N° d'ordre : 257

50376
1981
5

UNIVERSITÉ DES SCIENCES ET TECHNIQUES DE LILLE

THÈSE

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNIQUES DE LILLE

pour obtenir le titre de

DOCTEUR-INGÉNIEUR

(TRAITEMENT DE L'INFORMATION)

par

Bernard PETITPREZ

ETUDE ET REALISATION D'UN SYSTEME MULTIPROCESSEUR A PILOTAGE PAR LES DONNEES

Thèse soutenue le lundi 12 janvier 1981 devant la Commission d'Examen

Membres du Jury	MM. C. CARREZ	Président
	V. CORDONNIER	Rapporteur
	G. VANDECANDELAERE	Membre invité
	C. TIMSIT	Membre invité

P R O F E S S E U R S 1ERE C L A S S E

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

.../...

Professeurs 1ère classe (suite)

M. SALMER Georges	I.E.E.A.
Mme SCHWARTZ Marie Hélène	Mathématiques
M. SEGUIER Guy	I.E.E.A.
M. STANKIEWICZ François	Sciences Economiques
M TILLIEU Jacques	Physique
M. TRIDOT Gabriel	Chimie
M. VIDAL Pierre	I.E.E.A.
M. VIVIER Emile	Biologie
M. WERTHEIMER Raymond	Physique
M. ZEYTOUNIAN Radyadour	Mathématiques

P R O F E S S E U R S 2ème C L A S S E

M. AL FAKIR Sabah	Mathématiques
M. ANTOINE Philippe	Mathématiques
M. BART André	Biologie
Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mathématiques
M. BELLET Jean	Physique
M. BKOUCHE Rudolphe	Mathématiques
M. BOBE Bernard	S.E.S.
M. BODART Marcel	Biologie
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie
M. BOSQ Denis	Mathématiques
M. BREZINSKI Claude	I.E.E.A.
M. BRUYELLE Pierre (Chargé d"enseignement)	Géographie
M. CAPURON Alfred	Biologie
M. CARREZ Christian	I.E.E.A.
M. CHAMLEY Hervé	E.U.D.I.L.
M. CHAPOTON Alain	C.U.E.E.P.
M. COQUERY Jean Marie	Biologie
Mme CORSIN Paule	Sciences de la Terre
M. CORTOIS Jean	Physique
M. COUTURIER Daniel	Chimie
Mle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	E.U.D.I.L.
M. DEGAUQUE Pierre	I.E.E.A.
M. DELORME Pierre	Biologie
M. DEMUNTER Paul	C.U.E.E.P.
M. DE PARIS Jean-Claude	Mathématiques

.../...

M. DEVRAINNE Pierre	Chimie
M. DHAINAUT André	Biologie
M. DORMARD Serge	S.E.S.
M. DOUKHAN Jean Claude	E.U.D.I.L.
M. DUBOIS Henri	Physique
M. DUBRULLE Alain	Physique
M. DUEE Gérard	Sciences de la Terre
M. DYMENT Arthur	Mathématiques
M. FLAMME Jean Marie	E.U.D.I.L.
M. FONTAINE Hubert	Physique
M. GERVAIS Michel	S.E.S.
M. GOBLOT Rémi	Mathématiques
M. GOSSELIN Gavriel	S.E.S.
M. GOUDMAND Pierre	Chimie
M. GREVET Patrice	S.E.S.
M. GUILBAULT Pierre	Biologie
M. HANGAN Théodor	Mathématiques
M. HERMAN Maurice	Physique
M. JACOB Gérard	I.E.E.A.
M. JACOB Pierre	Mathématiques
M. JOURNEL Gérard	E.U.D.I.L.
M. KREMBEL Jean	Biologie
M. LAURENT François	I.E.E.A.
Mlle LEGRAND Denise	Mathématiques
Mlle LEGRAND Solange	Mathématiques (Calais)
Mme LEHMANN Josiane	Mathématiques
M. LEMAIRE Jean	Physique
M. LENTACKER Firmin	G.A.S.
M. LEVASSEUR Michel	I.P.A.
M. LHENAFF René	G.A.S.
M. LOCQUENEUX Robert	Physique
M. LOSFELD Joseph	I.E.E.A.
M. LOUAGE Francis	E.U.D.I.L.
M. MACKE Bruno	Physique
M. MAIZIERES Christian	I.E.E.A.
Mlle MARQUET Simone	Mathématiques
M. MESSELYN Jean	Physique
M. MIGEON Michel	E.U.D.I.L.
M. MIGNOT Fulbert	Mathématiques
M. MONTEL Marc	Physique
Mme NGUYEN VAN CHI Régine	G.A.S.
M. PARSY Fernand	Mathématiques
Mlle PAUPARDIN Colette	Biologie

Professeurs 2ème classe (suite)

M. PERROT Pierre	Chimie
M. PERTUZON Emile	Biologie
M. PONSOLLE Louis	Chimie
M. PORCHET Maurice	Biologie
M. POVY Lucien	E.U.D.I.L
M. RACZY Ladislas	I.E.E.A.
M. RICHARD Alain	Biologie
M. RIETSCH François	E.U.D.I.L.
M. ROGALSKI Marc	M.P.A.
M. ROUSSEAU Jean-Paul	Biologie
M. ROY Jean-Claude	Biologie
M. SALAMA Pierre	S.E.S.
Mme SCHWARZBACH Yvette (CCP)	M.P.A.
M. SCHAMPS Joël	Physique
M. SIMON Michel	S.E.S.
M. SLIWA Henri	Chimie
M. SOMME Jean	G.A.S.
Mle SPIK Geneviève	Biologie
M. STERBOUL François	E.U.D.I.L.
M. TAILLIEZ Roger	Institut Agricole
M. TOULOTTE Jean-Marc	I.E.E.A.
M. VANDORPE Bernard	E.U.D.I.L.
M. WALLART Francis	Chimie
M. WATERLOT Michel	Sciences de la Terre
Mme ZINN JUSTIN Nicole	M.P.A.

CHARGES DE COURS

M. TOP Géard	S.E.S.
M. ADAM Michel	S.E.S.

CHARGES DE CONFERENCES

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

Je tiens à remercier

Monsieur CARREZ, Professeur à l'Université de Lille I, qui m'a fait l'honneur de présider le jury de cette thèse, pour les nombreux conseils et pour l'aide qu'il m'a apportée tout au long de ce travail.

Monsieur CORDONNIER, Professeur à l'Université de Lille I qui après m'avoir orienté vers cette recherche, n'a cessé de me prodiguer conseils et encouragements.

Monsieur TIMSIT, Ingénieur de recherche à la CII HB pour l'intérêt qu'il a manifesté pour cette étude.

Monsieur VANDECANDELAERE, Directeur de l'Institut Supérieur d'Electronique de Nord d'avoir bien voulu examiner ce rapport.

Je remercie également

Mes collègues de laboratoire pour les échanges de vue que nous avons eus.

Madame LECOUFFE avec qui j'ai eu de fructueuses discussions qui ont contribué à l'avancement de ce travail.

Madame CARREZ, pour les travaux de programmation et de simulation qu'elle a effectués.

Madame CARON qui a dactylographié ce texte avec beaucoup de gentillesse et de compétence.

Monsieur et Madame DEBOCK qui avec beaucoup de soin et de diligence ont assuré le tirage de ce document.

A mes parents

TABLE DES MATIÈRES

CHAPITRE II : MAUD Machine d'assignation unique dynamique

1. Introduction
    - 1.1. Volume des communications
    - 1.2. Dynamique du contrôle
    - 1.3. Compatibilité technologique
2. Assignation unique par blocs
    - 2.1. Introduction
    - 2.2. Communications entre blocs
    - 2.3. Structure d'un bloc
    - 2.4. Création dynamique de blocs
    - 2.5. Exemples de programmes pour Maud
3. Présentation du modèle
4. Fonctionnement du système
    - 4.1. Les opérateurs de traitement
    - 4.2. L'opérateur mise à jour
    - 4.3. L'opérateur constructeur
    - 4.4. Déroulement d'un programme
5. Caractéristiques d'une architecture matérielle
6. Conclusions

CHAPITRE III : La mémoire de MAUD

1. Influence des caractéristiques du modèle sur le choix d'une architecture de mémoire
    - 1.1. Fonctions de la mémoire
    - 1.2. Accès associatif
    - 1.3. Accès multiples
    - 1.4. Mémoire commune
    - 1.5. Gestion de la mémoire
    - 1.6. Temps d'accès
    - 1.7. Extensibilité

CHAPITRE IV : Les échanges processeurs mémoire

1. Procédures d'échanges entre les processeurs et l'anneau
    1.1. Les processeurs de traitement
    1.2. Le processeur mise à jour
    1.3. Le processeur constructeur
2. Principe de fonctionnement de l'unité d'échange
    2.1. Objectifs d'une réalisation
    2.2. Déroulement d'un échange
    2.3. Décomposition de l'unité d'échange
3. L'unité de transfert
    3.1. Etude fonctionnelle
    3.2. Réalisation pratique
4. L'unité de recherche
    4.1. Analyse fonctionnelle
    4.2. Paramètres de recherche d'un cadre
    4.3. Paramètres de recherche à l'intérieur d'un cadre
    4.4. Durée de vie d'une demande
    4.5. Réalisation pratique
    4.6. Le module de recherche associative
    4.7. Le module de comptage de mots et de comptage de cadres
    4.8. L'unité de contrôle microprogrammée
    4.9. Communications processeur - unité de recherche
5. Exemple d'utilisation : le processeur pupitre
    5.1. Architecture du processeur
    5.2. Fonctions du processeur pupitre
6. Conclusion

CHAPITRE V : Simulation

1. Introduction
2. Simulation du modèle statique
    2.1. Représentation du système
    2.2. Configuration du système
    2.3. Parallélisme maximum

INTRODUCTION

Les besoins sans cesse croissants en puissance de calcul dans les systèmes informatiques ont conduit à l'étude d'architectures nouvelles capables d'exploiter le parallèlisme contenu implicitement ou explicitement dans les programmes.

La réalisation matérielle de telles architectures a été rendue possible pour une grande part grâce à l'évolution des circuits LSI. On a vu ainsi apparaître de nouveaux composants ayant des performances élevées. Leur faible coût, leur facilité d'emploi permettent d'envisager des machines pour lesquelles on réalise des groupements de quelques dizaines, quelques centaines, et même quelques milliers de ces circuits.

Cette révolution technologique n'a toutefois pas fait disparaître toutes les difficultés. Des problèmes tels que la répartition du travail et les communications entre plusieurs processeurs, leur synchronisation deviennent extrêmement complexes. D'autre part, la plupart de ces machines étant conçues suivant un modèle conventionnel, c'est à dire dirigé par les instructions, ne permettent pas de détecter le parallèlisme. Cette opération doit être effectuée avant l'exécution par le programmeur ou à l'aide de mécanismes automatiques au niveau de la compilation.

L'introduction des concepts de traitement dirigé par les données et d'assignation unique a permis de résoudre pour une grande part ces difficultés. Dans ces machines, l'exécution d'un programme n'est plus cadencée par le flot des instructions mais par le flot des données.

La plupart des études qui ont été menées à ce sujet exploitent ces concepts au niveau des instructions. Bien que cette méthode semble parfaitement naturelle, elle entraine un volume de communication important dans le système.

Dans ce travail nous nous proposons d'étudier l'architecture d'un modèle de machine dirigée par les données : Maud [LEC 79c], qui applique ces concepts non plus au niveau de l'instruction mais au niveau d'un bloc qui se compose d'un ensemble d'instructions et de l'ensemble des objets utilisés par ces instructions. Dans le chapitre II nous abordons une description de ce modèle.

Le chapitre III porte sur l'étude d'un système de communication dans une structure multiprocesseurs basé sur l'utilisation d'une mémoire commune. L'emploi d'une technologie de mémoire particulière nous a permis de résoudre un grand nombre de difficultés que l'on rencontre dans des architectures construites autour de mémoire classique.

Dans le chapitre IV les échanges nécessaires entre les processeurs et l'outil de communication dans le cas de Maud sont étudiés . Ils nous ont conduits à l'étude d'une unité d'échange évoluée qui autorise une grande variété de transferts. Sa souplesse d'utilisation permet de décharger les processeurs de tous les problèmes d'échanges.

L'architecture que nous proposons a été simulée.Le chapitre V présente les deux programmes de simulation que nous avons écrits ainsi que les premiers résultats que nous avons obtenus.

CHAPITRE I

MÉTHODES DE DÉTECTION ET D'EXPLOITATION DU PARALLÉLISME DANS LES ORDINATEURS

Le problème de la détection du parallèlisme contenu dans les programmes et de son exploitation effective au niveau des machines, a fait l'objet de très nombreuses études. Deux voies de recherche ont été principalement suivies :

- Au *niveau logiciel*, l'objectif a été de se donner les moyens et les outils pour exprimer ou détecter, à la compilation, le parallèlisme, implicitement ou explicitement présent dans un algorithme.

- Au *niveau matériel*, les recherches ont donné lieu à l'étude de nombreuses architectures de machines parmi lesquelles certaines ont débouché sur une réalisation concrète.

Nous passerons en revue, dans ce chapitre, quelques unes de ces voies de recherche.

1. Approches logicielles

Trois voies de recherche ont été suivies dans ce domaine :

- L'introduction au niveau des langages machines ou des langages évolués de quelques *instructions spécialisées* dans l'expression et le contrôle du parallèlisme.

- Etude de *méthodes automatiques* de détection du parallèlisme entre deux instructions ou deux groupes d'instructions.

- *Nouveaux concepts* de programmation et d'exécution des taches en vue d'assurer la prise en compte et le contrôle du traitement parallèle.

1.1. Utilisation de nouvelles instructions spécialisées

L'introduction du parallèlisme peut se faire soit par l'adjontion de nouvelles instructions machines soit par l'adjonction de nouvelles instructions dans les langages évolués.

1.1.1. Instructions FORK-JOIN [CONW 63]

Conway a proposé deux instructions de contrôle permettant de déclarer et de provoquer la parallélisation possible de deux séquences (FORK)

ou la réunion de deux séquences parallèles (*JOIN*).

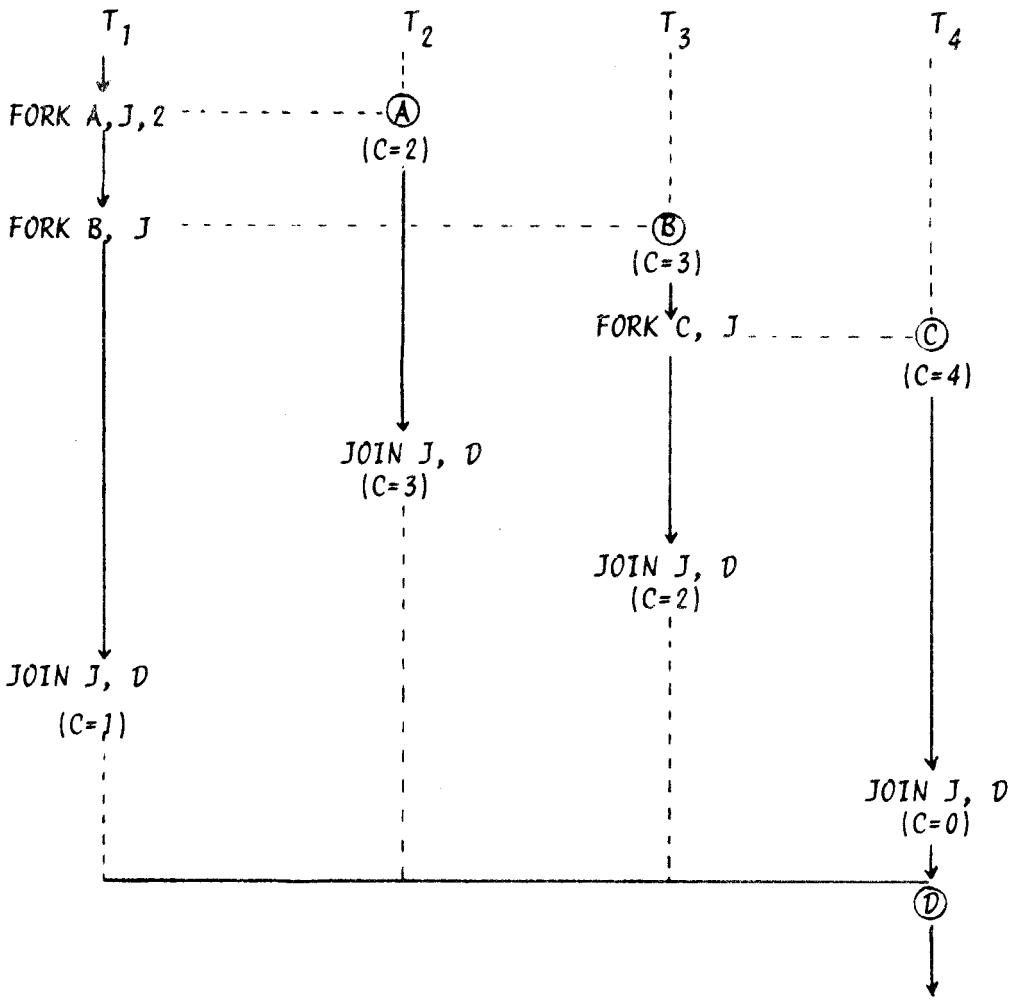


Fig 1 : Exemple d'utilisation des instructions FORK-JOIN

Le schéma ci-dessus illustre les mécanismes liés à l'exécution de ces instructions.

- *FORK A, J, N.* Lorsque l'instruction FORK apparait dans une séquence, elle provoque la création d'une activité parallèle. Celle-ci est désignée par une étiquette. FORK A initialise la séquence débutant par A. J désigne un compteur qui indique le nombre de séquences se déroulant en parallèle. N précise la valeur initiale de ce compteur.

- *FORK A, J* assure l'incrémentation du compteur J et l'exécution en parallèle de la suite d'instructions commençant par A.

- *JOIN J, B* assure la décrémentation du compteur J et la poursuite de l'exécution de la séquence d'instructions débutant par B si la valeur du compteur est nulle. Dans le cas contraire, l'unité de traitement est désactivée.

On constate que la suite du traitement est pris en charge par la dernière unité de traitement active. Ce type de parallèlisme ne prévoit aucun mécanisme de communication des données, il est donc conçu pour une exécution sur une architecture à mémoire commune.

1.1.2. Instructions en langage évolué.

L'introduction de nouvelles instructions dans les langages évolués donne, au programmeur, les moyens de spécifier que deux tâches peuvent se dérouler concuremment. Toutefois le programmeur doit connaître le parallèlisme existant dans son programme ou dans son algorithme. Ces méthodes ne permettent pas d'atteindre le parallèlisme optimal, c'est à dire celui qui entraine un temps minimal d'exécution, d'autant plus que les algorithmes conçus pour un traitement séquentiel sont généralement peu adaptés au traitement parallèle.

Des travaux ont été menés sur des langages tel que FORTRAN, PL1, APL ou ALGOL [DIJK 65], [VEIL 72].

Bien que ces méthodes permettent l'expression du parallèlisme présent dans les programmes ou dans les algorithmes, elles demandent au programmeur un effort particulier pour le détecter et l'exprimer, cette opération n'est pas toujours simple à réaliser au moment de la programmation.

1.2. Méthodes automatiques de détection du parallélisme

Ces méthodes permettent de déterminer automatiquement à partir d'un programme séquentiel, les opérations qui peuvent se dérouler simultanément.

Bernstein [BERN 66] a décrit les propriétés que doivent vérifier les tâches d'un programme pour que celles ci puissent être exécutées en parallèle sur un système multiprocesseur.

Il distingue trois types de tâches

- Les tâches *séquentielles*. L'exécution de celles-ci doit se faire dans un ordre déterminé.

- Les tâches *commutables*. Elles peuvent être exécutées dans un ordre quelconque mais elles ne sont pas parallèlisables.

- Les tâches *parallèles*. L'exécution de ces tâches peut se faire en parallèle.

Les conditions de Bernstein dépendent non seulement des caractéristiques propres du programme mais également de l'organisation mémoire du système multiprocesseur, qui sera chargé de l'exécution du programme. L'existence d'une mémoire locale, associée à chaque unité de traitement permet de simplifier les conditions requises pour la parallélisation.

Les conditions de Bernstein ont été reprises pour réaliser des programmes effectuant une analyse automatique de programmes écrits en langage évolué.

Ramamoorthy et Gonzalez [RAMA 69] ont mis au point des techniques de préparation de programmes séquentiels en vue d'une exécution sur une machine parallèle ainsi que des méthodes de répartition de tâches entre les différentes unités de traitement. La construction d'un graphe associé au programme conduit à la détermination de sous-graphes fortement connexes qui seront considérés comme constituant une tâche. L'analyse du graphe résultant permet ensuite de déterminer les moments d'exécution des différentes tâches.

Evans et Williams [EVAN 79] ont défini une méthode d'analyse et un programme qui peut être inclus dans un compilateur ALGOL 68 pour détecter le parallèlisme potentiel. Deux phases sont nécessaires.

- La phase d'*analyse* découpe le programme en "blocs" appelés stanzas. Les blocs sont composés d'un certain nombre d'instructions correspondant à

une structure du langage (boucle ...) ou utilisant un nombre limité de variables.

- La phase de *détection* détermine les relations existantes entre les blocs.

Les informations ainsi obtenues permettent de définir les blocs qui peuvent être exécutés en parallèle.

Il faut remarquer que la transformation d'un programme séquentiel en un ensemble de tâches parallèles est une opération longue et coûteuse. Ce traitement ne peut trouver de justification que pour de très gros programmes destinés à être souvent exécutés. Toutes les méthodes vues jusqu'à présent, cherchent à détecter ou à créer le parallèlisme avant l'exécution. L'introduction de nouveaux concepts de programmation permet de réaliser cette opération au cours de l'exécution du programme.

1.3. Nouveaux concepts de programmation en vue d'un traitement parallèle

Si un algorithme présente des propriétés intrinséques d'aptitude au parallélisme, celles-ci doivent apparaître et être reconnues indépendamment du langage de description de l'algorithme.

Les concepts qui sont présentés ici, s'appuient sur le fait que ces propriétés transparaissent non plus dans le programme et par conséquent à travers le langage support mais bien dans des relations entre les données elles-mêmes.

La relation la plus évidente est la relation de dépendance. Ces approches sont dites dirigées par les données.

1.3.1. Structuration en blocs d'un programme

Roucairol et Widory [Rouc 73] ont défini une forme générale de langage externe.

Un programme écrit dans ce langage se présente sous la forme d'une suite de blocs. Un bloc est constitué d'une séquence d'instructions d'affectations dépendantes d'un même ensemble de conditions. A chaque bloc

est associée une valeur unique d'une variable d'activation du programme qui conditionne l'exécution du bloc.

A l'intérieur du bloc, les listes de liens associées à chaque instruction permettent une exécution parallèle des instructions contenues dans le bloc.

1.3.2. Cadencement par les données : Data flow

Les premières études sur le *traitement cadencé par les données* ont été effectuées au MIT par l'équipe du professeur Dennis [DEN 74] en utilisant une représentation graphique des programmes.

Dans le langage défini par Dennis, un programme est un graphe orienté comportant deux sortes de noeuds :

- *noeuds de chainage* (link)

- *noeuds acteurs* (actor)

Les arcs qui relient ces noeuds peuvent être considérés comme des chainons le long desquels circulent des marques (tokens) auxquelles sont associées des valeurs. Il existe deux sortes de marques :

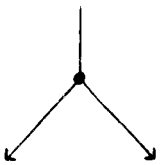
- les *marques de contrôle* associées à une valeur booléenne.

- Les *marques de données* associées à une valeur quelconque.

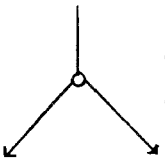
Dans un graphe, il peut exister plusieurs arcs d'entrée mais un seul arc de sortie.

Les principaux types de noeuds sont décrits figure 2.

- *noeuds de chainage*

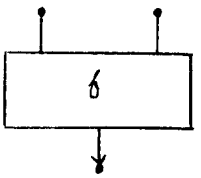


noeud de chainage de donnée

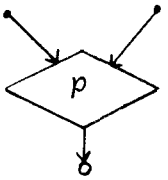


noeud de chainage de contrôle

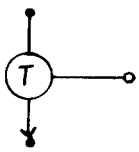
- *noeuds acteurs*



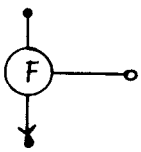
opérateur f



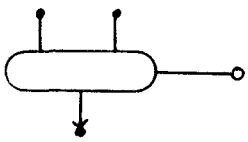
prédicat



porte T (true)



porte F (false)



jonction (merge)

Fig 2 : Principaux noeuds du langage de base du MIT

Un noeud acteur est activé lorsque les marques sont présentes sur tous ses arcs d'entrée et qu'il n'y a pas de marque sur l'arc de sortie.

- Les marques sont enlevées des arcs d'entrée.

- Le calcul est effectué à partir des valeurs associées aux marques d'entrée.

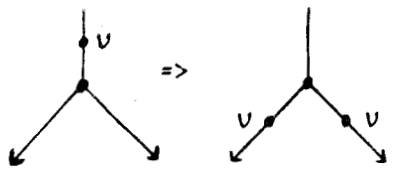
- Une marque "résultat" associée à la valeur calculée est placée sur l'arc de sortie.

Les noeuds de chainage permettent la duplication d'une marque lorsque celle-ci est destinée à plusieurs noeuds acteurs.

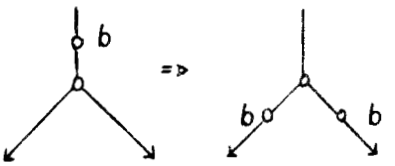
L'exécution d'un programme dirigé par les données est décrite par une suite d'états, chacun montrant le graphe muni de ses marques et de ses valeurs associées. On passe d'un état à l'autre par traversée des noeuds du

graphe suivant un certain nombre de règles dont les principales sont décrites ci-dessous.

-noeud de chainage

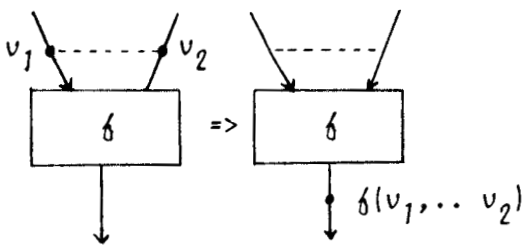


chainage de donnée

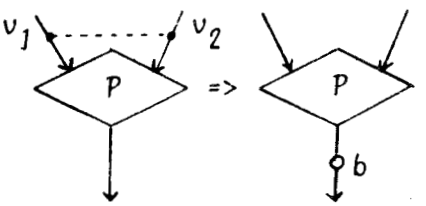


chainage de contrôle

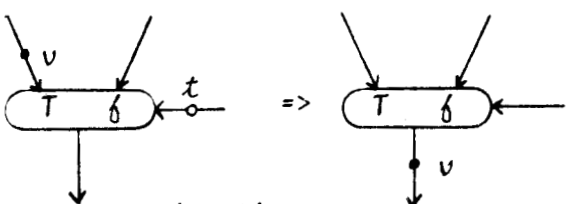
- noeuds acteurs



opérateur δ



prédicat



jonction

Fig 3 : Règles d'activation des noeuds du graphe

Exemple de programme

```
INPUT w, x ;
y:= x ; t:= 0 ;
WHILE t ≠ w DO ;
      BEGIN ;
      IF y > 1 THEN y:= y/2 ;
               ELSE y:= y * 3 ;
      t:= t+1 ;
      END ;
OUTPUT y ;
```

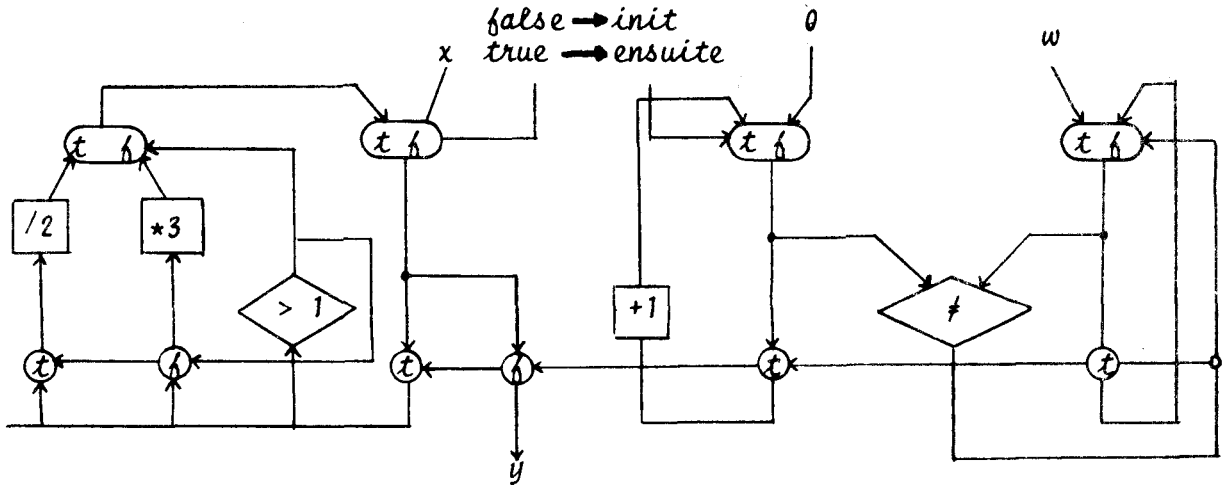


Fig 4 : Représentation graphique d'un programme Data flow

Une telle représentation est différente des représentations classiques. On n'a plus un contrôle général mais un contrôle local. Le parallèlisme apparait tout naturellement du fait que plusieurs noeuds peuvent opérer simultanément.

Un programme dirigé par les données est un ensemble partiellement ordonné d'opérations. Une opération ne peut s'exécuter que lorsque tous ses opérandes sont disponibles. De plus une opération ne produit pas d'effets de bord comme résultat de son exécution. Toutes les instructions dont les opérandes sont calculés peuvent s'exécuter en parallèle.

1.3.3. Concept d'assignation unique

Le concept d'*assignation unique* a été introduit par Tesler et Enea [TESL 68]. La définition donnée à l'assignation unique est la suivante :

"Une variable ne peut recevoir au plus qu'une valeur pendant toute l'exécution du programme".

Dans un langage à assignation unique, une instruction est une affectation simple ou généralisée. Le séquencement des instructions n'est plus fondé sur une écriture séquentielle du programme mais sur la disponibilité des variables nécessaires à l'exécution. En effet une instruction est exécutable dés l'instant où les variables qu'elle utilise comme données ont reçu une valeur. Tous les instructions possédant leurs données sont

exécutables simultanément. Il n'est plus nécessaire d'expliciter le parallélisme existant dans le programme.

Différents langages utilisant le concept d'assignation unique ont été définis. Nous allons en citer un certain nombre soit parce qu'ils ont introduit de nouvelles idées soit parce qu'ils ont été définis pour une architecture particulière.

COMPEL a été introduit par Tesler et Enea pour montrer la faisabilité d'un tel langage.

SAMPLE de Chamberlin [CHAM 71] est particulièrement adapté aux traitements numériques à haut degré de parallèlisme. Il apporte quelques aménagements à la définition initiale des langages à assignation unique en introduisant la notation *OLD* permettant dans un corps de boucle de référencer la valeur précédente d'une variable itérée. La valeur d'une variable itérée une fois fixée est unique pour toute la durée de l'itération courante.

ID (Irvine Data flow language) a été défini à l'université d'Irvine [AGP 78] et s'inspire du langage de base de Dennis. Un programme en ID est une *suite d'expressions* dont les quatre plus importantes sont les blocs, les expressions conditionnelles, les boucles et les applications de procédure. Etant supposé pouvoir être utilisé pour l'écriture de systèmes d'exploitation, des moyens de synchronisation et de gestion ont été inclus en particulier la notion de moniteur [AG 77]. Il existe des *variables simples* dont la valeur est représentée par une seule marque dans le graphe dirigé par les données ou des *variables "stream"* dont la valeur est représentée par une séquence ordonnée de marques, chacune portant une valeur simple. De plus l'entrée et la sortie d'une telle variable à travers un opérateur est complétement asynchrone, c'est à dire que toutes les marques d'entrée ne doivent pas être arrivées pour que des marques de sortie soient produites.

Le langage *SDG* défini à l'université de Newcastle est ainsi appelé parce que sa sémantique est décrite en utilisant des graphes orientés et structurés (Structured Directed Graph), c'est à dire des graphes

orientés dans lesquels on peut distinguer des sous graphes. La différence essentielle provient du fait qu'il est possible de choisir le mode d'activation d'un sous graphe, *activation par disponibilité ou par nécessité* par l'analyse de sa structure et de ses limites.

Le langage LAPSE s'inspire également des représentations graphiques. Il autorise les itérations et la *récursivité* [GWG 78]. Les variables dans le langage Lapse sont des variables simples. Elles peuvent être regroupées et sont alors déclarées comme enregistrement. La notation graphique utilise la notion de label qui permet la distinction des marques dans un graphe réentrant.

Le langage LAU [PLA 77] [SYR 76] a été défini par une équipe du CERT à Toulouse dans le cadre de l'étude d'un système complet, logiciel et matériel utilisant le concept d'assignation unique. Il n'est pas basé sur une représentation graphique et s'apparente aux langages de programmation habituels. Dans une boucle, un objet est décomposé en trois. *OLD* représente la valeur prise par la variable itérée au cours de l'itération précédente ; *New* représente la valeur pour l'itération courante ; *OUT* représente la valeur qui est connue à l'extérieur de la boucle.

1.4. Conclusions

Les nouveaux concepts qui viennent d'être définis remettent en cause les méthodes classiques de programmation. Ils présentent l'avantage d'expliciter d'une manière naturelle le parallèlisme, contenu dans un programme au moment de son exécution. Le programmeur n'a plus à charge de devoir le détecter au moment de l'écriture du programme. Par ailleurs ils permettent d'atteindre le parallèlisme maximum [URS 73].

Le concept de cadencement par les données et celui de l'assignation unique remettent également en cause l'architecture des machines. De nombreuses recherches ont été effectuées ou sont en cours à l'heure actuelle. Nous en citerons quelques unes dans la suite de ce chapître.

2) Méthodes matérielles d'exploitation du parallèlisme

Depuis quelques années, on a vu apparaître un grand nombre de projets qui ont permis l'introduction du parallèlisme au niveau matériel. Ceci est dû, pour une grande part à l'évolution de la technologie des composants. L'apparition et l'évolution des microprocesseurs permet d'envisager la réalisation de systèmes de plus en plus complexes par multiplication d'éléments intégrés simples et peu coûteux.

La facilité d'implantation de ces circuits permet l'interconnexion d'un grand nombre de ces circuits et d'atteindre des puissances de traitement comparable à celles des unités centrales de très gros systèmes.

2.1. Classification et critères de classification

Flynn a proposé une classification [FLYN 72] en prenant comme critères les flux d'instructions et de données. Il distingue quatre grandes classes de machines.

- *SISD* (Single Instruction, Single Data). Ce sont typiquement les monoprocesseurs, qui déroulent un flot d'instructions sur un flot de données.

- *MISD* (Multiple Instruction, Single Data). Il n'y a pas de réalisation matérielle de ce type de machine. Certains classent dans cette catégorie les machines pipelines.

- *SIMD* (Single Instruction, Multiple Data). Une machine de ce type déroule un flot d'instructions sur plusieurs flots de données, simultanément. La plus connue est certainement ILLIAC IV.

- *MIMD* (Multiple Instruction, Multiple Data). Plusieurs flots d'instructions se déroulent sur plusieurs flots de données.

Cette classification apparait insuffisante si on désire caractériser plus finement l'architecture d'une machine. Ceci est particulièrement vrai dans les structures multiprocesseurs. Il est nécessaire d'introduire de nouveaux critères de classement (MAZA 78). Les plus significatifs sont au nombre de quatre :

- *nature du séquencement.* On distingue les machines séquencées par les instructions (machines séquentielles classiques) et les machines séquencées par les données (machine data flow).

- *contrôle.* Ceci caractérise la manière dont s'effectue la coopération entre les processeurs, la répartition des taches, l'allocation des ressources et la synchhonisation. On distingue deux formes de contrôle totalement opposées : le *contrôle hiérarchique,* il existe un processeur maître qui assure le contrôle de l'ensemble de la machine ; le *contrôle coopératif,* dans ce mode de fonctionnement, les processeurs s'entendent entre eux pour se répartir le travail et se partager les ressources.

- *communications* entre les processeurs. Ceci caractérise les voies d'échanges entre les processeurs. Cet aspect sera étudié de manière plus approfondie dans le chapitre trois.

- *Spécialisation* ou *banalisation* des processeurs.

Malgré ces critères, il apparait encore difficile d'établir un classement. Bon nombre de machines aujourd'hui ne correspondent pas d'une manière aussi nette à tel ou tel critère.

2.2. Machines séquentielles

Une première manière d'exploiter le parallèlisme a été de multiplier les unités de traitement dans les machines classiques. Trois types de machines ont vu ainsi le jour.

2.2.1. Les machines pipelines

Le concept de pipeline représente une forme précise du parallèlisme qui consiste à proposer des moyens de réaliser une exécution simultanée de plusieurs tâches à partir d'une description série [CORD 78].

Dans une machine pipeline, un processus est décomposé en plusieurs sous processus, chacun d'entre eux est confié à une unité autonome et donc spécialisée. Par exemple le processus d'exécution d'une instruction peut se décomposer de la manière suivante :

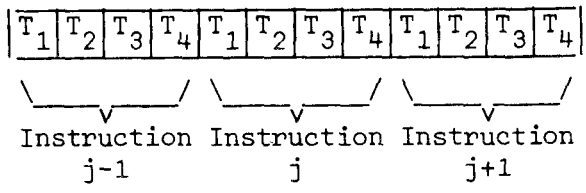
- T_1 : recherche de l'instruction

- T_2 : décodage

- T_3 : accés à l'opérande

- T_4 : exécution proprement dite

Si ce processus est exécuté par une unité unique, on obtient le diagramme des temps suivant.



S'il est possible de confier le travail à quatre unités, capables d'assurer chacune un des sous processus, le diagramme des temps devient

unités	T_1	T_2	T_3	T_4	T_5	T_6	T_7
exécution				Ij	Ij+1	Ij+2	Ij+3
accés opérande			Ij	Ij+1	Ij+2	Ij+3	
décodage		Ij	Ij+1	Ij+2	Ij+3		
recherche	Ij	Ij+1	Ij+2	Ij+3			

Cette technique est applicable à un travail qui peut être fait en n étapes. On augmente le débit d'un facteur n. Toutefois elle n'est efficace que si le travail doit se répéter souvent.

2.2.2. *Les machines tableaux*

Une machine tableau est une machine qui déroule un flot d'instructions sur plusieurs flots de données. Ces machines utilisent une unité de contrôle unique pour commander plusieurs unités de traitement qui exécutent toutes la même instruction sur des données différentes.

Illiac IV fut la première réalisation de ce type [BARN 68] [THUR 76]. Dans sa version complète, elle devait être composée de 256 processeurs élémentaires répartis en quatre sous tableaux de 64 processeurs. Chaque sous tableau

posséde une unité de contrôle qui décode les instructions et génère l'ensemble des signaux de contrôle pour les processeurs.

Illiac IV dispose d'une structure particulièrement adaptée au traitement matriciel et permet d'atteindre pour ce type de traitement des performances élevées. L'organisation des données dans un tel système pose toutefois quelques problèmes. L'utilisation d'une telle machine n'est intéressante que si le pourcentage de traitement parallèle est relativement élevé.

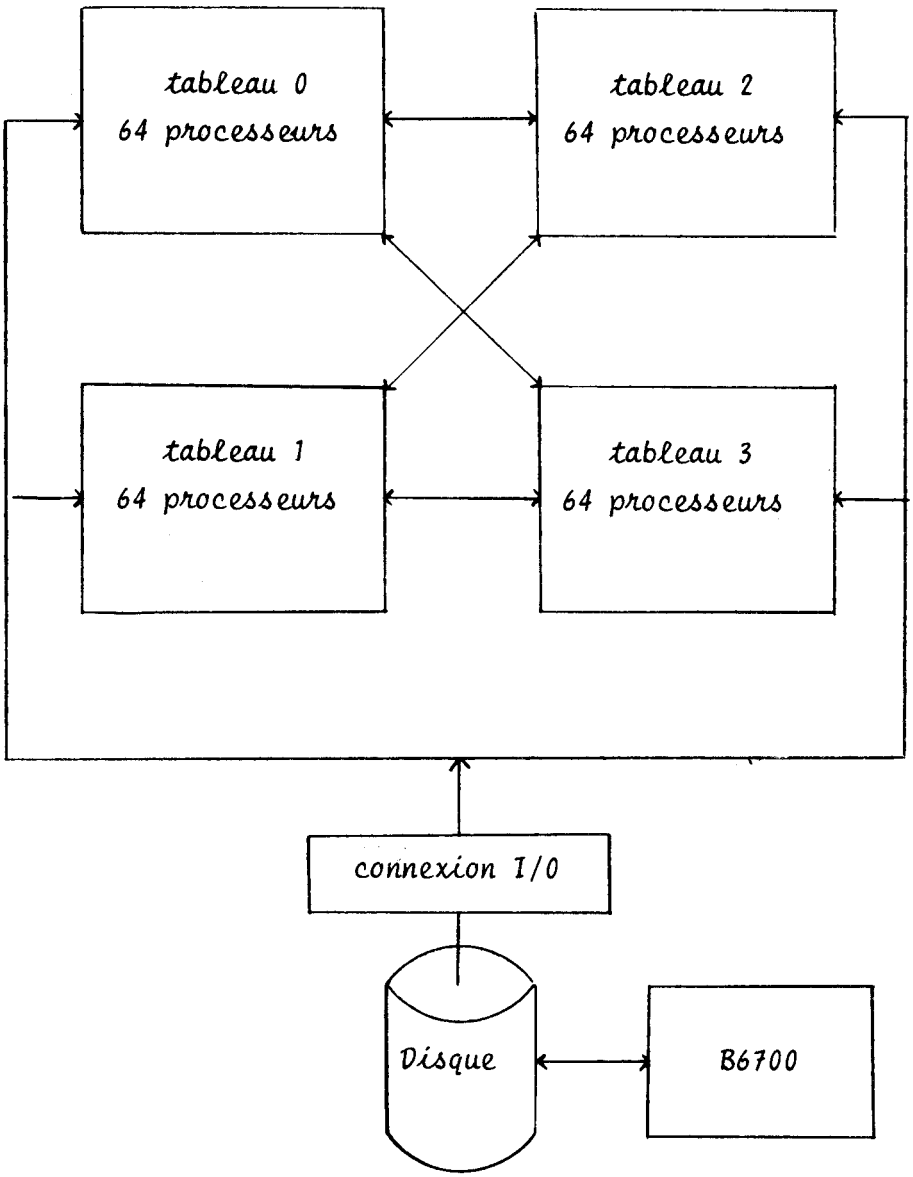


Fig 5 : Organisation générale de ILLIAC IV

Avec une construction prévue pour 1982 par Goodyear Aerospace, le système MPP (massively parallel processor) [KEN 80] a été conçu pour permettre le traitement d'images provenant de satellites. La figure 7 présente l'architecture générale du système.

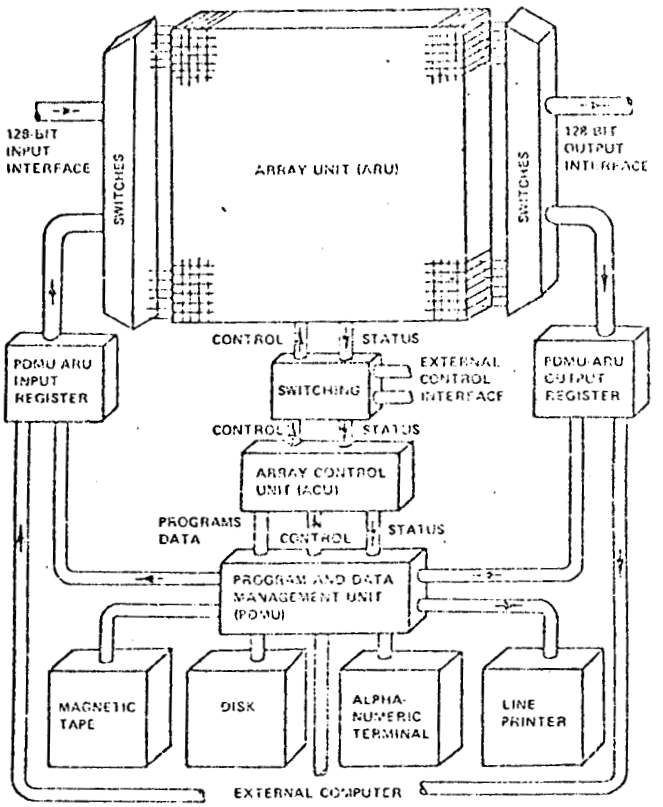


Fig 7 : Architecture du système MPP

Le système se décompose en un certain nombre d'unités qui sont les suivantes :

- l'ARU (array unit) comporte 16.384 processeurs élémentaires organisés sous la forme d'un tableau 128 x 128. Chaque processeur opère au niveau du bit afin de permettre une grande variété de taille d'opérandes. Ils peuvent communiquer avec leurs premiers voisins. La topologie du réseau d'interconnexion est similaire à celle utilisée dans ILLIAC IV. Le tableau est divisé en 32 groupes de 128 x 4. Un groupe additionnel étant prévu, pour le cas ou un processeur élémentaire serait en panne. Dans cette hypothèse, le groupe contenant ce processeur est déconnecté.

- L'ACU (array control unit) dont la fonction est d'assurer le contrôle des processeurs élémentaires. Elle se compose de trois unités qui fonctionnent en parallèle. L'une effectue le contrôle des processeurs pour des traitements

de tableaux,une deuxième assure le contrôle des entrées-sorties de l'ARU et la dernière exécute les traitements sur les scalaires.

- La PDMU (programme and data management unit) gère les flots d'instructions et de données pour le système.

- Les PDMU-ARU I/O Registers sont utilisés comme tampons pour les entrées-sorties de l'ARU mais permettent également le réordonnancement des tableaux de données.

Les performances annoncées pour le système sont les suivantes :

- addition de tableaux
  entiers 8 bits 6553 MOPS
  représentation flottante 32 bits 430 MOPS

- produits
  entiers 8 bits 1861 MOPS
  représentation flottante 32 bits 216 MOPS.

2.2.3. *Les architectures "multi-unités"*

C'est sans aucun doute dans ce domaine que l'on peut trouver le plus grand nombre de réalisations. Comme nous l'avons vu précédemment, dresser une liste exhaustive et établir un classement est un travail complexe et ne permet pas de définir une structure idéale. La réalisation de ces projets a été favorisée en grande partie, par le développement des micro-processeurs qui a permis d'abaisser le prix de revient de tels systèmes. Il est possible d'envisager des structures comportant un très grand nombre de microprocesseurs même si le taux d'utilisation de chacun est peu élevé.

Si un processeur se réduit à la notion d'unité centrale, nous pouvons définir deux tendances extrêmes :

- les *multiprocesseurs*. Ils comprennent un certain nombre d'unités centrales qui se partagent un même ensemble de mémoires et de périphériques. L'augmentation du nombre d'unités permet l'accroissement des performances. Cependant il existe généralement un goulot d'étranglement au niveau de la mémoire et au niveau des périphériques.

- Les *multiordinateurs* peuvent être caractérisés par le fait que chaque unité centrale dispose d'une mémoire locale formant ainsi un sous ensemble capable de fonctionner de manière autonome. Ces sous ensembles sont interconnectés par un réseau de communication.

Dire que toutes les architectures "multi-unités" appartiennent à l'une de ces deux catégories serait parfaitement illusoire. Il existe en effet un très grand nombre de variantes entre ces deux extrêmes. Ces structures ont le mérite d'avoir amélioré la fiabilité de systèmes informatiques. Il est possible de fonctionner en mode dégradé dans la mesure où plusieurs unités centrales sont capables d'effectuer le même travail.

Pour toutes les machines décrites ci-dessus, il existe des limites au degré de parallèlisme atteint dans le traitement d'une tâche.

- Les machines tableaux ne permettent de dérouler qu'un flot unique d'instructions et sont relativement spécialisées pour certaines classes de problèmes : traitement matriciel ou vectoriel.

- Les multiprocesseurs ont permis d'accroître les performances bien que le doublement d'une unité centrale n'entraine pas tout à fait le doublement de la puissance compte tenu des conflits supplémentaires qui peuvent se produire lors de l'accés aux ressources communes. Dans de tels systèmes, la mémoire devient rapidement un goulot d'étranglement et limite le degré de parallèlisme.

- Les multiordinateurs offrent sans aucun doute le meilleur potentiel d'exploitation du parallèlisme. Les problèmes se posent au niveau de la mise en oeuvre de ces machines : comment répartir le travail entre les différents processeurs ? Comment les coordonner ?

- Les machines séquentielles souffrent essentiellement du mode de contrôle qui est utilisé. Il impose l'expression des propriétés intrinséques du programme à l'aide du langage support et donc une description du parallélisme complétement figée avant l'exécution. Les machines data flow qui seront présentées dans le paragraphe suivant utilisent les relations de dépendances par les données et permettent la détection du parallèlisme à l'exécution.

2.3. Les machines Data flow

2.3.1. Les travaux du MIT [DEN 74]

Le modèle de base d'une machine capable d'interpréter les schémas data flow comporte une mémoire qui est un ensemble de cellules, chacune contenant une instruction et les opérandes de celle-ci. Chaque cellule est composée de trois parties :

- Un registre instruction qui détient le code opération et les adresses des registres auxquels les résultats de l'exécution doivent être communiqués.

- Deux registres opérandes.

Lorsqu'une instruction peut être exécutée c'est à dire que tous ses opérandes ont été définis, elle est prise en charge par l'*arbitre* qui la transmet à une *unité de traitement* en fonction du code opération. Les résultats sont communiqués aux registres opérandes destinataires par le distributeur. Les unités de traitement sont pipelinées de manière à accroître la vitesse de traitement.

Cette structure de base est complétée par une *unité de décision* et un *contrôleur* qui interpêtent les noeuds "portes" et "jonctions" associés aux instructions.

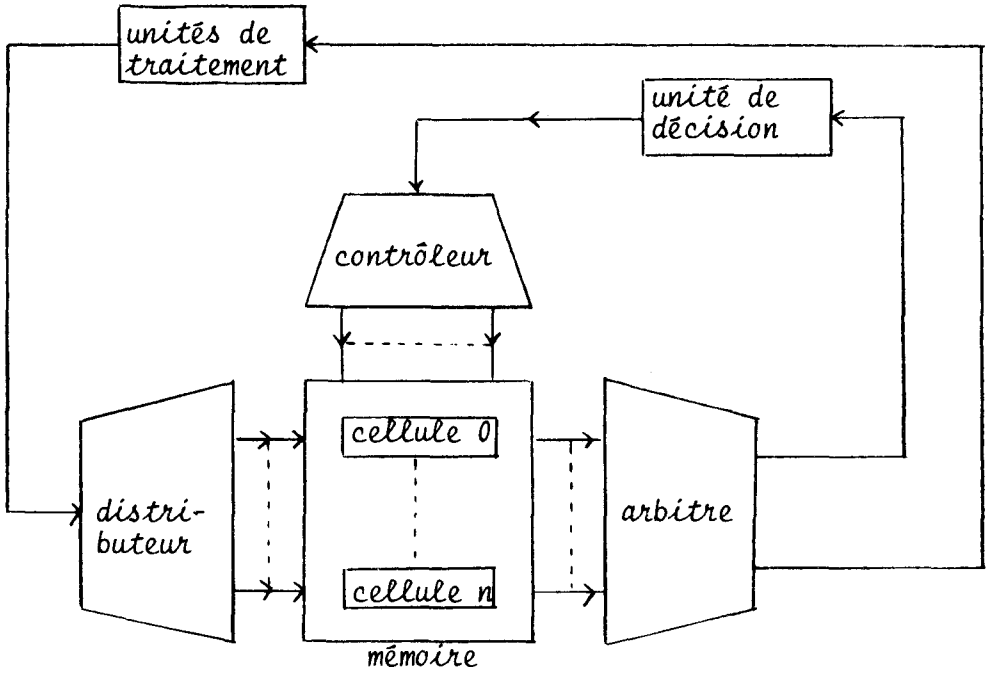


Fig 8 : Structure de base de la machine dirigée par les données du MIT

Dans une telle organisation, le programme complet doit résider dans la mémoire, c'est à dire qu'il doit exister autant de cellules que d'instructions dans le programme. Lorsqu'une instruction a été utilisée, elle peut disparaître. L'utilisation d'une mémoire à deux niveaux permet de limiter le nombre de cellules. Seules les instructions nécessaires à la progression du programme occupent les registres. Le reste du programme est stocké dans *la mémoire d'instructions*. L'unité *commande mémoire* assure les transferts de la mémoire instructions vers les cellules en fonction des demandes.

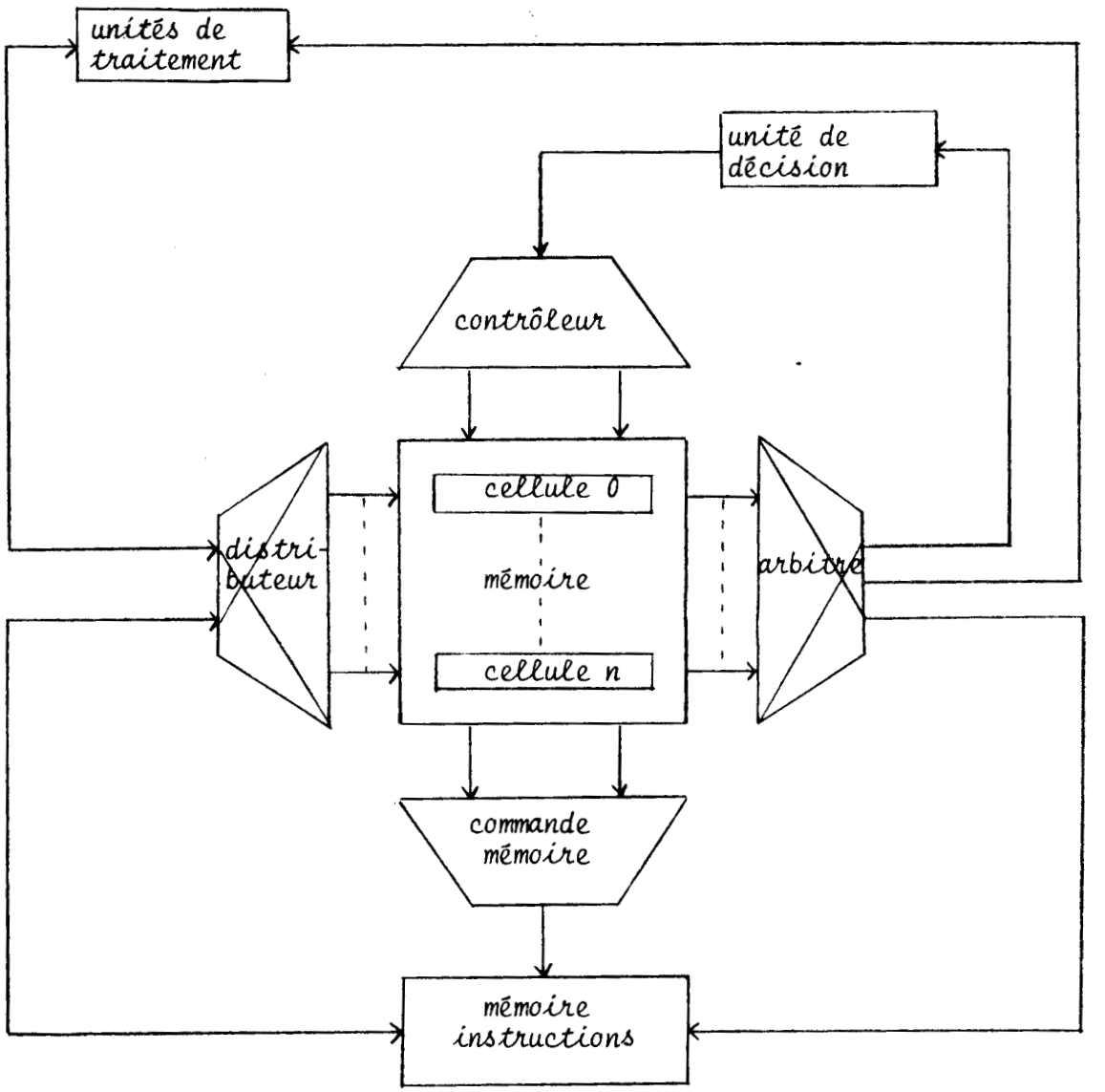


Fig 9 : Structure de base de la machine data flow du MIT avec mémoire auxiliaire

2.3.2. *Travaux de Rumbaugh*

Rumbaugh utilise le fait qu'un programme peut être décomposé en un ensemble de procédures, chacune pouvant être appelée une ou plusieurs fois, simultanément ou non au cours de l'exécution du programme.

La machine est composée de plusieurs unités asynchrones [RUM 77].

- Plusieurs *processeurs d'activation* exécutent le programme. Chaque processeur comprend :
  - . Une *mémoire locale* qui contient les instructions, les données et les compteurs de validation des instructions. Une instruction est exécutable lorsque son compteur passe à zéro.
  - . Une *unité de traitement pipeline* comportant un certain nombre d'opérateurs particuliers.
    Un *compteur d'activité* qui indique l'état du processeur.

- Le *gérant* qui crée, conserve et détruit les activations de procédure et les assigne aux processeurs pour l'exécution. Le gérant utilise trois tables :
  - . Une *table d'appel* qui contient les appels de procédures qui sont en attente.
  - . Une *table des processeurs* qui indique les activations de procédure qui sont en cours dans les différents processeurs.
  - . Une *table d'activation* qui pour chaque appel de procédure précise la procédure appelante et le lieu d'appel.

- Une *mémoire de structure* qui contient les structures de taille importante et qui ne peuvent résider dans les processeurs d'activation.

- Les contrôleurs de structures qui opèrent sur les structures en fonction des demandes des processeurs d'activation.

- La *mémoire programme* qui contient les procédures qui peuvent être appelées.

- La *mémoire rangement* qui contient temporairement les procédures inactives.

- Le *réseau d'échange* qui assure les échanges d'activation de procédure entre la mémoire programme, la mémoire rangement et les processeurs d'activation.

- Le *processeur d'entrée-sortie* qui assure l'exécution des procédures d'échange avec l'extérieur. Il peut accéder à la mémoire de procédure et faire des appels de procédures.

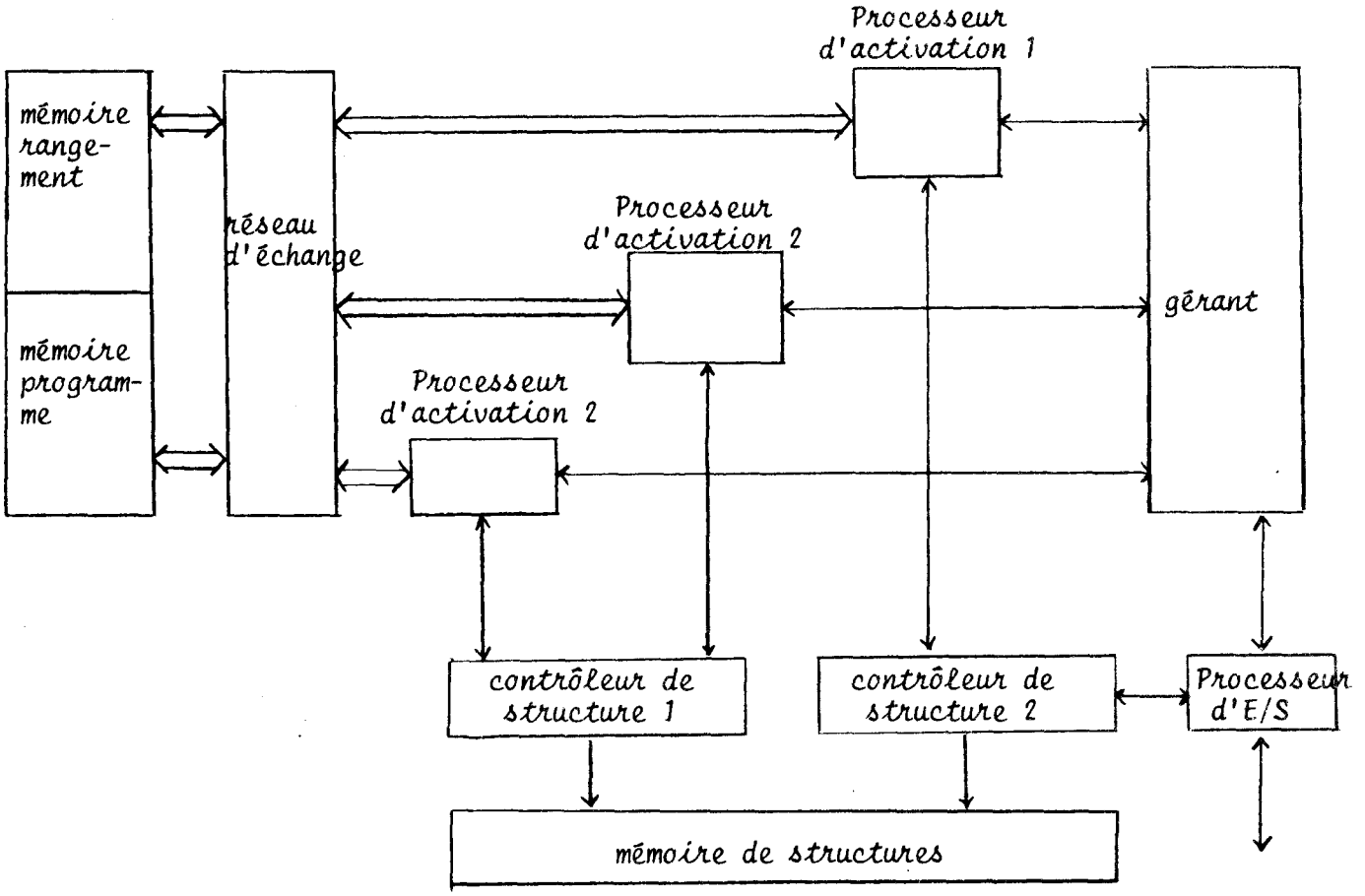


Fig 10 : Machine dirigée par les données de Rumbaugh

La présence d'un gérant qui assure un contrôle centralisé des processeurs risque de créer un goulot d'étranglement au niveau de l'allocation des procédures et donc d'entrainer une utilisation non optimale des processeurs. Malgré tout, il permet de connaître, à tout instant, l'activité de la machine par simple consultation des tables. Dans l'hypothèse où les processeurs pourraient s'allouer eux mêmes une tâche, il serait nécessaire de consulter chaque processeur pour connaître l'état de la machine. Le nombre de communications entre les processeurs et la mémoire est limité puisque les échanges ne sont effectués que pour des procédures, c'est à dire un ensemble d'instructions.

2.3.3. *Les travaux de l'université d'Irvine*

L'architecture de la machine est basée sur l'interpréteur de "débrouillement" (unraveling interpreter) qui a été développé à l'université d'Irvine [AG 77] [AGP 78]. La fonction de ce dernier est de générer le maximum de tâches parallèles à partir d'un programme ID. Il permet l'exécution simultanée d'appels distincts de la même procédure et l'expansion automatique des boucles.

La machine proposée est une *matrice de processeurs élémentaires* (PE) tous identiques. Afin de réduire les temps de communication, la machine peut se partitionner en domaines physiques comprenant chacun un certain nombre de processeurs élémentaires, de contrôleurs de mémoire et la partie associée du système de communication.

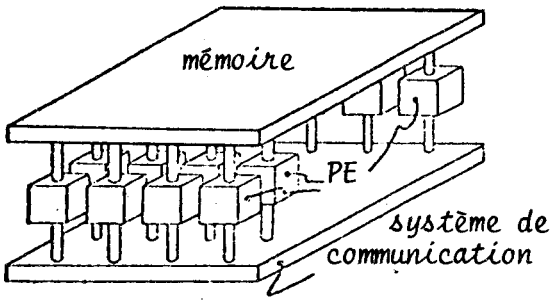


Fig 11 : Architecture proposée par l'université d'Irvine

Chaque domaine physique est composé de plusieurs sous domaines physiques dont l'organisation est donnée figure 12.

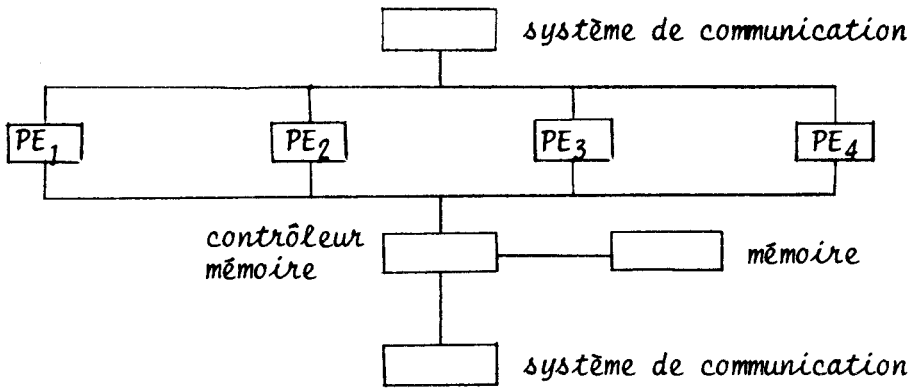


Fig 12 : Sous domaine physique

Un sous domaine comporte

- quatre *processeurs élémentaires* qui peuvent communiquer entre eux par l'intermédiaire d'un bus interne.

- une *mémoire locale* qui est gérée par le *contrôleur de mémoire*

- deux *systèmes de communication* , l'un permettant le transfert des résultats d'un sous-domaine vers un autre, le deuxième permettant des transferts entre les différents blocs de mémoire.

Chaque processeur élémentaire a une possibilité d'accés direct à la mémoire locale du sous-domaine et d'accés indirect à la mémoire locale des autres sous domaines par l'intermédiaire du contrôleur de mémoire. Toutefois pour limiter les conflits d'accés, le principe de redondance des informations est appliqué, c'est à dire que des informations peuvent être recopiées dans plusieurs blocs de mémoire. Ceci est particulièrement intéressant lorsqu'il s'agit de manipulation sur des structures.

Le système de communication qui permet le transfert des marques associées au résultat est formé par un ensemble de bus bidirectionnels. Les marques circulent dans le système de communication à la recherche d'un processeur libre ou d'un processeur qui les attend. Le partitionnement de la machine permet de réduire l'espace de circulation des marques et donc d'accroître la vitesse de traitement.

Ce type d'architecture est sans aucun doute, très intéressant. Un grand nombre de problèmes risque cependant de se poser au niveau de l'implémentation en particulier pour les systèmes de communication

2.3.4. *Travaux de l'Université de Newcastle*

L'objectif de l'université de Newcastle est de définir une machine de type MIMD universelle qui utilisent les meilleures caractéristiques des deux organisations flot de contrôle et flot de données.

Un modèle à flot de contrôle généralisé (generalized control flow) a été défini ainsi qu'un langage de haut niveau SDG qui est basé sur une représentation à l'aide de graphes orientés et structurés [TREL 78]. Dans

l'organisation GCF, les instructions de contrôle sont vues comme formant un graphe de contrôle orienté. A chaque instruction de contrôle sont associées plusieurs instructions. Elles permettent la synchronisation de l'exécution des instructions. Le modèle permet de supporter aussi bien une représentation d'un programme dirigé par les données, lorsqu'un haut degré de parallèlisme doit être exploité, qu'une représentation type Von Neumann, lorsqu'on utilise des langages conventionnels.

L'architecture de base utilise la notion de tâche. Une tâche est une instruction simple ou une procédure. La machine comporte trois unités :

- L'*unité de gestion des tâches* qui contient les noms, c'est à dire l'adresse des tâches qui doivent être traitées.

- L'*unité de traitement* qui comporte un grand nombre de processeurs élémentaires.

- L'*unité de mémoire* qui contient le programme et les données.

Lorsqu'un processeur élémentaire se libère, il extrait de l'unité de gestion des tâches, l'adresse d'une instruction à exécuter et recherche cette instruction dans la mémoire. Chaque instruction comporte : un code opération, les adresses des opérandes d'entrée, les adresses des opérandes de sortie, l'adresse des instructions suivantes.

Après avoir recherché les opérandes d'entrée, effectué l'opération demandée et rangé le résultat, le processeur place l'adresse de l'instruction suivante dans l'unité de gestion des tâches.

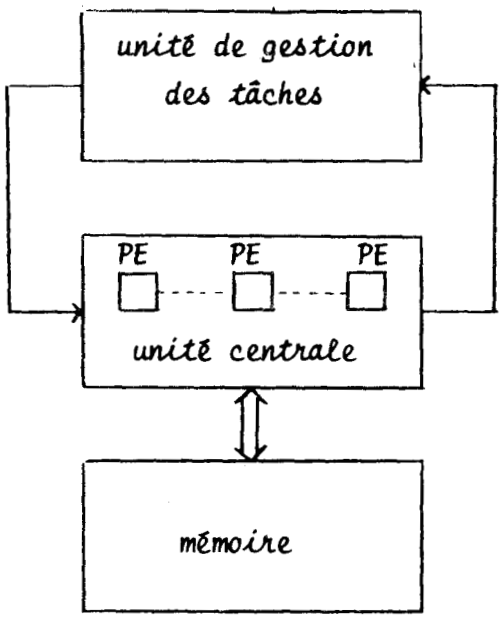


Fig 13 : Organisation CGF

Pour l'implémentation de ce modèle, l'unité de gestion des tâches est représentée par un tampon circulaire ou anneau qui est divisé en un certain nombre de segments chacun pouvant contenir un *paquet*.

Quatre types de paquets peuvent circuler le long de cet anneau.

- Le paquet *adresse* qui contient l'adresse d'une instruction suivante.

- Le paquet *incomplet* qui contient une instruction sans ses opérandes.

- Le paquet *complet* qui contient une instruction avec ses opérandes d'entrée.

- Le paquet *résultat* qui contient un opérande de sortie ainsi que l'adresse d'une instruction suivante.

Ces différents paquets sont fournis ou utilisés par trois unités placées autour de l'anneau :

- Le *processeur programme* qui est constituée d'une mémoire programme et d'un processeur d'accès, fournit un paquet *incomplet* en fonction de l'adresse contenue dans un paquet *adresse*.

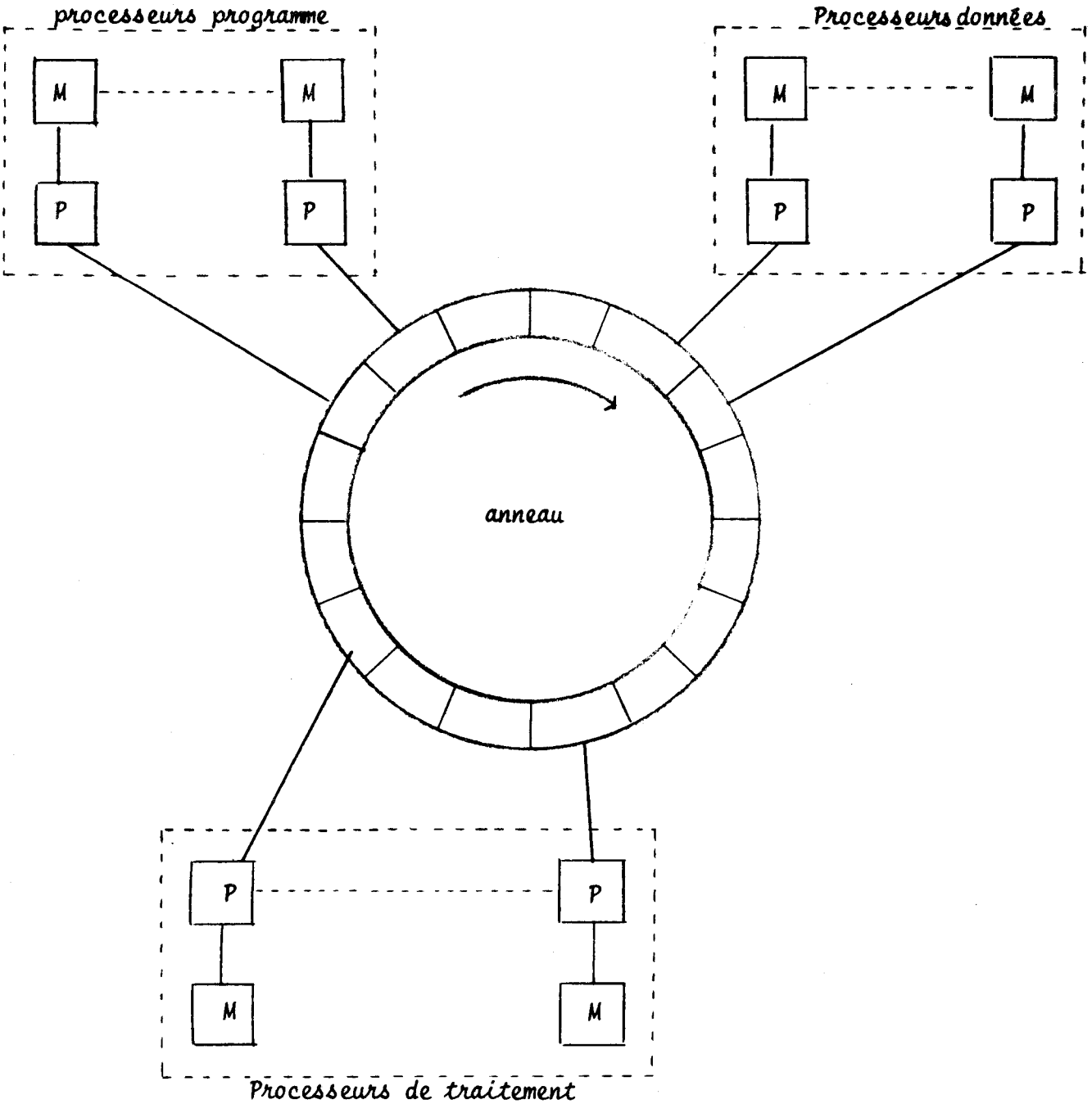


Fig 14 : Implémentation de l'architecture GCF

- Le *processeur données* qui est constitué d'une mémoire données et d'un processeur d'accés, traite deux types de paquets. A partir des paquets *incomplets*, il crée des paquets *complets* en fournissant à l'instruction ses opérandes d'entrée. Il traite également les *paquets résultats* auxquels il retire les opérandes de sortie pour en faire un paquet *adresse*. Ces opérandes de sortie sont stockés avant de devenir les opérandes d'entrée d'une prochaine instruction.

- Le *processeur de traitement* capture les paquets *complets*, exécute l'instruction sur les opérandes d'entrée et fournit en retour un paquet *résultat*.

Les deux premières unités constituent en fait l'unité de mémoire définie dans le modèle. Chacune des unités vues précédemment peuvent exister en plusieurs exemplaires et peuvent fonctionner de manière asynchrone.

Il existe une très grande analogie entre cette implémentation et celle que nous allons présenter par la suite pour notre modèle. Le choix d'un anneau en tant qu'outil de communication entre un ensemble d'unités asynchrones semble être bien adapté. Cependant l'utilisation d'un tel outil pour la transmission de paquets ne contenant qu'une instruction risque d'entrainer un temps de communication relativement grand par rapport au temps de traitement et une saturation de l'anneau.

2.3.5. *Travaux de l'université de Manchester*

L'architecture étudiée à Manchester est basée sur une structure circulaire le long de laquelle circulent les marques. A chaque valeur est associée un nom composé de deux parties. Une partie adresse de destination et une partie label qui sert dans les itérations [GWG 78].

La machine proposée se compose de cinq unités qui constituent une sortie de pipeline circulaire :

- La *mémoire programme* qui contient les instructions du programme. Chaque instruction comporte plusieurs champs. Le *champ fonction* spécifie l'opération à effectuer et, pour certaines instructions, une valeur littérale. Le *champ destination* peut se répéter plusieurs fois et précise l'adresse d'une instruction.

- L'*unité centrale* exécute les instructions qui ont reçu tous leurs opérandes d'entrée et fournit les résultats sous la forme d'un triplet (opérande, label, destination).

- L'*unité d'entrée-sortie* permet les communications avec l'extérieur.

- La *mémoire résultat* qui est une file. Les résultats peuvent y être stockés momentanément entre l'instant où ils sont produits et l'instant où ils sont consommés.

- La *mémoire d'appariement* (matching store unit) assure le regroupement des résultats qui seront ensuite communiqués à l'instruction qui les attend. Elle crée avec la mémoire programme des instructions exécutables.

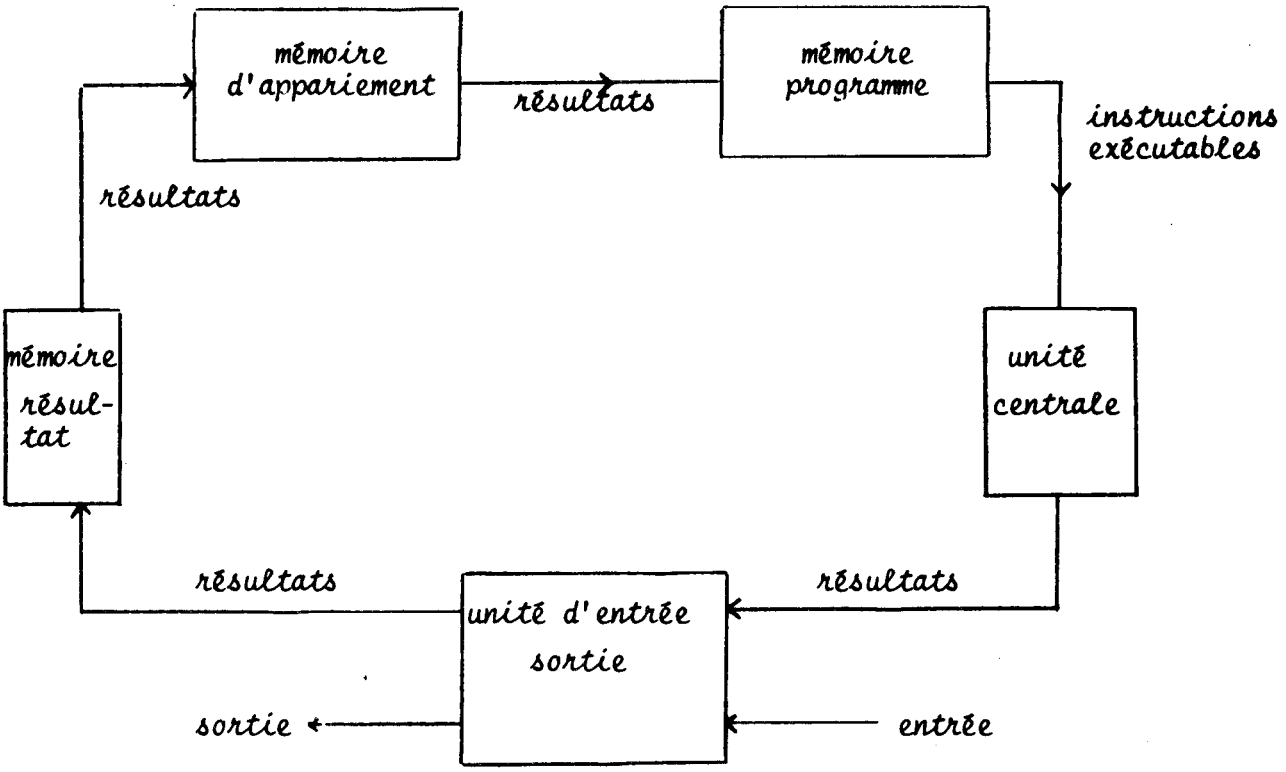


Fig 15 : Machine dirigée par les données de Manchester

Ici encore le contrôle dirigé par les données est appliqué au niveau des instructions. Il entraine un volume de communication considérable entre les différentes unités. L'introduction d'un nom propre associé à chaque opérande (label, destination) permet d'assurer de manière simple le transfert des résultats vers les instructions qui les attendent.

2.3.6. Les travaux du CERT

La machine dirigée par les données qui est en cours de réalisation à Toulouse, est basée sur le concept d'assignation unique [DUR 77] [SYR 76] [SYR 78]. Elle utilise la notion d'unités de programme.

Une unité de programme est un ensemble d'instructions ou d'unités de programme dont on peut lancer l'exécution et connaître la fin sur la base des données prêtes. Elle comporte un ensemble d'instructions, un ensemble de données d'entrée et un ensemble de données de sortie.

Un programme en langage machine se divise en trois zones. La première contient les instructions et les données proprement dites. La deuxième comporte trois bits de contrôle et la troisième est une zone de 1 bit qui précise que l'information contenue dans la première zone est une donnée.

La structure générale du processeur élémentaire d'assignation unique (PAU) appelé exécutant est représentée figure 15. Il se décompose en six unités :

- La *mémoire locale* contient l'ensemble ou une partie du programme ainsi que les données nécessaires à l'exécution des instructions.

- L'*unité de gestion* de la mémoire locale reçoit l'ensemble des requêtes et les satisfait au mieux des possibilités.

- L'*unité d'exécution* pipeline capable de traiter simultanément plusieurs instructions.

- L'*unité d'instructions* assure la recherche associative des instructions prêtes et la mise à jour des bits de contrôle. Les deux premiers indiquent l'état des opérandes d'entrée, le troisième est un indicateur dépendant de certaines conditions. Une instruction peut être exécutée lorsque les trois bits sont égaux à 1.

- L'*unité de données* détecte les violations d'assignation unique et la fin d'exécution d'unités de programme en utilisant les bits de la troisième zone.

- L'*unité d'état* connait l'état (actif ou passif) de toutes les unités de l'exécutant.

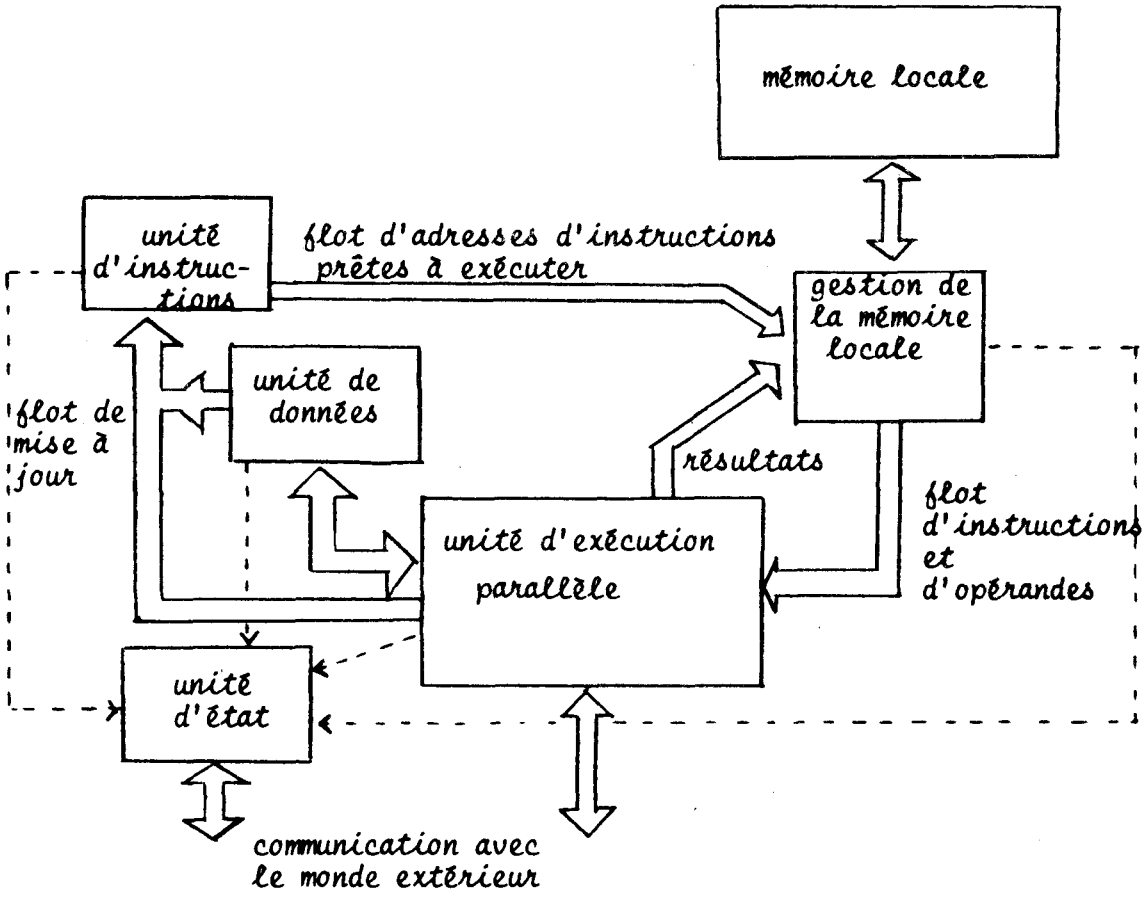


Fig 16 : Structure de PAU

Les communications entre les diverses parties de l'exécutant se font par l'intermédiaire de trois bus.

- Un bus bidirectionnel d'échange avec la mémoire.

- Un bus unidirectionnel d'échange avec l'unité d'instructions.

- Un bus bidirectionnel d'échange avec l'unité de données.

L'exécution d'une instruction se déroule de la manière suivante :

- *Détection d'une instruction* prête par l'unité d'instructions qui fournit son adresse à l'unité de gestion de la mémoire locale.

- *Lecture de l'instruction* par l'unité de gestion de la mémoire locale qui l'envoie à l'unité d'exécution.

- *Exécution de l'instruction* par l'unité d'exécution. Elle comporte une phase de lecture des opérandes d'entrée et une phase d'écriture du résultat en mémoire locale en plus de l'exécution proprement dite de l'instruction.

- *Propagation de la connaissance des résultats* qui est assurée au niveau de chaque unité composant le pipeline d'exécution.

- *Mise à 1 du bit associé au résultat* par l'intermédiaire d'une requête auprès de l'unité de donnée.

L'implémentation d'un processeur élémentaire a été étudiée précisément, simulée puis réalisée. La simulation donne des résultats très intéressants et montre que pour certains exemples, le taux de parallèlisme que l'on peut atteindre est très important.

L'ensemble des mises à jour impliquées par l'exécution d'une instruction est supporté par l'opérateur qui a exécuté cette instruction. Ce choix peut être génant dans la mesure où il immobilise l'opérateur. Ceci se justifie parce qu'un opérateur de mise à jour pourrait être un goulot d'étranglement. Ceci est dû en grande partie au fait que chaque instruction entraine des mises à jour, ceci ne serait peut être pas vrai si elles étaient effectuées après l'exécution complète d'une unité de programme.

La transmission des instructions et des opérandes d'entrée se fait séparément. Ceci entraine un surcroit de charge pour l'unité de gestion de la mémoire.

2.4. Conclusion

A travers les différents projets qui ont été analysés précédemment, on retrouve un certain nombre de traits communs autour desquels s'articulent toutes les approches du traitement dirigé par les données.

- Pour la plupart, une représentation graphique est utilisée soit comme modèle soit comme support d'expression d'un traitement. Elle présente l'avantage de faire apparaître clairement l'existence d'un parallèlisme potentiel dans un programme alors que les langages et les représentations conventionnelles ne permettent pas de le mettre en évidence.

- Toutes les études menées sur le traitement dirigé par les données conduisent à considérer en priorité les dépendances qui existent entre les données. Ceci entraine l'étude d'organisations mémoires originales dans lesquelles la notion d'adressage tend à disparaître. De réels supports associatifs permettraient sans aucun doute de résoudre un grand nombre de difficultés.

- Le concept d'assignation unique qui n'est pas directement lié au traitement dirigé par les données apparait comme un moyen de simplifier le contrôle. Le fait d'associer une valeur unique à un nom oblige non plus à gérer un espace d'adresses comme dans les machines classiques mais un espace de noms.

- Les architectures dirigées par les données semblent très bien adaptées au traitement parallèle, ceci parce qu'elles sont capables de détecter à l'exécution, les instructions qui peuvent être exécutées simultanément. Toutefois l'exploitation du parallèlisme au niveau des instructions entraine un volume de communications considérable dans le système et nécessite la définition d'opérateurs élémentaires.

Ces différentes remarques nous ont conduit à définir un modèle de machine à traitement dirigé par les données (MAUD) qui présente quelques aspects originaux vis à vis des architectures décrites précédemment.

En particulier les concepts de traitement dirigé par les données et d'assignation unique ne sont pas appliqués à l'échelle des instructions mais à celle d'un ensemble d'instructions qui sera appelé *Bloc*. Ceci offre un grand nombre d'avantages parmi lesquels :

- Grande puissance de traitement sans qu'il faille mettre en oeuvre une technologie spécifique.

- Emploi des microprocesseurs actuels.

- Programmation en grande partie classique.

- Introduction d'un contrôle dynamique

- Diminution du volume de communications.

CHAPITRE II

M A U D

MACHINE D'ASSIGNATION UNIQUE DYNAMIQUE

1) *Introduction*

Presque toutes les recherches sur les architectures data flow citées dans le chapitre précédent appliquent l'analyse d'un programme au niveau des relations entre instructions. Cette hypothèse est la plus simple et la plus naturelle. Elle se révèle cependant, peu compatible avec les points suivants.

1.1. *Volume des communications*

Un système multi-processeurs dans lequel la répartition des tâches se fait sans tenir compte des relations locales entre les données, impose que les processeurs soient en mesure, à chaque instruction, de communiquer, directement ou par l'intermédiaire d'une mémoire, opérandes et résultats.

L'outil de communication capable d'alimenter à ce rythme un nombre élevé de processeurs risque de constituer le goulot d'étranglement du système. Ceci est d'autant plus sensible que les accés à la mémoire doivent se plier à un modéle de structure de données de type graphe et que le traitement de ces structures risque de consommer un temps non négligeable.

1.2. *Dynamique du contrôle*

La structuration de l'ensemble des données sous forme d'un graphe exige la référence permanente et non ambigüe à un système d'accés par le nom. Ceci entraine une contrainte de dénomination figée des objets, élaborée au cours de la rédaction du programme.

Les efforts réalisés pour introduire quelques éléments dynamiques ont permis, au prix d'efforts complexes et de portée limitée, de supporter les boucles et les tableaux mais n'autorisent pas la création ou l'insertion de nouveaux objets (en particulier les procédures) dans le programme.

1.3. *La compatibilité technologique*

L'évolution de la micro-informatique de ces dernières années et les prévisions pour les années à venir, ne laissent que peu de place à des architectures de processeurs autres que celles dirigées par les instructions.

Un projet d'architecture qui voudrait rester réaliste, ne peut ignorer ce point et doit donc rejeter toute approche qui ne pourrait confier à des microprocesseurs classiques, une part importante de la charge globale.

2) Assignation unique par blocs

2.1. Définitions

Un programme pour Maud se compose d'un certain nombre d'entités appelées *blocs* [LEC 79a] [LEC 79b]. Un *bloc* contient un *ensemble d'instructions* et *les objets utilisés* par ces instructions.

Les blocs ne peuvent communiquer entre eux que par l'intermédiaire d'un nombre limité d'objets :

- Les *objets d'entrée* qui conditionnent l'exécution d'un bloc.

- Les *objets de sortie* qui sont calculés par les instructions d'un bloc et qui sont utilisés comme objets d'entrée par d'autres blocs.

La *règle d'assignation unique* est appliquée au niveau des blocs c'est à dire qu'un objet de sortie peut recevoir au plus une valeur pendant toute la durée d'exécution d'un programme. L'application de cette règle permet :

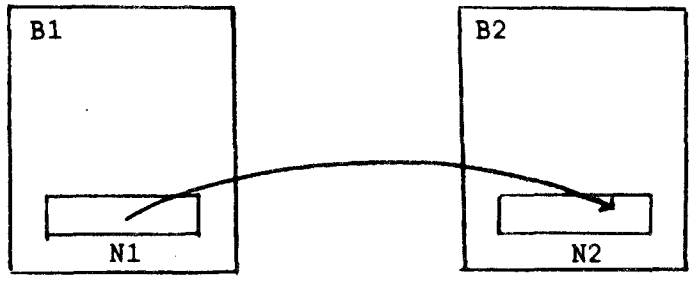
- L'expression des *dépendances entre blocs* et par conséquent la détection du parallèlisme potentiel à l'exécution.

- L'utilisation d'un *mécanisme de contrôle dirigé par les données* : un bloc peut être exécuté lorsque tous ses objets d'entrée ont reçu une valeur.

- Un *comportement déterministe* du programme : un objet de sortie qui a été calculé ne peut plus être modifié.

2.2. Communication entre blocs

Les communications entre les blocs sont réalisées en utilisant les objets d'entrée et les objets de sortie. Une valeur calculée par un bloc B1 doit être transmise au bloc B2.



Tous les objets utilisés dans Maud sont désignés par des *noms*. Les noms utilisés à l'intérieur d'un bloc (N1, N2) sont appelés noms internes et ne peuvent être utilisés à l'extérieur d'un bloc. Les communications se font en attribuant à chaque objet de sortie un nom particulier appelé *nom de communication*. Ceux-ci sont inconnus à l'intérieur des blocs.

Les noms sont gérés par l'intermédiaire de *domaines*. Un domaine est un ensemble de couples (Ni, Vi) où Ni est un nom et Vi la valeur associée. Il existe trois types de domaines :

- Les *domaines d'entrée* qui contiennent les noms de communication et les valeurs des objets d'entrée d'un bloc.

- Les *domaines de sortie* qui contiennent les noms de communication et les valeurs des objets de sortie d'un bloc.

- Les *domaines propres* qui contiennent les noms internes d'un bloc et les valeurs associées.

2.3. Structure d'un bloc

Un bloc se décompose en trois parties :

- Une partie *communication* réservée à Maud qui comprend le domaine d'entrée et le domaine de sortie.

- Une partie *interne* qui est composée d'un ensemble d'instructions et des objets qu'elles utilisent localement qui sont regroupés dans le domaine propre.

- Une partie *interface* qui permet d'assurer la transmission des valeurs entre la partie communication et la partie interne. Elle est constituée d'une *liste de correspondance* qui est un ensemble de couples (Ni, Nj) où Ni est un nom de communication et Nj un nom interne. On distingue la *liste de correspondance d'entrée* et la *liste de correspondance de sortie*.

Un bloc apparait comme un ensemble indivisible d'instructions qui peuvent être exécutées lorsque le *domaine d'entrée* du bloc est *complet*, c'est à dire que chaque nom de communication qui se trouve dans le domaine d'entrée a reçu une valeur. On a donc bien un mécanisme de contrôle de type dirigé par les données. Un domaine de sortie est fourni comme résultat de l'exécution. Un bloc peut, par conséquent se trouver dans trois états.

- Il est *en cours d'exécution*, il a été pris en charge par un opérateur capable d'exécuter les instructions.

- Il est *exécutable* lorsque son domaine d'entrée est complet et qu'il est en attente d'un opérateur de traitement.

- S'il existe dans le domaine d'entrée, un nom de communication qui n'a pas encore reçu sa valeur, il est dit *en attente*.

2.4. Création dynamique de blocs

Un premier modèle dit modèle statique [LEC 79c] a été défini mais il n'offre pas un maximum de possibilités au niveau de l'exploitation du parallèlisme.

Le découpage en blocs est complétement figé avant l'exécution du programme et ne permet pas dans la forme initiale l'expression des boucles de tailles importantes et les appels de procédure. D'autre part le modèle statique n'autorise pas la prise en compte des paramètres en vue d'une meilleure utilisation. L'écriture d'un corps de boucle sous la forme d'un bloc entraine nécessairement la création d'un nombre de blocs égal au nombre d'itérations, quand celui ci peut être déterminé avant l'exécution.

La plupart des inconvénients proviennent du fait qu'il n'est pas possible de créer de nouveaux blocs au cours de l'exécution d'un programme.

Le caractère dynamique est introduit dans le modèle par l'utilisation de deux nouvelles primitives au niveau de l'exécution des blocs : *execbloc* et *attendre*.

* *fonction des primitives*

La primitive *execbloc* permet d'exécuter un bloc dont un modèle se trouve dans une bibliothèque de blocs. Ce modèle est composé du code des objets locaux. Cette primitive permet, en particulier d'éviter l'écriture de blocs identiques aux objets d'entrée et objets de sortie prêts et d'exécuter des programmes non exécutables dans le modèle statique. Toutefois la primitive *execbloc* n'est pas qu'un simple appel de procédure. Elle est analogue à un FORK. L'exécution du bloc appelant se poursuit en parallèle avec celle du bloc appelé.

Ce dernier point implique l'existence d'une autre primitive permettant d'attendre que les blocs demandés lors de l'exécution des primitives *execbloc* aient élaborés leurs objets de sortie. C'est le rôle de la primitive *attendre*.

* *format des primitives*

Les paramètres nécessaires à la primitive *execbloc* sont au nombre de trois :

- Un nom de bloc contenu dans la bibliothèque

- Une liste de paramètres d'entrée qui sont des noms internes ou des expressions qui calculées fourniront une valeur.

- Une liste de paramètres de sortie qui sont des noms internes.

Pour la primitive *attendre*, le seul paramètre nécessaire est une liste de noms internes. Ces noms sont obligatoirement apparus dans une liste de paramètres de sortie d'un *execbloc*.

2.5. Exemples de programmes pour Maud

Exemple 1 : programme de calcul de CH(X) et de SH(X)
écriture algorithmique standard

```
Si X = 1 alors CH(X) := 1 ;
              SH(X) := 0 ;
         sinon CH(X) := (EXP(X) + EXP(-X))/ 2 ;
              SH(X) := (EXP(X) - EXP(-X))/ 2 ;
         fsi
```

Ce programme peut s'écrire de plusieurs manières pour Maud :

- première solution : programme P1

```
BLOC B0 ; entrée X ; sortie CH, SH ;
     si X = 0 alors CH := 1 ; SH := 0 ;
              sinon CH := (exp(X) + exp(-X)) / 2 ;
                    SH := (exp(X) - exp(-X)) / 2 ;
     fsi
fin bloc
```

Pour commencer l'exécution la valeur de X doit être fournie. Le programme P1 a été écrit de manière classique et n'utilise pas les possibilités de parallèlisme

- deuxième solution ; programme P2

```
BLOC B0 ; entrée X ; sortie X1, X2 ;
        X := X1 ;
        X := X2 ;
fin bloc

BLOC B1 ; entrée X1 ; sortie RP ;
     si X1 = 0 alors RP := 1 ;
              sinon RP := exp (X1) ;
     fsi
fin bloc

BLOC B2 ; entrée X2 ; sortie RM ;
     si X2 = 0 alors RM := 1 ;
              sinon RM := exp (-X2) ;
     fsi
fin bloc
```

```
BLOC B3 ; entrée RP, RM ; sortie CH, SH ;
        CH := (RP + RM) / 2 ;
        SH := (RP - RM) / 2 ;
fin bloc
```

Le programme est découpé en plusieurs blocs. B1 et B2 peuvent s'exécuter en parallèle. Les trois blocs doivent exister dans le système avant l'exécution.

- troisième solution : programme P3

```
BLOC B0 ; entrée X ; sortie CH, SH ;
local Y, Z ;
     si X ≠ 0 alors execbloc (B1 ; X ; Y) ;
                    execbloc (B1 ; -X ; Z) ;
                    attendre (Y, Z) ;
                    CH := (Y+Z) / 2 ;
                    SH := (Y-Z) / 2 ;
               sinon CH := 1 ;
                    SH := 0 ;
     fsi
fin bloc
```

Le bloc B1 ne sera extrait de la bibliothèque que si la valeur fournie par X est différente de 1. Dans ce cas la primitive *attendre* permet de suspendre l'exécution du bloc B0 en attendant que Y et Z soient évalués. Les deux copies du bloc B1 peuvent s'exécuter en parallèle.

exemple 2 : Programme comportant une boucle

```
Pour i jusqu'à n faire
       t(i) = F(i) ;
       fin faire
         programme d'utilisation des t(i)
```

Ce programme peut s'écrire

```
BLOC B0 ; entrée n, sortie ... ;
local ... ;
     Pour i jusqu'à n faire
          execbloc (B1 ; i ; t(i)) ;
     fin faire
     attendre (tous les t(i))
     autres opérations utilisant les t(i)
fin bloc
```

```
BLOC B1 ; entrée X ; sortie Y ;
        Y := F(X) ;
fin bloc
```

Les copies du bloc B1 peuvent s'exécuter en parallèle et l'exécution du bloc B0 sera reprise lorsque tous les t(i) auront été évalués et transmis au bloc B0.

3) *Présentation du modèle*

Dans le modèle, il existe cinq ensembles d'objets.

- l'*ensemble des blocs exécutables* (ensemble X) qui contient des blocs dont le domaine d'entrée est complet et qui n'ont pas encore été pris en charge par un opérateur de traitement. L'existence de cet ensemble est dû au fait que le nombre d'opérateurs dans une réalisation matérielle n'est pas illimité.

- L'*ensemble des blocs en attente* (ensemble A). Les blocs qui s'y trouvent sont ceux dont le domaine d'entrée est incomplet et qui attendent des valeurs fournies comme résultats d'exécution d'autres blocs.

- L'*ensemble des objets de sortie* (ensemble S) qui reçoit les domaines de sortie produits par l'exécution complète des blocs. Il contient les couples (nom, valeur) qui seront utilisés un à un pour compléter les domaines d'entrée des blocs en attente.

- L'*ensemble des demandes d'exécution* qui sont fournies par les opérateurs de traitement après l'exécution d'une primitive *execbloc*. Elles contiennent l'ensemble des informations nécessaires à la création d'un nouveau bloc, c'est à dire un nom de bloc permettant de désigner un bloc dans la bibliothéque, un domaine d'entrée, et un domaine de sortie dans lequel les valeurs associées aux noms de communication sont non évaluées parce qu'ils sont destinés à recevoir les résultats de l'exécution du bloc.

- La *bibliothèque* contient les modèles des blocs qui pourront être introduits dans le système suite à un *execbloc*. Ils comprennent un nom permettant de les identifier, la partie interne du bloc, c'est à dire les instructions et les objets locaux, et deux listes de noms internes qui correspondent aux objets d'entrée et de sortie.

Trois types d'opérateurs peuvent agir sur ces ensembles. Leur fonctionnement sera examiné plus en détail par la suite. Il s'agit

- des *opérateurs de traitement*. Ils assurent l'exécution des blocs exécutables. Ils peuvent fournir un domaine de sortie lorsque l'exécution complète du bloc est terminée ou une demande d'exécution après le traitement d'une primitive *execbloc* ou un bloc en attente après une primitive *attendre*.

- L'*opérateur de mise à jour* assure le transfert des valeurs contenues dans l'ensemble des objets de sortie vers les domaines d'entrée des blocs en attente et génère ainsi des blocs exécutables.

- Le *processeur constructeur* gère la bibliothèque de blocs et crée à partir des demandes d'exécution des nouveaux blocs qui sont placés dans l'ensemble des blocs en attente.

Le parallèlisme résulte de l'existence de plusieurs opérateurs de traitement capables de fonctionner simultanément et de façon autonome et de la structure pipeline du modèle. Le choix de transmettre à un opérateur, les instructions et les données sous la forme d'un paquet permet d'exécuter un bloc dès sa prise en charge par un opérateur, sans avoir à faire référence à une autre unité.

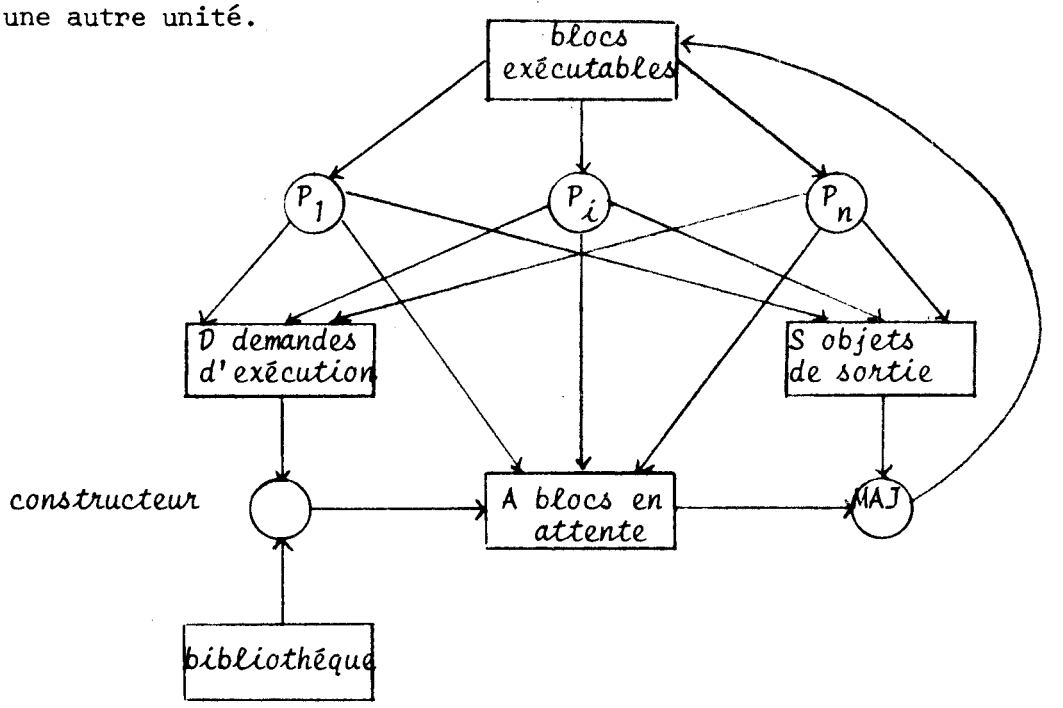


Fig 1 : MAUD : Modèle dynamique

4) Fonctionnement du système

4.1. Les opérateurs de traitement

La fonction des opérateurs de traitement Pi est la prise en charge des blocs exécutables qui se trouvent dans l'ensemble X. Ce traitement implique l'existence de deux primitives particulières qui auront pour but d'assurer la transmission des valeurs du domaine d'entrée vers le domaine propre avant l'exécution des instructions et la transmission des valeurs du domaine propre vers le domaine de sortie à la fin de l'exécution. Ces deux primitives s'appellent respectivement *début de bloc* et *fin de bloc*.

L'opérateur de traitement Pi extrait un bloc exécutable de l'ensemble X. Puis il exécute la primitive *début de bloc*. Ensuite il exécute la partie interne du bloc. S'il existe une primitive *execbloc*, son exécution entraine le dépot d'une demande d'exécution dans l'ensemble D, tandis que la primitive *attendre* entraine le dépot d'un bloc en attente dans l'ensemble A et la libération de l'opérateur. Lorsqu'il a complétement terminé l'exécution d'un bloc, il exécute la primitive *fin de bloc*, il dépose dans l'ensemble S un domaine de sortie et se libère. Lorsqu'un opérateur est à nouveau libre, il recherche un autre bloc exécutable dans l'ensemble X. S'il n'y en a pas il poursuit sa recherche jusqu'à ce qu'il en trouve un, il l'extrait alors de l'ensemble X.

- *Primitive début de bloc*

La primitive *début de bloc* permet de transmettre les valeurs qui se trouvent dans le domaine d'entrée vers le domaine propre. Pour réaliser cette transmission, l'opérateur utilise la liste de correspondance d'entrée qui associe un nom interne aux noms de communication des objets d'entrée. Pour chaque valeur d'un nom de communication Nc transmise vers un nom interne Ni le couple correspondant (Nc, Ni) est retiré de la liste de correspondance.

- *Primitive execbloc*

Les paramètres de la primitive *execbloc* servent à construire une demande d'exécution qui comporte le domaine d'entrée et le domaine de sortie du bloc demandé ainsi que le nom du bloc demandé. Lors de la création du domaine de sortie, il est nécessaire de modifier la liste de correspondance d'entrée du bloc appelant de manière à associer les noms de communication qui

fourniront les résultats du bloc appelé aux noms internes qui les attendent. Les transmissions des noms de communication à travers les blocs entrainent des modifications de la liste de correspondance de sortie [LEC 79c].

- *Primitive attendre*

La primitive *attendre* a pour but d'arrêter l'exécution d'un bloc en attendant les valeurs calculées par d'autres blocs. Il est donc nécessaire de recréer un nouveau domaine d'entrée pour le bloc. Les noms de communication qui ont été associés aux noms internes attendus, se trouvent dans la liste de correspondance d'entrée. Lorsque le domaine d'entrée a été modifié, le bloc en attente est rangé dans l'ensemble A et l'opérateur de traitement se libère.

- *Primitive fin de bloc*

Cette primitive effectue la transmission des objets de sortie de la partie interne du bloc vers la partie communication et range le domaine de sortie dans l'ensemble S. Pour réaliser cette opération, l'opérateur utilise la liste de correspondance de sortie. Cette liste ne doit pas être modifiée comme dans le cas de la primitive *execbloc*, en effet l'exécution du bloc est terminée et le bloc peut être détruit. Après le traitement de la primitive *fin de bloc*, l'opérateur de traitement peut se libérer.

4.2. L'opérateur mise à jour

L'opérateur mise à jour doit remplir deux fonctions. La première consiste à mettre à jour les domaines d'entrée des blocs en attente à l'aide des couples (Nc, Vj) se trouvant dans l'ensemble S. Pour chaque nom de communication Nc apparaissant dans le domaine d'entrée d'un bloc en attente, il recherche un couple (Nc Vj) appartenant à l'ensemble S. Si celui ci existe il place la valeur Vj dans le domaine d'entrée.

Afin de limiter la taille de l'ensemble S, une contrainte quant à l'utilisation des objets apparaissant dans les domaines de sortie a été introduite : C'est l'*utilisation unique* des valeurs. Un objet ne peut être utilisé qu'une fois et une seule. Ceci permet également de simplifier la tâche de l'opérateur mise à jour car il ne doit pas détecter le fait qu'un objet ne sera plus utilisé. Lors de la transmission d'une valeur, il peut détruire dans l'ensemble S le nom de communication et la valeur associée et ainsi récupérer le nom pour une communication ultérieure.

La deuxième tâche de l'opérateur consiste à détecter les blocs en attente dont le domaine d'entrée est complet, de les transférer dans l'ensemble X et de les supprimer de l'ensemble A.

4.3. *L'opérateur constructeur*

Son rôle est de construire un bloc à partir d'une demande d'exécution trouvée dans l'ensemble D et de ranger le bloc ainsi créé dans l'ensemble A. La partie communication du bloc se trouve dans la demande d'exécution et la partie interne se trouve dans le modèle du bloc existant en bibliothèque, seule la partie interface, c'est à dire les listes de correspondance sont à créer.

4.4. *Déroulement d'un programme*

Tous les blocs présents au début d'un programme sont des blocs en attente, toutefois l'un d'entre eux au moins doit avoir un domaine d'entrée complet pour que le traitement puisse commencer. Après passage devant l'opérateur mise à jour, ce bloc deviendra exécutable et sera pris en charge par un opérateur de traitement.

L'exemple ci-dessous illustre le déroulement d'un programme en décrivant l'évolution des parties interface et communication du bloc et l'évolution des ensembles d'objets.

```
BLOC B
DE (Nc1, V1), (Nc2, V2) ; DS (Nc, -) ;
LCE (NC1, A), (Nc2, B) ; LCS (Nc, X) ;
DP (A -), (B, -), (X, -), (Y, VY), (Z,VZ), (R1 -), (R2 -) ;
.
.
.
exec bloc (B1 ; Vy ; R1) ;
exec bloc (B2 ; Vz ; R2) ;
attendre (R2) ;
.
.
.
attendre (R1) ;
.
.
.
fin bloc ;
```

Instant du déroulement	DE	LCE	LCS	DS
Prise en charge de B	(Nc1,V1), (Nc2, V2)	(Nc1, A), (Nc2, B)	(Nc, X)	(Nc, -)
Début d'exécution	-	-	(Nc, X)	(Nc, -)
Après le 1er exec bloc	-	(NcA, R1)	(Nc, X)	(Nc, -)
Après le 2ème exec bloc	-	(NcA, R1),(NcB, R2)	(Nc, X)	(Nc, -)
Après le 1er attendre	(NcB , -)	(NcA, R1),(NcB, R2)	(Nc, X)	(Nc, -)
Reprise du bloc B	(NcB , VB)	(NcA, R1),(NcB, R2)	(Nc, X)	(Nc, -)
Suite d'exécution	-	(NcA, R1)	(Nc, X)	(Nc, -)
Après le 2ème attendre	(NcA , -)	(NcA, R1)	(Nc, X)	(Nc, -)
Reprise du bloc B	(NcA , VA)	(NcA, R1)	(Nc, X)	(Nc, -)
Suite d'exécution	-	-	(Nc, X)	(Nc, -)
Fin d'exécution	-	-	(Nc, X)	(Nc, -)

5) *Caractéristiques d'une architecture matérielle*

L'originalité du modèle par rapport aux architectures classiques ou en cours d'étude, justifie l'étude d'une implémentation et sa réalisation. Les objectifs qui nous ont guidés sont les suivants.

- Proposer une architecture peu couteuse et qui ne nécessite, dans la mesure du possible, que l'emploi de composants classiques (microprocesseurs standards).

- Le système doit être facilement extensible. L'augmentation du nombre d'opérateurs ne doit pas entrainer des modifications trop importantes au niveau de l'organisation de l'ensemble.

- La gestion de mémoire doit être relativement simple avec le moins de problèmes de synchronisation possible.

Au niveau du matériel, la réalisation des opérateurs peut se faire de plusieurs manières et en particulier en utilisant des microprocesseurs classiques. Pour les opérateurs de traitement, il sera nécessaire d'implanter les quatre primitives particulières : *début de bloc, fin de bloc, execbloc, attendre.*

L'opérateur mise à jour a pour rôle de mettre à jour les domaines d'entrée des blocs en attente à l'aide des valeurs qui sont associées aux noms de communication dans les domaines de sortie. Il semble souhaitable de disposer d'un opérateur capable de réaliser des accés associatifs dans les ensembles A et S.

L'opérateur constructeur, pour sa part, n'a pas de fonction particulière à réaliser. Il doit assurer essentiellement la gestion d'une bibliothèque de blocs et la création de nouveaux blocs.

Ces opérateurs seront réalisés à l'aide d'unités microprogrammables ou de microprocesseurs standards.

Les ensembles A, X, S et D sont des ressources critiques dans le système car elles peuvent être utilisées simultanément par plusieurs

opérateurs. Une organisation mémoire particulière capable d'assurer le stockage de ces différents ensembles et de supporter des accés simultanés a été étudiée. Elle sera présentée en détail dans le troisième chapitre de ce document.

La définition de cette mémoire doit tenir compte des caractéristiques du modèle :

- Les opérateurs n'ont pas de liaisons directes entre eux, toutes les communications se font à l'aide d'ensembles homogènes d'informations. La mémoire doit donc être non seulement un outil de stockage mais également un outil de communication.

- Au cours de son existence, un objet passe par plusieurs étapes, il est traité par un opérateur qui le transforme en un objet d'un autre type. En partant d'une demande d'exécution, les transformations sont faites d'abord par l'opérateur constructeur, puis par l'opérateur mise à jour et enfin l'opérateur de traitement. Il est intéressant d'en tenir compte par une architecture présentant des caractéristiques de fonctionnement pipeline.

- La mémoire doit permettre des accés associatifs pour le processeur mise à jour puisque ce dernier la consulte en permanence pour compléter les domaines d'entrée.

- Le nombre et la taille des objets sont variables. Après utilisation par un opérateur ceux ci peuvent être supprimés, ce qui peut conduire à une fragmentation de l'espace mémoire et par conséquent nécessite la gestion d'un espace libre.

CHAPITRE III

UNE MÉMOIRE CIRCULANTE COMME OUTIL DE COMMUNICATION DANS UN SYSTÈME MULTIPROCESSEUR

1) *Influence des caractéristiques du modèle sur le choix d'une architecture de mémoire*

Une réalisation matérielle complète du système, exposé dans le deuxième chapitre exige, non seulement, l'étude de l'implémentation des opérateurs, mais également la définition d'une organisation de mémoire adaptée au modèle. Le choix d'une architecture doit tenir compte des points suivants.

1.1. Fonction de la mémoire

La mémoire doit remplir deux fonctions dans le système. Elle assure le *stockage des différents objets* qui sont utilisés dans Maud, c'est à dire les blocs exécutables, les blocs en attente, les demandes d'exécution et les domaines de sortie mais elle est aussi, conformément au modèle, le seul *moyen de communication*. Ceux-ci ne disposent pas, en effet, de liaisons directes entre eux, les contrôles d'accés et de synchronisation nécessaires étant réalisés par les communications de ces divers types d'objets.

1.2. Accès associatif

Le fait d'associer un *nom de communication* à chaque valeur transmise d'un domaine de sortie vers un domaine d'entrée impose que l'opérateur chargé de ce transfert (opérateur mise à jour) soit capable d'identifier un couple (nom, valeur) par son nom. Celui-ci doit donc pouvoir accéder à l'ensemble des objets de sortie de manière associative. L'utilisation d'une mémoire facilitant techniquement de tels accés simplifierait considérablement la fonction de cet opérateur.

1.3. Accès multiples simultanés

A partir du modèle, il apparait clairement que chaque ensemble d'objets doit être accessible, en lecture et en écriture, simultanément par plusieurs opérateurs.

1.4. Mémoire commune

Le modèle fonctionnel fait apparaître plusieurs ensembles distincts de stockage, chacun pouvant contenir un type d'objet. Les exigences sont sensiblement les mêmes quel que soit l'objet. Celui-ci ne doit être stocké que de manière temporaire avant utilisation par un autre

opérateur. D'autre part les modes d'accés peuvent être banalisés, chacun des opérateurs peut spécifier le type de l'objet auquel il veut accéder. Par conséquent l'utilisation d'une mémoire commune semble techniquement et économiquement plus favorable. De plus dans une telle optique, la gestion de l'espace mémoire serait plus aisée.

1.5. Gestion de la mémoire

Celle-ci doit être la plus simple possible. L'utilisation de fonctions d'accés complexes provoquerait des pertes de temps et serait une source d'interférences. Un système de contrôle centralisé pour les accés risque d'être le goulot d'étranglement du système.

D'autre part le nombre et la longueur des objets stockés est très variable. Comme ceux-ci ne doivent être utilisés qu'une seule fois, ils peuvent être supprimés dés leur utilisation par un opérateur, ce qui peut conduire à une fragmentation de la mémoire et rendre, par conséquent, délicate la gestion de la place disponible.

1.6. Temps d'accés

On peut montrer que statistiquement le temps d'accés à une information donnée croit avec le volume de cette information. Les ordres de grandeur sont classiquement les suivants.

- temps d'accés à une instruction en mémoire centrale 1 µs
- temps d'accés à 1 page sur disque 10 à 30 ms.

En première approximation, il est possible d'assimiler un bloc à une page de machine classique ce qui donne un ordre de grandeur du temps d'accés souhaitable.

1.7. Extensibilité

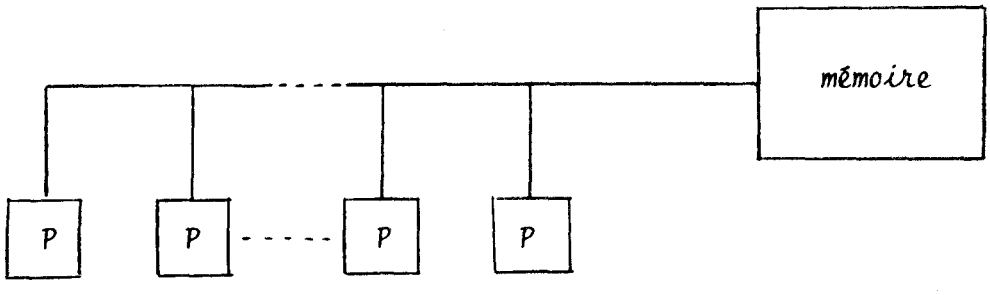
Le choix d'une architecture complétement figée n'est certainement pas souhaitable. Une augmentation éventuelle du nombre des opérateurs ne doit pas entrainer une réorganisation complète de la mémoire.

2) Communication entre processeurs par une mémoire commune

Dans une architecture à mémoire commune, les différents processeurs communiquent entre eux au travers d'un espace mémoire accessible à tous. La mémoire est alors un outil de communication et un outil de stockage. Les difficultés de réalisation, de gestion et d'extension dépendent essentiellement de la façon dont cette structure est implantée.

2.1. Accès à la mémoire par bus unique

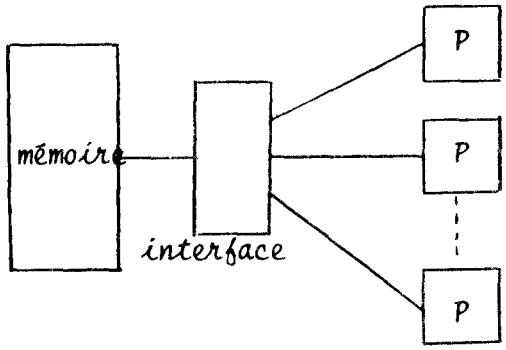
Les processeurs peuvent accéder à la mémoire à l'aide d'un bus commun



Un seul processeur posséde le contrôle du bus qui est partagé temporellement entre les différents processeurs selon une certaine politique d'allocation. Ce mode d'accés ne peut pas convenir puisque, à un instant donné, une *seule communication* peut avoir lieu, entre les processeurs et la mémoire, d'où une impossibilité d'avoir des accés simultanés.

2.2. Accès à la mémoire par un interface

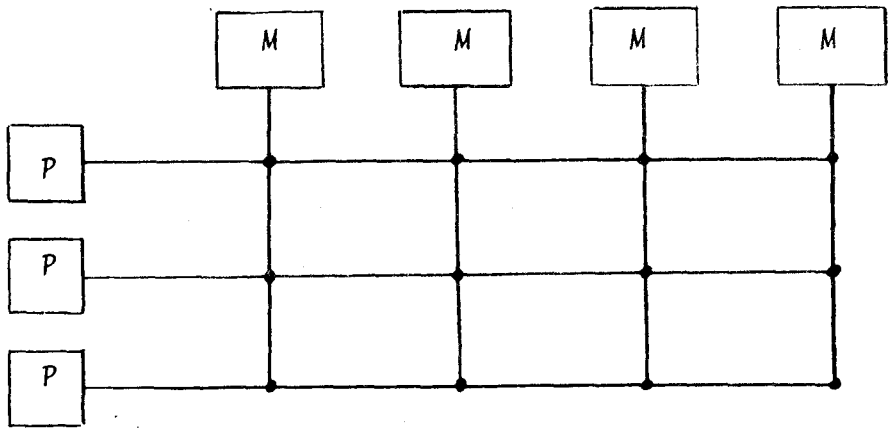
A la mémoire est attaché un interface auquel sont connectés tous les processeurs. Chaque processeur a son propre bus de communication. Le contrôle d'accés à la mémoire est réalisé par l'interface qui règle les conflits d'accés. Cette manière de connecter les processeurs à la mémoire présente deux inconvénients. Le premier réside dans le nombre, forcément limité, d'accés prévus dans l'interface, ce qui peut rendre l'adjonction d'un nouveau processeur impossible sans modification de l'interface et du mécanisme de contrôle. Le deuxième est qu'il existe un *goulot d'étranglement* entre l'interface et la mémoire proprement dite, dans le cas où le nombre de processeurs est relativement grand.



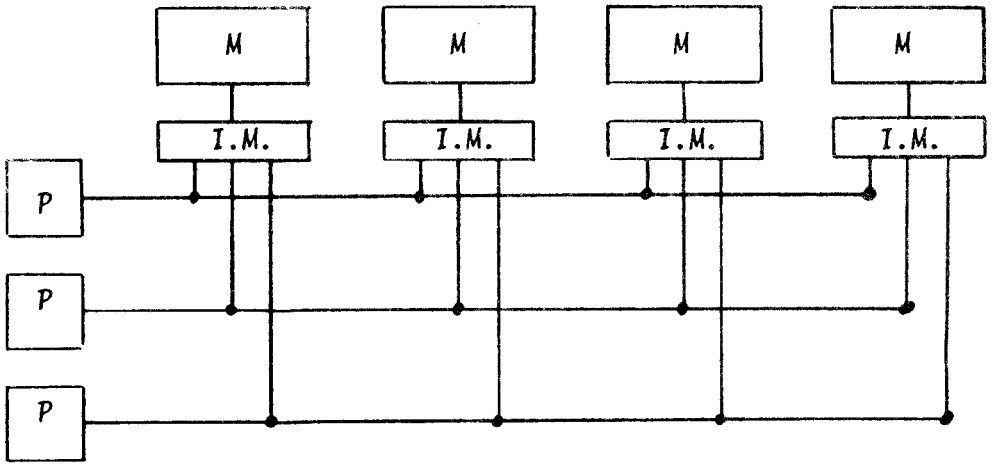
2.3. Mémoire découpée en plusieurs blocs (Réseau Cross-Bar)

Pour augmenter le nombre d'accés simultanés, la mémoire est souvent divisée en plusieurs blocs séparés et accessibles séparément.

La structure de connexion la plus communément utilisée est du type cross-bar. L'accés à chaque bloc de mémoire se fait au moyen d'un bus commun à tous les processeurs. L'inconvénient majeur du cross-bar est que sa complexité croit comme le produit du nombre de blocs par le nombre de processeurs. D'autre part, le contrôle d'un tel dispositif et complexe.



Pour régler les conflits d'accés, on peut adjoindre à chaque bloc de mémoire un interface. On retrouve ici encore les difficultés citées plus haut.



2.4. Conclusion

Toutes les techniques que nous avons énoncées présentent des inconvénients en regard des caractéristiques souhaitées.

Une mémoire adressable convient fonctionnellement au modéle mais nécessite un *système de gestion complexe*. Bien qu'il soit possible de résoudre les problèmes des *conflits d'accès*, ceux-ci entraineront inévitablement une perte temps au niveau des échanges.

Les *possibilités d'extension* sont *difficiles*, voir impossibles. Elles nécessitent obligatoirement une reconfiguration de l'ensemble de l'architecture et du dispositif de gestion.

Une mémoire adressable de par son mode d'accés rend difficile les accés associatifs. Il est nécessaire d'adjoindre un dispositif de contrôle complexe. De plus les temps d'accés deviennent relativement longs. Dans notre système la notion d'adressage n'intervient qu'à un niveau local. Au niveau global, le transfert des valeurs d'un bloc à un autre s'effectue à l'aide d'un nom de communication et il est possible de généraliser l'accés par le nom pour la sélection des blocs par un opérateur.

Toutes ces raisons nous ont amenés à proposer une organisation de mémoire réalisée à l'aide des nouvelles technologies : Les *mémoires circulantes*.

3) *Les mémoires circulantes*

L'apparition de nouvelles technologies de mémoires (CCD, mémoires à bulles magnétiques, registre à décalage de grande capacité), que nous regroupons sous le terme de *mémoire circulantes*, permet d'envisager des améliorations ou même des innovations dans l'architecture des ordinateurs. Du point de vue prix et performances, elles se situent entre les mémoires magnétiques (disques et bandes) et les mémoires électroniques à accés aléatoire. Elles ont, comme les premières, un accés séquentiel mais leur technologie et leurs performances les rapprochent des secondes.

3.1. *Technologie des mémoires circulantes*

D'une manière générale, une mémoire circulante peut se diviser en trois parties [PAN 77] [CORD 78].

- Un *milieu de propagation* le long duquel circule l'information mémorisée. Le contrôle de la circulation s'effectue à l'aide d'une horloge.

- Un *point d'entrée* où l'information à inscrire est présentée en séquence.

- Un *point de sortie* où l'information apparait après avoir effectué un parcours complet à travers le milieu de propagation.

Dans de nombreux cas, un circuit de communication externe ou commandable de l'extérieur permet de réinjecter au point d'entrée, l'information qui se présente au point de sortie pour assurer une circulation permanente en synchronisation avec une horloge de contrôle. Un point d'entrée et un point de sortie confondus, constituent une voie d'accés à l'information que l'on appelera dans la suite : *fenêtre d'accés*. C'est le seul point où il est possible d'accéder à l'information pour un usage externe.

Plusieurs technologies de mémoires circulantes ont vu ainsi le jour. On peut citer notamment :

- Les *mémoires CCD* (charge coupled device). Ce sont des circuits électroniques qui transfèrent l'information sous la forme d'une charge

électrique le long d'une rangée de condensateurs MOS. La présence ou l'absence de charge caractérise l'information binaire élémentaire.

- Les *mémoires à bulles magnétiques*. Dans un monocrital de matériau magnétique, il est possible de créer, de déplacer ou d'effacer des discontinuités très localisées que l'on appelle des bulles. Un réseau régulier les guide et fixe le pas de progression.

- Les *registres à décalage*. On distingue les registres dynamiques et statiques. Dans ces derniers,la circulation peut être arrêtée sans pour cela provoquer la perte de l'information.

- Les *mémoires à propagation de domaines*. Il s'agit en fait d'une variante des mémoires à bulles qui n'exigent pas un champ magnétique externe d'entretien.

La table qui est présentée à la figure 1 donne les principales caractéristiques des mémoires classiques et des mémoires circulantes.

3.2. *Utilisation des modules de mémoires circulantes*

Pour la réalisation d'un organe de mémoire complet les cellules de base que l'on peut trouver sur le marché, présentent à la fois une capacité insuffisante et un mode d'exploitation trop rigide. On est donc conduit à étudier le groupement de plusieurs cellules. Trois techniques peuvent être envisagées : le groupement série, le groupement parallèle, le groupement mixte.

- *Groupement série*. Il consiste à disposer les cellules les unes à la suite des autres en les contrôlant par la même horloge. Le point de sortie de la dernière cellule est relié au point d'entrée de la première de manière à former un anneau

	Taille du Chip bits	Coût centimes/bit	Temps d'accès µs	Débit Mb/sec	Densité Kb/cm^2	Consommation µW/bit	Consommation en attente (à l'arrêt) µW/bit	volatilité	Tensions d'alimentation V
RAM bipolaires	4 K (1)	10	0,05	20	20	200	20 (1)	OUI	+5
RAM MOS Statiques	4 K	1,5	0,3	3	50	100	10	OUI	+5
RAM MOS Dynamique	16 K	1	0,3	3	80	50	2	OUI	±5,+12
RAM C MOS	2 K	10	0,5	2	20	30	0,05	OUI	+5
ROM	64 K	5	0,5	2	50	30	30 (0)	NON	+5
PROM	16 K	3	0,1	10	20	100	100 (0)	NON	+5
REPROM UV	16 K	7	0,5	2	50	30	10 (0)	NON	+5
EE PROM	4 K	2	2	0,5 (2)	10	100	35 (0)	NON	+5,-12,-30
CCD	64 K	0,1	10^2	5	100	15	1,5	OUI	+12,-5
MBM	92 K	0,1	4.10^3	0,05	300	7 (3)	0 (3)	NON	+5,+12
Domaines	131 K (4)	0,2	450	5	15	50	0,4 (0)	NON	±5,±12

Fig 1 : Tableau comparatif des mémoires intégrées

(1) Dynamique

(2) Effacement 10 ms, écriture 1 ms

(3) Interfaces non compris

(4) Pour 1 substrat de 5 cm × 5 cm

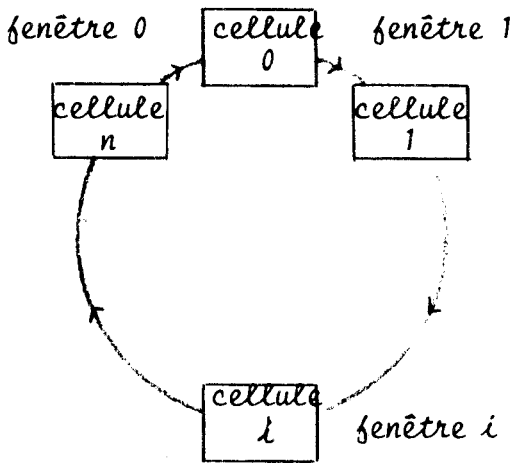


Fig 2 : Groupement série

Dans ce mode, il apparait autant de fenêtres que de cellules. Celles-ci peuvent être exploitées pour réaliser des accés multiples. Le problème des conflits d'accés est ici parfaitement résolu puisque les points d'accés pour chaque utilisateur sont physiquement indépendants.

L'augmentation de la capacité mémoire entraine en contrepartie un temps de circulation accru dans la même proportion. L'existence d'un grand nombre de fenêtres d'accés permet l'insertion d'un réseau de commutation entre les cellules pour réduire le temps d'accés. Nous reviendrons sur ce point dans la suite de ce chapitre.

- *Groupement parallèle*

Si la cellule assure le stockage au niveau du bit, il est possible de juxtaposer plusieurs cellules pour y déposer un mot entier qui sera considéré du point de vue accés comme un objet unique.

Le groupement parallèle consiste à reboucler plusieurs cellules sur elles-mêmes. Deux modes de stockage peuvent être envisagés :

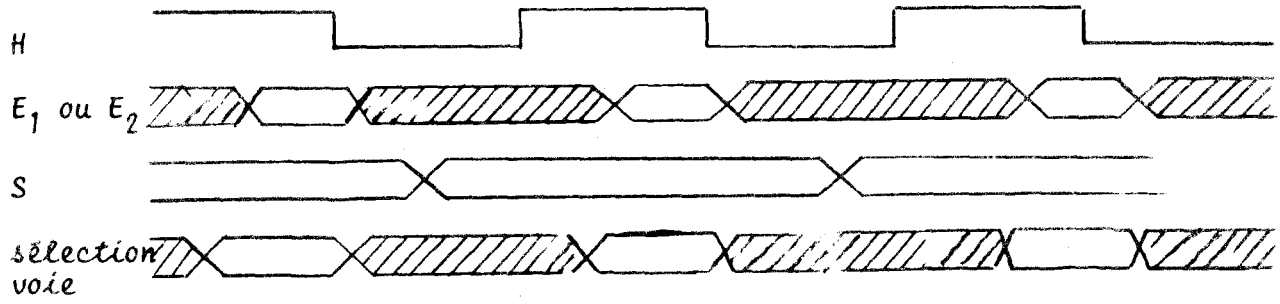


Fig 8 : Timing des signaux

Les performances du circuit sont les suivantes :

- fréquence d'horloge maximale : 1,5 MHz

- état décalage (H = low) durée minimale 250 ns

- état repos (H = hight) durée minimale 350 ns, durée maximale 100 µs.

L'entrée des informations s'effectue sur le front descendant de l'horloge.

4.2. La cellule de base [CORD 79]

La cellule de base a une capacité de 1 K octets. Elle nécessite par conséquent l'exploitation en parallèle de huit modules de 1K/1 bit, ceux-ci étant soumis aux mêmes signaux de contrôle.

Autour de chaque module figurent.

- Un *multiplexeur de voies d'entrée* qui permet de piloter l'entrée du registre à décalage par une voie choisie parmi quatre possibles. Le contrôle de ce multiplexeur exige deux signaux binaires.

- Un *amplificateur de sortie* car le module ne peut piloter en sortie qu'un seul poids TTL.

- Un *multiplexeur de sortie* à deux voies. L'une des voies est la sortie du module, l'autre est la sortie du multiplexeur d'entrée cité plus haut. Ce dispositif exige un signal binaire de contrôle. Son rôle est de permettre le shunt du module par mise en relation directe de l'entrée et de la sortie.

- Des groupements mixtes sont évidemment possibles et permettent de définir des configurations originales réalisant un compromis entre la capacité et le temps d'accés.

3.3. Application des mémoires circulantes [PAN 77] [VAN 79]

De par leurs caractéristiques, les mémoires circulantes peuvent intervenir comme niveau intermédiaire dans une hiérarchie de mémoire. L'aspect séquentiel n'est nullement défavorable puisque les changements de niveau se font par page, et donc séquentiellement.

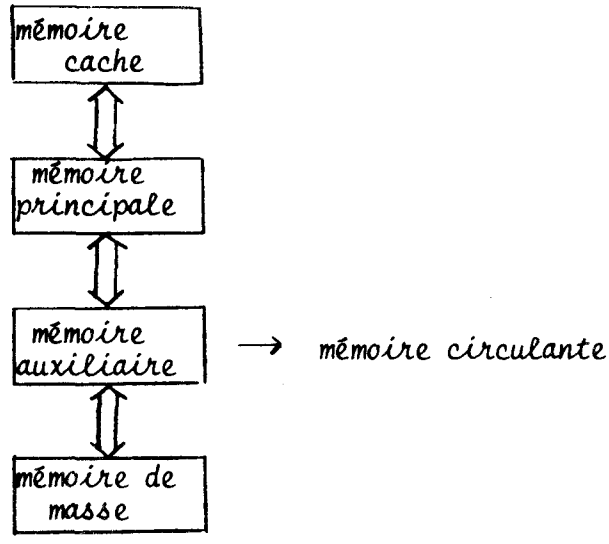


Fig 5 : Hiérarchie de mémoire avec des mémoires circulantes

Comme nous l'avons vu dans le paragraphe précédent, l'accés associatif à une mémoire classique nécessite une circuiterie externe importante. Dans le cas de l'examen successif de chaque mot, l'adressage devient parfaitement inutile et l'ordre d'examen des mots est généralement quelconque. Dans ce cas l'implémentation d'une mémoire associative à l'aide d'une mémoire circulante est simple et avantageuse même si elle n'apporte que des performances limitées par rapport à un modèle parallèle par mot.

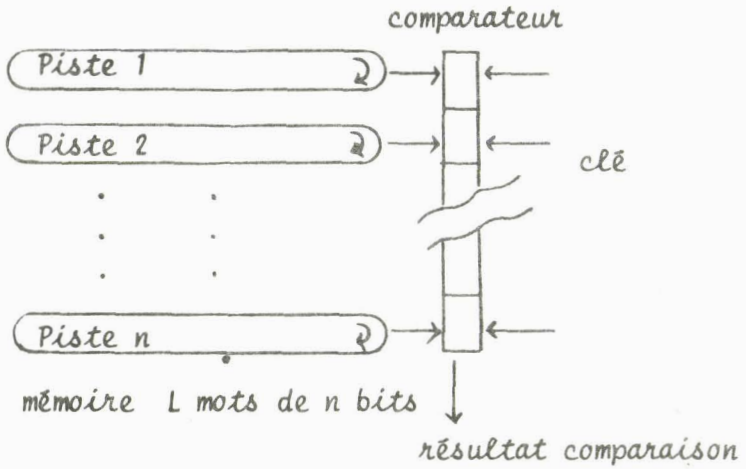


Fig 6 : Mémoire circulante à accès associatif en organisation parallèle

3.4. Conclusions

Comparativement aux mémoires classiques, les mémoires circulantes semblent mieux adaptées à la réalisation de la mémoire de Maud.

En temps qu'outil de stockage, les divers groupements possibles de cellules permettent de disposer d'une mémoire de capacité importante, accessible simultanément par plusieurs utilisateurs.

La caractéristique principale de ces circuits, c'est à dire, la circulation de l'information amène tout naturellement à les utiliser en tant qu'outil de circulation. Ceci permet également un balayage de l'ensemble des informations stockées et ainsi une recherche associative. Celle-ci qui n'était nécessaire que pour le transfert des valeurs, peut se généraliser aisément au niveau de la recherche d'un bloc par un opérateur.

Dans la suite de ce chapitre, nous nous attacherons à décrire l'outil qui a été réalisé ainsi qu'un certain nombre d'applications possibles. La simplicité du contrôle de circulation, la possibilité de réaliser des accés simultanés sans conflit, la suppression de la notion d'adressage en font un outil dont la mise en ouvre et la gestion restent simples. Des possibilités de reconfiguration locale permettent en outre, des réductions notables du temps d'accés.

4) Réalisation matérielle de la mémoire de Maud

La mémoire de Maud se compose de 32 *cellules identiques*, chacune ayant une capacité de 1 K octets, l'intérêt d'une réalisation modulaire résidant dans la facilité d'extension du système.

4.1. Le composant de base

Le composant constituant la mémoire (Figure 7) est le module 2533 de Signetics. Il s'agit d'un *registre à décalage* en technologie M.O.S. de 1K/1 bit

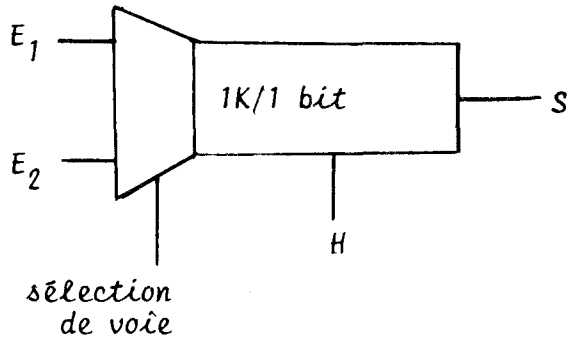


Fig 7 : Module de mémoire circulante (2533 Signetics)

Les voies d'accés à ce module sont au nombre de trois :

- deux *voies d'entrée* (E_1 et E_2). Les informations sont introduites en séquence par la voie qui est sélectionnée (E_1 ou E_2).

- Une *voie de sortie* (S) où l'on récupère l'information après un parcours complet le long du milieu de propagation.

Deux signaux permettent d'assurer le contrôle.

- L'*horloge* (H) assure la circulation des informations et fixe aussi la vitesse de propagation. La circulation de l'information à travers tout le circuit s'effectue donc en 1024 T si T est la période d'horloge choisie.

- Le signal de *sélection de voie* (stream select) qui permet de sélectionner la voie E_1 ou E_2.

Le composant est réalisé en technologie MOS statique c'est à dire qu'il est possible d'arrêter la circulation sans perdre l'information mémorisée.

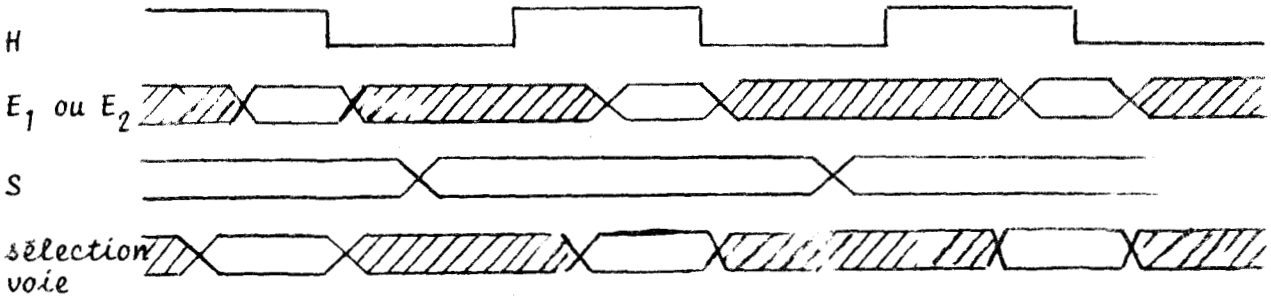


Fig 8 : Timing des signaux

Les performances du circuit sont les suivantes :

- fréquence d'horloge maximale : 1,5 MHz

- état décalage (H = low) durée minimale 250 ns

- état repos (H = hight) durée minimale 350 ns, durée maximale 100 µs.

L'entrée des informations s'effectue sur le front descendant de l'horloge.

4.2. La cellule de base [CORD 79]

La cellule de base a une capacité de 1 K octets. Elle nécessite par conséquent l'exploitation en parallèle de huit modules de 1K/1 bit, ceux-ci étant soumis aux mêmes signaux de contrôle.

Autour de chaque module figurent.

- Un *multiplexeur de voies d'entrée* qui permet de piloter l'entrée du registre à décalage par une voie choisie parmi quatre possibles. Le contrôle de ce multiplexeur exige deux signaux binaires.

- Un *amplificateur de sortie* car le module ne peut piloter en sortie qu'un seul poids TTL.

- Un *multiplexeur de sortie* à deux voies. L'une des voies est la sortie du module, l'autre est la sortie du multiplexeur d'entrée cité plus haut. Ce dispositif exige un signal binaire de contrôle. Son rôle est de permettre le shunt du module par mise en relation directe de l'entrée et de la sortie.

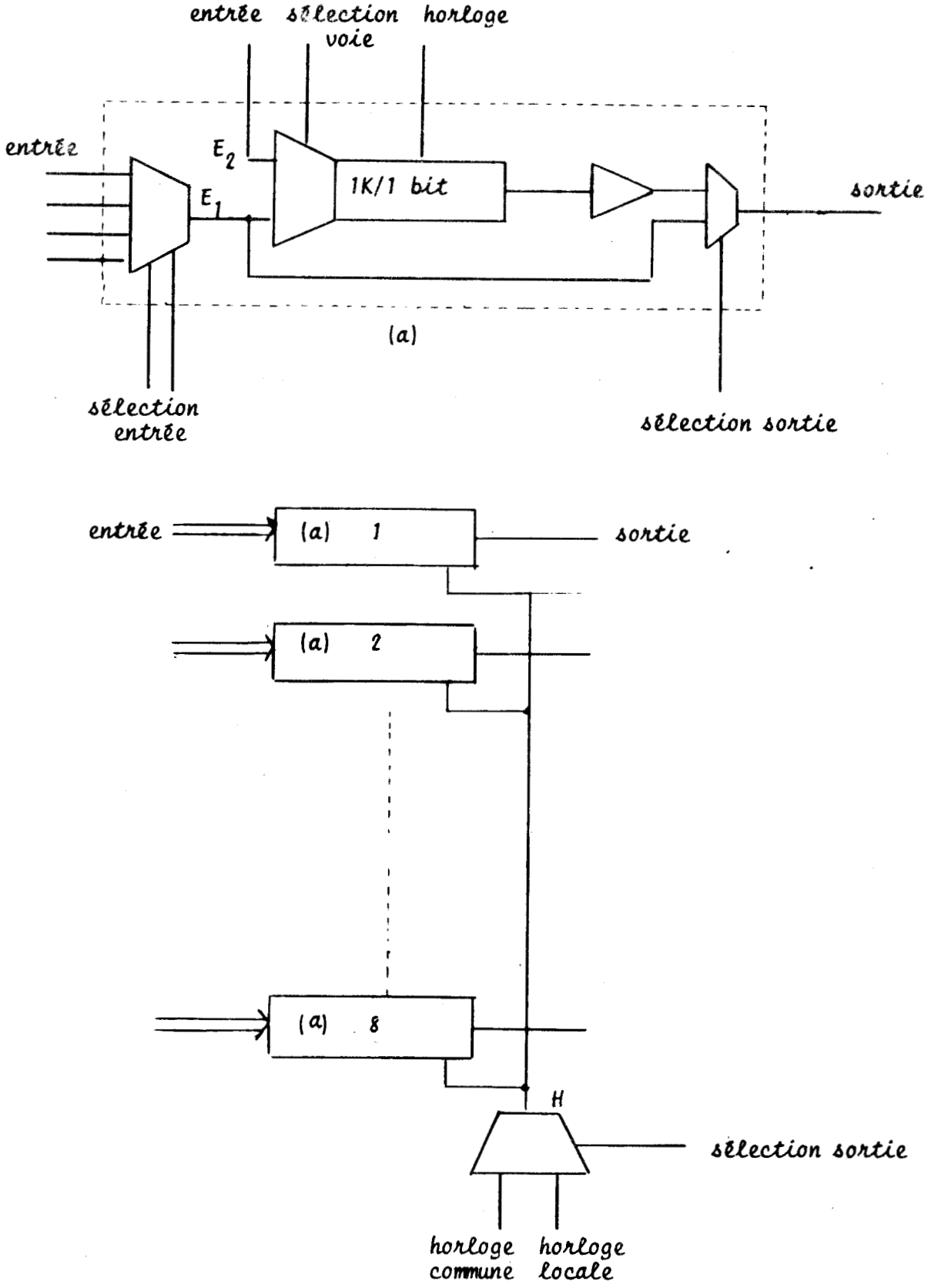


Fig 9 : Cellule de base de la mémoire circulante

Les signaux de contrôle sont communs à l'ensemble des circuits. Lorsqu'une cellule est court-circuitée grâce au multiplexeur de sortie, le signal de contrôle de ce dernier permet la sélection de l'horloge grâce à un *multiplexeur d'horloge* à deux voies. Il est possible ainsi de remplacer l'horloge commune par une horloge locale qui assure la circulation de l'information stockée dans la cellule isolée.

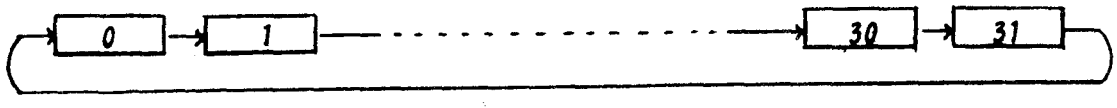
Un schéma complet d'une cellule de base est donné figure 9. L'ensemble donne ainsi une capacité de 32 K octets. Des raisons de fiabilité ont conduit à proposer le pilotage par une horloge de 1 MHz.

5) *Interconnexion des cellules*

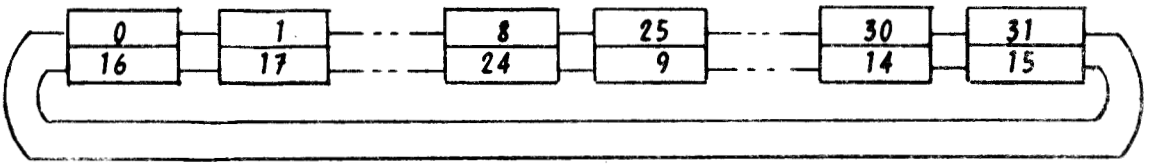
La conservation de l'information stockée s'effectue en rebouclant une ou plusieurs cellules les unes sur les autres, ainsi l'information circule sans modification le long des registres.

Le multiplexeur à quatre voies placé devant chaque module de mémoire permet de choisir pour chaque cellule un prédecesseur parmi quatre. Les liaisons intercellulaires ont été choisies de manière à pouvoir modifier la taille des mots dans la mémoire. Six configurations ont été ainsi choisies.

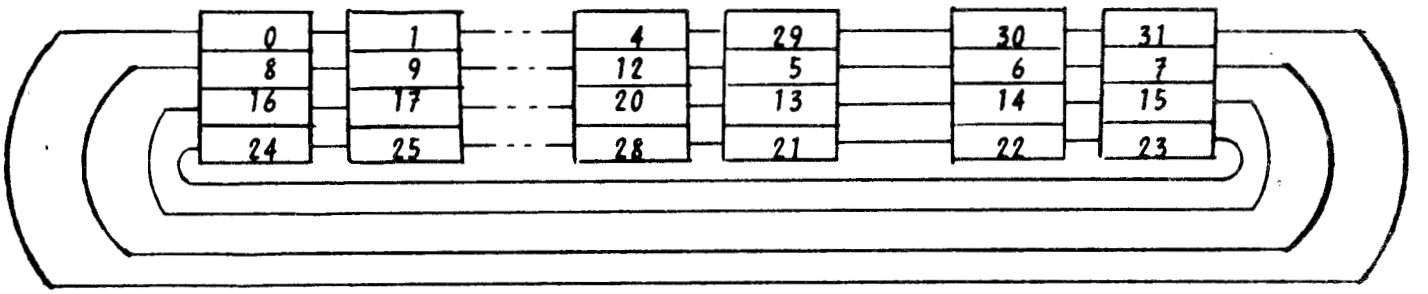
- La *configuration série*. Les 32 cellules sont connectées les unes derrière les autres, la sortie de la dernière étant rebouclée à l'entrée de la première de façon à constituer un anneau. La capacité est donc de 32 K octets.



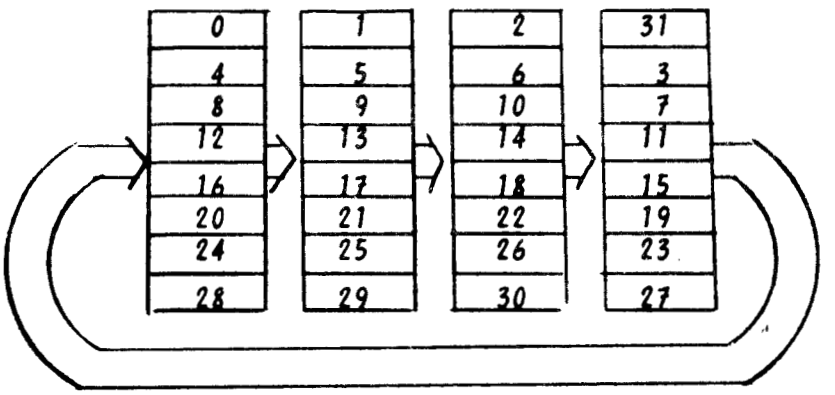
- La *configuration mot-court*. On réalise 2 anneaux indépendants de 16 cellules chacun. Si l'exploitation des informations se fait en parallèle on dispose alors de mots de 16 bits. La capacité devient donc de 16 K mots de 16 bits.



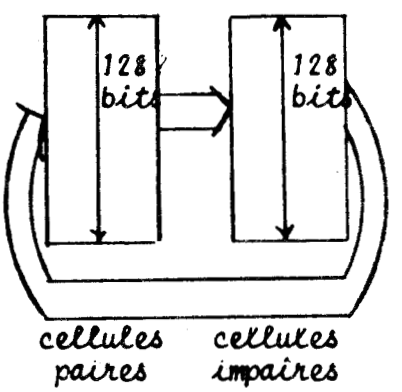
- La *configuration mot long*. L'anneau est divisé en 4 anneaux indépendants de 8 cellules chacun. Si l'information contenue dans les cellules est exploitée simultanément sur les 4 anneaux, la capacité est de 8 K mots de 32 bits



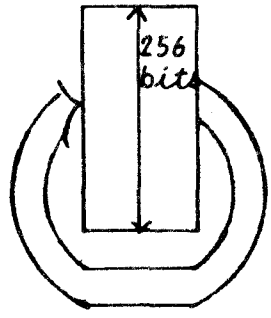
- La *configuration bloc long*. 8 cellules sont exploitées en parallèle. On dispose donc de 8 anneaux indépendants, de 4 cellules chacun, ce qui donne une capacité de 4 K mots de 64 bits.



- La *configuration bloc court*. 16 cellules sont exploitées en parallèle. On dispose donc de 16 anneaux indépendants de 2 cellules chacun, ce qui donne une capacité de 2 K mots de 128 bits.



- La *configuration parallèle*. Dans cette configuration, toutes les cellules de base sont rebouclées sur elles mêmes et sont toutes exploitées en parallèle. On dispose donc de 32 anneaux indépendants de 1 cellule chacun, ce qui donne une capacité de 1K mots de 256 bits.



Pour garantir la conservation de l'information, il est évident que, pour l'ensemble des cellules constituant un anneau, le contrôle de la circulation est assurée par une horloge commune.

Dans le but de simplifier le réseau et de réduire le nombre d'interconnexions entre les cellules, chaque cellule n'a au plus que trois prédecesseurs. Ceci justifie l'emploi du multiplexeur d'entrée. Le choix de l'une des six configurations se fait à l'aide de signaux de contrôle de ces multiplexeurs. La figure 10 présente les différentes liaisons intercellulaires et il est très facile d'y retrouver les différentes combinaisons.

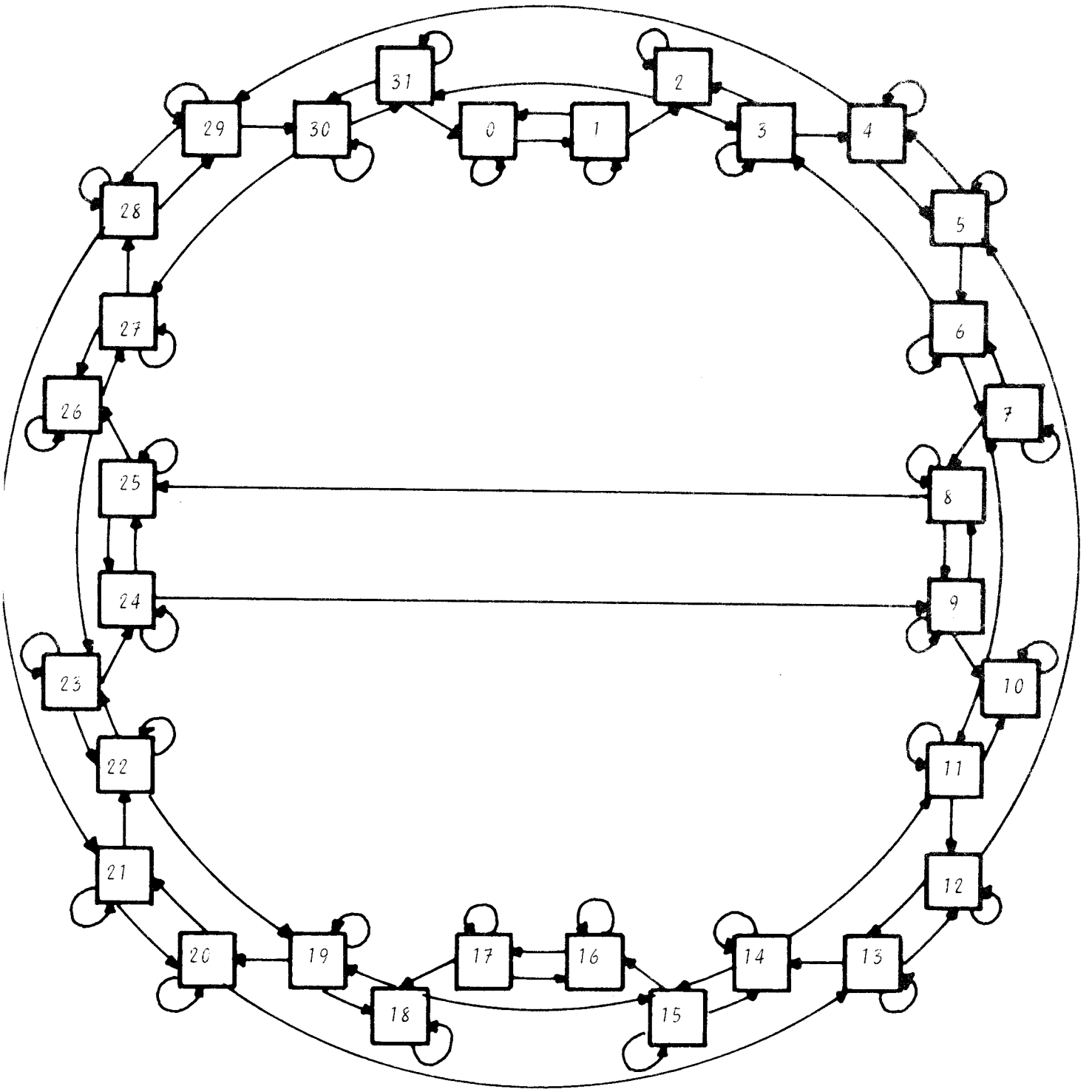


Fig 10 : Réseau d'interconnexion de la mémoire circulante

6) Les accès à la mémoire circulante

La lecture ou l'écriture dans cette mémoire ne peut se faire que de manière séquentielle au niveau des fenêtres d'accés. Afin de garder la possibilité d'effectuer des accés multiples, ces opérations ne doivent, en aucun cas bloquer la circulation dans l'anneau. Deux modes d'accés différents peuvent être envisagés :

6.1. L'accés par capture

Nous avons vu dans le paragraphe 4 qu'il était possible d'isoler une cellule à l'aide du multiplexeur de sortie, en court circuitant l'entrée et la sortie. Dés lors, un dispositif externe peut exploiter de manière séquentielle l'information contenue dans une cellule (figure 11).

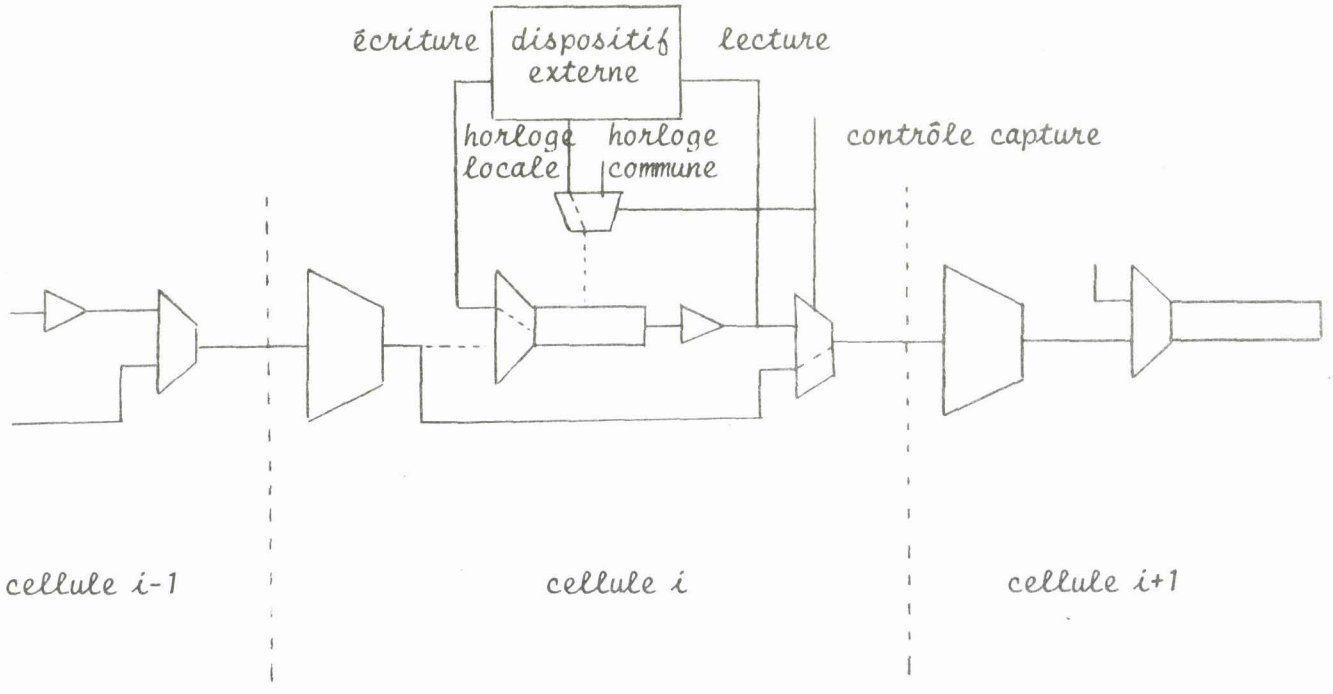


Fig 11 : Accés par capture

Après isolation de la cellule, la lecture s'effectue en sortie du module et l'écriture, sur la deuxième entrée du sélecteur de voie. Le contrôle de l'avance est assuré par une horloge locale qui est fournie par le dispositif externe.

Lorsque l'opération d'accés à la cellule est achevée, il est possible de la libérer c'est à dire de l'insérer à nouveau dans l'anneau.

6.2. L'accés à la volée

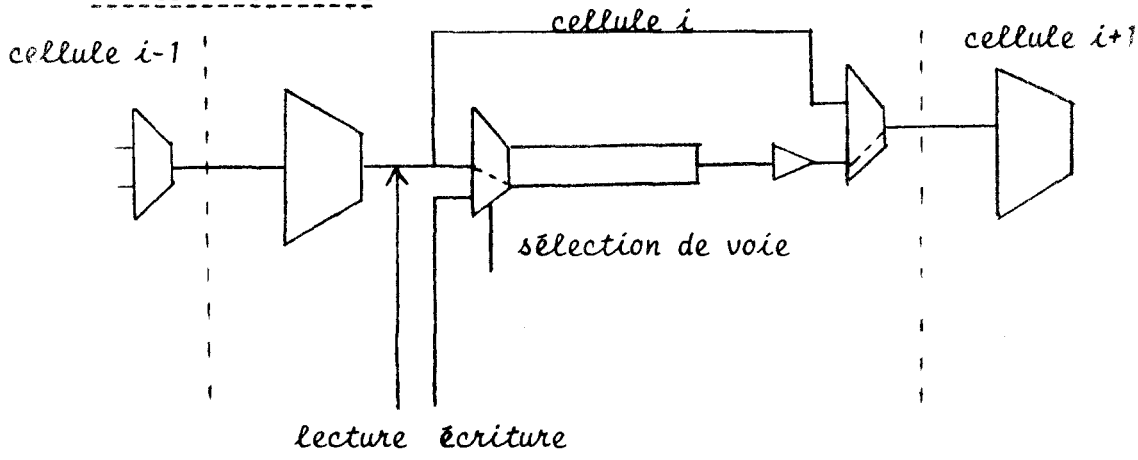


Fig 12 : Accés à la volée

Il est possible de lire par simple consultation, le mot présent à l'entrée du sélecteur de voie d'une cellule. Cette information est également transmise à l'entrée. Par conséquent la lecture n'est pas destructive.

L'opération d'écriture s'effectue comme dans le cas de l'accés par capture. L'information à mémoriser est présentée sur la deuxième entrée du sélecteur de voie tout en positionnant de façon adéquate le signal de contrôle de ce sélecteur. Lors d'une telle opération, l'information qui était fournie par la cellule précédente est perdue.

Pour ce mode de fonctionnement, les opérations d'accés doivent être synchronisées avec l'horloge commune. D'autre part trois types d'accés sont possibles.

- L'*accés lecture*. Un ou plusieurs mots sont lus au niveau d'une fenêtre par simple consultation de l'information qui est transférée d'une cellule à une autre. Cette lecture est non destructive.

- L'*accés écriture*. Il consiste à écrire un ou plusieurs mots dans la mémoire par la deuxième entrée du sélecteur de voie.

- L'*accès mise à jour*. Au niveau d'une fenêtre d'accés, l'information qui est transmise à la cellule suivante, est présente bien avant sa mémorisation effective. Il est donc possible de lire le mot présent en sortie, de le modifier suivant un certain traitement, et de le présenter en vue d'une écriture. Le temps d'exécution de cette mise à jour doit être inférieur à celui d'une période d'horloge compte tenu des tolérances sur le circuit.

Il faut noter que les deux familles ne sont pas exactement compatible du point de vue des voies d'accés. En accés à la volée, l'entrée et la sortie sont consécutives, tandis que en accés par capture, elles sont séparées par le module capturé. Toutefois, il est possible de les utiliser simultanément sur l'anneau sur des fenêtres distinctes.

L'accés à la volée, apparait, cependant, plus souple sur le plan de la gestion de l'anneau. En effet aucune variation n'intervient dans la longueur de l'anneau. Dans l'autre cas, il est nécessaire d'introduire un dispositif de contrôle de longueur.

7) Reconfiguration de la mémoire

La reconfiguration de la mémoire circulante est rendue possible grâce à la présence des multiplexeurs d'entrée et de sortie au niveau de chaque cellule. Bien qu'elle ne soit pas indispensable dans Maud, elle permet une diversité des applications de cette mémoire, nous en citerons un certain nombre en fin de ce chapitre.

Les différentes possibilités de reconfiguration autorisent en particulier :

- Une *variation de la longueur des mots mémorisés*. Le choix du réseau intercellulaire permet de réaliser six configurations différentes comme nous l'avons vu précédemment (§ 5).

- Une *variation de la longueur de l'anneau*. La longueur de l'anneau peut être modifiée de manière dynamique.

- Une *variation du temps d'accés*. L'inconvénient majeur des mémoires séquentielles est lié au fait que le temps d'accés à l'information qui est

mémorisée est proportionnel à la taille de la mémoire. L'utilisation des possibilités de reconfiguration permet de le réduire de façon notable et dans certains cas, de le rendre indépendant de la taille de la mémoire.

7.1. Reconfiguration par le multiplexeur d'entrée

Nous avons vu, lors de la présentation technique que l'entrée de chaque cellule est constituée par un multiplexeur d'entrée à quatre voies. L'information qui est donc mémorisée provient de l'une de ces quatre voies. En fait, pour la réalisation du réseau, seules trois voies sont nécessaires. En effet, nous avons cherché à minimiser le nombre de connexions matérielles, dans le souci de faciliter la réalisation technique. Une cellule dispose donc de trois prédécesseurs au maximum. La possibilité de sélectionner un prédecesseur pour chaque cellule peut être exploitée de trois manières différentes. Pour des raisons de conservation de l'information stockée, il est évident qu'une cellule ne peut être connectée avec un prédecesseur ou un sucesseur unique.

- *variation de la taille des mots.*

Cet aspect de la reconfiguration a déjà été cité précédemment. Le réseau de communication a été conçu de manière à permettre l'utilisation de la mémoire dans des applications diverses où la taille des mots qui doivent être exploités varie suivant les puissances de deux. Six configurations sont ainsi disponibles (8, 16, 32, 64, 128 et 256 bits). Les cellules sont groupées en série de manière à constituer des boucles fermées sur elles mêmes, celles-ci sont exploitées en parallèle et le contrôle de la circulation est assuré par une horloge commune.

- *Permutation des articles contenus dans une cellule*

Considérons la table des prédecesseurs de chaque cellule.

cellule	cellules précédentes	cellule	cellules précédentes
0	0, 1, 31	16	15, 16, 17
1	0, 1	17	16, 17
2	1, 2, 3	18	17, 18, 19
3	2, 3, 6	19	18, 19, 22
4	3, 4, 5	20	19, 20, 21
5	4, 5, 12	21	20, 21, 28
6	5, 6, 7	22	21, 22, 23
7	6, 7, 10	23	22, 23, 26
8	7, 8, 9	24	23, 24, 25
9	8, 9, 24	25	8, 24, 25
10	9, 10, 11	26	25, 26, 27
11	10, 11, 14	27	26, 27, 30
12	11, 12, 13	28	27, 28, 29
13	12, 13, 20	29	4, 28, 29
14	13, 14, 15	30	29, 30, 31
15	14, 15, 18	31	2, 30, 31

Fig 13 : Table des précécesseurs

L'exploitation de cette table montre qu'il est possible, pour chaque configuration, de décomposer chaque boucle en sous boucles tout en respectant la règle d'utilisation unique pour chaque cellule afin d'assurer la conservation de l'information mémorisée.

exemple *configuration 8 bits*.

L'anneau qui est constitué de 32 cellules groupées en série peut être subdivisé de la manière suivante.

- 2 boucles de 2 cellules 0 → 1 → 0 et 16 → 17 → 16
- 1 boucle de 4 cellules 8 → 25 → 24 → 9 → 8
- 1 boucle de 8 cellules 4 → 29 → 28 → 21 → 20 → 13 → 12 → 5
- 1 boucle de 16 cellules 2 → 31 → 30 → 27 → 26 → 23 → 22 → 19 → 18 → 15 → 14 → 11 → 10 → 7 → 6 → 3 → 2.

Le tableau ci-dessous indique le nombre de boucles de chaque taille qu'il est possible de réaliser dans chacune des configurations.

configuration	nombre de cellules par boucle									
	1	2	4	6	8	10	12	14	16	32
8 bits	32	16	9	2	6	6	22	28	23	1
16 bits	16	7	3	X	1	X	X	X	1	X
32 bits	8	3	1	X	1	X	X	X	X	X
64 bits	4	1	1	X	X	X	X	X	X	X
128 bits	2	1	X	X	X	X	X	X	X	X
256 bits	1	X	X	X	X	X	X	X	X	X

Fig 14 : Nombre de sous boucle réalisable.

exemple : *Configuration 8 bits*

* boucles de 6 cellules 7 → 8 → 25 → 24 → 9 → 10 → 7

8 → 25 → 26 → 23 → 24 → 9 → 8

Ces deux boucles ne peuvent pas être réalisées simultanément.

* boucle de 14 cellules

2 → 31 → 30 → 27 → 26 → 23 → 22 → 19 → 20 → 13 → 12 → 5 → 6 → 3 → 2.

Dans chaque configuration, le découpage de la mémoire en sous anneaux, en respectant la régle d'utilisation unique de chaque cellule (un seul prédécesseur est sélectionné pour chaque cellule) permet de réaliser un ré-arrangement des cellules. Ceci revient, en d'autres termes, à effectuer une permutation des articles contenus dans les cellules. A chaque organisation des différentes cellules de l'anneau correspond une permutation élémentaire.

Les différentes configurations ne présentent pas, bien évidemment les mêmes possibilités. En particulier dans la configuration bloc court (128 bits) et parallèle (256 bits), la reconfiguration ne présente aucun intérêt.

A l'aide des propriétés de l'ensemble des permutations, on démontre que toute permutation peut être exprimée uniquement à l'aide de cycles opérant sur des sous ensembles des n objets et en particulier comme un produit de transpositions.

En raison de la symétrie parfaite des liaisons intercellulaires, c'est la configuration 8 bits qui apparait la plus riche. On peut montrer qu'on peut réaliser des permutations qui constituent une base de l'espace des permutations. Pour cela, il suffit de montrer, par exemple que les transpositions qui consistent à échanger le contenu d'une cellule paire avec le contenu de la cellule impaire suivante et vice versa, peuvent être effectuées.

A l'aide de la figure 10, il est facile de voir que le réseau d'interconnexion permet d'effectuer la première des transpositions. Soit f_1 cette transposition, on peut écrire

$$f_1(i) = i + 1 \mod 32 \text{ pour } i \text{ pair}$$

et $$f_1(i) = i - 1 \mod 32 \text{ pour } i \text{ impair.}$$

La deuxième transposition f_2 n'est pas réalisable directement mais la combinaison de deux permutations élémentaires permet d'aboutir à un résultat similaire. On doit avoir :

$$f_2(i) = i + 1 \mod 32 \text{ pour } i \text{ impair}$$

et $$f_2(i) = i - 1 \mod 32 \text{ pour } i \text{ pair.}$$

Soit f_3 la permutation circulaire avec

$$f_3(i) = i + 1 \mod 32 \text{ pour } i \text{ quelconque}$$

On peut montrer que

$$f_2(i) = \underbrace{f_3 \circ f_3 \circ \ldots \circ f_3}_{31 \text{ fois}} \circ f_1 \circ f_3(i)$$

Dans la configuration 8 bits, il est, par conséquent possible d'effectuer une permutation quelconque des 32 articles. L'intérêt d'un tel dispositif est évident dans une opération de tri qui n'est rien d'autre que la réalisation d'une permutation de plusieurs éléments.

- *Réduction du temps d'accès à un article*

Dans le cas où on ne dispose que de la permutation circulaire, le temps d'accés à un article situé dans une cellule quelconque est proportionnel au nombre de cellules. Le tableau figure 15 donne le temps d'accés moyen dans chacune des configurations. To représente le temps de transfert du contenu d'une cellule vers une autre, c'est à dire 1024 T où T est égal à la période de l'horloge qui controle la circulation (si la fréquence d'horloge est de 1 MHz To = 1,024 ms).

Le temps moyen d'accés est donné par la formule N x To/2 où N est égal au nombre de cellules qui sont groupées en série. Il est indépendant de la fenêtre d'accés.

Configuration	Temps moyen d'accés
8 bits	16 To
16 bits	8 To
32 bits	4 To
64 bits	2 To
128 bits	To
256 bits	To/2

Fig 15 : Temps moyen d'accés avec permutation circulaire

Pour une fenêtre donnée il est possible de mettre en oeuvre le découpage de l'anneau en sous boucles dans le but d'amener plus rapidement un article dans une cellule donnée. Dans la suite des calculs nous prendrons comme hypothèse que la lecture s'effectue à la volée.

La notation suivante est employée. A chaque fenêtre d'accés qui est matérialisée par la liaison entre deux cellules est associé un numéro qui est le même que celui de la cellule qui suit la fenêtre d'accés.

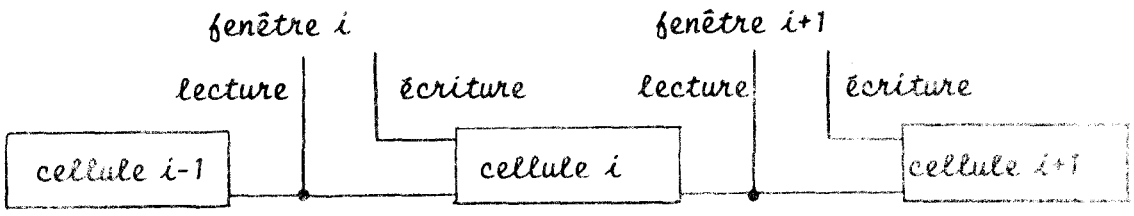


Fig 16 : Numérotation des fenêtres d'accès

L'exploitation de la table des prédécesseurs permet de déterminer pour chaque fenêtre, le temps minimal nécessaire pour amener le contenu d'une cellule quelconque. En prenant comme hypothèse un accès équiprobable aux cellules, il est alors possible de calculer un temps d'accés moyen pour chaque fenêtre.

exemple : configuration 8 bits, fenêtre 5

temps	cellule accessible
To	4, 5, 12
2 To	3, 11, 13
3 To	2, 6, 10, 14, 20
4 To	1, 9, 15, 15, 21, 7
5 To	0, 8, 24, 18, 22, 28
6 To	31, 23, 25, 17, 27, 29
7 To	30, 26, 16

Fig 17 : Temps minimal d'accès pour chaque cellule

temps moyen d'accés =4, 22 To
temps d'accés maximal 7 To

Un calcul identique peut être fait pour toutes les fenêtres dans une configuration donnée. Le tableau figure 18 fournit l'ensemble des résultats dans la configuration 8 bits.

fenêtre	tps accés moyen	tps accés max	fenêtre	tps accés moyen	tps accés max
0, 16	6,16 To	12 To	8, 24	4,59 To	8 To
1, 17	7,09 To	13 To	9, 25	4,18 To	9 To
2, 18	5,21 To	10 To	10, 26	4,53 To	10 To
3, 19	4,71 To	9 To	11, 27	4,71 To	11 To
4, 20	4,66 To	8 To	12, 28	4,65 To	10 To
5, 21	4,22 To	7 To	13, 29	4,34 To	9 To
6, 22	4,53 To	8 To	14, 30	4,91 To	10 To
7, 23	4,59 To	8 To	15, 31	5,22 To	11 To

Fig 18 : Temps accés moyen, configuration 8 bits

Ces résultats font apparaître que toutes les fenêtres d'accés ne sont pas équivalentes. D'autre part il est à remarquer que pour la plupart, le temps d'accés moyen ≤ 5 To = Log_2 N x To où N est égal au nombre de cellules c'est à dire 32.

Un calcul identique peut être fait pour toutes les configurations. Dans chacun des cas il existe une fenêtre privilégiée pour lequel le temps moyen d'accés est $\leq \text{Log}_2$ N x To. Le tableau figure 19 regroupe les différents résultats. Pour les configurations où la taille des mots est supérieur à 8 bits, on exploite l'information sur plusieurs fenêtres en parallèle. On utilise pour désigner la fenêtre la notation suivante (i_1, i_2, ...) où i_1, i_2 ... représentent les numéros des fenêtres qui sont exploitées en parallèle dans chacun des anneaux.

configuration	fenêtre	tps moyen d'accés	tps d'accés max
8 bits	9, 25	4,18 To	9 To
16 bits	(28,12)(30, 14)	2,75 To	8 To
32 bits	(31,7,15,23)	1,87 To	3 To
64 bits	(0,4,8,12,16 20,24,28)	1,25 To	2 To

Fig 19 : Temps d'accés moyen suivant la configuration

Pour les configurations 128 et 256 bits,le temps d'accés moyen ne peut pas être modifié. En effet les cellules sont groupées par paires ou rebouclées sur elles-mêmes et il n'est pas possible de modifier les liaisons. Les temps d'accés moyens sont respectivement To et To/2.

7.2. Reconfiguration par le multiplexeur de sortie

La sortie d'une cellule est constituée par un multiplexeur à deux voies. L'information qui est présentée à l'entrée de la cellule suivante provient soit de la sortie de la cellule après avoir transité le long du milieu de propagation soit directement de l'entrée de la cellule Le premier objectif de ce dispositif était de permettre l'extraction d'une cellule en vue d'une exploitation par un système extérieur qui assure lui même le contrôle de la circulation.

La technologie MOS statique, permet d'arrêter l'horloge de contrôle sans perdre l'information qui est stockée dans la cellule. Dés lors, ce dispositif peut être utilisé pour modifier la longueur de l'anneau. Toutefois les temps de propagation et de commutation au travers des opérateurs de court circuit exigent de réduire la fréquence de l'horloge dans le cas où un grand nombre de cellules sont court-circuitées.

On peut remarquer immédiatement, que ce mécanisme de régulation de la longueur de l'anneau est plus souple, La reconfiguration par le multiplexeur d'entrée n'autorisant que des boucles comportant un nombre pair de cellules et les possibilités étaient fonction de la configuration. Ici, au contraire le nombre de cellules dans une boucle peut être quelconque et ceci, quelque soit la configuration.

En court-circuitant plusieurs cellules consécutives il est possible d'échanger le contenu de deux cellules, Le temps nécessaire pour réaliser cette opération est To (temps de circulation dans une cellule) et il est indépendant des cellules considérées. L'intérêt essentiel d'une telle application réside dans la réduction du temps d'accés, En effet l'information stockée dans une cellule quelconque peut être amenée devant n'importe quelle fenêtre avec un temps moyen d'accés de To/2,

Enfin il est facile de montrer que toute permutation d'articles peut se décomposer sous la forme d'un produit de fonctions d'échange. Cependant la durée d'une telle opération risque d'être longue. En effet le nombre de permutations élémentaires est très limité.

L'emploi de la reconfiguration par le multiplexeur d'entrée et le multiplexeur de sortie peut se faire simultanément autorisant ainsi un large champ d'application de la mémoire circulante. La définition d'un système capable de gérer cette reconfiguration ne dépendant que de l'application.

8) *Architecture de Maud avec la mémoire circulante*

L'anneau de mémoire circulante est utilisé dans la *configuration "mot court"* soit 16 bits. Il se compose donc de 16 cellules de 1 K x 16 bits. 16 fenêtres d'accés sont disponibles permettant des accés simultanés à 16 processeurs. Ceux-ci accédent à l'information qui se présente au niveau de leur fenêtre, à la volée.

La figure 20 présente la structure générale de l'implémentation de Maud autour de la mémoire circulante. Les processeurs d'exécution sont placés les uns à la suite des autres sur des fenêtres consécutives. Ensuite, dans le sens de la circulation on trouve successivement les processeurs constructeur et mise à jour ; leur emplacement par rapport aux processeurs d'exécution ne doit pas être quelconque. Il permet de minimiser les temps de circulation. D'autre part il semble souhaitable de conserver la structure pipeline qui apparait au niveau du modèle. Il sera également nécessaire d'ajouter un processeur d'entrée-sortie afin de pouvoir communiquer avec le monde extérieur. Il convient de noter qu'il est possible d'implémenter plusieurs processeurs mise à jour ou constructeurs si nécessaire.

8.1. *L'anneau de mémoire circulante et les caractéristiques du modèle*

Les contraintes imposées par le modèle portaient essentiellement sur la nécessité de pouvoir assurer des accés simultanés et des traitements associatifs sans exiger une gestion complexe de l'ensemble.

Les deux fonctions de base de la mémoire de Maud sont effectivement implémentées. L'anneau constitue bien évidemment un outil de stockage

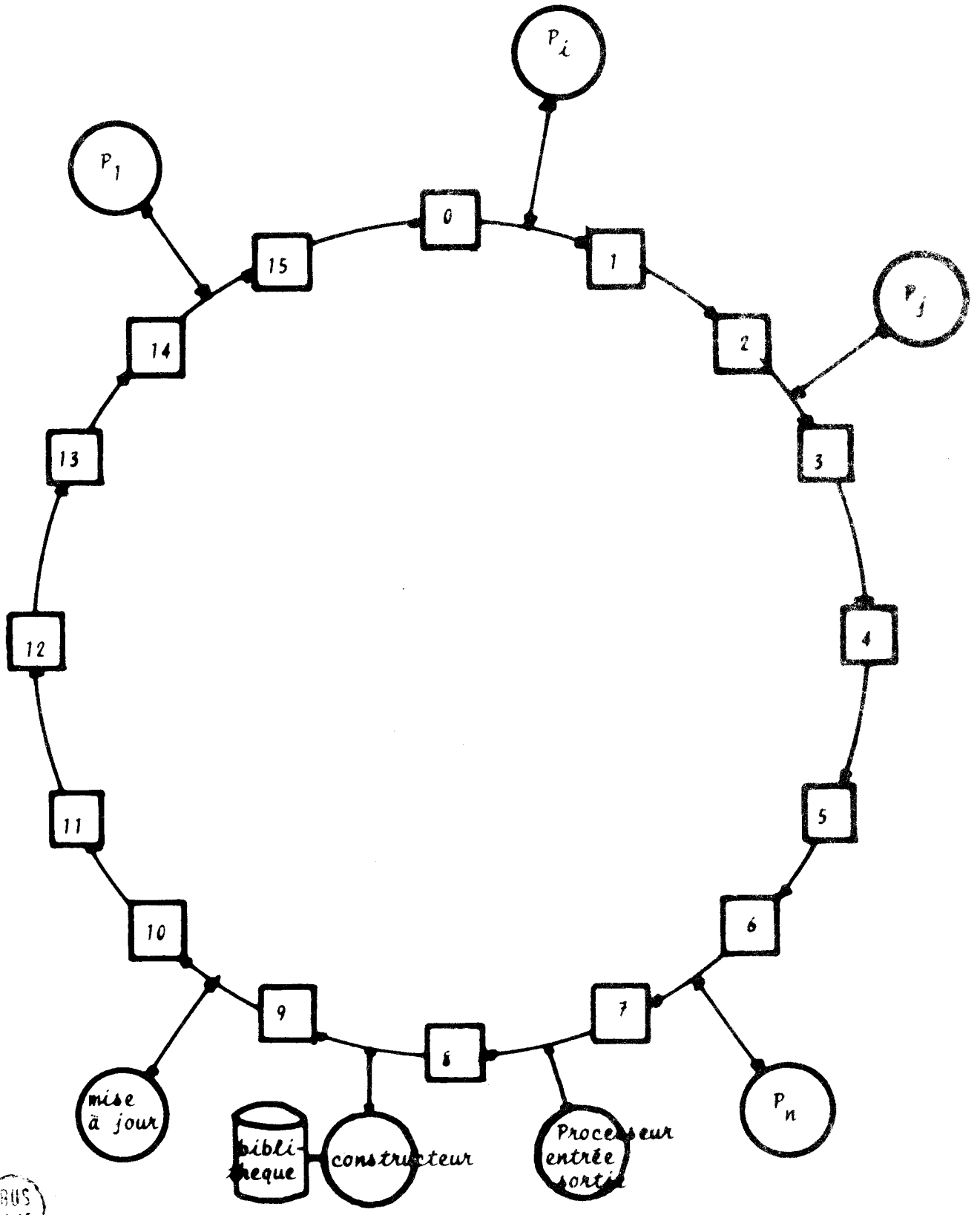


Fig 20 : Structure de l'implémentation de Maud

Enfin il est facile de montrer que toute permutation d'articles peut se décomposer sous la forme d'un produit de fonctions d'échange. Cependant la durée d'une telle opération risque d'être longue. En effet le nombre de permutations élémentaires est très limité.

L'emploi de la reconfiguration par le multiplexeur d'entrée et le multiplexeur de sortie peut se faire simultanément autorisant ainsi un large champ d'application de la mémoire circulante. La définition d'un système capable de gérer cette reconfiguration ne dépendant que de l'application.

8) *Architecture de Maud avec la mémoire circulante*

L'anneau de mémoire circulante est utilisé dans la *configuration "mot court"* soit 16 bits. Il se compose donc de 16 cellules de 1 K x 16 bits. 16 fenêtres d'accés sont disponibles permettant des accés simultanés à 16 processeurs. Ceux-ci accédent à l'information qui se présente au niveau de leur fenêtre, à la volée.

La figure 20 présente la structure générale de l'implémentation de Maud autour de la mémoire circulante. Les processeurs d'exécution sont placés les uns à la suite des autres sur des fenêtres consécutives. Ensuite, dans le sens de la circulation on trouve successivement les processeurs constructeur et mise à jour ; leur emplacement par rapport aux processeurs d'exécution ne doit pas être quelconque. Il permet de minimiser les temps de circulation. D'autre part il semble souhaitable de conserver la structure pipeline qui apparait au niveau du modèle. Il sera également nécessaire d'ajouter un processeur d'entrée-sortie afin de pouvoir communiquer avec le monde extérieur. Il convient de noter qu'il est possible d'implémenter plusieurs processeurs mise à jour ou constructeurs si nécessaire.

8.1. *L'anneau de mémoire circulante et les caractéristiques du modèle*

Les contraintes imposées par le modèle portaient essentiellement sur la nécessité de pouvoir assurer des accés simultanés et des traitements associatifs sans exiger une gestion complexe de l'ensemble.

Les deux fonctions de base de la mémoire de Maud sont effectivement implémentées. L'anneau constitue bien évidemment un outil de stockage

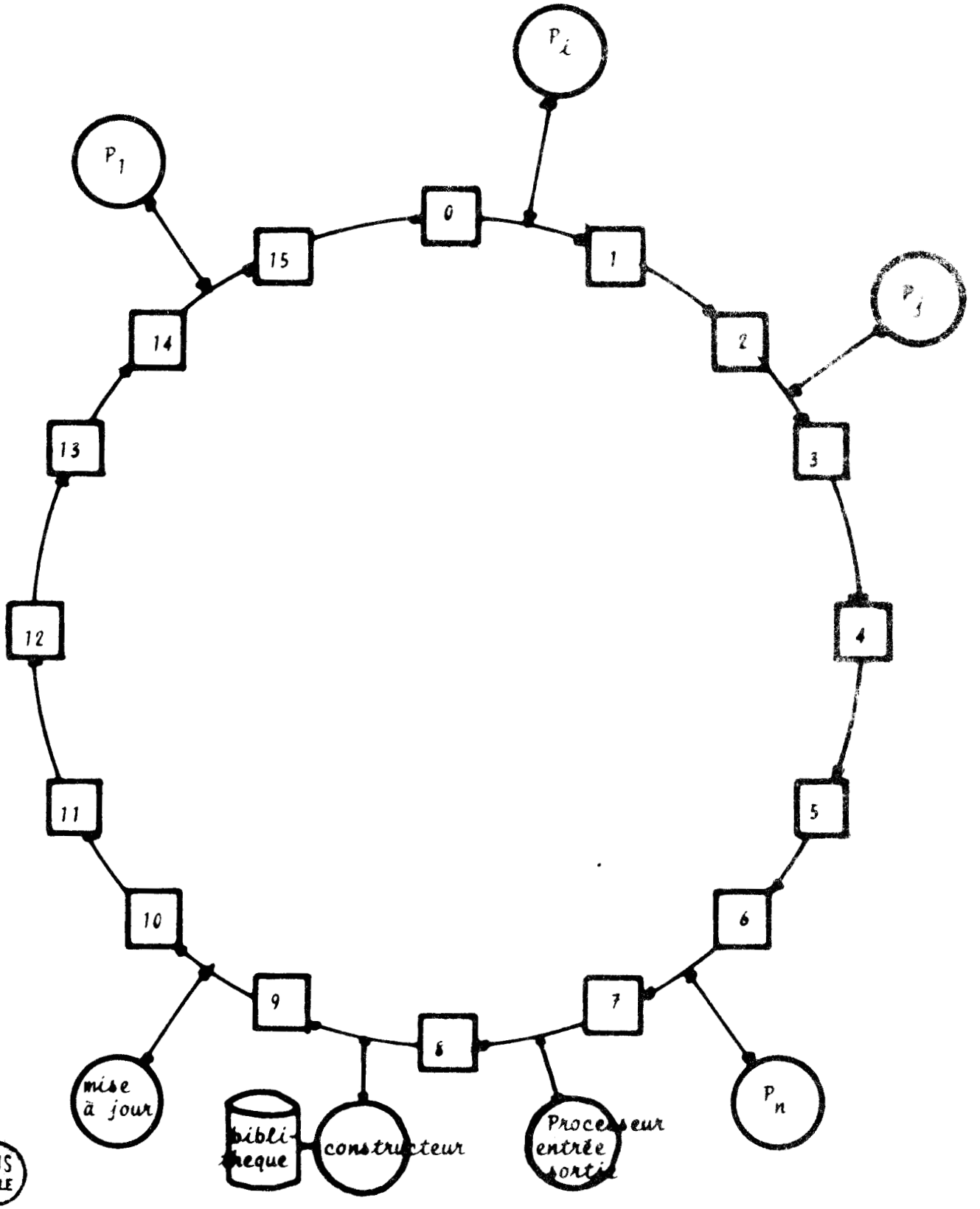


Fig 20 : Structure de l'implémentation de Maud

de capacité importante et de par la circulation de l'information, l'utilisation en tant qu'outil de communication dans un système multiprocesseur apparaît parfaitement naturelle.

C'est la possibilité d'effectuer des accés simultanés, sans avoir à résoudre de problèmes de conflits d'accés, qui a déterminé, pour une grande part notre choix. Les voies d'accés pour chaque processeur étant physiquement disjointes, il n'est pas nécessaire d'avoir un dispositif de gestion de priorité. D'autre part la circulation de l'information rend aisés les traitements associatifs. Il suffit en effet d'observer les informations disponibles au niveau d'une fenêtre, sans devoir adjoindre des mécanismes logiques complexes.

Le temps d'accés n'est certes pas en faveur de ce choix. Cependant les contraintes à ce niveau sont moins rigoureuses. Bien qu'étant une caractéristique importante du système, c'est le rapport du temps d'exécution des blocs par rapport au temps de circulation qui aura une influence prépondérante pour les performances globales.

8.2. *Organisation des informations*

La mémoire commune doit contenir tous les objets utilisés par Maud : Blocs exécutables, blocs en attente, demandes d'exécution, domaines de sortie. D'autres types d'objets seront peut-être nécessaires pour les communications avec les périphériques ou les erreurs mais leur forme sera proche de celle des objets cités.

Le choix d'une organisation particulière doit tenir compte des caractéristiques des objets qui sont communiqués entre les processeurs :

- Les objets varient dans leur nombre et dans leur taille. Le nombre de blocs peut varier dans une grande proportion d'un programme à l'autre et si la variation de taille est faible pour les blocs exécutables ou pour les blocs en attente, il n'en sera pas de même pour les domaines de sortie ou pour les demandes d'exécution. Le nombre de résultats fournis par un bloc et le nombre de paramètres transmis lors d'une demande d'exécution sont parfaitement quelconques.

- Les objets ont une durée de vie limitée dans l'anneau. L'ensemble des objets qui sont utilisés par Maud sont produits par un opérateur et sont consommés par un autre. Par exemple les demandes d'exécution n'effectuent pas un tour complet de l'anneau. Elles sont produites par les processeurs d'exécution et sont capturées immédiatement par le processeur constructeur qui se trouve derrière.

- Chaque objet n'est utilisé qu'une fois et une seule par un opérateur. Ceci signifie en d'autres termes que lors de la capture d'un objet, la place qu'il occupait dans l'anneau peut être libérée.

Plusieurs solutions peuvent être envisagées pour le stockage des différents objets :

- L'*anneau est divisé en plusieurs parties* (en nombre égal au nombre de types d'objets) de longueur égale ou non, chacune regroupant les objets d'un type. Cette solution n'est pas satisfaisante car elle ne fait qu'accroître les problèmes de gestion et fait une hypothèse rigide sur les capacités nécessaires.

- Les *objets sont rangés dans l'anneau les uns derrière les autres* dans un ordre quelconque. Chaque objet commence par un indicateur afin de le distinguer des autres et de désigner le type de l'objet. Pour pouvoir reconnaitre un indicateur d'une autre information, celui-ci doit être différent des informations qui peuvent se trouver dans l'anneau.

Il existe deux inconvénients majeurs avec cette méthode. Le premier est lié à la gestion de l'espace libre. La taille variable des objets qui peuvent être stockés dans l'anneau entrainera inévitablement le phénomène bien connu d'émiettement. En conséquence il serait nécessaire d'effectuer de temps à autre ou en permanence une opération de compactage afin de regrouper les informations utiles. Dans le cas des mémoires circulantes c'est une opération complexe qui nécessite obligatoirement deux fenêtres d'accés. Le deuxième réside dans la réalisation des communications entre les processeurs et l'anneau. Ceux-ci ne peuvent capturer ou déposer un objet qu'aprés avoir observé un indicateur. Comme ce dernier n'a pas de place bien définie, les processeurs doivent observer de façon permanente le contenu de l'anneau au niveau de la fenêtre d'accés.

- L'*anneau est divisé en secteurs logiques* de tailles identiques appelés *cadres*. Les cadres circulent en permanence dans l'anneau et les processeurs peuvent lire le contenu ou déposer un objet. Le contenu d'un cadre est spécifié par son *entête* qui doit comporter notamment le *type de l'objet* qui est stocké. Un cadre qui est inoccupé possède l'entête "vide" qui indique aux processeurs qu'ils peuvent y déposer quelque chose.

Lors d'une communication anneau-processeur, ce dernier attend que l'entête d'un cadre passe devant sa fenêtre d'accés. L'avantage de cette solution est qu'il n'est plus indispensable d'assurer une observation permanente. L'entête d'un cadre n'est pas fixé une fois pour toutes, il est modifié pour toute opération de lecture ou d'écriture du contenu du cadre.

Le choix de la taille des cadres doit tenir compte de considérations technologiques et de la taille des objets qu'ils doivent contenir . Le deuxième point évoqué risque d'avoir une influence importante sur les performances du système. Le temps d'exécution d'un bloc par un processeur peut dépendre en partie de la taille de celui-ci.

Un cadre peut contenir un bloc complet sans que sa partie interne, c'est à dire les instructions et les objets locaux, soit trop réduite. Dans ce cas la taille des blocs est définie une fois pour toutes et impose des contraintes au niveau de la compilation et du découpage des blocs. A l'opposé, il est possible de réduire la taille des cadres [LEC 79c]. Un bloc occupera alors plusieurs cadres et un processeur ne pourra commencer l'exécution que lorsqu'il aura rassemblé les différents composants constitutifs d'un bloc.

Quelle que soit la solution qui sera adoptée et qui sera, dans tous les cas, le résultat d'un compromis, il semble souhaitable de ne placer qu'un type d'objet par cadre afin de simplifier au maximum les échanges. Même si cela conduit à une sous utilisation temporaire de l'espace mémoire, le traitement global y gagnera en rapidité.

8.3. Les échanges opérateurs - anneau

Les opérateurs étant réalisés à l'aide de *microprocesseurs standards* ceux-ci ne peuvent travailler que sur des informations stockées dans une mémoire classique. Les échanges entre les opérateurs et l'anneau auront pour but d'effectuer le transfert des objets, de l'anneau vers une mémoire locale ou vice versa. Pour des raisons technologiques et afin de libérer le processeur de ces différentes opérations, ce travail sera confié à une unité d'échange associée à chaque opérateur. Les fonctions associées à la réalisation de cette unité seront étudiées plus en détail dans le chapitre suivant.

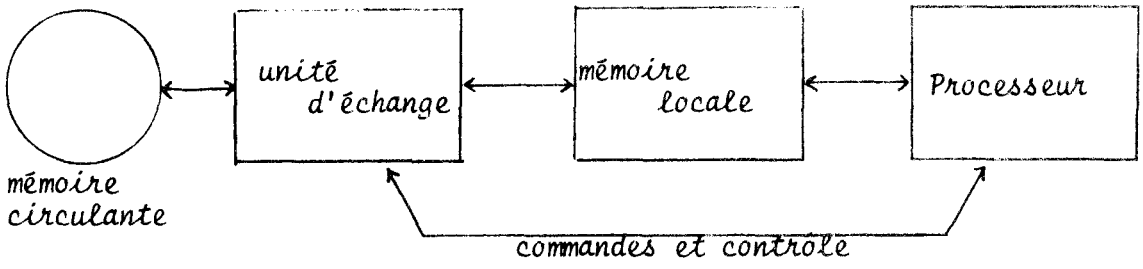


Fig 21 : Echanges entre les opérateur et l'anneau

9. Autres utilisations de la mémoire

9.1. Mémoire associative

Le principe de base d'une mémoire associative consiste à accéder à l'information stockée, non pas à l'aide d'une adresse comme dans le cas d'une mémoire conventionnelle, mais par le contenu. Ceci revient à dire qu'une comparaison est effectuée simultanément entre un mot "clé" qui caractérise l'information recherchée et l'ensemble des mots présents en mémoire.

On retrouve sur les mémoires associatives qui ont été réalisées trois types d'opérations fondamentales : interrogation, écriture et lecture.

- L'interrogation consiste à opérer une comparaison entre le contenu de chaque mot et celui d'un registre contenant le mot clé. Généralement on rajoute la possibilité de masquer un certain nombre de bits, ceux-ci n'intervenant pas dans la comparaison. Le résultat de cette comparaison s'inscrit pour chaque mot dans un bit réservé à cet effet (registre A).

- L'écriture entraine la modification des mots repérés par le registre A par inscription du mot clé dans les positions de bits qui ne sont pas marquées. Ce traitement s'effectue simultanément pour tous les mots.

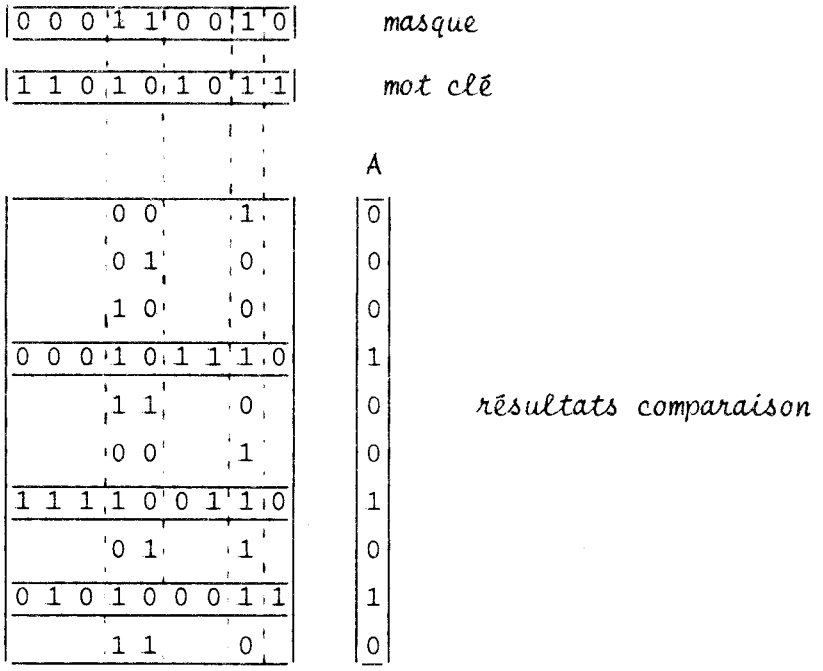


Fig. 22 : Mémoire associative

- La lecture permet de récupérer des résultats qui sont repérés par le registre A.

Les mémoires technologiquement associatives qui sont proposées sur le marché sont de faibles capacités. Pour résoudre ce problème une première solution consiste à utiliser une mémoire classique et à lui associer un dispositif logique capable d'assurer un balayage séquentiel de l'ensemble des positions. Il apparait tout de suite qu'il s'agit d'une opération longue et couteuse.

L'utilisation d'une mémoire circulante permet de résoudre partiellement cette difficulté. Il n'est plus nécessaire d'utiliser un dispositif de balayage puisque la circulation de l'information qui est le principe de base de cette mémoire remplit parfaitement cette fonction.

Dans le cadre de notre implémentation, deux modes de fonctionnement sont possibles :

- La *recherche parallèle*. L'anneau de mémoire est utilisé dans la configuration parallèle. On dispose donc de 32 boucles élémentaires d'une capacité de 1 K

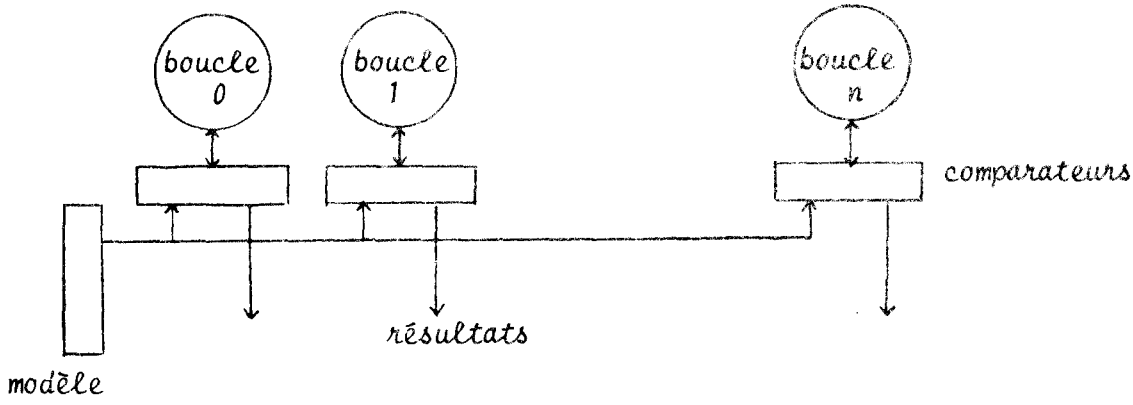


Fig 23 : Mémoire associative avec recherche parallèle

Un dispositif de comparaison entre le mot clé et l'information stockée est associé à chaque fenêtre d'accés auquel on peut adjoindre un dispositif de marquage. Le temps d'accés sera égal à To c'est à dire le temps d'une rotation complète pour une boucle.

- La *recherche série*. Dans ce mode de fonctionnement, on n'utilise q'une seule fenêtre d'accés. Si nous utilisons le même principe que pour la recherche parallèle le temps d'accés devient proportionnel au nombre de cellules. Il est possible d'exploiter la reconfiguration pour améliorer les performances de la façon suivante.

Pour pouvoir amener le plus rapidement possible l'information cherchée au niveau de la fenêtre d'accés, il est indispensable de pouvoir déterminer dans quelle cellule elle se trouve. Ceci impose que les objets stockés dans la mémoire soient ordonnés suivant les clés d'accés. La fourniture de la clé d'accés permet donc de connaître la position de l'information dans la mémoire.

La détermination des permutations à réaliser peut se faire sans ambiguité si à tout instant il est possible de connaître la position de la cellule qui contient l'information cherchée par rapport à la fenêtre d'accès.

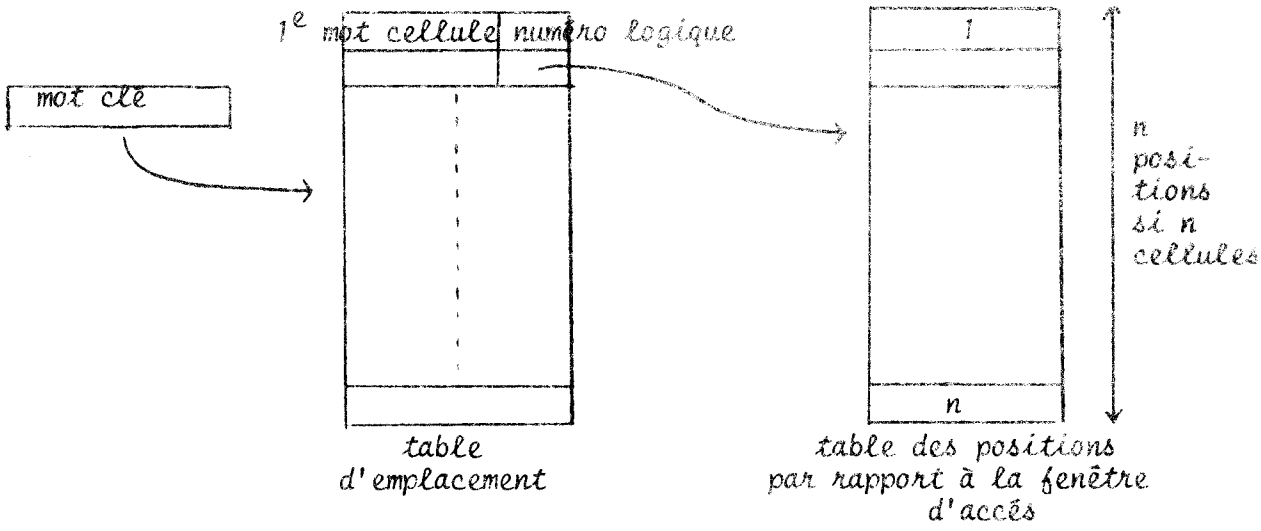


Fig 24 : Mécanisme de recherche série

La table d'emplacement associe au premier mot de chaque cellule un numéro logique. La connaissance du mot clé, par simple parcours de cette table fournit donc un numéro logique. Celui-ci permet de pointer sur une position de la table d'emplacement qui indique la position de la cellule recherchée et donc la suite des permutations à effectuer.

Toute permutation entraine une modification de la table des emplacement. La mise à jour de cette table se fait en effectuant la permutation inverse sur les positions.

Dans le cas où les permutations sont réalisées à l'aide du multiplexeur d'entrée et suivant la fenêtre d'entrée sélectionnée, le nombre de permutations élémentaires dont il faut disposer est limité. L'implémentation de la table peut se faire à l'aide de registres, un réseau d'interconnexion simulant les permutations.

exemple : configuration 16 bits fenêtre d'accés (28, 12).
La table des positions sera implémentée dans 16 registres. La figure 25 indique les communications nécessaires entre les registres. Le temps d'accés moyen sera égal à 2,75 To.

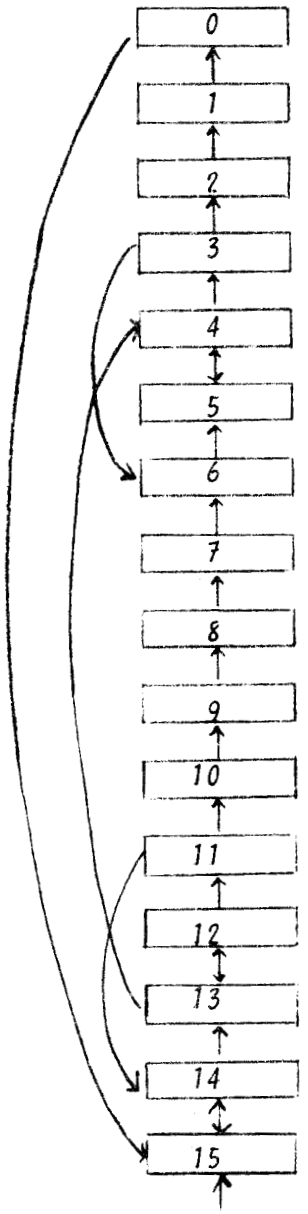


Fig 25 : Implémentation de la table des positions

Dans le cas où les permutations sont réalisées à l'aide du multiplexeur de sortie, il suffit d'échanger deux lignes de la table. Le temps d'accés moyen est égal à To/2.

Une telle mémoire peut être utilisée comme mémoire intermédiaire entre une unité de disque et une mémoire adressable. L'anneau peut assurer le stockage de plusieurs pistes et la reconfiguration permet d'accéder plus rapidement au niveau du secteur.

9.2. Mémoire paginée

Le principe est de diviser la mémoire en plusieurs boucles indépendantes chacune constituant un niveau de hiérarchie dans l'organisation mémoire. Une telle organisation a été proposée par Tien Chi Chen [TIEN 75]. Une seule fenêtre d'accès est disponible et permet d'accéder à l'information qui est stockée dans la boucle de niveau le plus haut. Les pages sont amenées progressivement du niveau le plus bas vers le niveau le plus haut en fonction des demandes.

L'anneau peut être décomposé en 8 boucles de 4 cellules (fig. 26). La transmission d'une page entre deux boucles s'effectue en échangeant le contenu de deux cellules. Lors de cette opération, les horloges de contrôle des boucles considérées doivent être identiques. Autrement, les horloges peuvent être différentes pour chacune des boucles.

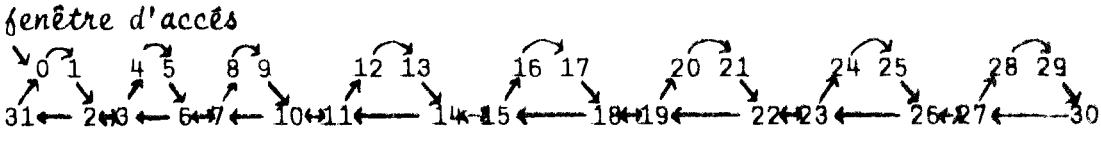


Fig. 26 : Mémoire paginée

Dans l'hypothèse où il n'y a pas de défaut de page, le temps d'accès moyen est de 2 To.

9.3. Mémoire de taille variable

L'inconvénient majeur d'une mémoire séquentielle est son temps d'accès. L'idée est d'ajuster au mieux sa longueur à l'application. Il faut donc pouvoir retirer ou rajouter un certain nombre de cellules en fonction d'une demande. Deux solutions sont possibles.

- La mise hors service de plusieurs cellules en les court-circuitant permet de réduire momentanément la taille de l'anneau.

- L'extension de l'anneau peut se faire en disposant, au niveau d'une fenêtre d'accès, de quelques cellules supplémentaires. Elles sont interconnectées (fig. 27) de manière à pouvoir les introduire une à une dans l'anneau.

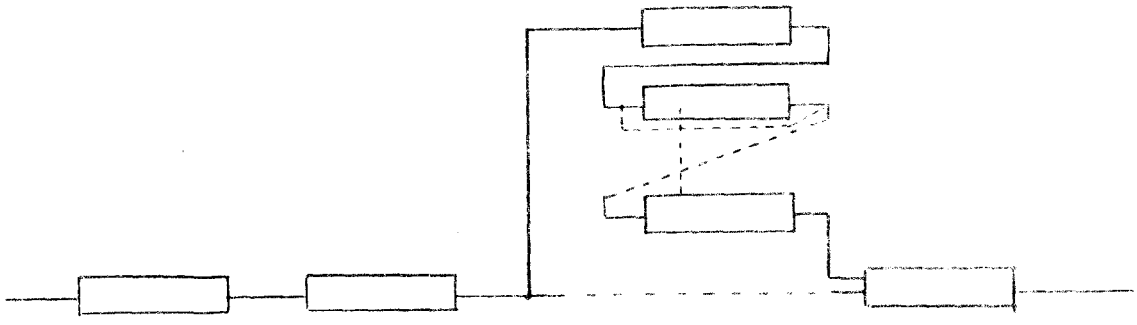


Fig 27 : Anneau de taille variable

Un dispositif centralisé assure la gestion de la longueur de l'anneau en fonction de la charge de l'anneau qu'il peut déterminer soit par l'observation de l'ensemble des fenêtres en parallèle, soit par observation au niveau d'une fenêtre et en effectuant un bilan sur un tour complet de l'anneau.

Cette manière de faire permet d'assurer une régulation de la charge de l'anneau et pourra, par conséquent, être exploitée au niveau de Maud afin d'optimiser les temps de circulation.

9.4. Mémoire de tri

De nombreux auteurs ont proposé des algorithmes d'accés à une cellule et de générations d'une permutation quelconque [STON 71] [STON 72] [STON 75] [LANG 76] [WONG 77] [CLERC 79].

Stone a étudié plus particulièrement les propriétés du réseau "perfect shuffle" (mélange parfait). Si on considère un ensemble de N éléments le "perfect shuffle" est une permutation qui est identique au parfait mélange d'un paquet de cartes. Elle peut s'écrire

$$S(i) = 2i \quad \text{si } 0 \leq i \leq N/2\ 1$$
$$= 2i+1-N \quad \text{si } N/2 \leq i \leq N-1$$

Il a demontré que combinée avec la permutation qui consiste à échanger les positions paires et impaires, il était possible de décomposer une permutation quelconque de N éléments.

D'une manière similaire, il serait possible d'exploiter les propriétés de la reconfiguration. Toutefois la définition d'un mécanisme de décomposition automatique semble particulièrement complexe.

10. Conclusion

La mémoire que nous avons définie dépasse largement du point de vue des possibilités, les besoins de MAUD. A ce titre elle apparait comme un outil de stockage et de communication adaptée à une structure multi-processeurs. Comme nous l'avons présenté, la simplicité de mise en oeuvre, et la grande variété des accés possibles permet d'envisager son utilisation dans le cadre d'autres architectures.

CHAPITRE IV

LES ÉCHANGES
PROCESSEURS - MÉMOIRE

L'exécution d'un programme dans Maud, nécessite de nombreuses communications entre les différents processeurs et l'anneau de mémoire circulante. Les informations qui sont transmises au cours de ces communications peuvent varier tant du point de vue de leur taille que du point de vue de leur type. La définition des procédures d'échange et la mise en place d'un mécanisme capable de gérer ces communications doivent tenir compte non seulement de l'organisation des informations dans la mémoire circulante mais également de contraintes techniques qui sont liées à la réalisation des processeurs.

- Dans le chapitre précédent, nous avons envisagé quelques organisations possibles pour le stockage des différents objets. Le choix qui doit être fait résulte d'un compromis à effectuer entre le taux d'occupation de la mémoire et la facilité d'accéder à l'information. La solution que nous avons retenue, consiste à découper l'anneau en secteurs logiques de tailles identiques appelés cadres. Chacun des cadres définis peut recevoir n'importe quel type d'objet : blocs exécutables, blocs en attente,domaine de sortie, ou demande d'exécution. Les premiers mots du cadre sont utilisés pour stocker un entête qui caractérise le contenu, c'est à dire son type et le nom du bloc, si nécessaire. D'autres renseignements pourront être rajoutés tels que le nombre de cadres nécessaires pour le stockage d'un bloc dans l'hypothèse où la taille de ce dernier est supérieure à celle d'un cadre, le numéro du cadre ... Le type d'un cadre n'est pas fixé une fois pour toutes. En effet son contenu change chaque fois qu'il est utilisé par un processeur. En particulier il doit exister un type vide qui désigne un cadre inoccupé et qui peut donc être utilisé par n'importe quel processeur pour le dépot d'un objet.

- Les contraintes techniques proviennent, pour une grand part, des choix qui ont été pris, au niveau de la réalisation des processeurs. Ceux-ci seront construits autour de microprocesseurs standards. Tous ceux qui sont disponibles, à ce jour, sur le marché, utilisent l'adressage, pour accéder aux instructions et aux données. Or à cette notion a été substituée celle d'accés par nom au niveau de Maud. Cette remarque a pour conséquence immédiate, la nécessité de transférer, pour tout traitement, le contenu des cadres de l'anneau dans une mémoire locale, adressable,associée à chaque processeur.

Dans une telle hypothèse, deux questions se posent : qui a l'initiative de ces transferts et qui en assure la gestion ?

Pour le premier point, la solution la plus naturelle est de confier cette charge au processeur lui-même. Celui-ci dispose, en effet, de toutes les informations nécessaires pour décider de l'instant auquel un transfert doit avoir lieu et sur quel type d'information il doit porter.

Lorsqu'un transfert a été demandé par un processeur, il faut gérer celui-ci, c'est à dire chercher un cadre correspondant à la demande et effectuer le transfert proprement dit. La difficulté provient essentiellement de la prise de décision pour l'acquisition d'une information. La recherche d'un cadre se fait de manière associative. Dans l'hypothèse, où le processeur ne peut travailler, qu'au niveau de sa propre fenêtre d'accés, cette prise de décision et, dans certains cas, l'acquisition du premier mot doivent pouvoir se faire au cours du même cycle d'horloge. Ces deux opérations ne peuvent être prises en charge par le processeur lui-même pour des raisons de vitesse.

La deuxième solution consiste à observer les entêtes des cadres avant leur passage devant la fenêtre d'accés. De cette manière le temps disponible pour décider de l'acquisition devient plus conséquent.

Pour certains types de transfert cet aspect de gestion peut demander beaucoup de temps. Il ne semble pas souhaitable d'augmenter la charge des processeurs. Ceci n'aurait pour effet que de réduire les performances globales du système.

Toutes ces considérations nous ont amenés à adjoindre à chaque processeur une unité d'échange qui soit capable, à partir d'une demande fournie par le processeur, de prendre en charge le transfert.

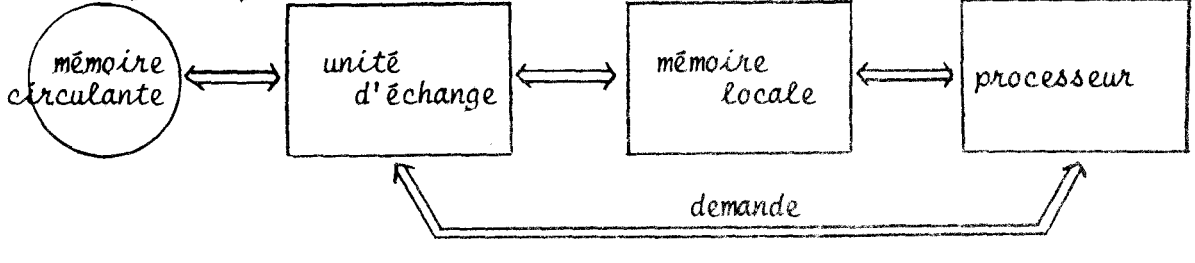


Fig 1 : Interface processeur - mémoire circulante

Pour des raisons de modularité, il semble souhaitable de définir une unité d'échange générale pour laquelle il sera possible d'affiner le rôle en fonction du processeur auquel elle sera associée.

1) *Procédures d'échanges entre les processeurs et l'anneau*

Lorsqu'un processeur veut entamer une communication avec l'anneau, il adresse une demande à son unité d'échange. Celle-ci doit définir sans équivoque, l'information souhaitée. Les opérations qui seront effectuées au cours de ces échanges dépendent du processeur et du type de cadre demandé.

Trois types d'action peuvent être opérés pendant le cycle d'horloge de la mémoire circulante.

- *opération de lecture*. Un mot est transféré de l'anneau vers la mémoire locale. Le contenu de l'anneau reste inchangé.

- *Opération d'écriture*. Un mot est transféré de la mémoire locale vers l'anneau. Le contenu de la mémoire locale reste inchangé.

- *opération mise à jour*. Il s'agit en fait d'une opération de lecture suivie d'une opération d'écriture au cours d'une seule période d'horloge.

L'existence de cette dernière opération apporte la possibilité d'effectuer l'échange de deux blocs différents. Ceci permettra, dans certains cas, de réduire de moitié la durée des échanges, en effet, l'opération d'écriture dans un cadre vide pour le dépot d'un bloc et la lecture d'un bloc se réduisent à un simple échange entre le contenu d'un cadre et le bloc qui se trouve dans la mémoire locale.

Il semble souhaitable, pour des raisons de modularité, de définir une unité d'échange générale, laquelle pourra être spécialisée en fonction de son application propre. Le choix entre la solution "blocs en attente dans l'anneau" ou "domaines de sortie dans l'anneau" [LEC 79c] n'a pas de conséquence directe sur l'étude de l'unité d'échange. Il n'entrainera que des modifications pour les spécifications des unités d'échange propres aux processeurs de traitement et au processeur mise à jour.

1.1. Les processeurs de traitement

Les procédures d'échanges entre les processeurs de traitement et l'anneau sont extrêmement simples. L'ensemble des opérations qui peuvent être effectuées se trouve résumé dans le tableau de la figure 2. Le processeur est dit *actif* s'il est en train d'exécuter un bloc exécutable, en *attente*, s'il cherche à déposer un objet dans l'anneau et *inactif* s'il n'a plus aucun traitement (exécution ou dépôt) à assurer.

indicateur cadre avant	opération	indicateur cadre après	état processeur
Bloc exécutable	néant	Bloc exécutable	actif
	lecture	cadre vide	inactif
	mise à jour	Bloc en attente	attente
	mise à jour	Domaine de sortie	attente
cadre vide	écriture	Bloc en attente	attente
	écriture	Demande d'exécution	actif
	écriture	Domaine de sortie	attente
	néant	cadre vide	actif ou inactif

Fig 2 : Procédures d'échange processeur de traitement - anneau

Lorsqu'un processeur de traitement est inactif, il recherche un bloc exécutable. Après capture, le cadre devient vide, s'il n'y a pas de bloc en attente ou de domaine de sortie en attente de dépot. Dans le cas contraire il échange le bloc en attente de dépot avec le bloc exécutable. Les échanges ne se font pas entre une demande d'exécution et un bloc exécutable. Car après l'exécution d'une primitive *execbloc*, le processeur poursuit l'exécution du bloc.

Lorsqu'un cadre vide se présente, il peut y déposer si nécessaire un bloc en attente, une demande d'exécution, ou un domaine de sortie. Pour chacun de ces mouvements,il modifie l'indicateur de cadre en conséquence.

Ces procédures sont valables dans l'hypothèse où les blocs en attente restent dans l'anneau et les domaines de sortie sont pris par le

processeur mise à jour.Dans l'autre hypothèse le tableau se modifie de la manière suivante

indicateur cadre avant	opération	indicateur cadre après	état processeur
Bloc exécutable	néant lecture mise à jour	Bloc exécutable cadre vide Bloc en attente	actif inactif attente
cadre vide	écriture écriture néant	bloc en attente demande d'exécution cadre vide	attente attente actif ou inactif
domaine de sortie	écriture	domaine de sortie	attente

Fig 3 : Procédures d'échange processeur de traitement anneau

Seul le dépot des domaines de sortie s'opère de manière différente. Les couples (nom, valeur) sont rangés dans des emplacements vides dans un cadre de type domaine de sortie. Il faudra de plus définir les modalités de gestion de ces emplacements vides [LEC 79c].

1.2. Le processeur mise à jour

Le rôle qu'il joue au niveau de Maud est très important et ses performances risquent d'être critique vis à vis des performances globales du système. Ici encore nous devons envisager les deux solutions possibles. Le tableau figure 4 résume les échanges dans le cas où les blocs en attente restent dans l'anneau.

indicateur cadre avant	opération	indicateur cadre après
Bloc en attente	écriture écriture	Bloc en attente Bloc exécutable
Domaine de sortie	lecture	cadre vide

Fig 4 : Procédures d'échange processeur mise à jour - anneau

Le processeur mise à jour range dans sa mémoire propre tous les domaines de sortie, c'est à dire l'ensemble des couples (nom , valeur) et laisse les blocs en attente dans l'anneau. Toutefois le traitement que doit effectuer le processeur sur les blocs en attente n'est pas comparable à celui que doit faire le processeur de traitement. Il se contente de mettre à jour le domaine d'entrée. Pour chaque nom de communication appartenant à un domaine d'entrée, il compare ce nom avec tous ceux dont il dispose la valeur dans sa mémoire propre. Dans le cas où le résultat de la comparaison est positif, il transmet la valeur. Après passage devant le processeur mise à jour, le bloc qui était initialement en attente peut être devenu bloc exécutable si le domaine d'entrée est complet sinon il reste en attente. Ici les transferts se font sur des parties de cadre dont la taille peut varier d'une opération à l'autre. D'autre part, il faut résoudre le problème de la comparaison qui doit s'effectuer à la volée.

Dans le cas de la deuxième solution, les blocs en attente sont rangés dans la mémoire propre et les domaines de sortie continuent à circuler dans l'anneau. Le tableau précédent se modifie de la manière suivante :

indicateur cadre avant	opération	indicateur cadre après
Bloc en attente	lecture mise à jour	cadre vide Bloc exécutable
Domaine de sortie	lecture	domaine de sortie
cadre vide	écriture	Bloc éxécutable

Fig 5 : Procédures d'échanges processeur mise à jour - anneau

Le mécanisme de transmission des valeurs est identique au cas précédent sauf que la comparaison est faite entre un nom de communication issu d'un domaine de sortie et l'ensemble des noms de communication des domaines d'entrée des blocs en attente stockés dans la mémoire propre. Les blocs en attente pour lesquels le domaine d'entrée a été complété, deviennent blocs exécutables et sont placés dans l'anneau, à la place d'un bloc en attente ou d'un cadre vide. Aucune modification n'intervient sur l'indicateur domaine de sortie.

1.3. Processeur constructeur

Les procédures d'échange sont très similaires à celles des processeurs d'exécution , aux indicateurs près. Le tableau figure 6 les résume.

indicateur cadre avant	opération	indicateur cadre après
demande d'exécution	lecture mise à jour	cadre vide bloc en attente
cadre vide	écriture	bloc en attente

Fig 6 : Procédures d'échange processeur constructeur - anneau

Le processeur constructeur prend dans sa mémoire locale, l'ensemble des demandes d'exécution. Il crée des nouveaux blocs qui peuvent être échangés avec une demande d'exécution ou placés dans un cadre vide. Si le domaine d'entrée est complet, le processeur pourrait fournir immédiatement un bloc exécutable. Ceci permettrait de réduire partiellement l'activité du processeur mise à jour.

2) Principe de fonctionnement de l'unité d'échange

D'une manière très générale, la fonction de l'unité d'échange peut se définir de la manière suivante : Assurer le *contrôle des échanges* entre la *mémoire circulante de communication* et une *mémoire adressable locale*. Cette dernière est supposée, en outre être exploitée en mutuelle exclusion par le processeur local.

2.1. Objectif d'une réalisation

Sans préjuger de la nécessité de réaliser certaines unités spécifiques, répondant à des besoins particuliers, le circuit doit assurer une large gamme de services correspondants à des échanges variés entre les mémoires concernées.

Il apparait souhaitable de définir un outil général d'accés à la mémoire circulante qui ne tient pas compte au niveau de la conception

des applications, qu'il aura à exécuter et qui seront spécifiques à certains processeurs (processeur d'exécution, processeur mise à jour).

Le contrôle de l'unité d'échange par le processeur local, ne doit pas exiger un surcroit de travail trop important pour ce dernier. L'unité d'échange doit être capable, de fonctionner de manière parfaitement autonome. On assure de la sorte la meilleure indépendance temporelle et fonctionnelle entre les processeurs d'une part et l'anneau d'autre part.

2.2. *Déroulement d'un échange*

Quelle que soit la fonction propre de l'unité d'échange, le processus d'un échange peut être décomposé en plusieurs étapes :

- L'unité d'échange part d'un état "repos". Elle reçoit une commande de transfert qui lui précise, sans équivoque possible, la nature de l'information et du transfert qui doit être effectué.

- A l'aide des renseignements contenus dans la commande, l'unité d'échange se place dans un état de recherche de secteurs qui sont des ensembles de mots consécutifs stockés dans la mémoire circulante. Il faut remarquer immédiatement que le nombre de secteurs peut être quelconque. Toutefois un secteur est obligatoirement inclus dans un cadre. L'ensemble des opérations de recherche s'effectue à la volée par consultation du mot présent au niveau de la fenêtre d'accés du processeur.

- Lorsque le premier mot d'un secteur à transférer passe devant la fenêtre d'accés, le transfert proprement dit peut être initialisé. Les mots sont lus ou écrits séquentiellement.

- Il reste à vérifier les conditions de fin de transfert et de fin de commande. Si cette dernière porte sur plusieurs secteurs, l'unité se replace dans un état de recherche. Dans le cas contraire elle retourne à l'état repos en attente d'une commande ultérieure, tout en signalant au processeur la fin de la commande.

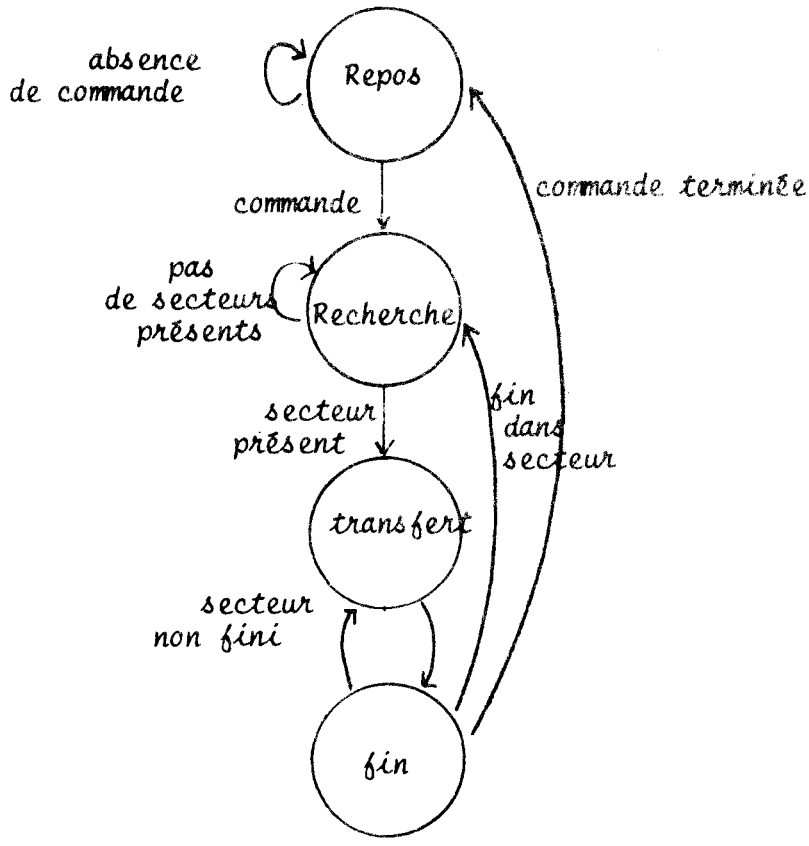


Fig 7 : Processus d'un échange processeur - anneau

Pour certaines applications, la durée de vie d'une commande est indéterminée et peut même avoir un caractère permanent. Dans ces conditions, le processeur doit se garder à tout moment la possibilité d'interrompre l'unité d'échange en vue de modifier la commande ou d'en fournir une nouvelle.

2.3. Décomposition de l'unité d'échange

Le processus d'un échange fait apparaitre deux fonctions particulières.

- *fonction d'observation et de recherche* en vue de délimiter les secteurs (début et fin de secteur) et de déterminer la durée des commandes.

- *fonction de transfert* en vue de gérer les échanges d'information proprement dits.

L'unité d'échange sera donc en fait composée de deux sous unités : *unité de recherche* et *unité de transfert*, chacune remplissant l'une des fonctions précédemment définies.

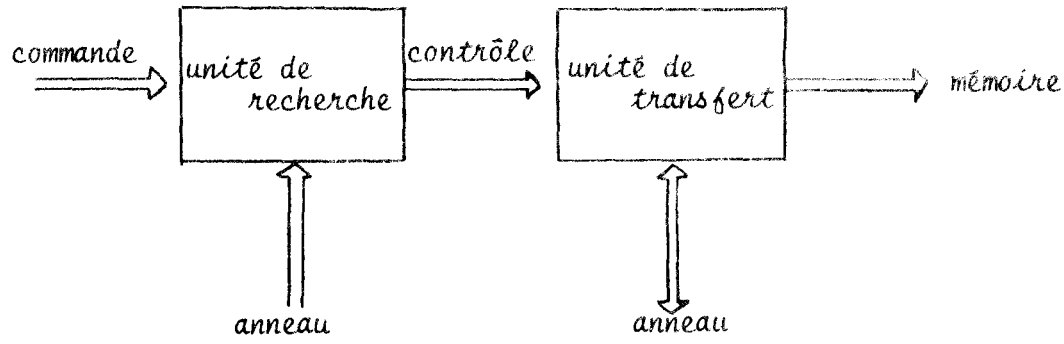


Fig 8 : Synoptique de l'unité d'échange

L'unité de transfert ne devient active que si un mot doit être transféré de l'anneau vers la mémoire locale ou vice versa. L'unité de recherche pourra donc en assurer le contrôle puisque c'est elle qui détecte sa présence au niveau de la fenêtre.

3. *L'unité de transfert*

3.1. *Etude fonctionnelle*

La fonction de l'unité de transfert doit se limiter à la gestion des voies de communication entre la mémoire circulante et la mémoire locale du processeur, c'est à dire :

- Les voies de lecture et d'écriture de la fenêtre d'accés et le signal de sélection des voies.

- Les bus de données et d'adresses de la mémoire locale.

Au niveau du fonctionnement général de l'unité de transfert, deux possibilités sont envisageables :

- Lorsque le début d'un secteur est détecté, l'unité de recherche communique à l'unité de transfert, l'ensemble des paramètres nécessaires à la gestion de la suite des opérations. Cette dernière peut alors continuer de façon autonome. Pour cette solution, deux difficultés apparaissent au

niveau de la définition des paramètres. Au cours d'un échange, le sens du transfert n'est pas obligatoirement le même pour les mots consécutifs d'un secteur pouvant entrainer par conséquent un volume de communication important entre les deux unités. D'autre part, lors de l'initialisation, il n'est pas toujours possible de connaître la longueur du secteur. Celle-ci doit donc être déterminée par l'unité de transfert et cette fonction est plus du ressort de l'unité de recherche.

- La deuxième solution consiste à retirer tout pouvoir décisionnel à l'unité de transfert. Celle-ci opère alors, au niveau du mot et ne joue qu'un rôle d'exécutant vis à vis de l'unité de recherche. Lorsqu'un échange doit avoir lieu sur le mot présent au niveau de la fenêtre d'accés, elle reçoit de l'unité de recherche deux paramètres qui sont respectivement un code d'opération (lecture, écriture, mise à jour) et l'adresse d'une position en mémoire locale .

D'un point de vue général, il est toujours plus simple lorsque deux unités doivent se partager une tâche de les situer l'une par rapport à l'autre dans une relation hiérarchique fixe (l'une maitre, l'autre esclave) plutôt que de les placer au même niveau en leur conférant une autorité réciproque l'une sur l'autre.

3.2. Réalisation pratique

L'élément central de l'unité de transfert est un automate dont le but est de générer l'ensemble des signaux de contrôle. Celui-ci est activé par l'unité de recherche lorsque le mot présent au niveau de la fenêtre d'accés doit être transféré dans la mémoire locale ou vice-versa. A partir du timing de la mémoire circulante il est facile de voir comment vont se dérouler les opérations.

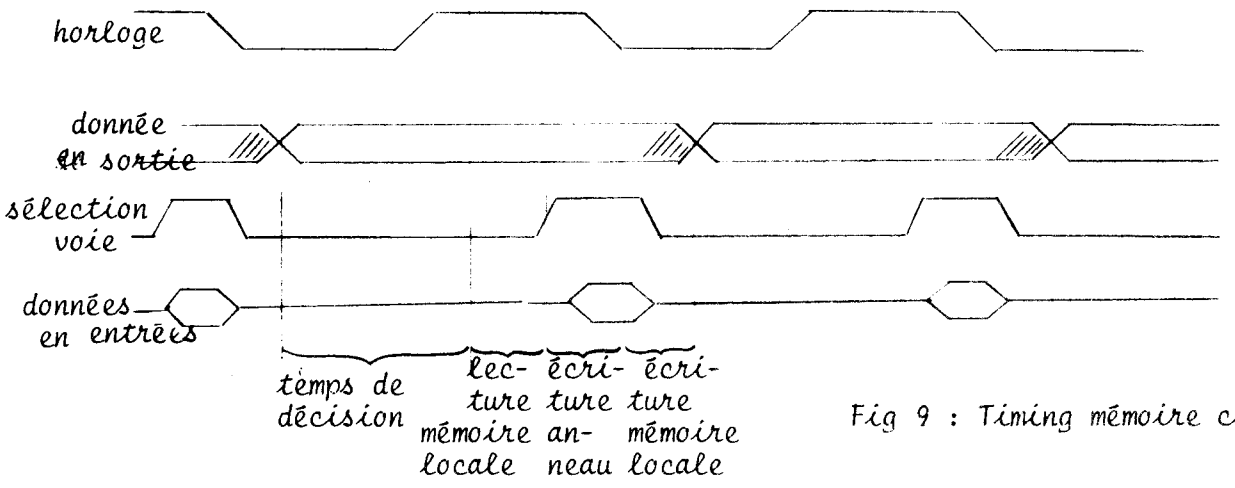


Fig 9 : Timing mémoire circulante

L'information apparait en sortie de cellule, 300 ns après le front descendant de l'horloge. L'unité de recherche ne pourra donc commencer son observation qu'à cet instant.

Pour écrire un mot dans la mémoire circulante, la donnée doit être présente 50 ns avant le front descendant de l'horloge. Si à cela on rajoute le temps d'accés à la mémoire locale il reste environ 400 ns pour décider de la capture.

Dans le cas d'une opération de mise à jour, une opération de lecture et une opération d'écriture sont nécessaires au cours du même cycle d'horloge. De plus il est indispensable de stocker momentanément l'une des deux informations qui seront échangées. L'information qui se présente au niveau de la fenêtre est sauvegardée dans un registre temporaire. L'écriture en mémoire locale peut donc se faire après le front descendant de l'horloge.

La réalisation matérielle de l'automate peut être envisagée de deux façons : *automate cablé* ou *automate microprogrammé*. Notre choix s'est porté sur la deuxième solution car elle permet de supporter d'éventuelles modifications ou l'adjonction de nouvelles opérations.

Le choix de la capacité de la mémoire locale associée au processeur sera fait en fonction de ses besoins propres. Un découpage en bloc de cet espace (réseau cross-bar) semble souhaitable afin d'autoriser des accés simultanés de la part du processeur et de l'unité d'échange sur des blocs disjoints.

Le contrôle de l'unité de transfert est assuré par l'unité de recherche qui lui fournit,

- Un *code opération* (2 bits) qui indique le type de transfert qui doit avoir lieu. Une extension à 3 bits peut être envisagée.

- Une *adresse en mémoire locale* (16 bits) permettant l'adressage d'une mémoire de 64 K mots.

- Un *signal de validation* qui autorise l'unité de transfert à commencer son cycle en se synchronisant sur l'horloge de la mémoire circulante.

4. L'unité de recherche

L'unité de recherche a pour fonctions :

- de *déterminer les secteurs de l'anneau* qui correspondent à la demande fournie par le processeur.

- d'*assurer le contrôle de l'unité de transfert.*

4.1. Analyse fonctionnelle

L'ensemble des objets utilisés par Maud est rangé dans les cadres physiques de taille fixe. Un échange entre la mémoire circulante et la mémoire locale ne peut donc porter que sur un ou plusieurs secteurs contenus dans un cadre. Pour cette raison, toutes les opérations de recherche peuvent se décomposer en deux temps.

- *Recherche d'un cadre.* Le ou les secteurs demandés sont inclus dans un cadre qui est caractérisé par son entête. Toute opération de recherche commence donc par l'observation des entêtes de cadre qui se présentent successivement au niveau de la fenêtre d'accés et la comparaison avec le type de cadre requis par l'échange.

- *Recherche à l'intérieur d'un cadre.* Lorsqu'un cadre demandé se présente, la deuxième phase de la recherche consiste à déterminer l'ensemble des positions mémoires pour lesquelles un transfert doit avoir lieu. Pour chacune d'elles, un mot de commande est transmis à l'unité de transfert qui se charge de la suite des opérations comme nous l'avons vu au § 3.

Pour assurer le bon déroulement de ces phases, le processeur doit fournir un ensemble de paramètres caractérisant l'information cherchée.

4.2. Paramètres de recherche d'un cadre

L'entête du cadre contient un ensemble d'informations qui permettent de caractériser sans ambiguité le contenu.

TYPE	NUMERO
NOM DU BLOC	NOMBRE DE CADRES

Fig 10 : Entête d'un cadre

Le *type* définit le bloc contenu dans le cadre : Bloc exécutable bloc en attente, demande d'exécution, domaine de sortie, cadre vide. Bien que le *numéro* du cadre n'a pas sa nécessité au niveau de Maud il sera utile pour des fonctions d'initialisation ou pour établir une trace du traitement. A certains cadres est associé un *nom de bloc*. C'est le cas en particulier des blocs exécutables et des blocs en attente. Dans l'hypothèse où un bloc ne peut être stocké dans un seul cadre, on précise le *nombre de cadres* qui sont utilisés.

Plusieurs critères peuvent être définis pour la sélection d'un cadre :

- Le cadre peut être spécifié par son *type*. C'est ce critère qu'utiliseront les processeurs d'exécution pour demander un bloc exécutable.

- La sélection du cadre peut se faire également par le nom du bloc qui est stocké. Ce mode est particulièrement utile si un bloc exécutable peut occuper plusieurs cadres. La recherche du premier cadre se fait par le type bloc exécutable. Lors de la lecture de celui-ci, l'unité relève le nom du bloc et effectue la recherche des autres cadres par le nom. Elle détermine l'instant où l'ensemble du bloc a été lu en ayant relevé le nombre de cadres qui avaient été utilisés pour stocker le bloc complet.

- Pour certaines applications particulières, il peut être intéressant de sélectionner un cadre par son numéro. Le caractère invariant de ce paramètre permet de sélectionner un cadre sans en connaitre à priori le contenu .

Les trois paramètres que nous venons de citer permettent la sélection des cadres pour toutes les procédures qui ont été présentées dans la première partie de ce chapitre. Toutefois une extension probable du nombre de ces paramètres doit pouvoir être envisagée en cas de modification de l'entête.

4.3. *Paramètres de recherche à l'intérieur d'un cadre*

Les paramètres de recherche à l'intérieur d'un cadre doivent permettre de délimiter l'ensemble des secteurs pour lesquels un échange doit

avoir lieu. Du choix de ces paramètres va dépendre la gamme de services que pourra remplir l'unité d'échange. A partir des procédures d'échanges nous pouvons envisager plusieurs modes de recherche de début et de fin de secteur.

- *Recherche par nom.* La détermination du secteur se fait par comparaison entre le contenu de la mémoire circulante et un mot clé fourni par le processeur (recherche associative). C'est ainsi que se fait la transmission des valeurs par le processeur mise à jour. La recherche des valeurs à communiquer à un domaine d'entrée se fait par l'intermédiaire du nom de communication.

- Dans un grand nombre de cas, le transfert porte sur l'ensemble du bloc contenu dans le cadre. Le paramètre nécessaire est alors la position du premier mot. C'est ce qu'on appellera la *recherche par position.* De la même manière, il est possible de délimiter un ou plusieurs secteurs en fournissant successivement pour chacun, les adresses du début et de fin. Ce mode de recherche combiné avec la recherche du cadre par numéro revient à associer à chaque mot dans l'anneau, une adresse.

- *Synchronisation sur signal externe.* Ce mode de recherche peut être utilisé lorsque l'organisation des informations à l'intérieur d'un cadre est parfaitement connue. Les secteurs peuvent être délimités par des signaux externes tels que l'horloge de contrôle de circulation dans l'anneau. Un exemple d'application pourrait être : "lire trois mots toutes les quatre positions mémoires".

- La longueur d'un secteur peut être caractérisée par le nombre de mots à transférer. La détection de début de secteur, se fait dans ce cas là, à l'aide de l'un des trois modes précédents.

4.4. Durée de vie d'une demande

L'ensemble des paramètres que nous venons de citer (§ 4.3) permettent de délimiter un ou plusieurs secteurs à l'intérieur d'un cadre. Il reste à caractériser la durée de vie d'une demande de transfert fournie par le processeur.

Le processeur mise à jour et le processeur constructeur effectuent des échanges à *durée indéfinie*. Le premier capture tous les domaines de sortie ou tous les blocs en attente, le deuxième toutes les demandes d'exécution. L'unité d'échange se trouve donc toujours dans un état "actif". Afin de gérer convenablement l'ensemble des informations qui lui sont fournies, le processeur doit être averti après chaque opération de transfert sur un cadre. D'autre part celui-ci doit pouvoir reprendre le contrôle afin d'inhiber la demande qui est en cours ou de la modifier, en particulier pour éviter toute saturation de la mémoire locale si le processeur ne peut consommer en temps réel l'ensemble des objets qui lui sont fournis.

La durée de vie de la demande peut être spécifiée par un *nombre* de cadres à transférer comme nous l'avons vu au § 4.2. Bien qu'elle ne soit pas directement nécessaire au niveau de Maud, la possibilité de quantifier la durée en nombre de tours d'anneau ou en nombre de cadres qui passent devant la fenêtre d'accés, semble intéressante. Elle permet d'assurer de manière aisée une observation du contenu de l'anneau pendant un temps donné. Si au cours du temps qui lui est imparti il n'y a pas de cadres correspondant aux spécifications de recherche, aucun transfert n'aura lieu pendant le déroulement de cette demande mais un signal de fin d'activité sera néanmoins délivré au processeur.

4.5. *Réalisation pratique*

La nécessité de disposer d'une grande diversité de fonctions et de pouvoir adapter le rôle de chaque unité de recherche selon le processeur auquel elle est associée, entraine tout naturellement l'étude d'un dispositif programmé ou microprogrammé. La définition d'un système entièrement cablé n'autorise aucune modification du fonctionnement et donc aucune adaptation.

Les contraintes de temps quant à la durée de la recherche sont très importantes. Le temps de recherche et de prise de décision au niveau du mot ne doivent pas exéder 400 ns. Cette exigence interdit immédiatement l'utilisation d'un microprocesseur standard et conduit elle aussi à l'étude d'un système microprogrammé.

L'ensemble des modes de recherche qui peuvent être envisagés ne doivent pas être implantés dans chaque unité de recherche. Il semble donc plus intéressant de définir une structure modulaire, chacun des modules étant indépendant.

L'unité de recherche comporte trois modules.

- Le module de *recherche associative.*
- Le module de *comptage de cadres.*
- Le module de *comptage de mots.*

L'ensemble de ces modules est piloté par un automate microprogrammable.

4.6. *Le module de recherche associative*

A partir des critères de sélection d'un cadre ou d'un secteur dans un cadre, on peut constater que la décision de capture ou non,apparait comme le résultat d'une comparaison entre une information dont dispose l'unité de recherche et fournie par le processeur et une information contenue dans la mémoire circulante ou spécifique à l'état d'avancement de l'anneau. Quelques exemples permettent de vérifier cette propriété :

- La recherche d'un cadre par type ou par nom s'effectue par comparaison entre le contenu de l'entête d'un cadre et du paramètre type ou nom fourni par le processeur lors de la demande.

- La transmission d'une valeur par le processeur mise à jour se fait lorsque le nom de communication contenu dans le domaine d'entrée du bloc en attente est identique à l'un des noms de communication des couples (nom, valeur) qui sont apparus dans un domaine de sortie et qui ont été pris par le processeur dans sa mémoire locale.

- Lors de la détermination des limites d'un secteur par adresse il doit y avoir équivalence entre les positions fournies par le processeur et l'état d'avancement de l'anneau.

L'idée qui a été développée était de réaliser un dispositif de comparaison général entre une information issue de l'anneau de mémoire circulante (contenu ou information de contrôle) et une information fournie par le processeur. Il faut remarquer que cet outil doit être capable d'opérer sur des mots (nom de communication) sur des octets (nom de cadre, type) ou sur des adresses dans un cadre (10 bits).

La difficulté essentielle pour l'implémentation réside dans le nombre variable des opérandes nécessaires pour effectuer la comparaison. L'information issue de l'anneau est unique. On ne lit qu'un seul mot à la fois. Par contre les paramètres fournis par le processeur peuvent être multiples. Le cas le plus typique est celui des noms de communication. La comparaison doit être effectuée entre le nom qui se trouve dans le domaine d'entrée et l'ensemble des noms qui apparaissent dans la mémoire locale. Il peut en être de même avec le type d'un cadre. Un processeur peut souhaiter accéder à plusieurs types de cadre (un processeur d'exécution peut déposer un domaine de sortie dans un cadre vide ou à la place d'un bloc exécutable).

Le temps qui peut être alloué pour obtenir le résultat de la comparaison est constant quel que soit le nombre de paramètres. L'utilisation d'un comparateur classique est donc impossible.

L'utilisation d'une mémoire permet de résoudre facilement cette difficulté. Le principe en est le suivant. On associe à chaque nom de bloc, à chaque type, à chaque nom de communication et à chaque adresse dans le cadre une position mémoire de 1 bit que le processeur place à 1 si l'élément ainsi désigné doit faire l'objet d'un transfert. En phase de recherche, cette mémoire est adressée avec l'information issue de l'anneau ce qui permet de savoir immédiatement s'il s'agit de l'information demandée. Le temps de comparaison se réduit alors au temps d'accés de la mémoire.

Au prix d'une complexité accrue, cette solution apparait comme la seule capable de délivrer simultanément le résultat de la comparaison d'un élément de l'anneau avec un ensemble d'éléments proposés comme paramètre par le processeur. Elle se révèle équivalente à 2^n comparaisons parallèles pour des mots de format n.

Pour effectuer le transfert des valeurs des domaines de sortie vers les domaines d'entrée, il faut utiliser une mémoire de 64 K x 1 bit. Le processeur mise à jour positionne à 1 tous les emplacements mémoires correspondant aux noms de communication lus dans un domaine de sortie. Lors du passage d'un domaine d'entrée, l'unité de recherche adresse cette mémoire avec le mot lu dans la mémoire circulante et demande un transfert si le résultat est égal à 1.

Pour le choix d'un cadre par type ou par nom, les opérations se déroulent de manière identique en positionnant à 1 les emplacements mémoires correspondant au type ou au nom.

De la même manière, la délimitation d'un secteur à l'intérieur d'un cadre se fait en positionnant à 1 les emplacements mémoires correspondant aux adresses de début et de fin du secteur.

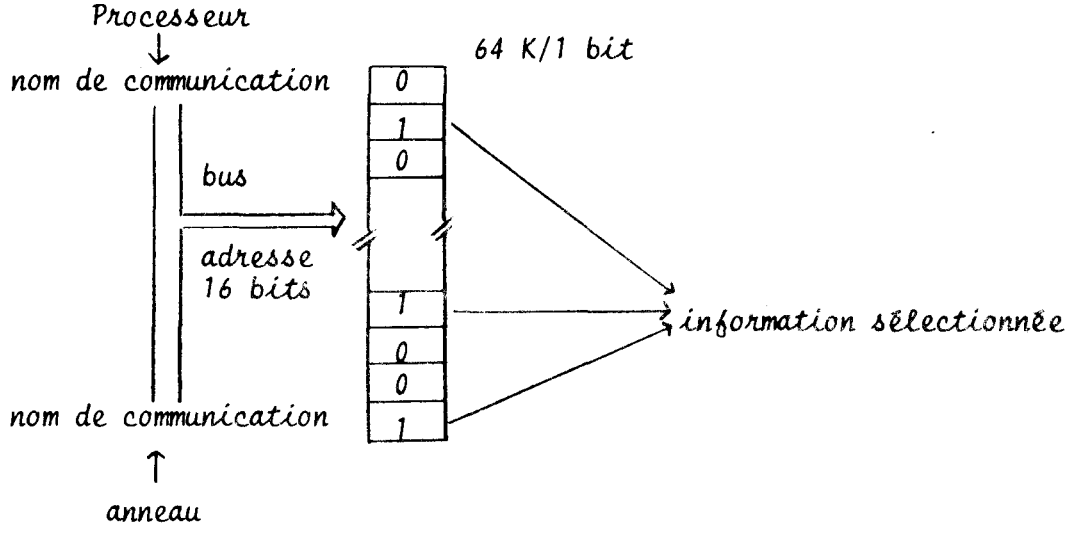


Fig 11 : Mémoire de comparaison pour nom de communication

L'unité d'échange doit être capable d'assurer dans certains cas la mise à jour du contenu de cette mémoire de comparaison. Par exemple, lorsque l'unité de recherche a trouvé un bloc exécutable pour un processeur d'exécution elle remet à 0 le bit correspondant au type exécutable dans la mémoire de comparaison. Si le bloc est rangé sur plusieurs cadres l'accés pour la suite du bloc se fait par le nom du cadre. Pour cela, l'unité de recherche doit avoir relevé le nom du bloc et positionné à 1 le bit correspondant à ce nom dans la mémoire de comparaison.

4.7. Le module de comptage de mots et de comptage de cadres

Ils sont nécessaires pour détecter la fin d'un secteur lorsque la longueur de celui-ci est donnéeen nombre de mots ou la fin d'une demande de transfert. Il s'agit en fait de compteurs qui peuvent être initialisés par une valeur fournie par le processeur avant la demande ou par l'unité d'échange en cours de transfert

4.8. L'unité de contrôle microprogrammée

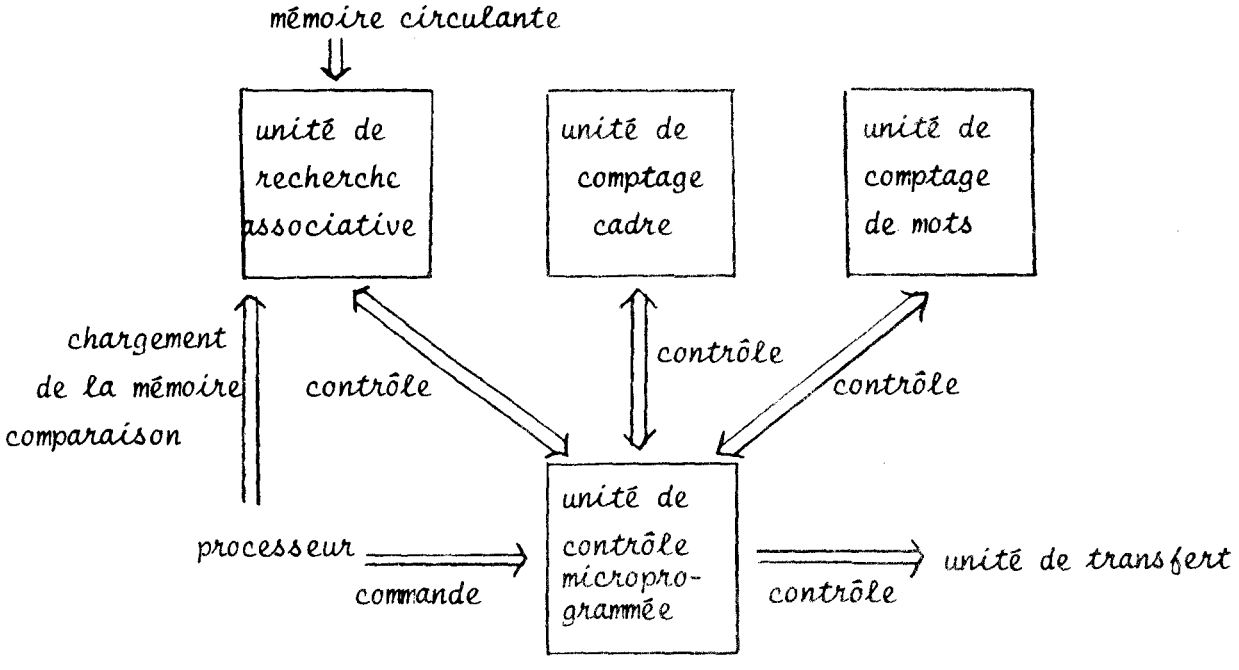


Fig 12 : Structure de l'unité de recherche

La fonction de cette unité est de générer l'ensemble des commandes nécessaires pour assurer le contrôle des trois modules constituant l'unité de recherche et de l'unité de transfert. Les opérations que l'unité d'échange peut effectuer sont entièrement définies par le microprogramme.

Comme pour l'unité de transfert, il s'agit d'un générateur de signaux qui contrôle un ensemble de dispositifs logiques. Il peut donc être réalisé à l'aide d'une PROM rebouclée sur elle-même, chaque mot du microprogramme contient en plus des diverses commandes l'adresse du mot suivant. Toutefois, le microprogramme ne se déroule pas de manière purement

séquentielle. Les commandes à générer dépendent d'évênements extérieurs (résultats de comparaison) et doivent être synchronisées avec l'horloge de l'anneau. Il est donc nécessaire de pouvoir effectuer des points de branchement à l'intérieur du microprogramme. Chaque mot du microprogramme contient deux adresses. L'une ou l'autre sera sélectionnée pour donner l'adresse du mot suivant en fonction d'évênements extérieurs.

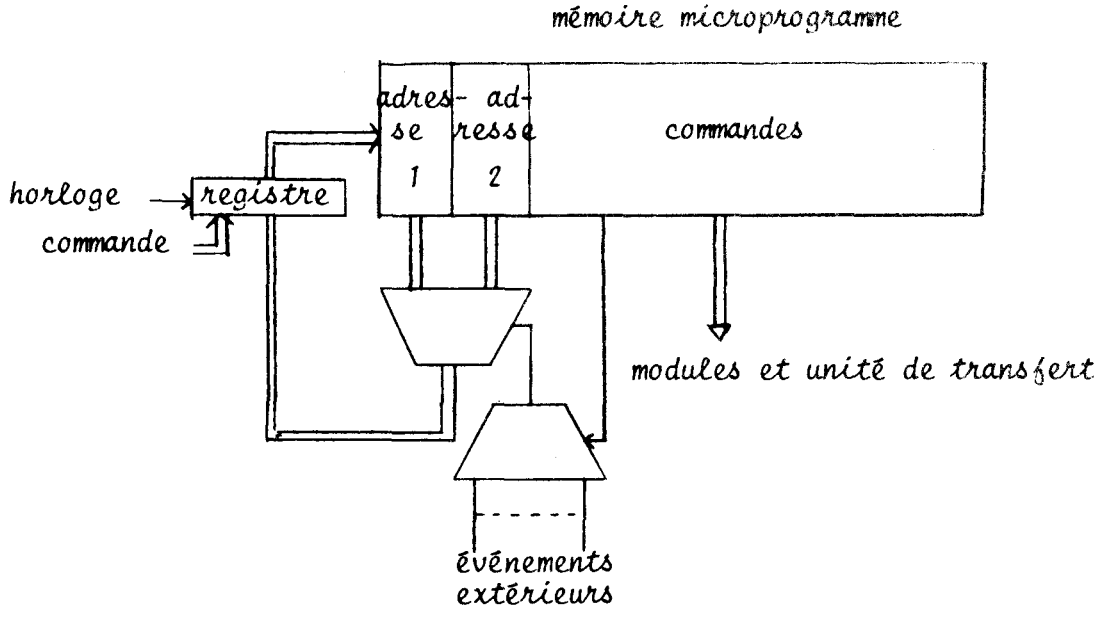


Fig 13 : Unité de contrôle microprogrammée

4.9. Communications processeur-unité de recherche

Le processeur garde à tout instant le contrôle de l'unité d'échange. Lorsqu'un transfert doit avoir lieu, il doit fournir à l'unité de recherche, l'ensemble des paramètres concernant la demande. Le fait d'utiliser une mémoire de comparaison permet de réduire considérablement le volume des communications nécessaires. Avant d'adresser sa demande le processeur configure la mémoire de comparaison en positionnant à 1 certains emplacements mémoires.

Pour initialiser le transfert il reste au processeur à fournir un couple (Co, ADMC).

- Un *code opération* qui permettra la sélection du microprogramme correspondant à l'échange désiré. Chaque unité d'échange pourra donc assurer un nombre limité d'échanges suivant la fonction du processeur auquel elle sera associée.

- Une *adresse en mémoire locale* qui indiquera le premier emplacement de la mémoire pour lequel un transfert doit avoir lieu. Le processeur alloue ainsi à l'unité d'échange une zone mémoire à laquelle il ne pourra plus accéder tant que l'échange n'est pas terminé.

Bien que le processeur garde l'initiative des opérations, il ne peut interrompre un échange à tout instant. Tout transfert commencé sur un secteur doit se poursuivre jusqu'à la fin du secteur afin de garantir la cohérence de l'information. Une interruption ne pourra donc être prise en compte que si l'unité d'échange se trouve dans un état de recherche de secteur. Celle-ci aura pour effet de ramener l'unité dans un état repos. Le processeur pourra alors modifier ou déposer une nouvelle demande lorsque l'unité sera dans un état repos.

5) *Exemple d'utilisation : Le processeur pupitre*

Le processeur dont il va être question dans la suite du paragraphe n'apparait pas dans la définiton de MAUD. La nécessité d'un outil d'observation de l'anneau s'est très vite fait sentir. Dans une phase de mise au point, il est nécessaire de disposer d'un outil capable de vérifier le fonctionnement de la mémoire circulante et le déroulement des échanges entre les processeurs et l'anneau. Dans une phase ultérieure, la fonction d'un tel processeur pourrait se résumer à l'observation permanente de l'anneau afin d'élaborer des renseignements sur l'état du système ou bien d'assurer certaines fonctions particulières telles que le chargement de bloc ou des opérations d'initialisation.

5.1. *Architecture du processeur*

La fonction de ce processeur se limite à l'initialisation de transferts entre sa mémoire locale et la mémoire circulante en vue de fournir un état du système.

Celui-ci a été réalisé autour d'un microprocesseur 8085 qui dispose de :

- 2 K octets de PROM qui contiennent le programme de fonctionnement du processeur.

- 4 K octets de RAM dont 2 K sont accessibles par l'unité d'échange en mutuelle exclusion avec le microprocesseur. Il est donc possible de transférer le contenu d'un cadre dans la mémoire locale.

Le microprocesseur contrôle en outre :

- Un clavier hexadécimal auquel ont été rajoutées quelques touches fonctions permettant à un opérateur de spécifier les opérations que doit exécuter le processeur et de modifier le contenu de la mémoire locale.

- Un écran qui permet de visualiser l'ensemble des renseignements sur l'état du système.

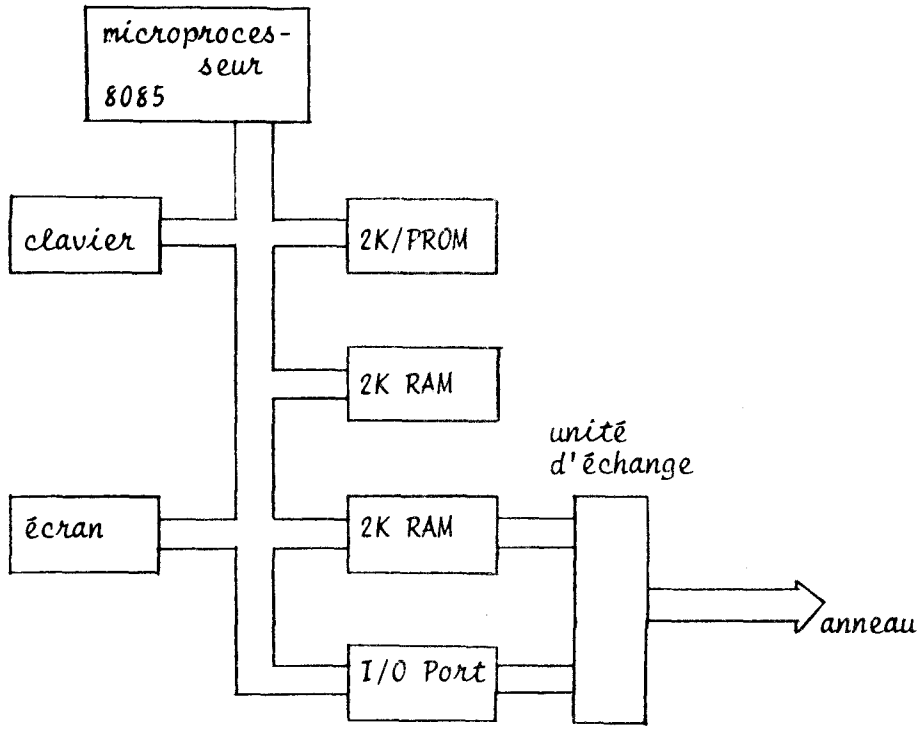


Fig 14 : Architecture du processeur pupitre

5.2. *Fonctions du processeur pupitre*

L'ensemble des opérations que le processeur pupitre peut être amené à exécuter se divise en trois sous-ensembles :

- La *fonction d'échange* avec l'anneau qui permet d'assurer l'observation et le chargement de la mémoire circulante.

- La *fonction de manipulation*. Toutes les modifications qui peuvent être entreprises sur le contenu d'un cadre sont effectuées au niveau de la mémoire locale.

- La *fonction d'initialisation* qui est nécessaire lors de la mise en route du système.

A chacune de ces fonctions correspondent des commandes qui peuvent être utilisées par l'opérateur au niveau du clavier selon ses besoins.

La fonction d'observation peut se faire à deux niveaux.

- A un niveau global. Le processeur exploite alors les informations contenues dans les entêtes des cadres et fournit sur l'écran un état du système évalué sur un tour de la mémoire circulante. La commande *DISPLAY ALL* provoque la lecture des entêtes de tous les cadres.

- A un niveau plus fin. La commande *DISPLAY* permet de lire le contenu d'un cadre, celui-ci étant caractérisé par son type, son numéro ou le nom du bloc qui s'y trouve. Une partie du contenu peut alors être visualisée sur l'écran.

La fonction de chargement consiste à pouvoir transférer le contenu de la mémoire locale dans l'anneau. C'est le rôle de la commande *INSERT*. Ici encore il faut spécifier le cadre.

La fonction de manipulation doit permettre à l'opération de charger la mémoire locale ou de modifier son contenu. Ceci est réalisé par la commande *ENTER* en spécifiant l'adresse en mémoire locale. La combinaison

des fonctions *DISPLAY ALL, DISPLAY, INSERT, ENTER* permettent d'assurer une grande variété de fonctions d'accés. Lorsque le processeur reçoit l'une de ces commandes, il élabore la commande pour l'unité d'échange en utilisant les paramètres de sélection les mieux adaptés.

La fonction d'initialisation consiste à configurer l'anneau dans un certain état lors de la mise en route du système. En particulier il est nécessaire de numéroter les cadres de la mémoire circulante. C'est le rôle de la fonction *INITIAL*, à laquelle correspond une fonction d'accés particulière au niveau de l'unité d'échange.

Dans une phase ultérieure, l'implémentation de nouvelles commandes pourra être envisagée afin de répondre à des besoins particuliers fonctionnement ou d'observation.

6. *Conclusion*

Bien que la définition de l'unité d'échange se soit placée dans le contexte MAUD, nous nous sommes efforcés de proposer un outil de portée relativement générale.Son application pourrait être aisément envisagée dans le cadre d'une autre utilisation de l'anneau. La définition d'une nouvelle application ne nécessite en effet qu'une modification du microprogramme.

CHAPITRE V

SIMULATION

Dans ce chapitre nous présentons les travaux de simulation que nous avons entrepris ainsi que les premiers résultats obtenus et les conclusions partielles que l'on peut tirer. La première partie présente d'une façon très générale la simulation et les objectifs que nous avions en effectuant ce travail. Dans un deuxième temps nous nous attacherons à étudier les deux produits que nous avons définis. Le premier simule le modèle statique [LEC 79c] que nous avons abordé très rapidement dans le deuxième chapitre, le deuxième, par l'introduction des primitives *execbloc* et *attendre* permet de simuler le modèle dynamique.

1. *Introduction*

L'idée première que nous avons développée, était de simuler le modèle en tenant compte de l'architecture proposée. Il nous fallait par conséquent simuler le fonctionnement de l'anneau et des différents processeurs tout en se définissant des paramètres qui permettent d'évaluer les performances que nous pouvons attendre du système.

Au niveau des processeurs d'exécution ce n'est pas l'exécution proprement dite du bloc qui nous intéresse mais au contraire l'influence du temps d'exécution. Un bloc peut donc être décrit par un temps d'exécution et une suite de primitives *execbloc* et *attendre*. Il est bien évident que pour garder un caractère réaliste à la simulation, l'évaluation des temps d'exécution doit tenir compte des performances des processeurs qui seront utilisés pour la réalisation matérielle. L'aspect description du bloc sera étudié plus en détail dans la suite de ce chapitre.

La conséquence du point précédent est que au niveau de l'anneau ce n'est pas la fonction de stockage mais la fonction de communication qu'il faut simuler. L'anneau apparait uniquement comme étant composé d'un certain nombre de cases, la variation de ce nombre permettant de modifier le temps de circulation.

Dans la simulation nous avons également tenu compte des possibilités introduites par l'existence des unités d'échange en particulier de la simultanéité des transferts et des fonctions propres des processeurs. Ce dernier ne se bloque pas systèmatiquement s'il ne peut pas déposer quelque chose dans l'anneau.

L'ensemble des temps que nous avons définis pour la simulation ont été quantifiés par rapport à un temps de base qui est égal au temps de circulation d'un cadre de l'anneau devant une fenêtre. Ainsi le temps d'exécution d'un bloc est évalué par rapport à un temps de circulation dans l'anneau.

Les objectifs de la simulation étaient multiples. Nous pouvons citer en particulier.

- Evaluation des performances du système pour différentes structures de graphe de programme. Ceci fournira des renseignements quant aux méthodes à mettre en oeuvre pour effectuer le découpage en blocs d'un programme et plus généralement sur l'organisation même des programmes.

- Influence de certains paramètres sur le comportement du système. Les paramètres que nous avons retenus sont les suivants :
  . nombre de processeurs autour de l'anneau
  . temps d'exécution d'un bloc
  . nombre de cadres dans l'anneau

- Aide pour le choix de certaines solutions et plus particulièrement détermination des objets qui doivent rester dans l'anneau en cours de traitement (bloc en attente ou domaine de sortie) [LEC 79c].

- Détermination des conditions de blocage du système et de saturation de la mémoire circulante ainsi que des dispositifs matériels et logiciels à implémenter pour éviter ces cas de figure.

La simulation est effectuée sur un IRIS 80 et les programmes ont été écrits en PASCAL. Le choix de ce langage a été fait pour les raisons suivantes :

- Le nombre et la longueur des objets sur lesquels nous sommes amenés à opérer est très variable. Ceux-ci sont gérés sous forme de liste que nous pouvons manipuler aisément avec PASCAL.

- La possibilité de définir des types d'objets complexes apparait très importante pour la description du système.

2. *Simulation du modèle statique*

2.1. *Représentation du système*

Dans le modèle statique, nous ne trouvons pas de primitives *execbloc* et *attendre*. Les seuls objets que nous avons par conséquent à décrire sont la mémoire circulante, les processeurs d'exécution et le processeur mise à jour. La structure de donnée utilisée est donnée figure 1.

Les processeurs d'exécution sont représentés sous la forme d'un tableau. Le nombre des processeurs qui sont connectés à la mémoire circulante est l'un des paramètres de la simulation. Chacun d'entre eux est décrit par quatre paramètres :

- La *fenêtre* précise l'emplacement du processeur et permet de déterminer le cadre auquel il peut accéder.

- Le processeur d'exécution peut se trouver dans quatre états. Il est *actif* s'il est en train d'exécuter un bloc exécutable. Au contraire il est *inactif* s'il recherche un bloc exécutable. L'état *attente* correspond à un processeur qui souhaite déposer un objet dans l'anneau. L'état *panne* permet de déconnecter un processeur de l'anneau.

- La grandeur *Bloc traité* est un pointeur qui indique l'emplacement mémoire où se trouve la description du bloc sur lequel le processeur opère.

- Au cours de l'exécution d'un programme, le processeur prendra successivement les états précédemment définis à l'exeception de l'état panne. En vue de réaliser une analyse statistique, nous évaluons les temps passés dans chacun de ces états,ceux-ci étant rangés dans le tableau *travail*.

L'anneau est lui aussi représenté sous la forme d'un tableau. Chacun des éléments caratérise l'un des cadres.

- *Etat ancien* et *nom bloc* sont nécessaires pour sauvegarder l'état du cadre avant toute opération.

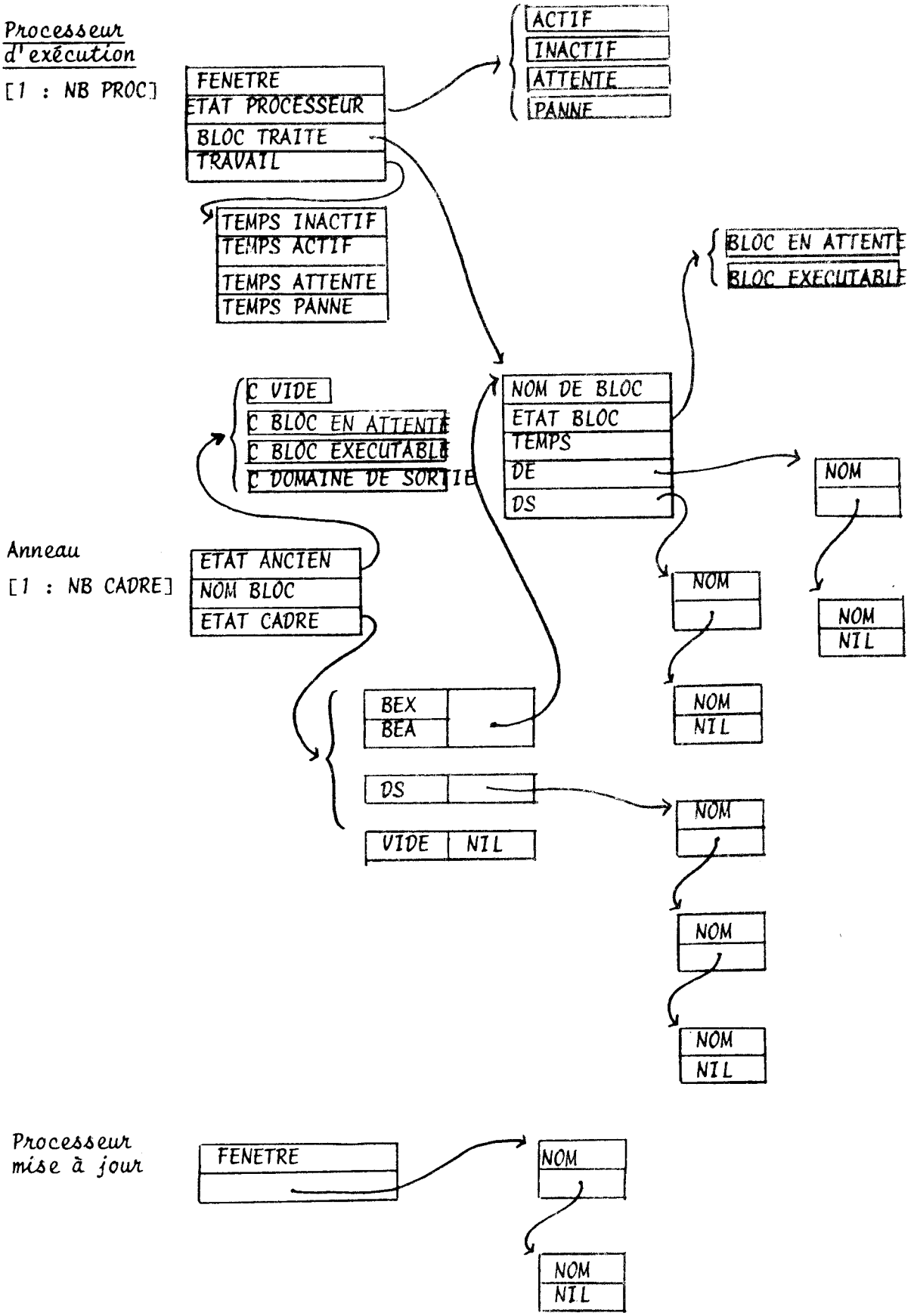


Fig 1 : Représentation du modèle statique

- *Etat cadre*, quant à lui, précise le contenu du cadre. S'il s'agit d'un bloc exécutable ou d'un bloc en attente, on pointe sur la description de ce bloc, s'il s'agit d'un domaine de sortie on pointe sur une liste de couples (nom, pointeur) qui décrivent le domaine de sortie.

Tout comme les processeurs d'exécution, le processeur mise à jour est décrit par sa *fenêtre* qui précise son emplacement autour de l'anneau. Un pointeur pointe sur une liste de couples (nom, pointeur) qui représente l'ensemble des couples (nom, valeur) qui se trouve dans la mémoire locale de ce processeur. Pour la simulation nous avons adopté provisoirement l'hypothèse que les blocs en attente restaient dans l'anneau et que les domaines de sortie étaient pris par le processeur mise à jour.

Puisque les processeurs n'exécutent pas réellement les blocs pour la simulation, il nous suffit de considérer qu'un processeur d'exécution, après avoir pris en charge un bloc exécutable, sera actif pendant un *temps* correspondant à la durée d'exécution de ce bloc. Pour décrire complétement ce dernier il faut rajouter à cette notion de temps le nom du bloc, son état (exécutable ou en attente) et deux pointeurs qui indiquent où sont représentés le domaine d'entrée et le domaine de sortie. Il s'agit en fait de deux listes de couples (nom, pointeur).

2.2. *Configuration du système*

Le modèle ainsi décrit ne permet pas l'exécution d'un programme. Il faut y rajouter un processeur supplémentaire appelé *"Injecteur"* dont la fonction sera de délivrer au système les différents blocs qui constituent le ou les programmes à exécuter. Avant de lancer l'exécution, l'injecteur dispose d'une liste de blocs exécutables ou en attente qu'il introduira dans l'anneau les uns à la suite des autres chaque fois que devant sa fenêtre d'accés se présente un cadre vide. Il est indispensable d'avoir au moins un bloc exécutable. Le domaine de sortie qu'il produira permettra d'assurer la mise à jour de nouveaux blocs en attente.

Comme nous l'avons signalé dans le chapitre III, la position relative des différents processeurs ne doit pas être quelconque. Le choix des emplacements peut avoir une influence sur le comportement du système,

Il semble souhaitable de garder la structure pipe-line qui apparait au niveau de la définition du modèle. Ceci permet dans une certaine mesure de réduire les retards de communication entre les sites dus à la propagation dans l'anneau. La configuration que nous avons simulé est schématisée figure 2.

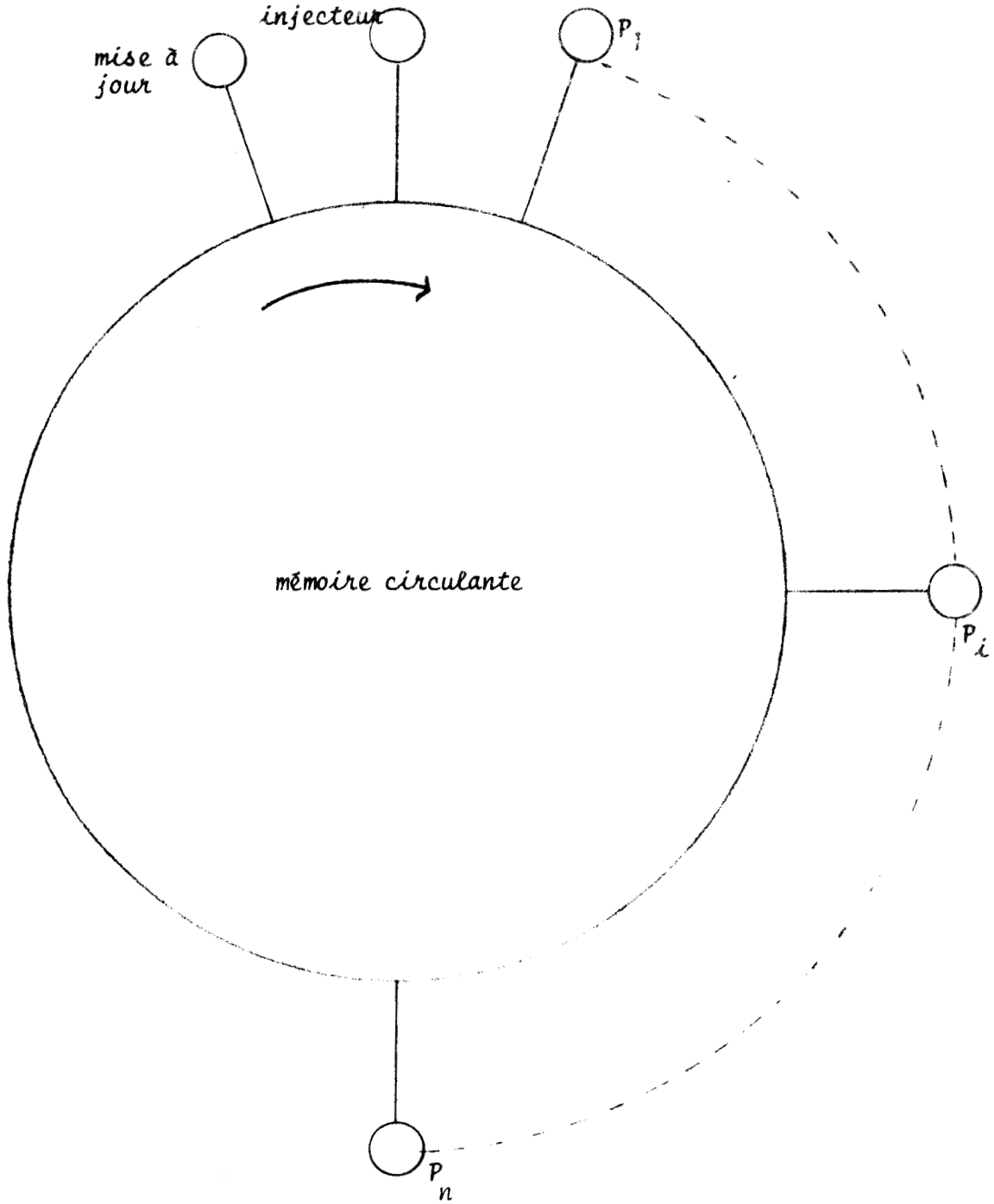


Fig 2 : Architecture de Maud pour la simulation du modèle statique

Parmi les paramètres sur lesquels il est possible d'influer il y a le nombre de cadres présents dans la mémoire circulante. Lors de l'adjonction d'un cadre et dans le but de ne pas modifier la position relative des processeurs entre eux, celui-ci est introduit entre le dernier processeur d'exécution et le processeur mise à jour. On pourrait envisager de placer le processeur mise à jour juste après les processeurs d'exécution et de rajouter alors des cadres avant l'injecteur.

Le mode de fonctionnement du modèle statique impose des contraintes quant au choix du nombre de cadres pour l'anneau. L'ensemble des blocs qui constituent le programme doit être parfaitement défini avant l'exécution. Les blocs en attente sont placés dans l'anneau tout comme les blocs exécutables. Pour que le programme puisse s'exécuter, il faut laisser au moins un cadre vide dans l'anneau pour qu'un processeur puisse y déposer un domaine de sortie. Ceci équivaut à définir un nombre de cadres minimal qui ne dépend que du nombre de blocs du programme.

> On peut facilement montrer que
>
> $$Nc\ minimum = N_{BEA} + 1 = N_B - N_{BEX} + 1$$
>
> ou Nc représente le nombre de cadres, N_B le nombre total de blocs N_{BEX} le nombre de blocs exécutables et N_{BEA}, le nombre de blocs en attente

Cette contrainte impose donc un temps minimum de circulation. Celle-ci n'existe plus dans le modèle dynamique puisqu'il est possible d'introduire les blocs que lorsqu'ils sont nécessaires.

Les paramètres qu'il est possible de modifier en vue de mesurer les performances du système sont très nombreux. En vue de faciliter l'exploitation des résultats et leur interprétation, nous avons défini des programmes d'essais relativement particuliers mais qui ont le mérite de bien faire apparaitre l'influence de chacun des paramètres. D'autre part comme nous le verrons dans le dernier exemple, le choix d'un programme complexe ne permet pas de tirer des conclusions au niveau du fonctionnement.

Les résultats que nous allons analyser dans la suite de ce paragraphe ont été obtenus à l'aide de quatre programmes constituant un échantillon de jeux d'essais.

Pour les trois premiers, les temps d'exécution de chacun des blocs sont identiques. Avec cette hypothèse, le graphe des dépendances entre les blocs du programme apparait comme étant composé d'une suite de niveaux. Au premier niveau il y a les blocs exécutables au lancement du programme. Les domaines de sortie qu'ils produiront, rendront exécutables tous les blocs du deuxième niveau. De même les domaines de sortie du 2ème niveau provoqueront l'exécution des blocs du troisième niveau et ainsi de suite jusqu'au dernier niveau. Il est alors facile de montrer que s'il existe un arc de dépendance entre deux niveaux non successifs, celui-ci peut être remplacé par une suite d'arcs entre des niveaux successifs. Un exemple d'une telle transformation est donné figure 3. Celle-ci n'affecte aucunement le déroulement du programme.

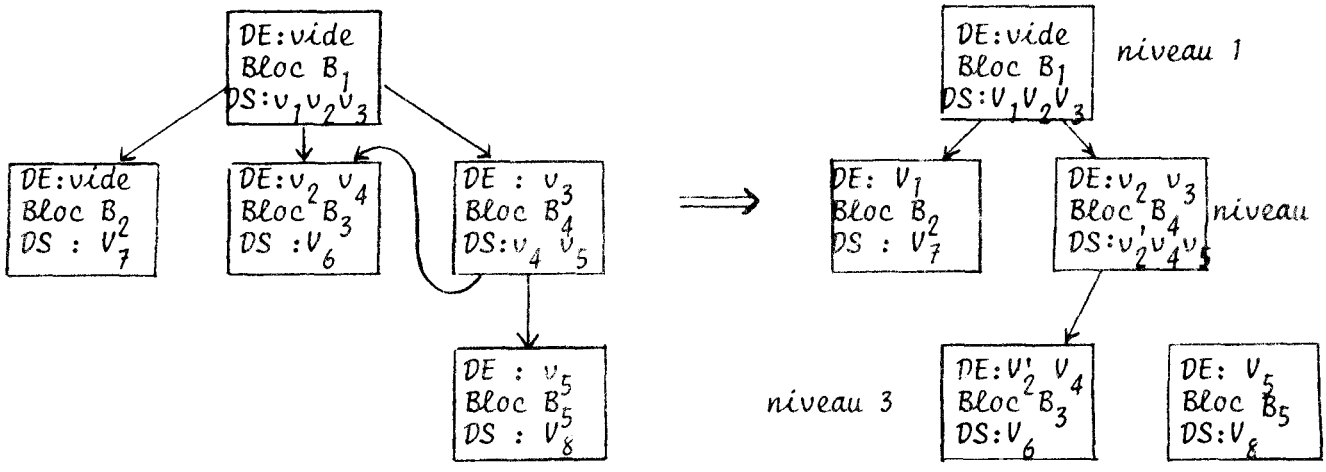
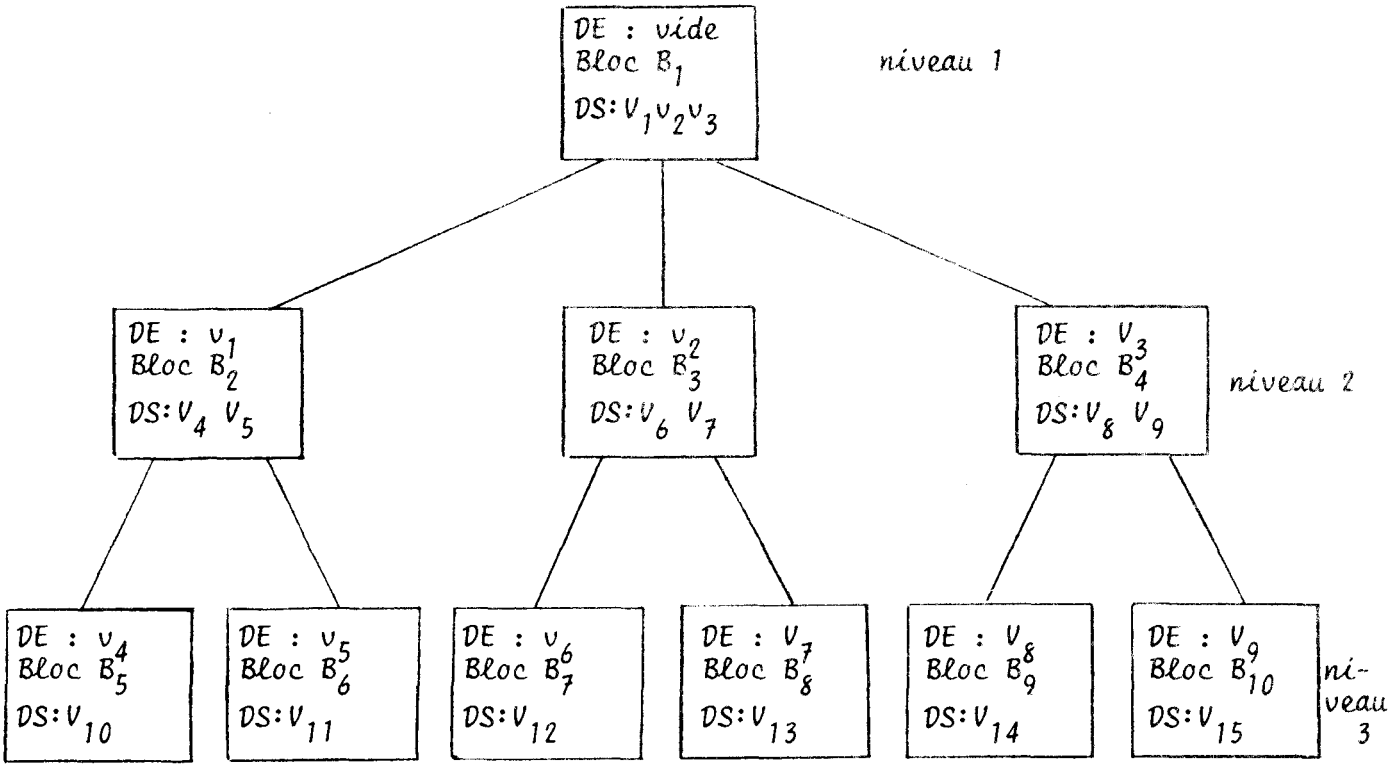


Fig. 3 : Transformation du graphe des dépendances

2.3. Parallèlisme maximum

Au cours de l'exécution d'un programme, le taux de parallèlisme c'est à dire le nombre de blocs exécutables simultanément, varie et passe par une valeur maximale. Pour tout programme, il existe donc un nombre optimal de processeurs au delà duquel, les performances du système ne peuvent plus être améliorées. En effet si le nombre de processeur continue à augmenter ceux-ci resteront inactifs et ne participeront pas à l'exécution du programme.

2.4. Exemple de simulation 1



Le programme est composé de 10 blocs dont un seul est exécutable initialement. La configuration minimale de l'anneau d'après la définition vue précédemment doit comporter 10 cadres. L'ensemble des résultats obtenus en faisant varier, le nombre de cadres dans l'anneau, le nombre de processeurs d'exécution et les temps d'exécution par blocs sont donnés sous la forme de courbes en annexe 1.

L'influence du temps d'exécution des blocs peut être mis en évidence en comparant les résultats obtenus sur notre système à ceux que l'on pourrait obtenir sur un système monoprocesseur.

temps d'exécution par bloc	temps de traitement monoprocesseur	temps de traitement avec MAUD
15	150	94
30	300	129
90	900	309

Le gain de temps obtenu est d'autant plus important que le temps d'exécution d'un bloc est grand. Le rapport entre le temps d'exécution et le temps de circulation joue un rôle primordial. Pour le découpage d'un programme en blocs,ceci implique que ces blocs aient un temps d'exécution relativement long et oriente le choix des parties de programme à regrouper de façon à minimiser les temps de circulation.

Les courbes données en annexe 1 représentent les variations du temps d'exécution total du programme en fonction de la longueur de l'anneau, donnée en nombre de cadres avec pour paramètre le nombre de processeurs d'exécution disponibles. L'ensemble des essais fait apparaître plusieurs caractéristiques remarquables.

- Pour chaque courbe il existe une longueur de l'anneau critique au delà de laquelle l'évolution du temps d'exécution total est linéaire. Pour un nombre de cadres inférieur on observe des variations importantes.

- La prolongation des droites obtenues dans la partie linéaire détermine un faisceau de droites concourantes sur l'axe des temps correspondant à une configuration de Maud pour laquelle la mémoire circulante ne comporte aucun cadre.

- L'ensemble des points qui détermine le fonctionnement du système se trouve sur les droites composant le faisceau. Toutefois il existe quelques cas particuliers, les points se trouvent alors sur un faisceau de droites parallèles au premières.

- La pente des droites qui composent le faisceau est proportionnel à la longueur de l'anneau. Elles sont définies par une équation de la forme

$T = To + k\ Nc$

où To représente l'ordonnée du point où toutes les droites sont concourantes et Nc le nombre de cadres.

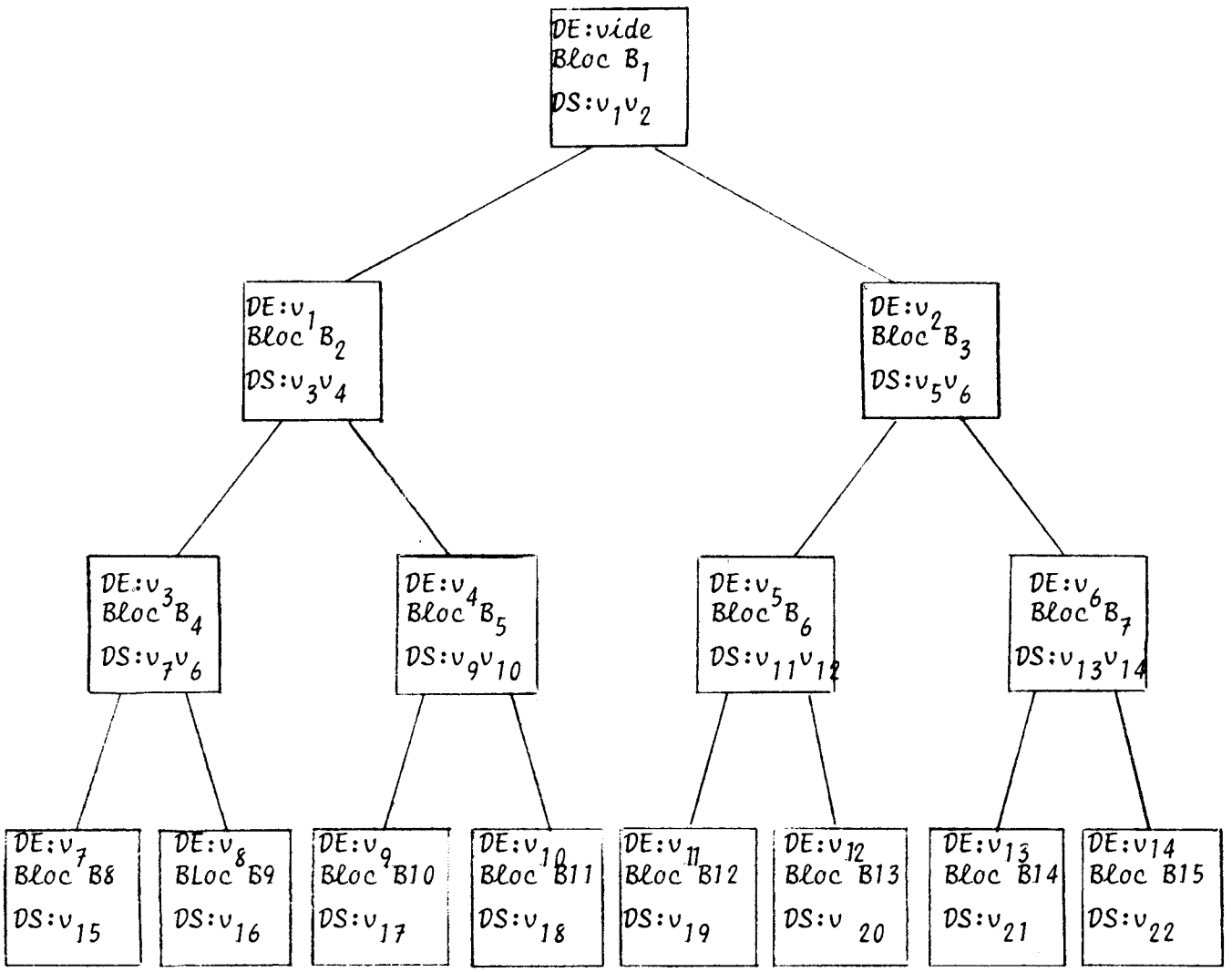
2.5. *Exemple de simulation 2 et 3*

Ces deux exemples avaient pour objectifs de confirmer les résultats obtenus à l'aide du premier et de mesurer l'influence de nouveaux

paramètres tels que le nombre de niveaux et le nombre de programmes actifs simultanément.

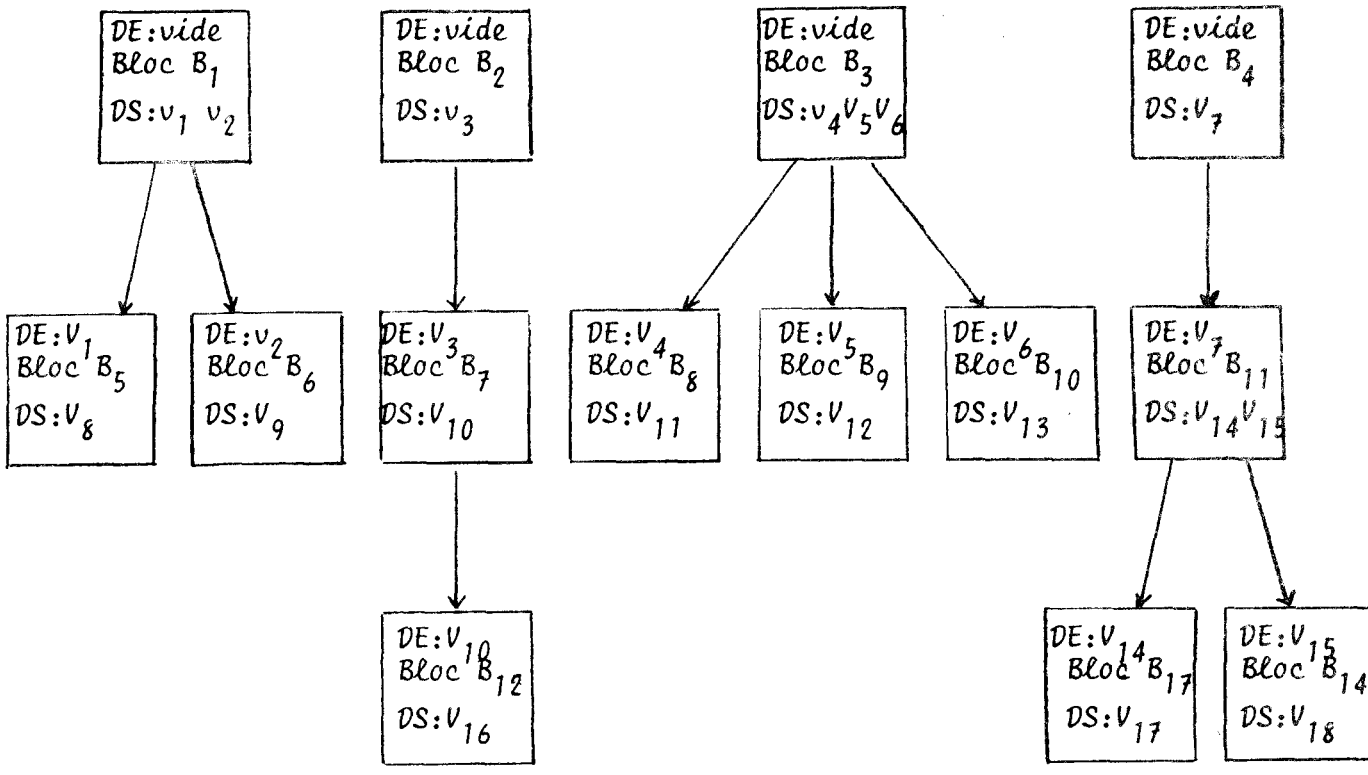
Exemple 2

Le graphe des dépendances a également une structure arborescente mais il comporte un niveau supplémentaire



Exemple 3

Les deux premiers exemples permettent d'évaluer les performances du système dans le cas où un seul programme est traité (monoprogrammation). Le troisième exemple porte sur le traitement de quatre programmes simultanément (multiprogrammation).



L'ensemble des courbes obtenues est fourni respectivement en annexe 2 et 3.

2.6. Analyse du faisceau et des variations

Pour les trois exemples que nous venons de citer, on retrouve l'existence d'un faisceau de droite qui détermine les performances du système. Celui-ci est entièrement déterminé par la pente minimale et le point de concours de l'ensemble des droites.

On peut remarquer que la droite de pente minimale n'est atteinte que sous certaines conditions.

- La longueur de l'anneau est supérieure ou égale à une valeur critique.

- Le nombre de processeurs d'exécution est égal au taux de parallélisme maximum. Une analyse fine du comportement du système montre que tous les processeurs doivent être actifs.

La valeur minimale de la pente ne dépend que du graphe des dépendances et plus particulièrement du nombre de niveaux

| P min = 2(nombre de niveaux - 1) + 1

Au contraire la longueur critique de l'anneau ne dépend pas du graphe des dépendances mais uniquement du temps d'exécution par bloc et de la façon d'introduire les blocs qui composent le programme dans l'anneau.

Lorsque les blocs sont déposés par l'injecteur niveau par niveau, les blocs en attente restent groupés en cours de traitement et l'espace vide dans l'anneau reste contigu. On atteint alors la longueur critique lorsque le temps de circulation pour effectuer un tour complet de l'anneau devient égal au temps d'exécution d'un bloc. Pour cette valeur, le domaine de sortie, produit à la fin de l'exécution d'un bloc sera déposé dans le cadre qu'occupait ce bloc et qui est devenu vide, après avoir effectué un seul tour d'anneau. En tenant compte de l'hypothèse quant à l'ordre des blocs, cela signifie que les domaines de sortie seront déposés dans l'anneau avant que les blocs en attente de ces valeurs n'aient effectué un tour complet. Dans le cas où les blocs sont déposés dans un ordre quelconque on peut définir la *longueur critique* de la manière suivante : c'est la longueur minimale de l'anneau telle que le dépot d'un domaine de sortie puisse se faire avant le passage des blocs en attente de ces valeurs devant le processeur d'exécution qui cherche à déposer et ceci indépendamment de la position des blocs en attente dans l'anneau au moment de la prise en charge du bloc exécutable. Ceci justifie d'ailleurs pleinement que pour cette longueur minimale, le temps de circulation pour effectuer un tour complet soit supérieur ou égal au temps d'exécution du bloc.

Lorsque cette limite est atteinte, la valeur de la pente minimale se calcule aisément en déterminant les retards entrainés par l'augmentation de la longueur de l'anneau. Puisque l'introduction des cadres supplémentaires se fait entre les processeurs d'exécution et le processeur mise à jour, le retard correspond au temps de décalage de deux cadres pour la transition entre chaque niveau. Le premier est dû au fait que le domaine de sortie sera pris en compte par le processeur mise à jour

un temps de décalage plus tard et la mise à jour du bloc en attente se fera avec deux temps de retard. Pour le dernier niveau, tout se passe comme s'il y avait un niveau supplémentaire mais en ne tenant compte que du retard introduit pour la prise en compte des derniers domaines de sortie.

exemple 1	3 niveaux P min = 5
exemple 2	4 niveaux P min = 7
exemple 3	3 niveaux P min = 5

On peut d'ailleurs remarquer à ce niveau que le choix du point d'introduction des cadres supplémentaires permet de modifier la valeur de la pente minimale. Si ceux-ci étaient introduits entre le processeur mise à jour et l'injecteur, la relation devient

P min = 2 x (nombre de niveaux - 1)

La mise à jour des blocs en attente se fait comme dans le cas précédent avec un temps de retard mais par contre c'est la prise en compte des blocs exécutables qui sera décalée. Il n'y a donc plus d'augmentation du temps total au dernier niveau.

La pente minimale n'est atteinte dans la partie linéaire que si le nombre de processeurs est optimal. Dans le cas contraire des blocs devenus exécutables ne seront pas pris en charge immédiatement par un processeur d'exécution. Ils seront donc amenés à faire un ou plusieurs tours d'anneau supplémentaires. C'est ce que nous appelerons dans la suite *conflit primaire*. Au delà du point critique, les blocs effectueront un tour supplémentaire. En fait c'est le rapport existant entre le temps d'exécution par bloc et le temps de circulation autour de l'anneau qui conditionne ce nombre de tours.

Pour un nombre de processeurs donné et toujours au delà du point critique, la pente de la droite qui décrit les performances du système est de la forme

$$P = P_{min} + k$$

où k est un entier dont la valeur dépend du nombre de conflits primaires

exemple 1

avec 6 processeurs P = P min

avec 3, 4 et 5 processeurs P = P min + 1

avec 2 processeurs P = P min + 2.

Le comportement du système peut être analysé d'une façon analogue lorsque la longueur de l'anneau devient inférieure à la valeur critique. On constate qu'il est possible de découper l'espace de variation du nombre de cadres et qu'il existe des points particuliers que nous appellerons *points critiques secondaires*. Ces points correspondent à une configuration de l'anneau pour laquelle tous les domaines de sortie sont déposés avant les blocs qui en attendent les valeurs. Ces points correspondent à une modification du comportement du programme. Ces différentes configurations particulières de l'anneau peuvent se définir de la manière suivante

ce sont les valeurs de Nc pour lesquelles la fonction $E(\frac{tex}{Nc+1})$ subit une discontinuité.

Nous pouvons d'ailleurs remarquer que la valeur du point critique correspond à une valeur de Nc pour laquelle $E(\frac{tex}{Nc+1}) = 0$.

Comme précédemment nous pouvons calculer la pente de la droite qui détermine les performances du système juste après un point critique secondaire, en évaluant les retards entrainés par l'augmentation de la longueur de l'anneau d'une unité. Le domaine de sortie sera pris en compte avec un temps de retard. Par contre pour la mise à jour d'un bloc en attente, il y aura un temps de décalage par tour effectué pendant l'exécution du bloc fournisseur. L'ensemble des temps accumulés provoque un temps de retard de $2+E(\frac{tex}{Nc+1})$ par transition entre deux niveaux successifs. En tenant compte du temps de retard dû à la prise en compte des derniers domaines de sortie la pente aura donc la valeur suivante,

$$P = [2 + E(\frac{tex}{Nc+1})] \text{ (nombre de niveaux - 1)} + 1$$

Nous pouvons d'ailleurs vérifier que la valeur de la pente minimale au delà du point critique est conforme à cette formule

exemple 1 pour un temps d'exécution par bloc = 30

si 10 < Nc < 15 P = 9

si 15 < Nc < 30 P = 7

si Nc > 30 P = P min = 5

Le calcul de la pente à l'aide de cette formule implique que les variations de pente d'une zone à l'autre dépendent du nombre de niveaux dans le graphe du programme. Pour l'exemple 1 cette transition se fait brutalement. Il n'en est pas de même pour l'exemple 2 pour lequel on passe par des pentes intermédiaires avant un point critique secondaire. La justification de ce phénomène ne peut se faire que si on analyse le comportement du programme de façon plus fine. Il est lié à l'existence de conflits que nous appelerons *conflits secondaires*. Lorsque la longueur de l'anneau augmente, certains domaines de sortie peuvent être déposés *avant* les blocs qui sont en attente de leurs valeurs et d'autres *après*. Dans ce deuxième cas nous dirons qu'il y a un *conflit secondaire*. La disparition progressive de ces conflits entraine des variations du temps total d'exécution et le passage sur des droites de pentes intermédiaires. Les différents essais que nous avons entrepris, ont fait apparaitre que la largeur de cette zone transitoire avant un point critique dépend du nombre de blocs par niveaux dans le graphe du programme.

La présence de conflits primaires dûe au fait que le nombre de processeurs est insuffisant, entraine également une augmentation de la pente de la droite. L'augmentation est proportionnelle au nombre de conflits et à la valeur de $E(\frac{tex}{Nc+1})$. L'analyse du comportement dans ce dernier cas est beaucoup plus complexe et semble peu intéressante.

La justification du point de concours des droites composant le faisceau est difficile puisqu'elle ne correspond pas à une configuration réaliste (nombre de cadres dans l'anneau = 0). Les essais que nous avons effectués montrent que ce point origine dépend

- du temps d'exécution par bloc
- du nombre de blocs dans le programme
- de l'ordre d'injection des blocs
- de l'ordre d'exécution des blocs

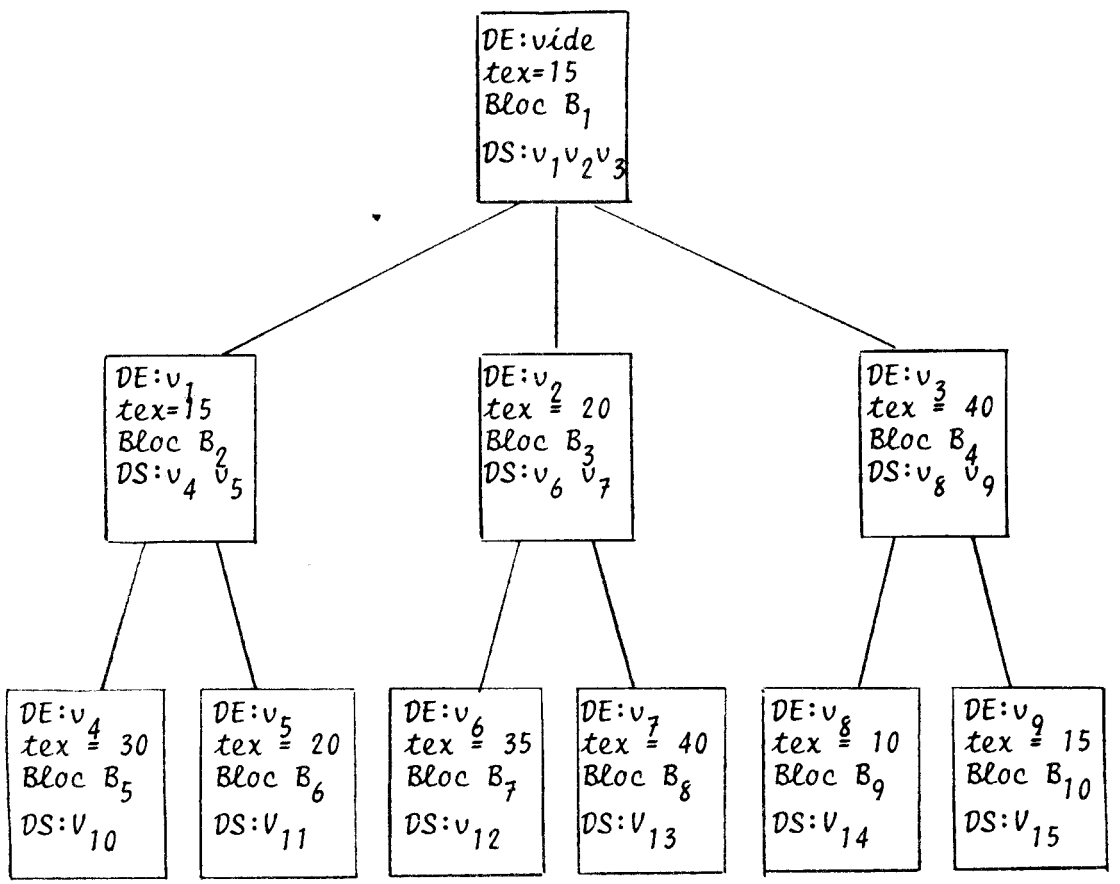
Dans le cas où les blocs sont injectés niveau après niveau on trouve que

> To = Tex + N_B-1
> où Tex est le temps d'exécution d'un bloc et N_B le nombre de blocs composant le programme.

Tout se passe comme si tous les blocs étaient exécutables initialement et tous les processeurs regroupés au niveau de la même fenêtre. Le terme N_B-1 n'est dû qu'au mode de fonctionnement de l'injecteur qui introduit un bloc à la fois dans l'anneau.

2.7. Exemple de simulation 4

L'analyse effectuée ne porte que sur des programmes particuliers pour lesquels les temps d'exécution des blocs sont tous identiques. Pour le quatrième programme d'essai, nous avons repris la même structure de graphe que l'exemple 1 mais en modifiant les temps d'exécution des blocs. Les différentes courbes obtenues sont fournies en annexe 4.



Bien que la structure du graphe n'a pas été modifié le taux de parallèlisme a été réduit. Il est possible de l'augmenter dans certains cas.Avec le choix des temps d'exécution, il y a au plus cinq blocs exécutables simultanément.

La forme des courbes n'est plus aussi remarquable que dans le cas précédent. Bien que l'évolution des performances semble se faire de façon linéaire pour certaines configurations, la définition d'un faisceau de droites concourantes n'apparait plus immédiatement. De plus la réalisation d'une analyse et la justification de certains phénomènes semblent impossibles ou tout au moins très complexes. Il n'est plus possible en effet de parler de niveaux dans le graphe du programme. Le temps total d'exécution dépend essentiellement du chemin critique du graphe c'est à dire le chemin de durée maximale allant du premier au dernier niveau.

2.8. *Conclusion*

L'analyse menée dans le cas des trois premiers exemples montre que avec certaines hypothèses, il est possible de déterminer, à partir du graphe du programme, la configuration optimale du système, c'est à dire celle pour laquelle le temps d'exécution du programme sera minimum. Elle sera obtenue pour un nombre de processeurs d'exécution optimum et pour une configuration de l'anneau correspondant à un point critique secondaire ou au point critique primaire.

Les graphes arborescents tel que nous les avons choisis peuvent ne pas paraître très réalistes.En effet tout programme doit se terminer par des sorties de résultats. L'analyse effectuée précédemment reste cependant valable. Il suffirait en effet de rajouter un bloc supplémentaire qui serait exécuté par un processeur d'entrée sortie et qui deviendrait exécutable en fin de programme.

3. *Simulation du modèle dynamique*

3.1. *Représentation du système*

L'introduction des primitives *execbloc* et *attendre* n'a entrainé que des adjonctions au niveau de la représentation du système. La structure de donnée que nous avons utilisée est donnée figure 4.

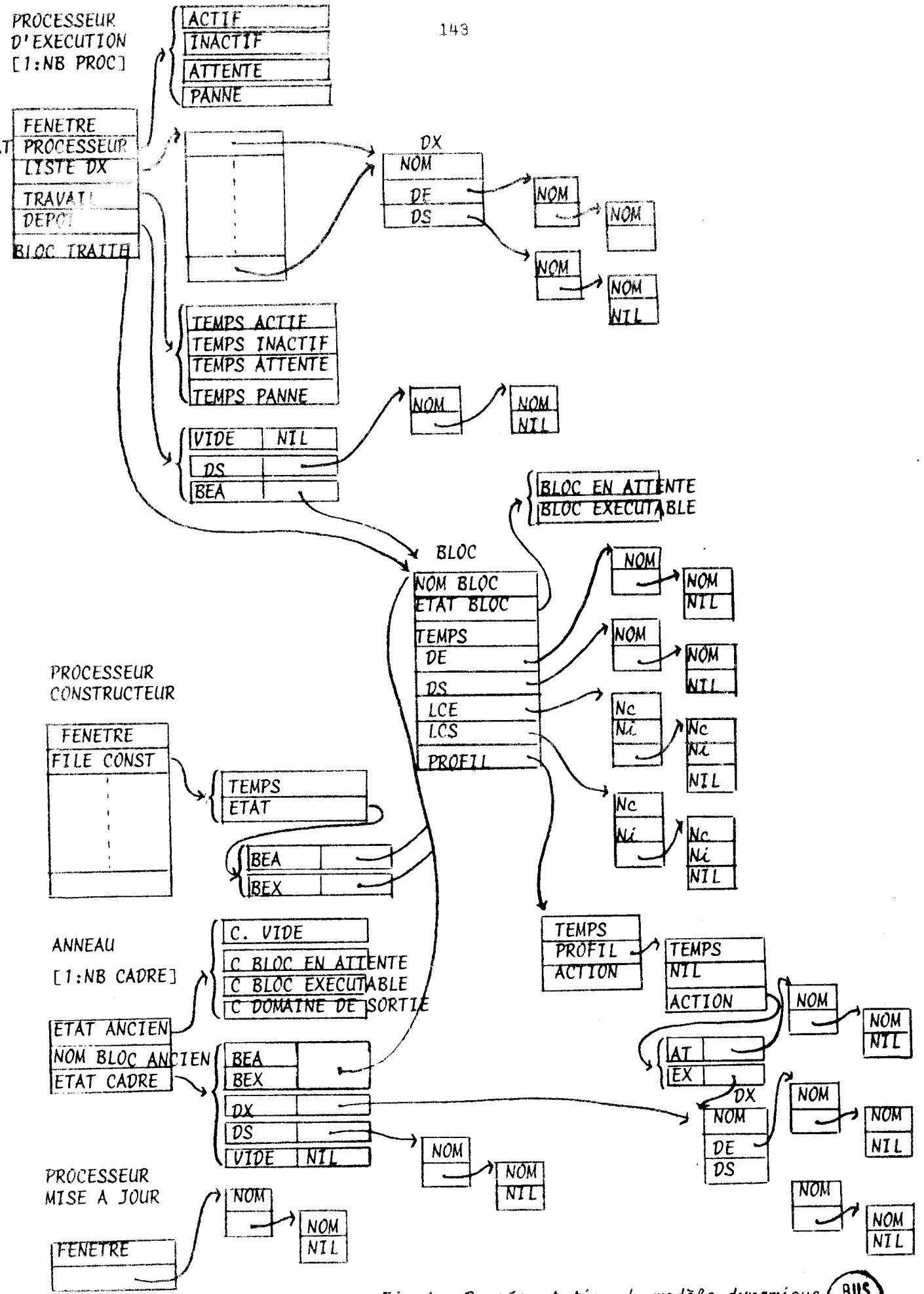


Fig 4 : Représentation du modèle dynamique

Les processeurs d'exécution doivent gérer en plus une liste de demandes d'exécution qui est représentée sous la forme d'un tableau de pointeurs qui designent l'emplacement des différentes demandes d'exécution en attente de dépôt. Celle-ci comporte en plus du nom du bloc appelé, deux listes de noms correspondant aux noms de communication du domaine d'entrée et du domaine de sortie.

La description d'un bloc ne peut plus se faire avec le temps d'exécution. Chaque bloc est décrit par un profil qui est une liste de durées et d'actions qui sont soit une primitive *execbloc* soit une primitive *attendre*. L'action suivante ne sera exécutée que lorsque la durée fournie dans la liste sera écoulée. Afin d'assurer correctement la gestion des communications, il devient indispensable d'introduire les listes de correspondances d'entrée et de sortie qui sont représentées sous la forme de couples (nom de communication, nom interne).

La description de l'anneau et du processeur mise à jour ne subit aucune modification. Le processeur constructeur quant à lui est représenté sous la forme d'une liste. Chaque élément de la liste comporte un temps et un état qui caractérise le nouveau bloc qui sera déposé. Le temps permet de déterminer l'instant à partir duquel le bloc pourra être déposé. Le processeur constructeur lorsqu'il reçoit une demande d'exécution ne peut délivrer instantanément le bloc demandé. Il doit consulter les blocs dont il dispose dans la bibliothèque. Le temps nécessaire pour réaliser l'ensemble de ces opérations sera l'un des paramètres de la simulation. Il sera évalué lui aussi par rapport au temps de décalage d'un cadre.

3.2. *Etat des travaux*

Bien que le simulateur du modèle dynamique soit opérationnel il ne nous a pas été possible pour l'instant de réaliser des essais en nombre suffisant pour dégager des résultats significatifs. D'autre part, le nombre de paramètres et la diversité des programmes d'essais que nous pouvons envisager nous amène à ne pas exploiter les résultats de la même manière que dans le cas du modèle statique. Mais à nous orienter vers une étude statistique du comportement du système.

3.3. *Conclusion*

Les résultats obtenus à l'aide du modèle statique font apparaître clairement le gain de temps par rapport à un système classique monoprocesseur. Celui-ci est d'ailleurs d'autant plus important que le temps d'exécution par bloc augmente. Dans le cas de l'exemple 1 il est de 38% pour un temps d'exécution comparable au temps de circulation et il atteint 66 % pour un temps d'exécution six fois plus important.

Les résultats auxquels on peut s'attendre avec le modèle dynamique seront encore plus favorables. En effet, les risques de blocage dans le modèle statique sont tels qu'il y a une longueur minimum de l'anneau pour que le programme puisse s'exécuter. Dans le cas du modèle dynamique, l'introduction des blocs se fait quand ceux-ci deviennent nécessaires pour la suite du traitement. Ceci permettra de réduire la taille de l'anneau et par conséquent le temps de circulation et permettra d'améliorer les performances globales du système.

CONCLUSION

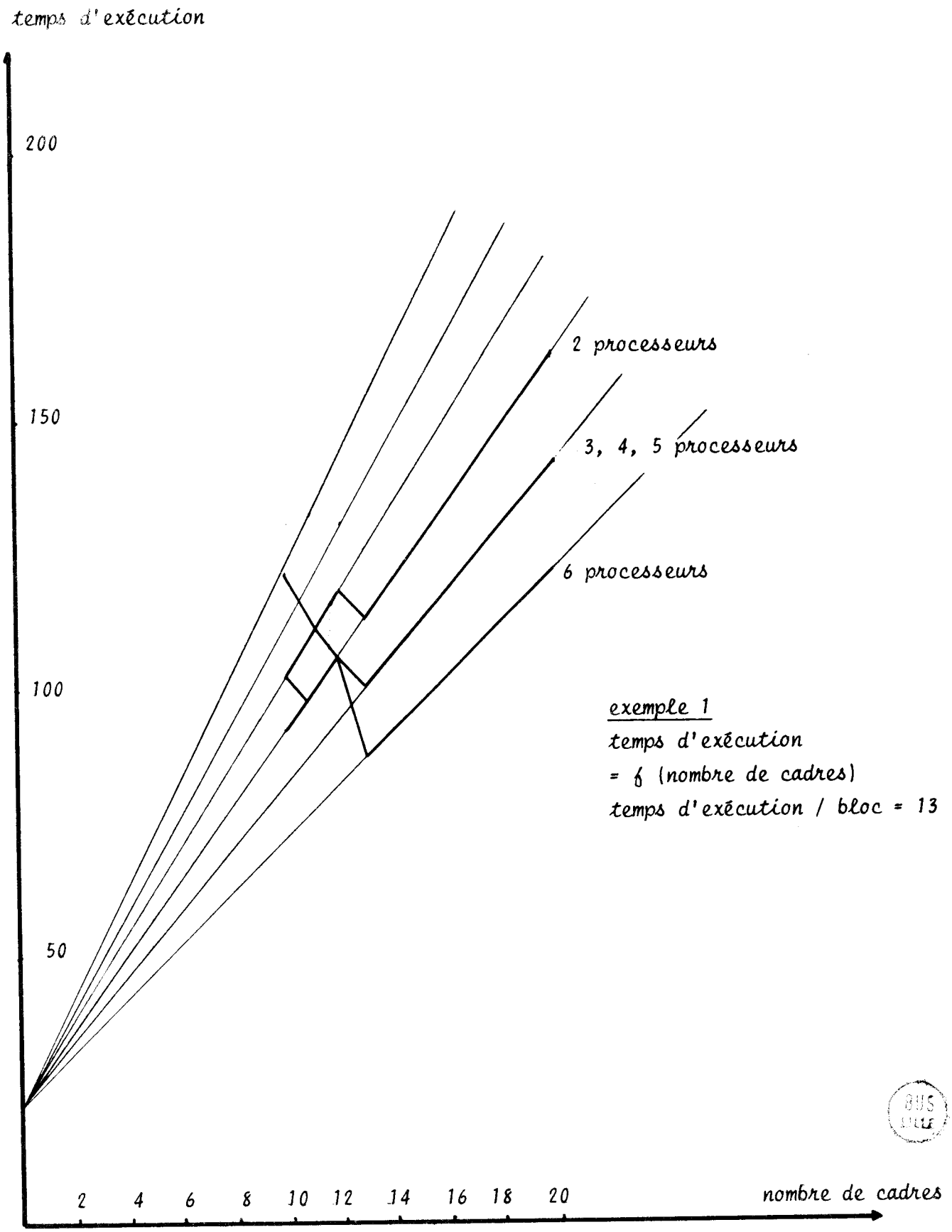
Dans ce travail nous n'avons abordé qu'une partie de l'architecture de Maud : La mémoire circulante, l'unité d'échange ainsi que le processeur pupitre qui ont été réalisés. Les essais effectués ont permis de vérifier la validité des choix initiaux.

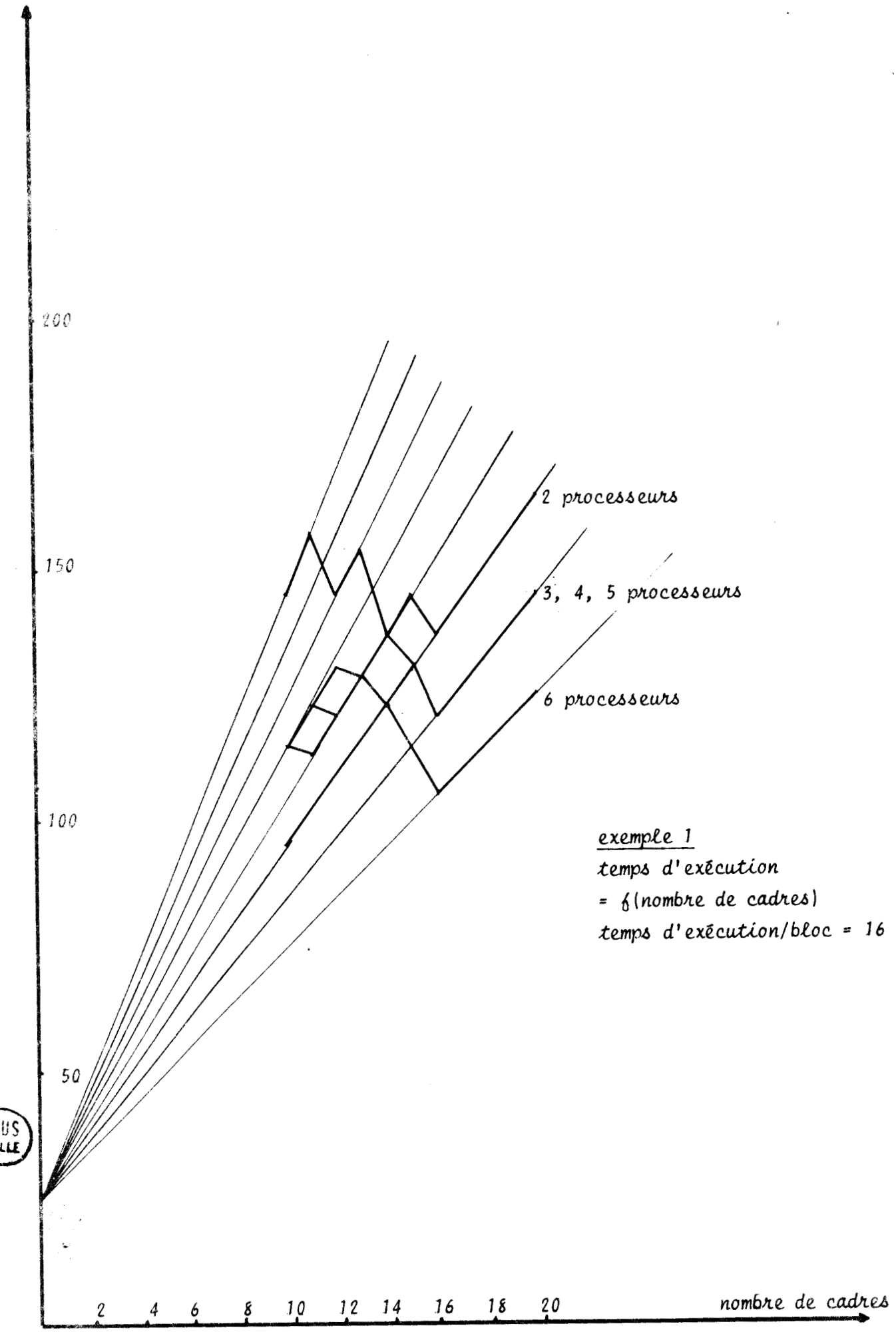
La suite du travail sur le plan matériel doit porter sur la réalisation des processeurs proprement dits : processeur d'exécution, processeur mise à jour et processeur constructeur. Outre le choix d'un processeur adapté sur lequel il sera possible d'implémenter les différentes primitives, il sera nécessaire de déterminer l'architure de la mémoire locale qui doit pouvoir être accédée par deux utilisateurs en exclusion mutuelle.

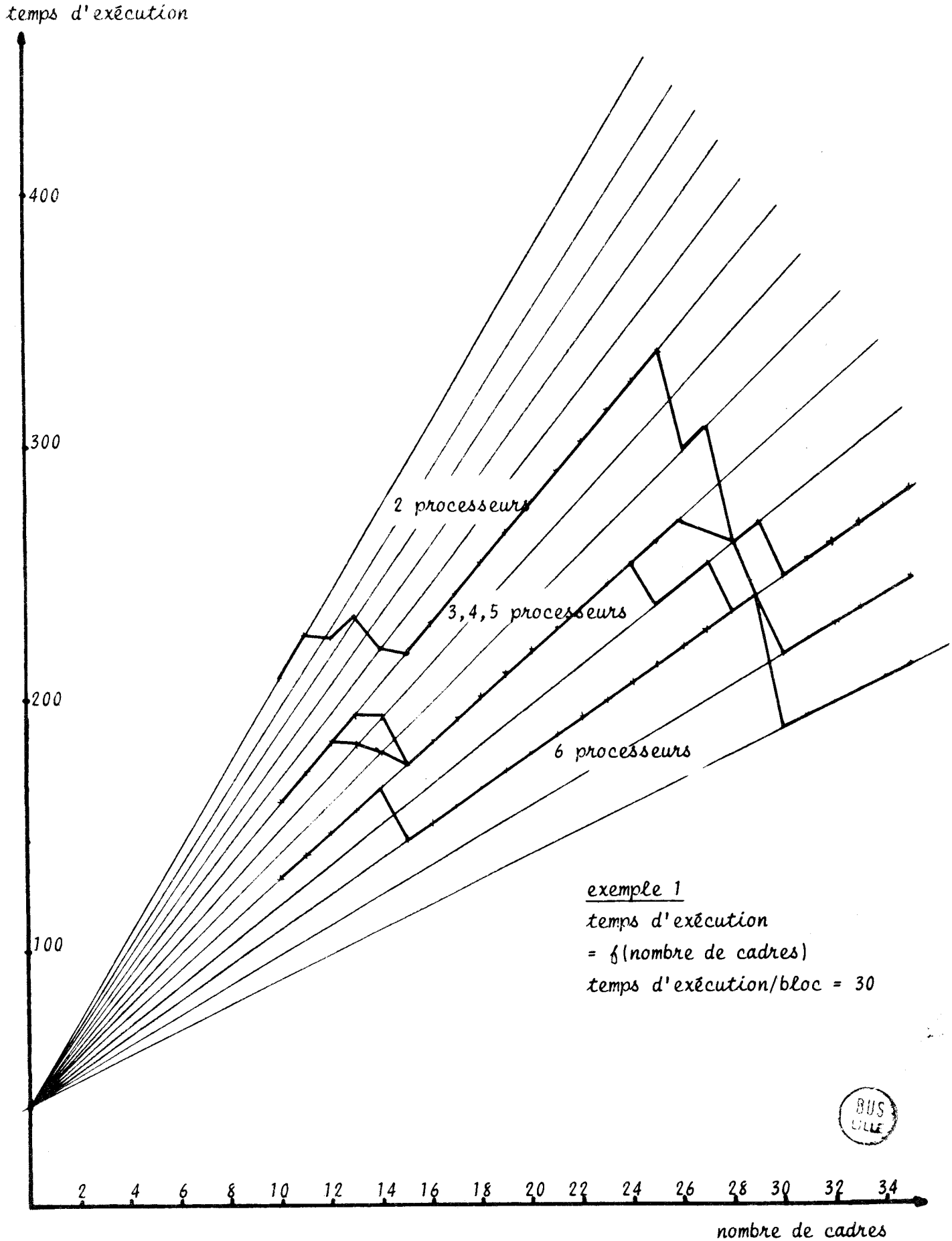
La simulation complète du modèle dynamique et une étude statistique pour mesurer l'influence des différents paramètres va permettre dans un premier temps de vérifier l'intérêt du modèle et des primitives *execbloc* et ***attendre***. Une analyse plus fine du comportement nous apportera des renseignements complémentaires pour la définition du processeur gérant d'anneau tant sur son rôle que sur son mode de travail.

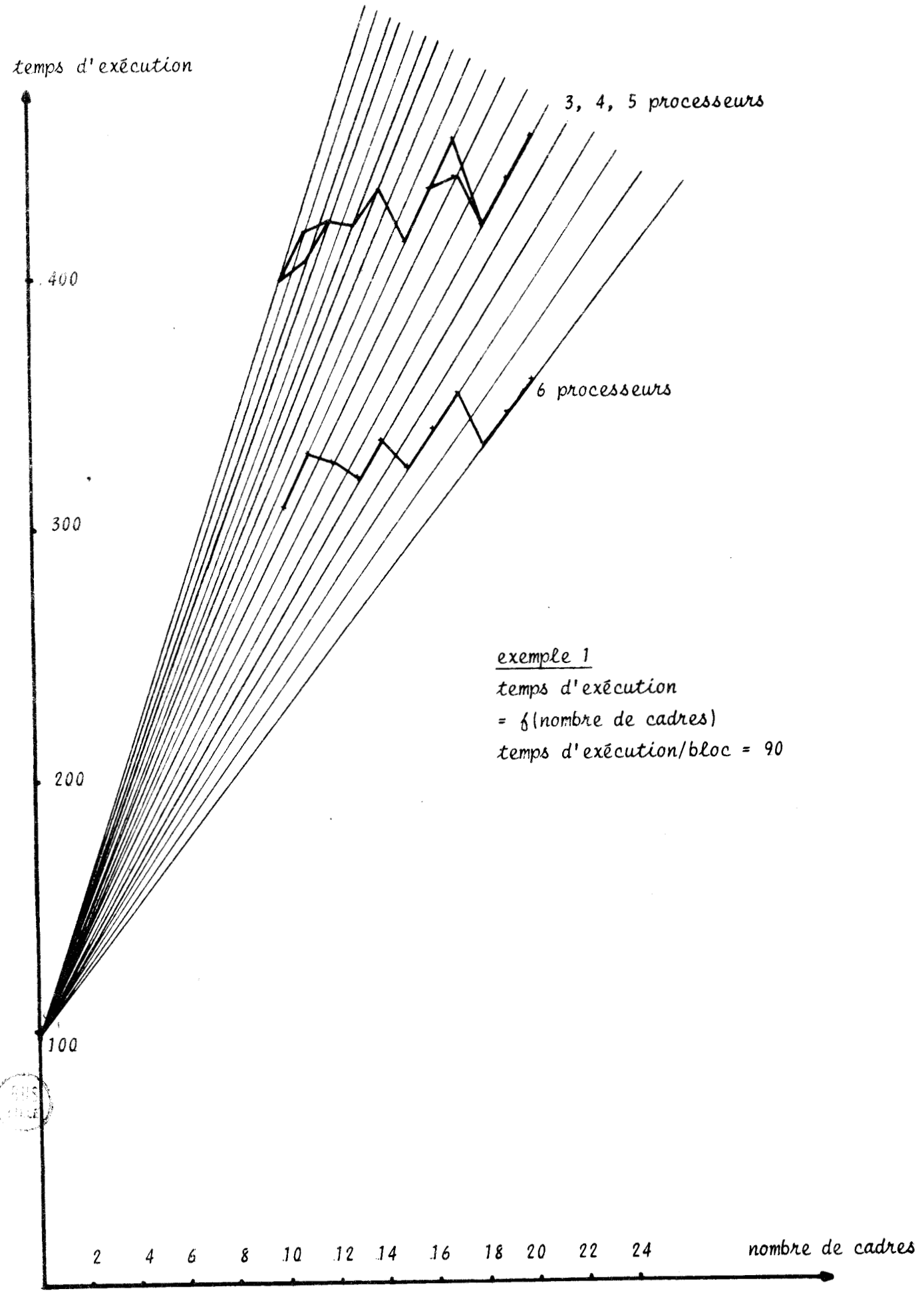
Parallèlement à ces deux aspects nous étudions tpus les problèmes logiciels, à savoir : Comment organiser un programme pour Maud ? Comment réaliser le découpage en blocs ? Quel langage utiliser pour écrire un programme et quel langage machine ?

ANNEXE 1

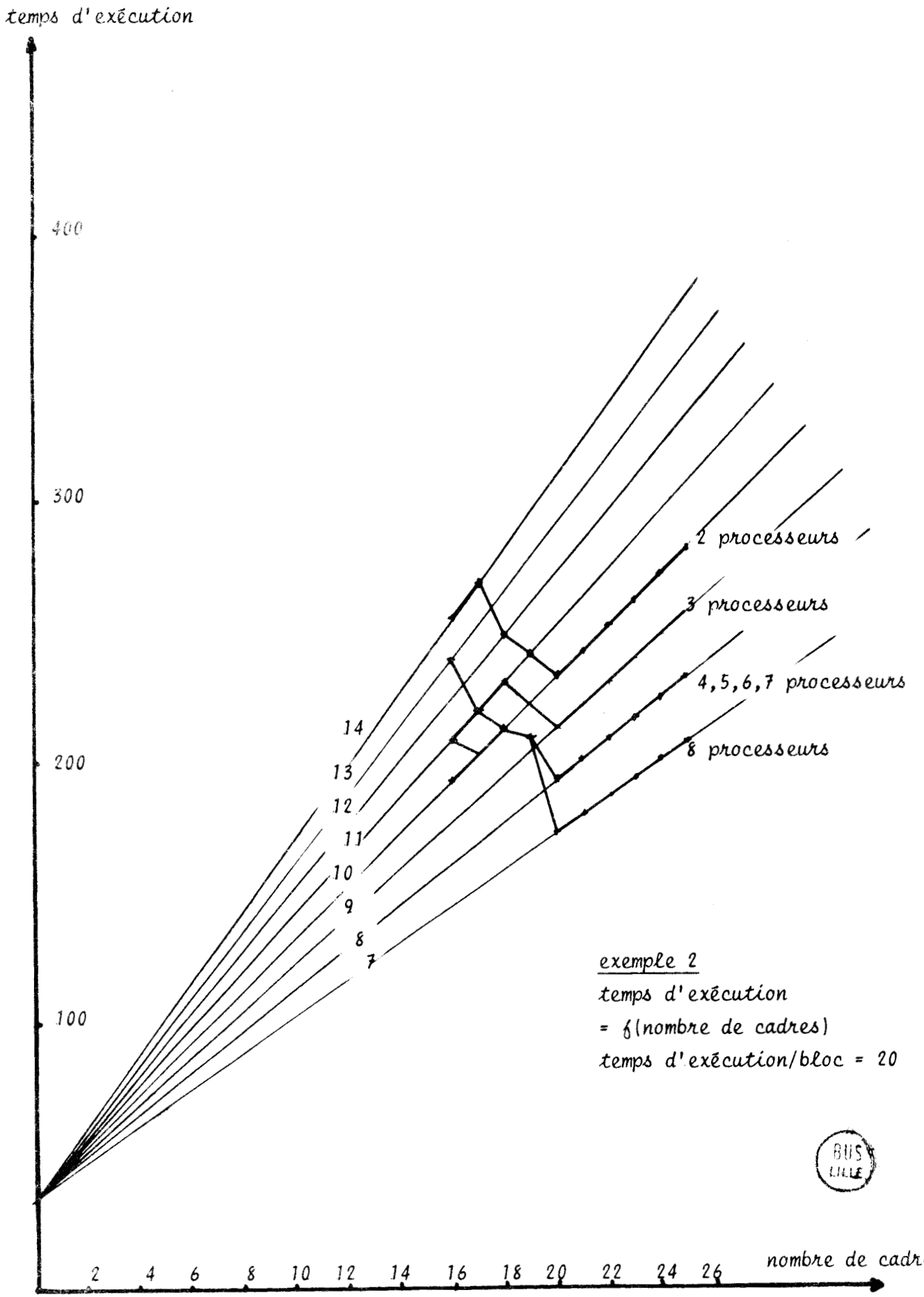

temps d'exécution
200
150
100
50
2 processeurs
3, 4, 5 processeurs
6 processeurs
exemple 1
temps d'exécution
= f (nombre de cadres)
temps d'exécution / bloc = 13
2
4
6
8
10
12
14
16
18
20
nombre de cadres


temps d'exécution
200
150
100
50
2 processeurs
3, 4, 5 processeurs
6 processeurs
exemple 1
temps d'exécution
= f(nombre de cadres)
temps d'exécution/bloc = 16
2
4
6
8
10
12
14
16
18
20
nombre de cadres

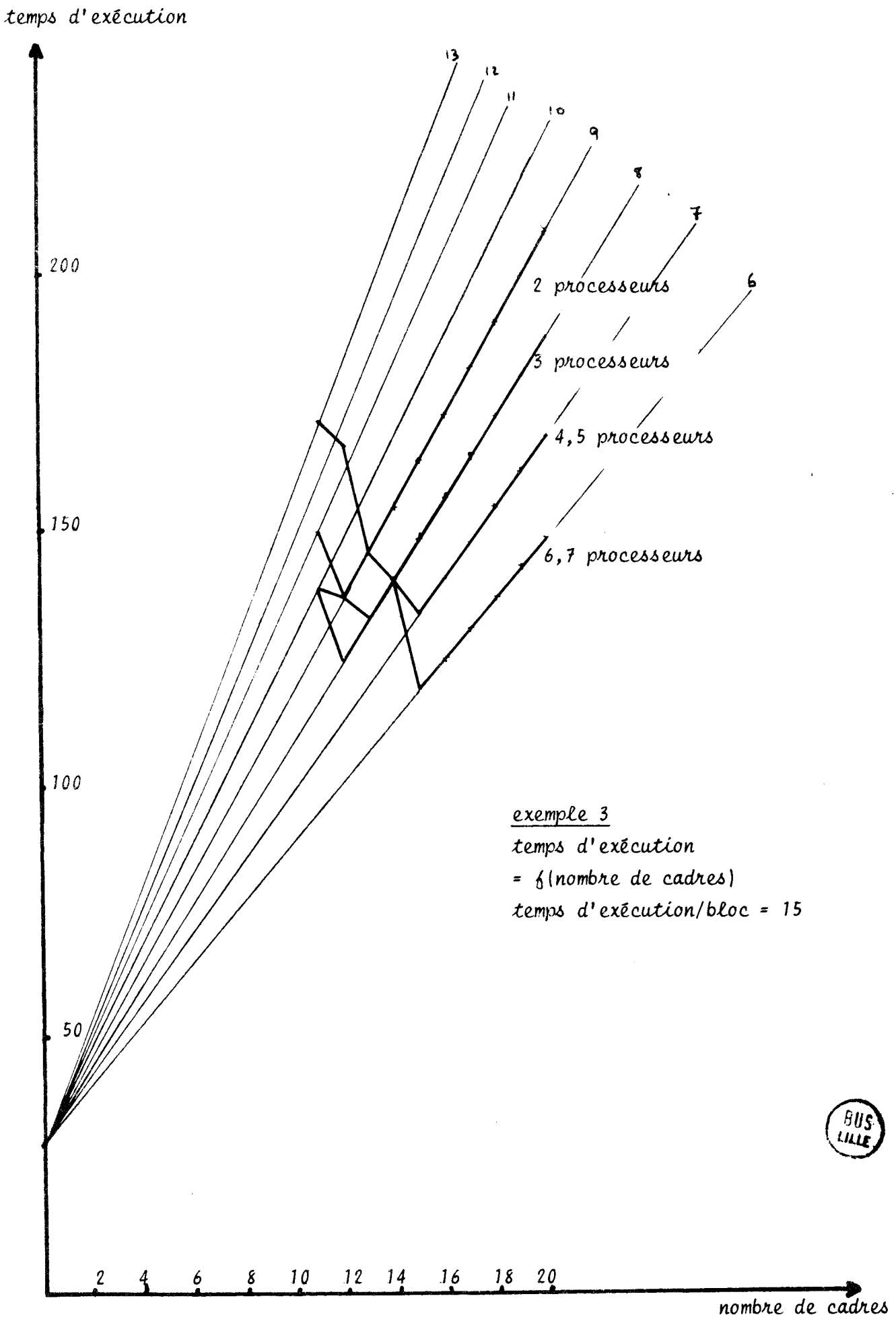

temps d'exécution
400
300
200
100
2 processeurs
3,4,5 processeurs
6 processeurs
exemple 1
temps d'exécution
= f(nombre de cadres)
temps d'exécution/bloc = 30
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34
nombre de cadres


temps d'exécution
3, 4, 5 processeurs
6 processeurs
400
300
200
100
exemple 1
temps d'exécution
= f(nombre de cadres)
temps d'exécution/bloc = 90
2
4
6
8
10
12
14
16
18
20
22
24
nombre de cadres

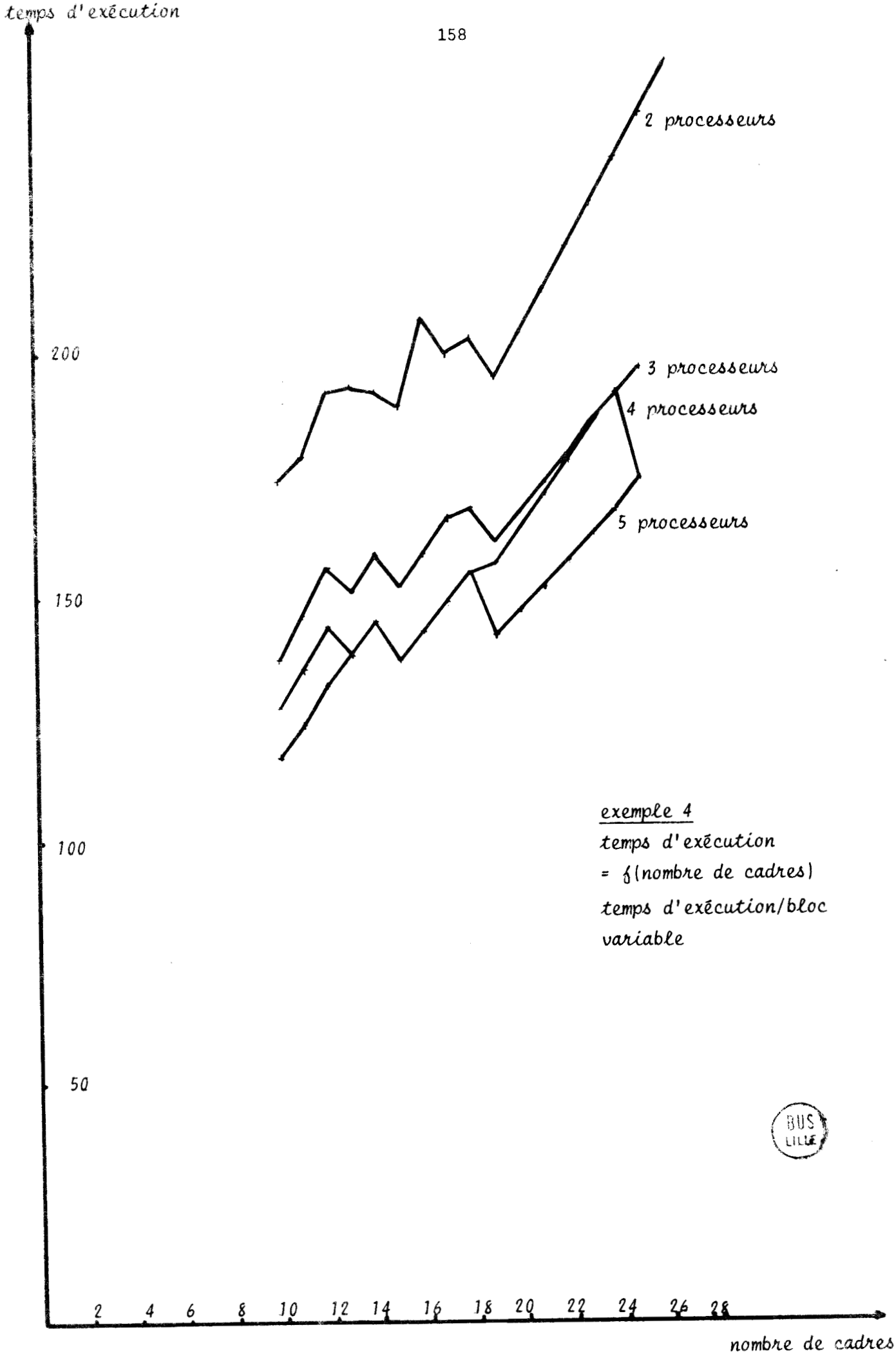
ANNEXE 2


temps d'exécution
400
300
200
100
2 processeurs
3 processeurs
4,5,6,7 processeurs
8 processeurs
14
13
12
11
10
9
8
7
exemple 2
temps d'exécution
= f(nombre de cadres)
temps d'exécution/bloc = 20
2 4 6 8 10 12 14 16 18 20 22 24 26
nombre de cadres

ANNEXE 3


temps d'exécution
13
12
11
10
9
8
7
6
200
150
100
50
2 processeurs
3 processeurs
4,5 processeurs
6,7 processeurs
exemple 3
temps d'exécution
= f(nombre de cadres)
temps d'exécution/bloc = 15
2
4
6
8
10
12
14
16
18
20
nombre de cadres

ANNEXE 4



exemple 4

temps d'exécution
= f(nombre de cadres)

temps d'exécution/bloc
variable

BIBLIOGRAPHIE

[AG 77] ARVIND et GOSTELOW K.P.
"A computer capable of exchanging processors for time"
Proceedings of IFIP Congress 1977

[AGP 78] ARVIND, GOSTELOW K.P. et PLOUFFE W.
"The ID preliminary report : an asynchronous programing language and computing machine"
TR 118 University of California. Irvine. Sept. 78

[BARN 68] BARNES G.H. et al
"The ILLIAC IV Computer"
IEEE Transactions on Computer Vol C17 N° 8 August 68

[CLER 79] CLERC P.
"Elaboration d'un nouveau modèle de permutations pour les mémoires circulantes. Application à la conception de disques électroniques"
Thèse université de Lille Mai 79

[CORD 78] CORDONNIER V.
"L'emploi des mémoires circulantes dans l'architecture des ordinateurs".
Publication du laboratoire de calcul de l'université de Lille 1978.

[CORD 78] CORDONNIER V.
"Architectures pipeline"
Support du cours IRIA sur le traitement parallèle.
Lille 29 mai - 2 juin 1978.

[CORD 79] CORDONNIER V.
"La mémoire circulante de MAUD"
Publication du laboratoire de recherche en architecture des systèmes et machines informatiques. Université de Lille N°2 1979.

[DEN 74] DENNIS J.B. et MISUNAS D.P.
"A preliminary architecture for a basic data flow processor"
Proceedings ACM SIGARH, IEEE Symposium on Computer Architecture
Dec 1974.

[DUP 77] DUPUY M., SCRIZZI A.
"Architecture matérielle d'un système multiprocesseurs"
Thèse université Pierre et Marie Curie Paris VI Janvier 1977

[DUR 77] DURRIEU G.
"Etude et définition de machine parallèle asynchrone à assignation unique"
Thèse Toulouse 1977

[EVAN 79] EVANS D.J. and WILLIAMS A.S.
"Analysis and detection of parallel processable code"
The computer journal vol 25 n° 1 1979

[FLYN 72] FLYNN M.J.
"Some computers organizations and their effectiveness"
IEEE Transactions on Computer C21 n° 9 sept 72

[GWG 78] GURD J., WATSON I., GLAUERT J.
"A multilayered data flow computer Architecture"
department of computer science, university of Manchester July 78

[KEN 80] KENNETH E., BATCHER
"Architecture of a massively parallel processor"
7th Annual symposium on computer Architecture La Baule 1980.

[LANG 76] LANG T. et STONE H.S.
"A shuffle exange Network with simplified control"
IEEE Transactions on Computer Vol C25 n° 1 Jan 76.

[LANG 76] LANG T.
"Interconnexions between processors and memory module using the shuffle exchange Network"
IEEE Transactions on Computer Vol C 25 n° 5 May 1976

[LEC 78] LECOUFFE M.P.
"MAUD : une machine d'assignation unique dynamique"
Publication du laboratoire de calcul de Lille n° 116 Sept 78.

[LEC 79a] LECOUFFE M.P.
"MAUD : A dynamic single assignement system"
Computers and digital techniques April 79 Vol 2 n° 2.

[LEC 79b] LECOUFFE M.P.
"Traitement dynamique en assignation unique par blocs"
Journées d'études "langages et machines cadencées par les données" Toulouse février 1979.

[LEC 79c] LECOUFFE M.P.
"Etude et définition d'un modèle de machine parallèle dynamique dirigée par les données"
Thèse 3ème cycle université de Lille Juillet 1979

[LEC 79d] LECOUFFE M.P. et TOURSEL B.
"The possible impact of software trends on microcomputer architecture".
EUROMICRO 5th symp. Göteborg. Aout 79

[LENF 77] LENFANT J.
"Fast random and sequential access to dynamic memories of any size"
IEEE Transaction on Computer Vol C26 n° 9 Sept 77

[LENF 77] LENFANT J.
"Parallel permutation of data : A benes Network control algorithm for frequently used permutations".
IRISA Rapport contrat SESORI IRIA 76-121.

[MAZA 78] MAZARE G.
"Structures multiprocesseurs, problèmes de parallèlisme, définition et évaluation d'un système particulier"
Thèse Grenoble juin 1978

[PAN 77] PANIGRAPHI G.
"The implications of electronic serial memories"
Computer Juillet 77

[PETI 80] PETITPREZ B.
"A flexible circulating memory for communication in a multi-processor"
EUROMICRO Symposium London Sept 80

[RUM 77] RUMBAUGH J.
"A data flow multiprocessor"
IEEE Transactions on computer vol C26 N° 2 Febr 77.

[STO 71] STONE H.S.
"Parallel processing with the perfect shuffle"
IEEE Transactions on computer Vol C20 N° 2 Feb 71

[STO 72] STONE H.S.
"Dynamic memory with enhanced data access"
IEEE Transactionson computer Vol C21 N° 4 Apr 72

[SYR 76] SYRE J.C. et al
"Techniques et exploitation de l'assignation unique"
Contrat SESORI 74-167 vol 9 (tomes 1 à 7)
Rapport final oct 76.

[SYR 78] SYRE J.C. et al
"Parallelism, control and synchronisation in a single assignement language"
Sigplan Notices n° 13 Janvier 78

[THUR 76] THURBER, K.J.
"Large scale computer architecture"
1976.

[TIEN 75] TIEN CHI CHEN et CHING TUNG
"Storage management operations in linked shift register loops"
IEEE Comp con 75.

[TOUR 79] TOURSEL B.
"Contribution à l'étude de l'architecture des ordinateurs : Proposition pour un nouveau mode de fonctionnement"
Thèse d'état université de Lille Sept 79.

[TREL 78] TRELEAVEN P.C. et al
"The design of highly concurrent computing system"
University of Newcastle TR July 78

[URS 73] URSCHLER G.
"The transformation of flow diagrams into maximally parallel forms"
Conference on parallel processing, Sagamore 1973

[VAN 79] VANLAER P.
"Etude d'une machine à flux simultanés"
Thèse université de Lille Avril 79

[VEIL 72] VEILLON et CAGNAT
"Cours de programmation en PL1 en langage PL1 : les concepts avancés"
Armand Colin 1972

[WONG 77] WONG C.K. et TANG D.T.
"Dynamic memories with faster random and sequential access"
IBM J. Res Devel. May 77