N° d'ordre : 1074

UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

THÈSE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

pour obtenir

LE TITRE DE DOCTEUR DE 3ème CYCLE

Informatique

par

Jacques BRYGIER

ETUDES D'ELEMENTS D'ARCHITECTURE POUR UNE MACHINE LANGAGE ADA

Thèse soutenue le 9 septembre 1983 devant la Commission d'Examen

Membres du Jury :

Président	C. CARREZ
Rapporteur	V. CORDONNIER
Membres	M. RENVOISE
	M. DOMINE
	M. CHEVANCE

PROFESSEURS 1ère CLASSE

M.	BACCHUS Pierre	Mathématiques
M.	BEAUFILS Jean-Pierre (dét.)	Chimie
M.	BIAYS Pierre	G.A.S.
M.	BILLARD Jean	Physique
M.	BONNOT Ernest	Biologie
M.	BOUGHON Pierre	Mathématiques
M.	BOURIQUET Robert	Biologie
M.	CELET Paul	Sciences de la Terre
M.	COEURE Gérard	Mathématiques
M.	CONSTANT Eugène	I.E.E.A.
M.	CORDONNIER Vincent	I.E.E.A.
M.	DEBOURSE Jean-Pierre	S.E.S.
M.	DELATTRE Charles	Sciences de la Terre
M.	DURCHON Maurice	Biologie
M.	ESCAIG Bertrand	Physique
M.	FAURE Robert	Mathématiques
M.	FOURET René	Physique
M.	GABILLARD Robert	I.E.E.A.
M.	GRANELLE Jean-Jacques	S.E.S.
M.	GRUSON Laurent	Mathématiques
M.	GUILLAUME Jean	Biologie
M.	HECTOR Joseph	Mathématiques
M.	HEUBEL Joseph	Chimie
M.	LABLACHE COMBIER Alain	Chimie
M.	LACOSTE Louis	Biologie
M.	LANSRAUX Guy	Physique
M.	LAVEINE Jean-Pierre	Sciences de la Terre
M.	LEBRUN André	C.U.E.E.P.
M.	LEHMANN Daniel	Mathématiques
Mme	LENOBLE Jacqueline	Physique
M.	LHOMME Jean	Chimie
M.	LOMBARD Jacques	S.E.S.
M.	LOUCHEUX Claude	Chimie
M.	LUCQUIN Michel	Chimie

M.	MAILLET Pierre	S.E.S.
M.	MONTREUIL Jean	Biologie
M.	PAQUET Jacques	Sciences de la Terre
M.	PARREAU Michel	Mathématiques
M.	PROUVOST Jean	Sciences de la Terre
M.	SALMER Georges	I.E.E.A.
Mme	SCHWARTZ Marie-Hélène	Mathématiques
M.	SEGUIER Guy	I.E.E.A.
M.	STANKIEWICZ François	Sciences Economiques
M.	TILLIEU Jacques	Physique
M.	TRIDOT Gabriel	Chimie
M.	VIDAL Pierre	I.E.E.A.
M.	VIVIER Emile	Biologie
M.	WERTHEIMER Raymond	Physique
M.	ZEYTOUNIAN Radyadour	Mathématiques

PROFESSEURS 2ème CLASSE

M.	AL FAKIR Sabah	Mathématiques
M.	ANTOINE Philippe	Mathématiques
M.	BART André	Biologie
Mme	BATTIAU Yvonne	Géographie
M.	BEGUIN Paul	Mathématiques
M.	BELLET Jean	Physique
M.	BKOUCHE Rudolphe	Mathématiques
M.	BOBE Bernard	S.E.S.
M.	BODART Marcel	Biologie
M.	BOILLY Bénoni	Biologie
M.	BONNELLE Jean-Pierre	Chimie
M.	BOSQ Denis	Mathématiques
M.	BREZINSKI Claude	I.E.E.A.
M.	BRUYELLE Pierre (Chargé d'enseignement)	Géographie
M.	CAPURON Alfred	Biologie
M.	CARREZ Christian	I.E.E.A.
M.	CHAMLEY Hervé	E.U.D.I.L.

M. CHAPOTON Alain	C.U.E.E.P.
M. COQUERY Jean-Marie	Biologie
Mme CORSIN Paule	Sciences de la Terre
M. CORTOIS Jean	Physique
M. COUTURIER Daniel	Chimie
Mle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	E.U.D.I.L.
M. DEGAUQUE Pierre	I.E.E.A.
M. DELORME Pierre	Biologie
M. DEMUNTER Paul	C.U.E.E.P.
M. DE PARIS Jean-Claude	Mathématiques
M. DEVRAINNE Pierre	Chimie
M. DHAINAUT André	Biologie
M. DORMARD Serge	S.E.S.
M. DOUKHAN Jean-Claude	E.U.D.I.L.
M. DUBOIS Henri	Physique
M. DUBRULLE Alain	Physique
M. DUEE Gérard	Sciences de la Terre
M. DYMENT Arthur	Mathématiques
M. FLAMME Jean-Marie	E.U.D.I.L.
M. FONTAINE Hubert	Physique
M. GERVAIS Michel	S.E.S.
M. GOBLOT Rémi	Mathématiques
M. GOSSELIN Gabriel	S.E.S.
M. GOUDMAND Pierre	Chimie
M. GREVET Patrice	S.E.S.
M. GUILBAULT Pierre	Biologie
M. HANGAN Théodore	Mathématiques
M. HERMAN Maurice	Physique
M. JACOB Gérard	I.E.E.A.
M. JACOB Pierre	Mathématiques
M. JOURNEL Gérard	E.U.D.I.L.
M. KREMBEL Jean	Biologie
M. LAURENT François	I.E.E.A.
Mle LEGRAND Denise	Mathématiques
Mle LEGRAND Solange	Mathématiques (Calais)
Mme LEHMANN Josiane	Mathématiques

M. LEMAIRE Jean	Physique
M. LENTACKER Firmin	G.A.S.
M. LEVASSEUR Michel	I.P.A.
M. LHENAFF René	G.A.S.
M. LOCQUENEUX Robert	Physique
M. LOSFELD Joseph	I.E.E.A.
M. LOUAGE Francis	E.U.D.I.L.
M. MACKE Bruno	Physique
M. MAIZIERES Christian	I.E.E.A.
Mle MARQUET Simone	Mathématiques
M. MESSELYN Jean	Physique
M. MIGEON Michel	E.U.D.I.L.
M. MIGNOT Fulbert	Mathématiques
M. MONTEL Marc	Physique
Mme NGUYEN VAN CHI Régine	G.A.S.
M. PARSY Fernand	Mathématiques
Mle PAUPARDIN Colette	Biologie
M. PERROT Pierre	Chimie
M. PERTUZON Emile	Biologie
M. PONSOLLE Louis	Chimie
M. PORCHET Maurice	Biologie
M. POVY Lucien	E.U.D.I.L.
M. RACZY Ladislas	I.E.E.A.
M. RICHARD Alain	Biologie
M. RIETSCH François	E.U.D.I.L.
M. ROGALSKI Marc	M.P.A.
M. ROUSSEAU Jean-Paul	Biologie
M. ROY Jean-Claude	Biologie
M. SALAMA Pierre	S.E.S.
Mme SCHWARZBACH Yvette (CCP)	M.P.A.
M. SCHAMPS Joël	Physique
M. SIMON Michel	S.E.S.
M. SLIWA Henri	Chimie
M. SOMME Jean	G.A.S.
Mle SPIK Geneviève	Biologie
M. STERBOUL François	E.U.D.I.L.
M. TAILLIEZ Roger	Institut Agricole

M. TOULOTTE Jean-Marc	I.E.E.A.
M. VANDORPE Bernard	E.U.D.I.L.
M. WALLART Francis	Chimie
M. WATERLOT Michel	Sciences de la Terre
Mme ZINN JUSTIN Nicole	M.P.A.

CHARGES DE COURS

M. TOP Gérard	S.E.S.
M. ADAM Michel	S.E.S.

CHARGES DE CONFERENCES

M. DUVEAU Jacques	S.E.S.
M. HOFLACK Jacques	I.P..A
M. LATOUCHE Serge	S.E.S.
M. MALAUSSENA DE PERNO Jean-Louis	S.E.S.
M. OPIGEZ Philippe	S.E.S.

LA REDUCTION DU FOSSE SEMANTIQUE (détail).

Je tiens à remercier

Monsieur CARREZ, Professeur à l'U.S.T.L., de me faire l'honneur de présider le Jury.

Monsieur CORDONNIER, Professeur à l'U.S.T.L., qui a accepté de me prodiguer conseils et encouragements dans la réalisation de cette thèse.

Monsieur RENVOISE, du Centre de Recherche de la Compagnie des Machines BULL, qui m'a accueilli au sein de son équipe et qui a inspiré puis dirigé mes travaux avec compétence et sympathie.

Monsieur CHEVANCE, Ingénieur au CIT TRANSAC, dont les travaux personnels ont marqué et marquent encore le domaine dans lequel s'inscrit cette étude.

Monsieur DOMINE, Ingénieur D.R.E.T., d'avoir bien voulu participer au Jury.

Mes remerciements s'adressent également à Agnès BRADIER, Pierre DOUSPIS, Jean-Christophe COTTET, Marc LEGOUX, Daniel ROCACHER et Dominique SCIAMMA, tous membres du Service Transformations de Programmes pour le soutien technique et moral qu'ils m'ont apporté tout au long de cette thèse, ainsi que l'ensemble des personnes qui m'ont aidé au sein du Groupe BULL.

Je remercie également, au nom des lecteurs éventuels de ce rapport, mon épouse Brigitte qui a sacrifié une grande partie de ses loisirs à réaliser un support dactylographié de grande qualité.

Je remercie Madame DEBOCK d'avoir assuré la reprographie de ce document avec son soin et sa gentillesse habituels.

A Brigitte,

A Mes Parents,

SOMMAIRE

oooooooooooo

PREAMBULE

La réalisation de cette thèse s'est effectuée au sein du Service Transformation de Programmes du Centre de Recherche de la Compagnie des Machines BULL dans le cadre du projet MLI faisant l'objet des marchés 80-254 et 81-611 financés par l'Agence pour le Développement de l'Informatique.

Une partie des éléments d'architectures proposés dans ce rapport a fait l'objet de contributions aux lot 4 et 5 de ce contrat.

L'autre partie de la présente étude a pour principales fonctions :

1. La validation de l'approche adoptée pour la machine langage ADA* au travers d'une argumentation basée sur les résultats les plus significatifs rencontrés dans la littérature en ce qui concerne les nouvelles orientations de l'architecture des machines ;

2. Les premiers éléments de définition et de méthodologie préfigurant les caractéristiques de la réalisation matérielle qui devrait être la prochaine étape du projet.

* ADA is a Trade-Mark of the United States Government.

CHAPITRE 0

INTRODUCTION

0. INTRODUCTION

Les progrès réalisés en informatique, tant dans la fabrication des ordinateurs et la façon de les utiliser que dans leur pénétration dans tous les secteurs de l'activité humaine, apparaîssent souvent gigantesques.

S'ils le sont effectivement dans la plupart des domaines, et notamment dans l'évolution de la taille et de la puissance des ordinateurs, il semble toutefois que la conception de ceux-ci garde une marge de progression importante dans la mesure où, jusqu'à maintenant, la définition des architectures logiques et matérielles n'a pas su ou pu développer des principes qui, pour la plupart, existent depuis longtemps.

Ce dernier point, permet d'introduire une précision importante qui ne semble pas toujours comme allant de soi à de nombreux concepteurs de machines : il faut toujours bien différencier l'architecture logique de l'architecture matérielle (dont les définitions précises sont données au début du chapitre 1).

L'héritage de von Neumann

Cette confusion, volontaire ou non, a entretenu (à moins qu'elle n'en soit la conséquence) une situation de plus en plus remise en cause par bon nombre de constructeurs d'ordinateurs dans le domaine de l'architecture, à savoir la persistance du modèle de **von Neumann** qui existe depuis 1945.

Une remarque importante s'impose d'emblée : la critique des machines du type von Neumann, qui fait l'objet du 1er chapitre de cette étude, ne porte pas sur le modèle en tant que tel qui garde de sérieux atouts en ce qui concerne l'implémentation matérielle (notamment sous l'aspect micro-machine), mais sur la trop grande conformité de l'architecture logique à des principes de base du modèle de von Neumann qui engendre un décalage sémantique important avec les concepts désormais manipulés aux niveaux langages ou même système d'exploitation.

Ainsi l'évolution de la réalisation matérielle des ordinateurs a t-elle été surtout basée sur les progrès (énormes) de la technologie et sur des améliorations architecturales le plus souvent ne remettant pas en cause le modèle de von Neumann mais qui, nous le verrons, ne sont pas à écarter pour autant.

Objectifs poursuivis par les concepteurs de nouvelles machines

Avant d'aborder les principes qui, à notre avis, doivent guider toute investigation dans le choix d'une architecture logique, il est intéressant d'évoquer quels sont actuellement les principaux axes de recherche visant à adapter ou remplacer les schémas d'architecture matérielle utilisés dans la quasi-totalité des ordinateurs existants.

La reconsidération de la configuration matérielle des machines s'élabore chez les concepteurs autour de trois idées, développées le plus souvent séparément :

1. L'exploitation du parallèlisme d'exécution à tous les niveaux ;

2. L'établissement d'un mode de pilotage du déroulement d'un programme différent de celui, classique, de cadencement séquentiel par les instructions ;

3. L'augmentation du pouvoir du matériel dans la prise en compte des fonctionnalités traditionnellement logicielles.

Quelle que soit l'orientation qu'il choisit, un projet original d'implémentation matérielle est motivé généralement par une augmentation significative des performances par rapport à un produit existant et d'un coût matériel équivalent.

Il semble à peu près évident que l'obtention de gains importants en temps d'exécution proviendra dans une large mesure de la réalisation de **systèmes multi-processeurs** lorsque seront parfaitement résolus les problèmes que ceux-ci engendrent (répartition des tâches, synchronisation, communication entre processus,...) ; les nouveaux pilotages de l'exécution d'un programme (data-flow, mixte,...) ont la même ambition mais n'ont pas jusqu'ici apporté la preuve formelle de leur validité.

L'intégration dans le matériel de fonctionnalités habituellement prises en charge par le logiciel offre par contre un éventail plus large d'avantages.

Cette approche, qu'exploitent en particulier les architectures généralement appelées **machines-langage(s)**, laisse espérer (certains aspects sont même d'ores et déjà validés par l'expérience) de réelles améliorations dans l'utilisation d'un système informatique sous les aspects suivants :

- **L'accroissement des performances** (bien que dans un rapport moindre que celui fourni par un système multi-processeur) ;

- La **sûreté de fonctionnement** ;

- La **facilité d'utilisation.**

La justification de ces affirmations se trouve largement exposée dans le chapitre 2 ; l'argumentation avancée à cet effet ne repose pas uniquement sur l'idée, un peu simple et peu nuancée, que le matériel possède toutes les qualités de performances et de fiabilité que n'a pas le logiciel et qu'il suffit de câbler suffisamment pour résoudre tous les problèmes.

Comment aborder l'architecture logique et l'architecture matérielle

La conception d'une nouvelle machine doit commencer d'abord par une phase de définition de l'architecture logique en fonction des spécifications d'un cahier des charges **cohérent** (une machine facile à utiliser, très performante, parfaitement fiable et très bon marché a peu de chances de voir le jour !) ; l'implémentation matérielle est alors une deuxième phase de la conception qui ne doit apporter que des modifications mineures à l'architecture logique définie précédemment, sauf naturellement si les objectifs en coût de matériel et en performances ne sont pas atteints (voir plus loin).

La réalisation matérielle peut alors faire appel à toutes les techniques existantes, basées ou non sur le modèle de von Neumann, pour supporter efficacement le niveau logique sans en altérer les principes fondamentaux.

L'architecture matérielle doit être conçue avec d'autant plus de soin que souvent, elle permet la validation de concepts élaborés au niveau logique qui, s'ils étaient mal pris en compte, introduiraient d'importantes dégradations de performances ; le concept de mémoire virtuelle, par exemple, offre de nombreux avantages sans, en contrepartie, pénaliser trop lourdement le système qui dispose d'un topographe muni de mémoires associatives chargées d'accélérer le processus de traduction adresse-virtuelle-adresse réelle.

Rien n'oblige donc l'architecture logique à épouser trop fidèlement les mécanismes utilisés par le matériel et de prévoir, par exemple, des instructions machines explicites de recherche en mémoire, de mise en pile dans le cas d'appels de sous-programmes ou utilisant des registres programmables.

La machine M3L (voir § 2.1.1) est une bonne illustration de la complémentarité mais de la différenciation des architectures logique et matérielle ; le langage machine, accessible à l'utilisateur, est une forme arborescente de type LISP mais la machine qui implémente ce langage est en réalité le support d'un langage de microprogrammation, LEM, qui interprète le code arborescent.

Les limites de la technologie

Malheureusement, dans l'optique VLSI qui est désormais la tendance naturelle des constructeurs de machines, la technologie existante ou prévisible à court terme possède des limites ; et même s'il est permis d'envisager que les technologies futures offriront une capacité d'intégration qui permettra la plus grande liberté de solutions dans la complexité des fonctionnalités à implémenter, la conception d'une architecture logique doit encore aujourd'hui tenir compte a priori des contraintes physiques que lui imposent les moyens mis à sa disposition pour la réalisation de son support.

Ainsi l'**orientation-objets**, qui est à l'honneur chez beaucoup de concepteurs de machines, possède-t-elle de réelles qualités d'adéquation à la représentation d'un niveau logique structuré, consistant et fiable ; l'implémentation matérielle présente cependant des risques d'inefficacité (en termes de performances) lorsque les objets, pour des raisons faciles à imaginer d'affinage de la protection, sont choisis relativement petits.

L'**iAPX 432** est un exemple significatif des problèmes rencontrés dans la réalisation du support en VLSI d'une architecture s'écartant des schémas traditionnels ; son orientation objets, son système de protection par domaines et capacités, son jeu d'instructions à format variable sont autant d'options satisfaisantes sur le plan logique qui ont abouti à une implémentation peu satisfaisante, malgré l'expérience et la haute technicité de INTEL qui a décidé à cet effet de revoir complètement le dessin des circuits.

Comment supporter au niveau matériel la conception descendante

Pourtant, l'expérience informatique a fait plus que le confirmer, l'approche descendante, donc structurée et modulaire, est la méthodologie de conception qui offre les meilleures garanties de limitation des erreurs ; le problème est donc de conserver au niveau matériel l'aspect modulaire issu du modèle logique et correspondant à la définition structurée des différentes fonctionnalités sans que cela entraîne l'obtention de performances rédhibitoires.

Outre l'approche multi-processeurs qui est la solution naturelle et privilégiée pour une architecture répartie conçue à cet effet , la résolution de ce problème ne peut venir que d'une intégration maximale des fonctionnalités dans le matériel ; ceci permet en effet de ne pas "diluer" ces dernières dans une configuration matérielle qui ne respecterait plus la spécificité des différents modules et contribuerait ainsi à la perte de fiabilité que justement la conception descendante tente d'empêcher.

Il est important de souligner que cette orientation à la réalisation d'architectures matérielles n'est pas un but en soi, mais le moyen (qui devrait devenir de plus en plus efficace à mesure que progresse la capacité d'intégration de la technologie) adopté pour valider un choix précis d'architecture logique ; cette remarque est en effet motivée par la controverse actuelle sur les mérites respectifs des machines de types RISC (Reduced-Instruction-Set-Computer) et celles de type CISC (Complex-Instruction-Set-Computer) auquel appartiendront forcément les machines intégrant un maximum de fonctionnalités dans le matériel ; sans intervenir plus précisément dans ce débat disons simplement que l'approche RISC, qui a pour objectif d'obtenir une amélioration des performances par l'implémentation sur une puce d'un langage machine très simple mais dont les instructions ont un temps d'exécution (monocycle) très rapide, offre un mélange d'avantages et d'inconvénients qui ne semble pas pouvoir la destiner à un usage universel.

Rigidité du matériel et évolution du logiciel

L'intégration d'un maximum de fonctionnalités offre un inconvénient apparemment majeur : le manque d'adaptabilité à des modifications ou des ajouts éventuels au niveau logique (instructions, commandes systèmes, mécanismes de protection,...) qu'implique la rigidité réputée comme inévitable du matériel.

L'adaptabilité est en effet une qualité très recherchée de l'architecture ; ce phénomène est dû à la fois aux besoins de compatibilité entre systèmes ressentis par (presque) tous les utilisateurs et à l'évolution permanente des techniques informatiques dont la plus courante est l'implémentation d'un nouveau langage ; certains auteurs pensent même que, dans un souci de performances, il serait souhaitable de pouvoir adapter la machine au programme et non pas seulement au langage !

Il existe heureusement une technique d'implémentation matérielle qui contribue grandement à la résolution de ce problème : il s'agit de la **microprogrammation.**

La microprogrammation, contrairement aux idées généralement admises, est beaucoup plus qu'un simple mode de réalisation ; son évolution et les perspectives qu'elle laisse entrevoir au travers des différentes formes qu'elle revêt en font un outil privilégié dans la conception d'architectures originales et particulièrement adapté pour le type de machine qui fait l'objet de cette présentation.

Exposé des thèmes développés dans le présent rapport

La conception d'une nouvelle machine n'est donc pas soumise à des contraintes de méthodologie ou de solutions prédéfinies immuables (autres que celles qu'imposent la nature humaine....) mais doit opérer des choix bien adaptés aux objectifs qu'elle s'est fixés.

L'étude présentée dans ce rapport s'attache à illustrer et à défendre cette thèse grâce, d'une part, à une argumentation basée sur d'importantes recherches bibliographiques et d'autre part, à la description d'éléments concrétisant l'approche ainsi déterminée.

Ainsi, après un examen critique des architectures s'inspirant trop fidèlement du modèle de von Neumann, le chapitre 1 expose les problèmes qu'elles engendrent et conclut sur l'ensemble des spécifications souhaitables dans ce que nous avons appelé le nouveau cahier des charges des ordinateurs auxquels devrait faire face la majorité des concepteurs de nouvelles machines.

Le chapitre 2 offre une famille de solutions destinées à résoudre les problèmes évoqués dans le chapitre précédent ; comme le laisse supposer la présente introduction, l'orientation proposée, englobant la définition de l'architecture aussi bien logique que matérielle, est basée sur le principe d'une plus grande participation du matériel à la prise en compte des concepts évolués développés dans le cadre de l'architecture logique.

L'option ainsi adoptée pose le problème de la complexité du matériel qu'elle engendre nécessairement ; les techniques de microprogrammation y apportent une réponse d'autant plus satisfaisante que leurs progrès,

tant dans le domaine de leur utilisation que dans celui de leurs performances, minimisent considérablement ses inconvénients traditionnels ; cette étude fait l'objet du chapitre 3.

L'illustration concrète de l'approche introduite dans le deuxième chapitre en matière d'architecture de machines est fournie dans le chapitre 4 avec la présentation du projet MLI ; celui-ci a en effet pour objectif de répondre aux principales spécifications des nouveaux cahiers des charges (performances mais surtout fiabilité et facilité d'utilisation) à travers un schéma d'exécution (langage machine évolué) et des fonctionnalités demandant une grande capacité d'interprétation du matériel.

Le chapitre 5 propose d'ailleurs un exemple de l'importance du matériel, tant dans sa capacité de prendre en compte une fonctionnalité issue de l'architecture logique que dans son rôle de validation de cette fonctionnalité qui, comme cela a été précédemment évoqué, risque d'introduire une certaine inefficacité dans le schéma d'interprétation ; la définition plus précise du T.A.L. présentée dans ce chapitre permet d'établir une première évaluation en temps de fonctionnement et en coût de matériel qui justifie a priori l'adressage de type lexical qui est un des aspects fondamentaux de MLI.

Le 6ème chapitre et dernier chapitre, qui expose les enseignements tirés de cette étude et les principales composantes des futures phases de définition du projet ainsi que son évolution prévisible, forme la conclusion de ce rapport.

CHAPITRE 1

CRITIQUE DES ARCHITECTURES CLASSIQUES

1. CRITIQUE DES ARCHITECTURES CLASSIQUES

Ce paragraphe remet en cause les architectures qui s'inspirent trop fidèlement du modèle de von Neumann ; il montre à cet effet l'inadéquation de telles architectures à répondre valablement aux nouveaux cahiers des charges qui sont ou seront bientôt proposés aux constructeurs d'ordinateurs.

L'efficacité ne doit plus s'exprimer uniquement en termes de performances mais également sous les aspects de sécurité de fonctionnement et de souplesse d'utilisation.

1.1. PRELIMINAIRES

Le terme "**architecture**" employé tout au long de ce rapport désigne principalement la configuration logique d'un ordinateur ; il est pourtant souvent assimilé à celui de configuration matérielle dans le contexte des architectures classiques ou de von Neumann (§ 1.2 et 1.3), étant donné le caractère standard que prennent celles-ci.

L'architecture logique est l'organisation structurelle et fonctionnelle de la machine telle que la voient les utilisateurs de son langage machine (éventuellement de son langage de microprogrammation) et de son système d'exploitation.

L'architecture matérielle est le mode d'assemblage des différents composants logiques "évolués" (registres, mémoires, A.L.U., bus,...) qui supportent l'architecture logique.

Les termes de "machine" et "d'ordinateur" sont considérés comme équivalents et désignent la structure matérielle ; celui de "système informatique" englobe le logiciel supporté par le matériel.

1.2. DESCRIPTION ET EVOLUTION DE L'ARCHITECTURE DES ORDINATEURS

1.2.1. Le modèle von Neumann

Historiquement, John von Neumann est reconnu presque unanimement [Gol 72] comme étant celui qui proposa de mémoriser les programmes de calcul au même titre que les données dans l'ancien modèle de Babbage (figure n°1).

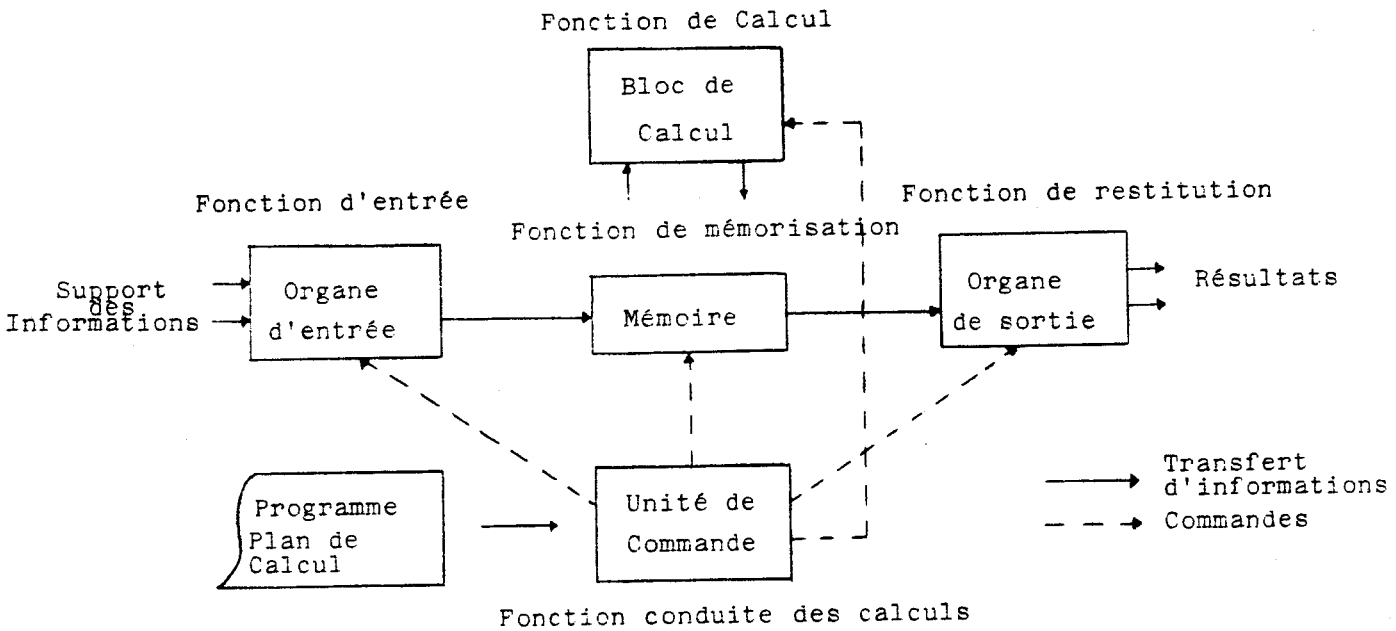


Figure N°1 : Structure de Babbage

Il introduisit ainsi la possibilité de ruptures de séquence automatiques dans les programmes avec branchement dans ces mêmes programmes désormais référencés en mémoire (figure n°2).

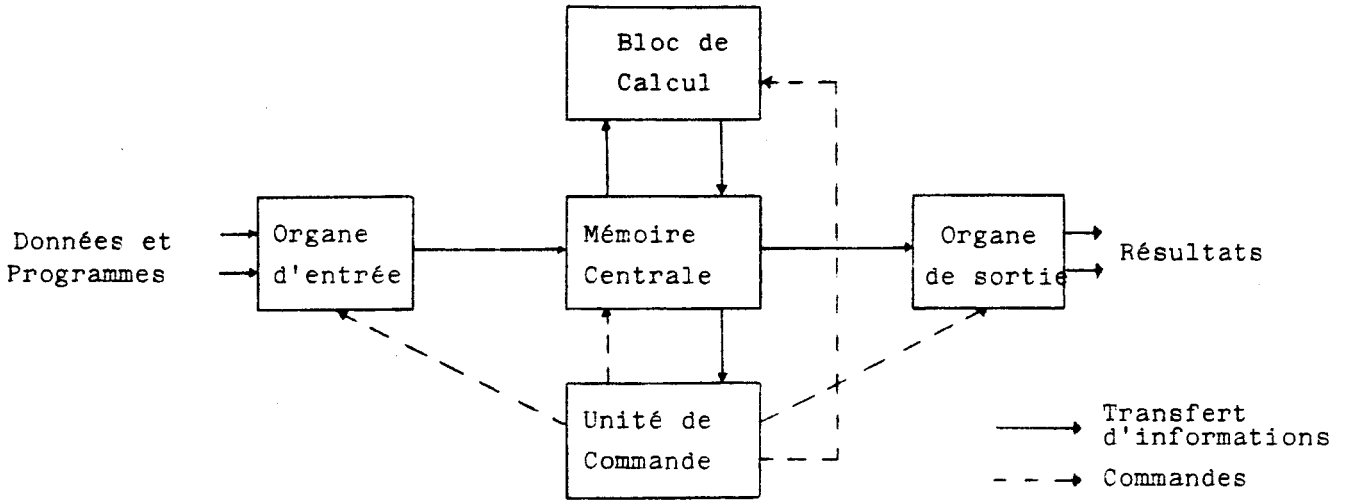


Figure n°2 : Structure de von Neumann

Le principe du compteur ordinal, et donc de l'exécution séquentielle (entre 2 branchements) des instructions du programme, reste la caractéristique principale d'une machine dite de von Neumann tout au moins telle qu'elle prédomine chez la plupart des informaticiens.

Une définition plus précise des caractéristiques d'une architecture basée sur les principes de von Neumann est donnée par MYERS [Mye 82] :

1. La mémoire est unique et adressée séquentiellement ;
2. La mémoire est linéaire ;
3. Il n'y a aucune distinction explicite entre les instructions et les données ;
4. Les données n'ont pas de parties permettant de les identifier.

La majorité des concepteurs de machines s'accordent pour reconnaître que la quasi-totalité des ordinateurs existant depuis 1945 ne sont que l'expression, sous des formes diverses, du même schéma de principe.

Il est également courant de définir les architectures de von Neumann par la forme de séquencement des programmes qu'elles supportent : **le cadencement par les instructions.**

Ce nouveau critère de classification est souvent confondu avec la séquentialité d'exécution des instructions qui n'en est que l'expression la plus répandue.

Il est en effet raisonnable d'imaginer que l'implémentation d'une machine originale supportant un langage unique de type non-procédural puisse se libérer de la séquentialité d'exécution des instructions alors que ce sont effectivement celles-ci qui assurent le déroulement du programme.

Avant de développer les aspects critiquables de l'évolution des architectures classiques* (§ 1.2.3), il paraît souhaitable de définir les différentes manières qu'ont les concepteurs de machines d'aborder actuellement l'architecture ; outre le supplément de précision qu'entraîne toujours une description suivant de nouveaux critères, cette présentation apporte surtout les preuves de la relative stagnation de l'évolution des architectures classiques.

1.2.2. Modes de conception et de caractérisation des machines

Les architectures classiques sont typiquement basées sur le principe de l'exécution d'un programme cadencée par les instructions.

* Le terme d'architecture classique a été préféré à celui d'architecture de von Neumann afin d'englober les machines offrant certaines caractéristiques différentes du modèle de base de von Neumann sans toutefois le remettre en cause.

La nature du cadencement de l'exécution d'un programme n'est devenue un critère de caractérisation que parce que d'autres types de "pilotage" sont apparus dans la conception d'architectures de machines :

- **Le pilotage par les données**, qui reste le plus étudié et qui engendre les machines dites "data-driven" ou encore "data-flow" ([DMi 74],...) ;

- **Le pilotage combiné (ou mixte)**, qui mélange le cadencement par les instructions et par les données ([Tou 79],...) ;

- D'autres types de pilotage qui abordent le cadencement sous un autre angle, en différenciant commandes et opérations par exemple ([Ren 82],...).

Il existe deux autres aspects fondamentaux de redéfinition des architectures auxquels s'attachent les principaux axes de recherche actuels :

- La reconnaissance et l'exploitation du parallèlisme dans l'exécution d'une application, allant du niveau système d'exploitation ("overlaps") au niveau microcommandes et qui a donné naissance par exemple aux machines multiprocesseurs ;

- L'augmentation de la part prise par le matériel dans le schéma d'interprétation des langages ou des concepts de haut niveau ; ce dernier point est en partie justifié par le gain en performances attendu mais surtout par la fiabilité désormais acquise par le matériel.*

Ces trois aspects caractéristiques d'une architecture -type de pilotage de l'exécution d'un programme, degré(s) de parallèlisme exploité, niveau d'intégration matérielle- ne sont naturellement pas exclusifs et bon nombre de nouvelles machines combinent les évolutions techniques issues de ces trois approches de conception architecturales.

* Des calculs prévisionnels utilisant la loi de Duane, ont permis d'estimer à 8 500 heures le MTBF (Mean Time Between Failure) de l'unité centrale du DPS 7/70 de CII Honeywell Bull sous forme micropackagée.

Les architectures classiques n'ont pas encore définitivement assimilé et implémenté les résultats issus de ces trois axes fondamentaux de la recherche actuelle ; les éléments qui ont constitué leur évolution sont en réalité d'un autre ordre.

La présente étude ne concerne que les machines à cadencement par les instructions, s'attache essentiellement au schéma d'interprétation et, dans le cadre de la critique des machines traditionelles qui suit, n'explicite pas l'exploitation du parallélisme de haut niveau pratiquement toujours inexistante.

1.2.3. Evolution des architectures classiques

Si l'on écarte les progrès de la technologie, cause principale de l'accroissement des performances mais d'une influence mineure sur la structure des architectures classiques, les transformations subies par celles-ci se résument en fait à deux types :

- d'une part, l'implémentation de concepts apparus avant 1960 ;
- d'autre part, l'amélioration des performances grâce à des techniques ne remettant pas en cause le modèle de von Neumann.

Le premier point fait allusion à certains aspects désormais courants dans la plupart des calculateurs mais qui ne font pas figure de véritable innovation.

A titre d'exemple, citons la microprogrammation (1951), la représentation en flottant (1954), la mémoire virtuelle (1959) ou l'adressage indirect (1958).

Or, pour reprendre l'avis de Dennis [Den 76], "d'importantes améliorations dans la méthodologie de développement de très grands programmes ne pourront avoir lieu en l'absence de changements importants dans la structure matérielle et l'organisation des systèmes informatiques".

Le deuxième type d'améliorations concerne des innovations architecturales comme les antémémoires, les mémoires associatives, les techniques d'anticipation dans le cas des ruptures de séquence ou même le traitement en pipeline.

Ces différentes techniques, non seulement ne remettent pas en cause le modèle de von Neumann, mais souvent, en tirent leur existence ; elles ont en effet pour principe d'accélérer non pas le traitement dans sa globalité mais la circulation des informations sur les différents chemins de données participant au support du traitement.

L'exemple du traitement en pipe-line est significatif dans le sens où ce concept n'est pas reconnu unanimement comme étant un schéma d'exécution radicalement différent ; il est plutôt considéré comme une technique d'implémentation applicable à de nombreux types d'architectures réputées différentes (SISD d'après HOCKNEY, SIMD d'après FLYNN ou MIMD d'après GORSLINE suivant la classification de FLYNN [HJe 81]).

Le fait que la plupart des machines utilisées (ou même conçues) actuellement intègrent des principes relativement anciens ne serait pas en soi critiquable s'il n'engendrait un certain nombre de problèmes que les architectures classiques arrivent peu ou pas à résoudre quand elles n'en sont pas directement la cause.

1.3. PROBLEMES ENGENDRES OU NON RESOLUS PAR LES ARCHITECTURES CLASSIQUES

Le principal problème, qui est en réalité un ensemble de difficultés rencontrées à tous les niveaux par l'utilisateur d'un ordinateur classique, est connu sous le nom de **fossé** ou **d'écart sémantique** (en anglais "semantic gap") et a été mis en évidence par GAGLIARDI [Gag 73].

Le fossé sémantique peut être défini comme étant la distance abstraite existant entre les concepts manipulés par les langages de haut niveau et ceux pris en compte par le matériel d'une machine.

Le décalage est évidemment croissant à mesure que s'élève le niveau des langages évolués ; de plus, dans le cadre des ordinateurs dotés d'un langage machine de bas niveau et d'une architecture orientée von Neumann, il a pour principales conséquences de :

1. Contribuer à l'accroissement du coût de développement et de mise au point du logiciel en augmentant la taille et la complexité des compilateurs à travers ses deux composantes principales, l'analyseur et le générateur de code ; l'analyseur voit ainsi son travail augmenter en raison des concepts de plus en plus évolués introduits par les langages de haut niveau (structures de données, contrôles...) ; les conséquences sur l'augmentation de la taille du générateur de code sont évidentes, d'autant plus lorsque le code machine est rudimentaire.

2. Introduire une dégradation de performance en augmentant la taille du code généré, souvent difficilement optimisable, en raison de la complexité des concepts à traduire évoqués dans le 1) ; la taille d'un programme est en effet un critère d'efficacité, ainsi que l'a fait apparaître une série d'études effectuées aux Etats-Unis [Bur 77] (une "indication importante sur l'adéquation d'une architecture pour une application donnée (programmes test) est la place en mémoire nécessaire à la représentation")*.

3. Détériorer la communication homme/machine en compliquant la mise au point des programmes soit dans le cas d'utilisation de programmes spéciaux de "debugging" qui posent les mêmes problèmes que les autres programmes de développement (conception difficile, erreurs possibles), soit dans le cas de lecture directe du code machine où il est très difficile de retrouver la sémantique du texte source (surtout quand celui-ci est passé par une phase d'optimisation du compilateur) ; de plus, le recours à l'examen des ressources matérielles demande au metteur au point l'acquisition de compétences supplémentaires (connaissance de l'architecture interne de la machine) tandis que la spécificité du matériel est un obstacle à sa polyvalence (la mise au point varie souvent énormément suivant les modèles d'ordinateurs) ;

* Le gain en performance qu'implique la compaction du code ne fait pas l'unanimité chez les concepteurs de machine ainsi que cela est souligné dans [KBB 82] ; les arguments en faveur de la compaction sont pourtant loin d'être négligeables et sont développés dans le § 2.1.2.1.

4. Ne pas répondre efficacement aux nouvelles exigences de sécurité et de protection du fonctionnement des systèmes informatiques en imposant au logiciel de prendre en compte les structures et les mécanismes de contrôle que n'offre pas le matériel (reconnaissance des données en mémoire, contrôles automatiques...) ; cela a pour conséquence de rendre la sûreté du fonctionnement sujette aux défaillances inhérentes à la complexité du logiciel et de permettre, grâce au code machine de bas niveau, la reconnaissance et l'accès à la représentation interne des objets manipulés par les programmes ou le système d'exploitation.

Les architectures classiques posent donc des problèmes difficiles à résoudre (mise au point des programmes, pertes de performances dues à la taille du code objet engendré par compilateur, contrôles implicites) ou même impossibles (protection de la représentation interne des objets).

Deux importants griefs peuvent être également formulés à l'encontre des architectures classiques :

- Si le schéma de principe de von Neumann n'est pas remis en cause, l'accroissement des performances ne peut venir à terme que de la seule évolution de la technologie (limite physique) ;
- Leur caractère figé et procustéen*, outre la non-portabilité qu'il engendre souvent, les prépare mal aux continuelles évolutions demandées aux systèmes informatiques (limite théorique).

La réalisation de nouvelles architectures doit donc s'attacher à résoudre les problèmes qui viennent d'être évoqués en visant des objectifs clairement définis.

* Le terme procustéen est dû à WILNER [Wil 72] qui a comparé les architectures classiques avec le brigand de l'Attique, PROCUSTE qui, après avoir détroussé les voyageurs, les mettaient sur un lit de fer en étirant les plus petits et en mutilant les plus grands afin de leur donner la taille exacte du lit ; les instructions et données manipulées par la machine sont les voyageurs destinés au lit de fer que représente la taille fixe des registres et des éléments de mémoire.

1.4. OBJECTIFS GENERAUX FIXES AUX NOUVELLES MACHINES

Quelles que soient les applications auxquelles est destinée une machine, sa réalisation doit offrir une **efficacité maximale.** Dès lors, la motivation principale à la conception de la plupart des nouvelles machines reste l'accroissement des performances par rapport aux modèles existants.

Il semble toutefois que la notion d'efficacité doive, à l'avenir, ne plus s'exprimer uniquement en termes de puissance de calcul accrue (comme elle l'a été jusque maintenant), mais que d'autres aspects deviennent importants voire indispensables :

- La fiabilité des moyens fournis par la machine dans l'aide à la construction et à la mise au point des programmes (pas de vérifications inutiles en cas d'erreur) ;
- La facilité d'utilisation de ces moyens et de l'ensemble des commandes des systèmes d'exploitation (diminution des erreurs de manipulation et du temps de formation de l'utilisateur).

Ainsi, en réduisant les coûts de développement, de mise au point et d'évolution du logiciel qui deviennent insupportables,* l'effort porté sur deux aspects de la **sûreté de fonctionnement** et de la **communication homme/machine** apporte un gain de productivité chez les programmeurs (logiciel de base ou applications) qui est un élément non négligeable d'efficacité dans l'utilisation de la machine.

La recherche des performances n'est donc qu'un aspect de l'efficacité globale d'un système informatique ; les moyens pour assurer cette efficacité doivent donc être désormais les objectifs poursuivis en priorité par de nouveaux types d'architecture, à savoir :

* En 1978, le coût de développement, de mise au point et de maintenance du logiciel employé par le Gouvernement des Etats-Unis était estimé à environ 8 milliards de dollars, incluant environ 4 milliards pour la Défense ; l'US Air Force a elle seule a dépensé environ 80 à 90 % du coût total d'acquisition de systèmes informatiques pour le logiciel, contre 15 % en 1955 [Chi 80].

1. La sûreté de fonctionnement, à travers ses composantes :

- La fiabilité du matériel (principal effort jusqu'à maintenant) ;
- La tolérance aux pannes matérielles ou logicielles (fonctionnement en mode dégradé) ;
- La résolution des problèmes de protection (intégrité des objets et droits d'accès) ;
- L'intégration d'un maximum de concepts et de technique facilitant l'aide à la mise au point de programmes (programmation en langages évolués, implémentation matérielle d'outils de mise au point, facilité d'utilisation de la machine...).

2. La facilité d'utilisation à tous les niveaux :

- Programmation en langage évolué ;
- Simplicité du système d'exploitation (à la conception et à l'utilisation) ;
- Mise au point de programmes agréable et ne demandant pas une connaissance approfondie de la structure interne de la machine ;
- Maintenance facile en cas de défaillances matérielles ou logicielles ;
- Facilité de mise à jour des programmes en fonction de l'évolution des cahiers des charges ou de la configuration matérielle.

3. La capacité d'extension et d'adaptation pour répondre aux exigences actuelles :

- La portabilité des programmes et donc la compatibilité des systèmes (en raison du coût élevé du logiciel) ;
- La capacité de supporter efficacement de nouveaux langages et de nouveaux environnements de programmation évolués (afin de prolonger la durée de vie d'un système).

Nous allons examiner maintenant les orientations que doit suivre la conception d'architecture pour respecter le plus efficacement possible le cahier des charges qui vient d'être détaillé.

CHAPITRE 2

POUR DE NOUVELLES ORIENTATIONS A LA CONCEPTION D'ARCHITECTURES

2. POUR DE NOUVELLES ORIENTATIONS A LA CONCEPTION D'ARCHITECTURES

Ainsi que l'expose l'introduction (Chapitre 0), l'amélioration importante des performances d'un ordinateur peut être principalement attendue dans l'exploitation systématique (voire intensive) du parallélisme à l'exécution des programmes grâce aux machines multi-processeurs (parallèles ou data-flow) ; l'objet de ce chapitre est, quant à lui, de présenter les différentes stratégies de conception (ainsi que les mécanismes d'implémentation qui en résultent) qui apportent de réelles solutions aux problèmes évoqués dans le paragraphe précédent à savoir la sûreté de fonctionnement, la facilité d'utilisation et la capacité d'extension et d'adaptation.

La présentation des différents schémas d'exécution d'un programme est, dans un premier temps, l'occasion d'introduire les types d'architectures que recouvre le terme de "**machine-langage(s)**". Une première forme de réduction du fossé sémantique, celle à laquelle s'attachent plus particulièrement les machines-langage(s), est ensuite développée, d'une part, à travers le choix du **langage-machine** et d'autre part, par l'intermédiaire des machines dites "**orientées-objet**".

Fiabilité et efficacité ont amené les concepteurs de machines à considérer avec acuité les problèmes vitaux posés par l'implémentation des **logiciels de base** et en particulier, des **systèmes d'exploitation.** A un rapide panorama des techniques utilisées plus ou moins classiquement dans les ordinateurs actuels, succède l'exposé des principaux mécanismes d'accès et de protection qui, par leur bonne intégration dans le matériel, répondent efficacement aux besoins, jusqu'ici mal pris en compte, des systèmes d'exploitation.

Enfin, sont présentés brièvement les autres domaines où peut s'exercer une prise en charge dès la conception architecturale (logique et/ou matérielle).

L'aspect performances, il est bon de le rappeler, reste important dans les motivations d'une nouvelle architecture.
C'est pourquoi, dans toutes les descriptions relatives aux différents éléments ou formes d'architecture développés dans ce paragraphe, figurent les éventuelles influences qu'entraîne leur adoption sur le gain (ou la perte) présumé de performances tout autant que sur les bénéfices attendus en matière de sûreté de fonctionnement, d'amélioration de la communication homme/machine ou même de facteur d'adaptabilité.

2.1. REDUCTION DU FOSSE SEMANTIQUE DU AUX LANGAGES EVOLUES

Le fossé sémantique dû aux langages évolués est celui évoqué au paragraphe 1.3 ; un autre "type" de fossé sémantique est décrit plus loin (§ 2.2) dont la réduction peut également en partie être assurée par le choix d'un nouveau schéma d'exécution et/ou par l'adoption d'une architecture "orientée-objet".

2.1.1. Les différents schémas d'exécution d'un programme

Classiquement, l'utilisateur d'un ordinateur se voit proposer deux niveaux de langage pour l'exécution des travaux qu'il veut confier à la machine (nous excluons pour l'instant le cas des machines microprogrammables) :

1. Le niveau langage évolué, qui est généralement très bien adapté à un ou plusieurs types d'applications (scientifique, base de données, graphique, temps réel...) mais volontairement indépendant de toute contrainte matérielle afin d'améliorer, par définition, la facilité de conception des programmes ainsi que leur portabilité ;

2. Le niveau langage machine, qui est par contre totalement dépendant de la machine et possède les caractéristiques inverses des langages de haut niveau : difficulté de création de programmes volumineux, manque de portabilité, absence de protection.

L'effort accompli dans la fabrication des ordinateurs ayant surtout porté sur la technologie (cf § 1.1), le niveau sémantique des langages machine est resté traditionnellement proche de celui des premières machines de von Neumann.

Les problèmes liés à la traduction des programmes écrits en langage évolué en code objet exécutable par la machine ont déjà été évoqués précédemment : c'est le fossé sémantique.

Pour réduire ce fossé sémantique, il existe deux méthodes empruntées à la Palice :

1. Baisser le niveau du haut : cela signifierait l'abandon des langages évolués et irait contre la tendance à l'amélioration de la communication homme/machine par convergence vers les langages naturels ;

2. Elever le niveau du bas : il s'agit de fournir des machines dont le langage de base soit le plus proche possible du ou des langages évolués destinés à être supportés par ces machines.

La première solution, sauf cas fortement spécifique, serait une régression non envisageable dans la conception des machines puisqu'elle aggraverait dans des proportions aujourd'hui rédhibitoires le coût d'un logiciel déjà beaucoup trop lourd.

La deuxième solution offre l'alternative suivante dans le choix du langage machine :

1. En faire un **langage intermédiaire** d'un niveau sémantique relativement élevé ;

2. Utiliser directement le langage évolué comme langage de base de la machine.

Afin de définir la hiérarchie des langages-machine qui, dans une large mesure, permet de préciser la notion de machine-langage, il est nécessaire de rappeler les schémas possibles d'exécution d'un programme.

Les différents schémas d'exécution d'un programme apparaissent dans la figure n°3 où sont regroupés les différents types de représentation d'un programme tel qu'il évolue au cours du traitement que lui fait subir la machine en vue de son exécution.

Il s'agit évidemment d'une synthèse sur l'ensemble des machines existantes :

- Chaque case correspond à un niveau de représentation d'une application ; l'assimilation à ces niveaux des différentes formes de représentation est explicitée suivant les types de machines (voir exemples) ;

- La forme d'une représentation n'implique pas que celle-ci puisse exister dans sa totalité à un instant donné et/ou à un endroit précis où elle pourrait être isolée ou recopiée (cas de l'interprétation) ;

- Chaque machine ne possède qu'un ou deux, voire trois, rarement plus, schémas d'exécution compatibles au sein d'une même architecture (voir exemples) ;

- La transition d'une forme à une autre peut varier suivant les machines ; une instruction en code machine peut être traduite en un micro-programme de manière câblée (passage par un décodeur câblé comme dans l'iAPX 432) ou par un interpréteur micro-programmé (machine M3L, voir plus loin).

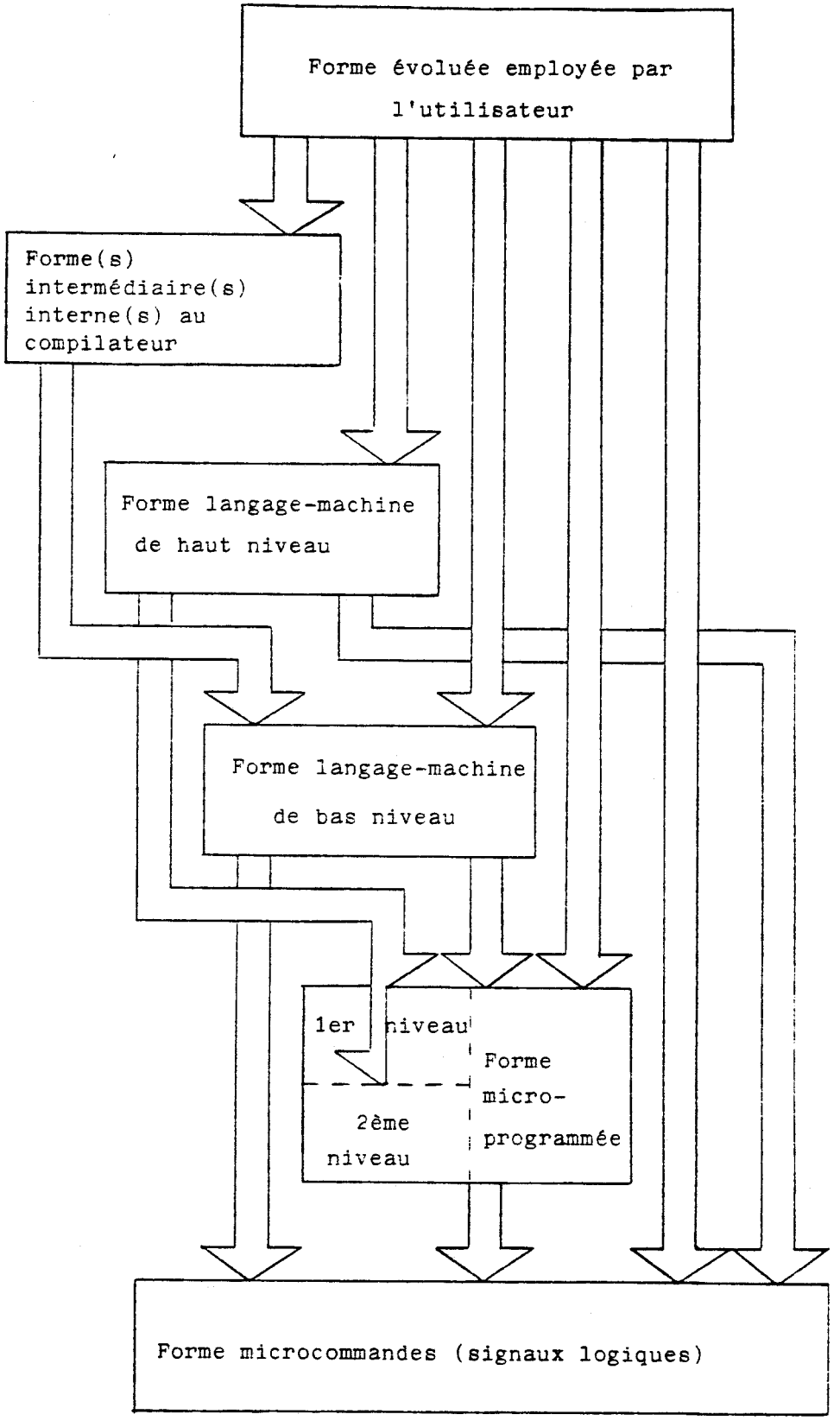


Figure n°3 : Niveaux de représentation d'un programme dans une machine

Examinons maintenant les principaux schémas d'exécution en les illustrant par des exemples précis.

a) La quasi-totalité des machines existantes assurent le traitement des programmes de la façon suivante (figure n°4a) :

- Traduction du programme écrit en langage évolué sous la forme d'instructions en code machine ; cette traduction peut s'effectuer soit sous la forme d'une **compilation** (avec passages éventuels sous différentes formes intermédiaires) qui engendre un texte-objet défini dans sa totalité, soit sous la forme d'une **interprétation** qui analyse et traduit le texte-source, instruction par instruction ;

- Exécution du code-machine de faible niveau sémantique (LOAD, STORE, JUMP...) après un décodage câblé ou par interprétation micro-programmée.

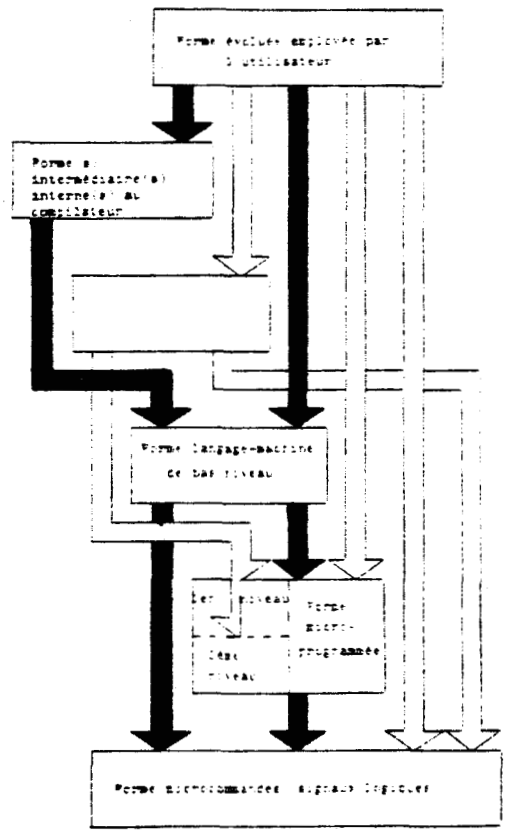


Figure n° 4a

Une même machine peut supporter à la fois le mode compilé et le mode interprété : la plupart des micro-ordinateurs ont un interpréteur BASIC résidant en ROM mais peuvent utiliser des compilateurs BASIC disponibles sur le marché.

Il est toutefois important de signaler que le code machine engendré par la compilation d'un texte source donné est très différent de celui qui correspondrait à l'ensemble des instructions machines résultant de l'interprétation de ce même texte-source (aucune optimisation statique n'est possible).

b) Une distinction préalable peut être faite concernant les machines disposant d'un langage machine de haut niveau -encore appelé intermédiaire- et réalisant des schémas d'exécution directe (figure n°4b) :

- Soit le langage intermédiaire est de si haut niveau qu'une simple opération d'assemblage suffit pour transiter de la forme langage évolué en forme langage machine ; la phase d'assemblage peut s'effectuer de manière logicielle ou également matérielle, comme c'est le cas dans le système SYMBOL [RSm 71] développé par Fairchild Camera et Instrument Corporation ;

- Soit le langage intermédiaire demande une véritable compilation (quoique moins complexe que celles nécessaires classiquement).

Ce dernier cas correspond aux machines dites "**orientée-langage**", traduction de l'expression "language-directed" due à Mc KEEMAN [Mck 67].

Deux phénomènes importants favorisent la faisabilité de ce type de schéma d'exécution :

- Les compilateurs actuels de langages évolués (PL/1, ADA...) engendrent un à plusieurs niveaux de langages successifs avant la génération du code : la forme de ces langages intermédiaires reste plus ou moins évoluées tout en facilitant l'implémentation finale ;

- Les progrès de la microprogrammation (§ 3.3.1.1) permettent de faciliter l'interprétation d'instructions plus riches sémantiquement que celles des habituels codes-machine.

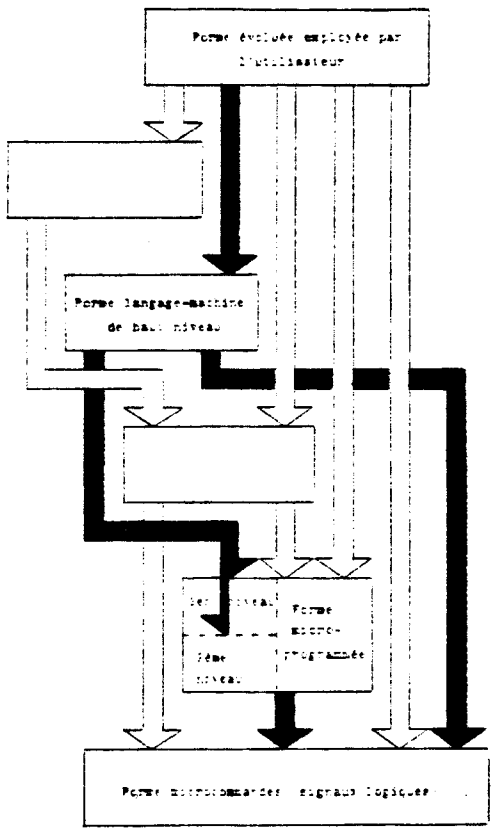


Figure n°4b

Ce nouveau schéma d'exécution se définit comme la réalisation de deux étapes consécutives :

1. La traduction d'un texte source écrit en langage évolué par un compilateur (ou un interpréteur) en une représentation intermédiaire écrite en langage machine relativement évolué : la phase de compilation (encore appelée précompilation) se distingue de ses équivalentes sur machine classique par l'élimination quasi-complète de l'étape peu formalisée de génération du code de bas niveau, et parfois, par la possibilité de retraduction de la forme intermédiaire en texte source (décompilateur dans M3L).

2. L'exécution de la forme intermédiaire qui peut s'effectuer de manière entièrement câblée, mais plus souvent par interprétation microprogrammée : à chaque instruction de langage machine évolué est associé un microprogramme.

De nombreuses études, n'ayant pas toujours abouti à une réalisation physique, se sont inspirées à des degrés divers de ce principe. Les machines effectivement implémentées appartenant à cet ensemble sont certainement celles qui ont contribué le plus à la notoriété des machines-langage(s).

La plupart des machines pouvant être apparentées à cette famille sont destinées à supporter un langage unique de programmation. Parmi les plus illustres, citons par exemple :

- Les machines orientées ALGOL, celles de Burroughs (B5500/6500/6700/7600) [Ear 80] ou la machine prototype développée à l'Université de Manchester, le système MU5 [Ica 78] ;

- Les machines orientées COBOL, avec deux machines commercialisées, le Burroughs 3500 et le récent NCR Criterion (qui a 2 architectures dont une orientée COBOL, [Tof 78, Sha 78], ainsi que la machine expérimentale de CHEVANCE [Che 73] ;

- Les machines orientées PASCAL, avec une machine construite à Grenoble à base de microprocesseurs en tranches, PASC-HLL [Sch 77] ; il faut remarquer que les P-codes [Naj 75] ou autres Q-codes [Per 81] sont aussi de bons exemples, sur le principe, de langages intermédiaires évolués ;

- Les machines orientées LISP, comme celle réalisée au MIT, la CONS machine [Sch 78].

Cette liste n'est évidemment pas exhaustive pour chaque langage évolué retenu, ni quant aux autres langages supportés (PL1, APL, BALM,...) ; ces compilations peuvent être retrouvées dans [Mye 82, Ste 77].

Il existe d'autres architectures empruntant le schéma d'exécution de type b) qui n'ont pas toutefois un seul langage cible.
Parmi elles se distinguent les architectures n'ayant qu'un seul langage intermédiaire et celles qui en supportent plusieurs.
Dans le premier cas, le langage intermédiaire reste suffisamment souple pour intégrer un maximum de concepts évolués communs à une grande majorité de langages de programmation de haut niveau ; c'est le choix qu'ont réalisé des machines comme le Rice Research Computer [Feu 72], l'IBM 38 [IBM 78], SWARD [Mye 77, Mye 80a, Mye 80b], la machine de BATTAREL et CHEVANCE [BCh 79] ou l'iAPX 432 [INT 81].
Dans le second cas, les langages intermédiaires sont mieux adaptés aux différents langages (ou familles de langages) auxquels ils correspondent ; l'exemple le plus illustre est bien sûr le Burroughs B 1700 [Wil 72a] qui possède plusieurs "S-langages" qu'il peut interpréter simultanément grâce à une microprogrammation dynamique ; la machine M3L [Per 81, Cas 80] construite à base de microprocesseurs en tranches à l'Université de Toulouse a adopté également ce principe et dispose de 2 langages intermédiaires, un orienté LISP et un orienté PASCAL ; cette solution avait également été retenue par LECUSSAN dans sa définition d'un émulateur généralisé [Lec 77a], puis de sa machine-système de programmation LTR [Lec 82].

Remarque :

l'IBM 38 fait partie des machines à deux niveaux de microprogrammation (le vertical VML et l'horizontal HMC) et offre l'exemple d'une machine possédant un schéma d'exécution "mixte".

En effet, chaque texte source (langage évolué) est compilé dans le "Instruction Interface Code" (langage intermédiaire) qui lui-même est compilé en IMPI (langage de microprogrammation) : la plupart des instructions en IMPI correspondent à des instructions du VMC mais certaines sont des appels de sous-microprogrammes en HMC.

c) Certains auteurs, comme CHU [Chu 75], estiment que la machine doit réaliser toutes les étapes de transformation du langage source sans faire appel à une quelconque phase logicielle.

Le langage évolué est directement interprété soit de manière microprogrammée, soit de manière câblée (figure n°4c).

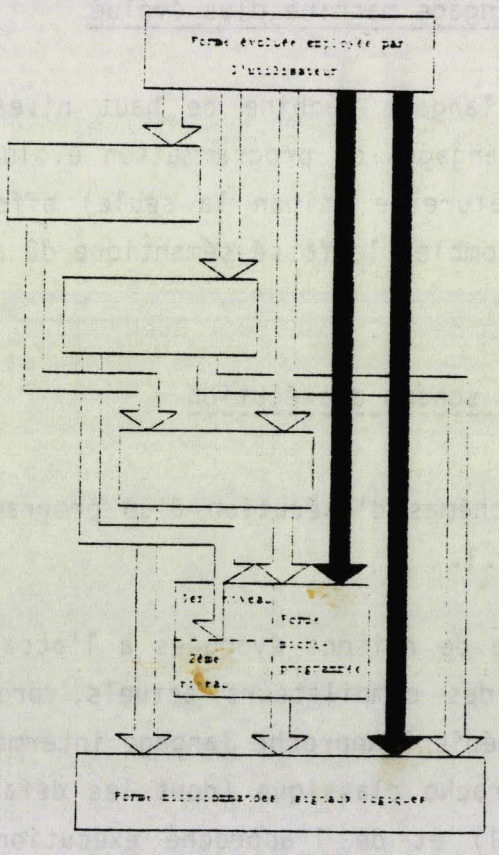


Figure n°4c

Bien que dans ce cas, le fossé sémantique ait disparu, ce type d'architecture pose des problèmes de vérification syntaxique et surtout de réalisation matérielle : en dehors de certains langages directement interprétables (BASIC, APL, LISP,...), la plupart des langages évolués impliquent un support matériel d'une complexité facile à imaginer.
De plus, comme le souligne DITZEL [Dit 80], ce type de schéma d'exécution spécialise beaucoup trop la machine, condamnée au langage unique (l'implantation de compilateurs d'autres langages pose dans ce cas de réels problèmes d'efficacité).

Parmi les réalisations (néanmoins) existantes, se trouve un produit comercialisé, l'IBM 5100/5110, qui interprète matériellement les programmes APL (en fait, le processeur central interprète matériellement un interpréteur logiciel APL qui interprète effectivement le programme en APL).

2.1.2. Vers un langage machine plus évolué

L'adoption d'un langage machine de haut niveau, éventuellement du même niveau que les langages de programmation évolués, est sans conteste la méthode la plus naturelle (sinon la seule) offerte au concepteur de machines soucieux de combler le fossé sémantique dû aux langages évolués.

2.1.2.1. Choix du schéma d'exécution

Les différents schémas d'exécution d'un programme ont été présentés dans le paragraphe 2.1.1.

Un certain nombre de raisons évoquées à l'occasion de cette présentation (organisation des compilateurs actuels, progrès de la microprogrammation), privilégie l'approche langage intermédiaire (figure n°4b), au détriment de l'approche classique (dont les défauts ont été largement exposés au chapitre 1) et de l'approche exécution directe (complexité extrême de l'architecture dans le cas de langages très évolués).

Ainsi, excepté peut-être pour certains langages se prêtant relativement bien à l'interprétation (BASIC, APL, LISP,...), le compromis le plus acceptable entre la meilleure adaptation de la machine au haut niveau des langages actuels (efficacité de fonctionnement) et une réalisation ni trop complexe et ni trop figée (efficacité de conception et d'évolution) apparaît être le choix d'un langage machine de niveau sémantique relativement élevé.

La justification de ce compromis passe donc par la réfutation des deux objections majeures qu'il fait naître :

1. L'adoption d'un jeu d'instructions de haut niveau sémantique peut le rendre trop dépendant d'un langage unique et pose le problème du support multi-langages de la machine ;

2. La puissance sémantique des instructions les rendent plus longues que les instructions machines traditionnelles en nombre de bits et en temps d'exécution, ce qui pose le problème des performances.

Des exemples de machines existantes évoquées précédemment (§ 2.1.1 b) permettent de minimiser le premier point : la richesse du jeu d'instructions ou la microprogrammation dynamique sont deux types de solutions concrètes répondant parfaitement aux besoins d'une programmation multi-langages.

Il reste cependant admis qu'en termes de performances, la meilleure efficacité est obtenue lorsque la machine est conçue pour un langage cible unique ou même un ensemble de langages appartenant à une même famille.

Ce problème des performances est souvent soulevé au travers des critiques évoquées dans le deuxième point ; il est en fait le coeur de la polémique qui oppose les partisans de RISCs (Reduced Instruction Set Computer) et ceux du CISCs (Complex Instruction Set Computer).

Certains projets de machines mono-puce réalisées en VLSI tels que MIPS [HJP 82] ou RISC I [PSe 81] défendent la thèse de l'accroissement de la vitesse d'exécution d'un programme sur microprocesseur par une réalisation matérielle optimisée de quelques instructions simples mais très rapides.

L'absence de tests véritablement significatifs ne permet pas pour l'instant de valider cette approche qui s'oppose aux tendances actuelles de la conception de microprocesseurs..

Les machines de type RISC garderont cependant toujours deux des principaux inconvénients des architectures classiques :

- La nécessité de compilateurs complexes (vérifié en particulier pour MIPS) ;
- L'amélioration des performances par les seuls progrès de la technologie.

La position d'un jeu d'instructions machines sémantiquement riches vis-à-vis du problème des performances s'établit autour de deux idées d'ordre général :

1. La compaction de code permet d'avoir un débit réduit de recherche d'instructions en mémoire tandis que la richesse sémantique des instructions favorise l'exploitation du parallélisme d'exécution ; la compaction de code peut donc être un facteur important d'amélioration globale des performances et, en tout état de cause, n'est pas un facteur de dégradation de ces performances ;*

* Une étude comparative réalisée par des ingénieurs de SPERRY UNIVAC et portant sur les trois principaux microprocesseurs 16 bits (INTEL 8086, ZILOG Z8000, MOTOROLA MC 68000) révèlent que, malgré la plus grande longueur des instructions du Z 8000 et du MC 68000 due à leur meilleure flexibilité et à un adressage plus complexe, le volume de code engendré pour ces deux microprocesseurs (pour une tâche donnée) est équivalent, voire inférieur (suivant les tâches), à celui engendré pour le 8086 et ceci, pour des performances équivalentes [BBC 80].

2. La justification majeure reste la réduction du fossé sémantique et les incontestables avantages qui en résultent (voir § 2.1.2.3)

2.1.2.2. Formats et aspects sémantiques

Différents éléments contribuent ou sont nécessaires à l'obtention d'une plus grande richesse sémantique des instructions machine :

- La variabilité du format ; elle permet la présence simultanée d'instructions à 0, 1, 2 ou 3 opérandes (peut-être plus) dans le jeu d'instructions (exemple de l'iAPX 432, figure n°5), l'utilisation de descripteurs et d'implicites sémantiques, autorisé des représentations différentes des opérandes (adresses ou valeurs) ;

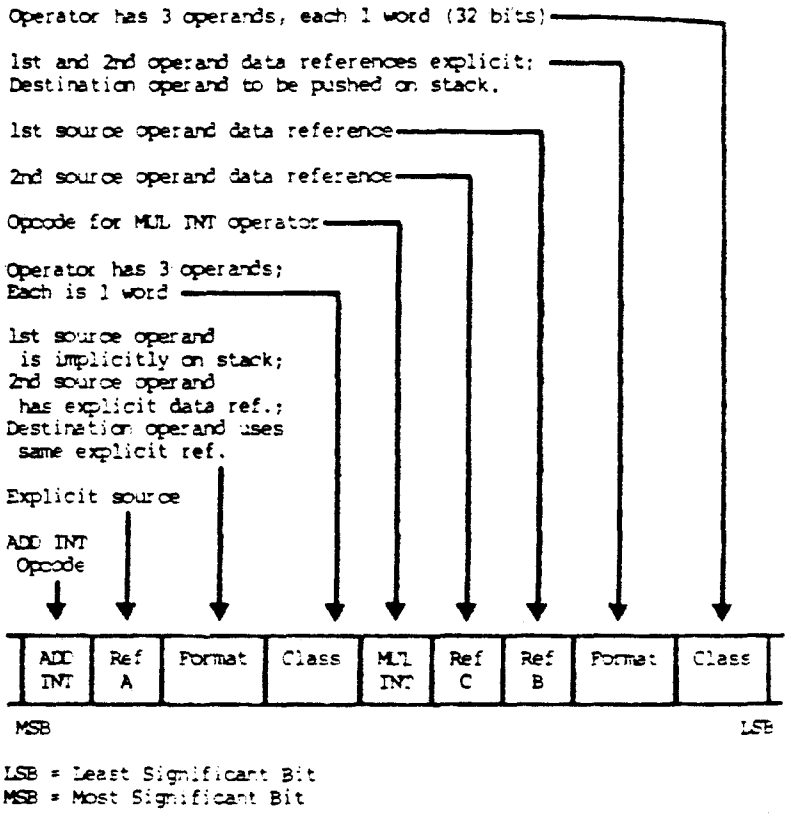


Figure n°5 : Illustration de la variabilité du code dans l'iAPX 432

- L'utilisation de descripteurs (figure n°6) ; ceux-ci facilitent la manipulation et les contrôles des objets structurés (voir § 2.2.2.2) en augmentant la puissance sémantique des instructions auxquelles ils se rattachent ;

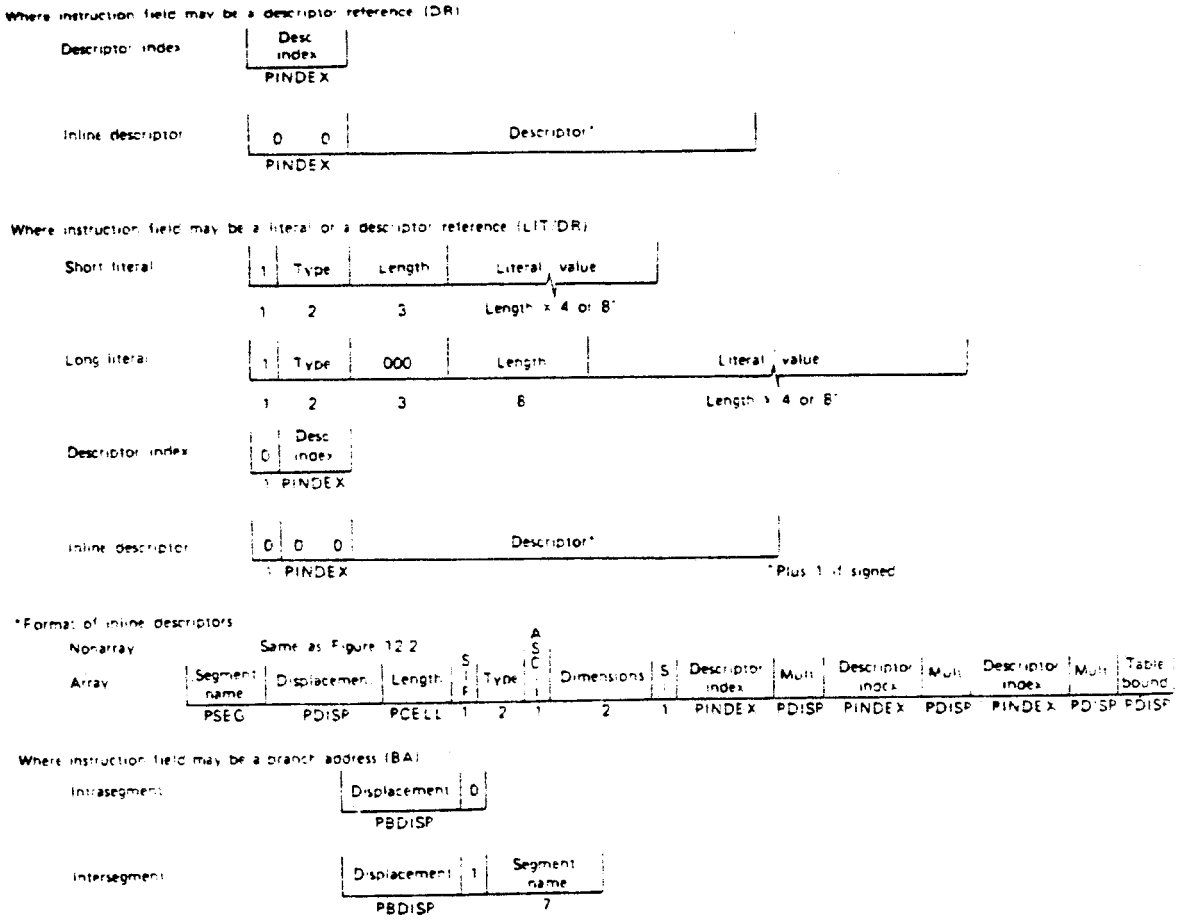


Figure n°6 : Exemples de formats de champs Instruction dans le B 1700

- L'adressage logique ; les opérandes ne sont plus les représentations de ressources physiques mais les équivalents des variables des langages évolués (voir la notion d'objets § 2.1.3.1 et les formes d'adressage § 2.2.2 et 2.2.3) ;

- L'usage de la pondération logique ou physique ; les techniques de pondération sont basées sur les occurrences dynamiques ou statique des informations en vue de l'optimisation du codage des instructions (la représentation du code machine dans le B 1700 [Wil 72b] en est un exemple) ;

Quelle que soit la structure donnée aux instructions (n-uplets, formes polonaises préfixées ou postfixées, linéaires ou arborescentes, orientées pile), celles-ci doivent intégrer certaines fonctions de commande évoluées (choix, répétition) et posséder une réelle puissance sémantique*.

2.1.2.3 Conséquences générales sur le fonctionnement

Le haut niveau sémantique du langage machine a d'abord deux effets sensibles en termes d'efficacité :

1. Une contribution importante à l'abaissement du coût de développement et de mise au point des compilateurs (voir § 1.3) ;

2. La compacité du code-machine qui diminue les accès mémoire et peut contribuer ainsi à l'amélioration des performances (voir § 2.1.2.1)

De plus, le fait de faciliter la création du logiciel de base (compilateurs mais aussi systèmes d'exploitation, utilitaires,...) améliore sensiblement sa fiabilité et contribue à la sûreté de fonctionnement.

* Les mesures de TUCKER et FLYNN puis de BROCA et MERWIN [BMe 73], faites à partir de l'exécution directe de FORTRAN, montrent que l'efficacité de l'interprétation est proportionnelle à la puissance sémantique du langage intermédiaire.

Les contrôles de différentes natures (optionnels ou implicites) dans les instructions, outre la réduction de l'effort demandé à la génération de code, sont un élément supplémentaire de fiabilité.

La sécurité (protection des objets) est également mieux assurée dans la mesure où l'utilisateur n'a pas le contrôle direct des ressources matérielles gérées à un niveau auquel il ne peut accéder.

Dans le cas où l'utilisateur doit "pratiquer" le langage machine (rédaction du compilateur, souci d'optimisation d'un programme, mise au point,...) le caractère plus symbolique des instructions est cause d'amélioration des relations homme/machine et, ceci entraînant cela, facteur de fiabilité (voir plus haut).

2.1.3. Les machines orientées-objet

2.1.3.1. La notion d'objet

Pendant longtemps, la maîtrise imparfaite des problèmes de compilation liés aux contraintes imposées par les architectures classiques a obligé les utilisateurs des langages de programmation de l'ancienne génération comme FORTRAN ou PL/1 à déclarer les variables ou les constantes suivant la taille de leur représentation.

Les langages modernes comme PASCAL ou ADA ne considèrent que des objets typés, définis par l'ensemble des valeurs qu'ils peuvent prendre et par l'ensemble des opérations qui peuvent être effectuées sur eux.

Si la plupart de ces objets sont généralement d'un type prédéfini, simple (entier, réel...) ou structuré (tableau, record,...), certains objets peuvent correspondre à des types abstraits définis par le programmeur quand les primitives du langage le lui permettent (c'est la cas notamment dans ADA).

Dans les machines dites **orientées-objets**, les objets conservent cette définition au niveau de l'interface machine : les **objets** sont alors des entités dont l'implémentation physique n'est connue que de l'architecture matérielle.

Même si la configuration physique de la mémoire, la taille limitée des registres ou la largeur des bus imposeront toujours un découpage de toute information transitant dans la machine, il semble souhaitable que ce conditionnement soit pris en compte par la machine et uniquement par elle (les avantages sont développés au § 2.1.3.2).

Ainsi, dans les machines dites orientées-objets comme l'IBM 38 ou l'iAPX 432, les concepts de segments, pages, fichiers, mots ou octets sont conservés mais leur gestion et entièrement assurée par l'architecture système et donc transparente à l'utilisateur.

Il faut également souligner le fait que le concept d'objet accrédité par ces architectures ne se limite pas aux seules structures de données rencontrées dans les langages évolués (tableaux, intervalles, records,...) mais englobe d'autres entités reconnues jusqu'à présent par les systèmes d'exploitation (processus, ressources, environnements d'exécution,...) comme le montre la figure n°7.

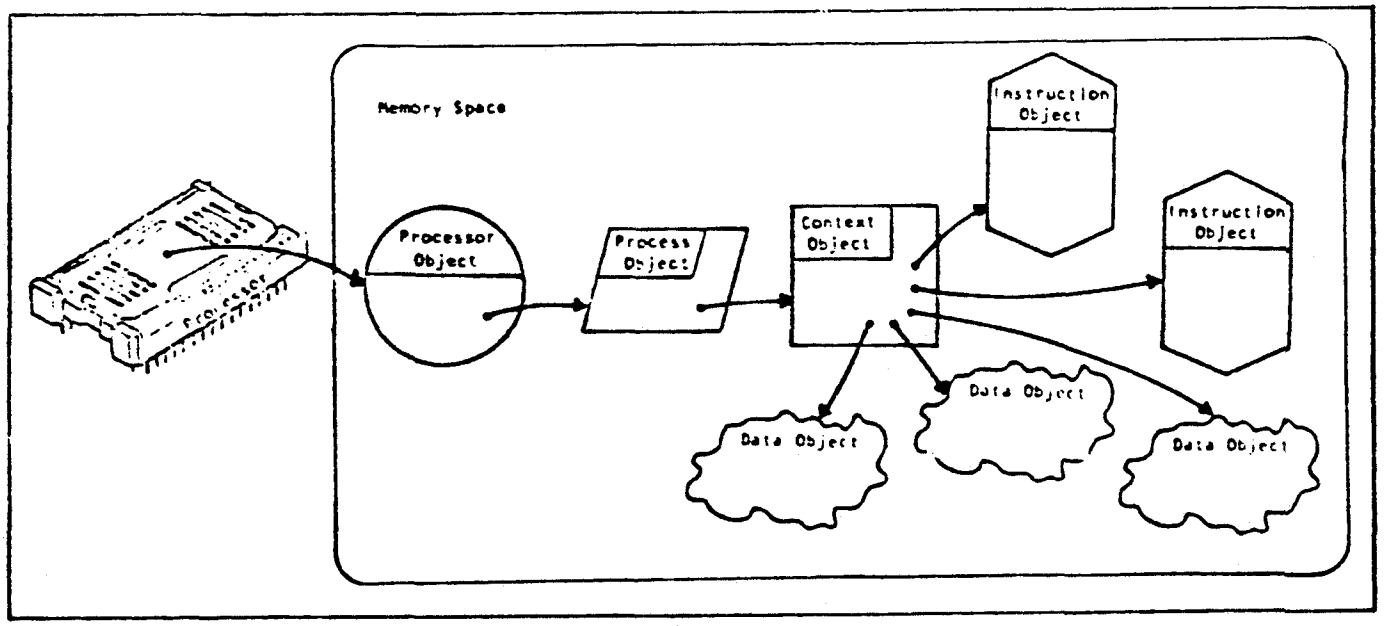


Figure n°7 : L'orientation objet de l'iAPX 432

2.1.3.2. Conséquences générales sur le fonctionnement

La prise en charge du concept d'objet au niveau de l'architecture entraîne la réduction du fossé sémantique au même titre que l'adoption d'un langage intermédiaire de haut niveau sémantique.

Son incidence sur l'efficacité revêt donc à peu près les mêmes aspects en simplifiant la conception et le rôle du compilateur : le travail de conversion des concepts et structures de données évolués est effectué en grande partie par le matériel.

Les implications dans le domaine de sûreté de fonctionnement rejoignent pour une part les avantages d'ordre général que procure la réduction du fossé sémantique mais se révèlent surtout significatives à travers les mécanismes d'accès que nécessite l'orientation objets (voir § 2.2).

En ce qui concerne la capacité d'évolution d'architectures orientées-objet, les conséquences peuvent être divergentes suivant le cas :

- Soit la machine ne manipule que des objets prédéfinis et/ou dépendants d'un certain type d'applications (scientifique, base de données,...) son adaptabilité est alors limitée ;

- Soit la machine offre la possibilité de construire des objets nouveaux (structures de données ou types abstraits) : son adaptabilité est naturellement plus importante.

2.2. LES NOUVELLES MISSIONS DE L'ARCHITECTURE

La motivation fondamentale des machines-langage(s) est claire : c'est prioritairement la réduction du fossé sémantique.

Les langages de programmation de haut niveau ne sont cependant pas les seules sources génératrices de concepts évolués ou de mécanismes élaborés que la machine doive supporter.

Leur prise en compte partielle ou totale par l'architecture constitue les nouvelles missions qui lui sont dorénavant assignées.

2.2.1. L'architecture et les systèmes d'exploitation

Certaines techniques introduites par les logiciels de base (systèmes d'exploitation, compilateurs, utilitaires,...) restent aujourd'hui le plus souvent implémentées classiquement de manière logicielle (mécanisme de protection, d'aide à la mise au point,...) et ralentissent, en particulier l'exécution des programmes.*

Le coût en volume de code machine qui en résulte peut être résorbé par les méthodes de réduction du fossé sémantique qui viennent d'être évoquées (nouveaux schémas d'exécution).

Un autre ensemble de solutions consiste à éviter le passage de l'implémentation en langage évolué, ou même souvent en langage machine, pour intégrer ces concepts directement au matériel (ou à l'architecture logique).

Les architectures classiques ont tenté d'apporter une aide aux systèmes d'exploitation en supportant matériellement quelques-unes de leurs fonctions de base ; elles introduisirent notamment les interruptions et les canaux d'E/S "intelligents" (UNIVAC 1103, IBM 702) pour la gestion des E/S, principalement les registres de base, les mémoires virtuelles (Atlas, B 5000) et les topographes (SDS 940) pour la gestion de la mémoire ainsi

* La vitesse potentielle du CDC Cyber 72 a été évaluée à 56,6 Mbits/s sans intervention du système d'exploitation et à 15,74 Mbits/s lorsque celui-ci opère [Kav 80].

que les bancs de registres, les piles, les registres de base et de borne implicites, les processeurs d'E/S entièrement programmables, les registres d'état de l'instruction et les protections (physiques) de la mémoire pour une meilleure prise en compte des données et des instructions [Fri 75].

Ces efforts , bien qu'importants et toujours utilisables, restent cependant partiels ; certaines machines comme SYMBOL ou MU5 (déjà évoquées) furent au contraire conçues pour être le support privilégié et parfaitement adapté aux systèmes d'exploitation qui leur étaient destinés.

Cette approche a maintenant dépassé le stade expérimental ainsi que le montre l'exemple d'INTEL qui vient de sortir le **80130 Operating System Kernel Chip** ; celui-ci supporte 35 primitives de base de système d'exploitation regroupant des opérations de gestion des travaux, des tâches, des segments mémoire, des interruptions et de communications inter-processeurs.

Une implantation matérielle de tout ou partie du système d'exploitation entraîne souvent sa rigidité et donc son inaptitude à se reconfigurer en fonction du type d'application qu'il pilote.*

Sa flexibilité est possible grâce à la micro-programmation dynamique et aux techniques de **migration verticale** qu'elle autorise.

La migration verticale est définie par STOCKENBERG et VanDAM [SVa 78] comme la combinaison de deux techniques :

- D'une part, la "migration" de certaines primitives qui, de logicielles, deviennent micro-programmées (passage dans une "couche" inférieure, suivant la hiérarchie "Software/Firmware/Hardware") ;
- D'autre part, la (re) configuration du système d'exploitation pour une (ou un ensemble de) tâche(s) donnée(s).

* L'immuabilité des fonctions système qu'implique une implémentation câblée ou microprogrammée en ROM peut néanmoins offrir des avantages en performances qui se justifient dans les machines spécialisées ou les systèmes embarqués.

Le remplacement de certaines procédures logicielles par leur équivalent microcodé n'est certes pas totalement nouveau comme le prouvent les exemples du VMAssist [Tal 75] ou du ECPS (Extended Control Program Support) [Kop 76] appliqués à l'IBM S/370.

Ces techniques restent cependant manuelles et à la charge complète du programmeur et du microprogrammeur.

Des systèmes d'évaluation et des outils de remplacement automatiques ont donc vu le jour depuis quelques années dans le but de fournir une véritable et importante amélioration des performances de programmes fréquemment utilisés et, en particulier, des systèmes d'exploitation.

Parmi toutes ces réalisations, citons en deux qui prouvent l'intérêt actuel porté à UNIX : le moniteur "firmware" développé à l'Université de Dortmund [HKa 82] et le modèle Maître/Esclave pour la mémoire de commande modifiable du PERKIN ELMER 3220 [RWi 81].

Les techniques de migration verticale ne constituent qu'un aspect de la prise en charge par le matériel ou l'architecture de tâches ou de mécanismes attribués classiquement au logiciel de base.

Certains de ces travaux (accès aux objets, protection,...) peuvent être en effet beaucoup mieux intégrés à l'architecture qu'ils ne le sont dans la plupart des machines classiques.

C'est plusieurs de ces aspects se réclamant principalement des systèmes d'exploitation qui sont maintenant développés ; certaines notions exposées ne sont pas nouvelles, voire novatrices : c'est en réalité la reconsidération de leur implémentation qui laisse espérer de nouvelles perspectives en matière de performances et de fiabilité.

2.2.2. Les objets auto-définis

Il existe deux formes d'implémentation du concept **d'auto-définition : les préfixes et les descripteurs.**

2.2.2.1. Les machines à préfixes

L'impossibilité de reconnaître le type d'une information à travers sa configuration binaire a poussé certains concepteurs à adjoindre un complément d'information à chaque entité manipulée par le matériel, sous la forme d'une suite de bits de taille fixe associés à chaque emplacement mémoire * (voir exemples de SWARD, figure n°8).

Burroughs utilisait déjà cette technique dans sa gamme B 6500/6700 : les premiers bits de chaque emplacement forment un préfixe (ou "tag") qui peut indiquer, entre autres une valeur, un nom ou un descripteur de segment [Cre 71].

Le préfixe peut avoir d'autres attributs que l'identification de type comme aide à la protection, au ramasse-miettes ou à la mise au point des programmes.

Malgré des avantages certains déjà mis en évidence par FEUSTEL [Feu 73] (et repris au § 2.2.2.3), les mémoires à préfixes ont le grand inconvénient de conduire (à priori) à un surcroît d'encombrements mémoire.

Des réalisations comme SWARD minimisent pourtant ces effets en faisant appel à la plupart des autres concepts de machines-langage(s) présentés dans cette étude.

* Pour obtenir un système efficace, il est nécessaire de posséder une mémoire centrale physiquement préfixée ; des problèmes de conditionnement peuvent se poser lors des transferts avec la mémoire secondaire (disques ou bandes) qui n'intègrent pas le concept de préfixe (cas des objets "pointer" dans l'IBM 38).

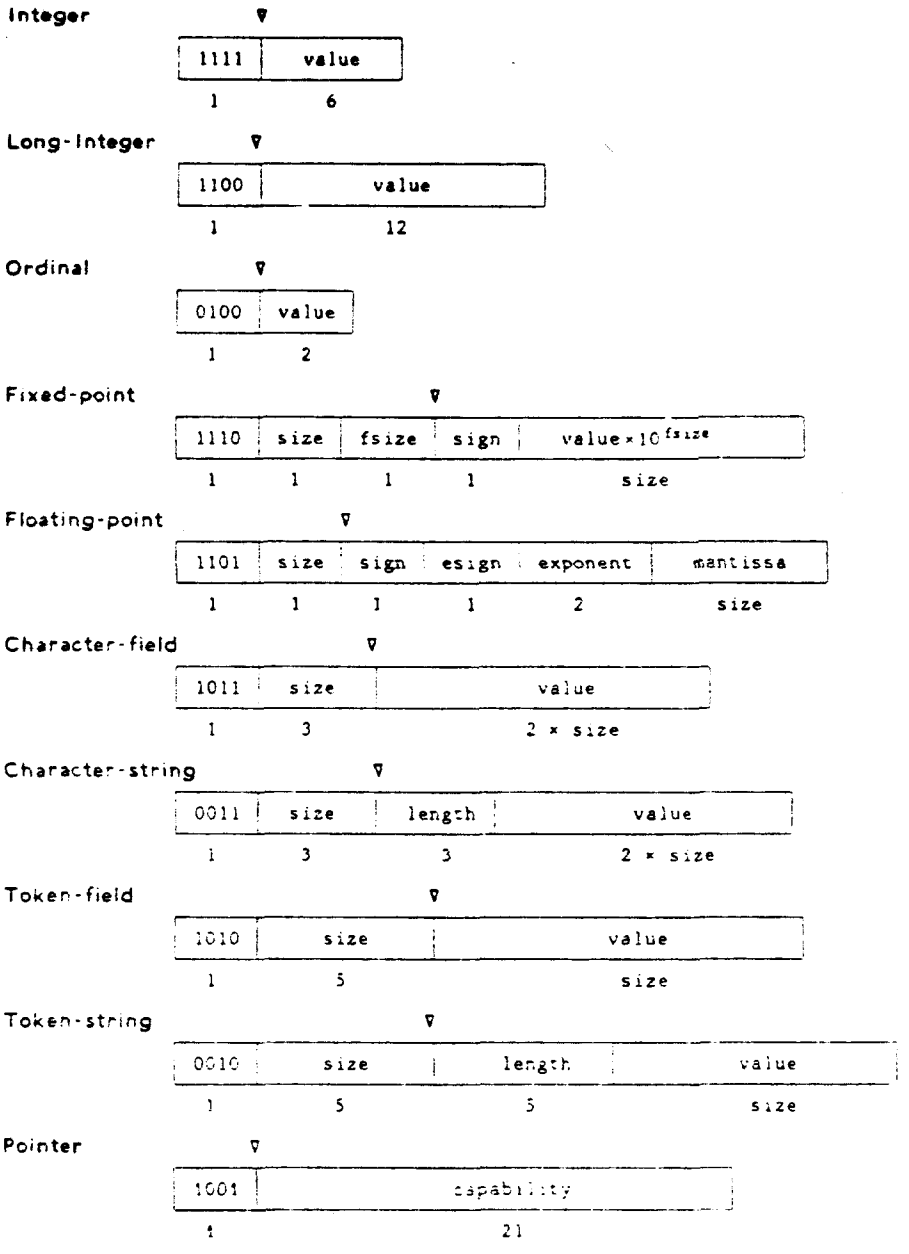
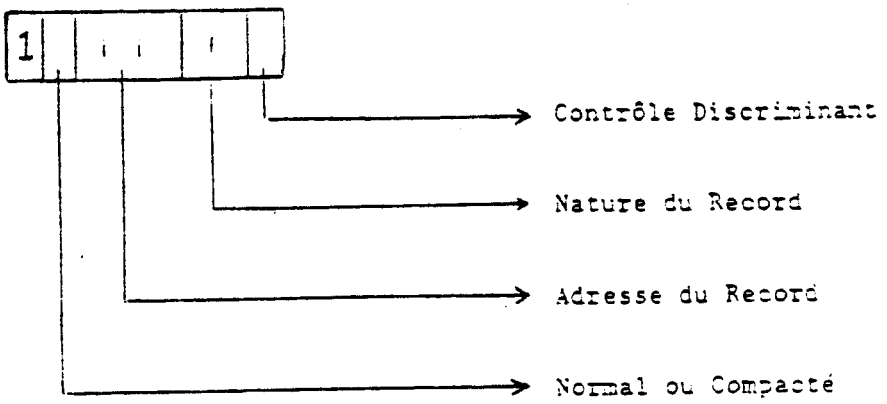


Figure n°8 : Types de cellules primitives dans SWARD

2.2.2.2. L'emploi des descripteurs

La notion de descripteur abordée dans ces lignes n'englobe pas celle, familière aux habitués des systèmes d'exploitation, de descripteur de segment : les principes d'auto-définition s'appliquent surtout aux données simples ou structurées.

Les descripteurs sont d'ailleurs surtout associés aux formes structurées de données comme les tableaux multi-dimensionnés, les tables ou encore les records (la figure n°9 propose un exemple de descripteur de composant de record dans MLI).



Normal ou Compacté : indique si les informations sur la taille et le déplacement du composant de record (voir D02) sont exprimés en octet ou en bit.

Adresse du Record : indique les possibilités suivantes pour l'adresse :

- adresse bit dans la pile
- adresse octet dans la pile
- adresse locale exprimé sur 1 à 3 octets suivant D01
- adresse globale exprimé sur 2 à 4 octets suivant D01

Nature du Record :

- record à accès direct
- record à accès dans le tas avec contrôle sur adresse
- record à accès dans le tas sans contrôle sur adresse

Contrôle Discriminant : indique si un contrôle est demandé ou non. Si oui, l'opérande est complété de Descriptifs de Contrôle de Discriminant (DCD).

Figure n°9 : Descripteur d'un composant de record dans MLI

Un descripteur est avant tout un ensemble structuré d'informations relatives à un objet ou à une collection d'objets :

- Type de l'objet (tableau, record, procédure,...) ;
- Description de la structure de l'objet (adresse du Oème élément et dimensions pour un tableau, bornes inférieures et supérieures pour un intervalle,...).

Le descripteur peut aussi, suivant l'implémentation, définir l'accès à l'objet (nom ou adresse de l'objet, objet lui-même,...).

Le MU5 [Ica 78] possède des descripteurs de tableaux à trois champs :

- Le premier champ (8 bits) indique la taille de chaque élément de tableau ;
- Le deuxième champ (24 bits) indique la taille du tableau ;
- Le troisième champ (32 bits) contient l'adresse du début de tableau.

Les descripteurs, contrairement aux préfixes, demandent toujours un accès mémoire supplémentaire (qu'ils indiquent l'adresse ou pas) ; ils permettent par contre d'éviter certains contrôles inutiles (repérés par le compilateur) qui sont presque toujours systématiques sur les machines à préfixes ; ils peuvent également être partagés et plus facilement mis à jour.

2.2.2.3. Conséquences générales sur le fonctionnement

Le principe de l'auto-définition des objets peut être très important sur le plan des performances globales de la machine. La généricité des instructions qu'il engendre, l'automatisation (parfois optionnelle) des contrôles et de conversion de type qu'il permet ont une incidence directe sur la compacité du code machine.

L'apport des préfixes à la réalisation de mécanismes de ramasse-miettes peut être également un facteur décisif d'efficacité dans certaines applications.

Dans la mesure où des contrôles de types permettent de détecter des erreurs de manipulation des données, l'auto-définition des objets est un moyen relativement simple de contribuer à la sûreté de fonctionnement.

Il faut également remarquer que les préfixes ou les descripteurs peuvent jouer un rôle intéressant dans la mise au point des programmes (suivi de variables, détection de non-initialisation,...).

2.2.3. Les machines à capacités

Le concept d'**adressage par capacités** n'est pas nouveau puisqu'il existe depuis 1966 [DVa 66] ; bien que largement discuté dans le cadre des systèmes d'exploitation, il n'a pas encore eu, jusqu'à présent, une grande influence sur l'évolution des architectures matérielles.

2.2.3.1. Evolution de la protection

Jusqu'à un passé récent, les problèmes de protection concernaient l'interdiction pour un utilisateur malveillant ou maladroit d'intervenir dans les fonctions réservées au système d'exploitation, voire dans l'environ-d'un autre utilisateur.

Le fonctionnement était basé dans le premier cas sur un mode **Maître-Esclave** tandis que le principe de l'**isolement** (chaque utilisateur possède une machine virtuelle autonome) était principalement utilisé dans le second.

La nécessité de plus en plus impérative de pouvoir partager les ressources et les informations a conduit à imaginer d'autres formes de protection à l'intérieur d'un système informatique.

Après le concept de **segmentation** (très lié à celui de mémoire virtuelle) sont donc apparus successivement dans ce but ceux d'**anneaux de protection**, **d'adressage par capacités** et de **domaines de protection.**

Avant de décrire brièvement ces mécanismes, il semble opportun de préciser les fonctions que doivent satisfaire "tout système de protection général" [CRO 75].

- "Assurer l'indépendance des objets qui doivent rester logiquement indépendants. (...)
- Permettre une protection de l'utilisation de l'information, en fonction de l'opération qui est tentée.
- Permettre une protection sélective de l'information partagée, en fonction de l'usager ou d'un groupe d'usagers".

Les deux premiers points maintiennent ce qui est également appelé l'**intégrité des objets**, alors que le dernier assure la **sélectivité** des accès en fonction des droits de l'utilisateur.

Nous ne reviendrons pas sur les notions de segmentation de l'espace adressable, de pouvoir d'un processus, de méfiance mutuelle, de révocation des droits acquis ou de problèmes d'étanchéité qui sont désormais familiers aux concepteurs de système (voir [Car 79]).

Les deux concepts développés dans cette étude sont intégrés dans un certain nombre d'architectures nouvelles se démarquant des machines classiques : l'adressage par capacités et les (petits) domaines de protection.

2.2.3.2. L'adressage par capacités

Le principe de la capacité est maintenant connu : il s'agit d'un ensemble d'informations (traditionnellement : < type de l'objet, droit d'accès, adresse logique de l'objet> ou <droits d'accès, nom de l'objet, repérage d'entité dans l'objet >) qui permettent l'accès à un objet en offrant toute la protection nécessaire.

L'objet possède alors un nom unique* qui représente une adresse logique interprétable uniquement par la machine pour la détermination de l'adresse physique.

L'adressage se fait classiquement par l'intermédiaire d'une Table Unique des Objets du Système (TUOS) comme l'indique la figure n° 10.

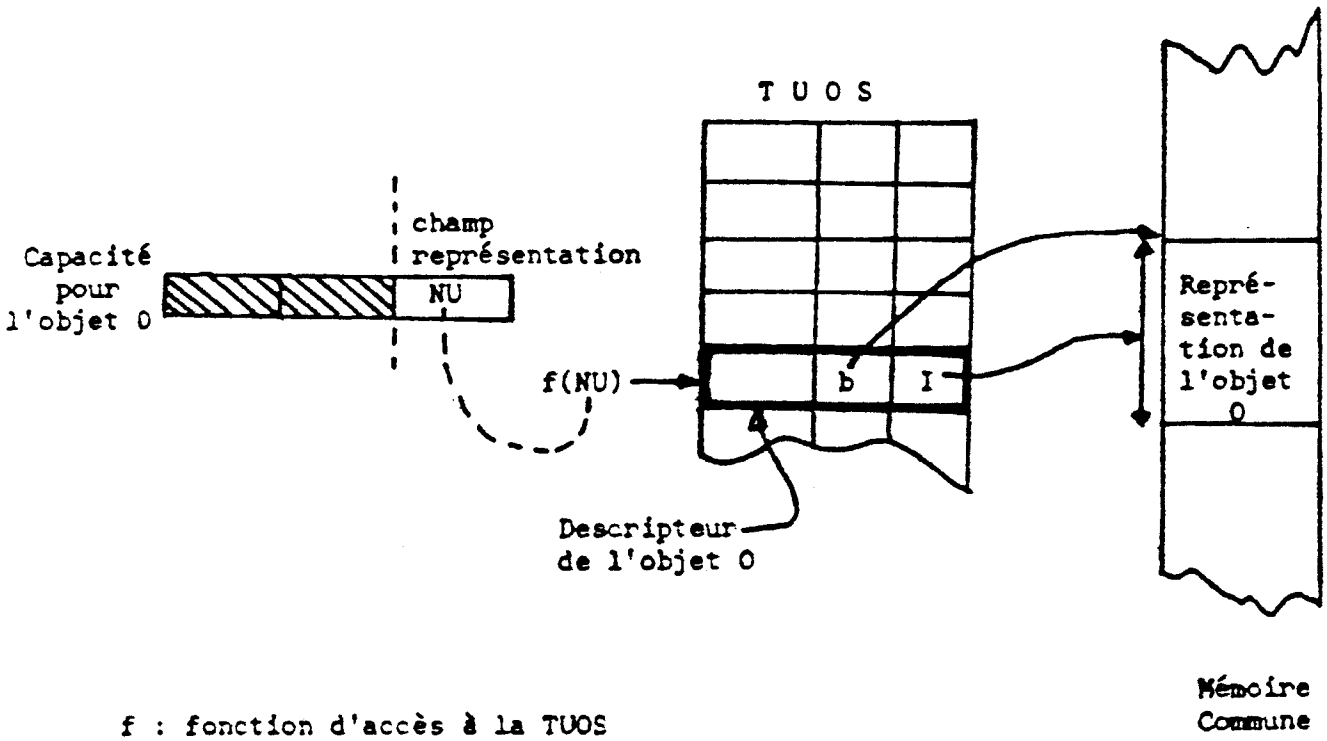


Figure n°10 : Schéma classique de l'adressage par capacités

L'adressage par capacités a surtout été implémenté en logiciel sur des systèmes expérimentaux comme CAL [LSt 76] et Hydra [CJe 75].

Il a fallu attendre le milieu des années 70 pour le voir présent dans les architectures (PLESSEY 250 [Eng 72, Eng 74, Cos 74, Fla 76], IBM 38, iAPX 432).

La présence de capacités introduit immédiatement le problème de leur propre protection (accès et création).

* Les noms peuvent être soit **temporels**, ils reflètent alors le temps de leur création, soit **spatiaux**, ils correspondent à la position du descripteur dans la table TUOS.

Deux types d'implémentation existent :

- Les capacités sont regroupées en C-listes référencées par des capacités système non modifiables (c'est l'approche PLESSEY 250, iAPX 432, CAP [NWa 77, Her 78], HYDRA, ou [Ren 76] et [Lan 78]) ;

- Les capacités sont "éparpillées" et leur identification est assurée par un préfixe (Burroughs 6700, IBM 38, le Rice Computer [Fen 72] l' Iliffe's Basic Machine [Ili 68]).

2.2.3.3. Conséquences générales sur le fonctionnement

Les avantages de l'adressage par capacités sont essentiellement liés à la sûreté de fonctionnement :

- Une protection complète (tous les accès aux objets sont contrôlés), souple (suppression du mode Maître-Esclave), modulaire (un accès interdit à une capacité ne met pas en danger tout le système) ;

- Une aide éventuelle à la mise au point des programmes en interdisant la référence à un objet dont la durée de vie est écoulée ("dangling reference problem ").

L'emploi des capacités, combiné à une orientation objets, favorise l'uniformisation et donc la simplicité et la cohérence des concepts pris en compte par l'architecture.

A priori, l'adressage par capacités semble induire une indirection systématique donc "pénalisante" de l'accès aux objets : l'appréciation de ses conséquences sur les performances doit cependant être défini globalement en tenant compte, d'une part, de son action réductrice sur la longueur des programmes (compaction du code machine due à la représentation des opérandes sous forme d'index d'une C-liste, par exemple) et d'autre part, de ce qu'il évite le coût additionnel en logiciel (à la compilation et à l'exécution) que nécessiterait la prise en charge d'un niveau de protection équivalent.

2.2.4. Les domaines de protection

Le concept de **domaine de protection** ne représente pas un mécanisme matériel ou logiciel précis ; au contraire, sa réalisation dépend des systèmes qui l'intègrent dans leur définition.

2.2.4.1. Caractéristiques

La matérialisation du pouvoir d'un processus sous la forme de domaines de protection a été proposée par LAMPSON [Lam 71].

D'après SPIER [Spi 73], un domaine de protection peut être défini comme étant "un espace indépendant d'adresses locales définissant l'ensemble complet des adresses pouvant être formulées par un ensemble d'instructions".

En fait, cette approche consiste à partitionner les objets accessibles à un processus de manière à ce que seuls soient disponibles les objets nécessaires à l'opération courante ; une technique peut être, par exemple, de regrouper dans des segments de données particuliers les capacités des objets accessibles au même moment (solution adoptée pour MLI, voir chapitre 4).

Le traitement du pouvoir d'un processus est, dans ce cas, assez dissemblable de celui engendré dans les systèmes à "anneaux de protection" ; dans ce dernier type d'organisation, le pouvoir d'un processus correspond à une valeur dans un ensemble ordonné symbolisant la hiérarchie dans les droits d'accès pour chaque processus de système.

Le pouvoir d'un processus est donc défini à partir de l'ensemble des listes de contrôle d'accès, de l'identité du processus et du niveau de protection ; le changement de pouvoir s'effectue par le changement d'anneau.

Le défaut de ce type d'approche est la rigidité provenant :

- Du faible nombre de niveaux de protection habituellement implanté (8 pour MULTICS, 4 pour le Système 64 de CII-HB, et l'INTEL iAPX 286) ;
- De la valeur hiérarchique stricte qui les unit ;
- De son inadéquation à la résolution des problèmes généraux classiques de protection (méfiance mutuelle,...).

Les domaines de protection apparaissent être une approche plus souple et plus modulaire pour le développement d'applications complexes en sous-systèmes.

2.2.4.2. Conséquences générales sur le fonctionnement

Les avantages des domaines de protection se situent naturellement presqu'exclusivement dans la catégorie "sûreté de fonctionnement" ; ils se résument principalement à trois aspects qui sont :

- La sécurité des programmes, qu'ils soient systèmes ou utilisateurs, est assurée et doit permettre de mettre en place le "principe du moindre privilège" ;
- La mise au point de programmes est facilitée par le découpage en domaines qui permet le repérage rapide d'une erreur et la limitation de sa propagation ;
- La modularité effective de l'ensemble des processus empêche la construction anarchique des programmes avec des modules d'interface souvent illisibles et peu apparents.

L'influence de l'implémentation des domaines de protection sur les performances de la machine varie suivant les choix d'architecture et reste de toute manière difficile à évaluer ; le problème est le même que dans le cas de l'adressage par capacités (voir § 2.2.3.3.).

2.2.5. Quelques autres aspects importants

D'autres mécanismes devenus indispensables au fonctionnement des ordinateurs sont implantés logiciellement dans la quasi-totalité des architectures classiques.

Leur prise en compte au niveau de l'architecture ou du matériel peut apporter une contribution significative à la réalisation des objectifs définis au § 1.4.

2.2.5.1. La gestion des appels et retours de sous-programmes

Les opérations nécessaires au traitement des **appels et retours de sous-programmes** (procédures, fonctions,...) sont devenus une composante importante à l'exécution des programmes ; l'avènement de langages comme ADA et de méthodologies de programmation fortement modulaires ne fait qu'accentuer cette tendance.

La sauvegarde du contexte de la procédure appelante, le passage de paramètres, le contrôle de visibilité des variables globales ou locales, les initialisations éventuelles ainsi que tous les autres mécanismes de protection et de branchement sont régulièrement implantés en logiciel dans les architectures classiques.

Les techniques de migration verticale appliquées aux instructions d'appel de procédures ont mis en évidence les gains substantiels en performances qui peuvent en être attendus. Et même si, comme l'affirme WILKES [Wil 82], il n'existe pas de véritable consensus sur la manière de les implémenter, l'ensemble des concepteurs de systèmes informatiques s'accordent pour estimer que les appels de procédures nécessitent une attention et un support tout à fait particuliers.*

2.2.5.2. La gestion des processus

Un **processus** est, suivant la terminologie habituelle, une suite temporelle d'exécutions d'instructions.

L'évolution de l'exploitation d'un ordinateur a conduit les systèmes à prendre en charge d'abord la **multiprogrammation** puis le **multi-traitement**.

La gestion des processus qui est classiquement prise en compte logiciellement par les systèmes d'exploitation recouvre les aspects suivants :

* Des mesures effectuées par KNUTH et confirmées par WULF indiquent que l'endroit qui demande le plus de temps dans la majorité des programmes se trouve dans le mécanisme d'enchaînement des sous-programmes [Wul 75].

- L'attribution des processus au(x) processeur(s) ; lorsque le système est mono-processeur, la simultanéité d'exécution est simulée par le partage temporel ("time-slicing") de l'unité centrale ; dans un contexte multi-processeurs, l'attribution se fait en fonction de l'architecture matérielle (hiérarchisée ou distribuée) et des stratégies ("policies") d'implémentation (notions de files d'attente, de priorité, de préemption,...).

- La communication-synchronisation entre processus ; longtemps traitées séparément, ces deux aspects de l'interaction entre processus sont de plus en plus souvent réunis à travers les concepts de langage ("rendez-vous" dans ADA, par exemple) ; qu'ils soient communs ou non, les mécanismes implémentant le type de synchronisation et le mode de communication restent voisins dans leur forme ou tout au moins mutuellement dépendants (activation d'un processus, interruption,...)

- La création et la destruction de processus ; la création d'un processus, qui est l'attribution d'un nom et la définition d'un vecteur d'état initial (programme, données d'entrée, pouvoir), ainsi que la destruction qui peut intervenir normalement à la fin de l'exécution du processus mais aussi à l'initiative du système (ou d'un autre processus) s'il détecte une erreur, exigent un ensemble d'opérations habituellement commandées par un programme système ("superviseur" ou "moniteur").

Deux arguments importants militent en faveur d'une plus grande part de l'intégration matérielle dans le schéma d'implantation des mécanismes de gestion des processus :

1. Des raisons d'efficacité et de sûreté déjà évoquées concernant les avantages de l'implémentation matérielle sur l'implémentation logicielle ;

2. L'expression de plus en plus souvent explicitée de ces mécanismes (en particulier ceux de la communication et de la synchronisation) dans la spécification des langages évolués actuels (ADA [USA 83], CONCURRENT-PASCAL [Bri 75], CHILL [BLW 79], LTR-V3 [CIM 78],..).

La prise en compte de ces deux points contribue ainsi, sous des formes différentes, à la réduction globale du fossé sémantique.

2.2.5.3. L'implémentation d'un ramasse-miettes

Un **ramasse-miettes** ("garbage collector") est l'ensemble des techniques qui permettent à un système informatique de posséder une mémoire (généralement virtuelle) ne contenant, à un instant donné ou périodiquement, que des informations encore accessibles au(x) processeur(s) : chaque emplacement cessant d'être référencé devient libre pour une nouvelle affectation.

Les mécanismes et algorithmes d'implémentation ont fait l'objet de nombreuses études comme le montre la présentation due à COHEN [Coh 81].

Les principales familles de ramasse-miettes peuvent être classées selon les types de méthodes utilisés : périodique avec tassements ([HWa 67, FNo 78],...), à compteurs de référence ([DBo 76], [Fri 76],...), continue avec pile ([Ste 75], [Mul 76],...), continue sans pile ([Dij 78],...), par demi-espaces ([Min 63], [Bak 78],...) ou par régions ([LHe 80]).

La présence d'un ramasse-miettes dans les systèmes informatiques se trouve justifiée dans les cas, de plus en plus fréquents, de gestion d'une mémoire dynamique nécessaire au traitement des listes (l'exemple le plus significatif est certainement le support de LISP*, et des informations arborescentes ou graphiques (au sens de PAIR et GAUDEL [PGa 79]).

Les objets de type pointer sont reconnus par la majorité des langages évolués (**ref** en ALGOL 68, **pointer** en PL/1, ↑ en PASCAL, **access** en ADA,...) et réalisent souvent (ALGOL 68, ADA,...) le support du type abstrait "tas" ("heap").

* L'étude d'un ramasse-miettes adapté à ce problème a été réalisée par LECOUFFE dans le cadre de sa machine LISP [Lec 77].

Cette notion de tas est reconnue comme nécessaire aux langages modernes par une majorité de concepteurs de langages, afin de répondre aux besoins actuels des utilisateurs (bases de données, structures graphiques,...).

Le concept de mémoire dynamique virtuellement infinie pose naturellement des problèmes lors de son implémentation sur le matériel, physiquement limité par définition.

Les mécanismes de Ramasse-Miettes deviennent alors indispensables (et même fondamentaux dans les systèmes embarqués) ; leur complexité, lourdement ressentie actuellement par les compilateurs ou les systèmes d'exploitation, ainsi que les opérations proches du matériel (allocations d'espaces mémoire) qu'ils nécessitent sont les raisons qui permettent de penser qu'une plus grande intégration de ces mécanismes au niveau architectural ou matériel conduira à un accroissement de leur efficacité et de leur fiabilité.

2.2.5.4. Le niveau unique de mémoire

Le **niveau unique de mémoire** ("single-level storage") signifie, pour un système informatique, la correspondance exacte entre la mémoire virtuelle et ce qui est communément appelée la mémoire d'archives.

Les mécanismes d'adressage et les fonctions d'accès aux objets apparaissent donc identiques au niveau du langage machine (unification du mode de mémorisation) ; de plus, les objets peuvent rester référencés (en conservant le même nom) après leur création et leur utilisation, sans avoir recours aux habituels fichiers temporaires.

Les avantages d'un niveau (logique) unique de mémoire sont principalement la réduction en coût de logiciel (élimination dans une large mesure de la complexité des E/S, telles qu'elles apparaissent à l'utilisateur (le volume d'E/S reste en effet globalement équivalent) et simplification des opérations effectuées par le système), l'indépendance technologique qu'il engendre (l'implémentation matérielle prend seule en charge le hiérarchisation de la mémoire et les correspondances qu'elle implique) et (peut-être) surtout son adéquation à l'orientation-objet évoquée au § 2.1.3.

La réalisation matérielle d'un niveau unique de mémoire reste cependant complexe et coûteuse : dans l'IBM 38, les adresses virtuelles des objets occupent 64 bits et le matériel en supporte directement 48.

Le niveau unique de mémoire pose également le problème de la portabilité des objets : même sur disque ou disquette, l'objet possède un nom unique qui peut correspondre à un autre objet dans le système similaire dans lequel il serait transféré.

2.2.5.5. Les outils d'aide à la mise au point

La mise au point d'un programme consiste à vérifier que le comportement de ce programme à l'exécution correspond exactement aux spécifications de l'application pour laquelle il est conçu.

La réalisation de cet objectif s'effectue (ou tente de s'effectuer !) actuellement à travers les différents stades qui constituent la vie de la majorité des programmes :

1. L'analyse du problème, la conception du ou des algorithmes, la traduction dans un langage de programmation ; les techniques sont ici essentiellement d'origine méthodologique (programmation structurée, "quality programming methods" [Chi 80],...) qui favorisent les langages actuels (modularité, encapsulage, typage de données,...) ; l'élaboration de preuves de programmes ([Flo 67,71], [Abr 82]...) reste encore complexe, incomplète et impraticable en milieu industriel.

2. La compilation ou l'interprétation du programme ; l'interprétation de certains langages (BASIC, APL, LISP,...) permet un mode interactif de mise au point qui facilite le travail du programmeur ; outre les contrôles syntaxiques, les compilateurs de langages comme ADA réalisent maintenant certains contrôles sémantiques dus pour la plupart au caractère fortement typé de ces nouveaux langages.

3. La vérification de cohérence et l'exécution symbolique ; ces deux méthodes de vérification sont encore au stade expérimental mais font naître de réels espoirs ; le vérificateur de cohérence ([Ren 80]), comme le système de compréhension de programmes ([Luk 80]) ou les méthodes d'exécution symbolique ([Kin 78], [BEL 75],...) sont des mécanismes d'investigation du comportement du programme visant à détecter certains types d'erreurs.

4. Les tests et les "benchmarks" ; c'est souvent la phase de mise au point la plus utilisée actuellement par les programmeurs ; deux étapes sont nécessaires : la production d'une série de tests adaptés au programme et la modification du programme source en fonction des résultats de son exécution (dans les conditions engendrées par les tests) sur la machine cible ; des outils spécifiques à la mise au point ont vu le jour : constitution d'un fichier historique comme dans EXDAMS ([Bal 69]), points de rupture et commandes associés (mise au point symbolique dans AIDS, [Har 79]), etc.

5. La maintenance* ; théoriquement inutile, la mise au point continuelle (elle accompagne le programme pendant toute la durée de sa vie) est due à l'imperfection des outils de vérification qui l'ont précédée ; il n'existe pas de méthodes actuellement adaptées à cette étape de mise au point sinon celles utilisées lors des tests, accompagnées d'une documentation détaillée et mise à jour continuellement.

La mise au point d'un programme, si elle doit être effectuée le plus rapidement (et le mieux) possible au niveau "macroscopique" (évaluation en hommes/heure ou hommes/semaine), ne nécessite pas de performances particulières au niveau temps-machine (le programme ne s'exécute pas dans un environnement "normal").

L'intégration d'outils de mise au point au niveau matériel ou architectural possède d'autres motivations :

* Il ne s'agit ici évidemment pas des mises à jour provoquées par des modifications des spécifications du programme mais des erreurs (les fameux "bugs") constatées dans son fonctionnement.

- L'existence de tels outils ; l'implantation de mécanismes de mise au point à l'intérieur de la machine est la garantie pour l'utilisateur de disposer d'outils encore trop rares (et trop coûteux) en logiciel ;

- L'adéquation des outils ; l'exécution d'un programme dépendant de la machine, il est normal que les outils de mise au point soient bien adaptés à la configuration matérielle : ils concourrent ainsi à l'efficacité de traduction des traitements symboliques ;

- La fiabilité des outils ; la sûreté des moyens de vérification est évidemment essentielle dans la recherche des erreurs d'un programme : pour cela, les éléments de contrôle et d'observation, introduits dans l'environnement d'exécution d'un programme doivent échapper aux risques de défaillance propres au logiciel ;

- Le coût des outils ; l'économie issue de l'implantation matérielle d'outils d'aide à la mise au point revêt deux aspects : d'une part, le coût matériel additionnel apparaît négligeable devant le prix actuel (et futur) des produits logiciels ; d'autre part, l'amélioration de la communication homme/machine alliée à l'efficacité ainsi apportées doivent augmenter sensiblement la productivité des programmeurs.

CHAPITRE 3

ROLE DE LA MICROPROGRAMMATION

3. ROLE DE LA MICROPROGRAMMATION

La microprogrammation, concept introduit par WILKES en 1951 [Wil 51], est devenue depuis plusieurs années une composante importante sinon privilégiée dans le domaine de l'architecture des machines, ainsi qu'en témoignent les nombreux congrès (et en particulier le "Microprogramming Workshop" annuel qui en est à sa seizième édition) qui se déroulent à travers le monde.

L'engouement incontestable pour la microprogrammation recouvre de nombreux domaines d'application, ce qui prouve sa diversité, sa facilité d'utilisation mais aussi son efficacité.

La microprogrammation n'est certes pas le remède à tous les maux qui accablent le concepteur d'architecture mais ce 3ème chapitre s'attache à démontrer les réelles possibilités qu'elle lui offre, notamment pour le "non-technologue".

A cet effet, après le rappel de quelques définitions de base, sont développées les caractéristiques générales de la microprogrammation (avantages, limites, évolution actuelle) ; la contribution de la microprogrammation dans la réalisation des nouveaux cahiers des charges, tel que ceux-ci sont définis au §1.4, constitue le deuxième thème important de ce 3ème chapitre.

Quelques conclusions sont ensuite présentées pour dégager les principales difficultés qui se posent dans la conception d'un projet de machine en milieu industriel.

3.1. DEFINITIONS DE BASE

L'ensemble des définitions de base, ainsi que l'historique, quelques considérations générales et quatre exemples de machines microprogrammées peuvent être retrouvées dans [Bry 82].

Dans ce paragraphe, ne sont présentées que les notions principales qui dépendent de l'utilisation de la microprogrammation.

3.1.1. Définition de la microprogrammation

Le modèle que WILKES introduisit en 1951, bien que rudimentaire, permet de préciser certaines notions toujours en vigueur actuellement.

Son modèle (figure n°11) était basé sur les constatations suivantes :

- Un calculateur traditionnel peut être décrit comme un ensemble élaboré de réseaux logiques appelés chemins de données ;
- Chaque chemin de données transite par des unités logiques élémentaires accomplissant chacune une **micro-opération** sépcifique sur le flux de données ;
- Les interconnections entre unités logiques élémentaires se font à l'aide de bus de données et de portes de commande (control gate) ;
- Tous ces chemins de données sont statiques par définition et ne peuvent être activés que par des signaux provenant du bloc de commande ;
- Les signaux de commande sont complétés par les impulsions d'horloge pour la synchronisation, le décodage, le séquencement et la logique de décision, et dirigent ainsi entièrement le travail du calculateur ;
- L'unité de commande (control unit) d'un ordinateur "classique" recherche et interprète les instructions-machine qui sont effectivement exécutées par un séquenceur câblé qui a pour tâche d'engendrer les signaux de commande décrits plus haut.

L'utilisation d'un séquenceur câblé fige donc complètement le jeu d'instructions (à moins de modifications du câblage) de la machine et augmente la complexité de sa conception proportionnellement au nombre de portes de commande nécessaires.

L'idée de WILKES a été de faciliter la modélisation du séquenceur en considérant l'ensemble des portes de commandes non pas comme un seul grand circuit logique combinatoire mais comme un ensemble de sites indépendants réalisant chacun une **micro-commande.**

Une instruction-machine pouvant être divisée en plusieurs micro-opérations autorisées par les micro-commandes appropriées, il suffit donc de spécifier quelles micro-commandes il faut utiliser (en tenant compte évidemment de leur synchronisation et de leur séquencement) pour réaliser l'instruction-machine voulue.

WILKES a alors préconisé de fixer le choix de certaines micro-commandes pouvant s'exécuter parallèlement dans un élément de mémoire appelé **micro-instruction.**

Le modèle qu'il proposa utilise deux mémoires mortes (à tores de ferrite) à sélection linéaire C et S, une impulsion d'horloge adéquate sur un décodeur en arbre et de la logique de décision.

Précisons tout d'abord la représentation en réseaux matriciels des deux mémoires mortes à sélection linéaire (C et S).

La sélection d'une ligne détermine un mot en mémoire (appelé micro-instruction) dans lequel un bit à 1 signifie qu'il y a couplage entre la ligne et la colonne correspondante et un bit à 0 absence de jonction.

Les colonnes de la matrice C représentent les lignes de commande qui forment les supports physiques des signaux de commande destinés aux portes de commande.

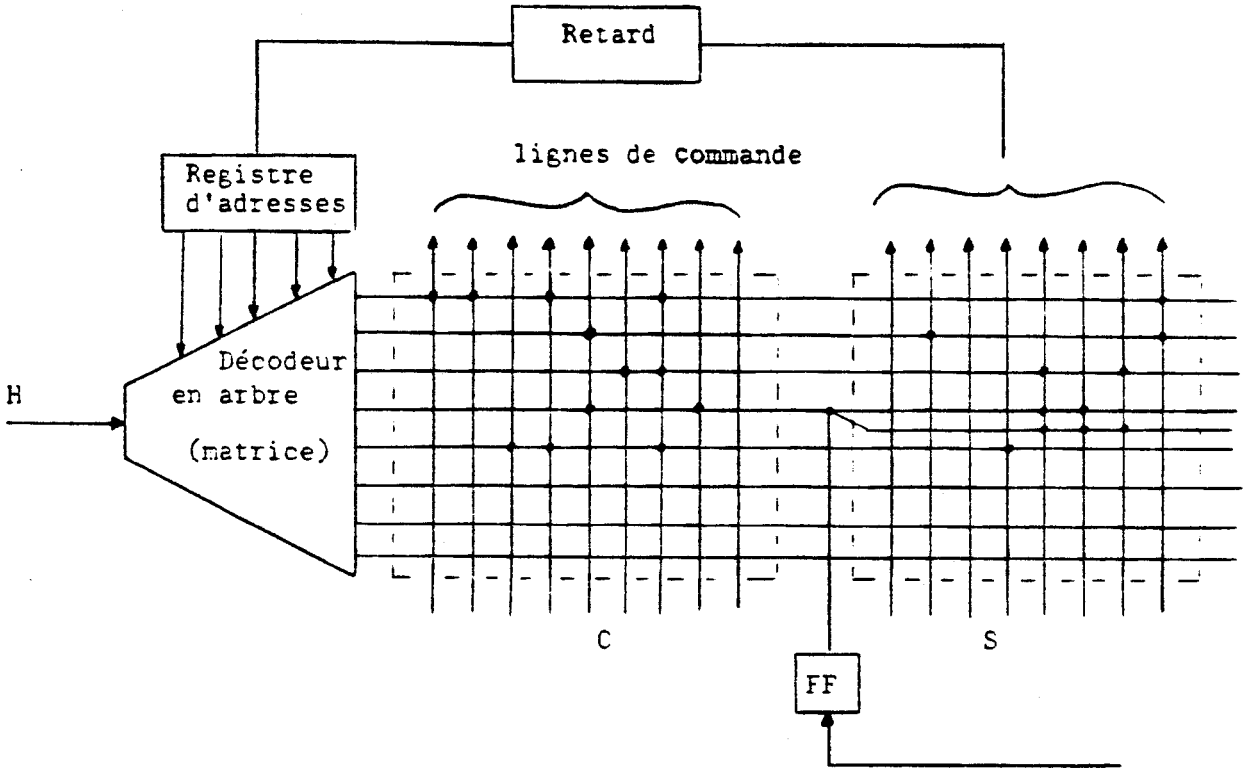


Figure n°11 : Le modèle de WILKES

Simultanément à l'ouverture de ces portes, des signaux sont émis sur les fils de sortie (colonnes de la matrice) du réseau S qui forment ainsi l'adresse de la micro-instruction suivante.

Cette adresse est transmise vers le registre d'adresse avec un certain retard déterminé par le temps de réaction des circuits électroniques mis en jeu au cours de l'exécution des micro-opérations.

WILKES introduit aussi 2 notions de base de la microprogrammation : il s'agit d'une part, du concept du **microprogramme** qui correspond à une séquence figée ou non de micro-instructions et d'autre part, du principe de la **mémoire de commande** (ROM ou même RAM) qui contient l'ensemble des micro-instructions.

La description de ce premier modèle de WILKES permet d'introduire la structure fondamentale d'une micro-instruction, quelque soit le format qu'on lui impose (nous verrons par la suite quelles formes on peut lui faire revêtir).

Une micro-instruction doit, en effet réaliser les deux fonctions suivantes :

1. Indiquer quelles micro-commandes doivent être activées ;

2. Fournir l'adresse de la micro-instruction suivante.

Cette deuxième information qui doit apparaître dans la micro-instruction peut correspondre à deux types de situation :

- Soit le choix de la micro-instruction suivante est fixé, quelque soit le contexte où s'exécute la micro-instruction courante ;

- Soit ce choix ne l'est pas.

Le premier modèle de WILKES envisage partiellement cette éventualité en insérant un mécanisme rudimentaire permettant un branchement conditionnel sur deux adresses de micro-instructions différentes.

Ce câblage de la ROM a d'ailleurs été rapidement abandonné mais le principe du branchement à une adresse déterminée dynamiquement reste fondamental dans tout système microprogrammé.

Les deux raisons principales en sont :

- La traduction par micro-instructions d'un branchement conditionnel existant dans le langage interprété du niveau supérieur ;

- L'utilisation de séquences de micro-instructions communes en vue de réduire la taille de la mémoire de commande.

Il convient peut-être d'expliciter ce deuxième point : chaque instruction de langage machine (ou même d'un langage de niveau supérieur) destinée à être exécutée par une machine microprogrammée est associée à un microprogramme.

La décomposition de chaque instruction en micro-opérations élémentaires et donc en microinstructions regroupant une ou plusieurs de ces micro-opérations, fait apparaître le nombre important de microinstructions communes aux diverses décompositions.

Il apparaît ainsi naturel de créer des microprogrammes faisant appel à un ou plusieurs sous-microprogrammes réalisant des séquences de micro-opérations fréquemment employées, sous réserve de disposer d'un mécanisme efficace de gestion des appels et des retours de sous-microprogrammes.

L'implémentation de mécanismes de branchement dans les microprogrammes peut se présenter sous des formes diverses ainsi que cela est montré dans [BGe 79].

Il faut enfin remarquer que les branchements peuvent être très fréquents : la plupart des microprogrammes écrits pour la gamme des DPS 7 de CII-Honeywell Bull comptent un branchement conditionnel toutes les 3 microinstructions, quand il ne s'agit pas de 9 microinstructions sur 10.

L'existence de branchements, conditionnels ou non, est une des causes majeures à la difficulté d'implanter des microprogrammes en mémoire de commande, surtout lorsqu'interviennent des besoins d'optimisation du microcode ; ce type de problème, auquel doivent faire face tous les microprogrammeurs, tend à être de plus en plus automatisé, étant donné la complexité qu'il peut atteindre dans le cas où la taille des microprogrammes s'accroît de manière importante.

Ces notions de base étant posées, il reste à trouver une définition précise et rigoureuse de la microprogrammation, applicable à toutes les techniques qui s'en réclament.

Beaucoup d'auteurs s'y sont essayés et si, pendant longtemps -depuis 1951 jusqu'au début des années 70- une définition globale satisfaisante s'était dégagée, des progrès technologiques récents et les transformations présentes ou prévisibles des fonctions de commande d'un ordinateur ont amené d'autres auteurs à contester la forme traditionnelle attachée à la microprogrammation.

Il est toutefois utile de connaître ce que recouvrait le terme de microprogrammation dans ses premières applications ; pour cela, la meilleure solution est peut-être de proposer la définition que donne Samir S. HUSSON dans son célèbre et impressionnant ouvrage : "Microprogramming - Principles and Practices" [Hus 70] :

> "La microprogrammation est une technique pour concevoir et implémenter comme une séquence de signaux de commande la fonction de commande d'un système de traitement de données afin d'interpréter les fonctions figées ou transformables dynamiquement de traitement de données.
> Ces signaux de commande, organisés élémentairement en mots qui sont rangés dans une mémoire de commande figée ou transformable dynamiquement, représentent les états des signaux qui commandent le flux d'information entre les différentes fonctions d'exécution ainsi que la transition ordonnée entre ces états".

Si l'on devait remettre en question cette définition, c'est surtout sur la notion de mémoire de commande que se porteraient les réserves bien qu'HUSSON ne précise pas qu'elle doive être nécessairement une entité matérielle indivisible, ce qui est une solution souvent abandonnée par les constructeurs de machines microprogrammées.

Cette définition a une autre caractéristique : elle reste très proche de la réalisation physique.

D'autres définitions plus "fonctionnelles", issues du développement considérable de la microprogrammation, sont apparues depuis ; DAVIDSON et SHRIVER [DSh 78] pour leur part, en distinguent deux :

1. La microprogrammation est le logiciel en mémoire de commande ROM (**firmware**)* qui fait qu'un ordinateur (la machine **hôte**) se comporte comme s'il était un autre ordinateur (la machine **cible**) ;

2. La microprogrammation est le niveau d'activité qui convertit les concepts d'un langage de programmation dans les opérations machine détaillées fournies par le constructeur ; on parle d'**émulation** quand le langage est de bas niveau et d'**interprétation** quand il est de haut niveau ; la microprogrammation est alors le programme dans la mémoire de commande qui permet au processus de réaliser une application.

Ces définitions permettent d'introduire certains des concepts largement répandus dans l'informatique et démontrent ainsi l'apport décisif de la microprogrammation à leur développement, voire à leur création (cf § 3.2 et 3.3).

3.1.2. Microprogrammation horizontale ou verticale

De même que le terme "microprogrammation" désigne un ensemble de techniques informatiques dont les contours varient suivant les auteurs, le critère d'horizontalité ou de verticalité, tel qu'on le rencontre dans la littérature, offre un éventail de définitions dont la compatibilité n'est pas toujours triviale.

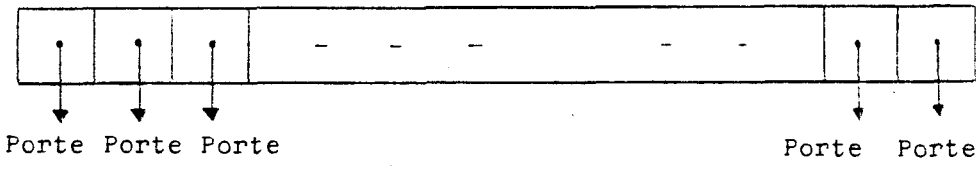
* Le terme "firmware" fut suggéré en 1967 par OPLER pour les microprogrammes conçus pour supporter le software ; il prévoyait la microprogrammation dans une mémoire rapide en lecture et lente en écriture comme étant la solution optimale pour les machines de la 4ème génération [Opl 67].

Selon RAMAMOORTHY [Ram 74], le niveau de commande d'un microprogramme sur les opérations élémentaires est déterminé par le format des micro-instructions dans le programme.

Il existe essentiellement 2 classes principales de formats de micro-instructions :

- Le format horizontal qui commande un seul niveau de portes ;
- Le format vertical qui commande des sous-systèmes fonctionnels durant plusieurs cycles machine.

Format horizontal :



Une micro-instruction horizontale contient plusieurs champs de micro-codes, chacun contrôlant directement une porte de commande.

Ces champs peuvent se réduire à un seul bit comme dans le modèle original de WILKES.

La finesse du contrôle permet une spécification compacte et efficace des micro-instructions.

Puisqu'une micro-instruction horizontale est exécutée en 1 cycle machine, la multiplicité des champs dans ce format offre la capacité d'utiliser pleinement les possibilités de traitement parallèle ; cette capacité, vitale pour l'optimisation des performances, est de plus en plus exploitée de manière automatique (voir § 3.2.3).

Un puissant ordinateur peut ainsi implémenter ses fonctions système de la manière la plus compacte et la plus efficace (voir § 3.2.2.1).

Il faut toutefois noter que dans une machine de taille relativement importante, la multiplicité des champs rend la microprogrammation plus complexe et donc pénible en l'absence de tout outil de développement adéquat.

Format vertical :

Code Opération	Registre Source	Registre Destination	Tag

Une micro-instruction verticale possède un format similaire à celui d'une instruction-machine traditionnelle.

C'est une séquence de fonctions élémentaires apparaissant habituellement spécifiées dans un champ code opération ; le format détermine également un ou plusieurs champs opérande, chacun représentant un registre source ou destination pour le transfert d'information.

Une micro-instruction verticale diffère d'une micro-instruction horizontale car elle représente une suite de micro-commandes demandant plusieurs cycles machine alors que la micro-instruction horizontale est exécutée en un seul cycle.

Le flux de données contenu dans une micro-instruction peut s'écouler en parallèle même si une micro-instruction verticale ne convient pas vraiment à la détermination d'opérations en recouvrement.

Notons que pour WILKES, la distinction entre les deux types de microprogrammation se fait de la manière suivante :

- Si les micro-commandes sont exécutées en 1 cycle machine, il s'agit de microprogrammation horizontale ;

- Si les micro-commandes sont exécutées en plus d'1 cycle machine, il s'agit de microprogrammation verticale.

Il faut cependant remarquer que la plupart des formats utilisés dans les machines microprogrammées se situent entre ces deux types et qu'il est souvent question de microprogrammation "plutôt horizontale" ou "plutôt verticale".

3.2. CARACTERISTIQUES PRINCIPALES

Le succès de la microprogrammation vient de ce que la technologie (initialement l'arrivée de ROM puis de RAM rapides), en la rendant compétitive vis-à-vis de la logique câblée, lui a permis de faire apprécier ses nombreux avantages, qui sont l'objet du premier paragraphe (3.2.1).

Puis, au traitement du problème souvent discuté des performances offertes par la microprogrammation (3.2.2), succèdent les différentes formes qu'elle revêt actuellement ainsi que la nature des travaux qui lui sont consacrés (3.2.3).

3.2.1. Les avantages d'ordre général

La microprogrammation, grâce à son principe tout autant qu'à ses diverses implémentations, procure au concepteur d'une machine, comme plus tard à son utilisateur, un certain nombre d'avantages dont l'utilité ou la qualité peuvent varier suivant les cas mais concourrent à justifier son choix dans un grand nombre de réalisations architecturales.

Ces avantages, HUSSON en a fait la liste exhaustive, tout au moins en ce qui concerne les aspects de conception et d'utilisation d'ordre général, dont voici les différents aspects :

1. Souplesse et adaptabilité : le code d'ordre peut être défini tard dans la période de conception de la machine, ce qui permet en parallèle la microprogrammation et l'implémentation du modèle du système. La microprogrammation permet aussi un grand degré de liberté dans les modifications qui surviennent lors de la conception de l'architecture du prototype ainsi qu'une importante capacité d'émulation une fois l'architecture établie.

2. Transformabilité : le simple remplacement de la matrice de commande permet de spécialiser la machine ou de sélectionner un jeu optimal d'instructions-machine pour aider le programmeur aux prises avec un environnement et des problèmes particuliers.

3. Facilité de conception, de maintenance et de contrôle : le principal inconvénient d'un système à commande câblée est que ces 3 aspects doivent être traités simultanément par le concepteur de la logique de commande. La microprogrammation permet au contraire, de mieux séparer ces 3 aspects et de résoudre avec plus de souplesse , donc plus de facilité, les problèmes qu'ils engendrent. Un simple bit de parité ajouté au mot en ROM permet, par exemple, de contrôler le séquencement des opérations.

4. Uniformité de conception : la microprogrammation donne un ordre, une uniformité et une modularité de conception de la fonction de commande comme cela existe pour la conception fonctionnelle d'une ALU. Ceci a pour avantage supplémentaire de former des microprogrammeurs compétents plus facilement que des concepteurs de logique de commande hardware.

5. Facilité d'enseignement : en raison de la simplicité et de la méthodicité de son approche, la microprogrammation est plus facile à enseigner aux ingénieurs de maintenance ainsi d'ailleurs qu'à tous ceux qui sont intéressés par l'architecture et l'organisation des systèmes.

6. Extension de la durée de vie d'un système : face à de nouveaux besoins, la microprogrammation peut transformer la fonction de commande pour actualiser ou assurer la reconversion d'un système inadapté.

7. Economie : il a été montré que la réalisation d'une architecture répondant aux besoins conjugués de la programmation commerciale et scientifique par l'intermédiaire d'un jeu d'instructions idéal, est plus chère en utilisant de la logique purement câblée qu'en utilisant la microprogrammation.

Ce dernier point fait allusion à la première véritable utilisation de la microprogrammation par un constructeur : la gamme IBM System 360 proposait en effet la même architecture logique (et donc une compatibilité parfaite entre tous les produits de la gamme) sous des formes différentes, la plus chère étant purement câblée et la moins chère naturellement totalement microprogrammée.

Par ailleurs d'immenses progrès furent et sont encore accomplis dans la réalisation et l'utilisation de la microprogrammation qui minimisent ou annulent ses tares originelles.

Les améliorations significatives intervenues dans les domaines de ses performances et de sa facilité d'utilisation (voir § 3.2.2 et 3.2.3) sont les causes principales de l'énorme intérêt que suscite depuis quelques années cette forme d'implémentation.

Il est à cet égard assez significatif de remarquer que la plupart des circuits en VLSI utilisent une mémoire de commande et que le MC 68000 par exemple possède même deux niveaux de microprogrammation : des micro-mots de 17 bits et des nano-mots de 68 bits.

Un autre exemple d'actualité symbolisant l'utilisation massive de la microprogrammation réside également dans la multiplication des circuits permettant la réalisation de microprocesseurs en tranches ("bit slice") développés par INTEL, Monolithic Memories Inc., Motorola, Texas Instruments ou Advanced Micro Devices Inc.

3.2.2. L'aspect performances

Le problème des performances revêt en ce qui concerne la microprogrammation deux aspects différents :

1. Dans quelle mesure une forme microprogrammée est-elle plus rapide que son équivalent logiciel ?

2. Dans quelle mesure une forme microprogrammée est-elle moins rapide que son équivalent câblé ?

3.2.2.1. Microprogrammation et logiciel

Un des éléments de réponse du premier point réside dans la multiplication des techniques de migration verticale (voir § 2.2.1).

Pour évaluer correctement l'impact de ces techniques, il importe d'abord de faire la différence entre l'amélioration de la vitesse d'exécution de la primitive "migrée"(qui est évidente et souvent très importante) avec l'effet que produit cette migration sur les fonctions d'un niveau supérieur ou les programmes d'application qui utilisent cette primitive.

Les mesures effectuées sur les systèmes utilisant les techniques de migration verticale font apparaître une amélioration globale moyenne de 50 % des performances d'application [SVa 78].

Le remplacement de certaines fonctions ou primitives logicielles dans des programmes d'applications fréquemment utilisés (en général, les systèmes d'exploitation et les compilateurs) par leur équivalent micro-programmé nécessite toujours une étude statistique préalable mais peut se produire sous deux formes différentes (en principe, non exclusives) :

- La substitution a lieu **a posteriori** dans des machines micro-programmables : il s'agit de techniques de migration verticale ;

- La substitution a lieu **a priori**, dès la conception de l'architecture le plus souvent dans des machines microprogrammées : il s'agit par exemple, du cas de l'iAPX 432.

Il semble intéressant de pouvoir bénéficier sur une même machine des avantages cumulés de ces deux méthodes : la microprogrammation, surtout si elle fait appel aux techniques d'optimisation*(voir § 3.2.3), devient alors un facteur incontestable d'accélération de la vitesse d'exécution des programmes.

3.2.2.2. Microprogrammation et matériel

La principale critique formulée à l'égard de la microprogrammation et qui, à l'origine, a retardé son développement est sans doute sa lenteur, présumée ou effective, vis-à-vis de la logique câblée.

L'arrivée sur le marché de mémoire ROM puis RAM ayant un temps d'accès de moins de 50 ns a été la première étape d'une série d'améliorations technologiques qui ont amené la microprogrammation à contester jusqu'au

* NICOLAU et FISCHER [NFi 81] affirment que grâce à l'emploi de techniques évoluées (qu'ils nomment "oracle") de compaction globale de microprogrammes de type horizontal, l'exécution de programmes scientifiques types peut être accélérée de 3 à 1000 fois sur une même machine.

monopole que semble posséder encore la logique purement combinatoire dans le domaine des applications demandant d'énormes capacités de calcul, comme le traitement d'images satellite, les calculs météorologiques, les analyses sismiques ou les systèmes de défense militaire.

Ainsi certains projets ([RVa 77], [SGi 81]) proposent-ils comme alternative aux "super-ordinateurs" généralement affectés à ces tâches, l'emploi de réseaux distribués de processeurs spécialisés, à hautes performances et microprogrammés horizontalement..

Il est généralement admis que la plupart des fonctions nécessitant un séquencement dans leur déroulement sont plus rapides lorsqu'elles sont réalisées sous forme de fonctions booléennes dans une logique purement câblée, que lorsqu'elles se fondent dans le moule d'une implémentation microprogrammée (dans la même technologie, cela va de soi).

Ce principe reste vrai tant que les fonctions considérés demeurent d'une complexité raisonnable ; en fait, la logique purement câblée doit faire face à trois types d'obstacles :

1. L'augmentation de la complexité des fonctions réduit l'écart de performance qui la sépare de la logique microprogrammée ;

2. Le coût de conception et de réalisation qu'elle entraîne tend à la réserver à des applications hautement spécifiques ;

3. La tendance actuelle des circuits VLSI est de dominer les problèmes de complexité engendrés par la fantastique progression de l'intégration par une conception de plus en plus automatisée et hiérarchiquement structurée ([Vui 83] et [Anc 83a]).

La logique sera donc de plus en plus implantée sur le silicium sous forme de modules prédéfinis et standardisés, même en ce qui concerne les parties opératives.

Et si actuellement les vues sont encore partagées sur l'implémentation de la partie commande d'un microprocesseur ([Anc 83b]), le critère de performances n'est plus le handicap légendaire de la logique microprogrammée ; en effet, comme le remarque TREDENNICK [Tre 82], il serait difficile de deviner lequel des trois plus importants microprocesseurs 16 bits (Z 8000, 8086, MC 68000) est de conception microprogrammée en mesurant leurs performances respectives grâce à un analyseur de logique fixé à leurs pattes.*

La microprogrammation tend donc de plus en plus, tant sous sa forme statique (machines microprogrammées) que sous sa forme dynamique (machines microprogrammables), à rejetter l'étiquette de technique d'implémentation agréable mais lente.

3.2.3. Environnement de production

La spécificité de la microprogrammation (notamment son rôle dans la conception de l'architecture et son caractère intimement lié au matériel) demande un environnement quelque peu différent de celui de la programmation classique (cette différence est illustrée par les figures Nos 12 et 13 qui sont des exemples empruntés à [Cha 75]).

En fait, comme le remarque WANG [Wan 82], le principal élément qui rend le développement du microcode plus difficile que celui du logiciel est que, dans presque tous les cas (tout au moins en ce qui concerne la microprogrammation comme outil de conception architecturale), les micro-programmes sont développés concurremment à l'élaboration du matériel.

* TREDENNICK pense même que la machine microprogrammée (le MC 68000) est plus rapide bien qu'il reconnaisse que ce n'est pas la même architecture qui est implémentée sur les trois puces.

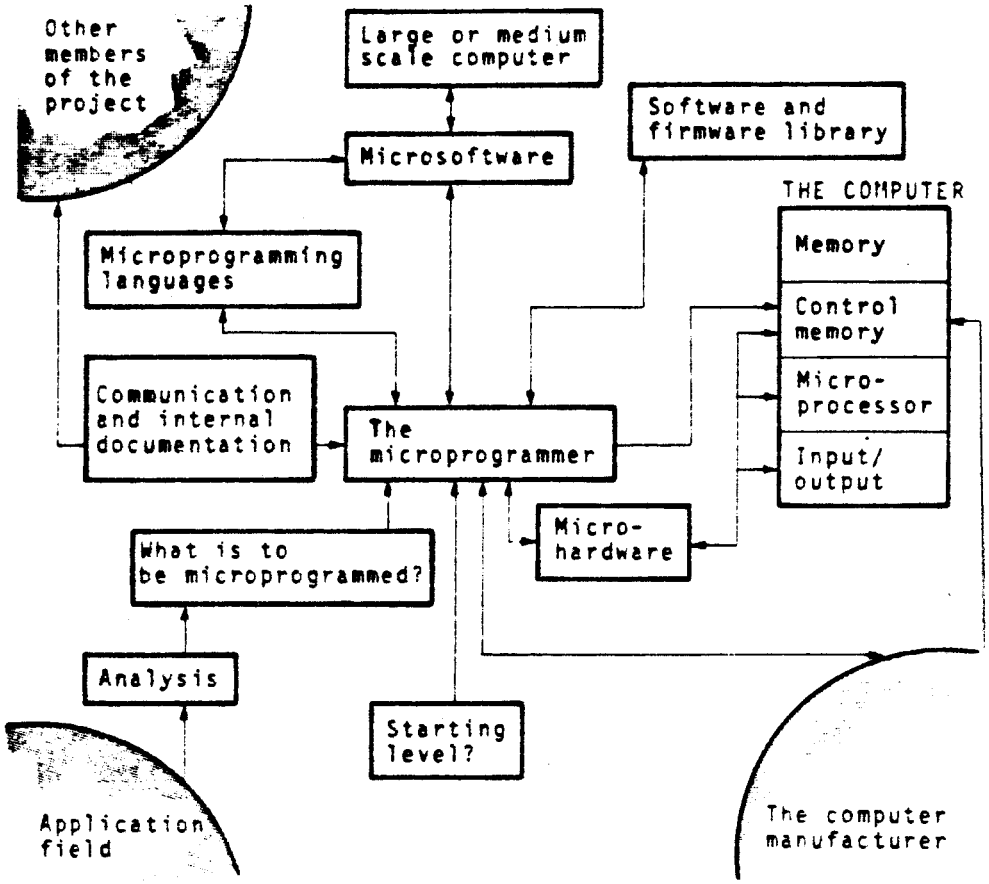


Figure n°12 : L'environnement du microprogrammeur

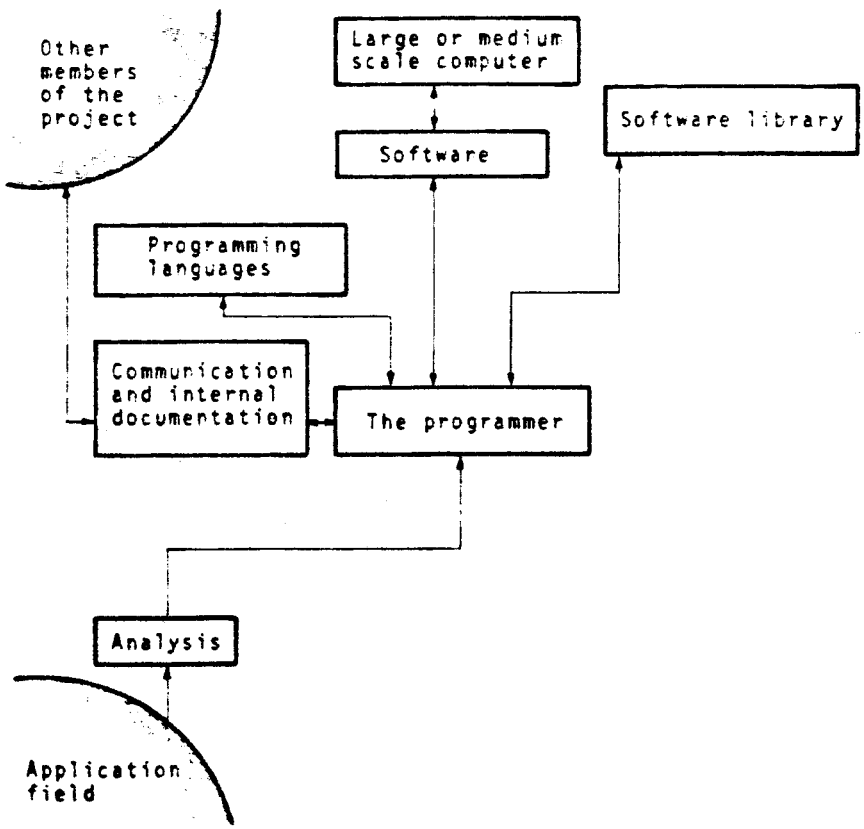


Figure n°13 : L'environnement du programmeur en logiciel

La production de quantités souvent importantes de microcode a cependant obligé les microprogrammeurs à se doter d'outils similaires à ceux apparus pour la production de logiciel et destinés à assurer la conception, la spécification, la construction, la vérification, le test, la mise au point et la maintenance du microcode.

Mais, avant d'aborder ces thèmes, il est utile de présenter les différents aspects liés à l'implantation de la microprogrammation et qui sont un élément important de sa caractérisation.

3.2.3.1. Supports et techniques d'implantation

La réalisation (et même parfois le concept) de mémoire de commande (cf § 3.1.1.) d'une machine microprogrammée a subi de profondes transformations depuis le modèle à tores de ferrite qu'avait imaginé WILKES.

Le principe reste le même dans le cas de machines pour lesquelles la microprogrammation n'est que la forme d'implémentation de leur fonction de commande : la mémoire de commande est alors une ROM, éventuellement une EPROM, qui a suivi l'évolution de la technologie puisque la mémoire de commande 16 bits du MITRA15 était réalisée en circuits intégrés bipolaires MSI pour une capacité de 512 ou 1K mots alors que la mémoire de commande également de 16 bits de l'Intel iAPX 432 General Data Processor est intégrée à l'une des deux puces qui le composent (le 43201) pour une capacité de 4K mots.

Les machines microprogrammables utilisent quant à elles des mémoires RAM, le plus souvent chargées à partir de la mémoire centrale, et peuvent même ne pas avoir de mémoire de commande physique : ainsi dans le Burroughs B 1701, tous les microprogrammes sont stockés en mémoire centrale.

L'organisation logique de la mémoire de commande (ROM ou RAM) peut revêtir plusieurs aspects (voir figure n°14) :

1. La structure la plus simple et la plus courante qui consiste à mettre une microinstruction par mot mémoire ;

2. La structure consistant à mettre 2 microinstructions par mot mémoire, ce qui permet de les lire en même temps ;

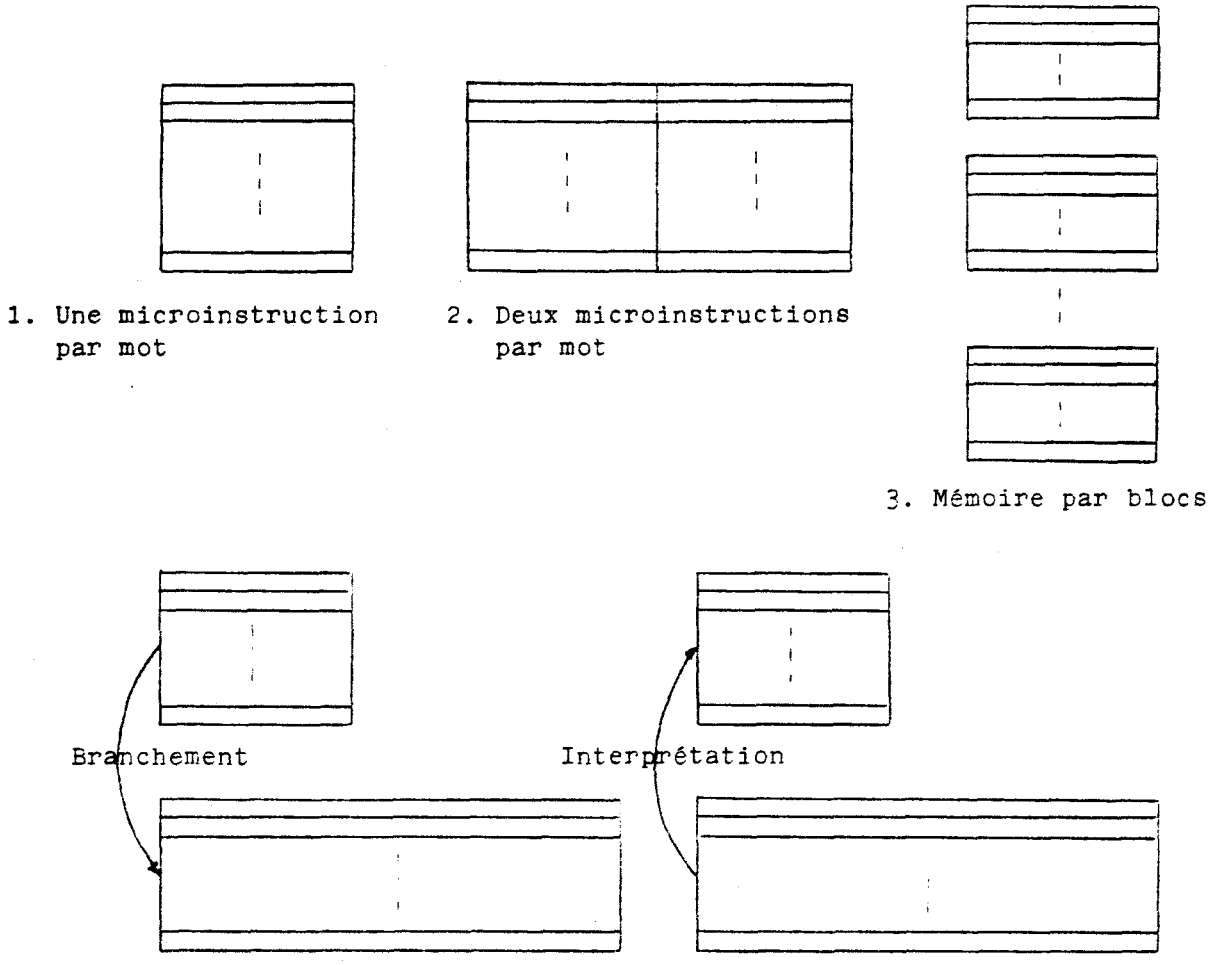
3. La structure consistant à diviser la mémoire de commande en plusieurs blocs, ce qui permet de réduire les adresses des microinstructions dans un même bloc ;

4. La structure consistant à diviser la mémoire de commande en 2 unités de largeur différentes ; la moins large contient les microinstructions demandant peu de bits (transfert de données entre registres, initialisation d'une microinstruction se trouvant dans l'autre unité) et la plus large contient les autres microinstructions, ce qui est une autre forme de réduction de la taille de la mémoire de commande ;

5. La structure à 2 niveaux de microinstructions qui permet une meilleure flexibilité dans la conception des microinstructions du premier niveau (souvent de type vertical) ; cette structure connaît un certain succès ainsi qu'en témoignent les nombreuses réalisations : MITRA 15, QM.1 de NANODATA, IBM 38...

Le format des microinstructions, horizontal ou vertical, a déjà été évoqué (§ 3.1.2.) : il est, avec le temps d'accès à la mémoire de commande, le principal élément qui détermine le temps d'exécution d'une microinstruction puisqu'il décide de la complexité du décodage à réaliser après la lecture de la microinstruction en mémoire.

Le niveau d'encodage des microinstructions, qui est étroitement lié à la caractérisation vertical-horizontal du microcode, se traduit donc par le volume de logique de décodage (qui peut être nul) associé à chaque champ de la microinstruction (qui peut se réduire à un unique bit).



1. Une microinstruction par mot
2. Deux microinstructions par mot
3. Mémoire par blocs
4. Partage basé sur la longueur des microinstructions.
5. Microprogrammation à deux niveaux

Figure 14 : Les différentes formes possibles de la mémoire de commande

Une technique particulière consiste également à conditionner le décodage de plusieurs champs par la valeur prise par un champ particulier de la même microinstruction ; c'est un type d'encodage appelé parfois "bit steering", par référence à l'Honeywell H 4200 qui fut un des premiers à l'adopter.

Quelle que soit leur fonction (logique de commande figée, émulation, microdiagnostics,...), les microprogrammes doivent être chargés sous une forme offrant le maximum de fiabilité, ce qui implique l'existence d'outils de développement évolués, et le maximum d'efficacité, ce qui implique des techniques d'optimisation en taille et en performances.

3.2.3.2. Les outils de développement

Il est fréquent d'établir un parallèle entre le développement actuel des techniques de production de microcode et le développement précédent des techniques de production de code machine.

La plupart des microprogrammeurs ont pendant longtemps (et même encore actuellement) entièrement réalisé "à la main" la forme définitive de leurs microprogrammes avant de les tester sur un simulateur logiciel ; cette méthode a ainsi contribué à rendre souvent la microprogrammation confidentielle et fastidieuse et les microprogrammeurs (choisis généralement parmi les débutants ou les sous-traitants) solitaires et peu productifs (ainsi qu'en témoigne la figure n° 15 tirée de [Cha 75]).

Aids to microprogramming	*Instructions per man-day*	*Total time to produce 10 000 microinstructions*
Simulator *	2	18 years
Microprogram assemblers and loaders	5 to 10	5 to 9 years
High level language microprogramming	10 to 25	2 to 5 years

Figure n°15 : Productivité de la microprogrammation

* Il s'agit du cas où le microprogrammeur ne possède qu'un simulateur pour tester ses microprogrammes, c'est-à-dire un programme qui simule l'action des microprogrammes sur une représentation logicielle de l'architecture matérielle ; l'écriture et les calculs d'implantation en mémoire de commande restent à la charge du microprogrammeur.

Le premier type d'aide offert pour l'écriture du microcode fut ce qu'il est convenu d'appeler les **microassembleurs** par analogie avec les assembleurs classiques (ils sont parfois appelés aussi **méta-assembleurs**).

Ils en partagent d'ailleurs les caractéristiques puisque, comme eux, ils fournissent un jeu de mnémoniques associées à chaque micro-opération (type horizontal) ou chaque microinstruction (type vertical), contrôlent éventuellement la légalité des microinstructions ainsi assemblées (type horizontal), permettent l'utilisation d'étiquettes (labels) pour définir des emplacements, effectuent le placement des microinstructions dans la mémoire de commande, établissent des tables de "cross-références", des topographies de chargement, etc. (ils peuvent même être associés à des **micro-éditeurs de liens** [Mcy 80]).

Les exemples de microassembleurs ne manquent pas, surtout chez les constructeurs de machines microprogrammées ou microprogrammables : que ce soit pour des microprocesseurs en tranches comme l'INTEL 3000 [Wil 76], des machines 32 bits à haute performance comme Data General ECLIPSE MV/8000 [Fir 80] ou le BBN Microprogramamble Building Block [Gla 81] ou encore la plupart des premières machines microprogrammables comme le HP 2100S, l'Interdata Model 85,...

L'étape suivante dans l'amélioration des outils de production du microcode est, logiquement, le recours aux langages évolués de microprogrammation qui conduit à l'élaboration de compilateurs nécessairement optimisants (voir § 3.2.3.3).

L'usage de langages de haut niveau, s'il demeure reconnu depuis longtemps dans son principe ([Hus 70], [DSh 78]), a rencontré de nombreuses difficultés dans son développement : le caractère apparemment trop proche du matériel de la microprogrammation, la taille à l'origine relativement réduite des microprogrammes, l'absence de véritable concertation entre les microprogrammeurs (souvent pour des raisons de confidentialité) sont quelques unes des causes généralement invoquées.

En fait, comme le souligne Marleen SINT [Sin 80], le principal problème n'est pas la conception mais l'implémentation de tels langages. Ainsi, les langages de microprogrammation non dépendants machine sont-ils souvent dérivés de langages existants : ALGOL 60 pour SIMPL [RTs 74] ou STRUM [Pat 81], PASCAL pour S* [Das 78] ou CHAMIL [Wei 80], APL [HHT 81], TAL [Bar 81] et même ADA (les microprogrammes de l'iAPX 432 furent d'abord écrits en ADA*).

L'implémentation de ces langages, c'est-à-dire la fabrication de compilateurs efficaces, se heurte par contre à trois types de difficultés :

- La structure du microcode est plus complexe que le code machine conventionnel, ce qui entraîne, notamment pour le type horizontal, la manipulation d'un nombre souvent considérable de microinstructions**;

- Le facteur performances est fondamental dans la réalisation des microprogrammes, ce qui implique une optimisation maximale lors de la génération du microcode (en particulier, l'exploitation intensive du parallélisme lorsqu'il est de type horizontal) ;

- La plupart des machines primitives microprogrammables offre un choix limité à la création de nouvelles primitives (la microprogrammation est trop liée à un certain type d'applications), ce qui a pour effet de restreindre l'intérêt, donc l'usage de leur microprogrammation par l'utilisateur.

C'est pourtant paradoxalement l'accroissement des difficultés (augmentation du nombre de microinstructions et de la taille des microprogrammes associée au besoin impératif de performances) qui condamne les microprogrammeurs à se forger les outils puissants et automatiques qui augmenteront simultanément leur productivité et la fiabilité de leurs microprogrammes.

* Il n'existe cependant pas de compilateur et les microprogrammes furent traduits à la main en plusieurs étapes jusqu'à la forme définitive du microcode [Ham 81].

** Si le DEC PDP 11/70 n'utilisait que 256 microinstructions, le DEC VAX 11/780 en utilise plus de 5000 ; HEWLETT PACKARD, quant à lui, a sorti récemment un micro-ordinateur mono-puce possédant plus de 9000 microinstructions [BDF 81] !

La figure n°16 offre un exemple de compilation de programmes écrits en langage évolué en microcode horizontal grâce à l'utilisation de langages intermédiaires et de techniques diverses d'optimisation de types logiciel et matériel (la machine cible est un TRW 2 AU 80 [SGi 81].

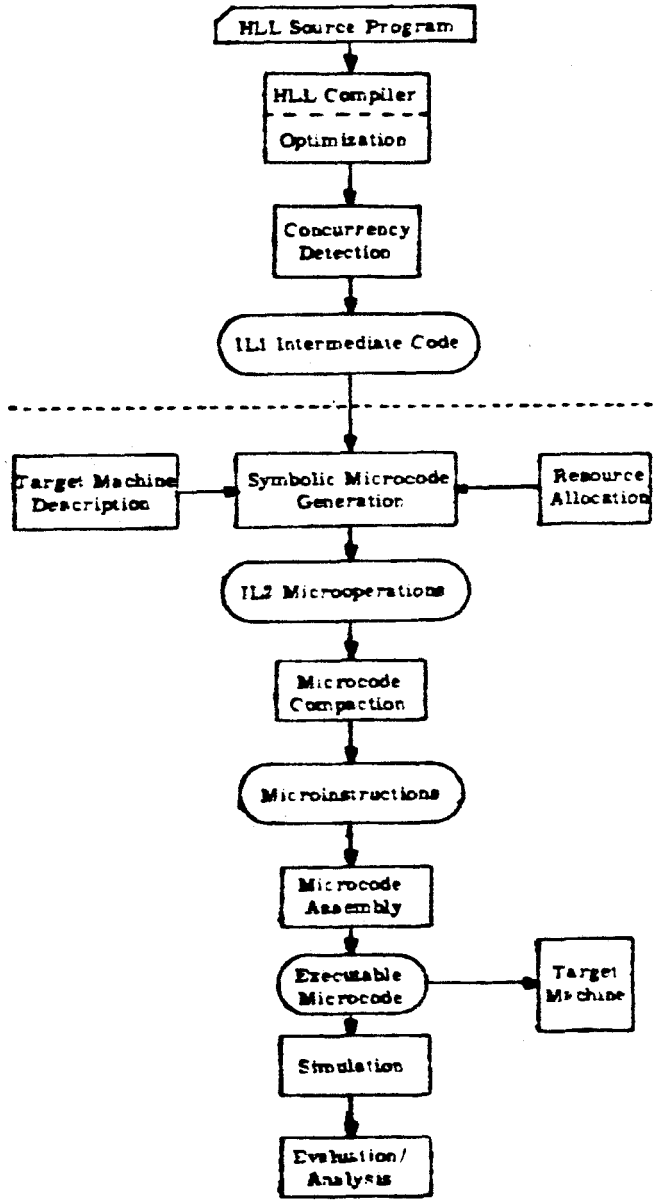


Figure n°16 : Procédure de compilation d'un langage évolué en microcode

En raison de son implantation au plus bas niveau de l'architecture matérielle, un microprogramme se doit d'être parfaitement correct, tant dans le but de ne pas fausser les primitives logicielles qui l'utilisent que dans le cas où il est le support d'un **microdiagnostic** (voir § 3.3.2).

La vérification, le test et la mise au point des microprogrammes ont donc toujours été particulièrement soignés ; la relative petite taille de ces derniers a, il est vrai, favorisé le développement et la qualité des méthodes qui en assurent la réalisation (c'est du moins l'avis généralement émis sur la meilleure praticabilité de ces techniques vis-à-vis de leurs équivalents en logiciel [DSh 78]).

Les techniques de mise au point sont donc nombreuses (de multiples exemples en sont donnés dans [DSh 78]) et vont du listing de cross-references (fourni par exemple par l'assembleur de microcode Micro 8 [Gre 81]) au vérificateur symbolique (tel qu'il existe dans le projet MBB [Gla 81]).

La vérification est d'autant plus importante (et difficile) que les microprogrammes, qu'ils soient de type vertical ou horizontal, sont presque toujours optimisés, manuellement ou par l'intermédiaire de compilateurs-générateurs de microcode.

L'optimisation des microprogrammes, comme cela a déjà été souligné, est une composante fondamentale à tout système, automatisé ou non, de génération de microcode.

3.2.3.3. Les techniques d'optimisation

Deux types de motivations (dont les effets se rejoignent le plus souvent) sont à l'origine du développement des techniques d'optimisation :

1. La réduction de la taille du support du microcode ; l'objectif est un gain de temps lors du transfert de la mémoire centrale vers la mémoire de commande dans le cas d'une microprogrammation dynamique, et le gain en surface occupée (voire en coût de fabrication) par le mémoire de commande (ROM ou RAM), problème particulièrement critique dans la conception des microprocesseurs en VLSI ;

2. La réduction du temps d'exécution des microprogrammes ; celle-ci est obtenue par la réduction de la taille des microprogrammes qui s'effectue en exploitant au maximum le parallèlisme offert en mode horizontal et en appliquant toute méthode d'optimisation locale ou globale que permet la forme du microcode.

L'ensemble des méthodes employées dans la réduction de la taille du microcode avait été partitionné en 1976 par AGERWALA [Age 76] de la façon suivante :

- Techniques de réduction de la taille en mots par la compaction du microcode horizontal et/ou par des optimisations semblables à celles largement répandues en logiciel ;

- Techniques de réduction de la taille en bits ; ce type de méthodes est évidemment utilisé lors de la conception de l'architecture matérielle et peut parfois être en conflit avec des techniques de compaction en mode horizontal ;

- Techniques de réduction du nombre d'états en considérant la fonction commande à microprogrammer comme un automate d'états finis ;

- Techniques heuristiques diverses dont la plus connue semble être le schéma de "commande résiduelle" proposé par FLYNN et ROSIN [FRo 71].

Ces trois derniers ensembles de techniques de réduction de la taille de la mémoire de commande visent essentiellement les objectifs définis par le premier type de motivation évoqué au début de ce paragraphe, et sont à rapprocher des techniques utilisées en VLSI comme par exemple, l'approche CCROM ("Compressed Control ROM") qui a permis de réduire du triple la mémoire de commande du TMS 9995 [Gut 80].

Le premier ensemble de méthodes répertoriées par AGERWALA s'est, quant à lui, singulièrement agrandi depuis ; la multiplication des machines véritablement microprogrammables par l'utilisateur a certainement favorisé le besoin et donc la création de telles méthodes.

Ces méthodes peuvent être grossièrement divisées en deux catégories :

a) Celles qui s'appliquent à la plupart des microcodes de type vertical et aux formes "machine-independent" que peuvent prendre dans un premier temps les microcodes de type horizontal ;

b) Celles qui s'appliquent aux microcodes de type essentiellement horizontal qui présentent des possibilités d'exploitation du parallélisme de bas niveau.

Le premier ensemble de techniques d'optimisation (a) regroupe surtout les techniques inspirées ou adaptées de celles qui existent dans les compilateurs usuels de langages de programmation évolués (FORTRAN, PL/1, PASCAL...) : élimination des redondances, déplacement de code, propagation des constantes,...

Ces optimisations s'inscrivent pour la plupart dans des schémas évolués de compilation de langages (généralement évolués) de microprogrammation et utilisant souvent un ou plusieurs langages intermédiaires (voir figure n°16) comme le prouvent de récents travaux ([BPa 80], [Mar 81], [SGi 81], [Pat 81], [PGS 81] ou [Vej 82]) ;

Il est important de remarquer que la majorité de ces projets appliquent les solutions préconisées par AGERWALA, c'est-à-dire principalement l'usage de langages évolués et de langages intermédiaires même pour la génération de microcode de type horizontal, options jusqu'alors peu utilisées (il faut citer néanmoins l'optimisation de microprogrammes de type vertical [KRa 71] et celle des microprogrammes de l'Interdata Model 85 [Lit 75]).

Le deuxième type d'optimisations (b) consiste en deux formes d'optimisation de microcode horizontal : la compaction locale et la compaction globale.

Le but de la compaction, qu'elle soit locale ou globale, est de placer plusieurs micro-opérations pouvant s'exécuter dans le même temps de cycle dans chaque microinstruction le plus astucieusement possible afin de minimiser le nombre de microinstructions donc la taille du microprogramme.

Le placement est tributaire :

- D'une part, de la précédence des données, c'est-à-dire l'ordre partiel imposé par la lecture et l'écriture de la mémoire de commande ;

- D'autre part, de l'usage autorisé des ressources de la machine sur laquelle le microcode s'exécutera.

Les premières méthodes, en particulier manuelles, de compaction furent surtout locales ([Dew 76], [Mal 78], [LDS 80], [NFi 81],...) ; elles doivent cette appellation au fait qu'elles n'opèrent que sur les séquences de microinstructions définies entre chaque branchement (nommées parfois SLMs -Straight Lines Microprograms- mais le plus souvent BBs -Basic Blocs-).

Rapidement, des techniques d'optimisation prenant en compte les branchements furent élaborés ([Das 77], [Woo 79]) et, en particulier, une méthode basée sur la fréquence d'utilisation des BBs due à FISCHER [Fis 79] qui a entraîné quelques projets d'implémentation ([Poc 80], [PGS 81], [Veg 82]).

Le problème posé par l'automatisation de ces techniques d'optimisation, c'est-à-dire leur prise en charge par des systèmes de compilation-génération de microcode, est celui, critique, de leur efficacité.

Un premier élément de réponse à cette légitime interrogation réside d'abord dans la quasi-impossibilité d'obtenir véritablement une forme "optimale" du microcode, quelles que soient les méthodes employées ; ce problème, défini comme NP-complet, a été mis en évidence par De WITT [Dew 76].

Le second argument justifiant l'emploi d'outils automatiques de génération et d'optimisation du microcode consiste en des études comparatives entre des microprogrammes écrits à la main et leurs équivalents générés automatiquement. Les résultats sont encourageants ainsi que le montrent SHERAGA et GIESER [SGi 81] qui ont constaté l'efficacité du microcode compilé (plus grand en volume de 5 à 10 % que celui écrit à la main mais environ 70 fois plus rapide à écrire), et surtout PATTERSON [Pat 81] qui, par l'intermédiaire de son système STRUM émulé sur HP-2115, obtient une version compilée à la fois plus rapide et plus courte que son équivalent émulé suivant des techniques traditionnelles utilisant un langage micro-assembleur.

3. 3. CONTRIBUTION A LA REALISATION DES NOUVEAUX CAHIERS DES CHARGES

Bien que la tendance générale soit de considérer la microprogrammation (firmware) comme un domaine spécifique intermédiaire entre le logiciel (software) et le matériel (hardware), il semble que les définitions peuvent en l'occurrence cacher une réalité qu'il convient de ne pas perdre de vue : la microprogrammation, quels que soient les outils de développement qui la font parfois assimiler au logiciel (langages évolués, compilateurs,...) est toujours étroitement liée aux ressources matérielles et doit donc être considérée comme une forme particulière d'implémentation matérielle.

Le choix d'une fonction commande microprogrammée ou microprogrammable doit en effet s'effectuer lors de la conception même de la machine dont la configuration matérielle reflète les options architecturales définies au niveau logique.

L'implémentation de mécanismes ou de primitives évolués sous forme microprogrammée doit donc être considérée comme une méthode particulière parmi celles qui contribuent à l'augmentation de la part prise par le matériel dans le schéma d'interprétation des langages ou des concepts de haut niveau.

L'apport de la microprogrammation, tant dans la réduction "des" fossés sémantiques (cf § 2.1 et 2.2) que dans la réalisation partielle des nouveaux cahiers des charges (cf § 1.4), mérite, à ce titre, d'être développé.

3.3.1. Réduction des fossés sémantiques

"Les" fossés sémantiques sont ceux, plus ou moins arbitrairement, définis dans le sous-chapitre 2 :

- Le fossé sémantique résultant de l'augmentation de la puissance sémantique et de la complexité des langages de programmation (cf § 2.1) ;

- Le fossé sémantique induit par l'ensemble des concepts évolués ne dérivant pas directement de la sémantique de ces langages de programmation (cf § 2.2).

3.3.3.1. Interprétation des langages machines de haut niveau

L'utilisation de la microprogrammation comme support privilégié dans le schéma d'exécution de langages évolués n'est pas nouvelle puisque dans le milieu des années soixante, MELBOURNE et PUGMIRE présentaient déjà un compilateur FORTRAN réalisé en microcode sur un petit ordinateur [MPu 65].

Depuis, l'usage de la microprogrammation dans le schéma d'exécution des langages de programmation a été préconisé, sous des formes diverses, par un grand nombre de concepteurs (le sujet tel qu'il apparaissait dans le milieu des années soixante-dix est traité dans [BSO 75]) ; cet usage, déjà largement reconnu dans son principe, trouve actuellement de nouvelles justifications dans les progrès accomplis dans le domaine de la microprogrammation en matière d'efficacité et de fiabilité (cf § 3.2).

Dans la présente étude, les arguments en faveur d'un langage machine de haut niveau sont développés dans le paragraphe 2.1.2.

Ce type d'approche consiste en l'adoption d'instructions machines sémantiquement riches et donc plus difficiles à interpréter par le matériel que les habituelles instructions machine de bas niveau.*

La complexité de la fonction commande de la machine qui en résulte justifie alors pleinement le choix d'une implémentation microprogrammée; cet avis est d'ailleurs partagé par un certain nombre de concepteurs de circuits VLSI comme LI [Li 82] qui estime que la meilleure implémentation possible pour des architectures supportant des fonctions complexes réside dans le choix de systèmes VLSI microprogrammés.

Il apparaît donc que l'adoption de la microprogrammation dans le schéma d'interprétation du langage machine "intermédiaire" (type b dans § 2.1.1) est la solution qui répond le mieux aux objectifs souvent contradictoires d'efficacité, de fiabilité et de facilité de conception puis d'évolution (dans le cas d'une machine microprogrammable pour ce dernier point), compte tenu de ses caractéristiques, largement exposées au paragraphe 3.2.

* Ce problème est parfaitement similaire à celui développé par BROADBENT [Bro 75] qui énumérait les avantages de l'interprétation (sur la compilation) des langages évolués et justifiait de plus l'emploi de la microprogrammation dans le cas d'un schéma interprétatif.

3.3.1.2. Support de mécanismes évolués

Le paragraphe 2.2 a mis en évidence les mécanismes et les primitives classiquement supportées par le logiciel de base et dont l'intégration dans le matériel serait un facteur d'amélioration de performance et de fiabilité.

Les techniques de migration verticale, dont les avantages sont évoqués précédemment (cf § 2.2.1 et § 3.2.2.1), sont l'exemple parfait de ce que, seule, la "microprogrammabilité" peut apporter : une contribution à la réduction d'un certain type de fossé sémantique, qui est à la fois modulaire, modulée et évolutive.

L'intégration de certains mécanismes évolués dans l'architecture matérielle ou logique dès la conception est également facilitée dans le cas d'une fonction commande microprogrammée qui permet alors l'implantation de tels mécanismes tout en limitant le supplément de matériel qu'ils pourraient nécessiter autrement.

FRIEDER [Fri 75], après avoir répertorié les différentes formes que prend le support "hardware-firmware" des traditionnelles fonctions de systèmes d'exploitation (reconnaissance rapide et gestion des évènements, élimination des déplacements excessifs de données par des mécanismes d'adressage sophistiqués, mécanismes de protection de tous genres,...) estime même que le rôle des techniques de microprogrammation dans le support de fonctions de systèmes d'exploitation peut être démontré de façon optimale dans les architectures distribuées.

3.3.2. Résumé des avantages spécifiques

La microprogrammation, que ce soit à travers ses avantages inhérents ou issus de son adéquation au support de principes particuliers, peut apporter un certain nombre de réponses aux nouveaux cahiers des charges destinés aux ordinateurs (§1.4) ;

Elle permet, en particulier, d'assurer :

1. La sûreté de fonctionnement, grâce à :

- Sa bonne adéquation à supporter les mécanismes de protection, de contrôle et de mise au point généralement à la charge du logiciel de base ;

- Sa capacité à fournir des **microdiagnostics***, beaucoup plus proches de la réalité matérielle que ne le peuvent les diagnostics établis classiquement en logiciel, qui rendent la maintenance plus facile et qui peuvent même permettre à la machine de fonctionner en mode dégradé ;

2. La facilité d'utilisation, grâce à :

- Sa capacité à offrir un support privilégié à l'interprétation de langages machines évolués (cf § 3.3.1.1) qui favorisent l'usage des langages de programmation évolués ;

- Son adéquation déjà évoquée à implémenter les outils nécessaires à la mise au point et surtout à la maintenance grâce aux micro-diagnostics **automatiques** ;

3. La capacité d'extension et d'adaptation, grâce à :

- Son principe même qui, en tant que logique de commande aisément (re) modifiable, en a fait le fondement de **l'émulation**, concept apparu avec elle qui signifie la possibilité pour une machine (hôte) d'exécuter des instructions appartenant à une autre machine (cible) ;

* Les avantages des microdiagnostics sont développés dans [RCh 72] ; ils sont souvent fondamentaux dans la maintenance (souvent automatique) des machines, petites ou grosses, de l'ECLIPSE MV/8000 à l'IBM 370/125.

- Son apport dans le mouvement incitant à la programmation en langage évolué (voir le point précédent) qui favorise la portabilité des programmes et donc la compatibilité des systèmes.

Dans la mesure où la réalisation de l'ensemble des spécifications inscrites dans ce qui est regroupé sous le terme de nouveaux cahiers des charges conduit à la prise en compte par les architectures logiques et donc matérielles de fonctions d'une complexité significative, la micro-programmation , dont la compétitivité vis-à-vis de la logique câblée en matière de performance et de fiabilité ne fait que croître, reste la meilleure solution possible, tout au moins dans le cadre d'un projet de machine qui a pour objectif de garder un coût raisonnable.

3.4. CONCLUSIONS

S'il fallait résumer le rôle occupé par la microprogrammation au sein du monde informatique, cela pourrait être de la façon suivante : la microprogrammation, à travers les différentes formes qu'elle revêt, se généralise de plus en plus chez les constructeurs d'ordinateurs, les réalisateurs de prototypes ou de systèmes dédiés et même les utilisateurs (dans le cas des machines microprogrammables) mais, le plus souvent, ne bénéficie pas encore d'outils puissants et automatiques de développement, malgré les nombreuses recherches réalisées en laboratoire (cf § 3.2.3).

Actuellement, la microprogrammation s'impose dans trois domaines importants et devrait, par ses progrès en performances et en facilité d'utilisation, accentuer son emprise :

a) Chez les constructeurs de minis ou gros ordinateurs, qui traduisent un maximum de fonctions habituellement logicielles sous forme microprogrammée (technique plus ou moins évoluées de migration verticale, cf § 2.2.1 et 3.2.2.1) ;

b) Chez les concepteurs de systèmes dédiés, qui utilisent pour la plupart des microprocesseurs en tranches et donc la microprogrammation qu'implique la réalisation de la fonction de commande ;

c) Chez de nombreux concepteurs de VLSI, qui apprécient la simplification au niveau du dessin, de la place occupée et de la standardisation qu'apporte la microprogrammation.

Quant à l'avenir, il semble que la microprogrammation doive jouer un rôle important dans ce qui est en passe de devenir la norme en matière d'architecture : les machines ou les systèmes multi-processeurs.

Déjà la plupart des moyens ou des gros ordinateurs sont conçus de manière à harmoniser le mieux possible les différents processeurs spécialisés qui forment leur structure interne (Unité Centrale, Processeur Mémoire, Contrôleurs d'E/S,...) ; c'est la cas par exemple du DPS 7/70 dont les différents processeurs sont microprogrammés séparément à l'aide du même système de développement (produit ALPHA-SIGMA).*

Des études (voir § 3.2.2.2) ont montré l'adéquation de réseaux de processeurs microprogrammés à répondre aux besoins de grande capacité de calcul traditionnellement assurés par les "super-calculateurs" alors que certains auteurs pensent même que le support microprogrammé des fonctions du système d'exploitation peut être démontré de façon optimale dans les architectures distribuées (voir § 3.3.1.2).

Cette orientation vers des systèmes "multi-microprogrammés" devrait nécessiter à terme la conception de mécanismes et de primitives permettant la communication-synchronisation entre tâches microprogrammées ; ces techniques pourraient s'inspirer de celles existant déjà pour les systèmes d'exploitation ou apparaissant dans la plupart des langages temps-réel mais devraient également bénéficier des caractéristiques originales de la microprogrammation, notamment son aspect très dépendant machine (certains projets vont dans ce sens, ainsi la machine multi-processeurs MAP construite à partir de 8 X 300 pour laquelle un noyau constitué de primitives destinées à la communication-synchronisation de microprogrammes est actuellement en cours d'élaboration [Cor 83]).

Avant de poursuivre, il nous semble important de repréciser que la microprogrammation n'est pas la panacée universelle en matière d'implémentation matérielle dans la mesure où son utilisation ne parvient pas toujours à éviter deux écueils importants :

a) La microprogrammation recouvre un ensemble considérable de techniques s'en réclamant, qui peuvent être très différentes les unes des autres (format des microinstructions, taille et temps d'accès des mémoires de commande, techniques de séquencement particulières,

*Dans le cadre de ce système de génération-simulation-implantation du microcode de type horizontal, les ingénieurs du groupe Bull qui en sont responsables ont même l'intention de procéder à la simulation simultanée de l'ensemble des microprogrammes implantés sur les différents processeurs.

mécanismes d'exécution des microprogrammes,...) ; il appartient au concepteur de l'architecture matérielle de bien choisir le type de microprogrammation qui correspond aux fonctions attribuées à la machine sous peine d'obtenir des performances peu conformes à ses prévisions ;

b) Même si le "moule" microprogrammé choisi semble satisfaire les exigences du niveau logique (langage machine, fonctions spécifiques système, mise au point,...), il convient d'exploiter au mieux les possibilités offertes par la microprogrammation adoptée en traduisant de manière optimisée les fonctions logiques choisies dans leur équivalent matériel (exploitation optimale du parallèlisme à l'exécution, emploi judicieux des sous-microprogrammes,...) une utilisation mauvaise ou maladroite de la microprogrammation peut ainsi avoir de facheuses conséquences sur les performances d'une machine.

Il importe donc, en premier lieu, de définir une micro-machine adaptée répondant bien aux spécifications du niveau logique qu'elle a à supporter.

Dans la grande majorité des cas, la conception de la **microarchitecture** se fait souvent avant la définition de la microprogrammation proprement dite, qui a pour tâche d'implémenter la **macroarchitecture** ; bien qu'ensuite, la modification de certaines ressources matérielles et l'écriture des microprogrammes se font généralement en parallèle mais de manière interactive*, le format des microinstructions est souvent définitivement figé par la structure de micromachines préalablement établie.

Ce problème n'est évidemment ni nouveau ni, apparemment, facile à résoudre.

Une solution, séduisante en théorie, pourrait consister en l'emploi combiné et judicieux d'une méthodologie de conception descendante, qui irait du niveau logique au niveau matériel en suivant des règles simples mais efficaces, et d'un ensemble de langages de description cohérent qui

*Ce dernier point nécessite la résolution de deux problèmes majeurs de communication : une modélisation et un environnement consistant communs aux équipes de "hard" et à celles de "firm", ainsi qu'une bonne entente entre services !

faciliterait le passage d'un niveau à un autre ; de tels outils n'existent pas encore même si de nombreuses études ont été ou sont réalisées dans ce but (ce sont le plus souvent des aides à la conception comme le système CASCADE [Mer 83] ou les outils spécifiquement orientés vers la C.A.O.-VLSI du type LUCIE [Guy 83] ou EMILIE [Nog 83]).

A notre connaissance, il n'existe dans la littérature qu'une seule véritable tentative de remédier à ce problème de l'adéquation de la micro-architecture à une macro-architecture logique donnée : il s'agit de la solution proposée par LI [Li 82], qui consiste en un système ISLV (Intelligent-Software-Like-VLSI) permettant, à partir d'une macro-architecture (éventuellement multi-processeurs) définie dans un langage évolué, de transformer simultanément ces spécifications de macro-architecture en une micro-architecture et en microprogrammes correspondants.

En ce qui concerne le rôle de la microprogrammation dans la conception de MLI (voir chapitre 4), la proposition d'implémentation matérielle esquissée à ce jour conduit à le définir ainsi : la microprogrammation doit être efficace en tant que support de la fonction commande du Processeur ADA et facile à utiliser en tant que support des fonctionnalités logiques évoluées (instructions complexes, commandes système,...) qui sont confiées à l'Unité Microprogrammée.

La recherche de l'efficacité conduit naturellement à une solution de type horizontal sur laquelle pourront être utilisées des techniques de compaction (locales ou globales) afin d'utiliser au maximum le parallèlisme potentiel d'exécution ; le format horizontal n'est pas, d'autre part, un obstacle à la facilité d'utilisation comme le prouvent les nombreux systèmes automatiques de génération de microcode (parfois même optimisé) existant ou en voie de réalisation (voir § 3.2.3).

Si la proposition d'architecture matérielle présentée au § 4.3 est maintenue, l'Unité de Commande du PA disposera d'une mémoire de commande de type RAM organisée de la façon suivante :

- Une zone réservée aux microprogrammes nécessaires à l'exécution des instructions les plus courantes et/ou les plus simples ; cette zone devra être protégée de toute écriture ;

- Une zone rechargeable contenant les microprogrammes nécessaires à l'exécution d'un instruction complexe reconnue préalablement par le Décodeur d'Instructions.

Cette zone de mémoire RAM est nécessaire dans la mesure où l'utilisation de la RAM de l'Unité Microprogrammée, située en dehors du PA (voir § 4.3), présenterait de réels problèmes de performance dûs à la communication par bus.

Des problèmes restent évidemment à résoudre, nécessitant des études plus approfondies et concernant :

- La stratégie de branchement des sous-microprogrammes ; les microprogrammes "extérieurs" pourront-ils utiliser des séquences de microinstructions de la zone réservée ?

- Le rôle de l'Unité Microprogrammée ; devra-t-elle, par exemple, "conditionner" les microprogrammes qu'elle doit acheminer vers l'Unité de Commande de manière à éviter l'appel de sous-microprogrammes restés en RAM "extérieure" ?

- La techique de chargement de la RAM du PA; sera-t-il indispensable d'établir un système d'anticipation du chargement de la RAM du PA en fonction d'un décodage d'instructions qu'il faudrait alors intégrer dans une réalisation en pipe-line ?

Il est difficile sans une première phase de simulation, d'avancer plus avant dans la description d'une microprogrammation de ce type ; les contraintes habituelles (temps, argent, compétences,...) à la réalisation de tout projet original ont souvent une influence déterminante sur les solutions adoptées, sans ignorer bien sûr les choix technologiques qui rendent toute implémentation matérielle dépendante des catalogues de composants et des règles d'utilisation qui les accompagnent.

Si l'usage (judicieux) de la microprogrammation semble s'imposer dans la réalisation de MLI comme dans beaucoup d'autres projets d'architecture, il existe une règle d'or que nous avons emprunté à RAUSCHER [Rad 80] et qui nous apparaît comme étant une conclusion en forme de mise en garde s'insérant parfaitement dans un tableau de la microprogrammation apparaissant peut-être par trop idyllique à l'issue de ce 3ème chapitre :

"La microprogrammation d'un algorithme inefficace ne le rend pas efficace !"

CHAPITRE 4

PRESENTATION DE MLI

4. PRESENTATION DE MLI

Le projet MLI (Machine Langage Intermédiaire) a été effectué dans le cadre des marchés 80-254 et 81-611 financés par l'Agence pour le Développement de l'Informatique.

A l'origine des travaux, se trouvent Claude RENVOISE, Chef du Service Transformation de Programmes du Centre de Recherche de CII-Honeywell Bull et Pierre DOUSPIS ; les personnes qui participèrent ensuite à l'ensemble des recherches furent Marc LEGOUX, Agnès BRADIER et Jacques BRYGIER.

Bien qu'éventuellement, l'architecture peut être destinée de manière générale à tous langages évolués à structure de blocs, le langage ADA a été choisi comme langage cible privilégié, notamment pour la définition du langage machine (appelé aussi intermédiaire), ce qui explique l'autre dénomination du projet qui porte sur les rapports le nom de machine langage ADA.

L'ensemble des études relatives à ce projet se trouve dans [RDo 81a], [RDo 81b], [RDL 82a] et [RDL 82b].

4.1. LES MOTIVATIONS

L'objectif du projet était de valider les choix non classiques concernant l'adressage et le langage machine comme solutions à l'amélioration d'une part, de l'efficacité du support des langages de programmation évolués, et d'autre part, de la sûreté de fonctionnement (§ 4.1.1.).

Le choix du langage ADA comme langage cible privilégié nécessite l'exposé des aspects particuliers qui font de lui un outil particulièrement bien adapté à la réalisation des objectifs définis précédemment (§ 4.1.2).

4.1.1. Efficacité et sûreté de fonctionnement

Les motivations principales qui ont présidé à la conception de MLI sont issues des constats suivants :

- L'implémentation de langages de programmation de plus en plus évolués sur des machines demeurés à un niveau de compréhension (code machine) relativement bas, a provoqué l'établissement d'un fossé sémantique (voir § 1.3) ;

- La protection et,de manière plus générale, la fiabilité des systèmes informatiques sont devenues une nécessité vitale qui devient de plus en plus critique à mesure qu'augmente la complexité de tels systèmes.

Afin de résoudre les problèmes posés par le fossé sémantique, le schéma d'exécution des programmes a été choisi du type b (voir § 2.1.1), c'est-à-dire qu'il a nécessité la création d'un langage machine de haut niveau, en l'occurrence particulièrement adapté à ADA.

Le langage machine intermédiaire n'est pas seulement un facteur d'efficacité dans le support des langages à structure de blocs mais offre également certaines caractéristiques intéressantes en matière de sûreté de fonctionnement : contrôles à l'exécution, simplification de la génération de code à la compilation, facteur d'amélioration de la mise au point des programmes...

La sûreté de fonctionnement, qui est la deuxième motivation importante, est assurée principalement par les options architecturales suivantes :

- Une segmentation à deux niveaux (Segments Virtuels et Segments Logiques) de la mémoire virtuelle qui allie la protection d'objets de taille relativement petite à un coût de fonctionnement raisonnable (voir § 4.2.1.1) ;

- Un adressage lexical qui, outre qu'il reflète parfaitement la structure de blocs des langages cibles (ADA, PASCAL,...), utilise la souplesse de la segmentation à deux niveaux pour implémenter le concept de "petits" domaines de protection (§ 4.2.1.2) ;

- L'intégration dans le matériel de mécanismes évolués comme un ramasse-miettes mais surtout d'outils et de ressources d'aide à la mise au point des programmes.

Les différentes options adoptées pour MLI qui viennent d'être énumérées ont également une influence sur d'autres aspects appartenant aux nouveaux cahiers des charges des ordinateurs (§ 1.4), notamment la facilité d'utilisation (programmation en langage évolué, outils de mise au point intégrés,...) et le gain attendu en performances (principalement au niveau logique par la compaction du code).

Plutôt que de préciser dès maintenant l'influence de chacun des aspects de l'architecture logique ou matérielle sur les objectifs fixés pour MLI, les améliorations de différentes natures apportées par chacun de ces aspects seront évoqués lors de leurs présentations respectives (§ 4.2).

Le choix du langage ADA s'inscrit dans le cadre de la recherche de la sûreté de fonctionnement mais possède également d'autres justifications ; l'ensemble des motivations ainsi que l'ensemble des caractéristiques qui en sont l'origine sont développés dans le paragraphe suivant.

4.1.2. Choix du langage ADA

Les raisons qui ont conduit au choix du langage ADA comme langage cible de l'architecture et surtout du langage machine évolué sont diverses :

- Le langage ADA, qui est un produit issu des laboratoires de recherche de CII-Honeywell Bull, financé par le "Department of Defense" américain, est amené à connaître un développement important comme en témoigne le nombre de projets d'implantation à travers le monde (compilateurs CII-Honeywell-Bull/Alsys/Siemens, Western Digital, implantation sur INTEL iAPX 432, environnement de programmation sur MV/4000 à MV/100000 développés par Data General et Rolm* , etc.)

- Les spécifications de ADA en font un langage algorithmique à vocation universelle (scientifique, gestion, temps réel,...) qui justifient d'autant plus son choix comme langage cible que le schéma d'exécution adopté (langage machine intermédiaire évolué) peut, suivant l'attitude prise pour son implémentation (voir § 2.1.1.), limiter l'efficacité à supporter plusieurs langages de programmation très différents les uns des autres ;

- La plupart des caractéristiques du langage ADA s'attachent à créer ou à renforcer la protection des objets et la validation des opérations ; ces aspects du langage, qui sont détaillés dans les paragraphes suivants, demandent en revanche la mise en place de mécanismes parfois complexes que la prise en compte par le matériel ou l'architecture permet de rendre plus fiables et plus performants.

La suite de ce paragraphe rapporte le rôle tenu par différents aspects du langage ADA dans l'établissement de la protection qui accompagne toute construction de programme.

* Rolm a été le premier industriel à construire un compilateur ayant réussi avec succès l'ensemble des tests de validation conçus par Softech ; le tout premier compilateur, validé au début 83, est en effet dû à l'Université de New-York.

Le but de cette étude n'étant pas la description de ADA, les concepts du langage évoqués plus loin sont supposés connus (cf [USA 83]) et ne seront brièvement détaillés que dans la mesure où la justification de leur rôle dans la contribution à la fiabilité des programmes ne sera pas évidente.

4.1.2.1. Eléments du langage contribuant à la fiabilité du logiciel

Outre la recherche de l'universalité, la fiabilité de l'exécution des programmes fut le principal objectif sur lequel se portèrent les efforts des concepteurs de ADA.

Le résultat de ces efforts apparaît au travers d'aspects différents mais caractéristiques du langage dont les principaux sont présentés ci-dessous.

a) Méthodologie de conception des programmes

Il est maintenant généralement admis qu'une programmation structurée et modulaire donne, tout au moins en ce qui concerne les langages de type procédural, les meilleurs gages de fiabilité ; la **structure de blocs**, alliée aux règles de visibilité qui l'accompagnent, les notions de **sous-programmes** et de "**package**", renforcées par celles d'**unités de bibliothèque** et de **sous-unité** sont les principaux éléments qui fournissent au programmeur les capacités de conception descendante, de hiérarchisation et de modularité de ses programmes (qu'il retrouve dans un contexte temps réel, grâce à la notion de "**task**") ; afin de permettre à la programmation d'épouser plus fidèlement les structures propres à chaque application, le langage offre également la possibilité d'implémenter des **types abstraits** (grâce aux "package") et de créer des structures de données relativement souples (types énumérations, discriminant dans les types record,...) ; quelques facilités diverses, comme par exemple le passage des paramètres par position, sont également offertes pour pallier (modestement) la complexité qu'engendre par ailleurs la richesse et les règles de protection du langage ;

b) Aide à la mise au point des programmes

Les avantages de la méthodologie de conception sont complétés pour la phase de mise au point des programmes par ceux de la **compilation séparée** qui s'avère quasiment indispensable pour les programmes de grandes tailles ; le typage des données (évoqué plus loin) et le concept d'**exception** peuvent également être des aides importantes à la mise au point en facilitant le repérage et l'identification des erreurs ;

c) Aide à la sûreté de nouveaux programmes

Le concept d'unités de programmes **génériques** permet d'assurer la mise au point d'un programme dont la structure, ainsi validée, peut servir à la création d'un nouveau programme sur lequel la plupart des vérifications n'ont pas à être effectuées ;

d) Protection des objets et sélectivité des accès

Un premier élément fondamental contribuant à l'intégrité des données est constitué par les concepts de **types** et de **sous-types** qui imposent des contrôles de validité lors des affectations ou autres opérations appliquées aux objets typés (dans cette optique, les **pointeurs** sont également typés) ; les accès aux variables ou paramètres du programme sont d'une manière générale soumis aux règles de visibilité s'appliquant aux sous-programmes comme aux tâches, mais surtout peuvent être protégés de façon plus restrictive grâce à l'"encapsulage" fourni par les packages que peuvent renforcer les notions de **types privés** et **privé limités** ; enfin, les règles de visibilité associées à l'emploi de types **dérivés** permet la sélectivité des opérations sur un type de base que partagent des unités de programme différentes ;

e) Mécanismes étudiés dans le support du temps réel

Un effort particulier a été accompli sur la cohérence et la fiabilité de ces mécanismes avec l'encapsulage des tâches, l'adoption d'une forme unique de synchronisation-communication (le **rendezvous**) et

certaines possibilités de prise en compte de phénomènes classiques en temps réel comme la mise en place de "time-out" ou la réalisation de l'exclusion mutuelle grâce aux **"accept"** ;

f) Traitement des erreurs

La capacité donnée aux programmes d'introduire des contrôles supplémentaires aux endroits critiques grâce aux **exceptions** est évidemment un gage supplémentaire de fiabilité des programmes car elle permet notamment de détecter l'erreur, d'effectuer un traitement immédiat dans le cas d'une signalisation locale, de rétablir ainsi localement la cohérence ou sinon, de propager les erreurs jusqu'à un niveau d'imbrication accessible aux procédures de recouvrement des erreurs qui sont à la charge du programmeur.

Malgré les apports considérables du langage ADA dans le domaine de la fiabilité des programmes, certains problèmes inhérents à la conception du logiciel demeurent cependant imparfaits ou non résolus.

4.1.2.2. Limites du langage face aux problèmes de protection

Le principal problème de protection est celui de la **méfiance mutuelle** entre sous-systèmes ; un sous-système est constitué d'un ensemble de procédures agissant sur des objets propres et correspond dans ADA aux unités de bibliothèque ou aux sous-unités.

L'**isolation réciproque**, qui est le premier point à réaliser, comporte deux aspects :

- D'une part, toute procédure doit s'exécuter dans un environnement où seuls les accès nécessaires à cette exécution sont autorisés (principe du moindre privilège) ; ceci est réalisé par le langage ADA dans le cas des **unités de bibliothèques** et dans celui des **sous-unités** grâce aux règles d'utilisation qui en assurent la gestion ;

la protection est néanmoins renforcée dans MLI pour éviter la transgression par l'utilisateur de ces règles d'utilisation (voir plus loin, le problème engendré par la fonction **unchecked conversion**) ;

- D'autre part, l'accès à un sous-système ne peut se faire qu'en des points d'entrée qu'il a définis, et où il pourra effectuer des contrôles sur les paramètres ; les contrôles sur les paramètres sont stipulés dans ADA et le principe de l'**encapsulage** (en particulier sous la forme de package) permet de définir les seuls entités visibles de l'extérieur du sous-système.

Le partage d'objets par des sous-systèmes pose les problèmes de la **révocation** (et de l'amplification) des **droits** et de l'**étanchéité**.

La révocation est la possibilité pour un agent (sous-système ou partie de sous-système), qui a transmis tout ou partie de ses droits sur un objet à d'autres agents, de leur retirer sélectivement ces droits.

La révocation des droits peut être immédiate ou différée ; si, grâce à une édition de liens (pseudo) dynamique* bien étudiée, une forme de révocation différée peut être obtenue, la révocation des droits immédiate reste un problème non résolu par ADA.

L'amplification des droits peut par contre être assurée par l'emploi adapté de types dérivés qui permettent l'extension de certaines opérations sur un type donné.

En ce qui concerne l'étanchéité, qui est d'empêcher qu'un sous-système divulgue ou mémorise des informations qui lui ont été transmises en paramètres, le problème n'est pas non plus résolu par ADA.

* L'édition de liens dynamiques est une solution retenue dans le cadre de MLI ; elle présente deux aspects : un aspect langage qui consiste à vérifier que des propriétés de cohérence entre les constituants ADA sont satisfaites et les élaborer si nécessaire, et un aspect système qui consiste à construire des structures de données système qui matérialisent le programme exécutable résultant dans la machine.

Dans leur volonté d'assouplir l'utilisation du langage, en particulier pour les phases de mise au point des programmes, les concepteurs d' ADA autorisent certaines manipulations qui remettent en cause toute la protection établie par ailleurs.

Ainsi, grâce à l'attribut address qui permet d'obtenir l'adresse d'un objet, un utilisateur peut faire de l'arithmétique sur cette adresse et, à l'aide de la fonction générique unchecked conversion, affecter le résultat de ses calculs à un objet de type access ; il a alors accès à un objet qui peut lui être parfaitement interdit et a pris alors en défaut toute la protection mise en place par l'ensemble des autres concepts du langage.

La machine ADA doit donc supporter efficacement l'ensemble des spécifications du langage (au même titre qu'un compilateur classique) mais doit en plus, pour tenir ses objectifs de sûreté de fonctionnement, limiter la portée du problème évoqué ci-dessus.

A cet effet, MLI s'est particulièrement attaché à offrir un support fiable et efficace à l'isolation des sous-systèmes, la protection des données cachées, la protection des données de l'utilisateur, la conversion entre types et à la prise en compte de certains traits du langage comme les types access ; la réalisation de ces objectifs est obtenue par l'adoption de certains aspects architecturaux originaux qui sont développés maintenant.

4.2. LES CARACTERISTIQUES PRINCIPALES

Les caractéristiques principales sont les aspects de MLI qui, en dehors de toute considération d'implémentation matérielle, traduisent la volonté d'assurer la réalisation des objectifs définis initialement (cf § 4.1) ; il s'agit de l'organisation de la mémoire virtuelle, du mécanisme d'adressage et du langage machine adoptés.

4.2.1. L'organisation de la mémoire

La fonction mémoire associée à la machine peut être abordée sous deux aspects :

- D'une part, l'organisation de la mémoire virtuelle telle que la gère le système d'exploitation ;

- D'autre part, le découpage de cette mémoire tel que le demande le mode d'adressage retenu pour le langage machine.

Si ce dernier point est une composante principale et définitive de l'architecture de MLI, l'organisation de la mémoire virtuelle est une proposition d'implémentation qui reste soumise à d'éventuelles transformations.

C'est néanmoins l'ensemble des aspects que revêt la mémoire virtuelle suivant le niveau d'approche (utilisateur, système, langage machine) qui est maintenant décrit.

4.2.1.1. Description de la fonction mémoire

Le principe de base de la mémoire de MLI, motivé par la recherche d'une protection maximale, est celui d'une segmentation à deux niveaux.

La segmentation d'une mémoire est une bonne approche de la protection car elle permet de regrouper en les isolant certaines entités aux caractéristiques similaires sur lesquelles s'exercent des droits comparables.

Le problème engendré par le partitionnement de la mémoire en **segments virtuels** est avant tout le coût de création et de gestion qu'il entraîne auprès du système d'exploitation.

Or, la protection des objets est d'autant mieux assurée que les segments peuvent être petits : ils contribuent ainsi à la réalisation de petits domaines de protection dont les avantages ont déjà été évoqués (cf § 2.2.4).

Pour ne pas avoir à supporter le coût élevé qu'entraînerait la seule utilisation de Segments Virtuels dans la réalisation d'une protection sélective, un nouveau type de segment a dû être introduit ; il s'agit du **Segment Logique** qui est une zone continue d'espace mémoire contenue dans un Segment Virtuel.

Ainsi le Segment Logique est reconnu au niveau du langage machine, notamment au travers de l'adressage lexical (voir § 4.2.2), mais le coût de sa création ou de sa destruction reste faible puisqu'il ne met pas du tout en jeu les mécanismes complexes de gestion de la mémoire virtuelle.

La mémoire virtuelle adoptée pour MLI est segmentée et paginée (512 octets par page) ; elle est d'une capacité de 2^{40} octets (une adresse virtuelle s'exprime sur 40 bits).

Deux types de segments sont donc accessibles au travers de **descripteur** :

- Les Segments Virtuels (SV) créés et détruits par le gestionnaire de la Mémoire Virtuelle, qui est traditionnellement une fonctionnalité logicielle ;

- Les Segments Logiques (SL) créés et détruits par des instructions spécifiques de la machine.

a) **Aspect architecture logique**

Les descripteurs de segment (SV ou SL), définis sur 64 bits (fig. 17) sont des formes de capacités sur lesquelles reposent le mode d'adressage (lexical) de la machine et donc l'accès aux objets (voir le rôle du T.A.L., Chapitre 5).

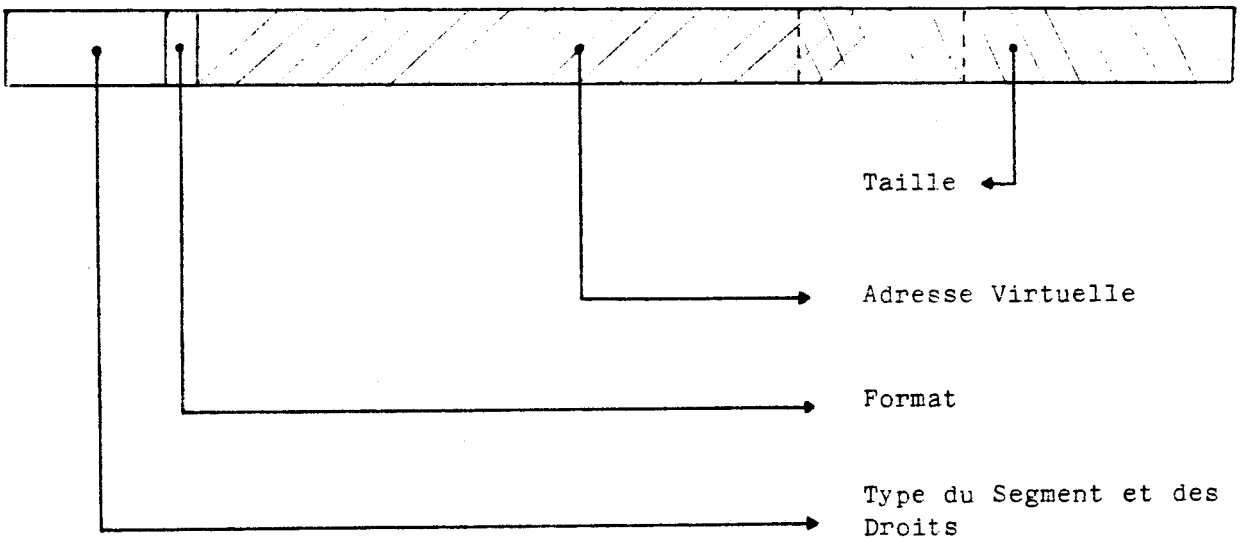


Figure n°17 : Descripteur de Segment

Les différents champs du descripteur ont les fonctions suivantes :

- Type du segment et droits : il indique la nature des informations que contient le segment (SLCL (cf § 4.2.2), données, tas,...) et les opérations autorisées sur le Segment ;

- Format : il indique la taille respective des deux derniers champs qui sont configurés de la manière suivante :

. Sans signification (descripteur nul) ;
. Adresse sur 40 bits et taille sur 14 bits (le segment est défini en octets) ;
. Adresse sur 37 bits et taille sur 17 bits (le segment est défini en double mots) ;
. Adresse sur 31 bits et taille sur 23 bits (le segment est défini en pages) ;

- Adresse virtuelle : il indique l'adresse d'implantation du Segment dans la mémoire virtuelle de 2^{40} octets ; selon la valeur du Format, l'adresse est exprimée en octets, en double mots ou en pages ;

- Taille : il indique la taille du segment ; cette taille est également exprimée sous trois unités différentes et peut prendre les valeurs maximales de 16 Koctets, 1 Moctets ou 4 Goctets suivant que l'unité est l'octet, le double mot ou la page.

Ces descripteurs sont utilisés pour décrire tous les types de segments manipulés au niveau logique (visibles du langage machine) par la machine ; ils se retrouvent, par exemple (cf § 4.2.2), dans les entrées de la TNL pour désigner les SL du Contexte Lexical, dans des catégories de SLCL pour désigner des SL de Données Locales, des SV de Code,...

Ces descripteurs sont manipulés par la machine, principalement pour accéder aux objets désignés par les instructions, grâce au T.A.L. (cf chapitre 5) qui les utilise pour fournir l'adresse virtuelle d'un objet désigné par son adresse lexicale.

Les 31 bits de poids forts de l'adresse virtuelle constituent le numéro de page virtuelle ; le numéro de page virtuelle est ensuite donné au topographe qui effectue la conversion Page Virtuelle/Page Réelle.

La transformation d'une adresse virtuelle en adresse réelle peut, à ce stade de l'étude, être envisagée comme s'apparentant à la solution adoptée par le System 38 d'IBM et qui utilise une table d'index* et une table des pages réelles** (voir figure n°18).

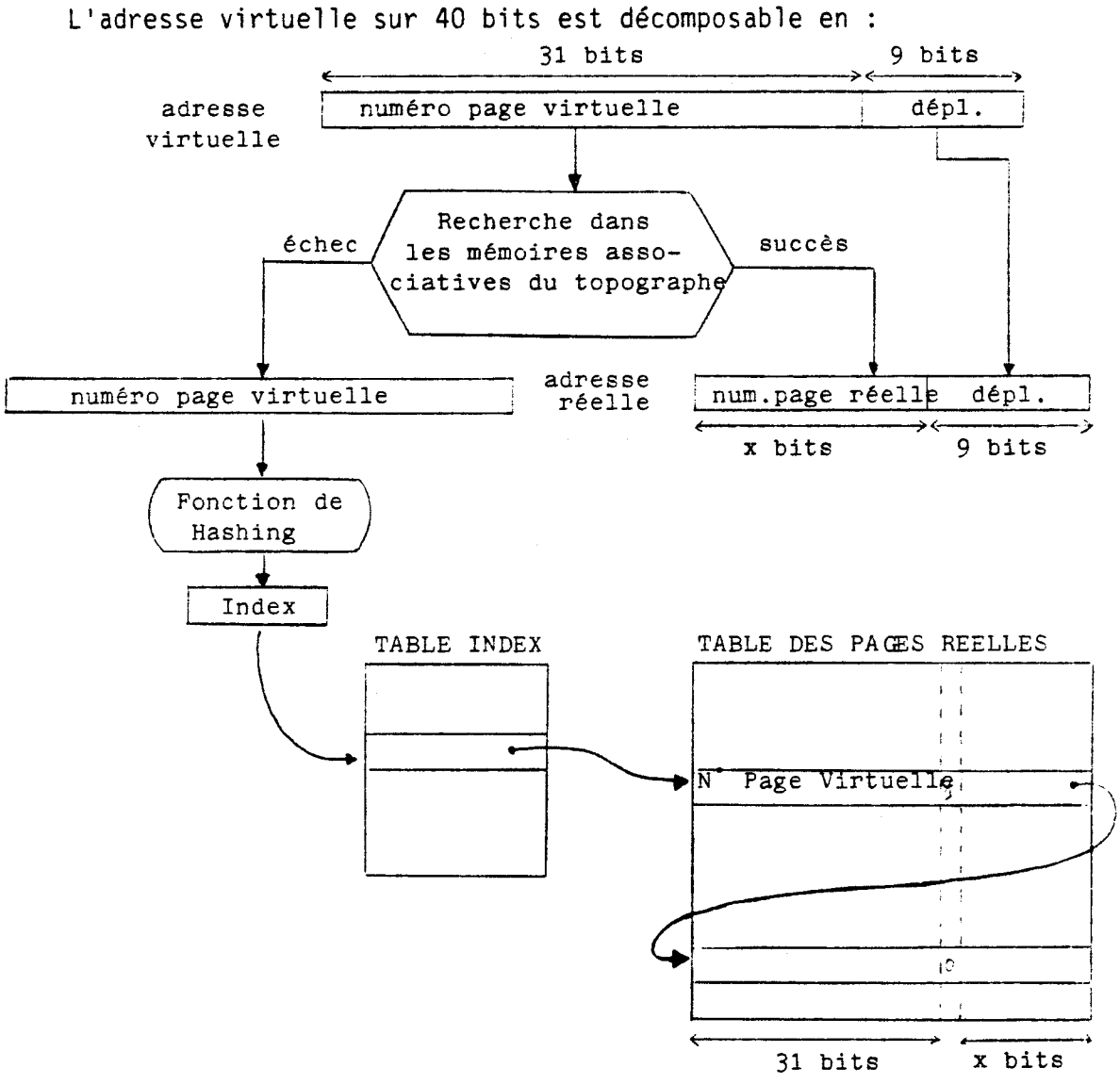


Figure n°18 : Décomposition de l'adresse virtuelle

* Le nombre d'entrées dans la table des index est à ajuster en fonction de la qualité de la fonction de HASCHING pour limiter le nombre de collisions.

**La taille de la table des pages réelles est égale au nombre de pages réelles qui pourra supporter la configuration physique de la machine.

b) **Aspect système**

La mémoire de MLI vue sous son aspect système ne fait pour l'instant l'objet que d'une proposition d'implantation, non définitivement entérinée et, en conséquence, sujette à de nombreuses transformations.

Cette proposition regroupe principalement les points suivants :

- La Mémoire Virtuelle de 2^{40} octets est découpée en Segments Virtuels de taille t1, t2,...tn en nombre N1, N2,...Nn (voir figure n°19) ;

- Ce découpage est effectué lors de l'élaboration du système par le responsable de l'installation en fonction de ses besoins particuliers, une configuration fournie par défaut étant toujours envisageable ;

- L'accès aux segments se fait par l'intermédiaire d'une Table des Groupes de Segments (TGS) puis d'un catalogue qui décrit l'association entre les SV du groupe et son support physique dans la mémoire d'archivage (figure n°20).

Chaque élément de la table TGS est composé de trois champs indiquant :

- La taille maximale d'un segment du groupe ti ;
- L'adresse de base d'implantation du groupe dans la Mémoire Virtuelle ;
- L'adresse du catalogue correspondant aux Ni Segments du groupe de taille maximale ti.

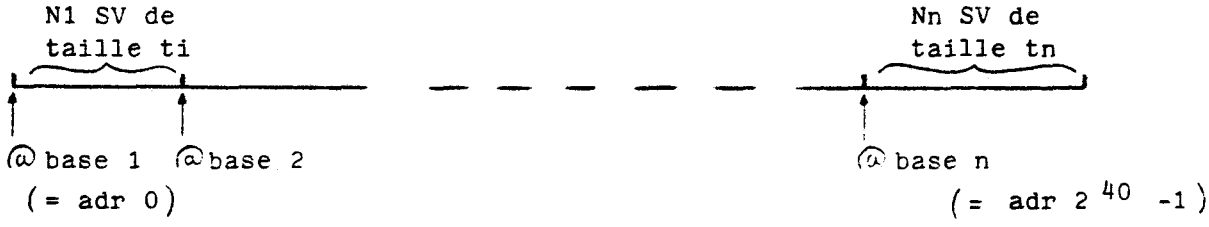


Figure n°19 : Représentation de la Mémoire Virtuelle

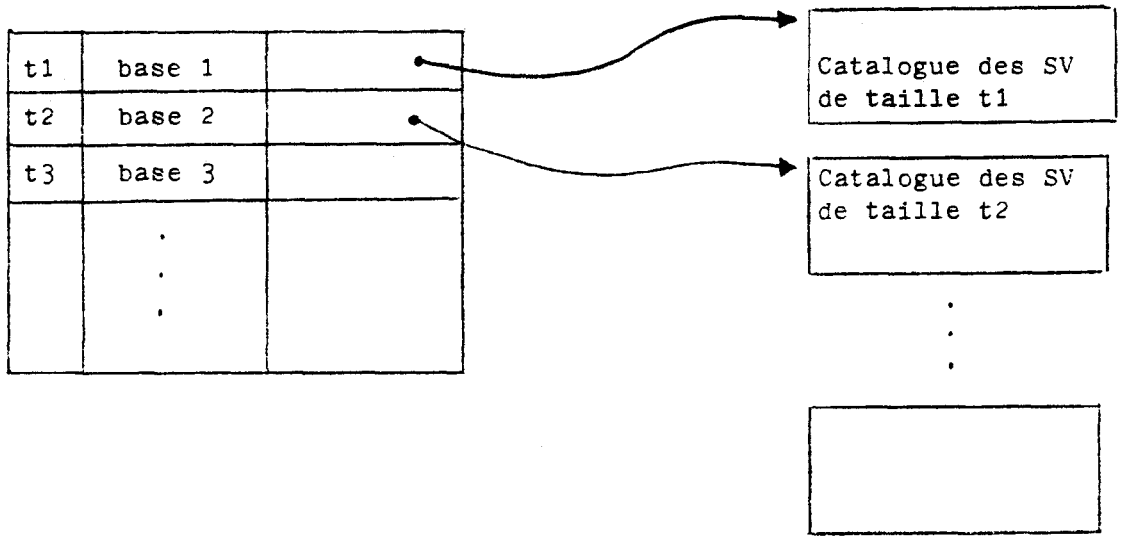


Figure n°20 : Principe d'accès aux segments

La table TGS et les catalogues associés permettent au gestionnaire de la Mémoire Virtuelle :

a) De résoudre rapidement les défauts de pages virtuelles ; il faut en effet être capable d'effectuer une correspondance entre l'adresse virtuelle en défaut et la position dans la Mémoire d'Archives pour amener en Mémoire Centrale le SV associé à cette adresse ou une ou plusieurs de ses pages virtuelles suivant que le mode de fonctionnement du système est en "segment ou page à la demande".

Ce problème est résolu de la manière suivante :
Lors d'un défaut de page constaté par le topographe, la table TGS est balayée pour retrouver le catalogue du groupe de SV comprenant la page virtuelle en défaut ; ce balayage peut faire l'objet de diverses optimisations : il peut être effectué par une suite d'accès dichotomiques si les entrées de TGS sont attribuées par adresses croissantes par exemple ; disposant du catalogue, l'entrée où est décrit le SV contenant cette page virtuelle est retrouvée grâce au calcul suivant :

(adresse virtuelle -@base i) / ti

Cette expression arithmétique reflète directement l'implantation fictive consécutive des SV du groupe ; il est alors possible d'accéder aux informations contenues dans l'entrée du catalogue qui permettent en particulier d'effectuer des transferts de la Mémoire d'Archives vers la Mémoire Centrale.

b) Une gestion performante de la Mémoire Virtuelle ;

L'équation introduite ci-dessus établit une correspondance biunivoque entre la zone d'espace virtuelle occupée par les SV du groupe d'une taille ti et les entrées du catalogue correspondant.

La gestion de la zone de Mémoire Virtuelle associée à tout groupe se ramène dès lors à la gestion des entrées du catalogue correspondant ; les entrées libres du catalogue peuvent, par exemple, être chaînées dans une liste libre.

L'introduction d'un SV d'une taille t dans la Mémoire Virtuelle consiste successivement à :

- Rechercher le groupe de segments dont la taille est la plus proche de taille désirée ; une consultation de la table TGS est nécessaire pour cela ;

- Consulter la liste libre du catalogue ainsi déterminé pour obtenir une entrée libre e qui est alors retirée de la liste ;

- Calculer l'adresse virtuelle d'implantation du segment qui est alors obtenue par le calcul suivant :

adresse virtuelle =@base i + e * ti

(C'est cette adresse qui figurera dans le descripteur du SV).

Le retrait d'un SV de la Mémoire Virtuelle s'obtient d'une manière symétrique ; à partir de son adresse virtuelle d'implantation dans la Mémoire Virtuelle il est facile de retrouver l'entrée du catalogue où il est décrit ; cette entrée est alors ajoutée à la liste des entrées libres du catalogue correspondant.

Note sur la TGS :

Cette table n'est consultée qu'exceptionnellement. Cependant, pour obtenir un compromis espace/temps satisfaisant, le nombre d'entrées doit être limité à 4, 8, 16 voire 32. Sa consultation peut être optimisée par des algorithmes adaptés tel que la dichotomie.

4.2.1.2. Conséquences sur le fonctionnement

Outre le choix d'une segmentation à deux niveaux (SV et SL) destiné, avec l'adressage lexical, à permettre la mise en place de petits domaines de protection, la mémoire de MLI doit posséder la structure et les mécanismes d'implémentation qui la rendront :

- Performante, compte tenu de l'objectif temps réel du langage ;

- Facile à reconfigurer sous sa forme virtuelle, sans contrainte particulière de la part du matériel ;

- Apte à assurer un schéma d'adressage optimum dans une configuration sans Mémoire Virtuelle.

La solution proposée pour la gestion des SV par le système d'exploitation a les avantages suivants :

- Une grande souplesse pour le découpage de la Mémoire Virtuelle puisqu'aux limites, il est possible d'avoir une configuration de 2^{40} segments de 1 octet ou une configuration de 256 segments de 4 G octets ;

- Le nombre de groupes de segments qu'il est possible de faire cohabiter n'est pas fixé par l'architecture matérielle mais dépend de la taille de la TGS choisie ;

- Le découpage de la Mémoire Virtuelle est laissé au responsable de l'installation qui peut faire une répartition selon des statistiques de fonctionnement de son système, surtout dans le domaine temps réel ;

- La gestion de la Mémoire Virtuelle est simple puisque l'allocation et la désallocation d'un SV dans la Mémoire Virtuelle consiste à prendre ou à libérer un SV du groupe de segments de la taille voulue ;

- Lors d'un défaut de page, que ce soit en segment ou en page à la demande, le passage de l'adresse virtuelle à la description en Mémoire d'Archives est rapide grâce à la TGS et aux catalogues associés (important dans un environnement temps réel) ;

- L'extraction ou l'implantation d'un fichier (ou Objet Virtuel) dans le système est simple puisqu'il y a distinction entre la Mémoire Virtuelle et la Mémoire d'Archives (contrairement à l'approche IBM 38 qui possède un niveau (logique) unique de mémoire).

Cette solution a néanmoins des contraintes qu'il est bon d'exposer :

- Les tailles de SV ne sont pas totalement libres, il en résulte une perte d'espace virtuel potentiel lors de la création des SV si les tailles ti ne sont pas bien ajustées aux besoins ;

- Un groupe de segments d'une taille donnée peut se trouver saturé, obligeant ainsi à créer le SV dans un groupe de segments d'une taille supérieure et donc entraînant une perte de l'espace fictif libre ;

- La distinction entre Mémoire Virtuelle et Mémoire d'Archives pose les problèmes classiques du nommage multiple et de gestion des fichiers.*

Des optimisations atténuant ces effets négatifs, notamment en ce qui concerne la taille des segments et la génération des adresses virtuelles, sont néanmoins possibles et exposées dans [RDL 82b].

* L'orientation "Mémoire Virtuelle différente de Mémoire d'Archives" apparaît encore jusqu'à présent nécessaire car elle permet de diminuer la taille des descripteurs de segment (noms de segment plus petits) et autorise une Mémoire d'Archives de taille quasi-infinie.

L'approche proposée a également une influence sur l'architecture matérielle et possède les avantages suivants :

- L'adresse virtuelle est limitée sur 40 bits (au lieu des 64 bits de l'approche IBM 38) ce qui permet une augmentation de performance due à la taille réduite des descripteurs, une intégration plus simple en VLSI par exemple car les registres sont moins larges, une diminution de la taille du BUS d'adresses ;

- La taille du SV n'est pas imposée par le matériel ;

- Les contrôles prévus par les descripteurs peuvent être mis en oeuvre (taille maximale du SV ou du SL, droits).

Il existe par contre une certaine perte de précision relative à la taille des segments due au fait que celle-ci doit être exprimée en double mots si elle est supérieure à 2^{14} octets ou en pages si elle est supérieure à 2^{20} octets ; la perte d'espace qui en résulte devrait toutefois rester négligeable.

4.2.2. L'adressage lexical

Le mode d'adressage de MLI est basé sur l'imbrication des blocs qu'implique le langage ; cette solution, qui n'est pas nouvelle (BURROUGHS B 6700, PASC-HLL, P-Code,..) , se distingue des implémentations habituelles par l'utilisation de SL qui lui permet d'affiner la protection.

4.2.2.1. Description

Dans les langages à structure de blocs, le niveau lexical correspond au niveau d'imbrication des sous-programmes depuis le niveau hiérarchique le plus haut (notion de programme principal) jusqu'au niveau le plus interne.

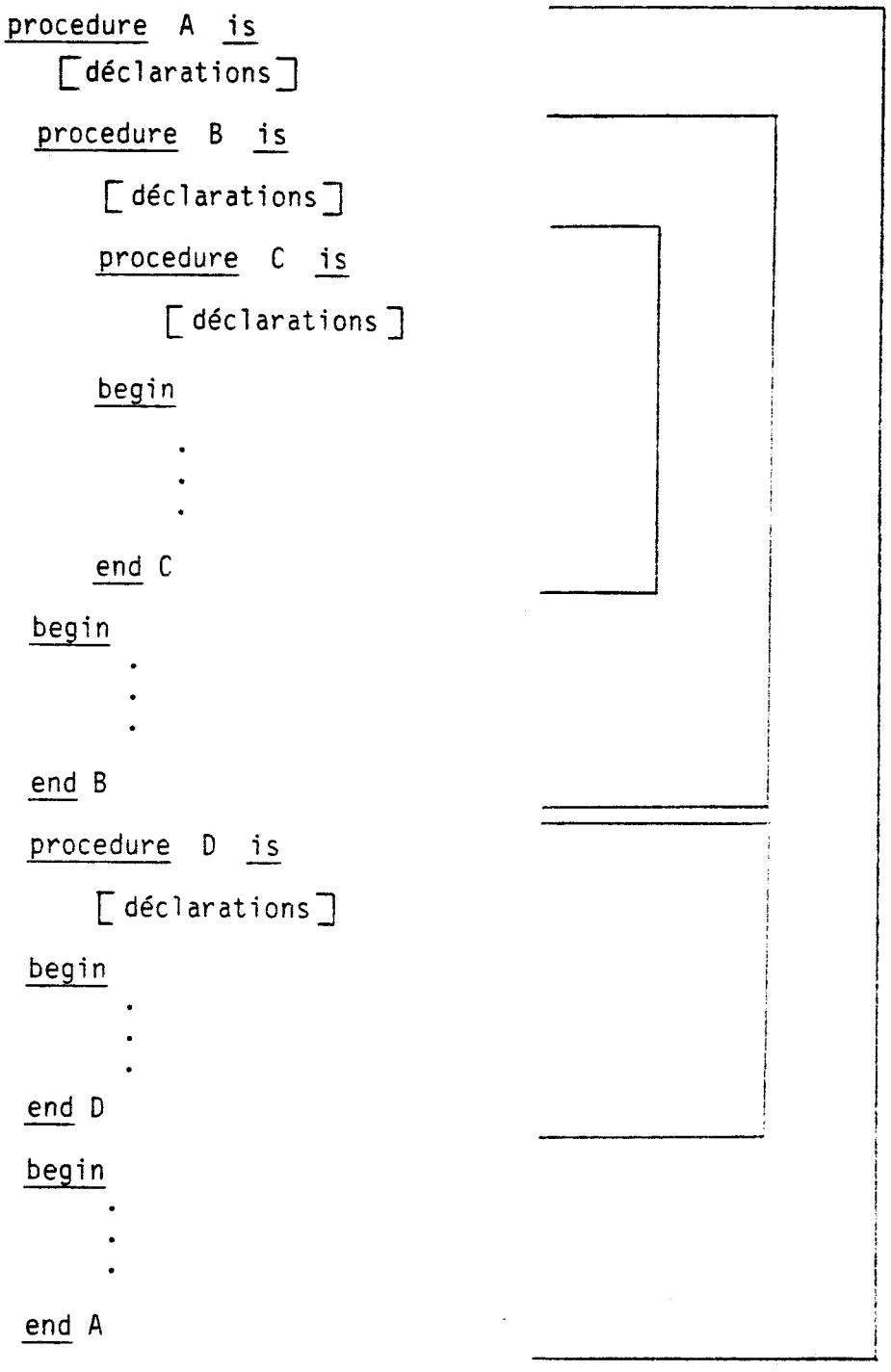


Figure n°21 : Illustration de la structure de bloc

Ainsi, dans l'exemple de la figure n°21, la procédure A étant supposée être au niveau lexical 1, les procédures B, C, D sont respectivement aux niveaux lexicaux 2, 3, 2.

Les règles de visibilité des langages à structure de blocs font qu'une unité de programme peut accéder aux objets (variables, procédures,...) déclarés dans les unités englobantes et uniquement dans celles-ci ; ainsi B "voit" les objets de A mais ne "voit" ni les objets de D, ni les objets de C.

La méthode classique d'implémentation de l'accessibilité aux objets à l'exécution utilise une pile d'activation ; la solution adoptée dans MLI consiste à matérialiser cette pile d'activation au travers d'une table en mémoire appelée **Table des Niveaux Lexicaux** (TNL) qui est incluse dans une structure plus riche appelée **Bloc de Contrôle de Processus** (BCP) associée à chaque processus (le rôle et la description des BCP sont exposés dans [RDo 81b]).

Chaque entrée de la table TNL contient un descripteur de Segment Logique (voir § 4.2.1.1) d'un type particulier, appelé Segment Logique de Contexte Lexical (SLCL).

Un SLCL est associé à toute partie de programme correspondant à un bloc défini par le langage (sous-programmes, package,...) et regroupe deux ensembles d'informations :

1. Des descripteurs de segments logiques (appelés aussi catégories) regroupant les objets du bloc par affinités : SL de données locales, SL des paramètres, SL des constantes,...

2. Des informations de gestion du contexte telles que :

   - L'adresse de retour ;
   - Le niveau lexical du bloc appelant ;

- Le compteur de tâches non terminées ;
- La capacité sur l'activation SLCL précédente de même niveau lexical (voir plus loin) ;
- L'indication du nombre de capacités de Segments Logiques dans le SLCL.

Les informations de gestion de contexte doivent bénéficier d'une protection sans faille : celle-ci est assurée par le fait que l'accès à un SLCL ne peut se faire que par l'intermédiaire d'un descripteur incluant des droits d'accès (comme une capacité*) qui n'autorisent que quelques instructions spécifiques (appel de sous-programme, sortie de sous-programme,...) à accéder au contenu du SLCL.

L'accès à un opérande manipulé par le langage machine se fait grâce à une **adresse lexicale**, se présentant sous la forme d'un triplet (i, c, d) dont voici la signification :

- i représente le niveau lexical et désigne un descripteur de SLCL (entrée de la TNL) ;
- c représente une catégorie d'objets et désigne un descripteur de SL (entrée d'un SLCL) ;
- d représente le déplacement dans le SL et désigne l'objet à accéder.

La figure n° 22 illustre le principe de l'adressage lexical par un schéma simplifié.

* Il est possible d'assimiler les SLCL et la TNL à des structures de regroupement de capacités notamment pour appréhender leur protection.

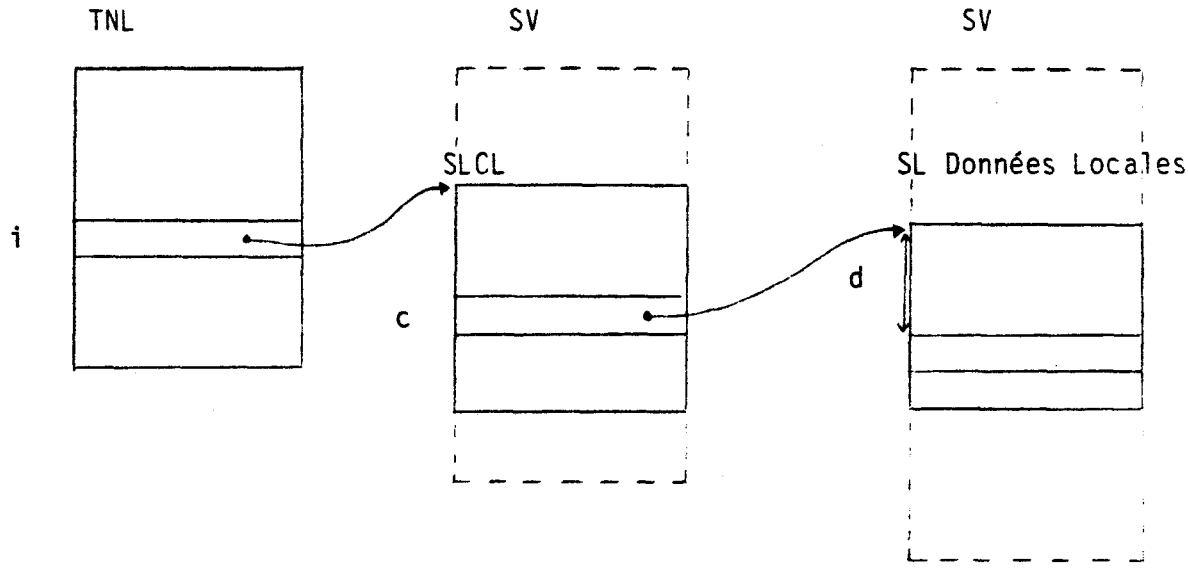


Figure n°22 : Principe de l'adressage de type lexical adopté dans MLI

La gestion de la TNL ainsi que les stratégies de regroupement des Segments Logiques (SLCL, SL de données,...) ne seront pas développés dans ce rapport mais peuvent être retrouvés dans [RD 81a].

Remarque : L'adresse lexicale d'un objet peut se présenter dans le langage machine sous deux formes différentes :

1. Adresse globale : elle est de la forme normale i, c, d ;
2. Adresse locale : elle ne comporte que les composantes c, d, la valeur de i étant le niveau lexical courant (qu'il faudra vraisemblablement mémoriser dans une ressource matérielle spécifique).

4.2.2.2. Conséquences sur le fonctionnement

Le choix de cet adressage de type lexical vise à réaliser en priorité des objectifs de sûreté de fonctionnement grâce à la réalisation des points suivants :

1. Prise en compte des règles de visibilité : il s'agit de l'avantage fondamental et classique de l'adressage lexical qui permet un premier contrôle de protection en comparant le niveau lexical i de l'adresse de l'objet à accéder avec le niveau lexical courant ic ; si i est supérieur à ic, la machine détecte une erreur et interdit l'accès à l'objet ; de plus, les mécanismes de gestion adoptés pour la TNL (une description détaillée est fournie dans [RDo 81a]) permettent d'assurer l'étanchéité des unités de bibliothèques ;

2. Réalisation de petits domaines de protection : l'existence de Segments Logiques qu'il n'est pas coûteux de prendre relativement courts permet de créer des domaines de taille suffisamment réduite pour pouvoir regrouper uniquement des objets soumis aux mêmes droits d'accès, tandis que l'existence d'un SLCL assure, grâce aux catégories, la protection de ces domaines que sont les SL de données locales, de paramètres, etc. ;

3. Simplicité de mise en oeuvre : les mécanismes assurant la gestion des niveaux lexicaux (voir [RDo 81a]), par l'intermédiaire de la TNL et des SLCL, évitent les problèmes de gestion de pile rencontrés dans les autres implémentations de l'adressage lexical (cf Burroughs B 6700).

L'adressage lexical pose évidemment le problème des performances, étant donné la suite d'opérations nécessaires à l'accès effectif à l'objet désigné par une adresse lexicale i, c, d ; ces opérations sont successivement :

- La comparaison entre i et le niveau lexical courant ic ;
- L'accès à l'entrée i de la TNL ;
- Le contrôle des droits sur le descripteur du SLCL ;
- L'accès à la catégorie c du SLCL ;
- Le contrôle des droits sur le descripteur du SL ;
- L'accès au SL ;
- L'accès à l'objet qui se trouve à un déplacement d de la base du SL.

Le nombre d'indirections et de contrôles nécessaires à l'obtention de l'objet (sans compter les transformations non répertoriées mais sous-jacentes "adresse virtuelle/adresse réelle"), justifié par le souci de protection, reste très pénalisant au niveau performance ; pour remédier à cet inconvénient important, un élément d'architecture matérielle spécifique a été conçu (voir T.A.L. chapitre 5).

L'accès aux objets peut donc s'effectuer dans un temps (qui englobe la conversion adresse virtuelle/adresse réelle) satisfaisant compte tenu du principe de l'exécution du code en pipe-line (cf l'évaluation des temps d'exécution des instructions de la machine langage ADA dans [RDL 82b]).

Le problème de performance introduit par l'adressage de type lexical étant parfaitement résolu, celui-ci ne garde que ses avantages considérables de simplicité de mise en oeuvre et surtout d'implantation de la protection des objets.

Il est également important de préciser que les mécanismes et les concepts développés pour l'adressage (niveau lexical, Segments Logiques, catégories,...) ont été particulièrement étudiés pour faciliter les opérations souvent délicates que nécessitent les appels et les retours de sous-programmes.

4.2.3. Le langage machine

Les instructions machine sont conçues pour expliciter de la manière la plus concise possible les différentes variantes des constructions de base du langage ADA, tout en leur conservant un caractère de généralité suffisant pour une implantation d'un autre langage à structure de blocs.

Le jeu d'instructions est particulièrement adapté à la traduction des expressions, des appels et retours de sous-programmes, des instructions de contrôle du flot d'exécution, des tâches et des mécanismes de rendezvous, à l'accès aux composants de record et de tableaux (ou même de tranches de tableaux).

Ce langage est de haut niveau sémantique et est appelé intermédiaire (origine de l'appellation MLI) car il correspond par sa nature et ses structures aux langages intermédiaires utilisés dans les compilateurs avant la phase de génération de code.

La structure des instructions de ce langage machine évolué permet donc en particulier, d'éviter les étapes intermédiaires qui, dans les machines à architecture classique, conduisent à effectuer des préservations de résultats temporaires dans des ressources visibles de la machine au prix d'un supplément d'instructions ; dans MLI, les expressions arithmétiques du programme sont traduites au moyen d'instructions à 2 ou 3 opérandes, un opérande désignant un immédiat, une variable scalaire, un composant de record ou un composant de tableau.

4.2.3.1. Caractéristiques principales

Il n'est pas question dans ce rapport de présenter le langage machine de manière exhaustive (le langage possède actuellement de l'ordre de 200 instructions) ; la description complète des instructions ainsi que les corrections apportées se trouvent dans [RDo 81b] et [RDL 82b].

En conséquence, la présente description se borne à exposer les principales caractéristiques du langage machine qui en font un langage différent de ceux rencontrés habituellement.

a) Le code machine est d'un haut niveau sémantique

Comme cela a déjà été souligné, le code machine est du type des langages intermédiaires utilisés par les compilateurs ;

Il est, par rapport aux codes machines classiques, plus symbolique et plus proche du langage source, ADA en l'occurrence.

A titre d'exemple, les appels ou les retours de sous-programmes sont exécutés en une seule instruction MLI.

b) Le code machine est linéarisé

Le choix (classique) d'un code linéarisé a été pris en fonction des inconvénients qu'offre l'autre alternative, c'est-à-dire le choix d'un code arborescent.

Le codage du langage machine sous la forme d'un arbre peut être intéressant dans le cadre d'un système d'édition des programmes interactif ; à la commodité de disposer d'une représentation unique pour l'exécution sur la machine et pour l'édition symbolique du programme en cas d'erreur, s'ajoute l'avantage, quand le langage machine est proche du langage source, de posséder un compilateur de taille réduite (c'est ainsi que la forme arborescente a été adoptée pour la machine orientée LISP, M3L).

En revanche, le choix d'un code arborescent, ne permet pas sa compaction et, surtout, pose des problèmes de mémorisation du point courant d'exécution lors des changements de contexte ; cette approche nécessite la gestion d'une pile qui doit elle-même être sauvegardée lors d'un changement de contexte en mode multi-programmation et dans le cadre d'exécution par tâches.

C'est donc la dégradation des performances et la taille du code engendré dans l'approche arborescente qui a imposé la solution linéarisée pour la forme du langage machine.

c) Le code machine est compact

La compaction du code, dont les avantages ont déjà été évoqués (voir § 1.3 et 2.1.2.1), est assurée en grande partie par le haut niveau sémantique des instructions et par leur linéarité.

Afin de renforcer la compacité des programmes traduits en langage machine , plusieurs autres options ont été retenues :

1. La variabilité du format des instructions ;
   Les instructions de la machine ont entre 0 et 3 opérandes et peuvent contenir des champs supplémentaires pour le ou les contrôles éventuels (voir plus loin) ; les opérandes sont eux-mêmes représentés sous forme de champs consécutifs de nature et de format variable (descripteur d'opérande, adresse lexicale,...) ; la longueur moyenne des instructions varie entre 4 et 120 octets mais le domaine de variation s'étend de 1 à plusieurs dizaines d'octets (tous les champs utilisés dans une instruction sont alignés octets*) ;

2. La présence d'une pile d'évaluation ;
   La pile d'évaluation sert à mémoriser des résultats intermédiaires nécessaires au calcul d'expressions arithmétiques complexes ou à l'accès à des objets d'une structure particulièrement élaborée (composant d'un record élément de tableau, élément de tableau multi-dimensionnel,...) ; bien qu'elle soit prévue comme devant être peu fréquente, l'utilisation de la pile joue un rôle d'implicite sémantique qui contribue à la compacité du code ;

3. L'usage de la pondération ;

   Le principe de la pondération est tiré du codage optimal de l'information étudié par HUFFMAN [Huf 52] ; ainsi les valeurs des immédiats (littéraux) sont codés sur une longueur de bit dépendant de leur fréquence d'utilisation qui est basée sur des études statistiques (solution déjà adoptée par WILNER pour le B 1700) ; la création d'instructions spéciales manipulant des valeurs privilégiées comme 0 ou 1 a également été envisagée ;

* Une première version du langage de MLI demandait une manipulation des instructions au niveau bit ; cette solution a été abandonnée pour des raisons de complexité d'implémentation (et de risques de dégradation de performances) mais les premières comparaisons réalisées entre l'ancien code (aligné bit) et le nouveau (aligné octet) montrent que la compaction reste sensiblement inchangée.

4. L'utilisation de descripteurs ;

Deux types très différents de descripteurs contribuent à la compaction du code :

- D'une part, des descripteurs "en ligne" qui sont des champs particuliers d'une instruction ; ils évitent au code la perte de compacité qu'entraîne une forme trop "procustéenne" quand ils servent à la caractérisation des opérandes ou à celle d'autres descripteurs destinés aux contrôles ; ils permettent surtout l'intégration des contrôles qui, sinon, demanderaient l'emploi d'instructions spéciales ou, comme dans les langages bas niveau traditionnels, l'utilisation de plusieurs instructions, quand ils sont équivalents aux descripteurs du 2ème type présentés ci-dessous ;

- D'autre part, des descripteurs non inclus dans le code, qui contiennent les informations nécessaires au(x) contrôle(s) demandé(s) dans l'instruction (descripteurs de tableaux, d'intervalles,...) ; la compacité obtenue par leur utilisation provient de l'économie d'espace réalisée sur la taille du code (au détriment, il est vrai, d'un accès mémoire supplémentaire).

  Remarque : Les descripteurs utilisés par le code machine ne sont pas associés à chaque objet du programme mais à chaque type d'objet ; cette approche permet d'obtenir un code plus compact en ne dupliquant pas des informations décrivant les propriétés d'un type pour tous les objets du types.

A défaut d'une énumération complète des instructions qui composent le langage machine et en guise d'illustration, l'instruction en langage évolué X.A = T(I) + Z.B est traduite (avec les contrôles nécessaires) en une seule instruction machine alors que dans les machines classiques, cette instruction est traduite dans au moins 2 à 4 et souvent plus d'instructions machine.

4.2.3.2. Conséquences sur le fonctionnement

Les conséquences attendues sur le fonctionnement de la machine engendrées par les différentes options prises pour le langage machine sont plus ou moins évoquées lors de leur présentation dans le paragraphe précédent.

Outre les avantages en termes de performance classiquement avancés des instructions à haut niveau sémantique (parallélisme possible au niveau de l'exécution, débit mémoire réduit de par la taille des instructions), le langage machine recherche surtout les gains en encombrement mémoire qui permet la compaction à travers les aspects déjà évoqués comme la pondération, l'utilisation d'une pile d'évaluation ou l'usage de descripteurs.

En ce qui concerne les descripteurs qui permettent le traitement efficace par la machine des types construits par le programmeur, il est important de souligner que, dans un souci d'efficacité, les descripteurs ne sont référencés que lorsqu'ils sont nécessaires, soit pour la réalisation des instructions, soit pour des contrôles de cohérence ; ainsi, par exemple, une affectation A := B impliquant deux objets du même type est réalisée sans l'utilisation de descripteurs.

Ainsi que cela a été annoncé au début de ce paragraphe (4.2.3), les instructions machines ont été conçues pour être particulièrement adaptées aux diverses constructions de base de ADA sans que cette spécificité soit trop contraignante dans l'éventualité du support d'un autre langage (à structure de blocs) ; les instructions utilisées dans le cas de manipulation de tableaux sont par exemple parfaitement réutilisables dans n'importe quel langage autorisant les tableaux comme type de donnée structurée.

Les mécanismes d'appels et de retours de sous-programmes, qui forment traditionnellement le principal obstacle à la génération d'un code performant (cf § 2.2.5.1), ont été soigneusement étudiés de manière à n'apparaître au niveau du code que sous la forme d'une seule instruction (par type d'appel) et de laisser la machine effectuer toutes les opérations nécessaires sans en imposer le choix et le déroulement au programmeur ou même au compilateur ; de plus, la complexité des appels et retours de sous-programmes que ces instructions sont capables de prendre en compte, permet d'affirmer que le support d'une grande partie des langages à structure de blocs (ayant sensiblement les mêmes règles de visibilité) par MLI ne poserait pas de gros problèmes en ce qui concerne cet aspect important de langage.

Afin d'illustrer les avantages du langage machine et principalement sa compaction, voici un exemple d'instruction MLI et ses équivalents dans différentes machines.

L'instruction est A : = B + C où
B et C sont des entiers
A un élément de l'intervalle [1, 10 000]
Un contrôle sur les 2 bornes est nécessaire à l'affectation.

La traduction en code machine MLI est alors l'instruction :

ASSIGNER, DO1,@B, DO1,@C, DO1,@A, DCA, BS avec

ASSIGNER : Code opération

DO1 : Descripteur d'opérande (indique la forme variable simple pour A, B et C)

@B,@C et@A : adresse lexicales de B, C et A

DCA : descripteur de contrôle d'affectation

BS : Valeur de la borne supérieure (la borne inférieure,étant égale à 1 ,a pu être intégrée au DCA)

Dans le cas où les variables sont locales (adresses lexicales de type local), le nombre d'octets que représentent cette instruction peut varier entre 10 et 13 octets.

Une traduction équivalente dans les codes machines différents donnerait les résultats suivants :

DPS/7	30 octets
MC 68000	16 à 18 octets
NS 16000	14 à 18 octets
Mini 6	26 à 30 octets

Il faut souligner que dans le cas où les adresses seraient globales, l'avantage de MLI serait encore plus probant mais les comparaisons sont plus difficiles à réaliser en raison des multiples cas de figures que peut rencontrer alors la traduction dans les différents codes machine.

4.3. PROPOSITION D'ARCHITECTURE MATERIELLE

Les grands principes de l'architecture logique étant acquis, une première proposition d'implémentation matérielle a été élaborée conjointement par Claude RENVOISE,qui est à l'origine de l'organisation générale, Marc LEGOUX et Jacques BRYGIER.

Cette proposition n'est pas une solution définitive ni même complète ; elle offre plutôt un cadre de réalisation dans lequel doivent s'insérer certaines fonctionnalités non encore prises en compte comme la gestion des E/S ou même les mécanismes de ramasse-miettes et de mise au point étudiés par ailleurs.

Seuls l'Approvisionneur d'Instructions (AI) conçu en grande partie par Marc LEGOUX qui réalisa le Buffer d'Instructions, que nous avons ensuite resitué dans son environnement et le Transformateur d'Adresses Lexicales (TAL) ont bénéficié d'une définition plus précise.

4.3.1. Organisation générale

Ce paragraphe a pour but de donner une vision globale de la proposition d'architecture matérielle à travers sa "philosophie" et présente un schéma général synoptique.

4.3.1.1. Principes et motivations

Ainsi que cela est évoqué dans l'Introduction, la réalisation d'un support matériel à une architecture logique conçue préalablement n'obéit pas à des règles de méthodologie très précises.

La "philosophie" d'implémentation matérielle adoptée pour MLI est la décomposition en **modules** qui est soumise aux deux objectifs suivants :

1. Le regroupement dans un même module des fonctionnalités de même nature afin de limiter et simplifier les interfaces logiques existants entre les modules ;

2. La définition de modules dont la complexité doit être compatible avec la capacité d'intégration disponible en technologie VLSI HBMOS4* en fonction de l'échéance visée.

Ainsi le regroupement de modules pourra être envisagé en fonction de l'évolution de la technologie, ce qui renforce l'hypothèse de la proposition d'architecture matérielle devant être considérée comme un premier support de validation des fonctionnalités logiques de MLI et non comme une réalisation définitive.

4.3.1.2. Description

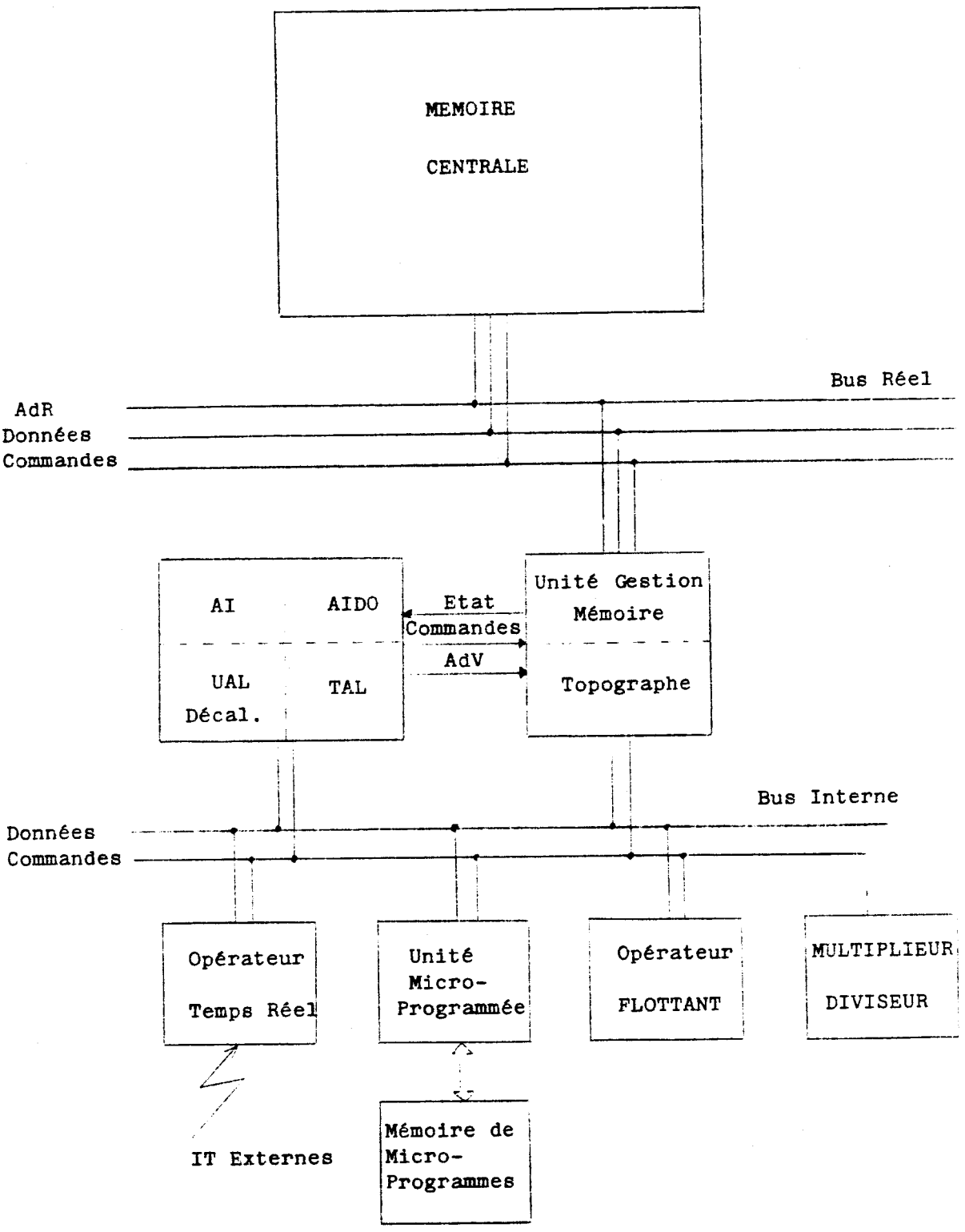
L'architecture matérielle proposée se compose principalement de deux parties (voir figure n°23) :

- La Mémoire Centrale MC ;
- Le Processeur Central PC.

Ces deux éléments sont en communication grâce à un bus principal appelé **Bus Réel** en raison des adresses réelles qu'il véhicule (par opposition aux adresses virtuelles).

Le PC est quant à lui, composé de plusieurs modules interconnectés soit directement pour certains d'entre eux, soit par l'intermédiaire d'un autre bus appelé **Bus Interne**.

* Il s'agit de la dénomination de la technologie de circuits intégrés de type HMOS développée à la Compagnie Bull telle qu'elle est prévue pour 1984.



AdR : Adresses Réelles
AdV : Adresses Virtuelles

Figure n°23 : Architecture matérielle de MLI

Cette architecture se veut extensible et devrait autoriser la multiplication :

- D'une part, des modules mémoire afin que la Mémoire Centrale puisse être organisée en plusieurs bancs ;

- D'autre part, des Processeurs Centraux qui réaliseraient ainsi une machine multi-processeurs configurable suivant le type d'applications demandées.

L'augmentation du débit mémoire nécessaire à l'alimentation des PC en instructions et en données peut alors conduire à deux types de solutions :

1. Le maintien du Bus Réel et l'utilisation de techniques d'entrelacement ;

2. Le remplacement du Bus Réel par un réseau d'interconnection (cross-bar, bénès,...) permettant la sortie simultanée de mots provenant d'adresses indépendantes sur les différents bancs.

Cette dernière solution nécessiterait alors certains aménagements comme, par exemple, l'association aux PC de mémoires-cache afin de diminuer le traffic vers la Mémoire Centrale.

Ce problème n'est mentionné ici que dans un souci de généralité et n'a pas connu d'études plus approfondies étant entendu que l'orientation multi-processeur se situe en dehors du cadre actuel du projet.

Remarque : Comme cela a déjà été annoncé, le problème des E/S n'a pas été abordé et la connection avec les unités d'échanges périphériques non encore analysée, ce qui explique leur absence sur le schéma.

La suite du paragraphe est consacrée maintenant à une présentation plus détaillée des différents modules prévus dans l'architecture matérielle telle qu'elle est actuellement proposée.

4.3.2. La Mémoire Centrale

La Mémoire Centrale est une mémoire adressable octet permettant la lecture ou l'écriture simultanée de 1 à 4 octets consécutifs.

Le concept classique de mot de 32 bits n'a pas été retenu car il semble préférable que la machine puisse manipuler, directement et sans coût additionnel, un nombre d'octets inférieur à quatre, notamment en écriture.

Cette implémentation correspond à la possibilité offerte au niveau du langage ; elle permet l'écriture d'un booléen sur un octet ou d'un petit entier sur deux octets en un seul cycle mémoire et contribue à réduire les contraintes d'implantation des variables des programmes en mémoire centrale (suppression de l'alignement mot réalisé habituellement par le compilateur pour simplifier, par exemple l'accès aux entiers).

L'organisation de la mémoire (représentée schématiquement par la figure n° 24) révêt actuellement la forme suivante :

- La mémoire est constituée de 4 blocs d'octets numérotés de 0 à 3 ayant leur propre logique d'adressage ;

- Un octet d'adresse i est contenu dans le bloc de numéros i mod 4 ; l'adresse octet du premier octet d'un groupe d'octets qu'il faut lire ou écrire en mémoire est donc constituée d'une adresse de tranche notée a (correspondant à un niveau d'adresse dans un bloc) et de 2 bits notés $n_1 n_0$ indiquant le numéro de bloc où se trouve le premier octet dans la tranche ; l'adresse octet est donc la concaténation $a n_1 n_0$;

- La mémoire permettant un accès simultané de 1 à 4 octets, le nombre d'octets pouvant être accédés est exprimé sur 2 bits notés $q_1 q_0$;

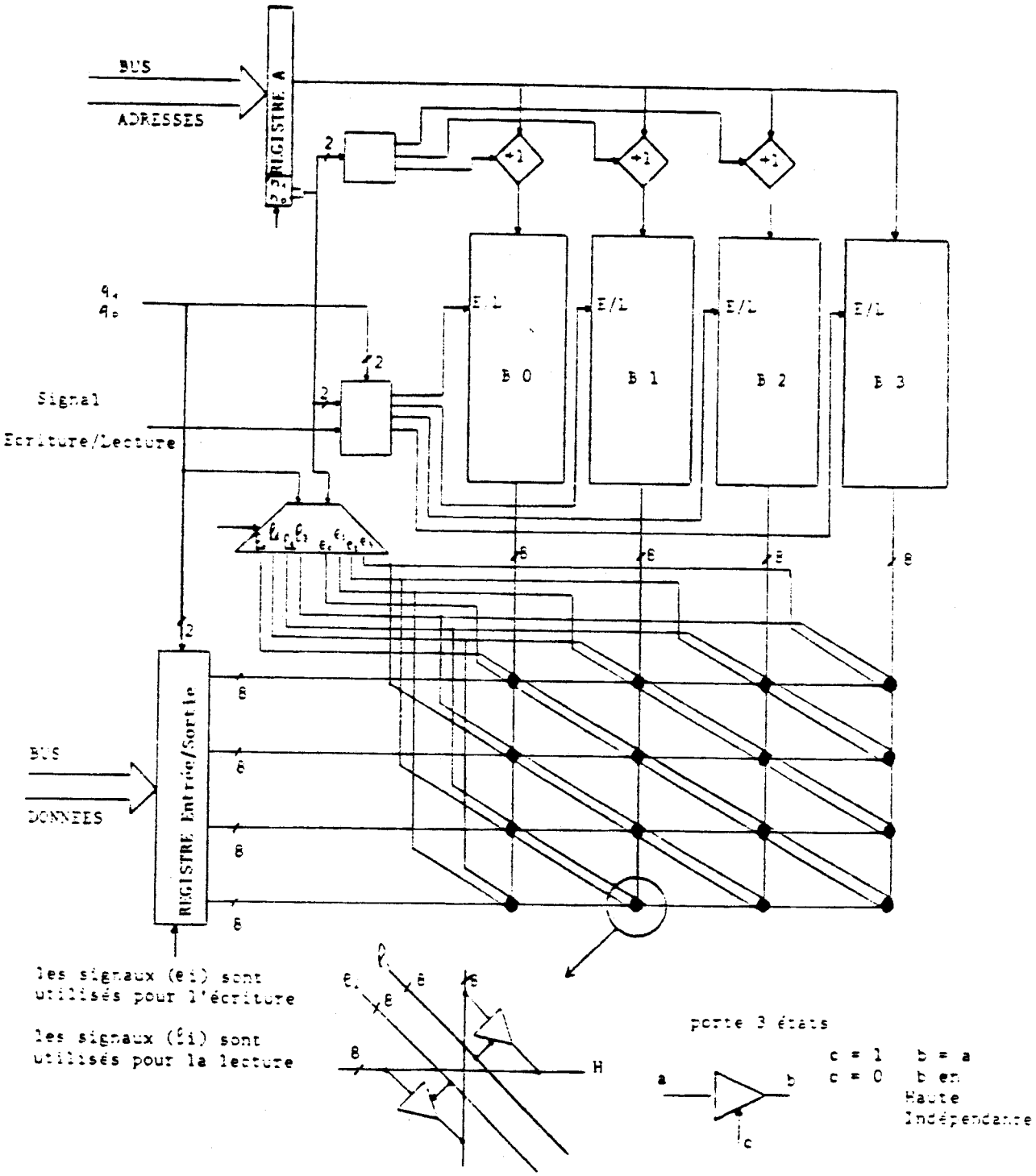
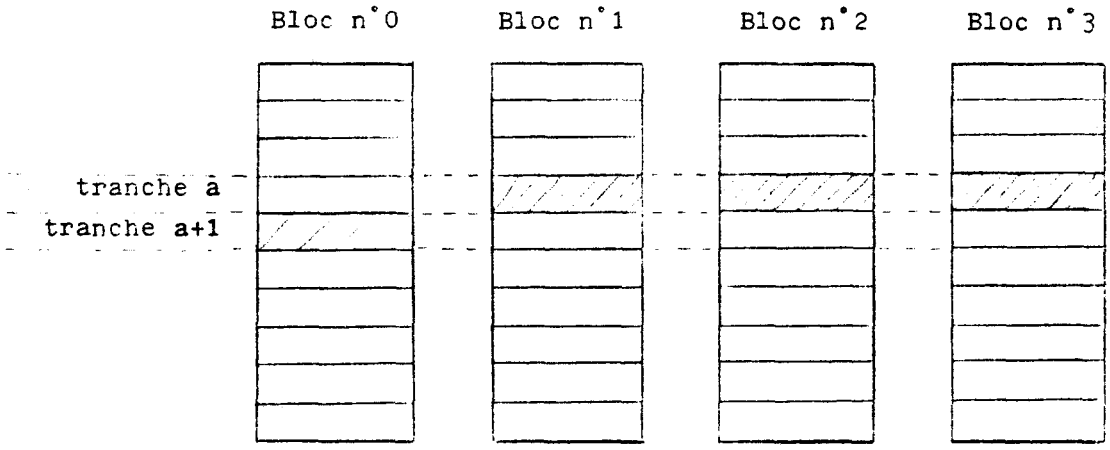


Figure n°24 : Organisation physique de la Mémoire

- Les deux bits n_1no commandent des additionneurs simplifiés placés devant 3 des 4 blocs octets afin d'augmenter conditionnellement de 1 l'adresse de tranche **a** ; ceci permet l'accès de 1 à 4 octets consécutifs situés dans une même tranche ou dans deux tranches consécutives (voir figure n°25) ;



q_1qo = 10 indique qu'il y a 3 octets à extraire
n_1no = 10 indique que le premier octet à extraire se trouve dans le bloc n°2.

Figure n°25 : Exemple d'un accès à un groupe de 3 octets

- La conjonction des bits n_1no et q_1qo permet de commander les signaux d'écriture différenciés sur les 4 bancs octet pour écrire simultanément de 1 à 4 octets consécutifs situés dans une même tranche ou dans deux tranches consécutives.

Remarque : Cette organisation particulière de la Mémoire Centrale demande à ce que les codes autocorrecteurs qui interviendront lors de la réalisation physique soient associés individuellement à chaque octet.

4.3.3. Le Processeur ADA

Le Processeur ADA est, parmi les éléments composant le PC, le plus original et le plus important ; c'est lui qui prend en charge l'exécution des instructions qu'il réalise lui-même ou qu'il fait réaliser par les autres éléments composant le PC.

Il dispose ainsi d'une logique suffisante pour réaliser lui-même les affectations ou encore les additions entières ; par contre, il confie les opérations en flottant à l'opérateur flottant et l'exécution des instructions complexes à l'Unité Microprogrammée.

En ce qui concerne le rôle de la microprogrammation dans le fonctionnement du PC, des études plus approfondies doivent être menées afin d'établir exactement la forme des microinstructions qu'implique la répartition des fonctions entre le PA et l'Unité Microprogrammée (Voir § 3.4).

Le PA est "le Processeur Langage" de la machine dans la mesure où lui seul connaît les caractéristiques langage de la machine (adressage lexical, descripteurs,...).

La machine langage ADA n'est pas une machine à registres, au sens conventionnel du terme, mais possède certains registres non programmables nécessaires à son fonctionnement (§ 4.3.3.1).

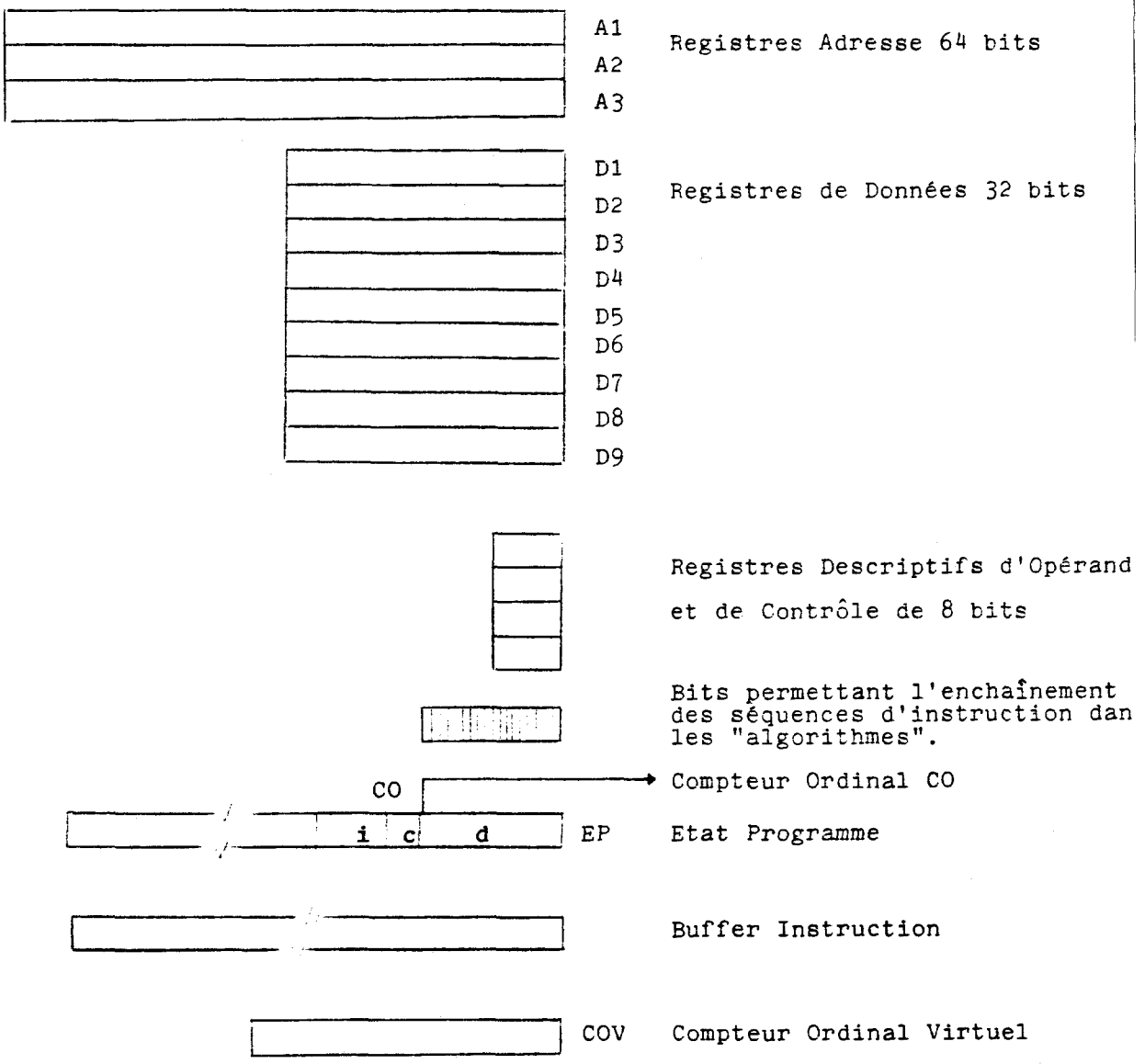
Le PA est constitué principalement :

- De l'Approvisionneur d'Instructions AI (§ 4.3.3.2) ;
- De l'Analyseur d'Instructions/Décodeur d'Opérandes AIDO incluant une U.A.L. (§ 4.3.3.3) ;
- Du Transformateur d'Adresses Lexicales qui est largement développé dans le chapitre 5.

4.3.3.1. Les Registres de PA

Les Registres de PA représentent d'une part, les registres de service, d'autre part, des registres généraux qui sont apparus nécessaires à l'exécution des instructions du langage machine qui est explicitée sous la forme d'une pseudo-programmation dans ce rapport [RDL 81b].

La plupart des registres dépendant de PA se trouvent physiquement implantés dans le module PA ; voici leur désignation :



L'Etat Programme EP est le registre correspondant à ce qu'il est de coutume d'appeler l'état programme ; l'état programme contient les informations de base concernant la tâche, la procédure et l'instruction courante.

Il permet de reconstituer :

- Au niveau tâche, l'environnement nécessaire au moniteur pour la gestion des tâches ;

- Au niveau procédure, l'environnement nécessaire pour assurer l'enchaînement des procédures ;

- Au niveau instruction, l'environnement nécessaire au processeur ADA pour la génération d'adresses virtuelles sur le code et les données.

EP comprend les champs suivants :

- T : Numéro de tâche, permet d'identifier de manière unique la tâche dans le système ;

- IAR : Inhibition des erreurs arithmétiques ; précise que ces types doivent être ignorés ;

- MAP : Indicateurs liés à la mise au point ;

- CO : Compteur Ordinal ;

- Rnr : Niveau Lexical Courant ;

- Rsp : Sauvegarde Procédure ; désigne la position de la zone de sauvegarde associée à la procédure courante dans le SLCL courant ;

- Rsv : Descripteur du Segment Virtuel dans lequel sont créés les Segments Logiques ;

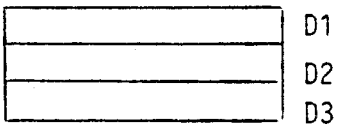
- Rtop_sv : Position de la zone occupée dans le segment désigné par Rsv ;

- SLP : Descripteur du Segment Logique contenant la pile d'évaluation ;

- SLPtop Position de la zone occupée dans le segment désigne par SLP.

Les autres registres dépendant de PA se trouvent physiquement implantés dans d'autres modules ; il s'agit :

- Des Registres du MUL - DIV (32 bits)



- Des Registres du FLOTTANT (64 bits)

Les Registres présentés sont pour l'instant issus des premières décompositions des instructions du langage machine et représentent un support aux registres que nécessitent l'architecture logique ; leur nombre et/ou leur fonction restent naturellement soumis aux contraintes matérielles éventuelles qu'entraîne toute implémentation.

Il n'est par exemple pas fait mention des registres nécessaires à la sauvegarde du contexte dans certains cas d'interruption de AIDO (lors de la recherche des Opérandes en mémoire).

4.3.3.2. L'Approvisionneur d'Instructions

Le besoin d'un élément autonome assurant l'alimentation du PA en instructions afin d'éviter toute dégradation de performance est né des deux constatations suivantes :

- Le Compteur Ordinal CO qui désigne l'instruction à exécuter contient une adresse lexicale triplet (i, c, d) ;

- Le Code machine possède un format extrêmement variable.

Ce dernier point a conduit simultanément à la mise en place d'un mécanisme de chargement des registres internes de PA avec un nombre variable d'octets dépendant de la taille des champs élémentaires des opérandes.

Le composant principal de AI est le **Buffer d'Instructions** (BI) qui reçoit les instructions provenant de la Mémoire Centrale ; sa taille est prévue pour être de l'ordre de 12 Octets mais doit être validée par simulation ; la logique mise en place en vue de l'exploitation de son contenu par AIDO permet d'extraire 1 à 4 octets consécutifs à n'importe quel endroit du BI.

Le fonctionnement de AI est de gérer l'alimentation du BI en fonction des commandes reçues par les autres éléments du PC (rupture de séquence, changement de contexte,...) ; une description beaucoup plus précise peut être trouvée dans [RDL 81b].

4.3.3.3. L'Analyseur d'Instructions-Décodeur d'Opérandes

La fonction de AIDO est d'abord d'analyser le code opération pour déterminer le traitement à effectuer et obtenir les propriétés caractéristiques de l'instruction ainsi que, dans la plupart des cas, de ses opérandes (nombre, type,...) ;

Chaque opérande de l'instruction est ensuite successivement analysé pour extraire sa valeur ou son adresse selon la nature de l'instruction (normale ou longue) et/ou la nature de l'opérande (source ou destination).

AIDO peut utiliser les services d'une Unité Arithmétique et Logique permettant les décalages sur 40 bits* ; la présence d'une U.A.L. intégrée est motivée par :

- Le décodage des opérandes ; celui-ci nécessite souvent des opérations arithmétiques nombreuses dans le cas où l'objet à accéder est un composant de record ou encore un élément de tableau ;

- L'exécution d'instructions machine ; certaines instructions correspondent à des opérations simples comme des additions entières ou logiques ou des opérations sur des chaînes d'octets ou de bits.

Le schéma de fonctionnement définitif de AIDO devra assurer un maximum d'efficacité au décodage des instructions ; il devra en particulier réaliser un parallélisme d'exécution (par pipe-line et/ou temps partagé) en raison des nombreuses interruptions qui interviennent par exemple dans le décodage des opérandes (nombreux accès mémoire possibles).

4.3.4. Les autres modules proposés

Outre le PA, le Processeur Central regroupe certains autres modules qui assurent l'interface entre le PA et son environnement (Topographe-UGM), complètent le PA dans le support du langage (Unité Microprogrammée), contribuent à la fois à l'interface avec l'environnement et au support du langage (Opérateur Temps Réel) ou servent à améliorer les performances à l'exécution (Opérateurs Flottants et MUL/DIV).

* La largeur de 40 bits de l'U.A.L. est imposée par l'exécution des instructions longues qui nécessitent la possibilité d'effectuer des opérations arithmétiques sur des adresses virtuelles.

La conception de ces modules n'a atteint pour l'instant que le stade des fonctionnalités qui sont évoquées brièvement.

4.3.4.1. Le Topographe - Unité de Gestion de la Mémoire

Le Topographe assure la tâche classique de conversion des adresses virtuelles en adresses réelles.

L'espace mémoire étant paginé, la conversion repose sur des tables résidentes en mémoire centrale (au moins pour la racine) qui décrivent les correspondances page virtuelle/page réelle ; l'occupation en mémoire des tables est liée à la taille de la mémoire réelle (organisation du type IBM System 38) plutôt qu'à la taille de la mémoire virtuelle (type NS 16000).

Afin d'accélérer les conversions, le Topographe devrait disposer d'une mémoire associative qui conserve les correspondances des dernières pages virtuelles référencées ; leur nombre est prévu à 12 mais devra être validé par simulation en fonction de la probabilité de succès désirée.

L'Unité de Gestion de la Mémoire a pour fonction de gérer les requêtes mémoires et de reconfigurer les mots lus de la mémoire, en particulier lorsqu'il s'agit de champs de bits qui se trouvent "à cheval" sur des mots ou des octets ; le cadrage des octets ou des champs de bits est explicité plus longuement dans [RDL 81b].

La réalisation de l'UGM est actuellement envisagée de manière à ce que les informations à lire ou à écrire en mémoire soient transmises ou reçues du PA ou des autres unités par l'intermédiaire du Bus Interne.

4.3.4.2. L'Unité Microprogrammée

Cette Unité est prévue pour assurer l'exécution des instructions du langage machine dont la complexité impose une taille des microprogrammes correspondants non supportable par le PA.

Le rôle de cette unité et, a fortiori, sa réalisation n'ont pas été clairement définis à ce stade de conception de l'architecture matérielle ; la solution évoquée au § 3.4 laisse supposer qu'en plus de ses fonctions de gestion de la mémoire RAM de microprogrammes (qui peuvent lui être adressés par l'environnement extérieur au PC), cette unité pourra être amenée à réaliser un certain nombre d'opérations préalables sur les microprogrammes (redéfinition des adresses en fonction de l'implantation dans la RAM du PA, contrôles des appels de sous-microprogrammes,...) dont elle assurera le transfert vers la RAM du PA conformément aux règles d'utilisation du Bus Interne.

La taille de la mémoire de commande associée devra être suffisamment grande pour contenir, non seulement les microprogrammes associés aux instructions complexes du langage machine, mais également les fonctions systèmes dont la stratégie d'implantation réunirait les deux possibilités présentées au § 3.2.2.1.

4.3.4.3. Les Opérateurs Arithmétiques

Pour assurer des performances satisfaisantes à la machine, l'association de deux processeurs arithmétiques câblés au Processeur ADA est envisagée ; ces processeurs pourront être choisis parmi des modules matériels existant ou pourront être redéfinis.

Il est actuellement prévu d'intégrer au Processeur Central :

- Un Opérateur Multiplication/Division qui doit réaliser la multiplication et la division de nombres entiers sur 8, 16 et 32 bits ;

- Un opérateur Flottant qui doit réaliser des opérations arithmétiques (addition, soustraction, multiplication et division) sur des nombres en flottant de 32 et 64 bits.

4.3.4.4. L'Opérateur Temps Réel

Cet opérateur a pour fonction de gérer l'horloge et les délais qui interviennent dans l'exécution des applications à la charge du Processeur Central.

La gestion des délais est envisagée sous la forme d'une implémentation d'un modèle utilisant des **Descripteurs de Délais** et qui est décrit dans [RDL 81b].

Précisons simplement que les Descripteurs de Délais peuvent être associés :

- A un délai logiciel au sens ADA utilisé comme tel par le PA ;

- A une interruption matérielle.

4.3.5. Le Bus Réel

Il ne s'agit ici que d'énoncer les principales caractéristiques que devra réunir le BusRéel sans en évoquer la réalisation matérielle.

L'architecture proposée autorisant l'échange entre la MC et le PC d'informations sur 32 bits, la largeur du bus doit être au minimum de 32 bits.

Compte tenu des performances souhaitées et en accord avec la décomposition des instructions machine déjà évoquée (§ 4.3.3.1), le temps de cycle de la MC envisageable est de 250 ns ; le bus doit donc avoir des performances compatibles et avoir une bande passante de 4 Mmots/s pour une version monoprocesseur.

Le multiplexage des adresses et des données n'apparaît pas souhaitable car le gain en nombre de fils (32) se ferait au prix d'une dégradation de performance probablement très importante et demanderait en outre, une logique supplémentaire sur toutes les cartes connectées au bus.

Les problèmes engendrés par les interruptions conduisent à l'examen de deux approches (Multibus/Versabus ou Megabus du Mini 6) qui est abordé dans [RDL 81b].

CHAPITRE 5

SCHEMA D'IMPLEMENTATION ET EVALUATION DU T.A.L.

5. SCHEMA D'IMPLEMENTATION ET EVALUATION DU TAL

5.1. OBJECTIFS DE L'ETUDE

Un des aspects caractéristiques de la machine langage ADA est le schéma d'adressage lexical qui reflète directement les règles de visibilité des langages à structure de blocs (cf § 4.2.2.).

Le Transformateur d'Adresses Lexicales (T.A.L.) a pour principale fonction la traduction d'une adresse lexicale en une adresse virtuelle qui, transformée en adresse réelle par le topographe de la machine, permettra d'adresser la mémoire centrale ; le T.A.L. effectue également d'autres opérations corollaires qui sont explicitées par la suite.

Le T.A.L. se situe à l'aval du décodage des instructions qui, après l'analyse de la structure des instructions machine, lui présente les opérandes sous forme d'adresse lexicale correspondant aux données destinées aux opérateurs de la machine (décodeur, MUL/DIV, opérateur flottant,...) qui réalisent les instructions ; dans cette organisation, le T.A.L. est pour AIDO le premier élément de la chaîne des fonctionnalités d'accès à la mémoire.

Si le T.A.L. est étudié en tant que module matériel spécifique, c'est que, pour répondre aux objectifs de la machine langage ADA, le souci d'efficacité d'interprétation du code intermédiaire l'exigeait.

Cette étude, nécessaire sur le principe, permet surtout de procéder à une première estimation des performances en temps et en coût de matériel à partir d'une définition précise du T.A.L.

La définition actuelle du code machine a conduit à un découpage fonctionnel du T.A.L. qui ne devrait pas être remis en question fondamentalement.

L'implémentation proposée reste plus sensible aux modifications tant en fonction de la technologie adoptée que du choix de l'architecture, qui doit être validé par des résultats statistiques non disponibles actuellement.

L'étude envisagée comporte trois étapes :

1. La description fonctionnelle de toutes les opérations demandées au T.A.L. ;

2. La description d'une implémentation possible avec la plupart des justifications des évaluations effectuées en temps et en nombre de portes ;

3. Le fonctionnement effectif du T.A.L. à travers le schéma d'implémentation proposé et à l'aide de graphes représentant le déroulement de chaque mode d'utilisation du T.A.L.

5.2. DESCRIPTION FONCTIONNELLE DU TAL

5.2.1. Adressage lexical et Adressage Virtuel

Voici ci-dessous brièvement rappelées les caractéristiques de la mémoire virtuelle et de l'adressage lexical de la machine ADA qu'il faut prendre en compte pour les fonctions de calcul d'adresse. Le principe de gestion de la mémoire virtuelle fait l'objet du § 4.2.1. tandis que l'adressage lexical a été décrit dans le § 4.2.2.

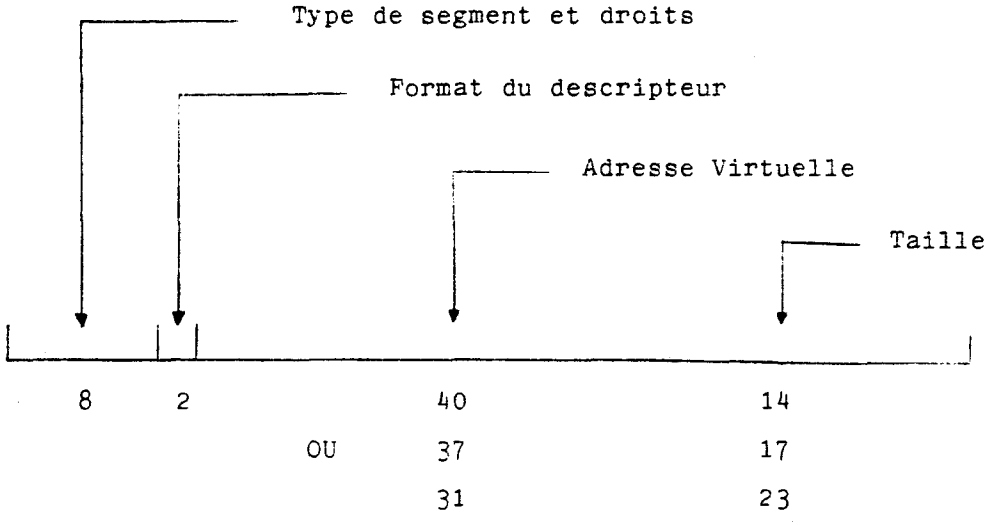
a) L'adressage virtuel

La mémoire virtuelle de la machine ADA est une mémoire imaginaire de 2^{40} octets, segmentée et paginée (pages de 512 Ø). Elle contient deux types de segments tous deux accessibles au travers de descripteurs :

- Les segments virtuels (notés SV) créés par le "Gestionnaire de la mémoire virtuelle", fonctionnalité strictement logicielle ;

- Les segments logiques (notés SL) parties contigues de SV créés avec l'aide de fonctionnalités matérielles.

Ces deux types de segments sont accessibles au travers de descripteurs de segment ayant la structure suivante :



Le champ Format indique soit un descripteur nul, soit un descripteur dont les caractéristiques, **Adresse virtuelle** d'implantation de segment dans la mémoire et **taille** du segment, sont exprimées en octet, double mot (8Ø) ou page (512 Ø).

Il faut noter que cette répartition des 64 bits du descripteur de segment permet de découper 2^{40} octets dans des segments de taille limite respectivement de 16 KØ, 1 MØ, 4 GØ.

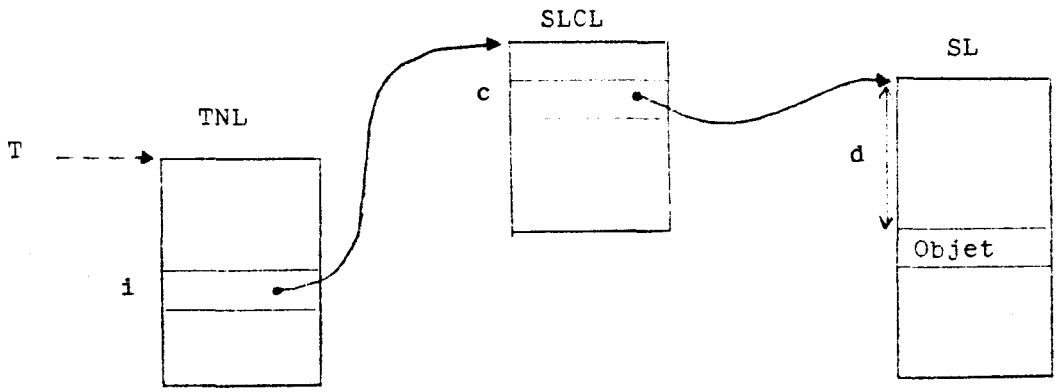
Ce descripteur est utilisé pour décrire tous les segments accessibles de la machine, que ces segments contiennent des informations "utilisateurs" (code machine, données,...) ou des informations "système" (tables TNL, segments SLCL,...). Notons toutefois que certains types de segments système (les SLCL par exemple) nécessitent des contrôles supplémentaires requiérant des données qui sont codées dans le champ **taille** du descripteur, celui-ci s'avérant intrinsèquement trop large pour la taille réelle de ce type de segment logique.

b) L'adressage lexical

Le T.A.L. transforme une adresse lexicale de la forme T, i, c, d en une adresse virtuelle sur 40 bits interprétable par le topographe de la machine.

Le principe de l'adressage lexical est le suivant :

- Pour chaque contexte associé à un numéro de tâche T, la désignation d'un objet se fait au travers de deux descripteurs comme le représente le schéma ci-dessous :



i : Niveau lexical de la partie de programme où est fait la demande d'accès à l'objet

c : Nom de la catégorie dans le SLCL (le nom N correspondant à la (N + 1)ième catégorie).

d : Déplacement dans le Segment Logique correspondant au nom du 1er octet significatif de l'objet à accéder.

Pour accéder à l'objet désigné par l'adresse lexicale T, i, c, d la suite des opérations à effectuer est la suivante :

- Déterminer le contexte de la tâche appelée T ;
- Lire le descripteur se trouvant au niveau i de la TNL et qui contient l'adresse virtuelle du début du SLCL ;
- Lire le descripteur se trouvant dans la catégorie c du SLCL et qui contient l'adresse virtuelle du début du Segment Logique où se trouve l'objet ;
- Lire l'objet se trouvant à un déplacement d du début du SL.

Cette liste d'actions est volontairement simplifiée et ne prend pas en compte les différents contrôles ainsi que les trois transformations "adresse virtuelle - adresse réelle" réalisées par le topographe de la machine.

Il est évident que même sans connaître précisément les mécanismes mis en jeu pour la réalisation de cet accès, le temps nécessaire tel qu'il vient d'être décrit est prohibitif pour n'importe quel schéma de décodage.

Afin de pallier cet inconvénient, un système d'accélération s'avère indispensable.

Le système proposé dans ce rapport offre une solution naturelle et classique dans son principe et qui consiste en :

a) Une mémoire associative conservant des descripteurs de segments ;

b) Des mécanismes réalisant des contrôles et la génération d'un ensemble d'informations nécessaires à l'accès à l'objet.

5.2.2. La fonction Mémoire Associative

Schématiquement, la mémoire associative peut être partagée en trois parties :

- Une matrice A contenant des informations du type T, i, c ou T, i utilisées pour la recherche associative proprement dite ;

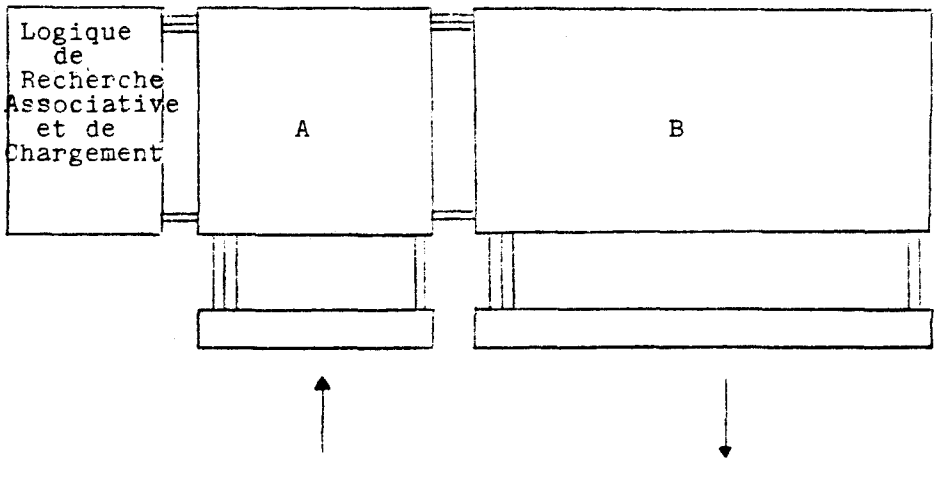
- Une matrice B contenant des informations du type descripteur de SL, tels qu'ils sont implantés dans les catégories c des SLCL ou des TNL ;

- Un ensemble de circuits logiques réalisant la recherche associative sur la matrice A et la lecture dans la matrice B ainsi que le chargement dans la matrice A et la matrice B.

La description de ces parties suffit presque à déterminer le fonctionnement de la mémoire :

- L'information présentée à la mémoire pour la recherche associative regroupe trois (ou deux, nous verrons plus tard pourquoi lors de la description du fonctionnement du T.A.L.) champs de l'adresse lexicale à transformer : T, i, c (ou T, i) ;

- Si cette information est présente (dans A), le descripteur associé (dans B) est rangé dans le registre de sortie, disponible pour une opération du T.A.L.



Ce schéma est essentiellement synoptique et ne laisse en rien présager d'une implémentation matérielle qui pourrait être soumise à des contraintes qu'il reste à définir (place, coût,...).

La solution d'implémentation proposée dans ce rapport répond globalement à certains de ces problèmes.

5.2.3. Description des modes d'utilisation

La mémoire associative est une partie importante du T.A.L. mais n'est en fait qu'une première étape de la transformation.

Cette étape permet d'obtenir un descripteur qui est l'information de base sur laquelle va s'articuler les différentes opérations qui conduisent à l'obtention d'une adresse virtuelle fiable grâce à divers contrôles effectués simultanément par le T.A.L.

Afin de rendre clairs les modes de fontionnement possibles du T.A.L., nous allons étudier chaque cas séparément, un cas étant caractérisé par les informations fournies en entrée au T.A.L. et les informations que celui-ci présente en sortie après traitement.

Pour cela, précisons la nature des différentes informations manipulées par le T.A.L. (la liste regroupe tous les cas possibles).

Informations en entrée :

- Champs de l'adresse lexicale : T, i,c,d (exprimé en bits ou en octets) ;
- l longueur en bits ou en octets de l'objet à accéder ;
- ic la valeur du niveau lexical courant ;
- L'ensemble des commandes qui pilotent l'unité de commande du T.A.L. ;
- Le descripteur de la TNL courante ;
- Les descripteurs de SLCL ou de SL destinés au chargement de B.

Informations en sortie :

- Adresse virtuelle sur 40 bits (adresse octet) ;
- n nombre d'octets consécutifs à extraire de la mémoire à partir du premier octet désigné par l'adresse virtuelle ;
- Champ d" servant d'information complémentaire en vue du repérage de l'objet dans la suite des octets à extraire lorsque la longueur de celui-ci est exprimée en bits ;
- Commandes à destination du topographe ;
- Commandes à destination de l'Unité de Commande Principale (pouvant être située pour l'instant dans le Décodeur d'Instructions).

NOTES

La représentation des signaux de commande en entrée comme en sortie n'est pas encore définitivement formalisée à ce stade de la conception ; ceci ne gêne en rien la description des fonctionnalités.

Les signaux relevant typiquement du matériel (horloge, interruptions,...) ne sont pas explicités ou sont intégrés dans les signaux de commande.

Conventions et terminologie

- Les noms des octets (ou de bits) sont de la forme 0, 1, 2,... N : le nom N d'un octet (ou d'un bit) représente le (N + 1)ème octet (ou bit) du segment ;
- La taille d'un segment représente le rang N du dernier octet de nom N - 1 de ce segment.

5.2.3.1. Cas favorable (mode 1 et 1b)

C'est le premier cas : l'information T, i, c se trouve en mémoire associative.

Les signaux présentés à l'entrée du T.A.L. représentent les informations suivantes :

- L'adresse lexicale T, i, c, d ;
- La longueur de l'objet à accéder l ;
- La valeur du niveau lexical courant i ;
- Les commandes.

1. La recherche en mémoire associative se fait sur T, i, c, pendant que la valeur de i est comparée avec celle de ic ;

2. Cette information (T, i, c) s'y trouvant, le descripteur associé est rangé dans le registre de sortie de la mémoire associative ;

3. Un premier traitement sur ce descripteur permet de reconnaître et d'exploiter chaque champ séparément (ce traitement sera détaillé par la suite)

   - Le champ "type" sert à effectuer certains contrôles relatifs aux droits associés au SV (ou SL) où se trouve l'objet à accéder ;
   - Le champ "format" indique le découpage "adresse-taille" du descripteur ;
   - Le champ "adresse" du SV (ou du SL) désigne l'adresse virtuelle appelée b du 1er octet du SV (ou du SL) ;
   - Le champ "taille" du SV (ou du SL) désigne la taille du SV (ou du SL) de l'objet à accéder ; le champ est dénommé t ;

Il faut, à ce stade de la description, distinguer deux cas : celui où le déplacement d et la longueur l sont exprimés en octet et celui où ils sont exprimés en bit.

a) d et l en octet (mode 1)

C'est le cas le plus simple et le plus courant.

4. Une première opération consiste en l'addition d + l qui permet de distinguer l'octet succédant au dernier octet significatif de l'objet à accéder ;

5. L'opération suivante se justifie logiquement : c'est la comparaison $d + l \leqslant t$ qui contrôle le non dépassement hors du segment ;

   Si ce test est négatif, le T.A.L. interrompt son activité et prévient l'Unité de Commande Principale qui traite les exceptions ;

6. L'adresse virtuelle du 1er octet à accéder est obtenue par l'addition b + d.

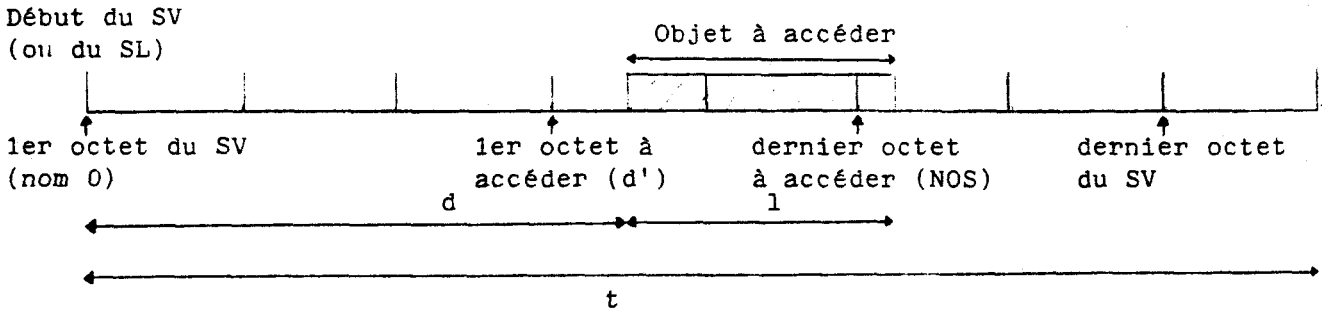
Si les tests effectués sur le "type" (champ du descripteur) se sont révélés positifs, le T.A.L. présente alors en sortie :

- L'adresse virtuelle du 1er octet à accéder en mémoire ;
- Le nombre d'octets à extraire n = l ;
- Les commandes destinées à l'Unité de Commande Principale et au topographe.

b) d et l en bit (mode lb)

Le premier problème rencontré provient du fait que la taille du segment (SV ou SL) est exprimée en octet et que la comparaison avec le nom de l'octet successeur du dernier octet significatif de l'objet à accéder oblige à déterminer ce dernier par des opérations préalables.

La première opération est donc de déterminer le nom d'un octet à l'aide des informations présentes dans le T.A.L. : d et l.



4. L'addition d + 1 donne une grandeur qui est le nom dans le segment (SV ou SL) du bit successeur du dernier bit significatif de l'objet à accéder ;

5. L'opération (d + 1)/8, qui correspond en fait à éliminer les 3 bits de poids faible permet d'obtenir une valeur traitée comme suit :

   - Si (d + 1) mod 8 = 0, cette valeur représente exactement le nom de l'octet successeur au dernier octet significatif de l'objet à accéder que l'on désigne par NOS (Nom de l'Octet Successeur) ;

   - Sinon, cette valeur doit être incrémentée de 1 pour obtenir ce même NOS ;

6. La comparaison NOS $\leqslant$ t peut alors se faire ; comme dans le mode 1, si ce test est négatif, le T.A.L. interrompt son activité et prévient l'Unité de Commande Principale ;

7. L'adresse virtuelle du 1er octet à accéder en mémoire est obtenue par addition d' + b, où d' est égal à d/8, ce qui représente le nom du 1er octet où se trouve le premier bit significatif de l'objet à accéder dans le segment ;

8. Afin d'obtenir le nombre d'octets à extraire, il faut effectuer l'opération suivante NOS - d' qui définit exactement n.

Si les tests effectués conformément au champ "type" extrait du descripteur recueilli dans le registre de sortie de la mémoire associative se révèlent positifs, le T.A.L. présente en sortie :

- L'adresse virtuelle du 1er octet à extraire en mémoire ;
- Le nombre d'octet à extraire n ;
- L'information complémentaire permettant le repérage de l'objet accédé dans la suite d'octets extraits : d" = d mod 8 ;
- Les commandes destinées à l'Unité de Commande Principale et au topographe.

5.2.3.2. Cas partiellement favorable (mode 2)

C'est le 2ème cas : l'information T, i, c ne se trouve pas en mémoire associative mais T, i s'y trouve.

Les signaux présentés à l'entrée du T.A.L. représentent les informations suivantes :

- L'adresse lexicale T, i, c, d ;
- La longueur de l'objet à accéder l ;
- La valeur du niveau lexical courant i ;
- Les commandes.

1. La recherche s'effectue en mémoire associative pendant que la valeur de i est comparée avec celle de ic.

   Nous sommes dans l'hypothèse où cette recherche s'avère infructueuse ; l'opération suivante est donc une reconfiguration de l'information à présenter à la mémoire associative :

   $$T, i, c \longrightarrow T, i$$

2. Une deuxième recherche s'effectue donc, qui permet le chargement du descripteur associé dans le registre de sortie de la mémoire associative ;

3. Un même type de traitement que dans le 1er cas permet de séparer les différents champs de ce descripteur, qui a le format des descripteurs contenus dans le TNL puisqu'il désigne le SLCL où se trouve la catégorie c désignée dans l'adresse lexicale :

- Le champ "type" est celui relatif avec SLCL ;
- Le champ format indique le découpage "adresse - taille - Cmax" ;
- Le champ "adresse" indique l'adresse virtuelle du 1er octet du SLCL que l'on appelle b ;

- Le champ "taille - Cmax", qui occupe dans le descripteur la place habituellement prise par "taille" uniquement, indique d'une part, la taille du SLCL (t) et d'autre part la valeur maximale pouvant être atteinte par le nom d'une catégorie dans ce SLCL ;

4. Un contrôle s'effectue en comparant la valeur de c (de l'adresse lexicale) avec celle de Cmax extraite du descripteur :

$$c \leqslant cmax \ ?$$

Si ce test est négatif, le T.A.L. interrompt son activité et prévient l'Unité de Commande Principale.

5. L'opération suivante consiste à calculer le déplacement dp par rapport au début du SLCL qui permet de repérer le descripteur de la catégorie c :

c * 8 = dp où c * 8 s'explique par le fait qu'un descripteur est défini sur 8 octets et que les catégories se trouvent en début de SLCL ;

6. L'adresse virtuelle du 1er octet à extraire en mémoire est obtenue en effectuant l'addition b + dp.

Si les tests effectués sur le "type" du descripteur se sont révélés positifs, le T.A.L. présente alors en sortie :

- L'adresse virtuelle du 1er octet à extraire en mémoire ;
- Le nombre d'octets à extraire n = 8 (l'objet à accéder est un descripteur de longueur fixe) ;
- Les commandes destinées à l'Unité de Commande Principale et au topographe.

5.2.3.3. Cas défavorable (mode 3)

C'est le 3ème cas : les informations T,i,c et T, i ne se trouvent pas en mémoire associative.

Les signaux présentés à l'entrée du T.A.L. représentent les informations suivantes :

- L'adresse lexicale T, i, c, d ;
- La longueur de l'objet à accéder l ;
- La valeur du niveau lexical courant i ;
- Les commandes.

1. Pendant que i est comparé à ic, une première recherche s'effectue dans la mémoire associative pour T, i, c ;

2. La première recherche s'avérant infructueuse, une deuxième recherche portant sur T, i s'effectue dans la mémoire associative ;

3. Cette deuxième recherche se soldant par un échec, le T.A.L. prévient l'Unité de Commande Principale.

Le processus d'accès à l'objet demandé devient alors très long car il faut dans un premier temps accéder au BCP, puis à la TNL pour enfin lire le descripteur du SLCL se trouvant au niveau i.

Il semble donc utile de posséder un registre, non nécessairement interne au T.A.L., qui contiendrait en permanence le descripteur de la TNL qui serait du type descripteur du SL :

Type, Format, Adresse, Taille
(b) ($\overset{\bullet}{t}$ = imax + 8) où imax représente le niveau lexical maximal associé à la TNL fixé par le système.

Le chargement de ce descripteur dans le registre de sortie de la mémoire associative permettrait au T.A.L. d'effectuer les opérations suivantes :

1. Traitement permettant le découpage pour la reconnaissance et l'exploitation des différents champs du descripteur ;

2. Contrôles liés au "type" du descripteur ;

3. Obtention de l'adresse virtuelle du 1er octet à extraire correspondant au premier octet significatif du descripteur se trouvant au niveau i de la TNL :

b + i * 8 (chaque descripteur est sur 8 octets)

4. Contrôle de non dépassement de la TNL : $i * 8 \leqslant t$?

Dans le cas où les contrôles se révèlent positifs, le T.A.L. délivre en sortie les informations suivantes :

- Une adresse virtuelle ;
- Le nombre d'octets à extraire en mémoire n = 8 (taille fixe des descripteurs) ;
- Les commandes destinées à l'Unité de Commande Principale et au topographe.

5.2.4. Utilisation du T.A.L. dans les divers processus d'accès à un objet

L'étude des 3 cas, qui, comme les mousquetaires sont 4 en réalité, va permettre de définir simplement les diverses étapes de fonctionnement du T.A.L. nécessaires à l'obtention de l'adresse virtuelle de l'objet à accéder, et celà, dans tous les cas de figure.

a) L'information T, i, c se trouve en mémoire associative

L'obtention de l'adresse virtuelle de l'objet à accéder se fait en un seul passage dans le T.A.L. en mode 1 ou 1b (ces modes correspondent au 1er cas étudié en octet ou en bit).

L'accès effectif est ensuite à la charge du topographe puis de l'Unité de Gestion de la Mémoire.

b) L'information T, i, c ne se trouve pas en mémoire associative mais l'information T, i s'y trouve

L'obtention de l'adresse virtuelle de l'objet à accéder se fait en 3 étapes :

- Obtention de l'adresse virtuelle du descripteur du SLCL contenu dans la catégorie de rang c (mode 2) ;

- Accès au descripteur réalisé par l'Unité de Gestion de la Mémoire (après passage dans le topographe) ;

- Chargement de ce descripteur et des champs associés T, i, c dans la mémoire associative et simultanément évaluation de l'adresse virtuelle de l'objet à accéder suivant le mode 1 ou 1b.

L'accès effectif à l'objet revient ensuite à l'Unité de Gestion de la Mémoire (après passage de l'adresse virtuelle dans le topographe).

c) Ni l'information T, i, c ni T, i ne se trouvent en mémoire associative

L'obtention de l'adresse virtuelle de l'objet à accéder se fait en 5 étapes :

- Obtention de l'adresse virtuelle du descripteur de SLCL contenu au niveau i de la TNL (mode 3) ;

- Accès à ce descripteur réalisé par l'Unité de Gestion de la Mémoire (après passage dans le topographe) ;

- Chargement de ce descripteur et des champs associés T, i dans la mémoire associative et simultanément évaluation de l'adresse virtuelle du descripteur du SL où se trouve l'objet à accéder suivant le mode 2 ;

- Accès à ce descripteur réalisé par l'Unité de Gestion de la Mémoire (après passage dans le topographe) ;

- Chargement de ce descripteur et des champs associés T, i, c dans la mémoire associative et simultanément, évaluation de l'adresse virtuelle de l'objet à accéder suivant le mode 1 ou 1b.

L'accès effectif à l'objet revient ensuite à l'Unité de Gestion de la Mémoire (après traduction de l'adresse virtuelle par le topographe).

5.3. SCHEMA D'IMPLEMENTATION PROPOSE

5.3.1. Fonction, principes et limites du schéma proposé

L'état actuel des études sur l'architecture de la machine langage ADA, reposant sur l'analyse du décodage des instructions du code machine, fait apparaître le besoin d'un accès rapide aux objets désignés par les opérandes.

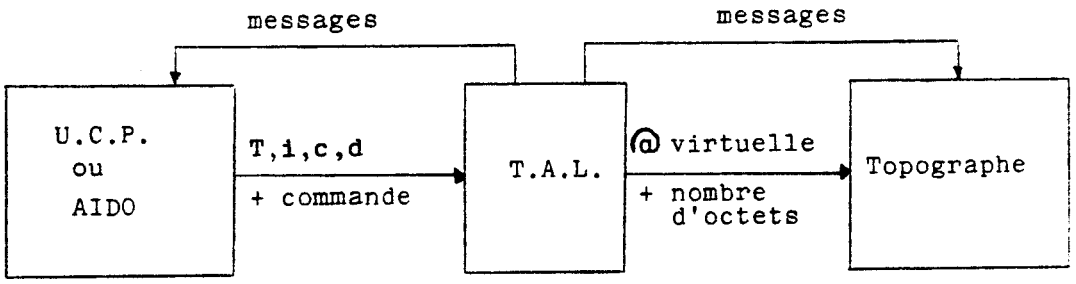
Cette nécessité conduit en particulier à réaliser les transformations d'adresses lexicales en adresses virtuelles de la manière la plus rapide possible.

La description fonctionnelle de l'ensemble des opérations découlant de ces transformations a permis de mettre en évidence la relative complexité de cet ensemble.

Cette constatation a conforté le besoin d'étudier au préalable un outil matériel dont l'estimation des performances validerait a priori l'aspect adressage lexical de la machine.

L'objectif présent est de réaliser un module matériel spécifique, une sorte de "boîte noire", que l'on pourra intégrer dans n'importe quel schéma global d'architecture.

Les seuls modules matériels visibles du T.A.L. sont l'Unité de Commande Principale (ou AIDO si celui-ci joue ce rôle) et le topographe :



Le schéma d'implémentation est le résultat d'une approche demeurée très fonctionnelle dans le choix de l'architecture comme celui de la définition des différents éléments matériels.

La conception de ce schéma s'inspire pourtant de deux principes parfois contradictoires :

- Réduire le nombre d'opérateurs en augmentant leur capacité d'utilisation (grâce, par exemple, à l'emploi de multiplexeurs) ;

- Accroître l'efficacité en permettant un parallélisme d'exécution maximal (utilisation de deux additionneurs par exemple).

Le schéma d'implémentation, tel qu'il est proposé, n'est évidemment que la première étape d'une réalisation véritablement matérielle.

La non définition de l'Unité de Commande du T.A.L. ne permet que la description d'un synchronisation rudimentaire mais largement satisfaisante pour l'estimation des performances effectuées à ce niveau.

Les schémas logiques explicités sont essentiellement fonctionnels et ne prennent pas en compte les optimisations possibles (emploi de NAND et de NOR).

D'une façon générale, aucune référence n'est faite à un catalogue particulier de composants.

Le choix de la technologie est évoqué dans le paragraphe 5.4 pour justifier les temps utilisés pour les évaluations.

La présentation du schéma d'implémentation se décompose en deux parties :

- Un dessin synoptique visualisant l'ensemble des modules matériels format le T.A.L. ;

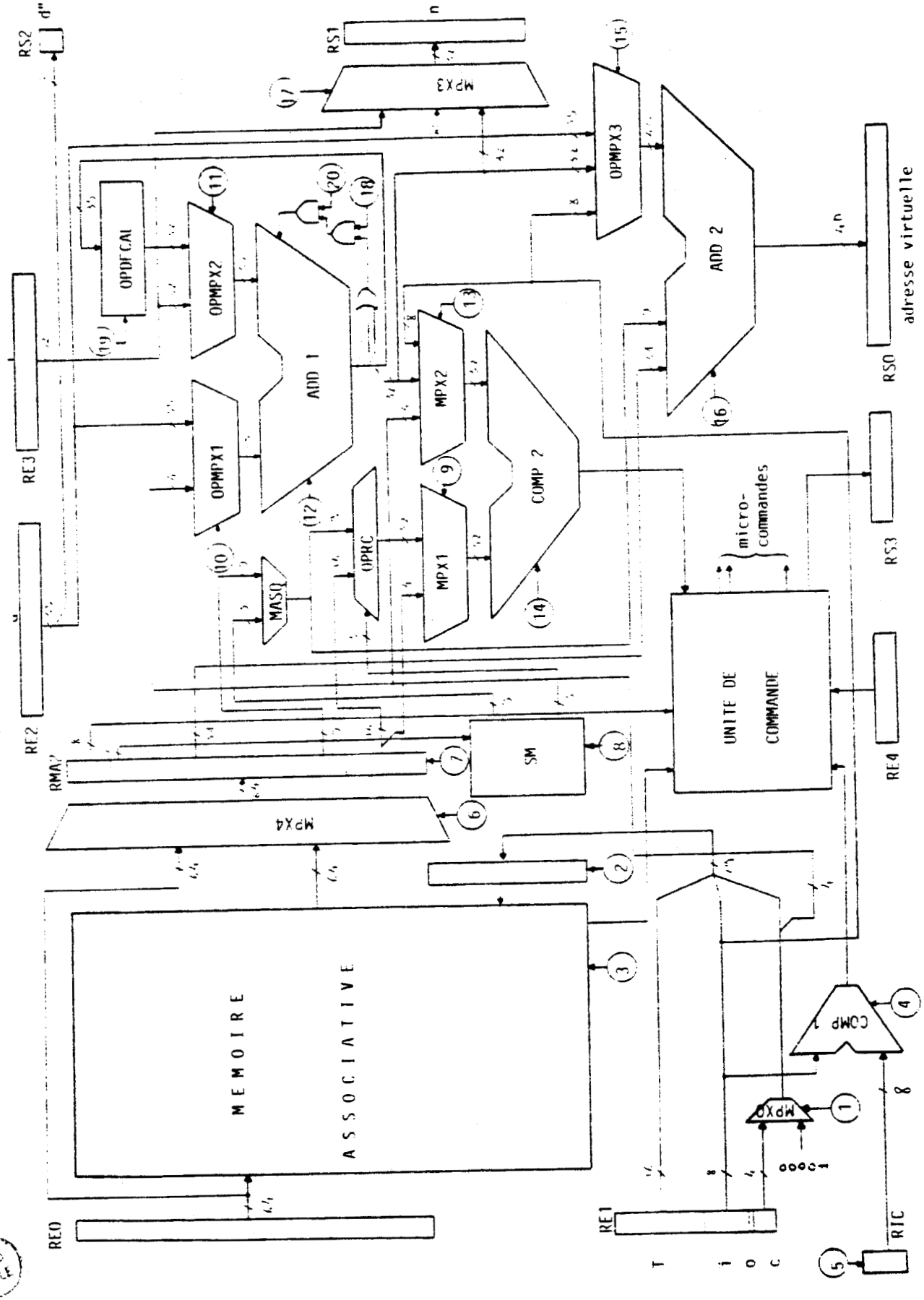


Schéma synoptique du T.A.L.

- La description plus détaillée de chaque groupe d'éléments associé à une fonctionnalité du T.A.L. en précisant pour chaque élément, le nombre de portes utilisées et le nombre de portes à traverser pour le chemin de donnée le plus long à l'intérieur de cet élément (but : calcul du temps de validation des signaux en sortie).

5.3.2. Description logique des différents modules

Afin de clarifier leur présentation, les modules matériels sont rassemblés au sein de groupes que l'on peut associer à une fonctionnalité précise du T.A.L.

La connaissance du rôle attribué à chacun de ces groupes permettra ainsi de mieux les intégrer au schéma d'ensemble et de faciliter la compréhension du fonctionnement global de celui-ci.

5.3.2.1. Les registres en entrée et en sortie

Tous ces registres sont de simples buffers ne possédant aucune fonction particulière, leur seule utilité est de pouvoir conserver une information pendant une durée définie par l'Unité de Commande.

Registres d'entrée

REO (64 bits) : contient soit un descripteur de segment destiné au changement de la mémoire associative, soit le descripteur de la TNL courante à RMA2 et provenant d'un registre interne à la machine.

RE1 (29 bits) : contient l'information destinée à la recherche associative soit :

- T qui est le numéro de tâche (16 bits)*
- i de l'adresse lexicale (8 bits)
- 1 bit de typage
- c de l'adresse lexicale (4 bits)

* Le nombre de tâches concourantes dans un même système est limité arbitrairement à 2^{16}, ce qui semble (très) largement suffisant.

Le bit de typage est nécessaire pour distinguer une information du type T, i, c (où c = 0) d'une information du type T, i.

RE2 (35 bits) : contient le déplacement d de l'adresse lexicale, exprimé en nom d'octet ou de bit.

RE3 (32 bits) : contient l, le nombre d'octets ou de bits composant l'objet à accéder.

Les valeurs se trouvant simultanément dans RE2 et RE3 sont toujours consistantes (octet ou bit) ; l'indication bit ou octet est donnée par l'Unité de Commande de la T.A.L.

RE4 (longueur non évaluée) : contient le code destiné à l'Unité de Commande du T.A.L. précisant le type d'opérations à effectuer ; ce code n'est pas encore formalisé au stade actuel de la conception de l'architecture.

Registres de sortie :

RS0 (40 bits) : contient l'adresse virtuelle du premier octet à accéder en mémoire.

RS1 (32 bits) : contient n, le nombre d'octets à accéder en mémoire.

RS2 (3 bits) : contient d", l'information nécessaire au cadrage de l'objet à accéder dans le cas où le d est en bit.

Ce registre est purement fonctionnel, d" représentant les 3 bits de poids faible de RE2 quand celui-ci contient une valeur exprimée en bit ; une véritable implémentation matérielle se contenterait d'un câblage adéquat sur RE2 ou d'élargir RS1 à 35 bits pour contenir ces 3 bits supplémentaires.

RS3 (longueur non évaluée) : contient l'ensemble des messages à destination de l'Unité de Commande Principale (ou Décodeur) et du Topographe.

5.3.2.2. Initialisation de la recherche associative et premier contrôle

Les deux premières opérations à la charge du T.A.L. dans le cas d'une transformation d'adresse lexicale vont être :

1. Lancer la recherche associative sur le contenu de RAM1 ;

2. Comparer le contenu de RIC avec les bits de RE1 représentant i.

Le chargement de RAM1 par le contenu de RE1 se fera partiellement par le passage dans un multiplexeur MPX0.

En effet, dans le cas où la recherche associative doit s'effectuer sur T, i, c, il faut masquer c et indiquer par le bit destiné à cet usage que l'information est du type T, i.

RIC est un registre interne à la machine contenant la valeur du niveau lexical courant ic.

COMP1 est un comparateur 8 bits réalisant la comparaison :

$$i \leqslant ic \ ?$$

Le résultat de ce test est communiqué directement à l'Unité de Commande du T.A.L.

RAM2 est un registre de 64 bits qui reçoit les descripteurs de Segment exploités ensuite par le T.A.L.

Ces descripteurs proviennent de deux sources différentes :

- La mémoire associative dans le cas d'une recherche réussie ;

- Le registre RE0 qui contient soit les descripteurs de Segment à charger en mémoire associative, soit les descripteurs de TNL courante.

La sélection entre ces 2 sources est opérée par un multiplexeur MPX4.

a) Description du MULTIPLEXEUR N°O (MPXO)

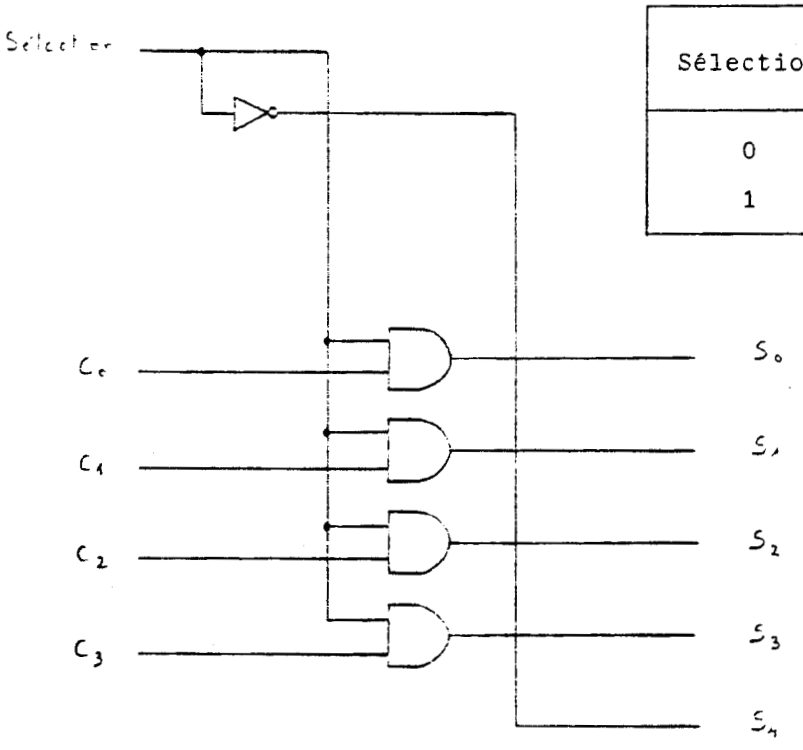
Il s'agit d'un simple opérateur de multiplexage entre 2 types de valeur :

- c plus 1 bit servant à différencier l'information sur laquelle s'effectuer la recherche associative
  (T, i, c où c = 0, est différent de T, i) ;

- La valeur 0001 qui spécifie que l'information sur laquelle s'effectue la recherche associative est du type T, i.

Entrées : - 4 provenant de RE1 et correspondant à c (ci) ;
- 5 en réalité intégrés au matériel ;
- 1 de sélection (k).

Sorties : - 5 à destination de RAM1 (Si).



Sélection	Sortie
0	00001
1	[c]o

Evaluation : Nombre de portes = 5

Chemin critique : 1 porte

5.3.2.3. Masquage et second contrôle

Ce deuxième groupe d'éléments réalise les fonctions qui permettent les opérations suivantes :

- Prise en compte des descripteurs de segment dans RAM2 après sélection dans MPX4 ;

- Identification du champ taille et du champ adresse grâce à SM, MASQ et OPRC ;

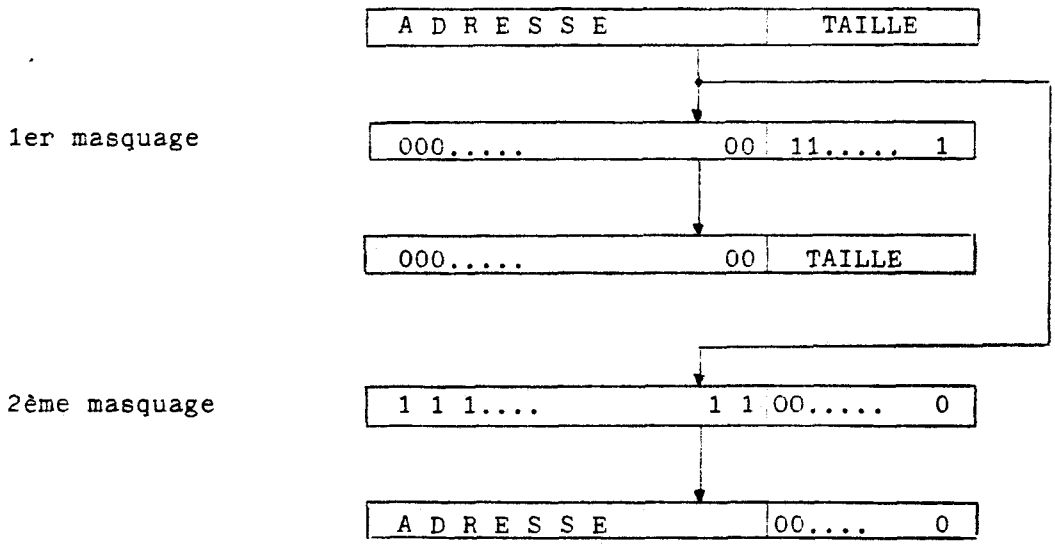
- Contrôles du type d + 1 $\leqslant$ t? et du type c $\leqslant$ cmax? grâce à MPX1, MPX2 et COMP2.

a) Description du SELECTEUR DE MASQUE (SM)

La présence d'un Sélecteur de Masque est due aux différents formats possibles des champs "ADRESSE" et "TAILLE" du descripteur de segment, suivant que ces valeurs sont exprimées en octet, double-mot ou page.

Il existe donc un masque par format possible ; ce masque est inversé pour un deuxième masquage comme cela est expliqué ci-dessous :

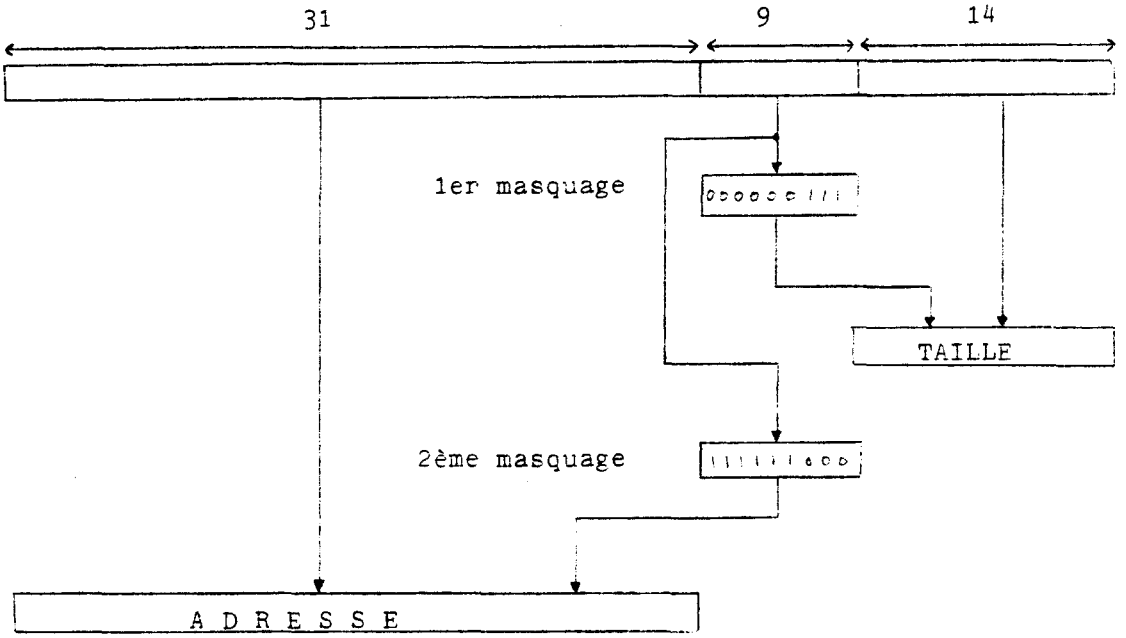
Exemple de configuration :



Les formats sont fixes et par conséquent, les masques associés sont fixés par le matériel.

La sélection du format - 2 bits - permet le choix parmi quatre possibles ; 3 formats sont utilisés actuellement entraînant la création de 3 masques (le 4ème format représente le descripteur nul).

En pratique, les masques n'ont pas besoin d'avoir une longueur de 54 bits puisque la partie critique est limitée aux bits 14 à 22 ; ainsi le masquage se fera de la manière suivante :

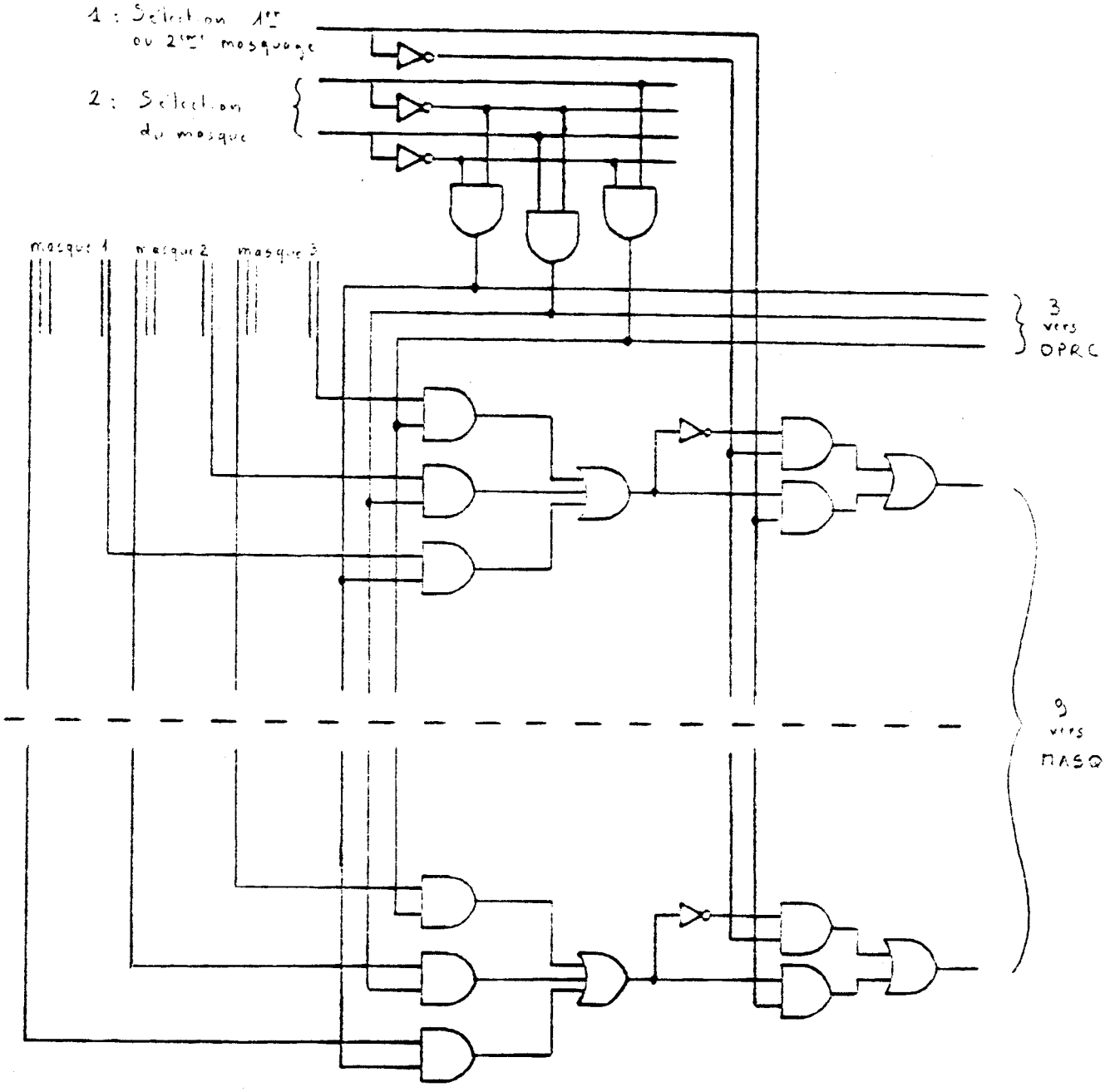


Exemple du cas où l'adresse est sur 37 bits et la taille sur 17

Entrées : - 2 provenant du champ "FORMAT" du descripteur et sélectionnant le masque ;
- 1 indiquant s'il s'agit d'un premier ou d'un second masquage et provenant de l'Unité de Commande du T.A.L.

Sorties : - 3 indiquant directement le format du descripteur à destination de OPRG ;
- 9 formant le masque à destination de MASQ.

SELECTEUR DE MASQUE



Evaluation : Nombre de portes : 6 + 9 x 8 = 78

Chemin critique : - 1er masquage = 6 portes

- 2ème masquage = 3 portes car la sélection du masque reste inchangée.

b) Description de l'OPERATEUR DE MASQUAGE

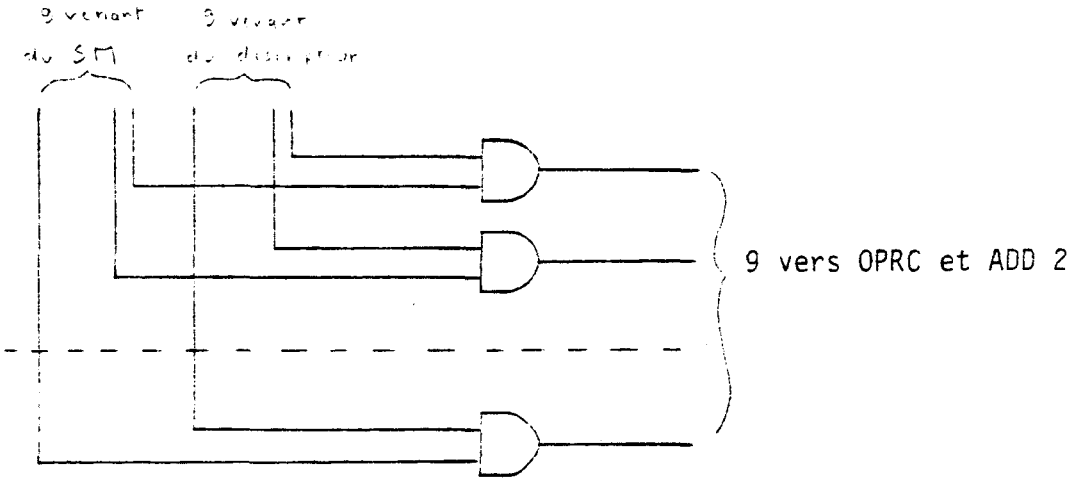
Cet opérateur de masquage est excessivement simple puisqu'il ne fait que réaliser la fonction ET entre l'information reçue du descripteur et le masque émis par le SELECTEUR DE MASQUE.

Les sorties de cet opérateur sont reliées soit à l'OPERATEUR DE RECONDITIONNEMENT (bits 0 - 22), soit à l'additionneur n°2 (bits 14 - 53).

Entrées : - 9 provenant de RMA2 ;
- 9 constituant le masque et provenant du SELECTEUR DE MASQUE.

Sorties : - 9 à destination de OPRC et correspondant au résultat du 1er masquage ;
- 9 à destination d'une entrée de l'additionneur n° 2 et correspondant au résultat du 2ème masquage.

OPERATEUR DE MASQUAGE



Evaluation : Nombre de portes : 9

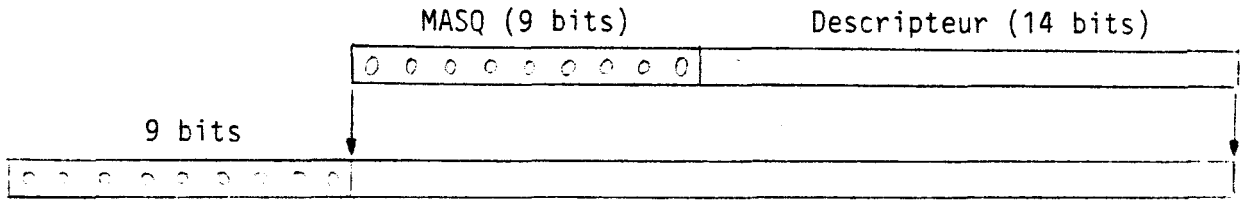
Chemin critique : 1 porte

c) Description de l'OPERATEUR DE RECONDITIONNEMENT (OPRC)

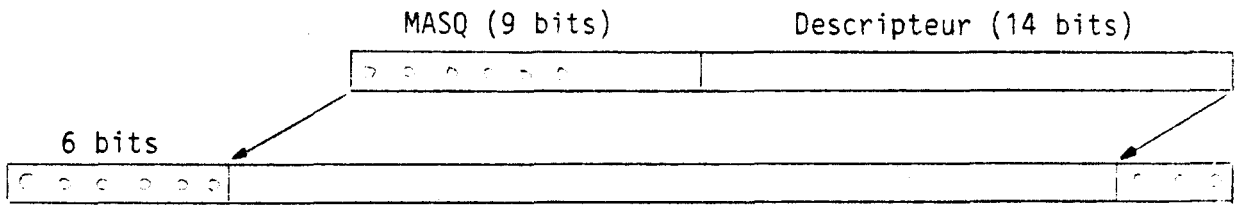
Cet opérateur se justifie par la nécessité de reconfigurer la valeur du champ "TAILLE" qui contient une valeur pouvant être exprimée en octet, double-mot ou page ; or, la comparaison doit toujours s'effectuer par rapport à une grandeur exprimée en octet.

Les 3 cas possibles conduisent aux 3 types de fonctionnement suivants :

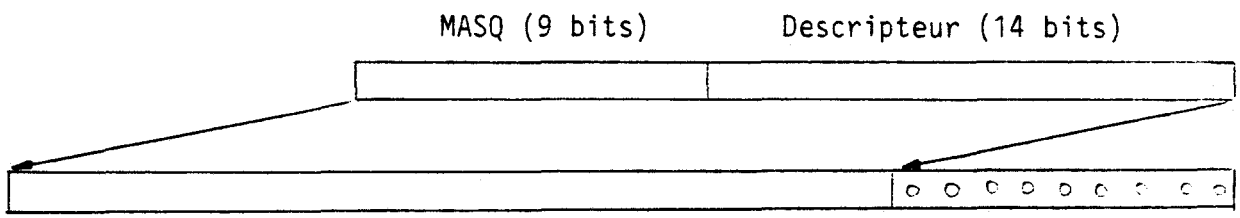
1) en octet : valeur sur 14 bits



2) en double-mot : valeur sur 17 bits



3) en page : valeur sur 23 bits



Cet opérateur ne prend pas en compte le cas où le descripteur est celui d'un SLCL ;

Le champ "TAILLE" est différent mais n'est pas utilisé par le T.A.L. ; le champ "CMAX" est directement présenté au multiplexeur MPX1, cette opération se faisant sans traitement puisque ce champ possède une implantation constante (quelque soit le format) dans le descripteur (il s'agit des 4 bits de poids faible).

Entrées : - 9 résultat du masquage opéré par MASQ ;
- 14 provenant du descripteur et correspondant aux bits de "TAILLE" non masqués ;
- 3 provenant du SM et indiquant le format à considérer (octet, double-mot ou page).

Sorties : - 32 représentant la valeur reconditionnée du champ "TAILLE" à destination du comparateur.

(Voir schéma page suivante).

d) Description du MULTIPLEXEUR N°1 (MPX1)

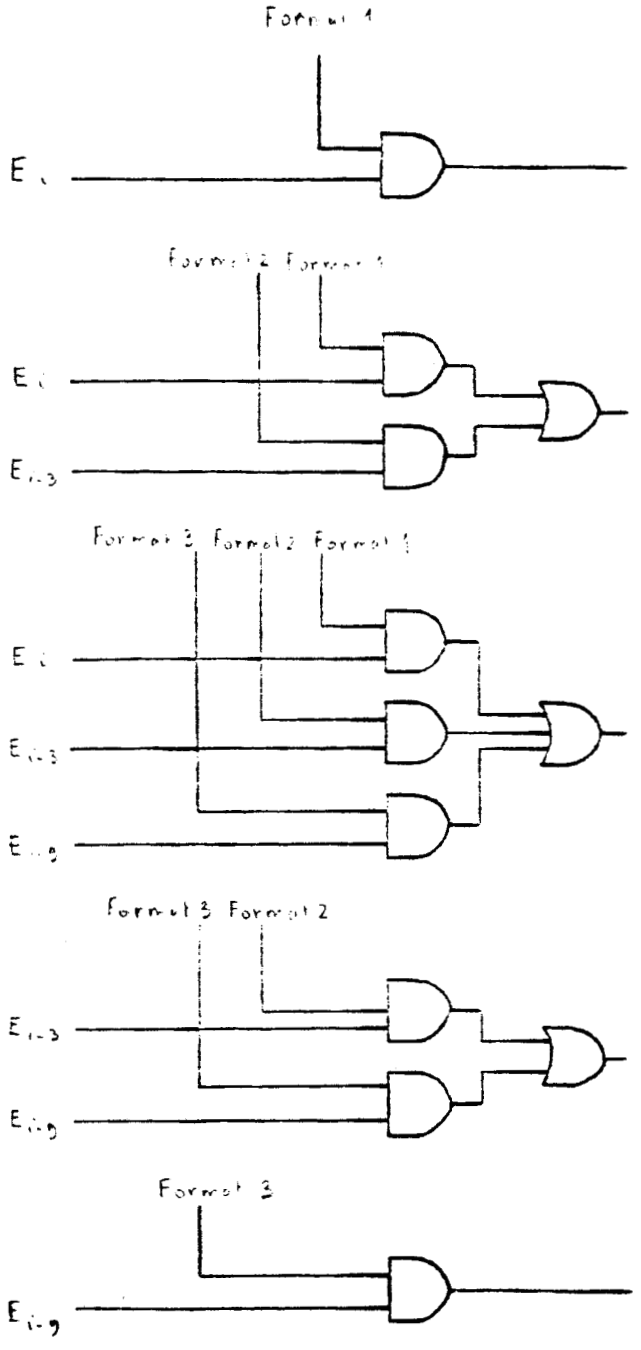
La fonction de cet opérateur est classique : il s'agit de choisir entre deux entrées possibles :

- 32 bits provenant de OPRC ;
- 4 bits provenant de RMA2 auxquels il faut ajouter 28 bits à zéro pour reconditionner cette valeur sur 32 bits;

Entrées : - 32 bits de OPRC et représentant la valeur de la taille du segment ;
- 4 provenant du descripteur et représentant la valeur du "CMAX" ;
- 1 sélectionnant les entrées voulues.

Sorties : - 32 à destination du comparateur COMP2.

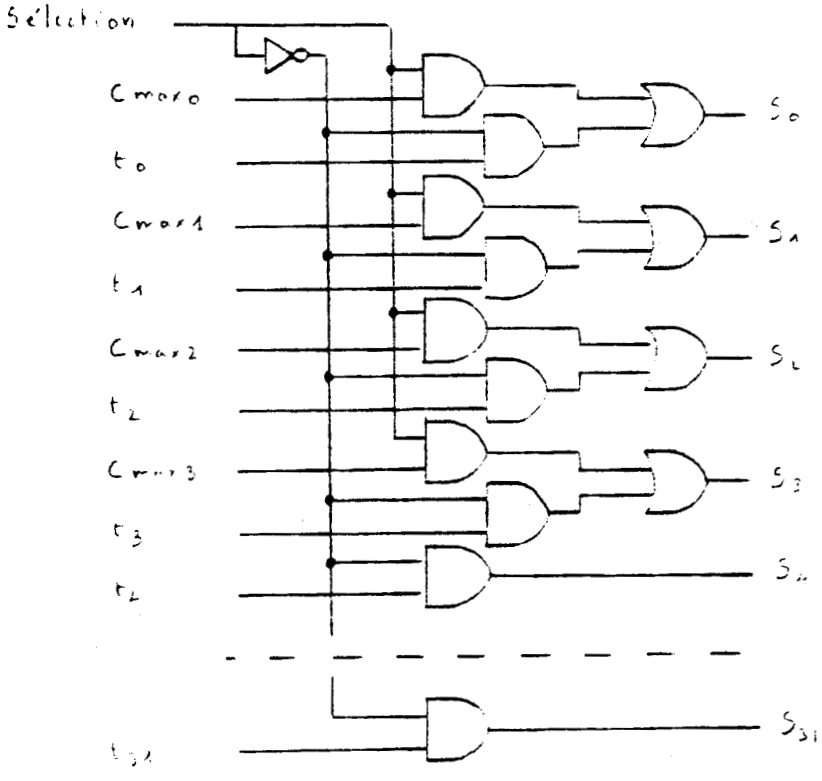
OPERATEUR DE RECONDITIONNEMENT



Evaluation : Nombre de portes = 3 x 1 + 7 x 3 + 14 x 4 + 3 x 3 + 6 x 1 = 95

Chemin critique : 2 portes

MULTIPLEXEUR 1



(ti) représentent les entrées provenant de OPRC
(cmaxi) représentent les entrées provenant de RMA2
(Si) représentent la sortie à destination du COMP2.

Sélection	Sortie
0	t
1	cmax

Evaluation : Nombre de portes = 4 x 3 + 28 + 1 = 41

Chemin critique : 3 portes

e) Description du MULTIPLEXEUR 2 (MPX2)

Cet opérateur assure le multiplexage entre les valeurs suivantes :

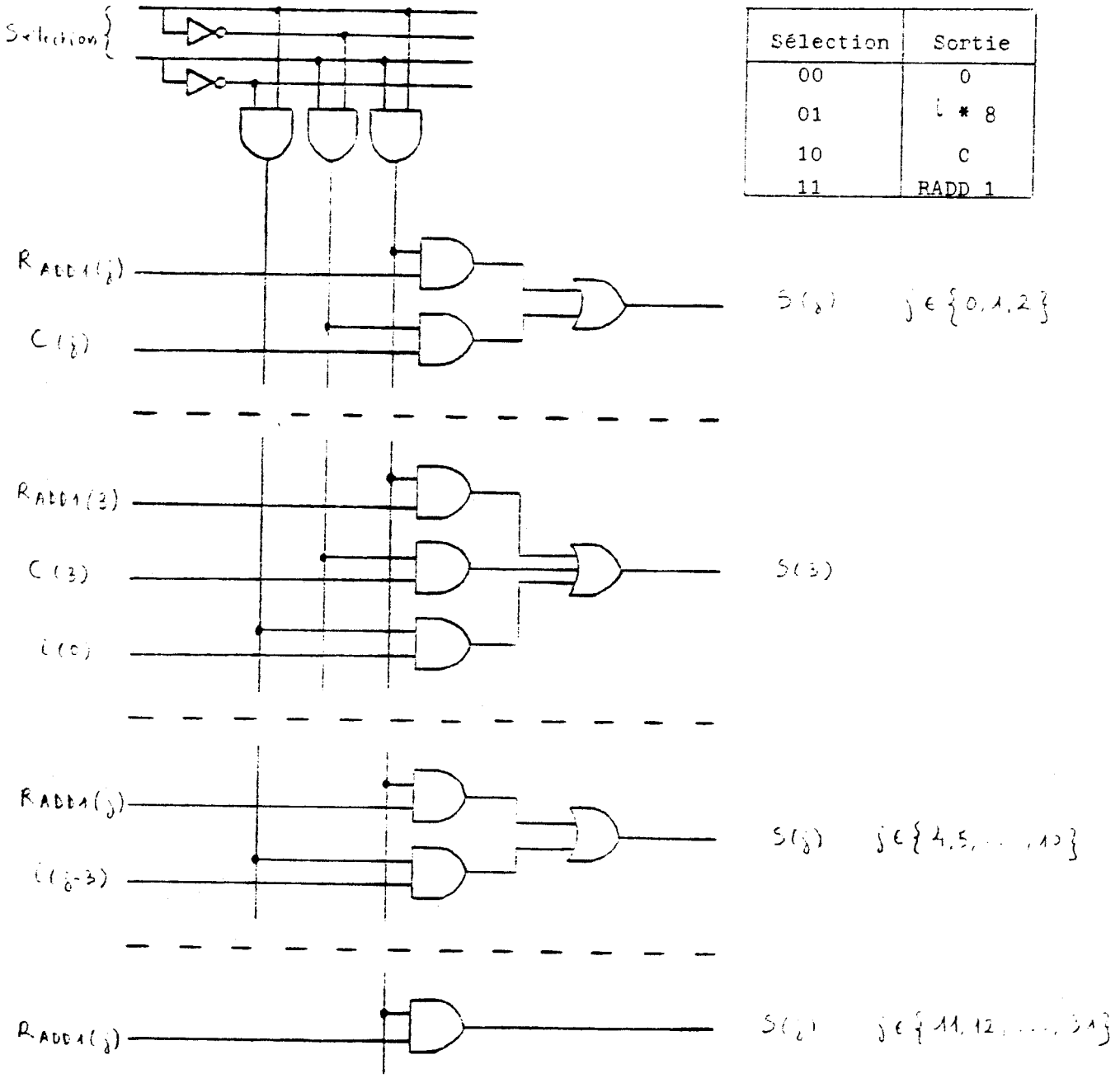
- c issu de l'adresse lexicale fournie à l'entrée du T.A.L. ;
- i* 8 où i issu de l'adresse lexicale ;
- NOS résultat de l'additionneur n°1 dans le mode 1b.

Le décalage assurant le produit i* 8 est effectué par le multiplexeur.

Entrées : - 4 provenant de RE1 et correspondant à c ;
- 32 provenant de l'additionneur n°1 et correspondant à NOS ;
- 8 provenant de RE1 et correspondant à i ;
- 2 sélecteurs.

Sorties : - 32 à destination du comparateur COMP2.

MULTIPLEXEUR 2



Sélection	Sortie
00	0
01	i * 8
10	C
11	RADD 1

Evaluation : Nombre de portes : 5 + 3 * 3 + 1 * 4 + 7 * 3 + 21 = 60

Chemin critique : 4 portes

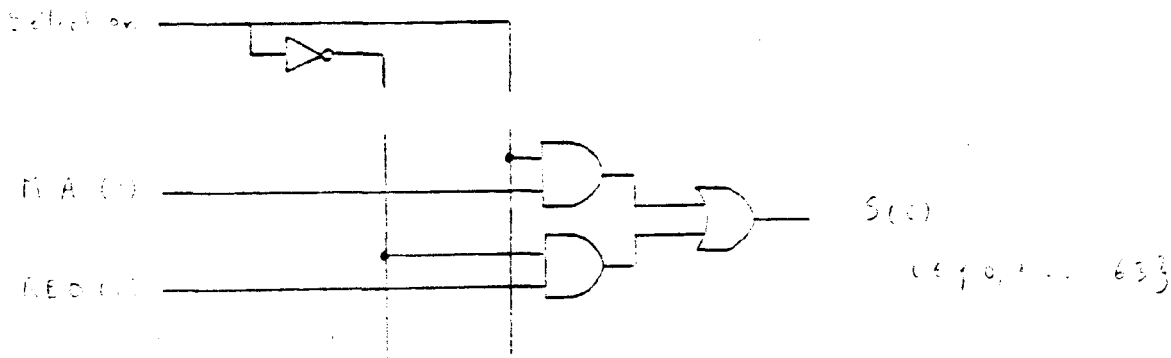
f) Description du MULTIPLEXEUR N°4 (MPX4)

Il s'agit d'un opérateur de multiplexage parfaitement classique.

Entrées : - 64 provenant de la mémoire associative ;
- 64 provenant du registre REO ;
- 1 de sélection.

Sorties : - 64 à destination de RMA2.

MULTIPLEXEUR 4



Evaluation : Nombre de portes = 1 + 64 x 3 = 193

Chemin critique : 3 portes

5.3.2.4. Calculs relatifs aux noms et aux nombres d'octets

Cet ensemble d'opérateurs a pour fonction de :

1. Réaliser les additions d + 1 en octet ou en bit ;
2. Calculer la valeur NOS dans le mode 1b ;
3. Calculer le nombre d'octets à accéder en mémoire (NOS-d') ;
4. Réaliser le décalage correspondant à c * 8 dans le mode 2.

L'utilisation de l'additionneur n°1 est explicité dans le tableau ci-dessous :

Opération	Destination	Bits significatifs à la sortie de ADD1	Sémantique
d + 1	MPX 2	0 - 31	Nom de l'octet successeur du dernier octet significatif de l'objet à accéder (mode 1)
d + 1	OPDECAL	5 - 34	Valeur intermédiaire destinée à calculer NOS dans le mode 1b.
d + 1 ou d + i + 1	OPDECAL et MPX2	0 - 31	Valeur de NOS dans le mode 1b.
NOS + $\overline{d'}$ + 1	OPDECAL	0 - 31	Nombre d'octets à accéder en mémoire dans le mode 1b.
C * 8 + 0	MPX3	0 - 31	Valeur de dp dans le mode 2

a) Description de l'OPERATEUR - MULTIPLEXEUR 1 (OPMPX1)

Cet opérateur doit opérer un multiplexage particulier, en effet, s'il possède 3 types d'entrées - c, d, 0 - il peut fournir 4 sorties possibles : c * 8, d, 0 ou $\overline{d'}$ (pour réaliser l'opération NOS - d').

La soustraction (NOS - d') est ainsi effectuée en faisant la somme de la valeur NOS et du complément à 2 de la valeur d' ; l'opération se fait en passage par l'additionneur : NOS + $\overline{d'}$ + C_y (retenue).

D'autre part, la valeur c doit être multipliée par 8 et nécessite donc un décalage à gauche de 3 positions, avant d'être conditionnée sur 35 bits.

Entrées :

- 4 provenant de RE1 et représentant la valeur de c appartenant à l'adresse lexicale fournie en entrée du T.A.L. ;

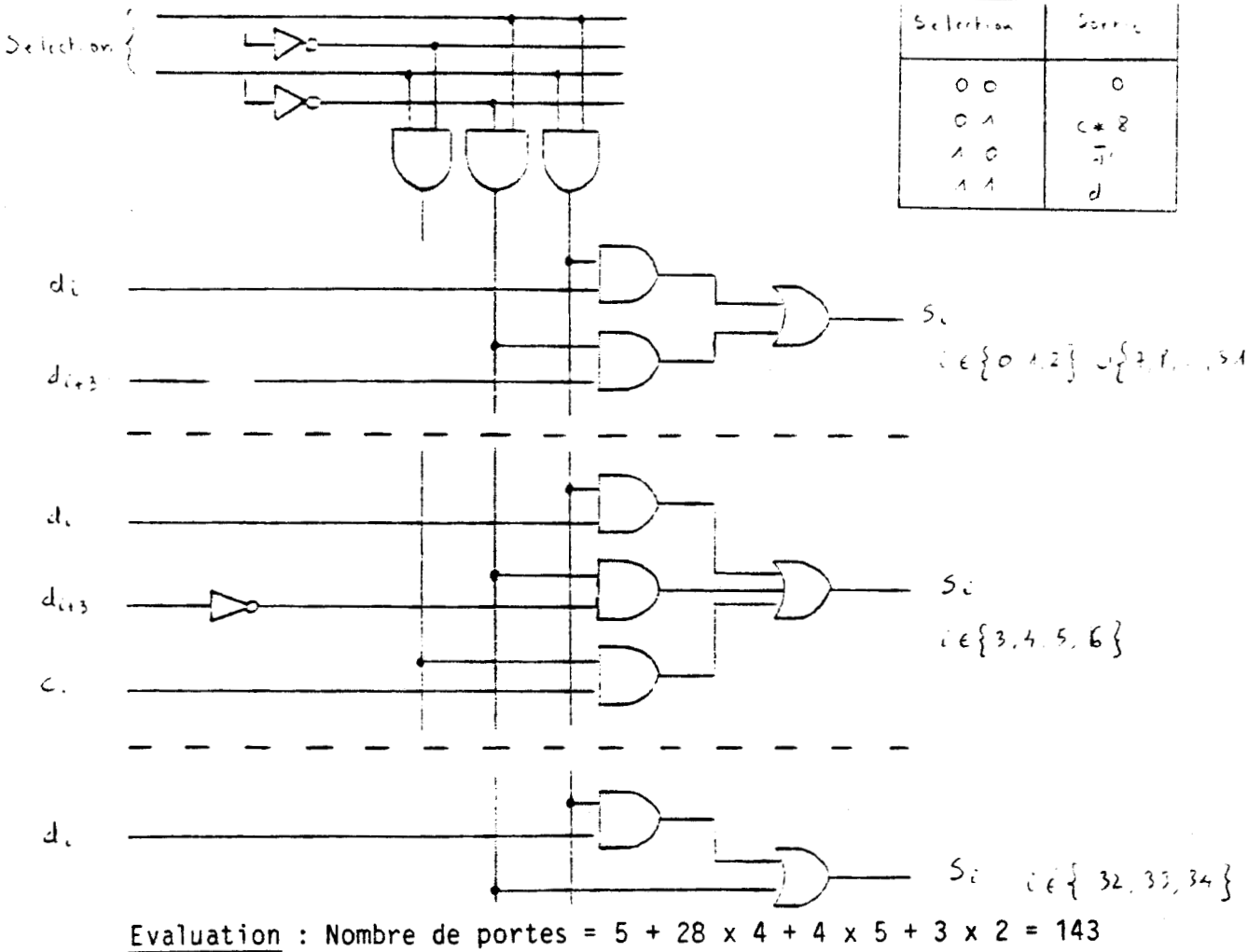
- 35 provenant de RE2 et représentant la valeur de d appartenant à l'adresse lexicale fournie en entrée du T.A.L. ;

- 2 de sélection.

Sorties :

- 35 à destination de l'additionneur ADD1.

OPERATEUR - MULTIPLEXEUR 1



Evaluation : Nombre de portes = 5 + 28 x 4 + 4 x 5 + 3 x 2 = 143

Chemin critique : 4 portes

b) Description de l'OPERATEUR - MULTIPLEXEUR 2 (OPMPX2)

Comme pour OPMPX1, cet opérateur effectue un reconditionnement en même temps qu'un multiplexage entre 2 types de valeur :

- 32 bits pour le résultat de l'opération antérieure contenue dans l'opérateur de décalage OPDECAL ;
- 32 bits pour la valeur de 1

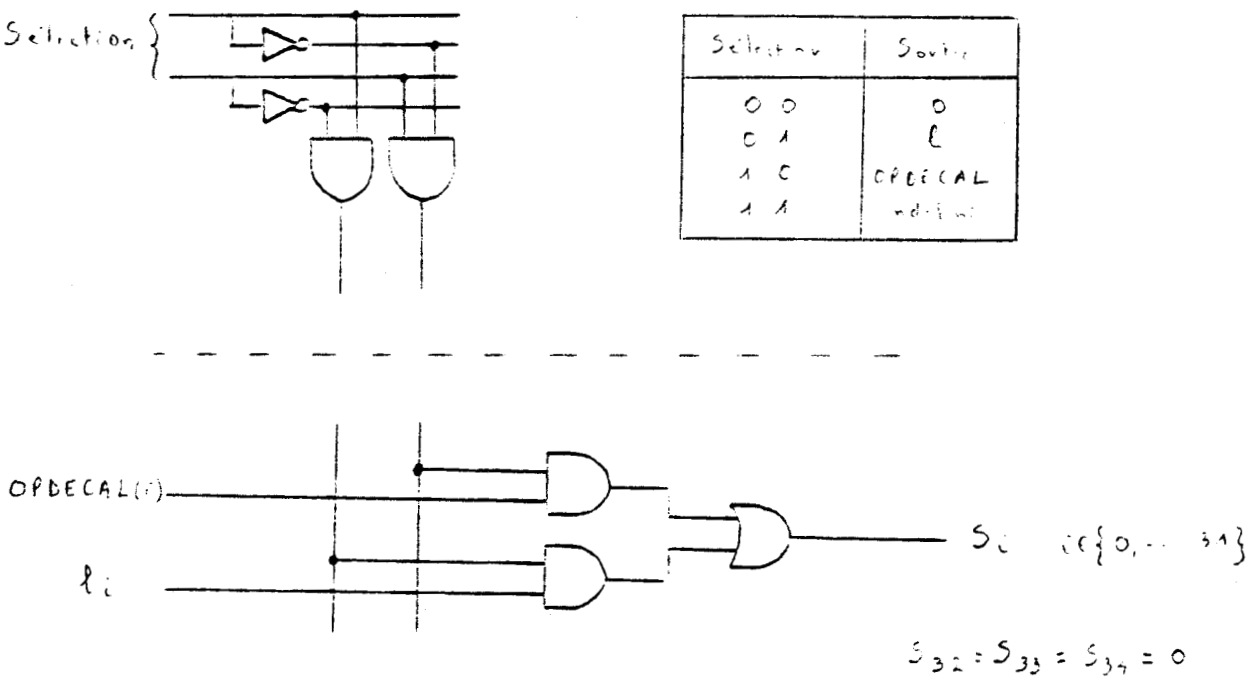
Cet opérateur doit donc conditionner sur 35 bits les valeurs qu'il reçoit.

Entrées :
- 32 bits provenant de OPDECAL ;
- 32 provenant de RE3 ;
- 2 sélecteurs.

Remarque : OPMPX2 doit pouvoir sélectionner la valeur 0 en sortie dans le mode 2.

Sorties :
- 35 à destination de l'additionneur ADD1.

OPERATEUR - MULTIPLEXEUR 2



Evaluation : Nombre de portes = 4 + 32 * 3 = 100

Chemin critique = 4 portes

c) Description de l'OPERATEUR DE DECALAGE (OPDECAL)

Cet opérateur reçoit une information sur 35 bits dont il valide soit les 32 bits de poids faible (0 - 31), soit les 32 bits de poids fort (3 - 34).

Il joue le rôle de l'accumulateur que nécessite toute série d'opérations utilisant un résultat intermédiaire.

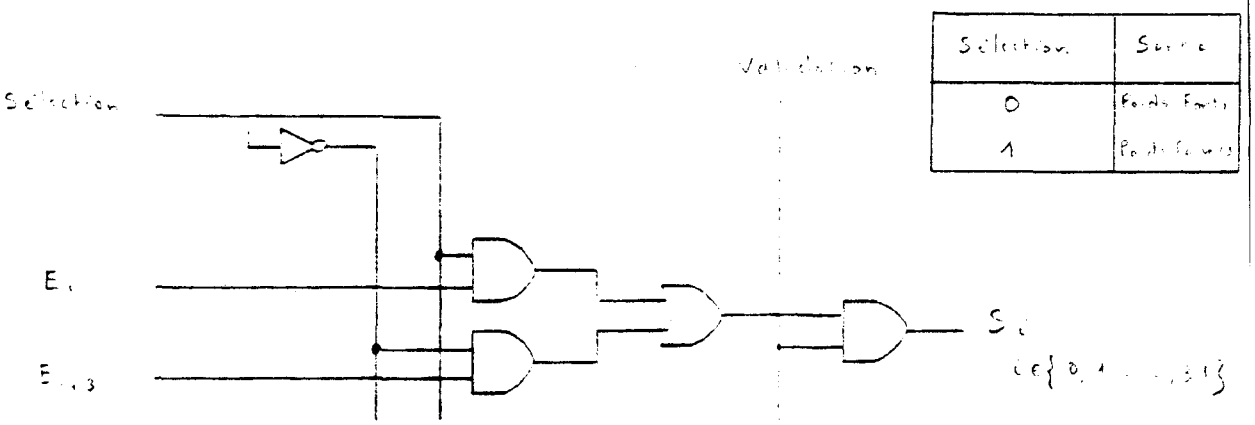
Entrées :

- 35 provenant de l'additionneur ADD1 ;
- 1 de sélection ;
- 1 de validation des signaux de sortie.

Sorties :

- 32 à destination de OPMPX2.

OPERATEUR DE DECALAGE



Evaluation : Nombre de portes + 1 + 32 * 4 = 129

Chemin critique = 3 portes + 1 porte de validation

5.3.2.5. Calcul de l'adresse virtuelle et changement du nombre d'octets n

L'adresse virtuelle fournie par le T.A.L. dans RSO peut représenter 3 types d'objet différents :

- Dans les modes 1 et 1b, il s'agit du nom du 1er octet significatif de l'objet à accéder ;
- Dans le mode 2, il s'agit du nom du 1er octet de la catégorie c du SLCL pointé par le descripteur du niveau i de la TNL courante ;
- Dans le mode 3, il s'agit du nom du 1er octet du descripteur se trouvant au niveau i de la TNL courante, située dans le BCP pointé par un descripteur se trouvant dans un registre interne de la machine.

Si b désigne l'adresse du segment où se situe l'information à accéder, les diverses opérations effectuées par ADD2 sont :

- b + d dans le mode 1
- b + d' dans le mode 1b
- b + dp dans le mode 2 (dp = c * 8)
- b + i * 8 dans le mode 3

OPMPX3 est le multiplexeur qui sélectionne pour l'entrée droite de ADD2 une des valeurs de l'ensemble d, d', dp, i * 8.

L'autre entrée de ADD2 est alimentée par l'adresse b obtenue grâce au 2ème masquage effectué dans MASQ.

Contrairement à ADD1, ADD2 n'utilise pas de retenue ; le nombre d'octets chargé dans RS1 a également diverses sources :

- 1 (RE3) dans le mode 1 ;
- NOS - d' (ADD1) dans le mode 1b ;
- K = 8 dans les modes 2 et 3.

MPX3 est le multiplexeur qui se charge de la sélection.

a) Description de l'OPERATEUR MULTIPLEXEUR N°3

Cet opérateur a pour fonction de multiplexer les valeurs suivantes :

- d issu de l'adresse lexicale fournie à l'entrée du T.A.L. ;
- dp résultat de l'opération c * 8 ;
- i * 8 où i est le niveau lexical issu de l'adresse lexicale à transformer par le T.A.L. ;
- d' qui représentent les 32 bits de poids fort de d (d' = d/8).

Le décalage de la valeur de i correspondant à la multiplication par 8 est assuré par le OPMPX3 de même que la prise en compte de d'.

Etant donné l'ensemble des valeurs que peut prendre dp seuls les 7 bits de poids faible sont significatifs.

Entrées :

- 35 provenant de RE2 et correspondant à d ;
- 7 provenant de l'additionneur n°1 et correspondant à dp ;
- 8 provenant de RE1 et correspondant à i ;
- 2 sélecteurs.

Sorties :

- 40 à destination de l'additionneur n°2.

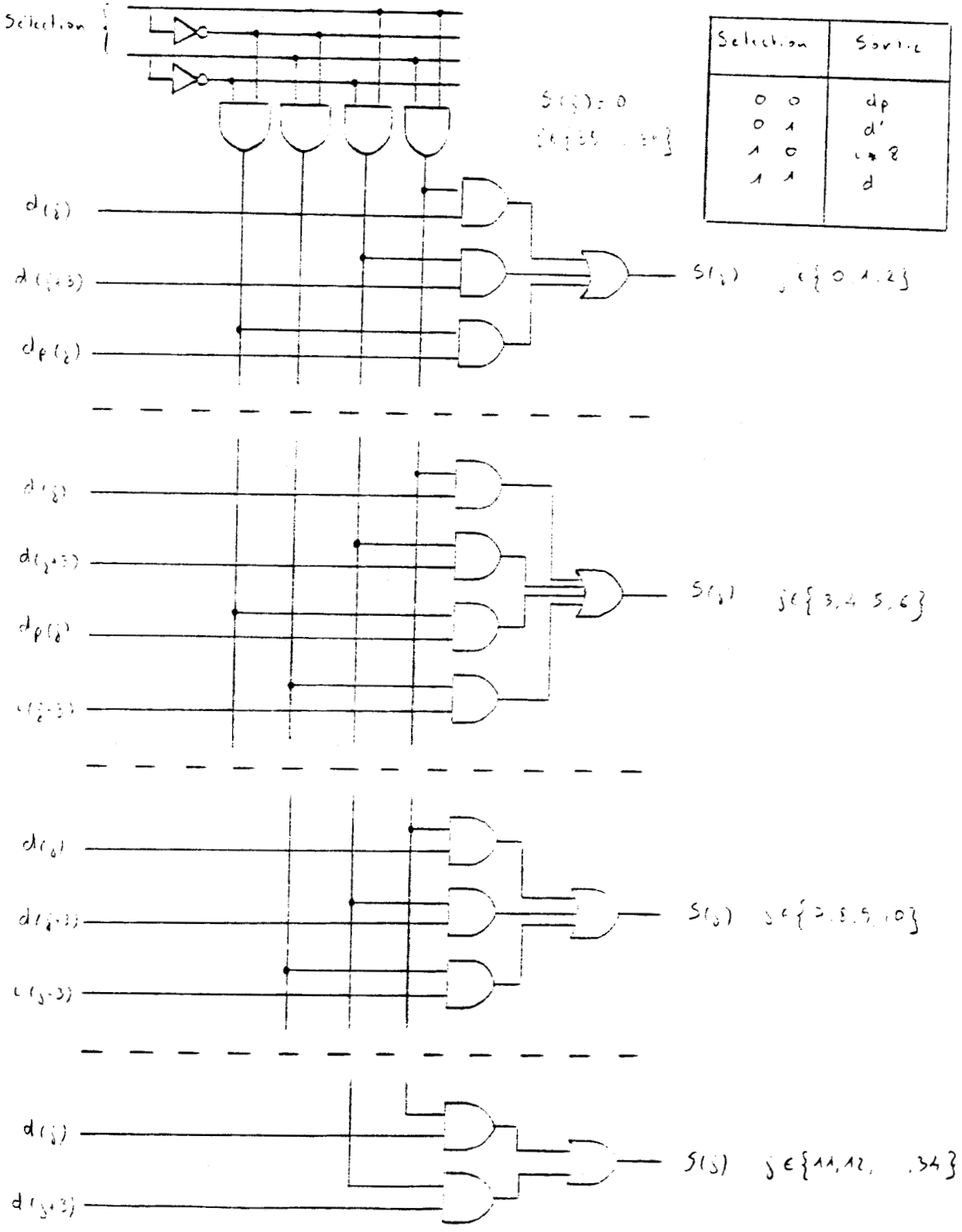
Précision concernant le schéma logique :

- Les sorties S (35) à S (39) ne sont pas représentées car elles sont constamment à 0.

Evaluation : Nombre de portes = 5 + 3 * 4 + 4 * 5 + 4 * 4 + 24 * 3 = 125

Chemin critique : 4 Portes

OPERATEUR MULTIPLEXEUR 3



Sélection	Sortie
0 0	dp
0 1	d'
1 0	i * 8
1 1	d

b) Description du MULTIPLEXEUR N° 3 (MPX3)

Cet opérateur a une fonction classique de multiplexage entre 3 valeurs possibles du nombre d'octets à accéder en mémoire :

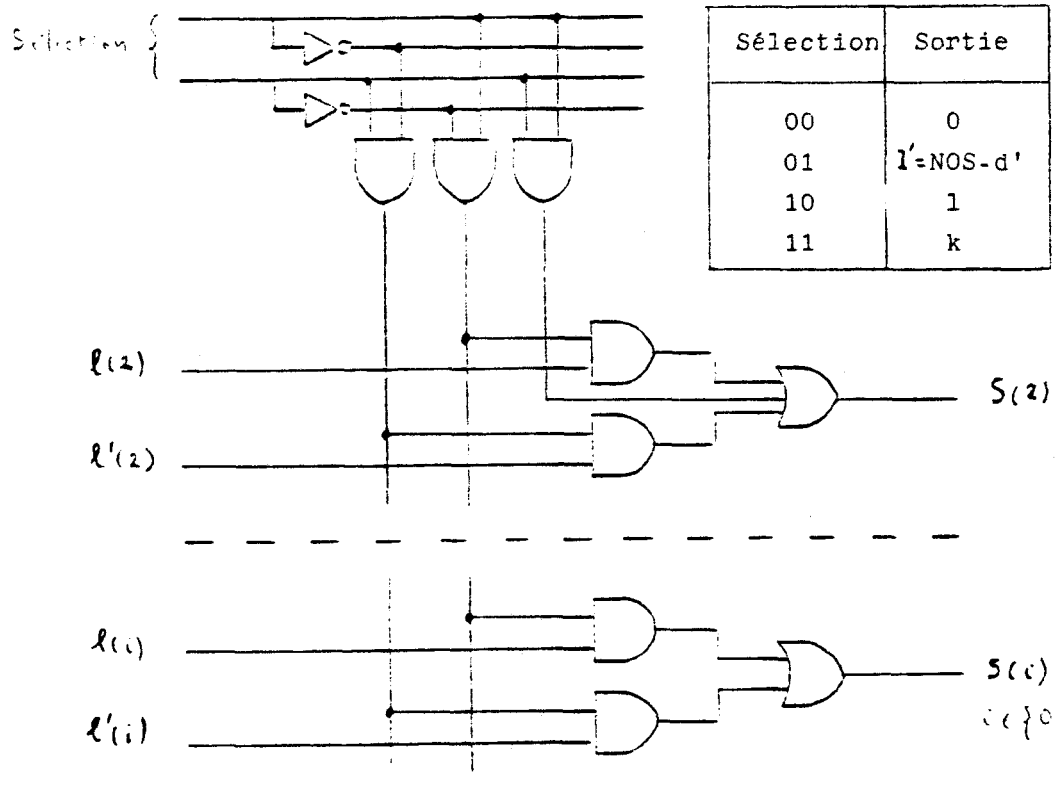
- l, nombre d'octets fourni à l'entrée du T.A.L. (mode 1) ;
- NOS - d', résultat d'opérations sur ADD1 (mode 1b) ;
- K, longueur d'un descripteur de segment (mode 2 et 3).

Il faut noter que K est une constante égale à 8 (la longueur d'un descripteur est toujours de 8 octets) ; cette constatation permet de représenter la valeur de K' que par 1 seul bit positionné à 1, intégré dans le matériel.

Entrées :
- 32 provenant de RE3 et correspondant à 1 .
- 32 provenant de la sortie de ADD1 et correspondant à NOS - d' ;
- 2 de sélection.

Sorties :
- 32 à destination de RS1.

MULTIPLEXEUR 3



Sélection	Sortie
00	0
01	l'=NOS-d'
10	1
11	k

Evaluation : Nombre de portes = 5 + 32 x 3 = 101
Chemin critique : 4 portes

5.3.2.6. La mémoire associative

Cette étude, effectuée en collaboration avec Agnès BRADIER, est une première approche de réalisation d'une mémoire pseudo-associative offrant les caractéristiques suivantes :

- 32 entrées ;
- Recherche pseudo-associative à partir du contenu d'une information de 29 bits (adresse T, i, c ou T, i) ;
- Gestion de la mémoire à l'aide d'un mécanisme L.R.U. câblé.

Remarques préliminaires :

Les différentes options présentées dans ce paragraphe, et choisies pour l'implémentation de la mémoire associative du T.A.L., résultent principalement de deux objectifs généraux : un fonctionnement rapide d'une part, et un faible encombrement d'autre part.

Les valeurs de certains paramètres sont arbitraires, et peuvent dépendre du type de la technologie envisagée pour la réalisation de la machine (nombre de blocs de mémoire RAM, tailles des mots dans chaque bloc, nombre d'entrées,...) ; il conviendra de valider rigoureusement le choix de ces paramètres par une étude statistique basée sur l'utilisation du code machine.

5.3.2.6.1. Description de la mémoire associative : MA

La mémoire associative MA est utilisée au sein du T.A.L. afin de permettre un accès rapide au descripteur de segment nécessaire pour la transformation d'adresse lexicale en adresse virtuelle.

Soit (T, i, c, d) l'adresse lexicale à transformer par le T.A.L.

Le rôle de MA est de fournir l'**unique** descripteur associé à l'information (k, T, i, c) s'il est mémorisé dans MA, ou de l'y inscrire sinon.

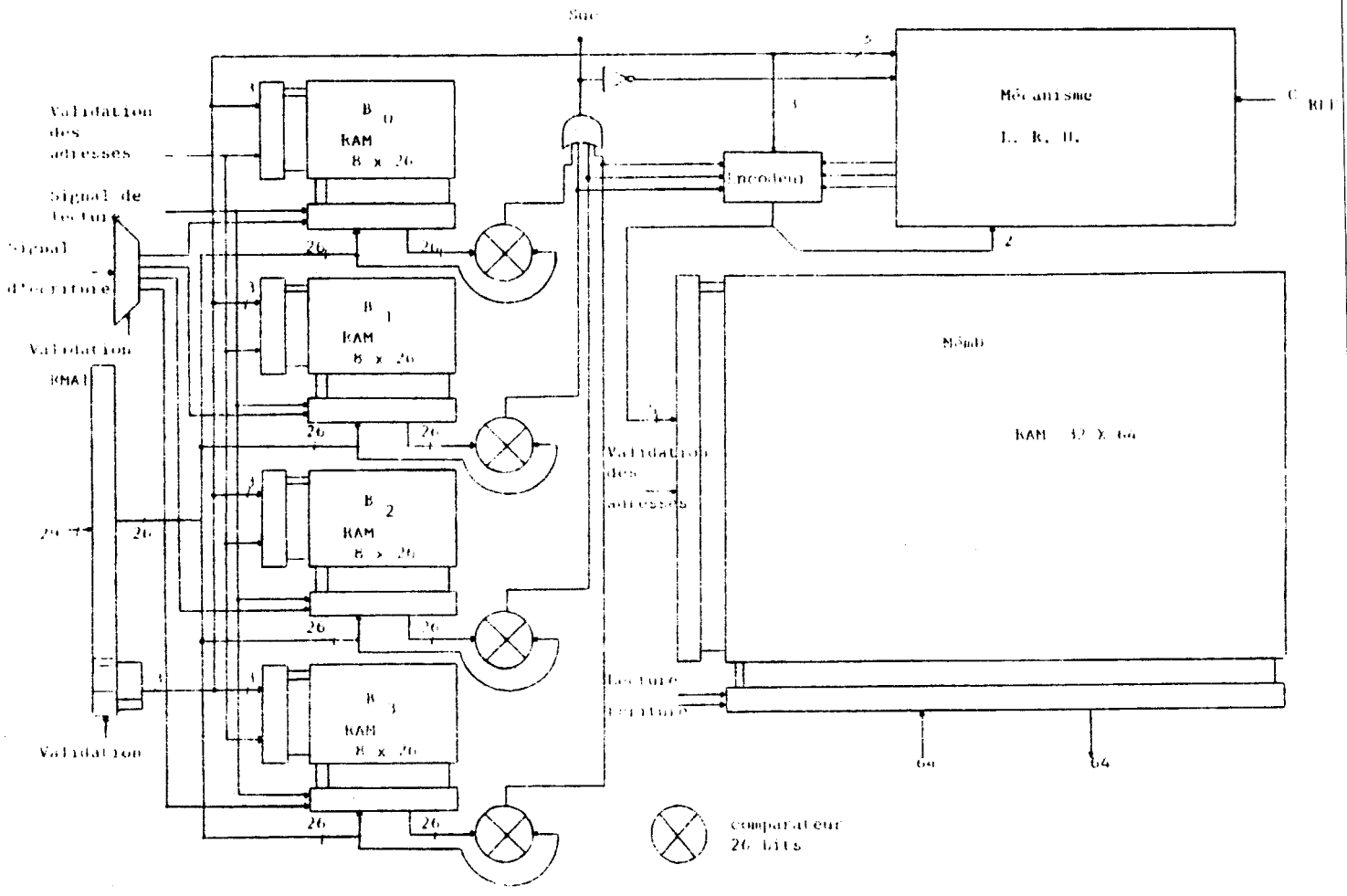


Schéma synoptique de la Mémoire Associative

T est le numéro de tâche (sur 16 bits : T_0 ... T_{15})

i est le niveau lexical (sur 8 bits : i_0 ... i_7)

c est le numéro de catégorie (sur 4 bits : c_0 ... c_3)

k est nul et n'est pas à prendre en compte si k = 1 (il s'agit du bit de typage évoqué précédemment).

La mémoire de stockage est subdivisée en deux parties :

- La première (appelée matrice A dans la description fonctionnelle) contient les informations à exploiter pour les recherches ; la comparaison séquentielle de 32 entrées de 26 bits étant prohibitive en temps, de même que la comparaison parallèle de 32 entrées étant prohibitive en matériel, nous avons organisé ces informations en 4 blocs (RAM) B_0 , B_1 , B_2 et B_3 de 8 mots chacun : cela permet l'accès et la comparaison en parallèle des informations de chaque bloc contenues à un même numéro de ligne ;

- La seconde (appelée Matrice B dans la description fonctionnelle) contient les descripteurs de segments ; c'est un bloc de 32 mots de 64 bits de mémoire vive (RAM) nommé MémD.

a) Description de la recherche pseudo-associative dans MA

La recherche associative dans MA, s'accomplit grâce aux opérations suivantes :

- Les 29 bits d'information à partir desquels l'accès est effectué, sont présentés dans le registre RMA1 ;
- Lorsque la commande de recherche associative est activée, les bits i_7 , c_2 , c_3 sont utilisés pour la sélection de la ligne n° $i_7c_2c_3$ dans chaque bloc B_0 ... B_3 ;
- Le contenu des 4 mots correspondants est chargé dans les 4 registres d'entrée-sortie de chaque bloc et est comparé en parallèle sur 26 bits au contenu de RMA1 tronqué des bits i_7, c_2 et c_3 ;

- Si l'une des 4 comparaisons aboutit, il y a émission d'un signal de succès ; celui-ci indique que le descripteur recherché se trouve mémorisé dans MémD et peut être lu ;
- Le numéro de bloc où il y a succès est alors encodé : 00 pour B_0 01 pour B_1, 10 pour B_2 et 11 pour B_3 ;
- La lecture du descripteur recherché est alors rendue possible.

Ainsi, l'encodage porte sur 4 signaux désignant les 4 blocs et provenant soit des comparateurs, soit d'un sous-système LRU ; ces 4 signaux sont transformés en 2 signaux appelés b_0 b_1 qui, concaténés à i_7, c_2, c_3 forment l'adresse à présenter à MémD.

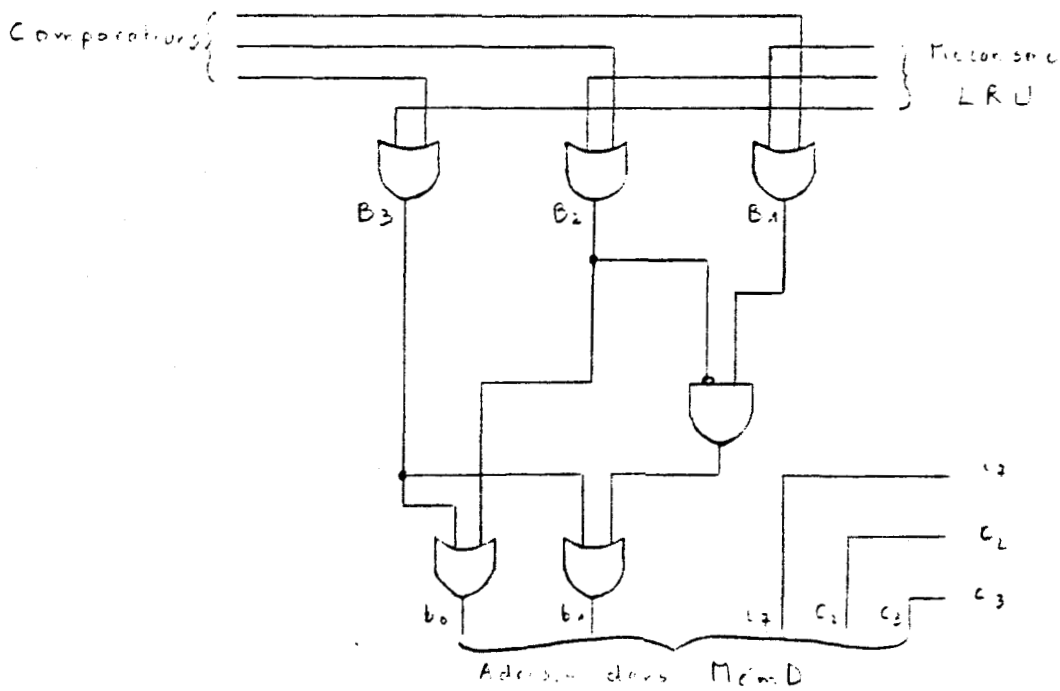
L'ENCODEUR n'utilise que les 3 signaux associés à B1, B2 et B3 : le signal associé à B0 se déduit des 3 autres.

De plus, pour tenir compte des cas de l'initialisation (plusieurs blocs peuvent être disponibles), une priorité est instaurée dans le choix de ces blocs.

Les équations logiques sont alors (après réduction) :

$$b_0 = B_2 + B_3$$
$$b_1 = B_3 + B_2\,B_1$$

Le schéma logique qui en découle est le suivant :



b) Caractéristiques de la recherche associative

La situation d'une même ligne dans les quatre blocs peut s'effectuer à l'aide de fonctions de **hashing** plus sophistiquées qu'une simple règle de troncature.

Celle qui est adoptée dans ce schéma possède pourtant des avantages qui nous ont paru décisifs :

- L'adresse de la ligne à sélectionner dans les blocs B_i est obtenue immédiatement puisque composée des bits i_7 , c_2 et c_3 ; du temps et de la logique supplémentaires sont ainsi évités ;
- Cette méthode permet de diminuer la taille des mots de chaque bloc puisque la mémorisation des bits composant l'adresse de sélection est inutile ; ceci a également pour conséquence de réduire la taille des comparateurs de 29 à 26 bits ;
- L'utilisation des 2 bits de poids faible consécutifs C_2C_3 implique que 2 descripteurs consécutifs (ou même voisins) seront mémorisés dans des entrées successives (ou voisins) de MA, ce que pourrait empêcher une autre fonction de hashing ; la présence simultanée de descripteurs consécutifs (ou voisins) permet ainsi l'exploitation du principe de localité appliqué aux catégories d'un SLCL ;
- Le choix du bit i_7 permet de ne pas trop restreindre le nombre de descripteurs de SLCL présents dans MA ; ces descripteurs correspondent en effet à des adresses lexicales de type k (= 1), T, i, ooo, de nature complètement différentes de celles du type k (= 0), T, i, 0000 qui correspondent à des descripteurs contenus dans les catégories n°0 des SLCL ; le fait de prendre i_7 au lieu de, par exemple, C1 permet de répartir ces adresses (où $C_0 = C_1$ $C_2 = C_3 = 0$) sur 8 entrées au lieu de 4.

c) Les autres opérateurs effectués par MA

Lorsque la recherche n'aboutit pas, le descripteur demandé doit être retrouvé suivant les processus indiqués aux § 5.2.3 et 5.2.4.

Un échec de la recherche associative signifie que la MA devra bientôt mémoriser l'information qu'elle ne possède pas et qui vient de lui être demandée ; ainsi, le descripteur devra être mémorisé dans MémD et l'adresse lexicale correspondante (sauf $i_7c_2c_3$) dans un des blocs B_i.

Le chargement dans les RAM se fait par l'intermédiaire des registres Lecture/Ecriture associés à chaque bloc et à MémD. Le choix de l'entrée pour une écriture dans MA suit la règle du LRU (Least-Recently-Used) qui veut que l'entrée désignée soit celle qui correspond à l'information la moins lue depuis le plus longtemps ; l'entrée choisie se trouve à la ligne contenant l'information LRU parmi les 4 que sélectionnent les bits $i_7 c_2 c_3$.

Le principe du LRU adopté ainsi que son implémentation câblée sont décrits maintenant.

5.3.2.6.2. Description du système LRU câblé associé à MA

a) Principe

A chaque numéro j de ligne correspond un sous-système LRU (noté SSj) permettant de retrouver l'information la moins utilisée depuis le plus longtemps (least-recently-used) entre les 4 informations de la ligne j des blocs B_0, B_1, B_2 et B_3 ; il existe donc 8 sous-systèmes SS_0.... SS_7.

Un seul de ces sous-systèmes est sélectionné, mis à jour ou sollicité, lors d'une lecture ou d'une écriture dans MémD.

Ce sous-système SSj est sélectionné à partir du numéro j de ligne correspondant à $i_7c_2c_3$.

Chaque sous-système SSj contient 4 compteurs, un par bloc, notés $\overset{\circ}{D}ji$ $\overset{'}{D}ji$ où $i \in \{0,..,3\}$.

Ce compteur correspond à un numéro d'ordre et porte donc sur 4 valeurs 0, 1, 2, 3.

Lorsqu'une entrée est référencée (lecture ou écriture effectives), son compteur est mis à son maximum (soit 3).

Seuls, les compteurs qui étaient supérieurs à ce dernier sont décrémentés de 1 ; les autres compteurs ne sont pas modifiés.

Lorsque plusieurs références à une même entrée ont lieu en séquence, aucun compteur du sous-système n'est modifié.

Ainsi, le compteur qui a la valeur maximum (3) correspond à l'entrée de la ligne j, référencée le plus récemment parmi celles des 4 blocs.

L'entrée LRU est celle qui a son compteur zéro.

b) Exemple

Considérons un sous-système SSj

a) Cas où une référence est faite à la ligne j du bloc B1 (lecture) :

1. Avant référence

bloc 3	1
bloc 2	0
bloc 1	2
bloc 0	3

2. Après référence

1
0
3
2

Dans le cas d'une écriture, c'est le bloc désignant le LRU (ici B2) qui est choisi. Le compteur associé est ensuite mis à jour suivant le même mécanisme.

b) Cas de l'initialisation : tous les compteurs sont à 0

1. Etat initial
(la mémoire associative est vide)

bloc 3	0
bloc 2	0
bloc 1	0
bloc 0	0

2. 1ère écriture
   B3 prioritaire sur B_2, B_1 et B_0

3
0
0
0

3. 2ème écriture
   B2 prioritaire sur B_1 et B_0

2
3
0
0

4. 3ème écriture
   B1 prioritaire sur B_0

1
2
3
0

c) Descriptions logique et matérielle du système câblé LRU

Implémentation d'un sous-système SSj

- Chaque compteur est formé de 2 bascules D ; la sortie de chaque bascule enregistre sur le front montant de la commande de référence (clock) l'état de l'entrée - elle ne change pas jusqu'à la commande de référence suivante ;

- Lorsque le numéro de bloc référencé est disponible ($b_0 b_1$), celui-ci est envoyé comme cemmande à 2 multiplexeurs 4 entrées M1 et M2 qui permettent d'obtenir les valeurs notées R_0 et R_1 des 2 bascules du compteur courant de l'entrée référencée ; ces deux valeurs permettent, au travers d'une certaine logique L_j, de fournir les entrées des bascules

  D°_{j0} $D^{\prime}_{j0}$, D°_{j1} $D^{\prime}_{j1}$, D°_{j2} $D^{\prime}_{j2}$, D°_{j3} $D^{\prime}_{j3}$,

  à partir des anciennes valeurs toujours présentées à la sortie des bascules ; ces nouvelles valeurs sont enregistrées au top d'horloge S_{ref}.

Les ensembles de logique L_0 , L_1, L_2 et L_3 sont identiques ; la logique qu'ils contiennent consiste à mettre à jour les valeurs des bascules selon le contenu du compteur référencé au moment de la sélection.

Cette logique est décrite de façon formelle à l'aide des équations suivantes :

Lorsque la commande S_{Ref} est activée

$\forall\ j \in \{0,..,7\}\ ;\ \forall\ i \in \{0,..,3\}$

$(D^{\circ}ji)' = \overline{D^{\circ}ji}\ \overline{D^{\prime}ji}\ \overline{R_0}\,\overline{R_1} + \overline{R_0}R_1 D^{\prime}ji + D^{\circ}ji\ D^{\prime}ji + D^{\circ}ji\ R_0$

$(D^{\prime}ji)' = \overline{D^{\circ}ji}\ \overline{D^{\prime}ji}\ R_0\overline{R_1} + D^{\circ}ji\ D^{\prime}ji\ (R_0 + R_1) + D^{\circ}ji\ \overline{D^{\prime}ji}\ (\overline{R_0} + \overline{R_1}) + R_0 R_1 D^{\circ}ji$

Ces équations fonctionnelles peuvent être simplifiées en considérant que la mise au maximum du compteur référencé est effectuée par la mise à un des bascules correspondantes.

Les équations deviennent alors :

$\forall\ j \in \{0,..,7\}\ ;\forall\ i \in \{0,..,3\}$

$(D^{\circ}ji)' = D^{\circ}ji\ D^{\prime}ji\ (\overline{R_0 R_1})$

$(D^{\prime}ji)' = \overline{R_0} D^{\circ}ji\ \overline{D^{\prime}ji} + R_0 \overline{D^{\circ}ji}\ D^{\prime}ji$

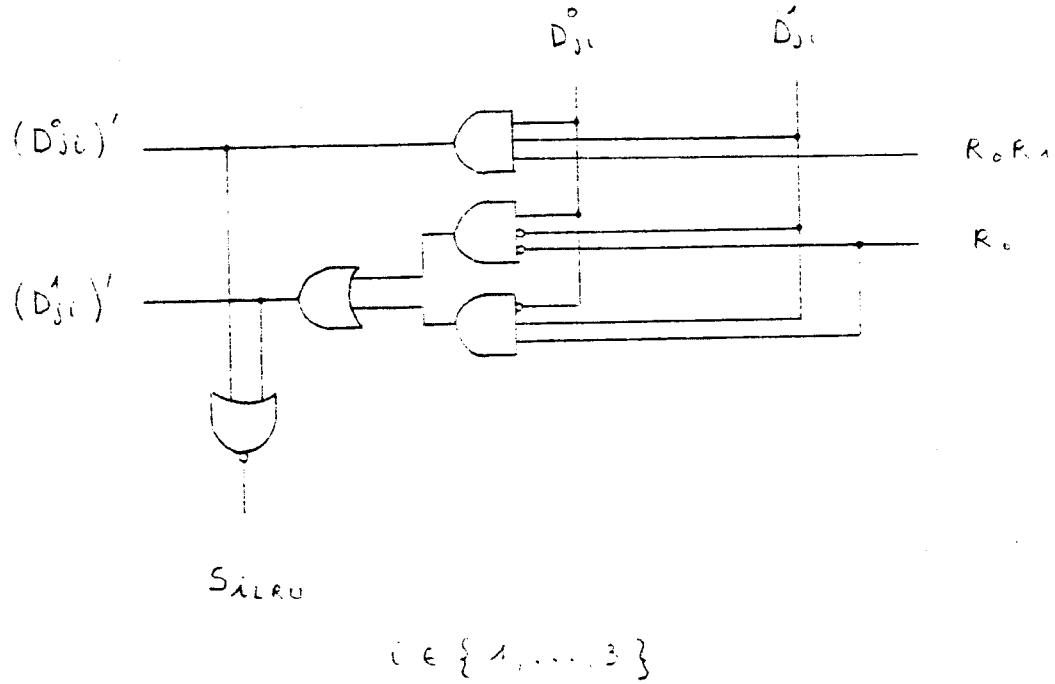
Soit $S_{i\ LRU}$ le signal indiquant si le compteur est LRU (compteur nul) :

$S_{iLRU} = \overline{D^{\circ}ji + D^{\prime}ji}$

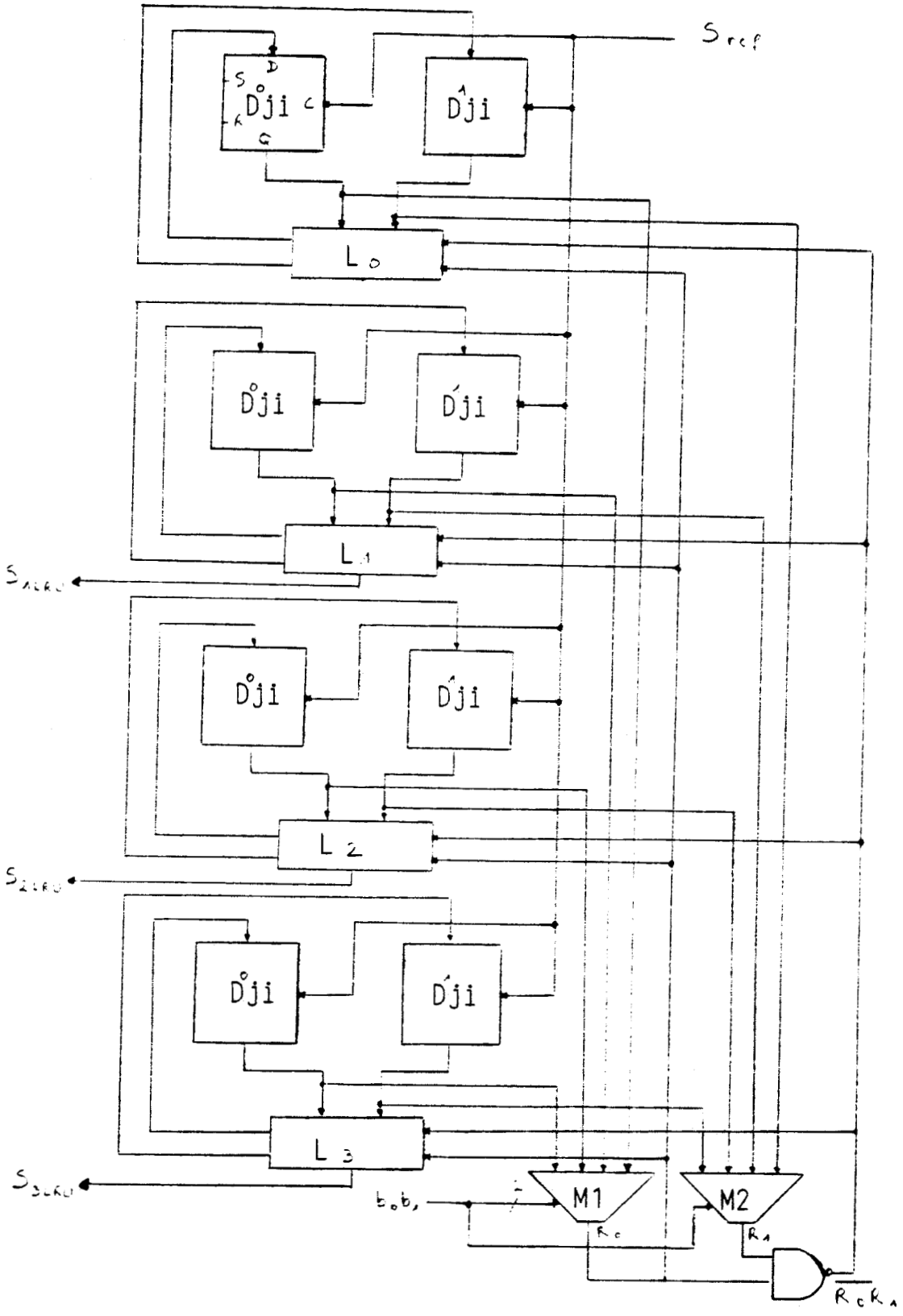
Remarque : Il existe toujours un compteur à zéro (LRU) ;
si $S_{1\,LRU} = S_{2\,LRU} = S_{3\,LRU} = 0$ alors nécessairement $S_{0\,LRU} = 1$

Cette constatation permet de simplifier le schéma logique en ne générant pas le signal S_{0LRU} qui est implicite.

Le schéma logique d'un ensemble Li est le suivant :



Le schéma d'un sous-système LRU SSj correspondant aux 4 entrées d'une même ligne j dans les blocs B_0, B_1, B_2, B_3 est représenté à la page suivante.



SYNOPTIQUE D'UN SOUS-SYSTEME SSj

5.3.2.6.3. Utilisation du système LRU au sein du mécanisme de la mémoire associative

Il arrivera qu'une recherche en mémoire associative n'aboutisse pas.

Dans ce cas, le signal d'échec (non succès) est transformé en commande de recherche du LRU ; de façon à préparer l'adresse de l'entrée où l'écriture sera possible.

Les résultats, fournis par le sous-système LRU SSj associé à la ligne $j = i_7c_2c_3$ de chaque bloc, sont filtrés à l'aide d'un démultiplexeur DMX1.

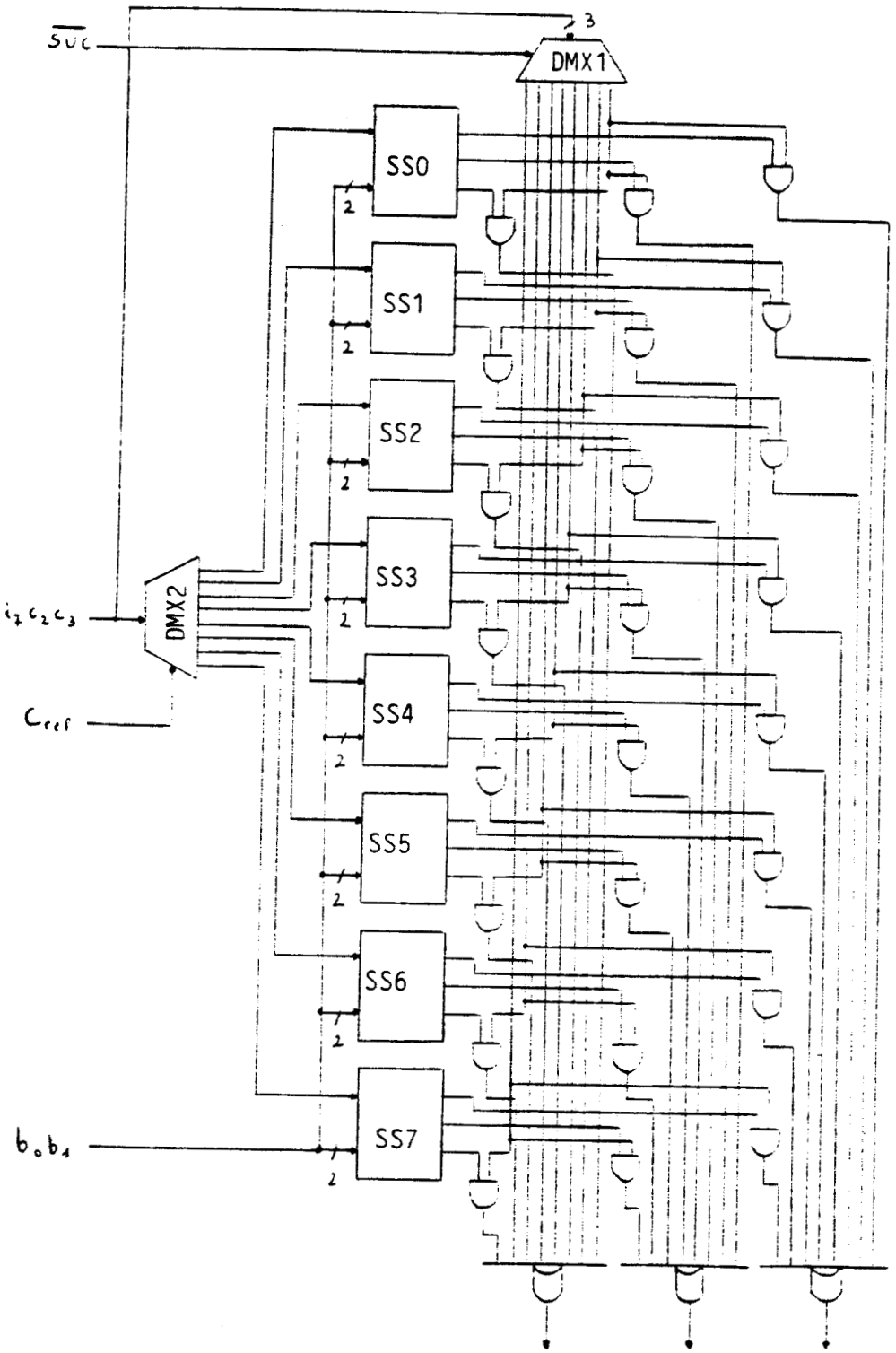
L'encodeur reçoit alors les trois signaux S_{1LRU}, S_{2LRU}, S_{3LRU} associés aux trois compteurs dans chaque bloc B_1, B_2, B_3.

(Le signal S_{0LRU} se déduit des trois autres signaux).

Un seul signal est à 1 ; c'est le LRU. L'encodeur délivre en sortie le code du numéro de bloc vainqueur qui, concaténé avec le numéro de ligne $i_7c_2c_3$, constitue l'adresse dans MémD où l'écriture de l'information sera effectuée.

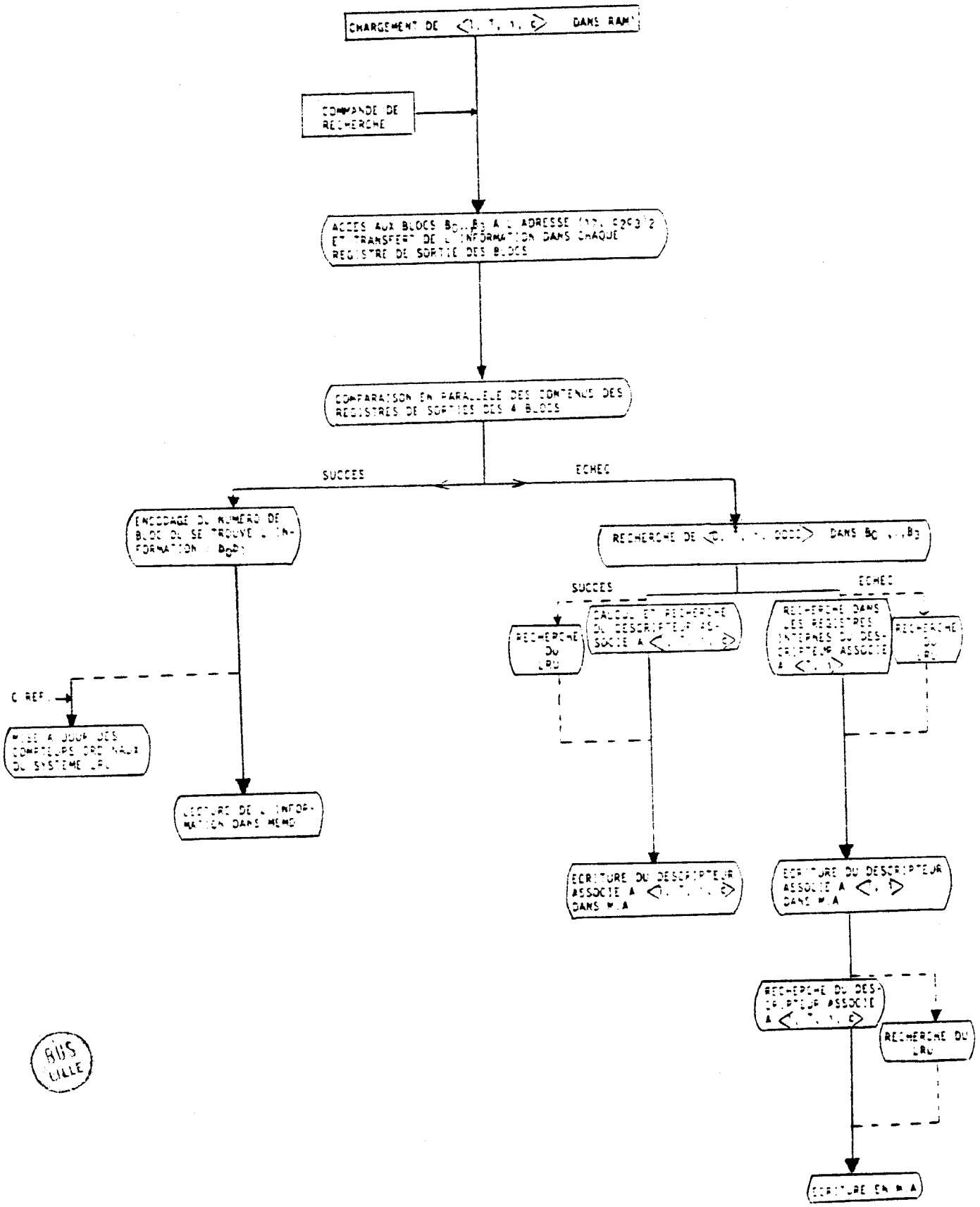
Ce numéro codé b_0b_1 est utilisé aussi pour l'écriture de l'adresse (k, T, i, c) dans le bon bloc B_{b0b1}.

Lorsque la commande d'écriture est effective (c_{REF}), le sous-système LRU associé à la ligne $i_7c_2c_3$ est mis à jour (filtrage du sous-système à l'aide du démultiplexeur DMX2).



SYNOPTIQUE DU MECANISME GENERAL LRU

Le déroulement des actions pour la recherche associative peut être schématisé de la façon suivante :



5.3.2.6.4. Commentaires et évaluations

La mémoire associative MA proposée dans ce paragraphe n'est pas ce qu'il est convenu d'appeler une "Fully Associative Memory" mais plutôt une "Set Associative Memory" (cf [BCB 74]) ; l'information qui sert à la recherche (dans ce cas : (0, T, i, c) ou (1, T, i, oooo)) n'est pas à un endroit quelconque de la mémoire et ne peut occuper que quatre positions possibles.

Ceci est dû à la méthode de hashing (en l'occurrence une simple troncature sur $i_7c_2c_3$) que supportent les mécanismes de simulation d'associativité décrits précédemment et qui font que MA est en réalité "pseudo-associative".

Le principe de gestion du chargement de MA est celui, relativement original, d'un LRU câblé ; si, d'un point de vue théorique, cette solution offre les meilleures garanties d'efficacité, celles-ci devront être validées par simulation dans de prochaines études.

Cette proposition d'implémentation de MA reste d'ailleurs dans son ensemble soumise aux éventuelles modifications qu'exigera une simulation voire une réalisation ultérieure.

Afin de contribuer à l'évaluation globale du TAL, en temps et en coût de matériel, une estimation basée sur le schéma d'implémentation actuel est quand même effectuée.

a) Evaluation en temps de fonctionnement

Le mode de fonctionnement dont l'évaluation est nécessaire aux calculs d'évaluation du TAL qui font l'objet du § 5.4. est celui de la recherche associative.

Pour cela, il suffit de considérer la suite d'actions suivantes (la constante $\bar{\tau}$ est explicitée au § 5.4) :

1. Chargement de k, T, i, c dans le registre RMA1 :

   Cette opération est prise en compte dans les calculs hors MA ;

2. Lecture en parallèle dans B_0, B_1, B_2 et B_3 à la ligne $i_7c_2c_3$:

   Le temps retenu est le temps d'accès à une RAM à 8 entrées en technologie HMOS : t (RAM8) = 19 ns

3. Comparaison en parallèle sur 26 bits :

   Une technique à base de portes OUEX permet de rendre cette comparaison équivalente au temps de passage dans 4 portes :

4. Encodage de l'adresse de l'information à accéder dans MémD :

   L'évaluation porte uniquement sur l'encodage, les opérations de mise à jour du LRU se poursuivent concurremment ; le temps est donc celui de passage dans 5 portes : $5\ \bar{\tau}$

5. Lecture dans MémD à la ligne $b_0b_1i_7c_2c_3$:

   Il s'agit du temps d'accès à une RAM 32 x 64 en technologie HMOS t (RAM 32) = 22 ns

Pour les besoins des calculs d'évaluation du TAL (voir § 5.4), la recherche associative est distinguée en deux parties :

- La recherche associative proprement dite correspondant aux actions (2) et (3) dont le déroulement est associé au temps TMA1 tel que :

$$TMA1 = t\ (RAM\ 8) + 4\ \bar{\tau}$$

- L'accès à l'information cherchée en cas de succès, correspondant aux actions (4) et (5) dont le déroulement est associé au temps TMA2 tel que :

$$TMA2 = 5\bar{\tau} + t\ (RAM\ 32)$$

b) Evaluation en coût de matériel

Comme pour l'ensemble du TAL, cette évaluation correspond à une estimation de la surface qu'occuperait la réalisation en circuits de type HBMOS 4 des schémas logiques proposés (le paragraphe 5.4.3. expose les règles et les limites de cette évaluation).

Les estimations suivantes sont exprimées dans l'unité utilisée par le fabricant : le pas de grille, noté L/g (voir 5.4.3.) :

- 4 RAM de 8 entrées (registres d'E/S et mécanismes d'adressage inclus): 6020 L/g ;

- 2 RAM de 32 entrées (registre d'E/S et mécanisme d'adressage inclus) : 7320 L/g ;

- 4 comparateurs (un comparateur correspond à 1 ET et 26 OUEX) : 4 + (6 + 26 * 7) = 752 L/g ;

- Encodeur (1 ET et 5 OU) : 26 L/g ;

- Mécanisme LRU (8 systèmes SSj + 2 démultiplexeurs + 24 ET + 3 OU 8 entrées) : 2629 L/g ;

- Autres éléments (1 INVERSEUR + 1 OU 4 entrées + 1 démultiplexeur) : 3 + 5 + 42 = 50 L/g.

Le total, correspondant à la Mémoire Associative, ne prend pas en compte la surface de routage (prise en compte globalement) : 16 797 L/g.

5.3.3. Evolutions et transformations possibles

Le schéma d'implémentation proposé est une première solution à la réalisation des fonctionnalités du T.A.L. décrites dans le paragraphe 5.2.

L'objectif est d'effectuer une première évaluation des temps de fonctionnement ; l'étude pourra conduire à une restructuration de l'architecture de MLI en fonction des temps d'exécution obtenus pour les différents modules qui peuvent induire une nouvelle organisation du parallélisme.

Le choix de la technologie et les contraintes qui lui seraient associées ou qui en découleraient peuvent amener également à une révision de l'architecture du T.A.L.

Par ailleurs, une étude plus précise du fonctionnement du T.A.L. dans son environnement matériel remettra certainement en cause, par exemple, les découpages en registre des entrées et sorties du T.A.L.

Enfin, comme cela a déjà été précisé, la mémoire associative proposée est une solution demandant à être validée et donc reste fortement sujette aux modifications.

Il n'en demeure pas moins que ce schéma "prototype" offre un premier support solide à une évaluation des temps de fonctionnement du T.A.L. et permet de situer le cadre des performances qu'on peut en attendre ainsi que le montre la suite de ce chapitre.

5.4. MODES DE FONCTIONNEMENT ET EVALUATIONS

5.4.1. Principe et conditions d'évaluation en temps

La description fonctionnelle du TAL a permis de dégager 4 modes de fonctionnement (cf § 5.2.3.).

La présentation du schéma d'implémentation a donné un support aux opérations mises en évidence par la description fonctionnelle.

Au stade actuel de la conception, seuls les comparateurs, les additionneurs et l'Unité de Commande ne sont pas entièrement définis.

Les estimations portant sur le temps de fonctionnement de chaque mode reposent sur des évaluations de 4 types différents référençant :

1. Les éléments matériels créés pour le TAL dont les schémas logiques sont détaillés au § 5.3 ; les temps sont calculés en dénombrant les portes qui se trouvent sur le chemin le plus long que doit parcourir un signal en entrée pour être valide en sortie ; ce nombre est alors multiplié par un temps moyen de propagation à travers une porte, $\overline{\tau}$, qui dépend de la technologie choisie ;

2. Les opérateurs classiques que sont les additionneurs et les comparateurs dont les temps de fonctionnement proviennent de la littérature ou des fabriquants ;

3. La mémoire associative dont l'estimation des performances repose sur un schéma d'implémentation choisi en partie à cause de contraintes devant être justifiées par la suite par des études statistiques appropriées ;

4. Les registres et le câblage qui induisent des temps de stabilisation et de propagation des signaux considérés à ce niveau comme négligeables compte tenu de l'incertitude régnant sur certains opérateurs.

Les opérations de contrôle dépendant de l'Unité de Commande (droits d'accès, de lecture, d'écriture,...) ne peuvent être définies pour l'instant, étant donné la non-formalisation du champ "TYPE" de descripteur de segment.

Il est toutefois raisonnable de penser que ces opérations se dérouleront dans un temps n'appartenant pas au chemin de données le plus long.

La sélection des signaux de commande par l'Unité de Commande du T.A.L. se fait pendant un temps "caché" dans la prise en compte des chemins critiques ; de la même façon, le temps d'initialisation pendant lequel sont émis les premiers signaux de commande est considéré comme négligeable car ces signaux sont toujours les mêmes quelque soit le mode de fonctionnement choisi et ne nécessitent pas de décodage préalable.

5.4.2. Graphes de fonctionnement

Afin d'établir une estimation globale du temps d'exécution de chacun des modes, un graphe explicitant les diverses actions effectuées sur tous les chemins de données est défini dans chaque cas.

Les actions apparaissant dans les graphes n'ont pas un niveau de détail identique : l'action "recherche associative" est plus complexe que l'action "sélection dans un multiplexeur", par exemple.

Les commandes mentionnées dans la description des actions sont des micro-commandes émises par l'Unité de Commande.

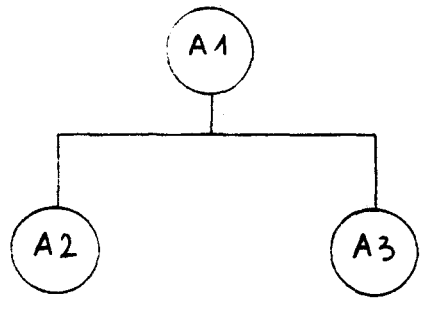
Le séquencement et la synchronisation de ces micro-commandes ne sont pas explicités.

Les commandes peuvent représenter plusieurs signaux.

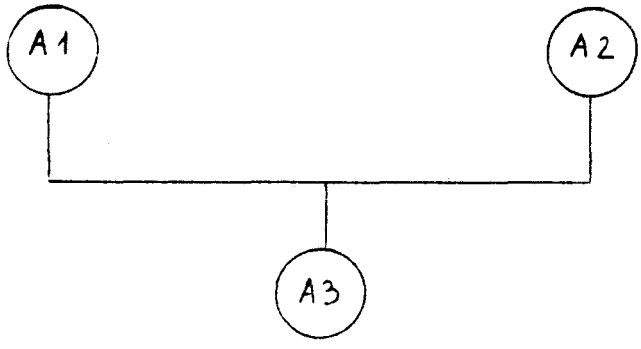
La progression d'une donnée sur un chemin de données est définie en lisant le graphe de haut en bas.

Cas de figures :

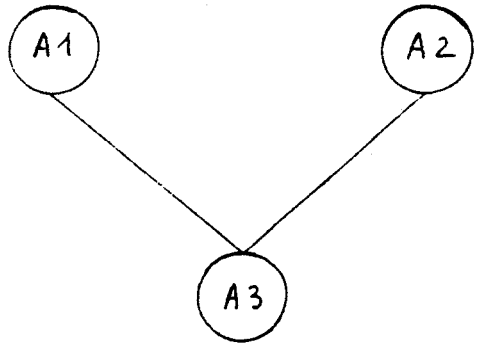
A2 est exécutée après A1



A2 et A3 sont exécutées après A1



A3 doit attendre l'exécution de A1 et A2 pour être exécutée



A3 est exécutée après A1 et A2 mais A1 (ou A2) peut provoquer l'exécution de A3 sans attendre la fin de l'exécution de A2 (ou A1)

Ce 4ème cas représente uniquement l'action d'arrêt de fonctionnement du T.A.L. qui peut avoir lieu :

- Après la fin normale des opérations du mode concerné ;
- Sur intervention d'actions de contrôles.

5.4.2.1. Le mode 1

Il s'agit du mode le plus fréquent ; c'est à partir de lui qu'a été conçue la structure primaire du T.A.L. (additionneurs et comparateurs).

Le schéma intègre un parallèlisme d'exécution maximum.

Définition des actions réalisant le mode 1 :

A0 : Prise en compte du contenu de RE4 par l'Unité de Commande ;
A1 : Chargement de RMA1 avec le contenu de RE1 - T, i, c + 1 bit de typage - (commande 1 et 2) ;
A2 : Recherche associative sur le contenu de RMA1 qui donne lieu à l'envoi d'un signal de succès vers l'Unité de Commande (commande 3 + commandes internes à la Mémoire Associative) ;
A3 : Comparaison entre le contenu de RIC et la partie de RE1 contenant i dans COMP1 qui donne lieu à l'envoi d'un signal vers l'Unité de Commande (commandes 4 et 5) ;

A4 : Recherche du descripteur qui donne lieu à l'envoi d'un signal de succès vers l'Unité de Commande (commandes internes à la Mémoire Associative) ;

A5 : Sélection des signaux de sortie par MPX4 pour RMA2 (commandes 6 et 7) ;

A6 : Contrôles sur le type du descripteur (commandes internes à l'Unité de Commande) ;

A7 : Sélection du masque dans SM (commande 8) ;

A8 : 1er masquage dans MASQ (sans commande) ;

A9 : Reconditionnement du champ "TAILLE" du descripteur dans OPRC (sans commande) ;

A10 : Sélection de la valeur du champ "TAILLE" par MPX1 (commande 9) ;

A11 : Sélection du contenu de RE2 (d) par OPMPX1 (commande 10) ;

A12 : Sélection du contenu de RE3 (l) par OPMPX2 (commande 11) ;

A13 : Addition (d + l) dans ADD1 (commande 12) ;

A14 : Sélection du résultat de ADD1 par MPX2 (commande 13) ;

A15 : Comparaison (d + l $<=$ t?) dans COMP2 qui donne lieu à l'envoi d'un signal vers l'Unité de Commande (commande 14) ;

A16 : 2ème masquage dans MASQ validant l'adresse (b) à l'entrée de ADD2 (commande 8) ;

A17 : Sélection du contenu de RE2 (d) par OPMPX3 (commande 15) ;

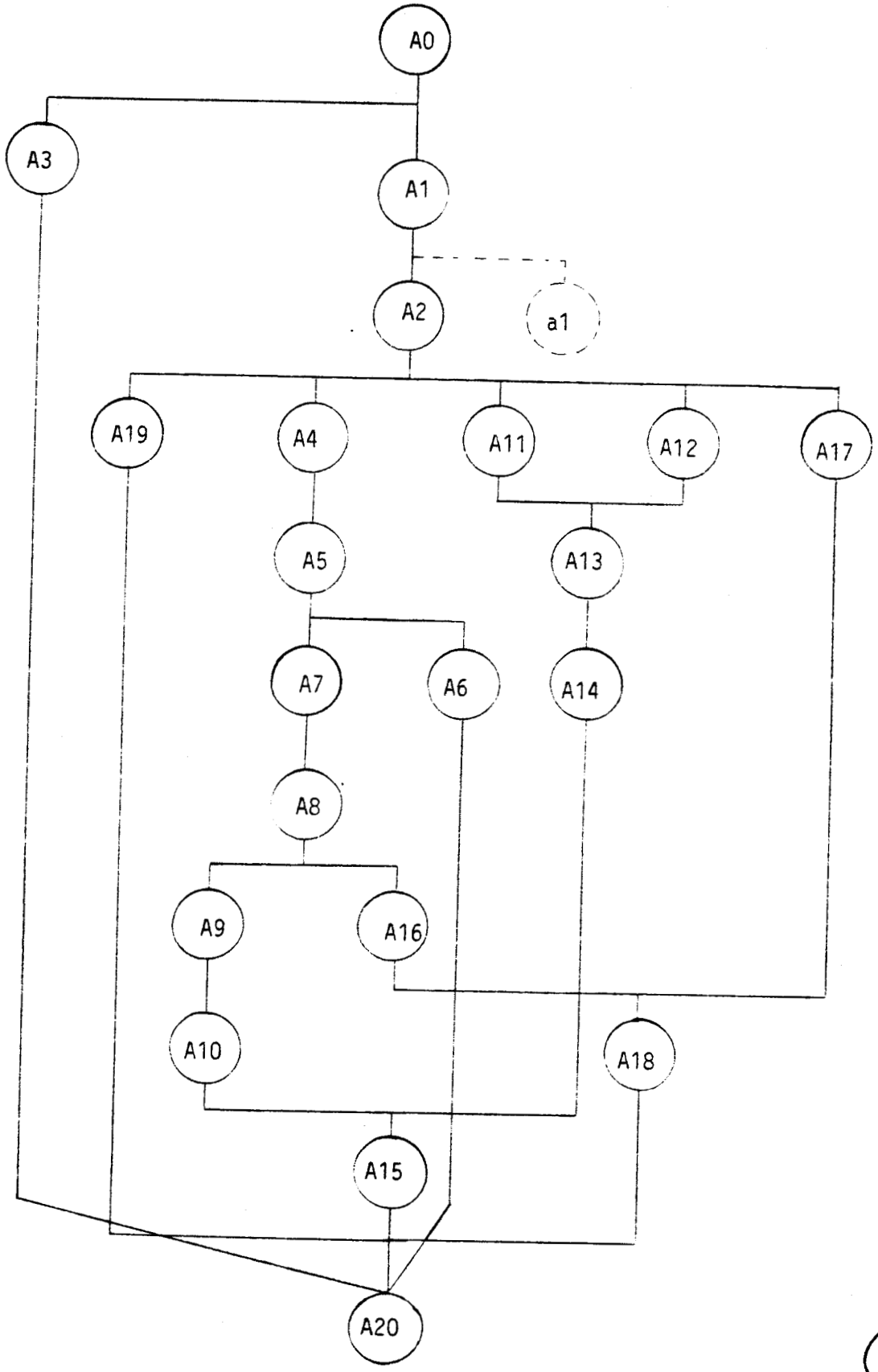
A18 : Addition (b + d) dans ADD2 (commande 16) ;

A19 : Sélection du contenu de RE3 (l) par MPX3 (commande 17) ;

A20 : Chargement de RS3 par l'Unité de Commande (commandes internes à l'Unité de Commande).

Définition des actions amorcées par l'Unité de Commande mais non prises en compte dans le schéma d'exécution du mode 1 :

a1 : Chargement de RMA1 avec une partie du contenu de RE1 (T, i) + la configuration 00001 sélectionnée par MPX0 (commandes 1 et 2).



Graphe de Fonctionnement du Mode 1

Evaluation du temps de fonctionnement

L'analyse des différents chemins de données a permis de déterminer comme étant le plus long le chemin de données suivant :

A0, A1, A2, A4, A5, A7, A8, A9, A10, A15, A20

ce qui équivaut au temps :
$16\overline{\tau}$ + TMA1 + TMA2 + TCOMP2

Le temps de fonctionnement du mode 1 est donc égal à 156ns ce qui fait un ordre de grandeur de 160ns.

5.4.2.2. Le mode 1b

C'est le mode corollaire au mode 1 : T, i, c se trouvent également en Mémoire Associative mais d et l sont exprimés en bit.

Ce mode est celui qui nécessite le plus d'opérations internes au TAL.

Définition des actions :

A0 : Prise en compte par l'Unité de Commande du contenu de RE4
A1 : Chargement de RMA1 avec le contenu de RE1 -T, i, c + le bit de typage- (commandes 1 et 2)
A2 : Recherche associative sur le contenu de RMA1 qui donne lieu à l'envoi d'un signal de succès vers l'Unité de Commande (commande 3 + commande interne à la Mémoire Associative)
A3 : Comparaison entre le contenu de RIC (ic) et la partie de RE1 représentant i dans COMP1 (i <= ic ?) qui donne lieu à l'envoi d'un signal vers l'Unité de Commande (commandes 4 et 5)
A4 : Recherche du descripteur qui donne lieu à l'envoi d'un signal de succès vers l'Unité de Commande (commandes internes à la Mémoire Associative)

A5 : Sélection des signaux de sortie par MPX4 pour RMA2 (commandes 6 et 7)

A6 : Contrôles sur le type du descripteur (commandes internes à l'Unité de Commande)

A7 : Sélection du masque dans SM (commande 8)

A8 : 1er masquage dans MASQ (sans commande)

A9 : Reconditionnement du champ "TAILLE" du descripteur par OPRC (sans commande)

A10 : Sélection de la valeur contenue dans "TAILLE" par MPX1 (commande 9)

A11 : Sélection du contenu de RE2 (d) par OPMPX1 (commande 10)

A12 : Sélection du contenu de RE3 (l) par OPMPX2 (commande 11)

A13 : Addition (d+l) dans ADD1 (commande 12)

A14 : Validation du test du d" - d+l mod 8 - (commande 18)

A15 : Sélection de 0 par OPMPX1 -R.A.Z.- (commande 10)

A16 : Validation des 32 bits de poids fort (d+1/8) par OPDECAL (commande 16)

A17 : Sélection des sorties de OPDECAL par OPMPX2 (commande 11)

A18 : Addition dans ADD1 : 0 + d+1/8 [+1] = NOS (commande 13)

A19 : Sélection des sorties de ADD1 (NOS) par MPX2 (commande 13)

A20 : Comparaison dans COMP2 (NOS $\leq$ t ?) qui donne lieu à l'envoi d'un signal vers l'Unité de Commande (commande 14)

A21 : Inversion du masque dans SM et 2ème masquage dans MASQ validant l'adresse (b) à l'entrée de ADD2 (commande 8)

A22 : Sélection des 32 bits de poids fort de RE2 (d') par OPMPX3 (commande 15)

A23 : Addition dans ADD2 (commande 16)

A24 : Validation des 32 bits de poids faible par OPDECAL -NOS- (commande 19)

A25 : Sélection d' par OPMPX1 (commande 10)

A26 : Sélection des sorties de OPDECAL (NOS) par OPMPX2 (commande 11)

A27 : Addition dans ADD1 avec retenue - NOS + d' + 1 - (commandes 12 et 20)

A28 : Sélection du résultat de ADD1 par MPX3 (commande 17)

A29 : Chargement de RS2 avec les 3 bits de poids faible de RE2 (d")

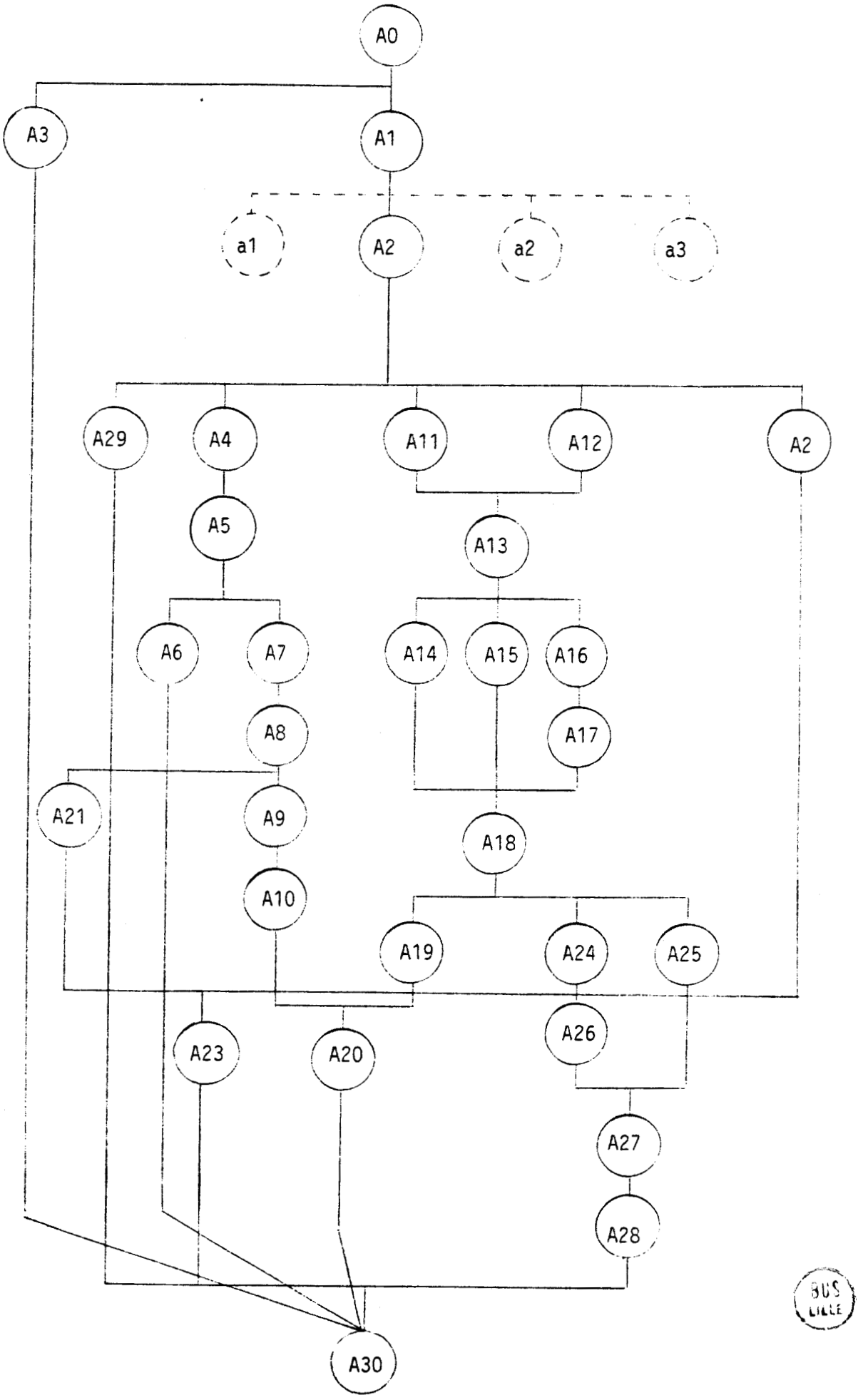
A30 : Chargement de RS3 par l'Unité de Commande

Définition des actions amorcées par l'Unité de Commande et non prises en compte dans le schéma d'exécution du mode 1b.

a1 : Sélection du contenu de RE2 (d) par OPMPX3 (commande 15)

a2 : Sélection du contenu de RE3 (I) par MPX3 (commande 17)

a3 : Chargement de RMA1 avec une partie de contenu de RE1 (T,i,c) + la configuration 00001 sélectionnée par MPX0 (commandes 1 et 2)



Graphe de Fonctionnement du mode 1b

Evaluation du temps de fonctionnement

L'analyse des différents chemins de données a permis de déterminer comme étant le plus long le chemin de données suivant :

A0, A1, A2, A11, A13, A16, A17, A18, A24, A26, A27, A28, A30

Ce qui équivaut au temps :

$22\overline{\tau}$ + TMA1 + TMA2 + TADD2

Dans une technologie de type HMOS, le temps de fonctionnement est donc évalué à 216ns ce qui conduit à considérer l'ordre de grandeur en mode 1b comme étant de 220ns.

5.4.2.3. Le mode 2

Ce mode correspond au cas où l'information (T,i,c) ne se trouve pas en Mémoire Associative mais où l'information (T, i) s'y trouve.

Définition des actions réalisant le mode 2

A0 : Prise en compte du contenu de RE4 par l'Unité de Commande

A1 : Chargement de RMA1 avec le contenu de RE1 -T,i,c + 1 bit de typage- (commandes 1 et 2)

A2 : Recherche associative sur le contenu de RMA1 qui donne lieu à l'envoi d'un signal d'échec vers l'Unité de Commande (commande 3 + commandes internes à la Mémoire Associative)

A3 : Comparaison entre le contenu de RIC (ic) et la partie de RE1 contenant i dans COMP1 (i <=ic ?) qui donne lieu à l'envoi d'un signal vers l'Unité de Commande (commandes 4 et 5)

A4 : Chargement de RMA1 avec une partie du contenu de RE1 (T,i) + la configuration 00001 sélectionnée par MPX0 (commandes 1 et 2)

A5 : Recherche associative sur le contenu de RMA1 qui donne lieu à l'envoi d'un signal de succès vers l'Unité de Commande (commande 3 + commandes internes à la Mémoire Associative)

A6 : Recherche du descripteur qui donne lieu à l'envoi d'un signal de succès vers l'Unité de Commande (commandes internes à la Mémoire Associative)

A7 : Sélection des signaux de sortie par MPX4 pour RMA2 (commandes 6 et 7)

A8 : Contrôles sur le type du descripteur (commandes internes à l'Unité de Commande)

A9 : Sélection du champ "CMAX" du descripteur par MPX1 (commande 9)

A10 : Sélection d'une partie du contenu de RE1 (c) par MPX2 (commande 13)

A11 : Comparaison (c $\leq$ CMAX ?) dans COMP2 qui donne lieu à l'envoi d'un signal vers l'Unité de Commande (commande 14)

A12 : Sélection de la valeur c*8 par OPMPX1 (commande 10)

A15 : Sélection des sorties de ADD1 (dp) par OPMPX3 (commande 15)

A16 : Sélection du masque dans SM (commande 8)

A17 : 2ème masquage dans MASQ validant l'adresse (b) à l'entrée de ADD2 (sans commande)

A18 : Addition (b+dp) dans ADD2 (commande 16)

A19 : Sélection de K' par MPX3 (commande 17)

A20 : Chargement de RS3 par l'Unité de Commande

Définition des actions amorcées par l'Unité de Commande et non prises en compte dans le schéma d'exécution du mode 2.

a1 : Sélection du contenu de RE2 (d) par OPMPX3 et OPMPX1 (commandes 10 et 15)

a2 : Sélection du contenu de RE3 (1) par MPX3 et OPMPX 2 (commandes 11 et 17)

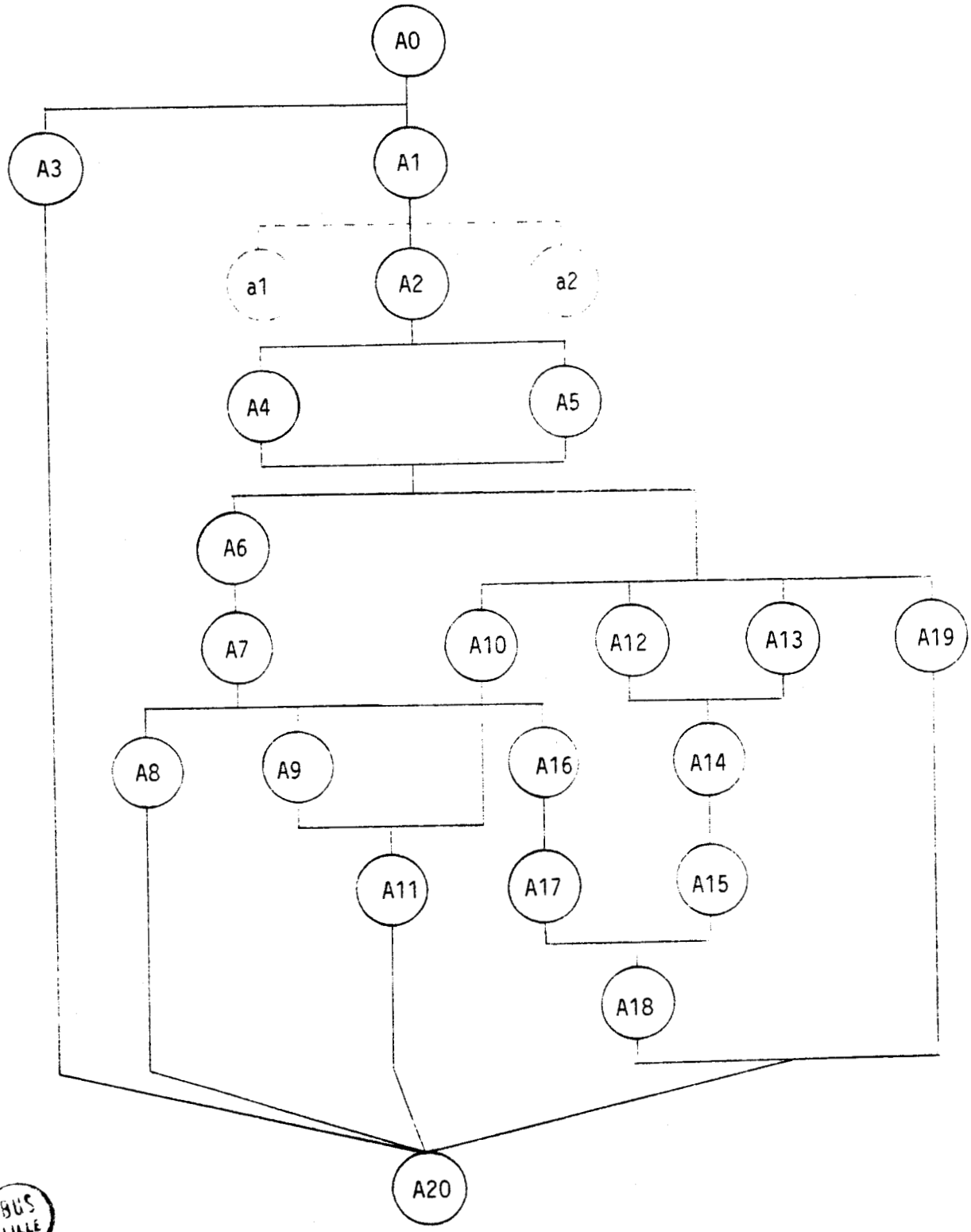
Evaluation du temps de fonctionnement

L'analyse des différents chemins de données a permis de déterminer comme étant le plus long le chemin de données suivant :

A0, A1, A2, A5, A12, A14, A15, A18, A20

ce qui équivaut au temps de :

$7\overline{C}$ + TMA1 + TMA2 + TCOMP2



Graphe de Fonctionnement du mode 2

Dans une technologie de type HMOS, le temps de fonctionnement est donc évalué à 157ns, ce qui conduit à prendre un ordre de grandeur équivalent au mode 1, c'est-à-dire 160ns.

5.4.2.4. Le mode 3

Il s'agit du mode de fonctionnement dans le cas où les 2 recherches associatives (sur T,i,c et sur T,i) se sont révélées infructueuses.

Définition des actions réalisant le mode 3

A0 : Prise en compte du contenu de RE4 par l'Unité de Commande
A1 : Chargement de RMA1 avec le contenu de RE1 -T,i,c + 1 bit de typage- (commandes 1 et 2)
A2 : Recherche associative sur le contenu de RMA1 qui donne lieu à l'envoi d'un signal d'échec vers l'Unité de Commande (commande 3 + commandes internes à la Mémoire Associative)
A3 : Comparaison entre le contenu de RIC (ic) et la partie de RE1 contenant i dans COMP1 (i $\leq$ ic ?) qui donne lieu à l'envoi d'un signal vers l'Unité de Commande (commandes 4 et 5)
A4 : Chargement de RMA1 avec une partie du contenu de RE1 (T,i) + la configuration 00001 sélectionnée par MPX0 B (commandes 1 et 2)
A5 : Recherche associative sur le contenu de RMA1 qui donne lieu à l'envoi d'un signal d'échec vers l'Unité de Commande (commande 3 + commandes internes à la Mémoire Associative)
A6 : Sélection du contenu de RE0 (descripteur de la TNL) par MPX4 et chargement de RMA2 (commandes 6 et 7)
A7 : Sélection du masque dans SM (commande 8)
A8 : 1er masquage dans MASQ (sans commande)
A9 : Reconditionnement du champ "TAILLE" par OPRC (sans commande)
A10 : Sélection du champ "TAILLE" par MPX1 (commande 9)
A11 : Sélection d'une partie de RE1 (i) par MPX2 qui la transforme en i*8 (commande 13)
A12 : Comparaison dans COMP2 (i*8 $\leq$ t ?) qui donne lieu à l'envoi d'un signal vers l'Unité de Commande (commande 14)
A13 : 2ème masquage dans MASQ validant l'adresse (b) à l'entrée de ADD2 (commande 8)

A14 : Sélection d'une partie de RE1 (i) par OPMPX3 qui la transforme en i*8 (commande 15)

A15 : Addition (b + i*8) dans ADD2 (commande 16)

A16 : Sélection de K par MPX3 (commande 17)

A17 : Contrôles sur le type du descripteur (commandes internes à l'Unité de Commande)

A18 : Chargement de RS3 par l'Unité de Commande

Définition des actions amorcées par l'Unité de Commande et non prises en compte dans le schéma d'exécution du mode 3.

a1 : Sélection du contenu de RE2 (d) par OPMPX1 et OPMPX3 (commandes 10 et 15)

a2 : Sélection du contenu de RE3 (I) par OPMPX2 et MPX3 (commandes 11 et 17)

a3 : Sélection d'une partie du contenu de RE1 (c) par MPX2 (commande 13)

a4 : Sélection de la valeur c*8 par OPMPX1 (commande 10)

a5 : Sélection de 0 par OPMPX2 (commande 11)

Evaluation des temps de fonctionnement

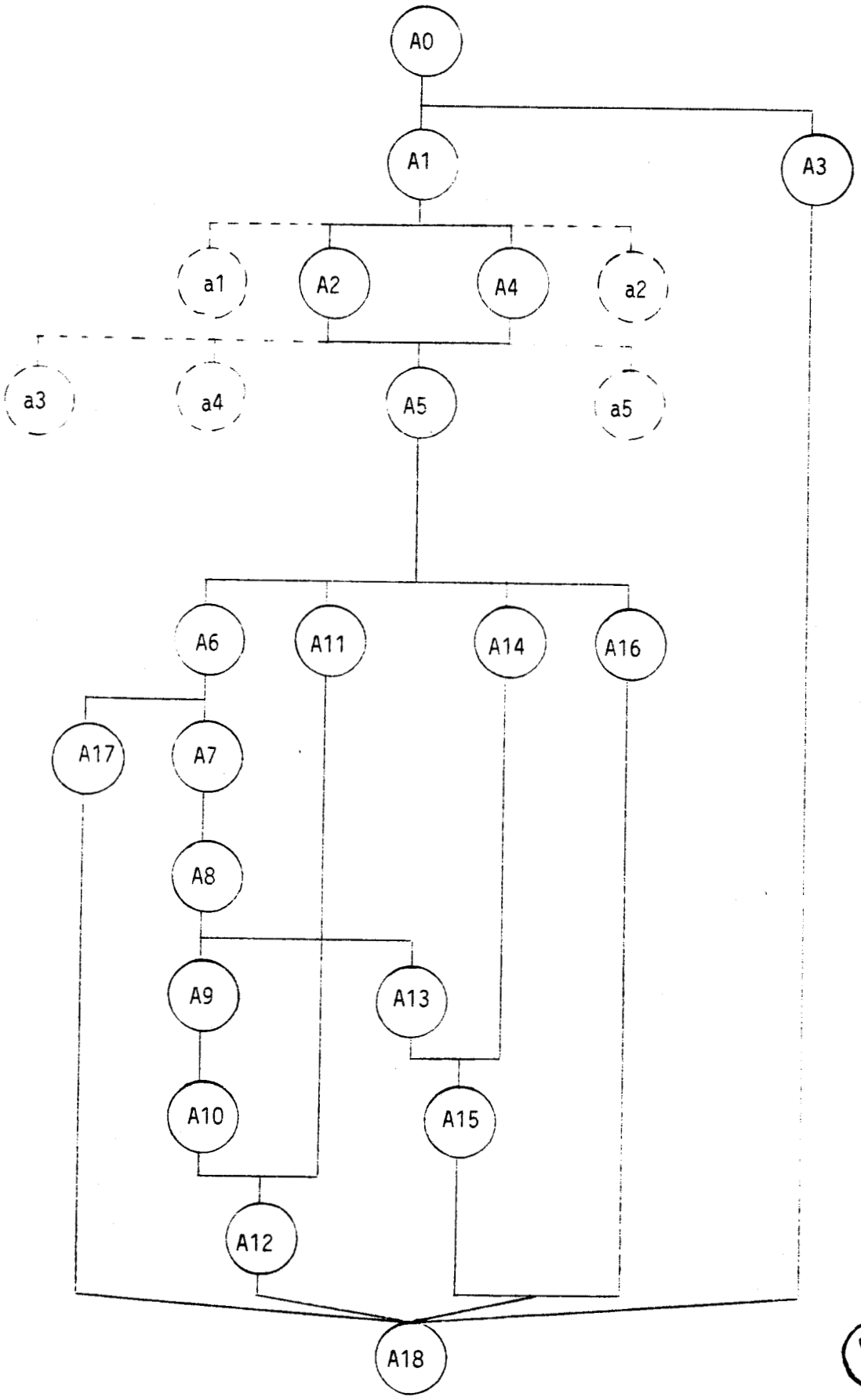
L'analyse des différents chemins de données a permis de déterminer comme étant le plus long le chemin de données suivant :

A0, A1, A2, A5, A6, A7, A8, A9, A10, A12, A18

ce qui équivaut au temps :

$16\overline{C} + 2*TMA1 + TCOMP2$

Dans une technologie de type HMOS, le temps de fonctionnement est donc de 148ns, ce qui conduit à prendre pour ordre de grandeur 150ns.



Graphe de Fonctionnement du mode 3

5.4.3. Evaluation en coût de matériel

Outre une estimation des temps de fonctionnement, il est tentant de vouloir évaluer le coût en matériel qu'engendrerait une telle implémentation.

Il semble même important de procéder à une telle évaluation lorsque l'objectif est, à terme, la réalisation en VLSI.

Malheureusement, ce type de travail se heurte à deux ensembles d'obstacles qui peuvent compromettre plus ou moins la validité des résultats obtenus à l'issue du travail d'évaluation ; il s'agit principalement :

- D'une part, de difficultés inhérentes au VLSI ; la conception d'un VLSI est en effet soumise à de nombreuses règles de dessin qui rendent difficile l'établissement a priori d'une correspondance entre un schéma logique et son équivalent intégré dans le silicium, même si le schéma logique est réalisé à partir d'éléments de base définis par le fabricant de circuits (problèmes de placement et de routage) ;

- D'autre part, de difficultés particulières à l'application ; à ce stade de sa conception, le T.A.L. offre des inconvénients certains à l'élaboration de son évaluation en coût de matériel, en l'occurence en surface occupée sur circuits de type HBMOS4 : son schéma logique est incomplet (la partie commande n'est pas définie), non optimisé (emploi de portes restreint aux ET et OU) et surtout n'utilise pas des composants élémentaires fournis par le fabricant de VLSI (cellules et macrocellules).

Une forme d'évaluation a cependant été retenue et revêt les aspects suivants :

- Les évaluations concernant les modules matériels classiques (registres, additionneurs, comparateurs) ont été fournies par le fabricant ;

- Les évaluations concernant les modules spécialement conçus pour le T.A.L. (multiplexeurs, masques,...) comptabilisent le nombre de portes ET ou OU utilisées, sans tenir compte des optimisations possibles ;

- Chaque dénombrement est multiplié par un terme correcteur fourni par le constructeur suivant le type d'éléments dénombrés (portes logiques, points mémoires,...) afin d'obtenir une évaluation en unités de surface ;

- L'unité de surface employée par le constructeur de VLSI est le **pas de grille** (noté L/g) qui correspond au mode d'implantation des masques en technologie HBMOS4.

Les limites de cette évaluation concernent naturellement la fonction de commande qui n'est pas définie ; compte tenu du faible nombre de micro-commandes et probablement d'états logiques, une approximation basée sur des analogies avec d'autres modules similaires permet d'avance le chiffre de 200 portes logiques constituant un système combinatoire câblé.

Les facteurs d'évaluation en nombre de pas de grille correspondant aux différents éléments comptabilisés sont les suivants :

Point mémoire RAM	: 2 à 3	
Point Registre	: 11	
Porte ET	: 6	moyenne 5
Porte OU	: 4	

Les estimations fournies par les fabricants de circuits HBMOS4 concernent :

- Un additionneur 40 bits : 1200 L/g
- Un additionneur 32 bits : 800 L/g
- Un comparateur 32 bits : 150 L/g
- Un comparateur 8 bits : 30 L/g
- 338 points registres : 3718 L/g

Le total fait donc : 5898 L/g

Les estimations concernant la Mémoire Associative ont déjà été effectuées (§ 5.3.2.6.4.) ; le résultat des calculs a donné le nombre suivant : 16 797 L/g.

L'ensemble de la logique spécifiquement étudiée pour le T.A.L. (§ 5.3.2), sauf la Mémoire Associative, regroupe 1 079 portes ; son équivalent en pas de grille est donc de 5 395 L/g.

L'Unité de Commande estimée à 200 portes logiques occupe donc une surface approximative de 2 200 L/g.

Sans tenir compte de la place utilisée pour les problèmes de routage, le T.A.L. demande la surface de 30 290 L/g.

Une approximation de la surface de routage à 15 % donne le chiffre définitif de l'ordre de : 34 800 L/g.

CHAPITRE 6

CONCLUSION

6. CONCLUSION

Les options architecturales que nous avons adoptées pour la machine langage ADA offrent de sérieux atouts pour la réalisation des objectifs d'efficacité, de sûreté de fonctionnement et de facilité d'utilisation auxquels s'est attaché le projet MLI ; la première partie de l'étude a permis en effet de justifier, sinon de valider, les choix de langage machine évolué et d'intégration dans le matériel de mécanismes sophistiqués concourant à la réalisation du nouveau cahier des charges et à la réduction des fossés sémantiques traditionnellement engendrés par les architectures (trop) classiques.

La description de l'évolution de la microprogrammation montre combien les techniques qui s'en réclament semblent appropriées à la définition d'une méthode d'implémentation matérielle privilégiée qui assurera la concrétisation des principes de réalisation de l'architecture logique adoptée.

Si l'approche "machine-langage" telle qu'elle est définie dans ce rapport, apparaît ainsi globalement démontrée, la définition du T.A.L. permet quant à elle, de valider a priori l'adressage de type lexical ; la décomposition des instructions machines effectuée dans [RDL 82] et basée sur une exécution de type pipe-line conduit en effet, grâce aux évaluations en temps de fonctionnement du T.A.L. dans ses différents modes, à des estimations de temps d'exécution d'instructions compatibles avec les objectifs de performances souhaités.

Evolution du projet

En ce qui concerne l'orientation définitive de l'architecture matérielle, celle-ci reste fondamentalement soumise aux choix technologiques arrêtés en la circonstance : fabrication de circuits VLSI personnalisés (approche retenue a priori pour MLI), utilisation de réseaux prédiffusés ou de microprocesseurs en tranches.

Quelle que soit l'orientation choisie, le travail de définition de l'architecture matérielle s'articulera autour de deux aspects majeurs : la simulation et la réalisation physique des circuits.

Si, dans le cas des réseaux pré-diffusés ou des microprocesseurs en tranches, le choix de la simulation est souvent fonction des moyens disponibles, de la disponibilité voire de la "culture informatique" du concepteur de machines, il est le plus souvent imposé dans le cas des circuits VLSI où chaque constructeur possède ses propres outils de définition.

De façon générale, les outils de conception/ description/simulation permettant une transition (relativement) automatique ou formalisée avec les outils de fabrication sont issus des efforts réalisés dans la C.A.O.-VLSI, comme en témoigne l'exemple de CASCADE [Mer 83].

Dans le cas où les outils de simulation (tout au moins pour les niveaux de conception les plus hauts) ne seraient pas imposés, le choix resterait difficile dans la mesure où aucun langage de description/simulation suffisamment général n'a réussi à s'imposer véritablement.

Choix d'un langage de description/simulation

Cet état de fait a d'ailleurs donné naissance il y a une dizaine d'années au projet CONLAN [PBB 80], qui avait pour objectif de créer un tel langage ; les travaux ont finalement abouti à la spécification, non pas d'un langage, mais d'une famille de langages possédant un noyau syntaxique commun dont chaque élément est particularisé suivant le niveau de description à l'aide d'un procédé uniforme d'extension, à partir d'un ensemble commun de notions primitives.

Les caractéristiques ainsi établies des langages dérivés de CONLAN ont apparemment suffisamment de similitude avec celles du langage ADA (type, sous-type, encapsulage, généricité,...) pour que, au cas où l'approche CONLAN ne pourrait être retenue, les premières phases de description/simulation soient envisageables en ADA ; outre les possibilités que devrait offrir l'environnement Compagnie (réalisation prochaîne d'un compilateur ADA sur DPS/7), cette solution aurait pour avantage d'établir une cohérence certaine entre tous les aspects du projet.

Perspectives à court terme

De manière plus précise, la spécification de l'architecture matérielle devrait se poursuivre avec une première approche du schéma d'exécution des instructions que les impératifs d'efficacité semblent nous amener à envisager de type pipe-line.

L'exécution du code devra prendre en compte également la possibilité d'optimisation dynamique de l'exécution des programmes en cours de définition par Agnès BRADIER [Bra 83].

Le choix de la microprogrammation devrait par ailleurs donner une certaine souplesse à la définition du décodeur afin de supporter les modifications du langage machine que ne manqueront pas d'engendrer les premières simulations, voire les premières statistiques.

Les différents éléments qui constituent cette étude (réflexions, argumentation et propositions en faveur d'une nouvelle orientation à la conception d'architecture) ont eu pour fonction d'installer le plus rigoureusement possible le cadre et les structures qui vont permettre au projet de rentrer dans une phase plus active de réalisation.

Le respect fidèle de la philosophie d'implémentation matérielle présentée dans ce rapport devrait, nous l'espérons, permettre de valider de manière satisfaisante les options prises par ailleurs dans la définition de l'architecture logique.

BIBLIOGRAPHIE

Abr 82 — ABRIAL J.R.
"Formal Programming (mars 82) et A Theoretical Foundation to Formal Programming (mai 82)"
Publications à paraître

Age 76 — AGERWALA T.
"Microprogram Optimization : A Survey"
IEEE Transactions on Computers C25 (10), p.962-973, oct. 76

Anc 83a — ANCEAU F.
"Architectures de Machines VSLI Spécialisées"
5ème Journée Francophone sur l'Informatique, Genève, p.27-36, janv. 83

Anc 83b — ANCEAU F.
"Real Assumptions concerning I.C. Design"
Cours INRIA - Conception de Circuits Intégrés p.1-6, juin 83

Bak 78 — BAKER H.G. Jr
"List Processing in Real Time on a Serial Computer"
CACM 21, 4, avril 78

Bal 69 — BALZER R.M.
"EXDAMS - Extendable Debugging and Monitoring System"
AFIPS Conf. Proc. Volume 34 AFIPS Press 69

Bar 81 — BARTLETT J.F.
"MicroTAL - A Machine-Dependent High Level Microprogramming Language"
14th Annual Microprogramming Workshop, p.109-114, Chatham oct 81

BBC 80 — BELL R.K., BELL W.D., COOPER T.C., Mc FARLAND T.K.
"The Big Three - Today's 16 Bits Microprocessor"
13th Annual Microprogramming Workshop, p.126-138, Colorado Sprints - déc. 80

BCB 74 — BELL J., CASADENT D., BELL D.
"An Investigation of Alternative Cache Organizations"
IEEE Transaction on Computer C23 (4), avril 74

BCh 79 — BATTAREL G.J., CHEVANCE R.J.
"Design of a High Level Language Machine"
National Computer Conference 79

BDF 81 — BEYERS J.W., DOHSE L.J., FUCETOLA J.P., KOSHIS R.L., LOB C.D., TAYLOR G.L., ZELLER E.R.
"32 Bits VSLI CPU Chip"
Proc. IEEE Inter. Solid State Circuits Conf., p.104-105 NY, fév. 81

BEL 75 — BOYER R.S., ELPAS B., LEVITT K.N.
"A Formal System for Testing and Debugging"
Intern. Conf. on Reliable Software, p.234-245, avril 75

BGe 79 — BIRE C., GERBER R.
"Microprocesseur en Tranches"
Ed. Techn. & Doc., Paris 79

BLW 79 — BRANQUART P., LOUIS G., WODON P.
"Towards a Quasi Formal Description of CHILL - A Sample"
CCITT Working Document, mai 79

BMe 73 — BROCA F.R., MERWIN R.E.
"Direct Microprogrammed Execution of Intermediate Text for HLL Computer"
Proc. of the ACM, Annual Conference, 73

BPa 80 — BAKER A.G. Jr, PARKER C.
"High Level Language Programs Run 10 Times Faster in Micro-store"
13th Annual Microprogramming Workshop, p.171-177, 80

Bra 83 — BRADIER A.
"Mécanismes matériels réalisant à l'exécution l'optimisation des Programmes"
Thèse 3ème Cycle, Lille, sept. 83

Bri 75 — BRINCH HANSEN
"The Programming Language Concurrent Pascal"
International Summer School on Language Hierarchies and Interfaces, Munich, Germany, juil. 75

Bro 75 — BROADBENT J.K.
"High Level Language Implementation Through Microprogramming and Systems Architecture"
Infotech State of the Art Report 23, p.337-357, 75

Bry 82 — BRYGIER J.
"Microprogrammation et Machines Microprogrammées"
Rapport interne CII-HB, Serv. Transf. Programmes, janv. 82

BSO 75 — BROADBENT J.K., SUFRIN B., OUTRED C.F.J., et al.
"Microprogramming : Language Support"
Microprogramming and Systems Architecture, Infotech State of the Art Report 23, p.45-91, 75

Bur 77 — BURR W.E. et al.
"Computer Family Architecture Via Test Programs"
ECOM-4528, Army Electronics Command, Fort Monmouth, NJ, 77

Car 79 — CARREZ C.
"La Protection dans un Système et son Expression dans les Langages de Programmation"
Panorama des Langages d'Aujourd'hui, Cargèse, AFCET, Bulletin Groplan n°8, 79

Cas 80 **CASTAN M.**
"Conception et Réalisation d'une Machine Spécialisée dans le Traitement des Formes Arborescentes"
Thèse de 3ème Cycle, Toulouse, 80

Cha 75 **CHAPTAL DE CHANTELOUP**
"Problems of Microprogram Production"
Microprogramming and Systems Architecture - Infotech State of the Art Report 23, p.241-259, 75

Che 73 **CHEVANCE R.J.**
"A Cobol Machine"
Interface SIGPLAN/SIGMICRO Harriman, NY, juin 73

Chi 80 **CHIN-KUEI Cho**
"An Introduction to Software Quality Control"
Ph.D., The MITRE Corporation and the Georges Washington University, A Wiley-Interscience Corporation, 80

Chu 75 **CHU Y.**
"Concepts of a High Level Language Computer Architecture"
Proc. ACM - Conf. Mineapolis, MN, oct. 75

CIM 78 **CIMSA**
"Manuel de Reference LTR"
(Mise à jour oct. 79) Ref. 5616/U2/FR, juin 78

CJe 75 **COHEN E., JEFFERSON D.**
"Protection in the Hydra Operating System"
Proceedings of the 5th Symposium on Operating System Principles, NY : ACM p.141-160, 75

Coh 81 **COHEN J.**
"Garbage Collection of Linked Data Structures"
Computer Survey 13,3, 81

Cor 83 **CORDONNIER V.**
"Noyau pour l'Ecriture de Microprogrammes Parallèles"
Rapport ERA 771, Publ. n°25, Lille, fév. 83

Cos 74 **COSSERAT D.C.**
"A Data Model Based on the Capability Protection Mechanism"
International Workshop on Protection in Operating Systems, 74

Cre 71 **CREECH B.**
"Architecture of the B6500"
Infotech Report 2, 71

CRO 75 **CROCUS**
"Système d'Exploitation des Ordinateurs"
Ed. Dunod Informatique, 75

Das 77 DASGUPTA S.
"Parallelism in Loop-Free Microprograms"
Information Processing 77, North Holland, p.745-750, 77

Das 78 DASGUPTA S.
"Towards a Microprogramming Language Schema"
11th Annual Microprogramming Workshop, p.144-153, 78

DBo 76 DEUTSCH L.P., BOBROW D.G.
"An efficient Incremental Automatic Garbage Collector"
CACM 19,9, sept. 76

Den 76 DENNIS J.B.
"Computer Architectures and the Cost of Software"
Computer Architectures News, 5 (1), p.17-21, 76

Dew 76 DEWITT D.J.
" A Machine Independent Approach to the Production of Optimal Horizontal Microcode"
Ph. D., Université du Michigan, juin 76

Dij 78 DIJKSTRA E.W.
"On-the-Fly Garbage Collection : An exercise in Cooperation"
CACM 21, 11, nov. 78

Dit 80 DITZEL D.R.
"Retrospective on High Level Language Computer Architecture "
the 7th Annual Symposium on Computer Architecture
Conference Proceedings IEEE, 80

DMi 74 DENNIS J.B., MISUNAS D.P.
"A Preliminary Architecture for a Basic Data Flow Processor"
Proc. ACM SIGARH, IEEE Symposium on Computer Architecture, déc. 74

DSh 78 DAVIDSON S., SHRIVER B.
"An Overview of Firmware Engineering"
Computer, Vol. 11, n°5, mai 78

DVa 66 DENNIS J.B., VAN HORN E.C.
"Programming Semantics for Multiprogrammed Computers"
Communications of the ACM, 9 (3), p.143-155, 66

Ear 80 EARNEST E.D.
"Twenty Years of Burroughs High Level Language Machine"
Proc. of the Intern. Workshop on HLL Computer Architecture, University of Maryland, College Park, MD, p.212-219, 80

Eng 72 ENGLAND D.M.
"Architectural Features of System 250"
Infotech State of the Art Report 14 on Operating Systems, 72

Eng 74 ENGLAND D.M.
"Capability Concept Mechanisms and Structure in System 250"
Intern. Workshop on Protection in Operating Systems, 74

Feu 72 **FEUSTEL E.A.**
"The Rice Research Computer : A Tagged Architecture"
proc. AFIPS SJCC, Vol. 40, Press, Montvale, p.369-377, 72

Feu 73 **FEUSTEL E.A.**
"On the Advantages of Tagged Architecture"
IEEE Transactions on Computer, C.22(7), p.644-656, 73

Fir 80 **FIRTH N.R.**
"The Role of Software Tools in the Development of the ECLIPSE MV/8000 Microcode"
13th Annual Microprogramming Workshop, p.54-58, 80

Fis 79 **FISCHER J.A.**
"The Optimization of Horizontal Microcode Within and Beyond Basic BLocks : An Application of Processor Scheduling with Ressources"
Ph. D., University of N.Y., oct. 79

FLa 76 **FERRIE J., LANCIAUX D.**
"Le Système Plessey 250"
Rapport Laboria n°168, 76

Flo 67 **FLOYD R.W.**
"Assigning Meanings to Programs"
Proc. Symposia in Applied Mathematics 19
American Math. Soc., p.19-32, 67

Flo 71 **FLOYD R.W.**
"Towards the INteractive Design of Correct Programs"
Proc. IFIP COngress, Ljubljana, 71

FNo 78 **FITCH J.P., NORMAN A.C.**
"A Note on Compacting Garbage Collection"
Computer Journal, Vol. 21, n°1, p.31-34, fev. 78

Fri 75 **FRIEDER G.**
"Microprogramming and Operating Systems"
Microprogramming and Systems Architecture, Infotech State of the Art Report 23, p.391-407, 75

Fri 76 **FRIEDMAN**
"Garbage Collecting a Heap which Includes a Scatter Table"
INF. PROCESS. LETT. 5,6, p.161-164, déc 76

FRo 71 **FLYNN M.J., ROSIN R.F.**
"Microprogramming : An Introduction and a Viewpoint"
IEEE Transaction on Computer, Vol. C-20, p.727-731, juil. 71

Gag 73 **GAGLIARDI U.O.**
"Report of Workshop 4 - Software Related Advances in Computer Hardware"
Proceedings of a Symposium on the High Cost of Software
Menlo Park, CA : Standford Research Institute p.99-120, 73

GLa 81 GEYER S., LAKE A.
"Development Tools for User Microprogramming"
14th Annual Microprogramming Workshop, p.74-77, oct.81

Gol 72 GOLDSTINE H.H.
"The Computer from Pascal to von Neumann"
Princeton University Press, 72

Gre 81 GREENBERG K.F.
"The Micro 8 Microcode Assembler"
14th Annual Microprogramming Workshop p. 78-82, oct. 81

Gut 80 GUTTAG K.M.
"Compressing Control ROM for VLSI Microprogrammed Microprocessors"
13th Annual Microprogramming Workshop, p.115- 121, déc 80

Guy 83 GUYOT A.
"L.U.C.I.E. : Langage Universitaire pour les Circuits Intégrés pour l'Enseignement"
Laboratoire IMAG Grenoble, Cours INRIA : Conception de Circuits Intégrés, juin 83

Ham 81 HAMMERSTROM D.
"A Panel Discussion on How can we write working microprograms - Past experiences, future prospects"
14th Annual Microprogramming Workshop, 81

Har 79 HART J.J.
"The Advanced Interactive Debugging System (AIDS)"
Sigplan Notices, Vol. 14, n°12, p.110-121, déc. 79

Her 78 HERBERT A.J.
"A New Protection Architecture for the Cambridge Capability Computer"
Operating System Review, 12 (1), p.24-28, 78

HHT 81 HOBSON R.F., HANNON P., THORNBURG J.
"High Level Microprogramming with APL Syntax"
14th Annual Microprogramming Workshop, 81

HJe 81 HOCKNEY R.W., JESSHOPE C.R.
"Parallel Computers"
Adam Hilgner Ltd, Bristol, 81

HJP 82 HENNESSY J,JOUPPI N., PRZYBYLSKI S. et al.
"MIPS : A Microprocessor Architecture"
15th Annual Microprogramming Workshop, 82

HKa 82 HOLTKAMP B., KAESTNER H.
"A Firmware Monitor to Support Vertical Migration Decisions in the UNIX Operating System"
15th Annual Microprogramming Workshop, p.153-162, 82

Huf 52 **HUFFMAN D.A.**
"A Method for the Constructions for Minimum Redundancy Codes"
Proc. IRE 40, sept 52

Hus 70 **HUSSON S.S.**
"MICROPROGRAMMING - Principles and Practices"
Prentice Hall, INC, 70

HWa 67 **HADDON B.K., WAITE W.M.**
"A Compaction Procedure for Variable Length Storage Elements"
Computer Journal, 10, 2, p. 162-165, 67

IBM 78 **IBM System/38**
"Introduction" et "Technical Developments"
Publication IBM, 78

ICa 78 **IBBETT R.N., CAPON P.C.**
"The Development of the MU5 Computer System"
Communications of the ACM, 21 (1) p.13-24, 78

Ili 68 **ILIFFE J.K.**
"Basic Machine Principles"
American Elsevier, N.Y., 68

INT 81 **INTEL iAPX 432**
"General Data Processor Architecture Reference Manual"
"Component User's Guide"
"Interface Processor Architecture Reference Manual"
Intel Corp., Santa Clara, CA, 81

Kav 80 **KAVIPURAPU K.M.**
"The Design of Architectures to Reduce Semantic Gap"
Ph. D. Thesis, SMU, 80

KBB 82 **KAVI K., BELKHOUCHE B., BULLARD E. et al**
"HLL Architectures : Pitfalls and Predilections"
Computer Architecture, 82

Kin 76 **KING J.C.**
"Symbolic Execution and Program Debugging"
Communications of the ACM, Vol.19, n°7, p.385-394, juil. 76

Kop 76 **KOPLIN M.R.**
"M138/M148 Performance Summary"
GUIDE Presentation, Juil. 76

KRa 71 **KLEIR R.L., RAMAMOORTHY C.V.**
"Optimization Strategies for Microprograms"
IEEE Transactions on Computer, p.783-794, juil. 71

Lan 78 **LANCIAUX D.**
"Mécanismes de Structuration des Systèmes à Capacités"
Thèse d'Etat, Lille, 78

Lam 71 **LAMPSON B.W.**
"Protection
Proc. Fifth Annual Princeton Conf. on Information Science and Systems, Princeton University, 71

LDS 80 **LANDSKOV D., DAVIDSON S., SHRIVER B. et MALETT P.W.**
"Local Microcode Compaction Techniques"
ACM Computing Surveys 12(3), p.261-294, sept. 80

Lec 77a **LECUSSAN B.**
"Emulation Généralisée : Détermination d'Outils Matériels Spécialisés dans L'Interprétation de Langages Evolués"
Thèse 3ème Cycle, Toulouse, mars 77

Lec 77b **LECOUFFE P.**
"Etude et Définition d'une Machine Langage LISP"
Thèse de 3ème Cycle, Lille, déc. 77

Lec 82 **LECUSSAN B.**
"Proposition de Définition d'une Architecture de Machine-Système de Programmation pour des Applications Temps-Réel"
Thèse d'Etat, Toulouse, déc. 82

LHe 80 **LIEBERMAN H. et HEWITT C.**
"A Real Time Garbage Collector that Can Recover Temporary Storage Quickly"
MIT Publications, 80

Li 82 **LI T.**
"A VLSI View of Microprogrammed System Design"
15th Annual Microprogramming Workshop, p.96-104, 82

LIt 75 **LANE C.J. et ITO M.R.**
"MPL-85 : A High Level Microprogramming Language Compiler"
Fall Compcon 75, Dig. Papers, p.49-52, 75

LSt 76 **LAMPSON B.W. et STURGIS H.E.**
"Reflections on an Operating System Design"
Communications of the ACM, 19(5), p.251-265, 76

Luk 80 **LUKEY F.J.**
"Understanding and Debugging Programs"
Int. J. Man-Machine Studies, (GB), 12(2), p.385-394, fév. 80

Mal 78 **MALLETT P.W.**
"Methods of Compacting Microprograms"
Ph.D., Université Louisiane Sud, déc. 78

Mar 81 **MARWEDEL P.**
"A Retargetable Microcode Generation System for a High Level Microprogramming Language"
14th Annual Microprogramming Workshop, p.115-123, 81

Mck 67 **Mc KEEMAN W.M.**
"Language Directed Computer Design"
Proceedings of the 1967 Fall Joint Computer Conf. Washington : Thompson, p.413-417, 67

Mer 83 **MERMET J.**
"Conception Assistée de Systèmes et Circuits Analogiques et Digitaux Electroniques : CASCADE"
Laboratoire IMAG, Grenoble, 83

Mey 80 **MEYERS W.J.**
"Design of a Microcode Link Editor"
13th Annual Microprogramming Workshop, p.165-170, 80

Min 63 **MINSKY**
"A LISP Garbage COllector Algorithm Using Serial Secondary Storage"
MIT Publications, oct. 63

MPu 65 **MELBOURNE A.J. et PUGMIRE J.M.**
"A Small Computer for the Direct Processing of FORTRAN Statements"
Computer Journal, 8(1), p.24-27, avril 65

Mul 76 **MULLER**
"On the Feasibility of Concurrent Garbage Collection"
Ph.D. Thesis, Techn. Hogeschool, mars 76

Mye 77 **MYERS G.J.**
"The Design of Computer Architectures to Enhance Software Reliability"
PH.D. Dissertation, Polytechnic Inst. N.Y., 77

Mye 80a **MYERS G.J.**
"SWARD-A Software Oriented Architecture"
Proceedings of the Intern. Workshop on High Level Language Computer Architecture. University of Maryland, p.163-168, 80

Mye 80b **MYERS G.J. et al.**
"A Hardware Implementation of Capability-Based Adressing"
Operating Systems Review, 14(4) p.13-25, 80

Mye 82 **MYERS G.J.**
"Advances in Computer Architecture - Second Edition"
Wiley Interscience Publication, 82

NAJ 75 **NORI K.V., AMMAN U. JENSEN K. et NAGELI H.**
"The Pascal (P) Compiler Implementation Notes"
Int. für Informatik, Eidgenössische Technische Hochschule, Zürich, 75

NFi 75 **NICOLAU A., FISCHER J.A.**
"Using an Oracle to Measure Potential Parallelism in SIngle INstruction Stream Program"
14th Annual Microprogramming Workshop, p.171-182, 81

Nog 83 **NOGUEZ G.**
"Le Système EMILIE"
Université Paris VI, Cours INRIA : Conception de Circuits Intégrés, Juin 83

NWa 77 **NEEDHAM R.M., WALKER R.D.H.**
"The Cambridge CAP Computer and its Protection System"
Proceedings of the 6th Symposium on Operating System Principles, N.Y. : ACM, p.1-10, 77

Opl 67 **OPLER A.**
"Fourth Generation Software"
Datamation, 13, p.22-24, janv. 67

Pat 81 **PATTERSON D.A.**
"An Experiment in High Level Language Microprogramming and Verification"
Comm. of the ACM, 24(10) p.699-709, oct. 81

PBB 80 **PILOTY R., BARBACCI M., BORRIONE D., DIETMEYER D., HILL F., SKELLY P.**
"An Overview of CONLAN - A Formal Construction Method for Hardware Description Languages"
Proc. IFIP Congress 1980, Tokyo et Melbourne, oct. 80

PER 81 **PERQ** "System Software"
Reference Manual, juil. 81

Per 81 **PERCEBOIS C.**
"Etude et Réalisation d'un Environnement de Microprogrammation Spécialisé dans le Traitement des Formes Arborescentes"
Thèse 3ème Cycle, Toulouse, 81

PGa 79 **PAIR C. et GAUDEL M.C.**
"Les Structures de Données et leur Représentation en Mémoire"
Deuxième Edition, INRIA, déc. 79

PGS 81 **POE M.D., GOODELL R., STEELY J.S.**
"Issues of the Design of a Low Level Microprogramming Language for Global Microcode Compaction"
14th Annual Microprogramming Workshop, p.88-94, 81

Poe 80 **POE M.D.**
"Heuristics for the Global Optimization of Microprograms"
13th Annual Microprogramming Workshop, p.13-22, déc. 80

PSe 81 **PATTERSON D.A. et SEQUIN C.H.**
"RISC I : A Reduced Instruction Set VLSI Computer"
Proc. of the 8th Annual Symposium of Computer Architecture Minneapolis, mai 81

RAd 80 **RAUSCHER T.G. et ADAMS P.M.**
"Microprogramming : A Tutorial and Survey of Recent Developments"
IEEE - Transactions on Computers, 29(1), janv. 80

Ram 74 **RAMAMOORTHY C.V.**
"A Survey of the Status of Microprogramming"
Advances in Information Systems Science, vol. 5, Plenum Press, 74

RCh 72 **RAMAMOORTHY C.V. et CHANG L.C.**
"System Modelling and Testing Procedures for Microdiagnostics"
IEEE Transactions on Computer, 21(11), p.1169-1183, nov. 72

RDL 82a **RENVOISE C, DOUSPIS P., LEGOUX M., BRADIER A., BRYGIER J.**
"Machine Langage ADA - Eléments d'Architecture, Rapport de Recherche CII Honeywell Bull, oct. 82

RDL 82b **RENVOISE C., DOUSPIS P., LEGOUX M., BRADIER A., BRYGIER J.**
"Machine Langage ADA - Eléments de Spécification de l'Architecture Matérielle"
Rapport de Recherche CII Honeywell Bull, déc. 82

RDo 81a **RENVOISE C., DOUSPIS P.**
"Etude d'un Schéma d'Adressage Protégé Adapté à l'Exécution de Programmes ADA"
Rapport de Recherche CII HOneywell Bull, juin 81

RDo 81b **RENVOISE C., DOUSPIS P;**
"Machine Langage ADA - Etude des Instructions Machine Adaptées"
Rapport de Recherche CII Honeywell Bull, déc. 81

Ren 76 **RENVOISE C.**
"Une Approche du Problème de la Sécurité des Systèmes Informatiques"
Rapport 43, Service Technique Central du Chiffre, 76

Ren 80 **RENVOISE C.**
"Méthodes booléennes d'Analyse de Programmes : Application à l'Etude de leur Cohérence"
Thèse d'Etat, Université Paris VI, juin 80

Ren 82 **RENVOISE C.**
"Data Flow Macroscopique - Présentation Générale"
Rapport de Recherche CII Honeywell Bull, juil. 82

RSm 71 **RICE R., SMITH W.R.**
"SYMBOL - A Major Departure from Classic Software Dominated by von Neumann Computing Systems"
Proc. of the 1971 Spring Joint Computer Conf. Montvale, NJ : AFIPS, p.575-587, 71

RTs 74 **RAMAMOORTHY C.V. et TSUCHIYA M.**
"A High Level Language for Horizontal Microprogramming"
IEE Transactions on Computers, 23(8), p.791-801, août 74

RVa 77 **RAMSEYER R.R. et VanDAM A.**
"A Multimicroprocessor Implementation of a General Purpose Pipe-Lined CPU"
Proc. 4th Annual Symposium Computer Architecture, p.29-34, mars 77

RWi 81 **ROSKOS J.E. et WINNER R.I.**
"Towards User Sharing of the Microprogramming Level Under UNIX on the PERKIN-ELMER 3220"
14th Annual Microprogramming Workshop, p.67-73, 81

Sch 77 **SCHOELLKOPF J.P.**
"Machine PASC-HLL : Définition d'une Architecture Pipe-Line pour une Unité Centrale Adoptée au Langage PASCAL"
Thèse 3ème Cycle, Grenoble, juin 77

Sch 78 **SCHOICHET S.R.**
"The LISP Machine"
Mini-Micro Systems, 11(5), p.68-74, 78

SGi 81 **SHERAGA R.J. et GIESER J.L.**
"Automatic Microcode Generation for Horizontally Microprogrammed Processors"
14th Annual Microprogramming Workshop, p.154-168, 81

Sha 78 **SHAPIRD M.D.**
"The Criterion Cobol System"
Proc. of the 1978 NCC, Montvale, NJ : AFIPS, p.1049-1054, 78

Sin 80 **SINT M.**
"A Survey of High Level Microprogramming Languages"
13th Annual Microprogramming Workshop, p.141-153, 80

Spi 73 **SPIER M.J.**
"A Model Implementation for Protective Domains"
Inter. Journal of Computer and Information Science 2(3), p.201-229, 73

Ste 75 **STEELE G.L. Jr**
"Multiprocessing Compactifying Garbage Collection"
CACM, 18(9), sept. 75

SVa 78 **STOCKENBERG J. et VanDAM A.**
"Vertical Migration for Performance Enhancement in Layered Hardware/Firmware/Software Systems"
Computer, 11(5), p.35-50, mai 78

Tal 75 **TALLMAN P.H.**
"Virtual Machine Assist Feature Microcode Implementation"
Microprogramming and Systems Architecture, Infotech State of the Art, Report 23, p.527-540, 75

TOf 78 **TANG T. et O'FLAHERTY K.**
"Virtual Machines and the NCR Criterion"
Datamation, 24(4), p.129-134, 78

Tou 79 **TOURSEL B.**
"Contribution à l'Etude de l'Architecture des Ordinateurs : Propositions pour un Nouveau Mode de Fonctionnement"
Thèse d'Etat, Lille, sept. 79

Tre 82 TREDENNICK N.
"The "Cultures" of Microprogramming"
15th Annual Microprogramming Workshop, p.79-83, 82

USA 83 Reference Manual for the ADA Programming Language
United States Department of Defence, fév. 83

Veg 82 VEGDAHL S.R.
"Phase Coupling and Constant Generation in an Optimizing Microcode Compiler"
15th Annual Microprogramming Workshop, p.125-133, 82

Vui 83 VUILLEMIN J.
"La Révolution Informatique dans la Conception et la Fabrication des Circuits Electroniques"
5èmes Journées Francophones sur l'Informatique, Genève, p.27-36, janv. 83

Wan 82 WANG D.T.
"Defensive Microprogramming"
15th Annual Microprogramming Workshop, p.84-88, 82

Wei 80 WEIDNER T.G.
"CHAMIL- A Case Study in Microprogramming Language Design"
SIGPLAN Notices, 15(1), p.156-166, janv. 80

Wil 51 WILKES M.V.
"The Best Way to Design and Automatic Calculating Machine"
Manchester University-Computer Inaugural Conf., London, 51

Wil 72a WILNER W.T.
"Design of the B1700"
AFIPS FJCC, 72

Wil 72b WILNER W.T.
"B1700 Memory Utilisation"
Ph.D., AFIPS FJCC, 72

Wil 76 WILLEN D.
"An Intel 3000 Cross Assembler"
SIGMICRO Newsletter, vol.7, p.87-95, déc. 76

Wil 82 WILKES M.V.
"KEYNOTE ADDRESS : The Processor Instruction Set"
15th Annual Microprogramming Workshop, p.3-5, 82

Woo 79 WOOD G.
"Global Optimization of Microprograms Through Modular Control Constructs"
12th Annual Microprogramming Workshop, p.1-6, nov. 79

Wul 75 WULF W.A.
"The Influence of High Level Languages on Microprocessor Design"
Microprogramming and Systems Architecture, Infotech State of the Art Report 23, p.225-240, 75