50376
1991
80

67018 50376
1991
80

CN 11/11

USTL
FLANDRES ARTOIS

LIFL UA 369 du CNRS

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

Numéro d'ordre : 677

Thèse

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNIQUES
DE LILLE FLANDRES ARTOIS

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

MAD : une machine virtuelle vectorielle
–
Conséquences sur l'architecture des machines vectorielles

par

Philippe Preux

Thèse soutenue le 17 Janvier 1991, devant la commission d'examen

Membres du jury :

Président, rapporteur	:	V. Cordonnier, Professeur Université Lille I
Rapporteur	:	P. Feautrier, Professeur Université Paris VI
Directeur de thèse	:	J-L. Dekeyser, Maître de conférences Université Lille I
Examinateurs	:	P. Bakowski, Professeur IRESTE - Nantes
		C. Eisenbeis, Chargée de recherche INRIA-Rocquencourt

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PAREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CONSTANT Eugène	Electronique
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie Appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean-Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilitéss
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BREZINSKI Claude	Analyse Numérique

M. BRIDOUX Michel	Chimie-Physique
M. CELET Paul	Géologie Générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. DAUCHET Max	Informatique
M. DEBOURSE Jean-Pierre	Gestion des Entreprises
M. DHAINAUT André	Biologie Animale
M. DOUKHAN Jean-Claude	Physique du Solide
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
M. FAURE Robert	Mécanique
M. FOCT Jacques	Métallurgie
M. FRONTIER Serge	Ecologie Numérique
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. LABLACHE-COMBIER Alain	Chimie Organique
M. LACOSTE Louis	Biologie Végétale
M. LAVEINE Jean-Pierre	Paléontologie
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique Atomique et Moléculaire
M. LEROY Jean-Marie	Spectrochimie
M. LHOMME Jean	Chimie Organique Biologique
M. LOMBARD Jacques	Sociologie
M. LOUCHEUX Claude	Chimie Physique
M. LUCQUIN Michel	Chimie Physique
M. MACKE Bruno	Physique Moléculaire et Rayonnements Atmosph.
M. MIGEON Michel	E.U.D.I.L.
M. PAQUET Jacques	Géologie Générale
M. PETIT Francis	Chimie Organique
M. POUZET Pierre	Modélisation - calcul Scientifique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. SALMER Georges	Electronique
M. SCHAMPS Joel	Spectroscopie Moléculaire
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. TILLIEU Jacques	Physique Théorique
M. TOULOTTE Jean-Marc	Automatique
M. VIDAL Pierre	Automatique
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne	Composants Electroniques
M. ANDRIES Jean-Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BART André	Biologie animale
M. BASSERY Louis	Génie des Procédés et Réactions Chimiques

Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique Atomique et Moléculaire
M. BERTRAND Hugues	Sciences Economiques et Sociales
M. BERZIN Robert	Analyse
M.BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie Végétale
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie Civil
M. BOIVIN Jean-Claude	Spectroscopie
M. BOUQUELET Stéphane	Biologie Appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BRASSELET Jean-Paul	Géométrie et Topologie
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie Animale
M. CATTEAU Jean-Pierre	Chimie Organique
M. CAYATTE Jean-Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie Structurale
M. CHIVE Maurice	Composants Electroniques Optiques
M. COMYN Gérard	Informatique Théorique
M. COQUERY Jean-Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques et Sociales
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique Nucléaire et Corpusculaire
M. COUTURIER Daniel	Chimie Organique
M. CRAMPON Norbert	Tectolique Géodynamique
M. CROSNIER Yves	Electronique
M. CURGY Jean-Jacques	Biologie
Melle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	Géologie Appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean-Paul	Informatique
M. DELORME Pierre	Physioliogie Animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
M. DENEL Jacques	Informatique
M. DE PARIS Jean Claude	Analyse
M. DEPREZ Gilbert	Physique du Solide - Cristallographie
M. DERIEUX Jean-Claude	Microbiologie
Melle DESSAUX Odile	Spectroscopie de la réactivité Chimique
M. DEVRAINNE Pierre	Chimie Minérale
Mme DHAINAUT Nicole	Biologie Animale
M. DHAMELINCOURT Paul	Chimie Physique
M. DORMARD Serge	Sciences Economiques
M. DUBOIS Henri	Spectroscopie Hertzienne
M. DUBRULLE Alain	Spectroscopie Hertzienne
M. DUBUS Jean-Paul	Spectrométrie des Solides
M. DUPONT Christophe	Vie de la firme (I.A.E.)
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FAUQUAMBERGUE Renaud	Composants électroniques

M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie Sturcturale
M. GAMBLIN André	Géographie urbaine, industrielle et démog.
M. GLORIEUX Pierre	Physique moléculaire et rayonnements Atmos.
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie Physique
M. GOURIEROUX Christian	Probabilités et Statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean-Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie Organique
M. GUILBAULT Pierre	Physiologie animale
M. HENRY Jean-Pierre	Génie Mécanique
M. HERMAN Maurice	Physique spatiale
M. HOUDART René	Physique atomique
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et Statistiques
M. Jean Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme (I.A.E.)
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KREMBEL Jean	Biochimie
M. LANGRAND Claude	Probabilités et statistiques
M. LATTEUX Michel	Informatique
Mme LECLERCQ Ginette	Catalyse
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
Melle LEGRAND Solange	Algèbre
M. LEGRAND Pierre	Chimie
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne
M. LE MAROIS Henri	Vie de la firme (I.A.E.)
M. LEROY Yves	Composants électroniques
M. LESENNE Jacques	Systèmes électroniques
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique théorique
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU Jean-Marie	Optique-Physique atomique
M. MAIZIERES Christian	Automatique
M. MAURISSON Patrick	Sciences Economiques et Sociales
M. MESMACQUE Gérard	Génie Mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie Organique
M. MORTREUX André	Chimie Organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
Mme MOUYART-TASSIN Annie Françoise	Informatique
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PARSY Fernand	Mécanique

M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. STEEN Jean-Pierre	Informatique

Je remercie Vincent Cordonnier d'avoir accepté de présider le jury ainsi que d'avoir rapporté cette thèse.

Je remercie Paul Feautrier d'avoir pris le temps de rapporter cette thèse. Je lui suis également reconnaissant de l'intérêt qu'il porte envers nos travaux et des critiques constructives qui m'ont permis d'améliorer la rédaction de cette thèse.

Je remercie Przemyslaw Bakowski et Christine Eisenbeis d'avoir accepté d'examiner mes travaux. Que Christine Eisenbeis soit remerciée des nombreuses remarques qu'elle a émises sur la rédaction de cette thèse lesquelles m'ont permis d'en améliorer la rédaction. Je tiens également à la remercier de l'intérêt qu'elle porte à nos travaux.

Je remercie Jean-Luc Dekeyser pour tout ce qu'il m'a donné l'occasion de faire avec lui depuis trois ans que nous travaillons ensemble. Je rends hommage à son enthousiasme comme chercheur, son charisme comme directeur de thèse et à sa confiance.

La recherche est un travail d'équipe. Aussi, j'aimerais remercier les membres de l'équipe avec qui j'ai travaillé. Je tiens tout particulièrement à exprimer ma gratitude à Philippe Marquet et Tahar Kechadi avec qui j'entretiens des rapports quotidiens.

Je remercie Joseph E. Lannutti, directeur du "Supercomputing Computation Research Institute" à Tallahassee - Floride, de m'avoir invité à venir travailler durant quelques mois dans son institut et de nous autoriser l'accès à ses supercalculateurs.

Enfin, je tiens à remercier tous mes amis qui m'ont aidé durant la réalisation de cette thèse. Quelque chose en émerge dont ils sont la source.

« as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem »

E. Dijkstra, 1972

« advances in hardware only create problems for software »

K.-C. Li et H. Schwetman, 1984

Table des matières

Introduction Générale 6

Partie I - Le traitement vectoriel I.1

Chapitre I - Le traitement vectoriel I.2

1. L'architecture des machines vectorielles I.2

1.1. Les machines tableaux I.2

1.2. Les machines pipelines I.4

1.3. Autres solutions pour les accès vectoriels I.16

2. Les langages vectoriels I.16

2.1. Le principe de conservation du parallélisme I.17

2.2. Les différents types de langages de programmation vectorielle I.17

3. Les langages machines des super-calculateurs I.19

3.1 Les supercalculateurs RISCs I.20

3.2. Les supercalculateurs CISCs I.21

3.3. Les changeants I.21

3.4. Caractéristiques communes - Conclusion I.22

4. Les machines abstraites I.24

4.1. Evolution I.25

4.2. Langages intermédiaires et machines abstraites parallèles I.27

4.3. Langages intermédiaires et machines abstraites vectoriels I.28

4.4. Conclusion sur les langages intermédiaires I.30

5. Conclusion de la partie I I.30

Partie II - MAD et Devil II.1

Chapitre II - MAD et Devil - Définitions II.2

Principes généraux II.2

1. La machine abstraite MAD et son langage Devil II.4

1.1. Introduction II.4

1.2. Description de la machine MAD II.6

1.3. Fonctionnement de la machine MAD II.16

1.4. Le langage Devil II.19

2. Exemple de programme Devil II.37

Chapitre III - MAD et Devil - Projections II.40

1. La projection de Devil II.40

1.1. Le modèle d'exécution des instructions de MAD - Définitions II.40

1.2. Ordonnancement logique II.43

1.3. Conséquences de la structure de Devil sur l'analyse physique II.46

2. L'implantation de Devil II.53

2.1. Le système de développement de programmes II.53

2.2. Projection sur stations de travail II.54

Chapitre IV - Projection de MAD par une machine-tableau II.57

1. Fonctionnalités de la mémoire vectorielle II.57

2. Modèle architectural II.58

3. Fonctionnement du PAV II.61

3.1. Schéma d'un accès en lecture aux éléments d'un vecteur II.61

3.2. Schéma d'un accès en écriture aux éléments d'un vecteur II.61

3.3. Les transferts vectoriels II.63

4. Description détaillée du PAV II.64

4.1. Structure générale du processeur d'adressage vectoriel II.64

4.2. Description des différentes unités du PAV II.67

4.3. Activités des unités lors d'accès vectoriels II.68

5. Définition d'un PAV modulaire II.70

6. Conclusion de la partie II II.77

Partie III - Le traitement vectoriel désordonné III.1

Chapitre V - Le traitement vectoriel désordonné III.2

1. Principe III.2

1.1. Exemple de fonctionnement TVD III.3

1.2. Les accès mémoires vectoriels ordonnés III.4

1.3. Les accès mémoires vectoriels désordonnés III.5

2. Justification du TVD III.6

3. Adaptations matérielles pour le support du TVD III.6

3.1. Adaptation des registres vectoriels III.7

3.2. Adaptation des unités fonctionnelles de calcul III.7

3.3. Adaptation de la mémoire et des ports mémoires III.8

4. Exemple de traitement vectoriel désordonné III.10

5. Simulations - Résultats III.12

5.1. Gains attendus du traitement désordonné III.12

5.2. Importance du temps d'occupation des bancs III.13

6. Conclusion sur le traitement vectoriel désordonné III.14

Conclusion Générale

Références bibliographiques

Bibliographie

Annexe - Les instructions de Devil

Introduction Générale

Le projet WEST développé au Laboratoire d'Informatique Fondamentale de Lille s'intéresse au traitement vectoriel. Dans ce cadre, il étudie d'une part les architectures vectorielles, et d'autre part les problèmes posés par leur programmation.

Les architectures des ordinateurs évoluent depuis leurs origines afin d'obtenir des vitesses de traitement de plus en plus élevées. Ces vitesses de traitement sont les seules qui intéressent les utilisateurs (essentiellement des scientifiques - physiciens, mathématiciens, ... - en ce qui concerne les machines vectorielles). Quelle que soit la "beauté" de l'architecture de la machine qu'ils utilisent, ils veulent avant tout que leurs programmes s'exécutent le plus rapidement possible. Partant de là, chaque génération d'ordinateurs a en point de mire la génération suivante, celle des supercalculateurs du moment. La dénomination de supercalculateurs n'est pas absolue : on s'accorde pour appeler supercalculateurs une machine possédant des performances d'un ordre de grandeur supérieure aux performances des machines "usuelles". Les volumineux supercalculateurs des années soixante font pâle figure face aux stations de travail d'aujourd'hui.

Malgré les qualités reconnues et réelles de Fortran (qualité du code généré, simplicité d'utilisation, ré-utilisation des programmes existants), des recherches sont actuellement menées sur des langages qui permettraient l'expression (aisée) d'algorithmes vectoriels complexes. Idéalement, un programme exprimé dans un tel langage est portable puisque le but poursuivi est plutôt d'exprimer l'algorithme que la manière de l'implanter. La portabilité interdit alors d'exprimer dans les algorithmes des astuces d'implantation pour que leurs compilations soient améliorées sur une architecture donnée.

Dans le cadre de notre projet logiciel, nous étudions des langages qui autorisent le programmeur à exprimer la connaissance qu'il possède sur l'algorithme qu'il programme (il indique ce qui est vectoriel, parallèle). Cette connaissance concerne également les données traitées (informations sur les dépendances entre données, leur allocation dans une mémoire hiérarchisée (distribuée), les types d'accès aux éléments des vecteurs).

Ce document présente des travaux qui ont été réalisés tant dans le domaine des langages vectoriels, que dans le domaine des architectures vectorielles. Il se décompose en trois parties distinctes. La première situe le contexte des travaux. Le deuxième présente un langage vectoriel intermédiaire et sa machine virtuelle. La troisième, sans lien réel avec la précédente, présente un modèle original d'exécution des opérations vectorielles.

Etant donné que ce document traite tant de langage que d'architecture, l'état actuel de ces deux aspects est présenté dans la première partie. Nous nous appuyons sur ce pré-requis dans les deux

autres parties. Les architectures vectorielles sont présentées. Nous nous intéressons particulièrement aux supercalculateurs vectoriels pipelines, machines les plus utilisées à l'heure actuelle lorsqu'il s'agit d'obtenir des performances en vitesse de traitements qui soient élevées. Les facteurs limitant leurs performances sont étudiés. La mémoire étant un facteur particulièrement important à ce sujet, l'organisation des mémoires de ces machines est décrite. Ensuite, le deuxième sujet de cet état de l'art est développé et concerne les langages spécifiques au traitement vectoriel. Nous présentons tout d'abord les langages de haut niveau utilisés pour la programmation des supercalculateurs vectoriels. Au niveau le plus primitif, nous décrivons les particularités des langages machines de ces machines. Une classification en est dégagée. Lors du développement d'un compilateur pour un langage visant la portabilité, une technique classique consiste à utiliser un langage intermédiaire. Par ailleurs, cette technique ayant été utilisée dans notre projet de développement du langage vectoriel EVA, les langages intermédiaires sont ensuite présentés dans cette première partie.

La deuxième partie présente le langage intermédiaire vectoriel Devil. Elle se compose des trois chapitres II, III et IV. Le chapitre II définit Devil et sa machine abstraite MAD. Il présente la structure de Devil et les choix qui ont été faits durant sa définition. Devil est un langage dont la vocation est de pouvoir s'exécuter sur de nombreuses classes d'architectures, tant scalaire, vectorielle, que parallèle. Les objets manipulés par MAD sont présentés. Le langage Devil est discuté. Dans ce chapitre, les intérêts de Devil quant à sa génération depuis un langage évolué vectoriel sont mis en évidence. Ensuite, le chapitre III discute la projection de Devil. Nous nous sommes essentiellement intéressés à la projection sur machines scalaires (station de travail) et sur supercalculateurs vectoriels pipelines (type Cray). Pour ces derniers, l'analyse du code menant à la génération d'un code qui autorise une utilisation optimale des ressources matérielles est présentée. Il est montré en quoi les caractéristiques de Devil participent à la simplification de la génération de code vectoriel. Le chapitre IV présente une réalisation matérielle de MAD. Celle-ci est effectuée dans le contexte des machines-tableaux. Nous nous intéressons particulièrement à la mémoire. Sa caractéristique essentielle est qu'elle contient des vecteurs en tant que tels. L'accès aux éléments d'un vecteur dispersé est vu comme un accès indirect vectoriel. Une réalisation matérielle de cette mémoire est proposée. Elle utilise une mémoire composée de bancs, et un processeur spécialisé dans l'accès à des vecteurs. Ce processeur permet de voir la mémoire comme une mémoire de vecteurs.

La troisième partie est composée du seul chapitre V. Celui-ci introduit un modèle d'exécution vectoriel pipeline original. Suite à une analyse du modèle d'exécution vectoriel pipeline classique (qualifié d'*ordonné*), nous proposons une technique permettant une amélioration des performances des calculateurs vectoriels. L'idée de base est de traiter les composantes d'un vecteur dans un ordre quelconque (leur ordre de disponibilité) et non dans l'ordre de leurs indices croissants, comme cela est effectué habituellement. L'implantation de ce modèle d'exécution sur un processeur vectoriel pipeline de type Cray X-MP est décrite. Les modifications matérielles à apporter aux unités du processeur sont décrites. Des résultats de simulations démontrent l'intérêt pratique de ce modèle d'exécution.

Partie I - Le traitement vectoriel

Dans cette première partie, nous présentons les différents aspects du traitement vectoriel. Nous examinons son côté matériel, puis son côté programmation.

Dans un premier temps, nous présentons l'architecture des machines vectorielles. Nous mettons l'accent sur l'organisation mémoire des machines vectorielles. Ensuite, nous étudions le développement de logiciels sur supercalculateurs, ceci à différents niveaux. Cette étude commence au niveau le plus élevé, niveau auquel travaillent la plupart des scientifiques utilisant ces machines, qui concerne les langages évolués de programmation vectorielle. Ensuite, au niveau le plus primitif, nous nous intéressons aux contraintes architecturales en présentant les langages machines de quelques supercalculateurs vectoriels pipelines. Dans cette présentation, nous nous appuyons sur une description des architectures de ces calculateurs. Enfin, à un niveau médian, nous nous intéressons aux langages et représentations intermédiaires d'un point de vue général (non restreint au domaine vectoriel). Ceux-ci établissent le lien entre langages de haut niveau et langages assembleurs.

Chapitre I
Le traitement vectoriel

1. L'architecture des machines vectorielles

Nous ne nous étendrons pas dans ce document sur une nème présentation du traitement vectoriel et des supercalculateurs vectoriels, notamment en ce qui concerne l'utilité et l'opportunité de tels traitements. Une littérature volumineuse et d'excellente qualité est disponible. Nous citerons simplement ici quelques références à des ouvrages traitant des architectures et de l'algorithmique vectorielle et parallèle tels [HWANG & al. 84a], [LAZOU 86], [STONE 87] et [HOCKNEY & al. 88].

Nous préférons restreindre l'exposé au strict nécessaire et introduire (ou plutôt rappeler) les notions indispensables à la lecture de ce document. Dans un premier temps, les modes élémentaires de fonctionnement vectoriel seront décrits, suivis de l'organisation mémoire des supercalculateurs. Nous nous intéresserons ici aux problèmes de conflits d'accès aux données en mémoire et aux solutions qui y ont été apportées.

Deux grandes familles de machines vectorielles sont à distinguer, les machines tableaux (*array-processors*) et les machines pipelines. Nous présentons les premières rapidement et nous nous arrêtons plus longuement sur les secondes. Ces dernières nous intéressent plus particulièrement dans la suite de ce document.

1.1. Les machines tableaux

1.1.1. Principes

La première famille de machines vectorielles est couverte par les machines tableaux. Celles-ci sont à inclure parmi les architectures SIMD dans la classification de Flynn [FLYNN 72]. Un modèle des architectures SIMD est présenté dans [SIEGEL 79]. Une présentation du mode de fonctionnement SIMD et de différents processeurs SIMD peut être trouvée dans [HWANG & al. 84a] et [HOCKNEY & al. 88].

Le principe des machines tableaux repose sur la multiplication du nombre d'unités de calculs (ou processeurs élémentaires - PEs). Toutes ces unités travaillent en parallèle à la réalisation d'une

même tâche et effectuent la même instruction en même temps. Chacune produit une composante du résultat. Un processeur maître contrôle l'activité des processeurs élémentaires. Les machines tableaux sont décrites dans [ZAKHAROV 84].

Une sous-classe des machines tableaux est celle des machines associatives. Celles-ci sont composées de multiples processeurs élémentaires et utilisent une mémoire associative. Les processeurs associatifs sont décrits dans [YAU & al. 77]. Nous citerons en exemple STARAN de Goodyear Aerospace et PEPE de Burroughs [HWANG & al. 84a].

Parmi les processeurs tableaux, nous citerons en exemple ILLIAC IV de Burroughs [BARNES & al. 68], MPP de Good Year [BATCHER 80], BSP de Burroughs [KUCK & al. 82], la Connection Machine de Thinking Machine Corporation [HILLIS 88] [TUCKER & al. 88] et le GF 11 d'IBM [BETTEM & al. 87].

1.1.2. Organisation générale des mémoires des machines tableaux

On distingue essentiellement deux modes d'organisation mémoire sur les machines tableaux (cf. fig. I.1.) selon que :

- les processeurs élémentaires se partagent la mémoire (cf. fig. I.1.a) comme sur BSP;
- les processeurs élémentaires possèdent chacun leur propre mémoire (cf. fig. I.1.b) comme sur Illiac IV, MPP, le GF 11 et la Connection Machine.

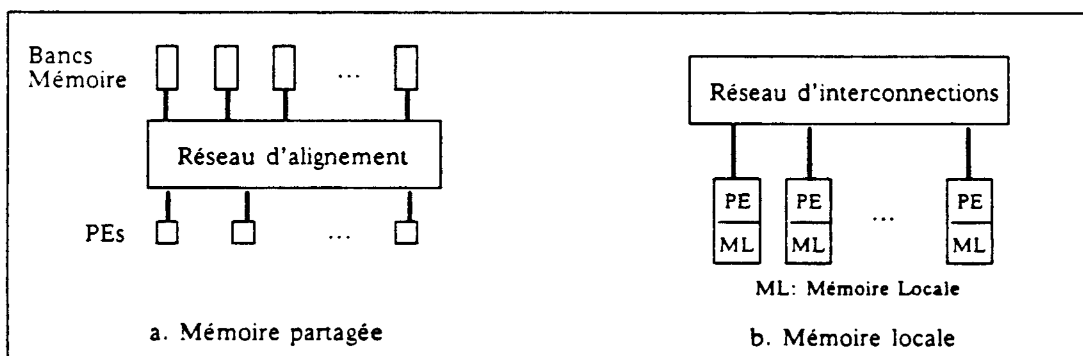


Figure I.1. -
Les deux types d'architecture tableau

Lorsque les PEs possèdent leur propre mémoire locale (ML), un réseau d'interconnections est utilisé pour les communications inter-PEs. Si les PEs se partagent la mémoire, ce réseau permet aux PEs d'accéder aux données dans les bancs mémoires (avec éventuellement un "décalage" entre les bancs et les PEs, d'où le terme de réseau d'*alignement*) ainsi que les communications inter-PEs. Une présentation des différents types de réseaux de communications pour les machines tableaux pourra être trouvée dans [FENG 81] ou dans [PREUX 88].

Machine tableau avec mémoire partagée

La mémoire est composée de bancs. Les PEs sont reliés aux bancs mémoires au travers d'un réseau d'alignement. Ce réseau d'alignement autorise l'accès en parallèle des PEs aux bancs mémoires. Ce réseau peut décaler les PEs par rapport aux bancs mémoires afin que les vecteurs puissent avoir leur première composante dans un banc quelconque. Par contre, cette architecture

est nettement moins performante dès qu'il s'agit d'accéder à des vecteurs à composantes espacées ou à des vecteurs dispersés. Sur BSP, l'accès aux composantes d'un vecteur dispersé se faisait de manière totalement séquentielle, élément par élément. L'accès aux composantes d'un vecteur régulièrement espacées est plus ou moins performant, selon le pas, pour une configuration donnée.

Bien entendu, la complexité du réseau d'alignement est à prendre en compte dans ces architectures. Pour que tout se passe au mieux, il faut que le réseau autorise toutes les permutations possibles reliant entrées et sorties (exceptées les permutations conflictuelles où deux entrées sont reliées à la même sortie). Cependant, un tel réseau devient vite très complexe matériellement, dès que les nombres d'entrées et de sorties augmentent.

Machine tableau avec mémoire locale

Dans ce type de machines, les processeurs élémentaires disposent d'une mémoire qui leur est locale. Les processeurs sont reliés par un réseau d'inter-connections. L'accès par un processeur aux données situées dans sa mémoire locale est immédiat. Par contre, pour accéder une donnée située dans la mémoire d'un autre processeur, il doit émettre une requête qui est véhiculée par le réseau d'inter-connexions. La donnée lui est envoyée de la même manière. Du fait du coût important des communications inter-processeurs, ces machines ont de bonnes performances lorsque chacun des processeurs exécute un algorithme traitant les données qui se trouvent dans sa mémoire. Par ailleurs, le coût des communications varie selon le nombre de processeurs qui sont directement accessibles. Par exemple, dans la machine MPP, chaque processeur possède quatre voisins. L'architecture de la Connection Machine est un hypercube de dimension douze. Chaque sommet est un nœud composé de 16 processeurs élémentaires. Ceux-ci sont placés sur une grille 4 x 4. Chaque nœud comprend également un routeur de messages. C'est ce routeur qui est connecté sur l'hypercube. Il envoie les messages aux processeurs de son nœud ou vers le routeur de l'un de ces douze voisins. Sur GF 11, les 566 processeurs sont reliés par un réseau ré-arrangeable Benes.

1.2. Les machines pipelines

La deuxième famille de machines vectorielles est celle des machines pipelines. Celles-ci ne recouvrent clairement aucune classe d'architectures de la classification de Flynn. Selon les machines et les auteurs, ces machines sont rangées parmi les architectures SISD, SIMD, MISD ou MIMD. Les principes de fonctionnement de ces machines sont présentés dans [RAMAMOORTHY & al. 77]. Celles-ci sont constituées à partir d'unités fonctionnelles segmentées (*pipelines*). Chaque opération est décomposée en une séquence de phases. Chacune est traitée par un segment (ou *étage*) de l'unité fonctionnelle. Plusieurs composantes d'un vecteur peuvent alors être calculées en parallèle, dans des étages différents du pipeline.

Les machines pipelines vectorielles se décomposent en deux grandes sous-classes, les machines mono-processeurs telles les Cray-1 [RUSSEL 78], Fujitsu VP-200 [MIURA 86] et Nec SX-2 [WATANABE & al. 86] et les machines multi-processeurs telles les Cray X-MP, Y-MP et 2 [THOMPSON 86] [SCHONAUER 87], IBM 3090 VF [TUCKER 86] ou la machine Cedar développée au CSRD, à l'Université d'Illinois [GAJSKI & al. 83] [JALBY 87].

De fait, l'augmentation de la puissances des supercalculateurs durant cette dernière décennie est essentiellement due à l'utilisation en parallèle de plusieurs processeurs, plutôt qu'à une diminution des temps de cycle d'horloges. Nous avons vu successivement apparaître les Cray X-MP avec deux processeurs, puis les Cray 2 et X-MP avec 4 processeurs, le Cray Y-MP avec 8 processeurs, les Nec SX-3 avec 4 processeurs et IBM 3090 VF avec 6 processeurs. On parle maintenant de 8 à 16 processeurs sur le Cray-3 et bien plus sur les futures machines Cray. On constate que l'augmentation des performances des machines est alors assez proche du nombre de processeurs.

1.2.1. Organisation générale des machines pipe-lines vectorielles

Nous présentons ici l'architecture des supercalculateurs vectoriels pipelines actuels. Il est clair que nous ne nous intéressons ici qu'aux supercalculateurs et que les mini-supercalculateurs (Convex, FPS, Alliant, ...) sont laissés de côté.

Les modes de fonctionnement vectoriel pipeline

Parmi les supercalculateurs vectoriels pipelines, deux modes de fonctionnement des processeurs sont distingués en accord avec le lieu où les unités fonctionnelles de traitements (arithmétiques, logiques et autres) accèdent à leurs opérandes. Il y a :

1. les processeurs mémoire à mémoire qui prennent leurs opérandes sources et écrivent leurs résultats directement en mémoire centrale. Les machines vectorielles antérieures au Cray-1 sont toutes des machines mémoire à mémoire. Par la suite, seul CDC a poursuivi dans cette voie (Cyber-205, ETA10);

2. les processeurs registre à registre qui prennent leurs opérandes sources et écrivent leurs résultats dans des registres vectoriels du processeur. C'est la règle générale pour les machines conçues à la suite du Cray-1 (excepté CDC). Tous les processeurs Cray et les supercalculateurs Japonais font partie de cette classe. Des instructions d'entrées / sorties réalisent le chargement du contenu des registres vectoriels depuis la mémoire et le rangement en mémoire. Le mode de fonctionnement consiste alors schématiquement en trois phases : chargement des registres vectoriels, exécution des traitements et écriture des résultats en mémoire.

Un troisième type, dérivé du précédent (processeurs registre à registre), concerne les machines registre à registre qui peuvent cependant accéder directement l'un de leurs opérandes en mémoire. Les seuls représentants bien connus de cette classe sont les machines vectorielles IBM 3090 VF. L'opérande se trouvant en mémoire est forcément un opérande source du traitement, le résultat étant rangé dans un registre du processeur.

Toutes ces classes de machines utilisent des pipelines de calcul du même genre (c'est à dire qu'ils réalisent le même type d'opération, acceptent en entrée un jeu de données à chaque cycle, et produisent un résultat par cycle - s'ils sont régulièrement approvisionnés - une fois le temps de latence écoulé).

Les machines mémoire à mémoire

Pour les machines mémoire à mémoire, un vecteur est accédé au moyen d'un *descripteur* qui indiquent d'une part l'adresse de base des éléments du vecteur, d'autre part le nombre d'éléments du vecteur. Sur Cyber-205 et ETA10, cette longueur est codée sur 16 bits, donc limitée à 64K éléments. Des requêtes d'accès aux vecteurs sources sont émises. Les éléments sont ensuite directement envoyés à l'entrée des pipelines. La sortie du pipeline (ou du dernier pipeline de la chaîne s'il y a chaînage) est alors directement renvoyée vers la mémoire (cf. fig. I.2.a).

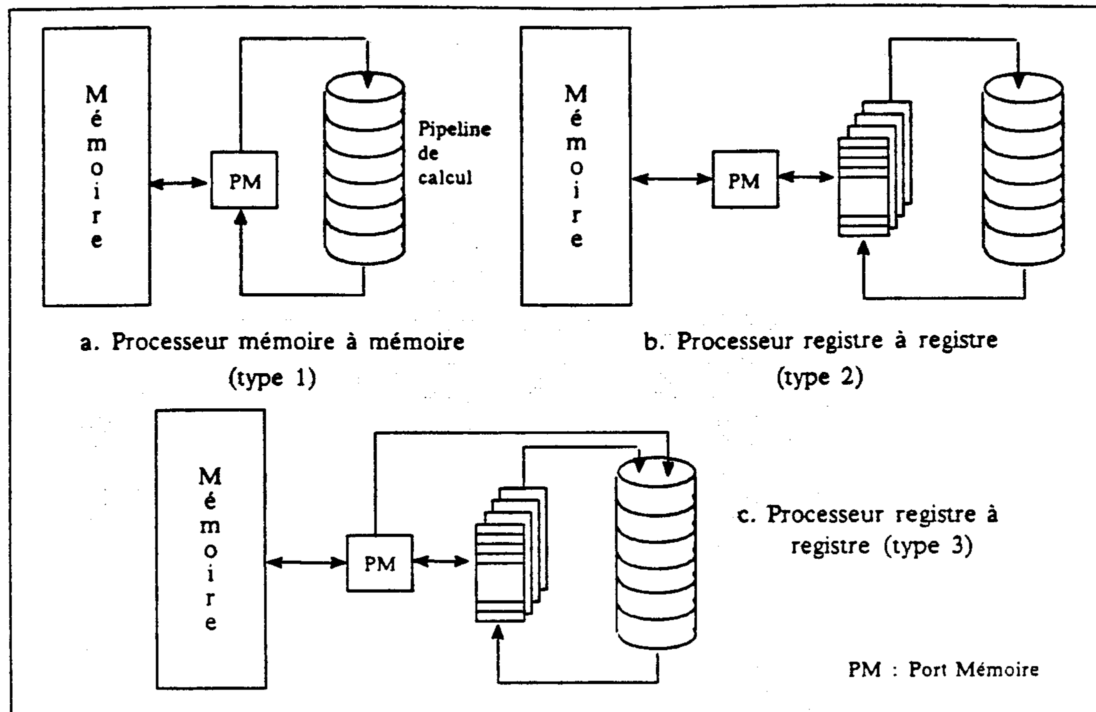


Figure I.2. -
Différents types de fonctionnement des processeurs vectoriels.
Note : le pipeline est représenté schématiquement. En fait, il peut résulter du chaînage de plusieurs pipelines physiques.

Les machines registre à registre

Pour les machines registre à registre (types 2 et 3), la longueur des opérandes vectoriels est spécifiée dans un registre du processeur. Toutes les opérations vectorielles (accès mémoires ou traitements à l'intérieur du processeur) sont réalisées sur le nombre d'éléments des vecteurs indiqué par ce registre (cf. fig. I.2.b et I.2.c). La valeur de ce registre doit généralement être inférieure au nombre d'éléments des registres vectoriels. Une exception notable est le processeur de l'Hitachi S-810 (cf. infra).

Les vecteurs traités dans une application scientifique typique comptent des milliers voire des millions de composantes. Les opérations de base des processeurs vectoriels ne prenant en compte que des vecteurs possédant un nombre relativement réduit d'éléments, les opérations vectorielles réelles sont découpées en une succession d'opérations de base (sur des vecteurs de taille limitée). En général, ce découpage (*strip-mining* en anglais) est laissé à la charge du compilateur. Seul parmi les supercalculateurs considérés ici, le processeur du Hitachi S-810 découpe lui-même (par un mécanisme matériel) les vecteurs à traiter.

Un ordonnanceur d'instructions ré-ordonne si possible le flux des instructions afin d'obtenir un parallélisme maximal à l'exécution. L'initiateur d'instructions a alors pour charge de déclencher les instructions dès que cela est possible (unité fonctionnelle libre, chemin de données entre les registres vectoriels et les unités fonctionnelles accessibles, données disponibles, ...). Parmi les machines qui nous intéressent ici, les instructions vectorielles ne sont pas ré-ordonnées. Par contre, les processeurs Cray ré-ordonnent à l'exécution les instructions scalaires en fonction de la disponibilité des unités, via un mécanisme matériel.

Le Cray-1 a introduit en 1976 la notion de chaînage de pipelines vectoriels qui permet de brancher la sortie d'un pipeline à l'entrée du pipeline suivant. De cette manière, un (ou plusieurs)

pipeline d'entrées / sorties peut être chaîné à l'entrée d'une chaîne de pipelines de calculs, la sortie de cette chaîne étant directement chaînée à un port mémoire qui effectue l'écriture du résultat. Le chaînage des pipelines est, en général, automatiquement détecté à l'exécution par le processeur et mis en place par l'initiateur d'instructions du processeur. Sur certaines machines, le chaînage des instructions est impossible (Cray-2 et IBM VF). Sur les Cyber-205 et ETA10, il est très réduit et non automatique. Une instruction spécifique indique que les deux instructions qui la suivent doivent être chaînées.

Les accès en mémoire (en lecture ou en écriture) sont réalisés par des unités fonctionnelles spécialisées du processeur, les ports mémoires. Alliée à la possibilité de chaînage des pipelines, la multiplication du nombre de ports mémoires d'un processeur autorise le maintien de hautes performances. Ainsi, sur un Cray Y-MP, chaque processeur possède quatre ports mémoires dont l'un est réservé exclusivement aux entrées/sorties. La réalisation de l'opération $\vec{A} = \vec{B}$ op. $\vec{C}$ où $\vec{A}$, $\vec{B}$ et $\vec{C}$ sont trois vecteurs résidant en mémoire et op. une opération vectorielle résultant d'un pipeline ou du chaînage de plusieurs pipelines est beaucoup plus rapide que sur une machine ne possédant qu'un ou deux ports mémoires (comme le Cray-2 ou le Fujitsu VP-200). Dans ce cas, deux ports mémoires réalisent le chargement des vecteurs $\vec{B}$ et $\vec{C}$. Leurs éléments sont directement envoyés sur l'entrée du pipeline. Sa sortie est envoyée au troisième port mémoire qui écrit directement le résultat en mémoire. Le Hitachi S-810 possède 4 ports mémoires autorisant une exécution performante d'opérations du genre $\vec{A} = (\vec{B} + \vec{C}) * \vec{D}$. Comme nous le reverrons plus loin, la simple multiplication du nombre de ports mémoires n'apporte pas forcément une augmentation des perfomances directement proportionnelle à ce nombre. Ainsi, l'expérience et des simulations montrent que la dégradation des performances due aux conflits d'accès aux bancs mémoires est supérieure à 30 % quand plus de trois ports mémoires sont actifs simultanément [TANG & al. 89].

1.2.2. Processeur vectoriel pipeline

Parmi les supercalculateurs, nous nous intéressons maintenant plus particulièrement aux machines registre à registre Cray, les supercalculateurs Japonais Fujitsu, Nec, Hitachi et l'IBM 3090 VF.

Malgré les spécificités propres à chacune, un schéma général de l'architecture de ces machines peut être établi. Toutes les machines considérées ici possèdent une architecture très similaire, héritée du précurseur en la matière, le Cray-1.

La figure I.3. présente un aperçu schématique de ces architectures. En aucun cas, cette figure ne se veut la réplique d'un processeur précis. C'est simplement un modèle montrant l'organisation d'un processeur vectoriel pipeline typique.

On distingue essentiellement deux parties, la mémoire et le processeur.

Le processeur est principalement composé de registres et d'unités fonctionnelles. Les registres se répartissent en registres de données (les registres vectoriels, scalaires et d'adresses), registres de contrôle des opérations vectorielles (VLG et masque) et registres utilisés pour contrôler l'activité du processeur (tampons d'instructions et divers autres registres non représentés - compteur ordinal, ...). Les unités fonctionnelles se composent d'unités de calculs (les pipelines vectoriels, scalaires et d'adresses), d'unités d'entrées/sorties (les ports mémoires) et d'unités de contrôle (initiateur et répartiteur d'instructions).

Les registres vectoriels sont en général configurés de manière fixe (il y en a un nombre donné, dont la taille est fixée par le matériel). Les Fujitsu VP ont la possibilité de faire varier le nombre et la taille de leurs registres vectoriels selon la valeur d'un registre de contrôle qui indique la taille des

registres (à la base, il y a 256 registres de 32 éléments dont le nombre peut se réduire à 128, 64, 32, 16 et 8 registres comportant chacun un nombre d'éléments en rapport - 64, 128, 256, 512 ou 1024).

Ces processeurs possèdent tous un registre VLG (VL sur Cray et Fujitsu, VCT sur IBM) donnant le nombre d'éléments sur lesquels les instructions vectorielles sont réalisées. Un ou plusieurs registres de masque (VM sur Cray, VMR sur IBM, MR0..255 sur Fujitsu) autorisent le contrôle d'instructions par un vecteur de bits.

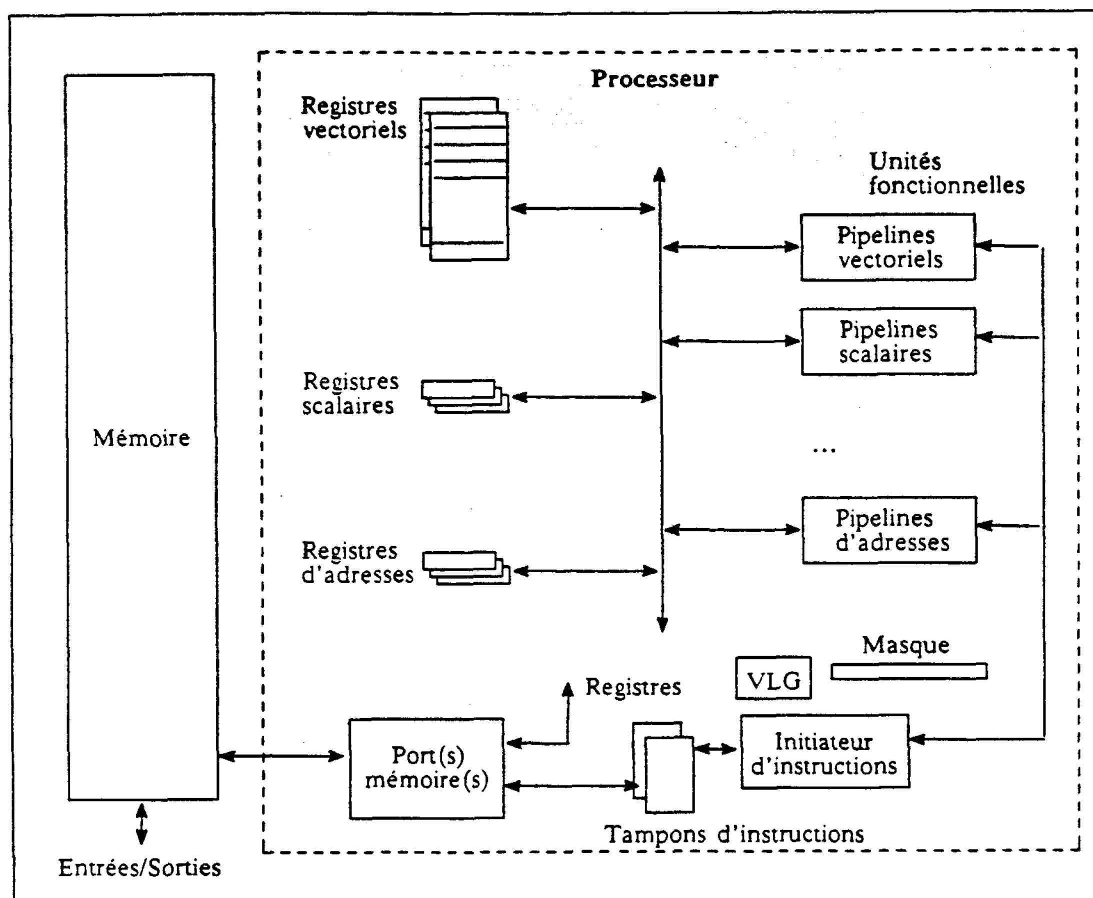


Figure I.3. -
Description schématique d'un processeur vectoriel pipeline

1.2.3. La mémoire des supercalculateurs vectoriels pipelines

Le spectre des capacités des mémoires centrales s'étend de quelques méga-octets à 256 méga-mots. Plus la capacité de la mémoire est étendue, plus le nombre de cycles-CPUs nécessaires à l'accomplissement d'un accès mémoire augmente.

Les CPUs sont assemblés à partir de circuits intégrés utilisant des technologies très rapides (technologie LSI à 70 ps pour le CPU du Nec SX-3 par exemple). Les mémoires des super-calculateurs tendent à devenir de capacités de plus en plus grandes (256 millions de mots de 64 bits pour le Cray-2). Cette taille entraîne l'utilisation d'une technologie moins rapide que celles des CPUs (technologie LSI à 1.6 ns pour la mémoire du Nec SX-3), essentiellement pour des contraintes économiques. Par ailleurs, la grande capacité des mémoires entraîne des temps plus élevés de décodage d'adresses et de contrôle des circuits de la mémoire.

Aussi, le temps de cycle TC d'un CPU (TC : environ 5 ns à l'heure actuelle) est très inférieur au temps de cycle TM de la mémoire centrale des super-calculateurs vectoriels (TM : de l'ordre de 30 à 40 ns pour les plus rapides). Nous avons donc le temps d'occupation d'un banc (*Busy Bank Time*) :

$$\tau = \frac{TM}{TC}$$

supérieur à un (τ supérieur ou égal à quatre pour les machines considérées ici). Aussi, lorsqu'un CPU déclenche un accès mémoire, il ne récupère le résultat que τ cycles plus tard. Si la mémoire est monolithique (d'un seul bloc) et n'autorise qu'un accès à la fois, le CPU ne peut effectuer qu'un seul accès mémoire tous les τ cycles. Face à ce problème, diverses solutions ont été envisagées afin d'augmenter le débit (ou *bande passante*) de la mémoire. Deux axes ont été étudiés, les améliorations matérielles et les améliorations logiques. Les solutions matérielles s'articulent autour des deux axes suivants :

1) accéder à plusieurs mots mémoires à la fois en augmentant la largeur des bus de données. Au lieu d'accéder un mot mémoire, plusieurs sont accédés à la fois (un *super-mot*);

2) autoriser plusieurs accès en parallèle à la mémoire. Une solution à cette technique repose sur le découpage de la mémoire en *bancs*.

Seules la mémoire de l'IBM 3090 VF et la mémoire partagée de l'ETA10 ne sont pas composées de bancs (la mémoire locale des processeurs de l'ETA10 est cependant composée de bancs).

Les améliorations logiques s'intéressent aux méthodes de rangement des données dans des mémoires composées de bancs. Dans certains cas, une analyse des accès réalisés aux éléments d'un vecteur dans un programme autorise une optimisation de ce rangement pour limiter, voire supprimer, les conflits d'accès.

L'accès à des super-mots

Cette solution consiste à accéder plusieurs mots contigus en mémoire en un seul cycle et à tous les envoyer en parallèle au processeur. Par exemple, le Cyber-205 effectue des accès à des super-mots de 512 bits (8 x 64 bits) lors des accès vectoriels. L'ETA10 a repris les accès par super-mots pour les transferts entre les processeurs et leurs mémoires locales.

L'accès à un vecteur contigu est donc une suite d'accès à des super-mots consécutifs en mémoire. Un accès à un vecteur avec pas se fait par sélection des éléments du vecteur parmi les super-mots. Au mieux (pas égal à deux), les performances des accès mémoires chutent de 50 % (il y a deux fois plus d'accès mémoire que dans le cas d'un accès contigu). Au pire, il n'y a qu'une seule composante du vecteur par super-mot (il y a N accès mémoire pour accéder un vecteur de N composantes). L'accès aux vecteurs dispersés fonctionne lui aussi par sélection des composantes utiles dans les super-mots. Les performances sont donc mauvaises pour ce type d'accès. Ceci constitue l'un des défauts majeurs du Cyber-205.

La technique d'accès à des super-mots a été reprise par IBM pour son co-processeur vectoriel de l'IBM 3090. Les échanges entre le co-processeur et sa mémoire locale sont réalisés par super-mots de 1024 bits (16 x 64 bits).

L'utilisation de bancs mémoire

Cette solution consiste tout d'abord à découper la mémoire, d'une capacité de M mots, en plusieurs morceaux (bancs), disons β, de même taille M / β. Les adresses mémoire sont alors réparties sur les bancs. Une méthode de répartition consiste à placer les adresses de manière à ce que deux mots consécutifs en mémoire se trouvent dans deux bancs successifs (le mot d'adresse @

se trouve dans le banc @ mod. β). Cette méthode de répartition des adresses parmi les bancs sera dénommée ici MODULO, en accord avec [RAGHAVAN & al. 90]. Elle s'implante facilement : en général, le nombre de bancs est une puissance de deux ($\beta = 2^j$). Aussi, le banc correspondant à une adresse est obtenue par décodage des j bits de poids faibles (ou, plus généralement, de j bits parmi les k bits d'une adresse).

Utilisant cette technique, le CPU peut émettre une requête à chaque cycle vers la mémoire, tant qu'une requête ne touche pas un banc en cours d'activité. Si la mémoire est découpée en β bancs, on peut ainsi espérer réaliser un maximum de β accès concurrents à la mémoire. Si $\beta \geq \tau$, on peut obtenir un accès mémoire par cycle CPU en moyenne.

Cette technique fonctionne très bien si les conditions suivantes sont réunies :

il y a un seul CPU en jeu;

le CPU ne dispose que d'un seul port mémoire;

le CPU accède des vecteurs contigus, dans des bancs distincts.

Pour chacune de ces conditions, les conséquences de son non-respect sont maintenant examinées.

Si plusieurs CPUs n'ayant qu'un seul port mémoire se partagent la mémoire, les bancs mémoires n'autorisant qu'un seul accès à la fois, un CPU risque de vouloir accéder une donnée se trouvant dans un banc en cours d'accès par un autre CPU. Ceci entraîne un conflit d'accès, donc une attente du deuxième CPU. Une dégradation du débit de la mémoire est donc observée.

Dans un système monoprocesseur où le CPU possède plusieurs ports d'accès mémoire (Cray X-MP/1, Nec SX-2, Fujitsu VP), le même problème peut survenir quand un port tente d'accéder une donnée se trouvant dans un banc en cours d'accès par un autre port. Il y a alors conflit d'accès au banc, d'où attente du port, d'où dégradation du débit de la mémoire du point de vue du deuxième port.

Dès que le CPU tente d'accéder à des vecteurs dont les éléments ne sont pas contigus en mémoire (c'est à dire des accès à des composantes régulièrement espacées ou dispersées aléatoirement – rassemblement/éclatement(1)), le risque de conflit d'accès aux bancs augmente.

Augmentation du nombre de bancs mémoires

Une première solution consiste à multiplier le nombre de bancs mémoires ($\beta \gg \tau$) pour réduire la probabilité qu'un accès mémoire entraîne un conflit. La solution idéale est alors d'avoir autant de bancs que de mots en mémoire. Cette solution n'est pas une solution très viable, les problèmes de connectiques augmentant avec le nombre de bancs.

Une autre approche consiste à utiliser un nombre de bancs premier. Ceci entraîne la réduction statistique du nombre de conflits de banc, donc une augmentation des performances de la mémoire [LAWRIE & al. 82]. Cette méthode donne accès en parallèle aux éléments d'une ligne, d'une colonne ou d'une diagonale d'une matrice, si ses dimensions ne sont pas multiples du nombre de bancs. Cependant, le réseau d'interconnections n'est plus symétrique et le calcul de numéro du banc contenant une adresse est alors assez complexe (modulo avec un nombre premier plutôt que modulo une puissance de 2 si la mémoire est divisée en 2^N bancs mémoires).

Structuration de la mémoire

Une deuxième solution (qui s'allie à la précédente) consiste à utiliser plusieurs niveaux de

(1) – Opérations connues sous l'anglicisme gather/scatter

découpage de la mémoire. La mémoire est tout d'abord découpée en *sections*(2), chacune étant elle-même décomposée en bancs. Cette technique est notamment implantée sur les calculateurs Cray depuis le modèle X-MP (cf. table I.1.). Un accès à un mot mémoire se "décompose" alors en un accès à la section contenant le banc visé et en un accès au banc lui-même. L'intérêt de cette technique réside essentiellement dans la réduction du nombre de chemins d'accès à la mémoire.

Pour sa part, la mémoire du Cray Y-MP se décompose en un niveau supplémentaire : chaque section est composée de sous-sections, elles-mêmes divisées en bancs.

Le gros défaut de cette décomposition de la mémoire en sections est l'apparition de conflits supplémentaires (inexistants dans un découpage simple en bancs). Cependant, ces conflits, dits *conflits de section*, sont peu pénalisants. Un accès bloqué au niveau d'une section est retenté au cycle suivant. Par contre, un accès bloqué au niveau du banc le reste éventuellement encore 4 cycles.

	Nombre de CPUs	Nombre de sections	Nombre de bancs par section	Capacité d'un banc
Cray X-MP/416(3)	4	4	16	0,25 M-mots
Cray Y-MP/432(3)	4	4 x 4	64 16 par sous-section	0,125 M-mots
Cray 2	4	4	32	2 M-mots

Table I.1. -
Quelques exemples de decoupage de la mémoire en sections et bancs

Répartition des adresses dans les bancs

Une autre solution consiste à répartir différemment les adresses (logiques) parmi les bancs (adresses physiques). Le but alors recherché est la répartition uniforme parmi les bancs des composantes des vecteurs, lors d'accès divers. Ces accès peuvent être avec pas (types d'accès qui recouvrent les accès aux lignes, colonnes, diagonales d'une matrice), concerner des sous-matrices d'une matrice, ...

L'entrelacement aléatoire des adresses sur les bancs est une technique issue de celle des "schémas de stockages décalés" (*skewed storage schemes*) introduits en 1968 pour ILLIAV IV [KUCK 68], et développés ensuite par de nombreux auteurs, dont [BUDNICK & al. 71] et [SHAPIRO 78] pour ne citer qu'eux et sans prétendre à l'exhaustivité. Ces schémas de stockages consistent à ranger les adresses en décalant les lignes les unes par rapport aux autres (cf. fig. 1.4.).

(2) - encore dénommées lignes ou quadrants, selon les constructeurs et les machines

(3) - La notation utilisée par Cray pour ses calculateurs X-MP et Y-MP indique le nombre de processeurs (ici 4), et la capacité de la mémoire en mots (ici 16 et 32 Méga-mots)

$Banc_0$	$Banc_1$	$Banc_2$	$Banc_3$
0	1	2	3
7	4	5	6
10	11	8	9
...	...	...	...

Figure I.4. –
Répartition des adresses parmi les bancs en utilisant la technique des schéma de stockages décalés d'une valeur de 1.

De manière générale, dans une mémoire comprenant β bancs et utilisant un schéma de rangement décalé d'une valeur κ, une adresse @ est localisée dans le banc $@ + \kappa \lfloor @ / \beta \rfloor \bmod \beta$. [HARPER & al. 87] ont montré qu'un schéma de rangement décalé de un est statistiquement meilleur qu'un simple entrelacement des adresses, sans décalage.

La répartition aléatoire des adresses est plus compliquée. Partant d'une adresse logique, l'unté de décodage d'adresses commence par calculer l'adresse physique (le numéro du banc) pseudo-aléatoirement. Cette transformation est réalisée selon une opération $\phi = M \cdot \lambda$, où ϕ et λ sont des vecteurs colonnes dont les composantes sont respectivement les bits des adresses physique et logique, et M est une matrice de dimensions $\beta \times \nu$, ν étant le nombre de bits d'une adresse. Cette multiplication de matrices est en fait transformée en une opération similaire où les multiplications sont remplacées par des et-logiques et les additions par des ou-logiques. L'utilisation de ces opérations logiques bit à bit minimise le temps de calcul des adresses physiques. L'accès est ensuite réalisé dans ce banc. La génération doit respecter un certain nombre de contraintes :

- pour une adresse logique, c'est toujours le même banc qui doit être calculé par le générateur d'adresses;
- deux adresses logiques différentes doivent posséder deux adresses physiques différentes;
- toute adresse physique doit correspondre à une adresse logique.

Pour résumer, le générateur d'adresses doit garantir une bijection entre l'ensemble des adresses logiques et celui des adresses physiques. Cette technique de répartition des adresses évite les conflits permanents des mémoires à β bancs entrelacés lorsque l'on accède un vecteur avec un pas de β ou β / 2 par exemple.

Cette technique est discutée dans [WEISS 89] et [RAGHAVAN & al. 90]. Elle est dénommée par ses derniers **LINEAIRE**. Elle est déjà présente dans des travaux antérieurs [FRAILONG & al. 85] où les auteurs utilisent des portes ou-exclusifs pour réaliser la conversion d'une adresse logique en une adresse physique. Elle est implantée sur la machine Cydra développée par Cydrome Inc. et décrite dans [RAU 89] de même que dans la machine RP3 d'IBM [PFISTER & al. 87]. Ainsi, les schémas de stockages décalés sont une technique de répartition des adresses parmi les bancs qui "mélangent" beaucoup moins les adresses que les méthodes utilisées sur le Cedar par exemple. Cette évolution est due à un désir d'obtenir des accès non conflictuels à des éléments de vecteurs dispersés plus ou moins régulièrement parmi ses composantes (et non plus seulement pouvoir réaliser des accès non conflictuels uniquement aux éléments des lignes, colonnes, diagonales et quelques autres structures remarquables des matrices).

[RAGHAVAN & al. 90] rapporte une étude précise des mécanismes de répartition aléatoire des adresses parmi les bancs mémoires. Ils comparent les deux méthodes **MODULO** et **LINEAIRE**,

montrant que cette dernière, bien que très intéressante dans certains cas, peut se montrer moins performante (car elle produit plus de conflits) dans d'autres. Le problème essentiel avec LINEAIRE, vient de ce que des conflits d'accès apparaissent aléatoirement, dans des cas où l'utilisation de la technique MODULO n'entraînerait aucun conflit. Pour pallier les défauts de chacune, les auteurs introduisent une méthode de répartition des adresses originales, tentant de combiner les avantages des deux solutions tout en évitant leurs défauts. Ainsi, les conflits permanents de la technique MODULO sont supprimés et les conflits aléatoires de LINEAIRE le sont également.

Différents types d'accès vectoriels

Dans cette section, nous exposons le déroulement des accès en mémoire aux composantes des vecteurs sur machines vectorielles pipe-lines. Sur une machine du type Cray Y-MP (c'est à dire, les Cray, Fujitsu, Nec et IBM), un vecteur est vu par le port mémoire comme une séquence de scalaires, ses composantes. Le port mémoire génère la séquence des adresses des composantes qu'il émet les unes après les autres vers la mémoire, au rythme maximum d'une requête par cycle(4). Elle génère la première requête. La requête est soit possible, soit impossible. Si la requête est possible, le port émet une requête visant l'adresse suivante de la séquence d'adresses au cycle CPU suivant. Si suite à un conflit, la requête est impossible, le port ré-émet l'adresse au cycle suivant. Tant que l'accès ne peut être satisfait, le port émet la même requête. Notons qu'une requête devient toujours possible au bout d'un temps fini, ceci étant garanti par le matériel.

1.2.4. Particularités des multi-processeurs vectoriels pipelines

Avec les Cray X-MP/2 (bi-processeurs) sont apparus les supercalculateurs vectoriels pipelines multi-processeurs. On distingue alors différents modes de fonctionnements qui sont brièvement décrits ici.

Le mode multi-tâches

Le mode multi-tâches où les tâches composant une application s'exécutent sur des processeurs différents. Pendant l'exécution d'une application, un certain nombre de processeurs lui sont alors entièrement dédiés. Les processeurs sont alloués à la tâche par le système d'exploitation. A une application peut être alloué 1, 2 ou 4 processeurs (sur Cray X-MP où le nombre de processeurs est limité à 4). Cette allocation des processeurs est statique, durant tout le temps d'exécution de l'application.

Parallélisation d'une application

Les deux modes de fonctionnement qui suivent concernent la parallélisation d'une même application, c'est à dire son découpage en plusieurs tâches.

Le mode macro-tâches

Dans le mode macro-tâches, un même programme est découpé en plusieurs tâches de tailles importantes. Typiquement, le temps d'exécution de chacune s'étend sur plusieurs millions de cycles processeur. Dans ce type de fonctionnement, le programmeur place des appels à des routines de création de tâches et de communications/synchronisations entre les tâches. On obtient ainsi un parallélisme d'exécution à gros grain.

Dans ce mode de fonctionnement, les processeurs sont alloués "à la demande" à la tâche. Ils sont alors partagés entre les différentes applications s'exécutant à un moment donné sur le Cray. Ceci entraine donc d'importantes commutations de contextes. Notons que sur des machines où le nombre de processeurs est élevé (cf. Cm* [GEHRINGER & al. 82]), une autre méthode consiste à allouer

(4) - Le rythme est réduit en cas de conflit (cf. infra)

l'ensemble des processeurs nécessaires à l'exécution d'une application durant toute sa durée. On évite ainsi les commutations de contextes entre des tâches d'applications différentes.

Le mode micro-tâches

Dans le mode micro-tâches, le découpage en tâches est beaucoup plus fin et se joue au niveau des instructions du langage de haut niveau. Le temps d'exécution est alors typiquement de quelques milliers de cycles processeurs pour chacune des micro-tâches. Cette technique consiste essentiellement à paralléliser les boucles des programmes. Le grain du parallélisme ainsi obtenu est alors assez fin. Une exécution du type "flux de données" peut alors être projetée sur l'architecture du Cray X-MP [REINHARDT 85].

Les techniques les plus classiques consistent pour le programmeur à placer explicitement des appels à des routines gérant les tâches et leurs communications. Une approche récente développée par Cray sur son modèle Y-MP consiste à réaliser ce découpage en suivant des directives de compilation indiquant les boucles qui peuvent être découpées [NAGEL 90]. En mode micro-tâches, les communications entre les processeurs utilisent essentiellement les registres partagés afin d'obtenir des vitesses d'échanges élevées, compatibles avec le grain du découpage.

Il faut noter qu'une commutation de tâches est peu coûteuse dans le cas du macro-tâches (temps de changement de contexte par rapport au temps d'exécution de la tâche), alors qu'elle n'est pas envisageable dans le cas du micro-tâches.

Communications inter-processeurs

Des mécanismes matériels ont dus être mis en place pour permettre les communications et la synchronisation entre les processeurs.

Sur Cray X-MP et Y-MP, elles sont réalisées via des "clusters". Chacun contient 8 registres d'adresses (SB - 24 bits sur X-MP, 32 sur X-MP EA et Y-MP), 8 registres scalaires (ST - 64 bits chacun) et 32 registres sémaphores (SM). Un cluster peut être partagé par zéro, un ou plusieurs processeurs pour les communications et les synchronisations. L'allocation des clusters aux processeurs est réalisée sous le contrôle du système d'exploitation. Il y a trois clusters sur X-MP/2, cinq sur X-MP/4. Les éventuelles étreintes mortelles sont détectées par le matériel. Dans un environnement multi-processeurs, les registres SB et ST sont utilisés pour le transfert d'adresses et de scalaires entre les CPUs et les registres SM sont utilisées au contrôle entre les CPUs. Notons que les registres spécialisés SB et ST autorisent des échanges rapides de données. Des échanges peuvent également être réalisés en passant par la mémoire centrale, la vitesse en étant alors limitée par la bande passante de la mémoire. Par contre, la taille des données transférées n'est plus limitée de la même manière.

Sur le Cray-2, le seul outil de synchronisation est un jeu de huit sémaphores. Les échanges de données passent alors nécessairement par la mémoire centrale.

ETA10 utilise une mémoire spécialisée d'un million de mots pour les communications et les synchronisations entre processeurs. Les processeurs accèdent cette mémoire un mot à la fois, et non par super-mots.

Les communications rapides entre les processeurs du NEC SX-3 passent par des registres spécialisés.

Etant donnée la richesse du mécanisme des X-MP et Y-MP par rapport au Cray-2, il semble clair que les premiers effectueront leurs communications inter-processeurs plus rapidement que le second. Des comparaisons entre X-MP et Cray-2 attestent de cette remarque [SIMMONS & al. 90].

La mémoire des multi-processeurs vectoriels Cray X-MP

Nous décrivons ici l'organisation de la mémoire du Cray X-MP. La mémoire est découpée en 64 bancs regroupés en 8 sections (cf. fig. I.5.). Chacun des 4 processeurs possède 4 ports

mémoires, dont l'un est dédié aux opérations d'entrées/sorties. Pour les 3 ports accédant la mémoire centrale, il ne peut y avoir que 2 opérations de lecture en même temps. Un accès à un scalaire ne peut avoir lieu en même temps qu'un accès vectoriel. Lors d'un accès à une donnée (un banc), trois types de conflits peuvent survenir. On distingue :

- les conflits de banc occupé. Un conflit de ce type apparait lorsqu'une requête vise un banc en cours d'activité. Le requête est alors rejetée par le banc. Le port mémoire ré-émet la requête au cycle suivant;
- les conflits de simultanéité. Un conflit de ce type apparait lorsque deux requêtes visent un même banc. Ces deux requêtes proviennent de deux processeurs différents. Le banc sélectionne l'un des deux accès. L'autre requête est rejetée et sera ré-émise au cycle suivant par le port mémoire;
- les conflits de section. Un conflit de ce type apparaît lorsque deux requêtes visent une même ligne. Les deux requêtes sont émises par deux ports d'un même processeur. Le circuit d'interconnection sélectionne l'un des deux accès qui est envoyé vers le banc destinataire de la requête. L'autre requête est rejetée et sera ré-émise au cycle suivant.

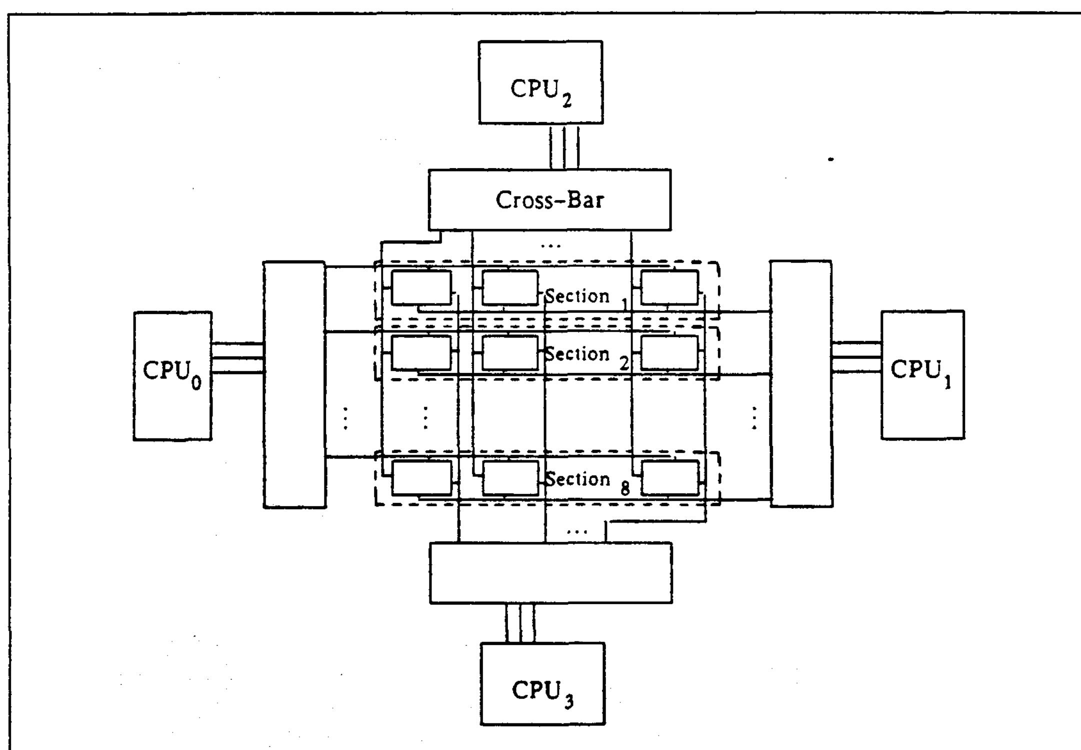


Figure I.5. -
Liaisons processeurs-mémoires et organisation
de la mémoire du Cray X-MP/4

Une autre approche aux problèmes de conflits mémoires : le Cray-2

Dès l'apparition du Cray-2, il a été reconnu que l'organisation de sa mémoire était inadaptée [SCHONAUER 87, p. 305-324] (cf. fig. I.6.). Sa grande capacité aurait pu être un avantage décisif. Malheureusement, un temps d'accès mémoire très long, des processeurs ne possédant qu'un seul port mémoire et le fait qu'un processeur n'a accès qu'à un seul quadrant (fixé à chaque cycle processeur) font que globalement, son organisation mémoire est faible. A un cycle donné, le processeur i a accès au quadrant j, au cycle suivant, au quadrant (j+1) mod. 4, ... Pour pallier cette

contrainte, les concepteurs du Cray-2 ont implanté un accès désordonné aux composantes des vecteurs en mémoire. Aussi, les registres vectoriels peuvent être chargés de manière aléatoire à partir de la mémoire. Du fait, le chaînage des pipelines est impossible (non seulement, le chaînage d'un pipeline d'entrées/sorties avec un pipeline de calcul, mais également entre deux pipelines de calcul(5)).

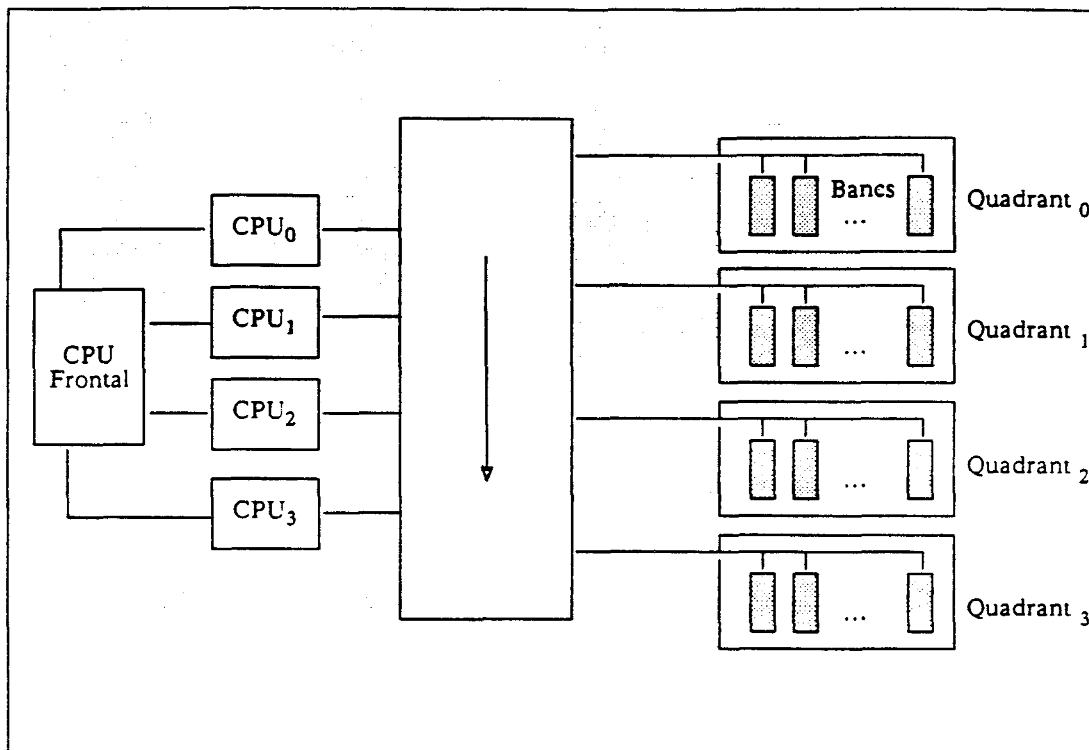


Figure I.6. -
Liaisons processeurs-mémoires et organisation de la mémoire
sur une machine du type Cray 2

Les machines Cray X-MP, Y-MP et 2 regroupent leurs bancs par sections (et sous-sections en ce qui concerne le Y-MP). A un cycle donné, les ports mémoires des Cray X-MP et Y-MP peuvent accéder n'importe quelle adresse en mémoire. Au contraire, un port mémoire de Cray-2 ne peut accéder une adresse que dans un quart de la mémoire (une quadrant) à chaque cycle, la section accessible changeant à chaque cycle processeur.

1.3. Autres solutions pour les accès vectoriels

Nous avons précédemment vu les méthodes couramment employées dans les super-calculateurs vectoriels "commerciaux". A côté de celles-ci, des méthodes sont en cours d'expérimentation. Ainsi, la représentation des matrices creuses utilisée dans le projet de machine ESP (Edinburgh Sparse Processor) en cours à l'université d'Edinburgh implique un adressage et un traitement particuliers pour ce type de matrices. ESP est présentée dans [IBBETT & al. 1989].

La machine ESP est destinée à exécuter des algorithmes de programmation linéaire dans lesquelles la densité des matrices (nombre d'éléments non nuls) varie de 1 à 20 %. Les matrices creuses sont

(5) - Ce qui, il faut le noter, n'est pas une conséquence directe du mécanisme d'entrées/sorties. Les concepteurs auraient pu définir le Cray-2 en n'autorisant pas le chainage des ports avec les unités fonctionnelles de calcul, tout en conservant les possibilités de chainage des unités fonctionnelles de calcul entre elles (cf. partie III).

représentées par deux listes associées, la liste des éléments significatifs (non nuls) et la liste des indices de ces éléments dans la matrice. Les unités fonctionnelles d'ESP sont prévues pour réaliser des calculs sur des matrices représentées de cette manière, et générer la matrice résultante sous cette forme également. A côté de cette technique optimisée pour le stockage des matrices creuses, ESP représente les matrices non creuses sous leur forme classique.

2. Les langages vectoriels

Nous présentons dans cette section la couche vectorielle supérieure, celle des langages vectoriels de haut niveau. Nous exposons tout d'abord le principe de *conservation de parallélisme*, lequel illustre ce qui semble être l'idéal en matière de projection d'algorithmes parallèles sur des architectures. Par la suite, nous présentons les trois techniques classiques de programmation vectorielle, en liaison avec ce principe de conservation.

2.1. Le principe de conservation du parallélisme

[HOCKNEY & al. 88 p. 376] indiquent quatre niveaux de parallélisme :

- parallélisme inhérent à l'algorithme (en dehors de tout langage de programmation) ($\mathcal{A}$),
- parallélisme atteint à l'implantation de l'algorithme dans un langage de haut niveau ($\mathcal{I}$),
- parallélisme atteint à la traduction de cet algorithme dans le langage machine ($\mathcal{L}$),
- parallélisme effectif à l'exécution de cet algorithme sur une machine donnée ($\mathcal{E}$).

Ils indiquent alors que dans une situation normale, le taux de parallélisme doit décroître dans le même ordre. C'est le principe de *conservation du parallélisme*. C'est à dire que nous avons :

$$\mathcal{A} \geqslant \mathcal{I} \geqslant \mathcal{L} \geqslant \mathcal{E}$$

Les langages de programmation qui disposent de constructions parallèles (structures de données ou structures de contrôle) autorisent le programmeur à exprimer ses algorithmes sans s'occuper des contraintes liées à l'architecture cible. Notons que la remarque bien connue selon laquelle "la transformation d'un algorithme parallèle en algorithme scalaire est beaucoup plus simple que la transformation inverse" montre bien que ce principe de conservation est une "bonne propriété" des systèmes de développements d'algorithmes parallèles.

Cependant, la nécessité d'autoriser la ré-utilisation des programmes déjà écrits en langages scalaires entraîne l'existence des vectoriseurs. L'utilisation de tels outils est due à la violation du principe de conservation du parallélisme (dans ce cas, on a $\mathcal{A} \geqslant \mathcal{I} \leqslant \mathcal{L} \geqslant \mathcal{E}$). Nombre de langages utilisés pour la programmation de machines parallèles sont tels que $\mathcal{A} \geqslant \mathcal{I} = \mathcal{L} = \mathcal{E}$. Ces langages ne sont pas portables. Ils sont efficaces, des structures non implantables inefficacement sur la cible n'existant pas dans ces langages.

En résumé, les langages vectoriels de la première catégorie (scalaires + vectoriseurs, langages de détection du parallélisme) sont tels que :

$$\mathcal{A} \geqslant \mathcal{I} \leqslant \mathcal{L} \geqslant \mathcal{E}$$

ceux de la deuxième catégorie (langages d'expression du parallélisme des machines) tels que :

$\mathcal{A} > \mathcal{S} = \mathcal{L} = \mathcal{B}$

et ceux de la troisième catégorie (langages d'expression du parallélisme des algorithmes) tels que :

$\mathcal{A} > \mathcal{S} > \mathcal{L} > \mathcal{B}$

2.2. Les différents types de langages de programmation vectorielle

Il y a trois techniques de programmation vectorielle qui vont faire chacune l'objet d'une description et d'une critique dans cette section. Un type de langage est associé à chacune de ces techniques.

La première technique recouvre les langages de détection du parallélisme, la deuxième les langages d'expression du parallélisme des machines et la troisième les langages d'expression du parallélisme des algorithmes.

2.2.1. Langage scalaire + vectoriseur

La première technique consiste à utiliser un langage scalaire tout à fait classique. Un *vectoriseur* transforme alors le source scalaire en un source "vectoriel". Cette transformation agit de différentes manières selon les vectoriseurs. Il peut s'agir d'inclusion de directives de compilation, regroupement d'instructions dans des boucles vectorielles, génération de "belles boucles"(6), ... Les langages scalaires sources communément utilisés sont Fortran 77 et C. Une utilisation de Pascal pour le Cray-1 peut également être notée [MADHAVJI & al. 82]. Le langage cible est soit un véritable langage vectoriel (Fortran 90 pour Parafrase-2 par exemple), soit un langage scalaire (Fortran 77) généré de manière à ce que les instructions vectorielles puissent être facilement retrouvées par la suite, lors de la génération de code. Une boucle (Do-Loop en Fortran) correspond à une instruction vectorielle.

Le grand intérêt de cette technique est l'utilisation d'un langage scalaire classique. De cette manière, les applications déjà développées peuvent être ré-utilisées, au prix d'une vectorisation.

Des éléments de vectorisation seront trouvés dans [PADUA & al. 86] et [ALLEN & al. 87]. Nous citerons les vectoriseurs Parafrase [KUCK 77] et Parafrase-2 [POLYCHRONOPOULOS & al. 90], PFC [ALLEN 82], VAST [BRODE 81], le vectoriseur interactif EAVE [BOSE 88] et VATIL [LICHNEWSKY & al. 85]. Une comparaison entre vectoriseurs sera trouvée dans [CALLAHAN & al. 88]. Plus généralement, les travaux concernant les vectoriseurs existent en très grandes quantités. Par ailleurs, depuis quelques années, suite à l'évolution des architectures, des paralléliseurs sont généralement associés aux vectoriseurs.

2.2.2. Langage vectoriel dédié à une cible

La deuxième technique consiste à utiliser un langage vectoriel (généralement une extension d'un langage scalaire classique comme Fortran ou C) proche de l'architecture sous-jacente. C'est à dire que les caractéristiques de la machine qui exécutera le programme sont directement prises en compte dans l'expression de l'algorithme. Par exemple, la taille des tableaux est égale au nombre de processeurs élémentaires d'une machine tableau ou à la taille des registres vectoriels. Ce type de

(6) - C'est à dire des boucles exprimant sous une forme itérative des instructions vectorielles, aisément identifiables par le compilateur.

langage se rapproche d'un "assembleur de haut niveau". Les programmes sont tout aussi non portables qu'en les programmant directement en assembleur.

Nous citerons Glypnir [LAWRIE & al. 75] et Fortran CFD [STEVENS 75] pour ILLIAC IV et Fortran Cyber FTN-200 [CDC 86] et une implantation de C sur Cray-2 [GISSELQUIST 86].

2.2.3. Langage autorisant l'expression d'algorithmes vectoriels

La troisième technique consiste à utiliser un langage vectoriel pur. Les algorithmes sont alors écrits vectoriellement, sans se soucier de l'architecture sous-jacente. A charge pour le compilateur de générer du code optimal pour la machine exécutant le programme. Depuis APL [FALKOFF & al. 73], quelques langages ont été développés pour autoriser l'expression d'algorithmes vectoriels. Nous citerons la définition très avant-gardiste en 1975 de Vectran chez IBM [PAUL & al. 78], d'Actus [PERROTT 79], de Hellena [JEGOU 87] et de EVA [DEKEYSER & al. 90c]. Par ailleurs, on peut noter l'utilisation du langage Ada pour la définition de paquetages de traitements vectoriels [VOLKSEN & al. 89].

Beaucoup plus important de part son utilisation effective, nous citerons pour terminer Fortran 90 (ex. Fortran 8x) [METCALF & al. 90] et le Fortran CFT (Cray Fortran), langage utilisé pour la programmation des calculateurs Cray. Ceux-ci sont difficiles à classer dans l'une des trois catégories précédentes. En effet, les boucles scalaires sont étudiées pour être vectorisées. Par ailleurs, des instructions vectorielles peuvent être écrites : a = b + c, où a, b et c sont des tableaux, est une instruction en Fortran CFT.

2.2.4. Discussion

Il faut bien distinguer les deux phases de *vectorisation* et de *génération*. Pour simplifier l'exposé, nous définissons dans ce document la *vectorisation* comme une analyse d'un programme qui détecte ce qui est de nature vectorielle et ce qui ne l'est pas. Cette analyse produit une *représentation vectorielle* où les opérations vectorielles sont mises en évidence. Pour sa part, la *génération* s'intéresse à la traduction d'une représentation vectorielle dans l'assembleur de la machine cible. Notons que la définition de la vectorisation donnée ici n'est pas en accord avec celle implicitement utilisée par certains auteurs pour lesquels elle n'est pas forcément totalement indépendante de la machine cible.

La vectorisation

La vectorisation est indépendante de la machine cible. Cette phase analyse les dépendances entre les données. A partir de celle-ci, les instructions sont analysées afin de déterminer ce qui peut être mis sous une forme vectorielle. La représentation vectorielle peut prendre différentes formes. Elle peut s'exprimer par l'intermédiaire d'un langage scalaire où les constructions du langage (boucles Do-Loops en Fortran) indiquent les instructions vectorielles (technique utilisée par Parafrase-2). Elle peut être un langage vectoriel (technique utilisée par PFC). Notons qu'un langage autorisant l'expression d'algorithmes vectoriels peut ainsi être utilisé comme représentation vectorielle (technique utilisée par EVA). La représentation vectorielle peut également avoir la forme d'un graphe (technique utilisée par VATIL).

La génération

La génération est dépendante de la machine cible. Les structures vectorielles détectées à la vectorisation sont transformées en langage machine de la cible. La génération inclut le découpage des instructions vectorielles (strip-mining), l'allocation des registres et l'ordonnancement statique des instructions.

Le découpage a pour but de transformer des traitements sur des vecteurs de tailles quelconque en une suite de traitements sur des vecteurs qui peuvent être effectivement traités par la machine cible.

L'allocation des registres consiste tout d'abord à placer les opérandes des instructions dans les registres (pour les processeurs registre à registre).

L'ordonnancement statique des instructions tente d'optimiser le partage des unités fonctionnelles entre les instructions. Le but final est de minimiser le temps d'exécution d'une séquence d'instructions, et du programme tout entier.

3. Les langages machines des super-calculateurs

Nous présentons ici les langages machines des supercalculateurs vectoriels pipelines. Nous nous attachons à découvrir leurs caractéristiques communes et leur différences. Nous dégageons une classification de ces langages machines (supercalculateurs RISCs - Reduced Instruction Set Computer / CISCs - Complex Instruction Set Computer), classification qui rappelle celle qui existe pour les architectures et langages machines des microprocesseurs scalaires classiques. Il ne faut pas tenter de rechercher un lien direct entre les architectures RISCs ou CISCs scalaires avec les architectures des supercalculateurs que nous qualifions de RISCs ou de CISCs.

Pour ces machines qui se veulent les plus puissantes, il est indispensable d'offrir à la base un jeu d'instructions permettant de tirer une puissance maximum de l'architecture et de pouvoir tirer partie de toutes les spécificités du matériel. Etant donnée la complexité inhérente aux processeurs vectoriels pipelines, leurs unités ont des fonctionnalités assez simples, très orientées et donc optimisées selon l'utilisation qui est faite de ces machines (calculs scientifiques, sur des nombres réels).

3.1 Les supercalculateurs "RISCs"

Les représentants numéro un des supercalculateurs RISCs sont les processeurs Cray. Leurs jeux d'instructions sont très "primitifs" et sont identiques en ce qui concerne les derniers X-MP commercialisés et le Y-MP. Ce jeu d'instructions est directement issu du Cray-1 auquel quelques améliorations ont été apportées. Sur le Cray X-MP, des instructions d'accès mémoires indirects (opérations de rassemblement / éclatement) et des instructions autorisant la synchronisation entre les processeurs (les calculateurs Cray étant devenus multi-processeurs) ont été introduites. Ce sont des processeurs registre à registre. Les interactions avec la mémoire sont donc simplifiées et localisées aux instructions d'accès mémoire.

Chaque unité fonctionnelle possède des fonctionnalités qui sont très précisément reflétées dans le jeu d'instructions par une ou plusieurs instructions. Par exemple, il n'y a ni unité fonctionnelle de division, ni instruction de division. Une division des éléments de deux vecteurs est réalisée en chaînant un pipeline calculant l'inverse des éléments d'un vecteur avec celui de multiplication. Ainsi, le calcul d'un simple inverse est moins coûteux que s'il avait utilisé une unité de division générale et le coût de la réalisation d'une division est minimisé par le chaînage des pipelines.

En plus de ces instructions, d'autres réalisent des transferts de données entre les registres et les chargements/déchargements de registres avec la mémoire. Finalement, des instructions prises en compte par l'unité de contrôle du processeur effectuent le contrôle du flux d'instructions. L'adressage des données en mémoire est très simple. Les instructions spécifient l'adresse de base de la zone mémoire contenant le vecteur. L'accès est soit contigu, avec pas ou réalisé via un

vecteur d'index contenu dans un registre vectoriel du processeur. Pour accéder un vecteur via un vecteur de bits, le vecteur de bits est préalablement transformé en vecteur d'index. Un accès dispersé via un vecteur d'index est ensuite réalisé. Les accès scalaires offrent l'adressage indirect indexé et autorisent également le transfert de blocs de registres B et T (caches pour les registres d'adresses A et les registres scalaires S).

L'initiateur d'instructions détecte les conflits et chaîne automatiquement les instructions qui peuvent l'être.

Par ailleurs, à un tout autre niveau, notons que les machines Cray sont les seules machines qui nous concernent ici et qui ne supportent pas la mémoire virtuelle. Bien que cela semble apparemment un défaut compliquant la programmation, c'est en fait un avantage au niveau matériel, mais aussi pour l'utilisateur qui contrôle totalement les accès à des données se trouvant en mémoire périphérique. Ainsi, il ne peut y avoir de défaut de page entraînant un blocage du processeur, avec tous les inconvénients que le déclenchement d'une interruption imprécise peut avoir au niveau du fonctionnement des pipelines.

Bien que n'autorisant pas le chaînage de ses pipelines, nous classons également le processeur du Cray-2 parmi les processeurs vectoriels RISCs. Nous rangeons également dans la classe des RISCs le processeur des machines NEC SX-2 et SX-3.

3.2. Les supercalculateurs "CISCs"

A l'opposé des processeurs Cray, il existe des processeurs dont les instructions sont assez complexes, à l'instar du Cyber-205 (dont le jeu d'instructions est repris et étendu sur l'ETA10) ou du co-processeur vectoriel de IBM VF. D'ailleurs, ce dernier est un peu particulier du fait que ses fonctionnalités ont été définies indépendamment de son architecture réelle (cf. infra). Aussi, il ne peut exister de correspondance un à un entre les instructions et les unités fonctionnelles de l'IBM VF. Ces processeurs se caractérisent soit par des jeux d'instructions étendus (Cyber-205) ou des modes d'adressages et de contrôle des opérations puissants (IBM).

Le Cyber inclut des instructions de calcul d'exponentielles, de calcul de moyenne, produits scalaires, ... Certaines de ces instructions sont microcodées. Travaillant de mémoire à mémoire, les opérations adressant des vecteurs non contigus sont décomposées en une première phase où les éléments du vecteur non contigu sont rassemblés pour former des vecteurs contigus qui seront traités durant la seconde phase (accès à un vecteur par pas, rassemblement selon un vecteur d'index ou compression via un vecteur de bits) : les instructions calculatoires ne traitent que des vecteurs contigus.

Concernant les modes d'adressages vectoriels, l'IBM peut exécuter ses instructions de calcul sous le contrôle d'un masque (*vector mask mode*). Il peut également accéder un opérande d'une instruction suivant un pas dont la valeur se trouve dans un registre scalaire. Bien que plus étendu que celui des Cray, son jeu d'instructions est beaucoup plus simple que celui du Cyber-205.

Les possibilités de chaînage de pipelines sont restreintes. Sur les Cyber et ETA, seuls les pipelines d'addition et de multiplication peuvent être chaînés. Ils ne le sont que si une séquence d'instructions d'addition et de multiplication est précédée par une instruction *link*. Sur l'IBM, seule une instruction combinant multiplication et addition autorise le chaînage (alors automatique) de pipelines. Ce sont les seuls cas où il y a chaînage sur ces machines.

3.3. Les changeants

Toute classification souffre d'individus qui s'insèrent mal dans le moule créé. Les processeurs des Fujitsu VP sont de ceux-là. Comme les CISCs, ils possèdent un jeu d'instructions étendu

(instructions de calcul de récurrences linéaires, recherche des extrema parmi les composantes d'un vecteur, ...). Certaines instructions combinent les fonctionnalités de plusieurs unités fonctionnelles. Par ailleurs, les instructions calculatoires peuvent être exécutées sous le contrôle d'un masque.

Cependant, comme les RISCs, le chaînage de leurs pipelines est possible et réalisé automatiquement par l'initiateur d'instructions. Comme les processeurs Cray, ce sont des processeurs registre à registre. Les instructions d'accès aux vecteurs contigus peuvent être contrôlées par un masque. De cette manière, des accès via des vecteurs de bits peuvent être réalisés.

3.4. Caractéristiques communes - Conclusion

Tous les processeurs des supercalculateurs vectoriels pipelines ont des langages machines qui se ressemblent beaucoup. Ceux-ci sont toujours des langages à 3 ou 4 adresses. Les différences et la classification qui en a été proposée reposent sur l'écart séparant les instructions des fonctionnalités des unités fonctionnelles. Nous discutons ici quelques caractéristiques générales qui se retrouvent dans les jeux d'instructions de ces machines.

Les vecteurs de bits

Concernant la manipulation et l'utilisation des vecteurs de bits, ceux-ci sont toujours créés pour des opérations de comparaisons vectorielles. Sur les processeurs RISCs, leur utilisation se réduit à de simples affectations contrôlées par le masque (instruction `merge` des Cray). Elle peut s'étendre au contrôle de l'activité des unités fonctionnelles, en masquant les composantes du résultat qui doivent être mises à jour (IBM, Fujitsu par exemple).

Apports pour la "vectorisabilité"

L'évolution des jeux d'instructions des supercalculateurs avec l'inclusion des opérations de rassemblement/éclatement et des opérations linéairement récurrentes (cf. infra) autorise un taux de vectorisation plus élevé. En effet, les expressions comportant une indirection dans l'accès aux composantes d'un vecteur du type :

```
V [ D [i] ] := exp
X := W [ D [i] ]
```

utilisent les instructions de rassemblement / éclatement vectorielles (le Cray-1 réalisait ces opérations de manière totalement séquentielle). Etant réalisée vectoriellement, le résultat d'une instruction de rassemblement peut être chainée à un autre pipeline, lui-même chainée à un port mémoire réalisant une opération d'éclatement. C'est le même principe pour les opérations de récurrences linéaires :

```
A [i] := B [i] * A [i-1] + C [i]
```

qui sont incluses dans le jeu d'instructions des Fujitsu et NEC. Ainsi, les boucles contenant une structure de ce type peuvent alors être reconnues de nature vectorielle par le compilateur (c'est à dire, exécutée de manière vectorielle par le processeur). Cela mène alors généralement au chaînage des unités d'addition et de multiplication.

Prise en compte des dépendances au niveau du matériel

Les dépendances entre données posent des problèmes lorsque plusieurs instructions peuvent être en cours d'exécution à un moment donné, comme cela est possible sur les processeurs pipelines discutés ici. Les dépendances posent de réels problèmes au programmeur lorsqu'elles concernent des données se trouvant en mémoire. En ce qui concerne les données présentes dans les registres vectoriels, des mécanismes sont généralement mis en place et utilisés par l'initiateur

d'instructions qui ne déclenche l'exécution d'une instruction que si ses opérandes sont disponibles. En ce qui concerne les données situées en mémoire, pour que l'exécution d'un programme soit exacte, il faut pouvoir garantir qu'une donnée a été écrite en mémoire avant qu'elle n'y soit lue. Pour cela, des instructions sont disponibles pour la prise en compte des problèmes de dépendances au sein d'un processeur.

Sur les processeurs Cray, deux instructions permettent de lever les problèmes de dépendances entre données au niveau d'un processeur. L'instruction DBM interdit que des accès en lecture et des accès en écriture soient réalisées en même temps. Si une lecture est déclenchée après une écriture, la lecture reste bloquée tant que l'écriture n'est pas terminée. Notons que l'instruction EBM autorise à nouveau que des lectures soient réalisées en parallèle à une écriture. Pour sa part, l'instruction CMR assure que tous les accès mémoires en cours peuvent être considérés comme terminés. C'est à dire qu'elle reste en cours d'exécution et qu'elle bloque les instructions suivantes tant que le moment de terminaison des derniers accès n'est pas connu. Par ailleurs, lorsqu'elle a terminé son exécution, si une instruction de chargement est déclenchée, on est assuré que la donnée lue sera correcte par rapport aux éventuelles écritures de cette donnée réalisées avant le CMR.

Sur le processeur des supercalculateurs Fujitsu, il est intéressant de noter les instructions VPT et VWT qui permettent la synchronisation et la résolution des problèmes de dépendances entre opérandes sources et cible d'instructions différentes. Une instruction VPT positionne une dépendance. Elle reste ensuite en cours d'exécution tant que les instructions qui ont été déclenchées avant elle sont en cours d'exécution. Les instructions suivantes sont déclenchées normalement (d'autres VPT sont éventuellement déclenchés à nouveau). Pour sa part, l'instruction VWT est en cours d'exécution tant que des VPT déclenchés avant elle sont en cours d'exécution. Aucune instruction se trouvant après le VWT dans le flot d'instructions ne peut être déclenchée tant que le VWT s'exécute. Ces deux instructions de synchronisation permettent donc la levée des dépendances de sortie et des antidépendances. Quant aux dépendances de flot, elles entraînent le chaînage des instructions (cf. fig. I.7.).

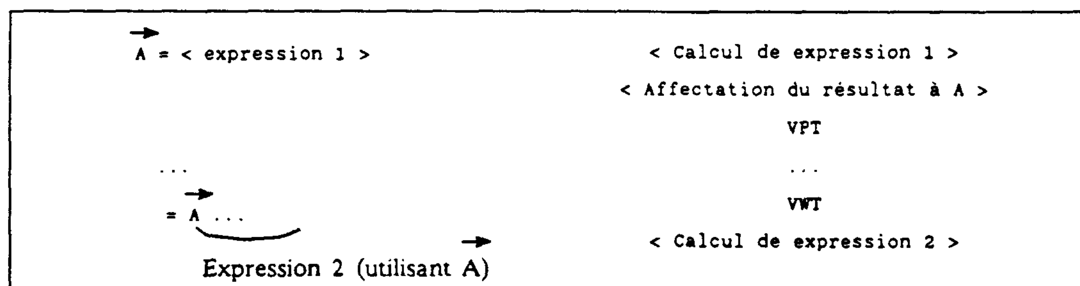


Figure I.7. –
Les instructions de contrôle des dépendances sur les Fujitsu VP

Dans un environnement multi-processeurs, les dépendances entre des données produites par un processeur et utilisées par un autre sont réglées par des communications inter-processeurs.

Résumé

	X-MP	Y-MP	2	SX-3	VP-200	VP-2000	IBM 370	Cyber-205	ETA^{10}
chainage des pipelines									
automatique	x	x		x	x	x			
non automatique								x	x
accès vectoriels en mémoire masqués				???				x	x
opérations masquées				???	x	x	x	x	x
récurrence linéaire				x		x			
opérations avec pas							x		
affectation masquée	x	x	x	x	x	x	x	x	x
instructions de communications inter-CPUs	x	x	x						x

RISC ⟶ CISC

Table I.2. -
Quelques caractéristiques de quelques supercalculateurs

La table I.2. résume quelques caractéristiques des supercalculateurs discutés ici. L'affectation masquée est l'instruction dénommée *merge* par Cray (différent de celle du Cyber-205), c'est à dire l'instruction qui effectue l'affectation de deux sources dans une cible sous le contrôle d'un masque.

4. Les machines abstraites

L'idée d'utiliser une étape intermédiaire de compilation entre le langage évolué et le langage machine remonte aux débuts de l'informatique, dans les années 50. Cette technique possède plusieurs avantages :

- la partie frontale du compilateur (prenant un source en langage évolué et générant du langage intermédiaire) est indépendante de la cible;
- un programme écrit en langage intermédiaire est indépendant d'une cible particulière;
- les parties frontales et arrières du compilateur sont assez simples à écrire;
- l'ajout d'un nouveau langage source implique l'écriture d'une seule partie frontale pour l'implantation sur toutes les machines supportées par le langage intermédiaire (cf. fig. I.8.a);
- l'ajout d'une nouvelle cible implique l'écriture d'une seule partie arrière pour que cette cible soit accessible depuis les langages évolués supportés par le langage intermédiaire (cf. fig. I.8.b).

Nous présentons dans cette partie différents langages et représentations intermédiaires ainsi que des machines abstraites. Les langages intermédiaires pour cibles scalaires ont fait l'objet de très

nombreux travaux. Pour les cibles parallèles, leurs développements ont mené et mènent actuellement à de nombreux travaux. Par contre, ce n'est pas le cas dans le domaine vectoriel. Nous nous intéressons ici à chacun de ces types de cibles.

Loin de viser l'exhaustivité, nous donnons ici quelques jalons dans l'histoire et l'évolution des langages intermédiaires. Il y a probablement à peu près autant (sinon plus) de LI que de langages évolués et le passage en revue de chacun ne serait pas d'un grand intérêt.

4.1. Evolution

En 1958, une commission [STRONG & al. 58] présente un projet très ambitieux visant à la définition d'un langage intermédiaire universel, UNCOL (UNiversal Computer Oriented Language). Ce langage est sensé se situer à l'interface entre tous les langages et toutes les machines. Trois ans semblaient nécessaires à sa définition. Une révision importante était attendue dix à quinze ans plus tard, pour suivre les progrès du matériel. Ce projet n'a jamais abouti. Une tentative d'implantation d'UNCOL a été rapportée. Etant donnée la diversité des langages (langages procéduraux, fonctionnels, logiques, orientés objets, ...), la réalisation d'un tel LI semble de plus en plus utopique et n'est plus à l'ordre du jour.

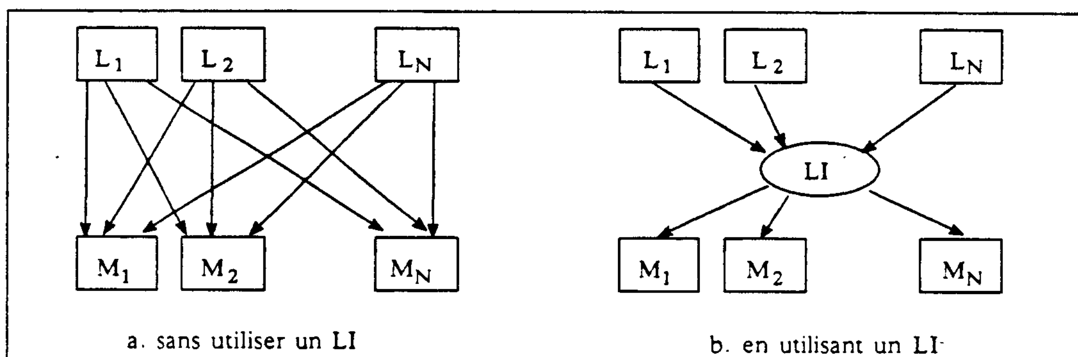


Figure I.8. -
Titre de la figure

Début 1960, le système SLANG [SIBLEY 61] tente une approche radicalement différente de celle de UNCOL. Dans celui-ci, un langage de description d'algorithme SLANG-POL (Problem Oriented Language) est utilisé afin d'exprimer l'algorithme à compiler. Celui-ci est transformé en une représentation intermédiaire, en langage EMIL (machine abstraite E-machine). Cette représentation est alors traduite en langage machine pour la cible. L'originalité repose sur l'utilisation d'une description de la machine cible à la compilation et le code en EMIL est généré en conséquence. Cette description de la machine est exprimée dans le langage SLANG-MD (Machine Description) et donne des indications sur l'organisation matérielle de la machine et sur la traduction des instructions EMIL. L'organisation matérielle est décrite d'après la nature et le nombre des différentes unités de stockage, le format des codes instructions, les registres d'index, l'existence ou non d'un mode d'adressage indirect, l'existence ou non de données de type réel, et diverses informations sur les dispositifs périphériques (dérouleurs de bandes, lecteurs de cartes, ...). Par ailleurs, à chaque instruction EMIL est associée la séquence d'instructions de la cible l'émulant. Contrairement à l'approche habituelle, la compilation est dirigée par les caractéristiques de la cible. Pour une cible donnée, certaines instructions EMIL ne sont pas générées car elles ne peuvent pas être traduites (efficacement) sur la cible. Sans préciser plus ses pensées, Sibley indique que SLANG n'était pas conçu pour être implanté sur toutes les machines. Par ailleurs, les deux problèmes principaux du projet étaient la définition du langage de description de machine SLANG-MD et l'efficacité du code produit.

4.1.1. Les processeurs de macros

Les premières approches pour l'écriture de programmes portables consistent à utiliser un macro-processeur. Le programme source est alors une suite d'appels de macros. Le compilateur utilise un fichier de définitions de macros. Chaque macro s'expanse dans le code qui est à produire pour exécuter l'action correspondant à la macro. Nous citerons SIMCMP (SIMple CoMPiler) en 1967 [ORGASS & al. 69] qui était assez primitif puis, STAGE 2 en 1969 [WAITE 70a] [WAITE 70b] qui est une amélioration de SIMCMP. STAGE 2 est décrit dans le langage de la machine virtuelle FLUB conçue spécialement pour écrire STAGE 2 et donc définie pour le traitement des macros. Le langage de FLUB se compose de 28 opérations et 2 pseudo-instructions. STAGE 2 se compose de 850 instructions FLUB. Au moins 15 implantations de STAGE 2 sur des machines différentes ont été réalisées.

4.1.2. Les langages intermédiaires compilés

En 1969, le O-code apparait comme l'un des tout premiers langages intermédiaires tels que nous les connaissons aujourd'hui [RICHARDS 71]. Celui-ci est conçu pour être le langage intermédiaire utilisé pour l'écriture de compilateurs portables BCPL. Les auteurs du O-code critiquent l'utilisation de macro-processeurs arguant l'impossibilité (ou presque) d'écrire des compilateurs pour un langage comme Algol. Les données manipulées dans le O-code sont toutes codées sur le même nombre de bits. Il y a trois classes de données, les locales, les rémanentes et les globales. Il y a 56 instructions. Elles ressemblent assez à des macros. La récursivité des fonctions et des procédures est supportée. L'évaluation des expressions utilise une pile. Il y a eut 10 à 20 implantations de BCPL avec le O-code.

Vers 1972, le langage Pascal est défini accompagné de son langage intermédiaire, le P-code. Différentes versions du P-code seront développées, P-4 (P-code standard), P-2 (qui est le P-code utilisé dans la définition du Pascal UCSD) et LASL (P-code utilisé comme langage intermédiaire pour un compilateur Pascal dédié au Cray-1 et un compilateur du langage MODEL, extension de Pascal pour l'écriture de grosses applications). Ces versions sont présentées et comparées dans [NELSON 79]. Le P-code, ou plutôt la P-machine, utilise une pile d'évaluation d'expressions. Elle manipule des données de types divers (entiers, caractères, réels) et possède des instructions pour accéder les éléments de structures ou de tableaux et pour manipuler des pointeurs ainsi que les objets pointés. On quitte résolument les "macro-like languages", un programme en P-code ressemblant beaucoup à un programme en assembleur.

Par la suite, de nombreux langages intermédiaires vont être proposés afin de permettre simplement la portabilité des langages, sur les machines scalaires essentiellement. En général, tous ces langages intermédiaires ont une forme similaire, ressemblant à de l'assembleur pour une machine à pile. Par ailleurs, les représentations intermédiaires des programmes sont générées de la même manière. L'acceptation d'une règle de la grammaire du langage entraîne la génération d'une certaine séquence de code intermédiaire. Nous citerons simplement quelques exemples comme Janus proposé par l'équipe de Waite en 1974 [COLEMAN & al. 74] [HADDON & al. 78], EM proposé par l'équipe de Tanenbaum et dédié à son " Amsterdam Compiler Kit" [TANENBAUM & al. 83], ou le A-code qui est une extension du P-code définie pour le langage Ada afin d'en supporter les spécificités [DOMMERGAARD 80] (tâches, exceptions, ...). Comme nous l'avons déjà signalé, toutes ces machines virtuelles sont à pile. De ce point de vue, quelques rares exceptions sont le Q-code qui est généré à partir du P-code pour la génération de code pour des machines à registres [WILK & al. 83], ou le Z-code, langage intermédiaire utilisé par le compilateur Algol [GARDNER 77]. A ces deux LI sont associées des machines virtuelles à registre, et non à pile.

4.1.4. Les représentations internes aux compilateurs

Les représentations internes utilisées par les compilateurs sont nombreuses. Sans avoir le souci d'exhaustivité, nous citerons les arbres, graphes, la notation polonaise inverse, les langages de

triplets, quadruplets. Les unes ont une structure complexe (arbres, graphes), les autres ayant une structure linéaire (triplets).

A titre d'exemples, nous citerons le langage intermédiaire (M-code) définit par C. Eisenbeis qu'elle utilise pour l'optimisation de micro-code (scalaire) de machines tableaux (ST-100 et FPS-164) [EISENBEIS 86]. Le M-code utilise une infinité de registres et une mémoire où sont stockées les données. Son jeu d'instructions comporte les opérations d'addition, soustraction, multiplication et division scalaires, lecture/écriture entre les registres et la mémoire et une insruction de transfert entre deux registres. Le code généré à partir de boucles Fortran (ne contenant pas de branchement) en M-code est optimisé, puis du code machine est généré. L'optimisation porte sur l'obtention d'un taux de parallélisme optimal et effectue une recherche et une élimination des sous-expressions communes et des morceaux de code inaccessibles.

Le vectoriseur Parafrase-2 [POLYCHRONOPOULOS & al. 90] accepte des sources en C et Fortran et génère son résultat en Fortran ou dans une extension vectorielle de C. Afin d'avoir un seul type d'analyse à réaliser, il commence par transformer le programme source dans une représentation interne. C'est sur cette représentation qu'il effectue ces analyses pour la vectorisation. Il transforme ensuite cette représentation interne vectorielle dans le langage de sortie.

4.1.5. Les langages intermédiaires interprétés

Un certain nombre de systèmes sont basés sur une pseudo-compilation. L'exécution consiste alors dans l'interprétation d'une représentation intermédiaire du programme source générée par un compilateur. Les systèmes Forth, Pascal-UCSD et Smalltalk fonctionnent selon ce principe.

Ce type de représentation intermédiaire est très bien adapté à des langages où il n'existe pas de séparation nette entre données et programmes (d'où son utilisation en Smalltalk). Aussi, un nouveau programme peut être généré par l'exécution d'un autre, et son exécution déclenchée ensuite. Dans ce genre de cas, où le programme doit agir sur lui-même, une compilation complète semble tout à fait inadaptée.

Par ailleurs, cette technique simplifie l'écriture des compilateurs. Le traducteur du langage évolué en langage intermédiaire est totalement portable. Seul un ensemble de primitives est à ré-écrire pour chaque cible, celle-ci étant généralement assez simple. C'est la raison essentielle de l'utilisation de la pseudo-compilation en Forth ou Pascal-UCSD.

4.2. Langages intermédiaires et machines abstraites parallèles

Une couche intermédiaire parallèle se compose d'un ensemble de primitives définies abstraitement (indépendamment de la machine sur laquelle elles sont implantées) qui propose au programmeur une couche de fonctionnalités lui simplifiant l'écriture d'applications. La définition de langages permettant la programmation de machines parallèles (ou plutôt, d'algorithmes parallèles) pose un problème non véritablement résolu à l'heure actuelle. Ces couches peuvent être implantées sur machines parallèles, mais aussi sur des machines scalaires, ou des réseaux de machines scalaires (stations de travail reliées par un réseau Ethernet par exemple). Ces couches peuvent être composées de plusieurs niveaux. Le niveau le plus bas se situe juste au dessus du matériel. Le niveau le plus élevé constitue un jeu de primitives, n'ayant plus (ou peu) de liens avec l'architecture sous-jacente. Cette décomposition en couches confère une plus grande fiabilité de même qu'elle simplifie l'écriture de la machine abstraite. Une présentation de ce modèle en couche sera trouvée dans [GARRETT 90].

Linda est l'une des premières couches abstraites ainsi définie [AHUJA & al. 86]. Linda vise tant les hypercubes (du type Intel) que des réseaux de micro- ou mini-ordinateurs reliés par Ethernet

(Micro-VAX). La définition de Linda repose sur celle de sa mémoire, l'espace des t-uplets (*Tuple-space*). Les données traitées dans Linda sont des ensembles (t-uplets) de valeurs atomiques. Un ensemble de processus se partagent cet espace de t-uplets. Chacun peut y écrire, y lire ou y prendre (consulter) un t-uplet. Un gestionnaire des processus exécute les requêtes d'accès à l'espace des t-uplets les unes après les autres. Tant qu'une requête émise par un processus ne peut être réalisée effectivement, le processus émetteur de la requête est bloqué. Un programme parallèle Linda consiste en un ensemble de processus, non ordonnés dans le temps. Au-dessus de Linda, des langages peuvent être étendus pour inclure les fonctionnalités de Linda. Ainsi, C-Linda et Fortran-Linda ont d'ors et déjà été implantés.

Nous citerons également VMMP qui fournit un ensemble de primitives visant la programmation des machines scalaires, les multi-processeurs à mémoire partagée (cf. Cray Y-MP) ou distribuée (réseaux de Transputers) [GABBER 89].

4.3. Langages intermédiaires et machines abstraites vectoriels

Peu de travaux concernant des langages intermédiaires vectoriels ont été réalisés à ce jour. Sur les machines parallèles, les LI proposent une couche virtuelle au dessus des architectures, ceci dans le but de simplifier leur programmation et d'autoriser la portabilité des programmes. Concernant les machines vectorielles, les utilisateurs n'ont jamais ressenti le besoin d'ajouter une couche virtuelle simplifiant leur programmation. Les problèmes de portabilité sont résolus partiellement par l'utilisation de Fortran 77 scalaire. Des approches expérimentales ont mené au développement de langages ou représentations intermédiaires. Par ailleurs, des architectures virtuelles ont été proposées et implantées par les grands constructeurs. Nous présentons ici ces différentes approches vectorielles.

4.3.1. Une représentation intermédiaire pour Actus

Actus est un langage vectoriel qui a été effectivement implanté sur architectures vectorielles pipelines (Cray-1) et SIMD (ICL DAP). Ce langage autorise l'expression du parallélisme des algorithmes directement dans sa syntaxe. Actus utilise une représentation intermédiaire. Le système de développement d'Actus est décrit dans [PERROTT & al. 83]. Un source Actus est transformé en un graphe abstrait mettant en évidence l'exécution parallèle du programme. Ce graphe est alors transformé en un autre graphe où sont prises en considération les caractéristiques de la machine cible. (Sur un Cray-1, cette transformation a pour effet, par exemple, de découper les boucles en tranches de 64 éléments - puisque les registres vectoriels contiennent 64 composantes.) Un convertisseur transforme alors la représentation intermédiaire d'un programme en assembleur ou langage machine de la cible en utilisant une description de la cible. Cette description indique les registres et instructions disponibles sur la cible. Le LI possède une structure s'apparentant au P-code et utilise une notation post-fixée. Actus manipulant des vecteurs, le LI étend les opérateurs scalaires pour qu'ils prennent en compte des opérandes vectoriels. Il inclut également une description des variables du programme, le LI travaillant directement avec les noms de variables du programme.

Nous allons dire quelques mots de la représentation de la machine cible utilisée par le convertisseur Actus. Cette description est basée sur les seules références disponibles sur le sujet que sont [PERROTT & al. 83] et [PERROTT & al. 85] qui présentent cette description pour le Cray-1. Le LI du langage Actus est traduit en langage cible par utilisation d'une table de traduction donnant la correspondance de chaque instruction du LI en une séquence d'instructions de la cible. Cette technique de génération se rapproche beaucoup de l'utilisation des macro-processeurs. A chaque instruction de la machine

abstraite de Actus est associée une séquence d'instructions en CAL(7). Une liste des classes de registres disponibles est jointe. Une classe de registres est un ensemble de registres du Cray-1 qui possèdent une même fonctionnalité et sont adressés de la même manière dans les codes instructions. A une classe est associée le type des valeurs que ses registres peuvent contenir. A une séquence d'instructions CAL est associé l'ensemble des registres utilisés dans ce code, les P-registres. Ceux-ci jouent le rôle de paramètres pour le bout de code. Ainsi, pour un bout de code donné, les données sources et cibles peuvent se trouver dans n'importe quel registre. En fait, selon la classe des registres contenant les données d'un bout de code, le code à générer peut différer. Aussi, à chaque instruction de la machine abstraite est associée une liste de bouts de code, dépendant de la classe des registres dans lesquels se trouvent les données. A côté de la table contenant la traduction des instructions du LI, une autre table contient des bouts de code utilisés pour le transfert de données entre des registres de classes différentes. Lors de la traduction d'une instruction LI en CAL, il y a donc d'abord identification des registres qui seront effectivement utilisés, sélection de la séquence de code ad hoc puis expansion du bout de code avec les registres effectivement utilisés. Cette identification entraîne le remplacement des P-registres par le nom des registres effectivement utilisés. De cette manière, le bout de code traduisant en CAL l'instruction LI a été obtenu. Le convertisseur peut passer à l'instruction suivante. Notons que l'allocation des registres se fait d'abord pour les objets (variables ou temporaires) des boucles les plus internes.

4.3.2. Machines abstraites vectorielles

Les deux grands constructeurs IBM et DEC ont introduit des modèles de machines abstraites virtuelles vectorielles (*vector facility* pour IBM [PADEGS & al. 88], *VAX vector architecture* pour DEC [BHANDARKAR & al. 90]) pour faire suite à leurs machines virtuelles scalaires (IBM Model 370 et VAX Architecture). Ce sont des définitions d'architectures vectorielles "génériques", chaque modèle de machine ou coprocesseur vectoriel se voulant une instanciation de cette architecture virtuelle. Les fonctionnalités de ces architectures sont spécifiées. Leurs langages machines (et le codage de leurs instructions) le sont également. Par contre, la nature exacte et le nombre des unités fonctionnelles et la taille des registres vectoriels ne sont pas précisés. Le nombre de composantes des registres vectoriels est un paramètre matériel. Ainsi, les programmes utilisant ces paramètres sont portables d'un modèle à un autre. Sur certain modèle, des instructions sont prises en compte par des unités fonctionnelles spécialisées, alors que sur d'autres elles sont émulées. De cette manière, des programmes vectoriels peuvent être émulés sur une architecture scalaire. Pour ces architectures abstraites, le langage machine est donc un langage portable sur toutes les machines de la gamme. Les problèmes de compatibilité se trouvent essentiellement au niveau de la réaction des programmes face aux interruptions.

4.3.3. Le V-code

Le V-code est un langage intermédiaire vectoriel développé actuellement à l'Université Carnegie-Mellon [BLELLOCH & al. 90a] [BLELLOCH & al. 90b]. Il sert d'intermédiaire entre d'une part Fortran 90, C* (extension parallèle du langage C pour la Connection-Machine), Paralation Lisp et CM-Lisp (Lisp pour la Connection-Machine), d'autre part la Connection-Machine (machine SIMD), le Cray Y-MP, la machine iWarp (machine parallèle développée à Carnegie-Mellon). D'emblée, la cible numéro un est la Connection-Machine. Comme le P-code, le V-code utilise une pile d'évaluation pour réaliser ses calculs. Sa mémoire contient uniquement des vecteurs : à toute adresse correspond un vecteur de taille quelconque. Ces vecteurs sont placés sur la pile afin de les traiter. Les vecteurs sont les seuls types d'objets manipulés : un scalaire est un vecteur à un élément. Des instructions vectorielles classiques (addition, ...), de permutation des éléments et de réduction sont disponibles.

(7) - CAL : Cray Assembly Language : Langage d'assemblage des processeurs Cray

4.4. Conclusion sur les langages intermédiaires

Il existe une distinction très nette entre les machines virtuelles dites *machines-union* et celles dites *machines-intersection*. Cette distinction apparaît au niveau du jeu d'instructions de la machine virtuelle, est très profonde et possède de multiples conséquences. Une machine-union possède un jeu d'instructions qui est l'union des jeux d'instructions de toutes les machines typiques (les machines cibles). Une machine-intersection possède un jeu d'instructions qui est un sous ensemble des fonctionnalités qui peuvent être attendues de toutes les cibles. Une analogie peut être établie entre, d'une part, machine-union et architecture CISC et, d'autre part, machine-intersection et architecture RISC. Les machines-union ont un ensemble d'instructions très étendu; des exemples sont UNCOL et le P-Code (219 instructions pour le P-Code). En général, toute instruction d'une machine-union possède un sens proche d'une instruction de la cible et se compile en une ou deux instructions.

Les instructions d'une machine-intersection sont assez peu nombreuses (une centaine). Chacune propose une fonctionnalité et se comporte comme une primitive. La machine-intersection propose ainsi une "boîte à outils" de primitives. Une machine-intersection est plus difficile à concevoir. La synthèse de son jeu d'instructions n'est pas évidente. Il faut dégager les fonctionnalités "génériques" des langages machines cibles directement visées et des cibles potentielles. Du fait de l'"éloignement" des instructions du langage intermédiaire par rapport à la cible, l'obtention d'un programme cible en code machine de bonne qualité nécessite une optimisation du code machine généré.

Une machine union ou une machine intersection pure n'existe pas dans la réalité et ne serait pas d'un grand intérêt. La génération d'un code de bonne qualité à partir d'une machine-union pure serait difficile. En effet, les instructions pêchent alors par une trop grande généralité. Des optimisations du code généré seraient nécessaires. Une machine-union pose d'une part le problème de la génération de son code à partir du langage de haut niveau, d'autre part le problème de sa traduction en langage cible. Sa génération est complexe du fait que le compilateur aura généralement plusieurs manières de traduire une même structure de haut niveau, d'où une étude de cas se révèle nécessaire qui doit mener au "bon choix". Sa traduction est complexe également car les instructions du langage intermédiaire ont alors une sémantique très précise (non générale). Il sera alors difficile de trouver une instruction sur la machine cible ayant exactement la même sémantique. Notons qu'une machine-intersection est plus facilement portable qu'une machine-union. L'ajout d'une nouvelle cible pour une machine-union risque en effet d'introduire des problèmes de traduction de ses instructions, avec des problèmes d'écarts sémantiques. Face à ces problèmes dus à des machines trop "pures", chaque machine abstraite est en fait une dérivée d'un de ces deux types de machines, plus ou moins orientée vers l'une des deux extrémités.

5. Conclusion de la partie I

Dans cette première partie, nous avons introduit et rappelé les notions de base du traitement vectoriel. Nous avons décrit l'architecture des supercalculateurs vectoriels pipelines ainsi que leur programmation. Nous avons constaté les évolutions des architectures, lesquelles ont suivi plusieurs axes :

- multiplication du nombre de processeurs : nous sommes passés des monoprocesseurs à des machines comprenant de 2 à 8 processeurs. La tendance est à une augmentation de ce nombre (Cray parle de 256 processeurs dans ces calculateurs, à la fin des années 90).

- évolution de l'organisation de la mémoire : nous sommes passés des processeurs mémoire à mémoire, utilisant une mémoire simplement découpées en bancs, aux processeurs registre à registre où la mémoire est structurée de façon arborescente (sections, sous-sections). Par ailleurs, des registres (vectoriels) et des caches sont disponibles avec des capacités de plus en plus importante. Le développement depuis le début des années 80 d'architectures hiérarchisées (Cedar, RP3, ...) confirme cette tendance.

- évolution des instructions de base des processeurs autorisant une exécution vectorielle d'opérations autrefois scalaires (accès vectoriels indirects, récurrences linéaires, ...).

L'utilisation de langages autorisant non pas la programmation d'une machine, mais l'expression d'algorithmes nous semble une voie à étudier. Nous allons ainsi vers des langages plus déclaratifs (mais demeurant procéduraux) où le programmeur doit donner les informations qu'il connaît sur l'algorithme qu'il programme afin qu'il puisse être compilé de manière efficace. Ceci passe nécessairement par une prise de conscience du problème de la part du programmeur. De toute façon, le programmeur est d'ors et déjà conscient d'un problème puisqu'il écrit ses sources Fortran en utilisant force directives de compilation ou d'autres "trucs" pour obtenir une exécution plus rapide de ses programmes.

Partie II - MAD et Devil

Dans cette deuxième partie, nous présentons en premier lieu la machine abstraite vectorielle MAD et son langage Devil. Nous nous intéressons tout d'abord à la définition de MAD, son fonctionnement, les objets qu'elle traite. La conception du langage Devil est ensuite présentée et justifiée. Devil donne à MAD son caractère de machine virtuelle intersection et lui assure sa portabilité. Les problèmes liés à la projection de MAD/Devil sur des architectures réelles sont ensuite discutées.

Après cette présentation logique, nous présentons une esquisse de réalisation matérielle d'une machine MAD. L'architecture de MAD est celle d'une machine tableau. Nous centrons notre exposé sur la mémoire d'une telle machine. Celle-ci est définie pour contenir des vecteurs ressemblant à ceux définis dans la machine MAD virtuelle.

Chapitre II
MAD et Devil – Définitions

Principes généraux

MAD a été conçue comme une machine abstraite. La compilation de programmes écrits en langages évolués utilise MAD pour générer du code pour une multitude de machines aux architectures diverses (scalaire, vectorielles pipelines, machines tableaux et les machines à mémoire distribuée). MAD est dédiée au traitement vectoriel.

Tout en reconnaissant les avantages de Fortran, nombre d'auteurs (parmi lesquels [PERROTT 79], [WETHERELL 80], [LI & al. 84], [WORLTON 81]) plaident pour l'utilisation et le développement de langages autorisant l'expression du caractère vectoriel des algorithmes. Ces langages doivent posséder des structures de base puissantes, tant en ce qui concerne l'expression des algorithmes qu'au niveau des structures de données. [WETHERELL 80] rapporte un sondage réalisé auprès de programmeurs scientifiques qui montre une forte demande en outils de description et de manipulation de structures de données élaborées. Il indique en conclusion : « *although the emphasis on data manipulation is surprising, it accords with my 'gut' feeling that data structures are at the heart of any language* ». C'est dans cet esprit que le langage EVA, la machine MAD et son langage Devil ont été définis.

Le langage EVA

EVA autorise l'expression directe d'algorithmes vectoriels. Le compilateur EVA génère du code pour la machine virtuelle MAD.

Nous commencerons par rappeler les objectifs visés à la conception du langage EVA et ses caractéristiques principales. Le langage vectoriel EVA a été présenté dans [DEKEYSER & al. 90b], [DEKEYSER & al. 90c] et sa grammaire complète est disponible dans [MARQUET 90]. EVA fera par ailleurs l'objet d'une partie de la thèse de doctorat de Ph. Marquet(8).

Les objectifs de EVA sont :

- de proposer un langage de programmation vectoriel offrant une approche différente de Fortran 90 quant à la manipulation de vecteurs;
- de proposer des outils vectoriels syntaxiques autorisant une lisibilité et une maintenance simplifiée des applications;

(8) - thèse dont la soutenance est prévue fin 1991

- d'être un langage portable sur toute architecture (ou presque), en visant avant tout les supercalculateurs et les stations de travail. Le développement et l'exécution d'applications peuvent ainsi être réalisées sur des machines différentes;
- de simplifier la tâche des programmeurs dans le développement d'applications vectorielles.

Par ailleurs, quelques propriétés caractéristiques de EVA peuvent être énoncées de la manière suivante :

- les algorithmes sont écrits de manière indépendante des architectures sur lesquelles ils s'exécuteront;
- une notation vectorielle explicite est utilisée plutôt que des appels à des routines vectorielles. Cette notation simplifie la génération d'un code efficace;
- les fonctions à résultats vectoriels généralisent les opérateurs vectoriels prédéfinis de EVA.

Le compilateur EVA utilise un fichier de paramétrage de la machine cible. Ces paramètres sont la taille (en bits) d'un mot mémoire et la taille des objets de base. Notons que la génération automatique de compilateurs/traducteurs pour différentes cibles est tout à fait en dehors de nos préoccupations dans ce document.

Devil et MAD

Devil étant utilisé comme langage intermédiaire du compilateur EVA, certaines caractéristiques sont communes à Devil et EVA et seront retrouvées ici.

Notons dès maintenant que le terme *compilateur* concernera le logiciel transformant une représentation d'un programme donnée en langage de haut niveau dans sa représentation en langage intermédiaire (*front end* dans la terminologie anglaise) et que le terme *traducteur* concernera le logiciel transformant cette représentation intermédiaire en un programme en assembleur ou langage machine de la cible (*back end* dans la terminologie anglaise). Associés aux termes compilateur et traducteur, les termes de *compilation* et *traduction* seront utilisés.

Les objectifs primordiaux de MAD/Devil sont les suivants :

- la définition d'une notion de vecteur puissante;
- la définition de Devil (MAD) comme une couche de niveau relativement élevé de traitements et de manipulations de vecteurs. Devil se situe entre les couches OS et outils de développements d'applications (cf. EVA);
- la définition d'une machine abstraite vectorielle autorisant la portabilité du langage EVA sur une multitude de cibles aux architectures très différentes (cf. fig. II.1). Les cibles visées appartiennent à de multiples classes d'architecture. Il y a les machines scalaires SISD (les stations de travail avant tout); les machines vectorielles pipelines (Cray, Fujitsu VP, IBM 3090 VF, ...) et les machines à architecture tableau (ICL DAP, Connection Machine..); les machines à mémoire distribuée (réseaux de Transputers ou de i860 à l'instar de l'iPSC, ...);

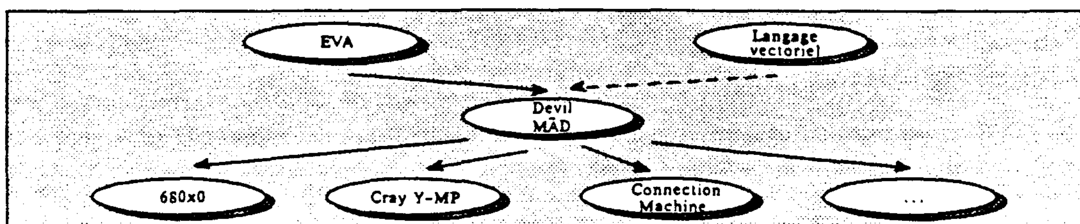


Figure II.1. -
MAD, une machine "au-dessus" des architectures existantes

- l'implantation de MAD (Devil) doit être efficace. Nous désirons que EVA soit utilisable effectivement sur les super–calculateurs par les scientifiques (physiciens et autres). En effet, EVA est sensé apporter une aide à la programmation scientifique vectorielle. Pour cela, il faut que les compilateurs EVA génèrent du code aussi efficace que les compilateurs Fortran (ou du moins, il faut montrer que cela est possible même si les implantations actuelles sont moins performantes que celles de Fortran).

- l'ajout d'une nouvelle machine cible doit être possible simplement. Les définitions de MAD et Devil ne doivent pas être modifiées pour cela. L'implantation sur une nouvelle cible doit se réduire à l'écriture d'un traducteur de Devil dans l'assembleur de la nouvelle cible. Par ailleurs, la nouvelle implantation doit être efficace.

1. La machine abstraite MAD et son langage Devil

Dans cette première partie, nous décrivons la machine abstraite MAD et son langage Devil. Nous nous attachons tout d'abord à définir et présenter la structure de la mémoire de MAD, les objets qu'elle manipule et leur allocation. Nous présentons ensuite le langage intermédiaire vectoriel Devil et ses caractéristiques. Des éléments de génération de code Devil à partir d'un langage (vectoriel) de haut niveau sont donnés. L'accent est mis sur les points de Devil qui permettent et simplifient sa traduction efficace pour une machine vectorielle.

1.1. Introduction

Particularités de MAD et Devil

Comme nous l'avons indiqué dans [DEKEYSER & al. 90d], les points centraux de la définition de Devil sont la définition des vecteurs, la conception du jeu d'instructions, le mode de réalisation des calculs et la mémoire vectorielle. Chacun de ces points a été pensé afin d'améliorer la portabilité de Devil, de même que l'efficacité de ses implantations. Ils sont présentés dans les paragraphes qui suivent.

Les vecteurs

La définition de vecteurs distingue la sélection des éléments d'un vecteur de leur traitement. La sélection inclut l'accès scalaire à un élément (indexation), les accès vectoriels contigus et avec pas et les accès vectoriels dispersés, via un vecteur d'index ou un vecteur de bits. Le traitement recouvre toutes les opérations qui mènent à une modification de la valeur des éléments d'un vecteur (par écriture). Aussi, nous avons associé un moyen de sélection (un *descripteur*) des composantes utiles (celles qui doivent être traitées) à une liste d'éléments. Cette association forme ce que nous nommons un vecteur. Plusieurs vecteurs peuvent se partager une même liste d'éléments. Chacun possède ses propres éléments sélectionnés (son propre "point de vue" sur la zone d'allocation). Par exemple, considérons la liste des éléments d'une matrice. Un premier vecteur peut en sélectionner les éléments de la première ligne, un deuxième vecteur en sélectionner les éléments de la première colonne, un troisième vecteur, les éléments non nuls de la diagonale principale et un quatrième, les éléments de la matrice supérieurs à un seuil donné. Une fois qu'un vecteur est construit à partir d'une liste d'éléments et un moyen de sélection, il peut être utilisé pour n'importe quel traitement, comme source ou comme cible. Chaque fois qu'un vecteur est utilisé, seuls les éléments sélectionnés sont utilisés dans les traitements ou modifiés quand le vecteur est utilisé comme cible d'une instruction.

Les instructions

Suivant le principe des primitives (ou "what vs. how", cf. § 2.1.1.), Devil a été conçu pour former un ensemble de primitives puissantes, tant pour le traitement scalaire que vectoriel. Une instruction se comportant comme une primitive plutôt que comme une instruction habituelle, la portabilité de Devil est grandement améliorée. Au lieu d'instructions qui correspondent de près aux instructions des machines cibles, les instructions Devil sont beaucoup plus générales que les instructions des machines cibles. Par exemple, l'instruction d'addition

```
add    a, b, c
```

réalise la somme des objets a et b et affecte sa valeur à l'objet c. Selon le type de ses opérandes, elle réalisera :

- l'addition de deux scalaires de types quelconques (non nécessairement le même type pour les deux). Le résultat est un scalaire ou un vecteur dont les éléments prennent alors tous la valeur du résultat de l'addition scalaire;
- l'addition d'un scalaire à tous les éléments d'un vecteur, chacun de type quelconque;
- l'addition de deux vecteurs de types quelconques, élément par élément.

Par ailleurs, une notion de fonction existe en Devil. Pour le passage d'argument, il y a des instructions pour indiquer explicitement le passage de paramètre soit en lecture seule, soit en lecture/écriture. Ces instructions ne s'intéressent pas à la présence ou non d'une pile de paramètres pour réaliser le passage de paramètres. La stratégie pour le passage effectif d'arguments est définie dans le traducteur.

Structure du langage

La représentation intermédiaire Devil ressemble à n'importe quel autre langage d'assemblage. Le jeu d'instructions est orthogonal. Ses instructions sont très puissantes par rapport aux instructions des cibles. Les modes d'adressages sont indiqués par la classe de l'opérande. Un moyen de spécifier le type des opérandes est nécessaire. Ce moyen est nommé un *attribut*. Chaque opérande possède un attribut. Un attribut est composé de plusieurs informations. Deux informations doivent être précisées et indiquent s'il s'agit d'un scalaire ou d'un vecteur ainsi que le type de l'objet. Pour les opérandes vectoriels, d'autres informations indiquent divers renseignements concernant le descripteur et la zone d'allocation, si elles sont connues à la compilation. Ces informations autorisent le traducteur à générer du code optimisé pour les accès vectoriels sur les supercalculateurs en éliminant de nombreux tests réalisés sinon lors de l'exécution. Le compilateur génère automatiquement ces attributs.

Le format des instructions de Devil

Les instructions Devil suivent le format général : `< op. > < sources > < cible >`. MAD n'est pas une machine à pile. MAD effectue tous ses traitements directement sur des objets. Malgré son usage habituel dans les langages intermédiaires (tels P-code et EM, cf. chapitre I), l'utilisation d'une pile pour l'évaluation d'expressions n'a pas d'avantage réel. L'optimisation de code est aussi compliquée pour une machine à pile que pour une machine n'en utilisant pas comme MAD (cf. [WULF 81]). De plus, il n'y pas de cible immédiate de MAD utilisant une telle pile. Plus généralement parmi les cibles potentielles de Devil, seuls les transputers utilisent une telle pile.

Les instructions de Devil peuvent se répartir en deux catégories. Il y a les instructions de traitements et les instructions de contrôle du flot d'instructions. Les premières effectuent les traitements sur les objets scalaires et vecteurs. Les secondes contrôlent l'exécution des programmes par MAD (sauts, ...).

Les temporaires

Devil donne une grande importance à la notion de temporaires (cf. code à trois adresses). Toute instruction de traitement peut entraîner soit la modification de la valeur d'un objet, soit la création d'un objet temporaire (ou avoir les deux effets). Typiquement, les objets temporaires sont des objets utilisés localement pour l'évaluation d'expressions. Ils doivent être utilisés (et ne peuvent être utilisés que) dans les fonctions où ils ont été produits. Ils sont écrits une seule fois et peuvent être lus plusieurs fois, bien qu'ils ne soient lus qu'une seule fois en général. L'utilisation des temporaires n'est en fait qu'une autre facette de la mise en application du principe des primitives. Il permet au compilateur de ne pas allouer les objets précisément et de laisser le traducteur les allouer au mieux, selon l'architecture cible. Ainsi, l'allocation des temporaires doit considérer les ressources matérielles effectivement disponibles (registres du CPU, mémoires caches, ...) pour autoriser un accès rapide à ces données éphémères.

La mémoire de MAD

Dérivant des concepts de mémoires (ou d'architectures) "taggées" [FEUSTEL 73] [GILOI & al. 77], nous avons conçu la mémoire de MAD comme une mémoire vectorielle, c'est à dire une mémoire contenant des scalaires et des vecteurs en tant qu'entités. Un accès scalaire lit ou écrit un objet scalaire. Un accès vectoriel lit ou écrit un objet vectoriel qui consiste en une séquence ordonnée de scalaires. La mémoire vectorielle réalise elle-même la sélection des éléments de la zone d'allocation d'un vecteur par le descripteur. Seuls les éléments sélectionnés sont réellement accédés.

Toujours en accord avec le principe des primitives, il n'y a pas de hiérarchie dans la mémoire de MAD. Toutes les données se trouvent en mémoire. La mémoire contient des scalaires et des vecteurs, en nombre virtuellement infini. L'unité de traitements effectue les calculs en chargeant ses données sources dans la mémoire et en y écrivant les résultats. MAD ne possède aucun registre afin d'y stocker les données à traiter ou les résultats. Cette caractéristique lui permet d'être totalement indépendante vis à vis du nombre et de la taille des registres scalaires ou vectoriels de ses cibles. Cette absence de hiérarchie de mémoire simplifie la définition de MAD. Le compilateur implante les objets dans la mémoire de MAD. Le traducteur s'occupe de l'implantation effective des segments de la mémoire de MAD. Il alloue les données dans les registres, dans les caches ou dans la mémoire centrale (ou ailleurs). Ainsi, les objets sont alloués de manière optimale quant aux contraintes espace utilisé/temps d'accès pour chacun.

L'unité de traitements de MAD

L'unité de traitements de MAD se compose d'unités fonctionnelles opératoires (celles qui traitent les données - addition, division, sinus, ...) et d'unités fonctionnelles de contrôle (répartiteur et initiateur d'instructions). Il existe virtuellement une infinité d'unités fonctionnelles opérationnelles de chaque type. Des constructions au niveau des programmes Devil autorisent le déclenchement en parallèle de plusieurs instructions. Le fait de doter la machine virtuelle de cette capacité d'exécution en parallèle d'instructions autorise le compilateur à se libérer totalement des problèmes d'obtention d'un parallélisme maximal à l'exécution. Du fait du nombre infini d'unités fonctionnelles, il n'a pas à se pré-occuper de la disponibilité des unités fonctionnelles pour le déclenchement des instructions.

1.2. Description de la machine MAD

La machine MAD est présentée dans cette section. Nous nous intéressons tout d'abord aux objets manipulés par MAD, les vecteurs et les scalaires, sous leur forme temporaire et non temporaire. La structure de la mémoire de MAD est ensuite décrite. L'allocation des objets par le compilateur dans la mémoire de MAD est discutée.

1.2.1. Présentation des objets traités par MAD

La machine MAD manipule des objets de deux catégories, les objets *scalaires* et les objets *vecteurs*. Le spectre de valeurs que peut prendre un objet (son type) est fixé lors de la définition de l'objet (à la compilation du programme). Ce type ne peut pas être modifié à l'exécution. Il existe quatre types pour les objets qui sont booléen, entier, flottant simple précision et flottant double précision.

Les scalaires

Les scalaires représentent les objets simples habituellement manipulés par les langages de programmation. Chaque scalaire possède une valeur. Un scalaire ne peut pas être de type booléen. Pour représenter une valeur scalaire booléenne, on utilise un scalaire entier. S'il a une valeur nulle, la condition est fausse. Sinon, la condition est vraie.

Les vecteurs

Les vecteurs sont habituellement vus comme des tableaux de scalaires. Cependant, il faut savoir dès maintenant que les vecteurs ne sont absolument pas implantés de cette manière. Tous les éléments d'un vecteur sont d'un même type, le type du vecteur. Les quatre types (booléen, entier, flottant simple et double précision) sont acceptés. Les éléments d'un vecteur peuvent être adressés un par un via une opération d'indexation, en partie (via l'extraction d'une tranche du vecteur, d'éléments répartis uniformément ou d'éléments dispersés) ou globalement. Dans le premier cas, on obtient un scalaire et on retrouve un traitement scalaire ordinaire. Dans les autres cas, on débouche sur le traitement vectoriel. Les éléments d'un vecteur sont ordonnés dans le sens où à chacun est associé un numéro d'ordre, son indice. Des opérations comme les additions, soustractions et produits scalaires de vecteurs sont possibles suivant leurs définitions mathématiques. D'autres opérations sont étendues aux vecteurs de manière non mathématique, mais plutôt comme application d'une même fonction sur l'ensemble des composantes du vecteur. Ainsi, le produit ou la division de deux vecteurs sont réalisés élément par élément. De même, les opérations trigonométriques acceptent un vecteur comme opérande source. Elles produisent alors un résultat vectoriel. L'opération est distribuée sur chaque composante de la source et produit la composante de même indice de la cible. Par essence, les vecteurs Devil sont mono-dimensionnels. Cependant, nous verrons plus loin que la représentation et la manipulation d'objets à deux, trois, ... n dimensions est tout à fait possible.

Les triplets

Pour des raisons d'efficacité, nous manipulons directement une catégorie de vecteurs entiers hybrides, les *triplets*, qui ont la forme :

[borne inférieure : borne supérieure : pas].

Ils sont fonctionnellement équivalents aux vecteurs d'entiers présentés précédemment : on peut toujours (syntaxiquement parlant) substituer un triplet par un vecteur d'entiers. Cependant, les triplets jouent un rôle important dans l'optimisation du code, de part leur représentation. Un triplet peut représenter n'importe quel vecteur d'entiers dont les éléments ont des valeurs successives ou séparées par un pas constant.

Le pas d'un triplet est entier non nul. Les bornes sont entières. Si le pas est positif, ce triplet est non vide si la borne inférieure est inférieure à la borne supérieure. Si le pas est négatif, ce triplet est non vide si la borne inférieure est supérieure à la borne supérieure. Les composantes a_i d'un triplet sont dans l'ordre :

$$a_0 = inf$$
$$a_1 = inf + pas$$

$$a_2 = inf + 2 \times pas$$
$$\ldots$$
$$a_n = inf + n \times pas$$

avec la condition $a_n \leq$ borne supérieure et a_n + pas > borne supérieure si le pas est positif,

$a_n \geq$ borne supérieure et a_n + pas < borne supérieure si le pas est négatif (inf représente la valeur de la borne inférieure).

L'utilisation de triplets est intéressante pour plusieurs raisons:

- trois entiers représentent un vecteur de taille quelconque. Le stockage et l'accès en mémoire (ou dans les registres des processeurs) des triplets est plus économique que celui des vecteurs représentés de manière étendue.
- certaines opérations arithmétiques sont des lois de composition internes à l'ensemble des triplets (addition, soustraction). Par ailleurs, ces opérations internes peuvent s'effectuer d'une façon économique au niveau temps de calcul (et sans encombrement mémoire supplémentaire). D'autres opérations combinant un triplet et un scalaire entier sont également réalisées d'une manière très économique (addition d'un entier à un triplet, multiplication d'un triplet par un entier par exemple).

1.2.2. Les vecteurs

Dans cette partie, nous présentons les vecteurs Devil, d'un point de vue fonctionnel tout d'abord, puis nous décrivons leur représentation dans la machine MAD.

Présentation fonctionnelle des vecteurs

Dans MAD, la structure de vecteur est beaucoup plus riche que celle définie pour des langages comme Fortran. Pour ces derniers, un vecteur est un simple tableau. Pour MAD, un vecteur est l'association de deux zones, la *zone d'allocation* et la *zone de description*. La zone d'allocation contient la liste des éléments du vecteur. La zone de description (ou encore *descripteur*) est le moyen de sélection des éléments effectifs du vecteur, c'est à dire des éléments faisant réellement parti du vecteur. Contrairement à une zone d'allocation, une zone de description ne peut pas être partagée entre plusieurs vecteurs. Le descripteur peut prendre quatre formes :

- vide (cf. fig. II.2.a) : tous les éléments de la zone d'allocation font réellement partie du vecteur. L'indice d'un élément effectif est le même que son indice dans la zone d'allocation;
- séquence d'index (cf. fig. II.2.b) : seuls les éléments de la zone d'allocation dont l'indice est présent dans la séquence sont réellement éléments du vecteur. Un élément de la zone d'allocation peut être non sélectionné, sélectionné une fois ou plusieurs fois;
- séquence de bits (cf. fig. II.2.c) : la séquence agit comme un masque sur les éléments de la zone d'allocation. Les éléments de la zone d'allocation correspondant à un 1 de la séquence de bits sont effectivement éléments du vecteur. Les autres ne sont pas éléments effectifs du vecteur;
- triplet (cf. fig. II.2.d) : le triplet est fonctionnellement équivalent à une séquence d'index, les éléments de la séquence d'index étant produits par le triplet. Cependant, l'accès aux éléments effectifs d'un vecteur est plus économique sur la plupart des machines si le descripteur est un triplet que s'il s'agit une séquence d'index.

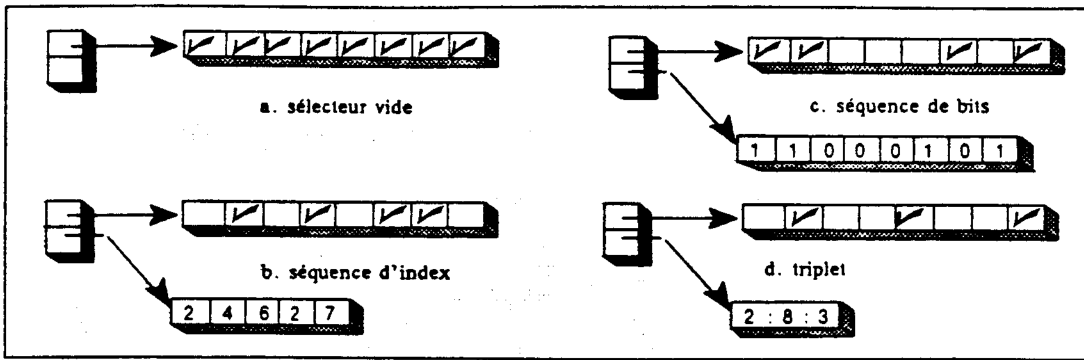


Figure II.2. -
Différents types de descripteurs de vecteurs dans MAD

Tous les vecteurs qui accèdent à une même zone d'allocation accèdent également à une même *zone d'informations* (cf. infra). Lorsqu'un lien est rompu entre un vecteur et une zone d'allocation, le lien correspondant entre le vecteur et la zone d'informations est rompu de la même manière. Cette zone contient tous les renseignements concernant la zone d'allocation associée. De cette manière l'information est unique et partagée entre tous les vecteurs accédant à la même zone d'allocation. Le maintien de la cohérence ne pose aucun problème. Les informations peuvent être mises à jour en passant par n'importe quel vecteur.

Concernant les problèmes de libération des zones d'allocation, nous avons simplement ajouté un compteur de références dans la zone d'informations. A chaque fois qu'un nouveau vecteur référence une zone d'allocation, son compteur de références est incrémenté. A chaque dé-référenciation, le compteur est décrémenté. Lorsqu'il atteint la valeur nulle, la zone d'allocation associée peut être libérée, ainsi que la zone d'informations.

Dans la suite de ce document, nous nommerons les *éléments d'un vecteur* les éléments de la zone d'allocation du vecteur en question sélectionnés par le descripteur du vecteur. Lorsqu'il s'agira des éléments de la zone d'allocation ou de la zone de description d'un vecteur, cela sera toujours dit explicitement.

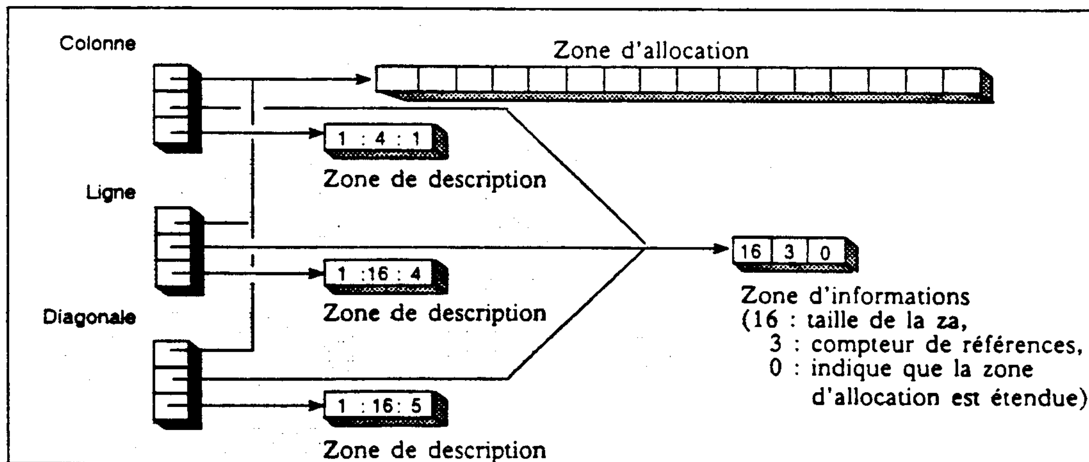


Figure II.3. -
Exemples de vecteurs se partageant la même zone d'allocation

Dans la figure II.3., la zone d'allocation représente les éléments d'une matrice 4 x 4. Ces éléments sont rangés colonne par colonne. Trois vecteurs référencent cette zone. La zone d'informations associée est également partagée entre ces trois vecteurs. Les trois vecteurs ont des

zones de description sous la forme de triplet. Le vecteur *colonne* accède les éléments de la première colonne du vecteur, le vecteur *ligne* ceux de la première ligne, le vecteur *diagonale* ceux de la diagonale principale.

La représentation des vecteurs

Un vecteur est uniquement manipulé via son *en-tête*. Celui-ci possède un format fixe (cf. fig. II.4.). Il se compose de six champs, za, zi, type, zd, lgth et tzd. Le rôle de chacun de ces champs est maintenant décrit. La structure de la zone d'informations est également décrite. Une zone d'informations se compose de trois champs, tza, cpt et za_sel.

❑ za : pointeur sur la zone d'allocation.

Le champ za contient une référence sur la zone d'allocation du vecteur.

❑ zi : pointeur sur la zone d'informations.

Une zone d'informations se compose des champs :

- tza qui contient le nombre d'éléments de la zone d'allocation. Cette valeur est fixe et n'est pas modifiée lors de l'exécution du programme;
- cpt qui contient le nombre de vecteurs se partageant la zone d'allocation, et donc se partageant également la zone d'informations;
- za_sel qui contient un indicateur binaire indiquant si la zone d'allocation est sous forme étendue ou sous forme de triplet. Seul un vecteur d'entiers peut posséder une zone d'allocation sous forme de triplet.

❑ type : type du descripteur.

Ce champ indique le type du descripteur du vecteur. Le champ peut prendre les valeurs vide, liste d'index, liste de bits ou triplet.

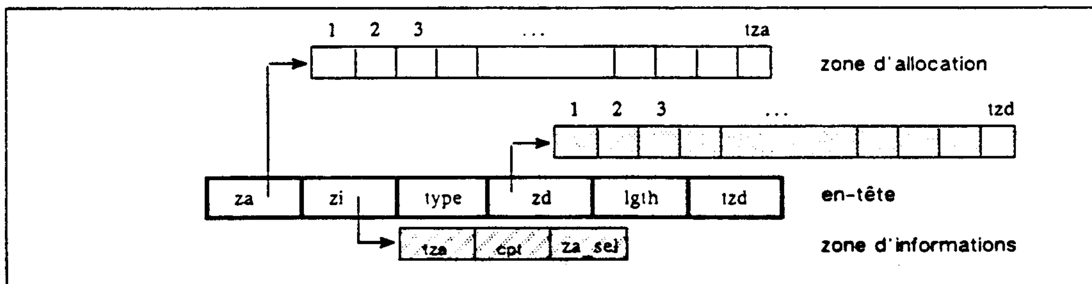


Figure II.4. –
Structure d'un en-tête de vecteur(9)

❑ zd : pointeur sur la zone de description.

Selon le type du descripteur, ce champ contient :

- une référence invalide (pointeur nul) si le *type* est vide;

(9) - Dans la suite de ce document, l'en-tête d'un vecteur sera représenté avec des traits gras, la zone d'informations hachurée, la zone de description avec un fond grisé et la zone d'allocation avec un fond blanc.

- une référence à une table d'entiers si le type est *liste d'index*;
- une référence à une table de bits si le type est *liste de bits*;
- une référence à un triplet d'entiers si le *type* est *triplet*

❑ lgth : nombre d'éléments effectifs du vecteur.

Le champ lgth contient le nombre d'éléments effectifs du vecteur, c'est à dire le nombre d'éléments de la zone d'allocation du vecteur sélectionnés par le descripteur. Selon le type du descripteur du vecteur, ce champ vaut :

- la valeur du champ tza de la zone d'information si le vecteur n'a pas de description;
- la valeur du champ tzd si le descripteur est une liste d'entiers;
- le nombre de bits à 1 si le descripteur est une liste de bits;
- la valeur de ((borne supérieure - borne inférieure + pas) / pas) si le descripteur est un triplet (0 si ce nombre est négatif).

❑ tzd : nombre d'éléments du descripteur.

Le champ tzd contient le nombre de composantes du descripteur. Selon le type du descripteur, ce champ vaut :

- 0 s'il n'y a pas de descripteur;
- le nombre d'éléments de la liste si le descripteur est une liste d'index ou une liste de bits;
- ((borne supérieure - borne inférieure + pas) / pas) si le descripteur est un triplet (0 si ce nombre est négatif).

Selon le type du descripteur, l'accès aux éléments d'un vecteur peut être un accès contigu (*type* vaut *vide* ou si la zone d'allocation est un triplet avec pas de un). Pour un accès dispersé, il nécessite une opération de rassemblement ou d'éclatement (*type* vaut *liste d'index*), une opération de compression ou d'extension (*type* vaut *liste de bits*) ou un accès dispersé à pas constant (*type* vaut *triplet* avec un pas différent de un).

Les éléments effectifs du vecteur de la figure II.5. sont : 10, 20, 50, 40, 20, 50, 20. Remarquons que bien que les éléments 20 et 50 ne soient qu'une seule fois présents dans la zone d'allocation, ils sont plusieurs fois éléments du vecteur (ceci parce qu'ils sont plusieurs fois sélectionnés par le descripteur). Cette apparition de doublons n'est possible que si le descripteur est de type liste d'index.

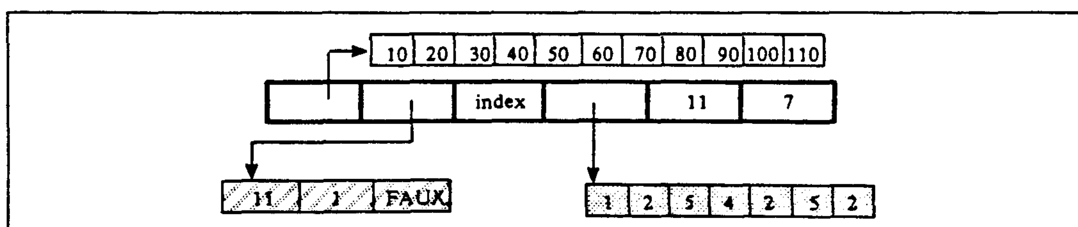


Figure II.5. -
Exemple de vecteur

1.2.3. Les temporaires

Les données temporaires existent en nombre virtuellement infini. Elles sont allouées lors de la traduction en code cible (le traducteur s'arrangera alors pour les placer dans des registres ou des

caches pour y avoir un accès aussi rapide que possible). Notons que cette allocation dans les registres ne doit pas être faite plus tôt car le compilateur génère du code indépendant par rapport à la cible. Les données temporaires sont des résultats partiels de l'évaluation d'une expression. Le compilateur génère également des temporaires pour la traduction de certaines structures de contrôle.

Un temporaire n'est jamais nommé; il est référencé via le numéro de l'instruction l'ayant calculé, ce numéro étant relatif à l'instruction l'utilisant. Exemple :

```
x := a * (b * c - d / e)
```

se traduit en "pseudo-Devil" (*pseudo* car tous les modes d'adressages ne sont pas explicités – par ailleurs, les objets ne sont pas réellement nommés mais référencés via leur adresse dans la mémoire de MAD) :

```
si si        mul      b, c
si si        div      d, e
si si        sub      &2, &1
si si si   n.mul      a, &1, x
```

La notation *&x* référence le temporaire résultant de la xème instruction précédente.

Les attributs (ils valent tous `si` dans cet exemple) indiquent que les opérandes sont des objets scalaires entiers. Le préfixe `n.` indique que l'instruction en question ne génère pas de résultat temporaire.

Les instructions `mul`, `div`, `add`, ..., à deux opérandes, produisent un résultat temporaire. Lorsqu'un troisième opérande est spécifié, celui-ci est pris comme cible. Par ailleurs, un temporaire peut également être généré (en plus du résultat mis dans la cible). La génération, ou non, d'un temporaire est spécifiée dans le code de l'instruction.

Il existe deux types de vecteurs temporaires : les vecteurs temporaires étendus et les triplets. La représentation d'un vecteur temporaire est différente de celle d'un vecteur non temporaire. Il ne possède pas d'en-tête. Un vecteur temporaire se compose uniquement d'une zone d'allocation contenant ses éléments et d'un champ contenant le nombre d'éléments que peut contenir sa zone d'allocation (cf. fig. II.6.). Ainsi, un vecteur temporaire étendu n'est jamais associé à un descripteur : c'est toujours un vecteur contigu.

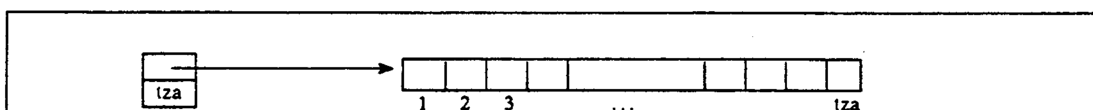


Figure II.6. –
Structure d'un vecteur temporaire

Un triplet est représenté par un triplet de scalaires entiers : < début, fin, pas >. Il représente le vecteur dont les éléments sont :

```
[ début, début + pas, ..., début + n * pas ] où n est tel que
(début + n * pas) <= fin < (début + (n + 1) * pas) pour pas > 0,
(début + n * pas) >= fin > (début + (n + 1) * pas) pour pas < 0
```

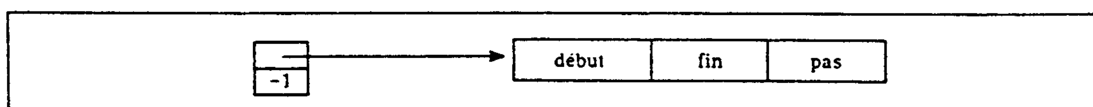


Figure II.7. –
Structure d'un triplet

Un triplet temporaire possède la même représentation qu'un vecteur temporaire étendu (cf. fig. II.7.), excepté l'indicateur -1 qui remplace la taille de la zone d'allocation (de taille fixe pour un triplet : 3) et qui informe qu'il s'agit d'un triplet.

Un temporaire est mis à valeur par une instruction. Sa valeur peut ensuite être lue plusieurs fois. C'est un objet à assignation unique. L'adressage d'un temporaire (en tant qu'opérande source d'une instruction) indique s'il s'agit de sa dernière utilisation ou non (référencé par ^ s'il sera ré-utilisé, par & sinon). Ceci permet de simplifier la gestion de la mémoire : après sa dernière utilisation, l'espace mémoire réservé à ce temporaire peut être libéré. Si sa dernière utilisation effective ne peut être indiquée, une pseudo-instruction (reltmp) est utilisée. Ainsi, l'utilisation d'un temporaire dans une boucle mais créé en dehors de la boucle nécessite ce mécanisme :

```
    < création du temporaire >
boucle:
    < instructions >
    < utilisation du temporaire -> adressage via ^nnn >
    < instructions' >
    < contrôle de la boucle >
    n.reltmp  &xxx
```

1.2.4. Les registres de MAD

MAD possède huit registres internes lui permettant de connaître et de contrôler son état. Elle ne possède pas de registre d'état. Les conditions sont des scalaires entiers positionnés par les instructions de comparaison, testés par les instructions de branchements conditionnels. Les registres de MAD sont :

PC : compteur ordinal. Sa valeur indique l'adresse de l'instruction suivante à exécuter dans le segment de code. Elle est incrémentée à chaque chargement d'une nouvelle instruction. Elle est mise à jour lors de l'exécution d'une instruction de saut, d'appel et de retour de fonction;

SP : pointeur de sommet de pile. Sa valeur indique le premier emplacement libre dans la pile de contextes (segment *context*). Sa valeur est mise à jour automatiquement lors de l'exécution d'instructions nécessitant une sauvegarde ou une restauration de contexte, c'est à dire les instructions d'appel ou de retour de fonction;

DBR : adresse de base du segment de données (segment *data*). Sa valeur est mise à jour automatiquement à l'entrée des fonctions;

LBR : adresse de base du segment de données locales à une fonction (segment *local*). Sa valeur est mise à jour automatiquement à l'entrée et à la sortie d'une fonction;

PAR : adresse de base du segment des paramètres d'appel d'une fonction (segment *param_rec*). Sa valeur est mise à jour automatiquement à l'entrée et à la sortie d'une fonction;

CALL : adresse de base du segment des paramètres des fonctions appelées par la fonction en cours d'exécution (segment *param_to_send*). Sa valeur est mise à jour automatiquement à l'entrée et à la sortie d'une fonction;

VLG : registre de longueur des vecteurs. Sa valeur indique le nombre d'éléments des vecteurs sur lesquels opèrent la plupart des instructions vectorielles (opérations arithmétiques par exemple). Sa valeur est modifiée par des instructions spéciales de gestion de ce registre.

VSP : pointeur de sommet de pile VLG (cette pile est localisée dans le segment *vlg*). Sa valeur est automatiquement mise à jour lors de la modification de la valeur du registre VLG.

Aucune hypothèse n'est établie concernant la taille de ces registres. Ainsi, aucune restriction sur la taille des programmes exécutés (PC), de la pile (SP) ou de la taille des vecteurs (VLG) n'existe dans MAD.

MAD ne possède pas de registre de travail. Tous les calculs se font sur des objets.

1.2.5. Le modèle mémoire de MAD

Les segments de la mémoire

MAD adresse ses données dans divers segments : segment de code, de données, de pile, de tas et de temporaires. Un nombre non limité de segments d'objets externes (commons) sont disponibles. Les segments sont référencés soit via des registres de base, soit via leur nom (*code*, *context*, *vlg*, ...). Les différents segments sont (cf. fig. II.8.) :

- le segment de code (*code*) est adressé directement, et exclusivement, via le registre PC. Ce segment est référencé par son nom;
- le segment de données (*data*) est adressé aléatoirement. Il y a deux genres de données : les données *globales* et les données *statiques*. Les premières sont locales au module où elles sont déclarées. Les secondes sont des objets rémanents aux fonctions. Elles sont uniquement visibles à l'intérieur des fonctions où elles sont déclarées et gardent leur valeur d'un appel au suivant. Toutes les fonctions d'un module se partagent le même segment de données. Les segments de données (un par module) sont alloués une fois, au lancement du programme. Ce segment est référencé via le registre DBR;
- le segment contenant les valeurs des paramètres d'appel (*param_rec*) de la fonction courante. Ce segment contient des références d'objets et non les valeurs des objets. Ce segment est adressé via le registre de base PAR;
- le segment des paramètres d'appel (*param_to_send*) des fonctions appelées par la fonction en cours d'exécution. Lorsqu'un appel de fonction est réalisé, ce segment devient le segment *param_rec* de la fonction appelée. Ce segment contient des références et des valeurs d'objets (pour l'interfaçage avec des fonctions C uniquement). Ce segment est adressé via le registre de base CALL;
- le segment des variables locales (*local*) contient les variables locales de la fonction en cours d'exécution. Un nouveau segment *local* est alloué à l'appel (c'est à dire à son activation par une instruction `call`) d'une fonction, libéré à son retour (c'est à dire à sa désactivation par exécution d'une instruction `return` ou `endfct`). Ce segment est adressé via le registre de base LBR;
- le segment *context* fonctionne comme une pile accédée via le registre SP. Le contexte des fonctions appelantes (c'est à dire la valeur courante des registres PC, DBR, LBR, PAR et VSP) est sauvegardé dans ce segment lors d'un appel de fonction (instruction `call`). Il est restauré lors du retour dans la fonction (après exécution par l'appelé d'une instruction `return` ou `endfct`). Ce segment est référencé par son nom;
- le segment *vlg* fonctionne comme une pile. Cette pile est associée au registre VLG. Il contient les anciennes valeurs du registre VLG, masquées localement par une modification de sa valeur. La valeur en sommet de pile est restaurée dans le registre VLG lorsque la valeur courante de VLG est invalidée. Le registre VSP pointe le sommet de la pile *vlg*. Ce segment est référencé par son nom;

- le segment de tas (*heap*) est adressé implicitement lors de manipulations d'objets par MAD. Il n'y a pas de mode d'adressage pour accéder directement une donnée se trouvant sur le tas. Il n'y a pas d'instruction d'allocation ou libération de zones dans le tas. Elles sont réalisées implicitement lors de la manipulation de vecteurs. Ce segment est référencé par son nom;

- le segment des temporaires (*seg_tmp*) contient les objets temporaires. C'est un segment à accès associatif composé de couples

  (< numéro instruction > < valeur du temporaire produit par cette instruction >).

  Un temporaire est accédé via le numéro de l'instruction l'ayant généré. Ce segment est référencé par son nom;

- les segments de commons et externes contiennent les variables partageables entre plusieurs modules. Chacun est référencé par son nom. Ce nom est celui de la variable externe ou du bloc common lequel peut contenir plusieurs variables externes. Pour adresser les données d'un segment de common, l'offset de l'objet dans le segment est spécifié.

Dans MAD, les segments ne sont jamais explicitement localisés dans la mémoire physique d'une cible. Ils sont accessibles, un point c'est tout. Toutes les considérations concernant l'implantation effective des données dépendent d'une instanciation particulière de MAD.

Hormis les segments de paramètres (*param_to_send* et *param_rec*), tous les segments contiennent les valeurs des objets. Ces segments ne contiennent jamais de référence à des objets. De ce point de vue, les segments de paramètres font exception puisqu'ils peuvent contenir des valeurs ou des références d'objets.

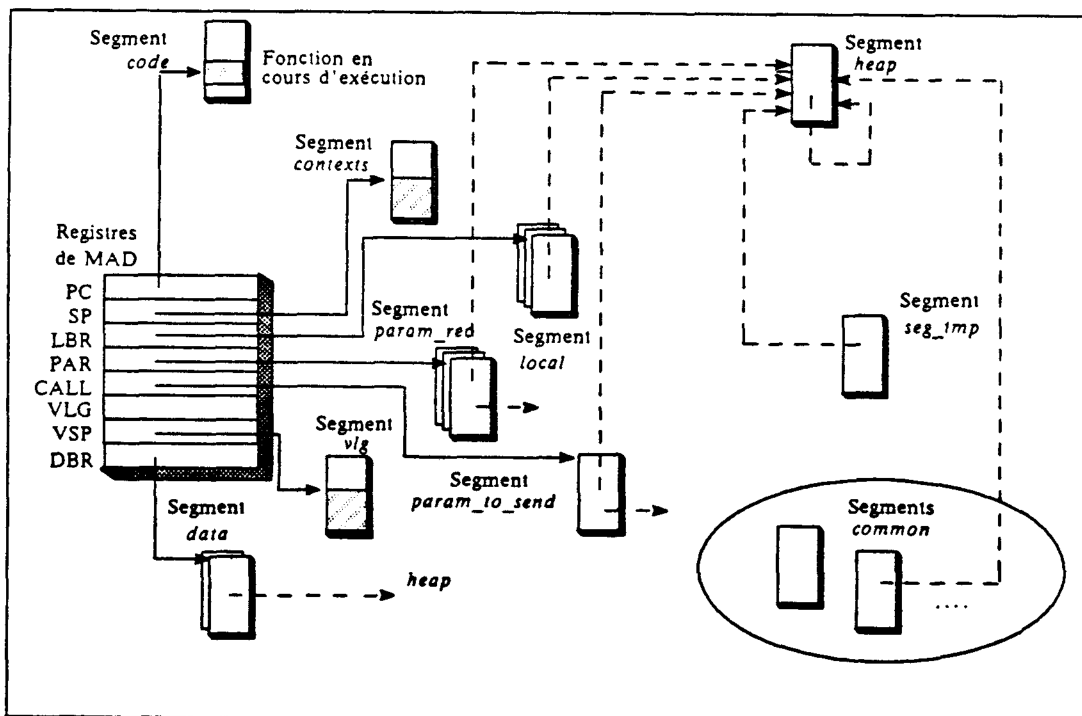


Figure II.8. -
Le modèle mémoire de MAD

Les segments *data*, *param_to_send*, *param_rec*, *local*, *seg_tmp*, *heap*, *common* peuvent contenir des en-têtes de vecteurs référençant des zones mémoires appartenant au segment *heap* (lignes discontinues dans la fig. II.8.).

1.2.6. Allocation des objets en mémoire

Les objets scalaires et vecteurs non temporaires sont alloués à la compilation dans les segments de MAD. Ils sont ensuite référencés dans le programme Devil par leur adresse virtuelle composée du nom du segment où ils se trouvent et d'un offset dans le segment. Dans un programme Devil, les tailles du segment de données et des segments de commons du module sont déclarées et chaque fonction définit la taille de ses segments *local* et *param_to_send*. Les segments *context*, *vlg*, *heap* et *seg_tmp* croissent dynamiquement à l'exécution.

A la traduction en assembleur cible, les segments de MAD sont projetés sur la mémoire de la cible, les adresses des objets se déduisant de cette projection.

1.3. Fonctionnement de la machine MAD

Dans cette section, nous présentons différents points du fonctionnement de MAD. Une large part est dédiée à l'appel de fonctions et les passages de paramètres sont également discutés.

1.3.1. Longueur des opérandes vectoriels

En général, une instruction vectorielle agit sur les VLG premiers éléments des vecteurs. Par ailleurs, le registre VLG est géré d'une façon assez particulière : au registre VLG est associée une pile de valeurs (segment *vlg*). Les valeurs de VLG y sont sauvegardées puis restaurées. Cette caractéristique autorise une gestion simplifiée de VLG : au cours de l'évaluation d'une expression vectorielle sur une longueur VLG_1, on peut facilement évaluer une sous-expression sur une longueur VLG_2 puis restaurer la valeur précédente (VLG_1) pour poursuivre le calcul, sans avoir à mémoriser explicitement VLG_1. Un offset (dont la valeur est contenue dans le registre VSP) dans le segment *vlg* précise le sommet de la pile.

1.3.2. Les instructions décrites

Reflétant les opérations masquées disponibles sur certaines machines, les opérations vectorielles peuvent être effectuées sous le contrôle d'un descripteur par MAD. Ces opérations sont alors dites *décrites*. La description est un vecteur d'index, un vecteur de bits, un triplet ou un scalaire entier. Seules les composantes du vecteur cible sélectionnées par le descripteur sont affectées avec le résultat de l'opération.

1.3.3. Les fonctions

Appels et retours de fonction

A l'appel d'une fonction, la valeur courante de VLG est empilée dans la pile *vlg*. Le contexte de l'appelant (contenu des registres PC, VSP, LBR, PAR, DBR) est sauvegardé dans le segment *context*. Puis, la valeur du registre CALL est copiée dans le registre PAR. Par ce biais, le segment *param_to_send* de l'appelant devient le segment *param_rec* de l'appelé. Le registre PC reçoit l'adresse de la première instruction de la fonction appelée.

La première instruction de la fonction alloue un segment pour les variables locales (le segment *local*) et un segment *param_to_send*. Les registres de base LBR et CALL sont initialisés respectivement avec les adresses de ces segments. Le registre de base DBR est également mis à jour.

Le code de l'appelée s'exécute.

La dernière instruction de l'appelé désactive la fonction courante. Elle désalloue le segment des locaux et le segment *param_to_send*. La valeur du registre PAR est copiée dans le registre CALL. Le contexte est restauré (donc, la valeur de VLG est restaurée). Le registre PC reprend ainsi sa valeur, dans le code de l'appelant. Les registres référencent donc à nouveau les segments de la fonction appelante. Son exécution est reprise.

Notons pour être complet, qu'au début de l'exécution d'une fonction, une routine spécifique d'initialisation des données locales au module où se trouve la fonction, est appelée si cette initialisation n'a pas déjà eu lieu auparavant.

Paramètres des fonctions

Passages de paramètres

Le mode de passage de paramètres entre fonctions Devil est le passage par référence. On distingue alors les modes de passages en lecture/écriture (`parout`) et en-tête constant (`parconst`). Pour permettre l'appel de fonctions C depuis Devil, il est possible de passer un paramètre en lecture seule (`parin`). Cependant, une fonction Devil ne peut en aucun cas récupérer un paramètre passé de cette manière. Par ailleurs, quel que soit le mode de passage des paramètres à une fonction Devil, celle-ci peut effectuer toute modification sur l'objet. Selon le mode, ces modifications seront visibles ou non de la fonction appelante. Nous discutons d'abord des modes de passages Devil qui sont classiques. Le mode de passage en lecture seule est présenté ensuite.

Pour un objet scalaire, les deux modes de passage de paramètres sont équivalents. Une référence sur sa valeur est placée dans le segment *param_to_send*. Sa valeur peut être modifiée par la fonction appelée. Les modifications restent valables au retour dans la fonction appelante.

Pour un objet vecteur, le passage en mode lecture/écriture est réalisé en plaçant une référence à l'objet dans le segment *param_to_send*. Sa valeur peut donc être changée par la fonction appelée, les éléments de la zone changés, ou les valeurs des champs de son en-tête modifiées par association d'une nouvelle zone d'allocation et d'une nouvelle zone de description. Le passage en mode en-tête constant n'autorise pas la prise en compte par l'appelant des éventuelles modifications sur les champs de l'en-tête réalisées par la fonction appelée. Pour cela, une copie de l'en-tête du vecteur à passer en paramètre est réalisée. Une référence sur cette copie est alors placée dans le segment *param_to_send*. Notons qu'après cette copie, il y a un partage de la zone d'allocation entre les deux vecteurs. Aussi, le compteur de références contenu dans la zone d'informations est incrémenté.

Par souci d'orthogonalité, mais aussi pour simplifier l'écriture de programmes en Devil, les objets temporaires peuvent également être passés en paramètre. Cependant, puisque ce sont des objets à assignation unique, les modifications que pourrait effectuer la fonction appelée sur leur valeur ne seront pas visibles au retour dans la fonction appelante. Aussi, pour un scalaire, une référence à une copie de la valeur du scalaire est placée dans le segment *param_to_send*. Pour un vecteur temporaire, un en-tête lui est alloué dans le segment *param_to_send*. Sa zone d'allocation est alors une copie de la zone du temporaire. De cette manière, la fonction appelée reçoit toujours un en-tête de vecteur quel que soit la nature (temporaire ou non) du vecteur passé effectivement en paramètre par la fonction appelante. Dans le cas où le passage en paramètre est la dernière utilisation du temporaire (la majorité des cas), des optimisations pourront être effectuées, sa valeur n'ayant pas alors à être dupliquée.

Les littéraux scalaires et vecteurs peuvent également être passés en paramètre. Pour éviter tout effet de bord (cf. passage de constantes en paramètre en Fortran), une référence à une copie de la valeur de l'objet est placée dans le segment *param_to_send*.

Interface Devil-Fortran

Les passages de paramètres en Fortran sont réalisés par référence. Aussi, les passages de paramètres en mode `parout` et `parconst` supportent l'appel d'un sous-programme ou d'une fonction

Fortran depuis Devil. De même, l'appel avec passage de paramètres d'une fonction Devil depuis Fortran ne pose pas de problème.

Interface Devil-C

Les passage de paramètres à une fonction C s'effectuent par valeur. Les paramètres `parout` et `parconst` sont récupérés normalement par une fonction C via leur référence. Par ailleurs, un mode de passage de paramètres en lecture (mode `parin`) a été ajouté. La fonction C récupère alors la valeur d'un objet (scalaire ou vecteur). Un objet passé selon ce mode est dupliqué et c'est cette copie (l'en-tête de la copie pour un vecteur) qui est passée dans le segment *param_to_send*.

Le retour de fonction avec renvoi de valeur

Une fonction Devil peut renvoyer la valeur d'un objet quelconque (de n'importe quel type, global, local, temporaire ou autre). Un objet temporaire reçoit la valeur de l'objet à renvoyer. C'est ensuite la référence de cet objet temporaire qui est passée à la fonction appelante comme valeur de retour de la fonction. L'instruction d'appel de la fonction peut affecter la valeur de cet objet à sa cible ou donner un résultat temporaire, l'objet renvoyé lui-même. Dans l'exemple qui suit,

```
           Corps de la fonction appelante              Corps de la fonction appelée

              < passage de paramètres >              f:
                   call     f                                < corps de la fonction f >
        vi vi vi  n.add     &1, b, c                 vi      return    objet
```

une fonction `f` est appelée. Elle renvoit un vecteur d'entier temporaire. Celui-ci est ensuite utilisé dans une instruction `add`, comme n'importe quel autre objet temporaire.

Notons qu'un vecteur renvoyé par une fonction est forcément un vecteur contigu, car temporaire. Par ailleurs, si l'objet renvoyé par la fonction est un objet temporaire, sa valeur n'a pas à être recopiée pour être renvoyée. L'objet lui-même peut être renvoyé directement.

1.3.5. Les branchements conditionnels

Sur de nombreux micro-processeurs (Motorola 680x0 et Intel 80x86 par exemple), les branchements conditionnels utilisent les bits conditions d'un registre d'état du processeur. Une instruction de branchement conditionnel teste un de ces bits ou une fonction sur plusieurs de ces bits. A une instruction correspond un test donné. Selon le résultat de ce test (bit à 0 ou à 1), le processeur effectue un branchement à une adresse spécifiée dans l'instruction ou continue en séquence.

Les instructions de branchements conditionnels de MAD fonctionnent selon un autre principe. MAD ne possède pas de registre d'état. Il existe deux instructions de branchement conditionnel. La condition à tester est l'un des opérandes de l'instruction, l'autre opérande indique l'étiquette de branchement. L'une des deux instructions réalise le branchement si la condition est vérifiée, l'autre si la condition n'est pas vérifiée. La condition est un objet scalaire entier. Elle est vérifiée si l'objet possède une valeur non nulle, non vérifiée si elle est nulle. Typiquement, une condition est mise à valeur par une instruction de comparaison. Ce peut être également le résultat de n'importe laquelle des instructions, dès lors que son résultat est scalaire entier. Par exemple, pour réaliser un branchement si le résultat d'une addition scalaire est nul, il suffit de passer le résultat de l'addition en opérande de l'instruction de branchement conditionnel.

```
si si     add a, b
        n.fbranch   &1, res_nul
                < résultat de l'addition non nul >
                ...
res_nul:
                < résultat de l'addition nul >
                ...
```

1.4. Le langage Devil

Le langage intermédiaire Devil est présenté dans cette section. Nous ne nous intéressons pas ici à une description ponctuelle de chacune de ses instructions. Celles-ci sont décrites in extenso dans [PREUX 90]. Une présentation rapide en est donnée en annexe. Celle-ci vise à éclaircir les exemples de code Devil qui sont donnés.

1.4.1. Définition du langage Devil

Le langage Devil offre un outil de spécification d'algorithmes vectoriels. Certains de ses aspects lui donnent un caractère proche des langages de haut niveau autorisant l'expression du parallélisme de type vectoriel. Parmi ceux-ci, on compte :

- puissance des instructions de base,
- manipulation d'objets (les vecteurs),
- définition de structures de haut niveau (fonctions et passages de paramètres).

D'autres aspects lui confèrent un aspect plus primitif :

- les expressions de haut niveau sont décomposées en séquences d'actions primitives,
- il n'y a pas de structures évoluées de contrôle du flot d'instructions (boucles),
- les objets sont manipulés via leurs adresses (virtuelles, dans la mémoire de MAD).

Les caractères de haut niveau permettent de rendre Devil indépendant de son implantation sur une architecture donnée. Ceux de bas niveau permettent de s'attaquer directement au problème de fond qu'est la génération de code. En effet, tout sucre syntaxique étant absent d'un programme Devil, il n'y a plus de problème d'analyse syntaxique de source, mais seulement des problèmes de génération de code optimal.

Caractérisation du langage Devil

En reprenant la classification des langages intermédiaires établie au premier chapitre, Devil est un langage intermédiaire RISC : il comporte peu d'instructions (une centaine), pas de hiérarchie de mémoire, pas d'adressage complexe (on dispose des adressages immédiat ou segmenté direct – connaissant la classe d'un objet, on peut l'accéder facilement), ses instructions sont généralement orthogonales. Aussi, Devil se rapproche beaucoup plus des machines-intersections que des machines-unions. Rappelons que l'un des objectifs du concept RISC (plus classiquement utilisé pour les architectures mais étendu ici aux langages) est de simplifier l'écriture de compilateurs. L'écriture du compilateur générant du Devil (EVA ou un autre langage de haut niveau) en est simplifiée.

Devil possède des aspects qui rappellent les langages de quadruplets, alors que d'autres rejoignent ceux des langages de triplets. Ainsi, les opérations ne donnant pas un résultat temporaire ont la forme de quadruplets :

```
a := b + c
```

qui se traduit en pseudo-Devil par :

```
n.add  b, c, a
```

Par contre, les instructions qui produisent un résultat temporaire ont la forme des triplets. Le temporaire résultat n'est pas nommé et il est référencé par la suite via le numéro de l'instruction

l'ayant calculé. Notons que d'autres instructions mixent les deux fonctionnements, une instruction pouvant générer à la fois un résultat dans un objet et produire un temporaire. Ainsi, l'expression (en EVA, ou C si l'on remplace alors := par =) :

```
a := (b := c + d) + 1
```

se traduit en pseudo-Devil par :

```
  add    c, d, b
n.add    &1, #1, a
```

La première instruction add calcule la valeur de b et donne un résultat temporaire. A celui-ci est ensuite ajouté la valeur 1 pour donner la valeur à affecter à *a*. Cette possibilité simplifie l'écriture du code généré qui, en l'absence de celle-ci, se serait écrit :

```
  add    c, d
n.move   ^1, b
n.add    &2, #1, a
```

La notation ^1 indique que le temporaire en question sera ré-utilisé par la suite (référencé par &2 dans l'instruction suivante).

Remarques quant à l'indépendance d'un source Devil par rapport à la machine cible

Un programme Devil n'est pas totalement indépendant de la cible. En effet, il est nécessaire de connaître quelques renseignements sur la cible pour écrire un programme Devil. Ces renseignements indiquent la taille des mots et des données de base (entiers, flottants) pour la cible. Cependant, ces informations sont utilisées de telle manière qu'il est facile de rendre un programme Devil indépendant de toute cible. Ainsi, un jeu de macros ayant pour valeurs les informations citées plus haut suffiront pour rendre son indépendance à un source Devil.

1.4.2. Principes de conception du jeu d'instructions

A la conception de Devil, nous nous sommes appuyés sur un ensemble de principes reconnus comme bons pour la définition de langages intermédiaires. Un tour d'horizon est réalisé sur ces principes. Leur mise en application simultanée se révèlant contradictoire, il faut essayer d'utiliser tel ou tel principe à bon escient et de manière adéquate.

[AHO & al. 89] indiquent, avant tout, deux avantages pour l'utilisation d'un langage intermédiaire :

- reciblage du compilateur simplifié. Seule la partie finale du compilateur est à réécrire. L'utilisation de la technique classique de *bootstrap* réduit cette partie à réécrire à peu de choses;
- possibilité d'optimisation du code intermédiaire indépendamment de la cible.

La complexité du matériel ne cesse d'augmenter. Ceci mène à des architectures de plus en plus puissantes. Les instructions du langage machine sont de plus en plus évoluées (au regard des modes d'adressages, types de données et des fonctionnalités des instructions). Pour qu'un langage de programmation (intermédiaire ou non) ait un intérêt sur une machine donnée, il faut qu'il soit d'un niveau plus élevé que celui du langage machine. Le fait qu'un langage donné soit de plus haut niveau qu'un langage machine entraîne qu'une instruction de ce langage se traduit en une séquence d'instructions machines. Au contraire, si le niveau du langage est inférieur à celui du langage machine, plusieurs instructions du langage se traduisent en une seule instruction du langage machine. Ce phénomène a été observé dans [ANCONA & al. 89] qui comparent le P-code avec les microprocesseurs 32 bits MC 68020, NS 32000 et i80286. Les auteurs montrent que certaines instructions du P-code sont de niveau inférieur aux instructions de ces microprocesseurs.

[NEWEY & al. 72] indiquent que trois points sont à considérer à la conception d'un langage intermédiaire :

- l'adéquation entre les spécificités de la machine virtuelle et les algorithmes à exécuter.

Dans notre cas, l'application de ce principe à mené à la définition d'instructions vectorielles au niveau de Devil, plutôt qu'à l'utilisation d'un langage intermédiaire scalaire, vectorisé ensuite.

- L'adéquation du langage de la machine virtuelle aux architectures existantes.

Ceci a mené à la définition de MAD et de son modèle d'exécution des instructions.

- Les limitations imposées par les outils utilisés pour la traduction.

[KORNERUP & al. 80] indiquent qu'il ne faut pas perdre, à la génération du code intermédiaire, d'informations pouvant mener à une meilleure qualité du code généré par le traducteur. Les auteurs indiquent des problèmes dus à l'utilisation du P-code quant à l'allocation en mémoire des objets par le compilateur : « *l'allocation ne devrait pas être réalisée par le compilateur. La forme intermédiaire devrait contenir toutes les informations de déclaration nécessaires reflétant celles du programme source* ». Les auteurs font la même remarque quant à l'allocation des registres. Cette remarque se rapproche du principe des primitives où le compilateur exprime ce qu'il y a à faire, et non comment, le "comment" idéal pour une structure donnée dépendant de la cible.

[IBSEN 84] (suivant en cela [KORNERUP & al. 80]), travaillant à la définition d'un langage intermédiaire (le A-code et sa A-machine) pour le langage Ada, indique que sa définition du A-code n'est pas tout à fait adéquate pour en réaliser la compilation : trop d'informations sur la structure de haut niveau du programme sont perdues.

[BAL & al. 86] indiquent qu'une représentation intermédiaire de haut niveau (arbre) est inadéquate pour la plupart des optimisations. Les arbres évitent l'analyse du flot de contrôle (puisqu'il se trouve implicitement dans la structure de l'arbre) mais les sauts inconditionnels posent des problèmes de représentation.

Les principes de WULF

[WULF 81] indique huit principes à suivre pour améliorer l'adéquation entre compilateurs et architectures cibles (au niveau de la définition d'un langage intermédiaire ou d'un langage d'assemblage pour un processeur) :

- *régularité* : si quelque chose est réalisé d'une certaine manière à un moment, cette chose doit toujours être réalisée de la même manière;

- *orthogonalité*(10) : la définition d'une machine ou d'un langage doit pouvoir constituer une partition et chaque partie doit être définie indépendamment des autres. Par exemple, les définitions des types de données, des adressages et du jeu d'instructions doivent être indépendantes;

- *composabilité* : les principes de régularité et d'orthogonalité étant respectés, il doit être alors possible de composer ces définitions régulières et orthogonales de n'importe quelle manière. Ainsi, on doit pouvoir utiliser n'importe quel type de données avec n'importe quel mode d'adressage, ...;

Ce principe est difficile à appliquer. Par exemple les registres d'un processeur sont rarement banalisés (cf. registres de données et registres d'adresses). Aussi, une instruction n'acepte pas

(10) - La définition que donne Wulf de la notion d'orthogonalité ne correspond pas à l'usage commun. Il nomme la notion classique d'orthogonalité la composabilité.
C'est dans son acception classique que nous utiliserons par la suite le terme orthogonalité.

tous les types de registre en opérande (l'instruction d'addition de deux registres de données est différente de l'instruction d'addition de deux registres d'adresses).

- *unicité* : il doit y avoir une seule manière de compiler une certaine structure, ou toutes les manières de la compiler doivent être possibles et équivalentes (aux points de vue de leur compilation, du résultat de leur compilation et de leur complexité à l'exécution);

Le respect de ce principe simplifie la génération. Aucune analyse du code n'est requise pour découvrir le meilleur code à générer dans un cas donné : c'est toujours le même code qui doit être généré.

- *primitives* : il vaut mieux proposer un jeu de primitives avec lesquelles on peut résoudre les problèmes que donner des solutions aux problèmes eux-mêmes;

La tendance CISC est d'offrir de riches structures de contrôle (branchements conditionnels, appels/retours de fonctions) dans le jeu d'instructions des processeurs. Bien que louables, ces efforts mènent couramment à des écarts sémantiques entre la définition des langages de haut niveau et les instructions visant à les émuler au niveau du langage machine. L'auteur indique :

« *A machine that attempts to support all implementation requirements will probably fail to support any of them efficiently* »

Notons que les principes d'unicité et des primitives simplifient la génération de code, donc l'écriture des compilateurs.

- *adressage* : les modes d'adressages doivent être conçus en gardant à l'esprit les différents types de données manipulables par la machine;

Ce principe reprend un peu le précédent pour les problèmes d'accès aux données (modes d'adressages indirects, indexés et combinés). Il existe parfois un écart sémantique entre les définitions du langage de haut niveau (structures, pointeurs, ...) vis-à-vis des modes d'adressages visant à les refléter.

Ces deux derniers points sont résumés par l'auteur par « *Some architectures have provided direct implementations of high-level concepts. In many cases these turn out to be more trouble than they are worth.* »

- *environnement* : alors que les opérations arithmétiques ou logiques sont prises en compte par le matériel, les environnements d'exécution ne le sont pas. Par environnement d'exécution, il faut entendre la gestion des contextes des fonctions et tâches, ... Ils doivent être fournis par logiciel;
- *déviations* : on doit ne dévier de ces principes que de manière indépendante de l'implantation.

Il est toujours tentant, sur une machine donnée, d'utiliser telle ou telle astuce contraire aux principes précédents, mais qui diminueront les temps d'exécution. Par exemple, l'utilisation de l'instruction de décalage logique pour réaliser une multiplication ou une division par 2 est fortement déconseillée, bien que l'opération soit réalisée plus rapidement. Par ailleurs, réaliser une multiplication par un décalage à gauche n'est pas assurée de toujours fonctionner (format des données ou autre). Ce genre de pratique est expressément déconseillé par Wulf.

Comme nous allons le voir, nombre de ces principes ont été repris pour la définition de Devil.

Respect de ces principes par les définitions de MAD et Devil

<u>Devil et les architectures cibles</u>

Dans les définitions de MAD et Devil, les hypothèses faites quant aux architectures cibles sont peu nombreuses. Aussi ces définitions recouvrent un large spectre d'architectures. Nous avons

seulement supposé que nous disposions d'unités de calcul et de contrôle et d'une unité de stockage des informations. Cette dernière doit autoriser l'accès direct aux données à partir d'une adresse; une quelconque structuration (registres ou autres, spécialisation de certaines zones mémoire) n'a pas été supposée. Nous avons supposé que l'unité de contrôle disposait des facilités habituelles (sauts inconditionnels et conditionnels, possibilité d'implanter des appels et retours de sous-programmes). Aucune hypothèse concernant le mode de passage des paramètres aux fonctions ou le mode de passage de leurs valeurs de retour n'a été faite. Nous avons fait très peu d'hypothèses sur l'existence, le nombre et la nature des unités fonctionnelles. Nous supposons simplement que toutes les opérations arithmétiques et logiques classiques sont réalisables. La nature vectorielle ou parallèle des cibles n'apparaît pas dans la définition de MAD ou de son langage. Seules des machines d'architectures trop différentes (architectures à flux de données par exemple) ont d'emblée été placées en dehors du champ des cibles visées.

Pour contribuer à l'indépendance vis-à-vis de l'architecture de la cible, nous avons établi nos définitions indépendamment du format des données du langage. Les formats des entiers ou des réels simple ou double précision ne sont pas précisés pour des soucis d'efficacité d'implantation. De même, les définitions ne précisent pas si une composante d'un vecteur de bits est représentée par un bit ou plus (octet, entier, ...). De cette manière, la projection de MAD sur une cible est libre d'utiliser les données de base de celle-ci.

Orthogonalité de Devil

En général, le jeu d'instructions de MAD est orthogonal. Non seulement les instructions acceptent généralement des opérandes scalaires ou vecteurs, mais ils sont indifféremment entiers, flottants simple ou double précision. Une même instruction accepte les deux catégories d'objets en opérande, avec mélange éventuel. Ce caractère orthogonal simplifie Devil en limitant son jeu d'instructions.

Le caractère orthogonal de Devil simplifie la génération de code Devil. Le compilateur n'a pas à rechercher, selon le type des opérandes, l'instruction qui les acceptent. En général, tout ce qui a un sens dans le langage de haut niveau ne pose aucun problème de compilation en Devil, les instructions acceptant tout type d'opérande. La génération des instructions peut être séparée de la génération des opérandes, ceci menant à une plus grande modularité dans l'écriture du compilateur [CORDY & al. 90].

Cette orthogonalité entraîne une même instruction de traitement à posséder un nombre assez grand de formes. Ainsi, une instruction d'addition en "cache" en fait 990 (!) si l'on combine tous les types d'opérandes possibles et les différents modes de fonctionnement de l'instruction (production d'un résultat temporaire ou non, description de l'instruction). Cependant, la traduction de l'instruction n'est pas si complexe que ce nombre pourrait le laisser croire. L'analyse des opérandes est séparée de l'analyse de l'instruction elle-même et de ses caractéristiques (cf. infra, la production des résultats - temporaires ou non - et la description des instructions). Par ailleurs, la génération des instructions Devil depuis un langage de haut niveau en est simplifiée : toutes les formes que l'on peut imaginer existent.

Transparence de Devil quant à la qualité du code généré

L'utilisation d'une couche intermédiaire ne doit pas être pénalisante. Ainsi, le code généré doit être d'aussi bonne qualité que si l'on compilait le langage de haut niveau directement, sans utiliser de couche intermédiaire. Une perte de qualité due à l'utilisation d'une couche intermédiaire est typiquement due à une perte d'informations qui sont connues au niveau du langage évolué et ne sont plus synthétisables au niveau intermédiaire. Ceci nous à mené à la transmission, lors de la compilation, d'un ensemble d'informations sur la structure du programme. En effet, la qualité du code généré dépend de la nature des structures de haut niveau qui ont été compilées. Les informations transmises le sont par le biais de pseudo-instructions, des attributs des opérandes, la spécification des classes d'allocation des objets (global, local, paramètre, ...), la production et l'utilisation de temporaires, ...

Ré-utilisabilité

Du fait de la quantité considérable de fonctions disponibles en bibliothèques, nous avons voulu que l'on puisse éditer des liens entre des objets Devil et des objets provenant de sources C ou Fortran, ou avec des bibliothèques (partage de variables ou de fonctions). Cela amène quelques conséquences sur la définition de MAD et Devil en ce qui concerne l'interface Devil/Fortran et Devil/C (au niveau du passage de paramètres entre fonctions pour l'essentiel).

Principe d'unicité

De nombreux (micro-)processeurs (tels MC 680x0, i80386, ...) possèdent des instructions optimisant l'exécution de formes particulières d'instructions plus générales, fréquemment utilisées. Par exemple, citons :

- l'instruction de mise à 0 de la valeur d'un opérande qui est un cas particulier de l'instruction d'affectation d'une valeur à un opérande;
- l'instruction de comparaison de la valeur d'un opérande à 0 qui est un cas particulier de l'instruction de comparaison;
- les instructions d'incrémentation et de décrémentation qui sont des cas particuliers d'addition et de soustraction;
- les instructions prenant un opérande immédiat de valeur faible (codé sur un petit nombre de bits, dans le même mot que le code instruction) par rapport aux instructions prenant le même genre d'opérandes mais avec des valeurs quelconques (alors codées sur plusieurs mots mémoires).

L'intérêt de ces instructions réside dans le fait qu'elles sont exécutées plus rapidement que les instructions générales. De telles instructions "optimisant localement le jeu d'instructions" n'existent pas en Devil. Ceci pour plusieurs raisons :

- simplification (orthogonailté) du jeu d'instructions;
- respect du principe d'unicité (une seule manière de réaliser l'addition de 1 à la valeur d'un objet par exemple : il n'existe pas d'instruction d'incrémentation);
- la prise en compte de telles considérations matérielles est inacceptable dans la définition de Devil.

L'utilisation de telles optimisations est réalisée par le traducteur.

Certaines opérations peuvent être codées de plusieurs manières en Devil. Par exemple, pour comparer deux vecteurs, trouver leurs éléments différents et créer un vecteur de bits en conséquence, il y a plusieurs solutions :

Solution 1	Solution 2
`eq v, w` `not &1`	`ne v, w`

Ceci entraîne une violation du principe d'unicité. La solution 2 est la solution la plus raisonnable. Un optimiseur de code Devil devrait éliminer les séquences d'instructions du genre de celle de la solution 1 et les transformer dans la forme de la solution 2.

Principe des primitives

L'application du principe des primitives a nécessité de dégager des concepts de haut niveau afin de pouvoir les implanter assez facilement sur les différentes cibles. Ceci à mené d'une part à la

définition des vecteurs telle qu'elle existe en Devil. Celle-ci est d'un niveau bien supérieur aux vecteurs traités par les processeurs vectoriels. D'autre part, la définition du jeu d'instructions a été influencée par ce principe. Ainsi, les instructions de Devil sont habituellement des généralisations des instructions ("orthogonalisation") des machines vectorielles.

Etre un langage intermédiaire a entraîné deux problèmes :

- être au-dessus des langages machines et architectures visées;
- être une forme primitive de langage autorisant l'expression du caractère vectoriel des algorithmes. Typiquement, la compilation ne doit recouvrir que ce qui est indépendant de la machine cible et la traduction ne doit recouvrir que ce qui est dépendant de la cible. Le compilateur doit prendre en charge les optimisations indépendantes de la cible lors de sa génération de code Devil (recherche de sous-expressions communes, extraction d'invariants des boucles, ...). Le traducteur doit effectuer toutes les optimisations liées à la cible (allocation des variables dans les registres, recherche de la traduction optimale des instructions Devil, ...).

Les structures de contrôles (tests, boucles) n'existent pas en Devil. Leur optimisation est donc plus difficile. Les structures de contrôle n'ont pas été introduites dans Devil pour les raisons suivantes :

- MAD est une machine intersection;
- écart sémantique possible entre les structures Devil et celles des langages de haut niveau. Pour des langages de haut niveau comme C, Ada ou EVA, la sémantique de la boucle pour n'est pas la même. Aussi, une notion unificatrice de boucle est difficile à définir;
- cette remarque n'a d'intérêt que pour les boucles où les données peuvent être mémorisées dans les registres du processeur. Or, dans un langage vectoriel explicite, il y a assez peu de boucles.

1.4.3. Les différentes familles d'instructions

Les instructions sont présentées en annexe. Une table (cf. table II.1) résume les points qui suivent. Les instructions de Devil peuvent être réparties en 4 familles :

- les pseudo-instructions sont des directives de traduction. Elles définissent ou déclarent des symboles. Elles transcrivent également des renseignements qui seront utiles à la traduction;
- les instructions scalaires se séparent en deux sous-familles, les instructions de contrôle de l'exécution des programmes (instructions de saut par exemple) et les instructions de traitement des scalaires (instructions arithmétiques et logiques classiques);
- les instructions vectorielles acceptent des opérandes vectoriels ou scalaires en source et produisent un résultat vectoriel. Ce sont toutes des instructions opératoires;
- les instructions mixtes acceptent des opérandes vectoriels en source et produisent un résultat scalaire.

Les instructions scalaires

Les instructions scalaires se répartissent en deux groupes : les instructions de traitement des objets scalaires et les instructions de contrôle de MAD.

Les instructions scalaires de traitement se divisent en deux sous-familles qui sont :

- les instructions réalisant les calculs arithmétiques, logiques, trigonométriques, ...
- les instructions de transferts réalisant les affectations classiques ou des affectations conditionnelles, l'objet affecté à la cible dépendant alors d'une condition.

Les instructions scalaires de contrôle se séparent en 4 sous-familles qui sont :

- les branchements conditionnels et inconditionnels effectuant les ruptures de séquence d'instructions (modifiant la valeur du registre PC);
- les instructions de gestion des appels/retours de fonctions. Elles effectuent des opérations de sauvegarde/restauration de contextes de la fonction appelante. Par ailleurs, elles modifient la valeur du registre PC pour exécuter la fonction appelée ou reprendre l'exécution de l'appelant;
- les instructions de passage de paramètres aux fonctions appelées effectuant les opérations de transferts ou partages de la valeur d'objets dans le segment *param_to_send*.
- les instructions de gestion du registre VLG effectuant la mise à valeur du registre VLG et gérant la pile associée (segment *vlg*).

Les instructions vectorielles

Les instructions vectorielles se décomposent en trois groupes :

- les instructions arithmétiques qui réalisent les calculs arithmétiques, logiques, trigonométriques, logarithmiques, les conversions, ...;
- les instructions d'accès/manipulations qui réalisent les opérations de transferts vectoriels (transferts "classiques" `move` ou concaténations `cat`) et les accès dispersés aux éléments des vecteurs (rassemblement/éclatement, compression/extension, merge, mask);
- les instructions de construction qui créent les vecteurs et les initialisent (`assoc`, `alloc`, `dassoc`).

En outre, les instructions vectorielles possèdent trois caractéristiques :

- génération d'un résultat temporaire, d'un résultat non temporaire ou les deux;
- possibilité de description ou non. Si elle est décrite, l'opération est réalisée sous le contrôle d'un descripteur qui sélectionne les composantes du résultat à mettre à jour;
- génération d'un résultat dépendant de la valeur courante de VLG ou non.

Toutes les combinaisons "type d'instruction vectorielle"/caractéristique n'existent pas. Le sens des opérations nous a dicté quelques règles indiquant les combinaisons valides. Ainsi, les instructions arithmétiques peuvent toutes générer un résultat temporaire, un résultat non temporaire ou les deux, être décrites ou non et génèrent toujours un résultat comprenant VLG composantes.

Pour leur part, les opérations de construction ne génèrent jamais de temporaires et ne sont jamais décrites. Seule l'une d'entre-elles (`sinitvlg`) génère un résultat ayant VLG composantes. Toutes les autres ne tiennent pas compte de la valeur courante de VLG.

Enfin, les instructions d'accès génèrent un résultat temporaire, non temporaire ou les deux. Leur réaction face à la valeur du registre VLG est variable.

Les instructions mixtes

Les instructions mixtes se décomposent en deux groupes :

- les instructions vectorielles de réduction comme le calcul de la somme des éléments d'un vecteur, l'accès à la composante de valeur maximale ou à son indice, ...

- les instructions d'accès aux champs de l'en-tête des vecteurs comme le nombre d'éléments d'un vecteur, la taille de sa zone d'allocation, ...

Comme les opérations vectorielles, les opérations mixtes possèdent les trois caractéristiques citées plus haut (génération de résultat temporaire, descriptible, résultat dépendant de la valeur de VLG). Toutes les caractéristiques sont acceptables pour les instructions de réduction. Par contre, les instructions d'accès aux champs des en-têtes ne peuvent ni être décrites, ni avoir un résultat dépendant de la valeur courante de VLG. Mais, elles génèrent des résultats temporaires, non temporaires ou les deux.

La table suivante résume les différentes classes d'instructions Devil. Pour chacune, elle indique les possibilités de description, de générer un temporaire ainsi que la prise en compte de la valeur courante de VLG.

/instructions-Devil

	descriptible	tmp	VLG
Les pseudo-instructions	.	.	.
Les instructions scalaires			
traitement			
opérations (+, -, *, ...)	.	x	.
transferts(move, select, ...)	.	x	.
contrôle			
branchements	.	.	.
appels de fonctions	.	x	x
passage de paramètres	.	.	.
gestion de VLG	.	.	.
Les instructions vectorielles			
opérations (+, -, *, ...)	x	x	x
accès/manipulations (move, cat, select, gath, merge, ...)	x	x	parfois
construction (assoc, alloc, dassoc, ...)	.	.	.
Les instructions mixtes			
opérations de réductions(Σ, Π, max, ...)	x	x	x
accès aux champs de l'en-tête	.	x	.

Table II.1 -
Structure du jeu d'instructions de Devil

Etant donnée leur généralité, les instructions calculatoires de Devil ne peuvent se ranger aisément dans cette classification. Par contre, une instruction munie des attributs de ses opérandes peut y être rangée. Ainsi,

```
s. s. s.   add
```

est une instruction scalaire de traitement. Par contre,

```
s. s. v.   add
```

et

```
v. v. v.   add
```

sont des instructions vectorielles opératoires.

La première instruction produit la somme de deux objets scalaires et l'affecte à un objet scalaire; la deuxième calcule la somme de deux objets scalaires et affecte sa valeur à toutes les composantes du vecteur cible; la troisième calcule la somme de deux vecteurs et range celle-ci dans un vecteur.

```
v. v. v.    gath
```

est une instruction vectorielle de manipulation,

```
v. s. s.    gath
```

est une instruction mixte de réduction,

```
v. s. v.    gath
```

est une instruction vectorielle de manipulation.

La première instruction effectue une opération de rassemblement ou de compression selon la nature de l'opérande de description (vecteur d'entiers ou de bits); la deuxième instruction affecte la valeur d'une composante d'un vecteur; la troisième affecte la valeur d'une composante d'un vecteur à l'ensemble des éléments d'un vecteur cible.

Comme exemple d'utilisation de la valeur de retour d'une fonction et du passage de paramètres, considérons l'expression suivante :

```
v := f (a, g (b + c), 3 * e)
```

Elle s'exprime en pseudo-Devil sous la forme :

```
  add       b, c
n.parout    &1, #0
  call      g
n.parout    a, #0
n.parout    &2, #INT-SIZE
  mul       #3, e
n.parout    &1, 2 x #INT-SIZE
n.call      f, v
```

Nous ne nous sommes pas intéressés aux attributs des instructions. Pour obtenir un source correcte, il faudrait les y ajouter. Cependant le code serait exactement le même que les objets soient des vecteurs ou des scalaires. Notons que s'il y a des vecteurs dans l'expression, les instructions précédentes se trouvent dans un bloc vectoriel. Elles sont donc nécessairement précédées par une instruction `push` et suivies d'une instruction `pop`.

1.4.5. Le typage des opérandes

Tous les opérandes sources sont typés dans le code instruction. Le résultat est typé dans le code instruction sauf s'il s'agit d'un temporaire. Si l'instruction ne produit son résultat que sous la forme d'un temporaire, des règles sont appliquées pour connaître le type du résultat qui est alors implicite. Le fait qu'un temporaire résultat soit un scalaire ou un vecteur dépend de l'instruction le générant et des opérandes de cette instruction. Lorsqu'un opérande résultat est spécifié, il y a forçage automatique de la valeur obtenue vers le type du résultat qui est aussi le type du temporaire.

L'ordre suivant est défini sur les types :

bit < entier < flottant simple précision < flottant double précision.

Parmi les instructions qui produisent un résultat temporaire :

- il y a les instructions qui, quel que soit le type des opérandes, génèrent un résultat de type constant. Par exemple, une instruction de comparaison génère forcément un scalaire entier si les deux opérandes sources sont scalaires, un vecteur de bits si l'un des opérandes sources est vectoriel;
- il y a les instructions qui, si les sources sont du même type, génèrent un résultat de ce même type et qui, si les sources sont de types différents, génèrent un résultat dont le type est le plus grand (au sens de l'ordre donné ci-dessus) des deux sources. Par exemple, l'opération d'addition suit cette règle. L'addition de deux scalaires entiers donne un temporaire scalaire entier. L'addition d'un vecteur d'entiers et d'un vecteur de flottants double précision donne un vecteur de flottants double précision.

1.4.6. Les attributs des opérandes

Un objet possède des caractéristiques statiques (connues à la compilation : sa catégorie et son type). S'il s'agit d'un vecteur, il possède également des caractéristiques dynamiques (connues seulement à l'exécution et qui peuvent changer à l'exécution). Ces dernières sont des propriétés que possède un vecteur à un moment donné de l'exécution du programme. Etant donnée la forme très générale des vecteurs et la puissance des instructions Devil par rapport aux instructions des langages machines existants, de nombreux tests sont nécessaires pour savoir comment exécuter une instruction Devil sur une machine. Ces tests sont par exemple liés à la nature du descripteur des vecteurs. Selon celle-ci, l'accès à un vecteur est contigu, non adjacent avec pas, nécessite un rassemblement, un éclatement, une compression ou une extension. Un mécanisme d'attributs associés aux opérandes des instructions a été mis en œuvre. Celui-ci autorise la génération du code adéquat si la forme du descripteur est connue à la compilation en évitant les tests à l'exécution. La suppression de ces tests permet un gain de temps à l'exécution. Dans un programme Devil, ces attributs peuvent avoir une valeur non significative mais leur utilisation est recommandée à chaque fois que cela est possible. Ils spécifient précisément la forme d'un opérande vectoriel, c'est à dire le type de son descripteur et de sa zone d'allocation. Ces attributs sont générés automatiquement par le compilateur EVA.

Les attributs indiquent sept informations. Deux d'entre elles sont indispensables dans une instruction (les attributs définissant la catégorie et le type de l'opérande). Les autres attributs sont facultatifs (ou du moins, ils peuvent être inconnus à la compilation). Une valeur indéterminée leur est alors assignée. Les différents attributs sont :

catégorie : indique si l'opérande est un scalaire ou un vecteur (attribut obligatoire);

type : indique le type du scalaire ou des éléments du vecteur (attribut obligatoire);

type du descripteur : indique le type de descripteur (liste de bits, liste d'index ou inconnu);

descripteur triplet ou non : dans le cas où le descripteur est une liste d'index, cet attribut indique si c'est un triplet ou non, ou si on ne le sait pas;

descripteur triplet à pas unité ou non : dans le cas où le descripteur est un triplet, cet attribut indique si son pas vaut 1, s'il est différent de 1 ou si on ne le sait pas.

Les deux attributs qui suivent ne sont valides que s'il s'agit d'un vecteur d'entiers.

zone d'allocation triplet ou non : indique si la zone d'allocation est sous la forme d'un triplet ou non, ou si on ne le sait pas;

zone d'allocation triplet à pas unité ou non : dans le cas où la zone d'allocation est un triplet, indique si son pas vaut 1, est différent de 1 ou si on ne le sait pas.

L'analyse de ces attributs par le traducteur Devil autorise la génération du code ad hoc. Ceci évite les tests à l'exécution. Dans le cas où l'opérande est imparfaitement spécifié, le traducteur génère le code pour effectuer les tests à l'exécution concernant les attributs inconnus.

1.4.7. Manipulation directe des triplets

La forme des triplets est conservée lorsque certaines opérations calculatoires les utilisent. Ainsi, l'ajout d'une valeur scalaire entière à un triplet ou le produit des éléments d'un triplet par un entier rendent un triplet (cf. fig. II.9.).

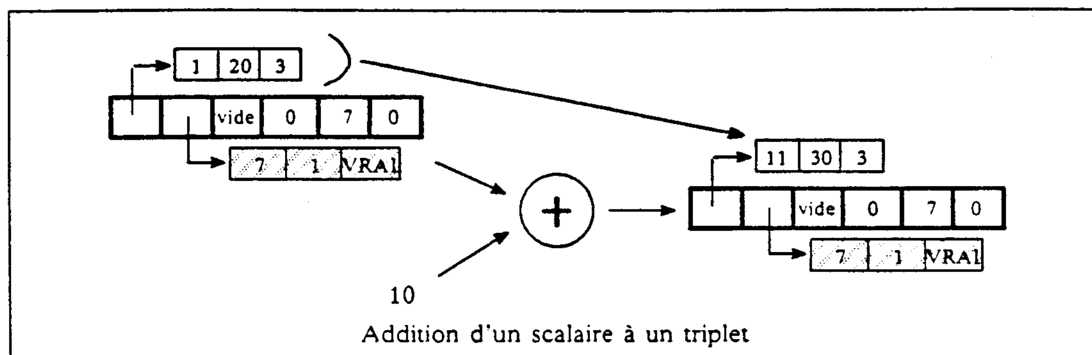


Figure II.9. -
Calculs sur les triplets

La manipulation de triplets possède plusieurs intérêts, du fait de leur représentation par trois entiers. Ils représentent une forme compacte de stockage des éléments d'un vecteur. De ce fait, l'accès aux éléments d'un vecteur représenté de cette manière est très rapide. Certaines opérations sont des lois internes à l'ensemble des triplets (addition); leur forme est alors conservée lors de ces calculs. Les opérations sur les triplets n'ont pas à être découpées (*strip-mining* - cf. la projection de Devil, § II.), ce qui améliore le code généré. Par ailleurs, sur Cray Y-MP, le traitement des vecteurs est plus intéressant que le traitement scalaire dès qu'un vecteur possède au moins deux éléments. Ainsi, il est intéressant, dans certains calculs, de considérer les triplets comme des vecteurs de trois éléments et de les traiter comme tels sur cette machine.

1.4.8. Les instructions décrites

Toutes les instructions vectorielles et les instructions mixtes opératoires du langage Devil peuvent être décrites. Par ailleurs, l'instruction (de contrôle de flot) d'appel de fonction `call` peut également être décrite. Une instruction décrite est une instruction qui est exécutée sous le contrôle d'un descripteur (scalaire entier, vecteur d'index ou de bits). Seules les composantes du résultat sélectionnées par le descripteur sont effectivement rangées dans les éléments correspondants du vecteur cible.

Par application du principe d'orthogonalité, les instructions générant un temporaire peuvent également être décrites sans aucune restriction. Dans ce cas, les composantes du temporaire non sélectionnées par le descripteur ont une valeur indéterminée. Si l'on reprend l'interprétation des temporaires comme étant des données locales à l'évaluation d'une expression, une telle opération s'interprète par un opérateur décrit. Ce type d'opérateur semble n'exister dans aucun langage de haut niveau à l'heure actuelle hormis des langages comme Lisp ou les langages objets qui en permettent la définition. La description des opérateurs est à l'étude dans EVA et devrait y être implantée à brève échéance.

1.4.9. Description de quelques instructions Devil

Nous nous attachons ici à décrire, sans les détailler, quelques instructions importantes. Tout d'abord, nous observons les instructions fondamentales que sont les instructions de construction de vecteurs (associations de zones d'allocation et de description à des en-têtes de vecteur). Nous étudions ensuite les instructions de manipulation des éléments des vecteurs (instructions de rassemblement / éclatement, ...) puis quelques instructions spécifiques.

Les instructions de construction de vecteurs

Les instructions de construction possèdent les fonctions suivantes :

- allocation d'une zone d'allocation et de la zone d'informations associée (instruction `alloc`);
- allocation d'une zone d'allocation et de la zone d'informations associée avec initialisation de la zone d'allocation (instruction `assoc`);
- partage d'une zone d'allocation et de la zone d'informations associée entre deux vecteurs (instruction `assoc`);
- partage d'une zone d'allocation et de la zone d'informations associée entre deux vecteurs et association d'un descripteur au vecteur construit (instruction `dassoc`).
- création d'un temporaire sous forme de triplet (instruction `sinit`);
- initialisation des champs d'un en-tête de vecteur à partir de zones d'allocation et d'informations pré-existantes (instruction `setza`).

Les zones d'allocation et d'informations sont exclusivement créées (allouées) via l'instruction `alloc` (mises à part les instructions de passage de paramètres `parin`, `parconst` et `parout`). Les instructions `assoc` et `dassoc` mènent à un partage de ces zones et sont les seules à créer une zone de description. Toutes les zones créées sont allouées dans le segment de tas de MAD. Notons qu'avant toute association, il y a dé-référenciation des anciennes zones du vecteur et éventuellement libération de ces zones si elles ne sont plus utilisées par aucun autre vecteur. Les zones de description étant privées, une zone de description est toujours désallouée. Les zones d'allocation et d'informations ne sont désallouées que si le compteur de références (dans la zone d'informations) prend une valeur nulle après décrémentation.

<u>L'instruction alloc</u>

L'instruction `alloc` est une instruction de création de vecteur "à partir de rien". Elle alloue une zone d'allocation de taille donnée et l'associe au vecteur spécifié. Une zone d'informations associée est également créée pour ce vecteur.

<u>L'instruction assoc</u>

L'instruction `assoc` est une instruction de création de vecteur en suivant un modèle (un autre vecteur). Le vecteur créé partage la zone d'allocation du modèle. Sa zone de description est une copie de la zone de description du modèle (il n'y pas partage). De cette manière, le vecteur créé possède les mêmes éléments effectifs et la même zone d'allocation que le modèle.

<u>L'instruction dassoc</u>

L'instruction `dassoc` est également une instruction de création de vecteur en suivant un modèle, mais fonctionnant, typiquement, par "affinage" de la description. En plus du vecteur à créer et du modèle, cette instruction prend un opérande qui joue le rôle de descripteur (on aurait pu dire qu'il s'agissait là d'une instruction `assoc` décrite, mais la sémantique du `dassoc` est différente de celle que

l'on pourrait imaginer pour une instruction `assoc` décrite). Comme pour l'`assoc`, il y a tout d'abord partage de la zone d'allocation entre le vecteur créé et son modèle. Par contre, sa zone de description est obtenue par composition de la zone de description du modèle avec le descripteur. Le descripteur sélectionne les éléments de la zone de description du modèle qui feront effectivement partie de la zone de description du vecteur créé. C'est cette composition que nous nommions plus haut "affinage" car, typiquement, une telle opération sert à obtenir un vecteur dont le descripteur sélectionne un sous-ensemble (éventuellement avec des répétitions) des éléments effectifs du modèle.

L'instruction setza

Une dernière instruction (`setza`) est disponible pour créer des vecteurs. Celle-ci crée un vecteur sans allouer de zones. Elle initialise les champs l'en-tête d'un vecteur avec les adresse de zones déjà allouées. L'existence de cette instruction repose sur, d'une part, des raisons d'efficacité et, d'autre part, sur l'interfaçage de Devil avec d'autres langages non vectoriels (Fortran). Elle, sera décrite dans la section concernant l'implantation de Devil, plus loin dans ce chapitre.

L'instruction sinit

La création de vecteurs temporaires ayant la forme d'un triplet passe exclusivement par l'utilisation de l'instruction `sinit`. Ses opérandes sont les valeurs des bornes inférieures et supérieures du triplet ainsi que son pas. L'association d'un triplet (par une instruction `assoc` ou `dassoc`) à un vecteur donne une forme de triplet à sa zone d'allocation.

Exemple

```
        { v1 et v2 sont initialisés,
          de taille 10 et non décrits. }
1.      vi vi        add       v1, v2
2.      vi vi      n.assoc     &1, v3
3.      si si si    sinit      #1, #10, #2
4.      si si si    sinit      #2, #10, #2
5.      vi vi vi   n.dassoc    V3, &2, impair
6.      vi vi vi   n.dassoc    V3, &2, pair
```

La somme des vecteurs *v1* et *v2* est calculée et placée dans un vecteur temporaire (ligne 1). Un vecteur (*v3*) est alors construit et initialisé à partir de ce vecteur temporaire (ligne 2). Des triplets temporaires sont construits (ligne 3 et 4) qui permettent la construction des vecteurs *impair* et *pair* (ligne 5 et 6). Ces vecteurs contiennent respectivement les éléments d'indices impairs et les éléments d'indices pairs de la somme précédemment calculée (ligne 1).

Les instructions de rassemblement / éclatement

Les définitions des instructions classiques de rassemblement / éclatement ont été étendues. Habituellement, ces opérations prennent un vecteur source, un vecteur cible et un vecteur d'index qui contrôle la réalisation de l'opération. En Devil, toujours pour son souci d'orthogonalité, cette définition est modifiée pour autoriser n'importe quel type de descripteur à la place de ce vecteur de contrôle. Ainsi, si le descripteur est un vecteur d'index ou un triplet, on retrouve les opérations classiques de rassemblement / éclatement. Si le descripteur est un vecteur de bits, on retrouve les opérations classiques de compression / extension. Enfin, si le descripteur est un scalaire, on obtient les opérations d'accès aux composantes des vecteurs.

La figure II.10 illustre les opérations de rassemblement/éclatement sur des vecteurs "simples" (sans descripteur). Pour les deux opérations, un même opérande descripteur est utilisé. Il s'agit d'un vecteur d'entier, donc les opérations sont de "classiques" accès en lecture ou écriture aux éléments d'un vecteur, via un vecteur d'index. Dans le résultat de l'opération d'éclatement, il est intéressant de noter les valeurs écrites dans les première et cinquième composantes du vecteur cible (60 et 10). Le résultat de l'opération pour ces deux composantes est non précisé au niveau de MAD, les

valeurs 1 et 5 apparaissant plusieurs fois dans le vecteur d'index. Cette imprécision au niveau de la définition de MAD est due à ce que le résultat de ce type d'opérations peut varier d'une machine à une autre. La définition stricte de l'opération d'éclatement dans MAD entraînerait d'éventuels écarts sémantiques avec les opérations pré-définies de la cible, d'ou une lourdeur pour sa réalisation sur de telles cibles. Ce problème n'apparait que dans le cas d'un descripteur non injectif, c'est à dire d'un vecteur d'index possédant plusieurs composantes de même valeur. Il est intéressant de noter qu'un langage comme Fortran 90 interdit de telles expressions [METCALF & al. 90].

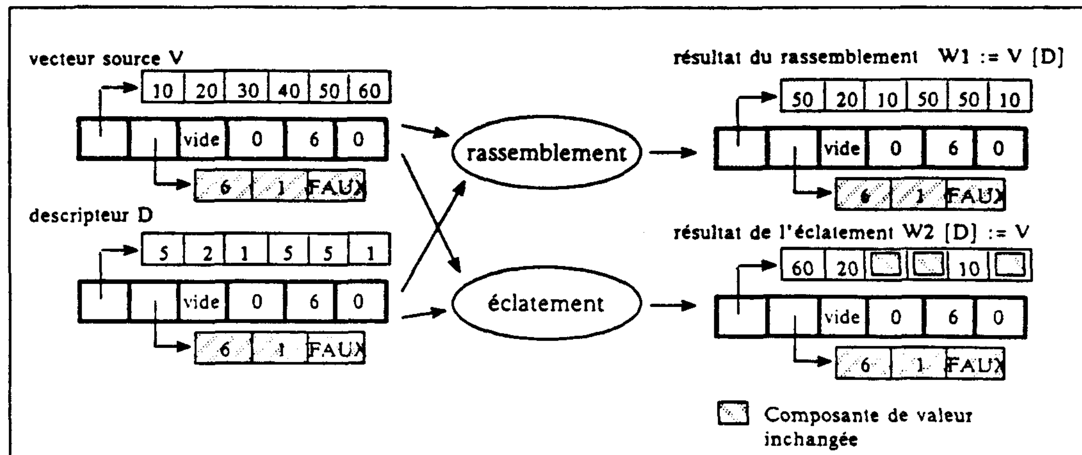


Figure II.10. -
Les instructions de rassemblement / éclatement

Aussi, l'imprécision sur la définition de l'opération d'éclatement avec un descripteur non injectif est du même ordre que les imprécisions sur les définitions des opérations arithmétiques et le format des nombres. Elles permettent d'adapter MAD aux spécificités de chaque cible. Cela peut limiter dans une certaine mesure la portabilité des applications. Cependant, on y gagne au niveau du code qui est généré, et donc au niveau de la vitesse d'exécution.

Manipulations des éléments de vecteurs

En plus des instructions de rassemblement / éclatement, nous regroupons sous ce vocable l'instruction d'affectation simple (**move**), des instructions de concaténations de vecteurs (**cat** et **ncat**), une instruction d'affectation masquée (**mask**) et une instruction de mixage (**merge**) des composantes de deux vecteurs sources contrôlées par un descripteur.

Les instructions de concaténation permettent pour l'une (**cat**) de mettre "bout à bout" les éléments de plusieurs vecteurs et de les ranger dans un vecteur cible, pour l'autre (**ncat**) à concaténer un nombre donné de fois les éléments d'un même vecteur.

L'instruction **mask** réalise l'affectation par masquage des sources par le descripteur. L'instruction **merge** réalise un double rassemblement contrôlé par le descripteur. Pour ces deux instructions, la première source est contrôlée par le descripteur, la deuxième par le complémentaire du descripteur (les éléments non sélectionnés par le descripteur sont alors sélectionnés).

L'instruction d'affectation dite simple (**move**) réalise l'affectation habituelle par recopie de la valeur de la source dans la cible. L'affectation d'un scalaire à un vecteur entraîne l'affectation de la valeur de ce scalaire aux VLG premières composantes effectives du vecteur. Cette instruction peut être décrite. Notons que si la source est un scalaire et que la cible est un vecteur, l'instruction

```
move s, v [ d ]
```

est équivalente à

```
scat s, d, v
```

Ceci entraîne une violation du principe de régularité. Cependant, l'interdiction de décrire l'instruction move aurait été une violation de l'orthogonalité du langage. Aussi, nous avons considéré que la violation de la régularité était limitée à une seule famille de cas (instruction scat avec une source scalaire et une cible vectorielle) ce qui est un moindre mal.

Accès aux extréma des composantes des vecteurs et à leurs indices

Des instructions sont disponibles pour :

- accéder aux valeurs maximales et minimales parmi les VLG premières composantes d'un vecteur;
- accéder aux indices des valeurs extrémales parmi les VLG premières composantes d'un vecteur;
- calculer le nombre de composantes à 0 ou 1 parmi les VLG premières composantes d'un vecteur de bits (ou autre);
- calculer l'indice du premier ou du dernier bit à 0 ou à 1 dans un vecteur de bits (ou autre), parmi les VLG premières composantes effectives du vecteur.

Ces instructions de réduction reflètent des opérations utiles dans l'expression d'algorithmes vectoriels. Les premières (accès aux extréma et à leurs indices) sont incluses dans le jeu d'instructions des supercalculateurs Fujitsu VP. L'instruction de comptage de bits à 0 ou à 1 reflète l'instruction "Population count" des Cray et la dernière, l'instruction "leading zero count" des Cray.

Instructions de comparaison

Des instructions de comparaison (test d'égalité, de différence, plus grand, plus petit, ...) sont disponibles. Sous leur forme scalaire, elles donnent simplement un résultat scalaire entier dont la valeur indique si le résultat de la comparaison est vrai ou faux. Ce résultat peut être utilisé par une instruction de branchement conditionnelle.

La forme vectorielle génère un vecteur de bits, le ième bit indiquant le résultat de la comparaison des ième composantes sources. Par la suite, ce vecteur de bits peut être utilisé pour réaliser une sélection d'éléments d'un autre vecteur, ou comme zone de description d'un vecteur, ou comme descripteur d'instructions décrites. Par exemple, supposons que l'on veuille compresser dans un vecteur *w* tous les éléments d'un vecteur *v* supérieurs à 100. Cette opération peut s'écrire sous la forme :

```
 { v et VLC sont initalisés }
vi si       ge    v, #100
vi vb vi    gath  v, &1, w
```

La première instruction crée un vecteur de bits indiquant les éléments supérieurs à 100 du vecteur *v*. Par la suite, le vecteur de bits résultat est utilisé pour compresser ses éléments dans *w*.

Affectation conditionnelle

La prise en charge d'expressions conditionnelles au sens défini dans le langage C (ou EVA) à nécessiter l'introduction en Devil d'une instruction d'affectation conditionnelle (select). Considérons l'expression scalaire :

```
a := ( < cond. > ? < exp. 1 > : < exp. 2 > ) + < exp. 3 >
```

Sans l'instruction select, elle devrait se traduire en :

```
    n.branch    < cond. >, sinon
      < calcul de exp. 1 >
      < calcul de exp. 3 >
    n.add       < exp. 1 >, < exp. 3 >, a
    n.jmp       fin
  sinon:
      < calcul de exp. 2 >
      < calcul de exp. 3 >
    n.add       < exp.2 >, < exp. 3 >, a
  fin:
```

Le code du calcul de l'`< exp. 3 >` est présent dans les deux branches du test pour tenir compte des éventuels effets de bord dus à l'exécution des `< exp. 1 >` et `< exp.2 >`. En utilisant l'instruction `select`, l'expression se traduit en :

```
    n.fbranch  < cond >, sinon
      < calcul de exp. 1 >
    n.jmp      fin_select
  sinon:
      < calcul de exp. 2 >
  fin_select:
      select     < cond. >, < exp. 1 >, < exp. 2 >
      < calcul de exp. 3 >
  n.add        &1, < exp. 3 >, a
```

Chaque sous-expression de l'expression conditionnelle peut être une expression conditionnelle elle-même. Aussi, la génération de code sans instruction `select` mène à une duplication de code importante. Cette instruction simplifie donc grandement la génération de code Devil.

Manipulation de vecteurs multi-dimensionnels

Comme il a déjà été dit, les vecteurs Devil sont mono-dimensionnels. Une matrice ou plus généralement un vecteur possédant n dimensions (variété à n dimensions) sera représentée linéairement. Lorsqu'une matrice est mise sous forme linéaire, ses éléments sont rangés par colonnes, les éléments de la colonne 1 suivis des éléments de la colonne 2, ... En dimension trois, les éléments d'un espace sont rangés par plans, les éléments du plan 1, les éléments du plan 2, ... Dès lors, les accès aux éléments des lignes, colonnes, diagonales, ... sont obtenus par sélection des éléments à l'aide de descripteurs adéquats.

Considérant une matrice (deux dimensions $\eta \times \eta$), si l'on suppose que ses éléments sont rangés colonnes après colonnes (d'abord les éléments de la première colonne, puis ceux de la deuxième, ... puis ceux de la dernière), l'accès aux éléments de la première colonne se fera simplement avec un descripteur entier contenant les valeurs 1 à η, l'accès aux éléments de la première ligne avec le triplet $[\ 1 : \eta \times \eta : \eta\ ]$, l'accès aux éléments de la diagonale principale par le triplet $[\ 1 : \eta \times \eta : \eta + 1\ ]$, ... De la même manière, on peut sélectionner les éléments de telle sous-matrice, les éléments de telles lignes et telles colonnes. Cependant, le "calcul à la main" des descripteurs devient vite fastidieux. C'est pourquoi, nous avons introduit l'instruction `dim` qui réalise le calcul des descripteurs.

L'instruction `dim` prend deux descripteurs (un descripteur pour les lignes et un descripteur pour les colonnes d'une matrice par exemple) et les compose pour obtenir le descripteur à appliquer sur le vecteur représentant une matrice. Les dimensions de la matrice doivent être spécifiées dans l'instruction `dim`. Par la suite, ce descripteur est composable avec un descripteur de plans pour sélectionner les éléments d'un espace. Récursivement, l'instruction `dim` délivre un descripteur pour une variété à n dimensions à partir d'un descripteur pour une variété à n-1 dimensions et un descripteur pour la n^{ème} dimension.

Nous décrivons maintenant la construction d'un nouveau descripteur par l'instruction `dim`. Une instruction `dim` prend cinq opérandes :

src 1 : descripteur pour la variété à n−1 dimensions;

src 2 : descripteur pour la nème dimension;

borne inf. 1 : l'indice auquel commence la numérotation dans la nème dimension;

borne sup. 1 : l'indice le plus élevé dans la nème dimension;

borne inf. 2 : l'indice auquel commence la numérotation dans la nème-1 dimension.

Le résultat de l'instruction dim est obtenu par concaténation des μ vecteurs :

```
< src 1 > - < borne inf. 1 > + 1 + < src 2 > [ i ] - < borne inf. 2 > * < taille >
```

μ représentant le nombre de composantes du vecteur < src 2 > (l'indice i varie de 1 à μ).

Considérons une matrice *m* 10x10. Dans le code suivant,

```
  sinit    #2, #8, #2
n.assoc    &1, v1
  sinit    #3, #6, #1
n.assoc    &1, v2
  dim      v2, v1, #1, #10, #1
n.gath     m, &1, w
```

l'instruction dim produit un vecteur d'index sélectionnant les éléments des colonnes 3 à 6 et des lignes 2, 4, 6 et 8. Le vecteur *w* contient ces éléments de la matrice *m*.

1.4.10. Extensions "simples" de MAD

Nous avons regroupé dans cette section un certain nombre de remarques d'ordre général concernant l'état actuel des définitions de MAD et Devil, de leurs améliorations et extensions futures et quelques problèmes divers. Ces remarques feront l'objet d'études dans le futur.

Les structures

La prise en compte de variables structurées (enregistrements) des langages de programmation du type EVA, Pascal ou C est envisageable en Devil. De tels variables seraient décomposées en objets de base, chacun représentant un champ de la structure. La seule restriction à leur utilisation en Devil seraient, dans l'état actuel des choses, le renvoi par une fonction d'une structure et l'accès à la valeur des champs d'une structure renvoyée.

Les pointeurs

La manipulation des pointeurs des langages de haut niveau comme Pascal ou C est également absente des définitions de MAD et Devil. L'extension de leurs définitions permettant la gestion de pointeurs, l'allocation et la désallocation de zones mémoires dans le tas semble facile à réaliser. Elle nécessiterait l'adjonction d'une nouvelle catégorie d'objets dans Devil, les pointeurs, et la définition de primitives de base pour leur manipulation et la gestion de structures dynamiques.

La mise au point symbolique

L'utilisation réelle d'un langage de haut niveau pour le développement d'applications requiert la possibilité d'utiliser des outils de mise au point symbolique. Pour que la mise au point d'un programme puisse être réalisée au niveau du langage de haut niveau dans lequel il a été écrit, des informations doivent être passées du langage de haut niveau vers l'assembleur. Avec ces informations, l'assembleur génère une table des symboles qui permettra au logiciel de mise au point de donner des informations sur les symboles définis au niveau le plus haut dans le programme. Pour

permettre cela, il faut définir un jeu de primitives qui transmettront les informations nécessaires du langage de haut niveau, à l'assembleur, par l'intermédiaire de Devil. Ces primitives doivent être suffisamment générales pour recouvrir les objets de différents langages de haut niveau (EVA, C, Pascal, Fortran).

Les exceptions

L'exécution d'instructions dans un processeur produit des exceptions. Par ailleurs, des interruptions venant de l'extérieur du processeur peuvent survenir. De telles interruptions devraient pouvoir être gérées, c'est à dire prises en compte et une routine de traitement spécifique, définie par l'utilisateur, déclenchée lorsqu'un tel événement survient. Les interruptions sont intrinséquement très dépendantes de la machine cible, tant du point de vue de leur type, que du point de vue des événements les déclenchant. De ce fait, une définition générale et indépendante de la cible semble difficile à établir si l'on veut qu'elle recouvre les possibilités des différentes cibles.

2. Exemple de programme Devil

Après la description du langage Devil, nous donnons un exemple de programme exprimé en Devil afin de "fixer les idées". Nous nous basons sur un source écrit en EVA. Celui-ci utilise un minimum de spécificités de EVA afin d'en simplifier la compréhension. Une numérotation des lignes d'instructions a été ajoutée pour simplifier l'exposé.

Dans cet exemple, nous supposons qu'un scalaire entier est stocké dans un mot mémoire, un en-tête de vecteurs sur six mots mémoires.

Nous présentons ici l'aspect général d'un programme Devil. Nous nous intéressons essentiellement aux opérandes, à leur accès et aux attributs.

2.1. Source EVA

```
1:    int s;
2:    common s;

3:    vect int c;
4:    int g := 10;

5:    vect int f (b)
6:        vect int b;
7:            vect int d <- 100, e <- 100;
8:            /* ... initialisations des vecteurs d et e */
9:            % 100 % return b / e + (c - d) * s;
10:   end;
```

Les lignes 1 et 2 déclarent une variable scalaire entière de classe *common*, compatible avec Fortran.

La ligne 3 est la déclaration d'un vecteur d'entiers globale dont la visibilité est réduite au module où il est déclaré. La ligne 4 est celle d'un scalaire entier avec son initialisation.

La ligne 5 est la déclaration d'une fonction renvoyant un vecteur d'entiers. La ligne 6 déclare le paramètre comme un vecteur d'entiers. La ligne 7 déclare deux vecteurs locaux à la fonction. Une zone d'allocation leur est allouée à chacun, contenant 100 scalaires entiers.

Après divers traitements non explicités (ligne 8), la ligne 9 calcule une expression vectorielle dont le résultat est renvoyée par la fonction. % 100 % spécifie que le calcul est effectué sur les cent premiers éléments des vecteurs apparaissant dans l'expression.

2.2. Code Devil généré

Le code généré par le compilateur EVA est alors le suivant (la numérotation des lignes a également été ajoutée pour l'exposé) :

```
1:      ........ si          n.const      size_static, #7

2:    f:: f_loc, f_par
3:      vi??????             n.vinit      loc@(0)

4:      si       vi??????    n.alloc      #100, loc@(0)
5:      vi??????             n.vinit      loc@(6)
6:      si       vi??????    n.alloc      #100, loc@(6)

7:      si                   n.push       #100
8:      vi?????? vie.n...      div        par@(0), loc@(6)
9:      vi?????? vie.n...      sub        glo@(0), loc@(0)
10:     vie.n... si            mul        &1, s
11:     vie.n... vie.n...      add        &3, &1
12:     vie.n...             n.free       loc@(6)
13:     vie.n...             n.free       loc@(0)
14:     vie.n...               return     &3
15:                          n.pop

16:     vie.n...             n.free       loc@(6)
17:     vie.n...             n.free       loc@(0)
18:     ........ si          n.const      f_loc, #12
19:     ........ si          n.const      f_par, #6
20:                          n.endfct

21:                          n.debistat
22:     si       si          n.common     s, #1
23:     vi??????             n.vinit      glo@(0)
24:     si       si          n.move       #10, glo@(6)
25:                          n.finistat
```

Les trois champs d'une ligne d'instruction sont aisément distingués : la partie gauche donne les attributs des opérandes; au centre, on trouve les mnémoniques des instructions; à droite, il y a les opérandes. Le source se décompose en trois parties. Tout d'abord (ligne 1), une déclaration de la zone des données locales aux modules. Ensuite (lignes 2 à 20), on trouve le corps de la fonction *f*. Enfin (lignes 21 à 25), quelques instructions concernent les données globales du module et leur initialisation. Nous décrivons maintenant ces instructions de plus près.

Prologue à la traduction du module (ligne 1)

La ligne 1 déclare la taille du segment de données du module (variables *c* et *g*) sous la forme d'une constante dont le nom est réservé à cet usage (*size_static*). *c* étant un vecteur, un emplacement pour stocker son en-tête est réservé. Ses zones d'allocation et de description seront allouées par ailleurs dans le programme.

La traduction de la fonction *f* (ligne 2 à 20)

<u>L'en-tête de fonction</u>

Une fonction en Devil commence par la déclaration d'une étiquette portant son nom, le suffixe `::` indiquant qu'elle est visible à l'extérieur du module. Deux noms d'étiquettes suivent (`f_loc` et

f_par) auxquelles une valeur est assignée plus tard, à la fin du code de la fonction (lignes 18 et 19). Ces valeurs donnent les tailles des segments *local* et *param_rec* de la fonction.

Format des opérandes

Nous pouvons faire quelques remarques générales sur le format des opérandes. Un préfixe # indique une valeur immédiate. Les autres objets (hormis les commons et les temporaires) sont données sous la forme nom_de_segment@(offset). Les nom_de_segments utilisés ici sont loc pour un objet local à la fonction, par pour un paramètre de la fonction et glo pour un objet local au module. Les objets sont alloués dans la mémoire de MAD par le compilateur à la génération du code Devil. Les offsets sont alloués de manière simple : le premier objet de la classe rencontré dans le source est à l'offset 0, le suivant à un offset égal à la taille du premier objet, ... Les objets de classe common sont référencés directement via leur nom. Enfin, les objets temporaires sont référencés par un opérande &nnn.

Format des instructions

Pour ce qui est des instructions, nous rappelons que le préfixe n. indique une instruction ne générant pas de résultat temporaire. Nous remarquons que le calcul de l'expression est réalisé par des instructions générant des temporaires. L'instruction return renvoit la valeur d'un temporaire. Cette utilisation de temporaires indique les dépendances de flot entre les instructions. Ces dépendances pourront mener à un chaînage des unités fonctionnelles sur une architecture supportant ce mode de fonctionnement.

Le corps de la fonction

Les premières instructions de la fonction (lignes 3 à 6) initialisent les vecteurs locaux *d* et *e*. Ensuite, on trouve la traduction de l'instruction return de la fonction. A la ligne 7, la longueur sur laquelle les vecteurs sont traités est placée dans le registre VLG de MAD. Les instructions suivantes (lignes 8 à 11) calculent la valeur de l'expression. Celle-ci sera retournée après la libération des zones des vecteurs locaux (lignes 12 et 13). Les instructions après le return ne sont pas exécutées dans cet exemple. Elles sont générées de manière standard lors de la traduction de la fin de la fonction EVA. Un optimiseur peut supprimer la génération de ces instructions. Notons que l'instruction pop n'est pas exécutée. La valeur correcte de VLG est restaurée automatiquement au retour de la fonction. Parmi cet épilogue de fonction, les tailles des segments *local* et *param_rec* sont spécifiées par des pseudo-instructions const. Enfin, l'instruction endfct (ligne 20) indique la fin de la fonction.

Les attributs des opérandes

Les attributs des opérandes vectoriels sont intéressants à observer. Pour les opérandes scalaires, seul le type de l'objet est spécifié (ici, il est i pour tous les scalaires, indiquant que ce sont des entiers). Pour les vecteurs, les attributs donnent des renseignements relatifs à leur description et à leur zone d'allocation. Une fois initialisés (instructions alloc aux lignes 4 et 6), les vecteurs locaux ont des attributs spécifiés. Ainsi, à leur utilisation aux lignes 8 et 9, les attributs indiquent qu'il s'agit de vecteurs dont la zone d'allocation est étendue (ce n'est pas un triplet - indiqué par le e dans l'attribut) et qu'ils ont une zone de description vide (le n de l'attribut). Par contre, peu d'informations sont disponibles sur les vecteurs paramètres et common (cf. lignes 8 et 9).

La zone d'initialisation des données du module (ligne 21 à 25)

A la traduction d'un module en EVA, une zone de déclaration des commons et d'initialisations des variables du module est générée. Cette zone débute par une instruction debistat (ligne 21) et se termine par un finistat (ligne 25). On y retrouve la déclaration du common *s*, avec sa taille (1 mot mémoire). On y trouve une instruction d'initialisation des champs de l'en-tête du vecteur *c* (glo@(0)) et du scalaire entier *g* (glo@(6)).

Chapitre III
MAD et Devil - Projections

1. La projection de Devil

Dans cette section, nous étudions l'écriture d'un traducteur autorisant la génération d'un code de bonne qualité pour une cible donnée. Le programme généré doit utiliser au mieux les ressources offertes par l'architecture de la cible et les possibilités de parallélisme. Nous nous intéressons donc avant tout aux machines possédant plusieurs unités fonctionnelles capables de fonctionner concurremment. Nous décrivons tout d'abord le modèle d'exécution de MAD et présentons les notions fondamentales de *blocs vectoriels* et *blocs parallèles*. Ces notions sont fondamentales pour la génération de code vectoriel à partir de Devil. Nous nous intéressons alors à l'ordonnancement (dit *logique*) des instructions établi au seul vue des dépendances entre données. De cet ordonnancement, un treillis de dépendances entre les instructions est obtenu. Cette analyse logique est indépendante de la cible. Prenant en considération les contraintes matérielles, nous nous intéressons à la seconde étape, menant à la génération de code cible, l'ordonnancement dit *physique*. Celui-ci est obtenu en partant du treillis de dépendances et produit la séquence des instructions composant le programme en tenant compte des unités fonctionnelles disponibles. L'allocation des données dans les registres (vectoriels) est alors réalisée ce qui peut modifier localement l'ordonnancement physique. Par essence, ces deux dernières phases (ordonnancement physique et allocation des registres) sont dépendantes de la cible.

1.1. Le modèle d'exécution des instructions de MAD - Définitions

Le modèle d'exécution des instructions Devil par MAD tente de prendre en compte les possibilités d'activités parallèles supportées par les ordinateurs actuels. Le modèle veut recouvrir les différents types de parallélisme qui existent dans les différentes classes d'architectures. Concernant l'exécution en parallèle de deux instructions, il existe deux types de parallélisme :

- l'exécution en parallèle d'instructions indépendantes;
- l'exécution en parallèle de deux instructions dépendantes par chaînage d'unités fonctionnelles pipelinées.

Nous avons voulu rendre compte d'un parallélisme élevé, où seules les contraintes d'ordre logique (les dépendances entre données) sont considérées et limitent l'exécution en parallèle d'instructions. MAD possédant un nombre virtuellement infini d'unités fonctionnelles, aucune contrainte physique liée à l'occupation des unités fonctionnelles n'intervient. Au niveau logique, ce sont les dépendances entre données qui limitent le parallélisme d'une part, et ordonnent finalement l'exécution des instructions d'autre part.

1.1.1. Les dépendances entre données

L'exécution d'instructions en parallèle doit prendre en compte les trois types de dépendances existants entre données. Notons :

IN (I) l'ensemble des objets dont la valeur est utilisée comme source par l'instruction I, et

OUT (I) l'ensemble des objets dont la valeur est modifiée par l'instruction I.

Dans ce qui suit, I2 est une instruction qui suit I1 de manière non nécessairement consécutive et les vecteurs utilisés sont tous indépendants les uns des autres. Les trois types de dépendances sont (cf. par exemple [PADUA & al. 86]) :

– dépendance de flot (notée δ). Une dépendance de flot intervient quand le résultat de I1 et utilisé comme donnée source de l'instruction I2, soit IN (I2) $\cap$ OUT (I1) $\neq \emptyset$:

```
I1        a <- b + c
I2        d <- a + e
```

où <- représente l'affectation. L'instruction I2 doit alors utiliser la valeur de *a* qui vient d'être calculée dans I1, et non pas sa valeur précédente.

– dépendance de résultat (notée $\delta°$). Une dépendance de résultat intervient quand I1 et I2 ont la même cible, soit OUT (I2) $\cap$ OUT (I1) $\neq \emptyset$:

```
I1        a <- b + c
I2        a <- d + e
```

Lorsque ces deux instructions ont été exécutées, *a* doit contenir la valeur de la somme de *d* et *e*.

– antidépendance (notée δ^{-1}). Une antidépendance intervient quand la cible de I2 et l'un des opérandes source de I1, soit OUT (I2) $\cap$ IN (I1) $\neq \emptyset$:

```
I1        b <- a + c
I2        a <- d + e
```

La valeur utilisée dans I1 doit être la valeur de *a* avant l'exécution de I2.

L'approche classique consiste à vectoriser un source scalaire (Fortran ou autre). Le vectoriseur fournit un graphe de dépendances des données duquel l'ordonnancement effectif des instructions est dérivé. Les dépendances mènent à la construction du treillis de dépendances dont l'exploitation permet d'ordonner logiquement les instructions.

1.1.2. Blocs vectoriels - Blocs parallèles

Nous introduisons ici deux constructions Devil fondamentales au regard des instructions vectorielles et de leur futur ordonnancement, les *blocs vectoriels* et les *blocs parallèles*. Les premiers ne sont que des constructions syntaxiques. Les seconds sont chargés de sémantique. Les premiers peuvent s'imbriquer, pas les seconds.

Les blocs vectoriels

Les blocs vectoriels sont simplement des blocs d'instructions dans lesquels les traitements vectoriels sont réalisés sur une taille de vecteurs fixe. La taille du bloc vectoriel est connue ou non à la compilation. Un bloc vectoriel peut s'imbriquer dans un autre dont la taille est différente. La valeur courante de VLG est alors localement surchargée (cf. le fonctionnement du registre VLG et du segment *vlg* § II.1.3.1.). Cette surcharge est réalisée par l'exécution d'une instruction push. La restauration de l'ancienne valeur de VLG est effectuée par l'instruction pop. De fait, un bloc vectoriel est un bloc d'instructions précédé par une instructions push et se terminant par une instruction pop. Les blocs vectoriels sont partitionnés en blocs parallèles. Les instructions de calcul sur les vecteurs (les instructions vectorielles calculatoires ou les instructions de réduction – cf. infra § II.1.4.3.) sont forcément localisées dans un bloc vectoriel. Seules des instructions scalaires (incluant les instructions de contrôle du flot d'instructions) ou certaines instructions vectorielles spéciales (dites instructions mixtes d'accès aux champs des en-têtes de vecteurs – cf. infra § II.1.4.3.) peuvent se trouver en dehors d'un bloc vectoriel.

Les blocs parallèles

Les blocs parallèles définissent une partition de l'ensemble des instructions d'un bloc vectoriel. Les instructions d'un bloc parallèle sont successives dans le bloc vectoriel. Les blocs parallèles sont exécutés les uns à la suite des autres, dans l'ordre où ils sont rencontrés dans le bloc vectoriel. A un instant donné, un seul bloc parallèle est en cours d'exécution (un bloc parallèle est en cours d'exécution si au moins l'une de ses instructions est en cours d'exécution). Un bloc parallèle est constitué d'un ensemble d'instructions s'exécutant virtuellement en parallèle, entre lesquelles n'existent que des dépendances de flot. Entre deux blocs parallèles d'un même bloc vectoriel peuvent exister toutes sortes de dépendances. Aussi, c'est pour respecter la sémantique du programme que les blocs parallèles sont exécutés dans l'ordre, les uns après les autres, sans recouvrement. La figure III.1. indique l'imbrication des blocs ainsi que les dépendances existant dans et entre les différents types de blocs.

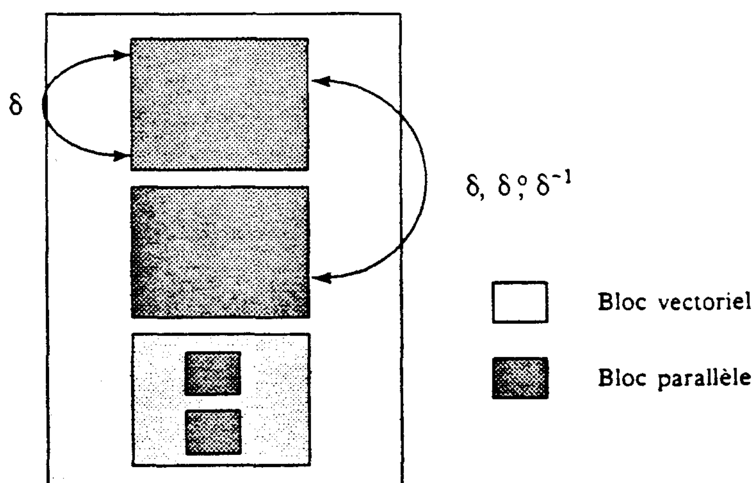


Figure III.1. –
Blocs vectoriels et blocs parallèles

L'exécution des instructions vectorielles est subordonnée aux *blocs parallèles* de Devil. Un bloc parallèle débute par une instruction push (début de bloc vectoriel), wait (instruction de synchronisation), call (appel de fonction) ou return (retour de fonction), et se termine par une instruction wait, push, pop (fin de bloc vectoriel), call ou return.

Par définition, il ne peut exister d'antidépendance ou de dépendance de résultat dans un bloc parallèle. Par contre, il peut exister des dépendances de flot. Les instructions entre lesquelles existent des dépendances de flot sont alors virtuellement chaînées. L'instruction, ou les instructions, en début de chaîne commencent leur exécution. Les instructions suivantes sont également déclenchées mais demeurent bloquées tant que les premières n'ont pas délivré de résultat (tout ou partie du résultat pour rendre compte du chaînage de pipelines).

Une instruction (`wait`) indique une fin de bloc parallèle (et le début du suivant). Son exécution force la terminaison de toutes les instructions du bloc parallèle précédent, avant le déclenchement des instructions du bloc suivant (qui seront donc, elles aussi, déclenchées en parallèle). L'instruction `wait` résout les problèmes de dépendances de résultat et les antidépendances. Deux instructions I1 et I2 telles que I1 δ^{o} I2 ou I1 δ^{-1} I2 doivent être séparées par une instruction `wait` pour que le résultat de leurs exécutions soit conforme à la sémantique que l'on en attend.

Pour terminer cette présentation, considérons l'exemple suivant

I1. $\vec{a} := \vec{b} + \vec{c}$

I2. $\vec{d} := \vec{e} + 3 \times \vec{f}$

Si tous les vecteurs $\vec{a}$, $\vec{b}$, $\vec{c}$, $\vec{d}$, $\vec{e}$ et $\vec{f}$ sont indépendants, ces deux instructions se traduisent dans le bloc parallèle :

```
        n.add    b, c, a
D1.       mul    3, f
D2.     n.add    &1, e, d
```

Il existe une dépendance de flot entre les deux instructions D1 et D2.

S'il existe une antidépendance dans l'instruction I2 (par exemple, il y a une récurrence et elle est la traduction de $A_{i+1} = \vec{e} + 3 \times A_i$, pour tous les i, les vecteurs $\vec{d}$ et $\vec{f}$ étant produits par l'association de la zone d'allocation de A et des descripteurs adéquats), elle se traduit en :

```
n.add    b, c, a
  mul    3, f
  add    e, &1
n.wait
n.move   &2, d
```

Cette traduction est celle qui est effectuée par le compilateur EVA. D'autres traductions sont possibles.

1.2. Ordonnancement logique

1.2.1. Principe de l'ordonnancement logique des instructions

Pour respecter la sémantique d'un programme Devil, une analyse des blocs parallèles est effectuée. Pour un bloc parallèle, un treillis de dépendances est construit sur l'ordre partiel défini par, I1 et I2 étant des instructions d'un même bloc parallèle :

$$I1 < I2 \iff I2 \ \delta \ I1.$$

Pour deux instructions, I1 < I2 signifie en fait que I1 doit être réalisée avant I2. L'ordre est partiel, des instructions pouvant être indépendantes l'une de l'autre, donc non comparables. Les arcs du treillis indiquent les dépendances δ. Considérons les deux instructions :

$$\vec{a} := (\vec{b} * \vec{c}) + (\vec{e} - \vec{f} * \vec{g})$$
$$\vec{h} := 3 / \vec{b} + 4$$

On suppose qu'aucune dépendance n'existe entre les données. Elles se compilent, en pseudo-Devil, dans un bloc parallèle dont le corps est :

```
n.push   < taille >
    mul  b, c
    mul  f, g
    sub  e, &1
  n.add  &3, &1, a
    div  #3, b
  n.add  &1, #4, h
n.pop
```

Ce bloc parallèle est transformé en un treillis de dépendances logiques (cf. fig. III.2.) en analysant simplement les dépendances de flot. Celles-ci interviennent forcément au niveau des objets temporaires et sont faciles à retrouver.

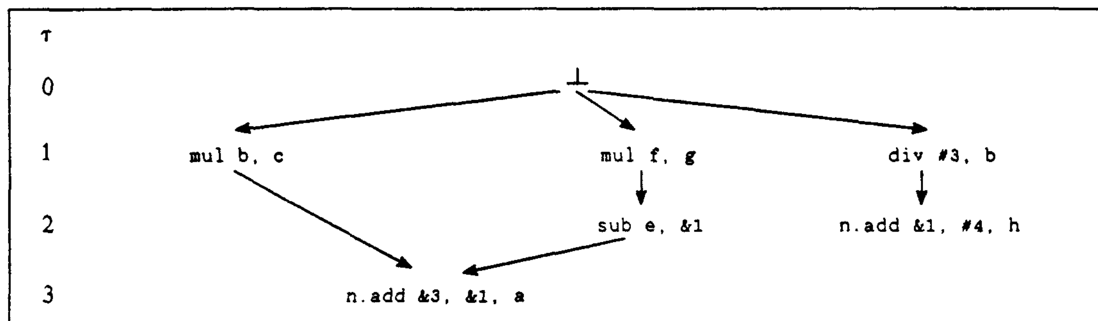


Figure III.2. -
Treillis de dépendances logiques

D'après ce treillis, on constate que logiquement parlant, le déclenchement de l'ensemble des instructions du bloc parallèle s'effectue en trois temps.

A la racine du treillis (qui se trouve en haut sur la figure III.2.) se trouve une instruction notée $\perp$ dont l'objet est soit de mettre à jour la valeur du registre VLG, soit d'attendre la terminaison de l'exécution du bloc parallèle précédent. Toutes les autres instructions du bloc parallèle dépendent logiquement (directement ou indirectement) de cette instruction.

A toute instruction I est associé son *numéro d'ordre de déclenchement logique* τ (I). τ est une application et est définie de l'ensemble des instructions d'un bloc parallèle (noté $\mathfrak{I}$) dans N. Cet ordre ne dépend que des dépendances entre données. L'application se définit aisément par récurrence. Pour une instruction $\imath$, $\tau(\imath)$ se définit par rapport aux instructions I_i (i variant de 1 à n) dont elle dépend ($\forall\ 1 \leq i \leq n$, $\imath < I_i$) par :

$$\tau(\imath) = \mathrm{Max}\ (\ \{\ \tau(I_i),\ I_i \in \mathfrak{I},\ I_i < \imath \text{ et } 1 \leq i \leq n\ \}\ ) + 1$$
$$\text{et } \tau(\perp) = 0$$

Ce treillis est ensuite modifié afin de lever les choix de déclenchement, en prévision d'architectures réelles n'autorisant pas effectivement l'exécution concurrente de toutes instructions d'un bloc parallèle. Ainsi, dans le treillis précédent, si un choix doit être fait quant à lancer `mul b, c` ou `mul f, g`, il faut donner la priorité de déclenchement à la seconde. En effet, son résultat est utilisé par une instruction dont l'ordre de déclenchement est inférieur à l'instruction utilisant le résultat de l'autre instruction `mul`.

En généralisant cette remarque, une opération dite de "justification par le bas" du treillis est réalisée en modifiant l'ordre de déclenchement des instructions (cf. fig. III.3.). On obtient alors

l'application τ', toujours de l'ensemble $\mathfrak{I}$ des instructions d'un bloc parallèle dans **N**, qui se déduit de l'application τ. Pour une instruction ı, on a :

$$\tau'(\imath) = \mathrm{Min}\ (\ \{\ \tau\ (I),\ \forall\ I \in \mathfrak{I},\ I_i > I \text{ et } I < \imath\ \}\ ) - 1$$
$$\text{et } \tau'(\imath) = \tau(\imath) \text{ si } \nexists\ I \in \mathfrak{I} \text{ et } \imath < I$$

τ'(ı) est l'*ordre de déclenchement logique effectif* de l'instruction ı.

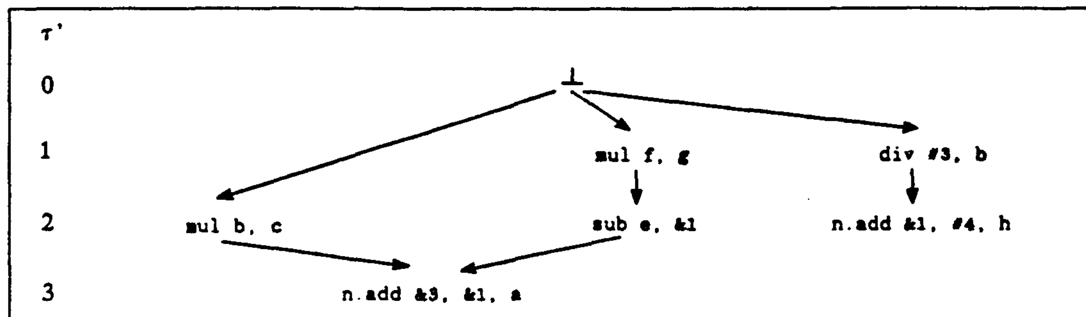


Figure III.3. -
Treillis de dépendances logiques modifié ("justifié vers le bas")

A partir de ce treillis modifié, la séquence de code pour le bloc parallèle est ordonnée. On obtient pour cet exemple :

```
  mul   f, g
  div   #3, b
  mul   b, c
  sub   e, &3
n.add   &3, #4, h
n.add   &5, &2, a
```

Notons que les références aux temporaires ont été modifiées pour demeurer cohérentes avec ce nouvel ordonnancement des instructions.

Considérant l'ordre de déclenchement logique effectif et prenant en compte les contraintes d'ordre physique (liées aux spécificités de la machine cible), l'ordre de déclenchement physique sera obtenu et le code généré. L'ordonnancement physique ne sera pas développé dans ce document. Il fait actuellement l'objet d'études pour la génération de code pour des machines autorisant du parallélisme réel, principalement en ce qui concerne le Cray Y-MP (activité concurrente de plusieurs unités fonctionnelles dans un processeur). Il utilise des techniques proches de la compaction de micro-code (cf. [TOKORO & al. 81]). Ces techniques ont été utilisées avec succès pour la génération de code pour le Cray-2 (cf. [EISENBEIS & al. 90]).

1.2.2. Remarques sur les dépendances et le chaînage

Notons qu'il n'y a dépendance de flot effective (et donc chaînage) entre deux instructions que si le résultat de la première est un temporaire qui est réutilisé par la seconde instruction. En aucun cas, le chaînage ne passe par un objet non temporaire. Ainsi, le corps de bloc parallèle suivant :

```
n.push   < taille >
  n.add  b, c, a
  n.move a, d
n.pop
```

est sémantiquement incorrect (il n'a pas de sens : son résultat est imprévisible). En effet, ses deux instructions doivent être lancées en parallèle. Or, le move utilise la valeur de *a* qui est modifiée par

l'instruction précédente. Aussi, le résultat est imprévisible. Ces deux instructions ne doivent pas se trouver dans un même bloc parallèle. Il aurait fallu écrire :

```
n.push    < taille >
  n.add   b, c, a
  n.wait
  n.move  a, d
n.pop
```

Dans ce code (composé de deux blocs parallèles), l'addition sera réalisée et la valeur de *a* affectée à *d* sera celle du résultat de l'addition.

Notons qu'une meilleure version serait :

```
n.push    < taille >
    add   b, c, a
  n.move  &1, d
n.pop
```

qui autorise le chaînage des deux instructions.

Selon le cas, des vecteurs se partageant la même zone d'allocation peuvent ou ne peuvent pas se trouver dans un même bloc parallèle, selon qu'il y a partage réel de certaines composantes de la zone entre les deux vecteurs ou non. Ainsi, le bloc parallèle suivant est sémantiquement correct (les triplets des instructions `dassoc` sont en fait obtenus par des instructions `sinit`) :

```
n.push      < taille >
  n.dassoc  v, [ #1 : #10 : #2 ], w1
  n.dassoc  v, [ #2 : #10 : #2 ], w2
  n.move    #-1, w1
  n.move    #0, w2
n.pop
```

w1 et *w2* se partagent la même zone d'allocation. Cependant, aucun élément de cette zone n'est sélectionné par le sélecteur des deux vecteurs. Donc, il n'y a pas de dépendance entre les deux instructions `move`. Celles-ci peuvent s'exécuter en parallèle. Les éléments d'indice pair reçoivent la valeur 0; les éléments d'indice impair prennent la valeur -1.

1.3. Conséquences de la structure de Devil sur l'analyse physique

A la suite de l'analyse logique qui vient d'être présentée, une analyse physique est effectuée. A son issue, le code exécutable aura été généré. Cette analyse prend en compte toutes les spécificités de l'architecture cible afin d'obtenir un code de bonne qualité. Elle se compose de trois phases successives. La première est celle de génération de code; la deuxième ordonne les instructions; la troisième alloue les registres.

La génération utilise les techniques d'expansion de macros. A une instruction Devil correspond un schéma de macro constitué d'une suite d'instructions de la machine cible. L'expansion est fonction du type des opérandes et des caractéristiques exactes de l'instruction (décrite, génération de temporaire). Elle est également fonction des instructions précédemment expansées. Les opérandes des instructions sont placés dans des registres dits virtuels. C'est à dire que l'on utilise ici les différents types de registres disponibles sur la cible, mais en supposant qu'il y en a un nombre illimité.

La phase d'allocation des registres consiste à transformer les registres virtuels en registres réels. Selon le nombre de registres virtuels en utilisation à un moment donné, les registres réels seront en nombre suffisant ou pas. S'ils ne sont pas assez nombreux, des opérations de transferts de contenu

de registres en mémoire doivent être réalisées. Pour cela du code est ajouté qui peut modifier localement l'ordonnancement des instructions.

Dans ce qui suit, nous donnons quelques indications sur ce que la structure de Devil apporte au niveau de la traduction. Afin de simplifier l'exposé, nous avons supposé que nous étudiions la génération de code pour une machine pipeline fonctionnant de registre à registre et traitant des vecteurs de 64 éléments (registres vectoriels de 64 éléments). Toute référence à ce nombre est en rapport avec cette remarque.

1.3.1. Les objets

Les vecteurs

Limitation du nombre de gather/scatter

La forme des vecteurs en Devil (association d'une zone de description à la zone d'allocation) représente une factorisation des opérations de sélection des éléments d'un vecteur. Dès qu'un vecteur est décrit plus d'une fois à l'aide d'un même descripteur, il est avantageux de réaliser une association. De cette manière, le code généré pourra plus facilement minimiser les accès indirects aux éléments du vecteur à l'exécution. Dans ce cas, un vecteur contigu contenant les éléments du vecteur décrit est créé puis utilisé à la place du vecteur décrit. S'il ne partage pas sa zone d'allocation avec un autre vecteur, cet utilisation du vecteur contigu ne posera pas de problème. Toutes les opérations sur le vecteur peuvent être réalisées en fait sur ce vecteur contigu. L'écriture des éléments du vecteur contigu dans la zone d'allocation du vecteur initial n'a à être effectuée que lors de l'utilisation de cette zone d'allocation dans une opération d'association. Par contre, si la zone d'allocation est partagée, tout dépend des éventuelles dépendances entre vecteurs (les deux vecteurs ont-ils des éléments effectifs communs ?). Ces dépendances sont reflétées par le découpage des blocs vectoriels en blocs parallèles. Aussi, dans un bloc parallèle, tout va pour le mieux (pas de dépendances). Par contre, à la fin d'un bloc parallèle, les morceaux de vecteurs doivent être replacés dans la zone d'allocation du vecteur. Au début d'un bloc parallèle, les morceaux de vecteurs utilisés dans le bloc sont chargés.

L'allocation des vecteurs en mémoire

A l'exception de certains passages en paramètre de vecteurs (passages de paramètres `parin` et `parconst` cf. § II.1.3.3.), il n'y a pas de création dynamique de vecteurs. Aussi, les en-têtes des vecteurs sont pré-alloués à la compilation. Leurs adresses en mémoire sont calculées à la compilation. Pour les vecteurs globaux à un module, leur en-tête est placé dans le segment de données du module. Pour les vecteurs locaux à une fonction, un emplacement leur est réservé dans le segment des locaux de la fonction. De même, la zone d'informations est allouée ou pré-allouée à la compilation. La zone d'allocation l'est également si sa taille est connue à la compilation. Une instruction `setza` permettra de relier l'en-tête à sa zone d'allocation et d'informations durant l'exécution. Les allocations statiques de mémoire diminue l'"overhead" du à la gestion des vecteurs Devil.

L'allocation des registres dans les blocs parallèles

A chaque opérande vectoriel est associé un registre vectoriel virtuel. Du code généré en début de bloc parallèle charge ses registres avec des morceaux de vecteurs. Du code généré en fin de bloc parallèle écrit les registres des morceaux des vecteurs.

L'allocation des registres inter-blocs parallèles

Si un bloc vectoriel est de taille inférieure ou égale à 64 (c'est à dire que les vecteurs sont traités sur moins de 64 éléments), de nombreuses optimisations peuvent être mises en jeu. Les vecteurs ne

sont alors composés que d'un seul morceau. De fait, on peut ne travailler qu'avec des registres virtuels. Le contenu des registres modifiés dans le bloc vectoriel sont écrits en mémoire à la fin de l'exécution du bloc vectoriel. Aussi, les registres virtuels ne sont pas alloués localement dans chaque bloc parallèle. Au contraire, une fois alloué à un vecteur, un registre lui reste associé durant l'exécution de l'ensemble du bloc parallèle.

Si la taille du bloc vectoriel est connue à la compilation, mais supérieure à 64, des optimisations restent possibles. Le nombre d'itérations à réaliser est connu à la compilation. Le contrôle de la boucle en est donc simplifié. Si le nombre d'itérations est faible (inférieur ou égal à 128 par exemple), la boucle peut être déroulée. Ceci supprime les instructions de contrôle. Ceci permet une meilleure utilisation des registres vectoriels.

Les temporaires

Nous étudions ici la gestion des temporaires et les classons dans deux catégories différentes : ceux qui sont utilisés localement dans le bloc parallèle où ils ont été produits, et ceux qui sont utilisés dans un bloc parallèle différent de celui où ils ont été produits. Par ailleurs, nous disons qu'un temporaire est *vivant* pendant l'exécution des instructions qui se trouve entre l'instruction qui l'a créé et la dernière instruction l'utilisant.

L'allocation des registres dans un bloc parallèle

Un temporaire créé et consommé dans le bloc parallèle est mis de préférence dans un registre. Dans un bloc parallèle, il n'y a pas de dépendance inter-itération. Aussi, un morceau de vecteur temporaire produit dans une itération sera utilisé dans la même itération, et pas une suivante. Aussi, si un temporaire n'est utilisé que localement dans le bloc parallèle où il a été produit, seul le morceau courant est utile; les autres n'ont pas a être mémorisés. Aussi un registre vectoriel lui est alloué. Par ailleurs, le contenu de ce registre n'aura pas à être écrit en mémoire à la fin du corps de la boucle pour un usage futur.

Les temporaires dont la vie s'étend sur plusieurs blocs parallèles

Pour les temporaires dont la durée de vie s'étend sur plusieurs blocs parallèles, le problème est très différent selon que le bloc vectoriel est de taille inférieure ou égale à 64 ou non, ou si la taille n'est pas connue à la compilation.

Si la taille du bloc vectoriel n'est pas connue à la compilation, le temporaire est mis en mémoire puisque l'on ne connaît pas d'avance sa taille. Une zone lui est allouée avant l'exécution de l'instruction le produisant. Ses morceaux y sont rangés les uns après les autres, à chaque itération de la boucle correspondant au bloc parallèle où il est créé. Ils seront rechargés depuis la mémoire dans le (ou les) blocs parallèles qui l'utiliseront. Ainsi, le bloc vectoriel suivant :

```
                                      ...
si                                  n.push        < variable >
vie.di..        vie.di..              add         v, w
                                    n.wait
vie.....        vie.di..            n.move        &2, v
                                    n.pop
                                      ...
```

se traduira dans une structure du type :

```
        < allocation espace du temporaire >
debut_bloc_par_1:
        < charger strips de v et w dans des registres >
        < calculer la somme de ces strips >
        < écrire strip résultant en mémoire >
        < contrôle de la boucle >
debut_bloc_par_2:
        < charger strip du temporaire >
        < l'écrire dans le vecteur v en mémoire >
        < contrôle de la boucle >
```

Si la taille est inférieure ou égale à 64, un temporaire peut être placé dans un registre et y être laissé tant qu'il est vivant (si le registre où il se trouve alloué n'est pas nécessaire pour une autre opération). On gagne ainsi les accès à la mémoire.

Si la taille est connue, mais supérieure à 64, la gestion des temporaires ressemblera à celle exposée précédemment quand la taille est inconnue : le temporaire devra être placé en mémoire. Seulement, l'adresse de la zone de stockage du temporaire sera connue à la compilation. Par ailleurs, certaines optimisations peuvent être mises en jeu du fait que la taille est connue à la compilation (cf. infra).

Gestion de la mémoire dans un bloc vectoriel

Dans un bloc vectoriel, le nombre de zones d'allocation nécessaires pour les vecteurs temporaires est limité au nombre maximum de vecteurs temporaires vivants à un moment donné dans le bloc vectoriel. Par ailleurs, seul le nombre des vecteurs utilisés dans un bloc parallèle différent de celui où ils sont produits importe. Les autres n'ont pas à être écrits en mémoire. Dès qu'un temporaire n'est plus utilisé, sa zone peut être utilisée par un autre vecteur temporaire.

1.3.2. Les blocs vectoriels et les blocs parallèles

Les blocs vectoriels

La taille du bloc vectoriel est affectée au registre indiquant la longueur des vecteurs (VL sur Cray). Sur Cray, un appel de fonctions n'utilise pas de pile de sauvegarde de contexte(11). Aussi, il sera intéressant que la sauvegarde de l'ancienne valeur de VLG soit réalisée non pas en mémoire, mais également dans une partie du bloc de registres B alors gérée en pile. Une sauvegarde en mémoire ne sera réalisée qu'au cas où un grand nombre de blocs vectoriels sont imbriqués les uns dans les autres. Une telle stratégie de sauvegarde de contexte possède de multiples avantages. D'une part, on évite un accès à la mémoire, donc d'éventuels conflits. Par ailleurs, un accès scalaire à la mémoire devrait être effectué. Hors, un accès scalaire ne peut avoir lieu concuremment à un accès vectoriel et bloque donc les autres ports mémoires. Tout ceci entraîne un temps de commutation de contexte beaucoup plus faible avec notre solution.

Si la taille du bloc vectoriel est connue à la compilation et est inférieure à 64, de nombreuses optimisations sont mises en place. Les vecteurs traités ne comportent qu'un seul morceau. Aussi, les blocs parallèles se traduisent dans de simples séquences d'instructions, et non pas en une succession de boucles. Tous les éléments des vecteurs sont contenus dans un registre vectoriel.

(11) - Nous indiquons par là que les instructions CAL n'effectuent ni sauvegarde en mémoire centrale lors d'un appel de sous-programme, ni restauration depuis la mémoire centrale au retour d'un sous-programme.

Dans le cas où cette taille est connue et supérieure à 64, certaines optimisations peuvent encore être mises en place. Le bloc parallèle se traduit dans une boucle dont le corps est la traduction des instructions du bloc parallèle, en ne traitant qu'un morceau de vecteurs à chaque itération. Le nombre d'itérations qui doivent être effectuées est connu à la compilation. Le contrôle de la boucle en est donc légérement simplifié. En ce qui concerne le déroulage de la boucle, des tests peuvent être supprimés par rapport au cas où la taille est inconnue. Ainsi, supposons que la boucle

```
boucle:
        < instruction 1 >
        < instruction 2 >
        < instruction n >
        < contrôle de la boucle >
```

puisse se dérouler une fois, un test de terminaison doit être inséré entre les deux corps de boucle (ligne discontinue) si la taille du bloc vectoriel est inconnue, ce test pouvant être supprimé si la taille est connue (la valeur de ce test est alors calculable à la compilation) :

```
boucle_deroulée:
        < instruction 1 >
        < instruction 2 >
        < instruction n >
      - - - - - - - - - -
        < instruction 1 >
        < instruction 2 >
        < instruction n >

        < contrôle de la boucle >
```

Si le nombre d'itérations est impair, les instructions du corps de la boucle sont ajoutées en sortie de boucle pour y être exécutée une dernière fois.

Les blocs parallèles

Les blocs parallèles d'un bloc vectoriel sont traduits les uns après les autres. Selon le cas, un bloc parallèle est traduit en une boucle ou en une simple séquence d'instructions.

Cas où la taille des vecteurs dans le bloc vectoriel est connue à la compilation

Si la taille du bloc vectoriel est inférieure ou égale à 64, les opérations vectorielles n'ont pas à être découpées. Le bloc parallèle est alors transformé dans une séquence d'instructions. Si la taille du bloc vectoriel est supérieure à 64, une boucle est construite autour des instructions du bloc parallèle. Le nombre d'itérations de la boucle est alors connue à la compilation.

Cas où la taille des vecteurs dans le bloc vectoriel est inconnue à la compilation

Grâce à la structure des blocs parallèles, la génération directe d'un code de bonne qualité est plus simple lors du découpage (*strip-mining*). Au lieu de découper une instruction à la fois, toutes les instructions d'un bloc parallèle sont placées dans le corps de la même boucle. Aussi, soit le bloc parallèle suivant :

```
n.push      < taille >
  n.add     a, b, c
  n.mul     d, e, f
n.pop
```

La traduction simple pour une machine registre à registre en serait :

```
        cpt := val. init.
    boucle 1:
        charger les morceaux (strips) de a et b en registres
        add a, b, c
        stocker morceau de c en mémoire
        cpt --
        si cpt < nombre de morceaux alors aller boucle 1

        cpt := val. init.
    boucle 2:
        charger les morceaux de d et e en registres
        mul d, e, f
        stocker morceau de f en mémoire
        cpt --
        si cpt < nombre de morceaux alors aller boucle 2
```

Une traduction plus efficace en est :

```
        cpt := val. init.
    debut:
        charger les morceaux de a, b, d et e en registres
        add a, b, c
        mul d, e, f
        écrire les morceaux de c et f en mémoire
        cpt --
        si cpt < nombre de morceaux alors aller debut
```

Les avantages sont simples : d'une part, il n'y a qu'une seule boucle au lieu de deux en séquences. On gagne donc le temps du contrôle de la structure itérative. D'autre part dans la deuxième version, et ceci est beaucoup plus important sur des machines supportant le fonctionnement en parallèle de leurs unités fonctionnelles, l'addition et la multiplication seront effectuées en parallèle et non pas séquentiellement comme dans la première version. La traduction sous la deuxième forme d'un bloc parallèle Devil est très simple. Concernant leur génération en Devil à partir d'un source EVA, la reconnaissance des blocs parallèles en EVA est simple. Des structures syntaxiques les indiquent.

1.3.3. Les optimisations standards

Nous étudions ici quelques optimisations couramment appliquées par les compilateurs qui peuvent être effectuées au niveau d'une analyse de source Devil.

Optimisations à lucarne

L'optimisation d'un code intermédiaire entraîne la réalisation d'optimisations indépendantes de la cible (à l'aide des techniques d'optimisation à lucarne). La définition d'un tel optimiseur simplifie les traducteurs qui n'ont plus à inclure cette analyse générique pour toutes les cibles ([TANENBAUM & al. 82] [BAL & al. 82]). L'étude d'un optimiseur de code Devil pourra faire l'objet de travaux ultérieurs. Les optimisations effectuées à ce niveau peuvent viser, entre autres :

- l'élimination des instructions inutiles (ajouter 0 à la valeur d'un objet ou multiplier par 1 la valeur d'un objet),
- l'uniformisation de la réalisation de certains calculs (une multiplication par −1 est remplacée par une instruction opp produisant la valeur opposée d'un objet),
- l'extraction d'instructions constantes d'une boucle,
- l'élimination des instructions inaccessibles, ...

Notons que tout ré-ordonnancement de code doit être réalisé à ce niveau avec beaucoup de circonspection. Etant donnée la structure des vecteurs et de multiples aspects liés à la projection sur machines réelles, certaines optimisations ne peuvent être réalisées à ce niveau, trop élémentaire.

Problèmes des appels de fonctions

Lors du passage en paramètre d'un vecteur, les attributs de celui-ci ne sont pas transmis. Pour éviter cette perte d'informations dans la mesure de ce qui est prévisible à la compilation, des analyses sont mises en place. En parallèle, des analyses visent à spécifier les attributs du vecteur retourné par une fonction.

Une première technique permettant la levée des problèmes de détermination des attributs des opérandes est la mise en ligne de la fonction. Cette mise en ligne est particulièrement intéressante dans le cas d'une fonction dont le corps est court et qui est appelée avec des paramètres dont les attributs sont connus à la compilation et sont différents les uns des autres. En effet, dans ce cas, le corps de la fonction peut être optimisé pour chacun des appels, alors que pour l'ensemble des appels, ce corps devrait conserver une forme générale. Aussi, sa mise en ligne sera grandement appréciée : le code mis en ligne traitera chacun des cas spécifiques d'appel au mieux.

```
                        ...

vie.di..           n.parout    vi
vie.di..           n.parout    wi
vi......vi??????   call        f, xi

                        ...

vie.db..           n.parout    vb
vie.db..           n.parout    wb
vi......vi??????   call        f, xb

                        ...

f::
 -- on suppose que a et b sont les
 -- paramètres formels de la fonction f
vi?????? vi?????? add          a, b
vi??????           return      &1
```

```
                              ...

vie.di..  vie.di..  vie.di..  n.add   vi, wi, xi

                              ...

vie.db..  vie.db..  vie.db..  n.add   vb, wb, xb

                              ...
```

Dans l'exemple précédant, la partie gauche indique le code tel qu'il a été généré par le compilateur EVA (ou un vectoriseur). On voit deux appels d'une même fonction, avec des paramètres vectoriels ayant des attributs différents (vecteurs d'entiers décrits par une liste d'entiers pour le premier appel, une liste de bits pour le deuxième appel). La fonction renvoit simplement en résultat la somme des deux paramètres. Dans le corps de *f*, les vecteurs ont des attributs indéterminés. En effet, la déclaration des paramètres indique uniquement qu'il s'agit de vecteurs d'entiers, sans informations sur leurs autres attributs. Aussi, des tests devraient être mis dans le code généré afin de déterminer le type exact des opérandes paramètres. L'analyse du source indique que la fonction est appelée avec des paramètres dont les attributs sont différents d'un appel à l'autre. Aussi, on ne peut pas simplement retirer ces tests. On peut dupliquer le corps de la fonction, chaque copie de la fonction correspondant à un type déterminé de paramètres. On peut également mettre la fonction en ligne. Cette solution possède les qualités habituelles de la mise en ligne d'une fonction : les changements de contexte et tout le mécanisme d'appel de fonction sont évités. Par ailleurs, sur un processeur possédant des tampons d'instructions, cela permet d'éviter le chargement d'un tampon d'instructions avec le corps de la fonction. La partie droite indique alors

le résultat de la mise en ligne de la fonction *f*. Les attributs des opérandes sont alors spécifiés. La génération de code pourra être meilleure en évitant de nombreux tests.

Si la fonction est appelée plusieurs fois avec des vecteurs dont les attributs sont les mêmes et que son corps est relativement long, elle pourra ne pas être mise en ligne, mais les tests à l'intérieur de son corps pourront cependant être évités.

Par ailleurs, la mise en ligne d'une fonction (ou du moins son analyse) permet d'éviter la perte des attributs des paramètres après l'appel de fonction. En effet, les paramètres étant passés en mode lecture/écriture (parout), la fonction appelée peut modifier leur valeur. Ainsi dans l'exemple précédent, après l'appel de la fonction *f*, les paramètres *vi*, *wi*, *vb* et *wb* ont, à priori, des attributs indiquant une description indéterminée. Hors, la fonction *f* ne modifie par la valeur de ses paramètres. Les paramètres d'appels conservent donc les mêmes attributs après l'appel. Par ailleurs, à moins que la fonction ne réalise une opération d'association du paramètre, les paramètres d'appels conservent les mêmes attributs, au moins au niveau de la description. La spécification des attributs des paramètres après un appel de fonction a donc de grosses conséquences sur la suite de la génération de code.

2. L'implantation de Devil

Nous commençons cette partie par quelques remarques générales sur la génération de code. Nous nous intéressons alors à l'implantation d'un prototype de traducteur Devil sur station de travail Sun équipée d'un processeur Motorola 68020 et d'un co-processeur 68881 pour le calcul sur nombres flottants.

Interprétation ou compilation ?

Pour l'implantation d'un traducteur Devil, il est possible d'écrire soit un interpréteur, soit un traducteur. L'utilisation d'un interpréteur a été écartée pour différentes raisons :

- un interpréteur n'est pas plus facile à écrire qu'un compilateur;
- la définition d'un interpréteur au-dessus d'un langage pré-existant (Lisp, C++, ...) n'est pas d'un grand intérêt. Son développement permet, il est vrai, de voir quelques problèmes qui n'avaient pas été vus auparavant et ceci, dans un laps de temps assez réduit. Cependant, la réalisation d'un interpréteur complet du langage est généralement malaisée et longue (le langage sous-jacent se prête mal à certaines constructions du langage d'où des difficultés pour refléter la sémantique exacte du langage) et est de toute façon inefficace (on ne pourra déduire de l'interpréteur aucune donnée utile concernant l'efficacité du code qu'aurait produit un traducteur générant un exécutable).

Une fois un noyau du traducteur mis en place, l'implantation de l'ensemble du traducteur devient assez simple, si l'on a bien su choisir et implanter adéquatement les bonnes primitives. Le jeu de primitives est alors une boîte à outils logiciels. Par ailleurs, on dispose d'un véritable compilateur et des tests d'efficacité peuvent être réalisés. De même, tous les problèmes d'efficacité de l'implantation peuvent être pris en compte lors du développement d'un traducteur. Les techniques de reciblage sont utilisables dès lors qu'un noyau du langage final est disponible.

2.1. Le système de développement de programmes

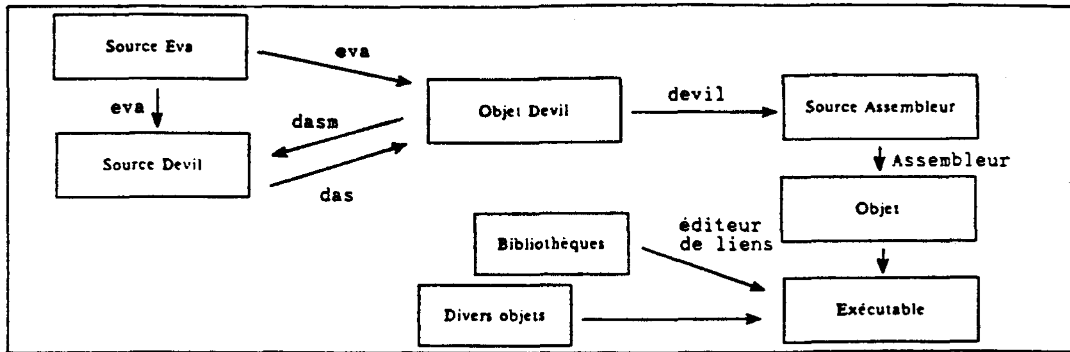


Figure III.4. –
Système de développement de programmes

Les différents composants logiciels du système de développement en langage Eva que nous avons développés sont :

`eva` : compilateur du langage Eva. Il génère un fichier Devil sous forme lisible ou codée (objet Devil);

`dasm` : désassembleur d'objets Devil (Devil sous forme codée). Il génère un fichier de texte en Devil sous forme lisible;

`das` : assembleur de sources Devil. A partir d'un texte Devil, il en génère la représentation codée. La sortie de la commande `dasm` est directement acceptée par la commande `das`;

`devil` : traducteur Devil. Partant d'un fichier Devil codé, il génère un fichier assembleur pour une machine cible donnée. Parmi les commandes que nous avons développées, c'est la seule qui dépende de la cible et ne soit donc pas portable.

Un assembleur local à la cible prend le fichier généré par la commande `devil` et produit un fichier objet. Si nécessaire, cet objet est alors relié à d'autres objets ou des bibliothèques par un éditeur de liens pour produire un fichier exécutable.

2.2. Projection sur stations de travail

Nous avons développé un prototype de traducteur Devil générant du code pour le processeur scalaire Motorola 68020 associé à un coprocesseur de calcul 68881, sur une station de travail Sun, sous le système d'exploitation Unix. C'est à une présentation de cette implantation qu'est dédiée cette section.

Nous avons successivement utilisé deux approches. La première consistait à obtenir rapidement un prototype de traducteur, incomplet, mais autorisant le test des structures fondamentales de Devil (manipulation des vecteurs, gestion des temporaires) et aussi du compilateur EVA tout entier (validité du code Devil généré). La deuxième implantation, qui est en cours, utilise une démarche générique, visant la création d'un squelette de traducteur, utilisable pour l'implantation sur différentes classes d'architectures (au moins les architectures scalaires et les architectures pipelines vectorielles). Pour cela, nous voyons le 68020 comme un processeur vectoriel traitant des vecteurs de longueur un (unités fonctionnelles traitants et registres contenants des vecteurs de longueur unité) ce qui uniformise notre vision de la traduction sur machines scalaires et vectorielles. Chacun de ces traducteurs génère son résultat sous forme d'un fichier assembleur.

Nous supposons ici la connaissance du processeur Motorola 68020 acquise par le lecteur. Nous rappelons simplement quelques faits qui suffiront à la lecture de la présente partie. Le 68020

dispose de 8 registres de données, de 8 registres d'adresses dont l'un (a7) est utilisé comme pointeur de sommet de pile et il dispose d'un registre d'état. Par ailleurs, il dispose de nombreux modes d'adressage simplifiant la compilation de langages évolués (adressages indirect et indirect indexé). Il traite des données situées dans ses registres ou en mémoire. Pour sa part, le co-processeur flottant MC 68881 ne traitent que des données situées dans ces propres registres. Ceux-ci sont chargés à partir des registres du 68020. Toutes les instructions classiques de calcul sur les réels y sont intégrées. Pour plus de renseignements sur ce processeur et son co-processeur, on pourra se référer à [DUBOIS & al. 87].

2.2.1. Première implantation

Comme nous l'avons dit, cette première implantation visait un développement rapide d'un prototype de compilateur EVA complet. Avant tout, ce sont les aspects essentiels de Devil qui se devaient d'être implantés (manipulation des vecteurs, gestion des temporaires, ...). Cette maquette nous a également permis d'écrire et, surtout, d'exécuter nos premiers programmes en EVA. Peu d'attention était portée à la qualité du code généré ou à l'implantation d'optimisations.

L'implantation de Devil sur un processeur scalaire ne pose pas de gros problèmes. Il faut *dé-vectoriser* les instructions de Devil. Notons cependant que la génération de code scalaire efficace nécessite la prise en compte des dépendances entre données pour l'allocation optimale des registres. Les techniques générales d'utilisation de treillis de dépendances (cf. § 2.3.) peuvent être utilisées pour optimiser le code généré au niveau de l'allocation des registres.

La traduction est alors très simple. Les instructions sont traduites les unes après les autres, sans tenir compte (ou presque) des instructions environnantes. A une forme d'instructions (type de l'instruction et types des opérandes) est associée une traduction standard. Cette traduction est "paramétrée" par le numéro des registres ou les adresses où sont stockés les opérandes.

L'implantation des temporaires

Les temporaires scalaires sont placés dans les registres de données du 68020. Lorsque tous les registres sont occupés, ceux-ci sont placés dans la pile. La place nécessaire au stockage de ces temporaires dans la pile est connue à la compilation. Bien entendu, dès qu'un registre se libère, un autre temporaire y est stocké à sa place.

Pour leur part, les temporaires vectoriels dont la taille de la zone est connue à la compilation sont placés dans la pile, les autres dans le tas. Les références à ces zones sont placées dans les registres d'adresses du 68020 tant qu'il y en a qui sont disponibles, sur la pile ensuite.

Implantation des segments en mémoire

Un processus Unix se compose de quatre segments, segments de code, de données, de pile et de tas. Les segments de MAD sont projetés sur ces quatre segments de la manière suivante :

- le segment de code (*code*) est implanté dans le segment de code du processus;
- le segment de données (*data*) est adressé aléatoirement. Il est implanté dans la zone de données du processus. Les segment de données des différents modules d'un programme sont reliés à l'édition de liens pour n'en former qu'un seul dans l'exécutable;
- les segments *param_rec*, *param_to_send*, *local*, *context* sont implantés sur la pile du processus (cf. fig. III.5.);
- le segment de tas (*heap*) est implanté dans le tas du processus;
- le segment *seg_tmp* n'existe pas en tant que tel, ceci pour des raisons d'efficacité du code. Les temporaires scalaires se trouvent soit dans les registres de données du 68020, soit sur la

pile. Les références aux temporaires vectoriels se trouvent soit dans les registres d'adresse du 68020, soit dans la pile. Les zones des vecteurs (temporaires ou non) se trouvent dans la pile, la zone de données ou sur le tas du processus;

- les segments de commons sont implantés dans la zone de données du processus.

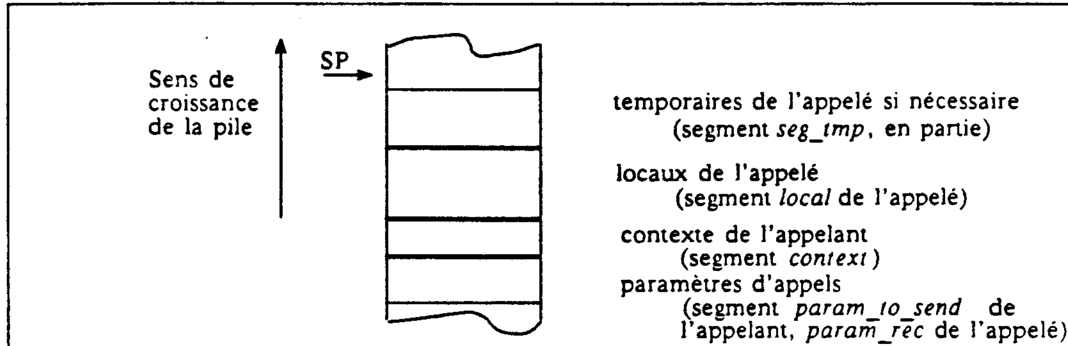


Figure III.5. -
Les segments de MAD projetés dans la mémoire d'un processus Unix

Le registre VLG

Le registre VLG, spécifiant la longueur des vecteurs à traiter, est implanté par une variable entière à laquelle une pile est associée. Cette pile est allouée dans le tas du processus. Elle peut grossir dynamiquement si sa profondeur initiale n'est pas suffisante. Le registre VSP représente le nombre de valeurs de VLG qui y ont été empilées.

Remarques diverses

Le MC 68020 utilise les codes conditions d'un registre d'état (SR) pour décider s'il doit se brancher ou non lors de l'exécution d'une instruction de saut conditionnel. MAD ne possède pas de tel registre et teste la valeur d'un objet scalaire pour savoir si elle doit réaliser le branchement ou non. Cette différence a rendu le code généré légèrement plus complexe, puisqu'il faut d'abord que le 68020 teste la valeur du scalaire pour positionner les bits du SR puis exécute une instruction de saut conditionnel.

2.2.2. Un traducteur générique : le squelette

Nous sommes en cours de développement d'une deuxième version du compilateur Devil. La caractéristique essentielle de celle-ci est que nous considérons le MC 68020 comme un processeur vectoriel traitant des vecteurs possédant un élément. De cette manière, nous construisons un squelette de compilateur pour processeurs vectoriels, le 68020 n'en étant alors qu'un cas particuliers. Nous concevons ce compilateur en deux couches, une couche indépendante de la cible faisant appel à des routines de génération de code spécifiques à la cible qui constituent la seconde couche. Ainsi, seule cette seconde couche sera à ré-écrire pour porter le compilateur sur un processeur vectoriel.

Concernant la projection des segments de MAD en mémoire, elle demeure inchangée par rapport à la première version, hormis le fait que le segment *local* des fonctions "terminales" (dont le corps ne contient pas d'appel de fonction) est alloué dans la zone de données et non sur la pile. Tous les segments *local* des fonctions terminales sont placés dans une même zone (dont la taille est le maximum des tailles de ces segments *local*).

Chapitre IV
Projection de MAD sur une machine–tableau

En accord avec le concept de mémoire d'objets [ASTHANA & al. 84], la machine abstraite MAD possède une mémoire vectorielle. Aussi, nous avons étudié le moyen de réaliser une telle mémoire vectorielle, associée à un processeur adressant ses opérandes vectoriels en tant qu'entités (contrairement aux processeurs habituels qui adressent les vecteurs comme des séquences de scalaires). Associé à cette mémoire, nous définissons un processeur à voir comme une réalisation matérielle de la machine virtuelle MAD.

MAD accède un vecteur via son en-tête. Nous aurions pu concevoir une mémoire adressant effectivement les vecteurs via un en–tête de ce genre. Cependant, nous avons simplifié un peu cet en–tête pour l'adressage des vecteurs. Nos premières idées en la matière ont été publiées dans [DEKEYSER & al 89]. Elles ont été poussées plus avant dans [DEKEYSER & al. 90a]. Elles sont ici développées et précisées.

Dans cette section, nous décrivons l'implantation de cette mémoire de vecteurs. La présentation est ici faite dans l'optique d'une machine tableau possédant un faible nombre de processeurs élémentaires. Une raison de ce choix est que suite à l'apparition de processeurs très puissants (processeur Intel 860 par exemple), l'utilisation de 4 processeurs en parallèle fournit déjà une puissance maximale aux alentours de 250 Mflops.

Nous présentons tout d'abord les fonctionnalités de la mémoire vectorielle telle que nous l'avons définie. Une description architecturale suit où nous introduisons le *processeur d'adressage vectoriel* permettant la réalisation d'accès vectoriels. Le fonctionnement de ce processeur est exposé ensuite et son activité sur quelques accès mémoires courants est détaillée. Pour terminer, nous montrons une construction itérative autorisant l'accès en parallèle aux composantes de vecteurs de taille arbitrairement grande.

1. Fonctionnalités de la mémoire vectorielle

Dans la mémoire vectorielle, un vecteur est adressé via une adresse vectorielle. Celle–ci se compose d'un triplet d'informations :

< adresse de base de la za, adresse de base de la zd, lg >

za signifie zone d'allocation et *zd* zone de description. Les trois informations de ce triplet seront désormais abrégées par < @za, @zd, lg > et seront écrites en italique dans le texte. *@zd* peut être nulle. Dans ce cas, le vecteur accédé est non décrit; il s'agit d'un accès contigu. *lg* est alors la longueur de la zone d'allocation. Sinon, *lg* est la longueur de la zone de description. Celle-ci est un tableau d'index des éléments sélectionnés de la zone d'allocation du vecteur à accéder. Ce type d'adressage représente un cas particulier d'accès à un vecteur Devil (descripteur liste d'index).

L'utilisation d'une telle technique entraîne un certain nombre de conséquences :

- simplification pour le CPU en ce qui concerne l'adressage d'un vecteur : il n'a pas à générer une séquence d'adresses (une adresse par composante du vecteur) mais un triplet d'informations;
- deux modes d'adressage vectoriel réels :

  direct : accès à un vecteur contigu ou à composantes espacées;

  indirect : accès à un vecteur dispersé, l'accès étant contrôlé par un vecteur d'index.

Par ailleurs, nous avons étendu les possibilités de cette mémoire vectorielle en lui permettant :

- d'exécuter localement les opérations mémoire à mémoire de transferts vectoriels. Durant l'exécution de ce type d'opérations, le CPU n'intervient pas puisqu'aucun traitement n'est à réaliser;
- d'adresser des vecteurs dont les composantes sont réguliérement espacées. Il suffit pour cela de remplacer dans le triplet d'adressage l'information "adresse de base de la zd" par la valeur du pas séparant les composantes.

Les accès contrôlés par un vecteur de bits font partie des extensions envisagées du PAV. En plus des triplets d'informations, une instruction est envoyée par le CPU vers la mémoire vectorielle indiquant s'il s'agit d'une lecture ou d'une écriture, d'une description par vecteur d'index ou un accès par pas, d'un accès direct ou indirect.

2. Modèle architectural

La mémoire vectorielle s'insère dans un modèle architectural où un séquenceur (Seq) contrôle les activités du processeur vectoriel (PV) composé de processeurs élémentaires, d'un processeur scalaire (PS) et de la mémoire vectorielle (cf. fig. IV.1).

Le séquenceur prend en charge la synchronisation des activités des différentes unités. Son activité est contrôlée par le flot d'instructions du programme en cours d'exécution. Il prend en charge les instructions de contrôle de l'exécution des programmes (sauts). Il répartit les instructions qu'il ne peut exécuter (accès mémoire, traitements scalaires et vectoriels) entre les trois autres unités (respectivement la mémoire vectorielle et les processeurs scalaire et vectoriel).

Le processeur scalaire prend en charge les traitements scalaires ainsi que les opérations terminales de réduction de vecteurs. Par exemple, pour le calcul de la norme d'un vecteur, le carré des composantes et leur somme est calculée par le processeur vectoriel. L'extraction de la racine carrée est réalisée par le processeur scalaire.

Le processeur vectoriel prend en charge les traitements vectoriels. Chaque processeur élémentaire traite une composante du vecteur et fournit une composante du résultat. Chaque

processeur élémentaire inclut des registres de travail. Par assemblage, ces registres, scalaires au niveau des PEs, forment des registres vectoriels au niveau du processeur vectoriel. Supposons que le processeur vectoriel se compose de N_{PE} processeurs élémentaires, le traitement d'un vecteur de N composantes correspond à l'un des trois cas :

$N = N_{PE}$: chaque PE prend en charge le calcul d'une composante du vecteur résultat;

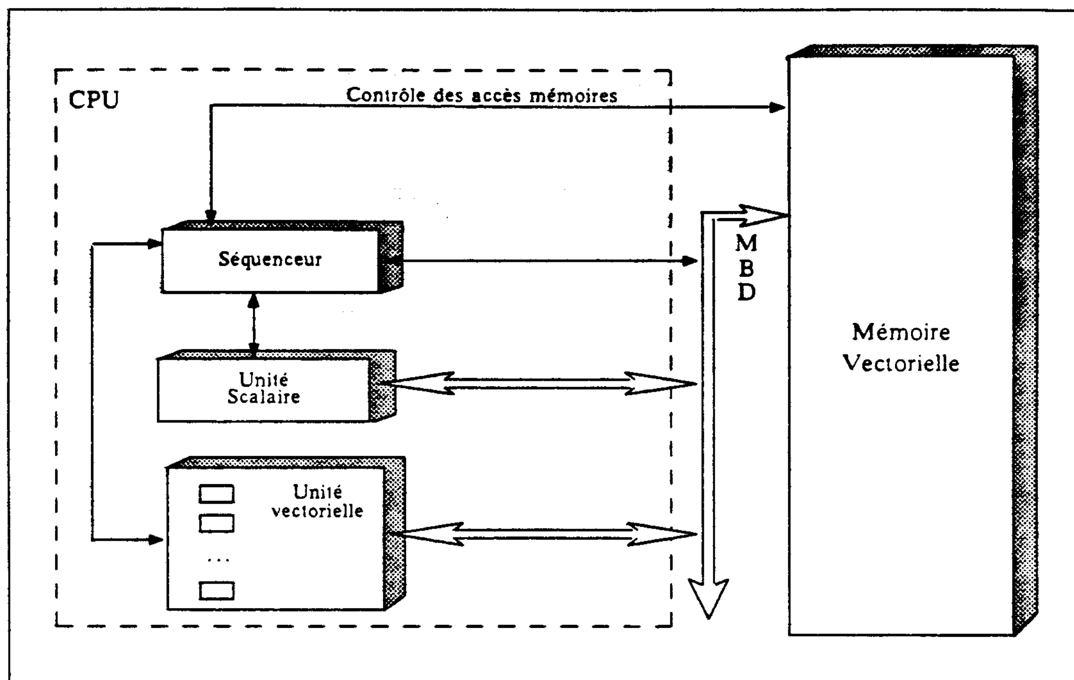


Figure IV.1 -
Modèle architectural

$N < N_{PE}$: N PEs participent au calcul du vecteur résultat. Les $N_{PE} - N$ autres PEs sont au repos;

$N > N_{PE}$: N peut alors s'écrire : $N = q \times N_{PE} + r$ ($0 <= q$, $0 <= r < N_{PE}$). Pendant q "cycles", les N_{PE} PEs effectuent leurs calculs : le PE_i calcule les composantes d'indices i, $i + N_{PE}$, $i + 2 \times N_{PE}$, ..., $i + (q - 1) \times N_{PE}$. Lors d'un dernier cycle, r PEs calculent les r dernières composantes du résultat. Les $N_{PE} - r$ autres PEs sont au repos.

Pour sa part, la mémoire vectorielle se compose de deux unités (cf. fig. IV.2) :

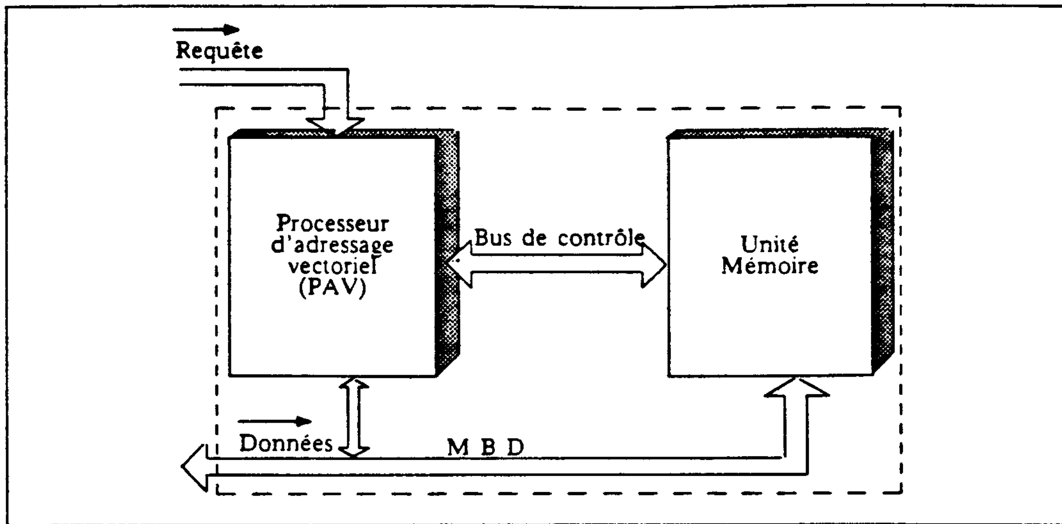


Figure IV.2 -
Structure de la mémoire vectorielle

- une unité de mémorisation;
- un processeur spécialisé dans l'adressage vectoriel qui reçoit les requêtes d'accès du séquenceur et déclenche les activités de la mémoire afin d'accéder aux vecteurs. Ce processeur porte le nom de "processeur d'adressage vectoriel" (PAV).

La mémoire (cf. fig. IV.3) est composée de bancs entrelacés et elle accepte plusieurs accès simultanés (autant que de PEs). Elle possède la logique nécessaire à la transformation d'adresses logiques en adresses physiques. Elle résout les conflits d'accès. Comme dans le DSPA [JEGOU 86], à chaque banc est associée une file d'attente où sont stockées les requêtes conflictuelles. Lorsqu'une requête atteint un banc en cours d'activité, cette requête est placée dans la file d'attente attachée au banc. Lorsque le banc a terminé l'exécution d'une requête, il examine sa file d'attente. Si une requête s'y trouve, elle est traitée. L'accès à toutes les composantes d'un vecteur est réalisé en même temps. Ainsi, lors d'une lecture, tous les PEs pourront lire leur bus de données en même temps lorsque le PAV aura émis un signal indiquant la fin d'activité de la mémoire. Les données y auront été tamporisées au fur et à mesure des accès mémoires effectifs.

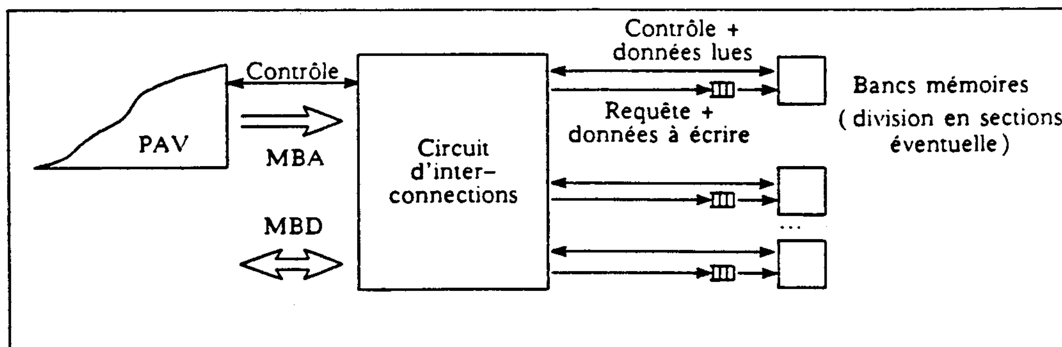


Figure IV.3 -
L'unité mémoire de la mémoire vectorielle

Un multi-bus d'adresses (MBA) relie le PAV à la mémoire. C'est sur ce bus que circulent les adresses effectives des composantes des vecteurs à accéder.

Sur le multi-bus de données (MBD) circulent les valeurs des composantes des vecteurs accédés. Ce multi-bus possède la logique nécessaire pour conserver les dernières informations qui y ont été écrites. Pour cela, un tampon y est relié. L'écriture d'une nouvelle valeur y est validée par l'unité quand elle écrit des données sur le multi-bus.

Dans ce qui suit, on notera TMB le nombre de bus composant un multi-bus.

Le processeur scalaire intervient dans les opérations de réduction. Il accède les composantes des vecteurs via un tampon (BUF) situé entre le multi-bus de données et son propre bus de données. Les accès mémoires sont contrôlés par le PAV en ce qui concerne les accès vectoriels, par le séquenceur en ce qui concerne les accès aux données scalaires. Les conflits d'accès à la mémoire entre accès vectoriels et accès scalaires sont résolus directement par le séquenceur. Celui-ci contrôlant le PAV, il ne déclenche d'activité mémoire vectorielle que si aucun accès scalaire est en cours et, réciproquement, ne déclenche pas d'accès scalaire tant que le dernier accès vectoriel qu'il a déclenché n'est pas terminé.

Dans ce modèle architectural, les différentes unités fonctionnent de manière totalement asynchrone les unes par rapport aux autres. Elles communiquent par une technique classique de *hand-shaking*. Dans l'unité vectorielle, les processeurs élémentaires ne fonctionnent pas en mode SIMD, mais plutôt en SPMD (Single Program, Multiple Data). Chacun exécute une fonction sur les données qu'il a chargées depuis la mémoire. Lorsqu'ils ont fini leur calcul et qu'ils doivent ranger les résultats en mémoire, ils se re-synchronisent. En résumé, les PEs fonctionnent en mode synchrone lors des accès mémoire, en mode asynchrone le reste du temps.

3. Fonctionnement du PAV

Dans cette section, nous décrivons le déroulement de divers types de requêtes mémoire :

- accès en lecture à un vecteur contigu;
- accès en lecture à un vecteur dont les composantes sont régulièrement espacées;
- accès en lecture à un vecteur dispersé;
- accès en écriture.

A la fin de la section, nous décrirons le déroulement d'une opération de transfert vectoriel.

3.1. Schéma d'un accès en lecture aux éléments d'un vecteur

Quel que soit le type d'accès en lecture réalisé, il suit un schéma constant :

1. le séquenceur émet une requête vers le PAV. Pour cela, il place le triplet d'informations sur les bus le reliant au PAV, indique le type d'accès à réaliser et déclenche l'activité du PAV par validation du signal START;
2. le PAV calcule les adresses effectives des composantes du vecteur à accéder et les mémorise dans un tampon interne. Une fois ce calcul réalisé, il en informe le séquenceur par validation du signal EOA;
3. le séquenceur déclenche l'accès effectif en mémoire en validant le signal WAKE-UP;

4. le PAV déclenche l'activité de la mémoire. Pour cela, il place les adresses effectives des composantes préalablement mémorisées dans le tampon interne sur les bus du multi-bus d'adresses. Il valide les signaux de sélection des bus des multi-bus, en accord avec le nombre de composantes du vecteur à accéder. Il indique à la mémoire que l'opération à réaliser est une lecture. Il déclenche l'activité de la mémoire par validation du signal MREQ;

5. la mémoire effectue l'accès, place les données sur le multi-bus de données et avertit le PAV que l'accès a été réalisé par validation du signal EOMA;

6. le PAV informe le séquenceur que les données sont disponibles sur le multi-bus de données en validant à nouveau le signal EOA;

7. le séquenceur peut alors indiquer au processeur scalaire ou processeur vectoriel, selon l'opération à effectuer, que les données sont disponibles sur le multi-bus de données.

Nous allons maintenant détailler le calcul des adresses effectives des composantes (point 3), seule différence entre les différents types d'accès en lecture.

Calcul des adresses effectives des composantes dans le cas d'un accès à un vecteur contigu

Pour un vecteur contigu, le calcul des adresses effectives est simple. Le PAV part de l'adresse de base *@za* et génère les adresses *@za*, *@za* + 1, *@za* + 2, ... *@za* + *lg* − 1.

Calcul des adresses effectives des composantes dans le cas d'un accès à un vecteur dont les composantes sont réguliérement espacées

Pour un vecteur dont les composantes sont réguliérement espacées, le PAV prend *@za* et *@zd* comme données. Dans ce cas, *@zd* est la valeur du pas (notée "Pas" dans ce paragraphe) séparant les composantes. Le PAV génère alors les adresses *@za*, *@za* + Pas, *@za* + 2 x Pas, ... *@za* + (*lg* − 1) x Pas.

Calcul des adresses effectives des composantes dans le cas d'un accès à un vecteur dispersé

Pour un vecteur dispersé dont l'accès est contrôlé par un vecteur d'index, le PAV réalise son calcul d'adresses en trois phases :

a) calcul des adresses effectives des composantes du vecteur d'index;

b) accès aux valeurs des index se trouvant aux adresses précédemment calculées;

c) calcul des adresses effectives des composantes du vecteur dispersé.

La phase a) revient à calculer les adresses effectives des composantes d'un vecteur contigu, le vecteur d'index étant un vecteur contigu. La seule différence réside dans le fait que l'adresse de base n'est pas *@za* mais *@zd*.

La phase b) est un accès mémoire à un vecteur contigu. Les adresses des composantes sont placées sur le multi-bus d'adresses, les signaux de sélection positionnés, indique à la mémoire qu'une opération de lecture doit être effectuée et déclenche finalement l'activité de la mémoire (validation du signal MREQ). La mémoire réalise l'accès et place les valeurs des index sur le multi-bus de données et valide le signal EOMA pour en avertir le PAV. Celui-ci accède alors la valeur des index sur le multi-bus.

La phase c) est particulière à ce mode d'accès. Notons i_0, i_1, ... i_{lg-1} les éléments du vecteur d'index préalablement accédés. Le PAV génère alors les adresses *@za* + i_0, *@za* + i_1, ... *@za* + i_{lg-1} qui sont les adresses effectives des composantes du vecteur dispersé. Ces adresses sont générées par

addition de l'adresse de base du vecteur à accéder aux éléments du vecteur d'index qui ont été accédés à la phase b).

3.2. Schéma d'un accès en écriture aux éléments d'un vecteur

Pour décrire un accès vectoriel en écriture, nous modifions le schéma donné plus haut, lequel devient :

1. le séquenceur émet une requête vers le PAV. Pour cela, il place le triplet d'informations sur les bus le reliant au PAV, indique le type d'accès à réaliser et déclenche l'activité du PAV par validation du signal START;

2. le PAV calcule les adresses effectives des composantes du vecteur à accéder et les mémorise dans un registre interne. Ce calcul est effectué de la même manière que pour un accès en lecture (cf. plus haut). Une fois ce calcul réalisé, il en informe le séquenceur par validation du signal EOA;

3. le séquenceur déclenche le positionnement par le processeur vectoriel, sur le multi-bus de données, des composantes du vecteur à écrire;

4. le séquenceur déclenche l'accès effectif en mémoire en validant le signal WAKE-UP;

5. le PAV déclenche l'activité de la mémoire. Pour cela, il place les adresses effectives des composantes préalablement mémorisées dans le tampon interne sur les bus du multi-bus d'adresses. Il valide les signaux de sélection des bus des multi-bus. Il indique à la mémoire que l'opération à réaliser est une écriture. Il déclenche l'activité de la mémoire par validation du signal MREQ;

6. la mémoire effectue l'écriture et avertit le PAV que l'accès a été réalisé par validation du signal EOMA;

7. le PAV informe le séquenceur que les données ont été écrites en mémoire en validant à nouveau le signal EOA.

3.3. Les transferts vectoriels

Le PAV prend lui-même en charge les transferts vectoriels mémoire à mémoire $\vec{W} := \vec{V}$. Bien entendu, les vecteurs source et cible peuvent chacun être contigus, à composantes régulièrement espacées ou dispersés. Un transfert vectoriel mémoire à mémoire suit le schéma suivant :

a) le séquenceur déclenche un accès en lecture au vecteur $\vec{V}$ < @za, @zd, lg > en indiquant que le PAV doit mémoriser la valeur des composantes dans son registre BD (par validation du signal BIN);

b) le PAV avertit le séquenceur de la fin de la lecture;

c) le séquenceur déclenche un accès en écriture au vecteur $\vec{W}$ < @za', @zd', lg' > en indiquant que les données à écrire se trouve dans le registre BD du PAV (par validation du signal BOUT).

Notons que les étapes b) et c) ne sont pas nécessairement consécutives. Le séquenceur peut déclencher des accès en lecture ou en écriture entre les deux si cela est nécessaire. Seule une opération de transfert vectoriel ne peut être réalisée ici car elle viendrait écraser les valeurs des composantes du vecteur source mémorisées dans BD.

4. Description détaillée du PAV

Dans cette section, une description interne du processeur d'adressage vectoriel est présentée au niveau de ses unités fonctionnelles. Les communications internes et externes au PAV sont également détaillées. Des exemples d'accès vectoriels sont détaillés au niveau des communications entre les différentes unités de la machine.

4.1. Structure générale du processeur d'adressage vectoriel

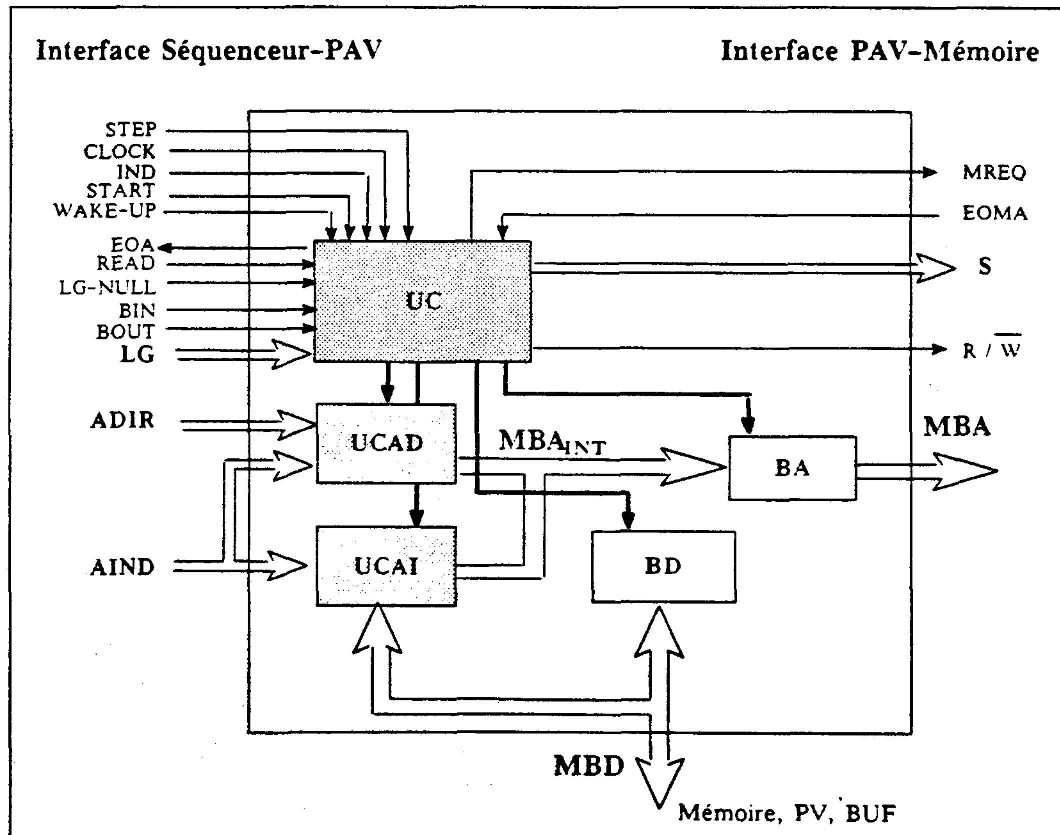


Figure IV.4 -
Structure interne du PAV

Les différentes unités fonctionnelles sont : (cf. figure IV.4) :

- l'unité de contrôle (UC) qui contrôle et séquence les activités du PAV. Elle reçoit les requêtes du séquenceur, émet des requêtes vers la mémoire et contrôle les autres unités du PAV;

- l'unité de calcul d'adresses directes (UCAD) qui réalise les calculs nécessaires pour la génération d'adresses des composantes des vecteurs contigus (adressage direct d'un vecteur ou accès aux composantes d'un vecteur d'index pour un adressage indirect) ou des vecteurs à composantes réguliérement espacées;
- l'unité de calcul d'adresses indirectes (UCAI) qui réalise les calculs nécessaires pour la génération d'adresses des composantes des vecteurs dispersés;
- une unité de mémorisation de données (BD) utilisée pour mémoriser le contenu du multi-bus de données lors des opérations de transferts vectoriels;
- une unité de mémorisation d'adresses (BA) utilisée pour mémoriser les adresses générées par l'UCAD ou l'UCAI avant que l'accès effectif à la mémoire ne soit déclenché.

Chaque unité du PAV réalise un traitement dont le temps d'exécution est fixe. Aussi, dans le PAV, les unités fonctionnent de manière synchrone.

Les signaux internes au PAV par lesquelles l'unité de contrôle commande les autres unités sont les suivants :

step	entre l'UC et l'UCAD, signal indiquant le type d'accès qui est soit un accès contigu, soit un accès non adjacent;
go-ucad	entre l'UC et l'UCAD, déclenchement de la lecture de la donnée en entrée sur ADIR (et de la donnée sur AIND s'il s'agit d'un accès non adjacent) suivie du calcul des adresses directes (éventuellement non adjacentes) et de l'envoi du résultat sur MBAINT.
load-aind	entre l'UC et l'UCAI, charger la donnée en entrée sur AIND;
go-ucai	entre l'UC et l'UCAI, charger le contenu du tampon de données, effectuer le calcul des adresses indirectes puis envoyer les adresses résultantes sur MBAINT.
load-ba	entre l'UC et le BA, charger le contenu du MBAINT dans le tampon d'adresses;
write-ba	entre l'UC et le BA, décharger le contenu du tampon d'adresses sur le MBA.
load-bd	entre l'UC et le BD, charger le contenu du MBD dans le tampon de données;
write-bd	entre l'UC et le BD, décharger le contenu du tampon de données sur le MBD.

Les signaux du PAV

Signaux de contrôle du PAV

CLOCK	Signal d'horloge cadençant les activités du PAV;

Signaux de contrôle émis par le séquenceur vers le PAV

STEP	Ce signal est échantillonné par le PAV lorsque l'entrée START est validée. Si IND = 0, alors ce signal est à 0 s'il s'agit d'un accès à un vecteur contigu, 1 s'il s'agit d'un vecteur à composantes espacées. Si IND = 1, ce signal n'est pas significatif;
IND	Ce signal est échantillonné par le PAV lorsque l'entrée START est validée. S'il vaut 0, il s'agit d'un accès à un vecteur contigu ou à composantes espacées

(accès direct). S'il vaut 1, il s'agit d'un accès à un vecteur dispersé (accès indirect);

START Déclenchement des activités du PAV pour le calcul des adresses effectives des composantes du vecteur à accéder;

WAKE-UP Déclenchement des activités du PAV pour l'accès effectif aux composantes du vecteur dont les adresses ont été calculées et se trouvent dans le tampon BA;

READ Ce signal est échantillonné par le PAV lorsque l'entrée START est validée. S'il vaut 0, il s'agit d'un accès en écriture (écriture des composantes d'un vecteur contigu, d'un vecteur espacé ou d'un vecteur dispersé - dans ce cas, il s'agit d'une opération d'éclatement). S'il vaut 1, il s'agit d'un accès en lecture (lecture des composantes d'un vecteur contigu, d'un vecteur espacé ou d'un vecteur dispersé - dans ce cas, il s'agit d'une opération de rassemblement).

LG-NULL Ce signal est échantillonné par le PAV lorsque l'entrée START est validée. C'est un signal de sélection du circuit qui valide la longueur du vecteur à accéder se trouvant sur le bus LG. Si ce signal vaut 0, le contenu de LG est invalide et le PAV n'est pas sélectionné. Si ce signal vaut 1, le contenu de LG est valide et le PAV est sélectionné;

BIN Ce signal est échantillonné par le PAV lorsque l'entrée WAKE-UP est validée. S'il vaut 1, le PAV mémorisera le contenu du MBD dans son tampon BD à la réception du signal EOMA de la mémoire;

BOUT Ce signal est échantillonné par le PAV lorsque l'entrée START est validée. S'il vaut 1, le PAV placera le contenu de son tampon BD sur le MBD avant la validation du signal MREQ;

Signaux de contrôle émis par le PAV vers le séquenceur

EOA Signal de fin d'activités du PAV. A la suite d'un START, il indique au séquenceur que les adresses effectives des composantes du vecteur ont été calculées. A la suite d'un WAKE-UP, il indique au séquenceur que les composantes ont été accédées en mémoire;

Signaux de contrôle émis par le PAV vers la mémoire

MREQ Sa validation déclenche une activité mémoire;

R / W Ce signal est échantillonné par la mémoire lorsque l'entrée MREQ est validée. S'il vaut 0, il s'agit d'un accès en écriture aux composantes d'adresses présentes sur le MBA. Le contenu du bus$_i$ du MBD est écrit à l'adresse véhiculée sur le bus$_i$ du MBA. S'il vaut 1, il s'agit d'un accès en lecture aux composantes d'adresses présentes sur le MBA. Le contenu de l'adresse présente sur le bus$_i$ du MBA est placé sur le bus$_i$ du MBD.

S Ces signaux sont échantillonnés par la mémoire lorsque l'entrée MREQ est validée. Ce sont les signaux de sélection des bus des multi-bus de données et d'adresses. Lorsque le signal S_i est valide, il indique que les bus 0 à i-1 des multi-bus d'adresses et de données véhiculent des données valides.

Signaux de contrôle émis par la mémoire vers le PAV

EOMA Ce signal est reçu après l'émission d'un signal MREQ du PAV vers la mémoire. Sa réception indique au PAV que l'accès mémoire demandé a été exécuté. Les

données ont été écrites si R/W valait 0, ont été lues et placées sur le MBD si R/W valait 1.

Les bus

ADIR — Ce bus véhicule l'adresse de base du vecteur accédé de manière contiguë ou espacée (@*za* pour un accès contigu ou espacé, @*zd* pour un accès dispersé);

AIND — Ce bus véhicule l'adresse de base (@*za*) du vecteur accédé de manière dispersée ou le pas (@*zd*) pour un vecteur accédé selon un pas. Dans le cas d'un accès à un vecteur contigu, ce bus ne contient pas d'information valide;

LG — Ce bus véhicule la valeur du champ *lg* du triplet d'accès vectoriel;

MBA — Multi-bus d'adresses;

MBD — Multi-bus de données.

4.2. Description des différentes unités du PAV

L'unité de calcul des adresses directes

L'unité de calcul des adresses directes (cf. figure IV.5) se compose d'additionneurs fonctionnant en parallèle, chacun calculant l'adresse effective d'une composante. Chacun additionne la valeur de l'adresse de base du vecteur (reçue sur le bus ADIR) aux valeurs 1, 2 ... N s'il s'agit d'un accès contigu, aux valeurs pas, 2 x pas, ... N x pas s'il s'agit d'un accès avec pas. La valeur du pas est reçue sur le bus AIND. La sortie de chacun des additionneurs est reliée à un bus du multi-bus d'adresses interne MBAINT. La réception du signal go-ucad de la part de l'unité de contrôle du PAV déclenche le calcul des adresses. Un multiplexeur piloté par le signal step généré lui aussi par le séquenceur du PAV sélectionne la valeur à additionner, selon le type d'adressage.

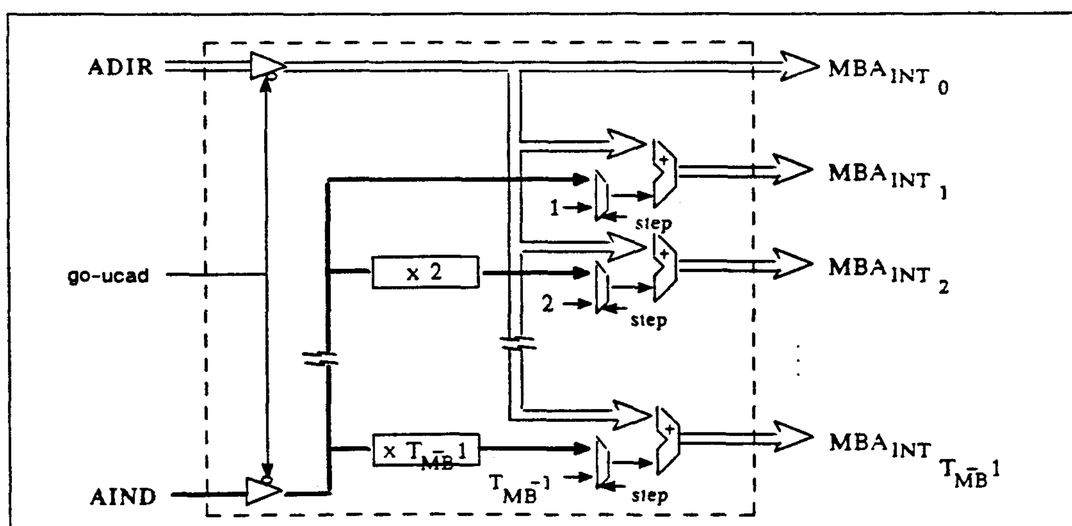


Figure IV.5 -
L'unité de calcul des adresses directes

Contrairement à ce que pourrait laisser croire la figure IV.5, nous n'utilisons pas des circuits multiplieurs pour calculer les adresses des composantes. Nous utilisons une logique plus simple à base de circuits de décalage et d'additionneurs, TMB ayant une valeur assez faible en réalité.

L'unité de calcul des adresses indirectes

L'unité de calcul des adresses indirectes (UCAI) se compose essentiellement d'additionneurs (cf. fig. IV.6). Le bus AIND véhicule la valeur du champ @*za* dans le cas d'un adressage vectoriel indirect.

La valeur en entrée sur le bus AIND est mémorisée dans un tampon lorsque le signal load-aind est validé. Les bus du multi-bus de données contiennent les valeurs des index qui viennent d'être accédées en mémoire. Les adresses effectives des composantes d'un vecteur dispersé sont calculées en parallèle par les additionneurs de l'UCAI à la réception du signal go-ucai. Elles sont ensuite placées sur le multi-bus d'adresses interne où elles sont mémorisées dans le tampon d'adresses.

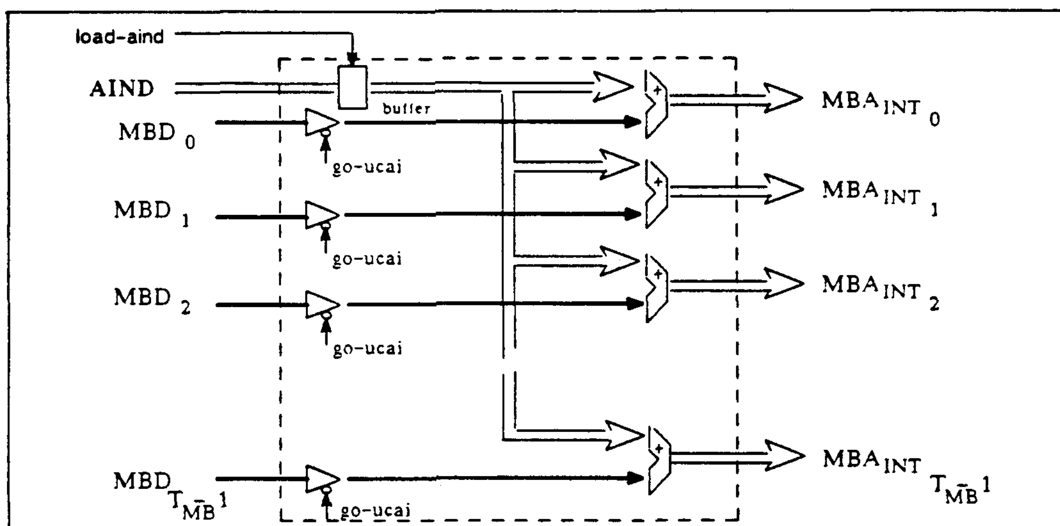


Figure IV.6 –
L'unité de calcul des adresses indirectes

L'unité de contrôle

Nous ne décrirons pas ici dans le détail la réalisation de l'unité de contrôle du PAV; ce serait fastidieux et sans intérêt. Nous dirons simplement que son activité interne et que ses interactions avec les autres unités du PAV sont réalisées de manière totalement synchrone. Par contre, les signaux reçus de l'extérieur doivent être traités de manière asynchrone.

Les unités de mémorisation de données et d'adresses

Les unités de mémorisation de données et d'adresses sont de simples registres. Il n'est pas nécessaire d'utiliser des registres pouvant être lus et écrits simultanément.

4.3. Activités des unités lors d'accès vectoriels

Nous présentons ici quelques chronogrammes indiquant les signaux utilisés et leur rôle lors de diverses activités du PAV. Les numéros indiqués dans le bas des chronogrammes se réfèrent aux différentes phases des schémas d'accès mémoires donnés plus haut.

Accès vectoriel indirect en lecture

Nous décrivons ici le séquencement d'opérations liées à la réalisation d'un accès vectoriel indirect en lecture (opération classique de rassemblement – cf. fig. IV.7).

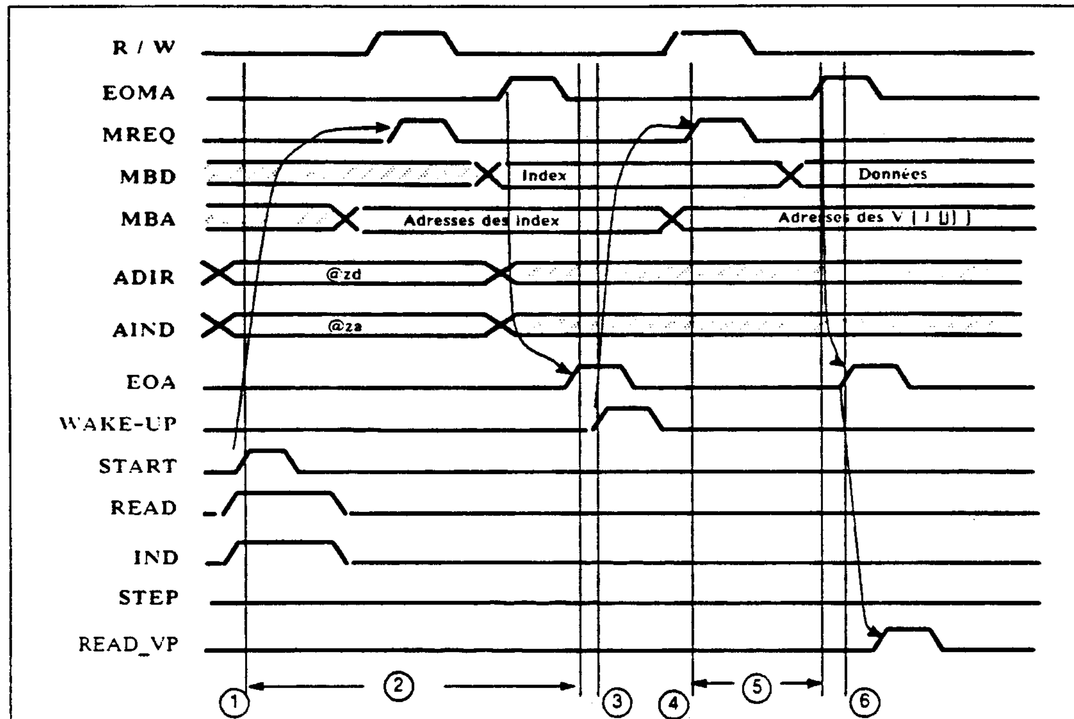


Figure IV.7 –

Chronogramme d'un accès en lecture au vecteur $\vec{V}$ via un index $\vec{I}$

(rassemblement $\vec{V}$ [$\vec{I}$])

Un accès indirect vectoriel se compose de 6 étapes. Tout d'abord, le séquenceur prépare l'accès en plaçant l'adresse vectorielle composée de l'adresse de base du vecteur à accéder (*@za*), l'adresse de base du vecteur d'index (*@zd*) et de sa longueur respectivement sur les bus AIND, ADIR et LG, et en validant les signaux de contrôle indiquant le type de l'accès (IND est à l'état haut pour un accès indirect, READ a l'état haut pour un accès en lecture). Une fois que l'accès est prêt, le séquenceur valide le signal START (étape 1). Le PAV échantillonne ses entrées et effectue le calcul des adresses effectives des composantes du vecteur à accéder (étape 2). Pour cela, les valeurs des index doivent être accédées. L'UCAD commence par calculer les adresses effectives des index à partir de l'adresse de base du vecteur d'index (*@zd*). L'accès en lecture est déclenché et la valeur des index est récupérée sur le multi-bus de données. Celles-ci sont alors additionnée par l'UCAI à l'adresse de base du vecteur à accéder (*@za*), ce qui donne les adresses effectives des éléments. Ces adresses sont placées dans l'unité de mémorisation des adresses du PAV, via le multi-bus d'adresses interne. Enfin, le PAV émet un signal de fin d'activité EOA (fin de l'étape 2). Le séquenceur déclenche la lecture effective par validation du signal WAKE-UP de réveil du PAV (étape 3). Le PAV place les adresses effectives sur le multi-bus d'adresses, indique à mémoire qu'il s'agit d'un accès en lecture (mise à l'état haut du signal R / W), valide les signaux S_i de sélection des bus valides sur les multi-bus (selon la longueur du vecteur à accéder) et déclenche l'accès mémoire en validant le signal MREQ (étape 4). La mémoire effectue l'accès aux composantes du vecteur, les place sur le multi-bus de données ou elles sont mémorisées et émet un signal de fin d'activité EOMA (étape 5). A la réception de ce signal, le PAV a terminé l'exécution de la requête et émet un signal de fin d'activité EOA vers le séquenceur (étape 6). Les données sont présentes

sur le multi-bus de données. Le séquenceur peut déclencher leur lecture par le processeur vectoriel (validation du signal READ-VP).

Transfert vectoriel entre deux vecteurs contigus

Nous décrivons ici le séquencement d'une opération de transfert vectoriel interne à la mémoire d'un vecteur source contigu, dans un vecteur cible contigu également (cf. fig. IV.8). Nous avons choisi une source et une cible contigus pour simplifier l'exposé étant entendu que n'importe quel type d'adressage est admissible pour la source ou la cible de l'opération de transfert.

Un transfert se compose de deux parties distinctes. D'une part, la lecture des valeurs des composantes du vecteur source (à gauche, les étapes sont préfixées par 'L'). A l'issue de cette lecture, leurs valeurs sont stockées dans le registre interne BD. D'autre part l'écriture des composantes de la cible (à droite, les étapes étant préfixées par 'E').

La lecture et l'écriture se compose chacune de 6 phases. A la lecture, pour permettre la mémorisation du contenu du multi-bus de données dans le PAV, le signal BIN est validé. Une fois les données accédées (fin de l'étape L5), elles sont placées dans le tampon BD. Pour l'écriture, le signal BOUT est validé et indique que les données présentes dans le tampon BD doivent être placées sur le multi-bus de données en fin d'étape E3 et vaide durant E4. C'est pendant la phase E4 que l'écriture effective des données en mémoire est réalisée.

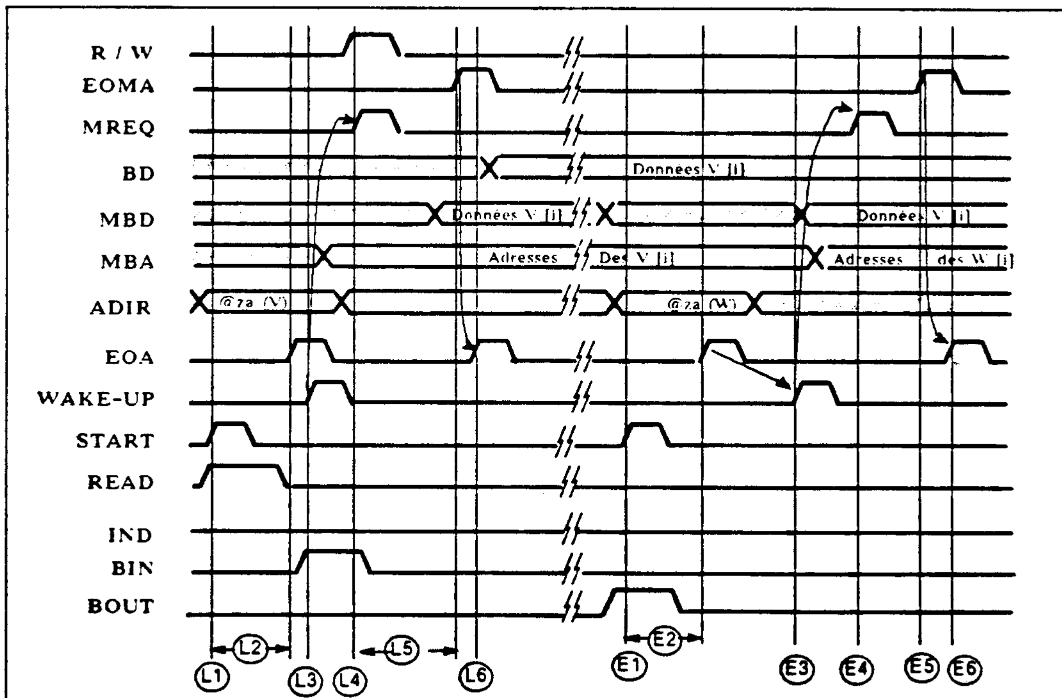


Figure IV.8 -

Chronogramme d'un transfert vectoriel mémoire-mémoire de vecteurs contigus $\vec{W} := \vec{V}$

5. Définition d'un PAV modulaire

Nous avons regardé comment pourrait être intégré un PAV dans un circuit intégré. Nous avons alors constaté que la longueur des vecteurs traités par un circuit est limité à 4 si on veut conserver

un nombre de broches raisonnable (environ 250 broches cf. fig. IV.9). Concédons qu'il s'agit là de petits vecteurs !

Aussi, nous avons conçu notre PAV de manière à ce qu'il soit modulaire, c'est à dire de manière à ce que N PAVs (avec N tel qu'il existe un entier positif n et N = 4n) puissent être assemblés pour traiter des vecteurs plus longs (jusqu'à 4 x N éléments). Pour en permettre l'assemblage, deux types de circuits annexes sont nécessaires. Par ailleurs, nous voulions que l'interface reste la même tant du côté du séquenceur que du côté de la mémoire. Côté mémoire, la seule différence réside dans le nombre de bus des multi-bus. Après avoir positionné les problèmes dus à l'assemblage des PAVs, nous décrirons notre solution de construction de PAV modulaire.

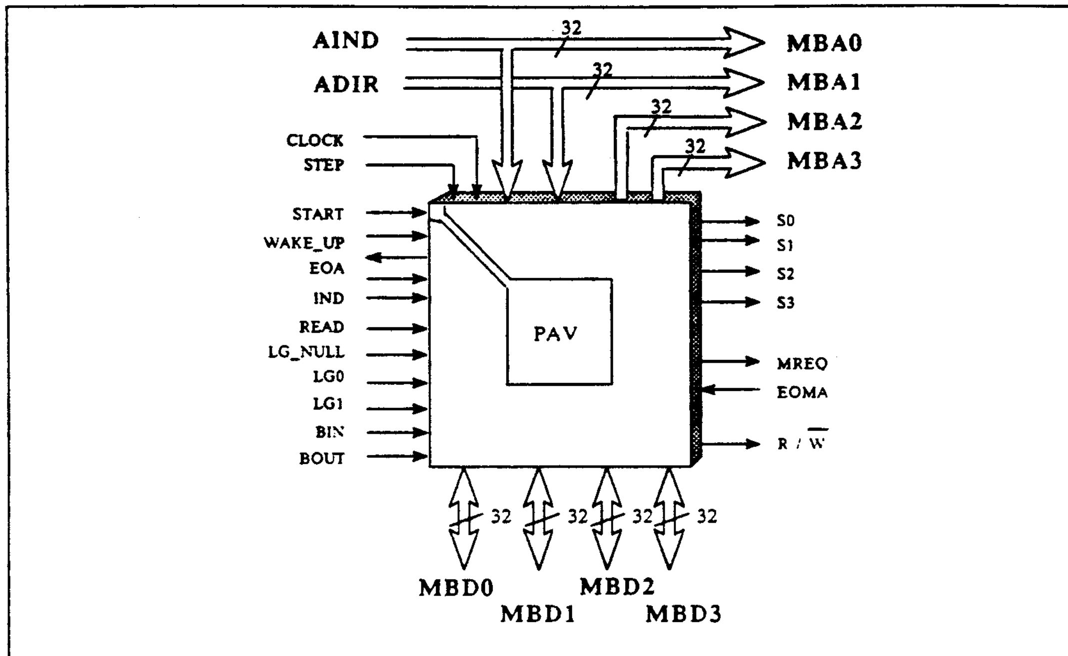


Figure IV.9 –
Les broches "importantes" d'un circuit PAV

Principe

Nous désirons accéder des vecteurs possédant un maximum de M (M = 4 x N) composantes. Pour cela, nous utiliserons N PAVs. En mettant en commun leur multi-bus d'adresses et de données, des multi-bus de largeur M bus seront obtenus. Chaque PAV prendra en charge l'accès à une tranche du vecteur : le PAV_i (i variant de 0 à M – 1) accédera à la tranche du vecteur dont les numéros de composantes sont 4 i, 4 i + 1, 4 i + 2 et 4 i + 3. Lors d'un accès à un vecteur de $\eta < M$ composantes (avec $\eta = 4q + r$), les q+1 PAVs PAV_0 ... PAV_q seront actifs, les autres étant inactifs. Les PAV_0 à PAV_{q-1} accéderont des vecteurs de 4 composantes. Le PAV_q accédera un vecteur de r composantes.

Pour réaliser un accès à η composantes, le séquenceur génère toujours le même triplet d'adressage vectoriel < @za, @zd, lg = η >, lg variant maintenant de 1 à M. A partir de ce triplet, chaque PAV doit recevoir un triplet lui permettant d'accéder à la tranche de vecteur adéquat. Ainsi pour un accès contigu, le PAV_i doit recevoir le triplet : < @za_i, NULL, lg_i > avec 0 <= i <= 3 et

```
@zai = @za + 4 i
lgi = 4 si i < q
```

$lg_i = r$ si $i = q$
$lg_i = 0$ si $i > q$ (PAV non sélectionné)

Pour un accès à un vecteur à composantes espacées, le PAV_i doit adresser le vecteur : < $@za_i$ = @za + 4 x i x Pas, Pas, lg_i >.

Pour un accès à un vecteur dispersé, le PAV_i doit adresser le vecteur < $@za_i$ = @za, $@zd_i$ = @zd + 4 i, lg_i >.

La génération des adresses vectorielles des sous-vecteurs est effectuée par une unité dite *distributeur d'adresses*.

On note que seule l'adresse de base du vecteur adressé de manière contiguë (ou avec un pas) doit être traitée. L'adresse de base du vecteur adressé indirectement peut être transmise directement aux PAVs. Pour ce type d'accès, les distributeurs d'adresses reçoivent donc un doublet d'informations.

Pour un accès avec pas, le pas doit être envoyé aux distributeurs d'adresses pour calculer les adresses vectorielles des sous-vecteurs. Pour ce type d'accès, les distributeurs d'adresses reçoivent donc un triplet d'informations.

Un distributeur d'adresses prend en entrée l'adresse vectorielle du séquenceur et génère les adresses vectorielles pour les PAVs qu'il contrôle. Afin de conserver le même aspect modulaire, un circuit distributeur d'adresses (DA) génère les adresses pour 4 PAVs. Par ailleurs, le DA est lui-même conçu de manière modulaire. De cette manière, PAVs et DAs peuvent être assemblés de manière modulaire, formant une arborescence dont les nœuds sont composés de DAs et les feuilles de PAVs.

Dans une telle arborescence, c'est le distributeur d'adresses qui se trouve à la racine qui reçoit du séquenceur l'adresse vectorielle < @, lg > pour un accès contigu ou dispersé, < @, pas, lg > pour un accès avec pas, à charge pour lui de la distribuer à ses fils. Nous constatons que l'adresse vectorielle qui sera ainsi distribuée dépend de la hauteur de l'arborescence. Notons h la hauteur par rapport aux feuilles à laquelle se trouve un distributeur d'adresses (une feuille se trouve à la hauteur h = 0).

Pour un accès contigu ou dispersé, un distributeur d'adresses recevant une adresse vectorielle < @, lg >, et se trouvant à la hauteur h devra transmettre à son $i^{ème}$ fils (les fils étant numérotés de 0 à 3) :

< @ + 4^{h+1} x i, lg_i > avec @ représentant *@za* si c'est un adressage direct contigu, *@zd* si c'est un adressage indirect et

$lg_i = 4^h$ si $Log_4\ lg > i$
$lg_i = r$ si $lg = 4^h + r$ (avec $r < 4^h$)
$lg_i = 0$ si $Log_4\ lg < i$

Pour un accès à un vecteur à composantes espacées, le distributeur d'adresses transmettra à son $i^{ème}$ fils l'adresse vectorielle < @za + 4^h x Pas x i, lg_i > avec lg_i défini comme précédemment. La valeur du pas est directement envoyée aux circuits par le séquenceur (cf. fig. IV.10).

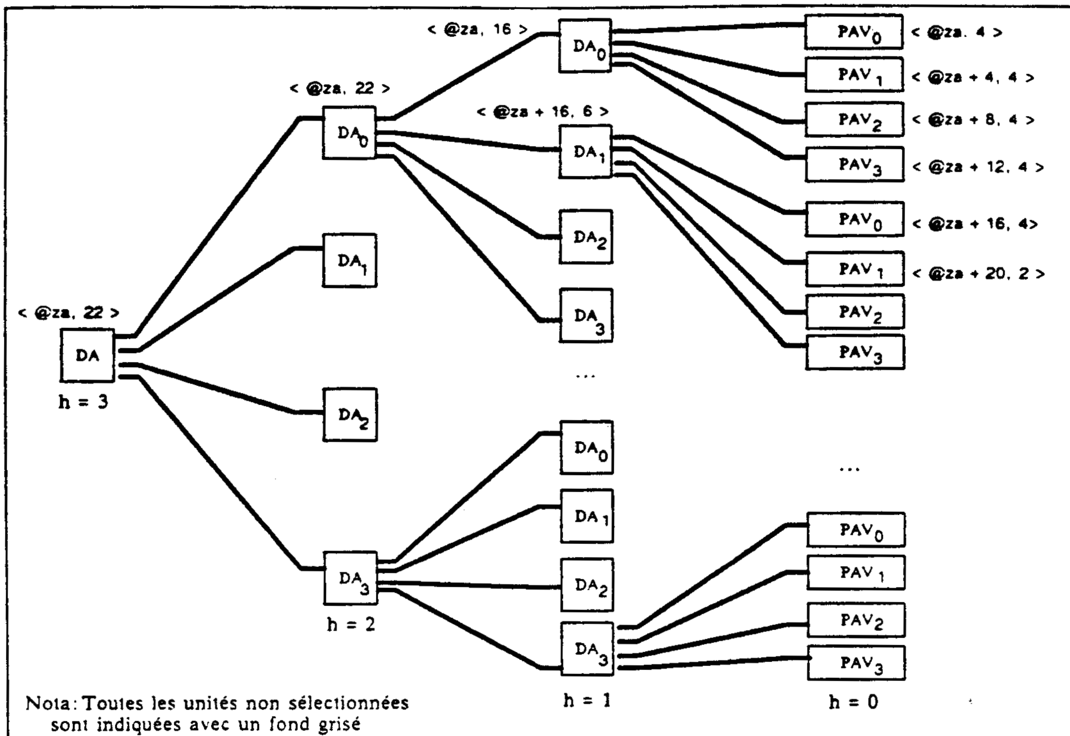


Figure IV.10 -
Arborescence de DAs et de PAVs.
Les adresses reçues par les circuits sélectionnés sont indiquées pour l'accès à un vecteur comprenant 22 composantes.

Le DA et les communications entre circuits DAs et PAVs

Nous décrivons ici le distributeur d'adresses ainsi que les communications entre séquenceur, DAs et PAVs. La synchronisation de ces différents circuits nous amènent à introduire et décrire un dernier type de circuit, le *centraliseur de synchronisations*.

Le circuit distributeur d'adresses

Un circuit distributeur d'adresses se compose de trois unités (cf. fig. IV.11):

- une Unité de Contrôle (UC) qui contrôle l'activité du circuit et commande les autres unités du distributeur d'adresses;
- une Unité de Calcul de LonGueurs (UCLG) qui prend en charge le calcul des lg_i;
- une Unité de Calcul d'Adresses (UCA) qui prend en charge le calcul des adresses de base $@za_i$.

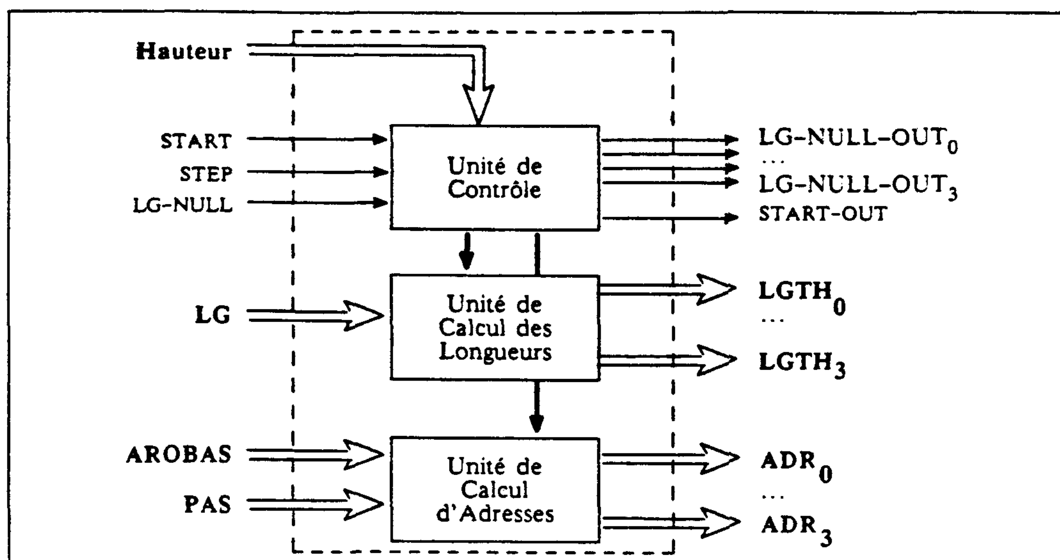


Figure IV.11 -
Structure d'un circuit distributeur d'adresses

L'adresse du vecteur accédé de manière dispersée (*@za* en cas d'accès indirect) est directement envoyée par le séquenceur aux PAVs. Un circuit distributeur d'adresses reçoit en entrée l'adresse de base du vecteur contigu à adresser sur le bus AROBAS. S'il s'agit d'un vecteur à composantes espacées, il reçoit également sur le bus PAS la valeur du pas.

Le signal STEP indique s'il s'agit d'un accès à un vecteur aux composantes réguliérement espacées ou non. Le bus LG véhicule le nombre d'éléments du vecteur à accéder (η). Le signal LG-NULL joue le rôle de sélecteur du circuit. Il valide ou invalide les autres entrées. Le bus Hauteur véhicule la hauteur à laquelle se trouve le circuit dans l'arborescence. Enfin, le signal START déclenche l'activité du distributeur d'adresses. C'est suite à sa réception que les différents signaux et bus sont échantillonnés.

En sortie, un distributeur d'adresses génère quatre signaux LG-NULL-OUT, chacun étant relié à un fils dans l'arborescence. Chacun valide, s'il est à l'état bas, le circuits fils auquel il est relié, jouant pour lui le rôle de signal LG-NULL. Le signal START-OUT est envoyé à tous les fils et joue pour eux le rôle de signal START.

Les bus LGTH et ADR (quatre couples de bus, chacun étant relié à un fils différent) véhiculent des données qui sont envoyées sur les bus LG et AROBAS des fils (ou LG et ADIR si les fils du distributeur d'adresses sont des PAVs). Les valeurs générées sur ces bus sont :

si LG-NULL est à l'état bas et que $0 < LG$, $LGTH_0 \equiv LG \bmod. 4$, $ADR_0 = AROBAS$
sinon les données sur ces bus sont invalides et les quatre signaux LG-NULL-OUT sont à l'état haut;

si LG-NULL est à l'état bas et que $LG > 4$, $LGTH_1 \equiv (LG - 4) \bmod. 4$, $ADR_1 = AROBAS + 4$
sinon les données sur ces bus sont invalides et $LG\text{-}NULL\text{-}OUT_{1,2,3}$ sont à l'état haut;

si LG-NULL est à l'état bas et que $LG > 8$, $LGTH_2 \equiv (LG - 8) \bmod. 4$, $ADR_2 = AROBAS + 8$
sinon les données sur ces bus sont invalides et $LG\text{-}NULL\text{-}OUT_{2,3}$ sont à l'état haut;

si LG-NULL est à l'état bas et que $LG > 12$, $LGTH_3 \equiv (LG - 12) \bmod. 4$, $ADR_3 = AROBAS + 12$
sinon les données sur ces bus sont invalides et $LG\text{-}NULL\text{-}OUT_3$ est à l'état haut;

Notons que tous les signaux issus du séquenceur vers le PAV qui ne sont pas pris en compte par le distributeur d'adresses sont directement envoyés vers les PAVs, comme auparavant. Concernant les signaux de synchronisation, un circuit "centralisateur de synchronisations" est utilisé générant un signal de synchronisation unique lorsqu'un certain nombre de signaux de synchronisation ont été reçus. Ce circuit sera décrit plus loin (cf. fig. IV.12).

Communications entre séquenceur, DAs et PAVs - Problèmes de synchronisation

Certains signaux n'ayant pas à être transformés sont envoyés directement aux PAVs par le séquenceur. Ce sont les signaux WAKE-UP, STEP, READ, IND, BIN, BOUT et le bus AIND. De même, le signal EOMA est directement envoyé par la mémoire à tous PAVs.

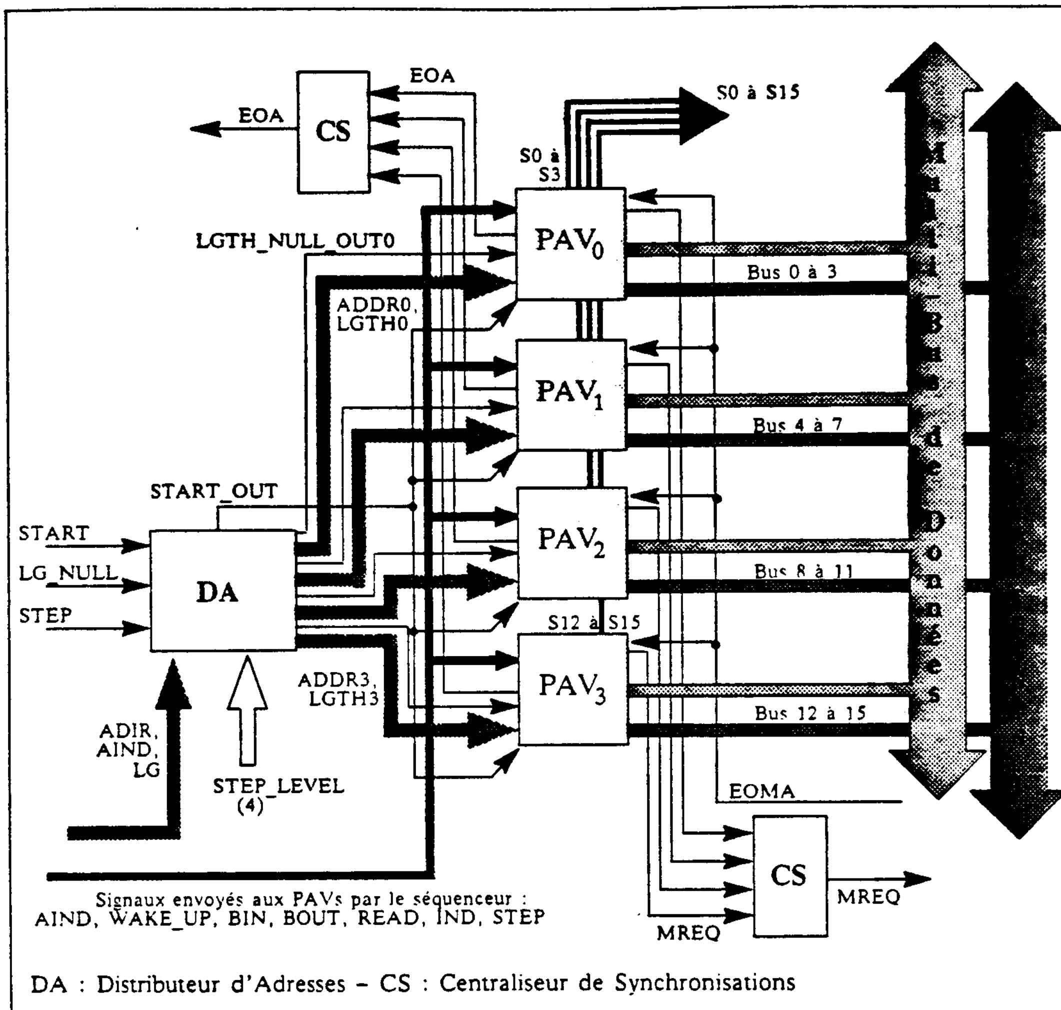
Un PAV non sélectionné doit quand même envoyer les signaux de synchronisation EOA et MREQ. Sinon, les circuits de synchronisation resteraient en attente indéfiniment. Aussi, dès que le signal START ou WAKE-UP est reçu par un PAV non sélectionné, celui-ci émet un signal EOA et un signal MREQ.

Les différents bus de transmission d'informations possèdent tous des tampons nécessaires à la mémorisation de leur contenu lors d'une écriture. Ainsi, une unité plaçant une donnée sur un bus n'a pas à s'occuper de son maintien sur le bus jusqu'à ce que l'unité destinataire du signal le prenne en compte.

Le centraliseur de synchronisations

Un circuit centralisateur de synchronisations reçoit en entrée quatre signaux de synchronisation. Il émet un signal de synchronisation global une fois qu'un signal a été reçu sur chacune de ses entrées. Quand il a émis un signal de synchronisation, il ré-initialise ses entrées en l'attente des synchronisations futures. Les centraliseurs de synchronisation fonctionnent de manière totalement asynchrone vis à vis des autres unités de la machine.

Les centralisateurs de synchronisations sont utilisés pour générer un signal EOA qui sera reçu par le séquenceur par mise en commun des signaux EOA émis par chacun des PAVs. Ils génèrent également le signal MREQ reçu par la mémoire par mise en commun des signaux MREQ émis par chacun des PAVs.



DA : Distributeur d'Adresses - CS : Centraliseur de Synchronisations

Figure IV.12 -
Un PAV modulaire à multi-bus de 16 bus

La figure IV.12 représente un processeur d'adressage vectoriel pour des multi-bus de 16 bus. Il se compose de 4 PAVs de base, d'un distributeur d'adresse relié au sequenceur et de deux centraliseur de synchronisation. L'un émet un signal général de fin d'activité des PAVs, l'autre un signal général d'activation de la mémoire. Le distributeur d'adresses reçoit la valeur quatre sur son bus STEP_LEVEL, indiquant que les PAVs auxquels il est relié traitent quatre composantes chacun. On constate aussi la distribution de couples (ADIR, LG) aux PAVs par le distributeur d'adresses. Les autres signaux porvenant du séquenceur sont directement envoyés aux PAVs.

6. Conclusion de la partie II

Dans cette deuxième partie, nous avons tout d'abord présenté le langage intermédiaire vectoriel Devil et sa machine virtuelle associée MAD. Ces définitions sont incluses dans un projet de développement de langage (EVA) autorisant l'expression explicite d'algorithmes vectoriels. Ce type de langage vise la description des algorithmes et des données manipulées plutôt que la description de l'implantation d'algorithmes, comme cela est réalisé habituellement. Les avantages de cette approche sont la portabilité des programmes et la clarté d'expression d'algorithmes vectoriels complexes. Les problèmes de portabilité sont résolus par l'utilisation du langage intermédiaire Devil. Un programme exprimé en Devil contient les informations nécessaires à sa traduction sur une architecture donnée, le code généré devant être de bonne qualité. La qualité ne doit pas souffrir de l'insertion d'une couche intermédiaire. Aussi, les informations présentes dans le source EVA doivent être reflétées dans le programme Devil correspondant. La définition de Devil tient compte de cette nécessité et possède des caractéristiques autorisant l'expression d'informations (attributs des opérandes, temporaires, classes d'objets). De plus, la définition de Devil (et celle de MAD) vise la simplicité afin de ne pas poser de problèmes particuliers de génération depuis le langage vectoriel EVA.

La définition de Devil repose sur la notion de vecteurs. Un vecteur est ici l'association d'une zone d'allocation qui contient ses éléments et d'une zone de description qui sélectionne les éléments effectifs du vecteur. Cette représentation tente de séparer ce qui concerne l'accès aux composantes des vecteurs de leurs traitements. Le jeu d'instructions de Devil a été défini de manière orthogonale et MAD est une machine-intersection. Il possède la forme générale d'un langage d'assemblage classique avec des notions liées aux langages de triplets. Ceci simplifie la compilation d'un programme en Devil. En outre, notre approche au niveau du langage EVA consiste à donner au programmeur les moyens d'exprimer ce qu'il sait sur l'algorithme et les données qu'il traite. Ces informations simplifient l'analyse du source que recouvre habituellement la phase de vectorisation. Toutes ces informations sont transmises et exprimées dans la représentation en Devil. Ainsi, on notera la notion de fonctions, les attributs des opérandes, les temporaires, le découpage en blocs vectoriels et parallèles, ... Par ailleurs, les définitions de MAD et Devil tentent d'être simples, afin de ne pas poser de problèmes artificiels lors de la compilation, lesquels seraient dus à leurs caractéristiques. Ainsi, la mémoire de MAD n'est pas hiérarchisée et ne pose aucun problème d'allocation des objets, MAD est une machine à registre et non à pile, ...

Après qu'une maquette de Devil ait été implantée sur station de travail, des études d'implantation de Devil sur Cray Y-MP et Connection-Machine sont en cours et déboucheront très prochainement sur une réalisation effective. Au delà de ces mises en œuvres, nous sommes conscients que le futur de Devil passe forcément par la prise en compte des caractéristiques qui sont apparues récemment sur les machines vectorielles. Il s'agit d'une part du parallélisme (multi-processeurs vectoriels Cray), d'autre part des mémoires hiérarchisées (multi-processeurs à structure hiérarchique du type Cedar).

Le parallélisme à grain fin pourra être pris en charge via les blocs parallèles. Les traitements sur les morceaux des vecteurs seront répartis sur les processeurs comme cela est réalisé habituellement. Nous rejoignons là les techniques d'"auto-tasking" développées par Cray. La prise en compte du parallélisme à gros grain (découpage d'une tâche en sous-tâches) nécessitera l'adjonction de constructions à Devil. Nous préférons l'ajout de constructions à l'ajout d'instructions de création de tâches, synchronisation, ... Nous demeurons ainsi dans l'esprit "déclaratif" du langage. Les programmes Devil refléteront ainsi les aspects parallèles des algorithmes, à charge au traducteur de générer le code adéquat à chaque cible. Il pourra alors conserver le même parallélisme si celui-ci

correspond au degré de parallélisme accessible sur la cible, ou "dé-paralléliser" l'algorithme pour des cibles possédant un parallélisme potentiel de degré moindre, ou même des cibles à architecture scalaire.

La prise en compte des mémoires hiérarchisées est également importante. Déjà, l'utilisation des temporaires dans Devil permet une distinction entre les données au niveau de leur rangement en mémoire. Deux approches sont envisageables. Un source Devil peut fournir des informations supplémentaires sur l'allocation des données. La difficulté réside dans le fait que la meilleure allocation des données peut ne pas être fixe durant l'exécution d'un programme. L'autre approche consiste à analyser le source Devil afin de déterminer l'allocation des données dans une zone de code qui entraînera les meilleures performances à l'exécution.

Dans le chapitre IV, nous avons proposé une architecture pour une machine MAD. Pour cela, nous l'avons défini comme une machine tableau. Un séquenceur contrôle l'activité des différentes unités. Nous nous sommes focalisés sur l'étude de la mémoire. Celle-ci est organisée de telle manière que des vecteurs du genre de ceux manipulés par MAD y soient directement accessibles. Afin d'en simplifier l'étude, nous nous sommes intéressés uniquement au cas des vecteurs décrits par une liste d'index, des vecteurs accédés par pas et les vecteurs contigus. Pour construire cette mémoire, nous proposons un circuit de calcul des adresses des composantes des vecteurs, le PAV. Ce circuit prend en entrée un triplet < adresse de base du vecteur, adresse de base de la liste d'index, nombre d'éléments > ou < adresse de base du vecteur, pas, nombre d'éléments > et réalise l'accès aux composantes. Le calcul des adresses est donc effectué en parallèle avec l'activité des processeurs élémentaires. Une fois les données accédées, le séquenceur déclenche l'accès des processeurs élémentaires à celles-ci. Un circuit a été défini réalisant la fonction de PAV. Pour des raisons de connectique, nous avons limité la taille des vecteurs traités par un circuit PAV à 4 composantes. Cependant, le PAV est défini de façon modulaire. Il est possible d'associer plusieurs PAVs (jusqu'à 4) afin de traiter des vecteurs plus longs. De telles associations de PAVs peuvent à leur tour être associées pour traiter des vecteurs toujours plus longs.

Dans une architecture de type tableau, l'utilisation d'un PAV est donc intéressante pour plusieurs raisons. Tout d'abord conceptuellement parlant, elle permet une nette distinction entre traitements et accès aux objets en mémoire. Par ailleurs, les vecteurs pris en compte ne sont plus de simples tableaux mais des objets vecteurs Devil sous une forme simplifiée. De fait, l'adressage des vecteurs par le séquenceur de la machine est simple : il n'adresse pas chaque élément mais l'ensemble des éléments effectifs. Par ailleurs, l'accès dispersé à un vecteur via un vecteur d'index ne nécessite pas que ce dernier réside dans le processeur. Enfin, du fait de la séparation entre traitements et accès mémoires, la préparation du prochain accès mémoire vectoriel est réalisée en parallèle avec l'activité des unités de calculs.

Des extensions et modifications dans la définition du PAV sont envisageables. La prise en compte de vecteurs MAD complets en est une. Pour cela, il faudrait modifier le PAV pour qu'il accepte des accès via un vecteur (liste) de bits. Le problème de ce type d'accès est que contrairement aux accès pris en compte à l'heure actuelle, le nombre de composantes du vecteur n'est pas connu à l'avance. Il est égal au nombre de bits à un de la liste de bits. Cette longueur du vecteur accédé devrait alors être fournie en résultat par le PAV. Mis à part cela, la manipulation d'en-têtes MAD ne poserait pas de problème. Il simplifierait d'ailleurs le protocole de communications entre le séquenceur et le PAV : l'adresse de l'en-tête du vecteur suffirait alors pour accéder ses composantes. Même les vecteurs temporaires pourraient être accédés de cette manière. Ce sont des vecteurs contigus. Il suffirait d'un signal indiquant si le vecteur accédé est temporaire ou non. Une prise en compte automatique par le matériel pourrait résoudre les problèmes dus à la longueur arbitraire des vecteurs MAD. L'accès à l'ensemble de leurs composantes pourrait être effectué en plusieurs temps.

L'adaptation d'un PAV a une machine pipeline est également réalisable. Le processeur pipeline émettrait des requêtes d'accès. Le PAV calculerait plusieurs adresses des composantes du vecteur

en parallèle (4 par exemple) et émettrait les requêtes d'accès effectifs à la mémoire. Pour traiter des vecteurs plus longs, on peut alors construire un PAV de plus grande taille en utilisant la modularité du PAV. Puisque l'on se trouve dans un environnement pipeline, on peut également n'utiliser qu'un seul PAV qui déclencherait plusieurs accès successifs à la mémoire. Dans les deux cas, les composantes sont alors remises en séquence pour leur chargement dans le processeur. Par rapport à l'utilisation habituelle des ports mémoires sur un processeur type Cray, nous gagnons un certain parallélisme dans le calcul des composantes.

Partie III - Le traitement vectoriel désordonné

Dans cette dernière partie, nous décrivons un modèle original d'exécution vectorielle pipeline dite *désordonnée*. Son but est de minimiser la durée des accès mémoires vectoriels en essayant d'accéder les composantes des vecteurs dans l'ordre où elles sont accessibles sans conflit en mémoire, et non dans un ordre fixé. Par ailleurs, ce gain au niveau des accès mémoires est conservé à l'exécution des instructions dans les unités fonctionnelles par le processeur.

Ce modèle d'exécution concerne les processeurs possédant des unités fonctionnelles vectorielles pipelinées, fonctionnant en mode registre à registre. Notre étude concerne une machine possédant une organisation générale (processeur et mémoire) de type Cray X-MP. La mémoire y est découpée en bancs entrelacés, ces bancs étant regroupés en sections. Le modèle est adaptable à d'autres types d'organisations mémoires et aux machines mémoire à mémoire. Ces points ne sont pas décrits ici. Ils pourront faire l'objet d'études ultérieures.

Après avoir défini le modèle de fonctionnement, nous donnons quelques éléments des conséquences matérielles sur les unités d'un processeur vectoriel pipeline pour implanter ce mode d'exécution. Nous terminons notre exposé avec quelques résultats de simulations, certains montrant des gains en performances intéressants.

Une première présentation du modèle d'exécution vectorielle désordonnée se trouve dans [DEKEYSER & al. 90e]. Le traitement vectoriel désordonné devrait faire l'objet du cœur de la thèse de T. Kechadi dont la soutenance est prévue fin 1991. Les aspects présentés ici y seront largement développés et complétés.

Chapitre V
Le traitement vectoriel désordonné

1. Principe

Par définition (en informatique), un vecteur est un ensemble ordonné d'éléments; c'est un tableau de nombres. De cette définition découle le principe du traitement vectoriel pipeline classique qui peut s'exprimer par :

« Les composantes du vecteur résultat d'une opération vectorielle sont calculées en séquence. Le calcul de la $i^{ème}$ composante du résultat ne débute qu'après le déclenchement du calcul de la $i-1^{ème}$. Les composantes du vecteur résultat sont obtenues dans le même ordre. »

Dans l'énoncé de ce principe, l'ordre des composantes des vecteurs semble être d'une importance capitale. Cependant, cet ordre sur le déclenchement des calculs entraînant un ordre sur l'obtention des résultats ne sert à rien (sinon, et ce n'est pas négligeable, à simplifier l'architecture). Pire, il mène à une dégradation des performances, principalement en ce qui concerne les accès aux vecteurs en mémoire : une requête d'accès à une composante d'un vecteur peut être bloquée par une autre, entraînant un blocage pour l'accès aux composantes suivantes.

Les concepteurs du Cray-2 ont tenté de résoudre ce problème en implantant un accès désordonné en mémoire aux composantes des vecteurs. Ce faisant, l'utilisation d'une telle stratégie a entraîné l'impossibilité de chaîner les pipelines d'entrées/sorties aux pipelines de calculs sur le Cray-2. Celle-ci est causée par l'utilisation de pipelines et d'unités fonctionnelles "classiques" dans le Cray-2 (classique dans le sens où les composantes sont traitées dans l'ordre des indices croissants).

Nous proposons une description d'unités fonctionnelles autorisant le chaînage des pipelines, en utilisant une méthode d'accès désordonné aux éléments d'un vecteur, ceci menant au *Traitement Vectoriel Désordonné* (TVD). Dans un premier temps, nous définissons le TVD à la suite de quoi nous ébauchons la description d'une unité fonctionnelle selon ce principe. Cette ébauche est ensuite progressivement raffinée jusqu'à obtenir la description d'un processeur et d'une mémoire fonctionnant sur le principe TVD.

Le TVD est une technique d'exécution d'opérations vectorielles que l'on pourrait qualifier d'exécution par disponibilité (cette technique n'a pas de rapport direct avec celle qui lui est synonyme en data-flow). Une composante du vecteur résultat d'une instruction dépend d'un

certain nombre de composantes sources. Le TVD consiste à calculer les composantes du résultat dont les sources sont disponibles. Les pipelines fonctionnant selon ce principe demeurent chainables. Aussi, notons qu'en aucun cas, les performances du TVD ne peuvent être inférieures à celle du modèle classique d'exécution ordonnée.

1.1. Exemple de fonctionnement TVD

Un exemple de fonctionnement (fig. V.1) permet de visualiser les différences entre le modèle d'exécution désordonnée et le modèle classique des machines pipelines vectorielles de type Cray X-MP (cf. chapitre I). Pour simplifier cet exemple, considérons une mémoire composée de 4 bancs organisés en 2 lignes. L'opération $\vec{C} = \vec{A}$ op $\vec{B}$ est réalisée avec $\vec{A}$, $\vec{B}$, $\vec{C}$ trois vecteurs de 8 éléments. L'accès mémoire demande 4 cycles, le calcul de l'opération demande également 4 cycles, y compris les transferts entre registres et ports mémoires. Les adresses des composantes de $\vec{A}$, $\vec{B}$ et $\vec{C}$ sont réparties sur les 8 bancs de la façon suivante :

A [1], B [1], C [1], A [2], B [2] et C [2] sont sur le banc 1,
A [3], B [3], C [3], A [4], B [4] et C [4] sont sur le banc 2,
A [5], B [5], C [5], A [6], B [6] et C [6] sont sur le banc 3,
A [7], B [7], C [7], A [8], B [8] et C [8] sont sur le banc 4.

L'exemple a été choisi afin de mettre en valeur le modèle désordonné. Il ne se veut pas nécessairement représentatif du type d'accès généralement réalisés sur un Cray. Les deux types de fonctionnement sont :

1. On considère que les adresses de toutes les composantes des vecteurs sont disponibles dès le lancement de l'exécution de l'opération ($\vec{C} = \vec{A}$ op $\vec{B}$) (figure V.1.a.).

2. On se place dans un environnement pipe-line traditionnel. Une adresse de composante est produite à chaque cycle pour chaque port mémoire. L'algorithme de choix s'applique alors à toutes les adresses des composantes déjà produites et non encore consommées à l'instant donné. Dans l'exemple proposé, les adresses arrivent dans l'ordre des indices (figure V.1.b.).

La partie haute du chronogramme de la figure V.1 représente l'activité mémoire des 4 bancs. La partie basse (cf. fig. V.1.c) représente l'activité des différents étages pipe-linés du processeur. Chaque unité fonctionnelle d'entrées/sorties est dédiée à un vecteur.

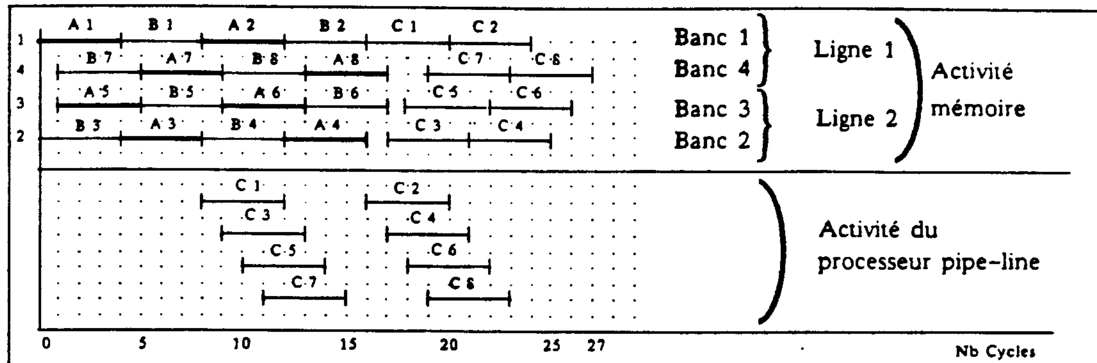


Figure 1.a. Fonctionnement désordonné.

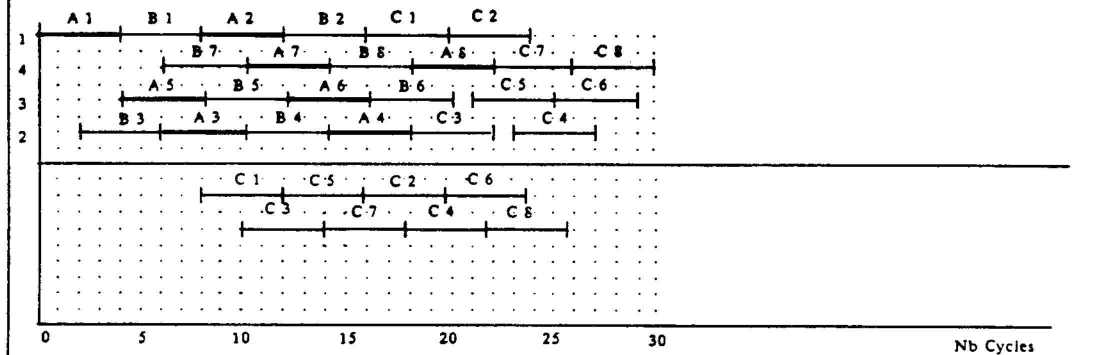


Figure 1.b. Fonctionnement désordonné (une adresse calculée / cycle / port)

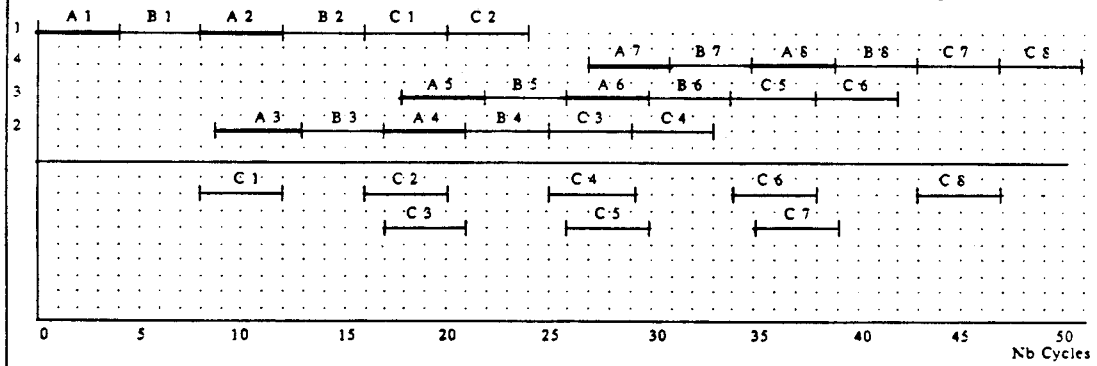


Figure 1.c. Fonctionnement (ordonné) du Cray X-MP

Figure V.1 –
Exécution pipeline désordonnée par rapport à exécution pipeline ordonnée

1.2. Les accès mémoires vectoriels ordonnés

Les gains en performance dus à l'utilisation du TVD sont la conséquence directe de l'accès désordonné à la mémoire. Pour cela, l'accès aux éléments d'un vecteur devant être réalisé, un port mémoire recherche parmi les accès qu'il a à réaliser ceux qui n'entraîneront pas de conflits d'accès. Il sélectionne l'un d'entre eux et émet une requête d'accès à cet élément. Par principe, cette requête ne peut être conflictuelle au sein d'un même processeur. Si le port ne trouve aucun accès qu'il puisse réaliser sans conflit, il n'émet pas de requête avant le cycle suivant.

Les conflits d'accès à la mémoire sont dus à un temps d'occupation de banc (*Bank Busy Time* - bbt) de la mémoire bien supérieur au temps de cycle des processeurs (4 pour le Cray X-MP,

jusqu'à environ 60 pour un Cray-2). Ils varient également selon les types d'accès mémoires effectués (accès contigus, avec pas ou dispersés) et le nombre d'accès concurrents à la mémoire. Sur une machine multi-processeurs organisée à l'instar du Cray X-MP, il existe trois types de conflits d'accès à la mémoire [CHEUNG & al. 86] (cf. chapitre I) :

conflit de section : deux ports mémoires d'un même processeur tentent d'accéder des bancs se situant dans une même section;

conflit de banc occupé : un accès mémoire est tenté sur un banc en cours d'activité;

conflit de simultanéité : deux ports mémoires (chacun sur un processeur distinct) tentent d'accéder un même banc mémoire, au même cycle.

Un port mémoire de Cray X-MP émet une requête à chaque cycle. S'il n'y a pas de conflit, elle est prise en compte par le banc mémoire contenant la donnée. S'il y a un conflit, la même requête est ré-émise au cycle suivant. Tous les ports mémoires de tous les processeurs travaillent en même temps, ce qui peut conduire à un nombre élevé de conflits à chaque cycle.

Dans les paragraphes qui suivent, nous étudions chacun des types de conflit. Nous proposons des solutions autorisant la levée des deux premiers types de conflit. Pour leur part, les conflits de simultanéité ne sont pas résolus, ce type de conflits n'apparaissant qu'en fonctionnement multi-processeur, lequel est ici laissé de côté. Pour commencer, nous présentons le principe général du fonctionnement des ports mémoires.

1.3. Les accès mémoires vectoriels désordonnés

1.3.1. Fonctionnement des ports mémoires

Quand un port mémoire reçoit une requête pour un accès vectoriel, il calcule les adresses des composantes au rythme d'une adresse par cycle. Dès qu'une adresse est disponible, elle est mise dans un tampon. Parallèlement, à chaque cycle le port sélectionne parmi les adresses présentes dans le tampon, une adresse pour laquelle il n'y a pas de conflit et émet la requête. Si un conflit de simultanéité est détecté lorsque la requête atteint le banc mémoire, l'une des requêtes est acceptée, les autres sont bloquées. Les requêtes rejetées par le banc sont laissées dans le tampon d'adresses. La requête acceptée en est retirée.

Au début, le tampon d'adresses est vide et il se remplit peu à peu, au fur et à mesure des requêtes mémoires conflictuelles. Bien entendu, tant qu'il n'y a pas de conflit, le tampon reste vide. En présence de conflit, le tampon se remplit. Aussi, au fur et à mesure, le port possède de plus en plus d'adresses parmi lesquelles il choisit un accès n'entraînant pas de conflit. Au fur et à mesure du remplissage du tampon, le port a donc, statistiquement, une probabilité de plus en plus importante d'émettre une requête. A l'inverse, une fois toutes les adresses des composantes calculées, le tampon a été rempli au maximum et se vide ensuite peu à peu. La probabilité de pouvoir émettre une requête diminue.

Dans des simulations qui nous ont servi d'étalon, nous avons supposé qu'un port mémoire disposait initialement de toutes les adresses des composantes du vecteur à accéder. Comparant cette solution idéale (non envisageable pour une implantation réelle) à la solution réaliste décrite ci-dessus, nous avons constaté que la solution idéale apportait peu de gains par rapport à la solution réaliste (5 à 6 %). Ce résultat tend à montrer que notre solution est très proche de la solution idéale quant à la sélection des éléments à accéder à chaque cycle.

1.3.2. Mécanismes de résolution des conflits

Les conflits de bancs occupés

Chaque banc mémoire indique son état (actif ou non) sur une ligne spécialisée. L'ensemble de ces lignes pour tous les bancs mémoires est utilisé par les ports mémoires pour lever les conflits de bancs occupés.

Les conflits de sections

Les ports mémoires d'un même processeur se partagent des informations autorisant la suppression des conflits de section. A chaque cycle, les ports choisissent une section à accéder, de telle manière qu'il ne puisse y avoir de conflit de section. Un mécanisme de priorité tournante sur les ports mémoires autorise la résolution des conflits d'accès.

Les conflits de simultanéité

La résolution des conflits de simultanéité n'est pas prise en compte. Un mécanisme partagé entre les processeurs et leur permettant de synchroniser leurs accès est requis pour la levée de ces conflits. Ce mécanisme n'est pas étudié ici. Il fera l'objet d'études dans un bref avenir. Quelques remarques à son propos seront trouvées plus loin.

En résumé, nous proposons une suppression des conflits de section et de banc occupé. Si un port peut émettre une requête sans risquer ce types de conflit, elle est émise. Sinon, le port n'émet pas de requête pour ce cycle. Notons que les mécanismes de résolution de conflits doivent nécessairement être rapides, la sélection et l'émission de la requête d'accès à la composante devant être réalisés en un cycle processeur.

La suppression des conflits simplifie la conception des cross-bars reliant les processeurs aux bancs mémoires (pour la résolution des conflits de section) et les bancs mémoires eux-mêmes (pour la résolution des conflits de bancs occupés).

2. Justification du TVD

Nous discutons ici l'adéquation du modèle d'exécution désordonnée aux traitements vectoriels. La question est de savoir si ce modèle peut réaliser les opérations vectorielles de base (les instructions des processeurs du type Cray telles les opérations arithmétiques et logiques).

D'emblée, l'exécution d'instructions du type $C_i := A_i$ op. B_i semblent ne poser aucun problème. Il y a bijection entre l'ensemble des couples (A_i, B_i) et l'ensemble des C_i. A, B et C étant distincts, il n'y a donc aucun problème de dépendances. L'ordre dans lequel sont exécutées les opérations sur les composantes des vecteurs n'a pas d'importance. Aussi, l'exécution de ce type d'opérations pourra éventuellement être accélérée grâce au TVD.

Les instructions de réduction du type $s := f$ (A_1, ... A_{VLG}) où s est un scalaire, ne posent pas plus de problème. Il n'y a aucune dépendance entre les données sources et cibles. Si l'opération réductrice est associative et commutative, la réduction des éléments du vecteur peut s'effectuer dans n'importe quel ordre. Le temps d'accès aux composantes du vecteur source étant minimisé, le temps d'obtention du résultat de f en sera également minimisé.

3. Adaptations matérielles pour le support du TVD

Le support du modèle d'exécution TVD par les processeurs vectoriels pipelines classiques est tout à fait envisageable et ce, au prix d'une assez faible complication au niveau matériel de ces

processeurs. Nous présentons ici les *adaptations* à apporter aux unités de ce type de processeur pour qu'ils supportent l'exécution TVD. Quatre types d'unités sont à adapter :

- les registres vectoriels;
- les unités fonctionnelles de calcul;
- les unités fonctionnelles d'entrées/sorties (ports mémoires);
- les bancs mémoires.

Par ailleurs, une unité d'accès mémoires est ajoutée dans chaque processeur. Chacun des ports mémoires du processeur y est relié.

Enfin, nous avons vu (cf. § 1.3.2.) que le TVD mène à une simplification des bancs mémoires et des circuits d'interconnections entre les ports mémoires des processeurs et les bancs du fait de la suppression des conflits de sections et de bancs simultanés.

3.1. Adaptation des registres vectoriels

A chaque composante d'un registre vectoriel, nous ajoutons un indicateur binaire indiquant si son contenu est valide ou non. Ceux-ci sont initialisés à chaque utilisation de ce registre comme cible d'une instruction. Ils sont alors remis à zéro. Par la suite, à chaque fois qu'une nouvelle composante de ce registre a été chargée, son bit de présence est validé. S'il s'agit d'une opération de chargement de registre depuis la mémoire, ce bit indique que la composante a été lue. S'il s'agit d'une opération calculatoire (arithmétique, logique, ...), ce bit indique que la composante a été calculée.

3.2. Adaptation des unités fonctionnelles de calcul

Chaque unité fonctionnelle est modifiée de deux manières (cf. fig. V.2). Tout d'abord, à chaque étage d'un pipeline est associé le numéro de la composante en cours de calcul à cet étage. Cette information accompagne donc le résultat au fur et à mesure de sa progression dans le pipeline. Elle permet le rangement du résultat dans la bonne composante du registre résultat (ou du port mémoire le rangeant en mémoire si celui-ci est chaîné au pipeline de calcul). Par ailleurs, l'unité fonctionnelle dispose d'un masque (M) dont chaque bit est associé à une composante du résultat et indique si cette composante doit être calculée. A chaque fois que le calcul d'une nouvelle composante du résultat est déclenché (les sources entrent dans le premier étage du pipeline), le bit correspondant du masque est invalidé, indiquant que cette composante n'est plus à calculer. Supposons qu'une instruction traite des vecteurs d'une taille λ. Lors de son déclenchement, le masque est initialisé. Pour cela, les λ premiers bits du masque sont initialisés à 1, les suivants sont initialisés à 0 : les λ premières composantes doivent être calculées mais pas les suivantes.

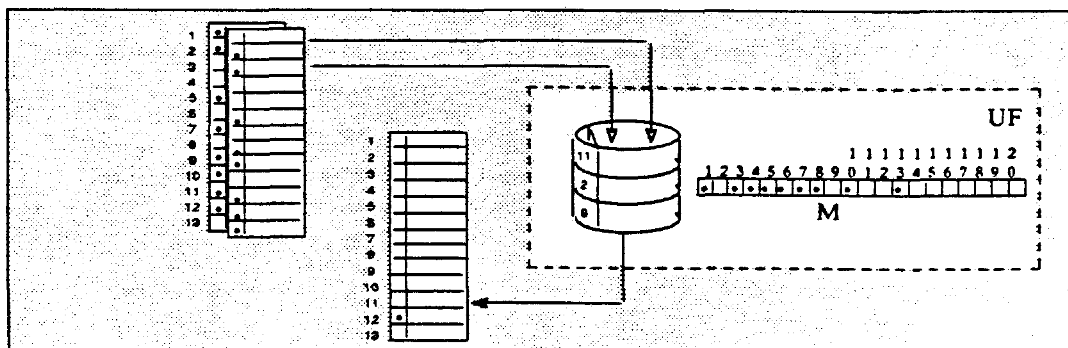


Figure V.2 –
Unité fonctionnelle d'un PVD

Par combinaison binaire (S1 ∧ S2 ∧ M) des bits de son masque (M) et des bits de présence des registres vectoriels sources (S1 et S2), l'unité fonctionnelle peut déterminer un couple de composantes sources disponibles non encore traitées. Elles les chargent dans son premier étage et met à jour son masque. L'instruction est terminée lorsqu'il n'y a plus de bit validé dans le masque de l'unité fonctionnelle et que tous ses étages sont vides.

Notons que la technique de recherche de la composante à calculer n'est pas fixée précisément pour l'instant. La combinaison binaire précédente est un moyen d'y arriver, ce n'est pas le seul. Des études sont en cours pour déterminer une technique efficace, tant au niveau performances (temps de réalisation de l'ensemble de l'instruction) qu'en ce qui concerne la complexité mtérielle.

3.3. Adaptation de la mémoire et des ports mémoires

3.3.1. La mémoire

Chaque banc mémoire possède une ligne sur laquelle il indique s'il est en cours d'activité ou s'il peut accepter une requête d'accès. Chacun de ces signaux est envoyé aux ports mémoires des processeurs.

Par ailleurs, les bancs peuvent être simplifiés puisqu'ils n'ont plus à résoudre les conflits d'accès. Ceux-ci sont résolus dans l'unité d'accès mémoire.

3.3.2. L'unité d'accès mémoire

L'unité d'accès mémoire (UAM) recherche les accès qui doivent être exécutés par les ports. A chaque port contrôlé par cette unité correspond une sous-unité composée d'une unité de calcul d'adresses et d'un tampon (cf. Fig. V.3).

Dans ce qui suit, nous utilisons le terme *couple* qui recouvre, dans le cas d'une lecture, un couple (adresse à lire en mémoire, indice de la composante lue), dans le cas d'une écriture, un couple (adresse où écrire la donnée, donnée à écrire en mémoire).

Les différentes unités sont :

- une unité de calcul d'adresse. Elle génère un couple à chaque cycle. Comme les autres unités fonctionnelles, elle possède un masque indiquant les composantes déjà traitées. Ce masque est initialisé en accord avec la valeur de VL au déclenchement de l'opération d'accès mémoire.

- un tampon. Pour chaque port, il contient des couples pour les composantes non déjà traitées. Un masque indique les données valides du tampon.

- un sélecteur d'adresses. Il utilise le masque des bancs occupés et les informations présentes dans les tampons pour trouver, à chaque cycle, des données accessibles sans conflit. La complexité de cette unité repose sur l'agorithme de choix implanté.

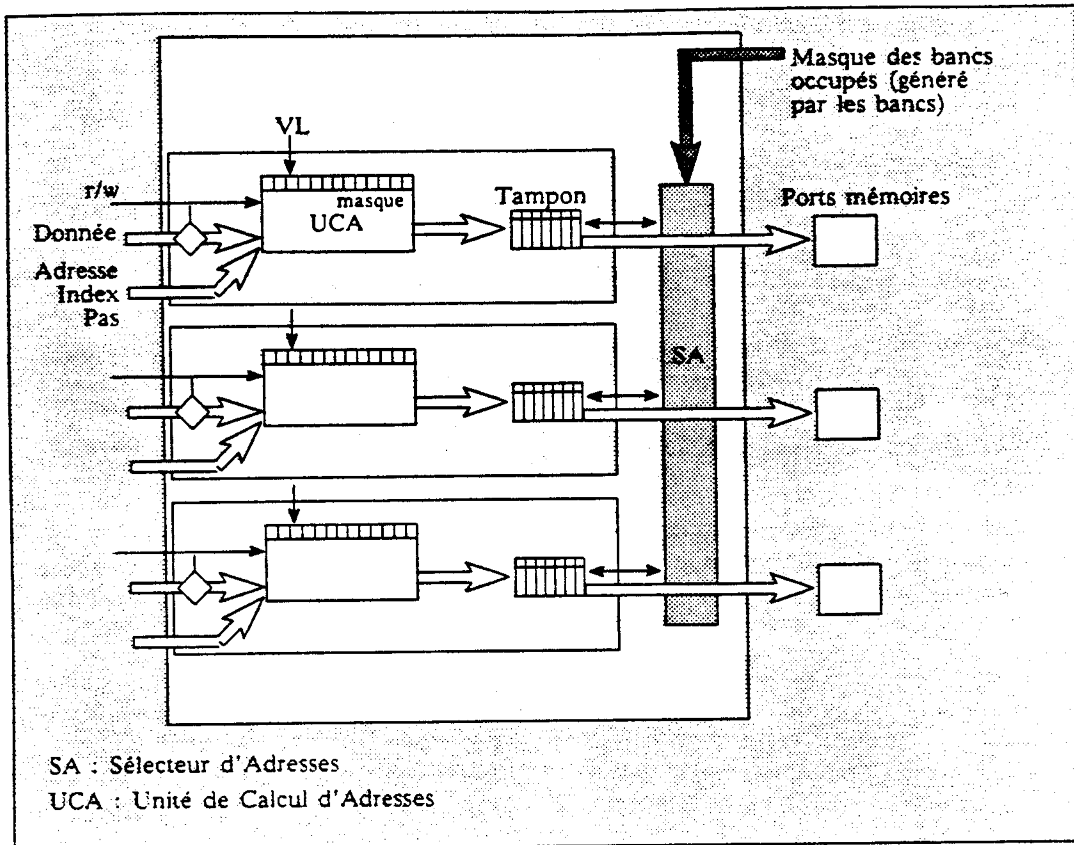


Figure V.3 -
Unité d'accès mémoire (3 ports)

A chaque cycle, les bancs mémoires indiquent leur état (actif ou inactif). L'ensemble de ces informations pour tous les bancs forment le masque des bancs occupés. Notons que l'état d'activité des bancs pourrait être calculé dans l'unité d'accès mémoires. Nous n'avons pas retenu cette possibilite pour deux raisons. D'une part, cela aurait compliqué cette unité (nécessité d'utiliser des décrémenteurs). D'autre part, dans la future extension multi-processeur du mécanisme, ce retour de la mémoire sera intéressant.

3.3.2. Les ports mémoires

Un port mémoire (cf. fig. V.4) prend en charge l'accès effectif à la mémoire, une fois une adresse n'entrainant pas de conflit d'accès découverte par l'unité d'accès mémoire. Il reçoit alors un couple indiquant l'adresse à accéder. Quel que soit le type d'accès effectué (lecture ou écriture), l'indice de la composante est placé dans un registre à décalage. Le contenu de ce registre est décalé à chaque cycle. Il comporte τ positions. Ainsi, lorsque l'accès a été réalisé, l'indice en ressort en même temps que la donnée accédée en mémoire. S'il s'agit d'une écriture, cette sortie de l'indice indique seulement que la donnée est écrite en mémoire. S'il s'agit d'une lecture, la donnée et l'indice forment alors un couple qui est renvoyé dans le processeur, pour que la donnée y soit stockée dans un registre vectoriel.

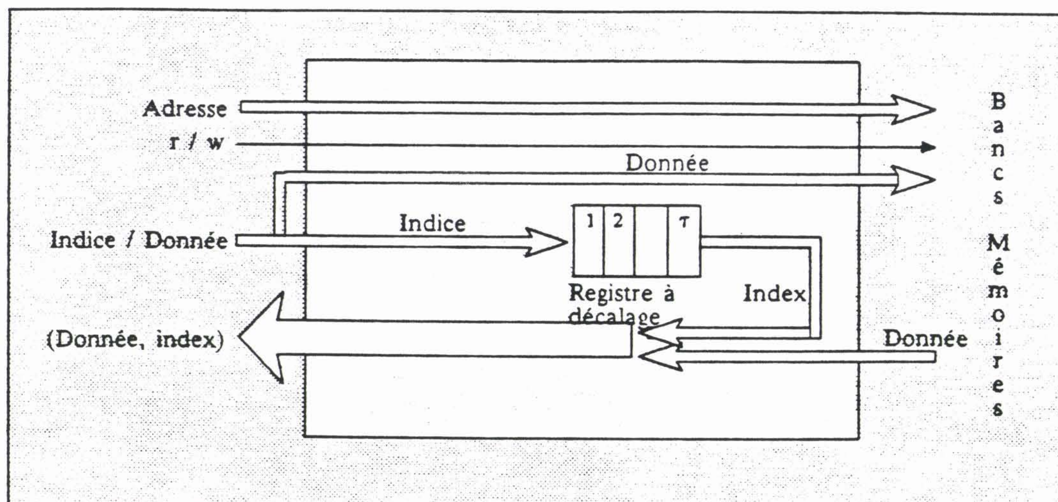


Figure V.4 -
Un port mémoire pour le TVD

La levée des conflits de ligne comporte deux problèmes distincts. D'une part, elle doit indiquer les accès qui ne doivent pas être déclenchés pour éviter les conflits de ligne. Lorsqu'un conflit est détecté, l'un des accès doit être élu, les autres rejetés. Ceci ne pose pas de problème majeur.

D'autre part, plusieurs accès se trouvant en attente dans le tampon, il faudrait que les ports arrivent à une situation dans laquelle ils puissent tous émettre une requête, sans déclencher de conflit de ligne, si cela est possible au regard du contenu des tampons de chaque port. Ceci est plus difficile à réaliser, à la vitesse de fonctionnement des ports mémoires. Par ailleurs, une telle configuration (chaque port émet une requête non conflictuelle avec celles des autres ports) n'existe pas à chaque cycle.

4. Exemple de traitement vectoriel désordonné

Prenons par exemple l'addition vectorielle : $\vec{A} = \vec{B} + \vec{C}$, et supposons que cette instruction se compile dans la suite d'instructions :

```
load    B, V0
load    C, V1
add     V0, V1, V2
store   V2, A
```

où V0, V1 et V2 sont des registres vectoriels. Supposons que le CPU dispose d'au moins trois ports d'accès mémoires (cf. Cray X-MP par exemple) pouvant travailler en parallèle.

Le port 1, piloté par l'unité d'entrée/sortie du port 1 (UP1), prend en charge le `load B, V0`.

Le port 2, piloté par l'unité d'entrée/sortie du port 2 (UP2), prend en charge le `load C, V1`.

Le port 3, piloté par l'unité d'entrée/sortie du port 3 (UP3), prend en charge le `store V2, A`.

L'unité d'addition (UADD) prend en charge l'opération `add V0, V1, V2`.

Le traitement est alors réalisé selon le schéma qui suit (cf. fig. V.5).

- Les bits de présence des composantes des registres vectoriels V0, V1 et V2 sont mis à zéro. Les bits des masques des unités fonctionnelles (UAM et UADD) sont initialisés selon la taille des vecteurs à traiter.

- Les 5 unités UAM, UP1, UP2, UP3 et UADD sont activées en parallèle. Les unités UAM, UP1 et UP2 commencent les opérations d'accès (désordonnés) à la mémoire afin de charger les registres vectoriels V0 et V1. L'UADD est en attente, le contenu des registres V0 et V1 étant pour l'instant invalide. L'unité UP3 (et la sous-unité de l'UAM commandant l'UP3) est également en attente, les bits de présence des composantes du registre V2 étant également à zéro.

- Des composantes des vecteurs $\vec{B}$ et $\vec{C}$ commencent à arriver dans les registres V0 et V1 dans un ordre quelconque. Leurs bits de présence sont mis à un une fois chargée. L'UADD recherche un indice i tel que les composantes $V0_i$ et $V1_i$ soient chargées (d'après les bits de présence) et tel que la ième composante du résultat n'ait pas déjà été calculée (d'après le masque de l'unité fonctionnelle). Dès qu'un tel couple est trouvé, celui-ci entre dans le premier étage du pipeline de l'UADD. A ce couple, elle associe son indice (i) et mémorise que le calcul du ième résultat est lancé (le bit i du masque interne à l'unité fonctionnelle est mis à 0). L'UADD se met ensuite à la recherche d'un autre indice j, tel que $V0_j$ et $V1_j$ soient disponibles et non déjà utilisées, et ainsi de suite ...

- Lorsqu'un résultat a été produit de l'UADD (toujours accompagné de son indice), il est dirigé vers le registre vectoriel résultat de l'opération (V2) où il est rangé. Le bit "information valide" correspondant à cette composante est positionné.

- De son côté, l'UAM recherche (pour l'UP3) dans le registre V2 les composantes ayant leur bit de présence positionné. Dès qu'une nouvelle composante est rangée dans V2, l'UAM la place dans son tampon correspondant à l'UP3. L'UAM sélectionne une requête n'entraînant pas de conflit s'il y en a et l'émet. Si une requête peut être émise pour la ième composante, l'UAM mémorise l'indice de cette composante par positionnement du bit i de son masque. Le couple ($Adresse_i$, $Donnée_i$) est envoyé vers le port mémoire 3.

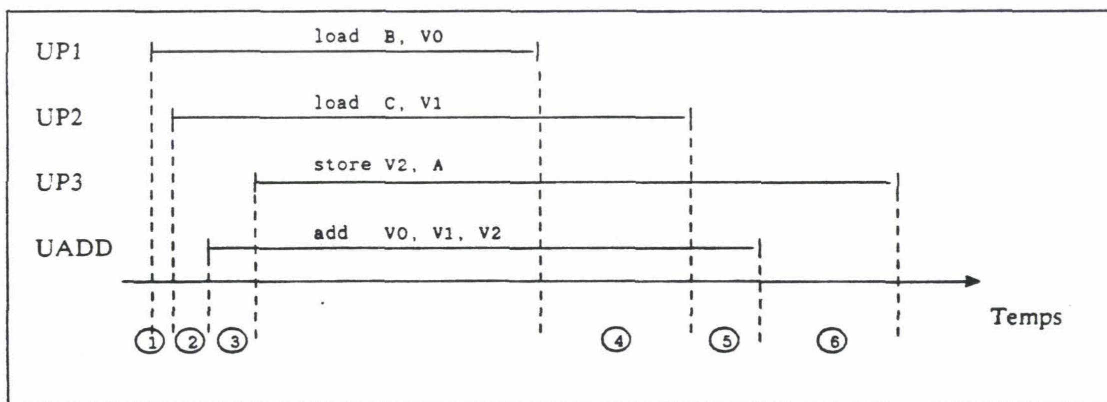


Figure V.5 -

Activité des unités du PVD durant l'opération $\vec{A} = \vec{B} + \vec{C}$

Les délais notés sur la figure V.5 sont dus aux causes suivantes :

① délai d'initiation d'une instruction;

② délai nécessaire à l'obtention d'un couple de composantes sources une fois les deux opérations de chargement déclenchées (très variable selon les types d'accès);

③ temps de latence du pipeline d'addition;

④ délai dépendant des conflits mémoires et des types d'accès (selon les cas, UP1 termine avant, après ou en même temps que UP2);

⑤ temps de latence du pipeline d'addition;

⑥ fin de rangement en mémoire des composantes de la cible. Ce délai dépend des conflits mémoires.

5. Simulations – Résultats

Nous avons réalisé une simulation du TVD. Les premiers résultats sont assez encourageants. Nous allons les présenter et les discuter ici rapidement. Les études seront poursuivies par T. Kechadi. Quelques uns de ces résultats sont présentés dans [DEKEYSER & al. 90e].

5.1. Gains attendus du traitement désordonné

Nous avons simulé l'activité de la mémoire, des ports mémoires, des pipelines de calcul lors de la réalisation d'opérations du type $\vec{A} = \vec{B}$ op. $\vec{C}$ consistant à charger les deux vecteurs $\vec{B}$ et $\vec{C}$ dans les registres, à les traiter (op.) et à écrire le résultat en mémoire. Le type de traitement effectué importe peu. Seul importe le temps de latence du pipeline ou de la chaîne de pipelines les effectuant.

Nous comparons le temps de réalisation de ce traitement sur une machine de type Cray X-MP (traitement ordonné) et sur une machine possédant le même type d'unités fonctionnelles mais fonctionnant de manière désordonnée. L'unité mémoire est semblable, mis à part les retours d'informations pour la résolution des conflits de bancs occupés décrits plus haut. Les courbes représentent le gain :

$$\gamma = \frac{\theta_1 - \theta_2}{\theta_1}$$

où θ_1 représente le temps d'exécution pour un traitement ordonné, θ_2 pour le traitement désordonné. La simulation est réalisée en mode mono-processeur. Par ailleurs, nous avons choisi les paramètres suivants :

traitements de vecteurs de 64 composantes;

32 bancs mémoires;

les bancs sont organisés en 4 sections de 8 bancs;

le temps d'occupation des bancs est de 4 cycles-processeurs;

le temps de latence du pipeline réalisant *op.* est de 4 cycles-processeurs;

Nous simulé le fonctionnement ordonné du type Cray X-MP et le fonctionnement désordonné, sur la même configuration matérielle. Les vecteurs sont accédés de manière pseudo-aléatoire. Des

accès contigus, avec pas ou dispersés sont réalisés avec une équi-probabilité. Le banc où se trouve leur première composante est aléatoire. Les valeurs des pas ou des indices du vecteur d'index en cas d'accès dispersé sont aléatoires. Les types d'accès aux vecteurs sont les mêmes dans les deux simulations, ordonnée et désordonnée.

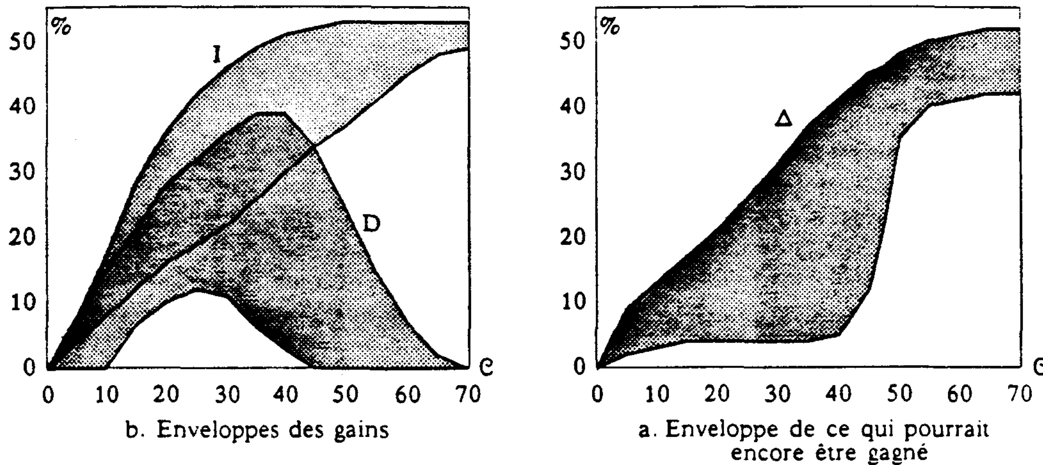


Figure V.6 -
Enveloppes des gains du modèle désordonné sur le Cray X-MP (D) et idéal (I) et différence Δ

Nous avons simulé 40.000 opérations $\vec{A} = \vec{B}$ op. $\vec{C}$. Nous avons obtenu les trois enveloppes I, D et Δ (cf. fig. V.6). L'abscisse indique le taux de conflit ℭ défini par :

$$\mathfrak{C} = \frac{\text{Nombre de conflits durant une exécution ordonnée}}{\text{Nombre de tentatives d'accès}}$$

L'ordonné indique le gain. L'enveloppe D représente le gain du modèle désordonné par rapport au modèle ordonné classique. L'enveloppe I, représente un gain dit "idéal" qui pourrait idéalement être gagné sur le modèle désordonné dans le cas où il n'y aurait aucun conflit d'accès à la mémoire (mémoire idéale dont les mots-mémoires supporteraient de multiples accès concurrents). L'enveloppe Δ indique ce qui peut encore être gagné "idéalement". C'est l'ensemble pour i variant de 1 à 40.000 (i est le numéro de la simulation), des points $\delta_i = I_i - S_i$, où S_i et I_i sont respectivement les gains du modèle désordonné sur le modèle ordonné et le gain idéal.

Grâce à ces enveloppes, on constate qu'il y a une valeur de ℭ (30 à 40 % de conflits) pour laquelle l'intérêt de notre modèle d'exécution désordonnée est maximal. C'est une valeur pour laquelle de nombreux conflits sont artificiellement créés du fait de l'ordre habituel des accès. Pour des valeurs de taux de conflits supérieures, nous arrivons à des cas où les accès visent un sous-ensemble des bancs de la mémoire ce qui entraîne de nombreux conflits de bancs occupés. Ceci est bien montré par le fait que l'enveloppe Δ atteint une valeur maximale pour ces valeurs du taux de conflits.

5.2. Importance du temps d'occupation des bancs

Nous avons réalisé des simulations en faisant varier le temps d'occupation des bancs élevé par rapport au temps de cycle du processeur (cf. Cray-2 où le temps d'occupation varie d'environ 20 à 60 cycles cpu, selon le type de mémoire - dynamique ou statique - et selon que la technique de

pseudo-banking est utilisée ou non). L'augmentation de ce paramètre amenuise le gain qui peut être attendue de l'exécution désordonnée. Nous avons simulé les temps d'exécution d'opérations $\vec{A} := \vec{B}$ op. $\vec{C}$, où $\vec{A}$, $\vec{B}$ et $\vec{C}$ sont résidants en mémoire et où les pipelines d'entrées/sorties et le pipeline de calcul de op. peuvent être chaînés les uns aux autres.

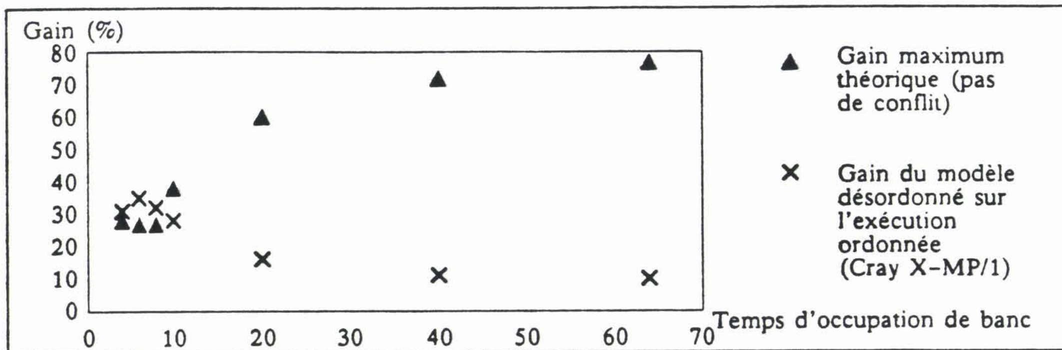


Figure V.7 -
Gains obtenus par une exécution désordonnée par rapport à une exécution ordonnée, en faisant varier le temps d'occupation des bancs.

Nous avons considéré l'organisation mémoire du Cray X-MP (et non celle du Cray-2) : chaque port mémoire a accès à l'ensemble de la mémoire, à tout moment. Nous nous sommes placés dans un contexte mono-processeur, le vecteur $\vec{A}$ étant contigu, le vecteur $\vec{B}$ ayant ses éléments séparés d'un pas de 4, ceux de $\vec{C}$ d'un pas de 3. L'analyse du gain attendu par utilisation d'une exécution désordonnée par rapport à une exécution ordonnée du type Cray X-MP en fonction du bbt de la mémoire est présenté à la figure V.7. Nous avons également fait figurer le gain maximum qui peut être attendu en considérant qu'aucun conflit d'accès mémoire n'a lieu, par rapport à l'exécution désordonnée. Nous constatons que pour des temps d'occupation de bancs faibles (moins de 10 cycles cpu), le gain du modèle désordonné est élevé (supérieur à 30 pour-cent). Le gain décroit ensuite jusque 10 pour-cent pour un temps d'occupation de bancs de 64 cycles cpu. Notons que l'explication complète de la forme des courbes fait intervenir les types d'accès effectués. Des phénomènes de "résonances" dus aux valeurs des pas d'accès avec le nombre de bancs apparaissent.

6. Conclusion sur le traitement vectoriel désordonné

Nous avons décrit un modèle de fonctionnement vectoriel pipeline original, dit désordonné. Dans ce domaine, l'approche classique consiste à résoudre les conflits d'accès mémoires. Notre approche est une alternative consistant à rechercher des accès mémoires n'entraînant pas de conflit (donc de blocage des unités foncitonnlles) et à les émettre en priorité. Nous avons montré que le traitement désordonné pouvait être mis en place au prix de modifications assez simples apportées aux unités des processeurs vectoriels actuels du type Cray X-MP. Des résultats de simulations montrent des gains en performances (vitesses d'exécution d'instructions vectorielles) très intéressants. Par ailleurs, il est clair que ce mode de fonctionnement ne peut pas être un handicap, les accès ne pouvant pas être plus lents que ceux de l'approche ordonnée.

Le Cray-2 autorise également un accès désordonné aux composantes des vecteurs en mémoire (sa mémoire est d'un type différent de celui que nous avons défini). Cependant, cela supprime les possibilités de chaînage de pipelines. Dans notre modèle, les pipelines peuvent encore être chaînés, de manière tout à fait classique.

Avec le traitement désordonné, les conflits de sections, de bancs occupés et de simultanéité sont supprimés : les accès entraînant un conflit ne sont pas déclenchés. Pour cela, les ports mémoires doivent s'échanger des informations très rapidement, chacun devant émettre une requête par cycle. Les simulations nous ont montré l'importance fondamentale de la prise en compte des conflits de simultanéité, laquelle nécessite des communications inter-processeurs. Ceci constitue un premier axe d'études menées actuellement.

Un deuxième axe d'études concerne la sélection des composantes des vecteurs à accéder. Selon le traitement qui est à réaliser, le choix des composantes entraîne des modifications quant au temps total de réalisation de l'opération. Pour un traitement du type $A_i := B_i$ op. C_i, si la ième composante du vecteur $\vec{B}$ est déjà chargée et non la ième composante du vecteur $\vec{C}$, il semble préférable de charger en priorité C_i si cela est possible (pas de conflit) plutôt qu'une autre composante (C_j) alors que B_j n'est pas déjà chargée. Ainsi, la sélection de la composante à accéder doit idéalement faire intervenir les composantes déjà accédées de même que les conflits mémoires qui peuvent être prévus pour les cycles à venir. Ceci mène à une grande complication du matériel et le temps de sélection des composantes n'est pas négligeable. Cependant, face à des algorithmes de choix beaucoup plus simples, des simulations récentes semblent indiquer que le gain qui peut être attendu de l'utilisation d'une telle technique de choix n'en vaut peut-être pas le coût matériel.

Par ailleurs, les études qui restent à mener sur le traitement vectoriel désordonné sont multiples. D'une part, les conséquences de l'organisation de la mémoire sur les performances attendues du TVD doivent être étudiées. L'organisation en sections/sous-sections du Cray Y-MP, ou celle du Cray-2 ont-elles des conséquences importantes sur l'intérêt du TVD ? Des mémoires où les adresses sont réparties pseudo-aléatoirement (cf. Cydra) doivent être prises en compte. Des architectures hiérarchisées du type du Cedar doivent également être considérées.

L'étude du TVD dans un environnemem multi-processeurs est en cours. La multiplication du nombre de processeurs est la solution actuellement prisée par les constructeurs pour obtenir des performances plus élevées. Notre modèle de fonctionnement doit conserver son intérêt dans ce cas. La résolution des conflits de simultanéité est alors un point crucial à étudier.

La réalisation matérielle est à l'étude. Il faut montrer d'une part la faisabilité du traitement désordonné au rythme de fonctionnement des processeurs vectoriels pipelines (quelques nano-secondes). D'autre part, il faut montrer que la complexité matérielle n'est pas disproportionnée par rapport au gain que l'on peut attendre de ce mode de fonctionnement. Des simulations utilisant les techniques VHDL seront effectuées.

Enfin, un modèle analytique du fonctionnement désordonné est à l'étude. Celui-ci est assez compliqué puisque notre modèle d'exécution n'est intéressant qu'à la condition que plusieurs accès concurrents à la mémoire soient effectués, ou que des accès par pas ou dispersés soient réalisés. Hors, ces types d'accès sont ceux que l'on a du mal à modéliser à l'heure actuelle pour des machines de type Cray X-MP. A l'aide de ce modèle, nous espérons pourvoir formuler le gain du TVD par rapport au fonctionnement du Cray X-MP. L'approche actuelle se basant sur des simulations pourrait être utilisée dans un premier temps afin d'obtenir un modèle expérimental.

Conclusion générale

Ce document s'insère dans un projet s'intéressant en premier lieu à la programmation d'algorithmes vectoriels. Cette étude est développée de manière indépendante de tout matériel particuliers, tout en gardant présent à l'esprit les problèmes d'implantation. Notre objectif est l'exécution de ces algorithmes sur toute catégorie d'ordinateurs, non nécessairement les machines à architecture vectorielle.

Dans notre approche, les algorithmes vectoriels s'expriment via un langage de haut niveau vectoriel, EVA. Dans ce langage, les vecteurs sont traités en tant que tels. Des structures algorithmiques adéquates en permettent des manipulations évoluées. Comme il est maintenant tout à fait reconnu, l'intérêt d'un langage de programmation passe forcément par sa portabilité sur différentes machines. Un langage intermédiaire, Devil, a donc été proposé pour résoudre les problèmes de portabilité. Devil autorise lui-aussi la manipulation directe de vecteurs, quoique d'une manière plus "primitive" que EVA.

C'est la description de Devil et sa capacité à se compiler dans du code efficace qui sont au cœur de ce document. Nous avons indiqué et justifié les choix qui ont guidé la définition de Devil. Devil est un langage orthogonal et suit le principe des machines-intersections. Ses caractéristiques rendent sa génération depuis un langage vectoriel de haut niveau assez simple. Nous avons ensuite indiqué en quoi la structure de Devil autorisait la génération de code efficace. Nous avons étudié la génération de code scalaire. Cette étude a mené à une implantation d'un traducteur Devil sur stations de travail. Nous avons étudié la génération de code pour les machines vectorielles pipelines. Cette étude débouchera prochainement sur une implantation d'un traducteur Devil sur le Cray Y-MP/432 de l'Université de Floride.

D'autres études (à mener) pourraient déboucher sur l'implantation de Devil sur machines tableaux (Connection Machine) ou les multi-processeurs à mémoire distribuée (hypercubes Intel). Par ailleurs, des études qui nous semblent importantes concernent les multi-processeurs pipelines. Celles-ci concerneront donc d'une part des machines multi-processeurs à mémoire partagée du type Cray-2, X-MP et Y-MP, d'autre part, les machines à structure arborescente du type Cedar, RP3 ou NYU Ultracomputer. Des problèmes non triviaux apparaissent pour l'implantation de Devil sur machines Cray si l'on désire partager automatiquement les traitements entre les processeurs. Déjà sensible pour ces machines à mémoire partagée du fait de la présence de caches et de registres vectoriels, la distribution des données pour des machines à mémoire hiérarchisée (Cedar, ...) devient un problème fondamental pour l'utilisation optimale des capacités de ces machines. La définition de langages de programmation adaptés et de techniques de génération de code afin de résoudre ce problème automatiquement s'amorce depuis quelques années en parallèle à la

définition architecturale de ces machines. Une étude statique (réalisée à la compilation) des flux de données dynamiques (à l'exécution), afin d'assurer une bonne distribution des données en mémoire, semble (très) difficile à réaliser à partir des langages actuels. La solution passe probablement par la définition de nouveaux langages. Se posent alors les problèmes de ré-utilisabilité de l'existant.

Notons que les problèmes de répartition des données sur une machine à mémoire distribuée du type de la Connection-Machine sont à étudier. Par ailleurs, l'utilisation des processeurs élémentaires de manière efficace pose un problème lorsqu'un programme traite des vecteurs de petite dimensions : comment utiliser efficacement une CM avec 64 K processeurs pour traiter des vecteurs de 10 ou 100 éléments ?

Derrière les problèmes de programmation, les problèmes d'architecture sont présents en filigrane. Le désir de générer du code de bonne qualité (développement des vectoriseurs) à entraîner des réflexions qui ont eu des conséquences sur l'architecture des ordinateurs. Certaines opérations de nature scalaire à l'origine des machines vectorielles sont aujourd'hui vectorisables (accès vectoriels dispersés, récurrences linéaires, ...). L'étude des flux de données à entraîner la multiplication des ports mémoires alimentant les unités fonctionnelles des processeurs, le règne actuel des machines vectorielles registre à registre et des évolutions de l'organisation des mémoires (découpage en sections, utilisation de caches, ...). Suivant ce type de points de vue, une évolution de la notion de vecteurs nous semble importante dans l'avenir. La transformation des vecteurs-tableaux en vecteurs de type Devil permet de reporter une partie des informations habituellement algorithmiques dans les structures de données. Nombre d'auteurs "réclament" ou constatent un besoin des scientifiques (théoriciens) de possibilités de définition et d'utilisation de structures de données évoluées dans les langages qu'ils utilisent. Notons que cette tendance s'inscrit dans le mouvement général de déplacement des connaissances se trouvant dans les programmes des procédures vers les structures de données (émergence des langages orientés objets, ...). Aussi, l'étude des mémoires vectorielles et de leur réalisation pourrait amener des résultats intéressants. Plus généralement, il nous semble que la prise en compte au niveau matériel de la gestion des structures de données (accès aux structures, à leurs champs, contrôle de cohérence des valeurs des champs, ...) devra être mise en place conjointement avec l'évolution des langages de programmation scientifique.

Par ailleurs, la "course" à la puissance de calcul passe par la multiplication des processeurs et des unités fonctionnelles des processeurs. Cependant, dans cette évolution, des améliorations plus locales des architectures sont les bienvenues. Toute avancée dans l'organisation des mémoires, des processeurs et dans leurs liaisons peut avoir son importance, notamment si elle est d'une faible complexité matérielle. C'est à ce niveau que nous plaçons le modèle d'exécution vectorielle désordonnée. La réalisation des opérations vectorielles sans ordre sur les indices des composantes des vecteurs semble très intéressante pour la suppression des conflits d'accès à la mémoire. Des études sur la validité de ce modèle sont en cours. Il est clair qu'il devra s'adapter au contexte des multi-processeurs vectoriels (type Cray avec 8, 16 processeurs ou plus). Les études concernant cette adaptation sont en cours.

Il semble qu'une nouvelle ère commence où il faudra définir des langages et des architectures mieux adaptés les uns aux autres, les deux devant mieux s'adapter à la problématique.

Références bibliographiques

Dans les références bibliographiques, les règles habituelles dictant le format des références ont été utilisées, hormis pour le titre des revues. Ainsi, nous avons utilisé les abbréviations :
CACM pour la revue *Communication of the Association of Computing Mechinary*,
IEEE Trans. on Comp. pour la revue *IEEE Transaction on Computers*,
IEEE Comp. pour la revue *IEEE Computer*.

[AHO & al. 89] Aho A., R. Sethi et J. Ullman, *Compilateurs - Principes, techniques et outils*, Inter-Editions (Paris : 1989)

[AHUJA & al. 86] Ahuja S., N. Carriero et D. Gelernter, "Linda and friends," *IEEE Comp.*, 19 (8), Août 1986, p. 26-34

[ALLEN & al. 82] Allen J. R. et K. Kennedy, "PFC : a program to convert Fortran to parallel form," *Proc. IBM Conf. on parallel computers and scientific computations*, Rome, Italie, 3-5 Mars 1982, in [HWANG 84b], p. 186-203

[ALLEN & al. 87] Allen R. et K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *ACM TOPLAS*, **9** (4), Oct. 1987, p. 491-542

[ANCONA & al. 89] Ancona M, A. Clematis et V. Gianuzzi, "32-bit Microprocessor Architectures and extended Abstract Machines for High Level Languages," in *Proc. EUROMICRO'89*, p. 349-354, 4-8 Sept. 1990, Cologne, RFA

[ANDERSON & al. 67] Anderson D. W., F. J. Sparacio et R. M. Tomasulo, "The IBM System/360 Model 91 : Machine Philosophy and Instruction-Handling," *IBM Jour. of Res. and Dev.*, **11** (1), p. 8-24, Janv. 1967

[ASTHANA & al. 84] Asthana A., H. V. Jagadish, J. A. Chandross, D. Lin et S. C. Knauer, "An Intelligent Memory System," *ACM Comp. Archi. News*, **16** (4) p. 12-20, Sept. 1988

[BAL & al. 86] Bal. H. E. et A. S. Tanenbaum, "Language- and machine-independent global optimization on intermediate code," *Computer Languages*, **11** (2), 1986, p. 105-121

[BHANDARKAR & al. 90]
Bhandarkar D. et R. Brunner, "VAX Vector Architecture," in *Proc. of the 17th Annual Int'l Symp. on Comp. Architecture*, 28-31 Mai 1990, Seattle, WA, USA, p. 204-215

[BARNES & al. 68] Barnes G. H., R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick et R. A. Stokes, "The ILLIAC IV computer," *IEEE Trans. on Comp.*, **C-17** (8), Août 1968, p. 746-757

[BATCHER 80] Batcher K. E., "Design of a massively parallel processor," *IEEE Trans. on Comp.*, **C-29**, (9), Sept. 1980, p. 836-840

[BETTEM & al. 87] Bettem J., M. Denneau et D. Weingarten, "The GF11 Parallel Computer," in [DONGARRA 87] p. 255-298

[BJØRNER & al. 80] Bjørner D. et O. N. Oest, "Towards a formal description of Ada," *Lecture Notes in Computer Science*, **98** (Springer-Verlag, 1980)

[BLELLOCH & al. 90a] Blelloch G. E., S. Chatterjee, F. Knabe, J. Sipelstein et M. Zagha, *VCODE Reference Manual (Version 1.1)*, Rapport CMU-CS-90-146, School of Computer Science, Université Carnegie Mellon, Pittsburgh, PA, USA, Juil. 1990

[BLELLOCH & al. 90b] Blelloch G. E. et S. Chatterjee, "VCODE : a data-parallel intermediate language," in *Proc. 3rd Symp. on the Frontiers of Massively Parallel Computation*, 8-10 Oct. 1990, Université du Maryland, College Park, MD, USA, p. 471-480

[BOSE 88] Bose P., "Interactive Program Improvement Via EAVE : an Expert Adviser for Vectorization," in *Proc. ICS'88*, p. 119-130, 4-8 Juil. 1988, St-Malo, France

[BRODE 81] Brode B., "Precompilation of Fortran Programs to Facilitate Array Processing," *IEEE Comp.*, **14** (9), Sept. 1981, p. 46-51

[BROSGOL 80] Brosgol B. M., "TCOLAda and the "middle end" of the PQCC Ada compiler," *ACM Sigplan Notices*, **15** (11), Nov. 1980, p. 101-112

[BUDNIK & al. 71] Budnick P. et D. J. Kuck, "The organization and use of parallel memories," *IEEE Trans. on Comp.*, **C-20** (12), Déc. 1971, p. 1566-1569

[CALLAHAN & al. 88] Callahan D., J. Dongarra et D. Levine, "Vectorizing compilers : a test suite ans results," *Rapport Argonne National Laboratory*, **ANL/MCS-TM-109**, Mars 1988 (repris sous une forme résumée dans *Proc. Supercomputing'88*, p. 98-105, 14-18 Nov. 1988, Orlando, Floride, USA)

[CDC 86] Control data Corp., *Fortran 200 Version 1 - Reference Manual* (Sunnyvale CA : Control Data Corp., 1986)

[CHEUNG & al. 86] Cheung T. et J. E. Smith, "A simulation study of the Cray X-MP memory system," *IEEE Trans. on Comp.*, C-35 (7), Jui 1986, p. 613-622

[CORDY & al. 90] Cordy J. R. et R. C. Holt, "Code generation using an orthogonal model," *Software - Practice and Experience*, **20** (3), Mars 1990, p. 301-320

[DEKEYSER & al. 89] Dekeyser J-L. et Ph. Preux, "Indirect Memory decoding for Vector Accesses," in *Proc. of the 1989 Int'l Symp. on Comp. Archi. and Digital Signal Processing*, p. 293-298, 11 - 14 Oct. 1989, Hong-Kong

[DEKEYSER & al. 90a] Dekeyser J-L., Ph. Marquet et Ph. Preux, "Vector Addressing Processor for direct and indirect Accesses," in *Proc. EUROMICRO'90*, p. 657-664, 27-30 Août 1990, Amsterdam, Pays-Bas

[DEKEYSER & al. 90b] Dekeyser J-L., Ph. Marquet et Ph. Preux, "EVA : an explicit Vector lAnguage," in *Proc. 10th Int'l Conf. in Comp. Science*, 23-27 Jui. 1990, Siantago, Chili

[DEKEYSER & al. 90c] Dekeyser J-L., Ph. Marquet et Ph. Preux, "EVA : an explicit Vector lAnguage. An Alternative Language to Fortran 90 (former 8x)," *ACM Sigplan Notices*, **25** (8), Août 1990, p. 53-71

[DEKEYSER & al. 90d] Dekeyser J-L., Ph. Marquet et Ph. Preux, "Devil : an intermediate vector language - definition and implementation," in *Proc. Workshop on Compilers for Parallel Computers*, 3-5 Déc. 1990, Paris, France

[DEKEYSER & al. 90e] Dekeyser J-L., T. Kechadi, Ph. Marquet et Ph. Preux, "Un modèle d'exécution vectorielle pipeline désordonnée," *Rapport Interne LIFL*, ERA-85, Oct. 1990 (en cours de referee)

[DIJKSTRA 72] Dijkstra E. W., "The Humble Programmer," *CACM*, **15** (10), Oct. 1972, p. 859-866

[DOMMERGAARD 80] Dommergaard O., "The design of a virtual machine for Ada," in [BJØRNER & al. 80], p. 435-605

[DONGARRA 87] Dongarra J. J., *Experimental parallel computing architectures*, Special topics in supercomputing (Amsterdam : North-Holland, 1987)

[DUBOIS & al. 87] Dubois R. et A. Boulesteix, *Le 68020 et ses coprocesseurs* (Paris : Editions PSI, 1987)

[EISENBEIS 86] Eisenbeis C., "Optimisation automatique de programmes sur « array-processors »," *Thèse de doctorat 3ème cycle*, soutenue le 30 juin 1986 à Paris VI.

[EISENBEIS & al. 90] Eisenbeis C., W. Jalby et A. Lichnewsky, "Compiler techniques for optimizing memory and register usage on the Cray-2," *Int'l Journal of High Speed Computing*, **2** (2), 1990, p. 193-222

[FALKOFF & al. 73] Falkoff A. D et K. E. Iverson, "The Design of APL," *IBM Journal of R & D*, **17** (4), Juil. 1973, p. 324-333

[FEILMEIER & al. 83] Feilmeier M., J. Joubert et U. Schendel (Eds.), *Proc. of the Int'l Conf. on Parallel Processing'83* (Amsterdam : North-Holland, 1984)

[FENG 81] Feng T. Y., "A Survey of Interconnection Networks," *IEEE Comp.*, **14** (12), Déc. 1981, p. 12-27

[FERNBACH 86] Fernbach S. (Eds), *Supercomputers - Class VI Systems, Hardware and Software* (Amsterdam : North-Holland 1986)

[FEUSTEL 73] Feustel E. A., "On the advantages of tagged architecture," *IEEE Trans. on Comp.*, **C-22** (7), Juil. 1973, p. 644-656

[FLYNN 72] Flynn M. J., "Some Computer Organization and Their Effectiveness," *IEEE Trans. on Comp.*, **C-21** (9), Sept. 1972, p. 948-960

[FRAILONG & al. 85] Frailong J. M., W. Jalby et J. Lenfant, "XOR-schemes : a flexible data organization in parallel memories," in *Proc. of the Int'l Conf. on Parallel Processing'85*, p. 276-283

[GABBER 89] Gabber E., "VMMP : a virtual machine for the development of portable and efficient programs for multiprocessors," in *Proc. Int'l Conf. on Parallel Processing 89*, **2**, p. 11-14

[GAJSKI & al. 83] Gajski D., D. Kuck, D. Lawrie et A. Sameh, "Cedar," *Rapport n° UIUCDCS-R-83-1123*, Department of Computer Science, University of Illinois, Urbana, Fév. 1983, in [HWANG 84b], p. 251-275

[GARDNER 77] Gardner P. J., "A transportation of Algol 68C," *ACM Sigplan Notices*, **12** (6), Juin 1977, p. 95-101

[GARRETT 90] Garrett P., "Abstract machines for scientific computation," *Supercomputer*, **7** (2), Mars 1990, p. 37-44

[GELERNTER & al. 90] Gelernter D. H., A. Nicolau et D. A. Padua (eds), *Languages and Compilers for Parallel Computing*, (Cambridge MA : MIT Press, 1990)

[GERHINGER & al. 85] Gerhinger E. F., A. K. Jones et Z. Z. Segall, "The Cm* testbed," *IEEE Comp.*, **18** (10), Oct. 1982, p. 40-53

[GILOI & al. 77] Giloi W. K. et H. Berg, "Introducing the concept of data structure architectures," in *Proc. Int'l Conf. on Parallel Processing 77*, 23-26 Août 1977, p. 44-51

[GISSELQUIST 86] Gisselquist R., "An experimental C Compiler for the Cray 2 Computer," *ACM Sigplan Notices*, **21** (9), Sept. 1986, p. 32-41

[GOTTLIEB 87] Gottlieb A., "An overview of the NYU ultracomputer project," in [DONGARRA 87], p. 25-96

[HADDON & al. 78] Haddon B. K. et W. M. Waite, "Experience with the universal intermediate language Janus," *Software-Practice and Experience*, 8 (5), Sept./Oct. 1978, p. 601-616

[HARPER & al. 87] Harper III D. T. et R. Jump, "Vector access performance in parallel memories using a skewed storage scheme," *IEEE Trans. on Comp.*, C-36 (12), Déc. 1987, p. 1440-1449

[HILLIS 88] Hillis D., *La machine à connexions*, (Paris : Masson, 1988)

[HOCKNEY & al. 88] Hockney R. W. et C. R. Jesshope, *Parallel Computers 2* (Adam Hilger Ltd, Bristol and Philadelphia, 1988)

[HWANG & al. 84a] Hwang K. et F. A. Briggs, *Computer Architecture and Parallel Processing*, Series in Computer Organization and Architecture (Mc GRAW-HILL 1984)

[HWANG 84b] Hwang K., *Tutorial on Supercomputers : design and application* (IEEE Computer Society Press, 1984)

[IBBETT & al. 89] Ibbett R. N., T. M. Hopkins et K. I. M. McKinnon, "Architectural Mechanisms to Support Sparse Vector Processing," in *Proc. of the 16th Ann. Int'l Symp. on Computer Architecture*, 28 mai - 1er Juin 1989, Jerusalem, Israël, ACM SIGARCH, 17 (3), p. 64-71

[IBSEN 84] Ibsen L., "A portable virtual machine for Ada," *Software - Practice and Experience*, 14 (1), Jan. 1984, p. 17-29

[JALBY 87] Jalby W., "Cedar architecture," in [LICHNEWSKY & al. 87], p. 41-51

[JEGOU 86] Jégou Y., "Le DSPA, un pipeline synchronisé par les données," *Rapport de Recherche INRIA* 574, Oct. 1986

[JEGOU 87] Jégou Y., "La langage vectoriel Hellena," *Rapport de recherche INRIA* 703, Jui. 1987

[JESSHOPE & al. 79] Jesshope C.R. et R. W. Hockney, *Infotech State of the Art Report : Supercomputers*, 2, C. R. Jesshope et R. W. Hockney Eds., (Maidenhead : Infotech Int. Ltd, 1979)

[KORNERUP & al. 80] Kornerup P. , B. B. Kristensen et O. L. Madsen, "Interpretation and code generation based on intermediate languages," *Software - Practice and Experience*, 10 (8), Août 1980, p. 635-658

[KUCK 68] Kuck D. J., "ILLIAC IV - Software and application programming," *IEEE Trans. on Comp.*, C-17 (8), Août 1968, p. 758-770

[KUCK 77] Kuck D. J., "A survey ofparallel machine organization and programming," *ACM Computing Surveys*, **9** (1), Mars 1977, p. 29-59

[KUCK & al. 82] Kuck D. J. et R. A. Stokes, "The Burroughs Scientific Processor (BSP)," *IEEE Trans. on Comp.*, **C-31** (5), Mai 1982, p. 363-375

[LAWRIE & al. 75] Lawrie D. H., T. Layman, D. Baer et J. M. Randal, "Glypnir - A Programming Language for Illiac IV," *CACM*, **18** (3), Mars 1975, p. 157-164

[LAWRIE & al. 82] Lawrie D. H. et C. R. Vora, "The Prime Memory System for Array Access," *IEEE Trans. on Comp.*, **C-31** (5), Mai 1982, p. 435-442

[LAZOU 86] Lazou C., *Supercomputers and their use* (Oxford Science Publications, 1986)

[LI & al. 84] Li K.-C. et H. Schwetman, "Implementing a scalar C compiler on the Cyber-205," *Software - Practice and Experimence*, **14** (9), Sept. 1984, p. 867-888

[LICHNEWSKY & al. 85] Lichnewsky A. et F. Thomasset, "Techniques de base pour l'exploitation automatique du parallélisme dans les programmes," *Rapport de Recherche INRIA 460*, Déc. 1985

[LICHNEWSKY & al. 87] Lichnewsky A. et C. Saguez, *Supercomputing*, (Amsterdam : North-Holland, 1987)

[LOUDEN 90] Louden K., "P-Code and compiler portability : experience with a Modula-2 optimizing compiler," *ACM Sigplan Notices*, **25** (5), Mai 1990, p. 53-59

[LUBECK 88] Lubeck O. M., "Supercomputer performance : the theory, practice and results," *Rapport LA-11204-MS*, Laboratoire Scientifique de Los Alamos, Nouveau-Mexique, USA, Janv. 1988

[MADHAVJI & al. 82] Madhavji N. H. et I. R. Wilson, "Cray Pascal," in *Proc. of the Sigplan'82 Symp. on Compiler Construction*, *ACM Sigplan Notices*, **17** (6), Juin 1982, p. 1-14

[MARQUET 90] Marquet Ph., "The EVA Language Grammar," *Note Technique LIFL*, **ERA-82**, Mai 1990

[METCALF & al. 90] Metcalf M. et J. Reid, *Fortran 8x explained - Revised edition*, (Oxford : Oxford Science Publications, 1989)

[MIURA 86] Miura K., "Fujitsu's supercomputer : FACOM vector processor system," in [FERNBACH 86], p. 137-151

[NAGEL 90] Nagel W. E., "Exploiting autotasking on a CRAY Y-MP : an improved software interface to multitasking," *Parallel Computing*, **13** (2), Fév. 1990, p. 225-233

[NELSON 79] Nelson P. A., "A comparison of Pascal intermediate languages," *ACM Sigplan Notices*, **14** (8), Août 1979, p. 208–213

[NEWEY & al. 72] Newey M. C., P. C. Poole et W. M. Waite, "Abstract Machine Modelling to Produce Portable Software – A Review and Evaluation," *Software-Practice and Experience*, **2** (2) Avril/Juin 1972, p. 107–136

[ORGASS & al. 69] Orgass R. J. et W. M. Waite, "A base for a mobile programming system," *CACM*, **12** (9), Sept. 1969, p. 507–510

[PADEGS & al. 88] Padegs A., B. B. Moore, R. M. Smith et W. Buchholz, "The IBM System/370 vector architecture : design considerations," *IEEE Trans. on Comp.*, **C-37** (5), Mai 1988, p. 509–520

[PADUA & al. 86] Padua D. A. et M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *CACM*, **29** (12), Déc. 1986, p. 1184–1201

[PAUL & al. 78] Paul G. et M. W. Wilson, "An introduction to VECTRAN and its use in scientific applications programming," in *Proc. of the 1978 LASL Workshop on Vector and Parallel Processors*, 20–22 Sept. 1978, Los Alamos, Nouveau-Mexique, USA, p. 176–204

[PERROTT 79] Perrott R. H., "A Language for Array and Vector Processors," *ACM TOPLAS*, **1** (2), Oct. 1979, p. 177–195

[PERROTT & al. 81] Perrott R. H. et D. K. Stevenson, "Considerations for the design of array processing languages," *Software – Practice and Experience*, **11** (7), Juil. 1981, p. 683–688

[PERROTT & al. 83] Perrott R. H., D. Crookes, P. Milligan et W. R. M. Purdy, "A Compiler for the synchronous parallel programming language Actus," in [FEILMEIER & al. 83], p. 475–480

[PERROTT & al. 85] Perrott R. H., D. Crookes, P. Milligan et W. R. M. Purdy, "A Compiler for an array and vector processing language," *IEEE Trans. on Software Engineering*, **SE-11** (5), Mai 1985, p. 471–478

[PFISTER & al. 87] Pfister G. F., W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. Mc Auliffe, E. A. Melton, V. A. Norton et J. Weiss. "An introduction to the IBM Research Parallel Processor Prototype (RP3)," in [DONGARRA 87], p. 123–140

[POLYCHRONOPOULOS & al. 90] Polychronopoulos C. D., M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung et D. A. Schouten, "The structure of Parafrase-2 : an advanced parallelizing compiler for C and Fortran," in [GELERNTER & al. 90], p. 423–453

[PREUX 88] Preux P., "Etude et conception d'un outil d'intercommunications pour la machine VECTRAN," *mémoire de DEA*, 1988

[PREUX 90] Preux P., "Présentation de la machine virtuelle MAD et de son langage Devil," *Publication interne LIFL* n° ERA-81, Mai 1990

[RAGHAVAN & al. 90] Raghavan R. et J. P. Hayes, "On randomly interleaved memories," in *Proc. Supercomputing'90*, 12-16 Nov. 1990, New York, NY, USA, p. 49-58

[RAMAMOORTHY & al. 77] Ramamoorthy C. V. et H. F. Li, "Pipeline Architecture," *ACM Computing Surveys*, **9** (1), p. 38-79, Mars 1977

[RAU & al. 89] Rau B. R., M. S. Schlansker et D. W. L. Yen, "The Cydra 5 stride-insensitive memory system," in *Proc. Int'l Conf. on Parallel Processing 89*, p. I-242, I-246

[REINHARDT 85] Reinhardt S., "A data-flow approach to multitasking on Cray X-MP computers," *ACM Operating Systems Review*, **19** (5), 1985, p. 107-114

[RICHARDS 71] Richards M., "The portability of the BCPL compiler," *Software - Practice and Experience*, **1** (2), Avril 1971, p. 135-146

[RUSSEL 78] Russell R. M., "The Cray-1 computer system," *CACM*, **21** (1), Janv. 1978, p. 63-72

[SCHONAUER 87] Schönauer W., "The Cray-2," in *Scientific Computing on Vector Computers*, (Amsterdam : North-Holland, 1987), p. 305-324

[SHAPIRO 78] Shapiro H. D., "Theoretical limitations on the efficient use of parallel memories," *IEEE Trans. on Comp.*, **C-27** (5), Mai 1978, p. 421-428

[SIBLEY 61] R. A. Sibley, "The SLANG system," *CACM*, **4** (1), Jan. 1961, p. 75-84

[SIEGEL 79] Siegel H. J., "A Model of SIMD Machines and a Comparison of Various Interconnection Networks," *IEEE Trans. on Comp.*, **C-28** (12), Déc. 1979, p. 907-917

[SIMMONS & al. 90] Simmons M. L. et H. J. Wasserman, "Performance comparison of the Cray-2 and Cray X-MP/416 supercomputers," *The Journal of Supercomputing*, **4**, 1990, p. 153-167

[STEVENS 75] Stevens K., "CFD - A Fortran-like Language for Illiac-IV," *ACM Sigplan Notices*, **10** (3), Mars 1975, p. 72-80

[STONE 87] Stone H. S., *High-performance computer architecture*, (Addison-Wesley, 1987)

[STRONG & al. 58] Strong J., J. Wegstein, A. Tritter, J. Olsztyn, O. Mock et T. Steel, "The problem of programming communication with changing machines," *CACM*, **1** (8-9), Août/Sept. 1958, (8) p. 12-18 et (9) p. 9-16

[TANENBAUM & al. 82] Tanenbaum A. S., H. Van Staveren et J. W. Stevenson, "Using peephole optimization on intermediate code," *ACM TOPLAS*, **4** (1), Jan. 1982. p. 21-36

[TANENBAUM & al. 83] Tanenbaum A. S., H. Van Staveren, E. G. Keizer et J. W. Stevenson, "A practical tool kit for making portable compilers," *CACM*, **26** (9), Sept. 1983, p. 654-660

[TANG & al. 89] Tang P. et R. H. Mendez, "Memory conflicts and machine performance," in *Proc. of Supercomputing'89*, 13-17 Nov. 1989, Reno, Nevada, USA, p. 826-831

[THOMPSON 86] Thompson J. R., "The Cray-1, the Cray X-MP, the Cray-2 and beyond : the supercomputers of Cray Research," in [FERNBACH 86], p. 69-81

[TUCKER 86] Tucker S. G., "The IBM 3090 system : an overview," *IBM J. of R. & D.*, **25** (1), Janv. 1986, p. 4-19

[TUCKER & al. 88] Tucker L. W. et G. G. Robertson, "Architecture and applications of the Connection Machine," *IEEE Comp.*, **21** (8), Août 1988, p. 26-38

[VOLKSEN & al. 89] Völksen G et P. Wehrum, "Ada for Scientific Computation on Vector Processors," *Journal of Pascal, Ada & Modula-2*, **8** (6), Nov./Déc. 1989, p. 16-32

[WATANABE & al. 86] Watanabe T., H. Katayama et A. Iwaya, "Introduction of NEC supercomputer SX-system," in [FERNBACH 86], p. 153-167

[WAITE 70a] Waite W. M., "Building a mobile programming system," *The Computer Journal*, **13** (1), Fév. 1970, p. 28-31

[WAITE 70b] Waite W. M., "The mobile programming system : STAGE 2," *CACM*, **13** (7), Juil. 1970, p.415-421

[WEISS & al. 84] Weiss S. et J. E. Smith, "Instruction Issue Logic in Pipelined Supercomputers," *IEEE Trans. on Comp.*, C-33 (11), Nov. 1984, p. 1013-1022

[WEISS 89] Weiss S., "An aperiodic scheme to reduce memory conflicts in vector processors," in *Proc. of the 16th Annual Int'l Symp. on Computer Architecture*, 28 Mai, 1er Juin 1989, Jerusalem, Israël, p. 380-386

[WETHERELL 80] Wetherell C., "Design considerations for array processing languages," *Software - Practice and Experience*, **10** (4), Avril 1980, p. 265-271

[WILK & al. 83] Wilk A. et W. Silverman, "OPTIMA - A portable P-code optimizer." *Software - Practice and Experience*, **13** (4), Avril 1983, p. 323-354

[WORLTON 81] Worlton J., "A philosophy of supercomputing," *Rapport LA-8849-MS*, Laboratoire Scientifique de Los Alamos, Nouveau-Mexique, USA, Juin 1981

[WULF 81] Wulf W. A., "Compilers and Computer Architecture," *IEEE Comp.*, **14**, 7, Juil. 1981, p. 41–47

[YAU & al. 77] Yau S. S. et H. S. Fung, "Associative processor architecture – a survey," *ACM Computing Surveys*, **9** (1), Mars 1977, p. 3–27

[ZAKHAROV 84] V. Zakharov, "Parallelism and array processing," *IEEE Trans. on Comp.*, C-33 (1), Jan. 1984, p. 45–78

Bibliographie

Il serait difficile de citer ici tous les articles et livres qui m'ont aidé dans la rédaction de ce document. Aussi, j'indiquerai simplement ici quelques références aux manuels de constructeurs qui m'ont permis d'obtenir des informations sur les architectures et langages machines des supercalculateurs discutés ici.

[CRAY 87] Cray Research Inc., *Cray X-MP Computer Systems Functional Description Manual*, Pub. HR-3005, Mendota Heights, MN 55120, USA, Fév. 1987

[CRAY 88a] Cray Research Inc., *Cray Y-MP Computer Systems Functional Description Manual*, Pub. HR-4001, Mendota Heights, MN 55120, USA, Jan. 1988

[CRAY 88b] Cray Research Inc., *Symbolic Machine Instructions - Reference Manual*, Pub. SR-0085 B, Mendota Heights, MN 55120, USA, Mars 1988

[CRAY 88c] Cray Research Inc., *CAL Assembler Version 2 Reference Manual*, Pub. SR-2003 C, Mendota Heights, MN 55120, USA, Juin 1988

[CDC 87] Control Data Corp., *CDC Cyber 200 Model 205 Computer System*, Pub. 60256020, Oct. 1987

[IBM 88] IBM Corp., *IBM Enterprise Systems Architecture/370 and System/370 - Vector Operations*, Pub. SA22-7125-3, Août 1988

[NEC 89] Nec Corp., *NEC Supercomputer SX-3 series - General Description*, Pub. GAZ02E-1, Août 1989

[SIEMENS 88a] Siemens, *Vector Processor System VP Series - Hardware Principles of Operation*, Pub. U2006-J-Z69-3-7600, Mars 1988

[SIEMENS 88b] Siemens, *Vector Processor System VP-EX Series - Hardware Principles of Operation*, Pub. U4012-J-Z69-1-7600, Jui. 1988

Annexe
Les instructions de Devil

Dans cette annexe, nous donnons la liste des instructions du langage Devil. Pour chaque instruction, nous indiquons les opérandes et décrivons brièvement sa fonction. Les instructions se regroupées par "classe de fonctionnalités". Elles ne sont pas présentées par famille, une même instruction pouvant appartenir à plusieurs familles en même temps, selon le type de ses opérandes.

Les pseudo-instructions

```
n.label     nom
```

Déclaration d'une étiquette de nom *nom*.

```
n.const     cst, #10
```

Déclaration d'une constante de nom *cst*, de valeur 10 (immédiate forcément).

```
n.extern    nom_de_symbole
```

Déclaration d'un symbole externe (importation du nom) de nom *nom_de_symbole*.

```
n.common    my_common, #100
```

Déclaration de zone common de nom *my_common* et de taille 100 mots mémoire (valeur immédiate forcément).

```
n.debistat
```

Début de zone d'initialisation des données statiques au module.

```
n.finistat
```

Fin de zone d'initialisation des données statiques au module.

```
n.function  nom_fct, #10, #20
```

Déclaration d'une fonction visible de l'extérieur du module (exportée) de nom *nom_fct*, ayant un segment *local* et un segment *param_to_send* de tailles 10 et 20 respectivement.

```
n.fctl      nom_fct_locale, #10, #20
```

Déclaration d'une fonction invisible de l'extérieur du module de nom *nom_fct_locale*, ayant un segment *local* et un segment *param_to_send* de tailles 10 et 20 respectivement.

```
n.endfct
```

Fin de la fonction courante.

```
n.forward    en_avant
```

Déclaration en avant d'un symbole de nom *en_avant*.

```
n.reltmp     & 10
```

Spécifie que le temporaire *&10* ne sera plus utilisé dans la suite du programme.

Les instructions de contrôle de flot

```
n.nop
```

Instruction nulle (si la cible dispose d'une instruction équivalente, `n.nop` est traduite par cette instruction).

```
n.wait
```

Instruction de contrôle du répartiteur d'instructions. Aucune nouvelle instruction n'est déclenchée avant que toutes instructions en cours n'aient terminé leur exécution.

```
n.branch     valeur, étiquette
```

Instruction de branchement conditionnel si la valeur de l'*objet* est non nulle à l'*étiquette*.

```
n.fbranch    valeur, étiquette
```

Instruction de branchement conditionnel si la valeur de l'*objet* est nulle à l'*étiquette*.

```
n.jmp        étiquette
```

Instruction de saut inconditionnel à l'*étiquette*.

Les instructions de gestion des appels de fonctions

```
  call       nom_de_fct

n.call       nom_de_fct

  call       nom_de_fct, valeur_de_retour
```

Appel de la fonction de nom *nom_de_fct*. La valeur de l'objet renvoyé par la fonction est affectée à l'objet *valeur_de_retour* si cet opérande est spécifié.

```
n.return

  return     objet
```

Retour de fonction avec renvoi de la valeur de l'*objet* s'il est spécifié.

```
n.parout     objet, offset
```

Passe la valeur de l'*objet* en paramètre, en mode lecture/écriture.

```
n.parin     objet, offset
```

Passe la valeur de l'*objet* en paramètre, en mode lecture.

```
n.parconst  objet, offset
```

Passe la valeur de l'*objet* en paramètre, en mode lecture de l'en-tête/écriture de la zone d'allocation.

Les instructions de gestion du registre VLG

```
  getvlg
n.getvlg    cible
  getvlg    cible
```

Produit la valeur courante du registre VLG.

```
n.push      objet
```

Si l'opérande est un vecteur, le registre VLG prend la valeur de son nombre d'éléments. Si l'opérande est un scalaire, sa valeur est affectée à VLG. L'ancienne valeur est sauvegardée dans le segment *vlg*.

```
n.popvlg
```

Restaure l'ancienne valeur de VLG depuis le segment *vlg*.

Les instructions vectorielles de construction

```
n.alloc     taille, vecteur
```

Allocation d'une zone d'allocation de *taille* spécifiée au *vecteur*.

```
n.assoc     source, cible
```

Partage de la zone d'allocation entre le vecteur *source* et le vecteur *cible*.

```
n.dassoc    source, descripteur, cible
```

Partage de la zone d'allocation entre le vecteur *source* et le vecteur *cible* et association du *descripteur* à la *cible*.

```
  sinit     début, fin, pas
```

Création d'un vecteur temporaire triplet de borne inférieure *début*, de borne supérieure *fin* et de *pas* spécifiés.

```
  sinitvlg  début, pas
```

Création d'un vecteur temporaire triplet de borne inférieure *début*, de *pas* spécifié et comportant VLG éléments.

```
n.free      vecteur
```

Désallocation de la zone des description du *vecteur* et dé-référenciation des zones d'allocation et d'informations de ce même *vecteur*.

Les instructions vectorielles de manipulation

```
  gath      source, descripteur
```

```
n.gath     source, descripteur, cible
  gath     source, descripteur, cible
```

Réalise l'opération de rassemblement *cible* = *source* [*descripteur*].

```
n.scat     source, descripteur, cible
  scat     source, descripteur, cible
```

Réalise l'opération d'éclatement *cible* [*descripteur*] = *source*.

```
  merge    source 1, source 2, descripteur
n.merge    source 1, source 2, vecteur de contrôle, cible
  merge    source 1, source 2, vecteur de contrôle, cible
```

Réalise un merge.

```
  mask     objet 1, objet 2, vecteur de contrôle
n.mask     objet 1, objet 2, vecteur de contrôle, cible
  mask     objet 1, objet 2, vecteur de contrôle, cible
```

Réalise une affectation masquée par le *vecteur de contrôle*, des sources *objet 1* et *objet 2* dans la *cible*.

Les instructions arithmétiques vectorielles et scalaires

Formes des instructions add, sub, mul, div, rem, or, xor, and :

```
  instr    source 1, source 2
n.instr    source 1, source 2, cible
  instr    source 1, source 2, cible
```

add réalise l'addition des deux sources et stocke le résultat dans la *cible*.

sub réalise la soustraction des deux sources et stocke le résultat dans la *cible*.

mul réalise le produit des deux sources et stocke le résultat dans la *cible*.

div réalise le rapport des deux sources et stocke le résultat dans la *cible*.

rem réalise le calcul du reste deux sources et stocke le résultat dans la *cible*.

or réalise le ou bit à bit des deux sources et stocke le résultat dans la *cible*.

xor réalise le ou-exclusif bit à bit des deux sources et stocke le résultat dans la *cible*.

and réalise le et bit à bit des deux sources et stocke le résultat dans la *cible*.

```
  opp      source
n.opp      source, cible
  opp      source, cible
```

Calcule l'opposé arithmétique de la *source* et la stocke dans la *cible*.

```
  not      source
n.not      source, cible
  not      source, cible
```

Calcule la négation bit à bit de la *source* et stocke la résultat dans la *cible*.

Les instructions de comparaison vectorielles et scalaires

Formes des instructions gt, lt, eq, ne, ge, le :

```
  instr    source 1, source 2
n.instr    source 1, source 2, cible
  instr    source 1, source 2, cible
```

gt réalise l'opération *cible = source 1 >= source 2*.

lt réalise l'opération *cible = source 1 > source 2*.

eq réalise l'opération *cible = source 1 == source 2*.

ne réalise l'opération *cible = source 1 != source 2*.

ge réalise l'opération *cible = source 1 > source 2*.

le réalise l'opération *cible = source 1 <= source 2*.

Les instructions d'affectation

```
n.vinit    vecteur
```

Initialise les champs de l'en-tête du *vecteur*.

Instruction d'affectation simple

```
  move     source
n.move     source, cible
  move     source, cible
```

Affecte la valeur de la *source* dans la *cible*.

Instruction d'affectation conditionnelle

```
  select   valeur, valeur si vrai, valeur si faux
n.select   valeur, valeur si vrai, valeur si faux, cible
  select   valeur, valeur si vrai, valeur si faux, cible
```

Réalise une affectation conditionnelle selon la *valeur* (vraie ou fausse) d'une des deux valeurs dans la *cible*.

Instructions d'initialisation de vecteurs à partir de constantes

```
  hexa     constante
```

```
n.hexa    constante, cible
  hexa    constante, cible
```

Convertit une *constante* hexadécimale en vecteur de bits *cible*

```
 octal    constante
n.octal   constante, cible
 octal    constante, cible
```

Convertit une *constante* octale vecteur de bits *cible*

```
 bit      constante
n.bit     constante, cible
 bit      constante, cible
```

Convertit une *constante* binaire en vecteur de bits *cible*

```
 string   chaîne de caractères
n.string  chaîne de caractères, cible
 string   chaîne de caractères, cible
```

Convertit une *chaîne de caractères* en vecteur d'entiers *cible*.

Instructions de création de vecteurs d'index multidimensionnels

```
dim       src 1, src 2, borne inf. 1, borne sup. 1, borne inf. 2
diml      src 1, src 2, borne sup. 1, borne inf. 2
```

Calcule un vecteur d'index pour la sélection d'éléments dans des objets représentant des variétés à n dimensions.

Les instructions de concaténation

```
 cat      s 1, s 2, ... s i, ... s n
n.cat     s 1, s 2, ... s i, ... s n, cible
 cat      s 1, s 2, ... s i, ... s n, cible
```

Concatène les éléments des sources *s i* et place le résultat dans la *cible*.

```
 catvlg   s 1, s 2, ... s i, ... s n
n.catvlg  s 1, s 2, ... s i, ... s n, cible
 catvlg   s 1, s 2, ... s i, ... s n, cible
```

Concatène les éléments des sources *s i* jusqu'à atteindre VLG éléments et place le résultat dans la *cible*.

```
 lcat     s 1, s 2, ... s i, ... s n
n.lcat    s 1, s 2, ... s i, ... s n, cible
 lcat     s 1, s 2, ... s i, ... s n, cible
```

Concatène les valeurs littérales *l i* et les rangent dans la *cible*.

```
  lcatvlg   s 1, s 2, ... s i, ... s n
n.lcatvlg   s 1, s 2, ... s i, ... s n, cible
  lcatvlg   s 1, s 2, ... s i, ... s n, cible
```

Concatène les VLG premières valeurs littérales *l i* et les rangent dans la *cible*.

```
  ncat      n, source
n.ncat      n, source, cible
  ncat      n, source, cible
```

Concatène *n* fois les éléments de la *source* et les rangent dans la *cible*.

```
  ncatvlg   n, source
n.ncatvlg   n, source, cible
  ncatvlg   n, source, cible
```

Concatène *n* fois les éléments de la *source* et les rangent les VLG premiers éléments dans la *cible*.

Les instructions de conversion vectorielles et scalaires

Formes des instructions btoi, itob, itor, rtoi, rtod, dtor, itod, dtoi :

```
  instr     source
n.instr     source, cible
  instr     source, cible
```

btoi convertit une *source* binaire en entier.

itob convertit une *source* entière en binaire.

itor convertit une *source* entière en flottant simple précision.

rtoi convertit une *source* flottant simple précision en entier.

rtod convertit une *source* flottant simple précision en flottant double précision.

dtor convertit une *source* flottant double précision en flottant simple précision.

itod convertit une *source* entière en flottant double précision.

dtoi convertit une *source* flottant double précision en entier.

```
  bitoind   source
n.bitoind   source, cible
  bitoind   source, cible
```

Transforme un vecteur de bits *source* en une *cible* vecteur d'entiers.

```
  indtobit  source
n.indtobit  source, cible
  indtobit  source, cible
```

Transforme un vecteur d'entiers *source* en une *cible* vecteur de bits.

Les instructions de réduction

Formes des instructions sigma, pi, valmax, valmin, indmax, indmin :

```
  instr    source
n.instr    source, cible
  instr    source, cible
```

sigma calcule la somme des éléments d'un vecteur *source*.

pi calcule le produit des éléments d'un vecteur *source*.

valmax recherche la valeur maximale parmi les éléments d'un vecteur *source*.

valmin recherche la valeur minimale parmi les éléments d'un vecteur *source*.

indmax recherche l'indice de la valeur maximale parmi les éléments d'un vecteur *source*.

indmin recherche l'indice de la valeur minimale parmi les éléments d'un vecteur *source*.

Les instructions de calculs scientifiques vectorielles et scalaires

Instructions binaires

Formes des instructions sin, cos, tg, arcsin, arccos, arctg, rec, sqr, sqrt, log10, exp, ln, abs, trunc, ceil :

```
  instr    source
n.instr    source, cible
  instr    source, cible
```

sin calcule le sinus de la *source*.

cos calcule le cosinus de la *source*.

tg calcule la tangente de la *source*.

arcsin calcule l'arc-sinus de la *source*.

arccos calcule l'arc-cosinus de la *source*.

arctg calcule l'arc-tangente de la *source*.

rec calcule l'inverse de la *source*.

sqr calcule le carré de la *source*.

sqrt calcule la racine carrée sinus de la *source*.

log10 calcule le logarithme (base 10) de la *source*.

exp calcule l'exponentielle (base e) de la *source*.

`ln` calcule le logarithme (base e) de la *source*.

`abs` calcule la valeur absolue de la *source*.

`trunc` calcule le de la *source*.

`ceil` calcule le de la *source*.

Instructions ternaires

Formes des instructions pow, max, min :

```
  instr    source 1, source 2
n.instr    source 1, source 2, cible
  instr    source 1, source 2, cible
```

pow calcule l'exponentiation de la *source 1* par la *source 2*.

max produit le maximum entre les deux sources.

min produit le minimum d'entre les deux sources.

Les instructions vectorielles à résultat scalaire

```
  all0     vecteur
n.all0     vecteur, cible
  all0     vecteur, cible
```

produit VRAI si tous les éléments du (bit-)*vecteur* sont 0, FAUX sinon

```
  all1     vecteur
n.all1     vecteur, cible
  all1     vecteur, cible
```

produit VRAI si tous les éléments du (bit-)*vecteur* sont 1, FAUX sinon

```
  nbuns    vecteur
n.nbuns    vecteur, cible
  nbuns    vecteur, cible
```

produit le nombre d'éléments à 1 dans le (bit-)*vecteur*.

```
  nbz      vecteur
n.nbz      vecteur, cible
  nbz      vecteur, cible
```

produit le nombre d'éléments à 0 dans le (bit-)*vecteur*.

```
  inpun    vecteur
n.inpun    vecteur, cible
  inpun    vecteur, cible
```

produit l'indice de la composante à 1 d'indice le plus petit dans le (bit-)*vecteur*.

```
  inpz     vecteur
n.inpz     vecteur, cible
  inpz     vecteur, cible
```

produit l'indice de la composante à 0 d'indice le plus petit dans le (bit-)*vecteur*.

```
  inlun    vecteur
n.inlun    vecteur, cible
  inlun    vecteur, cible
```

produit l'indice de la composante à 0 d'indice le plus grand dans le (bit-)*vecteur*.

```
  inlz     vecteur
n.inlz     vecteur, cible
  inlz     vecteur, cible
```

produit l'indice de la composante à 0 d'indice le plus grand dans le (bit-)*vecteur*.

Les instructions d'accès aux attributs vectoriels

```
  nbelts   vecteur
n.nbelts   vecteur, cible
  nbelts   vecteur, cible
```

produit le nombre d'éléments du *vecteur*.

```
  isdesced vecteur
n.isdesced vecteur, cible
  isdesced vecteur, cible
```

produit VRAI si le *vecteur* est décrit, FAUX sinon.

```
  isbited  vecteur
n.isbited  vecteur, cible
  isbitced vecteur, cible
```

produit VRAI si le *vecteur* est décrit par une séquence de bits, FAUX sinon.

```
  isindexed vecteur
n.isindexed vecteur, cible
  isindexed vecteur, cible
```

produit VRAI si le *vecteur* est décrit par une séquence d'entiers, FAUX sinon.

```
  zasize   vecteur
n.zasize   vecteur, cible
```

```
    zasize    vecteur, cible
```

produit le nombre d'éléments de la zone d'allocation du *vecteur* (valeur du champ *tza* de son en-tête)

```
    zdsize    vecteur
  n.zdsize    vecteur, cible
    zdsize    vecteur, cible
```

produit le nombre d'éléments de la zone de description du *vecteur* (valeur du champ *tzd* de son en-tête)

```
    za        vecteur
  n.za        vecteur, cible
    za        vecteur, cible
```

produit un vecteur composé des éléments de la zone d'allocation du *vecteur*.

```
    zdbit     vecteur
  n.zdbit     vecteur, cible
    zdbit     vecteur, cible
```

produit un vecteur composé des éléments de la zone de description, considérés comme des bits, du *vecteur*.

```
    zdindex   vecteur
  n.zdindex   vecteur, cible
    zdindex   vecteur, cible
```

produit un vecteur composé des éléments de la zone de description, considérés comme des entiers, du *vecteur*.

```
  n.setza     adresse za, taille za, adresse zi, cible
```

Range les informations *adresse za*, *taille za* et *adresse zi* dans les champs de l'en-tête du vecteur *cible*.

« Nous sentons que lors même toutes les questions scientifiques possibles on été résolues, notre problème n'est pas encore abordé. »

L. Wittgenstein, *Carnets*, 1915