Conception d'un poste de travail APL et de son unité de gestion mémoire

Ulysse Sakellaridis

ECOLE NATIONALE SUPERIEURE
DES MINES DE SAINT-ETIENNE

INSTITUT NATIONAL
POLYTECHNIQUE
DE GRENOBLE

N° d'ordre : 37 IS

THESE

présentée par

Ulysse SAKELLARIDIS

pour obtenir le diplôme de

DOCTEUR de 3 ème CYCLE

" SYSTEMES ET RESEAUX INFORMATIQUES "

CONCEPTION D'UN POSTE DE TRAVAIL APL ET DE SON UNITE DE GESTION MEMOIRE

Soutenue à Saint-Etienne le 7 Décembre 1983
devant la Commission d'Examen :

JURY

M. C. BELLISSANT — Président

MM. J.J. GIRARDOT
A. GUILLON
M. HABIB
F. MIREAUX
R. TISSERAND
— Examinateurs

ECOLE NATIONALE SUPERIEURE
DES MINES DE SAINT-ETIENNE

INSTITUT NATIONAL
POLYTECHNIQUE
DE GRENOBLE

N° d'ordre : 37 IS

THESE

présentée par

Ulysse SAKELLARIDIS

pour obtenir le diplôme de

DOCTEUR de 3 ème CYCLE

" SYSTEMES ET RESEAUX INFORMATIQUES "

CONCEPTION D'UN POSTE DE TRAVAIL APL ET DE SON UNITE DE GESTION MEMOIRE

Soutenue à Saint-Etienne le 7 Décembre 1983
devant la Commission d'Examen :

JURY

M. C. BELLISSANT — Président

MM. J.J. GIRARDOT
A. GUILLON
M. HABIB
F. MIREAUX
R. TISSERAND
— Examinateurs

Avant tout je tiens à remercier :

Monsieur Camille BELLISSANT, qui m'a fait l'honneur de présider le jury de cette thèse.

Monsieur Raymond TISSERAND, qui, malgré ses lourdes responsabilités, a accepté de juger mon travail.

Monsieur Alain GUILLON, qui a marqué son intérêt pour mon travail et a bien voulu participer à ce jury.

Monsieur Michel HABIB, qui a assuré un suivi vigilant de l'évolution de ce travail et ne m'a jamais ménagé ses conseils.

Messieurs Jean-Jacques GIRARDOT et François MIREAUX qui sont à l'origine de ce travail et ont été ses juges permanents. Sans leur longue expérience et leur infinie patience, un tel travail n'aurait pu aboutir.

Je remercie enfin Madame Georgette BONNEFOY et Messieurs BROSSARD, DARLES, LOUBET et VELAY qui ont mené à bien la réalisation matérielle du rapport, et tout le personnel présent et passé du département informatique de l'Ecole, dont la camaraderie a été une aide quotidienne.

Quant à mes parents, leur patience dans les moments difficiles et le réconfort qu'ils m'ont prodigué alors, sont au dessus de tout remerciement. Je leur dédie cet ouvrage.

INTRODUCTION

I) Avant-propos

L'extraordinaire évolution de la micro-informatique au cours de ces dernières années, et son utilisation de plus en plus fréquente dans les domaines les plus variés, sont à l'origine du développement des postes de travail autonomes scientifiques. Point de rencontre entre l'ordinateur se miniaturisant et le terminal devenant intelligent, le poste de travail a amené, avec l'introduction de la notion d'environnement de programmation, une véritable révolution dans la méthodologie de développement et d'utilisation de l'informatique.

A l'heure actuelle, on observe principalement deux courants en ce qui concerne le développement des environnements de programmation :

La première orientation, que nous qualifierons de "classique", consiste à adapter aux postes de travail autonomes des systèmes développés à l'origine pour des ordinateurs traditionnels, que ce soient des minis ou des gros ordinateurs, et reposant sur les mêmes principes fondamentaux. Le meilleur représentant de cette catégorie est le système UNIX. Le prix de revient, assez faible, d'un poste autonome et les progrès constants en logiciel de base qui ont permi la définition de noyaux de systèmes d'exploitation relativement peu volumineux, constituent les deux raisons essentielles de l'évolution des environnements "classiques" vers les postes de travail autonomes.

Dans la deuxième catégorie sont regroupés les environnements orientés vers l'intelligence artificielle. Les chercheurs de ce vaste domaine de l'informatique ont été amenés à passer assez rapidement à un autre "style" de programmation, à une autre manière de concevoir et de réaliser leurs applications, ceci essentiellement à cause d'une incompatibilité entre leur mode de pensée et le "mode de pensée" des systèmes classiques. Aussi, et pour des raisons de performances, ils ont commencé à réaliser en totalité, dès le début des années 1970, leurs systèmes spécifiques. De cette façon, un certain nombre de notions ont pris beaucoup d'importance, comme la programmation interactive, les programmes adaptatifs et l'accès agréable aux différentes ressources du système (systèmes Interlisp, Zetalisp).

Une analyse approfondie de ces deux courants permet de discerner en eux un certain nombre d'avantages mais aussi des inconvénients, comme par exemple la connaissance présumée de plusieurs langages différents et l'aspect très fermé des applications pour les premiers, la perte de la généralité du langage avec l'ajout d'extensions et l'absence notable de contrôles dans la manipulation des objets pour les autres.

Cette incompatibilité, mais aussi l'ensemble des inconvénients de ces deux systèmes, nous a amené, dans ce travail, à réfléchir sur un troisième type d'environnement. Le langage de base pour la conception de cet environnement est une version étendue d'APL. Ce langage est, à notre sens, le seul à répondre presque entièrement à nos besoins. En effet, la philosophie générale d'un système APL est fortement orientée vers la gestion efficace des contextes autonomes. En outre,

le langage s'intègre harmonieusement au sein d'un système qui lui est propre. Enfin, la littérature autour d'APL démontre que ce dernier est utilisé avec succès dans un grand nombre de domaines extrêmement variés. On remarque que la définition d'une version étendue (ou "adaptée") d'APL permet d'éliminer certains concepts que l'on peut qualifier d'archaïques, comme une gestion peu efficace des ressources de la machine, l'état assez primitif des outils de dialogue et des communications avec l'extérieur. On améliorerait ainsi grandement le degré de confort et la productivité de l'utilisateur.

Dans ce contexte d'environnement autonome de programmation à base d'APL, notre deuxième point de reflexion concerne la gestion d'un grand espace unique d'objets APL. Le dynamisme poussé à l'extrême, constitue la caractéristique fondamentale du langage APL, et rend ainsi assez délicate et couteuse non seulement la gestion des données, mais surtout l'accès individuel aux éléments de ces données. La conception d'un processus spécialisé qui se charge du contrôle et réalise les accès aux éléments des objets fait l'objet de la troisième et dernière partie de ce travail. Tout d'abord, ce processus est abordé en dehors de toute considération matérielle. Par la suite, on propose l'application de ces idées à la conception d'une unité de gestion mémoire spécialisée, et réalisable presque exclusivement en matériel. Nous nous sommes fixés un certain nombre de contraintes initiales de telle sorte que cette unité, non seulement remplisse pleinement son rôle dans le cadre spécifique d'un environnement APL, mais aussi puisse être utilisable au sein d'autres contextes, avec pratiquement la même efficacité.

II) Plan de la thèse

Le premier chapitre présente, à travers des exemples concrets de réalisation, les caractéristiques matérielles les plus intéressantes des postes de travail autonomes scientifiques.

Une étude critique des environnements de programmation existants ("classiques" et orientés vers l'intelligence artificielle) est présentée au début du deuxième chapitre, en mettant en évidence leurs avantages et inconvénients respectifs. Par la suite, dans ce même chapitre, nous présentons les aspects et les choix fondamentaux d'un troisième type d'environnement à base d'une version étendue du langage APL.

Dans le troisième chapitre, nous considérons un grand espace paginé et unique d'objets APL (le choix de la pagination est largement commenté) et nous présentons, en dehors de toutes considérations matérielles, un processus spécialisé dont le rôle est l'accès aux éléments des objets résidant dans cet espace, avec une gestion des droits d'accès, aussi bien au niveau de l'objet, qu'au niveau de la page accédée. Dans ce même chapitre, sont mis en valeur les aspects originaux de ce processus, à savoir les concepts de traitement local, de la traduction des adresses et de son mode de fonctionnement.

Le quatrième chapitre applique les idées exposées au chapitre précédent, en proposant la conception d'une unité de gestion mémoire orientée APL (UGM-APL) et

réalisable presque entièrement en matériel.

Enfin, une simulation (en APL...) de cette unité permet de valider sa conception.

TABLE DES MATIERES

"...

OBJET/STRUCTURE

1) La relation fondamentale Objet/Structure est à la base de notre activité perspective; elle exprime les rapports de définition réciproque entre notre perception des objets et celle des structures. On peut l'énoncer ainsi:
- Tout objet n'est perçu comme objet que dans un contexte, une structure qui l'englobe.
- Toute structure n'est perçue que comme structure d'objets composants.
- Tout objet de perception est en même temps, un OBJET en tant qu'il est perçu comme unité repérable dans un contexte, et une STRUCTURE en tant qu'il est lui-même composé de plusieurs objets.

2) Nous percevons les objets et les structures selon deux modèles d'attitude perceptive: l'identification et la qualification. La relation Objet/Structure s'énonce donc plus précisement en ces termes:
- Tout objet est IDENTIFIE comme objet dans un contexte, une structure qui l'englobe (mais à ce niveau, on ne le considère pas dans la totalité de ses caractères, on n'en retient qu'un trait, une valeur).
- Si l'on examine cet objet, on peut le QUALIFIER comme structure originale d'objets constituants. Ces objets constituants peuvent être à leur tour identifiés comme parties de cette structure et qualifiés par la structure dont ils sont eux-mêmes constitués, et ainsi de suite.

3) Cette relation définit donc une CHAINE objet-structure, qui descend vers l'infiniment petit quand on analyse l'objet comme structure d'objets constituants eux-mêmes analysables et ainsi de suite, et qui remonte vers l'infiniment grand, quand on situe l'objet dans la structure qui l'englobe, laquelle structure peut être à son tour considérée comme un objet dans un contexte, etc... Ce sont les "deux infinis" de la perception.

..."

Extrait de:
Guide des objets sonores,
par Michel Chion,
Ed. Buchet-Chastel, 1983.

CHAPITRE 1 : Les postes de travail autonomes scientifiques

L'évolution de l'informatique de ces dernières années et de ses utilisateurs, est à l'origine de la conception des postes de travail autonomes scientifiques. Leurs caractéristiques les plus intéressantes seront développées dans ce chapitre, en s'appuyant sur quelques examples concrets de réalisation.

1.1 L'evolution de l'informatique et les postes de travail

Avant d'aborder dans les détails une analyse à la fois générale et critique de l'évolution et des caractéristiques fondamentales des postes de travail autonomes scientifiques, il nous semble intéressant de rappeler quels ont été les grands moments de l'évolution de ces vingt dernières années de la science informatique. Ainsi, on pourra constater que la venue des postes de travail ne fait que respecter le sens naturel de cette évolution.

Jusqu'à la fin des années 1960, la quasi-totalité des activités informatiques étaient regroupées autour de gros systèmes, à la fois chers en fabrication et très lourds en utilisation. La taille physique de ces machines qui parfois nécessitaient plusieurs pièces pour y être installées, les sévères contraintes climatiques de fonctionnement, leur mode d'exploitation essentiellement en monoprogrammation et leur taux de panne relativement important, ont constitué des gros handicaps à l'évolution de ce type d'informatique.

Justement, à cette époque, le passage du semi-conducteur aux circuits intégrés et la maitrise des nouveaux modes d'exploitation comme la microprogrammation et le temps partagé, ont été à l'origine d'une nouvelle "révolution" dans le monde informatique : les mini-ordinateurs. Depuis, on a

constaté l'affirmation de la tendance de miniaturisation et du perfectionnement du matériel avec la fabrication de circuits de plus en plus denses et diversifiés. Et, c'est par hasard et sans beaucoup de conviction, qu'au début des années 1970, une petite société, spécialisée dans la conception et la fabrication d'automates programmables spécialisés (Intel pour ne pas la nommer...), a proposé un circuit programmable à usage général. Ainsi, le premier microprocesseur a vu le jour. A partir de ce moment-là, on a constaté une nette accéleration de l'évolution de l'informatique, aussi bien dans le matériel que dans le logiciel de base.

L'évolution du matériel est, à notre sens, la plus spéctaculaire. Elle est composée de deux courants, étroitement liés. Il y a d'une part le perfectionnement des circuits intégrés et d'autre part la conception de nouvelles architectures matérielles, à la fois plus souples et plus puissantes.

Dans le domaine des circuits intégrés, on remarque l'apparition de circuits spéciaux très performants. Les controleurs intelligents des périphériques, les processeurs flottants ou graphiques et les gestionnaires des réseaux locaux sont quelques exemples. Le degré de perfectionnement des microprocesseurs, exprimé notament par un jeu d'instructions très puissant, leur permet d'être comparables aux unités de traitement des mini-ordinateurs modernes actuels.

Notons enfin que l'ensemble des possibilités de

synchronisation et de communication des microprocesseurs a permis la réalisation d'architectures matérielles très élaborées, comme par exemple les systèmes multi-microprocesseurs incorporant à la fois des processeurs généraux et des processeurs spécialisés.

Le vaste domaine du logiciel a, lui aussi, connu une impressionnante évolution. A travers celle-ci, on aperçoit une première décomposition à deux niveaux du logiciel de base. L'évolution du premier niveau, celui qui se charge de gérer directement les ressources d'une machine, est étroitement liée à celle du matériel et aux nouveaux concepts architecturaux des systèmes. Quant au deuxième niveau de logiciel de base, il réalise les fonctions fondamentales du dialogue homme-machine. Il est important de noter ici que les qualités de ce niveau affectent directement le degré d'interactivité d'un système, tandis que les qualités du premier niveau agissent surtout sur les performances de celui-ci.

Voici donc, en quelques lignes, le sens de l'évolution de l'informatique dite "classique", vers le milieu des années 1970. Il nous parrait très intéressant de voir comment l'utilisateur de l'informatique, professionnel ou pas, a réagi face à cette évolution.

Il a d'abord observé un certain nombre de

constatations générales de cette évolution :

* La baisse spéctaculaire du coût du matériel ([THERESE], environ de 50% tous les 18 mois, depuis le début de la fabrication des circuits MSI), l'augmentation de sa fiabilité et la nette amélioration des conditions climatiques d'exploitation.

* L'étonnante progression du degré de miniaturisation (on remarque actuellement un doublement de densité des circuits intégrés tous les deux ans, mais cette progression était beaucoup plus spéctaculaire avant 198Ø) et la fabrication des circuits de plus en plus sophistiqués donnant ainsi lieu à la réalisation d'architectures très performantes.

* La mise à sa disposition des ressources matérielles de plus en plus importantes en nombre et en taille, tout en possèdant des qualités et des performances améliorées.

Ensuite, l'utilisateur a remarqué que les efforts des concepteurs de systèmes se sont orientés vers l'amélioration de la qualité du logiciel. Les fruits de ces efforts se sont concrétisés au milieu des années 197Ø avec, entre autres, les mini-ordinateurs virtuels et les utilitaires interactifs et performants. Mais ce sont là, tout simplement des logiciels qui avaient déjà fait leurs preuves au sein de grosses machines, et leur adaptation aux mini-ordinateurs n'a rien apporté d'extraordinaire, à part la disponibilité de ces logiciels sur des systèmes nettement moins onéreux.

C'est à cette époque où la clientelle utilisant la mini-informatique était pratiquement bien définie, que les premiers micro-ordinateurs ont fait leur apparition, en jouant ainsi les trouble-fête. A partir de ce moment-là, une véritable concurrence c'est créée sur le marché de la petite et moyenne informatique. Et cette concurrence s'est matérialisée par l'apparition de systèmes de plus en plus fiables et performants.

On peut dire que cette évolution parallèle est à l'origine des postes de travail autonomes scientifiques. En effet, les concepteurs de mini-ordinateurs, dans le souci de rendre leurs systèmes de plus en plus performants et les utilisateurs de plus en plus autonomes, ils ont eu l'idée de rapporter au niveau des consoles de visualisation un certain nombre de fonctions concernant l'affichage, la mise en pages des textes ou l'adressage de l'écran. Ce phénomène s'est amplifié et ainsi les consoles se sont transformées en véritables micro-ordinateurs.

De leur côté, les micro-ordinateurs ont évolué autour de cinq points. Premièrement, ils ont sû garder un prix de revient très abordable et incomparable avec celui d'un mini-ordinateur. Deuxièmement, la disponibilité de toutes leurs ressources est immédiate et très commode à utiliser. Troisièmement, ils ont développé un haut degré d'interactivité. Quatrièmement il y a eu des gros efforts d'amélioration de leur fiabilité. Enfin, leur matériel et leur logiciel

dispose, pour la plupart d'entre eux, des possibilités intéressantes de communication avec le monde extérieur.

Le développement des postes de travail autonomes scientifiques a été conditionné au départ par les originalités qu'a apporté cette double évolution des minis et des micro-ordinateurs. Plus qu'un ordinateur classique, celui-ci est un outil individuel et interactif dont les possibilités sont exploitées à fond. Dans cet esprit, différentes équipes de recherche ont été amenées à réaliser entièrement des systèmes autonomes, dediés à une application spécifique. Dans cette catégorie de systèmes, on peut mentionner quelques exemples : les machines orientées langages pour l'intelligence artificielle (système Interlisp de Xerox implémenté sur la machine Dorado, Zetalisp de M.I.T sur la NU-machine), celles orientées objets (au sens du système Smalltalk), ou encore la réalisation d'un réseau local de postes autonomes dans un contexte universitaire (système Spice de Carnegie-Mellon University).

1.2 La définition d'un poste de travail autonome scientifique

Il est dificile de donner de manière rigoureuse la définition d'un poste de travail autonome scientifique. Néanmoins, en analysant le comportement et les caractéristiques générales des réalisations existantes, on peut dresser une liste de propriétés de base qui permettront de classer un système dans la catégorie des postes de travail.

Avant de développer quelques exemples de réalisations existantes, on donnera une esquisse d'une définition très générale. Celle-ci nous aidera à focaliser l'analyse des exemples sur les points qui caractérisent un poste de travail, et auxquelles nous reviendrons par la suite.

Un poste de travail est un système informatique autonome et mono-utilisateur, avec les caractéristiques suivantes :

i) Il existe un grand confort d'utilisation, dû à une interactivité évoluée et à la présence et à la disponibilité immédiate de ressources importantes.

ii) Le logiciel de base du système est puissant, évolutif et orienté vers la conception et la

réalisation d'applications les plus variées.

iii) Le matériel de base qui sert à l'implémentation du dialogue homme-machine doit être très performant. Il est aussi caractérisé par la présence d'une unité arithmétique flottante puissante.

iv) Les possibilités matérielles et logicielles de communucations et d'interfaçage sont très développées, afin de rendre le système opérationnel dans un contexte de réseau local ou public.

1.2.1 Quelques exemples de réalisations existantes

a) SUN Microsystems

Le poste de travail SUN est réalisé autour d'un MC 68000. L'approche choisie ici est celle de l'utilisation du réseau local Ethernet.

Les autres caractéristiques matérielles de ce poste sont :

- jusqu'à 2 Méga-octets de mémoire centrale ;
- écran noir et blanc 1024*800 ;
- entrées-sorties gérées à travers un Multibus, ainsi que l'accès à Ethernet.

En ce qui concerne l'environnement logiciel de ce système, on remarque deux cas de figure : le poste seul et l'interconnexion de plusieurs configurations via Ethernet. Dans le deuxième cas (qui est de loin le plus intéressant), une version d'UNIX-Berkeley sera utilisée, permettant la gestion de plusieurs postes de travail cohabitant sur le réseau. Bien sûr, l'utilisateur beneficiera des tous les logiciels de base ou utilitaires qui peuvent tourner sous UNIX.

Enfin, il est intéressant de noter qu'un système graphique repondant à la norme SIGGRAPH/CORE est disponible.

b) Le système JERICHO

Quatre caractéristiques fondamentales ont guidé la réalisation de ce système :

1) Un environnement informatique possèdant une interactivité puissante.
2) Exécution directe des langages de haut niveau.
3) Avoir des interfaces matérielles homme-machine très performantes.
4) Interfaces de réseau évoluées.

Les idées de base se rapprochent à celles du système Dorado, avec en plus un souci sur son prix de revient.

Le processeur central est réalisé au moyen de circuits bipolaires en tranches. Le code d'ordre du processeur est entièrement microprogrammé et, mise à part l'unité arithmétique et logique, il contrôle l'accès à la mémoire, au(x) disque(s), au réseau local, et à un bus spécial à basse vitesse.

La mémoire du système est paginée à un niveau et adressable par un bus de 24 bits. La taille de la mémoire centrale peut varier de 512 kilo-octets à 16 Méga-octets.

La mémoire image de l'écran est adressable directement par le processeur, et elle est à double accès. La taille de cette mémoire est de 3 plans de 1Ø24*1Ø24 chacun.

La mémoire de masse "standard" du système est un disque de 195 Méga-octets. Jericho peut accepter un maximum de 255 disques. En ce qui concerne le réseau, celui-ci s'appelle Fibernet et possède un débit de 2 Méga-octets/seconde.

Le logiciel de base de Jericho est construit sur la définition de machines virtuelles. Ces machines, dont la définition comprend l'ensemble des instructions, des structures de données, des structures de contrôle et de gestion des ressources, sont implémentées en microcode dans le système. Il existe actuellement des machines virtuelles Pascal et Interlisp sur lesquelles toute la programmation se fait uniquement en ces langages.

c) Le système PERQ (Three Rivers)

Le coeur de ce système est constitué d'un processeur microprogrammé dont le code d'ordre est le P-code. Ce processeur assure à la fois les fonctions de calcul et les fonctions graphiques (les opérations graphiques de base sont aussi microprogrammées). La mémoire centrale de ce système peut atteindre 1 Méga-octet, et dans la configuration de base on trouve un disque de 24 Méga-octets. Un effort a été porté sur l'interaction homme-machine, avec notament un écran bit-map de 768*1Ø24 points, tablette sensitive ("touch tablet") en entrée, sortie vocale, fenêtrage de l'écran et un ensemble de facilités de manipulation de documents, au moins équivalent sinon supérieur aux systèmes de traitement de textes courants. Enfin, les postes PERQ peuvent être interconnectés par l'intérmédiaire du réseau local Ethernet.

Le système PERQ joue un rôle important dans le développement de la recherche en informatique en Grande Bretagne. Un constructeur national britannique a negocié un contrat de coopération avec Three Rivers, la société qui a conçu le système. A partir de cette base industrielle, a été lancée, sur le plan national britannique, une action de développement de logiciels autour des systèmes distribués utilisant le poste PERQ. Partant d'une base logicielle commune comprenant UNIX, FORTRAN 77, Pascal, GKS, l'ambition est de fédérer les efforts de la recherche en informatique britannique. A l'heure actuelle, sont en cours de développement Ada, Prolog, des applications CAO, bureautique, etc...

d) La NU-machine (M.I.T.)

Construit autour du NU-bus, mécanisme général de communication ou viennent se "greffer" un certain nombre de modules matériels, ce système présente quelques originalités.

La NU-machine se présente comme un ensemble de modules matériels spécialisés et attachés au NU-bus. Les spécifications de ce bus permettent la cohabitation de modules très diverses, sans nécessiter d'investissements matériels ou logiciels supplémentaires. Parmi les modules "standard", on trouve des modules processeur (ils existent à base de MC 68000 et de Z 8001), des modules de mémoire centrale (par 128 kilo-octets) et de DMA avec la mémoire secondaire et ceux qui augmentent l'interactivité (souris, boule, écran couleur haute définition,...).

Dans le module processeur on trouve le dispositif de gestion de la mémoire. Il s'agit d'une gestion paginée où, des 24 bits d'adresses du microprocesseur, on passe aux 30 bits du NU-bus.

Le principal module d'interface interactive contient le bit-map de l'écran vidéo (128 kilo-octets par plan) et supporte aussi les fonctions programmables de translation de coordonnées, de l'echelle de gris ou de couleurs, etc...

Une des plus grandes originalités du système NU est le "rho processor". Il s'agit d'un module banalisé qui

sert à l'implémentation des prototypes des modules spécialisés, comme par exemple un accélerateur graphique ou un processeur dedié à un langage.

Le coeur du "rho processor" est constitué d'un processeur 32 bits microprogrammable. Le repertoire de ses instructions est orienté plutôt vers la manipulation des données que vers le calcul numérique. Ce module utilise 4 kilo-mots pour le stockage des micro-instructions, accessibles par le NU-bus.

Pour ce qui concerne l'environnement logiciel de ce système, il comporte pour le moment un système UNIX monoprocesseur, C, Pascal, FORTRAN et un logiciel graphique répondant à la norme SIGGRAPH/CORE. Une version d'UNIX supportant la mémoire virtuelle est prévue pour cette année.

Enfin, il est intéressant de noter la possibilité de connecter le système NU à un réseau local, de débit 8 Méga-octets/seconde et avoir ainsi l'accès aux périphériques les plus divers.

e) Le système SPICE (Carnegie-Mellon University)

Le département Informatique de l'Université de Carnegie-Mellon (CMU), a mis en place un environnement scientifique informatique (SPICE) basé sur un ensemble de postes de travail interconnectés via un réseau local de débit de 10 Méga-octets/seconde, dans lequel on trouve aussi toutes les ressources partageables. Les caractéristiques matérielles d'un poste de travail dans l'environnement SPICE, sont les suivantes :

- Un processeur microprogrammé avec une vitesse d'exécution de 1 M micro-instructions/seconde. Disponibilité d'espace pour le stockage de 16k micro-instructions, accessibles à l'utilisateur.

- Adressage virtuel sur 32 bits, 1 Méga-octet minimum de mémoire centrale et 100 Méga-octets de mémoire secondaire paginée.

- Interface homme-machine : écran couleur haute définition (1024*1024), clavier, souris, entrée et sortie vocales, entrée d'images statiques.

D'après les concepteurs de SPICE, une grande importance est donnée à l'environnement logiciel. Parmi les facilités, on peut trouver un éditeur dirigé par la sémantique, un système de traitement de texte, une

réalisation importante de messagerie électronique entre les utilisateurs comprenant un système de mise à jour de rendez-vous, mais aussi des facilités de développement de gros programmes en plusieurs langages de haut niveau (création, correction, analyse, gestion des programmes, maintenance des bibliothèques). On peut encore citer l'existence de logiciel graphique d'aide à la conception de systèmes matériels, et des outils généraux comme la simulation de calculateurs de bureau ou les systèmes personnels de bases de données.

Notons enfin que la plupart des outils qui servent à définir l'environnement de programmation de SPICE sont très conditionnés par le choix de Ada comme étant le langage principal de programmation.

f) APOLLO Computer

L'approche choisie par ce constructeur fait jouer le rôle central au réseau local DOMAIN. La structure du poste de travail APOLLO, en particulier au niveau de la gestion de la mémoire, favorise la communication entre postes de travail, donne au réseau l'accès direct à la mémoire centrale et permet ainsi à un utilisateur d'exécuter plusieurs tâches sur plusieurs postes de travail différents et d'en contrôler cette dernière sur plusieurs fenêtres de son écran.

Plus précisement, un utilisateur connecté sur un noeud du réseau DOMAIN, peut "voir" la totalité de la mémoire disponible sur le réseau, sous forme d'un très grand espace d'objets paginés. La désignation d'un objet est universelle (la même pour tous les utilisateurs) et l'adressage à l'intérieur d'un objet se fait en indiquant le numéro de page et le deplacement dans celle-ci. L'accès à un objet distant (résidant dans la mémoire d'un autre noeud) se fait en deux phases : dans un premier temps une liaison est établie, à l'aide de la désignation universelle de l'objet ; ensuite, et suivant les besoins, les pages de l'objet à accéder sont transportées à travers le réseau dans l'environnement local du processus demandant. Chaque noeud du réseau DOMAIN peut être considéré comme un système informatique autonome, dont voici les principales caractéristiques matérielles : microprocesseur MC 68000, 16 Méga-octets de mémoire

virtuelle, jusqu'à 1.5 Méga-octets de mémoire centrale, écran bit-map 1Ø24*8ØØ, tablette sensitive de pointage de l'écran, interfaces asynchrones d'entrées/sorties série, accès au réseau DOMAIN (12 Méga-bits/seconde).

Du côté logiciel, on trouve le système d'exploitation AEGIS. Celui-ci est orienté objets (les données et les programmes reconnus par le système sont considérés comme des entités "génériques" ayant des attributs bien précis), gère la mémoire virtuelle et peut supporter jusqu'à 15 processus concurrents de 16 Méga-octets chacun, à chaque noeud du réseau. Parmi les principaux utilitaires de base, on peut remarquer la gestion interactive d'affichage qui comprend un éditeur orienté écran, la bibliothèque des primitives graphiques de base, la gestion des processus (communication, création, synchronisation sur évènement), un puissant interprète de commandes (style UNIX) et une gestion de directoires pouvant couvrir tout le réseau. Enfin, notons parmi les options logicielles disponibles : FORTRAN 77 ANSI, Pascal ISO, compilateur C et AUX, un environnement logiciel basé sur UNIX System III.

g) Le système DORADO (XEROX)

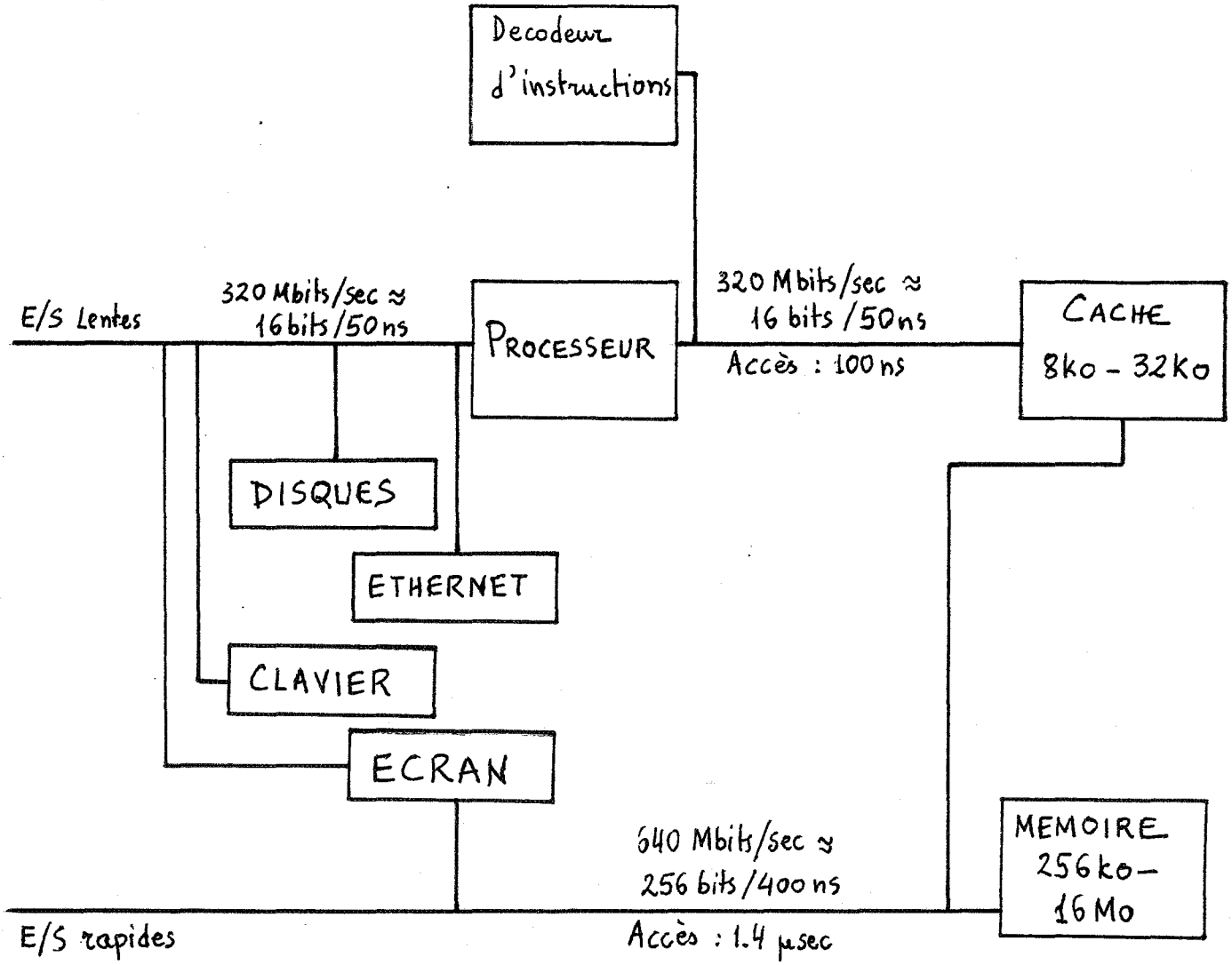
XEROX a été la première société à se lancer dans la conception de postes de travail autonomes scientifiques. Le premier système expérimental, nommé ALTO, a vu le jour en 1973. C'est en 1979 que le premier prototype DORADO, descendant direct de ALTO, a été réalisé.

DORADO peut être considéré comme un poste de travail à très hautes performances. Il s'agit d'une architecture matérielle microprogrammée à tous les niveaux (processeur central, controleurs périphériques, interaction homme-machine). La gestion de la mémoire est hierarchisée (utilise un cache), et elle est accessible uniquement par le processeur. Un haut degré de parallélisation a été recherché au niveau de l'exécution des micro-instructions, elles-mêmes assez élaborées (elles ont l'apparence de macros).

Le système DORADO supporte un utilisateur au sein d'un environnement de programmation qui couvre des domaines extrêmement variés, allant depuis le niveau des micro-instructions, jusqu'à celui des langages de haut niveau. Destiné à être intégré dans un environnement de travail et de recherche en informatique déjà existant, le système DORADO est resté compatible avec son ancêtre ALTO. Néanmoins, DORADO est optimisé pour l'exécution directe des langages. Un programme compilé est représenté par une suite de codes

indiquant l'exécution de micro-instructions ou des macros. De tels compilateurs spéciaux existent pour les langages Mesa, Interlisp et Smalltalk.

Le schéma suivant decrit l'architecture générale du système DORADO, et situe assez bien le degré de ses performances :



h) Le poste de travail SM 90

Initialement conçu pour être exploité comme commutateur intelligent des centrales téléphoniques, le système SM 90 a, depuis peu de temps, été adopté dans les milieux scientifiques multi-utilisateur car il possède quelques propriétés intéressantes :

Autour d'un bus général de communication, peut se développer une architecture évolutive à base de trois catégories de modules différents : les modules de traitement (microprocesseurs 16 et/ou 32 bits), les modules mémoire et les modules d'échange. En plus du bus général, un deuxième bus (local), deux fois plus rapide, peut être utilisé pour réaliser un ensemble modulaire de cartes autour de chaque module de traitement. Au sein de chaque module d'échange se trouve un microprocesseur 8 ou 16 bits qui gère les entrées-sorties. Enfin, le protocole de communication du bus général a été conçu de manière à être indépendant de la nature des processeurs utilisés et la gestion de la mémoire centrale est effectuée à travers une unité matérielle spécialisée, elle aussi indépendante de la nature des processeurs de traitement.

Un grand effort national est entrepris, afin de développer un environnement logiciel important et de bonne qualité sur SM 90. Actuellement, ce logiciel comprend un système d'exploitation compatible UNIX V7 ("smx"), les compilateurs C, Pascal et FORTRAN, ainsi qu'un système Lisp (Le-Lisp). En plus, une version du

système SOL existe actuellement sur SM 90, et sa commercialisation est prévue pour le dernier trimestre 1983.

Notons enfin l'existence d'un certain nombre de réalisations en cours et qui devraient s'achever à court ou à moyen terme. Parmi ses réalisations on peut citer le compilateur FORTRAN 77, le système Prolog, la norme graphique GKS, APL, Smalltalk, UNIX multiprocesseur, carte virgule flottante, connexion au réseau Ethernet, Modula-2, ou encore un système de base de données relationnelles.

= Remarque : Le tableau suivant regroupe les caractéristiques essentielles des postes de travail que l'on vient d'analyser. Dans le bas de ce tableau on trouve les caractéristiques des deux micro-ordinateurs personnels 16 bits les plus répandus, afin de permettre une comparaison des performances.

1.2.2 Les caractéristiques des postes de travail

Dans ces paragraphes, on essayera de regrouper toutes les caractéristiques communes des postes de travail qui se dégagent à travers les exemples. Une attention particulière sera pretée aux caractéristiques qui ne sont pas suffisament développées à notre avis mais qui devraient marquer le sens de l'évolution des postes de travail. On se limitera ici surtout aux caractéristiques matérielles, une étude detaillée des environnements logiciels sera abordée au chapitre suivant.

On peut regrouper en trois grandes catégories les caractéristiques matérielles des postes de travail autonomes scientifiques :

i) L'architecture matérielle de base, qui inclue les unités de traitement et de stockage.

ii) La communication homme-machine, qui affecte surtout l'interactivité d'un système.

iii) Enfin, la communication machine-machine qui regroupe les interfaces d'interconnexion avec d'autres systèmes, mais qui représente aussi le degré d'évolution d'une configuration.

La complexité des architectures de base des postes de travail s'avère très variable, allant depuis la machine mono-processeur classique, jusqu'aux systèmes multi-processeurs sophistiqués, en passant par les architectures microprogrammées très spécialisées.

Le principal avantage des architectures mono-processeur classiques est leur prix de revient très faible, dû à la simplicité de leur conception et à leur fiabilité. Dans ces systèmes (voir [SUN] et [APOLLO] parmi les exemples précédents), l'unité centrale qui est constituée dans la majorité des cas d'un microprocesseur 16 ou 32 bits, se charge de toutes les manipulations et contrôles de l'ensemble. Néanmoins, dans certaines réalisations de cette catégorie, on trouve des processeurs esclaves spécialisés qui gèrent soit l'interactivité (processeurs graphiques), soit la mémoire (circuits MMU).

Nettement plus sophistiquées et plus complèxes à mettre en oeuvre, les architectures microprogrammées constituent la voie principale des systèmes très spécialisés. Elles sont à l'origine des machines individuelles orientées vers l'intelligence artificielle ou la gestion des bases de données, et où les concepteurs ont cherché une grande efficacité dans ces domaines bien précis. En regardant quelques exemples de réalisations (comme [JERICHO], [PERQ], [NU], [SPICE], ou [DORADO]), on remarque que, à part l'efficacité au niveau de l'exécution des applications spécifiques dûe à un jeu d'instructions particulièrement adapté, un grand effort a été fait

pour améliorer l'interactivité de ce type de systèmes, avec la mise en place d'un matériel de visualisation très sophistiqué et d'un logiciel adéquat qui ne l'est pas moins. En plus, la gestion d'une mémoire hierarchisée et adaptée aux structures des données et des programmes exploités, se fait aussi par microprogramme. Toutes ces caractéristiques rendent l'utilisation de tels systèmes très agréable.

Au sein de la catégorie qui regroupe les architectures multi-processeurs (telles que la [NU] ou la [SM 90]), on peut observer deux courants principaux:

i) Les multi-processeurs "maître-esclaves" : Ces architectures sont constituées d'un processeur central qui contrôle le fonctionnement de la totalité du système et qui communique aux processeurs spécialisés (flottants, graphiques, gestionnaires de mémoire ou de bases de données) des travaux spécifiques et répétitifs.

ii) Les multi-processeurs "équivalents" : Autour d'une structure fondamentale qui est le bus de communication, viennent se greffer des processeurs généraux, la mémoire centrale et les interfaces. Dans un tel système, un processus logiciel, avant d'être exécuté, est décomposé en tâches plus élémentaires pouvant s'exécuter en parallèle. Des techniques connues comme les boîtes aux lettres, les sémaphores ou les interfaces logicielles, permettent à ces tâches de se synchroniser, si c'est nécessaire.

Cette dernière catégorie comporte des caractéristiques assez originales, et plusieurs raisons laissent à penser qu'elle constitue la voie principale d'évolution des postes de travail. En effet, de part leur principes de base, se sont des architectures évolutives et très facilement adaptables aux progrès technologiques. Ici on peut citer comme exemples le "rho processor" (module de traitement "générique" microprogrammable) de la NU-machine ou encore le cas de la SM 90. Précisement, la structure générale de cette dernière est faite de telle façon que l'on pourra, avec un minimum de modifications, la réconfigurer avec des unités de traitement construites autour de nouveaux processeurs plus perfectionnés. D'un autre côté, les logiciels qui possèdent un certain degré de parallélisme, comme les applications de l'intelligence artificielle, les langages traitant explicitement la concurrence (Ada, Modula-2), et les applications dirigées par la syntaxe ou le flot de données, peuvent être réalisées plus efficacement sur ce type d'architecture.

Le développement de l'interactivité des postes de travail, a permi la réalisation de systèmes très agréables à utiliser, tout en augmentant la productivité de l'utilisateur. Ce fait constitue une caractéristique importante et très appréciée.

A la place d'une définition, on peut dire que l'interactivité reflète la puissance et la fonctionnalité du dialogue homme-machine dans toutes les phases d'utilisation d'un système informatique (conception, réalisation, mise au point et exploitation

d'une application). Elle est constituée d'un matériel et d'un logiciel qui coopèrent étroitement, lui donnant ainsi son apparence finale.

Le matériel est étroitement lié à l'acquisition et la visualisation des données. Autour d'un écran graphique haute définition (au moins 1024*1024 points) monochrome ou couleur, on trouve actuellement au sein des postes de travail les lpus perfectionnés ([APOLLO], [NU], [SPICE] ou [DORADO]) des outils de pointage comme la souris, la boule et le stylet, ou encore des dispositifs de reconnaissance et de reproduction vocale.

Sous l'appellation d'environnement de programmation est regroupée actuellement la totalité des logiciels interactifs de base d'un système. Initialement défini par les concepteurs de postes de travail orientés langages ([DORADO], [NU]) et en s'appuyant sur un matériel interactif perfectionné, il regroupe tous les utilitaires de base relatifs au suivi intense et efficace d'une application pendant toute la durée de sa vie. Quelques exemples de tels utilitaires sont les éditeurs orientés écran multi-fenêtres, les assistants de programmation et les deverminateurs interactifs.

Enfin, abordons ici la dernière catégorie des caractéristiques matérielles des postes de travail, c'est à dire la communication machine-machine.

L'interconnexion des postes de travail au moyen d'un réseau local, procure un environnement multi-utilisateur. Ceci donne plusieurs avantages :

* Coût moindre : Des ressources matérielles et logicielles chères, comme par exemple les disques magnétiques ou optiques à grande capacité, les bandes et les imprimantes rapides, sont connectées au réseau et donc accessibles à tous les utilisateurs.

* Possibilités d'extension : Dans un réseau local on peut facilement ajouter soit des nouveaux services généraux, soit d'autres postes de travail, en augmentant ainsi les possibilités de l'environnement.

* Ce type d'environnement décentralisé permet à ses utilisateurs de bénéficier, aussi bien des avantages des configurations de type mini-ordinateurs qui sont la présence de ressources importantes et des logiciels fiables et performants, que ceux des micro-ordinateurs, à savoir le fonctionnement interactif autonome et l'accès, localement, à la totalité des ressources du poste de travail.

Ces caractéristiques laissent préjuger un avenir prommetteur des postes de travail. En effet, ils réunissent toutes les qualités que le scientifique (informaticien ou autre) attend de l'outil informatique. Et, fatalement, la place qu'ils occupent sur le marché devient de plus en plus importante au detriment des autres formes d'informatique plus classiques. Les mini-ordinateurs et les

micro-ordinateurs individuels de haut de gamme, dont la philosophie générale et certaines caractéristiques sont assez similaires avec celles de postes de travail, seront les plus dûrement affectés par cette évolution. Ceci tient essentiellement au fait que les postes de travail regroupent à la fois les avantages de ces deux catégories, sans hériter de leurs inconvéniants.

Quant aux gros ordinateurs, ceux-ci ne semblent pas affectés par cette évolution, car leurs domaines d'utilisation sont sensiblement différents. Néanmoins, ils peuvent parfaitement s'intégrer au sein d'un réseau local de postes de travail en jouant le rôle de serveur scientifique ou d'un exécutant puissant de grosses applications (météorologie, statistiques).

1.3 BIBLIOGRAPHIE DU CHAPITRE I

[SM 90-1] : Le poste de travail SM 90
Rapport INRIA, 1982

[SM 90-2] : Architecture multi-microprocesseurs et disponibilité : la SM 90
L'echo des RECHERCHES no 105, juillet 1981

[SPICE] : CMU Proposal for Personal Scientific Computing
A. Newell et autres, IEEE 1980

[DORADO] : A processor for a high-performance personal compuser
B. W. Lampson, K. A. Pier, IEEE 1980

[NU] : An approach to personal computing
S. A. Ward, C. J. Terman, IEEE 1980

[PERQ] : A commercially available personal scientific computer
B. Rosen, IEEE 1980

[JERICHO] : Jericho : A professional's personal computer system
N. R. Greenfeld, IEEE 1981

[APOLLO] : Apollo Domain architecture, preliminary report
Apollo Computer Inc., février 1981

[THERESE] : Le projet THERESE
Techniques et science informatique (TSI)

CHAPITRE 2 Les environnements existants; vers un environnement APL

Une étude critique des environnements de programmation existants est présentée au début de ce deuxième chapitre. Leur incompatibilité et surtout leurs inconvénients respectifs nous amènent à la définition d'un environnement de programmation à base d'une version étendue du langage APL.

2.1 Introduction

Nous avons abordé dans la chapitre précédent la description d'un certain nombre de postes de travail existant. Nous allons voir que ceux-ci, points de rencontre entre l'ordinateur se miniaturisant et le terminal devenant intelligent, ont amené une véritable révolution dans la méthodologie de développement et d'utilisation de l'informatique.

En effet, alors que l'évolution du matériel était prévisible, celle du logiciel apporte plus de surprises. Alors que l'on s'attendait à voir se multiplier les langages "algoliques" ou "pascaleux", on a vu apparaître des systèmes apportant des idées nouvelles et tout à fait différentes, tels CLU, Smalltalk, ou encore de vieux langages tels LISP évoluer grâce aux nouvelles possibilités qu'apportent ces environnements. Enfin, dernier point, on assiste à une transposition sur ces postes de travail de systèmes -tels UNIX [Bel 78]- développés à l'origine pour des ordinateurs traditionnels.

Ce sont ces différents aspects du logiciel que nous allons maintenant analyser, en nous intéressant d'abord aux environnements de travail classiques, ensuite aux environnements de type "intelligence artificielle". Une seconde partie développera leurs caractéristiques le plus intéressantes. Enfin, nous exposerons les idées générales permettant d'envisager la conception d'un poste de travail basé sur le langage APL.

2.2 Environnements de travail classiques

La définition d'un environnement "classique" repose sur les principes fondamentaux des systèmes d'exploitation que l'on trouve actuellement dans la plupart des minis et des gros ordinateurs (MULTICS, UNIX). Les raisons essentielles de l'évolution des environnements classiques vers les postes de travail autonomes sont les suivantes:

- Tout d'abord, le prix de revient d'un poste autonome est assez faible par rapport à celui d'un mini-ordinateur.

- Ensuite les progrès de ces dernières années en logiciel de base ont permis la définition de noyaux de systèmes d'exploitation relativement peu volumineux et néanmoins assez complets et performants.

Ces environnements sont conçus pour le développement d'applications scientifiques ou de gestion, écrites dans des langages compilés classiques (FORTRAN, COBOL, Pascal, PL/1, C...). On retrouve, à une échelle plus restreinte, les possibilités existant jusqu'à présent sur des sytèmes plus importants. Ce type d'environnement est utilisé par des informaticiens pour le développement de ces applications et par des scientifiques pour l'exploitation des mêmes applications. Il est très facile de mettre en place une structure d'accueil qui guide simplement et agréablement l'utilisateur non informaticien pendant

l'exploitation de son application.

De manière générale, les avantages des environnements classiques sont les mêmes que ceux des systèmes réalisés sur les minis et les gros ordinateurs. De plus le fait de proposer un tel système sur un poste de travail autonome offre une plus grande indépendance et surtout un contrôle et un accès plus facile et plus agréable aux ressources matérielles du poste. Du fait de l'existence d'une version du système par poste, ce dernier peut être personnalisé à loisir, selon le profil et les goûts de l'utilisateur, selon le type d'application et ceci sans altérer le fonctionnement des autres postes.

Basées sur ces environnements de type classique, on trouve également des réalisations de systèmes spécialisés comme des postes graphiques, des bases de données ou des machines de développement logiciel pour microprocesseurs.

Indépendamment des performances qui sont fonction de l'architecture plus ou moins sophistiquée du poste de travail autonome, mais qui sont bien évidemment plus restreintes que celles des grosses machines, ce type d'environnement est affecté d'un certain nombre d'inconvénients. Prenons comme exemple le système UNIX (et ses "compatibles" XENIX, AUX, ONYX,...) qui est actuellement le plus répandu et aussi le plus évolué de sa catégorie.

L'utilisateur doit manipuler plusieurs types d'objets, pas toujours clairement définis, à travers plusieurs utilitaires très différents, chacun possèdant sa propre syntaxe et sa propre sémantique. Le

développement d'un programme nécessite par exemple la connaisance plus ou moins profonde de quatre ou cinq langages: langage de commande, éditeur de texte, un langage de programmation évolué, l'utilitaire de mise au point, l'assembleur... Lś essais de standardisation (ou plus exactement, de choix de conventions communes entre les différents utilitaires, par exemple la définition de modèle -pattern-matching-) restent malgré tout assez limités.

Les applications conçues et réalisées sous UNIX sont assez fermées. Cette difficulté de faire coopérer des programmes entre eux provient essentiellement du manque d'outils adéquats. Le mécanisme de tubes ("pipes"), malgré la souplesse réelle qu'il apporte dans un certain nombre de cas, s'avère vite limité dans ses possibilités [Bel 78].

Comme pour la manipulations des objets, le traitement des erreurs est souvent peu puissant, et différent selon le niveau du système où elles se produisent: dans un programme de commande, dans un processus, etc.

Le fait que les programmes soient compilés nécessite l'utilisation d'utilitaires de mise au point (comme "adb") souvent hermétiques pour le profane.

La gestion de la mémoire secondaire, essentiellement sous forme de fichiers et de répertoires, demeure peu évoluée, malgré l'intérêt certain de la notion de hiérarchie de fichiers.

2.3 Environnements de type "Intelligence Artificielle"

Le domaine de l'intelligence artificielle est devenu ces dernières années très vaste et très varié. Citons ici quelques unes des branches fondamentales: reconnaissance des formes, analyse des langues naturelles, logique des prédicats du premier ordre, preuve automatique de théorèmes et de programmes, reconnaissance et synthèse vocale, conception de systèmes experts.

De par ses principes, l'environnement de programmation de l'intelligence artificielle est sensiblement différent de celui de l'informatique classique. En effet, au début des années 60, à l'époque où l'intelligence artificielle a commencé à se développer, n'existaient que des environnements et des langages assez peu adaptés à ses besoins.

Les chercheurs en intelligence artificielle ont ainsi été amenés à passer rapidement à un autre "style" d'informatique, à une autre manière de concevoir et de réaliser les applications. Au début, ils se sont contentés de bâtir leur propre logiciel de base (langage et système spécialisé LISP) sur des systèmes existants, mais, très vite, et surtout pour des raisons de performance, ils ont été amenés à concevoir en totalité leurs systèmes spécifiques. Dans cette nouvelle forme d'exploitation d'une machine, un certain nombre de notions ont pris beaucoup d'importance: programmation interactive, programmes adaptatifs et conversationnels, accès très agréable aux différentes ressources du système.

Petit à petit, ces nouveaux systèmes ont évolué, jusqu'à arriver, vers 1975, à la définition de la notion d'environnement de programmation: ensemble d'outils matériels et (surtout) logiciels, destinés à influencer, et, dans la majorité des cas, déterminer la classe des problèmes qu'un utilisateur peut étudier, mais aussi montrer jusqu'à quel niveau de complexité il peut progresser, et à quelle rapidité. En simplifiant, on peut dire que plus l'environnement est adapté et coopératif, plus l'utilisateur devient ambitieux et productif.

Pour illustrer ces principes, voici quels étaient les buts majeurs que s'étaient fixés les concepteurs d'INTERLISP [Tei 78], système orienté vers l'intelligence artificielle, entièrement conçu à partir du langage LISP:

Favoriser le développement d'applications orientées vers l'intelligence artificielle, qui ont la particularité de n'être pratiquement jamais complètement spécifiées à l'avance. L'utilisateur fait évoluer ses programmes en fonction de la compréhension du problème qu'il acquiert grâce aux résultats intermédiaires obtenus.

"Laisser faire" la machine: essayer d'utiliser au maximum les ressources informatiques qui coûtent de moins en moins cher, plutôt que les ressources humaines, de plus en plus coûteuses.

Ne pas non plus chercher à tout prix la simplicité des interfaces homme-machine. Cependant, ont pris naissance peu à peu des outils puissants tels Masterscope (outil d'analyse détaillée des

programmes constituant une application), "Do What I Mean" (mécanisme semi-automatique de correction d'erreurs, basé sur le système de traitement d'erreurs), ou le Programmer Assistant, sorte de manuel de référence intelligent comprenant à demi-mot.

Ainsi, l'environnement de programmation peut être considéré comme l'outil fondamental des systèmes orientés vers l'intelligence artificielle. Notre étude portera plus spécifiquement sur les systèmes construits autour du langage LISP, et en particulier INTERLISP et ZETALISP, qui sont les plus évolués parmi ceux-ci.

En premier lieu, nous décrirons les avantages du langage LISP. C'est un langage qui offre la possibilité de manipuler des programmes par d'autres programmes, offrant ainsi de grandes facilités pour la mise au point d'applications auto-évolutives. La syntaxe de LISP est extrêmement simple, et les programmes sont représentés de façon interne sous forme de listes, structures de données très facilement manipulables.

Autour du noyau de base viennent se greffer les différentes primitives système et extensions du langage. Celles qui nous intéressent ici concernent la notion de type, et l'introduction de nouvelles structures de données, comme les vecteurs ou les tableaux, accompagnées de primitives spéciales de manipulation et de conversion. Quant aux très nombreuses primitives système (plus de 800 dans INTERLISP, plus de 1400 dans ZETALISP, agrémentées de 400 variables système!), elles se présentent sous forme de fonctions LISP, et sont donc directement accessibles depuis n'importe quel programme.

Remarquons enfin que le fait de construire un système autour d'un langage interprété offre à tous les niveaux une interactivité accrue qui se traduit par une grande facilité de mise au point et un passage quasi-instantané du mode édition au mode exécution.

Enfin, les postes de travail LISP ont apporté l'idée, (et même fait la preuve), fondamentale à notre sens, qu'un même langage pouvait servir, non seulement comme langage de programmation et langage de commande, mais encore comme interface unique avec tous les utilitaires. Un exemple fameux est l'éditeur de texte EMACS, conçu au M.I.T., qui tire toute sa puissance du fait qu'il accepte des macro-commandes écrites en LISP.

Cependant, ces développements -souvent un peu anarchiques- autour du langage, ne vont pas sans un certain nombre d'inconvénients.

- Les extensions du langage de base pour obtenir un langage plus puissant ont provoqué une perte de la généralité du langage: au départ, LISP était conçu autour d'un petit nombre de primitives qui manipulaient agréablement une structure de données unique, la liste. Au fur et à mesure que la communauté des utilisateurs de LISP s'élargissait, le besoin de définir de nouvelles structures, plus complexes et mieux adaptées à certaines classes spécifiques de problèmes, s'est fait sentir. Ainsi sont apparus les tableaux, les chaînes de caractères, les tables hashcodées. Pour ces nouvelles structures, de nouvelles fonctions de manipulation ont été définies, fonctions naturellement spécifiques à chaque structure. De plus, la conversion d'un objet d'un type dans un

autre nécessite des fonctions de conversion, elles aussi spécifiques. Schématiquement, les objets manipulés par le LISP originel, trop élémentaires, ont nécessité la création de nouveaux types primitifs, d'où un grand nombre de nouvelles fonctions, et une grande complexité dans l'utilisation des objets. Pour conclure sur ce point, nous dirons qu'il est difficile d'ignorer la structure que l'on manipule. Ainsi, la primitive de comparaison se notera EQ pour des atomes, EQN pour des nombres, EQSTRING pour des chaînes de caractères, etc.

- La syntaxe du langage a elle-même subi une évolution similaire. Extrêmement simplifiée, elle s'avère rapidement fastidieuse à manipuler. Ceci explique les très nombreux ajouts que l'on trouve dans le langage: l'apostrophe, les crochets, la back-quote, les caractères d'échappement, et enfin les macro-caractères, toutes ces additions, souvent incompatibles d'un système à un autre, rendant le langage bien hermétique pour le profane.

- Dans les systèmes LISP -même les plus évolués- il n'existe en général pas de notion de catalogue ou de hiérarchie des objets: l'utilisateur a une vision "à plat" de l'espace des noms. Ceci peut entraîner, lors de l'utilisation d'applications écrites par d'autres, des conflits dans les noms des objets, conflits contournés par l'utilisation de noms "à rallonge" pour les utilitaires systèmes, tels "make-array-into-named-structure" ou "si:rename-with-new-definition-maybe" [Wei 83].

- Bien que le langage soit interprété, il n'existe pas,

ou peu, de contrôles dans la manipulation des objets par les primitives système. De nombreuses primitives système sont réalisées par des algorithmes très courts, en quelques instructions machines. Ainsi, EQ se contente de comparer les adresses mémoire de ses arguments pour en déduire s'ils sont égaux ou non. D'autres fonctions, telles RPLAC, peuvent modifier sélectivement un objet, avec le risque de créer une structure que le système ne sait plus gérer correctement. Ceci correspond en fait à une volonté délibérée des concepteurs de systèmes LISP qui préfèrent s'adresser à l'utilisateur averti, en lui offrant un minimum de contraintes. En contrepartie, cette approche offre l'avantage d'être moins coûteuse en temps d'exécution, puisque les primitives ne font qu'un minimum de vérifications. Là encore, en schématisant un peu, on pourrait dire que, compte tenu du bas niveau de la majorité des primitives du langage, on ne peut se permettre d'ajouter beaucoup de protections, qui pénaliseraient trop l'exécution.

- Enfin, nous ferons le reproche aux systèmes orientés vers l'intelligence artificielle d'être justement trop orientés vers l'intelligence artificielle, et de n'offrir que bien peu d'outils qui permettraient d'aborder grâce à eux des problèmes plus généraux.

2.4 Conception d'un poste de travail

Les deux types de postes de travail que nous venons de présenter semblent a priori fort éloignés l'un de l'autre. Il est cependant possible de faire une synthèse de ces deux conceptions.

Les points qui nous semblent essentiels sont principalement liés au langage:

- Utilisation d'un langage unique. (sur l'intérêt de cette notion, on consultera [Fra 83]). On ne sait pas encore encore se passer d'un langage spécialisé pour une communication non triviale avec une machine. Il est clair, cependant, qu'un langage unique, bien choisi, peut convenir. C'est ainsi que sous UNIX V7, on trouve "csh", qui est un interprète de commande destiné à remplacer le "shell" standard, et dont la syntaxe et la sémantique sont très proches de celles du langage C. Le problème avec les langages qui sont "presque" compatibles est bien entendu qu'ils ne font jamais la même chose... Nous approfondirons notre point de vue dans la partie consacrée à la réalisation du système.

- Généralisation de la notion d'objet. Typiquement, un langage de programmation manipule des objets simples et bien typés: entiers, réels, tables, enregistrements, etc. Un langage de commande, lui, va manipuler des programmes, fichiers, répertoires, chaînes de caractères. Faire coexister ces différents types d'objets ne peut que créer un langage confus et complexe, sauf si l'on sait généraliser ces notions dans un esprit de

simplification.

- Aspect interprété du langage. Les contraintes liées à la compilation imposent une liaison précoce entre les identificateurs et les objets qu'ils référencent. Au contraire, un langage de commande doit permettre de manipuler, potentiellement, n'importe quel objet du système, ce qui nécessite en général de retarder cette liaison au maximum. Un langage "universel" doit donc permettre cette liaison "au plus tard", autrement dit, renoncer en pratique à la notion de compilation en favorisant l'aspect interprétation.

- Outils de communication efficaces. Ce terme désigne essentiellement les mécanismes de dialogue avec la machine, et nous insisterons tout particulièrement sur des outils tels qu'un éditeur de texte pleine page et même multifenêtres, mais aussi les utilitaires du type formatage de texte, analyse de programmes, etc. Ces outils ne doivent pas être fermés, mais au contraire pouvoir être étendus de manière simple par des programmes écrits par l'utilisateur.

Enfin, certains points importants sont liés aux notions même introduites par les postes de travail, et pourraient se résumer en deux mots : autonomie et communication.

- Adaptation du matériel. Cet aspect avait été laissé de côté jusqu'à présent, car c'est celui qui pose le moins de problèmes. Le poste de travail doit prévoir une capacité de stockage magnétique moyenne (par exemple, 20 à 40 M octets, ce qui convient à une

très grande majorité d'applications; pour donner une comparaison, 10 M octets suffisent à conserver les programmes sources d'UNIX et de ses 200 utilitaires, ou encore de ZETA-LISP et ses 10000 utilitaires!), et un terminal alphanumérique connecté à 9600 bauds, ou, mieux, un bit map.

- Ouverture vers l'extérieur. Bien que cet aspect n'ait pas encore été abordé, il est clair que la communication avec d'autres info-sites est une préoccupation majeure des concepteurs de postes de travail. Il existe aujourd'hui des matériels et logiciels permettant de faire communiquer des sites UNIX par partage de leurs ressources fichiers. De même, les postes de travail tels la Lambda machine peuvent se connecter aux réseaux Arpanet ou Chaosnet. En effet, bien que le poste de travail soit idéal pour le développement autonome d'applications, l'exploitation effective de celles-ci nécessitera souvent la communication avec d'autres postes de travail, ou même de gros ordinateurs, pour l'accès à des bases de données, l'utilisation de périphériques spéciaux tels une photocomposeuse, etc.

2.5 Le choix d'APL

Le langage APL, que nous avons choisi comme langage de base de notre poste de travail, est, nous semble t-il, le seul à répondre presque entièrement à nos besoins.

La philosophie générale d'un système APL est fortement orienté vers une gestion efficace des contextes autonomes (langage et système sont en fait exploités à l'intérieur d'un espace de travail individuel). Elle est donc très proche de celle des postes de travail individuels.

Le langage s'intègre harmonieusement au sein d'un système qui lui est propre. Ainsi, beaucoup plus qu'un langage de haut niveau, APL apparaît comme un outil logiciel d'une rare puissance, et bénéficiant directement des services satellites de la programmation, comme l'édition, la mise au point interactive ou l'archivage. De plus, le fait qu'APL soit interprété donne à l'utilisateur la possibilité de bénéficier des services du système à tous les niveaux, en utilisant la même syntaxe.

Enfin, en consultant l'abondante littérature autour d'APL (en particulier les actes des congrès qui lui sont consacrés), on remarque qu'il est utilisé avec succès dans un grand nombre de domaines extrêmement variés. Cette universalité d'APL et, il faut le dire aussi, l'enthousiasme de ses partisans, sont de bonne augure si l'on envisage un système exclusivement basé sur ce langage.

Des environnements de programmation APL existent déja depuis une dizaine d'années, sur différents matériels qui peuvent se regrouper en trois catégories:

- Les architectures traditionnelles, réalisées essentiellement sur des ordinateurs classiques (par exemple, [Gir 76], [Ber 81b]).
- Les systèmes APL microprogrammés, dans lesquels un ensemble de microprogrammes, réalisant des fonctions

de bas niveau, mieux adaptées à APL, est mis en place (un tel exemple est [Has 73]).

- Enfin, ce que nous appellerons les machines APL. Ce sont des calculateurs autonomes (IBM 5100, MCM 700), qui proposent exclusivement le langage APL, et que l'on peut presque qualifier de postes de travail individuels.

Ces différentes implémentations présentent en outre l'avantage d'être quasiment compatibles avec la norme APL en cours d'adoption [ISO 83], ce qui est un point extrêmement positif.

Cependant, ces différentes approches reposent sur des concepts qui datent maintenant d'une quinzaine d'années. Malgré une bonne intégration du logiciel au sein du matériel, la gestion des ressources de la machine est assez limité, les outils de dialogue sont frustres, les communications avec l'extérieur sont quasi-inexistantes, le confort et la productivité de l'utilisateur restent réduits. De plus, le matériel utilisé n'étant pas en général spécifiquement adapté aux besoins d'un interprète APL, et celui-ci devant en outre se conformer aux possibilités d'un système d'exploitation conçu pour tout autre chose, on constate de faibles performances dans les applications de quelque importance.

Enfin, le langage APL, contrairement à LISP, apparaît comme très éloigné de la machine. Peut-on en attendre qu'il gère avec finesse toutes les ressources de celle-ci ?

2.6 La solution: un APL adapté ?

Cependant, il y a de fortes analogies entre LISP et APL, et nous allons montrer qu'un langage APL étendu possède en fait toutes les qualités que l'on peut attendre sur un poste de travail.

2.6.1 Les structures de base

De même que le LISP originel proposait une structure de données unique, la liste, APL propose une structure de données unique, le tableau, qui est une collection rectangulaire d'objets. Ces objets (dits aussi items du tableau) sont regroupés le long d'axes, un tableau pouvant avoir Ø, 1, 2,... ou N axes. Enfin, un tableau ayant Ø axe contient un élément unique. Le tableau est alors dit "scalaire", et peut être assimilé à cet élément unique.

Cependant, alors que LISP ajoute de nouvelles possibilités en multipliant les structures de données (vecteurs, chaînes, tableaux, tables hashcodées, etc), APL augmente sa puissance en généralisant la notion de scalaire, qui peut être un élément numérique ou caractère, mais peut aussi représenter un autre tableau. On voit que cette organisation permet maintenant de construire des objets très complexes, mais sur lesquels les primitives de restructuration d'APL, en petit nombre, s'appliquent dans toute leur généralité.

En effet, bien que la notion de tableau généralisé d'APL soit beaucoup plus riche que la structure de liste de LISP, APL fournit beaucoup moins de primitives

de manipulations de données que LISP. Ces fonctions sont en outre de plus haut niveau, plus générales, et, il faut le reconnaître, mieux conçues dans leur sémantique. Il en résulte qu'un programme APL est plus court, souvent plus général, et plus compréhensible qu'un programme LISP équivalent.

Les tableaux hétérogènes offrent toute la souplesse nécessaire à la représentation de collection d'objets. La structure de tableau décrit plus efficacement les chaînes de caractères que les listes ne permettent de le faire, favorisant en particulier l'adressage aléatoire des éléments, qui est impossible dans LISP. Une abondante littérature concerne les tableaux généralisés. On consultera par exemple [Bar 82], [Ber 81a], [Bro 79,82], [Ive 81], [Mor 82], [Ort 81], [Rab 82], [Rue 82] et [Smi 81a,b].

Cependant, afin de rendre APL effectivement utilisable comme langage de commande, il est nécessaire de rendre accessible une structure de données des systèmes APL, jusqu'alors cachée aux utilisateurs, celle de la table des symboles. Cette structure sera utilisée pour représenter les arborescences de notre système, à la manière des répertoires UNIX. En fait, l'usage des tables de symboles sera beaucoup plus général que ce qui est actuellement possible sous UNIX, et se rapprochera de celui qui peut être fait des listes de propriétés de LISP.

2.6.2 Fichiers, répertoires et programmes

Un fichier, surtout un fichier de type UNIX, peut être considéré comme une chaîne de caractères (ou, pour plus de commodité, comme un scalaire enclos

représentant un tableau de caractères), et manipulé comme tel. En fait, le notion même de fichier, survivance d'un passé obscur, nous semble amenée à disparaître. Un fichier ne nécessite plus de primitives spécifiques et restrictives: il est manipulé par les mêmes primitives que n'importe quel autre tableau APL. Certaines notions, propres aux langages de commande et liées aux objets manipulés (par exemple, pour un fichier, propriétaire, droits d'accès, dates de création et d'expiration), sont généralisées sous le terme de propriétés [Gir 82a].

Un répertoire sera une table, où l'accès se fait par nom: objet à part entière du langage APL, il devient explicitement manipulable et peut être intégré au sein de structures plus complexes.

Enfin, un programme -qui n'est autre qu'une fonction utilisateur habituelle, éventuellement extraite d'un autre répertoire que le répertoire courant- est lui aussi directement manipulable sous APL. On peut l'exécuter en lui fournissant des paramètres, on peut l'éditer, on peut aussi le combiner avec d'autres programmes au moyen d'opérateurs spécialisés.

Il est certain que l'aspect fonctionnel d'un langage comme APL devient tout à fait significatif lorsque les objets que l'on manipule sont des programmes. Ainsi l'opérateur de composition ⍤ [Ive 81], appliqué à des fonctions F et G sous la forme:

```
      G ⍤ F X
```

semble n'offrir qu'un intérêt limité. Cependant, cette notation, qui signifie formellement "appliquer l'algorithme G au résultat de F" (sous UNIX, on parle également de "filtre"), peut représenter l'exécution en

parallèle de F et G, chaque résultat scalaire de G étant transmis à F. On trouvera certains développements intéressants sur les opérations de composition de programmes dans [Shu 83], l'auteur avouant d'ailleurs s'être inspiré d'APL. On notera au passage que ce genre de notion existe déjà dans l'APL classique, puisque le produit interne est une composition des deux fonctions qui sont appliquées non pas aux opérandes du produit interne, mais bien à des scalaires individuellement extraits de ces opérandes.

2.6.3 L'environnement.

C'est probablement dans ce domaine que le plus gros effort est à faire. Il faut, en s'inspirant des réalisations qui ont fait leurs preuves, réaliser de nouveaux outils bien adaptés à APL.

Il est certain que l'environnement doit comporter, au départ, un ensemble minimum d'outils : éditeur de texte, formateur de texte, utilitaires d'analyse de programmes et d'applications, système de messagerie, etc. De nouveaux outils verront le jour au fur et à mesure de l'apparition de nouveaux besoins chez les utilisateurs.

Mais, plutôt que d'offrir une profusion d'utilitaires au fonctionnement figé, l'environnement, idéalement, doit se prêter à la personnalisation. Au lieu de se contenter des outils existant, qui souvent ne répondent qu'en partie à ses besoins, ou encore de devoir concevoir ex nihilo un outil qui lui convienne, l'utilisateur doit pouvoir adapter très aisément les outils disponibles. C'est bien ce genre de démarche que l'on trouve sur les postes de travail LISP, c'est aussi

celle que l'utilisateur expérimenté d'APL adopte le plus fréquemment.

Ces choix remettent en cause la notion habituelle d'utilitaire. En effet, dès qu'un utilitaire doit dialoguer avec l'utilisateur, l'approche classique consiste à réaliser, au sein de cet utilitaire, un interprète d'un langage spécialisé. Sous UNIX, c'est naturellement le cas du shell, mais aussi de l'éditeur de texte, des formateurs de texte (NROFF, TROFF et GROFF que l'auteur remercie au passage pour la belle présentation de ce rapport), des filtres (GREP, AWK), et, plus généralement, d'un certain nombre d'outils réalisant diverses fonctions non triviales. Malgré un désir évident de respecter certaines conventions, tous ces langages sont fondamentalement différents, restreints dans leurs possibilités, et nécessitent une certaine période d'adaptation de la part de l'utilisateur. Enfin, la programmation de tels utilitaires est en général complexe, la réalisation du seul langage de commande demandant souvent plus de travail que celle des fonctions spécifiques que doit remplir l'utilitaire!

Une approche à la fois plus rationnelle, plus économique et plus générale, est celle qui a été adoptée dans l'éditeur EMACS. L'utilisateur travaille dans un environnement dans lequel sont disponibles un certain nombre de primitives (100 à 400) gérant des objets spécialisés: fenêtres du terminal, zones tampons, fichiers de travail... Ces primitives peuvent être utilisées directement, éventuellement par l'intermédiaire d'abréviations du genre "ESC B" ou "Contrôle-X Contrôle-Z", ou encore peuvent être appelées depuis des programmes écrits en LISP,

bénéficiant alors de toute la puissance d'expression de ce langage. (On notera au passage que l'éditeur de texte LISP, EMACS, a été porté sous UNIX en C, fournissant EMACS-UNIX, quasi identique [Gos 81]; détail amusant et significatif, il a été intégré au sein de cet éditeur un mini-interprète d'un simili-LISP, permettant à l'utilisateur d'étendre les fonctionnalités de cet EMACS comme l'utilisateur de LISP le fait...)

Les différents utilitaires d'un poste de travail APL nous semblent donc devoir se présenter sous la forme d'environnements (jadis: zones de travail), dans lesquels on trouvera des "briques" logicielles spécialisées, probablement écrites dans un langage compilé, et de courtes fonctions APL réalisant le liant, qui seront elles aisément adaptables par l'utilisateur.

2.6.4 Les contraintes de réalisation du système

Il paraît utile, pour conclure cet partie, de dire quelques mots sur la réalisation effective d'un tel système.

Le choix du langage utilisé pour l'implémentation nécessite quelques explications. Il ne faut pas déduire de notre approche que nous envisageons APL comme unique langage utilisable sur notre poste de travail, ni qu'il sera utilisé pour la programmation du système, comme c'est le cas dans les systèmes à base de LISP. En effet, le noyau de base du système, gérant les ressources de la machine et assurant l'interprétation du langage APL, doit être écrit dans un langage proche de la machine, en l'occurence un langage de type C

convenant assez bien. Le logiciel de base sera lui-même réalisé partiellement en APL, et partiellement en langage de base. Le système assurera une liaison intime entre modules APL et modules compilés, les uns pouvant faire appel aux autres et réciproquement, cette approche laissant de surcroît la porte ouverte à l'intégration d'un compilateur APL.

Naturellement, les options choisies impliquent certaines conséquences dans la réalisation. Par exemple, la notion de commande système apparaît mal adaptée pour les communications avec le système, qui deviennent beaucoup plus critiques lorsqu'APL sert de langage de commande. Enfin, pouvoir manipuler potentiellement n'importe quel objet du système nécessite une conception originale de l'interprète et de son mécanisme d'adressage des objets, remettant en cause la notion de zone de travail.

C'est à cet aspect plus particulier d'un tel système APL que nous nous intéresserons dans la suite de ce chapitre.

2.7 L'univers des objets.

Traditionnellement, les objets gérés par un système APL correspondent à une hiérarchie à deux niveaux:

- Les objets du langage : fonctions et variables. Ces objets sont regroupés en zones de travail, dites "WS".

- Les WS elles-mêmes : elles ne sont pas connues de l'interprète du langage, et doivent être manipulées par des commandes spécialisées.

Un problème majeur d'APL est donc lié à l'espace de désignation des objets : l'interprète ne sait manipuler que les objets de la WS active. Il ne lui est pas possible d'accéder un objet d'une autre WS, sans recopier au préalable celui-ci dans la WS active. Cette opération est longue et coûteuse dans presque tous les systèmes. Enfin, il ne lui est pas possible d'aller écrire sélectivement un objet dans une WS inactive.

Sans tomber aussi bas que BASIC, qui ne permet de manipuler qu'une fonction à la fois (et encore!), cette limitation dans la désignation des objets représente souvent une gêne considérable lorsqu'il est nécéssaire de faire appel à des modules écrits par d'autres utilisateurs.

Deux phénomènes viennent encore compliquer cet état de choses: d'une part la limitation de la taille des WS (quelques dizaines à centaines de kilo-octets), d'autre part la vision "à plat" de la désignation des objets, problème déjà évoqué dans le cas des systèmes LISP. En particulier, la limitation en taille des WS entraine des contraintes importantes sur le nombre et la taille des objets, ce qui implique souvent de recourir à des solutions utilisant des fichiers, avec tous les désagréments que celà entraine.

Un autre problème - peut-être moins fondamental - est lié à la difficulté de communication entre deux processus APL : un échange d'objet doit se faire, soit par l'intermédiaire d'un fichier disque, soit encore

(dans les systèmes de type APLSV), par l'intermédiaire d'un processus de partage, ce qui, dans les deux cas, implique un double transfert de valeur et un stockage intermédiaire.

Enfin, un dernier problème qu'il paraît utile de citer est celui de la conservation des objets, qui nécéssite une écriture physique de la totalité du contenu de la WS active dans un fichier spécialisé, la WS inactive. Cette écriture doit être demandée explicitement par l'utilisateur. Elle peut être fréquente, donc coûteuse. Elle peut être moins fréquente, avec le risque de perdre, en cas de problème système, l'ensemble du travail effectué depuis la dernière sauvegarde.

Manifestement, ces problèmes sont historiquement liés aux conceptions "classiques" des architectures de système : les objets sont représentés par une partie de la mémoire centrale de la machine hôte.

L'utilisation de machines munies de dispositifs matériels de mémoire virtuelle n'a guêre changé cet état de choses. Même si la taille d'une WS a progressé d'un ordre de grandeur, les autres problèmes ont subsisté, avec plus d'actualité que jamais.

La solution nous semble passer par l'application des deux règles suivantes :

- Tous les objets existant dans le système doivent faire partie de l'espace d'adressage de l'interprète,

- Tous les objets existant dans le système doivent

pouvoir explicitement être désignés par l'utilisateur.

La première de ces règles impose donc le recours à un très large espace d'adressage, non pas comparable à l'espace d'adressage d'un micro-processeur (deux puissance vingt ou vingt-quatre), mais beaucoup plus important, de l'ordre de deux puissance quarante ou même plus. Il devient donc nécéssaire de faire appel, au sein du système, aux techniques de gestion de mémoire virtuelle.

La seconde règle nécéssite l'introduction dans le langage de la notion de désignation hiérarchisée. Il devient également nécéssaire de prendre en compte la notion de protection des objets. Nous ne nous étendrons pas sur ce point, qui est abordé plus en détail dans [Gir 82a].

Il est facile de se convaincre que cette approche répond à l'ensemble des problèmes évoqués ci-dessus. En effet, le fait d'avoir un très grand espace d'adressage facilite l'implémentation d'objets de taille importante et rend ainsi inutile l'emploi de fichiers. Enfin, la désignation hiérarchisée permettra de gérer plus facilement et avec beaucoup plus de sécurité des ensembles d'objets réalisant une application spécifique, et donc l'abandon de la notion de WS.

En se plaçant dans le domaine des postes autonomes, le partage des objets sera nécessaire à l'intérieur d'une configuration de réseau local. L'utilisateur pourra accéder à un objet "distant" (appartenant à un autre site) en établissant un "lien", ensemble d'informations localisant sans ambiguité l'objet. A

partir du moment où le lien est établi, l'utilisateur peut effectuer toutes les opérations autorisées par les attributs (caractéristiques) de l'objet.

2.8 Une amélioration de la gestion des objets

Naturellement, le reproche majeur que l'on peut adresser à la solution proposée est celle de son inefficacité potentielle. Alors que la sémantique relativement complexe des fonctions APL impose déjà des vérifications coûteuses lors de l'interprétation du langage, on peut se demander à juste titre si le recourt à une mémoire virtuelle ne va pas entraîner des coûts prohibitifs d'exploitation.

Nous allons tenter de montrer qu'en fait cette approche est tout à fait réalisable sur une architecture de machine classique, et qu'un tel système APL ne sera guêre plus coûteux qu'un système classique.

Une analyse plus fine du fonctionnement d'un système APL montre qu'en fait les accès aux objets eux-mêmes ne représentent qu'une faible partie des accès mémoire effectués par le processeur : Au mieux, un sur dix, souvent moins de un sur cent.

En première analyse, nous nous trouvons donc face à deux types d'accès:

- Les accès au programme réalisant l'interprètation du langage. Ils sont très fréquents (de 8Ø % à 99 % du total) et concernent un objet de relativement petite

taille (bien que celle d'un bon système APL puisse atteindre plusieures centaines de kilo-octets). L'interprète APL peut donc être résident en mémoire centrale, ce qui assure donc le temps minimal d'accès aux instructions machine.

- Les accès aux objets du langage. Ils sont peu fréquents, mais concernent des données qui ne sont pas directement accessibles par le processeur, la taille de l'espace des objets dépassant largement les capacités d'adressage des processeurs existant.

Si la gestion de notre espace virtuel est réalisée uniquement par du logiciel, l'accès à un élément devient relativement coûteux : le processeur, au lieu d'une instruction machine, va peut-être devoir en exécuter cinquante ou cent. Cependant, compte tenu de la faible proportion de ce type d'accès dans l'activité d'un système APL, celui-ci ne serait globalement que deux à vingt fois plus lent qu'un système classique, ce qui, dans certains cas, serait encore acceptable en regard des avantages proposés.

Essayons de mieux décrire les caractéristiques de cet espace virtuel.

- Ce n'est pas un espace virtuel au sens classique du terme (par exemple, l'espace virtuel géré par la base de données SOCRATE), mais plutôt un espace qui va en fait correspondre à l'ensemble des supports magnétiques disponibles sur la configuration, ou plus généralement, sur le réseau.

- Cet espace va contenir des objets, correspondant aux variables APL, de petite ou moyenne taille

(typiquement, quelques dizaines à quelques centaines de milliers d'éléments), et qui sont en général accédés dans leur totalité. On notera que cette taille d'objet est tout à fait compatible avec les possibilités d'adressage d'un micro-processeur.

En outre, l'interprète APL n'accède simultanément qu'à un petit nombre d'objets. Ces objets sont ceux qui interviennent dans un traîtement local : exécution d'une affectation, d'une fonction monadique ou dyadique, d'un opérateur, etc.

L'idée est donc de disposer sur le poste de travail, d'un mécanisme matériel, permettant au processeur d'accéder simultanément aux instructions de l'interprète et aux contenus d'un petit nombre d'objets spécifiés à l'avance.

Ce mécanisme s'apparenterait aux unités de gestion de mémoire classiques, à la nuance suivante : en fonctionnement normal, cette unité de gestion de mémoire orientée APL (nous dirons : UGM-APL) est inactive, et permet l'accès à la totalité de la mémoire physique. Sur une commande d'activation issue du processeur, l'UGM-APL permet l'accès aux éléments d'un objet spécifié à l'avance. L'élement est spécifié par son adresse à l'intérieur de l'objet, cette adresse étant disponible sur le bus adresse lorsque le processeur exécute une commande de chargement ou de déchargement.

Le but est d'obtenir, grâce à ce mécanisme, des performances comparables à celles d'un système APL classique, tout en bénéficiant de avantages décrits ci-dessus.

Les chapitres suivants explorent plus en détail cette idée, d'abord en examinant ses implications au niveau du logiciel, ensuite en proposant une description matérielle de cette UGM-APL, réalisable sur une carte en composants discrets.

Enfin, les annexes font le point sur les performances attendues, et décrivent une simulation de l'UGM-APL.

2.9 Conclusion

Nous avons tenté de montrer la faisabilité d'un poste de travail conçu autour du langage APL, langage de haut niveau aux possibilités multiples. Notre approche est restée exclusivement logicielle, afin de pouvoir utiliser les matériels existant. Nous verrons dans les chapitres ultérieurs qu'un gain important d'efficacité pourrait être obtenu par l'utilisation de composants matériels spécialisés.

2.10 BIBLIOGRAPHIE

[Bar 82] Sylvain Baron. Les tableaux généralisés. Journées APL de l'AFCET. Avril 1982.

[Ber 81a] Robert Bernecky. Representation for enclosed arrays. APL 81 ACM conference proceedings.

1981.

[Ber 81b] Christian Bertin. Aspects de la réalisation d'un système APL optimisé. Thèse de Docteur-Ingénieur. Ecole des Mines de Saint-Etienne. Décembre 1981.

[Bro 79] James A. Brown. Evaluating Extensions to APL. APL 79 ACM Conference Proceedings.

[Bro 82] James A. Brown. Understanding Arrays. APL 82 ACM Conference Proceedings.

[Fra 83] Christopher W. Fraser, David R. Hauson. A High-Level Programming and Command Language. in ACM Sigplan notices. Proceedings of the Sigplan 83 symposium on programming languages issues in software system. San Francisco, June 27-29, 1983.

[Gir 76] Jean-Jacques Girardot, François Mireaux. Réalisation d'un interprète complet du langage APL sur un mini-ordinateur. Thèse de Docteur-Ingénieur. Université de Nancy 1. Septembre 1976.

[Gir 82a] Jean-Jacques Girardot. La notion de sécurité dans une application. Journées APL de l'AFCET. Avril 1982.

[Gir 82b] Jean-Jacques Girardot. Architecture logicielle et matérielle d'une machine APL. Journées APL de l'AFCET. Novembre 1982.

[Gos 81] James Gosling, UNIX EMACS, C.M.U., January 81.

[Has 73] A. Hassit, L. E. Lyon. Implementation of a High level language machine. Communications of the ACM, Vol 14 n. 4. Avril 1973.

[ISO 83] Fifth Working Draft Standard for Programing Language APL. Document ISO TC97/SC5 WG 6 N38 Juin 1983.

[Ive 81] K. E. Iverson, R. Bernecky. Operators and enclosed Arrays. 1980 APL Users Meeting.

[Mor 82] T. More. Rectangularly Arranged Collections of Collections. APL 82 ACM Conference Proceedings.

[Ort 81] Don. L. Orth. A user's view of General Arrays. IBM T.J. Watson Research Center. 1981.

[Rab 82] Dave Rabenhorst, APL2 Language Reference Manual IBM. Juin 1982.

[Rue 82] K. F. Ruehr. A survey of extensions to APL. APL 82 ACM Conference Proceedings.

[Sak 83a] Ulysse Sakellaridis. An experimental APL system based on a multi-processor architecture. IEEE Mélécon 1983. Athènes, Mai 1983.

[Sak 83b] Séga Sako. Aspects de la gestion mémoire du système APL 90. Rapport de DEA. Ecole des Mines de Saint-Etienne. Septembre 1983.

[Shu 83] John Shultis. A Functional Shell. in ACM Sigplan notices. Proceedings of the Sigplan 83 symposium on programming languages issues in software system. San Francisco, June 27-29, 1983.

[Smi 81a] Bob Smith. Nested Arrays, Operators and Functions. APL 81 ACM Conference Proceedings.

[Smi 81b] Bob Smith, APL*PLUS Nested Arrays System. STSC Inc. Mars 1981.

[Tei 78] Warren Teitelman. INTERLISP Reference Manual XEROX Palo Alto Research Center. 1978.

[Bel 78] The UNIX Time-Sharing System. The Bell System Technical Journal Juillet-Août 1978.

[Wei 83] Daniel Weinreb, David Moon & Richard Stallman. LISP Machine Manual. MIT, January 83.

CHAPITRE 3 Aspects généraux de l'UGM-APL

Ce troisième chapitre est consacré au fonctionnement logique d'une unité de gestion mémoire orientée APL (UGM-APL). La présentation faite ici est volontairement générale, afin de permettre de dégager les concepts fondamentaux en faisant abstraction de certaines considérations (ou contraintes) matérielles et logicielles, liées à une réalisation sur un système donné.

3.1 Introduction

On a vu dans le chapitre précédent que lors de la réalisation d'un système APL, la gestion de la mémoire des données constitue un point délicat, ceci essentiellement à cause du grand dynamisme des données. Cette particularité amène la quasi-obligation de concevoir des fonctions d'accès et de contrôle des données qui soient très efficaces. De plus, comme dans cette nouvelle conception d'un environnement APL pour un poste de travail autonome on veut faire disparaître un certain nombre de limitations ou encore introduire de nouveaux concepts, l'efficacité de ces fonctions est encore plus nécessaire.

Par la suite, nous allons nous intéresser à ce problème des accès et des contrôles des données. Pour ce faire, nous definirons un processus spécialisé qui regroupera toutes les fonctionnalités d'accès et de contrôle des données et qui sera au service de l'interprète APL. Plus précisement, la définition d'un tel processus sera guidé par les points suivants :

- Tout d'abord, il faut respecter les fonctionnalités et avantages d'un grand univers unique d'objets. Cet univers, dans une configuration de réseau local, sera réalisé par un ensemble d'espaces répartis entre les différents postes du réseau. Dans un tel

contexte, pour chaque poste il y aura un tel processus capable de gérer les objets résidents et mentionner les accès aux objets distants (appartenant à un autre poste).

- Les mécanismes spécifiques à une mémoire, sur laquelle sera réalisé cet univers unique, devront être pris en compte par ce processus, et gérés en grande partie par lui.

- Un certain nombre de contrôles sur les objets seront effectués directement par lui, en facilitant ainsi la conception et la réalisation de l'interprète.

- Enfin, sa conception sera aussi basée sur un compromis entre la complexité et l'efficacité, pouvant ainsi nous amener à moyen terme à une réalisation pûrement matérielle (sous forme d'une carte ou d'un processeur spécialisé).

Notre "serveur d'objets" regroupera toutes les actions pour mener à bien la gestion d'un espace d'objets APL. Par la suite nous le désignerons sous le nom d'Unité de Gestion Mémoire orientée APL, ou encore: UGM-APL. Ce chapitre présente en details son fonctionnement logique, tandis que le chapitre suivant donnera les éléments matériels et logiciels nécessaires à sa réalisation.

3.2 Le principe de fonctionnement

Pour bien dégager le principe de fonctionnement de l'UGM-APL, revenons sur une spécificité d'un environnement APL :

Le langage APL est interprété. L'exécution d'un programme APL nécessite donc, d'une part le texte et les données de ce programme APL, d'autre part le texte du programme constituant l'interprète APL lui-même. On peut classer les accès mémoire effectuéspar le processeur en deux catégories :

La première catégorie regroupe les accès aux propres instructions du système et de l'interprète, pour pouvoir les exécuter. Dans cette même catégorie nous classons aussi les accès aux données de l'interprète lui-même (tables, pile d'exécution, accès aux objets du système,...).

Dans la deuxième catégorie on trouve les accès provoqués par l'interprète et visant les objets utilisateur (ou les temporaires) qu'il doit traiter.

On peut constater assez facilement que les accès de la première catégorie sont les plus fréquants :

- L'analyse d'une expression APL qui s'effectue avant chaque nouvelle exécution, est réalisée entièrement par les programmes de l'interprète. De plus, toute création d'objet temporaire demande une phase d'initialisation qui est indépendante de sa taille et qui n'affecte pas l'espace des objets.

- Différentes mesures effectuées sur APL\800 (système réalisé sur le mini-ordinateur PHILIPS P857) montrent que lors de l'interprétation d'une expression APL, 15 à 20% seulement des accès du système sont utilisés pour les données (et c'est probablement un maximum! La moyenne est certainement < 1% !).

Dans les systèmes APL recents, compte tenu du coût de plus en plus faible de la mémoire, la totalité du code de l'interprète est résidente en mémoire centrale, évitant ainsi une degradation de performances dûe au recouvrement.

Cette particularité de l'environnement APL dicte une contrainte fondamentale dans la conception de notre UGM-APL : l'accès par le processeur aux instructions de l'interprète, doit être le plus performant possible. Autrement dit, le temps de traversée de cette unité lors de la recupération des instructions de l'interprète doit être minimal.

Après ces remarques, abordons maintenant le principe de fonctionnement de l'UGM-APL, ce dernier étant

etroitement lié aux deux catégories d'accès mentionnées précédement :

- Lorsque le processeur accède aux instructions de l'interprète, il ignore complètement l'existence d'utilitaire de gestion mémoire. L'UGM-APL est alors dans un état que l'on peut qualifier d'"inactif". Ce mode d'accès, optimal du point de vue de temps de réponse, permet au processeur d'adresser toute la mémoire centrale. Néanmoins, cette approche implique une rigueur de programmation du système et de l'interprète, essentiellement parce que de tels accès aux objets seront dépourvus de toute sorte de sécurité, le contrôle d'accès aux objets étant assuré en grande partie par l'UGM-APL.

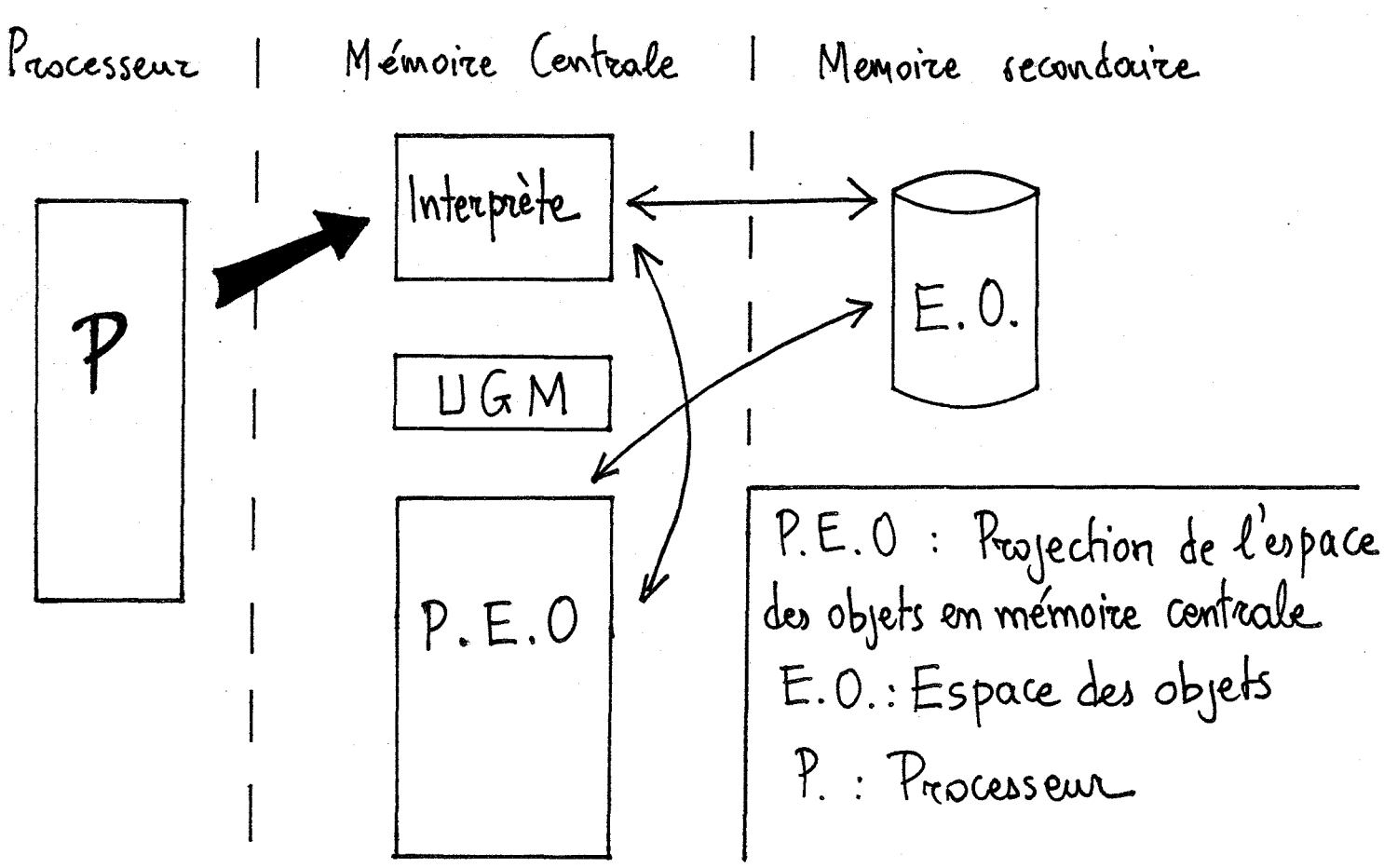
- L'interprète accède toujours les objets à travers l'UGM-APL. Pendant ces accès, l'UGM-APL est dans l'état "actif" : contrôle de la validité de l'accès, gestion de la topographie de l'espace, traduction des adresses, signalement des défauts de page et autres événements imprévus, etc...

Le schéma suivant illustre le principe de fonctionnement actif-inactif de l'UGM-APL :

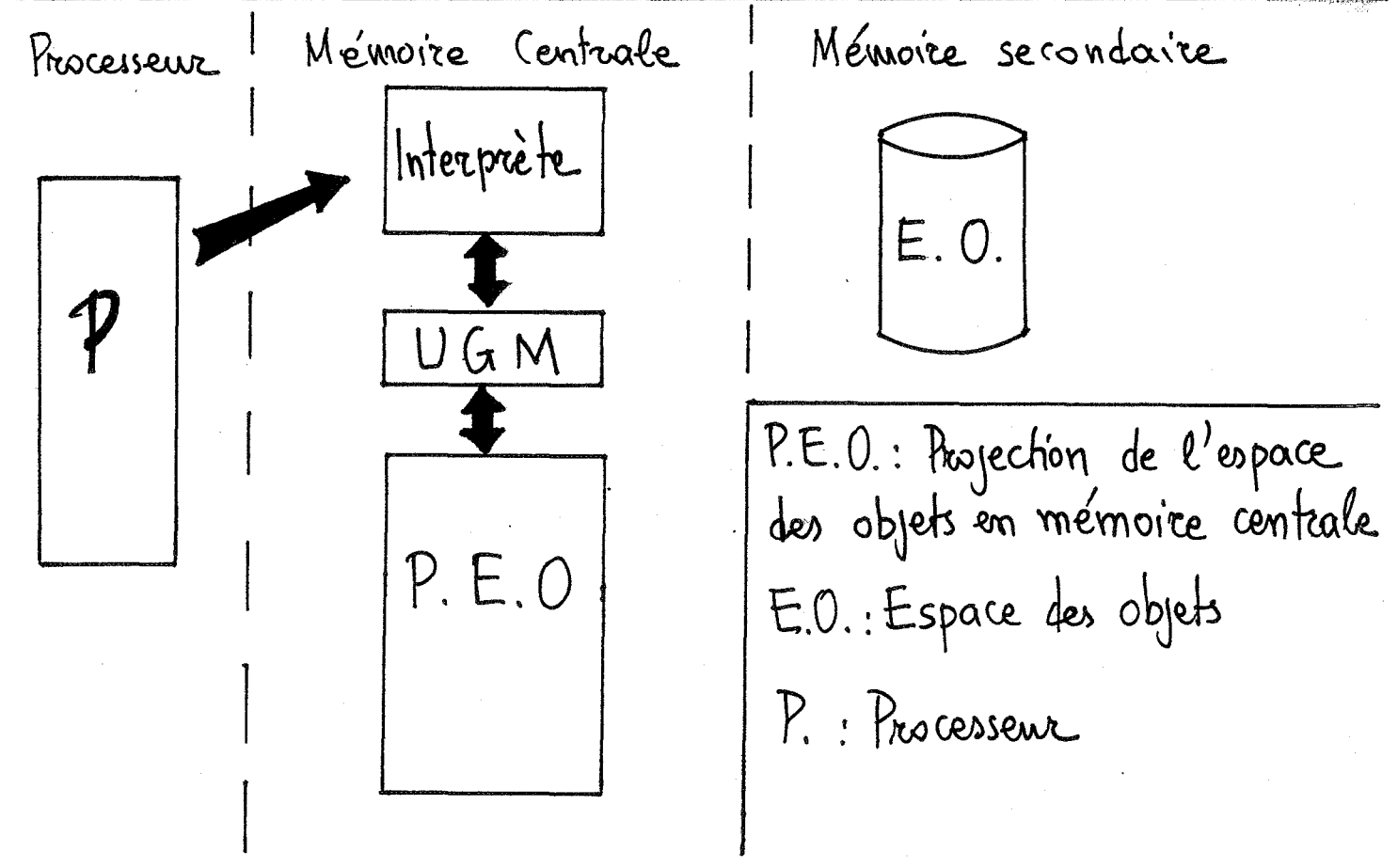
== a) Le processeur exécute les instructions de l'interprète qui se trouvent en mémoire centrale. Dans ce cas, l'UGM-APL est ignorée (état inactif). Néanmoins, l'interprète peut accéder, s'il le désire, à l'espace des objets, pour effectuer des opérations

spécifiques rapides (modification d'une table, installation ou ré'ecriture d'une page,...).

== b) Il s'agit ici du cas où, l'interprète veut accéder normalement aux objets lors de l'évaluation d'une expression APL. Pour se faire, il demande les services de l'UGM-APL en l'activant. La fin d'un accès aux objets est marqué par le passage de l'UGM-APL dans l'état inactif.



(a) Fonctionnement du système en état inactif de l'UGM-APL



(b) Fonctionnement du système en état actif de l'UGM-APL

Enfin, il existe un troisième point détaillé ultérieuement et qui concerne le contrôle exercé par l'interprète sur l'UGM-APL (différentes commandes d'initialisation, de test ou de mise à jour des données de l'UGM-APL).

On peut, en quelques lignes, montrer en quoi notre UGM-APL est différente dans sa finalité des autres types de gestionnaires des mémoires :

- Une unité de gestion mémoire "classique" en général gère tous les accès faits par le processeur, contrairement à l'UGM-APL qui s'occupe uniquement des accès aux objets utilisateur.

- Une unité "classique" gère une mémoire virtuelle de taille inférieure ou égale à l'espace d'adressage du processeur. Ici, ce n'est pas non plus une très grande mémoire virtuelle (de type Socrate : mémoire repliée sur un espace réel) qui nous intéresse. Aussi bien l'interprète que l'UGM-APL utiliseront presque directement la mémoire physique secondaire.

- Enfin, l'UGM-APL a un double rôle, puisque d'une part elle assure une traduction d'adresse (vision "paginée" de la mémoire des objets), et d'autre part elle gère un niveau supplémentaire, à savoir le niveau "objet" (vision "segmentée" de la mémoire, chaque segment correspondant à un objet).

En résumé, on peut dire que la vision de l'UGM-APL, lorsqu'elle est dans l'état actif, se limite à la partie de l'espace des objets qui est projetée en mémoire centrale. De plus, elle est toujours sous le contrôle absolu de l'interprète, au même titre qu'un processeur spécialisé et, en cas d'erreur (défaut de page, accès interdit), son action se limite à un strict compte rendu des faits.

3.3 Le mode d'adressage et le rangement des objets dans l'espace

Les différents choix de réalisation et de gestion d'un grand espace unique d'objets, conditionnent fortement non seulement la réalisation mais aussi la conception d'une unité de gestion mémoire. Dans ces paragraphes nous présentons la philosophie générale d'implémentation de l'espace des objets et l'adressage de ces objets dans l'espace.

Le concept d'un grand espace contenant des objets de taille parfois assez importante, nous amène naturellement à la notion de système à mémoire virtuelle, dont les avantages sont bien connus :

- Un processus qui s'exécute dans un tel système, accède les différents objets au moyen d'un adressage logique. Cette forme d'adressage donne accès à un espace mémoire beaucoup plus important que la mémoire centrale disponible, et lève ainsi des limitations sur la taille des objets.

- La mémoire centrale est allouée automatiquement par simple demande d'un processus. De ce fait, l'utilisateur n'a pas besoin d'estimer la taille

mémoire à allouer avant l'exécution d'un processus.

- Les différents objets accédés par un processus peuvent avoir des tailles très variables. La technique de partitionnement des objets importants (segmentation ou pagination) permet au système de ne charger en mémoire centrale qu'une fraction de chacun des objets. Ainsi on évite l'encombrement de la mémoire centrale par les objets de grande taille, mais en revanche la présence simultanée des parties de plusieurs objets différents est facilitée.

- Ces systèmes permettent la mise en place de grosses applications à un prix de revient assez faible: la mémoire virtuelle est réalisée sur un espace secondaire magnétique à accès aléatoire (disques) et une mémoire centrale de taille moyenne (de l'ordre d'1 Méga-octet) permet d'avoir des performances très satisfaisantes.

De manière générale, la pagination et la segmentation constituent les deux mécanismes de base d'implémentation d'une mémoire virtuelle :

- Dans une mémoire virtuelle segmentée, l'espace d'adressage logique (univers des objets) est organisé en segments de taille variable et à des adresses logiques contigues.

- La technique de la pagination consiste en un partitionnement de l'espace d'adressage logique en pages de taille fixe, à des adresses logiques contigues.

De plus, ces mécanismes sont complétés par les algorithmes qui dictent la politique générale de stockage de l'information :

- Algorithmes de remplacement : ils déterminent les informations qui doivent être retirées de la mémoire. En d'autres termes, ce sont des processus qui créent des regions mémoire disponibles.

- Algorithmes de recherche : ils déterminent l'instant où la donnée doit être chargée. Eventuellement, ils "prévoient" à l'avance une demande de transfert.

- Algorithmes de placement : ils déterminent l'endroit exact où la donnée doit être chargée. Autrement dit, ils choisissent une région non allouée.

Mais comparons les deux systèmes :

De manière générale, la segmentation est une technique plus difficile et plus délicate à mettre en

oeuvre que la pagination :

- Les segments à allouer, de part leur définition ont des tailles variables. Pour celà, il est nécessaire de fournir des stratégies de gestion de segments.

- Les critères de selection d'une région libre doivent être clairement définis, afin de répondre avec succès à la demande d'allocation. Or, comme les segments ont, a priori, des tailles variables, il est quasiment impossible de trouver une région libre ayant exactement la taille demandée. D'où la nécessité de gérer les résidus ou de procéder à des compactages de la mémoire disponible.

- A la libération d'une région, il est parfois nécessaire de regrouper des régions contigües pour permettre l'allocation des segments de taille importante.

- La fragmentation externe de la mémoire (la fragmentation interne en segmentation n'a pas de sens), telle qu'elle apparait au bout d'un certain temps de fonctionnement, nécessite périodiquement une restructuration de l'allocation de la mémoire (algorithmes de compactage et "ramasse-miettes").

Par contre, la technique de pagination présente plusieurs avantages :

- L'implémentation d'une mémoire virtuelle paginée permet un traitement "uniforme" de la mémoire disponible de la part de ses propres outils, et ceci parce que les algorithmes qui réalisent la pagination "voient" la mémoire centrale simplement comme un ensemble anonyme de blocs de stockage. Dans le cas de la segmentation, la mémoire centrale est considérée comme un ensemble de "trous" de taille variable.

- Les transports de pages (taille fixe) entre les deux niveaux de mémoire sont plus faciles à mettre en oeuvre que seux des segments (taille variable).

- La fragmentation interne (dans les mémoires paginées, la fragmentation externe n'a pas de sens) dans les systèmes paginés peut être contrôlée par deux paramètres :

== la taille plus ou moins grande de la page (en général, la page constitue l'unité de transfert dans une telle hiérarchie de mémoire) ;

== la définition plus ou moins fine d'un découpage de la page elle-même que nous appellerons par la suite sous-page, et qui constitue l'unité d'allocation pour les différentes demandes de mémoire.

L'extrème dynamisme des objets APL est la raison principale qui nous a amené à choisir la technique de pagination pour l'implémentation de l'univers unique

des objets. De plus, la pagination étant une technique plus facile à mettre en oeuvre que la segmentation, ce choix permet une réalisation plus aisée d'une unité de gestion mémoire, presque entièrement en matériel.

Ainsi, l'univers d'objets APL sera réalisé sur un espace virtuel paginé, dont voici les principales caractéristiques :

- L'espace virtuel est réalisé sur une mémoire périphérique de taille importante et à accès aléatoire. Dans les cas concrets, comme par exemple celui d'un poste de travail, il s'agit d'un ou plusieurs disques magnétiques.

- Cet espace est partitionné en pages, fragments de mémoire de taille faible et fixe. La page constitue l'unité de transfert entre l'espace de réalisation de la mémoire virtuelle et la mémoire centrale. La taille d'une page est calculée et fixée d'avance et elle est en rapport etroit avec la grande diversité des informations à stocker, afin de permettre une bonne gestion de la fragmentation interne.

- Toujours en fonction des caractéristiques des informations à stocker, chaque page est divisée en un nombre fixe de sous-pages. Ces dernières constituent l'unité d'allocation. De cette manière, on obtient une meilleure utilisation de l'espace pour les objets dont la taille est inférieure à celle d'une page.

Voyons maintenant comment un objet APL est désigné à l'intérieur d'un espace virtuel paginé.

Lorsque le processus "interprète APL" veut avoir accès à un objet APL résidant dans l'univers unique des objets, il emploie une désignation propre à chaque objet. On appellera par la suite cette désignation référence universelle. Il s'agit d'une chaîne de bits de taille fixe et relativement réduite regroupant les informations suivantes :

- L'adresse virtuelle de la première page de l'objet (plus éventuellement le déplacement en sous-pages). En fait, cette information n'est rien d'autre qu'une adresse disque, ceci parce que l'adresse d'un objet en mémoire centrale ne peut être que temporaire.

- Le numéro de l'espace où se trouve la première page de l'objet. On considère qu'à un site donné, peuvent être associés un certain nombre d'espaces, chaque espace étant formé d'un ensemble contigü de pages.

- Le numéro du site où se trouve l'espace. Cette information prend toute son importance si on considère que le "milieu" naturel d'un poste de travail est le réseau local.

- Un ensemble d'informations propre à l'objet, comme par exemple les droits d'accès ou son type. On appellera par la suite ces informations les attributs de l'objet.

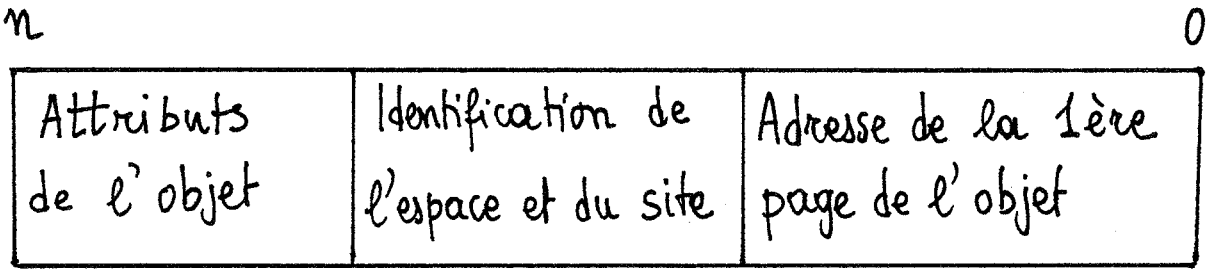


Schéma 1 : Définition générale d'une référence universelle (chaîne de n bits)

L'avantage principal de la désignation d'un objet au moyen d'une référence universelle est que pour toute l'existence de l'objet (depuis sa création jusqu'à sa destruction), celui-ci est désigné toujours avec le même "nom" unique par tous les processus locaux ou distants (processus d'un autre site). Néanmoins, les informations concernant l'objet qui résident dans une référence universelle, imposent un certain nombre de restrictions sur l'emplacement de celui-ci à l'intérieur de la mémoire virtuelle :

- La référence universelle d'un objet représente l'adresse virtuelle de la première page allouée à l'objet. Un objet, suivant sa taille peut être monopage ou multipage (peut être contenu sur une ou plusieurs pages). Lorsqu'il est multipage, alors il commence obligatoirement au début d'une page et, ses pages sont contigües dans l'espace virtuel d'adressage.

- Les pages d'un même objet, lorsque celui-ci est multipage, ne peuvent pas appartenir dans plusieurs sous-espaces différents du même ou d'un autre site. Notons ici que pour des raisons de portabilité des applications et de maintenance plus aisée du système, l'espace des objets d'un site peut être partitionné en sous-espaces.

- Enfin, un objet monopage commence toujours en frontière de sous-page et ne peut pas se trouver réparti sur deux pages différentes.

Tenant compte de ces restrictions, l'adressage par un processus d'un élément d'un objet peut s'effectuer par simple indexation à partir de l'adresse virtuelle figurant dans sa référence universelle (adresse de sa première page). D'autre part, du fait que les différentes pages d'un même objet sont contigües, un processus peut exercer un contrôle global sur l'objet en le considérant comme un segment paginé. C'est justement l'ensemble des attributs figurant dans la référence universelle de l'objet qui peuvent être utiles pour ce type de contrôles.

3.4 Le traitement local et les correspondances entre désignations

Dans les paragraphes précédents, nous avons présenté le principe de fonctionnement actif-inactif de l'UGM-APL. Aussi, nous avons détaillé la manière avec laquelle les objets sont rangés dans un espace virtuel paginé, ainsi que le moyen de les désigner (référence universelle). Par la suite, nous étudierons le comportement général de l'UGM-APL pendant les différentes phases de son utilisation. Enfin, nous présenterons les structures des données et les algorithmes nécessaires à son fonctionnement.

Essayons, tout d'abord, de placer le contexte dans lequel les objets utilisateur sont accédés par le processus interprète.

De manière générale, l'analyse d'une ligne (plus précisement d'une expression) APL, est décomposée en une suite d'opérations élémentaires de la forme :

<temporaire> ::=
[<paramètre gauche>] <fonction> [<paramètre droit>]

Prenons un exemple : inserer la chaîne X dans la chaîne Y à partir de son G-ième indice :

```
(G↑Y),X,G↓Y
```

Cette expression APL peut se décomposer de la façon suivante :

```
[1]   T1 ← G↓Y
[2]   T2 ← X,T1
[3]   T3 ← G↑Y
[4]   T_RES ← T3,T2
```

Pour chaque opération élémentaire (chacune des lignes [1] à [4]), l'interprète a besoin d'accéder jusqu'à trois objets : les deux premiers sont les paramètres de la fonction (dans ce cas toutes les fonctions primitives sont dyadiques), le troisième sert de résultat intérmédiaire. En plus, très fréquement, une fonction primitive est amenée à travailler itérativement sur des objets (c'est le cas par exemple de l'application de la fonction scalaire "+" dans l'expression A+B, où A et B sont des tableaux).

On remarque ainsi une certaine "localisation" en ce qui concerne les opérations élémentaires. En d'autres termes, on peut mentionner les deux caractéristiques concernant les opérations élémentaires :

- Une opération élémentaire utilise pour son exécution un algorithme de fonction primitive et nécessite l'accès à un petit nombre d'objets.

- Les algorithmes d'une opération élémentaire sont très souvent itératifs.

Partant de cette deuxième caractéristique, on remarque que l'accès à un élément d'un objet pendant une opération élémentaire doit être très rapide parce qu'il intervient souvent. Or, ces accès risquent d'être assez pénalisants si, chaque fois que l'interprète adresse un élément d'un objet, il utilise sa référence universelle. On peut surmonter cet inconvéniant en approfondissant cette notion de "localisation" d'une opération élémentaire, déjà quasi-naturelle au niveau d'APL. Par la suite, les termes opération élémentaire et traitement local pourront être employés indifférement pour désigner le même principe.

La localisation d'une opération élémentaire se réalise par un dialogue entre l'interprète et l'UGM-APL.

Une première solution peut être la suivante: Pendant ce dialogue, l'interprète indique à l'UGM-APL les objets qu'il sera amené à accéder pendant l'exécution d'un traitement local donné, en lui passant comme paramètres leurs références universelles respectives.

En réponse, l'interprète obtient de l'UGM-APL une "désignation locale" par objet, qui est beaucoup plus efficace à utiliser (on peut concevoir une désignation locale comme étant un numéro d'entrée dans une table peu volumineuse).

Au terme de ce dialogue, l'interprète peut commencer l'exécution de son traitement local en accédant les éléments des objets au moyen des désignations locales. Finalement, l'interprète signale la fin d'un traitement local à l'UGM-APL pour permettre la libération des désignations locales.

Une deuxième solution consiste à passer, en même temps que la référence universelle, la valeur de la désignation locale avec laquelle l'interprète désignera l'objet tout au long d'un traitement local. Avec cette méthode, ce n'est plus l'UGM-APL qui choisit les désignations locales, mais l'interprète.

Cette deuxième solution est plus facile à mettre en oeuvre. En effet, avec cette méthode, d'une part l'interprète reste le seul maître à choisir ses propres valeurs de désignations, et d'autre part la conception de l'UGM-APL est grandement facilitée par le fait que les différents contrôles sur les correspondances sont éliminés.

On arrive ainsi à établir un schéma très général d'algorithme en ce qui concerne l'exécution d'un traitement local :

EXECUTION D'UN TRAITEMENT LOCAL

1) Etablir les correspondances :
référence universelle <----> désignation locale

2) Exécuter le traitement local sur les objets
liés avec une correspondance définie en (1)

3) Libérer les correspondances :
référence universelle <----> désignation locale

3.5 Les structures des données logiques et les algorithmes de fonctionnement

Nous allons à présent détailler les trois parties qui composent le schéma général d'algorithme de l'exécution d'un traitement local. Pour chacune de ces parties, nous présenterons les structures de données logiques nécessaires et les algorithmes généraux qui les manipulent.

3.5.1 Etablissement des correspondances : référence universelle <----> désignation locale

Pendant cette première étape de l'exécution d'un traitement local, l'interprète utilise l'UGM-APL en vue de lui faire connaître les correspondances entre référence universelle et désignation locale.

Detaillons maintenant les actions successives qui doivent être entreprises pour la réalisation de cette première phase :

- Il faut tout d'abord que l'interprète établisse

une forme de dialogue avec l'UGM-APL. Pour celà, il doit mettre l'UGM-APL dans l'état actif (l'état inactif est le plus fréquent).

- Lorsque l'UGM-APL est activée, l'interprète peut lui indiquer l'ensemble des références universelles et des désignations locales correspondantes.

- L'UGM-APL se contente d'enregistrer les correspondances pour pouvoir les utiliser lors de l'exécution d'un traitement local.

- Enfin, l'UGM-APL passe à nouveau dans son état inactif habituel.

Essentiellement, deux structures de données sont nécessaires pour la réalisation de cette première étape.

La première structure sera utilisée pour communiquer les différents paramètres entre l'interprète et l'UGM-APL. On peut employer pour celà une structure de tableau à n entrées et accessible aussi bien par l'interprète que par l'UGM-APL. Chaque entrée de ce tableau pourra contenir une référence universelle, la désignation locale correspondante ou encore un message d'erreur. Les opérations autorisées sur cette structure (appelons-la structure de communication : SC), seront les suivantes :

i) De la part du processeur :

- Ecrire une référence universelle rui, la longueur L et sa désignation locale correspondante dli à l'entrée no i de SC : écrire_SCi(rui,L,dli). Le paramètre L indique la longueur de l'objet en multiples d'octets.

- Invalider toutes les informations de SC : clear_SC

ii) De la part de l'UGM-APL :

- Lire une référence universelle rui, la longueur L et sa désignation locale correspondante dli depuis l'entrée no i de SC : lire_SCi(rui,L,dli)

- Eventuellement, l'UGM-APL pourra deposer un message d'erreur ou de compte rendu dans une entrée no i de SC, si elle se trouve dans l'impossibilité de réaliser la correspondance i ou si cette dernière a déjà été établie : écrire_SCi(mess_i)

La deuxième structure de donnée est interne à l'UGM-APL et elle contient les correspondances d'un traitement local, au fur et à mesure qu'elles s'établissent.

Nous verrons que, pendant l'exécution du traitement local (deuxième phase), cette deuxième structure sera utilisée pour effectuer différents contrôles globaux sur les objets (à partir des attributs de leurs

références universelles).

Enfin, précisons quelques details sur l'activation de l'UGM-APL. Pour notre étude théorique du fonctionnement général de l'UGM-APL, nous supposerons que :

- L'activation de l'UGM-APL survient uniquement par une instruction spéciale de l'interprète : activer(mode). Les différents modes d'activation sont :

* "établissement des correspondances"

* "accès à un élément d'un objet"

* "libération des correspondances"

* "contrôle de l'UGM-APL"

Nous présentrons, chaque fois qu'il sera nécessaire, la signification de chaque mode.

- Le passage d'un état actif à l'état inactif est effectué par l'UGM-APL elle-même et survient exactement à la fin du travail pour lequel l'UGM-APL a été activée. Par la suite, ce passage à l'état inactif sera matérialisé par une instruction spéciale "désactiver", et perçue par l'interprète à l'aide de l'instruction "attendre(désactivation)".

```
   === ALGORITHME 1A /* interprète */


DEBUT
   clear_SC

      POUR toute rui FAIRE
           écrire_SCi(rui , L , dli)
      FINFAIRE

   activer(mode "établissement de correspondances")

   attendre(désactivation)  /* voir remarque */

FIN
```

== Remarque : Dans une implémentation matérielle de l'UGM-APL l'instruction "attendre(désactivation)" peut ne pas être nécessaire, vu que l'UGM-APL doit avoir accès aux différents bus du système (cf. chapitre suivant), et donc avoir la possibilité de bloquer tous les accès jusqu'à ce qu'elle soit dans un état inactif valide.

```
=== ALGORITHME 1B /* UGM-APL */ ===

DEBUT

    POUR toute entrée i FAIRE
        lire_SCi(rui , L , dli)
          /* au fur et à mesure de
          chaque lecture, elle met à
          jour sa structure interne */

    FINFAIRE
  désactiver

FIN
```

== (1) Remarque : L' attribution d'une dl à une ru est très dépendante d'une implémentation. Nous verrons au chapitre suivant une telle implémentation.

3.5.2 Exécution d'un traitement local

C'est dans cette deuxième partie que l'interprète utilise les services de l'UGM-APL afin d'accèder aux éléments des objets qu'il veut manipuler.

La vision de l'UGM-APL sur les objets et la mémoire centrale constitue le coeur de cette deuxième partie :

- La vision des objets : l'UGM-APL "voit" les objets de la même façon que l'interprète. A tout instant elle les connait par leur seul moyen de désignation, la référence universelle, et plus précisement par la partie qui contient leur adresse virtuelle (adresse en mémoire secondaire de la première page de l'objet).

- La vision de la mémoire : l'UGM-APL possède une vision assez restreinte de la mémoire, par rapport à l'interprète. En effet, elle ne peut exercer son contrôle que sur les objets utilisateur, tandis qu'elle ignore complétement la topographie des programmes système, c'est à dire ceux qui réalisent l'interprète. Pour elle, un objet est un segment paginé, les pages ayant une taille fixe et relativement faible. Comme tout accès à un objet se traduit par l'accès à un élément précis de ce dernier, l'UGM-APL doit connaitre une information supplémentaire concernant l'indexation relative par rapport au début de l'objet.

On voit donc qu'ici apparait le problème classique de la traduction d'une adresse virtuelle en adresse physique en mémoire centrale d'un élément d'un objet. Pour accomplir cette traduction, l'UGM-APL utilise l'adresse virtuelle du début de l'objet et le déplacement relatif par rapport à cette adresse qui lui est donné par l'interprète au moment de l'adressage de l'objet. Pour arriver à l'adresse physique de l'élément désiré, l'UGM-APL effectue deux opérations successives : d'abord elle calcule l'adresse virtuelle de l'élément de l'objet et ensuite elle tente de faire correspondre cette adresse à une adresse physique d'implantation en mémoire centrale. En cas de réussite, l'accès à l'élément est automatique. Par contre, s'il n'est pas possible de trouver l'adresse physique correspondante, alors elle positionne une interruption de défaut de page.

Résumons à présent les différentes actions entreprises par l'interprète et l'UGM-APL pour la réalisation d'un accès :

- L'interprète active l'UGM-APL en mode "accès à un élément d'un objet" et lui passe comme paramètres la désignation locale de l'objet, le mode d'accès (lecture ou écriture) et une valeur de déplacement relatif.

- L'UGM-APL commence son action en effectuant certains contrôles globaux sur l'objet. Pour celà, elle analyse différentes informations contenues en majorité

dans la référence universelle, comme par exemple les droits d'accès de l'objet (accessible en lecture et/ou en écriture) ou encore sa longueur (verification que l'adressage effectué par l'interprète reste dans les limites de l'objet).

- Si à l'issue de ces contrôles globaux aucune anomalie n'est détectée, l'UGM-APL procède au calcul de l'adresse virtuelle de l'élément. Les différentes méthodes qui existent dépendent étroitement d'une implémentation donnée. Nous présentons une telle méthode dans le chapitre suivant.

- L'UGM-APL poursuit son action en essayant de faire correspondre l'adresse virtuelle de l'élément à une adresse physique en mémoire centrale. Dans l'affirmative, elle adresse automatiquement l'élément, suivant le mode indiqué par l'interprète (en lecture ou en écriture), sinon elle signale un défaut de page. A la fin de cette action, elle se met dans l'état inactif.

- En cas de défaut de page, l'interprète met en oeuvre un mécanisme de chargement (si dans la mémoire restent des pages inutilisées) ou de remplacement de page (cas ou il faut évacuer une page de la mémoire centrale pour faire de la place). A la suite de cette opération, l'interprète adresse à nouveau sa requette d'accès à l'UGM-APL.

Les structures nécessaires pour la réalisation de l'accès à un élément d'un objet à travers l'UGM-APL sont les suivantes :

- Comme pour la phase précédente, une structure (SC) entre l'interprète et l'UGM-APL est nécessaire pour la communication des différents paramètres. Néanmoins, dans ce cas elle est plus simple à cause de la nature des paramètres. De la part de l'interprète, les différentes opérations sur la structure SC sont les suivantes :

== écrire_SCd(dl) : écrire dans le champ d de SC la désignation locale de l'objet à accéder;

== écrire_SCp(depl) : écrire dans le champ p de SC le déplacement relatif par rapport au début de l'objet;

== écrire_SCm(mode) : spécifier le mode d'accès (lecture ou écriture);

== écrire_SCv(valeur) : s'il s'agit d'une écriture, alors l'interprète donne la valeur de mise à jour;

== lire_SCv(valeur) : récupération de la valeur de l'élément, s'il s'agit d'une lecture;

== lire_SC(compte_rendu) : en cas de défaut de page ou de violation des accès à l'objet (interruption de la part de l'UGM-APL), l'interprète récupère un compte rendu qui indique :

- soit qu'il existe de pages non allouées

(disponibles) en mémoire centrale,

- soit une proposition de no de page à évaqcuer (par exemple la plus anciènement accédée).

Réciproquement, les opérations de l'UGM-APL sur la structure SC sont les suivantes :

== lire_SCd(dl)

== lire_SCp(depl)

== lire_SCm(mode)

== lire_SCv(valeur)

== écrire_SCv(valeur)

== écrire_SC(compte_rendu)

Ces instructions correspondent à la récupération des paramètres nécessaires à l'accès à un élément. De plus, les deux opérations d'écriture seront alternativement exécutées en cas de consultation de l'élément ou lorsque l'UGM-APL détecte une erreur de contrôle ou d'adressage.

- La deuxième structure est interne à l'UGM-APL et contient, entre autres informations, toutes les

correspondances entre références universelles et désinations locales des différents objets qui sont accédés pendant l'exécution d'un traitement local. L'UGM-APL peut effectuer sur cette structure (appellons-la SD) les opérations suivantes :

== adr_virtuelleSD(dl) : récupérer l'adresse virtuelle de la première page de l'objet (information contenue dans sa référence universelle), dont la désignation locale est dl.

== contrôleSD(objet,depl) : cette fonction réalise les différents contrôles d'accès globaux sur l'objet. Plus précisement :

- elle consulte les droits d'accès;

- elle vérifie que le déplacement proposé par l'interprète reste bien dans les limites de l'objet.

- Un troisième élément nécessaire à cette phase est un mécanisme de calcul d'adresses. La récupération de l'adresse physique de l'élément à accéder comporte deux parties distinctes :

- Dans un premier temps, à partir de l'adresse virtuelle de la première page de l'objet et le déplacement indiqué par l'interprète, l'UGM-APL calcule l'adresse virtuelle de la page où se trouve l'élément désiré.

- Ensuite, à l'aide d'une structure topographique faisant correspondre à chaque adresse virtuelle de page l'adresse d'implantation en mémoire centrale, l'UGM-APL récupère l'adresse physique de la page, si toutefois cette dernière existe en mémoire centrale.

La structure des correspondances entre adresses virtuelles et adresses physiques des pages, contient aussi les informations de contrôle d'accès au niveau de la page (par exemple interdiction de modification, page modifiée/non modifiée, compteur d'accès à la page).

Le calcul d'adresse virtuelle de la page ou se trouve l'élément (P_VIR) peut être représenté par l'opération suivante :

P_VIR := calcul_adr(adr_virtuelleSD(dl),depl)

Enfin, les opérations sur la structure topographique des correspondances des pages (ST) sont les suivantes :

== P_PHYS := adr_physST(P_VIR) : à la suite de l'exécution de cette opération, dans P_PHYS on peut trouver :

- soit l'adresse d'implantation de la page dont l'adresse virtuelle est spécifiée dans P_VIR ;

- soit une information qui ne correspond pas à une adresse physique (admettons par convention la valeur -1), auquel cas il s'agit d'un défaut de page.

== contrôleST(P_VIR) : cette opération a pour but de respecter les différents contrôles au niveau de la page. On peut par exemple convenir que cette opération donne comme résultat une valeur logique :

Ø : l'accès demandé peut avoir lieu parce qu'il respecte les conditions prédéfinies de la page ;

1 : l'accès demandé est incompatible avec les conditions prédéfinies de la page et l'UGM-APL envoie à l'interprète un message d'erreur tout en arretant la procédure d'accès.

```
   ===== ALGORITHME 2A /* interprète */


PROCEDURE accès (dl , depl , mode)

DEBUT

  clear_SC
  écrire_SCd(dl)
  écrire-SCp(depl)
  écrire_SCm(mode)

    SI mode = écriture ALORS
       écrire_SCv(valeur)
    FINSI

  activer(mode:"accès à un élément d'un objet")
  attendre(désactivation)
  CMPT_R := lire_SC(compte_rendu)

    SI CMPT_R non-vide ALORS
       gestion_page(CMPT_R) /*défaut de page */
       accès (dl , depl , mode)

     SINON SI mode = lecture ALORS
              lire_SCv(valeur)
           FINSI
    FINSI

FIN
FIN  /* accès */
```

```
   ===== ALGORITHME 2B /* UGM-APL */

PROCEDURE accès (dl , depl , mode)

DEBUT

  lire_SCd(dl)
  lire_SCp(depl)
  lire_SCm(mode)

    SI mode = écriture ALORS
       VAL := lire_SCv(valeur)
    FINSI

    SI contrôleSD(objet , depl) positif ALORS
       écrire_SC(compte_rendu : "mauvais contrôle
                                 global sur l'objet")
       désactiver  /* arret de la procédure */
    FINSI

  P_VIR := calcul_adr(adr_virtuelleSD(dl) , depl)

    SI contrôleST(P_VIR) =1 ALORS
       écrire_SC(compte_rendu : "incompatibilité
                 d'accès au niveau de la page")
       désactiver  /* arret de la procédure */
    FINSI

  P_PHYS := adr_physST(P_VIR)

    SI P_PHYS = 1 ALORS
       écrire_SC(compte_rendu : "défaut de page")
```

```
      désactiver  /*arret de la procédure */
    FINSI

    SI mode = écriture ALORS
       adresser_Ecr(P_PHYS , VAL)  /* écriture */
      SINON  /* lecture */
         VAL := adresser_Lec(P_PHYS)
         écrire_SCv(VAL)
    FINSI

  désactiver

FIN
FIN  /* accès */
```

** Remarque : Les opérations adresser_Ecr et adresser_Lec réalisent l'accès effectif à l'élément de l'objet, respectivement en écriture ou en lecture.

Tout au long de cette analyse concernant l'accès à un élément d'un objet, on a supposé que l'UGM-APL avait accès à une structure topographique contenant l'ensemble des pages présentes en mémoire centrale ainsi que l'adresse exacte de leur implantation. Cette structure correspond bien à la vision "restreinte" de l'UGM-APL par rapport à celle de l'interprète, telle qu'elle a été présentée au début du paragraphe 3.5.2. En plus, dans le contexte des postes autonomes, deux raisons essentielles ne permettent pas l'emploi des tables regroupant toutes les pages de l'espace des objets :

- Une des caractéristiques fondamentales d'un environnement APL est l'existence d'un espace unique d'objets très grand. Pour un tel espace, une table qui regroupe toutes ses pages est forcement assez importante et lente à exploiter.

- L'espace virtuel que l'on veut gérer est unique, et de ce fait il peut y avoir une correspondance directe avec l'espace physique sans nécessiter de tables de conversion.

3.5.3 La libération des correspondances

C'est l'action qui termine normalement un traitement local. Pendant cette phase, l'interprète indique à l'UGM-APL toutes les désignations locales qui ont servi tout au long du traitement pour accéder aux différents objets. Ceci permet à l'UGM-APL de liberer des désignations (et aussi des entrées dans sa table de désignations) pour pouvoir les utiliser ultérieurement.

La libération des correspondances peut se faire de plusieurs façons. Le choix d'une ou d'une autre technique est très dépendant d'une implémentation donnée (nous verrons un tel choix au prochain chapitre). Néanmoins, en voici deux :

- L'interprète procède comme pour l'établissement des correspondances : il envoie à l'UGM-APL les désignations soit une par une, soit globalement, et pour chacune il demande leur libération (invalidation des références universelles correspondantes).

- L'interprète envoie une commande spéciale dont l'effet est la libération de toutes les correspondances déjà établies.

3.6 Les différents contrôles et commandes

Jusqu'à maintenant, nous avons analysé le comportement de l'interprète et de l'UGM-APL pendant les trois phases d'un traitement local (établissement de correspondances, accès à un élément d'un objet, libération). Or, comme l'UGM-APL est autonome pendant son fonctionnement, il est nécessaire que l'interprète puisse apporter à des instants précis, des modifications aussi bien dans le comportement, qu'aux informations qu'elle connait. C'est là le rôle des différents contrôles, réalisés sous forme de commandes par l'interprète.

A ce niveau de notre étude, nous allons simplement énumérer les différents contrôles "utiles" ou "nécessaires". Nous verrons au chapitre suivant comment ils peuvent être réalisés dans un contexte logiciel et matériel bien précis.

De manière générale on peut classer les différents contrôles en deux catégories :

i) La manipulation et la mise à jour globale d'éléments de l'UGM-APL.

On peut regrouper dans cette catégorie deux types de commandes :

- celles qui facilitent la mise en oeuvre d'un contexte multi-processus (et éventuellement multi-processeurs). Les commandes qui permettent le chargement et la sauvegarde globale de la structure des correspondances des désignations locales, seront très efficaces pour réaliser des changements de contextes.

- celles qui initialisent l'UGM-APL. Parmi les informations nécessaires de telles commandes on peut citer la taille des pages et des sous-pages, les adresses des zones mémoire qui serviront à la communication des données entre l'UGM-APL et l'interprète ou encore son mode de fonctionnement.

ii) La manipulation et la mise à jour ponctuelle d'éléments de l'UGM-APL

Il est parfois nécessaire que l'interprète puisse avoir accès (en consultation et modification) soit aux informations concernant un objet, soit à celles concernant une page :

- Au niveau de l'objet, l'interprète devra pouvoir modifier ses droits d'accès ou encore sa longueur

maximale (si cette dernière n'est pas précisée lors de l'établissement des correspondances, alors l'UGM-APL pour des raisons de sécurité propose la longueur minimum autorisée).

- Au niveau de la page, il est nécessaire d'avoir accès aux informations suivantes :

* les droits d'accès à la page;

* la possibilité de maintenir en permanence une page en mémoire centrale.

== REMARQUE : La bibliographie de ce troisième chapitre est regroupée avec celle du chapitre 4, et figure à la fin de ce dernier.

CHAPITRE 4 Conception d'une UGM-APL pour un poste de travail autonome scientifique

Dans ce dernier chapitre, nous essayons de mettre en évidence les aspects fondamentaux d'une unité de gestion mémoire orientée APL. Pour celà, nous développons en details la conception d'une telle unité, réalisable presque entièrement en matériel.

4.1 Généralités

Cette dernière partie de notre étude sera consacrée à la conception d'un processeur matériel spécialisé (UGM-APL), qui assure la gestion des pages et les accès aux objets au sein d'un poste de travail autonome scientifique orienté APL. L'étude de cette conception, tout au long de cette partie, sera axée sur le développement logique de fonctionnement de l'UGM-APL, tel qu'il a été présenté au chapitre précédent.

Mais avant de rentrer dans les details de cette étude, présentons l'environnement immédiat de l'UGM-APL, ainsi que quelques choix qui ont guidé sa conception.

4.1 Le contexte d'implémentation de l'UGM-APL

Comme on l'a vu sur quelques exemples présentés au premier chapitre, la plupart des postes de travail autonomes sont construits autour d'un microprocesseur général ou microprogrammé de 16 ou 32 bits, avec une mémoire centrale dont la taille peut atteindre quelques Méga-octets, et une mémoire secondaire (disques) de

quelques dizaines, voir quelques centaines de Méga-octets. Partant de cette constatation, nous sommes amenés de respecter les deux points suivants :

- Il faut que l'UGM-APL puisse s'intégrer à un coût relativement faible dans la plupart des systèmes mono-microprocesseurs 16 et/ou 32 bits évolués;

- En tenant compte des évolutions technologiques, il faut que le degré de complexité de l'architecture de cette unité soit tel qu'après une première réalisation effective au moyen de composants discrets, on puisse la ramener, à court terme, dans un seul circuit intégré spécialisé.

Notons aussi une hypothèse de départ qui demeurera vraie pour toute la suite de ce chapitre : lorsque le processeur central voudra bénéficier des services de l'UGM-APL, il sera obligé d'établir avec elle une forme de dialogue. En dehors de ce dialogue, l'UGM-APL reste toujours dans un état "inactif" (n'affectant pas le comportement général du système). Ainsi, cette caractéristique permettra au processeur central d'adresser les instructions de l'interprète sans aucune pénalisation.

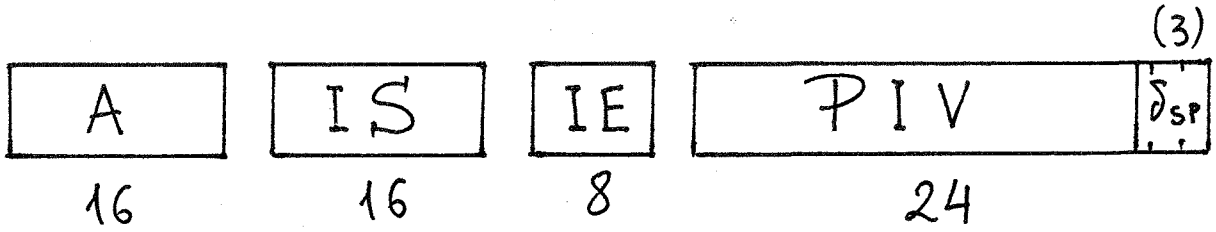
Précisons enfin que la forme de présentation de cette conception sera la suivante : nous analyserons successivement les trois phases qui réalisent un

traitement local (établissement des correspondances, accès à un élément, libération des correspondances). Pour chacune de ces phases nous présenterons les éléments de l'UGM-APL nécessaires, ainsi que leur utilisation. A la fin de cette analyse, nous étudierons les différents contrôles que le processeur central devra exercer sur l'UGM-APL, pour assurer la cohérence de fonctionnement de la totalité du système.

4.1.2 Définition de la référence universelle

La première notion qui intervient tout au long de la conception de l'UGM-APL est celle de la référence universelle. Nous avons déjà précisé la nature des informations contenues dans la référence universelle d'un objet. Ici nous allons la spécifier très précisément.

La référence universelle d'un objet est représentée par une file de 64 bits. A l'intérieur de cette file il existe un découpage logique en quatre champs :



Signification de chaque groupe :

A : Attributs (caractéristiques) propres à l'objet

IS : Identification de site

IE : Identification d'espace dans un site

PIV : Numéro de page initiale virtuelle de l'objet

dsp : Numéro de sous-page dans la PIV (*)

(*) : Pour les objets monopages qui ne commencent pas en frontière de page, dsp $\neq \emptyset$.

Un tel découpage logique des informations d'une référence universelle permet d'évaluer exactement les limites imposées aussi bien à l'espace virtuel qu'aux objets eux-mêmes :

- Le champ PIV (24 bits) permet l'accès à 2**21 pages virtuelles différentes, à l'intérieur d'un espace (taille maximale) qui, lui-même, se trouve sur un site donné.

- Le codage de l'identification de l'espace d'un site donné, sur 8 bits, permet d'avoir jusqu'à 256 espaces différents par site.

- On peut accéder jusqu'à 65536 sites compatibles différents, du fait que l'identification de site est représentée sur 16 bits.

- En ce qui concerne la limitation des objets, on verra que, lors de l'adressage à l'intérieur d'un objet, et pour un processeur avec un bus d'adresses de 24 bits, les 14 bits de poids forts servent à accéder la page où se trouve l'élément (longueur maximale d'un objet : 2**14 pages) et les 10 bits restants adressent un octet à l'intérieur de la page. Ce mécanisme donc impose notre choix d'une longueur de page de 1k octets (unité de transfert), tandis que la taille d'une sous-page (unité d'allocation) est fixée à 128 octets (dsp = 3 bits).

Par la suite, pour tout dialogue avec l'UGM-APL, l'interprète utilisera seulement l'adresse universelle d'un objet, qui est une partie de la référence universelle et définie par l'ensemble des informations IS+IE+PIV+dsp.

4.1.3 Séquencement des instructions du processeur

Pendant toutes les phases de conception qui vont être détaillées par la suite, les actions de l'UGM-APL effectuées entre deux instructions du processeur central risquent de poser un problème de temps au niveau du séquencement des instructions de ce dernier. Si tel est le cas, alors l'UGM-APL doit d'une part

pouvoir suspendre les activités du processeur pendant le temps nécessaire à l'accomplissement de son travail, et d'autre part, dans certains cas, générer une interruption d'un certain niveau. Ici, il faut noter le caractère volontairement universel de l'UGM-APL, c'est à dire le fait qu'elle pourra être connectée à faible coût à la plupart des systèmes actuels. Une solution qui nous parait satisfaisante, du fait de son universalité, est l'application du principe des interruptions vectorisées. Un processeur qui traite ce type d'interruptions, procède de la façon suivante : Lorsqu'il detecte la présence d'une interruption, il lit la valeur d'une case mémoire spécifique. Cette valeur, préalablement chargée par le processus qui a créé cette interruption, indique justement sa nature. Ainsi le processeur peut activer le gérant d'interruption correspondant. Pour la réalisation d'un dispositif d'interruptions vectorisées, un signal et un registre pouvant être accédé par le processeur, sont suffisants. Le signal indique la présence d'une interruption, tandis que dans le registre est chargée une valeur indiquant la nature de cette interruption.

Une autre solution consiste à réaliser, au niveau de l'UGM-APL, un système hiérarchisé d'interruptions. Pour celà, il suffit de disposer à la sortie de l'UGM-APL un petit nombre n (3 ou 4) de signaux. Une combinaison logique de ces signaux correspondant à un niveau précis d'interruption, il suffira alors d'un décodeur de fonctions de type n--->2**n pour résoudre le problème.

4.2 Etablissement des correspondances : référence universelle <----> désignation locale

Comme on l'a vu au chapitre précédent, le but d'un établissement de correspondances au sein d'un traitement local est de faire correspondre une désignation universelle d'objet (référence universelle) à une désignation locale, beaucoup plus efficace à utiliser. Ceci est possible puisque, lors de l'exécution d'un traitement local, un nombre restreint d'objets doit être accessible très rapidement.

Lorsque l'interprète veut établir une correspondance, il doit utiliser tout d'abord un registre spécial de l'UGM-APL, appellé Registre Adresse Universelle (RAU). La taille de ce registre est de 10 octets, mais pour cette première phase, l'interprète sera amené à n'utiliser que les 68 bits de poids faibles.

L'interprète adresse ce registre exactement comme une cellule mémoire, et charge dans celui-ci, soit par octet, soit par mots de 16 ou 32 bits (si son jeu d'instructions le permet), la partie de la référence universelle correspondant à l'adresse universelle pour laquelle il veut établir une correspondance. Ensuite, il utilise un double registre de 16 bits. L'octet de poids faibles est appellé Registre de Désignation Locale (RDL) et sert à charger la valeur de la désignation locale de l'objet dont l'adresse universelle figure dans RAU. L'octet restant, appellé

Registre Des Commandes (RDC) sert à activer l'UGM-APL, son contenu étant interprété comme une commande. La structure détaillée de ce registre RDC sera donée ultérieurement dans ce chapitre, mais notons simplement qu'un bit de ce registre, lorsqu'il est positionné, permet de sélectionner une structure interne de l'UGM-APL, nécessaire pour cette première phase (table TDL).

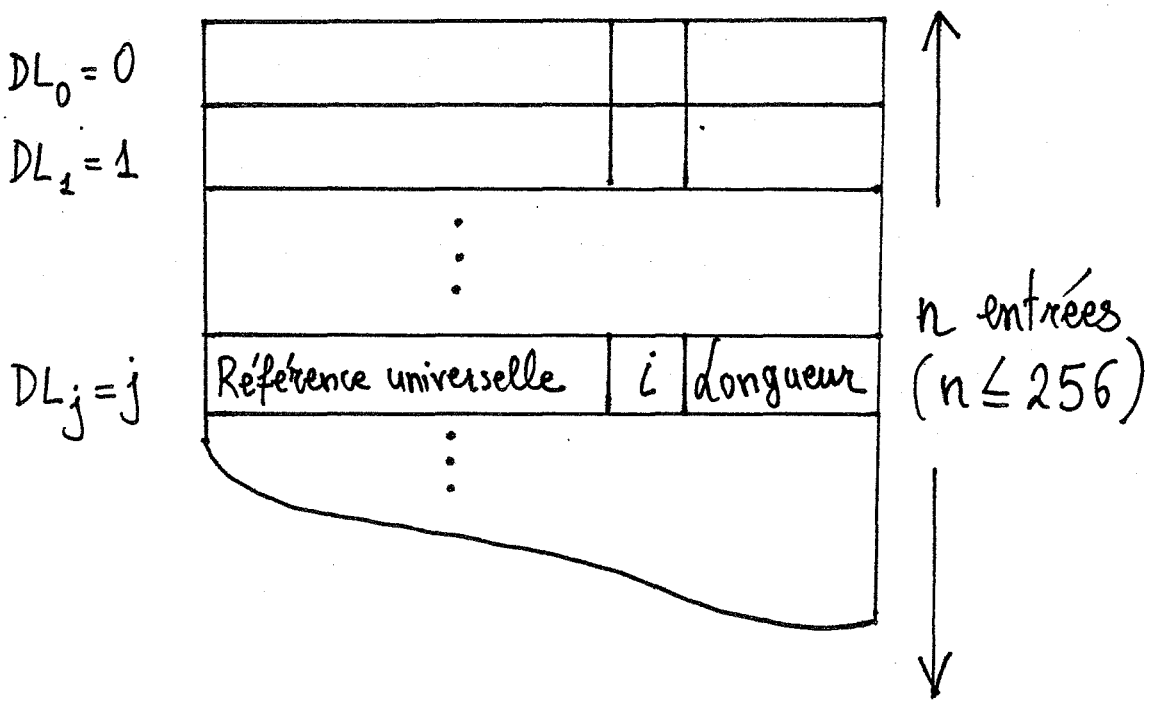
Les deux opérations de la part de l'interprète pour l'établissement d'une correspondance, sont les suivantes :

i) Ecrire l'adresse universelle, les bits de contrôle et la longueur de l'objet dans le registre RAU.

ii) Ecrire une valeur spécifique dans le registre RDC et le numéro de la désignation locale dl de l'objet dans le registre RDL. Cette deuxième opération doit se faire avec un chargement unique sur 16 bits (RDC et RDL). Elle a pour effet l'activation de l'UGM-APL en mode "établissement des correspondances".

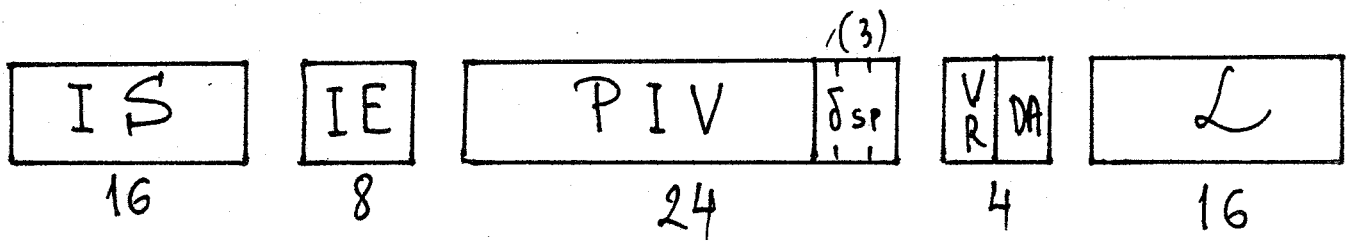
Ainsi activée, l'UGM-APL doit sauvegarder cette correspondance pour être capable de l'utiliser lors d'un accès à l'objet. Pour celà, elle utilise une structure matérielle interne, qui est un ensemble de registres structurés sous forme de table. L'entrée i de cette table correspond aux informations de l'objet de désignation locale i.

Le schéma général de cette Table des Désignations Locales (TDL) est le suivant :



La structure d'une entrée de cette table est la suivante :

Description detaillée d'une entrée
dans la table des désignations locales (TDL)



IS : Identification de site

IE : Identification d'espace dans un site

PIV : Numéro de page initiale virtuelle d'un objet

δsp : Numéro de sous-page dans la page PIV

VRDA : Indicateurs sur :
- entrée valide / non-valide
- droits d'accès à l'objet

$\mathcal{L}$: longueur en multiples d'octets d'un objet

$IS + IE + PIV + \delta sp$: adresse universelle de l'objet.

La sauvegarde de la part de l'UGM-APL de la correspondance demandée par l'interprète consiste tout simplement à mettre à jour l'entrée no dl (valeur dans RDL) de la table TDL, avec les informations figurant dans les 68 bits de poids faibles du registre RAU. Il est évident que c'est à la charge de l'interprète de gérer cette mise à jour, afin d'éviter la perte des correspondances qui, éventuellement, seraient utiles.

Processus de fonctionnement de l'UGM-APL pendant l'établissement d'une correspondance

- Activation de l'UGM-APL /* le processeur écrit dans dans les registres RDC,RDL*/

- Arrêt du processeur /* l'UGM-APL agit sur le signal HALT du processeur */

- entreeTDL(dl) := (RAU)

- Suspendre l'arrêt du processeur /* l'UGM-APL agit à nouveau sur HALT */

- Désactivation de l'UGM-APL

4.3 Accès aux éléments d'un objet

Cette fonction de l'UGM-APL constitue la partie fondamentale de l'exécution d'un traitement local. Pour qu'elle puisse se réaliser, il faut nécessairement que les correspondances des objets, dont l'interprète veut accéder les éléments, soient préalablement établies.

Comme pour la partie précédente concernant l'établissement des correspondances, nous allons d'ecrire successivement les actions entreprises par le processeur et l'UGM-APL, les éléments matériels nécessaires et enfin les différents algorithmes qui réalisent l'accès à un élément d'un objet.

Le principe de fonctionnement de cette partie repose sur un certain nombre de caractéristiques :

- L'espace des objets est paginé. Chaque page (unité de transfert) est divisée en un nombre fixe de sous-pages (unités d'allocation).

- Le processeur central doit supporter la gestion de défaut de page. Autrement dit, il est nécessaire que l'UGM-APL puisse interrompre le processeur au milieu d'un cycle d'adressage avec la possibilité de conserver

un contexte cohérent pour que l'instruction interrompue puisse reprendre après l'exécution du gérant de gestion de page.

- Le processeur, lorsqu'il veut adresser un élément d'un objet, le désigne toujours par son adresse virtuelle (adresse virtuelle de page et déplacement dans celle-ci).

Puisque l'une des caractéristiques principales de l'UGM-APL est de rester dans l'état inactif tant qu'un travail spécifique n'est pas demandé, l'interprète, avant de procéder à l'accès à un élément d'un objet, doit préalablement activer l'UGM-APL en lui précisant en même temps la nature du travail qui sera demandé par la suite. On a vu que lors de l'établissement d'une correspondance, le fait d'écrire une valeur dans les registres RDC et RDL non seulement provoque l'activation de l'UGM-APL, mais aussi précise la nature du service demandé.

4.3.1 Les instructions d'accès

Dans le cas de l'accès à un élément d'un objet, nous allons adopter une philosophie similaire. Néamoins, pour ne pas trop "spécialiser" la réalisation de l'interprète, et aussi pour rester le plus universel possible vis à vis du matériel à utiliser, tout accès à un élément se decomposera en deux instructions :

- activation de l'UGM-APL;

Pour cela, il suffit que le processeur écrit dans RDC la valeur Øxxxxxxx (le bit 7 indique le passage en mode "accès à un élément"), et dans RDL la désignation locale de l'objet qu'il veut accéder.

- instruction "normale" d'accès à un élément.

4.3.1.1 Instruction d'activation

L'instruction d'activation de l'UGM-APL lui procure les informations suivantes :

- Elle est activée pour un travail précis (dans ce cas c'est l'accès à un élément);

- Elle a un accès quasi-immédiat à l'adresse universelle de l'objet qu'elle devra accéder, par le biais de la désignation locale indiquée par l'interprète.

4.3.1.2 Instruction d'accès

Avec la deuxième instruction d'accès à un élément, l'UGM-APL obtient le reste des informations nécessaires à l'accès. Ces informations sont :

- le mode d'accès à l'élément (en lecture ou en écriture);

- l'emplacement exact de l'élément par rapport au début de l'objet (déplacement par rapport à l'adresse virtuelle de la première page de l'objet).

Cette deuxième instruction doit être une instruction de transfert (en lecture ou en écriture) entre le processeur et la mémoire des objets.

Généralement, l'exécution d'une telle instruction peut être décomposée en trois phases successives :

- décodage;

- exécution effective de l'instruction;

- préparation de l'instruction suivante.

Ici, on s'intéressera plus particulièrement à la deuxième phase, c'est à dire à l'exécution effective de l'instruction. Pour une instruction de transfert d'information (lecture ou écriture d'un mot mémoire), l'algorithme général d'exécution est le suivant :

1) Placer l'adresse sur le bus; valider cette adresse avec un signal de contrôle;

2) Emettre le signal du sens de transfert (lecture ou écriture); emettre le signal du format de la donnée à transferer;

3) --- Fonctionnement en écriture :

Envoi de la part de la mémoire ou du périphérique concerné d'un signal de prise en compte de la donnée.

--- Fonctionnement en lecture :

Envoi de la part de la mémoire ou du périphérique concerné d'un signal indiquant la présence de la donnée sur le bus.

4) Fin du cycle de transfert avec éventuellement une invalidation du bus de données.

C'est pendant l'exécution de cette partie de l'instruction de transfert que l'UGM-APL devra intervenir pour récupérer les informations supplémentaires nécessaires d'accès à un élément d'un objet. Ces informations sont :

- la valeur émise sur le bus d'adresses (adresse virtuelle de l'élément);

- le signal indiquant le sens du transfert et celui indiquant la quantité à transférer (1 octet, un mot de 16 ou 32 bits).

En plus, elle devra être capable d'émettre, suivant

les circonstances :

- un signal d'arrêt momentané du processeur (le même que celui envoyé lors de l'établissement d'une correspondance);

- des signaux permettant un blocage temporaire du bus, et un masquage des interruptions.

4.3.2 La traduction des adresses

Avec ces informations, voyons maintenant comment l'UGM-APL procède à la traduction des adresses.

Après son activation par le processeur, et après avoir récupéré les autres informations nécessaires à la traduction, l'UGM-APL envoie un signal d'arrêt au processeur et elle procède à la traduction. On peut décomposer cette dernière en trois parties :

- Contrôle sur l'objet adressé.

- Calcul de la page véritable virtuelle et du déplacement dans cette page.

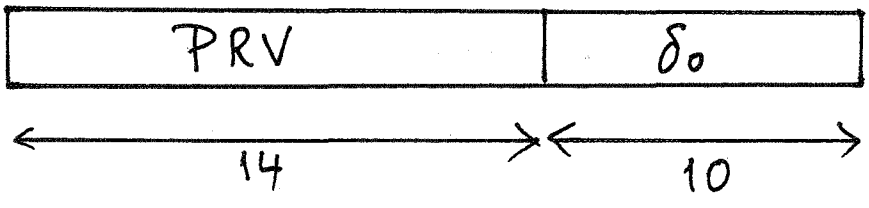
- Recherche dans la table des pages; récupération de l'adresse physique de page.

4.3.2.1 Contrôle sur l'objet adressé.

L'UGM-APL effectue un double contrôle sur l'objet qui sera adressé. Les informations nécessaires à ce niveau sont les suivantes :

- Le signal indiquant le sens du transfert;
- L'adresse universelle de l'objet;
- L'adresse de l'élément émise par le processeur.

L'UGM-APL connait, lors de son activation au moyen des registres RDC et RDL l'adresse universelle de l'objet à accéder. D'autre part, le format de l'adresse de l'élément, émise par le processeur est le suivant (on supposera par la suite que le processeur possède 24 bits d'adresses) :



PRV : Numéro de la page relative virtuelle, par rapport à la page initiale virtuelle de l'objet

figurant dans son adresse universelle.

do : Déplacement en octets dans la page (chaque page fait 1k octets), par rapport au début de PRV.

Le premier niveau de contrôle concerne les droits d'accès de l'objet. Pour celà, l'UGM-APL vérifie la compatibilité entre d'une part les signaux indiquant le sens du transfert et le mode de fonctionnement, et d'autre part les valeurs des bits concernant les droits d'accès de l'objet et qui se trouvent dans l'entrée correspondante de TDL. Si une violation des droits est détectée à la suite de ce contrôle, alors une interruption est positionnée et l'UGM-APL abandonne son travail en se désactivant.

Le deuxième niveau de contrôle (qui peut s'effectuer en parallèle avec le premier) concerne la taille de l'objet adressé. En effet, le champ L de l'entrée dans la table TDL allouée à l'objet, contient, en multiples d'octets, sa taille. Ainsi, l'UGM-APL peut vérifier que l'adresse de l'élément émise par le processeur se trouve bien dans les limites de la taille de l'objet. Si un débordement est détecté à ce niveau, alors une interruption est positionnée et l'UGM-APL suspend son travail.

Les éléments matériels de l'UGM-APL nécessaires aux contrôles de ce niveau, mais aussi à l'accélerarion de la suite du travail, sont les suivants :

- Le Registre d'Informations concernant l'Objet (RIO) contient la copie de l'entrée de la table TDL relative à l'objet adressé. Ce registre a la même longueur qu'une entrée dans TDL (68 bits), il peut être accessible par le processeur en mode maître et les informations qu'il contient restent dans un état cohérent, même après la suspension des activités de l'UGM-APL.

- Le Registre d'Adresse d'un Elément (RAE) contient l'adresse émise par le processeur lors de l'accès à un objet. La longueur de ce registre est de 32 bits et lorsque l'UGM-APL est utilisée avec un bus de 24 bits d'adresses, l'octet de poids forts contient toujours la valeur Ø. Ses autres caractéristiques sont les mêmes que pour le registre RIO.

4.3.2.2 Calcul de la page véritable virtuelle et du déplacement final

Nous abordons ici la méthode de calcul de l'adresse physique d'un élément d'un objet, en connaissant :

- son adresse universelle (dans le registre RIO);

- l'adresse virtuelle d'un élément de l'objet, émise par un processus externe (contenue dans le registre RAE).

Les étapes successives d'une traduction d'adresse sont les suivantes :

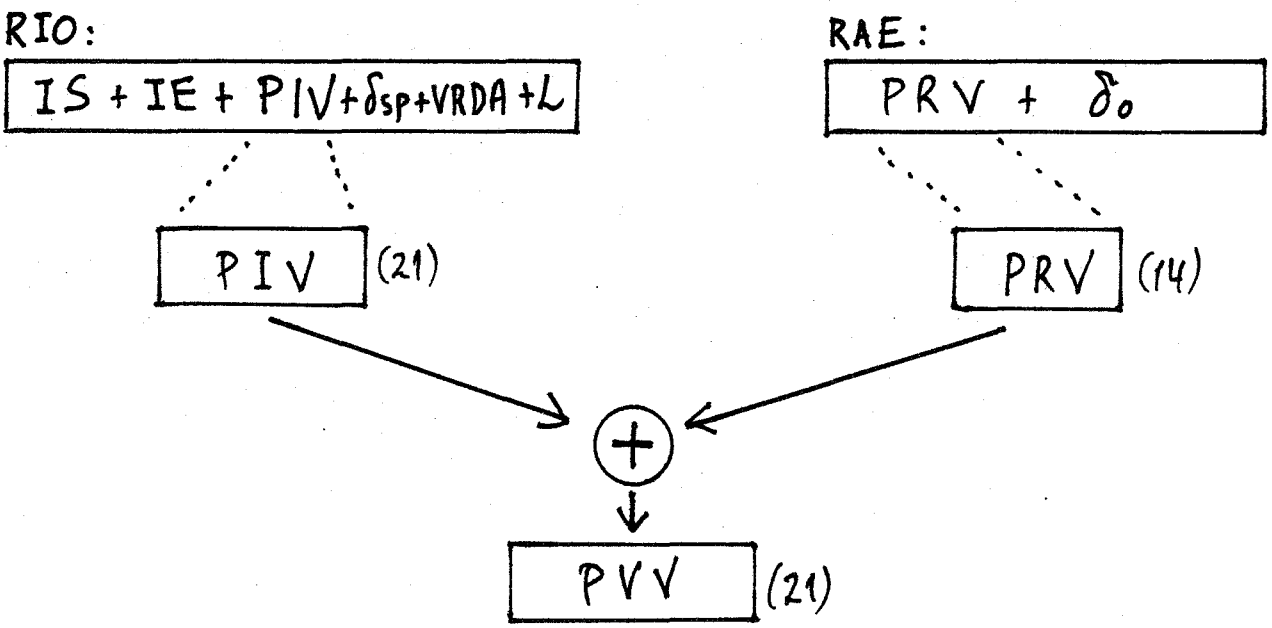
A) Calcul de l'adresse virtuelle de la page ou se trouve l'élément à adresser.

Pour trouver cette adresse, l'UGM-APL additionne les deux informations suivantes :

- l'adresse virtuelle de la page initiale de l'objet, qui tient sur 21 bits et qui est contenue dans l'adresse universelle (champ PIV dans le registre RIO);

- l'adresse virtuelle relative de la page (par rapport à PIV) ou se trouve l'élément à accéder, qui est fournie par le processus externe et contenue dans le registre RAE (champ PRV).

Voici le schéma detaillé de cette opération :



PVV : Page véritable virtuelle de l'élément (adresse virtuelle de la page ou se trouve l'élément à accéder).

Comme on le verra dans la prochaine partie, cette adresse virtuelle de page servira, avec d'autres informations, à rechercher l'adresse physique de la page, si bien sûr celle-ci est implantée en mémoire centrale, et si elle figure parmi les pages les plus récemment référencées.

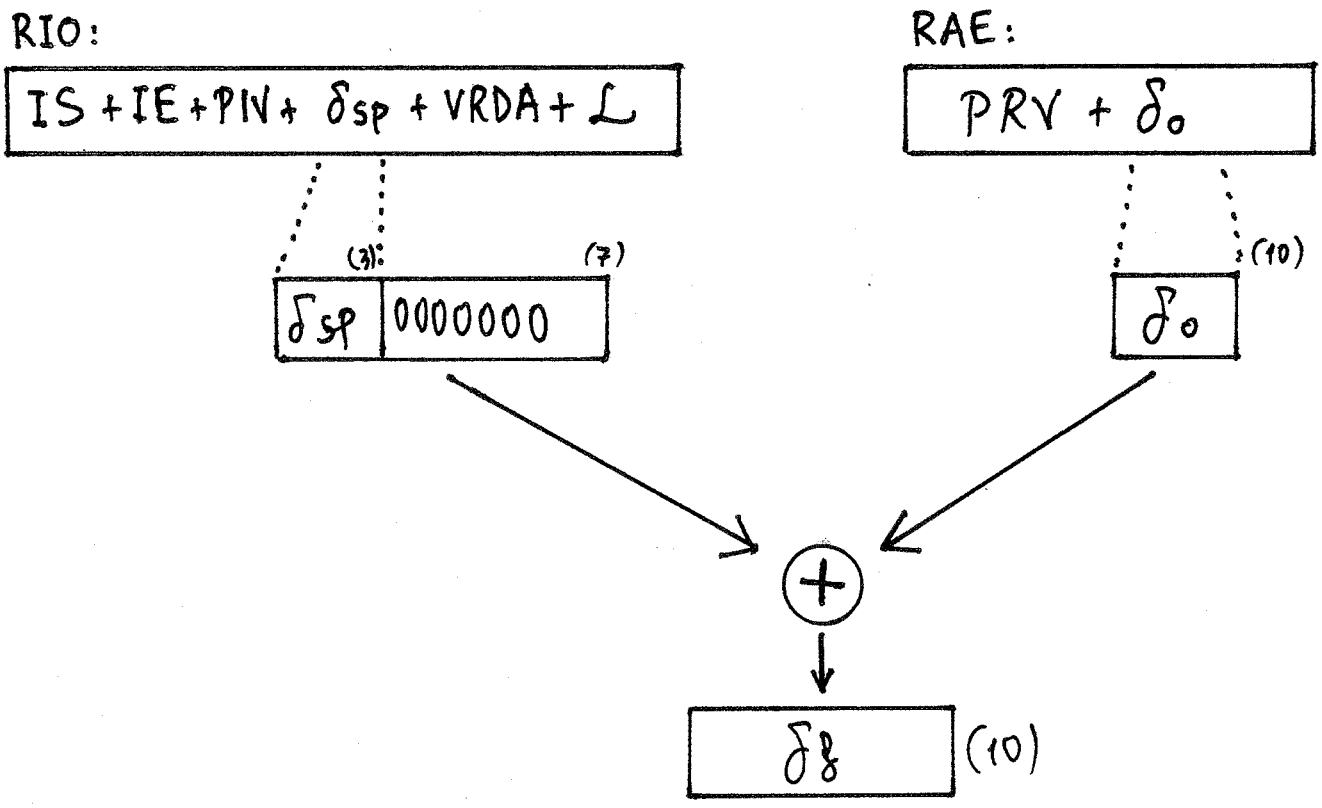
B) Calcul du déplacement final.

L'adresse de l'élément ne sera complète qu'après le calcul du déplacement final par rapport au début de sa page. L'UGM-APL réalise cette opération en parallèle avec celle d'ecrite dans (i). Elle consiste en l'addition des deux informations suivantes :

- le déplacement en sous-pages dsp. Il fait partie de la référence universelle de l'objet (donc dans RIO) et il est non nul uniquement si ce dernier est monopage et ne commence pas en frontière de page.

- le déplacement en octets dans la page do. Il est matérialis'par les 10 bits de poids faibles dans le registre RAE. Il faut noter que les 3 bits de poids forts de ce déplacement indiquent le numéro de sous-page ou se trouve l'élément, et ceci par rapport à la sous-page initiale de l'objet, indiquée par dsp.

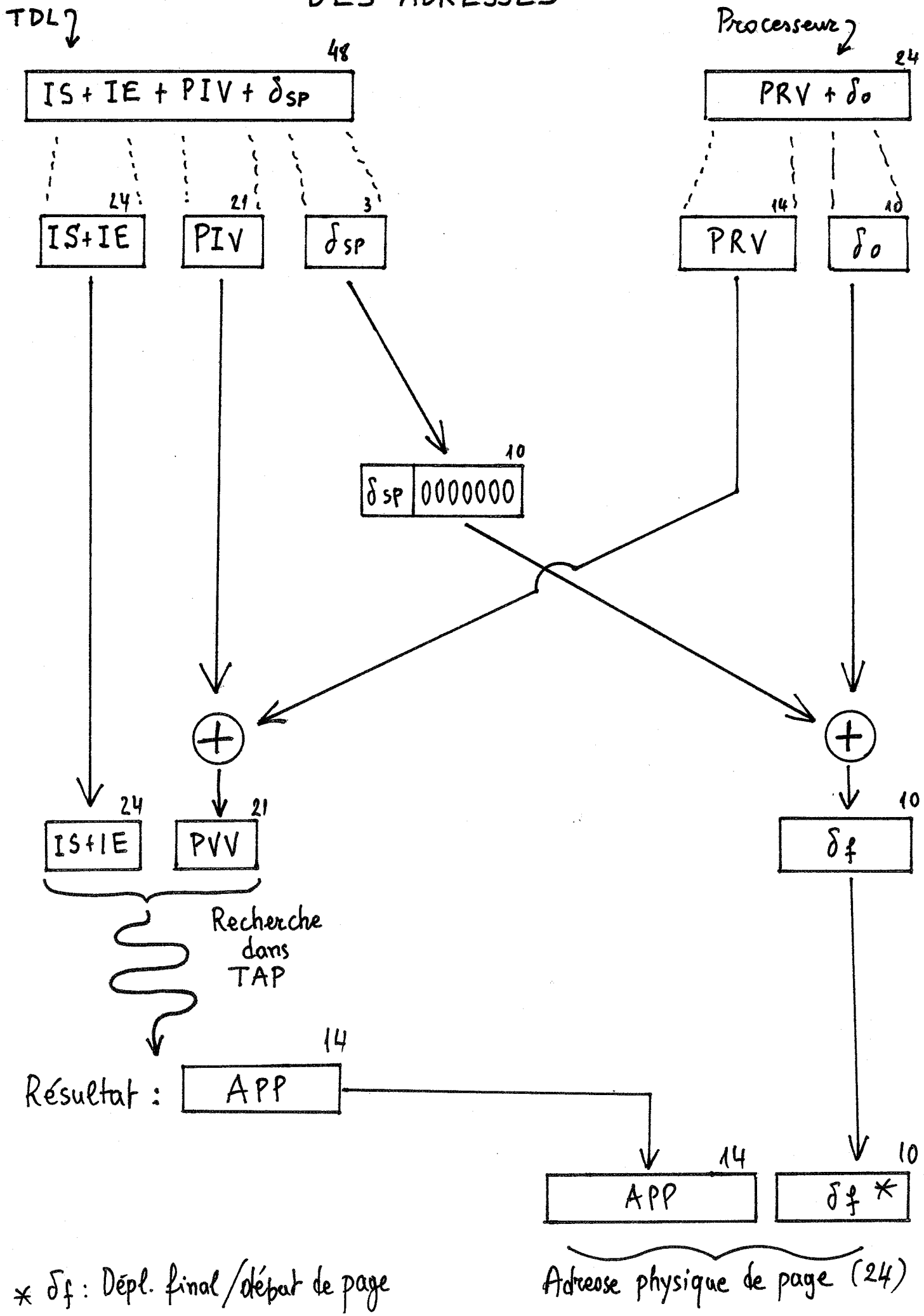
Voici le schéma de ce calcul :



df : Déplacement final (les 10 bits de poids faibles de l'adresse physique de l'élément).

La page suivante montre une traduction complète d'adresse.

TRADUCTION DES ADRESSES



4.3.2.3 Recherche dans la table des pages; récupération de l'adresse physique de page.

Dans cette dernière partie de la traduction des adresses, il faut, à partir de l'adresse véritable virtuelle de la page ou se trouve l'élément, de trouver l'adresse d'implantation physique équivalente, si bien sûr cette dernière se trouve en mémoire centrale.

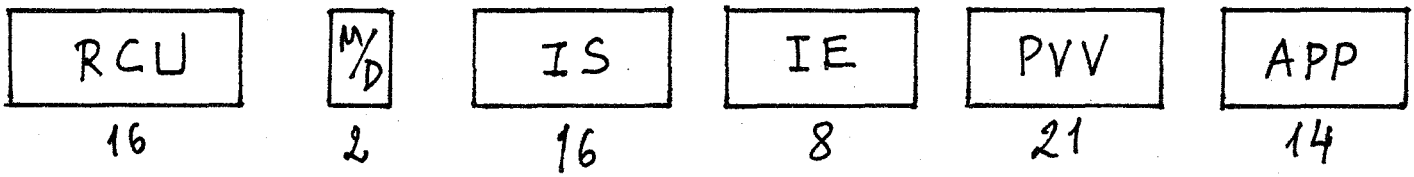
Pour celà, l'UGM-APL va consulter une table associative qui possède un petit nombre d'entrées (de l'ordre de 32 ou 128). Dans chacune de ces entrées, on trouve l'adresse virtuelle d'une page et son adresse d'implantation physique correspondante. Elle contient toutes les pages les plus récemment accédées par l'UGM-APL.

Ce principe d'une table associative des pages les plus fréquement accédées est en fait assez connu. A titre d'exemple, le co-processeur de gestion mémoire (NS 16Ø82) du microprocesseur NS 16Ø32 (National Semiconductors) peut gérer une mémoire virtuelle de 16 Méga-octets. Pour celà, il possède une table associative de 32 entrées qui lui permet de stocker les adresses physiques des 32 pages les plus récemment référencées. D'après les ingénieurs de National Semiconductors, dans 97% des cas le traducteur obtient l'adresse physique de la page directement par cette

table. Ce n'est donc que dans 3% des cas qu'il est obligé, soit de récupérer l'adresse dans les tables, soit de créer une interruption "défaut de page".

La structure associative qui permet à l'UGM-APL de trouver l'adresse physique d'une parmi les pages les plus référencées, s'appelle Table Associative des Pages (TAP) et elle est organisée de la manière suivante :

Déscription d'une entrée dans la table TAP (77 bits)



RCU : Registre Compteur d'Utilisation

M/D : Indicateurs :

M : page modifiée / non modifiée

D : entrée disponible / occupée

IS+IE : Identification de site et d'espace (même signification que dans la @ U)

PVV : Page "véritable" Virtuelle à accéder (PVV := PIV + PRV)

APP : Adresse Physique de la Page (en M.C.)

D+IS+IE+PVV : Partie associative de la table TAP

= Utilisation de la table TAP.

Le résultat de l'addition PIV+PRV donne l'adresse virtuelle de la page où se trouve l'élément à accéder (PVV). L'ensemble des informations D,IS,IE,PVV constitue la partie associative de la table TAP.

Lorsque le résultat PVV est calculé, alors une recherche associative dans TAP est declenchée, avec comme masque de recherche l'information D,IS,IE,PVV.

Pendant cette recherche associative, tous les registres RCU sont incrémentés.

Si une entrée de TAP contient l'image exacte du masque, alors la valeur de son RCU passe à zéro et la partie APP de cette même entrée constitue les 14 bits de poids forts de l'adresse physique de l'élément. En concaténant cette dernière information avec le déplacement final df, on obtient l'adresse physique de l'élément (APP,df).

Si par contre, à l'issue de la recherche associative, aucune entrée ne correspond aux informations du masque, alors une interruption de "défaut de page" est provoquée, et l'UGM-APL se désactive.

Le gérant d'interruption de défaut de page, peut alors ordonner à l'UGM-APL de copier toute la table dans la mémoire centrale et à partir d'une adresse qu'il lui précise dans un registre spécial prévu pour

cet effet et au moment de la commande. Ce registre s'appèlle Registre d'ADresses (RAD) et sa taille est de 32 bits. Lorsque le processeur a 24 bits d'adresses, alors l'octet de poids forts de ce registre contient toujours la valeur Ø. Une fois cette copie terminée, le gérant d'interruption pourra consulter les compteurs d'utilisation de chaque entrée pour décider de la page qui sera évacuée.

Lorsque le gérant a terminé le transfert des pages, il met à jour cette table et, avec une commande de chargement, l'UGM-APL met à son tour à jour la table TAP.

4.3.3 Les algorithmes de fonctionnement

Après avoir présenté les actions fondamentales et les éléments matériels nécessaires pour la réalisation de l'accès à un élément d'un objet, abordons finalement les algorithmes de fonctionnement de cette phase, aussi bien de la part du processeur que de l'UGM-APL.

Comme on l'a vu aux paragraphes précédents, les actions entreprises par le processeur peuvent se résumer aux deux instructions suivantes :

- Chargement dans RDC et RDL de la valeur Øxxxxxxx,dl. Ce chargement a pour effet d'activer l'UGM-APL en mode "accès à un selément d'un objet".

- Exécution d'une instruction d'adressage en mode indéxé (lecture ou écriture).

L'algorithme de fonctionnement de l'UGM-APL, quant à lui, est un peu plus complexe :

Algorithme de fonctionnement de l'UGM-APL lors de l'accès à un élément d'un objet.

```
- Activation de l'UGM-APL
  /* le processeur écrit dans RDC et RDL */

- Envoi d'un signal d'arrêt au processeur
  /* (HALT) */

/* Contrôle sur la validité de la correspondance */
   SI TDL(dl) non-valide ALORS
      interruption("correspondance non valide")
      FIN
      FINALORS

     SINON suspendre_arret_processeur /* HALT */
           /* l'UGM-APL attend l'instruction
           d'adressage indéxé de l'élément
           par le processeur */
   FINSI

- Détecter le cycle "validation des informations
  sur le bus d'adresses" et changer cette
  adresse dans le registre RAE

- Détecter le mode d'accès : lecture ou écritrure

- Envoi d'un signal d'arrêt au processeur
  /* agir sur HALT */

/* Première phase du calcul d'adresse */
- Exécution en parallèle des modules suivants :
   MODULE1 /* calcul de la page véritable virtuelle */
```

```
      PVV := PIV + PRV
   FINMODULE1

   MODULE2 /* calcul du déplacement final */
      df := (dsp , 0000000) + do
   FINMODULE2

/* Deuxième phase du calcul d'adresse :
   Calcul de l'adresse physique de la page
   Recherche associative dans TAP */

masque := IS , IE , PVV
recherche_ass(TAP)

   SI existe i t.q. vect_res(i) = 1 ALORS
      RCU(TAP(i)) := 0

      SI instr_lecture ALORS
         M(TAP(i)) := 1
      FINSI

      emettre APP en sortie
      FIN
      FINALORS

     SINON
        interruption("défaut de page")
     FINSINON

      FIN

   FINSI
```

Procédure FIN :

- Suspendre l'arrêt du processeur
  /* agir sur HALT */

- Désactivation de l'UGM-APL.

4.4 La libération des correspondances

La dernière action qui est effectuée pendant un traitement local est la libération des toutes les correspondances "référence universelle<---->désignation locale" qui étaient établies au début de celui-ci. Cette action peut être nécessaire car elle permet la libération des entrées dans la table TDL qui étaient occupées depuis le début du traitement local et qui, à priori, ne seront pas utiles dans un avenir immédiat.

Pour préserver la même philosophie d'utilisation de l'UGM-APL de la part du processeur, la réalisation d'une libération de correspondance emploie les mêmes mécanismes que ceux utilisés lors d'un établissement de correspondance ou d'accès à un élément. Plus précisement, le processeur a à sa disposition deux commandes avec lesquelles il peut libérer les correspondances :

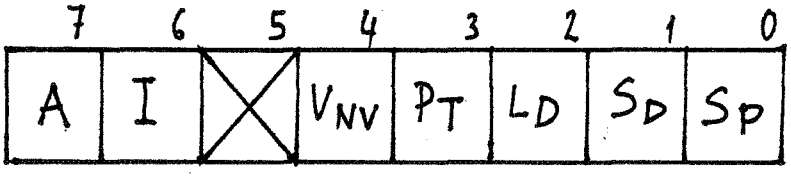
- Commande globale : le processeur écrit dans le registre RDC la valeur 10x11x10 (la valeur dans RDL peut être quelconque car elle est ignorée). Cette commande a comme effet de rendre non-valides toutes les entrées de la table TDL.

- Commande sélective : le processeur écrit dans le registre RDC la valeur 10x10x10 et dans RDL la valeur d'une désignation locale dl. L'UGM-APL rend non-valide uniquement l'entrée no dl de la table TDL. Le processeur utilise cette deuxième commande sélective lorsqu'il veut maintenir certaines correspondances pour les utiliser dans des traitements locaux ultérieurs.

4.5 Les commandes et contrôles de l'UGM-APL

Jusqu'a maintenant, nous avons analysé le comportement et la faisabilité des trois opérations fondamentales de l'UGM-APL qui réalisent un traitement local, à savoir l'établissement des correspondances, l'accès à un élément d'un objet et la libération des correspondances. Or, comme l'UGM-APL est indépendante pendant son fonctionnement, il est nécessaire que le processeur puisse apporter, à des instants précis, des modifications aussi bien dans son comportement, qu'aux informations qu'elle connait. C'est là le rôle des différentes commandes, adressées par le processeur que l'on va détailler dans ce paragraphe.

4.5.1 La définition du registre RDC



bit Ø : si = 1, alors sélection de la table associative des pages (TAP).

bit 1 : si = 1, alors sélection de la table des désignations locales (TDL).

bit 2 : Chargement/copie en mémoire centrale (Ø/1). Ce bit determine si une information doit être chargée dans une des tables de l'UGM-APL (chargement) ou, au

contraire, si elle doît être copiée dans un emplacement précis de la mémoire centrale par l'UGM-APL (copie).

bit 3 : Opération partielle/totale (Ø/1). Determine s'il s'agit d'une opération de transfert sur une seule entrée d'une table, ou sur la totalité de la table.

bit 4 : Entrée valide/non-valide (Ø/1). Ce bit valide ou rend non-valides les informations d'une entrée d'une table.

bit 5 : Réservé.

bit 6 : si = 1, alors l'UGM-APL est initialisée.

bit 7 : si = Ø, alors il s'agit d'un accès à un élément d'un objet.

4.5.2 Les commandes de l'UGM-APL

Le principe de la transmission des commandes par le processeur vers l'UGM-APL procède exactement avec le même esprit que celui qui régit les communications qui existent lors d'un traitement local. Bien évidement, lorsque l'exécution d'une commande est achevée,

A) Actions sur la table TDL

1) Copie globale de TDL ("dump")

C'est un établissement global de correspondances

i) RAD <--- @adr en mémoire centrale
ii) RDC <--- 10x01010
RDL <--- xxxxxxxx
iii) En @adr, le processeur trouve
une image de TDL

2) Copie sélective de TDL ("dump" sélectif)

i) RDC <--- 10x00010
RDL <--- dl
ii) Dans RAU(0 , 67) le processeur récupère
l'image de l'entrée no dl de TDL

3) Chargement global de TDL ("load")

i) A partir d'une @adr en mémoire centrale,
le processeur charge une image de TDL
ii) RAD <--- @adr en mémoire centrale
iii) RDC <--- 10x01110
RDL <--- xxxxxxxx

4) Chargement sélectif de TDL ("load" sélectif)

C'est un établissement sélectif de correspondance.

RAU <--- @U , V , D , L
RDC <--- 10x00110
RDL <--- dl

5) Invalidation totale de TDL

C'est une libération globale de correspondances.

RDC <--- 10x11x10

6) Invalidation sélective de TDL

C'est une libération sélective de correspondances.

RDC <--- 10x10x10
RDL <--- dl

7) Accès à un élément d'un objet

i) RDC <--- 0xxxxxxx
RDL <--- dl
ii) Adressage indexé de l'élément

B) Actions sur la table TAP

1) Copie globale de TAP ("dump")

i) RAD <--- @adr en mémoire centrale
ii) RDC <--- 10x01001
RDL <--- xxxxxxxx
iii) En @adr, le processeur trouve
une image de TAP

2) Copie sélective de TAP ("dump" sélectif)

i) RDC <--- 10x00001
   RDL <--- j
ii) Dans RAU(0 , 76), le processeur récupère
   l'image de l'entrée no j de TAP

3) Chargement global de TAP ("load")

i) A partir d'une @adr en mémoire centrale,
   le processeur charge une image de TAP
ii) RAD <--- @adr en mémoire centrale
iii) RDC <--- 10x01101
   RDL <--- xxxxxxxx

4) Chargement sélectif de TAP ("load" sélectif)

   RAU <--- image d'une entrée de TAP
   RDC <--- 10x00101
   RDL <--- j (no de l'entrée à mettre à jour)

5) Invalidation totale de TAP

   RDC <--- 10x11x01
   RDL <--- xxxxxxxx

6) Invalidation sélective de TAP

   RDC <--- 10x10x01
   RDL <--- j (entrée à invalider)

C) L'initialisation de l'UGM-APL

Elle se fait de manière extrèmement simple : le processeur charge dans RDC la valeur 11xxxxxx (la valeur de RDL est ignorée). Activée ainsi, l'UGM-APL libère toutes les entrées de ses tables et initialise ses registres avec une valeur prédeterminée.

4.5.3 Le contrôle sur l'UGM-APL

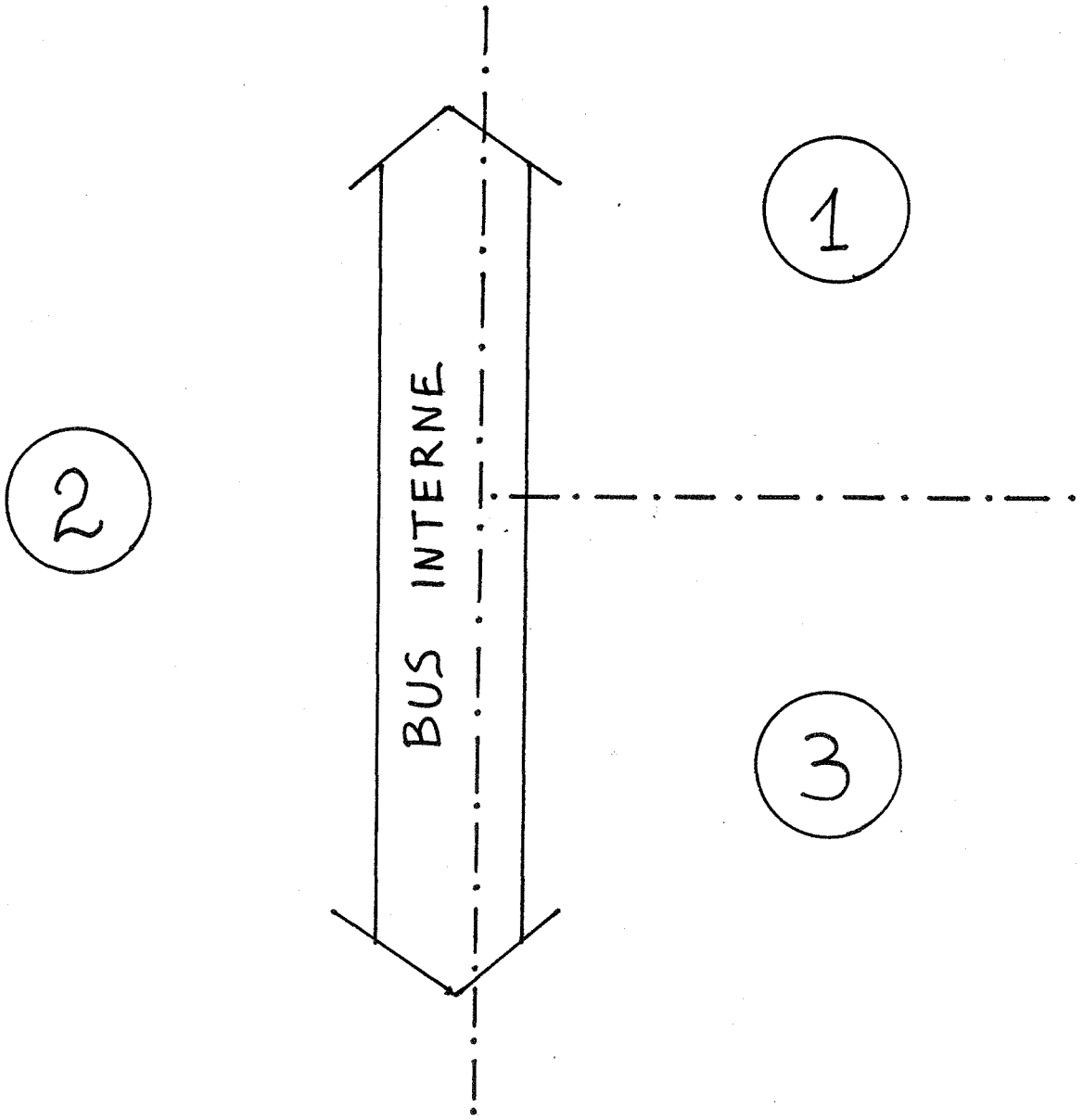
Le Registre de Status (RS) contient toutes les informations concernant l'état de fonctionnement de l'UGM-APL. Il peut être consulté par le processeur, et parmi les informations qu'il contient, notons les plus intéressantes :

- bit de défaut matériel ;
- défaut de page ;
- indicateur de page read-only / read-write ;
- désignation locale non définie ;
- accumulations de défauts de fonctionnement ;
- UGM-APL active ;
- erreur de commande.

4.6 Schéma général de l'UGM-APL

Pour compléter ce chapitre, nous présentons ici le schéma général de l'UGM-APL correspondant à la description qui a été faite jusqu'à présent.

Autour d'un bus interne, on trouve une décomposition de ce schéma en trois parties :



Ces différentes parties réalisent les fonctions suivantes :

(1) : - Etablissement et libération des correspondances
- Eléments de contrôle global sur les objets.

(2) : - Mécanisme de traduction des adresses;
- Eléments de contrôle des accès au niveau de la page.

(3) : - Séquencement du fonctionnement de l'UGM-APL;
- Interprétation et séquencement des commandes;
- Status et mode de fonctionnement;
- Mécanisme et logique des interruptions.

4.7 BIBLIOGRAPHIE DES CHAPITRES 3 ET 4

A) Ouvrages et articles généraux

[DEN 70] P. Denning : Virtual Memory ; Computing Surveys vol. 2 No 3, Sept. 1970.

[SNY 79] A. Snyder : A machine Architecture to Support an Object-Oriented Language ; PhD thesis, MIT 1979.

[BER 81] Ch. Bertin : Aspects de la réalisation d'un système APL optimisé ; Thèse de docteur Ingénieur, EMSE 1981.

[GIR 76] J.J. Girardot, F. Mireaux : Réalisation d'un interprète complet du langage APL sur un mini-ordinateur ; Nancy, 1976.

[GIR 82] J.J. Girardot : Architecture logicielle et matérielle d'une machine APL ; Journées APL de l'AFCET, Novembre 1982.

[MYE 82] G.J. Myers : Advances in Computer Architecture ; Ed. J. Wiley, 1982.

[DUB 82] R. Dubois, D. Girod : Les microprocesseurs 16 bits à la loupe ; Ed. Eyrolles, 1982

[REP 82-a] Special report on microsystems design techniques ; Computer design, June 1982.

[REP 82-b] Special report on supersystems designing

for high performance ; Computer design, Aug. 1982.

[REP 83] Special report on microprocessors and microcomputers ; Computer design, Oct. 1983.

[GRE 83] D. Mac Gregor, D.S. Mothersole : Virtual Memory and the MC68Ø1Ø ; IEEE MICRO, June 1983.

[BAS 8Ø] F. Basket : Pascal and Virtual Memory in a Z8ØØØ or MC68ØØØ Based Design Station ; IEEE 198Ø.

[RUM 83] M. van Rumste : The iAPX 432, a next generation microprocessor ; Microprocessing and Microcomputing 11,1983.

[AHU 8Ø] S.R. Ahuja, C.S. Roberts : An Associative/ Parallel Processor for Partial Match Retrieval Using Superimposed Codes ; IEEE 198Ø.

[SCH 83] S. Schmitt : Virtual Memory for Microcomputers : Four New Memory-Management Chips Pave The Way ; Byte, Apr. 1983.

[LAM 83] S. M. Lamb : Content-addressable memory uses 256-byte "superwords" ; Mini-Micro Systems, Oct. 1983.

[MET 83] S. M. Metcalf, R. M. Farber : 32-bit "Mega-micro" exploits hardware virtual memory and "RAM disk" ; Mini-Micro Systems, Oct. 1983.

[CRA 83] N. R. Crandal : Shared-memory computer handles heavy work loads ; Mini-Micro Systems, Oct. 1983.

[SAk 82] U. Sakellaridis : Les microprocesseurs de la deuxième génération : l'architecture matérielle des microprocesseurs 16 et 32 bits - les circuits spécialisés des familles des microprocesseurs 16 et 32 bits ; Journées APL de l'AFCET, APL et la micro-informatique, Lille, novembre 1982.

B) Documentation des constructeurs

[INT Ø1] Component Data Catalog, Jan. 1981.

[INT Ø2] MCS-86 User's Manual, July 1978.

[INT Ø3] iAPX 186 Technical Data, 1982.

[INT Ø4] iAPX 286 Technical Data, 1982.

[MOT Ø1] MC68ØØØ Advance Information

[MOT Ø2] MC68451 Advance Information

[ZIL] Z8ØØØ Family, Feb. 198Ø.

[NS Ø1] NS 16Ø32 High- performance Microprocessor

[NS Ø2] NS 16Ø82 Memory Management Unit (MMU)

[NS Ø3] NS 16ØØØ Microprocessor Family. Reprint of Technical Articles.

[APO 81] Apollo Domain Architecture ; Apollo Computer Inc. 1981.

[SIG -a] 16-bit ECL CAM (8*2) data sheet.

[SIG -b] 8220 Content Addressable Memory Element (CAM).

(OUF !)

CONCLUSION

Tout au long de ce travail, nous nous sommes appliqués à orienter notre étude autour d'un domaine précis de la science informatique : les postes de travail autonomes scientifiques et leurs environnements logiciels de programmation. Nous estimons que ce domaine a pris beaucoup d'importance ces dernières années et que dans un proche avenir deviendra une des branches fondamentales de l'informatique.

La réalisation future des postes de travail autonomes scientifiques reposera sur une téchnologie matérielle des plus modernes de l'informatique. Plus faciles à mettre en oeuvre que les ordinateurs traditionnels, les postes autonomes constituent le domaine idéal de recherche sur les nouvelles architectures matérielles de systèmes. Nous pensons que, à moyen terme, le degré de miniaturisation, les performances sans cesse améliorées et le prix de revient de plus en plus faible du matériel de base, seront les raisons essentielles pour lesquelles le poste de travail autonome pourra avoir des performances supérieures aux mini-ordinateurs actuels, et remplacera définitivement le terminal. D'ailleurs, les idées du projet japonnais sur les ordinateurs et l'informatique en général de la 5ème génération, vont dans ce même sens.

Le développement futur des environnements de programmation sera amené à influencer considérablement l'évolution du logiciel de base. Notre étude sur ce point précis a mis en évidence les avantages et les

inconvénients des deux types d'environnements existants ("classique et orienté vers l'intelligence artificielle). Un effort qui regroupe leurs avantages et qui unifie un certain nombre de notions, souvent très divergentes, a été fait avec notre étude sur un troisième type d'environnement à base d'une version adaptée d'APL. Dans cette même étude, le choix d'APL apporte directement tous les concepts relatifs à un contexte autonome de programmation, mais aussi la puissance de manipulation uniforme des structures élaborées. Nous pensons que ce choix devra constituer la dernière étape avant celle des langages naturels.

Il est toutefois vrai que le choix d'APL pose aussi le problème de la gestion de ses objets, extrêmement dynamiques par définition, à l'intérieur d'un espace virtuel paginé. C'est ce problème que nous avons étudié avec la conception d'un processus spécialisé de gestion des objets et d'accès à leurs éléments. Cette étude à été menée suffisamment loin pour que notre unité de gestion mémoire puisse être réalisable entierement en matériel.

A ce niveau, nous avons été amenés à dégager certains concepts relatifs au comportement de l'interprète lorsqu'il accéde aux objets. Ainsi les notions de référence universelle, désignation locale et établissement de correspondances au sein d'un traitement local, ont permis une gestion des objets plus rapide, plus souple et plus facile (cf. Annexe 1).

La conception de notre UGM-APL permet son utilisation au sein de systèmes autonomes avec peu de modifications de structure. Elle peut aussi être utilisée au profit d'autres contextes que APL, avec

pratiquement la même efficacité.

Comment peut-on envisager une réalisation matérielle de notre UGM-APL ? Une étude du marché de la fabrication des cartes spécialisées montre qu'une telle carte pourrait se vendre assez facilement si son prix n'exède pas les 16000FF, et devient rentable au delà de la 1000ième carte vendue. Il nous semble donc tout à fait possible d'envisager une réalisation sous forme de carte.

La fabrication d'un circuit spécialisé à la demande ne peut pas être envisagée pour l'UGM-APL. En effet, l'investissement dans ce cas est très important et un produit peut être considéré rentable qu'à partir de la 100000ième piece vendue. Par contre, sa réalisation à partir de circuits prédiffusés (FPLA : functionnal programmable logic arrays), reste dans le domaine du possible, parce que ces derniers sont beaucoup plus économiques. On estime en effet pouvoir rentabiliser les circuits fabriqués à partir des circuits prédiffusés, au delà du 100ième vendu.

L'évolution de l'informatique autonome de haut niveau se montre très prometteuse pour les années à venir. Nous estimons que notre but, proposer des solutions à certains problèmes matériels et logiciels qui nous semblent fondamentaux, a été largement atteint.

ANNEXE 1 : Les mesures théoriques sur l'UGM-APL

Il est difficile d'évaluer avec précision le gain obtenu par l'utilisation de l'UGM dans un système APL. En effet son utilisation entraîne une simplification de l'accès à tous les objets gérés par l'interprète depuis les tables d'allocations de la mémoire virtuelle, jusqu'aux variables APL, en passant par les fonctions utilisateurs et les tables de symboles. Elle influencera donc considérablement la conception de tout le système rendant les structures de données internes moins complexes et moins volumineuses et par conséquent plus faciles à gérer.

Nous allons toutefois présenter quelques résultats en nous appuyant sur le système APL/800 réalisé à l'Ecole des Mines de Saint-Etienne [BER 81]. Ce système est destiné au mini-oridnateur 16 bits Philips P857. Il se prête relativement bien à la comparaison puisqu'il gérait pratiquement complètement par logiciel une mémoire virtuelle paginée (avec des pages de 4K-octets). Nous supposerons que toutes les optimisations d'accès proposées dans [BER 81] ont été réalisées.

Dans tout ce qui suit nous supposons qu'il n'y a pas de défaut de page : les pages à accéder font partie de la configuration sous le système APL/800 et sont présentes dans la mémoire associative de l'UGM.

Le temps de traversée de l'UGM peut être évalué ainsi :

	mini (ns)	maxi (ns)
- Contrôle de la validité des correspondances	20	60
- Blocages des bus	20	60
- Calcul d'adresse	60	100
- Recherche associative	60	100
- Mise à jour de TAP	60	100
- Réactivation du processeur	20	60
- Traversées et chargement des registres internes	60	100
TOTAL :	300ns	580ns

Dans ces conditions et en fonction du jeu d'instructions du P800, nous pouvons considérer que l'accès à un élément coute 4 instructions lorsque la correspondance a été établie.

Pour accéder à un mot dans un objet il faut dans le système APL/800 environ 80 instructions machine. Avec l'UGM ce nombre se ramènerait à 10 en comptant l'établissement de la correspondance. Le gain est donc important mais il faut le tempérer par le fait que les accès suivants ne nécessitent plus que 3 instructions pour APL/800 alors qu'il en faut 4 avec l'UGM. Pour l'accès à quelques éléments le gain absolu reste toutefois important.

Considérons maintenant l'accès aux éléments d'un tableau APL au cours du traitement d'une fonction scalaire. Pour des opérandes mono-page l'accès à un élément prend 3 instructions dans APL/800. Pour des opérandes de plusieurs pages il faut rajouter 2 instructions permettant de tester le passage à la page suivante. Le temps de passage à la page suivante (de l'ordre de 80 instructions en supposant la page dans le "working set") est en fait peu important si on le rapporte au nombre d'éléments par page (1000 entiers de 16 bits par page de 4Koctets dans APL/800), évaluons-le à 0,5 instruction ce qui donne 5,5 instructions par accès. Nous pouvons donc construire le tableau suivant:

	APL/800	avec UGM
Accès aléatoire		
-premier accès	80	10
-accès suivants	80	4
Accès séquentiel		
-premier accès	80	10
-accès suivants (mono-page)	3	4
-accès suivants (multi-page)	5,5	4

Il faut également évaluer le gain relatif dans un ensemble d'opérations plus complexes. Prenons un cas assez défavorable à l'UGM; l'évaluation d'une fonction scalaire dyadique où l'accès aux éléments est séquentiel. Soit donc l'expression "A+B". Un comptage des instructions donne 750 instructions pour l'analyse syntaxique et 600 instructions d'initialisation de la fonction dyadique (accès aux descripteurs des opérandes, création du résultat, génération des procédures d'accès -qui sont compilées dans APL/800). Nous évaluons à 200 le nombre d'instructions

gagnées au cours de cette phase en utilisant l'UGM.

La boucle centrale d'exécution de la fonction et de passage à l'élément suivant peut, dans les deux cas être évaluée à 15 instructions, pour des opérandes non optimisés. Ce qui donne finalement:

	APL/800	avec UGM
analyse syntaxique	750	750
initialisation	600	400
exécution	P x 15 +3xW	P x 15 +3xU

avec P le nombre d'éléments de A (et B), W et U le nombre d'instructions d'accès à un élément. W et U sont multipliés par 3 puisqu'il y a accès aux deux opérandes et au résultat.

Nous avons donc les résultats suivants pour quelques valeurs de P (le dernier exemple -2000 éléments- considère des opérandes multi-pages):

nombre d'éléments	APL/800	avec UGM	gain relatif
10	1590	1420	10,6%
100	3750	3850	-2,6%
2000	64350	55550	13,7%

Ces gains (ou pertes) ne sont donc pas très importants. Il faut toutefois considérer que le système envisagé dans le cadre d'un poste de travail offre à l'utilisateur beaucoup de possibilités nouvelles (la suppression de la notion de fichiers, par exemple) qui pour la plupart entraîneront une augmentation importante de la taille des objets traités et surtout, vu cette taille, une augmentation des accès de type aléatoire (indexation). De plus les gains peuvent devenir beaucoup plus importants dans le cadre de la compilation des fonctions APL, compilation qui ferait énormément diminuer les coûts fixes constatés ci-dessus.

Bibliographie

[BER 81] Bertin Christian : Aspects de la réalisation d'un système APL optimisé. Thèse de Docteur Ingénieur, 1981

ANNEXE 2 : La simulation

```
        A INITIALISATION DE LA ZONE

        INIT


        A ETAT DE L'UGM

        STA
RDCRDL/0000   RS/0000   RAU/00000000000000000000   RAD/00000000
                        RIO/00000000000000000      RAE/00000000


        A RESET DE L'UGM
        RESET

        A NOUVEL ETAT DE L'UGM
        STA
RDCRDL/0100   RS/0000   RAU/00000000000000000000   RAD/00000000
                        RIO/00000000000000000      RAE/00000000


        A TABLE DES REFERENCES
        DTARU
 0/00   0000 00   000000 0 0000
 1/01   0000 00   000000 0 0000
 2/02   0000 00   000000 0 0000
 3/03   0000 00   000000 0 0000
 4/04   0000 00   000000 0 0000
 5/05   0000 00   000000 0 0000


        A TABLE DES PAGES
        DTAP
 0/00   000000000000 0000   0000 0
 1/01   000000000000 0000   0000 0
 2/02   000000000000 0000   0000 0
 3/03   000000000000 0000   0000 0
 4/04   000000000000 0000   0000 0
 5/05   000000000000 0000   0000 0
 6/06   000000000000 0000   0000 0
 7/07   000000000000 0000   0000 0


        A LECTURE ADRESSE 28
        WLOAD AD '28'
* MLOAD          000028 --> FFFF
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1


        A ECRITURE A CETTE ADRESSE
        0 1 0 1  1 1 0 0    0 0 0 0  1 1 1 1  WSTORE AD '28'
* MSTORE         5C0F ==> 000028


        A RELECTURE DE CE MOT
        WLOAD AD '28'
* MLOAD          000028 --> 5C0F
0 1 0 1 1 1 0 0 0 0 0 0 1 1 1 1
```

```
      A CREATION DE L'OBJET  "ι10"
      OBJET←IOTA 10
* SETRDCRDL  6901
* SETRDCRDL  : 6
* SETRDCRDL  0001  ⊠
* WSTORE       000000     0001 <== 0000
* 000019800000   0
DEFAUT UGM
RDCRDL/0001   RS/2010   RAU/0000CC000001000010000   RAD/00000000
                        RIO/0000CC00000180010       RAE/00000000
* SETRDCRDL  9100
* SETRDCRDL  : 1
* MSTORE         0000 ==> 000010
* MSTORE         0000 ==> 000012
* MSTORE         0000 ==> 000014
* MSTORE         0000 ==> 000016
* MSTORE         0000 ==> 000018
* MSTORE         0000 ==> 00001A
* MSTORE         0000 ==> 00001C
* MSTORE         0000 ==> 00001E
* MSTORE         0000 ==> 000020
* MSTORE         0000 ==> 000022
* MSTORE         0000 ==> 000024
* MSTORE         0000 ==> 000026
* MSTORE         0000 ==> 000028
* MSTORE         0000 ==> 00002A
* MSTORE         0000 ==> 00002C
* MSTORE         0000 ==> 00002E
* MSTORE         0000 ==> 000030
* MSTORE         0000 ==> 000032
* MSTORE         0000 ==> 000034
* MSTORE         0000 ==> 000036
* MSTORE         0000 ==> 000038
* MSTORE         0000 ==> 00003A
* MSTORE         0000 ==> 00003C
* MSTORE         0000 ==> 00003E
* MSTORE         0000 ==> 000040
* MSTORE         0000 ==> 000042
* MSTORE         0000 ==> 000044
* MSTORE         0000 ==> 000046
* MSTORE         0000 ==> 000048
* MSTORE         0000 ==> 00004A
* MSTORE         0000 ==> 00004C
* MSTORE         0000 ==> 00004E
* MSTORE         0000 ==> 000050
* MSTORE         0000 ==> 000052
* MSTORE         0000 ==> 000054
* MSTORE         0000 ==> 000056
* MSTORE         0000 ==> 000058
* MSTORE         0000 ==> 00005A
* MSTORE         0000 ==> 00005C
* MSTORE         0000 ==> 00005E
LECTURE DE LA PAGE 000019800000
* SETRDCRDL  A900
* SETRDCRDL  : 2
```

```
LA PAGE EST A L'ADRESSE PHYSIQUE 0400
INSTALLATION DANS L'ENTREE 0
* MSTORE          0000 ==> 000480
* SETRDCRDL   0001  ⊠
* WSTORE         000002     0001 <== 0001
* 000019800000    0
* MSTORE          0001 ==> 000482
* SETRDCRDL   0001  ⊠
* WSTORE         000004     0001 <== 0002
* 000019800000    0
* MSTORE          0002 ==> 000484
* SETRDCRDL   0001  ⊠
* WSTORE         000006     0001 <== 0003
* 000019800000    0
* MSTORE          0003 ==> 000486
* SETRDCRDL   0001  ⊠
* WSTORE         000008     0001 <== 0004
* 000019800000    0
* MSTORE          0004 ==> 000488
* SETRDCRDL   0001  ⊠
* WSTORE         00000A     0001 <== 0005
* 000019800000    0
* MSTORE          0005 ==> 00048A
* SETRDCRDL   0001  ⊠
* WSTORE         00000C     0001 <== 0006
* 000019800000    0
* MSTORE          0006 ==> 00048C
* SETRDCRDL   0001  ⊠
* WSTORE         00000E     0001 <== 0007
* 000019800000    0
* MSTORE          0007 ==> 00048E
* SETRDCRDL   0001  ⊠
* WSTORE         000010     0001 <== 0008
* 000019800000    0
* MSTORE          0008 ==> 000490
* SETRDCRDL   0001  ⊠
* WSTORE         000012     0001 <== 0009
* 000019800000    0
* MSTORE          0009 ==> 000492
* SETRDCRDL   6101
* SETRDCRDL   : 6


      A RU DU RESULTAT
      OBJET
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

      A EN HEXADECIMAL, PLUTOT
      D OBJET
0000CC0000010001


      A IOTA UTILISE LA DESIGNATION LOCALE ¨1¨
      DTARU
 0/00   0000 00   000000 0 0000
 1/01   0000 CC   000000 0 0010
 2/02   0000 00   000000 0 0000
 3/03   0000 00   000000 0 0000
 4/04   0000 00   000000 0 0000
 5/05   0000 00   000000 0 0000
```

```
      ⍝ ETAT DE LA TABLE DES PAGES
      DTAP
 0/00  0000CC000000 0001  0000 6
 1/01  000000000000 0000  000A 0
 2/02  000000000000 0000  000A 0
 3/03  000000000000 0000  000A 0
 4/04  000000000000 0000  000A 0
 5/05  000000000000 0000  000A 0
 6/06  000000000000 0000  000A 0
 7/07  000000000000 0000  000A 0


      ⍝ L'ENTREE 0 EST VALIDE ET CORRESPOND A UNE PAGE MODIFIEE
      ⍝ LES AUTRE ENTREES ONT UN COMPTEUR EGAL A 10


      ⍝ NOUVEAU CALCUL DE IOTA
      OBJ2←IOTA 3
* SETRDCRDL  6901
* SETRDCRDL  : 6
* SETRDCRDL  0001  ⊠
* WSTORE       000000    0001 <== 0000
* 000019800000   0
* MSTORE        0000 ==> 000500
* SETRDCRDL  0001  ⊠
* WSTORE       000002    0001 <== 0001
* 000019800000   0
* MSTORE        0001 ==> 000502
* SETRDCRDL  0001  ⊠
* WSTORE       000004    0001 <== 0002
* 000019800000   0
* MSTORE        0002 ==> 000504
* SETRDCRDL  6101
* SETRDCRDL  : 6


      ⍝ PAS DE FAUTE DE PAGE, CETTE FOIS,
      ⍝ CAR L'ALLOCATION S'EST FAITE DANS LA MEME.



      ⍝ ETAT DE LA TABLE DES PAGES
      DTAP
 0/00  0000CC000000 0001  0000 6
 1/01  000000000000 0000  000D 0
 2/02  000000000000 0000  000D 0
 3/03  000000000000 0000  000D 0
 4/04  000000000000 0000  000D 0
 5/05  000000000000 0000  000D 0
 6/06  000000000000 0000  000D 0
 7/07  000000000000 0000  000D 0


      ⍝ MODIFIONS LE POINTEUR D'ALLOCATION EN MEMOIRE VIRTUELLE
      ⍝ POUR PROVOQUER UNE NOUVELLE FAUTE DE PAGE


      FREEPT←47×128
```

```
      A NOUVEAU IOTA...
      OBJ3←ïIOTA 5
* SETRDCRDL  6901
* SETRDCRDL  : 6
* SETRDCRDL  0001  ⊠
* WSTORE       000000     0001 <== 0000
* 000019800005   5
DEFAUT UGM
RDCRDL/0001   RS/2010   RAU/0000CC00002F00010000   RAD/00000010
                        RIO/0000CC00002F80010      RAE/00000000
* SETRDCRDL  9100
* SETRDCRDL  : 1
* MSTORE         0000 ==> 000010
* MSTORE         3300 ==> 000012
* MSTORE         0000 ==> 000014
* MSTORE         000E ==> 000016
* MSTORE         0000 ==> 000018
* MSTORE         0000 ==> 00001A
* MSTORE         0000 ==> 00001C
* MSTORE         0000 ==> 00001E
* MSTORE         0000 ==> 000020
* MSTORE         000D ==> 000022
* MSTORE         0000 ==> 000024
* MSTORE         0000 ==> 000026
* MSTORE         0000 ==> 000028
* MSTORE         0000 ==> 00002A
* MSTORE         000D ==> 00002C
* MSTORE         0000 ==> 00002E
* MSTORE         0000 ==> 000030
* MSTORE         0000 ==> 000032
* MSTORE         0000 ==> 000034
* MSTORE         000D ==> 000036
* MSTORE         0000 ==> 000038
* MSTORE         0000 ==> 00003A
* MSTORE         0000 ==> 00003C
* MSTORE         0000 ==> 00003E
* MSTORE         000D ==> 000040
* MSTORE         0000 ==> 000042
* MSTORE         0000 ==> 000044
* MSTORE         0000 ==> 000046
* MSTORE         0000 ==> 000048
* MSTORE         000D ==> 00004A
* MSTORE         0000 ==> 00004C
* MSTORE         0000 ==> 00004E
* MSTORE         0000 ==> 000050
* MSTORE         0000 ==> 000052
* MSTORE         000D ==> 000054
* MSTORE         0000 ==> 000056
* MSTORE         0000 ==> 000058
* MSTORE         0000 ==> 00005A
* MSTORE         0000 ==> 00005C
* MSTORE         000D ==> 00005E
LECTURE DE LA PAGE 000019800005
* SETRDCRDL  A901
* SETRDCRDL  : 2
LA PAGE EST A L'ADRESSE PHYSIQUE 0800
INSTALLATION DANS L'ENTREE 1
* MSTORE         0000 ==> 000B80
* SETRDCRDL  0001  ⊠
* WSTORE       000002     0001 <== 0001
* 000019800005   5
* MSTORE         0001 ==> 000B82
* SETRDCRDL  0001  ⊠
* WSTORE       000004     0001 <== 0002
* 000019800005   5
```

```
* MSTORE        0002 ==> 000B84
* SETRDCRDL  0001  ⊠
* WSTORE       000006     0001 <== 0003
* 000019800005   5
* MSTORE        0003 ==> 000B86
* SETRDCRDL  0001  ⊠
* WSTORE       000008     0001 <== 0004
* 000019800005   5
* MSTORE        0004 ==> 000B88
* SETRDCRDL  6101
* SETRDCRDL  : 6
```

```
      A ETAT DE LA TABLE DES PAGES
      DTAP
 0/00  0000CC000000 0001  0005 6
 1/01  0000CC000028 0002  0000 6
 2/02  000000000000 0000  0012 0
 3/03  000000000000 0000  0012 0
 4/04  000000000000 0000  0012 0
 5/05  000000000000 0000  0012 0
 6/06  000000000000 0000  0012 0
 7/07  000000000000 0000  0012 0
```

```
      A VERIFIONS QU'IL Y A BIEN EU CREATION
      A DES OBJETS DANS LA MEMOIRE PHYSIQUE
```

```
      2 □CV M0000
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF 0000 3300 0000 000E 0000 0000 0
      000 0000 0000 000D 0000 0000 0000 0000 000D 0000 0000 0000 0000 0
      00D 0000 0000 0000 0000 000D 0000 0000 0000 0000 000D 0000 0000 0
      000 0000 000D 0000 0000 0000 0000 000D FFFF FFFF FFFF FFFF FFFF F
      FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
     • • •
```

```
      A LA PAGE 0 EST CELLE OU LE PROCESSEUR EFFECTUE UN DUMP
      A DE LA TABLE DES PAGES DE L'UGM EN CAS DE DEFAUT DE PAGE.
```

```
      A LA PAGE 1 DE LA MEMOIRE PHYSIQUE CONTIENT LA PREMIERE
      A PAGE ALLOUEE DE L'ESPACE VIRTUEL
```

```
      2 □CV M0001
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
      FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
      FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
      FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
      FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF 0000 0001 0
      002 0003 0004 0005 0006 0007 0008 0009 FFFF FFFF FFFF FFFF FFFF F
      FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
      FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
      FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
      FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF 0000 0001 0002 F
      FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
     • • •
```

```
      ⍝ ON Y RETROUVE DES DEUX OBJETS CREES, IOTA 10 DE 10 ELEMENTS
      ⍝ ET IOTA 3 DE TROIS ELEMENTS.

      ⍝ DUMP DE LA SECONDE PAGE VIRTUELLE ALLOUEE

      2 ⎕CV M0002
FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
       . . .

       FFF FFFF FFFF FFFF FFFF 0000 0001 0002 0003 0004 FFFF FFFF FFFF F
       FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
       FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
       FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
       FFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF F
       FFF FFFF FFFF FFFF

      )OFF
```

```
     ∇ Z←AD X
[1]    ⍝ CONVERSION D'UNE ADRESSE DECIMAL OU HEXA
[2]    →(' '=1↑0⍴X)/HEX
[3]    Z←1=,(24⍴2)⊤X
[4]    →0
[5]   HEX:Z←24 H X
     ∇

     ∇ OBJ←ALLOCATE N;Z
[1]    ⍝ MINI-GESTION DE MEMOIRE
[2]    ⍝ ALLOCATION DE N OCTETS
[3]    ⍝ ON ARRONDIT N A UN MULTIPLE DE 128
[4]    N←128×⌈N÷128
[5]    ⍝ ADDRESSE MEMOIRE ALLOUEE
[6]    Z←FREEPT
[7]    ⍝ M.A.J. FREEPT
[8]    FREEPT←FREEPT+N
[9]    ⍝ GENERATION D'UNE ADRESSE EFFECTIVE
[10]   OBJ←V_SITE,V_VIRT,(24 H Z÷128),16 H N÷128
     ∇

     ∇ CREATE_MEMORY V
[1]    ⍝ CREATION DE LA MEMOIRE
[2]    V←,V
[3]   LP:→(0=⍴V)/0
[4]    ⍎'M',,'0123456789ABCDEF'[16 16 16 16⊤V[0]],'←512⍴¯1'
[5]    V←1↓V
[6]    →LP
     ∇

     ∇ Z←D X
[1]    ⍝ IMPRESSION EN FORMAT HEXA
[2]    Z←'0123456789ABCDEF'[,2 2 2 2⊥⍉(Z,4)⍴(-4×Z←⌈(⍴X)÷4)↑X]
     ∇

     ∇ DEFAUPAGE
[1]    ⍝ TRAITEMENT DU DEFAUT PAGE
[2]    ⍝
[3]    ⍝ 1) ON DUMPE LA TABLE DES PAGE
[4]    (X '0000')WSTORE AD '110'
[5]    (X '0010')WSTORE AD '112'
[6]    (X '9100')WSTORE AD '100'
[7]    ⍝ ON CHOISIT LA PAGE A EVACUER.
[8]    V←8+4+5×⍳NP
[9]    MAX←⌈/M0000[V]
[10]   NUM←M0000[V]⍳MAX
[11]   ⍝ ON INSTALLE LA PAGE EN MEMOIRE.
[12]   'LECTURE DE LA PAGE ';D IDPAGE
[13]   →(¯1∈VPAGES)/OK1
[14]   *
[15]  OK1:NN←VPAGES⍳¯1
[16]   VPAGES[NN]←2⊥IDPAGE
[17]   ⍝ ON INDIQUE A UGM QUE LA PAGE EST LA
[18]   NUM EDP D IDPAGE,0 0 0,16 H NN
[19]   'LA PAGE EST A L''ADRESSE PHYSIQUE ';D 16 H NN×1024
[20]   'INSTALLATION DANS L''ENTREE ';D 4 H NUM
     ∇

     ∇ DTAP;I;V
[1]    ⍝ DUMP TABLE DES PAGES
[2]    I←0
[3]   LP:→(I≥1⍴⍴TAP)/0
[4]    V←(2 0⍕I),'/',(D(8⍴2)⊤I),'  ',(D TAP[I;P_ID],3⍴0),' ',(D TAP[I;P_
       ADD]),' '
[5]    V,' ',(D TAP[I;P_CNT]),' ',D TAP[I;P_MDI]
[6]    I←I+1
[7]    →LP
     ∇
```

```
     ∇ DTARU;I;V
[1]     ⍝ DUMP DE LA TABLE DES REF. UNIV.
[2]    I←0
[3]   LP:→(I≥1⍴⍴TARU)/0
[4]    V←(2 0⍕I),'/',(D(8⍴2)⊤I),'  ',(D TARU[I;C_IS]),' ',(D TARU[I;C_IE
       ]),' '
[5]    V,' ',(D TARU[I;C_PIV]),' ',(D TARU[I;C_VRDA]),' ',D TARU[I;C_L]
[6]    I←I+1
[7]    →LP
     ∇

     ∇ ENT EDL RU
[1]     ⍝ ETABLISSEMENT DESIGNATION LOCALE
[2]    RU←80↑RU
[3]    RU[B_0_15]WSTORE AD '102'
[4]    RU[B_16_31]WSTORE AD '104'
[5]    RU[B_32_47]WSTORE AD '106'
[6]    RU[B_48_63]WSTORE AD '108'
[7]    RU[B_64_79]WSTORE AD '10A'
[8]     ⍝ RENDRE VALIDE L'ENTREE
[9]    ((X '69'),8 H ENT)WSTORE AD '100'
     ∇

     ∇ ENT EDP DP
[1]     ⍝ ETABLISSEMENT DESIGNATION DE PAGE
[2]     ⍝ LA DP = 000<ID.PAGE> = 48 BITS
[3]    (X DP[0 1 2 3])WSTORE AD '102'
[4]    (X DP[4 5 6 7])WSTORE AD '104'
[5]    (X DP[8 9 10 11])WSTORE AD '106'
[6]     ⍝ ADD. MEMOIRE DE LA PAGE:
[7]     ⍝   00<NUM PAGE> = 16 BITS
[8]    (X DP[12 13 14 15])WSTORE AD '108'
[9]     ⍝ LES BITS DE PROTECTION...
[10]   ((X 'A9'),8 H ENT)WSTORE AD '100'
     ∇

     ∇ EXECUTE TST;MAT;MATIND;LINE
[1]     ⍝ EXECUTION AVEC TRACE DU PROGRAMME MAT
[2]    MAT←⎕CR TST
[3]    MATIND←1
[4]   LP:→(MATIND≥1⍴⍴MAT)/FIN
[5]    LINE←(+/∧\' '=LINE)↓LINE←MAT[MATIND;]
[6]    MATIND←MATIND+1
[7]    →(' ⍝'=1↑LINE)/LP,CO
[8]    '     ',LINE
[9]    ⍎LINE
[10]   →LP
[11]  CO:1↓LINE
[12]   →LP
[13]  FIN:
     ∇

     ∇ Z←N H C
[1]     ⍝ CONVERSION HEXADECIMAL => FILE DE BITS
[2]    →(1≠⎕TY C)/EN
[3]    Z←1=(-N)↑,⍉2 2 2 2⊤'0123456789ABCDE'⍳C
[4]    →0
[5]     ⍝ CONVERSION ENTIER => BITS
[6]   EN:Z←1=(N⍴2)⊤C
     ∇
```

```
    ∇ INIT
[1]     ⍝ CREATION DES VARIABLES DE LA SIMULATION
[2]     ⍝
[3]    ⎕IO←0
[4]     ⍝
[5]     ⍝ L'UGM CORRESPOND A 32 OCTETS MEMOIRE
[6]     ⍝ IMPLANTES EN /000100
[7]    UGMADDR←24 H '100'
[8]    UGMMASK←24 H 'FFFFE0'
[9]     ⍝
[10]    ⍝ LES CONSTANTES
[11]   NP←8
[12]   NR←6
[13]   CSZ←16
[14]    ⍝
[15]   RAUDES←80⍴0
[16]   RAUDES[(0 TO CSZ-1),(16 TO 15+45),64 TO 63+14]←1
[17]    ⍝
[18]    ⍝ VECTEURS D'ACCES
[19]   B_0_15←0 TO 15
[20]   B_16_31←16 TO 31
[21]   B_32_47←32 TO 47
[22]   B_48_63←48 TO 63
[23]   B_64_79←64 TO 79
[24]   ADD_H←0 TO 13
[25]   ADD_L←14 TO 22
[26]   ADD_LC←14 TO 23
[27]    ⍝
[28]    ⍝ CONTROLE DE L'UGM
[29]   ACT_BIT←7
[30]   RESET_BIT←6
[31]   VALID_BIT←4
[32]   GLOB_BIT←3
[33]   LOAD_BIT←2
[34]   SEL_TDL←1
[35]   SEL_TAP←0
[36]    ⍝
[37]    ⍝ ETAT DE L'UGM
[38]   DEFAUT←2
[39]   ERR_RDL←7
[40]   REF_INVAL←6
[41]   DEF_PAGE←11
[42]    ⍝
[43]    ⍝ GESTION DES TABLES
[44]    ⍝ LA TAP
[45]   TAP←(NP,62+CSZ)⍴0
[46]    ⍝ CHAMPS DE LA TAP
[47]   P_ID←0 TO 44
[48]   P_ADD←45 TO 58
[49]   P_VALID←59
[50]   P_MOD←60
[51]   P_MDI←59 TO 61
[52]   P_CNT←62 TO 61+CSZ
[53]    ⍝ LA TARU
[54]   TARU←(NR,68)⍴0
[55]    ⍝ CHAMPS DE LA TARU
[56]   C_IS←0 TO 15
[57]   C_IE←16 TO 23
[58]   C_PIV←24 TO 47
[59]   C_VALID←48
[60]   C_ΔSP←45 TO 47
[61]   C_NPG←24 TO 44
[62]   C_PID←0 TO 47
[63]   C_VRDA←48 TO 51
[64]   C_L←52 TO 67
[65]    ⍝
```

```
[66]    ⍝ LES TAMPONS DE COMMUNICATION
[67]   RDCRDL←16⍴0
[68]   RAU←80⍴0
[69]   S_TARU←0 TO 67
[70]   RS←16⍴0
[71]   RAD←32⍴0
[72]    ⍝
[73]    ⍝ REGISTRES INTERNES
[74]   RIO←68⍴0
[75]   RAE←32⍴0
[76]   ASSV←NP⍴0
[77]   INITGDM
[78]    ⍝
[79]    ⍝
[80]    ⍝ TRACE ET MISE AU POINT
[81]   TRACE←40⍴1
[82]   T_MSTORE←0
[83]   T_MLOAD←1
[84]   T_WSTORE←2
[85]   T_WLOAD←3
[86]   T_RDC←4
    ∇

    ∇ INITGDM
[1]     ⍝ INITIALISATION MINI-GDM
[2]     ⍝
[3]     ⍝ SIMULATION DE LA MEMOIRE
[4]    MSIZE←12
[5]     ⍝ 20 PAGES DE 1K OCTETS
[6]    CREATE_MEMORY 0 TO MSIZE-1
[7]    V_SITE←16⍴0
[8]    V_VIRT←8⍴1 1 0 0
[9]     ⍝
[10]   FREEPT←128
[11]    ⍝
[12]   VPAGES←MSIZE⍴¯1
[13]    ⍝ LA PAGE 0 UTILISEE PAR LE SYSTEME
[14]   VPAGES[0]←¯2
    ∇

    ∇ INTERRUPT N;W
[1]     ⍝ TRAITEMENT DES DEFAUTS UGM
[2]    RS[DEFAUT,N]←1
[3]    W←RDCRDL[ACT_BIT]
[4]     ⍝
[5]    'DEFAUT UGM'
[6]     ⍝ ETAT UGM
[7]    STA
[8]     ⍝ ON REND INACTIVE L'UGM
[9]    RDCRDL[ACT_BIT]←1
[10]   PROCESSEUR
[11]    ⍝
[12]    ⍝ ON RETABLIT L'ETAT ANTERIEUR
[13]   RDCRDL[ACT_BIT]←W
    ∇
```

```
     ∇ OBJ←IOTA N;INDEX
[1]    ⍝ SIMULATION DE IOTA MONADIQUE
[2]    ⍝
[3]    ⍝ 0) CREATION MEMOIRE POUR LE RESULTAT
[4]   OBJ←ALLOCATE N×2
[5]    ⍝
[6]    ⍝
[7]    ⍝ 1) ETABLISSEMENT LIEN.
[8]    ⍝
[9]   1 EDL OBJ
[10]   ⍝
[11]   ⍝
[12]   ⍝ 2) INITIALISATION
[13]  INDEX←0
[14]   ⍝
[15]   ⍝
[16]   ⍝ 3) BOUCLE D'EXECUTION
[17] LP:→(INDEX≥N)/FIN
[18]   ⍝
[19]   ⍝ ECRITURE DE LA VALEUR I
[20]  (16 H 1)WSTORE AD '100'
[21]  (16 H INDEX)WSTORE 24 H INDEX×2
[22]   ⍝
[23]   ⍝  GESTION DE LA BOUCLE
[24]  INDEX←INDEX+1
[25]  →LP
[26]   ⍝
[27]   ⍝
[28]   ⍝ 4) FIN ET LIBERATION OBJET.
[29] FIN:LIB 1
[30]   ⍝
[31]   ⍝
[32]   ⍝ SORTIE
[33]  →0
     ∇

     ∇ LIB ENT
[1]    ⍝ LIBERATION ENTREE
[2]   (0 1 1 0 0 0 0 1,8 H ENT)WSTORE AD '100'
     ∇

     ∇ Z←MLOAD ADDR;BANC
[1]    ⍝ LECTURE DE L'ADRESSE ADDR
[2]   BANC←2⊥ADDR[ADD_H]
[3]   ⍎'Z←2 ⎕TY M','0123456789ABCDEF'[16 16 16 16⊤BANC],'[2⊥ADDR[ADD_L
      ]]'
[4]   →(~TRACE[T_MLOAD])/0
[5]   '* MLOAD          ';D ADDR;' --> ';D Z
     ∇

     ∇ Z MSTORE ADDR;BANC
[1]    ⍝ ECRITURE DE Z A L'ADRESSE ADDR
[2]   BANC←2⊥ADDR[ADD_H]
[3]   ⍎'M','0123456789ABCDEF'[16 16 16 16⊤BANC],'[2⊥ADDR[ADD_L]  ]←3 ⎕T
      Y Z'
[4]   →(~TRACE[T_MSTORE])/0
[5]   '* MSTORE         ';D Z;' ==> ';D ADDR
     ∇
```

```
     ∇ PROCESSEUR
[1]    ⍝ APPEL SUR DEFAUT
[2]    →RS[DEF_PAGE]/K1
[3]    ⍝
[4]    ⍝ NON PRIS EN COMPTE PAR LA SIMULATION.
[5]    'INTERRUPTION NON PRISE EN COMPTE PAR LA SIMULATION'
[6]    *
[7]    →0
[8]    ⍝
[9]    ⍝
[10]   ⍝ TRAITEMENT DEFAUT PAGE
[11] K1:RS[]←0
[12]   ⍝ LA PAGE MANQUANTE EST DANS RAD.
[13]   DEFAUPAGE
[14]   →0
     ∇

     ∇ RESET
[1]    ⍝ FIL "RESET" DE L'UGM
[2]    RS[]←0
[3]    RDCRDL[ACT_BIT]←1
     ∇

     ∇ SELECTADDR
[1]    ⍝ SELECTION ADRESSE VIRTUELLE
[2]    ⍝ TRADUCTION DES ADRESSES
[3]    ΔPAGE←(2⊥ADDR[ADD_LC])+2⊥RIO[C_ΔSP],7⍴0
[4]    NPAGE←(2⊥RIO[C_NPG])+(2⊥ADDR[ADD_H])
[5]    ⍝ CREATION IDENTIFICATION DE LA PAGE
[6]    IDPAGE←RIO[C_IS,C_IE],1=(21⍴2)⊤NPAGE
[7]    '* ';D IDPAGE;'   ';NPAGE
[8]    ⍝ RECHERCHE ASOCIATIVE
[9]  SEL:→(~∨/(ASSV[]←TAP[;P_ID]∧.=IDPAGE)∧TAP[;P_VALID])/ERR
[10]   ⍝ LA PAGE EST PRESENTE
[11]   PNB←ASSV⍳1
[12]   ADDR1←TAP[PNB;P_ADD],1=(10⍴2)⊤ΔPAGE
[13]   ⍝ M.A.J. DES COMPTEURS
[14]   TAP[;P_CNT]←1=⍉(CSZ⍴2)⊤1+2⊥⍉TAP[;P_CNT]
[15]   TAP[PNB;P_CNT]←0
[16]   →0
[17]   ⍝ DEFAUT PAGE
[18] ERR:INTERRUPT DEF_PAGE
[19]   →(~RS[DEFAUT])/SEL
[20]   →0
     ∇

     ∇ SETRDCRDL Z;C;RDL
[1]    RDCRDL←Z
[2]    RDL←2⊥RDCRDL[8 TO 15]
[3]    RS[]←0
[4]    ⍝ TRACE EVENTUELLE
[5]    →(~TRACE[T_RDC])/NOTR
[6]    '* SETRDCRDL  ',(D RDCRDL),'  ','⌧ '[RDCRDL[ACT_BIT]]
[7]  NOTR:
[8]    ⍝ EST-CE UNE SELECTION ?
[9]    →(~RDCRDL[ACT_BIT])/SEL
[10]   ⍝ EST-CE UN DUMP ?
[11]   →(RDCRDL[SEL_TAP,SEL_TDL])/K1,K1
[12]   ⍝ EST-CE UN RESET ?
[13]   →(RDCRDL[RESET_BIT])/K3
[14]   ⍝ ERREUR !
[15]   INTERRUPT ERR_COM
[16]   →0
[17]   ⍝
```

```
[18]    ⍝  RESET
[19]  K3:RESET
[20]    ⍝
[21]    ⍝ FIN: RESET BIT DEFAUT
[22]  OK:RS[DEFAUT]←0
[23]   →0
[24]    ⍝
[25]    ⍝  DUMP D'UNE TABLE
[26]  K1:C←+/RDCRDL[SEL_TAP,SEL_TDL,LOAD_BIT,GLOB_BIT]/0 4 2 1
[27]   '* SETRDCRDL  : ';C
[28]   →(L1,L2,L3,L4,L5,L6,L7,L8)[C]
[29]    ⍝
[30]    ⍝ STORE/LOCAL/TAP
[31]  L1:RAU[]←RAUDES\TAP[RDL;]
[32]   →0
[33]    ⍝
[34]    ⍝ STORE/GLOB/TAP
[35]  L2:(8↓RAD)TSTORE TAP
[36]   →0
[37]    ⍝
[38]    ⍝ LOAD/LOCAL/TAP
[39]  L3:TAP[RDL;P_ID]←RAU[0 TO 44]
[40]   TAP[RDL;P_ADD]←RAU[50 TO 63]
[41]   TAP[RDL;P_CNT]←0
[42]   TAP[RDL;P_MDI]←0
[43]   TAP[RDL;P_VALID]←RDCRDL[VALID_BIT]
[44]   →0
[45]    ⍝
[46]    ⍝ LOAD/GLOB/TAP
[47]  L4:TAP[;]←RAD TLOAD⍴TAP
[48]   →0
[49]    ⍝
[50]    ⍝ STORE/LOCAL/TARU
[51]  L5:RAU[S_TARU]←TARU[RDL;]
[52]   →0
[53]    ⍝
[54]    ⍝ STORE/GLOB/TARU
[55]  L6:
[56]   →0
[57]    ⍝
[58]    ⍝ LOAD/LOCAL/TARU
[59]  L7:RIO←RAU[S_TARU]
[60]   RIO[C_VALID]←RDCRDL[VALID_BIT]
[61]   TARU[RDL;]←RIO
[62]   →0
[63]    ⍝
[64]    ⍝ LOAD/GLOB/TARU
[65]  L8:TARU[;]←RAD TLOAD NR,68
[66]   →0
[67]    ⍝
[68]    ⍝  SELECTION D'UNE REFER. LOCAL
[69]  SEL:→(RDL≥NR)/ERRH
[70]   →(~TARU[RDL;C_VALID])/ERRH
[71]   RIO[]←TARU[RDL;]
[72]   →0
[73]    ⍝
[74]    ⍝ ERREUR DE SELECTION
[75]  ERRH:INTERRUPT ERR_RDL
[76]   RDCRDL[ACT_BIT]←1
[77]   →0
    ∇

    ∇ STA
[1]    'RDCRDL/';D RDCRDL;'    RS/';D RS;'    RAU/';D RAU;'    RAD/';D RAD
[2]    '                        RIO/';D RIO;'       RAE/';D RAE
    ∇
```

```
     ∇ Z←ADDR TLOAD SIZ;S;I;N
[1]    ⍝ TABLE LOAD
[2]    N←(×/S←1 16×⌈SIZ÷1 16)÷16
[3]    Z←⍳0
[4]    I←0
[5]   LP:→(I≥N)/ELP
[6]    Z←Z,MLOAD(24⍴2)⊤(I×2)+2⊥ADDR
[7]    I←I+1
[8]    →LP
[9]   ELP:Z←1=(SIZ-S)↓S⍴Z
     ∇

     ∇ Z←A TO B
[1]    Z←A+⍳1+B-A
     ∇

     ∇ TROFF X
[1]    TRACE[X]←0
     ∇

     ∇ TRON X
[1]    TRACE[X]←1
     ∇

     ∇ TST
[1]    ⍝ TEST DE L'UGM
[2]    ⍝
[3]    ⍝ INITIALISATION UGM
[4]    RESET
[5]    ⍝
[6]    ⍝ DEFINITION D'UN OBJET
[7]    OBJ←IOTA 5
     ∇

     ∇ TST1
[1]    ⍝ RESET UGM
[2]    RESET
[3]    ⍝ MISE EN PLACE DE CERTAINES PAGES
[4]    0 EDP '0000330000100004'
[5]    1 EDP '0000330000180007'
[6]    2 EDP '0000330000080005'
[7]    3 EDP '0000330000000003'
[8]    4 EDP '0000330000380006'
[9]    ⍝ MISE EN PLACE DE LA RDU
[10]   2 EDL '0000330000000000400'
[11]   1 EDL '00003300003D001001000'
[12]   STA
[13]   WLOAD AD '25'
[14]   1 0 1 0 WSTORE AD '25'
[15]   1 0 WSTORE AD '100'
[16]   WLOAD AD '25'
[17]   ⍝ FIN
     ∇

     ∇ TST2
[1]    1 0 WSTORE AD '100'
[2]    (16⍴1 0 1)WSTORE AD '10'
     ∇

     ∇ TST3
[1]    0 1 WSTORE AD '100'
[2]    WLOAD AD '22'
     ∇
```

```
     ∇ ADDR TSTORE TAB;S;I;N;Z
[1]    ⍝ TABLE STORE
[2]    N←(×/S←1 16×⌈(⍴TAB)÷1 16)÷16
[3]    Z←,(-S)↑TAB
[4]    I←0
[5]   LP:→(I≥N)/ELP
[6]    Z[(16×I)+⍳16]←MSTORE(24⍴2)⊤(I×2)+2⊥ADDR
[7]    I←I+1
[8]    →LP
[9]   ELP:
     ∇

     ∇ VRESET
[1]    RDCRDL[ACT_BIT]←1
     ∇

     ∇ Z←WLOAD ADDR;LOCADR
[1]    ⍝ LECTURE D'UN MOT MEMOIRE
[2]    Z←16⍴0
[3]    ⍝ UGM ACTIVE ?
[4]    →(~RDCRDL[ACT_BIT])/VIRT
[5]    ⍝ UGM CONCERNEE ?
[6]    →(UGMADDR∨.≠UGMMASK∧ADDR)/NONCE
[7]    ⍝ OUI. ON RECUPERE L'ADRESSE LOCALE
[8]    LOCADR←2⊥⍉5↑ADDR
[9]    ⍝ AIGUILLAGE
[10]   →(LOCADR=0 2 4 6 8 10 14)/CTL,RU1,RU2,RU3,RU4,RU5,RST
[11]   ⍝ REFERENCE INVALIDE
[12]   RS[DEFAUT,REF_INVAL]←1
[13]   →0
[14]   ⍝
[15]   ⍝ REFERENCE MEMOIRE NORMALE
[16] NONCE:Z←MLOAD ADDR
[17]   →0
[18]   ⍝
[19]   ⍝ LECTURE DU REGISTRE CTL-RDL
[20] CTL:Z←(8⍴0),RDCRDL[B_8_15]
[21]   →0
[22] RST:Z←RS
[23]   →0
[24] RU1:Z←RAU[B_0_15]
[25]   →0
[26] RU2:Z←RAU[B_16_31]
[27]   →0
[28] RU3:Z←RAU[B_32_47]
[29]   →0
[30] RU4:Z←RAU[B_48_63]
[31]   →0
[32] RU5:Z←RAU[B_64_79]
[33]   →0
[34] VIRT:
[35]   →(~TRACE[T_WLOAD])/NOTR
[36]   '* WLOAD         ';D ADDR;'     ';D RDCRDL
[37] NOTR:
[38]   SELECTADDR
[39]   →(RS[DEFAUT])/END
[40]   ⍝ ON PEUT FAIRE LE LOAD
[41]   Z←MLOAD ADDR1
[42] END:VRESET
[43]   →0
     ∇
```

```
     ∇ Z WSTORE ADDR;LOCADR
[1]    ⍝ ECRITURE D'UN MOT MEMOIRE
[2]    Z←¯16↑1=Z
[3]    ⍝ UGM ACTIVE ?
[4]    →(~RDCRDL[ACT_BIT])/VIRT
[5]    ⍝ UGM CONCERNEE ?
[6]    →(UGMADDR∨.≠UGMMASK∧ADDR)/NONCE
[7]    ⍝ OUI. ON RECUPERE L'ADRESSE LOCALE
[8]    LOCADR←2⊥¯5↑ADDR
[9]    ⍝ AIGUILLAGE
[10]   →(LOCADR=0 2 4 6 8 10)/CTL,RU1,RU2,RU3,RU4,RU5
[11]   →(LOCADR=16 18)/RAD1,RAD2
[12]   ⍝ REFERENCE INVALIDE
[13]   RS[DEFAUT,REF_INVAL]←1
[14]   →0
[15]   ⍝
[16]   ⍝ REFERENCE MEMOIRE NORMALE
[17] NONCE:Z MSTORE ADDR
[18]   →0
[19] CTL:SETRDCRDL Z
[20]   →0
[21] RU1:RAU[B_0_15]←Z
[22]   →0
[23] RU2:RAU[B_16_31]←Z
[24]   →0
[25] RU3:RAU[B_32_47]←Z
[26]   →0
[27] RU4:RAU[B_48_63]←Z
[28]   →0
[29] RU5:RAU[B_64_79]←Z
[30]   →0
[31] RAD1:RAD[B_0_15]←Z
[32]   →0
[33] RAD2:RAD[B_16_31]←Z
[34]   →0
[35] VIRT:
[36]   →(~TRACE[T_WSTORE])/NOTR
[37]   '* WSTORE       ';D ADDR;'     ';D RDCRDL;' <== ';D Z
[38] NOTR:
[39]   SELECTADDR
[40]   →(RS[DEFAUT])/END
[41]   ⍝ ON PEUT FAIRE LE STORE
[42]   Z MSTORE ADDR1
[43]   ⍝ POSITIONNE BIT MODIFIE.
[44]   TAP[PNB;P_MOD]←1
[45] END:VRESET
[46]   →0
     ∇

     ∇ Z←X C
[1]    ⍝ CONVERSION HEXADECIMAL => FILE DE BITS
[2]    Z←1=,⍉2 2 2 2⊤'0123456789ABCDE'⍳C
     ∇
```

dernière page de la thèse

AUTORISATION DE SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974,

VU les rapports de présentation de MM. R. TISSERAND, J.J. GIRARDOT.

M. Ulysse SAKELLARIDIS

est autorisé à présenter une thèse en soutenance pour l'obtention
3ème cycle
du diplôme de DOCTEUR-~~INGENIEUR~~, spécialité "Systèmes et Réseaux Informatiques"

Fait à Saint-Etienne, le 29 Novembre 1983

LePrésident de l'INPG

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,

Le Directeur de l'EMSE,

M. MERMET
INGÉNIEUR GÉNÉRAL DES MINES
DIRECTEUR
ECOLE NATIONALE SUPÉRIEURE DES
MINES DE SAINT-ETIENNE

CONCEPTION D'UN POSTE DE TRAVAIL APL ET DE SON UNITE DE GESTION MEMOIRE

Ulysse SAKELLARIDIS

Mots-clés : APL, Lisp, UNIX, poste de travail, mémoire virtuelle, intelligence artificielle.

RESUME :

L'apparition des postes de travail autonomes scientifiques a beaucoup influencé l'évolution des environnements logiciels de programmation. Dans ce rapport, une étude comparative des systèmes matériels existants et des deux principaux environnements de programmation ("classique" et orienté vers l'intelligence artificielle) permet de situer cette évolution.

Par la suite, le travail porte sur la conception d'un troisième type d'environnement, fondé autour d'une version étendue du langage APL. Il est démontré que le choix d'APL permet de réaliser des environnements possédant les avantages de ceux déjà existants, sans forcément hériter de leurs inconvénients.

Le problème de la gestion d'un espace virtuel et unique d'objets APL est traité dans la deuxième partie de ce rapport qui aboutit à la conception d'une unité spécialisée, de telle sorte que celle-ci puisse être réalisable exclusivement en matériel.

Enfin, une simulation du fonctionnement de cette unité permet de valider sa conception.