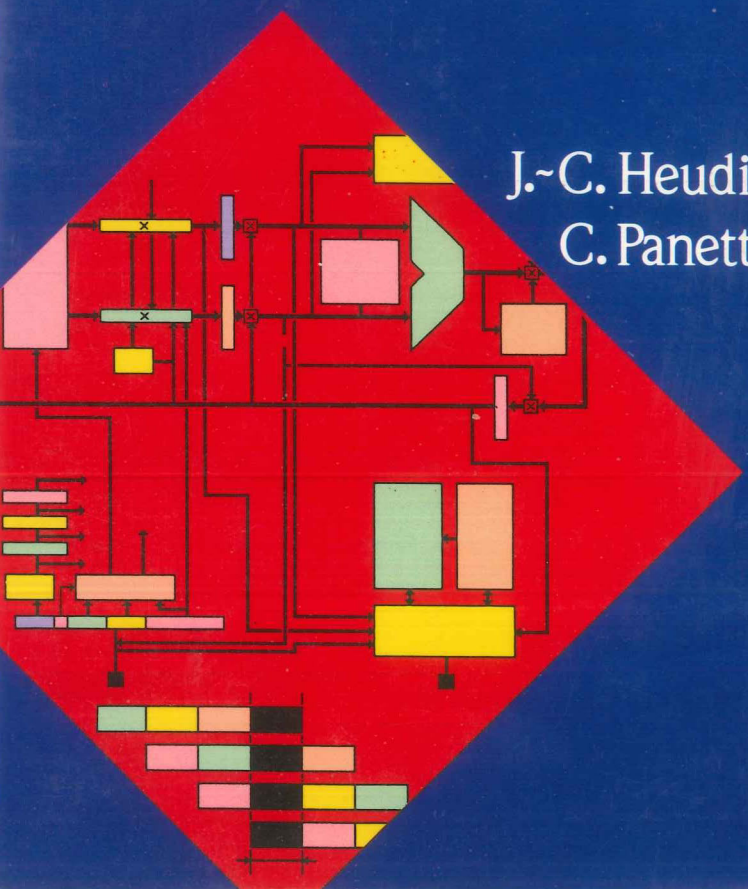
Les architectures RISC

Théorie et pratique des ordinateurs à jeu d'instructions réduit



J.-C. Heudin
C. Panetto

DUNOD
informatique

Les architectures RISC

Les architectures RISC

JEAN-CLAUDE HEUDIN

CHRISTIAN PANETTO

DUNOD
informatique

Présentation des auteurs

Jean-Claude HEUDIN est titulaire d'un doctorat de l'Université d'Orsay (Paris-Sud XI) en sciences physiques. Initiateur du projet KIM, il est, en outre, directeur technique et co-fondateur de la Société SODIMA S.A. Il est l'auteur de nombreuses publications internationales dans le domaine de l'Intelligence Artificielle et de l'architecture des systèmes informatiques. En marge de ses activités de recherche et industrielles, il participe à la formation d'étudiants 3ème cycle en architectures microélectroniques à l'Institut d'Electronique Fondamentale de l'Université Paris-Sud.

Christian PANETTO est professeur agrégé de génie électrique. Il enseigne l'électronique et l'informatique industrielle dans les classes post-baccalauréat. Il participe également à la formation d'enseignants en sciences et techniques industrielles ainsi qu'à celle de techniciens et ingénieurs dans le cadre de la formation continue. Les relations avec l'industrie lui ont permis de réaliser avec elle de nombreux projets. Il est l'auteur de plusieurs articles et ouvrages sur les systèmes logiques et l'informatique industrielle, dont le "Guide pratique des systèmes logiques" aux éditions E.T.S.F. et "Etudes de cas dans un environnement de productions" au éditions DUNOD.

ISBN : 2-04-019641-2

Sommaire

Introduction

En 1980, le sigle RISC apparaissait pour la première fois dans le cours du professeur D.A. Patterson à l'Université de Berkeley en Californie. Depuis, nous avons vu le terme "*Reduced Instruction Set Computer*" (ordinateur à jeu d'instructions réduit) envahir les colonnes de la presse spécialisée, retraçant l'engagement progressif des grands constructeurs informatiques dans cette voie.

Alors que l'histoire de la technologie informatique nous avait habitué à une augmentation croissante de la complexité des systèmes, cette nouvelle philosophie de conception semble radicalement inverser la tendance, prônant la simplicité pour l'efficacité. De fait, les performances mesurées sur ces nouveaux ordinateurs confirment les mérites de l'architecture RISC. Pourtant, beaucoup de techniciens éprouvent encore une certaine difficulté à isoler les véritables nouveautés qui justifient son avènement. Que cachent donc réellement ces quatre lettres qui ont révolutionné, en l'espace de quelques mois, le paysage informatique mondial ?

La réponse à cette question est le premier objectif de cet ouvrage. Ecrit à la fois pour le néophyte qui désire s'initier à cette nouvelle technologie et pour le spécialiste qui veut en savoir plus, ce livre aborde la théorie et la pratique des ordinateurs à jeu d'instructions réduit. Les lecteurs qui voudront aller plus avant et maîtriser parfaitement la méthodologie RISC, trouveront à la fin de l'ouvrage les références des principaux articles qui ont marqué le développement du concept RISC (indiquées dans le texte entre crochets [ ]).

Ce livre est composé de quatre chapitres : les trois premiers abordent l'histoire et les fondements théoriques de l'architecture RISC ; le quatrième donne ensuite, au travers de l'étude du processeur KIM™ (*Knowledge-based Integrated Machine* - Machine intégrée fondée sur la connaissance), une approche plus concrète des aspects matériels et logiciels sous-jacents.

A partir des précurseurs et des universités qui ont participé à son développement, le premier chapitre retrace les grandes étapes de l'avènement du concept RISC. Un tour d'horizon de la recherche mondiale est ensuite effectué qui met en évidence la richesse du sujet et ses retombées industrielles. Le chapitre conclut sur les prolongements possibles des travaux dans ce domaine.

Le second chapitre expose les principes fondamentaux qui régissent la conception d'un processeur RISC. Ceux-ci sont comparés aux approches plus traditionnelles pour mettre en évidence la nouveauté et la logique de la méthodologie.

Le troisième chapitre décrit succinctement les principaux processeurs commercialisés. Il présente également un récapitulatif des configurations étudiées, ainsi qu'un comparatif détaillé des caractéristiques architecturales, des performances et des domaines d'applications.

Le dernier chapitre donne une description détaillée du processeur KIM. Celui-ci est caractérisé par une architecture RISC spécifiquement conçue pour l'exécution efficace des programmes issus des travaux en Intelligence Artificielle. Au travers de son analyse et d'exemples pratiques, l'architecture KIM illustre les principes exposés lors des précédents chapitres. Elle met également en évidence les liens étroits nécessaires entre l'architecture d'un processeur et le logiciel qu'il doit exécuter. Des exemples illustrent les relations entre le jeu d'instructions du processeur KIM et le traitement symbolique, sous la forme d'exemples en Lisp.

Cet ouvrage n'a pas la prétention d'être exhaustif sur l'ensemble des techniques sous-jacentes à la conception ou à l'utilisation des processeurs RISC. Il représente néanmoins une synthèse assez complète sur "l'état de l'art" en la matière et les orientations des systèmes informatiques futurs.

1

Historique de l'architecture RISC

1. Les contraintes pour la conception d'un processeur

1.1. Des processeurs de plus en plus complexes

Traditionnellement, la conception d'un processeur va dans le sens d'un accroissement des fonctionnalités des architectures qui lui sont antérieures. Cette augmentation de la complexité peut être parfaitement illustrée par l'évolution des microprocesseurs chez l'ensemble des grands constructeurs de composants électroniques numériques. Ainsi, Motorola est passé progressivement du 6800 au 6809, puis du 68000 au 68010/12 et enfin du 68020 au 68030, 68040, etc. Il en a été de même pour son principal concurrent, Intel, avec la série des 8080, 8085, 8086, 80186, 80286, 80386 et 80486.

Cette démarche a été longtemps justifiée par le besoin de réduire le fossé sémantique entre les langages de haut niveau et le jeu d'instructions du processeur, tout en ménageant une compatibilité ascendante dans la gamme. En d'autres termes, il s'agit de réduire la différence de puissance d'expression entre le bas niveau, représenté par le code machine, et le haut niveau, représenté par le langage de programmation dit évolué, en gardant compatibles les codes exécutables. Cette évolution est rendue possible par les constants progrès technologiques qui permettent d'intégrer chaque jour plus de transistors sur une puce de silicium.

De ce fait, les trois contraintes principales pour la réalisation d'un processeur ont longtemps été :

1) l'adéquation à un large éventail de langages de programmation,

2) une compatibilité ascendante des jeux d'instructions,

3) le respect des objectifs de performance.

1.2. Relations avec les langages de programmation

Le premier point évoqué traduit l'influence des logiciels et en particulier des compilateurs dans la conception d'une architecture. L'objectif principal étant l'obtention d'une machine générale, la prise en compte d'un grand nombre de besoins se concrétise par l'ajout de fonctionnalités, la multiplication des modes d'adressages, une redondance importante au niveau même des instructions. Par exemple, le processeur MC68020 de la Société Motorola est caractérisé par une centaine d'instructions dont la majorité peut être exécutée selon 18 modes d'adressages différents. En outre, chacune d'entre elles peut manipuler des octets, des mots courts (16 bits) ou des mots longs (32 bits) [1]. Bien que centré sur les langages les plus utilisés, on ne peut pas dire que le processeur MC68020 soit plus particulièrement adapté à Fortran, qu'à C ou Pascal.

1.3. Compatibilité ascendante des jeux d'instructions

Le second point concerne la compatibilité ascendante des jeux d'instructions. Par ces termes, on entend la capacité d'une gamme de processeurs à exécuter un même code machine. Le but principal est d'assurer une meilleure pérennité des applications. Cet aspect, d'un intérêt plus commercial que technique, est cependant rapidement remis en question lorsque l'on veut profiter pleinement des nouvelles spécificités d'un processeur. Cette caractéristique permet néanmoins aux concepteurs de systèmes une transition plus douce lors du "portage" des compilateurs et des logiciels de base.

1.4. Performance des processeurs

Le troisième point consiste évidemment à obtenir la meilleure performance pour un éventail d'applications le plus large possible. La notion de performance est directement liée au temps nécessaire à l'exécution d'une tâche donnée. En pratique, elle dépend de trois critères fortement corrélés :

1) le nombre d'instructions nécessaire à l'exécution de la tâche,

2) le nombre moyen de cycles machines nécessaire à l'exécution d'une instruction,

3) la durée de chaque cycle machine.

Le nombre d'instructions nécessaire reflète généralement le niveau sémantique et l'adéquation du jeu d'instructions à la tâche exécutée. Plus simplement, moins il y a d'instructions et plus le "langage machine" est proche du "langage de haut niveau".

Le nombre de cycles moyen pour exécuter une instruction est une conséquence directe de l'architecture du processeur. Il se situe généralement entre 1 et 2 cycles par instruction pour la plupart des processeurs RISC, jusqu'à plus de 10 cycles par instruction sur des processeurs plus classiques, comme le VAX 780 [2].

Quant à la durée de chaque cycle machine, elle est directement dépendante de la technologie employée. Elle peut varier de

quelques nanosecondes pour des calculateurs en AsGa (Aséniure de Gallium) ou ECL (*Emitter-Coupled Logic*) jusqu'à plusieurs centaines de nanosecondes pour des processeurs réalisés en technologie plus classique, comme le CMOS (*Complementary Metal Oxide Semiconductor*).

La puissance d'un processeur s'exprime généralement en MIPS (Million d'Instructions Par Seconde). Etant donné une tâche, la performance P d'un processeur est proportionnelle à la vitesse de l'horloge S, et inversement proportionnelle au produit du nombre d'instructions de la tâche I et du nombre moyen de cycles par instruction C.

$$P = \frac{S}{I \cdot C}$$

P représente la performance en MIPS
I représente le nombre d'instructions du "benchmark"
C représente le nombre moyen de cycles par instruction
S représente la fréquence d'horloge

Exemples :

Motorola 68030	I=1.0	C= 5.2	S= 16.67	P= 3.21
Intel 80386	I= 1.1	C= 4.4	S= 16.67	P= 3.44
SPARC	I=1.2	C= 1.3	S= 16.67	P= 10.69

(I indique des millions d'instructions, donc P est en MIPS)

Fig.1 - Calcul de la puissance d'un processeur

En pratique, on parle également de "MIPS VAX" où la puissance est exprimée comme un facteur d'accélération par rapport au VAX 780 auquel on attribue, par définition, une puissance de 1 MIPS. Les mesures de temps d'exécutions effectuées sur des programmes jugés représentatifs, appelés *benchmarks*, permettent alors d'établir aisément les rapports de performances en les comparant aux mêmes programmes exécutés avec le processeur VAX 780.

1.5. Complexité horizontale et verticale

Au fil des années, l'ensemble des contraintes évoquées s'est donc traduit par une augmentation de la complexité des architectures et des jeux d'instructions associés. Cette augmentation de la complexité est à la fois horizontale et verticale. L'accroissement de complexité horizontale correspond au passage progressif de processeurs 4 bits, 8 bits et 16 bits, aux processeurs 32 bits qui équipent actuellement la majorité des ordinateurs. Nous noterons au passage qu'il existe également des processeurs caractérisés par un nombre de bits différents, mais ils représentent généralement des approches architecturales spécifiques qui sortent, à ce niveau, du cadre de l'étude. L'accroissement de complexité verticale est celle qui résulte de l'augmentation du nombre d'instructions et des modes d'adressages. Les processeurs traditionnels résultant de cette approche ont été qualifiés de CISC (*Complex Instruction Set Computer* - Ordinateur à jeu d'instructions complexe) par opposition au terme RISC (*Reduced Instruction Set Computer* - Ordinateur à jeu d'instructions réduit).

1.6. Les processeurs CISC

Les contraintes déjà évoquées ont conduit les concepteurs de systèmes à postuler que la performance d'un processeur provient en grande partie de l'exécution d'instructions complexes de haut niveau par le matériel et de son corollaire, c'est-à-dire la réduction du nombre d'instructions à exécuter pour une même tâche. De fait, ces arguments ont été renforcés par le coût de la mémoire et les nouvelles capacités d'intégration qui facilitent la migration des fonctionnalités jugées importantes au niveau du processeur.

Bien qu'il n'existe pas à proprement parler de méthodologie CISC ou de principes de conception bien établis, les "architectes" ont basé leur développement sur une relative indépendance entre la machine réelle et la machine logique vue de l'utilisateur. La technique employée repose sur un modèle d'exécution à plusieurs niveaux qui, par le biais de la microprogrammation, émule le modèle de machine souhaité sur une micromachine constituée d'un interpréteur matériel et d'opérateurs élémentaires.

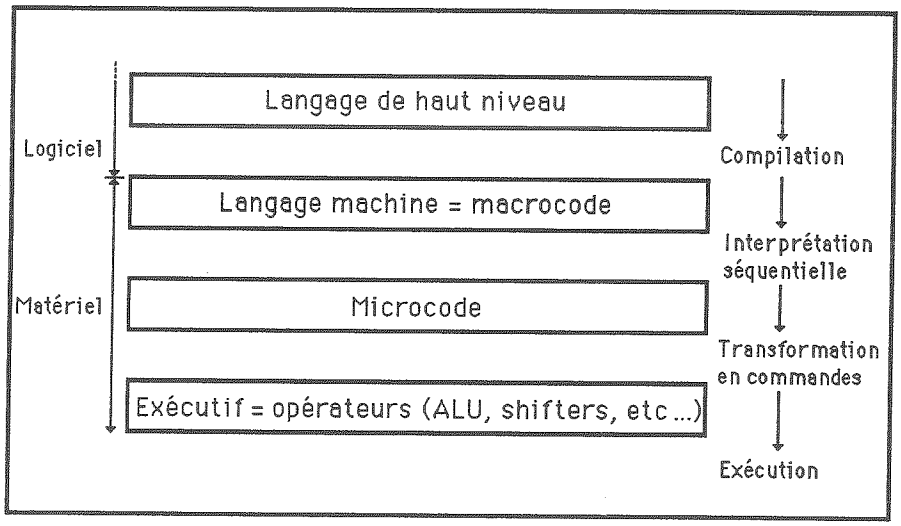


Fig.2 - Schéma d'exécution d'un processeur CISC

La figure suivante donne le schéma fonctionnel général d'un processeur CISC mettant en évidence la structure de la micromachine d'interprétation des microcodes et les opérateurs élémentaires.

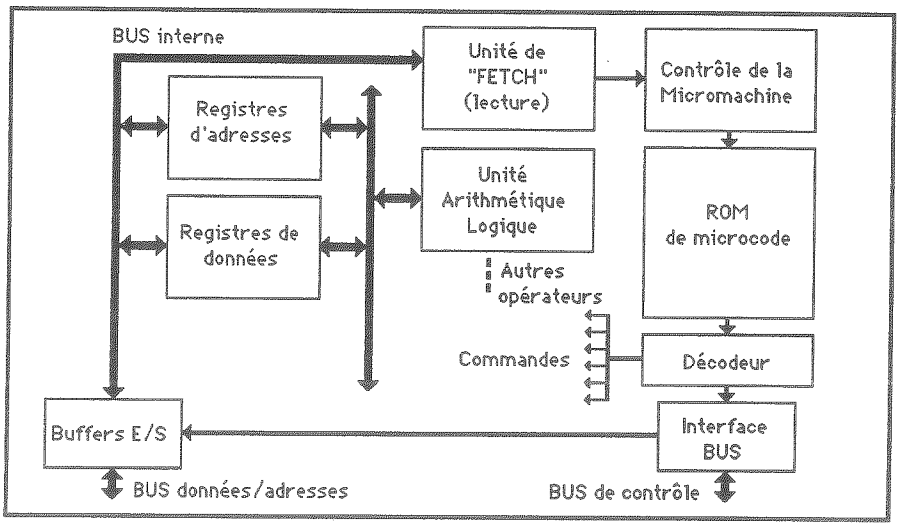


Fig. 3 - Structure générale d'un processeur CISC

Les contraintes de performance et de simplication d'écriture des compilateurs ont ensuite conduit les concepteurs à modifier progressivement ce schéma en réduisant les frontières entre les différents niveaux :

(1) la "microprogrammation horizontale" tend à rapprocher le niveau exécutif (les opérateurs matériel) des micro-instructions en élargissant le nombre de bits de commande.

(2) La "microprogrammation verticale" vise à réduire le fossé entre le langage de haut niveau et les microcodes, à la fois en augmentant la complexité des instructions et en essayant de réduire le nombre de microcodes nécessaires à leur exécution.

2. L'évolution technologique

2.1. Le contexte technologique évolue

Nous avons souligné, lors du précédent paragraphe, que l'évolution des familles de processeurs a été principalement guidée par l'évolution technologique, c'est-à-dire l'intégration de plus en plus de transistors sur une puce de silicium, la nécessité d'une compatibilité ascendante des jeux d'instructions et le postulat selon lequel la performance est directement liée à l'exécution d'instructions complexes. Cette évolution est plus historique que réellement guidée par des analyses scientifiques rigoureuses. En particulier, on peut noter que la première idée est de microcoder le plus grand nombre possible d'instructions jugées significatives, pour seulement, ensuite, résoudre les problèmes de performance à l'exécution.

Mais, dans le même temps, le contexte économique et technologique a fortement évolué. Des modifications significatives sont apparues qui ont suscité une remise en question des postulats antérieurs :

1) l'évolution des mémoires intégrées tend à diminuer l'intérêt d'une mémoire interne de microprogrammes dont le temps d'accès devient sensiblement identique à ceux des composants mémoires externes,

2) ce point est renforcé par l'avènement des techniques de mémoire cache,

3) les progrès réalisés en CAO (Conception Assistée par Ordinateur) électronique et au niveau des filières technologiques ASIC (*Applied Specific Integrated Circuit*) ouvrent de nouvelles possibilités aux concepteurs de systèmes,

4) le nombre très important de logiciels disponibles permet maintenant de mieux analyser le comportement des compilateurs.

2.2. L'évolution des circuits mémoires

Le premier point diminue directement l'intérêt de la technique de microprogrammation. Dans certains cas, on peut montrer qu'elle peut devenir un facteur de dégradation des performances. En effet, lorsque la complexité du processeur est trop importante, l'interprétation de plusieurs microcodes pour l'exécution d'une instruction augmente le nombre de cycles nécessaires (C) et la durée de chaque cycle (S). De ce fait, la puissance effective du processeur est directement affectée.

2.3. Les mémoires caches

Les recherches intensives menées sur les mémoires caches ont mis en évidence des gains de performance bien supérieurs à ceux apportés par la seule microprogrammation des fonctionnalités coûteuses en temps de calcul. Une mémoire cache est un petit tampon de mémoire très rapide placé entre le processeur et la mémoire centrale plus lente. Du fait de la propriété de localité des programmes informatiques, un système de cache permet d'obtenir une mémoire de taille importante, relativement peu coûteuse, avec un temps d'accès apparent satisfaisant. Nous reviendrons plus loin sur ce mécanisme, parfois également appelé "antémémoire" (Chap. 2 § 3.6.).

2.4. La Conception Assistée par Ordinateur

Les progrès obtenus au niveau des systèmes de Conception Assistée par Ordinateur (CAO) tendent également à rationaliser la conception des VLSI (*Very Large Scale Integration*), car plus adaptés à la réalisation de structures simples et régulières. Les technologies ASIC ont donné aux architectes de systèmes la possibilité de concevoir leur propre unité centrale adaptée à leurs besoins pour un coût raisonnable, privilège qui était uniquement réservé auparavant aux grandes firmes spécialisées dans la fabrication de composants électroniques.

2.5. Les compilateurs

Enfin, le point le plus important est sans conteste la maturité des techniques de compilation. Au vu de l'expérience acquise dans la réalisation des compilateurs et l'optimisation des codes générés, il semble indiscutable qu'un ensemble d'instructions simples et uniformes soit plus adapté. Il est clair que plus le jeu d'instructions est complexe et plus la conception d'un compilateur est longue et délicate. En particulier, il est pratiquement impossible d'exploiter pleinement les potentialités offertes par un jeu d'instructions complexe et, pour chaque instruction, de sélectionner le meilleur mode d'adressage. Alors que le maître-mot des concepteurs de logiciels est "portabilité", les compilateurs limitent implicitement l'utilisation d'un trop grand nombre d'instructions en définissant des méta-instructions, sorte de machines virtuelles dont le but est de définir une interface logicielle facilitant le portage des compilateurs. Le P-code utilisé dans le Pascal UCSD [3] et la machine virtuelle LLM3 pour le système Le_Lisp [4] sont deux exemples qui illustrent parfaitement cette approche.
De nombreuses recherches ont porté sur l'analyse des occurrences relatives des instructions générées par les principaux compilateurs, tels Fortran, C ou Pascal. Nous ne mentionnerons ici que les travaux de D. Fairclough ayant pour objectif d'établir un jeu d'instructions unique [5]. L'auteur établit huit classes d'instructions différentes pour en mesurer ensuite l'occurrence statistique moyenne sur un ensemble de programmes sélectionnés aléatoirement, incluant des assembleurs, interpréteurs, compilateurs, moniteurs, noyaux de

systèmes d'exploitations, librairies, tâche de contrôle de procédés et autres utilitaires systèmes. Les huit classes sont les suivantes :

1 - les instructions permettant la manipulation de données (*Data "Movement" Instruction Group*),
2 - les instructions de modification du séquencement du programme (*Program Modification Instruction Group*),
3 - les instructions arithmétiques (*Arithmetic Instruction Group*),
4 - les instructions de comparaison (*Compare Instruction Group*),
5 - les instructions logiques (*Logicial Instruction Group*),
6 - les instructions de décalage (*Shift Instruction Group*),
7 - les instructions de manipulation de bits (*Bit Instruction Group*),
8 - les instructions d'entrées-sorties et diverses (*Input/Output and Miscellaneous Instruction Group*).

OCCURENCE DES INSTRUC. EXECUTEES	POURCENTAGES DU JEU D'INSTRUC. POUR LE MC68000	GROUPE D'INSTRUC. (voir texte)	POURCENTAGES RELATIFS POUR LE MC68000
= 0.0%	30.3%	1	43.52%
<= 0.1%	3.9%	2	25.13%
<= 0.5%	21.1%	3	12.09%
<= 1.0%	11.8%	4	9.15%
<= 2.0%	10.5%	5	5.03%
<= 3.0%	13.2%	6	2.65%
<= 4.0%	0.0%	7	2.36%
<= 5.0%	6.6%	8	0.07%
> 5.0%	2.6%		
TOTAL 100%		TOTAL 100%	

Fig. 4 - Statistiques sur le jeu d'instructions du MC68000

La figure 4 donne les résultats mesurés sur le processeur 68000 de Motorola qui marque l'apogée des architectures CISC. Elle montre que les instructions de manipulation des données (groupe 1) représentent 43,52% des instructions exécutées et que seulement 2,6% du jeu d'instructions du processeur sont caractérisés par une occurrence supérieure à 5%. Le tableau montre, en outre, que 30,3% du jeu d'instructions n'est pas utilisé.

L'auteur conclut que 31 instructions sur les 76 que comporte le processeur 68000 auraient pu être éliminées sans remettre fondamentalement en question les performances du processeur.

2.6. La remise en question des processeurs CISC

Au cours du précédent paragraphe, nous avons insisté sur le fait que la conception d'un jeu d'instructions a toujours été, dans le passé, basée sur l'évolution plutôt que la révolution. La plupart des jeux d'instructions actuels sont de simples extensions de jeux antérieurs. La raison principale que nous avons évoquée est la nécessité d'une compatibilité ascendante dans une gamme de processeurs. Une explication complémentaire est la difficulté à concevoir un jeu d'instructions efficace et complet. La complexité de cette tâche force le concepteur à des choix heuristiques. En particulier, il doit émettre les meilleures hypothèses quant à l'utilisation ultérieure réelle des instructions qu'il définit. Dans un tel contexte, il trouve généralement plus sûr de se baser sur un jeu d'instructions existant plutôt que d'en bâtir totalement un nouveau.

Mais cette démarche conduit généralement à une augmentation de la complexité du jeu d'instructions, qui entraîne en retour un alourdissement de l'architecture et un ralentissement général du processeur par un accroissement du nombre de cycles nécessaires et de la durée de chaque cycle. En particulier, les unités de décodage et de séquencement sont directement affectées par l'ajout de fonctionnalités. Ainsi, dans la plupart des processeurs CISC, on peut estimer que l'ensemble des blocs de séquencement, contrôle et commande, occupe près de 50% de la surface du circuit, contre seulement 5% dans un "pur" processeur RISC.

En définitive, les postulats émis pour la conception des processeurs CISC semblent être remis en question par l'avancée technologique qui en permit l'essor, à la fois au niveau du matériel et au niveau des compilateurs.

3. Les précurseurs de l'approche RISC

3.1. Une idée déjà ancienne

Les deux premiers paragraphes ont introduit l'idée que l'avènement de la méthodologie RISC n'est pas une révolution, mais plutôt la formalisation, puis la reconnaissance d'un ensemble de critères de conceptions guidés par l'avancée technologique.

Nous verrons plus loin qu'un des principes fondamentaux de l'architecture RISC réside dans l'intégration des fonctionnalités à forte occurrence et seulement celle-ci. Déjà, le père des ordinateurs, John Von Neumann lui-même, déclarait en 1946 :

We wish to incorporate into the machine - in form of circuits - only such logical concepts as are either necessary to have a complete system or highly convenient because of the frequency which they occur.

(Nous voulons implanter dans la machine - sous la forme de circuits - seulement les concepts logiques nécessaires pour obtenir un système complet ou très pratique, justifiés par leur fréquence d'utilisation.)

Cette déclaration prouve que l'idée est loin d'être nouvelle. On peut dire que la notion d'une structure abstraite dotée d'un jeu d'instructions, a été introduite pour la première fois en 1964 par la société IBM (*International Business Machine Corporation*) avec l'annonce des systèmes 360. La machine IBM 360 est basée sur le principe déjà évoqué de microprogrammation qui permet de construire des instructions à partir de micro-instructions élémentaires présentes dans une mémoire ROM (*Read-Only Memory*) près du processeur. Un des principaux problèmes techniques réside dans le fait qu'aucun microcode n'est parfait, que leur mise au point est coûteuse et que la maintenance est difficile. Néanmoins, les systèmes IBM 370/168 ou Digital VAX 11/780 sont caractérisés par plus de 400.000 bits de microcode.

3.2. Le supercalculateur Cray-1

Mais un certain nombre d'ingénieurs comprirent rapidement que si la technique de microprogrammation donnait un excellent rapport flexibilité/performance, elle n'était pas une fin en soi.

Seymour Cray fut l'un des premiers architectes à concevoir des ordinateurs basés sur des jeux d'instructions simples et orientés vers l'utilisation des registres plutôt que de la mémoire. Les machines CDC6400 (1964) et CDC7600, puis le super-ordinateur Cray-1 sont les concrétisations de cette approche [6]. Ils représentent de ce fait les précurseurs des architectures RISC modernes. En 1975, Seymour Cray déclarait, à propos de ses réalisations :

[registers] made the instructions very simple ... That is somewhat unique. Most machines have rather elaborate instruction sets involving many more memory references in the instructions than the machines I have designed. Simplicity, I guess, is alway of saying it. I am all for simplicity. If it's very complicated, I can't understand it.

([Les registres] rendent les instructions très simples ... C'est un fait unique. La plupart des machines ont plutôt un jeu d'instructions sophistiqué impliquant beaucoup plus d'accès à la mémoire dans les instructions que dans les machines que j'ai conçues. La simplicité, je crois, est une manière de le dire. Je suis pour la simplicité. Quand c'est trop compliqué, je ne peux pas comprendre.)

3.3. La machine IBM 801

A la même époque, John Cocke, ingénieur de la société IBM travaillait avec son équipe sur un projet de système de commutation pour les télécommunications. De tels systèmes requièrent des contrôleurs très rapides. Ce projet fut abandonné, mais l'équipe repris la structure du contrôleur pour la transformer en un véritable ordinateur. Le résultat de cet effort fut l'annonce en 1979 de la machine IBM 801 [7]. D'après John Cocke, la réduction du nombre d'instructions fut plus une conséquence qu'un a priori. Leurs travaux, basés sur l'analyse des listings comportant les traces statistiques sur l'occurrence des instructions utilisées, les

convainquirent de ne pas adjoindre d'instructions complexes, là où elles pouvaient être efficacement remplacées par des séquences d'instructions plus simples.

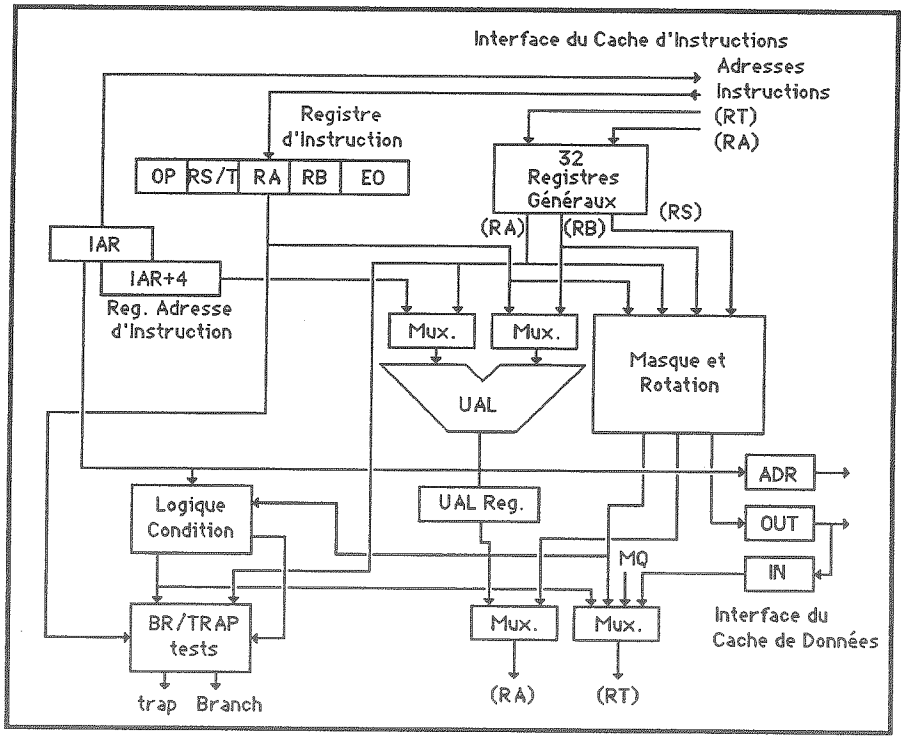


Fig. 5 - Synoptique du processeur IBM 801

La machine 801 fut conçue avec de la mémoire très rapide, en technologie ECL (*Emitter-Coupled Logic*) sur la base d'un jeu d'instructions aux formats fixes, chacune d'entre elles devant être exécutée en un seul cycle machine. Déjà, les solutions préconisées prévisageaient les principes de l'architecture RISC : une structure pipeline régulière, un jeu d'instructions simple et homogène sans microprogrammation. Ces deux principes seront étudiés en détail dans cet ouvrage (Chap. 2 § 2. et 3.).

Ainsi, que l'on parle de RISC ou de CISC, les fondements de ces architectures semblent provenir des travaux menés au sein de la société IBM. Il fallut cependant attendre 1980 pour que le terme RISC apparaisse explicitement pour la première fois. A la même époque, deux projets débutaient, baptisés RISC (*Reduced*

Instruction Set Computer) à l'Université de Berkeley pour le premier et MIPS (*Machine without Interlocked Pipeline Stages*) à l'Université de Stanford pour le second.

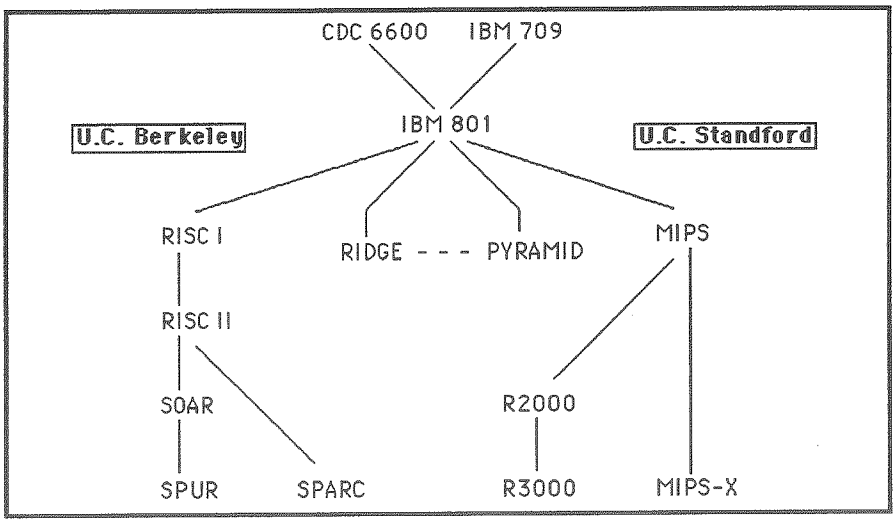


Fig. 6 - Généalogie succincte de l'architecture RISC

4. L'école RISC à l'Université de Berkeley

4.1. Le projet RISC-I

Le terme RISC fut introduit pour la première fois dans le cours du professeur D.A. Patterson à l'Université de Berkeley pendant l'année 1980. Le projet RISC-I débuta effectivement le 6 Janvier 1981 et aboutit le 23 Octobre suivant à la livraison des premiers prototypes. Le circuit fut conçu selon la célèbre méthode Mead-Conway [8] et intégré dans une technologie NMOS 2 microns. L'objectif du projet était d'explorer une alternative aux conceptions classiques. L'hypothèse initiale postulait que la réduction du nombre d'instructions permettrait d'obtenir une architecture VLSI optimisée et performante [9]. Pour obtenir ces résultats, le professeur

Patterson et son équipe formulèrent quatre contraintes principales sur la méthodologie de conception :

1) exécuter une instruction par cycle,

2) toutes les instructions ont une même taille,

3) les accès à la mémoire ne sont effectués qu'au travers de deux instructions : *load* et *store*, les autres opérant uniquement dans les registres,

4) l'adéquation aux langages de haut-niveau.

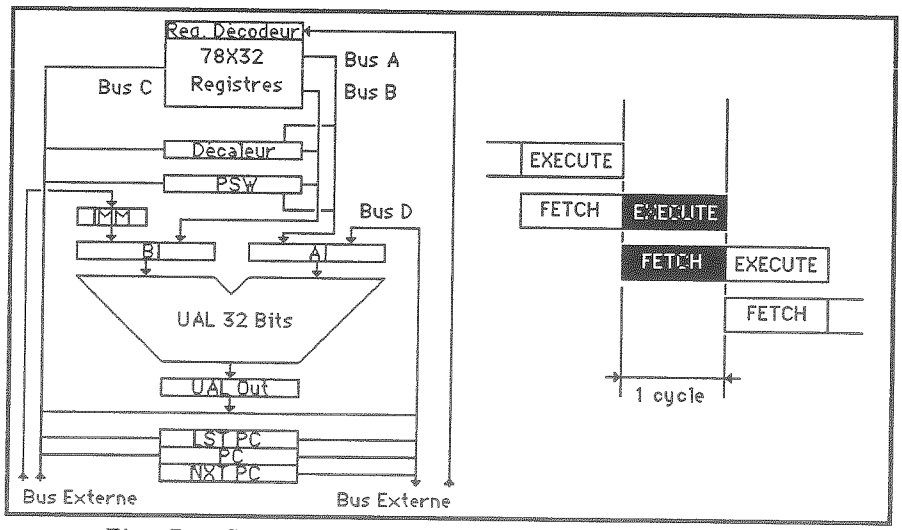


Fig. 7 - Synoptique et pipeline du processeur RISC-I

La définition de l'architecture et plus particulièrement du jeu d'instructions, fut largement inspirée par le langage C [10]. Il fut choisi d'une part pour sa large utilisation et, d'autre part, à cause de l'expertise existante à Berkeley. Pour la détermination des opérations à forte occurrence, les chercheurs se basèrent en premier lieu sur les différentes classes de variables. D'autres mesures portèrent ensuite sur la fréquence relative des constructions classiques des langages de haut-niveau, tels que *if, with, loop, case, goto, call* ou *return* [11].

Les observations les plus importantes furent les suivantes :

- les instructions d'appel (*call*) et de retour (*return*) sont celles qui exigent le plus de temps d'exécution,
- 80% des variables locales utilisées dans les programmes sont des scalaires,
- 90% des structures de données complexes sont des variables globales,
- la majorité des procédures n'échangent que six arguments au maximum,
- la profondeur maximale pour l'appel de procédure est en moyenne huit, une profondeur plus grande arrivant dans moins de 1% des cas.

Ces résultats suggèrent que les points les plus importants sont la manipulation efficace des variables locales, des constantes et l'accélération des appels et retours de procédure. Sur cette base, le jeu d'instructions de RISC-I fut divisé en quatre catégories pour un total de 31 instructions : arithmétique et logique, accès à la mémoire, branchements et opérations diverses. Le temps d'exécution d'un cycle fut déterminé par le temps nécessaire pour lire deux registres, effectuer une addition puis écrire le résultat dans un registre.

La plupart des instructions de RISC-I s'inspirent de la machine PDP-11 de Digital Equipment [12]. Les principales innovations sont :

1) le mécanisme de "fenêtre de registres" (*register windows*) associé aux instructions *call* et *return*,

2) l'instruction de "branchement retardé" (*delayed jump*).

L'objectif des fenêtres de registres est d'accélérer les appels et retours de procédure. En outre, l'occurrence des variables locales et des scalaires justifie l'utilisation de registres pour les stocker. Déjà, Baskett [13] et Sites [14] avaient proposé des structures de processeur caractérisées par des bancs multiples de registres pour éviter les coûteux mécanismes de sauvegarde et restitution de contexte. En effet, un appel de procédure (*call*) comprend généralement, outre le branchement à l'adresse de la fonction appelée, la sauvegarde du compteur ordinal, des variables locales et le passage des arguments de la fonction. Corollairement, un retour fonctionnel (*return*) doit passer les résultats et restituer le contexte de la procédure appelante avant d'effectuer le retour proprement-dit. Nous étudierons plus loin en détail le mécanisme des fenêtres

de registres, mis au point à Berkeley, qui permet l'exécution de l'ensemble de ces opérations en une seule instruction (voir Chap. 2 § 3.2.). Cette technique représente une des innovations majeures de l'architecture RISC et que l'on retrouve maintenant dans un nombre important de processeurs commercialisés.

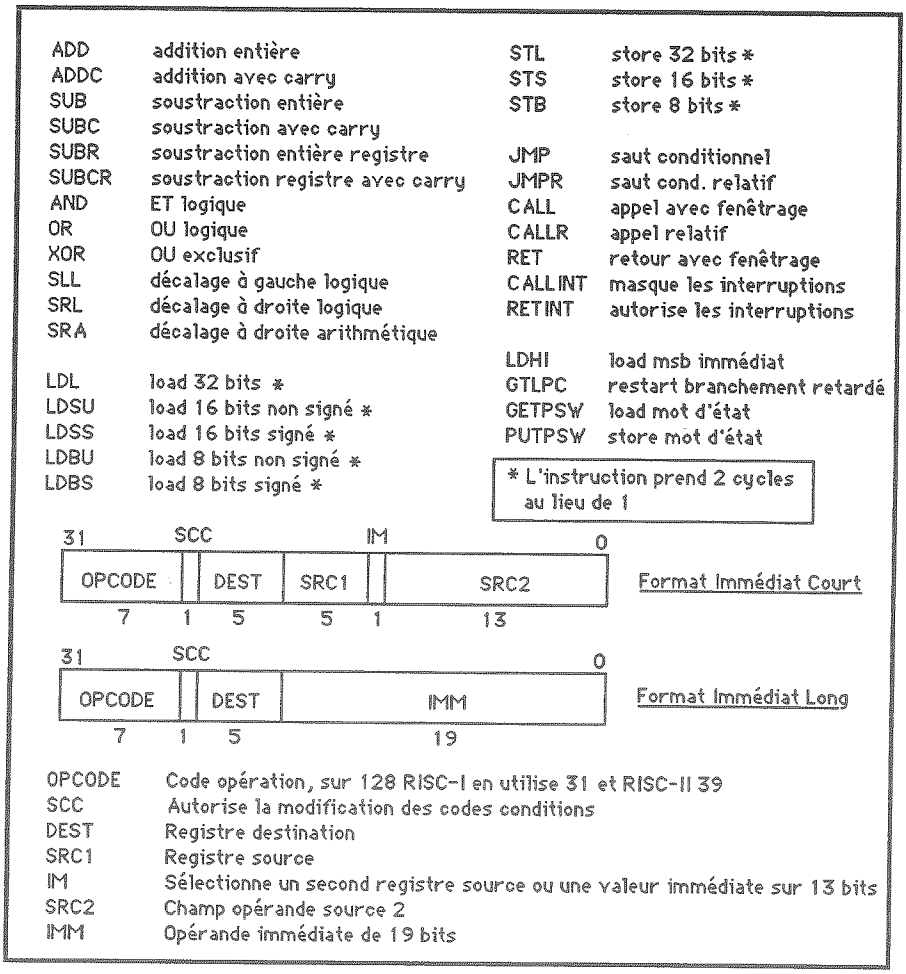


Fig. 8 - Le jeu d'instructions et les formats d'instructions RISC-I

Le mécanisme de branchement retardé, quant à lui, permet d'obtenir un schéma d'exécution pipeline régulier sans qu'aucune exception ne modifie le séquencement normal d'une instruction. En effet, les branchements posent le délicat problème d'une rupture dans le séquencement du processeur ayant pour objectif la lecture

(*fetch*) de l'instruction suivante à chaque cycle machine. Si un cycle est suffisant pour incrémenter l'adresse courante, deux cycles sont généralement nécessaires pour affecter au compteur ordinal une nouvelle adresse présente dans une opérande de l'instruction exécutée. Le principe expérimenté dans RISC-I repose sur l'insertion d'une instruction *no-operation* juste après l'instruction de branchement, qui laisse le temps nécessaire pour calculer la nouvelle adresse. Un optimiseur intégré au compilateur aura ensuite pour tâche de réarranger l'ordre des instructions pour tenter d'insérer à la place une instruction plus utile. Nous analyserons finement ce mécanisme au cours d'un prochain chapitre (voir Chap. 2 § 2.4.).

BENCHMARK	RISC-I		MC68000		VAX 11/780	
	N	T	N	T	N	T
STRING SEARCH	144	.46	.8	2.8	.7	1.3
BIT TEST	120	.06	1.2	4.8	1.2	4.8
LINKED LIST	176	.10	.7	1.6	1.2	1.2
BIT MATRIX	288	.43	1.1	4.0	1.0	3.0
QUICKSORT	992	50.4	.7	4.1	.9	3.0
ACKERMAN(3,6)	144	3200	-	-	.5	1.6
PUZZLE(SUBSCRIPT)	2736	4700	-	-	.5	2.0
PUZZLE(POINTER)	2796	3200	.9	4.2	.5	1.3
RECURSIVE QSORT	752	800	-	-	.6	2.3
SED(BATCH EDITOR)	17720	5100	-	-	.6	1.1
TOWERS HANOI(18)	96	6800	-	-	.8	1.8
Moyenne			.9±.2	3.5±1.8	.8±.3	2.1±1.1

N: Tailles des programmes C pour RISC-I et ratio des tailles comparées pour le MC68000 et le VAX 11/780

T: Temps d'exécution en millisecondes pour RISC-I et nombre de fois plus lent pour le MC68000 et le VAX 11/780

Fig. 9 - Performances RISC-I, MC68000 et VAX 11/780

La conception du processeur RISC-I prit environ 15 hommes/mois et la phase de routage pratiquement 12. Le résultat fut un circuit de 44.500 transistors, qui, malgré quelques erreurs minimes, fonctionna dès les premières mises sous tension. Le plus rapide des circuits RISC-I permit l'exécution d'un programme à la fréquence de 1,5 Mhz. Même à cette fréquence relativement faible, les expérimentations réelles sur carte, menées en Juin 1982, confirmèrent que RISC-I était plus rapide que la plupart des microprocesseurs commercialisés.

4.2. Le projet RISC-II

Parallèlement, Bob Sherbune et Katevenis, toujours sous la direction du professeur Patterson, débutaient la conception d'une seconde version plus efficace. L'ambition de cette nouvelle version était plus importante, bien que fondamentalement basée sur la même micro-architecture et le même jeu d'instructions. Baptisé "RISC-I Blue", puis ensuite RISC-II, le circuit fut conçu en parallèle avec la réalisation RISC-I (nommé alors RISC-I Gold) puis fabriqué et testé pendant l'année 1983.

Hiver 80	Idée de RISC	Patterson, Séquin
Printemps 80	Etude architecture	Patterson, 15 étudiants
Eté 80	Définition de l'architecture	Patterson, 4 étudiants
Eté/Fin 80	Compilateur, asm., simulations	Campbell, Tamir
Eté/Fin 80	Architecture RISC-I	Katevenis
Hiver 81	Architecture RISC-II	Katevenis
Hiver/Printemps 81	Conception CAO RISC-I	Fitzpatrick, Foderaro, Vandyke...
Eté 81/Printemps 82	Fabrication RISC-I	MOSIS XEROX
Eté 82	Test de RISC-I	Foderaro, VanDyke
Printemps/Eté 82	Carte RISC-I	VanDyke
Hiver 81/Hiver 83	Conception CAO de RISC-II	Katevenis, Sherburne
Printemps 83	Fabrication RISC-II	MOSIS XEROX
Eté 83	Test de RISC-II	Katevenis, Sherburne
1981/82	RISC/E ECL (étude papier)	Beck, Davis...
Printemps/Fin 82	Réalisation cache d'instructions	Hill, Lioupis, Nyberg, Sippel
Printemps 83	Fabrication du circuit cache	MOSIS XEROX
Eté 83	Test du circuit cache	Kioupis, Hill
Fin 82/Fin 83	Implantation RISC en CMOS	Takada
Hiver 83/...	Micro-ordinateur RISC-II	Lioupis, Campbell

Fig. 10 - Le projet RISC à Berkeley

L'ensemble de la conception du processeur RISC-II est parfaitement détaillé dans le mémoire de thèse de Katevenis [15]. Il fut également étudié une version ECL [16] ainsi qu'un système de mémoire cache [17]. Une différence importante entre les deux circuits réside dans l'organisation du pipeline : RISC-I est basé sur un pipeline à deux étages du type *fetch-execute* (lecture d'une instruction/exécution), alors que RISC-II tire parti d'un pipeline à trois étages *fetch-compute-write* (lecture/exécution/écriture).

Le circuit RISC-II est le véritable précurseur des architectures RISC modernes. Plus qu'une architecture, la méthodologie RISC bouleverse les postulats établis pour la conception d'un processeur en proposant une approche logique, basée sur une analyse

descendante rigoureuse qui conduit à la réalisation de processeurs à la fois simples et efficaces.

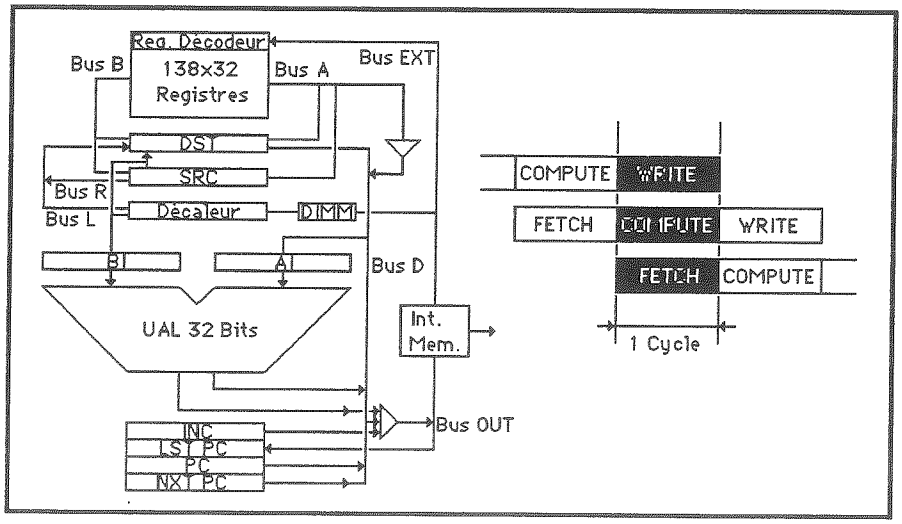


Fig. 11 - Synoptique et pipeline du processeur RISC-II

4.3. Le processeur SOAR

Mais l'équipe du professeur Patterson ne s'arrête pas en si bonne voie, puisqu'elle débute, en 1983, le projet SOAR (*Smalltalk On A Risc*) [18-19], un processeur RISC dédié à l'exécution de l'environnement Smalltalk, langage issu des travaux en Intelligence Artificielle menés au sein de la Société Xerox. SOAR, encore plus que RISC-II, démontre l'intérêt et la puissance de la méthodologie RISC.

SOAR marque la première tentative d'expérimentation de la méthodologie RISC pour l'exécution efficace du traitement symbolique. Comme RISC-II, il hérite directement des travaux menés sur le projet RISC de Berkeley, mais il profite également des recherches sur la machine IBM 801 et la première version du processeur MIPS. Au niveau des supports spécifiques pour le traitement symbolique, SOAR adopte une structure étiquetée (*tagged-RISC architecture*) initialement proposée sur la machine Burroughs 5000, mais surtout utilisée dans la station Symbolics 3600 [21]. Le processeur SOAR fut réalisé en deux versions, la première en technologie NMOS [22] et la seconde en CMOS pour une

complexité d'environ 35300 transistors [23]. Il influença ensuite nombre de recherches, dont certains travaux d'évaluation menés au sein de la Société Texas Instruments [24], certaines fonctionnalités du processeur SPARC de SUN Microsystems et, bien sûr, KIM sur lequel nous reviendrons en détail.

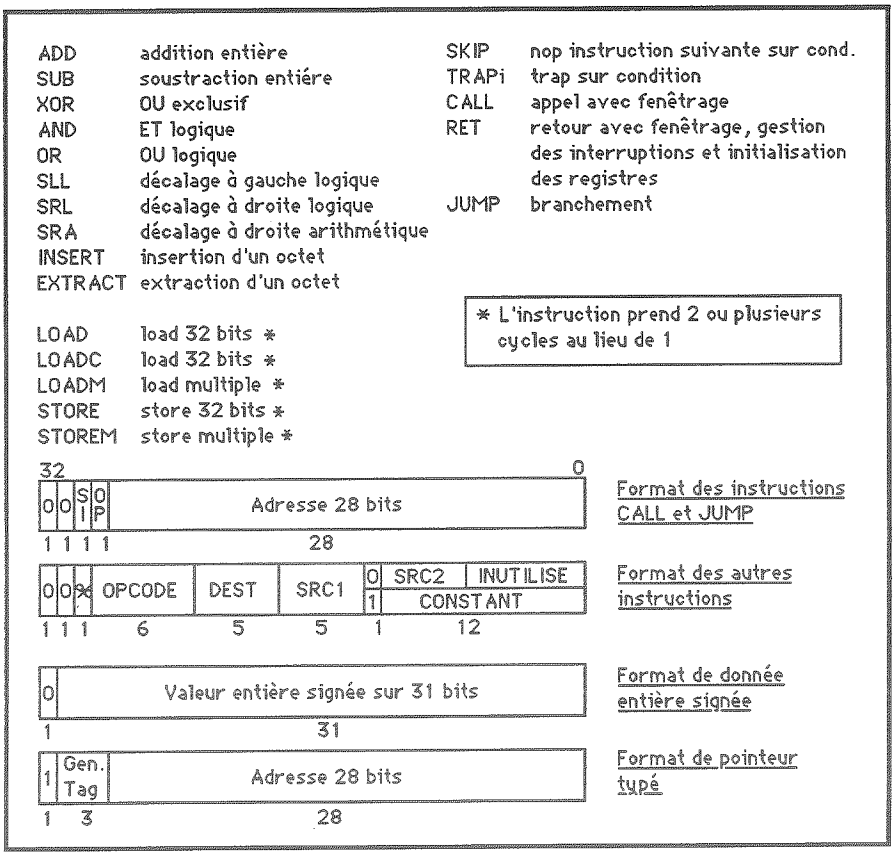


Fig. 12 - Formats et jeu d'instructions du processeur SOAR

4.4. La machine SPUR

En 1985, l'équipe RISC de Berkeley initialise un projet ambitieux. Baptisé SPUR (*Symbolic Processing Using RISC*) [25-26], il vise la conception d'une station de travail distribuée qui doit servir de moteur pour la recherche sur le traitement parallèle. Le projet SPUR

est complexe car il impose des efforts importants dans plusieurs domaines : ceux des circuits intégrés, de l'architecture des ordinateurs, des systèmes d'exploitation et des langages de programmation. Le système SPUR repose sur un bus rapide permettant l'intégration de 6 à 12 unités de traitement, comprenant chacune : le processeur RISC proprement dit, un accélérateur de calcul virgule flottante et un système de cache sophistiqué.

L'architecture SPUR est conçue spécialement pour supporter le langage Common-Lisp [27], standard adopté dans le monde entier pour la réalisation d'applications en Intelligence Artificielle. Si SPUR hérite directement des travaux antérieurs, les différences n'en sont pas moins importantes : un séquencement d'exécution sur quatre étages au lieu de trois pour RISC-II et SOAR, des supports spécifiques pour assurer une exécution efficace du traitement symbolique, une architecture de mémoire cache garantissant la cohérence des données dans la structure répartie.

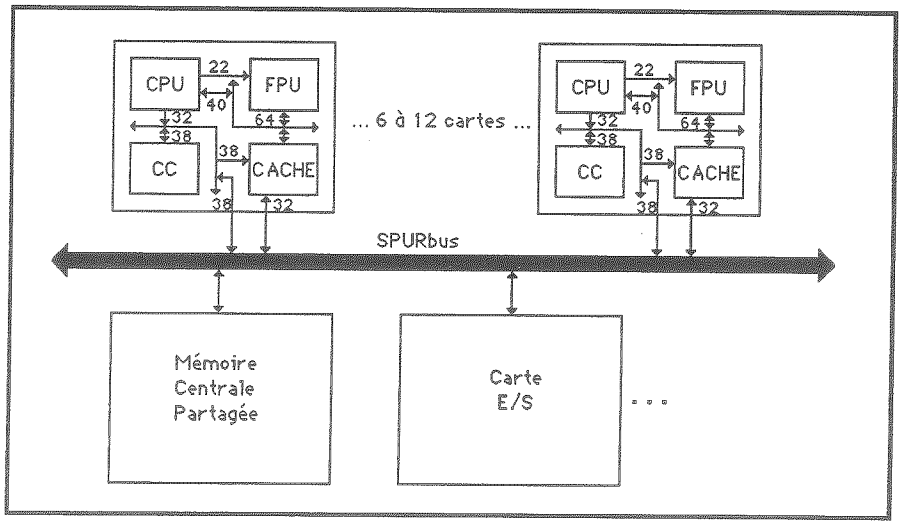


Fig. 13 - Architecture de la machine SPUR

L'architecture SPUR vise des performances bien supérieures à celles obtenues avec RISC-II ou SOAR. Comme ce dernier, il repose sur un modèle étiqueté pour l'accélération des mécanismes inhérents aux programmes de l'Intelligence Artificielle. Mais, contrairement à RISC-II, la structure du processeur est plus sophistiquée et l'architecture du système beaucoup plus complexe.

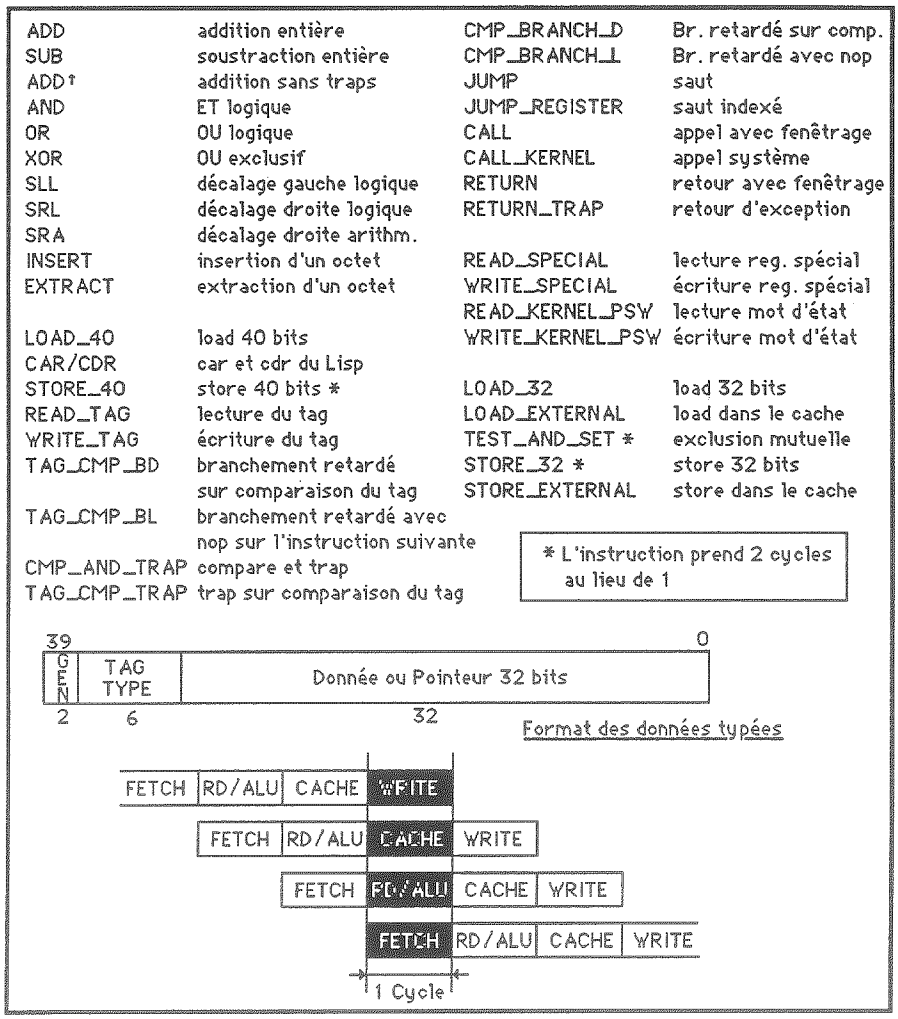


Fig. 14 - Pipeline, format des données et jeu d'instructions SPUR

Si les résultats des recherches menées à l'Université de Berkeley sont les plus connus, ils ne sont pas les seuls.

A quelques dizaines de kilomètres seulement de Berkeley, le professeur John Hennessy débutait, en 1981, le projet MIPS au sein de la célèbre Université de Stanford.

5. L'école MIPS à l'Université de Stanford

5.1. Le processeur MIPS

John Hennessy, professeur à l'Université de Stanford, fut également l'un des pionniers de l'architecture RISC. Après avoir réalisé avec succès un premier processeur [28], John Hennessy fut à l'origine de la Société MIPS Computer Systems, installée aujourd'hui à Sunny-Vale en Californie. Ce processeur, baptisé MIPS (*Machine without Interlocked Pipeline Stages*), comme RISC-I puis RISC-II, est basé sur la simplification du jeu d'instructions et une structure d'exécution pipeline. Mais le processeur MIPS, comparé à RISC-II, est encore plus simple. En particulier, la résolution des problèmes d'interblocages inhérents aux pipelines n'est pas résolu par une adjonction de matériel, mais par un logiciel spécifique nommé *reorganizer*. C'est précisément de cette caractéristique que provient l'acronyme qui forme le nom du processeur, tout en rappelant astucieusement que l'objectif de ces travaux est d'obtenir une architecture performante. Nous reviendrons en détail sur ce point technique plus loin.

Comme RISC-II, le processeur MIPS est une architecture 32 bits caractérisée par 31 instructions codées sur 32 bits. Celles-ci se répartissent en quatre catégories : les opérations arithmétiques et logiques, les instructions d'accès à la mémoire (*load/store*), les opérations de contrôle du séquencement et une série d'instructions spéciales, en particulier pour l'interfaçage avec des coprocesseurs.

Une différence importante avec RISC-II concerne le pipeline d'exécution. En effet, le pipeline MIPS, basé sur cinq cycles recouvrants, autorise l'exécution simultanée de trois instructions. L'exécution d'une instruction débute tous les deux cycles machines : le premier cycle (IF - *Instruction Fetch*) effectue la lecture de l'instruction en mémoire, puis incrémente le compteur ordinal (PC) ; le second cycle (ID - *Instruction Decode*) décode ensuite l'instruction ; le troisième cycle (OD - *Operand Decode*) décode les opérandes ; le quatrième cycle effectue l'écriture d'une opérande en mémoire si l'instruction est un *store* (OS - *Operand Store*) ou sinon exécute l'opération sur l'Unité Arithmétique et Logique (SX) ; enfin,

le cinquième cycle effectue la lecture d'un mot mémoire si l'instruction est un *load* (OF - *Operand Fetch*).

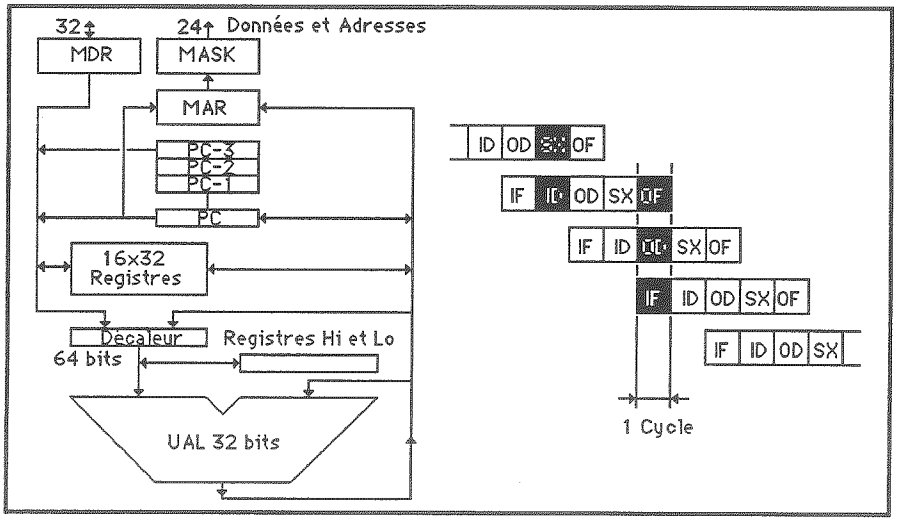


Fig. 15 - Synoptique et pipeline du processeur MIPS

La différence fondamentale avec RISC est l'étude conjointe de l'achitecture avec le compilateur associé. Ainsi, le jeu d'instructions MIPS est défini à deux niveaux :

1) le niveau utilisateur, constitué du langage assembleur qui fait abstraction des problèmes d'interdépendances dus au pipeline et des problèmes de latences dus aux instructions debranchement,

2) le niveau machine, exécuté par le processeur.

Après la phase de compilation puis d'assemblage, le code généré doit être préparé pour l'exécution par un logiciel appelé *reorganizer* que nous avons déjà évoqué. Ce dernier modifie l'ordre initial des instructions pour supprimer les instructions NOP (*No-Operation*) dues aux branchements et éliminer les conflits d'accès aux ressources (interdépendance des étages du pipeline). Une telle approche permet, d'une part d'obtenir un processeur très simple, et, d'autre part de gagner pratiquement 30% du temps d'exécution par rapport au code initial.

ADD	addition entière	BRA (1)	branchement relatif conditionel
AND	ET logique	BRA (2)	branchement relatif inconditionel
IC	insertion octet	JMP (1)	saut absolu direct
OR	OU logique	JMP (2)	saut avec base
RLC	rotation combinée	JMP (3)	saut indirect
ROL	rotation	TRAP	exception logicielle
SLL	décalage gauche logique		
SRA	décalage droite arithm.	LD (1)	load avec registre base
SRL	décalage droite logique	LD (2)	load indexé et registre base
SUB	soustraction entière	LD (3)	load registre base décalé
XC	extraction octet	LD (4)	load direct
XOR	OU exclusif	LD (5)	load immédiat
		MOV	mouvement de registres
SAVEPC	sauvegarde PC	ST (1)	store avec registre base
SET	reg. à 1 si condition	ST (2)	store indexé et registre base
	sinon 0	ST (3)	store registre base décalé
		ST (4)	store direct

Fig. 16 - Le jeu d'instructions MIPS

Le processeur MIPS fut réalisé en technologie NMOS 2 microns dans un boîtier de 84 broches. Composés de 24000 transistors, les premiers échantillons fonctionnèrent à 4 Mhz (250 ns par cycle machine) pour une performance crête de 2 MIPS.

5.2. Le projet MIPS-X

Depuis l'été 1984, l'équipe du professeur Hennessy travaille sur le projet MIPS-X : une machine multiprocesseur à mémoire partagée [29]. L'architecture de MIPS-X repose sur un processeur VLSI intégré dans une technologie standard CMOS 2 microns avec deux niveaux de métallisation. Une performance de 20 MIPS crête a pu être obtenue grâce à un format d'instruction plus simple, un cache d'instruction intégré et une horloge plus rapide, comparée à la machine MIPS initiale.

Les seules instructions câblées sont celles à très forte occurrence. Ces mêmes instructions ont été réalisées de manière à être exécutées en un seul cycle machine. Le jeu d'instructions MIPS-X est composé de trois catégories :

1) les instructions d'accès à la mémoire (*load* et *store*),

2) les instructions de branchement,

3) les instructions de traitement.

Cette dernière catégorie utilise uniquement un banc monolithique de 32 registres, contrairement à la plupart des autres machines RISC qui tirent parti du principe des fenêtres de registres mis au point à Berkeley. Une autre particularité de MIPS-X réside dans un pipeline à 5 étages modifié par rapport à celui implanté dans MIPS. Le premier cycle effectue la lecture de l'instruction (*Instruction Fetch*), le second cycle effectue la lecture des registres opérandes (*Register Fetch*), le troisième cycle est dédié à l'Unité Arithmétique et Logique (ALU) pour réaliser l'opération, le quatrième cycle est réservé aux instructions *load* et *store* pour les accès à la mémoire (MEM), enfin, le cinquième et dernier cycle écrit le résultat de l'instruction dans le registre destination (*Write Back*).

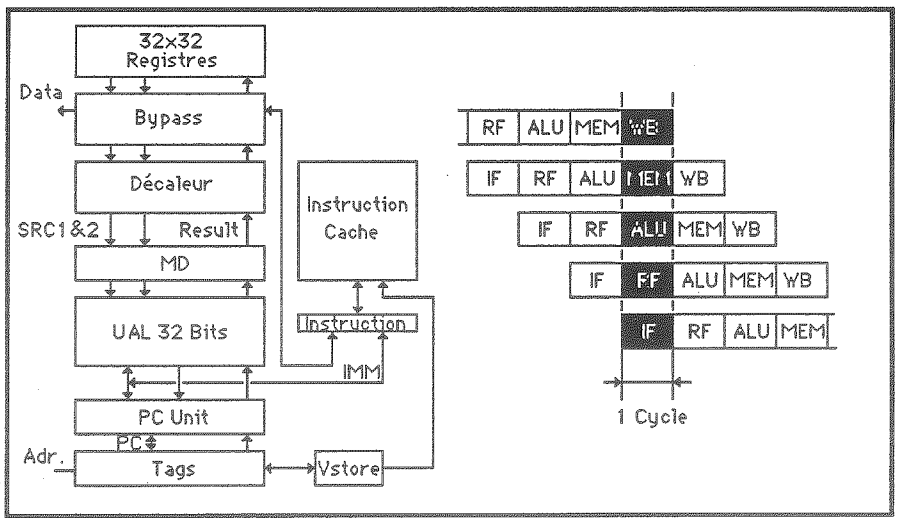


Fig. 17 - Synoptique et pipeline MIPS-X

La première version du processeur MIPS-X fut envoyée en fabrication en Mai 1986. Les premiers échantillons reçus en Octobre 1986 validèrent fonctionnellement le processeur à une fréquence de 16 Mhz.

Des modifications mineures de certains chemins de données et de l'interface extérieure permirent d'envisager un temps de cycle élémentaire de 50 ns.

LD	load	ADD	addition entière
ST	store	DSTEP	pas de division entière
LDF	load format flottant	MSTART	1er pas de multiplication entière
STF	store format flottant	MSTEP	pas de multiplication entière
LDT	load sans cache	SUB	soustraction entière
STT	store sans cache	SUBNC	soustraction sans carry
MOVFRC	move depuis coprocesseur	AND	ET logique
MOVTOC	move vers coprocesseur	BIC	mise à zéro d'un bit
ALUC	opération coprocesseur	NOT	complément à un
		OR	OU logique
ADDI	addition immédiat	XOR	OU exclusif
JPC	saut PC	MOV	mouvement de registres
JPCRS	saut PC et restitue état	ASR	décalage à droite arithmétique
JSPCI	saut indexé et sauve PC	ROTLB	rotation à gauche par octets
MOVFRS	move depuis reg. spécial	ROTLCB	idem complémenté
MOVTOS	move vers reg. spécial	SH	décalage
TRAP	exception logicielle	NOP	pas d'opération
HSC	halt et destruction (!)		

BEQ	branchement si égal
BGE	branchement si plus grand ou égal
BHS	branchement si supérieur ou égal
BLO	branchement si plus petit
BLT	branchement si inférieur
BNE	branchement si non égal

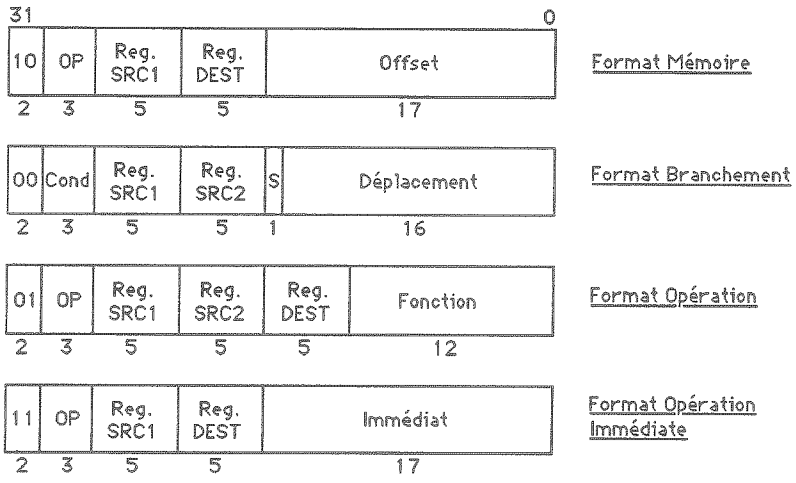


Fig. 18 - Formats des instructions MIPS-X

Une grande partie de l'effort de développement porta sur la réalisation du cache d'instruction intégré. Le but est d'obtenir un système de mémoire cache simple avec de bonnes performances.

Autrement dit, un cache de "petite taille" (2Kbytes) avec un bon taux de réussite (> 90%). Le système de cache intégré est organisé sous la forme de 32 bancs de 16 mots de 32 bits, dotés chacun d'une logique spécifique de gestion.

L'ensemble fut implanté finalement sous la forme d'un banc de mémoire statique 512 x 32 bits directement à côté du processeur sur la puce de silicium. Les résultats obtenus avec le système de mémoire cache intégré montre que chaque instruction nécessite en moyenne 1,5 cycle machine, ce qui permet d'obtenir une puissance efficace de 10 MIPS à 20 Mhz.

Mais le projet MIPS-X ne s'arrête pas à l'intégration d'un circuit VLSI. L'objectif de performance visé par MIPS-X consiste à dépasser la frontière des 100 MIPS efficaces pour une machine comportant plusieurs processeurs organisés autour d'un système de communication sophistiqué. Les architectures expérimentées reposent sur un modèle distribué hybride *Snoopy/Directory* où des grappes de processeurs MIPS-X (*cluster*) sont interconnectées au travers d'un réseau de communication.

Le principal problème dans une telle structure réside dans la résolution des problèmes de cohérence dans les différents systèmes de mémoire cache [30].

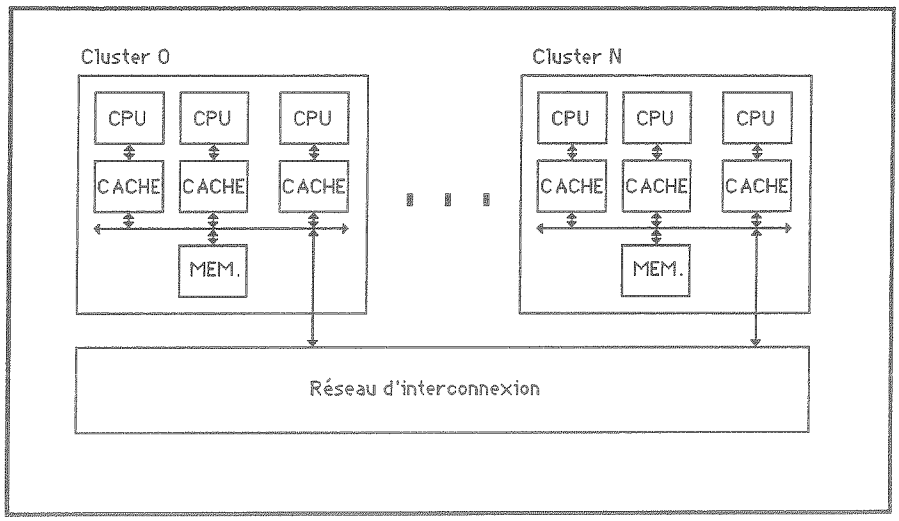


Fig. 19 - Architecture "Snoopy/Directory" de MIPS-X

6. La recherche au niveau mondial

6.1. Une recherche active

Si les fondements de l'approche RISC ont été définis aux Universités de Berkeley et de Stanford, rapidement de très nombreux travaux de recherche ont ensuite vu le jour. Citons, par exemple, les projets PIPE (*Parallel Instructions and Pipelined Execution*) de l'Université du Wisconsin [31], RIMMS (*Reduced Instruction Set Architecture for Multi-Microprocessor System*) de l'Université de Reading [32], ou encore la standardisation d'un jeu d'instructions CORE-MIPS à l'Université de Carnegie-Melon pour le DoD (*U.S. Department of Defense*) [33].

En continuant à réduire le nombre d'instructions, D. Tabak et son équipe ont conçu une architecture caractérisée par une unique instruction, baptisée SIC (*Single Instruction Computer*) [34]. La seule instruction du processeur SIC est une opération "MOVE", du fait de sa très forte occurrence dans un grand nombre de programmes. Un prototype du système SIC, appellé CMOVE, fut réalisé à l'Université de Ben Guréon en 1982 [35] pour des applications de contrôles spécifiques. Les principales caractéristiques de l'architecture CMOVE sont les suivantes :

1) le système n'a qu'une seule instruction sans code opération ; elle est de la forme : <MOVE From Source, To Destination>, où la "Source" représente l'adresse d'un mot source en mémoire et "Destination" l'adresse d'un mot destination en mémoire,

2) l'instruction est exécutée en deux cycles machines,

3) le processeur ne contient pas d'Unité Arithmétique et Logique ; ces opérations sont réalisées par des circuits externes adressables au moyen de l'instruction MOVE.

Une autre architecture originale a été proposée par Phil Koopman, qui définit le concept de machine WISC (*Writable Instruction Set Computer*) [36]. Le principe consiste à tirer parti à la fois des

avantages des architectures RISC et CISC, en donnant aux utilisateurs la possibilité de définir leur propre jeu d'instructions.

Un des objectifs fondamentaux de la recherche sur les architectures RISC est d'augmenter la performance des systèmes informatiques. Une synthèse des travaux actuels conduit à déterminer deux grandes orientations :

1) le premier thème de recherche a pour objectif l'intégration d'un processeur RISC dans une technologie très rapide comme l'ECL ou l'AsGa. Citons également, par exemple, la société japonaise HITACHI qui a réalisé, en 1989, un processeur 4 bits à jonctions Josephson capable de traiter 250 MIPS à une fréquence de 0,76 Ghz [37],

2) le second thème de recherche concerne l'étude du parallélisme, et cela à tous les niveaux : du microparallélisme entre opérateurs, jusqu'au macroparallélisme mettant en oeuvre plusieurs dizaines (voire centaines ...) de processeurs.

Les paragraphes suivants décrivent quelques uns des projets les plus avancés dans ces différents domaines.

6.2. Le processeur AsGa de Texas Instruments et Control Data

L'avancée récente de la technologie Arséniure de Gallium autorise à présent des possibilités d'intégration dans une technologie cinq à sept fois plus rapide que la meilleure filière de silicium, consommant moitié moins que l'ECL et plus tolérante aux radiations.

A la fin de l'année 1986, les sociétés Texas Instruments et Control Data débutaient conjointement la réalisation d'un processeur RISC AsGa pour l'agence américaine DARPA (*Defense Advanced Research Projects Agency*) [38-39]. Le circuit fut réalisé dans une technologie AsGa du type bipolaire à hétérojonctions avec des traits de 1.5 microns, pour une complexité estimée à 12000 portes. L'objectif principal était d'obtenir une puissance de traitement crête de 200 MIPS à 200 Mhz sur la base d'une architecture RISC issue des travaux de Berkeley et de Stanford. Outre l'unité centrale, le système développé intègre deux coprocesseurs de calcul flottant,

deux unités de gestion de la mémoire et deux caches. La conception du jeu d'instructions est largement inspirée par la machine MIPS, mais adaptée pour un modèle d'exécution pipeline de six étages spécifiquement étudié pour tirer parti de mémoires à accès pipelinés. Une seconde différence importante avec l'architecture MIPS réside dans la résolution par le matériel des problèmes d'interblocages du pipeline, au lieu d'être reporté au niveau des compilateurs.

Le chemin des données du processeur est de 32 bits et la longueur du chemin critique est d'environ trente portes logiques. Pour atteindre l'objectif visé d'une fréquence de fonctionnement à 200 Mhz, des retards de portes d'au maximum 160 ps sont à respecter afin de garantir un temps de cycle de 5 ns. Des données récentes semblent indiquer qu'un retard nominal de 250 ps a pu seulement être obtenu, limitant ainsi la fréquence de fonctionnement à 133 Mhz. En outre, la puissance d'une telle architecture est fortement réduite par l'insertion systématique de deux "NOPs" après chaque instruction de branchement et par la bande passante limitée de la mémoire. Ainsi, les simulations démontrent qu'une puissance efficace de 91 MIPS ne peut être dépassée sans une remise en question de l'architecture.

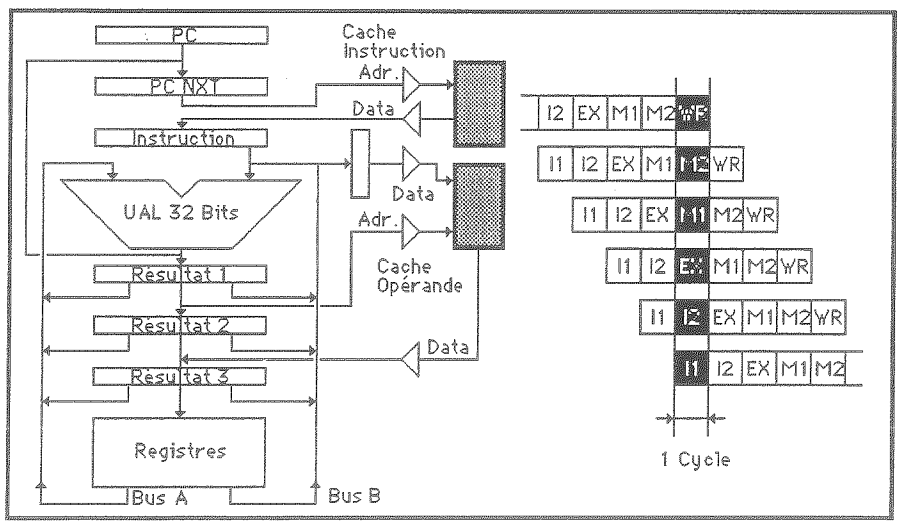


Fig. 20 - Synoptique et pipeline du processeur AsGa T.I.

6.3. Le processeur AsGa de McDonnell Douglas

La Société Mc Donnell Douglas développe également un processeur RISC en AsGa basé sur les travaux de Stanford [40-41]. Due aux spécificités de la technologie employée, les chercheurs de McDonnell Douglas ont modifié la conception initiale : la complexité du processeur est réduite tant au niveau du nombre de transistors nécessaires que des bus internes : la partie contrôle et le nombre d'étages du pipeline sont également réduits. Le système final comprend une unité centrale 32 bits, deux coprocesseurs de calculs, un contrôleur et une mémoire externe. L'ensemble est intégré dans une technologie Arséniure de Gallium du type *Enhancement Mode Junction FET* (E-JFET).

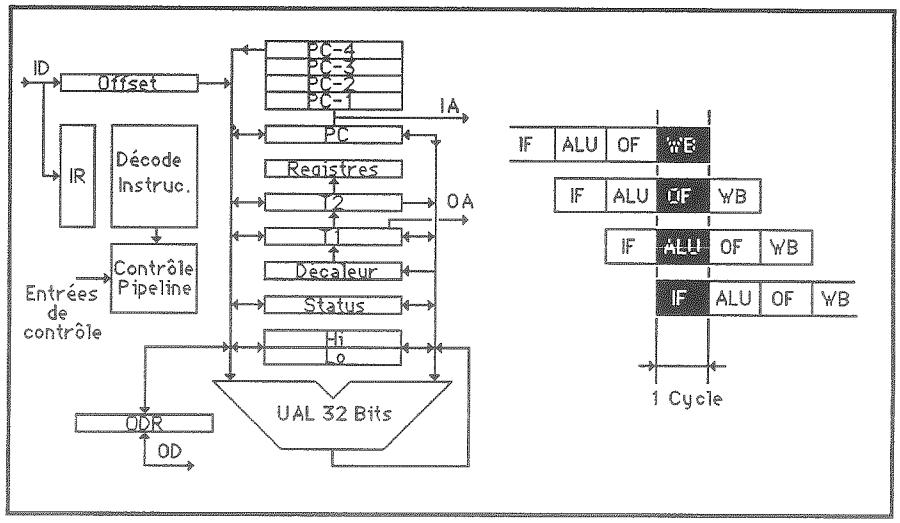


Fig. 21 - Synoptique et pipeline du processeur AsGa McD. Douglas

Le jeu d'instructions est similaire au processeur MIPS de Stanford, auquel sont adjointes les instructions de calcul virgule flottante. Toutes les instructions arithmétiques et logiques s'exécutent en un seul cycle et opèrent entre registres, alors que les instructions de calcul en virgule flottante durent plusieurs cycles. Le compilateur, comme pour MIPS, est partie intégrante de l'architecture. Il réalise les tâches d'optimisation globale, d'allocation des registres et de réorganisation du code.

Le processeur est intégré sur un circuit unique comprenant environ 10000 portes équivalentes. Le pipeline d'exécution n'est que de quatre étages au lieu de cinq pour MIPS : le premier étage effecue la lecture de l'instruction (*Instruction Fetch*), le second exécute l'opération sur l'Unité Arithmétique et Logique (ALU), le troisième réalise la lecture des opérandes (*Operand Fetch*) et le dernier écrit le résultat dans le registre destination (*Write Back*). Le processeur comprend 32 registres dont seulement 16 sont d'usage général. Huit registres sont des registres généraux optionnels et les huit restants sont dédiés à certaines tâches de contrôle ou d'exécution.

Le coprocesseur flottant est réalisé sur la base de la norme IEEE754. Il effectue la lecture des opérations en même temps que le processeur principal et les exécute en parallèle. Le coprocesseur est d'une complexité évaluée à 6000 portes équivalentes. Le troisième circuit comprend l'ensemble de la logique nécessaire au processeur principal et aux coprocesseurs, comme la prise en compte, le décodage et le masquage éventuel des interruptions. La génération des signaux d'horloges et les fonctions d'entrées/sorties sont également incluses dans le circuit contrôleur.

6.4. Le processeur AsGa de RCA

La réalisation d'un processeur AsGa en plusieurs circuits n'est pas la méthode la plus appropriée à cause des délais de transfert des signaux. D'un autre côté, la technologie actuelle n'autorise qu'un nombre restreint de transistors pour un circuit unique. Du fait de ces limitations, la société RCA a réalisé un système composé de seulement trois composants : l'unité centrale, un coprocesseur de calcul virgule flottante et un contrôleur [42-43].

Dans un processeur en Arséniure de Gallium, la lecture d'une instruction est théoriquement supérieure au temps nécessaire pour traverser le chemin interne des données. Ainsi, une amélioration du parallélisme pour l'accès à la mémoire et un accroissement du temps de traversée du chemin des données sont utilisés dans le processeur pour réduire les inconvénients de la phase de lecture (*fetch*). Le système RCA complet est pipeliné, réduisant ainsi les erreurs dues aux problèmes de calage des horloges. Le chemin des données est simplifié grâce à l'utilisation d'un additionneur et d'un décaleur-masqueur simplifiés.

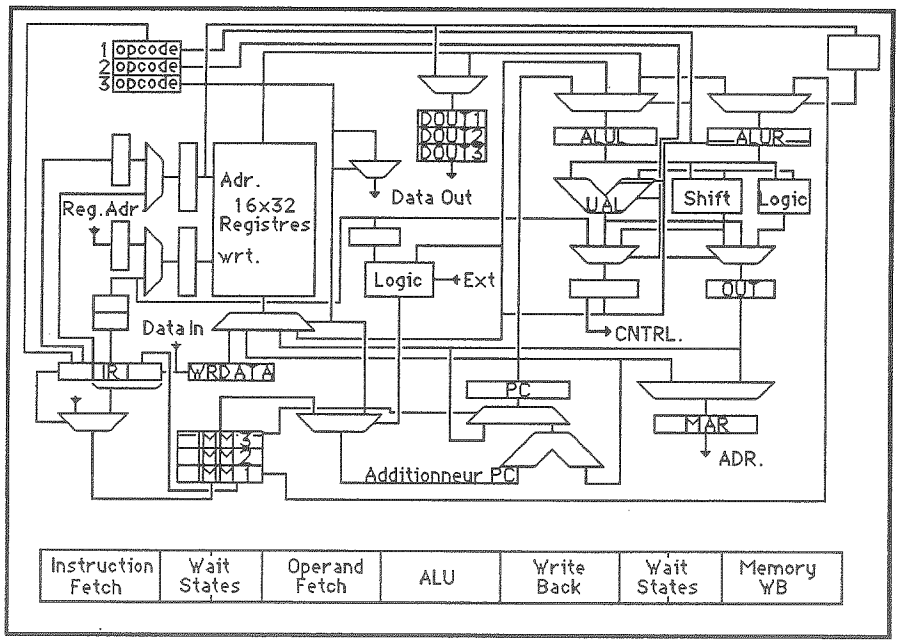


Fig. 22 - Synoptique et pipeline du processeur AsGa RCA

La conception du circuit fut précédée par un nombre important de travaux expérimentaux en collaboration avec l'Université de Purdue, afin de déterminer les choix fondamentaux de l'architecture. Ainsi, pour le pipeline d'exécution, un simulateur fut réalisé permettant de simuler différentes configurations possibles. Finalement, un pipeline à cinq étages fut choisi avec l'adjonction possible d'états d'attentes (*wait-states*). Comme pour MIPS, une attention particulière fut portée au niveau des compilateurs.

6.5. Le processeur AsGa de SODIMA

La Société SODIMA, en collaboration avec l'Université d'Orsay (Institut d'Electronique Fondamentale), développe une seconde génération de processeurs, basée sur la machine virtuelle KIM (le processeur KIM20 sera étudié en détail au cours du chapitre 4). Cette nouvelle version a pour objectif une amélioration importante des performances, ainsi qu'une meilleure adéquation aux contraintes opérationnelles des applications. Pour cela, une des filières retenue est une technologie précaractérisée Arséniure de

Gallium. La technologie AsGa au niveau ASIC, offre maintenant la possibilité d'intégrer près de 20000 portes équivalentes pour des fréquences allant de 300 Mhz à 3 Ghz.

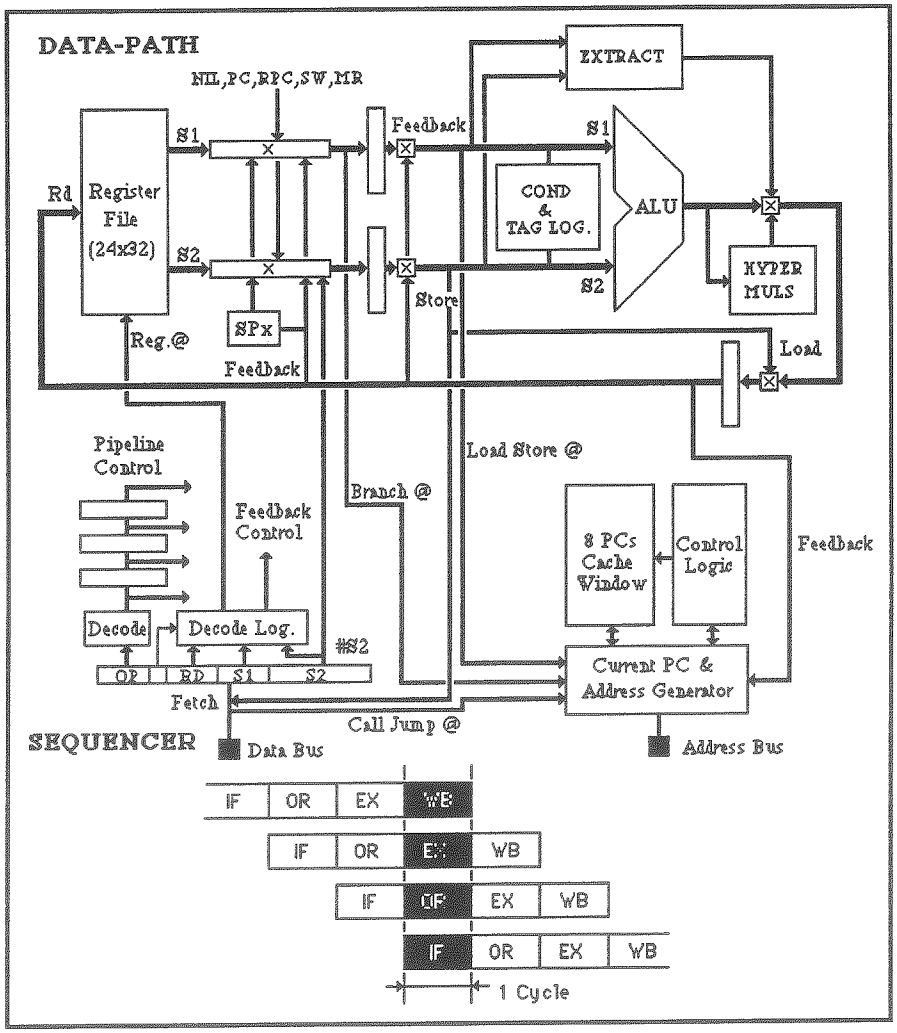


Fig. 23 - Synoptique et pipeline du processeur AsGa SODIMA

Le processeur KIM200 hérite directement des travaux antérieurs [44]. Cependant, des modifications sensibles de l'architecture sont effectuées pour tenir compte :

1) de la réduction de complexité nécessaire pour assurer une fabrication de bonne qualité, cette réduction vise à passer de 17000 (KIM20) à 10000 portes équivalentes,

2) une meilleure réponse aux contraintes temps réel des applications opérationnelles, en particulier au niveau des commutations de contexte et des temps de réponse suite à une interruption,

3) un élargissement du spectre d'application possible, par un support efficace des langages C et Ada.

Une modification architecturale majeure réside dans le choix d'une structure pipeline synchrone à quatre étages, contre trois pour la version CMOS. Ce dernier est composé d'un premier étage de *fetch*, un second de décodage et de lectures des opérandes, un troisième d'exécution de l'opération sur l'Unité Arithmétique et Logique et un quatrième d'écriture du résultat. La fréquence de fonctionnement visée est de 200 Mhz, ce qui correspond à un temps de cycle de 5 nanosecondes. Plus précisément, les caractéristiques attendues du processeur sont les suivantes :

- puissance de traitement efficace de 100 Mips (200 Mips crête),
- architecture RISC 32 bits dotée d'un espace d'adressage de 4 Giga-octets,
- supports spécifiques pour le traitement symbolique (architecture *tagged-RISC*),
- supports spécifiques pour la détection de pannes (*fault-tolerant system*),
- exécution efficace des langages Lisp, Prolog, C et Ada,
- temps de commutation de contexte inférieur à 1 microseconde,
- fabrication en conformité avec la norme MIL-STD-883, avec une résistance aux radiations testée à 100 MRAD en dose cumulée.

Une attention particulière est donnée au système de mémoire cache, indispensable pour tirer parti de la performance potentielle d'une telle architecture. Ainsi, un système de mémoire cache à deux niveaux est étudié, composé d'une petite mémoire très rapide (4 Kmots à 3 ns de temps d'accès) en Arséniure de Gallium, suivi d'un cache CMOS beaucoup plus conséquent mais moins rapide (128 Kmots à 25 ns). Les diverses simulations menées sur cette architecture ont mis en évidence une puissance efficace d'environ

100 MIPS avec des pointes de 200 MIPS pour les traitements effectués dans le premier niveau de cache [45].

6.6. Les architectures à mots d'instructions très larges

Nous avons déjà évoqué l'importance de la recherche sur le parallélisme au travers des projets SPUR de Berkeley et MIPS-X de Stanford. De très nombreux travaux sont menés pour tenter de mettre en oeuvre plusieurs dizaines de processeurs autour d'un système de communication. Sur ce dernier point, les réalisations vont du simple bus parallèle avec mémoire commune aux réseaux de communications du type hypercube [46] ou autres.

Un autre axe de recherche prometteur a pour objectif de dépasser la frontière de l'exécution d'une instruction par cycle. Dans un processeur RISC, le pipeline permet d'utiliser l'ensemble des fonctionnalités de l'architecture en exécutant en parallèle la lecture de l'instruction, son décodage, son exécution et l'écriture du résultat. Ainsi, il est possible d'obtenir pratiquement l'exécution d'une instruction à chaque cycle machine. L'idée qui vient naturellement ensuite consiste à lire simultanément plusieurs instructions et de les affecter à plusieurs unités d'exécutions par l'intermédiaire de multiples chemins de données.

Ce type d'architecture a été baptisé VLIW (*Very Long Instruction Word*) du fait de la présence d'un mot de contrôle du processeur souvent très large (500 bits sur la machine ELI-512 [47]). Plus formellement, une architecture VLIW est caractérisée par les propriétés suivantes :

1) elle possède une unité de contrôle unique générant, à chaque cycle, un mot de commande large,

2) chaque instruction longue est constituée de plusieurs opérations indépendantes couplées,

3) chaque opération peut être pipelinée et demande un petit nombre de cycles pour être exécutée.

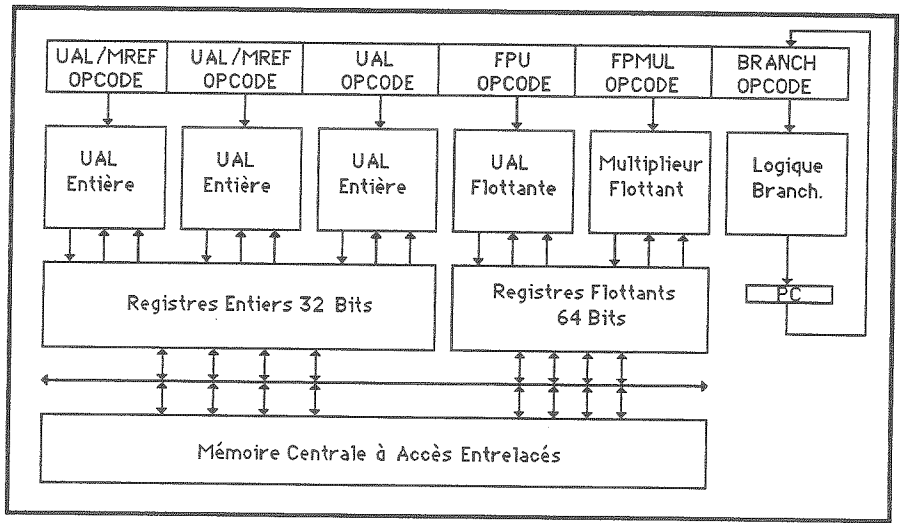


Fig. 24 - Architecture à mots d'instruction très larges (VLIW)

L'ensemble de ces propriétés permet de distinguer une machine VLIW des machines multiprocesseurs *data-flow*, ou des processeurs vectoriels. Historiquement, les premiers projets de machine VLIW sont largement inspirés par l'architecture du CDC 66000, du Cray-1, de l'IBM 360/91 et du processeur MIPS de Stanford. Comme pour ce dernier, une place prépondérante est laissée au compilateur qui devient très complexe pour tirer parti de l'architecture matérielle mise à sa disposition. La plupart des travaux dans ce domaine visent à appliquer ou à améliorer la technique de *trace-scheduling* qui permet de générer un code exécutable "parallélisé" [48]. Les performances espérées grâce à de telles architectures vont de 10 à 30 fois celles mesurées sur un processeur conventionnel, ce qui correspond globalement à l'exécution de 10 à 30 instructions RISC équivalentes par cycle.

D'autres axes de recherches voisins, baptisés "processeurs super-scalaire" ou "processeurs super-pipeline" visent des résultats analogues [49].

Si ces architectures à mots d'instructions très larges sont prometteuses, le prix à payer pour la performance semble être un accroissement notable de la complexité des processeurs et des techniques de compilation associées. Néanmoins, elles représentent certainement l'avenir des architectures RISC.

7. RISC versus CISC

Finalement, qu'est-ce qui différencie un processeur RISC d'un processeur CISC ? Comme nous l'avons vu, le terme RISC se réfère à une philosophie de conception plutôt qu'à un ensemble de caractéristiques architecturales. Il n'est probablement pas raisonnable de classer arbitrairement un processeur dans une catégorie à la seule vue du nombre de ses instructions ou de ses modes d'adressages. Surtout à une époque où les fabricants de composants avouent, comme par exemple Motorola pour le MC68030 [50], tirer parti des deux écoles.

Néanmoins, une différence importante est la définition d'une méthodologie de conception cohérente qui reste l'apanage de RISC. Une autre différence souvent invoquée lie les architectures RISC à des applications particulières alors que les processeurs RISC sont conçus pour être d'usage général. Nous verrons au cours du prochain chapitre que cette différence est fortement liée à la précédente. En effet, l'approche descendante sous-jacente à la méthodologie RISC est d'autant plus performante si le domaine d'application est bien cerné.

L'approche RISC peut très bien ête résumée par l'équation définissant la performance (Chap. 1 § 1.4.). Alors qu'une architecture CISC tente principalement de réduire le nombre d'instructions (I), pour augmenter la performance (P), l'architecture RISC vise à diminuer à la fois le nombre moyen de cycles par instructions (C) et le temps de cycle (1/S).

2

Les principes fondamentaux

1. La méthodologie RISC

1.1. Les différentes approches de la conception

Il existe plusieurs approches pour concevoir l'architecture d'un processeur. La plupart des processeurs CISC du commerce ont été conçus selon une approche ascendante (*bottum-up approach*) pour respecter les contraintes de compatibilité avec les versions antérieures, mais également pour minimiser la difficulté inhérente à la réalisation d'une nouvelle architecture. Dans ce modèle, le concepteur part d'une architecture matérielle pré-établie, à laquelle il adjoint l'ensemble des mécanismes nécessaires à l'exécution du langage de haut niveau ou de l'application. La seconde approche, appelée "médiane" (*middle approach*) a surtout été utilisée pour la réalisation de machines dédiées à l'exécution d'un langage particulier, comme par exemple la machine Lisp de Symbolics [51]. Dans ce cas, le concepteur n'a pas à concevoir une machine d'usage général. Il se restreint donc au langage de haut niveau choisi et, à partir des types de données et des opérateurs associés, il tente de

synthétiser une architecture matérielle efficace pour son exécution. La troisième approche possible est appelée descendante (*top-down approach*) puisqu'elle débute par l'analyse des opérations à forte occurrence pour ensuite déterminer l'architecture adaptée. La figure 25 donne une comparaison de ces différentes approches sous la forme d'un synoptique mettant en évidence les différents niveaux d'abstractions possibles.

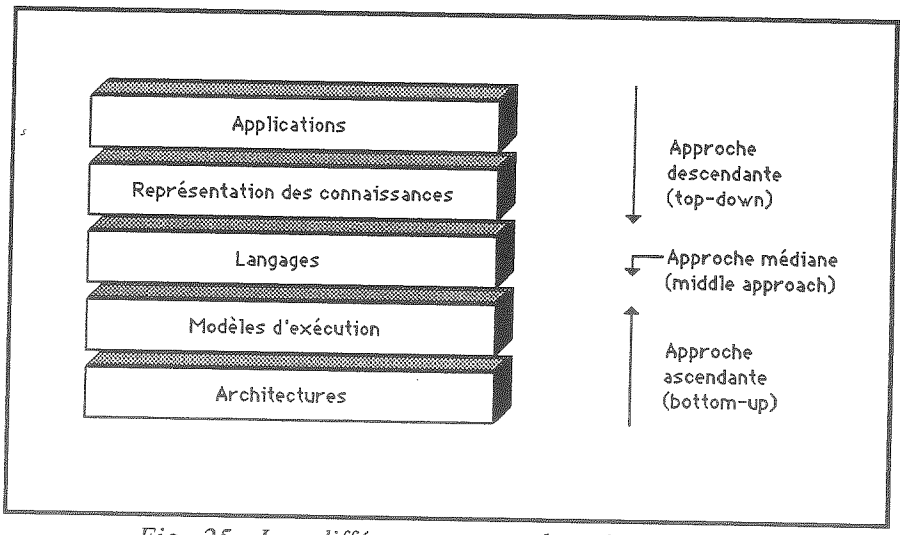


Fig. 25- Les différentes approches de conception

Il est bien évident que, bien souvent, le concepteur ne peut dans les faits se limiter à une seule approche. Par exemple, une conception descendante doit tenir compte des architectures existantes et des contraintes de réalisation. Ceci peut entraîner un certain nombre d'itérations au cours de la conception, jusqu'à l'obtention de l'architecture jugée satisfaisante.

1.2. Une approche descendante

Parmi les approches évoquées au paragraphe précédent, la méthodologie RISC emprunte une démarche purement descendante, mais guidée par un certain nombre de consignes au niveau de l'architecture matérielle finale. L'objectif fondamental réside dans la réduction du fossé sémantique entre les mécanismes à forte occurrence de l'application visée (ou du langage ciblé) et l'architecture électronique susceptible de les exécuter. Cette idée, en

pure conformité avec une méthodologie descendante, passe par une analyse rigoureuse des opérations exécutées, à partir d'un échantillon de programmes jugés significatifs et d'analyses statistiques effectuées sur les codes générés. Les opérations à fortes occurrences sont ensuite étudiées finement afin d'en déduire les opérateurs matériels susceptibles de les exécuter. Les opérateurs sélectionnés sont alors intégrés au sein d'une architecture régulière, qui doit permettre de profiter au maximum du microparallélisme possible entre les opérateurs. De ce fait, les structures *pipeline* et la technique de *scoreboarding* sont largement employées.

Initialement, la recherche menée sur le concept RISC vise à tirer parti du développement des technologies d'intégration VLSI pour concevoir des processeurs efficaces, compacts et fiables. Par conséquent, ce sont également les contraintes d'intégration (technologie, bibliothèque de fonctions, prédiffusé, précaractérisé, *custom* ou circuit mixte, etc.) ainsi que les outils logiciels de conception (CAO) qui vont majoritairement guider le concepteur vers les choix d'implantations matériels finaux. Par exemple, un banc de registres peut, selon les cas, être réalisé avec des bascules ou une mémoire. Certains constructeurs proposent également des "compilateurs" de banc de registres particulièrement optimisés.

1.3. Définition de la méthodologie

La philosophie RISC est basée sur une approche que nous résumerons par : une méthodologie de conception descendante guidée par un modèle architectural. Notons au passage que cette approche semble parfaitement convenir à une "compilation de silicium" sous la forme, pourquoi pas, d'un compilateur de processeurs RISC. Nous pouvons résumer la méthodologie par les quatre principaux points suivants :

1) analyse des applications ciblées pour déterminer les opérations à très forte occurrence,

2) optimiser l'architecture du chemin des données (*data-path*) et des opérateurs associés pour exécuter le plus rapidement possible ces opérations,

3) inclure d'autres instructions uniquement si elles ne remettent pas en question la structure établie (en brisant la régularité du

schéma d'exécution, ou en rajoutant trop de matériel), ne ralentissent pas l'exécution et sont relativement fréquentes,

4) reporter autant que possible, les fonctionnalités complexes au niveau du compilateur, pour garantir la simplicité et la régularité de l'architecture.

Ce dernier point vise en fait à reporter au niveau de la compilation les mécanismes complexes afin d'obtenir une architecture *run-time* particulièrement efficace. Nous avons parlé "d'approche descendante guidée". Voici maintenant les principales recommandations qui permettent de déterminer l'architecture finale en conformité avec la méthodologie RISC.

1.4. Les consignes architecturales

Les consignes architecturales sont le complément indispensable à l'analyse descendante. Elles permettent de guider le concepteur dans ses choix. Ces consignes sont au nombre de neuf :

1) Les instructions doivent être en nombre réduit et s'exécuter en un seul cycle machine,

2) les accès à la mémoire sont limités à deux instructions, baptisées *load* et *store*,

3) le décodage des instructions est câblé plutôt que microprogrammé,

4) le jeu d'instructions doit reposer sur un format fixe (généralement un mot machine de 32 bits) avec des modes d'adressages simples,

5) toute opération complexe doit être rejetée au niveau du compilateur,

6) l'ensemble de l'architecture doit au maximum tirer parti du modèle de pipeline déterminé,

7) le processeur doit posséder un large banc de registres internes, avec ou sans la technique de fenêtrage et de chevauchement,

8) le processeur doit être particulièrement adapté à l'adjonction de systèmes de mémoire cache et de coprocesseurs,

9) le jeu d'instructions est généralement conçu pour un domaine d'applications parfaitement déterminé.

Beaucoup de personnes pensent que ces consignes sont une conséquence des travaux menés sur RISC. Il n'en est rien puisque, pris séparément, ils ont tous été utilisés sur des processeurs CISC bien avant l'avènement de RISC. Rappelons encore une fois que la caractéristique déterminante dans RISC est l'apparition d'une méthodologie cohérente.

En particulier, le point (9) nous rappelle que la conception d'un processeur RISC est parfaitement ciblé vers un type d'application précis, pour lequel le jeu d'instructions est optimisé. Au contraire, un processeur CISC est généralement conçu pour un large éventail d'applications, en incluant des supports pour différents types d'environnement et de langages. Ce point est bien évidemment sujet à de nombreuses controverses justifiées, puisque la plupart des machines dédiées à un langage sont encore basées sur une structure CISC. D'autre part, il est difficile de cerner un domaine d'application précis lorsque, par exemple, le langage de haut niveau ciblé est C, langage d'usage général par excellence.

Dans les paragraphes suivants, nous allons reprendre chacun de ces points au travers de l'examen des deux tâches principales qui forment la conception d'un processeur RISC : (1) la définition du jeu d'instructions et (2) la définition de l'architecture pipeline qui va l'exécuter.

2. Un jeu d'instructions réduit et homogène

2.1. Détermination des catégories d'instructions

Nous avons insisté sur le fait que le jeu d'instructions doit être déterminé d'après une étude minutieuse des opérations à forte

occurrence, pour un domaine d'application parfaitement défini. La réalisation effective du jeu d'instructions est également guidée par la détermination des classes d'instructions qui doivent être également en petit nombre. Pour tout processeur RISC nous retrouverons trois catégories d'instructions indispensables :

1) les instructions arithmétiques et logiques,

2) les instructions de contrôle du séquencement,

3) les instructions d'accès à la mémoire.

Ces trois classes forment le coeur de tout jeu d'instructions RISC, auquel peuvent être adjointes une ou plusieurs autres classes plus spécifiquement dédiées à l'application visée. Ainsi, au cours de la description du processeur symbolique KIM, nous verrons une classe d'instructions spécialement adaptée au traitement de listes. Parfois, il existe également une classe d'instructions spéciales dans laquelle sont regroupées certaines fonctions spécifiques de contrôle, tel que l'accès au mot d'état ou tout autre registre spécialisé.

Rares sont les concepteurs qui partent de zéro pour la définition du jeu d'instructions. Il est vrai qu'il est fastidieux et probablement inutile de recommencer les travaux d'analyse en ce qui concerne les opérations arithmétiques et logiques. Ainsi, la plupart des concepteurs se basent directement sur les travaux de Berkeley et de Stanford pour déterminer les instructions arithmétiques et logiques. Les différences d'un processeur à l'autre proviennent ensuite des spécificités du domaine d'application visé, de l'intégration ou non d'un opérateur de calcul virgule flottante.

Un travail de réflexion et une étude plus approfondie sont nécessaires pour les instructions de contrôle. Dans ce cas, les analyses d'occurrences sont fondamentales pour déterminer les mécanismes adéquats. Néanmoins, quelque soit le langage ou l'application visée, nous retrouvons généralement explicitement les instructions *call* et *return* ou implicitement sous la forme d'un *jump-and-link*, les branchements retardés (*delayed branch*) conditionnels et inconditionnels.

Enfin, les instructions d'accès à la mémoire se résument soit à deux opérations *load* et *store* qui permettent respectivement de lire et de stocker un mot de 32 bits, soit plusieurs versions des mêmes

instructions autorisant le transfert de double-mots, mots, demi-mots et octets. Dans le premier cas, deux instructions d'insertion (*insert*) et d'extraction (*extract*) viennent compléter les opérations logiques pour permettre une manipulation efficace des octets.

La figure 26 donne un jeu d'instructions RISC "standard", tel qu'il a été imaginé par Thomas Gross et Robert Firth à l'Université de Carnegie-Mellon aux U.S.A. [52]. Initialement inspiré de la machine MIPS, il est aujourd'hui recommandé par le DoD aux Etats-Unis.

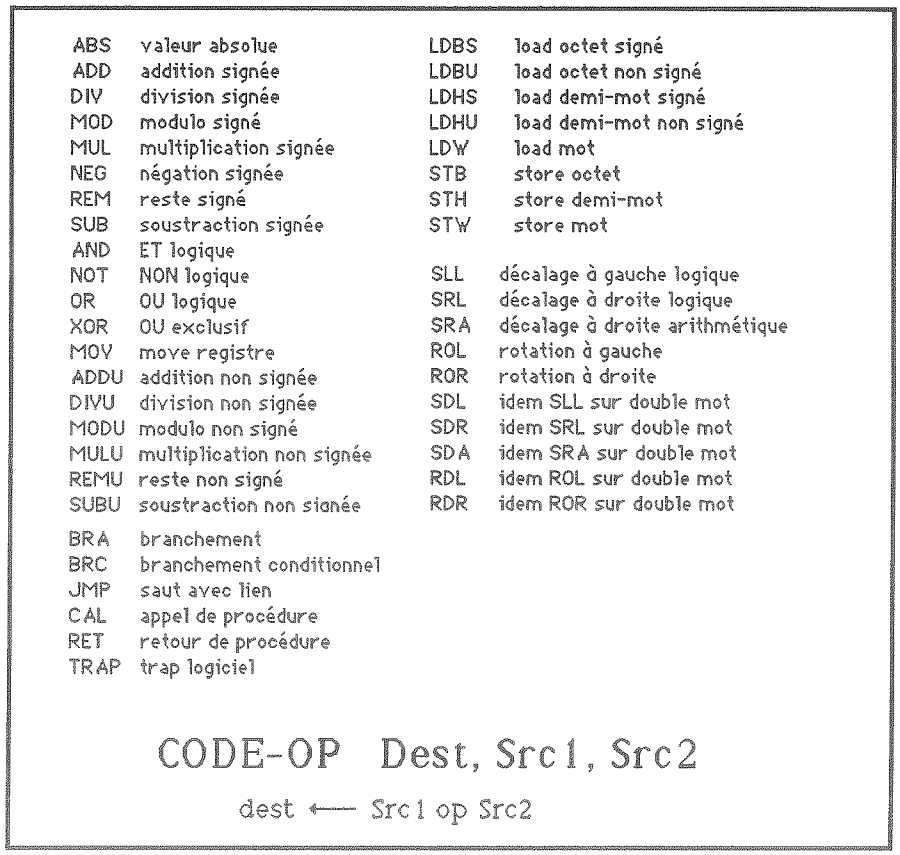


Fig. 26 - Le jeu d'instructions RISC "CORE-MIPS"

2.2. Les formats d'instructions

Une des consignes fondamentales de la méthodologie RISC est d'adopter un format fixe pour l'ensemble des instructions. La taille généralement retenue est équivalente à la taille d'un mot mémoire, c'est-à-dire dans la plupart des cas 32 bits. Ensuite, il reste à déterminer sur la base de cette taille fixe, le nombre exact et la composition des formats d'instructions. La majorité des "pures" architectures RISC est caractérisée par un petit nombre de formats et de modes d'adressage : de 1 à 4 suivant les processeurs. Par exemple, les processeurs conçus à l'Université de Berkeley sont basés sur deux formats principaux :

1) un format à trois opérandes permettant le codage de la majorité des instructions, dont les opérations arithmétiques et logiques. Dans ce format, outre l'inévitable code opération, nous retrouvons : l'adresse d'un premier registre source et une seconde opérande source. Cette dernière peut être alternativement un second registre ou une valeur immédiate.

2) Un format pour les instructions de contrôle. Après le code opération nous y retrouvons généralement une ou deux opérandes permettant la génération d'une adresse de branchement immédiate, relative ou indexée.

2.3. Un jeu d'instructions orienté registres

La méthodologie RISC est basée sur une utilisation intensive des registres internes du processeur. La raison en est évidente. Le goulet d'étranglement pour un processeur réside dans les accès à la mémoire, car généralement le temps d'accès à une mémoire "bon marché" est supérieur à celui de l'exécution d'une opération arithmétique ou logique classique (de 100 à 200 nanosecondes contre 20 à 50 nanosecondes pour une technologie CMOS "standard").

Un accroissement sensible des performances peut donc être obtenu si l'on arrive à limiter les accès à la mémoire externe. Cet objectif est atteint en augmentant le nombre de registres internes et en limitant explicitement les accès externes.

Le temps d'accès à un registre interne est du même ordre de grandeur qu'une opération arithmétique ou logique, il est donc très intéressant d'effectuer la majorité des opérations "en interne". De ce fait, la majorité des instructions d'une architecture RISC effectue la lecture de deux opérandes sources dans le banc de registres, exécute l'opération sur l'Unité Arithmétique et Logique, puis range le résultat dans un registre destination. La complexité relativement faible d'un processeur RISC permet en outre l'intégration d'un nombre important de registres (de 32 à plus d'une centaine) permettant ainsi le stockage de nombreuses valeurs, variables locales ou globales.

La limitation explicite des accès à la mémoire externe est obtenu en autorisant uniquement ces accès par deux instructions. L'instruction *load* permet de charger un mot de la mémoire vers un registre et l'instruction *store* permet le transfert vers la mémoire. Néanmoins, pour beaucoup de processeurs RISC, les occurrences d'accès à la mémoire (outre les lectures d'instructions *fetch*) représentent entre 20 et 25% du total des instructions exécutées. Nous reviendrons sur ce point très important lors de la discussion sur le choix de l'architecture qui doit supporter l'exécution des instructions.

2.4. Simplicité et régularité

La définition du jeu d'instructions va de pair avec la détermination du modèle d'exécution pipeline. Aucune instruction, du fait de sa complexité, ne doit remettre en question la régularité de son fonctionnement, car toute infraction à cette règle se paye ensuite chèrement au niveau des performances. Selon le nombre d'étages, généralement de 2 à 6, et selon les fonctionnalités attribuées à chacun d'eux, chaque instruction doit coïncider avec le séquencement prévu.

A titre d'exemple, c'est ainsi que les branchements retardés (*delayed-branch*) ont été mis au point. Dans la grande majorité des cas, l'instruction suivante qui doit être exécutée peut être trouvée à l'adresse suivante, juste après l'instruction en cours d'exécution. L'incrément d'une adresse est un calcul rapide qui peut se dérouler sans aucun problème dans le premier temps du pipeline d'exécution, dédié à la lecture de l'instruction (*fetch*). Ainsi, dès le code opération lu, le compteur de programme est automatiquement

incrémenté pour, qu'au cycle suivant, une nouvelle instruction puisse être lue et décodée.

Une instruction de branchement vient généralement rompre ce séquencement. En effet, il n'est généralement pas possible, dans un seul cycle machine, de calculer assez rapidement l'adresse de la prochaine instruction à exécuter. Pour résoudre ce problème et ainsi éviter de "briser" la régularité du pipeline d'exécution, la technique du "branchement retardé" est souvent utilisée. Il s'agit simplement de différer le branchement effectif le temps d'une instruction, à charge ensuite au compilateur-optimiseur d'insérer dans le "slot" laissé disponible une instruction utile (fig. 27).

D'autres exemples, comme les instructions conditionnelles, les instructions *load* et *store*, doivent également trouver des solutions pour permettre un schéma pipeline régulier et, par conséquent, garantir l'exécution d'une instruction à chaque cycle machine. Nous étudierons ces autres cas un peu plus loin dans l'ouvrage.

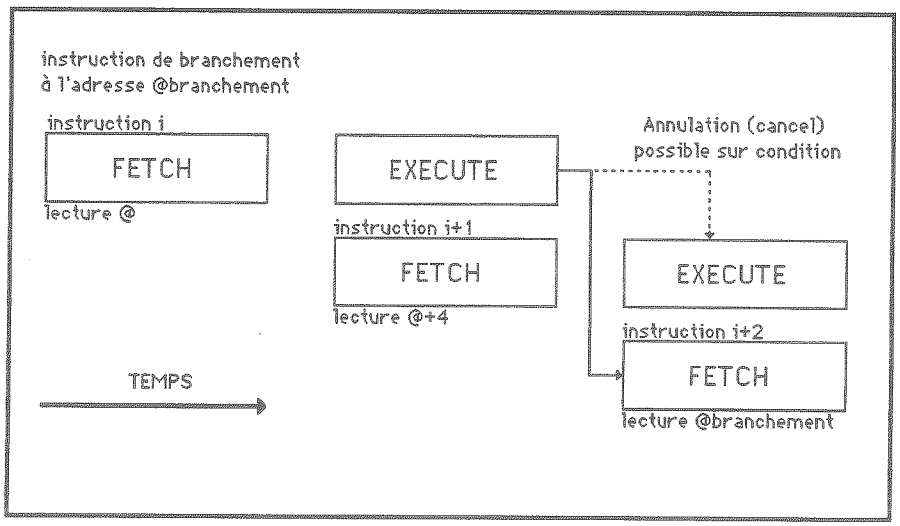


Fig. 27 - Le mécanisme du branchement retardé sur RISC-I

2.5. Un schéma d'exécution direct

Un point fondamental de la méthodologie RISC réside dans la volonté de réduire le fossé existant entre le langage de haut niveau visé (ou le type d'application) et la machine support de son exécution. Pour obtenir ce résultat, un moyen consiste à réduire le nombre de transformations nécessaires avant l'exécution effective par les opérateurs électroniques du processeur. En pratique, cet objectif peut être atteint en câblant les instructions (*hardwired instructions*) plutôt qu'en les microprogrammant. Nous obtenons alors un schéma d'exécution quasi-direct, parfaitement adapté aux techniques modernes de compilation. La figure 28 illustre ce point précis au travers de la comparaison des schémas d'exécution d'une machine microprogrammée et celui utilisé pour la réalisation de la plupart des processeurs RISC.

C'est grâce à la simplicité du jeu d'instructions, due à la fois au nombre restreint de formats et de modes d'adressage, de leur nombre réduit et de leur taille fixe, que le décodage dans un processeur RISC est très simple, (pour ne pas dire trivial). De ce fait, le nombre de "couches logiques" à traverser avant l'exécution effective de l'opération est réduit, permettant ainsi une exécution directe des instructions à chaque cycle, tout en autorisant des temps de cycles très courts.

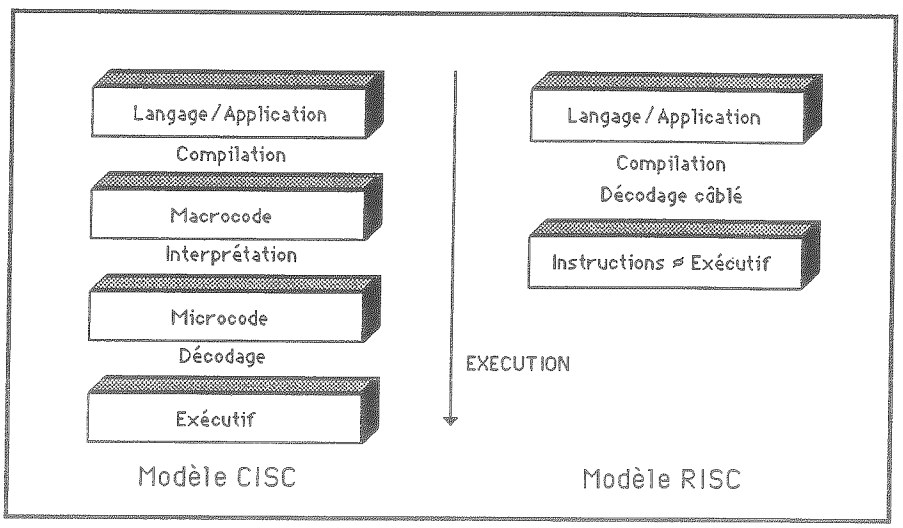


Fig. 28 - Les modèles d'exécution CISC et RISC

Si la détermination des instructions permet de garantir un niveau sémantique correct, les trois paramètres jouant sur la performance sont alors optimisés : nombre d'instructions pour une tâche (I), nombre de cycles par instructions (C), durée d'un cycle (1/S) (voir Chap. 1 § 1.4.). Le jeu d'instructions étant déterminé, avec une bonne idée des contraintes matérielles sous-jacentes, il reste encore à déterminer l'architecture globale qui permettra de les exécuter efficacement. C'est l'objectif des paragraphes qui vont suivre.

3. Une architecture pipeline régulière

3.1. Le chemin des données

Dans le paragraphe précédent, nous avons surtout insisté sur la méthode employée pour déterminer un jeu d'instructions réduit et homogène, dédié à un type d'application. Au cours de ce paragraphe, nous allons aborder les principaux traits architecturaux qui fondent un véritable processeur RISC.

Un processeur RISC peut être décomposé grossièrement en deux parties : (1) le chemin des données comprenant les opérateurs câblés et (2) la partie contrôle, elle-même comprenant généralement une unité de lecture et de décodage des instructions, ainsi qu'un séquenceur chargé de cadencer l'exécution. La structure orientée registre (Chap. 2 § 2.3.) repose sur une architecture organisée autour d'un banc de registres et d'une Unité Arithmétique et Logique. Pour permettre la lecture de deux opérandes sources et l'écriture d'un résultat simultanément, le banc de registres doit posséder deux ports de lecture et un port d'écriture. Les deux ports de lecture sont connectés à l'Unité Arithmétique et Logique, au travers de registres dits *pipelines*, chargés de garantir la synchronisation et le cadencement des opérations dirigées par le séquenceur.

A partir de ce schéma élémentaire, il est facile d'ajouter de nouveaux opérateurs dédiés à des instructions spécifiques. Il sera généralement plus astucieux d'intégrer de nouvelles unités fonctionnelles en parallèle avec l'unité entière, afin de ne pas

cumuler les temps de transfert, ou pour profiter du parallélisme alors possible. C'est, par exemple, la technique employée dans le 88100 et gérée par un mécanisme de *scoreboarding*. Néanmoins, un nouvel opérateur peut également être ajouté en série, pour éviter la duplication d'une fonction facilement générée par l'Unité Arithmétique et Logique, si le temps de transfert total reste inférieur au temps de cycle machine fixé.

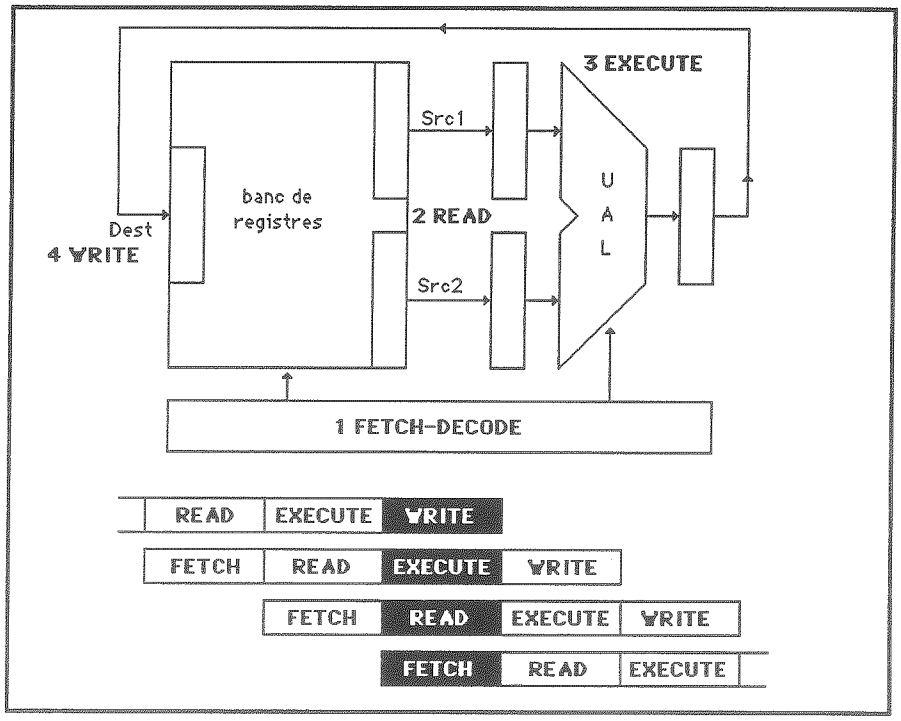


Fig. 29- Synoptique simplifié d'un chemin des données RISC

Ces remarques mettent en évidence les deux "chemins critiques" importants d'un processeur RISC. Ces derniers sont : (1) le temps d'accès à un registre interne et (2) le temps d'exécution d'une addition 32 bits signée sur l'Unité Arithmétique et Logique. Ces deux temps sont "critiques" car ce sont eux qui imposent le temps de cycle final du processeur. Une description de banc de registres et d'Unité Arithmétique et Logique pourra être trouvée par exemple dans [53].

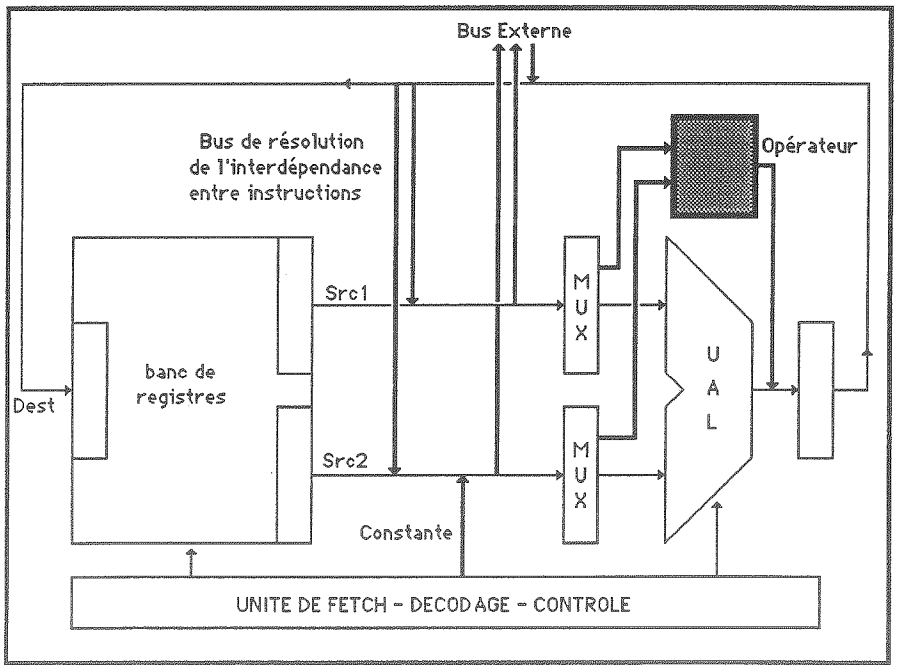


Fig. 30 - Chemin des données modifié

La figure 29 donne le synoptique simplifié d'un chemin des données, mettant en évidence le sens du flot des données en relation avec le pipeline d'exécution. La figure 30 donne ensuite le même chemin des données, dans lequel les problèmes d'interdépendance du pipeline sont résolus par deux bus supplémentaires. Ce problème intervient lorsqu'une instruction utilise pour registre source, le registre destination de l'instruction précédente. Or, un simple regard au schéma du pipeline (figure 29) permet de constater que ce dernier ne sera écrit qu'au cours du dernier cycle. Les deux bus permettent donc "d'intercepter" les valeurs avant leur écriture dans le banc de registre, si une instruction en nécessite l'usage. Nous noterons au passage, que ces deux bus ont été omis dans l'architecture MIPS, le problème d'interdépendance devant être résolu par le compilateur qui doit modifier l'ordre des instructions ou l'affectation des registres. Le synoptique modifié montre également l'arrivé d'une constante en provenance du mot instruction, les bus d'accès externes et l'adjonction d'un opérateur en parallèle avec l'Unité Arithmétique et Logique.

3.2. Les fenêtres de registres à recouvrement partiel

Avant de décrire une unité de lecture-décodage et un séquenceur simplifiés, attardons-nous un moment sur le principe des fenêtres de registres, largement utilisé dans RISC. Rappelons tout d'abord que les appels fonctionnels sont les mécanismes les plus coûteux en temps d'exécution et parmi les plus occurrents. Un appel fonctionnel, outre le branchement proprement-dit, doit sauvegarder le contexte de la procédure appelante (PC, mot d'état et variables locales) et passer les arguments à la fonction appelée. Corrolairement, un retour fonctionnel, en plus du branchement à l'adresse initialement sauvegardée, doit restaurer le contexte de la procédure et passer les résultats. Les analyses menées à l'Université de Berkeley ont montré que la majorité des procédures n'échange au maximum que six paramètres et que la profondeur d'appel maximale n'est supérieure à huit que dans moins de un pour cent des cas (Chap. 1 § 4.1.).

L'objectif du fenêtrage consiste à effectuer l'ensemble des tâches évoquées en un seul cycle machine. Pour ce faire, au lieu d'un banc de registre monolithique, plusieurs "fenêtres" contiguës sont regroupées sous la forme d'une roue. Chaque fenêtre est composée de plusieurs registres locaux (10 dans RISC-II), de registres d'entrées et registres de sorties dédiées aux échanges de données entre procédures (6 entrées et 6 sorties pour RISC-II). Lors de l'exécution d'une instruction *call*, la roue de fenêtres se déplace pour présenter un nouveau banc de registres locaux. Simultanément, le contexte de la procédure appelante a été ainsi sauvegardé. Le décalage des fenêtres n'est que partiel, puisqu'elles se chevauchent chacune de six registres pour permettre le transfert des arguments et des résultats.

Malgré sa faible occurrence théorique, un débordement intervient si le nombre de procédures appelées, sans retour, est supérieur au nombre de fenêtres disponibles dans le processeur. Ce problème est automatiquement détecté par les instructions d'appel et de retour, auquel cas une exception (*trap*) est générée. La fenêtre courante est référencée par un pointeur que nous appelerons CWP (*Current Window Pointer*). La première fenêtre allouée est, quant à elle, référencée par un pointeur que nous appellerons FWP (*First Window Pointer*). Lors de l'exécution d'une instruction d'appel de

procédure (*call* dans RISC-II ou *call/save* dans SPARC), le pointeur CWP est automatiquement incrémenté pour assurer le décalage de la roue de fenêtres. Si, dans ce cas, le pointeur CWP devient égal au pointeur FWP, alors le processeur génère une exception de dépassement de capacité des fenêtres (*window overflow*). Lors de l'exécution d'une instruction de retour (*return* ou *restore*), le pointeur CWP est décrémenté pour restituer le contexte de la procédure appelante et passer les résultats. Si, avant le décalage, le pointeur CWP devient égal au pointeur FWP, alors une autre exception est générée (*window underflow*).

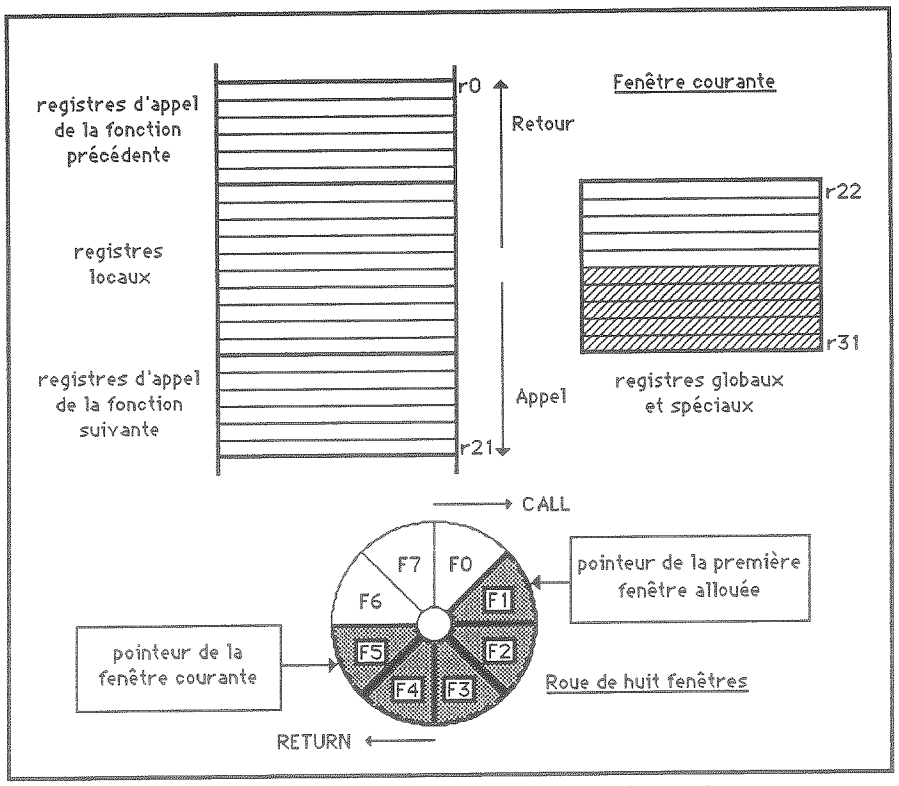


Fig. 31- Le mécanisme des fenêtres de registres

La roue de fenêtres fonctionne en fait comme une mémoire cache de la pile stockant généralement les paramètres d'appels et les résultats (*frame stack*). De ce fait, lors d'un débordement supérieur, une fenêtre peut être libérée en la sauvegardant dans une pile située dans la mémoire centrale. Corollairement, lors d'un débordement inférieur, une fenêtre peut être réimplantée dans la

roue depuis la pile située dans la mémoire centrale. Plusieurs stratégies ont été étudiées pour gérer efficacement ce mécanisme. Néanmoins, il semble que le transfert d'une fenêtre unique à chaque exception soit la plus usitée.

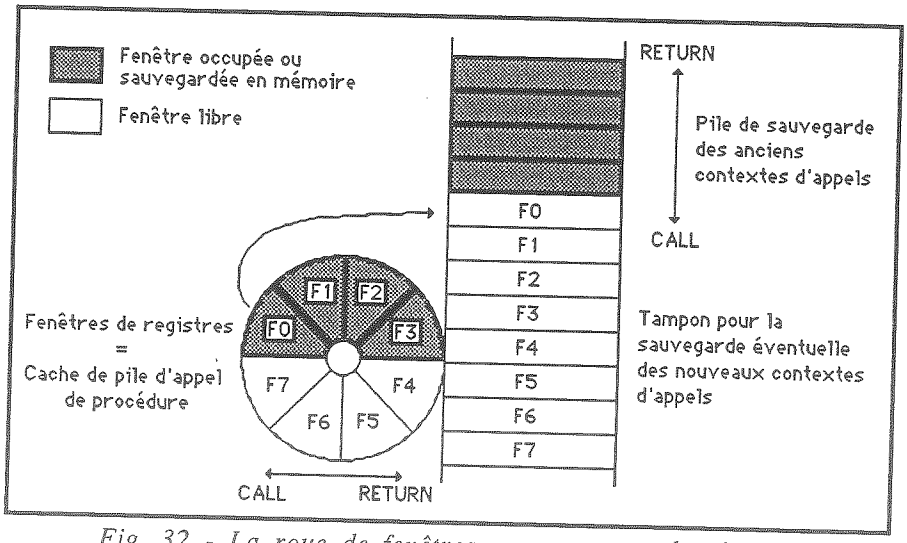


Fig. 32 - La roue de fenêtres comme un cache de pile

3.3. L'unité de lecture et de décodage

Cette partie d'un processeur RISC est chargée de lire l'instruction courante, de la décoder, puis de calculer l'adresse de l'instruction suivante. Du fait de la simplicité des formats d'instructions, le décodage est largement simplifié. Dans un bon nombre de processeurs RISC, il ne revient en fait qu'à stocker directement les différentes opérandes en provenance du registre instruction dans les registres pipelines et à générer un mot de commande à partir du code opération. La figure suivante illustre ce mécanisme au travers du processeur RISC-II de Berkeley.

Le calcul de l'adresse pour la lecture de l'instruction suivante (*fetch*) est également simple, puisque dans la majorité des cas, elle se trouve à l'adresse suivante. Dans ce cas, l'adresse courante est simplement incrémentée (il s'agit en fait d'un ajout de quatre si l'on raisonne au niveau octet).

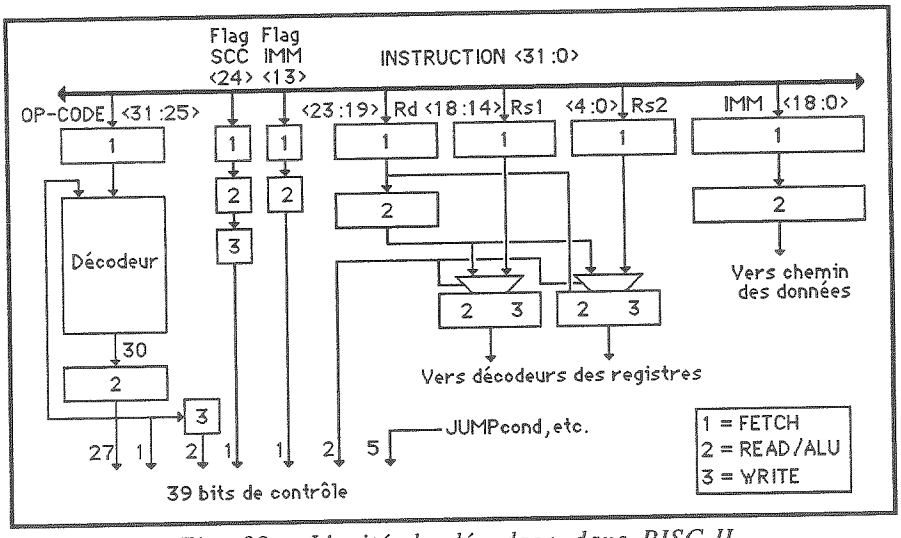


Fig. 33 - L'unité de décodage dans RISC-II

3.4. Le séquenceur et le pipeline d'exécution

Le séquenceur représente le centre nerveux d'un processeur RISC, puisqu'il a pour rôle de synchroniser et de cadencer l'ensemble des opérations nécessaires à son bon fonctionnement. Le séquencement est basé sur un modèle d'exécution pipeline synchrone. Le processeur est en quelque sorte divisé en étages, respectivement commandés par des registres de contrôle contenant "les ordres" à exécuter (voir paragraphe précédent). A chaque front de l'horloge interne, les mots de commandes vont être envoyés aux différents opérateurs, pendant que les suivants vont y être stockés. Une fois le pipeline "amorcé", ce cadencement va permettre de faire fonctionner, à chaque cycle, l'ensemble du processeur (voir les schémas pipeline des processeurs étudiés dans cet ouvrage). Il n'existe pas, à proprement parler, de pipeline idéal. Chaque modèle, à deux, trois ou plus d'étages, comporte des avantages et des inconvénients. Il convient donc au concepteur de choisir celui qui semble le plus adapté pour son application et, par conséquent, au jeu d'instructions établi.

Néanmoins, on peut généralement dire que, plus un pipeline est long, plus l'interface avec la mémoire peut être efficace. En retour,

un pipeline long va créer d'importants problèmes de régularité (de désamorçage en fait) dus aux instructions de branchement. A titre d'exemple, dans un pipeline à trois ou quatre étages, une seule instruction doit être réaffectée par le compilateur (*delay-slot*), alors que pour des pipelines plus importants, deux, voir trois emplacements sont nécessaires pour la réalisation des branchements retardés.

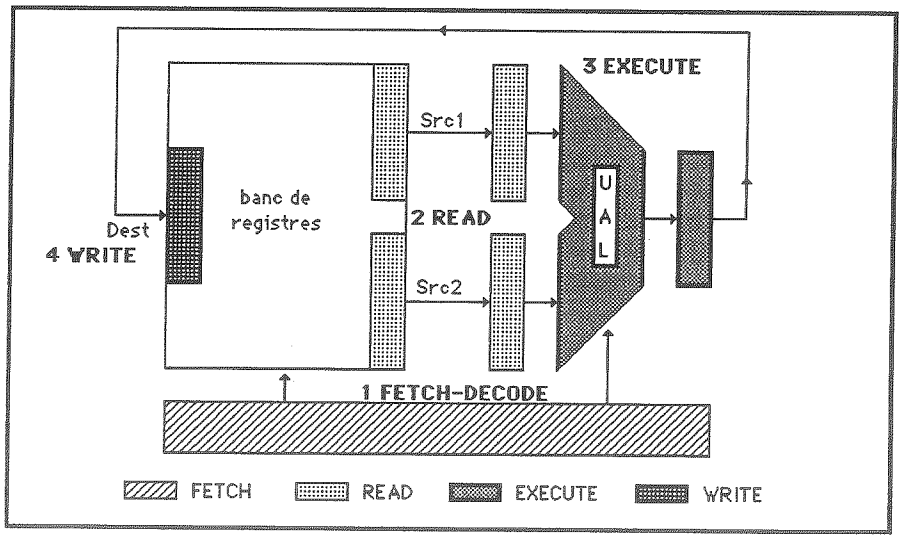


Fig. 34 - Pipeline d'exécution à quatre étages

Prenons le pipeline 4 étages de la figure 34 comme exemple. Le premier cycle effectue la lecture (*fetch*) et le décodage de l'instruction, c'est donc l'unité de lecture et de décodage qui fonctionne. Le second cycle effectue la lecture des opérandes, c'est donc le banc de registres qui est activé. Le troisième cycle exécute l'instruction sur l'Unité Arithmétique et Logique. Enfin, le troisième cycle stocke le résultat dans le registre destination.

Au travers de cette description simple, nous mettons en évidence le cadencement général des opérations nécessaires à l'exécution d'une instruction. Une fois le pipeline amorcé, les étages du processeur fonctionnent simultanément en chevauchant les différentes étapes des instructions exécutées. C'est ainsi qu'une instruction est globalement exécutée à chaque cycle, alors que dans les faits, elle en requiert plusieurs.

3.5. Les architectures Harvard

Si un processeur RISC vise théoriquement l'exécution d'une instruction par cycle, dans les faits plusieurs problèmes l'en empêchent. Le premier problème a déjà été abordé. Ce sont les ruptures d'exécution dues aux branchements qui sont en partie résolues par le mécanisme du branchement retardé. Un second problème important réside dans les accès à la mémoire de données par les instructions *load* et *store*.

Du fait du pipeline d'exécution, une instruction est lue à chaque cycle machine. L'inconvénient en retour est une saturation complète de la bande passante du bus externe et de la mémoire. Or, lors de la lecture ou de l'écriture d'un mot, il faut accéder à la mémoire. Ceci ne peut évidemment être effectué qu'en stoppant momentanément le pipeline, pour permettre le transfert de la donnée entre la mémoire et le processeur. Si l'on considère que les instructions *load* et *store* prennent deux cycles et qu'elles représentent près de 20-25% des instructions exécutées, nous nous éloignons sensiblement de l'objectif d'une instruction exécutée à chaque cycle.

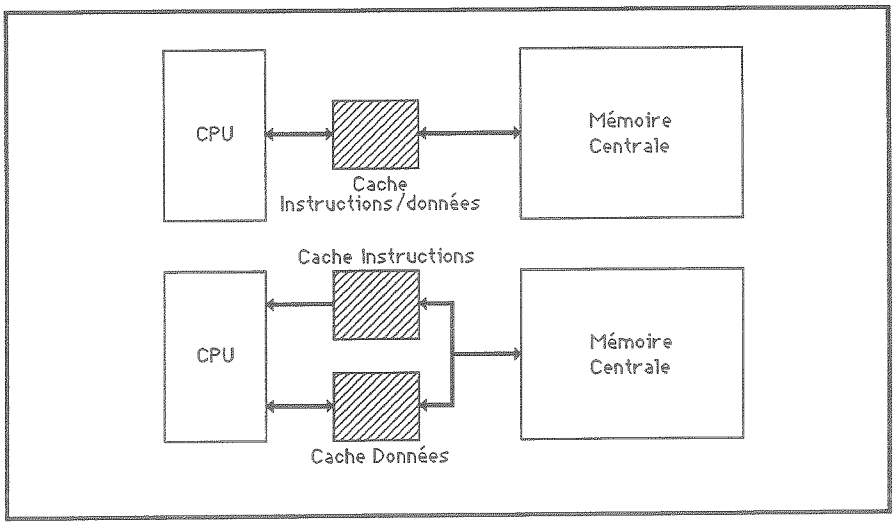


Fig. 35 - Architecture Harvard versus Von Neumann

Ce problème peut être résolu en adoptant une architecture, dite Harvard, qui sépare l'accès aux instructions des accès aux données. Ainsi séparé, un accès externe par *load* ou *store* n'entraîne plus de

rupture dans le séquencement générique du pipeline. C'est par exemple la solution retenue pour les processeurs Am29000, MC88100 et KIM™20. Si, théoriquement, cette technique nécessite deux bancs mémoires distincts, dans la pratique seuls les caches d'instructions et de données sont différents.

Nous allons maintenant aborder le dernier problème de rupture du séquencement pipeline. Il s'agit des temps d'accès à la mémoire centrale qui sont généralement supérieurs aux temps de cycle des processeurs. La solution consiste évidemment dans la mise en oeuvre de mémoires caches, appelées parfois également antémémoires.

3.6 Les mémoires caches

L'objectif d'un système de mémoire cache est d'obtenir, à partir d'une mémoire principale lente et d'un petit bloc de mémoire rapide, une mémoire de grande taille avec les temps de réponse de la mémoire rapide. Le mécanisme repose sur le maintien dans le bloc de mémoire rapide, appelé cache, des instructions et des données les plus accédées par le processeur. Une mémoire cache est généralement de 5 à 20 fois plus rapide et de 50 à 1000 fois plus petite que la mémoire principale. Comme un cache doit être extrêmement rapide, il est géré directement au niveau du matériel. Pour cette raison, sa stratégie de contrôle est, en général, relativement simple. Les systèmes de caches ont été largement étudiés et leur efficacité a été démontrée pour augmenter les performances des ordinateurs et diminuer leur coût [54].

Un cache fonctionne grâce aux propriétés de localité spatiale et temporelle des programmes. La localité temporelle signifie que les futures références mémoires ont une forte probabilité d'être semblables aux plus récentes. La localité spatiale signifie que les futures références mémoires seront proches des plus récentes. Un cache utilise la propriété de localité temporelle et spatiale, en conservant respectivement les dernières références et un groupe de données (ou d'instructions) les contenant.

La performance d'une mémoire cache se mesure généralement par le *hit ratio* (*versus miss ratio*) qui représente le rapport entre le nombre de références trouvées dans le cache et le nombre total de références générées par le processeur. Elle peut également être

mesurée en considérant le temps d'accès moyen vu par le processeur pour accéder à la mémoire.

Un système de cache peut être divisé en deux blocs, le premier spécifiquement dédié aux instructions et le second aux données. Cette architecture est employée plus efficacement dans le cas de processeurs du type Harvard (Chap. 2 § 3.5.).

Les processeurs RISC font un usage intensif des systèmes de caches du fait de l'augmentation des fréquences de fonctionnement. En effet, pour un processeur de 20 Mhz (50 cycles/nanoseconde), il est impossible de concevoir une mémoire centrale à la fois de taille importante et peu coûteuse (à 200 Mhz, le problème est encore plus évident).

Si les caches facilitent la conception des stations de travail performantes, il posent encore quelques problèmes pour les applications de contrôle pour lesquelles le non-déterminisme de leur stratégie (chargement, etc.) n'est parfois pas compatible avec les contraintes temps réel. Certain processeurs RISC intègrent directement un cache sur la puce (MIPS-X par exemple), beaucoup d'autres se voient adjoindre un boîtier supplémentaire spécialisé (SPARC, 88000, etc.). Le lecteur intéressé par l'architecture des systèmes de cache pourra se référer à [45] pour plus de détails.

4. Les controverses

4.1. L'augmentation de la taille du code

Le point clé de l'approche RISC réside donc dans cette phrase pleine de bon sens : "intégrer uniquement ce qui est réellement utile". La régularité et la simplicité des architectures élaborées sur ce principe ont donné lieu à la réalisation de processeurs dont les performances sont excellentes. Les détracteurs de RISC haranguent, comme principale objection, qu'un programme "RISC" a une taille beaucoup plus importante comparé au même programme compilé pour un processeur CISC, et, par conséquent, est moins efficace. Il est vrai que la taille d'un programme compilé sur une architecture RISC est sensiblement supérieure au même programme compilé sur une architecture CISC. Ce fait est principalement dû à l'émulation

nécessaire de certains types d'instructions et de modes d'adressage sophistiqués par des séquences d'instructions RISC.

On s'accorde généralement à tolérer une augmentation maximale de la taille du code d'environ 20%. Même dans ce cas extrême, si l'on prend comme hypothèse qu'à fréquence de fonctionnement équivalente, le processeur RISC exécute une séquence d'instructions trois à six fois plus vite qu'un processeur CISC, un programme typique est alors exécuté environ de deux à cinq fois plus rapidement. Néanmoins, une augmentation substantielle du volume d'instructions peut provenir de deux causes :

1) une taille fixe pour les instructions n'est pas toujours optimale dans le sens où certains champs de bits ne sont pas toujours nécessaires,

2) par définition, les instructions câblées doivent être à forte occurrence ; il s'ensuit que l'observation systématique d'un code trop volumineux peut provenir soit d'une défaillance survenue lors de l'étude préliminaire ayant pour but d'établir le jeu d'instructions optimal, soit de l'utilisation d'un processeur non adapté à l'application visée (i.e. les fortes occurrences du processeur RISC ne sont pas celles de l'application).

Dans les lignes qui vont suivre, nous allons passer en revue certaines caractéristiques de l'approche RISC, et pour chacune d'entre elles, en résumer les principaux avantages et inconvénients.

4.2. Un temps de cycle court

L'avantage d'un temps de cycle court est évidemment une grande rapidité d'exécution, surtout si à chaque cycle correspond l'exécution d'une instruction.

L'inconvénient réside dans la limitation du temps accordé pour le décodage et l'accès aux registres. Ces problèmes sont résolus par la simplicité même du décodage (Chap. 2 § 3.3.) et la technologie VLSI qui permet d'intégrer des fonctions logiques de plus en plus rapides.

4.3. Exécution d'une instruction par cycle

Encore une fois, l'avantage indéniable d'exécuter une instruction à chaque cycle machine est la performance.

L'inconvénient est la nécessaire simplicité (relative) du jeu d'instructions. Cela ne représente évidemment plus un désavantage lorsque l'on peut montrer que les instructions les plus occurrentes sont simples. Or, ce fait a pu être établi pour une majorité de compilateurs.

4.4. Accès à la mémoire par "load" et "store"

Ce choix permet de simplifier le jeu d'instructions et le modèle d'exécution pipeline. En outre, il limite implicitement les accès à la mémoire qui représentent le principal goulet d'étranglement pour un processeur.

Le désavantage est une restriction importante pour l'intégration de fonctions complexes du type : recherche de chaîne de caractères, conversions, fonctions graphiques sophistiquées, etc.

4.5. Un nombre de registres important

L'avantage est une importante puissance de traitement qui permet d'effectuer un maximum d'opérations en interne, sans accéder à la mémoire. Dans le cas du fenêtrage, il permet d'accélérer très sensiblement les appels et retours de procédures.

En retour, la technique de fenêtrage qui implique un nombre important de registres, accroît le temps nécessaire pour le décodage des adresses de registre et rend plus complexe la logique associée. De plus, les temps de commutation de contexte sont d'autant plus longs, ce qui peut être un obstacle pour certaines applications temps réel.

4.6. Utilisation de mémoires caches

La mise en oeuvre de systèmes de mémoire cache permet d'obtenir des systèmes performants à un coût abordable.

Il y a peu d'inconvénients si ce n'est pour certains processeurs qui imposent une architecture particulière, pas forcément compatible avec l'ensemble des applications.

4.7. Les branchements retardés

Le mécanisme du branchement retardé permet de conserver un schéma d'exécution pipeline régulier, sans "désamorçage" dû aux instructions de branchement.

Le désavantage est l'insertion systématique d'une instruction "NOP" après chaque instruction de branchement. Ce problème est résolu en ajoutant au compilateur, un optimiseur chargé de réarranger l'ordre des instructions.

4.8. Le contrôle câblé

Le contrôle câblé (*hardwired control*) permet, à l'aide d'une logique souvent très simple, d'obtenir une exécution rapide des instructions, dans un unique cycle machine.

Il ne permet pas l'excécution d'instructions complexes, comme par exemple des *load* ou *store* multiples, qui nécessitent des machines d'état spécifiques ou une microprogrammation verticale.

4.9. Le report de la complexité vers les compilateurs

Ce principe autorise la réalisation de processeurs simples et efficaces, véritables machines d'exécution (*run-time*).

Il nécessite évidemment en retour des compilateurs particulièrement bien optimisés et, par conséquent, difficiles à mettre au point et coûteux.

4.10. Finalement, RISC ou CISC ?

Malgré les quelques inconvénients cités, la technologie RISC l'emporte largement sur CISC, car, au bout du compte, le rapport complexité/performance reste le facteur déterminant. Pratiquement tous les rapports comparatifs montrent la supériorité du RISC, aussi bien pour la réalisation de stations de travail, que pour des microcontrôleurs spécialisés (voir par exemple [54]).

De plus en plus, les recherches actuelles tendent à montrer l'importance d'architectures "mixtes", matérielles et logicielles, où le compilateur est étudié en relation étroite avec le processeur. Ces travaux favorisent l'avancée de la technologie RISC et de ses prolongements VLIW ou "super-scalaire" (Chap. 1 § 6.6.). Le précurseur dans ce domaine fut sans conteste le projet MIPS de Stanford. Au niveau industriel, chaque constructeur propose maintenant sa propre architecture, plus ou moins "pure", mais toujours flanquée du slogan marketing "RISC".

Au cours du chapitre suivant, nous allons passer en revue les principaux processeurs RISC commercialisés. Pour chacun d'entre eux, une description précise de leur architecture permettra de bien cerner les choix effectués par les concepteurs, par rapport à l'approche RISC initiale.

3

Les différents processeurs RISC

1. Les processeurs RISC commercialisés

Depuis la naissance du concept RISC, concrétisée par les projets de recherche à Berkeley et à Stanford, nous assistons à une avalanche d'annonces, de conférences de presse et de manoeuvres stratégiques des grands constructeurs. Certains proposent leur propre architecture à grand renfort de publicité, d'autres insistent plutôt sur leur politique de partenariat. La raison de cette agitation technique et surtout médiatique est la naissance d'un marché nouveau. Même les cabinets d'experts en marketing et finances s'en mêlent, rivalisant de prévisions pour les ventes futures de processeurs RISC. Dans ce remue-ménage, chaque fabricant tente de se positionner au mieux, espérant soit gagner, soit ne pas perdre des segments de marchés jugés cruciaux pour l'avenir. Les deux grands domaines d'applications ouvertement visés par les fabricants de microprocesseurs sont les unités centrales de station de travail et les microcontrôleurs.

On peut dire que c'est grâce au décollage du marché des stations de travail que le concept RISC a réellement émergé des laboratoires de recherche pour envahir le monde industriel. L'argument principal en faveur de RISC réside dans un accroissement sensible des performances. Mais, pour les constructeurs de stations de travail, c'est également une solution pour s'affranchir de la tutelle des fabricants de semiconducteurs.

Le second secteur d'applications concerne les microcontrôleurs, pour lesquels les processeurs RISC apportent des solutions nouvelles qui viennent remplacer avantageusement les processeurs en tranches. Ce domaine couvre un large spectre d'applications, qui vont du processeur "postscript" dans les imprimantes laser, au processeur de contrôle spécialisé. Là encore, l'avantage décisif de RISC réside dans un excellent rapport complexité/performance qui favorise la réalisation de systèmes compacts et efficaces.

Dans ce chapitre, nous allons décrire successivement les principaux processeurs RISC commercialisés et technologiquement accessibles. Par ces termes, nous entendons les processeurs qui peuvent être achetés en tant que composants ou montés sur une carte d'évaluation. Parmi la multitude de produits existants, nous avons sélectionné le processeur SPARC de SUN Microsystems, le processeur R2000 de MIPS Computer Systems, le processeur 29000 de la société AMD, le processeur MC88100 de Motorola, le processeur 80960 d'Intel, le processeur Clipper de Fairchild, le processeur ARM commercialisé par VLSI Technology et enfin le Transputer de la société Inmos. Après un tour d'horizon rapide des autres processeurs RISC existants, comme le Precision de Hewlet Packard ou le Ridge 32, nous conclurons ce chapitre par une comparaison architecturale non-exhaustive des processeurs RISC évoqués.

2. Le processeur SPARC de SUN Microsystems

L'envolée commerciale des processeurs RISC débute véritablement en 1987 avec le lancement par SUN de la famille de stations de travail baptisée SUN4. Celle-ci est basée sur le processeur SPARC, un acronyme de *Scalable Processor ARChitecture* qui hérite directement des travaux menés sur RISC-II à Berkeley [56].

Plusieurs articles sur SPARC sont d'ailleurs cosignés par Patterson lui-même. L'ambition de SUN est d'imposer SPARC comme la norme RISC. Pour cela, la firme de Mountain View multiplie les contrats de partenariat, dont le principal est celui signé avec AT&T, le géant américain des télécommunications, mais également le père du système d'exploitation Unix. Pour assurer une large diffusion de SPARC, SUN noue des liens avec de nombreux fabricants de semiconducteurs : la société Fujitsu qui fournira les premiers processeurs SPARC sur une matrice prédiffusée CMOS 1.5 micron, Cypress Semiconductor, Bipolar Integrated Technology, LSI Logic, Texas Instruments et, plus récemment, Philips.

Comme RISC-II, le processeur SPARC repose sur une architecture relativement simple, qui doit lui permettre de s'adapter à la programmation en langage C et aux fonctionnalités du système d'exploitation Unix. Pour les applications nécessitant une part importante de calcul numérique, une interface avec un coprocesseur flottant fait partie intégrante de l'architecture. Les applications de l'Intelligence Artificielle n'ont également pas été laissées de côté, puisque SPARC tire parti des travaux menés sur SOAR (voir Chap. 1 § 4.3.) avec deux instructions arithmétiques capables de manipuler des données typées (instructions TADD et TSUB).

Le processeur SPARC comprend 136 registres internes à trois ports d'accès, deux de lecture et un d'écriture, organisés en huit fenêtres à recouvrement partiel lors des appels fonctionnels. Nous noterons au passage que le mécanisme de décalage des fenêtres de registres n'est pas géré directement par les instruction *call* et *return*, mais par deux instructions spécifiques, *save* et *restore*. Cette caractéristique présente l'avantage d'une gestion plus souple de la roue de registres, mais également l'inconvénient majeur de ralentir les appels et retours fonctionnels dès lors que l'on désire tirer pleinement parti du fenêtrage.

En outre, SPARC ne possède pas à proprement parler d'instruction *return*, mais d'une opération "jmpl" (*jump and link*) qui sauvegarde le PC dans un registre, puis effectue en deux cycle un branchement à une adresse déterminée.

Le pipeline d'exécution du processeur SPARC comprend quatre étages : le premier effectue la lecture de l'instruction (IF - *Instruction Fetch*), le second décode les opérandes (ID - *Instruction Decode*), le troisième exécute les opérations sur l'Unité

Arithmétique et Logique (EX - *Execute*) et le quatrième écrit le résultat dans le registre destination (WR - *Write*).

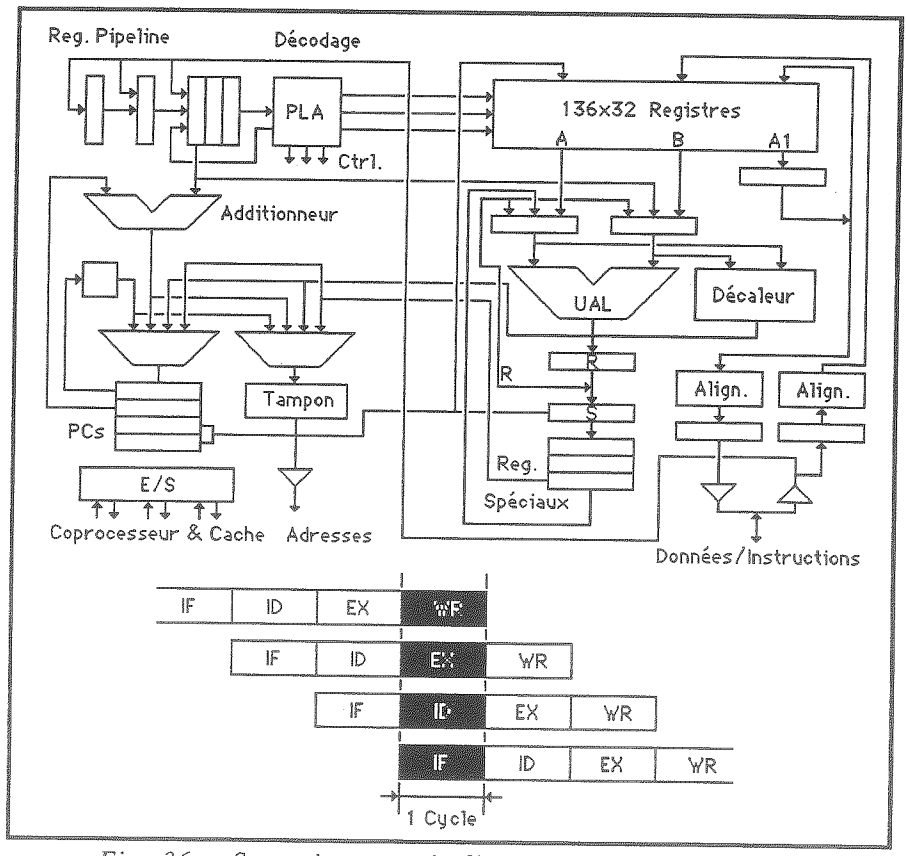


Fig. 36 - Synoptique et pipeline du processeur SPARC

Le jeu d'instructions SPARC comprend 64 instructions, dont à peu près la moitié s'exécute en un cycle machine une fois le pipeline amorcé. La seconde moitié des instructions comporte plusieurs versions des opérations *load* et *store* qui permettent la manipulation d'octets, de mots, de double mots ou des fontions d'interfaces avec le coprocesseur. Ces instructions prennent entre 2 et 4 cycles selon la taille de la donnée à transmettre. Les instructions de branchement du processeur SPARC sont de type retardées (*delayed branch*) et conçues sur le modèle RISC-II. Elles ne nécessitent qu'un seul cycle, à condition que l'instruction suivante (*delay-slot*) soit judicieusement affectée par le compilateur.

Conformément à la méthodologie RISC, SPARC est basé sur cinq formats d'instructions (*call*, *sethi*, *branch*, les opérations entières et les opérations flottantes) qui ont toutes la taille d'un mot mémoire, c'est-à-dire 32 bits.

L'architecture interne du processeur comprend deux Unités Arithmétiques et Logiques, la première pour les instructions, la seconde pour le calcul des adresses de branchement en parallèle. Comme RISC-II, SPARC est caractérisé par un bus d'adresse séparé du bus des données. Mais ce dernier est multiplexé pour la lecture des instructions et les accès du type *load* et *store*. L'avantage est une exellente compacité et une bonne gestion de la mémoire, mais, en retour, un bus commun aux instructions et aux données ralentit inévitablement les instructions *load* et *store* (Chap. 2 § 3.5.). Nous étudierons ce problème en détail lors de l'analyse du pipeline d'exécution du processeur KIM.

SPARC s'est affirmé dans le domaine des stations de travail et l'on peut dire que c'est un processeur d'usage général par excellence. Sa simplicité en fait un candidat sérieux pour la réalisation ASIC sous forme d'une macrocellule. Ainsi, LSI Logic propose aux concepteurs d'intégrer sur le même circuit, le coeur SPARC et d'y adjoindre les fonctions périphériques nécessaires à l'application, pour disposer d'une réalisation extrêmement compacte.

Un des objectifs techniques importants de SPARC est d'assurer un passage direct dans les technologies plus rapides. La première version, conçue par Fujitsu est annoncée pour une puissance de 11,9 MIPS à 16,7 Mhz. D'ores et déjà, la société américaine Cypress Semiconductor produit des circuits SPARC à 40 Mhz (famille 7C600) visant une performance supérieure à 25 MIPS [57].

Mais la course technologique ne s'arrête pas là, avec une version ECL proposée par Bipolar Integrated Technology, une version AsGa développée par la société Prisma, sans oublier bien sûr SUN en coopération avec Gigabit Logic. Pour plus de renseignements sur SPARC et sa conception, se référer aux articles publiés par SUN [58-66].

LDSB(A)	load signé 8 bits *	ADD(cc)	addition entiére
LDSH(A)	load signé 16 bits *	ADDX(cc)	addition avec carry
LDUB(A)	load non signé 8 bits *	TADDcc(tv)	addition typée
LDUH(A)	load non signé 16 bits *	SUB(cc)	soustraction entiére
LD(A)	load 32 bits *	SUBX(cc)	soustraction avec carry
LDD(A)	load 64 bits *	TSUBcc(tv)	soustraction typée
LDF	load flottant *	MULScc	pas de multiplication entière
LDDF	load double flottant *	AND(cc)	ET logique
LDFSR	load mot d'etat FP *	ANDN(cc)	ET logique et négation
LDC	load coprocesseur *	OR(cc)	OU logique
LDDC	load double copro. *	ORN(cc)	OU logique et négation
LDCSR	load mot d'état copro. *	XOR(n)	OU exclusif
STB(A)	store 8 bits *	XNOR(n)	NON-OU exclusif
STH(A)	store 16 bits *	SLL	décalage à gauche logique
ST(A)	store 32 bits *	SRL	décalage à droite logique
STD(A)	store 64 bits *	SRA	décalage à droite arithmétique
STF	store flottant *	SETHI	affecte les MSB du registre R
STDF	store double flottant *		
STFSR	store mot d'état flot.*	RDY	lecture registre Y
STDFQ	store double FP queue *	RDPSR	lecture mot d'état
STC	store coprocesseur *	RDWIM	lecture du masque de fenêtrage
STDC	store double copro. *	RDTBR	lecture du registre de base trap
STCSR	stote mot d'état copro.*	WRY	écriture du registre Y
STDCQ	store dble copro. queue *	WRPSR	écriture du mot d'état
LDSTUB(A)	load atomique 8 bits *	WRWIM	écriture du masque de fenêtrage
SWAP(A)	swap registres *	WRTBR	écriture du registre de base trap
		UNIMP	instruction inexistante
SAVE	incrément fenêtre	IFLUSH	flush le cache d'instruction
RESTORE	décrément fenêtre	FPop	opération flottante
Bicc	branchement sur cond.	CPop	opération sur le coprocesseur
FBicc	branchement sur cond. FP		
CBccc	branchement sur cond. CP		
CALL	appel sans fenêtrage		
JMPL	saut et lien *		
RETT	retour d'exception *		
Ticc	trap sur condition *		

* de 2 à 4 cycles selon l'instruction au lieu de 1
A accès à l'espace alterné
cc modification des codes conditions
tv trap sur débordement

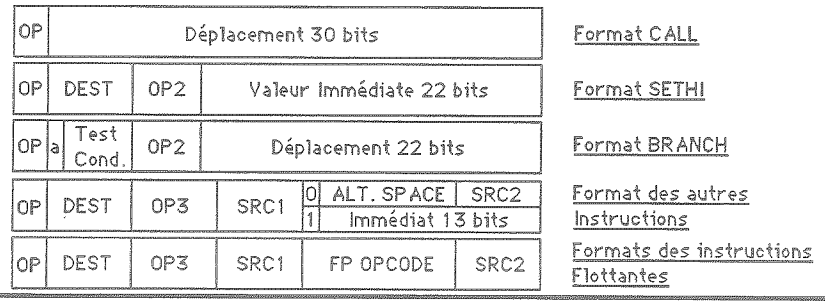


Fig. 37 - Formats et jeu d'instructions du processeur SPARC

3. Le processeur R2000 de MIPS Computer

En 1985, la société MIPS Computer Systems annonçait un processeur RISC, baptisé R2000, directement issu des travaux menés à l'Université de Stanford [67].

Bien que n'étant pas constructeur de stations de travail, cette firme, installée à Sunny Vale en Californie, vise des marchés similaires à ceux de SUN Microsystems, mais avec une stratégie différente. MIPS Computer vend des composants et des systèmes à des constructeurs qui intègrent la technologie MIPS dans leurs produits. Pratiquement, pas de ventes directes : le marché de MIPS Computer repose sur un ensemble de contrats de licence dont les plus importants sont Digital Equipment, NEC, Siemens, Silicon Graphics, Seiko et bien d'autres encore. Pour la fabrication des circuits, MIPS s'appuie principalement sur trois "fondeurs" : Integrated Device Technology, LSI Logic et Performance Semiconductor.

Le processeur MIPS R2000 est composé de deux parties intégrées sur le même circuit : un processeur RISC 32 bits et un système de contrôle de mémoire cache couplés à une unité de gestion de la mémoire (MMU). Le processeur R2000 est donné pour une puissance de 12 MIPS à 16,7 Mhz.

La partie processeur du circuit comprend 32 registres généraux sans mécanisme de fenêtrage, un compteur de programme (PC) et deux registres de 32 bits pour stocker les résultats des opérations de multiplication et division entières sur 64 bits (registres Hi et Lo). Nous noterons que le mot d'état n'est pas dans la partie processeur, mais dans la partie de contrôle.

Le jeu d'instructions R2000 est composé de 74 instructions réparties en six catégories: les instructions du type *load-store*, les instructions Arithmétiques et Logiques, les instructions de contrôle du séquencement, les instructions d'interfaçage avec le coprocesseur de calcul R2010, les instructions dédiées à la partie contrôle du circuit et quelques instructions spéciales.

Toutes les instructions sont codées dans un mot fixe de 32 bits, selon 3 formats : le format "Immédiat" (I-type), le format "Jump" (J-type) et le format "Registre" (R-type). Une caractéristique du jeu

d'instructions R2000 est l'absence d'instructions *call* et *return*, remplacées par les instructions *jump and link* et *jump and link register*. Les opérations de branchement sont du type retardé (*delayed branch*) et nécessitent une réorganisation du code généré par le compilateur pour ne pas perdre à chaque fois un cycle (*delay-slot*). Comme pour SPARC, le contrôle des instructions est câblé plutôt que microprogrammé.

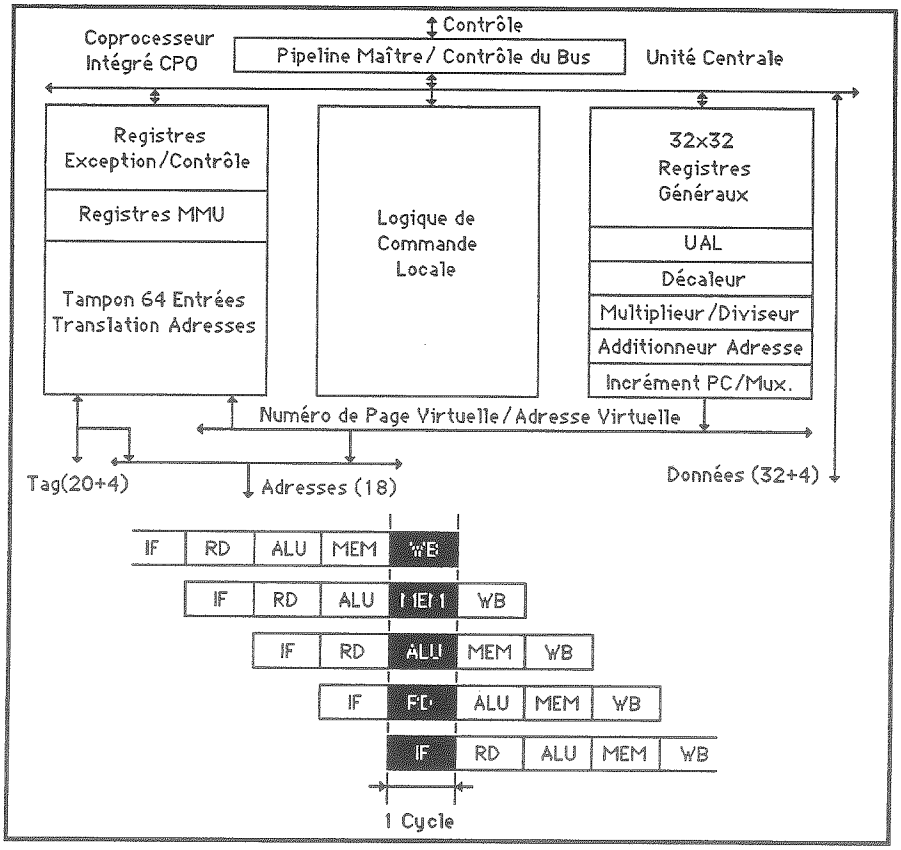


Fig. 38 - Synoptique et pipeline du processeur R2000

Le pipeline d'exécution du processeur R2000 comprend six étages : le premier effectue la lecture de l'instruction (IF - *Instruction Fetch*), le second lit les opérandes (RD - *Read*), le troisième exécute l'opération sur l'Unité Arithmétique et Logique (ALU), le quatrième cycle est réservé aux éventuels accès mémoire (MEM) et le dernier écrit le résultat de l'instruction dans le registre Destination (WB *Write Back*).

La partie contrôle du circuit comprend une unité de contrôle gérant en particulier les exceptions (*traps* et *interruptions*), une unité de gestion mémoire et une table *Lookup* de translation de 64 mots permettant de générer une adresse comprenant un numéro de page et une adresse virtuelle sur 16 bits. Le bus de données est multiplexé pour la lecture des instructions (*fetch*) et l'accès aux données stockées en mémoire (*load* et *store*).

Mais la principale caractéristique du processeur R2000 reste sa simplicité au niveau architectural. En effet, du projet MIPS initial, il hérite d'une structure matérielle débarrassée des problèmes d'interblocages dans les phases du pipeline. La force des processeurs MIPS réside dans une relation très étroite entre l'architecture du processeur et les compilateurs associés.

Le domaine d'applications visé par le processeur R2000 est le monde du calculateur universel sous Unix. Le jeu d'instructions a été particulièrement optimisé pour la compilation du langage C, mais permet également d'obtenir des codes très efficaces en Fortran ou en Lisp. Là encore, la force reconnue de MIPS réside dans la qualité des compilateurs. R2000 semble moins bien adapté pour les applications spécifiques à cause de l'unité de gestion mémoire et du système de contrôle de la mémoire cache, directement intégrés dans le même circuit, mais inutilisables dans la plupart des systèmes de contrôle.

Comme pour SPARC, l'avancée technologique est un atout majeur de MIPS qui permet d'obtenir rapidement des circuits en technologie rapide. Ainsi, en 1988, MIPS Computer Systems présentait le processeur R3000, compatible avec le circuit R2000, mais plus performant [68]. Réalisé en technologie Custom CMOS, le processeur R3000 permet d'atteindre une puissance de 20 MIPS à 25 Mhz. D'ores et déjà une version ECL (55 MIPS) du processeur MIPS est produite (R6000).

LB	load 8 bits
LBU	load non signé 8 bits
LH	load 16 bits
LHU	load non signé 16 bits
LW	load 32 bits
LWL	idem, octet aligné à gauche
LWR	idem, octet aligné à droite
SB	store 8 bits
SH	store 16 bits
SW	store 32 bits
SWL	idem, octet aligné à gauche
SWR	idem, octet aligné à droite
J	saut
JAL	saut et lien
JR	saut registre
JALR	saut et lien registre
BEQ	branchement si égal
BNE	branch. si non égal
BLEZ	branch. si égal zéro
BGTZ	branch. si sup. à zéro
BLTZ	branch si inf. à zéro
BGEZ	branch. si sup. ou égal zéro
BLTZAL	branch. si inf. zéro et lien
BGEZAL	branch. si sup. ou égal 0 et lien
SYSCALL	trap système
BREAK	trap de mise au point
LWCz	load coprocesseur 32 bits
SWCz	store coprocesseur 32 bits
MTCz	move vers coprocesseur
MFCz	move depuis coprocesseur
CTCz	move contrôle vers copro.
CFCz	move contrôle depuis copro.
COPz	opération du coprocesseur
BCzT	branch. si cond. vraie
BCzF	branch. si cond. fausse

ADDI	addition immédiate
ADDIU	addition immédiate non signée
SLTI	compare immédiat
SLTIU	compare immédiat non signé
ANDI	ET logique immédiat
ORI	OU logique immédiat
XORI	OU exclusif immédiat
LUI	load 16 bits MSB immédiat
ADD	addition entière
ADDU	addition non signée
SUB	soustraction entière
SUBU	soustraction non signée
SLY	comparaison registre
SLTU	comparaison registre non signée
AND	ET logique
OR	OU logique
XOR	OU exclusif
NOR	NON-OU logique
SLL	décalage à gauche logique
SRL	décalage à droite logique
SRA	décalage à droite arithmétique
SLLV	décalage multiple à gauche logique
SRLV	décalage multiple à droite logique
SRAV	décalage multiple à droite arithmétique
MULT	multiplication signée
MULTU	multiplication non signée
DIV	division entière signée
DIVU	division entière non signée
MFHI	move contenu du reg. Hi dans reg.
MFLO	move contenu du reg. Lo dans reg.
MTHI	move contenu reg. dans reg. Hi
MTLO	move contenu reg. dans reg. Lo
MTCO	move vers coprocesseur CP0
MFCO	move depuis coprocesseur CP0
TLBR	lecture indexée d'une entrée TBL
TLBWI	écriture indexée d'une entrée TBL
TLBWR	écriture aléatoire d'une entrée TBL
TLBP	test d'une entrée TBL
RFE	restitution du mot d'état après exception

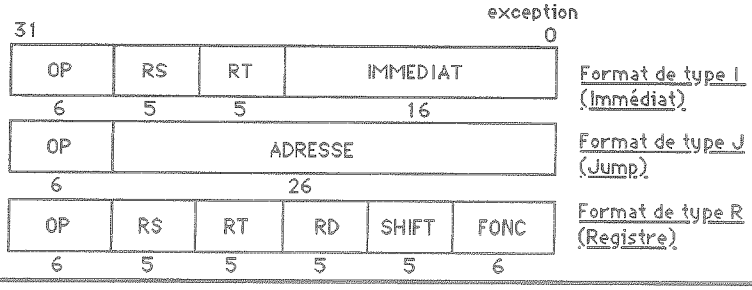


Fig. 39 - Formats et jeu d'instructions du processeur R2000

4. Le processeur Am29000 de AMD

En Mars 1987, la société AMD, rendue célèbre par la famille des processeurs 2900, adoptait une stratégie RISC pour conserver ses parts de marché. Le processeur Am29000 hérite directement des travaux menés à l'Université de Berkeley, tout en tirant parti de l'expérience d'AMD pour obtenir une architecture remarquablement souple [69].

Associée au coprocesseur de calcul virgule flottante Am29027, il a une puissance annoncée de 17 MIPS à 25 Mhz et de 20 MIPS à 30 Mhz. Le processeur Am29000 utilise 192 registres internes organisés en 8 fenêtres à recouvrement partiel, ou selon tout autre mode défini par l'utilisateur. L'avantage de cette méthode est bien évidemment sa souplesse, en particulier pour la gestion des contextes dans les applications temps réel. Cependant, cette souplesse devient un handicap important lorsque le compilateur veut profiter du mécanisme de fenêtrage. Dans ce cas, il faut pour chaque procédure un "prologue" de trois instructions et un "épilogue" de quatre instructions pour commuter les bancs de registres et vérifier les problèmes de débordement (*overflow* et *underflow*).

Le jeu d'instructions du processeur AMD est le reflet des multiples adjonctions faites sur l'architecture RISC de base. Cette caractéristique rend le 29000 assez éloigné de la philosophie RISC initiale. En effet, le jeu d'instructions comprend 112 codes opérations différents, répartis en neuf catégories : les instructions arithmétiques, les instructions logiques, les instructions de décalage, les instructions de comparaison, les opérations de mouvement de données, celles manipulant les constantes, les instructions de calcul en format flottant, les instructions de branchement, les instructions spéciales et enfin les opérations réservées. Les branchements sont du type retardé (*delayed-branch*) avec un cycle à optimiser par le compilateur. Comme SPARC, le processeur Am29000 possède une instruction *call*, mais pas d'instruction *return*. Cette dernière est généralement émulée par une instruction *jumpi* qui prend pour argument l'adresse de retour sauvée par l'instruction *call*. L'ensemble des instructions est codé sur un format fixe de 32 bits et s'exécute pour la plupart en un cycle, à l'exception des instructions *iret*, *iretinv*, *loadm* et

storem. Une caractéristique du processeur Am29000 réside dans la définition d'instructions non exécutées directement par le processeur qui déclenche alors un *trap*.

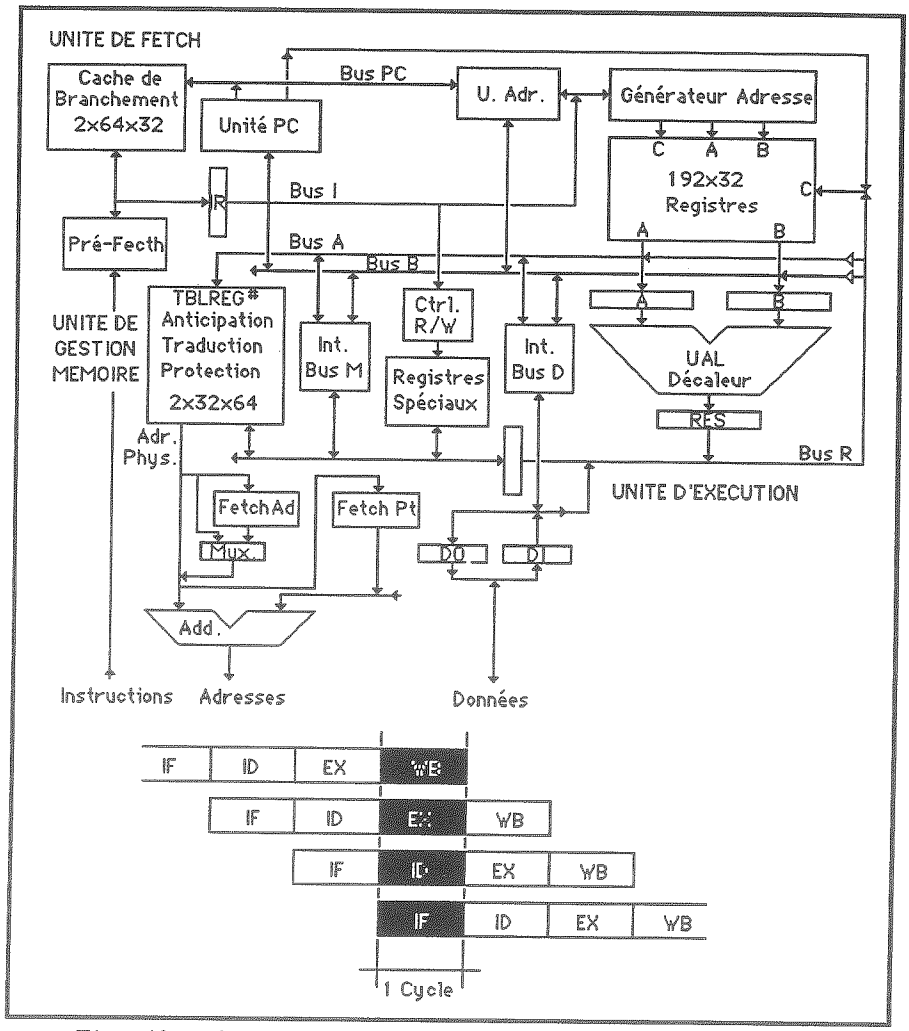


Fig. 40 - Synoptique et pipeline du processeur Am29000

Parmi celles-ci figurent les opérations virgules flottantes et les instructions de multiplication et division entière. Ces codes opérations correspondent à de futures extensions non encore réalisées qui doivent permettre un branchement rapide à des routines assembleur pour les émuler.

ADD	addition entière
ADDC	addition avec carry
ADDCS	addition signée avec carry
ADDCU	addition non signée avec carry
ADDS	addition signée
ADDU	addition non signée
SUB	soustraction entière
SUBC	soustraction avec carry
SUBCS	soustraction signée avec carry
SUBCU	sous. non signée avec carry
SUBR	soustraction inverse
SUBRC	sous. inverse avec carry
SUBRCS	sous. inv. signée avec carry
SUBRCU	sous. inv. non signée avec carry
SUBS	soustraction signée
SUBU	soustraction non signée
AND	ET logique
ANDN	ET-NON logique
NAND	NON-ET logique
OR	OU logique
NOR	NON-OU logique
XOR	OU exclusif
XNOR	OU-NON exclusif
SLL	décalage à gauche logique
SRA	décalage à droite arithmétique
SRL	décalage à droite logique
DIV	pas de division entière
DIV0	initialisation division
DIVIDE	division entière
DIVIDU	division non signée
DIVL	dernier pas de division
DIVREM	reste de division
MUL	pas de multiplication
MULL	dernier pas de multiplication
MULTIPLU	multiplication non signée
MULTIPLY	multiplication signée
MULU	pas de multiplication non signé
EXBYTE	extraction octet
EXHW	extraction 16 bits
EXHWS	idem, extension du signe
EXTRACT	extraction mot aligné
INBYTE	insertion octet
INHW	insertion 16 bits
LOAD	load
LOADL	load et verrouille
LOADM	load multiple
LOADSET	load et set
STORE	store
STOREL	store et verrouille
STOREM	store multiple

HALT	mode Halt
INV	invalidation
IRET	retour d'interruption
IRETINV	retour d'interruption, invalid.
JMP	saut
JMPF	saut si faux
JMPFDEC	saut et décrément si faux
JMPFI	saut indirecte si faux
JMPI	saut indirecte
JMPT	saut si vrai
JMPTI	saut indirecte si vrai
CALL	appel de procédure
CALLI	appel indirecte
EMULATE	trap routine d'émulation
CLZ	remise à zéro des compteurs
CONST	constante
CONSTH	constante positive
CONSTN	constante négative
CONVERT	conversion de format
CPBYTE	comparaison octets
CPEQ	comparaison égalité
CPGE	comparaison sup. ou égale
CPGEU	idem non signée
CPGT	comparaison supérieur
CPGTU	idem non signée
CPLE	comparaison inf. ou égale
CPLEU	idem non signée
CPLT	comparaison inférieur
CPLTU	idem non signée
CPNEQ	comparaison non égale
MFSR	move depuis un registre spécial
MFTBL	move depuis la TBL
MTSR	move vers registre spécial
MTSRIM	move immédiat vers reg. spé.
MTTBL	move vers la TBL
ASEQ	assert égale
ASGE	assert supérieur ou égale
ASGEU	idem non signé
ASGT	assert supérieur ou égale
ASGTU	idem non signé
ASLE	assert inférieur ou égale
ASLEU	idem non signé
ASLT	assert inférieur
ASLTU	idem non signé
ASNEQ	assert non égale

Fig. 41 - Le jeu d'instructions (1) du processeur Am29000

Le pipeline d'exécution du 29000 est basé sur un modèle à quatre étages qui résout les problèmes d'interblocage directement par le matériel. Le premier cycle effectue la lecture de l'instruction (IF - *Instruction Fetch*), le second décode les opérandes (ID - *Instruction Decode*), le troisième exécute l'opération sur l'Unité Arithmétique et Logique (EX - *Execute*) et le quatrième écrit le résultat dans le registre destination (WB - *Write Back*). Contrairement aux processeur SPARC et MIPS, l'Am29000 possède un bus d'accès aux instructions séparé du bus d'accès aux données, ce qui lui permet de ne pas briser la régularité du pipeline lors des accès à la mémoire par les instructions *load* et *store*.

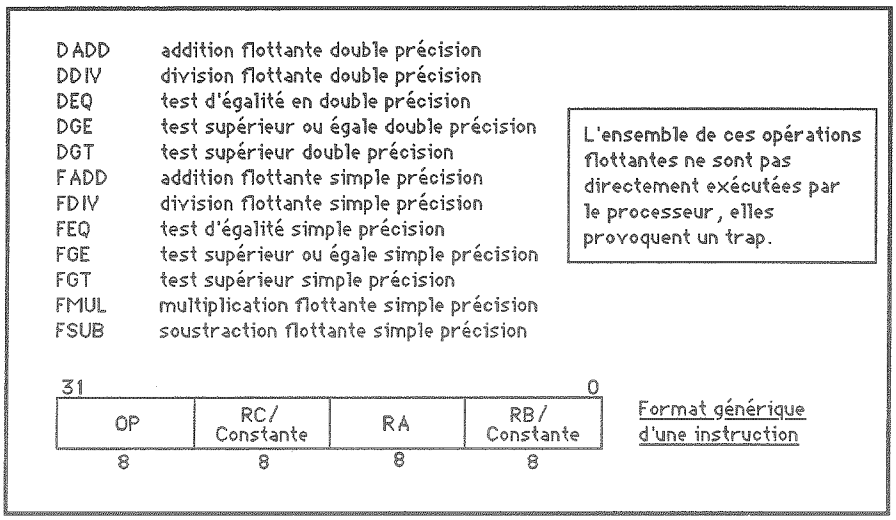


Fig. 42 - Format et jeu d'instructions (2) du processeur Am29000

Le processeur 29000 intègre également une mémoire cache de "cible de branchement" qui contient les 32 dernières adresses de branchement utilisées, ainsi que pour chacune de ces adresses, quatre instructions de la séquence de branchement. L'Am29000 comprend une unité de gestion de mémoire virtuelle paginée permettant d'accéder à 4 milliards d'octets par tâche pour un maximum de 256 tâches. Cette unité contient en outre une mémoire "tampon" qui stocke les adresses des 32 dernières pages et assure ainsi une traduction d'adresse en un seul cycle. Enfin, le 29000 possède un système de protection très proche de celui utilisé pour le système d'exploitation Unix [70].

Le processeur Am29000 est caractérisé par une architecture simple issue de Berkeley, mais complexifiée à l'extrême par l'ajout de fonctionnalités jugées intéressantes. De ce fait, beaucoup d'ingénieurs, actuellement, ne le présentent pas comme un processeur RISC, mais comme une machine hybride qui hérite des travaux de Berkeley et de l'expérience AMD.

Conçu à la fois pour servir d'unité centrale dans une station de travail ou de microcontrôleur pour une application spécifique, le 29000 est pourtant moins utilisé que SPARC ou MIPS. En effet, bien que la société AMD ait mis l'accent sur l'environnement de mise au point, son succès repose principalement sur la disponibilité des logiciels de sociétés tierces qui, en retour, n'investiront dans le 29000 qu'en cas de succès (!). Au niveau des applications spécifiques, il faut également reconnaître que le processeur SPARC, du fait de sa simplicité et des logiciels disponibles, prend bien souvent le pas sur le processeur d'AMD [71].

5. Le processeur MC88100 de Motorola

Baptisée MC88000, la famille RISC de Motorola ressemble à bien des égards au processeur Clipper de Fairchild (voir paragraphe suivant). Le "88000" se compose en fait de deux boîtiers distincts, le 88100 et le 88200, qui correspondent respectivement à l'unité centrale et à l'unité de gestion mémoire doublée d'un cache de 16 Koctets. Une configuration standard comprend une unité 88100 et deux circuits 88200, l'un servant de cache pour les instructions et l'autre pour les données. Cette caractéristique permet de classer le 88000 dans la famille des processeurs RISC à architecture Harvard [72]. Nous noterons au passage que l'on peut utiliser deux circuits 88100 pour les machines à tolérance de pannes, ainsi que huit circuits 88200 portant ainsi la mémoire cache à 128 Koctets. Le MC88100 est intégré sur une puce CMOS de 165000 transistors. Quant au MC88200, il compte près de 750000 transistors. Les deux circuits ont été conçus respectivement en 190 et 120 mois d'ingénieurs sur les outils Genesil et GDT de Silicon Computer Systems [73]. A 20 Mhz, un système 88000 exécute 16 MIPS et de 7 à 12 Mflop pour la partie flottante.

Le 88100 comprend 32 registres généraux, 21 registres de contrôle accessibles en mode superviseur nécessaire pour la gestion des exceptions et 11 registres de contrôle pour l'unité de calcul virgule flottante. Le jeu d'instructions comprend 51 instructions à format fixe de 32 bits, réparties en six catégories : les instructions arithmétiques, les instructions de calcul en format flottant, les instructions logiques, les opérations de manipulation de champ de bits, les instructions de contrôle de séquencement et les inévitables *load* et *store*.

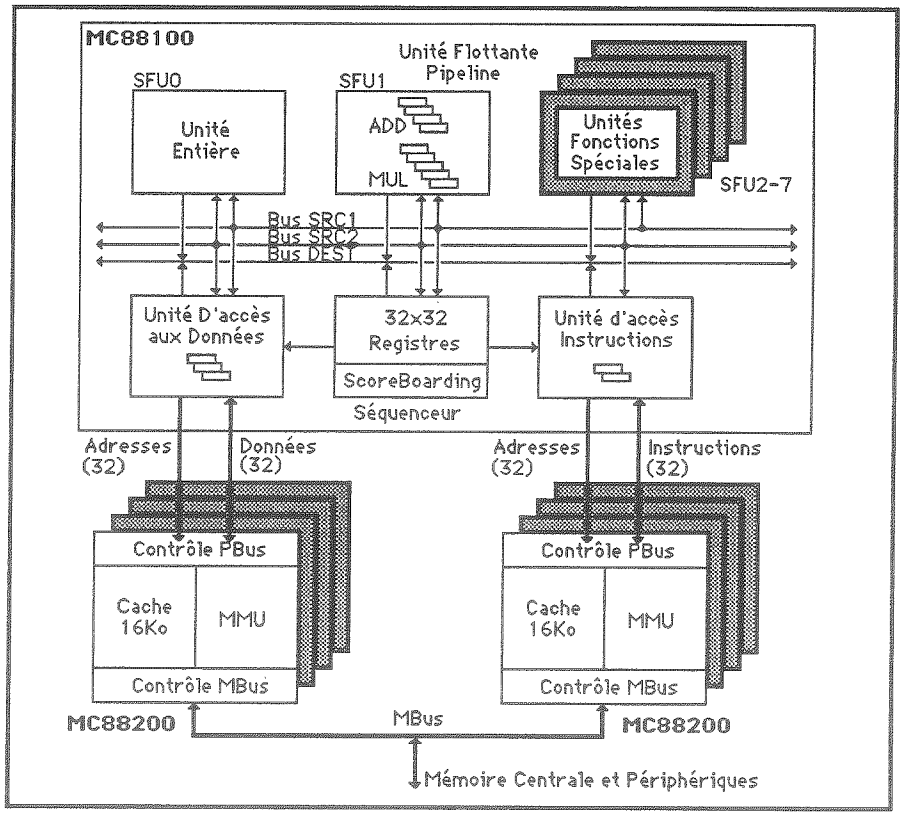


Fig. 43 - Synoptique du processeur MC88100

La plupart des instructions s'exécutent en un cycle machine grâce aux deux bus distincts. Les instructions de branchement sont retardées comme dans la majorité des autres processeurs RISC. Le décodage est entièrement câblé (pas de microcode) et le pipeline d'exécution général comprend classiquement quatre étages. Une des caractéristiques du processeur 88100 est d'intégrer, outre l'Unité

Arithmétique et Logique entière, une unité de calcul virgule flottante IEEE et un multiplieur 32 bits, câblé directement sur la même puce.
Depuis son annonce officielle, quoique tardive, le processeur RISC de Motorola s'impose dans la plupart des sociétés qui, historiquement, utilisaient la technologie Motorola. Un fait surprenant a priori est la concurrence relative entre le 88000 et la famille 68000, cheval de bataille du géant américain. Mais le marketing Motorola est assez subtil pour maintenir ces deux gammes en même temps sur des créneaux qu'il juge parallèles mais non concurrentiels.

ADD	addition entière	CLR	initialisation champ de bits
ADDU	addition entière non signée	EXT	extraction signée champ de bits
CMP	comparaison	EXTU	idem non signée
DIV	division entière	FF0	recherche 1er bit à 0
DIVU	division non signée	FF1	recherche 1er bit à 1
MUL	multiplication entière	MAK	construction champ de bits
SUB	soustraction entière	ROT	rotation registre
SUBU	soustraction non signée	SET	set champ de bits
FADD	addition flottante	LD	load registre depuis la mémoire
FCMP	comparaison flottante	LDA	load adresse
FDIV	division flottante	LDCR	load depuis registre de contrôle
FLDCR	load depuis reg. ctrl. FP	ST	store vers la mémoire
FLT	conversion flottante	STCR	store vers registre de contrôle
FMUL	multiplication flottante	XCR	échange registre de contrôle
FSTCR	load vers reg. ctrl. FP	XMEM	échange registre avec la mémoire
FSUB	soustraction entière		
FXRC	échange reg. ctrl. FP	BB0	branchement si bit à 0
INT	arrondi entier	BB1	branchement si bit à 1
NINT	idem entier le plus proche	BCND	branchement conditionnel
TRNC	tronque entier le plus proche	BR	branchement inconditionnel
		BSR	branchement à une procédure
AND	ET logique	JMP	saut inconditionnel
MASK	masque immédiat	JSR	saut à une procédure
OR	OU logique	RTE	retour d'exception
XOR	OU exclusif	TB0	trap si bit à 0
		TB1	trap si bit à 1
		TBND	trap sur test des limites
		TCND	trap conditionnel

Fig. 44- Le jeu d'instructions du processeur MC88100

Comme le 29000, le processeur Motorola vise à la fois les marchés spécifiques et les stations de travail. Dans le premier cas, on ne peut que regretter sa complexité et la (quasi) nécessité d'une configuration dotée de deux circuits 88200. Mais, pourvu de ses caches, le 88000 peut alors faire valoir l'ensemble de ses qualités :

efficacité, gestion mémoire performante et simplicité de conception de machines multiprocesseurs. Depuis son annonce, un consortium d'une vingtaine de constructeurs, baptisé 88 Open, s'est rangé derrière Motorola. Parmi eux, la société Data-General propose une station de travail aux performances très intéressantes et prépare une version ECL du processeur pour 1991.

Outre l'unité de calcul IEEE et l'unité entière, le processeur 88100 pourra intégrer dans l'avenir jusqu'à huit unités d'opérateurs spécialisés. Cette architecture prometteuse n'est pas sans rappeler les structures "VLIW" ou "super-scalaires" (Chap. 1 § 6.6.), du moins pour la partie opérative.

6. Le processeur CLIPPER de Fairchild

Le Clipper C100 fut l'un des tous premiers processeurs RISC disponibles sur le marché des composants électroniques. Le processeur a été initialement conçu par la société Fairchild, puis ensuite fabriqué par Intergraph. Bien que tirant parti de l'ensemble des projets RISC initiaux, le Clipper a repris de nombreux concepts mis en oeuvre dans le Cray-1. Par exemple, le processeur Fairchild fut le premier à intégrer plusieurs unités fonctionnelles en parallèle, comme une unité entière, une unité de calcul au format flottant, un système de cache et un système de gestion de la mémoire virtuelle. Une caractéristique intéressante, reprise ensuite dans le circuit Motorola MC88100, est le mécanisme de *scoreboarding*. Ce dernier permet d'améliorer le rendement du pipeline d'exécution pour la gestion de multiples unités opératives. En identifiant à tout instant les unités non utilisées, le décodeur d'instructions peut ainsi allouer un opérateur pour une nouvelle instruction. Le mécanisme repose sur la gestion d'un bit associé à chaque opérateur, autorisant le verrouillage ou la libération de la fonction.

Le Clipper C100 comprend 3 circuits VLSI implantés sur une microcarte : le processeur "entier", le coprocesseur de calcul en virgule flottante et un circuit intégrant l'unité de gestion mémoire, ainsi que deux mémoires caches de 4 Koctets. Comme le 88000, le Clipper repose sur une architecture Harvard qui sépare les accès aux instructions des accès aux données. A une fréquence de

fonctionnement de 33 Mhz, le C100 présente une performance de l'ordre de 5 MIPS. Une seconde version, baptisée C300, permet de doubler les performances grâce à une horloge voisine de 50 Mhz [68].

Le processeur Clipper contient 16 registres généraux 32 bits "utilisateur", doublés par 16 registres généraux 32 bits accessibles en mode "superviseur". Il comprend également 8 registres 64 bits pour l'unité virgule flottante accessibles dans les deux modes de fonctionnement, ainsi qu'un compteur de programme (PC) et deux mots d'états (un pour chaque mode). Chaque système de cache, outre les tables de transcodage et la mémoire cache proprement dite, intègre 5 registres de contrôle. Là encore, comme le circuit Motorola, il n'y a pas de mécanisme de fenêtrage possible.

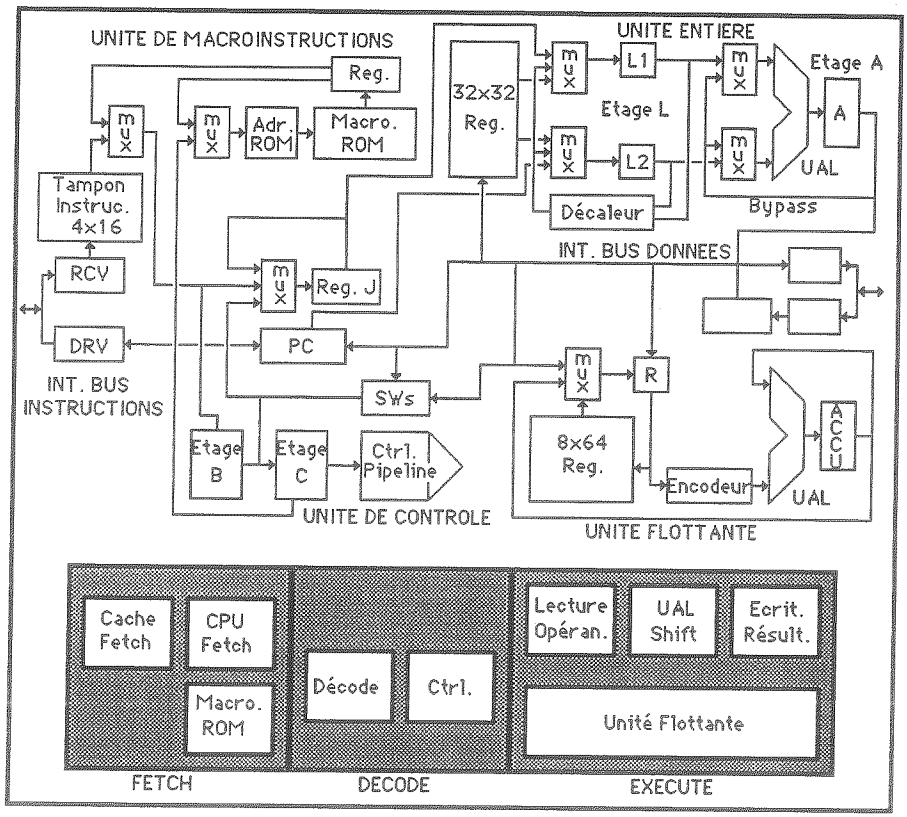


Fig. 45 - Synoptique et pipeline du processeur Clipper C100

Le jeu d'instructions du Clipper se compose de 101 instructions "câblées" et de 67 instructions de "haut-niveau" microcodées sous la forme de séquences d'instructions "câblées". L'ensemble du jeu d'instructions se répartit en 10 classes encodées grâce à 15 formats différents et 9 types d'adressages. Les catégories regroupent les instructions *load* et *store*, les opérations Arithmétiques et Logiques, les instructions de comparaison, les instructions de conversion en format flottant, les opérations de décalage et rotation, les instructions de manipulation de chaîne de caractères, les instructions de gestion de piles et les instructions de contrôle du séquencement.

ADDD	add. flottant double	MOVWP	move mot/reg. processeur
ADDI	add. immédiat	MOVWS	move mot/simple
ADDQ	add. rapide	MULD	multiplication double flottant
ADDS	add. flottant simple	MULS	multiplication simple flottant
ADDW	add. mot	MULW	multiplication mot
ADDWC	add. mot avec carry	MULWU	idem non signé
ANDI	ET logique immédiat	MULWUX	idem étendue
ANDW	ET logique mot	MULWX	multiplication mot étendue
B	branch. sur condition	NEGD	négation double flottant
BF	branch. cond. flottante	NEGS	négation simple flottant
CALL	appel de procédure	NEGW	négation mot
CALLS	appel système	NOOP	no operation
CMPC	comp. caractères	NOTQ	NON rapide
CMPD	comp. flottant double	NOTW	NON mot
CMPI	comp. immédiat	ORI	OU logique immédiat
CMPQ	comp. rapide	ORW	OU logique mot
CMPS	comp. flottant simple	POPW	dépile mot
CMPW	comp. mots	PUSHW	empile mot
CNVDS	conversion double/simple	RESTDn	restitution registres Fn
CNVDW	conv. double/mot	RESTUR	restitution reg. utilisateur
CNVRDW	idem ave arrondi	RESTWn	restitutions registres Rn
CNVRSW	conv. simple/mot arrondi	RET	retour de procédure
CNVSD	conv. simple/double	RETI	retour d'interruption
CNVSW	conv. simple/mot	ROTI	rotation immédiat
CNVTDW	conv. double/mot tronqué	ROTL	rotation mot long
CNVTSW	idem simple/mot	ROTLI	idem immédiat
CNVWD	conv. mot/double	ROTW	rotation mot
CNVWS	conv. mot/simple	SAVEDn	sauve registres Fn

Fig. 46 - Principales instructions du processeur Clipper C100 (1)

Ainsi, le décodage des instructions est câblé pour les instructions "simples" et microcodé sur une ROM (*Read Only Memory*) pour les autres. Le pipeline d'exécution du Clipper est basé sur un modèle à trois étages : le premier effectue la lecture de l'instruction et éventuellement la recherche du microprogramme correspondant, le second cycle décode les opérandes et génère les bits de contrôle, le

troisième cycle intègre lui-même un second pipeline à trois étages pour l'exécution de l'instruction. Ce dernier effectue la lecture des opérandes, exécute l'opération sur l'unité opérative sélectionnée, puis écrit le résultat dans le registre destination.

Le processeur Clipper est en fait une tentative pour combiner les avantages des architectures RISC et CISC. Le jeu d'instructions en est une illustration exemplaire avec ses 101 instructions câblées et ses 67 macros. Créé pour supporter efficacement Unix, le Clipper reste encore actuellement (1989) un des microprocesseurs RISC les plus vendus, en majorité d'ailleurs à la société Intergraph. En collaboration avec Intergraph, la Société Fujitsu travaille sur une version CMOS 20 MIPS ainsi qu'une version ECL de 60 MIPS entièrement compatibles.

DIVD	division double flottant	SAVEUR	sauve registres utilisateur
DIVS	division simple flottant	SAVEWn	sauve registres Rn
DIVW	division mot	SCALBD	scale double flottant
DIVWU	idem non signée	SCALBS	scale simple flottant
INITC	init. caractères	SHAI	décalage arithmétique immédiat
LOADA	load adresse	SHAL	décalage arithmétique long mot
LOADB	load octet	SHALI	idem immédiat
LOADU	idem non signé	SHAW	décalage arithmétique mot
LOADD	load double flottant	SHLI	décalage logique immédiat
LOADFS	load status flottant	SHLL	décalage logique long mot
LOADH	load demi-mot	SHLLI	idem immédiat
LOADHU	idem non signé	SHLW	décalage logique mot
LOADI	load immédiat	STORB	store octet
LOADQ	load rapide	STORD	store double flottant
LOADS	load simple flottant	STORH	store demi-mot
LOADW	load mot	STORS	store simple flottant
MODW	modulo sr un mot	STORW	store mot
MODWU	idem non signé	SUBD	soustraction double flottant
MOVEC	move caractères	SUBI	soustraction immédiat
MOVD	move double flottant	SUBQ	soustraction rapide
MOVDL	move double/mot long	SUBS	soustraction simple flottant
MOVLD	move long mot/double	SUBW	soustraction mot
MOVPW	move reg. proc./mot	SUBWC	idem avec carry
MOVS	move simple flottant	TRAPIN	trap si flottant non ordonné
MOVSU	move super./utilisateur	TSTS	exclusion mutuelle
MOVSW	move single/mot	WAIT	attente d'interruption
MOVUS	move utilisateur/super.	XORI	OU exclusif immédiat
MOVW	move mot	XORW	OU exclusif mot

Fig. 47- Principales instructions du processeur Clipper C100 (2)

7. Le processeur 80960 de Intel

Comme ses principaux compétiteurs, la Société Intel se devait de proposer une alternative RISC à la famile 286/386/486. Bien que l'on puisse discuter l'appellation de processeur RISC pour son produit, Intel annonça la disponibilité du 80960 pour les applications spécifiques [75]. Le 80960 est un processeur 32 bits réalisé en CHMOS 1.5 micron sur une puce de 350000 transistors. A 20 Mhz, il est capable d'exécuter 7.5 MIPS. Le processeur d'Intel intègre 32 registres de 32 bits généraux et quatre registres au format flottant. Vingt-huit de ces registres ne sont pas affectés à une fonction particulière, les autres étant pointeur de pile (*stack pointer*) et différents pointeurs pour les appels et retours fonctionnels (*frame pointer, previous frame pointer, return instruction pointer*). Les registres généraux sont divisés en deux groupes : 16 registres globaux et 16 registres locaux qui sont remplacés par un nouveau banc lors d'un appel de procédure et restitués lors d'un retour. Contrairement à la plupart des processeur RISC tirant parti du fenêtrage, les "fenêtres" du 80960 ne se recouvrent pas pour faciliter le passage des paramètres et des résultats.

Ainsi, le processeur 80960 comprend un "cache" de 64 registres correspondant à 4 fenêtres de 16 registres. Les versions futures pourraient intégrer jusqu'à 192 de ces registres pour un total de 12 fenêtres. Le 80960 est caractérisé par un jeu de 184 instructions dont seulement un noyau de 51 opérations sont véritablement RISC (*RISC Core Instruction Set*). Les instructions du 80960 sont codées sur seulement cinq formats : le format de contrôle (CTRL), pour les opérations de contrôle de séquencement, deux formats mémoires (MEMA et MEMB) pour les instructions *load* et *store*, un format spécial pour les instructions de comparaison-branchement (COBR) et un format registre (REG) pour l'ensemble des opérations arithmétiques et logiques. Conformément à la méthodologie RISC, le 80960 est une machine à 3 opérandes : deux sources et une destination, sélectionnées parmi l'ensemble des registres du processeur.

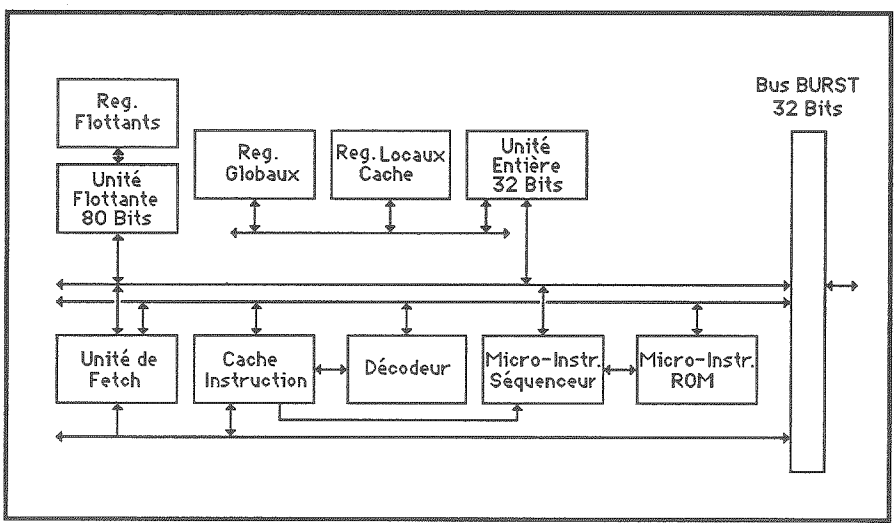


Fig. 48 - Synoptique du processeur Intel 80960

Comme le Clipper, le décodage du 80960 est basé sur une classique technique de microcodage (sur une ROM de 3K x 42 bits). De ce fait, la majorité des instructions s'exécute en plus d'un cycle machine. De la même manière, le processeur d'Intel comporte à la fois les modes d'adressages RISC et d'autres beaucoup plus complexes.

L'architecture du 80960 est basée sur un modèle pipeline entièrement géré par le matériel, qui, encore une fois, tire parti de la technique de *scoreboarding* pour tenter d'exécuter plusieurs instructions en parallèle. Ainsi, aucun optimiseur spécifique n'est nécessaire du fait de la gestion automatisée des conflits. Là encore, le 80960 est très éloigné des choix préconisés tant à Berkeley qu'à Stanford.

En plus de la classique unité entière, le processeur intègre une unité de calcul IEEE-754 de 80 bits et un cache d'instructions de 512 octets. Un unique bus de 32 bits permet d'accéder à la mémoire, doublé par 32 bits d'adresses qui permettent d'adresser un espace linéaire (!) de 4 Gigaoctets.

Les ingénieurs d'Intel avouent que le 80960 n'est pas à proprement parler un véritable processeur RISC, mais une architecture hybride visant le marché des microcontrôleurs [76]. De fait, le 80960 est beaucoup plus CISC que RISC, si ce n'est au niveau du noyau d'instructions "CORE RISC". Présenté comme une nouvelle génération

de microcontrôleur, qui fait suite aux 8048/8051/8096, le 80960 semble une timide approche vers le RISC. Cet état de fait est probablement dû à un désir de ne pas concurrencer ses principaux chevaux de bataille, le 80486 et le i860.

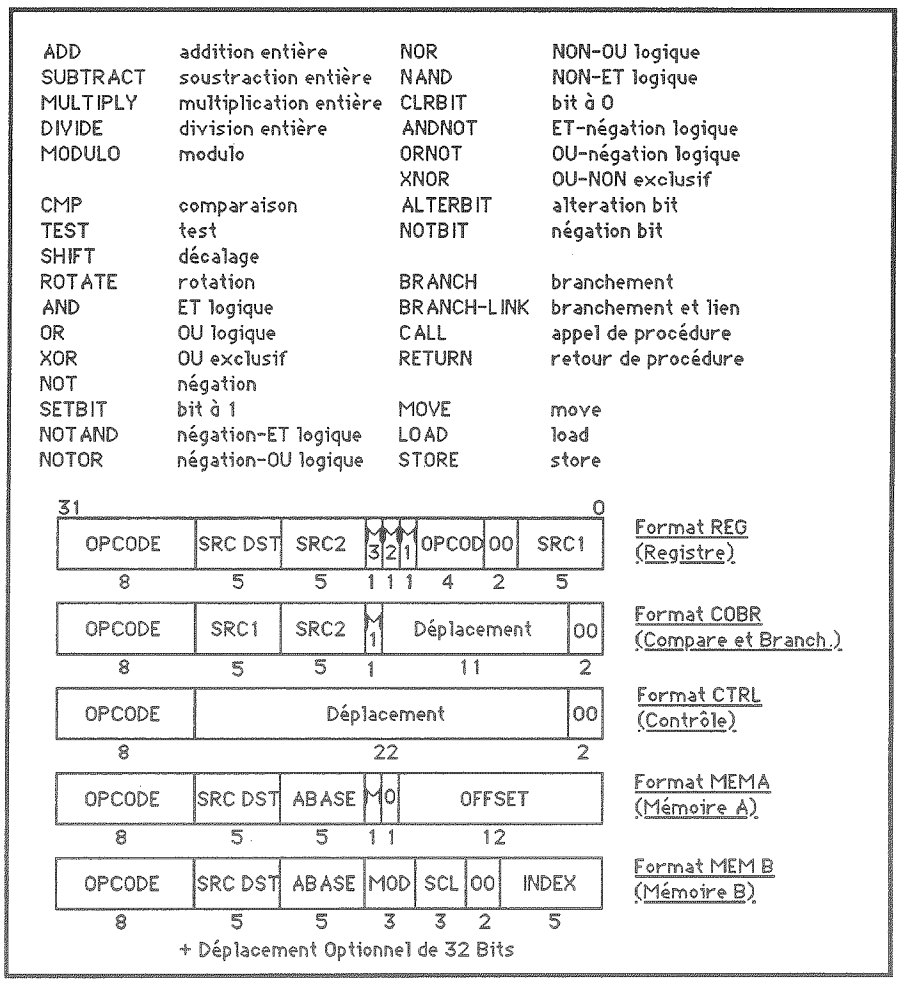


Fig. 49 - Formats et noyau d'instructions RISC du 80960

La Société Intel a également annoncé dernièrement (été 1989) une version "super-scalaire" du 80960, baptisée 80960CA. Ce nouveau processeur, entièrement compatible avec les versions antérieures, permet d'exécuter en moyenne deux instructions par cycle (de 1 à 3 en pratique), pour atteindre une puissance théorique de 66 MIPS

à 33 Mhz. En plus de la logique déjà présente dans l'ancienne version, le 80960CA intègre quatre canaux DMA et un système sophistiqué de contrôle des interruptions (possibilité de gérer 248 sources externes). Le i80960CA, d'une capacité de 500000 transistors, représente le premier circuit VLSI, disponible sur le marché, affirmant une orientation "super-scalaire" [77].

8. Le processeur i860 d'Intel

Pour le domaine des stations de travail, Intel a présenté récemment une nouvelle architecture baptisée i860, intégrée sur une seule puce de silicium contenant plus d'un million de transistors [78]. Le i860 tire parti, comme le 88000 et le Clipper, de l'architecture du Cray-1, en intégrant autour d'un coeur RISC, deux caches, une unité de gestion mémoire et une unité virgule flottante de 64 bits.

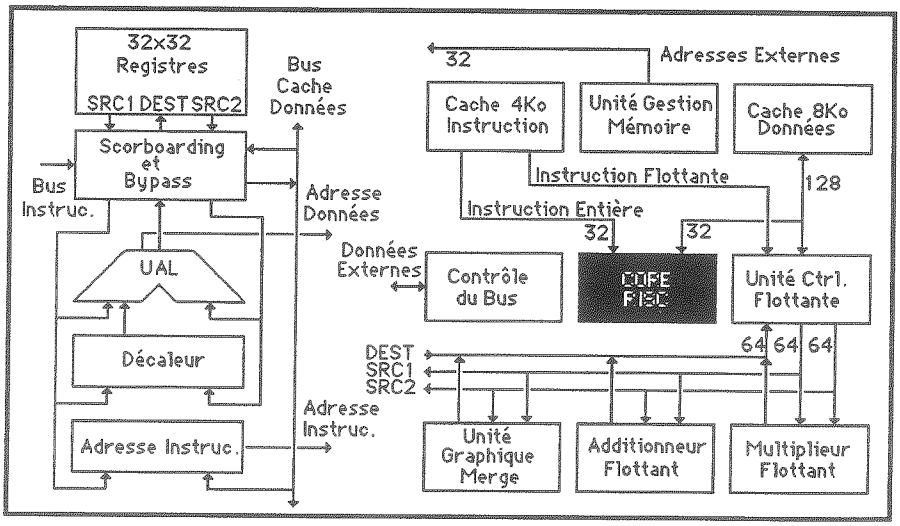


Fig. 50 - Synoptique du processeur i860 et de son coeur RISC

A 40 Mhz, le processeur i860 vise une puissance de 40 MIPS et 80 Mflops grâce à la mise en oeuvre de la technique de *scoreboarding* qui lui permet d'éliminer les temps morts qui se produisent lorsque l'on doit charger un registre à partir de la mémoire externe. Dans ce cas, le *scoreboarding* fournit la possibilité de travailler avec d'autres registres pendant l'accès à la mémoire. Contrairement au

80960, le i860 est beaucoup plus RISC et vient directement concurrencer le processeur de Motorola sur son propre terrain.

LD.X	load entier	AND	ET logique
ST.X	store entier	ANDH	ET logique haut
FLD.Y	load format flottant	ANDNOT	ET NON logique
PFLD.Z	idem pipeliné	ANDNOTH	idem haut
FST.Y	store format flottant	OR	OU logique
PST.D	store pixel	ORH	idem haut
		XOR	OU exclusif
IXFR	entier vers registre FP	XORH	idem haut
FXFR	FP vers registre entier		
		TRAP	exception logicielle
ADDU	addition non signée	INTOVR	trap sur overflow entier
ADDS	addition signée	BR	branchement direct
SUBU	soustraction non signée	BRI	branchement indirect
SUBS	soustraction signée	BC	branchement conditionel
		BC.T	branchement sur CC "taken"
SHL	décalage à gauche	BCN.T	branchement sur CC "not taken"
SHR	décalage à droite	BTE	branchement si égal
SHRA	décalage droite arithm.	BTNE	branchement si non égal
SHRD	décalage droite double	BLA	branch. sur LCC et addition
		CALL	appel de procédure
FLUSH	"flush" du cache	CALLI	appel indirect de procédure
LD.C	load depuis reg. de ctrl.		
ST.C	store vers reg. de ctrl.	FADD.P	addition flottante
LOCK	séquence verrouillée	PFADD.P	idem pipelinée
UNLOCK	déverrouillage	FSUB.P	soustraction flottante
		PFSUB.P	idem pipelinée
FMUL.P	multiplication flottante	PFGT.P	comparaison FP pipelinée GT
PFMUL.P	idem pipelinée	PFEQ.P	idem égalité
PFMUL3.DD	idem sur 3 étages	FIX.P	conversion entière
FMLOW.P	multiplication flot. basse	PFIX.P	idem pipelinée
FRCP.P	réciproque flottant	FTRUNC.P	conversion entière tronquée
FRSQR.P	idem racine carré	PFTRUNC.P	idem pipelinée
PFAM.P	add. et mul. FP pipelinée	FZCHKS	test tampon Z 16 bits
PFSM.P	sub. et mul. FP pipelinée	PFZCHKS	idem pipeliné
PFMAM	mul. et add. FP pipelinée	FZCHKL	test tampon Z 32 bits
PFMSM	mul. et sub. FP pipelinée	PFZCHKL	idem pipeliné
		FADDP	addition avec merge pixel
FISUB.Z	soustraction entier long	PFADDP	idem pipeliné
PFISUB.Z	idem pipelinée	FADDZ	addition avec merge Z
FIADD.Z	addition entier long	PFADDZ	idem pipeliné
PFIAD.Z	idem pipelinée	FORM	OU avec merge registre
		PFORM	idem pipeliné

Fig. 51 - Le jeu d'instructions du processeur i860

En effet, le processeur i860 est basé sur un jeu (relativement) réduit d'instructions. Au nombre de 76, ces instructions sont réparties en 12 classes : les instrucions *load* et *store*, les instructions *move*, les opérations arithmétiques, logiques et de décalage, les instructions de contrôle du séquencement, de contrôle du cache, les opérations de multiplication virgule flottante, addition virgule flottante, double-opérations flottantes, de calcul sur des entiers longs et enfin les instructions graphiques.

L'i860 intègre sur une seule puce pratiquement toutes les fonctionnalités nécessaires à la réalisation d'une station de travail graphique. A 40 Mhz, il est prévu pour éxécuter 40 MIPS crête, 80 Mflops en simple précision et 50000 dégradés de couleurs par seconde au niveau du triangle. Le circuit a été conçu dans une technologie CHMOS 1 micron. Il repose sur une architecture Harvard 32 bits pour le coeur RISC et une architecture 64 bits pour la partie flottante et graphique. En outre, l'i860 intègre un cache mémoire de 4 Koctets pour les instructions et un cache mémoire de 8 Koctets pour les données. L'architecture a été conçue pour permettre l'exploitation de plusieurs opérateurs. Ainsi, comme son homologue 80960CA dédié au contrôle, le processeur i860 peut dans certains cas exécuter plus d'une instruction par cycle machine.

9. Le processeur VL86C010 de VLSI Technology

Le processeur VL86C010 est une version de la machine RISC Acorn (ARM), fabriqué pour les applications spécifiques [79]. Il est également disponible chez le même constructeur en tant que méga-cellule pour la réalisation de circuits plus complexes organisés autour d'un coeur RISC. A 12 Mhz, le processeur VL86C010 est capable d'exécuter près de 6 MIPS. Le processeur ARM comprend un banc de 27 registres dont seulement 16 sont réellement utilisables en tant que registres généraux. Les registres restants sont dédiés à des fonctions spécifiques comme le compteur de programme (PC) et autres registres de contrôle.

Le jeu d'instructions comporte cinq principaux types d'opérations : les instructions de traitement de données, les instructions de transfert, les opérations de transfert de blocs de données, les

instructions de branchement non-retardé et les interruptions logicielles.

Chaque instruction contient un champ condition de 4 bits qui permet de transformer l'instruction en *No-Operation (skipped instruction)* si la condition spécifiée n'est pas vérifiée. La plupart des instructions manipulent deux opérandes dont la première est toujours un registre. La seconde opérande peut être un registre, initialement décalé, ou une valeur immédiate de 8 bits avec une capacité de rotation.

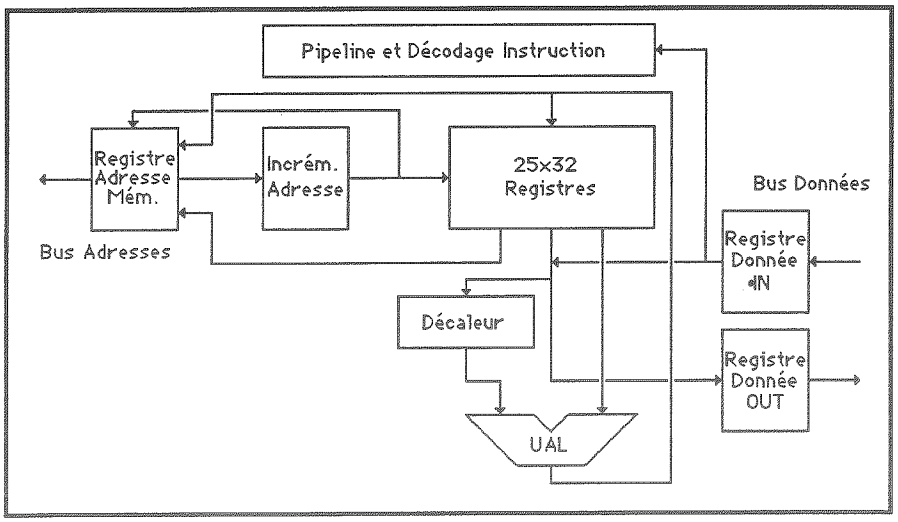


Fig. 52 - Synoptique du processeur VL86C010

L'ensemble du jeu d'instructions est codé sur un format fixe de 32 bits selon 11 formats prédéfinis. L'architecture du VL86C010 est une pure structure RISC basée sur un pipeline d'exécution à 3 étages. Le processeur est caractérisé par un bus de donnée de 32 bits et un bus d'adresse de 26 bits.

Du fait de sa simplicité et de son coût, le processeur ARM est une très bonne solution pour les applications spécifiques qui ne nécessitent pas une puissance de calcul importante ou une architecture sophistiquée. Malgré une audience encore faible, il reste un compétiteur à ne pas négliger, surtout pour les futures versions qui fonctionneront de 20 à 40 Mhz en technologie CMOS 1 micron. Outre l'unité centrale, VLSI Technology commercialise également un contrôleur de mémoire (MEMC VL86C110), un

contrôleur vidéo (VIDC VL86C310) et un contrôleur d'entrée-sortie (IOC VL86C410) [80].

La Société VLSI Technology a récemment annoncé une seconde génération du processeur ARM, baptisé VL86C020, réalisé dans une technologie CMOS 1.6 micron et fonctionnant à une fréquence de 20 Mhz. Une version 32 Mhz 1 micron est également planifiée, intégrant une mémoire cache de 4 Koctets.

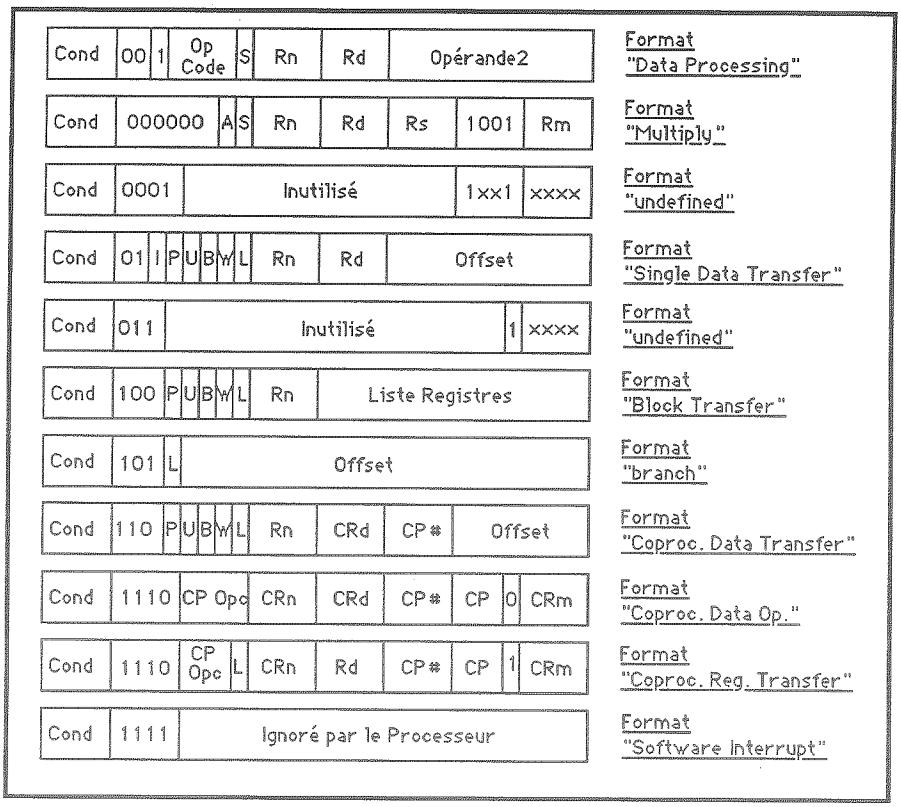


Fig. 53 - Les formats d'instructions du processeur VL86C010

10. Le processeur Transputer T800 d'Inmos

Le Transputer est, dans la famille RISC, un cas très particulier. Est-ce bien un processeur RISC d'ailleurs ? La gamme Transputer

comprend essentiellement trois modèles : l'IMS T212 à architecture 16 bits de 20 MIPS crête, l'IMS T414 à architecture 32 bits de 20 MIPS crête et enfin l'IMS T800 à architecture 32 bits de 33 MIPS crête [81].

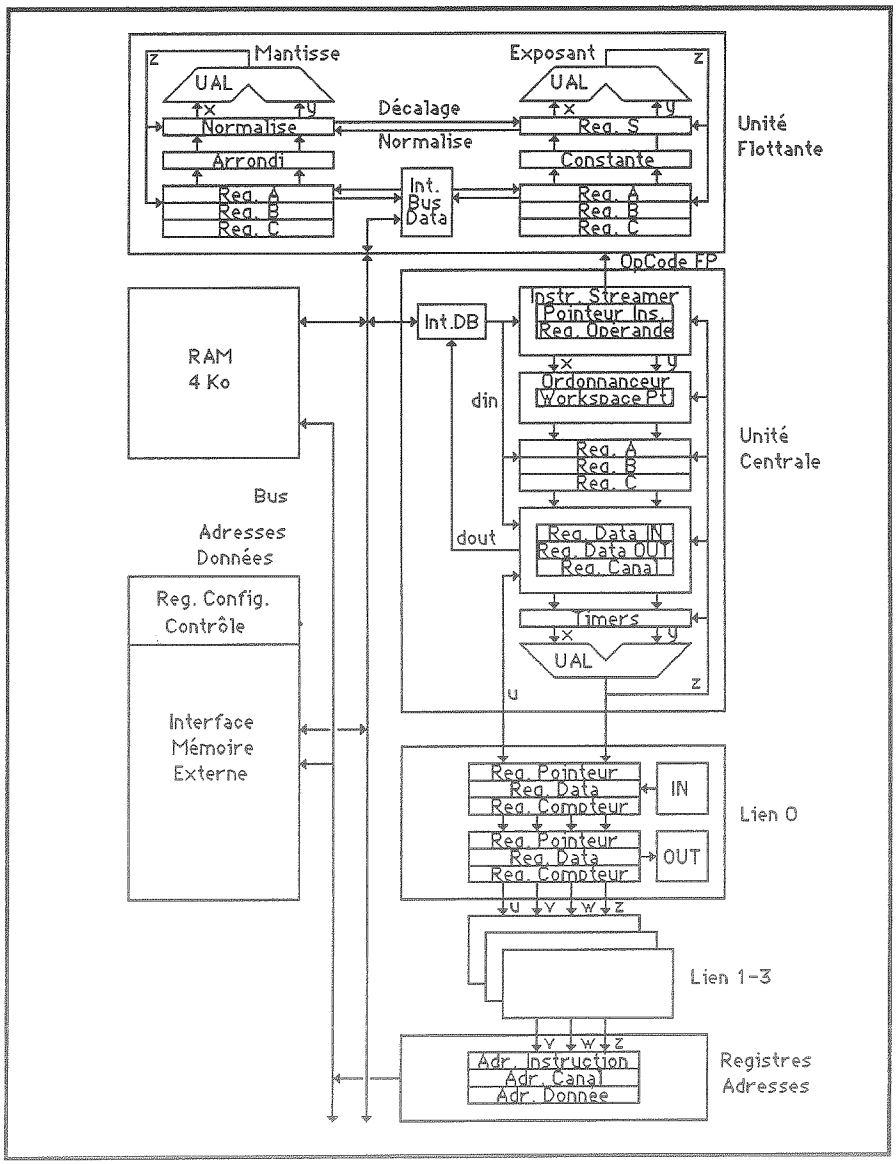


Fig. 54 - Synoptique du processeur IMS T800

Le T800 est le dernier-né des Transputer. Outre une fréquence de fonctionnement accrue et une mémoire interne sensiblement plus importante (4 Koctets contre 2 Koctets), il intègre une unité de calcul virgule flottante au format IEEE-754. Pour un cycle interne de 33 nanosecondes, il est capable d'exécuter 15 MIPS et 2,25 Mflops.

Le T800 ne comprend que six registres accessibles par l'utilisateur : le pointeur d'espace de travail, le pointeur d'instruction, le registre opérande et trois registres (A, B et C) qui forment une pile d'évaluation. Ces trois derniers registres servent de sources et de destinations pour la plupart des instructions.

Le jeu d'instructions a directement été conçu pour l'exécution du langage de haut-niveau OCCAM. Les instructions les plus fréquentes sont codées sous la forme d'un octet divisé en deux champs de quatre bits. Le premier champ représente le code opération et le second une donnée. Le jeu d'instructions comprend 162 opérations réparties en 20 groupes. Ceux-ci sont les suivants : les instructions d'appel de fonction et de branchement, arithmétiques et logiques, arithmétiques "longues", générales, mouvement de blocs 2D CRC et manipulation de bits, d'indexation et de manipulation de tableaux, de contrôle du timer, d'entrées-sorties, de contrôle, d'ordonnancement, de traitement d'erreur, d'initialisation du processeur, de chargement et stockage en format flottant, flottantes générales, d'arrondi en format flottant, de traitement d'erreur en format flottant, de comparaison en format flottant, de conversion et enfin d'arithmétique flottante.

Le décodage des instructions est microprogrammé sur une mémoire morte interne. L'exécution d'une instruction peut prendre jusqu'à plusieurs dizaines de cycles selon l'opération à effectuer. La principale caractéristique des processeurs Transputer réside dans les quatre liens de communication permettant l'échange de données entre processeurs dans une configuration parallèle. Dans ce cas, les transferts de données se font à la cadence de 20 Mbit/seconde sur chacune des quatre liaisons séries asynchrones bidirectionnelles. Bien qu'ayant débuté timidement, le Transputer est actuellement largement utilisé. Son principal marché reste cependant celui de la recherche et, plus particulièrement, celui du parallélisme du fait même de ses caractéristiques.

J	saut	AND	ET logique
LDLP	load pointeur local	OR	OU logique
PFIX	préfixe	XOR	OU exclusif
LDNL	load non local	NOT	NON logique
LDC	load constante	SHL	décalage à gauche
LDNLP	load pointeur non local	SHR	décalage à droite
NFIX	préfixe négatif	ADD	addition
LDL	load local	SUB	soustraction
ADC	addition constante	MUL	multiplication
CALL	appel de procédure	FMUL	multiplication fractionnel
CJ	saut conditionnel	DIV	division
AJW	ajuste espace de travail	REM	reste
EQC	égale constante	GT	plus grand que
STL	store local	DIFF	différence
STNL	store non local	SUM	somme
OPR	opération	PROD	produit registre A
LADD	addition long mot	MOVE2DINIT	init. move 2D bloc
LSUB	soustraction long mot	MOVE2DALL	copie bloc 2D
LSUM	somme long mot	MOVE2DNONZERO	copie octets ≠0 bloc 2D
LDIFF	différence long mot	MOVE2DZERO	copie octets =0 bloc 2D
LMUL	multiplication long mot		
LDIV	division long mot	CRCWORD	calcul crc sur un mot
LSHL	décalage à gauche long	CRCBYTE	idem sur un octet
LSHR	décalage à droite long	BITCNT	compte bit à 1
NORM	normalise	BITREVWORD	inverse les bits
		BITREVNBITS	inverse N bits LSB
REV	inverse		
XWORD	étend au mot	BSUB	subscript octet
CWORD	test du mot	WSUB	subscript mot
XDBLE	étend à un double	WSUBDB	idem double
CSNGL	test simple	BCNT	compte octet
MINT	entier minimum	WCNT	compte mot
DUP	duplique sommet de pile	LB	load octet
		SB	store octet
IN	input message	MOVE	move message
OUT	output message		
INB	input octet	LDTIMER	load timer
OUTB	output octet	TIN	timer input
ALT	alt start	TALT	timer alt start
ALTWT	alt wait	TALTWT	timer alt wait
ALTEND	alt end	ENBT	autorise timer
ENBS	autorise skip	DIST	inhibe timer
DISS	interdit skip		
RESETCH	initialisation canal	STARTP	début process
ENBC	autorise un canal	ENDP	fin process
DISC	inhibe un canal	RUNP	exécution process
		STOPP	stoppe process
		LDPRI	load priorité courante

Fig. 55 - Le jeu d'instructions du processeur IMS T800 (1)

RET	retour de procédure	TESTPRANAL	test processeur analyse
LDPI	load pointeur instruction	SAVEH	sauve reg. queue haute pri.
GAJW	ajuste espace trav. gén.	SAVEL	idem basse priorité
GCALL	appel générale	STHF	store haute pri. pt. front
LEND	fin de boucle	STHB	idem pointeur back
		STIF	store base pri. pt. front
CSUB0	test subscript depuis 0	STLB	idem pointeur back
CCNT1	test compte depuis 1	STTIMER	store timer
TESTERR	test erreur fausse		
SETERR	set erreur	FPURN	mode arrondi au plus près
STOPERR	stoppe si erreur	FPURZ	mode arrondi vers zéro
CLRHALTERR	clear halt-on-error	FPURP	mode arrondi positif
SETHALTERR	set halt-on-error	FPURM	mode arrondi négatif
TESTHALTERR	test halt-on-error		
		FPCHKERROR	test erreur unité flottante
FPENTRY	entrée unité flottante	FPTESTERROR	idem avec clear
FPREV	FP inverse	FPUSETERROR	set erreur unité flottante
FPDUP	FP duplication	FPUCLRERROR	clear erreur unité flottante
FPLDNLSN	load simple non local	FPADD	addtion flottante
FPLDNLDB	load double non local	FPSUB	soustaction flottante
FPLDNLSNI	load simple indexé non loc.	FPMUL	multiplication flottante
FPLDNLDBI	idem double	FPDIV	division flottante
FPLDZEROSN	load zéro imple précision	FPUABS	valeur absolue flottante
FPLDZERODB	idem double précision	FPREMFIRST	1er pas reste
FPLDNLADDSN	load non local add. simple	FPREMSTEP	pas d'itération reste
FPLDNLADDDB	idem double précision	FPUSQRTFIRST	1er pas racine carré
FPLDNLMULSN	idem avec mult. simple	FPUSQRTSTEP	pas d'itération racine carré
FPLDNLMULDB	idem avec mul. double	FPUSQETLAST	dernier pas racine carré
FPSTNLSN	store simple non local	FPUEXPINC32	multiplication par 2**32
FPSTNLDB	idem double précision	FPUEXPDEC32	division par 2**32
FPSTNLI32	idem entier 32 bits	FPUMULBY2	multiplication par 2
		FPUDIVBY2	division par 2
FPUR32TO64	conversion réel 32/64		
FPUR64TO32	conv. réel 64/32	FPGT	plus grand que flottant
FPRTOI32	conv. réel/entier 32	FPEQ	égalité flottant
FPI32TOR32	conv. entier 32/réel 32	FPORDERED	flottant ordonné
FPI32TOR64	conv. entier 32/réel 64	FPNAN	flottant NaN
FPB32TOR64	conv. bits 32/réel 64	FPNOTFINITE	flottant infini
FPUNOROUND	conv. réel 64/32 sans arr.	FPUCHKI32	test rang entier 32 bits
FPINT	arrondi entier	FPUCHKI64	idem 64 bist

Fig. 56 - Le jeu d'instructions du processeur IMS T800 (2)

Si le transputer est loin d'être un pur processeur RISC, il en possède néanmoins quelques traits caractéristiques, dont l'approche orientée vers la compilation d'un langage de haut niveau et le faible nombre de formats d'instructions. Pour le reste,. le processeur est plutôt basé sur une architecture CISC conventionnelle.

11. Les autres processeurs RISC

11.1. Le processeur CRISP D'AT&T

Le processeur CRISP (*CISC and RISC Instruction Set Processor*) est une autre tentative pour lier les avantages des deux types d'architectures. Conçu par le géant AT&T, il est caractérisé par un jeu de 33 instructions de longueur 16, 64 et 94 bits. Le décodage est câblé comme dans la majorité des processeurs RISC. Le processeur CRISP intègre 32 registres de 32 bits qui fonctionnent comme un cache de pile. Le pipeline d'exécution comprend classiquement trois étages. Le processeur est intégré sur une puce CMOS de 1.74 micron fonctionnant à 16 Mhz pour une puissance effective de 10 MIPS [82].

11.2. Le processeur ROMP d'IBM

Etant reconnue comme à l'origine du concept RISC, IBM se devait de commercialiser une machine organisée autour d'un processeur RISC. Ce dernier s'appelle ROMP (*Research Office products division MicroProcessor*) [83]. Conçu principalement pour le calcul scientifique et l'exécution du système d'exploitation Unix, il est également à la base du micro-ordinateur PC RT. Le processeur ROMP est constitué de deux circuits VLSI en technologie NMOS 1.8 micron et comprend un total de 112 instructions 16 et 32 bits. Le décodage, bien qu'essentiellement câblé, comprend également 256 mots de microcode. Le processeur intègre 16 registres généraux et 10 registres spécialisés. Le pipeline de trois étages permet à 6 Mhz d'obtenir une puissance de 2 MIPS.

11.3. Le processeur Precision de Hewlett-Packard

Le premier constructeur de stations de travail propose une architecture, baptisée Precision [84], basée sur la méthodologie RISC. Il est actuellement utilisé dans les stations de travail SPECTRUM pour les applications scientifiques et, bien sûr, pour

l'instrumentation qui représentent un des marchés porteurs de la Société Hewlett-Packard. Le processeur Precision est intégré sur un seul circuit NMOS de 1.5 micron pour une complexité de 115000 transistors. Le jeu d'instructions comprend 140 instructions caractérisées par une taille fixe de 32 bits et un format fixe. Le Precision comprend 32 registres généraux et 32 registres de contrôle. Il est en outre conçu pour des applications multiprocesseur, ou pour être utilisé avec des coprocesseurs.

11.4. Le processeur 90x de Pyramid Technology

La Société Pyramid Technology commercialise, pour les applications scientifiques et graphiques sous Unix, un super-mini calculateur organisé autour d'un processeur RISC baptisé 90x [85]. Ce dernier est réalisé en technologie Schottky TTL et fournit une puissance de traitement de l'ordre de 3 MIPS pour un temps de cycle de 125 nanosecondes. Le processeur 90x, bien que microcodé, comprend 90 instructions de 32, 64 ou 96 bits dont le fameux branchement retardé. Il contient 16 fenêtres de 64 registres de 32 bits. L'architecture est organisée autour de deux unités principales : une unité de lecture d'instruction (*fetch unit*) et une unité d'exécution (*execution unit*).

11.5. Le processeur Ridge 32 de Ridge Computers

Le Ridge 32 fut le premier processeur RISC commercialisé pour servir d'unité centrale à une station de travail (cette station fut commercialisée en France par Bull sous la dénomination de SPS9) [86]. Le processeur est réalisé sur plusieurs circuits MOS VLSI et en logique STTL pour obtenir une performance d'environ 3 MIPS avec un temps de cycle de 125 nanosecondes. Comme le 90x, le décodage des instructions est microcodé. Il comprend 170 instructions sur 16, 32 et 48 bits qui manipulent 16 registres 32 bits avec fenêtrage et recouvrement. Le pipeline d'exécution est basé sur un modèle à quatre étages : *fetch* et décodage de l'instruction, *fetch* des opérandes, exécution de l'opération et écriture du résultat.

11.6. Et bien d'autres...

Au début de ce chapitre, nous avons décrit les principaux processeurs RISC commercialisés qui sont disponibles au niveau du composant, de la carte ou du système complet. Dans un second temps, nous avons effectué un rapide tour d'horizon des processeurs RISC qui ne peuvent se trouver qu'au coeur d'une station de travail. La liste ne s'arrête pas là. Devant la multitude des produits existants, il n'est pas possible de prétendre à l'exhaustivité. Néanmoins, nous citerons encore quelques processeurs avec leur référence, ce qui permettra au lecteur plus particulièrement intéressé par l'un d'entre eux de se procurer la documentation auprès du constructeur.

La Société Integrated Digital Products commercialise les processeurs Whestone I et Whestone II, réalisés en technologie ECL [87]. Schlumberger travaille sur le projet FAIM-I [88] qui vise la réalisation d'une machine parallèle pour les applications de l'Intelligence Artificielle, basée sur un processeur RISC réalisé dans leurs laboratoires. Il en est de même avec la Société Xerox, au sein du célèbre laboratoire de Palo Alto (Xerox PARC) qui étudie une machine parallèle baptisée DRAGON [89] reposant sur un jeu de VLSI CMOS à architecture RISC. La Société Celerity Computing réalise des cartes et des stations de travail sur la base du processeur ACCEL qui permet d'obtenir une puissance de 3 MIPS à 10 Mhz [90]. Citons également le processeur RISC DN10000 de la Société Apollo [91] et le processeur 16 bits Forth de la Société Novix à Cupertino en Californie [92]. Enfin, la Société SODIMA en France commercialise le processeur KIM20 pour les applications spécifiques de l'Intelligence Artificielle [93]. Nous reviendrons en détail sur ce processeur au cours du prochain chapitre qui servira d'illustation pratique aux propos tenus dans cet ouvrage. Nous oublions forcément un bon nombre de produits qui peuvent prétendre à l'appellation RISC, nous espérons que leurs concepteurs nous en excuseront.

Avant de présenter plus en détail l'architecture du processeur KIM20, le dernier paragraphe de ce chapitre récapitule l'ensemble des caractéristiques architecturales des processeurs évoqués, compare succinctement leurs performances et leurs domaines d'applications.

12. Un comparatif des différents processeurs RISC

12.1.Un récapitulatif des configurations

PROCESSEUR	ORIGINE	TECHNOLOGIE
IBM 801	IBM	carte processeur MECL 10K
RISC-I	U.C. Berkeley	VLSI NMOS
RISC-II	U.C. Berkeley	VLSI NMOS
SOAR	U.C. Berkeley	VLSI NMOS 4 µ
SPUR	U.C. Berkeley	jeu de VLSI CMOS 2 µ
MIPS	Stanford University	VLSI NMOS 2 µ
MIPS-X	Stanford University	VLSI CMOS 2 µ
CDC GaAs	Control Data & Texas Ins.	3 VLSI HIIL AsGa
McDD GaAs	McDonnell Douglas	2 VLSI E-JFET AsGa
RCA GaAs	RCA	ED-MESFET AsGa
KIM200	SODIMA	versions VLSI CMOS et AsGa
SPARC	SUN Microsys./Cypress	jeu de VLSI CMOS
R2000	MIPS Computer Systems	CPU et FPU VLSI CMOS
R3000	MIPS Computer Systems	CPU et FPU VLSI CMOS
Am29000	Advanced Micro Devices	VLSI CMOS 1.2 µ
MC88100	Motorola	CPU et cache/MMU VLSI CMOS
Clipper C100	Fairchild	VLSI CMOS sur microcarte
80960	Intel	VLSI CHMOS 1.5 µ
i860	Intel	VLSI CHMOS 1 µ
ARM VL86C010	Acorn Computer & VLSI Tech.	jeu de VLSI CMOS 1 à 2 µ
Transputer T800	Inmos	VLSI CMOS
CRIPS	AT&T	VLSI CMOS
Pyramid 90X	Pyramid Technology	carte Schottky TTL
Ridge 32	Ridge Computers	STTL et jeu de VLSI MOS
ROMP	IBM	2 VLSI 1.8 µ
Precision	Hewlett Packard	VLSI NMOS 1.5 µ
Whestone I&II	Integrated Digital Products	VLSI Bipolaire ECL
FAIM-1	Schlumberger Palo Alto	Jeu de VLSI CMOS Custom
Dragon	Xerox PARC	Jeu de VLSI CMOS 2 µ
KIM20	SODIMA	VLSI CMOS 1.5 µ

Fig. 57 - Les configurations étudiées

Ce paragraphe a pour objectif d'effectuer une synthèse des architectures présentées. Le premier tableau (figure 57) récapitule les processeurs évoqués dans ce chapitre, en donnant la référence du constructeur, la configuration et la technologie de réalisation employée.

12.2. Un comparatif architectural

PROCESSEUR	CONTROLE	REGISTRES	FENETRAGE	HARVARD
IBM 801	câblé	32	N	O
RISC-I	câblé	78	6 x (4+6+4)	N
RISC-II	câblé	138	8 x (6+10+6)	N
SOAR	câblé	72	4 x (8+8+8)	N
SPUR	câblé	138	8 x (6+10+6)	N
MIPS	câblé	16	N	N
MIPS-X	câblé	32	N	N
CDC GaAs	câblé	16	N	O
McDD GaAs	câblé	16+16	N	O
RCA GaAs	câblé	16	fenêtres variables	?
KIM200	câblé	32	N	N
SPARC	câblé	136	8 x (6+10+6)	N
R2000	câblé	32+2	N	N
R3000	câblé	32+2	N	N
Am29000	câblé	192	tout mode	O
MC88100	câblé	32+32	N	O
Clipper C100	câblé+macros	32+8	N	O
80960	µprogrammé	64	4 * 16	N
i860	câblé	32+32	N	O
ARM VL86C010	câblé	27	N	N
Transputer T800	µprogrammé	6	N	N
CRIPS	unité décodage	32	N	?
Pyramid 90X	µprogrammé	528	16 * 64	?
Ridge 32	µprogrammé	16	?	?
ROMP	câblé+µcode	16+10	N	?
Precision	PLA+µcode	32	N	?
Whestone I&II	câblé+µcode	4	N	?
FAIM-1	Mach. état fini	?	N	?
Dragon	PLA	?	?	?
KIM20	câblé	138	8 x (6+10+6)	O

Fig. 58 - Récapitulatif des caratéristiques RISC (1)

I	nombre d'instructions	**E**	exécution = 1 cycle
F	nombre de formats	**L/S**	architecture Load-Store
DB	branchement retardé	**PS**	nombre d'étages du pipeline

PROCESSEUR	I	F	DB	E	L/S	PS
IBM 801	120	2	N	0	0	2
RISC-I	31	2	0	0	0	2
RISC-II	39	2	0	0	0	3
SOAR	20	2	0	0	0	3
SPUR	38+25	?	0	0	0	4
MIPS	31	5	0	0	0	5
MIPS-X	40	4	0	0	0	5
CDC GaAs	29+37	?	0	0	0	6
McDD GaAs	<64	1	0	0	0	4
RCA GaAs	<64	4	0	0	0	5
KIM200	16	2	0	0	0	4
SPARC	64	6	0	0	0	4
R2000	74	3	0	0	0	5
R3000	74	3	0	0	0	5
Am29000	112	1	0	0	0	4
MC88100	51	?	0	0	0	4
Clipper C100	101+67	15	?	N	0	3
80960	51+133	4	?	N	N	?
i860	76	?	0	0	0	?
ARM VL86C010	44	11	N	0	0	3
Transputer T800	162	1	N	N	N	?
CRIPS	33	?	N	0	?	0
Pyramid 90X	90	?	0	N	0	3
Ridge 32	170	?	N	N	0	4
ROMP	112	?	0	N	0	3
Precision	140	?	0	0	0	5
Whestone I&II	181	?	?	N	?	3
FAIM-1	64	?	0	0	?	2
Dragon	≠150	?	N	0	?	?
KIM20	32	1	0	0	0	3

Fig. 59 - Récapitulatif des caratéristiques RISC (2)

Les deux tableaux précédents réalisent une synthèse des architectures RISC évoquées. Pour chaque processeur, nous reprenons les principales caratéristiques d'une architecture RISC.

12.3. Un comparatif des performances

PROCESSEUR	COMPLEXITE	CYCLE	CYCLE/INS.	MIPS
IBM 801	?	?	1.1	?
RISC-I	44.500 trs.	667 ns	≠ 1.4	≠ 1
RISC-II	41.000 trs.	330 ns	≠ 1.4	≠ 2
SOAR	35.000 trs.	360 ns	≠ 1.4	≠ 2
SPUR	?	150 ns	≠ 1.4	≠ 5
MIPS	25.000 trs.	250 ns	≠ 1.4	8
MIPS-X	150.000 trs.	50 ns	≠ 1.4	10
CDC GaAs	12.000 portes	5 ns	2.2	91
McDD GaAs	41.000 trs.	10 ns	?	100 crête
RCA GaAs	?	5 ns	?	100
KIM200	10.000 portes	5 ns	1.4	100
SPARC	50.000 trs.	25 ns	1.4	25
R2000	100.000 trs.	60 ns	1.4	12
R3000	115.000 trs.	40 ns	1.25	16
Am29000	?	40 ns	1.5	12
MC88100	165.000 trs.	40 ns	1.25	20
Clipper C100	?	30 ns	6.7	4.9
80960	350.000 trs.	50 ns	2.7	7.5
i860	>1.000.000 trs.	25 ns	1-3/cycle	40 crête
ARM VL86C010	≠ 25.000 trs.	83 ns	2	4
Transputer T800	>200.000 trs.	33 ns	1-48	15
CRIPS	172.000 trs.	62 ns	?	10
Pyramid 90X	-	125 ns	?	2-4
Ridge 32	-	125 ns	?	1-4
ROMP	111.000 trs.	170 ns	?	2
Precision	115.000 trs	33 ns	?	10
Whestone I&II	?	50 ns	?	5-13
FAIM-1	?	?	?	?
Dragon	?	100 ns	?	?
KIM20	53.000 trs.	62 ns	1.2	10

Fig. 60 - Récapitulatif complexités et performances

Le tableau de la page précédente résume pour chaque processeur sa complexité, soit en nombre de transistors soit en nombre de portes et sa performance. Il est bien entendu que ces chiffres ne sont en aucun cas des mesures précises, mais donnent un ordre de grandeur à titre informatif. En outre ces chiffres sont, pour la plupart, basés sur des produits évolutifs au niveau de la technologie et, par conséquent également au niveau des performances.

12.4. Une taxonomie par application

Le dernier tableau (figure 61) établit une classification des processeurs selon les deux principaux types d'applications :

1) les stations de travail, mini et micro-ordinateurs,

2) les microcontrôleurs pour les applications spécifiques.

Pour chaque catégorie, un processeur donné reçoit un signe + s'il est parfaitement adapté, un signe - dans le cas contraire. En outre, la colonne "RISC?" permet de déterminer de la même manière la tendance architecturale qui prédomine pour chaque processeur. Il est bien entendu que cette classification n'engage que les auteurs, car elle n'est fondée que sur leur expérience et les données (parfois maigres) fournies par les constructeurs.

PROCESSEUR	RISC?	STATION DE TRAVAIL	MICROCONTROLEUR
IBM 801	+	-	+
RISC-I	+	+	-
RISC-II	+	+	-
SOAR	+	+	-
SPUR	+	+	-
MIPS	+	+	-
MIPS-X	+	+	-
CDC GaAs	+	+	-
McDD GaAs	+	+	-
RCA GaAs	+	+	-
KIM200	+	+	+
SPARC	+	+	+
R2000	+	+	-
R3000	+	+	-
Am29000	-	+	+
MC88100	+	+	+
Clipper C100	-	+	-
80960	-	-	+
i860	+	+	-
ARM VL86C010	+	-	+
Transputer T800	-	+	-
CRIPS	+	+	-
Pyramid 90X	-	+	-
Ridge 32	-	+	-
ROMP	-	+	-
Precision	-	+	-
Whestone I&II	-	+	-
FAIM-1	?	+	-
Dragon	?	+	-
KIM20	+	-	+

Fig. 61 - Récapitulatif des tendances

4

Un exemple le processeur KIM20

1. Une architecture RISC pour l'Intelligence Artificielle

1.1. Objectif du projet KIM

L'avancée technologique permet de concevoir des systèmes de plus en plus sophistiqués, mais elle favorise également un accroissement de la complexité potentielle des applications. Cette dernière est surtout sensible dans les domaines militaires et spatiaux, où l'environnement est le plus souvent inconnu et hostile. Face à ces contraintes de plus en plus sévères, l'intelligence devient indispensable à la "survie" d'un système. L'intelligence est une défense naturelle, dans le sens où nous pouvons penser que les premiers comportements "intelligents" sont apparus comme des propriétés émergentes de la complexité, efficaces contre l'anéantissement et l'inévitable accroissement de l'entropie. Si l'avènement des techniques issues des travaux en Intelligence Artificielle permet de résoudre nombre de problèmes, celles-ci nécessitent généralement des moyens informatiques importants : la connaissance, méta-niveaux imbriqués d'information, est

performante mais son exploitation est coûteuse en temps et en ressources [94]. Ces problèmes peuvent être résolus, en partie, par l'utilisation de processeurs spécialisés, mieux adaptés à l'exécution du traitement symbolique.

Depuis 1970, de nombreux travaux ont permis d'expérimenter de nouvelles architectures informatiques, avec comme objectif principal la conception de stations de développement pour les langages Lisp ou Prolog. Ces machines sont particulièrement bien adaptées aux prototypages rapides d'applications, aux simulations et aux développements de logiciels où l'aspect "interface homme-machine" est primordial. Les machines américaines Explorer [95] et Symbolics [96] en sont des illustrations exemplaires. Il existe donc des machines spécialisées pour l'application des techniques de l'Intelligence Artificielle, mais elles sont toutes issues de travaux visant la conception de postes de travail performants et conviviaux, dont le premier souci semble être la productivité. Outre la difficulté à les intégrer dans un environnement informatique traditionnel, temps réel ou non, leurs spécificités les rendent inadaptées aux applications embarquées, du moins dans un contexte avionique ou spatial. Leur encombrement, leur poids, leur consommation et leur fiabilité sont autant d'obstacles à une utilisation plus large de ces machines et, par voie de conséquence, des techniques d'Intelligence Artificielle dans ces domaines.

Les progrès continuels en intégration VLSI et en architecture de machines, autorisent maintenant la conception de processeurs à la fois plus performants et plus compacts. Ces processeurs présagent une nouvelle génération de systèmes mieux adaptés aux contraintes opérationnelles des applications. C'est dans ce cadre, celui des machines cibles, que le projet KIM trouve sa place. Sans avoir la prétention de résoudre l'ensemble des problèmes, son objectif est néanmoins de fournir aux concepteurs de systèmes un ensemble d'outils, matériels et logiciels, permettant une meilleure intégration de "l'intelligence" dans les applications industrielles, spatiales et militaires.

1.2. Historique du projet KIM

En 1985, le thème du "contrôle symbolique" était largement étudié au laboratoire Système de Perception de l'ETCA (Etablissement Technique Central de l'Armement). Dans un premier temps, la

robotisation d'un laser 5 kW pour la découpe de tôle a montré l'intérêt des techniques issues des travaux en Intelligence Artificielle dans le domaine du contrôle [97]. Une extension de cette recherche a ensuite été menée dans le cadre du projet HECATE (Hôte Emulateur pour la Conception d'Automates de Traitements Embarqués). Ce projet ambitieux a pour objectif la réalisation d'une machine capable d'émuler en temps réel des automates de vision pour diagnostiquer leur fonctionnement et envisager leur intégration [98]. Chaque unité opérative (module) de cette machine est constituée de plusieurs cartes électroniques comprenant chacune environ cinq cent circuits intégrés. La multiplication des configurations d'opérateurs et de chemins de données potentiels, donnent pour chaque module plusieurs milliers de traitements différents possibles. A partir d'un état donné, l'explosion combinatoire des cas à envisager rend paradoxalement difficile l'implantation d'un procédé de traitement d'image. La difficulté ne réside pas seulement dans le choix et l'affectation des tâches aux opérateurs, mais également dans un maintien constant de l'organisation de ce gigantesque réseau dynamique. Ce type de problème n'est pas spécifique à la machine HECATE, il se retrouve dans tous les systèmes que nous élaborons dès lors qu'ils atteignent un certain niveau de complexité : centrale, avion, satellite par exemple.

C'est dans ce contexte que nous avons étudié un système exécutif temps réel basé sur des techniques d'Intelligence Artificielle et apte à résoudre les problèmes de contrôle. Ce logiciel, baptisé KOS (*Knowledge-based Operating System* [99]), a suscité la définition d'une machine virtuelle adaptée à l'exécution du traitement symbolique, pour les applications nécessitant une grande efficacité et des temps de réponse très rapides : KIM, un acronyme de *Knowledge-based Integrated Machine* [100].

Parallèlement, au laboratoire de Micro-électronique de l'Université d'Orsay (Institut d'Electronique Fondamentale Paris-Sud/CNRS), le processeur PEARLS (Processeur Expérimental Adapté à la Recherche sur les Langages Symboliques) était en cours de réalisation [111]. Devant le manque de véritables machines cibles pour mettre en application les techniques de l'Intelligence Artificielle, la Société SODIMA et l'IEF ont joint leurs compétences en 1986 pour concrétiser le projet KIM. A la fin de l'année 1987, les travaux aboutissaient à la fabrication d'une version câblée du processeur : plus de quatre cents composants CMOS sur une carte triple-europe

étendue, conforme au standard VME. Cette première phase du projet, soutenue par le Département du Val-de-Marne (premier prix de l'innovation et de la recherche 1987), l'ANVAR (Association Nationale pour la Valorisation de la Recherche) et la DRET (Direction de la Recherche Etudes et Techniques), validait l'architecture et son orientation "machine d'exécution" [102-104].

Cette étape préliminaire achevée, la Société SODIMA commençait, dès 1988, l'intégration du processeur en un circuit VLSI, dans une filière technologique ASIC CMOS. Cette seconde phase, financée en partie par la DRET et le SERICS (Service des Industries de Communications et de Services), s'est concrétisée par la conception du premier processeur symbolique RISC à vocation industrielle [105].

1.3. Caractères généraux du Traitement Symbolique

Qu'il s'agisse du système KOS (voir paragraphe précédent), du langage Lisp ou, plus généralement, de l'Intelligence Artificielle, ces langages ou techniques reposent directement sur le traitement symbolique. Alors que l'informatique conventionnelle est parfaitement adaptée au calcul numérique, l'Intelligence Artificielle nécessite la manipulation d'objets symboliques qui permettent la représentation des connaissances, puis son traitement, sous la forme d'inférences et de prises de décisions.

Le traitement symbolique, bien que complémentaire, diffère du calcul numérique essentiellement par la dynamique exercée. Alors que la manipulation de chiffres requiert généralement des structures de données statiques, connues à l'avance, le traitement symbolique impose des structures de données complexes et créées au cours de l'exécution.

Résumons brièvement en quatre points les principaux obstacles à une exécution efficace du traitement symbolique sur une architecture conventionnelle. Ceux-ci sont les suivants :

- nécessité de structures de données dynamiques,
- appels et retours fonctionnels fréquents et coûteux,

- le test des types de données est réalisé à l'exécution plutôt qu'à la compilation,
- la gestion de la mémoire doit être dynamique.

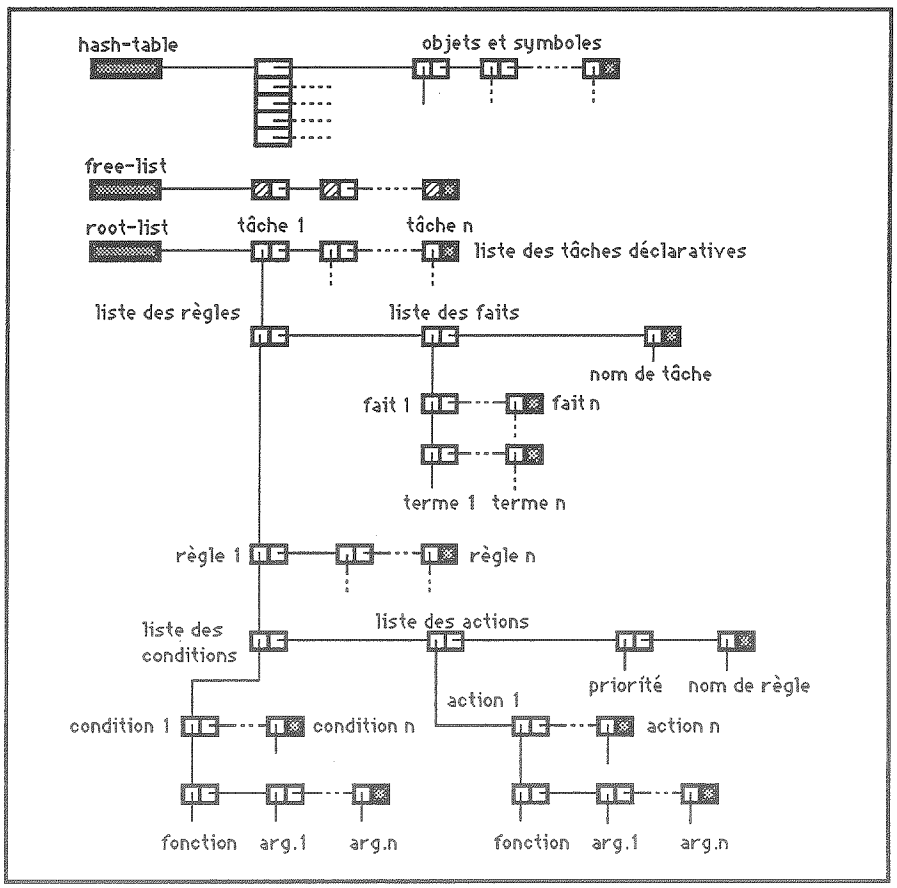


Fig. 62 - Exemple d'une structure de données dans un système I.A.

Le premier point fondamental concerne la manipulation de structures de données dynamiques. Le modèle Lisp [106-107], basé sur une représentation homogène des données et programmes sous la forme de listes et d'arbres, est universellement adopté. Mais la plus simple et répandue des primitives Lisp, comme la fonction "CAR" qui renvoie le premier élément d'une liste, nécessite plus d'une dizaine d'instructions pour une implantation sur un processeur classique comme le 68000.

En effet, le processeur de Motorola impose qu'une référence mémoire (pointeur) soit présente dans un registre d'adresses (A0-A7). Mais, dans ce cas, elle ne peut être manipulée aisément, car elle doit impérativement être stockée dans un registre de données (D0-D7).

Il s'ensuit un va-et-vient incessant des pointeurs lorsque l'on veut traiter des structures dynamiques avec ce processeur.

```
; Codage de la primitive CAR en Common-Lisp sur 68000
; la cellule CONS est initialement pointée par le registre A0

; calcul l'adresse de la fonction CAR en fonction du type

        move.l    a0,d2          ; copie dans d2
        lsl.l     #8,d2          ; décalage de 9 bits
        lsl.l     #1,d2          ; encore une fois
        and.l     #0x1F0,d2      ; on ne garde que le type
        move.l    d2,a2          ; doit être dans un reg. d'adresse
        jsr       CAR(a2)        ; saut indexé sur le type

; on arrive ici si le type est bien CONS, sinon les autres cas
; doivent être traités comme des ereurs

        movea.l   a0,d2          ; dans un registre de données
        and.l     #0xFFFFFF,d2   ; on masque le type
        movea.l   d2,a2          ; dans un registre d'adresse
        movea.l   (a2),a2        ; pointe le CAR

; à partir d'ici, on teste le type du CAR pour vérifier que l'on pointe
; bien sur un objet différent d'un UNBOUND ou pointeur invisible

        move.l    a2,d2          ; dans un registre de données
        lsl.l     #8,d2          ; décalage de 9 bits
        lsl.l     #1,d2          ; encore une fois
        and.l     #0x1F0,d2      ; on ne garde que le type
        move.l    d2,a3          ; dans un registre d'adresse
        jmp       TABLE(a3)      ; table de branchement

; enfin, si tout se passe bien, retour à l'appelant...
; l'ensemble a duré 128 cycles d'horloge, soit 18.2 µsec. à 10 Mhz

        rts                      ; bye
```

Fig. 63 - Inadéquation du 68000 pour la manipulation de pointeurs

Le second point concerne les appels et retours fonctionnels. La modularité et la récursivité sont deux caractéristiques fondamentales du traitement symbolique. De ce fait, comparés à des langages procéduraux classiques comme C ou Pascal, beaucoup plus d'appels et de retours de fonctions sont exécutés. On estime que la

plupart des dialectes Lisp passent entre 20 et 30% du temps d'exécution dans ce type de traitement. A l'Université de Berkeley, des mesures sur l'environnement Smalltalk ont montré que 50% du temps CPU était ainsi "consommé". En plus d'être fréquentes, ces opérations sont coûteuses. En effet, outre le branchement indexé sur le type de l'objet, un appel fonctionnel comprend également la sauvegarde du contexte courant (registres contenant les variables locales) et le passage des arguments. Corollairement, un retour fonctionnel doit passer les résultats et restituer le contexte de la procédure appelante.

Voyons maintenant le troisième point. L'exécution efficace et la flexibilité inhérente aux langages de l'Intelligence Artificielle dépend en grande partie du test dynamique des types de données (*dynamic type-checking*), technique de base pour l'implantation des fonctions génériques. L'ensemble de ces langages utilise le concept de fonction polymorphique où le type des opérandes varie et n'est effectivement déterminé qu'à l'exécution. Cette notion est parfois appelée *late-binding* par opposition aux tests de type réalisés par les compilateurs avant la génération de code et par conséquent bien avant l'exécution (*early-binding*). Un opérateur polymorphique est défini sur un ou plusieurs arguments dont les types peuvent être différents. Par exemple, une "addition générique" peut être appliquée sur les types suivants : mots binaires, entiers, flottants simple précision, double précision, entiers à précision variable, rationnels, complexes, vecteurs... Certains langages comme Lisp accentuent le problème en effectuant le test de type sur les valeurs et non sur les variables. Le test de type à l'exécution, communément appelé *run-time data-type checking*, est également à la base des langages et environnements orientés-objet [108]. Bien que cette technique soit indispensable, le calcul de la plupart des destinations pour l'appel de procédure en fonction du type des opérandes est pénalisant au niveau des performances sur un processeur non adapté.

Enfin, le dernier point concerne la nécessité d'une gestion dynamique de la mémoire. Qui dit structures de données dynamiques, doit dire également allocation rapide et récupération efficace des zones inutilisées. Puisque le traitement symbolique sous-entend des applications sophistiquées, il nécessite généralement une mémoire importante et un algorithme de restitution automatique performant. Ces programmes, plus souvent appelés *garbage-collector* (ramasse-miettes), font l'objet d'une

attention particulière depuis de nombreuses années (voir entre autres [109] pour un aperçu général des algorithmes employés). Mais les solutions préconisées sont encore loin d'être miraculeuses. En particulier, ce problème devient crucial dans le cas d'un système doté d'une mémoire virtuelle, car les performances des algorithmes actuels se dégradent proportionnellement avec la taille de la mémoire.

Pour tenter d'améliorer les performances, il suffit donc (théoriquement) de veiller à une implantation particulièrement soignée de ces mécanismes.

La définition du processeur KIM20 est basée sur ce principe : une architecture RISC parfaitement adaptée pour exécuter efficacement le traitement de listes, les appels et retours fonctionnels, le test de type à l'exécution, l'allocation et la restitution de cellules mémoires.

1.4. Le processeur KIM20

La machine virtuelle KIM a donc pour objectif de rassembler, dans un jeu d'instructions simple et compact, le noyau de primitives nécessaire et suffisant pour exécuter efficacement les techniques utilisées en Intelligence Artificielle. Nous avons vu que la méthodologie RISC est parfaitement adaptée à une telle démarche. En particulier, la volonté de câbler les opérations à forte occurrence, plutôt que de les microprogrammer, permet de réduire le nombre de transformations nécessaires entre le niveau exécutif (les opérateurs câblés) et le niveau applicatif (les programmes). Nous préférons, en effet, du moins pour une pure machine cible, un modèle plus direct qui diminue le fossé sémantique entre l'application et la machine support. Nous pouvons résumer en deux points l'approche préconisée :

1) concevoir un modèle d'éxécution sur la base d'une analyse descendante, à partir des travaux effectués dans le domaine du contrôle symbolique (Chap. 2 § 1.2.),

2) appliquer la méthodologie RISC avec, comme idée directrice, de câbler les opérations à forte occurrence sur une architecture pipeline régulière.

Nous avons à plusieurs reprises insisté sur la volonté de concevoir une architecture adaptée aux applications versus développement. Autrement dit, le processeur KIM20 correspond plus à un "microcontrôleur symbolique" qu'à un CPU de station de travail. Pour cela, plusieurs critères de conception ont été ajoutés au cahier des charges, qui garantissent une meilleure adéquation aux contraintes des applications :

- le processeur KIM20 doit être compact pour pouvoir être éventuellement embarqué,

- il doit être robuste et capable de supporter des contraintes opérationnelles variées,

- l'aspect temps réel est primordial avec, en particulier, les mécanismes d'interruption et de commutation de contexte,

- il doit faciliter la conception d'architectures simples et efficaces, pour cela, son interfaçage doit être aisé,

- le processeur KIM20 doit tirer profit du parallélisme à tous les niveaux : entre opérateurs (microparallélisme) et entre processeurs (macroparallélisme).

Au niveau structurel, le processeur KIM20 hérite des travaux de recherche effectués à l'Université de Berkeley et plus particulièrement des projets RISC II et SOAR. Comme ces derniers, KIM20 est basé sur une architecture pipeline synchrone à trois étages (*fetch - read/compute - write*) et un chemin de données (*data-path*) organisé autour d'une Unité Arithmétique et Logique (ALU) et d'un banc de fenêtres de registres à recouvrement partiel. Contrairement à RISC II et SOAR, KIM20 repose sur une structure Harvard, c'est-à-dire caractérisée par des chemins distincts (bus) pour l'accès au code exécutable et aux données.

On peut donc caractériser KIM20 comme un microcontrôleur spécialement optimisé pour la mise en oeuvre des techniques issues des travaux de l'Intelligence Artificielle. De ce fait, ces principaux domaines d'applications couvrent le contrôle de processus complexes, la robotique, le diagnostic embarqué, etc., en environnement industriel, militaire, avionique ou spatial. Les paragraphes suivants décrivent en détail les caractéristiques du processeur qui illustrent un aboutissement de la méthodologie RISC.

2. Le modèle de programmation

2.1. Le format des données

Le processeur KIM20 permet la manipulation de deux types principaux de données, codées sur un mot de 32 bits. Le premier type de données correspond aux valeurs numériques entières, le second aux "cellules typées".

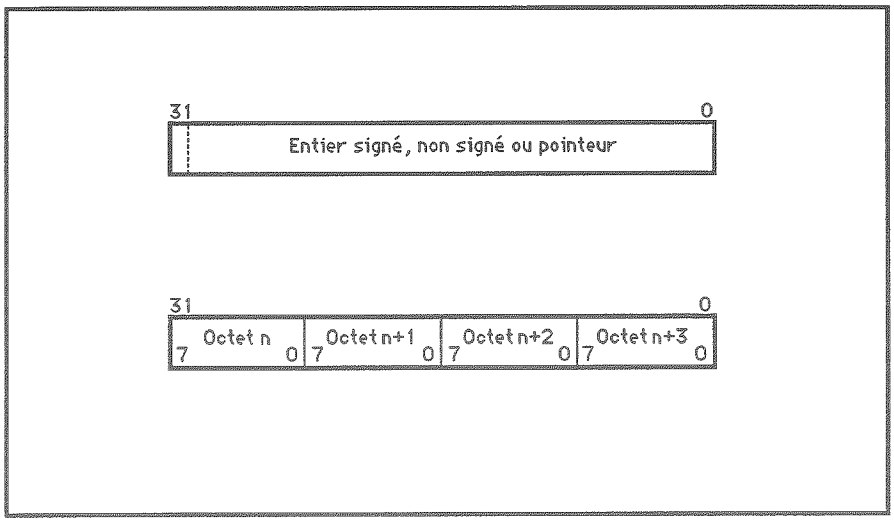


Fig. 64 - Format des données "entières"

Une valeur numérique entière est une donnée immédiate codée sur un mot de 32 bits, dont le bit de poids fort représente le signe. Ce format générique permet en outre le codage de l'ensemble des données classiquement utilisées : octets, chaînes de caractères, pointeurs, etc.

La cellule typée représente l'élément de base pour la structuration d'objets complexes. En effet, elle permet la construction de structures arborescentes, autorisant le codage d'informations structurées et dynamiques. Ce mode de représentation de l'information est largement dû à l'avènement du langage Lisp (*list-processing*).

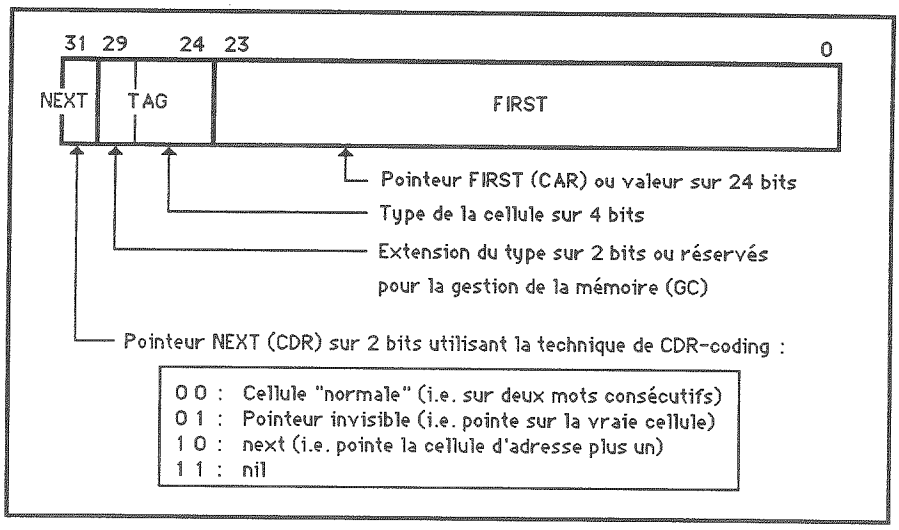


Fig. 65 - Format d'une cellule typée

Une cellule est composée de deux champs principaux. Le premier champ est appelé *first* et contient, sur 24 bits, soit l'adresse du premier élément de la liste (classique CAR du Lisp), soit une valeur immédiate. Le second champ est une étiquette (ou *tag*) qui permet de stocker plusieurs informations relatives à la sémantique de l'information que représente la cellule (i.e. son type) et à la gestion de la mémoire.

Le champ tag comprend 8 bits qui se répartissent ainsi :

1) un champ de 4 bits qui code le type de la donnée parmi 16 types prédéfinis,

2) deux bits distincts réservés pour l'algorithme de gestion de la mémoire ou une extension des types de base,

3) deux bits qui donnent l'adresse de la cellule suivante dans la liste (classique CDR du Lisp).

Le codage d'un champ CDR sur deux bits est issu des travaux sur la technique de *CDR-coding* expérimentée à Xerox Parc [110]. En effet, des statistiques montrent que deux bits suffisent pour 53.7 à 75.8% des cas (soit 61.3 en moyenne) et qu'un bon algorithme de récupération mémoire garantissant la linéarité permet d'atteindre 97% de réussite. Ce procédé permet ainsi le compactage des

structures de listes, ce qui représente un gain de place mémoire appréciable pour une machine cible et ne nécessite que peu de supports matériels spécifiques additionnels : principalement la détection automatique des exceptions au codage compact.

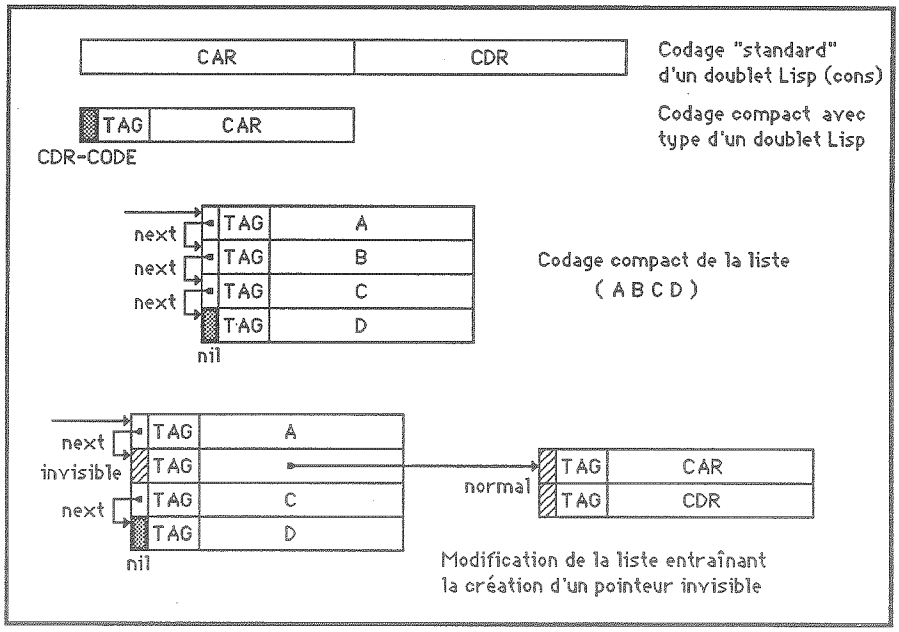


Fig. 66 - La technique du CDR-coding

Le codage du champ *CDR-code* retenu pour KIM20 est le suivant :

- Valeur 3 (CDR-NIL) :

  le pointeur vers l'élément suivant est égal au pointeur NIL qui indique que la cellule est la dernière de la liste.

- Valeur 2 (CDR-NEXT) :

  cette valeur représente l'adresse relative de l'élément suivant dans la liste, par rapport à la cellule courante. En l'occurrence, l'élément pointé est la cellule directement contiguë.

- Valeur 1 (CDR-CELL) :

  cette valeur signifie que la cellule courante est codée sur deux mots de 32 bits consécutifs. Les implantations des champs *first* et *next* sont alors dépendants du choix du concepteur de logiciel. Cette valeur provoque une exception lors de l'exécution d'une instruction *next* (équivalent au "cdr" du Lisp) ou *setn* (équivalent au "rplacd" du Lisp).

- Valeur 0 (CDR-INVISIBLE) :

  cette valeur donne une dernière possibilité d'adressage de l'élément suivant dans la liste. La cellule est alors appelée *invisible-pointer* ; son champ *first* doit contenir l'adresse de la cellule réelle. Cette valeur provoque une exception lors de l'exécution d'une instruction *next* ou *setn*.

2.2. Le format des instructions

Le jeu d'instructions de la machine KIM comprend 32 instructions. Conformément à la méthodologie RISC, elles ont toutes la taille d'un mot machine, c'est-à-dire 32 bits. Pour faciliter au maximum la recherche et le décodage des instructions, les 32 instructions sont codées sur un format unique. Celui-ci est composé de cinq champs indiquant le code opération, deux opérandes sources et une destination : KIM est donc une machine à trois opérandes. La syntaxe générale d'une instruction découle directement du format binaire.

Dans la syntaxe montrée en figure 67, *instruction* représente le mnémonique de l'opération à exécuter, "Rd" spécifie le registre destination parmi les 32 registres courants accessibles, "S1" spécifie de la même manière le premier registre source et "[#]S2" le second registre source ou une opérande immédiate de 16 bits si l'option "#" est active. Le jeu d'instructions est donc réduit et très homogène. En ce sens KIM est une pure architecture RISC. Il est caractérisé par un niveau syntaxique proche d'un assembleur conventionnel, mais sa sémantique le rapproche des primitives utilisées dans le noyau KOS. Ces caractéristiques en font un ensemble d'opérations adaptées à la programmation assembleur ou à la génération de code par compilateur.

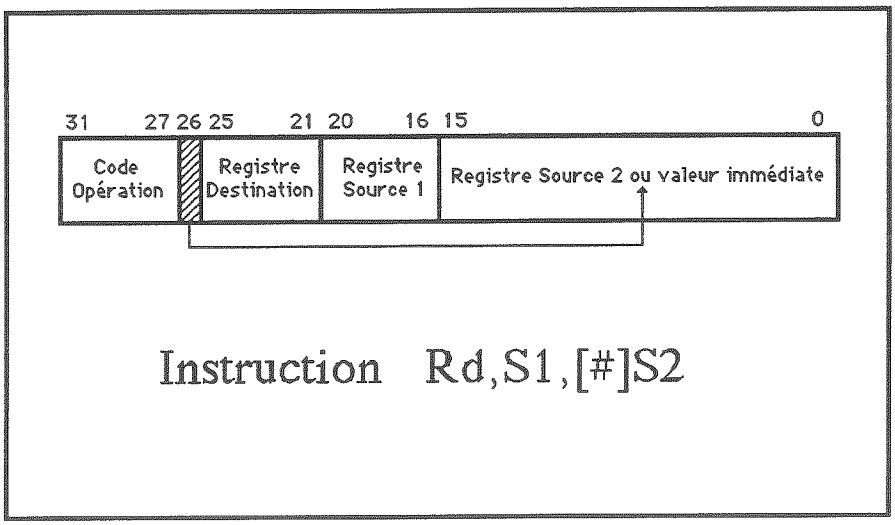


Fig. 67 - Format des instructions et syntaxe

2.3. Le jeu d'instructions

Le jeu d'instructions de la machine KIM peut être décomposé en quatre catégories dont les fonctionnalités visent à exécuter efficacement les mécanismes inhérents au traitement symbolique (Chap. 4 § 1.3).

- les opérations arithmétiques et logiques,.
- les opérations de contrôle de séquencement,
- les opérations de traitement de liste,
- les opérations d'accès particuliers aux mémoires.

Le premier groupe est basé sur le noyau d'instructions arithmétique et logique de RISC-II et de SOAR à quatre exceptions près : l'instruction *rotate* qui remplace avantageusement l'instruction de décalage à gauche de RISC-II, émulée simplement par l'instruction KIM "add rx,rx,rx" ; l'instruction *hash* qui permet d'accélérer les fonctions de "hash-coding" utiles en Lisp, Prolog, ou pour implanter des mécanismes d'unification efficaces (simulation de mémoire associative) ; l'instruction *hyper* sur laquelle nous allons revenir et enfin *case* qui constitue l'instruction permettant d'implanter les opérations polymorphiques par un calcul d'adresse en fonction du type des opérandes.

Instruction	Operands	Operation	Cycle
ADD	rd,s1,[*]s2	rd = s1+s2;	1
SUB	rd,s1,[*]s2	rd = s1-s2;	1
AND	rd,s1,[*]s2	rd = s1&s2;	1
OR	rd,s1,[*]s2	rd = s1 | s2;	1
XOR	rd,s1,[*]s2	rd = s1xors2;	1
ROTATE	rd,s1,[*]s2	rd = s1 shifted left by [*]s2 %32;	1
SHIFTRL	rd,s1	rd = s1 shifted right 1 bit; rd<31> = 0;	1
SHIFTRA	rd,s1	rd = s1 shifted right 1 bit; rd<31> = carry;	1
INSERT	rd,s1,[*]s2	rd = 0; rd<byte n°[*]s2<1-0>> = s1<7-0>;	1
EXTRACT	rd,s1,[*]s2	rd = 0; rd<7-0> = s1<byte n°[*]s2<1-0>>;	1
HASH	rd,[*]s2	rd = s2<31-24>+s2<23-16>+s2<15-8>+s2<7-0>;	1
HYPER	rd,s1,[*]s2	rd = 0; rd<4-0> = n°bit msb of (s1xor[*]s2);	1
CASE	rd,s1,[*]s2	rd = 0; rd<7-0> = s1<tag> & [*]s2<7-0>;	1
CALL	rd,*s2	rd = pc+1; pc = *s2; wp = wp+1;	1
RETURN		wp = wp-1; pc = pc(wp-1);	1
TRAP	*s2	pc = 0;	1
RETE	s1,[*]s2	pc = s1+[*]s2; wp = wp-1;	2
JUMP	*s2	pc = *s2;	1
BRANCH	s1,[*]s2	pc = s1+[*]s2;	2
COND	s1,[*]s2	if (test1 op test2 false) cancel next instruction;	1
FIRST	rd,s1,[*]s2	if ([*]s2<0> = 0) rd = (s1); else rd<first> = (s1)<first>;	1
NEXT	rd,s1	rd<tag> = (s1)<tag>; if (rd<cdr-code> = 3) rd<first> = 0; if (=2) rd<first> = s1+1; else trap;	1
SETF	s1,[*]s2	(s1)<first> = [*]s2;	1
SETN	s1,[*]s2	if ([*]s2 = 0) (s1)<cdr-code> = 3; if ([*]s2 = s1+1) (s1)<cdr-code> = 2; else trap;	1
WTAG	s1,[*]s2	(s1)<tag> = [*]s2<15-8> sélecté par <7-0>;	1
NEW	rd,[*]s2	rd = flp; flp = (rd)<first>; (rd)<tag> = [*]s2<15-8> sélecté par <7-0>; if (flc <= flt) trap; flc=flc-1;	1
FREE	s1	(s1)<tag> = 0; (s1)<first> = flp; flp = s1<first>; flc = flc + 1;	1
SEND	s1,[*]s2	(s1) = [*]s2;	1
PUSH	sp,[*]s2	sp = sp + 1; if (sp<7-0> = 0) trap; (sp) = [*]s2;	1
POP	rd,sp	rd = (sp); sp = sp -1; if (sp<7-0> = 0) trap;	1
LOAD	rd,s1	rd = (s1)<code>;	1
STORE	s1,[*]s2	(s1)<code> = [*]s2;	1

Fig. 68 - Résumé du jeu d'instructions KIM20

Le second groupe d'instructions est dédié au contrôle de séquencement. La plupart des principes utilisés dans ces instructions sont issus des travaux menés sur le projet SOAR de Berkeley (voir Chap. 1 § 4.3.). En particulier, nous avons retenu la technique du *cancelling branch* pour l'instruction *cond*, qui évite le désamorçage du pipeline d'exécution. De la même manière, les instructions *call* et *jump* sont limitées à une adresse de branchement immédiate, pour permettre le calcul de l'adresse de la

prochaine instruction à exécuter avant la fin du premier cycle du pipeline.

Le troisième groupe correspond au noyau d'instructions de traitement symbolique utilisé dans le logiciel KOS. On y retrouve les opérations *first* et *next* d'accès aux champs du même nom (assimilables aux CAR et CDR du Lisp), les opérations de modification *setf*, *setn* et *wtag* , assimilables pour les deux premières aux primitives RPLACA et RPLACD, l'instruction *new* implantant une opération "CONS" avec initialisation du *tag*, *free* qui restitue une cellule à la liste des commandes libres et, par conséquent, utile quel que soit l'algorithme de récupération utilisé.

Enfin, le dernier groupe d'instructions est constitué des opérations d'accès à la mémoire "programme" en lecture-écriture et à la mémoire de données en mode "pile lifo" (*last-in*, *first-out*) à forte occurrence en traitement symbolique. Une annexe donne, à la fin de l'ouvrage, une description détaillée de chaque instruction.

2.4. Organisation des registres

2.4.1. Introduction

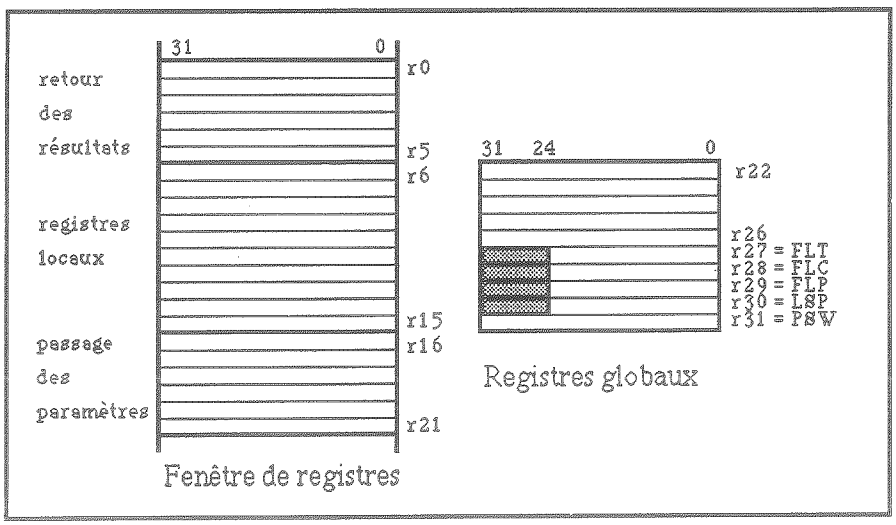


Fig. 69 - Le modèle de programmation KIM20

A tout moment, une fonction (composée d'une séquence d'instructions) peut accéder à 32 registres parmi les 138 intégrés dans le processeur.

Un ensemble de dix registres est inamovible et permet le stockage de données globales. Parmi eux, cinq sont banalisés et les cinq autres sont dédiés à des rôles particuliers.
Les vingt-deux autres correspondent à la fenêtre courante associée à la procédure en cours d'exécution. Le processeur KIM20 comprend huit fenêtres de ce type, recouvertes partiellement et organisées sous la forme d'une roue.

2.4.2. Les fenêtres de registres

Chaque fenêtre est constituée de vingt-deux registres de 32 bits. Les six premiers registres (r0-r5 ou x0-x5) et les six derniers (r16-r21 ou y0-y5) sont utilisés pour le passage des paramètres et le retour des résultats entre procédures. Les dix registres intermédiaires (r6-r15 ou l0-l9) sont réservés pour la manipulation des données locales à la procédure en cours d'exécution. Le processeur KIM20 intègre huit fenêtres identiques organisées sous la forme d'un buffer circulaire.

Deux champs particuliers du mot d'état (PSW) permettent l'identification de la fenêtre courante (WP - *Window Pointer*) et de la première fenêtre occupée (WO - *Window Overflow*).

Lors d'une instruction d'appel à une procédure (*call*), le pointeur de fenêtre WP est automatiquement incrémenté pour assurer le passage des paramètres, la sauvegarde des données locales et l'allocation d'un nouveau contexte local à la fonction appelée. Si, dans un tel contexte, le pointeur WP devient égal au pointeur WO, alors une exception *window overflow* est générée.

Lors d'une instruction de retour d'une procédure (*return*), le pointeur de fenêtre WP est décrémenté, pour assurer le retour des résultats et la restitution du contexte précédent. Si, avant le changement de contexte, le pointeur WP était égal au pointeur WO (avant décrémentation), alors une exception *window underflow* est générée.

Lors d'un débordement inférieur ou supérieur des fenêtres de registres, la tâche de sauvegarde ou de restitution des fenêtres est

à la charge du programme associé au traitement de l'exception (Chap. 2 § 3.2. et 4 § 4.2.).

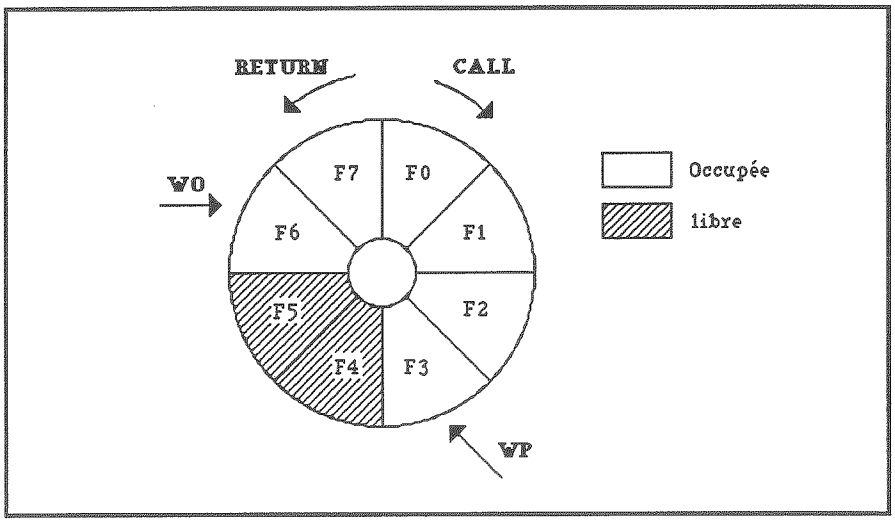


Fig. 70 - Les fenêtres de registres dans KIM20

2.4.3. Les registres globaux banalisés

Dix registres globaux sont accessibles constamment, quelque soit la fenêtre courante. De façon générale ces registres permettent le stockage des valeurs globales ou le traitement des exceptions. En fait, parmi ces dix registres, seulement cinq sont banalisés et peuvent être utilisés par le logiciel sans restriction particulière. Ces cinq registres sont r22 à r26 (ou g0-g4).

2.4.4. Le seuil de déclenchement du "garbage collector"

Le registre global r27 (FLT pour *Free-List-Threshold)* correspond au seuil de déclenchement de l'exception *Starvation*. Celle-ci est générée, par une instruction *New*, lorsque la valeur contenue dans r27 est supérieure ou égale à la valeur stockée, dans le registre global *Free-List-Count* (FLC), au début de l'instruction. Ce registre est constitué de 24 bits accessibles en lecture et en écriture.

2.4.5. Le compteur de cellules libres

Le registre global r28 (FLC pour *Free-List-Count*) correspond au compteur des cellules libres. Il indique à tout moment le nombre total de cellules libres chaînées dans la liste courante des cellules libres (*free-list*). Il est automatiquement mis à jour lors de l'exécution des instructions d'allocation (*new*) et de restitution (*free*) des cellules. Ce registre est constitué de 24 bits accessibles en lecture et en écriture.

2.4.6. Le pointeur de la liste des cellules libres

Le registre global r29 (FLP pour *Free-List-pointer*) correspond au pointeur de la liste courante des cellules libres. Il indique l'adresse de la première cellule libre. Il est automatiquement mis à jour lors de l'exécution des instructions d'allocation (*new*) et de restitution (*free*) des cellules. Ce registre est constitué de 24 bits accessibles en lecture et en écriture.

2.4.7. Le pointeur de pile

Le registre global r30 (LSP pour *Lifo Stack Pointer*) correspond au pointeur de pile courant. Lors de l'exécution d'une instruction *push*, ce registre est pré-incrémenté. Si l'octet de poids faible de ce pointeur devient égal à FF, une exception *stack overflow* est générée. Lors de l'exécution d'une instruction *pop*, ce registre est post-décrémenté. Si l'octet de poids faible de ce pointeur devient égal à FF, une exception *stack underflow* est générée. Ce registre est constitué de 24 bits accessibles en lecture et en écriture.

2.4.8. Le mot d'état

Le registre global (r31 ou PSW pour *Processor Status Word*) correspond au mot d'état du processeur KIM20. Celui-ci donne à tout moment les informations sur l'état du processeur.

Le mot d'état est composé de plusieurs champs aux fonctions particulières :

1) Les bits D'ALU (bits 31 à 28)

   Ce champ de 4 bits, accessible en lecture/écriture, donne pour chaque instruction exécutée les codes conditions résultant de la

dernière opération effectuée sur l'unité arithmétique et logique.

- Le bit 31 correspond à la *carry* qui est positionnée à 1 lorsque le résultat de l'opération dépasse le bit de poids fort des opérandes ;

- le bit 30 correspond à l'indicateur *zero* qui est positionné à 1 lorsque le résultat de l'opération est nul ;

- le bit 29 correspond à l'indicateur *negative* qui est positionné à 1 lorsque le bit de poids fort du résultat est à 1 ;

- le bit 28 correspond à l'indicateur *overflow* qui est positionné à 1 lors d'un débordement arithmétique, c'est-à-dire lorsque le résultat ne peut être correctement représenté sur le même nombre de bits que les opérandes.

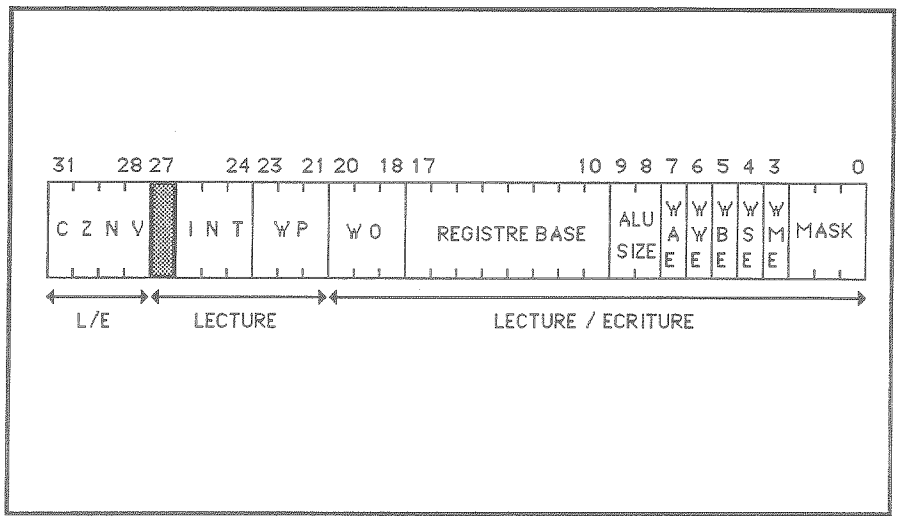


Fig. 71 - Le format du mot d'état PSW

L'Unité Arithmétique et Logique opère toujours sur des registres de 32 bits. Cependant, la taille effectivement considérée pour la mise à jour des bits d'ALU peut osciller entre 8, 16, 24 ou 32 bits, selon la valeur du champ ALU-SIZE (bits 8-9) du mot d'état. Dans les pages suivantes, on appellera plus simplement cette grandeur, la "taille de l'ALU".

2) Le bit O/U (bit 27) *Overflow/Underflow*

Ce bit spécifie le type de débordement rencontré lors des exceptions *window overflow*, *stack overflow* et *cdr_escape*. De façon générale, s'il est positionné à un, il indique un débordement supérieur (*overflow*), sinon il indique un débordement inférieur (*underflow*). Lors d'une exception *cdr-escape*, il indique si elle a eu lieu sur une instruction *next* (valeur 0) ou sur une instruction *setn* (valeur 1). Ce bit est accessible en lecture uniquement.

3) Les bits INT (bits 26 à 24) : *interrupt level*

Ce champ spécifie, le cas échéant, le niveau de l'interruption en cours de traitement. Il reste positionné à ce niveau jusqu'à la prise en compte logicielle de l'interruption. Après quoi, il est automatiquement remis à zéro. Le processeur KIM20 est doté de huit niveaux d'exception hiérarchisés. Ce champ est accessible en lecture uniquement.

4) Le champ WP (bits 21 à 23) : *Window Pointer*

Ces trois bits pointent sur la fenêtre courante de registres parmi les huit fenêtres disponibles du buffer circulaire. Ces bits, automatiquement mis à jour par les instructions de changement de contexte, sont accessibles en lecture.

5) Le champ WO (bits 18 à 20) : *Window Overflow*

Ces bits pointent sur la première fenêtre de registre utilisée et non encore sauvegardée. Lors des instructions de changement de contexte, *call* et *return*, ces bits sont comparés au pointeur courant (WP), pour la gestion des débordements. Ce champ est accessible en lecture et en écriture.

6) Le champ "registre BASE" (bits 10 à 17)

Ce champ de huit bits désigne la page courante de la mémoire code adressée par le compteur ordinal. Il est accessible en lecture et en écriture. Tout changement de page, en mémoire code, sera donc précédé d'une modification, explicite ou automatique (instructions *return/rete*), de ce registre dans le mot d'état. Lors d'une instruction *call*, il est sauvegardé en

même temps que le compteur ordinal, pour la restitution de l'adresse de retour de la procédure.

7) Le champ ALU-SIZE (bits 8 à 9)

Ce champ de deux bits détermine la taille virtuelle considérée par l'ALU pour la mise à jour des bits 28 à 31 du mot d'état. Les valeurs prises par ce champ sont :

- 0 pour désigner une ALU 32 bits,
- 1 pour désigner une ALU 24 bits,
- 2 pour désigner une ALU 16 bits,
- 3 pour désigner une ALU 8 bits.

Par défaut, 1 est positionné à 0 et est accessible en lecture et écriture.

8) Le bit WAE (bit 7) : *Write Alu bits Enable*

Ce bit est toujours à zéro dans le mot d'état mais doit être positionné à un lors d'une tentative d'écriture sur le mot d'état, pour valider la modification des quatre bits d'ALU.

9) Le bit WWE (bit 6) : *Write WO Enable*

Ce bit est toujours à zéro dans le mot d'état mais doit être positionné à un lors d'une tentative d'écriture sur le mot d'état, pour valider la modification du champ WO.

10) Le bit WBE (bit 5) : *Write Base register Enable*

Ce bit est toujours à zéro dans le mot d'état mais doit être positionné à un lors d'une tentative d'écriture sur le mot d'état, pour valider la modification du Base Register.

11) Le bit WSE (bit 4) : *Write ALU Size Enable*

Ce bit est toujours à zéro dans le mot d'état mais doit être positionné à un lors d'une tentative d'écriture sur le mot d'état, pour valider la modification du champ ALU-SIZE.

12) Le bit WME (bit 3) : *Write Mask Enable*

Ce bit est toujours à zéro dans le mot d'état mais doit être positionné à un lors d'une tentative d'écriture sur le mot d'état, pour valider la modification du champ Mask.

13) Le champ MASK (bit 0 à 2) : *Interrupt Mask*

Ce champ spécifie, parmi les huits niveaux d'interruption existants, le niveau maximal de prise en compte des exceptions. Ce champ est accessible en lecture et en écriture.

On remarquera que les huit bits de poids fort du mot d'état sont placés au même niveau que les huit bits de *tag* des cellules. Cette organisation permet d'utiliser les instructions *case* et *cond* sur le mot d'état comme sur les données typées.

De façon générale, lorsque du fait du pipeline, une affectation explicite d'un des cinq registres spéciaux (r27 à r31) intervient en même temps que sa mise à jour automatique par une instruction spécialisée, la modification explicite du registre est toujours prioritaire.

2.5. Le traitement des exceptions

2.5.1. Les niveaux d'exception

Le processeur KIM20 est doté de huit niveaux d'exceptions hiérarchisées. Le niveau 0 est le plus prioritaire et le niveau 7 le moins. Le tableau suivant donne l'affectation des niveaux d'exceptions :

- niveau 0 : pas d'interruption ou reset (NOINT/RESET)
- niveau 1 : overflow/underflow des fenêtres (WINDOV/WINDUN))
- niveau 2 : overflow/underflow pile (STACKOV/STACKUN)

- niveau 3	manque de cellule (STARVATION)
- niveau 4	exception du mécanisme de codage compact (CDR_ESCAPE)
- niveau 5	trap logiciel (TRAP)
- niveau 6	interruption externe A (INTERRUPTA)
- niveau 7	interruption externe B (INTERRUPTB)

2.5.2. Le reset

Le niveau zéro des exceptions correspond au reset du processeur. Un masque à ce niveau signifie qu'à part le *reset*, aucune exception ne sera prise en compte. Un *reset* du processeur KIM20 a pour effet de remettre dans un état initial l'ensemble du processeur. Cependant, seuls les deux registres spéciaux r30 (LSP) et r31 (PSW) sont initialisés automatiquement à zéro. Il conviendra au logiciel d'initialisation de "nettoyer" les autres registres globaux ainsi que toute la roue des registres locaux, avant leur utilisation.
Le mot d'état (PSW) étant à zéro, aucune exception ne pourra être prise en compte avant une modification explicite du champ *mask*.

Lors du *reset*, le compteur de programme interne (PC) est également initialisé à zéro, afin que la première instruction exécutée soit celle stockée à l'adresse zéro de la page zéro dans la mémoire code.

2.5.3. Le débordement des fenêtres

Cette exception est provoquée lors des instructions *call* et *return*, lorsque les pointeurs WP et WO du mot d'état sont égaux. La comparaison a lieu après l'incrémentation de WP, lors d'une instruction *call*, ou avant la décrémentation de WP lors d'une instruction *return*.

Si cette situation est provoquée par une instruction *call*, il s'agit alors d'un débordement supérieur (*overflow*) et le bit 27 du mot d'état est mis à un.

Si cette situation provient de l'exécution d'une instruction *return*, il s'agit alors d'un débordement inférieur (*underflow*) et le bit 27 du mot d'état est mis à zéro.

Cette exception signifie que les fenêtres internes sont toutes utilisées (*overflow*) ou inutilisées (*underflow*) et que le logiciel de traitement de cette exception devra libérer ou restituer une fenêtre de registres (Chap. 4 § 4.2.).

2.5.4. Le débordement de la pile

Cette exception est générée par les instructions *push* et *pop*, lorsque l'octet de poids faible du pointeur de pile (LSP ou r30) devient égal à FF (255). Dans ces deux cas, la comparaison est effectuée après la mise à jour du registre LSP. Si cette situation est provoquée lors d'une instruction *push*, il s'agit alors d'un débordement supérieur (*overflow*) et le bit 27 du mot d'état est mis à un. Si cette situation est provoquée lors d'une instruction *pop*, il s'agit alors d'un débordement inférieur (*underflow*) et le bit 27 du mot d'état est mis à zéro.

Le programmeur a ainsi à sa disposition un mécanisme simple de contrôle des débordements de la pile de données. En contrepartie, cette dernière devra être constituée de blocs contigus de 256 mots et devra être alignée sur des adresses modulo "8 bits". Cependant, plusieurs piles peuvent être gérées simultanément, par modification dynamique du registre LSP.

2.5.5. Le manque de cellules

Cette exception est générée par une instruction *new*, lorsque le compteur des cellules libres (FLC) devient inférieur ou égal au seuil de déclenchement du *garbage collector* (FLT ou r27).

La comparaison des deux registres est effectuée avant la mise à jour du registre FLC par l'instruction. Cette exception signifie au logiciel système que le niveau de la liste des cellules libres a franchi un seuil critique, qui nécessite l'appel du *garbage collector* pour la récupération des cellules inutilisées.

Les trois registres FLP, FLC et FLT étant accessibles en lecture et en écriture, plusieurs listes de cellules libres peuvent être gérées simultanément.

2.5.6. L'exception du mécanisme de codage compact

La technique du *CDR-coding* permet un codage compact des listes.

Dans le cas où un codage relatif par rapport à l'adresse de la cellule courante est impossible, l'exception "CDR_ESCAPE" est générée.

Celle-ci peut provenir d'une tentative de lecture du champ *next* par l'instruction *next*, ou d'une tentative d'écriture de ce même champ par l'instruction *setn*.

Pour l'instruction *next*, cette exception est provoquée si le champ "cdr" de la cellule lue est différent des valeurs 2 et 3 (CDR-NEXT, CDR-NIL). Le bit 27 du mot d'état est alors positionné à zéro.

Pour l'instruction *setn*, cette exception est provoquée si l'adresse de la cellule pointée est différente de zéro (NIL) ou de l'adresse de la cellule courante plus un. Dans ce cas, le champ "cdr" de la cellule pointée est alors positionné à zéro (CDR-INVISIBLE) et le bit 27 du mot d'état est positionné à un.

Le traitement de l'exception "CDR_ESCAPE" et, en particulier, le maintien d'un chaînage correct des cellules est à la charge du logiciel en utilisant par exemple la technique du "pointeur invisible" (Chap. 4 § 2.1.).

2.5.7. L'instruction "trap"

L'exécution de l'instruction *trap* provoque l'exception logicielle du même nom. Ce mécanisme peut être utilisé pour communiquer avec le système d'exploitation d'une manière homogène sous la forme d'exceptions (événements), ou pour étendre le jeu d'instructions du processeur (interprétation logicielle des opérandes, appel des macro-instructions correspondantes).

2.5.8. Les interruptions externes

Les niveaux d'exception 6 et 7 correspondent à deux interruptions physiques externes au processeur KIM20. Ces deux exceptions

peuvent être utilisées pour connecter une horloge temps réel, un mécanisme du type *bus error* ou des circuits périphériques.

2.5.9. Conditions de prise en compte des interruptions

La prise en compte d'une interruption par le processeur peut être momentanément invalidée, d'un point de vue logiciel, au travers du champ *mask* du mot d'état et, d'un point de vue matériel, lors de "sections critiques", où les interruptions risqueraient d'entraîner de sérieuses perturbations. A cet effet, le masque logiciel du mot d'état permet de filtrer les exceptions générées. Une exception ne peut être prise en compte que si son niveau n'est pas supérieur à celui du masque. Ainsi, un masque au niveau zéro inhibe toutes les interruptions, tandis qu'un masque au niveau sept les autorise toutes.

D'un point de vue matériel, il existe trois cas de figures particuliers où la prise en compte des interruptions est totalement invalidée par le processeur.

Le premier cas a lieu lors du second cycle des instructions *cond*, *branch* et *rete*. Ces instructions, en effet, interagissent avec l'exécution de l'instruction immédiatement suivante (*nop* conditionnel ou systématique) et l'interruption de cette dernière entraînerait, au retour de la procédure d'exception, un fonctionnement incorrect dû à une perte d'information.

Le second cas d'inhibition des exceptions a lieu lors de toute écriture sur le mot d'état (r31/PSW) pendant le second et le troisième cycle de l'instruction d'affectation. Cette restriction est essentiellement due au *base-register* pour prévenir la prise en compte d'une instruction entre une modification du registre de base et un appel ou retour fonctionnel à une page différente de la page courante. En effet, au retour de l'exception, la page courante restituée ne serait pas la page demandée.

Le dernier cas de section critique du processeur correspond à un laps de temps plus long qui sera abordé plus en détail. Il s'agit, lors d'une procédure d'exception, du temps nécessaire au système logiciel pour identifier le niveau de l'interruption à traiter et mémoriser l'adresse de retour au déroulement normal du programme.

Le traitement générique des exceptions est tel que les seules interruptions concernées par ces trois cas d'invalidation absolue sont les interruptions externes (niveau 6 et 7), car celles-ci sont asynchrones par rapport au déroulement des programmes.

Les exceptions ne sont pas mémorisées, c'est-à-dire que si elles surviennent alors qu'elles sont masquées, de façon logicielle ou matérielle, elles sont définitivement perdues. De ce fait, en ce qui concerne les interruptions externes, elles doivent être maintenues suffisamment longtemps pour être acceptées.

2.5.10. Séquencement générique des exceptions

Les exceptions internes au processeur sont toujours générées pendant le second cycle des instructions qui en sont la cause. Par ailleurs, elles sont détectées et prises en compte pendant l'exécution du premier cycle de chaque instruction (*fetch*).

Pour assurer une parfaite homogénéité des traitements, la solution choisie a été de "vider" parfaitement le pipeline entre la génération d'une exception interne et le début de la séquence de traitement. Pour ce faire, la génération d'une exception interne entraîne systématiquement une invalidation dynamique de l'instruction suivante ; la prise en compte d'une exception lors du premier cycle d'une instruction entraîne l'invalidation de cette même instruction, suivie de l'affectation à zéro du compteur ordinal (PC) et de l'inhibition de toute nouvelle interruption. Cette entrée du processeur en section critique est effectuée en positionnant un "masque absolu" interne.

La génération d'une exception interne aboutit donc toujours à l'abandon momentané des deux instructions immédiatement suivantes (*no operation*). Ce n'est qu'après ces deux instructions "noppées" que le traitement d'exception débute effectivement à l'adresse zéro de la page zéro de la mémoire code. Cette opération n'entraîne pas un changement de fenêtre. Le logiciel de traitement de l'exception peut alors utiliser tous les registres à sa disposition (locaux ou globaux) en les ayant sauvegardés au préalable.

Au cours du traitement d'exception, la première instruction *call* exécutée depuis l'entrée du processeur en section critique sauvegarde, non pas l'adresse de retour de l'appel fonctionnel (PC + 1), mais l'adresse de retour de la procédure d'exception. Cette

adresse est, dans tous les cas, l'adresse de la première instruction "noppée" après la génération de l'exception (voir Chap. 4 § 4.1. pour le code source de cette séquence particulière).

A la fin du traitement de l'exception, la procédure rend la main à la tâche interrompue par une instruction *return* ou *rete*, instruction qui repositionne également la fenêtre de travail initiale.

Par conséquent, une instruction *call* est nécessaire dans toute procédure de traitement d'une exception. C'est également cette instruction qui sort le processeur de section critique en revalidant le fonctionnement du masque logique présent dans le mot d'état.

Cette description du mode de prise en compte des exceptions termine la description du modèle de programmation visible par l'utilisateur. Passons maintenant à l'étude de l'architecture qu'il sous-entend.

3. Description de l'architecture matérielle

3.1. Le chemin des données

Le chemin des données du processeur KIM20 est basé sur le modèle RISC "classique" tel qu'il a été introduit au second chapitre. Il repose donc principalement sur un banc de registres connecté à une Unité Arithmétique et Logique par l'intermédiaire de trois bus. Les deux premiers (src1 et src2) fournissent les opérandes sources à l'opérateur sélectionné, le troisième (*dest*) permet l'écriture du résultat dans le registre destination. Sur ce schéma élémentaire, viennent s'ajouter les deux bus supplémentaires qui résolvent le problème d'interdépendances des opérandes dues au pipeline (voir Chap. 2 § 3.1.). Le processeur KIM20 est basé sur une structure Harvard (voir Chap. 2 § 3.5.) qui sépare les accès aux instructions des accès aux données. Ce choix est guidé par l'occurrence relativement importante des accès à la mémoire de données, pour le traitement symbolique. Par conséquent, nous retrouvons sur le chemin des données, les deux interfaces externes relatifs aux accès à la mémoire programme et à la mémoire de données.

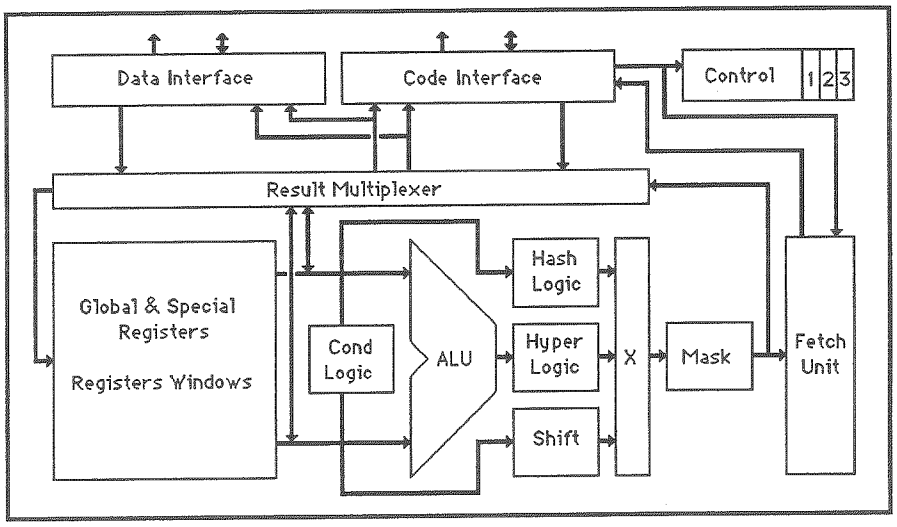


Fig. 72 - Le chemin des données

En parallèle avec l'Unité Arithmétique et Logique se trouve l'unité de décalage qui permet l'exécution des instructions de décalage et de rotation, ainsi que les opérateurs *cond* et *hash* pour les instructions du même nom. En série avec l'Unité Arithmétique et Logique, nous trouvons l'opérateur *hyper*, suivit d'une unité de masquage.

3.2. Intégration d'une instruction particulière

Prenons comme exemple l'instruction *hyper* qui illustre parfaitement l'intégration d'une instruction spécifique dans le chemin des données.

L'instruction *hyper* a pour objectif d'accélérer les algorithmes de routage dans une architecture multiprocesseur KIM20 organisée sous la forme d'une topologie hypercube [111].

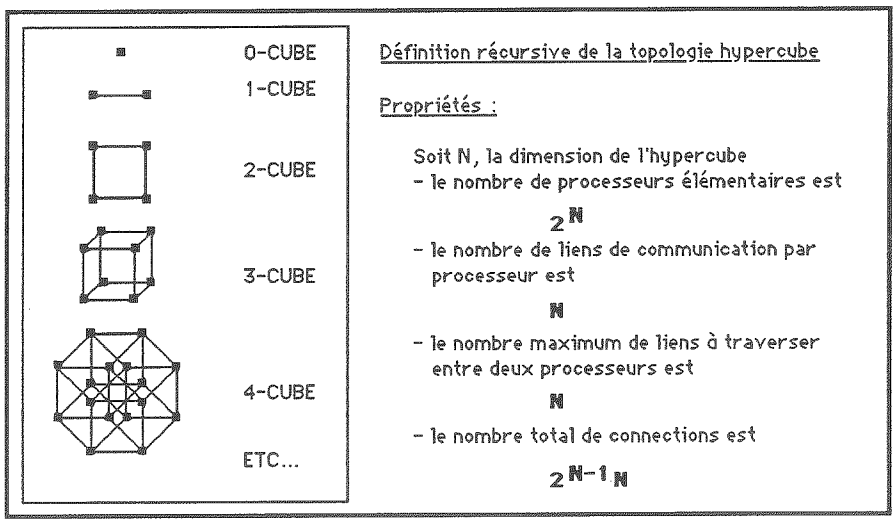


Fig. 73 - Définition de la topologie hypercube

L'instruction *hyper* du groupe "Arithmétique et Logique" correspond au routeur de messages dans une telle structure et permet ainsi l'accélération des logiciels d'envois de messages. *Hyper* calcule l'adresse relative du lien de communication qui réduit la distance (de Hamming) entre le processeur destination désigné par les 16 bits de poids faible du registre source S1 et le processeur courant désigné par les 16 bits de poids faible de l'opérande [#]S2. Le résultat est stocké dans le registre destination Rd, auparavant initialisé à zéro. Pour cela, les processeurs de la structure hypercube sont numérotés en binaire d'une façon naturelle (voir figure 74), où chaque processeur ne diffère de ses voisins que par un seul bit. Une simple opération "OU exclusif" (*xor*) entre les identifieurs du processeur courant et du processeur destination détermine alors les bits différents. Dans le cas où ils sont tous égaux, les bits de code condition du mot d'état indiquent que le message est arrivé à sa destination (distance de Hamming = 0). Dans le cas contraire, il suffit de prendre le premier bit de poids fort non nul (numéroté de 0 à 15) et son rang donne alors l'adresse relative du lien (un parmi 16) où le message devra être envoyé. On remarquera ici qu'un tirage aléatoire serait théoriquement plus satisfaisant, mais plus difficile à réaliser simplement dans un circuit intégré.

Nous noterons que ce choix permet également d'utiliser *hyper* pour optimiser les boucles présentes dans les algorithmes de

multiplication et division entières (voir Chap. 4 § 4.3.) ou de normalisation pour les fonctions de calcul en format flottant.

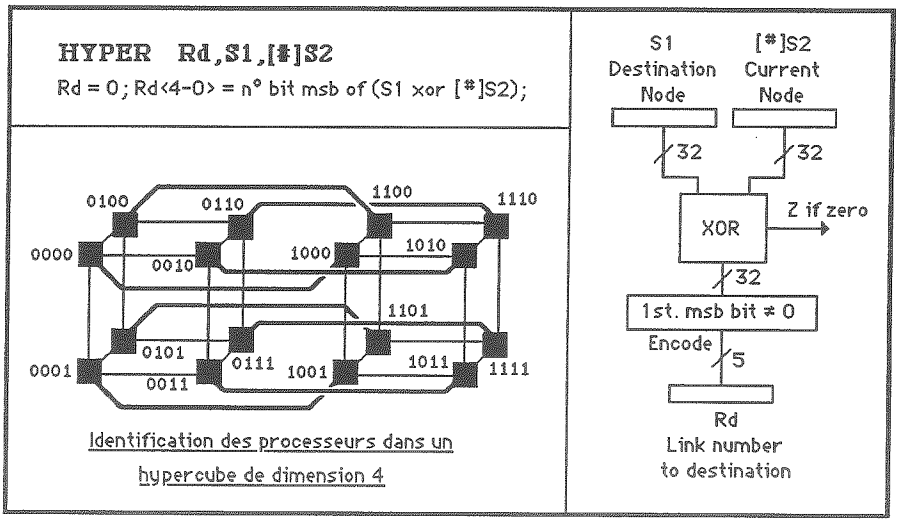


Fig. 74 - L'instruction "hyper"

La réalisation de l'opérateur *hyper* nécessite donc une opération "OU exclusif" réalisée sur les deux opérandes sources, suivie de la détection du premier bit non nul, puis de son codage. L'opération "OU exclusif" peut être efficacement réalisée par l'Unité Arithmétique et Logique puisqu'elle est déjà présente comme instruction de base. Il suffit donc de placer en série, avec l'Unité Arithmétique et Logique, les opérateurs logiques nécessaires en vérifiant que le temps de transfert total soit compatible avec le cycle du pipeline associé (figure 74).

3.3. L'unité de lecture et décodage des instructions

L'unité de lecture et de décodage des instructions est présentée par la figure 75. Sa structure diffère très peu de celle du processeur RISC-II. Nous y retrouvons les registres qui permettent la mémorisation des opérandes sources et destinations lors des différents cycles d'exécution du pipeline, ainsi que la logique de décodage du code opération chargé de générer les bits de contrôle des opérateurs câblés.

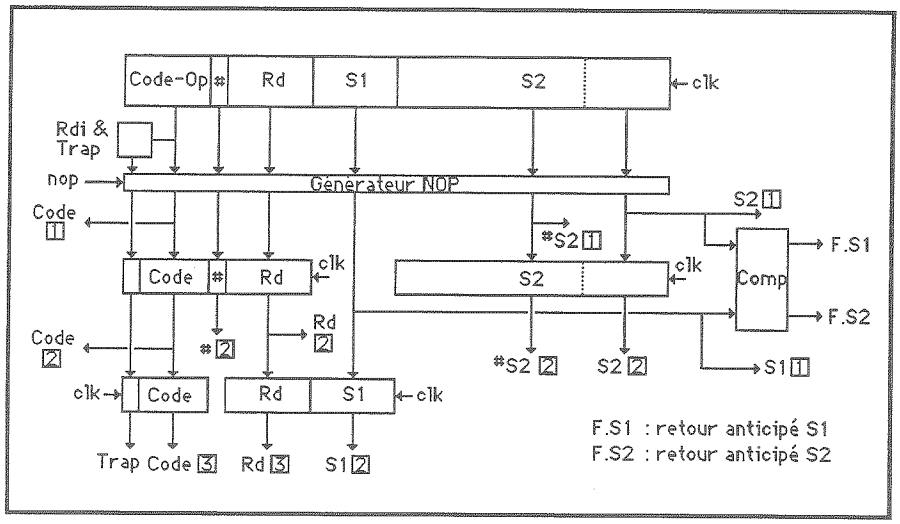


Fig. 75 - L'unité de séquencement et de décodage

3.4. Le pipeline d'exécution

3.4.1. Un pipeline synchrone à trois étages

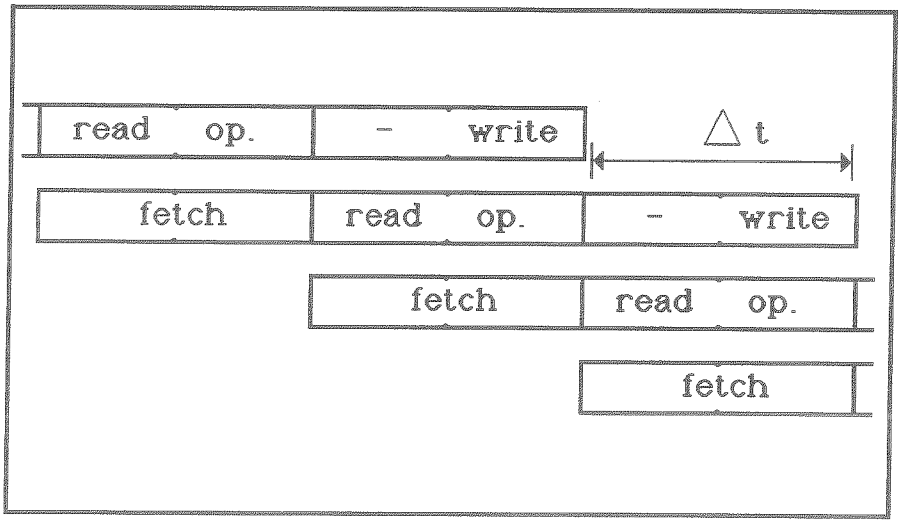


Fig. 76 - Le pipeline d'éxécution du processeur KIM20

Le pipeline du processeur KIM20 hérite directement des travaux menés sur RISC-II. Comme ce dernier, il est basé sur un modèle synchrone à trois étages.

Le premier *fetch* l'instruction à exécuter, le second effectue la lecture des opérandes puis exécute l'opération sur l'Unité Arithmétique et Logique, le dernier étage range le résultat dans un registre destination. A chaque cycle machine, ces trois étages sont exécutés simultanément, permettant ainsi la superposition de trois instructions exécutées séquentiellement.Voyons maintenant plus précisément le déroulement des différentes phases pour l'exécution des différents types d'instructions.

3.4.2. Séquencement d'une instruction Arithmétique et Logique

Toute instruction, quelle qu'elle soit, se déroule en trois cycles qui correspondent aux différents étages du pipeline d'exécution.

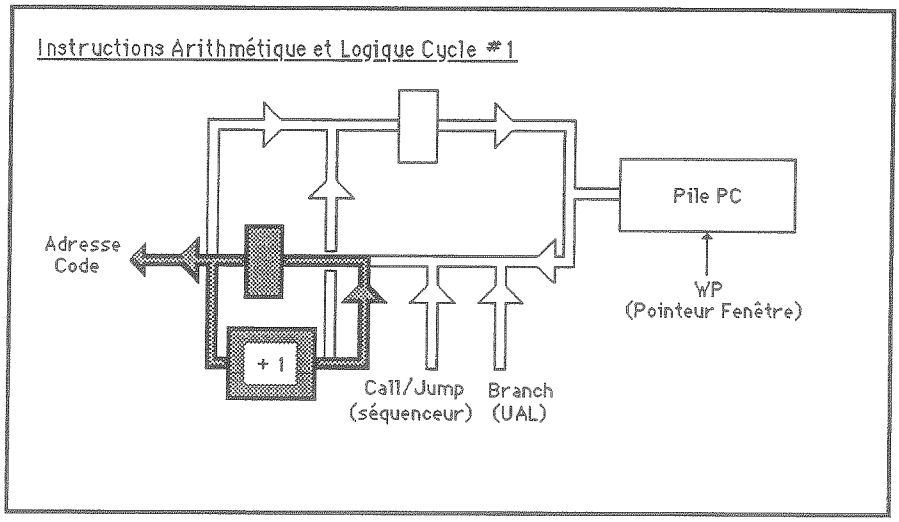


Fig. 77 - 1er cycle pour une instruction Arithmétique et Logique

Le premier cycle effectue la recherche de l'instruction dans la mémoire en présentant l'adresse sur le bus externe d'accès à la zone de mémoire "code". Dans le même temps, un incrémenteur génère l'adresse de la prochaine instruction à exécuter. Enfin, le décodage des adresses des opérandes est effectué pour préparer le cycle suivant.

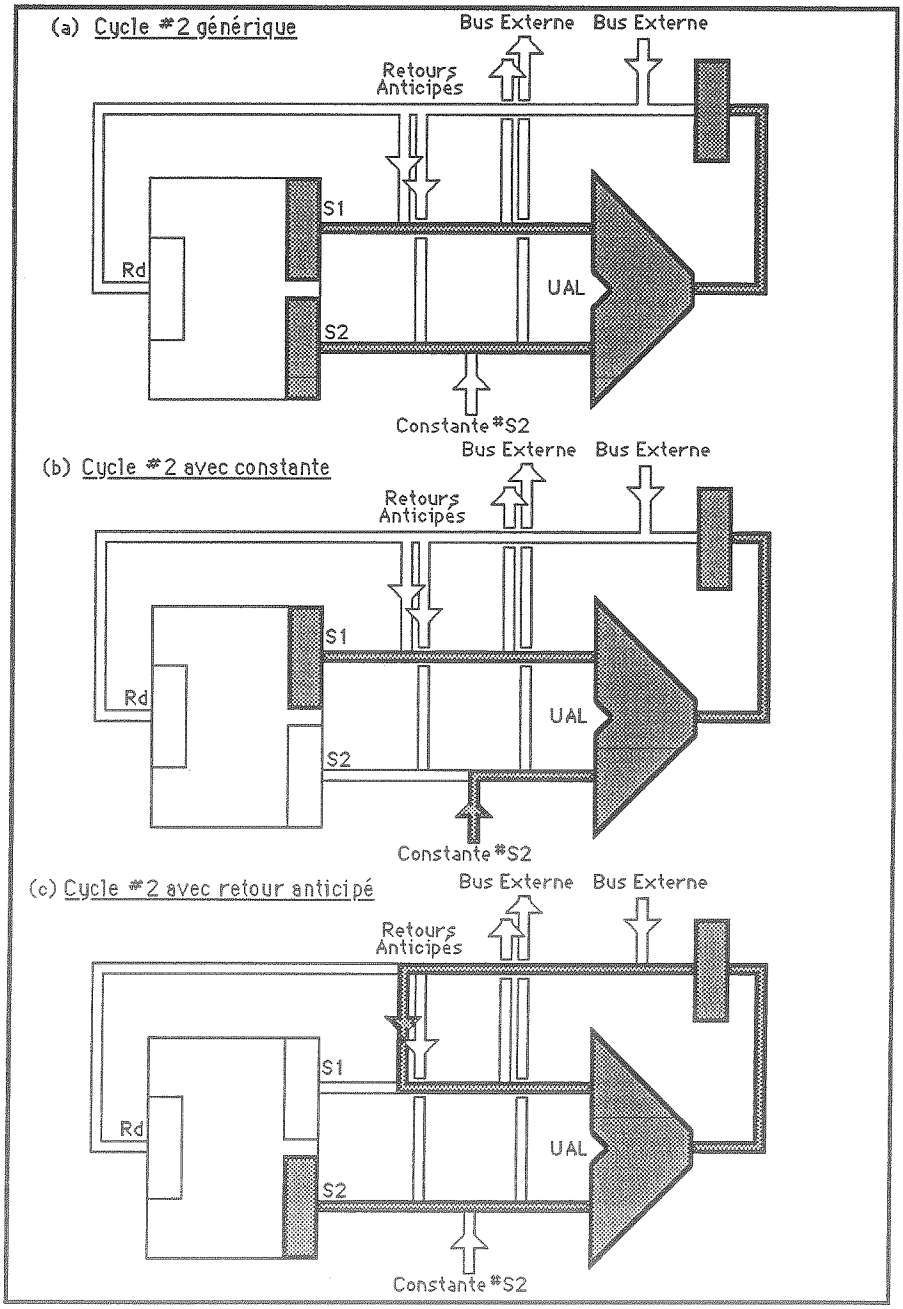


Fig. 78 - 2e cycle d'une instruction Arithmétique et Logique

Au cours du second cycle, plusieurs cas de figures peuvent se présenter :

1) la lecture des registres opérandes, suivie du calcul du résultat sur l'Unité Arithmétique et Logique (figure 78.a),

2) la lecture d'un registre opérande avec une constante, suivie du calcul du résultat sur l'Unité Arithmétique et Logique (figure 78.b),

3) la lecture des registres opérandes dont une des valeurs doit être obtenue par le "retour anticipé", suivie du calcul du résultat (figure 78.c).

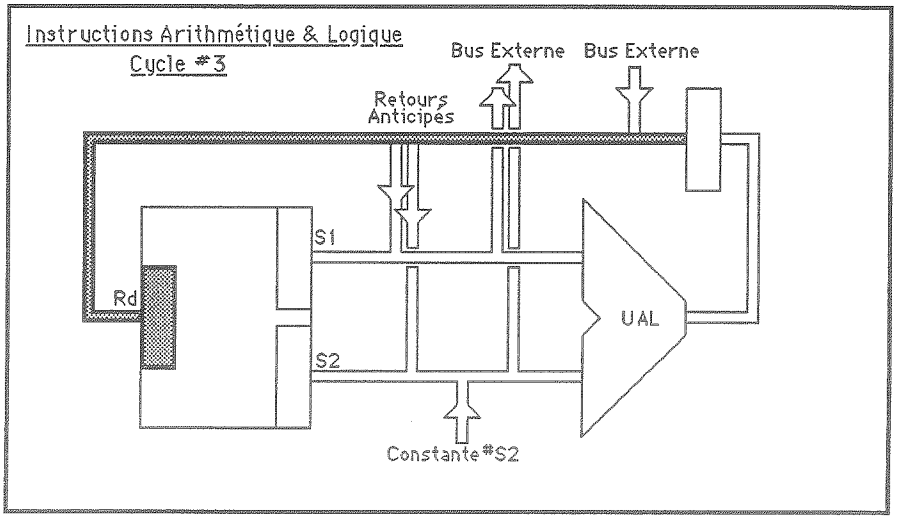


Fig. 79 - 3e cycle d'une instruction Arithmétique et Logique

Le troisième et dernier cycle calcule l'adresse du registre destination, puis effectue l'écriture du résultat dans le registre sélectionné (figure 79).

3.4.3. Séquencement des instructions d'accès aux bus

Le premier cycle des instructions d'accès aux bus est équivalent au premier cycle d'une instruction Arithmétique et Logique (paragraphe précédent).

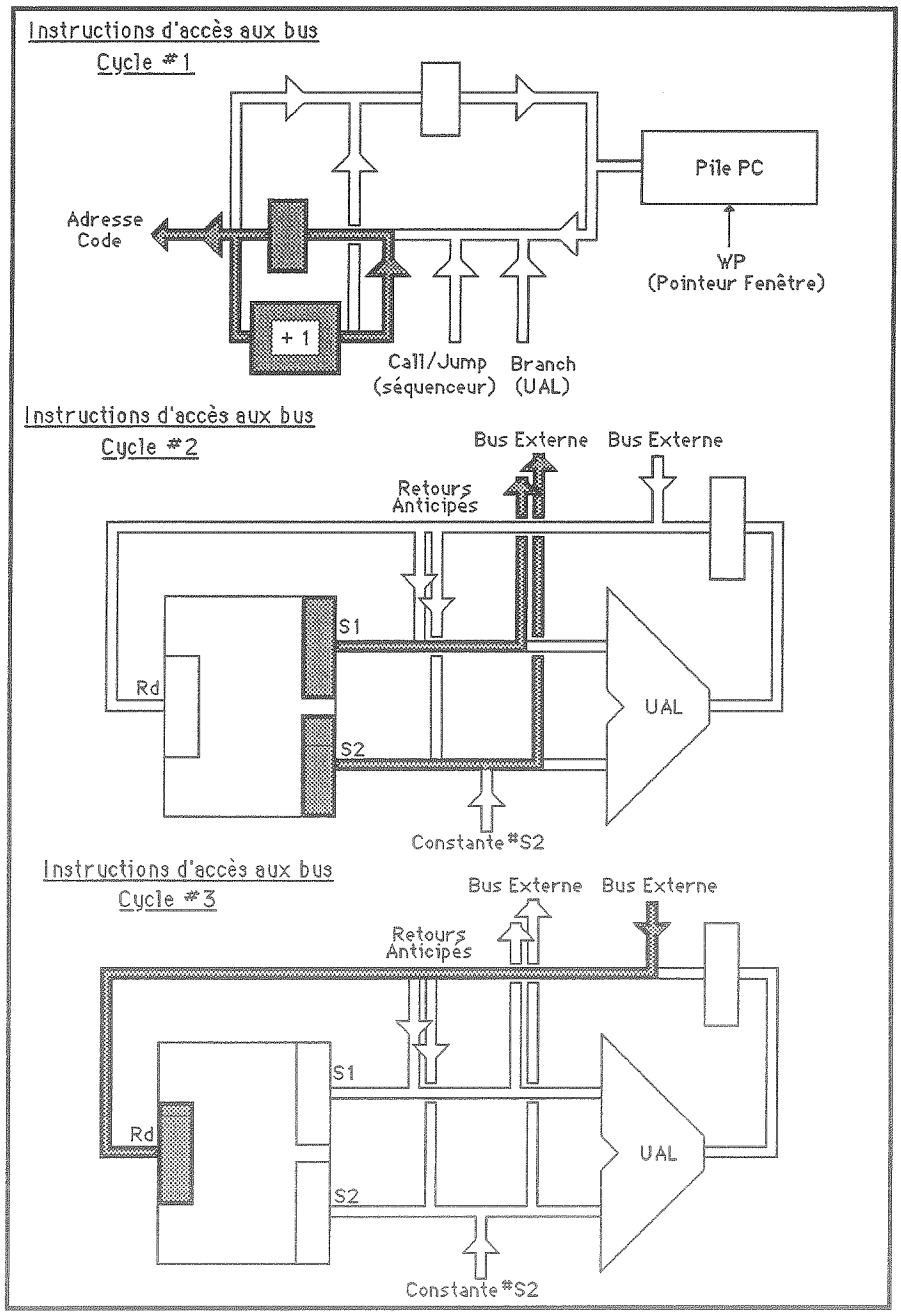


Fig. 80 - L'éxécution des instructions d'accès au bus

Le second cycle, quant à lui, effectue la lecture des opérandes et l'accès au bus externe proprement dit.

Pour les instructions "d'écriture" en mémoire, le troisième cycle ne fait rien puisqu'il n'y a pas d'écriture de résultat dans le banc de registres interne.

Pour les instructions de "lecture", le dernier cycle calcule l'adresse du registre destination, puis stocke le résultat dans le registre.

3.4.4. Séquencement des instructions de branchement

La plupart des instructions de branchement se déroulent pendant le premier cycle du pipeline d'exécution. Celui-ci effectue la recherche de l'instruction courante, puis calcule l'adresse effective de branchement.

La figure 81 donne les exemples des différents chemins possibles au travers des instructions *jump*, *branch*, *call* et *return*.

Les second et troisième cycles ne sont pas utilisés sauf par l'instruction *call* qui emploie le second cycle pour stocker l'adresse de retour dans la pile des compteurs de programme associé à la roue de fenêtres, et le troisième cycle pour stocker cette même adresse dans un registre destination directement exploitable par le logiciel.

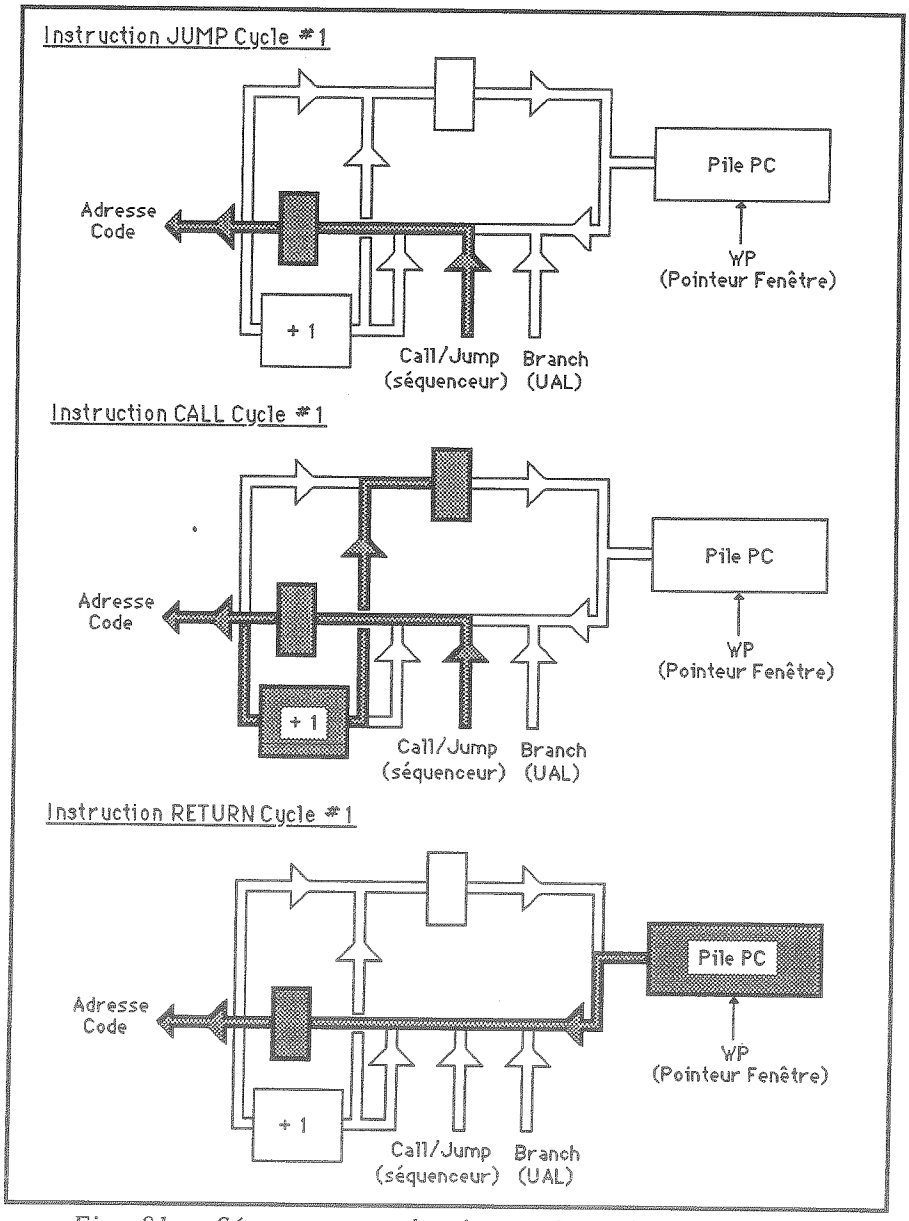


Fig. 81 - Séquencement des instructions de branchement

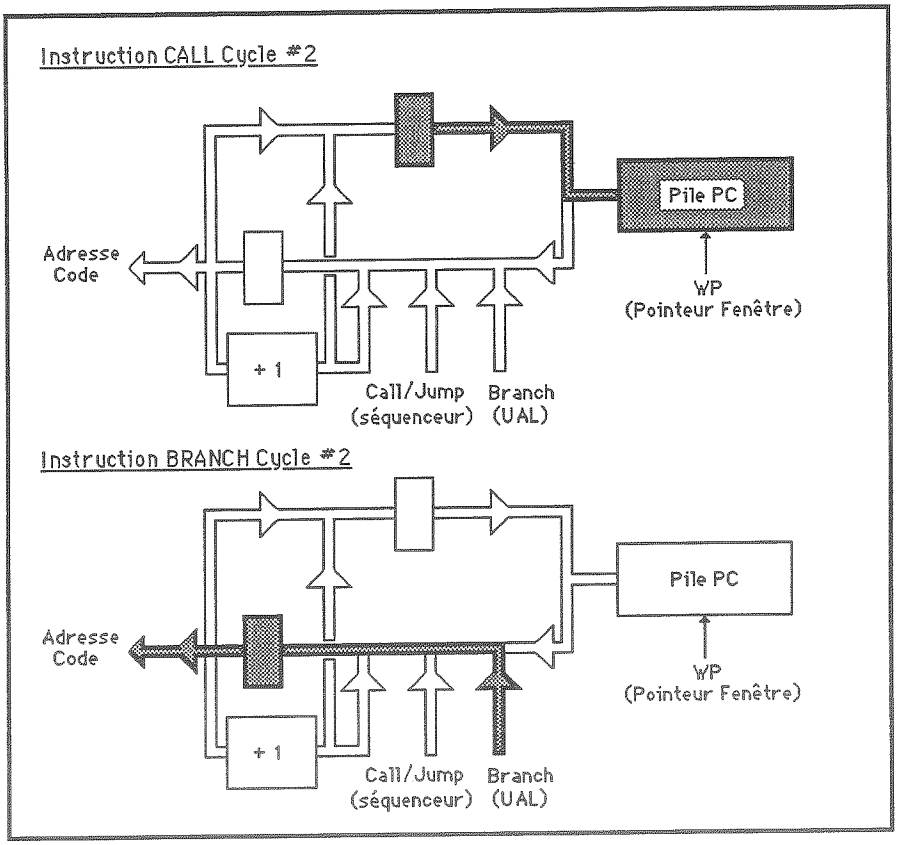


Fig. 82 - Séquencement du 2e cycle des instructions de branchement

3.5. Quelques détails sur la conception

L'architecture du processeur KIM20 a été expérimentée initialement sur une carte réalisée en technologie discrète. Celle-ci regroupe plus de quatre cents composants CMOS sur une carte triple-europe au format standard VME.

Baptisée KIM10, cette version du processeur a permis la validation de l'architecture, avant d'entamer l'étape décisive de l'intégration VLSI.

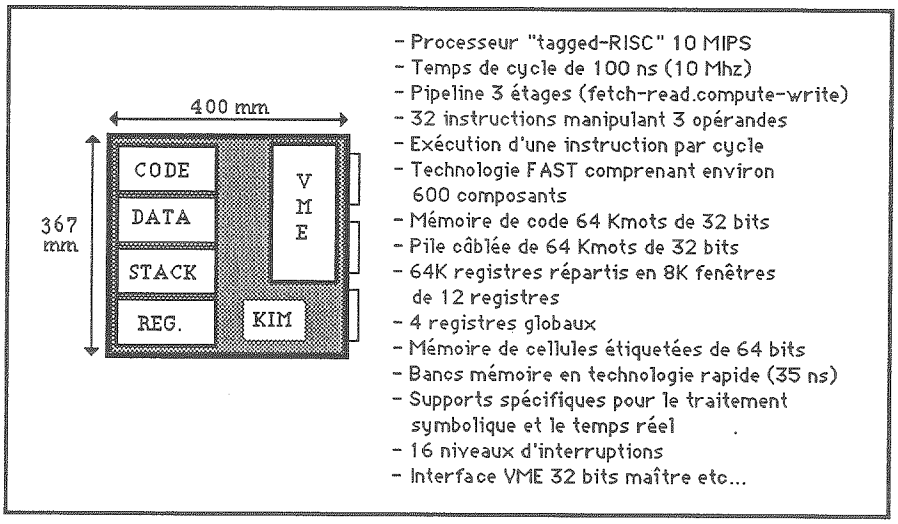


Fig. 83 - La carte processeur KIM10

La carte a été conçue sur un système Mentor Graphics, avec des phases de simulations de plusieurs dizaines d'heures (!) afin de vérifier le séquencement des diverses instructions. Une fois cette étape préliminaire achevée, le dessin de l'architecture VLSI a été réalisé sur une station VALID, puis simulé entièrement grâce au logiciel HILO-3. La réalisation en technologie précaractérisée 1.2 microns fut ensuite effectuée dans un premier temps avec les outils de développement de la Société NEC.

Afin d'améliorer les performances du processeur d'une part, et pour obtenir une source purement européenne d'autre part, une seconde version fut réalisée peu après chez la Société ES2 (European Silicon Structure).

En effet, l'optimisation du banc de registres, grâce à un compilateur de silicium de l'outil SOLO2000, permit une réduction sensible du temps d'accès aux registres qui représente un des paramètres fondamentaux du chemin critique, fixant les limites du temps de cycle.

Cette optimisation autorise également une réduction d'environ 1 millimètre de la taille de la puce sur chaque côté, malgré une technologie moins fine (1.5 micron).

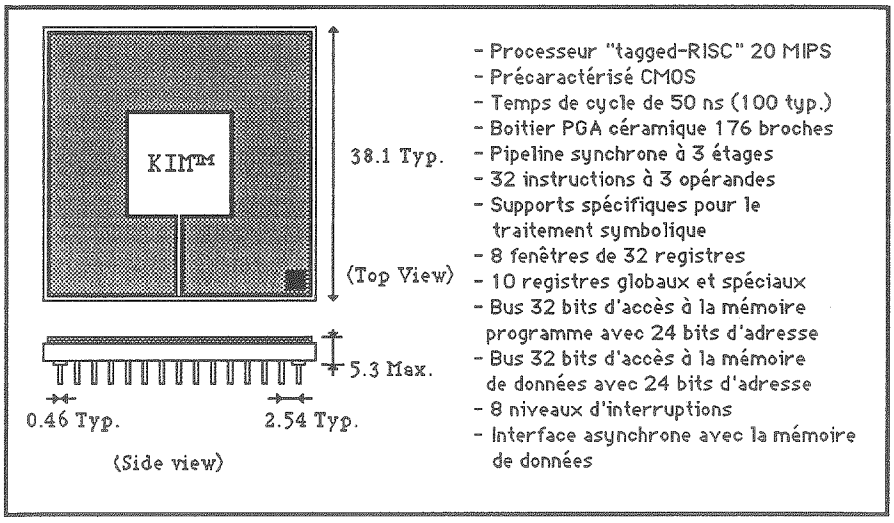


Fig. 84 - Le processeur VLSI KIM20

Le résultat final se présente donc sous la forme d'une puce d'environ 1 centimètre de côté, intégrant près de 17000 portes équivalentes. La version ES2 permet d'exécuter 10 MIPS efficace (16 MIPS crête) à une fréquence interne de 16 Mhz.

4. Exemples de programmation

4.1. La gestion des exceptions

Ce paragraphe comprend plusieurs exemples qui illustrent la simplicité de programmation du processeur KIM20. Cette simplicité provient essentiellement du nombre réduit d'instructions (32) et de l'unicité du format qui réduit la complexité.

Ainsi, au bout de quelques heures de pratique, un programmeur peut aisément maîtriser l'assembleur KIM20, alors qu'il faut plusieurs jours (voir plusieurs semaines) pour connaître parfaitement les subtilités d'un Motorola MC68020.

Le premier exemple correspond au programme d'une petite mémoire ROM, généralement située près du processeur, dont le rôle est de télécharger le code exécutable en provenance d'une mémoire re-prom (*reprogrammable R.O.M.*) présente sur le bus de données vers la mémoire programme, pour son exécution.

```
; Programme résident en micro-prom de 32 mots : chargement du contenu de la
; reprom en mémoire programme (séquence reset, pré-boot, boot...)

EVENTS_GATE:    org     ROM_BEGIN                   ; adresse exceptions

                cond    r31,*NRESET?                ; est-ce un reset?
                jump    *NO_RESET_EVENT             ; sinon exception
RESET_BEGIN:
                xor     r22,r22,r22                 ; reset registre r22
                add     r23,r22,*CODE_BEGIN         ; charge adresse code
                add     r22,r22,*REPROM_ADD_MSB     ; charge adresse msb
                rotate  r22,r22,*16                 ; décale vers msb
                add     r22,r22,*REPROM_ADD_LSB     ; ajoutte les lsb
RESET_LOOP:
                first   r24,r22,*0                  ; charge 1er octet
                insert  r25,r24,*0                  ; insert en octet 0
                add     r22,r22,*1                  ; adresse reprom suivante
                first   r24,r22,*0                  ; charge 2ème octet
                insert  r24,r24,*1                  ; insert en octet 1
                or      r25,r25,r24                 ; écrit dans résultat
                add     r22,r22,*1                  ; adresse reprom suivante
                first   r24,r22,*0                  ; charge 3ème octet
                insert  r24,r24,*2                  ; insert en octet 2
                or      r25,r25,r24                 ; écrit dans résultat
                add     r22,r22,*1                  ; adresse reprom suivante
                first   r24,r22,*0                  ; charge 4ème octet
                insert  r24,r24,*3                  ; insert en octet 3
                or      r25,r25,r24                 ; écrit dans résultat
                add     r22,r22,*1                  ; adresse reprom suivante

                store   r23,r25                     ; écrit le résulat en code
                add     r23,r23,*1                  ; adresse code suivante
                sub     r26,r23,*REPROM_SIZE/4+CODE_BEGIN

                cond    r31,*NZERO?                 ; est-ce la fin?
                jump    *RESET_LOOP                 ; sinon on continue

                jump    *RESET_EVENT                ; si oui la suite...
```

Fig. 85 - Programme de téléchargement au "reset"

La boucle *Reset-Loop* extrait, octet par octet, le programme exécutable de la mémoire Reprom. Elle stocke ensuite le résultat dans un registre en ayant soin de reformer chaque instruction, puis écrit cette instruction dans la mémoire programme.

La seconde partie du programme, située en mémoire vive, correspond à la table de branchement des exceptions lorsque l'événement traité n'est pas un *reset* matériel.

```
;  Programme générique de traitement des exceptions à l'adresse 0 de la mémoire
;  programme : temps de réponse à une interruption ≠ 1.1 µsec. à 10 Mhz
                org     CODE_BEGIN              ; début mémoire code

                jump    *RESET_EVENT            ; reset
                push    psw                     ; sauvegarde mot d'état
                push    g0                      ; sauvegarde global 0 (r22)
                case    g0,psw,*EXCEP_LEVEL     ; type de l'exception?
                sub     g0,g0,*1                ; masque d'exception -1
                or      psw,g0,*WE_MASK         ; valide le nouveau masque
                branch  g0,*EVENT_TAB           ; branchement traitement
EVENTS_TAB:                                     ; table des évenements

                jump    *WINDOW_UNDERFLOW       ; sous-passement fenêtrage
                call    g0,*STACK_UNDERFLOW     ; sous-passement pile
                call    g0,*NO_MORE_CELLS       ; appel du GC
                call    g0,*NEXT_ESCAPE         ; CDR-coding sur next
                call    g0,*TRAP                ; trap logiciel
                call    g0,*INTERRUPT_A         ; interruption matérielle A
                call    g0,*INTERRUPT_B         ; interruption matérielle B
                call    g0,*SPURIOUS            ;
                jump    *WINDOW_OVERFLOW        ; débordement fenêtrage
                call    g0,*STACK_OVERFLOW      ; débordement pile
                call    g0,*SPURIOUS            ;
                call    g0,*SETN_ESCAPE         ; CDR-coding sur setn
                call    g0,*SPURIOUS            ;
                call    g0,*SPURIOUS            ;
```

Fig. 86 - Table de gestion des exceptions et interruptions

Dans ce cas, une séquence *case* ... *branch* permet de calculer l'adresse de la routine à exécuter en fonction de l'exception rencontrée. Celle-ci correspondra à un appel de procédure stocké dans une table d'exception baptisée ici "EVENTS-TAB". L'instruction *call* de cette table étant le premier appel de procédure exécuté depuis l'exception, un retour de procédure *return* dans la routine de traitement permettra d'effectuer un retour à la procédure interrompue.

4.2. Les débordements de fenêtre

Deux des exceptions présentes dans la table des exceptions du paragraphe précédent, correspondent aux traitements des débordements de la roue de fenêtres lors des appels fonctionnels.

```
;; procédure de sauvegarde d'une fenêtre si débordement (Lydie Bol 1989)

WINDOW_OV:  xor     g0,g0,g0        ; registre nul
            push    g1              ; sauve g1
            ; lecture du pointeur de la pile de sauvegarde
            add     g1,g0,#PROCESS_ID
            first   g1,g1,#0        ; indice du process courant
            add     g1,g1,#PROCESS_TAB+PT_LOCAL_STACK
            add     g0,lsp,#0       ;sauve lsp utilisateur
            first   lsp,g1,#0       ; pile de sauvegarde
            ; sauvegarde des registres locaux
            push    l0              ; registres locaux
            push    l1
            push    l2
            push    l3
            push    l4
            push    l5
            push    l6
            push    l7
            push    l8
            push    l9
            push    y0              ; registres paramètres
            push    y1
            push    y2
            push    y3
            push    y4
            push    y5
            send    g1,lsp          ; mise à jour table des process
            ; mise à jour du pointeur WO dans le mot d'état
            add     lsp,g0,#0       ; récupère lsp utilisateur
            call    g0,#$+1         ; mémorise instruction interrompue
            rotate  g1,psw,#11      ; wp en poids faibles
            and     g1,g1,#7        ; à cause des bits write enable
            rotate  g1,g1,#18       ; wo <- wp
            or      psw,g1,#WE_WO   ; affectation wp
            ; c'est fini...
            pop     g1              ; restitution des globaux
            pop     g0
            pop     psw
            or      psw,psw,#WE_ALU+WE_ASZ+WE_MASK
            return
```

Fig. 87 - Routine de traitement de l'exception de débordement des fenêtres

Le premier cas possible se produit lorsqu'une instruction *call* est exécutée alors que la roue de fenêtres est "pleine" (*overflow*). Le second cas se produit lorsqu'une instruction *return* est exécutée alors que la roue de fenêtres est "vide" (*underflow*). Le code source de la figure 87 donne un exemple particulier d'une procédure résolvant le problème du dépassement de capacité. La première procédure stocke un contexte dans une pile située en mémoire, libérant ainsi une fenêtre pour l'appel courant. Un bon exercice pour maîtriser le fonctionnement de la roue de fenêtres consiste à écrire la routine "inverse" qui permet la restitution d'une fenêtre à partir d'une pile située dans la mémoire des données.

4.3. Multiplications et divisions entières

```
; Programme de multiplication entière 32 bits non signée : x1 et x2 opérandes
; (x et y registres pour le passage et retour, 1 registres locaux, rz = g0 = 0)

MULTIPLY:   xor     x3,x3,x3                ; résultat

            sub     12,x1,x2                ; recherche du plus petit
            cond    psw,#LOWER_OR_SAME?     ; x1 <= x2 ?
            jump    #$+4                    ; on continue
            add     12,x1,rz                ; sinon permutation
            add     x1,x2,rz                ;
            add     x2,12,rz                ;

            hyper   13,x1,rz                ; msb de x1
            add     12,rz,#31               ; nombre de décalages
            sub     12,12,13                ;
            rotate  x1,x1,12                ; x1 en poids forts

MUL_LOOP:   rotate  x3,x3,#1                ; res = res * 2
            rotate  x1,x1,#1                ; test du bit de poids fort
            cond    psw,#CARRY?             ; si carry...
            add     x3,x3,x2                ; res = += multiplie
            sub     13,13,#1                ; test si terminé
            cond    psw,#POSITIVE_OR_ZERO?  ;
            jump    #MUL_LOOP               ; sinon on contine

            return                          ; fin
```

Fig. 88 - Routine de multiplication entière non-signée

L'exemple précédent montre comment, à partir des instructions Arithmétiques et Logiques, un algorithme itératif de multiplication entière peut être programmé. Ce dernier est optimisé grâce à l'instruction *hyper* qui calcule l'indice de la boucle d'itération. Ce type d'algorithme représente une alternative aux méthodes nécessitant des "pas" de multiplication (*multiply and divide steps*). Ces instructions n'ont pas été implantées dans KIM20 du fait de la présence quasi-systématique d'un coprocesseur de calcul. Néanmoins, un exercice intéressant pour le lecteur consiste à coder les algorithmes de multiplication et division signés.

4.4. Quelques fonctions récursives

```
;; fonction de fibonnaci en Common-Lisp et en assembleur
;;
;;          (defun fib (n)
;;             (cond ((= n 0) 1)
;;                   ((= n 1) 1)
;;                   (t (+ (fib (- n 1))
;;                         (fib (- n 2))))))

FIB :    equ      *              ; fibonnaci
         sub      r4,r0,#0       ; test si n = 0
         cond     psw,#ZERO      ; sinon saut FIB1
         jump     FIB1           ;
         add      r0,r15,#1      ; alors résultat = 1
         return                  ; retour

         sub      r4,r0,#1       ; test si n = 1
         cond     psw,#ZERO      ; sinon saut FIB2
         jump     FIB2           ;
         add      r0,r15,#1      ; alors résultat = 1
         return                  ; retour

FIB1 :   sub      r8,r0,#1       ; n = n - 1
         call     lpc,#FIB       ; (fib n)
         add      r5,r8,#0       ; résultat

FIB2:    sub      r8,r0,#2       ; n = n - 2
         call     lpc,#FIB       ; (fib n)
         add      r0,r8,r5       ; résultat

         return
```

Fig. 89 - Source et code assembleur de la fonction de Fibonnaci

Le programme de ce paragraphe donne un exemple de fonction récursive classiquement écrite en Lisp pour l'évaluation des performances. Cette dernière est la célèbre fonction de Fibonnaci. Le programme Lisp correspondant est donné, puis sa traduction en assembleur KIM20, tel qu'il pourrait être généré par un compilateur Common-Lisp optimisé. La figure 90 donne le source en Lisp des fonctions Takeuchi et Ackerman, qui pourront servir d'exercices pratiques de programmation. Les lecteurs curieux tenteront ensuite d'analyser finement le comportement très récursif de ces fonctions devenues des classiques du genre, et l'apport du mécanisme de fenêtrage.

```
;; fonction de takeuchi

(defun tak (x y z)
    (if (not (< y x))
        z
        (tak (1- x) y z)
        (tak (1- y) z x)
        (tak (1- z) x y))))

;; fonction d'ackerman

(defun ack (n m)
    (cond ((zerop n) (1+ m))
          ((zerop m) (ack (- n 1) 1))
          (t (ack (- n 1) (ack (n (- m 1)))))))
```

Fig. 90 - Sources en Lisp des fonctions d'Ackerman et Takeuchi

4.5. Fonctions de création et de parcours de listes

Les deux fonctions de ce paragraphe permettent de mettre en valeur les capacités du processeur KIM20 pour la création de structures de données complexes, puis leur parcours. La procédure 2nc est une fonction Lisp qui construit un arbre binaire de profondeur n. La fonction cxr, quant à elle, permet le parcours de l'arbre passé en argument. Les sources en Lisp sont donnés pour chacune des fonctions. Le code assembleur correspondant est donné pour la fonction 2nc. Le codage de la fonction cxr n'est pas donné, pour laisser libre cours à l'imagination du lecteur désireux de s'initier aux techniques de parcours d'arbres avec KIM20.

```
;; la fonction 2NC crée un arbre de 2**n cons en testant la linéarité
;;
;;     (defun 2NC (n)
;;         (if (> n 0) (cons (2NC (1- n))
;;                           (2NC (1- n)))))
;;

2NC:        equ       *                ; fonction 2NC
            sub       r4,r15,r0        ; test si n > 0
            cond      psw,*NEGATIVE    ;
            jump      2NC_END          ; sinon retourne NIL
            new       r4,*NEW_CONS     ; alloue une cellule de type cons
            sub       r8,r0,*1         ; n = n - 1
            call      lpc,*2NC         ; appel récursif de 2NC
            setf      r4,r8            ; affectation du CAR
            sub       r8,r0,*1         ; n = n - 1
            call      lpc,*2NC         ; appel récursif de 2NC
            setn      r4,r8            ; affectation du CDR
            add       r0,r4,*0         ; r0 pointe la nouvelle cellule
            return                     ; fin
2NC_END:    add       r0,r15,*0        ; retourne NIL
            return                     ;

;; la fonction CXR parcourt une liste en comptant le nombre de cellules

                    (defun CXR (l)
                        (cond ((consp l)
                                (+ 1 (CXR (car l))
                                     (CXR (cdr l))))
                              (t 0)))
```

Fig. 91 - Fonction de création et de parcours de listes

4.6. Un petit évaluateur Lisp

Le langage le plus utilisé en Intelligence Artificielle est sans conteste Lisp. L'exemple de la figure suivante donne la structure générale d'un évaluateur du dialecte Lisp "Scheme" tel qu'il pourrait être programmé sur KIM20 [112]. Le rôle d'un évaluateur, rappelons-le, est de retourner la valeur d'un objet Lisp en fonction de son type.

```
; début de l'évaluateur d'ISALISP version 2.0 (T. Porcher)
; à l'appel r0 pointe l'objet à évaluer, au retour r0 stocke le résultat

F_EVAL   first    r4,r0              ; type de l'objet
         cond     r4,#OBJECT         ; est-ce bien un objet ?
         jump     #F_UNDEFINED       ; sinon indéfini
         case     r4,r4,#TYPE        ; traitement du type
         branch   r4,#TAB_OBJ        ; branchement multiple

TAB_OBJ  return                     ; ()
         jump     #ER_UNBOUND        ; unbound
         jump     #EV_CONS           ; cons
         return                      ; caractère
         return                      ; chaîne
         return                      ; vecteur
         jump     #EV_SYMBOL         ; symbole
         return                      ; fixnum
         return                      ; bignum
         return                      ; rationnel
         return                      ; réel
         return                      ; complexe
         jump     #ER_EVAL           ; stream
         jump     #ER_EVAL           ; process
         jump     #ER_EVAL           ; réservé
         jump     #ER_EVAL           ; réservé

; ensuite évaluation des objets non auto-évaluables (EV_SYMBOL...)

; exemples de quelques primitives
P_CAR    first    r0,r1              ; (car liste)
         return
P_CDR    next     r0,r1              ; (cdr liste)
         return
P_CONS   new      r0,#CONS           ; (cons car cdr)
         setf     r0,r1
         setn     r0,r2
         return
```

Fig. 92 - Un évaluateur "Scheme" simplifié

Ainsi, trois grandes classes d'objets doivent être traitées :

1) les objets auto-évaluables (les objets sont leur propre valeur) comme les entiers, flottants et autres atomes,

2) les listes dont chaque premier élément doit être une fonction à appliquer sur les autres éléments de la liste,

3) les symboles, dont la valeur doit être recherchée dans un champ spécifique de leur descripteur.

Le code source de l'évaluateur montre l'utilisation des instructions de calcul de branchement indexé sur le type de l'objet à évaluer, ainsi que la table de branchement associée. En outre, quelques classiques primitives Lisp mettent en évidence l'adéquation du jeu d'instructions pour le traitement symbolique.

4.7. Evaluation des performances

L'évaluation précise des performances d'une machine informatique est un art difficile qui demande rigueur et objectivité.

Nous ne donnerons ici que le résultat de plusieurs mesures effectuées sur des machines différentes. Ces mesures n'ont pas pour prétention d'être exhaustives, mais plutôt de donner des ordres de grandeur. Les expérimentations ont été effectuées sur la base du "benchmark de Gabriel" qui sert de référence pour l'évaluation des machines dédiées à l'Intelligence Artificielle [113].

Le tableau suivant donne la synthèse des résultats obtenus en prenant pour étalon le processeur VAX 11/780 exécutant le langage Le_Lisp.

Certaines machines sont présentes deux fois dans cette évaluation : une première fois avec le dialecte Le_Lisp de l'INRIA et une seconde fois avec Common-Lisp. Une rapide comparaison des chiffres dans chaque cas met en évidence que la performance finale ne dépend pas seulement du matériel, mais également du compilateur.

Ceci prouve une nouvelle fois, si besoin était, que la conception d'un processeur ne peut plus être dissociée des compilateurs.

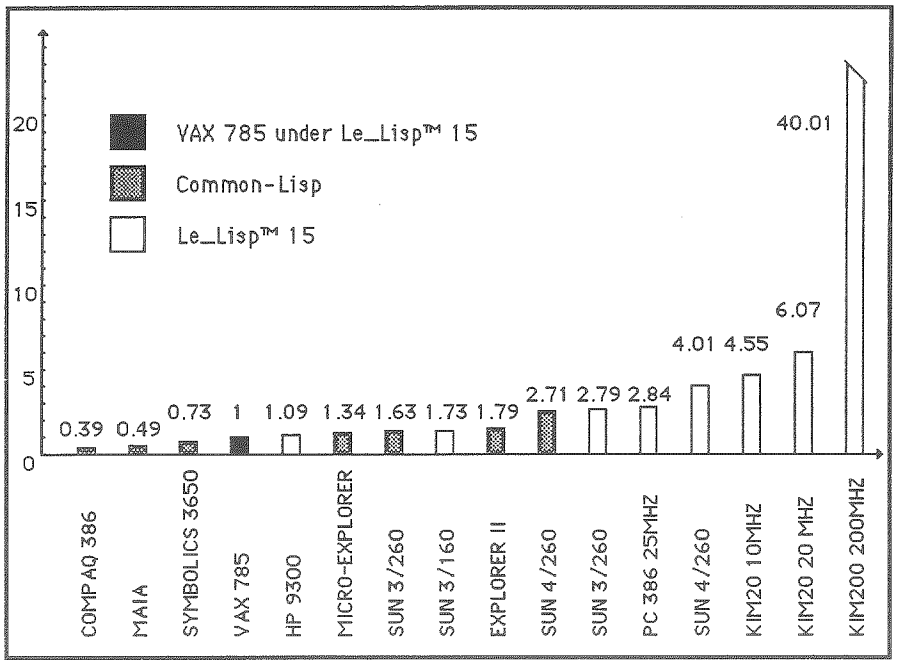
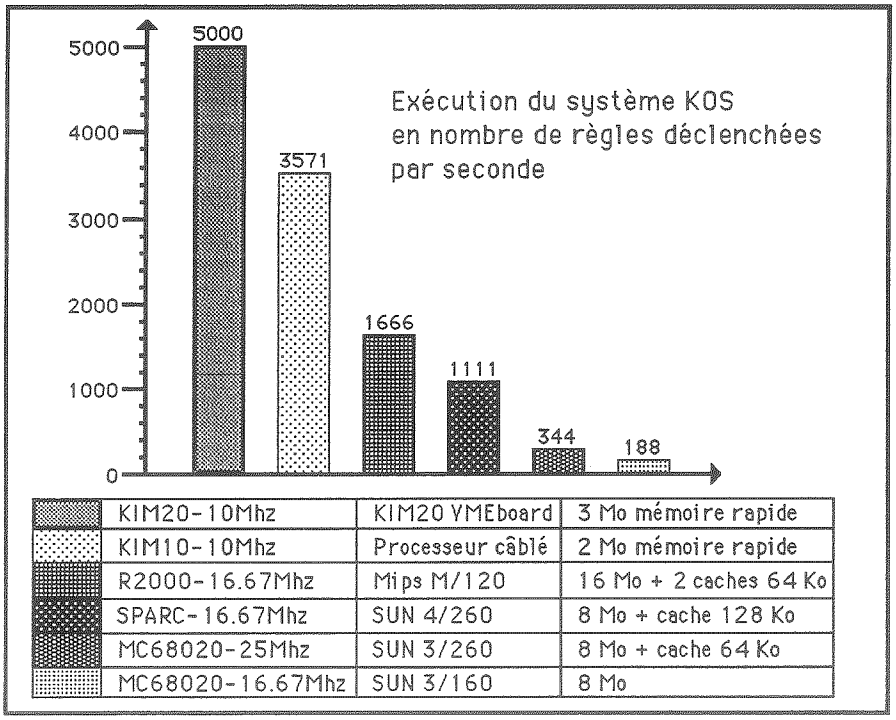


Fig. 93 - Performances comparées des machines Symboliques

Pour conclure au niveau des performances comparées, la figure suivante donne les résultats obtenus avec l'exécution du système exécutif à base de règles KOS, qui constitue une application particulièrement représentative. L'ensemble du logiciel est codé en langage C, auquel est adjointe une bibliothèque spécialisée pour le traitement symbolique, baptisée SPACE (*Symbolique Processsing Added to the C Environment*) [114].

La version KIM de la bibliothèque SPACE est codée en partie en assembleur pour tirer profit des spécificités de l'architecture, alors que pour les autres machines, cette bibliothèque est émulée en C. Nous noterons au passage que ce type d'approche est une bonne alternative aux environnements d'Intelligence Artificielle du type Lisp ou Prolog, par l'adjonction des primitives du traitement symbolique aux langages procéduraux tels que C ou Ada, plus répandus dans l'industrie.



	KIM20-10Mhz	KIM20 VMEboard	3 Mo mémoire rapide
	KIM10-10Mhz	Processeur câblé	2 Mo mémoire rapide
	R2000-16.67Mhz	Mips M/120	16 Mo + 2 caches 64 Ko
	SPARC-16.67Mhz	SUN 4/260	8 Mo + cache 128 Ko
	MC68020-25Mhz	SUN 3/260	8 Mo + cache 64 Ko
	MC68020-16.67Mhz	SUN 3/160	8 Mo

Fig. 94 - Exécution du système KOS

Conclusion

Cet ouvrage a présenté, au cours de ces quatre chapitres, la méthodologie RISC, tant au point de vue théorique que pratique. Le premier chapitre a retracé l'avènement du concept RISC depuis ses balbutiements jusqu'à sa reconnaissance industrielle. Le second chapitre a clairement établi les fondements de la méthodologie. Le troisième chapitre a effectué un large tour d'horizon des processeurs RISC actuellement commercialisés (1989). Le quatrième et dernier chapitre a illustré sa mise en pratique, au travers de l'étude du processeur KIM™20.

Malgré la vigueur de certains détracteurs, de moins en moins nombreux il est vrai, les processeurs RISC envahissent peu à peu l'ensemble des secteurs applicatifs de l'électronique et de l'informatique. Des stations de travail, parmi les plus performantes sont maintenant basées sur des architectures RISC directement issues des travaux de Berkeley et de Stanford. Les microcontrôleurs RISC remplacent les processeurs 8 bits d'autrefois et les coûteux systèmes en tranches utilisés au sein des applications industrielles.

Si la performance reste le critère de choix décisif pour l'utilisateur, c'est la formalisation d'une méthodologie cohérente qui a finalement bousculé les méthodes traditionnelles. Mais cet avènement n'est pas une révolution. Il est la conséquence de l'avancée technologique dans de nombreux domaines dont la microélectronique, la compilation de silicium, l'écriture de compilateurs.

L'architecture RISC représente un pas décisif pour l'informatique moderne. Elle est synonyme de rationnalisation et de pragmatisme. Outre ses applications directes dans les différents secteurs industriels, elle ouvre la voie à une recherche féconde. Les processeurs RISC, du fait de leur faible complexité et de leur adéquation aux techniques d'intégration VLSI, seront les premiers implantés dans les technologies nouvelles du type Sub-micronique en silicium ou Arséniure de Gallium.

Une multitude d'équipes de recherche se concentre sur l'étude d'architectures parallèles, susceptibles de décupler les performances des systèmes informatiques futurs. Déjà la frontière des 100 MIPS sur un circuit VLSI a été franchie, à quand les 1000 MIPS (GIPS ! - Giga Instruction Par Seconde) ?

Finalement, l'avancée technologique due à la méthodologie RISC peut être résumée par la simple équation déjà évoquée au début de l'ouvrage (Chap. 1 § 1.4.). Alors que les concepteurs de processeurs CISC visent l'amélioration des performances en élevant la sémantique des instructions, ce qui corollairement en réduit leur nombre pour l'exécution d'une tâche donnée (paramètre I), les concepteurs de processeurs RISC cherchent avant tout à optimiser conjointement le temps de cycle (paramètre 1/S) et le nombre de cycles nécessaires à l'exécution d'une instruction (paramètre C). Dans le cas d'un processeur d'usage général, cette optimisation ne peut être obtenue sans une dégradation sensible du nombre d'instructions (I). Mais, néanmoins, le résultat global reste excellent. Pour un processeur spécialisé comme KIM20, les trois paramètres peuvent être parfaitement optimisés du fait de la réduction du champ d'application et, par conséquent, de la meilleure adéquation du jeu d'instructions à ce dernier.

Dans un futur proche, les architectures super-scalaires et VLIW viendront "inverser" le paramètre C en permettant l'exécution de plusieurs instructions par cycle machine. Mais, l'avenir des architectures informatiques réside dans l'exploitation intensive du parallélisme. En effet, à l'équation initiale, nous pouvons ajouter le terme N, qui correspond au nombre de processeurs, et la fonction K qui pondère l'efficacité globale de l'architecture parallèle. Si nous savons actuellement construire des machines comprenant plusieurs milliers de processeurs (N grand) le principal obstacle reste encore celui de l'exploitation logicielle efficace de telles architectures.

$$P_t = K \left[N . \frac{S}{I_t . C} \right]$$

P représente la performance en MIPS pour l'exécution de la tâche t (homogène à une fréquence)
I représente le nombre d'instructions du "benchmark" t
C représente le nombre moyen de cycles par instruction
S représente la fréquence d'horloge du processeur
N correspond au nombre de processeurs
K représente une fonction variant entre 0 et 1 qui pondère l'efficacité globale de l'architecture. Elle dépend de l'architecture et du couplage entre les processeurs imposé par la tâche t.

Fig. 95 - Calcul de la puissance d'une architecture

Remerciements

Je tiens particulièrement à remercier tous ceux qui, par leur confiance, leurs compétences et leur dynamisme, soutiennent le projet KIM.

La première phase du projet a été en partie financée par l'ANVAR (Association Nationale pour la Valorisation de la Recherche), le Département du Val-de-Marne et la DRET (Direction de la Recherche Etudes et Techniques). L'intégration du processeur a été en partie financée par la DRET et le SERICS (Service des Industries et Communications et de Services).

J'exprime ma plus profonde gratitude aux chercheurs du laboratoire "Système de Perception" de l'ETCA (Etablissement Technique Central de l'Armement), et plus particulièrement au Professeur Bertrand Zavidovique, sans qui le projet KIM ne serait pas devenu ce qu'il est. Un coup de chapeau également aux chercheurs de l'IEF (Institut d'Electronique Fondamentale) de l'Université d'Orsay (Paris Sud/CNRS) et au Professeur Francis Devos en particulier.

Le projet KIM n'aurait sans doute pas abouti sans la participation active de l'équipe KIM au sein de la Société SODIMA : Lydie Bol, Christophe Métivier, Jean-Pierre Courrier, Eric Peltier et bien d'autres encore.

Un grand merci également à Robert Malka, Président Directeur Général de la Société SODIMA, pour son amitié et sa confiance.

Le projet KIM est une preuve irréfutable qu'il est possible de concevoir et réaliser un processeur en France. Gageons que les partenaires industriels potentiels accueilleront avec bienveillance le processeur KIM comme une alternative séduisante aux solutions américaines.

Jean-Claude Heudin

Annexe

LE JEU D'INSTRUCTIONS DU PROCESSEUR KIM20

ADD Integer signed addition

Syntax ADD Rd, S1, [#]S2

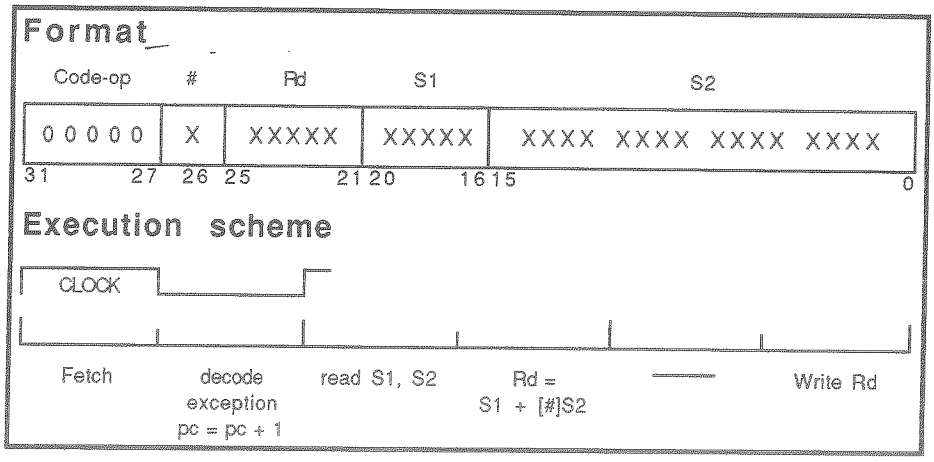
Operation Rd = S1 + [#]S2

Description Adds the content of source register S1, with the content of source register [#]S2, into the destination register Rd.

Condition codes (depending of the ALU-SIZE field in the Status-Word)

- C Set if a carry generated, otherwise cleared.
- Z Set if the result is zero, otherwise cleared.
- N Set if the result is negative, otherwise cleared.
- V Set if an arithmetic overflow is detected, otherwise cleared.

Format



Execution scheme

SUB Integer signed subtraction

Syntax SUB Rd, S1, [#]S2

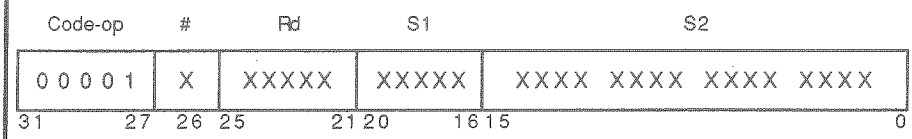
Operation Rd = S1 - [#]S2

Description Subtracts the content of source register [#]S2 from the content of source register S1, into the destination register Rd.

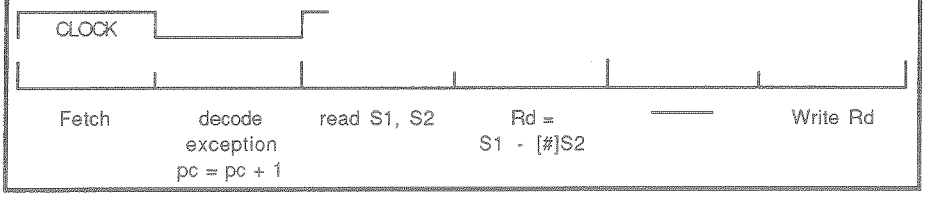
Condition codes (depending of the ALU-SIZE field in the Status-Word)

- C Set if a carry generated, otherwise cleared.
- Z Set if the result is zero, otherwise cleared.
- N Set if the result is negative, otherwise cleared.
- V Set if an arithmetic overflow is detected, otherwise cleared.

Format



Execution scheme



AND — Logical AND

Syntax AND Rd, S1, [#]S2

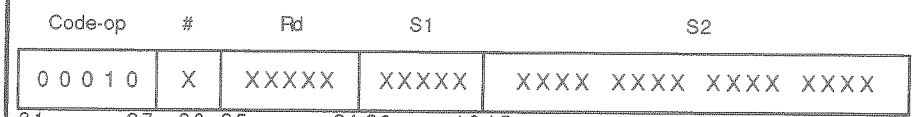
Operation Rd = S1 and [#]S2

Description Computes a logical AND between the content of source register S1, and the content of source register [#]S2, into the destination register Rd.

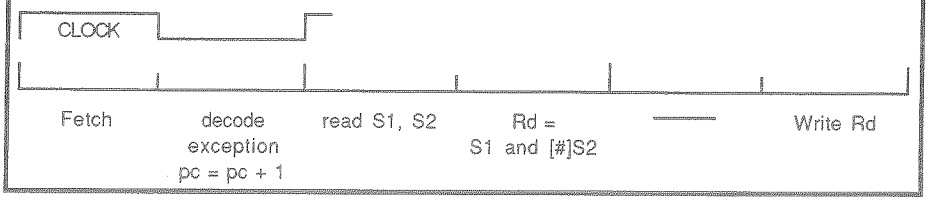
Condition codes (depending of the ALU-SIZE field in the Status-Word)

- C Always cleared.
- Z Set if the result is zero, otherwise cleared.
- N Set if the result is negative, otherwise cleared.
- V Always cleared.

Format



Execution scheme



OR — Logical OR

Syntax OR Rd, S1, [#]S2

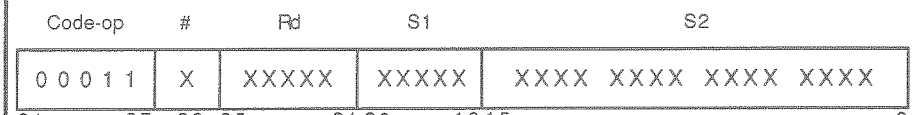
Operation Rd = S1 or [#]S2

Description Computes a logical OR between the content of source register S1, and the content of source register [#]S2, into the destination register Rd.

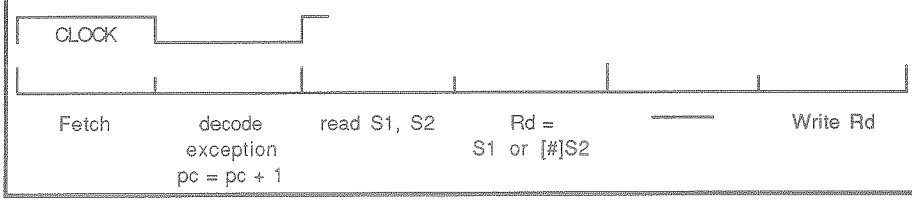
Condition codes (depending of the ALU-SIZE field in the Status-Word)

- C Always cleared.
- Z Set if the result is zero, otherwise cleared.
- N Set if the result is negative, otherwise cleared.
- V Always cleared.

Format



Execution scheme



XOR Logical exclusive OR

Syntax XOR Rd, S1, [#]S2

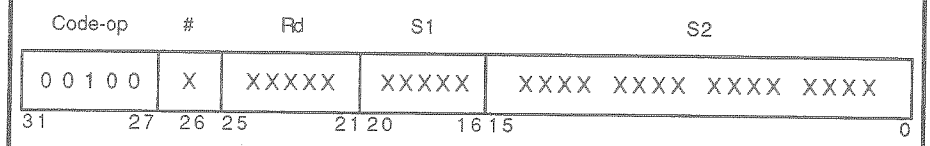
Operation Rd = S1 xor [#]S2

Description Computes a logical exclusive OR between the content of source register S1, and the content of source register [#]S2, into the destination register Rd.

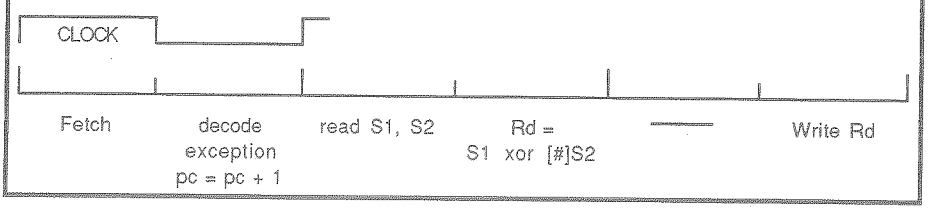
Condition codes (depending of the ALU-SIZE field in the Status-Word)

- C Always cleared.
- Z Set if the result is zero, otherwise cleared.
- N Set if the result is negative, otherwise cleared.
- V Always cleared.

Format



Execution scheme



ROTATE — Left rotate

Syntax ROTATE Rd, S1, [#]S2

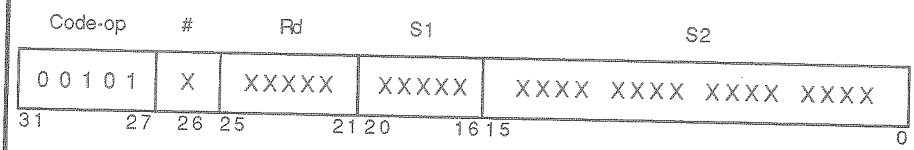
Operation Rd = S1 rotated [#]S2 times

Description Computes a left rotation on the content of source register S1, into the destination register Rd. The rotate count is specified by the source register [#]S2.

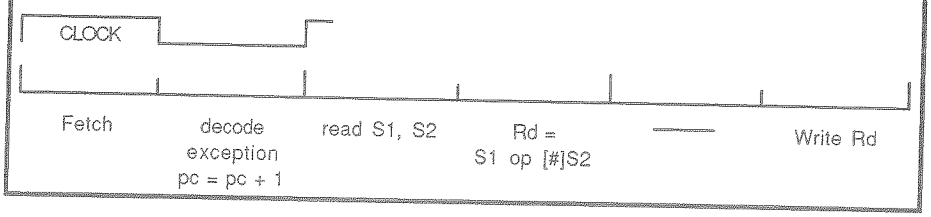
Condition codes (depending of the ALU-SIZE field in the Status-Word)

- C Always cleared.
- Z Set if the result is zero, otherwise cleared.
- N Set if the result is negative, otherwise cleared.
- V Always cleared.

Format



Execution scheme



SHIFTRL Right logical shift

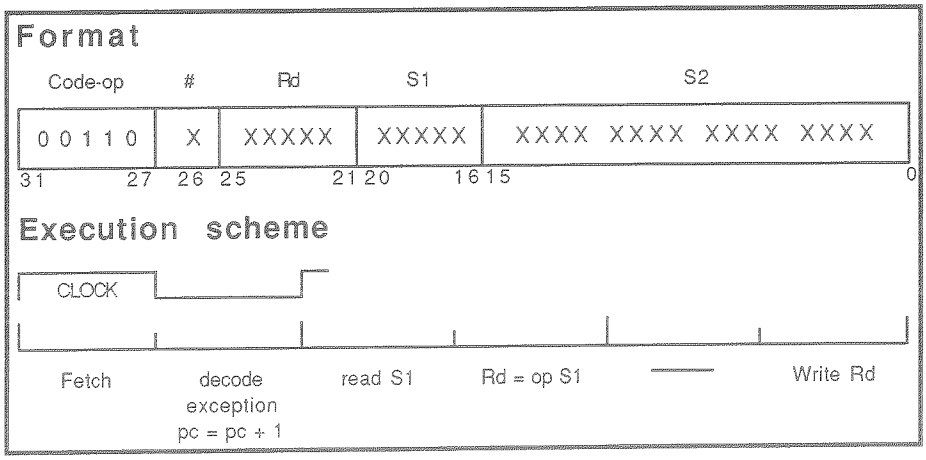
Syntax SHIFTRL Rd, S1

Operation Rd = S1 right shifted, Rd <31> = 0

Description Computes a one-bit right shift on the source register S1 into the destination register Rd. The carry is set to the value of bit 0 of the source register S1, and the uppest bit of the destination register is cleared.

Condition codes (depending of the ALU-SIZE field in the Status-Word)

C Equal to bit 0 of S1.
Z Set if the result is zero, otherwise cleared.
N Set if the result is negative, otherwise cleared.
V Always cleared.



SHIFTRA Right arithmetical shift

Syntax SHIFTRA Rd, S1

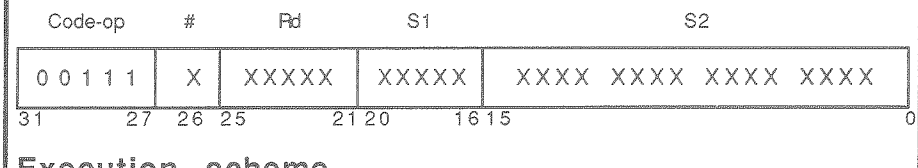
Operation Rd = S1 right shifted, Rd <31> = carry

Description Computes a one-bit right shift on the source register S1 into the destination register Rd. The uppest bit of the destination register is set to the value of the carry, and the carry is set to the value of bit 0 of the source register S1.

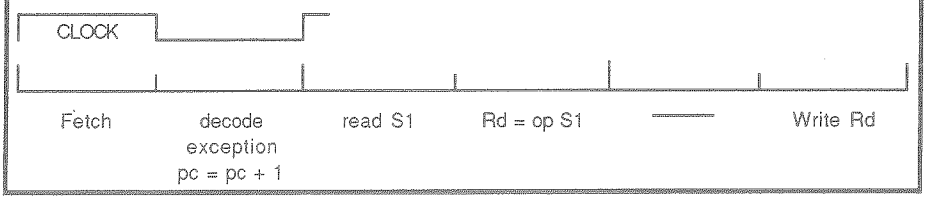
Condition codes (depending of the ALU-SIZE field in the Status-Word)

- C Equal to bit 0 of S1.
- Z Set if the result is zero, otherwise cleared.
- N Set if the result is negative, otherwise cleared.
- V Always cleared.

Format



Execution scheme



INSERT — Byte insertion

Syntax INSERT Rd, S1, [#]S2

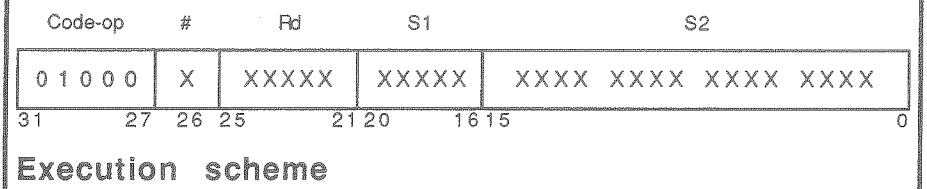
Operation
Rd = 0 ;
Rd<byte nb.<[#]S2<1:0>>> = S1<7:0>

Description Extracts the lowest byte from source register S1 and insert it into the byte of the destination register Rd indicated by the two lowest bits of the operand [#]S2. The other bytes of the destination register are cleared.

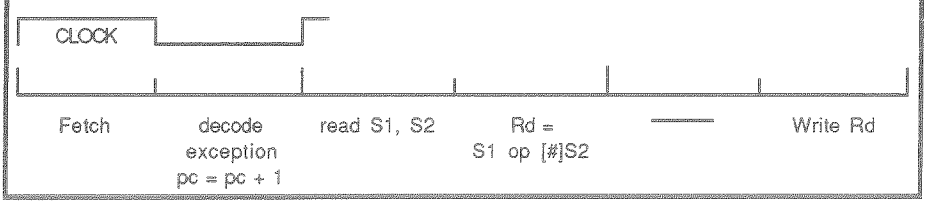
Condition codes (depending of the ALU-SIZE field in the Status-Word)

- **C** Always cleared.
- **Z** Set if the result is zero, otherwise cleared.
- **N** Set if the result is negative, otherwise cleared.
- **V** Always cleared.

Format



Execution scheme



EXTRACT Byte extraction

Syntax EXTRACT Rd, S1, [#]S2

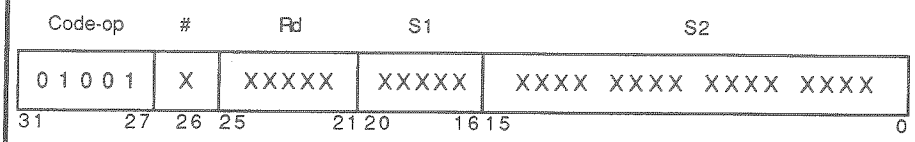
Operation
Rd = 0 ;
Rd<0:7> = S1<byte nb.<[#]S2<1:0>>>

Description Extracts from source register S1, the byte indicated by the two lowest bits of the operand [#]S2 (highest byte is 3, lowest byte is 0) and insert it into the lowest byte of the destination register Rd. The other bytes of the destination register are cleared.

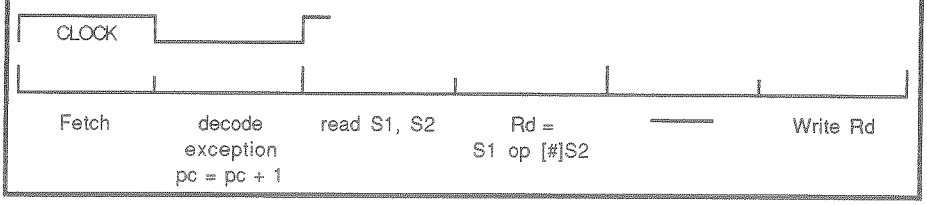
Condition codes (depending of the ALU-SIZE field in the Status-Word)

- C Always cleared.
- Z Set if the result is zero, otherwise cleared.
- N Set if the result is negative, otherwise cleared.
- V Always cleared.

Format



Execution scheme



HASH Hash code processing

Syntax Hash Rd, [#]S2

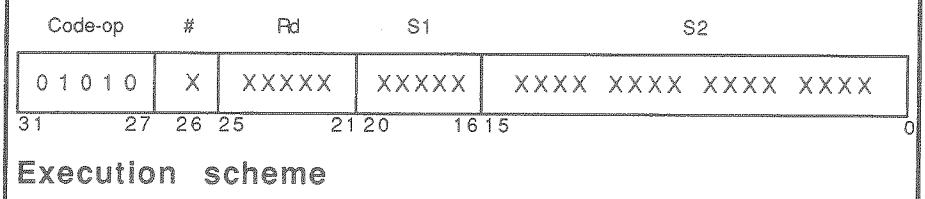
Operation
Rd<9:0> =
S2<31:24>+S2<23:16>+S2<15:8>+S2<7:0>
Rd<31:10> = 0

Description Adds the 4 bytes of source register S2 into the destination register Rd. The result is 10 bit significant.

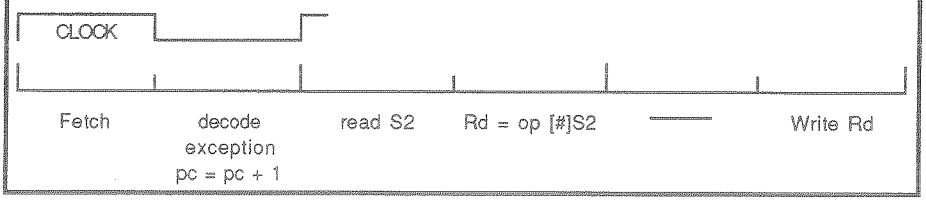
Condition codes (depending of the ALU-SIZE field in the Status-Word)

C Always cleared.
Z Set if the result is zero, otherwise cleared.
N Always cleared.
V Always cleared.

Format



Execution scheme



HYPER — Message routing

Syntax HYPER Rd, S1, [#]S2

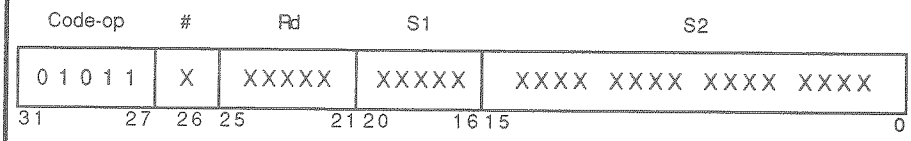
Operation Rd = 0 ; Rd<4:0> = nb. of the most significant bit of (S1 xor [#]S2)

Description Computes an exclusive OR, between the source register S1 and the source register [#]S2, and writes into the destination register Rd the number of the most significant bit of the intermediate result.

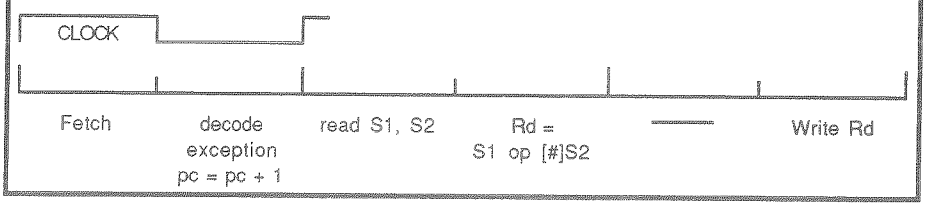
Condition codes (depending of the ALU-SIZE field in the Status-Word)

C Always cleared.
Z Set if the result is zero, otherwise cleared.
N Always cleared.
V Always cleared.

Format



Execution scheme



CASE Case processing

Syntax CASE Rd, S1, [#]S2

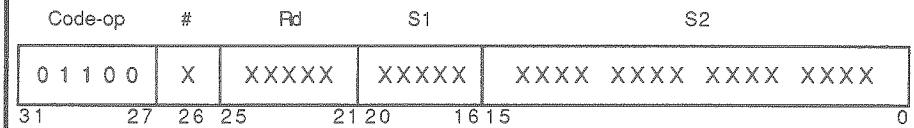
Operation
Rd = 0 ;
Rd<7:0> = S1<31:24> and S2<7:0>

Description Computes a logical AND between the highest byte of source register S1, and the lowest byte of source register [#]S2 and writes the result into the lower by of destination register Rd. The three higher bytes of destination register Rd are set to null.

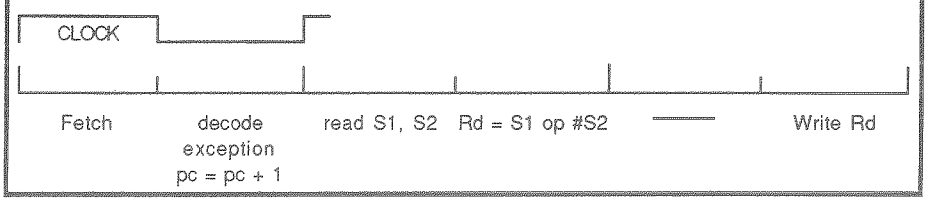
Condition codes (depending of the ALU-SIZE field in the Status-Word)

C Always cleared.
Z Set if the result is zero, otherwise cleared.
N Always cleared.
V Always cleared.

Format



Execution scheme



CALL — Call function

Syntax CALL Rd, #S2

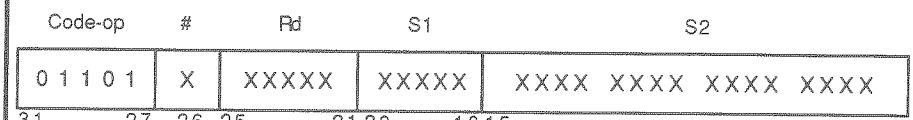
Operation PC_save = PC + 1 (1) or PC_excep (2) ;
PC = #S2 ; WP = WP + 1 ; Test overflow ;
Rd = PC_save

Description Computes a window change (WP = WP + 1) and a branch to the immediate address #S2. After incrementation, if WP becomes equal to WO, a window overflow exception is generated. The function return address is saved into destination register Rd. This return address is PC + 1 (1) unless this call is the first being executed by an exception handling procedure. In that case the return address is the address of the interrupted instruction (2).

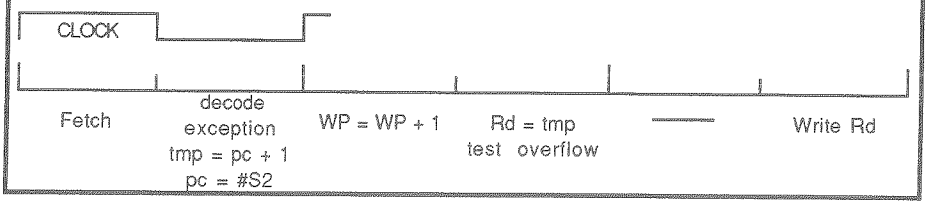
Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.

Format



Execution scheme



RETURN — Return from a function

Syntax RETURN

Operation

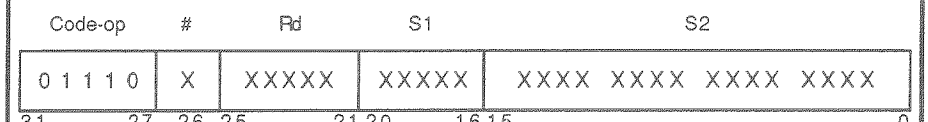
WP = WP - 1 ;
PC<15:0> = PC_saved<15:0>
Base register = PC _saved<23:16>

Description Computes a window change (WP = WP - 1) and a branch operation to the address saved during the corresponding call. The branch operation changes the content of the PC and base registers. If before decrementation WP is equal to WO a window underflow exception is generated.

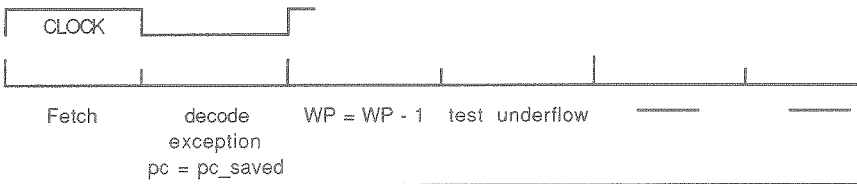
Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.

Format



Execution scheme



BRANCH Indexed branching

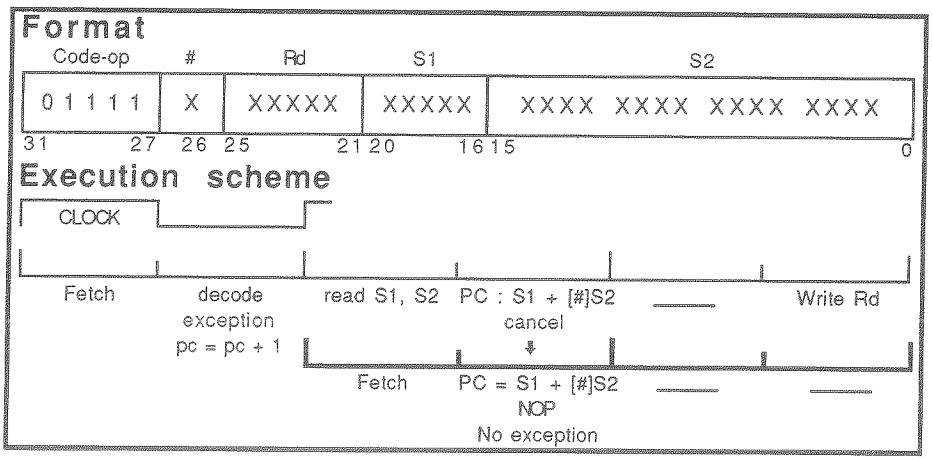
Syntax BRANCH S1, [#]S2

Operation PC = S1 + [#]S2

Description Computes a branch operation to the address resulting of the addition of the source register S1 and the source register [#]S2. This instruction needs two cycles to complete. A BRANCH instruction cancels automatically the execution of the following instruction. Exceptions are disabled during the second cycle of the BRANCH instruction.

Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.



JUMP Immediate jump

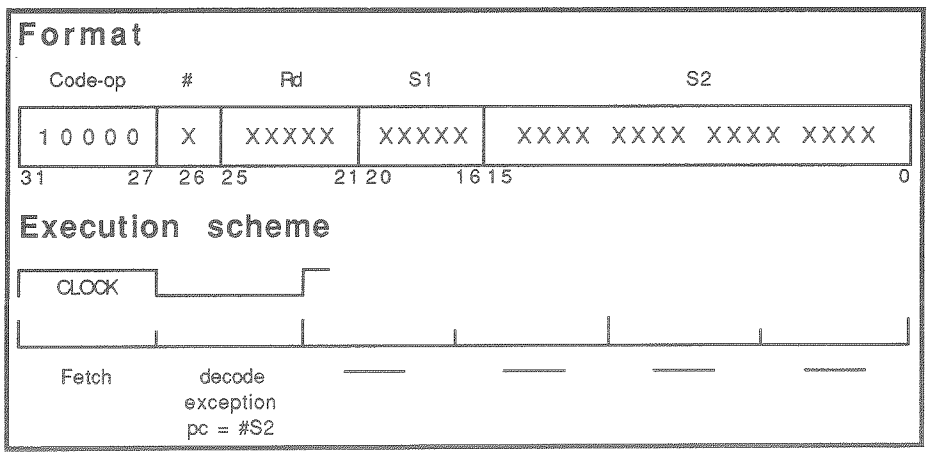
Syntax JUMP #S2

Operation PC = #S2

Description Computes a branch operation to the immediate 16-bit operand #S2 address.

Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.



COND — Condition test

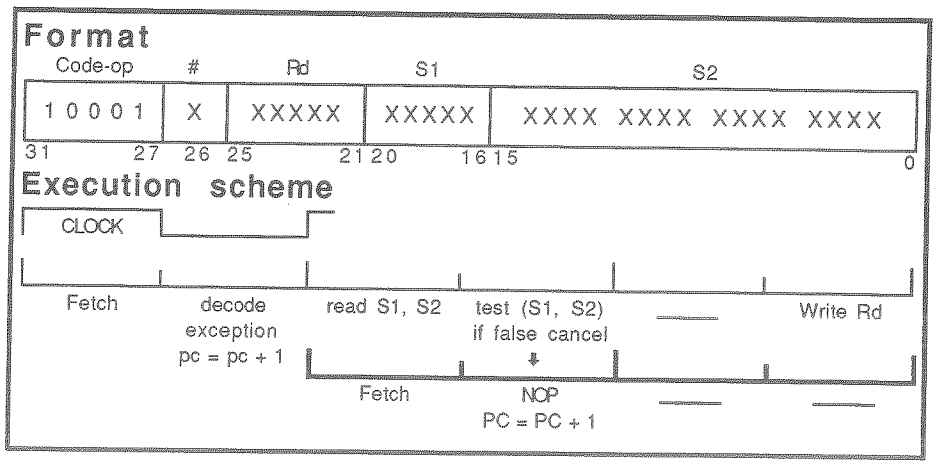
Syntax COND S1, [#]S2

Operation Cancel the following instruction if test (S1, S2) is false.

Description Computes upon the more significant byte of source register S1 the test specified by the operand [#]S2. If false, the execution of the following instruction is cancelled. The different tests are specified on the next page.

Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.



COND Condition test

Description of the test determined by [#]S2 :

- bits 0-7 : test2,
- bits 8-11 : test1,
- bits 12-14 : composed test with test1 and test2,
- bit 15 : final test of the cond instruction.

Test 2 :

Test if : [#]S2<7:4> and [#]S2<3:0> = S1<27:24> and [#]S2<3:0> is true

Test 1 : i = [#]S2<11:8>

i	formula	comments
0	<28>	overflow, mark2
1	<29>	negative, minus, mark1
2	<30>	zero, equal
3	<31>	carry, cdr-escape, higher or same*
4	<29> or <28>	less than
5	(<29> xor <28>) + 30	less or equal
6	<29> + <30>	negative or zero
7	!<30> and !<31>	invisible cell
8	!<30> and !<31>	cdr-cell
9	!<30> and !<31>	cdr-next
10	<30> and <31>	cdr-nil
11 to 15	0	(false)

Macro test : i = [#]S2<14:12>

0	test1
1	test2
2	!test1 and !test2
3	!test1 and test2
4	test1 and !test2
5	test1 and test2
6 or 7	0 (false)

Final test of the instruction : ([#]S2<15>)

If this bit is set, the result of the macro-test is inverted. If the final test is true, then the following instruction is canceled.

TRAP Software exception

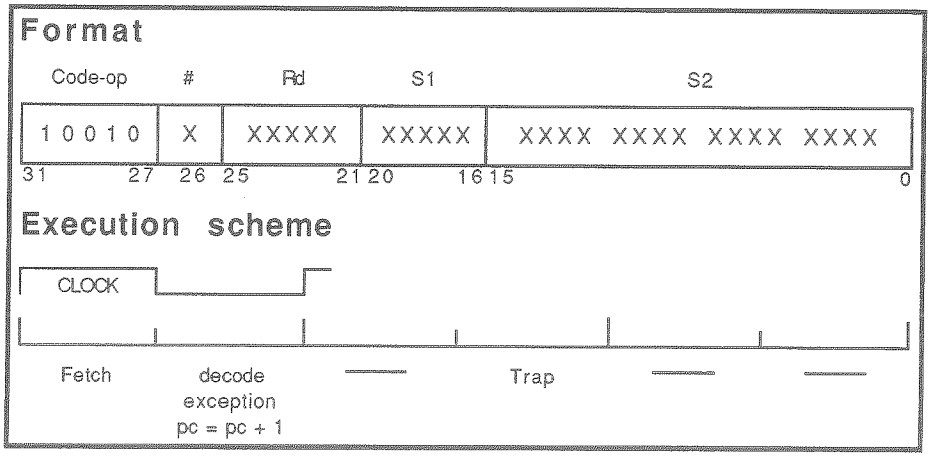
Syntax TRAP #S2

Operation Generates a level 5 exception.

Description This instruction generates a level 5 exception which is processed as any other exception. Operand #S2 has no specific meaning.

Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.



RETE Return from exception

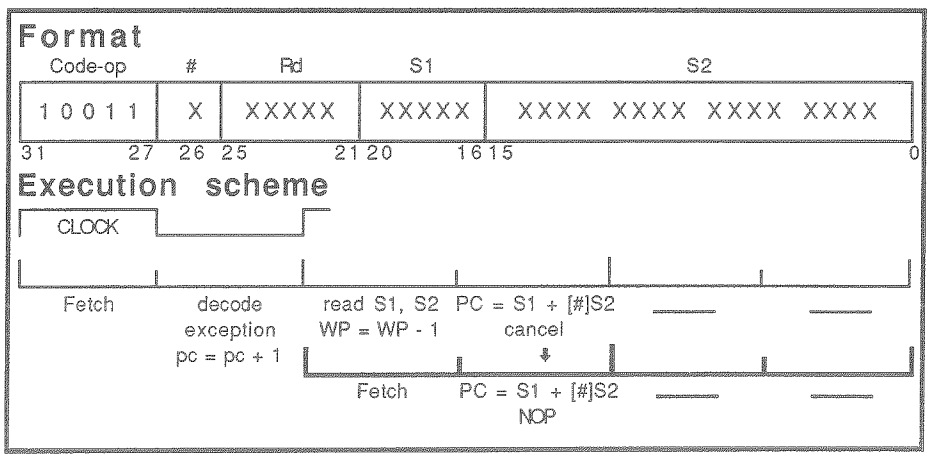
Syntax RETE S1, [#]S2

Operation PC = S1 + [#]S2 ; WP = WP - 1
Base register = PC_save <23:16>

Description Computes a window change (WP = WP - 1) and a branch operation to the address which results form the addition of the source register S1 and the source register [#]S2. This instruction needs two cycles to complete. A RETE instruction cancels automatically the execution of the following instruction. Exceptions are disabled during second cycle of the RETE instruction.

Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.



FIRST Read the "FIRST" field

Syntax FIRST Rd, S1, [#]S2

Operation

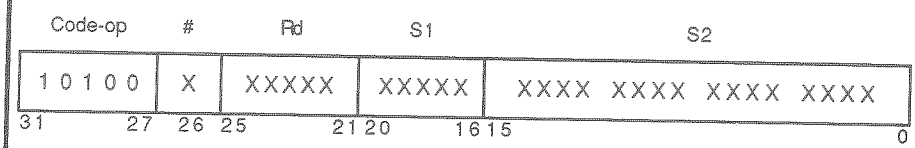
Rd<23:0> = DATA(S1)<23:0>
If [#]S2<0>= 0 then
    Rd<31:24> = DATA(S1)<31:24>
If [#]S2<0> = 1 then
    Rd<31:24> = 0

Description Performs a read of the cell located at the address given by the register S1 into the destination register Rd. If the lowest bit of the operand [#]S2 is set, the most significant byte of the destination register Rd is cleared.

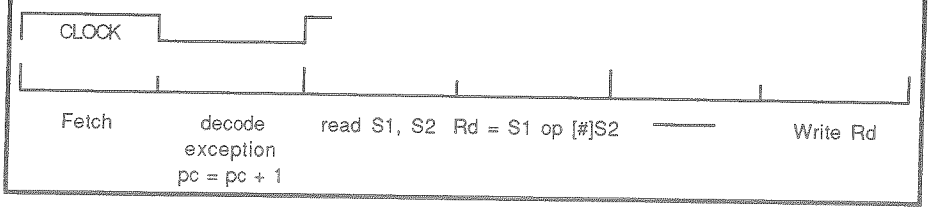
Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.

Format



Execution scheme



NEXT Read the "NEXT" field

Syntax NEXT Rd, S1

Operation

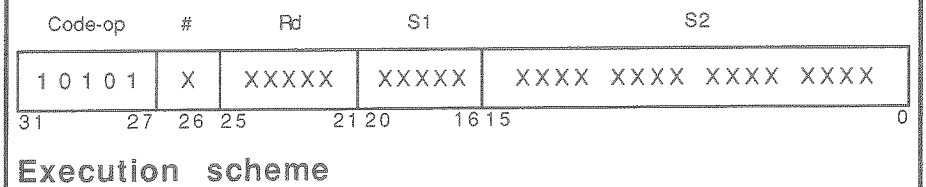
Rd<31:24> = DATA(S1)<31:24>
If Rd<31:30> = 3 then Rd<23:0> = 0
If Rd<31:30> = 2
Then Rd<23:0> = S1<23:0> + 1
Else Rd<23:0> = 0 and CDR-exception

Description Performs a read of the "NEXT" field of the cell located at the address given by the register S1, and writes its extended value with the tag into the destination register Rd. The code 3 corresponds to the value 0 (nil) the code 2 to (S1<23:0> + 1) (next cell) and the other codes to 0, with the generation of a CDR exception.

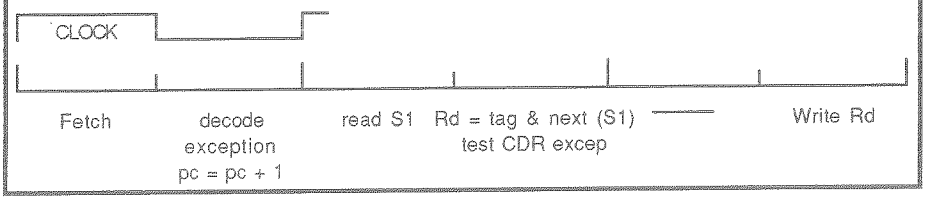
Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.

Format



Execution scheme



SETF Write the "FIRST" field

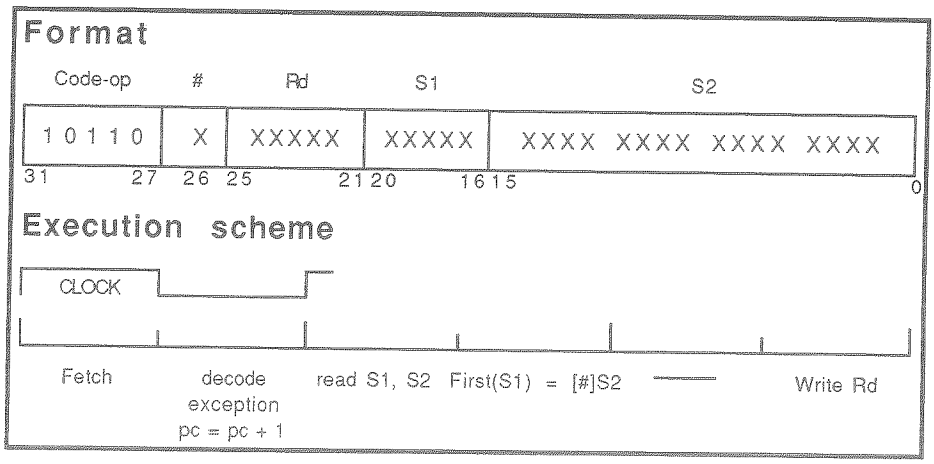
Syntax SETF S1, [#]S2

Operation DATA(S1)<23:0> = [#]S2<23:0>

Description Writes the "FIRST" field of the cell pointed to by the source register S1, with the content of the source operand [#]S2.

Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.



SETN Write the "NEXT" field

Syntax SETN S1, [#]S2

Operation

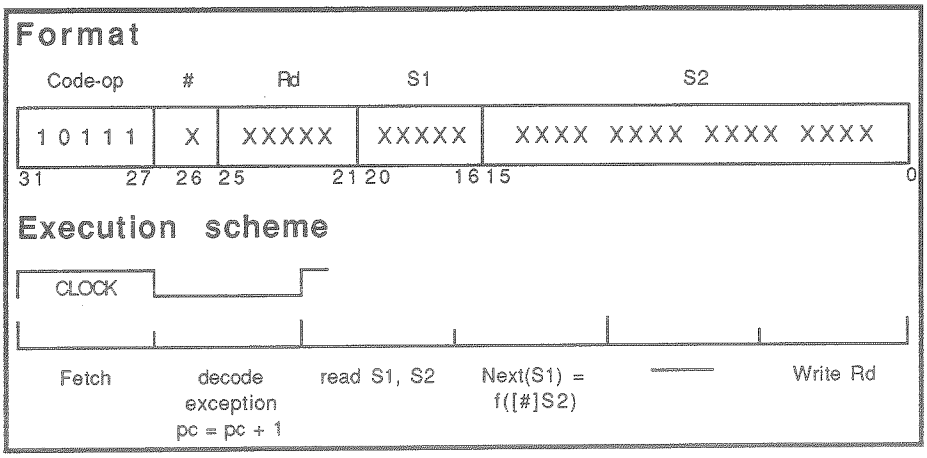
If [#]S2 = 0 then DATA(S1)<31:30> = 3
If [#]S2 = S1 + 1
then DATA(S1)<31:30> =2
Else DATA(S1)<31:30> = 0
and CDR exception

Description Write the "NEXT" field of the cell pointed to by the source register S1. The CDR code is computed regarding to the operands S1 and [#]S2.

Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.

Format



Execution scheme

WTAG Write the "TAG" field

Syntax WTAG S1, [#]S2

Operation

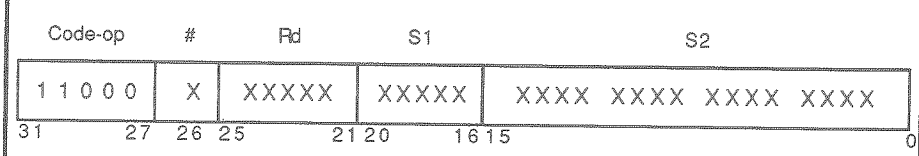
If [#]S2<6> = 1 DATA(S1)<31:30> = [#]S2<15:14>
Else DATA(S1)<31:30> = 0
If [#]S2<5> = 1 DATA(S1)<29> = [#]S2<13>
Else DATA(S1)<29> = 0
If [#]S2<4> = 1 DATA(S1)<28> = [#]S2<12>
Else DATA(S1)<28> = 0
If [#]S2<3> = 1 DATA(S1)<27:24> = [#]=S2<11:8>
Else DATA(S1)<27:24> = 0

Description Writes the "TAG" field of the cell pointed to by the source register S1. The value to write is contained in the 2nd byte of source operand [#]S2, while the first byte is a mask that authorizes the modification of the different fields of the tag (bit 3➡ type field ; bit 4➡ mark1 field ; bit 5➡ mark2 field; bit 6➡ cdr-code field).

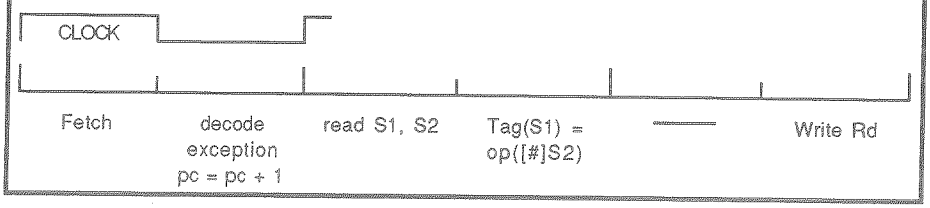
Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.

Format



Execution scheme



NEW — Cell allocation

Syntax NEW Rd, [#]S2

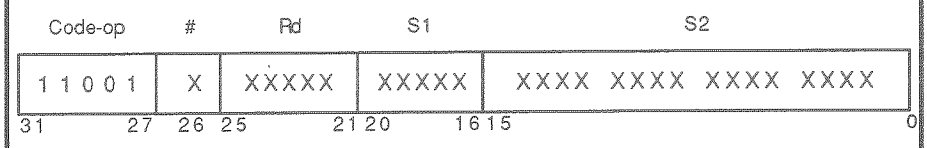
Operation Rd = FLP ; FLP = DATA(FLP)<23:0>
DATA(FLP)<31:24> = [#]S2 <15:8> masked by [#]S2<7:0>
If FLC<= FLT then exception ; FLC = FLC - 1

Description Allocation of a new cell, from the current free-cell list called "Free-List". The tag field of the allocatec cell is initialized with the content of the source operand [#]S2, like in the "WTAG" instruction. The address of the allocated cell is saved in the destination register Rd. The address of the first cell of the Free-List is automatically updated in FLP.

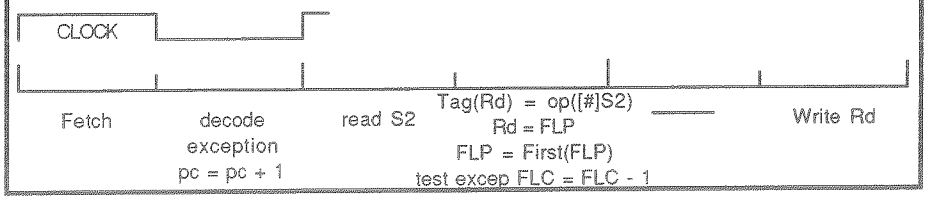
Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.

Format



Execution scheme



FREE — Cell garbaging

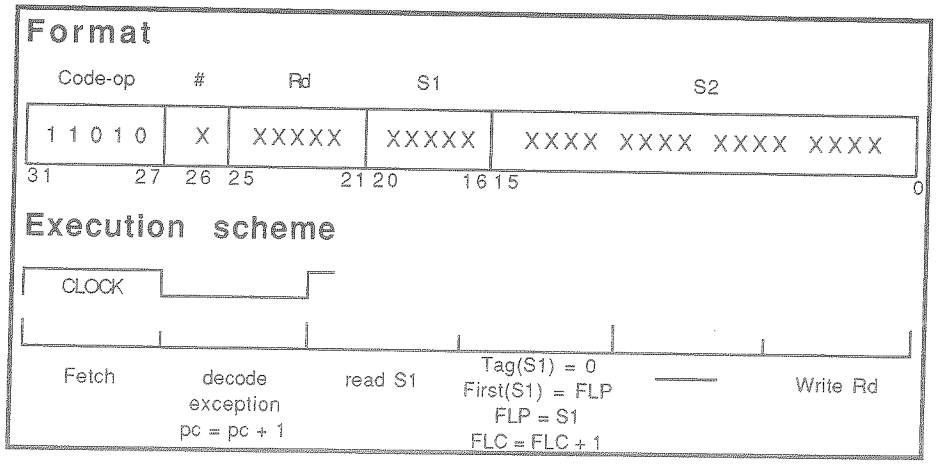
Syntax FREE S1

Operation
DATA(S1)<31:24> = 0
DATA(S1)<23:0> = FLP
FLP = S1<23:0> ; FLC = FLC + 1

Description Merges a free cell pointed to by the source register S1 at the top of the current "Free-List". The tag is automatically cleared, the CDR-code is cleared, which assumes that the free cells are chained with their "FIRST" field.

Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.



SEND Write the data memory

Syntax SEND S1, [#]S2

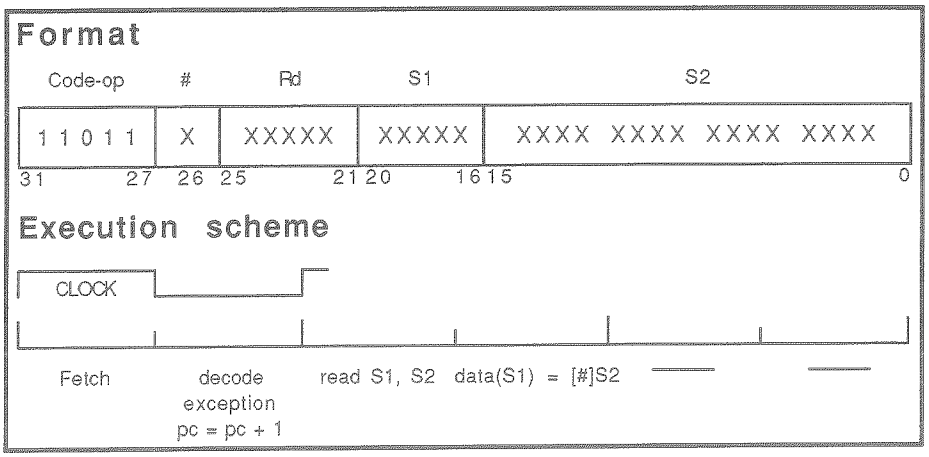
Operation DATA(S1)<31:0> = [#]S2<31:0>

Description Writes a 32 bit word into the data memory. The source register S1 contains the memory address, while the source operand [#]S2 contains the 32 bit word to write at this address.

Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.

Format



Execution scheme

PUSH — Push a data in a LIFO stack

Syntax PUSH SP, [#]S2

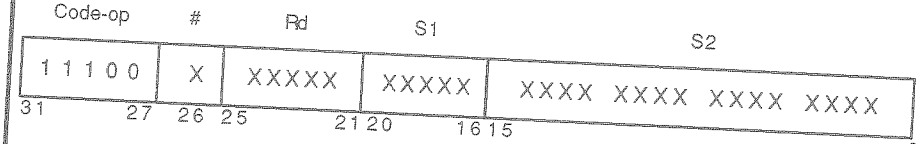
Operation SP = SP + 1 ; If SP<7:0> = FF then exception
DATA(SP)<31:0> = [#]S2<31:0>

Description Pushes a 32 bit word in a LIFO stack mapped in the data memory. The source register SP contains the address of the top of the LIFO stack, while the source [#]S2 ; the 32 bit word to push. If the lowest byte of SP becomes equal to FF (hexadecimal) a stack underflow exception is generated.

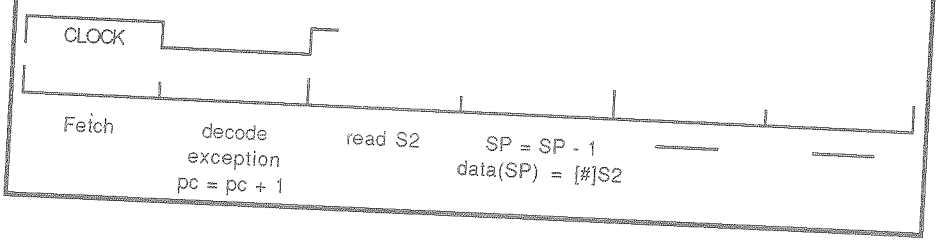
Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.

Format



Execution scheme



POP Pop from a LIFO stack

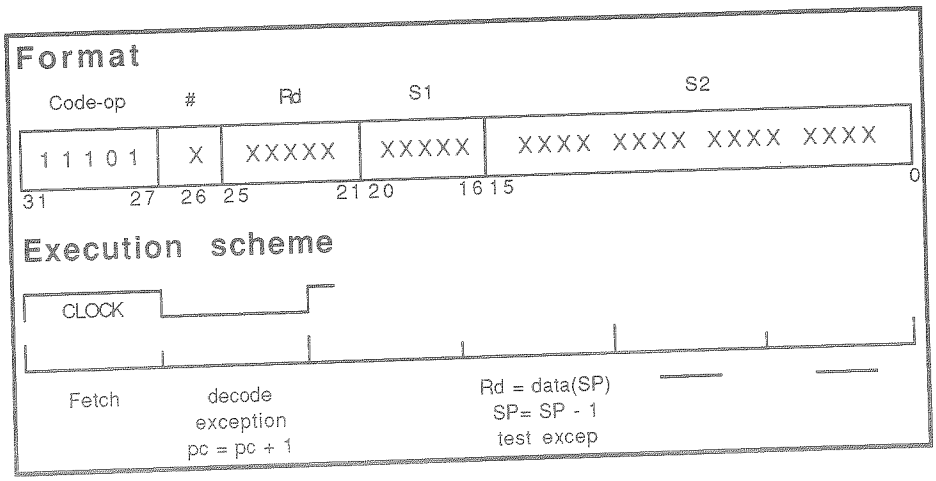
Syntax POP Rd

Operation Rd<31:0> = DATA(SP)<31:0>
SP = SP - 1 ; If SP<7:0> = FF then exeption

Description Pops out a value from the current LIFO stack pointed to by the register SP. The result is stored in the destination register Rd. If the lowest byte of SP becomes equal to FF (hexadecimal) a stack underflow exception is generated.

Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.



LOAD Read code memory

Syntax LOAD Rd, S1

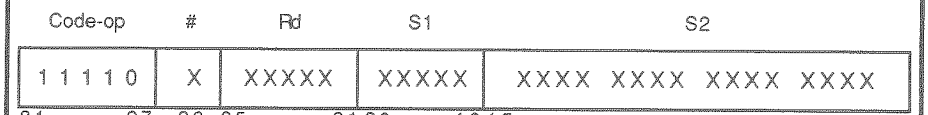
Operation Rd<31:0> = CODE(S1)<31:0>

Description Performs a read of a 32 bit word of code memory at the address contained in the source register S1. The word read is stored into the destination register Rd.

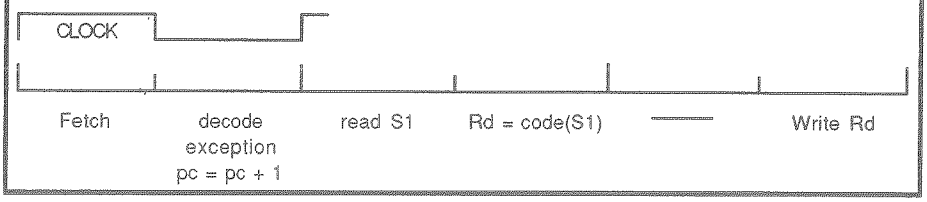
Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.

Format



Execution scheme



STORE Write code memory

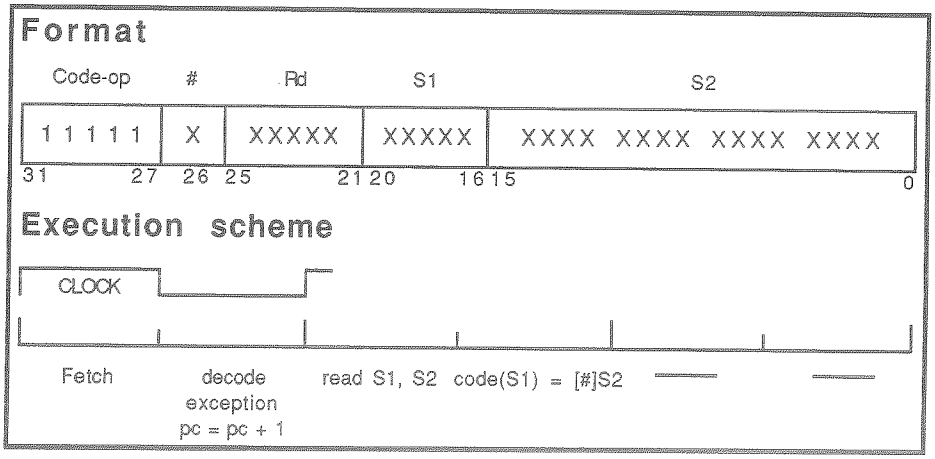
Syntax STORE S1, [#]S2

Operation CODE(S1)<31:0> = [#]S2<31:0>

Description Writes a 32 bit word into code memory at the address contained in the source register S1. The source operand [#]S2 contains the 32 bit word to write at this address.

Condition codes

C Unmodified.
Z Unmodified.
N Unmodified.
V Unmodified.



Bibliographie

[1] MOTOROLA INC. - *MC68020 User's Manual.* 1984

[2] W.D. STRECKER - *VAX-11/780 : a virtual address extension of the DEC PDP-11 family.* AFIPS Conference Vol. 47, 1978, NCC

[3] K.L. BOWLES - *Problem solving using Pascal.* Springer-Verlag, New-York, 1977

[4] J. CHAILLOUX - *La machine LLM3.* Rapport technique INRIA n° 55, Centre de Rocquencourt, Juin 1985

[5] D.A. FAIRCLOUGH - *A unique microprocessor instruction set.* Brigham University, Revue IEEE Micro, Mai 1982

[6] *The Cray-1 computer system.* Communication Ass. Computer Mach, Vol. 21, n° 1, 1978

[7] G. RADIN - *The 801 minicomputer.* ACM symposium on architectural support for programming languages and operating systems, Palo Alto, California (USA), Mars 1982

[8] C. MEAD, L. CONWAY - *Introduction aux systèmes VLSI.* InterEditions, 1983

[9] D.A. PATTERSON, C.H. SEQUIN - *A VLSI RISC.* UC Berkeley, Revue IEEE Computer, 1982

[10] B.W. KERNIGHAN, D.M. RITCHIE - *The C programming language.* Prentice-Hall, 1978

[11] D.A. PATTERSON, C.H. SEQUIN - *RISC-I : a Reduced Instruction Set VLSI Computer.* UC Berkeley, 8th internation symposium on computer architecture, Mai 1981

[12] DIGITAL EQUIPMENT CORPORATION - *PDP-11 architecture handbook.* 1983

[13] F. BASKETT - *A VLSI Pascal machine.* Public lecture, U.C. Berkeley, 1978

[14] R.L. SITES - *How to use 1000 registers.* Caltech conference on VLSI, Janvier 1979

[15] M.G.H. KATEVENIS - *Reduced instruction set architectures for VLSI.* The MIT Press, 1983

[16] R.A. BLOMSETH - *A big RISC.* USAF/UC Berkeley, UC Berkeley technical report, Juillet 1983

[17] D.A. PATTERSON, P. GARRISON, M. HILL, D. LIOUPIS, C. NYLBERG, T. SIPPEL, K. VAN DYKE - *Architecture of a VLSI Cache for a RISC.* U.C. Berkeley, 10th Annual International Symposium on Computer Architecture, AMC, 1983

[18] D. VUGAR, R. BLAU, P. FOLEY, D. SAMPLES, D.A. PATTERSON - *Architecture of SOAR : Smalltalk on a Risc.* UC Berkeley 11th ACM international symposium on computer architecture, 1984

[19] D. UNGAR, D.A. PATTERSON - *What price Smalltalk ?* IEEE Computer, Janvier 1987

[20] A. GOLDBERG, D. ROBSON - *Smalltalk-80 : the language and its implementation*, Addison-Wesley, 1983

[21] SYMBOLICS INC. - *The Symbolics 3600 technical summary.* Cambridge (MA), 1983

[22] D. SAMPLES, M. KLEIN, P. FOLEY - *SOAR architecture.* UC Berkeley technical report, 1985

[23] C.C. MARINO - *Smalltalk on a RISC-CMOS implementation.* UC Berkeley technical report, Juin 1985

[24] A. ALTMAN - *RISC-taking in symbolic processors.* TI engineering journal, Janvier/Février 1986

[25] M.D. HILL, D. A. PATTERSON et al. - *SPUR : a VLSI multiprocessor workstation.* UC Berkeley technical report, 1985

[26] M.D. HILL, D. A. PATTERSON et al. - *Design decision in SPUR.* Revue IEEE computer, Novembre 1986

[27] J.L. STEEL Jr. - *Common-Lisp : the language.* Digital Press, 1984

[28] J. HENNESSY, N. JOUPPI, F. BASKETT, A. STRONG, T. GROSS, C. REWEN, J. GILL - *The MIPS machine.* Stanford University, Proceeding COMPCON, Février 1982

[29] M. HOROWITZ, P. CHOW, J. HENNESSY et al. - *MIPS-X : A 20-MIPS peak, 32-bit microprocessor with on-chip cache.* IEEE journal of solid states, vol. SC-22, n° 5, Octobre 1987

[30] A. AGARWAL, R. SIMONI, J. HENNESSY, M. HOROWITZ - *An evaluation of directory schemes for cache coherence.* Stanford University, IEEE conference, Juin 1988

[31] J.E. SMITH, A.R. PLESZKUN, R.H. KATZ, J.R. GOODMAN - *Pipe : a high performance VLSI architecture.* IEEE international workshop on computer system organization, New Orleans, Mars 1983

[32] L. FOTI, D. ENGLISH, R.P. HOPKINS, D.J. KINNIMENT, P.C. TRELEAVEN, W.L. WANG - *Reduced-instruction set multi-microcomputer system*, Proceeding of the NCC, Juillet 1984

[33] R. FIRTH, J. GROSS - *Core set of assembly language instructions for MIPS-based microprocessors, version 3.2.* Carnegie-Mellon University, Technical Report, Octobre 1987

[34] G.J. LIPOVSKI - *The architecture of a simple, effective, control processor.* Microprocessing and microprogramming, Euromicro 76, North-Holland, 1976

[35] H.AZARIA, D. TABAK - *Design considerations of a single instruction microcomputer : a case study.* Microprocessing and microprogramming, vol. 11, n° 3, Mars 1983

[36] P. KOOPMAN - *The WISC concept.* US Navy, Byte, Avril 1987

[37] *250 MIPS pour un processeur 4 bits à jonctions Josephson.* Electronique Hebdo, n° 104 p. 19, Mars 1989

[38] R. FOS, K.J. KIEFER, R.F. VANGEN, S.P. WHALEN - *Reduced instruction set architecture for a GaAs microprocessor system.* Control Data Corporation, Revue IEEE computer, Octobre 1986

[39] *RISC : processeur 200 MIPS en Arséniure de Gallium.* Minis & micros, n° 318, Mars 1989

[40] T.L. RASSET, R.A. NIEDERLAND, J.H. LANE, W.A. GEIDEMAN - *A 32-bit RISC implented in enhancement-mode JFET GaAs.* Mc Donnell Douglas Company, Revue IEEE computer, Octobre 1986

[41] L. HARRINGTON et al. - *A GaAs 32-bit RISC microprocessor.* Mc Donnell Douglas Company - IEEE GaAs IC symposium, 1987

[42] W. HELBIG, V. MIKITINOVIC - *A DCFL E/D-MESFET GaAs experimental RISC machine.* RCA and Purdue University IEEE transactions on computers, vol. 38, n°2, Février 89

[43] V.M. MILUTINOVIC (Editeur) - *High-level language computer architecture.* Computer science Press, 1989

[44] J.C. HEUDIN, C. MÉTIVIER - *KIM200 : un processeur RISC 200 MIPS pour les applications de l'intelligence artificielle.* SODIMA S.A., 7e congrès AFCET/INRIA, Reconnaissance des formes et intelligence artificielle, Paris, Décembre 1989

[45] F. POIRIER - *Une mémoire cache multiniveau pour un processeur RISC en Arséniure de Gallium.* IEF/Paris XI,

Mémoire de DEA en électronique, Université Paris-Sud, Septembre 1989

[46] J.C. HEUDIN - *An hypercube architecture for concurrent object oriented computing.* SODIMA S.A., 1st european workshop on hypercube and distributed computers, IRISA/INRIA, Rennes, Octobre 1989

[47] J.A. FISHER - *Very large instruction word architectures and the ELI-S12.* Yale University, ACM conference, 1983

[48] J.A. FISHER - *Trace scheduling : a technique for global microcode compaction.* Yale University, IEEE transactions on computers, Juillet 1981

[49] J. HENNESSY - *Future Directions for RISC Processors.* Stanford University, MIPS Computer Systems, 1989

[50] T.L. JOHNSON - *The RISC/CISC melting pot : classic design methods converge in the MC68030 microprocessor.* Motorola Inc., Byte, Avril 1987

[51] G. EFLAND et al. - *The Symbolics Ivory Processor : a VLSI CPU for the genera symbolic processing environment.* Symbolics Inc., Janvier 1988

[52] R. FIRTH, TH. GROSS - *Core set assembly language instructions for MIPS-based microprocessors, version 3.2.* Carnegie-Mellon University, Technical Report, Octobre 1987

[53] M. MORRIS MANO - *Computer system architecture.* Prentice-Hall, 1982

[54] A.J. SMITH - *Cache memories.* Computing Survey, Septembre 1982

[55] D.V. KLEIN, R. FIRTH - *Final evaluation of MIPS M/500.* Carnegie-Mellon University, Technical Report, Novembre 1987

[56] SUN MICROSYSTEMS CORP. - *A RISC tutorial.* 1987

[57] CYPRESS SEMICONDUCTOR CORP. - *RISC 7C600, RISC familly users guide.* Juin 1988

[58] M. NAMOJOO et al. - *CMOS custom implementation of the SPARC architecture.* SUN Microsystems/Cypresse Semiconductor, IEEE COMPCON, San Francisco, Mars 1988

[59] L. QUACK, R. CHUCH - *CMOS gate array implementation of SPARC.* Fujitsu Microelectronics Inc., IEEE COMPCON, San Francisco - Mars 1988

[60] N. NAMJERO, A. AGRAWAL - *Sunrise : a high-performance 32-bit microprocessor.* SUN Microsystems, IEEE COMPCON, San Francisco, Mars 1988

[61] N. NAMJERO, A. AGRAWAL - *Implementing SPARC : a high-performance 32-bit RISC microprocessor.* SUN Microsystems, SUN technology, 1988

[62] R. GARNER et al. - *The scalable processor architecture (SPARC).* SUN Microsystems, IEEE COMPCON, San Francisco, Mars 1988

[63] S. KLEINIAN, D. WILLIAMS - *SUNOS on SPARC.* SUN Microsystems, IEEE COMPCON, San Francisco, Mars 1988

[64] A. AGRAWAL et. al. - *SPARC : an ASIC solution for high performance microprocessors.* SUN Microsystems, IEEE COMPCON, San Francisco, Mars 1988

[65] S. MUCHNIK et al. - *Optimizing compilers for the SPARC architecture : an overview.* SUN Microsystems, IEEE COMPCON, San Francisco, Mars 1988

[66] A. AGAWAL, et al. - *Design considerations for a bipolar implementation of SPARC.* SUN Microsystems, IEEE COMPCON, San Francisco, Mars 1988

[67] G. KANE - *MIPS R2000 RISC architecture.* Mips Computer Systems, Prentice Hall, Englewood cliffs (NJ), 1987

[68] M. GARROD - *R3000 et R3010 RISComponents.* Mips Computer Systems, March 1988

[69] ADVANCED MICRO DEVICES - *Am29000, 32-bit streamlined instruction processeur : User's manual.* Sunny Vale, Californie, 1988

[70] B. CASE - *32-bit microprocessor opens system bottlenecks.* AMD, Computer design, Avril 1987

[71] P. LORRAIN - *RISC, quel processeur et pourquoi faire ?* Minis & micros n° 321, Mai 1989

[72] MOTOROLA - *MC88100 & MC88200 : 32-bit third-generation RISC microprocessor.* Technical data, , 1988

[73] C. DOBBS, P. REED, TOMMY NG - *Supercomputing on chip.* Motorola/SCS, VLSI systems design - Mai 1988

[74] M. ACKERMAN, G. BAUM - *The Fairchild clipper : a microprocessor that attempts to balance the best of CISC and RISC.* Fairchild semiconductor, Byte, Avril 1987

[75] G. MEYERS, D. BUDDE - *The 80960 microprocessor architecture.* Intel Corp., Wiley, 1988

[76] S. MC GEADY - *A programmer's view of the 80960 architecture.* Intel Corp., 34th IEEE computer society international conference, COMPCON 89, San Francisco, Mars 1989

[77] *80960CA.* 32-bit high performance embedded processor data sheet, Intel Corp., 1989

[78] B. CASE - *Intel's i860 Sets New Performance Standard.* Microprocessor Report, Vol. 3, n°3, Mars 1989

[79] *VL86C010 : 32-bit RISC microprocessor.* VLSI Technology Inc., Janvier 1987

[80] VLSI TECHNOLOGY INC. - *VL86C010 RISC familly data manual.* 1987

[81] *The transputer data book : first edition 1989.* Inmos databook series, Novembre 1988

[82] A.D. BERENBAUM et al. - *Introduction to the CRISP Instruction Set Architecture.* IEEE COMPCON, 1987

[83] *IBM RISC workstation features 40-bit virtual addressing.* Computer Design, Février 1985

[84] J.S. BIRNBAUM, W.S. WORLEY - *Beyond RISC : high precision architecture.* Hewlett-Parckard journal, vol. 36, n°8, Août 1985

[85] R. RAGAN-KELLEY, R. CLARK - *Applying RISC theory to a large computer.* Computer design, Novembre 1983

[86] *Ridge 32 architecture : a RISC variation.* Proceedings of ICCD 1983, Octobre 1983

[87] Integrated Digital Products Inc. - *Whestone II computer technical summary*

[88] A.L. DOVIS, S.V. ROBISON - *The architecture of the FAIM-I symbolic multiprocessign system.* Schlumberger Inc., 9th Internation Joint Conférence on Artificial Intelligence, Août 1985

[89] L. MONIER, P. SIDHU - *The architecture of the dragon.* IEEE COMPCON, 1985

[90] CELERITY COMPUTING INC. - *ACCEL computer technical summurary*

[91] APPOLO INC. - *The PRISM Architecture : technical summary*

[92] W. WATSON, C. STEPHENS - *Novix : a radical approach to microprocessor design.* Computer Solutions of Byfleet, Electronics & Wireless World, 1988

[93] J.C. HEUDIN - *KIM : une nouvelle génération de processeurs RISC pour les applications de l'informatique.* SODIMA, Conférence sur les composants numériques et analogiques, Ministère de la Défense/Direction Scientifique/DRET, Arcueil, France, Avril 1989

[94] J.C. HEUDIN - *Architectures fondées sur la connaissance pour l'exécution et le contrôle de processus complexes.* Thèse de l'Univesité d'Orsay (Paris-Sud), Décembre 1988

[95] P.H. DUSSUD - *Lisp hardware architecture : the Explorer II and beyond* . Texas Instruments, First international workshop on

Lisp evolution and standardization, Paris, France, Février 1988

[96] B. EDWARDS, G. EFLAND, N. WESTE - *The symbolics I machine architecture : a symbolic processor architecture for VLSI implementation.* Symbolics Cambridge Research Center, VLSI System Group, 1987

[97] B. BURG, L. FOULLOY, J.C. HEUDIN, B. ZAVIDOVIQUE - *Behaviour rule systems for distributed process control.* ETCA, 2nd IEEE Conference on artificial intelligence applications, Miami-Beach (U.S.A.), Décembre 1985

[98] B. ZAVIDOVIQUE - *Rapport d'activités du laboratoire Système de Perception 1987-1988.* Etablissement Technique Central de l'Armement, 1988

[99] J.C. HEUDIN, B. BURG, B. ZAVIDOVIQUE -*Vers un système temps réel fondé sur la connaissance. Une approche : un système d'exploitation à règles pour le contrôle de processus complexes.* ETCA, 2e Colloque international d'intelligence artificielle, IIRIAM/ADI/CEA/AMEDIA, Marseille, France, Décembre 1986

[100] J.C. HEUDIN, C. MÉTIVIER, T. PORCHER, D. DEMIGNY, T. MAURIN, F. DEVOS - *Les implications de temps réel dans l'architecture des machines symboliques - Exemple : les commutations de tâches dans KIM.* SODIMA/IEF - Paris XI, 6ème Conférence "Reconnaissance des formes et intelligence artificielle", AFCET/INRIA, Antibes, France, Novembre 1987

[101] D. DEMIGNY, L. KESSAL, C. KOESTER, J.C. HEUDIN, T. MAURIN, F. DEVOS - *Reducing memory accesses on PEARLS : a symbolic processor.* IEF/Paris Sud, Microsystems BRNO 87, Septembre 1987

[102] J.C. HEUDIN - *Le projet KIM.* SODIMA S.A., Conférence sur les machines informatiques symboliques et neuromimétiques, Ministère de la Défense, Direction Scientifique DRET, Arcueil, France, Décembre 1987

[103] J.C. HEUDIN, C. MÉTIVIER, P. KAJFASZ, B. ZAVIDOVIQUE, F. DEVOS - *A compact symbolic processor for artificial intelligence applications.* SODIMA S.A./ETCA/IEF, 2nd IEEE conference on computers and application, Peking, Chine, Juin 1987

[104] J.C. HEUDIN, C. MÉTIVIER, D. DEMIGNY, T. MAURIN, B. ZAVIDOVIQUE, F. DEVOS - *A comparative study : microprogrammed versus RISC for symbolic processing.* SODIMA S.A./ETCA/IEF, SPIE 5th conference on applications of artificial intelligence proceeding, Vol. , Orlando, U.S.A., Mai 1987

[105] J.C. HEUDIN, JP. COURRIER, C. MÉTIVIER - *Toward embedded controllers for real-time applications of artificial intelligence.* SODIMA S.A., 2nd ACM international conference on industrial & engineering applications of artificial intelligence & expert systems, University Tennessee Space Institute/ACM/AAAI/ IEEE SIGART, Tullahoma, Tennessee, U.S.A., Juin 1989

[106] MC CARTHY - *History of Lisp.* Stanford University, Academic Press, 1981

[107] P.H. WINSTON, B.K.P. HORN - *Lisp* . Addison-Wesley, 1984

[108] P. COINTE - *Une introduction à la programmation par objets.* LITP, Janvier 1987

[109] T.H. MC ENTEE - *Overview of garbage collection in symbolic computing.* Texas Instruments, T.I. Engineering Journal, Janvier/Février 1986

[110] D.G. BOBROW, D.W. CLARK - *Compact encoding of list structure.* Xerox PARC Technical Report, Juin 1979

[111] D. CORTINOVIS - *Conception et simulation d'un langage objet pour une machine hypercube : SKIM.* Mémoire de DEA d'Intelligence Artificielle, Université Paris VI, Septembre 1989

[112] T. PORCHER - *Etude et réalisation d'un environnement Lisp sur KIM : ISALISP.* Mémoire de DEA en intelligence artificielle, Université Paris VIII, Octobre 1987

[113] R.P. GABRIEL - *Performance and evaluation of Lisp system.* The MIT Press, 1985

[114] J.C. HEUDIN - *Adjonction de primitives de traitement symbolique dans un langage procédural : l'exemple de SPACE.* SODIMA S.A., Convention IA 90, Paris, Janvier 1990

Index des illustrations

La Société SODIMA vous propose, à un prix attractif, le complément indispensable et idéal de cet ouvrage sur les architectures RISC.

- ☞ Simulateur logiciel du processeur symbolique RISC KIM™20 avec assembleur, éditeur de lien, disponible pour PC ou stations de travail SUN™Microsystems.

- ☞ Carte processeur KIM™20 10 MIPS pour PC ou compatibles.

- ☞ Carte processeur KIM™20 10 MIPS au standard VMEbus.

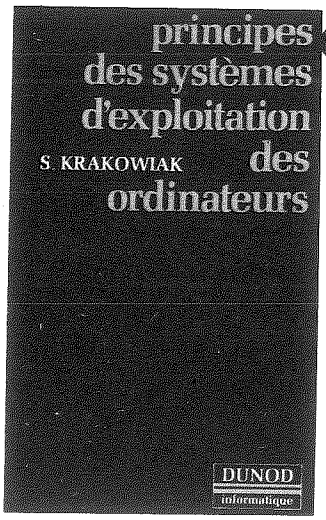
- ☞ Environnement Le_Lisp™ de l'INRIA pour KIM™20.

- ☞ Exécutif temps réel à base de règles KOS™ pour KIM™20.

Pour tous renseignements, contacter :

Monsieur Jean-Claude HEUDIN
SODIMA S.A.
Zone d'Activités de Courtaboeuf - 18, avenue du Québec
Boîte Postale EVOLIC 701 - 91961 LES ULIS CEDEX FRANCE
Téléphone : (1) 69 07 32 18
Télécopie : (1) 69 07 66 58

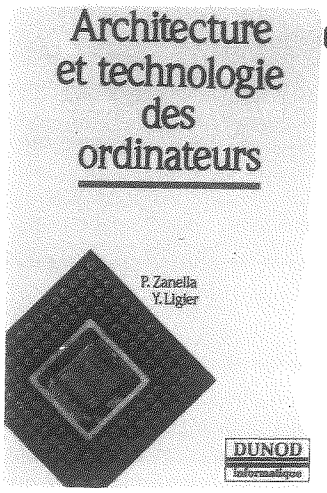
Dunod, le savoir informatique



Principes des systèmes d'exploitation des ordinateurs.

Fonction d'un système d'exploitation - Évolution historique - Structuration des systèmes logiciels - Mécanismes d'exécution et de communication - Gestion des activités parallèles - Réalisation des mécanismes de synchronisation - Désignation, conservation et liaison des objets - Systèmes de gestion de fichiers - Allocation de ressources - Allocation de mémoire - Structure d'un système multiprogrammé - Introduction aux systèmes informatiques répartis.

Une étude exhaustive des principes et fonctions des systèmes d'exploitation des ordinateurs, accompagnée de nombreux exercices. L'auteur est Professeur à l'Université Scientifique et Médicale de Grenoble.



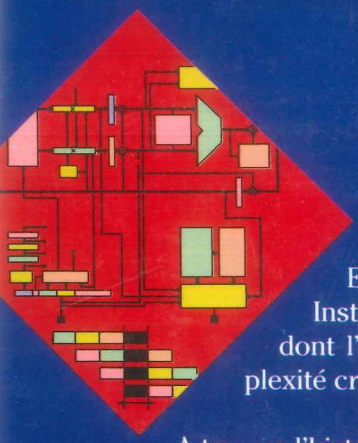
Architecture et technologie des ordinateurs.

Histoire de l'ordinateur - Représentation des informations - Circuits logiques - Technologie des composants électroniques - Mémoires - Unité centrale de traitement - Super-ordinateurs et microprocesseurs - Entrées et sorties - Téléinformatique - Codes détecteurs et correcteurs d'erreurs - Langage d'assemblage et programmation - Langages évolués - Structures de données - Modes et systèmes d'exploitation.

Une véritable petite encyclopédie qui présente tous les aspects de l'informatique et permet d'en suivre l'évolution rapide. P. Zanella travaille au CERN de Genève et Y. Ligier est chercheur au centre IBM de Yorktown (USA).

Pour découvrir nos autres titres, demandez le catalogue «Informatique» à votre libraire ou, par correspondance, à la librairie Dunod, 30 rue Saint-Sulpice 75006 PARIS.

Imprimerie GAUTHIER-VILLARS, Paris
Dépôt légal, Imprimeur, n° 3314
Dépôt légal : avril 1990 *Imprimé en France*



LES ARCHITECTURES RISC

En prônant la simplicité, le concept RISC (Reduced Instruction Set Computer) révolutionne l'informatique, dont l'évolution s'accompagnait jusqu'alors par une complexité croissante des systèmes.

A travers l'historique des ordinateurs à jeu d'instructions réduit, cet ouvrage explique les fondements théoriques de cette nouvelle architecture. Décrivant les principaux processeurs commercialisés, il dresse un tableau complet de leurs caractéristiques, performances et domaines d'application. L'étude détaillée de l'un d'entre eux permet d'illustrer les liens étroits entre le processeur et le logiciel qu'il doit exécuter.

C'est autour de cette architecture que tous les constructeurs développent les stations de travail des prochaines années. Ce livre, qui s'adresse aux professionnels désireux de comprendre l'évolution technologique actuelle, intéressera également les étudiants en informatique et électronique de l'université et des écoles d'ingénieurs.

9 782040 196417

ISBN : 2-04-019641-2