USTL FLANDRES ARTOIS — LIFL — U.A 369 du C.N.R.S

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

Nº d'ordre : 502

THESE

Nouveau Régime

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

HOLLANT Thierry

ETUDE D'UNE APPROCHE OBJET POUR LA CAO ELECTRONIQUE : APPLICATION A LA SIMULATION LOGIQUE

Thèse soutenue le 1er mars 1990 devant la commission d'Examen :

Président	CORDONNIER V.	Université de LILLE I
Rapporteurs	TRULLEMANS Ch.	Université de LOUVAIN LA NEUVE
	BAKOWSKI P.	IRESTE de NANTES
Directeur de Thèse	MERIAUX M.	LIFL de LILLE
Examinateurs	PAWLAK A.	GMD SANKT AUGUSTIN (RFA)
	PETITPREZ B.	ISEN – LILLE

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

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

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

PROFESSEUR EMERITE

M. A. LEBRUN

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

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

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

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

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

PROFESSEURS - 1ère CLASSE

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

M. BRIDOUX Michel	Chimie-Physique
M. CELET Paul	Géologie Générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. DAUCHET Max	Informatique
M. DEBOURSE Jean-Pierre	Gestion des Entreprises
M. DHAINAUT André	Biologie Animale
M. DOUKHAN Jean-Claude	Physique du Solide
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
M. FAURE Robert	Mécanique
M. FOCT Jacques	Métallurgie
M. FRONTIER Serge	Ecologie Numérique
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. LABLACHE-COMBIER Alain	Chimie Organique
M. LACOSTE Louis	Biologie Végétale
M. LAVEINE Jean-Pierre	Paléontologie
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique Atomique et Moléculaire
M. LEROY Jean-Marie	Spectrochimie
M. LHOMME Jean	Chimie Organique Biologique
M. LOMBARD Jacques	Sociologie
M. LOUCHEUX Claude	Chimie Physique
M. LUCQUIN Michel	Chimie Physique
M. MACKE Bruno	Physique Moléculaire et Rayonnements Atmosph.
M. MIGEON Michel	E.U.D.I.L.
M. PAQUET Jacques	Géologie Générale
M. PETIT Francis	Chimie Organique
M. POUZET Pierre	Modélisation - calcul Scientifique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. SALMER Georges	Electronique
M. SCHAMPS Joel	Spectroscopie Moléculaire
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. TILLIEU Jacques	Physique Théorique
M. TOULOTTE Jean-Marc	Automatique
M. VIDAL Pierre	Automatique
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne	Composants Electroniques
M. ANDRIES Jean-Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BART André	Biologie animale
M. BASSERY Louis	Génie des Procédés et Réactions Chimiques

Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique Atomique et Moléculaire
M. BERTRAND Hugues	Sciences Economiques et Sociales
M. BERZIN Robert	Analyse
M.BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie Végétale
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie Civil
M. BOIVIN Jean-Claude	Spectroscopie
M. BOUQUELET Stéphane	Biologie Appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BRASSELET Jean-Paul	Géométrie et Topologie
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie Animale
M. CATTEAU Jean-Pierre	Chimie Organique
M. CAYATTE Jean-Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie Structurale
M. CHIVE Maurice	Composants Electroniques Optiques
M. COMYN Gérard	Informatique Théorique
M. COQUERY Jean-Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques et Sociales
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique Nucléaire et Corpusculaire
M. COUTURIER Daniel	Chimie Organique
M. CRAMPON Norbert	Tectolique Géodynamique
M. CROSNIER Yves	Electronique
M. CURGY Jean-Jacques	Biologie
Melle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	Géologie Appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean-Paul	Informatique
M. DELORME Pierre	Physioliogie Animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
M. DENEL Jacques	Informatique
M. DE PARIS Jean Claude	Analyse
M. DEPREZ Gilbert	Physique du Solide - Cristallographie
M. DERIEUX Jean-Claude	Microbiologie
Melle DESSAUX Odile	Spectroscopie de la réactivité Chimique
M. DEVRAINNE Pierre	Chimie Minérale
Mme DHAINAUT Nicole	Biologie Animale
M. DHAMELINCOURT Paul	Chimie Physique
M. DORMARD Serge	Sciences Economiques
M. DUBOIS Henri	Spectroscopie Hertzienne
M. DUBRULLE Alain	Spectroscopie Hertzienne
M. DUBUS Jean-Paul	Spectrométrie des Solides
M. DUPONT Christophe	Vie de la firme (I.A.E.)
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FAUQUAMBERGUE Renaud	Composants électroniques

M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie Sturcturale
M. GAMBLIN André	Géographie urbaine, industrielle et démog.
M. GLORIEUX Pierre	Physique moléculaire et rayonnements Atmos.
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie Physique
M. GOURIEROUX Christian	Probabilités et Statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean-Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie Organique
M. GUILBAULT Pierre	Physiologie animale
M. HENRY Jean-Pierre	Génie Mécanique
M. HERMAN Maurice	Physique spatiale
M. HOUDART René	Physique atomique
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et Statistiques
M. Jean Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme (I.A.E.)
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KREMBEL Jean	Biochimie
M. LANGRAND Claude	Probabilités et statistiques
M. LATTEUX Michel	Informatique
Mme LECLERCQ Ginette	Catalyse
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
Melle LEGRAND Solange	Algèbre
M. LEGRAND Pierre	Chimie
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne
M. LE MAROIS Henri	Vie de la firme (I.A.E.)
M. LEROY Yves	Composants électroniques
M. LESENNE Jacques	Systèmes électroniques
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique théorique
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU Jean-Marie	Optique-Physique atomique
M. MAIZIERES Christian	Automatique
M. MAURISSON Patrick	Sciences Economiques et Sociales
M. MESMACQUE Gérard	Génie Mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie Organique
M. MORTREUX André	Chimie Organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
Mme MOUYART-TASSIN Annie Françoise	Informatique
M. NICOLE Jacques	Spectrochimie
M. NOTELET Françis	Systèmes électroniques
M. PARSY Fernand	Mécanique

M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. STEEN Jean-Pierre	Informatique

Mener à bien un travail de thèse n'est pas chose facile. Mais la tâche aurait été impossible sans l'aide de toutes les personnes de mon entourage. Je les en remercie vivement, qu'elles soient citées ou non dans cette page.

L'ISEN a mis a ma disposition tous les moyens matériels et humains nécessaires pour mener à bien ce travail. Plus particulièrement, le département informatique et son responsable M. PETITPREZ m'ont aidé et soutenu tout au long de ces années.

M. MERIAUX a eu la patience d'encadrer mes travaux, et m'a servi de lien permanent avec l'Université des Sciences et Techniques de Lille Flandres-Artois.

M. BAKOWSKY, par ses conseils, m'a permis d'enrichir la teneur de ce travail. M. TRULLEMANS, par ses critiques m'a permis une meilleure formulation de mes idées. Tous deux m'ont fait le plaisir d'être rapporteur officiel.

M. CORDONNIER a accepté la présidence du jury.

M. PAWLAC, malgré la distance et le problème de la langue, a accepté le rôle d'examinateur.

Christelle SOYEZ a assuré la mise en page de ce rapport, et Nicole HUBERT en a assuré la relecture.

SOMMAIRE

pages

ILLUSTRATIONS

INTRODUCTION GENERALE

L'accroissement spectaculaire au cours des dernières années, de la complexité des circuits électroniques à très haut niveau d'intégration (VLSI) a rendu indispensable le recours à des outils de conception assistée par ordinateur (CAO).

Face à un besoin de productivité de l'industrie de la conception, des solutions complètes sont proposées par quelques constructeurs (Daisy, ES2, Mentor...), regroupant l'ensemble des logiciels nécessaires tout au long de la conception, dans un environnement de type station de travail UNIX (Sun, Appolo,...).

Ces logiciels reposent en général sur deux critères d'utilisation effective : une interactivité soignée, l'information manipulée étant par nature graphique (schématique, chronogramme) et des performances élevées en vitesse de traitement pour appréhender la complexité des problèmes posés en VLSI.

D'autre part, la communication et l'homogénéité entre ces outils passent par une gestion de base de donnée performante.

Enfin, le développement, la maintenance, et l'évolution de tels produits posent des problèmes de méthodologie de conception logicielle particulièrement important.

Face à la diversité de ces besoins, l'informatique traditionnelle met en oeuvre des logiciels complexes, pas toujours satisfaisants, imposant souvent une méthode de travail avec une vue particulière de la conception. Or on constate la montée en puissance de nouveaux outils informatique, basés sur des concepts de haut niveau. Les langages tels que Prolog ou Lisp, le concept de programmation orientée objet, ou les systèmes à base de connaissances forment autant de voies nouvelles, permettant a réalisation outils de CAO plus intelligents. Parmi ces techniques, la programmation orientée objet retient particulièrement l'attention.

Au delà d'un certain phénomène de mode, les concepts objets apportent un avantage indiscutable dans la gestion des environnements de multifenêtrage graphique interactif, la quasi-totalité de ces environnements s'inspirant directement de la programmation objet (MacIntosh, Presentation Manager,...).

Plus récemment, le monde des systèmes de gestion des bases de données, s'est intéressé au phénomène objet, et certains SGBD relationnels offrent maintenant une extension objet, en attendant l'arrivée des approches purement objet.

Dans le domaine méthodologie de conception logicielle, le concept objet est également fortement présent, ainsi la méthode française MERISE est actuellement en train de s'enrichir d'objets, de même le succès d'ADA dans le domaine du génie logiciel provient de ses qualités de modularité et d'encapsulation que proposent tous les langages objets.

Enfin, et peut-être de la façon la plus évidente, la notion même d'objet (informatique) semble très adaptée à la modélisation des objets (du monde réel).

L'introduction des concepts objets pour la réalisation d'outils de CAO électronique apparait donc attractive. Le seul défaut mais obstacle particulièrement important, réside dans les performances limitées de ce type de langage et leur gourmandise en place mémoire. Ce défaut peut être amélioré par des architectures de machines plus adaptées, par la possibilité de compilation du langage, et éventuellement par la réécriture des morceaux de code les plus critiques. Dès aujourd'hui, ces environnements permettent au moins un maquettage rapide de concepts très évolués.

Cette approche objet de CAO électronique est illustrée dans cette étude, par la réalisation d'un simulateur logique en environnement Smalltalk.

En effet, la simulation logique semble de manière intuitive être bien adaptée à une telle approche. D'autre part, Smalltalk et historiquement et conceptuellement, un environnement objet particulièrement représentatif.

La première partie de cette thèse constitue une présentation rapide des domaines mis en jeu : conception VLSI et techniques et langages issues de l'Intelligence Artificielle, dont la Programmation Orientée Objet fait partie et que l'on peut considérer comme concept fédérateur.

L'intérêt du rapprochement de ces 2 domaines est alors mis en évidence. Le choix de la réalisation est décrit par ses caractéristiques et ses objectifs.

Le simulateur est alors étudié en détail d'abord par les méthodes de description des circuits à simuler, puis par la gestion de l'interactivité, et enfin par son fonctionnemnt interne.

La dernière partie présente quelques extensions autorisées par l'approche objet dans une recherche de l'amélioration de performances.

1. Conception assistée par ordinateur des circuits VLSI

Nous décrirons, dans cette première partie, les méthodes de conception de circuits VLSI et les principaux outils informatiques se rapportant à chacune des étapes.

1.1. Evolution des besoins

Depuis environ 25 ans, nous sommes passés de l'intégration d'un composant unique à des systèmes de plus d'un million de transistors.

Cette évolution s'est faite à travers différentes générations :

* SSI (Small Size Integration) :
  jusque 100 transistors, soit quelques portes logiques sur un chip

* MSI (Medium Size Integration) :
  jusque 1000 transistors, soit une fonction de type compteur, ou ALU

* LSI (Large Scale Integration) :
  jusque 100000 transistors, avec par exemple les microprocesseurs 8 bits

* VLSI (Very Large Scale Integration) :
  100000 et plus , tels que les microprocesseurs 32 bits, les mémoires DRAM 4M bits

Dès l'apparition des premiers systèmes LSI, il s'est avéré que les techniques traditionnelles de conception n'étaient plus valables. Là où le travail de quelques mois-hommes suffisait, il fallait maintenant envisager des projets de plusieurs dizaines d'années-hommes.

La nécessité d'une telle évolution s'est surtout fait sentir depuis 1975 et elle s'est traduite par deux approches complémentaires :

* approche structurée de la conception [Mea80]
* utilisation intensive de la CAO

Il est important de remarquer que seule l'électronique numérique demande de tels niveaux d'intégration. Mais de plus en plus apparaît le mixage logique/analogique sur un même support. La part du logique représente toutefois environ 90% des VLSI [Man81].

D'autre part la conception complète d'un circuit (full-custom) se chiffre en dizaine de milliers de dollars, elle ne se justifie donc que pour des circuits à très large diffusion. Des techniques de conception plus "grossières" ont dû être mises en oeuvre pour autoriser malgré tout le développement de VLSI particularisés (ASIC) à un moindre coût.

1.2. Conception structurée

Il s'agit de découper la spécification initiale du problème en sous-problèmes qui pourront eux-même être partitionnés jusqu'à atteindre un niveau de complexité abordable pour une description en composants élémentaires, portes logiques ou transistors (figure 1).

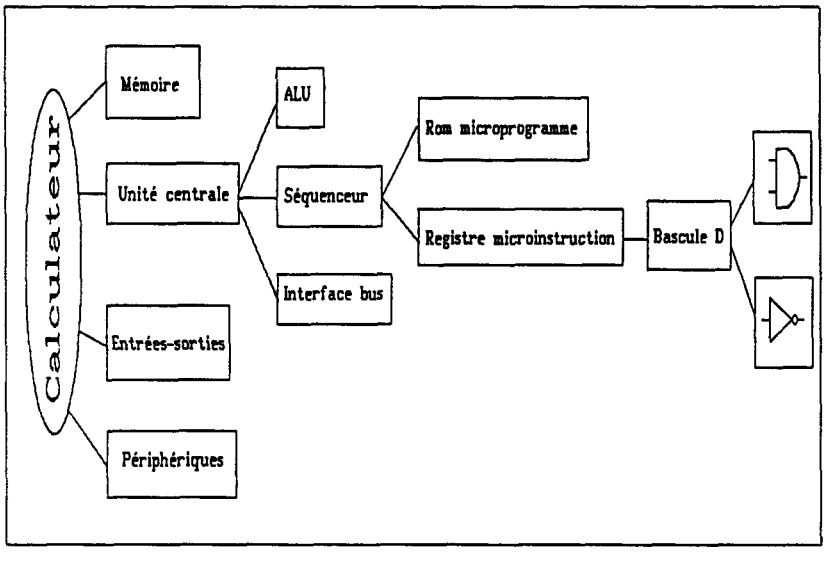


fig.1 Exemple de décomposition hiérarchique

On obtient ainsi une description hiérarchique, autorisant la répartition du travail sur toute une équipe de conception. Chaque passage d'un niveau à un autre est complètement validé afin d'éviter un retour sur les étapes précédentes.

L'approche est en premier lieu descendante (top-bottom). En fait, des retours arrières sur un niveau sont nécessaires, pour tenir compte des contraintes liées aux composants élémentaires utilisés pour la description au niveau inférieur. La conception vers le haut (bottom-up) prend parfois le relais, lorsqu'après une première descente dans les niveaux plus fins de spécification, on décide de se lancer dans la conception d'un ensemble de cellules de bases.

La décomposition par niveaux de la conception peut se faire de façon plus ou moins détaillée [MUR84] [Man81] [New81] [Kno81], nous retiendrons les 5 niveaux suivants :

* Algorithme
* Architecture
* Logique
* Electronique
* Layout

Chaque niveau dispose en général de son propre langage de description. Le passage d'un niveau à l'autre consiste donc en une "traduction" d'un langage en un langage plus élémentaire.

Le niveau "algorithme" pourra se décrire par un pseudo-code ou par un langage comportemental spécifique.

Au niveau "architecture", le concepteur raisonnera déjà en terme d'opérateurs et de chemins de données, on trouve alors une multiplicité de langages type transfert de registre (RTL : Register Transfer Language) tels que ceux décrits par [Kno81] [Erc85] [Lew83] [Sak83].

Le niveau "logique" décrira l'interconnexion de portes élémentaires à l'aide de langages de type netlist (description du réseau de portes et des connexions).

Le niveau "électronique" sera également un réseau interconnectant des composants actifs et passifs (transitors, résistances, capacités...).

Finalement le "layout" est par nature graphique (figure 2). Là encore, de nombreux langages permettent la description en terme de primitives graphiques, rectangles ou polylines tels que CIF ou GDS2.

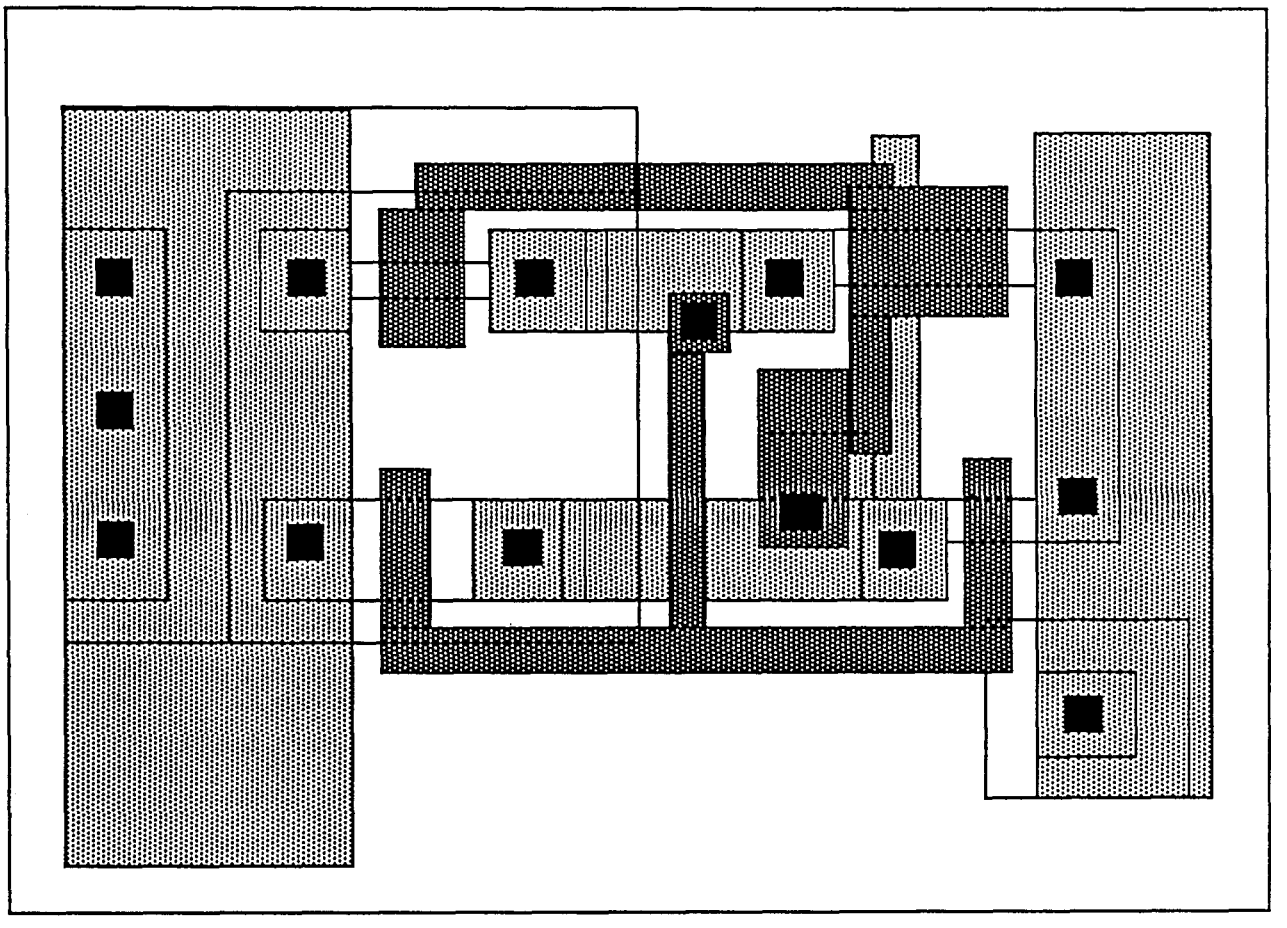


fig.2 dessin des masques d'un layout

Des études sur la normalisation des langages de description utilisés à chaque niveau sont en cours, afin d'autoriser l'échange entre concepteurs, et surtout entre systèmes de CAO.

Ainsi le langage VHDL [Sha86] [Wax89], dont l'étude a été initialisée par le Département de la Défense des USA et qui fait l'objet d'un groupe d'étude de normalisation IEEE, ou le langage EDIF [Eur86] [Wax89] poussé par un regroupement de constructeurs de stations de travail et d'outils CAO (Daisy, Mentor, Tektronix...) qui fait l'objet d'une norme ANSI.

La technologie utilisée pour la réalisation du circuit n'apparaît en principe qu'au niveau électronique. Elle doit être prise en compte le plus tard possible, afin de permettre une évolution du système indépendante de l'évolution technologique. Toutefois, ces contraintes interviennent obligatoirement dans les niveaux hauts de la conception, et obligent parfois à une reprise de spécification à un niveau supérieur.

Ainsi la conception de niveau logique ne peut s'abstraire de la technologie employée, puisque les primitives logiques disponibles peuvent être différentes [Mur84] : l'ECL réalise plus particulièrement des fonctions NOR, le MOS et le TTL travaillent en logique négative, le MOS aboutissant à des fonctions de complexité supérieure à celles obtenues en TTL (figure 3). La synthèse du niveau logique devra donc faire apparaître une décomposition à l'aide de telles primitives.

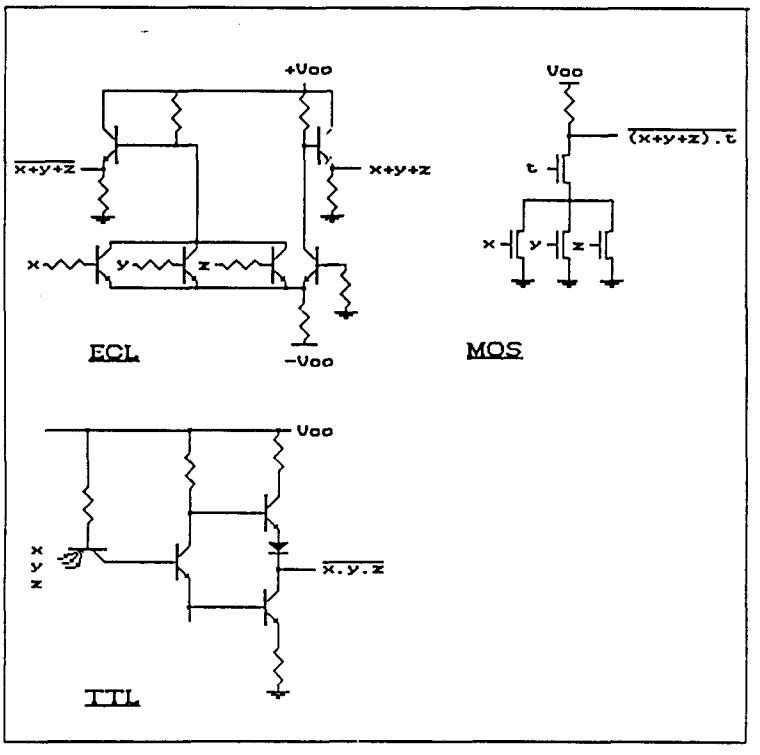


fig.3 schéma des portes de base ECL, TTL, MOS

1.3. Modularité

L'arborescence, résultant d'une conception structurée, aboutit à des circuits également structurés au niveau du layout. La description de chaque sous-bloc jusqu'au niveau le plus bas, permet une adéquation et une optimisation maximale aux besoins du système à concevoir. Toutefois, des solutions intermédiaires ont été développées, pour minimiser les coûts et les temps de conception, et rendre ainsi possible l'intégration de fonctions complexes, quelle qu'en soit la quantité produite.

On aboutit alors sur des règles de réalisation de sous-blocs et sur des règles de composition de ces sous-blocs [Lea86].

1.3.1. Réalisation des sous-blocs

* full custom
  optimisation en terme de surface et de performances par conception complète de chaque cellule

* full custom simplifié
  utilisation de structures régulières (registres, PLA, décodeurs...)

* cellules semi-custom
  interconnexion de transistors, de portes ou fonctions logiques prédéfinies, utilisant des bibliothèques de cellules standards

* cellules programmables
  fonctions réalisées par programmation de RAM, ROM, PLA...

1.3.2. Assemblage des sous-blocs

* placement et routage
  placement des blocs puis routage aléatoire (optimisation des longueurs de chemins) ou routage canal (blocs en structure régulière de tableau, interconnexion par les canaux interblocs)

* routage et positionnement
  définition de bus puis positionnement des blocs

* structures en tranches
  des cellules identiques sont assemblées pour traiter des informations de taille plus grande, soit en parallèle soit en série (cellules itératives)

* matrices de cellules
  pour les structures régulières de type mémoire

* mémoire distribuée
  assemblage de blocs dotés chacun de sa propre mémoire, unité de contrôle et logique de communication (structures systoliques, associatives...)

1.4. Les outils CAO

L'utilisation intensive de l'ordinateur est désormais une condition indispensable à la réalisation de tout projet. Différents logiciels sont concernés, à différents stades de la conception [Ped81].

1.4.1. La simulation

Une simulation sera appliquée à chacun des 5 niveaux cités au paragraphe 1.2, afin de valider l'étape.

En premier lieu, la simulation comportementale permet la vérification des algorithmes de fonctionnement global du système. On peut utiliser un langage informatique traditionnel ou un langage spécifique. A ce stade, aucun choix technique n'a encore été fait.

Le second niveau de simulation se base en général sur des langages à transfert de registres. Il permet de valider les choix techniques guidant le découpage du système en sous-ensembles.

Le passage aux composants logiques fait ensuite apparaître deux types de simulation :

- la simulation fonctionnelle
  le but est de vérifier la fonction réalisée, indépendamment de l'aspect technologique qui commence à apparaître

- la simulation temporelle
  en tenant compte des temps de propagation dus à la réalité physique des circuits, on peut détecter des erreurs de conception amenant des aléas temporels de fonctionnement. On peut également qualifier les vitesses de fonctionnement du circuit.

Enfin les deux dernières étapes de la conception, circuit électronique et layout, demandent une simulation fine des phénomènes électriques, en tenant compte par exemple des effets de ligne de transmission induits par les interconnexions, ainsi que les caractéristiques des transistors déduites des paramètres géométriques de leur réalisation.

Plus le niveau de simulation est proche du composant physique, plus les temps de simulation sont importants, puisque le nombre de composants de base augmente ainsi que le nombre de paramètres qualifiant chaque composant. Au niveau électrique, on dépasse rarement quelques centaines de composants, les temps de calcul se chiffrent alors en heures CPU, sur des machines puissantes.

Certains simulateurs couvrent les besoins de simulation sur plusieurs de ces niveaux. Il s'agit alors de simulateurs multi-niveaux, quand on reste sur un plan fonctionnel [Mer81], ou en mode mixte lorsque les phénomènes électriques peuvent être modélisés [Bor83] [New81] [Man81].

1.4.2. Testabilité

Tout au long de la conception, il faut penser à la possibilité de vérifier le bon fonctionnement du circuit. Ceci oblige à introduire en général des structures n'ayant aucun rôle dans la fonction du circuit mais permettant l'accès à ses états internes. Des méthodes ont été développées, telles que le Level Sensitive Scan Design (LSSD), les Scan Path ou les Built-In Logic Block Observation (BILBO) [Wil81] [Sch83] [Eic83].

En parallèle à ces méthodes de conception, il faut prévoir les séquences de test qui permettront la détection d'éventuels mauvais fonctionnements du circuit fabriqué. Une bonne séquence de test doit être la plus courte possible tout en vérifiant l'ensemble des composants du circuit.

La définition de ces séquences de test fait appel à deux outils complémentaires.

Le premier reprend la simulation logique du circuit mais en fournissant les sorties obtenues pour différents types de panne. Ces simulateurs supposent généralement des fautes uniques, de type collage à 0, collage à 1, ou court-circuit. Ils permettent de vérifier qu'une entrée donnée détecte un certain nombre de pannes, en produisant des sorties incorrectes.

Le second logiciel, en général couplé au précédent, génère des séquences de test les meilleures possibles, par des méthodes heuristiques [Bar86].

1.4.3. Implantation et vérification

La dernière étape de la conception doit aboutir sur les masques de fabrication du circuit. Il s'agit alors d'un problème graphique. Les différents constituants du circuit doivent être placés de façon à optimiser la place occupée sur le silicium.

Des contraintes technologiques doivent être respectées pour garantir un fonctionnement correct après fabrication (largeur de piste, distance entre éléments).

Lorsque les masques sont dessinés, différents logiciels de vérification peuvent intervenir. D'abord pour vérifier le bon respect des règles de dessin (vérificateur de garde en temps réel pendant la phase de dessin ou en temps différé après la saisie complète des masques), ensuite pour éventuellement extraire le schéma électrique et vérifier sa conformité avec le schéma de l'étape de conception précédente. Les différentes simulations peuvent être relancées avec les informations issues de cette extraction.

1.5. Conclusion

Conception et outils de CAO sont étroitement mêlés, on retrouve la même volonté de hiérarchie et de modularisation dans les différentes phases qui mènent au circuit final.

Il est donc souhaitable d'avoir un environnement CAO qui s'adapte avec souplesse aux besoins du concepteur, afin de laisser une liberté totale pour la créativité. Le dialogue homme-machine est un des points essentiels pour une bonne productivité, c'est à dire un produit final fiable et dont le temps de conception n'est pas prohibitif.

La deuxième caractéristique d'un bon outil de CAO, sera sa capacité à gérer efficacement les données de la conception, tant en mémoire centrale que sur mémoire de masse.

Enfin, la communication avec d'autres environnements de CAO et la facilité à intégrer de nouveaux formats de données constitueront un atout permettant l'intégration au sein de toute la chaîne de conception, en attendant une généralisation d'un langage standard de description tel que VHDL cité précédemment.

Une vision idéale de la CAO, le compilateur de silicium, serait d'obtenir une automatisation complète de la conception. Les techniques actuelles de synthèse ou d'optimisation sont des solution partielles à quelques problèmes spécifiques. Un tel système exige de doter l'ordinateur de toute la connaissance nécessaire, à tous les niveaux de la conception. Ceci relève typiquement de l'intelligence artificielle. Nous allons donc examiner dans la partie suivante, les langages et techniques issues de l'intelligence artificielle, et nous étudierons quelques exemples d'applications à la CAO.

2. Introduction aux techniques de l'intelligence artificielle

Née à la fin des années 1950, l'intelligence artificielle connait un renouveau depuis une dizaine d'année, grâce aux progrès tant de la technologie que des concepts fondamentaux. Parmi tous les apports de l'intelligence artificielle, nous examinerons successivement les principaux langages qu'elle a suscités, et la technique des systèmes à base de connaissances encore appelés systèmes experts. Des applications seront ensuite présentées afin de démontrer la richesse que ces techniques peuvent apporter dans un environnement de CAO électronique. [Cha88] [Win81] [Win84] [Roc89] [Lau86] constituent des éléments bibliographiques de présentation plus détaillée de l'intelligence artificielle.

2.1. Les langages de l'intelligence artificielle

2.1.1. Historique

Dès la fin des années 50 apparait la nécessité de développer de nouveaux langages pour traiter des informations symboliques. Les problèmes de démonstration automatique de théorèmes ou de jeu d'échec sur ordinateur conduisent en 1961 à l'apparition du langage LISP, développé par J. Mac Carthy au MIT.

A la fin des années 60, pour résoudre des problèmes de traitement de langage naturel, Colmerauer développe le langage PROLOG à partir des travaux des mathématiciens Herbrand et Robinson.

Enfin, au début des années 70, au Xerox Parc (Palo Alto Research Center), est développé un environnement de programmation, Smalltalk, destiné à faciliter l'usage de l'ordinateur par une interactivité poussée.

Parmi ces langages, seuls les deux premiers connaissent une diffusion rapide, grâce à des implantations sur des machines de toutes marques. Le langage LISP en particulier, de par son ancienneté, est disponible sur tout type de machines. Malheureusement, beaucoup de ces versions sont incompatibles par les extensions au langage de base apportées sur chaque implémentation. PROLOG a connu le même problème, mais à une plus petite échelle, et ces deux langages font maintenant l'objet de travaux de normalisation internationale.

Smalltalk, dans sa version Smalltalk/80, commence à se diffuser depuis début 80. Jusqu'alors, il n'était disponible que sur des machines spécifiques (stations de travail XEROX) à cause des contraintes matérielles qu'il suppose (souris et écran graphique bitmap haute résolution). L'avènement des stations de travail (Sun, Appolo, Tektronik...), et des micro-ordinateurs haut de gamme (80286/386,mémoire supérieure à 1Mo, écran EGA-VGA) contribue au succès grandissant de ce langage.

LISP, PROLOG et SMALLTALK ont été retenus dans cette présentation, parce qu'ils sont les représentants caractéristiques des notions (respectivement) de programmation fonctionnelle, programmation logique et programmation objet.

2.1.2. Caractéristiques générales de ces langages

a. LISP

LISP est un langage de traitement de listes. Ces listes peuvent contenir d'autres listes ou des atomes. Un atome est une information symbolique, par opposition aux notions de variables typées des langages traditionnels.

Le développement d'une application LISP consiste en l'écriture de fonctions, qui viennent se rajouter aux fonctions LISP prédéfinies. On travaille donc par enrichissement de l'environnement existant.

Les fonctions LISP sont décrites par des listes et appliquées en notation préfixée sur les arguments désirés. On a donc une homogénéité totale du langage puisque données et programmes ont la même forme de liste.

Le noyau de l'interpréteur réalise des évaluations en chaînes, chaque fonction pouvant appeler d'autres fonctions, ou s'appeler récursivement. L'évaluation d'une fonction retourne toujours une valeur unique qui peut être un atome ou une liste.

exemple :

```
(DEFUN INCREMENTER (X) (+ X 1))
```

La fonction DEFUN a 3 arguments. Le premier est un atome qui désigne un nom de fonction. Le second une liste des arguments de la fonction, et le troisième définit le corps de la fonction. DEFUN permet donc d'ajouter une nouvelle fonction à l'environnement de l'interpréteur LISP. L'évaluation de l'expression suivante :

```
(INCREMENTER 4)
```

retournera donc la valeur 5.

Les avantages du langage sont :

* manipulation symbolique
* structure de données puissante (liste)
* mécanisme d'évaluation de fonctions
* usage intensif de la récursivité

et les inconvénients :

* besoin de mémoire important
* syntaxe lourde (parenthésage intensif !)
* incompatibilité avec les langages traditionnels
* temps d'exécution importants sur architecture classique

b. PROLOG

Un interpréteur PROLOG manipule des prédicats du premier ordre, grâce à un double mécanisme de résolution et d'unification. Le développement d'une application PROLOG est donc davantage une description du problème en forme de règles, plutôt que la description de l'algorithme de traitement.

Le mécanisme de résolution assure la recherche d'une solution par application successive des règles décrivant le problème à résoudre. En cas d'échec dans l'essai d'une règle, un retour arrière permet l'examen des branches restant à parcourir.

L'unification permet, en paramétrant le problème posé, de donner les valeurs particulières pour lesquelles une solution existe.

exemple :

```
père(paul,pierre).
père(pierre,jean).
grand-père(x,y):-père(x,z),père(z,y)
```

Les prédicats père et grand-père décrivent des notions généalogiques et des faits connus.

L'interrogation suivante :

grand-père(x,jean)?

va déclencher une recherche des solutions, étant donné la connaissance du sujet généalogie introduite précédemment. La solution retournée sera :

x = paul

Tout comme LISP, PROLOG manipule des structures de listes participant à l'unification.

Les avantages de ce langage sont :

* mécanisme de raisonnement intégré (rapidité de développement, maquettage)
* information symbolique
* structure de données en liste possible
* gestion d'une base de faits (base de donnée)
* usage intensif de la récursivité

les inconvénients sont :

* mécanisme de raisonnement intégré (il faut s'y plier)
* besoin de mémoire important
* incompatibilité avec les langages traditionnels
* calcul numérique très mal intégré
* temps d'exécution importants sur architecture classique

c. SMALLTALK

Une description détaillée des concepts de SMALLTALK est développée au paragraphe 2.5. Seuls les principaux avantages et inconvénients liés à son utilisation seront présentés ici.

Parmi les avantages on peut citer :

* structures de données puissantes
* gestion totale de l'interactivité (souris + écran)
* rapidité du développement, maquettage
* programmation récursive possible

les inconvénients sont :

* besoin de mémoire important
* peu ouvert aux langages et aux systèmes d'exploitation classiques
* uniquement sur des machines type station de travail
* temps d'exécution importants sur machines à architecture traditionnelle

La figure 4 est une comparaison de syntaxe des 3 langages, pour l'écriture d'une fonction identique.

```
member (élément,[élément|queue]) :- !.
member (élément,[nonélément|queue])
                    :- member (élément,queue).
```

fig.4.a Exemple de fonction de test d'appartenance à une liste version PROLOG

```
(defun member (élément liste)
       (COND
          ( (null liste) false)
          ( (eq élément (car liste)) true )
          ( true (member élément (cdr liste)))
       ))
```

fig.4.b Exemple de fonction de test d'appartenance à une liste version LISP

```
! Object Method !
appartientA:uneListe
        uneListe isEmpty ifTrue:[^false].
        self = uneListe first ifTrue:[^true].
        ^self appartientA:uneListe removeFirst
```

fig.4.c Exemple de fonction de test d'appartenance à une liste version SMALLTALK

2.2. Systèmes à base de connaissances

Apparu au début des années 70, le concept de système à base de connaissance (système expert) se caractérise par la séparation de la connaissance et du programme de traitement informatique (moteur d'inférence).

La connaissance est introduite selon un certain mode de représentation propre au système choisi, le moteur d'inférence va manipuler cette connaissance selon une certaine stratégie.

2.2.1. Représentation de la connaissance

a. par règles

Une règle est une expression du type :

SI prémisse ALORS conclusion

La partie prémisse est constituée d'hypothèses sur des faits qui, si elles sont vérifiées, permettent d'affirmer la conclusion. Selon les types de faits et d'hypothèses admis, on distingue des sytèmes :

* d'ordre 0 où les faits sont des affirmations ou des négations simples
* d'ordre 0+ où les variables et les comparaisons sont admises
* d'ordre 1 quand les règles sont paramétrables (cas de PROLOG)
* d'ordre 2 enfin lorsque les opérateurs de relation sont eux-mêmes variables

Plus l'ordre est élevé, plus la base de connaissance peut s'exprimer sous forme condensée.

b. frames, réseaux sémantiques, représentation objet

Ces représentations permettent une structuration de la connaissance. Ainsi les frames caractérisent une entité par des attributs (facettes), des actions procédurales peuvent être associées à toute tentative d'accès à ces attributs (démons).

Les réseaux sémantiques définissent une classification entre les entités, par des liens de type (x est un y) ou (x est sous y). Enfin la représentation objet amène les principes d'héritage et de communication par messages.

Une représentation mixte, règle et connaissance structurée, permet la modélisation de tout type de problème.

2.2.2. Stratégie du moteur d'inférence

Le moteur d'inférence a pour but d'appliquer les règles pour faire évoluer la base de faits et, si possible, arriver à une solution.

Les règles peuvent être appliquées en chaînage avant, c'est-à-dire selon le principe de déduction classique hypothèse-conclusion. On peut également appliquer un chaînage arrière, dans ce cas on se fixe un but à prouver et on remonte vers les hypothèses.

Le choix intelligent de la prochaine règle à utiliser est primordial pour la rapidité du système. On peut ainsi explorer les règles en profondeur, en poussant chaque raisonnement au bout avant de commencer le suivant, ou en largeur, en appliquant toutes les règles possibles. L'ordre et la priorité des règles sont également critiques pour aboutir rapidement au résultat.

Toutes ces stratégies peuvent être combinées et évoluer dynamiquement par l'introduction de métaconnaissance (connaissance sur la connaissance).

2.2.3. Avantages et inconvénients

Un système à base de connaissance peut être utilisé quand il n'existe aucune bonne solution algorithmique traditionnelle. L'introduction de connaissances intuitives permet en effet la restriction de l'espace des solutions et augmente la vitesse de résolution d'un problème.

Une solution optimale n'est pas garantie, mais on obtient en général une bonne solution. D'autre part, de tels systèmes sont évolutifs, on peut sans cesse enrichir la connaissance.

Parmi les inconvénients, il faut remarquer la difficulté à faire ressortir la connaissance lorsqu'elle n'est pas formalisée et le coût de développement de tels systèmes.

2.3. Intelligence artificielle et CAO

L'apparition de l'intelligence artificielle en CAO est visible sur quatre niveaux : tout d'abord par l'influence des langages tels que LISP ou PROLOG, ensuite par divers outils résolvant des problèmes ponctuels en aide au concepteur, puis par des systèmes de synthèse automatique de circuits, et enfin par des systèmes intégrant globalement toutes les étapes de la conception.

2.3.1. Langages et CAO

LISP, de par son ancienneté et son origine U.S., est sans doute le langage le plus utilisé dans les applications de CAO.

La mise en évidence d'une véritable "culture" LISP et CAO, dans [Shr83], fait ressortir les avantages suivants :

* attrait du symbolique
* possibilité d'extension du langage en des langages spécifiques
* homogénéité données-fonctions qui aboutit à un style de programmation "guidé par les données"

Ceci les ayant conduit à des réalisations telles que le langage DPL de layout et son éditeur Daedalus, ou encore des générateurs de PLA, ou de chemins de données (datapath), sur des machines LISP.

Un second point remarquable est l'inspiration LISP évidente dans la syntaxe de la norme EDIF, dont un exemple vous est donné à la figure suivante.

```
(view schematic SK
( interface
   (comment "two input nand gate")
   (define inout port (multiple VCC GND))
   (define in port (multiple A B))
   (define out port (multiple C))
   (permutable A B)
   (body symbol
      (portImplementation A
         (figureGroup symbol
         (dot (point 0 40))))
      (portImplementation B
         (figureGroup symbol
         (dot (point 0 20))))
      (portImplementation C
         (figureGroup symbol
         (dot (point 70 30))))
      (figureGroup symbol
         (path (point 0 40)(point 10 40))
         (path (point 0 20)(point 10 20))
         (path (point 65 30)(point 70 30))
         (circle
            (point 60 30)(point 65 30))
         (shape
            (point 40 10)(point 10 10)
            (point 10 50)(point 40 50)
            (arc
                (point 40 50)
                (point 60 30)
                (point 40 10)))...
      )...
   )
)...)
```

fig.5 Exemple de syntaxe EDIF

Les possibilités de PROLOG sont également exploitées dans des systèmes tels que PROVE [Sri88], dont le but est la vérification d'un réseau de portes en logique combinatoire. L'auteur utilise une simulation hybride (signaux logiques + symboliques) pour découper un problème complexe en sous-problèmes.

Par exemple, la vérification d'une unité arithmétique et logique sera plus simple en considérant des opérandes symboliques et une sélection d'opération en binaire.

On obtient alors une expression algébrique pour chaque opération réalisée par l'ALU. PROVE utilise à la fois les possibilités symboliques du langage, la syntaxe du langage pour la description des circuits, et les capacités de raisonnement intégrées.

[Wat86] est un second exemple de mise en oeuvre directe des caractéristiques de PROLOG. Ce système effectue une réécriture d'un réseau de portes logiques traditionnelles, en un réseau équivalent de portes en technologie I2L. Là encore, l'expression et la résolution du problème s'expriment très naturellement en PROLOG.

2.3.2. Systèmes spécifiques à base de connaissances

La diversité des techniques employées en cours de conception d'un VLSI, demande une pluridisciplinarité au sein de chaque équipe de conception. L'approche système expert est un moyen pour compenser l'absence de compétences dans un domaine particulier, ou pour se décharger de tâches fastidieuses.

Ainsi le système PLA-ESS [Bre85] intègre la connaissance nécessaire au choix d'une méthode de test de PLA. Le choix, parmi 7 méthodes, se fait à partir de quatre classes de critères :

* caractéristiques de testabilité désirées
* modification sur le dessin original du PLA
* environnement de test disponible
* coûts induits

Weaver [Joo85] dispose de toute la connaissance d'experts confirmés en routage (routage en grille). Ecrit en OPS5 (générateur de système expert), il fait collaborer six systèmes experts dans une architecture à tableau noir (blackboard architecture).

Chaque expert émet un avis sur la prochaine action à entreprendre, selon sa propre base de connaissance. Un septième expert sert d'arbitre entre les six précédents pour décider de l'action à entreprendre. Les expertises proposées concernent :

* la propagation de contraintes (106 règles)
* la longueur des connexions (30 règles)
* les contraintes horizontales-verticales (68 règles)
* la fusion de lignes et de colonnes (70 règles)
* l'encombrement des canaux (10 règles)
* le bon sens (31 règles)
* l'arbitre (26 règles)

Un tel système épargne donc un travail long et fastidieux où seules la pratique et l'expérience permettent d'obtenir de bons résultats.

2.3.3. Synthèse automatique

Le Design Automation Assistant (DAA) de Kowalsky [Kow84] est remarquable pour sa méthodologie de conception. Il est basé sur le découpage en quatre niveaux de la conception proposé au Carnegy Mellon University (projet CMU/DA) :

1. niveau algorithme (langage ISPS)
2. réseau indépendant de la technologie (langage SCS)
3. réseau dépendant de la technologie (langage DIF)
4. réseau dépendant de la fabrication

Le but du système DAA est de passer du niveau 1 au niveau 2, et donc de "traduire" le langage ISPS en SCS.

Un premier prototype (70 règles) a été bati par interview de quatre concepteurs. Ce prototype a ensuite été utilisé pour la conception d'un microprocesseur de type 6502. La critique du résultat par d'autres concepteurs a permis d'enrichir la base de connaissances à 300 règles. Le système a alors été validé par la conception d'une architecture 370 simplifiée d'IBM. Le résultat a été approuvé par les concepteurs de chez IBM.

Le second intérêt de ces travaux est la mise en évidence de quatre étapes dans le travail de l'expert humain, étapes qui ont été reproduites dans le système expert :

* allocation globale (78 règles)
* allocation des chemins de données (55 règles)
* allocation des modules (47 règles)
* amélioration globale (46 règles)

Réalisé en OPS5 dans un environnement LISP, le DAA utilise deux fonctions d'estimation écrites en langage C, le système OPS5 étant très mal adapté au calcul numérique.

Des outils de synthèse sont également présentés dans [Sau86]. Basés sur le système de spécification et de simulation CADOC, ces travaux incluent une synthèse automatique de fonctions logiques avec optimisation logique et topologique, ainsi qu'un générateur de contrôleurs. L'intérêt de cette étude est la comparaison d'une version PROLOG et d'une version PASCAL du même système.

Pour le temps de développement, PROLOG a demandé une semaine de travail pour 200 lignes, PASCAL a nécessité 3 mois pour 3000 lignes.

Le temps d'exécution est bien entendu favorable au PASCAL (facteur 2 à 3) mais les solutions obtenues sont meilleures pour la version PROLOG (jusque 10% de termes produits en moins sur une synthèse de PLA).

2.3.4. Systèmes d'aide à la conception

V. Begg dans [Beg84], représente le système de CAO idéal comme guidé par une interface intelligente de type "conseiller", assurant un dialogue avec les concepteurs à toutes les étapes de la conception.

Le système doit accompagner le concepteur dans la manipulation des outils traditionnels, dans l'accès aux informations au sein de bibliothèques de données et de bases de connaissances de grandes tailles.

La conception ne doit pas, et ne peut pas, devenir complétement automatique, les technologies évoluant trop rapidement.

Le système conseiller doit donc permettre au concepteur d'intervenir à tous moments, pour apporter les modifications qui lui semblent nécessaires. La technique actuellement la plus apte à la réalisation d'un tel environnement est l'architecture à tableau noir où plusieurs systèmes experts collaborent, chacun ayant un "point de vue" bâti sur une connaissance spécifique. Ces systèmes experts devront être accompagnés d'une structure de représentation des informations puissante.

De tels systèmes commencent à apparaître tels que le planificateur ADAM [Kna86], qui génère un plan d'utilisation de différents outils de conception automatique à travers neuf systèmes experts communiquant par messages. La base de connaissance exprimée dans un sur-ensemble de LISP, est structurée en frames, regroupés en trois classes : tâche, matériel et style de conception.

Pour Brewer et Gajski [Bre86], la conception peut être guidée par un ensemble de systèmes experts communiquants, un par niveau d'abstraction, en utilisant une approche d'affinement itératif : chaque phase est évaluée et recommencée avec modifications des caractéristiques initiales si elle n'est pas satisfaisante.

Chaque expert réalise les tâches suivantes :

* propagation des contraintes
* choix d'un style et d'une stratégie
* raffinement
* optimisation
* évaluation

2.4. Conclusion

LISP, PROLOG et Smalltalk sont tous trois des langages symboliques utilisant au maximum la récursivité. Ils ont donc une grande puissance de traitement sur les données abstraites, mais en contrepartie les temps d'exécution sont souvent longs sur des machines à architecture traditionnelle, la place mémoire demandée étant grande et mal maitrisée. Ils restent donc éloignés des besoins en performance de la production industrielle, sauf dans des maquettages d'applications où ils sont particulièrement performants. En particulier pour le calcul intensif, la nécessité d'exploiter au mieux l'architecture de la machine, amène un choix de langages performants tels que le C, au détriment de la puissance des concepts manipulés. L'utilisation de machines dédiées pour le langage LISP telles que les machines Symbolics pour LISP amènent un gain de performance appréciable pour les langages étudiés. Malheureusement elles sont peu diffusées donc de coût élevé, et leur spécialisation les isole de l'informatique traditionnelle.

Par contre, l'arrivée des concepts objets à tous les niveaux de l'informatique, permet d'espérer une adaptation des architectures traditionnelles aux objets. De telles machines s'intègreraient naturellement à l'informatique actuelle de la même façon que les langages C ou Pascal supportent des extensions objets.

Les systèmes à base de connaissances, apportent une solution à une catégorie de problèmes mals résolues par l'algorithme. Au delà, des problèmes de propriété industrielle de la connaissance, ou de rapidité d'évolution de cette connaissance dans le domaine de la CAO électronique, on trouve à nouveau les problèmes de performances et d'intégration à la CAO traditionnelle.

Une CAO orientée objet peut constituer une passerelle entre le monde symbolique, de manipulation de connaissances, et une informatique classique.

2.5. Présentation des concepts SMALLTALK

2.5.1. Objet et Classe

La programmation orientée objet suppose en premier lieu l'identification des *objets* manipulés dans le problème que l'on désire traiter.

Un objet est une entité, dotée d'attributs statiques (des variables) et dynamiques (des programmes associés). Les attributs d'un objet dépendent de la *classe* à laquelle il appartient.

Lorsqu'une classe est définie, on peut créer des objets appartenant à cette classe par le mécanisme d'*instanciation*.

Chaque instance possède alors ses variables propres (*variables d'instance*), des variables partagées avec les autres objets de la même classe (*variables de classe*), et des variables globales accessibles à tous les objets du système.

Les variables SMALLTALK ne sont pas typées. Elles servent simplement à désigner d'autres objets. Une variable pourra donc au cours du temps désigner un objet de type nombre entier, puis un objet de type tableau (tableau d'objets !), pour terminer en désignant un objet de type chaîne de caractères. C'est la caractéristique commune aux langages issus de l'intelligence artificielle, de manipulation de symboles.

Le comportement dynamique d'un objet est défini par un certain nombre de programmes regroupés dans la classe. Il est donc le même pour tous les objets d'une même classe.

2.5.2. Messages et Méthodes

Tout "programme" SMALLTALK est en fait une succession d'envois de *messages*, destinés aux objets définis dans le système.

Un message est une requête pour obtenir une information de la part d'un objet.

Pour répondre à cette requête, l'objet connaît, de par sa classe, une *méthode* pour calculer la valeur désirée.

Une méthode est à nouveau un "programme" SMALLTALK, donc une succession de messages, qui aboutit à une valeur finale, retournée à l'objet émetteur du message.

Ainsi la structure SI/ALORS/SINON de la programmation classique devient ici un message " IfTrue:argument1 IfFalse:argument2 " envoyé à un objet de classe Boolean. Si cet objet est vrai, il évaluera le premier argument et retournera la valeur résultante. Sinon, il évalue le second argument et retourne la valeur résultante.

Les méthodes sont définies dans les classes. Tous les objets d'une même classe ont donc le même comportement lors de la réception d'un message. Le résultat retourné sera différent pour chaque objet puisque les valeurs des variables d'instances sont propres à chaque objet.

L'éclatement de la programmation en méthodes dans différentes classes confère aux langages objets une remarquable modularité. Les méthodes sont en général de petite taille (SMALLTALK = PARLER PEU), tout traitement spécifique étant assuré par l'objet concerné.

D'autre part, les méthodes sont définies pour une classe d'objets et non pour une application particulière, elles sont donc ré-utilisables dans toute autre application. L'écriture de nouvelles applications, ou l'adjonction de fonctionnalités supplémentaires à une application existante se fait alors de façon rapide et sûre.

On retrouve en particulier ces concepts dans bon nombre d'ateliers de génie logiciel, et plus spécialement dans le langage ADA, dont la notion d'encapsulation est proche de la programmation objet.

Un objet peut envoyer des messages à tous les objets qu'il connait, c'est-à-dire à ceux pointés par une variable d'instance, une variable de classe ou une variable globale. Mais il peut également s'envoyer des messages en indiquant comme destinataire *self* qui le désigne lui-même.

2.5.3. Hiérarchie des classes et héritage

Une classe est toujours définie comme *sous-classe* d'une classe plus générale. Ainsi, une classe des chiens peut être sous-classe de la classe des mammifères qui sera elle-même sous-classe de la classe des êtres vivants, qui finalement sera sous-classe de la classe Object.

La classe *Object* est la classe la plus générale de SMALLTALK, elle est au sommet de la hiérarchie des classes. La figure 6.a représente l'ensemble des classes prédéfinies de Smalltalk, sous-classes directes d'Object. La figure 6.b donne le détail de l'arbre des sous-classes de la classe Collection.

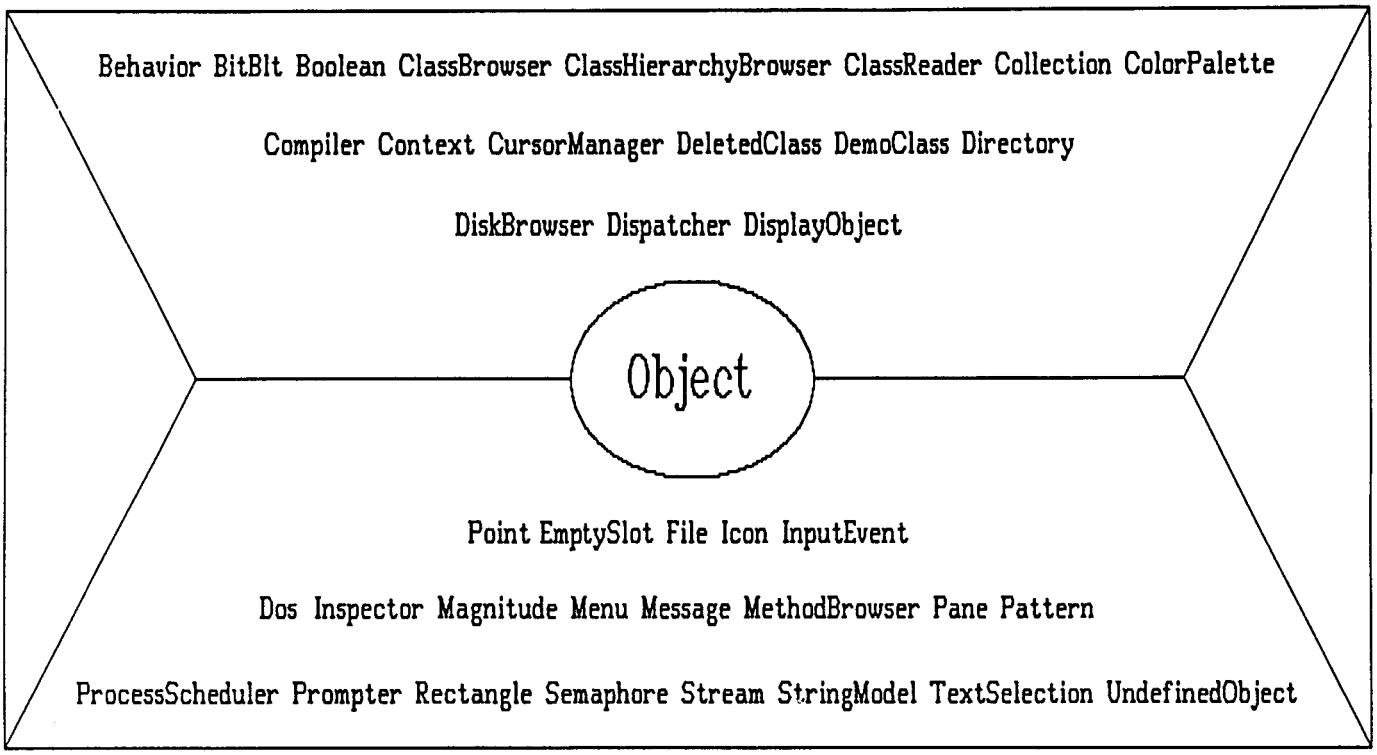


fig.6.a liste des sous-classes directes d'Object

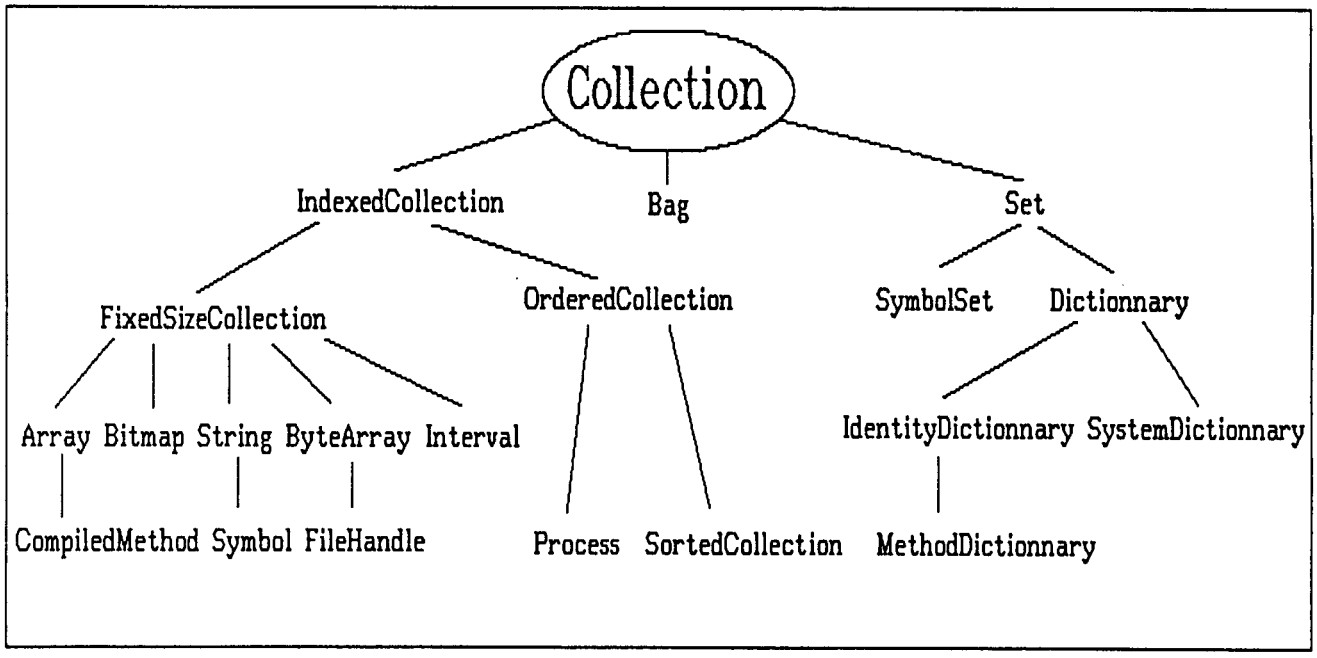


fig.6.b arbre des sous classes de Collection

A chaque niveau de la hiérarchie sont définies des variables de classe, des variables d'instance et des méthodes. Un objet créé dans la classe la plus basse aura accès à toutes les variables de classes définies dans sa classe et dans ses sur-classes. Il possédera les variables d'instances décrites à chaque niveau de la hiérarchie. Enfin, il *héritera* de toutes les méthodes définies dans sa classe et ses sur-classes.

Le premier intérêt de l'héritage est une description progressive du comportement complet d'un objet.

Ainsi la méthode "aboyer" pour un objet chien provient directement de la classe des chiens, mais la méthode "allaiter" est définie un fois pour toute dans la classe des mammifères, elle est partagée identiquement par tous les objets qui auront la classe mammifère dans leurs sur-classes.

Le second intérêt de l'héritage est la possibilité de redéfinir une méthode déjà présente dans une sur-classe. La méthode de la sur-classe est alors masquée par la nouvelle méthode de même nom. Ceci permet de particulariser certains comportements.

Par exemple, la classe Object contient une méthode "isNil", qui consiste à retourner la valeur faux. Seule la classe UndefinedObject redéfinit cette méthode en retournant la valeur vraie. Cette classe possède une instance unique appelée nil, et symbolisant un pointeur vide (objet non défini).

Dans certains cas, on redéfinit la méthode pour compléter le traitement qu'elle effectue. On pourra alors ,dans la nouvelle définition, appeler l'ancienne version de la méthode, définie dans une surclasse. Ceci pose un problème de récursion puisque la méthode doit appeler une méthode de même nom. Pour autoriser cela, on définit la variable *super*. Cette variable désigne l'objet lui-même tout comme self. Mais la recherche de la méthode commence dans la classe immédiatement supérieure à celle contenant la méthode en cours d'exécution. C'est donc bien la version de la sur-classe qui sera exécutée.

La méthode étant associée à une classe, il n'y a pas d'ambiguïté à définir plusieurs méthodes de même nom pour des classes distinctes. Cette propriété, appelée *polymorphisme*, permet de s'adresser de la même façon à des objets de structures tout à fait différentes, la structure interne est alors cachée pour les objets extérieurs.

Enfin, certaines classes ne possèdent pas d'instances, elles ne jouent qu'un rôle hiérarchique, en mettant en commun un certain nombre de méthodes. Elles sont appelées classes *abstraites*. Ainsi la classe mammifère ne possédera jamais d'instance puisqu'il n'existe pas d'animaux ne possédant que les caractéristiques communes aux mammifères, sans posséder de caractéristiques propres.

2.5.4. Les méta-classes

La notion de classe elle-même peut être définie comme un objet.

Chaque classe est objet unique de sa *méta-classe*. Les méta-classes décrivent la même arborescence que les classes. Au sommet de l'arbre, la méta-classe *Object Class* est sous-classe de la classe *Class*.

La classe Class contient, en particulier, les méthodes de définition de la classe et les méthodes d'instanciation.

Ainsi, la méthode "addClassVarName:" permet d'ajouter une variable de classe à la liste des variables déjà définies, de même la méthode "instVarName"" définit la liste des variables d'instance.

La méthode "addSelector:withMethod:" permet quant à elle la compilation d'une chaîne de caractères en une nouvelle méthode ajoutée aux méthodes déjà définies pour la classe.

Enfin, la méthode *new* permet la création d'un objet instance de la classe. Cette méthode est très fréquemment redéfinie pour permettre une initialisation de l'objet créé en même temps que sa création. Sinon toutes les variables d'instances sont initialisées à nil.

Le compilateur est également un objet, accessible depuis n'importe quelle méthode. On a donc la possibilité d'enrichir dynamiquement le comportement du système, en définissant de nouvelles classes, et en compilant de nouvelles méthodes en cours d'exécution de méthodes.

Cette caractéristique se retrouve aussi dans les langages LISP (fonction de définition de nouvelles fonctions DEFUN) ou dans PROLOG (fonctions ASSERT ou RETRACT pour ajouter ou retirer dynamiquement des clauses).

2.5.5. Environnement de travail

L'environnement SMALLTALK est fourni avec divers outils d'aide au développement.

L'écriture d'une application se fait par l'intermédiaire d'un *Class Hierarchy Browser* (figure 7). Il s'agit d'une fenêtre dotée d'une zone éditeur de texte (*TextPane*) couplée au compilateur. Les nouvelles méthodes créées le sont pour la classe courante, sélectionnée dans une liste des classes existantes. Lorsqu'une classe est sélectionnée, une seconde liste propose les méthodes déjà définies pour cette classe. On peut alors aisément ajouter, modifier ou détruire une méthode. Le Class Hierarchy Browser est donc un outil respectant complètement la modularité du langage, et permettant un accès rapide à n'importe quelle méthode.

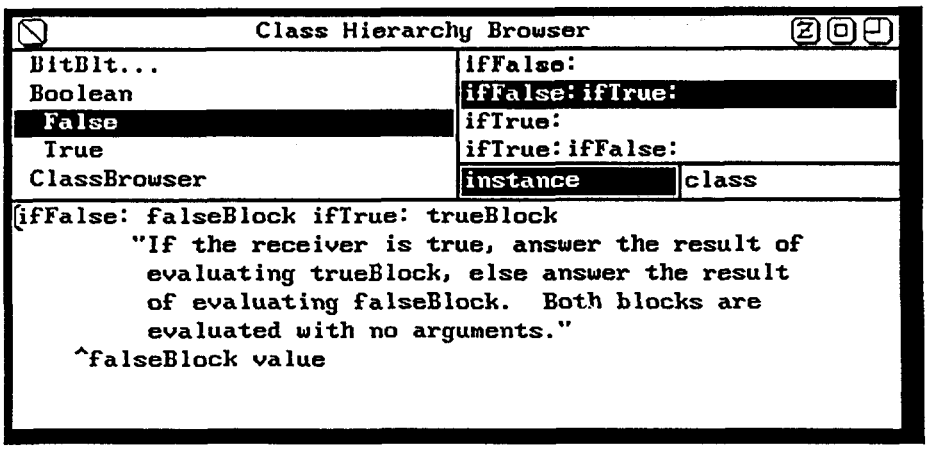


fig.7 La fenêtre Class Hierarchy Browser

Pour la mise au point, un debugger (figure 8) permet une trace complète des derniers messages échangés avant l'arrêt de l'exécution sur erreur ou sur point d'arrêt. Pour les points d'arrêts, l'exécution peut être relancée, en pas à pas ou en continu.

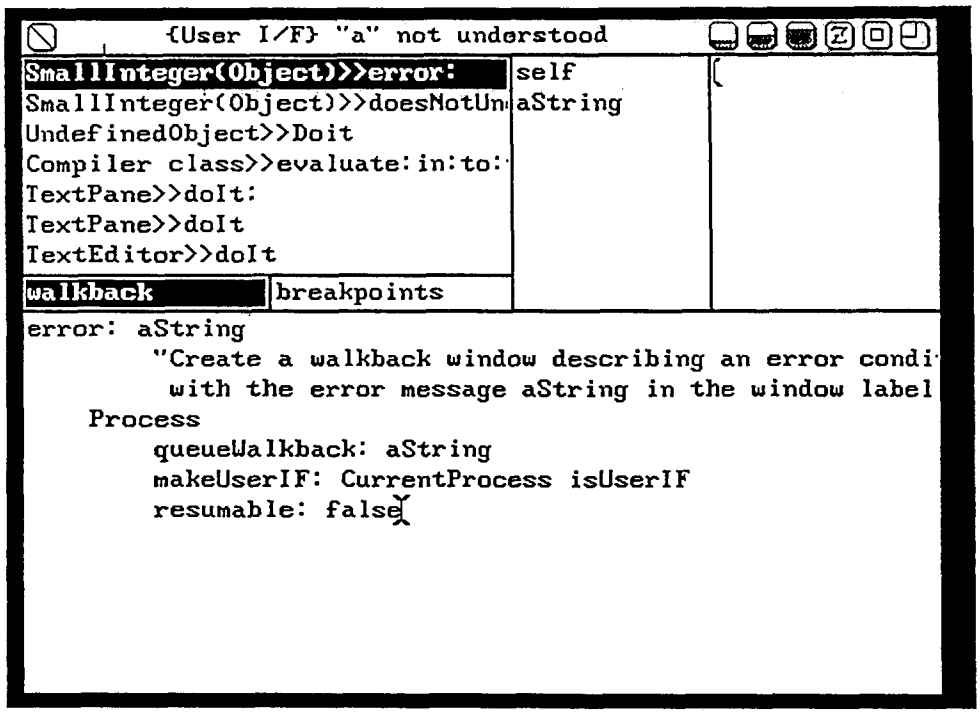


fig.8 La fenêtre de mise au point

Enfin chaque objet du système peut être *inspecté* par une fenêtre qui nous donne l'état de l'ensemble de ses variables d'instances (figure 9). Par simple sélection de la souris, une nouvelle fenêtre d'inspection s'ouvre sur l'objet pointé par la variable d'instance. Ce mécanisme autorise un parcours rapide de structures d'objets complexes.

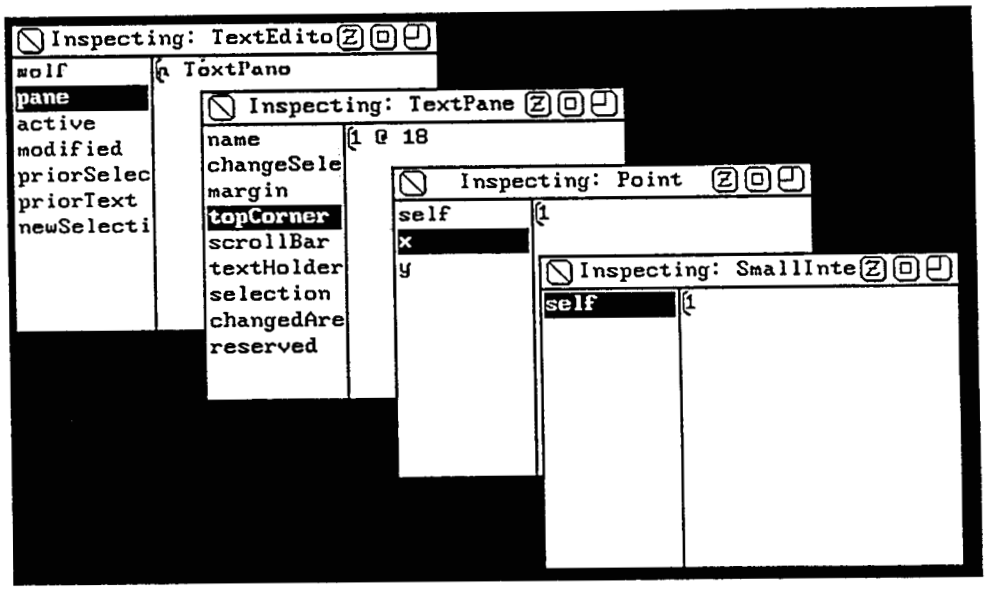


fig.9 Chaîne de fenêtres d'inspection

2.5.6. Graphique

Les possibilités graphiques de SMALLTALK reposent sur la notion de *forme*, image *bitmap* (tableau de bits) stockée en mémoire.

SMALLTALK V/286 autorise en outre l'association d'un tableau de bitmaps à une forme, pour manipuler des images couleurs (4 bitmaps pour 16 couleurs).

La classe *BitBlt* (Bit Block Transfer) permet le transfert rapide de rectangles d'une forme source à une forme destination. Différentes lois de combinaison avec l'image destination sont disponibles, ainsi qu'un mécanisme de masquage avant transfert.

Les primitives graphiques classiques (droites, cercles...) sont implémentées dans la classe *Pen*, sous-classe de BitBlt. Cette classe effectue le transfert d'une forme unique ayant la couleur et la largeur du trait désiré, vers la forme destination.

De même, l'écriture de texte dans une forme se fait par transfert d'une image du caractère contenu dans une forme prédéfinie, grâce à la sous-classe *CharacterScanner* de BitBlt

L'écran physique étant une forme particulière, tout affichage écran se fait par l'intermédiaire d'un BitBlt.
L'affichage à l'écran peut se faire par dessin direct dans la forme-écran, ou par tracé dans une forme intermédiaire qui est ensuite affichée à l'écran. L'image est alors mémorisée et réutilisable.

2.5.7. Multi-fenêtrage

La gestion du multifenêtrage se fait par la collaboration de trois entités: le modèle, la vue et le contrôleur.

La vue et le contrôleur travaillent en étroite collaboration; le contrôleur gère les événements qui se produisent lorsque la fenêtre est active, la vue gère son aspect visuel.

Des couples (vue,contrôleur) standards sont fournis en SMALLTALK. D'autres peuvent être développés pour des besoins spécifiques, mais l'utilisateur est habitué à travailler avec les premiers, il est donc souvent préférable de s'en contenter afin de préserver un aspect uniforme à toutes les applications (réduction des temps d'apprentissage d'une nouvelle application).

Toute fenêtre est constituée d'une fenêtre principale (*TopPane*) qui gère le cadre de la fenêtre (*frame*), ses déplacements, son changement de taille, sa destruction...

A l'intérieur de la fenêtre principale peuvent être placés des sous-fenêtres de type éditeur de texte (*TextPane*), des listes (*ListPane*) ou des graphiques (*GraphPane*).

L'application constitue la troisième composante de la fenêtre : le modèle. Elle devra assurer la création de la fenêtre, le remplissage des sous-fenêtres avec des informations, et la définition des méthodes spécifiques de traitement sur les événements qui peuvent subvenir dans la fenêtre. En particulier, la sélection dans la fenêtre par le bouton droit de la souris est souvent utilisée pour faire apparaître le menu des fonctions que propose l'application.

3. Les principes de base des travaux menés

3.1. Intérêt d'une approche objet de la CAO

La programmation orientée objet est particulièrement privilégiée pour le développement d'outils de CAO en électronique. En effet, elle possède toutes les qualités nécessaires à la création d'un environnement de conception traditionnel, tout en conservant une ouverture vers les évolutions futures de ces outils. De nombreux autres langages orientés objets sont apparus depuis 1970 (C++, Objective C, Eiffel ...), Smalltalk reste toutefois l'environnement objet de référence.

3.1.1. Gestion des données

La représentation objet est une des plus puissante méthode de représentation des informations issues de l'intelligence artificielle. En effet, en plus d'une description statique d'un objet par ses variables, elle apporte une description dynamique par le biais des méthodes et messages.

Dans une moindre mesure, la structuration par classe et l'héritage, enrichissent la description des données, par une structuration arborescente, et par un affinage progressif. Toutefois ces derniers concepts sont encore mal stabilisés dans le monde objet avec de nombreux travaux menés sur l'héritage multiple.

Ces caractéristique justifient alors les nombreux travaux tant sur le plan théorique que sur le plan applicatif, menés autour des systèmes de gestion de Bases de Données. [Gal89] [Rou89] ou encore les extensions objets proposées aux langages LISP [Coi82] ou PROLOG (EMICAT d'ESD).

Dans le domaine de la CAO, Marchesi et al.[Mar88], ont étudié la structuration des données et leur gestion dans un environnement objet (Smalltalk), à partir de la norme EDIF. Ils mettent en évidence la rapidité de développement et l'ouverture de cette approche, qui confèrent une grande modularité et une facilité de maintenance.

Le système FRED développé par Wolf [Wol86], permet la gestion d'une base de données de modules, en offrant à l'utilisateur des possibilités de recherche d'un module par critère, puis de personnalisation de ce module.

Son approche objet permet de caractériser des abstractions de module par des paramètres tels que taille, vitesse, consommation. Il utilise au maximum la notion d'héritage pour classer les modules. Il a utilisé une extension objet de LISP, son langage de description de module (ETHEL) est aussi une sur-couche de LISP.

3.1.2. Graphique et interactivité

Cette caractéristique a été l'un des principes fondamentaux qui a conduit à l'élaboration de Smalltalk. Les résultats ont été spectaculaires puisque la quasi-totalité des interfaces actuellement disponibles s'inspirent directement des travaux du Xerox PARC (Macintosh,X-windows,Presentation Manager).

L'approche objet de l'interactivité aboutit à la définition de fenêtres et de dialogues standards, c'est-à-dire dont la gestion de l'interaction est prédéfinie. Le passage par ces standards n'est pas indispensable, mais la définition de nouvelles fenêtres est longue et nécessite une connaissance approfondie du système.

Le succès du Macintosh est alors simple à expliquer, puisque les modes d'interaction utilisateur/application sont les mêmes, quelle que soit l'application (et donc, quel que soit le développeur).

On réduit ainsi les temps d'apprentissage d'un nouveau logiciel, et les temps d'adaptation lorsqu'on passe d'une application à une autre.

La définition par IBM du Common User Access (CUA) de son architecture SAA, relève du même principe d'ergonomie.

Il faut noter qu'un excès dans l'utilisation de l'interactivité conduit à des outils simples à l'apprentissage, mais d'usage quotidien fastidieux. Il faut prévoir et donner à l'utilisateur différentes méthodes pour réaliser une même fonction.

Certaines méthodes sont très interactives et simples d'emploi, d'autres plus systématiques nécessitent une bonne connaissance de l'application. Ainsi, dans Presentation Manager, toute sélection dans un menu par la souris peut être obtenue par une succession de touches au clavier déroulant les menus équivalents, ou par des touches fonctions d'accélération.

Dans le cadre de la simulation logique, le système INSIST de VanDerMeulen [Van87] [Van89] est un exemple poussé d'utilisation du graphique et de l'interactivité. Dans son approche, tout est traité graphiquement, depuis la saisie de schéma jusqu'à l'affichage des résultats de simulation. Des icônes sont employées à tous les niveaux, telles que l'ampoule électrique blanche (visualisation d'un état haut), l'ampoule noire (état bas) ou l'ampoule brisée (erreur de connexion sur un bus).

Cette approche très visuelle fait d'INSIST un système très attractif, mais éloigné des méthodes de travail habituellement employées en conception. L'interaction profonde entre l'édition graphique et la simulation en fait un système peu modulaire, la visualisation temps réel ralentit les simulations, et la description des séquences d'excitation est limitée à des horloges ou à de simples états logiques. Ce type d'outil peut, par contre, être un excellent support pédagogique pour l'apprentissage de l'électronique digitale.

3.1.3. Evolutivité des applications

Les caractéristiques "génie logiciel" de Smalltalk, en font un environnement approprié au développement de logiciels de grande taille.

La modularité de la programmation, permet une répartition simple du travail sur une équipe de développeurs et facilite la localisation d'erreurs en phase de mise au point et de maintenance.

L'encapsulation des données garantit la sécurité des données. Elle oblige une définition des moyens d'accès aux informations qui assurent une consolidation efficace des différentes parties du logiciel développées séparément.

La programmation par objet amène à définir les données avant les applications. Cette démarche que l'on retrouve dans des méthodes telles que MERISE permet une indépendance entre données et traitements. L'adjonction de nouvelles fonctions à un logiciel est alors possible sans remettre en cause l'existant. On peut remarquer que la méthode MERISE est en cours de modification pour inclure les concepts objets, qui apparaissent également dans les systèmes de gestion de base de données.

Enfin, chaque objet défini est réutilisable dans d'autres applications, l'environnement s'enrichit sans cesse de nouveaux objets, du plus général au plus spécifique.

Ces caractéristiques, s'ils sont présents dans Smalltalk, ne se retrouvent pas forcément dans d'autres langages objet. En particulier, les langages traditionnels étendus à des concepts objets (C++) n'obligent pas une méthode de conception homogène. On risque donc de retomber dans les problèmes liés à ces langages où seule la bonne volonté du programmeur garantit un développement propre des applications.

3.1.4. Ouverture sur des systèmes intelligents

Smalltalk constitue une plate-forme idéale pour l'intégration de systèmes à base de connaissances aux outils de conception traditionnels. Il possède une puissance de représentation des informations autorisant le développement de générateurs de systèmes experts tels que DORIS du CNET, ou même le langage PROLOG qui s'intègre facilement à un environnement objet, en lui donnant les aptitudes au raisonnement logique.

3.1.5. Ouverture sur le parallélisme

L'augmentation des temps de simulation, due à la complexité croissante des circuits, pose aujourd'hui des problèmes de simulation sur architectures parallèles [Smi86]. Les langages objets ou leurs dérivés (langages à acteurs) font l'objet de travaux pour leurs implémentations sur de telles architectures auxquelles ils sont particulièrement adaptés.

3.2. Problèmes de performances et matériel

Le manque d'ouverture sur des systèmes d'exploitation et des langages plus traditionnels constitue le principal inconvénient de Smalltalk. L'outil idéal mélangerait donc à la fois une approche objet pour la représentation des données, une capacité de traitement symbolique des informations, et un environnement de multifenêtrage interactif évolué. De tels langages risquent de se développer dans les années à venir, déjà la société Microsoft annonce un compilateur PASCAL orienté objet, qui sera sans doute compatible avec l'environnement Presentation Manager, lui aussi orienté objet.

Actuellement seules des extensions du langage C, telles que C++, permettent une programmation objet dans un cadre traditionnel. De plus, C++ est interfaçable avec le système de multifenêtrage X-Windows de l'environnement UNIX. L'inconvénient majeur de ce langage est une certaine lourdeur de programmation, la notion d'objet étant vue comme surcouche au langage. Toutefois, la rapidité d'exécution obtenue a conduit au développement d'un programme de transcodage de Smalltalk vers Objective C [Cox87].

Pour cette étude, le langage Smalltalk reste le plus approprié, pour son homogénéité des concepts objets, et pour ses outils d'aide au développement.

Les travaux décrits dans les pages suivantes ont été réalisés sur un PS2/80 d'IBM (microprocesseur 80386 d'Intel à 20 Mhz), doté de 4 Moctets de mémoire centrale, de 110 Moctets de disque dur, et d'un écran graphique haute résolution couleur VGA (640*480). Le coprocesseur arithmétique virgule flottante n'a pas été utilisé, les calculs étant tous réalisés en nombres entiers. La version Smalltalk V/286 de DIGITALK Inc. utilisée est un dérivé de Smalltalk/80, intégrant en particulier une extension de gestion de la couleur.

A titre indicatif, un programme de calcul des 100 premiers nombres entiers, testé en Smalltalk, en C sous OS/2 et en FORTRAN sur un IBM 4341 modèle 2 (puissance annoncée 1,3 Mips), donne une approximation de 1 Mips pour le PS/2 sous OS/2, et de 0,2 Mips pour la 'machine Smalltalk', c'est-à-dire une perte de performance d'un facteur 5 pour supporter l'environnement objet.

Cette perte de performance est due à une architecture peu adaptée aux concepts objets. Ainsi, un processeur microprogrammé orienté objet développé par Nojiri et al. [Noj86] aboutit à un gain d'exécution de programmes objets dans un facteur 10 à 20 par rapport à un 68000. On peut espérer, dans un avenir proche, une optimisation des processeurs classiques pour prendre en compte ces notions objets , qui seront sans doute présentes à tous les niveaux de l'informatique dans les années à venir.

3.3. Application à la simulation logique : objectifs

La convergence entre les besoins en outils de CAO et les caractéristiques des environnements objets tels que Smalltalk, ouvre de nombreuses pistes de réalisation où l'adéquation de l'approche objet pourra être mise plus finement en évidence.

Parmi l'ensemble des outils d'une chaîne de CAO, la simulation prend une part considérable. La simulation électrique relevant davantage d'une approche algorithmique classique et du calcul numérique, c'est l'aspect simulation logique qui a été retenu pour valider la démarche.

Les principes suivants ont guidé la réalisation :

3.3.1. Fonctionnalités complètes

Afin d'aborder tous les aspects de la simulation logique, le simulateur doit intégrer toutes les fonctions des outils actuellement disponibles. On disposera en particulier :

* d'un éditeur graphique

* d'une simulation sur plusieurs niveaux de complexité fonctionnelle, depuis les transistors MOS jusqu'aux mémoires ROM,RAM...

* de langages de netlists, de signaux d'entrée, d'initialisation des circuits de type mémoire.

* d'une simulation guidée par les événements, avec des temps de montée et de descente configurables pour chaque porte

* d'une visualisation graphique (chronogramme) des résultats, avec sortie imprimante.

* d'une prise en compte totale de l'aspect hiérarchique de la conception

Le paragraphe 3.5. reprend les principes de base de la simulation logique et les choix effectués dans cette étude.

3.3.2. Proche de la réalité

Les objets manipulés dans le simulateur doivent être aussi proches que possible des objets du monde réel. Ainsi, l'utilisateur travaille avec le simulateur de façon naturelle. Ceci se traduit par exemple dans les objets sondes, qui, posés sur le circuit, permettent une visualisation après simulation sur des écrans de type analyseur logique (cf chapitre 5).

3.3.3. Utilisation des classes et de l'héritage

Chaque fois que c'était possible, une classe a été définie pour regrouper des méthodes générales autour d'objets réels ou de concepts physiques. Ainsi la classe EtatLogique réalise les fonctions booléennes de base, la classe MotBinaire autorise la création ou l'affichage d'un mot binaire sous forme binaire, hexadécimale, ASCII...(cf chapitre 6).

L'héritage permet de minimiser le nombre de méthodes à écrire pour l'ajout d'une nouvelle classe. Ainsi l'interconnexion des éléments à simuler est implémentée dans la classe CircLog, l'ajout d'une nouvelle fonction combinatoire par exemple, ne nécessitera aucune méthode pour permettre son interconnexion aux autres portes et fonctions prédéfinies (cf chapitre 6).

3.3.4. Enrichissement de l'environnement personnalisé

Le compilateur est accessible à tout objet. On peut donc créer des classes et des méthodes personnalisées à un circuit à simuler.

On obtient ainsi un gain en performance et en occupation mémoire. Ces notions sont exploitées dans les classes génériques et dans la création de modèle (cf chapitres 4 et 6).

3.3.5. Tout est objet, tout objet est accessible

L'utilisateur peut à tout moment accéder aux composants de son circuit. Ainsi, la notion d'inspecteur de Smalltalk est étendue pour donner une vue logique des composants du circuit (cf chapitre 4).

Le simulateur lui-même est un objet particulier, il peut être utilisé par d'autres objets et les résultats de la simulation peuvent être récupérés simplement. Cette possibilité est illustrée par la première méthode de génération de primitives (cf chapitre 7).

3.3.6. Montrer l'ouverture du système

L'utilisation du langage PROLOG, intégré dans l'environnement Smalltalk, est réalisée par le système d'analyse de netlist afin d'illustrer l'ouverture sur des systèmes intelligents, ainsi que sur des systèmes de CAO extérieurs (cf chapitre 4).

3.3.7. Priorité à la vitesse de traitement

Certains choix possibles, entre concept objet pur et vitesse de traitement, ont été résolus en privilégiant les temps de réponse de simulation. Certaines structures standards ont été redéfinies pour favoriser la vitesse d'exécution. Ceci est particulièrement vrai pour les sous-classes de Collection, dont la puissance en fonctionnalité se répercute sur les performances en temps de calcul (cf chapitre 6).

3.3.8. Les composants sont des entités complètes

Autant que possible, les renseignements concernant un composant lui seront directement demandés. Ceci suppose d'attribuer les informations à l'objet dont elles dépendent. On obtient alors un style de programmation par parcours de la structure du circuit, illustré dans le regroupement de signaux binaires en signaux de haut niveau, ou dans les primitives combinatoires de type 2 (cf chapitre 7).

3.4. Simulation des circuits numériques

Nous allons reprendre rapidement les principes de la simulation logique et décrire quelques simulateurs existants, afin de mieux positionner les choix effectués pour cette réalisation.

3.4.1. Caractéristiques des simulateurs logiques

a. nombre d'états logiques représentés

La simulation sur deux niveaux logiques ne permet pas de prendre en compte les phénomènes de transitions d'un état à l'autre, ou les problèmes d'initialisation des circuits à simuler. On introduit en général un état "inconnu", la description du fonctionnement des circuits est alors ternaire, des états inconnus en entrée n'impliquent pas forcément un état inconnu en sortie.

Pour les simulateurs temporels, on peut également trouver des états "front montant" et "front descendant", modélisant plus finement les problèmes temporels qui peuvent apparaître.

Pour la prise en compte des circuits utilisant les possiblités du tristate, un état "haute impédance" est nécessaire. En effet, l'état inconnu est a priori un état 0 ou 1 de puissance normale (issu de la masse ou de l'alimentation), alors que la haute impédance est un état résistif (impédance forte vers la masse ou l'alimentation) qui n'entrera pas en conflit avec des états de puissance normale. Un état intermédiaire d'état logique inconnu et de puissance inconnue pourrait compléter les concepts d'état inconnu et d'état haute impédance. Enfin pour la simulation des transistors MOS, des niveaux logiques 0 et 1 dotés d'attributs de puissance de signal sont nécessaires.

Nous avons retenu un modèle à 6 niveaux, 0 et 1 logiques forts (Zo,Un), 0 et 1 logiques résistifs (Pu,Pd : Pull-Up et Pull-Down), un état inconnu (XX) et un état haute impédance (ZZ) assimilé à un état inconnu résistif. Ce choix est suffisant pour un simulateur plus orienté vers le fonctionnel que vers le temporel.

	puissance forte	puissance faible
état logique 0	Zo	Pd
état logique 1	Un	Pu
état inconnu	XX	ZZ

fig. 10 table de états logiques du simulateur

b. modèle temporel des portes

Le modèle le plus simple consiste à ne considérer aucun retard dans l'évaluation de l'état d'une porte. La simulation est alors simple mais ne permet pas la modélisation des phénomènes séquentiels où le facteur temps joue un rôle essentiel.

L'introduction d'un retard unitaire, identique pour toutes les portes autorise la simulation des circuits séquentiels, mais est totalement transparent aux erreurs temporelles qui peuvent apparaître dans le circuit réel.

L'étape suivante consiste alors à doter chaque porte de son propre temps de propagation. La plupart des aléas de fonctionnement sont alors visibles dans les résultats de simulation.

Deux modèles plus sophistiqués existent: le premier couple un temps minimum et un temps maximum, l'état de la sortie étant inconnu entre ces deux valeurs; le second effectue un calcul de la probabilité d'obtenir un état donné en sortie après un certain temps.

L'inconvénient du système min-max est un résultat de simulation généralement pessimiste pour un temps de calcul plus important, le second ,encore plus complexe, est très coûteux en temps machine.

La solution adoptée est de pouvoir assigner à chaque porte un temps de retard propre, différencié en montée et en descente et relatif à l'entrée qui a provoqué le changement d'état. Ceci réalise un bon compromis entre temps de calcul et finesse des résultats sur le plan temporel.

c. circuits de base pour la simulation

Chaque simulateur dispose d'un certain nombre de circuits de base pour la description du circuit de l'utilisateur. On y trouve ainsi par ordre de fonctionnalité croissante, des portes de logique booléenne, des systèmes combinatoires composés (multiplexeur par exemple), des systèmes séquentiels (bascules, registres), et des composants de haut niveau (RAM,ROM,PLA). La logique en transistors MOS demande un traitement spécifique, plus proche d'une simulation électrique, elle n'est donc pas forcément disponible.

La liste des primitives que nous avons adoptée n'est pas représentative des capacités du simulateur, elle peut être facilement étendue, mais elle possède quelques circuits représentatifs à chacun des niveaux cités précédemment.

d. langage d'entrée

La plupart des simulateurs autorisent 2 méthodes de description de circuit.

La première décrit un simple réseau de primitives interconnectées.

Elle est bien adaptée à la simulation de bas niveau, travaillant sur des portes et fonctions élémentaires.

La simulation multi-niveaux requiert la possibilité de décrire un circuit en terme fonctionnel. A chaque évènement en entrée, correspond une réaction du circuit calculée sur les valeurs d'entrée et ses états internes.

Un circuit ainsi défini peut alors s'interconnecter à d'autres circuits pour une simulation globale. Cette interconnexion est bien sur décrite par le même langage que précédemment.

Le choix minimum d'un langage d'interconnexion a été retenu, en définissant une syntaxe simple, et proche de la structure objet implantée pour la simulation.

Nous verrons au Chapitre 6 paragraphe 3, que la définition d'un langage comportemental devient inutile, dès lors que la structure du simulateur est ouverte, et que des classes et méthodes adéquates ont été définies.

3.4.2. Comparaison avec des simulateurs du marché

Les caractéristiques de 3 simulateurs utilisés en conception de circuits VLSI sont données à titre de comparaison. On trouvera dans [Har89] une étude comparative sur l'ensemble des outils actuellement disponibles sur le marché.

Le simulateur DLS II de DAISY travaille sur 9 niveaux logiques, 6 paramètres temporels par porte (nominal minimum et maximum en montée ou en descente) avec des vérifications possibles de contraintes de durée minimale d'impulsion, de relations entre signaux et de temps de stabilité de signaux avant ou après événement (temps de SETUP et de HOLD).

Les primitives du système incluent les portes logiques de base, les bascules, les RAM, les ROM, les PLA, et des opérateurs arithmétiques. Ce simulateur tourne sur des stations DAISY dédiées ou sur des stations de travail SUN.

Le simulateur EPILOG2 de THOMSON travaille sur cinq états logiques, avec des temps de propagation unitaires. Par contre, des primitives purement temporelles peuvent être introduites sous forme de retards purs ou de formeurs, fixes ou ajustés dynamiquement en cours de simulation. Les primitives disponibles sont les fonctions booléennes, des bistables, les ROM, les matrices, les sommes de produits, les produis de somme, la bascule D. le point mémoire bidirectionnel,etc... Ce simulateur tourne sur des systèmes IBM ou DIGITAL.

Enfin, le simulateur HILO de GENRAD travaille sur 5 niveaux logiques et prend en compte jusqu'à 6 temps de transit à travers les portes, variables avec la charge connectée en sortie. Les fonctions de base sont le AND, le OR, le NAND, le NOR, les noeuds (WIRE) réalisant des fonctions câblées, les registres, les RAM et les ROM. Ce simulateur tourne sur différents types de systèmes, depuis la station de travail jusqu'au mainframe IBM ou DIGITAL.

4. Description du circuit à simuler

Ce chapitre est consacré à la description du circuit par l'utilisateur, centrée sur la notion de "netlist". Les concepts de primitives et de modèles sont introduits pour la hiérarchisation de cette description. Le langage de description "netlist" propre au simulateur est ensuite décrit, ainsi que la structure de donnée chargée en mémoire pour la simulation, ces deux notions étant étroitement liées.

Ceci nous conduit alors à montrer l'intérêt de l'utilisation de PROLOG pour l'analyse de la netlist en vue de créer la structure de simulation.

Enfin, l'interface proposée à l'utilisateur pour entrer sa description de circuit est détaillée. L'aspect édition de texte de netlist profite pleinement de la ré-utilisabilité des classes et méthodes prédéfinies dans le système. L'édition graphique constitue une bonne illustration des possibilités graphiques de Smalltalk, et l'héritage permet encore une fois la personnalisation de systèmes MVC (Modèle Vue Contrôleur) prédéfinis.

4.1. Hiérarchie de description des circuits

Pour réaliser une simulation, l'utilisateur doit créer en mémoire un objet appelé circuit, entité qui regroupe des objets appelés composants. Les composants et leurs interconnexions sont définis dans des fichiers de description de circuits logiques (netlist). Ces fichiers peuvent être créés par un éditeur de texte classique. Pour respecter la démarche naturelle de conception hiérarchique, le composant pourra être une primitive de simulation ou un modèle.

4.1.1. Primitive

Une primitive de simulation est un composant dont le comportement est codé directement sous forme de méthodes Smalltalk. Le calcul de ses sorties se fait donc en utilisant pleinement les capacités de calcul de l'ordinateur, circuit universel. On peut dire à ce niveau que Smalltalk lui-même est le langage de description comportemental du circuit.

La description de nouvelles primitives se réduit à l'écriture de quelques méthodes décrivant le comportement du circuit, toutes les fonctions "utilitaires" peuvent être héritées d'une des classes existantes, ou inspirées des méthodes déjà présentes. On trouvera en annexe un exemple de description de primitive et les contraintes à respecter pour y introduire des nouvelles.

Certaines de ces primitives sont paramétrables, en particulier pour gérer les primitives à nombre d'entrées ou de sorties variables (portes ET à deux, trois, ... N entrées). La paramétrisation conduit à définir certaines méthodes du comportement uniquement sur demande. On génère alors des classes et des méthodes prenant en compte la dimension de la primitive. Les classes permettant cette génération automatique seront appelées classes génériques. Elles regroupent tous les comportements indépendants de la dimension du composant. On obtient ainsi un gain de place en mémoire, permettant d'augmenter la taille des problèmes soumis au simulateur (cf chapitre 6).

4.1.2. Modèle

Un modèle est une classe capable de générer sur demande un réseau spécifique de composants. Il s'agit donc d'une agrégation d'objets pouvant être eux-même des primitives ou des modèles.

Cette agrégation encapsule les composants du modèle pour toutes les opérations de création et de gestion du réseau de portes logiques, elle disparait au niveau simulation.

Un modèle est décrit par les mêmes fichiers de description netlist que les circuits. Il peut donc être simulé et validé en tant que circuit avant sa mise en oeuvre dans un circuit de niveau supérieur.

Le calcul des sorties en cours de simulation se fait en propageant les événements d'entrée à l'intérieur du réseau de ses composants. Son comportement est donc simulé et ne conduit pas à un gain de temps d'exécution.

Lorsque le modèle a été défini (création d'une classe), on peut reproduire autant d'exemplaires du modèle que nécessaires dans un circuit de plus haut niveau de complexité. Ceci amène des descriptions de circuits beaucoup plus simples, et qui respectent la modularité de la conception. De plus, les modèles ainsi définis sont ré-utilisables et permettent la création d'une bibliothèque de circuits standards.

Ainsi, une bascule D synchronisée sur état est décrite à partir de 6 portes ET et 5 portes NON (figure 11.a). La version maître-esclave de cette même bascule est constituée de deux bascules D sur état et d'une porte NON (3 lignes dans la netlist) (figure 11.b). Enfin un compteur par 16 peut-être bâti avec quatre de ces bascules D maître-esclave (4 lignes dans la netlist) (figure 11.c). Le chargement du compteur, en tant que circuit pour la simulation, générera automatiquement les 48 portes ET et 44 portes NON correspondantes.

Le paragraphe 4.5. reprend la structure interne des modèles et démontre la capacité de Smalltalk à prendre en compte une description hiérarchique des circuits.

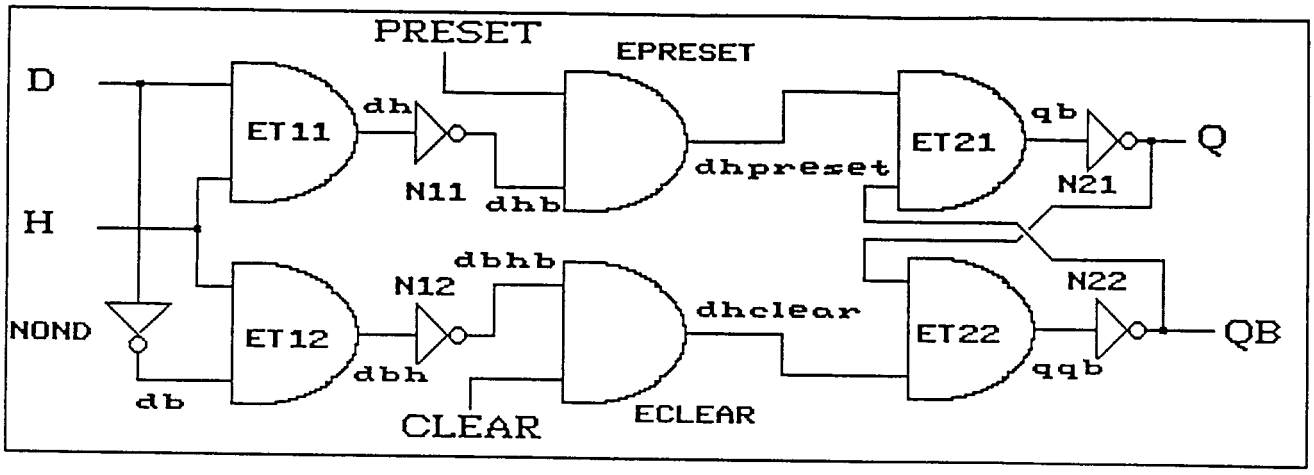


fig.11.a bascule D synchronisée sur état

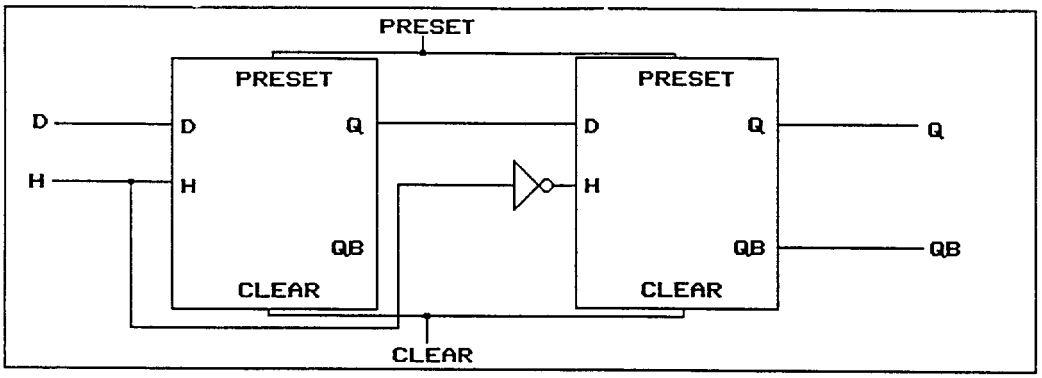


fig.11.b bascule D maitre-esclave

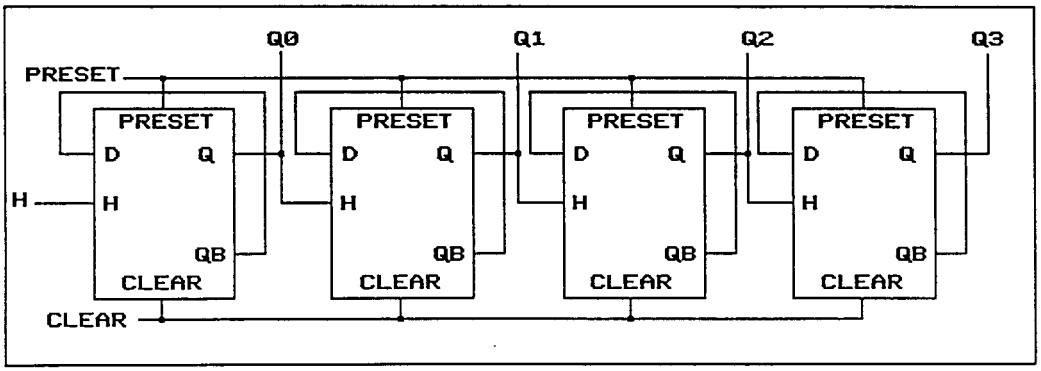


fig.11.c compteur par 16

4.2. Le langage d'entrée du simulateur

Chaque circuit est décrit par une ligne titre et par des lignes composants. Des lignes vides ou des commentaires (ligne débutant par un caractère %) peuvent être ajoutées au gré de l'utilisateur pour augmenter la lisibilité de la description.

4.2.1. La ligne titre

Le titre contient le nom du circuit. Ce nom servira d'identifiant au circuit créé, aux fichiers de séquences ou d'initialisation de la simulation. Ce sera également le nom du modèle créé si la netlist est utilisée comme description de modèle.

Les listes des noeuds d'entrée et de sortie du circuit complètent la carte titre. Elles permettent de vérifier que tous les noeuds du circuit sont connectés. Pour la simulation, les entrées pourront recevoir des séquences d'excitation, les sorties et les entrées seront d'office sous surveillance de sondes pour observer les résultats de la simulation.

4.2.2. Les lignes composant

Elles sont constituées des éléments suivants :

* un nom
* une liste de noeuds d'entrée
* une liste de noeuds de sortie
* un type de composant
* une liste de paramètres

Le nom du composant n'est utilisé que pour l'identification par l'utilisateur.

Le nombre d'entrées et de sorties fait l'objet d'une vérification quantitative en phase de génération. Une sortie non connectée sera représentée par un pseudo-noeud de nom '_'. Une compression de l'écriture, pour les séries de noeuds du type $nom_i,...,nom_j$, est autorisée sous la forme nomi@j. Ceci fera l'objet d'un prétraitement avant la phase d'analyse syntaxique. Chaque noeud a donc une existence propre, il ne s'agit pas de notion de bus, ce qui permet une extraction de noeuds simples ou de sous-ensembles.

Le type du composant est en principe le nom de la classe SMALLTALK dans laquelle sera créé le composant. Si cette classe n'existe pas, on essaiera :

* soit de charger un modèle de même nom (fichier nom.top)

* soit de faire générer cette classe par la classe générique si c'est une classe paramétrable (nom suivi de paramètres /x)

Ce mécanisme est totalement transparent à l'utilisateur qui n'a donc pas à se préoccuper de l'existence préalable de ces classes. Toutefois, ceci amène une augmentation du temps de chargement de la netlist en cours de traitement, il faut donc en avertir l'utilisateur.

La liste des paramètres correspond en général à des temps de transit facultatifs. S'ils sont absents, des valeurs sont prises par défaut. Cette liste peut également comporter tout autre type d'indications pour l'initialisation du composant. Ainsi, les composants mémoire demandent pour leur initialisation une indication sur le fichier définissant leur contenu au début de la simulation.

Enfin, la description de circuits à base de commutateurs (transistors mos) et le traitement particulier qu'ils subissent (voir paragraphe 6.4.6) amènent à définir :

* des noeuds particuliers, de nom zéro et un, représentant des noeuds à états logiques constants Zo et Un (masse et alimentation).

* des composants n'ayant pas d'existence propre dans le simulateur, de nom pullup et pulldown, indiquant une connexion résistive aux états logiques haut et bas. On se contente de donner alors la liste des noeuds connectés à un tel composant. Il n'y a donc qu'une seule liste de noeuds, pas de type ni de liste de paramètres.

exemple 1 :

```
Basd(h,d,preset,clear-q,qb)
%* bascule D sur état avec clear et preset
nond(d-db):Not()
et11(h,d-dh):And/2(3,5)
et12(h,db-dbh):And/2(3)
n11(dh-dhb):Not(1)
n12(dbh-dbhb):Not(1)
eclear(dbhb,clear-dhclear):And/2(3)
epreset(dhb,preset-dhpreset):And/2(3)
et21(dhpreset,qb-qq):And/2(3)
et22(dhclear,q-qqb):And/2(3)
n21(qq-q):Not(1)
n22(qqb-qb):Not(1)
```

L'exemple 1 correspond au circuit dont le schéma est fourni au paragraphe 4.1.2 (figure 11.a). On remarque que le composant nond est de type Not, ce qui correspond à une primitive. La liste des paramètres est vide, les temps de montée et de descente seront donc initialisés à une valeur par défaut.

Pour le composant et11, le type est paramètré par le '/2', qui indique une porte ET à deux entrées. Si la classe correspondante n'existe pas encore, elle sera générée par la classe And. La liste des paramètres contient le temps de montée et le temps de descente du signal de sortie. Pour le composant et12, la liste des paramètres ne contient qu'une valeur qui sera affectée à la fois au temps de montée et au temps de descente.

exemple 2 :

```
DemiAdd(x,y,c-s,c2)
nx(x-xb):Not(1)
ny(y-yb):Not(1)
nc(c-cb):Not(1)
pullup(c2)
T1(xb-i,c2):SwitchN(1)
T2(yb-i,zéro):SwitchN(1)
T3(xb-c,aaa):SwitchN(1)
T4(yb-c,bbb):SwitchN(1)
T5(x-j,s):SwitchN(1)
T6(y-j,c):SwitchN(1)
T7(xb-k,s):SwitchN(1)
T8(yb-k,c):SwitchN(1)
T9(x-l,s):SwitchN(1)
T10(yb-l,cb):SwitchN(1)
T11(xb-m,s):SwitchN(1)
T12(y-m,cb):SwitchN(1)
T13(y-aaa,c2):SwitchN(1)
T14(x-bbb,c2):SwitchN(1)
```

L'exemple 2 correspond à la figure 12. Il s'agit d'un demi-additionneur réalisé en transistors NMOS. on remarque l'utilisation du pseudo composant pull-up et du noeud réservé zéro.

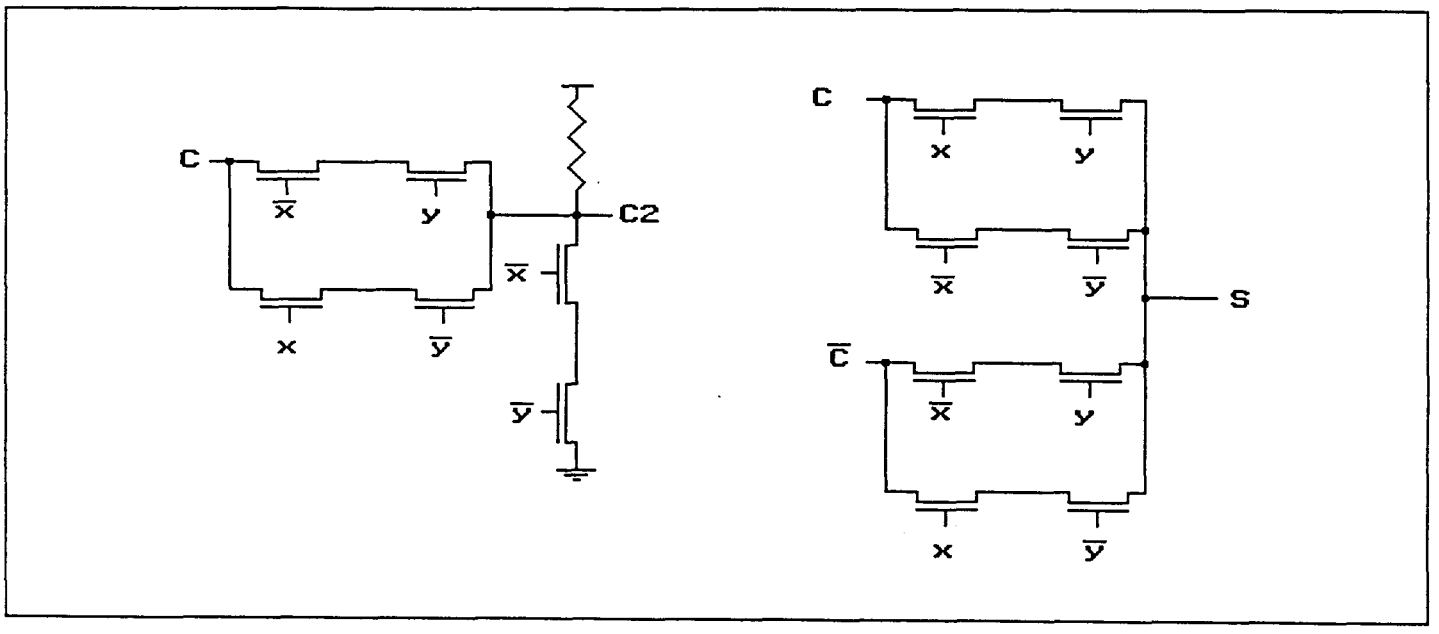


fig.12 demi-additioneur

exemple 3 :

```
Alu16(a0@15,b0@15,cin,m,s0@3-f0@15,cout,eq,g,p)
% Alu 2*16 bits constituée a partir de l'alu74181
alu1(a0@3,b0@3,cin,m,s0@3-f0@3,cout1,_,_,_):Alu74181()
alu2(a4@7,b4@7,cout1,m,s0@3-f4@7,cout2,_,_,_):Alu74181()
alu3(a8@11,b8@11,cout2,m,s0@3-f8@11,cout3,_,_,_):Alu74181()
alu4(a12@15,b12@15,cout3,m,s0@3-f12@15,cout,g,p,eq):Alu74181()
```

Dans ce dernier exemple, représenté à la figure 13, on trouve l'illustration de la compression de liste de noeuds : s0@3 correspond à s0,s1,s2,s3. D'autre part, les sorties de calcul anticipé de retenue des trois premiers 74181 ne sont pas connectées et sont donc remplacées par le caractère '_'. Alu74181 n'est pas une primitive du simulateur, mais si l'on détecte l'absence d'une classe portant ce nom, on chargera automatiquement le modèle correspondant contenu dans le fichier Alu74181.top.

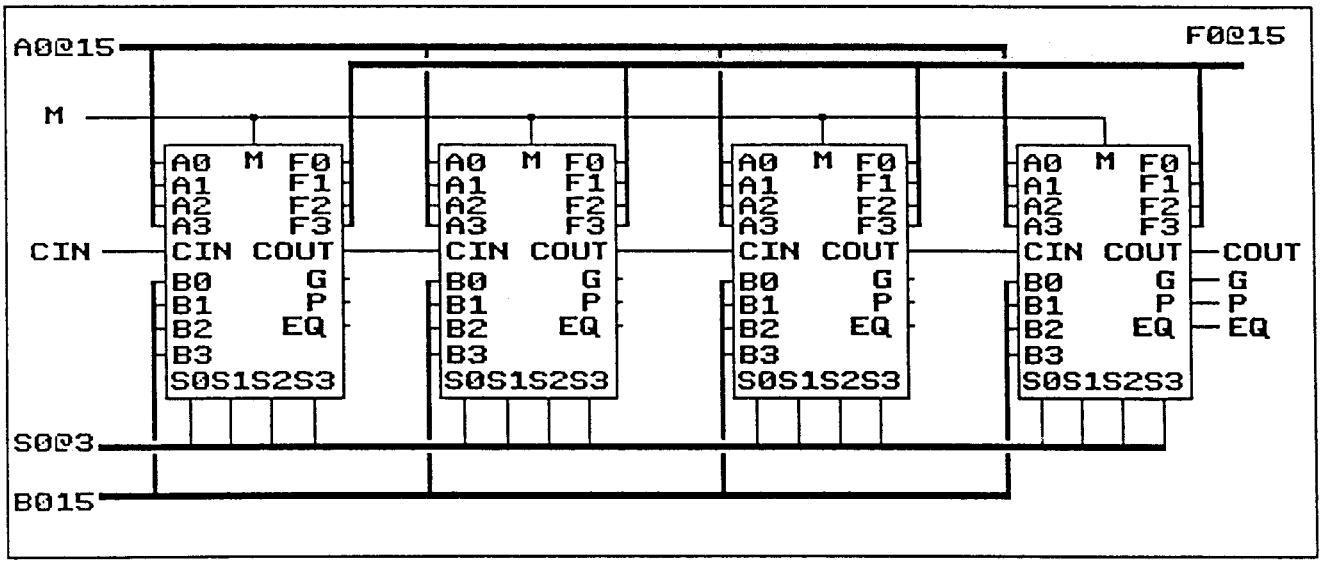


fig.13 Unité Arithmétique et Logique

4.3. Structure d'un circuit chargé en mémoire

Le but de l'analyse est de construire un objet de classe Circuit qui sera le point central de la simulation. Un circuit est décrit par :

* un nom
  le nom sert d'identifiant au circuit, il est mémorisé dans le dictionnaire système afin de pouvoir accéder au circuit

* une liste des entrées
  cette liste fait le lien entre le nom de l'entrée du circuit et l'objet plot d'entrée du circuit. Un plot est une primitive qui permet de distribuer un signal unique sur un ensemble de connexions. Il n'a aucun effet temporel, et il est créé automatiquement avec le circuit. C'est lui qui recevra les excitations de la simulation.

* une liste des sorties
  cette liste permet de mémoriser les noms des noeuds de sortie du circuit, afin de les remémorer à l'utilisateur en cours de simulation. Au début d'une simulation, toutes les entrées et les sorties du circuit sont placées sous surveillance de sondes de visualisation.

* une liste de noeuds
  cette liste permet le placement de sondes sur les noeuds internes du circuit, en faisant le lien entre le nom du noeud et l'élément de connexion correspondant (composant et nom de la sortie).

* une liste de composants
  pour chaque composant, on mémorise le nom,les noms des noeuds d'entrée, de sortie, le type, les paramètres et l'objet correspondant.

* l'objet simulation
  la définition des sondes, l'envoi de séquences d'excitation, la visualisation des résultats se font par l'intermédiaire d'un objet de type simulation associé au circuit.

* le répertoire des fichiers de netlist
  la structure arborescente de gestion de fichiers utilisée par Smalltalk V286 est identique à celle du MS-DOS. Elle permet une gestion efficace de la bibliothèque de circuits. Chaque circuit est associé à plusieurs fichiers, portant tous le même nom que le circuit et une extension caractéristique du contenu (.ram pour le fichier d'initialisation des mémoires vives par exemple). Tous les fichiers se rapportant au circuit doivent se trouver dans le même répertoire que le fichier de netlist.

Le circuit contient donc toute la sémantique présente dans la description netlist. Ceci est important pour que l'utilisateur puisse travailler avec les noms symboliques qu'il a choisis tout au long de la simulation. Une primitive ne connait que l'objet circuit auquel elle appartient. Son propre nom et le nom des noeuds auxquels elle est connectée, elle doit les demander à son circuit.

Par contre, chaque primitive connait le nom de ses noeuds d'entrée et de sortie, en tant que nom au niveau de sa classe (de son type). Ainsi une bascule D peut avoir une entrée d'horloge de nom 'h'. Pour l'utilisateur, cette entrée peut être reliée au noeud 'clock_phi1' du circuit. C'est ce nom qu'on lui présentera durant la simulation et qui est stocké au niveau de l'objet circuit.

Ce principe, que l'on retrouvera dans la structure des modèles, fait jouer le polymorphisme de Smalltalk. Un composant peut appartenir à un Circuit ou à un Modèle, il sera toujours capable de donner son nom par une méthode unique. De même, un Modèle est considéré comme composant du Circuit, il peut donc fournir son nom en tant que composant du circuit. La figure 14 résume les chemins d'évaluation pour un message initial demandant le nom.

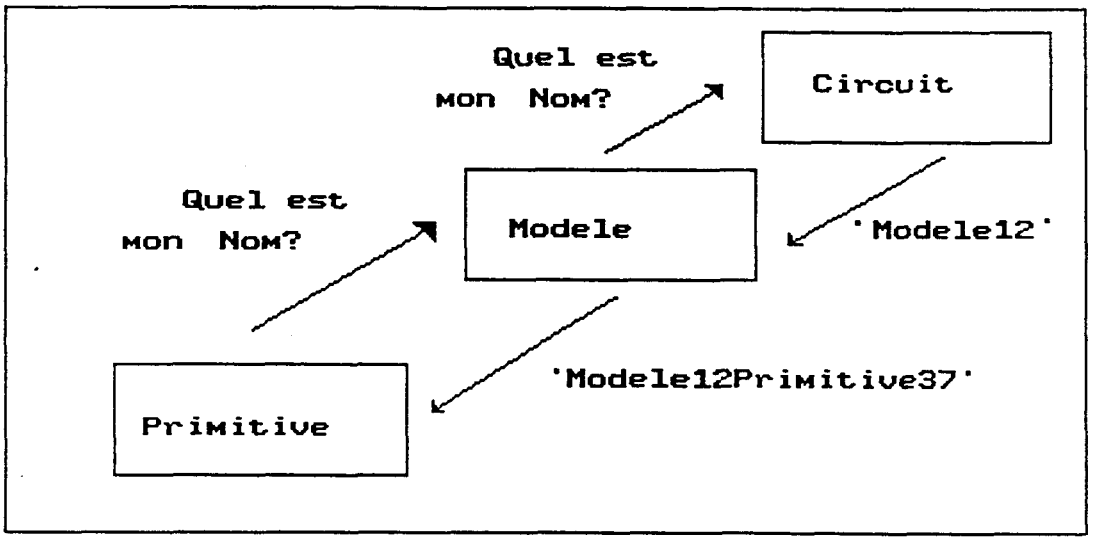


fig.14 Evaluation d'une demande de nom

4.4. Les phases de création d'un circuit

La construction de la structure d'objets représentant le circuit se fait en quatre étapes :

* prétraitement lexicographique
  c'est dans cette phase que l'expansion des groupes de noeuds nomi@j et le découpage en éléments syntaxiques sont réalisés.

* analyse syntaxique
  durant cette phase, la liste des composants de l'objet Circuit est construite mais les objets composants ne sont pas encore créés.

* analyse sémantique
  durant cette phase, la connexion de tous les noeuds définis est vérifiée. Un noeud non connecté sur une entrée et qui n'est pas une sortie du circuit est signalé à l'utilisateur. Un noeud d'entrée non connecté sur une sortie ni relié à une entrée du circuit provoque un arrêt du chargement du circuit avec un message d'erreur.

C'est également pendant cette phase que les réseaux de commutateurs (transistors MOS) sont prétraités, l'objet de simulation n'étant pas le commutateur mais des branches complètes du réseau. Enfin on vérifie l'existence de tous les types de composants utilisés. Si un type demandé ne correspond pas à une classe existante, le chargement du modèle correspondant est lancé ou la génération de la classe, s'il s'agit d'une classe générique.

* génération du circuit
  l'ensemble des objets-composants peuvent alors être créés. C'est l'occasion de vérifier la validité du nombre d'entrée, de sorties, et la validité de la liste des paramètres. Chaque classe (type de composant) possède ses propres méthodes d'initialisation.

4.5. Structure et création d'un modèle

Le chargement d'un modèle doit aboutir à la création d'une classe capable de générer sur demande les composants et les interconnexions des composants du modèle.

Pour rendre totalement transparent l'emploi de modèles plutôt que de primitives, l'objet créé à partir de cette classe doit être capable de répondre aux mêmes messages qu'une primitive normale (polymorphisme).

En particulier, tout message de connexion à l'intérieur du circuit est envoyé aussi bien aux composants primitifs qu'aux composants modèles. L'objet modèle doit rediriger ces messages vers ses propres composants concernés par la connexion, en entrée ou en sortie. La figure 15 illustre une demande de connexion à une entrée x d'un modèle. Ce modèle contient un inverseur et une porte Et, tous deux reliés à cette entrée.

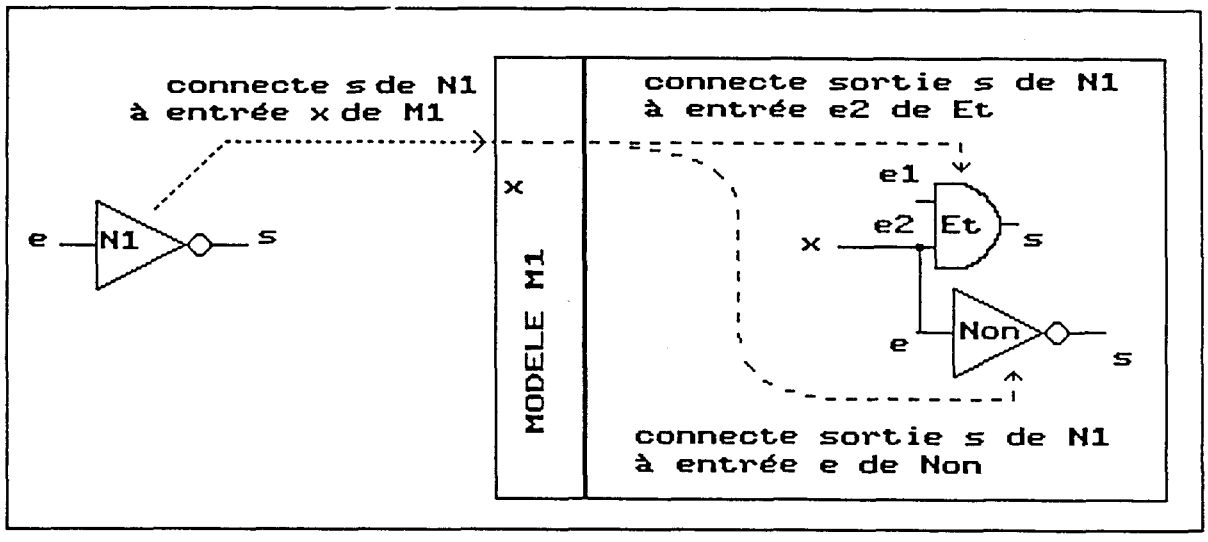


fig.15 Evaluation d'une demande de connexion par un modèle

On s'arrange toutefois pour que les messages échangés pendant la phase de simulation ne passent plus par l'intermédiaire du modèle, ce qui augmenterait inutilement le temps de simulation. Ceci est réalisé par la connexion directe de toutes les primitives entrant en jeu dans le circuit, qu'elles soient composantes du circuit lui-même ou composantes de modèles et sous-modèles du circuit.

Le modèle joue donc un rôle statique, en ce sens qu'il n'intervient qu'en dehors des phases de simulation. Il mémorise en particulier une structure analogue à la liste des composants d'un circuit.

Il fournit ainsi le nom de ses composants ou de ses noeuds internes par concaténation de son propre nom au niveau du circuit, et du nom du composant ou du noeud au niveau de la netlist du modèle. Ainsi dans l'exemple précédent l'inverseur n1 du modèle m8 du circuit se nommera 'm8-n1'

La méthode principale de chaque modèle est le new, c'est-à-dire la création d'un nouvel objet de cette classe. Le new reprend l'ensemble des messages de création de composants et d'interconnexions qui sont réalisés dans la phase de génération de circuit.

Les trois premières phases de traitement du fichier de description topologique (lexicographique, syntaxique, sémantique), sont identiques dans les deux cas de chargement de circuit ou de modèle. Seule la phase de génération est différente puisque qu'elle consiste en la création de la classe et en la création et la compilation des méthodes associées.

La classe générateurDeCircuit est responsable de la création de la structure de l'objet circuit en mémoire, la classe générateurDeModèle permet la création des classes modèles. Elles sont toutes deux classes filles de la classe AnalyseurDeTopologie, et héritent donc des méthodes d'analyses lexicographique, syntaxique et sémantique.

4.6. Utilisation du langage PROLOG

Les caractéristiques de la programmation orientée objet sont très complémentaires à celles du langage PROLOG. En effet, si PROLOG permet la description d'un problème à résoudre et fournit un moteur de résolution, Smalltalk offre une structuration puissante des connaissances et une simplicité de définition d'interfaces utilisateurs avancées, ou plus généralement de gestion des entrées-sorties.

Tous les aspects relevant de l'algorithmique plus classique ou du traitement des informations de types traditionnels (entiers, flottants, chaînes de caractères...) sont également à la charge de la programmation orientée objet.

Dans cette perspective, Smalltalk V286 est fourni avec un PROLOG parfaitement intégré à l'environnement objet. Un éditeur LogicBrowser permet l'écriture et la compilation en code Smalltalk des prédicats PROLOG. Une "classe logique" contient un ensemble de prédicats traitant d'un problème donné et un objet de cette classe peut recevoir des messages d'interrogation, qui retournent l'ensemble des solutions trouvées en cours de résolution. Inversement, le prédicat "is(x,y)" évalue y en tant qu'expression Smalltalk et unifie le résultat avec x.

Le langage PROLOG se prête bien au traitement de langages; historiquement, il a d'ailleurs été conçu pour résoudre des problèmes de langage naturel. Son utilisation pour l'analyse syntaxique des netlists constitue donc une solution élégante, ramenant l'écriture de l'analyseur à une simple description de la syntaxe sous forme de règles. Ceci permet de plus une détection et une signalisation beaucoup plus claire des erreurs de syntaxe.

exemples :

```
tiret(['-'|l],l,_):-!.
tiret(l,_,true):-erreur(l,' tiret (-) attendu').
```

Ce prédicat vérifie que le premier élément de la liste est un tiret et retourne la suite de la liste. Si le caractère n'est pas un tiret et que le flag de présence obligatoire est à true (troisième paramètre), on signale l'erreur en montrant la ligne et la position dans la ligne où s'est produite l'erreur.

```
circuit (nomCircuit,entrées,sorties,liste1):-
   nom(nomCircuit,liste1,liste2,true),
   parentheseOuvrante(liste2,liste3,true),
   initialiseOrderedCollection(entrées),
   listeDeNoms(entrées,liste3,liste4,true),
   tiret(liste4,liste5,true),
   initialiseOrderedCollection(sorties),
   listeDeNoms(sorties,liste5,liste6,true),
   parentheseFermante(liste6,liste7,true),
   ligneVide(liste7,true).
```

Ce prédicat définit la syntaxe de la carte titre d'un circuit. On retrouve tous les éléments syntaxiques présents dans cette carte. Le nom du circuit et deux listes (OrderedCollection) des noeuds d'entrée et de sortie sont retournés à la méthode Smalltalk appelante.

Cette méthode a pour inconvénient un temps d'exécution relativement long. PROLOG est en effet ici une surcouche de Smalltalk, l'analyse d'un fichier est alors relativement longue (environ 1 s par ligne traitée). Une version de PROLOG plus profondément intégrée à l'environnement objet permettrait de retrouver des temps comparables à la méthode d'analyse syntaxique programmée en Smalltalk.

D'autre part, la résolution est très gourmande en occupation de la pile système, au point que l'exécution doit être déroulée dans un processus détaché, ayant sa propre pile d'appel de méthode et s'exécutant complétement avant la reprise de la méthode appelante (priorité plus forte).

4.7. Ouverture sur d'autres systèmes CAO

Une des caractéristiques attendue d'un système CAO serait sa capacité à échanger des informations avec d'autres bases de données CAO. Dans le domaine de la simulation des circuits logiques, le problème provient de la diversité des solutions adoptées pour la prise en compte des aspects temporels, des niveaux logiques, ainsi que pour les primitives intégrées au système.

Afin d'illustrer les possibilités d'ouverture offertes par la programmation objet, une interface a été réalisée pour permettre la récupération des circuits standards présents dans les bibliothèques du simulateur HILO. Cet exemple a été choisi en raison du grand éloignement existant entre les deux simulateurs.

Cet éloignement nous impose un certain nombre de limitations quant aux fichiers importables :

* composants associés à des primitives communes

* réduction des paramètres temporels à un temps de montée et un temps de descente (choix multiple possible de la réduction : valeur min, valeur max, moyenne...)

* pas d'utilisation du langage fonctionnel

La commande EXPAND de HILO permet de générer un fichier d'exportation de la netlist d'un circuit, sous une forme propre, c'est-à-dire avec une syntaxe plus rigoureuse que celle autorisée par l'analyseur de netlist HILO. C'est donc ce type de fichier que l'on utilise pour importer les descriptions de circuits.

Afin de permettre une adaptation ou une extension de ces circuits, l'importation ne conduit pas directement à un circuit chargé en mémoire mais fournit un fichier netlist selon la syntaxe propre au simulateur. Il s'agit donc d'un mécanisme de traduction.

L'exemple de la traduction d'un fichier HILO décrivant un circuit de type 74169, compteur-décompteur décimal préchargeable, est fourni en annexe.

4.8. L'interface utilisateur

La gestion de la bibliothèque de circuits, et l'éditeur de texte, permettant la création ou la modification des fichiers netlist, ont été réalisés dans une sous-classe du gestionnaire de disque Smalltalk.

L'héritage est ici utilisé pleinement puisque sur 44 méthodes, 8 sont redéfinies et 10 nouvelles méthodes sont créées.

La fenêtre de dialogue, dont un exemple apparait sur la figure 16, est principalement composée de trois parties.

* la liste des répertoires
  en haut à gauche de la fenêtre, elle permet de sélectionner le répertoire courant. On peut compresser cette liste à un niveau de détail de l'arborescence quelconque. Ceci permet un parcours rapide de l'ensemble des fichiers contenus sur le disque, quel que soit la complexité de l'arborescence des répertoires. On accède par menu aux fonctions de création-suppression de répertoire.

* la liste des fichiers du répertoire courant

en haut à droite de la fenêtre. Pour le simulateur, cette liste est filtrée pour ne laisser apparaître que les fichiers de type adéquat:

- .top pour les netlist
- .seq pour les séquences de simulation
- .ram,.rom ... pour les initialisations
- .gpe,.gps pour les regroupements de noeuds

Le menu permet de sélectionner un de ces filtres, ou de réaliser des opérations traditionnelles sur fichiers (création, suppression, copie ...)

* un éditeur de texte

en bas de la fenêtre, il permet toutes les manipulations du texte, en utilisant au maximum les possibilités de la souris (sélection de texte, couper-coller, déplacement dans le texte...). Cette fonction d'édition est, bien sûr, prédéfinie dans Smalltalk et ne nécessite aucune adaptation quant à son utilisation dans ce contexte. Le menu a été enrichi de fonctions permettant de lancer l'analyse du texte en vue de la création d'un circuit, d'un modèle ou d'une primitive.

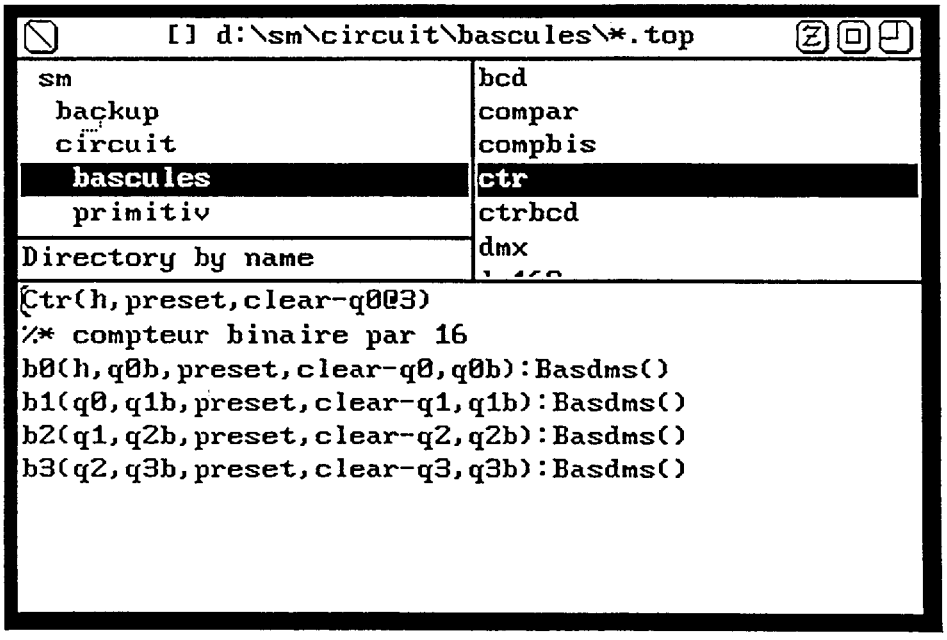


fig.16 Fenêtre de gestion de la bibliothèque de circuits

4.9. L'interface graphique

La conception Smalltalk du graphisme est surtout orientée vers la présentation d'informations à l'utilisateur. Il est intéressant d'explorer les possibilités du graphisme interactif demandé par un éditeur graphique de saisie de schéma, dans un environnement MVC.

Afin de respecter l'aspect hiérarchique du simulateur, cet éditeur fournit deux outils à l'utilisateur. Tout d'abord un éditeur de netlist qui permet la saisie du schéma d'un circuit, puis un éditeur de forme qui permet de donner une forme graphique aux modèles intégrés comme composants d'un circuit.

4.9.1. L'éditeur de netlist

Deux types d'objets sont manipulés pour la saisie d'un schéma : les composants et la connectique.

a. les composants

Ces composants sont des primitives ou des modèles. Leur forme graphique est constituée de deux rectangles :

* le boitier
* l'encombrement total boitier + broches

Cette structure unique, indépendante du type du composant, permet de faire jouer l'héritage pour de nombreuses fonctions. En particulier, le dessin est réalisé par une méthode commune, à partir d'une structure définissant les caractéristiques du boitier et de son brochage. Toutefois, certains boitiers redéfinissent cette méthode afin de gagner en lisibilité du schéma. Le dessin est alors programmé dans la méthode.

C'est le cas, en particulier, des fonction booléennes qui sont représentées par le symbolisme classique.

L'encombrement total du composant permet l'identification en cas de sélection à la souris par l'utilisateur. Si la sélection vise une broche du composant, on reconnait d'abord le composant puis on identifie la broche à l'intérieur du rectangle d'encombrement.

b. la connectique

On regroupe sous cette appellation les objets ne correspondant pas à un composant pour le simulateur.

La première catégorie possède les mêmes caractéristiques graphiques que les composants, il s'agit :

* des points de masse
* des points d'alimentation
* des pull-up et pull-down (résistances à l'alimentation ou à la masse)
* des jonctions (points de connexion de plusieurs fils)
* des plots d'entrée et de sortie du circuit

La seconde catégorie permet l'interconnexion des éléments du schéma (création des noeuds). Il s'agit des fils et des bus (regroupement d'un paquet de fils sur un chemin unique, représenté par un trait épais).

Leur tracé sur la feuille de travail exige le détournement des fonctions de gestion de la souris du contrôleur standard des GraphPane. On peut alors définir des fonctions évoluées telles que l'élastique (tracé dynamique d'un trait entre le dernier point sélectionné et l'extrémité du curseur de la souris), ou la notion de fil orthogonal (segments uniquement horizontaux et verticaux).

c. la fenêtre de travail

La figure 17 représente la fenêtre de l'éditeur de netlist. Les principales formes graphiques ont été placées dans la zone de travail à titre indicatif, elles ne correspondent à aucun circuit réél.

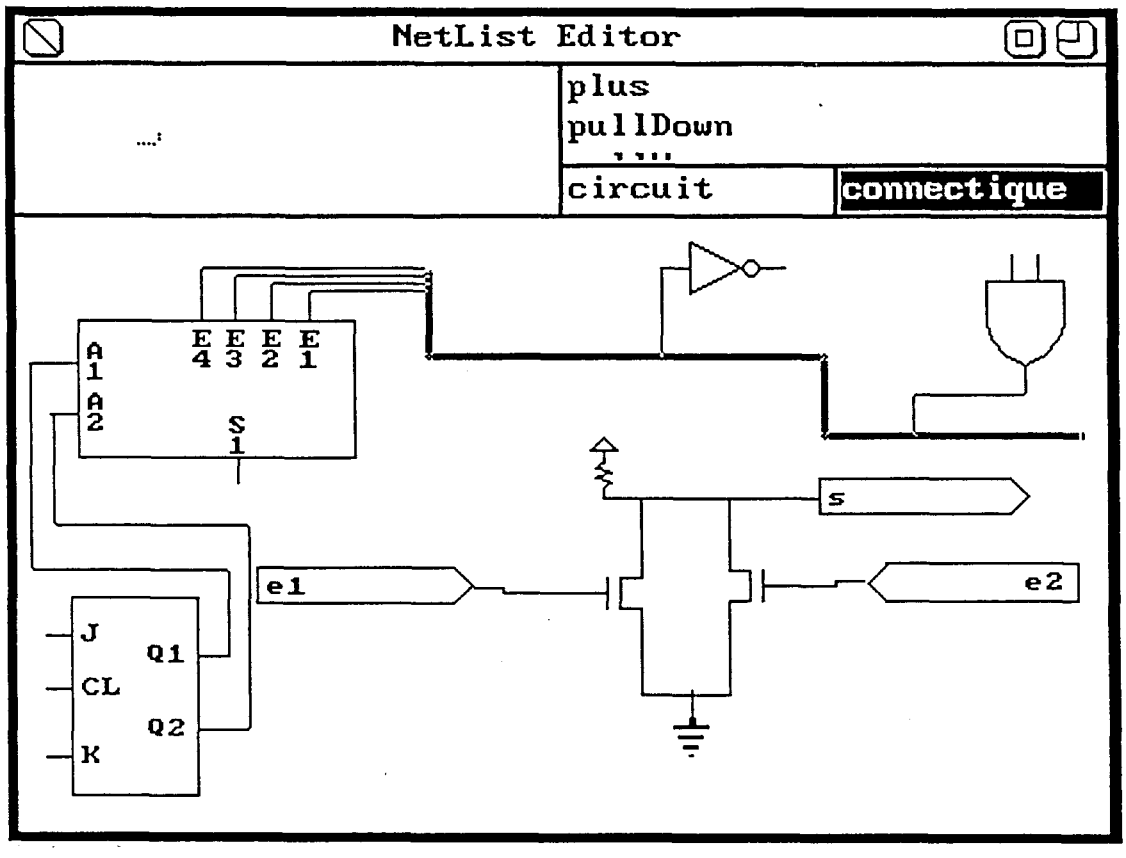


fig. 17 La fenêtre Editeur de Netlist

Ici encore, la recherche d'une interactivité agréable amène à redéfinir certaines fonctions standards. En particulier, la zone de travail étant beaucoup plus grande que la zone affichée à travers la fenêtre, il a été réalisé un défilement automatique lorsque la souris heurte l'un des bords de la fenêtre.

Ceci nécessite la définition de sous-classes au contrôleur et à la vue régissant les GraphPane. Ces sous-classe, détournent les méthodes de défilement originales afin de les adapter aux besoins exprimés.

On obtient alors un déplacement agréable à travers tout l'espace de travail, allant jusqu'à la possibilité de régler la vitesse de défilement, par sélection du pas de déplacement élémentaire.

4.9.2. L'éditeur de forme

Cet éditeur permet de donner une forme à des modèles, en vue de les intégrer dans des schémas de circuits utilisant ces modèles comme composants. Il autorise la définition de la taille du rectangle du boitier et le placement des broches d'entrée et de sortie autour de ce boitier.

La fenêtre correspondante, représentée à la figure 18, permet une construction totalement interactive de ces formes. Les fenêtres standards Smalltalk sont ici suffisantes.

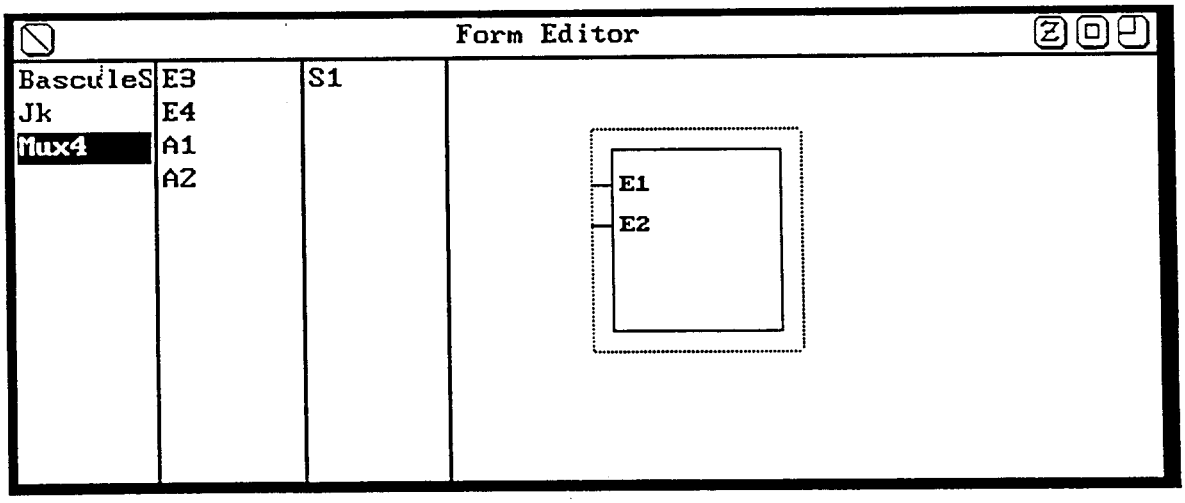


fig.18 La fenêtre Editeur de Forme

4.9.3. Fonctions particulières

a. création de la netlist

A tout moment, l'utilisateur peut demander la traduction de la structure dessinée sous forme de netlist qui apparaît dans une fenêtre espace de travail. Cette netlist peut alors être éditée, puis sauvegardée sur disque.

Afin de gagner en rapidité dans la saisie du schéma, les noms des noeuds et des circuits qui apparaissent dans la netlist sont générés automatiquement par l'éditeur.

Pour une meilleure lisibilité, l'utilisateur peut imposer ses propres noms sur tous les éléments du circuit, exceptés les noeuds reliés à la masse ou à l'alimentation qui portent les noms réservés zéro et un. Le nom d'un noeud relié à un bus est le nom du bus concaténé avec le numéro du fil correspondant.

b. sauvegarde d'un dessin

De par l'universalité de la notion d'objet, il n'existe aucune méthode de sauvegarde sur disque d'un objet. Il faut donc définir des méthodes adaptées à la structure des informations que l'on désire sauvegarder. En demandant à chaque objet d'écrire sa propre représentation sur fichier, on aboutit à des méthodes de sauvegarde simples, mais produisant une structure de fichier complexe et expansive. De plus, toute évolution de la structure risque de rendre incompatibles les formats des versions différentes. Un soin particulier doit donc être apporté à la définition des formats des fichiers de sauvegarde. On notera que la relecture suppose deux types d'action: la création d'un objet dans la classe adéquate puis l'initialisation des variables d'instances de cet objet.

4.9.4. Intégration au simulateur.

L'interface graphique a été conçue de façon relativement autonome de la partie simulation, seule la faisabilité de cette interface nous intéressait.

L'intégration totale avec le simulateur peut se faire aisément. Tout d'abord en supprimant le passage par le fichier de netlist, à condition de créer les circuits et connexions au fur et à mesure de leur apparition sur le schéma, ensuite par un lien fort avec la simulation pour la création de sondes par exemple.

La modification du schéma en cours de simulation posera des problèmes de cohérence de l'état des différents circuits, ce qui peut être résolu dans des cas simples, par exemple en relançant une simulation partielle à partir du point modifié.

4.10. Conclusions

La modularité de la conception hiérarchique s'implémente parfaitement dans la modularité de Smalltalk grâce, en particulier, au polymorphisme qui autorise une transparence du niveau fonctionnel des composants du circuit.

Un premier niveau d'interactivité homme/machine apparaît avec l'interface utilisateur de gestion de la bibliothèque de circuits. La réutilisation du gestionnaire de fichiers ,adapté aux besoins du simulateur, est un exemple parfait de la puissance des concepts de classe et d'héritage.

Les capacités de traitement symboliques de Smalltalk permettent une adaptation simple à toute syntaxe de description de netlist. L'usage de PROLOG pour ce traitement spécifique facilite encore plus la tâche, et met en évidence la simplicité d'intégration de systèmes intelligents à une application Smalltalk, avec la possibilité de répartir les tâches en fonction des possibilités de chaque système.

Enfin la réalisation de l'interface graphique a démontré qu'il était possible de réaliser du graphisme interactif de haut niveau, mais que les outils standards définis en Smalltalk sont un peu limités. Un ensemble de classes définissant des concepts graphiques évolués (segmentation, priorité, clipping...) serait un enrichissement intéressant du système de base. Ce problème ne se serait sans doute pas posé dans un système tel que C++ qui reste un langage traditionnel et peut donc s'interfacer avec des bibliothèques graphiques de haut niveau telles que GKS ou PHIGS. D'autre part, la gestion de l'écran en tant que tableau de pixel limite les performances, et s'adapterait mal aux possibilités des écrans graphiques intelligents actuellement disponibles sur les stations de travail haut de gamme.

5. Circuit et simulation : l'interface utilisateur

L'interactivité et la démarche "proche de la réalité" sont les deux points illustrés dans cette partie.

Après chargement en mémoire du circuit à simuler, l'utilisateur pourra agir sur son circuit par le biais de deux fenêtres :

* la fenêtre Circuit, pour retrouver toutes les informations liées à la description netlist du circuit;

* la fenêtre de Simulation, pour gérer les séquences d'entrées à appliquer au circuit et les écrans de visualisation des résultats de simulation.

Ces deux fenêtres sont présentées successivement: la première illustre la possibilité d'accès à toute la structure de circuit chargée en mémoire et propose un premier niveau de déroulement de la simulation. La seconde introduit de nouveaux objets qui vont permettre un déroulement interactif de la simulation : les groupes d'entrée, les groupes de sortie, les sondes et les écrans.

5.1. La fenêtre Circuit

C'est la première fenêtre qui apparaît après chargement du circuit en mémoire. Sur la figure 19, elle se trouve en arrière plan, la fenêtre d'avant-plan est la fenêtre de simulation. Tous les exemples de cette partie sont basés sur le même circuit dont la description netlist se trouve à la figure 16, et dont les schémas des composants et modèles se trouvent à la figure 11. Il s'agit d'un compteur binaire naturel par 16 'ripple carry'.

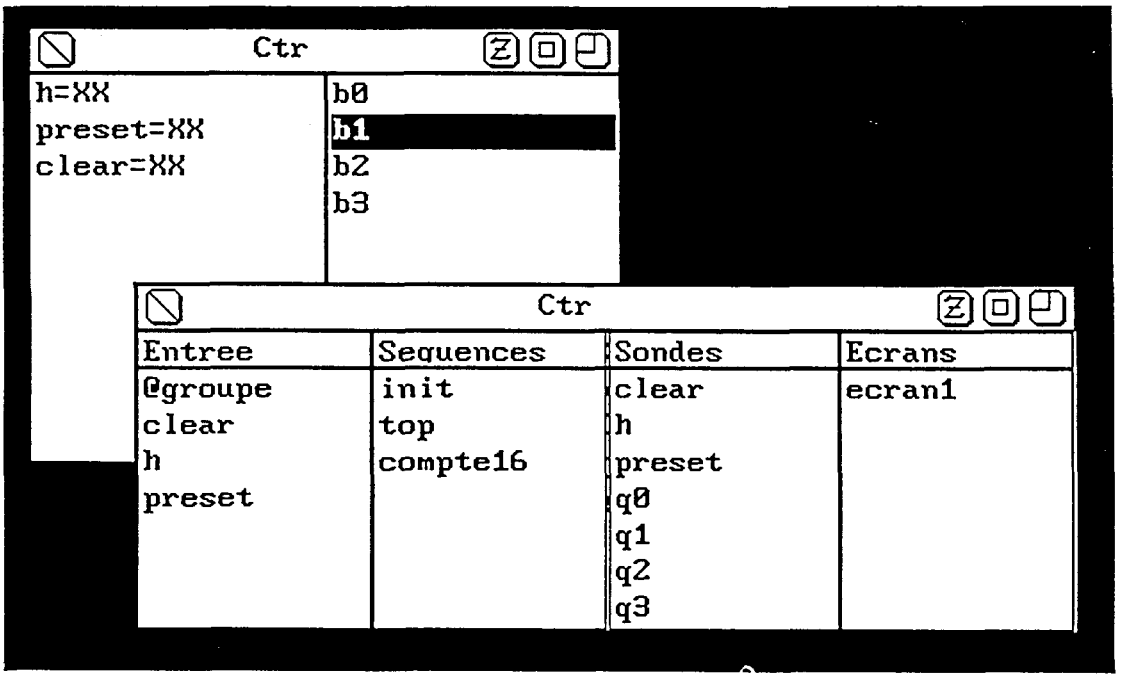


fig.19 Fenêtres circuit et simulation

Une première liste donne l'ensemble des noeuds d'entrée du circuit, avec leur état logique actuel.

Le menu associé à cette liste permet la modification d'une des entrées du circuit, et la propagation de l'événement par le simulateur.

On a donc la possibilité de mener une simulation totalement interactive, mais de très bas niveau au point de vue complexité des séquences d'entrée, comparable au test d'un prototype à partir d'états logiques positionnés par des séries d'interrupteurs.

Cette méthode a l'avantage d'être immédiate, elle est très utile pour les phases d'initialisation du circuit : établissement des alimentations (états 0 ou 1 permanents), séquences de remise à zéro ou de prépositionnement du circuit à un état connu.

La seconde partie de la fenêtre fournit la liste des composants du circuit, qu'ils soient primitives ou modèles.

On respecte la hiérarchie de description circuit/modèle, pour que l'utilisateur retrouve bien la même structure que celle qu'il a décrite dans sa netlist. Ici encore, on utilise pleinement la similitude comportementale des primitives et des modèles.

En sélectionnant un des composants du circuit, on provoque l'ouverture d'une fenêtre d'inspection de ce composant.

La notion d'inspecteur Smalltalk a été adaptée au simulateur, pour permettre le parcours de la structure du circuit et la vérification de l'état de chacun des composants.

On peut ainsi analyser l'état instantané du circuit (vérification au voltmètre de l'état des noeuds d'un prototype). C'est une analyse statique, à l'inverse de l'analyse de chronogramme que l'on examinera dans la simulation.

La fenêtre d'inspection a été complètement redéfinie. En effet, l'inspecteur Smalltalk standard donne l'image exacte de la structure mémoire de l'objet, ce qui n'est pas intéressant pour l'utilisateur qui n'a pas à connaître le fonctionnement interne du simulateur, ni la description des primitives de simulation.

Le contenu de la fenêtre d'inspection fournit donc à l'utilisateur la liste et l'état des entrées du composant, la liste et l'état des éventuelles variables internes, et la liste de composants connectés en sortie.

La sélection d'un composant connecté en sortie permet l'ouverture d'une inspection sur ce composant, on retrouve donc une possibilité de chaînage pour parcourir la structure du circuit.

Dans le cas d'un modèle, la fenêtre d'inspection contient en plus la liste des composants du modèle, liste inspectable, ce qui permet un parcours total mais hiérarchisé des composants primitifs du circuit.

La figure 20 représente une chaîne d'inspection partant de la bascule b1 du circuit, bascule qui se révèle être un modèle dont un des composants (porte et11) est inspecté à son tour. On ouvre une dernière inspection sur l'inverseur n11 connecté en sortie de la porte et11 précédente.

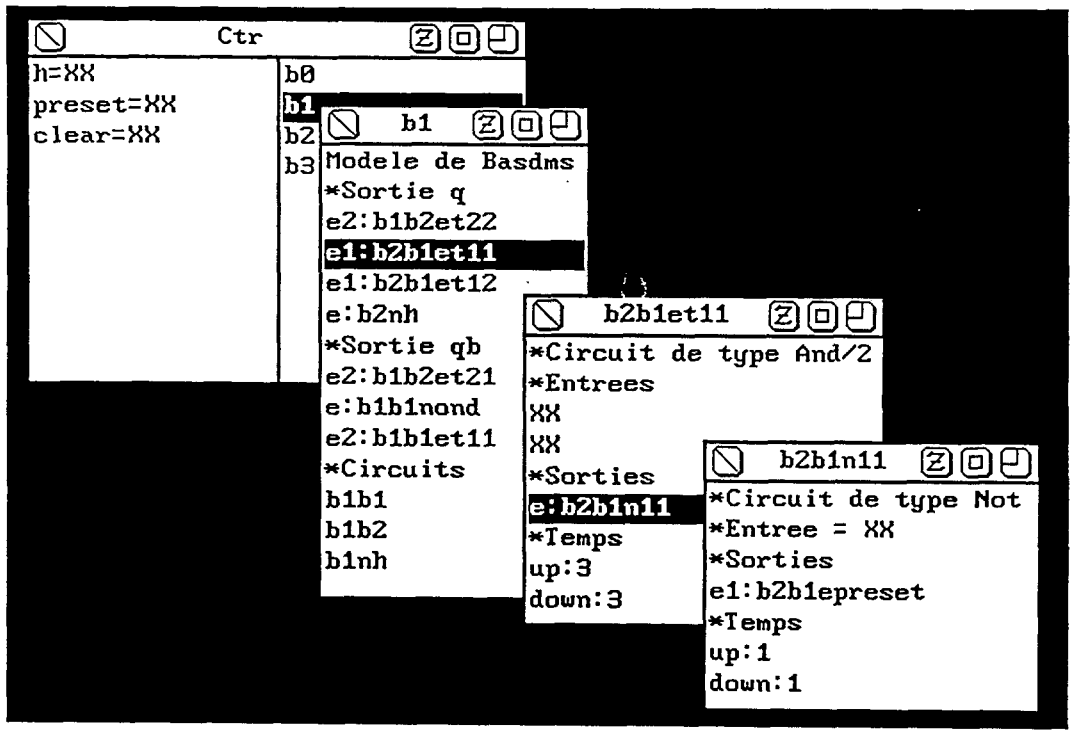


fig.20 Chaîne d'inspection de circuits

5.2. La fenêtre Simulation

Lorsque l'utilisateur désire mener une simulation plus complexe sur son circuit, il a la possibilité d'ouvrir une fenêtre Simulation, à partir de la fenêtre Circuit.

Cette fenêtre lui permettra la définition de séquence de simulation complexe et l'affichage des résultats sous forme de diagrammes temporels. Elle se présente sous la forme des quatres listes décrites ci-dessous .

5.2.1. Les entrées

Les entrées du circuit sont les seuls points sur lesquels peut agir l'utilisateur.

Afin de décrire simplement les séquences d'entrées, on a la possibilité de définir des groupes d'entrées, sur lesquels on appliquera des excitations de haut niveau.

Une excitation de haut niveau est l'expression condensée d'un ensemble d'états logiques, formant un mot dans un code spécifié. Les codes actuellement proposés sont le binaire, l'hexadécimal, le décimal et l'ASCII.

Au niveau de la simulation, ce sont quand même des éléments binaires qui seront propagés, il s'agit donc uniquement d'une facilité d'écriture des séquences d'entrée.

L'ensemble des groupes d'entrées définis sur le circuit peut être sauvegardé dans un fichier de type .gpe, afin de ne pas devoir redéfinir ces groupes à chaque début de simulation.

La méthode de définition de ces groupes est alors double. Soit on sélectionne les noeuds du groupe dans une liste des noeuds d'entrée, soit on édite un fichier de même nom que le circuit, de type .gpe, et on définit les groupes par éditeur de texte (utilisation de la fenêtre de l'interface utilisateur pour la gestion de la bibliothèque des circuits). La syntaxe est particulièrement simple, chaque groupe commençant par le nom du groupe et se terminant par le mot FIN.

Les noeuds d'entrée et les groupes apparaissent dans la première liste de la fenêtre de simulation. Les groupes se distinguent des noeuds simples par le préfixe '@'.

Un groupe peut être édité par simple sélection. On dispose alors d'une fenêtre à double liste. La première représente l'ensemble des noeuds d'entrée, la seconde la liste des noeuds déjà présents dans le groupe. Une sélection dans la première colonne ajoute le noeud à la deuxième liste. Une sélection dans la seconde colonne retire le noeud du groupe d'entrée. La figure 21 illustre cette constitution de groupe d'entrées.

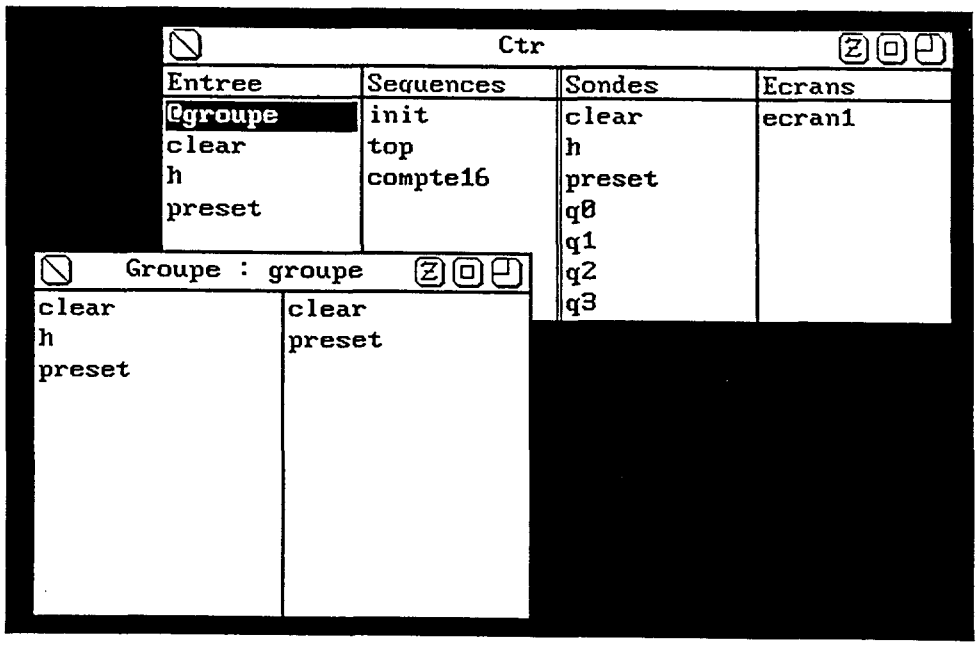


fig.21 Création des groupes d'entrées

5.2.2. Les séquences

La seconde liste de la fenêtre de simulation représente les séquences définies sur le circuit. Une séquence est une liste d'événements à produire à une heure donnée.

a. Constitution des séquences

Un événement peut être le positionnement d'une entrée (ou d'un groupe d'entrées) à une valeur donnée. Mais un événement peut être également l'exécution d'une autre séquence, avec un certain facteur de répétition.

Cette imbrication de séquence permet de définir des événements répétitifs tels que des signaux d'horloge.

exemple:

séquence top : mettre le noeud clock à 1 à l'heure 100
mettre le noeud clock à 0 à l'heure 200
séquence principale : répéter 10 fois la séquence top à l'heure 1000

Les temps indiqués sont des temps relatifs à un instant zéro où débute la séquence. Ainsi dans l'exemple précédent, l'horloge présentera un front descendant aux instants 1200,1400,1600...

L'imbrication des séquences peut être quelconque, seule la récursion est interdite, sous peine de bouclage infini durant la construction de la séquence.

Les séquences peuvent être sauvegardées, exécutées ou détruites. La sauvegarde crée un fichier de même nom que le circuit et de type .seq. De même que pour les groupes d'entrées, ces fichiers peuvent être créés ou modifiés par l'éditeur de texte sous l'interface utilisateur de gestion de la bibliothèque de circuits.

Lorsqu'une séquence utilise un groupe d'entrée, on sauvegarde (on recrée) le groupe d'entrée en même temps qu'on sauvegarde (on restaure) la séquence, sinon l'objet simulation n'est plus cohérent.

b. Syntaxe de définition de séquences

Une séquence commence par un nom et se termine par le mot FIN. Chaque ligne représente un événement. La première information est l'heure relative de l'événement.

Puis vient le nom de l'entrée, du groupe ou de la sous-séquence à appliquer. Une entrée simple est suivie de l'état logique appliqué (0,1 ou X). Un groupe d'entrée est suivi du nom du code utilisé (bin, dec, hex ou asc) puis de la valeur à appliquer à ce groupe. Une sous-séquence est suivie du mot clef 'répétition:' puis du nombre de fois où la séquence devra être répétée.

```
exemple:   comptage
              0 load = 1
              0 udb = 1
              0 @data Hexa D
              0 enabletb = 0
              0 clock = 1
              0 enablepb = 0
              0 clear = 0
              0 preset = 1
              50 clear = 1
              100 load = 0
              200 load = 0
              300 top Repetition: 20
              FIN
              top
              100 clock = 0
              200 clock = 1
              FIN
```

Ce fichier sert à la simulation d'un circuit LS169, compteur-décompteur préchargeable. Il décrit deux séquences, une séquence principale appelée comptage et une sous-séquence appelée top. Les huit premiers événements de la séquence comptage ont tous lieu au même instant. Ils servent à initialiser le circuit.

Le groupe @data constitue le mot auquel on précharge le compteur. Le compteur est ensuite remis à zéro par le clear qui est à zéro puis repasse à un à l'instant 50. On charge ensuite la valeur ODH par un état bas sur le lOAD puis on lance une séquence de comptage sur 20 tops d'horloge. La figure suivante est le chronogramme correspondant à cette séquence.

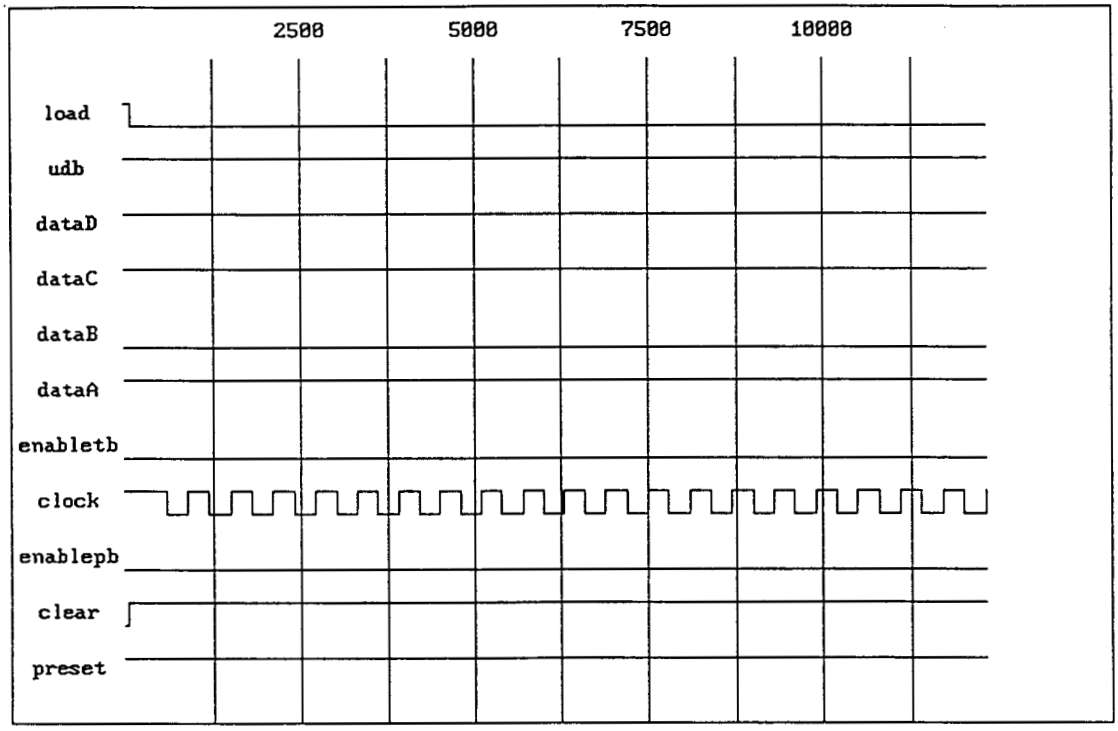


fig.22 Chronogramme de la séquence d'entrée pour le 74LS169

c. Création interactive des séquences

L'écriture des séquences peut être automatisée par une interaction utilisant au maximum la souris.

Pour cela, on propose à l'utilisateur une fenêtre à 2 colonnes. La première représente les choix possibles en cours de construction, la seconde représente l'état de la séquence. Cette structure a été choisie pour ne pas multiplier inutilement le nombre de fenêtres proposées à l'utilisateur. Le contenu de la première liste de la fenêtre évolue selon un graphe d'états finis à 5 états repris dans la figure ci-dessous.

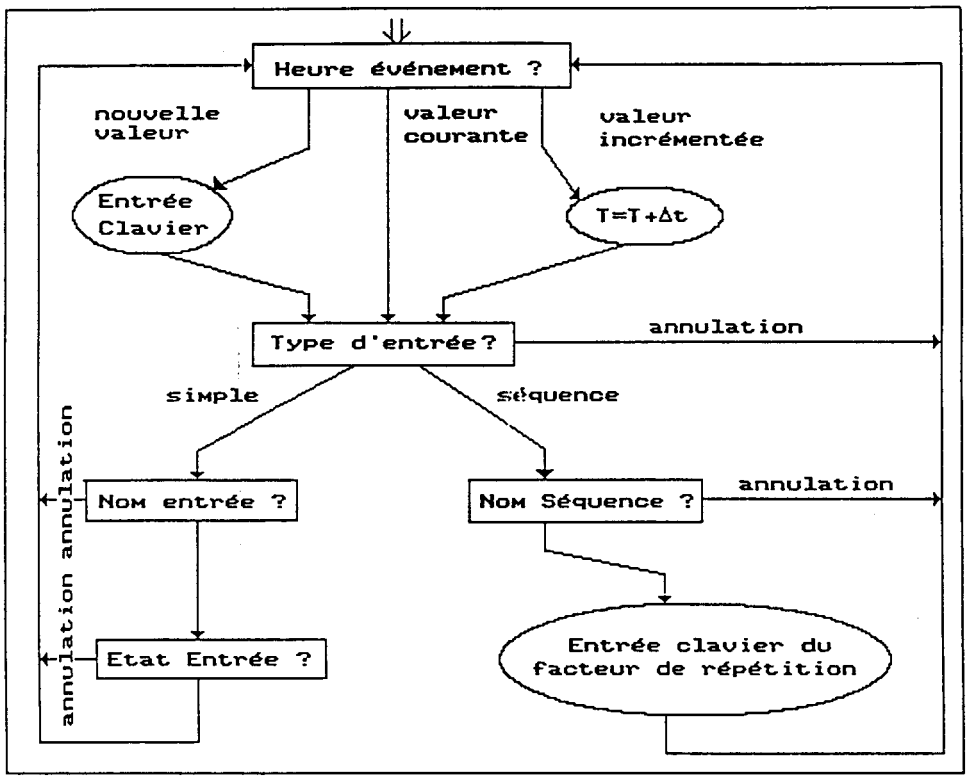


fig.23 Graphe de construction de séquences

La figure suivante représente la fenêtre en situation initiale de choix de l'heure du prochain événement.

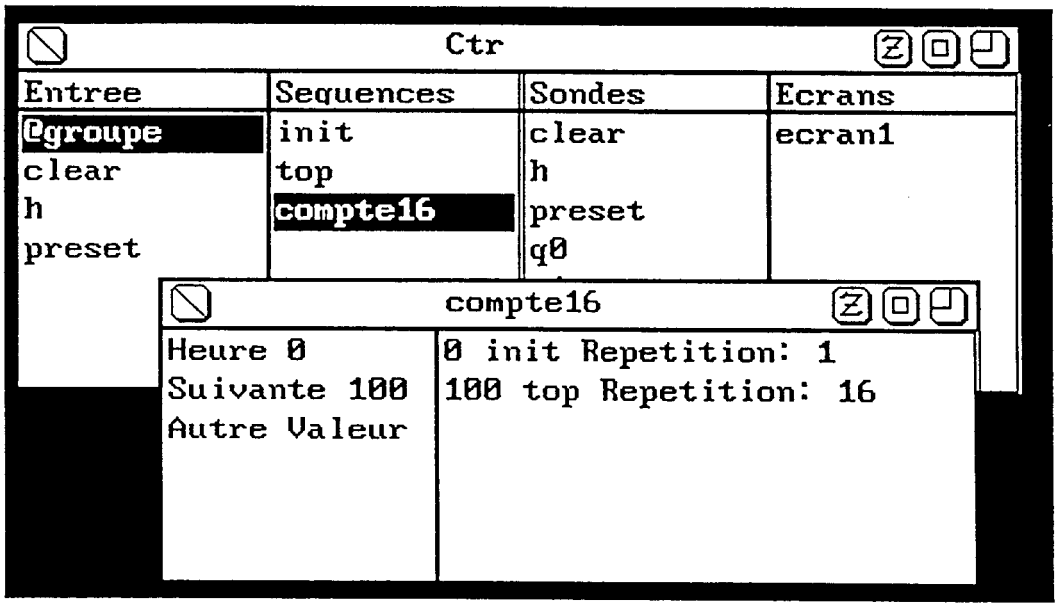


fig.24 Fenêtre de construction de séquence

On définit la notion d'heure courante et d'incrément. Toute entrée d'un nouvel événement se fait à la même heure que l'événement précédent ou un certain incrément de temps plus tard.

L'utilisateur peut donc dérouler la progression de sa simulation sans devoir taper systématiquement les heures auxquelles se produisent les événements.

L'heure courante ainsi que l'incrément de temps sont modifiables à tout moment. Ils sont au départ initialisés respectivement à 0 et 100.

Une fois l'heure choisie, on demande à l'utilisateur le type (entrée simple ou séquence) de l'événement. On lui propose alors la liste des entrées et des groupes d'entrées définis, ou alors la liste des séquences prédéfinies pour le circuit.

Finalement, il reste à rentrer la valeur de l'événement. S' il s'agit d'un noeud d'entrée simple, un menu propose les valeurs 0,1,X possibles. S'il s'agit d'un groupe, l'utilisateur devra sélectionner le code binaire utilisé puis la valeur dans ce code. Si l'événement est une sous-séquence, l'utilisateur devra entrer le facteur de répétition.

Les deux derniers cas font apparaître une utilisation obligatoire du clavier, pour l'introduction de valeurs définies sur des domaines trop importants pour être proposés sous forme de choix à l'utilisateur. Dans le cas de la sous-séquence, on propose tout de même un choix par défaut d'une répétition unique, dans ce cas l'utilisateur n'a pas à lâcher la souris pour repasser au clavier.

L'intervention de l'utilisateur pourrait être encore simplifiée, par l'adjonction de génération automatique de séquences de comptage ou de séquence aléatoire. Ceci s'intègre très facilement dans la structure décrite, en ajoutant par exemple un choix dans le menu de la fenêtre de création de séquence.

5.2.3. Les sorties

La troisième partie de la fenêtre de simulation permet de gérer les signaux de sortie que l'on voudra visualiser après simulation. L'état d'un noeud ne sera observable qu'à condition d'y avoir connecté un objet sonde.

a. Les objets sondes

La sonde, comparable à la sonde d'un analyseur d'état logique, est un élément de mémorisation des événements qui se produisent sur un noeud.

Les événements sont mémorisés sous la forme d'un couple (heure,état), il ne s'agit donc pas de l'échantillonage de l'état des sorties.

Si le nombre d'événements à mémoriser par les sondes était trop important pour la taille mémoire de la machine, il serait aisé d'étendre cette mémorisation sur un fichier disque, ou de ne déclencher la mémorisation que sur une combinaison d'événements externes qui correspondent aux instants intéressants de la simulation (sondes à mémorisation déclenchable).

L'adjonction de sonde au circuit se fait par sélection dans une liste des noeuds du circuit. Un exemple d'une telle liste apparaît sur la figure suivante.

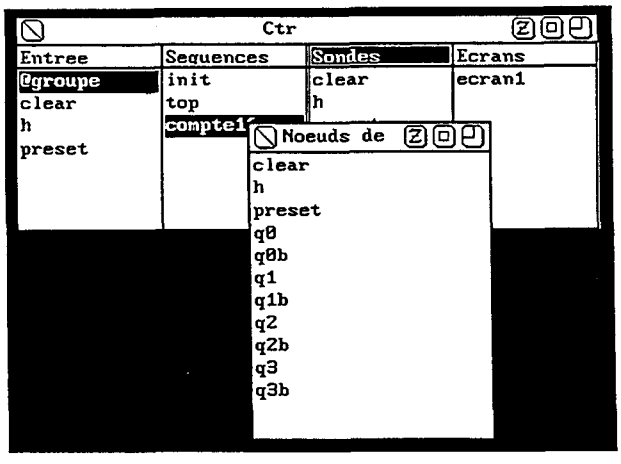


fig.25 Fenêtre de création de sondes

b. Groupes de sorties

Tout comme pour les groupes d'entrées, il peut être intéressant de regrouper des sondes, pour visualiser le mot binaire qu'elles constituent en un code de haut niveau (décimal, hexadécimal, ascii...).

Ces groupes peuvent être définis à tout moment, avant, pendant ou après la simulation, la codification n'est assurée qu'au moment de la visualisation.

Ces groupes sont définis par une fenêtre à double liste du même type que celle utilisée pour constituer les groupes d'entrées. Un exemple est donné dans la figure 26.

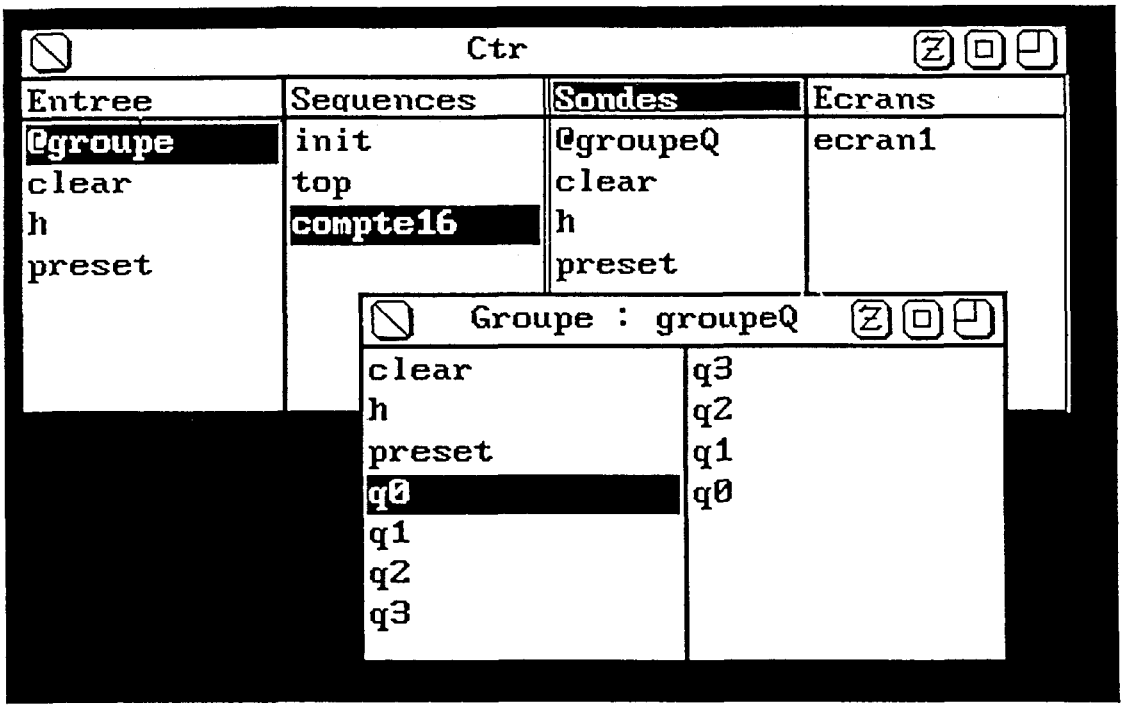


fig.26 Fenêtre de création de groupes de sorties

Les groupes de sorties sont sauvegardables et peuvent donc être modifiés sous éditeur de texte.

On peut également demander la constitution de groupes de sorties identiques aux groupes d'entrées (visualisation des entrées sous la même forme que celle utilisée dans la description de la séquence).

5.2.4. Les écrans graphiques

La dernière partie de la fenêtre de simulation permet de gérer des écrans de visualisation.

Un écran est un regroupement de sondes et de groupes de sondes, en vue de l'affichage du résultat de la simulation sous forme graphique (chronogrammes).

La création de l'écran se fait donc par sélection du contenu que l'utilisateur désire y voir apparaître. Une même sortie pouvant être sélectionnée plusieurs fois pour rendre le chronogramme le plus lisible possible (cf figure 27).

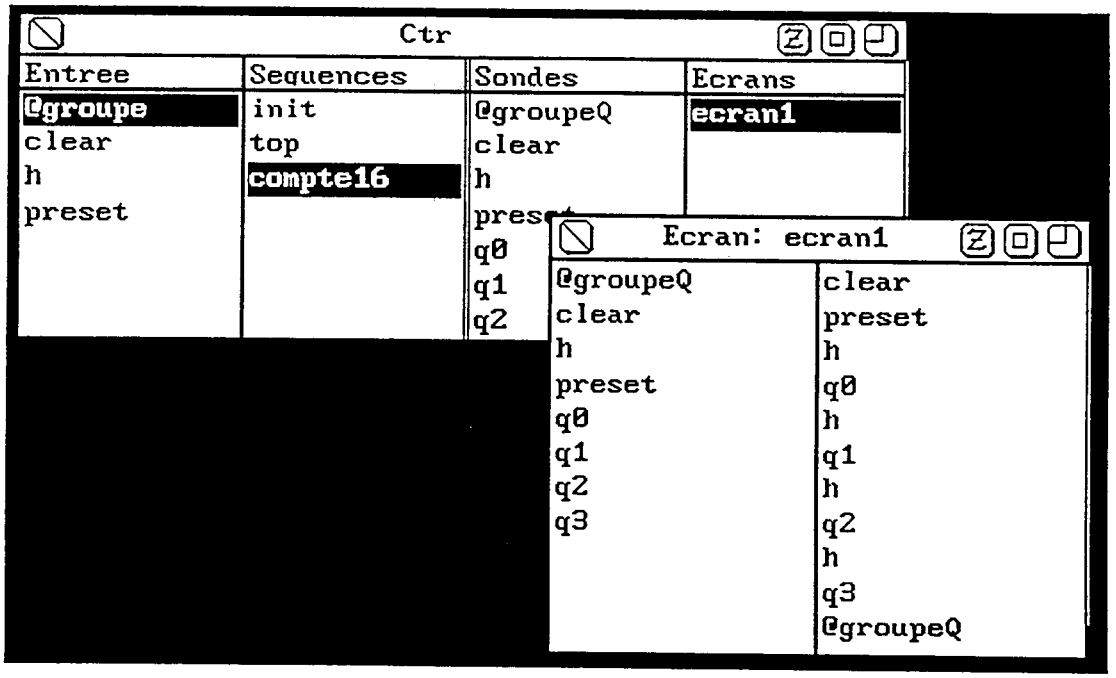


fig.27 fenêtre de création d'écran

L'affichage se fait sous la forme classique de chronogramme, avec un trait épais central pour symboliser un état indéterminé, et un changement de couleur pour symboliser les états résistifs.

Plusieurs écrans peuvent être définis, avec différents signaux ou visualisant différents instants de la simulation.

L'écran, à sa création, affiche l'évolution des signaux sur la durée complète de la simulation. Un système de zoom à grossissement quelconque et un déplacement à niveau de zoom constant permettent une analyse fine des détails des chronogrammes (cf figure 28).

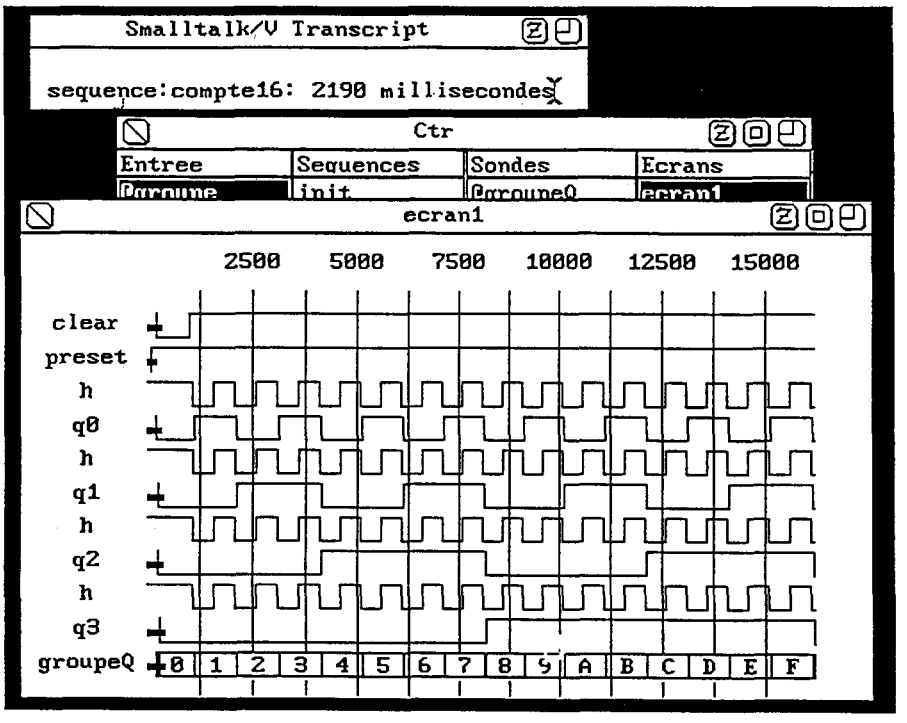


fig.28 fenêtre écran de visualisation

Un curseur commandé par souris permet un calcul des temps exacts de changement d'états des signaux. Il permet également de désigner les deux bornes de l'intervalle sur lequel on désire un zoom, aucune action au clavier n'est donc nécessaire.

Toute nouvelle séquence exécutée peut être reprise en compte sur l'écran, permettant un déroulement progressif et interactif de la simulation.

Enfin, le contenu de l'écran peut être envoyé sur imprimante, pour garder une trace des résultats de la simulation. Smalltalk travaille au niveau de l'écran en bitmap (tableau de points). On peut donc envoyer cette image sur tout périphérique acceptant ce type d'image. Un driver d'imprimante laser a été réalisé pour sortir les chronogrammes en différentes tailles, et en différentes orientations.

5.3. Conclusions

Le développement de l'interface utilisateur utilise pleinement les possibilités du concept MVC (Model View Controller), pour réaliser un dialogue avec l'utilisateur le plus naturel possible. On obtient ainsi une présentation et un mode de dialogue utilisateur similaire aux autres fenêtres Smalltalk, l'intégration à l'environnement est donc parfaite.

La création de fenêtres de dialogue à partir d'une fenêtre centrale (ici la simulation), permet une minimisation de l'action au clavier. L'utilisateur peut donc réaliser la majeure partie de sa simulation à l'aide de la souris uniquement. Ceci est particulièrement important pour l'ergonomie du système.

Plusieurs fenêtres et modes de dialogue ont été essayés avant la version finale. Les données et le but du dialogue ne changeaient pas, seuls la forme des fenêtres, la disposition des colonnes, le contenu des menus et l'interprétation des sélections étaient modifiés.

Chaque version faisait alors l'objet d'un test d'utilisation qui, généralement, révélait au bout de quelques minutes les défauts majeurs. Cette méthode n'est bien entendu possible que par la rapidité de maquettage autorisée par Smalltalk.

La gestion des événements et l'aspect visuel du résultat étant indépendant de l'application (du modèle), tout changement peut être très rapidement réalisé et testé. Seuls les événements de dialogue provoqués par l'utilisateur sont à traiter, les actions en résultant ne modifient aucunement la structure des données traitées.

Le résultat actuel semble assez agréable, il minimise les allers-retours entre la souris et le clavier, ainsi que les déplacements de la souris (ordre des commandes dans un menu par exemple). Des sécurités (demandes de confirmation) protègent les actions les plus critiques.

Il reste toujours la possibilité à l'utilisateur averti de modifier ces fenêtres, pour changer de mode de dialogue ou pour ajouter de nouvelles fonctions.

D'autre part, une alternative d'édition de texte est systématiquement proposée. En effet, pour des problèmes de grande taille, il peut être fastidieux de réaliser des sélections nombreuses quand un éditeur autorise la duplication et la modification rapide de lignes de texte.

La volonté de rester proche de la réalité prend pleinement son sens dans la manipulation d'objets, tels que les sondes ou les écrans de visualisation.

De même, la possibilité d'exprimer ou de visualiser un mot binaire en un code évolué rend le travail de l'utilisateur plus proche de l'image qu'il a du circuit, et donc rend transparent l'outil de simulation. L'utilisateur ne doit pas s'adapter au simulateur, c'est le simulateur qui se plie à sa vision du monde.

Ici encore, un utilisateur avec une connaissance minimale de Smalltalk, peut définir des systèmes de codage d'information qui lui sont propres, et l'ajouter aux systèmes classiques prédéfinis (hexadécimal, ASCII ...).

La traduction du dialogue en terme de classe se fait simplement, chaque fenêtre est en effet associée à un objet de classe différente. Ainsi on trouve des classes GroupeEntrée, GroupeSortie, EcranGraphique ... Ces objets sont fédérés à un objet de classe Simulation qui maintient des listes d'éléments définis. La fenêtre circuit est directement reliée à l'objet de classe Circuit qui a la connaissance du circuit à simuler.

6. Le simulateur

Cette partie décrit les mécanismes internes du simulateur (gestion des événements), et la structure des primitives de simulation. Les problèmes de performance de l'environnement objet seront abordés, ainsi que la possibilité d'ajouter de nouvelles primitives de niveau fonctionnel élevé.

6.1. Gestion des évènements

La simulation est conduite par une classe de simulation qui gère une liste de réveil de circuit.

Chaque changement d'état sur la sortie d'un circuit ajoute un Elément de Réveil à cette liste, composé de :

* l'heure prévue du réveil
* la liste des éléments de connexion concernés (couple circuit/entrée)
* la valeur à appliquer sur ces entrées.

La liste de réveil est triée par ordre chronologique, la boucle de simulation vient chercher à chaque itération l'élément de tête de cette liste.

Trois solutions de gestion de liste de réveil ont été essayées. La première version, la plus simple, consistait en l'utilisation d'une liste triée Smalltalk de la classe SortedCollection. L'avantage de la fonction prédéfinie est contrebalancé par un temps d'exécution médiocre. En effet, la classe SortedCollection est prévue pour gérer une liste triée dans une structure mémoire continue, toute insertion est donc coûteuse puisqu'elle nécessite un décalage des éléments de la liste.

La seconde solution est alors de créer une classe ListeTriée gérée en liste chaînée (figure 29). La structure est donnée à la figure suivante, elle est comparable aux listes LISP traditionnelles à base de paires pointées. On utilise pleinement la récursivité pour parcourir et insérer des éléments dans la liste.

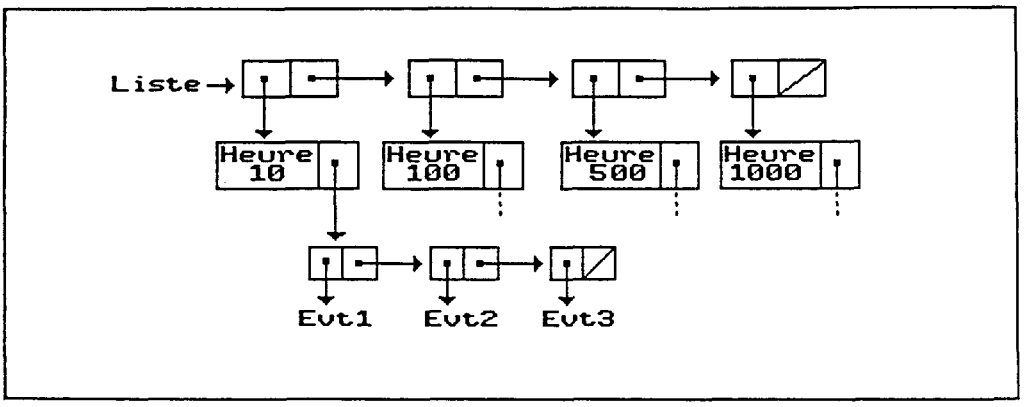


fig.29 Structure de ListeTriée

La troisième solution est une extension naturelle de la précédente [Ped81]. En effet, les insertions se font en général pour des événements proches. On peut imaginer la gestion en tableau des événements pour les N prochaines tranches de temps élémentaires. L'heure courante (T_{actuel}) désignera un certain emplacement P du tableau. L'insertion se fait alors en accès direct, à la position :

$$(P + T_{actuel} - T_{événement}) \text{ modulo } N$$

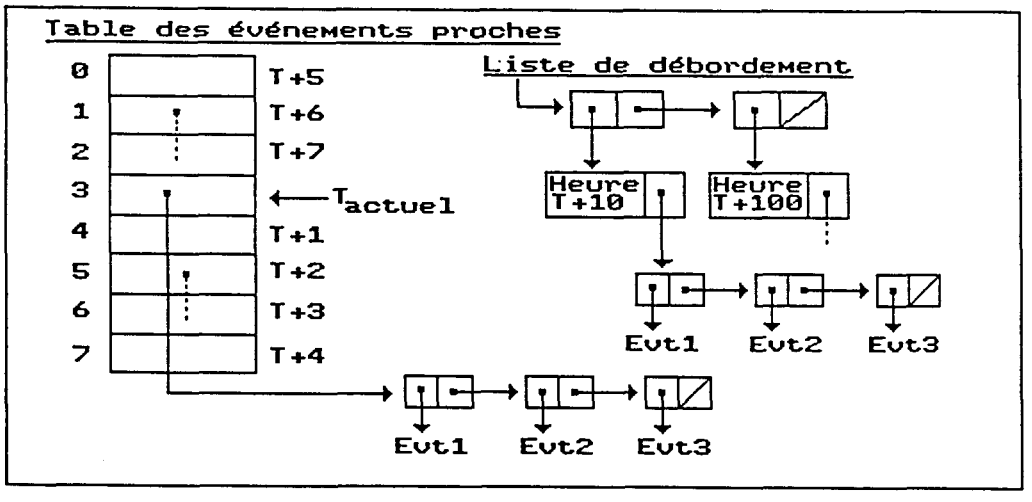


fig.30 Structure évoluée de gestion des événements

Les événements lointains (heure > $T_{actuel}+N$) seront stockés dans une liste de débordement d'éléments chaînés (figure 30). A chaque progression du temps courant, les éléments en liste de débordement sont ramenés dans le tableau si besoin. En outre il est conseillé de choisir une taille du tableau telle que $N = 2^k$, ainsi les opérations modulo se ramènent à un simple masquage.

tableau telle que N = 2^k, ainsi les opérations modulo se ramènent à un simple masquage.

Cette méthode s'est révélée moins performante que la gestion de liste triée en élément chaînés. En effet, si les principes à la base de cette méthode semblent de bon sens, elle ne peut être optimale qu'avec une gestion de la mémoire et des calculs d'accès proches de la structure physique de la mémoire centrale. Elle est donc à réserver à une programmation traditionnelle. Il aurait fallu la programmer directement en assembleur et donc sortir des concepts objets. A nouveau, une meilleure adéquation machine-concept objet permettra d'obtenir des performances acceptables.

6.2. Les états logiques

La notion de type de la programmation classique se traduit en langage objet sous forme de classe. Il est donc naturel de définir la notion d'état logique en une classe spécifique, capable de réaliser les opérations de base sur des variables booléennes.

De même qu'il existe les classes True, False ou UndefinedObject, pour générer les constantes uniques true, false et nil, on définira les états logiques de manière unique en garantissant cette unicité par un refus de création lorsqu'une instance existe déjà.

Six classes ont été prédéfinies, correspondant aux 6 états logiques traités par le simulateur: Un,Zo,XX,Pu (Un résistif),Pd (Zo résistif),ZZ (haute impédance).

Ces classes sont toutes sous-classes de EtatLogique. On peut créer des instances de cette sur-classe qui seront alors des états logiques abstraits, que l'on peut assimiler à des variables logiques. Cette définition de variables logiques permet d'envisager l'extension du simulateur à la simulation symbolique. Il suffira d'intégrer la possibilité d'affichage d'expressions symboliques dans les écrans de visualisation, ou de définir un mode de visualisation spécifique. L'excitation des circuits en symbolique sera également à intégrer aux possibilités actuelles.

6.3. Les mots binaires

On rencontre souvent des informations propagées par des ensembles d'états binaires, selon un code conventionnel.

La classe MotBinaire permet la manipulation de ce type d'informations. Il s'agit d'une classe ayant beaucoup de méthodes communes avec les tableaux mais dotée, en outre, de méthodes de transcodage, et d'opérations arithmétiques ou logiques en mot à mot.

On y trouve par exemple les opérations d'addition de mots binaires, de complément à deux, d'inversion, d'incrémentation. On trouve également des calculs de valeur décimale correspondante, ou d'affichage en binaire, hexadécimal, décimal ou ASCII.

Cette classe simplifie considérablement la description comportementale des circuits travaillant au niveau mot. L'objet MotBinaire est en effet comparable à un registre, et les méthodes de calcul définies sur cette classe sont comparables à des opérateurs. La description est donc très proche des langages à transfert de registre, avec une syntaxe Smalltalk relativement simple. Un exemple d'une telle description est donnée en annexe.

6.4. Les primitives

6.4.1. Structure globale des primitives

Les primitives sont caractérisées par les données suivantes :

* le père qui est l'objet auquel le fils appartient. Le père peut être un circuit ou un modèle. Toute information relative à la description netlist du circuit doit être demandée au père.

* la liste ou le tableau de listes de sorties contient tous les éléments connectés sur la ou les sorties de la primitive.

Chaque primitive est capable de répondre aux méthodes suivantes :

* nomsDesEntrées : la primitive retourne la liste des noms des entrées valides du circuit (noms des entrées du composant, pas des noeuds qui y sont connectés stockés au niveau du circuit). Ces noms seront employés comme méthode à un paramètre et sont donc suivis par le caractère ':'.

* nomsDesSorties : la primitive retourne la liste des noms des sorties valides.

* connectEntrée:duCircuit:àLaSortie: : ajoute un élément de connexion à la liste de sorties.

* étatCircuit et selectInfo : méthode d'inspection de la primitive. La première retourne une liste d'informations sur le composant, la seconde réagit à toute sélection par l'utilisateur d'un élément dans cette liste.

L'excitation d'une primitive se fait par envoi d'un message portant le même nom que l'entrée avec, comme argument, la nouvelle valeur. Cette valeur est mémorisée pour permettre un calcul de l'état de la primitive et de ses sorties. On retrouve une description du comportement par événement qui se rapproche des langages comportementaux.

D'autres méthodes ou variables seront décrites dans le chapitre amélioration des performances. Elles ne sont pas indispensables à la simulation normale des circuits, elles ont pour but de fournir des renseignements sur la structure et le comportement de l'objet. L'ensemble des méthodes définies sont décrites en annexe.

6.4.2. Les fonctions combinatoires standards

La description comportementale des primitive combinatoires est simple. La valeur des entrées est mémorisée dans des variables simples ou dans des mots binaires lorsque ces entrées sont de même type. Ainsi, pour un multiplexeur, on utilise 1 mot binaire pour les entrées, un mot binaire pour les adresses, et une variable simple pour l'entrée de validation du circuit.

Les primitives booléennes sont regroupées dans une classe particulière pour partager l'ensemble des caractéristiques propres à leur structure (2 à N entrées de même type, sortie unique). Toutes les fonctions de base sont disponibles (et, ou, non-et, non-ou, ou-exclusif, inversion, retard pur).

Les fonctions multiplexeur, démultiplexeur sont également présentes. Toute autre fonction peut être facilement ajoutée, le mécanisme d'héritage aboutit à une écriture réduite au strict minimum.

6.4.3. Les fonctions séquentielles standards

Les fonctions séquentielles nécessitent l'utilisation de variables d'état interne. Elles posent peu de problèmes de définition.

La classe Bascule regroupe les bascules D et JK, ici encore, l'adjonction d'autres modèles est facilitée par le partage du comportement commun au niveau de la sur-classe.

La fonction registre fournit une primitive très générale de mémorisation, avec remise à zéro, à un, incrémentation et décrémentation du contenu (utilisation en compteur).

6.4.4. La fonction haute-impédance

Les connexions de type bus peuvent être rajoutées par la définition de porte tri-states avec ou sans inversion. Le principe adopté dans la simulation vise à ne pas perturber le déroulement normal des opérations.

Toutes les connexions se font normalement. Mais pour les portes tri-states, on construit une liste de partage du noeud de sortie (liste des co-tri-states). A chaque déblocage d'une porte, on vérifie que les autres portes sont bien en haute impédance, sinon une erreur est signalée, et un signal à valeur indéterminée est propagé.

La liste des co-tri-states est construite lors de la phase d'analyse sémantique de la description netlist. Elle est communiquée aux portes concernées par un message spécial après création du circuit.

6.4.5. Les mémoires

Les fonction Ram, Rom et PLA sont traitées sur un même modèle. Le contenu de ces circuits peut être initialisé par lecture d'un fichier de valeurs. Ainsi toutes les mémoires vives sont initialisées par lecture du fichier de même nom que le circuit et de type .ram .

Le fichier contient toutes les initialisations des primitives du même type dans le circuit. Il faut passer un paramètre indiquant le numéro d'ordre dans le fichier de la séquence d'initialisation de chaque composant.

Le fichier d'initialisation possède une syntaxe simple. Chaque ligne désigne une valeur à stocker dans la mémoire, avec un facteur de répétition. La valeur peut être indiquée en binaire (B), décimal (D), hexadécimal (H) ou ASCII (A). L'initialisation se termine par le mot 'FIN'.

exemple:

```
4HA
2B0110
10AE
FIN
```

Cette séquence initialise les quatre premiers mots à la valeur hexadécimale A, les 2 suivants à la valeur binaire 0110, et les 10 derniers au code ASCII de la lettre E.

6.4.6. Les commutateurs

Dans le domaine de la très haute intégration en technologie MOS et dérivées, la conception se fait souvent en terme de transistors et non de portes logiques. L'avantage du transistor MOS est la simplicité de sa modélisation, puisqu'il est assimilable à un interrupteur (switch).

Dans le cadre de la simulation logique, cette modélisation pose problème. En effet, même avec un modéle aussi simple, la simulation relève davantage de problèmes électriques (calcul de charges, de tensions...) que de problèmes logiques fonctionnels.

En général, la fonction logique est réalisée par un groupe de transistors et non par un transistor seul. La définition d'une primitive transistor ne pourrait donc s'intégrer dans le simulateur.

La solution adoptée, inspirée de [Ama85], consiste à ne considérer que les objets réellement intéressants lors de la simulation. Ces objets sont les branches complètes de transistors reliant un noeud terminal (état logique constant ou entrée), au noeud de sortie considéré (figure 32.a).

Toute excitation envoyée à un réseau de transistors aboutit soit à une grille, soit à l'extrémité d'une branche. Elle est mémorisée puis les noeuds de sortie reliés à cette branche sont examinés.

Pour chaque noeud, on parcourt les branches qui y aboutissent, ce qui nous amène plusieurs états possibles. La discrimination se fait par considération sur la puissance des signaux amenés : un état fort (Zo,Un,XX) l'emportera sur un état faible (Pu,Pd,ZZ)(figure 32.b).

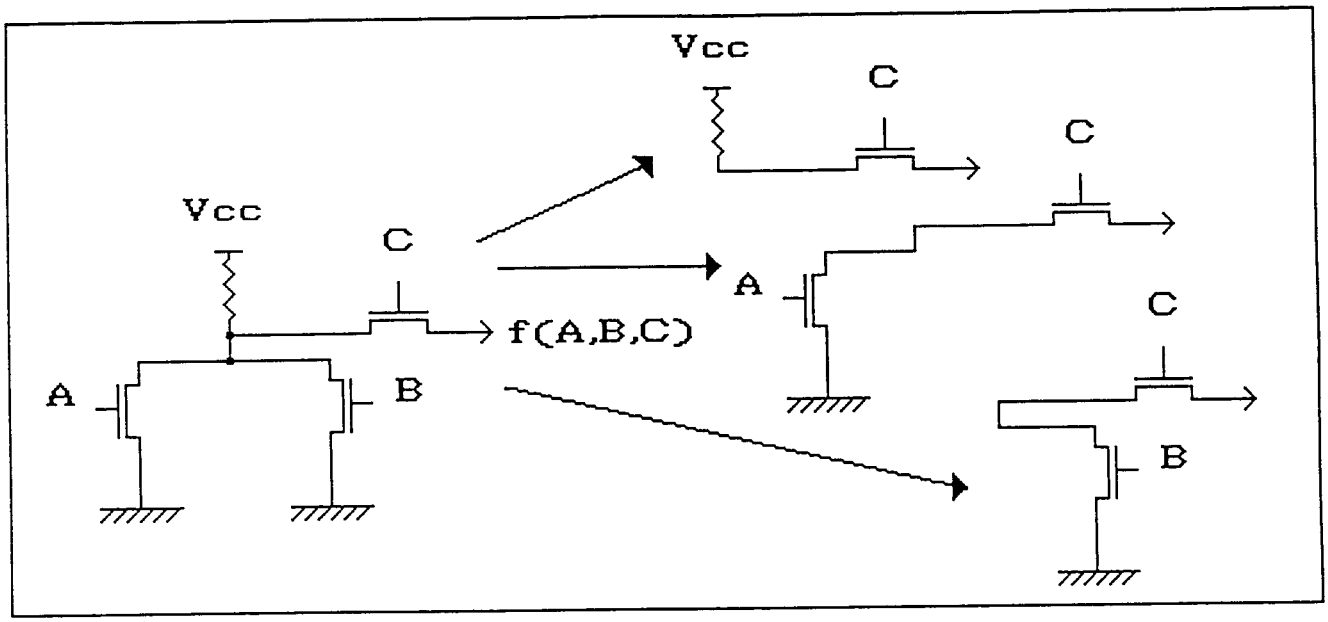


fig.32.a gestion des transistors MOS
décomposition en noeuds et branches

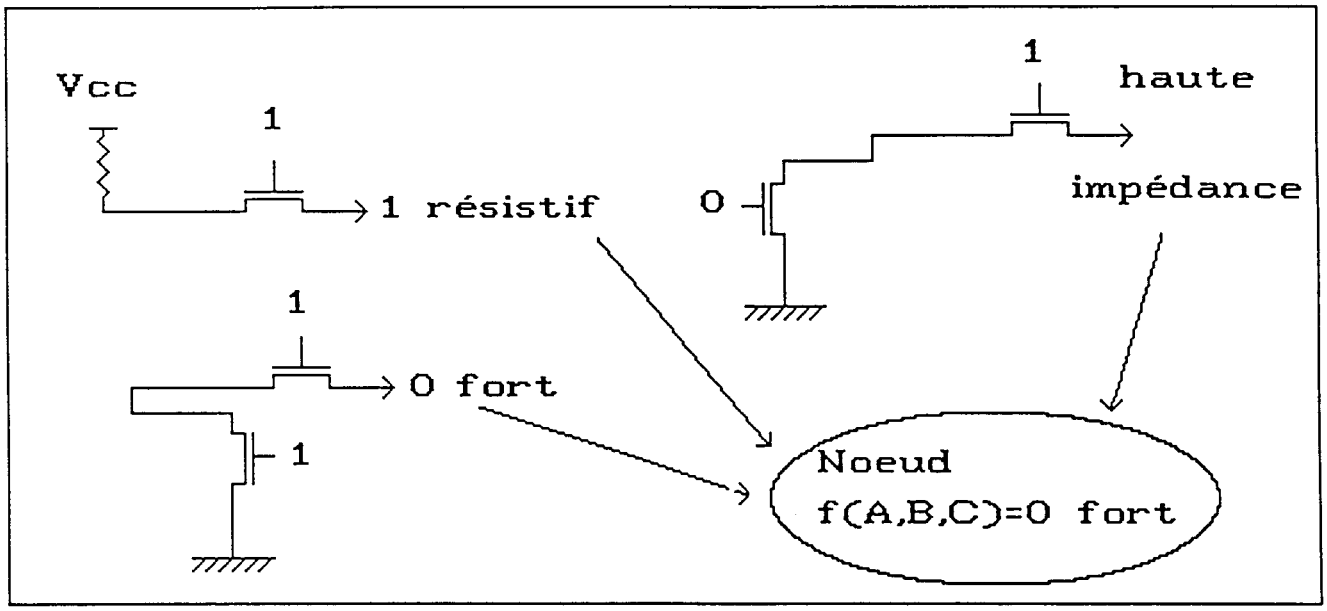


fig.32.b gestion des transistors MOS
résolution de conflit

La définition de deux niveaux de puissance (état logique fort ou état logique résistif) est le minimum permettant une compréhension fonctionnelle correcte d'un réseau.

Il pourrait être intéressant de créer des niveaux intermédiaires qui permettraient de tenir compte des pertes de puissances apportées par chaque transistor, et obtenir une simulation plus proche des réalités électriques. Des problèmes de partage de charges pourraient être ainsi traités. Toutefois, le nombre de niveaux de puissance sera toujours en nombre fini et relativement limité, la simulation ne sera jamais équivalente à une simulation électrique pure.

La structure de chemins et de noeuds permet la simulation dans tous les cas de circuits classiques. La complexité de la simulation n'est plus fonction du nombre de transistors, mais elle est fonction du nombre de branches dans le circuit. Ainsi le réseau de Tally, représenté à la figure 33, est un des cas les plus complexes possibles, puisque pour 18 transistors, il offre 140 branches, d'où une simulation particulièrement coûteuse.

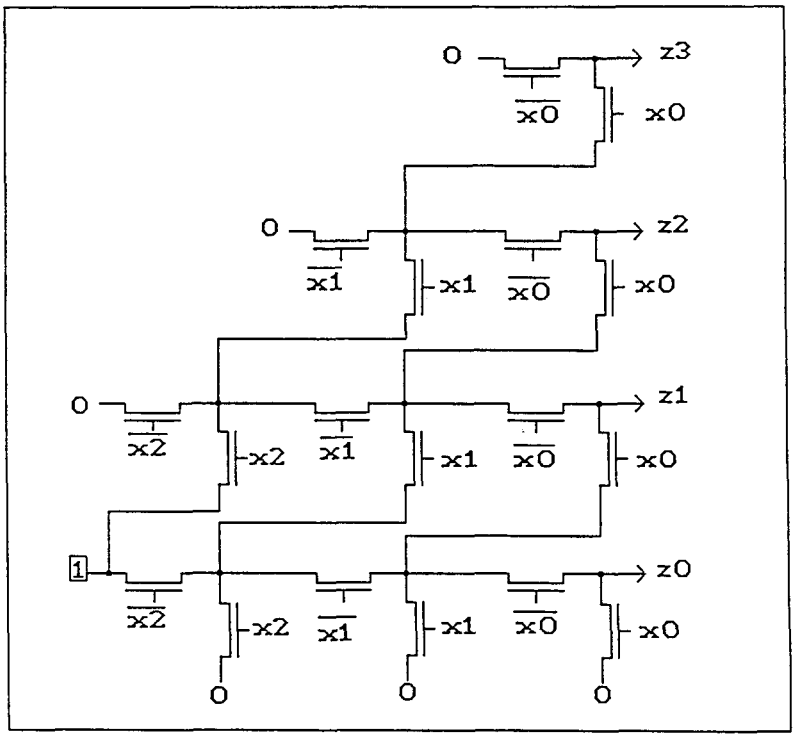


fig.33.a Réseau de Tally

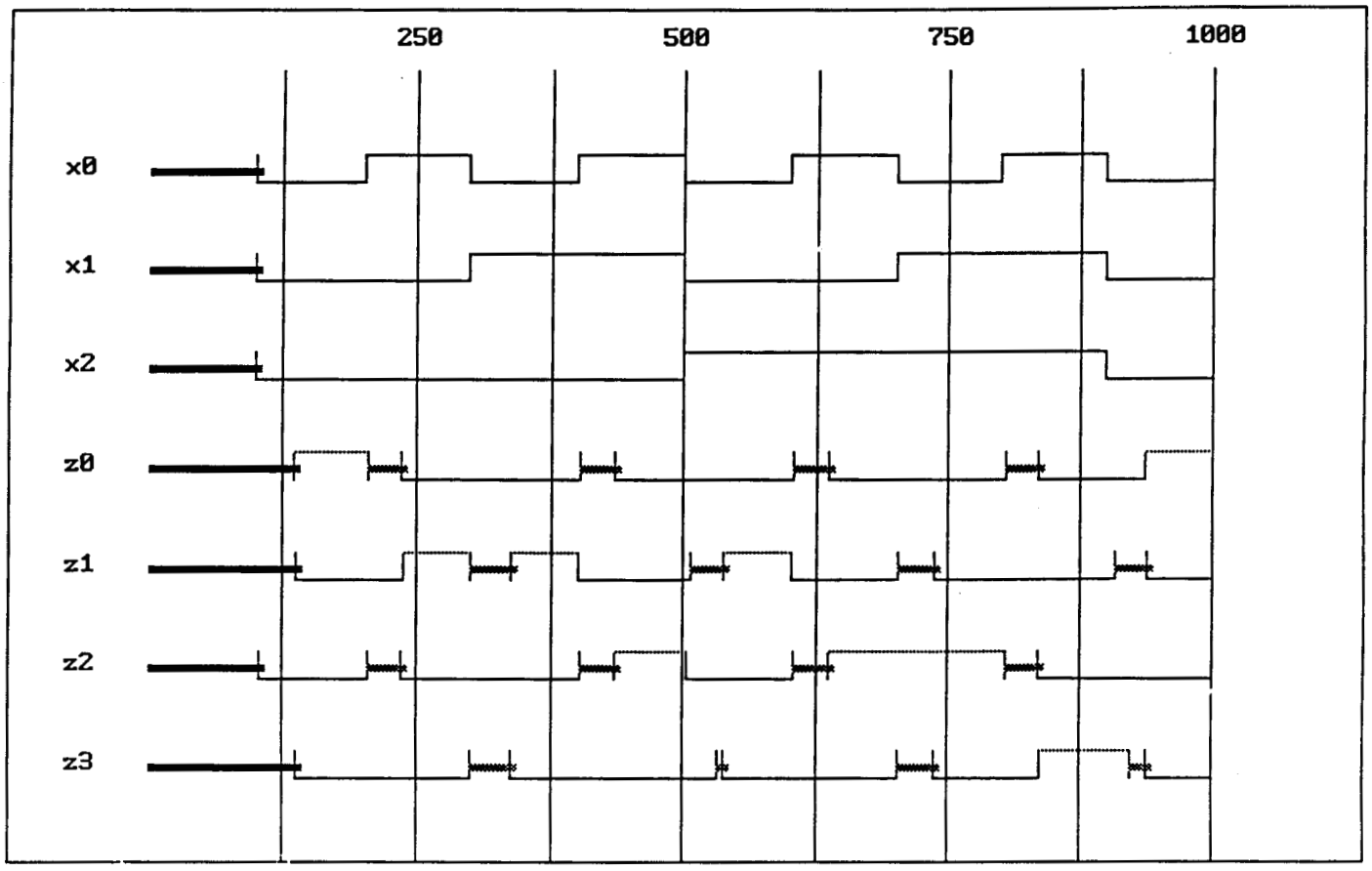


fig 33.b Chronogramme de simulation du réseau de Tally
(temps de simulation : 55s)

Pour la simulation des effets liés aux capacités des transistors MOS, une primitive supplémentaire PointMémoire est proposée. Elle permet simplement d'introduire un retard dans la prise en compte d'un changement d'état de l'entrée. On arrive ainsi à simuler des circuits dynamiques, tels que les registres à décalage dont la structure est représentée ci-après.

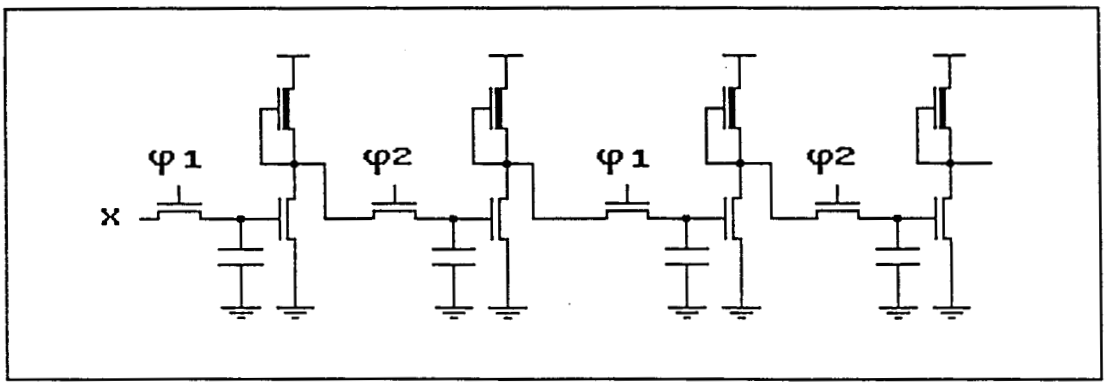


fig.34 Registre à décalage dynamique

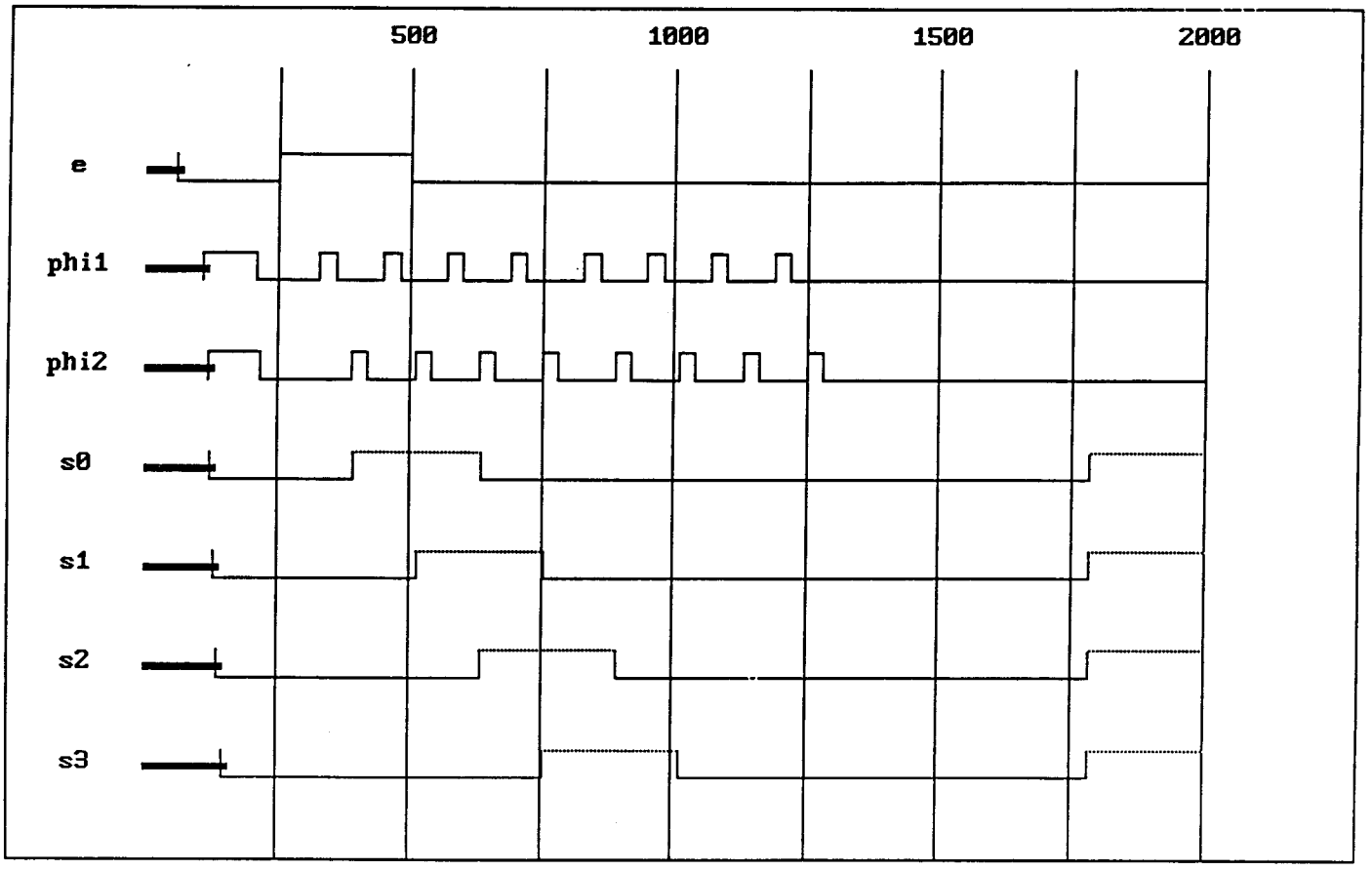


fig 34.b Chronogramme de simulation du registre à décalage
(temps de simulation : 2,75 s)

76.4.7. Objets de visualisation dynamique

Les composants électroniques ne sont que des objets parmi d'autres dans l'environnement Smalltalk. Rien ne nous empêche donc d'intégrer d'autres objets du réel dans le simulateur. En particulier, des objets tels que diodes électroluminescentes (LED), afficheurs 7 segments ou alphanumériques sont couramment employés comme éléments de visualisation des résultats de calcul d'un circuit numérique. Les capacités graphiques de Smalltalk permettent de créer de tels objets. Ainsi, un afficheur 7 segments sera un composant du circuit à part entière (inscrit dans la netlist).

Chaque réception d'un événement sur une de ses entrées provoque un affichage immédiat du nouvel état des segments. On peut donc assister en "temps réel simulé" au fonctionnement du circuit.

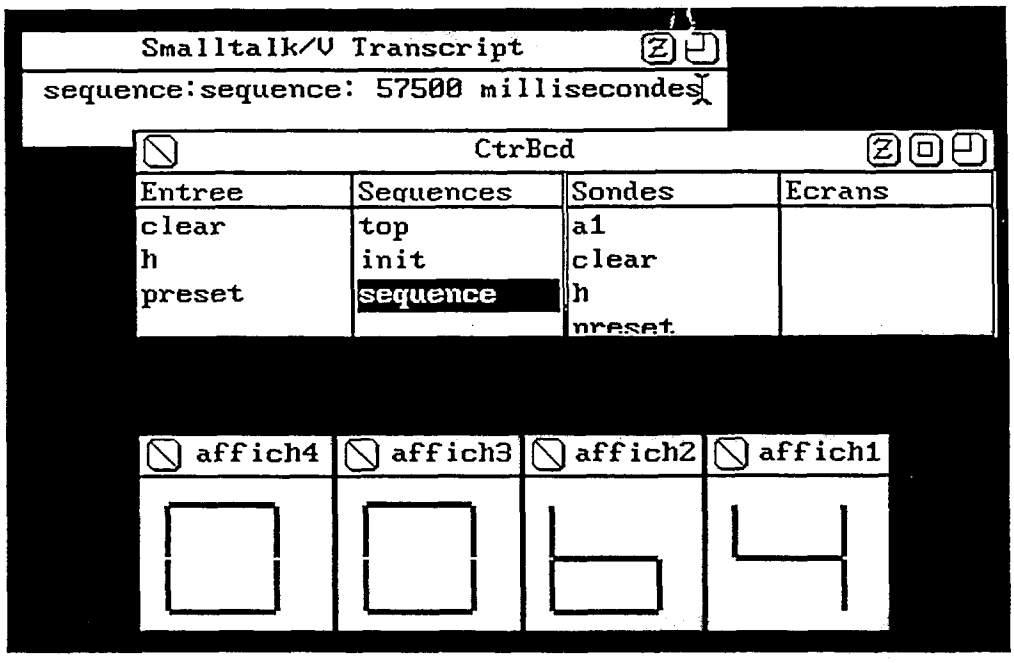


fig.35 Visualisation dynamique d'un compteur 16 bits

Cette possibilité est particulièrement spectaculaire, mais ralentit considérablement la simulation lorsque les changements visuels sont nombreux (pour un compteur à propagation par exemple, tous les états intermédiaires sont traduits sur l'afficheur). La figure 35 représente l'état des afficheurs après un comptage de 100 tops d'horloge (100 = 64H) sur un compteur 16 bits. Le temps de simulation (57 s) est indiqué dans la fenêtre Transcript.

6.5 Création de nouvelles primitives

Le simulateur est ouvert à la définition de nouvelles primitives, ce qui présente en premier avantage la possibilité d'accélérer les temps de simulation en remplaçant une simulation de composants élémentaires par un comportement directement codé en Smalltalk. Le second avantage est de

pouvoir simuler un circuit complet sans avoir forcément conçu chacun de ses composants jusqu'au niveau des composants élémentaires.

Les simulateurs classiques offrent des langages de description comportementale pour répondre à ces deux besoins. Ce sera ici le langage Smalltalk lui même qui jouera ce rôle, sans toutefois pénaliser l'utilisateur par un langage trop hermétique. En effet la définition de classes et de méthodes telles que EtatLogique ou MotBinaire, amènent à une description comparable aux traditionnels langages à transfert de registre. Il faut d'ailleurs se rappeler que la syntaxe Smalltalk elle-même est définie par des classes et méthodes standards.

Un exemple complet de description de primitive est donné en annexe. La majeure partie des méthodes à définir pourraient l'être de façon automatisée, en ne demandant à l'utilisateur que la partie de description de comportement du circuit.

6.6. Aspects temporels

Le simulateur gère des temps de montée et de descente pour chaque sortie en fonction de l'entrée ayant amené le changement d'état.

En pratique, les primitives sont réalisées pour utiliser un temps de montée et un temps de descente par groupe de sorties pour un groupe d'entrées donné. Ainsi, un multiplexeur demande 3 couples de temps de montée/descente par rapport aux entrées de données, d'adresses, et à l'entrée de validation.

Ces temps sont à fournir dans la netlist mais des valeurs par défaut sont systématiquement prévues. Une extension simple possible du simulateur pourrait faire varier ces temps par défaut en fonction de la technologie employée.

On trouve dans beaucoup de simulateurs des contrôles, tels que la durée d'impulsions ou le temps de stabilité de signaux avant et après un événement. La création de ces primitives temporelles serait

particulièrement simple dans la structure adoptée. En effet, ces primitives seraient très proches des objets de classe Sonde, avec une entrée (ou plusieurs pour des contrôles croisés sur plusieurs signaux), sans sortie, mémorisant les quelques événements précédents pour effectuer le contrôle requis et avertir l'utilisateur par un message adéquat en cas d'anomalie.

6.7. Classes génériques

La plupart des primitives décrites précédemment sont bâties sous forme de classes génériques. Une classe générique est une classe capable de générer automatiquement une classe de primitives, dont le nombre d'entrées peut être variable.

Ainsi, toutes les fonctions booléennes peuvent accepter un nombre d'entrées quelconque, seule la classe générique est définie, on crée sur besoin les classes des primitives correspondantes.

De même, les circuits de type mémoire peuvent varier en taille de données ou d'adresses.

Comme beaucoup de méthodes dépendent directement de la taille du problème, plutôt que de paramètrer ces méthodes pour les rendre générales, il est plus économique, du point de vue temps de simulation, de créer le code correspondant exactement à la dimension voulue.

D'autre part, comme chaque circuit possède une méthode par entrée, il aurait fallu fixer une taille maximale pour définir ces méthodes à l'avance, ce qui est ici évité.

Cette génération automatique de classes et de méthodes est faite de façon totalement transparente à l'utilisateur lors du chargement du circuit. On profite pleinement, là encore, des concepts de la programmation objet, ou compilateur, code binaire des méthodes, et texte source, sont des objets du système parfaitement accessibles à tout moment.

On remarquera que la mémorisation du texte source des méthodes générées est facultative, réduisant ainsi le temps de génération. Ceci est simplement assuré par détournement de la méthode d'enregistrement du compilateur, dans la classe principale Simul.

6.8. Conclusions

Le premier point intéressant de cette partie est la difficulté d'optimisation des performances qui sont la contrepartie naturelle de la richesse des concepts. Smalltalk offre un environnement riche d'une multitude de classes et de méthodes prédéfinies. Mais il est parfois nécessaire de redéfinir tout ou partie de ces classes pour obtenir un temps d'exécution acceptable.

Le style de programmation doit parfois être également alourdi pour gagner en rapidité, notamment dans toutes les structures itératives. On peut ainsi remarquer des performances intéressantes en récursion, mais des accès lents aux tableaux standards (classe Array). Quelques améliorations ont ainsi pu être apportées aux méthodes les plus sensibles.

On constate en second lieu l'interêt d'une homogénéité entre objets réels et objets Smalltalk. On aboutit ainsi, pour la description des primitives, à une expression proche des langages comportementaux ou des langages à transfert de registre. Toutefois, la simulation des transistors MOS prouve qu'un choix judicieux sur les objets effectivement implémentés conduit à des solutions élégantes.

En troisième point, la possibilité de définir des objets plus largement que les simples composants primitifs ouvre la porte à des simulations attractives et proches de la réalité. Ceci pourrait même être étendu à la définition d'objets analogiques par le biais d'objets convertisseurs, et aboutir à des possibilités de simulation mixte complète.

Ensuite, cette partie illustre la puissance de la structure objet + classe + héritage. De nombreuses classes d'objet peuvent être ajoutées en profitant au maximum des éléments déjà introduits dans le système.

L'héritage de méthodes, ou l'adaptation de méthodes existantes, autorisent la création simple de contrôles sophistiqués.

Finalement, la notion de classe générique permet de contrebalancer les besoins en mémoire d'un système tel que Smalltalk et ses limitations en performances. En effet, la création sur besoin de classes adaptées au circuit que l'on désire générer permet de ne garder en mémoire qu'un minimum de primitives, et d'intégrer sous forme compilée certaines caractéristiques du circuit.

7. Amélioration des performances

Cette dernière partie explore les possibilités qu'offre le concept même d'objet. Ainsi, le simulateur étant un objet, la simulation peut être conduite automatiquement par un objet de haut niveau. D'autre part, lorsqu'on donne à chaque objet les informations et les méthodes qui le concernent pour résoudre un problème donné, on obtient un style de programmation par parcours de la structure de données, où chaque objet contribue à son niveau à la réalisation de la fonction complète.

Les exemples qui suivent ont tous été construits dans le but d'améliorer les vitesses de simulation, par remontée fonctionnelle, c'est-à-dire par la création d'objets de plus haut niveau.

7.1. Signaux de haut niveau

La première solution vise à réduire le nombre d'évaluations de la sortie des composants, et le nombre de messages d'excitation qui transitent entre les circuits.

Dès que l'on monte dans la complexité des circuits, les informations deviennent très rapidement des mots binaires codés, plutôt que des simples signaux logiques.

Ainsi, les entrées d'adresse d'un circuit mémoire de 1M mots peuvent faire l'objet de 20 messages d'excitation portant chacun une information binaire, ou d'un seul message portant un message de type mot binaire de 20 bits.

La perte d'information correspondante est la synchronisation d'arrivée des bits du mot transporté. Ceci pose peu de problèmes pour deux raisons:

* la prise en compte de l'information se fait elle même en général de façon synchrone. Les états intermédiaires résultants d'une arrivée asynchrone des bits d'entrée sont très vite filtrés par les barrières temporelles des systèmes synchrones.

* l'émetteur de l'information génère souvent ces signaux en synchrone. La convention adoptée d'utiliser des temps de transit identiques par groupes de sortie en fonction de groupes d'entrée, favorise la propagation simultanée de tous les bits.

Seuls des cas anormaux de fonctionnement peuvent éventuellement être non détectés. L'utilisateur a la charge d'avoir auparavant vérifié le fonctionnement de son circuit par une simulation de niveau plus simple. Les fonctionnements anormaux peuvent être détectés à d'autres niveaux (par des contrôles temporels par exemple).

Si la décision d'utiliser de tels messages appartient à l'utilisateur, puisque la simulation s'en trouve modifiée, le mécanisme de regroupement des connexions se fait automatiquement. Chaque objet circuit regarde si ses groupes de sortie sont connectés identiquement sur un groupe d'entrée d'un autre circuit. Il déconnecte alors les n liaisons établies en standard à la construction du circuit et réalise une connexion unique, pour le transport du mot binaire complet.

Ceci oblige à doter de tels circuits d'une seconde liste de sorties, pour la mémorisation de ces connexions. La méthode de propagation est également modifiée. Elle agit sur les deux listes de sortie, puisque certaines liaisons bit à bit peuvent subsister.

Pour réaliser ce changement de type de connexion, chaque circuit peut, par une méthode adéquate, indiquer quels sont ses groupes d'entrées ou de sorties qui forment un groupe logique, vis-à-vis de son comportement. Chaque groupe d'entrée peut recevoir une excitation par un signal de haut niveau, une méthode de même nom est donc créée tout comme il existe une méthode d'excitation par entrée simple.

7.2. Primitives combinatoires type 1

La simulation d'un circuit combinatoire quelconque peut se faire facilement en connaissant sa table de vérité.

Un circuit chargé en mémoire peut être simulé pour constituer sa table de vérité, table qui sera utilisée pour les simulations de plus haut niveau de complexité.

A partir de la simulation, on génère donc une classe reprenant tous les aspects extérieurs du circuit combinatoire et gérant la table de vérité, en réponse aux modifications de ses entrées.

Le circuit combinatoire devient alors une primitive du simulateur à part entière, et il peut être utilisé comme composant dans d'autres circuits.

Le gain de temps de simulation est dû à la disparition du mécanisme de gestion d'événements entre tous les composants du circuits, et à des méthodes de calcul d'état qui se résument à un accès à la table de vérité.

Les aspects temporels sont repris dans un paragraphe suivant.

L'inconvénient majeur de cette méthode est la taille de cette table de vérité, établie sur les 3 états 0,1 et X, qui augmente de façon exponentielle avec le nombre d'entrées. Une limite apparaît naturellement pour les circuits à 10 entrées, au-delà desquelles la table de vérité contient plus de 64k éléments (3^{10}=59049)

Il faut alors trouver une solution différente pour transformer le circuit en primitive.

On notera que la définition d'une primitive unique permet, de plus, le regroupement des sorties en signaux de haut niveau comme vu précédemment.

Le gain de temps de simulation obtenu par cette méthode est environ un facteur 2. Ce facteur dépend de la complexité du circuit, notamment en terme de nombre de couches logiques traversées entre les entrées et les sorties.

7.3. Primitives combinatoires type 2

Une seconde solution à l'amélioration des performances vise à supprimer simplement la boucle de propagation d'événement, en gérant cette propagation à temps de transit nul.

En effet, tant que l'on reste en logique combinatoire, on peut reporter les contraintes temporelles sur les seules sorties du circuit.

Pour cela, on génère également une primitive, qui fera un calcul des sorties par évaluation successive des fonctions logiques rencontrées. Chaque noeud est mémorisé dans une variable interne. Tout circuit est capable d'évaluer ses sorties pour une combinaison d'entrées données.

Un événement d'entrée est donc propagé vers toutes les sorties auxquelles il est relié. Lorsque la propagation est terminée (stabilisation des états), on peut propager l'état des sorties vers les primitives suivantes, par le biais normal du simulateur. La simulation du fonctionnement de cette primitive est donc du type "zero delay", la fonction temps est réintroduite en sortie du composant. Ceci n'est bien sûr valable que pour des circuits purement combinatoires. Les problèmes temporels sont identiques à ceux de la première méthode vue au paragraphe précédent, et sont traités dans le paragraphe suivant.

La génération de la classe et des méthodes correspondantes se fait en parcourant les différentes branches du circuit chargé en mémoire et en demandant à chaque primitive les initialisations, variables d'entrée et mode de calcul des sorties, qu'elle aurait dû réaliser lors d'une simulation.

Pour cela, des méthodes particulières ont été définies pour délivrer le comportement des primitives, afin de les regrouper en une primitive unique.

Le gain obtenu, environ 20% du temps de simulation, est moins intéressant que pour la première méthode mais n'est pas limité en nombre d'entrée.

7.4. Aspects temporels

Le regroupement en une primitive unique d'un réseau combinatoire complexe va forcément impliquer une perte dans les informations temporelles contenues dans le circuit.

En effet, dans le cas de primitive de type 1, il faudrait mener une simulation comportant toutes les transitions possibles sur chaque entrée étant donné tous les états possibles sur les autres entrées, pour avoir une information complète sur les temps de transit.

Il faut donc adopter un compromis qui ne trahisse pas trop le comportement du circuit en simulation de plus haut niveau.

De multiples solutions sont possibles parmi lesquelles a été choisie, parce que pertinente, la solution suivante.

Chaque sortie se verra affecter d'un temps de transit relatif à l'entrée qui a été modifiée. Ce temps est calculé en prenant le temps de la plus longue branche reliant la sortie à l'entrée, temps de montée et temps de descente confondus.

On garantit ainsi un temps d'établissement des sorties qui sera systématiquement plus long que le temps réel.

Les chronogrammes de la figure suivante montrent la proximité des temps obtenus par simulation complète du circuit, ou par simulation d'une primitive de type 1 générée à partir de ce circuit. Le circuit est un décodeur BCD-7 segments comportant 32 portes.

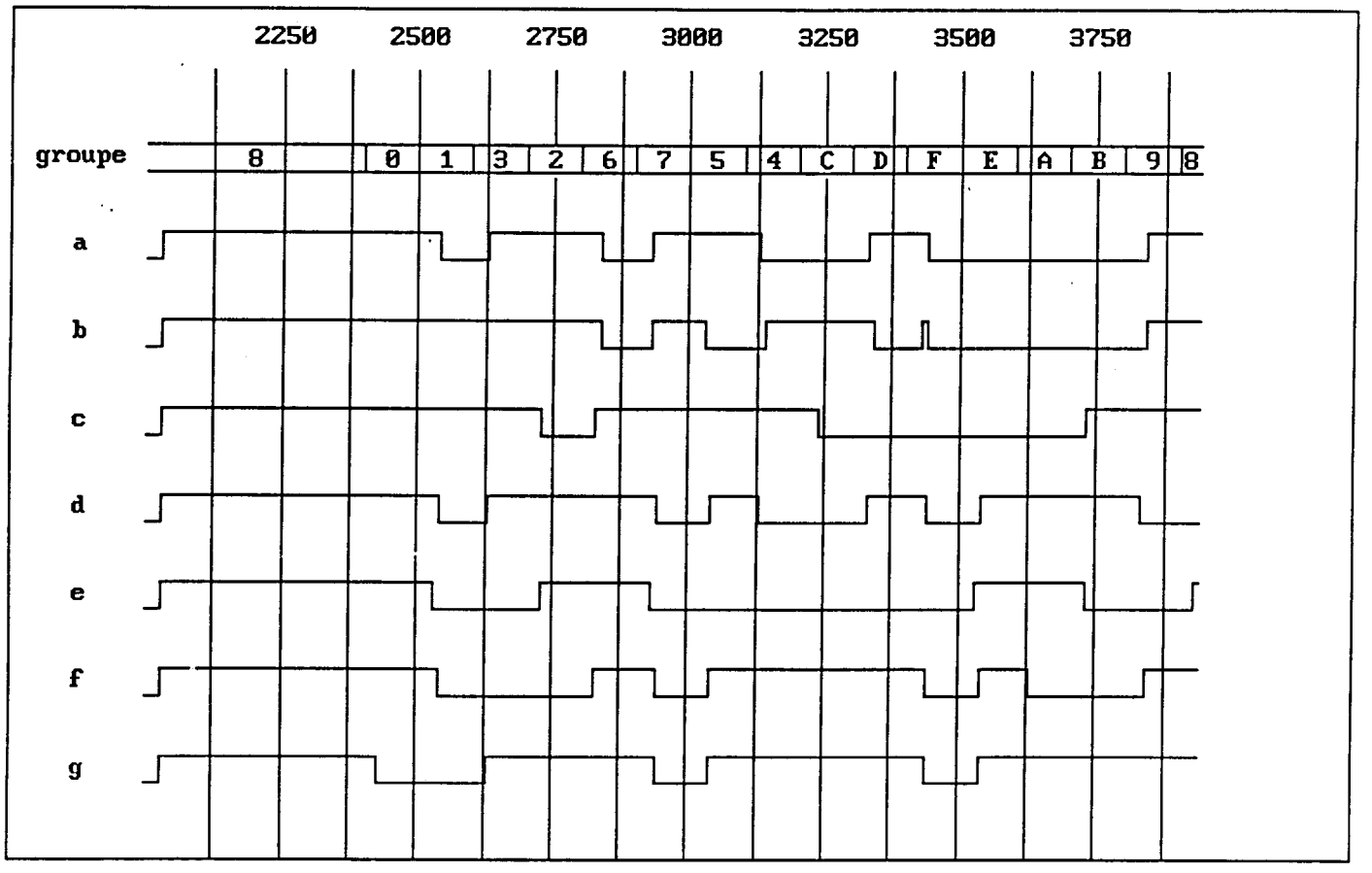


fig.36.a Chronogramme de simulation d'un décodeur 7 segments simulation normale

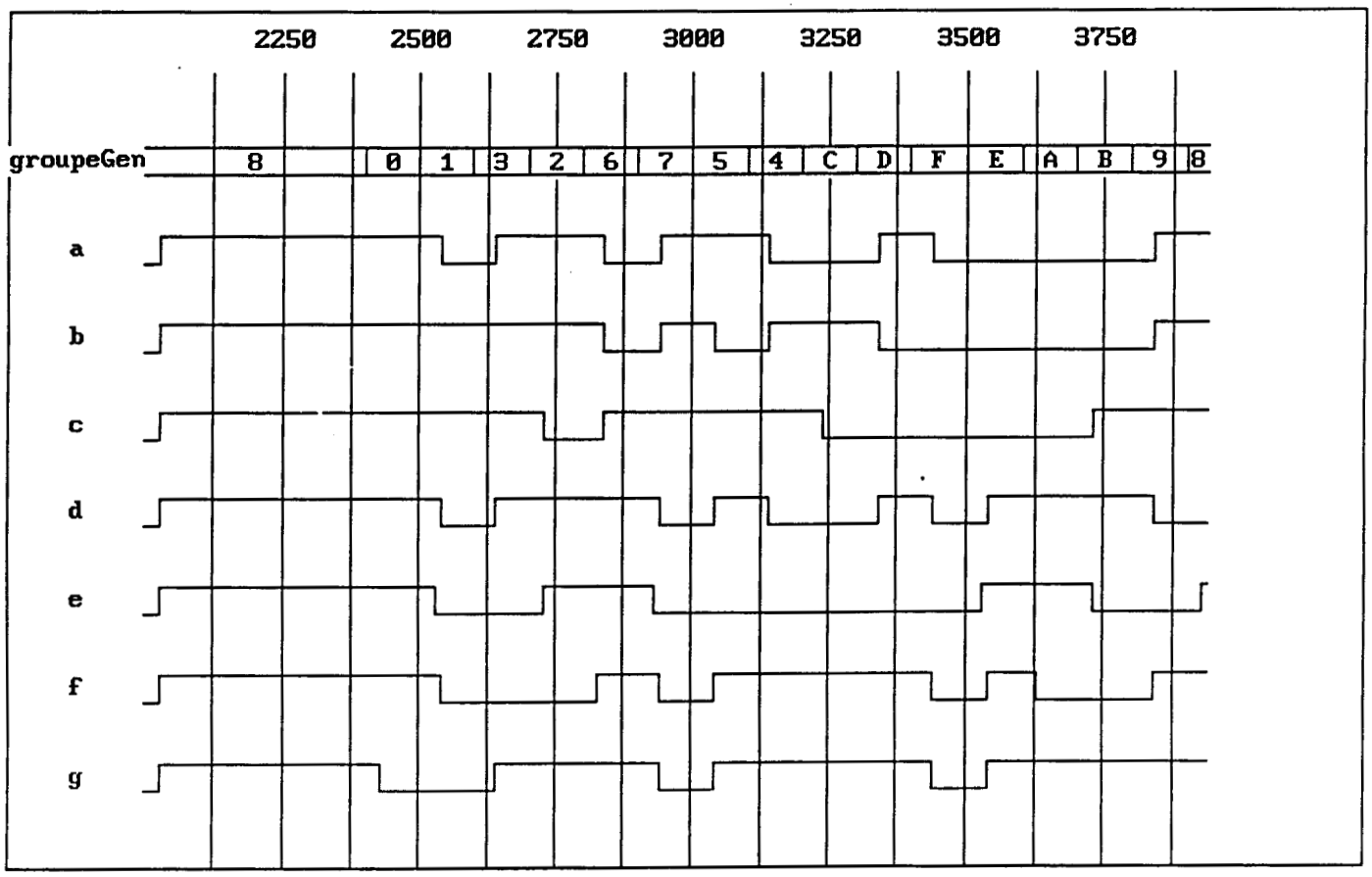


fig.36.b Chronogramme de simulation d'un décodeur 7 segments
simulation de la primitive générée

CONCLUSION

Le bilan de l'approche objet de la CAO présentée dans cette étude peut se faire du point de vue des deux acteurs de la CAO : le concepteur et l'utilisateur.

Pour le concepteur, le bilan est positif. Les qualités "génie logiciel" de Smalltalk, ses nombreux outils d'aide au développement, la rapidité de maquettage de nouvelles fonctions en font un langage propre à supporter le développement de logiciels de grande taille, avec une productivité inégalée.

Pour l'utilisateur, la qualité de l'interface graphique offre un confort de travail élevé. Mais au delà du simple aspect externe, la cohérence des objets manipulés avec le monde réel permet d'offrir à l'utilisateur un outil adaptable à sa propre vision du circuit à concevoir, contrairement aux couches multifenêtrage graphique, probablement inspirées des techniques objets, appliquées sur des logiciels de conception traditionnelle, où l'utilisateur doit se plier à la vision du circuit qu'on lui propose.

L'utilisateur bénéficie aussi de fonctionnalités nouvelles et puissantes grâce au traitement symbolique et à l'ouverture sur des systèmes intelligents.

Mais ce sera également l'utilisateur qui subira des temps de réponse élevés, dus à une mauvaise adéquation du langage objet à l'architecture de la machine.

De plus, l'environnement Smalltalk utilisé, pourrait être amélioré sur divers plans. Les aspects graphiques en particulier se révèlent limités par une sous-utilisation des possibilités du matériel graphique.

Une meilleure intégration aux environnements et langages traditionnels serait également appréciable, pour conserver un éventuel existant, et pour relayer la programmation objet lorsque les performances sont critiques.

Mais il faut malgré tout garder le concept d'objet comme base fondamentale et incontournable de toutes les applications, pour bénéficier des avantages réels d'une CAO au sens Conception Assistée par Objets.

BIBLIOGRAPHIE

[Ama85] P.Amadou et al.
"A switch level simulator based
on an efficient analysis of conducting paths"
abstracts ESSIRC 85
pp 286-290

[Bar86] D.S.Barclay, J.R.Armstrong
"A heuristic chip-level test generation algorithm"
proc 23rd Design Automation Conference,1986
pp 257-262

[Beg84] V.Begg
"Eléments d'introduction des systèmes experts en CAO"
Hermes publishing,1984

[Bor83] D.Borrione, M.Humbert, C.Le Faou
"Hierarchical mixed-mode simulation mechanisms
in the Cascade project "
VLSI design of digital systems, North Holland, 1983
pp 119-129

[Bre85] M.A.Breuer, X.Zhu
"A knowledge based system
for selecting a test methodology for a PLA"
22nd Design Automation Conference 1985
pp 259-265

[Bre86] F.D.Brewer, D.D.Gajski
"An expert-system paradigm for design data"
proc 23rd Design Automation Conference,1986
pp 62-67

[Cha88] J.J.Chabrier
"Sur les origines de PROLOG"
Génie logiciel et systèmes experts,n 12 juin 1988
pp 1-6

[Coi82] P.Cointe
"Une réalisation de Smalltalk en VLISP"
TSI vol 1 n 4, 1982
pp 325-340

[Cox87] B.J.Cox, K.J.Schmucker
"Producer : a tool for translating
Smalltalk 80 to Objective C"
ACM Sigplan Notices, vol 22, Dec 1987
pp 423-429

[Dig88] "Smalltalk/V286 : Tutorial and programming handbook"
manuel de référence,Digitalk Inc.

[Eic83] E.B.Eichelberger, E.Lindbloom
"Trends in VLSI testing"
VLSI design of digital systems
North Holland, 1983
pp 339-348

[Erc85] M.D.Ercegovac, T.Lang
"Digital systems and hardware/firmware algorithms"
John Wiley & Sons, 1985

[Eur86] J.P.Eurich
"A tutorial introduction
to the electronic design interchange format"
proc 23rd Design Automation Conference,1986
pp 327-333

[Gaz89] P.Galzin
"SGBD : le relationnel talonné par l'objet"
Le Monde Informatique,Juillet 1989
pp 25-26

[Gen85] "HILO-3 User manual"
GenRad Inc

[Joo85] R.Joobbani, D.P.Siewiorek
"WEAVER : a knowledge based routing expert"
22nd Design Automation conference 1985
pp 266-272

[Kna86] D.W.Knapp, A.C.Parker
"A design utility manager : the ADAM planning engine"
proc 23rd Design Automation Conference,1986
pp 48-54

[Kno81] H.J.Knobloch
"Description and simulation of complex digital systems
by means of the register transfer language RTS Ia"
Computer Design Aids for VLSI Circuits, Nato advanced study institutes
series, 1981
pp 285-320

[Kow84] T.J.Kowalski
"The VLSI design automation assistant :
a knowledge based expert system"
PhD thesis, Carnegie Mellon University,1984

[Kow85] T.J.Kowalski, D.E.Thomas
"The VLSI design automation assistant :
what's in a knowledge base"
22nd Design Automation conference 1985
pp 252-258

[Lau86] J.L.Lauriere
"Intelligence artificielle :
résolution de problèmes par l'homme et par la machine"
Eyrolles,1986

[Lea86] R.M.Lea
"VLSI parallel-processing chip architecture"
Algorithmics for VLSI, Academic press, 1986
pp 1-31

[Lew83] K.D.Lewke, F.J.Rammig
"Description and simulation of MOS devices
in register transfer languages"
VLSI design of digital systems, North Holland, 1983
pp 73-84

[Lin86] T.M.Lin, C.A.Mead
"A hierarchical timing simulation model"
IEEE trans. on CAD, vol CAD-5 n 1 january 1986
pp 188-197

[Man81] H.De Man, G.Arnout, P.Reynaert
"Mixed-mode circuit simulation technics
and their implementation in DIANA"
Computer Design Aids for VLSI Circuits
Nato advanced study institutes series, 1981
pp 113-174

[Mar88] M.Marchesi et al.
"Object oriented programming for VLSI CAD tool integration"
Tool integration and design environments
Elsevier Sciences Publishers, North Holland, 1988
pp 85-103

[Mea80] C.Mead, L.Conway
"Introduction to VLSI systems"
Addison-Wesley,Reading,1980

[Mer81] J.Mermet
"Functional Simulation"
Computer Design Aids for VLSI Circuits
Nato advanced study institutes series, 1981
pp 321-358

[Mur84] S.Muroga
"Logic Design of VLSI Electronic Circuit"
VLSI Algorithm and Architecture
North Holland, 1984
pp 277-299

[New81] A.R.Newton
"Timing, logic and mixed-mode simulation
for large MOS integrated circuits"
Computer Design Aids for VLSI Circuits
Nato advanced study institutes series, 1981
pp 175-240

[Noj86] T.Nojiri, S.Kawasaki, K.Sakoda
"Microprogrammable processor
for object-oriented architecture"
proceedings of the 13th Annual International Symposium on Computer
Architecture,1986
pp 74-81

[Ped81] D.O.Pederson
"Computer aids in integrated circuits design"
Computer Design Aids for VLSI Circuits
Nato advanced study institutes series, 1981
pp 1-17

[Roc89] C.Roche, J.P.Laurent
"Les approches objets et le langage LR02"
TSI vol 8 n 1 1989
pp21-39

[Rou89] J.P.Roux, G.Gardarin
"Les bases de données orientées objet"
01 Informatique, hors série Références, Juin 89
pp82-83

[Sak83] H.Sakuma
"NELSIM : a hierarchical VLSI design verification system "
VLSI design of digital systems, North Holland, 1983
pp 109-118

[Sau86] G.Saucier et al.
"ASYL : a rule based synthesis tool"
Avignon 1986
pp 1459-1485

[Sch83] H.G.Schwaertel
"Testing of VLSI"
VLSI design of digital systems, North Holland, 1983
pp 21-33

[Sch87] A.F.Schwarz
"Computer aided design of microelectronic circuits
and systems"
Academic Press, 1987

[Sha86] M.Shahdad
"An overview of VHDL language and technology"
proc 23rd Design Automation Conference,1986
pp 320-326

[Shr83] H.E.Shrobe
"AI meets CAD"
VLSI design of digital systems
North Holland, 1983
pp 387-399

[Smi86] R.J.Smith
"Fundamentals on parallel logic simulation"
proc 23rd Design Automation Conference,1986
pp 2-12

[Sri88] N.C.E.Srinivas, V.D.Agrawal
"Formal verification of digital circuits
using hybrid simulation"
IEEE circuits and devices magazine, January 1988
pp 19-27

[Sug86] A.Sugimoto et al.
"An object oriented visual simulator
for microprogramm development"
proc 23rd Design Automation Conference,1986
pp 138-144

[Van87] P.S.VanDerMeulen
"INSIST : Interactive simulation in Smalltalk"
proc. OSSPLA October 1987
pp 366-376

[Van89] P.S.VanDerMeulen
"Development of an interactive simulator in Smalltalk"
JOOP, January-February 1989
pp 28-44

[Wat86] T.Watanabe et al.
"Knowledge based optimal I2L circuit generator
from conventional logic circuit descriptions"
proc 23rd Design Automation Conference,1986
pp 608-614

[Wax89] R.Waxman, L.Saunders, H.Carter
"Abolishing the tower of Babel"
Spectrum vol.26 n 5, may 1989

[Wil81] T.W. Williams
"Design for testability"
Computer Design Aids for VLSI Circuits, Nato advanced study institutes
series, 1981
pp 359-416

[Win81] P.Winston, R.Horn
"LISP"
Addison-Wesley, Reading, 1981

[Win84] P.Winston
"Artificial intelligence"
2nd edition,Addison-Wesley,Reading,1984

[Wol86] W.Wolf
"An object oriented procedural database
for VLSI chip planning"
proc 23rd Design Automation Conference,1986
pp 744-751

ANNEXE 1

Exemple de codage d'une primitive

Le circuit que nous allons prendre en exemple est un comparateur de deux mots de 2 bits A et B. Nous verrons que l'extension a N bits se déduit directement de cet exemple. Le comparateur sera synchronisé sur un front montant d'une horloge h, pour sortir les 2 mots de 2 bits min et max.

1) Variables d'instances

Nous conviendrons de mémoriser l'état de l'horloge dans une variable h, les deux mots en entrée dans deux variables a et b, et nous supposerons que le circuit ne contient qu'un seul temps de propagation, en montée et en descente (les sorties sont équivalentes et ne dépendent que du signal d'horloge). On hérite de la classe CircLog ces deux variables de temps de propagation (tUp,tDown).

Les variables à déclarer seront donc : 'a b h'

2) Méthodes générales

Quatre méthodes sont à prévoir, pour une bonne intégration du circuit au simulateur.

Les deux premières indiquent les noms des entrées et des sorties :

```
nomsDesEntrées ^#(a0: a1: b0: b1: h:)
nomsDesSorties ^#(min0 min1 max0 max1)
```

Les deux dernières servent à initialiser les variables d'instances :
initialiseEntrees

```
" toutes les entrées seront initialisées à XX "
h:=XX. " h signal logique simple "
a:=MotBinaire newX:2 " a mot binaire 2 bits "
b:=MotBinaire newX:2 " b mot binaire 2 bits "
```

```
initialiseSorties:liste
        tUp:=tDown:=25. " temps par défaut "
        liste size=1 ifTrue:[ " cas ou 1 paramètre "
                tUp:=tDown:=liste at:1]
        liste size=2 ifTrue:[ " cas ou 2 paramètres "
                tUp:=liste at:1.
                tDown:=liste at:2]
        liste size > 2 ifTrue:[ " autres cas "
         self erreur: ' trop de paramètres : ',self nom.]
```

3) Méthodes d'excitation

Chaque entrée doit correspondre à une méthode d'excitation. Pour les entrées a et b, ces méthodes ont toutes le même modèle :

```
a0:uneValeur a at:1 put:uneValeur.
```

Seule l'entrée d'horloge va provoquer un changement possible sur les sorties. Il faut examiner toutes les transitions possibles de l'ancienne valeur vers la nouvelle. Les transitions 0-X X-1 ou X-X sont des fronts montants éventuels, où l'on choisira de renvoyer en sortie deux mots indéterminés (chaque bits à X). Cette méthode est valable dans toutes les primitives comportant une entrée horloge sur front.

```
h:uneValeur
        |hOld|
        hOld:=h.
        h:=uneValeur.
        h est0 ifTrue:[^#inchange]
        h est1 ifTrue:[
                  " de 1 vers 1 pas de changement "
                hOld est1 ifTrue:[^#inchange].
                  " de 0 vers 1 front montant "
                hOld est0 ifTrue:[^self propage:self etat].
                  " de X vers 1 front montant possible "
                ^self propage:(MotBinaire newX:4)]]
        h estX ifTrue:[
                  " de 1 vers X pas de front montant "
                hOld est1 ifTrue:[^#inchange].
                  " de 0 vers X ou de X vers X front possible "
                ^self propage:(MotBinaire newX:4]
```

Deux méthodes apparaissent dans la précédente, la première (propage) a pour but d'ajouter à la liste des événements le nouvel état de la sortie, la seconde (valeur) calcule l'état des 4 bits de sortie.

```
valeur
      |x y|
x:=a valeurDecimale.
y:=b valeurDecimale.
   " cas ou un des deux mots a des bits à X "
(x isNil | y isNil) ifTrue:[^MotBinaire newX:4]
   " concaténation (opérateur ,) du max suivi du min "
^ x>y ifTrue:[a,b] ifFalse:[b,a].

propage:unVecteurSortie
      Simulateur reveilleListe:listeDesSorties
                        a:unVecteurSortie
                      up:tUp
                    down:tDown
```

4) Méthodes pour groupement des entrées

Ce circuit travaillant sur des mots, on peut envisager l'utilisation de signaux de haut niveau. Il faut alors le doter de 4 méthodes supplémentaires pour qu'il puisse se connecter correctement.

```
groupesEntrées ^#( (a0: a1:) (b0: b1:) )
groupesSorties ^#( (1 2 3 4) )
nomsDesGroupesEntrées ^#( a: b:)
a:uneValeur a:=uneValeur
```

Il faut également modifier la méthode de propagation pour propager les éventuels signaux de haut niveau à ses successeurs stockés dans la liste des sorties 2:

```
propage:unVecteurSortie
        Simulateur reveilleListe:listeDesSorties
                                  a:unVecteurSortie
                                 up:tUp
                               down:tDown
        Simulateur reveille:listeDesSorties2
                                  a:unVecteurSortie
                                 up:tUp
                               down:tDown
```

ANNEXE 2

Liste des classes Smalltalk du simulateur

```
Simul (sous classe d'Object)
   AnalyseurDeTopologie
      GenerateurDeCircuit
      GenerateurDeModele
   CircLog
      Afficheur
         Afficheur7Segment
         AfficheurLED
      Memoire
         Ram
         Rom
      Modele
      PrimitiveCombinatoire
      Standards
         Bascules
            BasculeD
            BasculeJK
         Boole
            And
            Delay
            Nand
            Not
            Nor
            Or
            Xor
         DeMux
         Mux
         Reg
      Switches
         Chemin
         Noeuds
         PointMemoire
      UneEntree
         Plot
         Sonde
   GenerateurDePrimitive
      PrimitiveCombinatoire1
      PrimitiveCombinatoire2
   Simulation
      Circuit
      ContenuEcran
      ContenuNoeuds
      EcranG
      EtatLogique
         ZeroLogique
         UnLogique
         PullUp
         PullDown
         Indetermine
         HauteImpedance
      GroupeEntree
      GroupeSortie
      Sequence
      Simulation
```

```
   Structures
      SousCircuit
      SousCircuit2
      MotBinaire
      Excitation
      EltDeCnx
      EltTrie
      ListeTriee
      EltDeReveil
      ObjetNom
      Changement
   Traducteur
ModeleBrowser


PenAjout (sous-classe de Pen)
   BrocheView
      Editeur
      ModuleView
         InterditView
         JonctionView
         PlotView
         PMView
   FilView
      Busview


Module (sous-classe de Object)
   Jonction
      Plot
         PlotEntree
         PlotSortie
      Puissance
         Masse
         Plus
         Alimentation
         PullDown
         PullUp
   ModuleInterdit


ModelePrincipal (sous classe de Object)
   FormeModPrinc


Broche (sous classe de Object)
FeuilleModele (sous classe de Object)
NetList (sous classe de Object)
Noeud (sous-classe de Object)
Evenement (sous classe de InputEvent)
NetListPane (sous classe de GraphPane)
RectangleForma (sous classe de Rectangle)
```

Liste des classes PROLOG

Analyseur
- ElementsSyntaxiques
- GroupesSyntaxiques
  - AnalyseurHILO
  - AnalyseurSimul

Liste des classes Smalltalk standards ayant été modifiées

Form
Collection

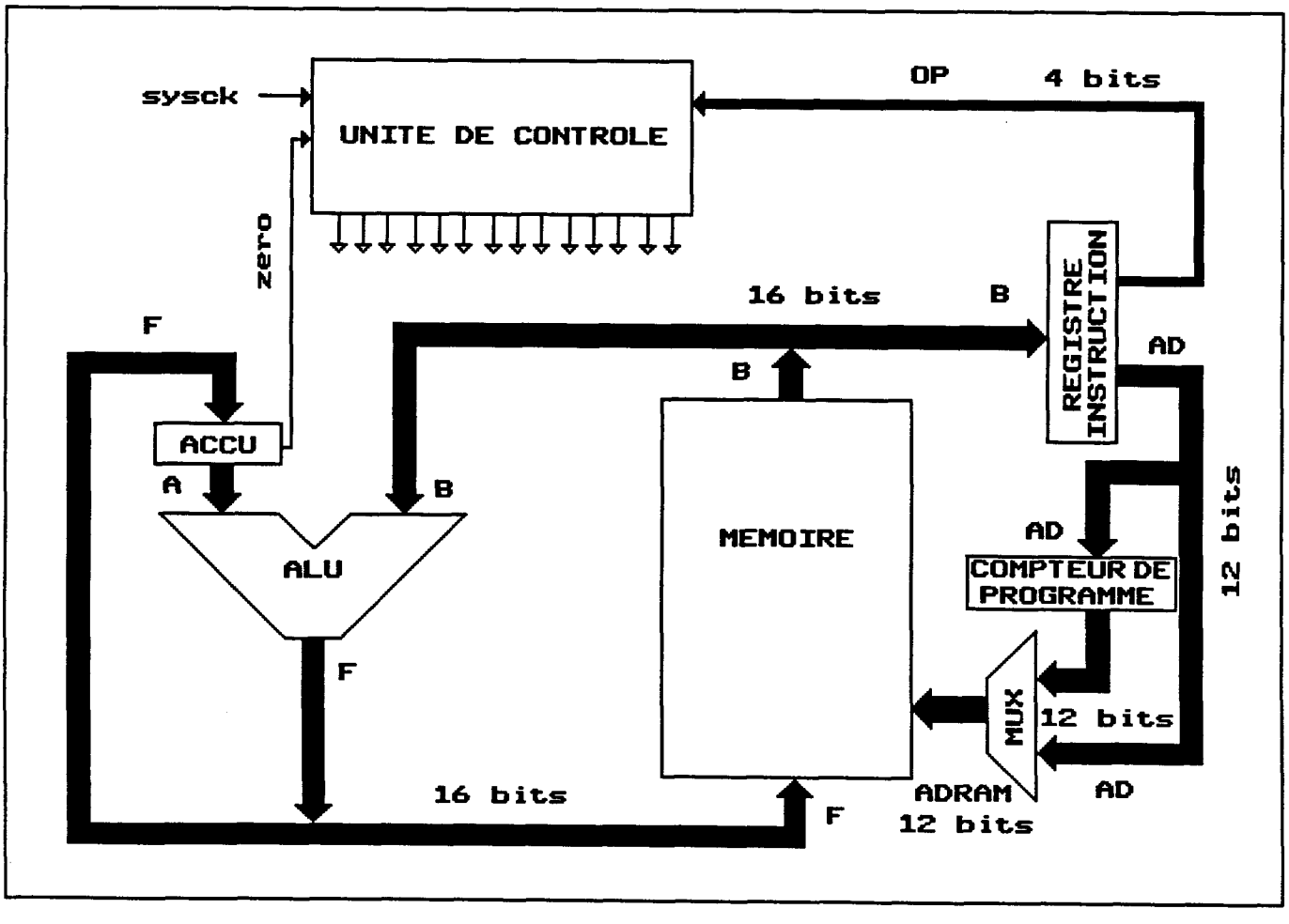
ANNEXE 3

Exemple de simulation de circuit complexe : le processeur TOY

Le processeur TOY est un processeur fictif, dont le schéma est donné à la figure ci-dessous. Les chemins de données sont sur 16 bits, les adresses sur 12 bits. L'unité arithmétique et logique est construite à partir du circuit 74181. Les instructions sont codées sur 16 bits, dont 4 bits de code opération et 12 d'adresse.

La simulation utilise une description détaillée des parties ALU et Unité de Contrôle à partir des portes logiques booléennes, les registres et mémoires utilisent les primitives évoluées du simulateur. Au total le circuit comprend 318 primitives.

Les fichiers de description netlist du processeur sont donnés dans les pages suivantes, ainsi que 2 chronogrammes résultant de l'execution d'un programme de test de lecture mémoire, incrémentation de l'accumulateur, écriture mémoire et remise à zéro de l'accumulateur. Ce programme se déroule sur 8 tops d'horloge (9,45 s de simulation) après l'initialisation (2,2 s de simulation). Le premier chronogramme montre l'évolution des signaux de contrôles du processeur, le second donne l'évolution du contenu des différents registres.



Fichiers de description du mini-calculateur TOY

1) Structure principale du calculateur

```
Toy(zero,un,reset1,reset2,sysck-ad0@11,a0@15)

unitArithmLog(a0@15,b0@15,alucb,alum,alu0@3-
                                f0@15,_,_,_,_):Alu16()

uc(ck1,ck2,op0@3-
           alu0@3,alum,aluc,ckac,ckir,adir,ckpc,incpc,wr):Seq()

ir(ckirsys,un,reset1,zero,zero,b0@15-ad0@11,op0@3):Reg/16()

pc(ckincpcsys,reset2,un,incpc,zero,ad0@11,zero,zero,zero,zero-
                                pcout0@11,_,_,_,_):Reg/16()

acc(ckacsys,reset2,un,zero,zero,f0@15-a0@15):Reg/16()

memram(wrck,un,adram0@7,f0@15-b0@15):Ram/8/16(1)

mux(un,pcout0@11,zero,zero,zero,zero,ad0@11,zero,zero,zero,zero
                          ,adir-adram0@7,_,_,_,_,_,_,_,_):Mux2d16()

net1(wr,sysck,reset1-wrck):Nand/3()
net2(ckincpc,sysck,reset1-ckincpcsys):And/3()
net3(ckir,sysck,reset1-ckirsys):And/3()
net4(ckac,sysck,reset1-ckacsys):And/3()
ou1(incpc,ckpc-ckincpc):Or/2()
non1(aluc-alucb):Not()
b(sysck,incpc,reset1,un-ck1,ck2):BasculeD()

% demo fournie:
% 1004 : load accu mem ad 0004
% 9000 : inc accu
% 0005 : store accu mem ad 0005
% B000 : raz accu
%
% table des code op:
%  0 store  1 load  2 jmpz  3 add
%  4 sub    5 or    6 and   7 xor
%  8 not    9 inc  10 dec  11 zero
% 12 nop   13 nop  14 nop  15 nop
```

2) Description du séquenceur

```
Seq(ck1,ck2,op0@3-alu0@3,alum,aluc,ckac,ckir,adir,ckpc,incpc,wr)
% sequenceur du processeur TOY
n1(op0-op0b):Not(1)
n2(op1-op1b):Not(1)
n3(op2-op2b):Not(1)
n4(op3-op3b):Not(1)

et1(op0b,op2b,op3b-alu01):And/3(1)
et2(op1,op2b-alu02):And/2(1)
et3(op0b,op1-alu03):And/2(1)
ou1(alu01,alu02,alu03-alu0):Or/3(1)

et4(op3b,op1b-alu11):And/2(1)
et5(op2-alu12):And/1(1)
et6(op1,op3-alu13):And/2(1)
ou2(alu11,alu12,alu13-alu1):Or/3(1)

et7(op3b,op1b,op0b-alu21):And/3(1)
et8(op2,op0-alu22):And/2(1)
et9(op3,op1,op0b-alu23):And/3(1)
ou3(alu21,alu22,alu23-alu2):Or/3(1)

et101(op3b,op2b-alu31):And/2(1)
et102(op3b,op1b,op0-alu32):And/3(1)
et103(op1,op0b-alu33):And/2(1)
ou10(alu31,alu32,alu33-alu3):Or/3(1)

et10(op0b,op1b,op2b-alum1):And/3(1)
et11(op3b,op1b,op0-alum2):And/3(1)
et12(op2,op1-alum3):And/2(1)
et13(op3,op1,op0-alum4):And/3(1)
ou4(alum1,alum2,alum3,alum4-alum):Or/4(1)

et14(op3,op2b-ckac1):And/2(1)
et15(op3b,op2-ckac2):And/2(1)
et16(op3b,op0-ckac3):And/2(1)
et17(ckac123,ck1-ckac):And/2(1)
ou5(ckac1,ckac2,ckac3-ckac123):Or/3(1)

et18(op1b-aluc):And/1(1)

et19(op3,ck1-ckir1):And/2(1)
et20(ck2-ckir2):And/1(1)
ou6(ckir1,ckir2-ckir):Or/2(1)

et21(op3b,ck1-adir):And/2(1)

et22(op3b,op2b,op1,op0b,ck1-ckpc):And/5(1)

et24(op3,ck1-incpc1):And/2(1)
ou7(incpc1,ck2-incpc):Or/2(1)

et25(op0b,op1b,op2b,op3b,ck1-wr):And/5(1)
```

3) Unité arithmétique et logique 16 bits

```
Alu16(a0@15,b0@15,cin,m,s0@3-f0@15,cout,eq,g,p)
% Alu 2*16 bits constituees a partir de l'alu74181
alu1(a0@3,b0@3,cin,m,s0@3-f0@3,cout1,_,_,_):Alu74181()
alu2(a4@7,b4@7,cout1,m,s0@3-f4@7,cout2,_,_,_):Alu74181()
alu3(a8@11,b8@11,cout2,m,s0@3-f8@11,cout3,_,_,_):Alu74181()
alu4(a12@15,b12@15,cout3,m,s0@3-f12@15,cout,g,p,eq):Alu74181()
```

4) ALU 4 bits 74181

```
Alu74181(x0@3,y0@3,cin,m-f0@3,cout,g,p,eq)
% ALU 74181 schema du data-book NS p.6-143
%
al1(s0@3,a0,b0-x0,y0):SousAlu()
al2(s0@3,a1,b1-x1,y1):SousAlu()
al3(s0@3,a2,b2-x2,y2):SousAlu()
al4(s0@3,a3,b3-x3,y3):SousAlu()
%
et1(y3-m11):Delay(1)
et2(x3,y2-m12):And/2(1)
et3(x3,x2,y1-m13):And/3(1)
et4(x1,x2,x3,y0-m14):And/4(1)
nor1(m11,m12,m13,m14-g):Nor/4(1)
%
et5(cin,x2,x1,x0,mb-m21):And/5(1)
et6(x1,x2,y0,mb-m22):And/4(1)
et7(x2,y1,mb-m23):And/3(1)
et8(y2,mb-m24):And/2(1)
nor2(m21,m22,m23,m24-k3):Nor/4(1)
%
et9(cin,x0,x1,mb-m31):And/4(1)
et10(x1,y0,mb-m32):And/3(1)
et11(mb,y1-m33):And/2(1)
nor3(m31,m32,m33-k2):Nor/3(1)
%
et12(mb,cin,x0-m41):And/3(1)
et13(mb,y0-m42):And/2(1)
nor4(m41,m42-k1):Nor/2(1)
%
xor11(x3,y3-xy3):Xor/2(1)
xor12(x2,y2-xy2):Xor/2(1)
xor13(x1,y1-xy1):Xor/2(1)
xor14(x0,y0-xy0):Xor/2(1)
xor21(xy3,k3-f3):Xor/2(1)
xor22(xy2,k2-f2):Xor/2(1)
xor23(xy1,k1-f1):Xor/2(1)
xor24(xy0,k0-f0):Xor/2(1)
%
etnet1(x0,x1,x2,x3,cin-xc):Nand/5(1)
net2(x0,x1,x2,x3-p):Nand/4(1)
net3(mb,cin-k0):Nand/2(1)
net4(xc,g-cout):Nand/2(1)
eteq(f0,f1,f2,f3-eq):And/4(1)
non1(m-mb):Not(1)
```

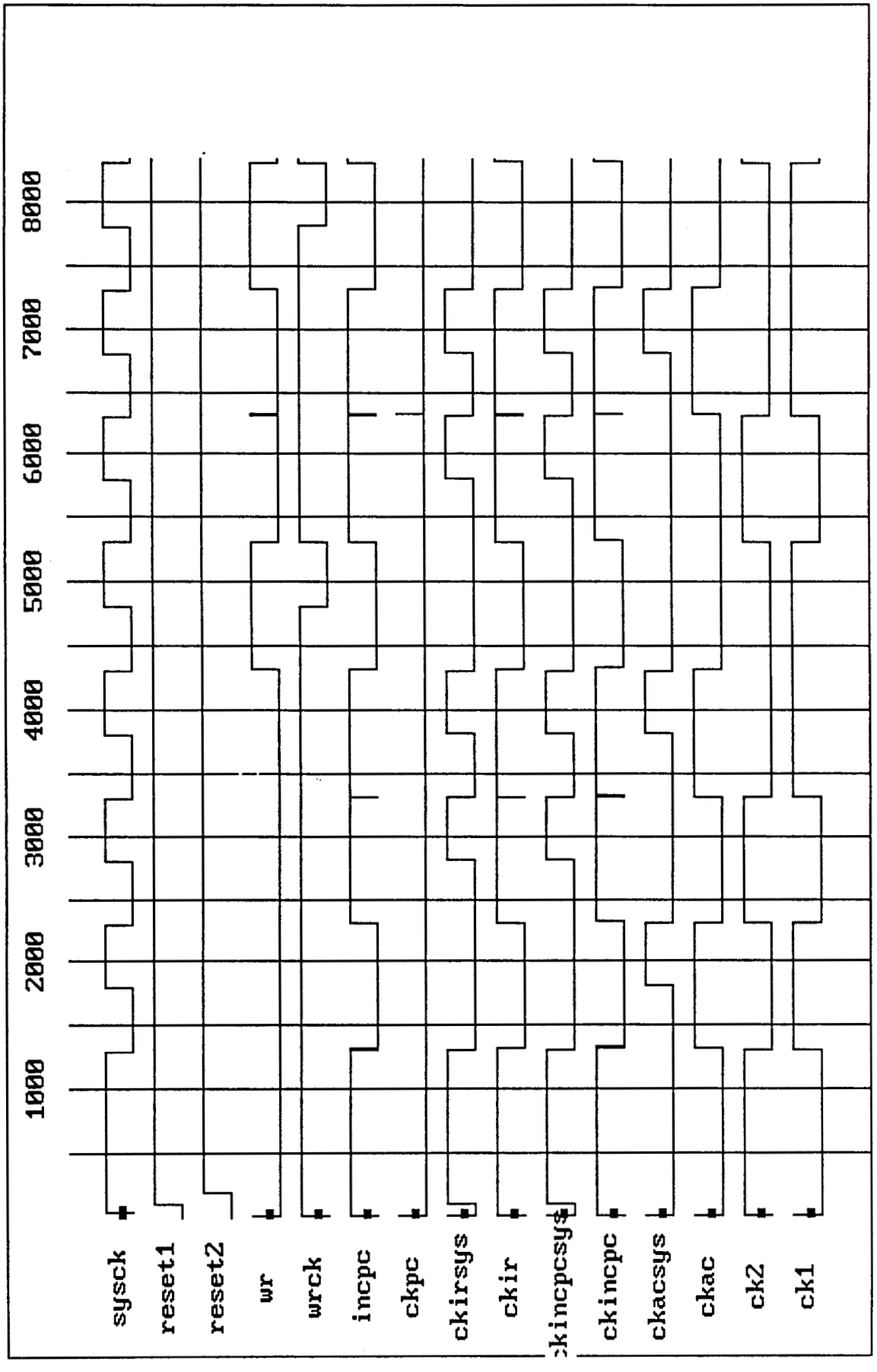
5) Sous-ensemble de l'ALU 74181

```
SousAlu(s0@3,a,b-x,y)
% sous-ensemble de l'alu 74181
et1(s3,a,b-k1):And/3(1)
et2(s2,a,bb-k2):And/3(1)
et3(s1,bb-k3):And/2(1)
et4(s0,b-k4):And/2(1)
et5(a-k5):Delay(1)
n(b-bb):Not(1)
no1(k1,k2-x):Nor/2(1)
no2(k3,k4,k5-y):Nor/3(1)
```

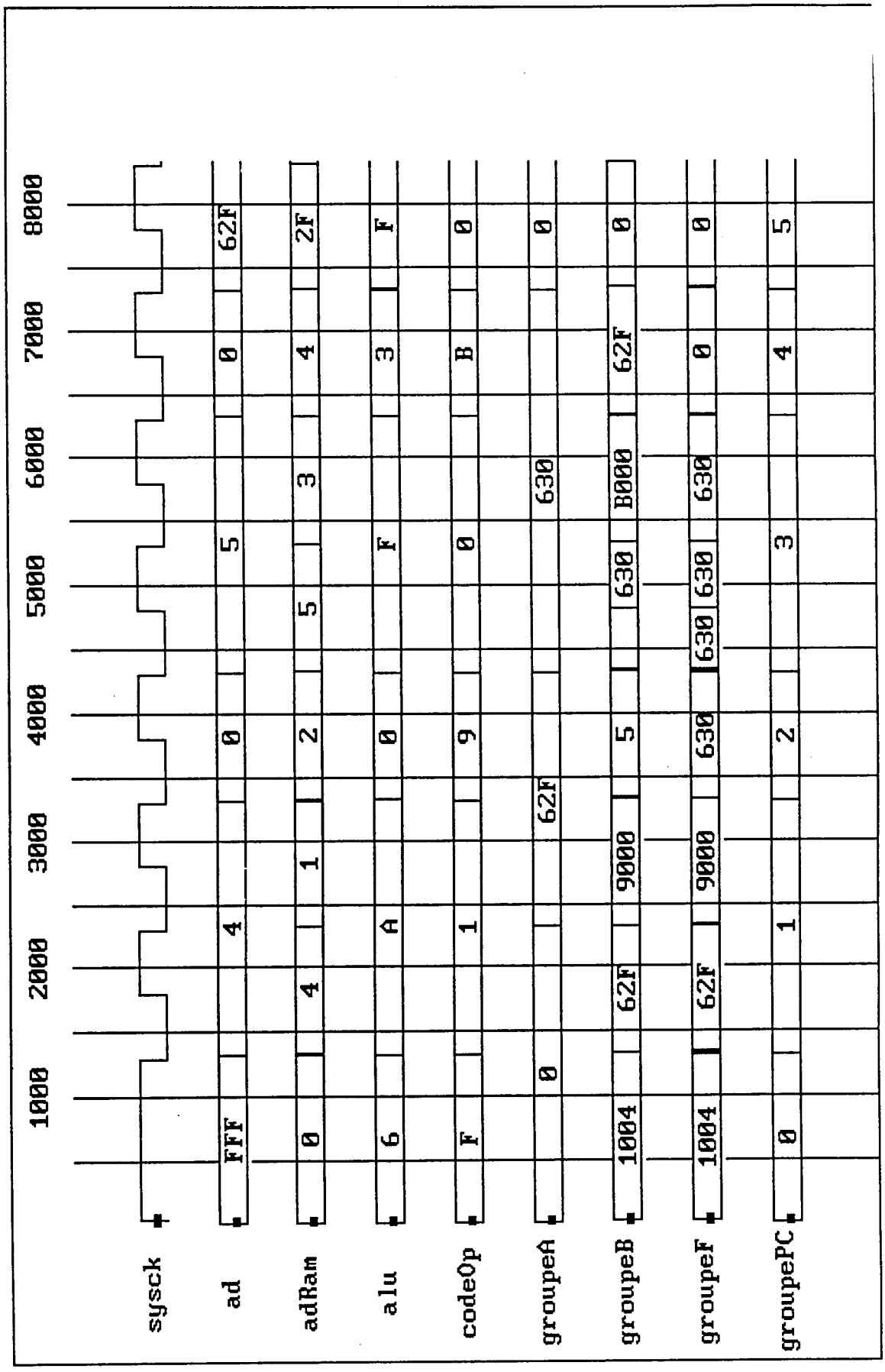
6) Multiplexeur 2 mots de 16 bits

```
Mux2d16(en,a0@15,b0@15,sel-s0@15)
m0(en,sel,a0,b0-s0):Mux/1()
m1(en,sel,a1,b1-s1):Mux/1()
m2(en,sel,a2,b2-s2):Mux/1()
m3(en,sel,a3,b3-s3):Mux/1()
m4(en,sel,a4,b4-s4):Mux/1()
m5(en,sel,a5,b5-s5):Mux/1()
m6(en,sel,a6,b6-s6):Mux/1()
m7(en,sel,a7,b7-s7):Mux/1()
m8(en,sel,a8,b8-s8):Mux/1()
m9(en,sel,a9,b9-s9):Mux/1()
m10(en,sel,a10,b10-s10):Mux/1()
m11(en,sel,a11,b11-s11):Mux/1()
m12(en,sel,a12,b12-s12):Mux/1()
m13(en,sel,a13,b13-s13):Mux/1()
m14(en,sel,a14,b14-s14):Mux/1()
m15(en,sel,a15,b15-s15):Mux/1()
```

Simulation du processeur TOY : chronogramme 1



Simulation du processeur TOY : chronogramme 2



ANNEXE 4

Lecture de fichier au format HILO

Le listing de la page suivante est la sortie fournie par HILO en utilisant la commande EXPAND. Elle est suivie du fichier produit après traduction par le simulateur.

Le circuit est de type 74169, c'est un compteur-décompteur synchrone 4 bits préchargeable constitué de 81 portes

```
CCTDM74ALS169(
W0,**UND
W1,**CK
W2,**IP[1]
W3,**IP[2]
W4,**IP[3]
W5,**IP[4]
W6,**ENP
W7,**GND
W8,**LOAD
W9,**ENT
W13,**Q[4]
W12,**Q[3]
W11,**Q[2]
W10,**Q[1]
W14,**RCO
W15);**VCC
AND
(HIGH=STRONG,LOW=STRONG)
G0(**G1[1]
W16,**W1[1]
W64,**DNQ[1]
W20);**DWN
AND
(HIGH=STRONG,LOW=STRONG)
G1(**G1[2]
W17,**W1[2]
W71,**DNQ[2]
W20);**DWN
AND
(HIGH=STRONG,LOW=STRONG)
G2(**G1[3]
W18,**W1[3]
W78,**DNQ[3]
W20);**DWN
AND
(HIGH=STRONG,LOW=STRONG)
G3(**G1[4]
W19,**W1[4]
W85,**DNQ[4]
W20);**DWN
AND
(HIGH=STRONG,LOW=STRONG)
G4(**G2[1]
W21,**W2[1]
W63,**DQ[1]
W25);**UP
AND
(HIGH=STRONG,LOW=STRONG)
G5(**G2[2]
W22,**W2[2]
W70,**DQ[2]
W25);**UP
AND
(HIGH=STRONG,LOW=STRONG)
G6(**G2[3]
W23,**W2[3]
W77,**DQ[3]
W25);**UP
AND
(HIGH=STRONG,LOW=STRONG)
G7(**G2[4]
W24,**W2[4]
W84,**DQ[4]
W25);**UP
AND
(HIGH=STRONG,LOW=STRONG)
G8(**G11[1]
W26,**W11[1]
W63,**DQ[1]
W30);**W14
AND
(HIGH=STRONG,LOW=STRONG)
G9(**G11[2]
W27,**W11[2]
W70,**DQ[2]
W30);**W14
AND
(HIGH=STRONG,LOW=STRONG)
G10(**G11[3]
W28,**W11[3]
W77,**DQ[3]
W30);**W14
AND
(HIGH=STRONG,LOW=STRONG)
G11(**G11[4]
W29,**W11[4]
W84,**DQ[4]
W30);**W14
AND
(HIGH=STRONG,LOW=STRONG)
G12(**G17
W32,**W10[2]
W35,**W3[1]
W31);**W10[1]
AND
(HIGH=STRONG,LOW=STRONG)
G13(**G18
W33,**W10[3]
W35,**W3[1]
W31,**W10[1]
W36);**W3[2]
AND
(HIGH=STRONG,LOW=STRONG)
G14(**G19
W34,**W10[4]
W35,**W3[1]
W31,**W10[1]
W36,**W3[2]
W37);**W3[3]
AND
(HIGH=STRONG,LOW=STRONG)
G15(**G21
W39,**W21
W25,**UP
W35,**W3[1]
W36,**W3[2]
W37,**W3[3]
W38,**W3[4]
W40);**W20
AND
(HIGH=STRONG,LOW=STRONG)
G16(**G22
W41,**W22
W20,**DWN
W35,**W3[1]
W36,**W3[2]
W37,**W3[3]
W38,**W3[4]
W40);**W20
AND(1:4:6,1:4:6)
(HIGH=STRONG,LOW=STRONG)
G17(**G12
W87,**BCLK
W1);**CK
NOR
(HIGH=STRONG,LOW=STRONG)
G18(**G10
W31,**W10[1]
W42,**W13
W6,**ENP
W9);**ENT
NOR(2:2:7,1:2:6)
(HIGH=STRONG,LOW=STRONG)
G19(**G3[1]
W35,**W3[1]
W16,**W1[1]
W21);**W2[1]
NOR(2:2:7,1:2:6)
(HIGH=STRONG,LOW=STRONG)
G20(**G3[2]
W36,**W3[2]
W17,**W1[2]
W22);**W2[2]
NOR(2:2:7,1:2:6)
(HIGH=STRONG,LOW=STRONG)
G21(**G3[3]
W37,**W3[3]
W18,**W1[3]
W23);**W2[3]
NOR(2:2:7,1:2:6)
(HIGH=STRONG,LOW=STRONG)
G22(**G3[4]
W38,**W3[4]
W19,**W1[4]
```

```
W24);**W2[4]
NOR(1:3:5,1:4:6)
(HIGH=STRONG,LOW=STRONG)
G23(**G23
W14,**RCO
W39,**W21
W41);**W22
NAND
(HIGH=STRONG,LOW=STRONG)
G24(**G4[1]
W65,**DI[1]
W43,**W5[1]
W47,**W6[1]
W51);**W7[1]
NAND
(HIGH=STRONG,LOW=STRONG)
G25(**G4[2]
W72,**DI[2]
W44,**W5[2]
W48,**W6[2]
W52);**W7[2]
NAND
(HIGH=STRONG,LOW=STRONG)
G26(**G4[3]
W79,**DI[3]
W45,**W5[3]
W49,**W6[3]
W53);**W7[3]
NAND
(HIGH=STRONG,LOW=STRONG)
G27(**G4[4]
W86,**DI[4]
W46,**W5[4]
W50,**W6[4]
W54);**W7[4]
NAND
(HIGH=STRONG,LOW=STRONG)
G28(**G5[1]
W43,**W5[1]
W31,**W10[1]
W55);**W8[1]
NAND
(HIGH=STRONG,LOW=STRONG)
G29(**G5[2]
W44,**W5[2]
W32,**W10[2]
W56);**W8[2]
NAND
(HIGH=STRONG,LOW=STRONG)
G30(**G5[3]
W45,**W5[3]
W33,**W10[3]
W57);**W8[3]
NAND
(HIGH=STRONG,LOW=STRONG)
G31(**G5[4]
W46,**W5[4]
W34,**W10[4]
W58);**W8[4]
NAND
(HIGH=STRONG,LOW=STRONG)
G32(**G6[1]
W47,**W6[1]
W26,**W11[1]
W59);**W9[1]
NAND
(HIGH=STRONG,LOW=STRONG)
G33(**G6[2]
W48,**W6[2]
W27,**W11[2]
W60);**W9[2]
NAND
(HIGH=STRONG,LOW=STRONG)
G34(**G6[3]
W49,**W6[3]
W28,**W11[3]
W61);**W9[3]
NAND
(HIGH=STRONG,LOW=STRONG)
G35(**G6[4]
W50,**W6[4]
W29,**W11[4]
W62);**W9[4]
NAND
(HIGH=STRONG,LOW=STRONG)
G36(**G7[1]
W51,**W7[1]
W42,**W13
W2);**IP[1]
NAND
(HIGH=STRONG,LOW=STRONG)
G37(**G7[2]
W52,**W7[2]
W42,**W13
W3);**IP[2]
NAND
(HIGH=STRONG,LOW=STRONG)
G38(**G7[3]
W53,**W7[3]
W42,**W13
W4);**IP[3]
NAND
(HIGH=STRONG,LOW=STRONG)
G39(**G7[4]
W54,**W7[4]
W42,**W13
W5);**IP[4]
NAND
(HIGH=STRONG,LOW=STRONG)
G40(**G8[1]
W55,**W8[1]
W26);**W11[1]
NAND
(HIGH=STRONG,LOW=STRONG)
G41(**G8[2]
W56,**W8[2]
W27);**W11[2]
NAND
(HIGH=STRONG,LOW=STRONG)
G42(**G8[3]
W57,**W8[3]
W28);**W11[3]
NAND
(HIGH=STRONG,LOW=STRONG)
G43(**G8[4]
W58,**W8[4]
W29);**W11[4]
NAND
(HIGH=STRONG,LOW=STRONG)
G44(**G9[1]
W59,**W9[1]
W31);**W10[1]
NAND
(HIGH=STRONG,LOW=STRONG)
G45(**G9[2]
W60,**W9[2]
W32);**W10[2]
NAND
(HIGH=STRONG,LOW=STRONG)
G46(**G9[3]
W61,**W9[3]
W33);**W10[3]
NAND
(HIGH=STRONG,LOW=STRONG)
G47(**G9[4]
W62,**W9[4]
W34);**W10[4]
NAND
(HIGH=STRONG,LOW=STRONG)
G48(**G13
W42,**W13
W8);**LOAD
NAND
(HIGH=STRONG,LOW=STRONG)
G49(**G14
W30,**W14
W42);**W13
NAND
(HIGH=STRONG,LOW=STRONG)
G50(**G16
W20,**DWN
W25);**UP
NAND(3,4)
(HIGH=STRONG,LOW=STRONG)
G51(**G15
```

```
W25,**UP
W0);**UND
NAND(1:2:4,1:2:4)
(HIGH=STRONG,LOW=STRONG)
G52(**G20
W40,**W20
W9);**ENT
NAND(1:1:4,1:1:5)
(HIGH=STRONG,LOW=STRONG)
G53(**G24[1]
W10,**Q[1]
W64);**DNQ[1]
NAND(1:1:4,1:1:5)
(HIGH=STRONG,LOW=STRONG)
G54(**G24[2]
W11,**Q[2]
W71);**DNQ[2]
NAND(1:1:4,1:1:5)
(HIGH=STRONG,LOW=STRONG)
G55(**G24[3]
W12,**Q[3]
W78);**DNQ[3]
NAND(1:1:4,1:1:5)
(HIGH=STRONG,LOW=STRONG)
G56(**G24[4]
W13,**Q[4]
W85);**DNQ[4]
NAND(1:4:5,1:4:5)
(HIGH=STRONG,LOW=STRONG)
G57(**REG[1]_G2
W66,**REG[1]_W2
W67,**REG[1]_W3
W68);**REG[1]_W5
NAND(1:4:5,1:4:5)
(HIGH=STRONG,LOW=STRONG)
G58(**REG[1]_G4
W69,**REG[1]_W4
W87,**BCLK
W67,**REG[1]_W3
W68);**REG[1]_W5
NAND(1:4:5,1:4:5)
(HIGH=STRONG,LOW=STRONG)
G59(**REG[1]_G6
W63,**DQ[1]
W67,**REG[1]_W3
W64);**DNQ[1]
NAND
(HIGH=STRONG,LOW=STRONG)
G60(**REG[1]_G3
W67,**REG[1]_W3
W66,**REG[1]_W2
W87);**BCLK
NAND
(HIGH=STRONG,LOW=STRONG)
G61(**REG[1]_G5
W68,**REG[1]_W5
W69,**REG[1]_W4
W65);**DI[1]
NAND
(HIGH=STRONG,LOW=STRONG)
G62(**REG[1]_G7
W64,**DNQ[1]
W63,**DQ[1]
W69);**REG[1]_W4
NAND(1:4:5,1:4:5)
(HIGH=STRONG,LOW=STRONG)
G63(**REG[2]_G2
W73,**REG[2]_W2
W74,**REG[2]_W3
W75);**REG[2]_W5
NAND(1:4:5,1:4:5)
(HIGH=STRONG,LOW=STRONG)
G64(**REG[2]_G4
W76,**REG[2]_W4
W87,**BCLK
W74,**REG[2]_W3
W75);**REG[2]_W5
NAND(1:4:5,1:4:5)
(HIGH=STRONG,LOW=STRONG)
G65(**REG[2]_G6
W70,**DQ[2]
W74,**REG[2]_W3
W71);**DNQ[2]
NAND
(HIGH=STRONG,LOW=STRONG)
G66(**REG[2]_G3
W74,**REG[2]_W3
W73,**REG[2]_W2
W87);**BCLK
NAND
(HIGH=STRONG,LOW=STRONG)
G67(**REG[2]]_G5
W75,**REG[2]_W5
W76,**REG[2]_W4
W72);**DI[2]
NAND
(HIGH=STRONG,LOW=STRONG)
G68(**REG[2]_G7
W71,**DNQ[2]
W70,**DQ[2]
W76);**REG[2]_W4
NAND(1:4:5,1:4:5)
(HIGH=STRONG,LOW=STRONG)
G69(**REG[3]_G2
W80,**REG[3]_W2
W81,**REG[3]_W3
W82);**REG[3]_W5
NAND(1:4:5,1:4:5)
(HIGH=STRONG,LOW=STRONG)
G70(**REG[3]_G4
W83,**REG[3]_W4
W87,**BCLK
W81,**REG[3]_W3
W82);**REG[3]_W5
NAND(1:4:5,1:4:5)
(HIGH=STRONG,LOW=STRONG)
G71(**REG[3]_G6
W77,**DQ[3]
W81,**REG[3]_W3
W78);**DNQ[3]
NAND
(HIGH=STRONG,LOW=STRONG)
G72(**REG[3]_G3
W81,**REG[3]_W3
W80,**REG[3]_W2
W87);**BCLK
NAND
(HIGH=STRONG,LOW=STRONG)
G73(**REG[3]_G5
W82,**REG[3]_W5
W83,**REG[3]_W4
W79);**DI[3]
NAND
(HIGH=STRONG,LOW=STRONG)
G74(**REG[3]_G7
W78,**DNQ[3]
W77,**DQ[3]
W83);**REG[3]_W4
NAND(1:4:5,1:4:5)
(HIGH=STRONG,LOW=STRONG)
G75(**REG[4]_G2
W88,**REG[4]_W2
W89,**REG[4]_W3
W90);**REG[4]_W5
NAND(1:4:5,1:4:5)
(HIGH=STRONG,LOW=STRONG)
G76(**REG[4]_G4
W91,**REG[4]_W4
W87,**BCLK
W89,**REG[4]_W3
W90);**REG[4]_W5
NAND(1:4:5,1:4:5)
(HIGH=STRONG,LOW=STRONG)
G77(**REG[4]_G6
W84,**DQ[4]
W89,**REG[4]_W3
W85);**DNQ[4]
NAND
(HIGH=STRONG,LOW=STRONG)
G78(**REG[4]_G3
W89,**REG[4]_W3
W88,**REG[4]_W2
W87);**BCLK
NAND
(HIGH=STRONG,LOW=STRONG)
```

```
G79(**REG[4]_G5
W90,**REG[4]_W5
W91,**REG[4]_W4
W86);**DI[4]
NAND
(HIGH=STRONG,LOW=STRONG)
G80(**REG[4]_G7
W85,**DNQ[4]
W84,**DQ[4]
W91);**REG[4]_W4
INPUTW0(**UND
G51.2**G15
,DM74ALS169.1**DM74ALS169
);
INPUTW1(**CK
G17.2**G12
,DM74ALS169.2**DM74ALS169
);
INPUTW2(**IP[1]
G36.3**G7[1]
,DM74ALS169.3**DM74ALS169
);
INPUTW3(**IP[2]
G37.3**G7[2]
,DM74ALS169.4**DM74ALS169
);
INPUTW4(**IP[3]
G38.3**G7[3]
,DM74ALS169.5**DM74ALS169
);
INPUTW5(**IP[4]
G39.3**G7[4]
,DM74ALS169.6**DM74ALS169
);
INPUTW6(**ENP
G18.3**G10
,DM74ALS169.7**DM74ALS169
);
SUPPLY0W7(**GND
DM74ALS169.8**DM74ALS169
);
INPUTW8(**LOAD
G48.2**G13
,DM74ALS169.9**DM74ALS169
);
INPUTW9(**ENT
G52.2**G20
,G18.4**G10
,DM74ALS169.10**DM74ALS169
);
UNIDW10(**Q[1]
DM74ALS169.14**DM74ALS169
,G53.1**G24[1]
);
UNIDW11(**Q[2]
DM74ALS169.13**DM74ALS169
,G54.1**G24[2]
);
UNIDW12(**Q[3]
DM74ALS169.12**DM74ALS169
,G55.1**G24[3]
);
UNIDW13(**Q[4]
DM74ALS169.11**DM74ALS169
,G56.1**G24[4]
);
UNIDW14(**RCO
DM74ALS169.15**DM74ALS169
,G23.1**G23
);
SUPPLY1W15(**VCC
DM74ALS169.16**DM74ALS169
);
UNIDW16(**W1[1]
G19.2**G3[1]
,G0.1**G1[1]
);
UNIDW17(**W1[2]
G20.2**G3[2]
,G1.1**G1[2]
);
UNIDW18(**W1[3]
G21.2**G3[3]
,G2.1**G1[3]
);
UNIDW19(**W1[4]
G22.2**G3[4]
,G3.1**G1[4]
);
UNIDW20(**DWN
G16.2**G22
,G3.3**G1[4]
,G2.3**G1[3]
,G1.3**G1[2]
,G0.3**G1[1]
,G50.1**G16
);
UNIDW21(**W2[1]
G19.3**G3[1]
,G4.1**G2[1]
);
UNIDW22(**W2[2]
G20.3**G3[2]
,G5.1**G2[2]
);
UNIDW23(**W2[3]
G21.3**G3[3]
,G6.1**G2[3]
);
UNIDW24(**W2[4]
G22.3**G3[4]
,G7.1**G2[4]
);
UNIDW25(**UP
G50.2**G16
,G15.2**G21
,G7.3**G2[4]
,G6.3**G2[3]
,G5.3**G2[2]
,G4.3**G2[1]
,G51.1**G15
);
UNIDW26(**W11[1]
G40.2**G8[1]
,G32.2**G6[1]
,G8.1**G11[1]
);
UNIDW27(**W11[2]
G41.2**G8[2]
,G33.2**G6[2]
,G9.1**G11[2]
);
UNIDW28(**W11[3]
G42.2**G8[3]
,G34.2**G6[3]
,G10.1**G11[3]
);
UNIDW29(**W11[4]
G43.2**G8[4]
,G35.2**G6[4]
,G11.1**G11[4]
);
UNIDW30(**W14
G11.3**G11[4]
,G10.3**G11[3]
,G9.3**G11[2]
,G8.3**G11[1]
,G49.1**G14
);
UNIDW31(**W10[1]
G44.2**G9[1]
,G28.2**G5[1]
,G14.3**G19
,G13.3**G18
,G12.3**G17
,G18.1**G10
);
UNIDW32(**W10[2]
G45.2**G9[2]
,G29.2**G5[2]
,G12.1**G17
);
UNIDW33(**W10[3]
G46.2**G9[3]
,G30.2**G5[3]
```

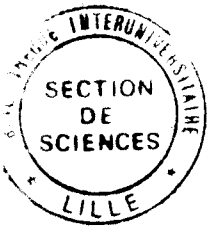
```
,G13.1**G18
);
UNIDW34(**W10[4]
G47.2**G9[4]
,G31.2**G5[4]
,G14.1**G19
);
UNIDW35(**W3[1]
G16.3**G22
,G15.3**G21
,G14.2**G19
,G13.2**G18
,G12.2**G17
,G19.1**G3[1]
);
UNIDW36(**W3[2]
G16.4**G22
,G15.4**G21
,G14.4**G19
,G13.4**G18
,G20.1**G3[2]
);
UNIDW37(**W3[3]
G16.5**G22
,G15.5**G21
,G14.5**G19
,G21.1**G3[3]
);
UNIDW38(**W3[4]
G16.6**G22
,G15.6**G21
,G22.1**G3[4]
);
UNIDW39(**W21
G23.2**G23
,G15.1**G21
);
UNIDW40(**W20
G16.7**G22
,G15.7**G21
,G52.1**G20
);
UNIDW41(**W22
G23.3**G23
,G16.1**G22
);
UNIDW42(**W13
G49.2**G14
,G39.2**G7[4]
,G38.2**G7[3]
,G37.2**G7[2]
,G36.2**G7[1]
,G18.2**G10
,G48.1**G13
);
UNIDW43(**W5[1]
G24.2**G4[1]
,G28.1**G5[1]
);
UNIDW44(**W5[2]
G25.2**G4[2]
,G29.1**G5[2]
);
UNIDW45(**W5[3]
G26.2**G4[3]
,G30.1**G5[3]
);
UNIDW46(**W5[4]
G27.2**G4[4]
,G31.1**G5[4]
);
UNIDW47(**W6[1]
G24.3**G4[1]
,G32.1**G6[1]
);
UNIDW48(**W6[2]
G25.3**G4[2]
,G33.1**G6[2]
);
UNIDW49(**W6[3]
G26.3**G4[3]
,G34.1**G6[3]
);
UNIDW50(**W6[4]
G27.3**G4[4]
,G35.1**G6[4]
);
UNIDW51(**W7[1]
G24.4**G4[1]
,G36.1**G7[1]
);
UNIDW52(**W7[2]
G25.4**G4[2]
,G37.1**G7[2]
);
UNIDW53(**W7[3]
G26.4**G4[3]
,G38.1**G7[3]
);
UNIDW54(**W7[4]
G27.4**G4[4]
,G39.1**G7[4]
);
UNIDW55(**W8[1]
G28.3**G5[1]
,G40.1**G8[1]
);
UNIDW56(**W8[2]
G29.3**G5[2]
,G41.1**G8[2]
);
UNIDW57(**W8[3]
G30.3**G5[3]
,G42.1**G8[3]
);
UNIDW58(**W8[4]
G31.3**G5[4]
,G43.1**G8[4]
);
UNIDW59(**W9[1]
G32.3**G6[1]
,G44.1**G9[1]
);
UNIDW60(**W9[2]
G33.3**G6[2]
,G45.1**G9[2]
);
UNIDW61(**W9[3]
G34.3**G6[3]
,G46.1**G9[3]
);
UNIDW62(**W9[4]
G35.3**G6[4]
,G47.1**G9[4]
);
UNIDW63(**DQ[1]
G62.2**REG[1]_G7
,G59.1**REG[1]_G6
,G8.2**G11[1]
,G4.2**G2[1]
);
UNIDW64(**DNQ[1]
G59.3**REG[1]_G6
,G62.1**REG[1]_G7
,G53.2**G24[1]
,G0.2**G1[1]
);
UNIDW65(**DI[1]
G61.3**REG[1]_G5
,G24.1**G4[1]
);
UNIDW66(**REG[1]_W2
G60.2**REG[1]_G3
,G57.1**REG[1]_G2
);
UNIDW67(**REG[1]_W3
G59.2**REG[1]_G6
,G58.3**REG[1]_G4
,G57.2**REG[1]_G2
,G60.1**REG[1]_G3
);
UNIDW68(**REG[1]_W5
G58.4**REG[[1]_G4
,G57.3**REG[1]_G2
,G61.1**REG[1]_G5
```

```
);
UNIDW69(**REG[1]_W4
G62.3**REG[1]_G7
,G61.2**REG[1]_G5
,G58.1**REG[1]_G4
);
UNIDW70(**DQ[2]
G68.2**REG[2]_G7
,G65.1**REG[2]_G6
,G9.2**G11[2]
,G5.2**G2[2]
);
UNIDW71(**DNQ[2]
G65.3**REG[2]_G6
,G68.1**REG[2]_G7
,G54.2**G24[2]
,G1.2**G1[2]
);
UNIDW72(**DI[2]
G67.3**REG[2]_G5
,G25.1**G4[2]
);
UNIDW73(**REG[2]_W2
G66.2**REG[2]_G3
,G63.1**REG[2]_G2
);
UNIDW74(**REG[2]_W3
G65.2**REG[2]_G6
,G64.3**REG[2]_G4
,G63.2**REG[2]_G2
,G66.1**REG[2]_G3
);
UNIDW75(**REG[2]_W5
G64.4**REG[2]_G4
,G63.3**REG[2]_G2
,G67.1**REG[2]_G5
);
UNIDW76(**REG[2]_W4
G68.3**REG[2]_G7
,G67.2**REG[2]_G5
,G64.1**REG[2]_G4
);
UNIDW77(**DQ[3]
G74.2**REG[3]_G7
,G71.1**REG[3]_G6
,G10.2**G11[3]
,G6.2**G2[3]
);
UNIDW78(**DNQ[3]
G71.3**REG[3]_G6
,G74.1**REG[3]_G7
,G55.2**G24[3]
,G2.2**G1[3]
);
UNIDW79(**DI[3]
G73.3**REG[3]_G5
,G26.1**G4[3]
);
UNIDW80(**REG[3]_W2
G72.2**REG[3]_G3
,G69.1**REG[3]_G2
);
UNIDW81(**REG[3]_W3
G71.2**REG[3]_G6
,G70.3**REG[3]_G4
,G69.2**REG[3]_G2
,G72.1**REG[3]_G3
);
UNIDW82(**REG[3]_W5
G70.4**REG[3]_G4
,G69.3**REG[3]_G2
,G73.1**REG[3]_G5
);
UNIDW83(**REG[3]_W4
G74.3**REG[3]_G7
,G73.2**REG[3]_G5
,G70.1**REG[3]_G4
);
UNIDW84(**DQ[4]
G80.2**REG[4]_G7
,G77.1**REG[4]_G6
,G11.2**G11[4]
,G7.2**G2[4]
);
UNIDW85(**DNQ[4]
G77.3**REG[4]_G6
,G80.1**REG[4]_G7
,G56.2**G24[4]
,G3.2**G1[4]
);
UNIDW86(**DI[4]
G79.3**REG[4]_G5
,G27.1**G4[4]
);
UNIDW87(**BCLK
G78.3**REG[4]_G3
,G76.2**REG[4]_G4
,G72.3**REG[3]_G3
,G70.2**REG[3]_G4
,G66.3**REG[2]_G3
,G64.2**REG[2]_G4
,G60.3**REG[1]_G3
,G58.2**REG[1]_G4
,G17.1**G12
);
UNIDW88(**REG[4]_W2
G78.2**REG[4]_G3
,G75.1**REG[4]_G2
);
UNIDW89(**REG[4]_W3
G77.2**REG[4]_G6
,G76.3**REG[4]_G4
,G75.2**REG[4]_G2
,G78.1**REG[4]_G3
);
UNIDW90(**REG[4]_W5
G76.4**REG[4]_G4
,G75.3**REG[4]_G2
,G79.1**REG[4]_G5
);
UNIDW91(**REG[4]_W4
G80.3**REG[4]_G7
,G79.2**REG[4]_G5
,G76.1**REG[4]_G4
);
.
```

Fichier généré au format propre du simulateur

```
Dm74als1(w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w15-w13,w12,w11,w10,w14)
g0(w64,w20-w16):And/2(0,0)
g1(w71,w20-w17):And/2(0,0)
g2(w78,w20-w18):And/2(0,0)
g3(w85,w20-w19):And/2(0,0)
g4(w63,w25-w21):And/2(0,0)
g5(w70,w25-w22):And/2(0,0)
g6(w77,w25-w23):And/2(0,0)
g7(w84,w25-w24):And/2(0,0)
g8(w63,w30-w26):And/2(0,0)
g9(w70,w30-w27):And/2(0,0)
g10(w77,w30-w28):And/2(0,0)
g11(w84,w30-w29):And/2(0,0)
g12(w35,w31-w32):And/2(0,0)
g13(w35,w31,w36-w33):And/3(0,0)
g14(w35,w31,w36,w37-w34):And/4(0,0)
g15(w25,w35,w36,w37,w38,w40-w39):And/6(0,0)
g16(w20,w35,w36,w37,w38,w40-w41):And/6(0,0)
g17(w1-w87):And/1(3,3)
g18(w42,w6,w9-w31):Nor/3(0,0)
g19(w16,w21-w35):Nor/2(3,3)
g20(w17,w22-w36):Nor/2(3,3)
g21(w18,w23-w37):Nor/2(3,3)
g22(w19,w24-w38):Nor/2(3,3)
g23(w39,w41-w14):Nor/2(3,3)
g24(w43,w47,w51-w65):Nand/3(0,0)
g25(w44,w48,w52-w72):Nand/3(0,0)
g26(w45,w49,w53-w79):Nand/3(0,0)
g27(w46,w50,w54-w86):Nand/3(0,0)
g28(w31,w55-w43):Nand/2(0,0)
g29(w32,w56-w44):Nand/2(0,0)
g30(w33,w57-w45):Nand/2(0,0)
g31(w34,w58-w46):Nand/2(0,0)
g32(w26,w59-w47):Nand/2(0,0)
g33(w27,w60-w48):Nand/2(0,0)
g34(w28,w61-w49):Nand/2(0,0)
g35(w29,w62-w50):Nand/2(0,0)
g36(w42,w2-w51):Nand/2(0,0)
g37(w42,w3-w52):Nand/2(0,0)
g38(w42,w4-w53):Nand/2(0,0)
g39(w42,w5-w54):Nand/2(0,0)
g40(w26-w55):Nand/1(0,0)
g41(w27-w56):Nand/1(0,0)
g42(w28-w57):Nand/1(0,0)
g43(w29-w58):Nand/1(0,0)
g44(w31-w59):Nand/1(0,0)
g45(w32-w60):Nand/1(0,0)
g46(w33-w61):Nand/1(0,0)
g47(w34-w62):Nand/1(0,0)
g48(w8-w42):Nand/1(0,0)
g49(w42-w30):Nand/1(0,0)
g50(w25-w20):Nand/1(0,0)
```

```
g51(w0-w25):Nand/1(3,4)
g52(w9-w40):Nand/1(2,2)
g53(w64-w10):Nand/1(2,2)
g54(w71-w11):Nand/1(2,2)
g55(w78-w12):Nand/1(2,2)
g56(w85-w13):Nand/1(2,2)
g57(w67,w68-w66):Nand/2(3,3)
g58(w87,w67,w68-w69):Nand/3(3,3)
g59(w67,w64-w63):Nand/2(3,3)
g60(w66,w87-w67):Nand/2(0,0)
g61(w69,w65-w68):Nand/2(0,0)
g62(w63,w69-w64):Nand/2(0,0)
g63(w74,w75-w73):Nand/2(3,3)
g64(w87,w74,w75-w76):Nand/3(3,3)
g65(w74,w71-w70):Nand/2(3,3)
g66(w73,w87-w74):Nand/2(0,0)
g67(w76,w72-w75):Nand/2(0,0)
g68(w70,w76-w71):Nand/2(0,0)
g69(w81,w82-w80):Nand/2(3,3)
g70(w87,w81,w82-w83):Nand/3(3,3)
g71(w81,w78-w77):Nand/2(3,3)
g72(w80,w87-w81):Nand/2(0,0)
g73(w83,w79-w82):Nand/2(0,0)
g74(w77,w83-w78):Nand/2(0,0)
g75(w89,w90-w88):Nand/2(3,3)
g76(w87,w89,w90-w91):Nand/3(3,3)
g77(w89,w85-w84):Nand/2(3,3)
g78(w88,w87-w89):Nand/2(0,0)
g79(w91,w86-w90):Nand/2(0,0)
g80(w84,w91-w85):Nand/2(0,0)
```



ABSTRACT

The adequacy of object concepts for the realisation of computer aided design environments is studied through the realisation of a SMALLTALK logic simulator.

If the object approach perfectly meets the traditional CAD requirements, it also allows an opening onto more original functions, thanks to its data representation power and to its symbolic processing capacities.

The modularity and the re-usability of the functions developed lead to a high productivity when writing the software.

The modelling swiftness of users' interfaces guarantees both ergonomics and conviviality, with a presentation which is close to the reality of objects in the real world.

The efficiency problems, due to a bad adjustment of machine architecture to object concepts, do not question the interest of such an approach.

Keywords

Computer Aided Design
Object Languages
Logic Simulation

RESUME

L'adéquation des concepts objets pour la réalisation d'environnements de conception assistée par ordinateur est étudiée au travers de la réalisation d'un simulateur logique en SMALLTALK.

Si l'approche objet répond parfaitement aux besoins traditionnels de la CAO, elle permet aussi une ouverture sur des fonctions plus originales, grâce à sa puissance de représentation des données et à ses capacités de traitement symbolique.

La modularité et la ré-utilisabilité des fonctions développées amènent une forte productivité lors de l'écriture de logiciels. La rapidité de maquettage des interfaces utilisateurs garantit l'ergonomie et la convivialité, avec une présentation proche de la réalité des objets du monde réel.

Les problèmes de performances, dus à une mauvaise adaptation de l'architecture des machines aux concepts objets, ne remettent pas en cause l'intérêt d'une telle approche.

Mots clefs

Conception Assistée par Ordinateur
Langages Orientés Objet
Simulation Logique