système est détaillée en utilisant des primitives type registre, additioneur, et mémoire. Le rôle de la simulation est de vérifier le comportement du circuit à travers la décomposition structurelle en utilisant différents modules, et de vérifier aussi la synchronisation sur les connexions entre les différentes primitives. On peut citer comme exemples de ce type de simulateurs : DDL [DIE74], KARL [HAR 79] ,CDL [CHU 74] et CASSANDRE [MER 73].

- Au niveau logique :
  Dans la partie suivante, nous allons discuter en détail ce type de simulation. On peut citer comme exemples de ce type : HILO [FLA81], TEGAS [SZY 72] et EPILOG [CHI 74].

- Au niveau interrupteur :
  Dans le chapitre IV nous allons discuter le rôle et les concepts de ce type de simulation. Des exemples de ce type de simulation sont MOSSIM [BRY 80], MURPHY [BAN 83] et MOSLAM [AMA 85].

- Au niveau électrique :
  La différence essentielle entre ce type de simulation et les autres types est que le temps et les valeurs portées par les connexions prennent leurs valeurs dans un sous ensemble de R (simulation de type continu) et non dans un ensemble discret de valeurs (simulation de type discret). La discussion détaillée de ce type sera présentée dans la troisième partie de ce chapitre. On peut citer comme exemples de ce type : SPICE [NAG 75], RELAX [LEL 82] et ELDO [HEN85].

La majorité des simulateurs qui sont utilisés pendant le processus de la conception mêlent différents types de simulation, ils sont appelés simulateurs de type multi niveau (multi level simulators), on les présentera dans la partie prochaine. En plus, il existe la possibilité d'intégrer les deux types de simulation : discret et continu dans un seul simulateur, appelé simulateur de type mode mixte (mixed-mode simulator) ou hybride. Ce type sera discuté en détail dans la troisième partie de ce chapitre.

1.2 Les différents traitement du temps :

On peut classer la simulation en deux types selon la représentation du temps : continue ou discrète. Le premier type est utilisé seulement au niveau électrique, tandis que le second est utilisé aux autres niveaux.

Le terme "continue" pour le premier type provient du fait que l'on manipule des valeurs de courant et de potentiel supposées être des fonctions continues du temps (dans le domaine **R**).

Dans le type discret, les valeurs de signaux sont discrétisées et supposées être des fonctions discrétes du temps (dans le domaine N). On va présenter ce type de temps en discutant deux aspects : les différents modèles de retard, et les différents algorithmes de temps.

Le retard peut être spécifié soit comme une caractéristique prédéfinie pour chaque élément ou groupe d'éléments, soit sous forme explicite comme valeur d'entrée à cet élément ou à ce groupe d'éléments.

Les différents modèles de retard sont [SZY 75], [MUR 86] :

- **Modèle de retard nul :**
  Ce modèle est utilisé dans les premier simulateurs où on suppose que tous les éléments dans le circuit n'ont pas de retard. En conséquence, le simulateur avec ce modèle ne peut pas simuler les circuits asynchrones.

- **Modèle de retard unitaire :**
  Ce modèle associe la même unité de temps (une seule unité) à tous les éléments dans le circuit, donc il permet la simulation de circuits asynchrones. Ce modèle peut être une bonne approximation, spécialement pour les technologies avec des caractéristiques de retard uniforme.

- **Modèle de retard nominal :**
  Ce modèle associe un retard individuel à chaque type d'élément ou à chaque élément. La majorité des simulateurs logiques adoptent ce modèle

de retard.

- Modèle de retard min-max :

Ce modèle est développé pour réaliser une analyse temporelle plus précise, où le temps de retard est défini par une paire de valeurs : le temps minimal et maximal pour un changement donné. Bien que la plupart des erreurs temporelles puissent être détectées avec ce modèle, il a de sérieux inconvénients ainsi :

1. le temps de la simulation est très long.
2. les résultats de la simulation sont très pessimistes.
3. La simulation de circuits type latch ou différentiels peut être exécutée de manière incorrecte, comme le montrent les figures (17),(18) [BEN 79].

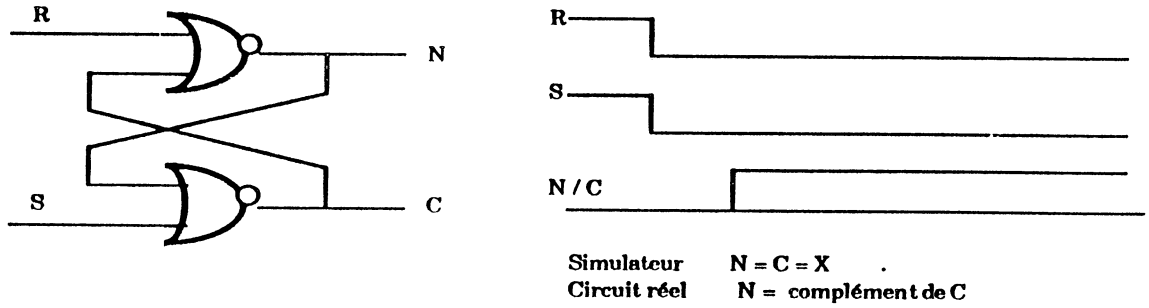


FIG . 17 : MODELE MIN_MAX DE RETARD

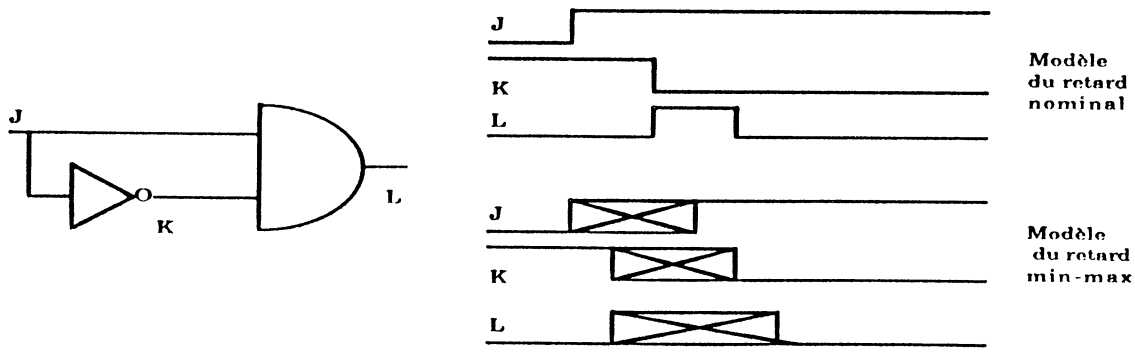


FIG . 18 : CIRCUIT DIFFERENTIEL ET MODELE DU RETARD

Ce modèle ne peut donc être utilisé que pour la simulation précise de petits circuits.

- Modèle de retard inertiel :
  Les modèles de retard présentés au-dessus, permettent la propagation d'impulsions quelque soit leurs largeurs ce qui n'est pas toujours vrai. Le modèle de retard inertiel permet la définition d'une largeur minimale d'impulsions que chaque élément peut propager (cf. figure 10, dans le chapitre I).

- Modèle distinguant le temps de montée et le temps de descente :
  Dans certaines technologies les temps de passage de 0 à 1 et de 1 à 0 sont différents. Le modèle permet alors de spécifier séparément les deux temps.

Il faut noter que certains simulateurs, comme le simulateur HILO [FLA81], peuvent inclure les différents modèles de retard dans un même environnement.

L'algorithme de temps peut être classé soit comme synchrone ou comme asynchrone selon que le changement de valeurs est lié ou non avec une horloge [SCH 72].

Dans la simulation synchrone, chaque opération a lieu sous le contrôle d'une horloge, tandis que dans la simulation asynchrone la valeur d'un signal peut être utilisée aussitôt que le changement d'état est fait.

Cette définition n'est pas liée au type du circuit (synchrone ou asynchrone). En effet les termes synchrone et asynchrone se rapportent à la manière de modéliser les représentations et les réponses du temps.

Donc un simulateur asynchrone peut être utilisé pour les circuits logiques type synchrone (l'inverse n'étant pas vrai). La simulation asynchrone est très utilisée pour la détection de certaines conditions temporelles comme "glitche" et "aléas".

On pourrait en déduire que seul le programme de simulation asynchrone doit être réalisé, puis que les deux types de circuits synchrones et asynchrones peuvent être analysés par ce programme. En réalité, la simulation synchrone est adoptée pour la plupart des simulateurs pour les raisons suivantes : la majorité des circuits opèrent en mode synchrone , la détection des "glitche" et "aléas" peut être faite en

utilisant les conditions de pire cas. L'utilisation d'un simulateur synchrone peut être plus simple et moins cher du fait que les événements sont contrôlés par l'horloge et que la simulation synchrone consomme moins de temps d'exécution que la simulation asynchrone.

1.3 Les différentes valeurs des signaux :

Les différentes valeurs des signaux dépendent des primitives de base associées à chaque simulateur. Par exemple au niveau électrique où on utilise des transistors, des capacités et des sources de courant et de tension, les valeurs sont exprimées en unités réelles .

A partir du niveau d'interrupteur jusqu'au niveau le plus haut dans la hiérarchie, les signaux sont représentés en utilisant des valeurs discrètes (il faut noter qu'au niveau algorithmique on peut utiliser des types entier, réel et énuméré).

La représentation des valeurs au niveau bas est divisée en deux types : type général et type spécial.

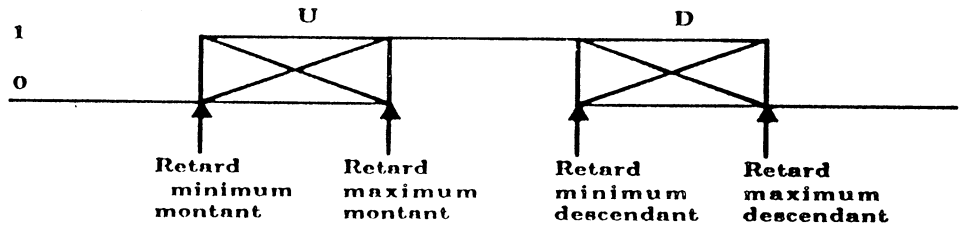


FIG . 19 : REPRESENTATION DE VALEURS < U > ET < D >

Le type général représente les valeurs d'états stables comme "0", "1", et "Z" (haute impédance pour le circuit à trois états).

Le deuxième type inclut les valeurs : "X" (inconnue), "U" (transition de 0 à 1), "D" (transition de 1 à 0) et "T" (transition entre 0,1 et la haute impedance Z). La valeur "X" est introduite pour bien manipuler les circuits séquentiels, les valeurs "D","U" et "T" sont introduites pour exprimer le modèle min-max de retard, la figure (19) montre la position de "D" et "U" dans la transition d'un signal.

Chaque simulateur selon ses objectifs utilise un ensemble de valeurs de signal. La plupart des simulateurs utilisent un ensemble de trois valeurs (0,1,X) ou quatre valeurs (0,1,X,Z).

Pour bien représenter les transistors MOS et prendre en compte leur fonctionnement bidirectionnel au niveau logique, le concept de la force d'un signal est utilisé [FLA 83]. En général, selon le nombre de forces, le nombre d'états est augmenté. Par exemple dans un système de trois valeurs (0,1,X) et trois forces "D" (driving), "R" (resistive) et "Z" (high-impedance), on a 9 états différents. Le simulateur SMILE [GON84] a 16 états, le simulateur THEMIS [DOS84] a 10 états, le simulateur TEGAS [THO75] a 5 états, le système CADOC [AMB83] a 7 états et HILO [FLA81] en a 6.

1.4 Algorithmes de simulation :

Il existe plusieurs techniques pour réaliser un algorithme de simulation. La plupart de ces techniques sont basées sur deux formes principales : dirigée par la compilation (compiler-driven) ou dirigée par la tabulation (table-driven) [BRE 75].

Technique de compilation :

La technique de compilation est la première méthode utilisée pour simuler un système logique [SES 62]. L'objectif d'un simulateur compilable est de générer à partir d'une description source (par exemple au niveau logique ou au niveau transfert de registre) une description dans un langage compilable et exécutable par un ordinateur. Si le compilateur génère un langage de bas niveau (assembleur) ou un langage de haut niveau type FORTRAN, le problème de base est toujours le même. Le traducteur doit restreindre les spécifications du temps et du comportement (synchrone ou concurrente) aux possibilités du langage.

En général, dans cette technique on ordonne les éléments du circuit (au niveau logique : les portes logiques, au niveau transfert de registres : les instructions) dans différentes couches selon la propagation des signaux par rapport aux entrées. Donc, l'exécution des différents éléments est passée d'une couche à l'autre jusqu' à ce que l'on génère la valeur de sortie, comme le montre la figure (20).

Comme résultat de cet arrangement, la mise à jour du circuit nécessite d'effectuer à nouveau la traduction et la compilation; la présentation du circuit comme une seule unité est préférable (pour traiter les différentes couches de manière collective). En plus il faut traiter les sorties bouclées (feed back outputs) de manière spéciale [SZY70], c'est à dire qu'il faut sélectionner les lignes bouclées et les couper en certains points dans le circuit. Si plusieurs lignes bouclées changent leurs valeurs au même instant, on en traite une à la fois (c'est à dire que sa valeur est propagée à travers le circuit pour déterminer son effet sur la sortie). Cette manipulation bien sûr demande plus de temps et de mémorisation.

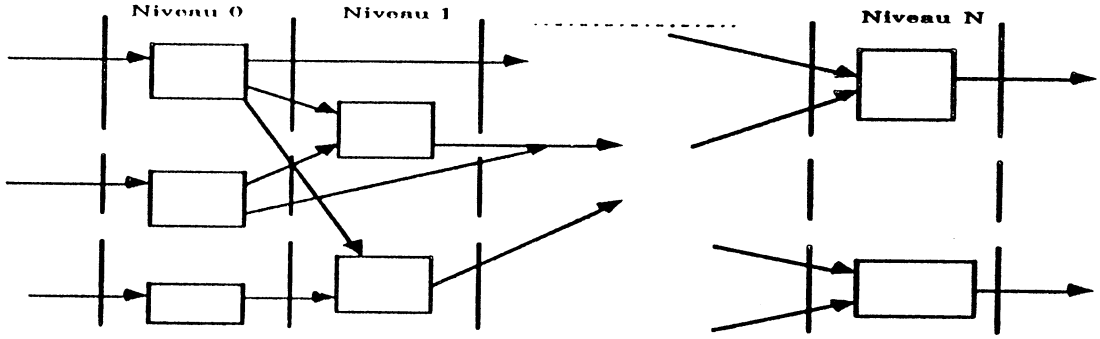


FIG. 20 : DIFFERENTS NIVEAUX (COUCHES) DANS LA TECHNIQUE DE COMPILATION

Pour les circuits synchrones complexes avec un pourcentage de changement d'éléments très élevé, cette technique de simulation du code compilé (levilized compiled-code ou LCC) est préférable, spécialement si l'on combine avec un accélérateur câblé [MIK86]. Des exemples de ce type sont : CASSANDRE [MER 73], et LASCAR [LEF 85]. Mais il faut noter que, aujourd'hui il y a beaucoup d'efforts concernant la simulation dirigée par la compilation [WAN 87], et [BRY 87].

Technique de tabulation :

Dans la deuxième technique de simulation "table-driven", on génère une structure de données (des tables ou listes) à la place du code exécutable comme dans la première méthode. En principe, la description du circuit est faite par la définition des éléments (les entrées primaires, les primitives de base ...). Pendant l'exécution le simulateur détermine l'élément à évaluer et utilise un programme général pour évaluer tous les éléments de n'importe quel type.

Cette technique est plus générale et peut manipuler les différents circuits séquentiels (synchrone et asynchrone) de manière plus efficace que la première technique. En plus, l'approche "trace sélective" [ULR 65] est utilisée pour améliorer sa performance. La plupart des simulateurs modernes utilisent cette technique, comme TEGAS [THO 80], HILO [FLA 81], et HELIX [SIL 83].

En conclusion, on peut dire que la technique de la simulation dirigée par événement offre la possibilité de simuler les circuits de types synchrones ou asynchrones avec des modèles complexes de temps. Cette technique est efficace dans le cas où le nombre d'éléments qui sont changés est inférieur ou égal à 20% [NEW 84] par rapport au changement des entrées.

Autres techniques :

En plus de ces deux techniques de base, il existe d'autres techniques de simulation que l'on présente en résumé.

- Technique de simulation parallèle :
  Normalement, dans le programme de simulation, un mot de machine hôte est utilisé pour une valeur de signal. Seulement un ensemble d'entrées et une configuration du circuit peuvent être simulés à chaque exécution. Si chaque bit dans le mot peut représenter une valeur d'entrée, on peut stocker plusieurs valeurs pour un même signal dans le mot et les traiter simultanément. Cette technique est utilisée dans la simulation de défauts [SES 62] et peut être applicable avec la technique de compilation [MIK 86] et la technique de tabulation comme TEGAS [SZY 70].

- Simulation par interprétation : DDLSIM [DUL 69]
  L'interprétation est un concept fréquent dans les langages de programmation. L'interpréteur stocke les entrées dans sa forme symbolique et exécute chaque instruction par un programme d'interprètation. Cette technique utilise des mécanismes de structures et de contrôle qui sont similaire à la technique de tabulation. En général, cette technique est applicable avec des circuits séquentiels.

- Simulation dirigée par les requêtes : [SMI 87]
  C'est une nouvelle approche pour simuler les circuits logiques. Le concept de base de cette technique est de propager les requêtes des sorties vers les entrées (de manière récursive) pour chaque sortie jusqu' à obtenir des valeurs stables (entrées primaires ou des valeurs intermédiaires) dans la fenêtre de temps. Cette technique est l'inverse de ce que l'on fait dans la technique "dirigée par événements". Il y a une très forte analogie entre cette technique et le D-algorithme de génération des vecteurs de test. Cette technique est rapide (on ne calcule que pour les sorties qui intéressent les concepteurs) et efficace, elle est applicable aux circuits séquentiels et combinatoires.

  En général, la combinaison des différentes caractéristiques qui ont été présentées au-dessus, génère les différents ensembles de propriétés associées à chaque simulateur. Le choix entre les différents types de simulateurs est fait selon les applications.

2. INTEGRATION DU MODELE FONCTIONNEL DANS LE SIMULATEUR LOGIQUE EPILOG

En général le simulateur logique est l'outil le plus utilisé pendant la phase de la vérification du système logique et de la génération de séquences du test. Cet outil est utilisé de façon extensive pendant la conception de différents systèmes logiques.

2.1 Rôle et concept du simulateur logique :

Les rôles du simulateur logique sont de simuler le comportement logique d'un circuit pour que le concepteur puisse vérifier si les fonctions logiques aussi bien que les performances temporelles du circuit sont satisfaisantes et aussi de vérifier l'existence d'erreurs temporelles. Utilisé pour l'analyse de défauts, le simulateur logique joue un rôle central en développant les différents ensembles de vecteurs du test.

En général le concept d'un simulateur logique peut être illustré comme suit :

- En utilisant un modèle du circuit on peut déterminer l'évolution dans le temps de ses signaux de sorties pour une séquence donnée de signaux d'entrées.

- Le modèle du circuit est décrit en terme d'éléments et des connections entre eux. Les éléments du circuit peuvent être des portes de base (NOT, AND, et OR), ou des portes complexes (LATCHES ou FLIP-FLOPS). Chaque élément a un type qui identifie sa fonction (son comportement).

- Le simulateur maintient la configuration de chaque élément, c'est à dire les valeurs de ses entrées, sorties, et l'état des variables internes (si elles existent). Un changement de la valeur d'une entrée, sortie ou d'une variable interne est défini comme un événement. Un événement se produit à un certain temps de simulation. Les commandes de simulation (stimulus) sont représentées sous la forme de séquences d'événements dont le temps est prédéfini.

- Le simulateur propage les événements globaux (changement d'une entrée ou sortie) entre les différents éléments du circuit. Un changement d'une variable interne d'un élément est considéré comme un événement local, donc est propagé seulement à l'intérieur de cet élément. Quand une entrée ou une variable interne d'un élément change, le simulateur détermine sa nouvelle sortie et son nouvel état interne. On appelle ceci une évaluation.

- L'évaluation d'un élément peut générer des événements globaux ou locaux. D'habitude, les retards sont associés aux opérations d'éléments et les événements générés doivent être classés selon leur heure d'activation. Les événements classés sont maintenus dans une liste d'événement (échéancier).

- Le mécanisme de gestion du temps de simulation manipule les événements tel qu'ils arrivent dans un ordre correct. Le processus de simulation continue tant qu'il y a des activités logiques dans le circuit, c'est à dire tant que la liste d'événements n'est pas vide.

- Quand il n'y a plus d'événement à traiter au temps courant de la simulation, ce temps est avancé à l'heure du prochain événement.

Ce concept de simulation logique s'appelle simulation dirigée par événement (event-driven simulation).

2.2 Le simulateur logique EPILOG :

EPILOG [EFC 86] est un simulateur logique défini et réalisé à THOMSON-EFCIS et utilisé au CNET. De façon à réaliser d'une manière efficace la validation de la conception logique et la génération des séquences de test, le système EPILOG possède les caractéristiques suivantes :

- EPILOG est un simulateur à cinq états : 0, 1,Ø (transition montante ou descendante), X (indéterminé) et Z (haute impédance). Cet ensemble d'états "logiques" permet la simulation fiable du comportement des circuits digitaux actuels et en particulier, des circuits intégrés en technologie MOS.

- Le simulateur comprend les modèles des éléments logiques les plus courants et offre la possibilité de création de modèles pour des fonctions logiques ou analogique plus complexes. Parmi les éléments de base, on trouve les ROM, les RAM, les PLA,...etc.

- Les temps de montée, descente et transition attribués à chaque élément sont définis par l'utilisateur qui leur attribue la valeur qu'il désire. Cette valeur peut être variable en fonction de l'état logique de l'élément lui-même et même de son contexte.

- EPILOG est piloté par événement (event-driven) avec la technique de "trace sélective".

Par la suite on présente des exemples en montrant les différentes possibilités de description en EPILOG.

EXEMPLE 1 DEMI-ADDITIONNEUR

```
**DESCRIPTION
*TOPOLOGIE
DEMIADD(A,B,RE-C,RS)
OUEX(A,B-AB)
OUEX(RE,AB-C)
ETNON(A,RE-RS3)
ETNON(A,B-RS2)
ETNON(RE,B-RS1)
ETNON(RS1,RS2,RS3-RS)
FIN
```

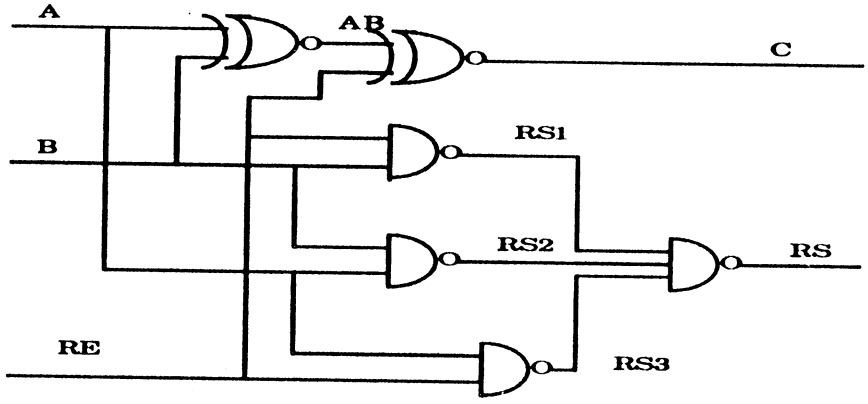


FIG. 21 : SCHEMA LOGIQUE D' UN DEMI-ADDITIONNEUR

Le schéma logique de cet exemple (figure (21)) peut être décrit de manière différente, comme le montre la figure (22) et l'exemple suivant.

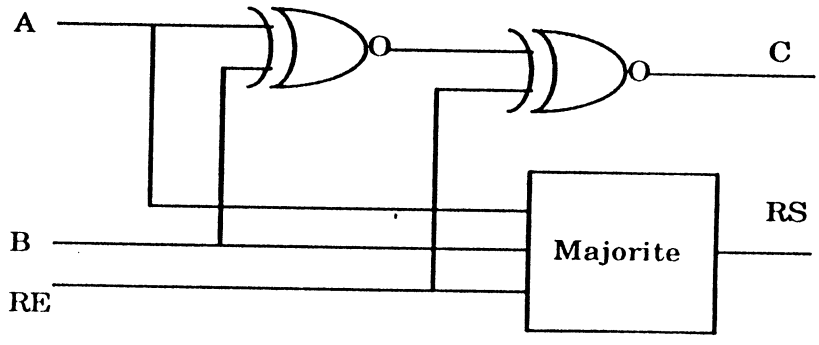


FIG. 22 : SCHEMA LOGIQUE D' UN DEMI-ADDITIONNEUR

EXEMPLE 2

```
**DESCRIPTION
*TOPOLOGIE
MAJORITE(E,F,G-S)
ETNON(E,F-EF)
ETNON(F,G-FG)
ETNON(E,G-EG)
ETNON(EF,FG,EG-S)
FIN
*TOPOLOGIE
DEMIADD(A,B,RE-C,RS)
OUEX(A,B-AB)
OUEX(AB,RE-C)
ET1-MAJORITE (A,B,RE-RS)  $ instantiation du bloc.
FIN
```

Cette description illustre la possibilité d'appel de bloc. C'est à dire on a commencé par la description du bloc "MAJORITE" en fonction des éléments de base (la première instruction **TOPOLOGIE). La construction du circuit (la deuxième instruction **TOPOLOGIE) s'est faite en utilisant deux primitives et une instantiation du bloc "MAJORITE".

Cette façon de décrire permet de structurer la description d'un circuit complet, en le découpant en fonctions que l'on peut simuler séparément puis assembler. Toutefois, de manière interne au simulateur, tout circuit est stocké de manière éclatée (mis à plat).

En plus, EPILOG permet à des personnes connaissant bien le logiciel d'introduire des modèles décrits sous forme d'algorithme (sous-programme FORTRAN). Les modèles disponibles sont les suivants : élément décrit par une équation ; élément décrit par une table d' entrées/sorties; mémoire RAM ; opérations fonctionnelles de n bits et modèle analogique à E/S logique. Nous avons utilisé cette propriété pour introduire le modèle FIDEL dans le simulateur.

Pour mieux connaître les caractéristiques de EPILOG on peut consulter [EFC 86]. Enfin EPILOG offre 4 types de simulation : simulation LOGIQUE (simulation 4 valeurs 0,1,X,Z avec retards nominaux de montée et descente), simulation FINE (simulation 5 valeurs 0,1,ø,X,Z avec retards minimaux et maximaux de montée et de descente), simulation de défauts type DEDUCTIVE et simulation de défauts type PARALLELE.

2.3 Problèmes du simulateur logique :

Bien que le simulateur logique soit un des outils les plus utilisé pendant le processus de la conception, certains problèmes limitent son utilisation dans certains cas :

- La diversité des technologies utilisées pour les circuits logiques nécessite l'augmentation du nombre des valeurs de signaux pour mieux exprimer l'état du signal dans la technologie MOS. La majorité des simulateurs logiques offrent cinq ou six valeurs pour mieux exprimer la technologie MOS, mais la manipulation interne ne donne pas des résultats exacts (vis à vis du niveau interrupteurs). On va discuter ce problème en détail dans le chapitre IV.

- Avec l'augmentation de complexité des circuits, la nécessité de vérification temporelle est augmentée et la simulation logique ne donne pas des résultats satisfaisants ; par conséquent des outils spéciaux sont demandés [MCW 80], [HIT 82] et [COE 84].

- Le niveau logique représente une phase intermédiaire dans le processus de la conception. Il faut pouvoir effectuer la vérification de conception aussitôt que possible aux niveaux les plus hauts (niveau de comportement ou fonctionnel).

- Enfin, le problème le plus important est l'augmentation du temps d'exécution et de la taille mémoire. Par la suite on va présenter ce problème en détail avec les différentes solutions utilisées.

Temps d'exécution et taille mémoire :

Le temps d'exécution d'un simulateur logique est une fonction du nombre d'événements à traiter, ce nombre est proportionnel à la taille du circuit à simuler. L'augmentation de la taille du circuit entraîne une croissance rapide du temps d'exécution pour la simulation.

C'est un des plus importants sujets d'étude dans le domaine de la conception assistée par ordinateur (CAO) pour les circuits VLSI que de développer les techniques de simulation à grande vitesse. On peut classer les différentes approches proposées et utilisées [MUR 86] et [ISH 87] par la suite :

- Introduction de modèle de plus haut niveau :
  Le modèle fonctionnel et la simulation fonctionnelle apportent une réduction importante du temps d'exécution et de taille de mémoire pour la simulation (cependant, en échange d'une perte de précision). Cette technique est présentée en détail plus bas.

- Amélioration des algorithmes de simulation :
  Cette amélioration se fait dans trois directions :

  (1) Techniques d'amélioration du logiciel : [ULR 72], [ULR 80] et [ISH85].
  (2) Utilisation des techniques de simulation de défauts : simulation parallèle [MIK 86] et simulation concurrente [ULR 82].
  (3) Utilisation de machines de simulation (hardware simulation engines). Ces techniques peuvent améliorer la performance de simulation par une facteur 1000 par rapport aux simulateurs type logiciels (software simulation). Les exemples de ce type sont [DEN82], [SAS 83] et [BLA 84].

Il faut noter que la simulation logicielle avec des performances élevées est plus flexible et portable que la simulation de type matériel. On peut combiner les différentes solutions citées ci-dessus dans une seule technique.

2.4 Simulation logico-fonctionnelle :

On a mentionné que le simulateur logique est l'outil le plus utilisé pour vérifier la conception détaillée d'un circuit. Cependant, il est difficile de simuler l'assemblage d'un circuit imprimé qui inclut des processeurs commerciaux existants puisque la structure logique interne de ces processeurs n'est généralement pas fournie. La simulation fonctionnelle [WIL 79], [CHA 76] peut être utilisée dans ce cas.

La simulation fonctionnelle signifie la description et l'exécution du fonctionnement d'un circuit soit comme un seul bloc (algorithmique), soit comme différents blocs fonctionnels interconnectés. Comme ce type de simulation manque de précision par rapport à la simulation logique, la simulation mixte qui permet de simuler des circuits incluant des portes logiques et des blocs fonctionnels est la solution. L'intégration du modèle fonctionnel FIDEL dans le simulateur logique EPILOG réalise cette approche [EL 84-a],[EL 84-b].

Il existe deux méthodes pour intégrer un modèle fonctionnel dans le simulateur logique [WIL 79]:

- Le modèle fonctionnel peut être implanté comme une primitive dans le système de simulation, comme une porte logique normale.

- Le modèle fonctionnel peut être décrit dans un langage fonctionnel (FIDEL) et compilé pour etre interprété par le simulateur.

La première méthode est facile à réaliser mais elle souffre de problèmes de maintenance et manque de généralité. Il est difficile de réaliser des primitives générales qui permettent de décrire tous les blocs souhaités et la technique ne permet pas d'exprimer de manière simple les détails souhaités par le concepteur.

La deuxième méthode peut souffrir des insuffisances du langage (selon les caractéristiques du langage fonctionnel utilisé), mais elle a l'avantage que le concepteur peut modéliser le circuit à n'importe quel niveau de détail dont il a besoin et de manière propre.

Toutefois les deux méthodes ne sont pas mutuellement exclusives, c'est à dire qu'on peut intégrer les deux méthodes dans le même système de simulation (c'est le cas de notre approche).

2.4.1 Interface du modèle fonctionnel avec le simulateur:

La base de notre approche est de retenir toutes les fonctionnalités du simulateur logique et de les étendre avec l'introduction de modèles fonctionnels complexes.

Un modèle fonctionnel est une boite noire définie par des entrées/sorties et une routine d'évaluation associée.

Pour définir et implémenter l'interface entre les modèles fonctionnels FIDEL et le simulateur logique EPILOG nous avons essayé de respecter les exigences suivantes :

- Le modèle fonctionnel et la porte logique doivent être compatibles et interchangeables.
- Le modèle fonctionnel doit prendre en compte les traitements temporels associés à la simulation logique par EPILOG .
- Le modèle fonctionnel doit permettre la manipulation de différents types et formes de données (bit, registre et scalaire).

Traitement de la valeur indéterminée (X) :

La compatibilité entre portes logiques et modèles fonctionnels nécessite la manipulation des mêmes formats de données de signaux. Le modèle fonctionnel doit donc traiter le même ensemble de valeurs que EPILOG (0,1,Ø,X et Z).

Les valeurs Z et Ø ne posant pas de problèmes particuliers , nous discuterons seulement des problèmes liés à la valeur X . Nous présentons deux techniques :

La technique de "buffer interface" :

La technique dite de "buffer interface" [BOS 77] et [ALI 78], utilise une forme de "buffer" pour intercepter toutes les entrées du modèle fonctionnel qui sont indéterminées ; les états indéterminés sont ensuite remplacés par toutes les combinaisons possibles de 0 et 1, et le modèle fonctionnel est évalué pour chaque combinaison. Si une sortie du modèle fonctionnel est différente pour deux évaluations, cette sortie est forcée à la valeur X. Le défaut lié à cette technique est le volume de calcul nécessaire. Par exemple, 4 états d' entrées qui sont à X, vont générer 16 évaluations du modèle.

La technique d'évaluation conditionnelle :

Dans la technique dite "évaluation conditionnelle" [WIL79] le compilateur de langage fonctionnel va générer un code qui traite de manière explicite la valeur X. Lors de l'évaluation du modèle un événement de sortie est considéré comme "conditionnel" si cet événement dépend d'une entrée ou d'un état qui est à X. Par exemple si on écrit

```
WHEN CLEAR BECOMES 1
        MAKE OUTPUT = 0;
```

La condition liée au "WHEN" peut être évaluée vraie, fausse ou indéterminée selon la valeur de "CLEAR". Si la valeur de "CLEAR" est égale à X, un événement conditionnel de valeur zéro est prédit pour la sortie "OUTPUT". Le simulateur va transformer l'événement conditionnel en un état à X, si la valeur courante de "OUTPUT" est différente de la valeur conditionnelle.

Cette manipulation spéciale est associée seulement à l'évaluation d'événements externes : au niveau interne on traite la valeur X de manière pessimiste, par exemple si l'on écrit :

```
MAKE OUTPUT = (A AND C ) OR ( B AND NOT C )
```

Si la valeur de C est à X le résultat sera à X. Cette technique présente le même

défaut que la précédente : grand volume de calculs pour manipuler les événements conditionnels, spécialement avec des modèles complexes comme les microprocesseurs.

Dans notre approche, on a adopté une manière simple pour la prise en considération de la valeur X.

Pendant l'exécution du modèle la propagation de la valeur X est faite de manière normale comme pour les valeurs 1 et 0 : on applique l'algèbre de Boole entre tous les valeurs.

Pour les opérations arithmétiques la valeur X est absorbante : le résultat d'une opération avec une valeur à X sera X.

Si la valeur d'une condition associée à un événement est à X on la considère par défaut comme étant la valeur fausse, donc on n' exécute pas les instructions associées à cet événement.

Si le concepteur veut tester la valeur X il doit l'indiquer de manière explicite, par exemple on peut écrire

```
WHEN CLEAR BECOMES #B X
  REPORT THAT ' CLEAR VAL IS X '
  MAKE OUTPUT = #B X ;
```

Le passage des valeurs entre le modèle fonctionnel et le simulateur s'effectue donc de manière simple. Les résultats dans certains cas, peuvent être pessimistes : c'est la responsabilité du concepteur de vérifier si la valeur X comme résultat représente une exécution normale ou anormale.

<u>Considérations temporelles :</u>

Les deux points essentiels du traitement du temps par le simulateur logico-fonctionnel sont :

- la transmission des dates du temps entre le simulateur logique et le modèle fonctionnel.
- la génération d'un échéancier interne au modèle fonctionnel.

On a mentionné dans le chapitre II que dans FIDEL on peut représenter les retards dans le modèle fonctionnel en utilisant l'instruction "**WITHIN**". La prise en compte de ces retards nécessite deux paramètres dans l'interface entre le modèle FIDEL et le simulateur EPILOG : l'heure actuelle de la simulation (le temps courant) et l'heure prochaine de simulation.

En général EPILOG est dirigé par événement, tous les événements sont stockés dans une liste d'événement ou un échéancier. La valeur de l'heure actuelle de simulation est déterminée par l'événement qui est en tête de cette liste. Si le modèle fonctionnel contient un élément de retard (par l'instruction **WITHIN**), il renvoie au simulateur logique la date de réveil (prochaine heure de simulation). S'il y a plusieurs actions qui sont associées à l'instruction **WITHIN**, ou s'il y a plusieurs **WITHIN** dans le modèle, on a besoin d'un échéancier interne au modèle pour stocker tous ces événements locaux. Cette liste d'événements locaux est associée à chaque modèle fonctionnel.

En plus pour prendre en compte les vérifications temporelles (temps de maintien, temps d' établissement et durée de changement), la valeur interne d'une variable du modèle fonctionnel est représentée par trois éléments : son état (0,1,X,Φ,Z), la date de dernière modification, le temps de stabilisation. L'utilisation de ces éléments donne une grande flexibilité à la modélisation et permet la génération des résultats exacts.

Représentation interne:

Le simulateur logique ne manipule qu'un type de données : le bit ; le modèle fonctionnel permet la manipulation de types de données plus riches (bit, registre et scalaire). L'interface entre le modèle fonctionnel et le simulateur logique effectue les transformations de format.

2.4.2 Structure générale :

Dans cette section, on va présenter la structure générale de notre interface et la structure de donnée interne nécessaire pendant la phase d'interprétation du modèle fonctionnel.

Comme le montre la figure (25), la structure générale est constituée par:

- le compilateur FIDEL qui génère la structure de données associée au modèle fonctionnel et la stocke dans la bibliothèque du simulateur (la bibliothèque spéciale de modèle)
- l'interpréteur qui exécute le code généré par le compilateur. Cet interpréteur est intégré dans le simulateur.
- l'environnement de simulation fourni par EPILOG.

Comme on l'a mentionné le compilateur vérifie la description FIDEL (analyse lexicale, syntaxique et sémantique) et génère le code interprétable associé au modèle dans un fichier intermédiaire (executed-coded-file ou ECF).

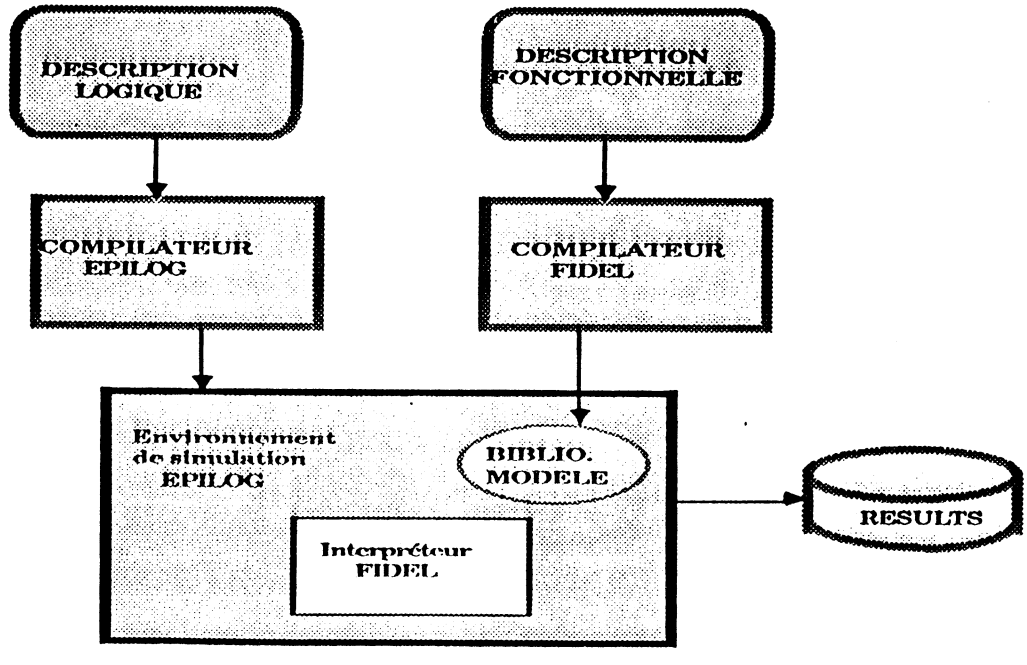


FIG. 23 : STRUCTURE GENERALE

Ce fichier qui est stocké de manière automatique dans la bibliothèque du simulateur logique , contient les informations internes du modèle codées dans la forme standard du simulateur EPILOG (instruction **MODELE du langage EPILOG).

Chaque modèle fonctionnel est stocké dans une zone spéciale (déterminée par un pointeur associé au modèle) dans la bibliothèque de modèles. Pendant la simulation, cette zone est copiée dans la table de description du circuit (DES). Chaque instantiation du modèle dans la description duplique la zone du modèle dans la table DES.

L'interface entre l'interpréteur FIDEL et le simulateur contient les informations suivantes : les valeurs de signaux d'entrées, la date courante de simulation, un pointeur dans la table DES, la table DES, la prochaine date de simulation et les valeurs des signaux de sorties.

L'interpréteur de FIDEL est écrit en FORTRAN et implémenté sur le système VAX/VMS . La structure interne de cet interpréteur utilise des listes indexées et l'allocation de mémoire dynamique pour présenter le fichier ECF. Cette structure de donnée est détaillée en [EL 84-b].

L'intégration des modèles fonctionnels FIDEL dans le simulateur logique EPILOG a été utilisée de manière extensive par les concepteurs du CNET et de THOMSON-EFCIS. La technique de modélisation utilisée dans notre approche offre la flexibilité désirée par le concepteur et soutient de manière directe le processus de conception. Le concepteur est capable de modéliser et simuler son circuit aux différents niveaux (logique, fonctionnel ou mixte).

La simulation logico-fonctionnelle s'est montrée un outil puissant et applicable à des circuits complexes comme : l'arbitre de bus ABC90 [OLI82], le microprocesseur TCOM et dans le domaine des processeurs de signal (UPTS [CAN85] et [BOU 87]).

Les gains en taille mémoire et en temps d'exécution sont particulièrement grands par rapport à une simulation purement logique (10 fois pour le cas le moins

favorable). Dans l'annexe B, on présente un exemple complet avec les résultats de simulation.

2.5 Proposition :

Dans cette section on propose un outil complet [BOU 85] utilisant :

- le GRAFCET [MOA 81] comme langage de description fonctionnelle.
- un éditeur graphique associé à ce formalisme.
- une interconnexion avec le simulateur logico-fonctionnel (FIDEL-EPILOG).

Description du GRAFCET :

Le formalisme du GRAFCET est dérivé du formalisme des réseaux de PETRI. Le GRAFCET est un graphe de contrôle avec un nombre de noeuds limité. Les éléments de base sont les places et les transitions.

Une place peut avoir deux états : active ou inactive. A une place est associé un ensemble d'actions que le système doit exécuter simultanément lorsqu'elles deviennent actives. Une action peut être une commande élémentaire ou une commande conditionnelle.

Les places sont reliées entre elles par les transitions. Les transitions représentent les différentes possibilités de passer d'un état stable à l'autre. Une réceptivité est associée à chaque transition, elle présente l'ensemble des conditions nécessaires pour déclencher la transition. Elle est constituée d'un événement tel qu'un changement de valeur d'une variable ou d'une condition logique liée aux valeurs internes, externes et aux états des places.

Règles d'interprétation :

Les places initiales du système, qui sont représentées par un double carré, sont activées de manière systématique quand le système est déclenché . Un jeton est utilisé pour marquer les places actives. La réceptivité associée à une transition est

vraie seulement si toutes les conditions ont été vérifiées et les places amont activées.

Le déclenchement d'une transition désactive les places amont et active les places aval en exécutant les actions associées à ces places. Le GRAFCET contrairement au réseau de PETRI est déterministe et donc convenable pour modéliser les systèmes réels.

L'éditeur graphique :

L'éditeur graphique permet d'entrer les trois types de description (GRAFCET, FIDEL et portes logiques) dans un environnement commun. Il est intégré au système CASSIOPEE [LEC82].

Cet éditeur permet au concepteur de définir son circuit comme un ensemble de boites noires qui peuvent elles-mêmes être divisées en partie contrôle et partie opérative. Par exemple la partie opérative peut être décrite en terme de portes logiques ou de blocs fonctionnels tandis que la partie contrôle peut être décrite en utilisant le GRAFCET ou le langage FIDEL. Toutes les informations sont stockées dans une base de données commune, la vérification de cohérence est faite de manière automatique.

Dans le cadre de notre travail, on a construit un logiciel qui permet la transformation automatique du GRAFCET en description fonctionnelle FIDEL [BOU 84], [FLA 84] .

L'outil que l'on a proposé facilite la tâche de description du concepteur en lui offrant toutes la flexibilité qu'il désire pour modéliser et simuler son circuit (un exemple complet est donné dans l'annexe C).

3. INTEGRATION DU MODELE FONCTIONNEL DANS LE SIMULATEUR ELECTRIQUE ELDO

La simulation électrique sert à prévoir le comportement physique (tenir compte de la technologie) et temporel précis d'un circuit : bon fonctionnement, vitesse, constantes de temps, niveaux de polarisation, consommation, réponse impulsionnelles des filtres, etc.

3.1 Rôle et concept de simulation électrique :

Du temps des composants discrets, la simulation électrique était utilisée dans la phase de pré-maquette pour prévoir la taille et le type des composants discrets à utiliser. On simulait des réseaux de plusieurs dizaines, voire une centaine de transistors. L'apparition des circuits intégrés de plus en plus denses a accru la demande en simulation électrique sur les deux aspects qualitatif et quantitatif.

Sur l'aspect qualitatif, la suppression de la phase de maquettage voit le travail éclaté en amont vers la simulation électrique et en aval vers le test des circuits finis. Comme la fabrication de ces circuits à densité d'intégration de plus en plus grande coûte de plus en plus cher, on a tendance à solliciter de plus en plus les simulateurs électriques avant la fabrication.

Sur l'aspect quantitatif, le nombre de transistors à simuler va de 1000 à 10 000 et la durée simulée de 1 à 10 microsecondes. Pour de telles tailles, les simulateurs électriques dits de deuxième génération, caractérisés par l'utilisation de la méthode de Newton, sont totalement inefficaces : dès que la taille dépasse plusieurs centaines de transistors, les temps de calcul atteignent plusieurs heures et les convergences deviennent de plus en plus hasardeuses.

La simulation électrique est un point clé de la conception assistée par ordinateur (CAO) de circuits VLSI, car c'est le seul lien entre le travail amont des concepteurs de circuits et le travail aval des technologues, modélistes et fondeurs. Elle fait la liaison entre les données technologiques (paramètres des modèles, modèles) et les données de circuit (réseau de transistors interconnectés)

afin de prédire le mieux possible le fonctionnement futur d'un circuit conçu dans une technologie donnée.

En général, l'analyse de circuits en utilisant la simulation électrique est basée sur les opérations suivantes :

- manipulation d'entrées.
- manipulation de modèles.
- formulation des équations.
- solution des équations et sortie des résultats.

Pour la manipulation d'entrées, le concepteur doit fournir la description de circuit comme un réseau et une liste d'éléments du réseau, incluant leurs paramètres et le type d'analyse à traiter.

La manipulation de modèles inclut l'accès à la bibliothèque des modèles du simulateur. Cette bibliothèque contient le comportement des dispositifs rencontrés dans les circuits (transistor MOS ou bipolaire, ligne RC,...). Les dispositifs sont paramètrés par des valeurs dépendant de la technologie. Le simulateur de manière générale permet au concepteur de définir et modifier les paramètres et d'utiliser les modèles.

La formulation d'équations consiste à construire la description mathématique du réseau électronique présenté au simulateur comme entrée. Ces équations peuvent être non linéaires et permettent l'analyse transitoire et fréquentielle, aussi bien que la réponse statique et dynamique. Les lois de Kirchoff de tension et courant, avec les paramètres des dispositifs permettent la construction des équations de circuit.

Une fois que les équations sont établies, elles sont résolues de manière numérique. Il y a des algorithmes pour développer les solutions numériques des différents types d'équations du circuit [BARR 85]. Par exemple, le réseau peut être décrit par un système d'équations algébriques linéaires simultanées, qui peut être résolu en utilisant la méthode de "Gaussian elimination".

Pour les équations algébriques non linéaires, la méthode de Newton-Raphson est la plus utilisée. Si le point de départ est suffisamment approché de la solution, et que les fonctions des modèles électriques sont dérivables, l'algorithme va toujours converger vers cette solution. Un mauvais point de départ peut amener cette méthode à diverger.

3.2 Problèmes de la simulation électrique :

La plupart de simulateurs électriques classiques (SPICE [NAG75], ASTP [WEE73]) utilisent pour résoudre les schémas implicites à plusieurs inconnues la méthode de Newton-Raphson, qui généralise la méthode de la tangente. Pour résoudre l'équation d'évolution en régime transitoire de signaux électriques :

$$dV/dt = F(V) \qquad (1)$$

On discrétise d'abord le temps, par exemple avec la méthode d'Euler rétrograde, on va chercher à résoudre successivement le schéma :

$$V_n = V_{n-1} + \text{pas}\ F(V_n) \qquad (2)$$

le pas étant la valeur $t_n - t_{n-1}$, la figure (24) représente l'algorithme de Newton.

```
POUR n = 1,2,...,N          (temps discétisé)
 DEBUT
  POUR i = 1,2,.....,       (numéro d'itération de Newton)
   DEBUT
                                  i
     Résoudre en le vecteur Vn des tensions sur les nœuds du circuit :

                  i-1       i-1   i        i-1                  i-1
     (I - pas * F( Vn )) . (Vn  - Vn ) = Vn -  Vn-1 -pas * F ( Vn )

      i       i      i-1
     En  = II  Vn -  Vn  II
   FIN
   Calculer le vecteur de prédicition explicite :
    0       i              i
   Vn+1 =   Vn + pas *  F( Vn )
FIN
```

FIG. 24 METHODE DE NEWTON

Les inconvénients de cette méthode sont la nécessité de calculer la matrice jacobienne des dérivées partielles de la fonction F, puis de résoudre un système linéaire en inversant cette matrice. Cela provoque une croissance du temps de calcul proportionnelle au carré du nombre de noeuds de circuit, et pénalise grandement ces simulateurs classiques dès que les circuits dépassent quelques centaines de transistors.

Cette situation a provoqué des recherches dans le but de réduire le coût de simulation électrique (taille de mémoire et temps de calcul). Ces recherches attaquent tous les aspects du problème de la simulation électrique , à partir de modèles mathématiques utilisés pour représenter les dispositifs intégrés jusqu' aux algorithmes pour résoudre l'intégration numérique et les équations non linéaires.

En général on peut classer les différentes recherches en deux catégories : des techniques liées à l'amélioration de la méthode de Newton, de nouvelles techniques en changeant les méthodes de base avec des nouvelles propriétés concernent la convergence et la stabilité ; ces dernières sont introduites dans les simulateurs électriques dits de troisième génération [HAC 81].

3.2.1 Techniques d'amélioration :

Une technique pour améliorer le temps de calcul du programme d'analyse de circuits est basée sur le calcul vectoriel [VLA 82]. Des sous-circuits répétitifs peuvent être analysés en parallèle ce qui gagne du temps. Pour évaluer le gain de temps de cette technique, une comparaison est faite entre SPICE2 [NAG 75] et CLASSEIE [VLA 82] un programme d'analyse de circuits de ce type. Pour un circuit suffisamment grand et régulier comme l'additionneur, le rapport est un ordre de grandeur par rapport à SPICE2.

Pour la majorité des circuits, la proportion des noeuds dont la tension varie à un instant donné diminue avec l'augmentation de la taille du circuit. Pour un circuit qui contient plus de 500 transistors, moins de 20% de noeuds changent leurs valeurs de manière significative dans un pas de simulation [NEW 84] ; on appelle ceci l'inactivité du circuit ou "latency". Les simulateurs électriques

exploitent cette inactivité en évitant de calculer les noeuds inactifs [NAG 75] ou les blocs de noeuds inactifs [SAK 80]. Si la tension d'un noeud ou le courant d'une branche d'un élément de circuit n'ont pas changés de manière significative au pas de calcul précédent, leurs contributions dans l'équation ne sont pas ré-évaluées, et leurs valeurs précédentes sont utilisées. Le même principe est utilisé au niveau du bloc: si tous les éléments qui attaquent ce bloc ne changent pas, le bloc est considéré inactif.

En général, toute ces techniques améliorent le temps de calcul (dans le meilleur cas d'un ordre de grandeur) en conservant la même précision que les simulateurs classiques.

3.2.2 Techniques de relaxation :

Les caractéristiques principales des techniques de relaxation par rapport aux autres techniques sont qu'elles n'ont pas besoin d'utiliser les méthodes directes pour résoudre les équations de grands systèmes électriques ; elles permettent aussi d'exploiter l'inactivité du circuit de manière efficace.

La technique de découplage a été introduite dans le type de simulation dite "timing", des exemples de ce type sont MOTIS [CHA 75] et MOTIS-C [FAN 77]. En principe dans ce type de simulation seule une itération de relaxation est faite à chaque pas de simulation tandis que une ou plusieurs itérations de Newton-Raphson peuvent être faites pour résoudre l'équation de noeud. Des comparaisons et critiques de ces systèmes peuvent être trouvées dans [HAC 81].

L'algorithme ITA (Iterative Timing Analysis) [KEL 82] est une nouvelle forme d'analyse temporelle . A chaque pas du temps, chaque noeud est calculé de manière séquentielle en utilisant une itération de Gausse-Seidel, et au contraire de la simulation "timing", cette procédure est répétée jusqu'à convergence. En principe, dans cette approche seule une itération de Newton-Raphson est utilisée pour résoudre l'équation de chaque noeud et les techniques "dirigé par événements et trace sélective" peuvent être utilisées pour exploiter l'inactivité du circuit.

L'approche ITA est intégrée dans le simulateur de mode mixte SPLICE [NEW81]; on peut trouver une comparaison entre ITA et OSR (One Step Relaxation), [HEN 85] la technique utilisé dans le simulateur ELDO [HEN 85],qui sera présentée dans la section suivante.

Un autre type de cette famille est le simulateur SAMSON [SAK 85], qui est un simulateur électrique dirigé par événements, pour exploiter l'inactivité du circuit pendant la simulation. Il est plus rapide que SPICE d'un ordre de grandeur avec une bonne précision. Cette performance est réalisée par la partition du circuit en sous circuits qui peuvent avoir des pas d'intégration différents selon leur activité. La précision de la solution est maintenue en contrôlant l'erreur qui résulte du découplage entre les sous circuits et de la troncature locale dans chaque sous circuit.

Enfin, pour terminer cette discussion des techniques de relaxation, il faut noter que l'utilisation des relaxations est apparue en 1982 avec le simulateur RELAX [LEL 82], qui utilisait un algorithme nommé "Waveform Relaxation Method" ou WRM [LEL 82] ; on va présenter une comparaison entre cette méthode et OSR dans la section suivante.

3.3 Le simulateur électrique ELDO

ELDO est un nouveau simulateur électrique rapide développé au CNET de Grenoble. Il permet la simulation de circuits analogiques, logiques et mixtes de différentes technologies. Les caractéristiques de ELDO sont sa rapidité et sa convergence dues à son algorithme à relaxation OSR (One Step Relaxation) que l'on présentera par la suite.

Une itération de relaxation de l'algorithme OSR consiste à résoudre successivement pour tous les noeuds ou blocs préalablement ordonnés, un schéma implicite à une inconnue, en prenant pour les noeuds environnants une valeur judicieusement choisie. Si le noeud environnant a déjà été calculé, on prend la valeur obtenue, sinon ce choix dépend du numéro d'itération de relaxation : à la première, on prend un prédicteur explicite et à toutes les suivantes, on prend le résultat de la relaxation précédente.

L'algorithme de OSR est présenté dans la figure (25), OSR a résolu le même schéma implicite global que l'algorithme Newton-Raphson dans les simulateurs classiques. L'algorithme OSR peut être utilisé à la place de celui de Newton-Raphson.

```
POUR n = 1,2,...,N            (temps dicrétisé)
 DEBUT
  POUR i = 1,2,.....          (numéro d'itération de relaxation)
   DEBUT
    POUR j = 1,2,....,p       (nœuds ou bloc de nœuds)
    DEBUT
                  j i
      Résoudre en Vn  le schéma implicite :

      j i      j i              j   1 i  2 i      j i j+1 i-1    p i-1
       Vn  =    Vn-1  + pas *  F( Vn , Vn  ,..., Vn ,  Vn  ,..., Vn )

      j i      j i    j i-1
       En  = I Vn  -  Vn  I

    FIN

     i         j i
    Dn =  Max  En
           j
  FIN
  Calculer le prédicteur explicite :

 j 0     j i          j
  Vn+1 =  Vn  + pas *  F

FIN
```

FIG. 25 : OSR ALGORITHME

La différence entre ELDO et RELAX est présentée en [HEN 86]. On remarque que les deux équations centrales des algorithmes OSR et WRM sont exactement les mêmes et que les deux méthodes ne différent que par l'ordre d'imbrication de trois boucles : le temps, les relaxations et les noeuds pour l'algorithme OSR; les relaxations, les noeuds et le temps pour l'algorithme WRM. Cela explique les analogies et les performances voisines de deux méthodes.

Mais l'avantage de l'algorithme OSR sur WRM est de ne plus mémoriser une simulation entière, car, à chaque pas de calcul, le vecteur d'état est suffisant pour la suite des calculs : il mémorise tout le passé de la simulation.

De plus, si un décalage apparaît à certain instant, il est corrigé tout de suite dans l'algorithme OSR, alors qu'il est traîné jusqu'à la fin d'une simulation dans l'algorithme WRM qui continue des calculs précis sur des données fausses. De plus, aux instants qui précèdent ce décalage, l'algorithme WRM recalcule inutilement le début inchangé de la simulation. Il est vrai que l'algorithme WRM ne recalcule pas les noeuds dont la forme d'onde ne bouge plus, mais bien que ce critère soit sévère (égalité de deux courbes à tous les instants), il n'est pas suffisant, pour garantir que ces noeuds ne doivent plus être recalculés à cause d'un changement de leur environnement.

Une comparaison a été faite entre SAMSON [SAK 85] et RELAX sur un oscillateur en anneau et le résultat a montré que RELAX est plus lent que SAMSON sur cet exemple. L'explication citée en [SAK 85] est que la méthode de WRM est "non causale", c'est à dire que dans cette méthode on utilise des valeurs futures de certains points pour calculer les valeurs présentes d'autres points. L'amélioration proposée en [WHI 84] pour accélérer la convergence est d'effectuer la relaxation partielle en divisant l'intervalle de simulation en fenêtres. A la limite, si on prend la taille de cette fenêtre égale à zéro, on retrouve la technique d' OSR.

Une comparaison pratique est faite entre les techniques ITA et OSR, le résultat de cette comparaison [EL 87] et [HEN 87] est présentée dans la figure (26). Dans le cas d'amplificateur opérationnel CMOS, OSR est légèrement plus lent que ITA, mais dans le cas d' un circuit numérique classique (AD4B : 4 bascules de type D, et 3 additioneurs à un bit) OSR est légèrement plus rapide que ITA. Pour le circuit de "ring-oscillator" avec un pas variable, ITA est légèrement plus rapide , mais avec un pas fixe (0.3 nanoseconde) il diverge. Le dernière exemple est un filtre à capacité commuté dans lequelle ITA a complètement divergé.

Le gain de temps CPU pour ITA est dû à l'économie de certains calculs dans la boucle interne. Mais cela entraîne une augmentation du nombre global des relaxations et dans certains cas une augmentation du temps total de simulation. Mais la critique majeure de la méthode ITA par rapport à OSR est que ITA est moins fiable, l'économie sur la boucle interne peut faire diverger la boucle externe. Par contre il n'existe pas de circuits où OSR diverge si ITA converge, la réciproque étant fausse.

	ITA			OSR		
	mean nb of relaxation	mean nb of call	call nb of MOS model(1000)	mean nb of relaxation	mean nb of call	call nb of MOS model(1000)
AD4B	3.2	1.6	81	2.3	2.1	68
OP-AMP	8.42	1.87	93	8.46	3.25	75
R OSC(V.S)	2.7	1.6	8	2.4	2.28	10
R OSC(F.S)	DIV	DIV	DIV	2.87	2.13	8
Switch-cap	DIV	DIV	DIV	3.3	2.78	132

FIG. 26 : COMPARAISON OSR - ITA

Enfin, ELDO représente le même spectre d'applications que SPICE : analyse en régime continu, en transitoire et en petit signal des circuits MOS, bipolaires, GaAs,etc.

3.4 Interface FIDEL-ELDO

On a mentionné dans la première partie de ce chapitre que les différentes formes de simulation peuvent être classées soit en fonction du niveau de description (fonctionnel, logique, interrupteur et électrique) soit en fonction de mode de manipulation du temps (discrète ou continue).

Il est évident que la plus grande partie d'un circuit VLSI est numérique, mais les problèmes critiques de cette partie peuvent être considérés comme des problèmes analogiques. Donc, pour simuler la partie digitale qui est critique aussi bien que la partie analogique où les niveaux de tension sont critiques et où il y a des bouclages à fort gain, les simulateurs classiques comme SPICE et ASTAP peuvent être utilisés de manière précise. Comme on l'a présenté dans la section précédente ce type de simulateur (mode continu du temps) est limité avec les problèmes de taille de mémoire et le temps de calcul.

Pour résoudre ces problèmes, on peut simuler au niveau le plus haut dans la hiérarchie à partir du niveau algorithmique comme ISPS [BAR79] jusqu'au niveau logique (TEGAS [SZY 72]) ou niveau interrupteur (MOSSIM [BRY 81]). Mais à ces niveaux on perd en précision (mode discret).

Pour améliorer le rapport de coût/ précision, et pour simuler des circuits mixtes (analogique-numérique), on a besoin d'intégrer les deux types de simulation (mode continu et discret) dans un seul environnement de simulation. Le simulateur qui résulte de cette intégration a reçu le nom de mode-mixte (mixed-mode simulator). Beaucoup d'efforts ont été faits pour ce type de simulation comme : DIANA [REY 80], SPLICE [NEW 79] et MOTIS [CHA 75].

De plus, de nombreux circuits VLSI utilisés dans les systèmes de télécommunication et de contrôle (les filtres à capacités commutées, les convertisseurs A/N et N/A, les multiplexeurs analogiques...) supportent bien l'utilisation de simulateur de type mode mixte pour les raisons suivantes :

1) une caractéristique importante et commune de cette famille de circuits est le mélange de parties analogiques et numériques qui suggère l'utilisation de simulation de mode mixte.
2) avec les circuits VLSI, les expériences ont montré que seulement quelques parties de circuit ont besoin d'être simulé avec précision tandis que la simulation du reste du circuit n'est pas très critique.
3) la complexité des circuits VLSI implique l'intégration de la simulation de haut niveau avec le simulateur électrique [GAR 79].

Par la suite on va présenter les différentes techniques pour implémenter la simulation de mode mixte [EL 86], [EL 87].

3.4.1 Différentes techniques d'implémentation :

Avant de présenter les deux techniques d'implémentation de la simulation de mode mixte, il faut d'abord citer deux principes que l'on va prendre en considération pendant la discussion de ces techniques :

1) comment gère-t-on l'interface entre les différents niveaux de description ?
2) comment contrôle-t-on l'activité et l'inactivité de différents blocs ou sous circuits pendant la simulation?

A : Approche de "Global-scheduler" :

Cette approche est basée sur le regroupement de différents simulateurs (électrique, interrupteur, logique et fonctionnel), chacun peut être utilisé de manière indépendante pour un seul niveau de simulation, ou de manière concurrente pour la simulation de niveaux multiples.

Les simulateurs individuels ne s'interfacent pas entre eux directement mais ils communiquent seulement au travers d'un "Global scheduler". Chaque simulateur est concerné seulement avec ses parties du circuit global, tandis que le "Global scheduler" est responsable pour l'interconnexion globale du circuit. Le traducteur du simulateur partage le circuit global en un ensemble de sous circuits pour chaque simulateur individuel et un réseau qui interconnecte tous ces sous circuits.

Le protocole de communication entre le "Global scheduler" et chaque simulateur individuel est basée sur la notification d'événements comme dans la technique de simulation logique [BRE 64], [ULR 65,69] et [SZY 75], et la technique de "event driven selective trace" est utilisée pour diminuer le temps d'analyse. La propagation de signaux entre les différents niveaux de simulation est manipulée par un convertisseur/ propagateur de signal, qui transforme la représentation du signal d'un format à l'autre, si c'est nécessaire. Cette interface est réalisée soit en utilisant une fonction spéciale ou des circuits équivalents.

Des exemples de cette approche sont SPLICE [NEW 79], le système de Sandia [DAN 82] , Hughes [DAS 82] et CASCADE [BOR 85].

B : Approche d'une interface directe:

Dans cette approche l'interface entre les différentes analyses est faite de manière directe et on n'a pas besoin de construire le "Global scheduler". Le circuit est partagé en deux parties, la partie analogique (plus critique) et la partie numérique (moins critique). Chaque partie peut être décomposée de manière hiérarchique en sous circuits. Dans notre cas c'est l'environnement électrique qui remplace le "Global scheduler", et la communication par événement est faite

seulement à l'entrée de la partie numérique . La propagation de signaux entre les deux parties est manipulée de manière simple et précise et basée sur les caractéristiques de la technologie.

Les détails de cette approche seront présentées dans la section concernant la structure générale de notre simulateur de mode mixte.

Il faut noter que l'expérience a montré que l'on n'a pas en réalité besoin d'utiliser le simulateur électrique aux premières étapes de conception, ce n'est donc pas utile d'inclure ce simulateur dans un environnement de simulation de niveaux multiples (multi-level simulation). Mais l'utilisation du simulateur de mode mixte aux niveaux les plus bas va fournir l'avantage de la précision du simulateur électrique aussi bien que l'avantage du coût (taille mémoire et temps de calcul) de l'analyse fonctionnelle.

3.4.2 Le simulateur FIDELDO :

Le simulateur mixte est le résultat de l'intégration de FIDEL dans l'environnement électrique d' ELDO. Cette intégration fonctionnelle peut être présentée comme un macro modèle de haut niveau [RAB 79] pour l'analyse électrique. Notre simulateur mixte supporte deux niveaux de modélisation : il permet au circuit d'être décrit comme un mélange de transistors MOS et de blocs fonctionnels. Les transistors MOS sont simulés de manière électrique, tandis que les blocs fonctionnels sont simulés de manière fonctionnelle.

Le circuit est décrit comme une interconnexion de blocs en utilisant le langage de description ELDO (qui est similaire à SPICE). Le langage supporte la description hiérarchique , les blocs au niveau le plus bas peuvent être des transistors MOS ou des blocs fonctionnels. Le bloc fonctionnel est déclaré comme un sous circuit avec les paramètres d'entrée/sortie pour l'interface. La description globale est analysée par ELDO et le résultat est utilisé par le simulateur.

La figure (27) montre la structure globale de notre système. La partie digitale du circuit est décrite en FIDEL et compilée par le compilateur de FIDEL. L'image du

simulateur mixte est obtenue par l'édition de liens entre le résultat de compilation, l'interpréteur FIDEL, l'analyseur de ELDO et le simulateur ELDO. L'interpréteur FIDEL peut accéder aux données internes pour l'exécution de la partie fonctionnelle. Cette exécution est activée, soit si une entrée de partie fonctionnelle a changé soit si le temps de simulation est arrivé à la valeur de réveil demandée par la partie fonctionnelle.

L'interface entre FIDEL et ELDO s' est fait de manière similaire à l'interface de FIDEL et EPILOG, mais pendant l'implémentation de cette interface entre les deux parties (analogique et logique) on a pris en compte les considérations suivantes :

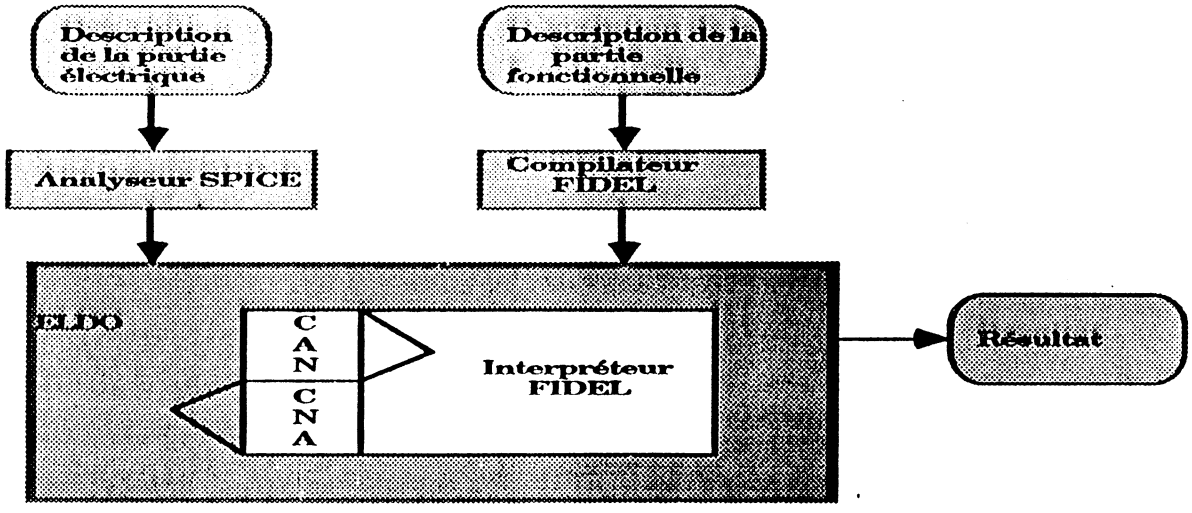


FIG 27: STRUCTURE GENERALE

1) Propagation et conversion de signaux :

A l'interface où des portes électriques qui fonctionnent en mode continu dirigent des blocs fonctionnels qui fonctionnent en mode discret, on a besoin de fonctions spéciales pour propager et mettre à jour les signaux. Ces fonctions sont basées sur les caractéristiques du transistor et sont similaires à celles de DIANA [ARN 78].

La traduction digital - analogique (CNA), traduit les valeurs numériques (0,1) dans les valeurs électriques correspondantes au VSS et VDD (par exemple 0 et 5 volts).

La traduction analogique-digital (CAN) prend en considération la variation d'un signal analogique pendant une période de temps T, qui est choisie par le concepteur selon la sensibilité du circuit. La figure (28) montre les cinq valeurs de tension pour transférer un signal analogique à la valeur digitale correspondant avec les différents cas possibles. Ces valeurs sont : max-high, min-high, max-low, min-low et la valeur intermédiaire. Les valeurs max-high et min-high détermine la fenêtre pour détecter la valeur logique "1". Les valeurs max-low et min-low détermine la fenêtre pour détecter la valeur logique "0". Un changement dans la valeur du signal qui ne dépasse pas la valeur intermédiaire ou plus court que la période T n'est pas pris en compte, comme ceci on filtre les "glitches".

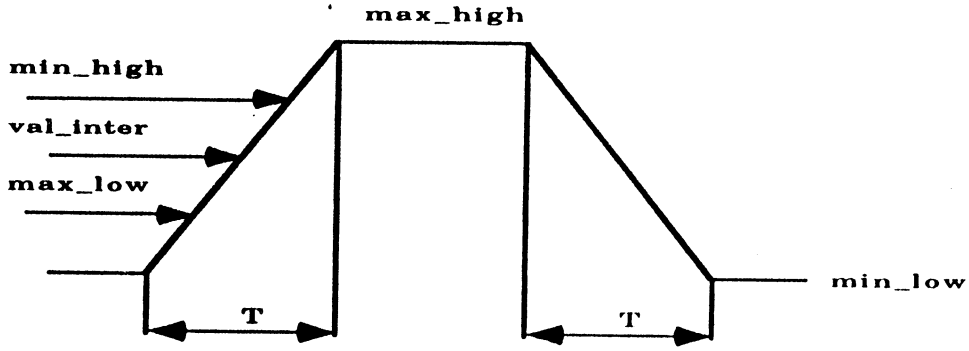


FIG . 28 : VALEURS DE CAN

2) l'état stable initial :

Avant que le simulateur électrique ne commence la phase d'exécution (simulation), il fait quelques calculs pour arriver à un état stable initial. Pour la partie analogique ces calculs dépendent des valeurs initiales qui sont proposées par le concepteur et de l'interconnexion entre les éléments du circuit. Pour la partie digitale, on doit activer l'interpréteur fonctionnel au temps T0 pour déterminer les valeurs initiales de la partie fonctionnelle.

Pour mieux implémenter ce simulateur mixte on a étudié deux approches différentes : soit FIDEL pilote ELDO comme un sous programme, soit ELDO pilote FIDEL comme un sous programme. Les deux présentations sont montrées par la suite :

Cas A : FIDEL pilote ELDO :

- PHASE INITIALE :

1) FIDEL commence le calcul initial de ses sorties et appelle ELDO.
2) ELDO calcule les nouvelles valeurs de ses sorties (entrées FIDEL) et donne la main à FIDEL.
3) FIDEL doit traiter cette phase en temps nul, aucun noeud ne doit être dans l' état X à la fin de cette phase.

- PHASE DE SIMULATION :

1) Au temps T, FIDEL traite tous les événements logiques antérieurs à tout événement électrique au même temps.
2) Si un événement électrique est détecté au temps T, FIDEL appelle ELDO en passant les valeurs d'entrée et la date du prochain événement logique : NT > T + H (H est le pas de calcul d'ELDO).
3) ELDO effectue la conversion de ses entrées et calcule un nouveau pas de simulation en respectant la contrainte :
   T + H < = NT
   ELDO calcule les noeuds analogiques par relaxation et effectue la conversion de ses sorties.
4) Si un changement à l'interface est détecté, ELDO demande à FIDEL de stocker un nouvel événement logique à ce temps et donne la main à FIDEL.
5) Si on ne détecte pas de changement, ELDO continue avec un nouveau pas de calcul (simulation) jusqu'à ce qu'on détecte un changement à l'interface ou que son temps de calcul devienne égal à la valeur de NT, et on donne la main à FIDEL.
6) Dans les deux derniers cas, FIDEL doit stocker un nouvel événement logique au temps NT (avec une priorité inférieure à n'importe quel événement logique enregistré à ce temps).

CAS B : ELDO pilote FIDEL :

- PHASE INITIALE :

  Au temps 0, ELDO effectue sa phase d'initialisation et appelle FIDEL pour calculer ses sorties (sans retard) à ce temps. Aucun noeud ne doit être à l'état X.

- PHASE DE SIMULATION :

1) Au temps T, ELDO effectue sa simulation avec un pas de calcul H, si ELDO détecte un changement à l'interface, il fait la conversion de valeurs et passe la main à FIDEL; si non il continue sa simulation avec un nouveau pas de calcul.
2) Quand il est appelé, FIDEL effectue son exécution. Si FIDEL détecte un événement logique interne , il l'enregistre et demande le réveil à l'heure NT >= T + H, et passe la main à ELDO avec les valeurs de sorties.
3) ELDO continue sa simulation avec un nouveau pas en respectant la contrainte : NT >= T + H et on continue comme dans le cas précédent.

REMARQUE :

Dans les deux cas ELDO fournit une valeur du retard associée à chaque sortie ELDO et FIDEL doit la prendre en compte pendant le calcul d'événements internes.

Pour des raisons de simplicité de programmation on a choisi la deuxième technique c'est à dire ELDO qui pilote FIDEL.

Enfin, pour montrer l'efficacité de notre simulateur FIDELDO, on a fait des comparaisons pratiques entre FIDELDO et ELDO seul. Cette comparaison s'est effectuée sur trois exemples : AD4B (4 bascules D et 3 additioneurs à un bit), partie d'une RAM de 512 mots de 8 bits et un convertisseur analogique/numérique [EL 87]. Un exemple de cette comparaison sera présenté dans l'annexe D, la figure (29) montre le résultat de cette comparaison. Le simulateur mixte est plus rapide

qu' ELDO d'un facteur dépendant du rapport entre les tailles de la partie numérique et de la partie analogique. Comme ELDO est environ un ordre de grandeur plus rapide que SPICE, alors le simulateur FIDELDO peut facilement être deux ordres de grandeur plus rapide que SPICE mais avec une bonne précision.

	ELDO	FIDELDO
AD4B	7 min	3 min
RAM	10 min	4 min
CAN	300 min	20 min

FIG . 29 : COMPARAISON ELDO-FIDELDO

CONCLUSION

Dans ce chapitre, on a présenté la simulation comme un outil de vérification et de validation des circuits VLSI. En général, la simulation peut être classée selon les différents niveaux de description : fonctionnel, logique, interrupteur et électrique, ou selon les modes de manipulation du temps : continu, discret.

Dans la deuxième partie on a décrit l'intégration de FIDEL dans le simulateur logique EPILOG qui aboutit au type de simulation de niveau mixte (logico-fonctionnel), et dans la troisième partie on a présenté l'intégration de FIDEL dans le simulateur ELDO comme un type de simulation de mode mixte. Ces deux applications sont une avancée dans le domaine de la simulation, dans le but de garder la précision tout en diminuant le coût de simulation des circuits VLSI.

CHAPITRE IV

LE SIMULATEUR FIDEL

INTRODUCTION

Dans le chapitre II, on a présenté FIDEL en tant que langage fonctionnel décrivant le comportement d'un circuit comme une relation entre les sorties et les entrées (en tenant compte du temps et des variables internes). L'élément de base à ce niveau de description est le modèle fonctionnel, qui peut représenter un circuit ou une partie du circuit. Les applications utilisant les modèles fonctionnels sont présentées dans le chapitre III, où l'on a discuté deux approches de simulation : la simulation de niveau mixte (logico-fonctionnelle) et la simulation de mode mixte (analogique - fonctionnelle).

Dans ces approches, on décrit le circuit ou une partie du circuit en FIDEL, et on effectue la simulation en utilisant l'environnement fourni par le simulateur logique (dans le cas du niveau mixte) ou électrique (dans le cas du mode mixte). Une caractéristique commune des deux environnements de simulation est que dans les deux cas on peut décrire le circuit de manière hiérarchique, la description étant éclatée au niveau interne avant de lancer la simulation. L'inconvénient de l'éclatement de la description est que chaque modification d'un sous circuit entraîne la recompilation de toute la description.

Dans ce chapitre, on présente le langage de description et l'environnement de simulation de FIDEL en tant qu'outil indépendant.

La première partie est consacrée à l'aspect langage de description: nous montrons dans cette partie les différents avantages d'une description hiérarchique par rapport à une description éclatée.

La deuxième partie est consacrée à l'introduction du modèle d'interrupteur en FIDEL : choix de description et implémentation d'algorithme de base.

L'utilisation de FIDEL comme langage de description structurelle est présenté dans la troisième partie. Les points abordés dans le chapitre I sur la description structurelle, sont discutés en détail et illustrés par différents exemples.

Enfin, l'environnement de simulation en FIDEL sera le thème des quatrième et cinquième parties : nous présenterons en détail la description des stimuli et l'algorithme de simulation.

1. CONCEPT DE DESCRIPTION HIERARCHIQUE EN FIDEL

La description hiérarchique est la base de la méthode descendante de conception. En utilisant le raffinement étape par étape on fournit des informations détaillées jusqu'au niveau le plus bas de la description. Cette description permet au concepteur de créer de manière indépendante les différentes parties d'une hiérarchie complexe.

La description hiérarchique en FIDEL est basée sur les principes présentés au paragraphe 4.2.2 du chapitre I.

L'utilisation de FIDEL permet de méler les descriptions structurelle et fonctionnelle à n'importe quel niveau de la hiérarchie. Le traitement de la hiérarchie par FIDEL ne nécessite pas la recompilation complète du circuit à chaque modification interne dans un sous circuit.

Avec la modélisation en FIDEL, les concepteurs peuvent réutiliser les unités déjà créées à chaque niveau de hiérarchie (sous réserve de l'existence d'une bibliothèque de modèles).

La définition et la réalisation de FIDEL "stand-alone" (c'est à dire un environnement de description et de simulation complets), essaient de satisfaire les contraintes suivantes :

- Description multi niveau :
  ceci permet au concepteur de spécifier le comportement, la structure ou le mélange des deux aux différents niveaux de description.

- Description hiérarchique :
  ceci permet au concepteur de décomposer son circuit en sous circuits, qui peuvent être simulés de manière indépendante, intégrés ensuite pour construire le circuit global.

- Description paramètrable :
  cette facilité permet au concepteur de spécifier un prototype générique d'une unité, et de l'utiliser pour générer différents composants de ce type.

- Description et simulation du parallélisme :
  ceci peut être fait de deux façons : soit localement au niveau d'une unité en utilisant l'instruction PAR ou les événements en concurrence ; soit globalement par l'activation simultanée de différentes unités à travers leurs interconnexions.

2. INTRODUCTION DU NIVEAU INTERRUPTEUR

Les niveaux les plus utilisés pour la vérification des circuits VLSI sont le niveau électrique et le niveau logique. La simulation électrique fournit des informations détaillées et précises, mais consomme du temps. Par contre la simulation logique est plus rapide, mais moins précise. La simulation au niveau interrupteur [BRY80] , [HAC 82] , [HEY 82] , [HEY 83] et [BAN 85] a été développée pour obtenir un compromis entre vitesse et précision.

2.1Concepts du niveau interrupteur (SWITCH) :

L'élément de base d'une description switch est l'interrupteur modélisant le transistor MOS représenté dans la figure (30).

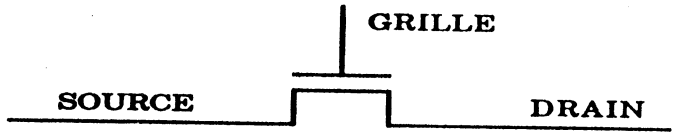


FIG. 30 : L'ELEMENT DE BASE DU NIVEAU SWITCH

Une description switch comporte essentiellement un réseau d'interrupteurs reliés par des lignes. La spécification plus ou moins fine des interrupteurs et des lignes peut varier d'une description à l'autre, mais la structure essentielle reste inchangée.

La principale caractéristique de ce type de réseau réside dans le caractère bi-directionnel des interrupteurs donc des connexions (contrairement aux portes logiques qui sont toujours unidirectionnelles).

La propagation d'un signal se réduit alors à deux opérations principales : rendre un transistor bloqué ou rendre un transistor passant, avec les opérations de mise à jour qui en découlent.

Il faut remarquer que le traitement des interrupteurs conduit implicitement à supposer que les transistors MOS sont symétriques et donc complètement bidirectionnels.

Cette contrainte n'est pas respectée dans la majorité des simulateurs switch existants qui supposent une asymétrie du transistor MOS (traitement différent du drain et de la source). Ceci est effectivement vérifié si le circuit ne comporte pas de porte de transmission.

Cette manière de traiter les circuits simplifie les calculs mais laisse de côté tous les problèmes de conflit sur les sorties (où se posent justement les problèmes de bidirectionnalité), et ne reflète pas exactement le principe de ce niveau, illustré par son nom : niveau interrupteur.

En résumé les principales différences entre le niveau logique et le niveau interrupteur [HEY 83] sont les suivantes :

- Comportement des interconnexions :
  Dans un réseau d'interrupteurs les noeuds d'interconnexions sont actifs :
  * le stockage des informations est dû aux capacités parasites des interconnexions et des interrupteurs.
  * la transmission des informations n'est pas instantanée.

- Les composants ne peuvent pas être modélisés par une fonction explicite fournissant les sorties en fonction des entrées, la notion d'entrée/sortie perdant son sens à cause du caractère bidirectionnel des interconnexions.

2.2 Modélisation du niveau interrupteur en FIDEL :

Un circuit MOS est modélisé par un ensemble d'interrupteurs à trois terminaux représentant les transistors. L'état d'un noeud de connexion est décrit par un couple (a, b) où "a" représente la force du noeud et "b" représente son niveau logique (valeur).

Pour définir la correspondance entre valeurs logiques des noeuds et tensions électriques nous utilisons le modèle de NMOS à 5-V [MED 79] dont nous rappelons les caractéristiques :

* tension de seuil de 1.0 V pour le transistor enrichi.
* tension de seuil -3.0 V pour le transistor déplété.
* rapport de géométrie entre le transistor de charge et le transistor de commande égal à 4 : 1 ; ceci correspond à une tension de seuil de commutation d'un inverseur à environ 2.3 V.

Les cinq valeurs logiques de notre modèle sont :

- La valeur logique "0" correspond à un "zéro fort", elle est interprétée en général comme le niveau logique le plus bas, elle représente une tension inférieure à 1.8 V.

- La valeur logique "0*" correspond à un "zéro faible", elle représente une tension de 1.8 V à 2.2 V . Si elle est appliquée à la grille d'un "latch" dynamique, elle peut décharger le drain si la source est à la masse.

- La valeur logique "X" représente un niveau indéterminé et correspond aux valeurs de tension de 2.2V à 2.6 V.

- La valeur logique "1*" qui représente "1 faible", elle représente les valeurs de la tension entre 2.6 à 4.0 V . Elle ne peut pas rendre un transistor de transmission complètement conducteur.

- La valeur logique "1" représente "1 fort", elle est toujours définie comme le niveau le plus haut. Elle représente les valeurs de la tension supérieure à 4.0 V .

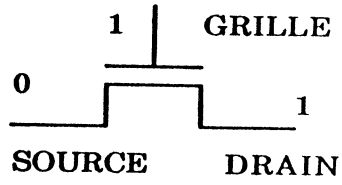


FIG. 31 : LE ROLE DE LA FORCE

Remarque :

Il faut noter que les notions "fort" et "faible" sont liés à la discrétisation des niveaux de tension .

Pour illustrer le rôle de la force du noeud, on prend l'exemple de la figure (31) et on suppose que les valeurs de chaque terminal sont les suivantes : grille = "1", source = "0" , drain = "1" ; il est alors impossible de prédire les nouvelles valeurs de source et de drain.

Des informations complémentaires doivent donc être fournies : ceci est fait par les spécifications de la force du noeud.

Pour simplifier le modèle de force, on classe les transistors en deux catégories de tailles (W/L), "large" et "petite" qui peuvent être affectées par le concepteur. Le but de cette classification est de simplifier le modèle en diminuant le nombre d'états. On peut de cette manière définir quatre forces pour un noeud de connexion :

- "I" :
  La force la plus forte ; elle représente la force des noeuds connectés à VDD, VSS ou une entrée primaire.

- "S" :
  représente la force d'un noeud qui est connecté à un noeud d'entrée soit à travers un transistor de type enrichi qui est complètement passant avec un W/L qui est large ; soit à travers un transistor déplété avec large W/L.

- "W" :
  représente la force d'un noeud connecté à un noeud d'entrée soit à travers un transistor enrichi qui n'est pas complètement passant et avec un large W/L, soit à travers un transistor déplété avec un petit W/L.

- "C" :
  c'est la plus faible; elle représente la force d'un noeud qui n'est pas connecté à un noeud d'entrée.

Quatre niveaux de force et cinq valeurs logiques sont suffisantes pour représenter tous les états des noeuds de connexions. Un certain nombre de combinaisons ne sont pas permises : par exemple un noeud qui a la force "I", peut seulement avoir les valeurs "0" ou "1", donc les couples (I,0*), (I,1*) et (I,X) ne doivent pas exister.

Modèle du transistor NMOS enrichi :

Pour calculer l'état d'un transistor NMOS enrichi, on suit les règles suivantes :

1) **Le transistor est bidirectionnel et symétrique : source et drain sont interchangeables.**

2) **Le comportement du transistor est indépendant de la force de grille, seule la valeur de grille est importante.**

3) **Pour rendre le transistor passant, il doit exister une différence de tension entre la grille et la source d'au moins 1V ce qui représente la tension de seuil du transistor NMOS enrichi.**

4) **Si la condition 3) est satisfaite, alors on tient compte des forces de source et drain pour déterminer le nouvel état du dispositif. Le nouvel état est déterminé par le noeud le plus fort, par exemple si les noeuds sont (I,0) et (W,1), alors c'est le (I,0) qui commande le nouvel état.**

5) **La valeur du noeud faible est déterminée par le noeud fort en gardant toujours la différence de tension entre la grille et la source (ceci montre bien l'intérêt de notions "fort" et "faible").**

6) **La force du noeud faible est déterminée selon les conditions que nous avons données dans la définition des différents niveaux de forces.**

7) **La force de la grille est changée à "C" pour modéliser l'effet de capacité de grille.**

8) **Enfin, si la tension de grille est "0", alors les forces de source et drain seront mises à "C" en gardant les anciennes valeurs logiques.**

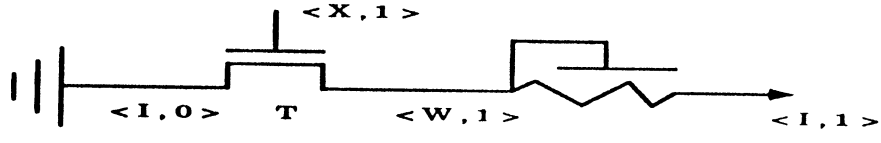


FIG . 32 : MODELE DU TRANSISTOR NMOS

La figure (32) montre un transistor NMOS enrichi T, dont la grille est à (X,1) ("X" ici signifie que la force de grille est sans effet), la source est à (I,0) (représente un noeud lié à la masse), et le drain est à (W,1) (représente un noeud connecté à un noeud d'entrée à travers un transistor de charge déplété avec W/L petit). Le prochain état est déterminé par le noeud (I,0) qui est le plus fort. La valeur logique du noeud faible (W,1) sera à "0" et sa force devient "S" selon la définition d'un noeud (S,0), c'est à dire un noeud qui est connecté à un noeud d'entrée (I,0) à travers un transistor enrichi avec un W/L large.

Ordonnancement d'états du noeud :

Le but de cette opération est de résoudre le cas où plusieurs terminaux sont reliés au même noeud, chacun essayant d'affecter un état au noeud. L'état résultant du noeud est déterminé en affectant une priorité aux différents états possibles en fonction des règles suivantes :

- si deux états sont de forces différentes : I> S >W > C.

- Si deux états ont la même force, le choix est arbitraire. Nous avons choisi les conventions suivantes : (I,1) > (I,0) ; (S,0) > (S,1) ; (W,0) > (W,1) ; et (C,1) > (C,0).

Donc, l'ordre global entre les différents états est :

(I,1) ; (I,0) ; (S,0) ; (S,1) ; (S,0*) ; (S,1*) ; (W,0) ; (W,1) ;
(W,0*) ; (C,1) ; (W,1*) ; (C,1*) ; (W,X) ; (C,X) ; (C,0*) ; (C,0).

En général l'état résultant d'un noeud est représenté par la relation d'ordre suivante :

$$O(\text{noeud}) = MAX[(a_1,b_1), (a_2,b_2), \ldots\ldots(a_n,b_n)]$$

Dans la figure (33), on calcule l'ordre du noeud N par :

$$O(N) = MAX[(C,0), (W,1*), (W,1), (S,0)] = (S,0)$$

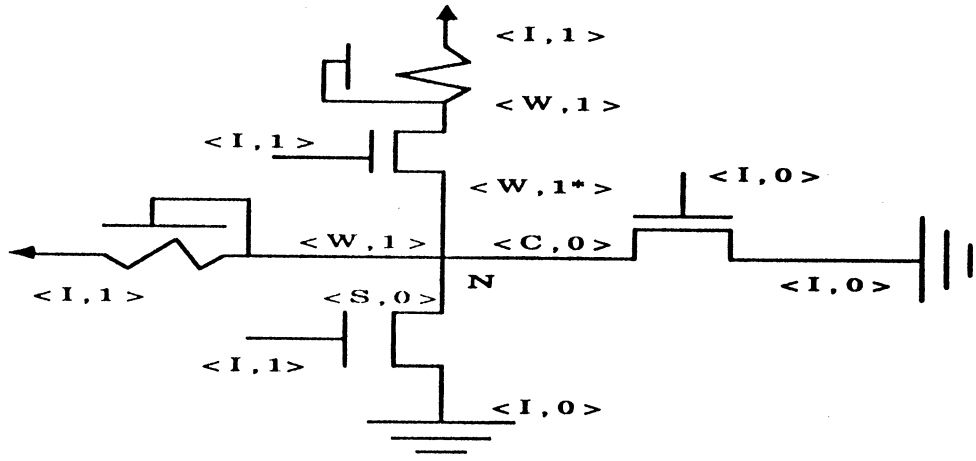


FIG . 33 : L' ORDRE DE NOEUDS

L'algorithme général :

Nous résumons l'algorithme général choisi dans notre approche :

- Initialisation des forces des noeuds qui ne sont pas noeuds d'entrée à "C" et de leurs valeurs logiques à "X" (si elles ne sont pas spécifiées).

- Recherche d'un état stable pour le circuit par itération. Chaque itération consiste en deux opérations :
  * calcul des nouveaux états de chaque transistor.
  * ordonnancement de noeuds comme illustré précédemment.

- Quand un état stable est atteint avec un ensemble d'entrée donné, réinitialisation de la force de chaque noeud qui n'est pas un noeud d'entrée à C , la valeur logique du noeud restant inchangée .

- application de l'ensemble d'entrée suivant.

Dans cette partie on a présenté le modèle et l'algorithme applicable au transistor NMOS enrichi; ceci peut être applicable aux autres types de transistor (NMOS déplété, PMOS enrichi et CMOS); les détails sont présentés dans [BAN 85].

Un exemple est présenté dans la figure (34), les trois itérations de l'algorithme sont présentées dans la figure (35).

Les limitations de cette approche sont : pas d'information temporelle dans le modèle, pas de prise en compte du partage de charges entre les noeuds. Pour

résoudre ces limitations, il faut prendre en compte les valeurs relatives des capacités de noeuds "C" aussi bien que la résistance du transistor. Mais, ceci va augmenter le nombre d'états des noeuds et en conséquence augmenter la complexité du modèle et l'algorithme.

L'introduction du niveau interrupteur en FIDEL donne la possibilité au concepteur d'étudier plus précisément le comportement d'un circuit. Nous avons également montré que FIDEL peut fournir un environnement de description et de simulation multi niveaux . Avant de terminer cette partie, il faut remarquer que l'on peut utiliser le modèle et l'algorithme présentés ici, pour calculer l'état initial dans notre simulateur mixte FIDELDO.

2.3 L'interface avec FIDEL :

La description de connexion d'un réseau d'interrupteurs (seul ou avec d'autres composants FIDEL) sera présentée dans la partie suivante. Nous allons essayer ici d'expliquer le dialogue et le passage de valeurs entre le modèle fonctionnel et le modèle d'interrupteur.

Le modèle d'interrupteur manipule cinq valeurs (0, 0*, X, 1*, 1), mais le modèle fonctionnel ne distingue pas entre (0 et 0*) et (1 et 1*). Donc, pour les passage de valeurs entre les deux modèle on ne traite que (0, 1 et X). C'est à dire que pour un noeud sortant du modèle d'interrupteur vers un modèle fonctionnel on ne tient pas compte de sa condition, mais seulement de sa valeur (0,1 et X). Dans le cas inverse, c'est à dire un noeud sortant du modèle fonctionnel vers le modèle d'interrupteur, on garde sa valeur (0,1 et X) mais on l'associe avec la condition (I).

Pour un noeud de type INPOUT, on suppose la priorité du modèle fonctionnel. C'est à dire qu'à chaque fois que ce noeud change de valeur pendant une évaluation du modèle fonctionnel, on le traite comme un noeud sortant. Si entre deux évaluations sa valeur ne change pas, on l'associe avec la condition faible (C) et on garde son ancienne valeur.

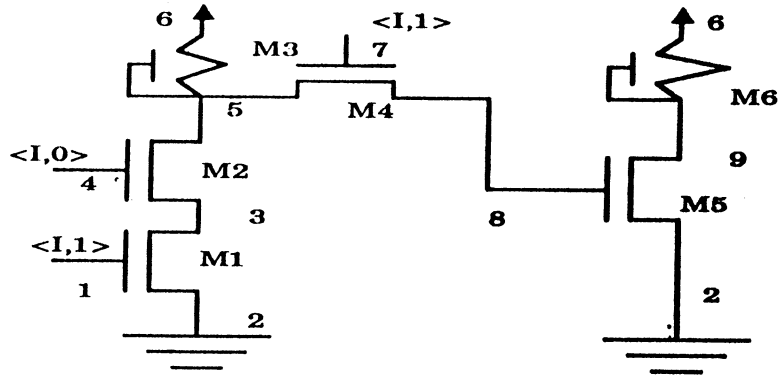


FIG. 34 : EXEMPLE D' UN CIRCUIT

Première étape

Terminaux	état courant	état prochain	entrée
M1.G M1.S M1.D	1: < I,1 > 2:<I,0> 3:<C,0>	1:<C,1> 2:<I,0> 3:<S,0>	1:<I,1> 2:<I,0> 4:<I,0> 6:<I,1> 7:<I,1>
M2.G M2.S M2.D	4:<I,0> 3:<C,0> 5:<C,0>	4:<C,0> 3:<C,0> 5:<C,0>	ordre
M3.d M3.S	6:<I,1> 5:<C,0>	6:<I,1> 5:<W,1>	1:<I,1> 2:<I,0> 3:<S,0> 4: <I,0> 5:<W,1> 6:<I,1> 7:<I,1> 8:<C,0> 9:<W,1>
M4.G M4.S M4.D	7:<I,1> 5:<C,0> 8:<C,0>	7:<C,1> 5:<C,0> 8:<C,0>	
M5.G M5.S M5.D	8:<C,0> 2: <I,0> 9:<C,0>	8:<C,0> 2:<C,0> 9:<C,0>	
M6.D M6.S	6:<I,1> 9:<C,0>	6:<I,1> 9:<W,1>	

Deuxième étape

Terminaux	état courant	état prochain	entrée
M1.G M1.S M1.D	1: < I,1 > 2:<I,0> 3:<S,0>	1: < C,1 > 2:<I,0> 3:<S,0>	1:<I,1> 2:<I,0> 4:<I,0> 6:<I,1> 7:<I,1>
M2.G M2.S M2.D	4:<I,0> 3:<S,0> 5:<W,1>	4:<C,0> 3:<C,0> 5:<C,1>	ordre
M3.d M3.S	6:<I,1> 5:<W,1>	6:<I,1> 5:<W,1>	1:<I,1> 2:<I,0> 3:<S,0> 4: <I,0> 5:<W,1> 6:<I,1> 7:<I,1> 8:<W,1*> 9:<W,1>
M4.G M4.S M4.D	7:<I,1> 5:<W,1> 8:<C,0>	7:<C,1> 5:<W,1> 8:<W,1*>	
M5.G M5.S M5.D	8:<C,0> 2: <I,0> 9:<W,1>	8:<C,0> 2: <C,0> 9:<C,1>	
M6.D M6.S	6:<I,1> 9:<W,1>	6:<I,1> 9:<W,1>	

Troisème étape

Terminaux	état courant	état prochain	entrée
M1.G M1.S M1.D	1: < I,1 > 2:<I,0> 3:<S,0>	1: < C,1 > 2:<I,0> 3:<S,0>	1:<I,1> 2:<I,0> 4:<I,0> 6:<I,1> 7:<I,1>
M2.G M2.S M2.D	4:<I,0> 3:<S,0> 5:<W,1>	4:<C,0> 3:<C,0> 5:<C,1>	ordre
M3.d M3.S	6:<I,1> 5:<W,01>	6:<I,1> 5:<W,1>	1:<I,1> 2:<I,0> 3:<S,0> 4: <I,0> 5:<W,1> 6:<I,1> 7:<I,1> 8:<W,1*> 9:<W,1>
M4.G M4.S M4.D	7:<I,1> 5:<W,1> 8:<W,1*>	7:<I,1> 5:<W,1> 8:<W,1*>	
M5.G M5.S M5.D	8:<W,1*> 2: <I,0> 9:<W,01>	8:<C,1*> 2: <I,0> 9:<S,0>	
M6.D M6.S	6:<I,1> 9:<W,1>	6:<I,1> 9:<W,1>	

FIG . 35 : EXECUTION D' UN EXEMPLE

3. DESCRIPTION STRUCTURELLE

Le terme description structurelle signifie (comme nous l'avons indiqué dans le premier chapitre) la description d'un circuit ou d'une de ses parties par un ensemble de modules et de leurs interconnexions. Ces modules sont eux aussi décomposables hiérarchiquement de la description de haut niveau jusqu' à la description des modules primitifs ou élémentaires.

3.1 Concept de description structurelle en FIDEL :

On a mentionné dans le chapitre I que le langage de description doit permettre la description de structure de façon explicite. Ces descriptions ne doivent pas être limitées à un seul niveau d'abstraction. Une description de structure explicite doit fournir seulement les informations de connexion des modules inclus. Il ne doit pas exister d'information de comportement dans une description de structure explicite. Pour que la simulation soit possible, la description comportementale doit apparaître au niveau le plus bas dans la hiérarchie de la description. Ces principes sont respectés dans la description de structure en FIDEL.

La séparation entre la description de structure et la description de comportement est très utile pour une simulation multi niveaux efficace. Puisqu'un comportement particulier peut être réalisé en utilisant plusieurs structures différentes, cette séparation des deux descriptions permet de les modifier de façon indépendantes.

La description de structure en FIDEL est basée sur la méthodologie de conception structurée, c'est à dire que le système au niveau le plus haut peut être vu comme un seul module abstrait (niveau fonctionnel), tandis qu'au niveau le plus bas d'abstraction, le système peut être présenté comme un réseau de modules primitifs (c'est à dire de modules non décomposables).

Dans FIDEL il n'existe pas de modules primitifs prédéfinis ; les feuilles dans l'arbre de description structurelle hiérarchique sont toujours soit des modèles fonctionnels (dont les fonctions sont fournies par les concepteurs), soit des transistors MOS (qui représentent des éléments de base sans fonction prédéfinie).

Cette approche a l'avantage de laisser au concepteur la liberté de définir ses modules primitifs selon ses besoins, et évite la construction d'une bibliothèque spéciale pour ces modules.

Enfin, la description structurelle en FIDEL permet la définition de composants génériques. Le composant générique est un composant avec un ensemble de paramètres fournis par le concepteur. Ces paramètres vont permettre au concepteur pendant la phase de compilation de générer les composants désirés.

3.2 Le modèle structurel :

Un modèle structurel en FIDEL a un nom, une liste d'éléments d'interface, une liste de composants avec leurs types, et la spécification des connexions des composants.

Le nom d'un modèle structurel est connu dans toute la description FIDEL (nom global). Il doit être unique dans un description FIDEL. En principe, dans la version actuelle de FIDEL, on doit décrire un composant avant de l' utiliser dans un modèle structurel (il n' y a pas de bibliothèque pour stocker les modèles). Il y a trois parties dans la définition d'un modèle structurel :

- l'entête du modèle (interface externe).
- la partie définition des éléments qui le constituent (variables, composants).
- la partie description de structure (l'interconnexion entre les différents composants).

La partie entête contient le nom du modèle (qui doit être unique dans la description) et les éléments d'interface avec le monde externe. Les types de ces éléments sont : **INPUT**, **OUTPUT** et **INPOUT** (pour la connexion bidirectionnelle).

Exemple :

SMODEL ADDER (CIN , A , B , COUT , SUM) ;

Dans la partie définition, on déclare les différents types d'éléments d'interface et les différents types de composants avec leurs instances; dans certains

cas on déclare des variables du type **INTEGER** utilisées dans une boucle de description ou comme indice de vecteur (élément de connexion ou composant).

Exemple :

```
DECLARE INPUT CIN, A(0:3), B(0:3);
DECLARE OUTPUT COUT, SUM(0:3) ;
DECLARE INTEGER I ;
```

Dans cet exemple on définit les éléments d'interface du modèle ADDER : les entrées sont CIN, A et B. La variable CIN a un seul bit, mais les variables A et B sont des vecteurs de quatre bits chacun. Les sorties sont COUT (un seul bit) et SUM qui est un vecteur de 4 bits. La variable I est une variable locale du type **INTEGER**.

La déclaration des composants spécifie l'instance et le type de chaque composant utilisé dans la description de structure. L'exemple suivant montre différents types de déclarations.

EXEMPLE :

```
DECLARE COMPONENT FADD1, FADD2 : FULLADD ;
DECLARE COMPONENT FADD(0:1) : FULLADD ;
DECLARE COMPONENT T1 : SWITCH (NMOS, 0 );
```

Dans la première ligne, on définit deux composants FADD1,FADD2 du type FULLADD (qui représente un modèle FIDEL déjà décrit), ce modèle peut être un modèle structurel ou fonctionnel.

La deuxième ligne représente la même déclaration mais en utilisant un vecteur de composants FADD qui contient deux instances du type FULLADD.

La troisième ligne représente la déclaration d'un composant du type **SWITCH** (interrupteur) . C'est un interrupteur NMOS avec la valeur de W/L petite (égale 0 dans l'exemple). La syntaxe de déclaration des interrupteurs est la suivante :

interrupteur ::= element_type "(" transistor_type "," "DEC-NUMBER" ")" *

element_type::= " **SWITCH** " , " **PULLUP** " , " **PULLDOWN** " *

transistor_type ::= " **NMOS** " , " **PMOS** " , " **CMOS** " *

La partie de description de structure représente l'interconnexion entre les différents composants déclarés. L'instruction de base est l'instruction **CONNECT**. Sa syntaxe est la suivante :

instruction_connect ::= "**CONNECT** " connection_element ","
connection_element { "," connection_element } *

L'instruction **CONNECT**, connecte au moins deux éléments. La syntaxe pour décrire un élément de connexion est la suivante :

connection_element ::= component_name "." connection_name ,
" **SUPPLY** " , " **GROUND** " *

connection_name ::= connection_instance , "**GATE**" , "**SOURCE**" ,"**DRAIN**" *

L'élément de connexion peut être donc la connexion d'un composant ou une entrée prédéfinie (**SUPPLY** et **GROUND**, qui ont les valeurs logique 1 et 0 respectivement). Le nom de connexion peut être un élément d'interface d'un composant ou bien l'un des trois terminaux d'un interrupteur (grille, source et drain). Pour mieux comprendre la structure de l'instruction **CONNECT**, nous l'illustrons par le schéma de la figure (34).

Exemple :

CONNECT CIRCUIT . INPUT1 , M1. **GATE** , M4 . **GATE**
CONNECT CIRCUIT . INPUT2 , M2 . **GATE**
CONNECT M1 . **SOURCE** , M5 . **SOURCE** , **GROUND**
CONNECT M3 . **DRAIN** , M6 . **DRAIN** , **SUPPLY**
CONNECT M1 . **DRAIN** , M2 . **SOURCE**
CONNECT M2.**DRAIN** , M3.**SOURCE** , M3.**GATE** , M4 . **SOURCE**
CONNECT M4 . **DRAIN** , M5 . **GATE**
CONNECT M5 . **DRAIN** , M6 . **SOURCE** , M6 . **GATE**

Remarque :

De point de vue sémantique, l'ordre de connexion est important. C'est à dire que la signification d'une instruction **CONNECT** est : connexion d'un élément source (entrée du modèle principal dans la hiérarchie ou sortie d'un composant de ce modèle) vers des éléments destinations (sortie du modèle principal ou entrée d'un composant de ce modèle). Si cette règle n'est pas respectée, un message d'avertissement sera généré.

En plus de cette règle, la vérification des dimensions des connexions est effectuée pendant la phase de traduction (interprétation du code structurel) avec la génération des messages d'avertissement (si besoin est).

Dans cette description CIRCUIT représente le modèle structurel , INPUT1, INPUT2 sont les deux entrées de CIRCUIT et ont les valeurs "1" et "0" respectivement. Les deux interrupteurs M3, M6 doivent être déclarés du type "**PULLUP**".

Dans certaines applications il est nécessaire de spécifier les connexions de manière régulière comme dans le cas des circuits systoliques. L'instruction "**FOR**" est utilisée dans ce cas.

Exemple :

```
FOR I =0 UPTO 3 DO
        CONNECT   ADDER . INP ( I ), FADD ( I ) . E ( I)
ENDFOR
```

En plus, l'instruction "**IF**" permet de décrire des connexions conditionnelles, comme le montre l'exemple suivant :

Exemple :

```
FOR I = 0 UPTO 3 DO
   IF I = 0 THEN CONNECT ADDER . CIN , FADD ( I) . CARIN ENDIF
   CONNECT FADD ( I - 1 ) . CAROUT,  FADD ( I) . CARIN
   IF I = 3 THEN CONNECT FADD ( I) .CAROUT, ADDER . COUT  ENDIF
ENDFOR
```

Nous avons décrit jusqu' à présent des structures régulières et de taille fixe. Une flexibilité supplémentaire est nécessaire pour spécifier des structures variables telles que : registres à décalage de taille variable, mémoires de taille variable ...

Les paramètres de généricité sont utilisés pour contrôler cette variabilité dans le nombre de composants, la taille d'un signal aussi bien que dans la valeur du retard dans un modèle fonctionnel. Ces paramètres sont déclarés dans l'entête du modèle :

Exemple :

PSMODEL ADDER (CIN , A , B , COUT , SUM **$** P1 **$** P2) ;

Dans ce cas on déclare deux paramètres génériques P1 , P2 . Ces paramètres peuvent être utilisés dans la partie définition des composants ou dans la partie description des connexions.

Exemple :

DECLARE INPUT CIN , A (0 : P1) , B (0 : P1) ;

.....

DECLARE COMPONENT FADD (0 : P1) : FULLADD ;

Les valeurs de ces paramètres sont déterminées : soit par l'instantiation du modèle en composant ; soit par l'environnement de simulation (cas du modèle principal).

Pour illustrer l'utilisation d'un modèle structurel paramètrable la figure (36) représente le schéma d'un additioneur 4 bits et la figure (37) représente la description FIDEL correspondante [EL 86].

En conclusion de cette partie on peut résumer les caractéristiques principales de la description structurelle en FIDEL :

- Représentation précise de l'information structurelle.
- Description homogène de tous les niveaux de la hiérarchie.
- Grande souplesse dans la définition des modèles primitifs.
- Paramètrisation de la description structurelle grâce aux paramètres de généricité .

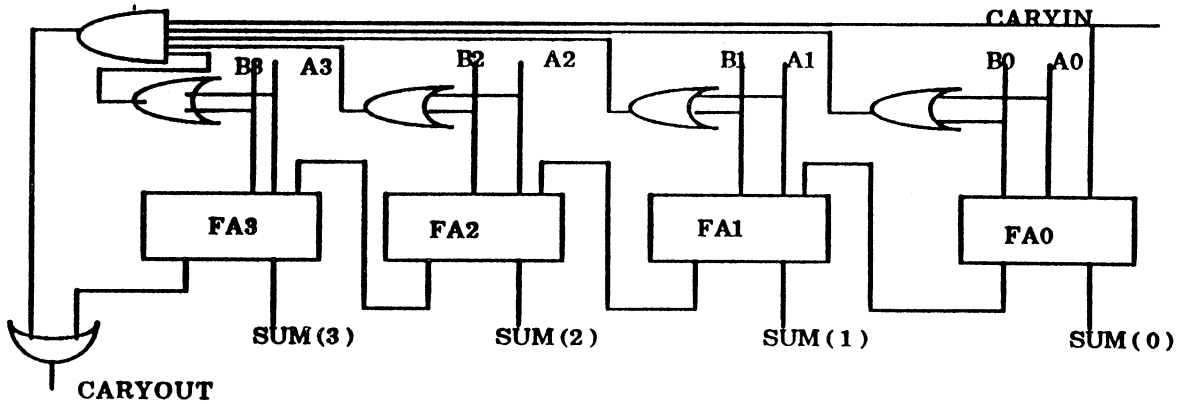


FIG. 36 :STRUCTURE D' UN ADDITIONNEUR 4 BITS

```
PFMODEL FIAND ( E , OUT $N1 ) ;

DECLARE INPUT  E ( 0 : N1 ) ;

DECLARE OUTPUT  OUT ;

DECLARE INTEGER I ;

INITIALIZE OUT TO 1 ;

FUNCTIONAL

WHEN E CHANGES
   FOR I = 0 UPTO N1 DO
      MAKE OUT = OUT AND E ( I )
   ENDFOR ;

ENDFUNCTIONAL
ENDMODEL
```

```
FMODEL TRIO ( E1 , E2 , S ) ;
DECLARE INPUT E1 , E2 ;
DECLARE OUTPUT S ;
FUNCTIONAL

WHEN  E1 CHANGES OR E2 CHANGES
      MAKE OUT = FLOR ( E1 , E2 ) ;

ENDFUNCTIONAL
ENDMODEL

FMODEL FULADD (CARIN , IN1, IN2 ,
                        OUT1 , OUT2 ) ;
DECLARE  INPUT CARIN , IN1 , IN2 ;
DECLARE OUTPUT OUT1 , OUT2 ;
FUNCTIONAL

WHEN  CARIN CHANGES OR IN1 CHANGES
         OR IN2 CHANGES
      MAKE OUT2 @ OUT1  =  CARIN + IN1 + IN2 ;

ENDFUNCTIONAL
ENDMODEL
```

```
SMODEL ADDER ( CARYIN , A , B , SUM , CARYOUT ) ;
DECLARE INPUT CARYIN , A ( 0 : 3 ) , B ( 0 : 3 ) ;
DECLARE OUTPUT  SUM ( 0 : 3 ) , CARYOUT ;
DECLARE  INTEGER  I ;
DECLARE  COMPONENT PAND : FIAND ( $ 4 ) , POR ( 0 : 4 ) : TIOR ;
DECLARE  COMPONENT FA ( 0 : 3 ) : FULLADD ;
STRUCTURAL

     FOR I = 0 UPTO 3 DO
        IF I ^= 0 THEN    CONNECT FA ( I - 1 ) . OUT2 , FA ( I ) . CARIN
        ENDIF
        CONNECT ADDER . A ( I ) , FA ( I ) . IN1 , POR ( I ) . E1
        CONNECT ADDER . B ( I ) , FA ( I ) . IN2 , POR ( I ) . E2
        CONNECT FA ( I ) . OUT1 , ADDER . SUM ( I )
        CONNECT POR ( I ) . S , PAND . E ( I )
     ENFOR
     CONNECT ADDER . CARYIN , FA ( 0 ) . CARIN
     CONNECT PAND . OUT , POR ( 4 ) . E2
     CONNECT FA ( 3 ) . OUT2 , POR ( 4 ) . E1
     CONNECT POR ( 4 ) . S , ADDER . CARYOUT

ENDSTRUCTURAL
ENDMODEL
```

FIG . 37 : DESCRIPTION STRUCTURELLE D' UN ADDITIONNEUR 4 BITS

4. L'ENVIRONNEMENT DE SIMULATION

Dans cette partie nous présentons l' environnement de simulation FIDEL , en commençant par la discussion du concept de simulation en FIDEL.

4.1 Concept de simulation en FIDEL :

La simulation en FIDEL tire partie de la méthode de conception hiérarchique et de la possibilité de description multi niveaux. Nous rappelons que les différents niveaux supportés par le simulateur sont le niveau fonctionnel, le niveau logique (représentation du modèle en terme d'opérateurs logiques) et le niveau interrupteur.

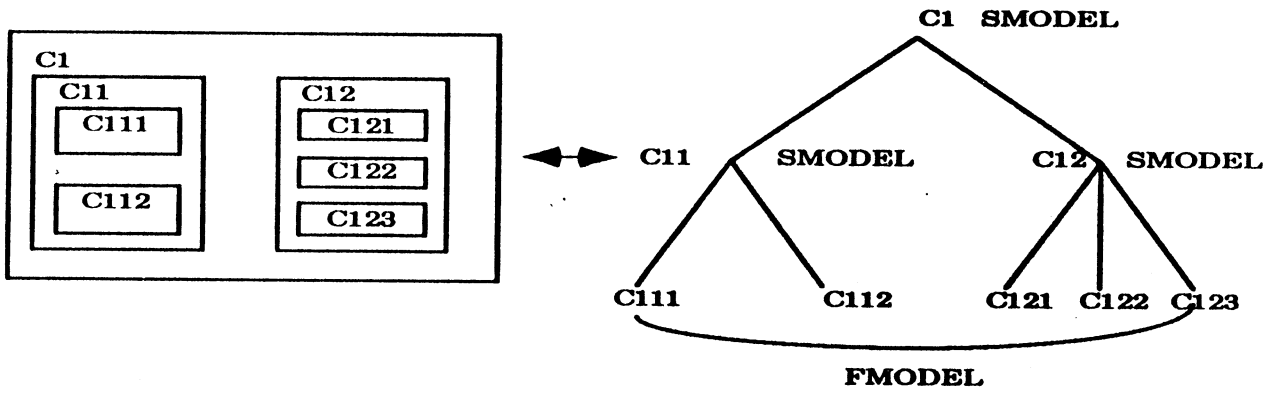


FIG. 38 : ARBORESCENCE HIERARCHIQUE

La hiérarchie de la description est conservée par la structure interne générée pour la simulation.Cela signifie, comme on l'a mentionné, que l'on ne fait pas d'éclatement de la description.

Cette description hiérarchique peut être présentée au niveau interne comme une arborescence (voir la figure (42)), où les feuilles sont des modèles fonctionnels primitifs ou bien des interrupteurs.

Pendant la simulation on parcourt cette arborescence selon les chemins d'activités pour déterminer les composants à activer. Cette approche est plus coûteuse en temps de calcul que les méthodes d' éclatement. Pour éliminer ce défaut, notre algorithme de simulation est implémenté de manière récursive. Les détails de cette implémentation sont présentés dans la prochaine partie.

Notre simulateur permet de traiter des circuits synchrones et asynchrones ou mixtes. Il est dirigé par événements avec la technique de "trace selective" . Il utilise cinq états ou valeurs de simulation. Les cinq valeurs sont **0 , 1 , Ø, X et Z** , où X représente l'état indéterminé , Ø l'état de transition (c'est dire de 1 à 0 ou de 0 à 1) et Z la haute impédance.

Le choix de la technique "dirigé par événement et trace selective" permet de réduire le temps de simulation : seuls les composants dont les entrées (au moins une) changent sont activés. Seules les sorties modifiées générent un événement à transmettre aux composants en aval.

4.2 La description du scénario de simulation (GMODEL):

Les rôles principaux de la description du scénario de simulation sont les suivants :

- Déterminer l'état initial du circuit.
- Générer les stimuli c'est à dire les différentes valeurs d'entrées à chaque pas de simulation.
- Tracer certaines variables pendant la simulation.
- Spécifier sur quelles conditions la simulation s'arrêtera (valeur maximale de temps de simulation, ou événement à détecter à la sortie).

Le modèle du contrôle de simulation consiste en trois parties :

- entête du modèle.
- définition des éléments .
- description des commandes de simulation.

L'entête du GMODEL contient seulement son nom. Ce nom doit être le même que celui du modèle principal (c'est à dire la racine de l'arborescence de la description). Il n'y a pas de liste d'interface ; les éléments d'interface sont ceux du modèle principal.

Exemple :

```
GMODEL ADDER ;
```

La partie définition contient les différentes séquences de stimuli et les valeurs d'horloge (signaux périodiques) à appliquer pendant la simulation ; on peut aussi définir des éléments locaux du type **STATE** et **INTEGER** utilisés pour l'affectation des variables d'entrées. La syntaxe de déclaration dans cette partie est la suivante :

global_declaration ::= "**DECLARE**" (local-declaration , stimuli_declaration) *

La déclaration locale pour le type **INTEGER** et **STATE** est la même que dans les modèles structurel et fonctionnel. La déclaration de stimulus est la suivante :

stimulus_declaration ::= clock_declaration , sequence_declaration *

clock_declaration : := "**CLOCK**" clock_name { " ," clock_name } *

clock_name ::= "ID " " (" period_val " ," zero_interval " ,"
one_interval " ," init_state ")" *

où les quatre paramètres sont des constantes de type "**DECIMAL** " .

Exemple :

DECLARE CLOCK H1 (100, 10, 10 ,1) ;

Cette déclaration spécifie un signal de durée 100 unités de temps et symétrique (le zéro et un ont chacun un intervalle de 10); la valeur initiale est à 1. Il faut remarquer ici que l'on n'a pas spécifié l'heure de début de ce signal, parce qu'elle est relative et que sa valeur sera déterminée dans les commandes de simulation où on applique ce signal. La variable H1 déclarée du type "CLOCK" ne peut pas être affectée.

La syntaxe de déclaration des séquences de stimulus est la suivante :

sequence_declaration ::= "**SEQUENCE**" seq_name { " ," seq_name } *

seq_name ::= "ID" seq_val ": " time_val { " ," seq_val " : " time_val } *

où les deux paramètres (seq_val et time_val) sont des constantes du type "**DECIMAL**". La séquence sert à donner un nom à une liste d'heures relatives croissantes. Cette séquence peut être ensuite appliquée à une connexion.

Exemple :

```
DECLARE SEQUENCE S1 ( 0:10 , 10: 15 , 5 :20 , 10 : 70 ) ;
```

Cette séquence contient quatre valeurs (0 , 10 , 5 et 10) qui sont appliquées aux heures relatives (10 , 15 , 20 et 70). Si l'on applique cette séquence dans la partie commande à une connexion à l'heure 20 de simulation, la séquence effectivement appliquée sera : valeur 0 à l'heure 30, valeur 10 à l'heure 35 , valeur 5 à l'heure 40 et enfin valeur 10 à l'heure 90.

La partie de description de commandes de simulation consiste en une ou plusieurs instructions "WHEN", comme dans le modèle fonctionnel , mais ici la condition d'instruction "WHEN" ne contient que la variable spéciale prédéfinie "TIME" qui indique l'heure de simulation.

Exemple :

```
WHEN TIME BECOMES 0
    MAKE A = S1 ;
```

Dans cet exemple on affecte la variable "A" du type "INPUT" (le type est vérifié pendant la compilation) par la valeur de S1 (selon la déclaration de S1 comme valeur, vecteur ou séquence de valeurs).

Le corps du GMODEL (partie commande) est le même que celui du modèle fonctionnel. Donc, on peut utiliser les différentes instructions du langage fonctionnel FIDEL (voir chapitre II).

On peut tracer les variables qui ne sont pas des variables primaires (entrée, sortie et entrée/sortie du modèle principal) de deux manières. La première est d'utiliser l'instruction "TRACE" dans la description fonctionnelle (voir chapitre II), la deuxième est d'utiliser l'instruction "SCHEDULE" dans les commandes de simulation.

Exemple :

```
SCHEDULE ADDER . E1 , FADD ( 1 ) . E1 , FADD ( 2 ) . E2
```

Dans cet exemple on demande la trace de la connexion E1 du composant ADDER, de la connexion E1 du premier composant du vecteur de composants FADD (on remarque ici que le nom de connexion n'est pas unique) et de la connexion E2 du deuxième composant du vecteur de composants FADD.

Remarque :

FIDEL n'offre pas la possibilité de présenter le chemin d'accés dans la commande SCHEDULE dans la version actuelle. C'est à dire qu'on ne peut pas visualiser une connexion interne d'un composant générique utilisé à un niveau quelconque dans la hiérarchie. Pendant la simulation on génére un message d'avertissement et on choisit la première occurrence dans l'arborescence.

Le simulateur va continuer l'exécution (simulation) jusqu'à l'absence d'événements à traiter ou bien jusqu'à la rencontre d'une condition d'arrêt. Cette condition peut être explicite (valeur maximale du temps de simulation), ou implicite (condition ou événement à détecter à la sortie pendant la simulation).

Exemple : arrêt sur dépassement du temps

```
WHEN TIME BECOMES 100
        EXIT;
```

Exemple : arrêt avec une condition

```
WHEN TIME BECOMES 40
.....
  IF SUM < 0 THEN EXIT ENDIF
  MAKE A = V1 ;

WHEN TIME BECOMES 50
.....
```

Dans cet exemple on teste à l'heure 40 de simulation la valeur de la sortie SUM, si elle est négative, alors on arrête la simulation, si non on continue l'exécution normale.

On remarque que le modèle de simulation utilise les mêmes instructions que celles du modèle fonctionnel. Un exemple complet sera présenté dans l'annexe E.

5. PROCESSUS DE SIMULATION

Cette partie sera consacrée à la discussion et la présentation de la structure interne de FIDEL (programmation, structure de données). Une grande partie de ce travail a été faite en coopération avec M. Cazal, pendant son projet de troisième année [CAZ 86].

Pendant l'implémentation du simulateur indépendant on a utilisé deux nouvelles propriétés qui le rende plus performant et plus adapté à l'utilisation en conception :

a) La première propriété est de conserver pendant la simulation la représentation hiérarchique de la structure. Cette conservation des hiérarchies a pour but de faciliter l'utilisation des modèles en bibliothèque : il est utile de pouvoir remplacer un modèle par un autre à l'intérieur d'une description, soit parce que l'on veut comparer les résultats des simulations, soit parce que l'on veut détailler certaines fonctions d'un circuit, dans l'application d'une méthode de conception descendante.

Grâce à la conservation des hiérarchies pour l'exécution , il n'est pas nécessaire , comme dans EPILOG ou ELDO, de "mettre à plat" l'ensemble d'une description après la moindre modification ou substitution. Il en résulte un gain de temps CPU entre deux modifications.

En contrepartie, l'ensemble des informations représentant la structure est plus volumineux et l'utilisation de ces informations ralentit l'exécution de la simulation.

Une analyse rapide de la rentabilité de la conservation des hiérarchies peut être effectuée : soit Ts , Ts' la durée de simulation respectivement avec et sans mise à plat pour une description donnée (Ts < Ts'), Tm est le temps de la mise à plat. Alors la mise à plat devient rentable lorsque le nombre N de simulation entre deux modifications devient tel que :

$N\,Ts' > N\,Ts + Tm$ soit : $N > Tm / (Ts' - Ts)$

En pratique, comme dans le cas de la simulation de défauts on effectue la simulation avec la mise à plat.

b) La deuxième propriété est de préserver un des avantages de la description fonctionnelle FIDEL par rapport à la description logique ou électrique : la possibilité de manipuler des entiers comme information de base, à la place du bit. Ainsi, pour les connexions entre les différents modèles, il est utile de ne pas systématiquement tout éclater au niveau du bit : par exemple, si une sortie d'un modèle qui est un vecteur de 8 bits est connectée directement à une entrée du même type d'un autre modèle, le passage d'information se fait au niveau de l'entier et non du bit, comme le montre la figure (39).

Cette propriété a pour but d'augmenter la rapidité d'exécution des simulations : l'information stockée en mémoire pour un modèle fonctionnel est toujours un entier , que ce soit pour un bit ou un vecteur de bits. La transmission directe de cette valeur évite le décodage-recodage des bits et peut se faire en une seule affectation. Cependant cet avantage disparaît dès qu'il y a éclatement des connexions.

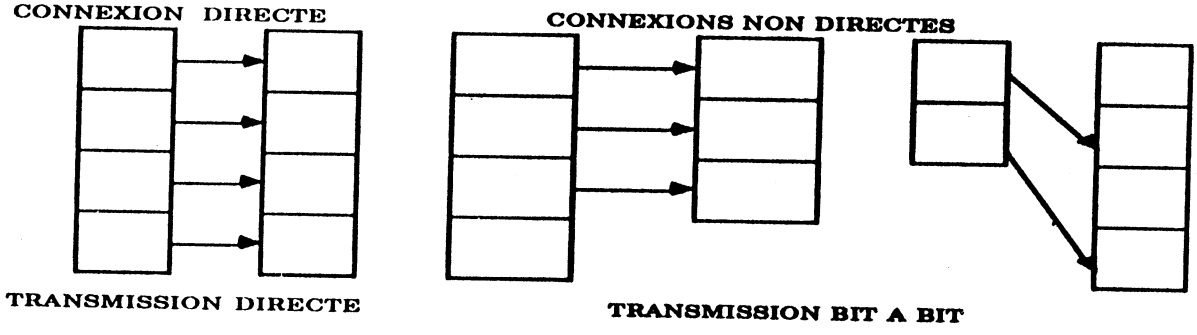


FIG . 39 : DIFFERENTS TYPES DE CONNEXION

5.1 Structure Générale :

La figure (40), montre la structure générale du système FIDEL. En commençant par le premier bloc, le concepteur décrit le système à vérifier en langage FIDEL (description fonctionnelle : **FMODEL**, description structurelle : **SMODEL** et description de commande de simulation : **GMODEL**). Cette description est fournie au compilateur FIDEL ; après l'analyse (lexicale, syntaxique et sémantique), le compilateur génère trois fichiers :

- le fichier du code interprétable ou exécutable (Executed Coded File ou ECF) pour chaque modèle fonctionnel.
- le fichier du code structurel (Structure Coded File ou SCF) pour chaque modèle structurel.

- le fichier du code de commande (Commande Coded File ou CCF) pour la description de scénario de simulation .

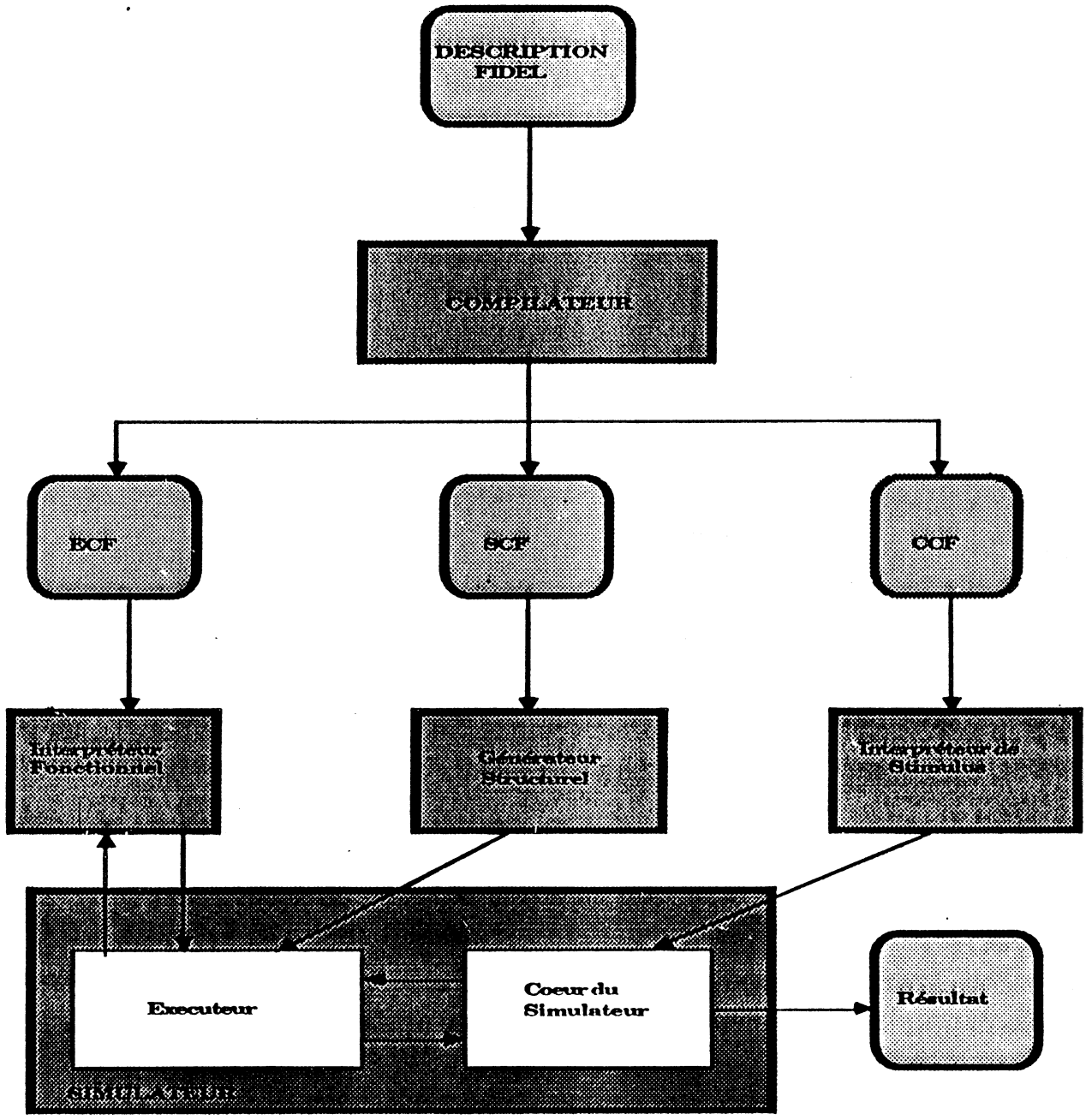


FIG. 40 : STRUCTURE GENERALE

Le SCF est interprété par le générateur de structure pour générer l'arborescence hiérarchique interne que l'on stocke et utilise pendant la phase de simulation. L'interprétation de CCF génère les commandes de simulation qui représentent les événements externes de simulation. Pendant la simulation on active l'interpréteur fonctionnel pour l'exécution de certains ECF.

Au chapitre II on a discuté les caractéristiques du code interprétable pour le modèle fonctionnel et le rôle de l'interpréteur fonctionnel en FIDEL. Dans cette partie on va discuter les principes du code structurel et du code pour le contrôle de simulation.

La définition de la structure de données représentant hiérarchiquement les modèles structurels est importante, car c'est de cette structure que dépendra l'efficacité de l'exécution des simulations. Il est pour cela nécessaire d'établir un compromis entre une redondance massive des informations (mise à plat du circuit) afin d'améliorer le temps d'exécution de la simulation, et l'absence complète de duplication afin de réduire les quantités d'espace mémoire nécessaire à la simulation sans trop ralentir la simulation.

Il est évident que le compilateur FIDEL est incapable de produire ces informations directement à partir de la description des modèles structurels, ne serait ce que parce qu'il n'opère qu'en une seule passe sur le programme source.

De plus, par souci de modularisation, il est intéressant d'établir une représentation intermédiaire de la description structurelle pour permettre des améliorations futures du compilateur de la manière la plus indépendante possible de la structure finale.

En outre la complexité de la génération de la structure finale impose une étape intermédiaire.

5.1.1 Représentation générale de l'information :

A partir d'ici, nous ferons la distinction entre les termes : un modèle (une description, qu' elle soit fonctionnelle ou structurelle) et un composant (une instantiation d'un modèle). Par exemple, on peut définir le modèle du demi-additionneur et utiliser deux composants de ce modèle pour définir un additionneur.

Le principe de toute la structure de représentation est de ne pas dupliquer inutilement l'information dans le cas où il existe plusieurs composants du même modèle. Pour cela un maximum d'informations est lié à chaque modèle (type du modèle, nombre d'entrées, nombre de sorties, nombre d'entrées/sorties,

nombre de paramètres, nombre de composants avec les emplacements des informations internes). Les seules informations qui sont reproduites au niveau de chaque composant ne sont que les informations qui apparaissent durant la simulation : il s'agit des valeurs des variables et des echéanciers internes des composants fonctionnels.

5.1.2 Représentation des connexions :

La représentation hiérarchique des connexions à l'intérieur d'un modèle structurel est réalisée à partir de deux tableaux internes : TABPRIM qui représente les connexions primaires (entrées sorties du modèle) et TABCONNECT qui représente les autres connexions liées à ce modèle. A chaque modèle correspond une zone de description des connexions dans ces tableaux.

Une connexion peut être détaillée au niveau le plus bas (bit) si nécessaire.

5.2 Le Simulateur :

L'exécution s'effectue en trois phases : phase de préparation de l'information, phase de simulation et phase de génération des résultats .

5.2.1 Phase de préparation :

Dans cette phase, on prépare toutes les informations et la structure interne dont on a besoin pendant la phase de simulation. Ceci inclut les opérations suivantes : interprétation du code structurel, allocation d'espace et initialisation, interprétation de commande de simulation et génération de liste d'événements.

(1) Interprétation du code structurel :
Dans cette opération on fait la transformation des données structurelles en une structure de données dynamique (basée sur la manipulation de liste et pointeurs). La vérification (sémantique) du code structurel et la génération de la structure hiérarchique dans une forme arborescente sont faites pendant cette opération.

(2) Allocation d'espace et initialisation :
Dans cette opération on fait l'allocation d'espace mémoire pour stocker les différentes tables et listes représentant la structure interne de la description en FIDEL. Cette structure interne inclut :

(a) Le tableau des connexions . Il y a deux types de connexion : directe et indirecte. La connexion directe signifie une connexion concernant la totalité des bits de deux variables de même taille . La connexion indirecte représente tous les autres cas.

(b) Le tableau de composants qui contient les informations liées au type de composant (FMODEL, SMODEL ou interrupteur): pointeur vers la zone de connexion, pointeurs vers le tableau de variables internes et le tableau de code à exécuter (dans le cas du modèle fonctionnel).

(c) L'échéancier interne de chaque composant fonctionnel s'il existe un retard dans la description, et la liste de composants en amont et en aval pour chaque fil de connexion.

(3) Création des commandes de simulation et de l'échéancier :
En FIDEL il y a deux types d'événements : les événements externes d'une part qui proviennent de l'utilisateur dans la description du scénario de simulation, et ne concernent que les entrées du modèle principal, les événements internes d'autre part qui proviennent d'un retard dans le modèle fonctionnel à la suite d'une exécution précédente (instruction **WITHIN** dans un **FMODEL**). Tous ces événement sont stockés dans l'échéancier global, à partir duquel on détermine l'activité pendant la simulation. L'interprétation du code de commande de la simulation génère le premier type d'événement ; pendant l'exécution on peut générer le deuxième type.

5.2.2 Phase de simulation :

Dans cette phase l'exécution du comportement du circuit a lieu. A chaque heure de simulation le coeur du simulateur active l'exécuteur (décrit plus bas) pour le modèle principal de la structure arborescente des modèles et pour chaque événement externe. En retour il reçoit la liste des sorties immédiatement modifiées ainsi que l'heure de réveil (prochain événement interne ou sortie modifiée avec

retard). La gestion des événements internes est transparente au coeur du simulateur, mais celui-ci doit explicitement activer le modèle principal à l'heure de réveil.

La présentation de l'algorithme d'exécuteur (fig.41) ne traite que le cas d'exécution sans délai. Nous présentons en détail les deux cas d'exécution (avec et sans délai).

```
PROCEDURE   EXECUTEUR   ( entrées : Composants, liste d'entées changées,
                          sorties : liste des sorties changées ) ;

Si type_modèle = Fonctionnel

    * Appeler l' interpréteur de modèles fonctionnel

    * Renvoyer la liste de sorties changées

Si type_modèle = Structurel

    * D' après les entrées changées, ETABLIR la liste des composants
      à exécuter avec , pour chacun d' eux , la liste des entrées changées

    * Tant que la liste des composants n' est pas vide faire :

        * Pour chaque composant de la liste faire :

            * EXECUTEUR (Composants, liste d'entrées changées,
                         liste des sorties changées) ;

            * ETABLIR d'après les sorties changées de
              ce composant, la liste des composants à réexécuter

        * La liste de composants à réexécuter
          devient la liste de composants

    * Renvoyer la liste des sorties changées
```

FIG. 41 : ALGORITHME RECURSIF DE SIMULATION

a) Cas de l'exécution sans délai :

Un composant est à exécuter à un instant donné parce qu'une ou plusieurs entrées sont changées à cet instant. Le résultat de cette exécution est une liste de sorties qui changent à la même heure puisqu'il n' y a pas de délai.

Dans cette méthode les entrées de plusieurs composants peuvent changer en même temps, mais on ne tiendra compte des conséquences de leur exécution qu'après les avoir tous exécutés : l'ordre d'exécution des composants d'une liste n'a donc aucune influence sur les résultats de la simulation.

b) Cas de l'exécution avec délai :
Le concepteur dispose pour les modèles fonctionnels d'une instruction "WITHIN" qui permet de retarder l'effet de changement d'une sortie ou d'une variable interne.

L'exécuteur gère la prise en compte des délais pendant l'exécution de la manière suivante :

- à chaque composant fonctionnel est alloué un échéancier interne qui est constitué d'une file des sorties retardées dans l'ordre croissant des heures de changement. Cet échéancier est tenu à jour par l'interpréteur des modèles fonctionnels, donc éventuellement modifié chaque fois que l'exécuteur active le composant fonctionnel.

- lorsqu'une sortie est retardée, son changement doit s' effectuer à l'heure indiquée dans l'échéancier. Pour pouvoir exécuter le composant fonctionnel à cette heure, l'exécuteur doit indiquer au coeur du simulateur qu'une exécution doit avoir lieu à cette heure : ceci est réalisé en indiquant au coeur du simulateur l'heure du prochain réveil.

- Dans l'algorithme d'exécuteur, lorsqu'on établit la liste des composants à exécuter, il faut rajouter les composants qui sont à réveiller, soit parce qu' ils sont fonctionnels et que la première heure de l'échéancier interne correspond à l'heure actuelle, soit parce qu'ils sont structurels et qu'un composant fonctionnel qu'ils contiennent vérifie cette condition.

5.2.3 Phase de génération des résultats :

C'est la phase finale dans le processus de la simulation. Dans cette phase le fichier du résultat (History File ou HF) que l'on génère pendant la phase de simulation est traité séquentiellement pour imprimer les valeurs désirées pour les entrées/sorties du circuit. Par défaut les variables d'entrées et de sorties qui changent de valeur vont être stockées dans ce fichier ; en plus s'il existe des variables internes ayant fait l'objet d'une commande de trace (soit dans la description fonctionnelle soit dans la description de commande de simulation), elles

vont être aussi stockées dans ce fichier. Le fichier HF contient tous les changements de valeurs des variables tracées, avec l'heure correspondante.

Le simulateur FIDEL est un programme d'environ 15 000 lignes écrit en langage PASCAL sur un calculateur VAX/VMS 785. Le simulateur est utilisé au CNET ; citons quelques exemples : RAM dynamique , une UAL pour un processeur de signal, un processeur auto-testable, le circuit ABC90 (arbitre de bus) et une étude spéciale de l'architecture d'un processeur de signal [BOU87]. La performance de ce simulateur vis à vis du simulateur mixte avec EPILOG est dans un rapport de 10 (plus rapide), et vis à vis d'autres systèmes sera présenté dans le prochain chapitre.

CONCLUSION

Le simulateur FIDEL [EL 86] a pour caractéristiques : une vérification temporelle précise, une structure hiérarchique, le mélange de la description de circuits synchrone et asynchrone, la description modulaire et l'utilisation d'un seul langage FIDEL pour les différents types de description (fonctionnelle, structurelle et environnement de simulation). Elles sont bien adaptées à une simulation hiérarchique multi niveau.

Comme les modèles communiquent seulement à travers leurs entêtes, si la description d'un modèle est changée tandis que son entête reste le même, la recompilation de tout le circuit n'est pas nécessaire, ce qui rend manifeste l'aspect hiérarchique.

De plus, si la liste d'entrées / sorties primaires pour le modèle principal (structurel ou fonctionnel) est la même pour tous les niveaux dans la hiérarchie de la description , on peut comparer facilement les résultats de simulation de différentes descriptions.

En conclusion, le système FIDEL (stand-alone) fournit au concepteur de circuits intégrés un langage de description et un outil de simulation souple, facilement abordable et utilisable.

CHAPITRE V

EVALUATION DE LANGAGES DE DESCRIPTION DU MATERIEL

INTRODUCTION

Dans la description d'un système logique deux types essentiels et différents d'informations peuvent être capturés : le comportement projeté du système et les détails de sa réalisation. La première demande un langage de description de comportement, tandis que la deuxième peut être exprimée par un langage de description structurelle.

De manière idéale, les informations de comportement et de structure doivent être exprimées en utilisant un seul langage. Ce langage unique est possible parce qu'un appel de fonction dans la description de comportement, dans lequel tous les arguments sont des signaux, correspond bien à une description structurelle aussi bien qu'à une description fonctionnelle.

On a présenté ces principes et les différentes propriétés du langage de description de matériel au chapitre I ; on a mentionné dans ce chapitre que les applications majeures des langages de description sont : la documentation, la vérification et la synthèse.

Tous ces aspects seront etudiés dans le présent chapitre, qui est une évaluation de langages de description. Le but de cette étude est :

- de dresser un bilan concernant l'état de l'art des langages de description de matériel.

- d'évaluer FIDEL vis à vis des autres langages.

Cette étude sera présentée en deux parties. La première partie sera consacrée à une étude et comparaison théoriques de certains langages de description par rapport à FIDEL. Cette partie est basée sur le travail de M.CAZAL dans son projet de D.E.A [CAZ 86].

La deuxième partie sera consacrée à une étude pratique de différents langages de description. La base de cette partie est le rapport [KJE 86] qui présente une évaluation et un "benchmark" dans le but de sélectionner un langage pour la vérification et la synthèse de système logique.

1. EVALUATION THEORIQUE

Le domaine de la description du matériel a donné lieu au développement de nombreux langages de description. C'est l'effort porté à la conception assistée par ordinateur des systèmes logiques qui en est à l'origine. Cependant, depuis les premières apparitions, au milieu des années 60, aucun langage standard ne s'est imposé sur le marché.

A travers les créations de nouveaux langages , on peut suivre l'évolution des concepts utilisés pour la description du matériel. Les premiers langages étaient des langages de description structurelle basés sur des éléments primitifs (portes au niveau logique) et donc ne s'intéressaient qu'au niveau logique. Puis l'émergence de LSI de plus en plus compliqués a provoqué l'apparition des langages de description fonctionnelle.

Dans cette partie, on a choisi un ensemble de langages récents, tous développés pendant les années 80 , et si une certaine similitude transparaît entre ces langages, elle provient des caractéristiques communes demandées à ces langages de description.

Par la suite, on donne la liste des langages étudiés avec leur origine (milieu purement universitaire ou industriel ou, comme c'est souvent le cas, les deux à la fois) :

- CADOC [CRA 85]
  Origine : IMAG - LCS (Laboratoire des Circuits et Systèmes)
  Date : 1983
  Le système CADOC est opérationnel , il a été développé dans le cadre d'un contrat DAII puis du projet européen CVT (CAD for VLSI circuits in Telecommunication).

- CASCADE [MER 85], [BOR 85]
  Origine : IMAG - ARTEMIS (Atelier de Recherche sur les TEchniques Mathématiques Informatiques des Systèmes).
  Date : 1980
  Le système CASCADE (Circuit And Systems Computer Aided Design & Engineering) a été developpé à l'université de Grenoble et puis dans le

cadre d'un projet européen de 1983 à 1985. Il est le résultat de plus de 15 ans de recherche dans le domaine de la CAO de circuits intégrés. Un premièr prototype est opérationel.

- ELLA [PRA 85]
  Origine : RSRE (Royal Signals and Radar Establishment) GB.
  Date : 1980.
  ELLA (Electronic Logic LAnguage) est actuellement commercialisé par la société PRAXIS SYSTEM qui a travaillé avec RSRE pour l'implémentation, ce qui a coûté 50 hommes-année.

- HILO 2 [FLA 81]
  Origine : BRUNNEL UNIVERSITY - GB
  Date : 1981.
  Ce langage est un des plus utilisés dans les milieux industriels (Vendu par la société GENRAD).

- IRENE [MAR 86]
  Origine : IMAG - TIM3
  Date : 1983
  Ce langage a été associé avec KARL (université de Kaiserslautern -RFA) pour donner le système KARENE, dans le cadre du projet CVT outil de synthèse de matériel.

- SISIM [SIG 85]
  Origine : SIGRAD
  Date : 1984.
  SISIM a été conçu par une équipe de 5 personnes et la société essaie actuellement de commercialiser ce produit.

- VHDL [LIP 86]
  Origine : DoD (Departement of Defence) EU.
  Date : 1983 .
  Le projet VHDL est pratiquement achevé et mobilise une équipe importante issue des sociétés Intermetrics, IBM et Texas Instrument. Il est financé par le DoD.

L'étude qui suit va inclure tous les langages sauf VHDL que l'on a présenté en détail au chapitre I (il sera cependant pris en compte dans la comparaison finale), et ELLA que l'on présentera en détail dans la deuxième partie de ce chapitre (étude pratique).

Les critères de base de cette étude sont :

1) La description structurelle.
2) Les types de données de communication.
3) La description du comportement .
4) La prise en compte du temps.
5) La prise en compte de la hiérarchie.
6) Le type de matériel concerné.
7) Les applications du langage.

1.1 Analyse des langages :

La liste précédente permet déjà de juger les langages selon l'étendue de leurs possibilités. Cependant, pour un langage conçu pour être utilisé dans le cadre industriel, il faut prendre en compte d'autres critères comme la facilité d'utilisation. En effet un langage possédant toutes les propriétés peut être utilisé avec difficulté parce qu'il est trop complexe.

1.1.1 CADOC : [CRA 85]

Dans le langage CADOC un circuit ou une partie de circuit peut être décrit de manière formelle en utilisant un modèle générique paramètré. Ce modèle est conçu comme Ressource Générique Fonctionnelle ou RGF (c'est l'équivalent d'un modèle paramètrable en FIDEL). Il existe aussi la notion de ressource algorithmique qui est la traduction immédiate de la notion de fonction dans les langages de programmation.

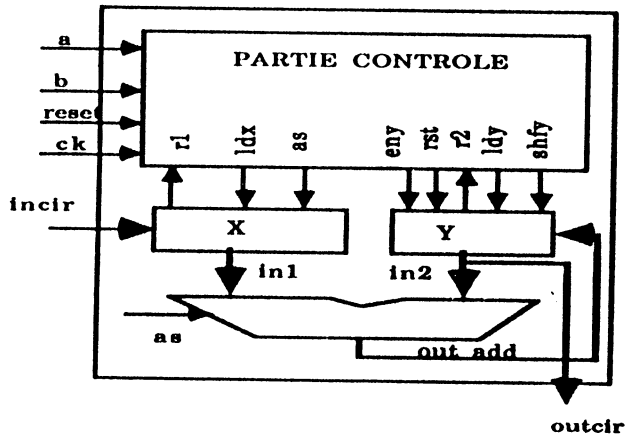
Description structurelle :

La description structurelle dans CADOC est assurée par la composition des RGF et utilise exclusivement des composants définis par l'utilisateur, sans primitives prédéfinies, comme le montre la figure (42).

Les composants ou occurrences de RGF sont déclarés dans l'entête d'une RGF composée, mais il n' y a pas de possibilité de décrire des matrices de composants.

Types de données de communication :

Toute variable utilisée dans une RGF doit être déclarée et typée (cette contrainte comme dans FIDEL permettant des vérifications de cohérence en particulier lors des connexions de différentes ressources).




```
RGF CIRCUIT ( a,b,reset,ck,incir,outcir) ;
TYPE
  int_8 = [0..255] ;
ENTREE
  incir : int_8 ;
 a , b , reset : BOOL ;
 ck  : FRONT ;
SORTIE
  outcir  : int_8 ;
VARINT
 r1 , r2 , ldx , ldy , rst , shfy, eny , as : BOOL ;
 out_add , in1 , in2 : int_8 ;
RGF
   ctrl_gen ( ck1 : FRONT ; reset, in1,in2,
              loadx,loady,shift,outy,
              adsb : BOOL ) ;
    srr_gen (load,shift,clear:BOOL;
             in : int_8 ; msb : ENTIER ;
             out : int_8 ; lsb : ENTIER ;
             PARAM n : ENTIER ) ;
    ual_gen(in1,in2 : int_8;op:BOOL;
            out : ENTIER  ) ;
    reg_gen(in,out : ENTIER ; load : BOOL;
            PARAM n : ENTIER) ;
RCONST
    control : ctrl_gen ( ck,reset,a,b,r1,r2,ldx,
                         ldy,rst,shfy,eny,as);
    x : reg_gen (incir,in1,ldx ; 8) ;
    y : srr_gen (ldy,shfy,rst,out_add,zero,
                 in2,- ;8) ;
    add_sub : ual_gen (in1,in2,as,out_add);
CONNECT
  in2 = outcir ;
```

FIG.42 : DESCRIPTION STRUCTURELLE EN CADOC

CADOC est fortement typé, tous les types de données possibles sont utilisables pour les communications avec l'extérieur. Les types de base sont : les booléens (0, 1, X, U, Z), les entiers (entiers relatifs, X, U, Z), les fronts (M : front montant, F : front descendant), et les types énumérés définis par l'utilisateur. Les types structurés sont : les enregistrements et les tableaux multi dimensionnés.

En plus du type et indépendamment de celui-ci, une donnée de communication doit appartenir à une des trois classes suivantes : ENTREE, SORTIE et BIDER (bidirectionnel). Deux autres classes VARINT et ALGO correspondent à des variables internes dans les descriptions structurelle et comportementale.

Description du comportement :

Le comportement d'un circuit feuille est décrit par l'intermédiaire d'un graphe interprété et temporisé ou GIT, dont la sémantique est proche du GRAFCET ou des Réseaux de Pétri.

Un GIT est de manière classique un graphe bi-partie dont les deux types de noeuds sont les places et les transitions, chaque noeud étant repéré par un identificateur. Aux places seront associées les actions ou les opérations de modification des variables définies dans la RGF et aux transitions seront associées des réceptivités (conditions) dont le franchissement permettra l'évolution du comportement en cours de simulation. La figure (43) montre la représentation graphique et textuelle d'un GIT.

CADOC offre une grande variété de types d'affectations et d'expressions des conditions. Les expressions construites en CADOC utilisent les variables et les constantes avec les opérateurs habituels et aussi avec des fonctions prédéfinies (FM, FD, CHANGE,TEMPO et STABLE) et des ressources algorithmiques qui sont des fonctions écrites par l'utilisateur dans un langage proche des langages de programmation (type Pascal).

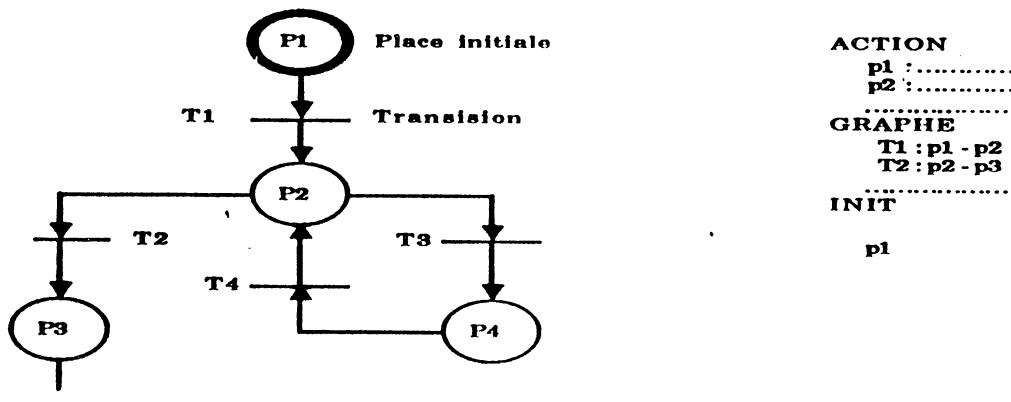


FIG. 43 : SQUELETTE D' UN GIT

Grâce au GIT, l'utilisateur de CADOC peut gérer le parallélisme dans sa description, aussi bien que dans un langage non procédural comportant des instructions gardées. La description du comportement est paramétrable et en plus l'utilisateur peut définir des assertions statiques pour contrôler les valeurs des paramètres et même des assertions dynamiques pour contrôler la bonne exécution d'un modèle pendant la simulation.

Prise en compte du temps :

Le temps en CADOC peut être vu à deux niveaux : externe comme une suite entière strictement croissante et d'incrément un, et interne comme un intervalle entre deux instants consécutifs absolus et subdivisé en un nombre de sous intervalles à priori illimité et géré à la simulation.

Il y a des fonctions qui facilitent la prise en compte du temps comme : CHANGE, TEMPO, DDA, STABLE et DOCX. De plus, la prise en compte du temps au niveau du comportement est basée sur l'utilisation de la notion d'affectation de chronogrammes, qui peut être vu comme une généralisation de la notion d'affectation avec retard. En effet, alors qu'une affectation avec retard est définie par le couple : nouvelle valeur - date de changement, une affectation avec chronogramme est définie par une suite finie de couples qui représentent une partie de l'histoire future de la variable (on peut remarquer l'analogie entre les séquences en FIDEL et les chronogrammes en CADOC).

Prise en compte de la hiérarchie :

CADOC offre beaucoup de souplesse pour la description du comportement, cependant le langage est moins adapté à la description structurelle .

Le langage permet le mélange des deux descriptions, ceci montre un objectif de ce langage qui est la description de circuits possédant une partie contrôle et une partie opérative. La partie contrôle étant représentée par un GIT et la partie opérative par des sous circuits, le tout dans une seule RGF.

En général, la description finale d'un circuit s'arrête à un niveau assez élevé de représentation. De plus les sous circuits doivent eux mêmes être décrits dans le cadre d'un GIT, même si l'on descend au niveau portes logiques.

Matériel concerné :

Bien sûr, il ressort de ci-dessus que CADOC ne s'adresse presque exclusivement qu'aux circuits séquentiels du type microprocesseur. Son utilisation sera lourde pour une architecture de type systolique (faiblesse de la description structurelle), encore plus pour les circuits combinatoires.

Applications :

CADOC n'a pas été seulement conçu pour la description et la simulation de matériel, mais aussi pour la synthèse du matériel, en particulier la génération automatique de la partie contrôle à partir d'un GIT. Il permet la génération de test fonctionnel.

En conclusion, CADOC est un exemple de langage typique pour un créneau bien particulier : la conception de circuits de type microprocesseur. Avec la richesse de ses types et les différentes fonctions le langage est bien adapté en conception à des niveaux élevés de description.

1.1.2 CASCADE :

Le langage CASCADE est présenté dans les rapports de recherches [MER 85] et [BOR 85]. Il s'agit de la juxtaposition de six langages qui correspondent chacun à un niveau de description mais partagent un noyau important des caractéristiques communes (approche de CONLAN, voir chapitre I).

Les langages sont : LASSO (niveau système), LASCAR et CASSANDRE (niveau transfert de registres), POLO (niveau logique), CASTOR (niveau interrupteur) et IMAG (niveau électrique).

Description structurelle :

Les sous langages issus de CASCADE possèdent des propriétés de description structurelle communes : les types de composants (non primitifs) sont déclarés dans l'entête de la description ; les instances de composant sont définies sous la même directive "USE" avec les paramètres qui décrivent les interconnexions.

Il faut noter que pour un sous langage donné les composants d'une description structurelle peuvent être décrits dans un autre sous langage. Seuls les sous langage de bas niveau utilisent des primitives, c'est à dire les portes logiques au niveau POLO et les interrupteurs au niveau CASTOR.

Certains langages, par exemple CASSANDRE permettent de simplifier la description de structures régulières par l'utilisation d'instructions de matriçage de connexions. De plus toute description en CASCADE est paramètrable.

Types de données de communication :

Il est évident que les types de données dépendent du niveau considéré et différent d'un sous langage à un autre. CASCADE est fortement typé.

Les types de base sont nombreux (INT, LOGICAL, BOOL, PULSE...) et propres à certains sous langages seulement. A partir de ces types de base l'utilisateur peut définir des tableaux multi dimensionnés. De plus on peut définir des types énumérés. CASCADE propose un environnement de description et de simulation en mode mixte où la transformation d'un type donné à un autre type est faite de manière automatique et vérifiable.

Malgré cette profusion de types, la communication entre deux descriptions de sous langages différents est assuré de la manière suivante :

- Soit parce que le type de donnée de communication existe dans les deux sous langages,
- Soit parce qu'il existe des compatibilités entre types prises en compte lors de la simulation,
- Soit parce que l'utilisateur donne lui même les fonctions de compatibilité au simulateur.

En général, la direction de communication est spécifiée de manière indépendante du type dans l'interface pour tous les sous langages parmi les attributs suivants : IN (entrée), OUT (sortie), INOUT (bidirectionnel) et ND (Non Directionnel).

Description du comportement :

Tout d'abord, les sous langages POLO et CASTOR ne possèdent pas de description du comportement, ils utilisent des primitives structurelles pour les feuilles de la hiérarchie.

Les règles générales de la description de comportement en CASCADE sont les suivantes :

- L'ordre lexical des instructions n'a pas d'importance, l'ordre d'exécution des instructions n'étant pas séquentiel (sauf en LASSO).

- Les instructions de base sont des modifications de valeurs de variables, et des appels de procédures.
- Ces instructions peuvent être contrôlées par les instructions classiques (IF, CASE ...).

Le langage LASSO est procédural, les blocs d'instructions étant gardés ; il permet de décrire n'importe quel graphe de contrôle et en plus possède des primitives de décision telles que TEST, INDEX qui peuvent sélectionner des instructions. Les variables traitées sont abstraites, ce qui oriente LASSO vers le niveau système.

Les deux langages LASCAR et CASSANDRE sont du niveau transfert de registres, les instructions sont des instructions de connexion entre les signaux d'interface ou internes de type registre (LATCH, REGISTER) contrôlables sur des niveaux ou sur des fronts d'horloge.

Le langage IMAG représente le niveau électrique. L'utilisation de ce langage introduit la notion de mode mixte en CASCADE. De plus, il existe deux langages pour décrire les spécifications temporelles, TPDL et les commandes de simulation IWDL [CAB 85]. A ce stade, il faut remarquer la séparation du langage de commande et l'interactivité du simulateur.

Il faut remarquer que l'on peut présenter la description de la partie contrôle sous forme de listes d'instructions étiquetées par chaque état d'un automate. Un seul état de l'automate est actif à un instant donné. Dans la description du comportement l'utilisateur peut utiliser les assertions pour contrôler les valeurs et les actions pendant la simulation.

La prise en compte du temps :

La prise en compte du temps dans POLO et CASTOR est basée sur l'utilisation d'attributs du retard (typique, min, max). Dans LASCAR et CASSANDRE le temps est pris en compte sous forme de délais dont la valeur est choisie en fonction de l'unité de base ou pas élémentaire.

Dans LASSO, le délai s'applique à toutes les instructions gardées à la fois ; il s'applique à chaque instruction de connexion de manière indépendante dans LASCAR et CASSANDRE. De plus, il existe une possibilité d'affectation de

chronogramme par le langage de commande IWDL.

Prise en compte de la hiérarchie :

L'utilisation d'un sous langage par niveau peut paraître à priori contraignante pour le concepteur. Mais pour la description d'un système matériel, il est non seulement possible de mélanger les différents niveaux en utilisant les sous langages (aspect multi niveau de CASCADE), mais il est possible de mélanger la description de structure et de comportement dans une seule unité, ceci pour tous les sous langages, sauf POLO et CASTOR bien sûr.

L'utilisateur peut jouer sur ce degré de liberté dans le processus de la conception. De plus le fort typage des données lui permet d'orienter le sous langage vers un niveau donné. On peut remarquer que les possibilités de description de bas niveau sont les plus rigides (sauf IMAG).

Matériel concerné :

La multiplicité des sous langages fait en sorte que tous les types de matériel peuvent être décrits en CASCADE.

Applications :

CASCADE est un outil de simulation multi niveau. Les sous langages sont aussi utilisés pour la simulation de défauts ainsi que pour la génération de vecteurs du test et la synthèse logique.

1.1.3 HILO II :

Les concepteurs de ce langage proposent une version multi niveau par extension d'une version antérieure qui est mono niveau : HILO logique.

Description structurelle :

Un circuit peut être décrit en HILO comme une composition hiérarchique d'éléments qui sont des objets prédéfinis et/ou des fonctions. Les communications entre les unités sont décrites par des variables de connexion et/ou par l'intermédiaire d'événements.

Une description structurelle est donc une liste d'éléments interconnectés, préalablement déclarés et pouvant être d'un des trois types suivants : un circuit, une porte ou une fonction. Les primitives de base sont les portes logiques (AND, OR,...) et même les interrupteurs (TRANIF0).

HILO ne prévoit pas de facilités pour les structures régulières et les seuls paramètres possibles concernent l'instantiation d'éléments primitifs ou fonctionnels.

Types de données de communication :

Les communications ont lieu par l'intermédiaire de fils de connexion. Les fils sont obligatoirement typés. Ils peuvent prendre un des types prédéfinis : INPUT, OUTPUT, UNIDIRECTIONNEL, TRI-state, SUPPLY... Les valeurs possibles sont : 0 , 1 , X , et Z, selon les types.

L'événement est un autre moyen de communication qui ne correspond pas à une réalité physique.

Description du comportement :

La description fonctionnelle d'un circuit en HILO utilise des objets plus complexes qui sont : des événements déclenchant le calcul des nouveaux états des variables et des registres activés par des opérateurs prédéfinis.

La structure d'une description du comportement en HILO est celle d'un langage non procédural : celle-ci se présente comme une liste d'instructions gardées, la garde étant un événement.

WHEN (événement) DO (actions)

La possibilité d'expression d'événement est très large en HILO, l'événement peut être provoqué par le changement d'une entrée ou activé grâce à l'instruction EVENT. Une expression d'événement est une expression portant sur un enchaînement ou une répétition d'événements en tenant compte éventuellement d'événements prioritaires :

WHEN 12 * (CKA THEN CKB) THEN (EVA OR
FLAG (0,X TO 1)) RESET EVX DO.....

Ceci représente une occurrence (12 fois) d'un événement (deux horloges sucessives) suivi par un autre événement (l'union entre l'événement nommé EVA et le changement de FLAG à la valeur 1) et la condition de reset de l'événement nomé EVX.

HILO, n'admet pas d'assertions utilisateurs mais une description du comportement est paramètrable, les paramètres portant sur les données temporelles.

Prise en compte du temps :

Le temps est pris en compte dans la description du comportement par l'instruction WAIT, qui peut retarder l'exécution de chaque instruction gardée :

WHEN (événement) WAIT 10 DO (actions)

L'action ou la liste d'actions ne sera exécutée que 10 unité du temps après la validation de l'événement.

Deux fonctions booléennes STEADY et WIDTH, facilitent aussi la prise en compte du temps en permettant de contrôler la stabilité des signaux entre les changements de valeurs ou d'événements. Ces fonctions permettent aussi de spécifier les temps d'établissement et de maintien (setup et hold).

Au niveau des portes logiques on peut associer des spécifications temporelles (min-max, typique et pire cas). Il faut noter l'existence de la notion d'affectation de chronogramme.

Il faut noter que la temporisation dans la description du comportement est utilisée pour spécifier l'instant d'exécution de l'action mais qu'il est impossible d'affecter une durée à l'action elle-même.

Prise en compte de la hiérarchie:

HILO permet la décomposition modulaire hiérarchique et multi niveau d'une description.

Les différents types prédéfinis du langage, tant dans la description de structure (portes logiques), que dans la description de comportement (registre, RAM) orientent l'utilisateur vers certains types de description. Dans le processus de conception, l'utilisateur passe successivement par des étapes imposées par le langage.

Le niveau fonctionnel d'HILO, compte tenu des types de données et du nombre restreint d'instructions de contrôle (IF et CASE), parait plus proche d'un langage de transfert de registre. Au niveau logique (et même interrupteur), c'est la description de structure qui est facilitée.

HILO permet la séparation entre la partie contrôle et la partie opérative d'un circuit. Le graphe de contrôle (réseau de Petri par exemple) peut être traduit facilement en description HILO par un enchaînement d'instructions gardées. Les communications entre la partie contrôle et la partie opérative peuvent se limiter à des événements.

Matériel concerné :

Grâce à son mécanisme de communication par événement, HILO permet la description de systèmes séquentiels asynchrones. Cependant, la définition externe d'une horloge le rend tout à fait adapté à la description de circuits synchrones.

HILO permet la description de tous les types d'architecture mais de manière moins lisible pour les architectures systoliques que pour les architectures classiques. HILO décrit facilement les circuits combinatoires, surtout au niveau logique.

Applications :

HILO est surtout orienté vers la simulation, le grand nombre de primitives disponibles a pour but d'augmenter la performance de simulation. Il faut

remarquer que HILO est beaucoup utilisé pour la génération de vecteurs de test et la simulation de défauts.

En conclusion, la succès de HILO, qui est considéré comme le standard actuel sur le marché des langages de description, montre qu'il répond aux besoins de la CAO de circuits intégrés tout en restant à la portée de ses utilisateurs.

Cependant, l'apparition de nouveaux concepts dans les langages de description de matériel, plus orientés vers les descriptions de haut niveau et les applications de synthèse logique, risque de réduire sa prépondérance actuelle.

1.1.4. IRENE :

Le langage IRENE est en fait la juxtaposition de deux langages : IRENE-S pour la description structurelle et IRENE-C pour la description comportementale.

Description structurelle :

Le langage IRENE-S permet de décrire une hiérarchie d'unités. Il n'existe pas d'éléments prédéfinis dans le langage. Les feuilles de la hiérarchie sont décrites uniquement par une liste d'instructions dont l'ordre n'est pas important. Les instructions représentent des objets ayant une réalisation matérielle.

Les variables manipulées dans cette description sont appelées physiques, elles ont plusieurs types possibles. Les variables physiques peuvent être mémorisées (REG, LATCH, RAM) ou non mémorisées (INST : fils de connexion). Les constantes physiques sont ROM et PLA .

Les opérateurs du langage sont des opérateurs classiques logiques, arithmétiques, de comparaison et de concaténation prenant en compte les dimensions des opérandes et du résultat.

Les instructions sont soit l'opérateur de chargement d'un registre (connexion uni directionnelle de registre avec des expressions logiques), soit l'opérateur de connexion bidirectionnelle, ou un constructeur de forme complexe tel que IF, CASE..OF (qui représente un multiplexeur), CASE..OF à gauche (qui représente un démultiplexeur), ces instructions peuvent être imbriquées. La figure (44), montre la

sémantique matérielle du IF .. THEN.. ELSE. Ces listes d'instructions peuvent apparaître à n'importe quel niveau de la hiérarchie.

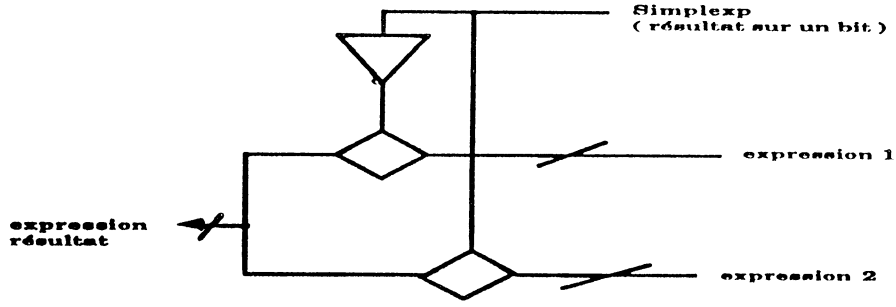


FIG. 44 : SEMANTIQUE MATERIEL D' UN IF-THEN-ELSE
(REALISATION NMOS)

Il existe d'autre part des instructions de boucle (FOR, REPEAT) qui peuvent contrôler un groupe d'instructions en utilisant des variables algorithmiques. L'utilisation d'une variable algorithmique facilite la description des structures régulières. Ces variables peuvent être passées en paramètres.

Types de données de communication :

Les communications se font par l'intermédiaire de variables physiques dont les types ont été définis ; celles-ci sont considérées comme des fils. L'ensemble des valeurs qu'elles peuvent stocker est toujours (0, 1 et U), où U est la valeur indéfinie.

Dans l'interface de communication on classe en plus les variables selon trois types : ENTREES, SORTIES et BIDIRECTIONNELLES. Ceci impose des restrictions dans l'utilisation des variables pour l'affectation. Les ordres ACTIVATE et la définition de séquencement sont aussi des moyens de communication.

Description du comportement :

La description en IRENE-C consiste en une hiérarchie de modules, un module décrivant le comportement d'une partie du système. La notion d'instantiation d'un module n'existe pas en IRENE-C. Un module n'est pas un modèle générique, il décrit une entité du système coopérant avec les autres modules et/ou unités structurelles (en cas de description mixte).

Le langage IRENE-C est proche du matériel car il utilise les même instructions et types prédéfinis que le langage IRENE-S qui sont orientés vers le niveau transfert de registre. Il y a trois types de description : une description qui manipule les

variables physiques au niveau transfert de registres (proche de la description de structure), une description qui manipule des variables de comportement grâce aux mêmes instructions complexes (IF,CASE,...) mais interprétées ici au niveau du comportement et une description particulière basée sur les ordres ACTIVATE pour représenter les commandes d'une partie contrôle.

Le langage comportemental peut être qualifié de multi niveau avec possibilité de description au niveau algorithmique et transfert de registres, mais le niveau logique ne peut être considéré ici que comme une version du niveau transfert de registres.

Prise en compte de temps :

Le temps est représenté dans IRENE par une suite d'unités de base ou pas élémentaires. L'originalité d'IRENE est l'emploi de séquencement à la place de délais ou retards.

La définition d'une séquence (nombres et noms de phases ainsi que le type de séquence) se fait dans l'entête de module de IRENE-C, avec les possibilités d'imbrication de séquences soit explicitement à l'intérieur d'un seul module, soit à travers l'imbrication des modules. Par exemple :

SEQUENCE auto1 /2/ : $phi1 , $phi2 ; GENERAL ENDSEQ

Une séquence décrit un automate avec ses états ; la durée des états ou des phases est égale à un multiple du pas élémentaire correspondant à l'unité de la séquence la plus fine. Il est aussi possible de décrire une horloge en utilisant des séquences.

Pour exprimer les changements par rapport à l'ordre normal d'exécution, il existe des instructions de séquencement GOTO, REPEAT..UNTIL, FOR pour les séquences de type GENERAL. L'exécution des instructions se fait de manière synchrone, mais il existe en plus une possibilité de prise en compte des signaux asynchrones pour modéliser le RESET.

Prise en compte de la hiérarchie:

IRENE permet la conception dans une approche descendante selon les étapes suivantes :

- Définition de comportement au niveau algorithmique (IRENE-C).
- Remplacement des variables algorithmiques par des variables physiques du niveau transfert de registres tout en précisant le séquencement.
- Séparation de la partie contrôle décrite en IRENE-C sous la forme d'un automate de séquencement et de la partie opérative décrite en IRENE-S.

Que ce soit dans IRENE-C ou IRENE-S, le concepteur peut décomposer sa description en utilisant des hiérarchies de modules. Il faut noter qu'au niveau interne on éclate la description avant de la simuler (comme dans CADOC).

Cependant la définition des séquences ainsi que les types prédéfinis au niveau transfert de registres rendent le langage IRENE extrêmement rigide à l'utilisation.

Matériel concerné :

IRENE est le langage le plus restrictif pour les types de matériel pouvant être décrits. En effet, la notion de séquencement dans IRENE-C ne peut s'appliquer qu'à des circuits séquentiels synchrones du type microprocesseur, décomposables en partie contrôle et partie opérative. Il n'est pas question de l'utiliser pour d'autres types de matériel.

Applications:

Comme CADOC, IRENE n'est pas prévu seulement pour la description et la simulation de circuits intégrés, mais aussi pour la synthèse de la partie contrôle décrite en IRENE-C et de la partie opérative décrite en IRENE-S.

En conclusion, IRENE et CADOC sont des langages très comparables quand à leur objectif. Mais IRENE est plus lié à un type de matériel, ce qui limite son utilisation.

1.1.5 SISIM :

Les informations obtenues sur SISIM étant succinctes et très générales, on va essayer de présenter ici tout ce qui peut en être retiré.

Description structurelle :

SISIM comporte quatre niveaux de description structurelle :

- Une description globale qui se veut la réplique exacte du système matériel en utilisant des équipotentielles (broches) et des composants. Cette description n'est pas hiérarchique.
- Une description à l'intérieur d'un composant qui peut être hiérarchique. Un composant peut être décrit en effet par : un modèle de comportement ou un assemblage de composants.
- Une description au niveau logique qui est traitée à part et qui utilise soit des portes logiques pré-existantes (g_structure) soit des éléments pré-existants dans une bibliothèque (b_structure).
- Une description au niveau d'interrupteur (m_structure), elle aussi traitée à part et utilisant comme primitives des transistors MOS discrétisés ou interrupteurs.

Types de données de communication :

Les composants communiquent entre eux uniquement par les événements véhiculés par les fils.

Description du comportement :

Une caractéristique de SISIM est d'utiliser directement le langage C pour la description du comportement d'un composant. Ce type de modèle est appelé b_modèle.

Prise en compte du temps :

La prise en compte du temps dans SISIM se fait par la définition de délais entre les entrées et les sorties d'un bloc. Une caractéristique de SISIM est de séparer complètement le comportement et le temps. Alors que le comportement est décrit par des b_modèles en langage C, les temps de propagation sont décrits par des p_modèles entre les différentes broches d'un composant.

De même au niveau logique ou interrupteur, pour prendre en compte le temps on associe de manière indépendante à ces description des p_modèles.

Prise en compte de la hiérarchie:

Il y a trois niveaux de description : niveau comportemental (en langage C), niveau logique (description structurelle) et niveau interrupteur (description structurelle). Ces trois niveaux peuvent être mélangés pour décrire de manière hiérarchique et multi niveau le matériel.

Mais, autant au niveau algorithmique l'utilisateur peut profiter de toute la souplesse de description offerte par le langage C, autant au niveau logique et interrupteur le choix est limité par l'utilisation de primitives. Il semble qu'il existe un décalage entre le premier et les deux autres niveaux qui peut gêner le concepteur, ceci étant dû à l'absence d'un niveau transfert de registres. Cependant il semble exister une bibliothèque de cellules paramètrables (registres, aditionneurs...).

Une autre limitation de SISIM semble être aussi la pauvreté du système de communication entre les composants, qui est réduit à des équipotentielles même aux niveaux les plus élevés de description.

Matériel concerné :

Les informations disponibles ne permettent pas de discerner les types de matériel plus favorables à SISIM, les auteurs de SISIM déclarent que ce langage permet la simulation de tous systèmes électroniques à partir de la carte jusqu'aux circuits VLSI de grande complexité.

Application :

SISIM est orienté vers la spécification et la simulation. Le mécanisme de simulation est différent selon le type du modèle. Mais la nature compilée de SISIM implique probablement une perte d'efficacité par rapport à un simulateur logique classique en particulier pour les circuits logiques ayant peu d'activités simultanées. De ce fait SISIM apparaît comme un simulateur de comportement permettant des descriptions logiques, mais à un coût de simulation plus élevé si l'on veut simuler un gros circuit en logique.

En conclusion, SISIM se présente comme une utilisation du langage C pour la description du matériel.

L'utilisation d'un langage non spécifique, une certaine connaissance en programmation exigée pour le concepteur vont peut être freiner, dans la pratique, l'utilisation de SISIM.

1.2. Essai de comparaison avec FIDEL :

Le but n'est pas ici de prouver que FIDEL réalise la synthèse des langages présentés précédemment en en sélectionnant les aspects favorables. Il s'agit plus de classer les différents langages selon leurs tendances et propriétés afin de mieux situer FIDEL dans cet ensemble.

Pour cela, nous allons nous intéresser à quelques caractéristiques déjà étudiées.

1.2.1 La gamme de niveaux du langage :

Tous les langages étudiés précédemment sont multi niveaux mais cette notion varie d'un langage à un autre. Nous essayons de préciser la véritable portée de cette propriété en présentant dans la figure (45) le degré d'adaptation de chaque langage à chaque niveau, ainsi que la possibilité de décrire un bloc à un niveau donné autrement que par l'interconnexion de primitives .

Le schéma permet d'évaluer la continuité de description du langage, c'est à dire la facilité pour passer d'un niveau à l'autre dans l'optique d'une conception descendante.

VHDL, FIDEL et CADOC y sont bien adaptés, ainsi que CASCADE mais cela résulte dans ce dernier cas d'un changement de langage. SISIM est un bon exemple de langage discontinu, ce qui provient de l'existence de primitives structurelles de bas niveau à coté d'un langage classique de haut niveau. HILO et IRENE se présentent plus comme des langages avec un niveau privilégié mais qu'on peut utiliser à d'autres niveaux.

Niveaux Langages	Algorithme	Fonctionnel	RTL	Logique	Switch
CADOC	* (1)	*	*	* (1)	
CASCADE (3)	*	*	*	* (2)	* (2)
FIDEL	* (1)	*	*	*	* (2)
HILO	* (1)	*	*	* (2)	* (2)
IRENE	* (1)	*	*	* (1)	
SISIM	*	* (1)	* (1)	* (2)	* (2)
VHDL	*	*	*	* (1)	

(1) : la possibilité existe, mais le langage n' est pas particulièrement adapté

(2) : en utilisant des primitives

(3) : CASCADE le seul langage qui offre le niveau électrique (IMAG).

FIG. 45 : POSSIBILITES DE DESCRIPTION DES LANGAGES SELON LES DIFFERENTS NIVEAUX

1.2.2 La facilité de la description structurelle :

Tous les langages admettent la construction structurelle hiérarchique indépendamment du niveau de description, mais à un degré d'efficacité variable. Deux propriétés essentielles sont liées entre elles : la possibilité de condenser les descriptions structurelles régulières et la possibilité de paramètrer ces descriptions. Elles permettent de répartir les langages en deux catégories. A l'intérieur de chaque catégorie les langages sont aussi classés selon l'étendue des possibilités offertes (figure 46).

IRENE-S et IRENE-C, sont présentés séparement parce que la décomposition modulaire hiérarchique de IRENE-C a une description structurelle très rustique car il n'y a pas de notion d'instance d'unités.

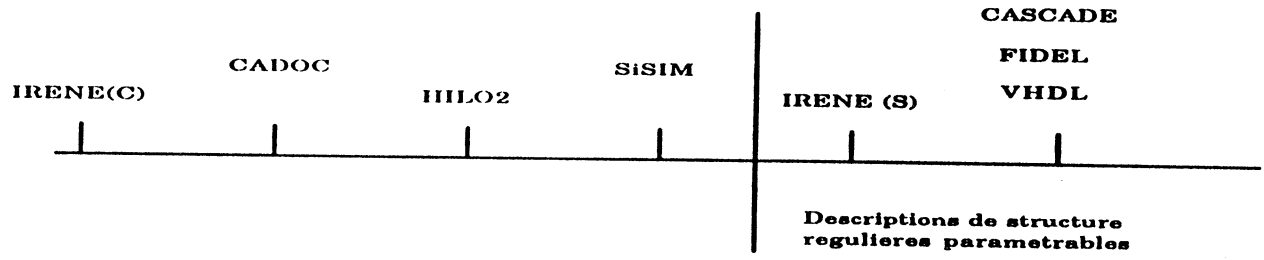


FIG. 46 : POSSIBILITE DE DESCRIPTION STRUCTURELLE

Cette distinction n'apparaît pas pour CASCADE car les unités de description sont toutes compatibles, indépendamment du sous langage.

On peut conclure en affirmant que FIDEL fait partie avec CASCADE et VHDL des langages qui offrent les possibilités souhaitables pour la description structurelle dans le cadre d'un langage multi niveau.

1.2.3 Type de description du comportement :

Dans un premier temps on va essayer de classer les langages selon le degré de ressemblance à un langage de programmation type ADA, C et PASCAL.

La comparaison entre les langages de description du matériel est faite de manière empirique au vu d'exemples de description, sauf pour les langages qui dérivent plus ou moins directement d'un langage de programmation. Le résultat de cette comparaison est donnée sous la forme d'une échelle, comme le montre la figure (47).

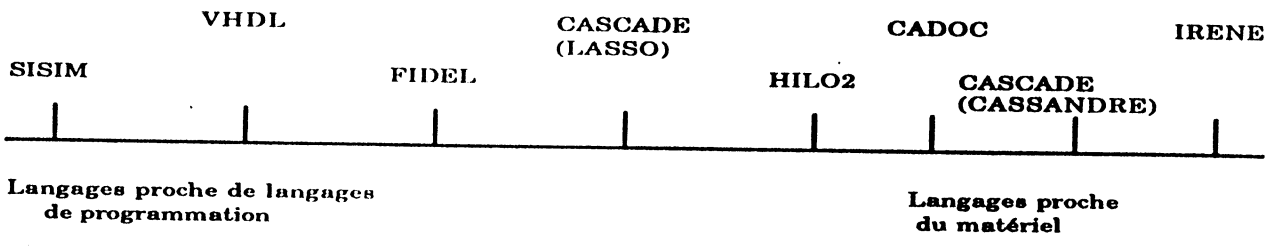


FIG.47 : Classement des langages selon le type de description comportementale

SISIM est ici le seul langage qui impose d'utiliser directement un langage de programmation, le langage C. L'utilisation directe du langage ADA, favorisée par sa richesse (parallélisme, mécanisme de compilation séparée, bien adapté à la gestion de bibliothèque...), est actuellement sujet à discussion [BAR 84] et [SUM 85].

VHDL en est pourtant dérivé et avec une syntaxe propre et quelques instructions spécifiques (pour la prise en compte du temps par exemple), il conserve les avantages d'ADA. De même, PASCAL a son langage dérivé HELIX que l'on va présenter dans la deuxième partie de ce chapitre.

La polémique qui existe sur l'utilisation directe ou dérivée des langages classiques peut trouver un compromis : en effet, pour la simulation il semble de plus en plus indispensable d'utiliser un "coeur de simulateur" : ceci permet que le mécanisme très lourd de la simulation et la prise en compte du temps n'apparaissent pas dans la description.

Derrière VHDL, on trouve CASCADE (pour LASSO), FIDEL et HILO qui ont un mécanisme analogue de description à base d'instruction gardées puis CADOC avec sa description à base de graphe mais qui offre les mêmes possibilités que les langages précédents. Enfin IRENE auquel on peut associer CASCADE (pour CASSANDRE et LASCAR) dont les descriptions sont à base de séquencement.

A l'opposé des langages proches de langages de programmation on rencontre des langages plus proches mais aussi plus dépendants du matériel si l'on se réfère à la gamme de matériel concerné. On peut alors établir un parallèle entre les différentes propriétés de ces langages, comme le montre la figure (48).

Evidemment, ce classement ne repose que sur la tendance principale du langage. Dans tous ces langages des mécanismes ont été rajoutés pour combler les lacunes et multipler les possibilités, parfois en contraste avec le style de base de la description.

Par exemple, une méthode très utilisée (dans CASCADE, HILO et SISIM) consiste à définir des primitives de description structurelle au comportement prédéfini . De même dans des langages proche des langages de programmation (HELIX), des types d'objets manipulés peuvent être prédéfinis (registre, bascule...). Inversement, il est possible d'utiliser des fonctions (CADOC et IRENE) écrites avec des instructions algorithmiques classiques pour étendre les possibilités du langage.

Certains langages initialement non procéduraux (VHDL, FIDEL) possèdent des instructions ou des directives permettant d'exprimer l'enchaînement des instructions de manière séquentielle.

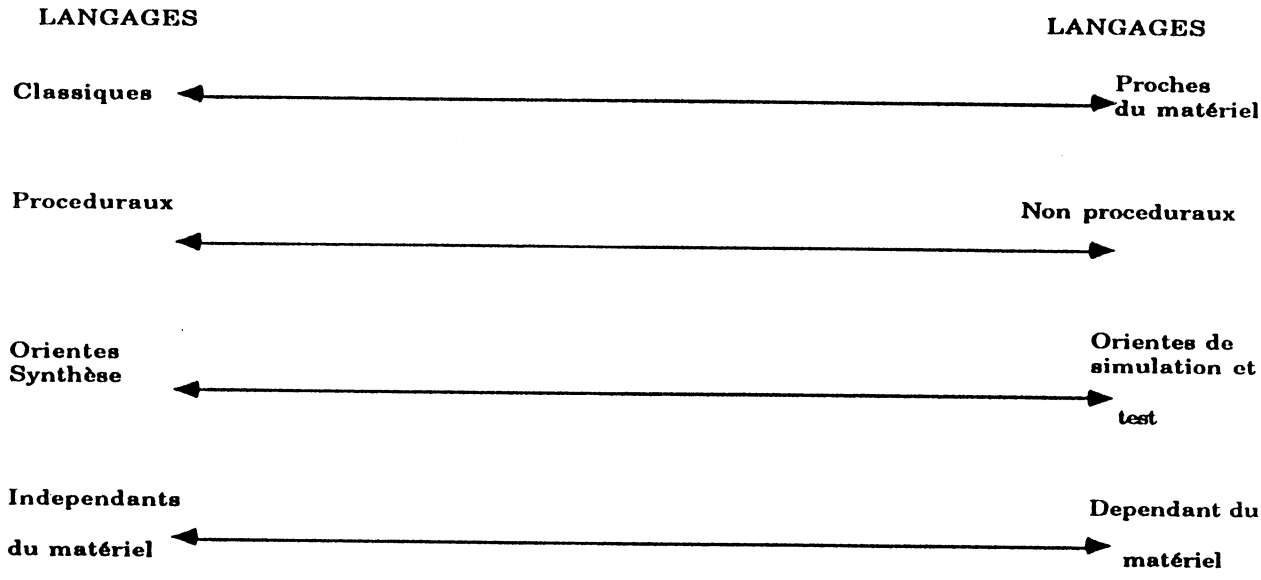


FIG. 48 : PARALLELE ENTRE LES DIFFERENTES PROPRIETES DE LA DESCRIPTION COMPORTEMENTALE

1.3 Conclusion :

Au vu des propriétés précédentes, FIDEL s'insère en bonne place parmi les langages étudiés. Assez proche de VHDL pour les possibilités de description structurelle et de comportement (avec les facilités dérivées d'ADA en moins), FIDEL permet en plus la description au niveau interrupteur.

Par contre, FIDEL n'est pas à l'inverse de CASCADE, CADOC et IRENE un langage prévu pour la synthèse de matériel. Il ne permet pas à l'inverse d'HILO la génération de vecteurs de test. Mais dans le domaine de la simulation FIDEL est aussi complet que SISIM et HILO.

La performance des simulateurs sera le thème de la deuxième partie de cette évaluation. Pour terminer cette partie, il faut remarquer que FIDEL est en bonne position dans l'ensemble des langages de description de matériel.

Actuellement, il n'existe pas encore de langage standard et complet pour tous les domaines d'applications dans le processus de la conception. L'essentiel est toujours de répondre au besoin du concepteur, pour cela aujourd'hui HILO est le plus utilisé , mais il sera probablement remplacé par un autre langage (VHDL ???).

2. EVALUATION PRATIQUE

Nous présentons ici la deuxième étape de notre évaluation. La base de cette partie est le rapport d'évaluation [KJE 86] fait sous la direction de "the Royal Norwegian Council for Scientific and Industrial Research", pour sélectionner un langage de description dans le cadre d'un projet Norvégien. Les principales caractéristiques demandées au langage de description sont les suivantes :

- Le langage va être utilisé dans le processus de la conception comme un outil de : spécification, description, vérification et évaluation.

- Le langage ou un sous ensemble du langage va être exploité comme une interface pour la synthèse automatique.

- Le langage doit être disponible (une version) dans l'environnement VAX.

- Le langage sélecté doit être distribué par une organisation commerciale garantissant les développements, le support et la maintenance.

Dans ce rapport cinq langages ont été sélectionnés :

- DACAPO [BRU 85] : Dosis Gmbh, Dortmund, Germany.
- ELLA [MOR 85] : Praxis plc , Bath , England .
- HELIX [SIL 83] : Silvar-Lisco , Palo Alto , California .
- RTSIa [KNO 80] : Technischen Hochschule , Darmstadt.
- TEXSIM/B [CAL 85] : GE Calma company , Austin, Texas.

Les concepts et les caractéristiques de chaque langage ont été évalués sur un exemple réel : le Am2910 (un contrôleur microprogrammé). Ce circuit est de complexité moyenne. Son fonctionnement est facile à comprendre, et permet de tester les modes de comportement synchrone, asynchrone et bouclé. Cet exemple permet de tester les opérations logiques et arithmétiques d'importance générale.

Les descriptions et les résultats de simulation pour chaque langage sont présentés. Les points forts et faibles de chaque langage sont discutés. Pour nous cet exemple représente une bonne occasion de tester de plus FIDEL sur un circuit réel, et de mettre en évidence les avantages et les inconvénients de FIDEL vis à vis des autres langages.

La démarche suivie dans cette partie comporte trois étapes :

- la première étape résume les caractéristiques principales de chaque langage.
- la deuxième étape présente une comparaison entre les six langages (incluant FIDEL) à travers un ensemble de tables.
- la troisième étape compare le temps CPU de simulation associé à chaque langage et présente la conclusion finale sur chacun.

Pour plus de détails sur chaque langage et sur les simulations effectuées le lecteur se refèrera au rapport [KJE 86] ; pour FIDEL les détails sont présentés dans l'annexe E.

2.1 Caractéristiques générales :

2.1.1 DACAPO :

DACAPO [DOS 86],[RAM 86] et [BRU 85], est développé à l'université de Dortmund (RFA), en coopération avec Siemens. Ce langage est basé sur une syntaxe de type PASCAL avec les extentions nécessaires pour un langage de description de matériel (HDL).

Une description DACAPO apparaît comme un programme PASCAL, avec une entête, un modèle principal et des déclarations de procédures (sous-modèles). Ces procédures peuvent être appelées par le modèle principal ou par d'autres procédures (notion d'imbrication). La structure générale d'une description DACAPO est la suivante :

```
Procedure top_level  (* le modèle principal *)
      " déclarations de constantes et variables globales "
          procedure sub-1 ( " entrée/ sortie " ) ;
                " déclaration locale "
                begin
                    " description de procédure " ;
                end ;
          procedure sub_2 ( E / S ) ;
                .....
```

```
(* description du modèle principal *)
  begin
    .....
  end .
```

La modularitée en DACAPO est réalisée grace aux concepts de procédure, de fonction et d'imbrication de blocs "begin-end". L'existence de variables globales et locales augmente la flexibilité de la modélisation. Il faut noter que le concept de structure explicite (description explicite de connexions entre sous modèles) a été omise : seule la structure implicite est présentée (communication par variables globales et passage de paramètres).

Ce langage utilise un modèle de type Réseau de Pétri temporisé et interprété pour la description fonctionnelle d'un circuit ou une partie de circuit. Une entrée graphique est disponible.

En plus DACAPO offre aux utilisateurs des constructions algorithmiques de contrôle de l'exécution et du flux de données :

- SEQBEGIN pour l'exécution séquentielle
- COBEGIN pour l'exécution concurrente.
- PARBEGIN pour l'exécution parallèle.

La différence entre COBEGIN et PARBEGIN est basée sur la manipulation temporelle dans chaque instruction, comme le montre la figure (49).

Le langage DACAPO offre deux classes de variables : variables explicites (à mémorisation) et variables implicites (sans mémorisation). A une variable explicite est associé un signal qui active sa fonction de mémorisation. Une variable implicite prend une nouvelle valeur à chaque fois qu'on l'affecte.

En d'autres termes les variables explicites font référence aux éléments matériels que sont les bascules, les registres, les mémoires.. ; les variables implicites décrivent les connexions (ou les terminaux) et les parties combinatoires du circuit. Un temps de réaction peut être associé à ces variables implicites.

Le type de base de données en DACAPO est le bit (0,1) étendu aux chaînes de bits. Cette manipulation au niveau bit est intéressante pour les expressions booléenes, par contre elle est fastidieuse pour les opérations arithmétiques.

CODE	TEMPS COURANT
seqbegin	0
A := B;	1
B := A + 2 delay (10);	11
end ;	11
parbegin	11
C := A delay (8);	19
A := C delay (8);	19
end ;	19
cobegin	19
C := A + 1 delay (7);	(26)
D := A + 2 delay (4);	(23)
end ;	26

FIG.49 : MECANISME DU TEMPS EN DACAPO

Les autres types de données sont les entiers (INTEGER) et les variables de temps (TIMEVAR). Il existe aussi des types structurés : tableaux et enregistrements.

Pour la manipulation du temps le langage offre les primitives DELAY et des objets de type "événements". Par défaut, la valeur du retard (DELAY) est 1 (retard unitaire). Il faut remarquer que (cf. figure 49) dans le bloc PAR les valeurs de retard sont toujours les mêmes, alors que dans le bloc CONC elles peuvent être différentes. La configuration d'un bloc PAR avec retard en DACAPO est équivalente à l'utilisation d'un bloc WITHIN en FIDEL.

Les événements peuvent être présentés en deux classes : changement d'un front ou condition booléenne classique.

EXEMPLE :

```
AT ( UP (signal) ) DO ...          (* changement de front *)
AT ( DOWN ( signal) ) DO ...       (* changement de front *)
AT ( CHANGE ( signal) ) DO ...     (* changement de front *)
WHEN condition DO ...              (* condition booléene *)
```

La première classe d'événements permet de décrire les changements dynamiques, tandis que la deuxième permet de spécifier les conditions statiques.

DACAPO offre enfin un environnement de simulation facile et lisible pour les utilisateurs. Cet environnement contient : l'éditeur graphique, les fichiers de stimuli à compiler, un processeur pour l'affichage et mémorisation des résultats de simulation. L'environnement de simulation est de type " batch".

2.1.2 ELLA :

Le concept de base de ELLA est d'offrir un nombre minimal de primitives et de types de signaux, et de permettre au concepteur de construire son modèle avec souplesse et liberté. Le langage est utilisé pour modéliser le comportement d'un circuit aux différents niveaux de description.

Le module de base en ELLA est la fonction ou FN, qui contient les déclarations d'entrée/sortie et la partie de description du comportement (body). Une fonction peut inclure d'autres fonctions. ELLA permet des descriptions récursives.

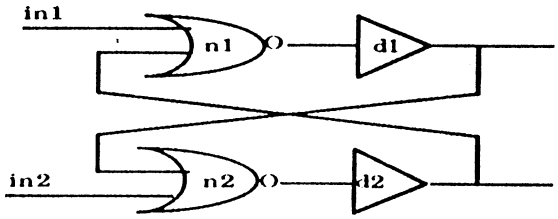
Le comportement d'un circuit peut être décrit de deux façons différentes:

- description fonctionnelle (relation entre les entrées et les sorties) en utilisant l'instruction CASE .
- description structurelle avec instantiation de composants et connexions en utilisant les instructions MAKE et JOIN.

EXEMPLE : description fonctionnelle d'un additioneur

```
Type bool = NEW ( H |L ) .
FN ADDER = ( bool : a b cin ) -> ( bool , bool ) :
    CASE ( a , b , cin ) OF
      (H , H ,H ) : ( H , H ),
      (( H , H ,L) | ( H , L , H ) | (L, H , H )) : ( L, H ),
      ( L , L, L ) : ( L ,L)
    ELSE (H , L )
  ESAC.
```

Description structurelle d'une bascule (R-S) (figure 50)




```
FN RSFF = ( bool : in1 in2 ) --> ( bool,bool ) :
  ( MAKE NOR : n1 n2 ,
               DEL1 : d1 d2.
    JOIN ( in1 , d2 ) --> n1 ,
         ( in2 , d1 ) --> n2 ,
              n1 -> d1 ,
              n2 -> d2.
    OUTPUT ( d1 , d2 ) ).
```

FIG. 50 : DESCRIPTION STRUCTURELLE EN ELLA

ELLA permet de définir des macro modèles (MAC), qui représentent des fonctions paramétrées. La seule primitive matérielle en ELLA est la RAM, qui est utilisée comme une fonction avec trois paramètres d'entrées et un de sortie.

EXEMPLE :

```
TYPE data = NEW ( L| H | X ),
          address = NEW  ad / ( 0 .. 255 ),
          enable = NEW ( on | off ).
FN RAM256 = ( data, address , enable ) -> data  : RAM ( x).
```

En ELLA il n'y a pas de types de base, mais on peut créer les différents types en utilisant le type énuméré.

```
TYPE  bool = NEW ( L | H | X ),
          instr = NEW ( load | add| sub | move ),
```

En ELLA, il y a deux instructions de contrôle : IF-THEN-ELSE et CASE. La manipulation du temps dans le langage est simple ; ELLA n'offre que le contrôle séquentiel et l'utilisation de fonction DELAY qui permet l'existence de différents types de retard (propagation, min-max, retard inertiel).

```
FN DEL1 = ( bool) -> bool : DELAY ( x , 1 ),
```

Cette déclaration décrit un retard de propagation d'une unité du temps, la valeur initiale de la sortie de DEL1 est (x).

```
FN DEL2  = (bool) -> bool : DELAY ( z , 2 , x , 5 ),
```

Cette déclaration décrit une valeur minimale de propagation égale à 2, une valeur maximale égale à 5 et le 'x' représente la valeur d'ambiguïté dans cet intervalle.

FN DEL3 = (bool) -> bool : IDELAY (z , 3),

Enfin, cette déclaration présente le retard inertiel de valeur 3.

L'avantage principal de ELLA est son environnement (ELLA Application Support Environnement ou EASE) qui contrôle les données à travers une base de données, et la gestion d'une bibliothèque. Le concept de "import" et "export" permet au concepteur d'utiliser une bibliothèque et des modèles communs.

Par contre, l'environnement de simulation en ELLA est plus limité : il n'y a pas possibilité de définir le format de sortie textuelle des résultats. Il est possible d'effectuer la simulation en mode "interactif" et "batch", et en utilisant un package spécial le résultat de la simulation s'affiche de façon graphique.

2.1.3 HELIX :

HELIX présente un outil de simulation, dans lequel le comportement des blocs est modélisé par le langage ADLIB [HIL 79] et l'interconnexion de ces blocs est modélisé par le langage de description structurelle SDL [COR 81].

HELIX est basé sur la syntaxe et la sémantique de PASCAL avec des constructions supplémentaires pour contrôler les événements. Les étapes successives pour obtenir un modèle exécutable en HELIX sont les suivantes :

1) Description de chaque module en ADLIB.
2) Dessin des interconnexions des modules sur l'éditeur graphique (CASS).
3) Précompilation de la description en ADLIB pour générer le code PASCAL.
4) Edition de lien avec les différentes routines en HELIX.
5) Le modèle du simulateur est maintenant un programme PASCAL .

HELIX utilise le composant comme entité structurelle de base. Chaque composant peut être partitionné en sous processus avec des contrôles d'activations séparés. Les primitives de contrôle peuvent être regroupées en deux catégories :

- primitives PASCAL (IF-THEN-ELSE, CASE-OF, WHILE-DO, FOR-DO, GOTO) dont la sémantique est classique,
- primitives ADLIB (CHECH, UPON, SYNC, WAITFOR....) .

EXEMPLE :

SUBPROCESS
sync : UPON clk CHECK clk DO ...
async : UPON true CHECK a , b , c , d , e DO ...

Le premier sous processus (sync) indique que l'exécution des actions sera effective sur le front montant d'horloge 'clk' : changement de 'clk' (précisé par CHECK clk) synchronisé avec valeur haute (précisé par UPON clk).

Le deuxième sous processus (async) indique que l'exécution des actions sera effective si une variable parmi la liste associée à l'instruction CHECK change quelque soit la date (condition toujours vraie précisée par UPON true). Il y a aussi la primitive d'affectation avec retard :

ASSIGN r.carry TO line_1 SYNC clk1 PHASE 1

qui signifie que la valeur portée par r.carry sera affectée à line_1 au premier front montant de la phase 1 de l'horloge clk1.

Il existe aussi la primitive "WAITFOR" pour bloquer l'évolution courante d'une action tant que la condition associée est fausse. Une expression de contrôle spécifie les instants d'évaluation de la condition testée. Les instants sont définis soit par une période d'échantillonnage (DELAY), soit par un événement (SYNC et PHASE), soit par une liste de ports d'entrée/sortie (CHECK) .

EXEMPLE :

WAITFOR current >= 0.01 DELAY période_échantillon
WAITFOR ack = 1 SYNC ck PHASE 3
WAITFOR a and b CHECK (a , b)

HELIX fournit aussi des procédures de démasquage (SENSITIZE) ou de masquage (DESENSITIZE) pour rendre visible ou masquer les variations de

certaines portes d'entrée/sortie d'une description. La procédure "DETACH" bloque l'exécution d'un sous processus tant qu'il n'y a pas de variations sur une des variables auxquelles le sous processus est sensible.

Enfin, une caractéristique importante de HELIX est la notion de sous processus pour réaliser des fonctions simples, exécutées indépendamment du processus principal. La synchronisation entre les sous processus est contrôlée par les primitives "PERMIT" et "INHIBIT" dont le paramètre est le nom du sous processus activé ou inhibé.

Tous les types de données standards de PASCAL sont utilisés en HELIX (INTEGER, REAL, BOOLEAN et CHAR) ainsi que les types structurés (vecteur et enregistrement). HELIX offre en plus le type "REGISTER" qui est un vecteur de booléen avec des opérations spéciales.

La seule manière de manipuler le temps en HELIX est l'utilisation de "DELAY" associé à une expression d'affectation "ASSIGN". Tous les sous processus activés au même instant sont exécutés en parallèle, chaque sous processus exécutant son code de manière séquentielle.

L'interface utilisateur en HELIX n'est pas simple : absence de guides et nécessité de récourir en permanence au manuel d'utilisation. La création d'une description nécessite une précompilation HELIX, une compilation du PASCAL généré et une édition de liens.

La simulation HELIX est très rapide parce que le code généré est directement exécutable (pas de phase d'interprétation). La définition de stimulus est simple et flexible et le résultat de simulation peut être présenté sous forme de tableaux ou de graphiques. La simulation peut être exécuté de manière "interactive" ou "batch".

2.1.4. RTSIa :

Le langage RTSIa est inspiré du langage DDL [DIE 78] (Digital Design Language) et utilise une syntaxe proche de PASCAL. Le concept de base pour la modélisation est l'automate de Mealy (automate d'état fini avec des transitions d'état conditionnée).

Les deux porteurs de base sont "REGISTER" et "TERMINAL" ; le premier peut stocker des données et représente les variables d'états ; le second représente les connexions.

La description d'un modèle en RTSIa apparaît comme suit :

partie déclaration : entrée/sortie : terminals ,
terminals ,
registers,
MICRO
partie instructions : transfert de registre
connexion de terminals
instruction IF
instruction CASE

MICRO est une facilité de macro mais sans paramètres, il existe aussi une primitive nommée "UNIT" avec des paramètres d'entrée/sortie de type "terminal".

Les seuls types de données en RTSIa sont le "terminal" et "register" qui sont des ensembles de bits ou valeurs booléens (0 , 1). RTSIa ne permet pas la définition de ports bidirectionnels. Les valeurs 'X' et 'Z' ne sont pas modélisés de manière directe.

Exemple de partie de déclaration :

- INPUT - START , DATA [0 : 7] , CNT [4 : 2] ;
- OUTPUT - STOP , BUS [0 : 7] ;
- TERMINAL - A , B , BUSA [0 : 7] , XBAR [1 : 4 , 1 : 4] ;
- REGISTER - DFF , SHIFT [0 : 7] , ROM [0 : 1023 , 15 : 0] ;

Il y a seulement deux types de contrôle en RTSIa, IF-THEN-ELSE et CASE. Le traitement temporel est pris en compte par la machine synchrone d'état fini.

On peut distinguer les quatre phases d'une horloge de la façon suivante :

- REGISTER - CLOCK [1 : 0] ...
/ EQ (CLOCK , #00B2) / CLOCK <= #01B2 , (niveau bas)

/EQ (CLOCK , #01B2)/ CLOCK <= #11B2 , (front montant)
/EQ (CLOCK , #11B2)/ CLOCK <= #10B2 , (niveau haut)
/EQ (CLOCK , #10B2)/CLOCK <= #00B2 , (front descendant).

L'ordre des instructions en RTSIa est quelconque : la notion de parallélisme est implicite. Le langage ne permet pas la description structurelle explicite.

L'environnement de simulation en RTSIa est réduit aux commandes de la machine hôte. La compilation n'est pas séparable.

2.1.5 TEXSIM/B :

TEXSIM (TEGAS EXtended SIMulator) est un simulateur permettant la vérification des circuits digitaux aux niveaux fonctionnel et logique. Il est la nouvelle génération de simulateurs après TEGAS5 [THO 80]. Il utilise le langage TDL/S (TEGAS Design Language/Structural) pour la description structurelle et permet une hiérarchie jusqu'à 30 niveaux. TEXSIM contient environ 80 primitives de base aux différents niveaux (interrupteur, logique et fonctionnel). Les primitives fonctionnelles sont paramètrées.

Le langage de description du comportement TEXSIM/B est inspiré des langages ADA,PASCAL et C. Il utilise également quelques instructions de VHDL .

Chaque module du comportement peut être décrit comme suit :

```
module < name > ( < input/output_nodes> ) is
< declarations >
initial
< initialization of internal variables >
behavior
< body >
end < name > ;
```

L'interface est décrite par une liste de déclaration de noeuds d'entrée/sortie avec leurs types, attributs et les options. La partie de description du comportement est séquentielle ; elle contient des conditions, des opérations et des instructions de gestion des événements générés (qui sont associées seulement aux noeuds de

sortie). Un exemple de description comportementale d'un registre à décalage (4 bits) est présenté :

```
module MATCHP ( CLK,ENABLE,DATAIN : in bit ; S1001 : out bit ) Is
    REG : bits [ 0..3 ] ;
    Behavior
    if ENABLE && CLK then
       REG := DATAIN & REG [ 0..2 ] ;
    end_if ;
    S101 <== (10) REG [0] /\ ~ REG [1] /\ ~ REG [ 2 ] /\ REG [3 ] ;
    print " current_time, CLK,ENABLE,DATAIN,REG : ",
              current_time, " ", CLK,ENABLE,DATAIN," ", REG ;
    print ;
    end MATCHP ;
```

Les types de base en TEXSIM sont "integer" et "bit", le premier est l'entier de la machine hôte (32 bits, complément à deux), le deuxième peut avoir les quatre valeurs (0,1,X et Z). A partir de ces types on peut construire les types structurés (vecteurs et tableaux). TEXSIM utilise les attributs pour spécifier la directivité des noeuds (IN ou OUT). Les options sont utilisées pour déterminer la sensitivité de chaque entrées (WATCH ou IGNORE).

Un module est activé chaque fois qu'une entrée avec l'option "WATCH" change, tandis que le changement des variables avec l'option "IGNORE" ne provoque aucune activation. Les options associées aux sorties sont "SPIKE","NOSPIKE" et "REJECT" pour déterminer les valeurs d'une sortie affectée plusieurs fois pendant l'activation du modèle. L'option "SPIKE" génère un "X" entre deux mise à jour, "NOSPIKE" garde toutes les valeurs afféctées à la sortie et "REJECT" ignore la dernière mise à jour.

TEXSIM n'offre que l'instruction "IF-THEN-ELSE" comme structure conditionnée de contrôle (le CASE sera introduit dans la prochaine version). Il y a deux instructions de contrôle non conditionnées "GOTO" et "EXIT", ainsi que les instructions de boucle "LOOP", "WHILE", et "DO".

Le temps en TEXSIM est géré grâce à l'utilisation de retard associé à l'affectation des variables. La figure (51) montre un exemple de communication

entre deux processus en utilisant le retard interne comme un événement d'activation.

TEXSIM permet la vérification temporelle grâce aux primitives suivantes :

I ' PREV_INPUT : retourne l'anciens valeur de I (previous value).
I ' PREV_CHANGE : retourne la date de dernière modification de I.
I ' SET_UP : retourne l'intervalle du temps depuis la dernière modification.

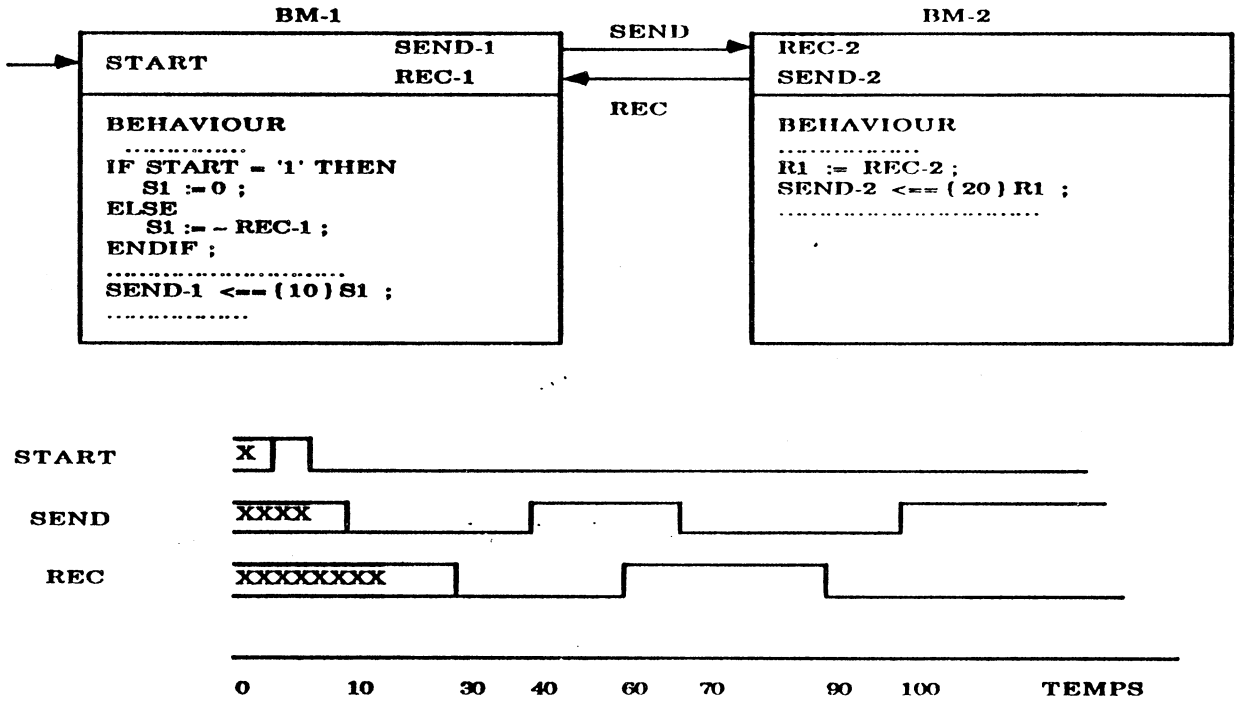


FIG. 51 : PROCESSUS DE COMMUNICATION EN TEXSIM/B

Par exemple on peut détecter le front descendant sur une variable en utilisant l'expression :

```
IF CLOCK = '0'
&& CLOCK ' PREV_INPUT = '1'
&& CLOCK ' PREV_CHANGE = CURRENT_TIME
THEN.....
```

Il faut remarque que ces attributs existent en FIDEL mais de manière implicite.

TEXSIM n'offre pas d'environnement de support aux utilisateurs sur VAX, mais il existe sur la station de travail "APOLLO". La simulation peut être "interactive" ou "batch" avec des facilités de "trace/debug". Le résultat peut être présenté sous forme de tableaux de différents formats (binaire, octal,hexa ou décimal), et en mode graphique sur "APOLLO".

2.2. Comparaison détaillée :

Dans cette partie, on va présenter une comparaison détaillée entre les six langages (incluant FIDEL) en utilisant des tableaux (1,2,3,4) pour montrer les différentes propriétés de chaque langage.

2.3 Conclusion de la comparaison :

On présente d'abord la table (5) qui contient le résultat des simulations effectuées dans chaque langage sur l'exemple AM2910. On présente ensuite les différentes résultats de cette comparaison.

Il faut noter que la simulation de cet exemple (AM2910) est effectuée sur différentes machines, donc pour comparer de façon propre le temps de simulation de chaque système (langage), on doit tenir compte des rapports de puissance entre chaque machine. La machine VAX/780 est prise comme machine de référence. Ce rapport est environ de 0.5, 1 et 1.8 pour les machines 750, 780 et 785 respectivement.

Donc, les temps de simulation ramenés sur VAX 11/780 sont : DACAPO (21.5 S), ELLA (55 S), FIDEL (10.8 S), HELIX (16 S), RTSIa (10.5 S) et TEXSIM/B (133.5 S). Ce calcul montre que TEXSIM/B est le plus lent. Ceci est dû à l'utilisation du bit comme type de base (par exemple si les 12 bits d'un vecteur sont changés au même instant,12 événements sont générés). Les performances de FIDEL sont sensiblement les mêmes que celles de RTISa.

Les concepts de base en DACAPO (réseau de Pétri, construction de contrôle) permettent le parallélisme de comportement aux différents niveaux d'abstraction. Les différents tâches de modélisation, de simulation et de génération de stimuli sont bien intégrées. Le formalisme du langage est utilisable pour la synthèse automatique mais on a besoin d'amélioration pour : ajouter plus de types de données, gérer la bibliothèque, et permettre la compilation séparée.

TABLE 1 : Concept du langage

PROPRIETE SYSTEM	Concept de structure	Concept de comportement	Mecanisme de contrôle	Mecanisme du temps
DACAPO	Procédures avec paramètres E/S	PASCAL avec procédures	séquentiel, synchrone et concurren	Asynchrone
ELLA	FN (E/S) MACRO et JOIN récursive	Description récursive	Séquentiel, machine d'état fini	Asynchrone
FIDEL	Composants avec E/S dans un langage structurel	Description fonctionnelle	Séquentiel, synchrone et parallèle	Asynchrone
HELIX	Composants avec E/S	PASCAL avec procédures	Séquentiel, synchrone, conccurent	Asynchrone
RTSIa	Procédures avec E/S (externes)	PASCAL avec Externals	Synchrone	Synchrone
TEXSIM	Module avec E/S dans un langage structurel	VHDL, modules compilés sépérament	Séquentiel, synchrone	Asynchrone

TABLE 2 : Modularité, contrôle et temps

PROPRIETE / SYSTEME	Concept de modularité	Construct de contrôle	Construct du temps
DACAPO	BEGIN...END Procédures (param) Fonction (Param)	if-then-else, case,repeat-until,while, do, for-do, interrupt	at up | down | change | Do, when C do
ELLA	FN (param) MACRO(par) MAKE, JOIN	if-then-else, case	DELAY
FIDEL	MODEL (par) MACRO (par) CONNECT (hiérarchique)	if-then-else, select,while, for,when	when rises, falls,changes, becomes,setup, hold,within
HELIX	BEGIN...END PROC (par) FUN (par) PROC EXT	if-then-else, case,while, repeat,for, (de)senstize	transmit, check,becomes, setup,hold, upon, waitfor
RTSIa	BEGIN...END MACRO UNIT (par)	if-then-else, case	transfert chaque cycle
TEXSIM/B	MODULE (par) hiérarchique	if-then-else, case,while, do,loop,exit, goto	current-time prev-time, prev-change set-up

TABLE 3 : TYPE DE DONNEES

PROPRIETE SYSTEM	Type de base	Types à définir	Global/local variables
DACAPO	Bit-3 : 0,1,X Bit-7 : 0,1,X,L H,Y,Z integer (32) Timevar (64)	Chaîne de bits vecteur de bits, enregistrement, procédures export	Global, Local
ELLA		Type énumeré Type composé, integer	Local
FIDEL	Bit : 0,1,X,Z integer, constant	vecteur de bits (1D,n bits), tableau de bits (2D, n bits)	Local
HELIX	Boolean, integer, pointeur,real, character	chaîne de bits vecteur,record type énumeré	Global, Local
RTSIa	Bit : 0 , 1	Vecteurs de bits (1D,2D)	LOCAL
TEXSIM/B	Bit : 0, 1, X , Z Integer Constant	chaîne de bits vecteur (n D)	Local

TABLE 4 : Caractéristiques générales

PROPRIETE / SYSTEM	Compilation séparée	Bibliothèque	Structure explicite	Machine hôte	Langage d' implémentation
DACAPO	NON	NON	NON, mais des fichiers internes existent	VAX,UNIX	PASCAL,C
ELLA	OUI	OUI	NON	VAX	ALGOL 68
FIDEL	NON	NON	OUI	VAX	PASCAL, FORTRAN IV
HELIX	NON	OUI pour structure	OUI	VAX, APPOLO IBM	PASCAL
RTSIa	NON	NON	NON, mais des fichiers internes existent	VAX	FORTRAN IV
TEXSIM/B	OUI	OUI pour structure	OUI	VAX APOLLO	FORTRAN IV

TABLE 5 : Temps d' exécution de AM2910

SYSTEME	T1	T2	T3	Machine hôte
DACAPO	20S	43S	9S	VAX 11/750
ELLA	40S	110S		VAX 11/750
FIDEL	4S	6S		VAX 11/785
HELIX	30S	16S		VAX 11/780
RTSIa	14S	21S		VAX 11/750
TEXSIM/B	280S	267S	27S	VAX 11/750

T1 : Temps de compilation
T2 : Temps de simulation
T3 : Temps d' affichage de résultat.

ELLA est le seul langage qui ne fournisse pas de primitive de base. L'environnement EASE est facile pour la manipulation des modèles et de la bibliothèque. ELLA est bien adapté pour la description au niveau le plus haut. Le système de synthèse peut utiliser ELLA comme langage d'entrée, mais la sémantique semble difficile à traiter. Les améliorations possibles sont : fournir une interface de simulation d'utilisation plus simple .

HELIX est un outil pour créer des simulateurs dédiés, l'utilisation du langage demande une certaine connaissance de la programmation. Les différentes utilités proposées manquent de formalisme, ce qui limite l'utilisation du langage comme entrée pour la synthèse logique. L'interface utilisateurs a besoin d'amélioration.

Les applications de RTSIa sont limitées, mais il peut être utilisé pour la synthèse de machine d'état fini au niveau logique.

TEXSIM/B fournit un bon environnement de simulation logico-fonctionnelle (comparable à FIDEL-EPILOG) qui facilite la tâche de conception dans la méthode descendante. Les différentes améliorations sont : possibilité de décrire le parallélisme dans le module fonctionnel, compilation et exécution plus rapide.

CONCLUSION

Dans ce chapitre nous avons présenté l'évaluation (théorique et pratique) de FIDEL par rapport aux autres langages de description de matériel. Le but est d'essayer de classer les différents langages selon leurs tendance et propriétés afin de mieux situer FIDEL dans l'ensemble des langages de description.

Au vue de cette étude, FIDEL s'insére en bonne place parmi les différents langages étudiés, tant au niveau des concepts que de l'utilisation pratique. Certaines améliorations restent nécessaires : enrichir les types de données de base, améliorer la construction et la gestion de la bibliothèque (intégration réelle avec CASSIOPEE), permettre la compilation séparée des modules et améliorer la présentation des résultats de simulation.

CONCLUSION

L'étude des propriétés et des concepts des langages de description de matériel nous a permis de mettre l'accent sur des points importants que nous avons pris en considération lors de la définition et de l'implémentation de FIDEL.

La représentation de la description FIDEL, comme une boite noire, assure l'indépendance du langage vis à vis des technologies et des méthodologies de conception. La structure de base du modèle fonctionnel est bien adaptée à l'ensemble d'éléments de base pour décrire un système logique. Le temps et le parallélisme sont définis de manière simple et efficace et peuvent être utilisés à tous les niveaux de la description.

L'intégration de FIDEL dans le simulateur logique EPILOG qui aboutit à une simulation de niveau mixte (logico-fonctionnel), et dans le simulateur électrique ELDO qui aboutit à une simulation de mode mixte (électrico-fonctionnel) représentent deux applications importantes. Ces deux applications présentent une avancée dans le domaine de la simulation, dans le but de garder la précision tout en diminuant le coût de simulation des circuits VLSI (par exemple les grosses mémoires RAM).

Nous avons présenté le système complet de FIDEL (description fonctionnelle, description structurelle et environement de simulation) avec les différentes caractéristiques (vérification temporelle , description hiérarchique, mélange de description synchrone et asynchrone et utilisation d'un seul langage) qui sont bien adaptées à une simulation hiérarchique et multi niveau.

Dans l'étude d'évaluation des langages de description de matériel, FIDEL s'est inséré en bonne place tant au niveau des concepts que de l'utilisation pratique.

Ceci nous permet de dire que FIDEL a atteint la majorité des objectifs qui étaient fixés à sa définition. Il reste toutefois des recherches à apporter : enrichir les types de données de base, comment permettre aux utilisateurs de définr leurs types et opérateurs, étudier les problèmes posés pour l'extension de la description au niveau système.

Naturellement il reste encore beaucoup de problémes techniques à résoudre dans le domaine de CAO pour les circuits VLSI. En ce qui concerne la description et la simulation de ces circuits, il s'agit de mettre en évidence l'utilisation d'un

langage commun qui doit être efficace et puissant même s'il est complexe. Ce fait est caractérisé par les trois points suivants :

- L'utilisation d'un seul langage dans la description du circuit à tous les niveaux (à partir du niveau système jusqu'au niveau logique) assure la continuité de description et facilite la tâche de description et la construction du simulateur multi-niveau.

- Le fait d'utiliser le même langage pour fournir des informations tant aux concepteurs qu'aux testeurs, va établir un bon lien entre ces deux communautés. La suppression d'une barrière entre ces deux disciplines entraînera une réduction de coût et une amélioration de la qualité de conception des circuits. L'utilisation de ce langage pour aider à calculer la longeur de vecteurs de test au niveau fonctionnel en est un exemple.

- L'avancement des outils de synthèse demande de plus en plus l'existence de ce langage commun pour faciliter la tâche de transformation d'un niveau à l'autre. Par exemple : les modèles de signaux et de retards doivent être les mêmes entre le système de spécification/simulation et le système de synthèse.

REFERENCES

REFERENCES

[ALI 78] G. Alia et al
" LSI Components Modelling in a Three Valued Functional Simulation", Proc. 15 th Design Automation Conference, 1978.

[AMA 85] P. Amadou et al
" MOSLAM : A Switch Level Simulator based on an efficient analysis of conducting paths", ESCIRC 1985.

[AMB 83] P. Amblard et al
" CADOC : A Functional specification and simulation tool", Proc IEEE Int. Conf. on Computer Aided Design (ICCAD'83), 1983.

[AND 85] V.G.Andrew et al
"Approches toward silicon compilation", IEEE Circuits and Devices Magazine, May, 1985.

[ARN 78] G.Arnout, H.De Man
" The Use of Threshold Function and Boolean Controlled Network Elements for Macromodelling of LSI Circuits", IEEE J. Solid-State Circuits, Vol. SC-13, Jun. 1987,pp.326-332.

[ASS 87] J. Assael, P. Senn, M.S. Tawfik
" Switched capacitor filter silicon compiler", VLSI Symp., Japan, May 1987.

[BAN 83] P. Banerjee, J.A. Abraham
" MURPHY : A logic simulator for MOS VLSI circuits", Coordinated Science Laboratory, Illinois,Urbana, 1983.

[BAN 85] P. Banerjee, J.A. Abraham
" A Multivalued Algebra for Modeling Physical Failures in MOS VLSI Circuits", IEEE Trans. on Computer Aided Design, Vol. CAD-4, No. 3, July 1985.

[BAR 75] **M. R. Barbacci**
" A Comparison of Register Transfer Languages for Describing Computers and Digital Systems", IEEE Tran. Computers, Vol. C-24, No. 2 , Feb. 1975, pp. 137-150.

[BAR 79] **M.R.Barbacci**
" Instructions Set Processor Specificatios for Simulation, evaluation and synthèse", Proc. 16 th Design Automation Conference, 1979, pp. 64-72.

[BAR 81] **M . R . Barbacci**
" Instruction Set Processor Specifications (ISPS) : The Notation and Its Applications", IEEE Trans. Computer, Vol.C- 30, No. 1, Jan. 1981, pp.24-40.

[BAR 85] **M.R. Barbacci, T. Uehara**
Computer, Vol. 18, No. 2 (Special issue on CHDL's), Fev. 1985.

[BARR 85] **G.Barros**
"A circuit simulation Tutorial", VLSI design 1985.

[BEL 85] **C. Bellon et al**
" CADOC system : a tool for multilevel description and test generation for VLSI circuits", Proc. 7 th Int. Symp. on Computer Hardware Description Languages and their Applications (CHDL' 85), Aug 1985,pp. 364-380.

[BEN 79] **L. Bening**
"Developments in computer simulation of gate level physical logic", Proc. 16 th Design Automation Conference, June 1979, pp. 561-567.

[BER 84] **J.M. Berger et al**
" LOF : A bulding Tool for Flexibel blocks Libraries ", Proc. IEEE International Conference on Computer Design,ICCD' 84, pp. 851-856.

[BLA 84] T. Blank
" A survey of Hardware accelerators used in computer aided design", IEEE Design & Test of computers, Vol.1, No.3, 1984, pp. 21-39.

[BOR 79] D. Borrione, J.F. Grabowiecki
" Informal Introduction to LASSO : a Language for Asynchronus System Specification and Simulation", Proc. EURO IFIP 79, London, Sept. 1979.

[BOR 81] D. Borrione
"Langages de description de systèmes logiques",Thèse d'état INPG Grenoble,Juil 1981.

[BOR 85] D. Borrione, C. Le Faou
" Overview of the CASCADE multilevel Hardware Description Language", Proc. 7 th Int. Symp. on Computer Hardware Description Languages and their Applications (CHDL' 85), Aug. 1985, pp. 239-260.

[BOS 77] A.K. Bose, S.A. Szygenda
" Detection of static and dynamic hazards in logic nets", Proc. 14 th Design Automation Conference, 1977.

[BOU 84] D. Boucher et al
" GRAFCET as Description and Simulation tool at the functional level", Proc. IEEE Int. Symp. Circuits & Systems (ISCAS' 84), 1984.

[BOU 85] D.Boucher, H. EL Tahawy et al
" A complete Functional Level Tool", Proc. IEEE Int. Symp. Circuits & Systems (ISCAS' 85), 1985.

[BOU 87] D. Boucher, J. Le Roux
" Description and simulation of pyramidal architecture for image processing", Proc MedWest Conf. Circuits & Systems, 1987.

[BRE 64] **M.A. Breuer**
" Techniques for the Simulation Of Computer Logic", Commu. Ass. Comput. Mach., Jul. 1964, pp. 443-446.

[BRE 75] **M.A. Breuer**
" Digital System Digital Simulation : Languages, Simulation & Data Base", Pitman, 1975.

[BRU 85] **R. Brück, F. Kleinjohann, and F. Rammig**
" Synthesis of Modular Controllers from CAP/DSDL (DACAPO) Description", Proc. of the 7 th International Conference on Computer Hardware Description Languages, August 1985, pp. 79-97.

[BRY 80] **R. E. Bryant**
" An algorithm for MOS logic simulation", Lambda, Vol. 1, No. 3, Fourth quarter 1980, pp. 46-53.

[BRY 81] **R.E. Bryant**
" MOSSIM : A Switch Level Simulator for MOS LSI", Proc. 18 th Design Automation Conference, 1981, pp. 786-790.

[BRY 87] **R. E. Bryant et al**
" COSMOS : A Compiled Simulation for MOS Circuits", Proc. 24 th Design Automation Conference, 1987.

[CAL 85] "TEXSIM Refrence Manual, Release 3.1", Vol. 1-2, GE-Calma, Austin 1985.

[CAB 85] **G.Cabadi et al**
"Experiences in CONLAN based formal verification of HDL's", Proc. of the 7 th International Conference on Computer Hardware Description Languages, August 1985.

[CAN 85] **M. Cand et al**
" Specifications du UPTS" Note Technique, CNET Grenoble, 1985.

[CAZ 86] **M. Cazal**
"Etude sur langages de description", proj. D.E.A, INPG 1986.

[CAZ 86] **M.Cazal**
" Spécifications Structurelles pour le langage FIDEL", Proj. 3 ème Année , INPG 1986.

[CHA 76] **S.G. Chappell et al**
" Functional simulation in the lamp System", Journal of Design Automation & fault tolerant computing, Vol. 1, No.3, May 1976.

[CHA 75] **B. Chawla et al**
" MOTIS- A MOS timing Simulator", IEEE Trans. Circuits Syst. Vol. CAS- 22, Desc 1975, pp. 301-310.

[CHI 76] **C. Chicoix, J. Thuel, R. Tulloue**
" EPISODE : A set of tools oriented to logic integrated circuit design verification and testing", ESSCIRC , Toulouse, Sept. 1976.

[CHU 72] **Y. Chu**
"Introducing Computer Design Language", Compcon 72, Sept. 1972, pp. 215-218.

[CHU 74] **Y. Chu**
Computer , Vol. 7, No. 12 (Special issue on CHDL's), Dec. 1974.

[COE 84] **D. R. Coelho, C. Neti**
" Timing Verification using a general behavioral Simulator", Proc. IEEE Int. Conf. on Computer Design (ICCD'84), 1984.

[COR 81] **W.E. Cory, W. A. Van Cleemput**
" Symbolic simulation for functional verification using ADLIB and SDL", Proc. 18 th Design Automation Conference, 1981, pp. 82-89.

[CRA 85] **M.D. Crastes**
" Specification et Simulation Fonctionnelles de Circuits Complexes : Le système CADOC : ", Thèse Doct. Ing. INPG 1985.

[DAH 66] **O.J. Dahl, K. Nygaard**
" SIMULA - An ALGOL- based Simulation Language", Comm. ACM, Vol. 9, Sept. 1966, pp. 671-678.

[DAN 82] **M.E.Daniel, C.W.Gwyn**
" CAD system for IC Design", IEEE Trans. CAD, Vol. CAD-1,Jan 1982, pp. 2-12.

[DAS 82] **H.W.Daseking et al**
" VISTA : A VLSI CAD System", IEEE Trans. CAD, Vol. CAD-1, Jan 1982, pp. 36-51.

[DEL 80] **M. Delaunay, J.F. Grabowiecki**
" Transformateur de Grammaire LL1", Note Technique, Sept. 1980. R.R. IMAG 234.

[DEN 82] **M.M. Denneau**
" The Yorktwon Simulation Engine : introduction", Proc 19th Design Automation Conference, 1982, pp. 55-59.

[DIE 74] **D.L. Dietmeyer**
" Introducing DDL", Computer , Vol. 7, No. 12 (Special issue on CHDL's), Dec. 1974.

[DIE 78] **D.L. Dietmeyer**
" Logic Design of Digital System", Chapter 2.8 : An Introduction to DDL, Allyn and Bacon, Inc., Boston 1987.

[DOS 84] **M.H. Doshi et al**
" THEMIS- A mixed mode multilevel hierarchical interactive digital circuit simulator", Proc. 21 st Design Automation Conference, 1984.

[DOS 86] " DACAPO II Manual. Version 2.0" March 1986.

[DUL 69] **J.R. Duley, D.L. Dietmeyer**
" Translation of a DDL Digital System Specification to Boolean Equations", IEEE Trans. Computers, Vol. C-18, April 1969, pp. 305-313.

[EFC 86] "EPILOG : manual d' utilisation", EFCIS 86.

[EL 84 a] **H. El Tahawy et al**
" Functional Modeling for Logic Simulation", IEEE Int. Conf. on Computer Design (ICCD' 84), 1984.

[EL 84b] **H.EL Tahawy et al**
" Integration of Functional Model in Logic Simulator", Proc. Int. Computer Symp. (ICS'84), Taipei, 1984.

[EL 86] **H.EL Tahawy et al**
" Integration of Functional Model in Electric Simulator", Proc. IEEE Int. Symp. Circuits & Systems (ISCAS' 86), 1986.

[EL 86] **H. EL Tahawy et al**
" FIDEL : A Multilevel Hardware Description and Simulation Language" Proc. Integrated Circuit Technology Conf. , Limeric, Ireland, Sept 1986.

[EL 87] **H. EL Tahawy et al**
" A New Implementation Technique for the Simulation of Mixed (Digital-Analog) VLSI circuits", Proc. IEEE Int. Conf. on Computer Aided Design (ICCAD' 87), 1987.

[EST 78] **G. Estrin**
" A Methodology for the Design of Digital Systems, supported by SARA at the age of one", Proc. National Computer Conf., june 1978, pp. 313-324.

[FAN 77] **S. P. Fan et al**
" MOTIS-C : a new circuit simulator for MOS LSI circuits", Proc. IEEE Int. Symp. Circuits & Systems (ISCAS' 77), 1977.

[FIG 73] **M.A.Figueroa**
"Analyses of languages for the design of digital computers", Univ. of Illinois,Urbana,IL, Tech.Rep. May 1973.

[FLA 81] **P.L. Flake et al**
" HILO mark 2 Hardware Description Language" ,Proc. 5th Int. Symp. on Computer Hardware Description Languages and their Applications (CHDL'81), 1981 ,pp.95-108.

[FLA 83] **P.L. Flake et al**
" An algebra for logic strength simulation " , Proc. 20 th Design Automation Conference, june 1983, pp. 615-618.

[FLA 84] **E. Flamand**
" A Complete and Automatic System for Sequencer Design", Proc. IEEE Int. Conf. on Computer Design (ICCD' 84), 1984.

[GAR 79] **R.J. Gardher, P.B. Weil**
" Hierarchical Modeling and Simulation in VISTA", Proc; 16 th Design Automation Conference, 1979.

[GON 84] **M. Gonauser et al**
" SMILE - A mutilevel Simulation System", Proc. IEEE Int. Conf. on Computer Design (ICCD' 84), 1984.

[GOR 69] **G. Gordon**
"System Simulation", Prentice-Hall, Inc.,Englewood Cliffs, N.J., 1969.

[HAC 81] **G.D. Hachtel, A. L. Sangiovanni-Vincentelli**
" A survay of Third generation simulation techniques", Proceedings IEEE, Vol. 69,No. 10, Oct. 1981.

[HAC 82] **G.D. Hachtel, M.R. Lightner**
" Implication Algorithm for MOS Switch-Level Functional Macromodelling, Implication and Testing", Proc. Design Automation Conference, 1982, pp. 691-698.

[HAR 79] **R. W. Hartenstein, E. V. Puttkamer**
" KARL : A Hardware Description Language as a part of a CAD tool for LSI", Proc. 4 th Int. Symp. on Computer Hardware Description Languages and their Applications, Oct 1979.

[HEL 86] **J.P. Hellmann**
" Programming with FACILE", note interne, CNET Grenoble, 1986.

[HEN 85a] **B. Hennion, P. Senn, D. Coquelle**
" A new Algorithm for Third Generation Circuit Simulators : the One Step Relaxation Method", Proc. 22 nd Design Automation Conference, June 1985.

[HEN 85b] **B. Hennion, P.Senn**
" ELDO : A new third generation circuit simulator using the One Step Relaxation Method", Proc. IEEE Int. Symp. Circuits and Systems (ISCAS'85), 1985.

[HEN 86] **B. Hennion**
" Méthodes numériques utilisées dans le programme de simulation électrique ELDO", Analyse des Telecommunication, Tome 41, No. 1-2, 1986.

[HEN 87] **B. Hennion et al**
" ELDO : A general Purpose Third Generation Circuit Simulator Based on the OSR Method", ECCTD 1987.

[HEY 82] **M.H.Heydemann et al**
" Implementation Issue for Multiple Delay Switch-Level Simulation", IEEE Int. Conf. Circuits & Computers, Sep. 1982, pp. 46-49.

[HEY 83] **M.H.Heydemann**
" A Survey of MOS Logic Simulation Tools", ESSIRC 1983.

[HIL 79] **D. Hill, W. VanCleemput**
" SABLE : A tool for generating Structured, multilevel Simulation", Proc 16 th Design Automation Conference, 1979, pp. 272-279.

[HIL 79] **D. Hill**
" ADLIB : A Modular, Strongly-Typed Computer Design Language", Proc. 4 th Int. Symp. on Computer Hardware Description Languages and their Applications, Oct 1979.

[HIT 82] **R. Hitchcoch**
" Timing Verification and the Timing Analysis Program", Proc. 19 th Design Automation Conference, 1982.

[HIT 83] **C. Y. Hitchcock, D.E. Thomas**
" A method of automatic data path synthesis", Proc. 20 th Design Automation Conference, 1983, pp. 484-489.

[ISH 85] **N.Ishiura et al**
" High speed logic simulation on a vector processor", Proc. IEEE Int. Conf. on Computer Aided Design (ICCAD' 85), 1985, pp. 119-121.

[ISH 87] **N. Ishiura et al**
" High-Speed Logic Simulation on Vector Processors", IEEE Trans. Computer-Aided Design , Vol. CAD-6, No. 3, May 1987, pp. 305-321.

[JAM 86] **R.Jamier**
"Génération automatique de parties opératives de circuits VLSI de type microprocesseur", Thèse Ing. NOV. 1986.

[JUL 86] **C.Jullien, A. Leblond**
" A Data base Interface for An Integrated CAD System", Proc. 23 th Design Automation Conference, 1986, pp.760-767.

[KEL 82] **J.E. Kelckner et al**
" Electrical Consistency in Schematic Simulation", Proc IEEE Int. Conf. Circuits & Computers, 1982, pp. 30-34.

[KIV 69] **P.J. Kivita et al**
" The SIMSCRIPT II Programming Language", Prentice-Hall, Inc., Englewood Cliffs, N.J. 1969.

[KJE 86] **S. Kjetil, J.Á. Einar**
" Benchmarking and Evaluation of Hardware Description Language for Design of VLSI Systems", Rapport, ELAB 1986.

[KNO 80] **H.J. Knobloch**
" Description and Simulation of complex Digital Systems by means of Register Transfer Language RTSIa", Nato advanced Study on Computer Design Aids for VLSI circuits, Sogesta-Urbino, Italy, July-August 1980.

[KOW 83] **T.J. Kowalski , D.E. Thomas**
" The VLSI design automation assistant : prototype system", Proc. 20 th Design Automation Conference, 1983, pp. 479-483.

[LAR 85] **J . L. Lardy, J . Boulvin , et A. Girard**
" La conception de circuits intégrés silicium au CNET ", L'Echo des RECHERCHES No. 121, 1985, pp. 23-32.

[LEC 82] **J . Lecourvoisier et Al**
"A design methodology based upon symbolic layout and CAD tools", 19 th Design Automation Conference, june 1982, Las Vegas.

[LEF 85] **C.LE Faou**
"Hierarchical Multi-level Mixed-Mode Simulation in CASCADE", R.R IMAG No 513, 1985.

[LEL 82] **E. Lelarasmee et al**
" The Waveform Relaxation Method for the time-domain analysis for large scale integrated circuits", IEEE Trans. CAD, Vol. CAD-1, No. 3 , Aug. 1982, pp. 131-145.

[LIP 77] **G. J. Lipovski**
" Hardware Description Languages : Voices from the Tower of Babel", Computer, Vol. 10, No. 6 , june, 1977,pp.14-17.

[LIP 86] **R. Lipsett et al**
" VHDL - The Language", IEEE Design & Test, Apr. 1986.

[MAR 86] **S. Marine**
" IRENE un langage pour la description, simulation et synthèse automatique du matériel VLSI", Thèse INPG, 1986.

[MCW 78] **T.M. McWilliams , L.C.Widdoes**
" SCALD : structured computer-aided logic design", Proc. 15 th Design Automation Conference, 1978, pp. 271-277.

[MCW 80] **T.M. McWilliams**
" Verification of Timing Constraints on Large Digital Systems", Proc. 17 th Design Automation Conference, 1980.

[MED 79] **C. Mead, L. Conway**
" Introduction to VLSI Systems", Reading, M.A : Addison-Wesley, 1979.

[MER 73] **J. Mermet**
" Etude méthodologique de la Conception Assistée par Ordinateur des systèmes logiques : CASSANDRE", Thèse d'état, USMG, Grenoble April 1973.

[MER 85] **J. Mermet**
" Several Steps Toward A Circuits Integrated System : CASCADE", Proc. 7 th Int. Symp. on Computer Hardware Description Languages and their Applications (CHDL' 85), Aug. 1985, pp.226-237.

[MIK 86] **C. Mike, R. Palkovic**
" LCC Simulators Speed Development of Synchronous Hardware", Computer Design, March 1, 1986,pp. 87-92.

[MOA 81] **M. Moala et al**
" A design tool for the multilevel description and simulation of systems of interconnected modules", 3 rd Annual Symp. on Computer Architecture, Tampa, 1981.

[MOR 85] **J.D. Morison et al**
" The Design Rationale of ELLA, A Hardware Design and Description Language", Proc. 7 th Int. Symp. on Computer Hardware Description Languages and their Applications (CHDL' 85), Aug. 1985, pp. 303-320.

[MUR 86] **S. Murai et al**
" Logic Simulation Programs", in Logic Design and Simulation, Advances in CAD for VLSI, Vol. 2, Edit by E. Hörbst, North-Holland, 1986.

[NAG 75] **L. W. Nagel**
" SPICE2 A computer program to simulate semiconductor circuits", University of California, Berkely, ERL Memo No. ERL-M520, May 19

[NEW 79] **A. R. Newton**
" Techniques for the Simulation of Large Scale Integrated Circuits", IEEE Trans. Circuits & Systems, Vol. CAS-26, Sept. 1979, pp. 741-749.

[NEW 81] **A.R.Newton**
" Timing, logic and mixed-mode simulation for large MOS integrated Circuits", in Computer Design Aids for VLSI circuits , the Netherlands : Sijithoff and Noordhoof, 1981, pp. 175-240.

[NEW 84] **A. R. Newton, A.L. Sangiovanni-Vincentelli**
"Relaxation-Based Electrical Simulation", IEEE Trans. Computer-Aided Design, Vol. CAD-3, No. 4, Oct 1984.

[OLI 82] **V. Olive , D. Rouquier**
" Design of an Integrated bus arbiter", Congres AFCET, Informatique, Nov 1982.

[PAR 87] **A. C. Parker et al**
"Automating the VLSI Design process Using Expert Systems and Silicon Compilation", Proceeding IEEE, Vol 75, No 6, 1987.

[PAW 81] **A. Pawlak, J. Jezewski**
"MODLAN - A Language for multilevel Description and Modelling of Digital Systems", Proc. 5 th Int. Symp. on Computer Hardware Description Languages and their Applications (CHDL'81), 1981, pp. 79-93.

[PIL 83] **R.Piloty et al**
"CONLAN report", Lecture Notes in Computer Science, Edited by G.Goss et J.Hartmanis, Springer-Verlag, 1983.

[PIL 85] **R. Piloty, D. Borrione**
"The CONLAN project : concepts, Implementations and Applications", Computer, Feb. 1985 , pp. 81-92.

[POI 86] **M. Poize, P. Venier, F. Mohen**
" Editeur graphique et interface utilisateurs", note interne, CNET Grenoble, 1986.

[PRA 85] " The ELLA User Manual", Praxis System Ltd., 1985.

[RAB 79] **N.G.Rabbat et al**
" A Multilevel Newton Algorithm with Macromodeling and Latency for the Analysis of Large Scale Nonlinear Circuits in the Time Domain", IEEE Trans. Circuits & Systems, Vol. CAS-26, Sept. 1979, pp. 733-741.

[RAM 86] **F. J. Rammig**
" Mixed Level Modelling and Simulation of VLSI Systems", in Logic Design and Simulation, Advances in CAD for VLSI, Vol. 2, Edit by E. Hörbst, North-Holland, 1986.

[RAM 86] **F.J. Rammig**
" Modelling and Simulation Concepts of DACAPO II", Dosis GmbH, 1986.

[REY 80] **P.H. Reynaert et al**
" DIANA : A mixed-mode Simulator with a Hardware Description Language for Hierarchical design of VLSI", Proc. IEEE Int. Conf. Circuits & Computers, Oct 1980, pp.356-360.

[ROG 86] **L. Roger, E. Marchner, and M . Shahdad**
" VHDL the language ", IEEE Design & TEST, April 1986, pp. 28-41.

[SAK 80] **K. A. Sakallah, S.W.Director**
" An activity-Directed Circuit Simulation Algorithm", Proc. IEEE Int. Conf. Circuits & Computers, 1980, pp. 1032-1035.

[SAK 85] **K. A. Sakallah, S.W.Director**
" SAMSON2 : An Event Driven VLSI Circuit Simulator", IEEE Trans. Computer Aided Design, Vol. CAD-4, No. 4, 1985.

[SAS 83] **T. Sasaki et al**
" HAL : A block level hardware logic simulator", Proc. 20 th Design Automation Conference, 1983, pp. 150-156.

[SCH 72] **B.H. Scheff, S. P. Young**
" Gate-Level Logic Simulation", in Design Automation of Digital Systems, Edit. M. A. Breuer, 1972.

[SCH 85] "Projet ESPRIT Tashe No. 2 - CVS WP2 ".

[SES 62] **S. Seshu , D.M. Freeman**
" The Diagnosis od Asynchronous Sequential Switching Systems", IEEE Trans. Elec. Computers, Vol. 11, Aug 1962, pp. 459-465.

[SHA 85] **M . Shahdad et Al**
"VHSIC Hardware Description Language", Computer, FEB . 1985 , pp. 94-103.

[SHI 79] **S. G. Shiva**
"Computer Hardware Description Languages- A Tutorial", Proceedings of IEEE, Vol. 67, No. 12, Desc. 1979, pp.1605-1615.

[SIG 85] " SISIM User Manual", Sigride, 1985.

[SIL 83] "Helix 1.3 Reference Manual", Silvar-Lisco doc. No. M-025-1. Oct. 1983.

[SLU 84] **E. Slutz, G. Okita, J. Wiseman**
" Block Description Language (BDL) : A Structural Description Language", Proc. 21 th Design Automation Conference 1984, pp. 81-85.

[SMI 87] **S. P. Smith et al**
" Demand Driven Simulation : BACKSIM", Proc. 24 th Design Automation Conference, 1987.

[SZY 70] **S.A. Szygenda et al**
" A model and Implementation of a universal time delay simulator for large digital nets", Spring Joint Computer Conf. 1970.

[SZY 72] **S.A. Szygenda**
" TEGAS - Anatomy of a general purpose test generation and simulation system", Proc. 9 th Design Automation Workshop, June 1972.

[SZY 75] **S. A. Szygenda, E.W.Thompson**
" Digital Logic Simulation In a Time-Based, Table-Driven Environment, Part 1. Design Verification", Computer, March 1975.

[TAY 82] **R. Taylor, P. Wilson**
" Process-oriented language meets demands of distributed process", Electronics, November 30, 1982, pp. 89-95.

[THO 75] **E.W. Thompson, S.A.Szyegnda**
" Three levels of accuarcy for the simulation of different fault types in digital circuits", Proc. 12 th Design Automation Conference, 1975.

[THO 80] E. Thompson et al
" TEGAS Design Lanugage (TDL) - A system for modular Description and Top-Down/ Bottom-Up Verification of LSI and VLSI", IEEE International Sym. on Circuits ans Systems, 1980, pp. 300-303.

[ULR 65] E. Ulrich
" Time-sequenced Logical Simulation Based on Circuit Delay and Selective Tracing of Active Network Paths", ACM National Conference, 1965, pp. 437-448.

[ULR 69] E.Urlich
" Exclusive Simulation of Activity in Digital Network", Commu. Ass. Comput. Mach. , Feb. 1969, pp. 102-110.

[ULR 72] E. Ulrich et al
" Fault-Test Analysis Technique Based on Logic Simulation", Proc. 9 th Design Automation Workshop, 1972.

[ULR 80] E. Ulrich
" Table Lookup Technique for Fast and Flexible Digital Logic Simulation", Proc. 17 th Design Automation Conference, 1980.

[ULR 82] E. Ulrich , D. Herbert
"Speed and accuracy in digital network simulation based on structural modeling", Proc. 19 th Design Automation Conference, 1982.

[VAN 77] W.M. VanCleemput
" An Hierarchical Language for the Structural Description of Digital Systems", Proc. 14 th Design Automation Conference, 1977,pp.377-385.

[VAN 79] W. M.VanCleemput
" Computer Hardware Description Languages and their Applications", Proc 16 th Design Automation Conference, June 1979.

[VLA 82] **A. Vladimirescu, D.O.Pederson**
" Performance Limits of the CLASSE circuit simulation program", IEEE Int. Symp. Circuits & Systems (ISCAS'82), 1982.

[WAN 87] **L-T Wang et al**
" SSIM : A software Levelized Compiled-Code Simulation", Proc. 24 th Design Automation Conference, 1987.

[WEE 73] **W.T.Weeks et al**
" Algorithms for ASTAP- A network Analysis Program", IEEE Trans. Circuit Theory, Vol. CT-20, Nov. 1973, pp. 628-634.

[WHI 84] **J. White, A.L. Sangiovanni-Vincentelli**
" RELAX2 : A Waveform Relaxation Based Circuit Simulation Program", Proc. IEEE Int. Conf. on Computer Aided Design (ICCAD'84), 1984.

[WIL 79] **P. Wilcox**
" Digital Circuit Simulation using Functional Models", IEEE Int. Symp. Circuits and Systems (ISCAS'79), 1979.

[WIL 82] **Y-P.L. Willie**
"HISDL- A Structural Description Language", Communication of the ACM, Vol. 25, No. 11, Nov. 1982, pp.823-830.

[ZIM 79] **G. Zimmermann**
" The MIMOLA Desgin System ", 16 th Design Automation Conference, 1979, pp. 53-58.

ANNEXE A : CARTE SYNTAXIQUE DE FIDEL

(1) Fidel ::= Fidel_d "ENDMODEL" { Fidel_d "ENDMODEL" } *

(2) Fidel_d ::= Global_env , Struct_env , Funct_env *

(3) Global_env ::= Global_heading_block global_definition_block
Functional_principal_block *

(4) Global_heading_block ::= "GMODEL" "ID" ";" *

(5) Global_definition_block ::= Global_declaration_part ";"
{ Global_declaration_part ";" }
{ Initialize_part } *

(6) Global_declaration_part ::= "DECLARE" (Local_declaration ,
Stimuli_declaration) *

(7) Local_declaration ::= Type_local Signal_name_decl
{ "," Signal_name_decl } *

(8) Type_local ::= "INTEGER" , "STATE" *

(9) Signal_name_decl ::= "ID" ["(" Trimmer_decl ["," Trimmer_decl] ")"] *

(10) Trimmer_decl ::= Trimmer_option_decl ": " Trimmer_option_decl *

(11) Trimmer_option_decl ::= "ID" , "DEC_NUMBER" *

(12) Stimuli_declaration ::= Clock_declaration , Sequence_declaration *

(13) Clock_declaration ::= "CLOCK" Clock_name { "," Clock_name } *

(14) Clock_name ::= "ID" "(" "DEC_NUMBER" "," "DEC_NUMBER" ","
"DEC_NUMBER" "," "DEC_NUMBER" ")" *

(15) Sequence_declaration ::= "SEQUENCE" Seq_name { "," Seq_name } *

(16) Seq_name ::= "ID" "(" Seq_value { "," Seq_value } ")" *

(17) Seq_value ::= Constant_value ":" "DEC_NUMBER" *

(18) Constant_value ::= "DEC_NUMBER" , "BINARY_NUMBER" ,
"OCTAL_NUMBER" , "HEXA_NUMBER" *

(19) Struct_env ::= Struct_heading_block Struct_definition_block
Struct_principal_block *

(20) Struct_heading_block ::= "**PSMODEL**" Parameter_heading ,
"**SMODEL**" Non_parameter_heading *

(21) Parameter_heading ::= "ID" "(" Model_formal_parameter_list
Id_parameter ")" ";" *

(22) Non_parameter_heading ::= "ID" "("Model_formal_parameter_list ")" ";" *

(23) Model_formal_parameter_list ::= "ID" { "," "ID" } *

(24) Id_parameter ::= "$" "ID" { "$" "ID" } *

(25) Struct_definition_block ::= Struct_declaration_part ";"
{ Struct_declaration_part ";" } *

(26) Struct_declaration_part ::= "**DECLARE**" (Argument_declaration ,
Local_declaration , Component_declaration) *

(27) Argument_declaration ::= Argument_type Signal_name_decl
{ "," Signal_name_decl } *

(28) Argument_type ::= "**INPUT**" , "**OUTPUT**" , "**INPOUT**" *

(29) Component_declaration ::= "**COMPONENT**" Component_name_decl
{ "," Component_name_decl } *

(30) Component_name_decl ::= Component_instance_decl
{ "," Component_instance_decl } ":"
Component_type *

(31) Component_instance_decl ::= "ID" ["(" Trimmer_comp_decl
["," Trimmer_comp_decl] ")"] *

(32) Trimmer_comp_decl ::= Trimmer_option_comp_decl [":"
Trimmer_option_comp_decl] *

(33) Trimmer_option_comp_decl ::= "ID" , "DEC_NUMBER" *

(34) Component_type ::= Primitif_type , Funct_type *

(35) Primitif_type ::= Switch_type "(" Transistor_type "," Transistor_scale ")" *

(36) Switch_type ::= "**SWITCH**" , "**PULLUP**" , "**PULLDOWN**" *

(37) Transistor_type ::= "**NMOS**" , "**PMOS**" , "**CMOS**" *

(38) Transistor_scale ::= "DEC_NUMBER" , "ID" *

(39) Funct_type ::= "ID" ["(" Id_parameter_value ")"] *

(40) Id_parameter_value ::= "$" Constant_value { "$" Constant_value } *

(41) Struct_principal_block ::= "**STRUCTURAL**" Block "**ENDSTRUCTURAL**" *

(42) Funct_env ::= Function_heading_block Function_definition_block
{ Macro_generation_block } Functional_principal_block *

(43) Function_heading_block ::= "**PFMODEL**" Parameter_heading ,
"**FMODEL**" Non_parameter_heading *

(44) Function_definition_block ::= Function_declaration_part ";"
{ Function_declaration_part ";" }
{ Initialize_part ";" } *

(45) Function_declaration_part ::= "**DECLARE**" (Argument_declaration ,
Local_declaration) *

(46) Initialize_part ::= "**INITIALIZE**" Signal_name_init
{ "," Signal_name_init } "**TO**" Init_value *

(47) Signal_name_init ::= "ID" ["(" Trimmer_init ["," Trimmer_init] ")"] *

(48) Trimmer_init ::= "DEC_NUMBER" [":" "DEC_NUMBER"] *

(49) Init_value ::= Dynamic_value , Constant_value *

(50) Dynamic_value ::= "**RISES**" , "**FALLS**" *

(51) Macro_generation_block ::= "**MACRO**" "ID" ["(" Macro_parameter_list ")"]
Block "**ENDMACRO**" *

(52) Macro_parameter_list ::= "ID" { "," "ID" } *

(53) Functional_principal_block ::= "**FUNCTIONAL**" When_clause ";"
{ When_clause ";" } "**ENDFUNCTIONAL**" *

(54) When_clause ::= "**WHEN**" Clause Block *

(55) Clause ::= Time_clause , Condition *

(56) Time_clause ::= "**TIME**" "**BECOMES**" "DEC_NUMBER" *

(57) Condition ::= Sample_condition_and { ("**OR**" , "|" , "**XOR**")
Sample_condition_and } *

(58) Sample_condition_and ::= Sample_condition { ("**AND**" , "&")
Sample_condition } *

(59) Sample_condition ::= Event_expr ["OCC" "DEC_NUMBER"] *

(60) Event_expr ::= Event { "**AFTER**" "DEC_NUMBER" Event } *

(61) Event ::= Exp1 { Op_relation Exp1 } *

(62) Op_relation ::= "=" , "^=" , "<" , "<=" , ">" , ">=" *

(63) Exp1 ::= Exp2 { ("+" , "-" , "@") Exp2 } *

(64) Exp2 ::= Exp3 { ("*" , "/" , "**MOD**") Exp3 } *

(65) Exp3 ::= Exp4 { ("**SHL**" , "**SHR**" , "**SCL**" , "**SCR**") Exp4 } *

(66) Exp4 ::= Exp5 , ("**NOT**" , "^") Exp5 *

(67) Exp5 ::= Signal_name [(Dynamic_value , "**CHANGES**" ,
"**BECOMES**" Constant_value , Timing_test)] ,
Constant_value , Function , "(" Condition ")" *

(68) Block ::= Mode_statement { Mode_statement } *

(69) Mode_statement ::= Statement , Mode_operator Block "**ENDMODE**" *

(70) Mode_operator ::= "**TC**" , "**OC**" , "**SM**" , "**NS**" *

(71) Statement ::= Check_block , Simple_statement *

(72) Check_block ::= If_block , Seclect_block , While_block , For_block ,
Par_block , Seq_block , Within_block *

(73) If_block ::= "**IF**" Condition "**THEN**" Block ["**ELSE**" Block] "**ENDIF**" *

(74) Select_block ::= "**SELECT**" Signal_name Block { ",," Block } "**ENDSELECT**" *

(75) While_block ::= "**WHILE**" Condition "**DO**" Loop_block_list "**ENDWHILE**" *

(76) For_block ::= "**FOR**" "ID" "=" For_var ("**UPTO**" , "**DOWNTO**") For_var
["STEP" For_var] "DO" Loop_block_list "ENDFOR" *

(77) For_var ::= Exp1 *

(78) Loop_block_list ::= Loop_block { Loop_block } *

(79) Loop_block ::= For_block , While_block , If_block , Select_block , Seq_block ,
Simple_statement *

(80) Par_block ::= "**PAR**" Block_Par { Block_par } "**ENDPAR**" *

(81) Block_par ::= If_block , Select_block , Par_block , Seq_block ,
Simple_statement *

(82) Seq_block ::= "**SEQ**" Block "**ENDSEQ**" *

(83) Within_block ::= ("**WITHIN**" , "**WITHINP**") Trimmer "**TO**" Trimmer
Block "**ENDWITHIN**" *

(84) Simple_statement ::= Event_statement , Assignement_statement ,
Connect_statement , Message_statement ,
Test_statement , Branche_statement ,
Macro_statement , Schedule_statement , "**NOOP**" *

(85) Event_statement ::= "**EVENT**" "ID" { "," "ID" } *

(86) Assignement_statement ::= "**MAKE**" Assignement_var "=" Condition *

(87) Assignement_var ::= Signal_name { ("," , "@") Signal_name } *

(88) Connect_statement ::= "**CONNECT**" Connection_element ","
Connection_element { "," Connection_element } *

(89) Connection_element ::= Component_instance "." Connection_name ,
"**SUPPLY**" , "**GROUND**" *

(90) Connection_name::=Connection_instance ,"**GATE**" , "**SOURCE**" ,"**DRAIN**" *

(91) Component_instance ::="ID" ["(" Trimmer_Comp [","Trimmer_comp]")" *

(92) Trimmer_comp ::= Trimmer_option_comp [":" Trimmer_option_comp] *

(93) Trimmer_option_comp ::= Expl *

(94) Connection_instance ::= "ID" ["(" Trimmer_connect
["," Trimmer_connect] ")"] *

(95) Trimmer_connect ::= Trimmer_option_connect
[":" Trimmer_option_connect] *

(96) Trimmer_option_connect ::= Expl *

(97) Schedule_statement ::= "SCHEDULE" Schedule_name
{ "," Schedule_name } *

(98) Schedule_name ::= Schedule_comp_name "." Schedule_connect_name *

(99) Schedule_comp_name ::= "ID" ["(" Trimmer_schedule_comp
["," Trimmer_schedule_comp] ")"] *

(100) Trimmer_schedule_comp ::= "DEC_NUMBER" [":" "DEC_NUMBER"] *

(101) Schedule_connect_name ::= "ID" ["(" Trimmer_schedule_connect
["," Trimmer_schedule_connect] ")"] *

(102) Trimmer_schedule_connect ::= "DEC_NUMBER" [":" "DEC_NUMBER"] *

(103) Message_statement ::= "REPORT" "THAT" """ CARACTER_STRING """
[Signal_name] *

(104) Test_statement ::= Trace_statement , Examine_statement *

(105) Trace_statement ::= "TRACE" ("ON" , "OFF") Signal_name
{ "," Signal_name } *

(106) Examine_statement ::= "EXAMINE" Signal_name { "," Signal_name } *

(107) Brance_statement ::= "**EXIT**" ["MODEL"] *

(108) Macro_statement ::= "**GENMACRO**" "ID" ["(" Macro_parameter_map ")] *

(109) Macro_parameter_map ::= "ID" { "," "ID" } *

(110) Function ::= Logic_function , Shift_function , Other_function *

(111) Logic_function ::= ("**FAND**" , "**FLOR**" ," **FXOR**") "(" Signal_name
"," Signal_name { "," Signal_name } ")" *

(112) Shift_function ::= ("**FSHL**" , "**FSHR**" , "**FSCL**" , "**FSCR**") "(" Signal_name
{ "," Signal_name } *

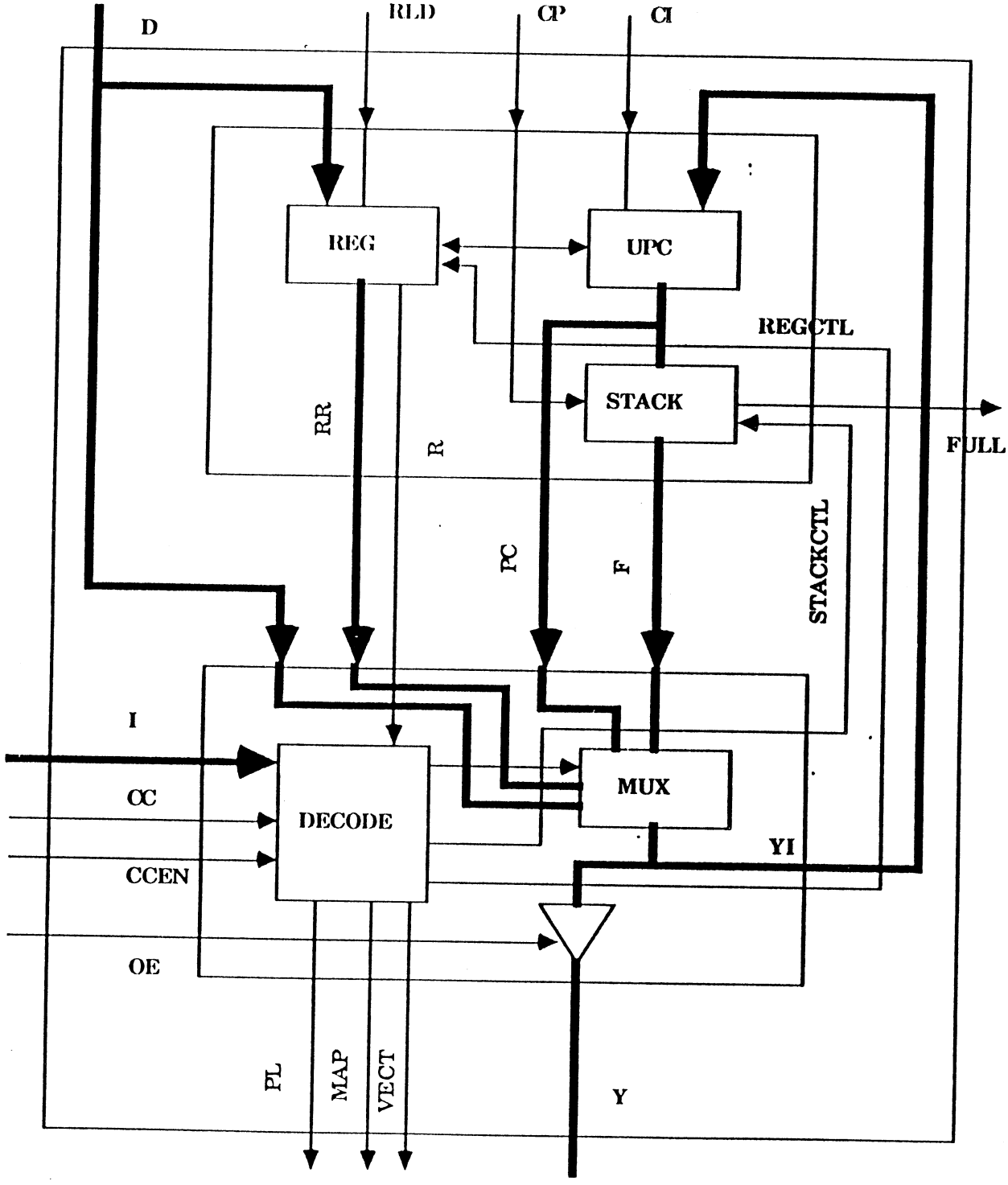
(113) Other_function ::= ("**DECODE**" , "**POSL**" , "**POSR**" , "**EVEN**" , "ODD" ,
"**COUNT**") "(" Signal_name ")" *

(114) Timing_test ::= "[" "DEC_NUMBER" "," "DEC_NUMBER" "]" *

ANNEXE B : EXEMPLE FIDEL + EPILOG

L'exemple qu'on le présente ici est le circuit AM2910 (benche-mark exemple).

B.1 Décomposition fonctionnelle :



B.2 DESCRIPTION FIDEL

```
(*-------------------------------------------------------------------------*)
(*                                                                         *)
(*     AM2910 - MICROCODE-CONTROLLER MODELLED IN FIDEL.V3                  *)
(*                                                                         *)
(*                BY H.EL TAHAWY DEC/86                                    *)
(*                                                                         *)
(*-------------------------------------------------------------------------*)

FMODEL INSTPLA (INIT,I,CC,CCEN,R,PL,MAP,VECT,REGCTL,STACKCTL,MUXCTL) ;

DECLARE INPUT INIT,I ( 0 : 3 ) , CC , CCEN , R ;

DECLARE OUTPUT PL , MAP , VECT , REGCTL ( 0 : 1 ) , STACKCTL ( 0 : 1 ) ;

DECLARE OUTPUT MUXCTL ( 0 : 2 ) ;

DECLARE STATE PASS , IND ( 0 : 1 ) ;

FUNCTIONAL

   WHEN INIT BECOMES 1

        MAKE REGCTL = 1

        MAKE MUXCTL = 0

        MAKE IND = 0

        MAKE STACKCTL = 2 ;

   WHEN I CHANGES OR CC CHANGES OR CCEN CHANGES OR R CHANGES

    MAKE PASS = CCEN OR NOT CC
    MAKE PL = 1
    MAKE MAP = 0
    MAKE VECT = 0

    SELECT I(0:3)

       (*  JUMP ZERO *)

       MAKE REGCTL = 1
       MAKE MUXCTL = 0
       MAKE STACKCTL = 3  ,,

       (* COND JSP PL *)

       MAKE REGCTL = 1
       IF PASS = 1 THEN  MAKE MUXCTL = 1
                         MAKE STACKCTL = 0

                   ELSE MAKE MUXCTL = 4
                         MAKE STACKCTL = 2
```

```
(* JUMP MAP *)

MAKE PL = 0
MAKE MAP = 1
MAKE REGCTL = 1
MAKE MUXCTL = 1
MAKE STACKCTL = 2  ,,

(* COND JUMP PL *)

MAKE REGCTL = 1
IF PASS = 1 THEN MAKE MUXCTL = 1
            ELSE MAKE MUXCTL = 4
ENDIF
MAKE STACKCTL = 2 ,,

(* PUSH/COND LD CNTR *)

 IF PASS = 1 THEN MAKE REGCTL = 2
             ELSE MAKE REGCTL = 1
 ENDIF
 MAKE MUXCTL = 4
 MAKE STACKCTL = 0 ,,

 (* COND JSP R/PL *)

 MAKE REGCTL = 1
 IF PASS = 1 THEN MAKE MUXCTL = 1
             ELSE MAKE MUXCTL = 2
 ENDIF
 MAKE STACKCTL = 0  ,;

 (* COND JUMP VECT *)

 MAKE PL = 0
 MAKE VECT = 1
 MAKE REGCTL = 1
 IF PASS = 1 THEN MAKE MUXCTL = 1
             ELSE MAKE MUXCTL = 4
 ENDIF
 MAKE STACKCTL = 2  ,,

 (* COND JUMP R/PL *)

 MAKE REGCTL = 1
 IF PASS = 1 THEN MAKE MUXCTL = 1
             ELSE MAKE MUXCTL = 2
 ENDIF
 MAKE STACKCTL = 2  ,,

 (* REPEAT LOOP/CNTR <> 0 *)

 IF R = 1 THEN MAKE REGCTL = 1
               MAKE MUXCTL = 4
               MAKE STACKCTL = 1
```

```
          ELSE MAKE REGCTL = 0
               MAKE MUXCTL = 3
               MAKE STACKCTL = 2
ENDIF ,,

(* REPEAT PL/CNTR <> 0 *)

MAKE STACKCTL = 2
IF R = 1 THEN MAKE REGCTL = 1
              MAKE MUXCTL = 4
          ELSE MAKE REGCTL = 0
               MAKE MUXCTL = 1
ENDIF  ,,

(* COND RTN *)

MAKE REGCTL = 1
IF PASS = 1 THEN MAKE MUXCTL = 3
                 MAKE STACKCTL = 1
            ELSE MAKE MUXCTL = 4
                 MAKE STACKCTL = 2
ENDIF ,,

(* COND JUMP PL & POP *)

MAKE REGCTL = 1
IF PASS = 1 THEN MAKE MUXCTL = 1
                 MAKE STACKCTL = 1
            ELSE MAKE MUXCTL = 4
                 MAKE STACKCTL = 2
ENDIF  ,,

(* LD CNTR & CONT *)

MAKE REGCTL = 2
MAKE MUXCTL = 4
MAKE STACKCTL = 2 ,,

(* TEST AND LOOP *)

MAKE REGCTL = 1
IF PASS = 1 THEN MAKE MUXCTL = 4
                 MAKE STACKCTL = 1
            ELSE MAKE MUXCTL = 3
                 MAKE STACKCTL = 2
ENDIF  ,,

(* CONTINUE *)

MAKE REGCTL = 1
MAKE MUXCTL = 4
MAKE STACKCTL = 2  ,,

(* THREE WAY BRANCH *)

MAKE IND(0) =  PASS
MAKE IND(1) = R
```

```
        SELECT IND(0:1)

                MAKE REGCTL = 0
                MAKE MUXCTL = 3
                MAKE STACKCTL = 2  ,,

                MAKE REGCTL = 0
                MAKE MUXCTL = 4
                MAKE STACKCTL = 1  ,,

                MAKE REGCTL = 1
                MAKE MUXCTL = 1
                MAKE STACKCTL = 1 ,,

                MAKE REGCTL = 1
                MAKE MUXCTL = 4
                MAKE STACKCTL = 1

        ENDSELECT

    ENDSELECT ;

ENDFUNCTIONAL

ENDMODEL


FMODEL MUX (INIT,D,RR,F,PC,MUXCTL,YI) ;

DECLARE INPUT INIT , D ( 0 : 11 ) , RR ( 0 : 11 ) , F ( 0 : 11 ) ;

DECLARE INPUT PC ( 0 : 11 ) , MUXCTL ( 0 : 2 ) ;

DECLARE OUTPUT YI ( 0 : 11 ) ;

FUNCTIONAL

   WHEN INIT BECOMES 1

        MAKE YI = 0 ;

   WHEN D CHANGES OR RR CHANGES OR F CHANGES OR PC CHANGES

        OR MUXCTL CHANGES

        WITHIN 13 TO 13

           SELECT MUXCTL(0:2)

                  MAKE YI = 0 ,,

                  MAKE YI = D ,,

                  MAKE YI = RR ,,

                  MAKE YI = F ,,
```

```
                MAKE YI = PC ,,

                NOOP ,,

                NOOP ,,

                NOOP

            ENDSELECT

        ENDWITHIN ;

ENDFUNCTIONAL

ENDMODEL


FMODEL OUTBUFF (OE,YI,Y) ;

DECLARE INPUT OE , YI ( 0 : 11 ) ;

DECLARE OUTPUT Y ( 0 : 11 ) ;

FUNCTIONAL

   WHEN OE CHANGES OR YI CHANGES

        IF OE = 1 THEN MAKE Y = #BZ
                  ELSE MAKE Y = YI
        ENDIF ;

ENDFUNCTIONAL

ENDMODEL


FMODEL UPC (INIT,CP,CI,YI,PC) ;

DECLARE INPUT INIT , CP , CI , YI ( 0 : 11 ) ;

DECLARE OUTPUT PC ( 0 : 11 ) ;

FUNCTIONAL

   WHEN INIT BECOMES 1

        MAKE PC = 0 ;

   WHEN CP RISES

        WITHIN 30 TO 30

                IF CI = 1 THEN MAKE PC = YI + 1

                          ELSE MAKE PC = YI
```

```
                ENDIF

        ENDWITHIN ;

ENDFUNCTIONAL

ENDMODEL


FMODEL REG(INIT,D,RLD,CP,REGCTL,R,RR) ;

DECLARE INPUT INIT , D ( 0 : 11 ) , RLD , CP , REGCTL ( 0 : 1 ) ;

DECLARE OUTPUT R , RR ( 0 : 11 ) ;

FUNCTIONAL

    WHEN INIT BECOMES 1

        MAKE RR = 0 ;

    WHEN CP RISES

        IF RLD = 0 THEN MAKE RR = D
                   ELSE SELECT REGCTL(0:1)

                            WITHIN 30 TO 30 MAKE RR = RR - 1 ENDWITHIN ,,

                            NOOP ,,

                            WITHIN 30 TO 30 MAKE RR = D ENDWITHIN ,,

                            NOOP

                        ENDSELECT

        ENDIF

        WITHIN 30 TO 30

                IF RR = 0 THEN MAKE R = 1
                          ELSE MAKE R = 0

                ENDIF
        ENDWITHIN ;

ENDFUNCTIONAL

ENDMODEL


FMODEL STACK(INIT,CP,PC,STACKCTL,F,FULL) ;

DECLARE INPUT INIT , CP , PC ( 0 : 11 ) , STACKCTL ( 0 : 1 ) ;
```

```
DECLARE OUTPUT F ( 0 : 11 ) , FULL ;
DECLARE STATE STACK ( 1 : 5 , 0 : 11 ) ,STACKBUFF ( 0 : 11 ) ;
DECLARE INTEGER STACKPTR ;
FUNCTIONAL
    WHEN INIT BECOMES 1
        MAKE STACKPTR = 0 ;
    WHEN CP RISES
        SELECT STACKCTL(0:1)
            MAKE STACKPTR = STACKPTR + 1
            IF STACKPTR > 5 THEN MAKE STACKPTR = 5
                            ELSE MAKE STACK(STACKPTR,0:11) = PC
                                 MAKE STACKBUFF = PC
            ENDIF ,,
            IF STACKPTR ^= 0 THEN MAKE STACKBUFF = STACK(STACKPTR,0:11)
                                  MAKE STACKPTR = STACKPTR - 1
            ENDIF ,,
            IF STACKPTR = 0 THEN MAKE STACKBUFF = #BX
                            ELSE MAKE STACKBUFF = STACK(STACKPTR,0:11)
            ENDIF ,,
            MAKE STACKPTR = 0
        ENDSELECT
        WITHIN 30 TO 30 MAKE F = STACKBUFF ENDWITHIN ;
ENDFUNCTIONAL
ENDMODEL
```

B.3 DESCRIPTION EPILOG :

```
ESSAI DE SIMULER AM2910
**BIBLIO
*EFFACER
AM2910
FIN
$---------------------------
$ DESCRIPTION DE STRUCTURE
$---------------------------
**DESCRIPTION
*TOPOLOGIE
AM2910(INIT,I#0:3,CC,CCEN,OE,RLD,CI,CP,D#0:11-PL,MAP,VECT,Y#0:11)
ET1-INSTPLA(INIT,I#0:3,CC,CCEN,R-PL,MAP,VECT,REGCTL#0:1,STACTL#0:1,MUCTL#0:2)
ET2-MUX(INIT,D#0:11,RR#0:11,F#0:11,PC#0:11,MUCTL#0:2-YI#0:11)
ET3-OUTBUFF(OE,YI#0:11-Y#0:11)
ET4-UPC(INIT,CP,CI,YI#0:11-PC#0:11)
ET5-REG(INIT,D#0:11,RLD,CP,REGCTL#0:1-R,RR#0:11)
ET6-STACK(INIT,CP,PC#0:11,STACTL#0:1-F#0:11,FULL)
FIN
$---------------------------
$ COMMANDE DE SIMULATION
$---------------------------
**COMMANDE
LOGIQUE   $ TYPE DE SIMULATION
AM2910
HMAX=11600 $ TEMPS MAXIMAL DE SIMULATION
$---------------------------
$  PARTIE DE DECLARATION
$---------------------------
DECLARATION
HORLOGES
CP=0,11600,0,100,100
VECTEURS
V0=B0000
V1=B1000
V2=B0100
V3=B1100
V4=B0010
V5=B1010
V6=B0110
V7=B1110
V8=B0001
V9=B1001
V10=B0101
V11=B1101
V12=B0011
V13=B1011
V14=B0111
V15=B1111
T0=B000000000000
T1=B000100000000
T2=B001000000000
T3=B001100000000
T4=B000000000011
T5=B010000000000
```

```
T6=B000000000010
T7=B010100000000
T8=B011000000000
T9=B000000000100
T10=B011100000000
FIN
$---------------------------------------
$ DEMANDE DE VISUALISATION DES SIGNAUX
$---------------------------------------
VISUALISATION
CP,I#0:3,D#11:0,Y#11:0,YI#11:0,PC#11:0,F#11:0,RR#11:0,CC,CCEN,RLD, ;
MUCTL#0:2,STACTL#0:1,REGCTL#0:1,PL,MAP,VECT
$---------------------------------
$ DEBUT DE SCENARIO DE SIMULATION
$---------------------------------
SIMULATION
GS
HEURE=0
INIT=1
HEURE=1
INIT=0
HEURE=115
OE=0
CI=1
RLD=0
CC=1
CCEN=0
I#0:3=V7
D#11:0=T0
HEURE=315
RLD=1
HEURE=515
I#0:3=V14
HEURE=915
CC=0
I#0:3=V1
D#11:0=T1
HEURE=1115
I#0:3=V14
D#11:0=T0
HEURE=1515
I#0:3=V3
D#11:0=T2
HEURE=1715
I#0:3=V14
HEURE=2115
I#0:3=V2
D#11:0=T3
HEURE=2315
I#0:3=V14
D#11:0=T0
HEURE=2915
I#0:3=V4
D#11:0=T4
HEURE=3115
I#0:3=V14
D#11:0=T0
```

```
   176- **EDITION
   177- AM2910
   178- *SYMBOLIQUE
   179- B=CP
   180- HD=I#3:0
   181- HD=D#11:0
   182- HD=Y#11:0
1EPILOG2 -V2.5- (10.09.86) ESSAI DE SIMULER AM2910 Le  7/10/87 Page     6
----------------------------
```

```
    HEURE  C IIII DDDDDDDDDDDD YYYYYYYYYYYY
           P #### ############ ############
             3210 119876543210 119876543210
                  10           10
```

```
      0 X XXXX XXXXXXXXXXX XXXXXXXXXXX
      0 0 XXXX XXXXXXXXXXX          000
    100 1 XXXX XXXXXXXXXXX          000
    115 1    7          000         000
    200 0    7          000         000
    300 1    7          000         000
    315 1    7          000         000
    330 1    7          000         000
    400 0    7          000         000
    500 1    7          000         000
    515 1    E          000         000
    528 1    E          000         001
    600 0    E          000         001
    700 1    E          000         001
    730 1    E          000         001
    743 1    E          000         002
    800 0    E          000         002
    900 1    E          000         002
    915 1    1          100         002
    928 1    1          100         100
    930 1    1          100         100
   1000 0    1          100         100
   1100 1    1          100         100
   1115 1    E          000         100
   1128 1    E          000         003
   1130 1    E          000         003
   1143 1    E          000         101
   1200 0    E          000         101
   1300 1    E          000         101
   1330 1    E          000         101
   1343 1    E          000         102
   1400 0    E          000         102
   1500 1    E          000         102
   1515 1    3          200         102
   1528 1    3          200         200
```

1530	1	3	200	200
1600	0	3	200	200
1700	1	3	200	200
1715	1	E	200	200
1728	1	E	200	103
1730	1	E	200	103
1743	1	E	200	201
1800	0	E	200	201
1900	1	E	200	201
1930	1	E	200	201

HEURE	C	IIII	DDDDDDDDDDDD	YYYYYYYYYYYY
	P	####	############	############
		3210	119876543210	119876543210
			10	10

1943	1	E	200	202
2000	0	E	200	202
2100	1	E	200	202
2115	1	2	300	202
2128	1	2	300	300
2130	1	2	300	300
2200	0	2	300	300
2300	1	2	300	300
2315	1	E	000	300
2328	1	E	000	203
2330	1	E	000	203
2343	1	E	000	301
2400	0	E	000	301
2500	1	E	000	301
2530	1	E	000	301
2543	1	E	000	302
2600	0	E	000	302
2700	1	E	000	302
2730	1	E	000	302
2743	1	E	000	303
2800	0	E	000	303
2900	1	E	000	303
2915	1	4	003	303
2930	1	4	003	303
2943	1	4	003	304
3000	0	4	003	304
3100	1	4	003	304
3115	1	E	000	304
3130	1	E	000	304
3143	1	E	000	305
3200	0	E	000	305
3300	1	E	000	305

3330	1	E	000	305
3343	1	E	000	306
3400	0	E	000	306
3500	1	E	000	306
3530	1	E	000	306
3543	1	E	000	307
3600	0	E	000	307
3700	1	E	000	307
3715	1	8	000	307
3728	1	8	000	304
3730	1	8	000	304
3800	0	8	000	304
3900	1	8	000	304

HEURE	C	IIII	DDDDDDDDDDDD	YYYYYYYYYYYY
	P	####	############	############
		3210	119876543210	119876543210
			10	10

3915	1	E	000	304
3928	1	E	000	308
3930	1	E	000	308
3943	1	E	000	305
4000	0	E	000	305
4100	1	E	000	305
4130	1	E	000	305
4143	1	E	000	306
4200	0	E	000	306
4300	1	E	000	306
4330	1	E	000	306
4343	1	E	000	307
4400	0	E	000	307
4500	1	E	000	307
4515	1	8	000	307
4528	1	8	000	304
4530	1	8	000	304
4600	0	8	000	304
4700	1	8	000	304
4715	1	E	000	304
4728	1	E	000	308
4730	1	E	000	308
4743	1	E	000	305
4800	0	E	000	305
4900	1	E	000	305
4930	1	E	000	305
4943	1	E	000	306
5000	0	E	000	306
5100	1	E	000	306

5130	1	E	000	306
5143	1	E	000	307
5200	0	E	000	307
5300	1	E	000	307
5315	1	8	000	307
5328	1	8	000	304
5330	1	8	000	304
5400	0	8	000	304
5500	1	8	000	304
5515	1	E	000	304
5528	1	E	000	308
5530	1	E	000	308
5543	1	E	000	305
5600	0	E	000	305
5700	1	E	000	305
5730	1	E	000	305

HEURE	C	IIII	DDDDDDDDDDDD	YYYYYYYYYYYY
	P	####	############	############
		3210	119876543210	119876543210
			10	10

5743	1	E	000	306
5800	0	E	000	306
5900	1	E	000	306
5915	1	E	000	306
5930	1	E	000	306
5943	1	E	000	307
6000	0	E	000	307
6100	1	E	000	307
6115	1	C	400	307
6130	1	C	400	307
6143	1	C	400	308
6200	0	C	400	308
6300	1	C	400	308
6315	1	5	000	308
6328	1	5	000	000
6330	1	5	000	000
6343	1	5	000	400
6400	0	5	000	400
6500	1	5	000	400
6515	1	E	000	400
6528	1	E	000	309
6530	1	E	000	309
6543	1	E	000	401
6600	0	E	000	401
6700	1	E	000	401
6730	1	E	000	401

6743	1	E	000	402
6800	0	E	000	402
6900	1	E	000	402
6930	1	E	000	402
6943	1	E	000	403
7000	0	E	000	403
7100	1	E	000	403
7115	1	C	002	403
7130	1	C	002	403
7143	1	C	002	404
7200	0	C	002	404
7300	1	C	002	404
7315	1	9	500	404
7328	1	9	500	500
7330	1	9	500	500
7400	0	9	500	500
7500	1	9	500	500
7530	1	9	500	500
7600	0	9	500	500

HEURE	C	IIII	DDDDDDDDDDDD	YYYYYYYYYYYY
	P	####	############	############
		3210	119876543210	119876543210
			10	10

7700	1	9	500	500
7715	1	E	000	500
7728	1	E	000	501
7730	1	E	000	501
7800	0	E	000	501
7900	1	E	000	501
7930	1	E	000	501
7943	1	E	000	502
8000	0	E	000	502
8100	1	E	000	502
8115	1	A	000	502
8128	1	A	000	309
8130	1	A	000	309
8200	0	A	000	309
8300	1	A	000	309
8315	1	E	000	309
8328	1	E	000	503
8330	1	E	000	503
8343	1	E	000	30A
8400	0	E	000	30A
8500	1	E	000	30A
8530	1	E	000	30A
8543	1	E	000	30B

```
      8600 0    E          000          30B
      8700 1    E          000          30B
      8730 1    E          000          30B
      8743 1    E          000          30C
      8800 0    E          000          30C
      8900 1    E          000          30C
      8915 1    6          600          30C
      8928 1    6          600          600
      8930 1    6          600          600
      9000 0    6          600          600
      9100 1    6          600          600
      9115 1    E          000          600
      9128 1    E          000          30D
      9130 1    E          000          30D
      9143 1    E          000          601
      9200 0    E          000          601
      9300 1    E          000          601
      9330 1    E          000          601
      9343 1    E          000          602
      9400 0    E          000          602
      9500 1    E          000          602
      9515 1    4          004          602
1EPILOG2 -V2.5- (10.09.86) ESSAI DE SIMULER AM2910 Le  7/10/87 Page   11
-----------------------------

    HEURE  C IIII DDDDDDDDDDDD YYYYYYYYYYYY
           P #### ############ ############
             3210 119876543210 119876543210
                  10           10



      9530 1    4          004          602
      9543 1    4          004          603
      9600 0    4          004          603
      9700 1    4          004          603
      9715 1    B          700          603
      9730 1    B          700          603
      9743 1    B          700          604
      9800 0    B          700          604
      9900 1    B          700          604
      9915 1    F          000          604
      9928 1    F          000          000
      9930 1    F          000          000
      9943 1    F          000          603
     10000 0    F          000          603
     10100 1    F          000          603
     10115 1    B          700          603
     10128 1    B          700          700
     10130 1    B          700          700
     10200 0    B          700          700
     10300 1    B          700          700
```

```
10315 1     E           000          700
10328 1     E           000          604
10330 1     E           000          604
10343 1     E           000          701
10400 0     E           000          701
10500 1     E           000          701
10530 1     E           000          701
10543 1     E           000          702
10600 0     E           000          702
10700 1     E           000          702
10715 1     A           000          702
10728 1     A           000          304
10730 1     A           000          304
10800 0     A           000          304
10900 1     A           000          304
10915 1     E           000          304
10928 1     E           000          703
10930 1     E           000          703
10943 1     E           000          305
11000 0     E           000          305
11100 1     E           000          305
11130 1     E           000          305
11143 1     E           000          306
11200 0     E           000          306
11300 1     E           000          306
```



```
----------------------------


    HEURE  C IIII DDDDDDDDDDDD YYYYYYYYYYYY
           P #### ############ ############
             3210 119876543210 119876543210
                  10           10
```

```
11330 1     E           000          306
11343 1     E           000          307
11400 0     E           000          307
11500 1     E           000          307
11515 1     0           000          307
11528 1     0           000          000
11530 1     0           000          000
11600 0     0           000          000
```

```
  183- *SYMBOLIQUE
  184- HD=YI#11:0
  185- HD=PC#11:0
  186- HD=F#11:0
  187- HD=RR#11:0
```

HEURE	YYYYYYYYYYYY IIIIIIIIIIII ############ 119876543210 10	PPPPPPPPPPP CCCCCCCCCCC ############ 119876543210 10	FFFFFFFFFFF ############ 119876543210 10	RRRRRRRRRRR RRRRRRRRRRR ############ 119876543210 10
0	XXXXXXXXXXX	XXXXXXXXXXX	XXXXXXXXXXX	XXXXXXXXXXX
0	000	000	XXXXXXXXXXX	000
100	000	000	XXXXXXXXXXX	000
115	000	000	XXXXXXXXXXX	000
200	000	000	XXXXXXXXXXX	000
300	000	000	XXXXXXXXXXX	000
315	000	000	XXXXXXXXXXX	000
330	000	001	XXXXXXXXXXX	000
400	000	001	XXXXXXXXXXX	000
500	000	001	XXXXXXXXXXX	000
515	000	001	XXXXXXXXXXX	000
528	001	001	XXXXXXXXXXX	000
600	001	001	XXXXXXXXXXX	000
700	001	001	XXXXXXXXXXX	000
730	001	002	XXXXXXXXXXX	000
743	002	002	XXXXXXXXXXX	000
800	002	002	XXXXXXXXXXX	000
900	002	002	XXXXXXXXXXX	000
915	002	002	XXXXXXXXXXX	000
928	100	002	XXXXXXXXXXX	000
930	100	003	XXXXXXXXXXX	000
1000	100	003	XXXXXXXXXXX	000
1100	100	003	XXXXXXXXXXX	000
1115	100	003	XXXXXXXXXXX	000
1128	003	003	XXXXXXXXXXX	000
1130	003	101	XXXXXXXXXXX	000
1143	101	101	XXXXXXXXXXX	000
1200	101	101	XXXXXXXXXXX	000
1300	101	101	003	000
1330	101	102	003	000
1343	102	102	003	000
1400	102	102	003	000
1500	102	102	003	000
1515	102	102	003	000
1528	200	102	003	000
1530	200	103	003	000
1600	200	103	003	000
1700	200	103	003	000
1715	200	103	003	000
1728	103	103	003	000
1730	103	201	003	000
1743	201	201	003	000
1800	201	201	003	000

1900	201	201	003	000
1930	201	202	003	000

HEURE	YYYYYYYYYYYY	PPPPPPPPPPPP	FFFFFFFFFFFF	RRRRRRRRRRRR
	IIIIIIIIIIII	CCCCCCCCCCCC	############	RRRRRRRRRRRR
	############	############	119876543210	############
	119876543210	119876543210	10	119876543210
	10	10		10

1943	202	202	003	000
2000	202	202	003	000
2100	202	202	003	000
2115	202	202	003	000
2128	300	202	003	000
2130	300	203	003	000
2200	300	203	003	000
2300	300	203	003	000
2315	300	203	003	000
2328	203	203	003	000
2330	203	301	003	000
2343	301	301	003	000
2400	301	301	003	000
2500	301	301	003	000
2530	301	302	003	000
2543	302	302	003	000
2600	302	302	003	000
2700	302	302	003	000
2730	302	303	003	000
2743	303	303	003	000
2800	303	303	003	000
2900	303	303	003	000
2915	303	303	003	000
2930	303	304	003	000
2943	304	304	003	000
3000	304	304	003	000
3100	304	304	003	000
3115	304	304	003	000
3130	304	305	304	003
3143	305	305	304	003
3200	305	305	304	003
3300	305	305	304	003
3330	305	306	304	003
3343	306	306	304	003
3400	306	306	304	003
3500	306	306	304	003
3530	306	307	304	003
3543	307	307	304	003
3600	307	307	304	003
3700	307	307	304	003

3715	307	307	304	003
3728	304	307	304	003
3730	304	308	304	003
3800	304	308	304	003
3900	304	308	304	003

HEURE	YYYYYYYYYYYY	PPPPPPPPPPPP	FFFFFFFFFFFF	RRRRRRRRRRRR
	IIIIIIIIIIII	CCCCCCCCCCCC	############	RRRRRRRRRRRR
	############	############	119876543210	############
	119876543210	119876543210	10	119876543210
	10	10		10

3915	304	308	304	003
3928	308	308	304	003
3930	308	305	304	002
3943	305	305	304	002
4000	305	305	304	002
4100	305	305	304	002
4130	305	306	304	002
4143	306	306	304	002
4200	306	306	304	002
4300	306	306	304	002
4330	306	307	304	002
4343	307	307	304	002
4400	307	307	304	002
4500	307	307	304	002
4515	307	307	304	002
4528	304	307	304	002
4530	304	308	304	002
4600	304	308	304	002
4700	304	308	304	002
4715	304	308	304	002
4728	308	308	304	002
4730	308	305	304	001
4743	305	305	304	001
4800	305	305	304	001
4900	305	305	304	001
4930	305	306	304	001
4943	306	306	304	001
5000	306	306	304	001
5100	306	306	304	001
5130	306	307	304	001
5143	307	307	304	001
5200	307	307	304	001
5300	307	307	304	001
5315	307	307	304	001
5328	304	307	304	001
5330	304	308	304	001
5400	304	308	304	001

5500	304	308	304	001
5515	304	308	304	001
5528	308	308	304	001
5530	308	305	304	000
5543	305	305	304	000
5600	305	305	304	000
5700	305	305	304	000
5730	305	306	304	000

HEURE	YYYYYYYYYYYY	PPPPPPPPPPP	FFFFFFFFFFF	RRRRRRRRRRR
	IIIIIIIIIIII	CCCCCCCCCCC	############	RRRRRRRRRRR
	############	############	119876543210	############
	119876543210	119876543210	10	119876543210
	10	10		10

5743	306	306	304	000
5800	306	306	304	000
5900	306	306	304	000
5915	306	306	304	000
5930	306	307	304	000
5943	307	307	304	000
6000	307	307	304	000
6100	307	307	304	000
6115	307	307	304	000
6130	307	308	304	000
6143	308	308	304	000
6200	308	308	304	000
6300	308	308	304	000
6315	308	308	304	000
6328	000	308	304	000
6330	000	309	304	400
6343	400	309	304	400
6400	400	309	304	400
6500	400	309	304	400
6515	400	309	304	400
6528	309	309	304	400
6530	309	401	309	400
6543	401	401	309	400
6600	401	401	309	400
6700	401	401	309	400
6730	401	402	309	400
6743	402	402	309	400
6800	402	402	309	400
6900	402	402	309	400
6930	402	403	309	400
6943	403	403	309	400
7000	403	403	309	400
7100	403	403	309	400
7115	403	403	309	400

7130	403	404	309	400
7143	404	404	309	400
7200	404	404	309	400
7300	404	404	309	400
7315	404	404	309	400
7328	500	404	309	400
7330	500	405	309	002
7400	500	405	309	002
7500	500	405	309	002
7530	500	501	309	001
7600	500	501	309	001

HEURE	YYYYYYYYYYYY	PPPPPPPPPPPP	FFFFFFFFFFFF	RRRRRRRRRRR
	IIIIIIIIIIII	CCCCCCCCCCCC	############	RRRRRRRRRRR
	############	############	119876543210	############
	119876543210	119876543210	10	119876543210
	10	10		10

7700	500	501	309	001
7715	500	501	309	001
7728	501	501	309	001
7730	501	501	309	000
7800	501	501	309	000
7900	501	501	309	000
7930	501	502	309	000
7943	502	502	309	000
8000	502	502	309	000
8100	502	502	309	000
8115	502	502	309	000
8128	309	502	309	000
8130	309	503	309	000
8200	309	503	309	000
8300	309	503	309	000
8315	309	503	309	000
8328	503	503	309	000
8330	503	30A	309	000
8343	30A	30A	309	000
8400	30A	30A	309	000
8500	30A	30A	309	000
8530	30A	30B	304	000
8543	30B	30B	304	000
8600	30B	30B	304	000
8700	30B	30B	304	000
8730	30B	30C	304	000
8743	30C	30C	304	000
8800	30C	30C	304	000
8900	30C	30C	304	000
8915	30C	30C	304	000
8928	600	30C	304	000

8930	600	30D	304	000
9000	600	30D	304	000
9100	600	30D	304	000
9115	600	30D	304	000
9128	30D	30D	304	000
9130	30D	601	304	000
9143	601	601	304	000
9200	601	601	304	000
9300	601	601	304	000
9330	601	602	304	000
9343	602	602	304	000
9400	602	602	304	000
9500	602	602	304	000
9515	602	602	304	000

HEURE	YYYYYYYYYYYY	PPPPPPPPPPPP	FFFFFFFFFFFF	RRRRRRRRRRR
	IIIIIIIIIIII	CCCCCCCCCCC	############	RRRRRRRRRRR
	############	############	119876543210	############
	119876543210	119876543210	10	119876543210
	10	10		10

9530	602	603	304	000
9543	603	603	304	000
9600	603	603	304	000
9700	603	603	304	000
9715	603	603	304	000
9730	603	604	603	004
9743	604	604	603	004
9800	604	604	603	004
9900	604	604	603	004
9915	604	604	603	004
9928	000	604	603	004
9930	000	605	603	004
9943	603	605	603	004
10000	603	605	603	004
10100	603	605	603	004
10115	603	605	603	004
10128	700	605	603	004
10130	700	604	603	003
10200	700	604	603	003
10300	700	604	603	003
10315	700	604	603	003
10328	604	604	603	003
10330	604	701	603	003
10343	701	701	603	003
10400	701	701	603	003
10500	701	701	603	003
10530	701	702	304	003
10543	702	702	304	003

8930	600	30D	304	000
9000	600	30D	304	000
9100	600	30D	304	000
9115	600	30D	304	000
9128	30D	30D	304	000
9130	30D	601	304	000
9143	601	601	304	000
9200	601	601	304	000
9300	601	601	304	000
9330	601	602	304	000
9343	602	602	304	000
9400	602	602	304	000
9500	602	602	304	000
9515	602	602	304	000

HEURE	YYYYYYYYYYYY	PPPPPPPPPPPP	FFFFFFFFFFFF	RRRRRRRRRRRR
	IIIIIIIIIIII	CCCCCCCCCCCC	############	RRRRRRRRRRRR
	############	############	119876543210	############
	119876543210	119876543210	10	119876543210
	10	10		10

9530	602	603	304	000
9543	603	603	304	000
9600	603	603	304	000
9700	603	603	304	000
9715	603	603	304	000
9730	603	604	603	004
9743	604	604	603	004
9800	604	604	603	004
9900	604	604	603	004
9915	604	604	603	004
9928	000	604	603	004
9930	000	605	603	004
9943	603	605	603	004
10000	603	605	603	004
10100	603	605	603	004
10115	603	605	603	004
10128	700	605	603	004
10130	700	604	603	003
10200	700	604	603	003
10300	700	604	603	003
10315	700	604	603	003
10328	604	604	603	003
10330	604	701	603	003
10343	701	701	603	003
10400	701	701	603	003
10500	701	701	603	003
10530	701	702	304	003
10543	702	702	304	003

```
                ###
                012


          0  X  X  X  XXX
          0  X  X  X  000
        100  X  X  X  000
        115  1  0  0  010
        200  1  0  0  010
        300  1  0  0  010
        315  1  0  1  010
        330  1  0  1  010
        400  1  0  1  010
        500  1  0  1  010
        515  1  0  1  001
        528  1  0  1  001
        600  1  0  1  001
        700  1  0  1  001
        730  1  0  1  001
        743  1  0  1  001
        800  1  0  1  001
        900  1  0  1  001
        915  0  0  1  100
        928  0  0  1  100
        930  0  0  1  100
       1000  0  0  1  100
       1100  0  0  1  100
       1115  0  0  1  001
       1128  0  0  1  001
       1130  0  0  1  001
       1143  0  0  1  001
       1200  0  0  1  001
       1300  0  0  1  001
       1330  0  0  1  001
       1343  0  0  1  001
       1400  0  0  1  001
       1500  0  0  1  001
       1515  0  0  1  100
       1528  0  0  1  100
       1530  0  0  1  100
       1600  0  0  1  100
       1700  0  0  1  100
       1715  0  0  1  001
       1728  0  0  1  001
       1730  0  0  1  001
       1743  0  0  1  001
       1800  0  0  1  001
       1900  0  0  1  001
       1930  0  0  1  001
1EPILOG2 -V2.5- (10.09.86) ESSAI DE SIMULER AM2910 Le  7/10/87 Page   21
-----------------------------


    HEURE  C  C  R  MMM
           C  C  I  UUU
```

```
E D CCC
N   TTT
    LLL
    ###
    012

1943 0 0 1 001
2000 0 0 1 001
2100 0 0 1 001
2115 0 0 1 100
2128 0 0 1 100
2130 0 0 1 100
2200 0 0 1 100
2300 0 0 1 100
2315 0 0 1 001
2328 0 0 1 001
2330 0 0 1 001
2343 0 0 1 001
2400 0 0 1 001
2500 0 0 1 001
2530 0 0 1 001
2543 0 0 1 001
2600 0 0 1 001
2700 0 0 1 001
2730 0 0 1 001
2743 0 0 1 001
2800 0 0 1 001
2900 0 0 1 001
2915 0 0 1 001
2930 0 0 1 001
2943 0 0 1 001
3000 0 0 1 001
3100 0 0 1 001
3115 0 0 1 001
3130 0 0 1 001
3143 0 0 1 001
3200 0 0 1 001
3300 0 0 1 001
3330 0 0 1 001
3343 0 0 1 001
3400 0 0 1 001
3500 0 0 1 001
3530 0 0 1 001
3543 0 0 1 001
3600 0 0 1 001
3700 0 0 1 001
3715 0 0 1 110
3728 0 0 1 110
3730 0 0 1 110
3800 0 0 1 110
3900 0 0 1 110
```

HEURE	C C	C	R	MMM
	C C	L	L	UUU
		E	D	CCC
		N		TTT
				LLL
				###
				012
3915	0	0	1	001
3928	0	0	1	001
3930	0	0	1	001
3943	0	0	1	001
4000	0	0	1	001
4100	0	0	1	001
4130	0	0	1	001
4143	0	0	1	001
4200	0	0	1	001
4300	0	0	1	001
4330	0	0	1	001
4343	0	0	1	001
4400	0	0	1	001
4500	0	0	1	001
4515	0	0	1	110
4528	0	0	1	110
4530	0	0	1	110
4600	0	0	1	110
4700	0	0	1	110
4715	0	0	1	001
4728	0	0	1	001
4730	0	0	1	001
4743	0	0	1	001
4800	0	0	1	001
4900	0	0	1	001
4930	0	0	1	001
4943	0	0	1	001
5000	0	0	1	001
5100	0	0	1	001
5130	0	0	1	001
5143	0	0	1	001
5200	0	0	1	001
5300	0	0	1	001
5315	0	0	1	110
5328	0	0	1	110
5330	0	0	1	110
5400	0	0	1	110
5500	0	0	1	110
5515	0	0	1	001
5528	0	0	1	001
5530	0	0	1	001
5543	0	0	1	001
5600	0	0	1	001
5700	0	0	1	001
5730	0	0	1	001

HEURE	C C N	C C E	R L D	MMM UUU CCC TTT LLL ### 012
5743	0	0	1	001
5800	0	0	1	001
5900	0	0	1	001
5915	1	0	1	001
5930	1	0	1	001
5943	1	0	1	001
6000	1	0	1	001
6100	1	0	1	001
6115	1	0	1	001
6130	1	0	1	001
6143	1	0	1	001
6200	1	0	1	001
6300	1	0	1	001
6315	1	0	1	010
6328	1	0	1	010
6330	1	0	1	010
6343	1	0	1	010
6400	1	0	1	010
6500	1	0	1	010
6515	0	0	1	001
6528	0	0	1	001
6530	0	0	1	001
6543	0	0	1	001
6600	0	0	1	001
6700	0	0	1	001
6730	0	0	1	001
6743	0	0	1	001
6800	0	0	1	001
6900	0	0	1	001
6930	0	0	1	001
6943	0	0	1	001
7000	0	0	1	001
7100	0	0	1	001
7115	0	0	1	001
7130	0	0	1	001
7143	0	0	1	001
7200	0	0	1	001
7300	0	0	1	001
7315	0	0	1	100
7328	0	0	1	100
7330	0	0	1	100
7400	0	0	1	100

7500 0 0 1 100
7530 0 0 1 100
7600 0 0 1 100

HEURE	C C E N	C C D	R L	MMM UUU CCC TTT LLL ### 012

7700 0 0 1 100
7715 0 0 1 001
7728 0 0 1 001
7730 0 0 1 001
7800 0 0 1 001
7900 0 0 1 001
7930 0 0 1 001
7943 0 0 1 001
8000 0 0 1 001
8100 0 0 1 001
8115 0 0 1 110
8128 0 0 1 110
8130 0 0 1 110
8200 0 0 1 110
8300 0 0 1 110
8315 0 0 1 001
8328 0 0 1 001
8330 0 0 1 001
8343 0 0 1 001
8400 0 0 1 001
8500 0 0 1 001
8530 0 0 1 001
8543 0 0 1 001
8600 0 0 1 001
8700 0 0 1 001
8730 0 0 1 001
8743 0 0 1 001
8800 0 0 1 001
8900 0 0 1 001
8915 0 0 1 100
8928 0 0 1 100
8930 0 0 1 100
9000 0 0 1 100
9100 0 0 1 100
9115 0 0 1 001
9128 0 0 1 001
9130 0 0 1 001
9143 0 0 1 001
9200 0 0 1 001

```
        9300 0 0 1 001
        9330 0 0 1 001
        9343 0 0 1 001
        9400 0 0 1 001
        9500 0 0 1 001
        9515 0 0 1 001
1EPILOG2 -V2.5- (10.09.86) ESSAI DE SIMULER AM2910 Le  7/10/87 Page   25
-----------------------------


     HEURE  C C R MMM
            C C L UUU
              E D CCC
              N   TTT
                  LLL
                  ###
                  012



        9530 0 0 1 001
        9543 0 0 1 001
        9600 0 0 1 001
        9700 0 0 1 001
        9715 1 0 1 001
        9730 1 0 1 001
        9743 1 0 1 001
        9800 1 0 1 001
        9900 1 0 1 001
        9915 1 0 1 100
        9928 1 0 1 100
        9930 1 0 1 110
        9943 1 0 1 110
       10000 1 0 1 110
       10100 1 0 1 110
       10115 0 0 1 100
       10128 0 0 1 100
       10130 0 0 1 100
       10200 0 0 1 100
       10300 0 0 1 100
       10315 0 0 1 001
       10328 0 0 1 001
       10330 0 0 1 001
       10343 0 0 1 001
       10400 0 0 1 001
       10500 0 0 1 001
       10530 0 0 1 001
       10543 0 0 1 001
       10600 0 0 1 001
       10700 0 0 1 001
       10715 0 0 1 110
       10728 0 0 1 110
       10730 0 0 1 110
       10800 0 0 1 110
       10900 0 0 1 110
       10915 0 0 1 001
```

```
        10928 0 0 1 001
        10930 0 0 1 001
        10943 0 0 1 001
        11000 0 0 1 001
        11100 0 0 1 001
        11130 0 0 1 001
        11143 0 0 1 001
        11200 0 0 1 001
        11300 0 0 1 001
1EPILOG2 -V2.5  (10.09.86) ESSAI DE SIMULER AM2910 Le  7/10/87 Page   26
-------------------------------------

      HEURE  C C R MMM
             C C I UUU
               E D CCC
               N   TTT
                   LLL
                   ###
                   012



        11330 0 0 1 001
        11343 0 0 1 001
        11400 0 0 1 001
        11500 0 0 1 001
        11515 0 0 1 000
        11528 0 0 1 000
        11530 0 0 1 000
        11600 0 0 1 000


   193- *SYMBOLIQUE
   194- B=REGCTL#0:1
   195- B=STACTL#0:1
   196- B=PL
   197- B=MAP
   198- B=VECT
1EPILOG2 -V2.5- (10.09.86) ESSAI DE SIMULER AM2910 Le  7/10/87 Page   27
-------------------------------------

      HEURE  RR SS P M V
             EE II I A E
             GG AA   P C
             CC CC     T
             II II
             LL LL
             ## ##
             01 01



          0 XX XX X X X
```

```
    0 10 01 X X X
  100 10 01 X X X
  115 10 01 1 0 0
  200 10 01 1 0 0
  300 10 01 1 0 0
  315 10 01 1 0 0
  330 10 01 1 0 0
  400 10 01 1 0 0
  500 10 01 1 0 0
  515 10 01 1 0 0
  528 10 01 1 0 0
  600 10 01 1 0 0
  700 10 01 1 0 0
  730 10 01 1 0 0
  743 10 01 1 0 0
  800 10 01 1 0 0
  900 10 01 1 0 0
  915 10 00 1 0 0
  928 10 00 1 0 0
  930 10 00 1 0 0
 1000 10 00 1 0 0
 1100 10 00 1 0 0
 1115 10 01 1 0 0
 1128 10 01 1 0 0
 1130 10 01 1 0 0
 1143 10 01 1 0 0
 1200 10 01 1 0 0
 1300 10 01 1 0 0
 1330 10 01 1 0 0
 1343 10 01 1 0 0
 1400 10 01 1 0 0
 1500 10 01 1 0 0
 1515 10 01 1 0 0
 1528 10 01 1 0 0
 1530 10 01 1 0 0
 1600 10 01 1 0 0
 1700 10 01 1 0 0
 1715 10 01 1 0 0
 1728 10 01 1 0 0
 1730 10 01 1 0 0
 1743 10 01 1 0 0
 1800 10 01 1 0 0
 1900 10 01 1 0 0
 1930 10 01 1 0 0
1EPILOG2 -V2.5- (10.09.86) ESSAI DE SIMULER AM2910  Le  7/10/87 Page   28
-----------------------------


     HEURE  RR SS P M V
            EE TT L A E
            GG AA   P C
            CC CC     T
            TT TT
            LL LL
            ## ##
            01 01
```

```
1943 10 01 1 0 0
2000 10 01 1 0 0
2100 10 01 1 0 0
2115 10 01 0 1 0
2128 10 01 0 1 0
2130 10 01 0 1 0
2200 10 01 0 1 0
2300 10 01 0 1 0
2315 10 01 1 0 0
2328 10 01 1 0 0
2330 10 01 1 0 0
2343 10 01 1 0 0
2400 10 01 1 0 0
2500 10 01 1 0 0
2530 10 01 1 0 0
2543 10 01 1 0 0
2600 10 01 1 0 0
2700 10 01 1 0 0
2730 10 01 1 0 0
2743 10 01 1 0 0
2800 10 01 1 0 0
2900 10 01 1 0 0
2915 01 00 1 0 0
2930 01 00 1 0 0
2943 01 00 1 0 0
3000 01 00 1 0 0
3100 01 00 1 0 0
3115 10 01 1 0 0
3130 10 01 1 0 0
3143 10 01 1 0 0
3200 10 01 1 0 0
3300 10 01 1 0 0
3330 10 01 1 0 0
3343 10 01 1 0 0
3400 10 01 1 0 0
3500 10 01 1 0 0
3530 10 01 1 0 0
3543 10 01 1 0 0
3600 10 01 1 0 0
3700 10 01 1 0 0
3715 00 01 1 0 0
3728 00 01 1 0 0
3730 00 01 1 0 0
3800 00 01 1 0 0
3900 00 01 1 0 0
```



```
HEURE  RR SS P M V
       EE TT I A E
       GG AA   P C
       CC CC     T
       TT TT
       LL LL
```

```
          ## ##
          01 01


       3915 10 01 1 0 0
       3928 10 01 1 0 0
       3930 10 01 1 0 0
       3943 10 01 1 0 0
       4000 10 01 1 0 0
       4100 10 01 1 0 0
       4130 10 01 1 0 0
       4143 10 01 1 0 0
       4200 10 01 1 0 0
       4300 10 01 1 0 0
       4330 10 01 1 0 0
       4343 10 01 1 0 0
       4400 10 01 1 0 0
       4500 10 01 1 0 0
       4515 00 01 1 0 0
       4528 00 01 1 0 0
       4530 00 01 1 0 0
       4600 00 01 1 0 0
       4700 00 01 1 0 0
       4715 10 01 1 0 0
       4728 10 01 1 0 0
       4730 10 01 1 0 0
       4743 10 01 1 0 0
       4800 10 01 1 0 0
       4900 10 01 1 0 0
       4930 10 01 1 0 0
       4943 10 01 1 0 0
       5000 10 01 1 0 0
       5100 10 01 1 0 0
       5130 10 01 1 0 0
       5143 10 01 1 0 0
       5200 10 01 1 0 0
       5300 10 01 1 0 0
       5315 00 01 1 0 0
       5328 00 01 1 0 0
       5330 00 01 1 0 0
       5400 00 01 1 0 0
       5500 00 01 1 0 0
       5515 10 01 1 0 0
       5528 10 01 1 0 0
       5530 10 01 1 0 0
       5543 10 01 1 0 0
       5600 10 01 1 0 0
       5700 10 01 1 0 0
       5730 10 01 1 0 0
1EPILOG2 -V2.5- (10.09.86) ESSAI DE SIMULER AM2910 Le  7/10/87 Page   30
-----------------------------


     HEURE  RR SS P M V
            EE TT L A E
            GG AA   P C
```

```
      CC CC      T
      TT TT
      LL LL
      ## ##
      01 01


    5743 10 01 1 0 0
    5800 10 01 1 0 0
    5900 10 01 1 0 0
    5915 10 01 1 0 0
    5930 10 01 1 0 0
    5943 10 01 1 0 0
    6000 10 01 1 0 0
    6100 10 01 1 0 0
    6115 01 01 1 0 0
    6130 01 01 1 0 0
    6143 01 01 1 0 0
    6200 01 01 1 0 0
    6300 01 01 1 0 0
    6315 10 00 1 0 0
    6328 10 00 1 0 0
    6330 10 00 1 0 0
    6343 10 00 1 0 0
    6400 10 00 1 0 0
    6500 10 00 1 0 0
    6515 10 01 1 0 0
    6528 10 01 1 0 0
    6530 10 01 1 0 0
    6543 10 01 1 0 0
    6600 10 01 1 0 0
    6700 10 01 1 0 0
    6730 10 01 1 0 0
    6743 10 01 1 0 0
    6800 10 01 1 0 0
    6900 10 01 1 0 0
    6930 10 01 1 0 0
    6943 10 01 1 0 0
    7000 10 01 1 0 0
    7100 10 01 1 0 0
    7115 01 01 1 0 0
    7130 01 01 1 0 0
    7143 01 01 1 0 0
    7200 01 01 1 0 0
    7300 01 01 1 0 0
    7315 00 01 1 0 0
    7328 00 01 1 0 0
    7330 00 01 1 0 0
    7400 00 01 1 0 0
    7500 00 01 1 0 0
    7530 00 01 1 0 0
    7600 00 01 1 0 0
```

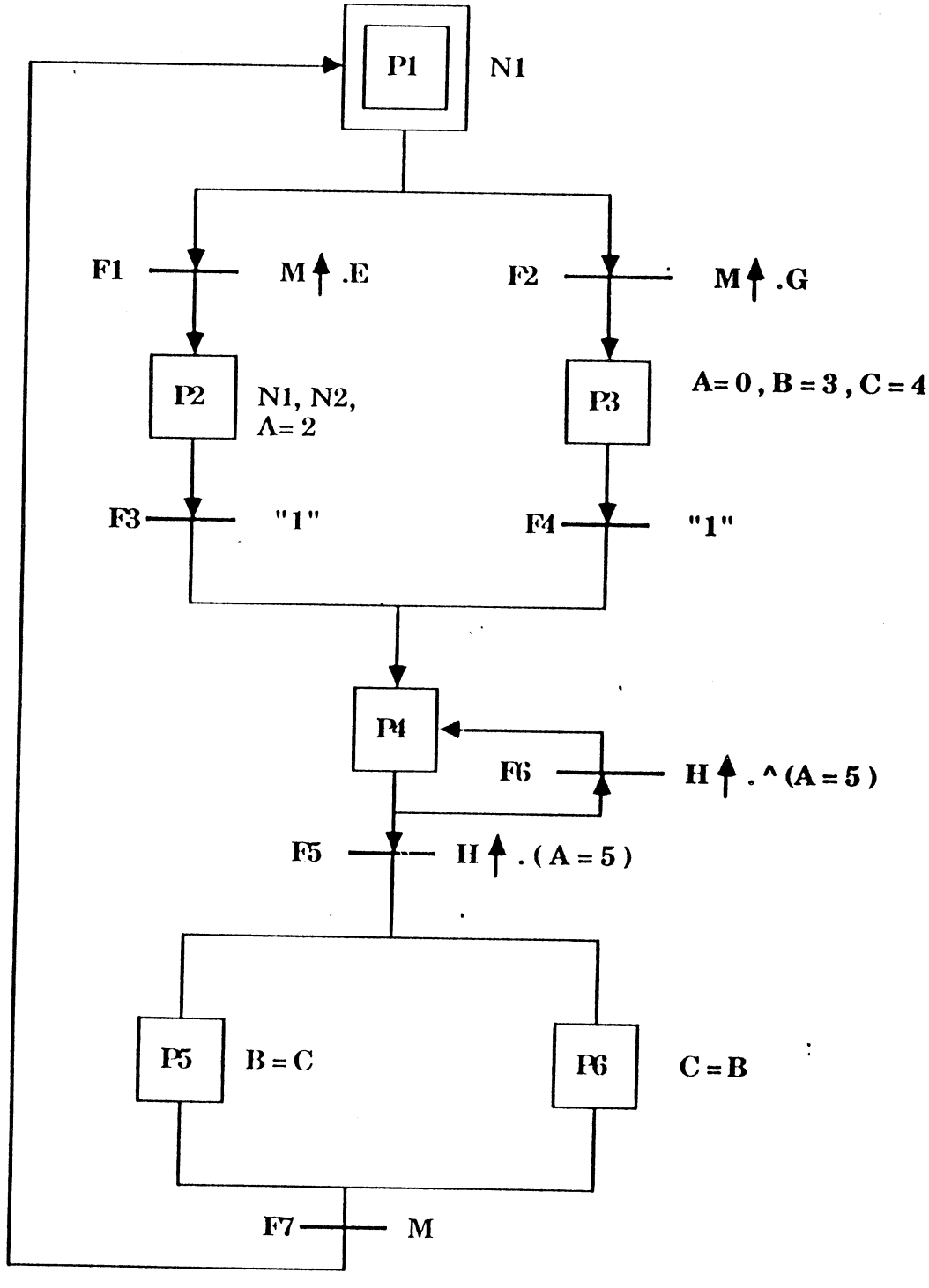
HEURE	RR EE GG CC TT LL ## 01	SS TT AA CC TT LL ## 01	P L 	M A P 	V E C T
7700	00	01	1	0	0
7715	10	01	1	0	0
7728	10	01	1	0	0
7730	10	01	1	0	0
7800	10	01	1	0	0
7900	10	01	1	0	0
7930	10	01	1	0	0
7943	10	01	1	0	0
8000	10	01	1	0	0
8100	10	01	1	0	0
8115	10	10	1	0	0
8128	10	10	1	0	0
8130	10	10	1	0	0
8200	10	10	1	0	0
8300	10	10	1	0	0
8315	10	01	1	0	0
8328	10	01	1	0	0
8330	10	01	1	0	0
8343	10	01	1	0	0
8400	10	01	1	0	0
8500	10	01	1	0	0
8530	10	01	1	0	0
8543	10	01	1	0	0
8600	10	01	1	0	0
8700	10	01	1	0	0
8730	10	01	1	0	0
8743	10	01	1	0	0
8800	10	01	1	0	0
8900	10	01	1	0	0
8915	10	01	0	0	1
8928	10	01	0	0	1
8930	10	01	0	0	1
9000	10	01	0	0	1
9100	10	01	0	0	1
9115	10	01	1	0	0
9128	10	01	1	0	0
9130	10	01	1	0	0
9143	10	01	1	0	0
9200	10	01	1	0	0
9300	10	01	1	0	0
9330	10	01	1	0	0
9343	10	01	1	0	0
9400	10	01	1	0	0
9500	10	01	1	0	0
9515	01	00	1	0	0

HEURE	RR EE GG CC TT LL ## 01	SS TT AA CC TT LL ## 01	P L	M A P	V E C T
9530	01	00	1	0	0
9543	01	00	1	0	0
9600	01	00	1	0	0
9700	01	00	1	0	0
9715	10	01	1	0	0
9730	10	01	1	0	0
9743	10	01	1	0	0
9800	10	01	1	0	0
9900	10	01	1	0	0
9915	10	10	1	0	0
9928	10	10	1	0	0
9930	00	01	1	0	0
9943	00	01	1	0	0
10000	00	01	1	0	0
10100	00	01	1	0	0
10115	10	10	1	0	0
10128	10	10	1	0	0
10130	10	10	1	0	0
10200	10	10	1	0	0
10300	10	10	1	0	0
10315	10	01	1	0	0
10328	10	01	1	0	0
10330	10	01	1	0	0
10343	10	01	1	0	0
10400	10	01	1	0	0
10500	10	01	1	0	0
10530	10	01	1	0	0
10543	10	01	1	0	0
10600	10	01	1	0	0
10700	10	01	1	0	0
10715	10	10	1	0	0
10728	10	10	1	0	0
10730	10	10	1	0	0
10800	10	10	1	0	0
10900	10	10	1	0	0
10915	10	01	1	0	0
10928	10	01	1	0	0
10930	10	01	1	0	0
10943	10	01	1	0	0
11000	10	01	1	0	0
11100	10	01	1	0	0
11130	10	01	1	0	0
11143	10	01	1	0	0

```
     11200 10 01 1 0 0
     11300 10 01 1 0 0
1EPILOG2 -V2.5- (10.09.86) ESSAI DE SIMULER AM2910 Le  7/10/87 Page   33
----------------------------


    HEURE  RR SS P M V
           EE TT L A E
           GG AA   P C
           CC CC     T
           TT TT
           LL LL
           ## ##
           01 01



     11330 10 01 1 0 0
     11343 10 01 1 0 0
     11400 10 01 1 0 0
     11500 10 01 1 0 0
     11515 10 11 1 0 0
     11528 10 11 1 0 0
     11530 10 11 1 0 0
     11600 10 11 1 0 0



PHASE **EDITIO   TEMPS  CPU  :      14.24   SECONDES
                 ELAPSED TIME :  0h  0mn 29s
```

ANNEXE C : EXEMPLE DE GRAFCET ET FIDEL

C.1 : Description en Grafcet :



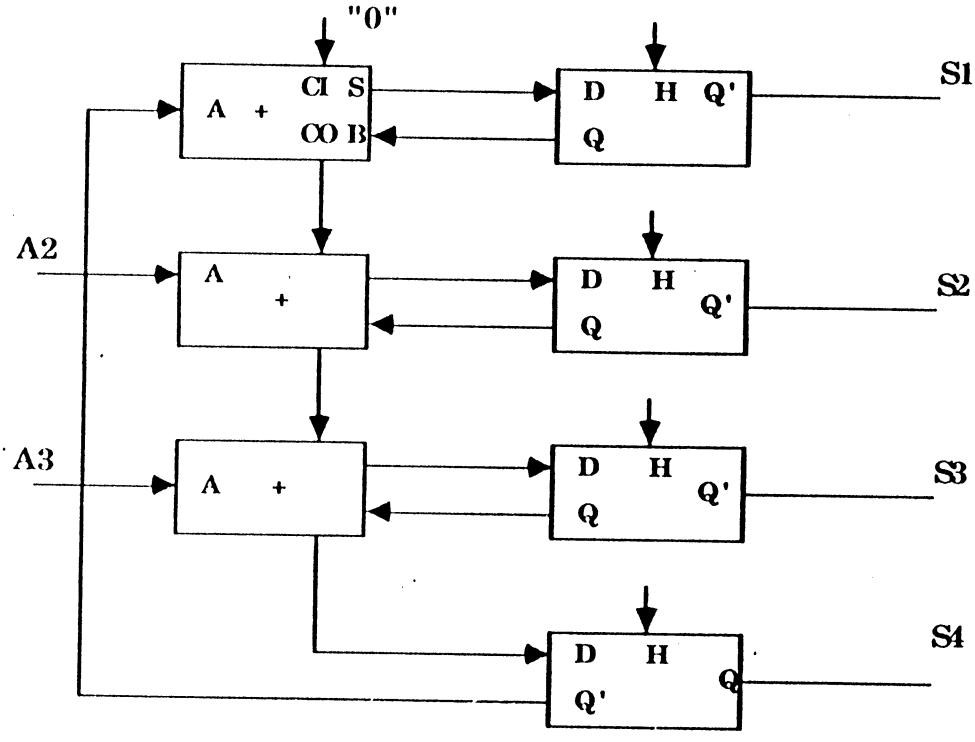
C.2 : Description en FIDEL :

```
FMODEL GRA1 ( E , G , M , H , N1 , N2 , B , C , A, P ) ;

DECLARE INPUT  E , G , M , H  ;
DECLARE OUTPUT B ( 0 : 3 ) , C ( 0 : 3 ) , N1 , N2 , A ( 0 : 3 ) ;
DECLARE  OUTPUT  P ( 1 : 6 ) ;
DECLARE  STATE  F ( 1 : 7 ) , FLAG , HFRONT , MFRONT ;

INITIALIZE  P , C , N1  TO  1 ;
INITIALIZE  A , N2 , HFRONT , MFRONT  TO  0 ;

FUNCTIONAL
   WHEN H RISES    MAKE HFRONT = 1 ;

   WHEN M RISES    MAKE MFRONT = 1 ;

   WHEN  E CHANGES OR G CHANGES  OR M CHANGES OR H CHANGES
        MAKE F = 0
        MAKE F (1) = MFRONT & E & P (1)
        MAKE F (2) = MFRONT & G & P (1)
        MAKE F (3) = P (2)
        MAKE F (4) = P (3)
        IF  A = 5  THEN  MAKE  FLAG = 1
                   ELSE  MAKE  FLAG = 0
        ENDIF
        MAKE  F (5) = HFRONT  &  FLAG  &  P (4)
        MAKE  F (6) = HFRONT  &  NOT FLAG  &  P (4)
        MAKE  F (7) = M  &  P (5)  &  P'(6)
        MAKE  HFRONT = 0  MAKE MFRONT = 0  ;

   WHEN  F  CHANGES
        PAR
          IF  F (1) = 1 THEN  MAKE  A = 2  ENDIF
          IF  F (2) = 1  THEN  MAKE  A = 0  MAKE B = 3  MAKE C = 4  ENDIF
          IF  F (3) = 1  OR F (4) = 1  THEN  MAKE  A = A + 1  ENDIF
          IF F (5) = 1  THEN  MAKE  B = C  MAKE  C = B  ENDIF
```

```
    IF F (6) = 1 THEN MAKE A = A + 1 ENDIF
  ENDPAR
  MAKE P (1) = F (7) | P (1) & NOT F (1) & NOT F (2)
  MAKE P (2) = F (1) | P (2) & NOT F (3)
  MAKE P (3) = F (2) | P (3) & NOT F (4)
  MAKE P (4) = F (3) | F (4) | F (6) | P (4) & NOT F (5)
  MAKE P (5) = F (5) | P (5) & NOT F (7)
  MAKE P (6) = F (5) | P (6) & NOT F (7)
  MAKE N1 = P (1) | P (2)
  MAKE N2 = P (2) | P (4) ;

ENDFUNCTIONAL
ENDMODEL
```

ANNEXE D : EXEMPLE DE FIDEL + ELDO

L'exemple qu'on présente ici est le circuit AD4B (3 bits d'additionneur et 4 bascules de type D). On présente d'abord le schéma fonctionnel et on divise le circuit en deux parties (une partie en FIDEL et l'autre partie en ELDO). On termine par la présentation de résultats de simulation.



D.2 Description en FIDEL :

```
FMODEL ADD (N1,A1,A2,N4,N7,N12,CIN,N13,N3,N6,N11) ;

DECLARE INPUT N1,A1,A2,N4,N7,N12,CIN ;
DECLARE OUTPUT N13,N3,N6,N11 ;
DECLARE STATE N5,N8;
INITIALIZE N5,N8 TO 0 ;

FUNCTIONAL

WHEN N1 CHANGES  OR N4 CHANGES
     WITHIN 0 TO 10 MAKE N3 = CIN XOR N1 XOR N4
                    MAKE N5 = N1 & CIN OR N4 & CIN OR N1 & N4
     ENDWITHIN ;

WHEN N5 CHANGES OR N7 CHANGES
     WITHIN 0 TO 10 MAKE N6 = A1 XOR N5 XOR N7
                    MAKE N8 = A1 & N5 OR A1 & N7 OR N5 & N7
     ENDWITHIN ;

WHEN N8 CHANGES OR N12 CHANGES
     WITHIN 0 TO 10 MAKE N11 = N8 XOR N12 XOR A2
                    MAKE N13 = N8 & N12 OR N8 & A2 OR N12 & A2
     ENDWITHIN ;

ENDFUNCTIONAL
ENDMODEL
```

D.3 Description de circuit en ELDO :

```
FIDELDO 2.3 NEWAD4B.CIR

(* Description de parametres de modeles *)

.MODELE MOD1 NMOS NIV=2 EOX=25.0N MUO=600 NB=2.0E+16
+ DPHIF=0.6 VL=2.0E+5 KW=2.24U KL=2.24U  GW=3.91U GL=0.7U DINF=0.1
+ VE=10.0E+4 LDIF=10U  CDIFSO=0.0001 CDIFPO=0.0
+ DW=0.0  DL=0.8U  REC=0.15U VTO=0.55 KB=0.1 TG=0.06

.MODELE MOD2 PMOS NIV=2 EOX=25N MUO=225 NB=2.0E+16
+ DPHIF=0.7 VL=1.9E+5  KW=0.52U  KL=4U GW=0.453U GL=0.7U DINF=0.17
+ VE=7.35E+4 LDIF=10U  CDIFSO=0.000316 CDIFPO=0
+ DW=0 DL=1.5U REC=0.5U VTO=0.55 KB=0.34 TG=0.14

.MODELE MOD3 NMOS
+ NIV=2 DW=0.5U  DL=1.1U EOX=60N DPHIF=0.7
+ VTO = -4.0  MUO=446.0  KB = 0.4  KW = 0.0U  KL = 0.0U  GW = 2.4U  GL = 0.5U
+ DINF=0.14  VE=12.8E+4 REC = 0.4U TG = 0.05 VL = 1.0E+5 SH = 0.1
+ NB=1E+15

.MODELE MOD4 NMOS
+  NIV = 2  DW = 0.5U  DL = 1.1U  EOX = 60.0N  DPHIF = 0.7
+ VTO = 0.7  MUO = 672.0  KB = 0.234  KW = 2.17U  KL = 0.49U  GW = 4.325U
```

```
+ GL = 0.872U  DINF = 0.105  VE = 7.71E+4  REC =  0.4U  TG = 0.05  VL = 8.55E+4
+ SH = 0.1     NB = 1.0E+15

(* Description de circuit *)

.SOUS-CIRCUIT BASCD VDD VSS H D Q QB

M1 N1 N2 0 VSS MOD4 W=15U L=3.5U
M2 N2 N1 0 VSS MOD4 W=15U L=3.5U
M3 N2 H 0 VSS MOD4 W=15U L=3.5U
M4 N4 N2 0 VSS MOD4 W=15U L=3.5U
M5 N1 N5 0 VSS MOD4 W=15U L=3.5U
M6 N4 H 0 VSS MOD4 W=15U L=3.5U
M7 N4 N5 0 VSS MOD4 W=15U L=3.5U
M8 N5 N4 0 VSS MOD4 W=15U L=3.5U
M9 N5 D 0 VSS MOD4 W=15U L=3.5U
M10 Q N2 0 VSS MOD4 W=15U L=3.5U
M11 Q QB 0 VSS MOD4 W=15U L=3.5U
M12 QB Q 0 VSS MOD4 W=15U L=3.5U
M13 QB N4 0 VSS MOD4 W=15U L=3.5U
M14 VDD N1 N1 VSS MOD3 W=6.0U L=5.0U
M15 VDD N2 N2 VSS MOD3 W=6.0U L=5.0U
M16 VDD N4 N4 VSS MOD3 W=6.0U L=5.0U
M17 VDD N5 N5 VSS MOD3 W=6.0U L=5.0U
M18 VDD Q Q VSS MOD3 W=6.0U L=5.0U
M19 VDD QB QB VSS MOD3 W=6.0U L=5.0U

C1 N1 0 0.029P
C2 N2 0 0.048P
C3 N4 0 0.044P
C4 N5 0 0.046P
C5 Q 0 0.066P
C6 QB 0 0.053P

.FINS BASCD

XB1  VDD VSS H N3 N4 S1 BASCD
XB2  VDD VSS H N6 N7 S2 BASCD
XB3  VDD VSS H N11 N12 S3 BASCD
XB4  VDD VSS H N13 S4 N1 BASCD

C1 N3 0 0.07P
C2 N4 0 0.04P
C3 N6 0 0.07P
C4 N7 0 0.04P
C5 N5 0 0.01P
C6 N8 0 0.01P
C7 N13 0 0.01P
C8 N1 0 0.04P
C9 N11 0 0.07P
C10 N12 0 0.04P

(* Interface avec le modele fonctionnel *)

.fonc n1 a1 a2 n4 n7 n12 cin , n13 n3 n6 n11

(* Valeurs de signaux *)
```

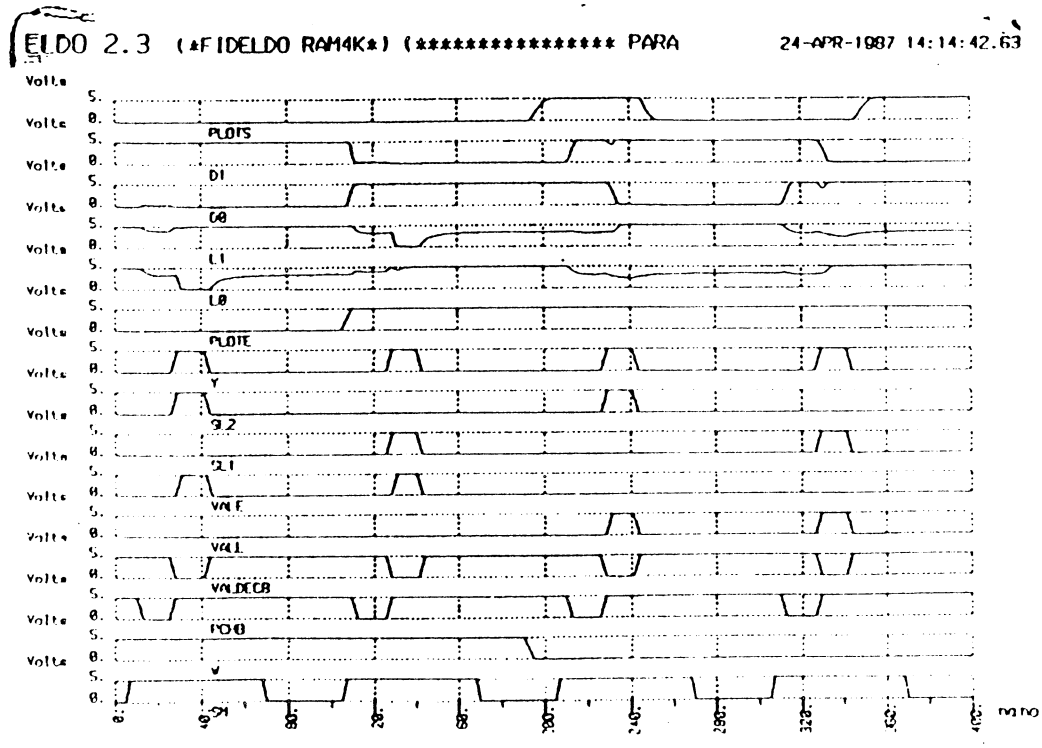
```
.CHRENT A1 ON 5 F
.CHRENT A2 ON 5 F
.CHRENT H  ON 0   5N 5   15N 5   20N 0  30N 0  P
.CHRENT VDD  ON 5 F
.CHRENT VSS ON -2.5 F

(* Demande de visualisation de signaux *)

.VISUALISER H S1 S2 S3 S4 N3 N6 N11 N13
.FIN
```

D.4 Resultats de simulation :

```
 CNET Copyright  ELDO  VERSION : 2.4 du 15 MAI 1987
  1-SEP-1987  11:42:53.87

 TEMPERATURE :  2.70000E+01 degres C

calcul de :  0.0000000000000E+00 a :  4.8000000000000E+02
-----------------------------------------------------
front montant sur: N4       a  2.3E+01
-----------------------------------------------------
front montant sur: N7       a  2.3E+01
-----------------------------------------------------
front montant sur: N12      a  2.3E+01
appel fidel no :   2 a  2.3E+01 he =  0.0E+00
-----------------------------------------------------
front descendant sur: N1       a  5.0E+01
appel fidel no :   3 a  5.0E+01 he =  0.0E+00
-----------------------------------------------------
front descendant sur: N4       a  5.1E+01
appel fidel no :   4 a  5.1E+01 he =  0.0E+00

***>TEMPS CPU ECOULE:   0mn30s720ms<***

Il reste :       400 pas d'affichage a calculer
-----------------------------------------------------
front descendant sur: N7       a  8.1E+01
appel fidel no :   5 a  8.1E+01 he =  0.0E+00
-----------------------------------------------------
front descendant sur: N12      a  1.1E+02
appel fidel no :   6 a  1.1E+02 he =  0.0E+00
-----------------------------------------------------
front montant sur: N7       a  1.1E+02
appel fidel no :   7 a  1.1E+02 he =  0.0E+00
-----------------------------------------------------
front descendant sur: N7       a  1.4E+02
appel fidel no :   8 a  1.4E+02 he =  0.0E+00
-----------------------------------------------------
front montant sur: N1       a  1.7E+02
-----------------------------------------------------
front montant sur: N7       a  1.7E+02
-----------------------------------------------------
```

```
front montant sur: N12        a  1.7E+02
appel Fidel no :   9 a  1.7E+02 he =  0.0E+00

***>TEMPS CPU ECOULE:    1mn 7s180ms<***

Il reste :        300 pas d'affichage a calculer
-------------------------------------------------
front descendant sur: N1        a  2.0E+02
-------------------------------------------------
front descendant sur: N7        a  2.0E+02
appel Fidel no :  10 a  2.0E+02 he =  0.0E+00
-------------------------------------------------
front montant sur: N4        a  2.0E+02
appel Fidel no :  11 a  2.0E+02 he =  0.0E+00
-------------------------------------------------
front descendant sur: N12       a  2.3E+02
appel Fidel no :  12 a  2.3E+02 he =  0.0E+00
-------------------------------------------------
front montant sur: N7        a  2.3E+02
appel Fidel no :  13 a  2.3E+02 he =  0.0E+00
-------------------------------------------------
front descendant sur: N7        a  2.6E+02
appel Fidel no :  14 a  2.6E+02 he =  0.0E+00

***>TEMPS CPU ECOULE:    1mn44s440ms<***

Il reste :        200 pas d'affichage a calculer
-------------------------------------------------
front montant sur: N1        a  2.9E+02
-------------------------------------------------
front montant sur: N7        a  2.9E+02
-------------------------------------------------
front montant sur: N12       a  2.9E+02
appel Fidel no :  15 a  2.9E+02 he =  0.0E+00
-------------------------------------------------
front descendant sur: N1        a  3.2E+02
-------------------------------------------------
front descendant sur: N4        a  3.2E+02
appel Fidel no :  16 a  3.2E+02 he =  0.0E+00
-------------------------------------------------
front descendant sur: N7        a  3.5E+02
appel Fidel no :  17 a  3.5E+02 he =  0.0E+00

***>TEMPS CPU ECOULE:    2mn17s810ms<***

Il reste :        100 pas d'affichage a calculer
-------------------------------------------------
front descendant sur: N12       a  3.8E+02
appel Fidel no :  18 a  3.8E+02 he =  0.0E+00
-------------------------------------------------
front montant sur: N7        a  3.8E+02
appel Fidel no :  19 a  3.8E+02 he =  0.0E+00
-------------------------------------------------
front descendant sur: N7        a  4.1E+02
appel Fidel no :  20 a  4.1E+02 he =  0.0E+00
-------------------------------------------------
front montant sur: N1        a  4.4E+02
```

```
appel fidel no :   21 a  4.4E+02 he =   0.0E+00
-------------------------------------------------
front montant sur: N7        a  4.4E+02
-------------------------------------------------
front montant sur: N12       a  4.4E+02
appel fidel no :   22 a  4.4E+02 he =   0.0E+00
-------------------------------------------------
front descendant sur: N1     a  4.7E+02
-------------------------------------------------
front descendant sur: N7     a  4.7E+02
appel fidel no :   23 a  4.7E+02 he =   0.0E+00
-------------------------------------------------
front montant sur: N4        a  4.7E+02
appel fidel no :   24 a  4.7E+02 he =   0.0E+00

***>TEMPS CPU ECOULE:    2mn56s950ms<***

Il reste :          0 pas d'affichage a calculer



 place memoire matrice creuse :          0
courant moyen:  0.000E+00 %latence:  9.140E+00 nb moyen relax:  2.159E+00
nb maxi de relaxations du transitoire:          4
 Le nombre moyen d'appels fonction par resolution:  1.821E+00
dont:  1.409E+00 points fixes et:  4.123E-01 secantes
 un seul appel: 5.877E-01 deux appels:  1.111E-01
 le nombre d'appel a newton :          0
 le nombre moyen d'iteration pour un newton : 0.000E+00
 nb de minima:          0 nb max appel:          10
 nb de composants:         76 nb de noeuds:          36
 nb d'appels mos:     144813 cournoeud:       29934 jus:          0
 pas rejetes:        194pas acceptes:        436

***>TEMPS CPU ECOULE:    2mn58s270ms<***
```



ANNEXE E : DESCRIPTION DE STRUCTURE ET DE SIMULATION EN FIDEL

E.1 Description de structure :

```
(*-------------------------------------------------------------------------*)
(*-------------- THE STRUCTURAL DESCRIPTION IN FIDEL STAND_ALONE -------- *)
(*-------------------------------------------------------------------------*)

SMODEL AM2910(INIT,I,CC,CCEN,OE,RLD,CI,CP,D,PL,MAP,VECT,Y) ;

DECLARE INPUT INIT,I(0:3),CC,CCEN,OE,RLD,CI,CP,D(0:11) ;

DECLARE OUTPUT PL,MAP,VECT,Y(0:11) ;

DECLARE COMPONENT PLA : INSTPLA , MUX1 : MUX , BUFF : OUTBUFF ;

DECLARE COMPONENT UPC1 : UPC , REG1 : REG , PILE : STACK ;

STRUCTURAL

  CONNECT AM2910.INIT,PLA.INIT,MUX1.INIT,UPC1.INIT,REG1.INIT,PILE.INIT
  CONNECT AM2910.I,PLA.I
  CONNECT AM2910.CC,PLA.CC
  CONNECT AM2910.CCEN,PLA.CCEN
  CONNECT AM2910.OE,BUFF.OE
  CONNECT AM2910.RLD,REG1.RLD
  CONNECT AM2910.CI,UPC1.CI
  CONNECT AM2910.CP,UPC1.CP,PILE.CP;REG1.CP
  CONNECT AM2910.D,MUX1.D,REG1.D
  CONNECT REG1.R,PLA.R
  CONNECT PLA.PL,AM2910.PL
  CONNECT PLA.MAP,AM2910.MAP
  CONNECT PLA.VECT,AM2910.VECT
  CONNECT PLA.REGCTL,REG1.REGCTL
  CONNECT PLA.STACKCTL,PILE.STACKCTL
  CONNECT PLA.MUXCTL,MUX1.MUXCTL
  CONNECT REG1.RR,MUX1.RR
  CONNECT MUX1.YI,BUFF.YI,UPC1.YI
  CONNECT BUFF.Y,AM2910.Y
  CONNECT UPC1.PC,MUX1.PC,PILE.PC
  CONNECT PILE.F,MUX1.F

ENDSTRUCTURAL
ENDMODEL
```

E.2 Description de scenarion de simulation :

```
(*------------------ SIMULATION ENVIRONMENT --------------------------*)

GMODEL AM2910 ;

DECLARE CLOCK H1(11600,100,100,0) ;

FUNCTIONAL
```

```
WHEN TIME BECOMES 0
     MAKE INIT = 1 ;

WHEN TIME BECOMES 1
     MAKE INIT = 0 ;

WHEN TIME BECOMES 115
     MAKE OE = 0
     MAKE CI = 1
     MAKE RLD = 0
     MAKE CC = 1
     MAKE CCEN = 0
     MAKE I = 7
     MAKE D = 0 ;

WHEN TIME BECOMES 315
     MAKE RLD = 1 ;

WHEN TIME BECOMES 515
     MAKE I = 14 ;

WHEN TIME BECOMES 915
     MAKE CC = 0
     MAKE I = 1
     MAKE D = #H100 ;

WHEN TIME BECOMES 1115
     MAKE I = 14
     MAKE D = 0 ;

WHEN TIME BECOMES 1515
     MAKE I = 3
     MAKE D = #H200 ;

WHEN TIME BECOMES 1715
     MAKE I = 14 ;

WHEN TIME BECOMES 2115
     MAKE I = 2
     MAKE D = #H300 ;

WHEN TIME BECOMES 2315
     MAKE I = 14
     MAKE D = 0 ;

WHEN TIME BECOMES 2915
     MAKE I = 4
     MAKE D = #H003 ;

WHEN TIME BECOMES 3115
     MAKE I = 14
     MAKE D = 0 ;

WHEN TIME BECOMES 3715
     MAKE I = 8 ;
```

```
WHEN TIME BECOMES 3915
     MAKE I = 14 ;

WHEN TIME BECOMES 4515
     MAKE I = 8 ;

WHEN TIME BECOMES 4715
     MAKE I = 14 ;

WHEN TIME BECOMES 5315
     MAKE I = 8 ;

WHEN TIME BECOMES 5515
     MAKE I = 14 ;

WHEN TIME BECOMES 5915
     MAKE CC = 1 ;

WHEN TIME BECOMES 6115
     MAKE I = 12
     MAKE D = #H400 ;

WHEN TIME BECOMES 6315
     MAKE I = 5
     MAKE D = 0 ;

WHEN TIME BECOMES 6515
     MAKE CC = 0
     MAKE I = 14 ;

WHEN TIME BECOMES 7115
     MAKE I = 12
     MAKE D = #H002 ;

WHEN TIME BECOMES 7315
     MAKE I = 9
     MAKE D = #H500 ;

WHEN TIME BECOMES 7715
     MAKE I = 14
     MAKE D = 0 ;

WHEN TIME BECOMES 8115
     MAKE I = 10 ;

WHEN TIME BECOMES 8315
     MAKE I = 14 ;

WHEN TIME BECOMES 8915
     MAKE I = 6
     MAKE D = #H600 ;

WHEN TIME BECOMES 9115
     MAKE I = 14
     MAKE D = 0 ;

WHEN TIME BECOMES 9515
```

```
        MAKE I = 15
        MAKE D = #H004 ;

  WHEN TIME BECOMES 9715
        MAKE CC = 1
        MAKE I = 11
        MAKE D = #H700 ;

  WHEN TIME BECOMES 9915
        MAKE I = 15
        MAKE D = 0 ;

  WHEN TIME BECOMES 10115
        MAKE CC = 0
        MAKE I = 11
        MAKE D = #H700 ;

  WHEN TIME BECOMES 10315
        MAKE I = 14
        MAKE D = 0 ;

  WHEN TIME BECOMES 10715
        MAKE I = 10 ;

  WHEN TIME BECOMES 10915
        MAKE I = 14 ;

  WHEN TIME BECOMES 11515
        MAKE I = 0 ;

ENDFUNCTIONAL
ENDMODEL
```

E.3'Resultats de Simulation :

```
*** FIDEL V3.1 *** IMAG_CNET LE 13-11-86 genere le 14-JAN-1987 a 12:36:37.83

************************************************
            INITIALIZATION  PHASE
************************************************

 CPU TIME   :     3.80000E+00
 ELAPSED TIME :      0 H      0 M      9 S


************************************************
            GENERATION  PHASE
************************************************

 CPU TIME   :     6.40000E-01
 ELAPSED TIME :      0 H      0 M      2 S


************************
```

```
   SIMULATION TIME ===>      0
**************************

input values are :
    1    0
    3    2
    3    2
    3    2
    3    2
    3    2
    3    2
    3    2
    3    2

 output values are :
Y          -->  av:          3 nv:          0
--> idaval :          2 idnval :          0

*************************
    SIMULATION TIME ===>   115
*************************

input values are :
    0
    7
    1
    0
    0
    0
    1
    3
    0

 output values are :
PL         -->  av:          3 nv:          1
--> idaval :          2 idnval :          0
MAP        -->  av:          3 nv:          0
--> idaval :          2 idnval :          0
VECT       -->  av:          3 nv:          0
--> idaval :          2 idnval :          0

*************************
    SIMULATION TIME ===>  2115
*************************

input values are :
    0
    2
    0
    0
    0
    1
    1
    3
  768

output values are :
```

```
PL          -->  av:           1 nv:          0
--> idaval :          0 idnval :          0
MAP         -->  av:           0 nv:          1
--> idaval :          0 idnval :          0

*************************
    SIMULATION TIME ===>  2315
*************************

input values are :
    0
   14
    0
    0
    0
    1
    1
    3
    0

 output values are :
PL          -->  av:           0 nv:          1
--> idaval :          0 idnval :          0
MAP         -->  av:           1 nv:          0
--> idaval :          0 idnval :          0

*************************
    SIMULATION TIME ===>  8915
*************************

input values are :
    0
    6
    0
    0
    0
    1
    1
    3
 1536

 output values are :
PL          -->  av:           1 nv:          0
--> idaval :          0 idnval :          0
VECT        -->  av:           0 nv:          1
--> idaval :          0 idnval :          0

*************************
    SIMULATION TIME ===>  9115
*************************

input values are :
    0
   14
    0
    0
    0
```

```
     1
     1
     3
     0

 output values are :
PL          -->  av:           0 nv:           1
--> idaval :           0 idnval :           0
VECT        -->  av:           1 nv:           0
--> idaval :           0 idnval :           0

**********************************************
            SIMULATION  PHASE
**********************************************

 CPU TIME   :     6.10000E+00
 ELAPSED TIME :      0 H      0 M     28 S
```

RESUME

Cette thèse discute dans un premier temps des propriétés et des concepts des langages de description de matériel HDL, ceci nous a permis de mettre l'accent sur des points importants que nous avons pris en considération lors de la définition et l'implémentation de notre travail.

Ensuite, le langage FIDEL pour la description (fonctionnelle et structurelle) et la simulation de circuits intégrés VLSI est présenté, en insistant sur les différentes caractéristiques de ce langage qui sont adaptées à une simulation hiérarchique et multi niveaux.

Deux outils de simulation, logico-fonctionnelle et électrico-fonctionnelle, sont présentés. Ces deux outils présentent une avancée dans le domaine de la simulation, dans le but de garder la précision tout en diminuant le coût de simulation des circuits VLSI.

Une évaluation des différents langages de description selon leurs domaine d'application et propriétés est présentée. Au vue de cette évaluation, FIDEL s'insére en bonne place, tant au niveau des concepts que l'utilisation pratique.

MOTS-CLEFS

Langages de description du matériel, HDL, description fonctionnelle, description structurelle, description hiérarchique, simulation, simulation fonctionnelle, simulation logique, simulation électrique, simulation multi niveaux, simulation de mode mixte, système de CAO, dispositif MOS, VLSI.