N° d'ordre : 368

ANNEE 1985

LIFL

U.A. 369 du C.N.R.S.

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

THÈSE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

FLANDRES ARTOIS

pour obtenir le titre de

DOCTEUR D'INGENIEUR

en Informatique

par

Gilles GONCALVES

CONTRIBUTION AU PROJET MAUD : ETUDE DU PROCESSEUR DE MISE A JOUR ET DES COMMUNICATIONS DANS L'ANNEAU

Thèse soutenue le 9 mai 1985 devant la Commission d'Examen

Membres du Jury	V.	CORDONNIER	Président
	C.	CARREZ	Rapporteur
Mme	M.P.	LECOUFFE	Examinateur
	B.	TOURSEL	Examinateur
	J.P.	SANSONNET	Examinateur

P R O F E S S E U R S CLASSE EXCEPTIONNELLE

M.	CONSTANT Eugène	I.E.E.A.
M.	FOURET René	Physique
M.	GABILLARD Robert	I.F.E.A.
M.	MONTREUIL Jean	Biologie
M.	PARREAU Michel	Mathématiques
M.	TRIDOT Gabriel	Chimie
M.	VIVIER Emile	Biologie
M.	WERTHEIMER Raymond	Physique

P R O F E S S E U R S 1ère classe

M.	BACCHUS Pierre	Mathématiques
M.	BEAUFILS Jean-Pierre (dét.)	Chimie
M.	BIAYS Pierre	G.A.S.
M.	BILLARD Jean (dét.)	Physique
M.	BOILLY Bénoni	Biologie
M.	BOIS Pierre	Mathématiques
M.	BONNELLE Jean-Pierre	Chimie
M.	BOUGHON Pierre	Mathématiques
M.	BOURIQUET Robert	Biologie
M.	BREZINSKI Claude	I.E.E.A.
M.	CELET Paul	Sciences de la Terre
M.	CHAMLEY Hervé	Biologie
M.	COEURE Gérard	Mathématiques
M.	CORDONNIER Vincent	I.E.E.A.
M.	DEBOURSE Jean-Pierre	S.E.S.
M.	DYMENT Arthur	Mathématiques

PROFESSEURS 1ère classe (suite)

M. ESCAIG Bertrand	Physique
M. FAURE Robert	Mathématiques
M. FOCT Jacques	Chimie
M. GRANELLE Jean-Jacques	S.E.S.
M. GRUSON Laurent	Mathématiques
M. GUILLAUME Jean	Biologie
M. HECTOR Joseph	Mathématiques
M. LABLACHE COMBIER Alain	Chimie
M. LACOSTE Louis	Biologie
M. LAVEINE Jean Pierre	Sciences de la Terre
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. MIGEON Michel Recteur à Grenoble	E.U.D.I.L.
M. MIGNOT Fulbert (dét.)	Mathématiques
M. PAQUET Jacques	Sciences de la Terre
M. PROUVOST Jean	Sciences de la Terre
M. ROUSSEAU Jean-Paul	Biologie
M. SALMER Georges	I.E.E.A.
M. SEGUIER Guy	I.E.E.A.
M. SIMON Michel	S.E.S.
M. STANKIEWICZ François	S.E.S.
M. TILLIEU Jacques	Physique
M. VIDAL Pierre	I.E.E.A.
M. ZEYTOUNIAN Radyadour	Mathématiques

P R O F E S S E U R S 2ème classe

M. ANTOINE Philippe	Mathématiques (Calais)
M. BART André	Biologie
Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mathématiques
M. BELLET Jean	Physique
M. BERZIN Robert	Mathématiques
M. BKOUCHE Rudolphe	Mathématiques
M. BODARD Marcel	Biologie
M. BOSQ Denis	Mathématiques
M. BRASSELET Jean-Paul	Mathématiques
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie
M. CARREZ Christian	I.E.E.A.
M. CAYATTE Jean-Louis	S.E.S.
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
M. CROSNIER Yves	I.E.E.A.
M. CURGY Jean-Jacques	Biologie
Mle DACHARRY Monique	Géographie
M. DAUCHET Max	I.E.E.A.
M. DEBRABANT Pierre	E.U.D.I.L.
M. DEGAUQUE Pierre	I.E.E.A.
M. DELORME Pierre	Biologie
M. DELORME Robert	S.E.S.
M. DE MASSON D'AUTUME Antoine	S.E.S.
M. DEMUNTER Paul	C.U.E.E.P.

PROFESSEURS 2ème classe (Suite 1)

M. DENEL Jacques	I.E.E.A.
M. DE PARIS Jean-Claude	Mathématiques (Calais)
Mle DESSAUX Odile	Chimie
M. DEVRAINNE Pierre	Chimie
M. DHAINAUT André	Biologie
Mme DHAINAUT Nicole	Biologie
M. DORMARD Serge	S.E.S.
M. DOUKHAN Jean-Claude	E.U.D.I.L.
M. DUBOIS Henri	Physique
M. DUBRULLE Alain	Physique (Calais)
M. DUBUS Jean-Paul	I.E.E.A.
M. FAKIR Sabah	Mathématiques
M. FONTAINE Hubert	Physique
M. FOUQUART Yves	Physique
M. FRONTIER Serge	Biologie
M. GAMBLIN André	G.A.S.
M. GLORIEUX Pierre	Physique
M. GOBLOT Rémi	Mathématiques
M. GOSSELIN Gabriel (dét.)	S.E.S.
M. GOUDMAND Pierre	Chimie
M. GREGORY Pierre	I.P.A.
M. GREMY Jean-Paul	S.E.S.
M. GREVET Patrice	S.E.S.
M. GUILBAULT Pierre	Biologie
M. HENRY Jean-Pierre	E.U.D.I.L.
M. HERMAN Maurice	Physique
M. JACOB Gérard	I.E.E.A.
M. JACOB Pierre	Mathématiques
M. JEAN Raymond	Biologie
M. JOFFRE Patrick	I.P.A.

PROFESSEURS 2ème classe (suite 2)

M. JOURNEL Gérard	E.U.D.I.L.
M. KREMBEL Jean	Biologie
M. LANGRAND Claude	Mathématiques
M. LATTEUX Michel	I.E.E.A.
Mme LECLERCQ Ginette	Chimie
M. LEFEVRE Christian	Sciences de la Terre
Mle LEGRAND Denise	Mathématiques
Mle LEGRAND Solange	Mathématiques (Calais)
Mme LEHMANN Josiane	Mathématiques
M. LEMAIRE Jean	Physique
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique
M. LOSFELD Joseph	C.U.E.E.P.
M. LOUAGE Francis(dét.)	E.U.D.I.L.
M. MACKE Bruno	Physique
M. MAIZIERES Christian	I.E.E.A.
M. MESSELYN Jean	Physique
M. MESSERLIN Patrick	S.E.S.
M. MONTEL Marc	Physique
Mme MOUNIER Yvonne	Biologie
M. PARSY Fernand	Mathématiques
Mle PAUPARDIN Colette	Biologie
M. PERROT Pierre	Chimie
M. PERTUZON Emile	Biologie
M. PONSOLLE Louis	Chimie
M. PORCHET Maurice	Biologie
M. POVY Lucien	E.U.D.I.L.
M. RACZY Ladislas	I.E.E.A.
M. RAOULT Jean François	Sciences de la Terre
M. RICHARD Alain	Biologie

PROFESSEURS 2ème Classe (suite 3)

M. RIETSCH François	E.U.D.I.L.
M. ROBINET Jean-Claude	E.U.D.I.L.
M. ROGALSKI Marc	Mathématiques
M. ROY Jean-Claude	Biologie
M. SCHAMPS Joël	Physique
Mme SCHWARZBACH Yvette	Mathématiques
M. SLIWA Henri	Chimie
M. SOMME Jean	G.A.S.
Mle SPIK Geneviève	Biologie
M. STAROSWIECKI Marcel	E.U.D.I.L.
M. STERBOUL François	E.U.D.I.L.
M. TAILLIEZ Roger	Institut Agricole
Mme TJOTTA Jacqueline (dét.)	Mathématiques
M. TOULOTTE Jean-Marc	I.E.E.A.
M. TURRELL Georges	Chimie
M. VANDORPE Bernard	E.U.D.I.L.
M. VAST Pierre	Chimie
M. VERBERT André	Biologie
M. VERNET Philippe	Biologie
M. WALLART Francis	Chimie
M. WARTEL Michel	Chimie
M. WATERLOT Michel	Sciences de la Terre
Mme ZINN JUSTIN Nicole	Mathématiques

CHARGES DE COURS

M. ADAM Michel	S.E.S.

CHARGES DE CONFERENCES

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

Je tiens à remercier

Monsieur **Vincent CORDONNIER**, Professeur à l'Université de LILLE I, qui, après m'avoir accueilli dans son laboratoire, m'a fait l'honneur de présider le Jury de cette thèse.

Monsieur **C. CARREZ**, Professeur à l'Université de LILLE I, qui a assumé la direction de cette thèse et qui n'a cessé de me prodiguer de précieux conseils.

Monsieur **J.P. SANSONNET**, Ingénieur à la CGE, d'avoir bien voulu examiner ce rapport.

Monsieur **B. TOURSEL**, Professeur à l'Ecole Universitaire des Ingénieurs de LILLE, pour sa participation à ce jury.

Madame **M.P. LECOUFFE**, Maître-Assistant à l'Université de LILLE I, avec qui j'ai eu de fructueuses discussions qui ont contribué à l'avancement de ce travail.

Madame **M.J. CARREZ**, Ingénieur au CNRS, pour sa précieuse aide apportée aux travaux de simulation.

Mesdames **C. LAVERDISSE et B. VANDROEMME** qui ont dactylographié ce texte avec beaucoup de gentillesse et de compétence.

Monsieur **H. GLANC** qui avec beaucoup de soin et de diligence a assuré le tirage de ce document.

I-INTRODUCTION

Une nouvelle génération d'ordinateurs appelée la 5ème génération est apparue depuis 1982 dans la course aux machines à hautes performances. Cette génération s'est fixée pour objectif de fournir les ordinateurs des années 1990 [MOT 83] destinés principalement aux traitements symboliques (base de données, programmation logique, système expert). Les quatre générations qui l'ont précédée durant ces quarantes dernières années reposent principalement sur une évolution de la technologie utilisée pour les construire : le tube électronique, le transistor, le circuit intégré et le circuit intégré à grande échelle (V.L.S.I.). Ces générations ont augmenté leur puissance de traitement numérique de sept ordres de magnitude tout en faisant chuter de manière significative leur prix de revient. On estime l'évolution des besoins en traitement de trois ordres de magnitude pour cette décennie afin de résoudre des problèmes complexes en physique, aérodynamique, météorologie, balistique, infographie, traitement de la parole, intelligence artificielle et autres...

La plupart des ordinateurs utilisés de nos jours n'ont guère subi de changements fondamentaux au niveau architecture matérielle ces trente cinq dernières années face au modèle élaboré par J.V. NEUMANN. Dans ce modèle un processeur unique exécute un flot d'instructions sous contrôle d'un registre nommé compteur de programme. Ce registre repère l'instruction du programme à exécuter. Une fois celle-ci exécutée le compteur de programme s'incrémente de manière à pointer sur l'instruction suivante à exécuter. A tout moment, une instruction au plus s'exécute. De plus, les langages de programmation supportés par ce modèle obéissent aux mêmes lois de séquencialité de l'exécution d'un programme.

De part sa structure générale, la machine de J.V. NEUMANN ou machine à flot de contrôle est un outil qui a prouvé son efficacité dans le passé. Toutefois, son mode d'exécution séquentiel reste un handicap pour une exécution rapide, et ceci malgré l'adjonction de technique "pipeline" dans le cycle traditionnel "recherche instruction, décodage, exécution". Les progrés enregistrés durant cette période sur la puissance de ces calculateurs sont principalement dus aux efforts pour développer des circuits de plus en plus rapides. Mais on peut alors se poser la question :

" Combien de temps cela peut-il durer encore ?"

Pour de nombreux chercheurs, la réponse aux besoins en puissance de traitement passe par la recherche et l'exploitation du parallélisme inhérent des programmes. Cette recherche devra se faire tant sur le plan architecture matérielle que sur le plan architecture logicielle. En effet, continuer à utiliser des langages de programmation classique conduirait à une surcharge du système pour résoudre les problèmes de communication et de synchronisation qui pénaliserait le gain obtenu par l'apport d'une architecture matérielle parallèle.

Trois grands groupes font figure d'école dans la course aux machines à hautes performances.

Le premier d'entre eux concerne les architectures bâties autour d'un monoprocesseur rapide de type flot de contrôle . De nombreux crédits sont investis pour l'étude de circuit à base de matériaux nouveaux comme l'arséniure de gallium. Des gouvernements comme celui des Etas-Unis d'Amérique avec le projet VHSIC (very high-speed integrated circuit) ou de la Grande Bretagne avec le programme VHPIC (very high performance integrated circuit) interviennent activement dans ce groupe. Pour des besoins de compatibilité avec leurs systèmes existants, les grands constructeurs d'ordinateurs en font leur principal cheval de bataille.

Le second groupe se caractérise par l'utilisation de quelques processeurs rapides et processeurs spécialisés pour supporter des techniques de "pipeline" et de vectorisation. C'est le cas notamment de certains constructeurs comme CRAY RESEARCH avec le CRAY-2 ou le CRAY X-MP et de CONTROL DATA CORPORATION avec le CYBER 205.

Enfin avec l'apparition de circuits VLSI (microprocesseur, coprocesseur), le dernier groupe entreprend la construction d'architectures regroupant plusieurs dizaines voire même plusieurs centaines de ces circuits par une exploitation systématique du parallélisme inhérent des programmes. Dans ce groupe interviennent pour l'essentiel des chercheurs de d'Universités à l'exception de la présence de certaines compagnies comme INMOS (Transputer) en Grande-Bretagne, DENELCOR (HEP 1) aux USA et la NIPPON TELEGRAPH AND TELEPHONE PUBLIC CORPORATION (5ème génération) au Japon.
Une des architectures des plus attractives qui figure dans ce groupe est l'architecture dirigée par les données ou à flot de données (Data flow computer). Dans une telle architecture le parallélisme maximal inhérent aux problèmes est exploité puisque toutes les entités ou instructions, pour lesquelles les données sont calculées, peuvent être exécutées en parallèle si le nombre de processeurs le permet.

L'objectif de cette thèse s'insère dans l'un des projets (le projet MAUD) developpés depuis quelques années au Laboratoire LA 369 dans le pôle "Architecture à Flot de Données".

MAUD est un modèle de machine dirigée par les données dans lequel le flot de données est appliqué au niveau macroscopique (i.e groupe d'instructions) plutôt qu'au niveau de l'instruction.

Le plan adopté pour cette thèse sera la suivant :

- le premier chapitre de type bibliographique présente une synthèse des de présente une synthèse des différents composants qui interviennent pour la définition d'une machine à contrôle décentralisé*. Une classification de ces machines proposée par TRELEAVEN y est présentée.

- le second chapitre est une description du projet MAUD tel qui se présentait au début de cette thèse. Il reprend de manière succincte les travaux qui ont fait l'objet de deux thèses [LEC 79] et [PET 81]

- le troisième chapitre propose une implémentation pour les Processeurs d'Exécution de MAUD. Ce sont ces processeurs qui ont la charge d'exécuter en parallèle les groupes d'instructions pour lesquels les données sont évaluées.

- le quatrième chapitre étudie un des processeurs spécialisés de MAUD, le processeur MISE A JOUR. Le rôle de ce processeur est très important dans la mesure où il fournit aux Processeurs d'Exécution les groupes d'instructions à exécuter.

- le cinquième chapitre présente des résultats de la simulation de modèle dynamique qui a été menée conjointement à la réalisation d'un prototype.

- Des extensions du modèle dynamique sont proposés dans le dernier chapitre afin de pallier aux insuffisances mises en évidence dans le chapitre précédent. Des résultats comparatifs de simulation y sont décrits.

Dans la conclusion, nous évoquons brièvement quelques idées pour la réalisation d'un second prototype plus performant.

Note * : le terme contrôle décentralisé est utilisé par opposition au contrôle centralisé de type J.V. NEUMANN évoqué plus haut.

CHAPITRE I

Introduction aux machines à contrôle décentralisé

INTRODUCTION

A. Architecture à contrôle centralisé type Von Newmann

B. Architecture de type non Von Newmann

1. Motivations
2. VLSI
3. Nouveaux langages de programmation

    a. langages fonctionnels
    b. langages orientés objet
    c. langages de programmation logique

C. Classifications de machines

1. Flux de données et flux d'instructions
2. Contrôle du séquencement
3. contrôle du séquencement et mécanismes d'accés aux données

D. Organisation à flot de contrôles

E. Organisation dirigée par les données

F. Organisation de type réduction

1. Réduction de chaines
2. Réduction de graphes

G. Exemples de travaux

1. Architectures dirigées par les données
2. Architectures dirigées par la nécessité

Introduction

Durant plus de trente années, les principes fondamentaux utilisés pour la conception d'architectures pour ordinateurs n'ont guère évolués [ORG 79]. Ces architectures sont toutes ou presque basées sur les principes établis par VON-NEUMANN en 1945.

Ces dix dernières années ont vu naître, avec l'avènement du microprocesseur une foison d'architectures nouvelles, toutes caractérisées par une volonté de rompre avec un ou plusieurs des principes précédents.

Dans ce chapitre, de type bibliographique, nous présentons un panorama des différentes voies de recherches qui ont contribué à la définition d'architectures nouvelles à contrôle décentralisé* c'est l'une de ces voies de recherches qui a été développée depuis 1978 au Laboratoire d'Informatique de l'Université de LILLE I par la conception et la réalisation d'un modèle d'architecture dirigée par les données appelé MAUD (Machine à Assignation Unique Dynamique) [LEC 81].

Les lecteurs familiers du domaine des architectures à contrôle décentralisé pourront se passer de la lecture de ce chapitre.

Note * : Dans cette catégorie d'architecture on place des architectures parallèles (multi-processeurs) dont le contrôle est décentralisé (pas de processeur maître) ou répartie. L'exécution concourante d'une tache sur une telle architecture se caractérise par les points suivants :

- plus d'un processus participe à cette tache
- chaque processus n'a qu'une vue partielle de l'état global des autres processus.

A. Architecture à contrôle centralisé type Von-Neumann

C'est Von-Neumann qui a fait faire en 1945 le pas décisif à la mécanisation du traitement digital de l'information en introduisant deux nouveaux concepts :

* le programme enregistré : le programme à exécuter et les données du programme sont mémorisés dans les cellules mémoires du calculateur
* la rupture de séquence : introduction d'une instruction de rupture de séquence conditionnelle permettant de rendre automatique les opérations de décision logique.

Depuis la plupart des ordinateurs fonctionnant selon le schéma originel de Von-Neumann, nous les appelerons machines de type Von-Neumann.

Ces machines sont caractérisées par :

* une organisation linéaire de cellules mémoires de taille fixe
* un espace d'adressage des cellules mémoires à un seul niveau
* un langage machine de bas niveau : on effectue des opérations simples sur des opérandes élémentaires.
* un contrôle centralisé et séquentiel de l'exécution appelé aussi contrôle topographique
* une architecture monoprocesseur comprenant un processeur (P), un système de communication (C) et une mémoire (M) (Fig. 1).

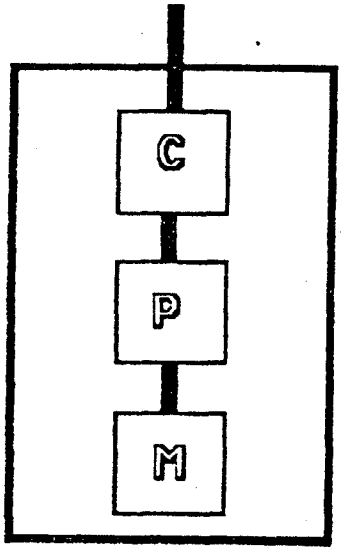


Fig. 1 Machine Type Von-Neumann

La renonciation à tout ou partie de ces cinq principes peut conduire à définir des machines que nous appelerons non Von Neumann.
Cependant il existe une hiérarchie dans les ruptures avec les principes traditionnels de Von-Neumann. La remise en question essentielle concerne le contrôle de l'exécution d'un programme. C'est cette voie de recherche que nous allons détaillée par la suite.
D'autres voies de recherche ont conduit néanmoins au développement de machines langages LISP [LEC 77], [SAN 84] ou ADA [BRY 83] marquées par le désir de supprimer le fossé sémantique séparant les langages évolués des langages machines.

B. Architectures de type non Von-Neumann

1. Motivations

Les progrès réalisés ces vingt dernières années dans certains domaines de recherche permettront d'envisager d'autres organisations d'architectures qui devront répondre à des besoins nouveaux. Ces domaines recouvrent plus particulièrement :

* Intelligence artificielle dans laquelle on trouve des méthodologies d'expression de la connaissance et du raisonnement, des techniques d'interface homme-machine telle que la reconnaissance de la parole ou la synthèse d'image.
* le génie logiciel qui offre de nouveaux langages de haut niveau (ADA, MODULA II, PROLOG, OCCAM), des environnements de programmation ou ateliers logiciels construits autour de système d'exploitation aux structures souples comme UNIX.
* le génie matériel qui propose des architectures multiprocesseurs, des architectures à contrôle distribué ou des réseaux d'ordinateurs.
* la technologie du VLSI apporte de nouveaux circuits plus puissants et de nouveaux outils d'aide à la conception de circuits personnalisés.

La question que l'on est amené à se poser pourrait être :
"quelles doivent être les caractéristiques de l'architecture d'un ordinateur à vocation générale ?".
De nombreux chercheurs sont d'accord pour affirmer que celle-ci sera du type architecture à contrôle décentralisé.
Parmi les domaines de recherche susceptibles d'apporter une réponse, on peut citer pêle-mêle :

- les supercalculateurs
- les processeurs VLSI
- les machines langage (PROLOG - ADA - SMALLTALK)
- les processeurs base de données
- les ordinateurs de la cinquième génération

Bien que les motivations des différents groupes de recherches qui contribuent à l'élaboration de ces architectures soient variées on peut néanmoins souligner quatre grandes directions interagissantes.

* les composants VLSI : l'utilisation de composants VLSI permet d'envisager des architectures composées d'un grand nombre de processeurs identiques intégrant les fonctions mémoire, communication et calcul sur un même circuit
* langages nouveaux : De nouvelles classes de langages, parmi lesquelles on ne citera que les plus répandues, c'est-à-dire les langages fonctionnels, les langages de logique du 1er ordre et les langages orientés objet, semblent particulièrement bien adaptés aux architectures non conventionnelles
* exploitation systématique du parallélisme de manière à accroître les performances de l'ordinateur dans des données de pointe comme la météo, la reconnaissance de la parole et l'avionique
* le transfert de fonctions programmées vers des fonctions cablées afin d'obtenir des temps de réponse plus rapides.

2. VLSI

Les échelles d'intégration permettent de concevoir des microprocesseurs contenant 100.000 transistors et bientôt près de 10^6 transistors. L'IPAX 432 d'Intel [MIN 81], le MAC 32 des laboratoires BELL ou le 32032 de National Semiconductor [MIN 84] sont les précurseurs de cette nouvelle génération. La technologie submicronique est sur le point d'aboutir et permettra la conception de circuits comportant plusieurs centaines de milliers de transistors. Cependant il est à noter qu'au plus l'intégration devient dense, plus les couts de conception et de tests augmentent dramatiquement. C'est pourquoi MEAD et CONWAY [MEA 80] ont largement contribués à ce domaine, en définissant une nouvelle méthodologie dans la réalisation des circuits à haut degré d'intégration.

Un circuit VLSI bien conçu sera un circuit dont l'architecture possèdera une ou plusieurs des propriétés suivantes :

* Le circuit est défini autour d'un nombre restreint de cellules élémentaires. Ainsi peu de cellules différentes ont besoin d'être conçues et testées. Les circuits systoliques, avec leur structure régulière et répétitive, semblent bien adaptés aux VLSI et suscitent un engouement assez marquant auprès des chercheurs. H.T. KUNG, pour ne citer que le plus connus d'entre eux, à proposé l'implémentation de plusieurs circuits systoliques parmi lesquels un circuit systolique programmable spécialisé [FIS 83].

* Les chemins de données et de contrôles sont simples et réguliers. Chaque cellule est interconnectée sur un réseau, localement et de façon identique, évitant les connections longues et irrégulières et facilitant ainsi les extensions futures par association de plusieurs circuits.

* Les communications intercellules devront se faire entre cellules adjacentes.

* L'architecture utilisera au maximum le pipeline ou le multitraitement.

Le circuit Transputer développé chez INMOS [BAR 83] est un exemple frappant de la volonté de banaliser le problème de la communication des processeurs à l'intérieur d'une architecture multiprocesseurs . Celui-ci possède quatre voies de communications et un ensemble de primitives de synchronisation câblées permettant de communiquer avec quatre processeurs voisins.

3. Nouveaux langages de programmation

De nouveaux langages de programmation, pour lesquels le parallélisme est implicite, semblent bien adaptés aux architectures à contrôle décentralisé. Trois catégories de ces langages ont particulièrement marqué les annales de la recherche ces dernières années. Ce sont les langages fonctionnels, les langages orientés objets et les langages de programmation logique.

3.a. Les langages fonctionnels

Un programme écrit dans un langage de programmation fonctionnelle consiste en une liste d'équations sur lesquelles le programmeur ne fait aucune hypothèse quant à l'ordre d'exécution de celle-ci. En règle générale ces langages se différencient des langages usuels par le fait que :

- Les équations d'un programme n'ont pas forcément besoin d'être exécutées séquentiellement. Seule la dépendance des données permet d'établir un ordre à l'exécution autorisant ainsi un parallélisme à l'exécution.

- La notion de variable (i.e. objet dans la valeur peut varier en cours d'exécution) n'apparaît pas.

J. BACKUS a défini dans son article [BAC 78] une syntaxe permettant de décrire un langage fonctionnel à l'aide de "formes".

Dans cette famille de langages on peut différencier deux catégories :

- Les langages à assignation unique.

- Les langages applicatifs purs.

* Les langages à assignation unique : Ces langages respectent la règle d'assignation unique qui précise <<qu'une variable ne peut figurer qu'une fois au plus dans une partie gauche d'une instruction d'affectation d'un programme>>. De nombreux langages ont été développés parmi lesquels on trouvera LAU [COM 76], LUCID [ASH 77], ID [ARV 78], VAL [ACK 79]. Cette règle d'assignation unique est très pratique pour déterminer les dépendances de données d'un programme et par delà détecter le parallélisme potentiel d'un programme. Ces langages sont bien adaptés aux machines dirigées par les données.

* Les langages applicatifs purs : Cette famille de langages qui regroupe des langages comme PURE LISP [MCC 62], SASL [TUR 79], se caractérise par l'absence d'opérations de contrôle (si, tant que, etc...). Un programme se présente comme une suite de fonctions mathématiques et seules les dépendances de données qui existent entre ses fonctions permettent d'imposer un ordre à l'exécution autorisant ainsi le parallélisme. Une variable d'un programme ne peut être réécrite ce qui implique que l'insertion d'un nouvel élément à une structure de données se traduit par la création d'une nouvelle structure comprenant l'ancienne structure et le nouvel élément. Un exemple de programme écrit en SASL nous montre que celui-ci semble particulièrement adapté aux machines à contrôle par nécessité (demand-driven).

définition. fact M where fact 0 = 1
fact M = M * fact (M-1).

Exécution pour M = 3. fact 3

1ère substitution
(3 * fact 2)
2ème substitution
(3 * (2 * fact 1))
3ème substitution
(3 * (2 * (1 * fact 0)))
4ème substitution
(3 * (2 * (1 * 1)))
1ère réduction
(3 * (2 * 1))
2ème réduction
(3 * 2)
3ème réduction
(6)

3.b. Les langages orientés "objet"

Ces langages sont basés sur la sémantique des acteurs (SMALLTALK [BYT 81], PLASMA [MAR]) et implémentent des opérations de bases comme :

- Création d'un acteur.
- Transmission de message.
- Continuation.

La sémantique des acteurs repose sur le concept unique d'acteurs dans lequel l'évaluation d'un programme est décrite en terme de transmission de messages. Un acteur est une entité capable de recevoir des messages et d'émettre des messages en réponses aux messages reçus. La réception d'un message encore appelé "évènement" constitue l'élément fondamental qui permet de décrire le comportement d'un programme en une suite ordonnée d'évènements.

Le parallélisme vient du fait que plusieurs évènements peuvent se dérouler simultanément.

3.c. Les langages de programmation logique

Les récents progrès obtenus dans le domaine du raisonnement automatique ont permis de donner des schémas de base pour la formulation des problèmes en programmation logique du 1er ordre (5ème génération [YOK 82]). Le langage PROLOG développé en 1972 par COLMERAUER et ROUSSEL à l'Université de Marseille est certainement le plus répandu de tous. Un programme PROLOG consiste en une liste de clauses liant les objets entre-eux. Ces clauses se subdivisent en deux catégories les prédicats et les assertions.

Un prédicat est une relation du type père (Jacques, Henri) qui signifie que Jacques est le père d'Henri.

Les assertions sont des relations du type :

grand-père (X, Y) ← père (X, Z), [mère (Z, Y) ou père (Z, Y)]

avec X, Y, Z variables.

Cette assertion signifie que X est le grand-père de Y que si et seulement si les conditions suivantes sont vérifiées :

X père de Z et (Z mère Y ou Z père Y).

Pour de plus amples informations concernant la méthologie de la programmation logique on se reportera aux ouvrages de R. KOWALSKI [KOW], [KOW 74].

De tels langages préfigurent les logiciels capables de favoriser la mise en oeuvre d' architectures parallèles ou distribuées, évoquées dans le paragraphe précédent.

C. Classifications des machines

1. Flux de données et flux d'instructions

FLYNN [FLY 72] a fourni une classification des architectures des ordinateurs en considérant le flux des données et le flux des instructions séparemment. Celle-ci a permis de recenser quatre types d'organisations :

* SISD (Single Instruction Single Data) caractérise les monoprocesseurs qui déroulent un flot d'instructions sur un flot de données.

* SIMD (Single Instruction Multiple Data) permet de dérouler simultanément un flot d'instructions sur plusieurs flots de données (cf. ILLIAC IV).

* MISD (Multiple Instruction Single Data) certains classent dans cette catégorie les machines "pipelines".

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

Cette classification, bien que couramment utilisée pour spécifier d'un point de vue macroscopique une architecture, ne permet pas de prendre en compte un certain nombre de critères significatifs dans les structures multiprocesseurs tels que :

- Nature du contrôle du séquencement des instructions.

- Le mécanisme d'accès aux données du programme.

- Les communications entre les processeurs.

- La manière dont les processeurs coopèrent (contrôle centralisé ou réparti).

2. Contrôle du séquencement

TOURSEL [TOU 81] a comblé en partie ces lacunes en définissant des schémas de machines qui prennent en compte la nature du contrôle du séquencement des cellules à exécuter. Une cellule est formée d'une partie d'instruction et une partie donnée. Cette classification a permis de recenser 8 modes de contrôle qui se répartissent en trois classes.

- Les machines à commande centrée sur les instructions dans lesquelles la fonction d'enchaînement qui permet de déterminer la ou les cellules suivantes a exécuter, dépend de la partie instruction et non de la partie donnée de la cellule en cours.

- Les machines à commande centrée sur les données.

- Et les machines à commande mixte dont la fonction d'enchaînement dépend à la fois de l'emplacement de l'instruction et de l'emplacement de la donnée de la cellule en cours de traitement.

Plusieurs nouveaux modes de contrôle apparaissent dans sa classification qu'il serait intéressant d'implémenter dans une architecture.

3. Contrôle du séquencement et mécanismes d'accès aux données

TRELEAVEN [TRE 81] a proposé une classification dans laquelle sont répertoriés les cinq types d'organisations fondamentales :

- Les organisations à flot de contrôles.
- Les organisations à flot de données.
- Les organisations à commande par la nécessité.
- Les organisations logiques.
- Les organisations orientées acteurs.

Cette classification bien que voisine de la précédente (nature du contrôle est spécifiée) se différencie de celle-ci par le fait que le mécanisme d'accès aux données y est décrit.

Nous décrivons ci-dessous les critères de cette classification.

3.a. Mécanisme d'accès aux données

Ce mécanisme définit la manière dont les opérandes d'une instruction sont obtenus. On recense trois types d'accès :

* Accès immédiat (opérande littéral). L'opérande est connu à la compilation et figure dans chaque instruction utilisant cet opérande.

* Accès par valeur concerne les opérandes dont la valeur est envoyée aux instructions qui l'utilisent dès que celle-ci est connue.

* Accès par référence. L'opérande est accessible par référence c'est-à-dire en précisant une adresse où celui-ci a été mémorisé.

3.b. Mécanisme de contrôle

Il permet d'indiquer la fonction d'enchaînement des instructions à exécuter. Il existe trois modèles :

- Modèle séquentiel : Le mécanisme de contrôle est du type topographique, les instructions s'exécutent séquentiellement.

- Modèle parallèle : Le mécanisme de contrôle signale la disponibilité des opérandes d'une instruction. L'instruction est exécutée lorsque tous ses opérandes sont devenus disponibles.

- Modèle récursif : Le mécanisme de contrôle signale qu'un argument est nécessaire. Une instruction est exécutée quand l'un des arguments qu'elle produit est demandé par l'instruction en cours.

Après avoir défini ces deux mécanismes on peut effectuer une étude comparative des différentes organisations citées plus haut [TRE 84].

D. Organisation à flot de contrôles

Ce type d'organisation possède un mécanisme d'accès aux données immédiat ou par référence. Les références font partie intégrante de l'instruction et permettent d'accéder a un ensemble de cellules mémoires partagées. Généralement le mécanisme de contrôle est du type séquentiel et plus rarement du type parallèle.

Pour illustrer ces principes nous allons prendre un exemple volontairement très simple. Soit à évaluer l'expression X = (Y+Z) * (Y-2) où X, Y, Z sont des variables initialisées respectivement à nil, 10, 2.

Dans un modèle à mécanisme de contrôle séquentiel le programme machine correspondant à l'exemple pourrait être du type quadruplet :

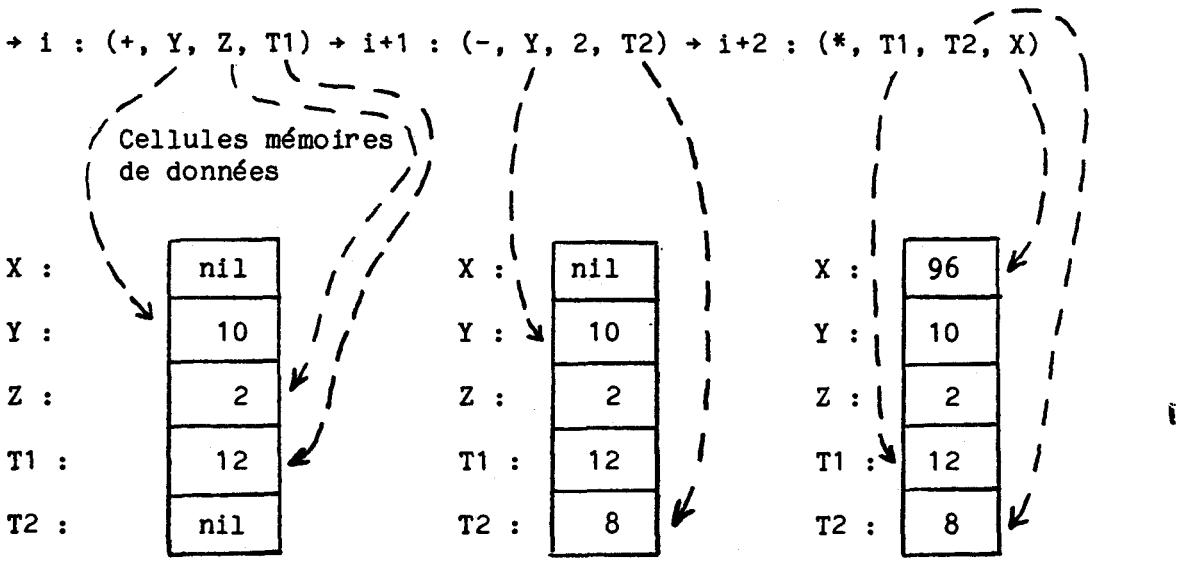
i :	(+, Y, Z, T1)	
i+1 :	(-, Y, 2, T2)	
i+2 :	(*, T1, T2, X)	

cellules mémoires de programme

où T1, T2 sont des variables temporaires
Y, Z sont des références mémoires
2 un littéral entier

numéro cellule programme — quadruplets

L'exécution de ce programme est décrit ci-dessous avec les notations suivantes : ⟶ mécanisme de contrôle
⇢ mécanisme d'accès aux données par référence



La fonction d'enchaînement de l'instruction suivante a exécuter respecte la topographie du programme enregistré.

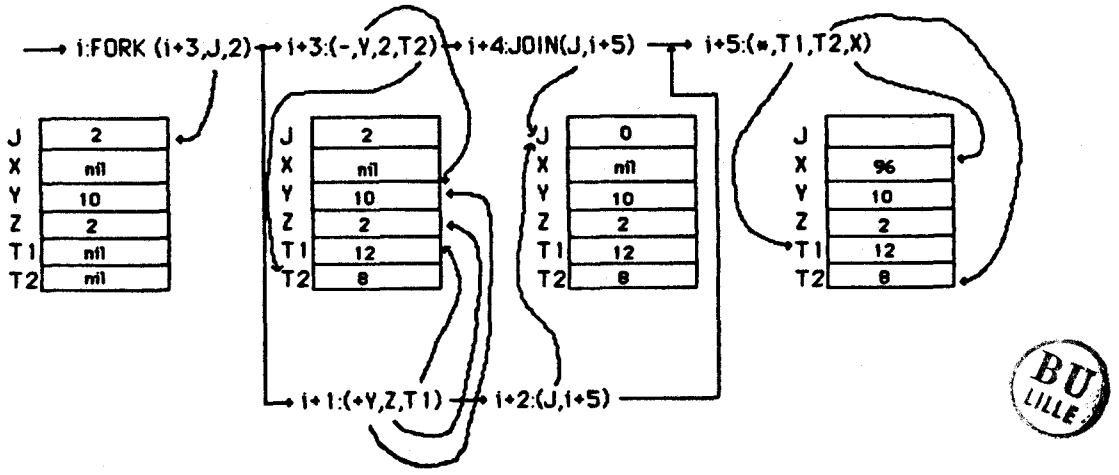
Dans une organisation à contrôle parallèle on dispose d'outils pour exprimer le parallélisme d'un progromme. L'écriture de l'exemple précédent pourrait être la suivante en utilisant les opérateurs FORK et JOIN définis par CONWAY [CON 63].

Cellules mémoires
de programme

i	FORK (i+3, J, 2)
i+1	(+, Y, Z, T1)
i+2	JOIN (J, i+5)
i+3	(- ; Y, 2, T2)
i+4	JOIN (J, i+5)
i+5	(*, T1, T2, X)

où J est un compteur qui indique le nombre de séquences se déroulant simultanément.

L'exécution de ce programme serait la suivante sur une machine biprocesseurs :



Sur cet exemple trop simple n'apparaît pas le gain en temps d'exécution, cependant celui-ci peut être considérable dans le cas où les séquences à exécuter simultanément comportent plusieurs dizaines d'instructions.

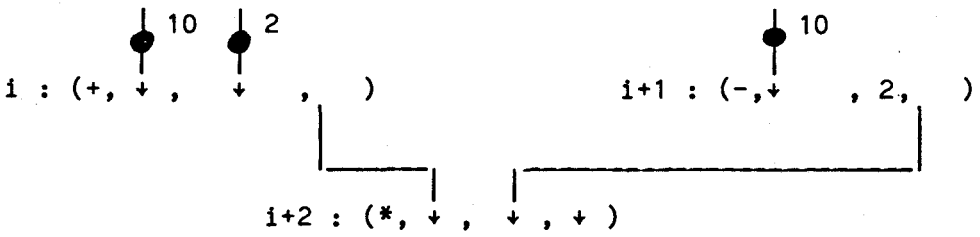
De par leurs mécanismes indépendants de contrôle d'enchaînement et d'accès aux données les organisations à flot de contrôle s'adaptent parfaitement aux programmes manipulant des structures de données complexes (celles-ci figurent en un seul exemplaire) alors qu'elles entraînent un "overhead" supplémentaire pour les programmes évaluant des expressions simples (nombreuses références mémoires pour la mise à jour de variables temporaires).

E. Organisation à flot de données

Une organisation à flot de données est caractérisée par un mécanisme de données de type littéral ou par valeur et par une fonction d'enchaînement de type parallèle.

Ces mécanismes sont étroitement liés par un graphe dans lequel les noeuds représentent les instructions et les arcs symbolisent les dépendances de données entre les instructions sur lesquels circulent les données sous forme de marques.

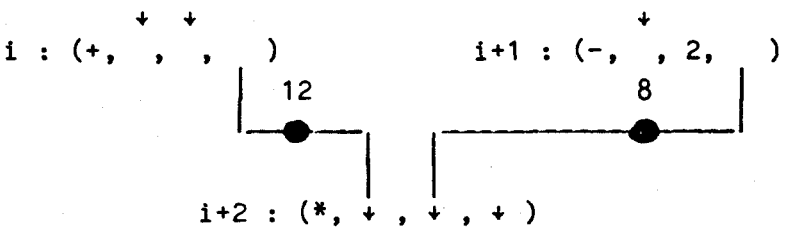
Le graphe correspondant à l'exemple précédent pourrait être à l'instant t :



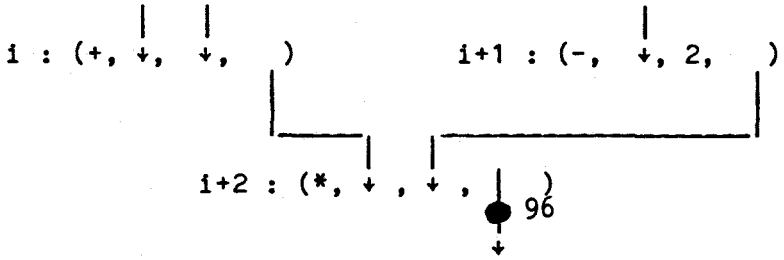
Une instruction est activée lorsque ses opérandes sont présents (i.e. tous les arcs entrants ont reçu une marque).

Une fois déclenchée, l'instruction consomme ses marques d'entrée et l'opération est exécutée. Une marque de sortie contenant le résultat est alors envoyée sur chaque arc sortant (i.e. aux instructions qui utilisent ce résultat).

A l'instant t les instructions i et i+1 peuvent être déclenchées et le graphe à l'instant t+1 serait



A son tour l'instruction i+2 est déclenchée à l'instant t+1 qui produira une marque en sortie de valeur 96.



Les organisations à flot de données se distinguent des précédentes par les points suivants :

- Les données transitent directement vers les instructions qui les utilisent.

- Le parallélisme dans la fonction d'enchaînement ne fait pas intervenir l'utilisateur (i.e. l'utilisateur n'emploie pas d'instructions spéciales pour exprimer celui-ci), seule la dépendance de données autorise celui-ci.

- Les instructions consomment les marques (i.e. données) qui ne peuvent dont être réutilisées.

- Il n'existe pas de données partagées.

- La fonction d'enchaînement et le mécanisme d'accès aux données sont identiques et supportés par le graphe appelé aussi graphe de dépendances de données (graphe "data flow").

Le fait qu'il n'y ait pas de données partagées pose un réel problème lorsqu'un programme manipule des structures de données qui ne peuvent être mises à jour que par une seule instruction. Faut-il allouer la structure prise globalement à l'instruction qui la manipule ou bien considérer les éléments de cette structure pris individuellement et n'allouer que les éléments manipulés par l'instruction ?

Les organisations à flot de données semblent mieux adaptées pour les programmes manipulant des données simples.

F. Organisation dirigée par la nécessité

Un controle par necessité (i.e. Demand-driven) peut être supporté par un mécanisme de réduction. Ces organisations de type réduction se subdivisent en deux classes :

- Les organisations type réduction de chaînes,
- et les organisations type réduction de graphes

Les premières sont caractérisées par un mécanisme d'accès aux données par valeur tandis que les secondes par un mécanisme d'accès par référence.

Les deux possèdent un mécanisme de contrôle récursif.

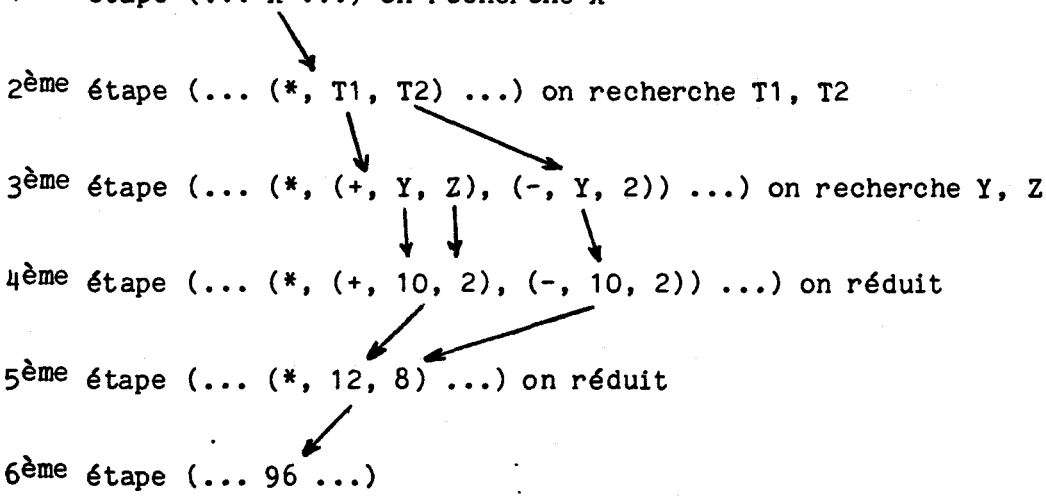
Les réductions de chaînes

Toute référence à une définition (i.e. fonction) entraîne la création d'une copie de cette définition à l'intérieur de l'instruction ou fonction qui la référence.

Notre exemple pourrait s'écrire :

```
X : (*, T1, T2)
T1 : (+, Y, Z)
T2 : (-, Y, 2)
 Y : 10
 Z : 2
```

L'évaluation de l'expression (...X...) conduirait au schéma d'exécution suivant :



Dans cet exemple on peut remarquer les différentes phases d'extension et de compression que subit l'expression que l'on cherche à évaluer, qui compliquent la localisation des sous-expressions réductibles lors d'une évaluation parallèle.

Les réductions de graphes

L'instruction qui demande la valeur associée à une définition particulière accède par référence à cette définition. Cette définition est partageable par toutes les instructions qui l'utilisent, aucune copie n'est faite comme dans le cas précédent. Notre exemple peut être décomposé comme suit :

```
Instruction     j : (... X ...)

Définition de : X  : (*, X1, X2,  )
                X1 : (+, Y, Z,  )
                X2 : (-, Y, 2,  )
                Y  : 10
                Z  : 2
```

L'exécution de J se déroulera selon les étapes suivantes :

1ère étape : L'instruction J désire connaître la valeur associée à X

X : (*, X1, X2, J/1). On mémorise dans la définition de X l'identité du demandeur.

2ème étape : L'instruction X désire X1, X2

X1 : (*, Y, Z, X/1)
X2 : (-, Y, 2, X/2)

3ème étape : X, Z sont demandés

X1 : (+, 10, 2, X/1)
X2 : (-, 10, 2, X/2)

4ème étape : X1, X2 peuvent être réduites

X(*, 12, 8, J/1)

5ème étape : X peut être réduite.

6ème étape : J : (... 96 ...)

Dès qu'une expression est réduite sa valeur vient immédiatement remplacer sa définition. Ainsi, si on demande de nouveau l'évaluation d'une expression déjà réduite on pourra renvoyer directement sa valeur.

Dans l'hypothèse où une expression en cours d'évaluation reçoit une nouvelle demande, une gestion de la liste des demandeurs (i.e. les expressions vers lesquelles on retournera la valeur) est à prévoir.

Les organisations de type réduction prélèvent leur parallélisme à l'exécution par le fait que plusieurs expressions peuvent être réduites simultanément.

La nécessité de l'obtention d'un opérande pour l'évaluation d'une expression impose le sequencement à l'exécution.

Le mécanisme d'accès aux définitions (copie pour la réduction de chaînes ou mise à jour sur place pour la réduction de graphes) semblent prédestiner les organisations de type réduction de chaînes aux calculs manipulant des opérandes simples alors que les organisations de type réduction de graphe supportent mieux les structures de données complexes.

Le tableau de la figure 2 résume les différentes relations qui existent entre les organisations présentées ici et leurs mécanismes associés.

MECANISME DE CONTROLE	MECANISMES D'ACCES AUX DONNEES : par valeur (et litérale)	par référence (et litérale)
séquentiel	S.ARCHITECTURE [TOU 79]	Type VON NEUMANN à flot de controle
parallele	DATA FLOW	Parallele à flot de controle
récursif	REDUCTION DE CHAINES	REDUCTION DE GRAPHES

Figure 2

G. Exemples de travaux

G.1. Les architectures Data flow

Ces dix dernières années ont été marquées par une volonté croissante de mettre en oeuvre des architectures de type Data flow. J.P. DENNIS fut le premier à proposer une architecture Data flow en 1974. Depuis une vingtaine de laboratoires répartis au USA, en Europe et plus récemment au Japon (5ème génération) ont développés de nouvelles architectures de ce type. Le tableau suivant donne une liste non exhaustive des laboratoires qui ont contribué au développement de machines Data flow.

- M.I.T [DEN 74]
  (DENNIS J.B)

- IRVINE [ARV 83]
  (Arvind)

- Manchester [WAT 79]
  (Gurd)

- Toulouse [COM 76]
  (Syre)

- Newcastle [TRE 82]
  (Treleaven)

- NTT Tokyo [AMA 82]
  (Amaniya)

- LILLE [LEC 79a]
  (Lecouffe)

G.2. Les architectures dirigées par la nécessité (Demand driven)

Bien que le développement d'architecture de type "Demand driven" ait suscité moins de passion que les architectures de type "Data flow", une dizaine de laboratoires répartis dans le monde ont contribués à son avancement. J. BACKUS [BAC 78] pour ses travaux sur la programmation fonctionnelle et BERKLING [BER 75] sont les premières personnes à s'être intéressées à ce nouveau mode d'organisation. Le tableau suivant donne une liste non exhaustive des laboratoires qui ont participé à son développement pour ces dix dernières années :

- NEWCASTLE [TRE 80]
  (Treleauen)

- GMD [BER 75]
  (Berkling)

- North Carolina [MAG 80]
  (Mago)

- CAMBRIDGE [CLA 80]
  (Norman)

- UTAH AMPS [KEL 78]
  (Keller)

CONCLUSION

Le chapitre que nous venons de présenter a permis de mettre en évidence plusieurs points fondamentaux :

* Nécessité d'accroître les performances d'un système par l'exploitation systématique du parallélisme inhérent à un programme afin de répondre aux besoins de certains domaines d'application (Intelligence Artificielle, etc...).

* Nécessité de rompre avec un ou plusieurs des concepts de la machine Von Neumann de manière à répondre au premier point.

* Les architectures à contrôle par disponibilité ou par nécessité autorisent une exploitation optimum du parallélisme implicite contenu dans un programme.

L'architecture de la machine décrite dans les chapitres suivants est du type dirigée par les données. Une exploitation de l'assignation unique permet de déduire les dépendances de données.

Ses aspects originaux par rapport aux travaux existants résident dans plusieurs caractéristiques.

* Le mécanisme dirigé par les données est appliqué non pas au niveau d'une instruction mais au niveau d'un groupe d'instructions qui s'exécutent de manière conventionnelle.

* L'assignation unique est respectée au niveau des objets de sortie calculés par un groupe d'instructions.

* L'adjonction de primitives permettant de donner un caractère dynamique à l'exécution.

* Mise en oeuvre d'un outil de communication résolvant d'une manière triviale les problèmes de synchronisation.

CHAPITRE II

Le projet MAUD

Introduction

A. Présentation du modèle MAUD

A.1 Un data flow macroscopique

A.2 Assignation unique au niveau des blocs

A.3 Communication entre blocs

A.4 Structure d'un bloc

A.5 Primitives du modèle

A.6 Les états d'un bloc MAUD

A.7 Architecture du modèle

B. Implémentation de l'architecture de MAUD

B.1 Anneau de mémoire circulante

B.2 Module de mémoire circulante

B.3 Accès à la mémoire circulante

B.4 Gestion de la mémoire circulante

B.5 Unité fonctionnelle

B.6 Unité d'échange

B.7 Le processeur Pupitre

Conclusion

Introduction

Ce chapitre rappelle les notions de base qui ont été introduites depuis le démarrage du projet MAUD (Machine à Assignation Unique Dynamique). Ces travaux ont fait l'objet de deux thèses [LEC 79a] [PET 81] auxquelles le lecteur pourra se référer pour de plus amples informations. Le lecteur familier au projet MAUD pourra laisser de côté ce chapitre.

La première partie de ce chapitre présente brièvement le modèle de l'architecture MAUD tel qu'il a été défini dans la thèse de Madame LECOUFFE [LEC 79a].

La seconde partie décrit l'implémentation qui a été choisie pour la réalisation d'un prototype [PET 81].

A. Présentation du modèle MAUD

A.1 Un "data flow" macroscopique

MAUD est une machine dirigée par les données dont le contrôle "data flow" est appliqué à un niveau macroscopique. A ce niveau macroscopique les objets manipulés par MAUD ne sont plus des instructions mais des blocs d'instructions. Un bloc est constitué d'un ensemble d'instructions à exécuter de manière séquentielle et des objets utilisés par ces instructions.

A.2 Assignation unique au niveau des blocs [LEC 79b]

Un programme pour MAUD est constitué d'un ensemble de blocs qui ont la possibilité de communiquer entre eux par l'intermédiaire d'un nombre fini d'objets :

- Les objets d'entrée: ce sont les objets nécessaires au lancement de l'exécution du bloc

- Les objets de sortie: ils sont produits par l'exécution d'un bloc et communiqués à d'autres blocs qui les utilisent comme objets d'entrée

Le concept d'assignation unique, introduit par TESLER et ENEA [TES 68], est appliqué en ce qui nous concerne au niveau des blocs : "un objet de sortie ne pourra recevoir plus d'une valeur pendant toute la durée d'exécution du programme".

Cette contrainte que l'on retrouve dans tous les langages fonctionnels présente des propriétés intéressantes car elle permet d'exprimer de manière naturelle :

- Les dépendances de données qui existent entre les blocs d'un même programme et par delà la détection du parallélisme interbloc

- Le contrôle dirigé par les données, en ce sens qu'on peut lancer l'exécution d'un bloc dès que tous ses objets d'entrée ont été calculés.

A.3 Communication entre blocs

La communication entre les blocs se fait au travers des objets d'entrée et de sortie. A chaque objet referencé dans un programme MAUD est associé un nom interne qui permet de l'identifier à l'intérieur d'un bloc. Ce nom interne (NI) est local au bloc et peut différer d'un bloc à l'autre. Pour assurer une cohérence entre les objets envoyés vers l'extérieur et les objets reçus de l'extérieur on leur associe un nom de communication* (NC) qui est global à l'ensemble des blocs du programme. Ce nom sert uniquement pour la communication interbloc, il est donc inconnu à l'intérieur du bloc. La figure 1 illustre ce mécanisme de communication.

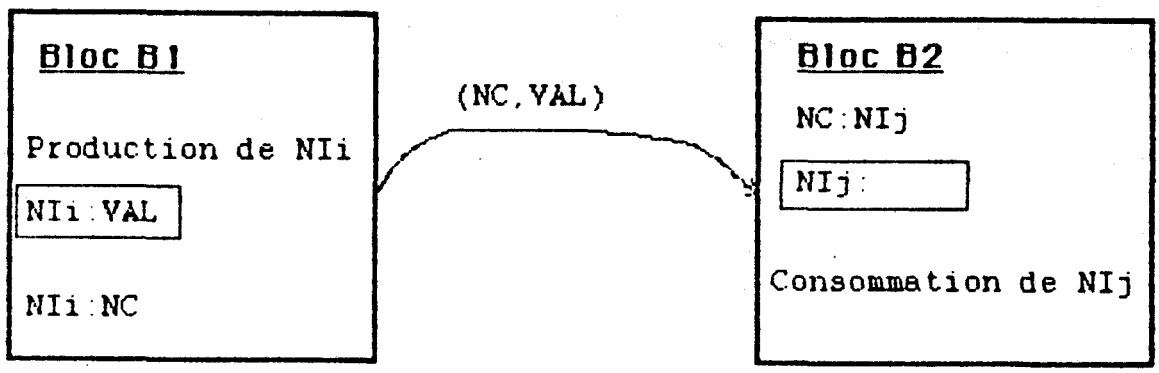


Fig. 1 Communication interbloc
Le bloc B1 produit un objet de valeur VAL qu'il référence NIi
Le bloc B2 consomme cet objet référencé sous le nom NIj

Note* : Les noms de communication sont en nombre limité dans la réalisation du prototype. Une fois que ceux-ci sont consommés, ils sont récupérés de manière à les réutiliser. La gestion des noms de communication devra être assurée par un processeur ou processus spécifique. La tache de celui-ci consistera à fournir aux processus demandeurs les noms de communication qu'ils réclament. A chaque nom de communication est associé un objet qui devra respecter la règle d'assignation unique. Cette gestion n'est pas étudiée dans cette thèse.

A.4 Structure d'un bloc

La structure d'un bloc pour MAUD qui a été définie est présentée ci-dessous. Les objets manipulés par un bloc ont été scindés en plusieurs domaines :

- Domaine d'entrée (D.E) : où se trouve l'ensemble des couples (NC, VAL) ou (NC, non évalué) associés aux objets d'entrée de ce bloc. NC et VAL représenteront pour la suite de cette thèse respectivement un nom de communication et une valeur associée à un nom de communication; Un bloc ne peut s'exécuter que si aucun couple n'est de type (NC, non évalué).

- Domaine de sortie (D.S) : constitué des couples (NC, VAL) ou (NC, non évalué) relatifs aux objets de sortie de ce bloc. En fin d'exécution du bloc, il ne comporte plus de couple de type (NC, non évalué).

- Domaine propre (D.P) : qui contient les couples (NI, VAL) associés aux objets internes de ce bloc. NI représente le nom interne sous lequel est connu l'objet à l'intérieur du bloc.

Un bloc pour MAUD peut être vu comme un assemblage de 3 parties indissociables :

- Une partie interne formée du domaine propre et d'un ensemble d'instructions qui utilise les objets de ce domaine propre. Ces instructions sont à exécuter de manière conventionnelle, c'est-à-dire qu'elles appartiennent à un programme en langage machine de type VON NEUMANN.

- Une partie communication qui comprend le domaine d'entrée et le domaine de sortie, définis plus haut, nécessaires à la communication interbloc.

- Enfin une partie interface qui permet la transmission des valeurs entre la partie communication et la partie interne. Cette interface est constituée de deux listes de correspondance nommées respectivement liste de correspondance d'entrée (notée LCE) et liste de correspondance de sortie (notée LCS). Ces listes sont constituées d'un ensemble de couples (NC, NI). Un couple de la liste LCE représente ce que nous appelerons par la suite une relation de consommation d'un objet de nom de communication NC, NI étant le nom interne sous lequel cet objet sera désigné dans ce bloc. Initialement, elle associe aux noms de communication du domaine d'entrée les noms internes de ces objets.
- Un couple de la liste LCS représente une relation de production d'un objet de nom interne NI, NC étant le nom de communication sous lequel cet objet sera désigné à l'extérieur de ce bloc. Initialement elle associe aux noms de communication du domaine de sortie les noms internes de ces objets. Ces listes peuvent être modifiées en cours d'exécution du bloc par les diverses primitives que nous allons décrire.

La figure 2 représente la structure interne d'un bloc pour MAUD.

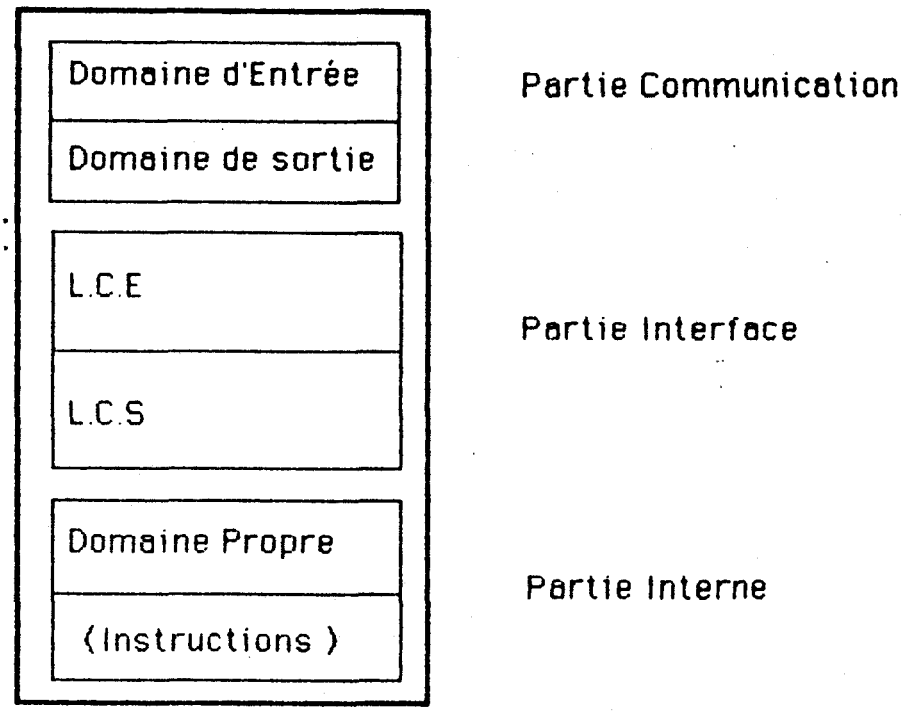


Fig. 2 : Structure d'un bloc pour MAUD

A .5 Les primitives du modèle MAUD

Il a été défini un jeu de primitives nécessaires à la prise en compte de la structure des blocs MAUD. Ce sont :

Primitive début de bloc : qui assure le transfert des valeurs (VALi) entre la partie communication (domaine propre) au travers de la partie interface (liste de correspondance d'entrée). Cette primitive est la première instruction que devra exécuter un Processeur d'Exécution lorsqu'il prend en charge un nouveau bloc de l'ensemble X. A la suite de cette primitive, le domaine d'entrée du bloc et la liste LCE sont vides. Les noms de communications NC consommés peuvent être alors restitués au fournisseur de noms de communication.

Primitive fin de bloc : elle permet la transmission de valeurs de la partie interne du bloc (domaine propre) vers la partie communication (domaine de sortie) au moyen de la partie interface (liste de correspondance de sortie). Elle correspond de ce fait à la dernière primitive que le Processeur d'Exécution exécutera afin de produire un domaine de sortie contenant les objets calculés par ce bloc. A l'issue de cette primitive la liste LCS ne comporte plus aucun couple (NC, NI).

Les actions de ces primitives sont montrées figure 3. Un bloc exécutable se présente donc de la manière :

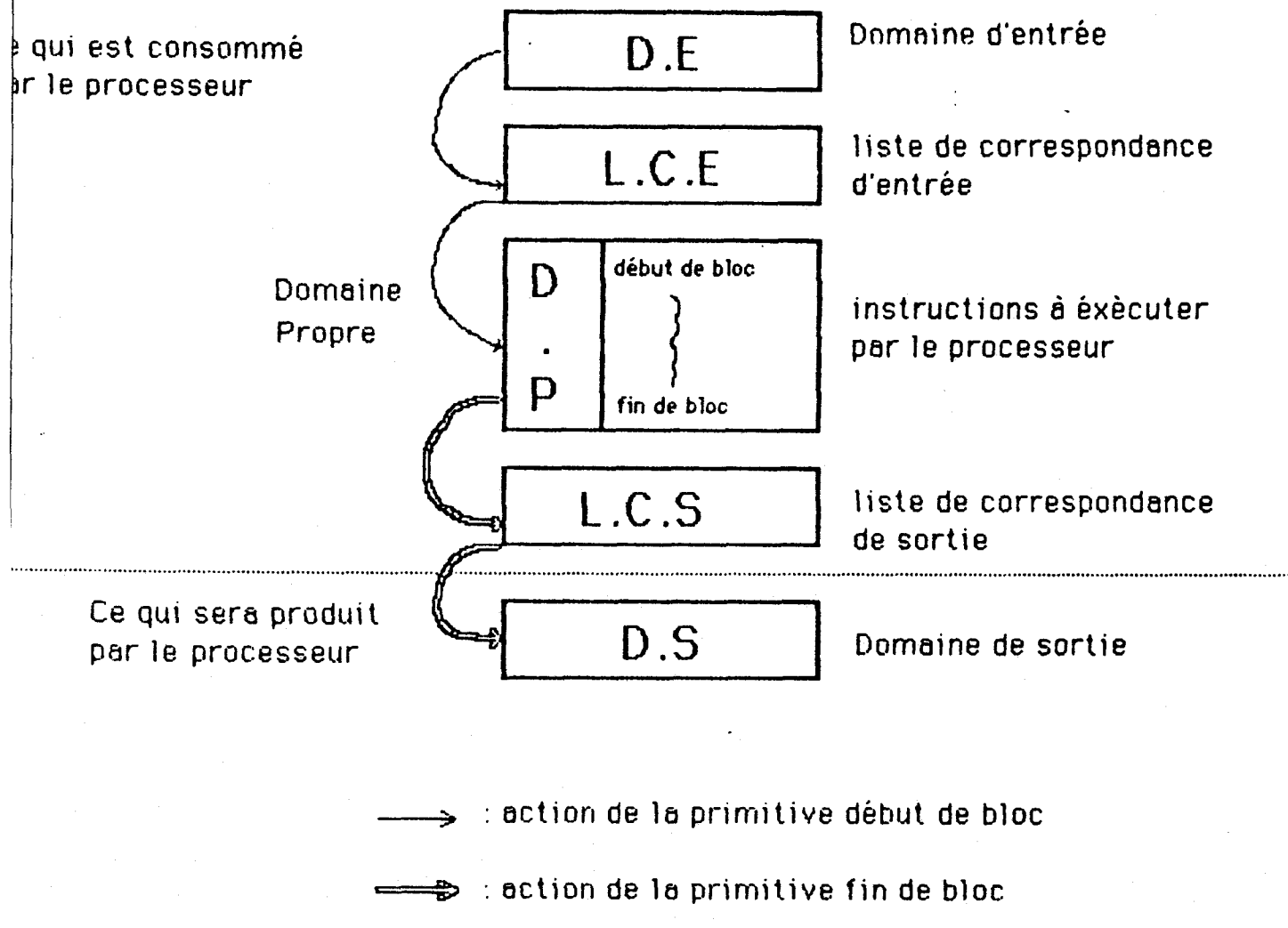


Fig. 3 Transmission des valeurs

Deux autres primitives ont été également définies pour donner au modèle MAUD un caractère dynamique qui va lui permettre d'offrir davantage de parallélisme. Il s'agit de :

La primitive EXECBLOC (nom de bloc, objets d'entrée, objets de sortie)

Elle permet de créer un nouveau bloc MAUD à partir d'un modèle se trouvant dans une bibliothèque (résidante sur disque) et connu sous le nom de bloc passé en paramétre. Cette primitive devra également indiquer les objets d'entrée utilisés par ce bloc et les objets de sortie que l'on souhaite éventuellement voir revenir au bloc qui exécute cette primitive.

Les objets d'entrée permettront la constitution du D.E de l'appelé. Un tel objet peut être :

- soit un objet dont la valeur est connue de l'appelant
- soit un objet qui sera produit ultérieurement par le bloc appelant ou par un autre bloc issu du bloc appelant. Si l'objet NI intervient déjà dans une relation de consommation (NC, NI) LCE de l'appelant, cette relation est retirée de la LCE. Dans le cas ou l'objet NI n'intervient pas dans une relation de consommation, une relation de production (NC, NI) est crée et ajoutée à la liste LCS de l'appelant. Le nom de communication est dans les deux cas transmis à l'appelé pour constituer son domaine d'entrée.

Les objets de sortie permettront de constituer le D.S de l'appelé. Si un tel objet NI intervient déjà dans une relation de production (NC, NI) ε LCS de l'appelant, cette relation est alors retirée de la LCS, sinon une relation de consommation (NC, NI) est crée et ajoutée à la LCE. Le nom de communication est dans les deux cas transmis à l'appelé.

- La primitive ATTENDRE {objet ; }$^+$

Cette primitive trouve sa justification lorsqu'un bloc Bo qui a exécuté une primitive EXECBLOC (B1 ; ;) a besoin de consommer des objets produits par ce bloc B1. Les paramètres de cette primitive sont les noms de communications associés aux objets à consommer. Les relations de consommation (NC, NI) de ces objets sont recherchées dans la LCE pour constituer un nouveau domaine d'entrée du bloc, et le bloc est momentanément suspendu. Le bloc appartient alors à l'ensemble A. Lorsque son domaine d'entrée sera complet, son exécution sera reprise avec l'appel d'une primitive début de bloc permettant le transfert des valeurs depuis le domaine d'entrée et le domaine propre et la suppression des relations de consommation utilisées de la LCE.

Pour illustrer l'utilisation de ces primitives considérons l'exemple d'un programme qui calcule l'intégrale d'une courbe gaussienne f pour la méthode des trapèzes (fig. 4) sur un intervalle a, b, divisé en M intervalles égaux.

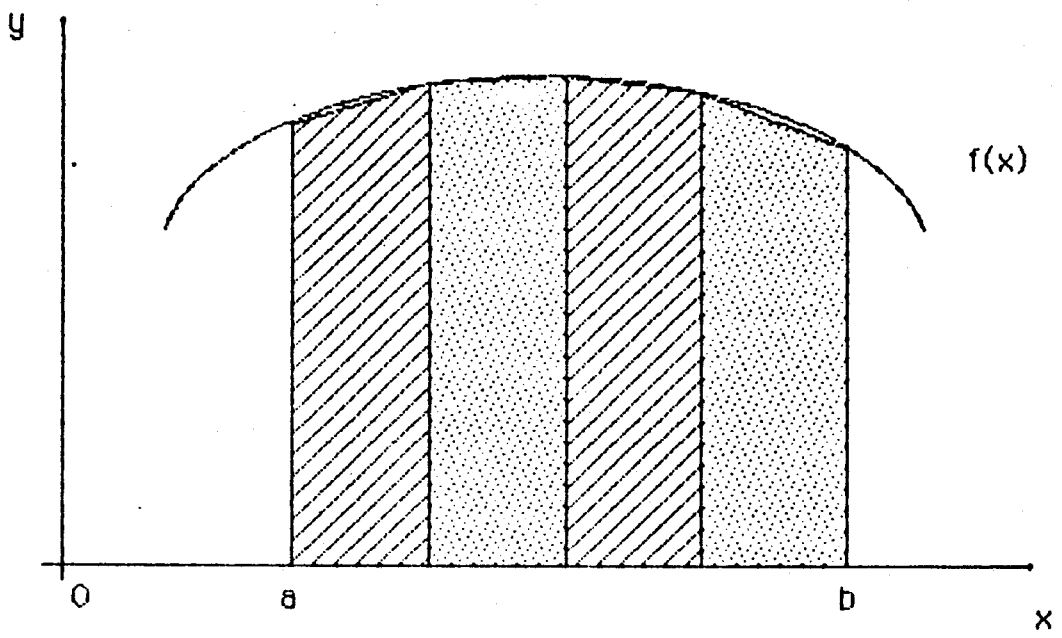


Fig. 4

Le calcul de l'intégrale I par cette méthode conduit à l'approximation de I par la formule suivante

$$I = \Delta x \left[f(a) + f(b) + \sum_{i=1}^{n-1} f(xi) \right]$$

avec $\Delta x = b-a/n$ et $x_i = a + [i*\Delta x]$

L'écriture d'un programme MAUD qui évaluerait I pourrait être le suivant au moyen de deux blocs MAUD :

```
Bloc INTEGRALE : entrée   f, n, a, b ; sortie I
     locale : J, Δx
     début bloc
      Δx : = (b-a)/n
      Pour J = 0 à n
        faire EXECBLOC (AIRE ; f, Δx, a, J ; Aj) fait
      ATTENDRE [Aj pour o≤ J ≤ n ]
                          i=n-1
      I : = (1/2 {Ao + An} + ∑  Ai)  Δ x
                          i=1
     fin bloc
```

```
Bloc AIRE : entrée f, Δ x, a, j ; sortie Aj
     locale : x
     début bloc
      x : = a + (j * Δ x)
      Aj : = f(x)
     Fin bloc
```

A.6 Les états d'un bloc MAUD

En fonction du nombre d'objets déjà évalués de son domaine d'entrée un bloc MAUD passe par une suite d'états qui le caractérise de manière univoque. Nous parlerons dans ce qui suit de bloc exécutable ou de bloc en attente au lieu de bloc MAUD.
Un bloc est dit exécutable si l'ensemble des objets de son domaine d'entrée ont reçu une valeur. On parle alors de domaine d'entrée complet.

Un bloc est dit en attente s'il ne posséde pas un domaine d'entrée complet. Une ou plusieurs relations de consommation lui font défaut.

A.7 Architecture du modèle

Le modèle MAUD nécessite l'existence de plusieurs unités fonctionnelles et de plusieurs unités mémoires spécifiques.
L'architecture du modèle est représenté figure 5, elle comprend :

* Unités fonctionnelles

-{PEI} : ensemble de processeurs élémentaires ou Processeurs d'Exécution destinés à exécuter les blocs exécutables de l'ensemble X.

-M à j : processeur spécifique Mise à Jour dont la fonction est double. D'une part assurer la transmission des objets de sortie produits (S) vers les blocs en attente de ces objets (A) et d'autre part détecter si le domaine d'entrée d'un bloc en attente devient complet.
La fonction de ce processeur est détaillée au chapitre III de cette thèse.

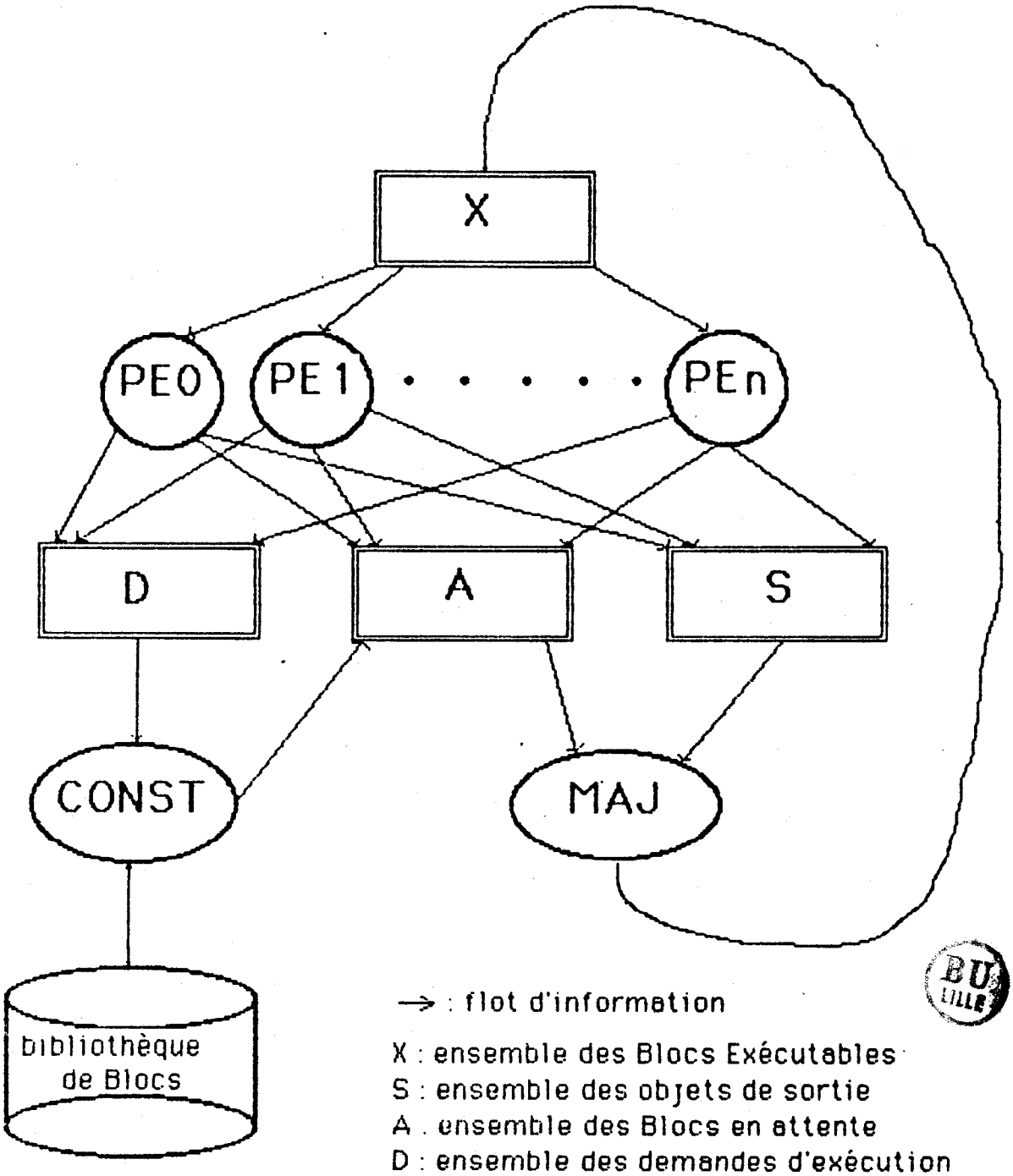


Fig. 5 Modèle Dynamique de MAUD

-Const : Processeur spécifique Constructeur dont le rôle est de construire un bloc exécutable ou en attente à partir de chacune des demandes d'exécution (i.e primitives EXECBLOC) formulées par les Processeurs d'Exécution. Il posséde pour cela une bibliothèque de blocs qui contient les modèles des blocs à construire.

* Unités mémoires

X : Contient l'ensemble des blocs exécutables. Chacun d'entre eux est pris en charge dès qu'un Processeur d'Exécution devient libre.

A : Représente l'ensemble des blocs en attente. Ces blocs seront réinjectés, par le processeur MAJ, dans l'unité mémoire dès que leurs domaines d'entrée deviendront complets.

S : Cet ensemble recueille les objets de sortie produits par les Processeurs d'Exécution lorsqu'ils terminent l'exécution d'un bloc (i.e primitive FIN BLOC).

D : Cette unité mémoire renferme les demandes d'exécution (i.e primitive EXECBLOC) formulées par les Processeurs d'Exécution.

Note : - Pour qu'un programme MAUD puisse s'exécuter il faut qu'il existe au moins un bloc dans l'unité mémoire X. Par le biais des primitives EXEBLOC celui-ci pourra lancer l'exécution d'autres blocs.

- Le parallélisme résulte de l'existence de plusieurs processeurs d'exécution capables de fonctionner simultanément de façon autonome et de la structure pipeline du modèle.

B. Implémentation de l'architecture de MAUD

Les objectifs qui ont guidé la définition d'une implémentation adaptée à l'architecture de MAUD, étaient les suivants :

* l'emploi de composants VLSI classiques (microprocesseurs standards) relativement peu coûteux, et bien supportés au niveau des outils de développement

* l'extensibilité du système dont on ne veut pas figer les performances par une limitation du nombre de processeurs de traitement.

* la gestion de la mémoire devra être facile, il faut se prémunir facilement des problèmes de synchronisation inhérents aux architectures parallèles.

* la gestion des sections critiques que sont les ensembles A, X, S, D du modèle dynamique ne doit pas entraîner une "surcharge" trop importante du système.

B.1 L'anneau de mémoire circulante

Parmi les différents types d'organisation de mémoire désormais classiques (mémoire commune à accés par bus unique, réseau cross-bar, réseau d'interconnexion etc...) il a été choisi une organisation construite autour d'un anneau de mémoire circulante (fig. 6). Une telle organisation a déjà été choisie pour implémenter la machine G.C.F (Generalized Central Flow) de l'Université de Newcastle [TREL 78], la machine DDP de Texas Instrument [COOT 79] toutes deux de type dirigées par les données et une machine de type réduction [TREL 80].
L'utilisation d'une mémoire circulante s'est justifiée par les avantages suivants :

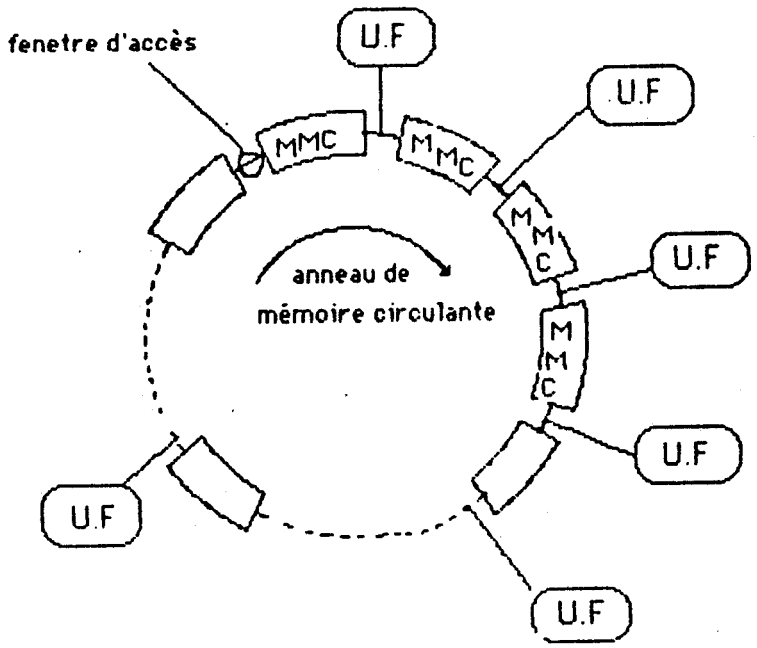
* l'accés à cette mémoire commune est très simple puisque chaque processeur dispose d'une fenêtre d'accés sur celle-ci qui lui est propre et physiquement indépendante des autres fenêtres. La simultanéité des accés à cette mémoire est donc possible simplement.

* La circulation permanente de l'information devant les fenêtres d'accés permet d'implémenter des fonctions d'accés associatives de manière triviale.

* Lorsque la mémoire circulante est utilisée, comme c'est le cas ici, dans une organisation en anneau elle présente les caractéristiques d'un outil de stockage d'information.

* Son mécanisme permanent de circulation d'information en fait un bon outil de communication interprocesseur.

* Extensibilité simple



U.F. Unité fonctionnelle de MAUD

MMC. Module de Mémoire Circulante

Fig. 6. L'anneau de Mémoire Circulante

L'anneau de mémoire circulante permet de recevoir les objets des ensembles X, S, A, D introduits précédemment.

B.2 Module de mémoire circulante

Les modules de mémoire circulante ont été réalisés à partir de cellules de registres à décalage de 1 k bits (2533 signetics) de technologie MOS.
La mémoire circulante comporte à l'heure actuelle seize modules d'une capacité de 1k mots de 16 bits chacun. L'horloge de base a une fréquence de 1 Mhz ce qui confère à la mémoire un débit d'information de 16 M bits/s.
Une cellule de registres à décalage se présente comme un ensemble de trois parties distinctes [CORD 78] (fig.7) :

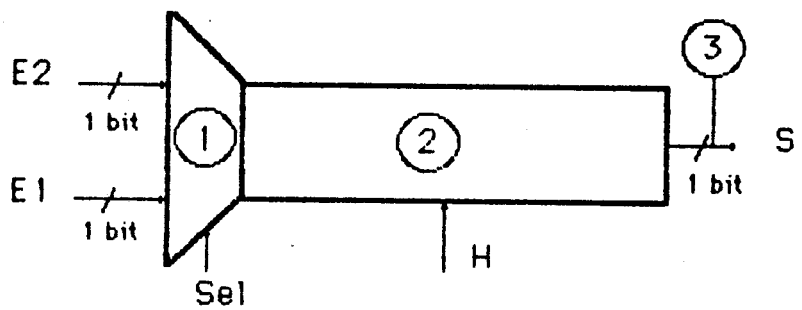


Fig. 7 Une cellule mémoire circulante

1- Deux voies d'entrée E_1, E_2 multiplexées par un signal de sélection (SEL) où l'information présente sur la voie sélectionnée est introduite en séquence (au rythme de l'horloge H) dans le milieu de propagation.

2- Un milieu de propagation dans lequel se propage l'information issue de l'entrée sélectionnée au rythme de l'horloge

3- Une voie de sortie S où l'on récupère les informations après un parcours complet le long du milieu de propagation.

Un module de la mémoire circulaire est l'association parallèle de 16 cellules de registres à décalage. Ce module correspondant en quelque sorte à la notion de secteur physique que l'on rencontre sur les disques magnétiques, et que nous appelerons par la suite "cadre". La capacité d'un cadre est donc de 1 kmots de 16 bits (fig. 8).

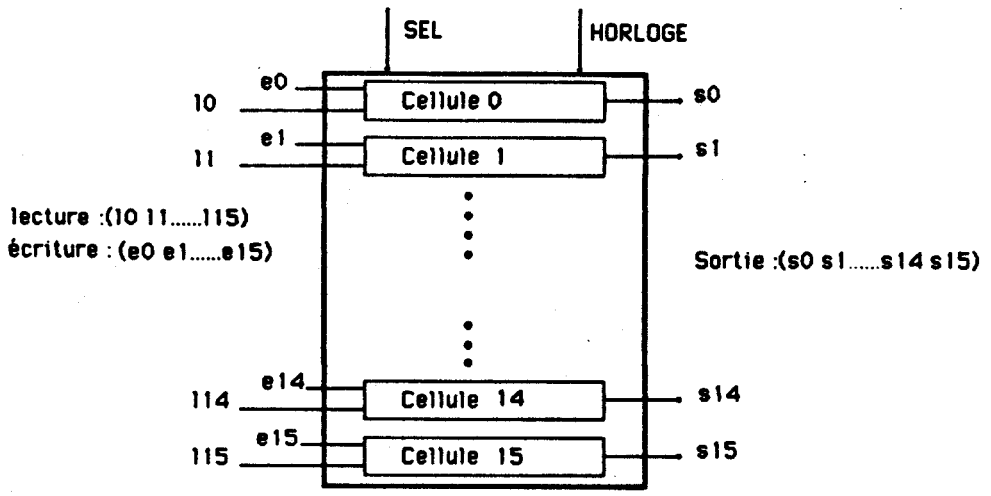


Fig. 8 Un cadre de MAUD

B.3 Accés à la mémoire circulante

Les modules ou cadres de mémoire circulante ont été implémentés de manière à réaliser deux types d'accés sur ceux-ci :

-1 Accés par capture : Le module est isolé du reste de la mémoire circulante par l'intermédiaire d'un multiplexeur placé en sorte qui permet de court-circuiter celui-ci (fig. 9). Un dispositif externe peut alors exploiter le contenu du cadre au rythme d'une horloge locale. La lecture se fait par consultation de la sortie S, l'opération d'écriture se fait par sélection de l'entrée E_2.

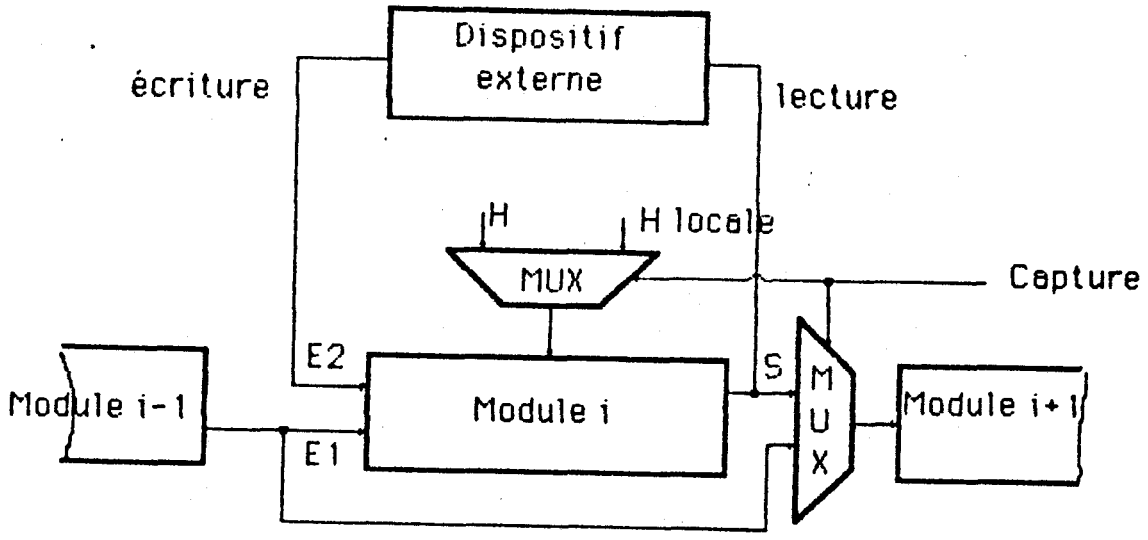


Fig. 9 Accés par capture

-2 Accés à la volée : Cet accés s'effectue en temps réel (i.e à la fréquence de l'horloge H de la mémoire circulante). Les opérations d'écriture se font par l'intermédiaire de l'entrée E_2 tandis que les opérations de lecture se font par consultation au vol de l'information présente sur l'entrée E_1. Ce mode d'accés autorise un troisième type d'opération appelée opération d'échange (lecture simultanée à une écriture). L'opération d'échange est rendue possible par le fait que l'information qui transite d'une cellule vers la cellule voisine est présente bien avant sa mémorisation effective dans la cellule voisine. Le chronogramme de la figure 10 permet de se rendre compte de cette possibilité.

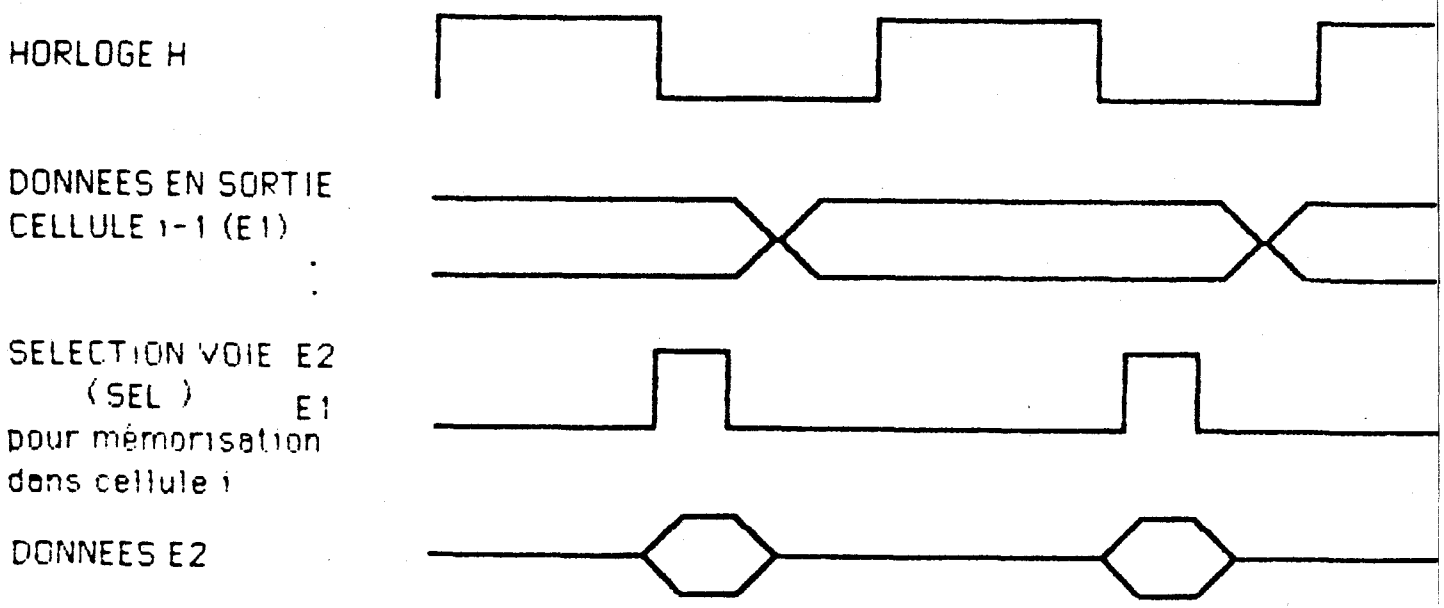


Fig. 10 Lecture de E_1 suivie de l'écriture de E_2

B.4 Gestion de la mémoire circulante

Comme nous l'avons indiqué précédemment, nous avons opté pour une gestion simple de la mémoire circulante, c'est-à-dire que celle-ci sera découpée en secteurs logiques de 1 k -16 bits appelé "cadre logique". Chaque cadre est susceptible de recevoir un seul des objets définis dans le modèle théorique (bloc exécutable, bloc en attente etc...) De façon analogue à un disque, ces cadres (ou secteurs) sont étiquetés par une entête qui renseigne sur le type du bloc qui y réside. Le type d'un cadre n'est pas quelque chose de fixe mais pourra évoluer selon les besoins du système.

B.5 Unité fonctionnelle

L'utilisation de microprocesseurs standards pour l'implémentation des processeurs fonctionnels de MAUD, nous a contraint à définir une interface [PET 81] ou Unité d'échange entre le processeur et la mémoire circulante afin de substituer l'accés aléatoire du microprocesseur à l'accés purement séquentiel de la mémoire circulante. Cette unité d'échange a été conçue de manière à gérer automatiquement le transfert d'informations entre la mémoire circulante et la mémoire locale du processeur et vice versa (fig. 11).

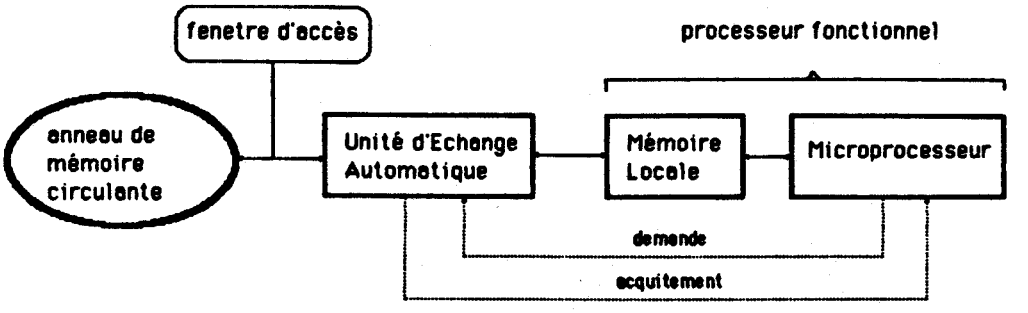


Fig. 11 Poste de travail

B.6 Unité d'échange

Pour des raisons de modularité, il a été défini une unité d'échange générale dans laquelle la structure microprogrammée permet d'affiner le rôle de celle-ci en fonction du processeur auquel elle est rattachée (Processeur d'Exécution, processeur Constructeur etc...). La fonction de cette unité peut se définir de la manière suivante :
assurer le contrôle des échanges entre la mémoire circulante et une mémoire adressable locale.

Trois types d'actions 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. Cette opération ne détruit pas le contenu de l'anneau.

- opération d'écriture : un mot est transféré de la mémoire locale vers l'anneau. Le contenu de la mémoire locale reste inchangé.

- opération de mise à jour : c'est une opération de lecture suivie d'une opération d'écriture au cours d'une seule période d'horloge.

B.7 Le processeur Pupitre [PET 81]

Ce processeur, bien que n'apparaissant pas dans la définition de MAUD, a été rendu nécessaire par les besoins suivants :

- nécessité de disposer d'un outil capable de vérifier le fonctionnement de la mémoire circulante au niveau des échanges entre les processeurs et l'anneau.

- possibilité d'initialiser le système et de charger un programme à exécuter.

L'architecture du processeur est montrée figure 12, elle comprend :

- une unité centrale 8085

- une mémoire morte de 2k octects contenant le programme de fonctionnement du processeur

- une mémoire vive de 4k octects dont 2k octects sont accessibles par l'unité d'échange en mutuelle exclusion avec le microprocesseur

- un clavier écran permet de dialoguer avec le logiciel résident du microprocesseur.

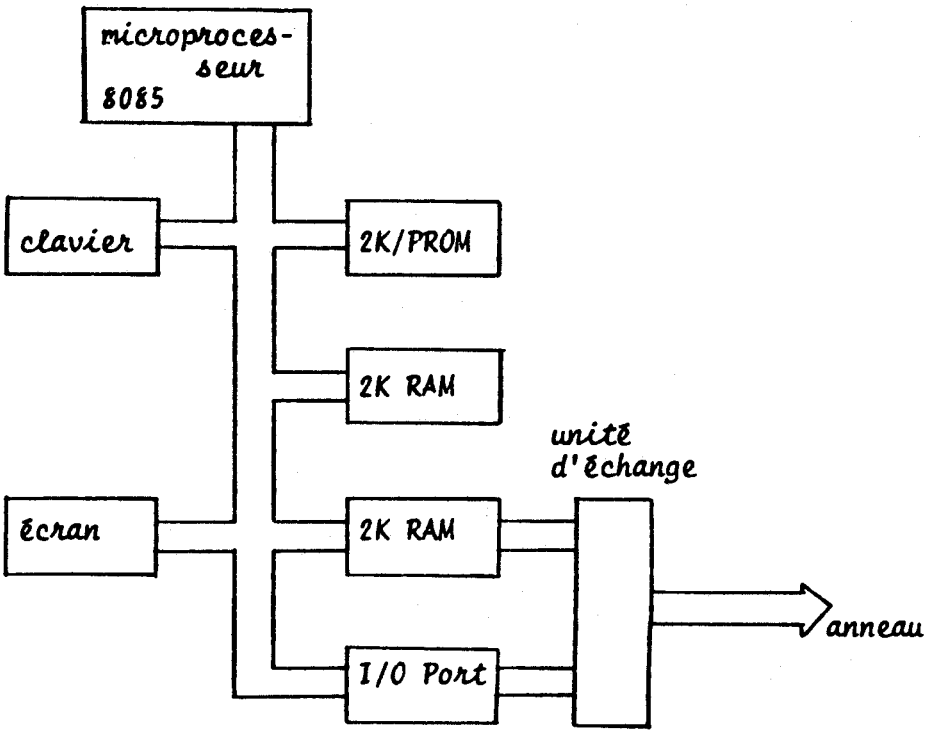


Fig. 12 Processeur Pupitre

CONCLUSION

Nous avons rappelé dans ce chapitre tous les constituants nécessaires à la compréhension des chapitres suivants.

Ce chapitre a présenté en outre l'état du prototype en cours de réalisation lorsque cette thèse fut entreprise.

CHAPITRE III

Processeur d'Exécution

Introduction

1. Aspects fonctionnels du Processeur d'Exécution

   1.a Rôle
   1.b Etats d'un Processeur d'Exécution
   1.c Les communications entre le Processeur d'Exécution et l'anneau

2. Architecture du Processeur d'Exécution

   2.a L'unité de traitement
   2.b L'unité d'échange automatique

Conclusion

Introduction

Ce chapitre décrit dans une première partie les aspects fonctionnels du Processeur d'Exécution ainsi que les communications qu'il amène au niveau de l'anneau de mémoire circulante.

La seconde partie présente l'architecture générale d'un Processeur d'Exécution. Les implémentations des différentes unités qui le composent y sont décrites.

- Le Processeur d'Exécution -

I- ASPECTS FONCTIONNELS DU PROCESSEUR D'EXECUTION

I-a Rôle

Le Processeur d'Exécution a pour fonction la prise en charge des blocs exécutables de l'ensemble X défini dans le modèle théorique de MAUD. Pour cela il devra prélever de la mémoire circulante un bloc exécutable dès que celui-ci se présentera sous la fenêtre d'accés à laquelle il est rattaché.
En plus des instructions standards qui constituent la partie opérative proprement dite d'un bloc exécutable, il devra être capable de reconnaître et d'exécuter les primitives définies dans le modèle théorique, c'est-à-dire :

- primitive début de bloc
- primitive fin de bloc
- primitive EXECBLOC
- primitive ATTENDRE

* Primitive début de bloc

La primitive début de bloc est la première instruction que doit exécuter un Processeur d'Exécution après avoir extrait de l'anneau de mémoire circulante un bloc exécutable.

Cette primitive a pour rôle de transmettre les valeurs qui se trouvent dans le domaine d'entrée vers le domaine propre du bloc exécutable. Cette transmission s'effectue au moyen de la liste de correspondance d'entrée qui associe, à chaque nom de communication des objets d'entrée, un nom interne du domaine propre. La valeur ainsi transmise ne peut-être réutilisée et le nom de communication est restitué au fournisseur des noms de communication.

* Primitive Fin de bloc

La primitive Fin de bloc est l'instruction qu'exécute un Processeur d'Exécution lorsqu'il termine le traitement d'un bloc exécutable. Cette primitive effectue la transmission des objets de sortie de la partie interne du bloc vers la partie communication au moyen de la liste de correspondance de sortie. Elle a pour finalité de construire un bloc de type "domaine de sortie", destiné au processeur Mise à Jour, et qui sera inséré dans la mémoire circulante dès que la configuration de l'anneau le permet.

* Primitive EXECBLOC

En cours de traitement d'un bloc exécutable un processeur peut être amené à demander l'exécution en parallèle d'un bloc dont un modèle figure dans une bibliothèque de blocs. Cette primitive consiste à construire un bloc de type "demande d'exécution" qui sera envoyé au processeur Construction par le biais de la mémoire circulante. Ce bloc devra fournir, outre le nom du bloc demandé, les informations pour permettre la construction de son domaine d'entrée ainsi que celle de son domaine de sortie. Le fournisseur des noms de communication devra fournir les noms de communication nécessaires à la prise en compte de cette primitive

* Primitive ATTENDRE

La primitive ATTENDRE a pour effet de suspendre l'exécution d'un bloc jusqu'à ce que les objets précisés dans la liste des paramétres de cette primitive aient été évalués par d'autres blocs. Les objets attendus ne pouvant être que des objets apparaissant dans le domaine de sortie d'un bloc appelé par ce bloc ou récursivement d'un bloc appelé par le bloc appelé par ce bloc, c'est-à-dire qu'il doit exister une relation de consommation (NC, NI) appartenant à la liste L.C.E. du bloc. L'algorithme consiste à recréer un nouveau domaine d'entrée pour le bloc qui exécute cette primitive et dans lequel on fait apparaître les noms de communication qui ont été associés aux noms internes des objets attendus. Une fois le domaine d'entrée reconstruit, le processeur réinjecte dans l'anneau de mémoire circulante le bloc avec le type "bloc en attente".

1.b Etat d'un Processeur d'Exécution

Nous avons vu précédemment qu'un processeur peut être amené à effectuer des échanges avec la mémoire circulante. Ces demandes d'échanges avec l'anneau se résument comme suit :

1- capture de bloc exécutable (bloc BEX)
2- dépôt de demande d'exécution (bloc DX)
3- dépôt de domaine de sortie (bloc DS)
4- dépôt de bloc en attente (bloc BEA)

Pour éviter de bloquer systématiquement le processeur dans le cas 2, lorsqu'il n'y a pas de place dans l'anneau, on introduit une file d'attente de ces demandes (file DX). Par contre pour les cas 3 et 4 le bloc en cours ne peut être poursuivi, une telle file d'attente est donc inutile. On dira qu'un processeur a besoin de faire un transfert impératif lorsqu'il exécute l'une des primitives suivantes :

- EXECBLOC avec file DX pleine
- ATTENDRE
- FIN de bloc.

De façon usuelle, un Processeur d'Exécution est caractérisé à un instant t par son état : actif, en attente, ou inactif. La définition des états d'un processeur est alors la suivante :

* état actif : un processeur est dit actif lorsqu'il est en train d'exécuter un bloc exécutable sans avoir besoin d'effectuer un transfert impératif.

* état attente : on parle de processeur en attente, lorsqu'un processeur a besoin d'effectuer un transfert impératif avec la mémoire circulante.

* état inactif : un processeur est inactif lorsqu'il ne satisfait pas la définition de l'état actif ou attente, c'est-à-dire qu'il n'est pas en train d'exécuter un bloc exécutable et qu'il n'a pas de transfert impératif.

1.c Les communications entre le Processeur d'Exécution et l'anneau

En fonction de l'état dans lequel se trouve un Processeur d'Exécution, il sera amené à formuler des demandes de transfert d'information avec l'anneau de mémoire circulante. Par convention, nous définirons par la suite un transfert de type.

* dépôt : un processeur qui insère un bloc dans la mémoire circulante

* capture : un processeur qui prélève, de la mémoire circulante, un bloc

* échange : un processeur qui réalise de manière simultanée un dépôt et une capture.

Le graphe de la figure 1 représente l'automate des états du Processeur d'Exécution en fonction du type d'opération que celui-ci exécute. Les opérations que nous considérons ici, sont uniquement celles qui influent sur l'état du processeur. Ces opérations sont :

- les primitives EXECBLOC, FIN BLOC, ATTENDRE
- les transferts Dépôt, Capture, Echange

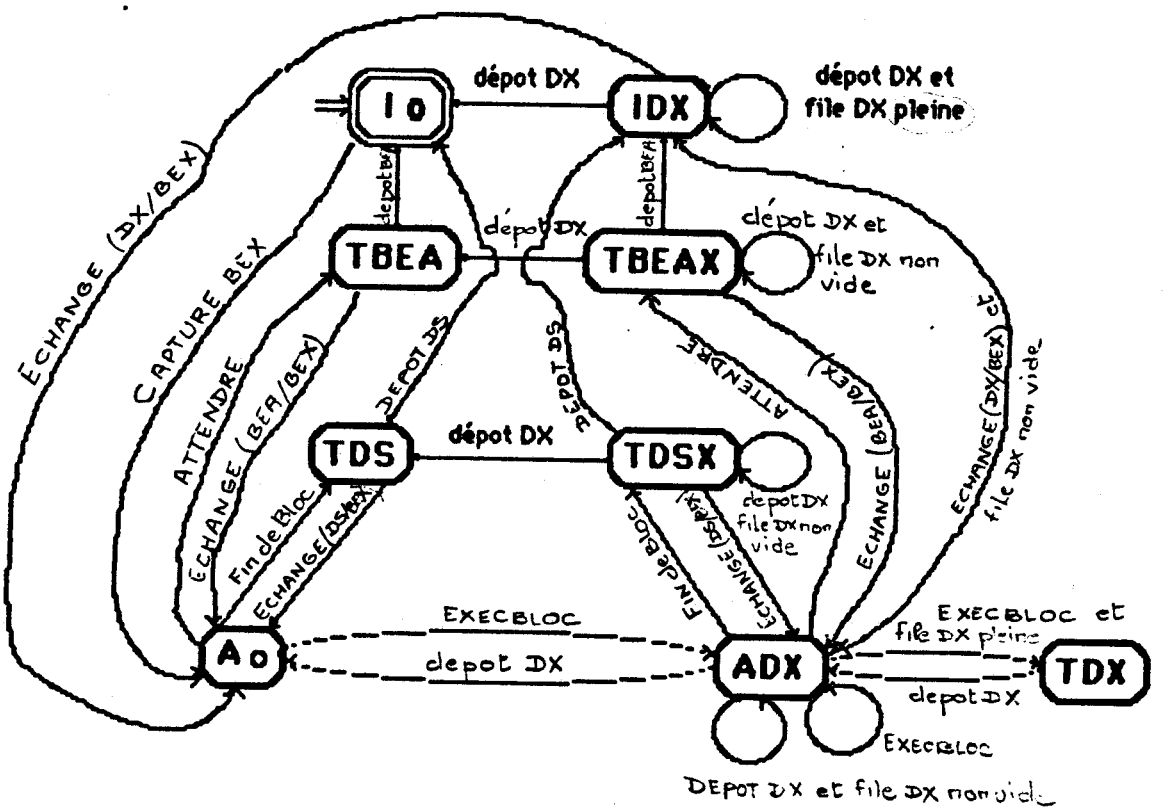


Fig. 1 Automate d'état du Processeur d'Exécution

La signification des états de cet automate est la suivante :

* I_O : état inactif avec file DX vide. Cet état est l'état initial et final de cet automate
* I_{DX} :état inactif avec file DX non vide
* A_O : état actif avec file DX vide
* A_{DX} : état actif avec file DX non vide
* T_{DS} : état d'attente pour dépôt DS avec file DX vide
* T_{DSX} : état d'attente pour dépôt DS avec file DX non vide
* T_{BEA} : état attente pour dépôt BEA avec file DX vide
* T_{BEAX} : état attente pour dépôt BEA avec file DX non vide
* T_{DX} : état attente pour dépôt DX avec file DX pleine

L'abréviation BEX est utilisée pour désigner un bloc exécutable. Nous pouvons remarquer que sur ce paragraphe ne figure pas la primitive début bloc car celle-ci est implicitement liée à la capture d'un bloc exécutable.

II- ARCHITECTURE DU PROCESSEUR D'EXECUTION

L'architecture du Processeur d'Exécution découle des aspects fonctionnels que nous avons présentés. Ceux-ci ont mis en évidence deux caractères fondamentaux du Processeur d'Exécution qui sont :

* un caractère traitement qui consiste à exécuter les instructions d'un bloc exécutable

* un caractère communication qui permet d'assurer les opérations de transfert entre le processeur et la mémoire circulante

A ces deux caractères on a fait correspondre respectivement une Unité de traitement et une Unité d'Echange automatique.
Nous décrivons dans les paragraphes suivants l'implémentation que nous avons choisie pour la réalisation de ces deux unités.

II-a Unité de traitement

II-a-1 Définition de l'unité de traitement

L'unité de traitement a pour rôle d'exécuter les instructions d'un bloc exécutable. Pour ce faire, nous avons vu précédemment qu'elle est contrainte à demander à l'unité d'échange des opérations de transfert caractéristiques de l'état dans lequel se trouve le Processeur d'Exécution.

Deux fonctions incombent donc à l'unité de traitement :

- une fonction de traitement qui consiste à dérouler les instructions d'un bloc exécutable (primitives comprises)

- une fonction de demande de service auprès de l'unité d'échange qui sera chargée de les honorer.

La fonction de traitement nécessite quelques explications en ce qui concerne l'exécution des primitives. Les primitives FIN DE BLOC, EXECBLOC, et ATTENDRE conduisent respectivement à la création d'un bloc de type domaine de sortie, demande d'exécution et bloc en attente. Ces blocs seront construits à l'intérieur d'une mémoire locale en vue de leurs insertions dans la mémoire circulante par l'unité d'échange.

La fonction de demande de service consiste à envoyer à l'unité d'échange un message précisant le type de service demandé. Celui-ci devra comporter en outre :

* le type de transfert demandé. Celui-ci est déterminé de manière univoque par l'état dans lequel se trouve le processeur (voir automate figure 1)

* l'adresse en mémoire locale où se trouve le bloc concerné par le transfert.

II-a-2 Implémentation de l'unité de traitement [GLA 82]

Construite autour d'un système à microprocesseur 16 bits Z 8000 l'implémentation comporte comme le montre la figure 2 :

* un microprocesseur Z 8002. L'espace d'adressage réduit et le caractère monotache inhérent à l'unité de traitement nous ont conduis à choisir une version du Z8000 non segmentée

* un système entrée/sortie relié à une console pour la mise au point de la carte unité de traitement

* une mémoire propre constituée de deux parties :

  - 1ère partie : mémoire morte comportant un logiciel de mise au point et le logiciel de base du processeur (primitives)

  - 2ème partie : mémoire vive de travail

* une mémoire locale (RAM) en double accés transparent pour l'unité de traitement et l'unité d'échange. Cette mémoire est utilisée pour les communications entre le Processeur d'Exécution et l'anneau de mémoire circulante.

* un port d'échange des "statuts servant à la fois pour l'envoi de demandes de service vers l'unité d'échange, la réception d'acquitements de service de l'unité d'échange sur interruption de celle-ci ainsi qu'à l'initialisation de l'unité d'échange (Clear Unité d'Echange).

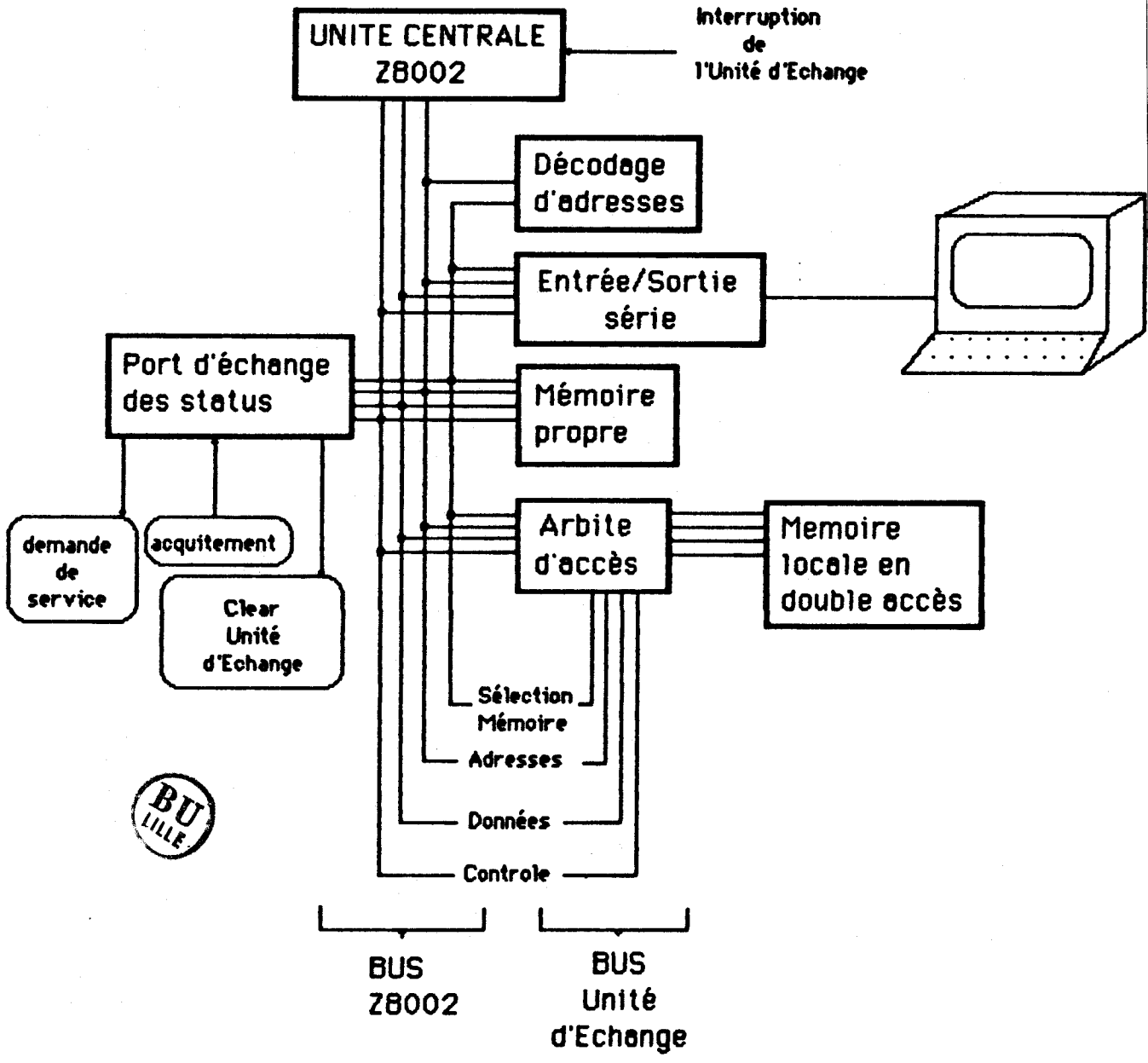


Fig. 2 Implémentation de l'unité de traitement

II-b Unité d'échange automatique [GON 82]

2-b-1. Définition

Cette unité d'échange assure la communication d'informations entre la mémoire circulante et un Processeur d'Exécution. C'est elle qui gère les transferts de blocs entre un cadre de la mémoire circulante et la mémoire locale de l'unité de traitement.
En fonction de l'état dans lequel le Processeur d'Exécution se trouve l'unité de traitement sera amenée à formuler des demandes de service particulier, à l'unité d'échange.
A chacune de ces demandes l'unité d'échange recherche un bloc de l'anneau qui lui permette (tableau figure 3) d'honorer cette demande.

Etat processeur	demande de service	type bloc recherché
inactif	* capture d'un bloc exécutable (BEX) * dépôt d'une demande d'exécution (DX)	BEX cadre vide (CV)
actif	* dépôt d'une demande d'exécution (DX)	cadre vide (CV)
attente	* dépôt d'une demande d'exécution (DX) * dépôt d'un domaine de sortie (DS) * dépôt d'un bloc en attente (BEA)	cadre vide (CV) CV ou BEX CV ou BEX

Fig. 3 Type des demandes de service

En cours d'exécution, l'unité de traitement peut formuler une nouvelle demande de service bien que les demandes précédentes n'aient été pas encore honorées par l'unité d'échange. Celle-ci peut être :

- un dépôt de DX précédé d'un ensemble de demandes DX
- un dépôt de DS " "
- un dépôt de BEA " "

le langage engendré par l'unité de traitement peut être décrit par l'expression : DX^* (DX+DS+BEA)

L'unité d'échange devra donc disposer d'un mécanisme capable de sélectionner une demande de service parmi plusieurs.

2-b-2. Architecture de l'unité d'échange automatique

L'aspect fonctionnel de cette unité met en évidence trois fonctions principales :

* fonction de gestion des demandes de l'unité de traitement
* fonction d'observation de l'anneau en vue de rechercher les blocs désirés
* fonction de transfert pour gérer les communications proprement dites

Trois sous-unités vont asssurer ces fonctions, ce sont dans l'ordre :

* unité gestionnaire des demandes
* unité de recherche
* unité de transfert

L'architecture de l'unité d'échange est montrée figure 4.

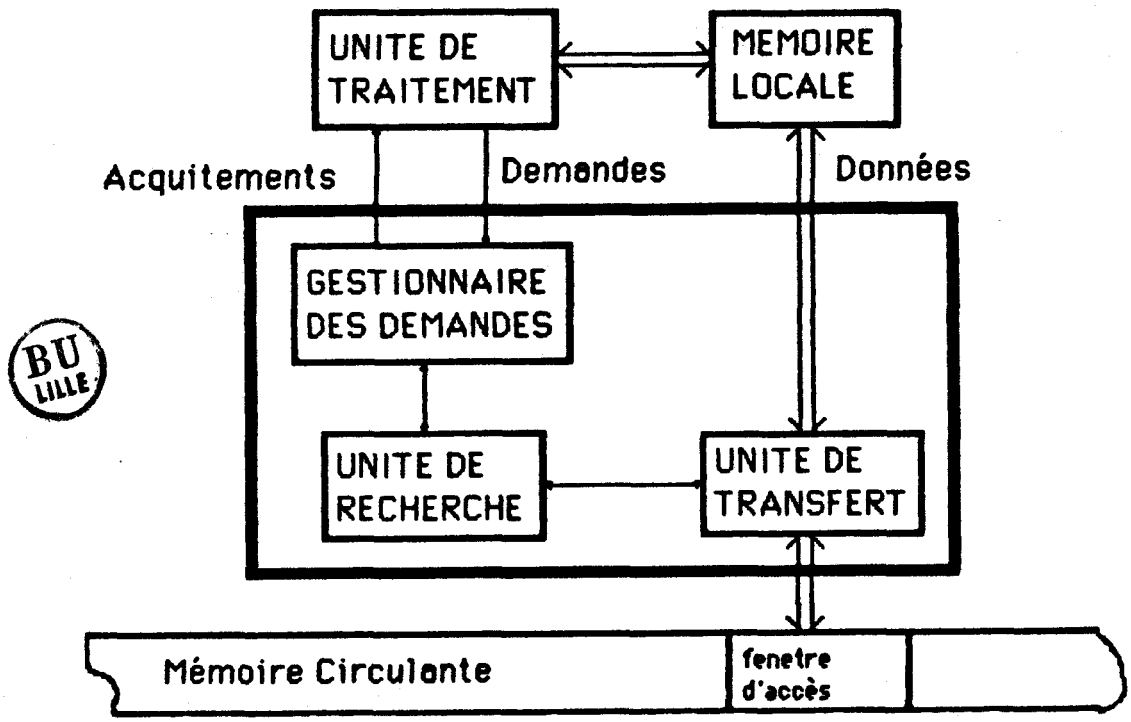


Fig. 4 Unité d'échange

De manière simpliste le fonctionnement de l'unité d'échange peut être assimilé au fonctionnement de n'importe quelle unité centrale classique c'est-à-dire qu'il comprend trois phases :

* une phase de recherche (Fetch) qui consiste à relever le type de bloc qui défile sous la fenêtre d'accés du processeur. L'unité de recherche intervient dans cette phase

* une phase de décodage, qui permet de décider du comportement à adapter et dans laquelle intervient le gestionnaire des demandes

* une phase d'exécution du transfert dans laquelle intervient l'unité de transfert sous contrôle de l'unité de recherche.

Nous décrivons dans les lignes qui suivent l'architecture et l'implémentation que nous avons choisies pour la réalisation de ces sous-unités.

Le gestionnaire des demandes

Son rôle se limite à sélectionner une demande de service parmi l'ensemble des demandes que l'unité de traitement a pu formuler, en fonction du type du bloc courant (i.e bloc situé sous la fenêtre d'accés du processeur) relevé par l'unité de recherche. La demande sélectionnée sera transmise à l'unité de recherche pour exécution.

Implémentation

Les demandes de service que reçoit le gestionnaire des demandes étant implicitement liées à l'état du processeur, il suffit que celui-ci inscrive périodiquement son état dans un registre (registre d'état processeur) pour que le gestionnaire des demandes en prenne connaissance. Un second registre (registre type bloc courant) mis à jour cycliquement par l'unité de recherche permet au gestionnaire de demande de connaître le type de bloc qui défile sous la fenêtre d'accés du processeur. La sélection d'une demande de service parmi plusieurs étant déterministe (voir automate fig. 1) un simple circuit programmable (ROM sélecteur de demandes) l'implémente. Enfin, un dernier registre (registre demande sélectionnée) mémorise pour l'unité de recherche la demande de service sélectionnée (fig.5).

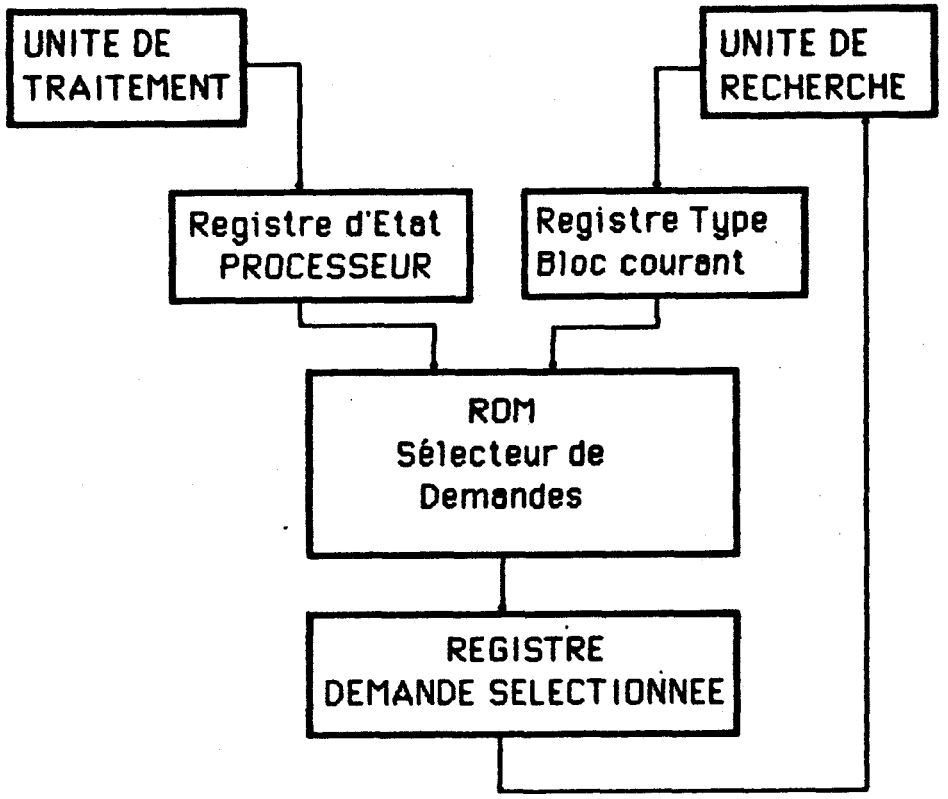


Fig. 5 Le gestionnaire des demandes

Unité de recherche

Cette unité plus complexe que la précédente assure trois fonctions distinctes :

- recherche du type du bloc courant
- contrôle de l'unité de transfert
- acquittement d'une demande de service

* La fonction de recherche du type du bloc courant consiste à détecter le début d'un bloc qui passe sous la fenêtre d'accés du processeur afin d'y relever le type de ce bloc. Le type du bloc figure dans une entête comme le montre la figure 6.

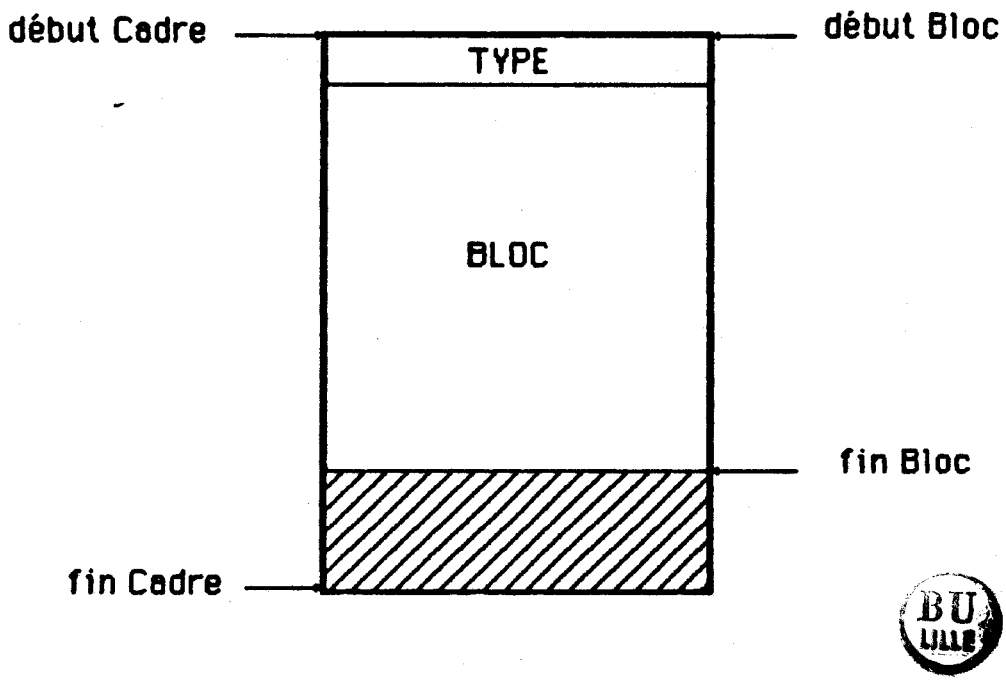


Fig.6 Structure d'un bloc

La gestion de la mémoire circulante étant réalisée au niveau d'un cadre de longueur fixe, il suffit d'un signal périodique début de cadre pour indiquer à l'unité de recherche la présence d'un nouveau bloc sous la fenêtre d'accés du processeur.
Le type de bloc relevé est alors envoyé au gestionnaire des demandes.

* la fonction de contrôle permet d'indiquer à l'unité de transfert :
  -le type de transfert à effectuer (aucun, dépôt de bloc, capture de bloc, échange de blocs)

  - l'adresse en mémoire locale où le transfert doit s'effectuer. Un générateur d'adresses de transfert mis à jour périodiquement par l'unité de traitement assure cette tache. Cette adresse est fonction de la demande de service sélectionnée.

Deux autres signaux de synchronisation (mot présent, transfert autorisé) signalent à l'unité de recherche le moment où respectivement celle-ci peut lancer une opération de lecture ou d'écriture dans la mémoire circulante. (Voir chronogramme de la figure 7). Ces signaux tiennent compte des caractéristiques de la mémoire circulante (définies dans le chapitre II).

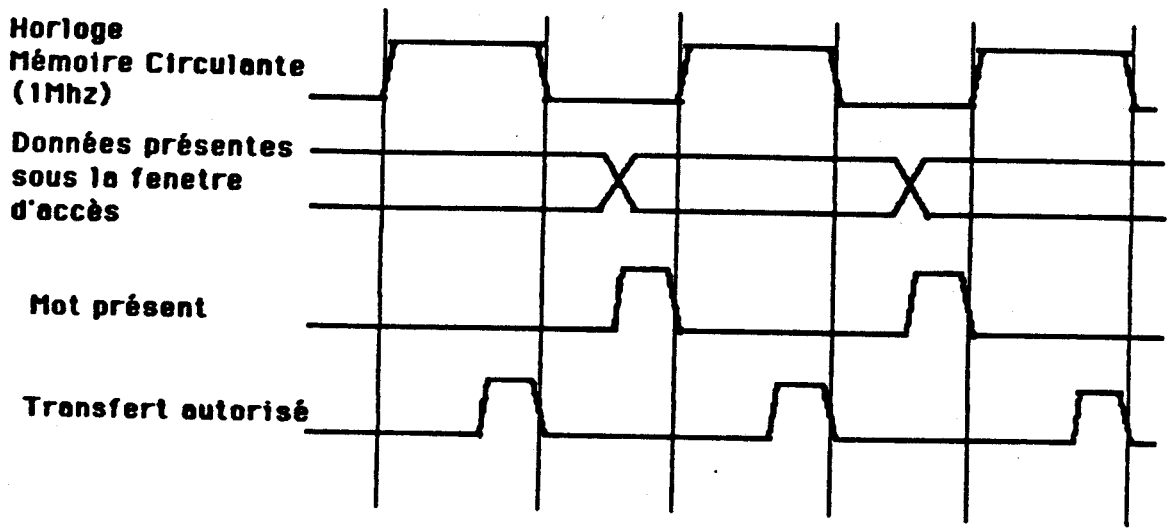


Fig. 7 Signaux de synchronisation

*L'acquitement d'une demande de service consiste à interrompre l'unité de traitement pour lui communiquer par l'intermédiaire de son registre port d'échange des status le type de service effectué. Celle-ci pourra alors en déduire le nouvel état du processeur et mettre à jour le registre d'état processeur (voir figure 1 automate d'états du processeur).

Implémentation

L'implémentation de l'unité de recherche (fig. 9 est réalisée autour d'un automate microprogrammable fonctionnant à une vitesse de 16 MHz. Cette vitesse lui permet, pendant un cycle de la mémoire circulante (1µs), d'effectuer les tests et les commandes nécessaires pour les transferts de bloc. La demande de service sélectionnée par le gestionnaire des demandes fournit l'adresse de début du microprogramme correspondant à exécuter. Le déroulement du microprogramme est assuré par un séquenceur capable d'effectuer des ruptures conditionnelles à la réception d'évènements extérieurs choisis. Le format d'une microinstruction est du type à champs fixes et comprend (fig. 8) :

- une partie commande nécessaire au pilotage des différentes unités
- une partie séquencement pour le déroulement du microprogramme sélectionné. Un champ test permet de sélectionner un évènement extérieur (signaux de synchronisation, compteur de mots, etc...). En fonction du test, le champ adresse 1 ou le champ adresse 2 représentera l'adresse de la prochaine microinstruction à exécuter dans le microprogramme courant.

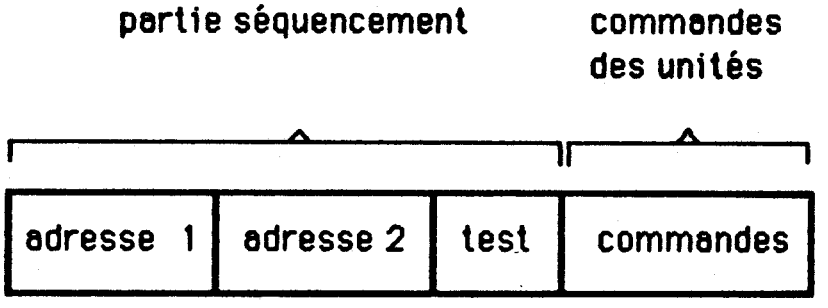


fig. 8 format d'une microinstruction

Un compteur de cadres et un compteur de mots autorisent des opérations de transfert par bloc de taille variable. Un bloc peut alors occuper un ou plusieurs cadres de la mémoire circulante. Dans cette éventualité une mémoire de comparaison permet d'effectuer des recherches selon le type du bloc ou le nom de ce bloc ; c'est notamment le cas lorsqu'après avoir prélevé de la mémoire circulante le premier cadre d'un bloc exécutable de nom X qui tient sur plusieurs cadres non forcément consécutifs, on désire récupérer tous les cadres de nom X qui interviennent dans le bloc.

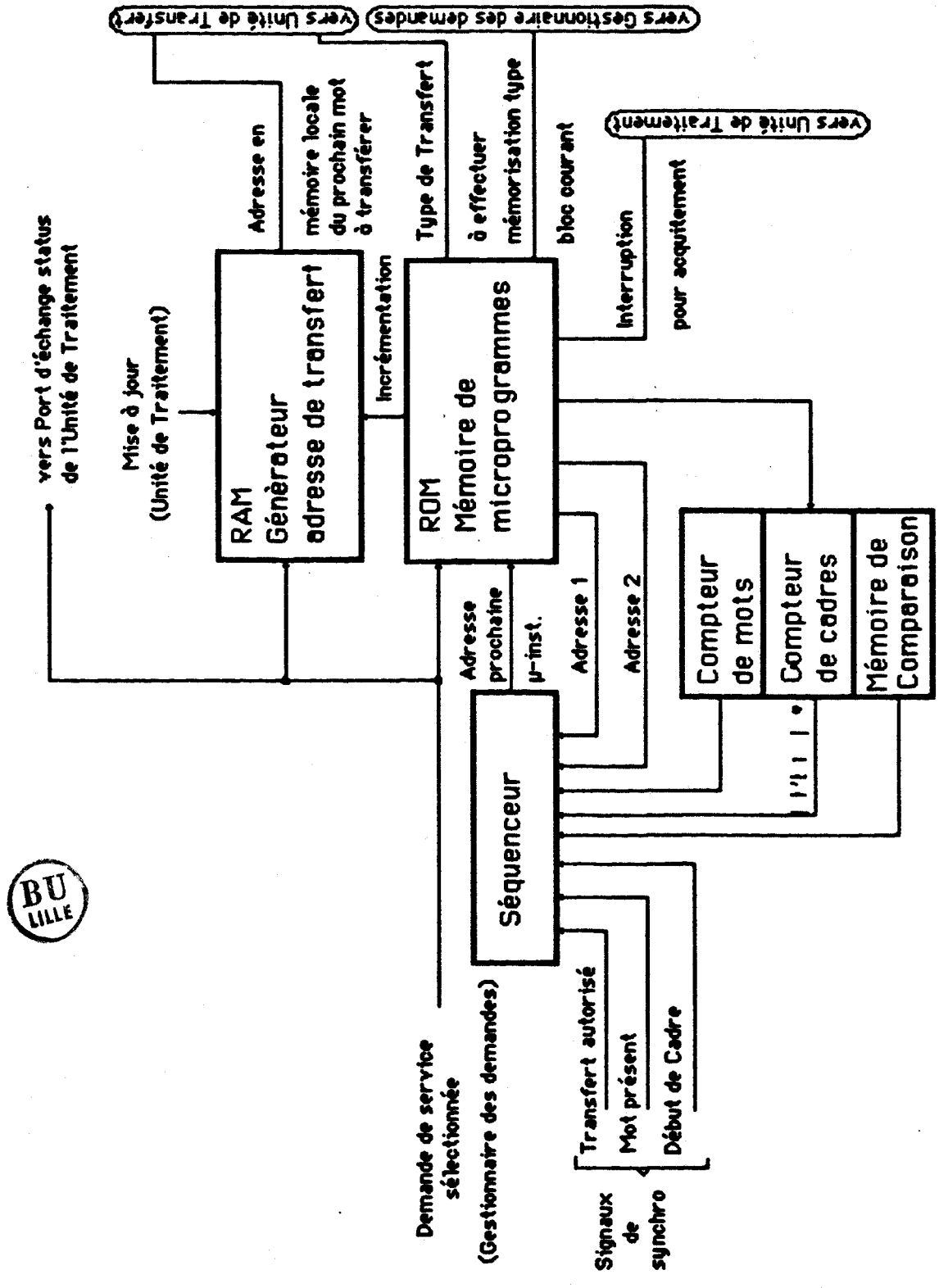


Fig. 9 Implémentation de l'unité de recherche

Le comportement de l'unité de recherche peut être assimilé à l'automate de la figure 10.

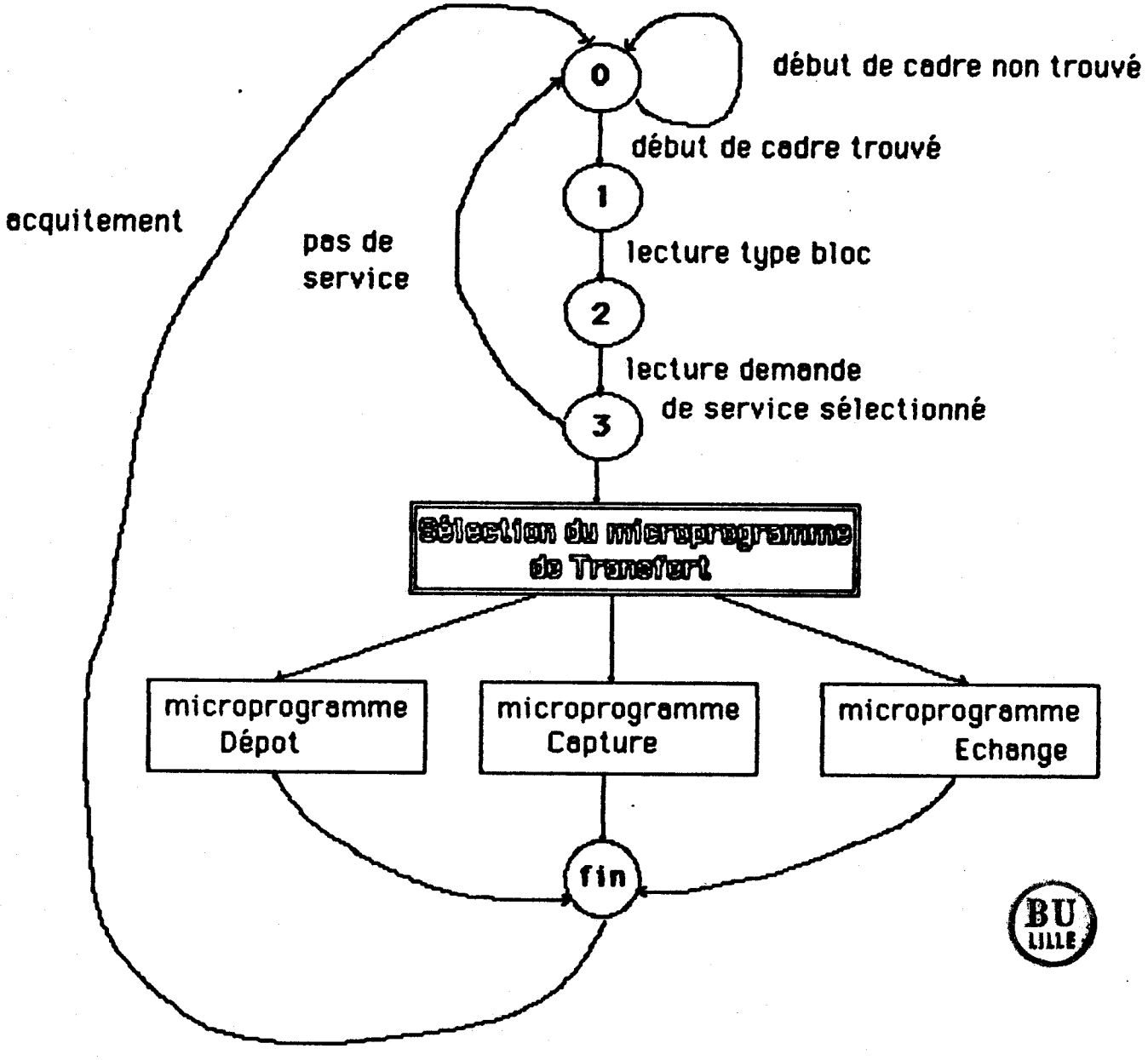


Fig. 10

Unité de transfert

Cette unité assure la gestion des voies de communication entre la mémoire locale du processeur et la mémoire circulante.
Pour cela elle reçoit de l'unité de recherche :

- le type de l'opération à réaliser (lecture, écriture, mise à jour, rien).
- l'adresse en mémoire locale du mot concerné par l'opération de transfert.

L'implémentation de cette unité (Fig. 11) est constituée :

+ d'un automate microprogrammable fonctionnant à 16 Mhz qui exécute le microprogramme correspondant à l'opération demandée

+ d'un régistre qui relève en permanence le mot présent devant la fenêtre d'accés au rythme de l'horloge de la mémoire circulante (1 Mhz). Son contenu peut être lu par l'unité de recherche, qui décide ou non de son transfert.

+ d'interfaces d'accés à la mémoire locale du processeur. L'accés à cette mémoire ne pertube pas l'unité de traitement (mémoire à double accés transparent).

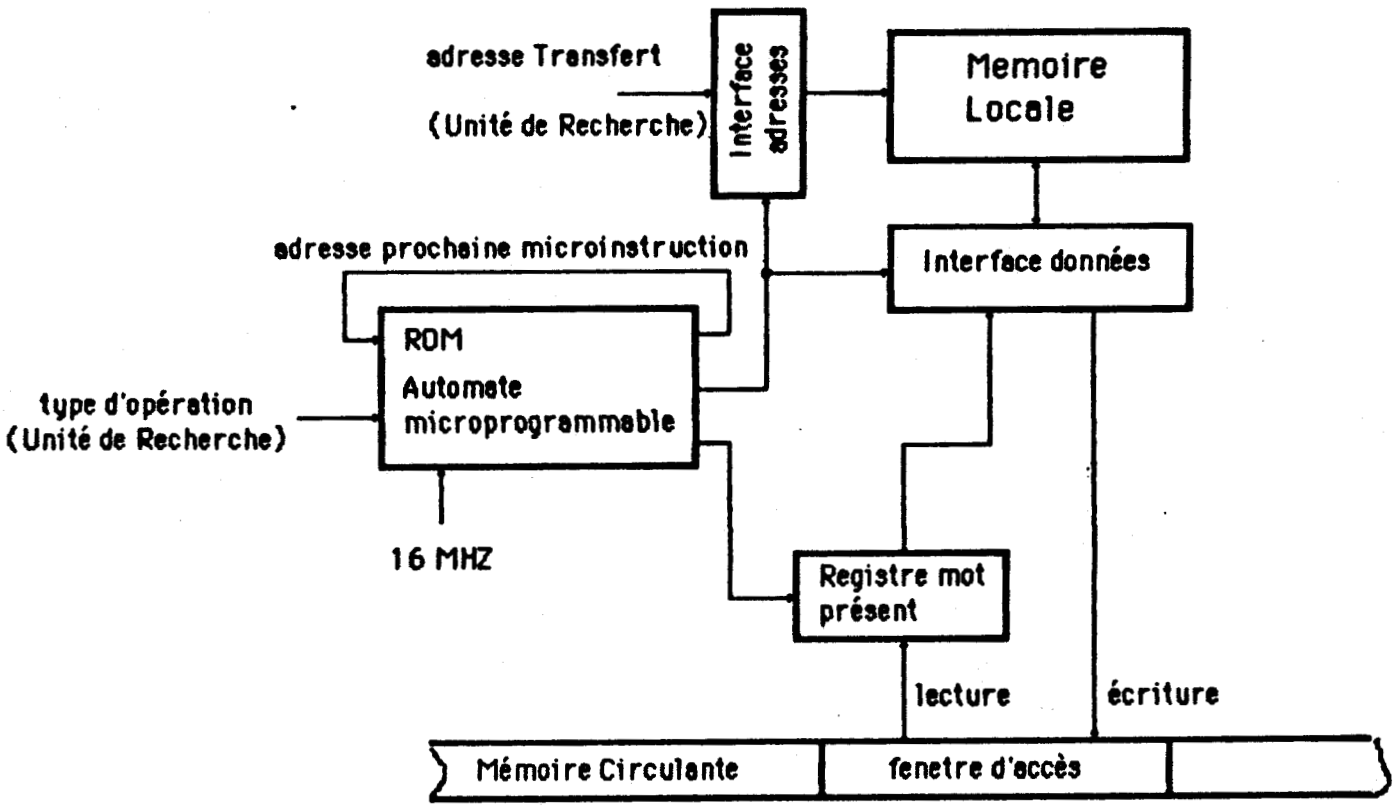


Fig. 11 Unité de transfert

Le fonctionnement de l'unité de transfert respecte le chronogramme de la figure 12.

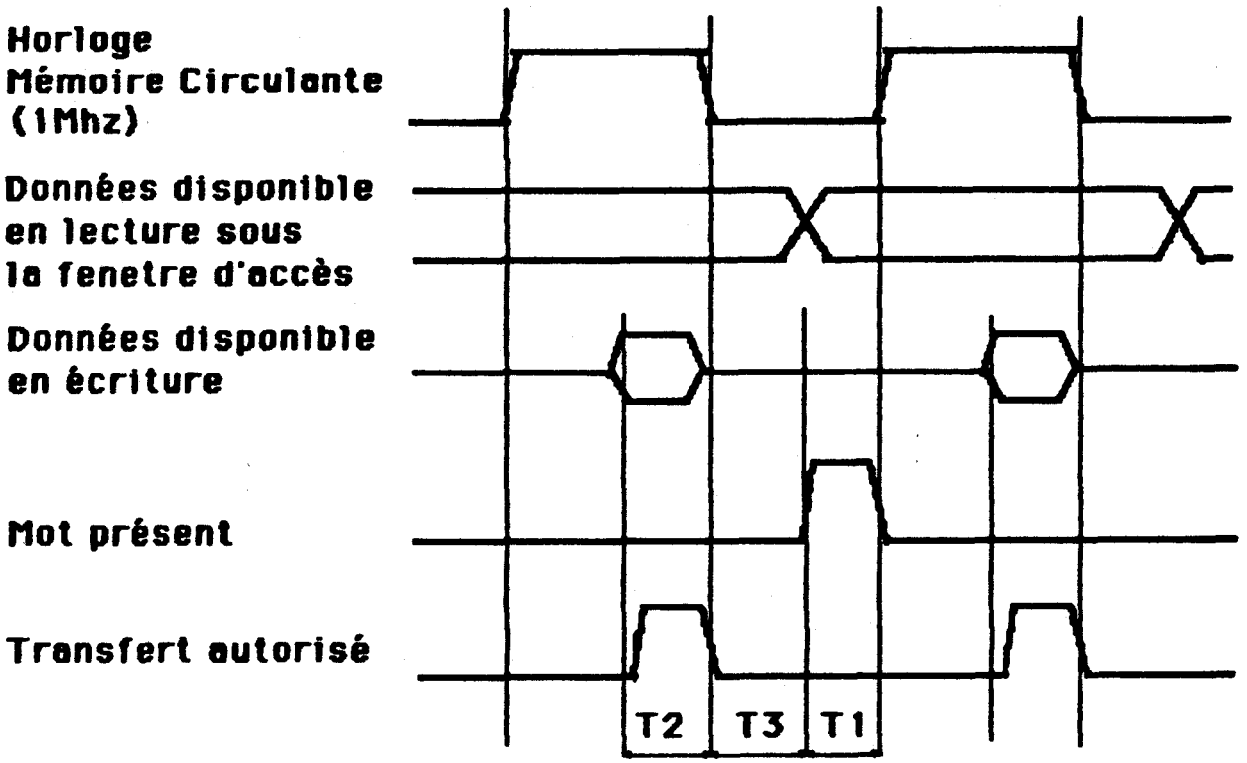


Fig. 12 Accés à la mémoire circulante

T_1 : représente l'intervalle de temps pendant lequel l'unité de transfert effectue la lecture du mot présent devant la fenêtre d'accés du processeur. Celui-ci est mémorisé dans le registre mot présent.

T_2 : concerne l'instant où l'unité de transfert effectue une opération d'écriture en mémoire circulante

T_3 : correspond à l'écriture en mémoire locale du mot relevé au cours de T_1 et présent dans le registre mot présent.

Selon l'opération à réaliser une ou plusieurs périodes interviennent :

- opération de lecture : T_1 suivie de T_3
- opération d'écriture : T_2
- opération de mise à jour : T_1, T_2 et T_3

Conclusion :

Nous avons exposé ici les grandes caractéristiques de l'unité d'échange que nous avons développée pour palier à l'adressage séquentiel de la mémoire circulante avec l'adressage aléatoire des microprocesseurs standards que l'on utilise pour la réalisation des processeurs fonctionnels de MAUD. La structure microprogrammée de cette unité permet d'envisager son utilisation pour d'autres applications que la notre.

Dans le chapitre 4 relatif au processeur Mise à Jour nous supposerons la structure de cette unité connue et nous nous intéresserons uniquement aux structures de données et aux mécanismes nouveaux que nous avons intégrés dans celle-ci pour la réalisation des différentes fonctions spécifiques au processeur Mise à Jour.

CHAPITRE IV

Le Processeur Mise à Jour

Introduction

A. Les fonctions de mise à jour et de sélection dans les machines dirigées par les données

- A.1 Machine dirigée par les données de type statique ou dynamique
- A.2 Structure des mémoires utilisées
- A.3 Fonction mise à jour centralisée ou répartie
- A.4 Structure de l'unité de mise à jour
- A.5 Mécanisme de détection des instructions prêtes
- A.6 Communication unité de mise à jour et unités d'exécution

B. Fonctionnalité du processeur Mise à Jour de MAUD

- B.1 Rôle
- B.2 1er mode
- B.3 2ème mode
- B.4 3ème mode

C. Architecture du processeur Mise à Jour

D. Implémentation du processeur Mise à Jour

- D.1 Introduction
- D.2 Contraintes physiques
- D.3 Implémentation de la T.P.U
- D.4 Implémentation de la T.C.A.S
- D.5 Modification de l'entête du bloc
- D.6 Conclusion

Introduction

Ce processeur spécialisé est sans doute celui qui réclame le plus d'attention tant au niveau fonctionnel qu'au niveau implémentation. Ce processeur peut être considéré comme le cerveau de la machine dans la mesure où c'est lui qui produit les blocs exécutables pour les Processeurs d'Exécution de MAUD. Il peut s'apparenter en ce sens à l'unité de contrôle et de commande d'une machine type VON NEUMANN. C'est lui en effet qui contrôle et organise l'enchaînement des activités dans MAUD à l'échelon macroscopique par l'implémentation d'un mécanisme dirigé par la disponibilité des données.

Cette fonction primordiale risque d'être, dans une architecture mal équilibrée, un goulot d'étranglement par le biais duquel le parallélisme obtenu à l'exécution se trouverait nettement inférieur au parallélisme optimum du programme à exécuter. En effet, il faut pouvoir produire des blocs exécutables suffisamment vite pour que les Processeurs d'Exécution ne soient pas inactifs pendant un laps de temps trop prohibitif.

Ce chapitre se décompose en quatre parties. La première partie présente les différentes architectures de la fonction mise à jour et de la fonction sélection des entités exécutables des projets portant sur les machines dirigées par les données. Nous essaierons de dégager les caractéristiques qui permettent de les différencier. La seconde partie de ce chapitre est une analyse fonctionnelle du processeur Mise à Jour. Trois modes de fonctionnement y sont étudiés. La troisième partie définie l'architecture adoptée pour le mode de fonctionnement que nous avons choisi. L'implémentation de cette architecture est décrite dans la dernière partie.

A. Les fonctions de mise à jour et de sélection dans les machines dirigées par les données

Une analyse détaillée des différents modèles de machines dirigées par les données réalisés ou en cours de réalisation a permis d'établir les caractéristiques de la fonction mise à jour. Ces caractéristiques que nous allons détailler sont les suivantes :

1- Machine dirigée par les données de type statique ou dynamique
2- Structure des mémoires utilisées
3- Fonction mise à jour centralisée ou répartie
4- Structure de l'unité de mise à jour
5- Mécanisme de détection des instructions prêtes
6- Communication unité de mise à jour et unités d'exécution

A.1 Machine dirigée par les données de type statique ou dynamique

Une machine dirigée par les données est du type statique lorsque la condition pour le déclenchement d'une instruction prête à être exécutée est la suivante (en prenant comme support un graphe dirigé par les données) : "Un noeud ou instruction du graphe peut être déclenché s'il existe une marque sur chacun de ses arcs d' entrée et si il n'existe aucune marque sur chacun de ses arcs de sortie". Ce modèle bien que très simple à implémenter ne permet pas de parallélisme au niveau des boucles de traitement. Chaque itération d'une boucle devra s'exécuter séquentiellement. L'exemple suivant illustre ce problème

```
FOR     I : 1 to N   D O
BEGIN   X := A [I] + 1 ;
        B [I] := X * * 2
END ;
```

Du fait de la condition restrictive qu'un arc d'un graphe ne peut véhiculer qu'au plus une marque on ne peut lancer qu'une itération à la fois.
Pour pallier à cet inconvénient ARVIND, [ARV 80] a proposé un mécanisme de coloriage des marques qui permet d'associer à chaque itération une marque de couleur différente. La condition de déclenchement d'une instruction devient alors la suivante :" Un noeud ou instruction du graphe peut être déclenché s'il existe une marque de couleur identique sur chacun de ses arcs d'entrée". Ce mécanisme de coloriage permet également de supporter les appels de fonctions. A chaque appel d'une fonction est associé une couleur qui décrit le contexte de la procédure appelante. Une machine dirigée par les données capable de supporter les itérations parallèles et les appels de fonction multiples et récursifs est du type dynamique.
Selon le type d'une machine dirigée par les données la fonction mise à jour sera plus ou moins complexe en raison du mécanisme de coloriage qui est à prendre ou non en considération. Une machine dirigée par les données du type dynamique nécessitera la mémorisation d'informations (couleur) supplémentaires.

A.2 Structure des mémoires utilisées

La fonction mise à jour nécessite pour sa mise en oeuvre un nombre plus ou moins important de mémoires selon l'implémentation choisie pour traduire le mécanisme dirigé par les données.

De manière générale un programme dirigé par les données sera une suite d'instructions dont la structure ressemblera au format suivant :

* Un champ opération précisant le type d'opération à effectuer
* Un champ opérande indiquant les opérandes qui interviennent dans l'exécution de l'opération
* Un champ lien qui permet de connaître les destinataires qui recevront le résultat de l'opération
* Un champ contrôle qui peut être omis. Ce champ précise les conditions pour le déclenchement de l'instruction.

Nous décrivons brièvement les types d'implémentation que nous avons rencontrés.

1er La machine de BURKOWSKI [BUR 81] (Université de Manitoba - Canada).

Cette machine dirigée par les données est du type statique. La structure de l'unité de mise à jour (Fig. 1) fait apparaître une seule mémoire dans laquelle sont mémorisés les différents champs décrits plus haut.

Une unité de modification assure deux rôles :

* La mise à jour des champs opérandes et contrôle
* La détection des instructions prêtes.

La détection d'une instruction prête provoque la mise en file d'attente de son adresse.

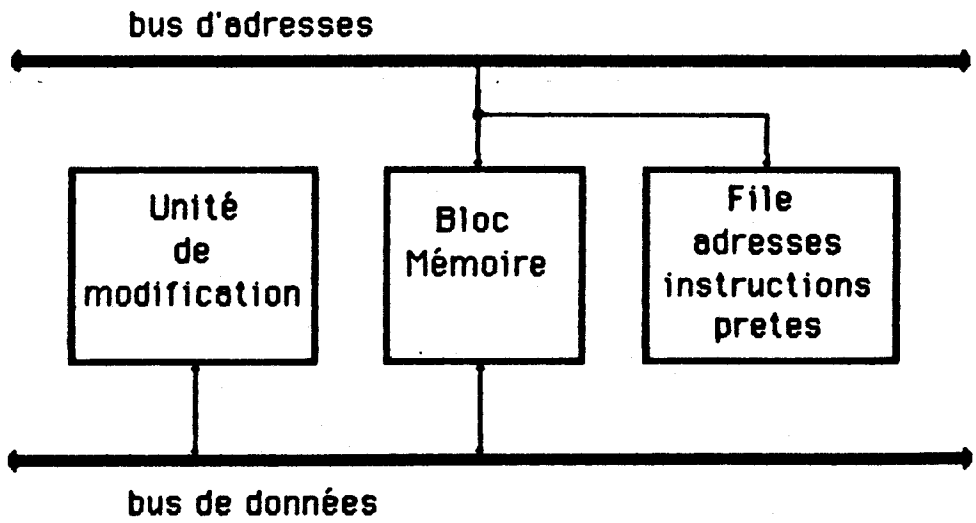


Fig. 1 Structure à une mémoire

2ème Université de Manchester [GUR 82]

Pour les machines dirigées par les données de type dynamique il y a souvent séparation du champ opérande du reste de l'instruction. Deux mémoires sont alors utilisées (fig. 2). La première mémorise les opérandes en attente d'un partenaire (Instruction à plusieurs opérandes). Lorsque tous les opérandes d'une instruction ont été évalués, ceux-ci sont envoyés ainsi que l'adresse de l'instruction destinataire à la seconde mémoire.

La seconde mémoire renferme les trois champs restant du format général d'une instruction (opération, lien, contrôle). Elle est chargée de construire des instructions prêtes en fonction des paquets opérande que lui envoie la première.

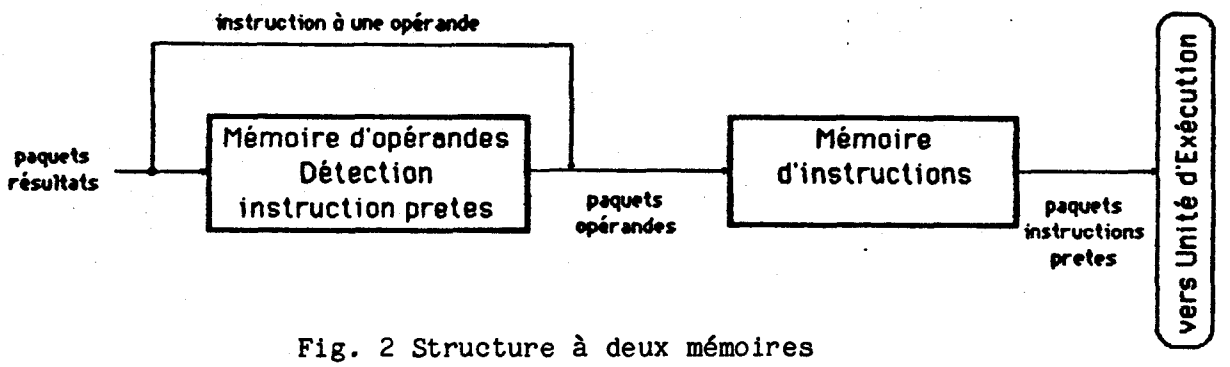


Fig. 2 Structure à deux mémoires

3ème A Data Flow Processor Array [TAK 83] (Japon)

Une implémentation différente pour une machine dirigée par les données de type dynamique consiste à éclater les trois champs (opération, opérandes, liens) sur trois mémoires (Fig. 3).

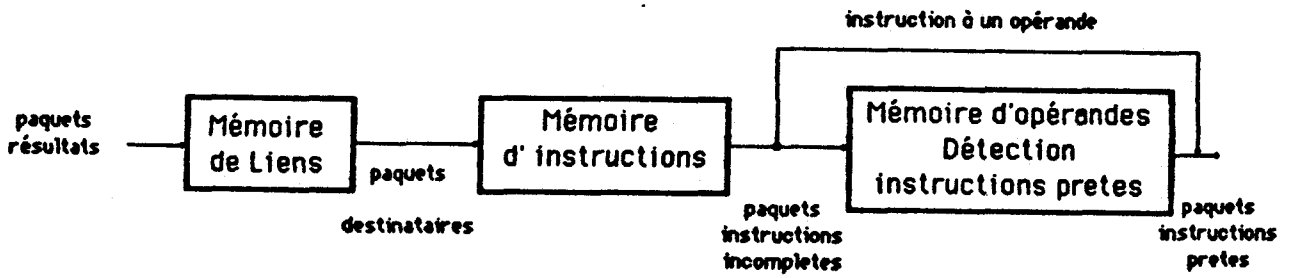


Fig. 3 Structure à trois mémoires

L'avantage d'une telle organisation vient du fait que l'unité d'exécution ne produit qu'un seul paquet résultat pour l'ensemble des destinataires. La mémoire de liens crée pour chaque paquet résultat reçu autant de paquets destinataires nécessaires.

4ème Les travaux de RUMBAUGH [RUM 77] (M.I.T)

Cette implémentation utilise aussi trois mémoires organisées différemment (Fig. 4) :

* Une mémoire d'instructions dans laquelle figurent les champs opération et liens
* Une mémoire d'opérandes destinée à recevoir les opérandes résultats
* Une mémoire de compteurs pour le déclenchement des instructions. Le passage à zéro de ce compteur indique que l'instruction correspondante est prête à être exécutée.

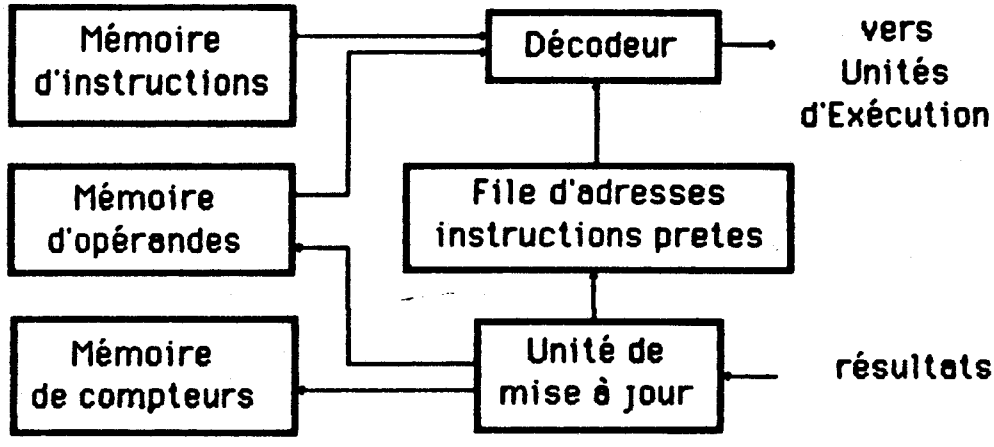


Fig. 4 Structure à 3 mémoires

5ème Le projet LAU [SYR 80]

L'architecture de la fonction mise à jour est bâtie autour de trois mémoires également (Fig. 5)

* Une mémoire locale à chaque processeur d'exécution contient à la fois les instructions et les opérandes. Le format d'une instruction est du type code opération, adresses opérandes, adresse résultat. Le mécanisme d'accés aux opérandes est donc du type par référence.
* Une mémoire destinée à recevoir le champ lien des instructions qui s'implémente sur l'unité de contrôle des données.
* Une mémoire dans laquelle on trouve le champ contrôle de chaque instruction (bits C_0, C_1, C_2). Cette mémoire est supportée par l'unité de commande d'instructions.

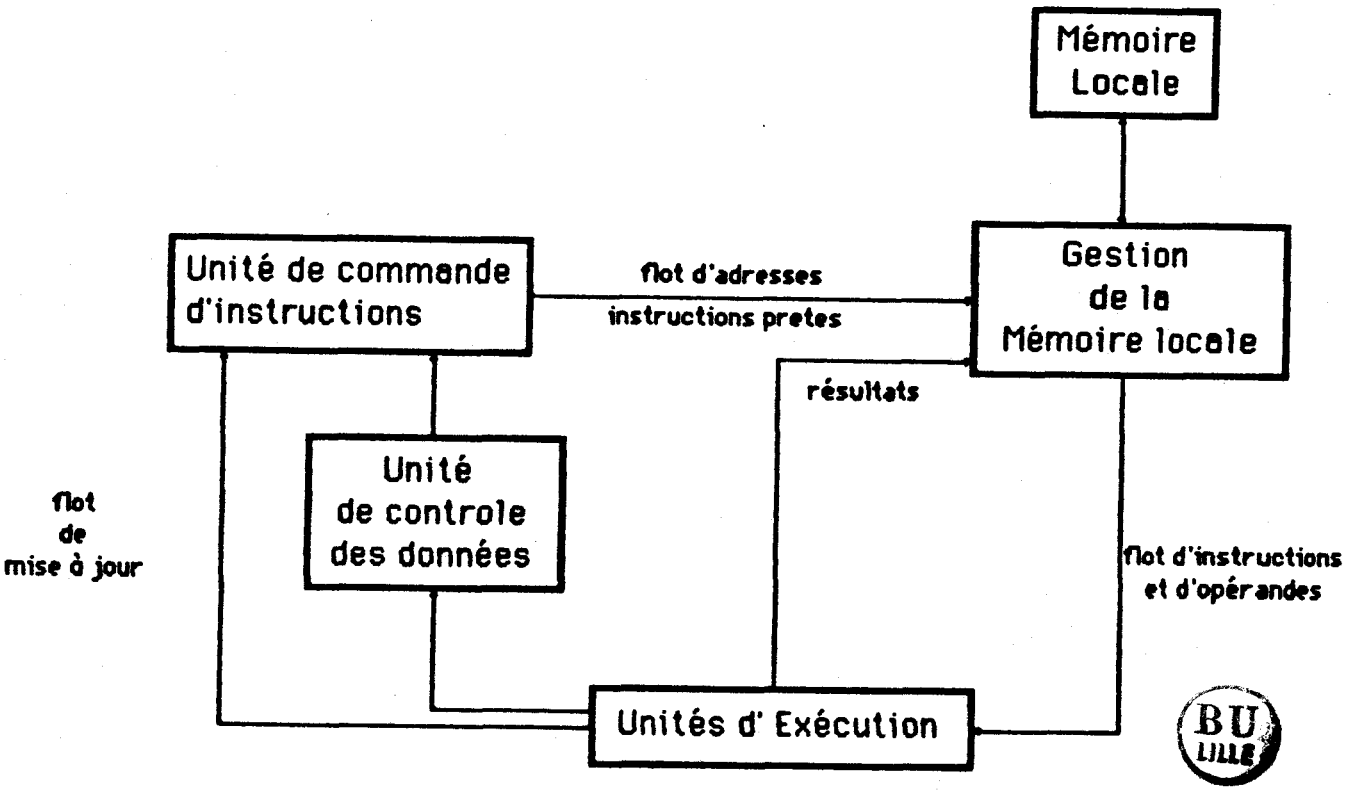


Fig. 5 Structure à trois mémoires

A.3 Fonction mise à jour centralisée ou répartie

La fonction mise à jour peut être centralisée sur une unité ([GUR 82], [LEC 79]) ou répartie sur plusieurs unités.

Mise à jour sur une unité :

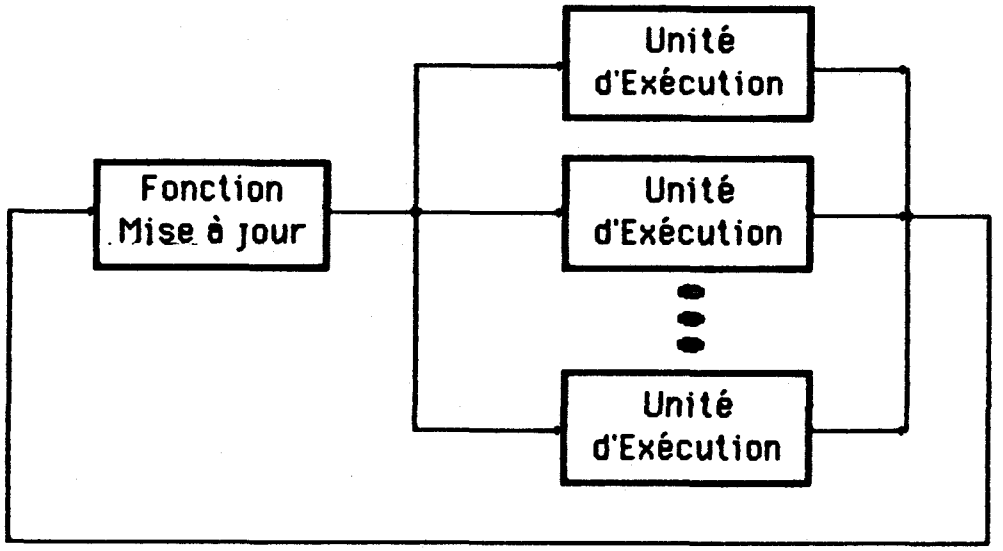


Fig. 6 Fonction mise à jour centralisée

Mise à jour sur plusieurs unités : La machine de DENNIS [DEN 80] (M.I.T)

Il faut prévoir un système de communication qui permet d'envoyer les paquets résultats vers l'unité de mise à jour appropriée (fig. 7).
Les unités d'exécution peuvent émettre deux types de paquet résultat :

- paquet résultat local
- paquet résultat global

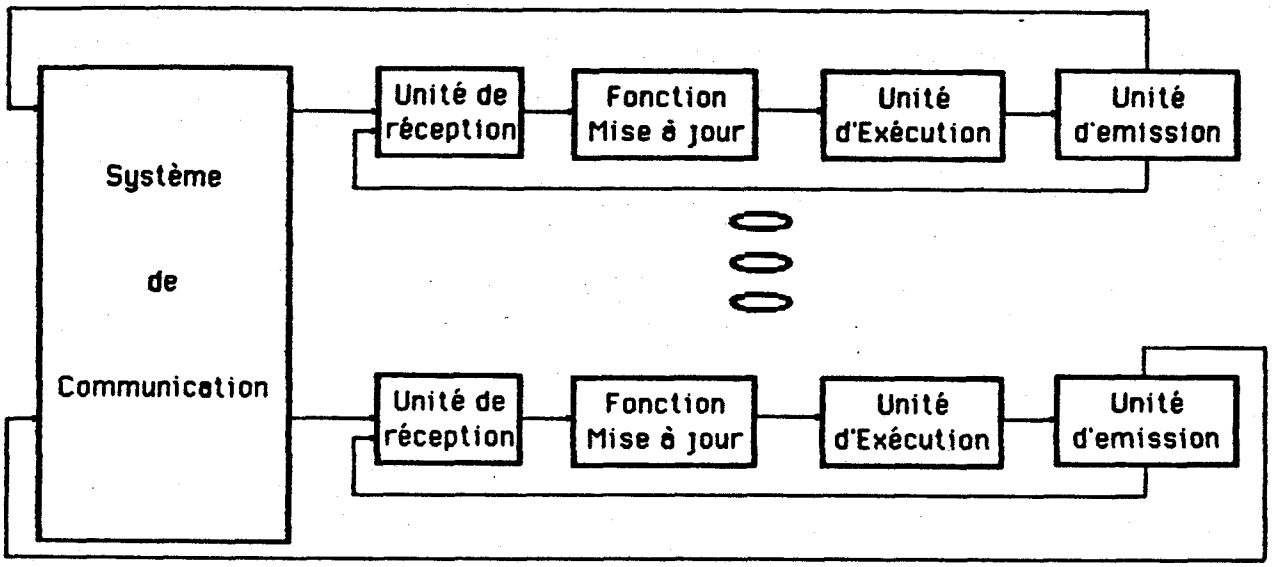


Fig. 7 Fonction mise à jour répartie

A.4 Structure de l'unité de mise à jour

Pour les machines dirigées par les données dotées d'une fonction de mise à jour à plusieurs mémoires deux modes de fonctionnement sont envisageables : mode pipeline, mode parallèle.

Dans le mode pipeline la production d'une instruction prête à être exécutée se fait généralement en deux étapes :

* recherche des opérandes
* recherche du code opération

La figure 8 nous montre une implémentation possible en mode pipeline [ARV 83].

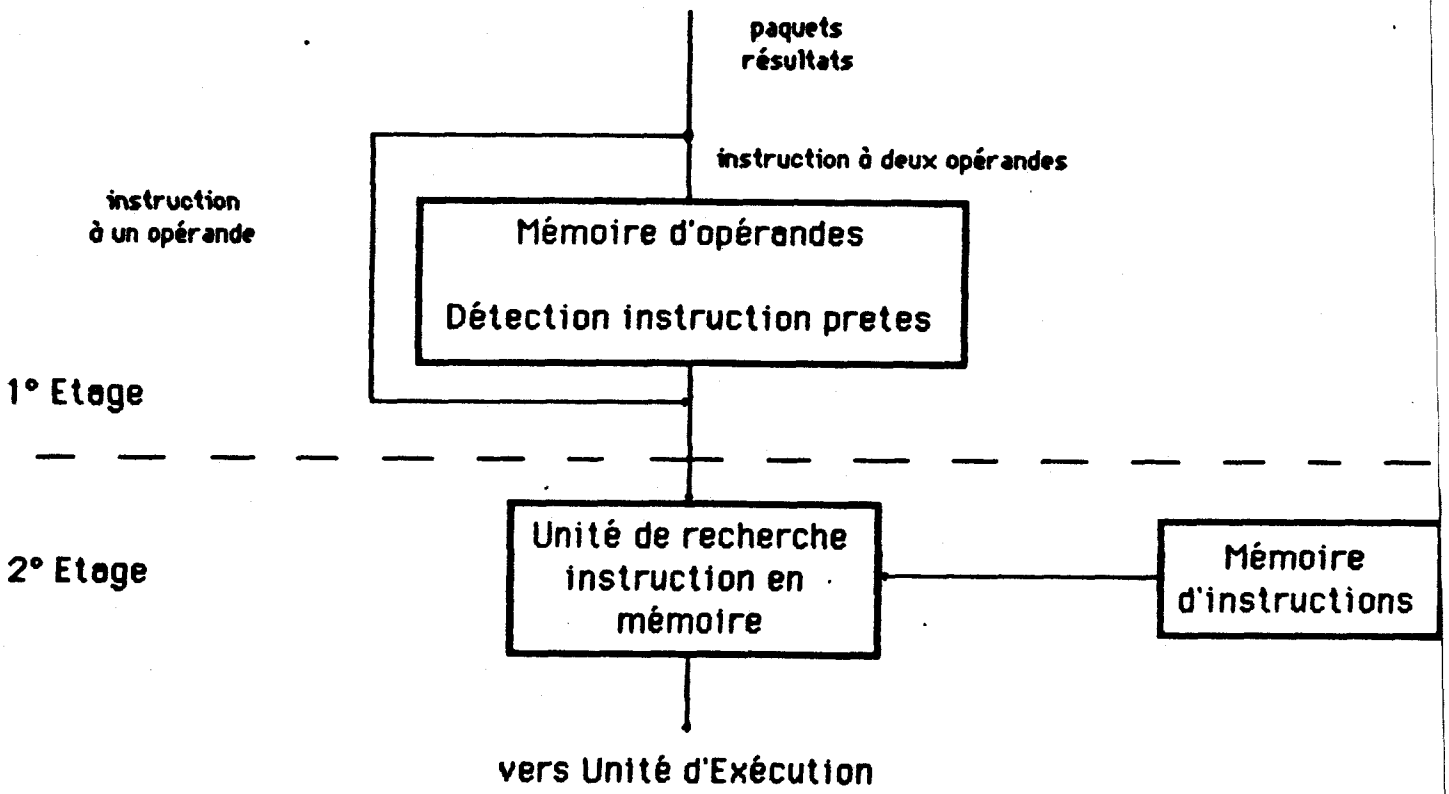


Fig. 8 Structure pipeline relevée dans la machine d'ARVIND

Les deux étapes précédentes peuvent se faire simultanément, on aboutit alors au mode parallèle. [SMI 84] (Fig. 9).

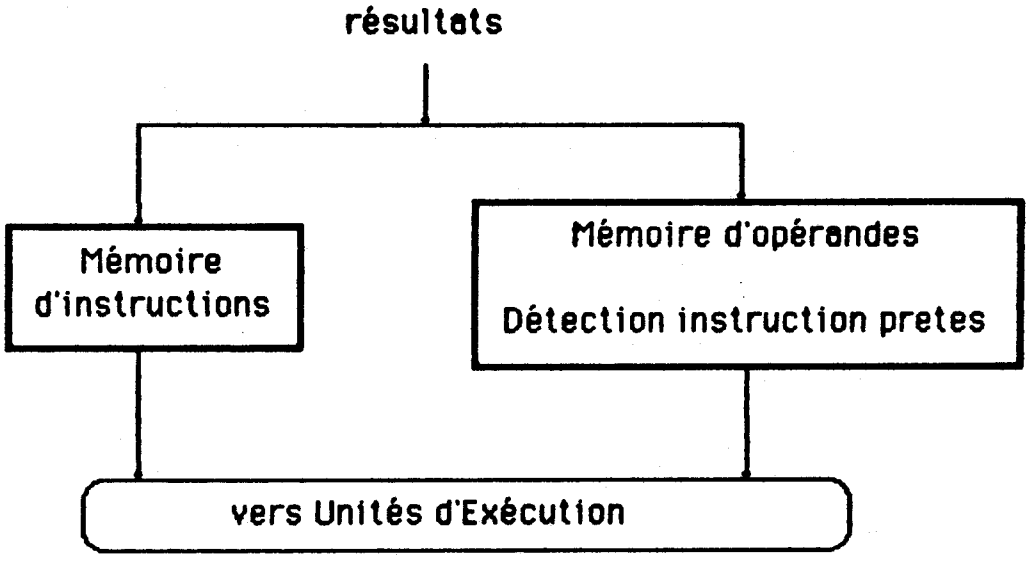


Fig. 9 Structure parallèle relevée du projet SIGMA-1 (Japon).

Dans le cas où l'instruction n'est pas prête, on ne produit rien.

A.5 Mécanisme de détection des instructions prêtes

Selon le type de la machine dirigée par les données (statique ou dynamique) le mécanisme de détection des instructions prêtes est plus ou moins complexe.

Pour une machine de type dynamique le mécanisme de détection consiste à vérifier la disponibilité des opérandes de l'instruction pour chaque paquet résultat reçu.

Pour une machine de type statique, il faut vérifier en plus que les instructions destinataires du résultat sont prêtes à recevoir celui-ci (signal d'acquittement) afin de respecter la condition de déclenchement que nous avons cité au paragraphe précédent.

La disponibilité des opérandes peut s'effectuer de trois manières possibles :

* lecture d'un compteur qui indique le nombre d'opérandes restant à évaluer. Le passage à zéro de ce compteur signale une instruction prête à être exécutée. Ce modèle a été utilisé dans les travaux de RUMBAUGH [RUM 77] et le projet AMOS [ROC 84].

* lecture de bits de présence qui indiquent les disponibilités des opérandes. Cette solution a été implémentée en outre dans LAU [SYR 80] et [SOW 82].
* Détection d'un partenaire dans le cas d'opération à deux opérandes. Ce mécanisme est certainement celui le plus utilisé, on le trouve notamment dans [ARV 83], [GUR 82] et la machine DDDP [KIS 83].

La prise en compte des signaux d'acquitement peut s'opérer par des mécanismes similaires. Un modèle avec bits de présence a été utilisé dans [BUR 81].

A.6 Communication unité de mise à jour et unités d'exécution

La communication entre l'unité de mise à jour et l'unité d'exécution peut s'effectuer de plusieurs manières :

* communication par paquets
* communication par requêtes

Dans la communication par paquets, l'unité de mise à jour produit des paquets d'instructions prêtes à être exécutées pour les Processeurs d'Exécution. Ceux-ci émettent des paquets résultats à destination pour l'unité de mise à jour (fig. 10). Des files d'attente interfacent les différentes unités de manière à synchroniser les flots d'information.

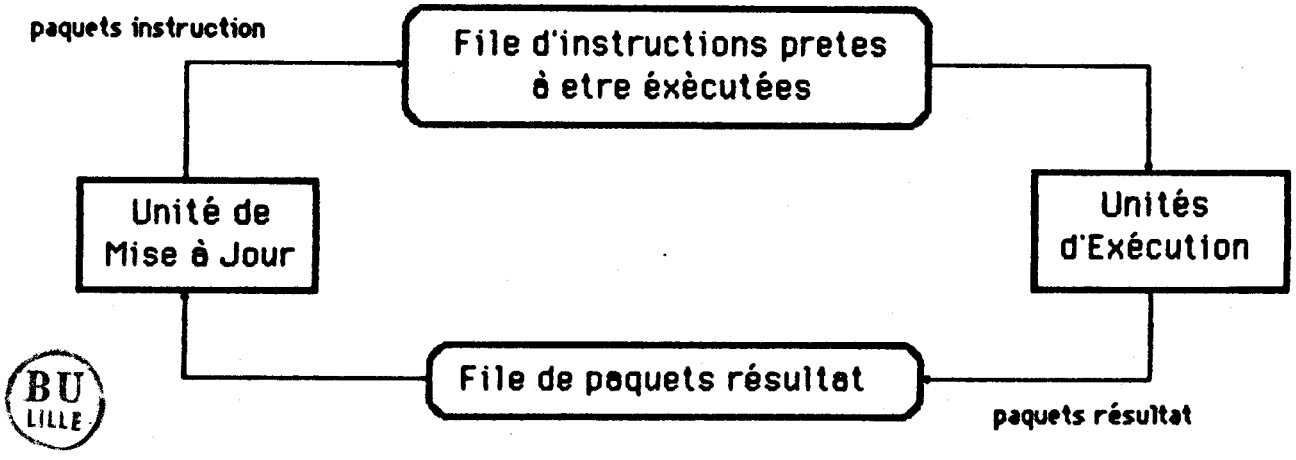


Fig. 10 Communication par paquet

Ce type de communication est le plus répandu parmi les projets que nous avons examinés.

La communication par requêtes fait intervenir un véritable dialogue entre l'unité d'exécution et l'unité de mise à jour (fig. 11). Ce type de communication a été implémenté dans l'architecture développé à l'Université d'Illinois [SOW 82].

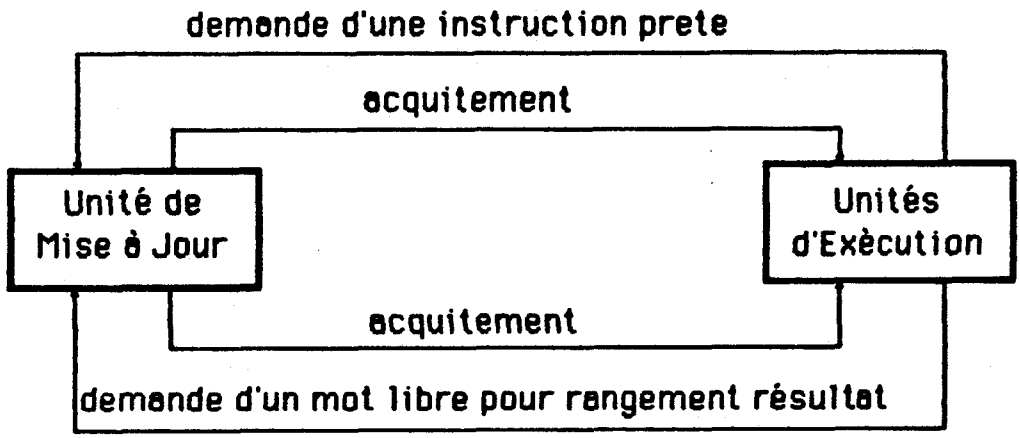


Fig. 11 Communication par requêtes

B. Fonctionnalité du processeur Mise à Jour de MAUD

B.1 Rôle

Le processeur Mise à Jour est un processeur spécialisé de MAUD dont le rôle est de fournir des blocs exécutables aux Processeurs d'Exécution. Pour ce faire il doit assurer la mise à jour des domaines d'entrée des blocs en attente en fonction des domaines de sortie fournis par les Processeurs d'Exécution (fig. 12).

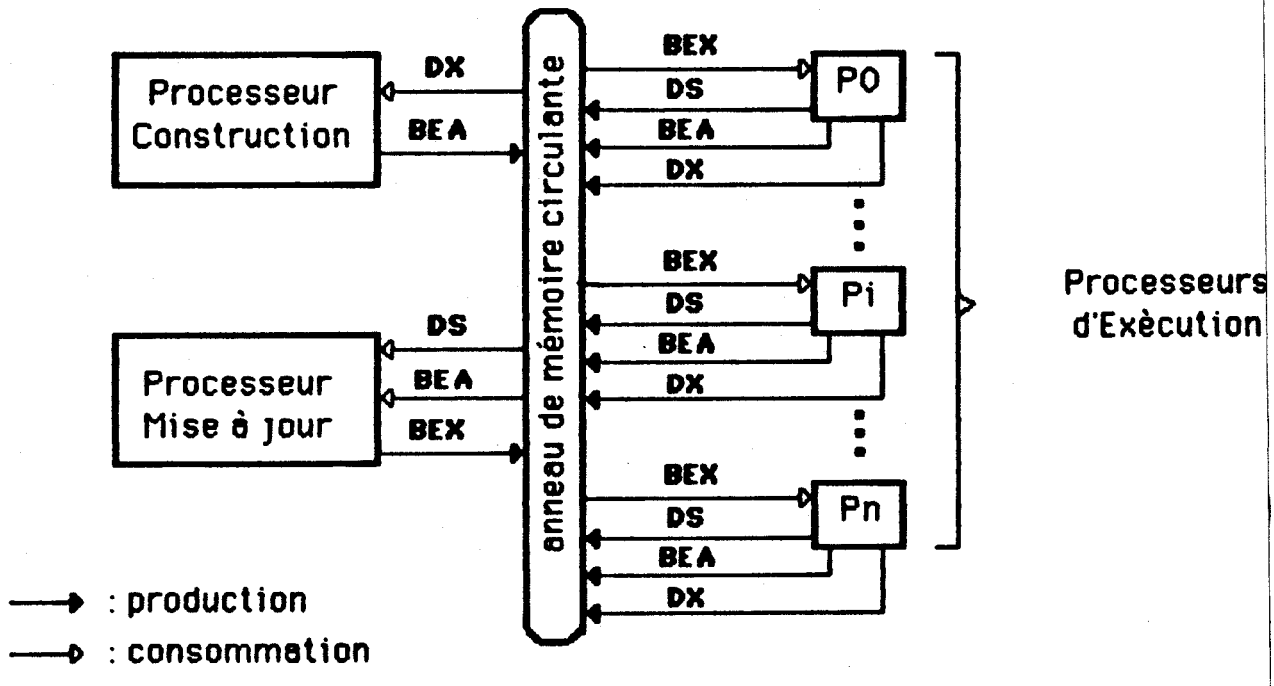


Fig. 12 Le processeur Mise à Jour de MAUD

Cette opération de mise à jour consiste en réalité en deux points précis :

* satisfaire les relations de consommation présentes dans les domaines d'entrée des blocs en attente en fonction des relations de production présentes dans les domaines de sortie.

* détecter si le domaine d'entrée du bloc BEA que l'on vient de modifier est complet et produire un bloc BEX dans l'affirmatif.

L'utilisation d'une mémoire circulante dans l'implémentation de MAUD permet d'envisager plusieurs fonctionnements possibles pour le processeur Mise à Jour selon que l'on utilise ou non la propriété de stockage d'information de celle-ci.

Le processeur Mise à Jour a besoin de consommer deux types de blocs (BEA, DS) pour pouvoir fournir des blocs exécutables.

Trois fonctionnements sont alors envisageables :

* 1er Mode : le processeur Mise à Jour range dans une mémoire propre tous les domaines de sortie et laisse circuler les blocs en attente dans l'anneau
* 2ème Mode : Le processeur Mise à Jour range dans une mémoire propre tous les blocs en attente. Les domaines de sortie restent dans l'anneau.
* 3ème Mode : La mémoire propre du Mise à Jour renferme les blocs en attente et les domaines de sortie.

Regardons pour chacun de ces modes les conséquences qu'ils amènent pour la définition du processeur Mise à Jour.

B.2 1er Mode : Domaines de sortie dans mémoire propre du Mise à Jour, BEAs dans l'anneau

Le processeur doit capturer tous les domaines de sortie qui passent devant sa fenêtre d'accés et constituer en mémoire propre une liste des couples (nom de communication, valeur) qu'il détient.

Pour chaque bloc en attente qui circule devant sa fenêtre d'accés il doit regarder s'il ne détient pas un ou plusieurs couples (nom de communication, valeur) attendus par ce bloc, et dans l'affirmatif recopier ce couple dans le domaine d'entrée examiné. Une fois le domaine d'entrée examiné, il faut s'assurer que celui-ci est complet pour pouvoir modifier l'entête de ce bloc par l'intitulé "bloc exécutable".

```
Algorithme 1 : début

        prêt : = vrai ;

        tant que NON fin du domaine d'entrée

          faire    acquérir le prochain couple (NC, non évalué) ;

                   Si ⌐ (NC, vaP) ε Liste du Maj* alors
                      transférer dans domaine d'entrée (Nc, val) ;
                      retirer (Nc, val) de la liste du Maj

                   Sinon laisser dans domaine d'entrée (Nc, non
                         évolué) ;
                         prêt : = faux ;

                   fsi
          fait

        Si prêt alors transformer bloc en BEX fsi

        fin
```

La charge de l'anneau serait donc la suivante :

Blocs pouvant circuler plus d'un tour	blocs circulant moins d'un tour
* blocs exécutables (BEX)	* domaines de sortie (DS)
* blocs en attente (BEA)	* demandes d'exécution (DX)

Cette solution devra impérativement s'accompagner d'une procédure de vérification de la charge de l'anneau afin que les situations de saturation de celui-ci soient détectées et corrigées par un retrait temporaire d'un certain nombre de BEA ou BEX vers une mémoire secondaire du processeur Mise à Jour.

Note *: cette recherche peut être longue, l'utilisation d'une mémoire associative permetterait d'accélérer le processus.

A.3 2ème mode : Domaine de sortie dans l'anneau, BEA dans la mémoire propre du Mise à Jour.

Cette solution est la solution complémentaire de la précédente, le processeur Mise à Jour capture tous les blocs en attente qui passent devant sa fenêtre d'accés de manière à constituer une liste des couples (nom de communication, non évalué) en mémoire propre.

Chaque fois qu'un domaine de sortie (DS) défile sous la fenêtre d'accés du Mise à Jour ce dernier regarde si celui-ci détient des couples (NCi, Vali) attendus dans la liste établie à l'étape précédente, et recopie alors ces couples dans les domaines d'entrée des blocs en attente correspondants.

Algorithme 2 : début

tant que NON fin du domaine de sortie

faire acquérir le couple suivant (NC, VAL) ;

Si ⫞ (NC, non évalué) ε liste du mise à jour ** alors

- transférer le couple (NC, VAL) dans le domaine d'entrée ;

- retirer (NC, non évalué) de la liste du mise à jour

Sinon laisser le couple (NC, VAL) dans le domaine de sortie

fsi

fait

fin

Comme pour la solution 1, la mise à jour devra procéder à une vérification du domaine d'entrée des blocs BEA concernés par la mise à jour, afin de détecter les blocs dont le domaine d'entrée est complet, pour les transformer en blocs exécutables et les injecter dans l'anneau dès que possible.

** Là aussi une recherche associative est indispensable pour une recherche rapide.

La charge de l'anneau devient cette fois-ci :

bloc pouvant circuler plus d'un tour	bloc circulant moins d'un tour
* blocs exécutables (BEX)	* blocs en attente (BEA)
* domaines de sortie (DS)	* demandes d'exécution (DX)

De même que pour la solution 1, la taille de l'anneau étant finie, il faut élaborer une gestion des blocs DS et BEX afin de ne pas avoir une prolifération de ceux-ci qui nous conduirait à une saturation de l'anneau.

Conclusion sur les solutions 1 et 2

Les solutions 1 et 2 doivent toutes deux utiliser un mécanisme de gestion qui peut s'avérer complexe (solution 2) pour éviter une saturation de l'anneau. Des résultats de simulation ont permis de mettre en évidence des situations de famine où les processeurs passent la plus grande partie de leur activité à attendre un bloc intéressant pour y faire un transfert. Ces situations de famine peuvent intervenir de plusieurs manières selon le type de solution retenue :

Solution 1 :

a- l'anneau peut se saturer de demandes d'exécution, cette situation ne conduit cependant pas à un interblocage car le processeur Constructeur les retire de l'anneau au fur et à mesure de leur passage devant celui-ci.

b- l'anneau peut contenir uniquement des blocs exécutables, les Processeurs d'Exécution étant tous actifs. Aucun des Processeurs d'Exécution ne peut alors déposer une demande d'exécution si la liste des demandes d'exécution en attente de dépôt est saturée pour tous les processeurs. On a donc une possibilité de blocage.

c- L'anneau se sature de blocs en attente, les proceseurs sont tous actifs et désirent effectuer l'un des dépôts suivants :
- demande d'exécution (liste saturée)
- domaine de sortie
- bloc en attente
De nouveau, il y a blocage de la machine.

Le cas b et c peuvent être détectés facilement par un processeur spécialisé appelé le Gérant d'anneau (voir chapitre VI de cette thèse)

Solution 2 : a- situation identique au cas a de la solution 1
b- situation identique au cas b de la solution 1
c- l'anneau ne contient que des domaines de sortie et tous les processeurs sont soit inactifs ou soit en attente.

Les situations b, c sont détectables aussi par le Gérant d'anneau.

3ème mode : DS et BEA dans mémoire propre du Mise à Jour

Les solutions 1 et 2 nous ont montré une des faiblesses de notre implémentation inhérente à la taille finie de l'anneau de mémoire circulante. Deux objectifs primordiaux ont été pris en considération pour la définition d'une troisième implémentation possible du processeur Mise à Jour :

a- Ne pas surcharger l'anneau de mémoire circulante ; c'est-à-dire qu'il faut extraire de l'anneau tous les blocs qui ne peuvent pas être consommés en moins d'un tour d'anneau afin de réduire les états attente des processeurs

b- La mise à jour devra être aussi rapide que possible, c'est-à-dire en temps réel (i.e à la vitesse de la mémoire circulante).

Les solutions 1 et 2 sont incompatibles avec le critère a, tandis que la solution 3 ne laisse subsister dans l'anneau que des blocs pouvant être consommés par les Processeurs d'Exécution.

La figure 13 représente un diagramme temporel possible pour la production d'un bloc exécutable.

L'objectif a permet de réduire les temps d'attente pour dépôt tandis que l'objectif b minimise le temps pour l'opération de mise à jour, les temps de propagation étant quant à eux fixes et liés à la longueur de l'anneau.

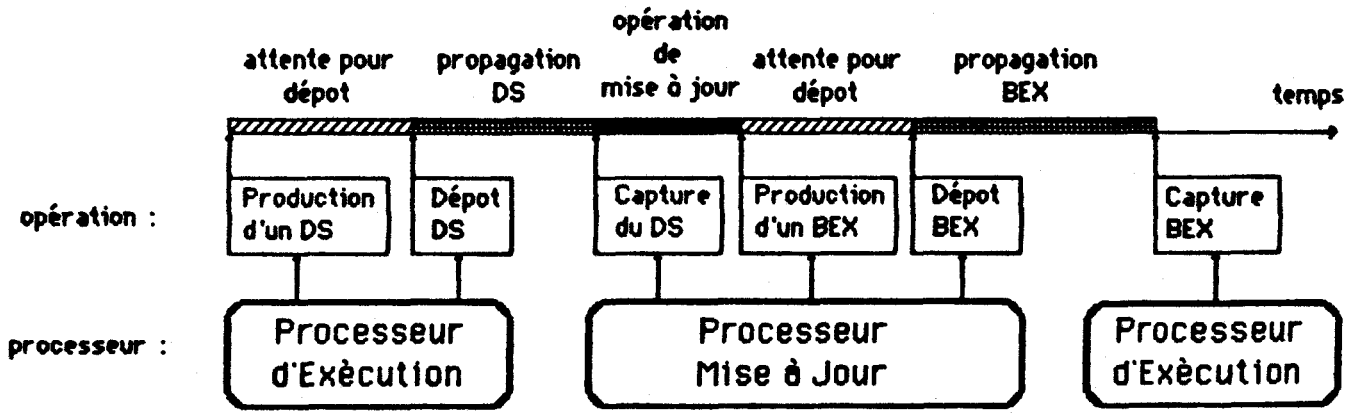


Fig. 13 Production d'un bloc exécutable

La charge de l'anneau dans l'optique de la solution 3 sera :

bloc pouvant circuler plus d'un tour	bloc circulant moins d'un tour
- blocs exécutables	- blocs en attente - domaines de sortie - demandes d'exécution

Cette solution laisse subsister néanmoins un risque de saturation de l'anneau par des blocs exécutables, lorsque tous les Processeurs d'Exécution sont dans un état actif ou attente (voir tableau figure 13 chapitre II) et qu'ils désirent déposer une demande d'exécution dans l'anneau.

Pour pallier à ce risque de blocage on peut concevoir d'implémenter un processeur spécialisé le Gérant d'anneau qui ne laisserait dans l'anneau qu'un nombre limité de blocs exécutables ayant une forte probabilité d'être capturés par les Processeurs d'Exécution en moins d'un tour d'anneau. Le chapitre VI sur la simulation présente un fonctionnement possible de ce processeur spécialisé.

Algorithmes du processeur Mise à Jour

Chaque fois qu'un bloc nouveau se présente sous la fenêtre d'accés du processeur Mise à Jour, celui-ci consulte l'entête du bloc et déroule un algorithme particulier (voir algorithme a)

```
Algorithme a    Type bloc : [BEX, DX, DS, BEA, Vide]
                CAS type bloc dans
                BEX, DX : ne rien faire
                DS : algorithme b (traitement de DS)
                BEA : algorithme c (traitement de BEA)
                Vide : algorithme d (traitement de cadre vide)
                Fin CAS
```

avec BEX : bloc exécutable
DX : demande d'exécution d'un bloc
BEA : bloc en attente
DS : Domaine de sortie
Vide: Cadre vide

Pour effectuer ses opérations de mise à jour le processeur Mise à Jour disposera d'une mémoire propre constituée de trois sous-ensembles :

- une mémoire (MDS) contenant les objets de sortie (couples (NC, VAL)) non consommés

- une mémoire (MBEA) des blocs en attente qui ont été produits
- une mémoire (MBEX) renfermant les blocs en attente qui sont devenus exécutables par suite des opérations de mise à jour.

Nous pouvons maintenant décrire les algorithmes qui régissent le comportement du processeur Mise à Jour. Aucune hypothèse n'est faite sur la méthode d'accés à ces mémoires. Celle-ci sera étudiée dans la paragraphe implémentation du Mise à Jour.

Algorithme b : * traitement de DS *

début

tant que NON fin du domaine de sortie

faire Acquérir le couple suivant (NC, VAL) ;

Si ∃ (NC), non évalué) ε MBEA*

alors transférer le couple (NC, VAL) dans le domaine d'entrée

Si domaine d'entrée est complet

Alors - modifier l'entête du BEA par "BEX"
- transférer le nouveau BEX dans MBEX
- retirer le BEA de MBEA

Fsi

Sinon transférer le couple (NC, VAL) dans MDS

fsi

fait

Fin

* Note : si l'on veut des temps de recherche rapides ces accés devront se faire associativement

Algorithme c : Le bloc courant est un bloc en attente. On effectue la mise à jour de son domaine d'entrée puis on teste si celui-ci est complet.

```
Début : * Traitement de BEA *

        Pret : = vrai ;

        Tant que NON fin domaine d'entrée

        faire  acquérir le prochain couple (NC, non évalué)

               Si ⌉ (NC, VAL) ε MDS* alors

                  transférer dans domaine d'entrée (NC, VAL) ;
                  Retirer (NC, VAL) de MDS

               Sinon  laisser dans domaine d'entrée (NC, non
                      évalué) ;
                      pret : = faux ;

               fsi

        fait

        Si NON prêt      alors
            Transférer le bloc BEA en mémoire MBEA

        Sinon  Modifier l'entête du bloc BEA par "BEX"

        Fsi

Fin
```

Algorithme d Lorsque le type du bloc courant est un cadre vide le processeur Mise à Jour insère dans l'anneau un bloc exécutable.

```
Début   * traitement de cadre vide *

        Si   MBEX NON vide    alors

           - Acquérir le prochain bloc BEX de MBEX
           - l'injecter dans le cadre vide
           - retirer le BEX de MBEX

        Fsi

Fin
```

C. Architecture du processeur Mise à Jour

Pour les raisons invoquées dans l'introduction de ce chapitre, les opérations de mise à jour devront s'effectuer en temps réel, c'est-à-dire à la vitesse de défilement de l'anneau (1 Mhz). Le processeur devra profiter de la lenteur relative de l'anneau pour faire les opérations de transfert décrites dans les algorithmes précédents. Il nous a semblé nécessaire d'utiliser des mémoires associatives pour accéder aux informations requises, compte tenu des impératifs de vitesse.

* L'algorithme b nous suggère l'utilisation d'une première mémoire associative "TCAS" (Table des relations de consommation à satisfaire) dans laquelle sont mémorisées l'ensemble des relations de consommation (NC, non évalué) présentes dans la mémoire MBEA. Cette mémoire comprend pour chacun des noms de communication NC qui s'y trouve, un pointeur désignant l'endroit où est mémorisé le couple (NC, non évalué) en mémoire MBEA et un numéro identifiant le bloc en attente dans lequel il intervient.

* L'algorithme c nécessite pour son exécution une seconde mémoire associative "TPU" (Table des relations de production utilisables) dans laquelle on trouve l'ensemble des relations de production (NC, valeur) non utilisées au cours de l'algorithme b. A chaque nom de communication NC qui s'y trouve, est associé un pointeur repérant le couple (NC, valeur) en mémoire MDS.

* La phase de détection du domaine d'entrée complet présente dans l'algorithme b et c nous a conduit à définir une troisième mémoire associative "MCOMPT" (mémoire de compteurs) qui, pour chaque bloc BEA présent dans la mémoire MBEA répertorie le numéro de ce bloc ainsi que le nombre de relations de consommation non encore évaluées. Le numéro d'un bloc est attribué par le processeur Mise à Jour lorsque celui-ci décide de le capturer. Ce numéro est restitué lorsque le bloc devenu bloc exécutable est candidat pour son insertion dans l'anneau de mémoire circulante.

Le numéro de bloc et non le nom du bloc BEA est nécessaire par le fait que plusieurs blocs BEA de nom identique peuvent être présents en mémoire MBEA (propriété de réentrance des machines dirigées par les données de type dynamique).

La figure 14 représente ces trois mémoires associatives nécessaires au déroulement des algorithmes précédemment décrits du processeur Mise à Jour.

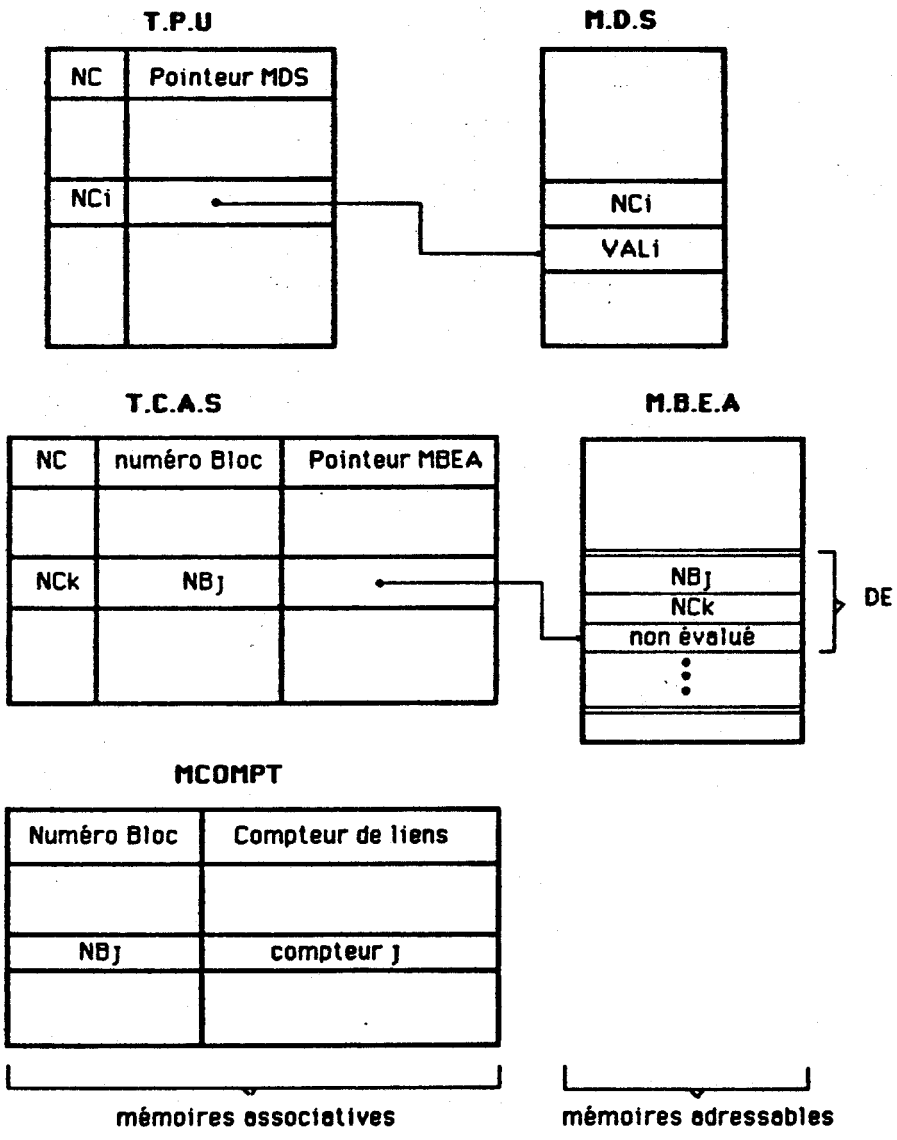


fig 14. Mémoires du processeur Mise à Jour

D. Implémentation du processeur Mise à Jour

D.1 Introduction

Le modèle de MAUD décrit au chapitre II fait état d'un mécanisme d'assignation unique des objets qui sont communiqués entre les différents blocs de MAUD. Un objet reçoit au plus une valeur pendant l'exécution d'un programme et la communication de sa valeur se fait à l'aide d'un nom de communication unique.

Nous avons vu dans les paragraphes précédents que les accés sur les mémoires associatives T.P.U. et T.C.A.S. se faisaient au moyen du nom de communication de l'objet que l'on est en train de traiter. Bien que l'usage de mémoire associative soit agréable pour tout mécanisme nécessitant une recherche rapide sur un grand jeu de données, celles-ci n'existent pas sur le marché en taille suffisamment importante pour qu'on puisse envisager de les utiliser dans ce genre d'application.

La table 1 donne les caractéristiques des mémoires associatives les plus couramment utilisées à ce jour

Désignation	Organisation	temps d'accés	technologie	boitier
F100142	4 x 4	2,7 µs	ECL	24
10155	8 x 2	17 µs	ECL	18
3104	4 x 4	30 µs	TTL	24
N822013	4 x 2	65 µs	TTL	16
SCM 5533	8 x 8	250 µs	CMOS	48

Table 1 Mémoires associatives commercialisées

On peut remarquer que le module associatif 8 x 8 de par son temps d'accés (25 ans) ne peut convenir pour des applications nécessitant un temps de recherche rapide (* 10 ns).

La difficulté à trouver sur le marché des composants une mémoire associative de grande capacité (dizaine de kbits) provient de deux raisons majeures :

* La nécessité d'intégrer un nombre important de comparateurs élémentaires. Pour une mémoire de 1k 8 bits il faudrait intégrer 8k comparateurs. Néanmoins, l'utilisation des techniques VLSI laissent à présumer que cette difficulté devrait disparaître dans un proche avenir.
* La limitation du nombre de pattes de circuit. D'une manière générale une mémoire associative opérant sur des mots de M bits, comporte :

* en pattes d'entrée : N bits pour l'opérande recherché
  N bits pour le masque

* en pattes de sortie : 1 bit indiquant que l'opérande recherché est présent

  M bits d'information associé à l'opérande trouvé

C'est pourquoi vraisemblablement on ne peut envisager d'intégrer des mémoires associatives de capacité équivalente aux mémoires à accés aléatoire, sur un seul boitier standard.

Pour pallier à ce problème, un chercheur de l'Université de DORTMOUND [TAV 82] en Allemagne a envisagé une organisation de mémoire associative différente qui permet une extensibilité comparable aux composants de mémoire classique. Cependant, la sophistication et le temps de réponse de cette mémoire nous a conduit à l'écarter de notre implémentation.

Un second palliatif aux mémoires associatives est l'utilisation de mémoire RAM à temps d'accés très faible, on parlera alors de mémoire pseudo-associative dans lesquelles la donnée recherchée est utilisée comme adresse. I.Watson et J.G.D.Silva [WAT 83] ont défini une mémoire pseudo-associative qui utilise un mécanisme d'adressage de type "hash coding" permettant une mise à jour rapide pour une implémentation sur machine dirigée par les données. C'est à ce second type de mémoire associative que nous allons nous intéresser.

Les noms de communication, utilisés dans MAUD étant en nombre limité, sont récupérés au fur et à mesure de leur consommation (voir chapitre II).

Le mécanisme d'assignation unique sous-jacent à MAUD permet d'envisager une implémentation possible de ces mémoires à l'aide de mémoire conventionnelle dans laquelle le nom de communication est utilisé pour sélectionner une position mémoire. Bien évidemment ces mémoires conventionnelles devront être suffisamment rapides vis-à-vis de la vitesse de l'anneau afin d'autoriser un traitement du bloc en temps réel c'est-à-dire pendant son défilement sous la fenêtre du processeur Mise à Jour. Nous parlerons dans ce qui suit non plus de mémoires associatives mais de mémoire pseudo-association.

D.2 Contraintes physiques

Les contraintes physiques que nous avons considérées pour la définition d'une implémentation du processeur Mise à Jour sont de deux types :

* contraintes liées à la technologie de la mémoire circulante
* contraintes liées à la structure des blocs manipulés par le processeur Mise à Jour

Format des blocs manipulés par le Mise à Jour

Le comportement du Mise à Jour est déterminé de manière univoque par le type du bloc présent sous sa fenêtre d'accés. Afin de comprendre les algorithmes qui régissent son fonctionnement il est nécessaire de se rappeler le format des blocs qu'il est amené à manipuler.

Un bloc "domaine de sortie" est constitué comme le montre la figure 15a

* d'une partie entête caractéristique de ce bloc
* d'une partie renfermant des couples (NCi, VALi)
* d'un indicateur NIL qui délimite la fin logique du bloc à l'intérieur d'un cadre physique de 1 k mots de 16 bits.

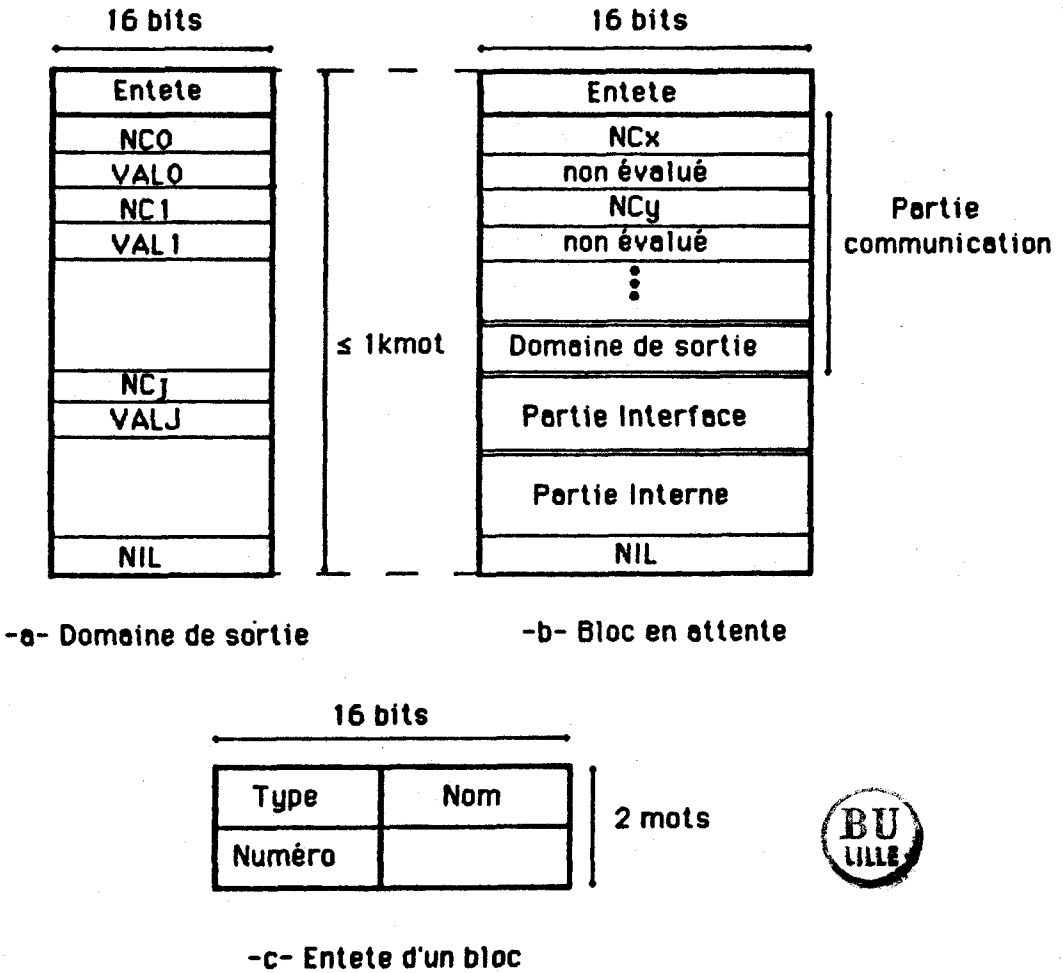


Fig. 15

Note * : le champ valeur peut occuper 1 ou plusieurs mots en fonction du type de l'objet. Le type est précisé dans le champ NC

La figure 15b représente la structure générale d'un bloc en attente (BEA). Elle comprend :

* une partie entête identificatrice
* une partie communication constituée du domaine d'entrée et du domaine de sortie. Le domaine d'entrée répertorie les relations de consommation (NC, non évalué) que le processeur mise à jour devra satisfaire pour que le bloc puisse devenir un bloc exécutable
* une partie interface qui permet la transmission de valeur entre la partie interne et la partie communication
* une partie interne renfermant le domaine propre et un ensemble d'instructions agissant sur les objets du domaine propre
* un indicateur NIL qui délimite le bloc.

L'entête commune à tous les blocs qui circulent dans MAUD est détaillée figure 15c et se compose :

* un champ TYPE qui précise la nature du bloc (BEX, BEA, DS etc...)
* un champ NOM de bloc qui identifie le bloc dans la bibliothèque des blocs
* un champ NUMERO permet une numérotation des blocs.

Contraintes technologiques de la mémoire circulante

D'après la structure des blocs que le processeur Mise à Jour manipule, il est indispensable d'effectuer les opérations de mise à jour selon le diagramme de la figure 16 et 17.

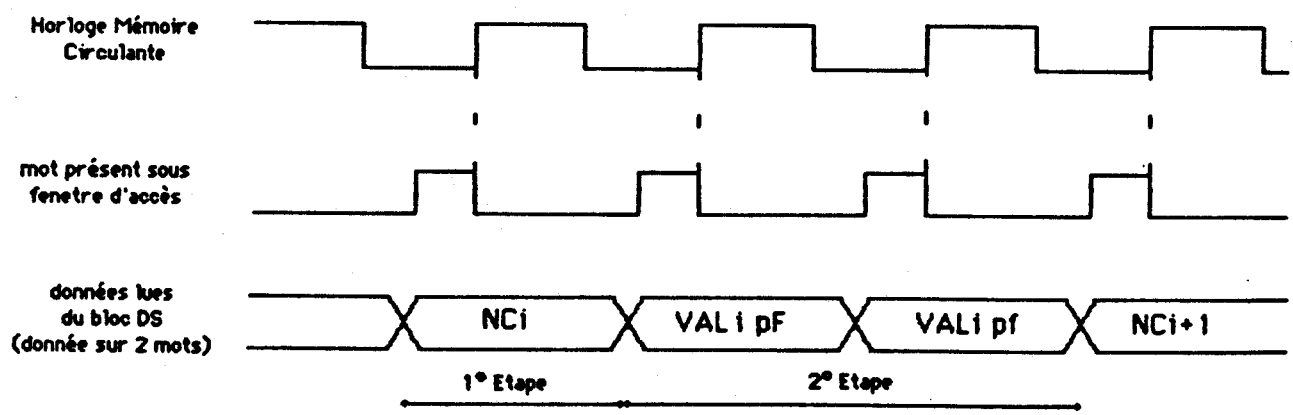


Fig. 16 Lecture d'un bloc domaine de sortie

La scrutation d'un domaine de sortie se déroule en deux étapes:
- La première étape (1 cycle d'horloge) permet d'effectuer un accés associatif par le nom de communication NC à l'intérieur des mémoires pseudo-associatives T.P.U et T.C.A.S. A l'issue de cet accés on peut décider du type de transfert à effectuer pour la seconde étape.
- La seconde étape réalise le transfert de la valeur associée au nom de communication vers la mémoire MDS ou la mémoire MBEA. Cette étape peut durer un ou plusieurs cycles d'horloge selon le type de la valeur à traiter (entier, réel, simple mot, double mot etc...)

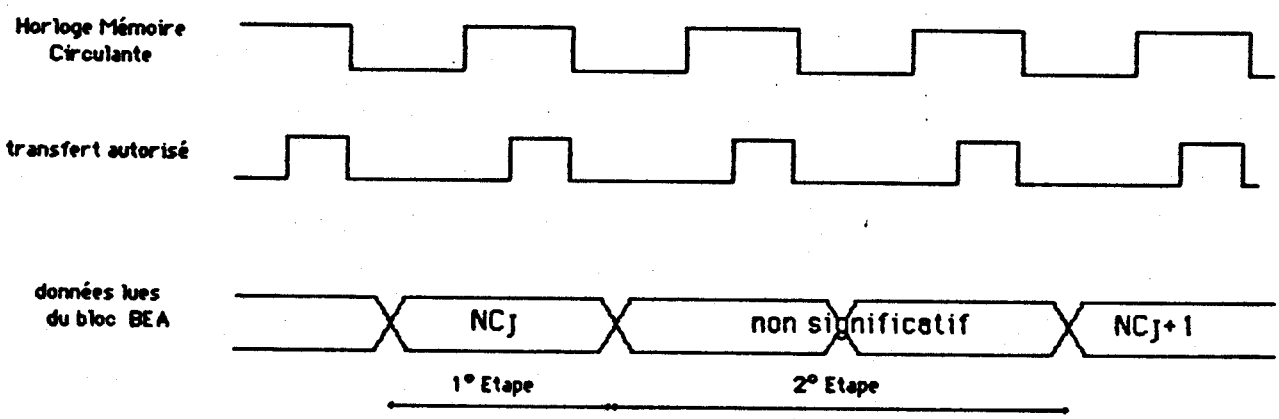


Fig.17 lecture d'un bloc BEA

La lecture d'un bloc en attente se fait également en deux étapes. La première étape est identique au cas précédent, tandis que la seconde étape réalise l'écriture en mémoire circulante de la valeur associée au nom de communication lu dans l'hypothèse ou la première étape a permis d'établir son existence en mémoire MDS. L'écriture est synchronisée par le signal transfert autorisé.

D.3 Implémentation de la T.P.U
(table des relations de production utilisables)

Nous avons vu précédemment que cette table devait nous informer sur l'ensemble des couples (NCi, VALi) présents dans la mémoire MDS. La mémoire MDS ne contenant que des informations simples, nous avons choisi de fusionner celle-ci à l'intérieur de la T.P.U afin d'éviter un adressage indirect pour accéder à une information VALi.

Trois types d'informations seront nécessaires :

* le nom de communication NCi
* le type de l'opérande (entier, réel, ...)
* la valeur VALi de l'opérande

L'information NCi étant utilisée pour adresser la mémoire pseudo-associative T.L.U, elle n'a plus de raison d'être dans l'implémentation. Cependant, afin de savoir si cette information est effectivement présente, un bit de présence devient nécessaire (BP_i). La figure 18 donne l'implémentation choisie pour cette mémoire.

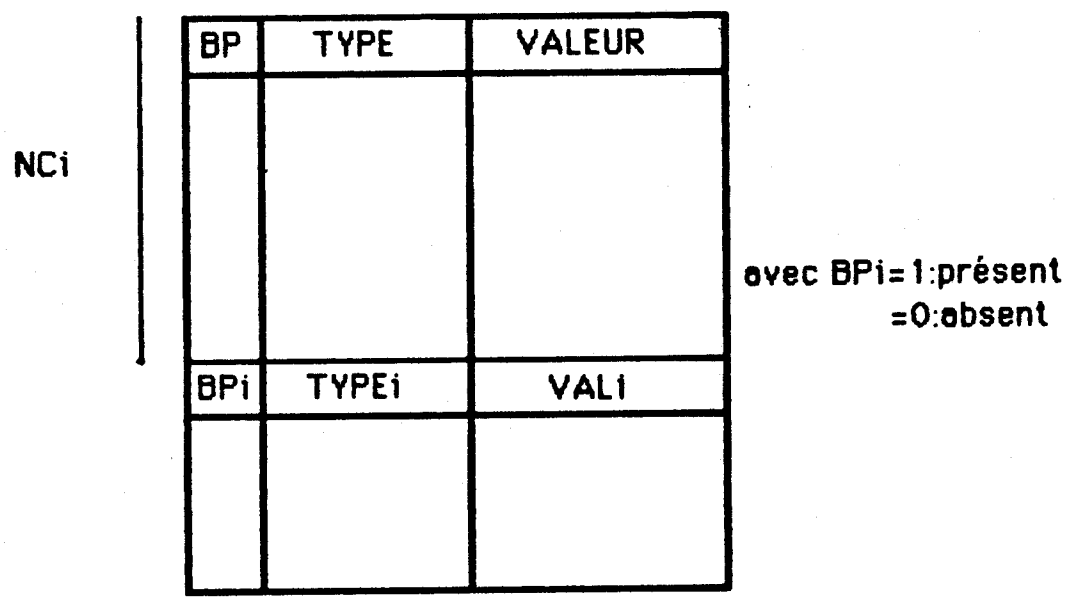


Fig. 18 T.L.U

D.4 Implémentation de la T.C.A.S
(Table des relations de consommation à satisfaire)

Cette mémoire pseudo-associative contient les couples (NC_i, non évalués) relevés au cours de l'algorithme c décrit précédemment. Le caractère complémentaire des mémoires T.P.U et T.C.A.S nous suggère de fusionner ces deux mémoires en une seule table T.L (table de liens). Cette table contient pour chaque nom de communication NC_i référencé (Fig.19) :

* un bit de présence BP
* un champ type de l'opérande
* un champ VAL_i destiné à recevoir la VALEUR de l'opérande
* le numéro du bloc BEA_i dans lequel intervient ce nom de communication

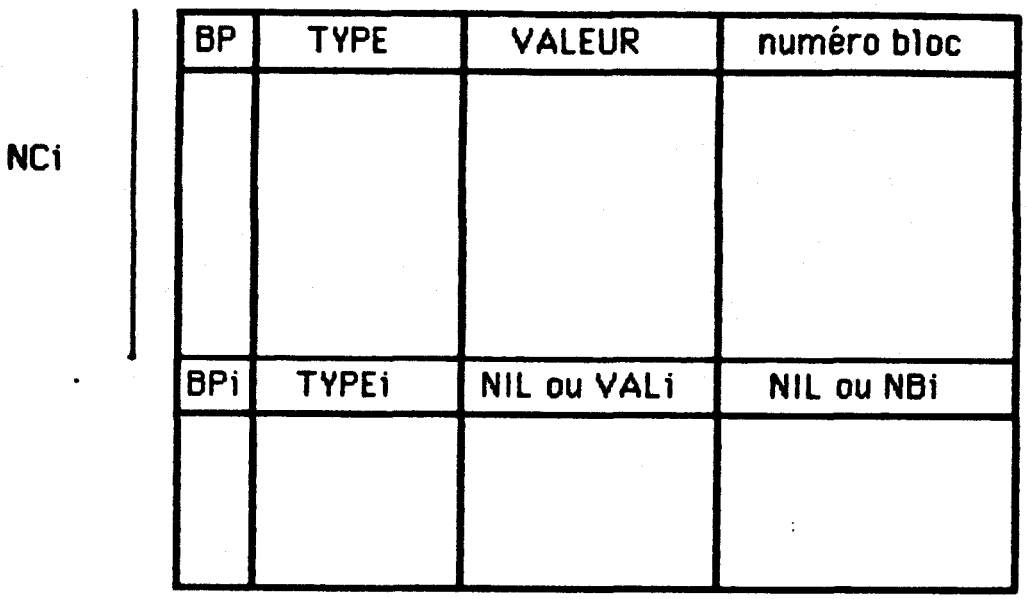


Fig.19 T.L (Fusion de T.L.U et T.C.A.S)

Les noms de communication NC_i obéissant à la règle d'assignation unique, il ne peut y avoir que deux accés au plus à une information référencée par NC_i. Une première fois pour relever un couple de type (NC_i, VAL_i), une seconde fois pour relever une référence non satisfaite (NC_i, non évalué) ou inversement. Le tableau de la figure 20 permet de comprendre le mécanisme de cette table utilisée au cours des algorithmes b, c.

Type bloc scruté	BP à l'instant t	Description	BP à l'instant t+1
DS	0	On relève un couple (NCi,VALi)	1
DS	1	On satisfait une référence	0
BEA	0	On relève un couple (NCi,non évalué)	1
BEA	1	On satisfait une référence	0

Fig. 20 Mécanisme de la table T.L

Cette mémoire est couplée à une troisième mémoire associative MCOMPT qui pour chaque numéro de bloc BEA, présent dans la table T.L, comptabilise le nombre de références manquantes (objets d'entrée non évalués).

Le principe de cette mémoire est le suivant :

Au cours d'une scrutation d'un bloc DS et pour chaque nom de communication NC_i référencé dans celui-ci, on fait un accés associatif à la mémoire T.L pour y mémoriser la valeur VAL_i associée. Si le bit BP de présence est positionné à zéro (valeur non encore attendue) BP est mis à un.Dans le cas contraire, BP est mis à zéro puis parallélement à cette opération on décrémente d'une unité la valeur des références manquantes du bloc BEA correspondant (Figure 21).

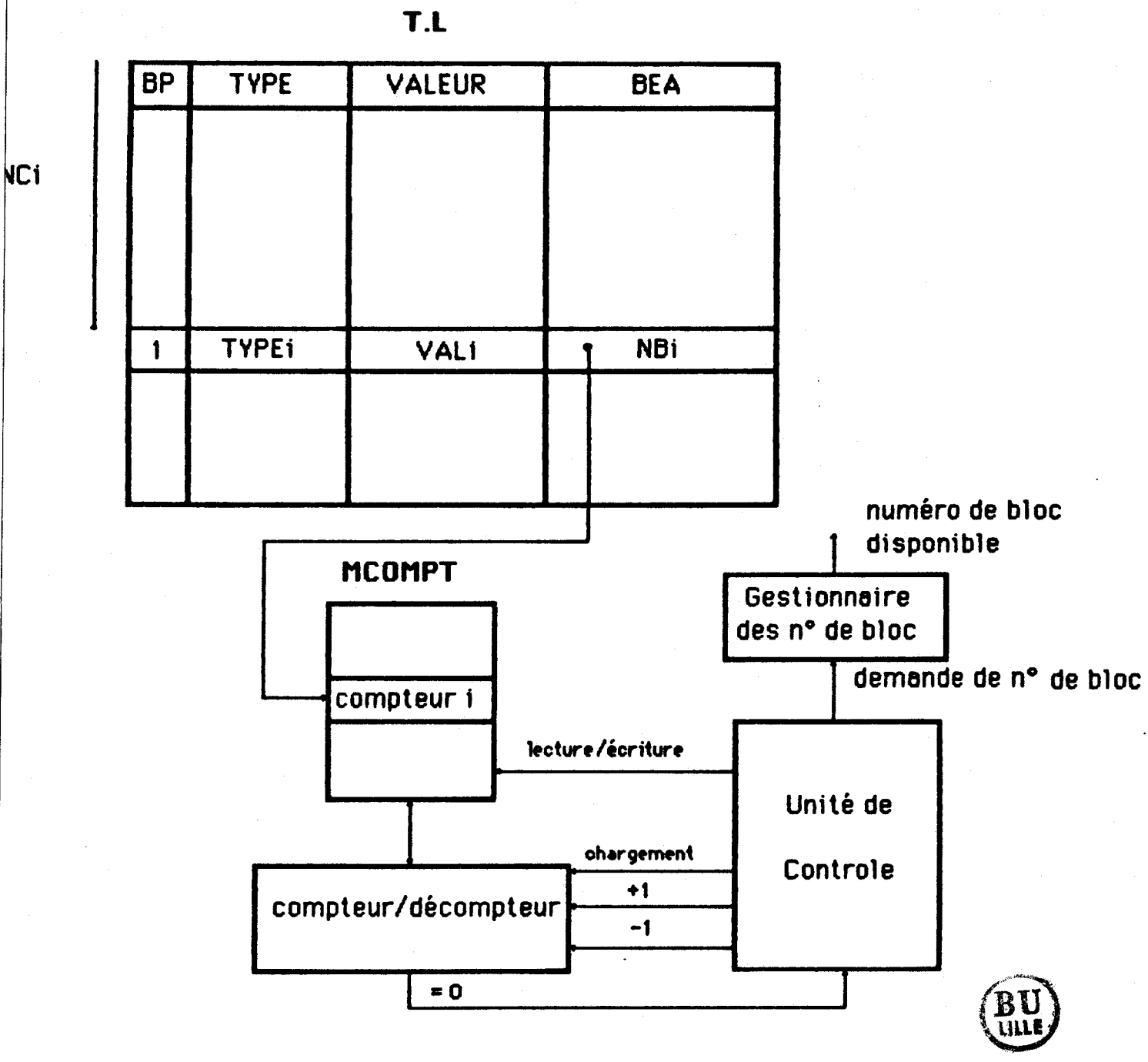


Fig. 21 Mémoire T.L et Table de Compteurs

Le passage à zéro de ce compteur indique à l'unité de contrôle que le bloc BEA_i à son domaine d'entrée complet. L'unité de contrôle envoie alors le numéro NB_i du bloc vers une seconde unité qui sera chargée de construire un bloc exécutable BEX_i à partir du bloc BEA_i (présent dans la mémoire MBEA) et de son domaine d'entrée (présent dans la T.L).
La scrutation d'un bloc DS pouvant donner lieu à la production d'un ou plusieurs blocs exécutables, les numéros de ces blocs sont gérés en file (fig. 22).

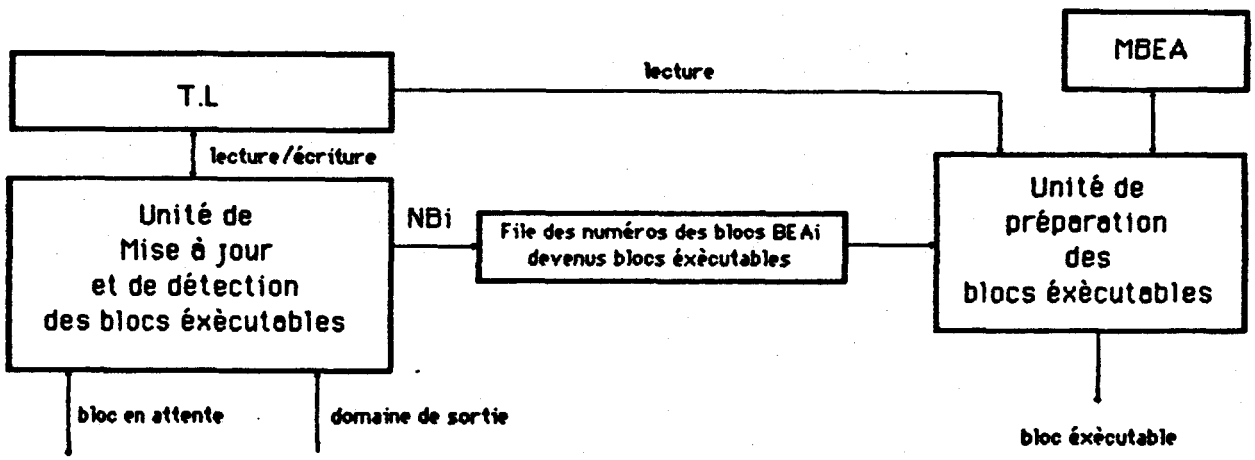


Fig. 22 File des numéros de bloc

L'unité de Mise à Jour et de détection des blocs exécutables comprend les modules suivants de la figure précédente :

- -Mémoire de compteurs
- -Compteur/Décompteur de références
- -Gestionnaire des numéros de bloc
- -Unité de contrôle qui pilote l'ensemble

Dans l'hypothèse où le bloc scruté est un bloc en attente, la procédure d'accés à cette table T.L devient la suivante :

Pour chaque nom de communication appartenant à une relation de consommation (NC, non évalué) qui figure dans le domaine d'entrée du bloc, on fait un accés associatif dans la table T.L afin de déterminer si une relation de production (NC, Valeur) existe. Dans le cas positif (Bit de présence positionné à 1) la valeur associée au nom de communication est transmise au bloc BEA suivant le diagramme d'écriture présenté dans le paragraphe précédent et le Bit de présence est mis à zéro. Dans le cas contraire (Bit de présence positionné à 0) le bit de présence est positionné à 1 le type de l'opérande attendu est mémorisé, ainsi que le numéro du bloc qui lui a été attribué par le gestionnaire des numéros de bloc. Le numéro du bloc est fourni lors de la première référence à la table T.L non satisfaite. Il est restitué lorsque le bloc devenu exécutable est inséré dans l'anneau. Une liste circulaire permet de s'affranchir des problèmes de sa gestion.

Simultanément à l'accés dans la T.L, on accède à la mémoire de compteurs et on incrémente, le compteur associé au numéro de bloc, d'une unité pour chaque référence non satisfaite.

A l'issue du passage complet du domaine d'entrée du bloc scruté le processeur Mise à Jour peut décider de capturer le bloc BEA si celui-ci n'a pas un domaine d'entrée complet. Lorsque le domaine d'entrée est complet le bloc BEA est laissé dans l'anneau et le processeur Mise à Jour doit procéder à la modification de son entête.

D.5 Modification de l'entête du bloc

Si le bloc que l'on est en train de scruter est un domaine de sortie il faut modifier son entête en inscrivant comme nouveau type "cadre vide". Cette modification peut se faire aisément grâce aux opérations d'échange entre Automate d'Accés et Mémoire Circulante (Lecture suivie d'une écriture). Par contre, lorsque le bloc traité est un BEA on ne peut dire à l'avance si celui-ci va devenir exécutable par suite de l'algorithme c.

Plusieurs solutions sont envisageables :

a- La mise à jour est effectuée lors du prochain passage du bloc devant le Mise à jour. Si le bloc que l'on a mis à jour devient exécutable, alors le mise à jour conserve son nom et lorsque celui-ci passe pour la 2ème fois devant la fenêtre d'accés du Mise à jour, on fait la modification de l'entête. Cette solution bien que très simple pénalise la charge de l'anneau (critère a de notre sélection non satisfait)

b-Action processeur gérant : Le processeur Gérant situé avant le Mise à Jour capture tous les blocs BEA et ne les réinjecte que si toutes les valeurs attendues par leur domaine d'entrée sont parvenues au processeur Mise à Jour. Celui-ci peut donc faire la transformation BEA → BEX de façon systématique pour chaque bloc BEA qu'il scrute. Le processeur Gérant devra donc comporter en mémoire locale une table T.L analogue à celle décrite pour le processeur Mise à Jour. La charge de ce processeur en sera accrue, et son existence devient nécesaire. Il faut remarquer que la fonction gérant est alors physiquement indépendante de la fonction Mise à Jour (Fig. 23).

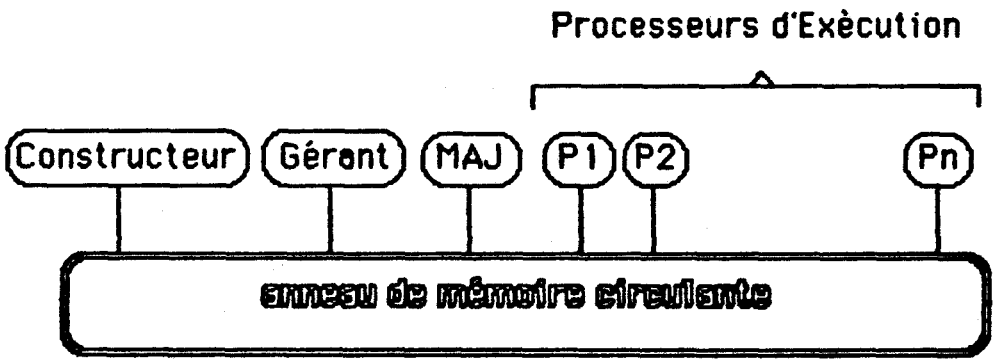


Fig. 23 Configuration du système

c- Action processeur n°1 (P_1)
Dans ce cas c'est le premier Processeur d'Exécution qui suit le processeur Mise à Jour qui modifie systématiquement le bloc BEA laissé par le Mise à jour. Cette solution présente d'une part l'inconvénient de mélanger la fonction de traitement du Processeur d'Exécution et la fonction mise à jour, d'autre part l'activité du Processeur d'Exécution risque d'être sévèrement ralentie.

d- Le processeur Mise à jour dispose de 2 fenêtres d'accés consécutives sur l'anneau

La première fenêtre permet d'effectuer les opérations de mise à jour décrites précédemment.

La seconde fenêtre permet deux types opérations (Fig. 24)

- la première consiste à modifier l'entête d'un bloc BEA lorsque son passage devant la première fenêtre a permis de détecter que celui-ci est devenu bloc exécutable. Dans le cas contraire, le bloc BEA est transféré en mémoire MBEA du processeur Mise à Jour.

- La seconde opération permet d'insérer un bloc exécutable dans l'anneau lorsque le type du bloc est "cadre vide" ou un bloc BEA que l'on a décidé de capturer.

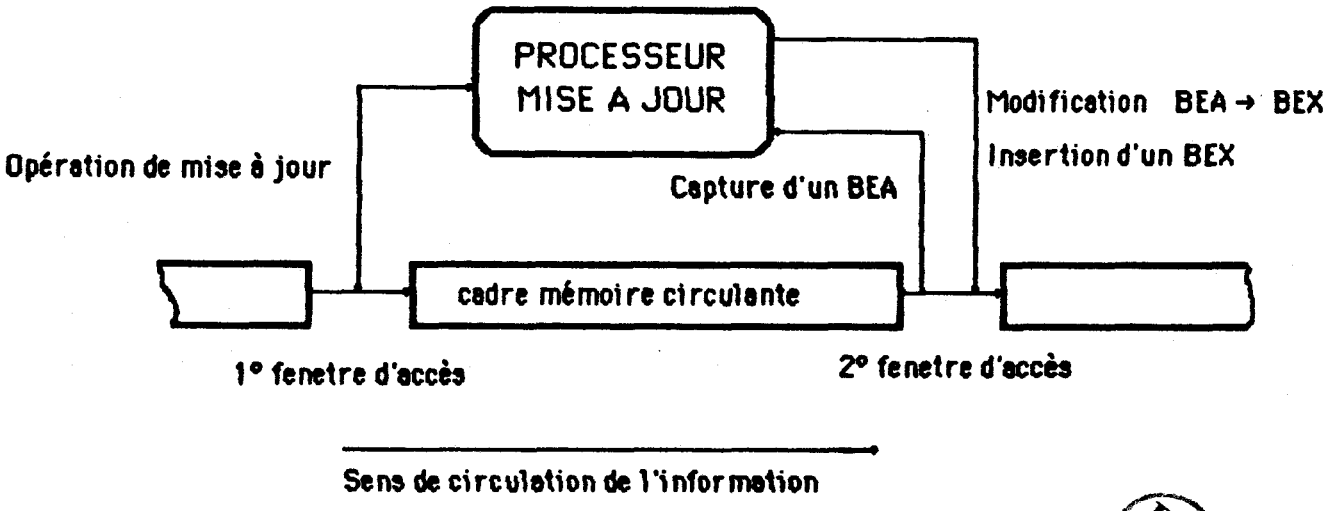


Fig. 24 Les fenêtres du processeur Mise à jour

Cette dernière solution nous a paru préférable aux autres par ses aspects locaux et simples, la fonction mise à jour reste locale au processeur Mise à Jour. Le schéma de la figure 25 donne la structure générale du processeur Mise à Jour. Nous y trouvons :

-deux unités de transfert microprogrammées analogues à celles décrites dans le chapitre III de cette thèse.

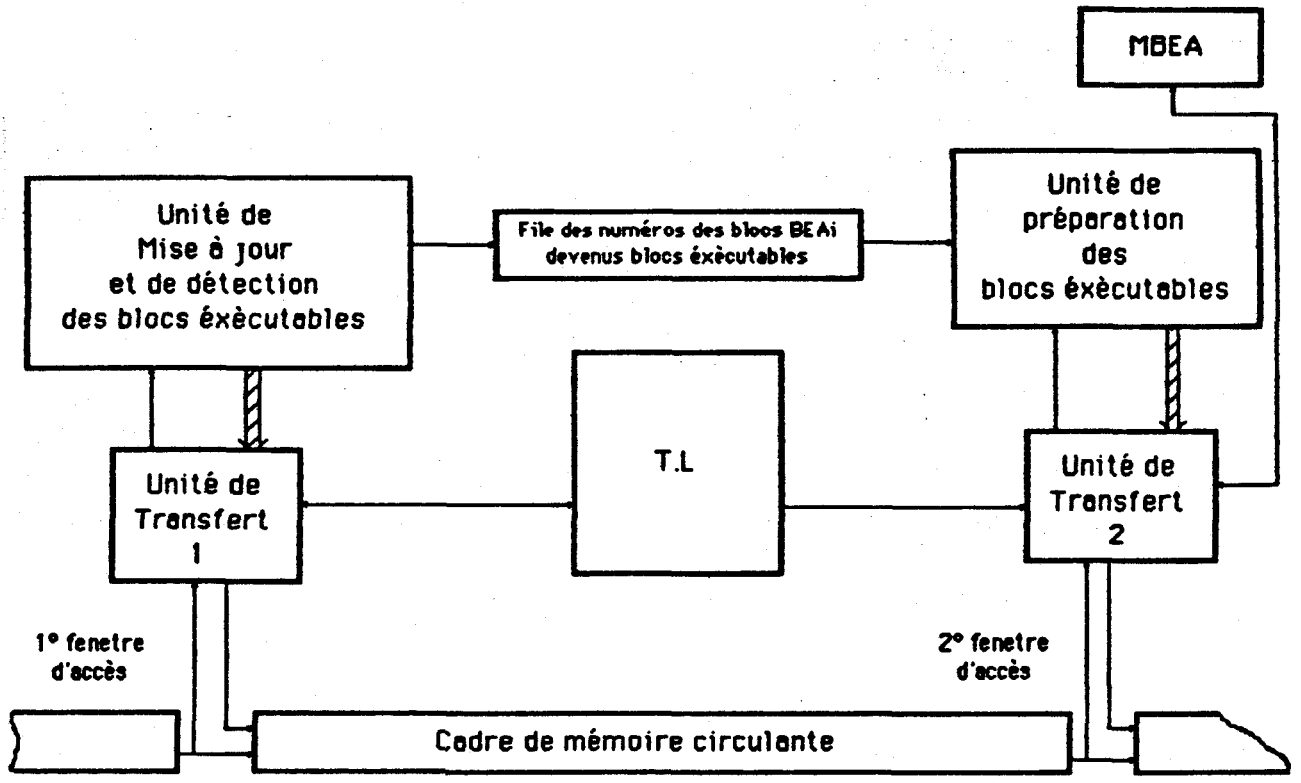


Fig. 25 Structure complète du processeur Mise à Jour

-Une unité microprogrammée qui assure la mise à jour de la table des liens ainsi que la détection des blocs devenus exécutables

-Une unité microprogrammée dont la fonction est de préparer les blocs devenus exécutables en vue de leur insertion dans l'anneau de mémoire circulante

-Une file destinée à recevoir les numéros des blocs devenus exécutables

-Une mémoire T.L qui mémorise les relations de communication entre les blocs

-Une mémoire MBEA qui renferme les blocs en attente

Comparativement aux architectures que nous avons étudiées dans le paragraphe 1 de ce chapitre, nous pouvons dresser les caractéristiques de notre processeur Mise à Jour :

-Mécanisme dirigé par les données de type dynamique

-Structure à trois mémoires (MBEA, T.L, MCOMPT)

-Fonction mise à jour centralisée

-La structure de Mise à jour fait apparaître un "pipeline" entre l'unité de Mise à jour et l'unité de préparation des blocs exécutables

-Mécanisme de détection des blocs exécutables par compteur de références (M COMPT)

-La communication entre le processeur Mise à Jour et les Processeurs d'Exécution est du type par paquets au travers de l'anneau de mémoire circulante

CONCLUSION

L'architecture présentée dans ce chapitre permet une mise à jour des domaines d'entrée incomplets en temps réel.

Les domaines de sortie produits par les Processeurs d'Exécution sont consommés au fur et à mesure de leur passage devant la fenêtre d'accés du processeur Mise à Jour.

L'utilisation d'une mémoire pseudo-associative apparaît comme une solution préférable à une solution logicielle coûteuse en temps de recherche.

La structure microprogrammée des unités autorise des modifications ultérieures aisées.

CHAPITRE V

SIMULATION DU MODELE DYNAMIQUE

Introduction

A. Le langage de simulation

B. Le modèle dynamique

B.1. Configuration du système

B.2. Résultats de simulation

B.2.1. Premier exemple : TRI-FUSION

B.2.2. Second exemple : Système linéaire

C. Le modèle théorique avec files

C.1. Présentation

C.2. Résultats de simulation

C.2.1. Premier exemple

C.2.2. Second exemple

Conclusion

Ce chapitre présente les travaux de simulation entrepris parallèlement à la réalisation de notre prototype. Des résultats et des conclusions partiels y sont évoqués.

Introduction

Une première simulation du modèle statique présenté dans [PET 81] a permis de montrer l'influence de plusieurs paramètres sur le comportement de notre modèle. Les paramètres retenus sont les suivants :

* Nombre de Processeurs d'Exécution,

* Nombre de cadres constituant l'anneau de mémoire circulante,

* Temps d'exécution d'un bloc.

Cette simulation a montré l'importance du temps d'exécution d'un bloc par rapport au temps de circulation de ce bloc dans l'anneau. Il est apparu nécessaire de disposer d'une bibliothèque de blocs dont le temps d'exécution soit supérieur ou égal au temps de circulation de ceux-ci.

Les courbes qui y figurent montrent qu'il existe une longueur critique de l'anneau au delà de laquelle l'évolution du temps d'exécution total est linéaire (proportionnel au nombre de cadres de l'anneau).

Ces résultats nous ont guidés pour le choix du nombre de processeurs a implémenter dans la réalisation du prototype. Pour une configuration comportant une dizaine de Processeurs d'Exécution, un anneau constitué d'une quinzaine de cadres apparaît comme la longueur optimale.

L'implémentation des primitives EXECBLOC à ATTENDRE nous a conduit à l'écriture d'une simulation du modèle dynamique afin de mieux percevoir le caractère dynamique de ce modèle. Cette simulation répond à deux objectifs primordiaux :

* Connaître le temps d'exécution d'un programme d'essai en fonction des paramètres de la machine (nombre de Processeurs d'Exécution, taille de l'anneau, etc...) et le gain réalisé par rapport à une machine monoprocesseur classique.

* Fournir une trace des échanges d'informations entre les différents processeurs fonctionnels et l'anneau de communication. Cette trace permettra en outre d'avoir un état de la charge de l'anneau et d'évaluer ainsi son efficacité.

Comme nous l'avons précisé dans les chapitres précédents les échanges entre l'anneau et les processeurs de MAUD se faisant toujours au niveau d'un cadre physique, l'unité de temps, qui a été choisie pour quantifier les résultats de la simulation, est le temps nécessaire pour voir défiler la totalité d'un cadre devant la fenêtre d'un processeur. Ainsi le temps d'exécution d'un bloc séquentiel est évalué par rapport à cette unité de temps.

Un bloc à exécuter pour le simulateur MAUD sera donc une suite de traitements séquentiels (exprimés en nombre d'unités de temps) et d'appels de primitives EXECBLOC ou ATTENDRE. Un programme d'essai est constitué d'une bibliothèque de blocs accessibles par le processeur CONSTRUCTEUR.

L'ensemble des processeurs définis dans le modèle dynamique intervient dans cette simulation. A celui-ci a été rajouté un processeur supplémentaire appelé 'INJECTEUR' qui permet de délivrer au système le ou les blocs qui constituent le programme à exécuter.

Le chapitre se découpe en quatre parties. La première partie présente la syntaxe utilisée pour décrire les programmes à simuler. La seconde partie s'attache à montrer les caractéristiques du modèle dynamique. La troisième partie est une modélisation du modèle théorique qui nous permettra de pousser plus en avant nos conclusions qui constitueront la dernière partie de ce chapitre.

A. Le langage de simulation

Ce langage permet de décrire l'ensemble des blocs qui constitueront la bibliothèque de référence utilisée par le processeur CONSTRUCTEUR. Le langage utilisé est du type structuré et comporte les structures de base suivantes :

- Séquence d'actions.

- Itération.

- Appel de primitives de parallélisation.

Structure d'un bloc

Un bloc est constitué de la manière suivante :

* Un nom de bloc qui l'identifie dans la bibliothèque.

* Une partie déclaration dans laquelle figurent les relations de production et les relations de consommation relatives au bloc.

* Un corps de bloc qui décrit l'algorithme à simuler.

La figure 1 représente la structure d'un bloc.

```
BLOC : NOMBLOC ;
DE :  liste des relations de consommation ;
DS :  liste des relations de production ;
DEBUT ;
        { instruction ;}
FIN ;
```

Figure 1 : Un bloc.

Partie déclaration

La partie déclaration DE ou DS contient la liste des identificateurs des objets que l'on veut transmettre d'un bloc à un autre :

```
DE : E, N, T ;
DS : S, O, R ;
```

Corps de bloc

Le corps du bloc est délimité par les mots clés DEBUT et FIN. Les instructions du corps peuvent être l'une des quelconques instructions suivantes :

- Instruction de séquence.
- Instruction REPETER
- Instruction de lancement d'exécution d'un bloc.
- Instruction ATTENDRE.

Instruction séquence

Elle permet de simuler des parties de traitement que l'on doit exécuter de manière séquentielle. On indique alors le temps (en unités de temps) nécessaire pour exécuter cette séquence.

Exemple : [10] ;

Cette instruction correspond au traitement d'une partie de programme que l'on doit exécuter séquentiellement et dont la durée d'exécution est de 10 unités de temps.

Instruction REPETER

Afin d'alléger l'écriture de certains blocs où une partie du traitement est à réitérer un nombre de fois, l'instruction REPETER a été implémentée.

Deux variantes de cette instruction sont possibles.

1ère Version : Le nombre d'itérations est fixe et connu au moment de l'écriture du bloc.

Exemple : REPETER 10 ;

{ instruction ;} +

fin REPETER ;

2ème Version : Le nombre d'itérations est inconnu au moment de l'écriture du bloc et celui-ci peut varier en cours d'exécution d'un programme de simulation.

Exemple : REPETER * ;

{instruction ;} +

fin REPETER ;

Le nombre d'itérations sera fourni par le processeur 'INJECTEUR' au moment où celui-ci insère le nom du bloc à exécuter dans le système.

Les deux versions utilisent le mot-clé fin REPETER qui délimite la liste des instructions à itérer.

Instruction de lancement d'exécution d'un bloc

Cette instruction permet de lancer l'exécution d'un bloc de la bibliothèque parallèlement à l'exécution du bloc qui exécute cette instruction.

On précise alors le nom du bloc dont on veut lancer l'exécution ainsi que la liste des identificateurs des objets qui figurent dans son domaine d'entrée suivie de celle des identificateurs qui figurent dans son domaine de sortie.

Exemple : TRI (A, B, C ; E, F, G) ;

Le bloc de nom TRI est lancé pour exécution. Son domaine d'entrée est constitué de trois objets dont les identificateurs sont respectivement A, B et C. De la même manière les identificateurs des objets de son domaine de sortie sont respectivement E, F et G.

Il est à noter que l'on ne pourra lancer que des blocs ayant déjà été définis dans une phase antérieure.

Instruction ATTENDRE

Cette instruction permet de récupérer la valeur des objets calculés par les blocs dont on a lancé l'exécution à l'aide de l'instruction vue précédemment. Cette instruction a pour effet de suspendre l'exécution du bloc qui l'exécute tant que la liste des objets mentionnés n'est pas entièrement évaluée.

Suivant le désir de récupérer une partie ou l'ensemble des valeurs des objets cités plus haut, deux formes sont envisageables.

1ère Forme : ATTENDRE PARTIEL

Permet de récupérer la valeur des objets mentionnés dans une liste.

Exemple : ATTENDRE (A, B).

Un seul objet d'identificateur A et d'identificateur B seront récupérés.

2ème Forme : ATTENDRE TOTAL

Cette forme autorise la récupération de tous les objets associés aux mêmes identificateurs mentionnés dans une liste.

Exemple; ATTENDRE * (A, B).

Tous les objets d'identificateur A et B seront attendus.

Cette deuxième forme prend toute sa signification lorsqu'elle apparaît à l'intérieur d'un bloc du type :

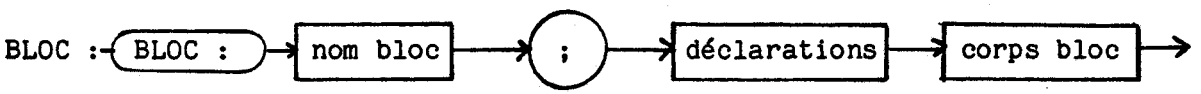
```
début ;
      [5] ;
      répéter 10 ;
            [5] ;
            TRI (A, B ; C, D) ;
      fin répéter ;
      attendre * (C, D) ;
fin ;
```

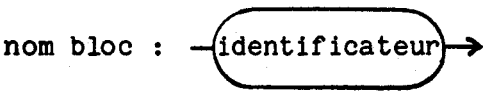
L'instruction attendre * (C, D) permettra de récupérer les valeurs associées aux dix objets d'identificateur C et aux dix objets d'identificateur D.

Il est à noter que les identificateurs qui apparaissent dans un programme sont associés à l'exécution à des noms dynamiques uniques afin de respecter la règle d'assignation unique.

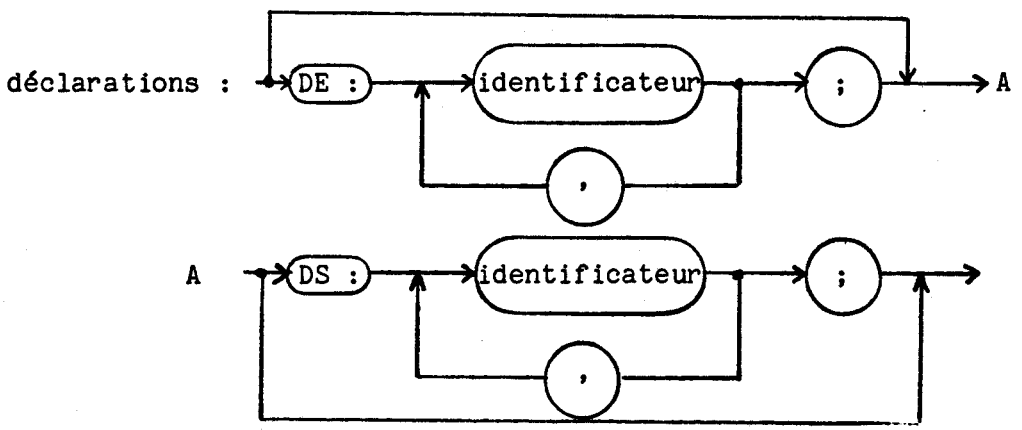
La syntaxe complète du langage de simulation est présentée sur les diagrammes qui suivent.

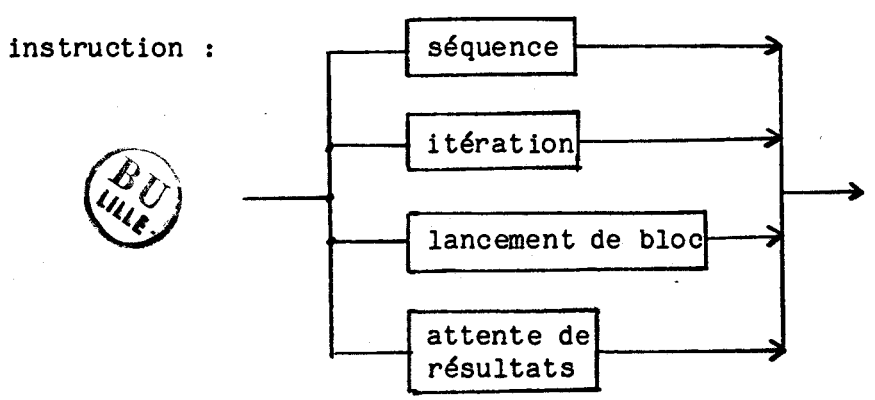
Diagrammes syntaxiques

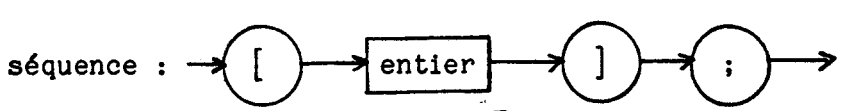


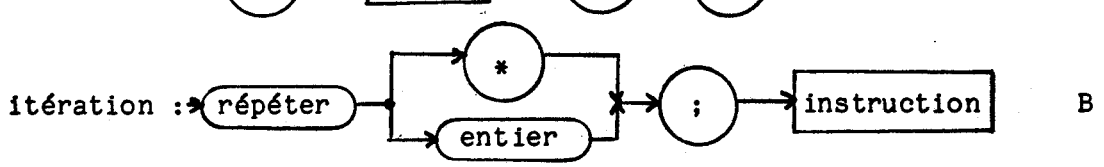


identificateur : chaîne alphanumérique de 8 caractères.



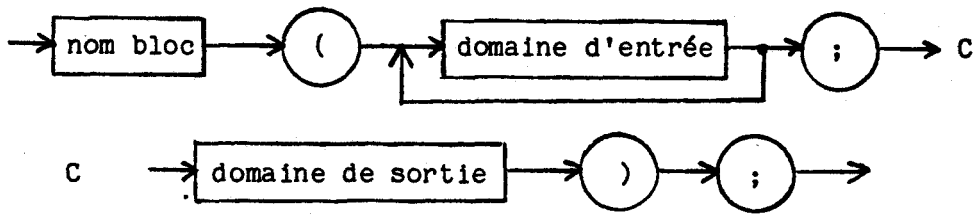






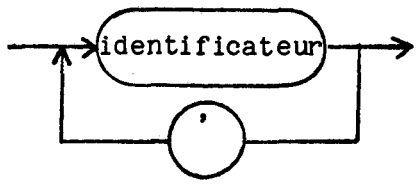
B

Lancement de bloc :

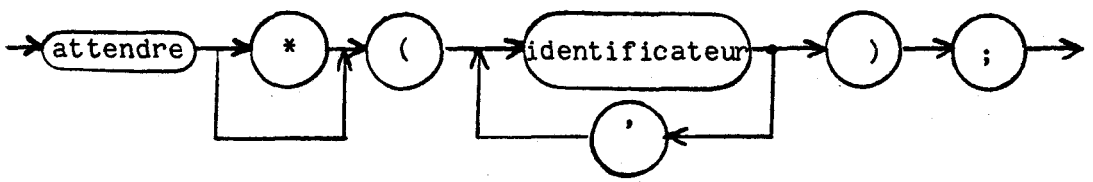


domaine d'entrée :

domaine de sortie :



attente de résultats :



Il est à noter que ces diagrammes ne peuvent représenter la condition sémantique associée à l'instruction "attente de résultats" qui est la suivante : Une instruction "attente de résultats" ne peut attendre que des résultats qui seront fournis par des blocs dont on a lancé auparavent l'exécution par l'intermédiaire d'une instruction "lancement de bloc".

B. Le modèle dynamique

B.1. Configuration du système

La configuration du modèle dynamique ainsi simulée apparaît sur la Figure 2. La structure pipeline qui apparaissait au niveau de la définition du modèle a été conservée de manière à réduire les temps de communication entre les différents sites dus à la propagation dans l'anneau.

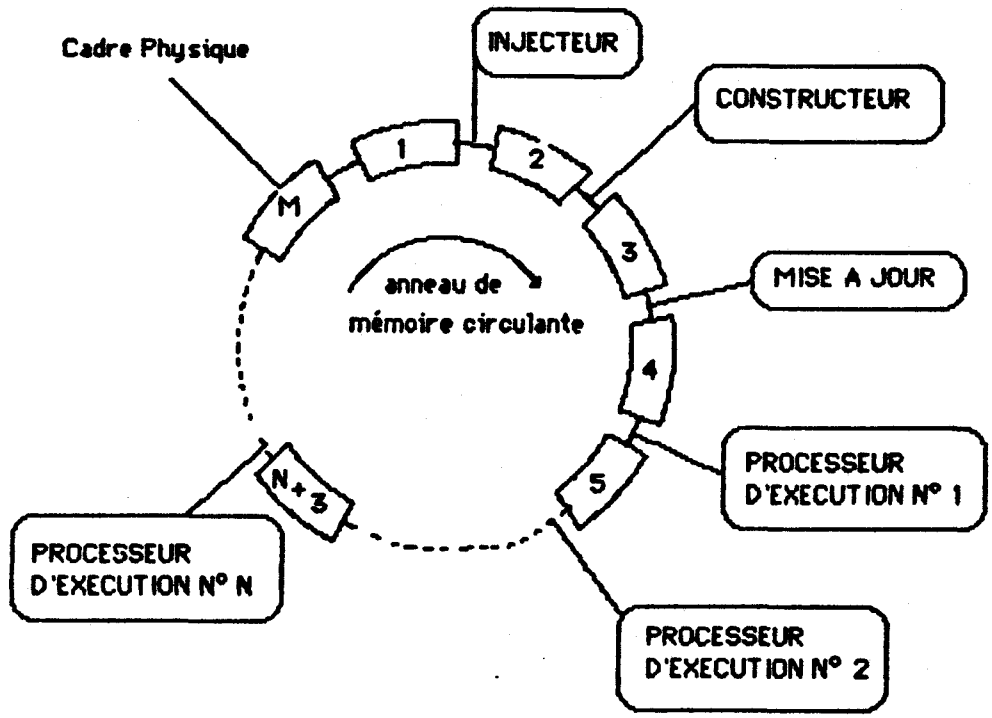


Figure 2 : Le modèle dynamique.

Les paramètres de la simulation sur lesquels nous pourrons agir sont principalement le nombre n de Processeurs d'Exécution et le nombre m de cadres physiques. La relation suivante devra toujours être vérifiée :

$$n+3 \leq m$$

3 représente les fenêtres d'accès du processeur Maj, Constructeur et Injecteur.

B.2. Résultats de simulation

1er Exemple : Programme de TRI-FUSION.

Il s'agit d'un programme de TRI-FUSION qui consiste à diviser une liste initialement non triée en seize sous-listes. Chaque sous-liste est ensuite triée individuellement, puis on possède à la fusion deux par deux des sous-listes triées de façon à n'obtenir qu'une seule liste triée en fin d'algorithme (Figure 3).

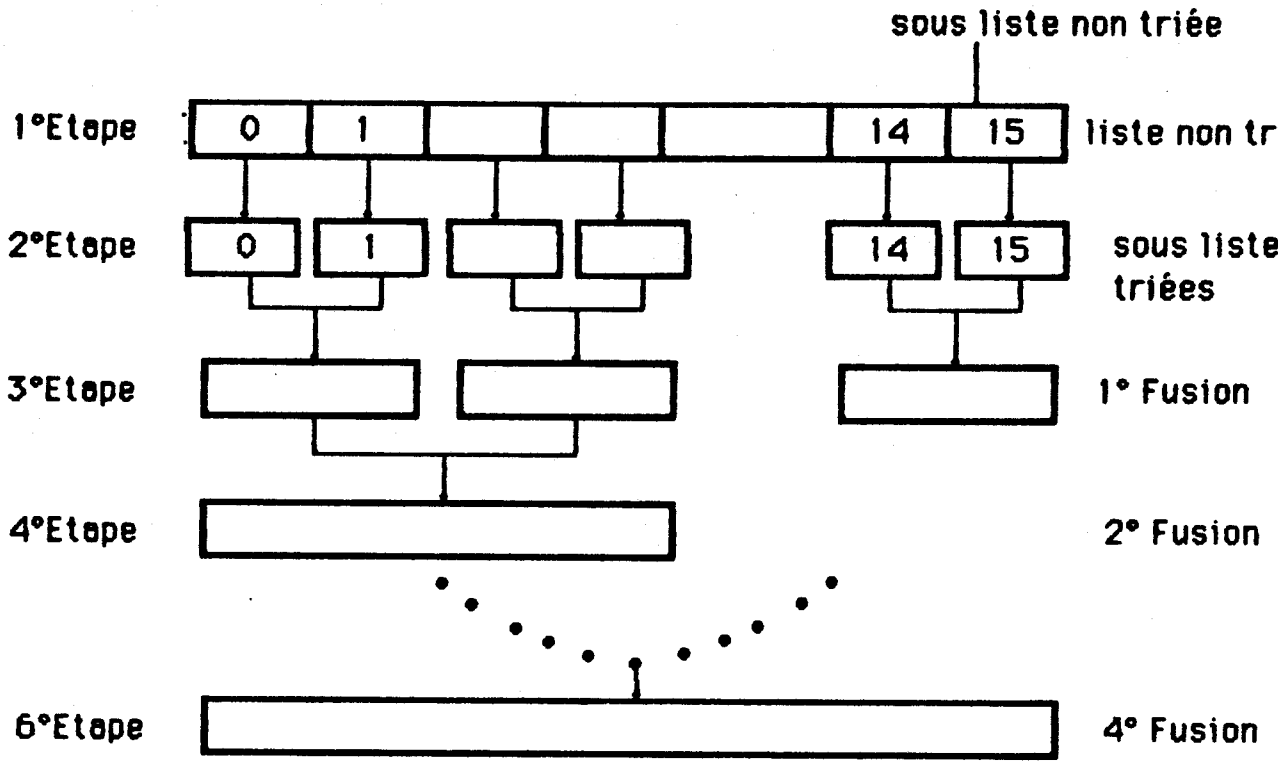


Figure 3 : TRI-FUSION.

L'écriture de ce programme, en utilisant la syntaxe définie auparavant, est la suivante.

Bibliothèque de blocs

Le programme est constitué de trois blocs. Un premier bloc (TRIFUS) a pour tâche de lancer et de coordonner les opérations de TRI et de FUSION. La récupération de la liste triée se fait également par ce bloc.

Un second bloc (TRI) est chargé de trier les sous-listes non triées. Un dernier bloc (FUS)permet de réaliser les opérations de fusion de deux sous-listes triées.

L'écriture de ces blocs est la suivante :

Bloc TRIFUS

```
BLOC : TRIFUS ;
DE : ;
DS : ;
Début ;
      [1] ;
     répéter 16 ;
         TRI (; A) ;
     fin répéter
     répéter 2 ;
         répéter * ;
              FUS (A, A ; B) ;
         fin répéter ;
         répéter * ;
              FUS (B, B ; A) ;
         fin répéter ;
     fin répéter ;
     attendre (A) ;
fin ;
```

BLOC TRI

```
BLOC : TRI ;
DE : ;
DS : C ;
début ;
    [1] ;
    répéter 128 ;
        [1] ;
    fin répéter ;
fin ;
```

BLOC FUS

```
BLOC : FUS ;
DE : ;
DS : D ;
début ;
    [1] ;
    répéter * ;
        [1] ;
    fin répéter ;
fin ;
```

Les blocs TRIFUS et FUS contiennent des instructions répéter * dont les valeurs d'itérations seront fournies par le processeur Injecteur.

Pour le bloc TRIFUS ces valeurs représentent le nombre de FUSIONS que l'on désire lancer simultanément et correspondent respectivement à :

8 pour les premières FUSIONS (3ème Etape)
4 pour les secondes FUSIONS (4ème Etape)
2 pour les troisièmes FUSIONS (5ème Etape)
1 pour la dernière FUSION (6ème Etape)

Pour le bloc FUS il faut tenir compte que l'opération de FUSION nécessite un temps d'exécution proportionnel à la longueur des listes à fusionner c'est-à-dire :

8 Unités de temps (3ème Etape)
16 Unités de temps (4ème Etape)
32 Unités de temps (5ème Etape)
64 Unités de temps (6ème Etape)

Les courbes des figures 4 et 5 représentent les résultats de simulation obtenus pour cette bibliothèque d'essai. Celles-ci représentent le temps d'exécution en fonction du nombres de cadres de l'anneau. Nous avons le nombre de processeurs comme paramètre.

Nous pouvons faire les remarques suivantes :

- Pour un nombre de processeurs donné on passe par un palier minimum de temps d'exécution.

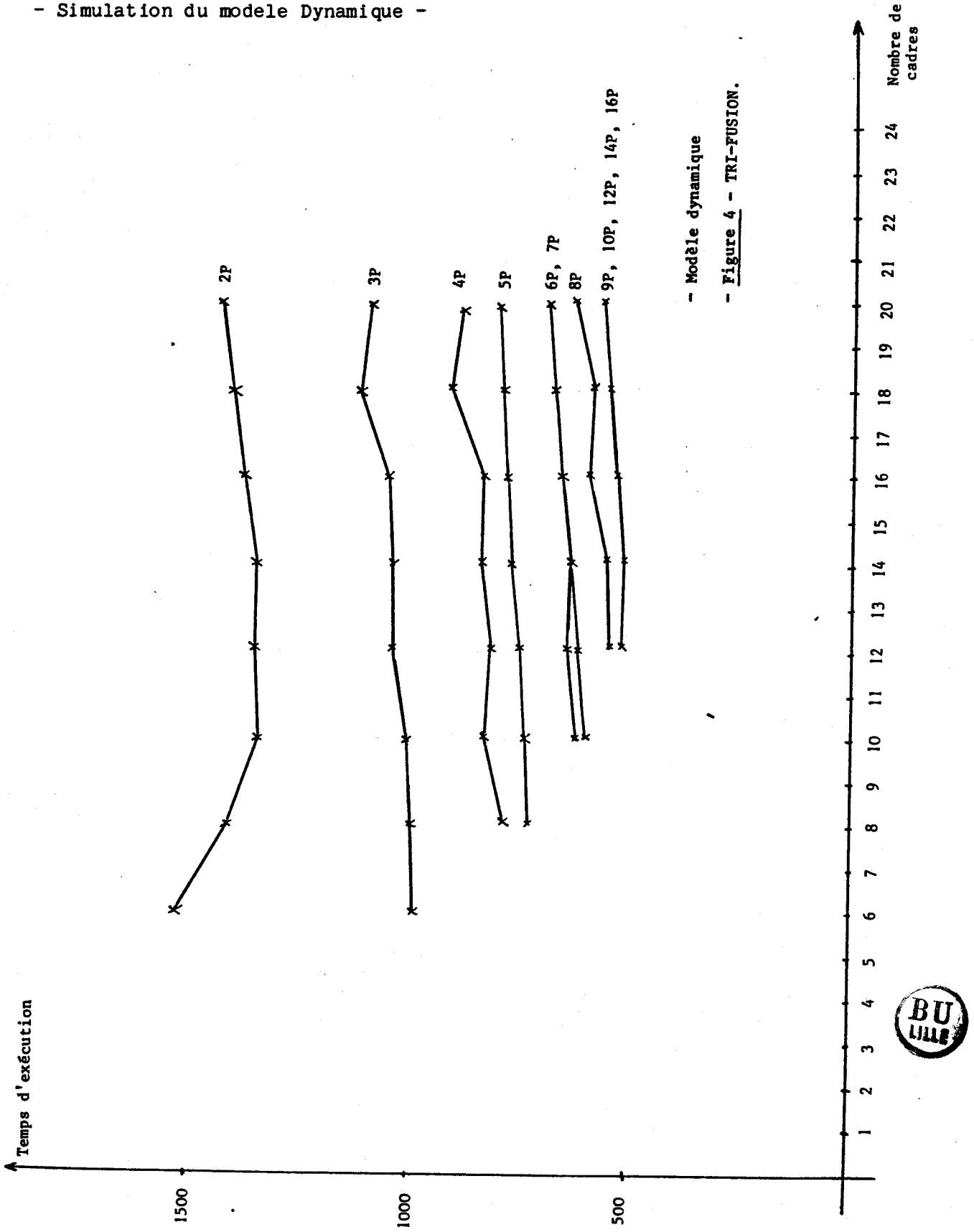


fig 4.

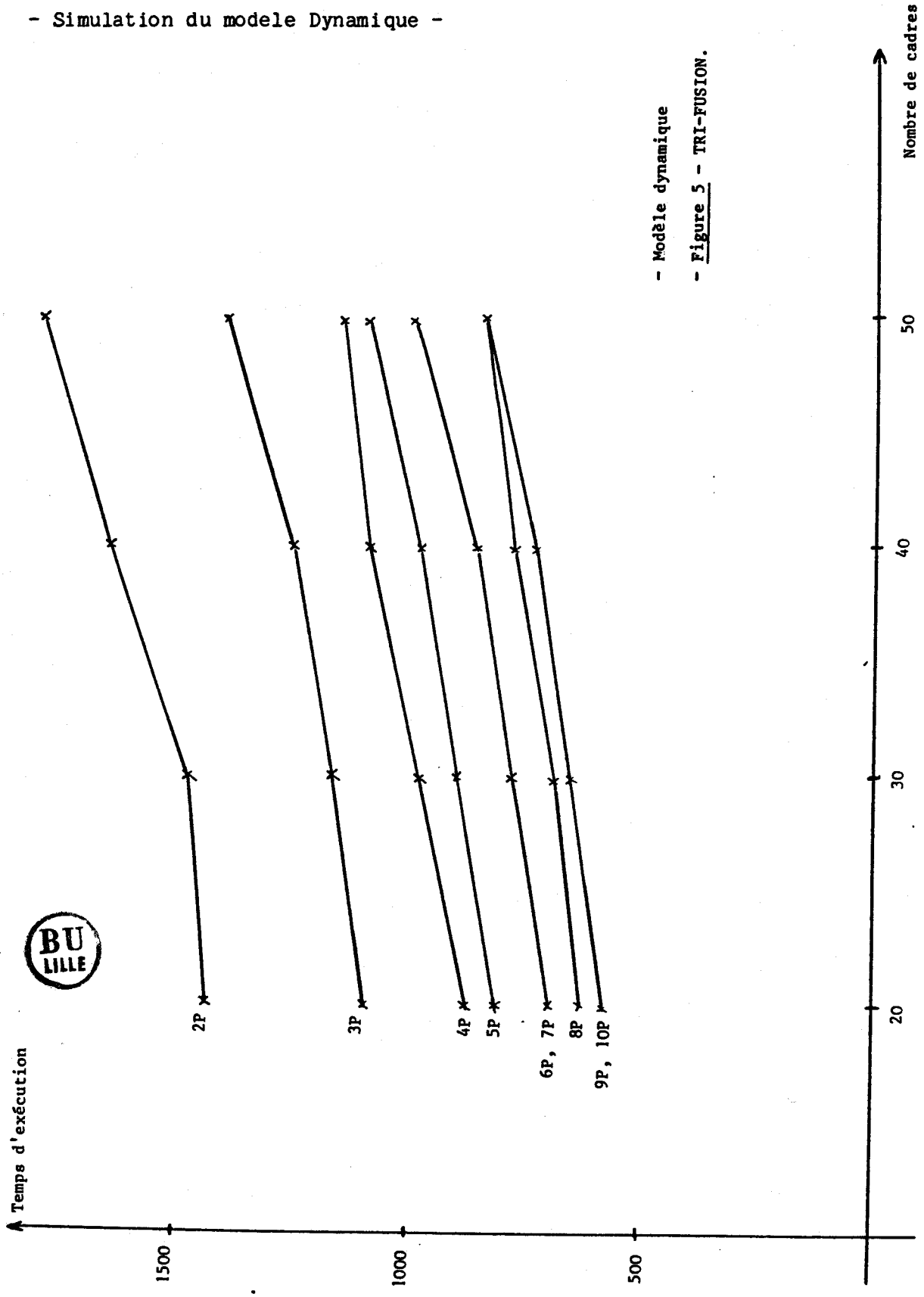


fig 5

- Après ce palier on observe une dégradation des performances due au temps de communication de l'anneau qui devient trop important vis-à-vis du temps d'exécution des blocs élémentaires.

- Le temps minimal d'exécution est obtenu pour 9 processeurs.

- Au delà de 9 processeurs on n'observe plus d'amélioration des performances du système, ceci provenant du fait qu'on a atteint le parallélisme optimal à l'exécution.

Pour une configuration comportant 9 processeurs à 14 cadres, l'activité des processeurs est représentée sur le tableau de la figure 6.

Processeur n°	INACTIF	ACTIF	ATTENTE
1	139	382	0
2	233	288	0
3	251	270	0
4	251	270	0
5	261	260	0
6	261	260	0
7	261	260	0
8	261	260	0
9	333	188	0

NOMBRE DE CADRE DE L'ANNEAU : 14
NOMBRE DE PROCESSEUR : 9

TEMPS TOTAL DE TRAITEMENT : 521
TEMPS EQUIVALENT MONOPROCESSEUR : 2438
GAIN : 74 %

Figure 6

Nous remarquerons que les 9 processeurs sont actifs. Leur taux d'activité est représenté au tableau de la figure 7.

Le taux d'activité moyen est de 52 %.

Le temps total d'exécution est de 521, alors que celui-ci serait de 2438 sur une machine monoprocesseur. Le gain obtenu est donc de 74 % (2438 - 521 / 2438).

n° Processeur	Taux d'activité en %
1	73
2	55
3	52
4	52
5	50
6	50
7	50
8	50
9	36

Figure 7 : Taux d'activité des Processeurs d'Exécution.

Pour cet exemple aucun des processeurs n'est passé par l'état attente ce qui signifie que l'outil de communication (i.e. mémoire circulante) n'a jamais été saturé.

Le taux d'activité des processeurs décroît avec le numéro de celui-ci, car les processeurs situés juste après le processeur Maj ont un rôle privilégié de part le sens de circulation de l'anneau de mémoire circulante.

Nous pouvons donner une représentation du graphe de dépendances des blocs pour ce programme (figure 8). Celui-ci peut nous renseigner sur le parallélisme maximal du programme et donc sur le nombre maximal de Processeurs d'Exécution que le système doit comporter.

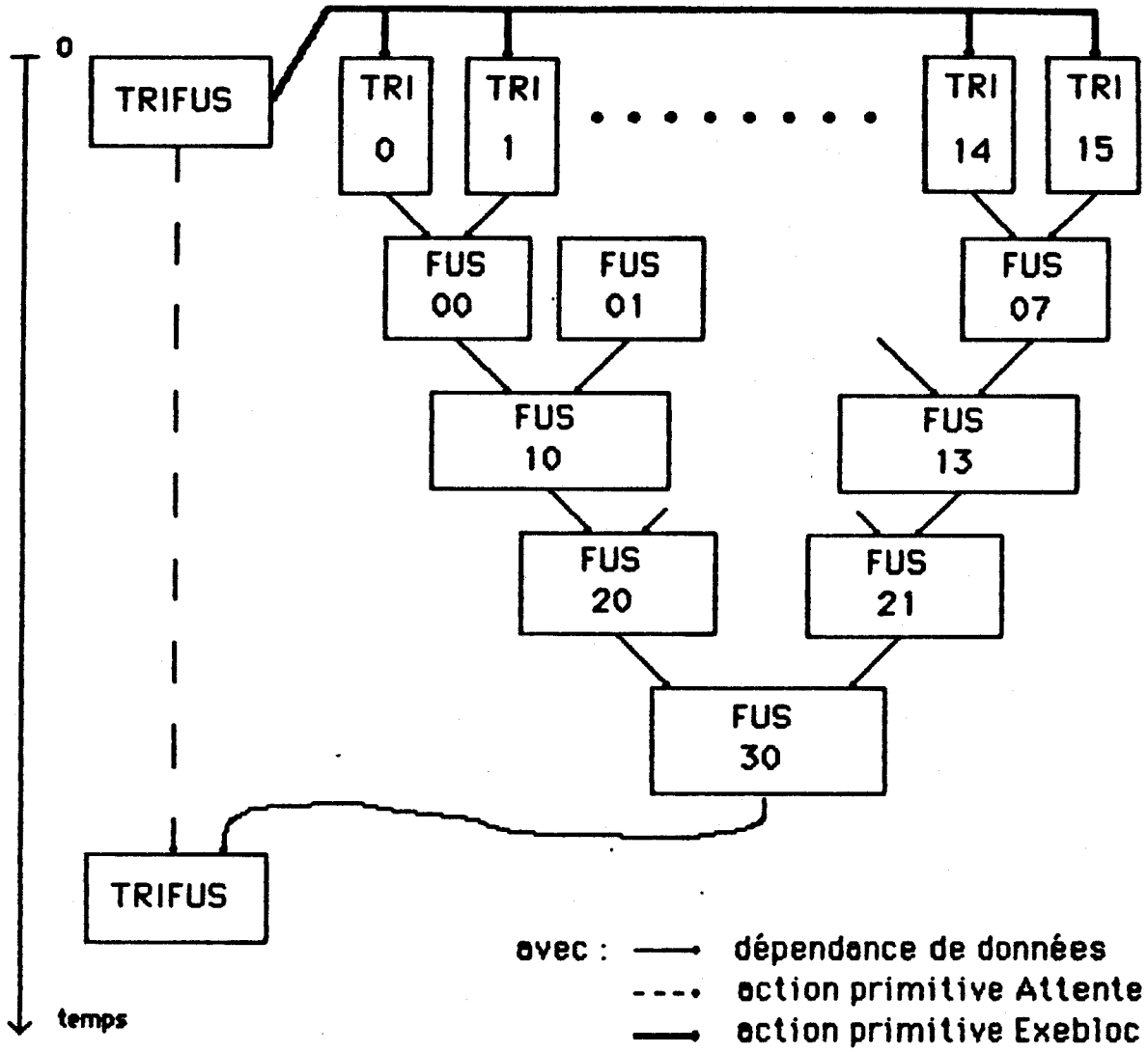


Figure 8

Le parallélisme maximal est obtenu lorsque les seize blocs TRI et le bloc TRIFUS sont exécutables.

On observe donc une différence importante entre le parallélisme maximal (17) et le parallélisme optimal à l'exécution qui est de 9.

Pour expliquer cette différence nous allons considérer d'autres problèmes dont les programmes correspondants sont plus simples à interpréter.

2ème Exemple : Résolution d'un système linéaire.

L'idée de ce programme est la résolution d'un système linéaire par la méthode de JACOBI. Ce programme consiste à lancer en parallèle l'exécution d'un calcul de coordonnée pour des jeux de données différentes. Une fois l'ensemble des calculs de coordonnée terminé, un test de convergence est effectué pour savoir s'il faut réitérer le processus précédant.

Le programme écrit en pseudo-langage sur une machine de type V-Neumann serait le suivant :

```
début ;

TEST := VRAI ;

        Tant que      TEST faire

            Pour j de 1 à 50 faire
                .
                .
                .
                Calcul de coordonnée A[j] ;

            fait ;
            .
            .
            .
            Convergence ;
            .
            .
            .
        fait ;

fin
```

La procédure convergence positionne le booleen TEST selon l'état de la convergence.

Le même programme écrit dans le langage de simulation serait le suivant, en tenant compte du fait que la procédure "calcul de coordonnée" peut s'exécuter concouramment sur plusieurs jeux de données :

```
BLOC : SYSTLIN ;
DE :;
DS ;:
début ;
    [1] ;
    répéter 10 ;
        répéter 50 ;
            CALCOR1 (; A) ;
        fin répéter ;
      attendre * (A) ;
      [3] ;
    fin répéter ;
fin ;
```

```
BLOC : CALCOR1
DE :;
DS : B ;
début ;
    [4] ;
fin ;
```

Deux blocs sont nécessaires pour la modélisation de ce programme. Le premier bloc SYSTLIN permet de lancer en parallèle l'exécution du calcul de coordonnée (second bloc CALCOR1).

Le test de convergence n'étant pas significatif pour notre simulation nous avons borné à 10 le nombre d'itérations.

La figure 9 représente les résultats obtenus à la simulation de ce programme pour un nombre de processeurs variant de 3 à 5.

Avec 5 processeurs on s'aperçoit que le 5ème demeure toujours inactif (figure 10).

Pour les configurations avec 3 ou 4 processeurs, le 4ème processeur est en famine fréquente dans la mesure où il passe par l'état attente durant un laps de temps supérieur à l'état actif (Figure 10 et 11).

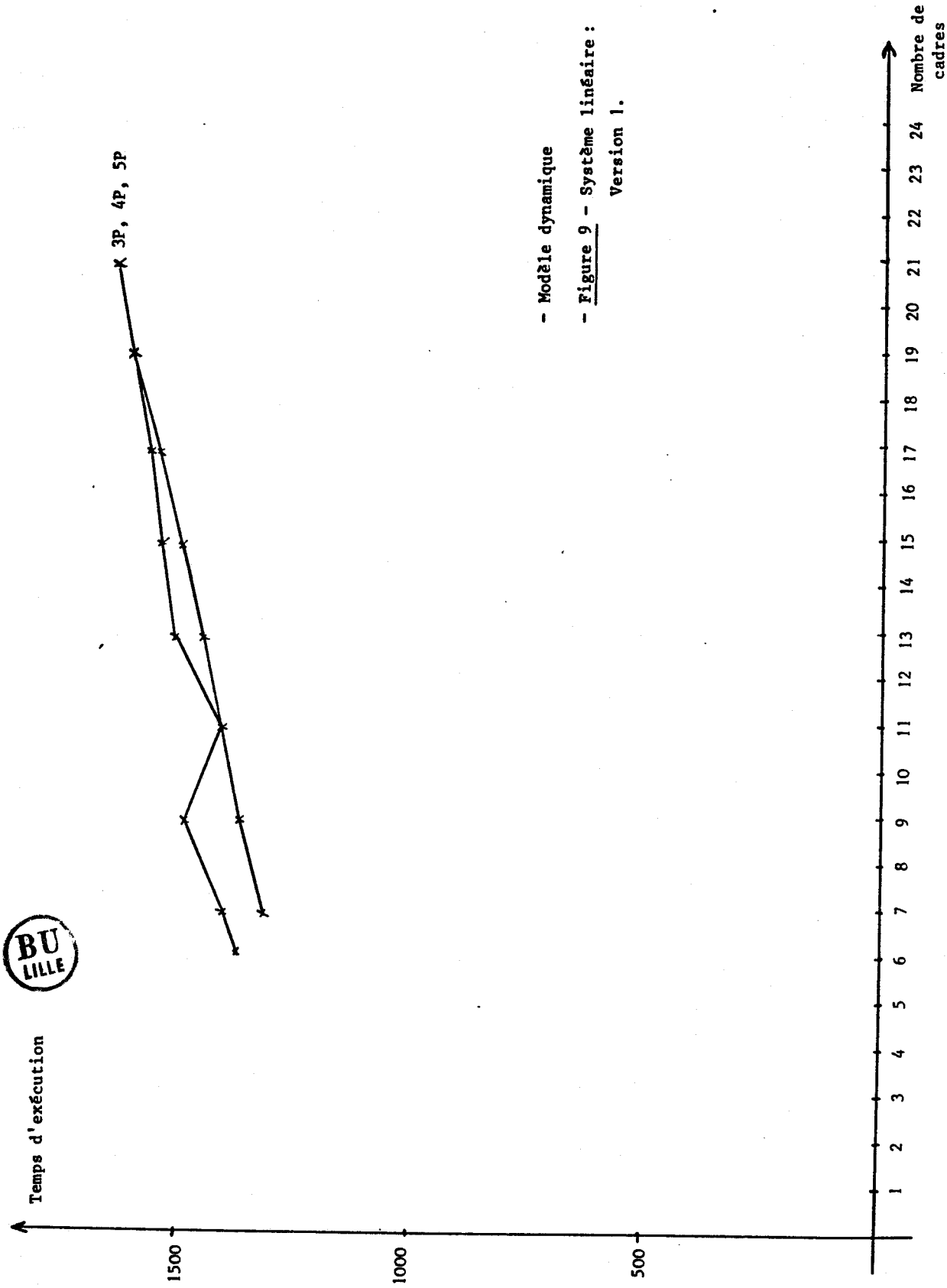


fig 9.

Processeur n°	INACTIF	ACTIF	ATTENTE	
1	172	1182	0	
2	374	800	180	
3	414	760	180	
4	614	320	420	PROCESSEUR EN FAMINE
5	1354	0	0	PROCESSEUR INACTIF

NOMBRE DE CADRE DE L'ANNEAU : 9
NOMBRE DE PROCESSEUR : 5

TEMPS TOTAL DE TRAITEMENT : 1354
TEMPS EQUIVALENT MONOPROCESSEUR : 3062

Processeur n°	INACTIF	ACTIF	ATTENTE	
1	176	1222	0	
2	418	840	140	
3	498	760	140	
4	718	240	440	PROCESSEUR EN FAMINE
5	1398	0	0	PROCESSEUR INACTIF

NOMBRE DE CADRE DE L'ANNEAU : 11
NOMBRE DE PROCESSEUR : 5

TEMPS TOTAL DE TRAITEMENT : 1398
TEMPS EQUIVALENT MONOPROCESSEUR : 3062

fig 10.

Processeur n°	INACTIF	ACTIF	ATTENTE	
1	172	1182	0	
2	374	800	180	
3	414	760	180	
4	614	320	420	PROCESSEUR EN FAMINE

NOMBRE DE CADRE DE L'ANNEAU : 9
NOMBRE DE PROCESSEUR : 4

TEMPS TOTAL DE TRAITEMENT : 1354
TEMPS EQUIVALENT MONOPROCESSEUR : 3062

fig 11.

Une deuxième version de ce problème a été écrite afin de mettre en évidence l'influence du temps d'exécution du bloc sur les performances du système. Ce caractère était déjà présent dans la simulation du modèle statique.

Dans cette version le temps de calcul du bloc 'calcul de coordonnée' devient 17 Unités de temps. Le programme correspondant devient alors :

```
BLOC: SYSTLIN ;
DE :;
DS :;
début ;
    [1] ;
    répéter 10 ;
        répéter 50 ;
            CALCOR2 (; A) ;
        fin répéter ;
        attendre * (A) ;
        [3] ;
    fin répéter ;
fin ;
```

```
BLOC : CALCOR2 ;
 DE :;
 DS : B ;
 début ;
     [17] ;
 fin ;
```

La courbe de la figure 12 donne les résultats de simulation.

Pour un nombre de processeurs variant de 3 à 10 les temps minimaux obtenus décroissent linéairement. Le taux de parallélisme obtenu à l'exécution devient nettement plus important que celui obtenu dans la première version.

Pour une configuration comportant 10 processeurs on obtient le meilleur temps d'exécution qui est de 1552 Unités de temps. Au delà de 10 proceseurs (non représentés) aucune amélioration n'est observée.

Le parallélisme optimal à l'exécution est donc de 10, bien que le parallélisme maximal théorique soit supérieur.

Le graphe de dépendances des blocs nous permet de l'évaluer (figure 13).

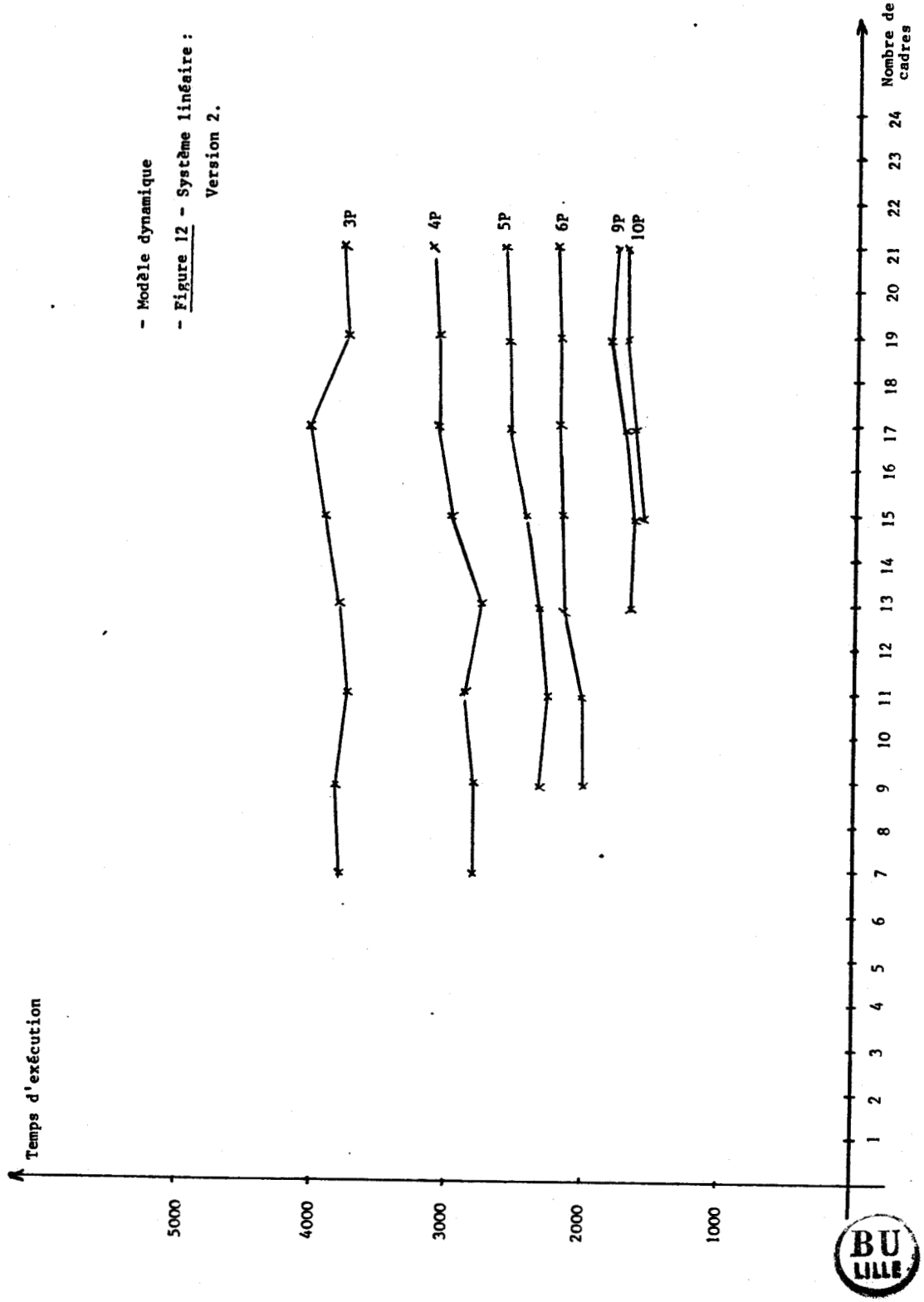

- Modèle dynamique
- Figure 12 - Système linéaire :
Version 2.
Temps d'exécution
Nombre de cadres
3P
4P
5P
6P
9P
10P
5000
4000
3000
2000
1000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

fig 12.

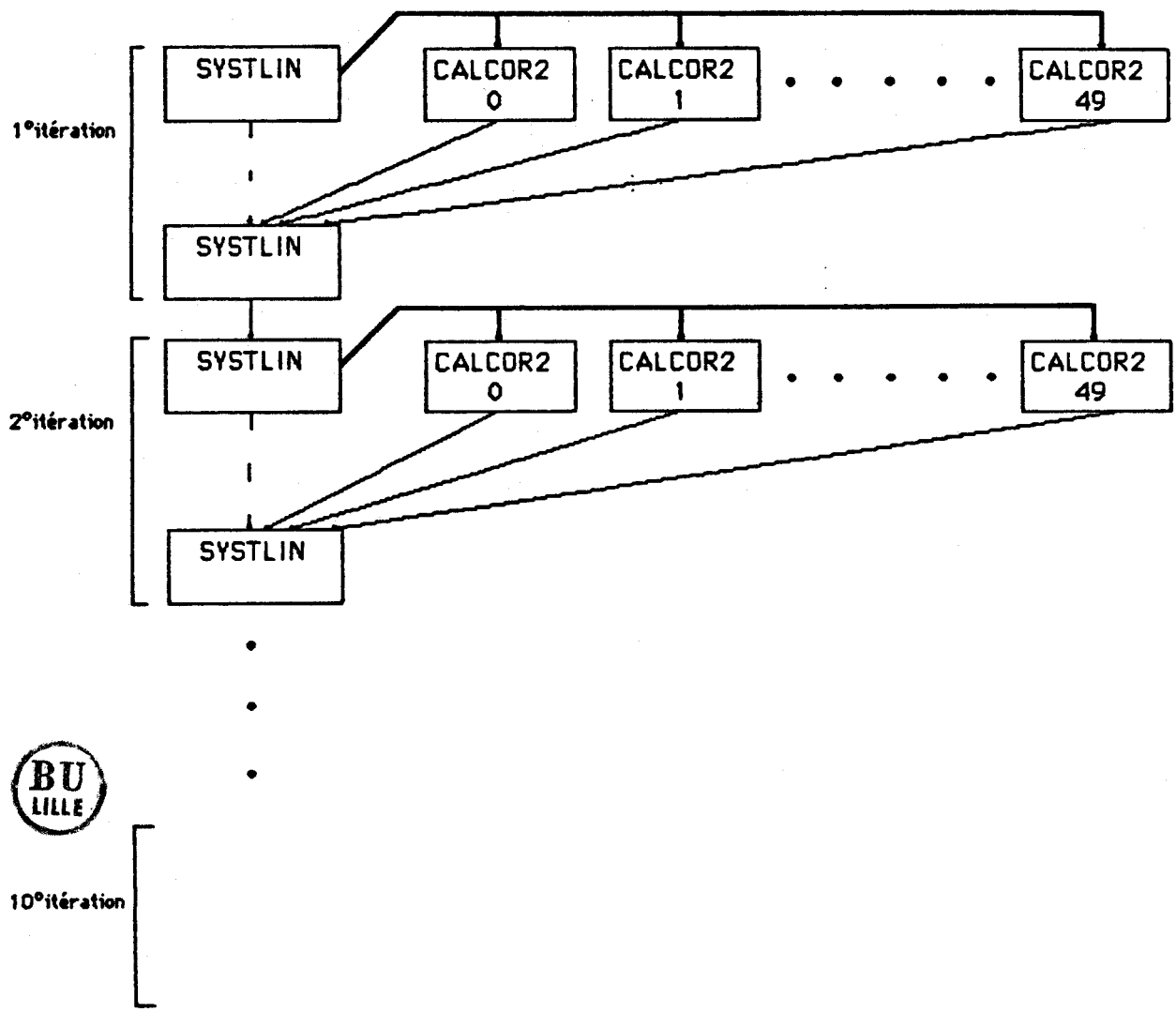


Figure 13 : Graphe de dépendance SYSTLIN.

Sur ce graphe, il apparaît clairement que le parallélisme maximal théorique est de 51.

La différence entre le parallélisme maximal théorique et le parallélisme optimum à l'exécution est inhérente à notre architecture et s'explique de la manière suivante :

Le bloc SYSTLIN qui permet de lancer les 50 blocs CALCOR2 ne dépose effectivement dans l'anneau qu'une demande d'exécution du bloc CALCOR2 toutes les deux unités de temps.

Celles-ci seront traitées par le processeur constructeur qui pourra à son tour déposer un bloc en attente CALCOR2 toutes les deux unités de temps.

Le processeur Mise à jour pourra également transformer ces blocs en blocs exécutables CALCOR2 toutes les deux unités de temps.

Schématiquement on peut représenter le fonctionnement décrit précédemment par le "pipe-line" de la figure 14.

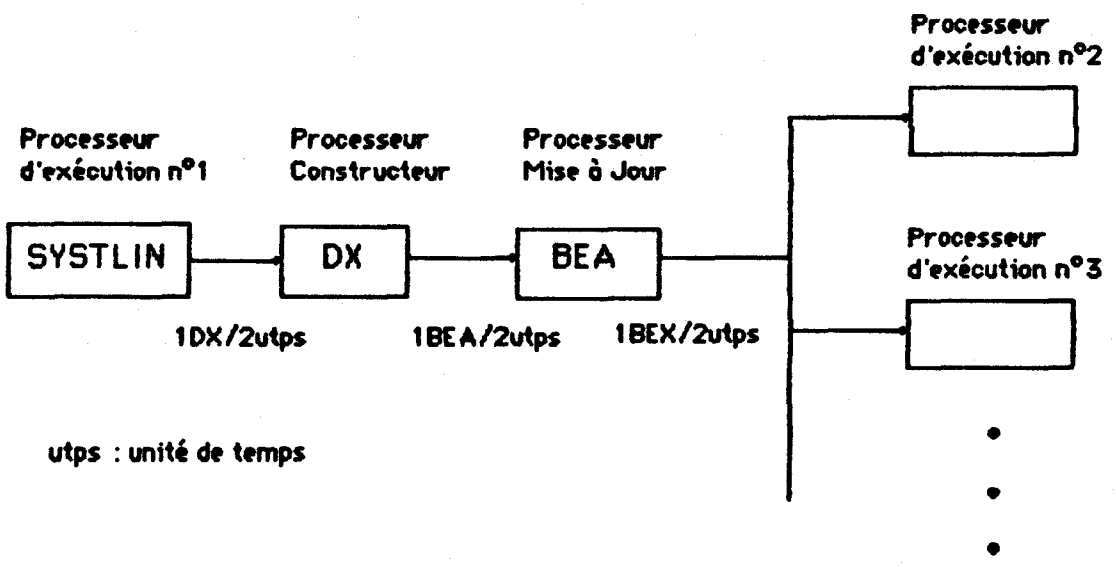


Figure 14 : Fonctionnement de type "pipe-line".

Globalement on ne fournit qu'un bloc exécutable CALCOR2 tous les deux unités de temps, à l'ensemble des Processeurs d'Exécution.

Le temps d'exécution du bloc CALCOR2 étant de 17 Unités de temps, un Processeur d'Exécution capturera une fois sur 9 le bloc exécutable CALCOR2 fourni par le processeur Mise à jour.

Le nombre optimal de Processeurs d'Exécution nécessaires pour le traitement des 50 blocs CALCOR2 est donc lié par la relation :

Nombre optimal de processeurs = ⌈ (temps d'exécution bloc) * débit bloc executable ⌉

donc ici = ⌈ 17 * 0,5⌉ = 9.

Neuf processeurs seraient donc suffisants pour le "calcul des coordonnées" (CALCOR2) et un processeur pour l'exécution du bloc SYSTLIN. On retrouve bien que le parallélisme optimal à l'exécution est de 10.

Pour une configuration comportant 10 processeurs et 13 cadres (voir tableau fig 14bis) le gain obtenu par rapport à une machine monoprocesseurs est de 83 %.

La relation liant le nombre optimal de processeurs et le temps d'exécution bloc restera vérifiée pour tout problème pouvant se décomposer d'une manière analogue au problème SYSTLIN à condition que le débit bloc exécutable demeure constant. Pour la version 1 du problème SYSTLIN la figure 9 nous indique que le meilleur temps est obtenu pour 4 processeurs et alors que la relation précédente nous donne un nombre optimal de processeurs égal à 3.

Un examen de la trace fournie par le simulateur nous révèle que le débit des blocs exécutables n'est pas resté constant ceci pour des raisons liées à la charge de l'anneau.

Processeur n°	INACTIF	ACTIF	ATTENTE
1	320	1232	0
2	502	1020	30
3	502	1020	30
4	502	1020	30
5	502	1020	30
6	502	1020	30
7	502	1020	30
8	652	850	50
9	832	680	40
10	832	680	40

NOMBRE DE CADRE DE L'ANNEAU : 13
NOMBRE DE PROCESSEUR : 10
TEMPS TOTAL DE TRAITEMENT : 1552
TEMPS EQUIVALENT MONOPROCESSEUR : 9562
GAIN : 83 %

fig 14bis.

C. Le modèle théorique avec files

C.1. Présentation

La simulation du modèle dynamique décrite précédemment ne nous permet pas d'évaluer les performances de l'implémentation que nous avons choisie pour celui-ci. Il serait intéressant de pouvoir calculer les temps de communication introduits par l'utilisation d'une mémoire circulante afin de juger de l'opportunité d'une telle solution par rapport à d'autres implémentations possibles. C'est dans cet objectif que nous avons écrit la simulation du modèle théorique avec files. Ce modèle permet de simuler le modèle théorique de base, l'aspect communication a été volontairement supprimé, seul l'aspect fonctionnel des processeurs a été conservé.

Si l'on se réfère au modèle théorique rappelé au Chapitre II de cette thèse nous pouvons observer qu'il existe trois types de communication interprocesseur.

Le premier de ces types est une communication de type "demande d'exécution" (DX) entre les Processeurs d'Exécution et le processeur Constructeur.

Le second type de communication concerne l'envoi de domaines de sortie (DS) ou de bloc en attente (BEA) entre d'une part les Processeurs d'Exécution et le processeur Maj et d'autre part le processeur Constructeur et le processeur Maj.

Le troisième type de communication est réalisé par le processeur Maj qui transmet des blocs exécutables (BEX) aux Processeurs d'Exécution.

Une manière triviale de gérer ces communications consiste à associer à chacun des types précédents une file d'attente.

La figure 15 montre la configuration globale du système avec files.

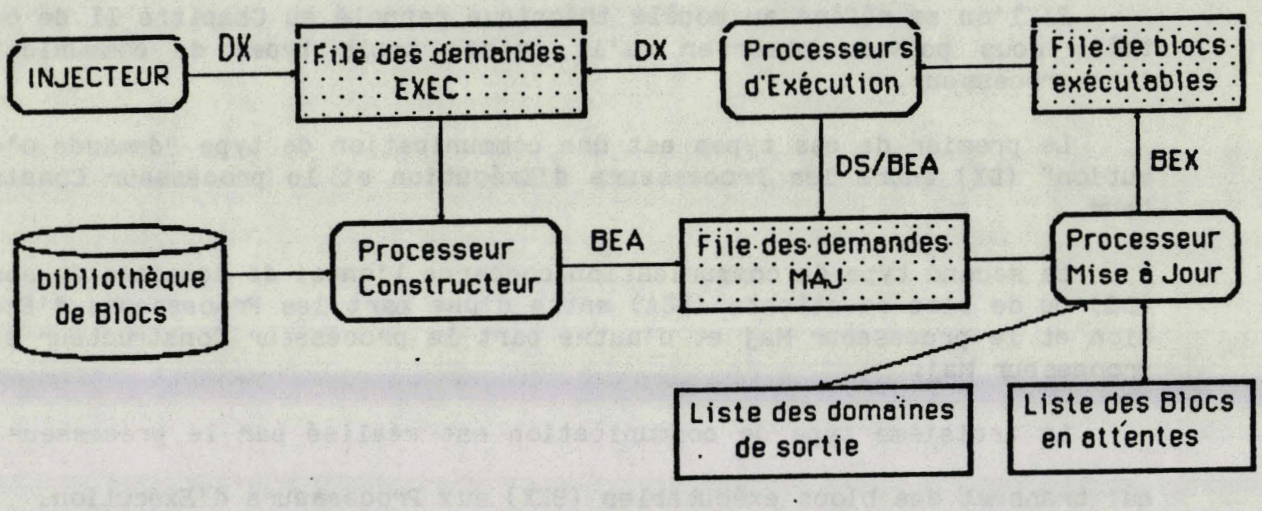


Figure 15 : Modèle théorique avec files.

Aucune contrainte sur la taille des files n'est formulée.

A chaque unité de temps une file peut être accédée simultanément par plusieurs processeurs. La seule contrainte vient du fait que le processeur Constructeur et Maj ne peuvent traiter qu'une seule demande par unité de temps.

Ce modèle nous renseigne sur trois points fondamentaux :

* Détermination du temps d'exécution minimal d'un programme exécuté sur le modèle MAUD.

* Evaluation du parallélisme maximal d'un programme.

* Estimation des temps de communication par comparaison avec les résultats des autres modèles.

C.2. Résultats de simulation

Nous reprenons dans ce paragraphe les exemples de programmes introduits dans le paragraphe précédent de manière à pousser plus loin leur interprétation.

C.2.1. Premier exemple : Programme de TRI-FUSION

La figure 16 donne les temps d'exécution en fonction du nombre de processeurs introduits dans le modèle théorique avec files.

Le temps minimal d'exécution est de 300 pour une configuration avec 17 Processeurs d'Exécutions.

Le parallélisme de 17 est obtenu après que le bloc TRIFUS ait lancé l'exécution des 16 blocs TRI.

On observe également sur la courbe de la figure 16 un palier pour un nombre de processeurs variant de 9 à 15.

Ce palier s'explique par le graphe de dépendances qui régit l'exécution des blocs TRI et des blocs FUS. La figure 16 bis donne de manière simpliste l'ordonnancement des blocs pour une configuration avec 9 processeurs et 15 processeurs.

Pour toutes les configurations comprises entre 9 et 15 on obtient un ordonnancement qui fournit le même temps d'exécution.

Nous pouvons estimer les temps de communication introduits par le modèle dynamique du paragraphe précédent.

Pour cela on compare le temps d'exécution $T_{F,N}$ du modèle théorique pour N processeurs avec le temps minimal $T_{M,N}$ obtenu dans le modèle dynamique avec le même nombre N de processeurs.

La différence $T_{F,N} - T_{M,N}$ reste constante quelque soit N et égale à 50 Unités de temps environ.

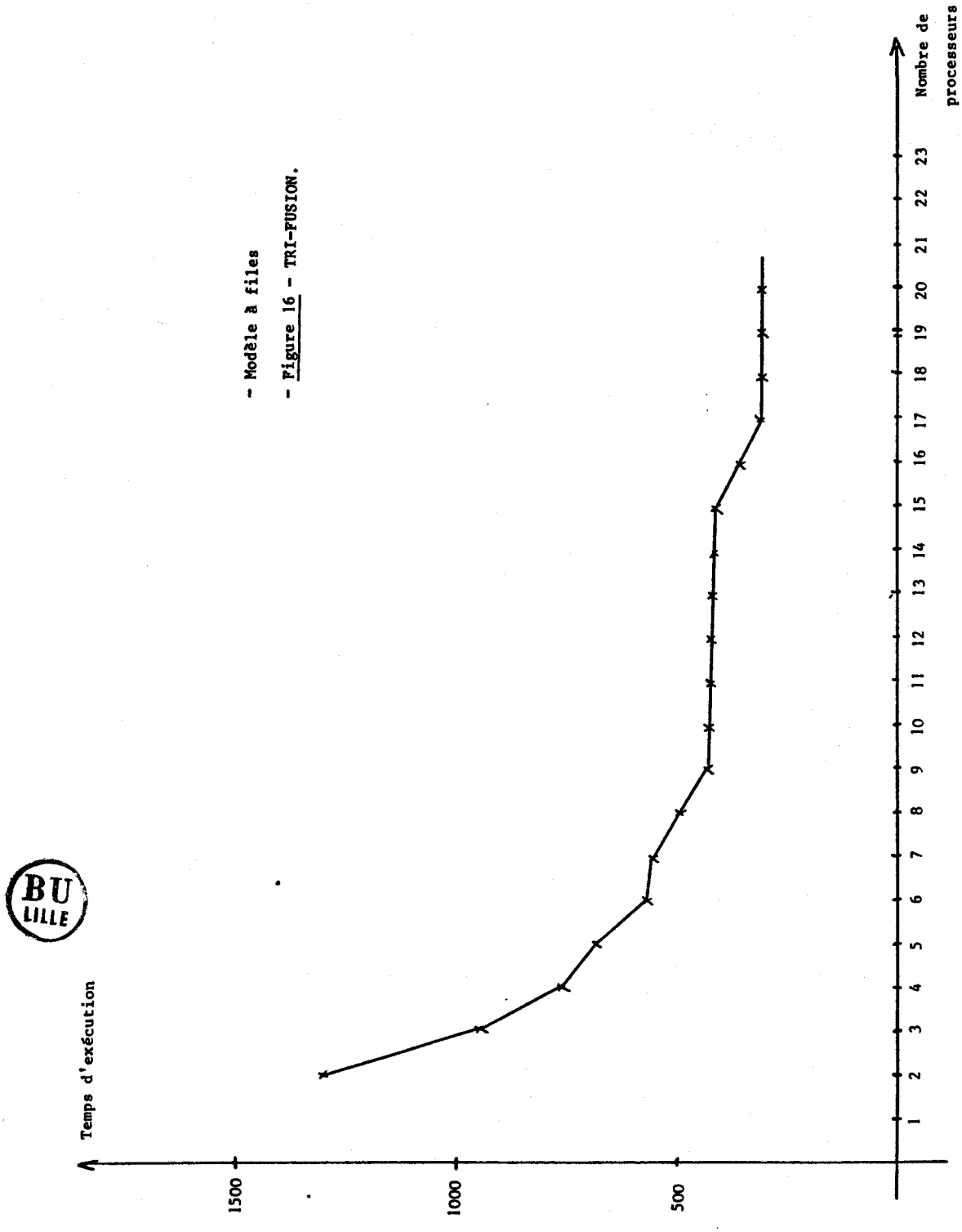


fig 16.

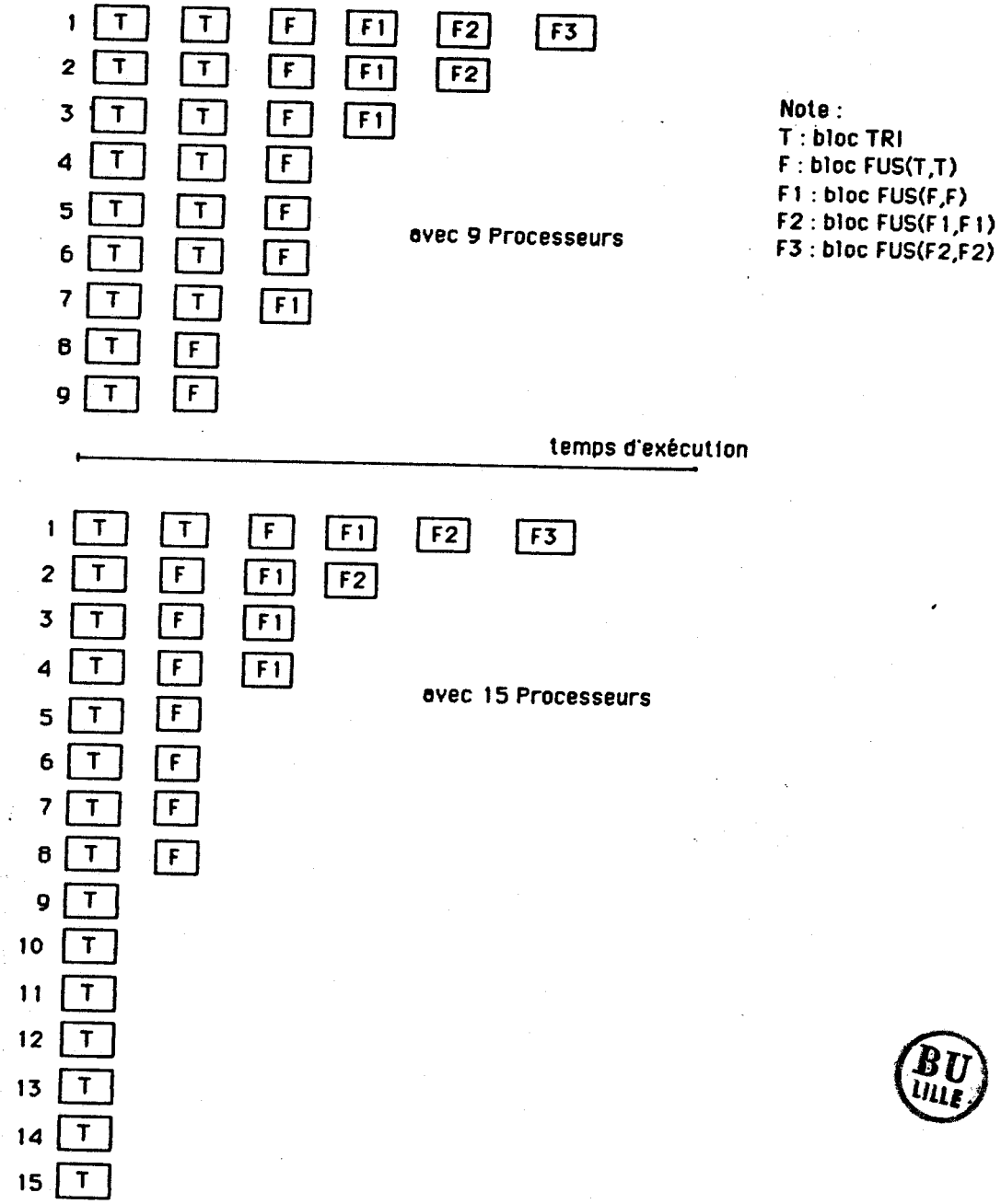


fig 16bis.

Si on compare cette différence avec le temps minimal obtenu dans le modèle dynamique (9 processeurs, 14 cadres) qui était de 521 Unités de temps, on peut estimer à moins de 10 % l'"overhead" introduit par l'outil de communication (i.e. mémoire circulante).

C.2.2. Second exemple : Système linéaire

Les résultats obtenus pour les versions 1 et 2 sont représentés figures 17 et 18. Ces courbes font apparaître les mêmes remarques que pour le modèle dynamique.

Pour la version 1 le temps d'exécution du bloc CALCOR1 étant de 4 Unités de temps, le parallélisme optimal est obtenu pour 3 proceseurs. Un processeur exécute le bloc SYSTLIN et lance une demande d'exécution du bloc CALCOR1 toutes les deux unités de temps. Deux autres processeurs suffisent pour exécuter alternativement les blocs CALCOR1.

Pour la version 2 le temps d'exécution du bloc CALCOR2 étant de 17 Unités de temps, le parallélisme optimum devient 10 processeurs. La courbe de la figure 18 fait apparaître une relation du type :

Temps d'exécution * Nombre de processeurs = Constante

pour un nombre de processeurs variant de 2 à 10.

Pour les deux versions la relation qui lie le nombre optimal processeurs avec le temps d'exécution du bloc (CALCOR1 ou CALCOR2) reste vérifiée :

$$\text{Nombre optimal de processeur} = 1 + \left\lceil \frac{\text{Temps d'exécution du bloc}}{\text{Période des demandes d'exécution}} \right\rceil$$

avec 1 représente le processeur nécessaire pour l'exécution du bloc initial SYSTLIN, temps d'exécution du bloc signifie le temps d'exécution du bloc CALCOR1 ou CALCOR2 et période des demandes d'exécution représente le temps qui sépare deux appels de la primitive EXECBLOC dans le bloc SYSTLIN (= 2 dans notre cas).

Pour la version 2, nous pouvons évaluer les temps de communication du modèle dynamique. Pour une configuration comprenant 10 processeurs et 13 cadres, celui-ci est égal à 330 Unités de temps, et représente environ 20 % du temps d'exécution. Pour cet exemple l'"overhead" introduit par la mémoire circulante devient important.

- Modèle à files
- Figure 17 - Système linéaire : Version 1.

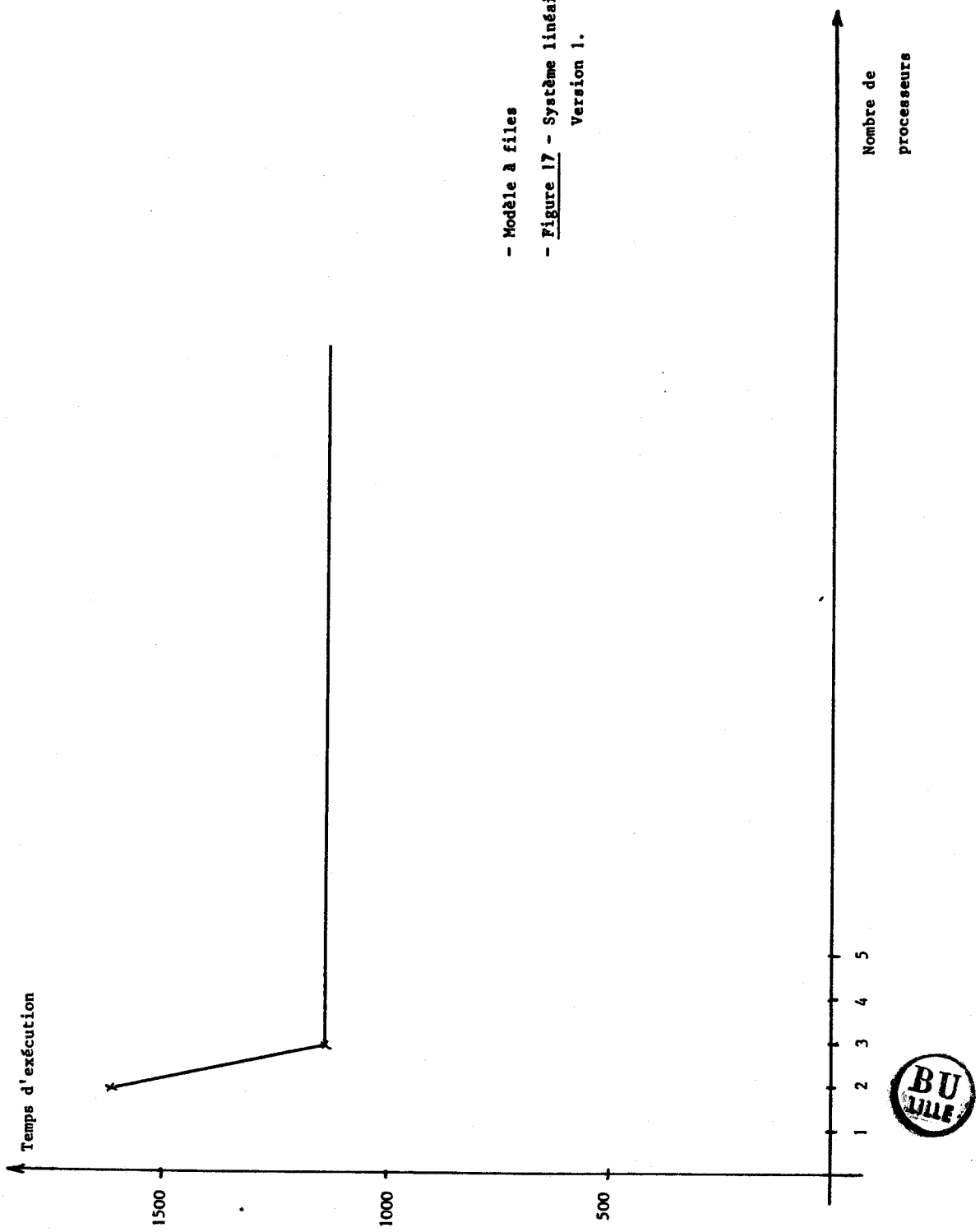


fig 17.Systeme linéaire, version 1

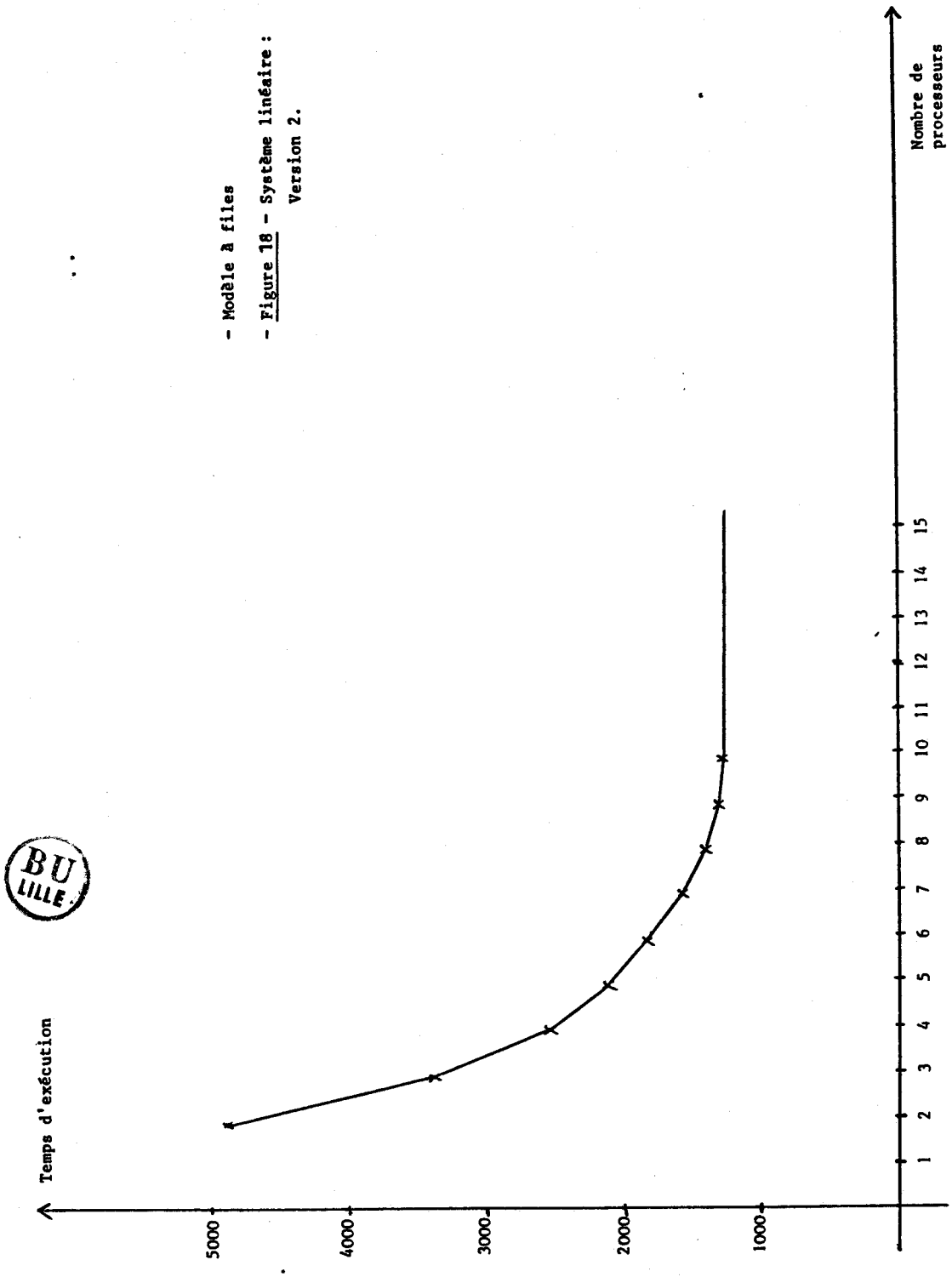


fig 18.Systeme linéaire, version 2

Conclusion

La simulation du modèle dynamique a permis de montrer les points fondamentaux suivants :

- Amélioration des temps d'exécution par rapport à une machine classique, lorsque le problème à traiter présente suffisamment du parallélisme.

- Influence du temps d'exécution des blocs sur le parallélisme optimal à l'exécution par rapport à la longueur de l'anneau de mémoire circulante. Le temps d'exécution d'un bloc doit se rapprocher du temps de propagation d'un bloc durant un tour d'anneau.

- Influence du débit avec lequel on lance une série de blocs à exécuter concurremment, sur le parallélisme optimal à l'exécution. Pour des problèmes du type de l'exemple "SYSTLIN" la relation est la suivante :

Nombre de processeurs optimal = 1 + ⌈ Temps d'exécution d'un bloc * Débit demandes d'exécution⌉ .

Ces remarques devant être prises en considération pour la conception d'un logiciel permettant le découpage d'un programme, écrit dans un langage de programmation structuré classique, en blocs pour MAUD.

Dans le chapitre suivant de cette thèse nous présentons quelques insuffisances de notre implémentation. Des solutions y sont proposées.

CHAPITRE VI

"Communication de l'anneau de mémoire circulante"

Introduction

A. Problèmes de l'anneau de mémoire circulante

- A.1 - Problème de famine
- A.2 - Problème de blocage

B. Anneau Inverse

- B.1 - Principe
- B.2 - Résultats de simulation

C. Le Gérant d'anneau

- C.1 - Principe
- C.2 - Résultats de simulation

D. Modèle avec plusieurs anneaux

- D.1 - Motivations
- D.2 - Présentation du Modèle
- D.3 - Etude fonctionnelle du Modèle
- D.4 - Implémentation du Modèle à plusieurs anneaux
- D.5 - Configuration du système
- D.6 - Résultats de simulation

Conclusion

Introduction

Ce chapitre présente dans une première partie quelques limitations du modèle dynamique. Plusieurs solutions pour remédier à ces limitations sont décrites dans les parties 2, 3 et 4 de ce chapitre.

La deuxième partie présente un mécanisme de régulation de la charge des Processeurs d'Exécution par l'adjonction au modèle dynamique d'un anneau inverse.

La troisième partie complète le modèle précédent par insertion d'un processeur Gérant dont la fonction est de réguler la charge de l'anneau.

La quatrième partie propose une reconfiguration dynamique de l'anneau de mémoire circulante en plusieurs anneaux de manière à se rapprocher des performances du modèle théorique à files. Une implémentation possible y est décrite.

A. Problèmes de l'anneau de mémoire circulante

L'anneau de mémoire circulante étant de capacité finie, il peut survenir des situations où celui-ci se trouve saturé.
Les processeurs ne peuvent alors plus déposer de nouveaux blocs, et en conséquence, ceux-ci passent par un état attente jusqu'à ce qu'un cadre de l'anneau leur permette d'y déposer leurs blocs.

Ces situations apparaissent dans les différents cas suivants :

* Anneau saturé par des domaines de sortie destinés au processeur Mise à Jour
* Anneau saturé par des blocs en attente destinés au processeur Mise à Jour
* Anneau saturé par des demandes d'exécution destinées au processeur Constructeur
* Anneau saturé par des blocs exécutables destinés aux Processeurs d'Exécution

Selon la nature du processeur que l'on considère, ces situations peuvent ou non ralentir l'activité de celui-ci.
Du fait de la configuration du modèle dynamique donnée dans le chapître précédent seul un sous ensemble de ces situations peut intervenir selon la nature du processeur.
Les Processeurs d'Exécutions et le processeur Constructeur sont concernés par les quatre situations alors que le processeur Mise à Jour n'est concerné que par les situations 1, 2 et 4.
L'examen des répercutions que ces situations amènent sur le comportement d'un processeur, on peut tirer les remarques suivantes.

Pour le processeur Mise à Jour les situations 1 et 2 ne lui sont pas nuisibles puisque les blocs qui y interviennent lui sont destinés et seront par conséquent consommés au fur et à mesure de leur passage devant celui-ci.
La situation 4 pour le Processeur Mise à Jour n'est pas également néfaste, car si telle situation il y avait, elle indiquerait qu'il n'est plus nécessaire de produire de nouveaux blocs exécutables. L'activité du processeur Mise à Jour bien que ralentie par cette situation, n'affecterait en rien les performances globales du système. En conclusion, le processeur Mise à Jour n'est nullement affecté par ces situations.
Pour le processeur Constructeur, la situation 3 ne lui est pas nuisible, car il consomme au fur et à mesure de leur passage les demandes d'exécution et cela lui permet de déposer des blocs en attente si besoin est.
L'activité du Constructeur se trouve inévitablement ralentie dans les situations 1 et 2.
En ce qui concerne la situation 4, les mêmes remarques que pour le processeur Mise à Jour peuvent être faites.
Le processeur Constructeur est donc seulement affecté par les situations 1 et 2.
Pour les Processeurs d'Exécution, les quatre situations peuvent lui être défavorables.

Cependant certaines situations peuvent être plus nuisibles que d'autres. Les situations 1 et 2 ne surviennent que pendant un laps de temps assez court car les Processeurs d'Exécution ne déposent qu'un seul domaine de sortie ou bloc en attente au cours de l'exécution d'un bloc.

La situation 3 peut s'établir pendant une durée assez longue car un Processeur d'Exécution peut déposer plusieurs demandes d'exécution lors de l'exécution d'un bloc.
C'est cette situation que nous considèrerons par la suite, qui bien que n'entraînant pas un blocage du système (i.e les demandes d'exécution sont consommées en temps réel par le processeur Constructeur). Nous parlerons alors de situation de famine lorsqu'un processeur ne peut exécuter un dépôt car il ne voit passer que des cadres occupés.

La situation 4 peut entraîner une situation de blocage du système lorsque, les processeurs étant tous actifs avec une file de demande d'exécution pleine, ceux-ci exécutent une nouvelle fois la primitive EXECBLOC. Nous donnons dans ce qui suit un exemple de situation de famine et un exemple de situation de blocage.

A.1 Un problème de famine

Nous présentons ici un programme de simulation dans lequel on va lancer en parallèle un grand nombre de blocs exécutables de manière à illustrer la situation de famine évoquée plus haut.

Le programme d'essai que nous allons considérer est un programme de simulation du comportement d'un transistor unipolaire. Ce programme permet, par simulation du comportement des porteurs majoritaires à l'intérieur d'un transistor, d'obtenir les caractéristiques du composant considéré. La simulation du comportement de chaque porteur majoritaire est basée sur une méthode probabibiste de Monté-Carlo.

L'algorithme général utilisé pour cette simulation peut être décrit de la façon suivante:

Programme Caractéristiques d'un Transistor

Début : - Initialisations

- Pour t variant de 0 à T par pas de dt

Faire _ Pour M variant de 1 à N par pas de 1

Faire - Calcul de la trajectoire du porteur majoritaire de numéro M par la méthode de Monte-Carlo.

Fait

Si ¬ test convergence

alors Calcul des nouvelles composantes du champ électrique au temps t par résolution d'un système de Poisson

fsi

Fait

Fin

Dans ce programme le calcul de la trajectoire de chaque porteur (N au total) peut être fait de manière concurrente.
Nous proposons dans ce qui suit plusieurs versions de ce programme dans notre langage de simulation de façon à mettre en évidence certains problèmes liés au découpage des blocs pour MAUD.
Jusqu'à présent, nous avons considéré des programmes d'essais ne comportant qu'un seul niveau d'EXECBLOC, dans les différentes versions que nous allons détailler, nous considèrerons deux niveaux d'EXECBLOC.

1ère Version Trois blocs sont utilisés pour décrire le programme d'essai.

```
Bloc : Pois ;
DE : ;
DS : ;
Début ;
        [1] ;
        répéter  10 ;
                MONT ( ; A) ;
                attendre (A) ;
                [10] ;
        fin répéter ;
fin ;
```

avec

```
bloc : MONT ;
DE : ;
DS : B ;
début ;
      [1] ;
      répéter 250 ;

        ELEC ( ; c) ;

      fin répéter

      attendre * (C) ;

fin ;
```

et

```
Bloc : ELEC ;
DE
DS : D ;
début
      [12] ;
fin
```

Le bloc Pois permet de lancer l'exécution du bloc MONT qui effectue le calcul des trajectoires des porteurs par la méthode de Monté Carlo,puis effectue la résolution du système de Poisson.
Le bloc MONT lance en parallèle l'exécution du calcul des trajectoires des porteurs (ici on considère 250 porteurs).
Le bloc ELEC effectue le calcul de trajectoire d'un porteur majoritaire.
Le tableau de la figure 1 donne les résultats de simulation du modèle dynamique pour une configuration de 16 cadres avec 13 Processeurs d'Exécution.
Nous constatons que le parallélisme obtenu à l'exécution n'est que de 8 ceci pour les raisons que nous avons évoquées dans le chapitre précédent.
Un seul Processeur d'Exécution se trouve en famine (processeur n°8) et le gain obtenu par rapport à une machine monoprocesseur est de 83%.

Processeur n°	INACTIF	ACTIF	ATTENTE	
1	606	5392	0	
2	918	4800	280	
3	918	4800	280	
4	1048	4680	270	
5	1038	4680	280	
6	1038	4680	280	
7	1248	4440	310	
8	1698	1680	26 20	PROCESSEUR EN FAMINE
9	5998	0	0	
10	5998	0	0	
11	5998	0	0	
12	5998	0	0	
13	5998	0	0	

NOMBRE DE CADRE DE L'ANNEAU : 16
NOMBRE DE PROCESSEUR : 13

TEMPS TOTAL DE TRAITEMENT : 5998
TEMPS EQUIVALENT MONOPROCESSEUR :35152
GAIN : 83%

Fig 1

2ème Version - Afin d'améliorer le parallélisme obtenu à l'exécution, nous avons écrit une seconde version de ce programme dans laquelle le bloc POIS lance l'exécution de deux blocs MONT qui eux-mêmes lanceront l'exécution de 125 blocs ELEC, le débit des demandes d'exécution est ainsi augmenté.

Le programme devient le suivant :

```
Bloc :  Pois ;
DE : ;
DS : ;
début ;
       [1] ;
       répéter 10 ;
        MONT ( ; A1) ;
        MONT ( ; A2) ;
        attendre (A1, A2) ;
        [10] ;
       fin répéter
fin ;
```

avec

```
Bloc : MONT ;
DE : ;
DS : B ;
début ;
      [1] ;
     répéter 125 ;
        ELEC ( ; C) ;
     fin répéter
      attendre * (C) ;
fin ;
```

Le bloc ELEC reste identique à la première version.

Le tableau de la figure 2 montre les résultats obtenus pour cette version avec une configuration du système identique. Bien que le parallélisme obtenu à l'exécution soit supérieur à la version 1, le temps d'exécution reste du même ordre de grandeur.

Processeur n°	INACTIF	ACTIF	ATTENTE
1	966	5062	10
2	938	3000	2100
3	1108	3720	1210
4	1108	3720	1210
5	1088	3720	1230
6	1068	3420	1480
7	1288	2880	1870
8	1258	1800	2980
9	1258	1800	2980
10	1248	1800	2990
11	1248	1680	3110
12	1238	1440	3360
13	1238	1080	3720

PROCESSEURS EN FAMINES

NOMBRE DE CADRE DE L'ANNEAU : 16
NOMBRE DE PROCESSEUR : 13

TEMPS TOTAL DE TRAITEMENT : 6038
TEMPS EQUIVALENT MONOPROCESSEUR :35192
GAIN : 83%

Fig 2.

En fait l'examen des Processeurs d'Exécution de numéro 8 à 13 nous montre que ceux-ci sont restés en attente un laps de temps supérieur à leur temps d'activité. Nous dirons par la suite que ces processeurs sont en famine

3ème Version - Une troisième version de ce programme a été décrite de façon à généraliser les observations faites à propos de la seconde version. Le programme en est le suivant :

```
Bloc : Pois ;
DE : ;
DS : ;
Début ;
      [1] ;
      répéter 10 ;
          répéter 10 ;
             MONT ( ; A ) ;
          fin répéter ;
          attendre * (A) ;
          [10] ;
     fin répéter ;

fin
```

avec

```
Bloc : MONT ;
DE : ;
DS : B
Début ;
      [1] ;

      répéter 25 ;
           ELEC ( ; C ) ;
      fin répéter ;
      attendre * (C) ;
fin ;
```

et BLOC ELEC inchangé.

Le tableau de la figure 3 nous montre que les processeurs de numéro 6, 7, 8, 9, 10 et 12, 13 sont également en famine.

Processeur n°	INACTIF	ACTIF	ATTENTE	
1	696	5922	0	
2	1058	5210	350	
3	1118	4490	1010	
4	1318	3650	1650	
5	1768	2680	2170	
6	1658	2320	2640	PROCESSEURS EN FAMINES
7	1878	1480	3260	
8	1948	880	3790	
9	1608	650	4360	
10	1148	650	4820	
11	1268	3000	2350	
12	1198	2640	2780	PROCESSEURS EN FAMINES
13	1198	2050	3370	

NOMBRE DE CADRE DE L'ANNEAU : 16
NOMBRE DE PROCESSEUR : 13

TEMPS TOTAL DE TRAITEMENT : 6618
TEMPS EQUIVALENT MONOPROCESSEUR : 35562
GAIN : 81 %

Fig 3.

Conclusion : Nous avons mis en évidence certaines situations de famine inhérentes à l'implémentation du modèle dynamique.
De plus, nous avons constaté que le fait d'augmenter le parallélisme théorique du programme d'essai ici considéré n'augmentait en rien les performances globales de notre système, ceci étant dû aux situations de famine qu'il engendre.
Le logiciel de découpage automatique de programme en blocs pour MAUD devra tenir compte de ces observations.

A.2 Problème de blocage

Nous avons évoqué en introduction de ce paragraphe une situation de blocage dans laquelle tous les Processeurs d'Exécution, étant actifs avec leur file de demande d'exécution pleine, exécutent une primitive EXECBLOC alors que l'anneau est saturé de blocs exécutables.

Nous reprenons pour mettre en évidence ce problème la version 3 du programme d'essai décrit précédemment avec une configuration comprenant 11 cadres et 8 Processeurs d'Exécution.
Les résultats obtenus (Fig. 4) montrent qu'au temps 124 le système est bloqué et l'anneau est rempli de blocs exécutables.
Cette situation est donc dramatique pour notre système si l'on n'y prend pas garde.

```
FIN....OU....BLOCAGE  TEMPS =  124
CONFIGURATION DE L ANO
BLOC EXECUTABLE ELEC FENETRE 1
BLOC EXECUTABLE ELEC FENETRE 2
BLOC EXECUTABLE ELEC FENETRE 3
BLOC EXECUTABLE ELEC FENETRE 4
BLOC EXECUTABLE ELEC FENETRE 5
BLOC EXECUTABLE ELEC FENETRE 6
BLOC EXECUTABLE ELEC FENETRE 7
BLOC EXECUTABLE MONT FENETRE 8
BLOC EXECUTABLE ELEC FENETRE 9
BLOC EXECUTABLE ELEC FENETRE10
BLOC EXECUTABLE ELEC FENETRE11
*****************************************
TEMPS TOTAL=  124
 P * INACTIF *   ACTIF * ATTENTE *
 1 *       5 *      97 *      22 *  ATTENTE
 2 *      21 *      44 *      59 *  ATTENTE
 3 *      24 *      40 *      60 *  ATTENTE
 4 *      27 *      40 *      57 *  ATTENTE
 5 *      30 *      40 *      54 *  ATTENTE
 6 *      35 *      40 *      49 *  ATTENTE
 7 *      39 *      40 *      45 *  ATTENTE
 8 *      42 *      40 *      42 *  ATTENTE
*****************************************
```

Fig 4.

-B- Anneau Inverse

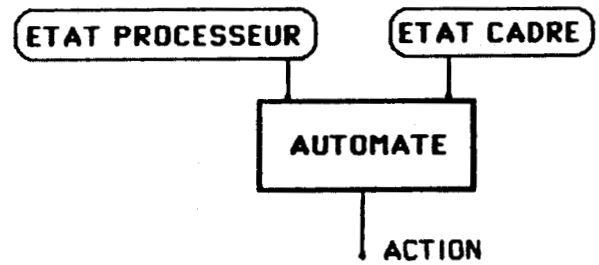
B.1. Principe

Les objectifs principaux de cet anneau inverse sont :

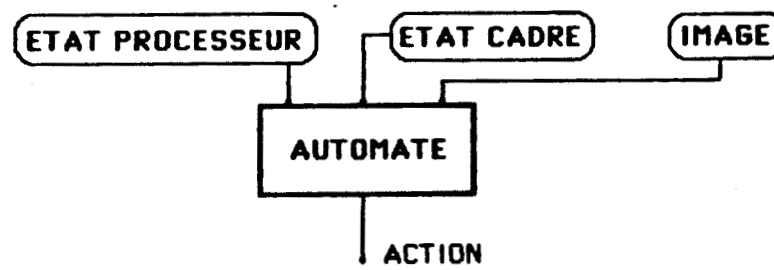
* Eviter les états d'attente prolongés d'un Processeur d'Exécution (famine)
* Répartir la charge de travail sur l'ensemble des Processeurs d'Exécution (i.e éviter les états d'Inactivité des processeurs).

De part le mécanisme de circulation de l'information à travers l'anneau, les Processeurs d'Exécution situés en tête de l'anneau sont privilégiés et donc plus prioritaires que les processeurs placés en queue de l'anneau.

L'idée de base pour atténuer cette priorité relative des processeurs est d'instaurer un mécanisme qui permette à un processeur Pi d'avoir une image des processeurs P_j (j >i) situés après lui. Cette image reçue par le processeur P_i permet alors d'influer sur le comportement de l'automate auquel il est associé. L'automate peut décider ou non de laisser passer un bloc qui l'intéresserait au profit des processeurs P_j. La figure 5 montre le mécanisme de l'automate avec et sans régulation.



a - sans régulation



b - avec régulation

figure 5

L'implémentation de cet anneau inverse est réalisée au moyen d'un ensemble de compteurs qui remonte une information sur les états des processeurs P_j qui sont situés après un processeur P_i donné (i>j) (voir Fig. 6).

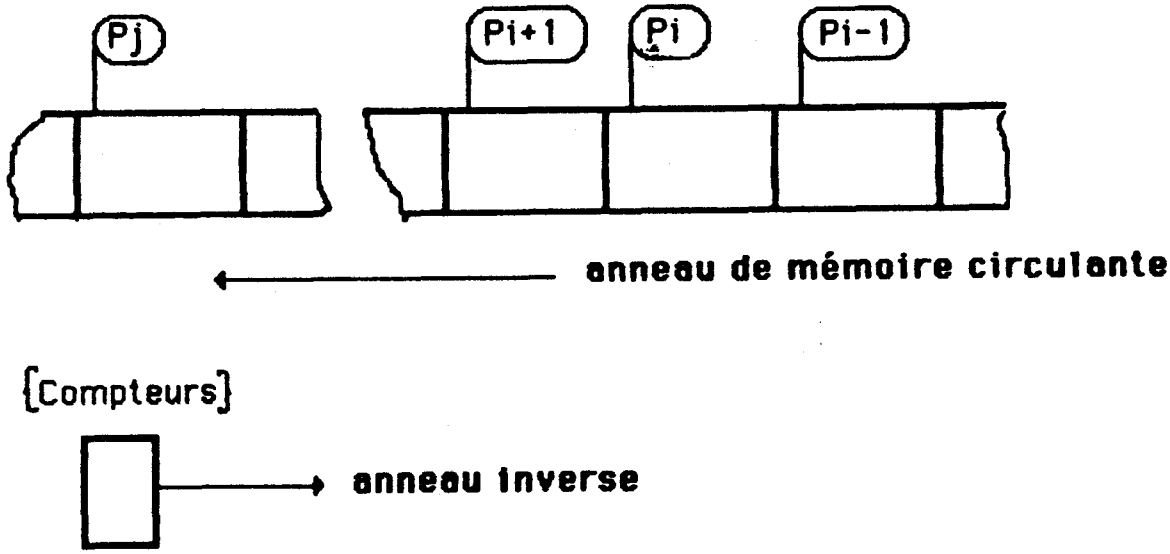


Fig. 6 Anneau Inverse

A chaque unité de temps l'automate, associé au Processeur d'Exécution P_i, a besoin de connaître l'image des processeurs P_j de rang supérieur avant d'entreprendre une action. Pour une configuration de l'anneau de mémoire circulante comportant N cadres avec une vitesse de défilement fixée conventionnellement pour notre simulation à 1 cadre/Unité de temps, il faut un anneau inverse N fois plus rapide.

Définition des compteurs

Afin d'obtenir une image précise des Processeurs d'Exécution P_j de rang supérieur à P_i nous avons associé à chaque état possible du Processeur d'Exécution un compteur. Le tableau de la figure 7 rappelle les différents états par lesquels un Processeur d'Exécution peut passer. Au total neuf compteurs sont nécessaires à l'anneau inverse. Ce tableau nous montre également que les blocs recherchés par un Processeur d'Exécution sont de deux types :

- Bloc exécutable
- ou Cadre Vide

C'est donc la prise ce ces blocs que nous allons réguler dans l'algorithme de l'automate associé au Processeur d'exécution.

L'algorithme de l'automate du processeur P_i consistera, pour chaque unité de temps, à consulter l'état des compteurs de manière à prendre une décision sur le type de transfert à effectuer ou non. Les compteurs mis à jour seront ensuite transmis à l'automate du processeur Pi-1.

Etat Processeur	Type dépôt à réaliser	Cadre Recherché	Compteur associé
Actif	rien	----	C_1
	demande d'exécution	cadre vide	C_2
Inactif	rien	Bloc exécutable	C_3
	demande d'exécution	Bloc exécutable ou cadre vide	C_4
Attente	domaine de sortie sans demande d'exécution	Bloc exécutable ou cadre vide	C_5
	domaine de sortie avec demande d'exécution	Bloc exécutable ou cadre vide	C_6
	Bloc en attente sans demande d'exécution	" "	C_7
	Bloc en attente avec demande d'exécution	" "	C_8
	File demande d'exécution pleine	Cadre Vide	C_9

Fig. 7 Etats du Processeur d'exécution

L'ensemble des compteurs est remis à zéro à chaque unité de temps.

De manière à répondre aux critères fixés plus haut, nous avons fixé une priorité relative entre ces compteurs.
Par ordre de priorité décroissante, nous avons :

- Etat d'attente pour limiter la famine (compteurs C5 à C9)
- Etat Inactif pour répartir la charge de travail sur l'ensemble des processeurs (compteurs C4 puis C3)

- Etat actif (compteurs C2 puis C1).

Les compteurs C5 à C9 qui interviennent dans l'état attente ont la possibilité de recevoir lors de la simulation une priorité relative paramétrable. Pour ce faire, on indique le type de dépôt que l'on veut rendre prioritaire, domaine de sortie, bloc en attente ou demande d'exécution.

Algorithme de l'automate

Nous avons vu précèdemment que l'automate associé au Processeur d'Exécution P_i doit tenir compte de trois facteurs avant d'efffectuer un dépôt :

* Etat du processeur P_i et du type du dépôt à réaliser
* Etat du cadre présent sous la fenêtre d'accés (cadre courant)
* Etat des compteurs C1 à C9 relatifs aux Processeurs d'Exécution P_j (j>i)

L'algorithme simplifié suivant décrit le comportement de l'automate en fonction de ces trois facteurs.

Début :

Si NON dépôt (P_i) alors Si cadre courant = (cadre vide ou bloc exécutable)
alors - Rechercher le compteur Cj = 0 le plus prioritaire qui réclame ce type de bloc et faire Cj : = Cj - 1

fsi

Sinon Si cadre recherché = cadre courant
alors Si ⌐ compteur Cj≠0 plus prioritaire que P_i
alors - laisser le cadre
- C_j : = C_j - 1

Sinon Si ⌐ compteur C_j≠0 de même priorité que P_i
alors Si Clé (P_i)=0 alors - réaliser le dépôt
Sinon - laisser le cadre
- C_j : = C_j - 1
fsi
Clé (P_i) : = clé (P_i) + 1
Sinon - réaliser le dépôt
fsi
fsi
fsi

fsi
Incrémenter le compteur C_i correspondant à l'état de P_i.
fin

Un processeur P_i qui désire effectuer un dépôt ne pourra effectivement le faire que si deux conditions sont réalisées :

* le type du cadre recherché correspond au type du cadre courant
* Il n'existe pas de processeur P_j (avec j>i) plus prioritaire que P_i

Dans le cas où il existe un processeur P_j de même priorité, une $Clé_i$ associée au processeur P_i autorise ou non le dépôt. Le rôle de cette Clé permet d'inverser la priorité relative entre deux processeurs de même catégorie (i.e même priorité). Cette Clé est aussi paramétrable dans la simulation.

Lorsqu'un processeur P_i laisse un cadre au profit d'un processeur Pk (j>i) plus prioritaire, on décrémente d'une unité le compteur Cj dans lequel Pk intervenait. De même lorsqu'un processeur P_i crée un cadre vide par suite d'une opération de capture de bloc exécutable, on met à jour le compteur C_j le plus prioritaire qui réclamait un cadre vide.

B.2 Résultats de Simulation

Nous donnons ici les résultats de simulations obtenus (figure 8) pour les versions 1, 2 et 3 du problème évoqué dans le paragraphe précédent avec une configuration du système identique (16 cadres - 13 processeurs), la priorité étant donnée au dépôt de demande d'exécution.

Nous pouvons faire les remarques suivantes :

* La charge de travail de chaque processeur (taux d'activité) est effectivement mieux répartie que sur le modèle dynamique

* Les états attentes sont répartis sur l'ensemble des processeurs.

* Les situations de famine sont diminuées, voire inexistantes. Dans la version 1 et 2 aucun des processeur n'est en famine. Dans la version 3, six processeurs demeurent en famine au lieu de 7 dans le modèle dynamique.

* Les temps d'exécution obtenus sont, pour la version 2 et 3, inférieurs à ceux obtenus dans le modèle dynamique.

Processeur n°	INACTIF	ACTIF	ATTENTE
1	3365	2882	2
2	2991	3236	22
3	3451	2725	73
4	3239	2894	116
5	3333	2786	130
6	2895	3166	188
7	3105	2974	170
8	3618	2437	194
9	2901	3104	244
10	3802	2243	204
11	3753	2185	311
12	3854	2052	343
13	3396	2468	385

PAS DE FAMINE

NOMBRE DE CADRE DE L'ANNEAU : 16
NOMBRE DE PROCESSEUR : 13
TEMPS TOTAL DE TRAITEMENT : 6249

FIGURE 8.a

Processeur n°	INACTIF	ACTIF	ATTENTE
1	2669	3199	121
2	2546	3109	334
3	2627	3024	338
4	2647	2880	462
5	2557	2833	599
6	2542	2797	650
7	2820	2377	792
8	2573	2500	916
9	2118	2670	1201
10	2423	2485	1081
11	1750	2715	1524
12	2081	2401	1507
13	2508	2202	1279

PAS DE FAMINE

NOMBRE DE CADRE DE L'ANNEAU : 16
NOMBRE DE PROCESSEUR : 13
TEMPS TOTAL DE TRAITEMENT : 5989

FIGURE 8.b

Processeur n°	INACTIF	ACTIF	ATTENTE
1	1461	3894	1086
2	1467	3543	1431
3	1773	3140	1528
4	1559	3332	1550
5	1321	2838	2282
6	1439	2549	2453
7	1172	2773	2496
8	1784	2002	2655
9	1200	2357	2884
10	1380	2531	2530
11	1524	2220	2697
12	1232	2198	3011
13	1240	2245	2956

PROCESSEURS EN FAMINES

NOMBRE DE CADRE DE L'ANNEAU : 16
NOMBRE DE PROCESSEUR : 13
TEMPS TOTAL DE TRAITEMENT : 6441

FIGURE 8.c

Fig 8.

Pour la version 2 le gain obtenu est de 0.8% alors qu'il est de 2.6% pour la version 3.

En ce qui concerne la version 1 le temps d'exécution obtenu est supérieur au modèle dynamique. Ceci est principalement dû au parallélisme peu élevé de cette version (8 Processeurs d'Exécution actifs dans le modèle dynamique).

Conclusion

Les objectifs que nous nous étions fixés pour le modèle avec l'anneau inverse sont atteints :

* Meilleure répartition de la charge sur l'ensemble des Processeurs d'Exécution
* Situations de famine diminuées.

programmes présentant un taux élevé de parallélisme (version 2 et 3).

Le gain obtenu par rapport au modèle dynamique reste néanmoins relativement faible (2.6% pour la version 3).

C. Le Gérant d'anneau

C.1 Principe

Le modèle de l'anneau inverse défini précèdemment laisse subsister le problème du blocage de l'anneau de mémoire circulante par saturation avec des blocs exécutables. Cette situation est illustrée figure 9.

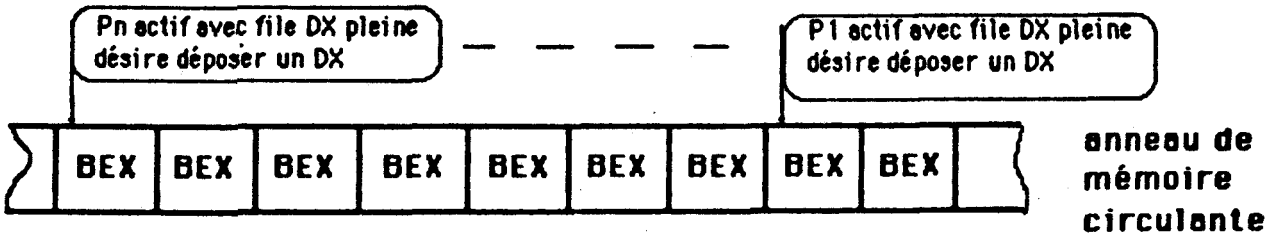


Fig. 9 Situation de blocage

Pour pallier à ce problème, nous avons imaginé l'existence d'un processeur Gérant dont le rôle serait de ne laisser subsister dans l'anneau qu'un nombre N limité de blocs exécutables susceptibles d'être capturés par les Processeurs d'Exécution. Ce nombre N peut être obtenu au moyen de l'information remontée par les compteurs de l'anneau inverse. En effet, ceux-ci nous renseignent sur l'état des processeurs et par delà sur leurs besoins (voir figure 7 compteurs C3 à C8).

La tâche du Gérant consistera donc, pour chaque unité de temps, à comptabiliser les besoins en blocs exécutables des Processeurs d'Exécution puis à insérer ou à capturer un bloc exécutable de l'anneau. Pour cela le Gérant disposera en mémoire d'une file de blocs exécutables.

Algorithme du Gérant

L'algorithme du Gérant peut être décrit de la manière suivante :

```
début
    C = C3 + C4 + C3 + C6 + C7 + C8    * on comptabilise les demandes de
                                         blocs exécutables

    Si  C ≠ 0

       alors Si cadre courant = bloc exécutable alors ne rien faire

             Sinon Si cadre courant = vide

                   alors Si File bloc exécutable non vide

                         alors - Extraire un bloc exécutable de la file
                               - L'injecter dans l'anneau

                         fsi

                   fsi

             fsi

       Sinon  Si cadre courant = bloc exécutable

              alors - capturer bloc exécutable

                    - le placer dans la file des blocs exécutables

              fsi

fin
```

Configuration du système

La configuration du modèle avec Gérant est représentée figure 10.

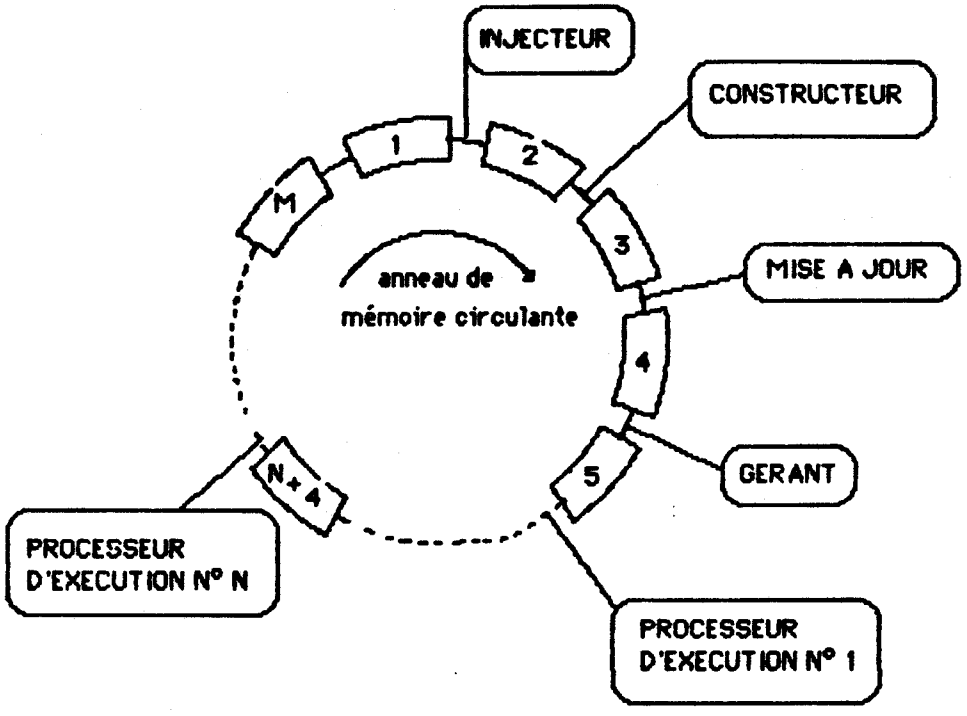


Fig. 10 Configuration avec Gérant

Sur cette représentation ne figure pas l'anneau inverse constitué des neuf compteurs.
Bien que le processeur Gérant dispose d'une fenêtre d'accés qui lui est propre, nous aurions pu intégrer sa fonction sur le site du processeur Mise à Jour puisque celui-ci dispose également d'une file de blocs exécutables en mémoire locale.

C.2 Résultats de Simulation

La situation de blocage que l'on avait mise en évidence dans le paragraphe 1 de ce chapitre n'est plus observable.

Le tableau de la figure 11 nous montre les résultats obtenus pour le même programme d'essai 3ème version avec une configuration bloquante pour le modèle dynamique (12 cadres - 8 processeurs).

Processeur n°	INACTIF	ACTIF	ATTENTE
1	970	5758	720
2	1368	5025	1055
3	1565	4671	1212
4	1740	4331	1377
5	1879	4199	1370
6	1744	4156	1548
7	1971	3826	1651
8	2242	3656	1550

PAS DE FAMINE

PLUS DE BLOCAGE

NOMBRE DE CADRE DE L'ANNEAU : 12
NOMBRE DE PROCESSEUR : 8
TEMPS TOTAL DE TRAITEMENT : 7448

Fig 11.

Les tableaux de la figure 12 présentent les résultats obtenus pour les versions 2 et 3 du programme d'essai avec une configuration de 17 cadres et 13 Processeurs d'Exécution.

Comparativement aux résultats obtenus pour le modèle avec anneau inverse on peut tirer les remarques suivantes :

- le temps d'exécution est soit du même ordre de grandeur (Version 2) ou soit inférieur (Version 3).
- les processeurs en famine sont moins nombreux (3 contre 6 pour la version 3).

Processeur n°	INACTIF	ACTIF	ATTENTE
1	2395	3460	140
2	2540	3123	332
3	2584	3003	408
4	2512	2919	564
5	2345	2977	673
6	2662	2628	705
7	2631	2523	841
8	2426	2657	912
9	2423	2570	1002
10	2383	2499	1113
11	2274	2293	1428
12	2880	2066	1049
13	1815	2474	1706

PAS DE FAMINE

NOMBRE DE CADRE DE L'ANNEAU : 17
NOMBRE DE PROCESSEUR : 13
TEMPS TOTAL DE TRAITEMENT : 5995

FIGURE 12.a

Processeur n°	INACTIF	ACTIF	ATTENTE
1	1236	4335	576
2	1562	3638	947
3	1608	3369	1170
4	1767	2994	1386
5	1499	2953	1695
6	1603	2827	1717
7	1426	2815	1909
8	1725	2411	2011
9	1596	2519	2032
10	1736	2379	2032
11	1955	2039	2153
12	2341	1895	1911
13	2815	1448	1884

PROCESSEURS EN FAMINES

NOMBRE DE CADRE DE L'ANNEAU : 17
NOMBRE DE PROCESSEUR : 13
TEMPS TOTAL DE TRAITEMENT : 6147

FIGURE 12.b

Fig 12.

Si l'on compare le temps d'exécution (version 3) obtenu avec celui du modèle dynamique, nous avons un gain de 7.1%.

Conclusion

Le modèle avec Gérant apparaît satisfaisant car il permet d'éviter les situations fatales pour notre système (i.e blocage supprimé).

Néanmoins, le modèle avec Gérant n'est rentable que pour des programmes présentant un taux élevé de parallélisme puisqu' il intègre les caractéristiques du modèle avec anneau inverse.

D. Modèle avec plusieurs anneaux

D.1 Motivations

L'objectif principal de ce modèle est de proposer une configuration de notre outil de communication (ici la mémoire circulante) qui améliorerait les performances du système de manière à approcher celles du modèle théorique avec files.

Plusieurs observations sur la simulation du modèle dynamique nous ont conduits aux remarques suivantes :

* Il est inutile de véhiculer des blocs qui ne contiennent pas d'informations (Cadre Vide)

* Il semble souhaitable d'adapter la longueur de l'anneau en fonction des besoins du système, ceci afin de réduire les temps de communication dans le cas d'un anneau sous utilisé.

* Il serait judicieux de ne faire défiler devant la fenêtre d'un processeur que des types de blocs susceptibles d'être utilisés par celui-ci.

Le modèle théorique avec files répond parfaitement à ces critères, il nous appartient de définir dans ce qui suit une configuration de la mémoire circulante qui s'apparente à ce modèle.

D.2 Présentation du modèle

Le modèle théorique avec files a permis de mettre en évidence l'existence de trois files servant à la communication inter_processeurs (voir fig. 6). Ce sont :

* la file des demandes d'exécution pour la communication entre, d'une part les Processeurs d'Exécution et le processeur Constructeur, et d'autre part l'Injecteur et le processeur Constructeur.

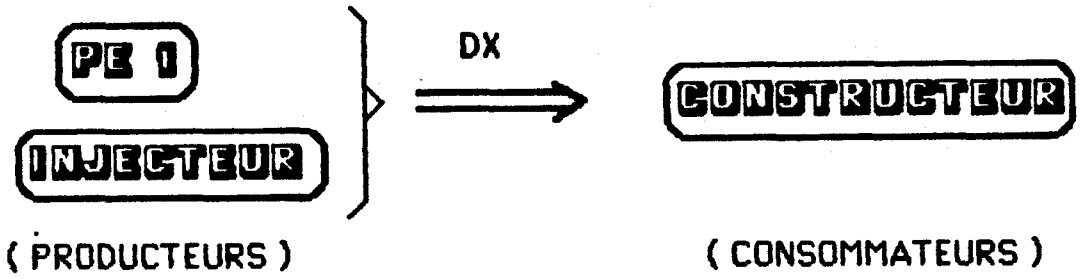
* la file des DS et BEA pour la communication entre d'une part les Processeurs d'Exécution et le processeur Mise à jour et d'autre part le processeur Constructeur et le processeur Mise à jour.

* file de blocs exécutables pour la communication entre le processeur Mise à jour et les Processeurs d'Exécution.

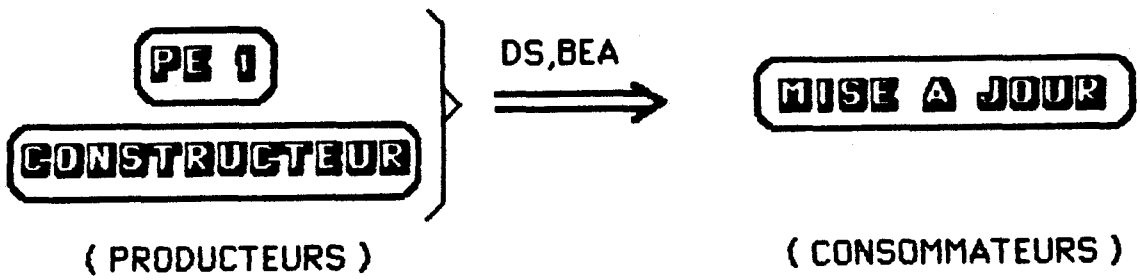
Ces trois files seront implémentées dans la mémoire circulante à l'aide de trois anneaux de communication.

La structure de ces anneaux est la suivante :

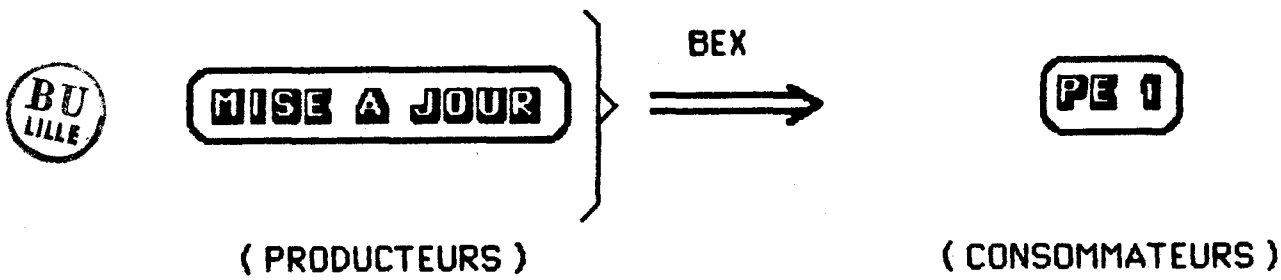
* anneau numéro 0 : contient l'ensemble des blocs destinés au processeur Constructeur (blocs DX uniquement).



* anneau numéro 1 : renferme les blocs destinés à être traités par le processeur Mise à Jour (blocs DS ou BEA)



* anneau numéro 2 : il est constitué des blocs exécutables BEX



A ces trois anneaux est associée une relation du type Producteur-Consommateurs.
Le nombre de cadres constituant ces anneaux peut être appelé à varier dynamiquement en cours d'exécution.

Il faut noter cependant que chacun des anneaux comporte au minimum un cadre (i.e module de mémoire circulante) qui symbolise la tête de chacune des files que l'on a implémentées. Cette tête est rendue nécessaire pour éviter des problèmes de conflits d'accés multiples entre les différents processeurs.
Soit N le nombre de cadres constituant la mémoire circulante, N_0, N_1 et N_2 le nombre de cadres constituant respectivement l'anneau 0, 1, 2 la relation suivante est toujours vérifiée :

$$\sum_{i=0}^{2} N_i \leq N \quad \text{et} \quad 1 \leq N_i \leq (N-2) \; \forall \; i$$

D.3 Etude fonctionnelle du modèle

La structure des anneaux étant définie nous pouvons examiner le comportement du système dans une telle structure.

Les Processeurs d'Exécution

A chaque processeur est associé comme auparavant une fenêtre d'accés sur la mémoire circulante et un cadre lui permettant d'effectuer des échanges avec celle-ci (fig. 13).

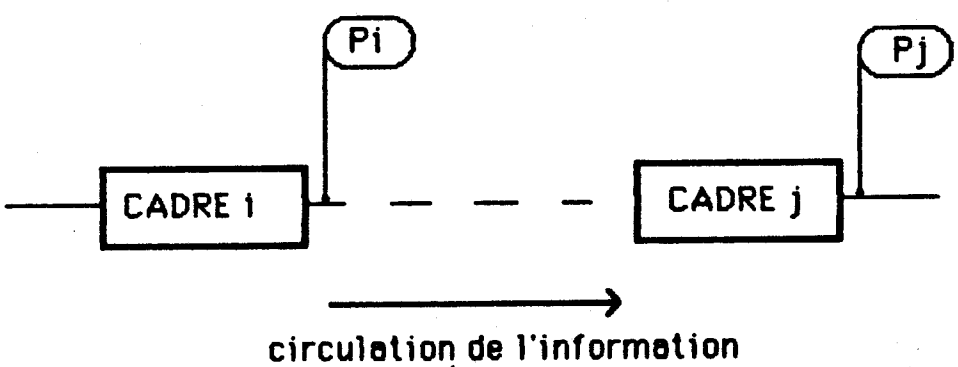


Fig. 13 Les Processeurs d'Exécution

La particularité de ce nouveau modèle vient du fait que le cadre i est contrôlé par le processeur P_i qui décide de l'insertion du cadre dans tel ou tel anneau. Celui-ci peut en outre effectuer les opérations suivantes :

* Insertion du cadre $_i$ dans l'anneau $_j$ avec $j \in [0,1,2]$
* Isolation du cadre $_i$ de la mémoire circulante par exemple dans l'hypothèse où le cadre est vide et que le processeur P_i ne désire pas effectuer des transferts.

Règles de transfert

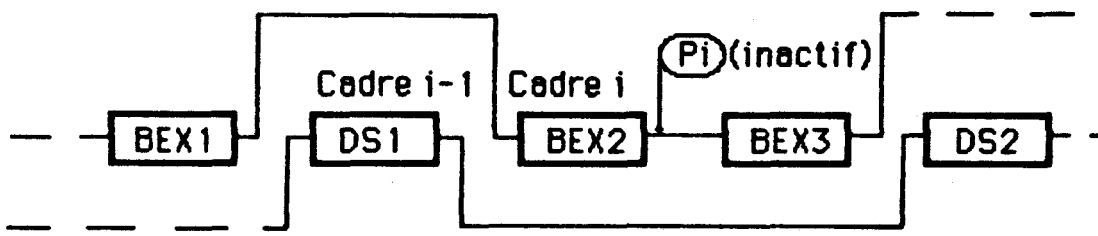
Un processeur peut effectuer des transferts de plusieurs types qui sont les suivants :

* dépôt : c'est un échange d'informations entre la mémoire locale du processeur et la mémoire circulante symbolisé par ML → MC
* capture : c'est l'opération inverse de la précédente symbolisée par MC → ML
* echange : c'est la réunion des deux opérations précédentes que l'on symbolise par MC ↔ ML

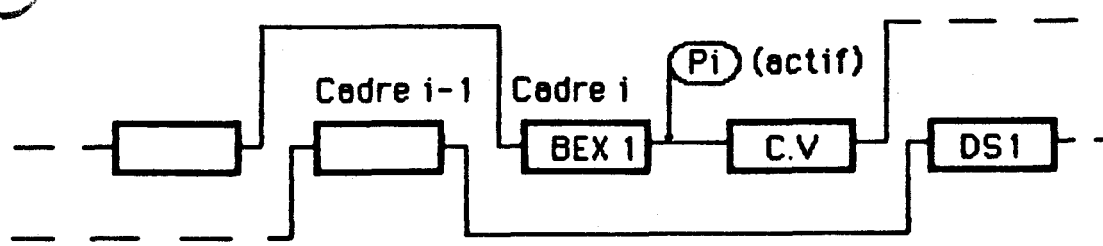
Les règles qui régissent les opérations de transferts d'un processeur sont décrites ci-dessous.

DEPOT : Pour effectuer un dépôt un processeur P_i doit disposer :
* soit d'un cadre $_i$ à l'état isolé
* ou soit d'un cadre $_i$ vide

CAPTURE : Un processeur P_i qui désire capturer un bloc doit disposer de ce bloc à l'intérieur de la cellule qu'il contrôle.
Après avoir capturé un bloc de l'anneau, il faut injecter un cadre vide afin d'assurer la circulation de l'information à l'intérieur de l'anneau considéré. Nous verrons dans ce qui suit un procédé pour éviter cette opération (fig .14).



a. avant capture (temps t)



b. après capture (temps t + 1)

Fig. 14 Opération de capture

Sur cet exemple le processeur P_i inactif a inséré la cellule qu'il contrôle (cadre $_i$) dans l'anneau numéro 2 afin d'y capturer un bloc à exécuter.

ECHANGE : Une cellule ne pouvant figurer à un instant donné que dans l'un des trois anneaux, l'opération d'échange n'est réalisable que si les deux conditions suivantes sont satisfaites :

* Le cadre $_i$ du processeur P_i contient effectivement un bloc exécutable (opération de capture possible)
* Le cadre $_i$ doit être inséré dans l'anneau concerné par l'opération de dépôt c'est-à-dire pour une opération d'échange BEX-DX l'anneau N°0 alors que pour une opération d'échange BEX-DS ou BEX-BEA le cadre $_i$ est inséré dans l'anneau N°1.

Règle d'isolation d'un cadre

Un processeur P_i ne peut décider d'isoler le cadre $_i$ qu'il contrôle que si et seulement si les deux conditions suivantes sont remplies :

* Le cadre $_i$ est vide
* Le processeur P_i est actif et ne désire pas effectuer un dépôt

Dans une telle hypothèse le cadre est isolé de la mémoire circulante de manière à diminuer les temps de propagation de l'information à l'intérieur des différents anneaux.

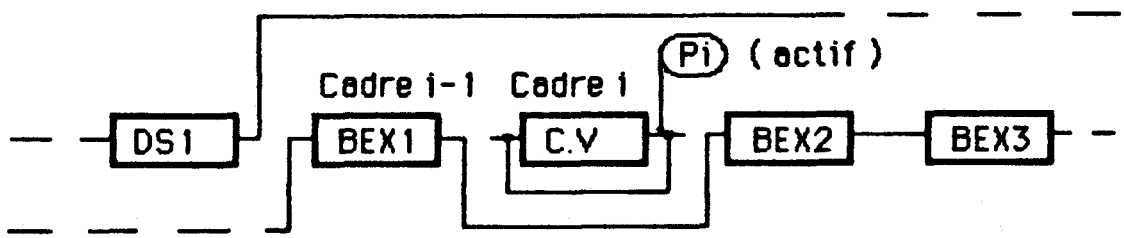


Fig. 15

La figure 15 montre une configuration possible de la mémoire circulante dans laquelle le processeur P_i actif isole le cadre $_i$.

Règles d'insertion à l'intérieur d'un anneau

Nous allons définir dans ce qui suit les règles qui régissent l'insertion d'un cadre dans tel ou tel anneau. Le critère d'insertion repose sur trois paramétres :

* l'état du cadre $_i$ que l'on considère
* l'état du processeur $_i$
* le type de dépôt que le processeur $_i$ désire effectuer

Le tableau de la figure 16 donne un aperçu des différents cas de situation qui peuvent se produire.

Nous pouvons faire les remarques suivantes :

* pour les opérations de capture de bloc BEX (Processeur Inactif sous dépôt à réaliser), la cellule (i.e cadre $_i$) contrôlée par le processeur P_i est isolée de la mémoire circulante ceci afin d'éviter de réinjecter un cadre vide dans l'anneau considéré.

* pour les opérations d'échange (BEX-DX) dans le cas d'un processeur P_i inactif avec dépôt DX, le cadre $_i$ est directement inséré dans l'anneau qui va recevoir le dépôt (ici DX → anneau 0)

* dans le cas de figure où aucune opération de transfert n'est possible, on se contente de faire circuler l'information dans le bon anneau.

Nous allons proposer dans le paragraphe suivant une implémentation possible de notre modèle.

ETAT PROCESSEUR P i	TYPE DEPOT à réaliser	ETAT CADRE i	ACTIONS à entreprendre	BITS de COMMANDE C1	C2	C3
ACTIF	**RIEN**	Isolé	aucune	X	X	1
		C.V	Isoler cadre i	X	X	1
		DS,BEA	Insertion dans anneau 1	0	1	0
		BEX	Insertion dans anneau 2	1	X	0
		DX	Insertion dans anneau 0	0	0	0
ACTIF	**DX**	Isolé	Insertion dans anneau 0, Dépot DX	0	0	0
		C.V	Insertion dans anneau 0, Dépot DX	0	0	0
		DS,BEA	Insertion dans anneau 1	0	1	0
		BEX	Insertion dans anneau 0, Echange BEX-DX	0	0	0
		DX	Insertion dans anneau 0	0	0	0
INACTIF	**RIEN**	Isolé	Insertion dans anneau 2	1	X	0
		C.V	Insertion dans anneau 2	1	X	0
		DS,BEA	Insertion dans anneau 1	0	1	0
		BEX	Isoler le cadre i, Capture du BEX	X	X	1
		DX	Insertion dans anneau 0	0	0	0
INACTIF	**DX**	Isolé	Insertion dans anneau 0, Dépot DX	0	0	0
		C.V	Insertion dans anneau 0, Dépot DX	0	0	0
		DS,BEA	Insertion dans anneau 1	0	1	0
		BEX	Insertion dans anneau 0, Echange BEX-DX	0	0	0
		DX	Insertion dans anneau 0	0	0	0
ATTENTE	**DS + DX ou DS**	Isolé	Insertion dans anneau 1, Dépot DS	0	1	0
		C.V	Insertion dans anneau 1, Dépot DS	0	1	0
		DS,BEA	Insertion dans anneau 1	0	1	0
		BEX	Insertion dans anneau 1, Echange BEX-DS	0	1	0
		DX	Insertion dans anneau 0	0	0	0
ATTENTE	**BEA + DX ou DS**	Isolé	Insertion dans anneau 1, Dépot BEA	0	1	0
		C.V	Insertion dans anneau 1, Dépot BEA	0	1	0
		DS,BEA	Insertion dans anneau 1	0	1	0
		BEX	Insertion dans anneau 1,Echange BEX-BEA	0	1	0
		DX	Insertion dans anneau 0	0	0	0
ATTENTE	**File DX pleine**	Isolé	Insertion dans anneau 0, Dépot DX	0	0	0
		C.V	Insertion dans anneau 0, Dépot DX	0	0	0
		DS,BEA	Insertion dans anneau 1	0	1	0
		BEX	Insertion dans anneau 2	1	X	0
		DX	Insertion dans anneau 0	0	0	0

Fig 16.

D.4 Implémentation du modèle à plusieurs anneaux

Deux unités sont nécessaires pour la réalisation de ce modèle. Ce sont:

* une unité de communication qui réalise les liens physiques entre les différents cadres qui appartiennent au même anneau
* une unité de commande qui décide des liens physiques à réaliser

La figure 17 donne un aperçu de la configuration

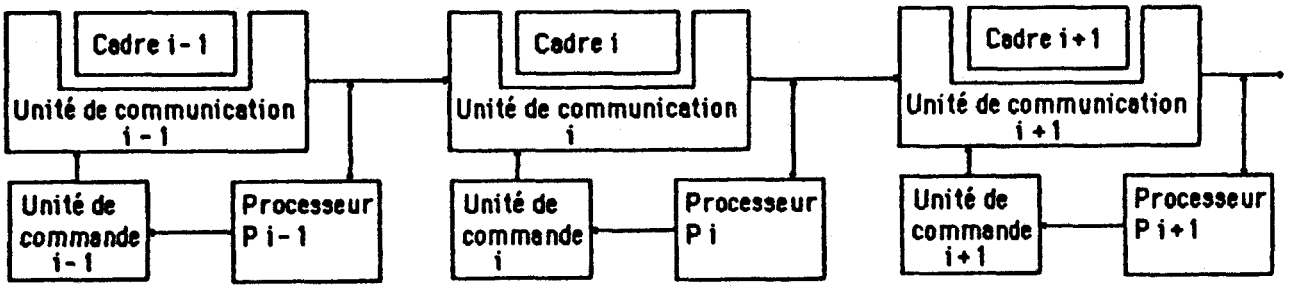


Fig. 17

Unité de communication

Nous allons dans une première partie nous intéresser à une solution avec deux anneaux puis dans une seconde partie à une solution avec trois anneaux.

a - Solution avec deux anneaux

Nous pouvons utiliser des cellules élémentaires (2x2) analogue à celle de Banyan [LIP 79] (fig.18).

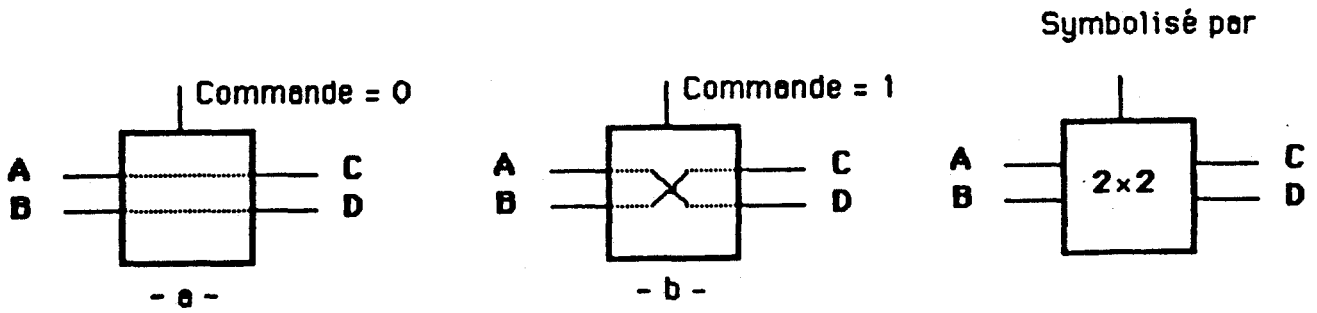


Fig. 18 Cellule élémentaire

Cette cellule permet de connecter deux entrées de 1 bit (A et B) à deux sorties de 1 bit (C et D).
La configuration (fig.18-a) relie A à C et B à D tandis que la configuration (fig. 18-b) permet de relier A à D et B à C. Cette cellule présente en outre l'avantage de commuter rapidement d'une configuration vers l'autre.
La figure 19 montre une implémentation possible à base de ces cellules élémentaires.

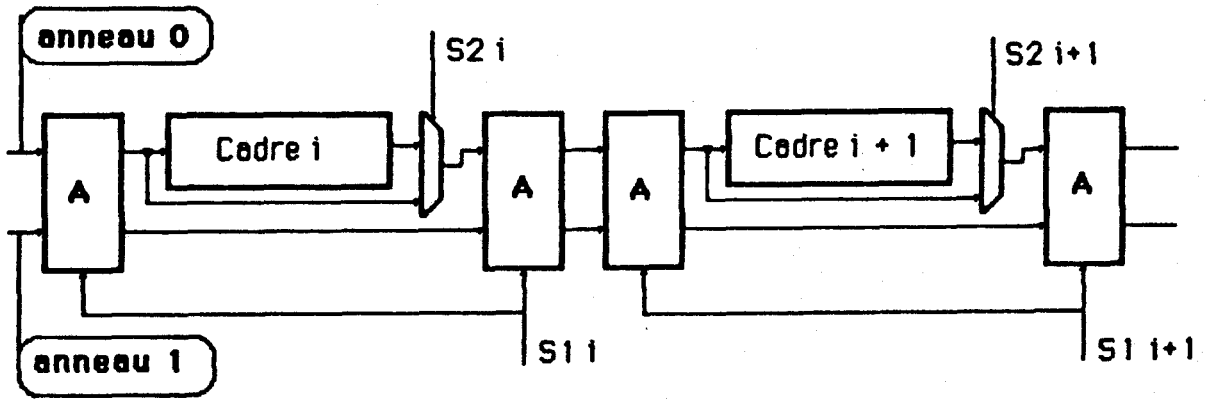


Fig. 19 Configuration avec deux anneaux

Les modules A sont des cellules élémentaires 2x2 définies précédemment. La commande $S2_i$ permet d'insérer le cadre $_i$ dans l'anneau 0 où l'anneau 1.
La commande $S1_i$ permet de commander un multiplexeur qui isole le cadre $_i$ du reste de la mémoire circulante.

b - extension à trois anneaux

Une extension de l'unité de communication à trois anneaux est possible en utilisant une cellule 4x4 construite à base de cellules élémentaires 2x2 (fig. 20).

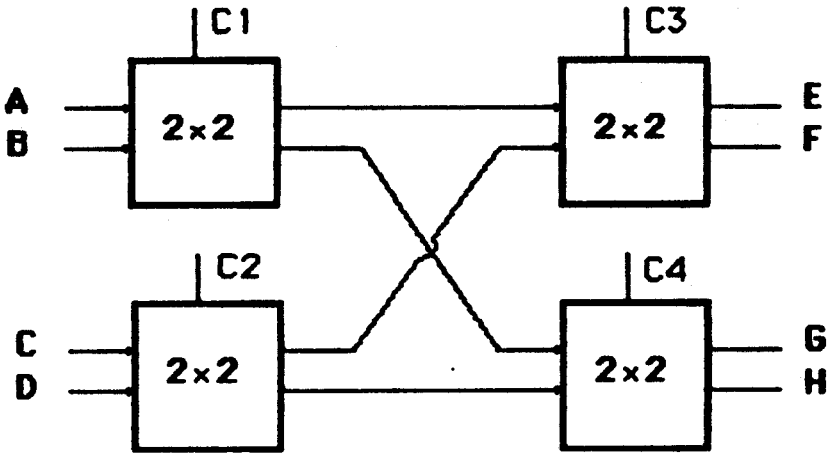
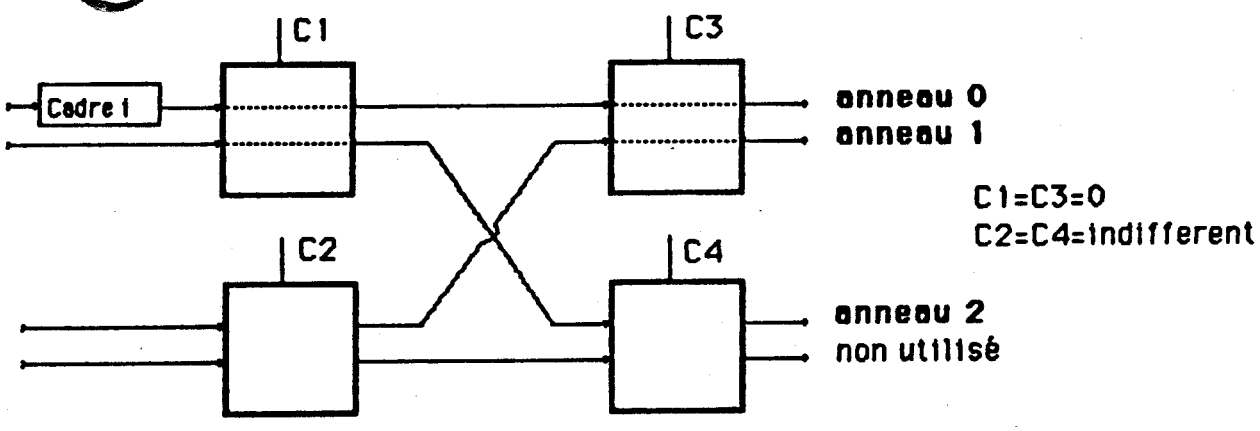


Fig. 20 Cellule 4x4

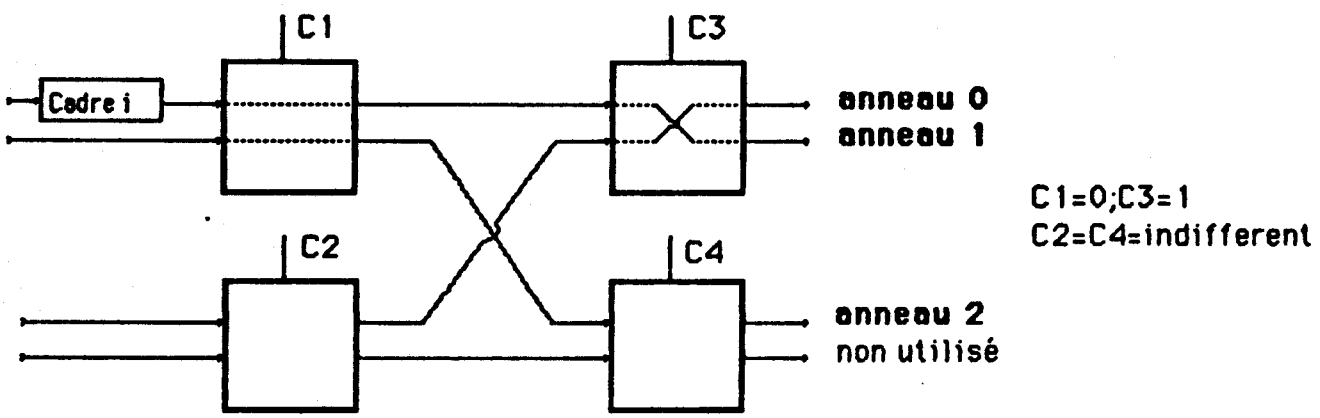
Une entrée et une sortie de cette cellule ne sont en fait pas utilisées dans le modèle à trois anneaux. Ainsi en prenant comme convention la sortie E (anneau 0), la sortie F (anneau 1) et la sortie G (anneau 2) l'insertion d'un cadre $_i$ dans l'un des anneaux se traduirait par l'une des configurations suivantes (partie gauche omise) (fig. 21) :

* cadre $_i$ dans anneau 0



- Communication de l'anneau de mémoire circulante -

* cadre $_i$ dans anneau 1



* cadre $_i$ dans anneau 2

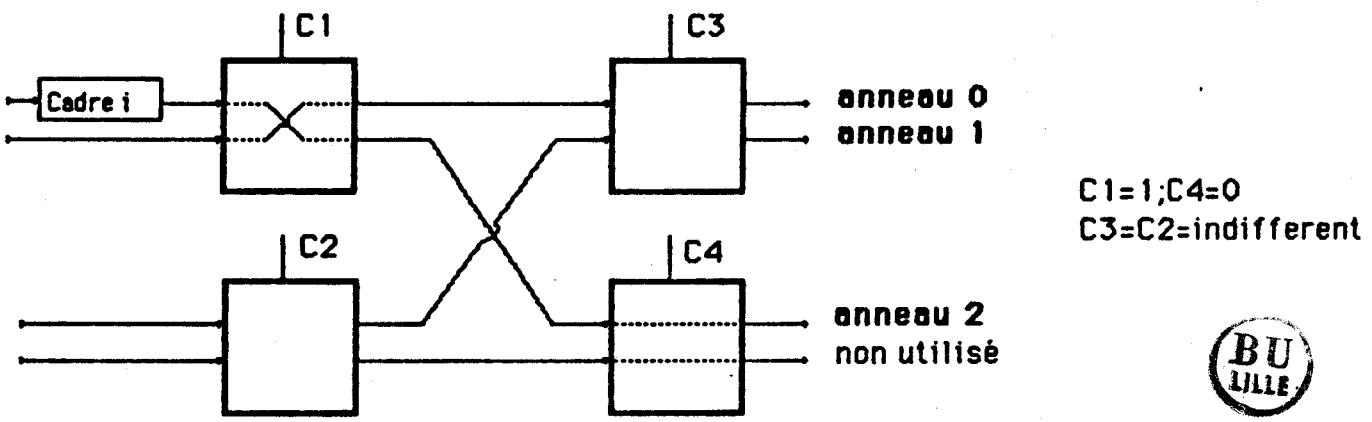


Fig. 21

En conclusion, nous pouvons remarquer que les cellules élémentaires 2 et 4 ne sont pas nécessaires dans notre application. La figure 22 représente une implémentation possible avec deux cellules élémentaires

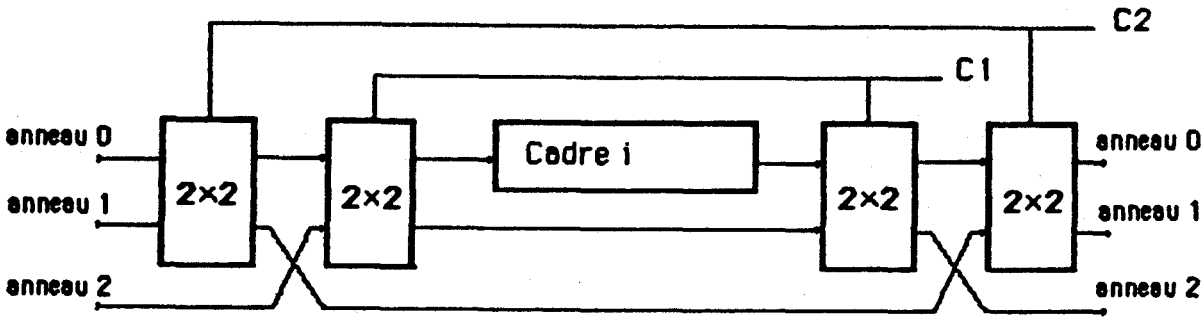


Fig. 22 Configuration simplifiée

La réalisation de l'unité de communication peut se résoudre donc de manière très simple si on utilise les cellules élémentaires que nous avons décrites.
Nous allons regarder dans le paragraphe suivant la structure de l'unité de commande qui pilote ces cellules.

Unité de commande

Cette unité est chargée de fournir les bits C_1, C_2 nécessaires à la commande des cellules élémentaires de Banyan.
Pour ce faire, l'unité de commande doit disposer si l'on se réfère au tableau de la figure 16 :

* de l'état du processeur (+ type dépôt)
* du type du cadre $_i$

Sur ce tableau nous avons fait figurer les commandes C_1, C_2 relatives aux cellules élémentaires de Banyan et la commande C_3 associée au multiplexeur d'isolement du cadre. La figure 23 montre la structure générale de l'unité de commande et de l'unité de communication

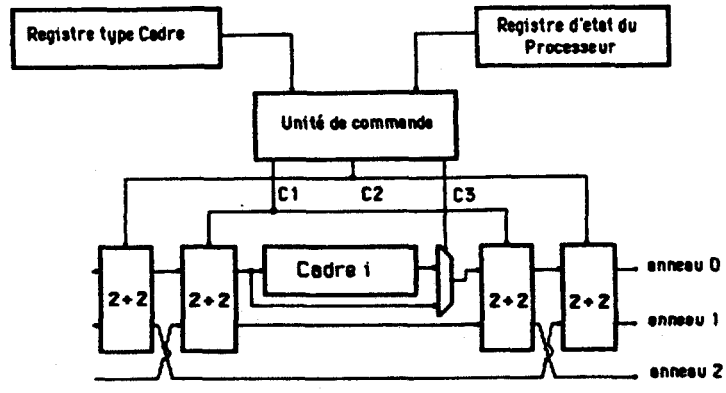


figure 23

L'unité de commande peut être réalisée facilement à partir de circuit programmable standard de type ROM ou FPLA.
La seule contrainte technologique est celle liée à la vitesse de la mémoire circulante. La figure 24 donne le diagramme temporel de l'unité de commande.

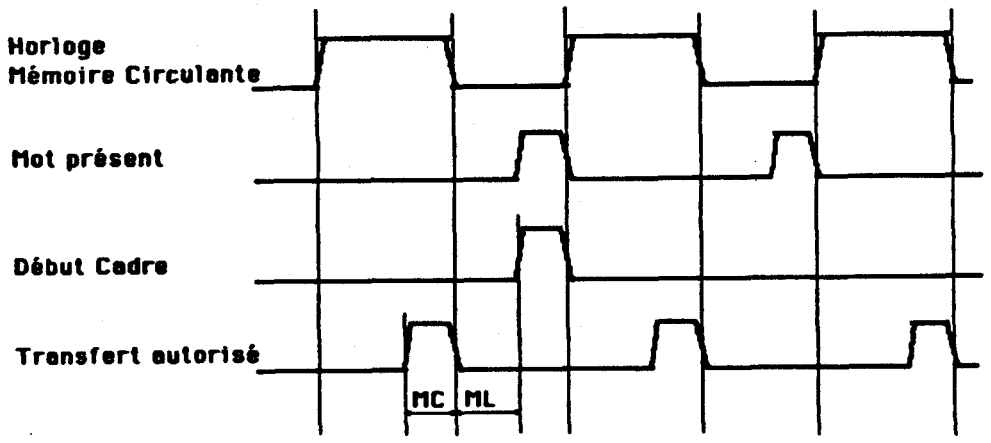


Fig. 24

Le signal mot présent précise que la donnée suivante du module de mémoire circulante est stable en sortie. On peut alors capturer la donnée si nécessaire.
Début Cadre nous indique que l'on passe sur le cadre logique suivant.
Transfert autorisé est un signal nécessaire à l'unité de transfert associée au processeur.
MC correspondant à une écriture dans la mémoire circulante (dépôt).
ML correspondant à une écriture dans la mémoire locale (capture).
L'unité de commande doit insérer le cadre dans l'un des anneaux à chaque début de cadre.
En fait il doit le faire juste un peu avant le signal début de cadre de manière à ce que la configuration de la mémoire circulante soit stable au moment où les processeurs commencent à scruter celle-ci.
L'insertion des cadres ne peut donc se faire que pendant le temps ML, c'est-à-dire que pour une horloge mémoire circulante de 1 Mhz on disposerait de 250 monosecondes pour la commande des cellules élémentaires 2x2. Ce temps est largement suffisant vu la simplicité des circuits à commander.
En conclusion, la réalisation matérielle d'un modèle comportant trois anneaux distincts, à partir d'une mémoire circulante constituée de cadres, est simple et n'utilise que des circuits classiques.

D.5 Configuration du système

La configuration totale de notre modèle est visualisée figure 25. On y trouve :

* le processeur Injecteur associé au cadre 1
* le processeur Constructeur disposant de deux fenêtres (cadre 2 et 3)
* le processeur Mise à Jour associé aux cadres 4 et 5
* les Processeurs d'Exécution et leurs cadres respectifs.

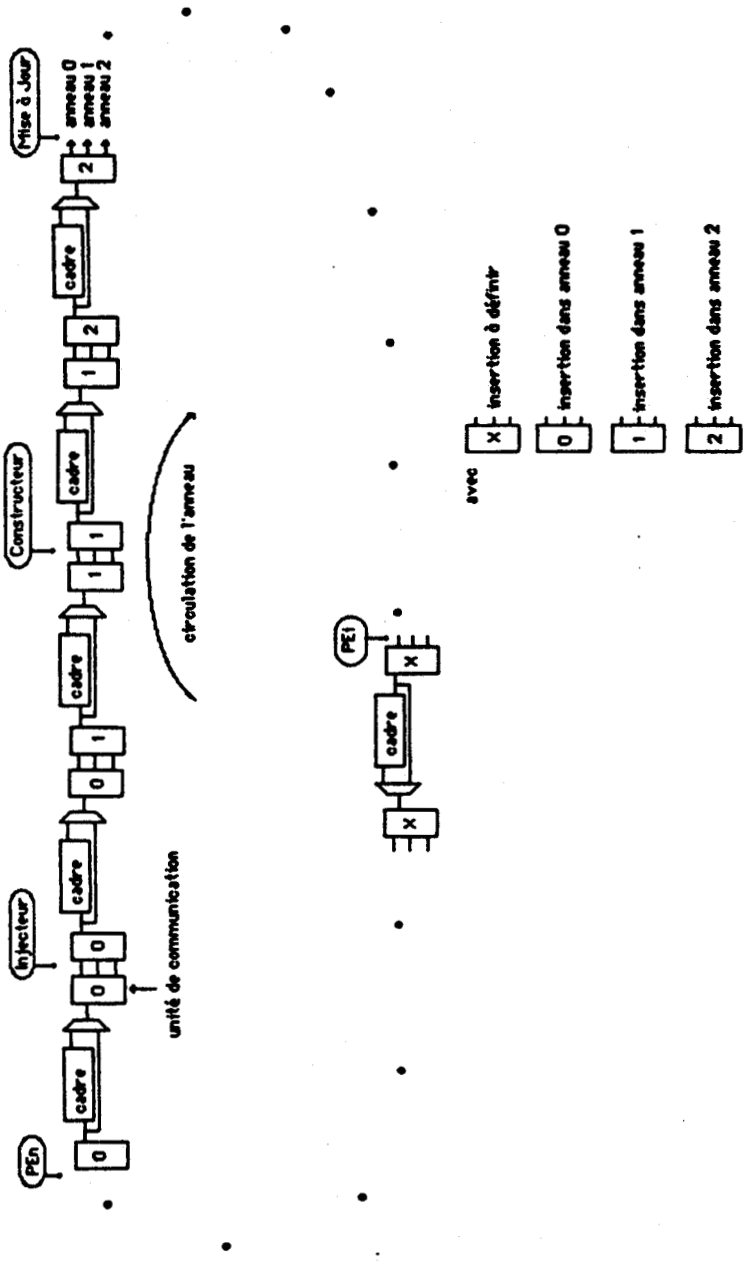


Fig 25.

Les cadres 2, 3, 5 sont des cadres spéciaux en ce sens qu'ils représentent respectivement les têtes de file des DX, DS ou BEA, BEX, des anneaux 0, 1 et 2.
Ces cadres sont toujours insérés respectivement dans l'anneau 0, 1 et 2 et ne peuvent être isolés.
Nous pouvons remarquer que sur ce modèle ne figure pas le processeur Gérant ceci pour la raison suivante :

> La fonction du Gérant qui consistait à écarter de la mémoire circulante les blocs exécutables BEX que les Processeurs d'Exécution ne réclamaient pas est ici implicitement liée à la fonction de l'anneau 2.En effet le processeur Mise à jour injecte un bloc BEX que si un Processeur d'Exécution a inséré le cadre qu'il contrôle dans l'anneau 2. L'algorithme du Mise à Jour consistera donc à injecter un bloc BEX tant qu'il existera un cadre vide dans l'anneau 2. De plus, le fait qu'un processeur P_i contrôle l'insertion du cadre $_i$, permet d'assurer une certaine régulation des opérations de transfert entre processeurs et mémoire circulante.

D.6 Résultats de Simulation

Une simulation du comportement de ce modèle a été effectuée. Nous reprenons ici les problèmes de simulation que nous avons étudiés dans ce chapitre et le précédent.

Problème TRI-FUSION

Le temps d'exécution n'est que de 438 unités de temps (fig. 26) contre 521 unités de temps pour le modèle dynamique. Le gain obtenu est donc de 15.9 %. En comparaison avec le temps d'exécution obtenu pour le modèle théorique avec files (410 unités) on est très proche des limites théorique du modèle.

Processeur n°	INACTIF	ACTIF	ATTENTE
1	98	340	0
2	150	288	0
3	168	270	0
4	178	260	0
5	178	260	0
6	178	260	0
7	178	260	0
8	236	202	0
9	280	158	0
10	298	140	0

NOMBRE DE CADRE DE L'ANNEAU : 15
NOMBRE DE PROCESSEUR : 10
TEMPS TOTAL DE TRAITEMENT : 438

Fig 26.

Problème Système Linéaire

La configuration choisie pour ce problème est constituée de 10 Processeurs d'Exécution et de 15 cadres.

Le temps d'exécution obtenu est de 1323 unités de temps qui est très proche du temps d'exécution obtenu pour le modèle avec files (1275 unités).

Le tableau de la figure 27 nous montre qu'aucun des processeurs n'est passé par l'état attente, ce qui n'était pas le cas avec le modèle dynamique.

Processeur n°	INACTIF	ACTIF	ATTENTE
1	91	1232	0
2	303	1020	0
3	303	1020	0
4	303	1020	0
5	303	1020	0
6	473	850	0
7	473	850	0
8	473	850	0
9	473	850	0
10	473	850	0

NOMBRE DE CADRE DE L'ANNEAU : 15
NOMBRE DE PROCESSEUR : 10
TEMPS TOTAL DE TRAITEMENT : 1323

Fig 27.

Le gain obtenu par rapport au modèle dynamique est de 14.7 %.

Problème caractéristique d'un transistor

1ère Version : le temps d'exécution est de 5386 unités de temps (figure 28) alors qu'il est de 5335 unités pour le modèle théorique avec files (figure 29).
Le gain obtenu par rapport au modèle dynamique est de 10.2%. Pour cette version aucun des Processeurs d'Exécution n'est passé par l'état attente.

Processeur n°	INACTIF	ACTIF	ATTENTE
1	114	5272	0
2	1066	4320	0
3	1066	4320	0
4	1066	4320	0
5	1066	4320	0
6	1186	4200	0
7	1186	4200	0
8	1186	4200	0
9	5386	0	0
10	5386	0	0
11	5386	0	0
12	5386	0	0
13	5386	0	0

NOMBRE DE CADRE DE L'ANNEAU : 18
NOMBRE DE PROCESSEUR : 13
TEMPS TOTAL DE TRAITEMENT : 5386

Fig 28.

Processeur n°	INACTIF	ACTIF	ATTENTE
1	62	5272	0
2	295	5040	0
3	295	5040	0
4	295	5040	0
5	415	4920	0
6	415	4920	0
7	415	4920	0
8	5335	0	0
9	5335	0	0
10	5335	0	0
11	5335	0	0
12	5335	0	0
13	5335	0	0

NOMBRE DE CADRE DE L'ANNEAU : 18
NOMBRE DE PROCESSEUR : 13
TEMPS TOTAL DE TRAITEMENT : 5335

Fig 29.

2ème Version : Les mêmes remarques peuvent être faites (figure 30). Le temps d'exécution n'est plus que de 5346 unités de temps contre 5386 dans la première version. Ce modèle comme le modèle à files, permet de mieux exploiter le parallélisme de cette version, ce qui n'était pas le cas des autres modèles (dynamique, anneau inverse).
Les performances sont voisines du modèle théorique (figure 31).
Le gain obtenu par rapport au modèle dynamique est de 11.4%

Processeur n°	INACTIF	ACTIF	ATTENTE
1	634	4712	0
2	786	4560	0
3	1266	4080	0
4	1386	3960	0
5	1386	3960	0
6	1986	3360	0
7	2706	2640	0
8	3306	2040	0
9	3306	2040	0
10	4386	960	0
11	4386	960	0
12	4386	960	0
13	4386	960	0

NOMBRE DE CADRE DE L'ANNEAU : 18
NOMBRE DE PROCESSEUR : 13
TEMPS TOTAL DE TRAITEMENT : 5346

Fig 30.

Processeur n°	INACTIF	ACTIF	ATTENTE
1	1123	4112	0
2	1275	3960	0
3	1275	3960	0
4	1395	3840	0
5	1515	3720	0
6	1515	3720	0
7	1515	3720	0
8	1635	3600	0
9	3075	2160	0
10	3195	2040	0
11	5115	120	0
12	5115	120	0
13	5115	120	0

NOMBRE DE CADRE DE L'ANNEAU : 16
NOMBRE DE PROCESSEUR : 13
TEMPS TOTAL DE TRAITEMENT : 5235

Fig 31.

3ème Version : Dans cette version nous avons un taux très élevé de parallélisme, c'est pourquoi les résultats de la figure 32 font état d'un processeur en famine (le numéro 13). Néanmoins, cette famine n'est pas gènante pour le temps total d'exécution (5546 unités) puisque celui-ci avoisine celui obtenu avec le modèle théorique (5485 unités voir figure 33). Le gain obtenu par rapport au modèle dynamique est, dans ce cas, de 16.2 %.

Processeur n°	INACTIF	ACTIF	ATTENTE	
1	964	4582	0	
2	1406	4140	0	
3	2116	3430	0	
4	2216	2940	390	
5	2396	2680	470	
6	2496	2200	850	
7	2546	3000	0	
8	2426	2320	800	
9	2526	1840	1180	
10	2266	1720	1560	
11	2666	2880	0	
12	2776	2770	0	
13	2646	1120	1780	PROCESSEUR EN FAMINE

NOMBRE DE CADRE DE L'ANNEAU : 18
NOMBRE DE PROCESSEUR : 13
TEMPS TOTAL DE TRAITEMENT : 5546

Fig 32.

Processeur n°	INACTIF	ACTIF	ATTENTE
1	1113	4372	0
2	1355	4130	0
3	1485	4000	0
4	1445	4040	0
5	1525	3960	0
6	1605	3880	0
7	1765	3720	0
8	2565	2920	0
9	4245	1240	0
10	4605	880	0
11	4605	880	0
12	4645	840	0
13	4725	760	0

NOMBRE DE PROCESSEUR : 13
TEMPS TOTAL DE TRAITEMENT : 5485

Fig 33.

Conclusion

Nous avons mis en évidence dans ce chapitre quelques inhérences du modèle dynamique simulé dans le chapitre V.

Des trois modèles que nous avons été amenés à étudier pour pallier à ces carences, le modèle avec trois anneaux semble, par les résultats de simulation obtenus, le plus performant et le plus proche des caractéristiques du modèle théorique avec files. De plus, sa simplicité de mise en oeuvre (cellule élémentaire 2x2) permet d'envisager son intégration dans une architecture qui se veut plus performante.

CONCLUSIONS

Les préoccupations au commencement de cette thèse étaient de deux ordres :

1. Poursuite de l'implémentation d'un prototype organisé autour d'un anneau de mémoire circulante ; cette étape s'est concrétisée par :

   - l'étude et la réalisation d'un poste Processeur d'Exécution

   - l'étude de la faisabilité du processeur Mise à Jour.

2. Analyse des performances de la machine par une simulation de celle-ci.

La première partie a permis de définir un processeur Mise à Jour fonctionnant en temps réel (i.e vitesse de la mémoire circulante) en mettant à profit les caractéristiques de la mémoire circulante.

La seconde partie nous a montré quelques insuffisances de notre implémentation :

1. le temps de propagation de l'information est relativement élevé (plusieurs millesecondes)

2. le temps d'exécution des blocs élémentaires doit être de l'ordre de grandeur du temps de propagation si on veut obtenir un parallélisme satisfaisant à l'exécution.

3. le débit avec lequel le processeur Constructeur (resp. le processeur Mise à Jour) fournit les blocs en attente (resp. les blocs exécutables) est un goulot d'étranglement par le biais duquel on ne peut atteindre à l'exécution le parallèlisme théorique d'un programme.

4. Des situations de famine pénalisent les Processeurs d'Exécution situés en bout d'anneau.

Plusieurs modèles de simulation ont permis de résoudre partiellement ces insuffisances par l'adjonction de fonctions et de mécanismes spéciaux à l'implémentation existante :

- le modèle avec anneau Inverse, et le modèle avec Gérant ont éliminé les situations de famine

- le Modèle avec 3 anneaux a résolu les problèmes liés au temps de propagation, au temps d'exécution des blocs et à la famine.

Néanmoins, le point 3 n'a pu être remédié et ceci pour deux raisons principales :

- d'une part le principe des échanges avec le format de la mémoire circulante nous permet d'injecter un seul bloc (BEA ou BEX) dans un cadre de celle-ci.

- d'autre part l'unicité du processeur Constructeur et du processeur Mise à Jour qui apparaît dans le modèle théorique est reportée au niveau de l'implémentation.

Une remise en question de certains choix par rapport à ceux de l'implémentation existante permettrait de résoudre complètement ces insuffisances. Parmi ces choix, nous pouvons citer pêle-mêle :

- une augmentation en largeur des cadres de l'anneau qui permettrait soit d'obtenir des blocs élémentaires à traiter plus importants, ou soit d'insérer plusieurs blocs dans un seul cadre.

- une augmentation dynamique et virtuelle de l'anneau en longueur qui autoriserait le processeur de Mise à Jour à retirer de l'anneau un certain nombre de blocs exécutables pour les mémoriser sur une unité de disque afin de ne pas saturer l'anneau de mémoire circulante (fig. 1). Les blocs isolés seraient réinjectés dans l'anneau, une fois la charge de celui-ci favorable.

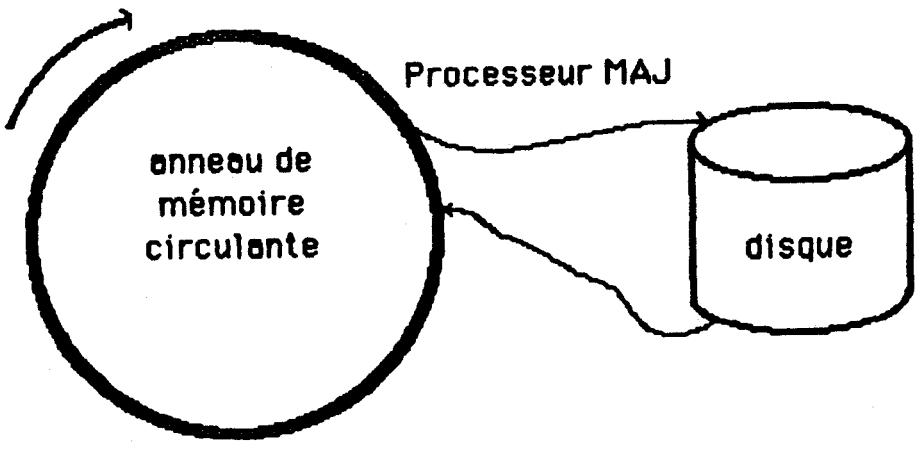


Fig. 1 Anneau virtuel de mémoire circulante

- Conclusion -

- Un élargissement de l'espace de la mémoire propre des Processeurs d'Exécution de manière à séparer la partie statique (i.e code) et la partie dynamique (i.e données) des blocs élémentaires. Chaque Processeur d'Exécution disposerait en mémoire propre de la bibliothèque des blocs à traiter. De cette façon on ne ferait transiter dans l'outil de communication que les données (i.e partie dynamique). Le format des blocs pourrait alors être réduit ce qui autoriserait la présence de plusieurs blocs dans un cadre physique de l'anneau.

- Une duplication du processeur Constructeur et du processeur Mise à Jour (fig. 2) qui permettrait d'augmenter le débit des BEX, BEA.

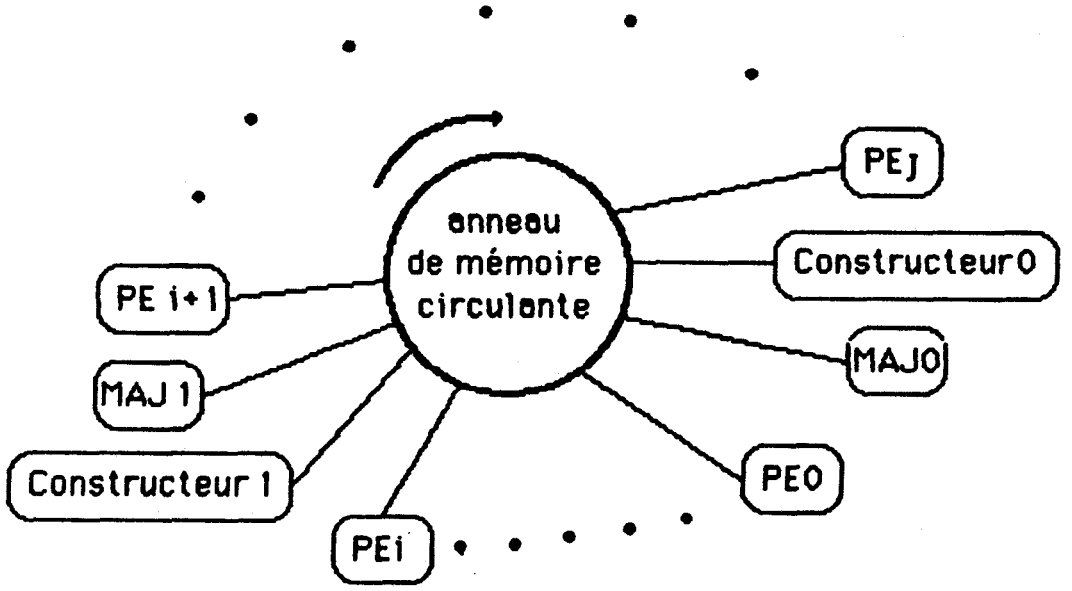


Fig. 2

La duplication du processeur Mise à Jour nécessiterait néanmoins une profonde réorganisation de celui-ci.

- L'utilisation d'un nouvel outil de communication à gestion simple comme par exemple un réseau de Transputer dont le principe de fonctionnement s'apparente par certains aspects avec le modèle MAUD.

- Une décentralisation de la fonction du processeur Constructeur et du processeur Mise à Jour qui conduirait à la définition d'un processeur "data flow" dans laquelle on intègrerait sur un site à la fois la fonction du Processeur d'Exécution, du processeur Constructeur et du processeur Mise à jour (Fig. 3) au moyen d'un "pipeline" circulaire.

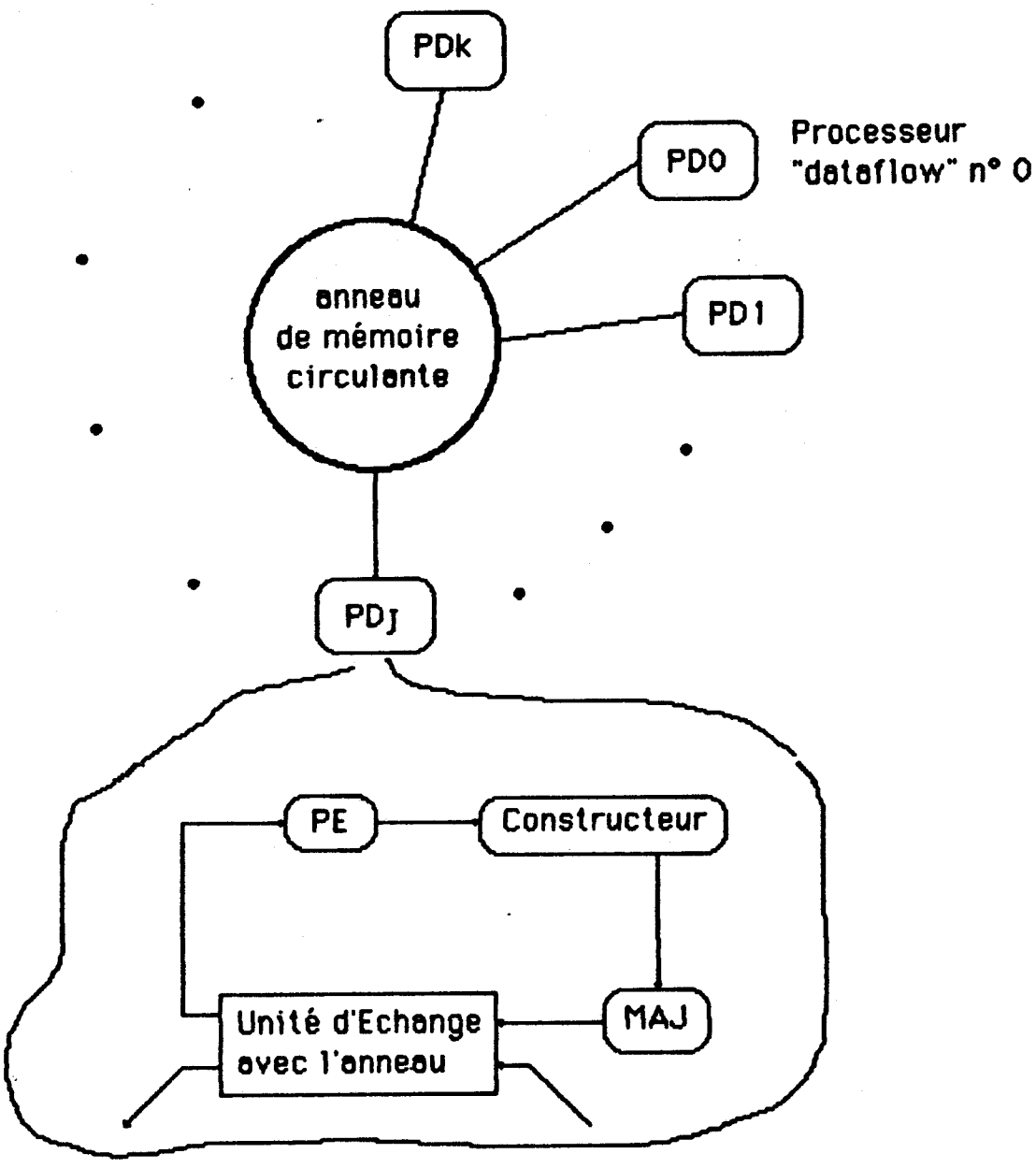


Fig.3 Processeur "Data Flow"

- Ce sont ces deux derniers points que nous nous proposons d'examiner à l'avenir pour la conception éventuelle d'un second prototype.

Parallèlement à ces aspects architecturaux, le problème du découpage d'un programme en blocs pour MAUD, fait l'objet d'une autre thèse.

De plus, il reste à étudier la façon dont l'architecture va être à même de supporter les structures de données complexes que l'on rencontre dans certains langages de programmation évolués. Faut-il confier la gestion de ces structures à un processeur spécialisé comme le fait ACKERMAN [ACK 78] ou bien définir comme l'a fait ARVIND [ARV 83] une mémoire de structure répartie sur les Processeurs d'Exécution ? Le problème reste ouvert.

- Bibliographie -

[ACK 78] **ACKERMAN W.B.** "A structure processing facility for data flow computers". Proceeding I.C.C.P. 1978

[ACK 79] **ACKERMAN W.B.** and **DENNIS J.B.** "VAL: a value oriented algorithmic language, preliminary reference manual" Laboratory for Computer Science, MIT, Tech.Report TR-218 (June 1979)

[ARV 78] **ARVIND** and all "Anasynchronous Programming Language and Computing Machine" Dept of Information and Computer Science

[ARV 80] **ARVIND** and all " a data flow architecture with taged tokens" Laboratory for computer science - MIT - Sept. 80

[ARV 83] **ARVIND** and **R.A. JANNUCCI**" A critique of multiprocessing VON NEUMANN style" Proceeding of the 10th annual symposium an Computer Architecture - 1983

[AMA 82] **AMAMIYA M.** and all "A list processing oriented data flow machine architecture" Proc of the 1982 National Computer Conference. AFIPS 1982

[ASH 77] **ASHCROFT E.A.** and **WADGE W.W.** "Lucid, a non procedural language with iteration" Communication ACM Vol 20, n°7 (July 1977)

[BUR 81] **F.J. BURKOWSKI** "A multi-user data flow architecture" Proceeding of the 8th annual symposium on Computer Architecture - May 81

[COR 79] **CORNISH M.** and others "The TI data flow Architectures : the power of concurrency for Avionics" Proceedings of the third conference on Digital Avionics Systems - Nov. 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

[BAC 78] **BACKUS J.** "Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs" Communication ACM Vol 21, n°8 (August 1978)

[BAR 83] **BARRON I.** et al "Transputer does 5 or more MIPS even when not used in parallel" Electronics vol 56 n°23 NOV 1983

[BER 75] **BERKLING K.** "Reduction language for Reduction Machines" Proc. Second. Int. Symp. Computer Architecture (1975)

[BRY 83] **BRYGIER J.** "Etude d'eléments d'architecture pour une machine langage ADA" Thèse 3eme cycle, Université de LILLE 1, Sept 1983

[BYT 81] **BYTE** "Special issue on the Smalltalk Programming Language" BYTE Auguste 81

[CLA 80] **CLARKE T.J.W.** et al "SKIM: the S,K,I Reduction Machine" Proc. Lisp 80 Conf. (1980)

[COM 76] **COMTE D., SYRE J.C.** et al "The LAU parallel system: software definition and implementation through a multimicroprocessor architecture" Euromicro 1976

[CON 63] **CONWAY M.E.** "A multiprocessor system design" Proceeding FJCC 1963

[DEN 74] **DENNIS J.B** et **MISUNAS D.P.** "A preliminary architecture for a basic data flow processor" Proc. ACM SIGARH, IEEE Symp. on Computer Architecture, DEC 1974

[DEN 80] **J.B.DENNIS** "Data flow super computers" COMPUTER - Nov. 1980

[FIS 83] **FICHER A.L.** et al "Architecture of the PSC: a programmable systolic chip" Proc. 10° Int. Symp. on Computer Architecture, June 1983.

[FLY 72] **FLYNN M.J.** "Some computer organisation and their effectiveness" IEEE Trans.Comp. Vol C21, n°9, Sept 72

[GLA 82] **GLASSET J.L** et **SOUGEN L.** "Processeur d'Exécution pour un système multiprocesseur à pilotage par les données" Mémoire d'Ingénieur ISEN 82

[GON 82] **GONCALVES G.** "An automatic exchange unit between a circulating memory and a microprocessor" Symposium Euromicro 82

[GUR 82] **J. GURD** et **I. WATSON** "A practical data flow computer" COMPUTER Feb. 1982 Vol. 15 n°2

[KEL 78] **KELLER R.M.** et al "A Loosely coupled applicative multi-processing system" Proc. Nat. Comp. Conf. (1978)

[KOW] **KOWALSKI R.** "Logic for problem solving" North Holland Press

[KOW 74] **KOWALSKI R.** "Predicate logic as programming language" Information processing 74, North Holland publishing compagny (1974)

[KIS 83] **M. KISHI, H. YASUMARA, Y. KAWAMURA** "DDDP : A Distributed Data Driven Processor" Proceeding of the 10th annual symposium an Computer Architecture - 1983

[LEC 79a] **LECOUFFE M.P.** :"Etude et définition d'un modèle de machine parallèle dynamique dirigée par les données". Thèse docteur 3ème cycle -USTL de Lille I - Juillet 1979

[LEC 79b] **LECOUFFE M.P.** :"MAUD : A dynamic single assignement system" Computers and digital Techniques - April 79 - Vol. 2 n°2

[LEC 81] **LECOUFFE M.P.** "A multiprocessor architecture using a circulating memory" 3rd Conference on the European Cooperation in Informatics: trends in Information Processing System. Munich, October 1981

[LIP 79] **G.J. LIPOVSKI, A.R. TRIPATHI** "Packet switching in Banyan" Proc. of the 6th Ann. Symp. on Comp. Arch., Philadelphie 79, pp. 160-167

[MAG 80] **MAGO G.A** "A cellular computer architecture for functionnal programming" Proc. IEEE COMPCON 80 (Feb 80)

[MAR] **MARCOUX A.,POMIAN C.** "Langages basés sur la sémantique des acteurs" Bulletin GROPLAN AFCET n° 9

[MCC 62] **Mc CARTY J.** et al "Lisp 1.5 Programmer's manual" MIT press (1962)

[MEA 80] **MEAD C.A.** and **CONWAY L.A.** "Introduction to VLSI systems" Addison Wesley 1980

[MIN 81] **MINIS ET MICROS** n°135 1981

[MIN 84] **MINIS ET MICROS** n°212 1984

[MOT 83] **TOHRU MOTO-OKA**, Overview to the fifth Generation Computer System Project, 1983 ACM

[ORG 79] **ORGANICK E.I.** "New directions in computer system architectures" Euromicro Journal vol 5 n°4, July 79

[PET 81] **PETITPREZ B.** :"Etude et réalisation d'un système multiprocesseur à pilotage par les données". Thèse de Docteur-Ingénieur - USTL de Lille I Janvier 1981

[ROC 84] **ROCACHER D.** "Etude d'une architecture dirigée par les données appliquée au langage ADA" Thèse de 3ème cycle - Université de Lille I -Oct. 1984

[RUM 77] **RUMBAUGH J.**" A data flow Multiprocessor" IEEE Transactions on Computers - Vol C 26, n°2, feb. 1977

[SAN 84] **SANSONNET J.P.** "La machine Lisp M3L" TSI vol 3 n°6 1984

[SHI 84] **T. SHIMADA, K. HIRAKI, K. NISHIDA** "An Architecture of a data flow machine and its evaluation".

[SOW 82] **M. SOWA** and **T. MURATA** "a data flow Computer Architecture with program and token Memories" IEEE Transactions on Computers - Vol. C 31 n°9 sept. 1982

[SYR 80] **J.C. SYRE** "Etude et réalisation d'un système multiprocesseur MIMD en assignation unique" 1980. Thèse de Doctorat d'Etat - Université de Toulouse

[TAK 83] **N. TAKAHASHI** and **M. AMAMIYA** " a data flow processor array system : design and analysis"

[TAV 82] **D. TAVANGARAN** "A novel Modolar Expansable Associative Memory" Euromicro Symposium - ANTWERP - Nov. 82

[TES 68] **TESLER L.G. et ENEA H.T.** "a Language design for concurrent processes" Proceedings AFIPS SJCC 1968

[TOU 79] **TOURSEL B.** "Contribution à l'étude de l'architecture des ordinateurs: propositions pour un nouveau mode de fonctionnement" Thèse docteur es sciences mathématiques, Université de LILLE 1, Septembre 1979

[TOU 81] **TOURSEL B.** "A classification of computer control modes" Juin 81, Workshop on taxonomy in computer architecture (IFIP WG 101)

[TREL 78] **TRELEAVEN P.C.** et **all** "The design of highly concurrent computing system" - University of NEWCASTLE T R - July 78

[TREL 80] **TRELEAVEN P.C.** and **MOLE G.F.** "A multiprocessor Reduction Machine for user defined Reduction Languages". Proc. seventh int. Symp. on Computer Architecture (May 1980).

[TRE 81] **TRELEAVEN P.C.** "Fifth Generation Computer Architecture Analysis" Proc. Int. Conf. on fifth Generation Computer Systems (October 1981)

[TRE 82] **TRELEAVEN P.C.** et al "Combining data flow and control flow computing" Computer Journal vol 25 n°1 ,1982

[TREL 84] **TRELEAVEN P.C.** and **ISABEL GOUVENIA LIMN** "Fuure Computers: Logic, dataflow,....., control flow" Computer march 1984

[TUR 79] **TURNER D.A.** "A new implementation technique for applicative languages" Software practice and experience vol 9 (1979)

[WAT 79] **WATSON I.** and **GURD J.** "A prototype data flow computer with token labelling" Proc. Nat. Comp. Conf. vol 48 (1979)

[WAT 83] **JGD DA SILVA** et **I. WATSON** "Pseudo associative store with hardware hashing' IEE Proc Vol. 130 PTE n° 1 - January 1983

[YOK 82] **YOKOI T.** and all "Logic programming and a dedicated high performance personal computer" Fith Generation Computer Systems, edited by Moto.Oka T. North Holland Publishing Company 1982