Machine PASC-HLL : définition d'une architecture pipe-line pour une unité centrale adaptée au langage PASCAL

Jean-Pierre Schoellkopf

THESE

présentée à

Institut National Polytechnique de Grenoble

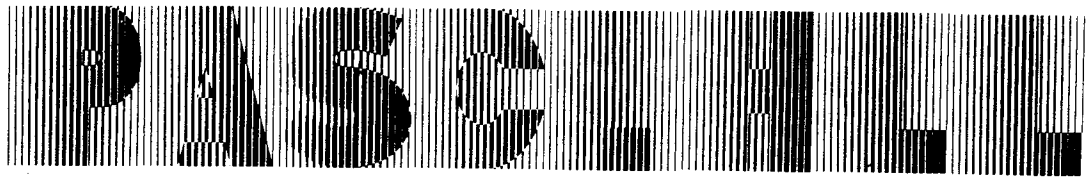
pour obtenir le grade de

Docteur de 3ème cycle

Génie Informatique

par

Jean-Pierre SCHOELLKOPF



MACHINE PASC-HLL : DEFINITION D'UNE ARCHITECTURE PIPE-LINE POUR UNE UNITE CENTRALE ADAPTEE AU LANGAGE PASCAL.

Thèse soutenue le 28 juin 1977 devant la Commission d'Examen :

Président : L. BOLLIET

Examinateurs : F. ANCEAU
C. GIRAULT
Ph. JORRAND
C. OTRAGE

THESE

présentée à

Institut National Polytechnique de Grenoble

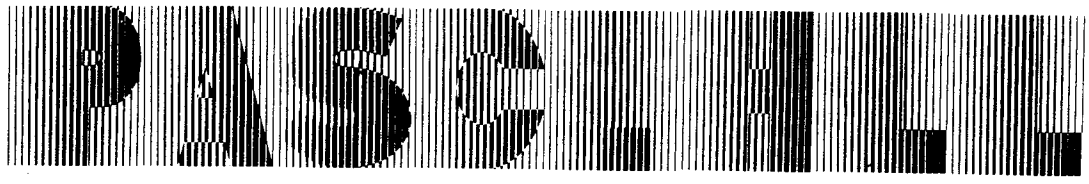
pour obtenir le grade de

Docteur de 3ème cycle

Génie Informatique

par

Jean-Pierre SCHOELLKOPF



MACHINE PASC-HLL : DEFINITION D'UNE ARCHITECTURE PIPE-LINE POUR UNE UNITE CENTRALE ADAPTEE AU LANGAGE PASCAL.

Thèse soutenue le 28 juin 1977 devant la Commission d'Examen :

Président : L. BOLLIET

Examinateurs : F. ANCEAU
C. GIRAULT
Ph. JORRAND
C. OTRAGE

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE — ANNEE 1976/1977

Président: Monsieur Philippe TRAYNARD
Vice-Présidents: M. Pierre Jean LAURENT
M. René PAUTHENET

PROFESSEURS TITULAIRES

MM. BENOIT Jean	Radioélectricité
BESSON Jean	Electrochimie
BLOCH Daniel	Physique du solide
BONNETAIN Lucien	Chimie minérale
BONNIER Etienne	Electrochimie et électrométallurgie
BRISSONNEAU Pierre	Physique du solide
BUYLE BODIN Maurice	Electronique
COUMES André	Radioélectricité
DURAND Francis	Métallurgie
FELICI Noel	Electrostatique
FOULARD Claude	Automatique
LESPINARD Georges	Mécanique
MOREAU René	Mécanique
PARIAUD Jean Charles	Chimie physique
PAUTHENET René	Physique du solide
PERRET René	Servomécanismes
POLOUJADOFF Michel	Electrotechnique
VEILLON Gérard	Informatique fondamentale et appliquée

PROFESSEURS SANS CHAIRE

MM. BLIMAN Samuel	Electronique
BOUVARD Maurice	Génie mécanique
COHEN Joseph	Electrotechnique
LACOUME Jean Louis	Géophysique
LANCIA Roland	Electronique
ROBERT François	Analyse numérique
ROBERT André	Chimie papetière
ZADWORNY François	Electronique

MAITRES DE CONFERENCES

MM. ANCEAU François	Mathématiques appliquées
CHARTIER Germain	Electronique
GUYOT Pierre	Chimie minérale
IVANES Marcel	Electrotechnique
JOUBERT Jean Claude	Physique du solide
LESIEUR Marcel	Mécanique
MORET Roger	Electrotechnique nucléaire
PIAU Jean Michel	Mécanique
PIERRARD Jean Marie	Mécanique
SABONNARDIERE Jean Claude	Informatique fondamentale et appliquée
MME SAUCIER Gabrièle	Informatique fondamentale et appliquée

CHERCHEURS DU CNRS (Directeur et Maîtres de recherche)

M. FRUCHART Robert	Directeur de recherche
MM. ANSARA Ibrahim	Maître de recherche
CARRE René	Maître de recherche
DRIOLE Jean	Maître de recherche
LANDAU Ioan Doré	Maître de recherche
MATHIEU Jean Claude	Maître de recherche
MUNIER Jacques	Maître de recherche

Pourquoi faire simple
quand on peut faire compliqué ?

J.ROUXEL, "les Shadoks"

"Si c'eust esté pour rechercher
la faveur du monde, je me fusse
mieux paré et me présenterois
en une marche estudiée..."

de Montaigne,

ce premier de Mars mille cinq cens quatre ving.

"...ce n'est pas raison que tu
employes ton loisir en un subject
si frivole et si vain."

Je tiens à remercier,

Monsieur BOLLIET, professeur à l'USMG, qui a bien voulu me faire l'honneur de présider le jury de cette thèse,

Monsieur GIRAULT, professeur à PARIS VI, qui a bien voulu juger ce travail,

Monsieur JORRAND, maître de recherche à l'ENSIMAG, qui nous a prodigué ses conseils pour améliorer la présentation des mécanismes complexes d'exécutions parallèles,

Monsieur OTRAGE, ingénieur de recherche, qui a suggéré cette étude et nous a conseillé lors de nos premiers balbutiements,

Monsieur ANCEAU, maître de conférences à l'ENSIMAG, qui a su nous encourager tout au long de ce travail en nous prodiguant ses conseils, et qui a tout mis en oeuvre pour qu'une réalisation voit le jour,

Monsieur BAILLE, mon coéquipier pour cette volumineuse étude, qui a participé à tous les instants de ce travail et contribué par ses critiques et son aide matérielle à l'élaboration de ce document,

Tous les membres de l'Equipe de recherches en architecture d'ordinateurs, qui m'ont encouragé de leurs conseils dans une ambiance de solidarité constructive,

Madame DIAZ, qui a assuré la dactylographie de ce texte dans des conditions difficiles avec beaucoup de patience et une grande compétence,

et enfin le Service de reprographie du CICG qui a assuré le tirage de ce document avec le soin habituel.

Jean-Pierre SCHOELLKOPF

A mon épouse Christine,
quelque peu délaissée pour PASC-HLL,

et à Alain GREBERT, décédé en septembre 1974,
qui est à l'origine des idées développées dans cette étude.

TABLE DES MATIERES

AVANT-PROPOS

Ce projet de recherche a pris corps au fil des ans, commençant par un D.E.A. en octobre 1972*. Il se concrétise aujourd'hui.

Il lui manquait cependant un nom qui le caractérise sans ambigüité et porte en lui-même une certaine sémantique, sans toutefois faire de tort à l'honorable Blaise PASCAL (peut-être est-il mécontent de l'utilisation qui est faite de son nom par les informaticiens).

Nous avons donc fait appel à la langue anglaise pour décrire notre projet comme étant

"a Pipelined Architecture bit Slice Computer for High Level Language"

ce qui donne miraculeusement PASC-HLL, rappelant le langage PASCAL et introduisant le symbole HLL qui met en évidence l'aspect machine-langage de haut niveau.

Ainsi, nous avons le sentiment d'être en règle avec Blaise PASCAL, même si cela pose des problèmes de prononciation à nos contemporains (il existe bien d'autres mnémoniques dans le langage des informaticiens, oh combien moins poétiques et bien souvent dépourvus de sémantique).

Il est également nécessaire de préciser dans quel contexte cette présentation se situe. C'est le deuxième volet d'une série de trois thèses présentant trois aspects distincts d'un vaste projet: la thèse de R. FORTIER (Octobre 1974 [1]) a montré une approche de définition d'un langage et d'une architecture pour une machine adaptée au langage PASCAL, la seconde (celle-ci) concrétise cette approche par la description d'une architecture pipeline dont la réalisation (en cours) sera décrite dans la future thèse de G. BAILLE, en préparation pour la fin de l'année 1977.

Cette étude est donc le fruit d'un travail d'équipe, qui se concrétise aujourd'hui par la réalisation physique, dans une Université française, d'un ordinateur de grande puissance: ce projet ambitieux, soutenu financièrement par l'IRIA et le

*Sur une idée de C.OTRAGE.

CNRS (7ème Plan), a pu se poursuivre grâce à la ténacité de son responsable scientifique* dans le cadre offert par l'Equipe de Recherche en Architecture d'ordinateurs de l'ENSIMAG, qui nous a toujours encouragé de son soutien et prodigué ses conseils.

En quoi le projet PASC-HLL apporte-t-il des notions nouvelles pour la conception des systèmes d'exploitation? Par sa vocation à exécuter d'une manière performante des programmes manipulant des données, elle introduit clairement la notion de processeur de traitement par opposition à un processeur de gestion d'un système d'exploitation. Dans un système classique, ces deux processeurs sont multiplexés sur un seul processeur physique, qui exécute alternativement des programmes de traitement et des programmes de gestion du système d'exploitation, évidemment écrits dans le même langage machine. Un processeur peut-il être adapté à ces deux fonctions à la fois? On nous répondra que l'on peut réaliser n'importe quelle fonction avec un ordinateur mais au prix de quel volume de programmes! Leur taille ne reflète-t-elle pas l'inaptitude du langage-machine à exprimer les fonctions à réaliser?

Quelques études visent à améliorer le fonctionnement global des systèmes, en multipliant par exemple le nombre des Unités Centrales. Ne vaudrait-il pas mieux remettre en cause la nature des Unités Centrales, pour les adapter à la fonction que l'on attend d'elles, plutôt que d'en connecter plusieurs, en se créant artificiellement des problèmes de voisinage? Cette étude propose une approche radicalement opposée et espère susciter des réflexions sur ce thème de recherche.

Nous présentons également un langage-machine, dont l'étude a été abordée dans la thèse FORTIER [1]. Ce langage, voisin de celui des machines BURROUGHS [2], est de type post-fixé . Cette notion a cependant été généralisée, en introduisant des opérations de contrôle du programme, et des opérateurs spéciaux de manipulation de données. Sa définition est issue du langage-source (PASCAL) et de mesures qui ont été réalisées sur des programmes-type. Les travaux de GREBERT [5] et de WORTMAN [6] ont joué un rôle d'initiateur en donnant des recettes intéressantes. Nous avons appliqué ces recettes, en les développant, et nous avons surtout pris

*François ANCEAU

conscience des liens étroits qui existent entre un langage et son interpréteur: rien ne sert de définir un "beau" langage machine, très proche du langage de haut niveau, si son interprétation est problématique ou doit être programmée. Nous pensons en effet pouvoir affirmer que la définition d'un langage, quel que soit son "niveau" dans la hiérarchie des langages doit tenir compte des deux couches entre lesquelles il se situe: la couche supérieure qui est celle de l'utilisation de ce langage pour écrire des programmes, et la couche inférieure qui est l'interpréteur des instructions qui constituent ces programmes. Comme plusieurs couches existent forcément dans tout ordinateur, cette conviction conduit à dire que toutes ces couches doivent toutes être conçues dans une étude unique. A titre de contre-exemple, imaginons que l'étude de PASC-HLL se soit arrêtée à la définition du langage-machine, pour des raisons de crédits ou pour dautres raisons fort courantes comme la peur de remettre en cause le style de langage des ordinateurs. Il aurait alors pu être envisagé (ou imposé) d'étudier l'interprétation de langage PASC-HLL par un miniordinateur par exemple. Notre étude aurait alors perdu la plus grande partie de son intérêt, et nous aurions peut-être même pu montrer qu'il aurait mieux valu étudier la compilation de PASCAL pour le langage de ce miniordinateur, plutôt que de microprogrammer un interpréteur avec un langage de microprogrammation inadapté.

Notre étude ayant pu, et nous en remercions toux ceux qui l'ont soutenue, dépasser le stade de la "machine-papier-simulée-par-microprogrammation-sur-une-machine-de-base", nous avons abordé la phase de conception d'une architecture spécialisée, sans perdre de vue la définition du langage qui a été très souvent remise en cause, et en tenant compte de la réalisation future et sous-jacente des processeurs qui voyaient le jour dans notre esprit.

L'idée originale de l'architecture proposée découle d'une décomposition naturelle du fonctionnement d'une Unité Centrale: elle accède à des instructions, elle accède à des données, elle exécute des opérateurs. Il suffisait donc de trouver un mécanisme, si possible simple, qui permette de confier à trois processeurs spécialisés l'exécution de ces trois fonctions qui apparaissent naturelllement.

Le gain théorique de performance est évident. Le gain réel est encore inconnu et nous expliquerons pourquoi. Il semblait cependant intéressant d'étudier complétement les possibilités d'une telle décomposition, qui n'est pas déraisonnable si l'on en juge à l'utilisation qui en est faite dans quelques Unités Centrales très puissantes déjà commercialisées (IBM 3033).

De plus, cette architecture met encore en évidence la notion de processeurs fonctionnels qui conduit à la réalisation de machines de petite taille, ultra-spécialisées dans la fonction qu'on leur demande de réaliser. Elle permet d'autre part d'appliquer les méthodes de conception descendante pour la définition de chacun des processeurs.

Malgré leur petite taille due à leur spécialisation, les processeurs de PASC-HLL sont chacun une véritable Unité Centrale qui interprète un langage spécialisé. Leur réalisation n'aurait pu être envisagée sans l'apparition providentielle des "macrocomposants" appelés à tort "microprocesseurs en tranches" [7].
En effet, ces composants électroniques regroupent dans un seul boîtier plusieurs composants classiques (MSI) et permettent de construire aisément des structures microprogrammées: la démarche descendante suggère la définition d'un langage de microprogrammation spécialisé en fonction de l'écriture du programme d'interprétation du langage-machine. Cette définition a été faite pour chacun des processeurs en tenant compte du choix des macrocomposants: ils introduisent une notion de cycle, correspondant à l'exécution d'une microinstruction. Il suffit donc de connaître ce qui est "exécutable en un cycle" pour définir le langage de microprogrammation.

La réalisation électronique représente l'étape suivante de la démarche: elle sera présentée en détail dans la troisième thèse (G.BAILLE, fin 1977)[8].

- La <u>première partie</u> de ce document présente les aspects généraux de PASC-HLL: les "aspects système d'exploitation", après avoir situé PASC-HLL dans l'environnement d'un système d'exploitation, décrivent l'organisation des données manipulées par la machine (son espace d'adressage) et les problèmes d'entrée/sortie qui illustrent la répartition du traitement des éléments de fichiers par un

compromis entre la programmation de certaines procédures et la microprogrammation des fonctions de conversion entre une représentation externe (alphanumérique) et interne (binaire). Enfin, dans un deuxième chapitre, les principales caractéristiques du langage-machine et l'architecture générale de PASC-HLL sont successivement présentées.

- La deuxième partie décrit successivement les cinq processeurs qui constituent PASC-HLL. Pour chacun d'eux, on situe son rôle dans l'architecture pipeline, on présente les problèmes de synchronisation et l'algorithme général qu'il doit réaliser. Ceci nous conduit à proposer des microprogrammes, écrits d'une manière symbolique, pour décrire les fonctions à réaliser. Ces microprogrammes "formels" seront à leur tour étudiés afin de définir des machines électroniques spécialisées dans leur interprétation (la machine électronique interprète le langage de microprogrammation).

L'ensemble des trois thèses relatives à PASC-HLL couvre ainsi une conception globale d'ordinateur, de l'utilisateur du langage de haut niveau au fonctionnement des composants électroniques. De plus, cette étude théorique se concrétise aujourd'hui par la réalisation physique d'un prototype qui sera, nous l'espérons, en mesure de montrer si les mécanismes proposés présentent l'intérêt que nous leur supposons.

En annexe, un essai de formalisation de la méthode de conception descendante est présenté.

PREMIERE PARTIE

CHAPITRE 1 : LES ASPECTS "SYSTEME" DE "PASC-HLL"

CHAPITRE 2 : PRESENTATION GENERALE DE L'ARCHITECTURE DE "PASC-HLL"

CHAPITRE 1

A - INTRODUCTION: Insertion de PASC-HLL dans un "système"

B - L'ESPACE D'ADRESSAGE DE LA MACHINE

I - Structuration de l'espace d'adressage
II - Adressage des quatre zones de l'espace virtuel
III - Organisation des zones CODE et EXTERNE
IV - Organisation de la zone CONTEXTE
V - Organisation de la zone DYNAMIQUE

C - LES ENTREES/SORTIES

I - Structure des informations
II - Exécution des opérations d'entrée/sortie
III - Les entrées/sorties évoluées
IV - Conclusion

A - INTRODUCTION

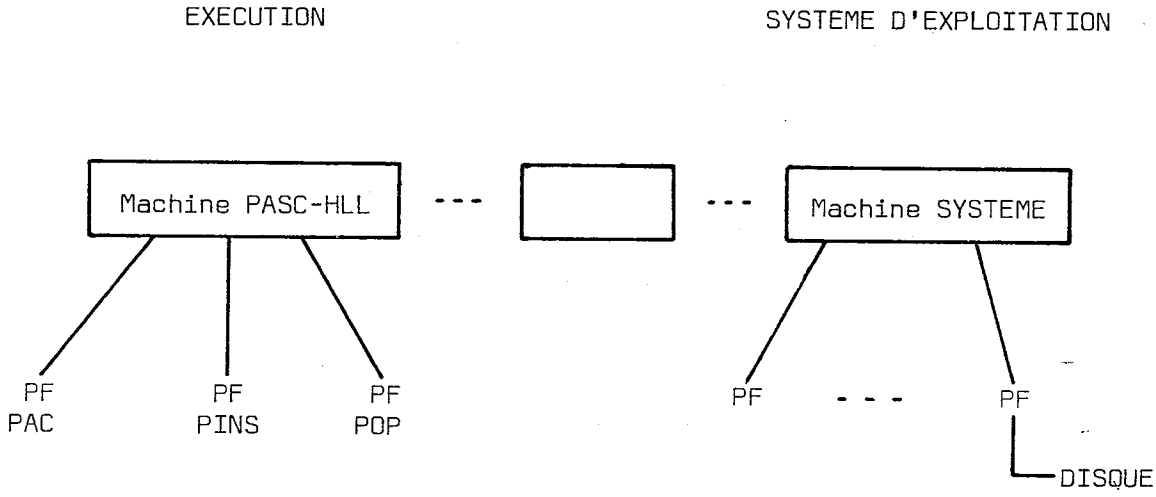
La machine PASC-HLL a été définie comme un processeur de traitement spécialisé dans l'exécution de programmes manipulant des données situées dans une Mémoire Centrale. Cette définition implique qu'elle n'est pas concernée par les opération d'entrée/sortie, au sens du transfert d'informations entre un périphérique et la Mémoire Centrale, ni par la gestion globale des programmes soumis au système d'exploitation par les utilisateurs.

Cette approche illustre la notion de processeurs fonctionnels que l'on retrouve dans[4], qui fait apparaître une répartition des fonctions à réaliser: dans notre cas, PASC-HLL réalise la fonction d'exécution des programmes et les fonctions "système d'exploitation" sont laissées à la charge d'une autre Unité centrale, que nous appellerons la Machine Hôte exécutant le Système Hôte, ou tout simplement Machine-Système.

Cette étude d'un processeur spécialisé dans le traitement fait d'ailleurs apparaître les spécifications d'un processeur spécialisé dans l'exécution d'un système d'exploitation, dont le travail se réduirait :

- à la gestion des programmes dans un système de type "traitement par lot",
- à la gestion de la Mémoire Centrale,
- au dialogue avec les organes d'entrée/sortie (périphériques).

Cette décomposition fonctionnelle ne doit pas être restreinte à deux processeurs (PASC-HLL et la Machine-Système-Hôte), et une approche hiérarchisée de la décomposition doit être envisagée: la machine PASC-HLL elle-même en est un exemple, avec ses trois processeurs fonctionnels de traitement des instructions (PINS), d'accès aux opérandes (PAC) et d'exécution des opérateurs (POP). En ce qui concerne la Machine Système, plusieurs études sont en cours, qui proposent par exemple un processeur de gestion de fichiers [9], et répartissent les fonctions systèmes entre plusieurs processeurs spécialisés [3].



I.- MULTIPROGRAMMATION FONCTIONNELLE

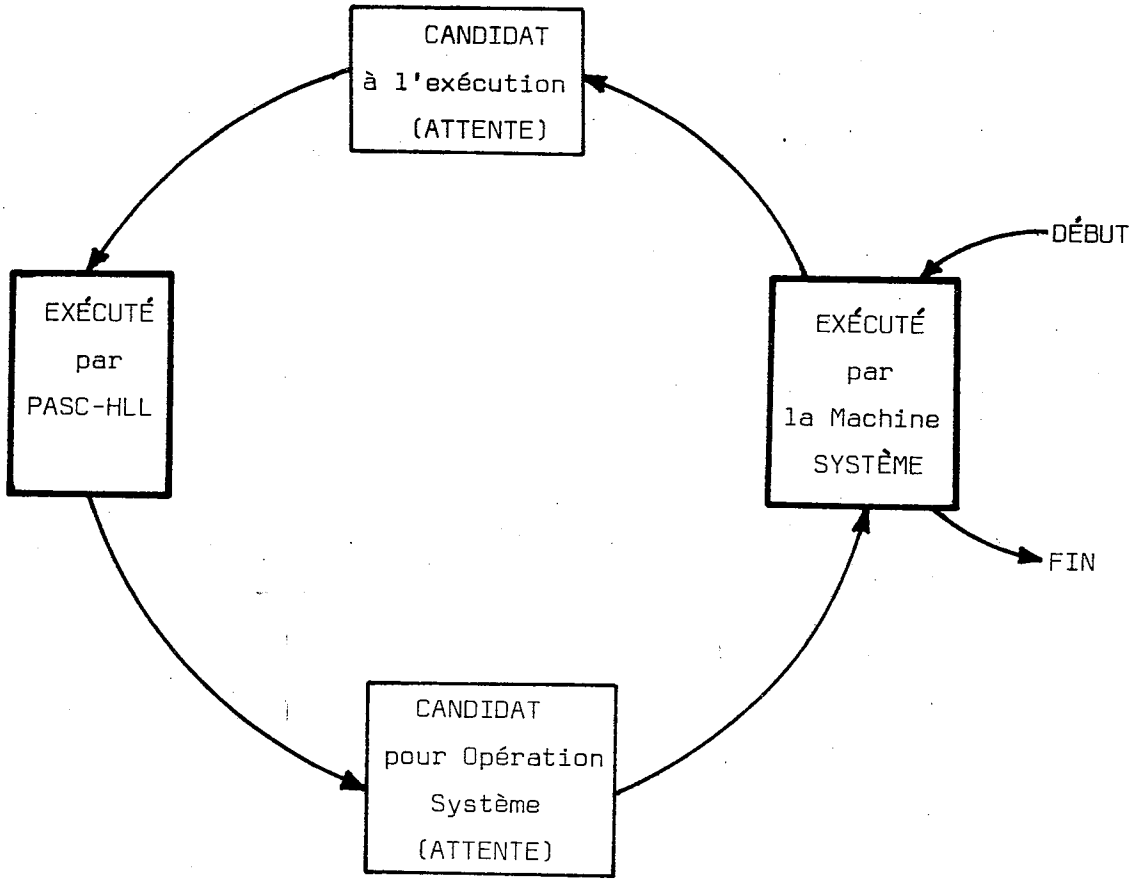
La répartition fonctionnelle choisie implique un mode de fonctionnement particulier que nous appellerons multiprogrammation fonctionnelle caractérisée par le fait que plusieurs programmes sont présents dans le système global à un instant donné, l'un requérant la fonction d'exécution (PASC-HLL), l'autre une fonction système exécutée par la Machine Système.

Suivons la vie d'un programme dans le système global:

- Son initialisation est du ressort de la Machine-Système qui réalise les opérations d'Edition de Liens entre les modules et les fichiers. Elle se termine lorsque tout est prêt pour l'exécution. Le programme devient alors "candidat à l'exécution".

- Tout programme "candidat à l'exécution" peut être fourni à la machine PASC-HLL. Cette dernière exécute les instructions qui le composent et arrête cette exécution lorsqu'elle détecte une instruction-système qui n'est pas de sa compétence. Le programme devient alors "candidat pour une Opération-Système".

La figure présente les différents états possibles pour un programme.



II - REALISATION DU PARTAGE DES PROGRAMMES

Le fonctionnement en multiprogrammation fonctionnelle suppose un "transfert" des programmes entre la fonction d'exécution et la fonction "système". La réalisation physique d'un tel transfert serait trop lente et poserait de toutes manières des problèmes de conflits d'accès aux informations situées dans une mémoire partagée.

La solution proposée repose sur l'utilisation d'une Mémoire Centrale unique partagée par les deux processeurs (PASC-HLL et la Machine-Système). Elle suppose, dans les conditions idéales:

- que la Mémoire Centrale soit physiquement composée de plusieurs blocs indépendants,
- que les deux processeurs puissent physiquement accéder à tous les blocs.

Dans ces conditions, toutes les données relatives à l'exécution d'un programme se trouvent dans le même bloc physique. Chaque bloc physique est alternativement accédé par PASC-HLL (lorsqu'il contient le programme en exécution) et par la Machine-Système (lorsqu'il contient un programme pour lequel une opération système est exécutée).

Cette organisation de la Mémoire Centrale suppose de plus l'existence d'un mécanisme de communication entre les deux unités centrales:

- la Machine Centrale doit fournir à PASC-HLL l'adresse (numéro) du descripteur du programme candidat à l'exécution ;

- PASC-HLL doit avertir la Machine-Système que le programme contenu dans le bloc physique qui lui était alloué requiert l'exécution d'une opération système, et demande ainsi l'allocation d'un autre bloc physique contenant le candidat suivant à l'exécution.

Exemple de réalisation:

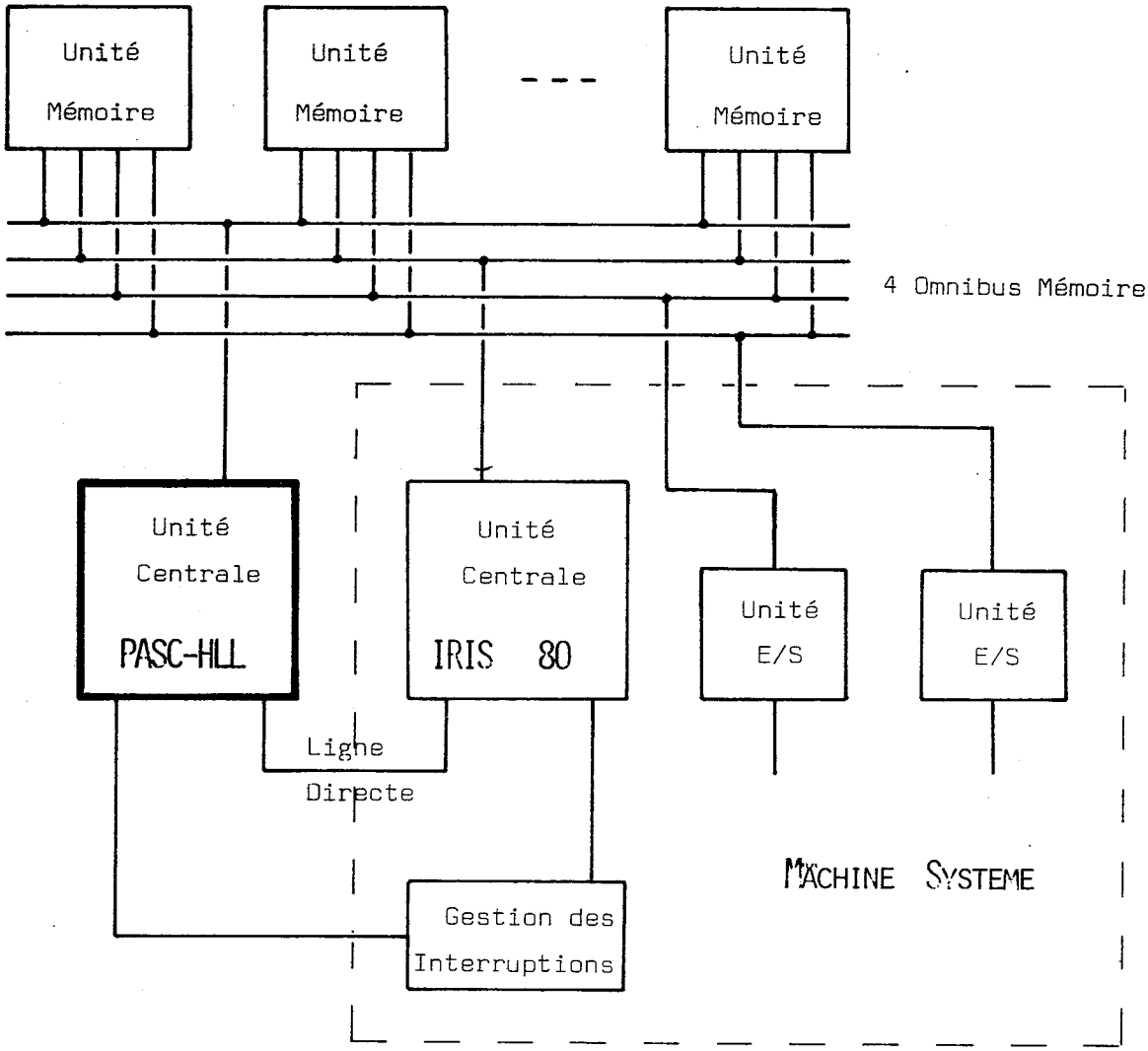
Un tel système est à première vue réalisable à titre d'exemple, sur un ordinateur IRIS 80.

Nous nous limitons ici à donner une approche du problème de connexion à un tel ordinateur, montrant que l'idée est réalisable, et également pour susciter des réflexions sur une nouvelle organisation d'un système classique.

La Mémoire Centrale de l'ordinateur IRIS 80 est composée de 1 à 8 Unités Mémoire (UM) possédant de 2 à 8 voies d'accès (auxquelles sont associés des niveaux de priorité) et constituée chacune de 1 ou 2 blocs, chaque bloc contenant 128 ou 256 K octets.

On peut donc avoir, dans une configuration Mémoire maximum, 16 blocs de 256 K octets autonomes et asynchrones utilisés simultanément par 8 utilisateurs connectés aux 8 voies d'accès possibles.

Une configuration possible pour l'insertion de PASC-HLL dans un système IRIS 80 serait:



La liaison programmée (instruction WD écriture directe) pourrait être utilisée par l'Unité Centrale IRIS 80 pour transmettre à l'Unité Centrale PASC-HLL le numéro du bloc (4 bits) dans lequel se trouve le programme à exécuter. A l'opposé, PASC-HLL émettrait un "Interruption" pour avertir l'Unité Centrale IRIS 80 qu'elle doit abandonner un programme pour cause d'opération-système.

Dans une telle organisation, il n'existe pas de conflit d'accès à un bloc de la Mémoire Centrale, sachant que:

- les liaisons physiques (OMNIBUS) sont indépendantes,
- une seule Unité Centrale accède à un bloc à un instant donné.

Il n'y a donc pas de problème de priorité entre les deux unités centrales.

Se pose cependant la question de comptabilité avec les mécanismes d'adressage virtuel généralement utilisés dans les systèmes IRIS 80 classiques, qui décomposent la Mémoire Centrale en "pages" de 2 K octets: dans chaque Unité Centrale IRIS 80, une Mémoire Associative de 16 mots permet d'obtenir directement les numéros de 16 pages réelles associées au programme en cours d'exécution.

Comme nous le verrons plus loin, la machine PASC-HLL, à cause de son organisation "pipeline", ne peut pas se permettre de trouver une "faute de page" lors d'un accès à la Mémoire Centrale, parce qu'elle serait alors incapable de sauvegarder son contexte: au contraire, son mode de fonctionnement impose que toutes ses données soient présentes en Mémoire Centrale. Il est donc indispensable que le système d'exploitation bloque un certain nombre de "pages" réelles contenant les données utilisées par PASC-HLL.

S'il est impossible au système d'exploitation de compacter toutes les données relatives à un programme dans le même bloc physique, nous pouvons envisager une solution intermédiaire qui consiste à implanter chacune des quatre zones de données (voir plus loin la définition de ces zones) utilisées par PASC-HLL dans des ensembles de pages réelles contigües, dont le nombre dépend de la taille réelle de ces zones.

Les données relatives à un programme seraient alors implantées dans des blocs physiques différents en Mémoire Réelle, et des conflits d'accès pourraient alors se produire, qui auraient pour effet de ralentir les deux Unités Centrales ainsi que les Unités d'échange, donc de diminuer les performances du système global.

Une étude importante reste donc à faire pour étudier une connexion éventuelle à un ordinateur IRIS 80, qui a été choisi comme exemple à cause de son existence à GRENOBLE et de la facilité technique de connexion à sa mémoire.

De plus, la définition d'un nouveau système d'exploitation reste à faire, qui permettrait d'envisager de nouvelles techniques d'exploitation basées sur la notion de multiprogrammation fonctionnelle plutôt que sur celle de multiprogrammation aléatoire dirigée par l'occurrence de "fautes de page" ou de "tranches de temps".

B - L'ESPACE D'ADRESSAGE DE LA MACHINE

Nous abordons ici un des aspects "système" de la machine PASCALE. Afin de rendre plus aisée et plus souple la connexion de la machine avec la Mémoire Centrale gérée par le système d'exploitation "hôte", nous l'avons dotée d'un processeur d'interface avec la Mémoire Centrale. Ce processeur est appelé Processeur Mémoire et noté PME.

I.- STRUCTURATION DE L'ESPACE D'ADRESSAGE

Les processeurs composant la machine (PINS, POP et PAC) adressent les informations dans un espace d'adressage virtuel composé de quatre zones.

La première zone, appelée CODE, contient toutes les informations en lecture seule générées par le compilateur: la table des TYPES, la table des CONSTANTES, le code des procédures.

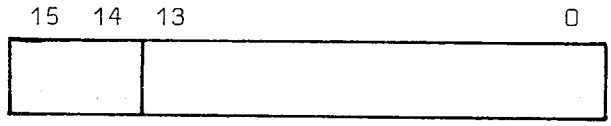
La seconde zone, appelée EXTERNE, contient les mêmes informations que la zone CODE, informations qui sont relatives aux modules externes issus de compilations séparées. C'est sur cette zone que s'effectue le recouvrement (OVERLAY) des modules externes.

La troisième zone, appelée CONTEXTE, contient la pile de contexte de l'exécution du programme et reflète l'imbrication dynamique des procédures.

La dernière zone, appelée DYNAMIQUE, est réservée à l'allocation dynamique des données.

II.- ADRESSAGE DES QUATRE ZONES DE L'ESPACE VIRTUEL

II.1. Chacun des processeurs (PINS, POP et PAC) sait dans quelle zone il désire lire ou écrire une information. Il envoie donc une adresse de la forme suivante:



où les bits 14 et 15 indiquent le numéro de la zone, et les bits 0 à 13 le déplacement dans cette zone.

On remarque que chaque zone peut contenir jusqu'à 16 Kmots (14 bits de déplacement).

II.2. Le processeur PME reçoit donc une adresse virtuelle. Il la convertit en une adresse réelle de la manière suivante:

Lors de l'initialisation d'un travail, le système d'exploitation fournit à une adresse fixe, une table des zones réelles, qui décrit l'implantation réelle des quatre zones en Mémoire Centrale.

Cette table contient, pour chaque zone, l'adresse de son implantation et sa longueur.

	adresse réelle	longueur
0		
1		
2		
3		

Table des zones réelles

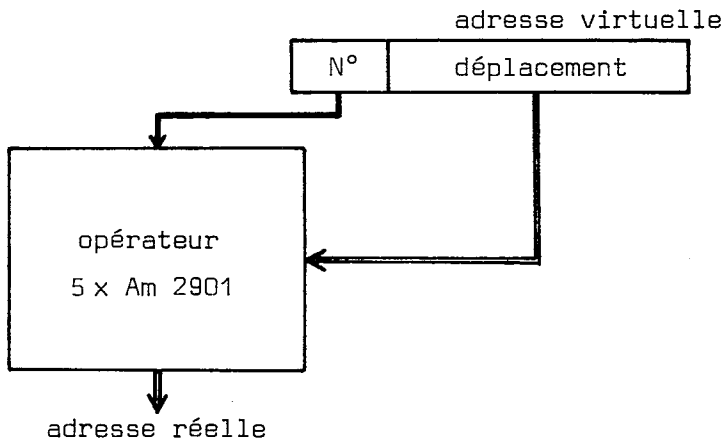
Le processeur PME, qui dispose d'un opérateur Am 2901 de 20 bits, initialise quatre registres de base avec les adresses réelles des quatre zones.

Ainsi, pour toute opération de Lecture/écriture en Mémoire Centrale demandée par l'un des processeurs PINS, POP ou PAC, le processeur PME effectue:

1/ une translation dynamique d'adresse par l'opération

BASE(N°ZONE) + DEPLACEMENT

2/ une protection en écriture pour les deux zones CODE et EXTERNE, qui ne contiennent que des informations en lecture seule.



Remargue :

Si le processeur PME détecte une tentative d'écriture dans l'une des zones CODE ou EXTERNE, il arrête l'exécution et avertit le système du fait qu'un processeur de la machine PASCALE commet une erreur grave et se trouve donc en panne.

II.3. Avantages

1/ Le premier avantage consiste en le fait que le système d'exploitation "hôte" peut implanter chacune des quatre zones n'importe où dans la Mémoire Centrale, en fonction de sa gestion propre et de la taille de chacune des zones.

2/ Le second avantage est relatif à la taille variable des quatre zones:

- la taille de la zone CODE est parfaitement connue: elle est donnée par le compilateur.

- la taille de la zone EXTERNE est celle du module EXTERNE présent en mémoire centrale: elle peut donc changer lorsqu'on change de module externe, mais là encore sa taille est donnée par le compilateur.

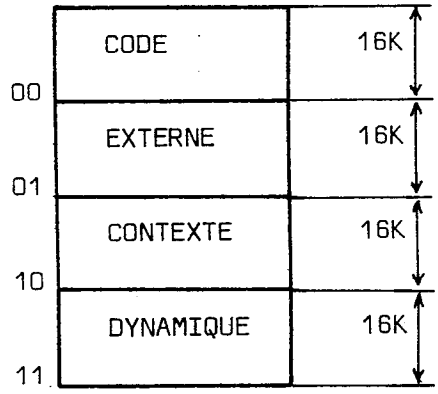
- la taille de la zone CONTEXTE est fonction de la complexité du programme (imbrication dynamique des procédures) et du nombre de variables manipulées. Elle sera donc définie par le programmeur, lorsqu'il demande l'exécution de son programme: c'est un paramètre d'une "instruction-système".

- la taille de la zone DYNAMIQUE est elle aussi dépendante des données manipulées: elle peut être nulle dans le cas où le programmeur ne fait aucune allocation dynamique. Ce sera également un paramètre d'une "instruction-système".

3/Enfin, toutes les informations relatives à un travail sont translatables: elles peuvent être implantées par le système d'exploitation n'importe où en mémoire, et changer de place selon les besoins du système (une zone peut être sauvegardée sur disque par le système et remise en mémoire à une adresse différente).

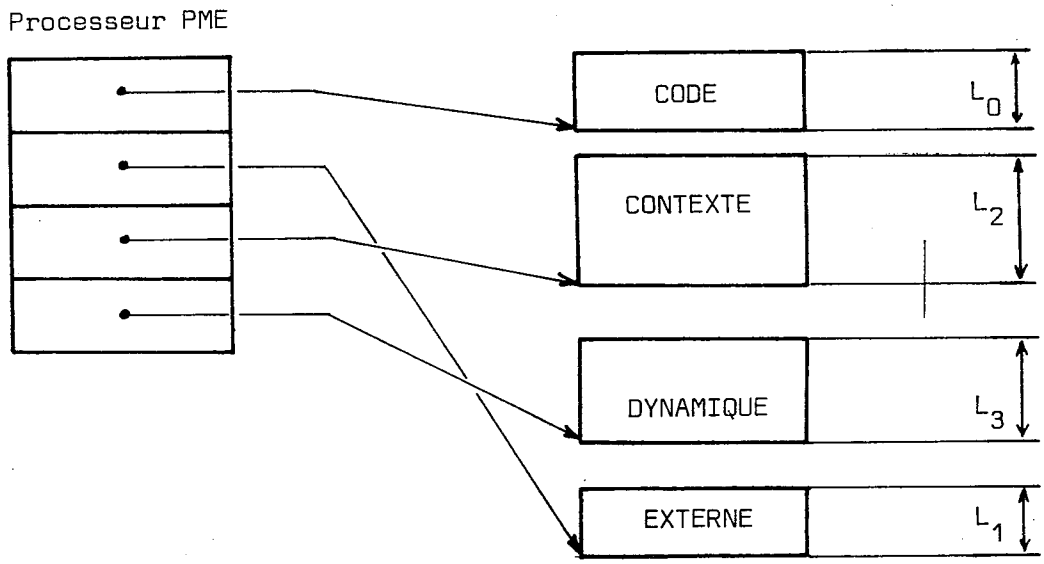
Mémoire virtuelle de la
Machine PASCALE

Espace virtuel:



Mémoire Réelle du
système "HOTE"

Espace réel:



III.- ORGANISATION DES ZONES CODE et EXTERNE

La zone CODE contient le module Principal généré par le compilateur. Son organisation a été conçue:

1/ pour faciliter le travail du compilateur,

2/ pour permettre une édition de liens facile et rapide entre le Module Principal et les Modules Externes,

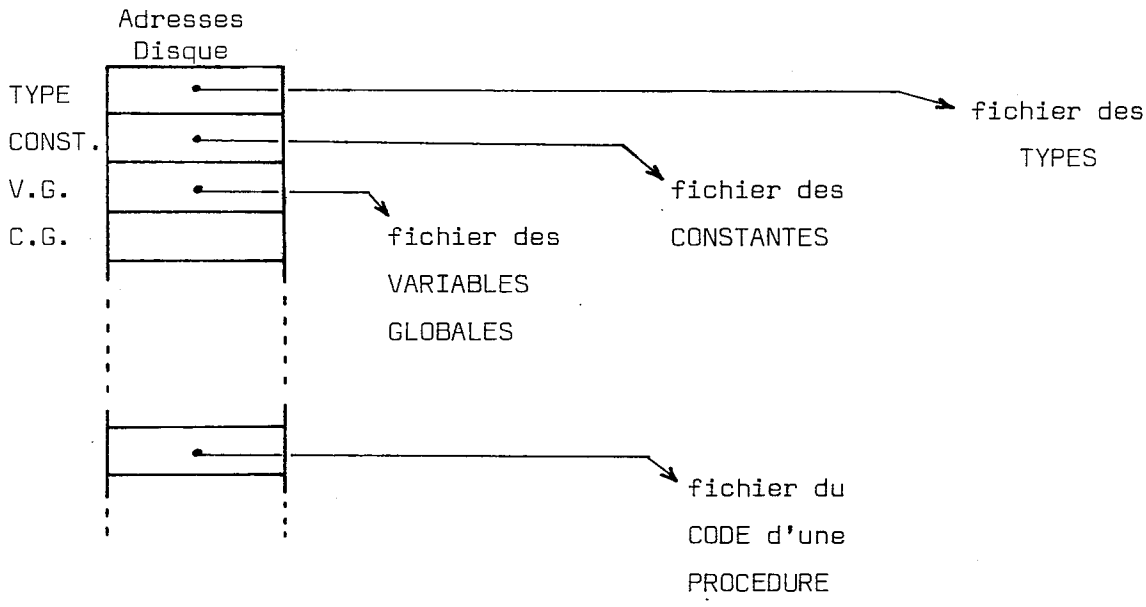
3/ pour permettre l'exécution de procédures externes.

III.1. Résultat d'une compilation

Au cours du processus de compilation, plusieurs catégories d'informations sont générées:

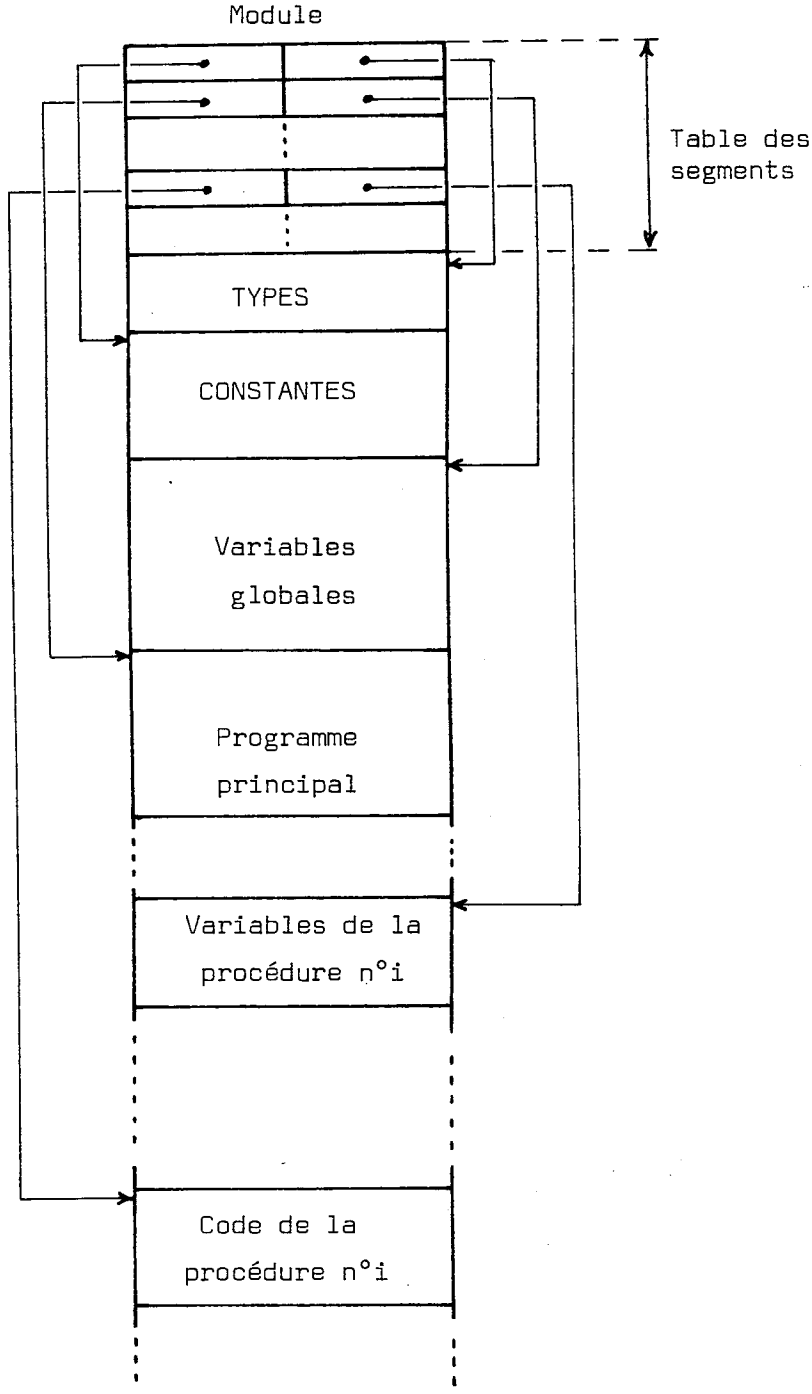
- une table des TYPES construite conformément aux déclarations de types,
- une table des CONSTANTES,
- les descripteurs de paramètres et de variables pour chaque procédure,
- enfin, le code-machine pour chaque procédure.

Toutes ces informations sont regroupées, à la fin du processus de compilation, sous la forme d'un Descripteur de Module, contenant les adresses des fichiers construits.



III.2. Chargement d'un module

Avant de pouvoir exécuter un module, toutes les informations qui le composent sont lues sur disque et chargées en Mémoire Centrale: on obtient ainsi un Module exécutable, qui est chargé soit dans la zone CODE, soit dans la zone EXTERNE, selon l'organisation suivante:



Les N premiers mots d'un MODULE constituent la TABLE DES SEGMENTS de ce module.

. le premier mot de la table des segments (déplacement =0) contient l'adresse relative de la table des TYPES et celle de la table des CONSTANTES (ces deux adresses sont lues par le processeur POP pour construire l'adresse d'un descripteur de type ou celle d'une constante).

. le (i+1)ième mot de la table (déplacement=i) contient les informations relatives à la procédure n°i (i est le paramètre des instructions CALL pour PAC et ENTER pour POP).

Remarque:
On aurait pu réaliser une segmentation réelle pour les procédures, mais les problèmes posés par les modules externes sont dans ce cas insolubles. Les différents segments (procédures) sont donc supposés être chargés dans une zone continue (CODE ou EXTERNE).

III.3. Adressage des procédures externes

1/ définition des compilations séparées

Nous avons choisi, en accord avec d'autres concepteurs de compilateur PASCAL ([ ]), de prévoir la notion de compilations séparées de plusieurs morceaux d'un même programme.

L'organisation choisie, voisine de celle de PL/1, est caractérisée par l'existence d'un Module Principal, défini par l'entête:

PROGRAM <NOM> ;

{ programme PASCAL

Dans la description d'un tel programme, peuvent apparaître des déclarations de procédures "externes" sous la forme suivante:

procédure <NOM> {<liste de paramètres>} ; PASCAL ;

Plusieurs procédures (ou fonctions) "externes" peuvent ainsi être déclarées et sont normalement "appelées" dans le programme principal.

Ces procédures (ou fonctions) "externes" appartiennent à des Modules Externes qui sont compilés séparément. La structure d'un Module Externe est la suivante:

- sa description ne commence pas par le mot-clé program mais par

EXTERN <NOM> ;

Aucune déclaration de variables globales ne peut être faite, mais par contre des types peuvent être déclarés, qui sont propres à ce module (la déclaration de types est indispensable pour la description des paramètres).

Viennent ensuite des déclarations de procédures (ou de fonctions) dont les noms sont ceux des "points d'entrée" dans le module externe.

Ainsi, dans l'exemple suivant:

```
EXTERNE   TRIGO ;
TYPE
      ≀
  function  SIN(- - - ): real ;
        ≀
  function COS(- - - ): real ;
        ≀
```

le module TRIGO a deux points d'entrée qui ont pour noms SIN et COS.

2/ édition de liens et chargement

Le travail de l'éditeur de liens consiste à mettre à jour la table des segments du module principal.

Résultat de la compilation du Module principal

Le compilateur fournit la table des externes.

NOM	#S
TOTO	4
JOJO	5
MOMO	8
KOKO	10

Résultat de la compilation des modules externes

Le compilateur fournit une table des entrées pour chaque module.

Module 1

NOM	#S
KOKO	1
TOTO	4
MOMO	15

Module 2

NOM	#S
JOJO	1

Mise à jour de la table des segments du Module Principal

(au moment du chargement du Module principal)

TOTO → #4	1	#M=1	#S=4
JOJO → #5	0	#M=2	#S=1
MOMO → #8	1	#M=1	#S=15
KOKO → #10	1	#M=1	#S=1

Dans l'exemple, le module #1 est chargé dans la zone EXTERNE.

3/ liaison avec un module externe

Toute procédure déclarée "EXTERNE" dans le programme principal est affectée d'un numéro de segment et occupe un mot dans la table des segments du module principal.

Lorsque le processeur PINS reçoit une instruction CALLEXT(i), il accède au descripteur de segment n°i qui contient les informations suivantes:

segment n°i

P	#M	#S

- un indicateur de présence P dont la valeur "1" indique que le module #M a bien été chargé dans la zone EXTERNE.

- le n° du module externe #M qui contient la procédure externe.
- le n° #S du segment de la procédure dans le module externe.

En cas d'absence du module (P=0), le processeur PINS envoie une instruction de sauvegarde vers ses deux esclaves PAC et POP et avertit le système qu'il s'est produit une "faute de module externe" n° #M.

Le système est supposé capable de trouver l'adresse disque d'un module et de mettre à jour la table des segments du module principal.

4/ adressage des types et des constantes

La difficulté provient du fait des compilations séparées: les descripteurs de types peuvent appartenir à deux zones différentes, la zone CODE pour les types de programme principal, la zone EXTERNE pour les types d'une procédure externe, et le même problème se pose pour les constantes.

4.1. nécessité de vérification des types de paramètres

Supposons qu'une procédure soit déclarée "externe" de la manière suivante:

(D1) procédure PEXT (P1: <type1> ; P2 : <type2>) ; PASCAL ;

Dans le module externe, compilé séparément, sa déclaration comme "point d'entrée" est:

(D2) procédure PEXT (P'1: <type'1> ; P'2: <type'2>) ;

Sachant qu'il n'y a aucun lien entre deux compilations séparées, on ne peut que vérifier, au cours de la compilation du programme principal, qu'un appel "PEXT (<exp1>, <exp2>);" est compatible avec la déclaration (D1).

Au moment de l'exécution de cet appel de procédure, nous devons donc prévoir une vérification de la compatibilité des types des paramètres ainsi que de leur nombre. La seule solution consiste à prendre les descripteurs des paramètres formels dans le module externe, afin de vérifier dynamiquement qu'il y a bien compatibilité.

4.2. réalisation de l'appel d'une procédure externe

Lorsque le compilateur analyse une instruction d'appel de procédure, il sait si celle-ci est interne ou externe. Dans le 2ème cas, il génère les instructions suivantes:

pour PEXT(<exp1>, <exp2>) ;

```
CALL-EXT  #S(PEXT)
    ≀  <exp1>
PARAM-EXT  SP, -3
    ≀  <exp2>
PARAM-EXT  SP, -4
ENTER-EXT  #S(PEXT)
```

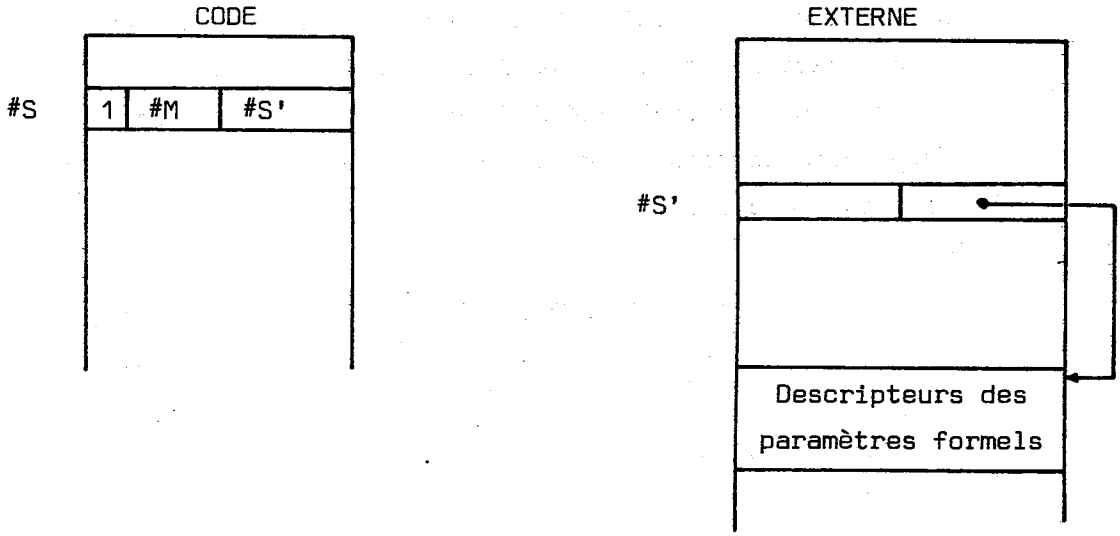
1) l'instruction "CALL-EXT(#S)" est accédée par le processeur PINS, qui accéde au segment #S localisé à l'adresse #S dans la zone CODE.

Si le bit de présence vaut '1', alors l'exécution peut continuer puisque le Module externe qui contient PEXT est présent en Mémoire centrale.

Dans le cas contraire, l'exécution est arrêtée: des instructions de sauvegarde du contexte sont envoyées à PAC et à POP et le système est averti par PINS qu'il y a "faute du module #M ".

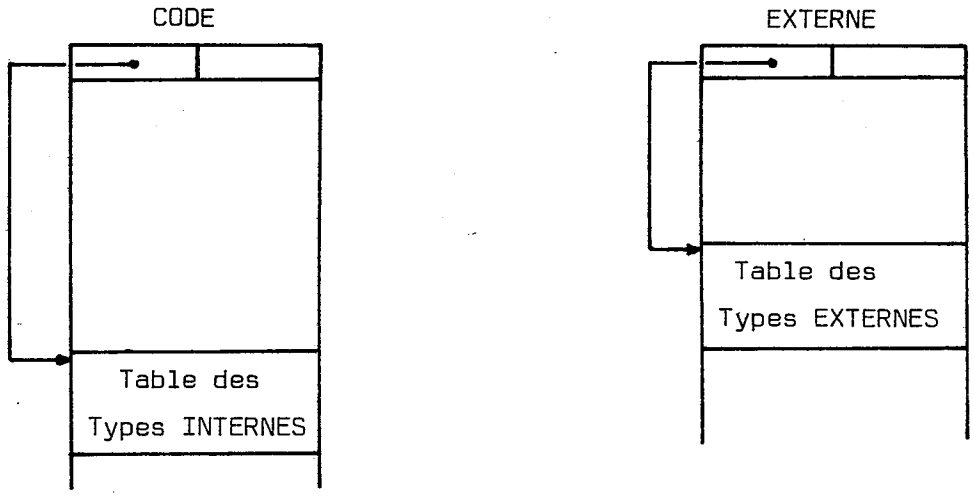
2) l'instruction "CALL-EXT(#S)" est reçue par le processeur PAC uniquement si le module #M est présent. Le descripteur de segment principal #S

contient le numéro #S' du segment externe. Cette indirection permet à PAC d'accéder aux descripteurs des paramètres formels déclarés dans la procédure externe.



Ces descripteurs de paramètres formels sont recopiés sur la pile de contexte (zone CONTEXTE).

3) le processeur POP reçoit l'instruction PARAM-EXT. Le suffixe "EXT" lui indique que le descripteur de type du paramètre se trouve dans la table des types externe, dont l'adresse est donnée par le segment n°0.



4) le processeur PINS accède l'instruction ENTER-EXT(#S). Il accède au segment n°#S dans la table des segments principale, qui lui donne le n°#S'

du segment de la procédure externe, qui contient l'adresse du code de cette procédure. Il passe en mode "EXTERNE", c'est-à-dire que toutes les instructions seront accédées dans la zone EXTERNE et il continue l'exécution, en envoyant l'instruction ENTER-EXT aux deux processeurs PAC et POP.

5) le processeur PAC reçoit l'instruction ENTER-EXT. Il a noté dans la pile de contexte l'adresse, dans la zone EXTERNE, des descripteurs des variables de la procédure externe. Il les recopie sur la pile de contexte et passe en mode "EXTERNE".

6) le processeur POP reçoit l'instruction ENTER-EXT. Il accède au segment #0 du module externe, qui contient la base des types externes et la base des constantes externes. Il mémorise ces deux bases dans deux registres internes et passe en mode EXTERNE.

4.3. réalisation du retour d'une procédure externe

Lors de la compilation d'un module externe, toute fin de procédure du premier niveau (point d'entrée externe), conduit à la génération de l'instruction RETURN-EXT.

1) le processeur PINS accède à l'instruction RETURN-EXT

Il sort alors du mode EXTERNE et retourne dans le segment appelant du module principal.

2) le processeur PAC reçoit l'instruction RETURN-EXT

Il sort du mode EXTERNE et réalise un retour normal de procédure en mettant à jour la pile de contexte et les chaînages statiques et dynamiques.

3) le processeur POP reçoit l'instruction RETURN-EXT

Il remet à jour les bases des types et des constantes en allant lire leur valeur dans le segment #0 du module principal, puis il sort du mode EXTERNE.

Remarque: dans le mode EXTERNE, les trois processeurs adressent la zone EXTERNE pour l'accés aux descripteurs de segments, de types, de paramètres et de variables et aux constantes.

4.4. problèmes propres aux fonctions externes

Supposons la déclaration suivante:

dans le programme principal:

```
function F(P1: <type1>) : <type2> ; PASCAL ;
```

et dans le module externe:

```
function F(P'1: <type'1>) : <type'2>;
  {
begin
     { code
     F := <expression>
end ;
```

Le résultat d'une fonction est passé en paramètre par référence. Ainsi, dans le programme principal, le compilateur crée une variable, appelée F, dont le type est <type2>.

Tout appel de la fonction F(par exemple F(<exp>)) donne lieu à la génération de:

```
Call-Ext (#S)
Id F
Param-Ext SP,-3
  {   <exp>
Param-Ext SP,-4
Enter-Ext (#S)
Nom F
```

Lors du passage du premier paramètre, le processeur POP peut ainsi vérifier qu'il y a compatibilité entre <type2> donné par le descripteur de F et <type'2> donné par le descripteur du paramètre formel, résultat de la fonction.

Le passage de paramètre par référence permet de plus de transmettre la valeur du résultat de la fonction depuis le module externe par l'instruction AFFECT LOC,-3 générée à l'occurrence de F := <expression>.

IV.- ORGANISATION DE LA ZONE CONTEXTE

Les trois processeurs PINS, PAC et POP utilisent une pile. Ces trois piles sont implantées dans la zone CONTEXTE.

- le processeur PINS empile et dépile des descripteurs de contrôle des segments du programme

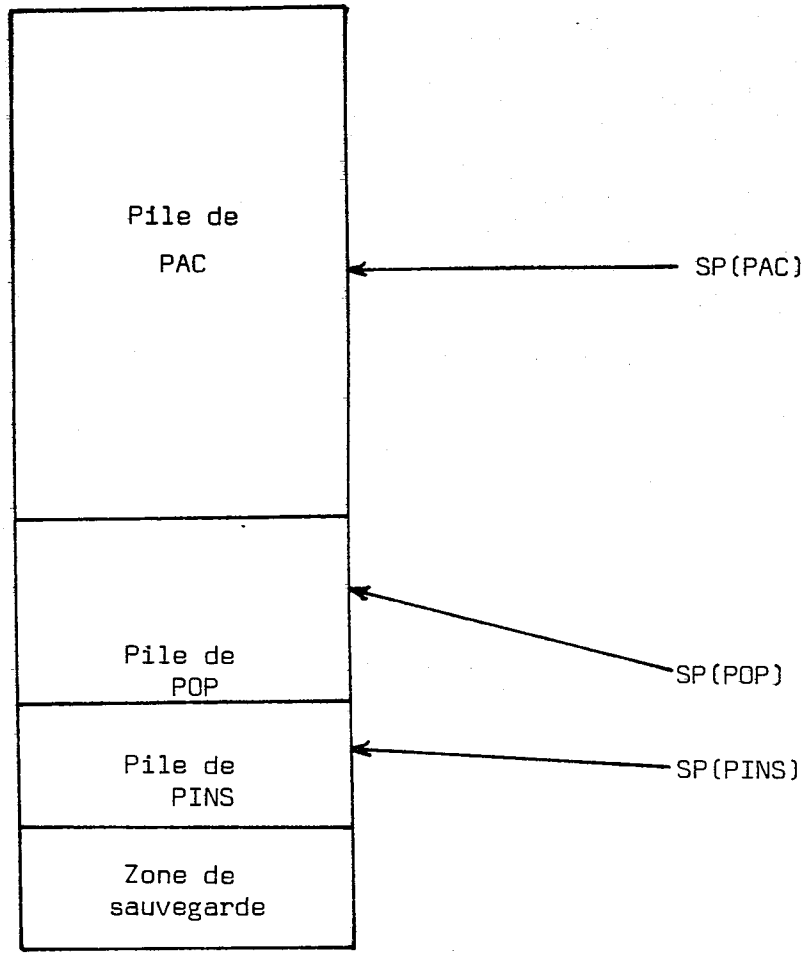
- le processeur POP empile et dépile des résultats intermédiaires longs ainsi que des descripteurs de contrôle de boucle (valeur courante et valeur limite de l'indice)

- le processeur PAC gère la pile de contexte selon l'imbrication des procédures (cette pile contient les variables des procédures et les marques de blocs)

Ces trois piles sont adjacentes en Mémoire Centrale et les processeurs PINS et POP disposent d'une pile interne de 16 mots (RAM).

Une zone fixe, située "en bas" de la zone CONTEXTE, permet de sauvegarder les adresses des sommets des piles et les registres de base en ce qui concerne le processeur PAC.

Image de la zone CONTEXTE au cours de l'exécution



Initialisation par le système

La valeur initiale de la zone de sauvegarde est chargée par le système avec:

SP(PINS) = <taille Z de sauvegarde

SP(POP) = <SP(PINS) + Taille Pile(PINS)

SP(PAC) = <SP(POP) + Taille Pile(POP)

W(PAC) = 0

B0(PAC) = <SP(PAC) + Taille Var-Prédéfinies>

B2(PAC) = ... = B6(PAC) = 0

Les valeurs de la base des types et de la base des constantes du Module Principal sont également chargées en adresse #2.

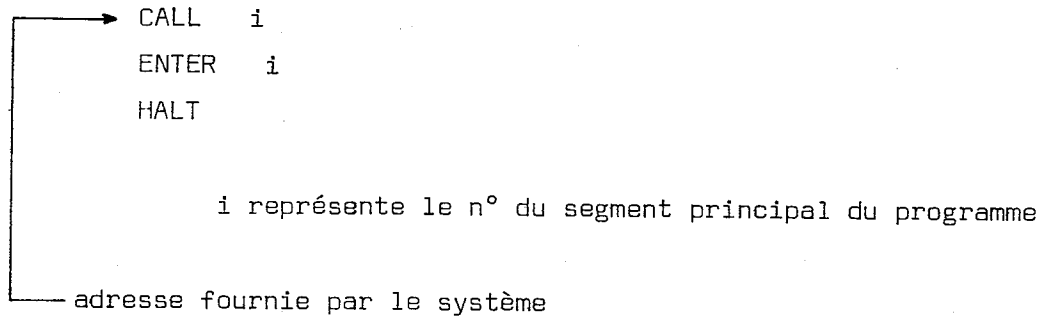
Sauvegarde par les processeurs

Lorsque PAC et POP reçoivent de la part de PINS une instruction de sauvegarde complète, ils écrivent les informations précédentes dans la zone de sauvegarde, après avoir vidé leur pile interne en mémoire centrale.

Restauration du contexte

Dans un but de simplification, la restauration d'un contexte doit être identique à l'initialisation première. Cela suppose donc que le processeur PINS trouve au sommet de sa pile l'adresse de la première instruction à exécuter, c'est-à-dire l'appel du programme principal dans le cas de l'initialisation: cette instruction d'appel est générée par le compilateur, c'est le point d'entrée, dont l'adresse est mise par le système comme sommet de pile du processeur PINS.

Séquence initiale d'instructions



V.- ORGANISATION DE LA ZONE DYNAMIQUE

La zone DYNAMIQUE est réservée à l'allocation dynamique de variables au cours de l'exécution d'un programme.

Le langage PASCAL permet la création d'espaces en mémoire pour contenir des informations repérées par des POINTEURS. Tout pointeur est déclaré comme étant une variable dont la valeur représente l'adresse de l'élément pointé et il est caractérisé par le type de cet élément.

Ainsi, on déclare:

var P : ↑<type>

et l'élément pointé par P est de type <type>.

V.1. L'allocation dynamique est commandée par l'instruction

NEW(P) ;

de PASCAL. Cette instruction peut spécifier des paramètres supplémentaires sous la forme

NEW(<pointeur> , <liste d'aiguillages>) ;

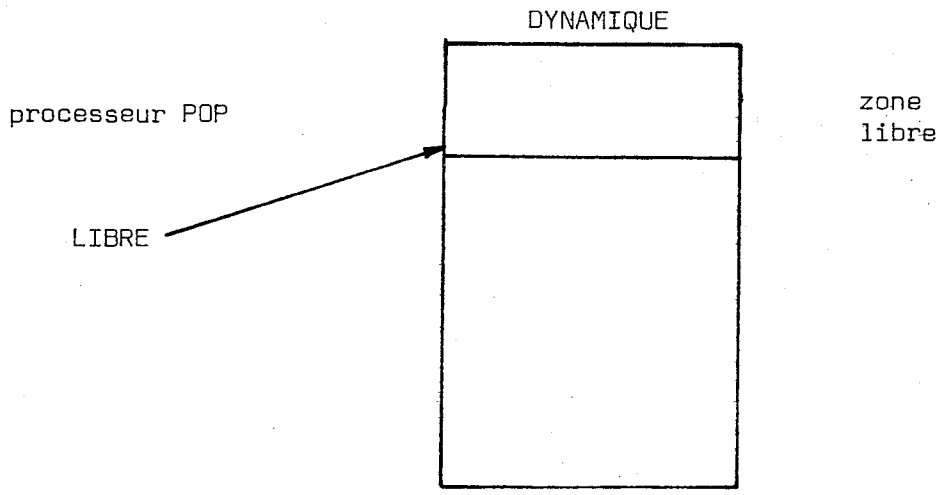
où la <liste d'aiguillage> définit le choix de branches successives dans une structure RECORD qui peut avoir plusieurs branches sous la forme d'une arborescence. (voir opérateur NEW, chapitre POP)

Cette liste d'aiguillage a pour but de spécifier un sous-type du type RECORD et définit dynamiquement la taille de l'élément ainsi créé.

V.2. La libération d'espace est commandée par l'instruction DISPOSE (<pointeur>), qui spécifie la libération de toute la zone située "au dessus" du mot pointé par le <pointeur>.(voir opérateur DISPOSE, chapitre POP)

V.3. Adressage de la zone DYNAMIQUE

La zone DYNAMIQUE est adressée uniquement par le processeur POP, au moyen de "pointeurs", lorsqu'il accéde à la valeur des éléments "pointés".



Les mouvements du pointeur LIBRE sont commandés par les instructions NEW et DISPOSE qui respectivement, créent et libèrent des éléments dans la zone.

La valeur de ce pointeur est évidemment sauvegardée en Mémoire Centrale par le processeur POP lorsqu'un changement de contexte se produit.

C - LES ENTREES-SORTIES

INTRODUCTION

Tout programme écrit en langage évolué fait obligatoirement référence à au moins un fichier de sortie, qui contiendra les résultats exploitables de l'exécution du programme.

Le langage PASCAL fournit au programmeur deux fichiers prédéclarés (INPUT et OUTPUT) qui correspondent à un "fichier-carte" et un "fichier-imprimante". On peut cependant déclarer d'autres fichiers, de la manière suivante:

<nom de fichier> : FILE OF <type>.

Le premier problème consiste à définir un lien entre l'exécution du programme et le système d'exploitation: ce lien est établi par le numéro du fichier, défini par l'ordre des déclarations des fichiers dans le programme, et constitue un des paramètres de l'appel du système d'exploitation par la machine PASCALE, un autre paramètre donnant évidemment la nature de l'opération d'entrée/sortie demandée au système.

Cet appel au système est réalisé par une "interruption" lancée vers la machine "hôte" qui exécute le Système.

I.- STRUCTURE DES INFORMATIONS

I.1. Aspect système

Le système d'exploitation possède une table des fichiers relatifs au programme principal. Cette table contient deux éléments:

1/ l'adresse du TAMPON qui contient la valeur du dernier élément du fichier accédé (ce TAMPON est localisé dans la zone CONTEXTE, en adresse basse).

2/ l'adresse d'un BLOC DE DESCRIPTION du fichier. Ce bloc,exploité par le système d'entrée/sortie, contient toutes les informations nécessaires à la méthode d'accés au fichier (adresse du périphérique, adresse sur le périphérique, mode d'accés, longueur d'un bloc, etc...).

I.2. Aspect machine PASCALE

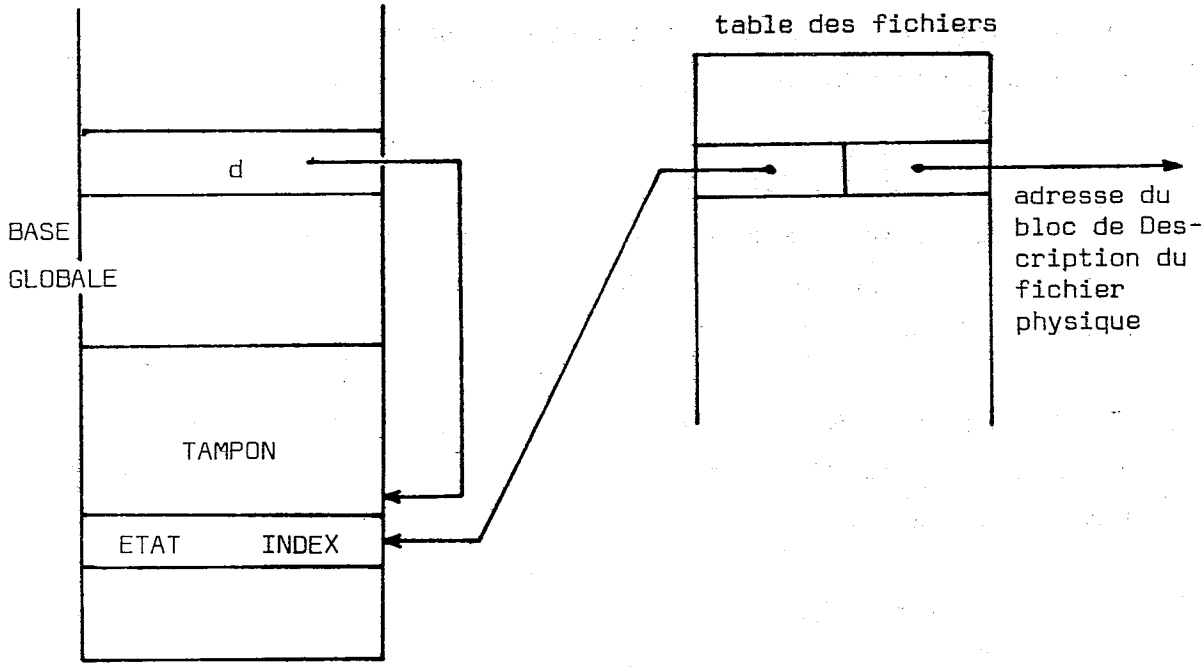
1/ Les ordres d'entrée/sortie émis par la machine PASCALE spécifient le numéro du fichier concerné. Ce numéro est utilisé par le système pour accéder au fichier par l'intermédiaire de la Table des Fichiers.

2/ Les accés à la valeur de l'élément du fichier sont faits par les processeurs PAC et POP par référence à une variable structurée, dont le champ PV est égal à l'adresse du TAMPON associé au fichier.

3/ Le test de la fin du fichier (fonction EOF) et la transmission de la valeur de l'index dans le cas de fichiers indexés, utilisent un MOT DE CONTROLE localisé dans le premier mot de TAMPON.

1.3. Adresse de la zone TAMPON

Pour chaque fichier déclaré (ou prédéclaré), la zone TAMPON, dont la taille est égale à la taille maximum d'un élément du fichier plus 1 mot de contrôle, est adressé à la fois par la machine PASCALE par l'intermédiaire de la variable associée au fichier, et par le système par l'intermédiaire de la table des fichiers, selon la figure



zone CONTEXTE

II.- EXECUTION DES OPERATIONS D'ENTREE/SORTIE

Un programme écrit en langage évolué (PASCAL par exemple) contient des ordres d'entrées/sorties élémentaires et des accés à des éléments de fichiers.

Le programme demande un accés à l'élément suivant (GET(FICHIER)) puis référence l'élément fourni par le système (FICHIER↑). Le travail de la machine langage qu'est la machine PASCALE se limite à détecter les ordres d'entrées/sorties et à les transmettre au système d'exploitation qui l'accueille.

II.1. Détection des ordres d'entrée/sortie

L'organisation "pipe-line" de la machine PASCALE fait apparaitre un processeur maître, le processeur PINS qui est la première station du "pipe-line" et deux esclaves qui sont les processeurs PAC et POP.

Problème: Est-il possible de confier à l'un des esclaves (PAC ou POP) la charge de détecter une condition d'arrêt de l'exécution? La réponse est négative. La preuve peut en être faite par l'absurde.

Supposons que PAC détecte une condition d'arrêt. A cet instant précis, sa file d'attente d'instructions n'est pas vide et le processeur PINS continue d'analyser les instructions suivantes: il lui est impossible de retrouver le contexte de l'instruction qui produit l'arrêt du processeur PAC et donc de préparer une sauvegarde du contexte général de la machine.

Conséquence: Tout arrêt de l'exécution d'un programme doit être commandé par la station maître du pipeline, en l'occurrence le processeur PINS.

En particulier, toute opération d'entrée/sortie doit être détectée par le processeur PINS.

II.2. Les ordres d'entrée/sortie

Les ordres d'entrée/sortie élémentaires utilisés dans la machine PASCALE sont:

- GET n° fichier = lecture de l'élément suivant pour un fichier séquentiel ou lecture de l'élément dont la clef est égale à l'INDEX passé en paramètre pour un fichier indexé.

- PUTn° fichier = écriture de l'élément courant dans la position suivante pour un fichier séquentiel ou dans la position dont la clef est égale à l'INDEX passé en paramètre pour un fichier indexé.

- RESET n° fichier = remise au départ d'un fichier séquentiel.

- REWRITE n°fichier = passage en écriture d'un fichier qui était en lecture.

Ces quatre ordres sont envoyés par le processeur PINS au système d'exploitation, avec un paramètre égal au numéro du fichier. C'est le système qui exécute l'opération d'entrée/sortie, alors que la machine PASCALE sauvegarde son contexte et se met en attente d'un nouveau travail à exécuter.

II.3. Accés aux éléments des fichiers

Lorsqu'une opération d'entrée/sortie a été exécutée par le système d'exploitation, la machine PASCALE reprend l'exécution du programme dans lequel figurent des instructions d'accés (en lecture ou écriture) à l'élément courant du fichier. La variable associée au tampon du fichier est référencée comme n'importe quelle variable du programme et son type peut être quelconque. C'est un tableau de 80 caractères (par exemple) pour le fichier prédéclaré INPUT et 132 caractères (par exemple) pour le fichier prédéclaré OUTPUT.

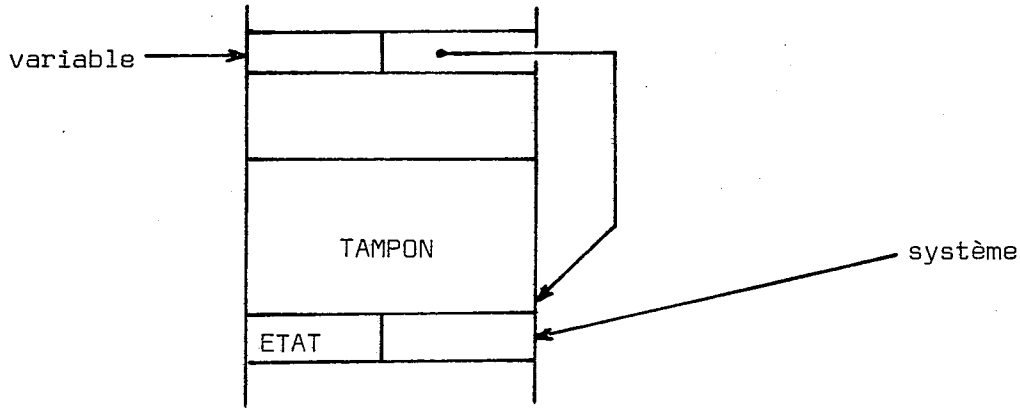
II.4. Test de fin de fichier

Le programmeur dispose de la fonction EOF (fichier) qui est compilée en:

```
NOM < fichier >
EOF
```

où "EOF" est un opérateur binaire qui donne un résultat booléen.

Son opérande est un descripteur de variable, dont le champ "adresse de la valeur" pointe sur le TAMPON du fichier. A cette adresse moins 1, se trouve le mot de contrôle du fichier qui indique, en particulier, l'état "FIN DE FICHIER", état positionné par le système:



II.5. Accés aux fichiers indexés

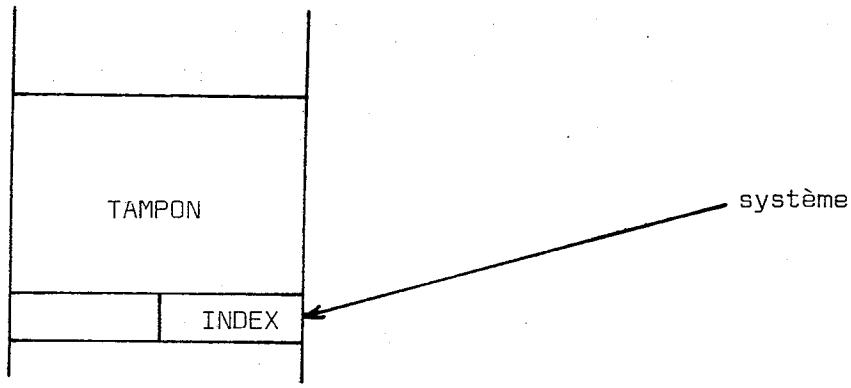
Le programmeur peut spécifier l'index de l'élément du fichier, sous la forme:

GET(<fichier>, <expression>) ou PUT(<fichier>,<expression>).

Cet appel de fonction est compilé en:

NOM <fichier>
<expression>
AFFINDEX
GET/PUT n° fichier

où AFFINDEX est un opérateur binaire, dont l'exécution par le processeur POP conduit à l'affectation de la valeur de <expression> au champ INDEX du mot de contrôle du fichier.



III.- LES ENTREES/SORTIES EVOLUEES

Le programmeur dispose de deux fichiers séquentiels prédéclarés, les fichiers INPUT et OUTPUT, qui correspondent repectivement à un "fichier-carte" et un "fichier imprimante". Le langage PASCAL donne la possibilité d'utiliser des procédures d'entrée/sortie évoluées sous la forme suivante:

III.1. La procédure READ(<variable>)

Cette procédure commande d'abord la lecture de la carte suivante, puis la recherche sur cette carte d'une constante symbolique de même type que la <variable> , et enfin l'affectation de la valeur de la constante symbolique à la <variable>.

Cette procédure complexe du langage PASCAL ne peut pas être directement interprétée par microprogrammation, par le fait qu'elle fait intervenir tous les processeurs de la machine PASCALE et surtout par le fait qu'un ordre d'entrée/sortie doit être émis vers le système.

Sa réalisation est donc programmée en langage-machine et les diverses procédures qui la réalisent sont des points d'entrée dans le module externe que nous appellerons BIBLIOTHEQUE DU SYSTEME.

Les diverses procédures correspondent aux différents types de constantes symboliques qui peuvent apparaître sur une carte et qui sont:

type de constante	nom de la procédure
entier	READINT
réel	READREAL
booléen	READBOOL
scalaire	READSCAL
caractère	READCHAR
ensemble scalaire	READSETSCAL
ensemble entier	READSETINT
chaîne	READALFA

Toutes les procédures précédentes sont appelées avec un paramètre par référence. Leur déclaration est par exemple:

```
procédure  READINT (var N : integer) ;
```

et la programmation de READ(T[I+1]) est équivalente à celle de

```
READINT(T[I+1])
```

qui est compilé en:

```
CALL-EXT   #READINT
NOM   T
NOM   I
SUCC
INDEX
PARAM-EXT   SP,-3
ENTER-EXT   #READINT
```

L'écriture de ces procédures est faite une fois pour toutes par les programmeurs-système, qui disposent de deux variables prédéfinies appelées NIMPUT et NOUTPUT, qui donnent respectivement la position dans la carte lue et la position dans la ligne écrite pour les deux fichiers prédéfinis INPUT et OUTPUT. Ces deux variables sont implantées à deux adresses fixes en tant que variables globales (noms(0,0) et (0,1)).

III.1.1. la procédure READINT

La première instruction est un appel au système: on lui demande la lecture de la carte suivante. Le pointeur NINPUT est initialisé à ZERO et on recherche la première occurrence d'une constante entière (suite de chiffres décimaux). La constante entière trouvée est rangée dans la variable SYMB, qui est de type "chaîne de caractères", puis on programme l'affectation du symbole SYMB au paramètre N.

```
N := SYMB ;
```

Lors de l'exécution, le processeur POP reçoit un ordre d'affectation d'une chaîne de caractères (variable SYMB) à une variable entière (N).

La conversion "chaîne de caractères" → "valeur entière" est microprogrammée dans le processeur POP qui exécute une séquence de multiplications par 10 selon l'algorithme:

```
N := 0;
for  I := 0 to L ;
   N := 10*N + SYMB[I]
```

où L est la longueur de la chaîne de caractères.

La multiplication par 10 est exécutée en 2 microinstructions:

1/ (N+N)*2 → M ⇒ M = 4*N

2/ (M+N)*2 → N ⇒ N = (4*N + N)*2 = 10*N

On réalise ainsi une très bonne performance pour cette conversion, en faisant un compromis entre la programmation et la microprogrammation.

II.1.3. la procédure READSCAL

Un scalaire symbolique se présente comme un identificateur (chaîne de caractères commençant par une lettre et limitée à 8 caractères).

La procédure accède au symbole et la conversion "chaîne de caractères" → "valeur scalaire" est microprogrammée dans le processeur POP.

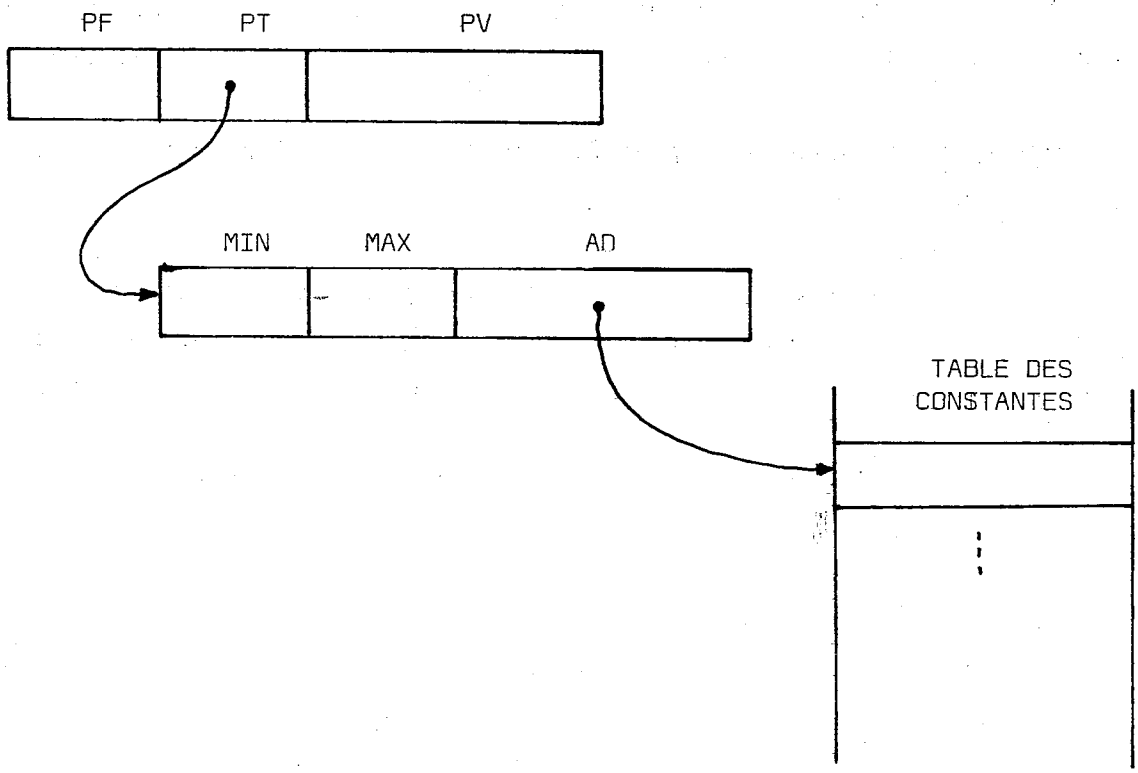
Réalisation de cette conversion:

Rappelons que le descripteur d'une variable de type scalaire contient l'adresse d'un descripteur de type scalaire, qui lui-même contient l'adresse de la première constante symbolique dans la table des constantes. Cette structure est donnée par la figure .

La conversion consiste en une recherche du symbole lu sur la carte dans la table des constantes. Cette conversion est microprogrammée:

- accés au descripteur de type,
- recherche dans la table des constantes,
- affectation du numéro d'ordre du symbole à la variable de type scalaire.

En cas d'échec dans les comparaisons, le processeur POP émet un message d'erreur vers le système.



III.1.5. la procédure READSET

Cette procédure vérifie la syntaxe d'une constante représentant un ensemble sur une carte:

[<constante>, <constante>, ...]

Lorsqu'une constante a été trouvée, elle est mise dans SYMB et l'instruction S := S UNION SYMB permet d'ajouter un élément à l'ensemble, sachant que la conversion "chaîne de caractères"→"valeur scalaire" est microprogrammée et exécutée par le processeur POP lors de l'accés au deuxième opérande de l'opérateur UNION.

III.2. La procédure WRITE (<expression>: <longueur>)

Le programmeur dispose d'une procédure d'impression prédéclarée sur le fichier-imprimante prédéfini.

Elle lui permet de commander l'impression de la valeur d'une <expression>, en spécifiant une <longueur> d'impression (nombre total de caractères).

Le type de l'<expression> peut être n'importe quel type simple: entier, réel, scalaire...

Comme les conversions sont faites par microprogrammation et qu'il n'y a pas de problèmes de reconnaissance d'une syntaxe comme sur la carte d'entrée, tous les ordres d'écriture pourraient être réalisés par une seule procédure externe, appelée WRITE qui aurait pour paramètre une <expression> de type quelconque et la <longueur> d'impression.

Malheureusement, les problèmes de formatage sont propres à chaque type simple: par exemple un entier est "cadré" à droite, une chaîne de caractères est "cadrée" à gauche. Pour cette raison, nous avons, comme dans le cas de la procédure READ, défini autant de procédures externes que de types simples, ce qui présente l'avantage de résoudre le problème du type du paramètre de ces procédures.

type de l'expression	nom de la procédure
entier	WRITINT
réel	WRITREAL
booléen	WRITBOOL
scalaire	WRITSCAL
caractère	WRITCHAR
ensemble scalaire	WRITSETSCAL
ensemble entier	WRITSETINT
chaîne	WRITALFA

Toutes les procédures précédentes sont compilées une fois pour toutes dans le module externe appelé BIBLIOTHEQUE du SYSTEME. Comme pour les procédures de lectures évoluées, un compromis a été trouvé entre la programmation et la microprogrammation: la conversion "expression" → "chaîne de caractères" est microprogrammée (affectation SYMB := valeur), alors que le formatage sur la ligne de sortie est programmé.

Exemple:

```
procédure WRITINT (N: integer; L: 1..100);
var SYMB: array [0..11] of CHAR ;
begin  SYMB := N ;  ← conversion microprogrammée
   /*  le résultat est cadré à droite dans SYMB */
       puis rangement de SYMB, avec troncation ou extension à gauche,

end ;
```

Remarque: La conversion "valeur entière" → "chaîne de caractères" est microprogrammée dans le processeur POP. L'algorithme de conversion suppose une succession de divisions par 10, dont les restes donnent les chiffres successifs en partant des plus faibles poids, ce qui facilite le cadrage à droite.

Il n'existe pas, à notre connaissance, de méthode simple de division d'un nombre binaire par 10, alors que nous avons donné une méthode de multiplication par 10 en 2 microinstructions (cf. III.1.1. du même chapitre). Une étude a cependant été faite pour réaliser la division par 10 à l'aide d'une succession de divisions par 8. Elle consiste à décomposer le résultat de la division par 8 de la manière suivante:

$$N = 8 * Q_8 + R_8$$

est écrit sous la forme $N = 10* Q_8 + (R_8-2*Q_8)$. On fait $Q_{10} \leftarrow Q_8$ et on pose $D_8 = R_8 - 2*Q_8$.

Si D_8 est compris entre 0 et 9, alors D_8 est bien le reste de la division par 10. Sinon, on fait une nouvelle division par 8 et on obtient:

$$D_8 = 10*Q'_8 + (R'_8-2*Q'_8).$$

On ajoute Q'_8 à Q_{10} et on pose de nouveau

$$D'_8 = R'_8 - 2*Q'_8.$$

On recommence ainsi jusqu'à trouver un dividende D_8 compris entre 0 et 9. Cet algorithme est performant pour un dividende petit:

- 1 pas pour $10 \leq N \leq 15$
- 2 pas pour $16 \leq N \leq 19$
- 1 pas pour $20 \leq N \leq 23$

mais il faut par exemple 7 pas pour N = 28341

D_8	Q_8	R_8	Q_{10} partiel	
28341	3542	5	3542	
-7079	-885	1	2657	
1771	221	3	2878	
-439	-55	1	2823	
111	13	7	2836	
-19	-3	5	2833	
11	1	3	2834	← Q_{10}
R_{10} → 1				

Sachant que chaque pas peut nécessiter un nombre important de cycles (environ 8 cycles), il vaut mieux utiliser l'algorithme classique de division qui nécessite 16 cycles quelle que soit la grandeur du dividende.

III.3. les procédures WRITELN et READLN

Ce sont des cas particuliers des procédures WRITE et READ qui spécifient que l'écriture ou la lecture se font sans changer de ligne ou de carte.

On ajoutera donc un paramètre booléen à chaque procédure READ ou WRITE qui spécifiera cette condition.

IV.- CONCLUSION SUR LES ENTREES/SORTIES

La machine PASCALE étant avant tout une machine-langage destinée à l'exécution de programmes, nous avons réduit le traitement des entrées/sorties au minimum, c'est-à-dire à la conversion de données symboliques en données codées, en laissant le soin au système d'exploitation "hôte" de résoudre les problèmes de gestion et d'accés aux périphériques.

Cependant, il faut remarquer le compromis "programmation-microprogrammation" trouvé dans la réalisation des entrées-sorties évoluées: toutes les conversions sont microprogrammées, ce qui représente un avantage considérable sur une machine classique dans laquelle elles sont programmées.

CHAPITRE 2

PRESENTATION GENERALE DE L'ARCHITECTURE de PASC-HLL

A - DEFINITION DU CODE-MACHINE

B - MECANISME D'INTERPRETATION

A - DEFINITION D'UN CODE MACHINE ADAPTE A LA COMPILATION ET A L'EXECUTION DES LANGAGES DE HAUT NIVEAU A STRUCTURE DE BLOCS (ET SPECIALEMENT PASCAL).

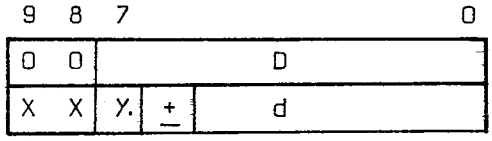
Tout ordinateur est défini, en grande partie, par son code machine dont les caractéristiques influent grandement sur les performances des programmes (facilité d'écriture, concision des programmes, encombrement des programmes exécutables, fiabilité software, rapidité d'exécution). Dans le cas précis qui nous concerne d'une machine langage, le concepteur a l'avantage de connaître le langage de haut niveau qu'il aura à compiler et peut choisir un code machine adapté à la compilation puis à l'exécution des programmes compilés. Les critères de choix de ce code machine sont complexes et font appel à deux notions principales qui s'avèrent être étroitement liées entre elles:

- la première notion est celle de compaction du code généré. En effet, un programme "réside" sur des supports d'information généralement coûteux (disques, bandes ...) et ne passe qu'une infime partie de sa "vie" en mémoire centrale. On a donc tout intérêt à obtenir des programmes exécutables de plus faible encombrement possible, pour réduire les coûts de "résidence" en mémoire secondaire et les temps de transfert en mémoire centrale.

Comment peut-on compacter un programme?

Une instruction étant constituée par un code opération et des opérandes, la taille d'une instruction pourra être optimisée en réduisant la taille des opérandes et leur nombre. Sachant que les instructions les plus fréquentes référencent une (ou plusieurs) variable(s), on cherche à définir un espace d'adressage minimum pour ces variables.

Concrétement, dans le cas d'un langage à structure de blocs, des mesures sur des programmes ont montré qu'il était suffisant de pouvoir référencer jusqu'à 256 variables dans le bloc principal, et jusqu'à 64 variables et 64 paramètres dans les blocs (ou procédure) intermédiaires. D'autre part, en ce qui concerne le langage PASCAL, une imbrication statique des procédures pouvant aller jusqu'à 6 niveaux est suffisante . En conséquence, l'espace d'adressage des variables d'un programme peut être limité à 10 bits, en spécifiant un niveau lexicographique et un déplacement de la manière suivante:



$0 \leq D \leq 255$

$-64 \leq d \leq +63,$

Cette compaction de l'adresse d'un opérande permet de réduire à 16 bits la taille des instructions d'accés aux opérandes (6 bits pour le code opération et 10 bits pour le couple (niveau, déplacement)).

Une deuxième solution pour réduire la taille d'une instruction est de limiter le nombre de ses opérandes explicites, en définissant des opérandes implicites. Cette réduction est obtenue en choisissant par exemple un code machine de structure post-fixée ou polonaise , dans laquelle apparaissent seulement deux types d'instructions qui sont les instructions d'accés à un opérande dont l'adresse est explicite et des opérateurs (unaires ou binaires) dont les opérandes sont implicites.

exemple:

l'expression A + B * C

est compilée en	Accés A	2 octets
	Accés B	2 octets
	Accés C	2 octets
	*	1 octet
	+	1 octet
		8 octets

Si l'on compile l'expression précédente dans une code classique (IBM 360), on obtient dans le cas le plus favorable d'optimisation par le compilateur:

L R,B	4 octets
M R,C	4 octets
A R,A	4 octets
	12 octets

ce qui fait déjà apparaître un facteur de compaction important, mais non significatif pour un programme complet

Sans optimisiation:	L R1,A	4 octets
	L R2,B	4 octets
	M R2,C	4 octets
	AR R1,R2	2 octets
		14 octets

Cette première méthode de compaction a des effets sensibles mais limités, en particulier on ne pourra jamais réduire l'encombrement d'une valeur immédiate donnée. Cependant, il est possible de définir plusieurs types de valeurs immédiates. En effet, des mesures ont montré qu'environ 95% des valeurs immédiates référencées dans un programme peuvent être codées sur 8 bits, et que les constantes égales à "zéro" ou à "un" sont très fréquemment référencées. On peut donc compacter le code en introduisant les instructions suivantes:

ZERO (préfixe)	1 octet
UN (préfixe)	1 octet
LIT8 (préfixe, valeur)	2 octets
LIT16 (préfixe, valeur)	4 octets

dont l'encombrement est inversement proportionnel à la fréquence d'utilisation et où le paramètre "préfixe" indique le "type" de la constante.

L'espace d'adressage des instructions peut également être étudié afin de réduire l'encombrement des instructions de branchement. En effet, tout programme écrit en langage de haut niveau est naturellement structuré, d'abord en procédures (ou blocs), puis en groupes d'instructions délimités par des instructions de test ou de boucles. Si l'on conserve cette structuration au niveau du code machine, on aboutit à la notion de segments de contrôle, qui peuvent s'imbriquer les uns dans les autres comme des blocs. On utilise alors un système d'adressage implicite qui permet de référencer implicitement soit le début, soit la fin du segment.

Exemple:

la structure while <exp> do <instruction> peut être compilée en:

```
LOOP (m)  <-------------- 2 octets
  <exp>
WHILE     <-------------- 1 octet
  <instruction>
ENLOOP    <-------------- 1 octet
```

Dans ce cas, l'instruction LOOP(m) définit le début et la fin d'un segment (par

son paramètre m). Les deux instructions WHILE et ENDLOOP référencent implicitement le début et la fin de la boucle et peuvent être codées sur seulement 8 bits. Dans un code classique, on aurait généré deux instructions BRANCH, codées chacune sur 4 octets.

On remarque, sur cet exemple, un facteur 2 pour la réduction de l'encombrement statique, et un facteur 4 pour l'accés dynamique aux instructions au cours de l'exécution: en effet, quand on "tourne" dans la boucle, seules les instructions WHILE et ENDLOOP sont accédées en mémoire centrale où elles n'occupent que deux octets au lieu de 2 instructions BRANCH qui occupent 8 octets.

- la deuxième notion importante pour la définition d'un code machine "compact" concerne la puissance des instructions, ou plus précisément leur aptitude à exprimer les notions sémantiques complexes contenues dans le langage de haut niveau. Le problème consiste à trouver, pour chaque notion sémantique évoluée, un ensemble d'instructions minimum qui soit cependant suffisant pour que la machine puisse, éventuellement en plusieurs étapes, exécuter toutes les opérations nécessaires à la réalisation de cette notion évoluée. Là encore, la compaction des instructions est obtenue en définissant des opérandes implicites et des structures de données évoluées, spécialement définies en fonction de leur utilisation par des instructions compactes.

Nous donnons ici plusieurs exemples d'application de cette notion.

Examinons d'abord le cas bien connu de l'INDEXATION d'un tableau, déclaré par exemple par:

```
type  TAB = array [3..15] of array [1..80] of  CHAR ;
var   A, B, C : TAB ;
```

Les trois variables A, B et C sont de type TAB, qui définit un tableau à deux dimensions dont les éléments sont de type "caractère" et occupent chacun 1 octet.

Dans un code classique, une référence à un élément du tableau, par exemple A[I,J] est compilée en:

```
      4    LA  R3, adresse de A
      4    LA  R0, adresse courante
      4    L   R1, borne sup de A (=15)
      2    CR  R1, R4            R4 contient l'INDEX I
1°    4    BH  erreur
dim   4    L   R1, borne inf de A (=3)
      2    SR  R1, R4
      4    BL  erreur
      4    M   R1, pas de A (=80)
      2    AR  R3, R1       R3 contient l'@ de A

           LA  R0, adresse courante
           L   R1, borne sup de A (=80)
           CR  R1, R5       R5 contient l'INDEX J
           BH  erreur
2°         L   R1, borne inf de A(=0)
dim        SR  R1, R5
           BL  erreur
           M   R1, pas de A(=1)   inutile
           AR  R3, R1
```

et toutes ces instructions sont nécessaires si l'on veut vérifier que l'index est bien compris entre les bornes inférieure et supérieure.

Sans cette vérification on obtient:

```
           LA  R3, adresse de A          4
           L   R1, borne inf             4
1°         SR  R1, R4                    2
dim        M   R1, pas de A              4
           AR  R3, R1                    2

           L   R1, borne inf
2°         SR  R1, R5
dim        AR  R3, R1
```

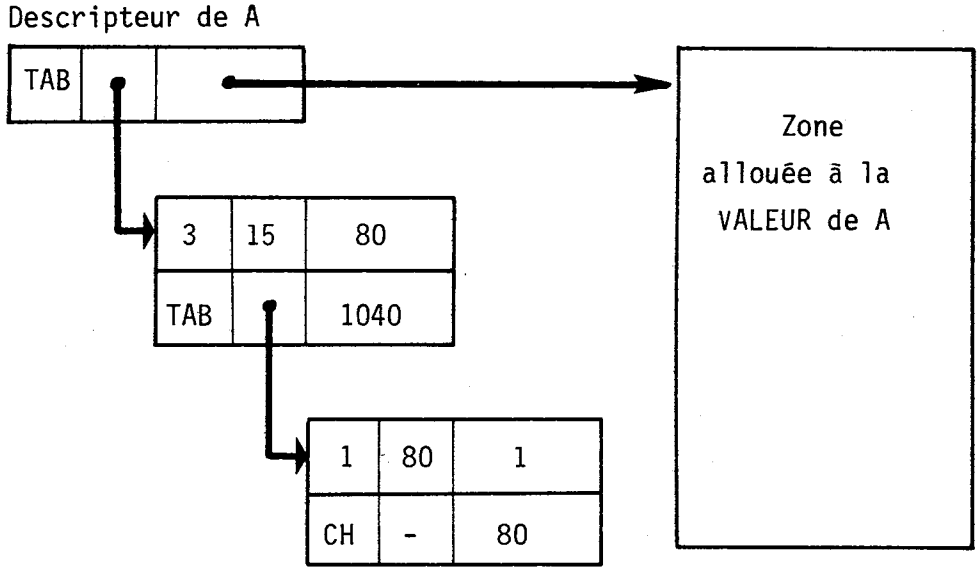
On remarque donc que pour chaque calcul d'indexation, il faut environ 4/5 instructions sans vérification et une dizaine d'instructions avec vérification, soit environ 16 ou 32 octets de code en langage 360.

Par contre, la solution adoptée dans la machine PASCHLL repose sur la notion de descripteur de variable qui fait référence à un descripteur de type contenant toutes les informations nécessaires au calcul d'index.

Reprenons l'exemple précédent. La référence A[I,J] est compilée en:

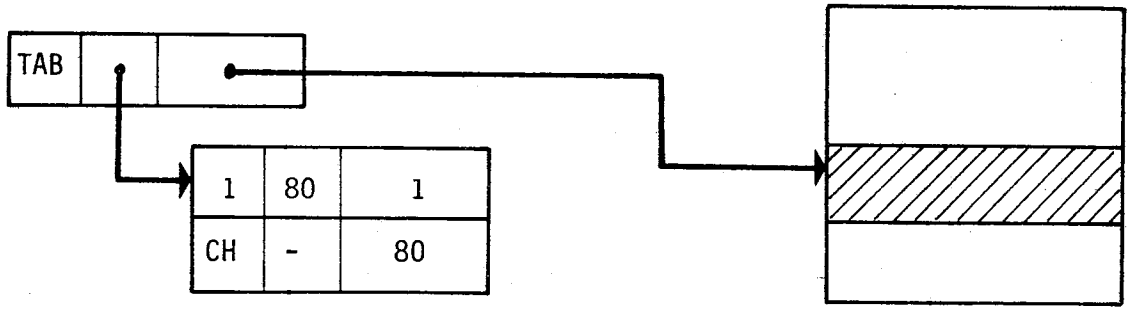
Accés A
Accés I
INDEX
Accés J
INDEX

L'instruction "Accés A" accède au descripteur de la variable A qui a la structure suivante:

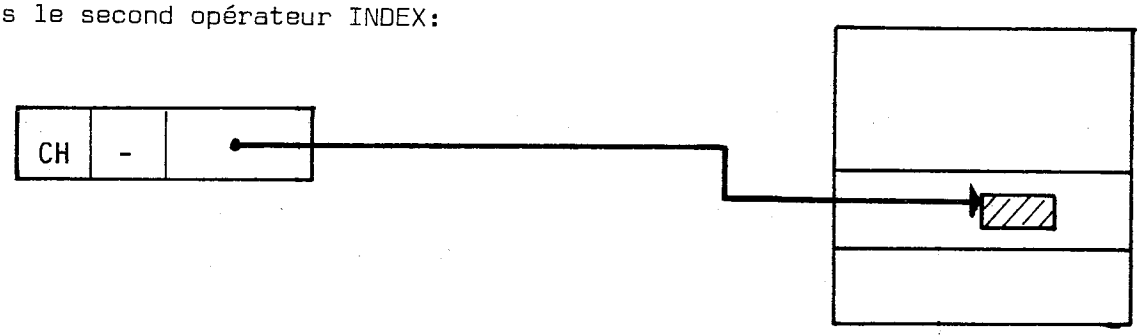


L'opérateur INDEX est un opérateur binaire qui a comme premier opérande un descripteur de variable de type "tableau" et comme deuxième opérande un descripteur de variable de type "scalaire". Ses deux opérandes sont implicitement définis par la notation post-fixée. On voit que la machine, utilisant la structure des informations définie par la figure, peut réaliser toutes les opérations de vérification des bornes, de multiplication par le pas, grace à la seule instruction INDEX, qui fournira comme résultat le descripteur de l'élément du tableau.

Ainsi, après l'exécution du premier opérande INDEX, on obtient:



et après le second opérateur INDEX:



On remarquera que le facteur de compaction est très important dans ce cas: le descripteur d'une dimension occupe 8 octets, et chaque référence nécessite un seul octet de code (opérateur INDEX). Le facteur de compaction peut donc aller jusqu'à 30 dans le cas d'une vérification des bornes.

Le deuxième exemple d'application de cette notion d'instructions puissantes est celui de l'appel de procédures, qui pour les langages à structure de blocs, est très important. On associe à chaque procédure un "descripteur" qui donne le nombre de ses paramètres et les descripteurs de chacun d'eux, le nombre de ses variables et leurs descripteurs.

On peut alors compiler l'appel de procédure suivant:

```
Call  P(I,J) ;
```

en

```
CALL P
Accés I
PARAM 1
Accés J
PARAM 2
ENTER P
```

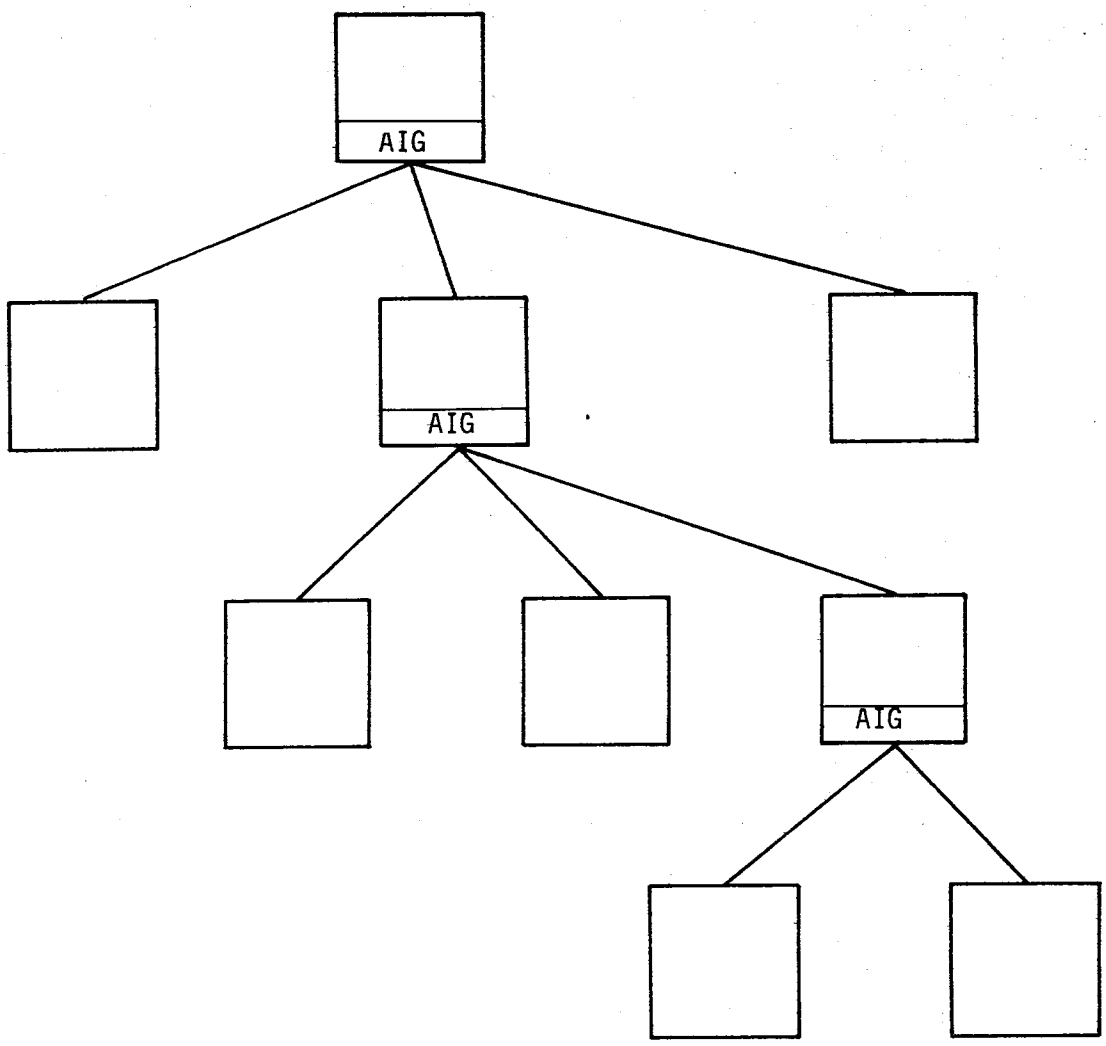
où l'instruction CALL P accéde aux descripteurs des paramètres et prépare leur "passage" qui sera réalisé par les instructions PARAM et où l'instruction ENTER P réalise l'entrée effective dans la procédure (sauvegarde de l'adresse de retour et branchement) et construit les descripteurs des variables locales à la procédure. La structure de données associée à chaque procédure permet, sans instructions supplémentaires, de vérifier que le nombre des paramètres et leurs types sont corrects et de réaliser les conversions éventuelles.

Un troisième exemple concerne la structure de donnée évoluée qui est celle du RECORD de PASCAL. Elle permet de décrire les "champs" d'une structure, pouvant avoir des "types" quelconques, et organisés en "branches" qui forment une arborescence, dans laquelle un seul chemin est défini à un moment donné de l'exécution.

exemple:

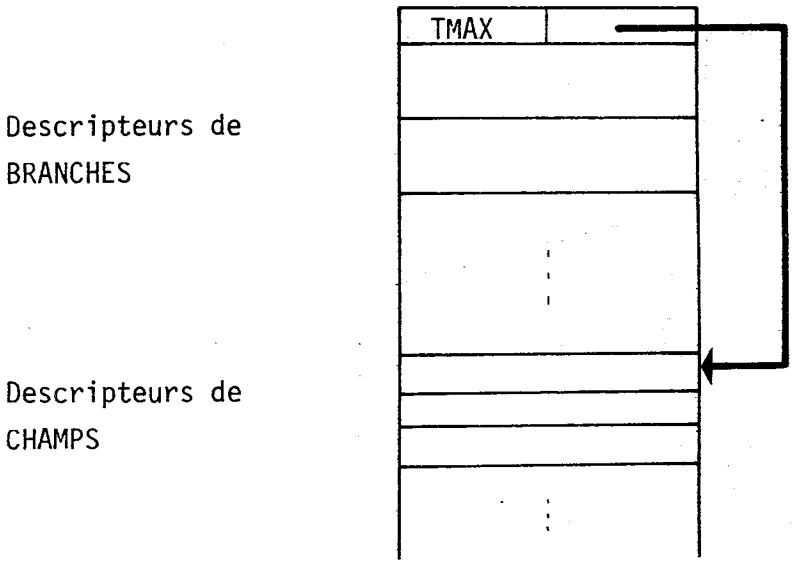
```
PERSONNE = record  NOM, PRENOM: alfa ;
                   NAISSANCE: date ;
                   case SEXE: (masc., féminin) of
                   masculin: (  ...   )
                   féminin: (  ...   )
           end
```

Généralement, un RECORD est un MODELE d'arborescence qui peut être représenté par la figure:



Le dernier champ de chaque "noeud" de l'arborescence peut définir, par sa valeur, un aiguillage vers les différents "fils" possibles. Durant l'exécution, les valeurs courantes des aiguillages définissent un sous-ensemble de l'arborescence, et seuls les champs qui lui appartiennent ont une existence, et sont donc accessibles.

Dans la machine PASC-HLL, à chaque type "RECORD" est associée la structure de données suivante:



Chaque descripteur de BRANCHE a le format:

FILS	FRERE	PERE	AIG
TC	NBE	ADE	

et chaque descripteur de CHAMP a le format

PF	PT	BR	DEP

L'ensemble des descripteurs de BRANCHE décrit complètement l'arborescence: elle peut être "descendue" en utilisant les chaînages FILS et FRERE, ce qui permet de calculer la taille exacte associée à chaque chemin en otant la taille corrigée TC à la taille maximum TMAX pour les besoins de l'instruction d'allocation dynamique. L'arborescence peut également être "remontée", en utilisant le chaînage BR du descripteur de CHAMP et le chaînage PERE des descripteurs de BRANCHE, lorsqu'on veut vérifier qu'un CHAMP existe, c'est-à-dire que l'aiguillage d'adresse AIG dans la structure a bien une valeur qui appartient à la liste d'étiquettes dont le nombre est NBE et l'adresse ADE.

CONCLUSION

Ce bref exposé concernant le code machine de PASC-HLL montre les avantages obtenus pour la compaction du code généré (qui peut globalement approcher d'un facteur 4) et fait apparaître une tentative d'introduction d'instructions spécialisées puissantes qui, associées à des structures de données adéquates, transfèrent au niveau de la machine l'exécution d'opérations qui sont spécifiées par des instructions pour un code classique (style IBM 360). On assiste donc à un remplacement de la programmation classique par la microprogrammation d'instructions plus puissantes, ce qui présente également un intérêt pour la rapidité d'exécution.

Nous prendrons comme dernier exemple celui de la conversion d'une chaîne de caractères représentant un nombre décimal en sa valeur binaire. Dans une machine classique, cette conversion est programmée et nécessite l'exécution d'un nombre important d'instructions classiques. Dans la machine PASC-HLL, lorsque l'affectation d'une chaîne de caractères à une variable entière est rencontrée, (une seule instruction de deux octets) cette conversion est faite automatiquement par un microprogramme, et en particulier, chaque multiplication par la base 10 est réalisée en deux microinstructions, soit environ 300 nanosecondes.

Un dernier aspect du code machine est celui de la fiabilité qui peut être obtenue durant l'exécution d'un programme: à la valeur de chaque variable sont associés un indicateur de type "hardware" (entier, réel, tableau,...), un indicateur de type "software" (entier compris entre 0 et 15 par exemple) et un indicateur d'initialisation de cette valeur qui permet de vérifier que toute valeur lue a bien été initialisée. Ces caractéristiques pénalisent un peu la rapidité d'exécution, mais elles sont indispensables au moins dans les étapes de mise au point, et elles peuvent être choisies "en option".

Des prévisions sur cet aspect langage sont donnés dans la description des trois processeurs qui sont chacun concernés par des aspects différents de l'interprétation (exemple: voir le processeur PAC pour les appels/retours de procédure, voir le processeur POP pour l'indexation...).

B - LA DEFINITION D'UN MECANISME D'EXECUTION EN "PIPE-LINE" DU CODE MACHINE DE TYPE POST-FIXE, UTILISANT UNE FILE D'ATTENTE PLUTOT QU'UNE PILE

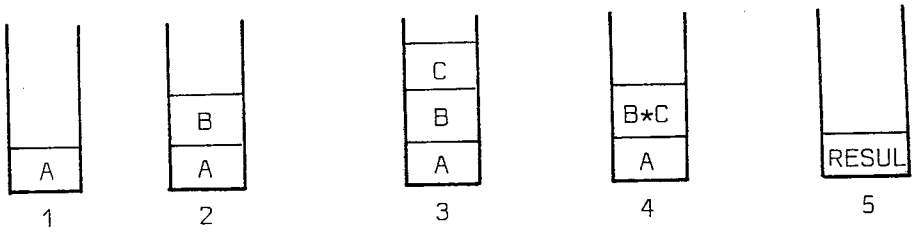
I - INTRODUCTION

L'exécution d'un code machine post-fixé est quasi automatique si l'on utilise une PILE et certaines machines sont réalisées selon ce principe (B6500, HP3000...). Cependant, l'utilisation d'une PILE impose un traitement séquentiel des instructions d'accés (opération PUSH) et des opérateurs qui prennent leurs opérandes au sommet de la pile et y rangent leur résultat.

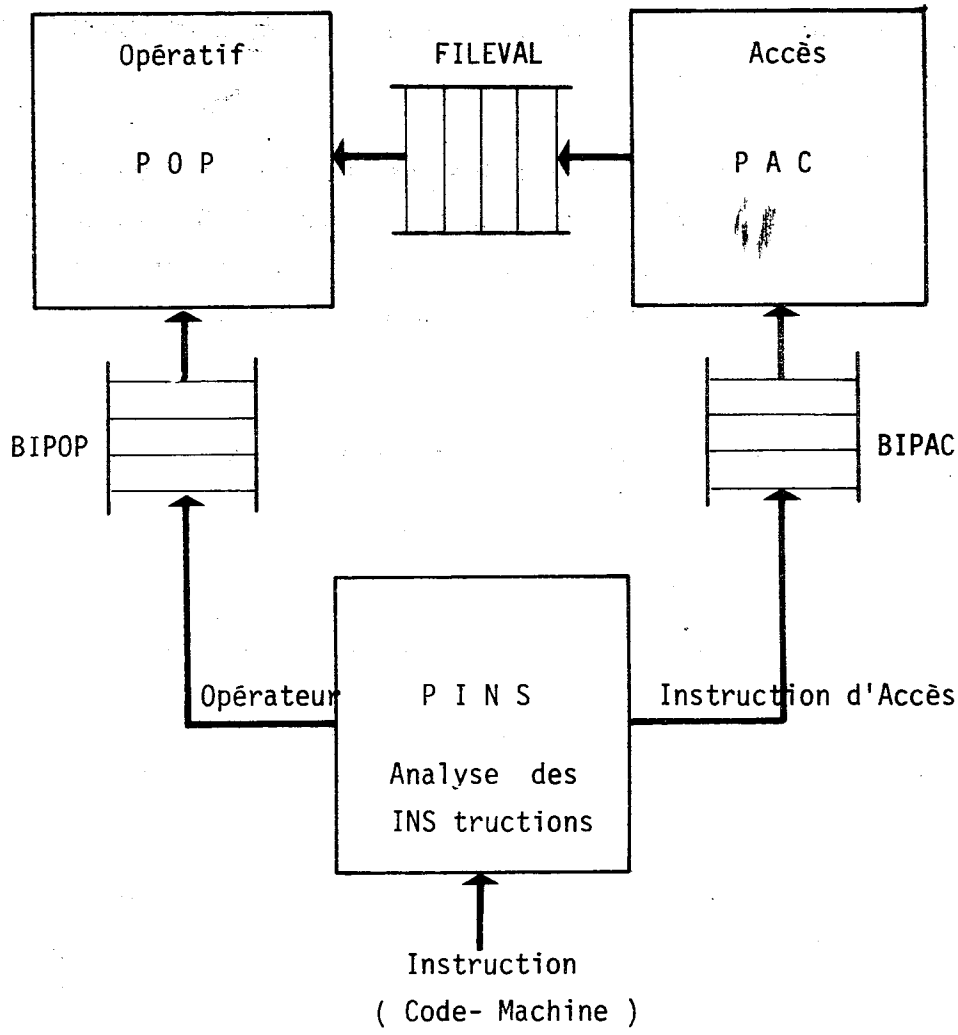
exemple:

exécution de A+B*C

```
Code  PUSH A
      PUSH B
      PUSH C
      MULT
      ADD
```



Un des aspects importants de l'étude de la machine PASC-HLL a porté sur la définition d'un nouveau mécanisme d'exécution d'un code post-fixé qui utilise une file d'attente à la place de la pile. Un tel mécanisme présente en effet l'avantage d'autoriser le parallélisme entre l'accés aux opérandes, leur rangement dans la file d'attente et l'exécution des opérateurs qui extraient leurs opérandes de cette même file d'attente. Qui dit parallélisme, dit gain de performance théorique. Nous avons donc choisi de développer ce mécanisme afin de faire la preuve qu'il est possible d'obtenir de bonnes performances sans recourir à une complexité trop grande (quand on pense à certaines machines monstrueuses), en décomposant naturellement l'exécution d'un code post-fixé qui présente les avantages cités dans la première partie.



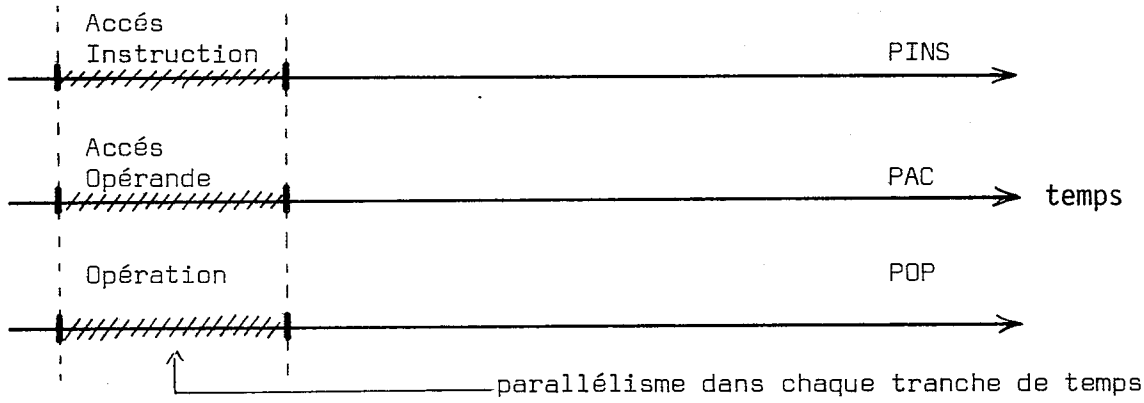
L'organisation décrite par la figure fait apparaitre trois processeurs spécialisés appelés PINS, PAC et POP.

Le rôle du processeur PINS est d'accéder aux INStructions en mémoire centrale et de les analyser pour envoyer les instructions d'ACcés au processeur PAC, à travers la file d'attente BIPAC, et pour envoyer les OPérateurs au processeur POP, à travers la file d'attente BIPOP. Ces deux files d'attente (BIPAC et BIPOP) sont réalisées par des composants électroniques spéciaux (First IN, FIRST OUT Buffer Memory, 16 mots de 4 bits) indiquant l'état VIDE pour le lecteur et l'état PLEIN pour l'écrivain, et assurant la gestion automatique de la file d'attente.

Le processeur PINS a également pour rôle d'exécuter les instructions de contrôle du programme, qui définissent les débuts et fins de boucle, les branchements, les appels de procédure,...

Cette structure globale permet d'illustrer les possibilités de parallélisme d'une telle architecture: au même instant, un processeur (PINS) exécute une étape de l'accés aux instructions, un second (PAC) exécute une étape de l'accés à un opérande, un troisième(POP) exécute une étape d'une opération arithmétique et logique.

Le diagramme séquentiel classique d'un processeur unique est remplacé dans le cas de PASC-HLL par les diagrammes parallèles:



Il est à remarquer la désynchronisation qui existe entre les trois processeurs qui opèrent au même instant sur des données indépendantes: l'instruction d'accés lue en mémoire par PINS à l'heure H ne sera exécutée par le processeur PAC qu'à l'heure $(H+d_{PAC})$, et l'opérateur lu en mémoire par PINS à l'heure (H+1) ne sera exécutée par le processeur POP qu'à l'heure $(H+1+d_{POP})$.

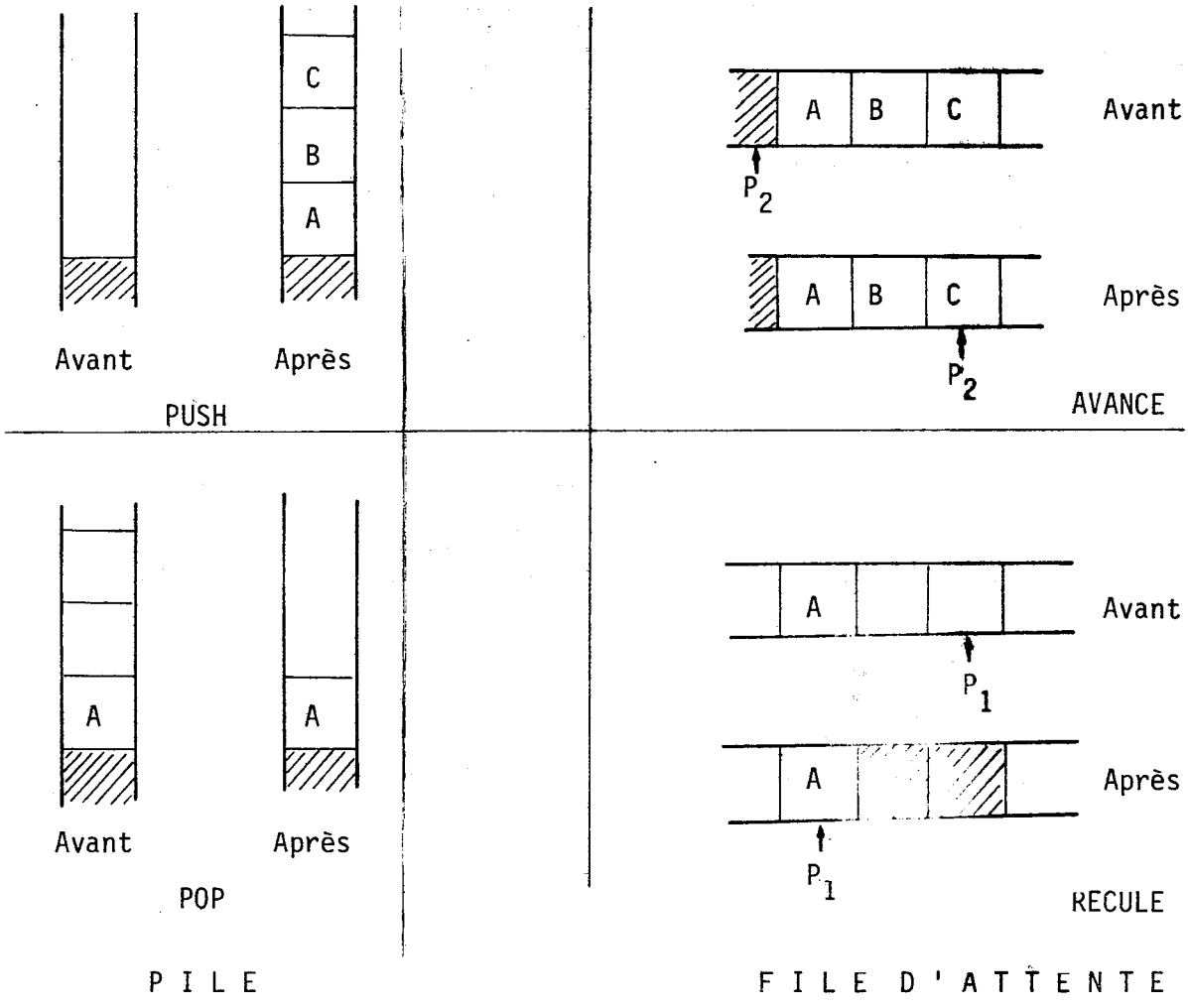
On remarque qu'il faut que $(d_{POP}+1)$ soit supérieur à d_{PAC}, ce qui signifie que l'opérande accédé par PAC est présent dans la file d'attente FILEVAL lorsque POP désire l'utiliser comme opérande.

Cette désynchronisation ne pose pas de problèmes de synchronisation, par le fait que les files d'attente jouent le rôle de tampons et que leur gestion est automatique.

La présentation du mécanisme d'exécution a été volontairement simplifiée dans l'exposé précédent. Nous allons maintenant définir plus précisément la gestion de la file d'attente des opérandes FILEVAL remplie par PAC et vidée par POP.

L'extraction des opérandes par le processeur POP n'est pas automatique et ne suit pas les règles simples des files d'attente à cause de la structure post-fixée du code machine. En effet, les opérandes ne sont pas utlisés dans le même ordre qu'ils sont rangés: on commence par ranger un nombre n d'opérandes, puis on utilise les deux derniers comme opérandes d'un opérateur binaire, etc... Il est donc nécessaire de "sauter" par dessus un nombre n d'opérandes, ce que nous appellerons AVANCER, ou au contraire RECULER dans la file d'attente pour récupérer des opérandes restés en attente.

L'opération AVANCER de n est équivalente à n opérations PUSH sur une pile, alors que l'opération RECULER est équivalente à la récupération d'un opérande resté en attente au fond de la pile.



En conséquence, les deux opérandes d'un opérateur binaire sont repérés par deux "pointeurs" P_1 et P_2, et le processeur PINS analyse la structure du code machine post-fixé, de manière à prévoir l'image théorique de la file d'évaluation et à pouvoir ainsi insérer des "extra-ordres"

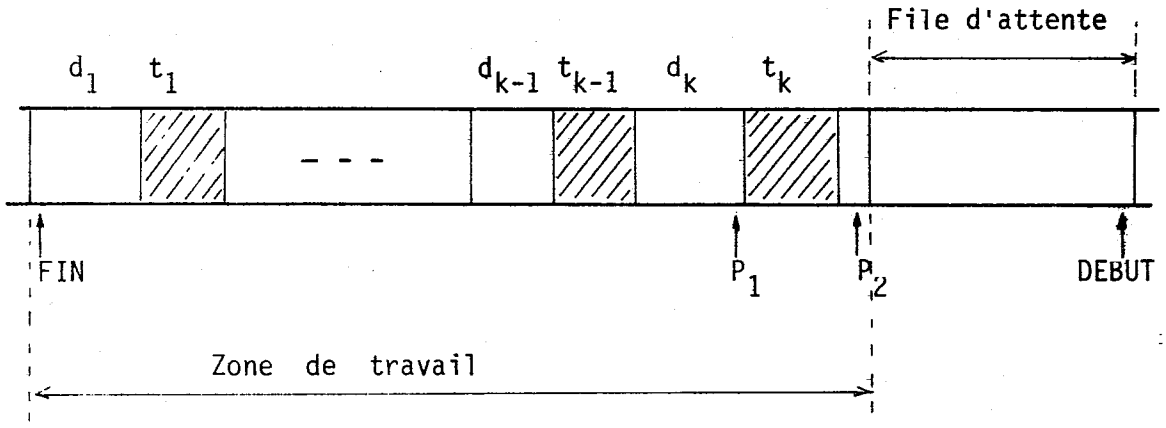
AVANCE(n) = "$P_2 \leftarrow P_2+n$ et $P_1 \leftarrow P_2-1$"

et

RECULE(n) = "$P_1 \leftarrow P_1-n$"

entre les opérateurs envoyés dans BIPOP.

Image théorique de la file d'évaluation



On voit sur le schéma que lorsque tous les opérandes du groupe de largeur d_k ont été utilisés, il faut générer un ordre RECUL (t_{k-1}) et, qu'inversement, un nouveau groupe de largeur d_{k+1} peut être rencontré qui conduit à la génération d'un ordre AVANCE(d_{k+1}).

Les deux suites $\{d_k\}$ et $\{t_k\}$ sont tenues à jour par le processeur PINS en fonction de la suite des instructions d'accés et des opérateurs qui sont trouvés dans le code.

II - INTRODUCTION D'UN MODELE POUR UNE CHAINE POST-FIXEE

I/- DEFINITIONS

Une chaîne post-fixée représente la traduction d'une EXPRESSION arithmétique ou logique définie dans un langage de haut niveau, en une suite d'INSTRUCTIONS d'ACCES notées a_i^j et d'OPERATEURS notés o_i^j.

DEFINITION D'UN GROUPE

On appelera GROUPE une séquence d'instructions d'accés et d'opérateurs ayant la structure suivante:

$$G_i = \{a^1_i, \ldots, a^{n_i}_i, o^1_i, \ldots, O^{m_i}_i\}.$$

Les entiers n_i et m_i représentent respectivement le nombre d'instructions d'accés et le nombre d'opérateurs du groupe G_i.

Le modèle utilisé pour une chaîne post-fixée est:

$$\{G_1, G_2, \ldots, G_p\}$$

où p représente le nombre de groupes.

Exemple:

chaîne post-fixée: A, B, C, +, D, *, E, +, /

obtenue à partir de: A/((B+C)*D+E)

Cette chaîne est composée de 3 groupes:

G_1 = {A, B, C, +}

G_2 = {D, *}

G_3 = {E, +, /}

DEFINITION DU RANG D'UNE INSTRUCTION D'ACCES

La notion de RANG d'une instruction d'accés dans l'expression post-fixée correspond à l'ordre dans lequel les instructions d'accés sont envoyées au processeur d'accés, et donc à l'ordre dans lequel les données accédées seront rangées dans la file d'évaluation. Ceci est une hypothèse. Ce rang peut être défini par la fonction:

$$RA(a_i^j) = \sum_{k=0}^{i-1} n_k + j \ , \quad n_0 = 0$$

DEFINITION DU RANG D'UN OPERATEUR

Cette notion de RANG d'un opérateur dans l'expression post-fixée correspond à l'ordre dans lequel les opérateurs sont envoyés au processeur opératif, et donc à l'ordre dans lequel ces opérateurs seront exécutés. Ceci est une hypothèse.

Ce rang peut être défini par la fonction:

$$RO(O_i^j) = \sum_{k=0}^{i-1} m_k + j \ , \quad m_0 = 0$$

PROPRIETES DE LA FONCTION RA ET FONCTION P

Par hypothèse de fonctionnement, on associe directement au rang $RA(a_i^j)$ d'une instruction d'accés a_i^j la position dans la file d'évaluation de la donnée d_i^j accédée par l'instruction d'accés. On introduit la fonction P, qui donne la POSITION de la donnée accédée par:

$$P(d_i^j) = RA(a_i^j)$$

Remarque: on aurait pu introduire la fonction d'accés A qui associe d_i^j à a_i^j:

$$A(a_i^j) = d_i^j$$

On aurait alors $P(A(a_i^j)) = RA(a_i^j)$

donc $AOP = RA$

Conséquence: les propriétés de la fonction $P(d_i^j)$ sont les mêmes que celles de la fonction $RA(a_i^j)$.

(P1) Propriété: La première donnée accédée se trouve en position 1

$$P(d_1^1) = 1$$

(P2) Propriété: La position de la dernière donnée accédée est égale au nombre N d'instructions d'accés

$$P(d_p^{n_p}) = \sum_{k=0}^{p} n_k = N$$

(P3) Propriété: Les dernières données correspondant à deux groupes consécutifs G_{i-1} et G_i sont distantes de n_i positions,

$$P(d_i^{n_i}) - P(d_{i-1}^{n_{i-1}}) = n_i$$

en effet

$$P(d_n^{n_i}) = \sum_{k=0}^{i-1} n_k + n_i = P(d_{i-1}^{n_{i-1}}) + n_i$$

(P4) Propriété: Deux données correspondant à deux instructions d'accés consécutives sont distantes de 1 position:

- dans un même groupe

$$P(d_i^j) = P(d_i^{j-1}) + 1 \qquad (j > 1)$$

- pour deux groupes consécutifs

$$P(d_i^j) = P(d_{i-1}^{n_{i-1}}) + 1 \qquad (j = 1)$$

RELATION ENTRE LES FONCTIONS RO et RA

Une propriété caractéristique des expressions post-fixées peut être exprimée par la formule suivante:

(P5) $\forall i, \forall j, \quad RO(O_i^j) \leq RA(a_i^{n_i}) - 1$

Cette propriété est due au fait qu'un opérateur binaire utilise deux opérandes et que son résultat est utilisé comme opérande d'un opérateur suivant. Cette relation se décompose en:

$$m_1 \leq n_1 - 1,\ m_1 + m_2 \leq n_1 + n_2 - 1,\ \ldots \text{ et plus généralement } \Sigma m_k \leq \Sigma n_k - 1$$

Conséquence: les deux suites $n_1, \ldots, n_p$ et $m_1, \ldots, m_p$ doivent toujours vérifier les relations précédentes pour que la suite $G_1, \ldots, G_p$ puisse être considérée comme une expression post-fixée.

Remarque: la présence possible d'opérateurs unaires explique l'existence de l'inégalité dans la relation précédente.

RELATION ENTRE UN OPERATEUR ET LA POSITION DE SES OPERANDES

La chaîne post-fixée étant donnée, le problème consiste à trouver deux fonctions $P1(O_i^j)$ et $P2(O_i^j)$ qui donnent respectivement la position du premier et du deuxième opérande de l'opérateur binaire O_i^j, sachant que les positions des données sont définies par la fonction $P(d_i^j)$.

Hypothèse: on suppose que le résultat d'un opérateur binaire est rangé dans la file en une position $R(O_i^j)$ qui est égale à la position $P2(O_i^j)$ du deuxième opérande de l'opérateur.

(P6) $\forall i, \forall j : R(O_i^j) = P2(O_i^j)$

Remarque: cette hyptohèse répond à un double souci de récupération d'une position dans la file, et de progression de la position des résultats intermédiaires, afin de "vider la file par le bas".

DEFINITION DE LA FONCTION P2

La fonction P2 donne la position du deuxième opérande binaire. Elle est définie par les régles d'évaluation de la manière suivante:

- le deuxième opérande du premier opérateur O_i^1 du groupe G_i est la dernière donnée $d_i^{n_i}$ de ce groupe.

Soit $P2(O_i^1) = P(d_i^{n_i})$

- le deuxième opérande d'un opérateur O_i^j, $j > 1$, du groupe G_i, est le résultat de l'opérateur précédent O_i^{j-1} .

D'après P6, on sait que $R(O_i^1) = P2(O_i^1)$, donc

$\forall i \in \{1,p\}, \quad \forall j \in \{1,m_i\}$

(P7) $\qquad P2(O_i^j) = P(d_i^{n_i})$

Conséquence: la fonction P2 est une fonction en escalier qui change de valeur lorsqu'on change de groupe.

Une transition se produit entre $O_i^{m_i}$ et O_{i+1}^1.

On passe alors de $P2(O_i^{m_i}) = P(d_i^{n_i})$ à $P2(O_{i+1}^1) = P(d_{i+1}^{n_{i+1}})$, soit une progression de $P(d_{i+1}^{n_{i+1}}) - P(d_i^{n_i}) = \sum_{k=0}^{i+1} n_k - \sum_{k=0}^{i} n_k = n_{i+1}$ positions.

GENERATION DE LA FONCTION P2

Nous associons à la fonction P2 une variable (au sens algorithmique) ou pointeur appelé P2 dont la valeur est égale à celle de la fonction.

La génération de la fonction peut alors être exprimée en termes de modifications de la valeur de la variable associée.

La réalisation de cette fonction suppose qu'on soit capable de:

- détecter le passage d'un groupe à un autre (transition $O_i^{m_i} \rightarrow a_{i+1}^1$) ;
- compter le nombre n_{i+1} d'instructions d'accés dans le groupe G_{i+1} ;
- insérer entre $O_i^{m_i}$ et O_{i+1}^1 un ordre spécial dont l'exécution par le processeur opératif ait pour effet de modifier P2 de la manière suivante: $P2 \leftarrow P2 + n_{i+1}$

Cet ordre est appelé AVANCE et la suite des ordres envoyés au processeur opératif est donc:

$$\begin{array}{l} \text{AVANCE}(n_1) \\ \quad O_1^1 \\ \quad \vdots \\ \quad O_1^{m_1} \\ \text{AVANCE}(n_2) \\ \quad \vdots \\ \quad O_i^{m_i} \\ \text{AVANCE}(n_{i+1}) \\ \quad O_{i+1}^1 \\ \quad \vdots \end{array}$$

On suppose de plus que la valeur initiale de P2 est égale à 0, et on est sûr que la succession des valeurs de P2 sera conforme à la propriété (P7).

DEFINITION DE LA FONCTION P1

La fonction P1 donne la position du premier opérande de l'opérateur binaire O_i^j. Son expression n'est simple que pour un cas particulier relatif au premier opérateur O_i^1 d'un groupe.

En effet, d'après les règles d'évaluation, on a

$$P1(O_i^1) = P(d_i^{n_i-1}) \quad \text{si} \quad n_i > 1$$

$$P1(O_i^1) = R(O_{i-1}^{m_{i-1}}) \quad \text{si} \quad n_i = 1 \text{ et } i > 1$$

D'après la propriété (P6), on a toujours

(P8) $$P1(O_i^1) = P2(O_i^1) - 1$$

En effet

$$1/\quad P(d_i^{n_i-1}) = P(d_i^{n_i}) - 1 = P2(O_i^1) - 1$$

$$2/\quad R(O_{i-1}^{m_{i-1}}) = P2(O_{i-1}^{m_{i-1}}) = P(d_{i-1}^{n_{i-1}}) = P(d_i^{n_i}) - n_i$$

$$= P2(O_i^1) - 1$$

Conséquence: La fonction P2 étant définie et l'ordre AVANCE étant connu, on peut l'utiliser pour générer une partie de la fonction P1 qui coïncide avec le premier opérateur O_i^1 d'un nouveau groupe.

En effet l'ordre AVANCE étant inséré entre $O_{i-1}^{m_{i-1}}$ et O_i^1, on peut exécuter:

$$P2 \leftarrow P2+n_i,$$

puis $$P1 \leftarrow P2-1,$$

opérations qui assurent que la propriété (P8) est vérifiée.

Règle: Les règles d'évaluation d'une expression impliquent qu'une donnée utilisée une fois comme premier opérande n'est plus utilisée. Cette propriété est"visible" quand on utilise une PILE: le premier opérande est écrasé par le résultat. Seul un résultat intermédiaire est conservé, jusqu'à ce qu'il soit à son tour utilisé comme premier opérande et disparaisse.

APPLICATION DE LA REGLE PRECEDENTE

La position définie par $P1(O_i^j)$ ne sera plus accédée après l'exécution de l'opérateur O_i^j, elle devient donc un TROU dans la file. On peut donc systématiquement décrémenter la variable P1 qui porte la valeur de la fonction.

$$P1 \leftarrow P1-1$$

Le problème consiste à savoir si la nouvelle valeur de P1 coïncide avec celle de la fonction.

Problème

Est-ce que $P1(O_i^j)-1 = P1(O_i^{j+1})$ si $j < m_i$ (E1)

ou $P1(O_i^j)-1 = P1(O_{i+1}^1)$ si $j = m_i$ (E2)

Le problème se découpe en deux cas:

1/ Le changement de groupe ($j=m_i$)

On peut affirmer que l'égalité (E2) est fausse parceque

$$P1(O_{i+1}^1) = P(d_{i+1}^{n_{i+1}-1}) \quad \text{si} \quad n_{i+1} > 1$$

et

$$P1(O_{i+1}^1) = R(O_i^{m_i}) = P(d_i^{n_i}) \quad \text{si} \quad n_{i+1} = 1.$$

Dans les deux cas, on a:

$$P1(O_i^{m_i}) < P(d_i^{n_i})$$

donc

$$P1(O_i^{m_i})-1 < P1(O_{i+1}^1)$$

Cependant, on sait qu'un changement de groupe est détecté entre $O_i^{m_i}$ et O_{i+1}^1 et qu'un ordre AVANCE(n_{i+1}) est généré, qui permet d'assurer que la nouvelle valeur de P1 est égale à celle de la fonction ($P1(O_{i+1}^1)=P2(O_{i+1}^1-1)$)

2/ A l'intérieur d'un groupe($j<m_i$)

On sait que $P1(O_i^1) = P2(O_i^1)-1$

$$= P(d_i^{n_i})-1$$

En décrémentant P1 n_i fois, la suite des valeurs de la fonction P1 sera:

$$P(d_i^{n_i-1}), P(d_i^{n_i-2}), \ldots, P(d_i^1), P(d_{i-1}^{n_{i-1}})$$

qui sont bien les valeurs de la fonction

$$P1(O_i^1),\ P1(O_i^2),\ \ldots,\ P1(O_i^{n_i-1}),\ P1(O_i^{n_i}).$$

Après l'exécution de $O_i^{n_i}$, tous les opérandes du groupe $G_i(d_i^{n_i-1}, \ldots, d_i^1)$ ainsi que le résultat du dernier opérateur $O_{i-1}^{m_{i-1}}$ du groupe précédent ont été utilisés.

On peut affirmer qu'il existe alors un TROU de largeur au moins égale à 1: il y a au moins un TROU en position $P(d_{i-1}^{n_{i-1}})-1$, TROU créé par l'opérateur $O_{i-1}^{m_{i-1}}$.

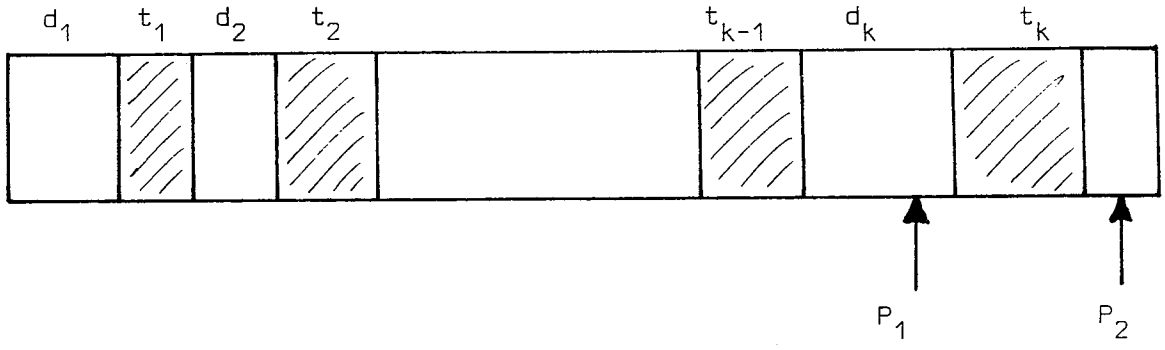
Remarque : ce problème ne se pose pas pour le premier groupe parce que $m_1 \leq n_1 - 1$ d'après (P5).

Conséquence : il est nécessaire de connaître la largeur et la localisation des TROUS créés par l'exécution des opérateurs précédents, afin de pouvoir calculer la fonction P1.

2/ ALGORITHME DE GESTION DES TROUS

Image théorique de la file

On introduit deux suites $\{d_k\}$ et $\{t_k\}$ qui représentent respectivement la largeur des zones occupées et celle des TROUS, sachant qu'on a une alternance entre les zones occupées et les TROUS.



On construit les deux suites $\{d_k\}$ et $\{t_k\}$ de manière à vérifier les égalités:

$$(E3) \qquad \sum_{k=1}^{k} d_i + \sum_{i=1}^{k} t_i = P(d_i^{n_i}) = P2(O_i^j)$$

et

$$(E4) \qquad \sum_{i=1}^{k} d_i + \sum_{i=1}^{k-1} t_i = P1(O_i^j)$$

La valeur de t_k représente la différence entre la valeur de la fonction P2 et celle de P1, pour tout opérateur O_i^j. En particulier pour l'opérateur O_i^1 (1er opérateur d'un groupe), on aura $t_k=0$ avant l'exécution et $t_k=1$ après.

Construction de l'image théorique

L'image théorique de la file est construite par le processeur de contrôle qui analyse la chaîne post-fixée et anticipe sur l'état de la file au moment de l'exécution. Cette anticipation peut être vue comme une exécution formelle qui n'est concernée que par les positions des opérandes et ne tient pas compte de leur valeur.

L'algorithme de construction des suites $\{d_k\}$ et $\{t_k\}$ est basé sur la détection des transitions.

1/ Transition $a_i^j \rightarrow a_i^{j+1}$

Le dernier élément d_k de la suite $\{d_k\}$ a pour valeur le nombre d'instructions d'accés déjà rencontrées dans le groupe G_i. Cette transition est interprétée par:

$$d_k \leftarrow d_k+1$$

et on envoie l'instruction a_i^{j+1} au processeur d'accés.

2/ Transition $a_i^{n_i} \rightarrow O_i^1$

Toutes les instructions d'accés du groupe G_i ont été analysées ; leur nombre est égal à la valeur de d_k puisqu'on fait n_i fois "$d_k \leftarrow d_k+1$" et si l'on suppose que d_k a été correctement initialisé.

On peut donc utiliser la valeur de d_k comme paramètre de l'ordre AVANCE qui est généré avant l'opérateur O_i^1.

Le processeur opératif recevra donc la séquence:

$$\vdots$$
$$m_{i-1}$$
$$O_{i-1}$$
$$\text{AVANCE}(d_k) \text{ avec } d_k=n_i$$
$$O_i^1$$
$$\vdots$$

L'opérateur binaire O_i^1 étant envoyé au processeur opératif, on sait que l'exécution de cet opérateur va avoir pour effet de créer un TROU d'une part et de décrémenter P1 d'autre part.

On exécute donc:

$$d_k \leftarrow d_k - 1 \quad \text{et} \quad t_k \leftarrow t_k + 1.$$

On est ainsi sûr que l'image de la file est bonne et que les égalités (E3) et (E4) sont vraies.

Preuve:

- la somme $\sum_{i=1}^{k} d_i + \sum_{i=1}^{k} t_i$ est inchangée puisque d_k a été décrémenté et t_k incrémenté.

Donc (E3) reste vraie.

- la somme $\sum_{i=1}^{k} d_i + \sum_{i-1}^{k-1} t_i$ est plus petite de 1, puisque seul d_k y apparait. Comme P1 a été décrémenté, (E4) reste vraie.

3/ Transition $O_i^j \rightarrow O_i^{j+1}$

Dans le cas où j est égal à n_i, on voit que les n_i variables du groupe G_i ont été utilisées comme premier opérande. Ce fait se traduit par "d_k=0".

Dans le cas où d_k est égal à zéro, la valeur de P1 correspond à la position d'un TROU, dont la largeur est égale à t_{k-1}.

Conséquence: pour que la valeur de P1 soit égale à $P1(O_i^{j+1})$ il faut générer un ordre appelé RECULE, d'un nombre de positions égal à la longueur du TROU.

$$\Rightarrow \text{RECULE}(t_{k-1})$$

Comme on avait $P1 = P1(O_i^j)$, on obtient $P1 = P1(O_i^j) - t_{k-1}$.

(E4) étant supposée vraie pour l'opérateur O_i^j on obtient qu'elle est vraie pour l'opérateur suivant:

$$P1 = \sum_{i=1}^{k} d_i + \sum_{i=1}^{k-1} t_i - 1 - t_{k-1} = \sum_{i=1}^{k-1} d_i + \sum_{i=1}^{k-2} t_i$$

L'état de la file doit être modifié, parce que deux TROUS sont maintenant adjacents ($d_k=0$).

On fait donc:

$t_{k-1} \leftarrow t_{k-1} + t_k$ compaction

$k \leftarrow k-1$ suppression

ce qui assure que (E3) est toujours vérifiée.

Résumé:

l'algorithme de la transition $O_i^j \rightarrow O_i^{j+1}$ est:

si $d_k=0$ alors début générer RECULE(t_{k-1})

$t_{k-1} \leftarrow t_{k-1} + t_k$

$k \leftarrow k-1$

fin

$d_k \leftarrow d_k - 1$

$t_k \leftarrow t_k + 1$

générer O_i^{j+1}

4/ Transition $O_i^{m_i} \rightarrow a_{i+1}^1$

Cette transition s'accompagne de la création d'un nouvel élement pour les suites $\{d_k\}$ et $\{t_k\}$, caractérisé par $d_k=1$ et $t_k=0$. L'instruction d'accés a_{i+1}^1 est envoyée au processeur d'accés.

3/ EVALUATION DU TAUX D'EXTRA-ORDRES

Considérons une expression composée de N opérandes répartis en p groupes.

Le nombre d'ordres AVANCE est égal à p: il y a génération d'un ordre AVANCE par groupe.

Pour l'évaluation du nombre d'ordres RECULE, il est nécessaire de raisonner par itération. Plaçons nous donc pendant l'exécution du groupe n°i: sachant que tout ordre RECULE compacte deux "trous", le nombre maximum de RECULES possibles pendant l'exécution du groupe i est au plus égal au nombre des groupes d'opérandes adjacents (i-1) moins le nombre de RECULES déjà effectués pendant l'exécution des groupes précédents.

Si l'on note k_i le nombre de RECULE effectués au cours de l'exécution du groupe i, on a la relation:

$$0 \leq k_i \leq (i-1) - \sum_{j=1}^{i-1} k_j$$

En faisant une sommation terme à terme sur tous les indices i, on obtient:

$$0 \leq \sum_{j=1}^{p} k_j \leq \sum_{j=1}^{p} (j-1) - \sum_{j=1}^{p-1} \left(\sum_{i=1}^{j-1} k_i \right)$$

qui peut s'exprimer sous la forme

$$\sum_{j=1}^{n} \sum_{i=1}^{j} k_i \leq \frac{n(n-1)}{2}$$

On obtient donc $\sum_{i=1}^{p} k_i$ par soustraction:

$$\sum_{j=1}^{p} \sum_{i=1}^{j} k_i - \sum_{j=1}^{p-1} \sum_{i=1}^{j} k_i = \sum_{i=1}^{p} k_i \leq \frac{p(p-1)}{2} - \frac{(p-1)(p-2)}{2} = p-1$$

Le nombre total d'ordres RECULE générés pour une expression composée de p groupes est donc compris entre 0 et (p-1). Cependant, l'existence de p groupes implique qu'il y a au moins p opérations sans génération d'un ordre

RECULE. Sachant que pour N opérandes, on a N-1 opérateurs binaires, il reste seulement N-1-p opérateurs binaires pouvant s'accompagner d'un RECULE.

Conclusion: pour que le maximum d'ordres RECULE puisse être atteint, il faut que la relation

$$N-p-1 \geq p-1 \quad \text{soit} \quad N \geq 2p$$

soit vérifiée.

Il faut donc distinguer les deux cas:

$$N<2p \quad \text{et} \quad N \geq 2p.$$

Posons $T = \dfrac{NB(AVANCE) + NB(RECULE)}{NB(OPERATEUR) + NB(OPERANDE)}$

On sait déjà que:

$$T = \frac{P + NB(RECULE)}{N + (N-1)}$$

en introduisant les valeurs minimum et maximum de NB(RECULE), et en considérant les deux cas, on obtient:

pour $N \geq 2p$ $\qquad \dfrac{p}{2N-1} \leq T \leq \dfrac{2P-1}{2N-1}$

pour $N \leq 2p$ $\qquad \dfrac{p}{2N-1} \leq T \leq \dfrac{N-1}{2N-1}$

Exemple:

L'expression (A+B)*(C+D)
donne la chaîne A,B,+,C,D,+,*,
caractérisée par 4 opérandes (N=4) et 2 groupes (p=2)
D'après la formule on obtient:

$$\frac{2}{7} \leq T \leq \frac{3}{7}$$

et un nombre RECULE compris entre 0 et 1.

Remarque:

Le taux maximum est atteint pour N=2p et il a pour valeur $\frac{2p-1}{4p-1}$: il reste donc inférieur à 50%, qui est asymptote, et sera dans la majorité des cas (expressions très simples) égal au minimum qui est $\frac{p}{2N-1}$.

De plus, cette évaluation ne tient pas compte des opérateurs UNAIRES, dont la présence peut impliquer des ordres AVANCE, mais aucun ordre RECULE.

Le taux d'extra-ordres sera donc généralement inférieur à l'évaluation précédente si l'on tient compte de la présence des opérateurs UNAIRES.

III - INTRODUCTION D'UN PROCESSEUR DE GESTION DE LA FILE

Un tel mécanisme d'accès à la file d'évaluation par l'intermédiaire de "pointeurs" conduit à l'utilisation d'une mémoire à accès aléatoire (RAM ou Random Access Memory) dont la gestion des adresses est réalisée par un processeur spécialisé, appelé FILE.

Ce processeur FILE reçoit les requêtes des deux processeurs PAC et POP, l'un pour ranger un nouvel opérande, l'autre pour extraire un opérande ou modifier un pointeur (ordre AVANCE et RECULE).

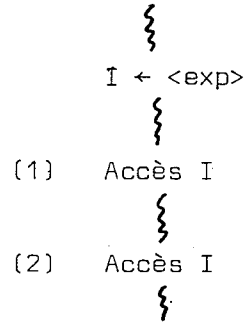
D'autre part, ce processeur FILE gère les adresses d'une autre file d'attente utilisée pour résoudre le problème des dépendances, appelée FILEDEP.

1/ Position du problème des dépendances:

A chaque fois qu'une modification d'une variable est détectée par le processeur PAC, le NOM(10 bits) de cette variable est noté dans cette file d'attente dont une partie est ASSOCIATIVE. Pour toute instruction d'accès spécifiant le NOM d'une variable, une recherche associative est faite (qui dure 30 nano-secondes) qui indique si la variable est sous dépendance ou si

elle vient d'être modifiée. Dans le premier cas, il est inutile d'aller chercher la valeur de cette variable puisqu'elle n'est pas à jour: on établit une indirection qui sera résolue par le processeur POP au moment de l'affectation. Dans le second cas, la valeur de la variable qui vient d'être modifiée est disponible dans la file des dépendances: un accès à la mémoire centrale est ainsi économisé. Cette file contient donc le WORKING-SET (de longueur 16) des variables référencées par le programme dans une tranche de temps déterminée.

Exemple:



Tant que l'affectation de <exp> à I n'est pas réalisée par le processeur POP, les accès à I ((1) et (2)) sont différés.

Formalisation du problème des dépendances:

L'explication de ce problème est compliqué à cause du décalage temporel entre les processeurs d'ACcès et OPératif.

Supposons qu'à l'heure H, le processeur PINS envoie l'instruction AFFECT(I) dans la file d'attente d'instructions BIPAC, et l'opérateur binaire AFFECT dans la file d'attente d'instructions BIPOP.

Le processeur PAC accède à cette instruction à l'heure H+a, et le processeur POP à l'heure H+b.

- Si H+b < H+a, le processeur POP est en avance et devra attendre que les opérandes relatifs à l'opérateur AFFECT aient été fournis par PAC. Il attend donc jusqu'à l'heure H+a+ε.

- Si H+b > H+a, un problème va se poser pour l'accès à la variable I dans l'intervalle de temps compris entre H+a et H+b: en effet la valeur de I n'a pas encore été modifiée par le processeur POP (elle le sera à l'heure H+b).

Solution:

- lorsque le processeur PAC reçoit une instruction AFFECT(I) il note le "nom" I (10 bits) dans la partie associative d'une file d'attente appelée "File des Dépendances".

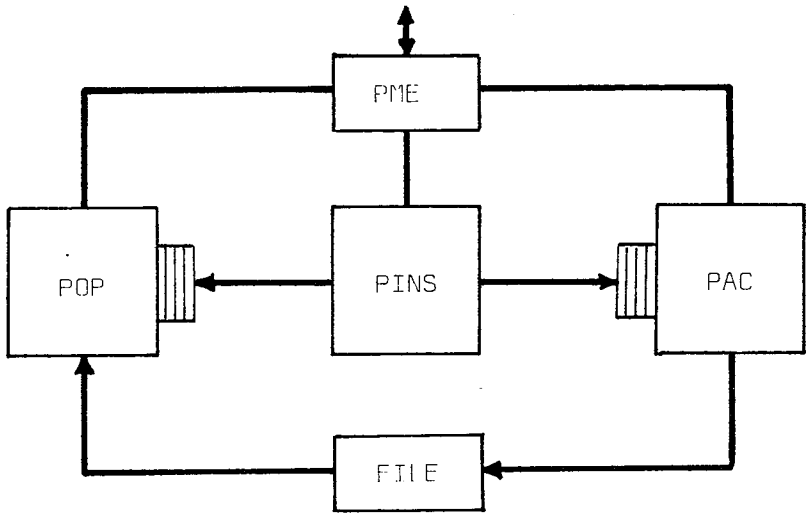
- lorsqu'il reçoit une instruction ACCES(I), il fait une recherche associative qui lui indique si cette variable est sous dépendance ou non.

En conséquence, à l'heure H+a, le nom I est noté dans la partie associative. Si une instruction d'accès à I est reçue entre l'heure H+a et l'heure H+b, le processeur PAC saura que la variable I est sous dépendance. En conséquence, il créera une indirection, de telle sorte que la position dans la file d'avaluation puisse être "remplie" par le processeur POP à l'heure H+b.

Par contre, à l'heure H+b, le processeur POP pourra "lever" la dépendance, en enlevant l'indicateur dans la file des dépendances et remplir les cases que PAC n'aura pas pu remplir, en parcourant les indirections qui sont construites comme une chaîne de reprise (comparable à celle qui est construite par un assembleur pour les étiquettes non encore définies).

De plus, l'existence d'une Mémoire Associative pour résoudre le problème des dépendances, est mise à profit pour la création d'un CACHE sur les dernières variables modifiées: après l'heure H+b, la valeur de la variable I devient directement accessible dans la file des dépendances. Elle le restera jusqu'à son écrasement (la taille de la file des dépendances a été limitée à 16 mots, les variables les plus anciennement modifiées seront donc écrasées selon un mode FI-FO).

on obtient le schéma complet de l'architecture de la machine PASC-HLL.



Cette rapide présentation de l'architecture de la machine PASC-HLL met en évidence une approche de décomposition fonctionnelle du processus d'interprétation d'un code machine de type post-fixé, qui fait apparaître principalement trois processeurs spécialisés dans des fonctions limitées et totalement différentes: l'accés aux instructions et le contrôle du déroulement du programme pour PINS, l'accés aux opérandes pour PAC, l'exécution des opérateurs pour POP.

Les performances d'une telle structure sont en théorie intéressantes, mais elles dépendent d'une part de la réalisation physique de chacun des processeurs, qui est présentée dans un chapitre suivant, et d'autre part, de l'aptitude à bien anticiper sur le choix des alternatives: ce problème est propre à toutes les machines "pipe-line" qui voient leur puissance théorique cruellement diminuée par les nombreux choix qui "vident le pipe-line" et anéantissent les efforts d'anticipation. Dans cette machine, nous avons l'avantage de connaître la structure des programmes qui permet de choisir l'alternative la plus probable: PASC-HLL se trompe d'alternative une fois sur N dans le cas où le programme "tourne" N fois dans une boucle, et la reprise d'erreur y est quasi immédiate.

2/ Expression du problème de l'anticipation

Lorsqu'à l'heure H le processeur PINS accède à une instruction de branchement conditionnel, il ne connait pas la valeur du prédicat (expression booléenne) qui permettrait de choisir la bonne alternative. En effet cette valeur doit être évaluée par le processeur opératif POP, qui la connaîtra seulement à l'heure H+a.

Le processeur PINS choisit donc, à l'heure H, l'alternative la plus probable, dans la mesure du possible, et continue son travail de préparation en accédant aux instructions de l'alternative choisie. Cependant, il se place dans un état appelé conditionnel, dans lequel il s'interdit d'exécuter des opérations irréversibles, ou d'envoyer au processeur PAC des instructions dont l'exécution serait irréversible: il se limite en fait à un travail de préparation.

A l'heure H+a, le processeur POP connait la valeur du prédicat qu'il vient d'évaluer. Sachant que le processeur PINS a fait un choix, il est en mesure de décider si ce choix était bon ou mauvais.

- si le choix était bon, il fait sortir le processeur PINS de son état conditionnel, et donc lui ôte ses scrupules à faire son travail de préparation, l'autorisant ainsi à continuer l'exécution en le faisant sortir d'une phase d'inaction éventuelle.

- si le choix était mauvais, le processeur POP prend conscience que son maître PINS a fait un travail de préparation inutile et que son collègue PAC a exécuté des instructions d'accès inutiles. Le processeur POP émet donc une sorte d'interruption qui doit avoir pour effet:

. d'annuler le travail de PAC en supprimant les opérandes rangés dans les files d'évaluation et de dépendance: cela est fait par simple modification des pointeurs sur ces files ;

. de vider les files d'attente BIPAC et BIPOP qui contiennent des instructions relatives à la mauvaise alternative ;

. de reprendre l'exécution au début de la bonne alternative: pour ce faire le processeur PINS, lorsqu'il fait un choix, calcule toujours l'adresse de cette alternative et la range au sommet de sa pile de contrôle.

Une telle organisation n'est performante que s'il est possible de connaître l'alternative la plus probable. C'est possible dans PASC-HLL, à cause de la diversification des instructions de branchement qui portent en elles-mêmes cette notion de probabilité: elles indiquent par exemple si un test est relatif à la sortie d'une boucle, pour laquelle la probabilité de sortir de la boucle est inférieure à celle d'y tourner.

Exemples de structures de contrôle

1/ Les boucles

Elles sont de plusieurs types qui peuvent être schématisés par:

a/ boucle WHILE

syntaxe PASCAL = WHILE<exp> DO <inst>

syntaxe PASC-HLL = LOOP(m) <exp> WHILE <inst> ENDLOOP

b/ boucle REPEAT

syntaxe PASCAL = REPEAT <inst> UNTIL <exp>

syntaxe PASC-HLL = LOOP(m) <inst><exp> UNTIL

c/ boucle EXITIF

syntaxe PASCAL = LOOP <inst1> EXITIF <exp><inst2> END

syntaxe PASC-HLL = LOOP(m) <inst1><exp> EXITIF <inst2> ENDLOOP

d/ boucle FOR

syntaxe PASCAL = FOR <V> := <exp1> TO <exp2> DO <inst>

syntaxe PASC-HLL = <exp1><exp2> FOR(m)

AFFECT(<v>) <inst> ROF

Les différentes syntaxes des boucles de PASC-HLL font apparaître une

mise en préfixé des instructions LOOP(m) et FOR(m) qui définissent le début de la boucle et donnent en paramètre l'adresse de la fin de la boucle (CO+m). Le processeur PINS choisit de rentrer dans la boucle. S'il se trompe il dispose de l'adresse de sortie de la boucle.

Les opérateurs de contrôle WHILE (sortie si FAUX), UNTIL et EXITIF (sortie si VRAI) sont placés en notation post-fixée derrière le prédicat.

Les politiques suivies par PINS sont les suivantes:

- il passe en séquence pour WHILE et EXITIF,
- il remonte au début de la boucle pour UNTIL et ROF (sortie si l'index dépasse la valeur MAX).

Par contre, l'instruction ENDLOOP utilisée dans les boucles WHILE et EXITIF est inconditionnelle: le processeur PINS remonte au début de la boucle.

Remarque 1:
La boucle FOR est décomposée en une boucle à indice croissant (instructions FORUP(n) et ROFUP) et une boucle à indice décroissant (instructions FORDOWN(m) et ROFDOWN).

Remarque 2:
Lorsque le processeur POP reçoit une instruction FORUP et FORDOWN et que la comparaison de **<**exp1**>** (valeur initiale) et de **<**exp2**>** (valeur finale) est bonne, il crée sur sa pile interne un descripteur de contrôle contenant la valeur courante de l'indice et sa valeur finale, descripteur qui est utilisé par les opérateurs ROFUP et ROFDOWN pour tester la condition de sortie de la boucle (voir plus loin le processeur POP).

2/ L'aiguillage à 2 directions
Cet aiguillage est issu des structures IF-THEN-ELSE selon les syntaxes:

a - syntaxe PASCAL = IF<exp> THEN <inst>
syntaxe PASC-HLL = <exp> IF(m) <inst> FI

b - syntaxe PASCAL = IF <exp> THEN <inst1> ELSE <inst2>

syntace PASC-HLL = <exp> IF(m) <inst1> NEHT(n)
ELSE <inst2> FI

Dans la structure b/ l'instruction NEHT(n) permet de sortir du segment de controle <inst1> et de sauter par dessus la deuxième alternative.

L'instruction IF(m) donne en paramètre l'adresse de la deuxième alternative, celle qui n'est pas choisie par le processeur PINS et exécutée uniquement si ce dernier s'est trompé.

L'instruction ELSE permet d'indiquer que l'on entre dans un segment de contrôle, utilisé pour l'adressage des étiquettes.

3/ L'aiguillage multi-directionnel

Il permet de faire un choix parmi un nombre quelconque d'alternatives, en fonction de la valeur scalaire d'une variable.

Syntaxe PASCAL = CASE <exp> OF
$<l_0>$: $<inst_0>$
⋮
$<l_p>$: $<inst_p>$
END

Syntaxe PASC-HLL = <exp> CASE(m)
$<l_0>$ OF (n_1) $<inst_0>$ FO
$<l_1>$ OF (n_2) $<inst_1>$ FO
⋮
$<l_p>$ OF(n_{p+1}) $<inst_p>$ FO
ESAC

Le paramètre de l'instruction CASE(m) définit l'adresse de l'instruction qui suit l'instruction ESAC: cette adresse est utilisée par l'instruction FO pour sortir du segment de contrôle.

La probabilité pour que <exp> soit égale à l'une des valeurs spécifiées dans la liste de valeurs $<l_i>$ dépend du nombre total de valeurs possibles et du nombre d'étiquettes de $<l_i>$. Cette probabilité dépend uniquement du PROGRAMMEUR et nous lui proposons donc le marché suivant: si vous souhaitez que PASC-HLL réalise rapidement votre aiguillage, tout en utilisant le minimum de place, placez tout simplement les cas les plus fréquents en tête de la liste des cas.

En effet, seul ce marché entre le programmeur et le concepteur est satisfaisant: les machines classiques considèrent que tous les cas sont équiprobables, et réalisent l'aiguillage à l'aide d'un tableau de branchements dans lequel un branchement indexé permet d'accéder. Une telle solution est rapide (2 instructions) mais son encombrement est catastrophique: si les valeurs possibles sont comprises entre 0 et n-1, il faut n instructions de branchement qui occupent 4*n octets! D'autre part, de nombreux "trous" peuvent exister dans ces 4*n octets, qui correspondent aux valeurs appartenant à aucune liste d'étiquettes $<l_i>$.

Nous avons donc cherché un compromis qui consiste à comparer séquentiellement la valeur de <exp> avec celle de $<l_0>$, puis $<l_1>$, ... jusqu'à trouver l'égalité, à moins que l'on ne tombe sur l'instruction ESAC, qui indique la fin de la structure de contrôle CASE.

Cette recherche séquentielle est optimisée par le fait que les 3 processeurs y participent: PAC range les valeurs des listes $<l_i>$ dans la file d'évaluation, POP effectue les comparaisons, PINS anticipe en supposant que le cas précédent n'est pas le bon et en préparant l'accès aux valeurs du cas suivant: le paramètre de l'instruction $OF(n_i)$ donne l'adresse de la liste d'étiquette $<l_i>$, permettant ainsi de sauter par dessus le segment $<inst_{i-1}>$. On trouvera dans la description du processeur PINS les algorithmes relatifs à l'exécution de chacune de ces instructions.

IV -CONCLUSION

La mise en oeuvre de cette architecture "pipeline" est présentée au fil des chapitres suivants.

Les détails relatifs aux mécanismes de synchronisation entre les différentes stations y sont donnés.

La complexité de présentation de ces mécanismes nous a obligé à un exposé simplifié dans ce chapitre: au niveau plus élémentaire de la réalisation, nous montrerons que les choses se simplifient ponctuellement.

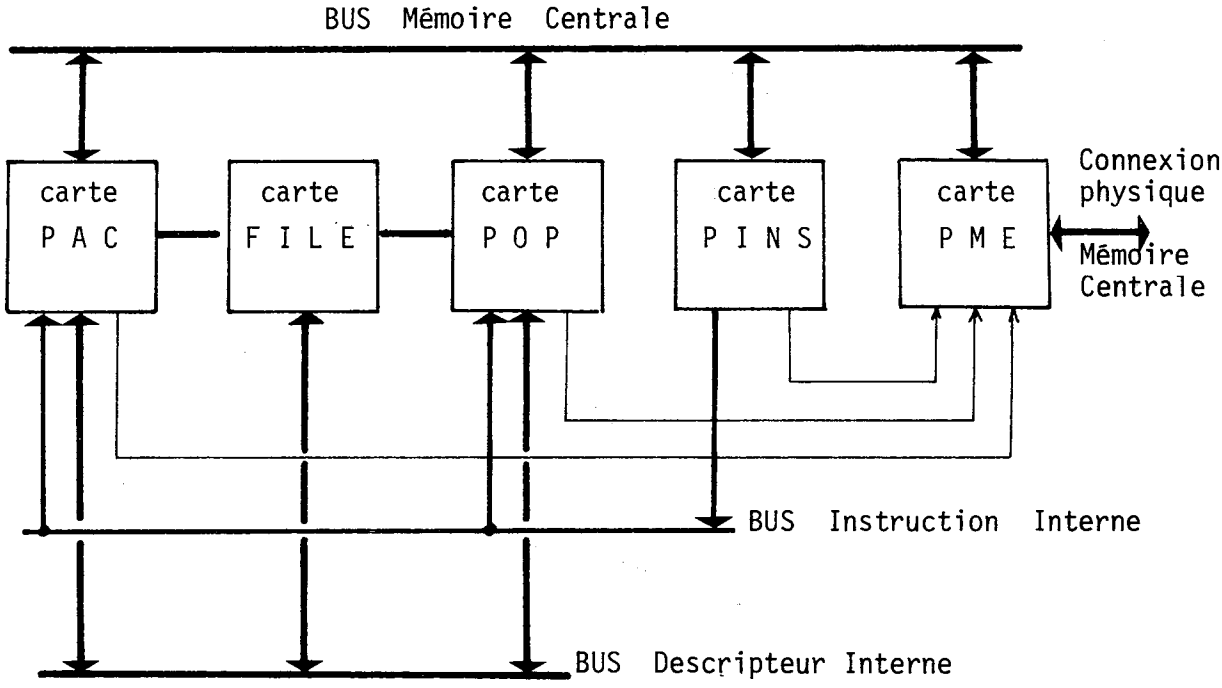
DEUXIEME PARTIE

Après la présentation des aspects généraux de PASC-HLL, nous décrirons dans cette deuxième partie les cinq processeurs qui constituent l'architecture pipeline.

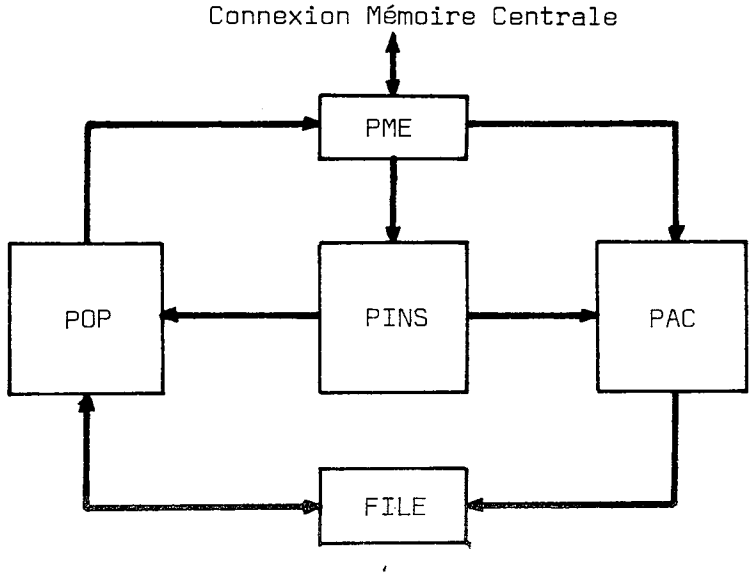
Pour chacun d'eux le chemin de données est présenté, qui est systématiquement organisé autour des macrocomposants SFC 92500, et les algorithmes à réaliser sont décrits sous la forme de microprogrammes symboliques.

La structure physique de PASC-HLL représentée par cinq "cartes" reliées par un "fond de panier", reflète exactement la structure théorique. Elle est illustrée dans la figure suivante.

ARCHITECTURE PHYSIQUE



ARCHITECTURE THEORIQUE



CHAPITRE 3

LE PROCESSEUR DE TRAITEMENT DES INSTRUCTIONS PINS

- Accès aux instructions en mémoire
- Analyse de la chaîne post-fixée
- Exécution des instructions de contrôle

DESCRIPTION DU PROCESSEUR P I N S

INTRODUCTION

Le processeur PINS joue un rôle central dans la machine: c'est lui qui accède aux instructions du programme et les distribue aux deux processeurs exécutants PAC et POP, générant au passage les "extra ordres" nécessaires à la gestion de la file d'évaluation.

La description de ce processeur est décomposée en trois parties qui correspondent aux trois fonctions principales qu'il réalise:

1/ l'accès aux instructions en mémoire,

2/ l'analyse de la chaîne post-fixée d'entrée,

3/ l'exécution des instructions de contrôle.

L'algorithme de ce processeur n'est pas classique en ce sens que plusieurs étapes d'interprétation sont nécessaires. Ce sont:

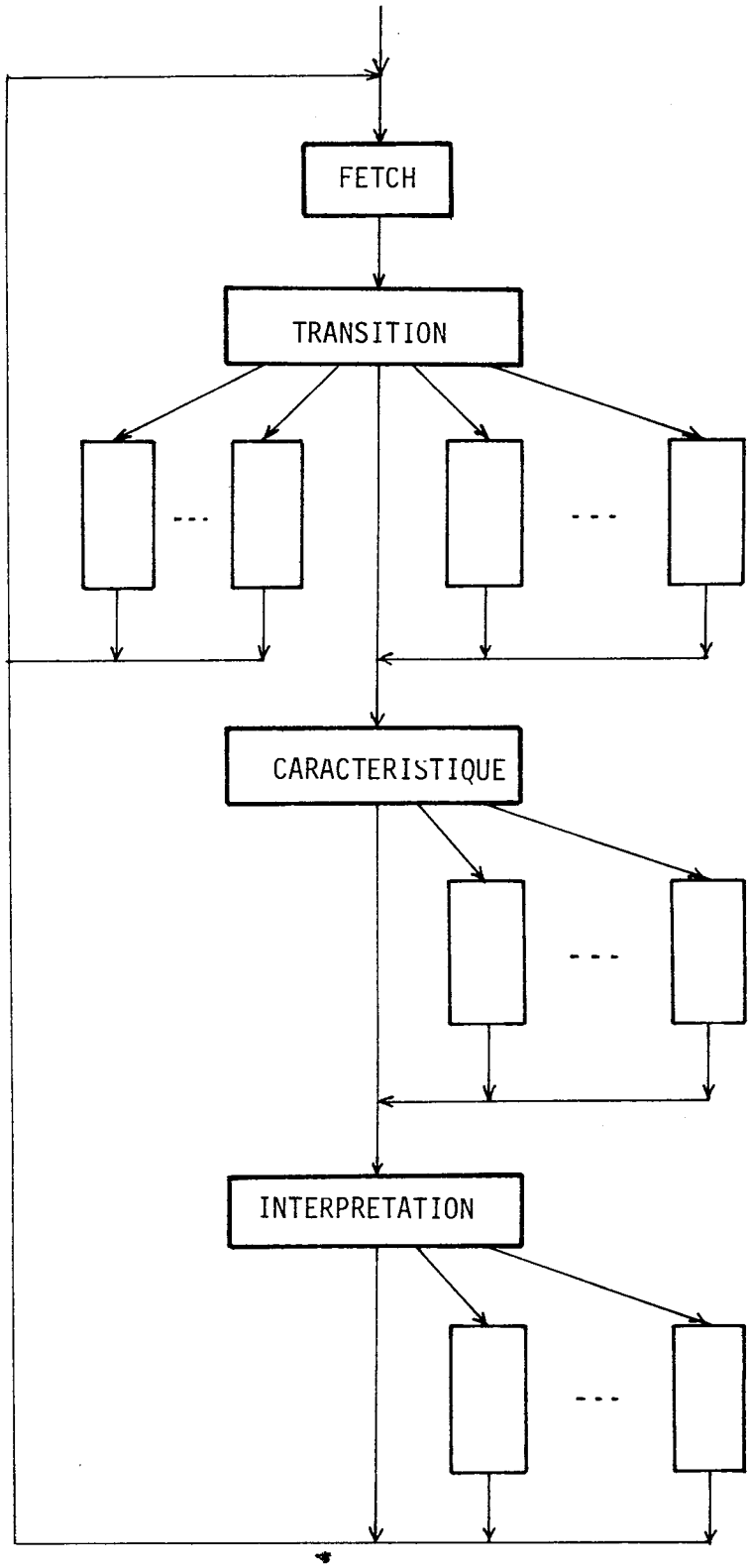
- le traitement des Transitions entre instructions d'accès, opérateurs et instructions de contrôle qui commandent la gestion de l'image théorique de la file d'évaluation et la génération des "extra-ordres".

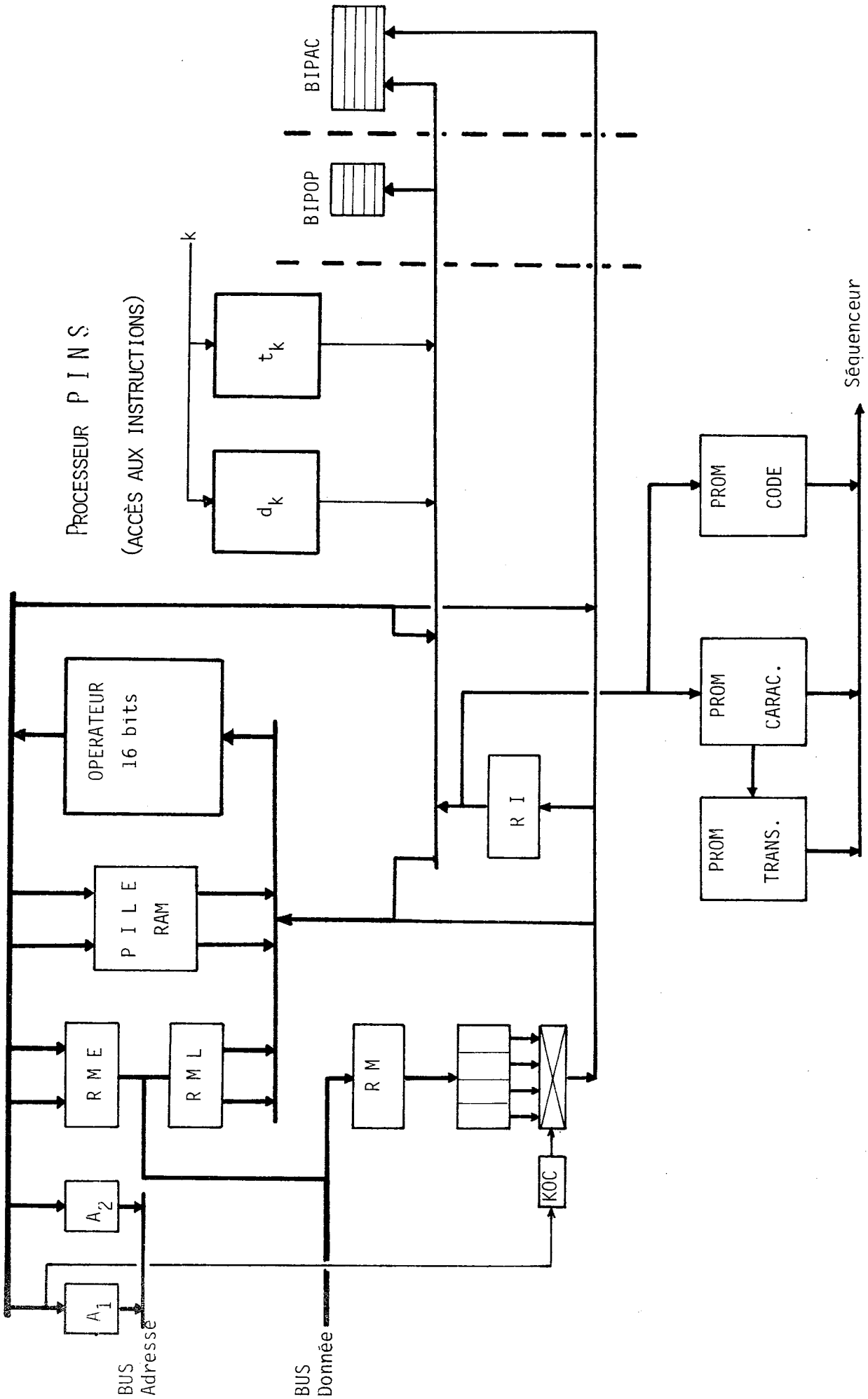
- le traitement des Caractéristiques associées à chaque instruction reçue.

- et finalement l'exécution proprement dite de l'algorithme d'interprétation associé à l'instruction (de contrôle).

La structure générale de l'algorithme à réaliser est schématisé par la figure suivante:

ALGORITHME DU PROCESSEUR P I N S




PROCESSEUR P I N S
(ACCÈS AUX INSTRUCTIONS)
BIPAC
BIPOP
k
t_k
d_k
OPERATEUR 16 bits
P I L E RAM
R I
R M E
R M L
R M
KOC
A_2
A_1
BUS Adresse
BUS Donnée
PROM CODE
PROM CARAC.
PROM TRANS.
Séquenceur

I - Le codage des instructions

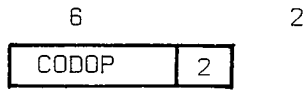
La structure pipeline de PASC-HLL présente l'originalité suivante: la même instruction est successivement décodée par plusieurs processeurs et elle implique des interprétations différentes pour chacun de ces processeurs.

Par exemple, un opérateur d'addition est interprété par le processeur PINS comme un opérateur binaire qui modifie l'image théorique de la file d'évaluation et doit être envoyé au processeur POP ; pour le processeur FILE c'est un opérateur binaire qui implique l'envoi de FILEVAL(P_1) au processeur POP ; pour ce dernier c'est un opérateur binaire qui implique la lecture de l'opérande fourni par le processeur FILE et finalement l'exécution de l'addition.

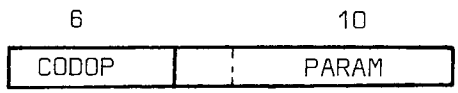
Cette complexité pose des problèmes de codage qui seront complètement présentés dans la deuxième thèse [8] .

Nous nous contentons de présenter ici les différents formats des instructions qui ont été définis par la méthode descendante [ANNEXE].

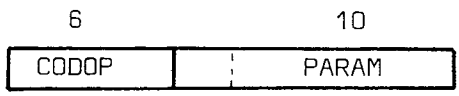
Instruction de contrôle sans paramètre



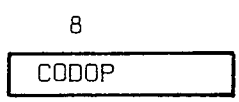
Instruction de contrôle avec paramètre



Instruction d'accès

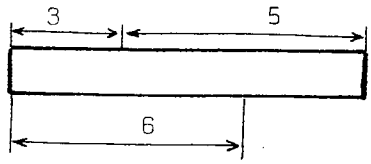


Opérateur



Seuls les 6 premiers bits d'une instruction sont décodés par le processeur PINS, qui peut donc reconnaître 64 classes d'instructions.

Comme nous avons besoin de coder 32 opérateurs binaires et 32 opérateurs unaires, nous avons choisi le découpage suivant pour les coder:



- les 5 bits de droite caractérisent un opérateur parmi 32,
- les 6 bits de gauche sont décodés par le processeur PINS.

Il y a donc un recouvrement de 3 bits, ce qui signifie que les opérateurs binaires occupent 8 classes d'instructions différentes vis-à-vis du processeur PINS.

Il reste donc 64-2x8=48 classes d'instructions pour coder les instructions autres que les opérateurs, ce qui s'avère suffisant.

A titre d'introduction, nous présentons ci-après une méthode de gestion d'une Mémoire Locale en PILE, qui sera utilisée pour les deux processeurs PINS et POP, et que nous ne décrirons qu'une seule fois (ci-après) pour utiliser ensuite les primitives PUSH, POP et ACCES dans l'exposé des microprogrammes.

II- GESTION D'UNE MEMOIRE LOCALE EN PILE

Les processeurs PINS et POP utilisent tous deux une PILE, l'un pour gérer des descripteurs de contrôle du programme, l'autre pour stocker les résultats intermédiaires "longs" et les descripteurs de contrôle des boucles FOR.

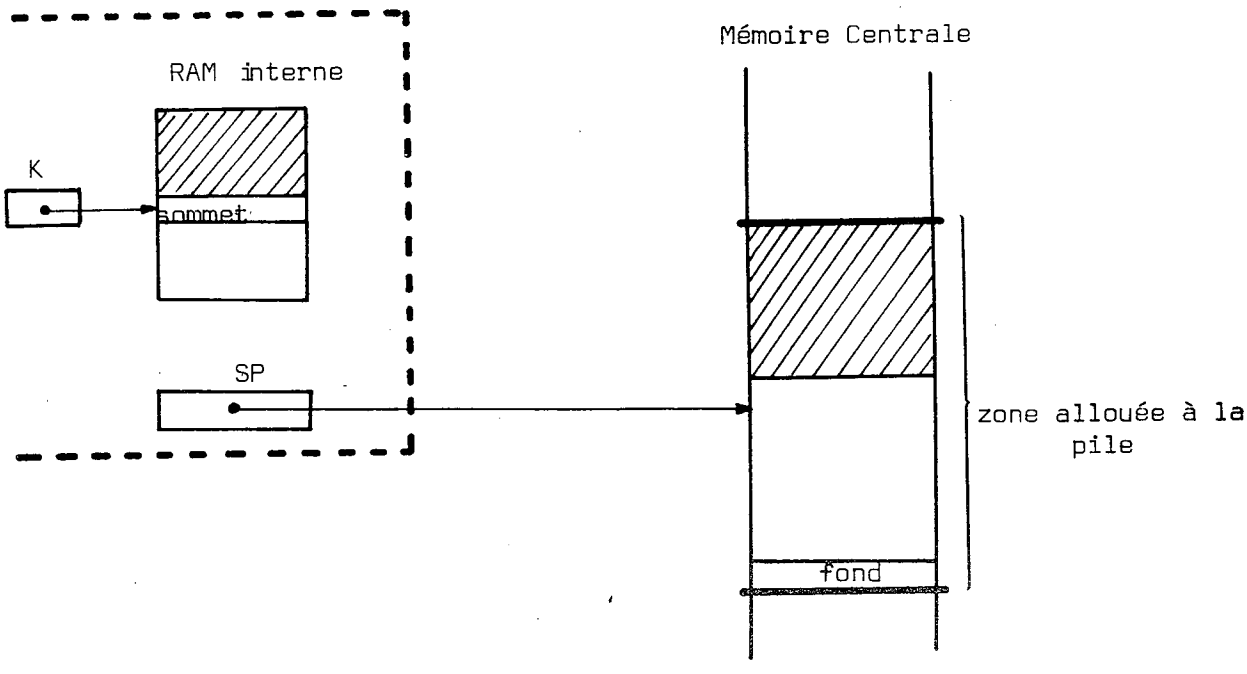
Nous devons trouver un compromis pour l'implantation de ces piles et nous sommes en cela guidés par les contraintes suivantes:

- réduction du nombre d'accés à la mémoire centrale,
- limitation de la taille d'une mémoire locale,
- réduction du temps nécessaire à la sauvegarde du contexte, sachant que la machine travaille en multiprogrammation.

Le compromis retenu consiste à doter le processeur d'une petite mémoire locale de 16 mots qui "se prolonge" en mémoire centrale, dans une zone fixe pouvant aller jusqu'à 256 mots. Cette solution répond aux 3 critères énoncés, mais elle suppose la réalisation d'un mécanisme de gestion interne de cette pile.

I.- DEFINITION FORMELLE DE LA PILE

Le processeur dispose d'une mémoire locale de 16 mots (RAM), contenant le sommet de la pile. L'élément sommital (le dernier entré) est pointé par un compteur K (4 bits = 1 circuit intégré). Un registre de 16 bits, appelé pointeur de pile et noté SP, donne l'adresse de l'élément sommital en mémoire centrale.



Lorsque la pile interne est pleine, l'élément le plus ancien est recopié en mémoire centrale (opération PUSH).

Inversement, lorsque la pile interne est vide, l'élément le plus récemment empilé est lu en mémoire centrale et recopié dans la pile interne.

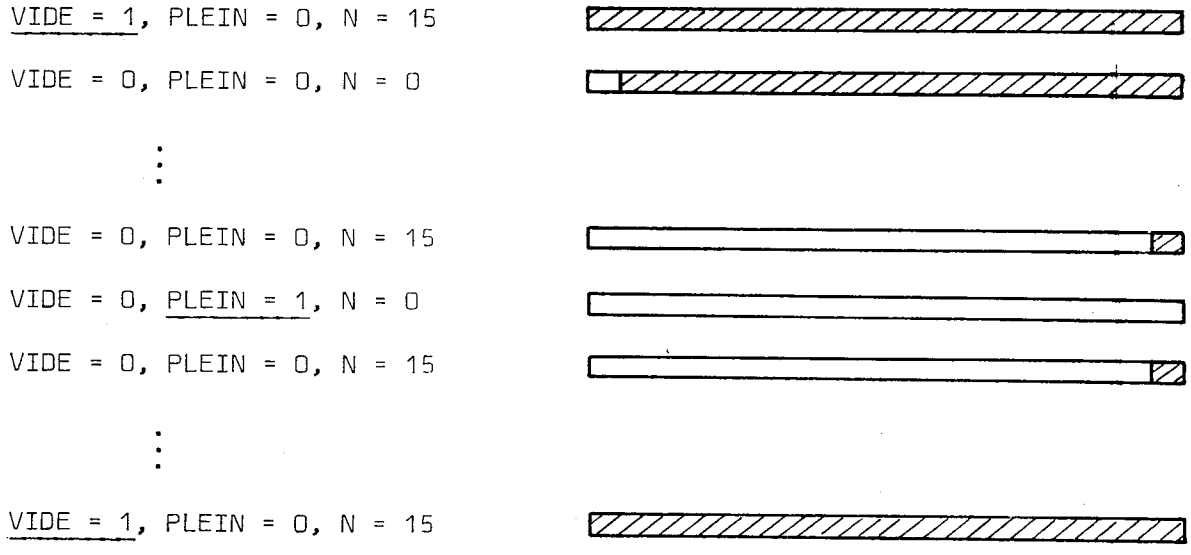
II.- REALISATION DU CONTROLE DE LA PILE INTERNE

Le problème du contrôle de la pile interne consiste à connaître son remplissage, caractérisé par le fait qu'elle est "vide" ou "pleine".

On définit un second compteur N, dont la valeur donne, modulo 16 et à une unité près, le nombre de mots occupés dans la pile interne. Deux indicateurs (bascules) définissent respectivement l'état "vide" et l'état "plein".

Les deux compteurs K et N, ont des valeurs indépendantes, en particulier, la valeur initiale de K est quelconque.

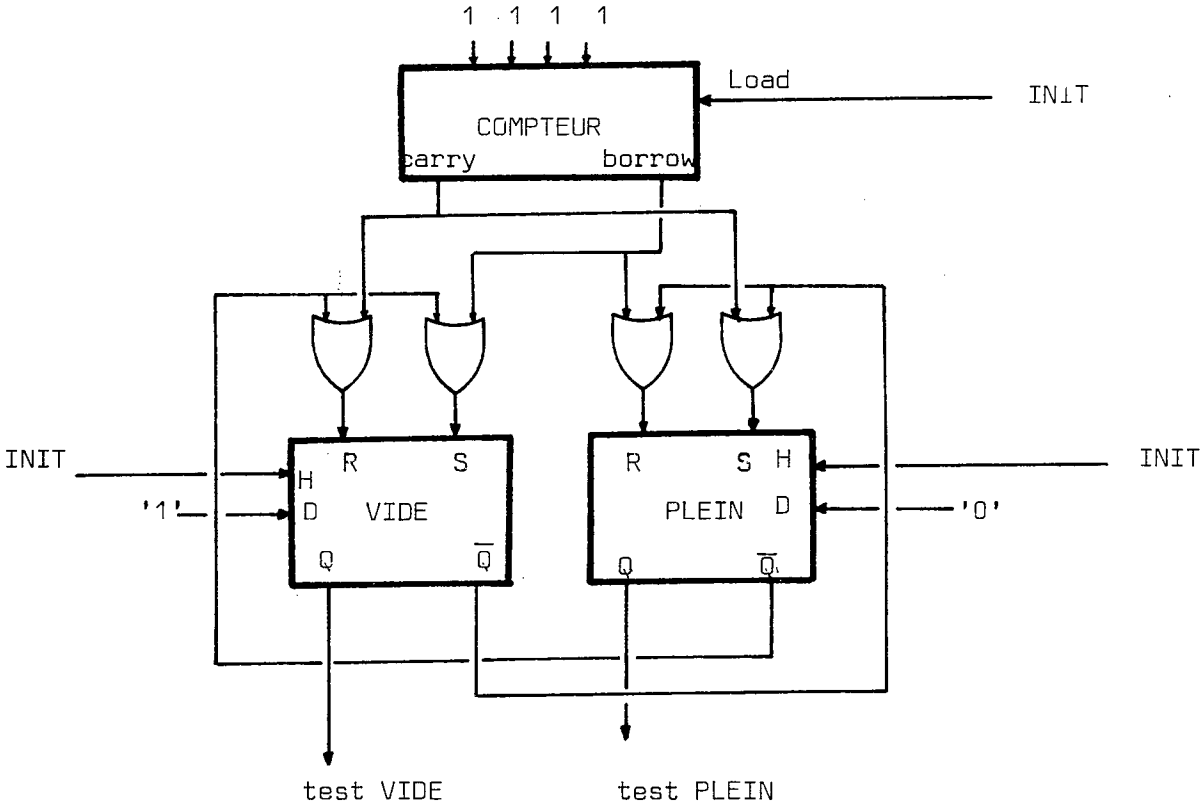
ETAT D'OCCUPATION DE LA PILE INTERNE:



On remarque que:

VIDE ← 0 si N passe de 15 à 0 et PLEIN = 0
VIDE ← 1 si N passe de 0 à 15 et PLEIN = 0
PLEIN ← 0 si N passe de 0 à 15 et VIDE = 0
PLEIN ← 1 si N passe de 15 à 0 et VIDE = 0

D'où la réalisation qui en découle directement:



Le compteur utilisé est un 74192, la réalisation nécessite 3 circuits

III.- MICROPROGRAMMATION DE GESTION DE LA PILE

Chacun des processeurs (PINS et POP) qui utilisent une pile exécute les micro-programmes suivants, qui seront structurés comme des sous-microprogrammes, appelés par un JSR et "retournés" par un RTS.

III.1. opération PUSH

Lorsqu'un nouvel élément doit être empilé, il faut s'assurer que la pile interne n'est pas pleine, sinon il faut libérer une case.

On peut donc supposer que le test de l'indicateur PLEIN se fait au moment de l'appel du sous-programme: ainsi l'instruction de branchement JSR (PUSH[PLEIN]) conduira soit en PUSH (si PLEIN = 0), soit en PUSH + 1 (si PLEIN = 1).

Microprogramme:

```
PUSH(PLEIN = 0): K+1 → K  et N+1 → N, RTS
PUSH+1 (PLEIN = 1): Y = SP+1 → SP, Y → RADM, K+1 → K
                    PILE(K) → RME, Ordre(EMC), RTS
```

III.2. opération POP

Si la pile interne est vide, il faut décrémenter la valeur du registre SP, sinon on décrémente celle de compteur K.

Microprogramme:

```
POP(VIDE = 0):      K-1 → K  et  N-1 → N, RTS
POP+1(VIDE = 1):    SP-1 → SP, RTS
```

III.3. opération d'accés au sommet de pile

Dans le cas où la pile interne est vide et que l'on désire accéder au sommet de la pile, il faut aller le chercher en mémoire centrale: on appelle donc le sous-programme ACCES par la microinstruction:

si VIDE alors JSR(ACCES)

Microprogramme:

```
ACCES: Y = SP, SP-1 → SP, Y → RADM, N+1 → N, Ordre(LMC)
       Attente(MC), RML → PILE(K), RTS
```

III.4. sauvegarde de la pile interne

En cas de sauvegarde du contexte des processeurs (multiprogrammation), le contenu de la pile interne est sauvegardé en mémoire centrale, ainsi que la valeur du registre SP:

Microprogramme:

```
SAUVEGARDE: si VIDE alors JMP(SAUVESP)
            attente(MC), Y = SP+1 → SP, Y → RADM
            Pile(K) → RMA, K-1 → K et N-1 → N, ordre(EMC),
                                          JMP(SAUVEGARDE)
SAUVESP: attente(MC), Y = 0, Y → RADM
         SP → RME, ordre(EMC)
```

B - LE MODULE D'ACCES AUX INSTRUCTIONS

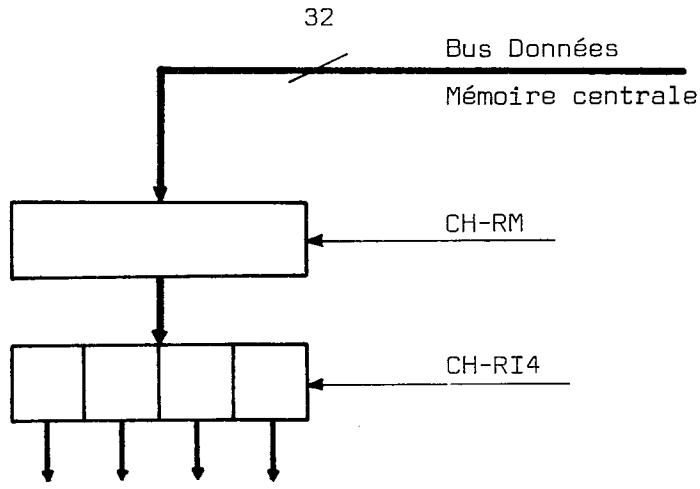
I - Principe du module d'accès

Le processeur PINS est chargé d'accéder aux instructions qui composent le programme contenu en Mémoire Centrale. L'accès à une instruction dépend de la nature de l'instruction précédente, qui définit ou non une rupture de séquence. D'autre part, une instruction étant constituée d'un nombre variable d'octets, elle peut être "à cheval sur" plusieurs mots.

Afin d'éviter une attente trop longue entre l'obtention de deux mots consécutifs, il est nécessaire de prévoir un système de "pipe-line" à deux niveaux constitué par:

1/ un registre de 32 bits pouvant recevoir un mot lu en mémoire centrale, appelé RM ;

2/ un registre de 4 fois 8 bits, appelé RI4, pouvant recevoir le contenu de RM et dont les 4 octets sont accédés séquentiellement.

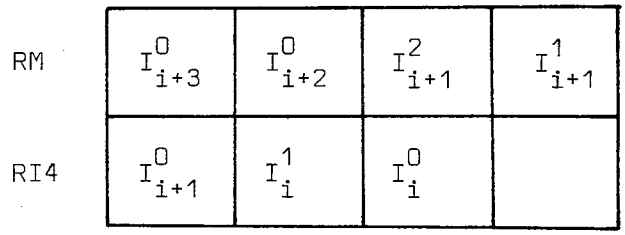


Cette structure constitue une "file d'attente" de longueur un.
L'information contenue dans la file d'attente est transférée dans RI4

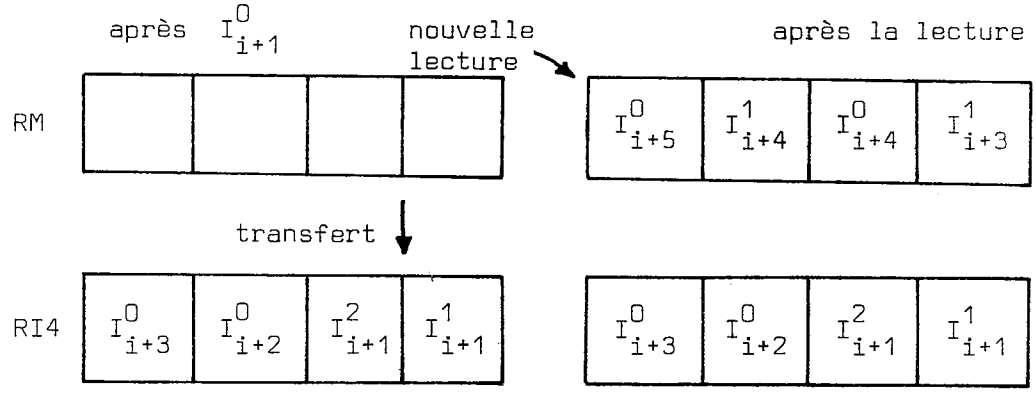
(registre de sortie) quand RI4 est vide et RM plein. Une nouvelle demande de lecture peut être faite lorsque RM est vide.

L'exécution d'instructions séquentielles est ainsi optimisée car on peut disposer des octets constituant l'instruction en cours dans RI4, et des octets constituant l'(les)instruction(s) suivante(s) dans RM.

Exemple:



Dans l'exemple ci-dessus, en supposant que l'octet d'instruction couramment exécuté est I^0_i, on voit qu'on peut exécuter séquentiellement les instructions I_i, I_{i+1}, I_{i+2} sans faire de lecture-mémoire, et qu'une lecture pourra être demandée dès que I^0_{i+1} aura été décodé.



II - Contrôle du Module d'Accès

Afin de simplifier le contrôle du module d'accés par le microprogramme du processeur PINS, nous avons réduit l'interface à deux ordres et une condition d'attente.

Les deux ordres sont les suivants:

1/ l'odre appelé INIT correspond à une rupture de séquence qui conduit à une initialisation du module d'accés. La file d'attente constituée par RM et RI4 est alors considérée comme vide, même si RM contient les octets constituant l'instruction qui suit celle qui vient d'être exécutée. Une adresse de branchement doit être fournie au Module d'accés en même temps que cet ordre INIT.

2/ l'ordre appelé SEQ correspond à la demande de l'octet suivant, soit au cours de l'exécution d'une instruction composée de plusieurs octets, soit pour l'accés à l'instruction suivante.

La condition d'attente porte sur l'indication de présence de l'octet d'instruction suivant. Une bascule est utilisée qui est testée par le microprogramme et positionnée par le module d'accés lors de la réception d'un ordre INIT ou bien lorsque les octets contenus dans RI4 ont tous été accédés.

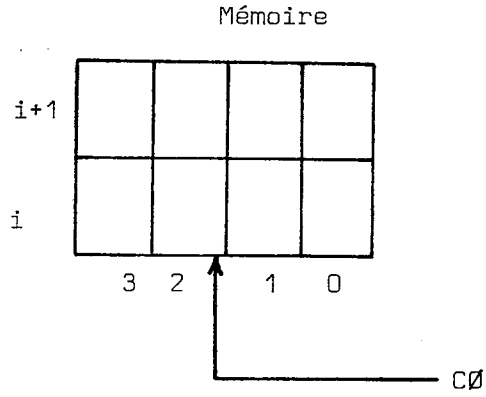
III - Gestion de l'adresse de l'instruction suivante

L'adresse de l'instruction suivante est une adresse d'octet puisque les instructions sont de longueur variable. Elle est mémorisée dans une variable de 16 bits appelée Compteur Ordinal (CO) contenue dans une Opérateur 16 bits constitué par 4 tranches SFC 92901.

Lors d'une rupture de séquence (initialisation), cette adresse est envoyée au module d'accés, en même temps que l'ordre INIT. Le module d'accés extrait de cette adresse les 2 bits de poids faible qui indiquent le numéro d'un octet dans un mot de 4 octets et les 14 bits de poids fort qui sélectionnent un mot dans la mémoire centrale.

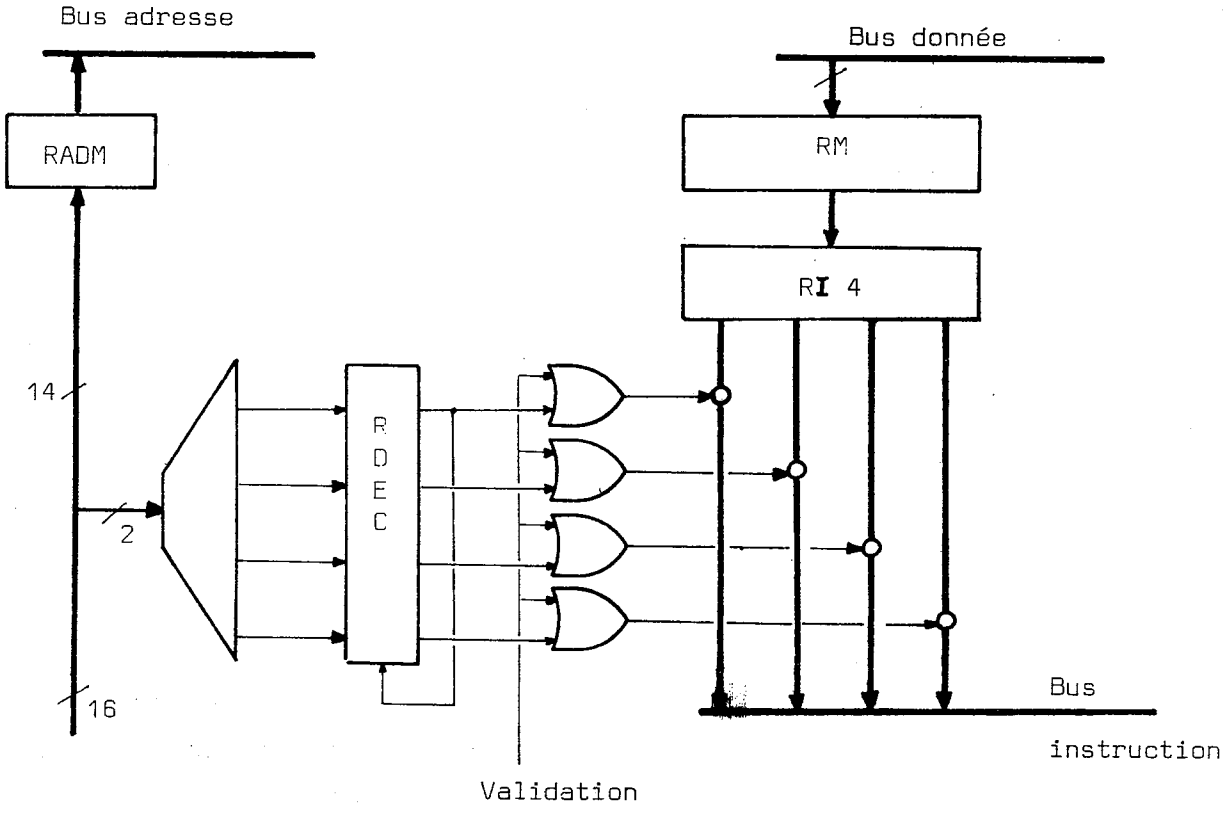
Sachant que nous avons choisi un système "pipe-line", il y aura un décalage entre la valeur contenue dans le compteur ordinal et celle de l'adresse envoyée vers la mémoire.

Exemple:



- CØ contient
  - . i comme poids fort
  - . 2 comme poids faible
- l'adresse mémoire vaut (i+1)

Nous adoptons donc le chemin de données suivant:



Le registre adresse mémoire RADM est un compteur initialisé par l'ordre INIT et incrémenté pour la lecture du mot suivant. Son contenu diffère généralement de celui du Compteur Ordinal (il lui est supérieur).

Les deux bits poids faible de l'adresse de branchement sont décodés. La sortie du décodeur est utilisée pour charger un registre à décalage KOC dont les sorties parallèles sélectionnent un des octets du registre RI4.

Exemple: poids faible de l'adresse = 2

	3	2	1	0
- valeur initiale de KOC	1	0	1	1
- après un 1er ordre SEQ	0	1	1	1
- après un 2ème ordre SEQ	1	1	1	0

Le registre à décalage KOC doit donc être structuré pour un décalage circulaire et le passage de 0 à 1 du bit de poids faible indique que tous les octets de RI4 ont été accédés et qu'il faut donc recharger RI4 avec le mot suivant.

Les différentes commandes de ce chemin de données sont générées par un module de contrôle qui reçoit:

- les ordres INIT et SEQ venant du microprogramme,
- l'ordre FLM venant de la mémoire et qui indique une Fin de Lecture d'un Mot.

Le module émet:

- un ordre DBM (Demande Bus Mémoire) vers la mémoire,
- un signal NEXTOC vers le microprogramme dont la valeur 0 indique qu'il lui faut attendre que l'octet suivant soit disponible.

Remarque: Le caractère asynchrone de ce Module de Contrôle implique l'utilisation de deux monostables pouvant être déclenchés par les évènements désynchronisés qui peuvent se produire (ordres du microprogramme et fin de lecture mémoire).

C - ANALYSE DE LA CHAINE POST FIXEE

L'algorithme d'analyse de la chaîne post fixée pour la génération des "extra-ordres" a été exposé dans diverses publications ([ ],[ ], ...). Nous nous contenterons d'en décrire ici la réalisation.

I. IMPLANTATION DES SUITES $\{d_k\}$ et $\{t_k\}$.

Les deux suites $\{d_k\}$ et$\{t_k\}$ évoluent comme une PILE dont on accède les deux éléments sommitaux.

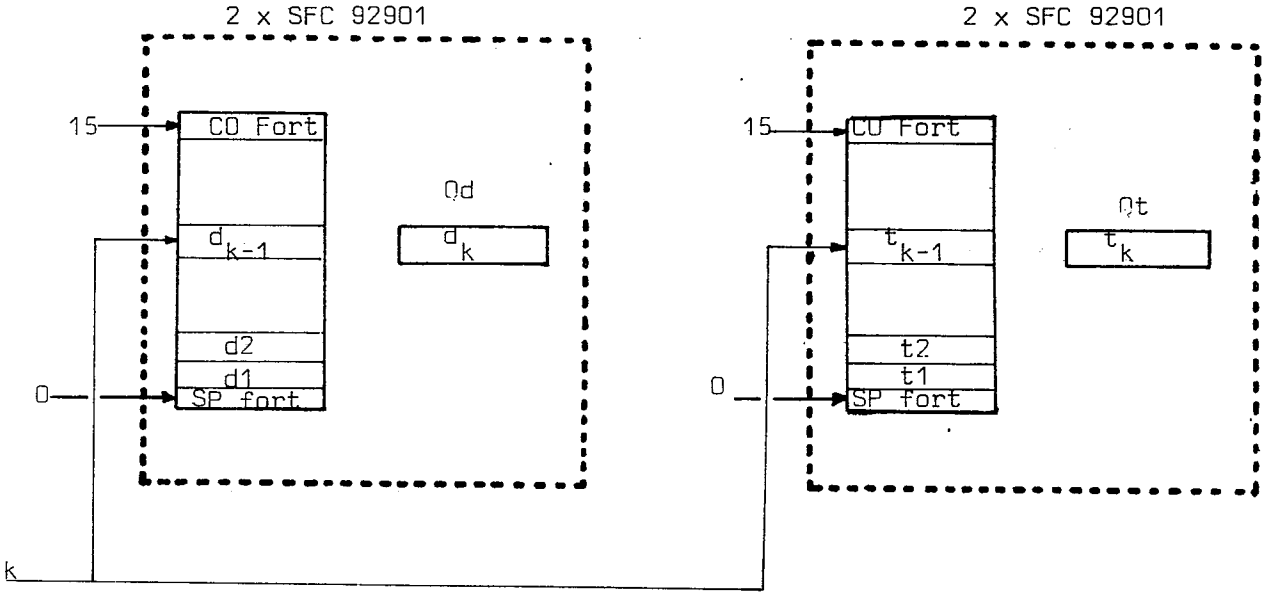
D'autre part, il est fréquent de modifier à la fois d_k et t_k. La solution qui nous a semblé être la plus intéressante consiste à utiliser deux tranches SFC 92901, l'une pour gérer la suite $\{d_k\}$, l'autre pour gérer la suite $\{t_k\}$.

Le sommet de la pile, constitué par l'élément d_k ou t_k, est implanté dans l'accumulateur qui sera noté Q_d ou Q_t.

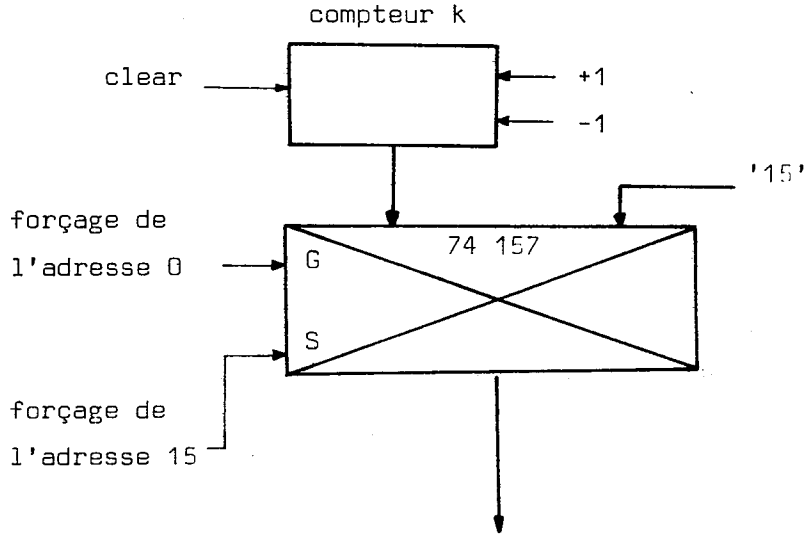
Les éléments d_1, d_2, ..., d_{k-1} et t_1, t_2, ..., t_{k-1}, sont implantés dans la mémoire RAM interne à la tranche SFC 92901 et adressés par un compteur externe appelé k.

Finalement, nous avons choisi d'implanter le Compteur Ordinal CØ "à cheval" sur les tranches qui contiennent les suites $\{d_k\}$ et $\{t_k\}$, s'où le schéma général d'implantation des variables.

(De même le sommet de la pile de contrôle SP est "à cheval" sur les quatre tranches).



L'adressage des éléments des mémoires RAM internes est réalisé simplement de la manière suivante:



entrées A et B des SFC 92901
adresse de la RAM

II - REALISATION DES ALGORITHMES DE GESTION DE LA FILE

Les algorithmes de gestion de la file d'évaluation sont basés sur la détection des transitions entre les divers types d'instructions qui sont:

- les instructions d'accés,
- les OPERATEURS,
- les instructions qui définissent une FIN D'EXPRESSION,
- et les instructions de CONTROLE.

Toutes les transitions ne sont pas autorisées et certaines doivent être détectées comme étant des erreurs. Les transitions autorisées conduisent à des interprétations particulières qui correspondent à l'exécution d'une séquence de microinstructions.

La complexité du problème, due au grand nombre de cas particuliers, peut être diminuée par l'utilisation de petites mémoires PROM servant au décodage.

2.1. Le codage des caractéristiques d'une instruction

Chaque instruction présente, vis-à-vis du processeur PINS, un certain nombre de caractéristiques qui sont:

1/ l'appartenance à l'un des types suivants: ACCES, OPERATEUR , FIN D'EXPRESSION, CONTROLE.

2/ le fait d'avoir une longueur simple ou une longueur double selon le tableau suivant

	simple	double
instruction d'accés	2 octets	4 octets
opérateur	1 octet	2 octets

3/ le fait de devoir être envoyée vers le processeur POP ou vers le processeur PAC, ou vers les deux.

4/ le fait de ne pas être exécutée si le processeur PINS se trouve dans l'état conditionnel parce qu'il a anticipé sur le choix à faire,

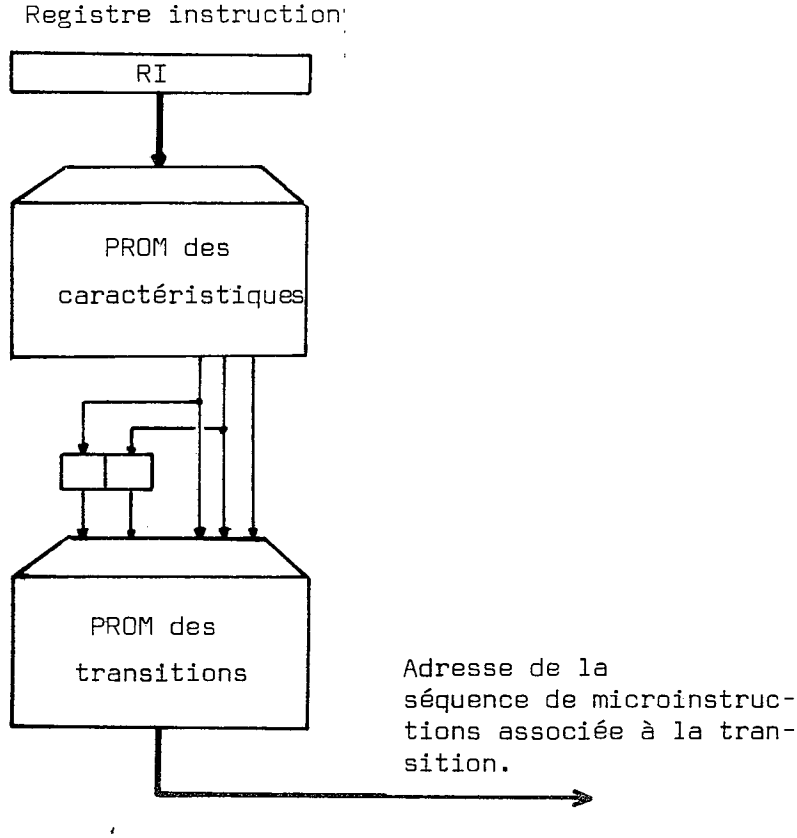
5/ le fait de mettre le processeur PINS dans l'état conditionnel.

Toutes les caractéristiques précédentes son regroupées dans une mémoire PROM de décodage, dont l'adresse est directement issue du Code Opération de l'instruction.

2.2. Réalisation des transitions

Un registre d'état, appelé ETAT, et constitué de deux bascules D, mémorise le type de l'instruction en cours d'exécution. L'adresse de la séquence de microinstructions propre à l'exécution d'une transition dépend du type de l'instruction précédente, mémorisé dans ETAT, et de celui de l'instruction courante.

Nous utilisons une deuxième mémoire PROM dont l'adressage est illustré par la figure suivante:



Si les types des instructions sont codés

00 pour Accés

01 pour Opérateur

10 pour Fin d'expression

11 pour Contrôle

alors le contenu de la PROM des transitions est

adresse d'entrée		adresse de branchement
00	00	ACCES → ACCES
00	01	ACCES → OPERATEUR
00	10	ACCES → FIN D'EXP
00	11	ACCES → CONTROLE ⇒ ERREUR
01	00	OP → ACC
01	01	OP → OP
01	10	OP → FE
01	11	OP → CT ⇒ ERREUR
10	00	FE → ACC ⇒ début d'exp
10	01	FE → OP ⇒ ERREUR
10	10	FE → FE ⇒ ERREUR
10	11	FE → CT
11	00	CT → ACC ⇒ début d'exp
11	01	CT → OP ⇒ ERREUR
11	10	CT → FE ⇒ ERREUR
11	11	CT → CT

état courant

état précédent

Un troisième bit est utilisé comme caractéristique de certaines instructions spéciales qui peuvent conduire à des transitions particulières. C'est le cas de CALLF, appel de fonction, qui nécessite un traitement spécial pour les transitions:

ACCES → CALLF, OPERATEUR → CALLF,

FE → CALLF et CONTROLE → CALLF.

De même, la transition FE → FE est possible uniquement entre une instruction FORUP ou FORDOWN et l'instruction AFFECT.

Et enfin la transition ACCES → OF est détectée pour générer l'extra-ordre OF(n) vers BIPOP.

Remargue: Le registre ETAT, réalisé par deux bascules D, peut être chargé (entrée H), mis à 00 (entrée Clear) ou mis à 11 (entrée Preset) pour initialiser le processeur soit dans l'état Contrôle (11) soit dans l'état Accés (00) après un retour de fonction (instruction RETURNF).

2.3. ALGORITHMES DES TRANSITIONS (MICROPROGRAMMES)

2.3.1. Transitions ACCES → ACCES

Une instruction d'accés est composée de 2 ou 4 octets et doit être envoyée vers le processeur d'accés. L'élément d_k, contenu dans Q_d, doit être incrémenté.

Microprogramme:

ACC:

1 : Attente(NEXTOC), CØ+1 → CØ, Ordre(SEQ)

2 : Attente(BIPAC-NON-PLEIN), Q_d+1 → Q_d
(RI,M) → BIPAC, JMP([ROM])

Le contenu de la ROM de décodage est égale

- à l'adresse de la séquence de FETCH pour une longueur simple,
- à l'adresse de la microinstruction notée 3 pour une longueur double.

3 : Attente(NEXTOC), CØ+1 → CØ, Charge RI, Ordre (SEQ)

4 : Attente (NEXTOC), CØ+1 → CØ, ordre (SEQ)

5 : Attente (BIPAC-NON-PLEIN), (RI,M) → BIPAC, JMP(FETCH)

2.3.2. Transition ACCES → OPERATEUR

Un ordre AVANCE ou INIT doit être envoyé au processeur POP selon que l'expression est commencée ou non, comme indiqué par la bascule DEBEXP.

Microprogramme:

1 : Attente (BIPOP-NON-PLEIN), $Y_d = Q_d$,
(AVANCE/INIT, Y_d) → BIPOP, DEBEXP ← 0

OPE:

2 : Attente (BIPOP-NON-PLEIN), RI → BIPOP, JMP([ROM])

L'adresse fournie par la ROM de décodage est:

FETCH pour un Opérateur UNAIRE de longueur 1,
UN2 " " " 2,
BIN " BINAIRE " 1.

UN2:

3 : Attente (NEXTOC), CØ+1 → CØ, charge RI, ordre (SEQ)

4 : Attente (BIPOP-NON-PLEIN), RI → BIPOP, JMP(FETCH)

BIN:

5 : Q_d 1 → Q_d, Q_t+1 → Q_t, Valide (d_k=0), JMP(FETCH)

La microinstruction n°5 décrémente d_k et incrémente t_k et on mémorise le passage à 0 de d_k pour un test ultérieur.

2.3.3. Transition OPERATEUR → OPERATEUR

La file d'évaluation est mise à jour si d_k=0 et un ordre RECULE est généré dans ce cas.

Microprogramme:

1 : si ($d_k \neq 0$) alors JMP(OPE)

2 : attente(BIPOP-NON-PLEIN), $Y_t = t_{k-1}$, (RECULE, Y_t) → BIPOP

3 : $Q_t + t_{k-1} \rightarrow Q_t$ et $d_{k-1} \rightarrow Q_d$, k-1 → k, JMP(OPE)

L'étiquette OPE donne l'adresse de la 2ème microinstruction de la transition ACCES → OPERATEUR.

2.3.4. Transition OPERATEUR → ACCES

La file d'évaluation est mise à jour si d_k=0, puis le traitement se poursuit comme pour la transition ACCES → ACCES (étiquette ACC).

Microprogramme:

1 : si ($d_k \neq 0$) alors k+1 → k, JMP(ETIQ)

2 : Q_t + t_{k-1} → t_{k-1}

RAZQD:

3 : 0 → Q_t et 0 → Q_d, JMP(ACC)

ETIQ:

4 : Q_t → t_{k-1} et Q_d → d_{k-1}, JMP(RAZQD)

2.3.5. Transition ACCES → FIN D'EXPRESSION

Un ordre AVANCE ou INIT est généré, puis on se branche au traitement de l'opérateur de contrôle.

Microprogramme:

1 : Attente(BIPOP-NON-PLEIN), Y_d=Q_d,
(INIT/AVANCE, Y_d) → BIPOP, JMP(CONTROLE)

2.3.6. Transition OPERATEUR→ FIN D'EXPRESSION

Un ordre RECULE est généré si d_k=0, puis on se branche au traitement de l'opérateur de contrôle.

Microprogramme:

1 : si $d_k \neq 0$ alors JMP(CONTROLE)
2 : Attente (BIPOP-NON-PLEIN), Y_t=t_{k-1},
(RECULE, Y_t) → BIPOP, JMP(CONTROLE)

2.3.7. Transition FE ou CONTROLE→ ACCES

C'est l'initialisation de la file d'évaluation.

Microprogramme:

1 : $0 \rightarrow Q_d$ et $0 \rightarrow Q_t$, k ← 0 et DEBEXP ← 1, JMP(ACC)

2.4. Traitement propre à l'appel de fonction

2.4.1. Transition ACCES → CALLF

Une succession d'ordres SAUVE(d_k), RECULE(t_{k-1}), SAUVE(d_{k-1}), ..., SAUVE(d_1) est envoyée vers BIPOP.

Microprogramme:

1 : attente(BIPOP-NON-PLEIN), $Y_d=Q_d$, (INIT/AVANCE,Y_d) → BIPOP,

SUITE:

2 : JSR(ATTENTE(ERREUR, $\overline{ETC}$))

3 : attente(NEXTOC), CØ+1 → CØ, Ordre(SEQ)

4 : attente(BIP AC-NON-PLEIN), (RI,M) → BIPAC

5 : attente(BIPOP-NON-PLEIN), $Y_d=Q_d$, (SAUVE, Y_d) → BIPOP

TEST:

6 : si k=0 alors JMP(EMPILE)

7 : attente(BIPOP-NON-PLEIN), $Y_t=t_{k-1}$, (RECULE, Y_t) → BIPOP

8 : attente(BIPOP-NON-PLEIN), $Y_d=d_{k-1}$, (SAUVE, Y_d) → BIPOP

9 $d_{k-1} + Q_d \rightarrow Q_d$, k-1 K, JMP(TEST)

2.4.2. Transition OPERATEUR → CALLF

Une mise à jour de la file peut être nécessaire si $d_k=0$, puis on continue comme pour la transition ACCES → CALLF(étiquette SUITE).

Microprogramme:

1 : si $d_k \neq 0$ alors JMP(SUITE)

2 : $Q_t+t_{k-1} \rightarrow t_{k-1}$ et $Q_d+1 \rightarrow Q_d$, JMP(SUITE)

2.4.3. Transition FE ou CONTROLE → CALLF

Après avoir exécuté la procédure d'attente de sortie de l'état conditionnel, on génère

Microprogramme:

1 : JSR(ATTENTE(ERR, $\overline{ETC}$))

2 : attente(NEXTOC), CØ+1 → CØ, ordre(SEQ), 0 → 0_d

3 : attente(BIPAC-NOn-PLEIN), (RI,M) → BIPAC, JMP(EMPILE)

2.5. Traitement propre à OFn

L'instruction OF est une FIN d'EXPRESSION qui présente une caractéristique spéciale pour la seule transition où elle apparait: ACCES → FE. On génère en effet pour cette transition un extra-ordre appelé OF, avec un paramètre n égal au nombre de valeurs littérales qui étiquettent le cas courant. Le processeur POP peut ainsi comparer la valeur de l'expression qui précède le CASE avec les n valeurs immédiates fournies par le processeur FILE.

Microprogramme:

1 : attente(BIPOP-NON-PLEIN) $Y_d=Q_d$, (OF, Y_d) → BIPOP, JMP(CONTROLE)

2.6. Réalisation des opérations sur les suites $\{d_k\}$ et $\{t_k\}$ et sur le compteur ordinal

Les divers algorithmes à réaliser font apparaître un certain nombre d'opérations à faire sur les suites $\{t_k\}$ et $\{d_k\}$ ainsi que sur le compteur

ordinal. Ces opérations doivent être réalisées par les tranches SFC 92901. Elles sont listées ci-après:

poids forts	poids faibles	adresse
$Y = RAM(A)+D+C_{IN}$	$Y = RAM(A)+D$	CØ
$Y = RAM(A)$	$Y = RAM(A)$	CØ
$Y = RAM(A)+D+C_{IN} \rightarrow RAM(B)$	$Y = RAM(A)+D \rightarrow RAM(B)$	CØ
$Y = RAM(A)+1+C_{IN} \rightarrow RAM(B)$	$Y = RAM(A)+1 \rightarrow RAM(B)$	CØ/SP
	$Y = D \rightarrow RAM(B)$	CØ
$Y = Q$	-	-
-	$Y = RAM(A)$	k
$Y = RAM(A)$	-	k
$Y = Q+1 \rightarrow Q$		
$Y = Q-1 \rightarrow Q$	$Y = Q+1 \rightarrow Q$	-
-		
-	$Y = Q+RAM(A) \rightarrow RAM(B)$	k
-	$Y = Q+RAM(A) \rightarrow Q$	k
$Y = D \rightarrow Q$	$Y = D \rightarrow Q$	-
$Y = Q \rightarrow RAM(B)$	$Y = Q \rightarrow RAM(B)$	k
$Y = D$	$Y = D$	-
$Y = RAM(A)-1+C_{IN} \rightarrow RAM(B)$	$Y = RAM(A)-1 \rightarrow RAM(B)$	SP
$Y = RAM(A)$	$Y = RAM(A), RAM(A)-1 \rightarrow RAM(B)$	SP

Liste des opérations à réaliser

III- EXECUTION DES INSTRUCTIONS DE CONTROLE

Les instructions de contrôle présentent des caractéristiques différentes vis-à-vis du processeur PINS. En particulier, certaines instructions constituent une FIN D'EXPRESSION, d'autres instructions doivent être envoyées comme OPERATEUR au processeur POP.

C'est pourquoi l'interprétation des instructions de contrôle commence par un examen des caractéristiques.

3.1. Analyse des caractéristiques

La mémoire PROM de décodage des caractéristiques fournit 4 indicateurs qui sont:

1/ l'indicateur noté ATTBIT qui indique que le processeur PINS doit attendre d'être sorti de l'état conditionnel avant d'exécuter l'instruction.

2/ l'indicateur noté PACBIT qui indique que l'instruction doit être envoyée vers le processeur PAC.

3/ l'indicateur noté POPBIT qui indique que l'instruction doit être envoyée vers le processeur POP.

4/ l'indicateur noté CONDBIT qui indique que le processeur PINS doit se mettre dans l'état conditionnel.

Sachant que zéro, un, deux, trois ou quatre de ces indicateurs peuvent être présents, le problème se pose quant à l'ordre dans lequel ils doivent être testés.

1/ La première solution consiste à faire un test séquentiel, en respectant les priorités.

Microprogramme:

```
si ATTBIT  alors  JSR(ATTENTE[ERR, ETC])
si PACBIT  alors  JSR(SENDPAC[L])
si POPBIT  alors  JSR(SENDPOP)
si CONDBIT alors  [Ordre(ETC←1) et JMP([ROM])]
JMP([ROM])
```

Cette solution présente le gros inconvénient d'obliger à exécuter tous les tests, même dans le cas où aucun indicateur n'est présent. Elle n'est donc pas optimale pour le nombre de microinstructions exécutées, bien qu'elle le soit pour la taille du microprogramme.

2/ Une deuxième solution consiste à faire un aiguillage à n directions, en utilisant directement la valeur des n indicateurs pour construire une adresse de branchement dans le microprogramme , cette adresse correspondant exactement à la séquence de microinstructions à exécuter.

D'autre part, en cas de présence de l'indicateur d'ATTENTE, il est intéressant de tester simultanément cet indicateur et la sortie de la bascule ETC qui indique l'état conditionnel. On force donc la valeur $\overline{ETC}$ sur l'entrée OR_0 du séquenceur SFC92909 et on valide la valeur $\overline{ETC.ERREUR}$ qui indique que PINS s'est trompé dans son choix sur l'entrée $\overline{ZERO}$ du SFC92909.

Microinstruction d'aiguillage

JMP(CARAC), Valide(TROMPE), Valide($\overline{ETC}$)

L'adresse de l'instruction suivante est donc:

- ZERO si le processeur PINS S'est trompé,
- ("1", ATTBIT, PACBIT, LBIT, POPBIT, CONDBIT, $\overline{ETC}$) dans l'autre cas.

D'où le microprogramme:

3.1.1. Aucun bit n'est présent

Adresse = (1,0,0,0,0,0, $\overline{ETC}$)
On sélectionne la sortie de la ROM de décodage comme adresse suivante.
Microprogramme:

```
100000  JMP(ROM), Valide(TROMPE)
100001      "          "
```

3.1.2. Présence de POPBIT

Adresse = (1,0,0,0,1,0, $\overline{ETC}$)
On envoie RI dans BIPOP et on sélectionne la ROM comme adresse suivante.
Microprogramme:

```
1000100  attente(BIPOP), RI → BIPOP, JMP( ROM )
1000101      "                  "
```

3.1.3. Présence de POPBIT, PACBIT et LBIT

Adresse = (1,0,1,1,1,0, $\overline{ETC}$)
On envoie RI dans BIPOP et (RI,M) → BIPAC.
Microprogramme:

```
1011100  attente(BIPOP), RI → BIPOP, JMP(1011110)
1011101  attente(BIPOP), RI → BIPOP
1011110  attente(NEXTOC), CØ+1 → CØ, Ordre(SEQ)
1011111  attente(BIPAC), (RI,M) → BIPAC, JMP(ROM)
```

3.1.4. Présence de ATTBIT

Adresse = (1,1,0,0,0,0,$\overline{ETC}$)
On fait une attente sur la même microinstruction, en validant $\overline{ETC}$ sur l'entrée OR_0.

Microprogramme:

1100100 JMP(1100100), Valide($\overline{ETC}$), Valide(TR)

1100101 attente(BIPOP), RI → BIPOP, JMP(ROM)

3.1.6 Présence de ATTBIT, POPBIT et CONDBIT

Adresse = (1,1,0,0,1,1,$\overline{ETC}$)

Comme 3-1-5, puis on entre dans l'état conditionnel en mettant ETC à "1".

1100110 JMP(1100110), Valide($\overline{ETC}$), Valide(TR)

1100111 attente(BIPOP), RI → BIPOP, 1 → ETC, JMP(ROM)

3.1.7. Présence de ATTBIT, PACBIT et LBIT

Adresse = (1,1,1,1,0,0,$\overline{ETC}$)

Deux octets sont envoyés à BIPAC.

Microprogramme:

1111000 JMP(1111000), Valide($\overline{ETC}$), Valide(TR)

1111001 Attente(NEXTOC), CØ+1→CØ, Ordre(SEQ)

1111010 Attente(BIPAC), (RI,M)→BIPAC, JMP(ROM)

TABLEAU DES ADRESSES DE BRANCHEMENT

1000 000	Aucun bit présent	ETC = 1	suite
1000 001	"	ETC = 0	suite
1000 100	POPBIT et ETC et	ETC = 1	suite
1000 101	"	ETC = 0	suite
1011 100	POPBIT, PACBIT et	ETC = 1	suite
1011 101	"	ETC = 0	suite
1100 000	ATTBIT et	ETC = 1	Attente
1100 101	"	ETC = 0	suite
1100 100	ATTBIT et POPBIT et	ETC = 1	attente
1100 101	"	ETC = 0	suite
1100 110	ATTBIT, POPBIT, CONDBIT et	ETC = 1	attente
1100 111	"	ETC = 0	suite
1110 000	ATTBIT, PACBIT et	ETC = 1	attente
1110 001	"	ETC = 0	suite
1111 000	ATTBIT, PACBIT, LBIT et	ETC = 1	attente
1111 001	"	ETC = 0	suite

3.2. Exécution des instructions de contrôle

Le processeur PINS exécute, dans l'ordre

- le traitement des transitions (mise à jour de l'état de la file d'évaluation et génération des extra-ordres) ;

- le traitement des caractéristiques (attente, envoi à POP, envoi à PAC, entrée dans l'état conditionnel) ;

- et enfin le traitement de l'instruction de contrôle elle-même.

L'adresse de la séquence de traitement est contenue dans une ROM de décodage du code opération.

3.2.1. Entrée dans un segment de contrôle avec paramètre

Les instructions concernées par ce cas sont:

THEN m
LOOP m
FORUP m
FORDOWN m

Leur traitement consiste à empiler un descripteur de segment caractérisé par:

DEBUT = CØ+1 et FIN = CØ+m

Microprogramme

1: attente(NEXTOC), Charge RI, CØ+1 → CØ, Ordre(SEQ), JSR(PUSH)
2: attente(NEXTOC), CØ+1 → CØ et DEBUT, Ordre(SEQ)
3: CØ+M → FIN, JMP(FETCH), Valide(TR)

Les deux premières microinstructions réalisent l'accès au paramètre m de l'instruction.

3.2.2. Instruction ELSE

C'est une entrée dans un segment sans définition de l'adresse de FIN de segment.

Microprogramme

1: JSR(PUSH)
2: CØ → DEBUT, JMP(FETCH), Valide(TR)

3.2.3. Fin de boucle

Les instructions concernées par ce cas sont:

ENDLOOP
UNTIL
ROFUP
ROFDOWN

L'adresse du DEBUT du segment est prise comme adresse suivante, mais le descripteur de segment n'est pas dépilé.

Microprogramme

1: si VIDE alors JSR(ACCES)
2: DEBUT → CØ et ADMC, Odre(INIT), JMP(FETCH)

3.2.4. Test de sortie de boucle

Les instructions concernées par ce cas sont:

WHILE
EXITIF

Le processeur PINS n'a aucun traitement à faire, l'adresse fournie par la ROM est donc celle de la séquence de FETCH.

3.2.5. Sortie de segment de contrôle

Les instructions concernées sont:

RETURN
FØ

L'adresse de FIN de segment est prise comme adresse de l'instruction suivante et le descripteur de segment est dépilé.

3.2.6. Destruction de segment

Les instructions concernées par ce cas sont:

EXIT

EXITFOR

Microprogramme

JMP(FETCH), Valide(TR)

3.2.7. Sortie de segment THEN, instruction NEHT m

Le processeur PINS réalise une sortie de segment et un branchement à l'adresse CØ+m.

Microprogramme

1: attente(NEXTOC), Charge RI, CØ+1 → CØ, ordre(SEQ)

2: attente(NEXTOC), CØ+1 → CØ

3: CØ+M → CØ et ADMC, ordre(INIT)

3.2.8. Instruction de branchement: GOTOm

Le déplacement m est relatif au DEBUT du segment courant.

Microprogramme

attente(NEXTOC), charge RI, DEBUT(K) → CØ, ordre(SEQ)

attente(NEXTOC), CØ+M → CØ et DMC, ordre(INIT) JMP(FETCH), valide(TR)

3.2.9. Entrée dans une structure CASE

instruction: CASE m

Un segment de contrôle est construit, avec une adresse de DEBUT encore inconnue (elle sera déterminée par le processeur POP) et une adresse de FIN égale à CØ+m.

Microprogramme

1: attente(NEXTOC), charge RI, CØ+1 → CØ, ordre(SEQ)

2: attente(NEXTOC), CØ+1 → CØ, ordre(SEQ), JSR(PUSH)

3: CØ+M → FIN(K), JSR(PUSH)

4: JMP(FETCH), Valide(TR)

3.2.10. Début d'un cas: instruction OFm

Le cas présent n'est exécuté que si le processeur PINS s'est trompé.

Le paramètre m donne l'adresse du début du cas suivant qui est "préparé".

Microprogramme

1: attente(NEXTOC), charge RI, CØ+1 → CØ, ordre(SEQ)

2: attente(NEXTOC), CØ+1 → CØ, ordre(SEQ),JSR(POP)

3: CØ -> DEBUT, JSR(PUSH)

4: CØ -> FIN

5: CØ+M → CØet ADMC, ordre(INIT), JMP(FETCH), valide(TR)

3.2.11.Fin de tous les cas: instruction ESAC

Deux segments de contrôle doivent être dépilés et on continue en séquence.

Microprogramme

1: JSR(POP)

2: JSR(POP)

3: JMP(FETCH), Valide(TR)

3.3. Traitement d'une erreur de choix

Le processeur PINS anticipe sur le choix d'une alternative, dans le cas d'un branchement conditionnel. Le processeur POP le fait sortir de l'état conditionnel dans lequel il se trouvait (ETC0) ou au contraire positionne la bascule d'erreur (ERREUR1).

Donc, dans le cas où ETC=1 et ERR=1, le processeur PINS doit exécuter une séquence de reprise d'erreur qui consiste à prendre la FIN du segment courant comme adresse de l'instruction suivante et à dépiler le descripteur du segment.

Le cas d'erreur forçant l'entrée $\overline{\text{ZERO}}$ du séquenceur, cette séquence commence à l'adresse ZERO.

Microprogramme

```
0000 0000 : si VIDE alors JSR(ACCES)
            FIN → CØ et ADMC, ordre(INIT), ETC → 0, ERR → 0, ETAT→11,JSR(POP)
            JMP(FETCH)
```

Remarque: l'automate des transitions est forcé dans l'état CONTROLE(ETAT ← 11)

3.4. Séquence de "FETCH" d'une instruction

Cette séquence consiste à attendre que le premier octet constituant l'instruction soit fourni par le module d'accés. Dés qu'il est disponible, il peut être chargé dans le registre instruction RI, l'octet suivant est demandé (ordre SEQ) et le compteur ordinal incrémenté (CØ+1 → CØ). Le code opération contenu dans RI définit les caractéristiques de l'instruction, qui elles-mêmes définissent l'adresse de la séquence correspondant à la transition. On se branche donc à l'adresse définie par la mémoire PROM appelée TRANS.

Microprogramme

```
FETCH : attente(NEXTOC), CØ+1 → CØ, charge RI, ordre(SEQ)
FETCH 1 : JMP(TRANS), charge ETAT, valide(TROMPE)
```

D - CONCLUSION

Cette description du processeur PINS a montré son caractère original de station principale du pipeline, qui requiert de sa part une grande spécialisation pour traiter des problèmes aussi divers que la gestion du Compteur Ordinal, la construction du modèle de la file d'évaluation et la génération des instructions internes à la machine vers les deux stations secondaires PAC et POP. Ses performances ne semblent cependant pas être critiques.

CHAPITRE 4

LE PROCESSEUR D'ACCES AUX DONNEES PAC

A - INTRODUCTION

Le processeur PAC réalise la fonction d'Accès aux opérandes qu'il range, par l'intermédiaire du processeur FILE, dans la file d'évaluation. Il reçoit principalement des instructions d'accès appelées NOM, dont le paramètre (s,d) permet de calculer l'adresse d'un descripteur, et les instructions LITERAL qui permettent d'insérer une valeur littérale comme opérande.

De plus, il réalise la gestion de la pile de contexte dont les modifications sont commandées par les instructions d'appel (CALL), d'entrée (ENTER) et de retour (RETURN) de procédure.

La définition du processeur PAC a bien entendu été menée d'une manière descendante (fonction à réaliser → implantation des variables → réalisation microprogrammée → réalisation physique). Cette démarche n'est pas explicitée ici, mais elle est sous-jacente. Afin de faciliter la compréhension de l'exposé, nous présentons dans la figure suivante l'architecture interne définie pour le processeur PAC, en insistant sur le fait que cette architecture est le résultat de l'étude des fonctions à réaliser.

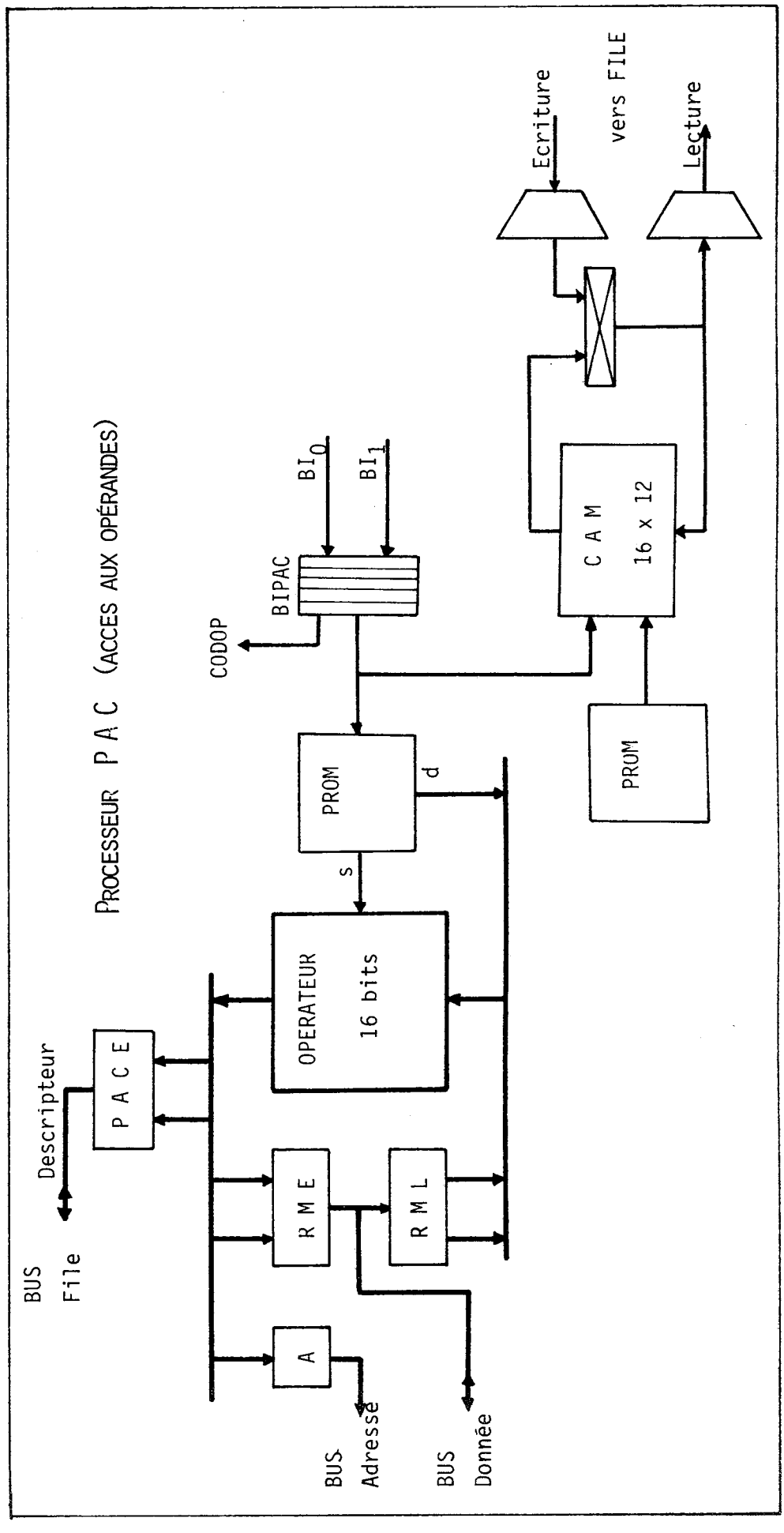
En avant propos de ce chapitre, nous présentons la synchronisation entre les processeurs PAC et FILE, basée sur un mécanisme de "demande-acquittement". Puis le calcul d'adresse, la gestion de la Mémoire Associative constituant la file des dépendances, la construction des valeurs littérales et la réalisation des instructions d'appel et retour des procédures sont successivement présentés.

B - SYNCHRONISATION ENTRE PAC ET FILE

Le processeur PAC demande les services du processeur FILE pour ranger un descripteur dans la file d'évaluation (instruction d'ACCES) ou pour initialiser une dépendance (instruction d'AFFECTATION).

Une bascule de synchronisation, appelée DPAC, est mise à 1 par PAC lors d'une demande, et mise à 0 par FILE lorsqu'il a effectué le travail demandé.
Les paramètres de la demande sont:

- une bascule CODEPAC, chargée à partir de la valeur de BIPAC 14, qui distingue les instructions d'accès et celles d'affectation ;
- la valeur BIPAC(10) qui distingue AFFECT et PARAM;
- une bascule TROUVE, résultat de la recherche associative ;
- un signal RESOLU qui indique l'état de la dépendance trouvée.


PROCESSEUR P A C (ACCES AUX OPÉRANDES)
Ecriture
vers FILE
Lecture
BI0
BI1
BIPAC
CODOP
C A M
16 x 12
PROM
d
s
PROM
OPERATEUR
16 bits
Descripteur
P A C E
BUS
File
R M E
R M L
A
BUS
Adresse
BUS
Donnée

C - LE CHEMIN DE DONNEE DU PROCESSEUR PAC

Le processeur PAC communique avec la Mémoire Centrale par l'intermédiaire de deux tampons RML et RME réalisés par le circuit Am2907 : il lit des descripteurs de variables et des mots de contrôle (marques de blocs), il écrit des mots de contrôle et sauvegarde ses registres de base.

Il fournit des descripteurs de variables au processeur FILE, par l'intermédiaire du tampon PACE, pour que ce dernier les range dans la file d'évaluation FILEVAL.

Il reçoit ses instructions du processeur PINS par l'intermédiaire de la file d'attente BIPAC : les opérandes des instructions sont soit des NOMS (S,D) qui doivent être décodés, soit des valeurs immédiates.

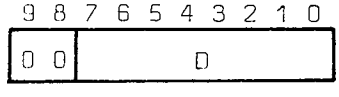
I - RAPPELS SUR LES MODES D'ADRESSAGE

Le NOM des variables du programme en cours d'exécution apparaît dans les instructions d'accès comme un couple

(Niveau Lexicographique = S , Déplacement = D).

Afin de satisfaire aux caractéristiques des programmes tout en minimisant la taille des instructions d'accès qui sont très fréquentes, il a été retenu le codage suivant du NOM d'une variable :

- Adressage des variables globales :



Le champ D représente un nombre positif compris entre 0 et 255, ce qui permet d'adresser jusqu'à 256 variables globales.

- Adressage des variables non globales

9	8	7	6	5	4	3	2	1	0
0	1	0	s				d		
0	1	1	s				d		
1	0	0	s				d		
1	0	1	s				d		
1	1	0	s				d		

Le déplacement d représente un entier signé, de signe s, compris entre -64 et +63, ce qui permet d'adresser jusqu'à 64 variables et 62 paramètres dans une procédure (les déplacements -1 et -2 sont réservés à la marque du bloc; le premier paramètre a pour déplacement -3).
On peut d'autre part avoir jusqu'à 6 niveaux d'imbrication lexicographique, ce qui représente une limitation raisonnable pour la complexité statique d'un programme, sachant que la décomposition en modules est toujours possible et même souhaitable (voir chaplitre).

- Adressage relatif au sommet de pile SP

9	8	7	6	5	4	3	2	1	0
1	1	1	1				d		

Entre le début du passage des paramètres défini par l'instruction CALL et l'entrée effective dans une procédure définie par l'instruction ENTER, les paramètres sont adressés par rapport au sommet de pile SP qui pointe sur la future base du bloc alloué à la procédure.
Au moment de l'entrée dans la procédure, le NOM des paramètres doit être changé (voir Recherche Associative, paragraphe 3.4.3.)

- Adressage relatif au registre W

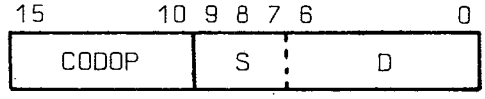
9	8	7	6	5	4	3	2	1	0
1	1	1	0				d		

L'instruction WITH du langage PASCAL permet de définir un environnement d'adressage particulier à l'intérieur d'une procédure, pour accélérer

l'accès aux champs d'une même structure (RECORD) : le descripteur d'une ou plusieurs variables structurées est mis en "facteur" et il est adressé directement par rapport à un registre de base spécial appelé W. .

II - DECODAGE DU NOM D'UNE VARIABLE

Les instructions d'accès contenant le NOM d'une variable ont le format suivant:



Le calcul de l'adresse du descripteur d'une variable à partir de son NOM consiste à ajouter un déplacement (entier relatif) à la valeur du registre de base correspondant au niveau lexicographique de la variable (S).
Deux problèmes se posent:
-1/ générer un numéro de registre de base à partir du champ S
-2/ générer un déplacement signé sur 16 bits à partir du champ D.
La solution la plus simple consiste à utiliser une petite mémoire PROM de décodage dont l'adresse est donnée par les bits 6,7,8 et 9 de l'instruction.
La sortie de cette PROM définit:
- un numéro de registre de base envoyé sur une entrée "adresse de RAM" de l'opérateur Am2901,
- un bit de signe qui permet d'étendre le déplacement à 16 bits avant de l'envoyer sur l'entrée "D" de l'opérateur Am2901.
Cette solution permet de faire "en un cycle" le calcul Base + Déplacement.

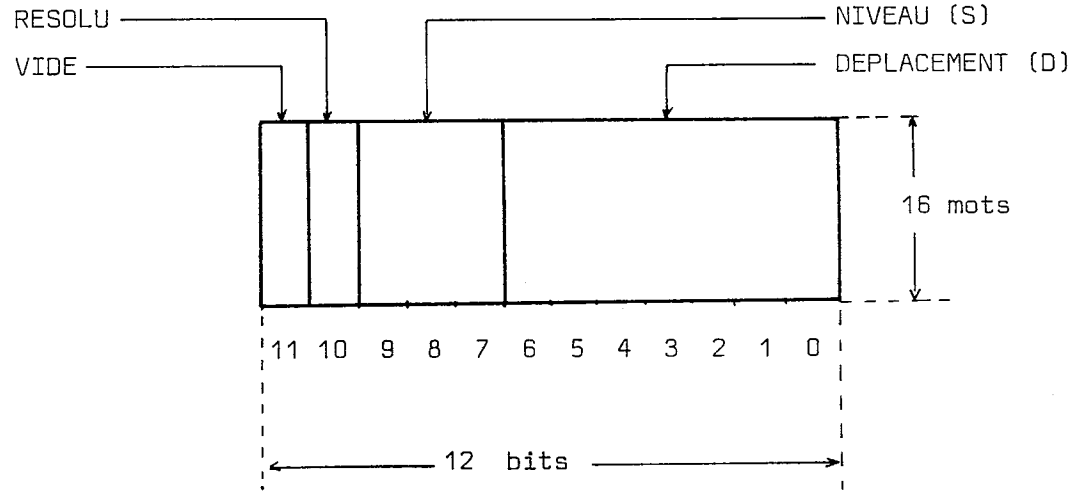
D - GESTION DE LA MEMOIRE ASSOCIATIVE

La Mémoire Associative est destinée à résoudre le problème des dépendances (voir chapitre 2/B). L'adresse des informations qu'elle contient est identique à celle des descripteurs de dépendances, qui sont contenus dans la File des dépendances (mémoire RAM) gérée par le processeur FILE.

I - FORMAT DES INFORMATIONS

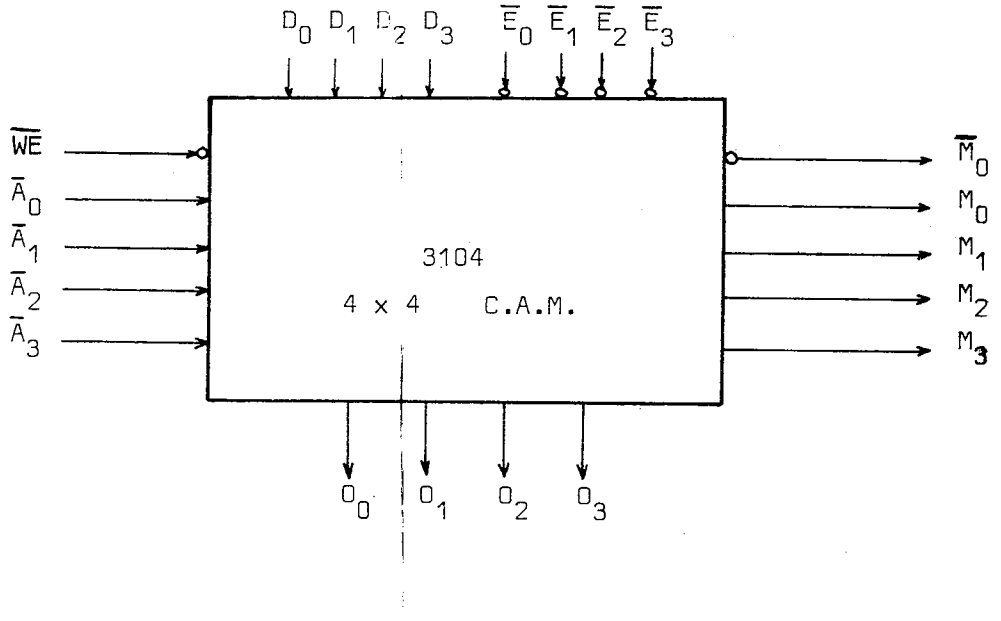
Les informations contenues dans la Mémoire Associative représentent :
- le NOM d'une variable, codée sur 10 bits, où 3 bits indiquent le numéro lexicographique (voir chapitre),
- l'indicateur RESOLU qui signale que la variable est ou n'est plus sous dépendance,
- l'indicateur VIDE qui signale l'occupation d'un mot de la mémoire.

D'où la structure suivante :

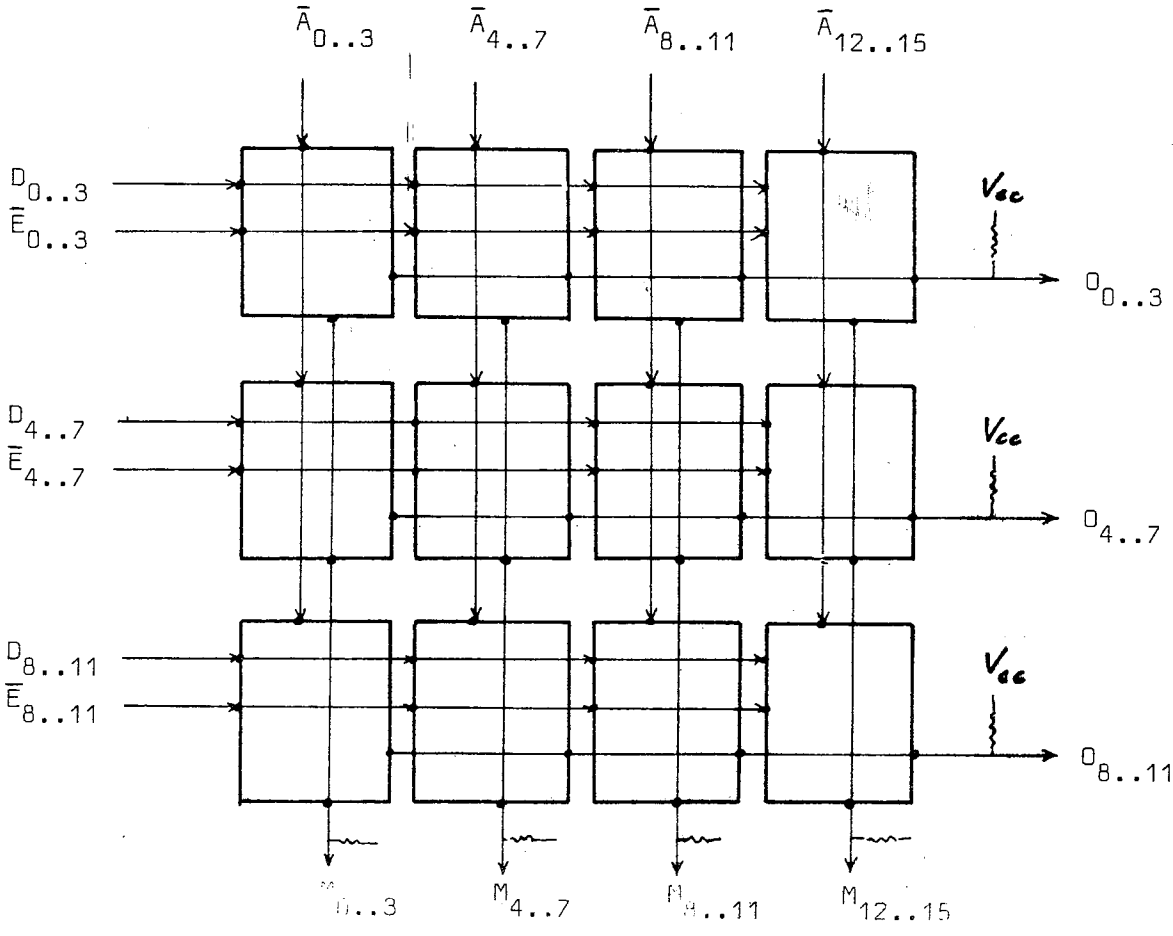


II - DESCRIPTION DE LA MEMOIRE ASSOCIATIVE

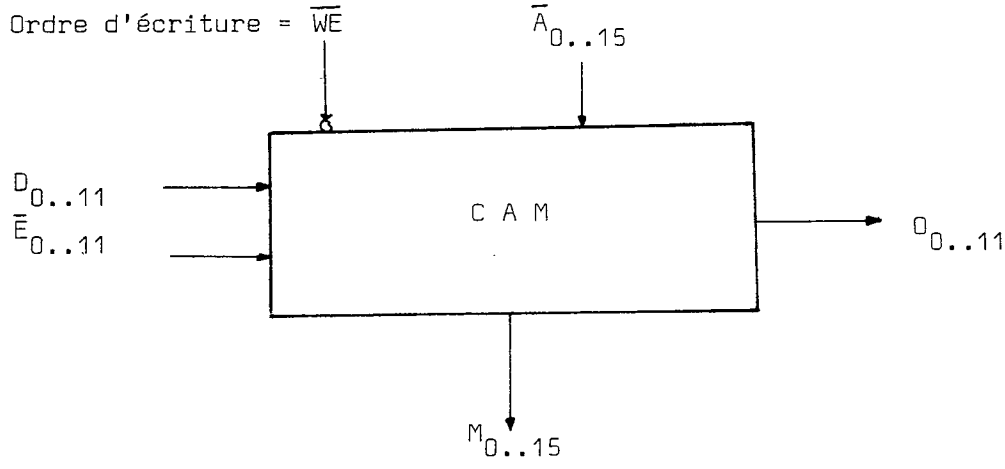
Le composant électronique utilisé est le circuit de Mémoire Associative INTEL 3104 ("Content Addressable Memory" ou CAM), qui se présente comme une matrice de 4 mots de 4 bits. Il est conçu pour permettre de comparer des données d'entrée D_i avec les valeurs Q_i de ses cellules de mémorisation. Les sorties M_j indiquent qu'il y a correspondance ("match"). Cette mémoire peut être utilisée en lecture ou écriture, comme une mémoire RAM, avec la particularité que l'adresse n'est pas codée.



La réalisation d'une configuration de 16 mots de 12 bits requiert la construction d'une matrice de 4 fois 3 circuits INTEL 3104 comme l'indique la figure suivante :



Nous représenterons désormais la matrice précédente par la boîte noire suivante, que nous appellerons CAM :



III - NATURE DES OPERATIONS SUR LA MEMOIRE ASSOCIATIVE

3.1. Initialisation

Au début de l'exécution d'un programme, la CAM doit être initialisée en forçant l'indicateur VIDE à la valeur '1' pour tous les mots. Sachant que l'on peut écrire en parallèle dans plusieurs mots, il suffit de générer une adresse égale à 00...0 en entrée de la CAM.
Cette adresse peut être obtenue après une recherche associative obtenue en ne validant aucun des bits. On obtient ainsi une correspondance pour tous les mots, donc un vecteur de sortie M égal à 11...1, qu'il suffit de complémenter.

Remarque : le fonctionnement de la CAM est conforme à la logique en ce sens que la proposition "n'importe quoi égale n'importe quoi" est toujours vraie.

Algorithme d'initialisation :

1 Recherche associative avec $\bar{E}_i$=1 pour i=0..11
donne M_j=1 pour j=0..15
2 Ecriture avec $\bar{A}_j = \bar{M}_j$ pour j=0..15, $\bar{E}_{11}$=0 et D_{11}=1

3.2. Ecriture d'un nouveau NOM dans la CAM

Lorsque le processeur PAC reçoit une instruction AFFECT(S,D), il demande au processeur FILE de créer un descripteur de dépendance. Ce dernier dispose d'un pointeur appelé NIN, qui indique la position du premier mot libre dans la file des Dépendances et la CAM.

Le processeur File doit donc écrire (S,D,RESOLU,VIDE) à l'adresse indiquée par le pointeur NIN. Cette adresse codée sur 4 bits est décodée par un circuit "décodeur 4->16" (74 154), dont la sortie est telle que :

$\bar{S}_{NIN}=0$ et $\bar{S}_j=1$ pour j=0..15 et j≠NIN.

Les valeurs des champs S et D proviennent de la sortie du FI-FO d'instructions BIPAC.

On force D_{10} et D_{11} à 0, et on valide tous les bits par $\bar{E}_i=0$ pour i=0..11.

3.3. Résolution d'une dépendance

Le processeur FILE, à la demande du processeur POP, peut supprimer l'ancienne occurrence d'une variable (par $D_{11}=1$ et $\bar{E}_{11}=0$), ou bien marquer une dépendance comme étant RESOLUe (par $D_{10}=1$ et $\bar{E}_{10}=0$) en écrivant dans la CAM.
L'adresse utilisée pour l'écriture provient d'une recherche associative antérieure , ou est obtenue par décodage de la valeur d'un pointeur.

3.4. Recherche associative

3.4.1. Lorsqu'une instruction d'accès est décodée, le processeur PAC recherche si le nom (S,D) est présent dans la CAM. Il valide donc les bits 0 à 9 et le bit 11 ($\bar{E}_0=...=\bar{E}_9=\bar{E}_{11}=0$), avec $D_{0..9}=(S,D)$ et $D_{11}=0$.
Le résultat de la recherche est un vecteur de 16 bits. La présence de l'information chechée est indiquée par l'existence d'un bit égal à 0 dans le vecteur. Un encodeur de priorité indique cette présence, et permet d'encoder l'adresse sur 4 bits (voir sa réalisation plus loin).

3.4.2. Le changement de contexte qui se produit lors d'un appel de procédure (ou de fonction) nécessite une recherche associative sur le champ qui contient le niveau lexicographique S, pour éviter les conflits possibles en cas de récursivité (voir chapitre).
Cette recherche est faite en validant les bits 7 à 9 et le bit 11,
avec $D_{7..9}=S$ et $D_{11}=0$.
On obtient ainsi un vecteur de correspondance pouvant avoir plusieurs zéros, et donnant les positions de tous les mots qui contiennent des NOMS de niveau S. Tous ces mots sont marqués VIDEs ($\bar{E}_{11}=0$ et $D_{11}=1$) en utilisant le vecteur de correspondance comme adresse d'écriture.

3.4.3. Lors d'un passage de paramètres, avant l'entrée dans une procédure, les NOMs des paramètres sont rangés dans la CAM pour chaque instruction PARAM (S,D). Le niveau lexicographique spécifié fait référence au sommet de pile SP comme registre de base, et le nom (S,D) se présente sous la forme :

9 8 7 6 5 4 3 2 1 0

1 1 1 1 x x x x x x

Sachant qu'à l'intérieur d'une procédure tout paramètre est référencé comme une variable locale de déplacement négatif, son NOM sera différent de celui qui figure dans la CAM. Afin de bénéficier du mécanisme des dépendances qui est très efficace pour les paramètres, nous allons changer les NOMs des paramètres de 111 en S au moment de l'entrée dans la procédure (instruction ENTER).

On fait une recherche associative en validant les bits 6 à 9 et en affichant $D_6=...=D_9=1$. Puis on modifie les bits 7 à 9, dans les mots où il y a correspondance, par $\bar{E}_7=...=\bar{E}_9=0$ et $D_{7..9}$=NIVEAU.

3.4.4. Récapitulatif des opérations sur la CAM

	$\bar{E}_{0..5}$	$\bar{E}_6$	$\bar{E}_{7..9}$	$\bar{E}_{10}$	$\bar{E}_{11}$	D_6	$D_{7..9}$	D_{10}	D_{11}
Processeur PAC									
Rech.S,D,V	0	0	0	1	0	BI	BI	x	0
Rech.S,V	1	1	0	1	0	x	NIV	x	0
Rech.1111,V	1	0	0	1	0	1	111	x	0
Rech.PARTOUT	1	1	1	1	1	x	x	x	x
Ecr.V	1	1	1	1	0	x	x	x	1
Ecr.NIV	1	1	0	1	1	x	NIV	x	x
Processeur FILE									
Ecr.S,D,V,R	0	0	0	0	0	BI	BI	0	0
Ecr.V	1	1	1	1	0	x	x	x	1
Ecr.R	1	1	1	0	1	x	x	1	x

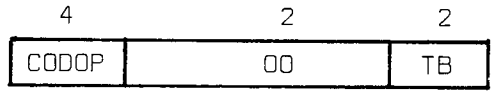
Légende: (S,D)=Nom de variable, V=Vide, R=Résolu, BI=FI-FO d'instructions BIPAC
NIV=Niveau Courant de la procédure

E - CONSTRUCTION DES LITTERAUX

Le processeur PAC reçoit, dans sa file d'attente d'instructions BIPAC, des instructions d'accés immédiat qui contiennent des valeurs littérales.

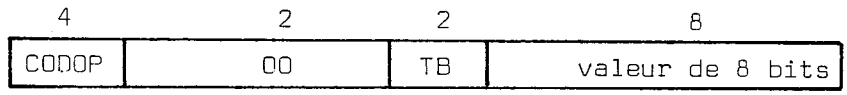
Ces instructions sont:

1 - les instructions ZERO et ONE dont le format est:

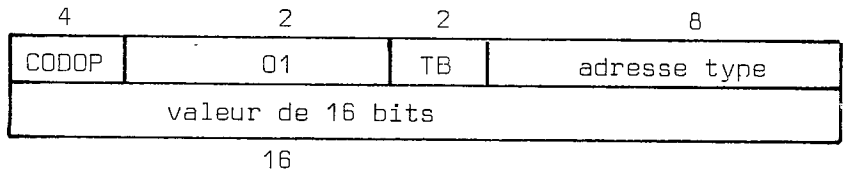


2 - les instructions LITERAL dont les formats sont:

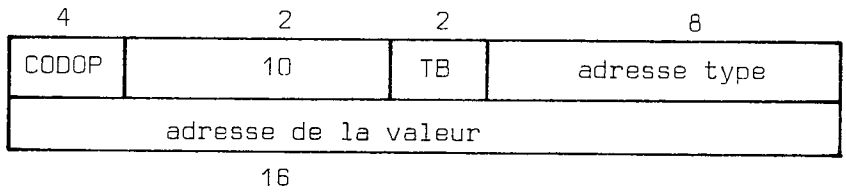
LIT 8



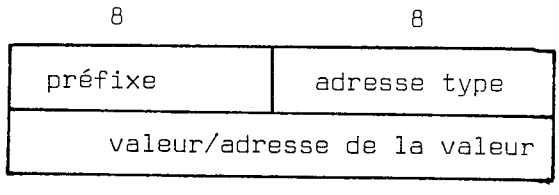
LIT 16



LIT 32



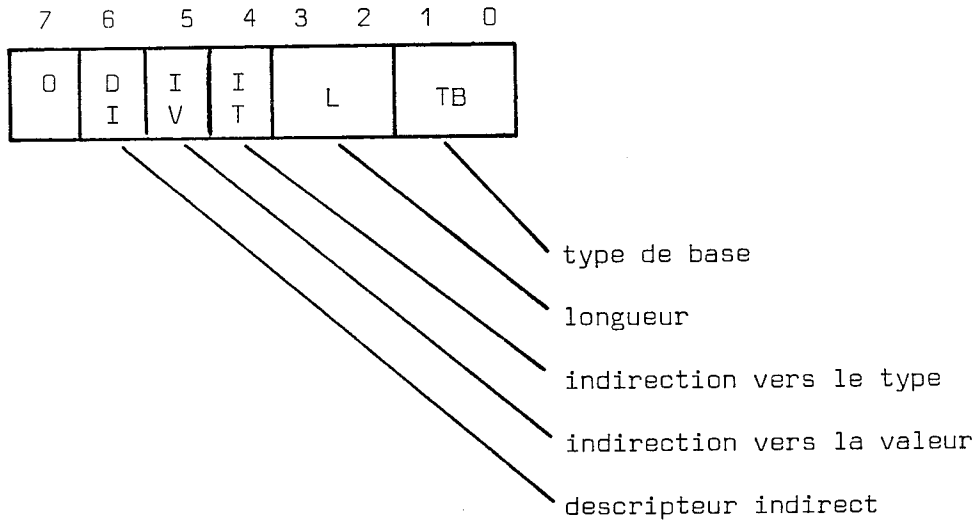
Le processeur d'accés doit construire, à partir d'une telle instruction, un descripteur de variable dont on rappelle ici le format:



1 - Construction du préfixe

La construction du préfixe est simplifiée par le fait que le premier octet de l'instruction en contient la moitié correspondant aux champs longueur (L) et type de base (TB).

Format du préfixe:



- instructions ZERO et ONE

DI	IV	IT			
0	0	0	1	00	TB

- instruction LITERAL de longueur 00 (1 octet)

0	0	0	1	00	TB

- instruction LITERAL de longueur 01 (2 octets)

0	0	0	1	01	TB

- instruction LITERAL de longueur 10 (2 octets)

0	1	0	1	10	TB

indirection sur la valeur

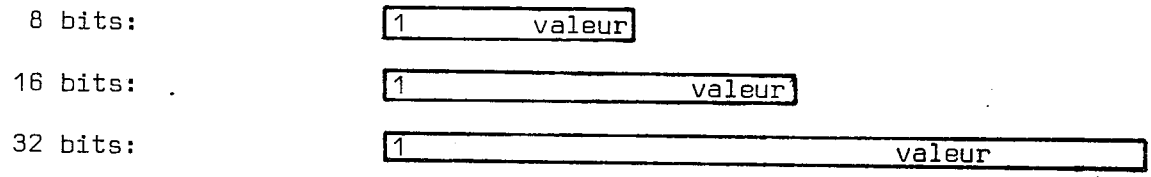
2. Construction du champ valeur/adresse de la valeur

Les valeurs des variables manipulées par la machine sont toutes caractérisées

par une indirection d'initialisation, indispensable pour évrifier, au cours de l'exécution d'un programme, que toutes les valeurs référencées sont effectivement initialisées, et ont bien un sens.

La présence de cet indicateur auprès de chaque valeur diminue évidemment par deux l'ensemble des valeurs possibles pour une chaîne de bits de longueur donnée, mais répond à un besoin crucial de vérification dynamique du comportement d'un programme, qui ne peut être assuré que de cette manière.

Format des valeurs initialisées



Remarque:

Les valeurs ne sont marquées que lorsqu'elles sont en mémoire centrale, dans un descripteur de variable ou dans la zone valeur associée à une variable structurée.

a/ construction des valeurs 0 et 1

Une valeur de 8 bits, avec son indicateur d'initialisation, est construite à partir de l'opérateur 16 bits du processeur PAC:

- l'opérateur donne une valeur nulle,
- le bit 7 est forcé à 1,
- le bit 0 est pris dans le code opération (bit 12) dans lequel on distingue les instructions

ZERO (101000xx) et ONE (101100xx)

bit 12

b/ construction des autres valeurs

Dans les autres cas (LIT8 et LIT16) les valeurs sont déjà construites dans l'instruction. Il suffit donc de transférer le contenu de BIPAC dans le champ valeur du descripteur construit.

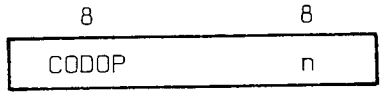
c/ construction de l'adresse de la valeur littérale

Le paramètre figurant dans le deuxième demi-mot composant l'instruction LIT32 représente le déplacement de la valeur littérale relatif à la base des constantes du programme couramment exécuté. Sachant qu'un changement de programme peut se produire, à cause des possibilités de compilations séparées, la valeur de la base des constantes peut changer et cette valeur doit être gérée par le processeur POP.

On indique donc dans le préfixe que le champ PV du descripteur représente un déplacement par rapport à la base des constantes dont POP connait la valeur.

3. Extension du signe d'un paramètre

Le processeur PAC reçoit des opérations spéciales relatives à la modification du sommet de pile (MONTE et DESCEND) dont le format est le suivant:



Le paramètre n doit être étendu à 16 bits, pour être ajouté à la valeur d'un registre interne (SP par exemple).

L'extension est faite par la mémoire ROM qui décode le nom des variables (cf. II). Un bit de contrôle est introduit en adresse poids fort de cette ROM, de manière à distinguer les deux fonctions.

Pour les adresses 16 à 31, le contenu de la ROM définit l'extension à 16 bits.

F - LES INSTRUCTIONS D'APPEL ET DE RETOUR DE PROCEDURE

Une caractéristique importante de la machine PASCALE est représentée par le fait que l'appel d'une procédure se fait en deux temps, définis par l'instruction CALL et l'instruction ENTER, entre lesquels les paramètres sont évalués et effectivement "passés", selon la syntaxe suivante:

```
CALL    #S
    <exp0>
PARAM   SP, -3
    <exp1>
PARAM   SP, -4
   :
   :
    <expn>
param   SP, - (3+n)
ENTER   #S
```

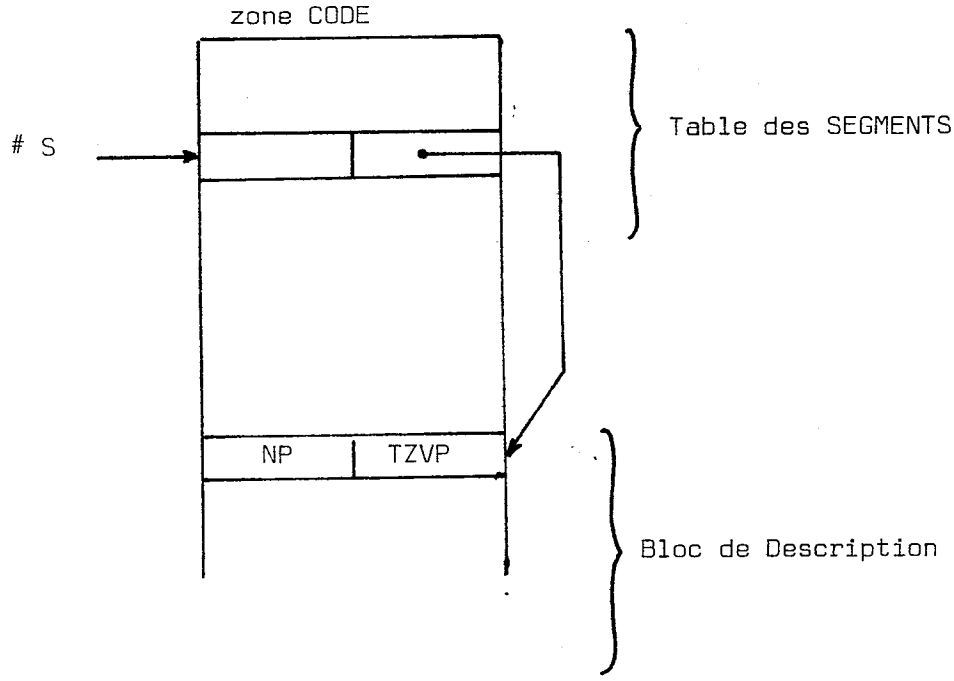
L'exécution de l'instruction CALL prépare le passage des paramètres, sans changer le contexte d'adressage et l'instruction ENTER change le contexte et créé les descripteurs des variables locales.

I - L'instruction CALL

Cette instruction a pour paramètre le n° du segment #S, associé à la procédure dont le descripteur se trouve dans la "table des segments" (voir § SYSTEME) au déplacement #S par rapport au début de la zone CODE ou EXTERNE selon le mode.

.a/ lecture du bloc de description du segment

Le descripteur du segment (situé au déplacement #S de la zone CODE ou EXTERNE) contient l'adresse dans cette zone du bloc de description des paramètres de la procédure.



Microprogramme:

D = ROM(1) = paramètre, Y=D, Y→RADM, LMC(CODE/EXT)

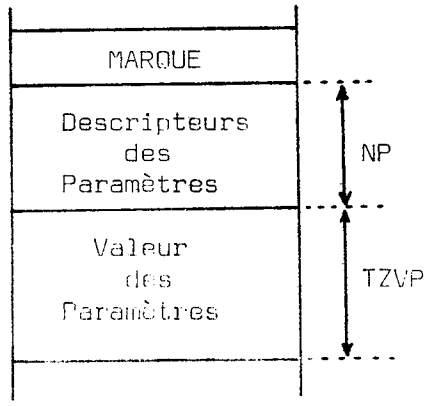
Attente(MC), D=RML_0, Y=D→Q, Y→RADM, LMC(CODE/EXT)

L'accumulateur Q contient alors l'adresse du premier mot du bloc de description des paramètres.

.b/ <u>préparation de la zone des paramètres</u>

Le premier mot du bloc de description, en cours de lecture, contient le nombre de paramètres NP et la taille de la zone valeur des paramètres TZVP.

Image de la pile de contexte:



La base des valeurs des paramètres est la valeur courante de SP, qui est chargée dans le registre de travail BV. Puis la taille de la zone valeur des paramètres TZVP est mise dans un registre de travail RT et le registre SP est incrémenté.

Microprogramme:

```
        SP→BV
        Attente(MC), D=RML0, D→RT
        SP+RT→SP, si ZERO alors JMP(NP)
```

Dans le cas où TZVP=0, on passe à l'étiquette NP pour tester le nombre de paramètres. Dans le cas contraire (TZVP≠0), on met à ZERO toute la zone pointée par BV et sur une longueur TZVP.

Microprogramme:

```
        Y = BV→AD, Y→RADM
        Y = 0→RME0
        Y = 0→RME1, EMC(CTX)
BOUCLE: RT-1→RT
        si ZERO alors JMP(NP)
        Attente(MC), Y=AD+1→AD, Y→RADM, EMC(CTX), JMP(BOUCLE)
```

Le nombre de paramètres NP est lu dans RML_1 et mis dans le registre NP, il est testé à zéro et SP est incrémenté.

Microprogramme:

```
NP: D = RML1, Y=D→NP
    SP+NP→SP, si ZERO alors JMP(MARQUE)
```

Dans le cas où le nombre des paramètres n'est pas nul (NP≠0) les descripteurs des paramètres sont recopiés.

.c/ recopie des descripteurs de paramètres

Les descripteurs de paramètres se trouvent dans le bloc de description, à l'adresse contenue dans 0 plus 1. Ils sont lus un à un et recopiés sur la pile de

contexte à partir de l'adresse SP moins 1. La valeur de NP est décrémentée, dans le registre RT, pour contrôler la boucle de recopie.

Microprogramme:

```
          NP→RT
          SP→AE
BOUCLE:   Attente(MC), Y=Q+1→Q, Y→RADM, LMC(CODE/EXT)
          Attente(MC), D=RML1, Y=D, Y→RME1
          D=RML0, Y=D, Y→RME0, si IV alors JMP(*+2)
          D=RML0, Y=D+BV, Y→RME0
          Y=AE-1→AE, Y→RADM, EMC(CTX)
          RI-1→RT
          si NONZERO alors JMP(BOUCLE)
```

On remarquera que dans le cas où le paramètre contient une indirection sur la valeur (IV=1), on ajoute BV au déplacement contenu dans RML_0 (champ PV).

.d/ préparation de la marque du bloc

L'ancienne valeur de SP, contenue dans BV, est sauvegardée au sommet de la pile, ainsi que la valeur courante de lecture dans le bloc de description, contenue dans Q.

Microprogramme:

```
MARQUE:   Attente(MC), Y=SP, SP+1→SP, Y→RADM
          Y=Q+1, Y→RME1
          Y=BV, Y→RME0, EMC(CTX)
```

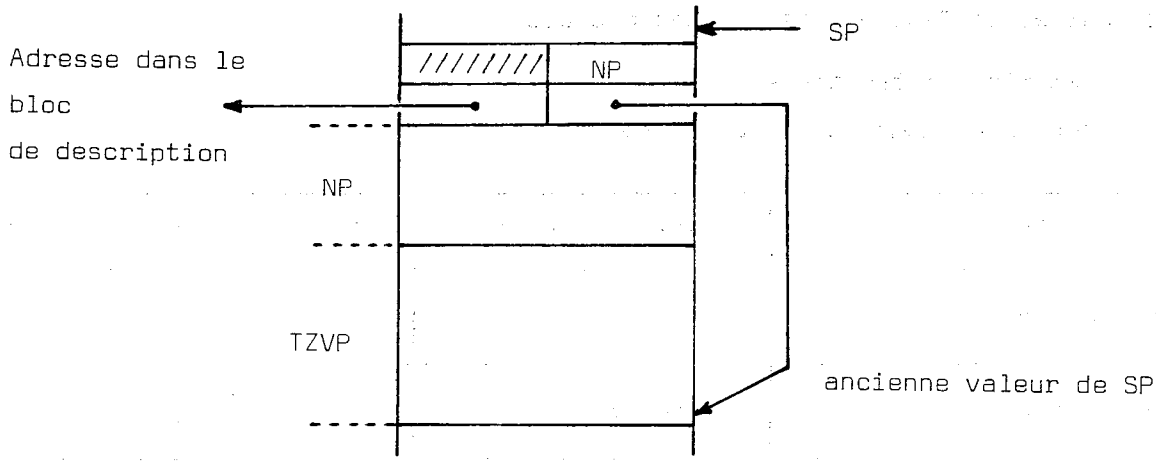
Le nombre des paramètres, contenu dans NP, est également empilé, qui sera décrémenté par les instructions PARAM, et dont la valeur ZERO sera testée par l'instruction ENTER afin d'assurer la vérification dynamique du nombre des paramètres (utile pour les compilations séparées et pour le passage de procédures comme paramètres).

Microprogramme :

Attente(MC), Y = SP, SP+1 → SP, Y → RADM

Y=NP, Y → RME_0, EMC(CTX)

A la fin de l'exécution de l'instruction CALL on a l'image suivante pour la pile de contexte:



On vérifie enfin que la valeur de SP n'est pas supérieure à celle de la limite de la pile de contexte.

Microprogramme:

LIMITE - SP

si NEGATIF alors JMP(ERREUR-DEPASSEMENT)

.e/ performances de l'instruction CALL

- dans le cas (assez fréquent) d'un appel de procédure sans paramètre il faut:
  - 12 microinstructions
  - 2 lectures Mémoire (δ de segment+descripteur)
  - 2 écritures Mémoire (marque de bloc).

- dans le cas (très fréquent) d'un appel de procédure ayant N paramètres sans indirection sur la valeur (TZVP=0), il faut:

14+6*N	microinstructions	(26 pour N=2)
2+N	lectures mémoire	(4 pour N=2)
2+N	écritures mémoire	(4 pour N=2)

- dans le cas (peu fréquent) d'une zone valeur des paramètres de taille M non nulle, il faut ajouter au cas précédent:

3+3*M microinstructions

M écritures mémoire (mise à zéro)

conditions	#microinstructions	#lectures	#écritures
0 paramètre	12	2	2
N paramètres(M=0)	14+6*N	2+N	2+N
N paramètres(M≠0)	17+6*N+3*M	2+N	2+N+M

II - L'instruction ENTER

Cette instruction est exécutée après le passage des paramètres.

.a/ Vérification du nombre des paramètres

Chaque instruction PARAM décrémente le nombre des paramètres localisé au sommet de la pile de contexte. Il contient ZERO si tous les paramètres ont effectivement été passés (leur nombre est défini par la déclaration de la procédure et donc par son bloc de description, et non par l'endroit d'où elle a été appelée).

Microprogramme:

Y = SP-1 → SP, Y → RADM, LMC(CTX)
Attente(MC), D=RML_0, Y=D
si NONZERO alors JMP(ERREUR-PARAM).

.b/ Mise à jour du chaînage dynamique

Le sommet de pile contient l'adresse courante dans le bloc de description de la procédure, qui est chargée dans Q et remplacée par le chaînage dynamique représentant la base de la procédure appelante contenue dans DISPLAY(NVC).

Microprogramme:

Y = SP-1, Y → RADM, LMC(CTX)
Y = DISPLAY(NVC), Y → RME_1
Attente(MC), D = RML_0, Y = D, Y → RME_0, EMC(CTX)
D = RML_1, D → Q

.c/ Mise à jour du chaînage statique

Le niveau courant, celui de la procédure appelante, est contenu dans le registre NVC. Il est échangé avec celui de la procédure appelée, défini comme paramètre de l'instruction ENTER (champ S du nom (S,D)).

Microprogramme:

D = NVC, Y = D, Y → RME_1
D = S, Y = D, Y → NVC

Le chaînage statique, égal à la valeur de l'ancienne base du même niveau, contenue dans DISPLAY(NVC), est empilé et la nouvelle base est positionnée.

Y = DISPLAY(NVC), Y → RME_0
Y = SP+1 → SP, Y → RADM, EMC(CTX)
SP → DISPLAY(NVC)

.d/ Mise à jour de la mémoire associative

En cas de récursivité, des variables appartenant à deux procédures distinctes peuvent porter le même NOM: on fait donc une recherche associative sur le champ S = NVC et on marque VIDES toutes les cases pour lesquelles il y a correspondance.

Microprogramme:

Recherche(NVC, $\overline{VIDE}$)
Ecriture ($\overline{VIDE}$)

De plus, les paramètres étant adressés par rapport à SP entre les instructions CALL et ENTER, on doit charger leur niveau pour qu'ils puissent être adressés comme des variables locales de déplacement négatif.

Microprogramme:

Recherche(SP, $\overline{VIDE}$)
Ecriture(NVC)

.e/ Recopie des descripteurs des variables locales

Le registre Q pointe sur le descripteur de l'ensemble des variables dans le bloc de description de la procédure. Le descripteur DEV contient le nombre de variables locales NV et la taille de la zone valeur des variables TZVV.

Microprogramme:

Y = Q, Y → RADM, LMC(CODE/EXT)
Attente(MC), D = RML_1, Y = D → NV, Valide(F=0)
si F=0 alors JMP(FIN), SP → AE
D = RML_0, Y = D → TZVV

Dans le cas où NV=0 (pas de variables locales) on se branche à l'étiquette FIN. Dans le cas contraire, on recopie les descripteurs des variables locales sur la pile de contexte, à partir de l'adresse AE. La base des valeurs des variables est provisoirement positionnée dans SP.

Microprogramme:

SP+NV → SP
BOUCLE : Attente(MC), Y = Q+1 → Q, Y → RADM, LMC(CODE/EXT)
Attente(MC), D = RML_1, Y = D, Y → RME_1
si $\overline{IV}$ alors JMP(+2), D = RML_0, Y = D, Y → RME_0
D = RML_0, Y = D+SP, Y → RME_0
Y = AE, AE+1 AE, Y → RADM, EMC(CTX)
NV-1 → NV, Valide(F=0)
si F ≠ 0 alors (BOUCLE)

.f/ Mise à zéro de la zone valeur

Dans le cas où la taille de la zone valeur n'est pas nulle, on met cette zone à ZERO (valeurs non initialisées).

Microprogramme:

```
Attente(MC Y=TZVV, 0 → RME0
BOUCLE: si F=0 alors JMP(FIN), 0 → RME1
        Y=SP, SP+1 → SP, Y → RADM, EMC(CTX)
        Y=TZVV-1 → TZVV, JMP(BOUCLE)
```

.g/ Mise à jour des registres SP et W

Un chaînage entre les différentes valeurs du registre W est établi au sommet de la pile.

Microprogramme:

```
FIN: Attente(MC), Y=W, Y → RME0
     Y=SP, SP+1 → SP, Y → RADM, EMC(CTX)
     SP → W
```

.h/ Vérification du débordement

La nouvelle valeur du registre SP est comparée avec celle de la limite de la zone CTX.

Microprogramme:

```
LIMITE-SP
si NEGATIF alors JMP(ERREUR-DEBORDEMENT)
```

.i/ Performances de l'instruction ENTER

Elles sont résumées dans le tableau suivant, où N représente le nombre de variables locales et M la taille de la zone valeur des variables.

conditions	microinstructions	lecture	écriture
N=0	20	3	3
N≠0 et M=0	24+7*N	3+N	3+N
N≠0 et M≠0	23+7*N+3*M	3+N	3+N+M

III - L'instruction RETURN

Cette instruction termine l'exécution de toute procédure.

.a/ Restauration du registre W

L'ancienne valeur de ce registre a été empilée à l'adresse pointée par W moins 1.

Microprogramme:

Y=W-1, Y RADM, LMC(CTX)
Attente(MC), D=RML_0, D → W

.b/ Mise à jour de la mémoire associative

A cause de la récursivité, des variables appartenant à la procédure dans laquelle on retourne peuvent porter le même nom que des variables de la procédure que l'on quitte: on les supprime de la mémoire associative.

Microprogramme:

Recherche(NVC, $\overline{VIDE}$)
Ecriture(VIDE)

.c/ Restauration des registres de base

La marque du bloc, pointée par DISPLAY(NVC)-1, contient le niveau de la procédure appelante, les chaînages statique et dynamique et l'ancienne valeur du sommet de pile SP.

Chaînage statique:

Y=DISPLAY(NVC)-1 → SP, Y → RADM, LMC(CTX)
Attente(MC), D=RML_0, D → DISPLAY(NVC)

Restauration du niveau d'appel:

D=RML_1, Y=D, Y → NVC

Chaînage dynamique:

Y=SP-1, Y →RADM, LMC(CTX)
Attente(MC), D=RML_1, D → DISPLAY(NVC)
D=RML_0, D → SP

On se retrouve ainsi dans un état identique à celui d'avant l'instruction CALL.

.d/ Performances de l'instruction RETURN

8 microinstructions et 3 lectures mémoire.

IV - L'appel et le retour des procédures externes

Comme nous l'avons exposé dans les aspects SYSTEME, certaines procédures peuvent appartenir à un module externe. Au cours de l'exécution, le processeur PAC va recevoir des instructions CALL-EXT, ENTER-EXT et RETURN-EXT qui seront prises en considération pour l'adressage des deux zones CODE et EXTERNE, selon la valeur d'une bascule de MODE(interne ou externe).

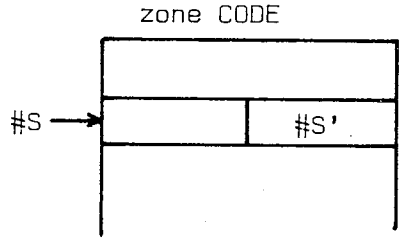
.a/ L'instruction CALL-EXT

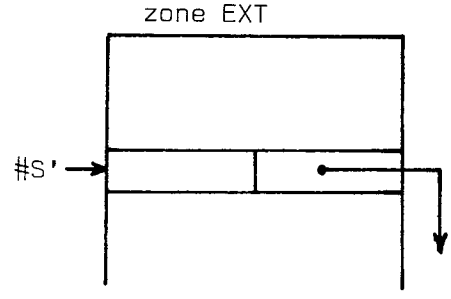
Cette instruction ne peut apparaître que dans le mode INTERNE, c'est-à-dire dans le module principal. Son interprétation est identique à celle de l'instruction CALL, mis à part l'accés au bloc de description, par l'intermédiaire des DEUX tables de segments.

Il faut d'abord utiliser le numéro du segment passé en paramètre de l'instruction CALL-EXT pour accéder à la table des segments du module principal (zone CODE):

D=PARAM, Y=D, Y → RADM, LMC(CODE)
Attente(MC), D=RML_0, Y=D, Y → RADM, LMC(EXT)

puis on obtient un numéro de segment dans la table du module externe. On lit ce segment et on continue comme pour l'instruction CALL.





.b/ L'instruction ENTER-EXT

Cette instruction ne peut apparaître que dans le mode INTERNE, c'est-à-dire dans le module principal. Le processeur PAC sait que l'adresse du bloc de description

des variables est relative à la zone EXTERNE. Il force donc son adressage:

Y=Q, Y → RADM, LMC(EXT)

D'autre part, à la fin de l'interprétation de cette instruction, le processeur PAC se met dans le mode externe:

MODE → EXT.

.c/ L'instruction RETURN-EXT

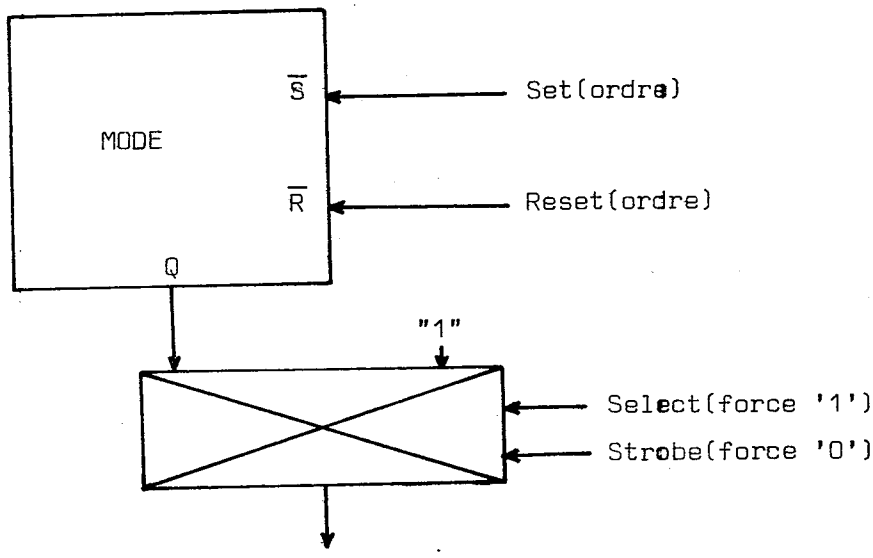
Cette instruction ne peut apparaître que dans le mode EXTERNE, dans lequel seule la zone EXTERNE est adressée. Le processeur PAC revient dans l'état INTERNE: MODE → INT.

.d/ Gestion de la bascule de MODE

La valeur 0 ou 1 de cette bascule permet d'adresser soit la zone CODE soit la zone EXTERNE. On doit pouvoir

- sélectionner cette bascule (select=0)
- forcer un '1' (select=1)
- forcer un '0' (strobe=1)
- mettre à 0 (reset) ou à 1 (set) la bascule.

D'où le schéma:



G - MICROPROGRAMMES DU PROCESSEUR PAC

I.- DECODAGE DES INSTRUCTIONS D'ACCES

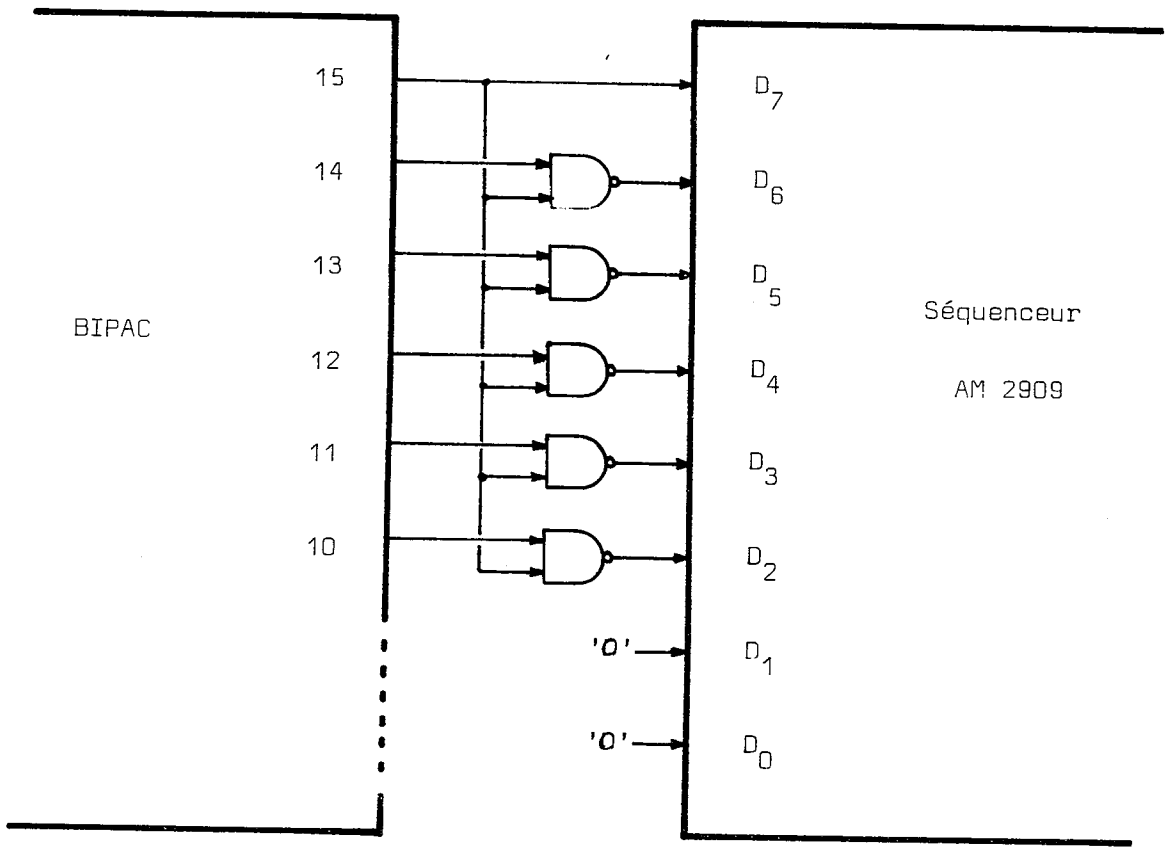
Le codage des instructions reçues par le processeur PAC est donné par le tableau . Ce codage fait apparaître que:

- le bit de poids fort est toujours égal à 1, ce qui permet une vérification ;

- tous les codes-opération sont caractérisés par une chaîne de 5 bits (les bits 10 à 14), ce qui permet d'associer à chaque code une adresse dans le microprogramme, soit directement, soit par l'intermédiaire d'une mémoire ROM (1 circuit ROM 27LS09).

La détection d'erreur peut être réalisée automatiquement en associant au bit de poids fort l'adresse du traitement de l'erreur qui serait 01111100 dans l'exemple suivant:

Exemple de réalisation possible:



Instruction mnémonique	code	
AFFECT s,d	1 0 0 0 0 0 x x	
PARAM s,d	1 0 0 0 0 1 x x	
CALL s,d	1 0 0 0 1 0 x x	
CALLF s,d	1 0 0 0 1 1 x x	
ENTER	1 0 0 1 0 0 0 1	
ENTEREXT	1 0 0 1 0 1 0 1	
RETURNEXT	1 0 0 1 1 0 0 0	
RETURNFEXT	1 0 0 1 1 1 0 0	
RETURN	1 0 1 0 0 1 0 0	1 bit distingue ENTER et RETURN
RETURNF	1 0 1 0 1 0 0 0	
ZERO	1 0 1 0 0 0 x x	
ONE	1 0 1 1 0 0 x x	
LIT(8)	1 1 0 0 0 0 x x	
LIT(16)	1 1 0 0 0 1 x x	
LIT(32)	1 1 0 0 1 0 x x	
NOM s,d	1 0 1 1 0 1 x x	
ID s,d	1 0 1 1 1 0 x x	
MONTE (n)	1 1 0 1 0 0 x x	
DESCEND (n)	1 1 0 1 0 1 x x	

Codage des instructions pour PAC

II.- SEQUENCE D'ACCES A L'INSTRUCTION SUIVANTE (FETCH)

La fin de l'interprétation d'une instruction comporte la commande d'extraction du mot suivant de la file d'attente des instructions BIPAC.

La séquence d'accès à l'instruction suivante doit donc commencer par tester l'état de la file d'attente:

Attente (BIPAC - Non - Vide)

Quelle que soit l'instruction suivante, la probabilité d'avoir un calcul d'adresse à effectuer est importante. On peut donc, dans cette première micro-instruction, sélectionner la ROM de décodage du nom (S,D) et effectuer une recherche associative.

Paramètres pour le calcul d'adresse

U = ROM(O), @A = S^*, Y = RAM(@A)+D→O, Y → RAM

Paramètres pour la mémoire associative

PCAM = Recherche(S,D, $\overline{VIDE}$), mémorisation du vecteur MATCH, et de l'indic. TROUVE

Définition du séquencement

On spécifie l'entrée D comme adresse de branchement.

On exécute ainsi, en une microinstruction, les opérations suivantes:

- décodage du nom (S,D),
- calcul d'adresse (BASE(S^*)+D^*) et rangement dans le registre Adresse mémoire et l'accumulateur,
  recherche associative et mémorisation des indicateurs.

Dans le cas où l'instruction suivante ne nécessite aucun calcul d'adresse, les opérations effectuées ne servent à rien, mais cela ne gêne en rien.

III.- INTERPRETATION DES INSTRUCTIONS D'ACCES

III.1. Instruction NOM(s,d)

Si le nom (s,d) a été "TROUVE" dans la CAM, le processeur FILE est capable de trouver le descripteur de la variable dans la file des dépendances si la dépendance est résolue, ou bien de créer un descripteur de dépendance.

Dans le cas contraire, le descripteur de la variable doit être lu en mémoire centrale, puis transféré au processeur FILE qui le rangera dans la file d'évaluation.

Algorithme:

```
NOM     si TROUVE alors ordre(LMC), JMPR(ATMC)
        DPAC←1, ALLOCAM←1, JMPR(ATFILE)

ATMC    attente(MC), D = RML1, Y = D, Y→PACE1
        D = RML0, Y = D, Y PACE0, DPAC←1, ALLOCAM←1, JMPR(ATFILE)

ATFILE attente(DPAC=0), ALLOCAM←0, NEXOUT, JMPR(FETCH)
```

III.2. Instruction ID(s,d)

Cette instruction est réservée au passage d'un paramètre par référence, et uniquement dans le cas d'une variable simple dont la valeur est contenue dans le descripteur. Le processeur PAC doit construire un descripteur indirect, caractérisé par la valeur 1 du bit DI du préfixe, et dont le champ adresse de la valeur contient l'adresse du descripteur du paramètre effectif (une adresse d'octet).

Algorithme:

```
ID      Ordre(LMC), Y = Q, Shift left(Y)→RAM(B), B = RT
        A = B = RT, Y = RAM(A)+RAM(B), Y→PACE0
        Y = 0, Shift Right(Y)→Q (Q = 100....0)
        Attente (FINLEC), D = RML1, Y = Q ou D, Y→PACE1
                DPAC←1, ALLOCAM←1,TROUVE←0, JMP(ATFILE)
```

III.3. Les instructions littérales

Deux microinstructions sont nécessaires pour construire les deux parties d'un descripteur. La première construit un préfixe immédiat.

Dans le cas de LIT16 et LIT32, un accès à l'élément suivant de la file d'attente est nécessaire (ordre NEXOUT).

III.4. Les instructions d'affectation AFFECT et PARAM

Même si le nom de la variable affectée est trouvé dans la file de dépendances, on doit construire un descripteur d'indirection et fournir au processeur POP l'adresse du descripteur de la variable. Dans le cas d'une instruction PARAM, il peut y avoir une dépendance, mais elle concerne deux variables distinctes correspondant à deux procédures distinctes: il faut donc dans tous les cas accéder au descripteur situé en mémoire centrale.

Algorithmes:

```
AFFECT si TROUVE alors   demande lecture, JMPR(ATMC)
                         Y = 0, shift left(Y)→RT
       Y = RT+RT, Y→RF0,[DPAC←1, ALLOCAM←1, JMPR(ATFILE)]

ATMC   Y = RT+RT, Y→RF0
       attente(FINLEC), D=RML1, Y=D, Y→RF1     [          "          ]

PARAM  demande lecture, Y = 0, shift left(Y)→RT
       Y = RT+RT, Y→RF0
       attente(FINLEC), D = RML1, Y = D, Y→RF1
       Y = SP-1, Y→RADM, LMC
       attente(FINLEC), D = RML0, Y = D-1, Y→RME0, EMC ,[     "       ]
```

CHAPITRE 5

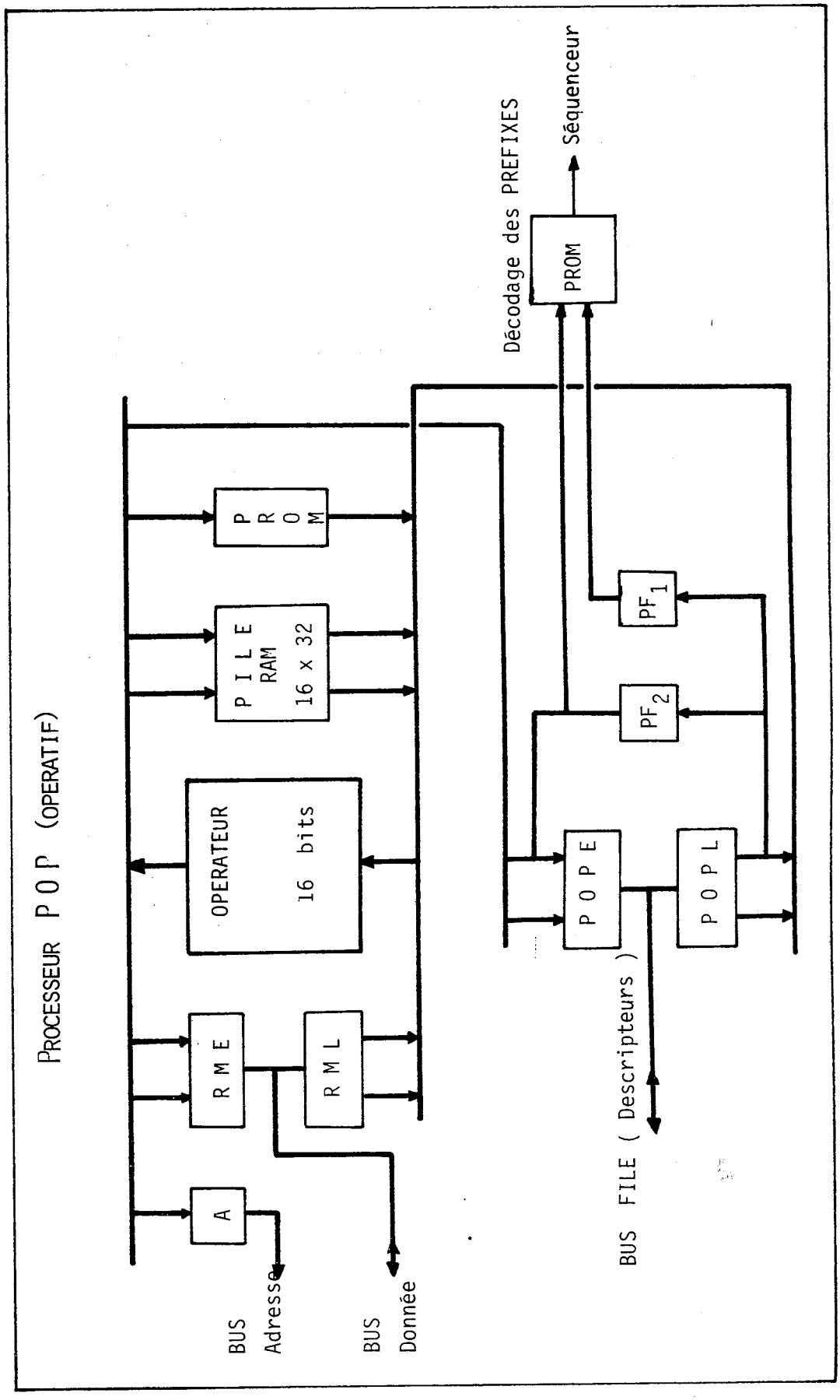
LE PROCESSEUR OPERATIF POP

A - INTRODUCTION

Comme son nom l'indique, le processeur POP est spécialisé dans l'exécution des OPérations arithmétiques ou logiques sur les valeurs des variables, ou des OPérations d'adressage permettant d'accéder aux valeurs des variables (accès à un champ d'une structure, à un élément de tableau, ...).

Il reçoit ses instructions du processeur PINS, par l'intermédiaire de la file d'attente BIPOP, et les opérandes lui sont fournis par le processeur FILE qui les extrait de la file d'évaluation FILEVAL.

Il joue d'autre part le rôle de contrôleur du déroulement algorithmique du programme, en ce sens qu'il connait le résultat des prédicats décidant du chemin à prendre dans l'algorithme, et peut ainsi indiquer que l'anticipation faite par le processeur PINS est justifiée ou qu'elle ne l'est pas (voir chapitre III).


PROCESSEUR POP (OPERATIF)
Décodage des PREFIXES
Séquenceur
PROM
PROM
PILE RAM 16 x 32
OPERATEUR 16 bits
RME
RML
A
POPE
POPL
PF1
PF2
BUS Adresse
BUS Donnée
BUS FILE (Descripteurs)

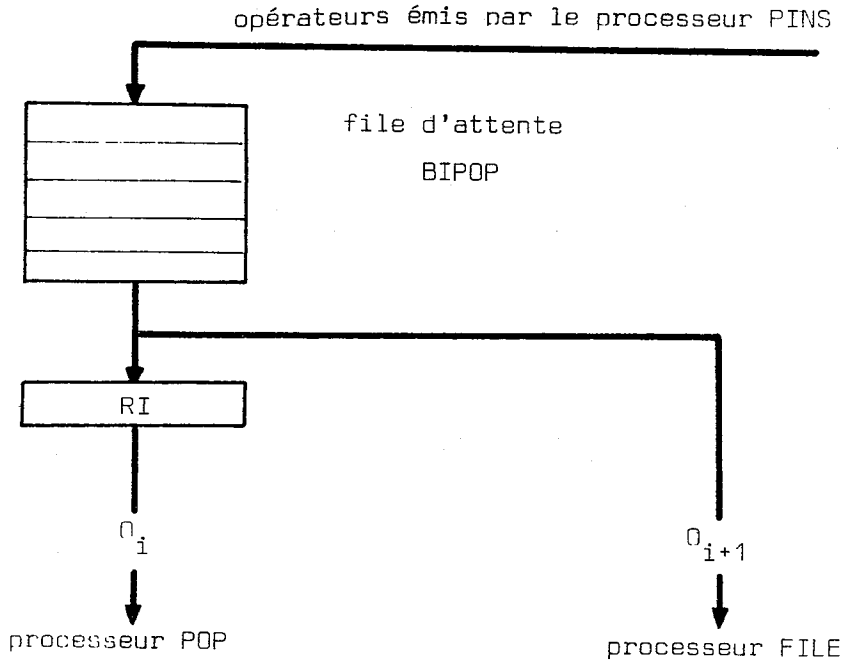
Nous étudierons d'abord la synchronisation entre POP et FILE, puis les problèmes de l'anticipation (opérations de contrôle) et présenterons enfin quelques algorithmes d'exécution des opérateurs les plus significatifs.

Afin de faciliter la compréhension de ce chapitre, le schéma de l'architecture interne du processeur POP est présenté ci-après, qui donne une idée des différentes variables de travail qui pourront être manipulées par lui.

B - SYNCHRONISATION ENTRE POP ET FILE

Le processeur POP demande les services du processeur FILE lorsqu'il a besoin d'un opérande associé à un opérateur. La recherche de la performance nous a incités à définir un niveau supplémentaire de "pipeline" entre ces deux processeurs: pendant que POP exécute l'opérateur O_i, FILE peut préparer l'opérande relatif à l'opérateur O_{i+1}.

La réalisation d'un tel mécanisme se résume au fait que le processeur FILE doit avoir une instruction d'avance sur le processeur POP, ce qui se traduit par le schéma suivant:



La file d'attente BIPOP est remplie par PINS, qui teste l'indicateur PLEIN.

Le processeur FILE teste l'indicateur VIDE lorsque POP lui fait une requête.

Le processeur POP commande l'extraction d'un nouvel opérateur (O_{i+1}) lorsqu'il a été préparé par FILE (demande acquittée).

Une seule bascule R.S, appelée DPOP, suffit pour réaliser la synchronisation entre les deux processeurs, la demande étant positionnée par POP, l'acquittement par FILE.

Lorsque POP a récupéré dans un tampon POPL, l'opérande relatif à l'opérateur suivant, il commande l'extraction de l'instruction suivante, puis demande de nouveau les services du processeur FILE.

A - L'ordre INIT(n)

Il correspond au début de l'évaluation d'une expression post-fixée. C'est un ordre d'avancement des pointeurs, distinct de l'ordre AVANCE(n) par le fait qu'il n'y a pas de sauvegarde d'un résultat intermédiaire (voir chapitre II, 2ème partie). Si l'avancement est possible, le processeur FILE lit l'opérande en FILEVAL(P_2), le place dans le tampon POPL et il acquitte la demande.

Lorsque POP a terminé l'exécution de l'instruction précédente, il teste DPOP. Si DPOP=0, il sait que l'instruction suivante a été préparée par FILE: il accède donc à cette instruction INIT, récupère l'opérande contenu dans POPL et émet une nouvelle demande de service au processeur FILE.

B - L'odre AVANCE(n)

Cet ordre est identique à l'ordre INIT pour la récupération du nouvel opérande FILEVAL(P_2+n), mais le dernier résultat intermédiaire doit être sauvegardé dans FILEVAL(P_2). Ceci nécessite donc deux phases de dialogue entre POP et FILE.

- la première phase est identique à l'exécution de INIT,
- la deuxième phase est caractérisée par une deuxième demande au processeur FILE, pour lui indiquer que l'opérande rangé par POP dans le tampon POPE peut être sauvegardé dans FILEVAL(P_2).

Les détails de ces échanges sont donnés dans le chapitre VI.

C - Opérateur unaire

Le processeur FILE n'a aucun travail à exécuter pour un opérateur unaire: il acquitte immédiatement la demande de POP, qui dispose de l'opérande dans un registre local R_2 et exécute cet opérateur.

D - Opérateur binaire

Le processeur POP attend l'acquittement du processeur FILE qui lui fournit le premier opérande dans le tampon POPL, le deuxième opérande étant dans le registre local R_2.

E - Opérations répétitives

Les ordres SAUVE(n), RESTAURE(n) et OF(n) générés respectivement pour la sauvegarde de la file d'évaluation ou sa restauration en cas d'appel de fonction, ou pour la recherche d'un CAS dans une structure "CASE", supposent un dialogue plus complexe entre POP et FILE: ce dernier se charge de contrôler la fin du dialogue (en décrémentant le paramètre n) et indique la fin du dialogue par le positionnement d'une autre bascule testée par POP (voir § VI).

C - EXECUTION DES OPERATIONS DE CONTROLE

Le processeur POP évalue les expressions booléennes dont les valeurs déterminent le chemin à prendre dans le programme.

Il connait, pour chaque opération de contrôle, le choix qu'a réalisé le processeur PINS et peut ainsi l'avertir qu'il s'est trompé, en émettant une interruption par l'ordre TROMPE, ou bien le faire sortir de l'état conditionnel (ETC←0) si le choix était le bon.

Examinons les principaux opérateurs:

A - Les opérateurs THEN et WHILE

Rappel de la syntaxe:

PASCAL : IF <EXP> THEN <INST>

PASC-HLL : <EXP> THEN(m) <INST> FI

Si la valeur de <EXP> est fausse, alors PINS s'est trompé dans son choix.
De même pour la structure WHILE:

PASCAL : WHILE <EXP> DO <INST>

PASC-HLL : LOOP(m) <EXP> WHILE <INST> ENDLOOP

Microprogramme

si FAUX alors ordre(TROMPE), JMP(FETCH)
ordre(ETC←0), JMP(FETCH)

B - Les opérateurs UNTIL et EXITIF

PASCAL : REPEAT <INST> UNTIL <EXP>

PASC-HLL : LOOP(m) <INST><EXP> UNTIL

et

PASCAL : LOOP <INST1> EXITIF <EXP><INST2> END

PASC-HLL : LOOP(m) <INST1><EXP> EXITIF <INST2> ENDLOOP

Dans ce cas, le processeur PINS s'est trompé si la valeur de <EXP> est VRAIE.

D'où le microprogramme:

si VRAI alors ordre(TROMPE), JMP(FETCH)
ordre(ETC←0), JMP(FETCH)

C - Les opérateurs FORUP(FORDOWN), ROFUP(ROFDOWN)

PASCAL : FOR I := <EXP1> TO <EXP2> DO <INST>

PASC-HLL : <EXP1><EXP2> FORUP(m) AFFECT(I)
<INST> ROFUP

L'opérateur FORUP doit d'abord tester si <EXP1> est inférieure ou égale à <EXP2>. Si oui, le choix de PINS est bon (il a choisi d'entrer dans la boucle). Le processeur POP empile alors un descripteur de contrôle de boucle contenant la valeur courante et la valeur limite de l'indice.

Microprogramme

```
      ≀  Accès à <EXP1> → R1
    R2 - R1
    si POSITIF alors ordre(ETC←0), JMP(BON)
    ordre(TROMPE), JMP(FETCH)
BON JSR(PUSH)
    R2 → PILE1
    R1 → PILE0, JMP(FETCH)
```

De même pour l'opérateur ROFUP, on compare la valeur courante incrémentée de 1 à la valeur limite. Si la comparaison est favorable, le processeur PINS a fait le bon choix.

Microprogramme

```
      ≀  Accès au sommet de pile
    si VIDE alors JSR(ACCES)
    PILE0 + 1 → R2
    PILE1 - R2
    si POSITIF alors ordre(ETC←0), JMP(BON)
    ordre(TROMPE), JSR(POP) ← dépile
    JMP(FETCH)
BON R2 → PILE0, JMP(FETCH)
```

D - PREPARATION DES OPERATIONS ARITHMETIQUES

La machine PASC-HLL est une machine à préfixe (TAGGED Architecture [FEUSTEL]). Chaque variable est décrite par un "descripteur" directement créé par le compilateur ou dynamiquement construit par les opérateurs d'indexation ou d'accès à un champ.

Chaque descripteur donne le TYPE DE BASE de la valeur de la variable, qui donne la sémantique de l'opération à effectuer, par exemple une addition entre un entier et un réel, ou une affectation d'une chaîne de caractères à un entier (d'où conversions internes).

Le descripteur donne également des indications relatives à la localisation physique de la valeur et à sa taille physique.

Une valeur peut être:

- dans le descripteur,
- en mémoire dans la zone CONTEXTE,
- en mémoire dans la table des CONSTANTES,
- au sommet de la pile interne à POP.

L'accès à la veleur d'une variable comporte également la vérification du bit d'initialisation associé. La position physique de ce bit est indiquée par le préfixe du descripteur.

Etudions, à titre d'exemple, l'accès à la valeur d'un entier long de 32 bits:

1/ On envoie l'adresse de la valeur à la mémoire centrale,

2/ On récupère (dans RML_0) le demi-mot de poids faible qui contient un demi mot complet non signé,

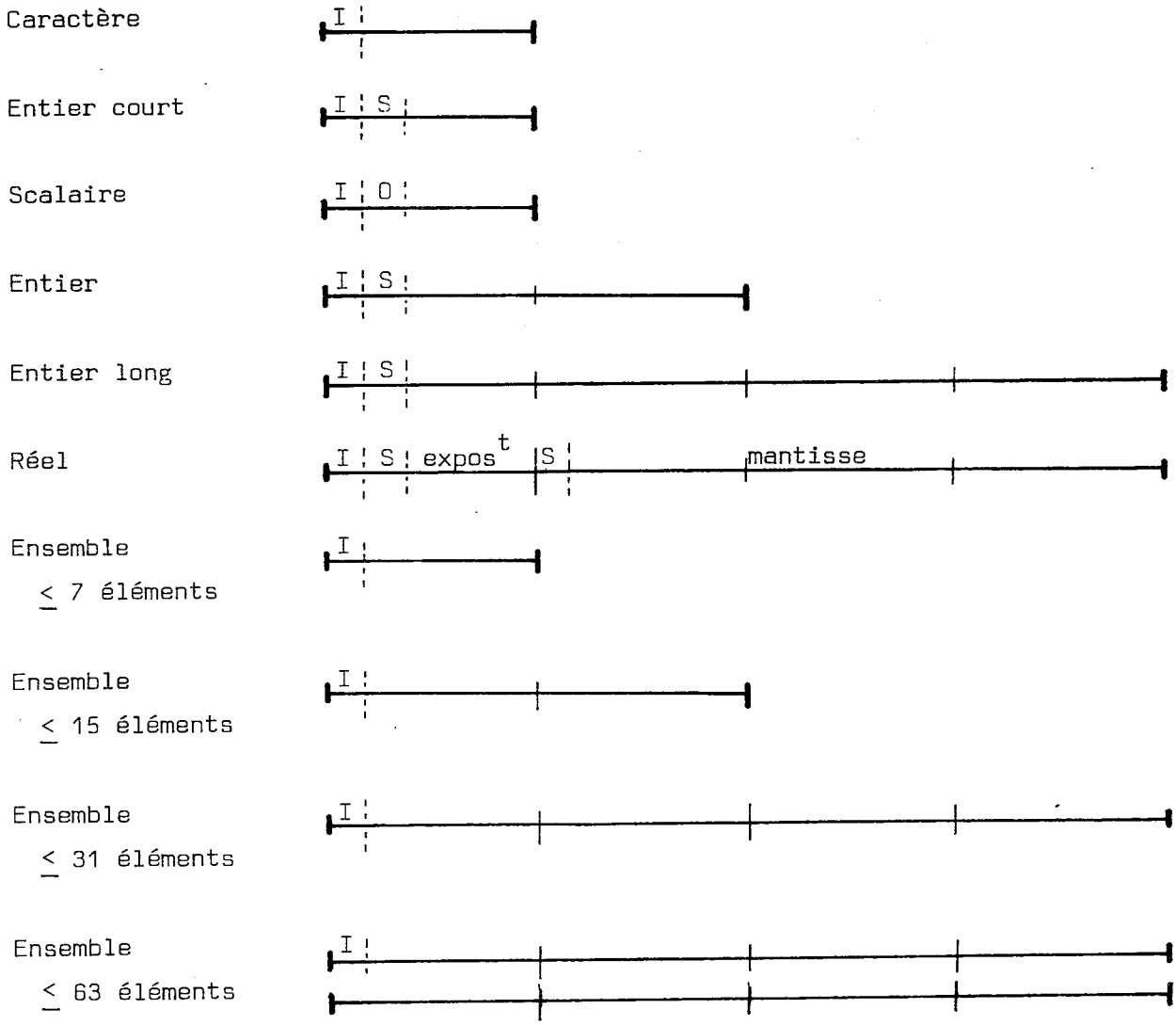
3/ On récupère (dans RML_1) le demi-mot de poids fort qui contient le bit d'initialisation I et le bit de signe S. On teste "au passage" la valeur du bit I.

Règle de formattage:

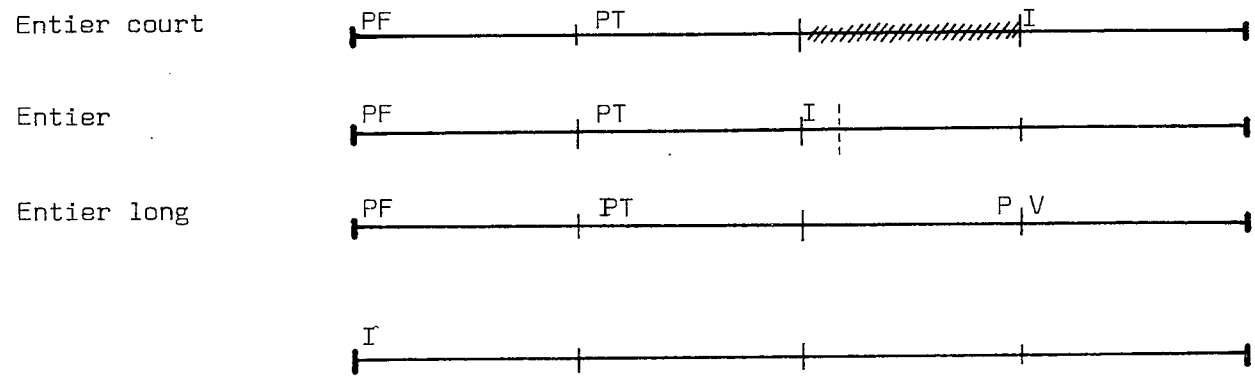
Le bit d'initialisation est toujours placé en position de plus fort poids, donc à gauche du bit de signe dans le cas d'un entier signé. Ce bit est actif à '1', ainsi l'initialisation à ZERO d'une zone de la mémoire assure que toutes les valeurs sont dans l'état "non initialisé" (I=0).

Le format des valeurs en mémoire centrale ou dans les descripteurs est donné par la figure suivante:

Formatage des données:



Descripteurs de variables



En plus de la vérification du bit d'initialisation, on doit pouvoir construire un demi-mot (16 bits) complet pour l'opérateur du processeur POP.

Selon le type de la valeur accédée, il faut réaliser les 6 cas suivants:

- prendre l'octet gauche ou droit complet et l'étendre par des '0' ;
- prendre l'octet gauche ou droit signé et l'étendre par des '0' ;
- prendre l'octet gauche ou droit signé et l'étendre par son signe ;
- prendre le demi-mot complet ;
- forcer un '0' en poids fort du demi-mot ;
- étendre le signe du demi-mot.

Ces six fonctions sont réalisées par un système de multiplexeurs qui sera présenté dans la deuxième thèse [8].

E - EXECUTION DES OPERATEURS

Nous ne présenterons ici que les opérateurs les plus significatifs: l'indexation, l'accès à un champ, l'allocation dynamique et les opérateurs arithmétiques compliqués comme la multiplication et la division. Il serait en effet long et fastidieux de décrire tous les opérateurs de comparaison par exemple dont la réalisation ne pose d'ailleurs pas de problèmes intéressants.

I - L'opérateur INDEX

Cet opérateur binaire a comme premier opérande un descripteur de variable de type "tableau" et comme deuxième opérande une valeur scalaire qui représente la valeur de l'index.

Le champ PI du descripteur du premier opérande donne accès au descripteur du "type-tableau", dans lequel on trouve:

- les bornes inférieure (MIN) et supérieure (MAX) de l'index,
- la taille d'un élément (PAS),
- les champs PF et PT de l'élément qui permettront de construire le descripteur de l'élement ainsi accédé.

Microprogramme

1/ Accès au premier mot du descripteur de type

$Y = PT_1$ + BASETYPE → AD, Y→RADM, LMC(CODE)

2/ Vérification que MIN ≤ INDEX ≤ MAX

Attente(MC), RML^{D}_{0}(MAX) - R_2
Si POSITIF alors JMP(TESTMIN),= RML^{G}_{0}(MIN) + R_2
JMP(ERREUR.MAX)

TESTMIN: Si ZERO alors JMP(ZERO), = RML^{G}_{0}(MIN) + R_2
Si POSITIF alors JMP(CALCUL), 1→RT
JMP(ERREUR.MIN)

3/ On accède au deuxième mot du descripteur de type et on fait le calcul (INDEX.MIN)*PAS, après avoir testé si le PAS vaut 1, 2, 3 ou 4 octets pour tenir compte de la fréquence de ces tailles.

CALCUL Y = AD+1, Y→RADM, LMC(CODE)
RT+1 → RT
Attente(MC), D=RML_0(PAS)
si ZERO alors JMP(PAS1), D-1→Q
si ZERO alors JMP(PAS2), RT=Q
si POSITIF alors JMP(PAS34), RT=Q

4/ Dans le cas où le pas est supérieur à 4 octets, on effectue la multiplication classique:

D = RML_0(PAS), D+1 → Q
0 → RT
(16 fois) (RT+R_2)/2 → RT ou (RT+0)/2 → RT
ADD: Q+R_1 → R_2 , JMP(FIN)

5/ Pour les cas particuliers, on optimise la multiplication:

PAS1 $R_1 + R_2 \rightarrow R_2$, JMP(FIN)

PAS2 $R_2 + R_2 \rightarrow Q$, JMP(ADD)

PAS34 si ZERO alors JMP(ADD), $(R_2+R_2)*2 \rightarrow Q$

$Q - R_2 \rightarrow Q$, JMP(ADD)

6/ Finalement on construit le descripteur de l'élément accédé:

ZERO $R_1 \rightarrow R_2$

FIN $D = RML^{D}_{1}(PT)$, $D \rightarrow PT_2$ et $RML^{G}_{1}(PF) \rightarrow PF_2$

JMP(FETCH)

Performances de l'opération INDEX

Le nombre d'accès à la mémoire est toujours égal à 2, pour lire les 2 mots constituant le descripteur du tableau. Le nombre de microinstructions est donné par le tableau:

Conditions	nombre MICRO
INDEX = MIN	6
PAS = 1	11
" 2	13
" 3	15
" 4	14
> 4	16 + 15

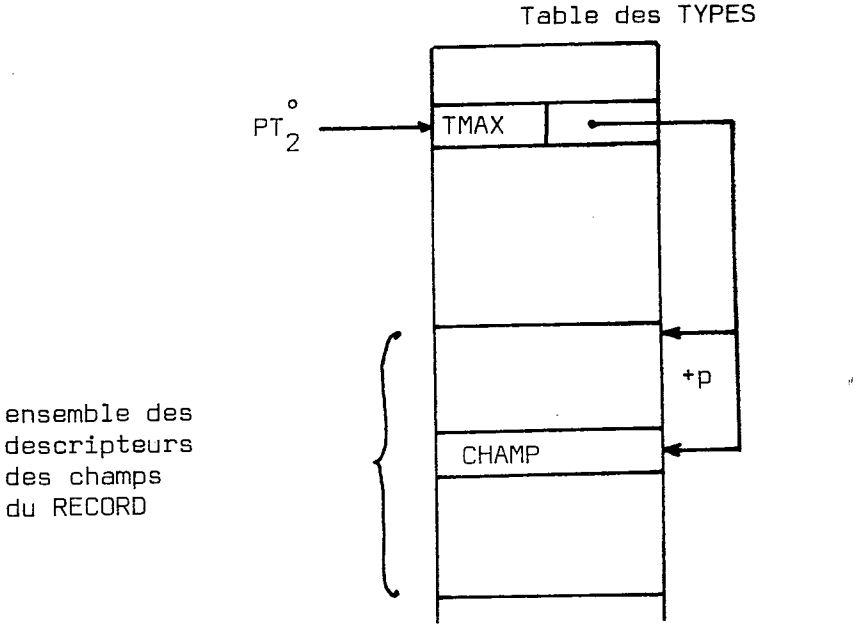
Dans une machine classique (IRIS 80 ou IBM 360) l'indexation est programmée en langage-machine: elle nécessite environ une dizaine d'instructions. Le gain en nombre d'accès à la mémoire, et en même temps d'exécution, est donc important.

Machine classique	PASC-HLL
10 instructions de 32 bits	1 instruction de 8 bits
$\simeq$ 20 µs	entre 1 µs et 5 µs
$\simeq$ 40 octets	9 octets
rapport temps:	entre 4 et 20
rapport encombrement:	$\sim$ 4

II - Accès à un CHAMP dans un RECORD - Opérateur CHAMP(p)

L'opérateur CHAMP(p) est un opérateur unaire qui a un paramètre p, indiquant le déplacement du descripteur formel du champ dans la description de la structure RECORD.

La tête du RECORD est donnée par le champ PT du descripteur de la variable dont on veut accéder à un champ (PT°_2 = BASETYPE+PT).

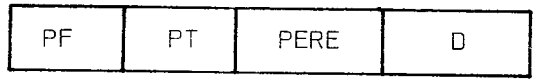


La première opération consiste à lire en mémoire, dans la table des types, la TETE du type RECORD qui donne l'adresse du premier champ. Le paramètre p, issu de la file d'attente d'instructions BIPOP, est ensuite ajouté à cette adresse pour permettre la lecture du descripteur du champ.

Microprogramme:

1: $Y=PT^{\circ}_2$, Y → RADM, lecture(TYPE), NEXTOUT(BIPOP)
2: Attente(BIPOP), D=BIPOP, D → RT
3: Attente(MC), $D=RML_0$, Y=RT+D, Y → RADM, lecture(TYPE)
4: si CONTROLE-CHAMP alors JMP(CONTROLE)

Le format d'un descripteur de champ est le suivant:



- PF et PT donnent respectivement le préfixe et l'accés au type du champ,
- PERE donne l'adresse, dans la table des types, de la branche à laquelle appartient le champ,
- D donne le déplacement de la valeur du champ dans la zone allouée à la valeur du RECORD.

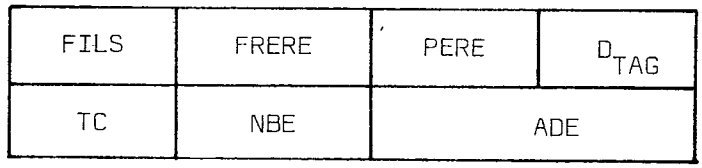
Dans le cas où l'option "contrôle dynamique de l'existence logique d'un champ" n'est pas en fonction, on met à jour les champs PF, PT et PV du descripteur du champ et l'exécution de l'instruction est terminée:

5: Attente(MC, $D=RML_0^D$, $D+R_2^0 \rightarrow R_2^0$

6: $D=RML_1^D$, $D+BASETYP \rightarrow PT_2^0$, $RML_1^G \rightarrow PF_2$ et JMP(FETCH)

Dans le cas contraire (option en fonction), il faut préparer le contrôle de l'existence du champ en remontant toutes les branches de l'arborescence.

Le format d'un descripteur de branche est le suivant:



Les chaînages FILS et FRERE, ainsi que le champ TC (Taille Corrigée) sont utilisées dans l'exécution des instructions d'allocation dynamique (voir

Dans le cas du contrôle, il s'agit de vérifier que l'aiguillage (TAG) dont la valeur a pour déplacement D_{TAG} dans le RECORD, a bien une valeur qui appartient à la liste des étiquettes dont le nombre est donné par NBE et l'adresse par ADE (dans la table des constantes).

CONTROLE: R_2^0 R_2^1 = sauvegarde de l'adresse de RECORD

Attente(MC), $D=RML_0^D$, $D+R_2^0 \rightarrow R_2^0$

$D=RML_1^D$, $D+BASETYP \rightarrow PT_2^0$, $RML_1^G \rightarrow PF_2$

$D=RML_0^G$, $D \rightarrow PERE$

TESTBR: si ZERO alors JMP(FETCH)

Dans le cas où le champ accédé appartient à la tête du RECORD, son PERE est égal à ZERO et il n'y a aucun contrôle à faire.

Dans le cas contraire,on lit en mémoire le descripteur de branche:

Y=PERE+BASETYP AD, Y → RADM, lecture(TYPE)

Attente(MC), $D=RML_0^G$, D → PERE (PERE de la branche)

$D=RML_0^D$, $Y=R_2^1+D$, Y → RADM, lecture(CTX ou DYN)

La valeur de l'aiguillage est lue en mémoire et mise dans VTAG afin d'être comparée à celle des étiquettes.

Attente(MC, D=RML(octet), D → VTAG

Y=AD+1, Y → RADM, lecture(TYPE

On lit le deuxième mot du descripteur de branche qui donne l'adresse et le nombre des étiquettes:

Attente(MC), $D=RML_0$, Y=D+BASECONST → AD

Y → RADM, lecture(CONST)

$D=RML_1^D$, D → NBE

La valeur VTAG est comparée successivement avec les 4 étiquettes contenues dans un mot de 32 bits:

TEST: attente(MC), $D=RML_1^G$, D-VTAG

si ZERO alors JMP(EGAL), $D=RML_1^D$, D-VTAG

si ZERO alors JMP(EGAL), $D=RML_0^G$, D-VTAG

si ZERO alors JMP(EGAL), $D=RML_0^D$, D-VTAG

si ZERO alors JMP(EGAL), NBE-1 NBE

si ZERO alors JMP(ERREUR

Y=AD+1 → AD, Y → RADM, lecture(CONST), JMP(TEST)

En cas d'égalité, on teste si la branche a un PERE et on recommence la vérification s'il existe:

EGAL : PERE, JMP(TESTBR)

PERFORMANCES DE L'OPERATEUR CHAMP

Dans le cas où l'option "contrôle de champs" n'est pas en fonction, il faut faire deux lectures en mémoire et exécuter 6 microinstructions.

Dans le cas contraire,

- le contrôle d'un champ appartenant à la tête du record demande deux lectures en mémoire et 8 microinstructions ;

- celui d'un champ appartenant à une branche demande au moins 6 lectures mémoire et au moins 19 microinstructions, sachant que pour chaque couche de l'arborescence il faut compter au moins 4 lectures mémoire et 11 microinstructions supplémentaires.

conditions	lectures	microinstructions
pas de contrôle	2	6
contrôle tête	2	8
contrôle couche 1	$\geq$6	$\geq$19
contrôle couche 2	$\geq$10	$\geq$30
contrôle couche 3	$\geq$14	$\geq$41
⋮		

Remarque:

Les valeurs minima du tableau ci-dessus sont atteintes lorsque toutes les branches n'ont qu'une seule étiquette, ce qui est le cas le plus fréquent.

III -LES OPERATEURS D'ALLOCATION DYNAMIQUE

Le processeur POP dispose de deux registres LIBRE et MAXDYN qui pointent respectivement sur le premier élément libre et la limite de la zone DYNAMIQUE.

- La procédure d'allocation

NEW(<pointeur>, <tag 1>, ..., <tagn>)

est compilée en:

```
<pointeur>
NEW
<tag 1>
CORREC
  :
<tag n>
CORREC
FINCORREC
```

Les deux opérateurs NEW et CORREC sont présentés ci-après. L'opérateur FINCORREC définit simplement une "fin d'expression".

- La procédure de libération

DISPOSE(<pointeur>)

est compilée en:

```
<pointeur>
DISPOSE
```

L'opérateur DISPOSE, qui est également une "fin d'expression" est présenté ci-après.

1.- L'Opérateur NEW

Cet opérateur unaire a pour opérande un descripteur de variable de type POINTEUR. Le type des éléments pointés est donné par le préfixe. S'il s'agit d'une variable structurée (ARRAY ou RECORD), le descripteur du type est lu en mémoire qui donne la taille maximum d'un élément de la structure.

```
        si TYPE-STRUC alors Lecture(TYPE), Y = PT^0_2, Y→RADM,
                                                JMP(STRUC)
        JMP(TAILLE(LONG)), 1→TMAX
TAILLE(00)      JMP(SUITE)                              TMAX = 1 octet
TAILLE(01)      TMAX+TMAX→TMAX, JMP(SUITE)              TMAX = 2 octets
TAILLE(10)      (TAMX+TMAX)*2→TMAX, JMP(SUITE)          TMAX = 4 octets
TAILLE(11)      TMAX+TMAX→TMAX
                (TMAX+TMAX)*2→TMAX, JMP(SUITE)          TMAX = 8 octets

STRUC           Attente(MC), D = RML_1, D→TMAX, JMP(SUITE)
```

La taille maximum TMAX ainsi obtenue est ajoutée à la valeur du pointeur LIBRE sur la zone DYNAMIQUE. S'il y a débordement (LIBRE+TMAX _ MAXDYN) alors la valeur 'NIL' est affectée au pointeur, sinon la valeur de LIBRE lui est affectée et LIBRE est incrémenté de TMAX.

```
SUITE           LIBRE + TMAX→TMAX
                TMAX-MAXDYN
                si NEGATIF alors JMP(AFFECT), 0→R_1
                LIBRE→R_1
                TMAX→LIBRE
AFFECT
```

On réalise ensuite l'affectation de l'adresse de l'élément alloué, contenue dans R_1, à l'opérande décrit par PF_2, PT^0_2 et R^0_2.

S'il s'agit d'un descripteur d'indirection (ID=1), on fait:

```
Y = R^0_2, Y→RADM
Y = PT^0_2-BASETYP, Y→RME^D_1 et PF_2→RME^G_1
Y = R_1, Y→RME_0, lecture(CTX)
```

S'il ne s'agit pas d'un descripteur d'indirection, donc dans le cas d'un élément d'une structure, on fait:

$Y = R^{0}_{2}$, Y→RADM, (Y_1, Y_0)→AOCTET

$Y = R_1$, Y→RME_0 ou RME_1, écriture(CTX ou DYN)

Sachant qu'il peut y avoir une correction de taille à réaliser, on fait pointer le champ PT^{0}_{2} sur la première branche du RECORD en l'incrémentant:

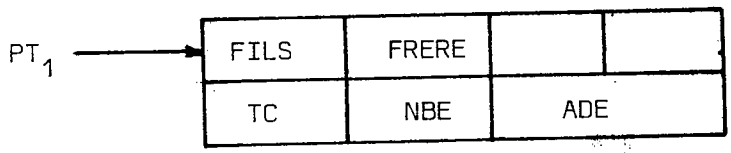
$PT^{0}_{2}+1 \rightarrow PT^{0}_{2}$, JMP(FETCH)

Ainsi l'opérateur CORREC qui suivra éventuellement aura accés à un descripteur de branche.

2.- L'Opérateur CORREC

C'est un opérateur binaire dont le premier opérande est un descripteur dont le champ PT pointe sur une branche de RECORD et le deuxième opérande une expression de type scalaire.

L'opérateur CORREC va parcourir toutes les branches soeurs (pointeur FRERE) jusqu'à trouver une étiquette égale à la valeur de l'expression.



Accés au descripteur de branche après avoir testé son existence:

```
        Y = PT1
        si ZERO alors JMP(ERREUR1)
DEBUT   Y = PT1+1, Y→RADM, lecture(TYPE)
        attente(MC), D = RML^G_1, D→TC
        D = RML^D_1, D→NBE
        D = RML_0, Y = D+BASECONST→AD, Y→RADM, lecture(CONST)
TEST    attente(MC), D=RML^G_1, D-R^0_2
        si ZERO alors JMP(EGAL), D=RML^D_1, D-R^0_2
        si ZERO alors JMP(EGAL), D=RML^G_0, D-R^0_2
```

si ZERO alors JMP(EGAL), D = RML^{D}_{0}, D-R^{O}_{2}

si ZERO alors JMP(EGAL), NBE 1→NBE

si ZERO alors JMP(FRERE)

Y = AD+1 AD, Y→RADM, Lecture(CONST), JMP(TEST)

Dans le cas où les comparaisons ne donnent pas "égalité" pour une branche, on suit le chaînage FRERE, après avoir lu le premier mot du descripteur de branche:

FRERE Y = PT, Y→RADM, Lecture(TYPE)

attente(MC), D = RML^{D}_{1}

si ZERO alors JMP(ERREUR2)

D = RML^{D}_{1}, Y = D+BASETYP→PI_{1}, JMP(DEBUT)

Dans le cas où on a trouvé "égalité" de l'expression avec une étiquette, on peut corriger la taille de l'élélement alloué par le facteur correctif TC.

EGAL LIBRE-TC→LIBRE

Puis on va lire le descripteur de la branche (premier mot) pour avoir l'adresse du FILS de cette branche, afin de préparer l'exécution de l'opérateur CORREC qui suivra éventuellement. Ce résultat est mis comme deuxième opérande dans PT^{O}_{2}.

Y = P1, Y→RADM, lecture(TYPE)

Attente(MC), D=RML^{G}_{1}, D→PT^{O}_{2}, JMP(FETCH)

Remarque:

L'erreur n°1 correspond au fait que l'on veut accéder à une branche qui n'existe pas: on a spécifié trop d'aiguillages.

L'erreur n°2 correspond au fait que l'on a spécifié un aiguillage dont la valeur ne correspond à aucune des étiquettes d'une couche donnée.

IV - REALISATION DE LA MULTIPLICATION

Le processeur opératif POP reçoit l'opérateur binaire MULT qui lui indique que les deux opérandes doivent être multipliés l'un par l'autre. Il connaît, grâce aux deux préfixes, les longueurs respectives des deux opérandes ; les deux champs L_1 et L_2 des préfixes indiquent la longueur des chaînes de bits qui représentent les deux valeurs, mais la longueur significative peut être plus petite.

exemple:

L1 = 10 indique que la valeur V_1 est contenue dans une chaîne de 32 bits. Si les 16 bits de poids fort sont égaux au bit de signe des 16 bits de poids faible, alors la longueur significative de V_1 est seulement 16 bits, d'où $L_1 \leftarrow 01$.

Il est important pour le processeur POP de connaître la longueur significative parce qu'il ne dispose que d'un opérateur de 16 bits, et que, par conséquent, il est plus de deux fois plus lent pour traiter des opérandes de 32 bits.

Conséquence:

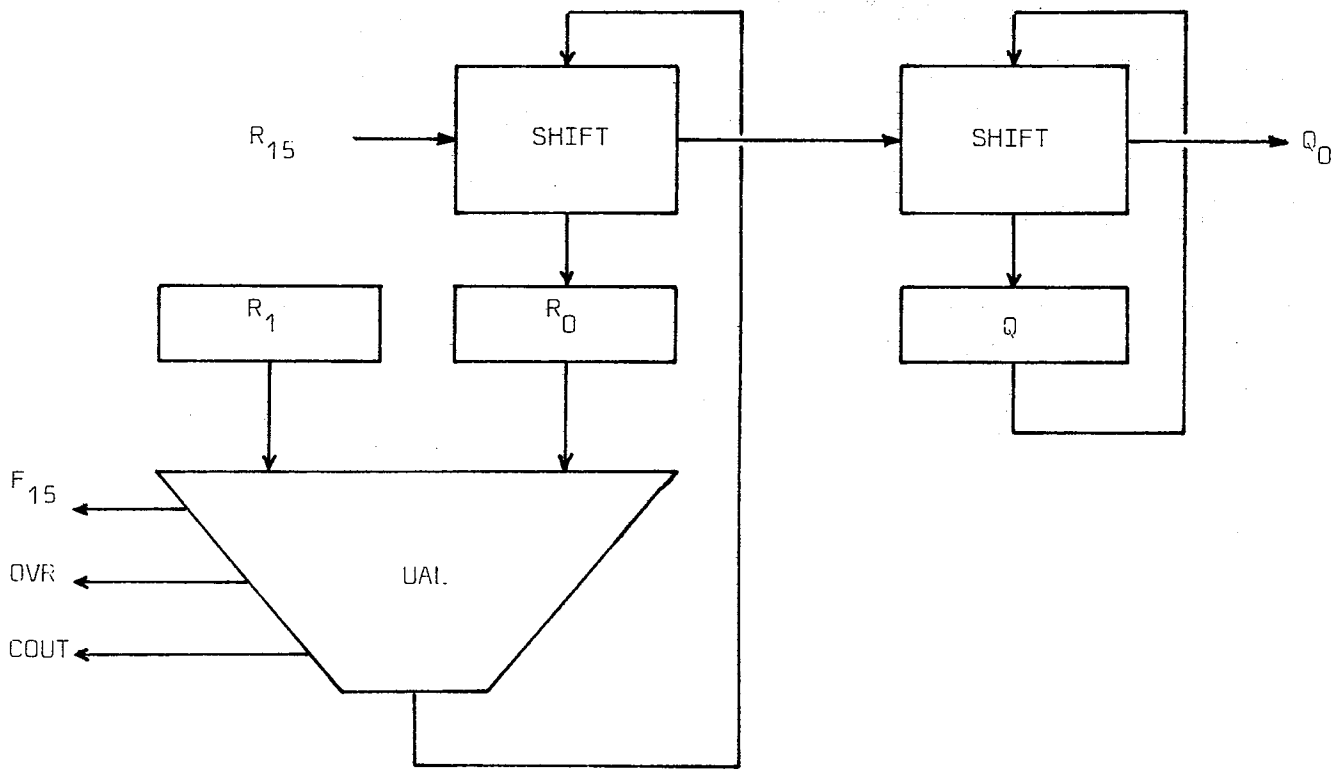
Pour les opérations complexes telles que la multiplication (ou la division), le processeur POP cherchera à opérer uniquement sur les longueurs significatives. De plus, tous les résultats d'opérations seront affublés de la longueur significative (exemple: une multiplication de 16 bits par 16 bits pourra donner un résultat sur 16 bits).

Dans ce qui va suivre, on suppose que les longueurs référencées sont des longueurs significatives.

I. Multiplication de base 16*16

La multiplication de base du processeur POP est celle qui opère sur deux opérandes de 16 bits. Elle est réalisée par une séquence d'addition et de décalage vers la droite:

- le multiplicande est placé dans un registre R_1,
- le multiplicateur est chargé dans l'accumulateur Q,
- le résultat partiel est initialisé à ZERO dans R_0.



L'opérateur AM 2901, représenté partiellement dans la figure précédente, est capable d'exécuter "en un cycle" une addition R1+R0 et le rangement du résultat décalé de 1 position vers la droite, en même temps que le décalage de l'accumulateur Q.

La réalisation de la multiplication présente plusieurs variantes correspondant à la représentation des opérandes: en effet, le multiplicande peut être signé (représentation en complément à 2) ou non signé, ainsi que le multiplicateur. On a donc 4 combinaisons possibles.

I.1. Les deux opérandes sont signés

Le multiplicande étant signé, le bit de poids fort du résultat partiel (RAM_{15}) est égal au signe du résultat R_0+R_1, ou à son complément en cas de débordement.

Donc $RAM_{15} = F_{15} \oplus OVR$.

Le multiplicateur étant lui aussi signé, la 16ième opération doit être une soustraction R_0-R_1 si le multiplicateur est négatif.

15 fois:	(R0+R1)/2 → R0	ou	(R0+0)/2 → R0
	si Q_0=1		si Q_0=0
1 fois:	(R0-R1)/2 → R0	ou	(R0-0)/2 → R0
	si Q_0=1		si Q_0=0

I.2. Le multiplicateur n'est pas signé

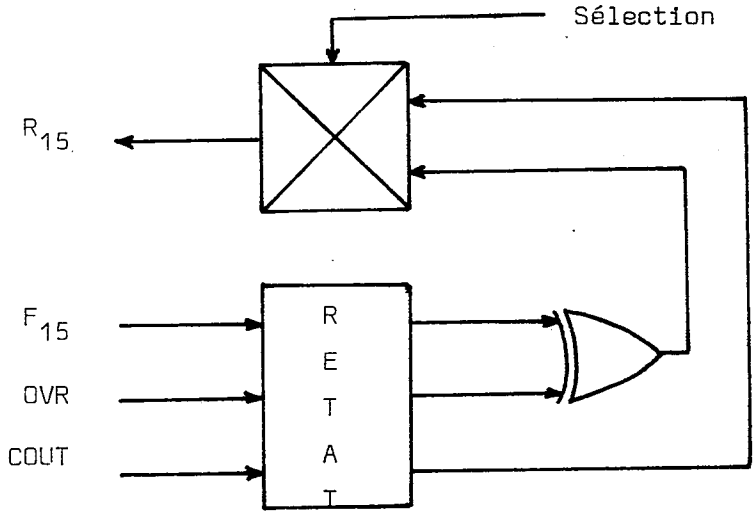
Le bit de poids fort du multiplicateur ne représentant pas un signe, la 16ième opération est également une addition.

I.3. Le multiplicande n'est pas signé

Dans ce cas, le résultat partiel d'une addition est toujours positif, mais peut être accompagné d'une retenue. On réalise donc $RAM_{15} = C_{OUT}$;

Conséguence:

L'entrée RAM_{15} recevra soit C_{OUT} soit $F_{15} \oplus OVR$, ce qui donne lieu au montage suivant:



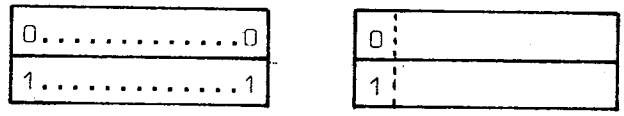
1.4. Test de la longueur significative du résultat

Les poids faibles du résultat sont dans l'accumulateur Q, les poids forts dans R_0. Le résultat devant globalement se trouver dans le couple (R'_2, R_2), il faut transférer Q dans R_2 et R_0 dans R'_2.

On profite de ces transferts pour tester le bit de signe des poids faibles et tester la signification des poids forts:

```
        Q → R2, (10) → L2
        si POSITIF alors JMP(TEST), R0 → R'2
        R0
TEST    si ZERO alors (01) → L2
```

Ainsi, les résultats tels que (1) ou (2) sont réduits à



16 bits significatifs (L_2=01), ce qui représente une optimisation intéressante pour les opérations suivantes.

1.5. Test de la nullité d'un des opérandes

Sachant que si l'un des opérandes est nul, alors le résultat est nul, il est intéressant de détecter ce fait, à condition que le test ne pénalise pas les performances.

Dans notre cas, les tests ne "coûtent rien", parce qu'ils sont faits en même temps que des transferts obligatoires.

1/ transfert du 1er opérande dans Q

```
OP1 → Q
```

2/ Transfert du 2ème opérande dans R1 et test du 1er opérande

```
OP2 → R1, si ZERO alors JMP(ZERO)
```

3/ Initialisation du résultat partiel et test du 2ème opérande

```
0 → R0, si ZERO alors JMP(ZERO)
```

1.6. Tableau des performances

16*16	1er opérande NUL	3 cycles
	2ème opérande NUL	4 cycles
	Résultat POSITIF	22 cycles
	Résultat NEGATIF	23 cycles

(16*32)	$B_2 = 0$	3 cycles
	$B_1 = 0$	28/30 cycles
	$A_1 = 0$	24 cycles
	général	45/47 cycles

II. - Multiplication 16*32

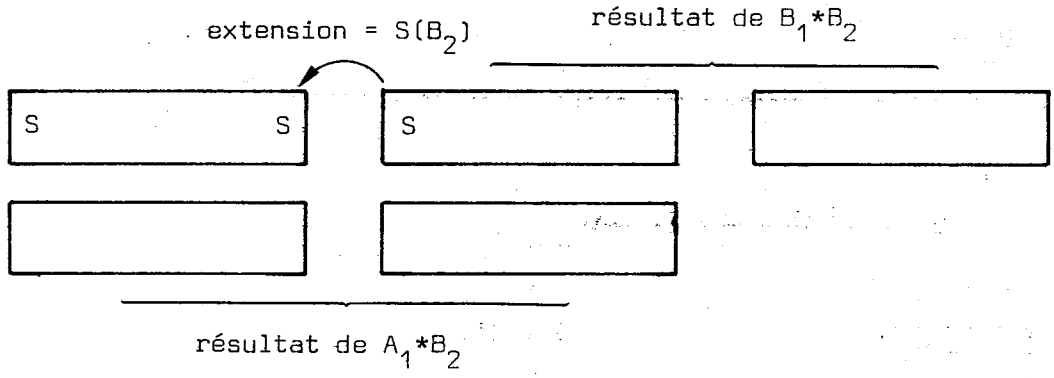
Soit B_2 le multiplicateur de 16 bits en complément à 2 et $(A_1 . 2^{16}+B_1)$ le multiplicande de 32 bits en complément à 2.

La multiplication $(A_1 . 2^{16}+B_1)\ B_2$ est décomposée en $A_1\ B_2 . 2^{16}+B_1\ B_2$.

La difficulté provient du fait que:

- A_1 est signé
- B_1 n'est pas signé
- B_2 est signé.

On a donc une multiplication $B_1 * B_2$ d'un multiplicande B_1 non signé et d'un multiplicateur B_2 signé. Le signe de ce résultat partiel sera donc celui de B_2, dont on devra tenir compte lors du calcul du résultat final.



Fin de la 1ère multiplication

$Q \rightarrow R_2$

$R_0 \rightarrow R'_2$

$0 \rightarrow R''_2$, si POSITIF alors JMP(SUITE)

$\overline{R''}_2 \rightarrow R''_2$

Fin de la 2ème multiplication

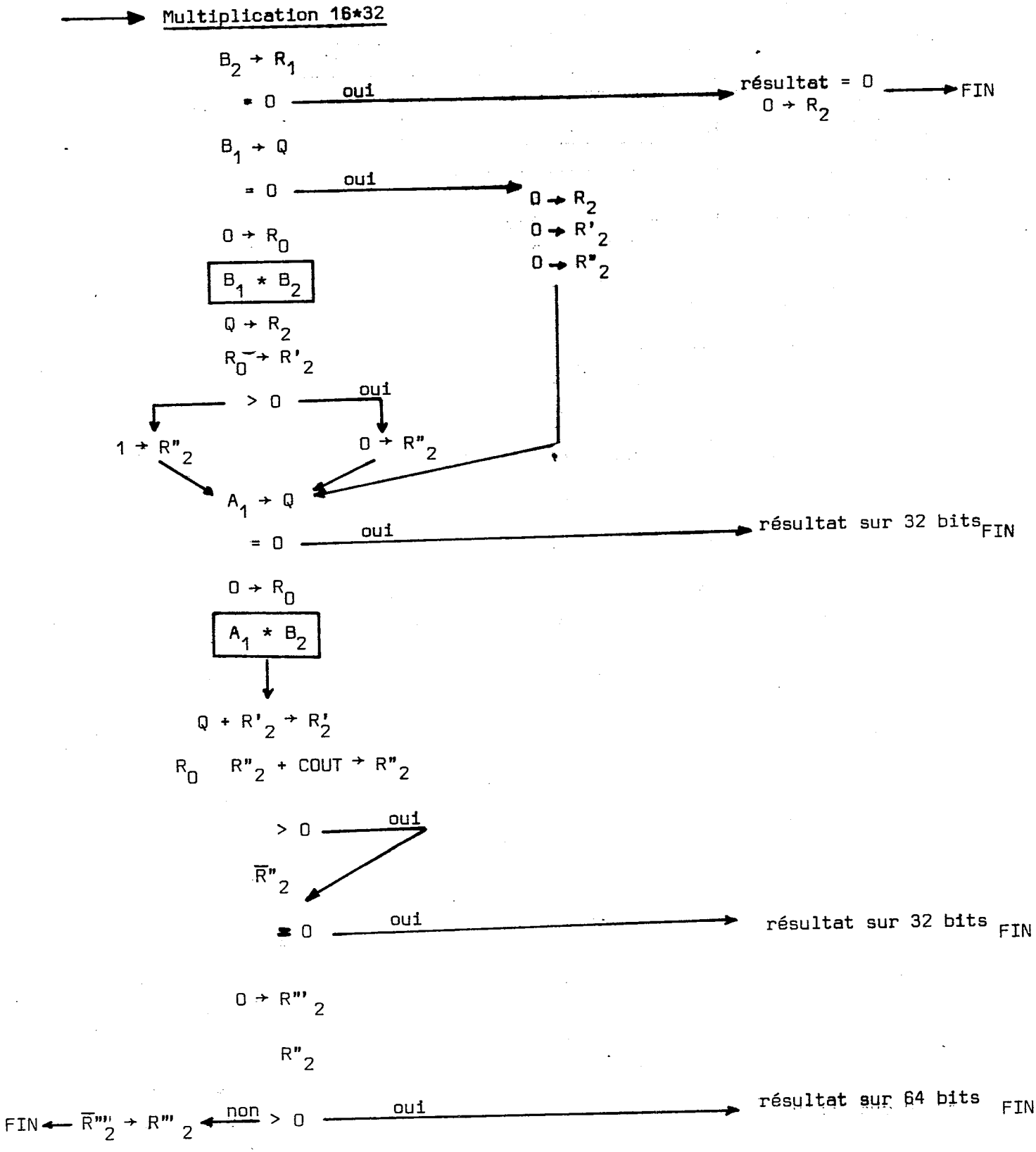
$Q + R'_2 \rightarrow R'_2$, $(11) \rightarrow L_2$

$R_0 + R''_2 + C_Y \rightarrow R''_2$, si POSITIF alors JMP(TEST)

$\overline{R''}_2$

si ZERO alors $(10 \rightarrow L_2)$ et JMP(FIN), $0 \rightarrow R'''_2$

$\overline{R'''}_2 \rightarrow R'''_2$, JMP(FIN)


Multiplication 16*32
B2 → R1
= 0
oui
résultat = 0
0 → R2
FIN
B1 → Q
= 0
oui
0 → R2
0 → R'2
0 → R"2
0 → R0
B1 * B2
Q → R2
R0 → R'2
> 0
oui
1 → R"2
0 → R"2
A1 → Q
= 0
oui
résultat sur 32 bits
FIN
0 → R0
A1 * B2
Q + R'2 → R'2
R0 R"2 + COUT → R"2
> 0
oui
R"2
= 0
oui
résultat sur 32 bits
FIN
0 → R"'2
R"2
FIN
R"'2 → R"'2
non
> 0
oui
résultat sur 64 bits
FIN

TEST $0 \rightarrow R'''_2$, si ZERO alors $(10 \rightarrow L_2)$ et JMP(FIN)

Comme pour la multiplication 16*16, on regarde si le résultat de la multiplication 16*32 est significatif sur 32 bits, sinon il est étendu à 64 bits, pour des raisons de format (les longueurs connues par le processeur POP sont 8 bits (L=00), 16 bits (L=01), 32 bits (L=10) et 64 bits (L=11), ce qui correspond aux notions d'alignement (octet, demi-mot, mot et double mot).

M_2 significatif sur 16 bits $\Rightarrow$ $0 \leq M2 < 2^{15}$

M_1 significatif sur 32 bits $\Rightarrow$ $M_1 \geq 2^{15}$

Résultat significatif sur 32 bits $\Rightarrow$ $M_1 * M_2 < 2^{31}$

si $M_1 < 2^{16}$, donc si $A_1 = 0$
on a $0 \leq M_1 * M_2 < 2^{16} * 2^{15} = 2^{31}$

Donc, si A_1=0, on est certain que le résultat est significatif sur seulement 32 bits.

III.- Multiplication 32*32

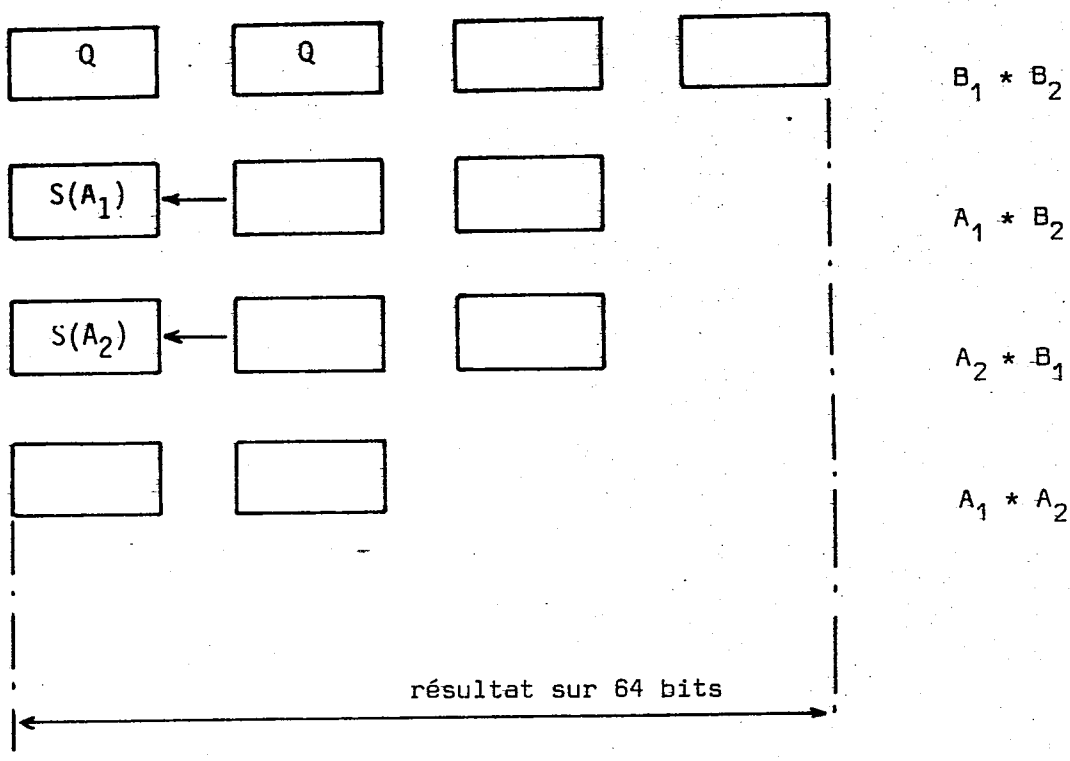
Les deux opérandes sont en complément à 2 et sous la forme $(A_1 . 2^{16}+B_1)$ et $(A_2 . 2^{16}+B_2)$. Ils sont significatifs sur 32 bits, donc on n'a pas à la fois A_1=0 et B_1=0 ni à la fois A_2=0 et B_2=0, mais on peut avoir un ou plusieurs facteurs nuls (par exemple B_1=0 et A_2=0).

La multiplication est décomposée en:

$$A_1*A_2 . 2^{32}+(A_1*B_2 + A_2*B_1).2^{16} + B_1*B_2$$

Les deux facteurs B_1 et B_2 ne sont pas signés et les deux facteurs A_1 et A_2 le sont.

On effectue donc les 4 multiplications et on étend le signe du résultat.



Le résultat de la multiplication peut être significatif sur seulement 32 bits. En effet, on sait que $A_1 . 2^{16} + B_1 \geq 2^{15}$ et $A_2 . 2^{16} + B_2 \geq 2^{15}$, donc le résultat est $\geq 2^{30}$. S'il est $< 2^{31}$, alors il est significatif sur 32 bits seulement.

Posons $A_1 . 2^{16} + B_1 = 2^{15} + \varepsilon_1$, $\varepsilon_1 \geq 0$

$A_2 . 2^{16} + B_2 = 2^{15} + \varepsilon_2$, $\varepsilon_2 \geq 0$

Il faut trouver ε_1 et ε_2 tels que

$$(2^{15} + \varepsilon_1) * (2^{15} + \varepsilon_2) < 2^{31}$$

soit $$2^{30} + (\varepsilon_1 + \varepsilon_2) . 2^{15} + \varepsilon_1 * \varepsilon_2 < 2^{31}$$

Une solution triviale est $\varepsilon_1 = \varepsilon_2 = 0$.

L'ensemble des solutions est donné par la formule suivante en posant:

$$\varepsilon_1 + \varepsilon_2 = S \quad \text{et} \quad \varepsilon_1 * \varepsilon_2 = P$$

$$2^{15} . S + P < 2^{30}$$

pour $\varepsilon_1 = 0$ on obtient $\varepsilon_2 < 2^{15}$

$\varepsilon_1 = 1$ on obtient $\varepsilon_2 < 2^{15} \cdot \dfrac{2^{15}-1}{2^{15}+1}$

$\vdots$

$\varepsilon_1 = 2^i$ on obtient $\varepsilon_2 < 2^{15} \cdot \dfrac{2^{15}-2^i}{2^{15}+2^i} = 2^{15} \cdot \dfrac{2^{15-i}-1}{2^{15-i}+1}$

et plus généralement $\varepsilon_2 < 2^{15} \cdot \dfrac{2^{15}-\varepsilon_1}{2^{15}+\varepsilon_1}$

Exemple:

pour $\varepsilon_1 = 2^{13}$ on obtient $\varepsilon_2 < \frac{4}{5} \cdot 2^{15}$

pour $\varepsilon_1 = 2^{14}$ on obtient $\varepsilon_2 < \frac{1}{3} \cdot 2^{15}$

Remarque:

Ce résultat est directement transposable à la multiplication 16*16: un résultat est significatif sur 16 bits seulement s'il est inférieur à 2^{15}.

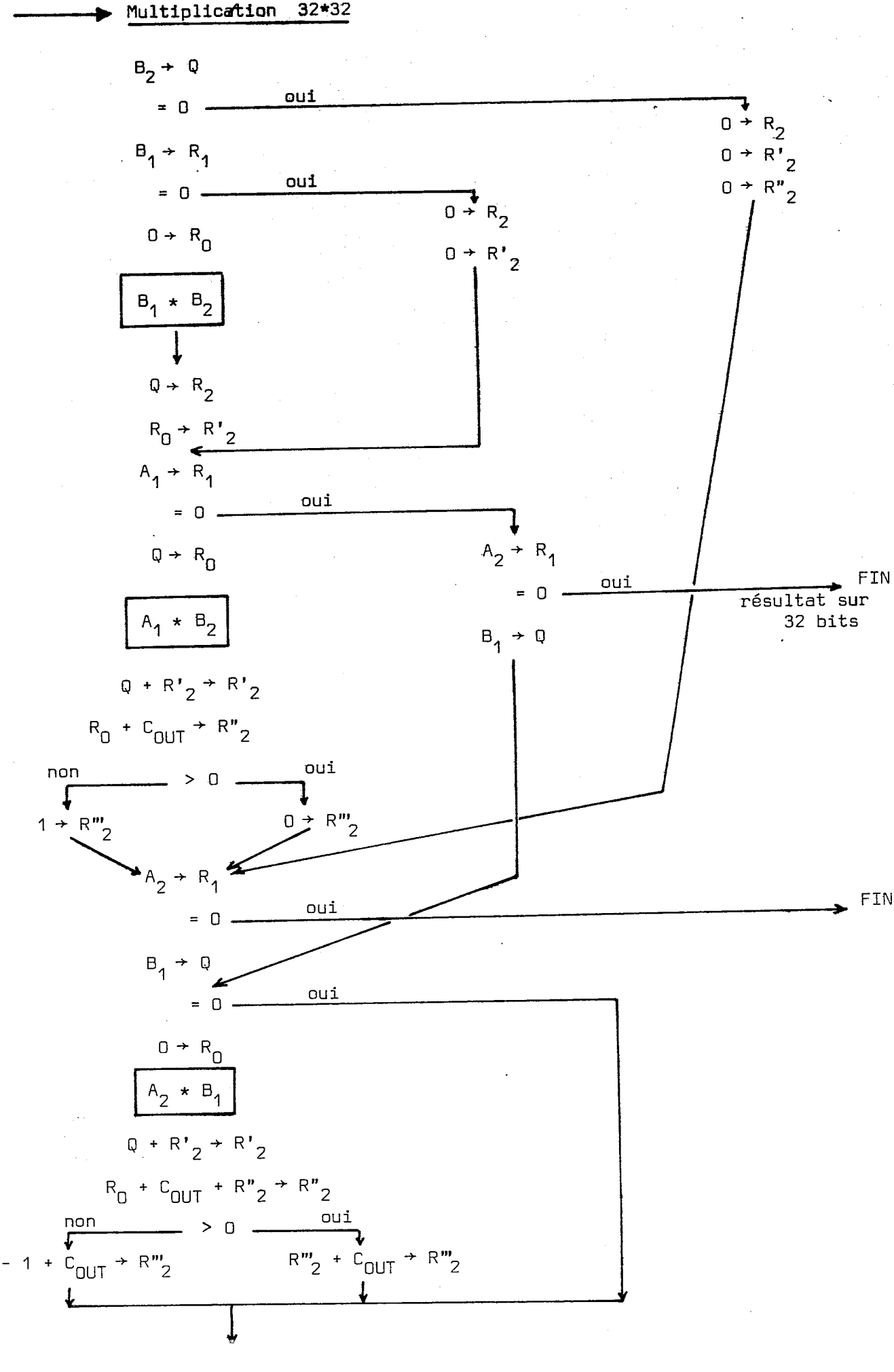
Conséquence importante:

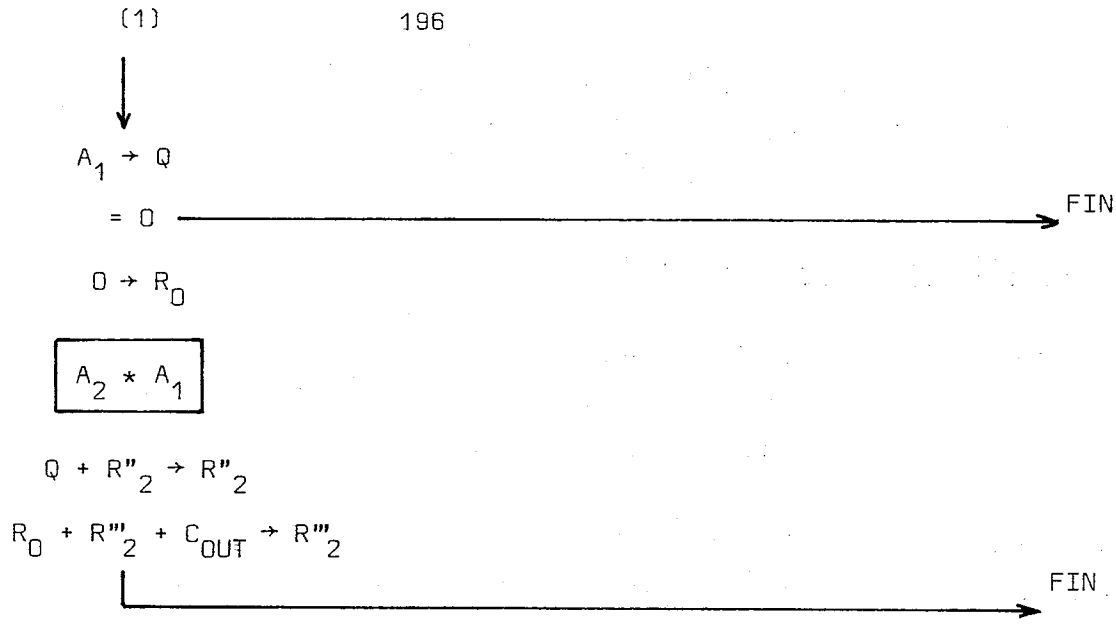
Le résultat d'une multiplication de deux opérandes significatifs sur 32 bits ne peut être significatif sur 32 bits que si les deux opérandes sont inférieurs à 2^{16}, c'est-à-dire si $A_1 = A_2 = 0$.

On sait que $M_1 \geq 2^{15}$ et $M_2 \geq 2^{15}$.

Supposons que $M_1 \geq 2^{16}$, alors $M_1 * M_2 \geq 2^{31}$, donc le résultat n'est plus significatif sur 32 bits.

Ce raisonnement est valable pour des opérandes positifs.


Multiplication 32*32
B2 → Q
= 0
oui
0 → R2
0 → R'2
0 → R"2
B1 → R1
= 0
oui
0 → R2
0 → R'2
0 → R0
B1 * B2
Q → R2
R0 → R'2
A1 → R1
= 0
oui
A2 → R1
= 0
oui
FIN
résultat sur 32 bits
Q → R0
A1 * B2
B1 → Q
Q + R'2 → R'2
R0 + COUT → R"2
non
> 0
oui
1 → R"'2
0 → R"'2
A2 → R1
= 0
oui
FIN
B1 → Q
= 0
oui
0 → R0
A2 * B1
Q + R'2 → R'2
R0 + COUT + R"2 → R"2
non
> 0
oui
R"'2 - 1 + COUT → R"'2
R"'2 + COUT → R"'2



PERFORMANCES DE LA MULTIPLICATION 32*32

Conditions	Nombre de cycles
Cas général	84
$B_1 = 0$	49
$B_2 = 0$	47
$A_1 = 0$	47
$A_2 = 0$	44
$A_1 = A_2 = 0$	25
$A_1 = B_2 = 0$	29
$A_2 = B_1 = 0$	28
$B_1 = B_2 = 0$	28

V - ALGORITHMES DE DIVISION ENTIERE

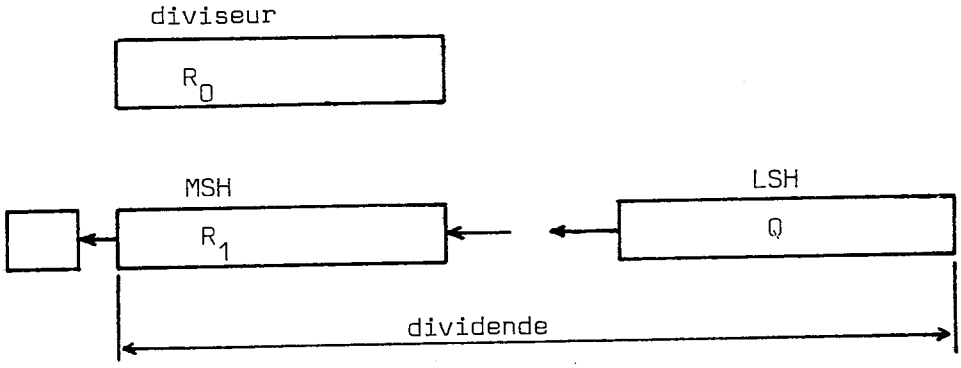
Sachant que nous disposons d'un opérateur de 16 bits, la division entière la plus simple consiste à diviser un dividende E de 32 bits par un diviseur D de 16 bits, pour obtenir un quotient Q sur 16 bits et un reste R sur 16 bits.

Cette opération est réalisée par une séquence d'addition ou soustraction suivie d'un décalage à gauche de une position: à chaque étape, le diviseur D est soustrait aux poids forts du diviseur E. Si le résultat est positif, la puissance correspondante de E contient une fois D et on entre un '1' dans le poids correspondant du quotient Q.

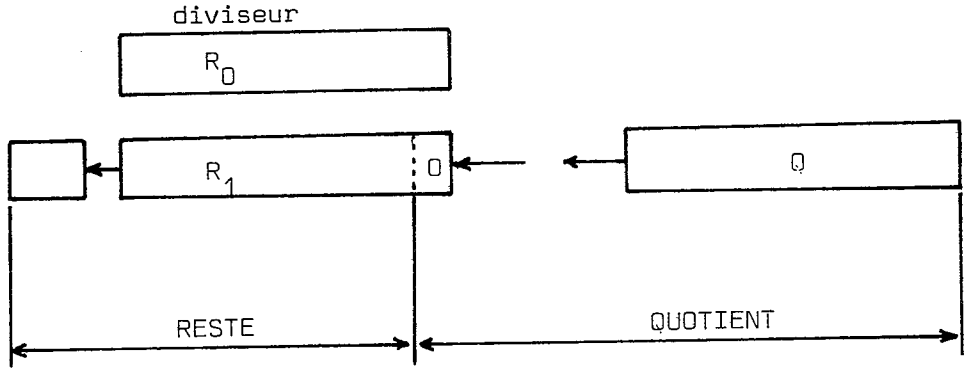
Si le résultat est négatif, on entre un '0' dans le quotient et on fait une addition au pas suivant pour compenser l'effet de l'opération précédente.

Lors de la première opération de soustraction, le résultat doit être négatif, sinon la division est impossible (quotient trop grand).

ETAT INITIAL:



ETAT FINAL:



REALISATION DE LA DIVISION AVEC L'OPERATEUR AM 2901

- le diviseur (2ème opérande) est placé dans un registre R_2,
- le dividende (1er opérande) est décomposé en poids faibles placés dans l'accumulateur Q et en poids forts placés dans un registre R_1.

Les deux opérandes sont positifs, ce qui suppose éventuellement une ou deux complémentations, et le calcul du signe du résultat.

1er cycle: le diviseur est soustrait au dividende, le résultat est décalé circulairement à gauche de 1 position.

2ème cycle: on teste si le résultat précédent est négatif. Sinon, alors la division 32 bits par 16 bits est impossible.

1: (R1-R0)*2 → R1, $Q_0 \rightarrow \overline{RAM}_{15}$, FLAG → RAM_{15}, $RAM_0 \rightarrow Q_{15}$

2: si $\overline{FLAG}$ alors JMP(IMPOSSIBLE)

On effectue ensuite 16 cycles d'addition ou soustraction, l'opération étant déterminée par la valeur de flag.

(16 fois) (R1±R0)*2 → R1

+ si flag=1
- si flag=0

Selon que l'on exécute l'opération DIV(division entière) ou l'opération MOD (reste d'une division entière), on s'intéresse respectivement au quotient ou au reste.

ETUDE DU QUOTIENT

Le quotient est obtenu dans l'accumulateur Q. Si le bit de plus fort poids du quotient vaut '1', il y a en quelque sorte un débordement, donc le résultat est donné sur 32 bits:

Q → R2

0 → R'_2 et (10) → L_2

Dans le cas contraire le résultat tient sur 16 bits:

$Q \rightarrow R2$ et $(01) \rightarrow L2$

Algorithme

$Q \rightarrow R2$, $(01) \rightarrow L2$

si $\overline{F15}$ alors JMP(FIN)

$0 \rightarrow R'_2$, $(10) \rightarrow L2$, JMP(FIN)

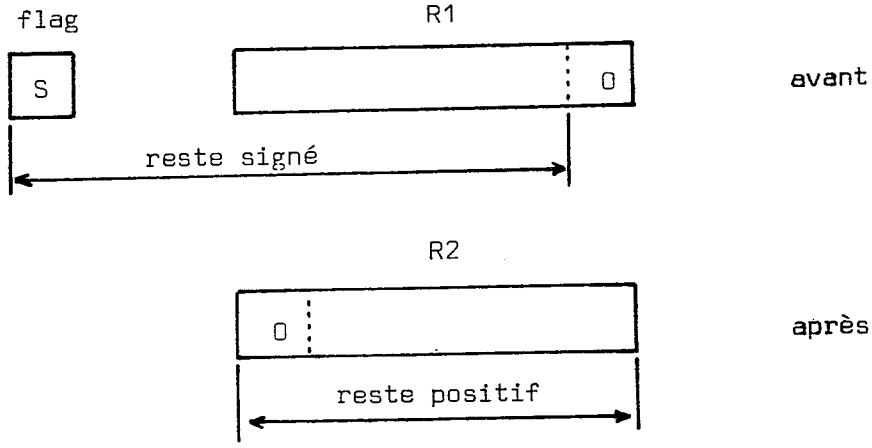
ETUDE DU RESTE

Le reste est obtenu dans R1, à la place des poids forts du dividende, mais décalé à gauche d'une position. Il faut donc le décaler à droite, et, éventuellement, lui ajouter la valeur du diviseur s'il est négatif.

Algorithme

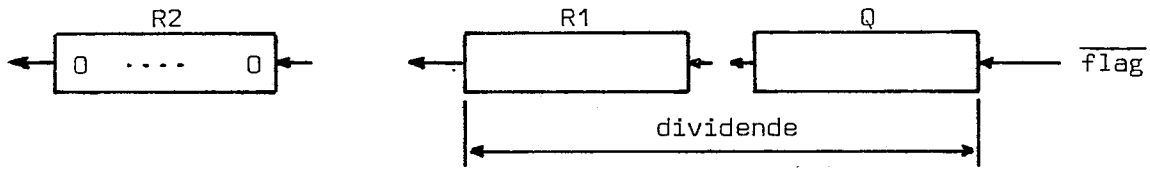
si $\overline{FLAG}$ alors JMP(FIN) et $R1/2 \rightarrow R2$ et $(01) \rightarrow L_2$

$R1+R0 \rightarrow R2$



CAS D'IMPOSSIBILITE

Lorsque la première soustraction (R_1-R_0) donne un résultat non négatif, on sait que le quotient ne tient pas sur 16 bits: il faut donc étendre la capacité du quotient à un nombre plus élevé de bits, c'est-à-dire 32 dans notre cas. On étend donc le dividende à gauche en lui ajoutant 16 bits de poids forts tous nuls et on exécute le même algorithme.



R2-R0 → R2, flag=F15

Q*2 → Q, Q_0=$\overline{flag}$, flag=Q_{15}

R_1*2 → R_1, RAM_0=flag, flag=RAM_{15}

R_2*2 → R_2, RAM_0=flag, flag=RAM_{15}

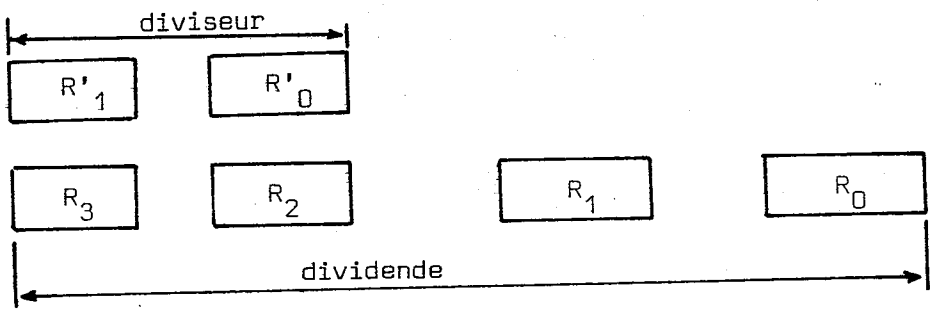
16 fois {
- $R_2 \pm R_0 \rightarrow R_2$
- Q*2 → Q
- R_1*2 → R_1
- R_2*2 → R_2

16 fois {
- $R_2 \pm R_0 \rightarrow R_2$
- Q*2 → Q
- R_1*2 → R_1
- R_2*2 → R_2

DIVIDENDE DE LONGUEUR 64 bits

Un résultat de multiplication peut se trouver sur 64 bits. Dans ce cas, le diviseur est étendu à 32 bits. Si la première soustraction donne un résultat positif, alors une arreur est détectée parce que le quotient ne tient pas sur 32 bits.

On peut également prévoir une division de 64 bits par 64 bits, sur le même principe, mais cela conduit à des performances très faibles.



Algorithme

R_2-R_0 → R_2

R_3-R'_1-OVR → R_3

si $\overline{F15}$ alors JMP(ERREUR) → DEPASSEMENT

$R_0 * 2 \rightarrow R_0$

$R_1 * 2 \rightarrow R_1$

$R_2 * 2 \rightarrow R_2$

$R_3 * 2 \rightarrow R_3$

(32 fois)
$R_2 \pm R'_0 \rightarrow R2$

$R_3 \pm R'_1 + C_{OUT} \rightarrow R_3$

$R_0 * 2 \rightarrow R_0$

$R_1 * 2 \rightarrow R_1$

$R_2 * 2 \rightarrow R_2$

$R_3 * 2 \rightarrow R_3$

VI- LES OPERATEURS ENSEMBLISTES

Les opérations d'union ou d'intersection de deux ensembles sont respectivement réalisées par un OU et un ET logique sur deux chaînes de bits. Elles ne posent donc pas de problèmes.

Par contre, le test de l'appartenance (opérateur IN) ou la création d'un ensemble (opérateurs ELEM et INTERVAL) requièrent l'utilisation d'une mémoire PROM, dans le chemin de données, pour générer des profils binaires de 16 bits de 2 types:

- soit un '1' en ième position,
- soit une suite de '1' à partir de la ième position.

1 - Opérateur IN

Son premier opérande est un entier sur 8 bits, son second opérande est un ensemble.

Microprogramme

$V_1 \rightarrow$ ADROM, BOOL $\rightarrow TB_2$

$ROM.R_2 \rightarrow R_2$

On obtient directement la valeur booléenne FAUX caractérisée par la valeur ZERO et la valeur VRAI par une valeur différente de ZERO.

2 - Opérateur ELEM

Il ajoute à l'ensemble qu'est son premier opérande un bit en position égale à la valeur du deuxième opérande: le résultat est un ensemble.

$R_2 \rightarrow$ ADROM, $V_1 \rightarrow R_2$
ROM+$R_2 \rightarrow R_2$

3 - Opérateur INTERVAL

Ses deux opérandes sont des entiers qui indiquent les positions extrêmes des éléments qui constituent l'"intervalle". Pour des raisons de formatage, on construit systématiquement un ensemble de 64 bits, qui sera "UNI" par un opérateur d'UNION à l'ensemble en cours de construction.

$R_2 \rightarrow$ ADROM
$0 \rightarrow R_2^i$ pour i=0 à 3
ROM $\rightarrow R_2^j$
$\bar{R}_2^i \rightarrow R_2^i$ pour i=0 à 3
$R_1 + 1 \rightarrow$ ADROM
ROM.$R_2^j \rightarrow R_2^j$
$0 \rightarrow R_2^i$ pour i>j

CHAPITRE 6

LE PROCESSEUR DE GESTION DES FILES

PROCESSEUR F I L E

A - INTRODUCTION D'UN PROCESSEUR FILE

La machine PASCALE a une architecture pipe-line permettant la désynchroni ation des processus de contrôle, d'accés et opératif résultant de la décomposition naturelle du processus d'exécution d'une expression.

Pour réaliser la désynchronisation entre ACcés et OPération, on a introduit une FILE d'EVALuation dans laquelle le processeur d'ACcés (PAC) range les descripteurs de variables accédés et dans laquelle le processeur OPératif (POP) vient extraire les opérandes et ranger les résultats intermédiaires.

Pour résoudre le problème des dépendances sur la valeur des variables en voie de modification (problème posé par toutes les machines ayant une architecture pipe-line) une file de dépendances (FILEDEP) a été introduite.

La gestion des deux files FILEVAL et FILEDEP demande un certain nombre de pointeurs, les uns utilisés par le processeur d'ACcés, les autres par le processeur OPératif:

- pointeurs dans FILEVAL:

  . ECR: pointeur d'écriture donnant l'adresse à laquelle PAC doit ranger le prochain descripteur.

  . P2: indique à POP la position du deuxième opérande d'un opérateur diadique et sert de pointeur d'écriture d'un résultat intermédiaire (ou définitif)à POP .

  . P1: indique à POP la position du premier opérande pour un opérateur diadique (ou de l'opérande pour un opérateur monadique.

. FIN: donne la limite de l'espace de travail utilisé par POP.

- pointeurs dans FILEDEP:

. NIN: donne l'adresse à laquelle PAC doit venir initialiser la prochaine dépendance.

. NOUT: donne l'adresse à partir de laquelle POP devra résoudre des dépendances.

Chacun de ces pointeurs est utilisé soit par PAC, soit par POP, on peut donc les diviser en deux classes:

- ECR, NIN , connus de PAC seulement,
- P2, P1, FIN, NOUT, connus de POP seulement.

Cette classification est un peu simpliste car on a,à tout moment, besoin de comparer un pointeur d'une classe avec un pointeur de l'autre classe. exemples:

- accés à un nouvel opérande:

Avant d'écrire un descripteur de variable en FILEVAL (ECR), on doit s'assurer que cette écriture n'aille pas écraser un opérande à l'intérieur de la zone de travail de POP, donc comparer ECR et FIN.

- transition opérande-opérateur dans la chaîne post-fixée d'entrée:

Une telle transition détectée par le processeur de contrôle PINS entraîne la génération d'un extra-ordre AVANCE(N) pour POP qui devra, entre autre, incrémenter de N son pointeur P2. Avant d'exécuter P2 ← P2+N, on doit s'assurer qu'il n'y aura pas dépassement du pointeur ECR, donc comparer P2+N avec ECR.

Pour alléger d'une part les échanges de pointeurs entre POP et PAC, et

d'autre part pour supprimer des algorithmes de ces deux processeurs la gestion de FILEVAL et de FILEDEP, on a été amené à introduire un quatrième processeur: le processeur FILE, contenant:

- la file d'évaluation,
- la file des dépendances,
- les pointeurs sur ces files dans ses registres internes,
- un opérateur permettant toutes les opérations sur les pointeurs,
- son propre microprogramme permettant toute la gestion de manière autonome.

Il communique avec les deux autres processeurs par un bus de 32 bits, le bus FILE, dont il est le maître et qui sert à véhiculer les descripteurs de variables.

Quand PAC doit ranger un descripteur ou bien quand POP a besoin soit d'un opérande, soit de ranger un résultat intermédiaire, ils formulent une requête au processeur FILE qui sera prise en compte sous certaines conditions.

FILE exécute les opérations sur les pointeurs et les transferts de données nécessaires, puis il s'acquite et attend de nouvelles requêtes.

Les pointeurs ne sont plus connus ni de POP, ni de PAC, ce sont des variables internes à FILE.

FILEVAL et FILEDEP sont des mémoires de 16 mots de 32 bits, donc 4 bits suffisent pour les adresses. Les opérations à exécuter sur les pointeurs sont simples et peuvent être prises en compte par une tranche AM 2901 dans laquelle les 6 pointeurs + 1 registre de travail RT pourront être implantés.

B - STRUCTURE DU CHEMIN DE DONNEES DU PROCESSEUR FILE

Le processeur FILE est bâti autour de ces deux bus dont il est le maître:

- le bus FILE bidirectionnel (BF) de 32 bits véhiculant les données de PAC vers POP via les FILES,
- le bus pointeur (2x4 bits unidirectionnels BPTE et BPTS) permettant l'adressage dans les FILES.

Remarque:

Tous les registres source et destinations de ces bus sont commandés par la partie contrôle du processeur FILE, y compris ceux situés sur PAC et POP.

Le bus FILE accepte des données venant de:

source
- la file d'évaluation (FILEVAL),
- la file des dépendances (FILEDEP),
- le registre d'écriture du processeur PAC (PAC-E),
- le registre d'écriture du processeur POP (POP-E),
- le bus pointeur de sortie RTS (pour la constitution de chaînes de reprise pour les dépendances).

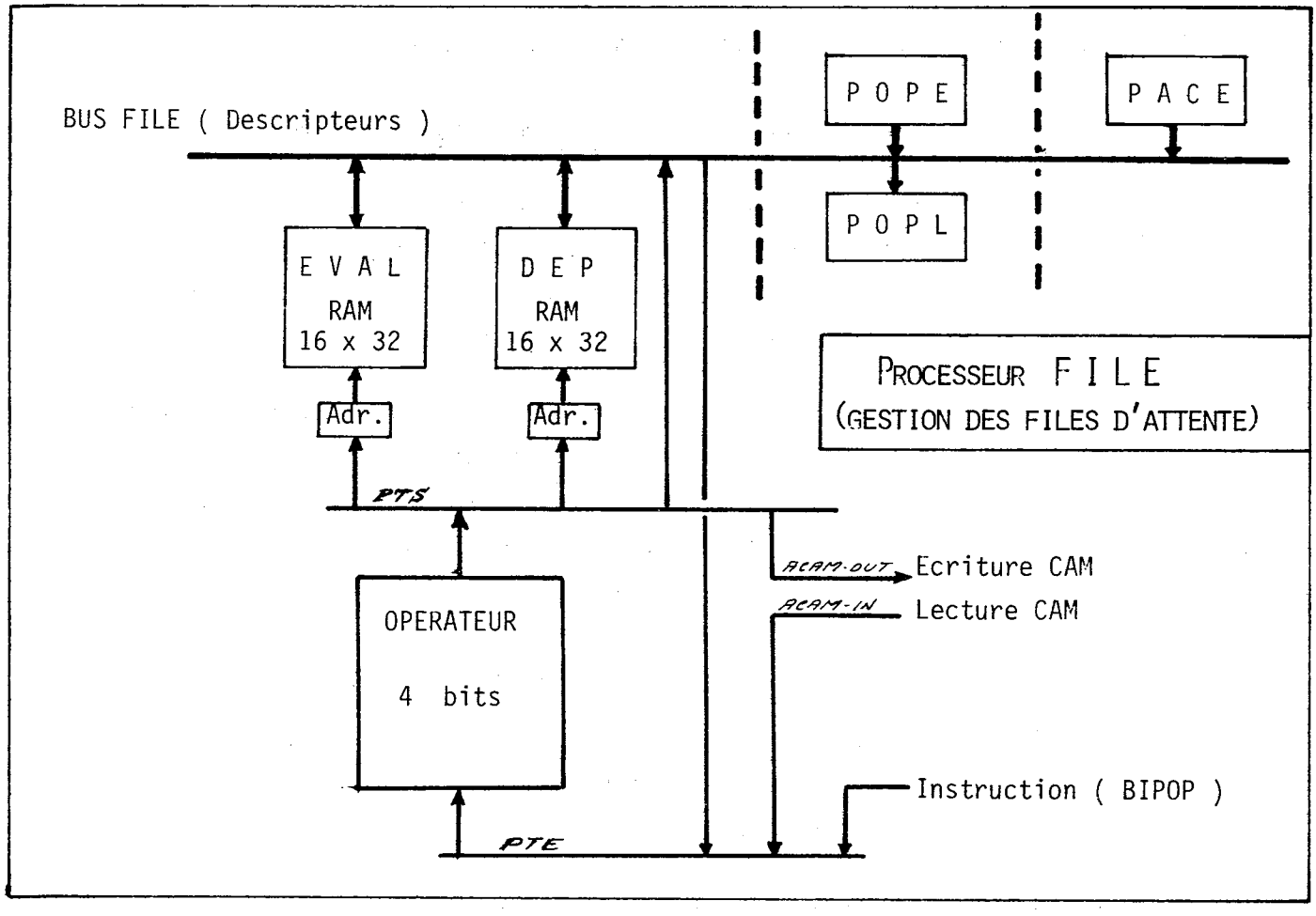
Il peut avoir pour destination:

destination
- la file d'évaluation,
- la file des dépendances,
- le registre de lecture du processeur POP (POP-L).

Le bus pointeur est divisé en:

- bus PTE ayant une seule destination, l'entrée D de l'opérateur AM 2901 multiplexée entre:
  - . ACAM-OUT l'adresse associée venant de PAC,
  - . BIPOP (poids faible) contenant le paramètre N nécessaire à certaines instructions (par exemple AVANCE(N)),
  - . les 4 bits poids faible du bus FILE.

- bus PTS ayant une seule source possible (la sortie Y de l'opérateur AM 2901) et comme destination:
  - . ADEVAL le registre d'adresse dans FILEVAL,
  - . ADDEP le registre d'adresse dans FILEDEP,
  - . ACAM-IN (adresse d'écriture des paramètres RESOLU ou VIDE dans la partie associative de FILEDEP se trouvant sur PAC).



C - PARTIE CONTROLE - STRUCTURE GENERALE DU MICROPROGRAMME

- CYCLE D'ALLOCATION

Le processeur FILE est autonome et décide pour qui il doit travailler: pour cela une priorité sur les demandes a été établie et FILE peut en outre masquer certaines demandes.

Les requêtes sont mémorisées dans des bascules situées sur le processeur FILE et positionnées à 1 par POP ou PAC.

FILE peut être sollicité de trois manières différentes:

. par l'intermédiaire de la bascule TROMPE mise à un par POP

On a vu que, pour les instructions conditionnelles PINS choisit une alternative par anticipation sur la valeur d'un prédicat évalué par POP. En cas de mauvais choix, POP en avertit d'une part PINS et d'autre part FILE, par l'intermédiaire de la bascule TROMPE.

C'est la demande la plus prioritaire (priorité 1) et elle ne peut être masquée. Elle joue le rôle d'une interruption non masquable prise en compte au moment du cycle d'allocation du processeur FILE. Celle-ci doit exécuter un algorithme de remise à jour des files d'EVAluation et des DEPendances.

. par l'intermédiaire de la bascule DPOP mise à 1 par POP

Cette requête est utilisée par POP en fonctionnement normal pour demander un opérande, ranger un résultat partiel ou définitif ou pour l'exécution d'un extra-ordre sur les pointeurs.

Elle peut être masquée par exemple par une instruction AVANCE(N) qui donnerait pour nouvelle valeur de P2 une valeur supérieure modulo 16 à celle du pointeur d'écriture ECR.

. par l'intermédiaire de la bascule DPAC mise à un par PAC

DPAC est la demande la moins prioritaire, elle est utilisée par PAC quand il a un descripteur à transmettre à FILE.

Elle peut être masquée par exemple par une instruction d'ACCES si ECR+1 devient égal au pointeur FIN donnant la limite inférieure de la zone de travail utilisée par POP.

- RECHERCHE DE L'INSTRUCTION A EXECUTER (point d'entrées dans le microprogramme de FILE)

La fin du cycle d'allocation se traduit par un branchement, soit:

→ au début de l'algorithme de remise à jour des Files (TROMPE),
→ soit à l'adresse donnée par BIPOP (Allocation à POP),
→ soit à l'adresse donnée par CODEPAC (Allocation à PAC).

Pour chaque cas, la première instruction lévera les masquages des demandes pour le prochain cycle d'allocation.

. instructions passées par PAC

Elles sont de deux types: ACCES ou AFFECT, chacun d'eux étant paramétré par l'indicateur TROUVE et le bit RESOLU venant de la partie associative de la file des DEPendances située sur PAC. CODEPAC sera donc constitué par un bit de BIPAC distinguant le type ACCES du type AFFECT plus les deux paramètres TROUVE et RESOLU.

Ceci donne les points d'entrées suivant dans le microprogramme de FILE:

- ACCES, $\overline{TR}$: FILE doit aller chercher un descripteur chargé dans PACE pour le transférer en FILEVAL(ECR).

- ACCES, TR, $\overline{RES}$: FILE doit construire un maillon de chaîne de reprise dans FILEVAL la variable étant sous dépendance. Pour cela

il doit transférer le contenu de FILEDEP(ACAM) dans FILEVAL(ECR) et modifier le champ pointeur de FILEDEP(ACAM):

FILEDEP(ACAM). POINTEUR ← ECR.

ACAM est l'adresse résultant de la recherche associative sur le SD faite par PAC avant de formuler sa requête à FILE.

- ACCES, TR, RES: Le descripteur à écrire en FILEVAL(ECR) se trouve dans la zone "cache" de la file des dépendances. FILE n'a qu'à le recopier, ce qui permet ainsi de gagner un accés à la mémoire centrale:

FILEVAL(ECR) ← FILEDEP(ACAM)

- AFFECT, $\overline{TR}$: FILE doit d'une part aller chercher le mot construit par PAC dans PACE pour le ranger en FILEVAL(ECR) et d'autre part initialiser une dépendance en écrivant le pointeur NIL dans FILEDEP(NIN).

- AFFECT, TR, $\overline{RES}$: FILE met PAC en attente et retourne au cycle d'allocation.

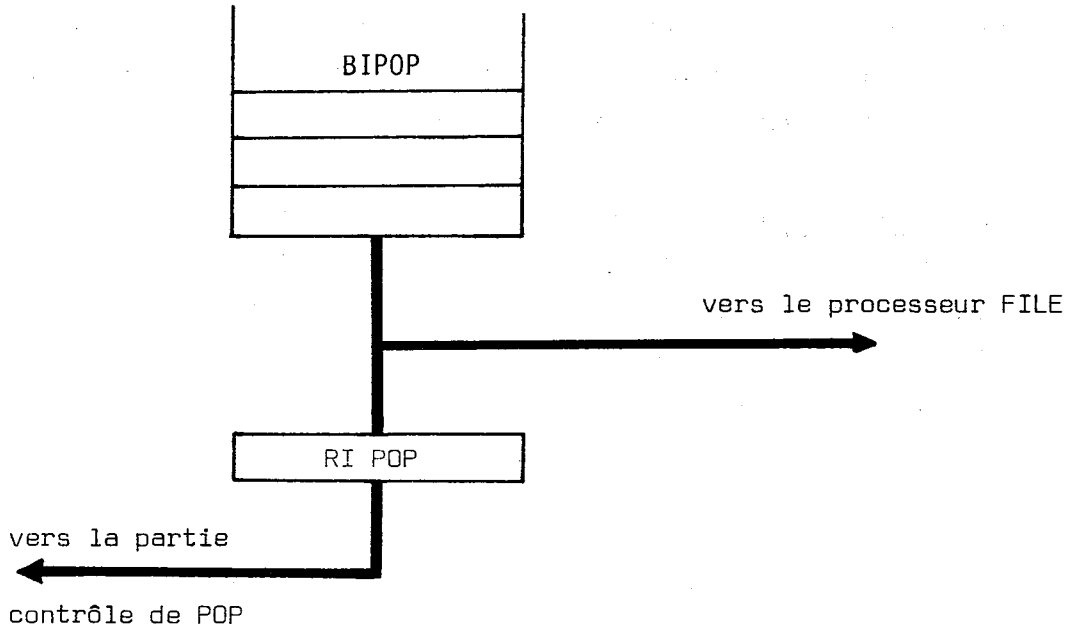
- AFFECT, TR, RES : FILE inscrit le bit (TROUVE) dans CAM(ACAM) puis il doit initialiser une nouvelle dépendance comme dans le cas d'un AFFECT, $\overline{TR}$ à la différence près que le descripteur à écrire en FILEVAL(ECR) provient en partie de FILEDEP(ACAM) pour les poids forts (préfixe et pointeur sur type) et en partie de PACEo(adresse de la valeur).

. instructions passées par POP

Vis-à-vis de POP, FILE a un rôle de préparation (passage des opérandes nécessaires à l'évaluation d'un opérateur par exemple).

Il parait donc judicieux d'introduire un niveau supplémentaire de pipeline entre FILE et POP. Ainsi parallélement à l'exécution d'une instruction par POP, FILE peut analyser et préparer éventuellement le transfert d'un opérande vers POP.

Pour cela, un registre tampon RIPOP entre BIPOP et le contrôle de POP a été introduit: pendant que POP traite l'instruction de RIPOP, FILE analyse celle située à la sortie de BIPOP.



Chaque fois qu'il fait une recherche de l'instruction suivante se traduisant par un chargement de RIPOP et un décalage dans BIPOP, POP positionne sa requête (DPOP←1).

Les instructions codées dans BIPOP donnent le point d'entrée suivant dans le microprogramme de FILE:

- CTRL1 et UNAIRE: CTRL1 ou UNAIRE se présente vis-à-vis de FILE comme des opérateurs monadiques pour lesquels il a déjà ses opérandes Donc il n'a pas de transfert de données à faire. FILE s'acquitte simplement (DPOP←0) et repasse à son cycle d'allocation.

- CTRL2 et BINAIRE: CTRL2 et BINAIRE se présentent vis-à-vis de FILE comme des opérateurs diadiques. POP dispose déjà du deuxième opérande et il faut lui passer le premier, c'est-à-dire exécuter le transfert du descripteur se trouvant en FILEVAL(P1) dans le registre de lecture POPL. Il doit en outre décrémenter P1 de 1 et, si la nouvelle valeur de P1 devient égale à FIN, faire FIN ← P2-1 pour récupérer de la place dans FILE D'EVALuation. Ensuite il remet à zéro la requête de POP et remonte à son cycle d'allocation.

- AFFECT POP: AFFECT POP est un opérateur diadique particulier. Le processeur FILE doit incrémenter P2 de 1 si cette incrémentation est possible (sinon il met POP en attente volontaire) pour pouvoir passer à POP FILEVAL(P2 contenant un descripteur sous la forme PF, PT, PU, PV étant l'adresse à laquelle POP devra ranger le résultat qu'il est en train d'évaluer.

Quand POP a terminé son évaluation, il doit retourner son résultat à FILE qui le range dans l'ancien FILEVAL(P2) (utilisation du registre de travail RT), puis doit résoudre les dépendances éventuelles en allant lire FILEDEP.(NOUT) et en remontant la chaîne de reprise dans FILEVAL en écrivant POPE.

FILE doit aussi masquer la partie associative de la file des dépendances avec le bit RESOLU.

Ensuite il remet DPOP à zéro et remonte à son cycle d'allocation.

- INIT(N): Cette instruction suit une instruction de "fin d'expression" dans BIPOP. Elle est générée par PINS quand il rencontre une transition ACCES → OPERATEUR dans une séquence:

<fin d'expression><accés> <opérateur>.

FILE doit incrémenter son pointeur P2 de N si cette incrémentation est possible (sinon il met POP en attente volontaire: ordre ATVPOP) et range

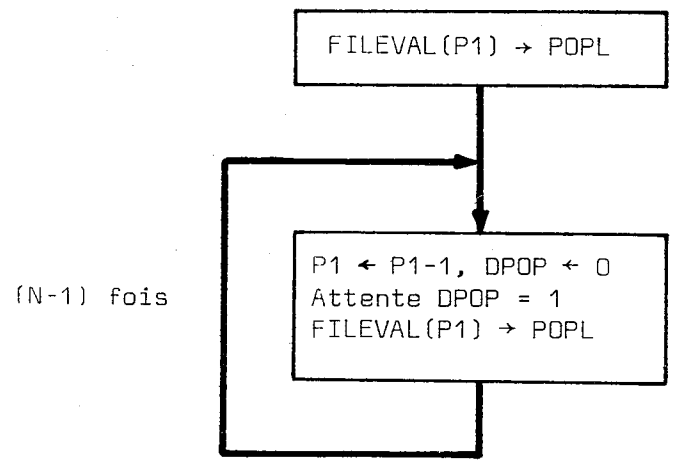
dans POPL FILE(P2) qui est le descripteur de son deuxième opérande pour l'opérateur diadique suivant (ou l'opérande de l'opérateur monadique suivant). Ensuite il remet DPOP à zéro et remonte à son cycle d'allocation.

- AVANCE(N): Cette instruction suit un opérateur dans BIPOP. Elle est générée par PINS quand il rencontre une transition ACCES → OPERATEUR si les instructions d'accés étaient précédées d'un opérateur.

FILE exécute le même algorithme que pour INIT(N) dans une première phase, puis quand POP a terminé l'exécution de l'opérateur précédent, il retourne à FILE le résultat que celui-ci range dans l'ancien FILE(P2).

- SAUVE(N): Cette instruction est générée par PINS lors d'une transition opérateur → appel de fonction. Il s'agit de sauvegarder N opérandes de la FILE d'EVAluation pour faire place nette en vue de l'évaluation de la fonction dont le résultat constitue un opérande de l'expression à évaluer.

Le transfert de N opérandes entre FILE et POP se fait par le pointeur P1 au rythme imposé par POP. L'algorithme est le suivant:



Après le transfert de N opérandes, FILE s'acquitte définitivement et remonte à son cycle d'allocation.

Remarque:
Cette instruction se déroule en synchronisme entre POP et FILE, seul le premier descripteur peut être passé pendant l'exécution par POP de l'opérateur précédent.

- INITSAUVE(N): Cette instruction est générée par POP quand l'occurrence d'un appel de fonction intervient après une instruction d'accés.

Elle est interprétée par FILE comme la concaténation d'une instruction INIT(1) suivie d'une instruction SAUVE(N).

- RECUL(N): PINS connait l'image de la file d'évaluation sous la forme d'une suite

$$\{(d_0, t_0), \ldots; (d_i, t_i); \ldots; (d_k, t_k)\}$$

Quand il rencontre un opérateur diadique il décrémente d_k et incrémente t_k. Quand d_k devient nul, il génère un ordre RECUL (t_{k-1}) pour que le pointeur P1 saute par dessus t_{k-1} cases "vides" de FILEVAL.

L'interprétation de RECUL(N) sera donc:

P1 ← P1-N ; puis remise à zéro de DPOP et retour au cycle d'allocation.

- OF(N): Il s'agit de passer à POP N étiquettes une à une pour qu'il puisse les comparer avec un prédicat qu'il vient d'évaluer. Excepté pour la première étiquette, cette instruction doit se dérouler en synchronisme avec POP. Celui-ci peut interrompre le déroulement de cette instruction par le fil TROMPE.

L'exécution de cette instruction commence comme l'exécution d'une instruction INIT(N) qui a pour effet de passer la première étiquette à comparer.

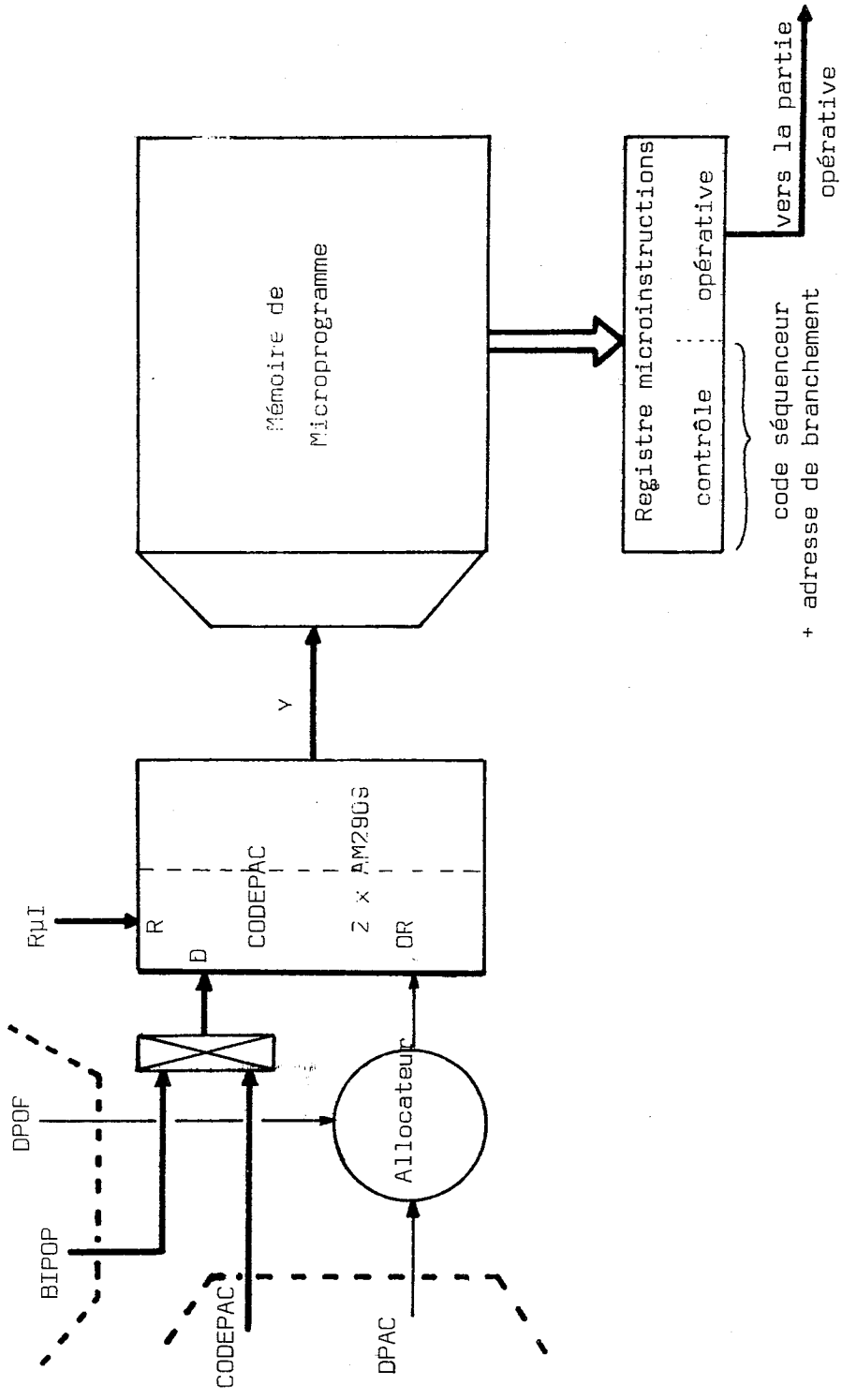
S'il n'y a pas égalité POP demande une nouvelle étiquette que FILE lui passe par P1:

FILEVAL(P1) → POPL

P1 → P1-1

et ceci jusqu'à (N-1) fois.

S'il y a égalité, POP en avertit FIFE par TROMPE qui aura pour effet une remise à jour des files



CHAPITRE 7

LE PROCESSEUR DE DIALOGUE AVEC LA MEMOIRE CENTRALE

LE PROCESSEUR PME

La connexion de la machine PASC-HLL au système-hôte est réalisée par le Processeur PME qui assure l'interface avec la Mémoire Centrale de la machine-hôte. Cette dernière n'étant pas encore connue, nous avons dû en définir un modèle réaliste, et nous avons choisi une Mémoire Centrale hypothétique dont le temps de cycle serait situé entre 350 et 400 nanosecondes.

Cette hypothèse introduit un facteur 2 entre le temps de cycle probable de la machine PASC-HLL et celui de la Mémoire Centrale, ce qui paraît raisonnable tant du point de vue de la technologie des mémoires RAM que du point de vue des performances des macrocomposants SFC 92900, et respecte une hiérarchie de performances satisfaisante.

A / - LE CHEMIN DE DONNEES DU PROCESSEUR PME

Comme nous l'avons défini dans le chapître relatif à l'espace d'adressage , le processeur PME joue le rôle d'allocateur de la ressource Mémoire Centrale entre les trois processeurs PINS, PAC et POP : telle est sa fonction de contrôle que nous exposerons plus loin.

Sa fonction opérative consiste à réaliser une translation d'adresse : une adresse "virtuelle" émise par un processeur doit être transformée en une adresse "réelle", correspondant à l'implantation "réelle" dans la Mémoire Centrale, en ajoutant un déplacement de 14 bits à la valeur de la base de la zône dont le numéro est codé sur 2 bits (de poids fort de l'adresse virtuelle).

I - CHARGEMENT DES REGISTRES DE BASE DES 4 ZONES

La valeur des 4 registres de base doit être fournie par la Machine-Système. Par convention, ces 4 valeurs seront implantées aux adresses 0,1,2 et 3 (adresses réelles) du bloc de Mémoire alloué à PASC-HLL. Ainsi, pour démarrer

l'exécution d'un programme, le processeur PME commence par charger ses 4 registres de base dans la RAM d'un opérateur SFC 92901 (5 tranches), en lisant les 4 premiers mots en Mémoire Centrale.

II - EXECUTION DU CALCUL D'ADRESSE

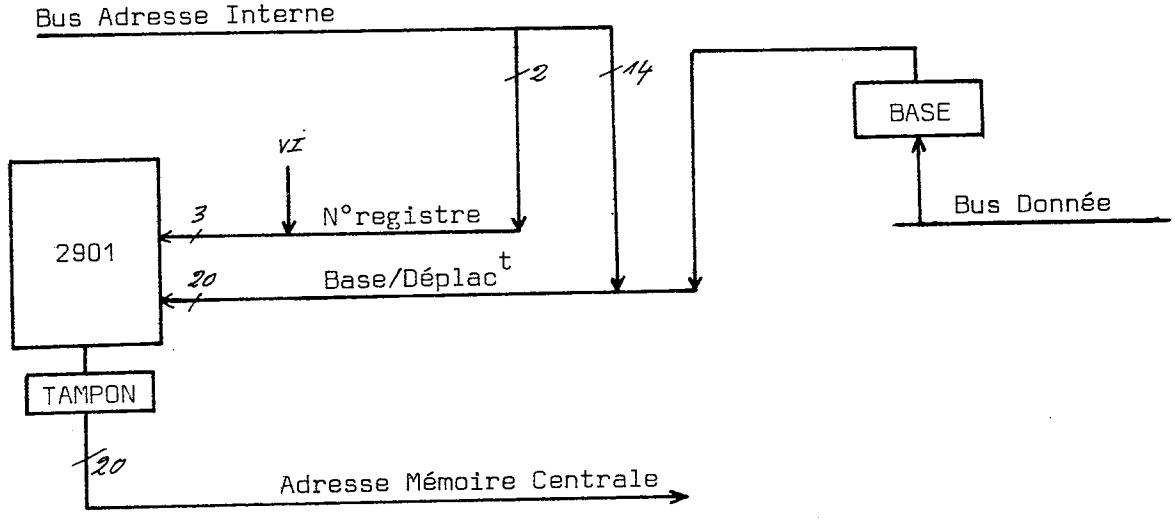
Lorsque les 4 registres de base ont été chargés, le processeur PME devient disponible pour satisfaire les demandes de lecture/écriture émanant des 3 autres processeurs. Lorsqu'il alloue la ressource Mémoire Centrale à l'un d'eux, il obtient une adresse virtuelle, qu'il décompose en (N° de zone,Déplacement), et il exécute l'opération :

BASE(N° de zone) + Déplacement

dont le résultat donne l'adresse réelle qui est envoyée à la Mémoire Centrale.

III - SCHEMA DU CHEMIN DE DONNEES

Il faut multiplexer à la fois le N° de registre envoyé à l'opérateur et la donnée d'entrée, selon que l'on se trouve en MODE "initialisation" ou en MODE "allocation". La sortie de l'opérateur doit également être "tamponnée" si l'on veut avoir un recouvrement entre les opérations d'allocation et les échanges avec la Mémoire.



B / - LA FONCTION D'ALLOCATION

Cette fonction doit être réalisée très rapidement, de manière à pouvoir utiliser la Mémoire Centrale à sa vitesse maximum. Nous avons donc cherché à faire en parallèle la fonction d'allocation, le calcul de l'adresse réelle et les échanges avec la Mémoire.

I - DISTINCTION ENTRE ADRESSE ET DONNEE

Le parallélisme que l'on recherche suppose une indépendance entre l'allocation du bus des adresses et celle du bus des donées : le premier est alloué au prochain utilisateur, le second à celui dont l'opération est en cours.
Ainsi, pendant un cycle, le processeur i est connecté au bus des données, et le processeur j au bus des adresses : le processeur PME peut donc calculer l'adresse réelle pour le processeur j, adresse qui sera envoyée à la Mémoire au cycle suivant.

Pendant les cycles d'allocation, le processeur PME prend en compte les demandes, et met à 1 une bascule BMA_i correspondant au demandeur le plus prioritaire. Pendant le cycle d'opération qui suit le calcul d'adresse est effectué, le résultat est chargé dans le TAMPON d'adresse, et l'opération Mémoire est demandée : elle dure l'équivalent de deux cycles de microinstruction.

II - INTERFACE ENTRE PME ET LES PROCESSEURS UTILISATEURS

Chaque processeur utilisateur de la Mémoire Centrale communique avec PME par deux signaux de contrôle :
- un signal de demande DEM_i,
- un signal d'allocation $ALLOC_i$.

Il est d'autre part connecté , lorsque la Mémoire lui est allouée, à des bus qui sont :

- le bus ECR dont la valeur indique Lecture/ecriture,
- les deux bus DEBOP et FINOP qui indiquent le début (allocation du bus des données) et la fin (libération) d'un échange avec la Mémoire,
- le bus Adresse constitué de 16 bits d'adresse interne et de 4 bits de sélection des octets à l'intérieur d'un mot,
- le bus de Données de 32 bits (+parité).

La figure suivante donne le microprogramme du processeur PME.

0	0→RAM(4) et RADM, ordre(DEBOP)	
1	NOOP	
2	RAM(4)+1→RAM(4) et RADM, ordre(DEBOP)	
3	D→RAM(0)	charge BASE(0)
4	RAM(4)+1→RAM(4) et RADM, ordre(DEBOP)	
5	D→RAM(1)	charge BASE(1)
6	RAM(4)+1→RAM(4) et RADM, ordre(DEBOP)	
7	D→RAM(2)	charge BASE(2)
8	NOOP	
9	D→RAM(3)	charge BASE(3)
10	ordre(MODE←1)	passe en mode "allocation"
11	RAM(N°de zone)+Déplac^{t}→TAMPON	boucle sur calcul d'adresse

Les microinstructions n° 0,2,4 et 6 commandent la lecture des mots d'adresses 0,1,2 et 3 (contenu de RAM(4)). Sachant que deux cycles sont nécessaires pour la lecture les valeurs des BASES sont chargées par les microinstructions n° 3,5, 7 et 9.
En mode "allocation" le processeur PME exécute en permanence la microinstruction n° 11 qui effectue le calcul d'adresse qui ne sera pris encompte que lorsqu'une opération d'échange sera effectivement initialisée.

Il suffira donc d'une seule "tranche" SFC 92909 pour réaliser le séquencement de ce microprogramme constitué de 12 microinstructions : PME sera donc le plus petit processeur microprogrammé de PASC-HLL.

C O N C L U S I O N

CONCLUSION

Ce document ne donne aucun résultat chiffré sur les performances globales de PASC-HLL. Il apparait cependant que PASC-HLL fait environ deux fois moins d'accès à la Mémoire Centrale qu'une machine classique, donc pourrait être pratiquement au moins deux fois plus "puissante" qu'une machine de la même catégorie travaillant sur la même Mémoire Centrale.

Les performances individuelles des processeurs pour l'exécution d'opérateurs complexes montrent cependant certains avantages, surtout si l'on considère qu'il existe dans PASC-HLL un haut degré de parallélisme, et que les fonctions de gestion du système d'exploitation sont réalisées en parallèle avec le traitement des programmes.

Nous avons bien entendu envisagé la mise en oeuvre d'une simulation progammée pou réaliser des mesures de performances. Le premier obstacle rencontré est la finesse requise pour cette simulation : elle doit être faite au niveau de chaque cycle de microinstruction, pour tenir compte du parallélisme réel au niveau des files d'attente et du partage de la Mémoire Centrale. Le volume des programmes de simulation, le temps de mise en oeuvre et le coût des heures de calcul qui auraient été nécessaires nous ont conduits à abandonner cette voie, qui présentait de plus un risque d'enlisement.

Nous avons donc fait un PARI sur la faisabilité de PASC-HLL (et en ce sens nous n'avons fait qu'imiter Blaise PASCAL) : la réalisation physique est en cours. Parallèlement nous développons le système MADAM, un Matériel d'Aide au Développement d'Applications Microprogrammées, qui permet de contrôler la mise au point "en temps réel" de l'ensemble des cinq processeurs microprogrammés(un microprocesseur connecté à un Télétype permet de charger et modifier les microprogrammes ainsi que de diriger leur exécution par un système de points d'arrêt et de pas-à-pas).

Nous espérons que cet outil permettra, dans un avenir très proche, de mesurer les performances réelles du prototype de PASC-HLL.

Nous pensons d'autre part que cette étude a couvert de nombreux aspects de l'informatique et de l'architecture des ordinateurs, pour lesquels elle apporte des éclairages nouveaux et suscitera peut-être des approches nouvelles, tant pour l'organisation des systèmes que pour la définition des langages-machine. Elle représente également un projet intéressant et significatif pour la connaissance du fonctionnement et de la mise en oeuvre de ces composants modernes que sont les macrocomposants SFC 92900, et qui ouvrent des perspectives nouvelles aux machines microprogrammées pour la conception de processeurs spécialisés, qui restent utiles là où les microprocesseurs classiques sont dépassés.

Cette réalisation en cours, dans une université française, d'une machine ambitieuse par son architecture porte en elle-même son intérêt, et nous espérons que cette expérience pourra susciter d'autres investigations constructives.

Enfin cette étude illustre une démarche de conception descendante qui est difficile à suivre dans un contexte industriel figé, mais qui peut trouver une application intéressante dans la conception de nouveaux produits : elle permet de trouver les meilleurs compromis entre les moyens mis en oeuvre et les besoins à satisfaire. Il faut cependant se limiter à la considérer comme une philosophie constructive et non pas comme une automatisation inhumaine de la conception des ordinateurs qui restera marquée par l'art des architectes.

ANNEXE

ANNEXE 1

METHODOLOGIE DE CONCEPTION DESCENDANTE

Présentation générale

PLAN

INTRODUCTION

La démarche de conception d'un calculateur est guidée par le but qu'on s'est fixé: traiter une classe de problèmes donnée.
Tout calculateur spécialisé doit être conçu de manière à répondre aux spécifications initiales:

- quel est le travail à réaliser? (PROJET)
- de quels moyens dispose-t-on pour atteindre le but fixé? (MOYENS)

Une bonne connaissance du but à atteindre nous semble être essentielle pour mener à bien une telle démarche.

"Si vous ne savez pas où vous allez, vous avez de fortes chances de vous retrouver ailleurs".

LJ. PETER et R.HULL (Principe de PETER 1969)

On peut envisager deux méthodes opposées pour définir un calculateur:

1 - Méthode ascendante (MOYENS → PROJET)

La première consiste à regrouper des opérations élémentaires existantes (moyens) pour construire des opérations plus complexes susceptibles de satisfaire les désirs des utilisateurs. C'est la technique ascendante classique, qui conduit généralement à la définition de calculateurs inadaptés aux besoins de leurs utilisateurs humains.

"Pour aller quelque part, en général, le plus simple était encore de partir de là où on voulait aller, et avec du temps et un peu de chance, on y arrivait effectivement".

J.ROUXEL (LES SHADOKS, 1968)

2 - Méthode descendante (PROJET → MOYENS)

La seconde méthode que nous développons dans cette étude est dite "descendante".

"Il est beaucoup plus intéressant de regarder où l'on ne va pas, pour la bonne raison que là où l'on va, il sera toujours temps d'y regarder quand on y sera".

J.ROUXEL (LES SHADOKS, 1968)

Partant des spécifications émises par les utilisateurs futurs du calculateur (projet), la méthode descendante consiste à définir progressivement une architecture adaptée (moyens), jusqu'au niveau de la réalisation technologique qui constitue le but de la démarche. Nous pensons en effet qu'actuellement, les architectes de calculateurs possèdent une bonne connaissance de ce qu'il est souhaitable de définir, au niveau du logiciel, pour répondre au mieux aux besoins des utilisateurs humains.

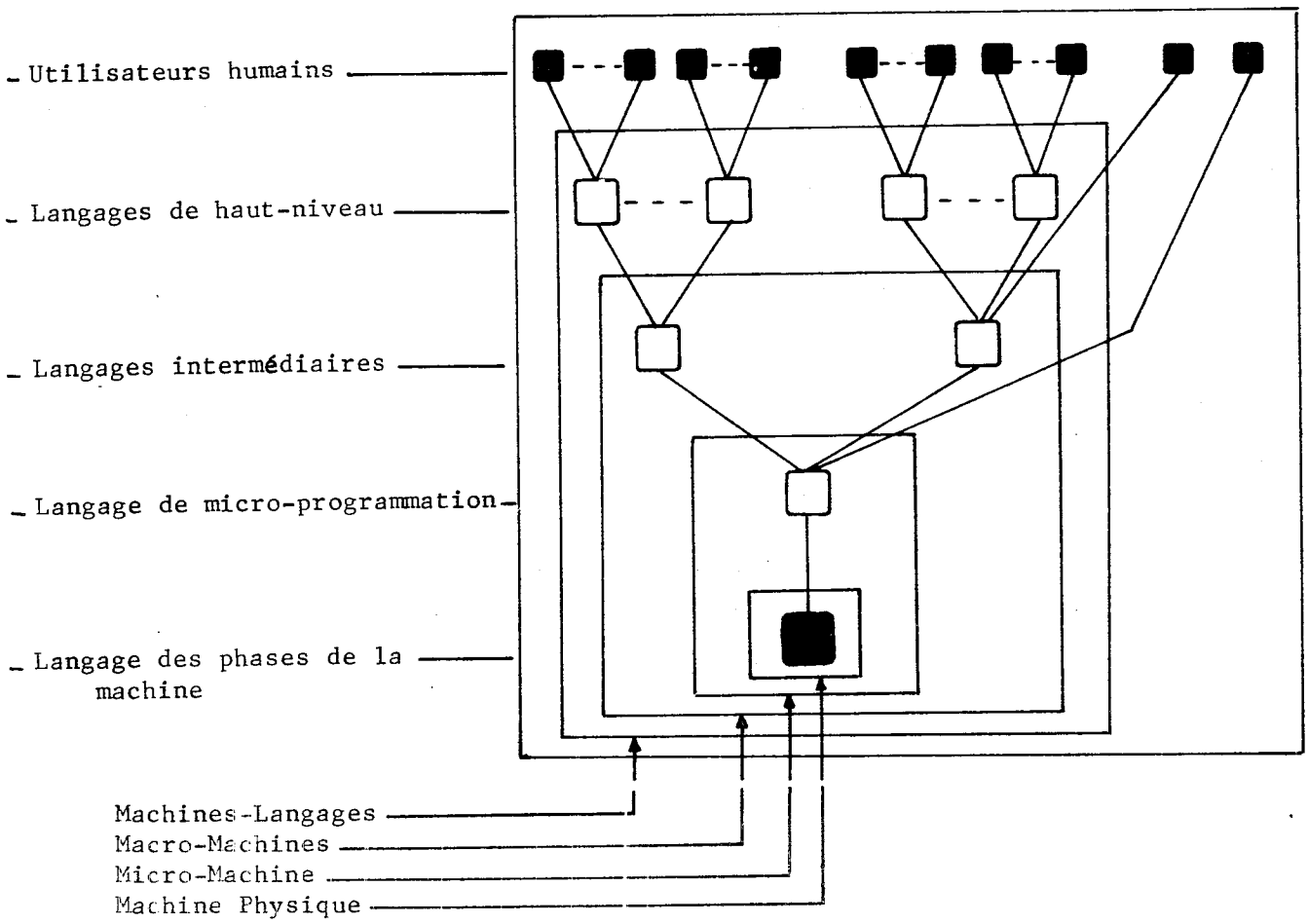
D'autre part, nous pensons que l'évolution des possibilités du matériel (LSI en particulier) autorise une approche descendante pour la définition d'architectures nouvelles basées sur un matériel nouveau et adapté à son utilisation. La façon dont la nature du but à atteindre influence chaque pas de la démarche est très variable et doit faire l'objet d'une grande attention.

I - ORGANISATION VERTICALE D'UN CALCULATEUR

La complexité des problèmes des utilisateurs humains d'un calculateur impose une décomposition de ces problèmes sur des structures de plus en plus simples (machines) en utilisant des formalismes de plus en plus simples (langages) jusqu'à ce qu'un traitement physique puisse être envisagé.

Généralement, les calculateurs sont organisés d'une manière verticale: la complexité du traitement décroît depuis les langages de haut niveau jusqu'aux langages directement interprétés par un assemblage de composants électroniques.

figure 1



I/1. NIVEAUX D'INTERPRETATION

Une machine informatique peut être vue comme un empilement de niveaux d'interprétation de langages de plus en plus simples qui constituent une hiérarchie: chaque niveau est tel qu'il émule (voir figure 1) les niveaux qui le précèdent dans la hiérarchie.

Sachant que le but d'une machine informatique est de répondre aux besoins de ses utilisateurs en respectant des critères de coût, performance, fiabilité...., il semble raisonnable d'envisager une conception progressive, guidée par les spécifications initiales des utilisateurs.

Si nous prenons l'exemple d'une machine adaptée au traitement d'un langage évolué, appelons L_0 ce langage de haut niveau. L'utilisateur communique avec la machine par l'intermédiaire de "programmes" écrits dans le langage L_0. La première étape d'une démarche descendante consiste à rechercher la manière optimale, en regard des possibilités usuelles des calculateurs, d'interpréter les programmes du langage L_0 : c'est l'étude des spécifications initiales, qui, selon la nature de L_0 peut conduire à la définition d'un langage intermédiaire L_1, directement issu de L_0 et qui soit "interprétable". (Cet aspect de la démarche est présenté dans une autre partie du présent rapport: "Etude d'une machine PASCAL").

A partir de ce stade, nous entrons dans le cycle des niveaux d'interprétation.

Le langage L_1 est "interprétable" et on peut définir l'algorithme d'interprétation I_1.

Cet algorithme d'interprétation est conditionné par l'existence d'utilisateurs du langage L_0 dont les programmes-type, traduits dans le langage L_1, définissent les caractéristiques de leur interprétation: on peut

donc mesurer le comportement dynamique de l'interpréteur de L_1 , et choisir ainsi, en connaissance de cause, une réalisation de l'algorithme d'interprétation de L_1 , qui tiendra également compte des libertés et des contraintes liées au type de réalisation que l'on projette (influence du but).

On choisira, par exemple, un ensemble de "primitives" qui deviendront des MICRO-INSTRUCTIONS, et l'algorithme d'interprétation I_1 du langage L_1 deviendra un MICRO-PROGRAMME P_1 .

L'ensemble des primitives précédentes définit un langage L_2 , (dans lequel est écrit le MICRO-programme), dont on peut également définir l'algorithme d'interprétation I_2 . Le processus est ainsi itéré jusqu'à ce qu'un langage L_n soit tel que son algorithme d'interprétation I_n soit jugé réalisable sous la forme d'un assemblage de composants électroniques.

La caractéristique essentielle d'une telle approche est qu'elle permet d'aborder chaque niveau d'interprétation en connaissance des caractéristiques qu'il doit remplir: chaque niveau doit émuler les niveaux qui le précèdent, sa spécialisation est donc parfaitement déterminée. Les compromis possibles sont guidés par ces caractéristiques: l'architecte est en possession du "cahier des charges" de la réalisation qu'il doit définir à chaque niveau.

I/2. UTILISATION D'UN LANGAGE DE DESCRIPTION DE HAUT NIVEAU. DEFINITION D'UN SYSTEME D'ASSISTANCE POUR L'ARCHITECTE.

Dans la démarche proposée, l'architecte est amené à décrire les algorithmes d'interprétation $I_1,\ldots,I_n$ des langages $L_1,\ldots,L_n$ successivement définis.

Il semble intéressant de lui proposer un langage unique de description qui soit

- puissant et agréable d'emploi, c'est-à-dire un langage de haut niveau,
- adapté aux contraintes inhérentes à la technologie et qui permette l'utilisation de notions comme le parallélisme et la synchronisation des processus.

Nous avons choisi comme langage de base le langage PASCAL, pour plusieurs raisons:

- l'étude parallèle de la machine PASCAL nous a permis d'acquérir une bonne connaissance de ce langage,
- la simplicité de la syntaxe de PASCAL,
- l'existence dans PASCAL des structures conditionnelles relativement adaptées à la description des données.

Le langage de description, appelé LD , est développé dans cette étude: sa définition sera élaborée progressivement, ses principales caractéristiques étant définies dans le chapitre II.

Un système d'assistance est également proposé qui fournit à l'architecte les outils suivants:

- outils d'évaluation: chaque niveau d'interprétation étant spécialisé dans l'émulation des niveaux précédents, des mesures dynamiques doivent être effectuées dont les résultats guident l'architecte dans ses choix ;

- outils d'édition: le passage d'un niveau d'interprétation au niveau suivant, développé au chapitre II, consiste en un choix d'un nouveau langage et de son interprétation. L'algorithme d'interprétation I_i, écrit dans le langage LD ($I_{i/LD}$) est implanté comme un programme d'un nouveau langage L_{i+1} : il devient, après des modifications et un codage, un programme $I_{i/L_{i+1}}$, et un nouvel algorithme d'interprétation $I_{i+1/LD}$ est écrit. Les modifications de ces programmes sont rendues plus aisées par l'utilisation d'un système d'édition présenté au chapitre II.

I/3. ORGANISATION DES NIVEAUX DE MEMORISATION.

Au cours de la démarche, des "êtres" sont définis qui vont constituer les données et les variables manipulées par les différentes machines.

Ces êtres apparaissent à un niveau d'interprétation donné:

- les fichiers au niveau des utilisateurs,
- les variables au niveau du langage de haut niveau,
- un compteur ordinal et des variables de travail au niveau de chaque interpréteur...

L'architecte dispose d'autre part d'une hiérarchie de mémorisation:

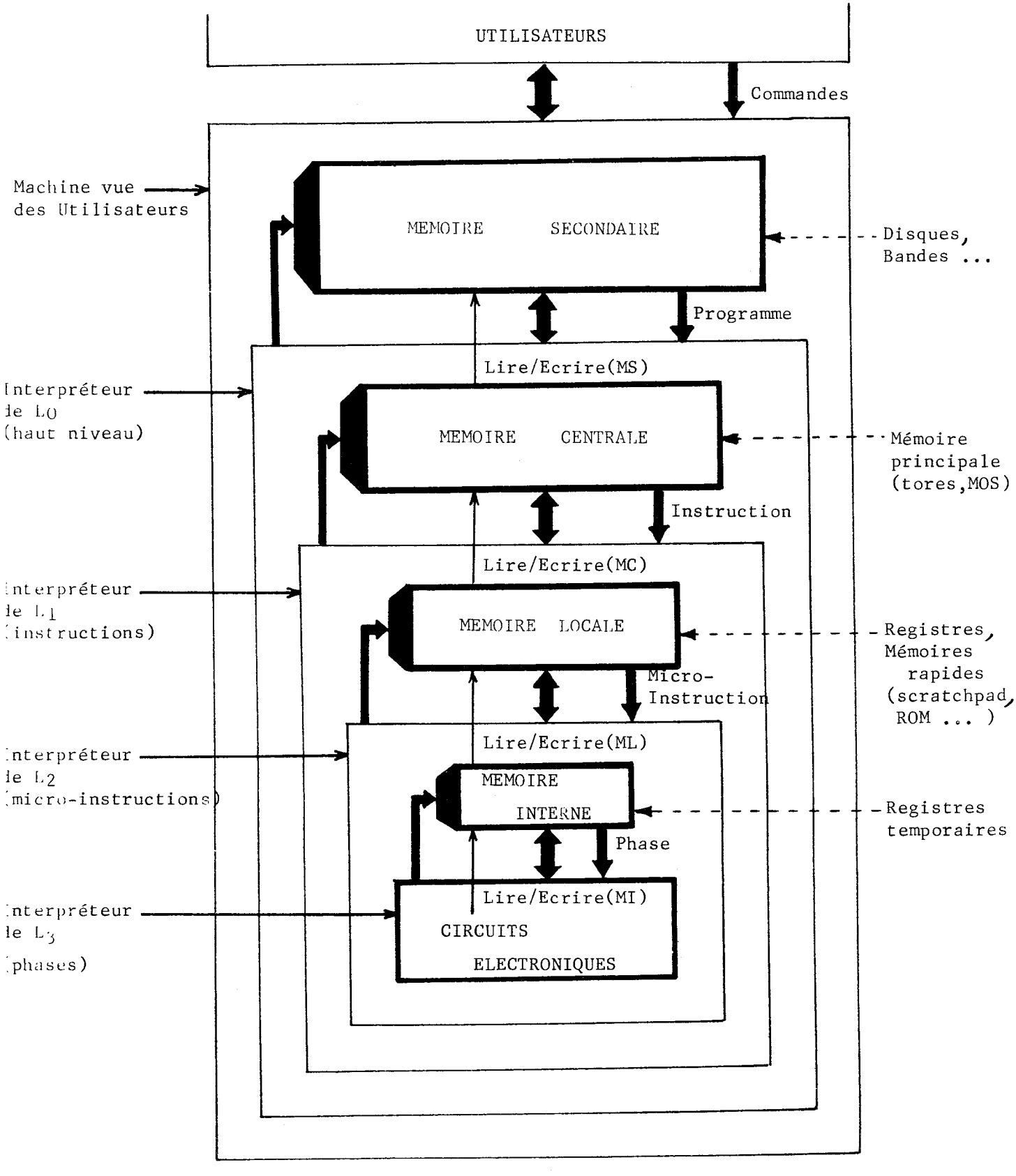
- une mémoire dite secondaire (disques, bandes...),
- une mémoire dite centrale,
- une (ou des) mémoire(s) dite(s) locale(s) et des registres,
- des registres temporaires (LATCHES).

Une approche descendante permet de choisir l'organisation d'un niveau de mémorisation de manière à ce qu'il soit adapté au niveau d'interprétation dans lequel apparaissent les êtres mémorisés.

Une telle heuristique permet d'assurer:

- une bonne relation entre la fréquence d'accès à un "être" mémorisé et la rapidité de cet accès,
- une organisation des éléments de mémorisation qui garantisse une bonne utilisation des ressources de la machine (chemin de données adapté au cheminement des données par exemple).

ORGANISATION HIERARCHISEE DES NIVEAUX DE MEMORISATION



II - DESCRIPTION D'UNE ETAPE ENTRE DEUX NIVEAUX D'INTERPRETATION SUCCESSIFS

Au début de chacune des étapes de la méthode descendante, le concepteur dispose d'une description formelle de l'algorithme dont il doit choisir une réalisation. Ce sera, dans le cas d'une hiérarchie d'interprétation, la description formelle de l'algorithme d'interprétation d'un langage. Soit L_i ce langage.

Comme nous l'avons proposé plus haut, cette description formelle pourra être écrite dans un Langage de Description (LD) commun à toutes les étapes et support de la méthodologie.

Soit donc $I_{i/LD}$ l'interpréteur du langage L_i décrit dans le formalisme du langage LD .

Le but du concepteur est de définir une manière de réaliser l'algorithme décrit par $I_{i/LD}$.

Deux cas sont possibles:

- soit $I_{i/LD}$ est d'une complexité trop grande, dans quel cas sa réalisation est rendue indirecte: on cherche à définir un ensemble de primitives dont l'interprétation sera moins complexe (donc plus réalisable), pour exprimer l'algorithme donné.
- soit I_i est directement "réalisable" (compte tenu de critères économiques) par un assemblage de circuits électroniques: chaque primitive du langage L_i peut être effectivement réalisée par un transfert d'informations physiques dans des circuits électroniques (entre deux impulsions d'horloge). Dans ce cas le but est atteint.

Nous nous plaçons dans le premier cas:

- le travail du concepteur consiste à définir une "machine" spécialisée dans la réalisation de l'algorithme donné (par exemple l'algorithme d'interprétation d'un langage de programmation L_i).

Concevoir une "machine" spécialisée suppose une connaissance aussi précise que possible du "travail" qu'elle doit réaliser. On risque en effet de faire fausse route si le problème à résoudre est mal connu. Il est donc nécessaire de développer une méthode d'investigation pour appréhender les paramètres du problème donné.

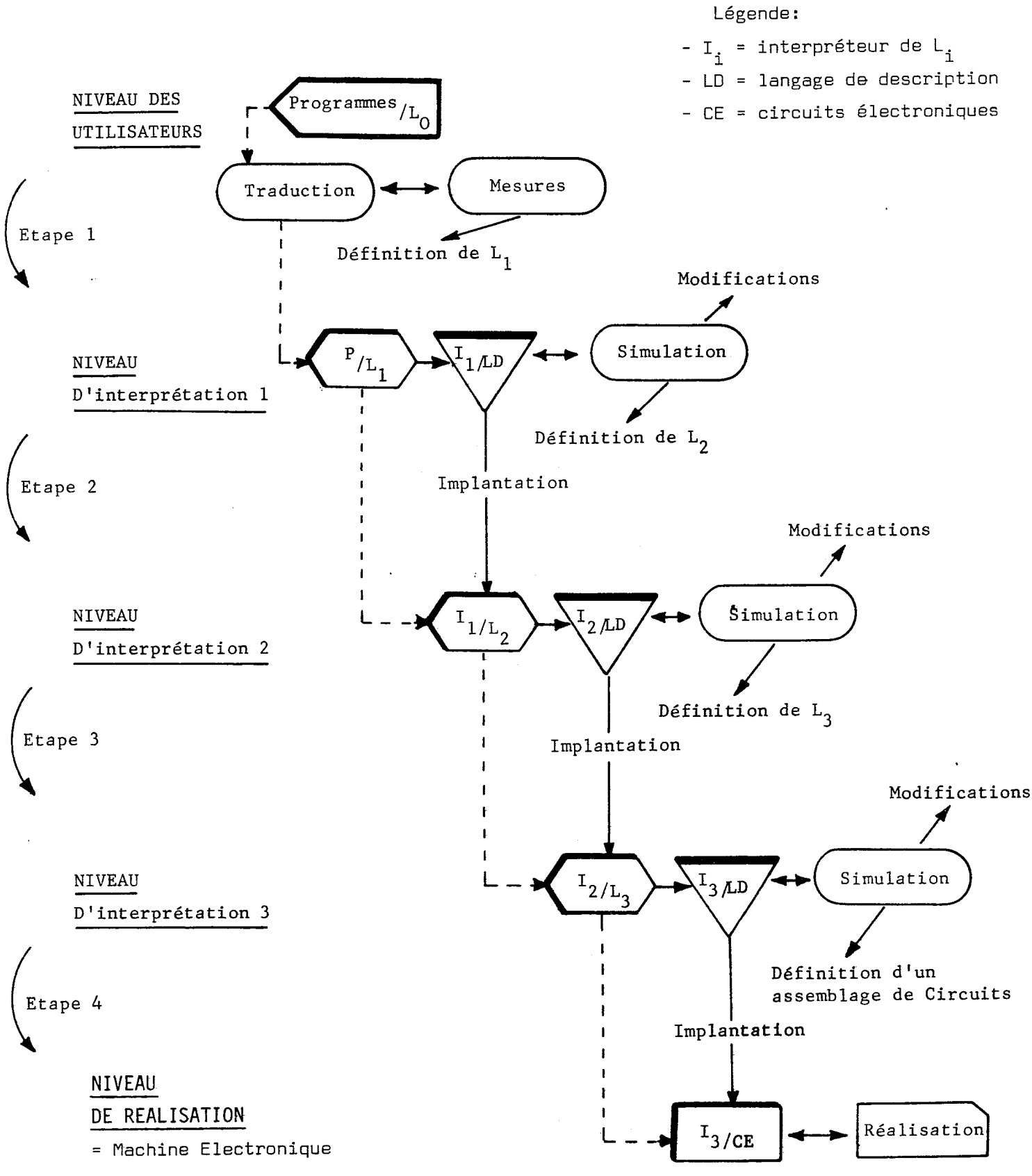
Nous avons choisi, pour ce faire, une heuristique basée sur des mesures dynamiques, réalisées sur un modèle du problème:

- le modèle est constitué du programme $I_{i/LD}$.

Nous considérons que cette description constitue un modèle représentatif du problème et que sa simulation fournit une bonne approximation du problème dans le contexte de sa réalisation dynamique future.

- $I_{i/LD}$ a pour données P_0 (fourni par l'utilisateur), I_1, ..., I_{i-1}.

- PRESENTATION SCHEMATIQUE DE LA DEMARCHE DESCENDANTE.
- HIERARCHIE DES NIVEAUX D'INTERPRETATION.



II/1. EVALUATION DE L'ALGORITHME A "REALISER"

Soit I_i la description, dans le formalisme de LD , de l'algorithme à "réaliser" (en général un algorithme d'interprétation).

Quand on étudie un algorithme quelconque de traitement de l'information, on s'intéresse à ses caractéristiques dynamiques, ou son "comportement" pendant son exécution. En particulier dans le cas qui nous intéresse, les caractéristiques dynamiques de l'algorithme vont conditionner sa réalisation:

il est donc important de pouvoir les mesurer.

Nous nous trouvons dans une organisation hiérarchisée (voir figure 3): au début de l'étape n°i, les niveaux d'interprétation n°1 à n°(i-1) ont été précédemment définis et réalisés. On peut donc simuler l'interprétation de programmes-type du niveau de départ, et mesurer le comportement de I_i au cours de cette simulation.

Un programme de description I_i est constitué d'une liste d'instructions du langage LD . Le but de l'évaluation est de connaître la fréquence dynamique de chacune de ses instructions, au cours de l'interprétation des programmes-type.

Un programme en LD est constitué de deux types d'instructions:

1/ les instructions de séquencement (IS) définissent l'ordre dans lequel les instructions du programme doivent être exécutées,

2/ les instructions opératives (IØ) définissent les opérations à faire sur les valeurs des variables du programme.

II/1.1. Séquence indivisible

On appelle Séquence Indivisible d'Instructions Opératives (et on notera SIIØ) une séquence d'instructions $(I_i;\ldots;I_n)$ opératives où:

- seule la première instruction (I_1) peut être étiquetée,

- le séquencement entre I_j et I_{j+1} , pour j=1 à j=n-1 , est implicite (I_{j+1} est <u>toujours</u> exécutée après I_j).
- l'instruction qui suit (dans la liste du programme) la dernière instruction opérative (I_n) n'est pas une instruction opérative.

Avec la définition précédente d'une SIIØ , toutes les instructions opératives qui la composent ont le même nombre d'exécutions.

Il suffit donc de compter le nombre des entrées dans une telle séquence.

II/1.2. <u>Séquence contrôlée</u>

Une séquence d'instructions dont l'exécution est contrôlée par une instruction de séquencement a un nombre d'exécutions qui dépend:

- du nombre des exécutions de l'instruction de séquencement,
- des valeurs dynamiques des prédicats évalués par l'instruction de séquencement.

Il faut donc compter

1/ le nombre des exécutions des instructions de contrôle,
2/ le nombre des entrées dans les différentes séquences contrôlées par l'instruction de séquencement.

<u>Remarque</u>: Si une instruction de séquencement contrôle p séquences, il suffit de compter les entrées dans p-1 séquences (gain de 1 compteur sur 2 pour le IF-THEN-ELSE).

Une séquence contrôlée par une IS commence soit par une SIIØ , soit par une IS .

→ il faut (et il suffit de) mettre des points de comptage
- sur chaque instruction de séquencement,
- sur chaque IØ commençant par une SIIØ ,

pour connaître le nombre d'exécutions de toutes les instructions du programme.

II/1.3. Génération des instructions de comptage

L'insertion automatique des instructions de comptage lors de la compilation du programme de description a été envisagée, mais elle présente les inconvénients des solutions automatiques: l'utilisateur désire souvent avoir un comptage partiel et peu coûteux.

Nous avons donc choisi la solution manuelle:

les points de comptage sont marqués par l'utilisateur dans le programme sous la forme d'un commentaire spécial:

/ * $ $ i * /

où i indique le numéro d'un compteur.

Le compilateur génère des instructions correspondant à l'incrémentation du compteur numéro i .

L'impression de la valeur des compteurs est laissée à la charge de l'utilisateur en fin de programme.

Cette phase de simulation est importante. Elle présente deux aspects:

- elle permet de valider (ou au contraire de remettre en cause) certains choix effectués dans les étapes précédentes.
- elle fournit une première approximation de ce qui est fréquent (donc important) dans l'algorithme: ces mesures seront un paramètre important pour la suite de la démarche.

II/2. CHOIX DES OPERATIONS SEMANTIQUES "PRIMITIVES"

Le concepteur dispose, à cette étape, d'une description de son algorithme, dans le formalisme du langage LD.

II/2.1. Influence du formalisme utilisé

La description n'est qu'un modèle de l'algorithme.

L'utilisation d'un tel formalisme permet

- de concrétiser des notions abstraites (opération indispensable pour la communication à l'intérieur d'une équipe de conception) ;
- de donner une définition non ambigüe de la sémantique du problème ;
- de simuler le problème avant d'en choisir une réalisation.

Cependant, il est nécessaire de distinguer, dans cette méthode, quelle est la part du formalisme utilisé pour la description, et quelle est celle de la sémantique propre du problème.

Nous pensons qu'il faut isoler la notion de variables de celle d'opérations au profit de la première notion. En effet, tout algorithme est caractérisé par les "êtres" qu'il manipule: c'est la sémantique de chacun de ces "êtres" qui conditionne les opérations à réaliser.

En conséquence, il apparaît que, indépendamment de toute représentation ou de tout formalisme, nous devons nous intéresser en premier lieu aux "êtres" manipulés, qui seront les variables référencées dans la description de l'algorithme, afin de déterminer quelles sont les opérations sémantiques réellement importantes à réaliser sur ces "êtres".

Le formalisme du Langage de Description sera considéré comme intermédiaire (un support), la sémantique des variables sera étudiée en tant que principal mobile du choix d'une réalisation.

Le programme $I_{i/LD}$ décrit une séquence finie d'opérations à effectuer, dans un ordre déterminé, sur un ensemble d'êtres abstraits que nous appellerons "environnement" ou "espace de noms" de l'algorithme. Cet "environnement" est décrit sous la forme d'un ensemble de variables déclarées dans $I_{i/LD}$. (Nous appellerons également ces variables, les "variables de l'interpréteur I_i").

II/2.2. Destruction du formalisme

Le concepteur doit choisir, guidé par les résultats des mesures de la phase d'évaluation, et par la puissance qu'il désire donner à sa réalisation de l'algorithme, un découpage du programme de description $I_{i/LD}$, de manière à faire apparaître ce qu'il considère comme devant être des opérations primitives. Ce découpage peut impliquer une modification du formalisme initial.

Un système d'édition est mis à la disposition du concepteur pour lui permettre de modifier éventuellement le texte initial ($I_{i/LD}$) et de le découper en une suite de chaînes de caractères.

En effet, seul le concepteur connaît la sémantique des opérations qui constituent $I_{i/LD}$: lui seul est capable de pondérer la réalisation de son algorithme en choisissant, dans son texte, les chaînes de caractères qu'il désire considérer comme des NOMS d'opérations primitives.

II/2.3. Découpage du programme de description de l'algorithme

Si l'on fait provisoirement abstraction de la partie déclarative (déclaration de variables), et que l'on s'intéresse au corps du programme constitué d'une séquence d'opérations sur les variables déclarées, on peut considérer chaque Caractère Syntaxique du programme comme faisant partie d'une opération dont la sémantique est définie par le langage LD.

exemple: soit la chaîne de caractères "CØ := CØ+1".

Les caractères syntaxiques en sont :

"CØ", ":=", "CØ", "+" et "1".

D'après la sémantique de LD, le sens de cette suite de caractères syntaxiques est le suivant:

"ajouter 1 à la valeur de la variable de nom CØ et affecter cette nouvelle valeur à la même variable".

Pour le concepteur, cette chaîne de caractères possède une sémantique plus abstraite:

"incrémenter le compteur ordinal (symbolisé par CØ) de la valeur 1".

Le problème de conception est alors le suivant:

- doit-on considérer la chaîne "CØ := CØ+1" comme étant le NOM d'une primitive?
- ou bien, au contraire, doit-on considérer que cette chaîne symbolise une sémantique trop riche et qu'elle doit être décomposée en plusieurs sous-chaînes à sémantique plus pauvre?

Dans ce dernier cas, on décomposera la chaîne, par exemple, de la manière suivante

"CØ := [CØ+1]"

- où "CØ+1" est une 1ère primitive (ajouter 1 à la valeur de CØ),
- et "CØ:=[]" est une 2ème primitive (affecter une valeur à CØ).

REMARQUE IMPORTANTE:

On pourrait envisager d'introduire, à ce niveau de découpage, de nouvelles variables:

exemple: "CØ := [CØ+1]" deviendrait "α := CØ+1";
"CØ := α".

Nous préfèrerons, dans une première approche, une notation plus abstraite qui ne fait pas encore appel à une quelconque réalisation des primitives.

exemple: "A := M(CØ) + B" pourra être découpé en

A := [[M(CØ)] +[B]] ,

forme équivalente à:

"α_1 := M(CØ)" ; "α_2 := B" ;

" β := α_1 + α_2" ;

"A := β" ;

II/2.4. Critères de découpage

Le choix de PRIMITIVES peut être concrétisé par un marquage du programme initial. Ce choix est guidé par les mesures effectuées précédemment sur ce programme: le concepteur connaît le nombre des occurrences dynamiques associé à chaque "chaîne de caractères".

La règle suivante peut être appliquée:

> On pourra, dans un premier temps, considérer comme opération PRIMITIVE (essentielle) toute "chaîne" ayant un nombre d'occurrences dynamiques élevé.

D'autre part, la richesse sémantique d'une chaîne peut impliquer un découpage: si le concepteur juge que la chaîne est trop riche sémantiquement par raport à la puissance qu'il désire donner à sa réalisation, il va chercher à définir des primitives plus élémentaires.

II/2.5. Introduction du parallélisme dans les primitives

Le regroupement de plusieurs "instructions" du programme initial peut être choisi par le concepteur pour constituer une primitive.

exemple: "A := 1 et B := C"

Nous devons prévoir la possibilité de déterminer, dans une primitive, un ordre de réalisation des différentes opérations sémantiques qui la constituent.

Ordre implicite défini par LD :

Dans le Langage de Description, les expressions sont évaluées de gauche à droite, en fonction des priorités des opérateurs. L'affectation est postérieure à l'évaluation.

Cet ordre implicite peut être contraignant et même inutile étant donné qu'il n'y a pas d'effets de bord possibles, nous proposons donc de donner la possibilité de le remplacer par un ordre explicite dans certains cas, en introduisant les notions de parallélismes synchrone et collatéral, ainsi que de séquentialité explicite.

Une instruction opérative définit des opérations à faire sur les valeurs d'un sous-ensemble des variables du programme.

Elle se présente sous la forme

<variable> ':=' <expression>,

Nous admettrons que l'exécution de toute instruction opérative se décompose en:

1/ évaluation de l'<expression>,

2/ affectation de la valeur de l'<expression> à la <variable>.

remarque:

Cette décomposition se retrouve à tous les niveaux d'interprétation, jusqu'au niveau technologique:

1/ évaluation = établissement d'un circuit combinatoire,

2/ affectation = écriture dans un élément de mémorisation.

II/2.5.1. les affectations multiples

Dans le cas où on affecte la valeur de l'<expression> à plusieurs variables, et que l'ordre des affectations est indifférent, on peut parler de parallélisme collatéral.

Cependant, on peut être conduit à décrire un parallélisme synchrone, toutes les affectations étant dans ce cas réalisées au même instant.

On utilisera donc les symbolismes suivants:

- affectation multiple collatérale:
  <variable>{ ','<variable>}$^{+}$ ':=' <expression>
- affectation multiple synchrone:
  <variable>{ ',,' <variable>}$^{+}$ ':=' <expression>

(+ = au moins une fois).

exemples:

parallélisme collatéral: A,B := AD-1 ;
parallélisme synchrone: AD,, RAM := AD-1 ;

remarque 1:

L'ordre dans lequel les <variables> apparaissent dans la <liste de variables> est indifférent, que ce soit pour le parallélisme collatéral ou pour le parallélisme synchrone. Le système d'édition doit donc être capable de reconnaître que les chaînes "A,B := CØ+1" et "B,A := CØ+1" sont équivalentes.

remarque 2:

Le parallélisme synchrone des affectations multiples, placé dans un contexte temporel, fait apparaître la notion d'actions simultanées.

Par contre, le parallélisme collatéral suppose que la valeur de l'<expression> soit disponible pendant un certain temps pour permettre l'exécution des différentes affectations séquentiellement.

II/2.5.2. séquence d'affectations collatérales

Le parallélisme collatéral envisagé porte sur les opérations d'affectation, et pas sur l'évaluation des <expressions>.

Syntaxe:

<variable> ':=' <expression> {',' <variable> ':=' <expression>}$^{+}$

(+ = une fois au moins)

Notations:

* une instruction opérative est notée:

$I\emptyset$ = "Y := E(X_1, X_2, ..., X_p)"

où Y, X_1,....., X_p sont des noms de variables.

* une séquence d'affectations collatérales est notée

$I\emptyset_1$','$I\emptyset_2$',' ... ','$I\emptyset_n$ avec n >= 2

avec $I\emptyset_1$ = "Y_1 = E_1(X_2^1, X_2^1,..., $X_{p_1}^1$)"

⋮

$I\emptyset_n$ = "Y_n = E_n(X_1^n, X_2^n,..., $X_{p_n}^n$)".

Conditions de validité

Pour qu'une séquence d'affectations collatérales soit valide, il faut que les deux propriétés suivantes ne soient pas vérifiées.

Propriété 1:

S'il existe i et j, (i≠j) tels que Y_i=Y_j, alors le résultat de la séquence d'affectations collatérales dépend de l'ordre des affectations.

Propriété 2:

S'il existe i, j, k tels que i≠k et X_j^k=Y_i , alors la valeur affectée à Y_k dépend de l'ordre dans lequel $I\emptyset_i$ et $I\emptyset_k$ sont exécutés.

exemple: A := CØ+1 , B := A+B

Méthode de vérification

Au cours de la compilation d'une telle séquence du programme de description, les Y_i et les X_j^k sont notés dans deux tables distinctes. Si, pour la compilation de $I\emptyset_i$, il existe j tel que X_j^i=Y_i, alors on ne note pas X_j^i dans la table des X.

En fin de séquence d'affectations collatérales on vérifie que:

1/ tous les Y_i sont différents, sinon la propriété 1 est vérifiée,

2/ l'intersection des 2 tables est vide, sinon la propriété 2 est vérifiée.

II/2.5.3 séquence d'affectations synchrones

Le parallélisme synchrone envisagé porte sur les opérations d'affectations, et pas sur les évaluations des expressions .

Syntaxe:

<variable> ':=' <expression>{',,'<variable>':='<expression>}$^+$

(+ = au moins une fois)

Notations (comme en II/2.5.2.)

Une séquence d'affectations synchrones est notée

$I\emptyset_1$',,'$I\emptyset_2$',,' ... ',,'$I\emptyset_n$ avec n >= 2

Condition nécessaire de validité

Tous les Y_i(i=1 à n) doivent être différents.

Cette condition est facile à vérifier à la compilation: pour chaque séquence d'affectations synchrones, on note les Y_i dans une table. En fin de séquence, on vérifie que tous les éléments de la table sont différents.

Remarque:

La condition de validité précédente peut être étendue au cas où deux variables Y_i et Y_j partagent la même méthode d'accès.

Exemple:

Y_i= T(1) et Y_j= T(2) ; Y_i et Y_j sont deux éléments du même tableau T à accès unique. On notera dans ce cas le nom T($) dans la table et on découvrira ainsi que la condition n'est pas vérifiée.

II/2.5.4. séquence d'affectations collatérales et affectations multiples

Soit la séquence $IØ_1$','$IØ_2$',' ... ','$IØ_n$ avec n >= 2

avec $IØ_1$ = "Z_1 := $E_1(X_1^1, \ldots, X_{p_1}^1)$"

⋮

$IØ_n$ = "Z_n := $E_n(X_1^n, \ldots, X_{p_n}^n)$"

1er cas:

Il existe i tel que Z_i = Y_1',,'Y_2',,' ... ',,'Y_{n_i} avec n_i >= 2 .

On vérifie comme dans 2/ les conditions de validité.

2ème cas:

Il existe i tel que Z_i = Y_1','Y_2',' ... ','Y_{n_i} avec n_i >= 2 .

On vérifie comme dans 2/ les conditions de validité.

II/2.5.5. séquence d'affectations synchrones et affectations multiples

Soit la séquence $IØ_1$',,'$IØ_2$',,' ... ',,'$IØ_n$ avec n >= 2

1er cas:

Il existe i tel que Z_i = Y_1',,'Y_2',,' ... ',,'Y_{n_i} avec n_i >= 2 .

On vérifie comme dans 3/ les conditions de validité.

2ème cas:

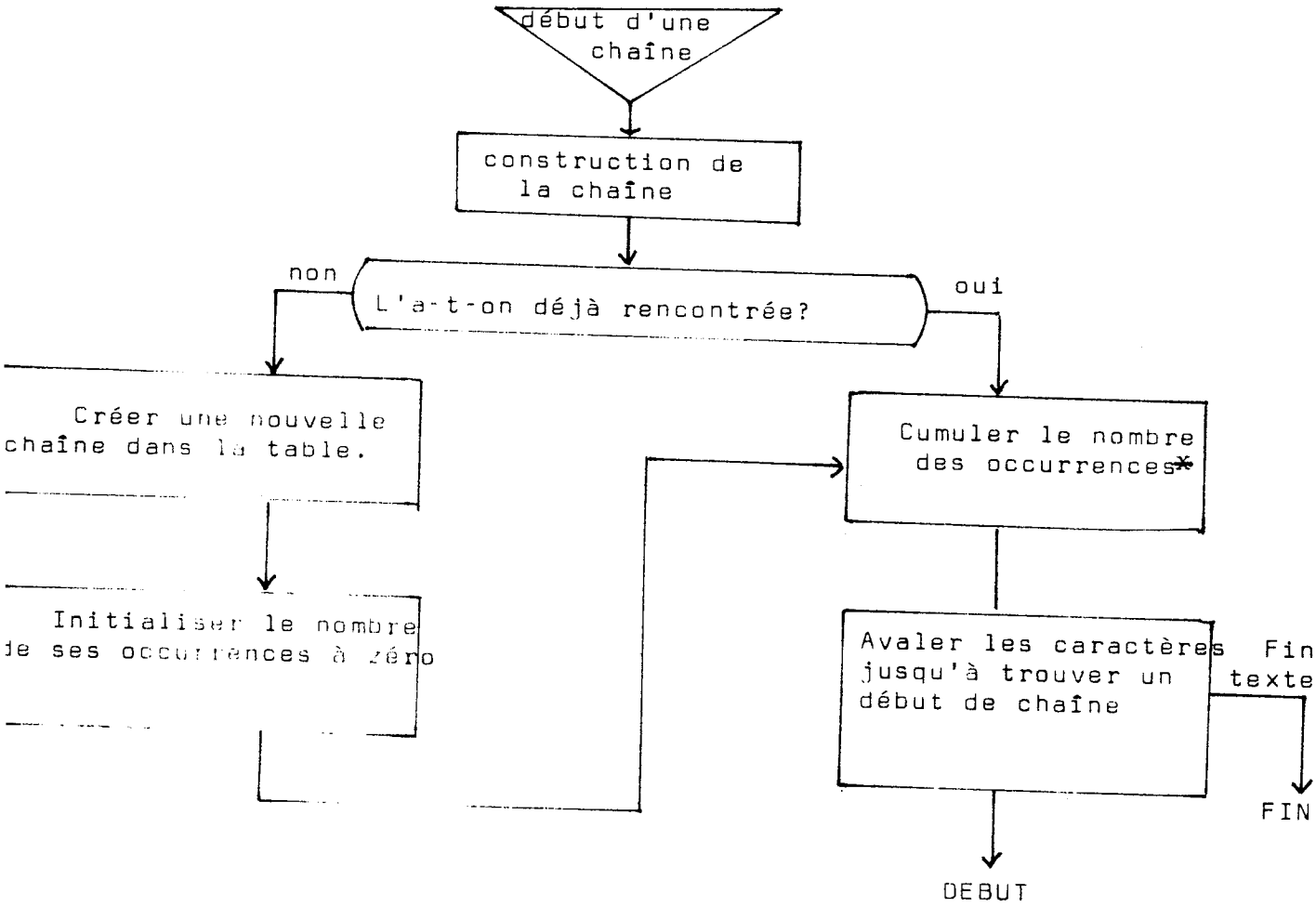
Il existe i tel que Z_i = Y_1','Y_2',' ... ','Y_{n_i} avec n_i >= 2 .

Ce cas présente une ambigüité sémantique: on mélange en effet le parallélisme synchrone des instructions opératives et le parallélisme collatéral dans une affectation multiple. On interdira donc ce cas.

II/3. ANALYSE DU TEXTE DECOUPE EN CHAINES

Le texte initial ayant été découpé en Chaînes de Caractères à l'aide d'un outil d'édition, le système d'assistance peut analyser ces chaînes et fournir un premier diagnostic:

ORGANIGRAMME



* Ce nombre d'occurrences a été mesuré dans la phase d'évaluation (§ II/1.)

II/4. CHOIX D'UN SOUS-ENSEMBLE DE PRIMITIVES

Le nombre de chaînes différentes définies dans la phase précédente est en général assez grand.

Le travail du concepteur va maintenant consister en une réduction du nombre de primitives différentes. Cette réduction peut être faite:

1°/ en exprimant une primitive ayant un faible nombre d'occurrences à l'aide de primitives existantes qui ont déjà un nombre important d'occurrences,

2°/ en regroupant plusieurs primitives en une seule, par l'introduction de un ou plusieurs paramètres.

Dans les deux cas de réduction, le nombre des occurrences est cumulé, et le nombre total de chaînes différentes est réduit.

exemple 1:

soit la primitive "while <expression> do goto i"
dont le nombre d'occurrences est faible.
On l'exprimera en termes de la primitive
"if <expression> then goto i else goto j"
dont le nombre d'occurrences est élevé.

exemple 2:

soient les primitives "A := B+C" et "A := B-C"
On introduit un paramètre pouvant prendre les valeurs + et - et la nouvelle primitive est notée

"A := B $1 C"($1 :(+,-))

Alors "A := B+C" devient "A := B $1 C"(+),
et "A := B-C" devient "A := B $1 C"(-).

Après cette phase de réduction, le concepteur obtient un nouvel ensemble de primitives, éventuellement paramétrées, dont les noms sont des chaînes de caractères.

Le travail de conception est là encore guidé par les fréquences des primitives, par leur nombre et par leur contenu sémantique. Le système d'assistance est conçu pour guider le concepteur dans cette pondération.

II/5. TRANSFORMATION DU TEXTE INITIAL

Un sous-ensemble de primitives ayant été choisi, le texte initial $I_{i/LD}$ peut être exprimé comme une liste de ces primitives: c'est un programme appartenant au nouveau langage L_{i+1} dont on vient de définir les "instructions".

Le système d'assistance, tenant compte des opérations de réduction précédentes, peut exprimer le texte initial comme une liste de primitives.

exemple:

Le concepteur a, dans un premier temps, isolé la chaîne "A := A+1".
Dans un deuxième temps, il a réduit le nombre de primitives et choisi la primitive "$1 := $1 $2 1"(A,+) pour exprimer "A := A+1".
On pourra donc construire un nouveau texte.

Texte initial	Nouveau texte
⋮	⋮
"A := A+1" ;	"$1 := $1 $2 1"(A,+) ;
⋮	⋮
"A := B+C" ;	"A := B $1 C"(+) ;
⋮	⋮

Le nouveau texte est maintenant constitué d'une suite d'appels de procédures. Les noms de ces procédures sont des chaînes de caractères, éventuellement paramétrées, les paramètres en sont des symboles (noms de variables ou d'opérateurs).

II/6. DESCRIPTION DES PRIMITIVES

La sémantique des primitives choisies est abstraite: elle est relative à l'environnement (espace de noms) précédent dans la hiérarchie d'interprétation.

exemple:

> Soient les primitives "A := A+1", "B := B+1", "C := C+1" qui ont été regroupées en "$1 := $1+1"(A), "$1 := $1+1"(B), "$1 := $1+1"(C). On n'a plus qu'une seule primitive dont le paramètre $1 peut prendre les "valeurs" symboliques A, B ou C.

NOTATION: procédure "$1 := $1+1"($1 : (A, B, C)) ;

Les paramètres des primitives (symbolisées par des procédures de LD) sont des NOMS de variables appartenant à l'espace de noms précédents.

La description d'une réalisation des primitives suppose que les noms de l'environnement i deviennent des noms de l'environnement courant i+1: un lien doit être établi entre les variables abstraites du niveau i et de nouvelles variables du niveau i+1. Ce lien est appelé l'IMPLANTATION.

Regroupement de plusieurs variables de l'environnement i dans une seule variable de l'environnement i+1.

La complexité d'une machine est une fonction croissante du nombre de variables qui constituent son environnement (cardinal de son espace de noms). La recherche de l'économie et de la simplicité se traduit par une réduction de ce nombre, qui peut être obtenue par des regroupements.

exemple:

> Les registres généraux de la Machine 360 sont au nombre de 16 pour le programmeur. Cependant la plupart des micro-machines 360 ne possèdent qu'une variable structurée de 16 éléments ou plus (appelée Mémoire locale ou Scratchpad).

II/6.1.1. Critères fonctionnels de regroupement

Dans les phases précédentes, le concepteur a défini un certain nombre de "primitives" paramétrées par des noms de variables de l'environnement i .

exemple:

"A := A+X"		"$1 := $1+$2"(A, X) ;
"B := B+Y"	→	"$1 := $1+$2"(B, Y) ;
"C := C+Z"		"$1 := $1+$2"(C, Z) ;
"D := D+T"		"$1 := $1+$2"(D, T) ;

La primitive précédente peut être décrite par une déclaration de procédure:

procédure "$1 := $1+$2"($1 : (A, B, C, D) ; $2 : (X, Y, Z, T)) ;

où

"$1 := $1+$2" est le nom de la primitive,
$1 peut prendre comme "valeurs" les noms A, B, C ou D,
$2 peut prendre comme "valeurs" les noms X, Y, Z ou T.

Nous disons que, dans l'exemple précédent, les variables A, B, C, D d'une part, les variables X, Y, Z, T d'autre part, sont fonctionnellement équivalentes pour la primitive "$1 := $1+$2".

exemple emprunté au HP/3000:

Les variables de la micro-machine HP/3000 sont regroupées en trois tableaux différents:

- le 1° contient X, Z, PL, SP_0,
- le 2° contient TR_0, ..., TR_3(sommet de la pile),
- le 3° contient P, PB, DB, DL, SM, 0, STATUS, CPX_1, CPX_2, SP_2, SP_3, OPND.

Cette répartition est essentiellement fonctionnelle: les opérandes des primitives de la micro-machine HP/3000 sont implantés dans des tableaux différents pour simplifier l'exécution des opérations binaires et des opérations sur la pile:

Opérations entre: X et P, X et DB, X et Q...

Comparaisons entre: Z et SM, PL et P

Opération sur pile entre: TR_i et OPND.

II/6.1.2 contraintes dues au regroupement

On peut être amené à choisir un regroupement de variables de types différents dans un même tableau.

Soit α_1 de type t_1, ..., α_n de type t_n qu'on souhaite regrouper dans un tableau X . Quel doit être le type t des éléments de X sachant que les types $t_1 \ldots t_n$ sont tous différents?

La première solution est empruntée au langage PASCAL: le type t est construit comme étant l'UNION des types $t_1 \ldots t_n$, de la manière suivante:

```
t = record case PREFIXE: (v1, ..., vn) of
        v1: (C1 : t1) ;
        :
        vn: (CN : tn)
    end
```

On peut alors déclarer le tableau X par

X : array $(\alpha_1, \ldots, \alpha_n)$ of t

et la valeur de $X(\alpha_i)$. PREFIXE, qui est égale à v_i, indique que l'élément de nom α_i est de type t_i.

L'inconvénient du formalisme de PASCAL réside dans le fait que des noms différents doivent être affectés aux éléments du tableau, suivant le type auquel ils appartiennent.

exemple:

la variable α_i doit être référencée par $X(\alpha_i).C_i$

la variable α_j doit être référencée par $X(\alpha_j).C_j$

Nous préférerons à la structuration précédente, la suivante:

```
t = record  PREFIXE: (v1, ..., vn) ;
    INFØ : case PREFIXE of
              v1: t1 ;
                 ⋮
              vn : tn
    end
```

II/6.1.3. ordres d'implantation par regroupement

Le concepteur ayant choisi, selon des critères fonctionnels ou économiques, de regrouper plusieurs variables dans un même tableau, il doit établir un lien, au niveau du Langage de Description, entre l'environnement i et l'environnement i+1.

Nous proposons de mettre à sa disposition des ordres d'implantation.

exemple:

```
Implante α1, α2, ..., αn  dans  X: array (α1, α2, ..., αn)  of
          record  PREFIXE: (v1, ..., vp) ;
                  INFØ: case PREFIXE of
                          v1:  <type 1>  ;
                             ⋮
                          vp:  <type p>
          end
```

Le système d'assitance peut alors vérifier que pour tout α_i ($i \in \{1,\ldots,n\}$) il existe $j \in \{1, \ldots, p\}$ tel que <type j> soit le type de α_i.

A la fin des ordres d'implantation, le système vérifie également que toutes les variables de l'environnement i ont une correspondance dans le nouvel environnement i+1.

II/6.1.4. implantation des tableaux.

On retrouve, à tous les niveaux de la conception, l'utilisation de tableaux pour des raisons de simplification de l'espace de noms propre à un niveau (la micro-machine ne "connaît" que sa mémoire locale, et pas les mots qui la composent, la machine physique ne "connaît" que ses multiplexeurs et pas les fils qui sont en entrée de ces multiplexeurs), et pour des raisons technologiques: l'utilisation de mémoires et de multiplexeurs permet un gain considérable sur le coût des machines.

L'implantation d'un tableau est un choix important pour le concepteur: il s'agit de définir la méthode d'accès à un élément quelconque du tableau. Cela suppose:

1/ de définir une correspondance entre un ensemble de noms (les noms des éléments du tableau) et un ensemble (généralement ordonné) d'emplacements équivalents pouvant "contenir" les éléments du tableau.
C'est le choix de l'ADRESSAGE.

2/ de définir une correspondance entre l'ensemble des valeurs possibles des éléments du tableau et l'ensemble des valeurs que peut "contenir" un emplacement.
C'est le choix du FORMAT.

II/6.1.5. définition d'une ressource, processus parallèles asynchrones.

Un certain nombre d'organes constituant un calculateur peuvent être considérés comme des organes périphériques de l'unité centrale.

De même différents processus software peuvent simuler des phénomènes asynchrones (On peut également être conduit à décrire plusieurs unités centrales travaillant en parallèle).

Les échanges d'informations (données ou évènements) doivent cependant être réalisés d'une manière synchrone: une information ne peut être à la fois lue (par le récepteur) et modifiée (par l'émetteur).
(réf. NOGUEZ).

Dans le cadre de ce travail, nous sommes dans un contexte de simulation: le parallélisme synchrone de plusieurs processus sera simulé par un seul processeur: l'interpréteur du langage de description.

Les communications d'informations entre les processus seront faites par l'intermédiaire de variables dites globales, l'accès simultané à une variable partagée étant bien entendu impossible dans un contexte de simulation. Cependant, les problèmes liés au partage des variables (mutuelle exclusion) seront décrits et simulés.

a) Notion de ressource

Définition générale: on appelle ressource une entité logique capable de réaliser un ensemble de fonctions déterminé, moyennant des conventions de synchronisation et de communication de données déterminées. Une ressource est activée de l'extérieur et réalise les fonctions demandées de manière autonome et dans un temps fini.

Remarque: cette notion de ressource est voisine de celle de moniteur [HS.71].

Description d'une ressource dans le Langage de Description.

Une ressource est décrite par un Programme.

Elle dispose de variables de travail qui lui sont propres, et de variables dites "externes" ou globales, qu'elle partage avec d'autres programmes. Les fonctions internes de la ressource sont décrites comme des procédures (ou des fonctions-procédures).

Structure schématique:

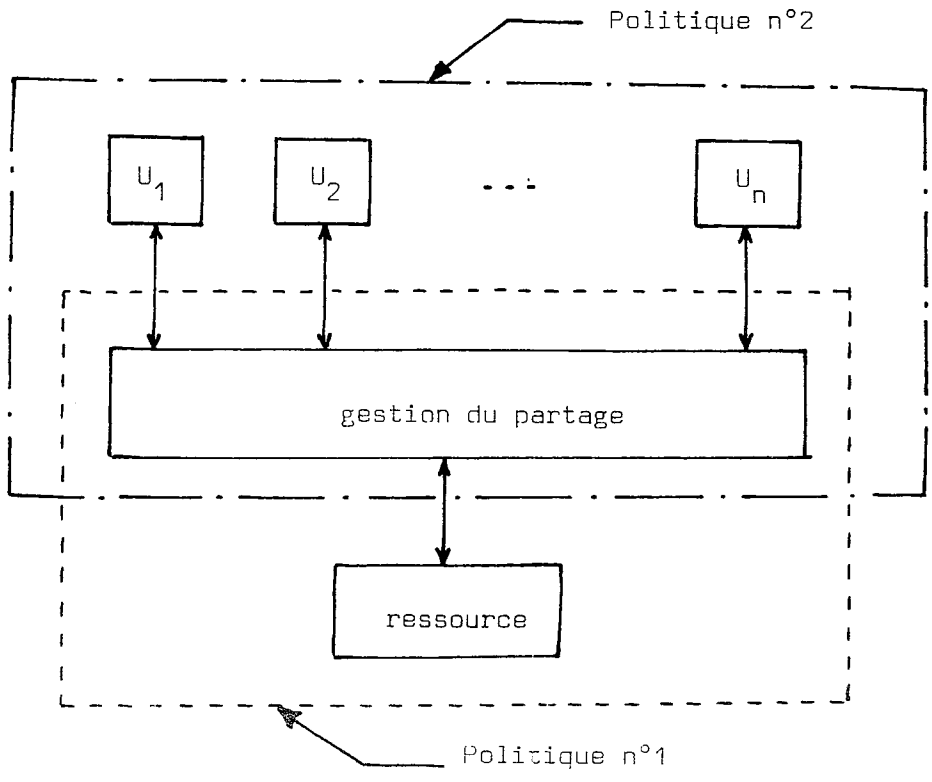
RESSOURCE <nom> {(<liste de paramètres>)} ;
GLOBAL déclaration des variables de communication partagées avec d'autres programmes.
VAR déclaration des variables locales de la ressource.

```
PROCEDURE      <nom> ;
                     ← variables locales à la procédure
     1ère fonction
PROCEDURE      <nom> ;
                     ← variables locales à la procédure
     2° fonction
                     ⋮
BEGIN
        corps du programme (gestion interne de la ressource)
END.
```

Une ressource unique peut être activée par plusieurs programmes se déroulant de manière indépendante. On parlera dans ce cas de ressource partagée et des utilisateurs multiples de cette ressource. Dans ce cas, on pourra choisir deux politiques différentes:

1/ la ressource elle-même gère le partage de son activité entre ses utilisateurs ;

2/ les utilisateurs gèrent entre eux le partage de la ressource. Dans ce dernier cas, la ressource n'a plus qu'un seul utilisateur connu.



Dans tous les cas, la ressource n'a qu'un utilisateur possible à un instant donné. Nous supposerons donc, dans une première approche, que c'est la ressource qui choisit son utilisateur (politique n°1). Dans ces conditions, le travail de fond de la ressource consistera en un choix parmi toutes les demandes d'activation d'une de ses fonctions, selon des critères déterminés (priorité, barillet,....).

Exemple:

Soient n utilisateurs de la même ressource.
La priorité de chaque utilisateur est égale à son numéro (l'utilisateur n°1 a la priorité la plus forte).
L'utilisateur n°i dépose sa demande dans DEM(i).
La travail de sélection d'un utilisateur peut être décrit par :

```
/* sélection*/
  i := 1 ;
loop
exit if  DEM(i) ;          }
   i := (i+1) modulo n     }   choix du demandant le + prioritaire.
end ;

/* l'utilisateur n°i a été sélectionné */
/* sa demande est prise en compte */
        :
DEM(i) := 0 ;      /* la ressource avertit l'utilisateur n°i que
                      son travail est terminé */
```

b) Implantation des ressources dans le Langage de Description

Structure globale

Une description est constituée d'un ensemble de programmes.
Tous ces programmes sont supposés exécutés en parallèle. Ils communiquent entre eux par l'intermédiaire de variables dites variables externes.

```
programme P1 ;      programme P2 ;                 programme Pn
  externe             externe          .....        externe
```

Accès aux variables externes

Un programme peut faire référence à toutes les variables externes qui sont déclarées dans son entête. On suppose qu'il n'y a pas de conflit lors de l'accès à une variable externe par plusieurs programmes.

Déclaration de synonimie

La même variable externe peut être nommée de plusieurs manières différentes par plusieurs programmes.

Supposons que les variables A_1, A_2, ..., A_n soient déclarées dans des programmes différents comme variables externes de même type.

Si dans un autre programme, ces variables doivent être groupées fonctionnellement dans le même tableau, on déclare dans ce programme:

externe A syn A_1, A_2, ..., A_n : array (1...n) of <type> ;

La référence à A(i) dans le programme où A est déclaré est alors équivalente à la référence à A_i dans le programme où A_i est déclaré.

II/7. IMPLANTATION DU PROGRAMME INITIAL, CODAGE DES PRIMITIVES

II/7.1. Interpréteur Procédural

A cette étape, le concepteur dispose:

- d'une part du programme initial qui a été découpé et exprimé comme une liste de primitives éventuellement paramétrées.

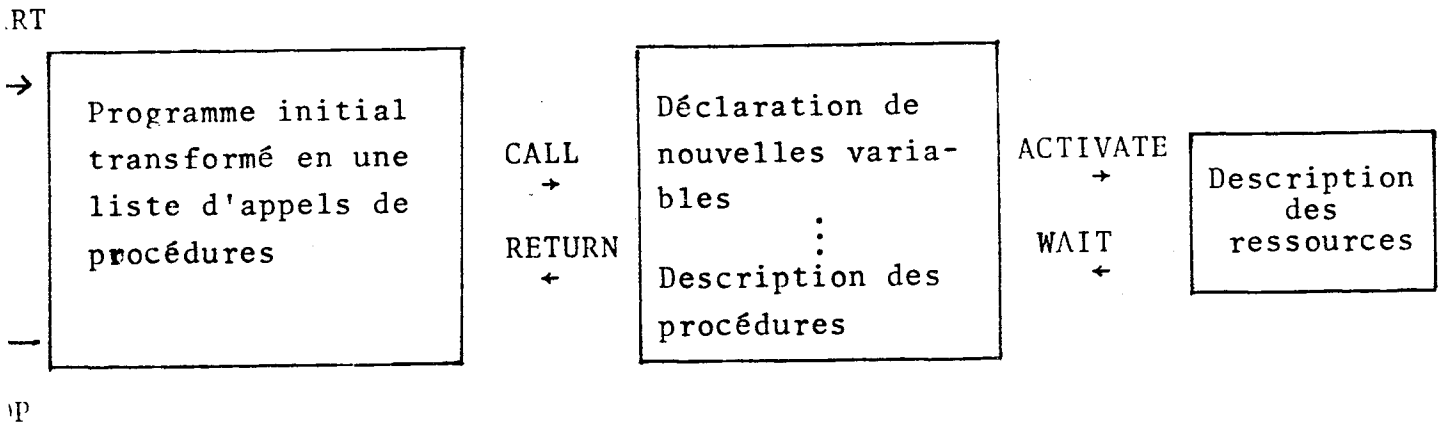
Ce programme se présente maintenant comme une liste d'appels de procédures (paramétrées).

exemple:

Texte initial	Nouveau texte
if A = 0 then goto 10;	"if $1=0 then goto $2"(A,10);
10 : A := A + 1;	10: " $1 := $1+1"(A);

- d'autre part, de la description de chacune des primitives choisies, en fonction du nouvel environnement.

L'ensemble des descriptions de ces primitives et les déclarations des variables du nouvel environnement constitue l'interpréteur procédural du langage de primitives P_{i+1}.



Fonctionnement de l'interpréteur procédural

L'interpréteur procédural se présente sous la forme d'un programme dont le corps est constitué par la liste des appels de procédures définie à partir du texte initial $I_{i/LD}$: c'est la projection de $I_{i/LD}$ sur l'ensemble de primitives P_{i+1} choisi.

Soit $I_{i/P_{i+1}}$ cette liste d'appels de procédures.

L'appel successif des procédures simule l'interprétation de $I_{i/P_{i+1}}$.

exemple:

```
        .
        .
        .
        procédure "$1 := $1+1" ($1 : (A, B, C));
        begin     R1 := ML($1);
                  ML($1) := R1+1
        end;
begin   "$1 := $1+1"(A);                          APPEL
        "$1 := $1+1"(B);          .
             .                    .   I i/P i+1
             .                    .
             .                    .
end .
```

Cette simulation permet de vérifier que la projection de $I_{i/LD}$ sur l'ensemble de primitives P_{i+1} respecte la sémantique initiale: le nouveau programme doit être équivalent au programme initial quant à ses résultats (pour les mêmes entrées).

II/7.2. Codage des primitives

La dernière phase de l'étape consiste en un choix du codage des primitives, de manière à pouvoir implanter le programme $I_{i/P_{i+1}}$ sous une forme interprétable, c'est-à-dire comme une liste d'instructions du langage L_{i+1}.

L'implantation peut être envisagée, d'une manière très simple, comme l'affectation d'un NUMERO D'ORDRE à chaque instruction composant le programme.

L'accès à une instruction est réalisé par INDEXATION d'un TABLEAU de constantes qui regroupe la liste ordonnée des instructions codées.

Nous retrouvons donc le problème du regroupement de données de types différents, que nous nous proposons de résoudre en utilisant la structure conditionnelle (record... case...) présentée précédemment: il est en effet inutile d'étudier, à ce niveau, un codage fin (en terme de BITS) des informations, car seule l'organisation logique est importante au niveau de la conception. L'étude de la représentation interne des informations sera faite dans l'étape suivante, guidée par les mesures dynamiques et par les facteurs d'optimisation du niveau suivant.

II/7.3. Interpréteur des primitives codées: $I_{i+1/LD}$

Chaque instruction apparaissant dans $I_{i/L_{i+1}}$ a reçu un numéro d'ordre.

On définit une nouvelle variable pour l'environnement n° i+1 : le compteur d'instructions.

exemple:
- compteur ordinal (instructions) — niveau 2
- compteur de micro-instructions — niveau 3
- compteur de phases — niveau 4

Cette variable contient le numéro de l'instruction en cours d'interprétation et les opérations de séquencement de l'interprétation sont réalisées en modifiant sa valeur.

On adjoint à l'algorithme d'interprétation:

- une séquence d'initialisation du compteur d'instructions,
- une séquence de lecture et de décodage de l'instruction à interpréter,
- une séquence de modification de la valeur du compteur d'instruction dans la description de chaque instruction.

exemple:

- la description de la primitive "case RI of"
  pourra être: CI := CI + RI.CODOP ;

- celle de "call $1"($1:(P1, P2, P3))

  pourra être: RETOUR := CI+1 ; /*introduction de la
                CI := $1 ; variable RETOUR*/

où le paramètre $1 a pour valeur le numéro de la première instruction de la procédure appelée.

CONCLUSION

La méthodologie proposée met en évidence les différents niveaux d'interprétation et de mémorisation envisageables dans la conception d'une machine informatique, permettant la définition de structures spécialisées (MOYENS) répondant à des besoins précis (PROJET).

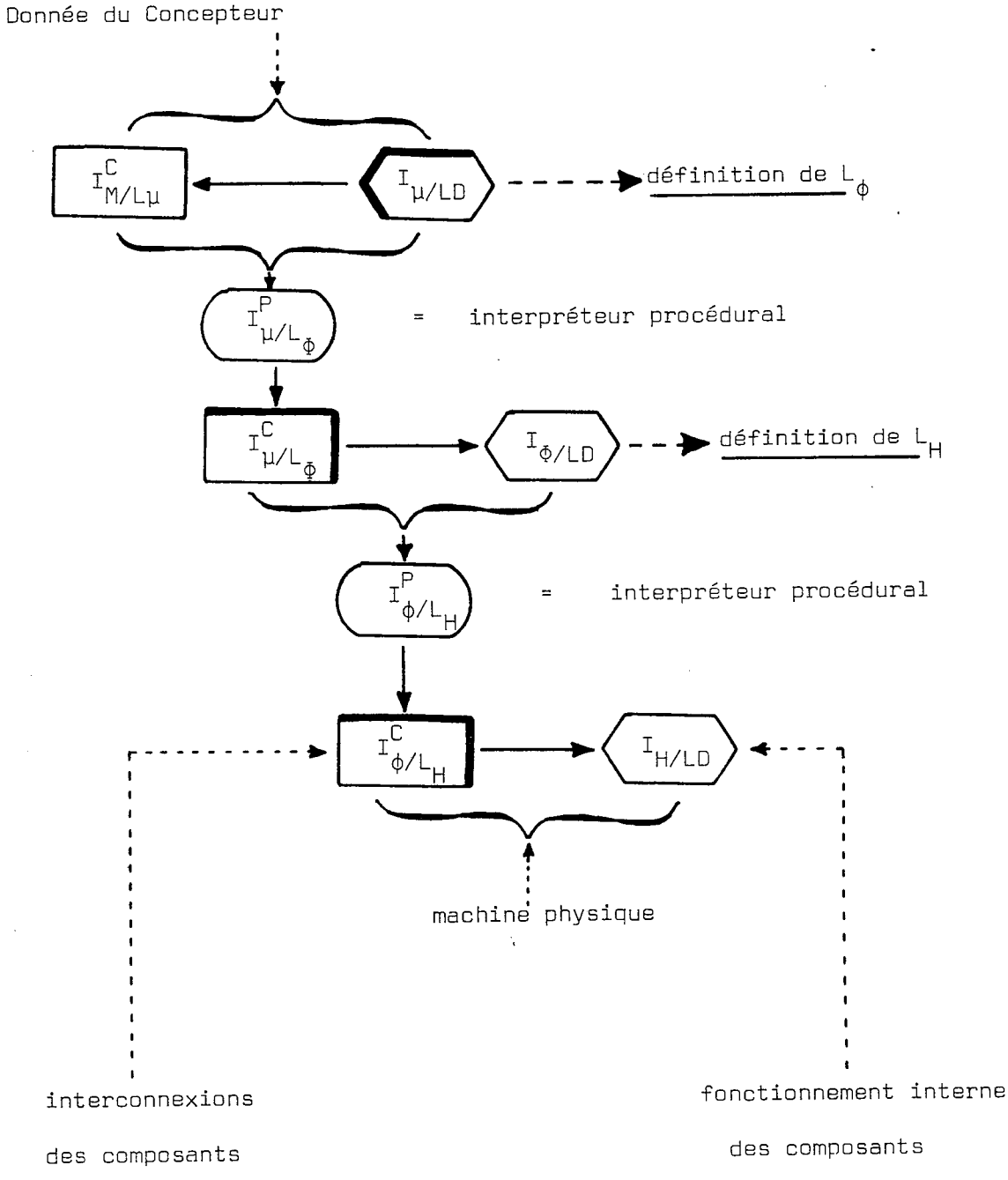
De nombreux points de détail restent à préciser et l'application d'une telle méthode à des projets de taille significative (machine PASCAL,...) permettra de prouver son bien-fondé et de parfaire son utilisabilité.

ANNEXE 2

METHODE DESCENDANTE
Application au langage des phases

P L A N

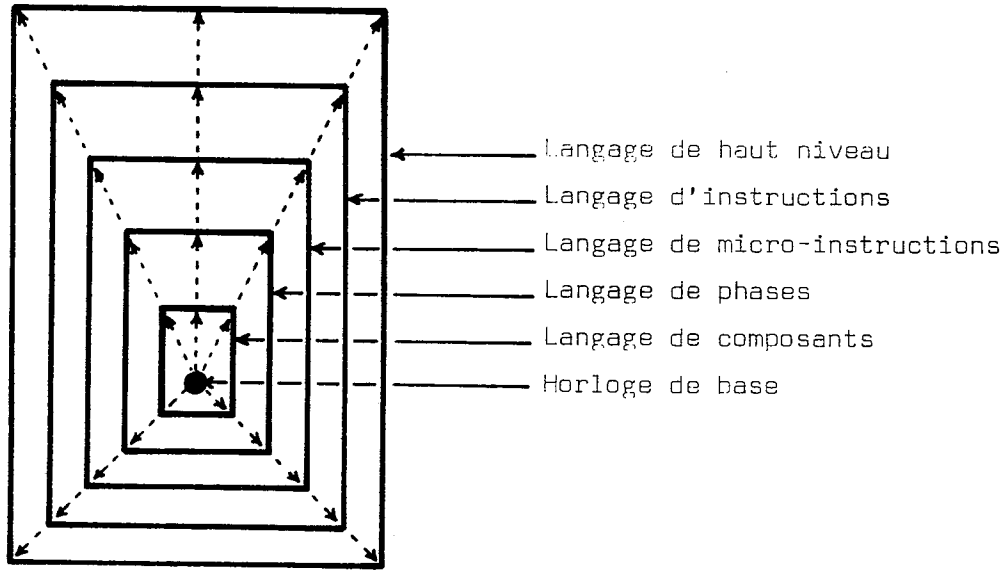
SCHEMA GENERAL DE LA DEMARCHE



I.- RAPPELS SUR LA DEMARCHE DESCENDANTE

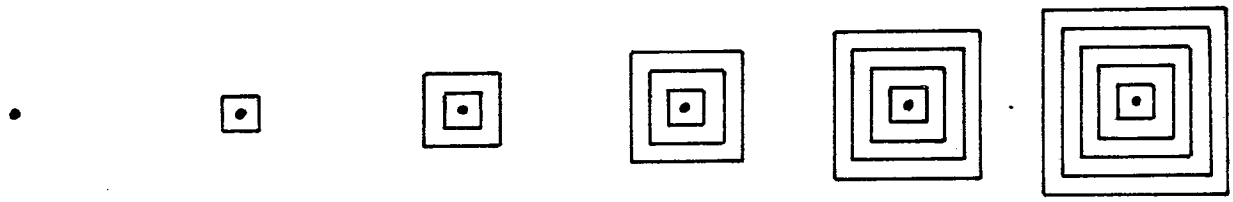
I.1.- IMBRICATION DES NIVEAUX D'INTERPRETATION

Toute machine électronique évoluée est organisée en niveaux d'interprétation imbriqués qui peuvent être vus comme des couches concentriques: une couche n'a d'existence que par les couches qu'elle englobe.

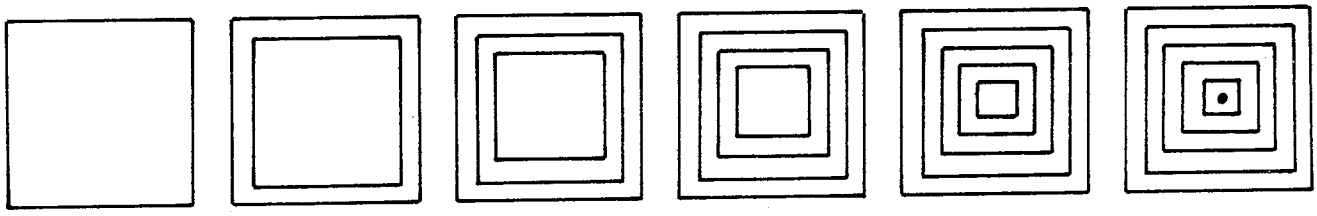


La conception d'une machine électronique est généralement menée du niveau le plus interne (horloge) vers le niveau le plus externe (langage d'instructions), parce qu'il est plus aisé de concevoir une nouvelle couche à partir d'une machine existante que de définir une machine abstraite à partir d'une couche abstraite.

Approche ascendante: ⟶



Approche descendante: ⟶



Nous parlerons indifféremment de couche ou de niveau d'interprétation.

Imbrication:

Le niveau i exécute un langage L_i, et il est lui-même codé comme un programme écrit dans un langage L_{i+1} reconnu par le niveau i+1.

L'approche descendante consiste à partir du niveau le plus externe (niveau 0), pour définir successivement les langages L_1, L_2, L_3... et les niveaux 1, 2, 3,... associés à ces langages.

I.2.- PASSAGE D'UN NIVEAU D'INTERPRETATION A L'AUTRE

Supposons le processus de conception descendante engagé et plaçons-nous au niveau i.

a) Hypothèse de travail

Nous disposons de la description, sous la forme d'un programme écrit dans un Langage de Description LD, de l'algorithme d'interprétation d'un langage L_i. Soit $I_{i/LD}$ ce programme, qui peut être appelé interpréteur formel du langage L_i.

b) But recherché

Le but recherché est de définir une réalisation de l'interpréteur formel donné et si possible la meilleure réalisation au sens des contraintes de coût et de performances précisées au début de l'étude.

c) Définition d'un niveau d'interprétation i+1 pour réaliser le niveau i donné

Etant donné l'interpréteur formel $I_{i/LD}$, la méthode descendante suggère la définition d'un nouveau langage L_{i+1}. Une telle définition d'un langage d'instruction à partir d'un algorithme consiste à extraire de l'algorithme un ensemble fini de primitives qui soit suffisant pour en définir une réalisation.

$I_{i/LD}$ —extraction→ ensemble de primitives

1/ Interpréteur procédural

Chaque primitive extraite de $I_{i/LD}$ peut être considérée comme une procédure qui peut éventuellement être paramétrée.

La projection de $I_{i/LD}$ sur l'ensemble des procédures constitue ce que nous appellerons un interpréteur procédural et que nous noterons $I^{P}_{i/L_{i+1}}$.

Structure d'un interpréteur procédural

Un interpréteur procédural se compose de deux parties:

- une partie active,
- une partie passive.

- La partie active d'un interpréteur procédural est constituée par une séquence d'appels de procédures: c'est la projection de l'algorithme donné sur l'ensemble des procédures.

- La partie passive d'un interpréteur procédural est constituée par une liste de description du corps des procédures. Le corps d'une procédure n'est exécuté que lorsque la procédure est appelée. C'est en ce sens que cette partie est passive.

Sémantique d'un interpréteur procédural

Un interpréteur procédural est équivalent à l'algorithme dont il est issu car il n'en représente qu'une forme plus structurée.

exemple trivial:

soit l'algorithme:

```
A := B ;
D := A+1 ;
```

Définissons les primitives "A:=B" et "D:=A+1".

L'interpréteur procédural est:

```
appel "A:=B";
appel "D:=A+1";
```

partie active

```
déclare A, B, D:entier ;
procédure "A:=B";
    début A:=B fin;
procédure "D:=A+1";
    début D:=A+1 fin;
```

partie passive

Avantages de la notion d'interpréteur procédural

1/ Cette notion permet d'étudier, par simples manipulations de chaînes de caractères, un ensemble de primitives suffisant et optimisé à partir d'un algorithme quelconque.

2/ L'obtention de la partie active est automatique: les chaînes sélectionnées représentent la suite des appels de procédures.

3/ Les variables issues de l'algorithme donné disparaissent de la partie active de l'algorithme ; leurs modifications sont

réalisées dans le corps de procédures. Cette distinction permet d'étudier proprement la projection des variables de l'algorithme donné sur un nouvel ensemble de variables directement manipulé par les procédures: on localise ainsi d'une manière précise le problème de l'implantation des variables.

2/ Interpréteur codé

L'interpréteur procédural défini dans la phase précédente est exécutable: sa partie active définit l'ordre des appels des procédures.

Nous dirons donc que le contrôle de l'exécution d'un interpréteur procédural est réalisé par sa partie active.

Une telle solution n'est pas toujours adoptée dans la conception d'un interpréteur, c'est même la solution opposée qui est généralement réalisée dans les niveaux d'interprétation élevés.

Il est en effet plus répandu de coder les instructions d'un langage sous la forme de données inertes qui sont lues (FETCH) puis décodées (DECODE) par un interpréteur qui lui, possède le contrôle de l'exécution.

Méthode

La partie active de l'interpréteur procédural est transformée en un vecteur de données codées de manière à être reconnues par l'interpréteur. La partie passive de l'interpréteur est rendue active par l'adjonction d'une séquence de contrôle qui est capable d'accéder à une instruction localisée dans le vecteur des instructions codées et d'activer l'une des procédures selon la valeur de la donnée extraite.

Passage de l'intepréteur procédural à l'interpréteur codé

Une transformation simple de l'interpréteur procédural permet d'obtenir le programme codé (vecteur d'instructions codées): c'est le travail généralement réalisé par tout "assembleur", qui est en outre simplifié dans notre cas par le fait que le codage des instructions reste symbolique: on peut en effet définir une structure de données qui décrive tous les types d'instructions possibles.

exemple:

```
INSTRUCTION-360 = structure choix  CODOP:(LR, AR, ...,MVC)parmi
                        LR, AR, ... : <type RR>;
                        L, A, ...   : <type RS>;
                        CLI, MVI,...: <type SI>;
                        MVC, ...    : <type SS>
                  fin
```

Sémantique d'un interpréteur codé

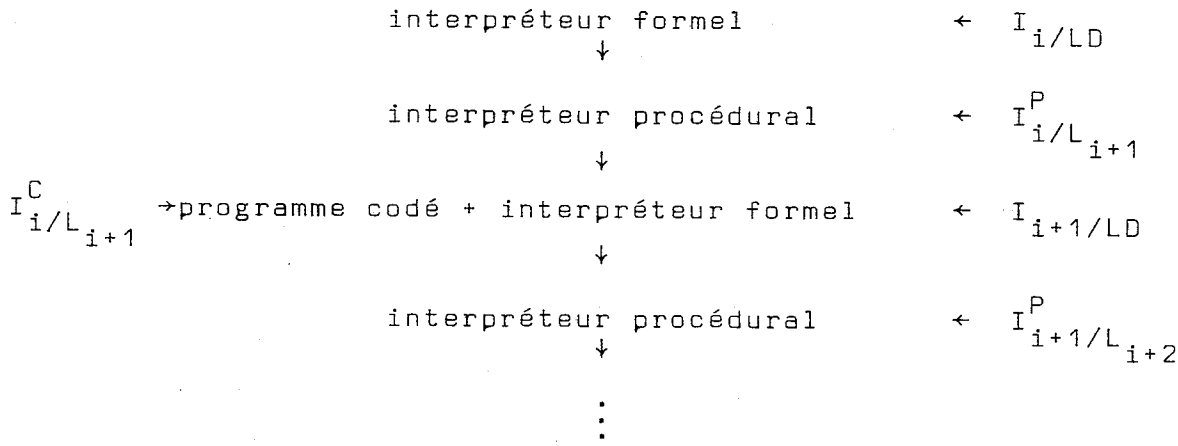
L'interpréteur codé est un programme exécutable écrit dans le langage LD, qui est équivalent à l'algorithme donné au départ de l'étape.

Partant de la donnée de $I_{i/LD}$, nous avons ainsi défini, par l'intermédiaire de l'interpréteur procédural $I^{P}_{i/Li+1}$, un nouveau langage L_{i+1} dans lequel nous avons codé $I_{i/LD}$ qui est donc devenu $I^{C}_{i/L_{i+1}}$. Nous avons d'autre part défini l'interpréteur formel du langage L_{i+1} sous la forme de $I_{i+1/LD}$.

Conséquence: Nous avons remplacé $I_{i/LD}$, algorithme donné, par l'interpréteur formel $I_{i+1/LD}$ qui a comme programme $I^{C}_{i/L_{i+1}}$. Nous avons ainsi défini et simulé une nouvelle couche.

I.3.- CONVERGENCE DE LA METHODE

La méthode ne converge pas si l'on s'en tient à suivre le cheminement décrit en I.2.: en introduisant une nouvelle couche d'interprétation, on obtient un nouvel interpréteur formel qui n'a pas d'existence physique en dehors du formalisme de LD. Il est donc nécessaire de rompre, à une étape de la démarche, l'enchaînement:



Solution proposée

L'enchaînement précédent met en évidence la transmission du contrôle de l'exécution de l'interpréteur procédural vers l'interpréteur formel du programme codé.

La solution consiste à conserver le contrôle de l'exécution dans le programme codé: l'interpréteur du langage codé devient ainsi inerte et n'exécute les instructions que lorsqu'il en reçoit l'ordre de la part du programme qui prend un caractère dynamique

Une telle solution, pour être réalisable, suppose un certain nombre de propriétés, tant pour le programme dynamique, que pour l'interpréteur inerte.

Ces propriétés sont détaillées dans le chapître suivant consacré au langage des phases.

Exemple

A titre d'illustration de l'imbrication des niveaux d'interprétation et du problème de contrôle, nous avons imaginé l'exemple suivant, emprunté à la vie de tous les jours d'un ouvrier spécialisé.

Cet exemple fait des suppositions quant au fonctionnement interne du cerveau humain, mais ne résoud pas le mystère du contrôle interne qui permet l'enchaînement des idées reçues: le troisième niveau de contrôle de l'exemple.

Supposons un ouvrier dont le rôle est de commander une machine.

La machine possède n boutons-poussoirs correspondant aux n opérations qu'elle est capable d'exécuter (jeu d'instructions). La suite des opérations à commander est assez complexe, à tel point que l'ouvrier est incapable de s'en souvenir. Pour y remédier, il a pris soin de noter, sur une feuille de papier, la suite des opérations à commander.

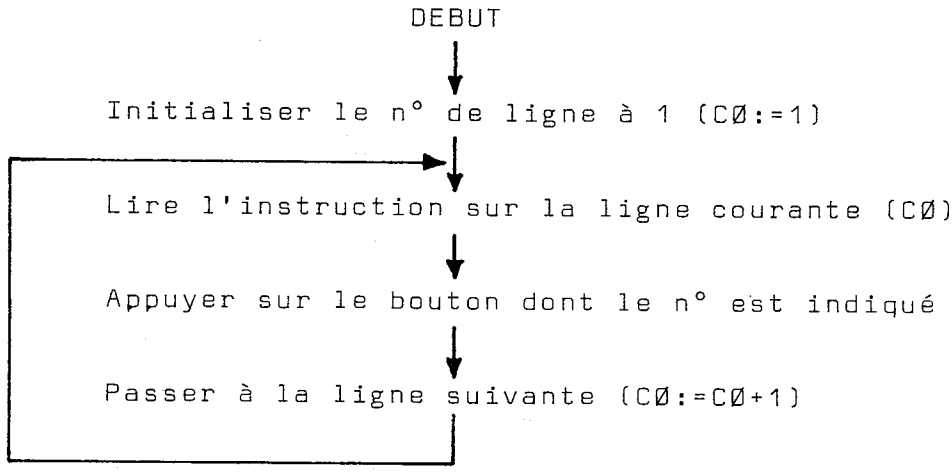
Feuille de papier:

LIGNE	ACTION
1	1
2	4
3	2
4	3
⋮	

La feuille de papier joue donc le rôle d'une MEMOIRE dans laquelle est inscrit le programme à exécuter. Chaque ligne contient une instruction: le numéro du bouton à appuyer.

1 - Premier niveau de contrôle

En premier lieu, l'ouvrier doit se souvenir qu'il faut exécuter l'instruction inscrite sur la première ligne. Il doit d'autre part être capable de progresser d'une ligne à l'autre. Supposons donc qu'il mémorise dans son cerveau le numéro de la ligne (compteur ordinal). L'ouvrier exécute donc:



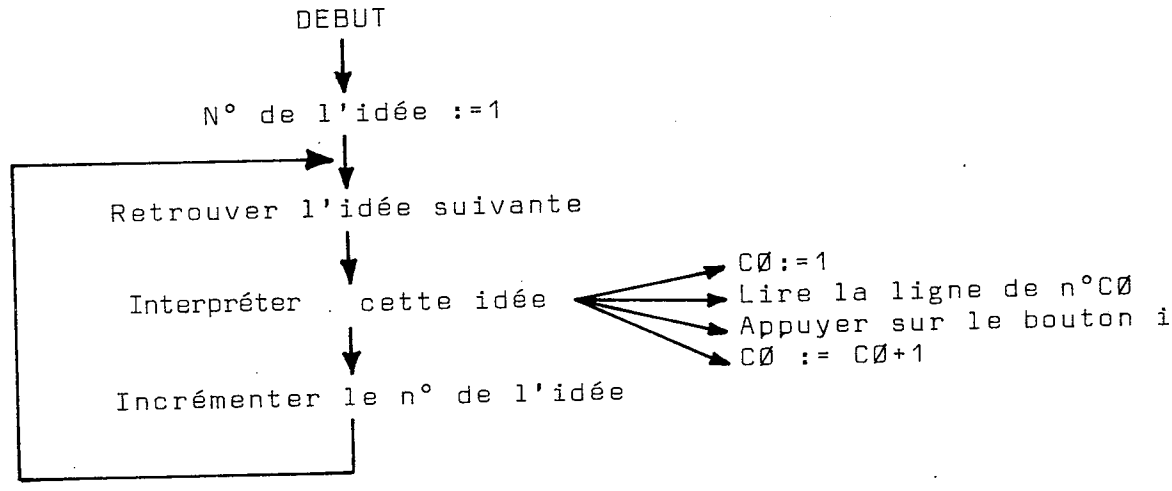
L'ouvrier joue donc le rôle de la partie contrôle, la machine jouant celui de la partie opérative de la machine informatique composée de l'association ouvrier-machine.

2 - Deuxième niveau de contrôle

Le fonctionnement très simple de la machine informatique décrite ci-dessus suppose que l'ouvrier ait suivi un stage de formation (apprentissage) pendant lequel il aurait appris l'algorithme décrit plus haut.

Si l'on schématise le fonctionnement du cerveau de l'ouvrier, on peut supposer que chacune des instructions de l'algorithme est mémorisée dans une "case" du cerveau et qu'un mécanisme d'adressage permet de retrouver chaque instruction (appelée IDEE).

Le deuxième niveau de contrôle est alors celui qui permet d'aller chercher dans une case du cerveau l'instruction suivante (l'IDEE suivante).



3 - Troisième niveau de contrôle

L'algorithme précédent suppose lui aussi un niveau supplémentaire de contrôle pour réaliser l'enchaînement des idées.

On a ainsi mis en évidence deux machines imbriquées.

II.- DEFINITION DU LANGAGE DE PHASES

II.1.- SITUATION DANS LA DEMARCHE

Plutôt que de parler des langages L_0, L_1, ..., L_i, ... et des niveaux d'interprétation associés, nous emploierons désormais une notation plus concrète.

Soit donc $I_{\mu/LD}$ l'interpréteur formel d'un langage L_μ, qui sera dans la majorité des machines un langage de micro-programmation. Dans cette terminologie, le micro-programme sera noté $I^C_{M/L\mu}$, car c'est le codage dans L_μ de l'interpréteur formel $I_{M/LD}$ du langage machine L_M.

Nous nous plaçons donc dans le cas où la donnée est l'interpréteur formel $I_{\mu/LD}$, tous les niveaux précédents ayant été définis.

II.2.- LE PROBLEME DU CONTROLE DE L'INTERPRETATION

Comme nous l'avons montré dans le précédent chapitre, la convergence de la méthode descendante repose sur le contrôle de l'exécution. Est-il donc possible de résoudre ce problème, au moins partiellement, à ce niveau de la méthode? Réservant notre réponse, nous dirons qu'il devient indispensable de trouver une solution, sachant qu'on ne peut plus repousser le problème indéfiniment et qu'une solution programmée du contrôle de l'interpréteur de L_μ serait bien trop lente et coûterait le prix d'une mémoire supplémentaire. Il serait cependant possible de concevoir une machine dans laquelle l'interpréteur du langage de phases irait extraire ses instructions d'une mémoire: on pourrait alors parler de nano-programmation.

Nous choisirons cependant de résoudre le problème du contrôle au niveau du langage des phases, puisqu'il doit être résolu.

Comme nous l'avons suggéré dans le précédent chapitre, la solution consiste à doter le programme codé du contrôle de l'exécution et de rendre passif

l'interpréteur du langage de phases. Cela suppose que le langage de phases choisi vérifie un certain nombre de propriétés.

II.3.- CARACTERISTIQUES D'UN LANGAGE DE PHASES

a/ Comme dans les étapes précédentes, la donnée du concepteur est un interpréteur formel $I_{\mu/LD}$ décrit dans le formalisme de LD, à partir duquel nous allons sélectionner un ensemble fini de primitives qui constituera l'ensemble des instructions du langage L_ϕ.

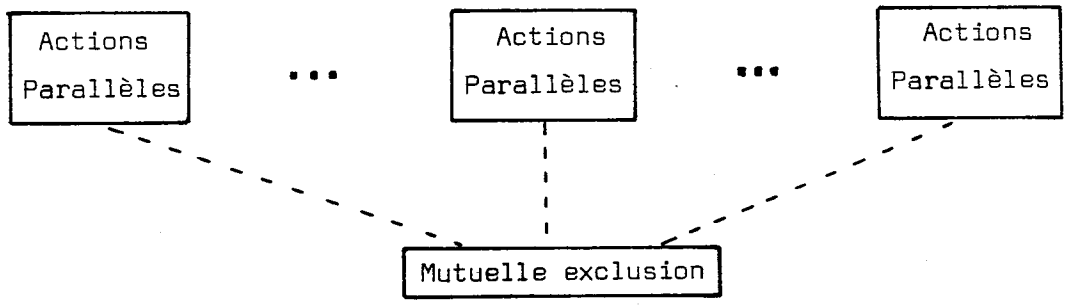
Notre but est de définir un langage L_ϕ tel que l'interprétation de chacune de ses instructions ne nécessite aucun contrôle.

L'interprétation d'une instruction ne peut donc consister qu'en un ensemble d'actions qui sont

- soit exécutables en parallélisme collatéral,
- soit mutuellement exclusives.

Plus généralement, l'interprétation sans contrôle d'une instruction peut être caractérisée par

des groupes d'actions parallèles mutuellement exclusifs



b/ Définition du langage L_ϕ

Les primitives du langage L_ϕ doivent être sélectionnées dans l'algorithme donné $I_{\mu/LD}$, de manière à respecter la propriété énoncée plus haut.

La sélection des chaînes opérée dans les étapes précédentes n'est plus possible par le fait que $I_{\mu/LD}$ est exprimé dans le formalisme d'un langage séquentiel.

exemple:

soit l'algorithme en LD:

```
DEBUT: A := 0;
      choix <exp 1> parmi
            0: début  A := A+1;
                      si <exp 4> alors  B := A
               fin ;
            1: début choix <exp 2> parmi
                      0 : B := 0;
                      1 : B := B+1
                  fin;
                      C := B+1
               fin ;
            2: A := 0;
            3: si <exp 3> alors  B := B-1
      fin;
            allera DEBUT;
```

1) On associe à chaque action ou groupe d'actions sa condition algorithmique d'exécution.

Ainsi A := A+1 doit être exécuté si <exp 1> est égal à zéro,

B := A si <exp 1>=0 et si <exp 4> ='VRAI"

2) On associe toutes les actions mutuellement exclusives dans une même primitive.

3) On isole les actions qui doivent être exécutées séquentiellement dans des primitives différentes.

exemple précédent:

```
procédure P1;
      début  A := 0 fin;
procédure P2;
      début choix <exp 1> parmi
            0 : A:= A+1;
            1 : choix <exp 2> parmi
                        0 : B := 0;
                        1 : B := B+1
                fin;
            2 : A := 0;
            3 : si <exp 3> alors B := B-1
           fin
      fin;
procédure P3;
      début choix <exp 1> parmi
            0 : si <exp 4> alors B := A;
            1 : C := B+1;
            2 :  ;
            3 :
            fin
      fin
```

L'ensemble des primitives de L_ϕ est dans ce cas l'ensemble {P1, P2, P3}.

c/ Propriétés des primitives de L_ϕ

Chaque primitive de L_ϕ doit être constituée de groupes d'actions parallèles mutuellement exclusifs. En conséquence, un certain nombre de propriétés doivent être vérifiées.

1 - Ambiguïté collatérale

Le parallélisme collatéral de deux actions signifie que l'ordre dans lequel les deux actions sont exécutées est indifférent, donc quelconque. Il faut donc prévoir l'occurrence d'ambiguïtés qui pourraient avoir pour effet un résultat qui dépende de l'ordre d'exécution.

exemple:

soit A := B+1 , B := 0

suivant l'odre dans lequel les deux actions sont exécutées, A recevra soit l'ancienne valeur de B incrémentée de 1, soit la valeur 1.

Règle générale: dans une primitive de L_ϕ, appelée phase, on ne peut pas à la fois lire la valeur d'une variable et modifier la valeur de la même variable.

Cas particulier: une exception à la règle générale existe, qui se limite au cas particulier "A := A+1", ou "A := A-1", et que nous étudierons plus en détail dans un prochain chapître.

2 - Ambiguïté algorithmique

Deux actions figurant dans la même primitive peuvent être conditionnées par deux conditions algorithmiques différentes, tout en étant exécutables en parallélisme collatéral.

exemple:

si <exp 1> alors B := 0 , si <exp 2> alors A := B+1

Il se peut, algorithmiquement, que <exp 1> et <exp 2> ne prennent jamais la valeur 'VRAI' simultanément, mais une telle description ne permet pas d'en être certain. On peut donc retrouver une ambiguïté collatérale dans le cas où les deux conditions <exp 1> et<exp 2> sont vraies simultanément.

Conclusion

Le concepteur doit s'assurer qu'il n'existe aucune ambiguïté collatérale ni aucune ambiguïté algorithmique.

Les interdépendances entre les variables ne peuvent donc apparaître que pour des actions qui sont algorithmiquement mutuellement exclusives.

Si toutefois une ambiguïté subsiste, elle sera levée en reportant certaines actions dans une primitive différente.

Remarque 1(référence bibliographique):

On pourra se reporter au rapport final du précédent contrat pour une présentation plus théorique des problèmes du parallélisme collatéral.

Remarque 2:

On aurait pu présenter le problème des ambiguïtés en termes d'Assignation unique. En effet, soit l'exemple:

```
si <exp> alors  A := 1
         sinon  A := 0
```

Nous pouvons assurer la mutuelle exclusion des actions en se référant à la sémantique de la construction si ... sinon. Nous pouvons également utiliser un autre formalisme:

```
A := si <exp> alors 1 sinon 0
```

qui met en évidence l'assignation unique.

Cette décomposition est cependant inintéressante pour la suite de la démarche car elle fait disparaître la distinction entre les deux actions A := 1 et A := 0 qui apparaîtront dans l'étape suivante.

II.4.- INTERPRETEUR PROCEDURAL DU LANGAGE DE PHASES

Ayant défini un ensemble de primitives vérifiant les propriétés énoncées plus haut, l'obtention de l'interpréteur procédural du langage des phases ($I^P_{\mu/L\phi}$) est quasi immédiate:

- la partie exécutive de $I^P_{\mu/L\phi}$ est la liste des appels des procédures

précédemment définies, cette liste étant ordonnée de manière à respecter la même séquentialité que dans l'algorithme donné I_{μ/L_ϕ}.

Reprenons l'exemple précédent:

$I_{\mu/LD}$ est en page 16, les procédures de L_ϕ sont en page 17, l'interpréteur procédural associé $I^P_{\mu/L\phi}$ est:

```
DEBUT: appel P1 ;
       appel P2 ;
       appel P3 ;
       allera DEBUT ;
```

- la partie passive de l'interpréteur procédural est constituée par le corps des procédures, qui peut être exécuté sans aucun contrôle supplémentaire lorsque la procédure est appelée.

```
procédure P1 ;

procédure P2 ;

procédure P3 ;
```

II.5.- INTERPRETEUR CODE DU LANGAGE DES PHASES

Le problème du contrôle de l'exécution que nous avons posé plus haut trouve ici sa solution. Cette solution consiste à laisser l'initiative du contrôle à la partie exécutive de l'interpréteur procédural $I^P_{\mu/L\phi}$. Cette partie exécutive n'est pas transformée en un vecteur d'instructions codées comme dans les étapes précédentes, mais elle devient un programme dynamique.

Méthode de codage du programme dynamique

- On introduit une variable entière baptisée ϕ, qui reçoit une séquence de valeurs entières.

- On définit l'interpréteur $I_{\phi/LD}$ du programme dynamique codé $I^C_{\mu/L\phi}$

sans aucun contrôle supplémentaire de la manière suivante:

$I\phi_{/LD}$ = si ϕ = 1 alors appel P1,
si ϕ = 2 alors appel P2,
si ϕ = 3 alors appel P3.

Programme dynamique codé $I^{C}_{\mu/\ L\phi}$:

DEBUT: ϕ := 1;
ϕ := 2;
ϕ := 3;
allera DEBUT.

Le programme précédent symbolise dans le formalisme de LD une séquence infinie d'affectations des valeurs 1, 2 puis 3, 1, 2 puis 3, ... à la variable ϕ.

Conclusion:

Nous avons ainsi introduit un interpréteur dont l'exécution est contrôlée par une donnée. Cette donnée est la variable ϕ dont la valeur valide l'exécution de l'une des primitives de L_{ϕ}.

On peut donc appeler $I_{\mu/LD}$ un algorithme contrôlé par une donnée variable. Nous montrerons dans le chapître suivant qu'une telle formalisation correspond non seulement à la réalité des machines existantes, mais encore permet d'introduire un niveau supplémentaire de langage qui n'a encore jamais été montré à notre connaissance.

Ce niveau supplémentaire est introduit dans le chapître suivant: c'est la réalisation de l'interpréteur du langage L_{ϕ} à l'aide de primitives d'un nouveau langage L_{H} qui sera lui, directement "interprété" par l'ensemble des composants électroniques qui constituent la machine physique.

III.- DEFINITION DE LA MACHINE ELECTRONIQUE

III.1.- DEFINITION D'UN LANGAGE L_H DONT L'INTERPRETATION EST DIRECTEMENT REALISABLE PAR DES COMPOSANTES ELCTRONIQUES

1/ On distingue deux types de composants électroniques:

a - Les Circuits Combinatoires

Un circuit combinatoire évalue en permanence une fonction arithmétique ou logique de ses entrées. On considérera un tel circuit comme étant l'interpréteur d'une instruction de L_H.

exemple:
Soit la fonction (ou expression) "A = B".
On peut la considérer comme une instruction interprétée par un comparateur électronique.

b - Les Circuits Séquentiels

Un circuit séquentiel exécute une action élémentaire lorsqu'il est sollicité. On considérera un tel circuit comme étant l'interpréteur d'une instruction de L_H.

exemple:

Soit l'action "A := A+1", considérée comme une instruction de L_H interprétée par un compteur électronique.

2/ Sélection des primitives de L_H

La sélection des primitives de L_H correspond à un choix de composants électroniques qui soit suffisant pour réaliser l'interprétation du programme donné $I_{\phi/LD}$.

Dans l'exemple proposé à la page 16, la liste des primitives sélectionnées seraient:

"A := 0", "A := A+1", "B := 0", "B := B+1", "B := B-1",
"B := A", "C := X" où X = "B + 1", "<exp 1>", "<exp 2>", "<exp 3>".

a - Les variables mémorisées

La sélection des primitives fait apparaître des procédures d'affectation d'une valeur à une variable. De telles primitives sont interprétables par des circuits séquentiels qui mémorisent et modifient la valeur d'une variable.

Règle: toutes les primitives qui modifient la valeur d'une variable mémorisée sont interprétées par un circuit séquentiel unique, que nous appellerons sous-interpréteur multiple.

b - Les variables combinatoires

La valeur d'une expression peut être symbolisée par une variable qui devient alors synonyme de la fonction. Nous associons à chaque primitive interprétée par un circuit combinatoire, une variable combinatoire qui contient le résultat de l'évaluation.

Toutes les expressions arithmétiques ou logiques ne sont pas directement interprétables par un circuit combinatoire.

Il est parfois nécessaire de sélectionner comme primitive une ou plusieurs sous-expressions dans une expression complexe.

exemple:
"$\neg$(A.B+C.D)" peut être sélectionnée comme une primitive de L_H

composant: SN7451

par contre (A+B).(C+D) doit être décomposé

en X = "A+B"
Y = "C+D"
compte tenu des composants disponibles,

et devient finalement "X.Y", primitive réalisable.

c - Obtention d'un ensemble de circuits électroniques

La sélection des primitives à partir de $I\phi_{/LD}$ fait donc apparaître l'ensemble des circuits électroniques nécessaires.

L'association est simple:

- à chaque primitive combinatoire correspond un circuit combinatoire ;
- à chaque ensemble de primitives référençant la même variable mémorisée correspond un circuit séquentiel multiple.

III.2.- INTERPRETEUR PROCEDURAL DE LANGAGE L_H

Les primitives sélectionnées dans l'étape précédente peuvent être décrites:

- sous la forme de procédures pour les primitives réalisables par un circuit séquentiel;
- sous la forme de fonctions pour les primitives réalisables par un circuit combinatoire.

exemple:

```
fonction  "A=B" : booléen ;
  début   "A=B" := (A=B) fin ;
procédure  "X:=X+1";
  début   X:=X+1 fin ;
```

interpréteur procédural:

```
si "A=B" alors appel "X:=X+1"
```

a - Problème du contrôle de l'interpréteur procédural

Nous avons vu précédemment que le contrôle de $I_{\mu/LD}$ provenait de l'interpréteur codé $I^{C}_{\mu/L\phi}$ qui était symbolisé par:

```
DEBUT : Φ ← 1;
        Φ ← 2;
        Φ ← 3;
        goto DEBUT.
```

Si l'on veut transmettre une activité d'un niveau à l'autre, il est nécessaire de définir une réalisation physique de la seule partie dynamique de la machine

La solution consiste à définir un circuit combinatoire qui interprète l'instruction complexe précédente:

"DEBUT : Φ ← 1; Φ ← 2; Φ ← 3 ; goto DEBUT".

Un tel circuit combinatoire existe et peut être élaboré à partir d'un oscillateur à quartz d'une manière très classique.

La primitive précédente apparaît dans l'interpréteur procédural I^{P}_{ϕ/L_H} comme une instruction toujours vraie. Le circuit combinatoire correspondant fournit comme résultat la valeur de la variable Φ qui évolue dans le temps et transmet ainsi le contrôle aux différentes primitives de l'interpréteur procédural.

Conclusion

L'interpréteur procédural I^{P}_{Φ/L_H} est contrôlé par la variable Φ dont la valeur est automatiquement générée par un circuit combinatoire. Il ne nécessite donc aucun contrôle.

b - Simulation de l'interpréteur procédural

Pour réaliser une simulation, il est nécessaire de générer la suite des valeurs de la variable Φ. On pourra donc structurer l'interpréteur procédural de manière à faciliter sa simulation.

exemple:

```
DEBUT: Φ:=1;                    procédure INTERP;
       appel INTERP;            début choix Φ parmi
       Φ:=2 ;                         1:appel P1;
       appel INTERP;                  2:appel P2
       goto DEBUT;              fin
```

III.3.- INTERPRETEUR CODE DU LANGAGE L_H

L'interpréteur procédural I^{P}_{Φ/L_H}, malgré la correspondance théorique entre les primitives de L_H et des composants électroniques, reste une entité programmée qui n'a de sens que par la sémantique du langage de description LD. Il permet seulement de simuler un enchaînement de primitives de L_H, mais non une exécution physique.

Le problème est donc de coder la partie exécutive de l'interpréteur procédural de telle sorte qu'il puisse être interprété par des composants électroniques qui remplaceraient eux-mêmes la partie descriptive des procédures et fonctions.

1/ Codage des appels des procédures

Le codage d'un appel de procédure (instruction de L_H) doit satisfaire aux deux conditions suivantes:

1 - le codage doit être reconnu par le circuit séquentiel qui interprète l'instruction ;

2 - le codage doit être tel que l'instruction ne soit interprétée que si, et seulement si, les conditions algorithmiques sont réalisées.

Un composant électronique séquentiel reçoit sur des broches d'entrée des fils conducteurs portés à un potentiel variable.

Classiquement, on a deux potentiels possibles. On peut donc considérer chaque fil porteur d'un potentiel électrique comme une variable booléenne porteuse d'une valeur logique 0 ou 1.

Un composant électronique est généralement un interpréteur multiple: à chaque instruction qu'il est capable d'exécuter correspond une broche d'entrée porteuse d'une valeur logique.

Un composant exécute une instruction lorsque la variable logique correspondant à l'entrée associée à l'instruction change de valeur, dans un sens donné.

a/ Notion de transition

La valeur d'une variable logique peut passer de 0 à 1 ou de 1 à 0.

Nous appellerons tout changement de valeur, une transition orientée.

Transition montante: passage de la valeur de 0 à 1.
Transition descendante: passage de la valeur de 1 à 0.

Convention:

Dans un premier temps, nous prendrons comme convention qu'un composant électronique séquentiel est sensible à une transition montante, c'est-à-dire qu'il interprète une transition montante comme un ordre d'exécuter l'instruction associée à l'entrée sur laquelle la transition s'est produite.

Remarque importante

Pour que deux transitions montantes puissent se succéder dans le temps, il est nécessaire qu'une transition descendante se produise après la première transition montante.

Dans le temps, on doit donc avoir une alternance de transitions montantes et de transitions descendantes.

La valeur d'une variable booléenne ne peut pas être quelconque à l'origine des temps: de cette valeur initiale dépend le type de la première transition.

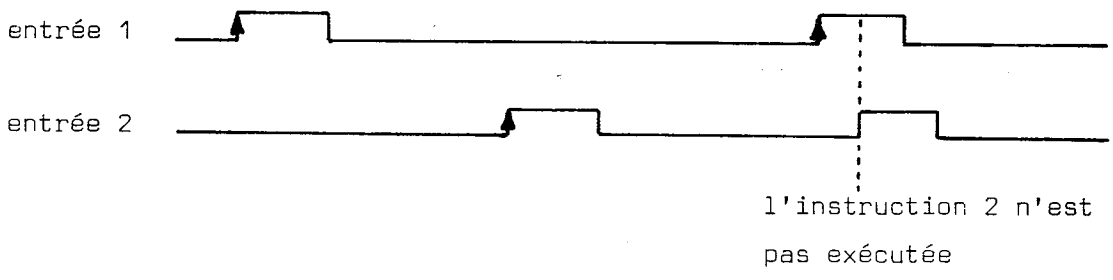
Valeur initiale	Suite des transitions
0	M, D, M, D, ...
1	D, M, D, M, ...

b/ Le problème des ressources multiples

Certaines ressources physiques peuvent exécuter plusieurs instructions différentes. Cela se traduit par le fait qu'elles disposent de plusieurs entrées de type instruction.

Une telle ressource ne peut évidemment réaliser qu'une seule instruction dans un intervalle de temps donné. Cet intervalle de temps est défini par l'instruction en cours d'exécution.

exemple:



Conséquence:

Les différentes entrées-instruction d'une ressource physique multiple ne peuvent pas être utilisées d'une manière quelconque, sous peine de conduire à des résultats aléatoires: elles doivent être synchronisées.

Solution:

On rend mutuellement exclusives les activations de toutes les instructions. Il existe deux possibilités:

1/ soit assurer la mutuelle exclusion algorithmique dans une même phase: toutes les instructions peuvent être activées au même instant, mais une seule est activable à chaque étape (solution synchrone).

2/ soit assurer la mutuelle exclusion temporelle: toutes les instructions sont activées à des instants différents (solution asynchrone).

exemple 1

si Φ = i alors choix <exp> parmi
0 : <instruction 0>
⋮
n : <instruction n>

exemple 2

si Φ = 0 alors <instruction 0>
⋮
si Φ = n alors <instruction n>

Il existe une solution hybride, qui consiste à utiliser les deux types de mutuelle exclusion.

exemple 3

si Φ = 0 alors choix <exp 0> parmi
0 : <instruction 0>
1 : <instruction 1>

si Φ = 1 alors choix <exp 1> parmi
0 : <instruction 2>
1 : <instruction 3>
2 : <instruction 4>

Conclusion

Les propriétés vérifiées par le langage des phases lors de l'étape précédente assurent que le problème des ressources multiples est correctement résolu.

c/ Evaluation des variables logiques

Les contraintes de codage des instructions de L_H doivent être respectées dans la définition de la valeur des variables logiques associées aux entrées instruction des composants électroniques.

Soit "A := A+1" le nom d'une variable logique associée à l'entrée-instruction d'une ressource physique de type compteur. Il s'agit de définir une fonction logique dont la valeur sera affectée en permanence à la variable "A := A+1", de telle sorte que la suite des valeurs de la fonction définisse les instants auxquels l'instruction doit être exécutée.

L'origine du dynamisme de l'interpréteur procédural est située dans l'instruction combinatoire

"DEBUT : Φ := 1; Φ := 2; allera DEBUT"

qui fournit une séquence de valeurs de la variable Φ Cette instruction combinatoire est exécutée en permanence, nous l'appellerons "fonction génératrice des phases" (horloges).

Le problème consiste à coder les valeurs générées par cette fonction, de manière à pouvoir transmettre l'activation des différentes instructions, en définissant les variables logiques en fonction de la variable Φ.

d/ Sémantique du contrôle de I^{P}_{ϕ/L_H}

Soit si "Φ=1" alors si "CONDITION" alors "INSTRUCTION", dont la sémantique est la suivante:
si la fonction "Φ=1" a la valeur "VRAI", alors si la fonction "CONDITION" a la valeur "VRAI", alors appeler l'"INSTRUCTION".

Le test de la valeur de la fonction "CONDITION" ne peut être fait que si l'on est certain que cette fonction a bien été évaluée (l'évaluation n'est pas instantanée). D'après les caractéristiques du langage de phases, on sait que la valeur d'une fonction ne peut dépendre que du résultat d'une phase précédente: il suffit donc de s'assurer que l'intervalle de temps qui sépare deux phases est suffisant pour permettre l'évaluation de toutes les fonctions dont la valeur est utilisée dans la phase.

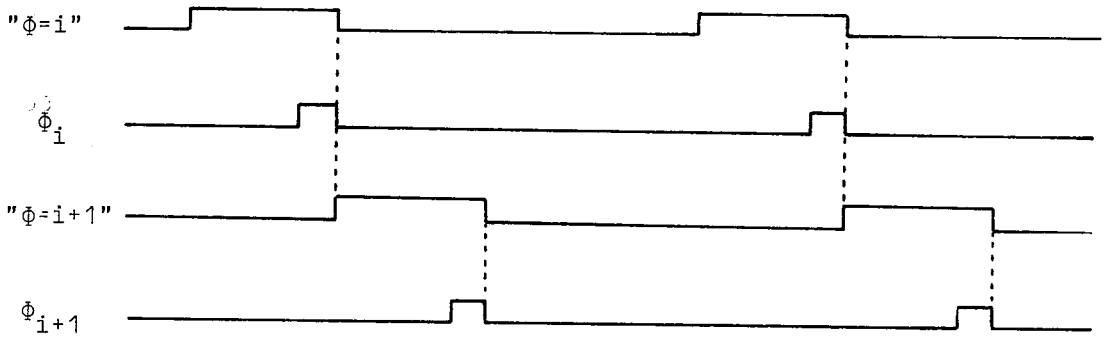
La variable "INSTRUCTION" doit d'autre part avoir pour valeur celle d'une fonction dont le résultat passe de 0 à 1 (transition montante par convention) si, et seulement si, la condition algorithmique d'activation de l'instruction est vraie, et ceci dans l'intervalle de temps où la condition "Φ=i" reste vraie.

Conséquence:

Il faut que la variable Φ soit codée de telle sorte qu'on puisse prévoir l'instant où la condition "Φ=i" va devenir fausse.

Solution pour le codage de la variable Φ

On associe à la variable Φ autant de variables logiques Φ_i que Φ peut prendre de valeurs différentes, de la manière suivante:



On peut ainsi définir une variable logique instruction comme étant égale à l'intersection logique d'une variable Φ_i et de la fonction "CONDITION" algorithmique dont l'évaluation peut être validée par le résultat de la fonction "Φ=i".

exemple:

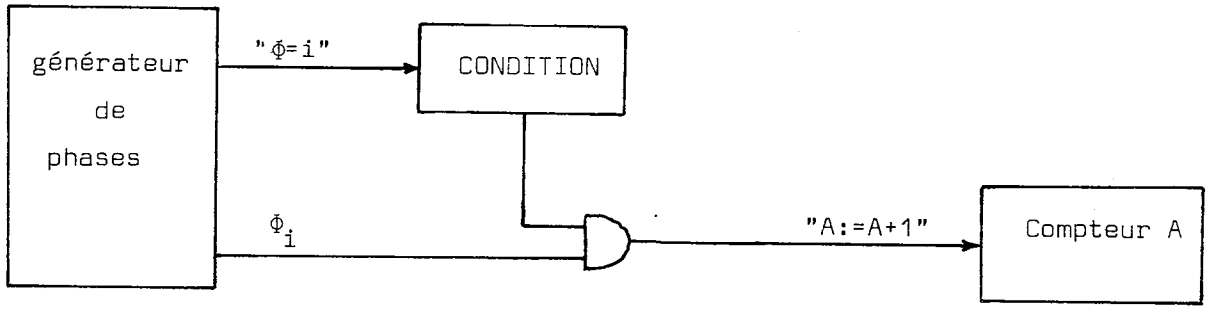
si "Φ=i" alors si "CONDITION" alors "A := A+1"

codage

"A := A+1" = Φ_i. CONDITIONVALIDEE

CONDITIONVALIDEE = "Φ=i".CONDITION

Symbolisme



e/ Sémantique de l'interpréteur codé I^{C}_{ϕ/L_H}

. L'interpréteur codé est constitué par une liste d'instructions codées de la forme

<variable logique> = <expression>

où toutes les <expressions> représentent les sorties de composants électroniques.

. Une <expression> contient des noms de paramètres qui sont des variables logiques égales à des sorties de composants: les paramètres représentent les entrées des circuits combinatoires.

. Certaines variables logiques représentent les entrées-instructions des circuits séquentiels.

Conclusion

L'interpréteur codé décrit toutes les connexions entre les entrées et les sorties de tous les composants électroniques.

D'autre part, l'ensemble des composants électroniques utilisés représente l'interpréteur du langage codé L_H.

III.4.- LES PROBLEMES DE SIMULATION

La définition d'un interpréteur formel $I_{H/LD}$ permettrait de simuler l'interprétation du langage L_H et en particulier d'interpréter le programme codé I^{C}_{ϕ/L_H}.

Ce problème de simulation suppose une étude plus poussée de la sémantique de L_H, et en particulier suppose qu'on soit capable d'extraire d'une variable logique son aspect temporel.

Nous abordons le problème de la simulation du hardware avec un ensemble de variables banalisées, contrairement à CASSANDRE qui dispose d'une part du monde des horloges porteuses de la notion de temps, d'autre part du monde des signaux porteurs de valeurs logiques.

La suite de cette étude portera sur ce problème.

LISTE DES PUBLICATIONS RELATIVES A CE PROJET

F.ANCEAU, J.ROHMER, JP.SCHOELLKOPF
"Architecture des calculateurs et synchronisation"
Journées IRIA, TOULOUSE, juin 73

R.FORTIER, JP.MARTIN, JP.SCHEOLLKOPF
"Etude d'une machine PASCAL"
Rapport de DEA, Université de GRENOBLE, juin 73

F.ANCEAU, R.FORTIER, JP. SCHOELLKOPF
"Conception descendante des machines informatiques: application à une machine PASCAL"
Journées AFCET "Machines orientées langage, machines orientées système"
2/3 mai 74, l'Alpe d'Huez

JP. SCHEOLLKOPF
"A top-down approach of design and functionak description of hardware"
ACM German Vhapter, IEE German section - Workshop on CHDL'S, DARMSTADT, 31/7 1/8/74.

JP. SCHOELLKOPF
"Microprogramming: a step of a top-down design methodology"
7th annual workshop on microprogramming, MICRO 7, SIGMICRO, PALO ALTO 29/9 1/10/74

F.ANCEAU, R.FORTIER, JP.SCHOELLKOPF
"Conception descendante des machines informatiques: application à une machine PASCAL"
Rapport de contrat SESORI N°73042/1.81, novembre 1974

G.BAILLE, JP.SCHOELLKOPF
"Evaluation d'une expression post-fixée sur une file d'attente et réalisation d'une machine pipeline de haut niveau"
Séminaire de programmation ENSIMAG, GRENOBLE, 6/6/75.

G.BAILLE, JP. SCHOELLKOPF
"Evaluation of polish form expression on a FI-FO queue: a new approach towards the realization of a high level pipeline computer"
SAGAMORE Computer conférence, SYRACUSE University

F.ANCEAU, G.BAILLE, JP.SCHOELLKOPF
"Machine informatique électronique destinée à l'exécution parallèle d'un langage post-fixé"
Brevet ANVAR N°75 29 473, déposé le 26/9/75

F.ANCEAU, G.BAILLE, JP.SCHOELLKOPF
"Conception descendante, machine PASCAL"
Rapport intermédiaire contrat SESORI N°74 156, 10/10/75

G.BAILLE, JP.SCHOELLKOPF
"A pipeline polish string computer"
AFIPS 1976 National computer conference, NCC, NEW YORK, juin 76

JP. SCHOELLKOPF
"PASC-HLL: a pipelined architecture bit slice computer for high level language"
COMPCON 77, IEEE, WASHINGTON DC, septembre 77

JP.SCHOELLKOPF, G.BAILLE
"PASC-HLL: définition et réalisation d'un calculateur "pipeline" adapté au traitement des langages évolués"
Séminaire d'informatique de l'ENSIMAG, juin 77

ARTICLES REFERENCES

[1] R.FORTIER
"Etude et définition du langage intermédiaire et d'une machine formelle multiprocesseurs orientée vers l'exécution du langage PASCAL"
Thèse de 3ème cycle, Université de GRENOBLE, 14/10/75

[2] Basic principles of the B 6500
BURROUGHS Corporation

[3] G.H. POUJOULAT
"The CORAIL building block system"
EUROMICRO Newsletter, octobre 76

[4] F.ANCEAU
"Some design problems relating to system oriented computer architecture"
Symposium on logical organization of computers, VARSOVIE, septembre 1975

[5] A.GREBERT
"Space programming language machine study"
SAMSO TR 72.117, mai 1972

[6] DB. WORTMAN
"A study of language directed computer design"
CSRG.20, VOF TORONTO, dec.1972

[7] Microprocesseur SFC 92900
SESCOSEM, octobre 1976

[8] G.BAILLE
"Machine PASC-HLL: réalisation avec des microprocesseurs en tranches d'une Unité centrale multiprocesseur adaptée au langage PASCAL"
A paraître, thèse de 3°cycle INPG (fin 1977)

[9] G.BERGER-SABBATEL
"Etude et évaluation sur maquette de l'implantation hardware d'algorithmes de gestion de fichiers, adaptés aux petits systèmes"
Contrat SESORI N°74 136, juillet 75

ARTICLES DE REFERENCE

MULLERY, SCHOUER & RICE
"A problem oriented SYMBOL processor"
SJCC 1963

CRE ECH - Architecture of the B6500
COINS 69 - 3nd International symposium on computer and information science
MIAMI 1969

CHESLEY & SMITH
"The hardware implemented high level machine language for SYMBOL"
SJCC 1971

ABRAHMS
"An APL machine"
STANDFORD University, Computer department, n° CS 70 158, 1970

R.A.
"An implementation of Blaise on the B1726"
TR 81, june 1974, University of NEW YORK

N.WIRTH
"The Programming language PASCAL"
1970

K.JENSEN & N.WIRTH
"An user manual for PASCAL"
Avril 1974

J.K. ILLIFE
"Basic machine principles"
AMER.ELSEVIER - NEW YORK, 1968

Mc KEEMAN
"Language directed computer design"
FICC 1967, pp.413/417

BARTON
"Ideas for computer organization"
COINS 69, 3rd International symposium on computer and information science
MIAMI, 1969

F.ANCEAU
"Systèmes hiérarchisés"
Journées IRIA, ST PIERRE DE CHARTREUSE, novembre 1973.

F.ANCEAU
"Contribution à l'étude des systèmes hiérarchisés de ressources
dans l'architecture des machines informatiques"
Thèse de doctorat d'Etat - GRENOBLE - 5 décembre 1974

dernière page de la thèse

AUTORISATION DE SOUTENANCE

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

VU le rapport de présentation de :

- Monsieur François ANCEAU, Maître de Conférences à l'I.N.P.G.

Monsieur Jean-Pierre SCHOELLKOPF

est autorisé à présenter une thèse en soutenance pour l'obtention du titre de DOCTEUR DE TROISIEME CYCLE, spécialité "Génie Informatique".

A Grenoble, le 6 Juin 1977

Ph. TRAYNARD
Président
de l'Institut National Polytechnique