Architecture des Ordinateurs
Deuxième partie

Cécile Germain Daniel Etiemble

Licence d'Informatique - IUP Miage - FIIFO

Table des matières

5 Microarchitecture du processeur **3**
5.1 Les composants matériels élémentaires 4
5.1.1 Combinatoires 5
5.1.2 Horloge 7
5.1.3 Mémorisation 7
5.1.4 Bus internes du processeur 10
5.2 Chemin de données 11
5.2.1 L'architecture DE99 11
5.2.2 L'exécution des instructions 12
5.2.3 Une architecture 3 bus 16
5.2.4 Performance 17
5.3 Contrôle 17
5.3.1 Contrôle câblé 19
5.3.2 Performance 21
5.3.3 Microprogrammation 22
5.4 Conclusion 23

6 Microarchitecture pipelinée **25**
6.1 Principe du pipeline 26
6.2 Le pipeline classique pour l'exécution des instructions 27
6.2.1 Tâches élémentaires 27
6.2.2 Pipeline pour les instructions UAL 28
6.2.3 Pipeline et accès mémoire 30
6.3 Les contraintes du pipeline 31
6.3.1 Ressources matérielles 31
6.3.2 Format d'instruction fixe et simple - Cache d'instruction 32
6.3.3 Architecture chargement-rangement 33
6.4 CISC et RISC 33

6.5 Les limitations du pipeline. 35
6.5.1 Aléas de données. 35
6.6 Aléas de contrôle 43
6.6.1 Les différents types d'aléas de contrôle 45
6.6.2 Traitement des aléas de contrôle 46
6.7 Micro-architecture pipelinée 48
6.8 Instructions flottantes 49
6.9 Architectures superscalaires 52
6.9.1 Dépendances de ressources 53
6.9.2 Dépendances de données 54
6.9.3 Aléas de contrôle 56
6.10 Conclusion 59

7 Hiérarchie mémoire : la mémoire centrale 61
7.1 Introduction 61
7.1.1 Motivation 61
7.1.2 Le principe de localité 62
7.1.3 Un exemple: le produit matrice-vecteur 64
7.2 Fonctionnement du cache 64
7.2.1 Succès et échec cache 65
7.2.2 Traitement des échecs en lecture 66
7.2.3 Les écritures 67
7.2.4 En résumé 68
7.3 Correspondance entre le cache et la mémoire principale . . . 69
7.3.1 Correspondance directe 69
7.3.2 Correspondance associative 71
7.3.3 Correspondance associative par ensemble 72
7.3.4 Algorithmes de remplacement 74
7.4 Performances 74
7.5 Performances : diminuer le taux d'échec 75
7.5.1 Origines des échecs 75
7.5.2 Organisation du cache 76
7.5.3 Optimisations logicielles 78
7.6 Performance : le temps de rechargement 80
7.6.1 Les circuits DRAM 80
7.6.2 La liaison mémoire principale - cache 84
7.7 Conclusion 89

8 Mémoire virtuelle **91**
8.1 Introduction 91
8.1.1 Organisation de principe 92
8.2 Mécanismes de traduction 94
8.2.1 Table des pages 94
8.2.2 TLB 96
8.2.3 Mémoire virtuelle et caches 98
8.3 Mécanismes de protection 99
8.3.1 Protection 101
8.3.2 Partage 102
8.4 Implémentations de la table des pages 103
8.4.1 Organisation hiérarchique 103
8.4.2 Table des pages inverse 106
8.5 Conclusion 107

9 Bus et Entrées-Sorties **109**
9.1 Introduction 109
9.2 Les disques 110
9.2.1 Organisation 110
9.2.2 Accès aux disques 111
9.3 Processeurs d'E/S 112
9.3.1 Un exemple très simple 112
9.3.2 Processeurs d'E/S 113
9.4 Les bus 114
9.4.1 Fonctionnalités 114
9.4.2 Arbitrage 114
9.4.3 Transfert de données 116
9.4.4 Bus asynchrone 117
9.4.5 Le bus PCI 119
9.4.6 Amélioration des performances 120
9.5 Traitement des Entrées/Sorties 121
9.5.1 Typologie 121
9.5.2 Mécanismes d'interruption 122
9.6 Interruptions et exceptions 123
9.6.1 Les situations exceptionnelles 124
9.6.2 Gestion des situations exceptionnelles 127

Chapitre 5

Microarchitecture du processeur

Les chapitres précédents ont présenté les éléments de l'architecture logicielle d'un processeur, qui est définie par son jeu d'instructions. Ce chapitre étudie la réalisation d'une architecture logicielle.

Par exemple, quels dispositifs faut-il mettre en oeuvre pour additionner deux registres dans un troisième, ce qui réalisera l'instruction ADD Rd,Ra, Rb ? Il faut disposer d'une part d'organes de mémorisation, qui réaliseront les registres Rd, Ra, Rb, d'un circuit combinatoire capable de faire des additions, mais aussi des fonctions de sélection des registres (on ne veut pas additionner Ra et Rd dans Rb !), de lecture de l'instruction dans la mémoire, et d'incrémentation du compteur de programme. Bien sûr, cet exemple est très élémentaire. Il l'est même trop, car la réalisation d'une architecture logicielle est un problème d'optimisation globale : maximiser le débit d'instructions IPC.

La réalisation possède une caractéristique fondamentale : contrairement aux niveaux supérieurs d'un système informatique, elle n'est pas programmable (fig. 5.1). La réalisation des actions impliquées par chaque instruction est fixée définivement, par le matériel (*hardwired*). C'est une différence essentielle. Par exemple, pour réaliser l'instruction C a = b+c, avec le jeu d'instructions du chapitre précédent, le compilateur dispose d'une certaine liberté : mode d'adressage, ordre des chargements, choix des registres. A l'inverse, une réalisation donnée d'une architecture logicielle doit définir l'exécution des instructions à partir de composants matériels. Une telle réalisation est appelée une *microarchitecture*.

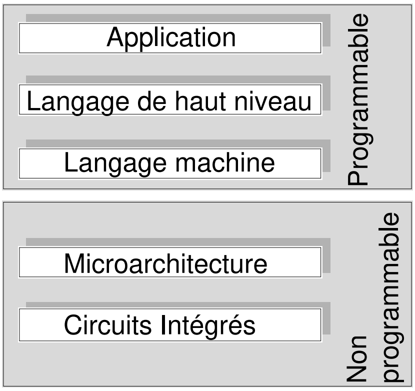


Figure 5.1: Les niveaux d'un système informatique

Un jeu d'instruction peut être réalisé par plusieurs microarchitectures : ce chapitre et le suivant en présenteront deux exemples simples. Le travail de l'architecte de microprocesseurs est précisément de définir une microarchitecture efficace, en fonction des conditions données de la technologie.

La première partie de ce chapitre présente les composants matériels élémentaires. La seconde et la troisième montrent comment assembler ces composants pour implémenter une architecture logicielle exemple, DE99, par une micro-architecture très simple, qu'on appellera DE99/NP. Le chapitre suivant étudiera une réalisation alternative qui augmente l'IPC.

5.1 Les composants matériels élémentaires

Il s'agit ici seulement de définir *fonctionnellement* (spécification) les composants matériels qui seront utilisés comme blocs de base pour la conception d'une micro-architecture : en termes de CAO de circuits, ce seraient les éléments de la bibliothèque. Leur réalisation relève de la conception logique et micro-électronique. La présentation qui suit est donc purement taxonomique et très élémentaire. On trouvera dans [6] une présentation des aspects logiques, dans [7] une étude détaillée de la conception logique des circuits flottants, et dans [5] une introduction à la conception microelectronique de ces circuits.

Il faut distinguer quatre grandes classes : combinatoire, horloges, mémorisation, bus.

5.1.1 Combinatoires

Les circuits combinatoires réalisent des fonctions booléennes. Ils transforment l'information, et sont dépourvus de toute capacité de mémorisation. On distingue : les portes logiques ; les PLA, qui réalisent des fonctions booléennes quelconques ; des circuits spécialisés (additionneurs, UAL, unités de calcul flottant), qui sont optimisés pour la vitesse.

Portes logiques

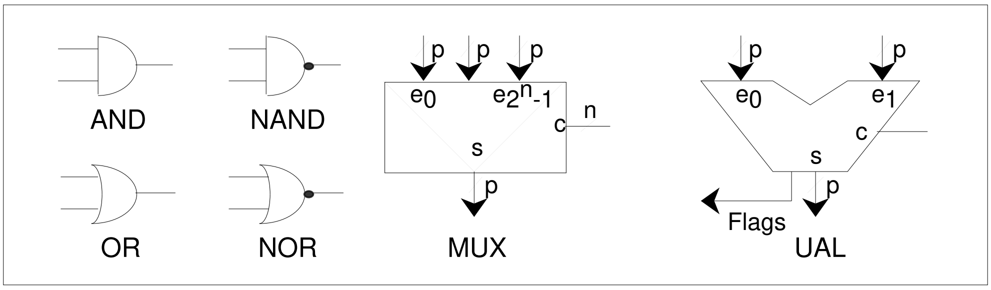


Figure 5.2: La représentation conventionnelle de quelques portes et circuits logiques

Elles réalisent les fonctions logiques élémentaires : NOT, AND, OR, XOR etc. La fig. 5.2 présente leur représentation conventionnelle.

PLA

Avec n variables booléennes $e_0, e_1, \ldots e_n$, on peut définir 2^{2^n} fonctions booléennes distinctes (table. 5.1).

Un PLA (Programmable Logic Array) réalise un ensemble de fonctions logiques arbitrairement définies, de ses entrées.

En fait, le terme de PLA se réfère à une technologie logique particulière de réalisation. Toute fonction booléenne peut s'écrire sous une forme particulière, appelée forme disjonctive normale, dérivée de sa définition en extension (table de vérité) comme le montre l'exemple de la fig. 5.3. Un PLA réalise les monômes conjonctifs et les disjonctions sous forme de matrices de points de croisement.

On peut en particulier réaliser sous forme de PLA la fonction de décodage : pour n entrées, on a 2^n sorties. La sortie i est à 1 si et seulement si les n

e	f0	f1	f2	f3
0	0	0	1	1
1	0	1	0	1

Table 5.1: Les 4 fonctions booléennes d'une variable

e_1	e_0	f
0	0	1
0	1	0
1	0	1
1	1	1

$$f = \overline{e_1}.\overline{e_0} + e_1.\overline{e_0} + e_1.e_0$$

Figure 5.3: La forme disjonctive normale d'une fonction logique

entrées codent le nombre i.

Multiplexeur

Un multiplexeur (fig. 5.2) possède 2^n entrées de données, une entrée de commande sur n bits, et une sortie. La sortie est égale à l'entrée e_i si l'entrée de commandes code le nombre i.

Unité Arithmétique et Logique

L'UAL est le principal organe de traitement de l'information dans un processeur. Elle a deux entrées de données, sur n bits, une sortie correspondant à l'opération effectuée, sur n bits, et éventuellement une sortie correspondant aux drapeaux (flags) positionnés par l'opération. Elle possède aussi une entrée de commande, qui sélectionne l'opération à effectuer. Elle effectue les opérations arithmétiques et logiques, les décalages et rotations. Notons que, du point de vue temporel, ce sont les opérations arithmétiques qui déterminent le temps de traversée de l'UAL, à cause des temps de propagation de la retenue. On note t_{UAL} ce temps de propagation.

Dans la suite, on utilisera également deux circuits plus particuliers : un incrémenteur et un extenseur de signe. L'incrémenteur ajoute une quantité fixe (câblée) à l'entrée, par exemple $s = e + 4$. L'extenseur de signe transforme une entrée sur p bits en une sortie sur n bits par extension du bit de poids fort : $s_{p+1}, \ldots, s_{n-1} = e_p$.

Cette revue n'est en aucune facon exhaustive : manquent en particulier les unités de calcul flottant et les unités de multiplication et division entières.

Caractéristiques temporelles

Chaque unité combinatoire est caractérisée par un *temps de propagation* : délai séparant le changement d'état de l'entrée du changement corrélatif des sorties.

5.1.2 Horloge

Une horloge est un signal périodique, symétrique ou non. Dans la suite, on le suppose toujours symétrique. Il crée une référence de temps commune pour l'ensemble du processeur.

La période de l'horloge, T_c, est le *temps de cycle* du processeur. Bien sûr, l'architecte souhaite diminuer autant que possible ce temps ; il convient donc d'étudier les contraintes qui déterminent T_c. Comme on le verra dans la suite, la fréquence de l'horloge est déterminée par les délais de propagation dans et entre les divers organes du processeur.

5.1.3 Mémorisation

Les circuits combinatoires décrits ci-dessus ne retiennent pas l'information : tout changement des entrées est répercuté, après le délai de propagation, sur les sorties. L'architecture logicielle suppose au contraire que l'information est retenue indéfiniment (registres, mémoire). Les circuits du même nom remplissent cette fonction.

Registres

On ne considère ici que des registres opaques. La sortie s délivre l'état que présentait l'entrée e lorsque le signal d'ecriture W a présenté une transition (mémorisation sur front), par exemple montante. La figure 5.4 présente le fonctionnement temporel idéal d'un registre opaque : la sortie ne change pas, malgré la variation de l'entrée, jusqu'à la transition de la commande, d'où le nom de registre *opaque*.

En réalité, s ne reflète pas instantanément l'état des entrées : la transition de l'entrée doit précéder celle de la commande d'un temps minimum (temps

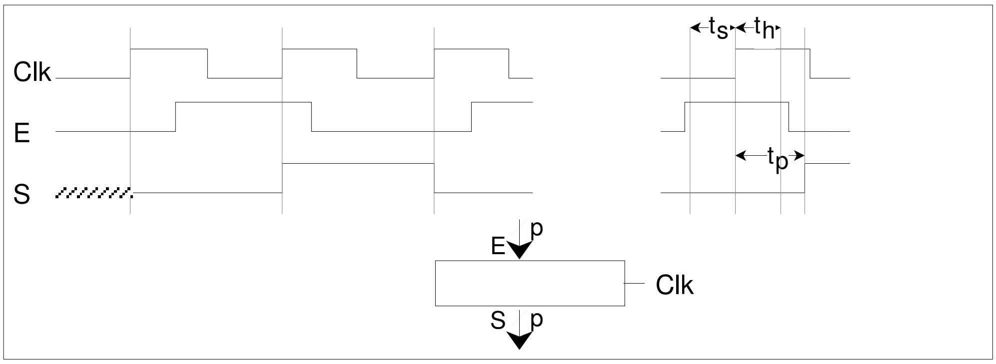


Figure 5.4: Fonctionnement temporel d'un registre opaque

de setup, t_s), et être maintenue après (temps de hold, t_h), et la propagation de l'entrée vers la sortie consomme un temps non nul t_r. On note T_{reg} le délai entre la transition sur l'entrée et celle sur la sortie : $T_{\text{reg}} = t_s + t_r$.

Dans la suite, on supposera que le signal d'écriture W est obtenu par un ET de l'horloge et d'un signal WR positionné à 1 lorsqu'on veut écrire dans le registre. Tous les registres ne peuvent donc être écrits qu'en fin de cycle, sur la transition montante du signal de l'horloge. Ils enregistrent alors l'état de leur entrée.

Considérons un transfert registre à registre à travers un circuit combinatoire (fig. 5.5) de temps de propagation T_p. Si le registre d'entrée est écrit sur la transition i de l'horloge, et que le résultat de l'opération combinatoire doit être écrit dans le registre de sortie sur la transition $i+1$ de l'horloge, la période d'horloge doit être supérieure ou égale au temps de propagation du circuit combinatoire augmentée du temps nécessaire au bon fonctionnement des registres. On voit donc apparaître la contrainte suivante sur le temps de cycle :

$$T_c \geq T_{\text{reg}} + T_p$$

Banc de registres

Un banc de registres est un ensemble de registres $R_0, \ldots, R_{n-1}$ qui partagent leurs entrées et leurs sorties. Il est caractérisé par le nombre de lectures et d'écritures simultanées : par exemple, un banc à 2 lectures et 1 écriture

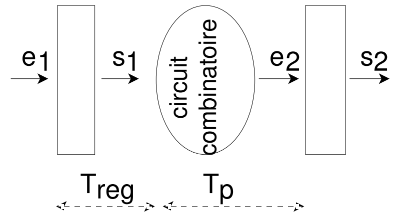


Figure 5.5: Contrainte temporelle sur un transfert registre-registre

possède deux sorties et une entrée de données. L'entrée de commande C sélectionne les registres lus et le registre modifié parmi $R_0, \ldots, R_{n-1}$.

L'intérêt d'un banc de registres, par rapport à n registres individuels, est un moindre encombrement. En revanche, n registres individuels peuvent tous être lus et écrits simultanément. En outre, le temps de traversée d'un banc de registres T_{Breg} est supérieur à celui d'un registre individuel.

Mémoire

La technologie des circuits mémoires actuels est complexe et variée. Ces circuits sont utilisés à l'extérieur du processeur, pour réaliser les caches de second niveau, et la mémoire principale. On considère ici une abstraction simplifiée, qui correspond à peu près à une mémoire statique (SRAM) utilisée pour réaliser un cache de premier niveau.

Une mémoire ne possède qu'un seul port de données, soit en entrée, soit en sortie ; elle possède également une entrée d'adresse, qui sélectionne le mot lu ou écrit, et une entrée de commande, qui déclenche l'écriture du mot adressé. Le *temps d'accès* T_a de la mémoire est le temps qui sépare l'établissement de l'adresse de la disponibilité de la donnée.

La fig. 5.6 illustre les contraintes temporelles liées au fonctionnement de la mémoire, lors d'une lecture. Le registre R_x est connecté à l'entrée d'adresse, le registre R_y reçoit le mot lu. Si l'adresse est écrite dans R_x sur la transition i de l'horloge, la donnée lue doit être disponible avant la transition $i+1$ de l'horloge pour permettre la lecture d'un mot à chaque cycle. On a donc :

$$T_c \geq T_a + T_{\mathrm{reg}}$$

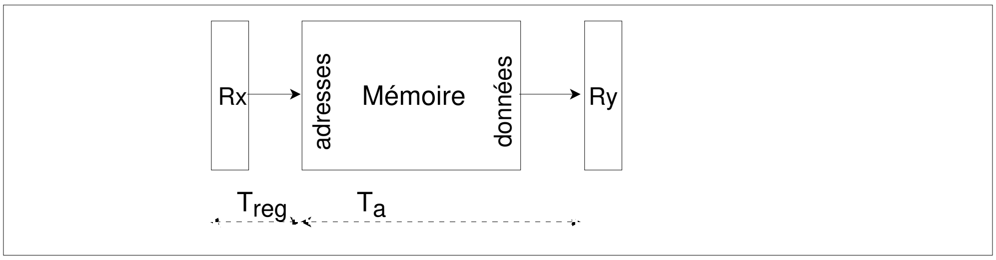
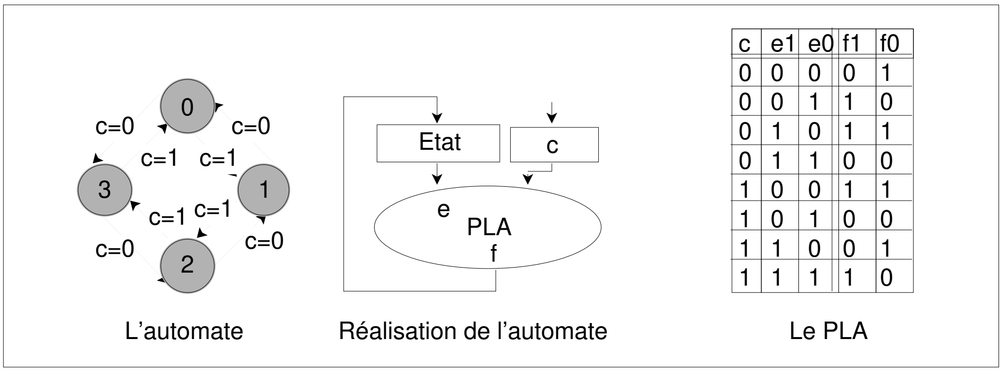


Figure 5.6: Contraintes temporelles liées à la mémoire



c	e1	e0	f1	f0
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	0

Figure 5.7: Un exemple de réalisation d'un automate.

Automates

La combinaison de registres opaques et de circuits combinatoires permet en particulier de réaliser en matériel des automates d'états finis. La fig. 5.7 montre comment réaliser un automate compteur ou décompteur, entre 0 et 3. L'entrée e, qui provient d'une commande externe, contrôle le mode (compteur 0, décompteur 1) ; l'état est codé sur 2 bits et échantilloné dans un registre ; le PLA réalise la fonction $f \equiv e + 1 \bmod 4$ en mode compteur, $f \equiv e - 1 \bmod 4$ en mode décompteur.

5.1.4 Bus internes du processeur

Les bus internes ne réalisent ni fonction combinatoire, ni mémorisation. Ce sont des liens électriques passifs. Leur importance provient de ce qu'il ne serait pas économique, en terme de surface, d'établir un chemin entre chaque paire de registres (ou autre organe de mémorisation) d'un processeur. Il

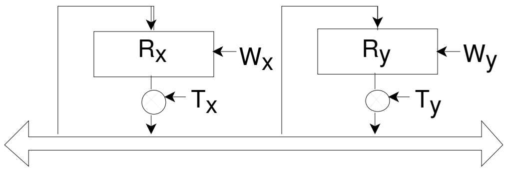


Figure 5.8: Transfert registre-registre à travers un bus partagé

existe donc un petit nombre de bus, auxquels sont connectés un ensemble de registres (fig. 5.8). Lorsque les registres sont connectés en entrée, il n'y a pas de problème particulier. En revanche, un seul registre peut délivrer sa sortie sur un bus donné, sous peine de court-circuit. Ceci demande l'intervention d'une *barrière de bus*, dont la commande devra être activée à propos.

Le transfert d'un registre R_x vers un registre R_y à travers un bus met donc en jeu deux commandes : l'ouverture de la barrière de bus T_x et l'écriture du registre destination W_y.

5.2 Chemin de données

La mise en oeuvre des composants précédents dans un processeur comporte deux aspects, qui correspondent à deux sous-ensembles du processeur : le chemin de données d'une part, qui contient les organes de mémorisation et l'UAL ; d'autre part, la partie contrôle, qui génère les commandes de ces circuits. Dans cette partie, nous supposerons que les commandes peuvent être générées de façon appropriée. La partie suivante étudiera la partie contrôle. Pour étudier cette mise en œuvre concrètement, on définit d'abord une architecture logicielle exemple, l'architecture DE99

5.2.1 L'architecture DE99

DE99 est une architecture 32 bits. La table 5.2 décrit le jeu d'instructions, qui est réduit à quelques instructions caractéristiques. La fig. 5.3 décrit les formats des instructions et le codage des instructions. La mémoire est adressable par octets, avec des adresses 32 bits.

L'exécution d'une instruction se décompose en deux phases : la lecture de l'instruction, qui est la même pour toutes les instructions, et l'exécution proprement dite, qui est spécifique. Nous étudierons d'abord l'exécution

Mnémonique	Format	Description
ADD, SUB	Reg-Reg	Rd ← Ra OP Rb ; RCC modifié
	Reg-Imm	Rd ← Ra OP ES(Imm) ; RCC modifié
AND, OR,	Reg-Reg	Rd ← Ra OP Rb
	Reg-Imm	Rd ← Ra OP ES(Imm)
CMP	Reg-Reg	Ra - Rb ; RCC modifié
LD	Reg-Reg	Rd ← Mem [Ra + Rb]
	Reg-Imm	Rd ← Mem [Ra + ES(Imm)]
ST	Reg-Reg	Mem [Ra + Rb] ← Rd
	Reg-Imm	Mem [Ra + ES(Imm)] ← Rd
Bcond	Bcht	si cond vraie PC ← PC + ES(imm)

Table 5.2: Les instructions de DE99

de chaque instruction, indépendamment, puis le moyens les plus efficaces de combiner ces chemins de données individuels, avec la lecture de l'instruction.

Toutes les instructions qui utilisent les registres généraux R0, ..., R31 ont un format à au plus trois opérandes registres, dont deux en lecture et un en écriture. Un banc de registres à deux lectures et une écriture est donc bien adapté pour implanter ces registres généraux.

5.2.2 L'exécution des instructions

Instructions Arithmétiques et Logiques, format Reg-Reg

Ces instructions ont pour schéma commun Rd ← Ra OP Rb. Il suffit donc de connecter les sorties du banc de registre sur l'entrée d'une UAL, et la sortie de l'UAL sur l'entrée du banc de registres (fig 5.9).

Instructions Arithmétiques et Logiques, format Reg-Imm

Ces instructions ont pour schéma commun Rd ← Ra OP ES(Imm_{16}). Après la lecture de l'instruction, celle ci-est contenue dans un registre particulier, le registre instruction RI, qui n'est pas visible à l'utilisateur : il n'existe pas d'instruction qui le lise ou qui l'écrive explicitement. L'immédiat est donc contenu dans les 16 bits de poids faible du registre RI. Il suffit donc de connecter $RI_{0:15}$ à un extenseur de signe, et la sortie de cet extenseur à l'une des entrées de l'UAL (fig 5.9).

Reg-Reg

5	1	5	5	5	11
codop	0	Rd	Ra	Rb	xxxxxxxxxxx

Reg-Imm

5	1	5	5	16
codop	1	Rd	Ra	Imm

Bcht

5	4	23
codop	cond	Imm

Table 5.3: Formats de l'architecture DE99

Instructions	Codop
UAL	0xxxx
LD	10xx0
ST	10xx1
Bcond	110xx

Table 5.4: Les codes opérations

Les contraintes temporelles découlant de l'exécution des instructions arithmétiques et logiques sont donc du type transfert registre-registre à travers un circuit combinatoire :

$$T_c \geq T_{\mathrm{Breg}} + T_{\mathrm{UAL}}.$$

Chargement

En format Reg-Reg, ces instructions ont pour schéma commun Rd $\leftarrow$ Mem[Ra + Rb].

La solution la plus simple est decrite fig. 5.10 (a). L'ensemble du chemin devant être parcouru en un cycle, le temps de traversée de l' UAL s'ajoute au temps d'accès mémoire T_a pour contraindre le temps de cycle :

$$T_c \geq T_{\mathrm{Breg}} + T_{\mathrm{UAL}} + T_a.$$

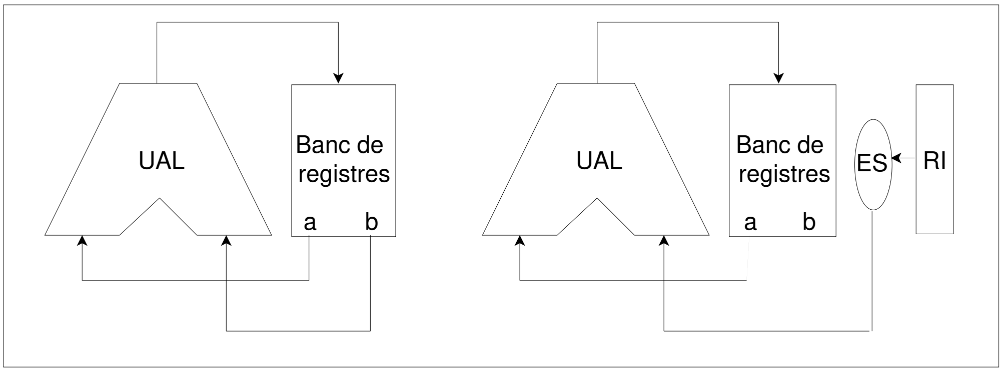


Figure 5.9: Chemins de données pour l'exécution des instructions arithmétiques et logiques

Il faut bien comprendre que le temps de cycle est déterminé de façon unique pour toutes les instructions. La contrainte précédente va donc approximativement doubler le temps d'exécution de toutes les instructions. Une solution plus pertinente consiste à effectuer le chargement en 2 cycles : les autres instructions, par exemple arithmétiques et logiques, ne seront pas pénalisées. La fig. 5.10 (b) montre comment implanter cette solution, en interposant un registre RAdM entre l'UAL et la mémoire, pour recueillir le résultat du calcul d'adresse. RAdM est un registre interne du processeur : il ne fait pas partie de l'architecture logicielle, c'est un choix d'implémentation. On a alors

$$T_c \geq \mathrm{Max}(T_{\mathrm{Breg}} + T_{\mathrm{UAL}}, T_{\mathrm{reg}} + T_a).$$

Le format Reg-Imm se traite de façon analogue, en substituant l'extension de signe de $\mathrm{RI}_{0:15}$ à Rb.

Rangement

En format Reg-Reg, ces instructions ont pour schéma commun Mem[Ra + Rb] $\leftarrow$ Rd.

Cette instruction ne peut pas être réalisée en 1 cycle, avec le matériel retenu jusqu'ici : elle imposerait la lecture dans le même cycle des trois registres Ra, Rb et Rd. Il est parfaitement possible de disposer d'un banc de registres à 3 lectures simultanées, mais avec un coût supplémentaire en complexité matérielle. En outre, le temps de cycle serait pénalisé comme

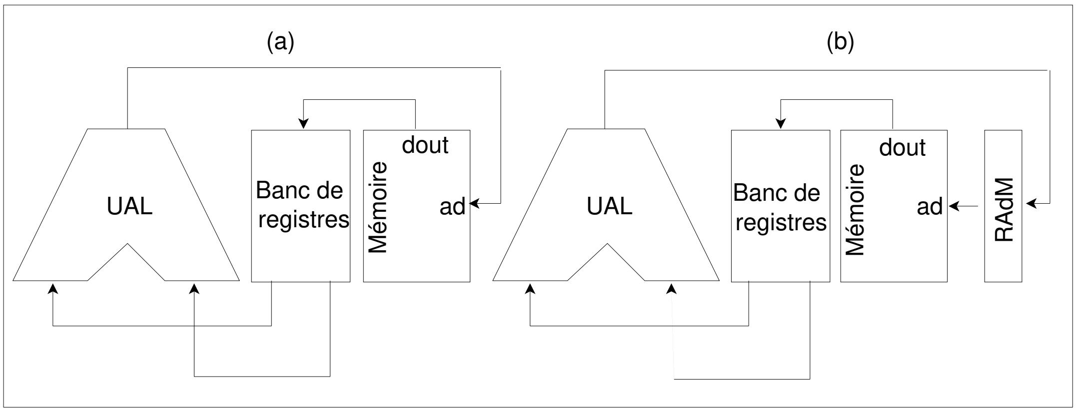


Figure 5.10: Chemin de données pour le chargement en format Reg-Reg. Sans (a) ou avec (b) registre d'adresse

dans le chargement, solution (a). La solution naturelle consiste à réaliser le rangement en 2 cycles, avec l'architecture de la fig. 5.11.

Le format Reg-Imm se traite de façon analogue, en substituant l'extension de signe de $RI_{0:15}$ à Rb.

Branchements

Le registre compteur de programme de l'architecture logicielle est réalisé par un registre opaque PC. Les branchements simples correspondent au transfert $PC \leftarrow PC + ES(Imm_{24})$. L'UAL peut être utilisée pour réaliser le calcul

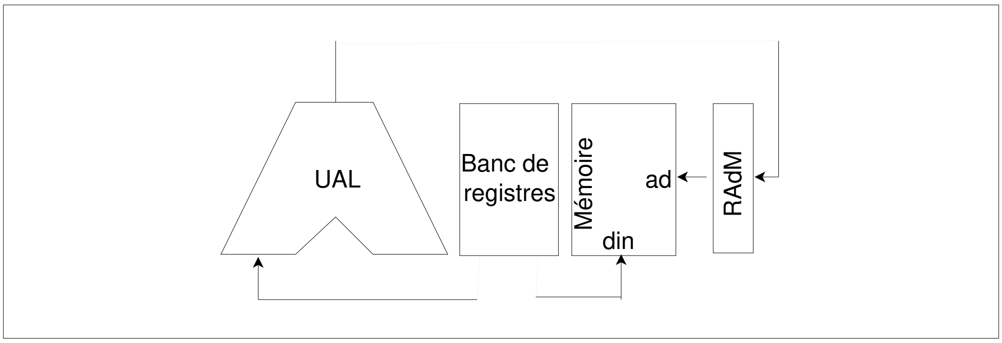


Figure 5.11: Chemin de données pour le rangement en format Reg-Reg.

d'adresse (fig. 5.12).

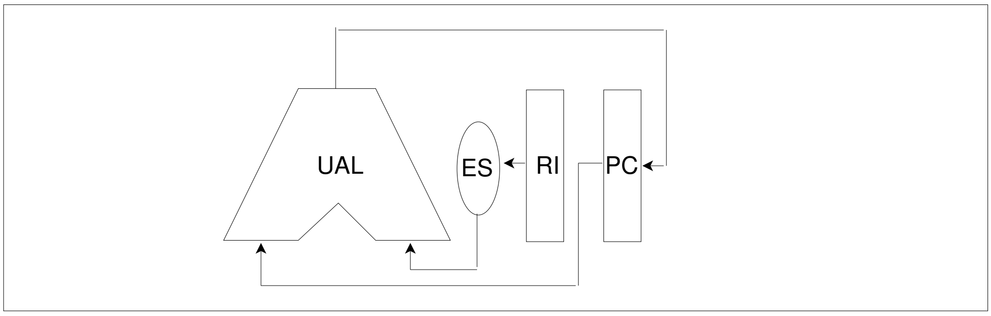


Figure 5.12: Chemin de données pour les branchements

5.2.3 Une architecture 3 bus

L'assemblage des constructions précédentes doit s'effectuer en minimisant le nombre de liaisons dédiées. Les opérations arithmétiques et logiques utilisent trois opérande (deux sources, un résultat), l'architecture doit donc au minimum comprendre trois bus.

La fig. 5.13 présente une chemin de données global organisé autour d'une architecture à 3 bus. Les bus A et B supportent les opérandes sources, le bus R l'opérande résultat dans les instructions arithmétiques et logiques. La seule réelle nouveauté de cette architecture est le traitement de la lecture de l'instruction, avec en particulier un incrémenteur associé au registre PC.

La phase de lecture de l'instruction comporte deux actions : l'acquisition de l'instruction (RI $\leftarrow$ Mem[PC]) et l'incrémentation du compteur de programme (PC $\leftarrow$ PC + 4), pour l'exécution en séquence. Avec une architecture à 3 bus, il n'est pas possible d'utiliser l'UAL pour incrémenter PC et lire simultanément l'instruction en mémoire pour la ranger dans RI : quatre bus seraient nécessaires, ou bien une lecture en 2 cycles avec un registre tampon, qui pénaliserait l'exécution de toutes les instructions. Comme c'est la seule occurence de ce problème, il est plus économique d'utiliser un incrémenteur dédié.

Pour vérifier que l'architecture de la fig. 5.13 supporte l'architecture logicielle, nous décrirons l'exécution de chaque instruction en *langage transfert* (fig. 5.14). Le langage transfert décrit des affectations depuis un registre vers un registre ; une ligne correspond à un cycle ; on note également les bus

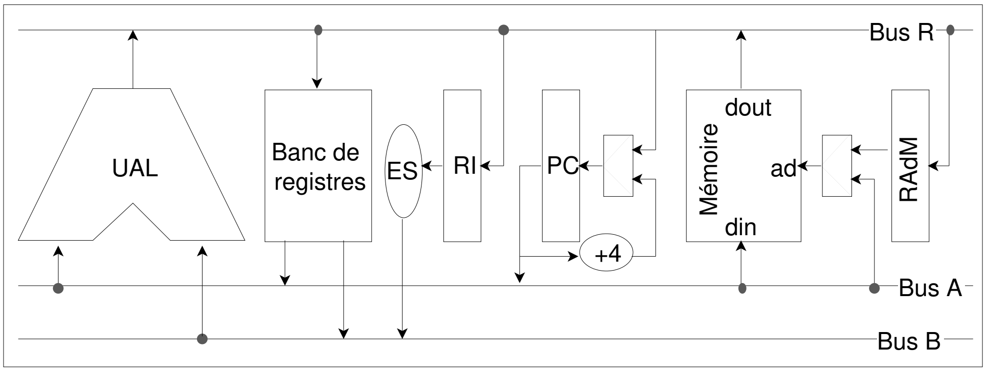


Figure 5.13: Architecture à 3 bus

et les circuits combinatoires qui supportent les transferts.

5.2.4 Performance

Les instructions s'exécutent en 2 cycles, un pour la lecture et un pour l'exécution, sauf les instructions d'accès mémoire, qui consomments 3 cycles. En supposant que 20% des instructions soient des chargements ou des rangements, le nombre de cyles par instruction (CPI) est :

$$\text{CPI} = 0,8 \times 2 + 0,2 \times 3 = 2,2.$$

Donc IPC = 0,45.

5.3 Contrôle

L'organe de contrôle a pour but de générer les signaux qui *commandent* chacun des organes du chemin de données. En particulier, les signaux de contrôle des transferts : signaux Ti de contrôle de transfert des registres vers les bus, signaux d'écriture Wi dans les registres ; et les signaux de contrôle des opérations effectuées par les différents opérateurs : commandes de l'UAL, commandes de lecture et d'écriture dans la mémoire, commande des multiplexeurs.

La partie contrôle doit aussi assurer le séquencement des phases individuelles d'exécution d'une instruction : dans l'exemple ci-dessus, l'évolution du processeur est représenté par l'automate d'états finis de la fig. 5.15. Dans

- Lecture de l'instruction
  PC $\leftarrow$ PC + 4 via INC4 ; RI $\leftarrow$ Mem (PC) via Bus A, bus R.
- UAL Reg-Reg
  Rd $\leftarrow$ Ra OP Rb via bus A, bus B, UAL et bus R.
- UAL Reg-Imm
  Rd $\leftarrow$ Ra OP ES ($RI_{0:15}$) via bus A, bus B, UAL et bus R.
- Chargement Reg-Reg
  Cycle 1 : RAdM $\leftarrow$ Ra + Rb via bus A, bus B, UAL et bus R.
  Cycle 2 : Rd $\leftarrow$ Mem[RAdM] via bus R.
- Chargement Reg-Imm
  Cycle 1 : RAdM $\leftarrow$ Ra +ES($RI_{0:15}$) via bus A, bus B, UAL et bus R.
  Cycle 2 : Rd $\leftarrow$ Mem[RAdM] via bus R.
- Rangement Reg-Reg
  Cycle 1 : RAdM $\leftarrow$ Ra + Rb via bus A, bus B, UAL et bus R.
  Cycle 2 : Mem[RAdM] $\leftarrow$ Rd via bus A.
- Chargement Reg-Imm
  Cycle 1 : RAdM $\leftarrow$ Ra + ES($RI_{0:15}$) via bus A, bus B, UAL et bus R.
  Cycle 2 : Mem[RAdM] $\leftarrow$ Rd via bus A.
- Branchements
  PC $\leftarrow$ PC + ES($RI_{0:22}||00$) via bus A, bus B, UAL et bus R.

Figure 5.14: Description de la micro-architecture DER99/NP en langage transfert.

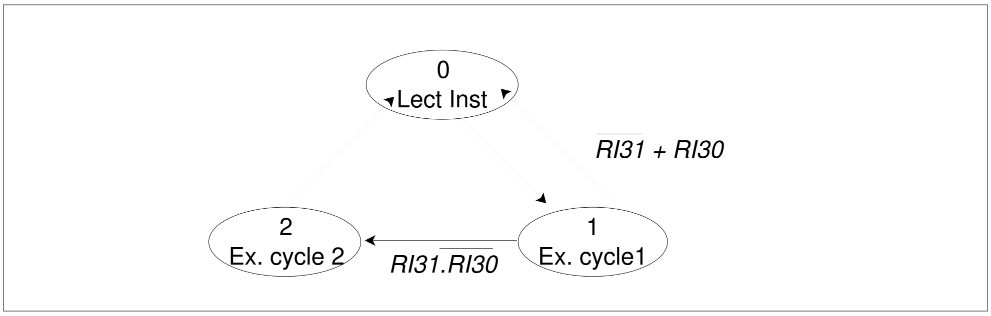


Figure 5.15: Automate de contrôle de la machine DER99/NP

cet automate, les conditions de la transition sont notéees en italique sur les arcs. Les conditions correspondent au codage des instructions présentées ci-dessus.

Enfin, la partie contrôle doit également assurer le traitement des évènements anormaux, typiquement les exceptions provenant de causes internes à l'unité centrale, et les interventions provenant de l'extérieur du microprocesseur, typiquement les interruptions externes. Nous ne traiterons pas cet aspect.

Le contrôle étant par définition une réalisation matérielle, les différents types de contrôle se différencient par leur implémentation : contrôle câblé ou microprogrammé.

5.3.1 Contrôle câblé

Dans le contrôle câblé, l'ensemble des signaux de contrôle sont générés par des automates d'états finis implantés à partir de registres et de logique combinatoire.

L'organisation du contrôle câblé est décrite fig. 5.16. Un circuit combinatoire, typiquement réalisé par un ou plusieurs PLA, décode l'instruction, qui est contenue dans le registre RI. Le résultat du décodage est l'ensemble des fils de contrôle du chemin de données (opérateurs et transferts). L'état d'avancement de l'exécution de l'instruction est contrôlé par un automate d'états finis, qui est lui-même réalisé par un registre d'état et un PLA. Cet état est en entrée du PLA des commandes : les commandes à générer ne sont pas les mêmes pour la deuxième et la troisième phase des instructions d'accès mémoire ; de même pour la première phase, correspondant à la lecture de l'instruction.

La table. 5.5 donne l'exemple du contenu du PLA des commandes pour la

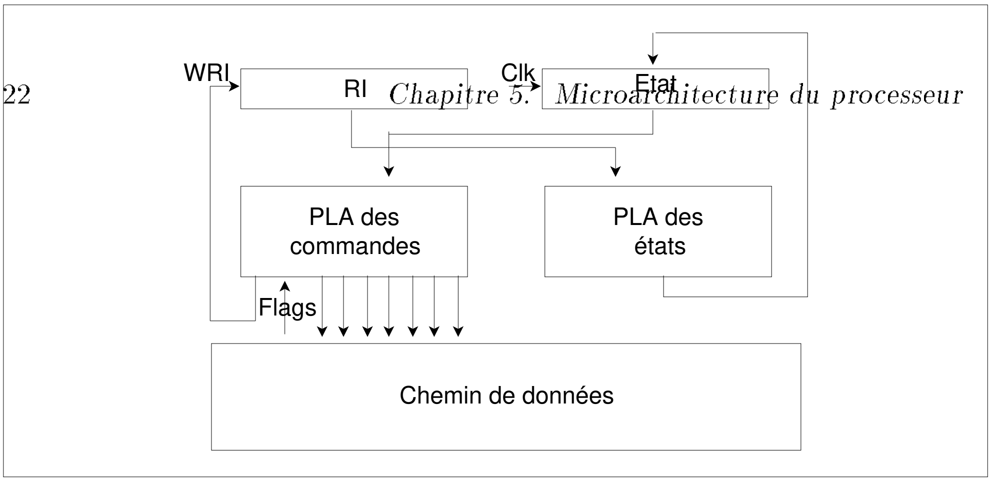


Figure 5.16: Principe du contrôle câblé

machine DER99/NP. Cette table explicite, en termes de signaux de contrôle, la description en langage transfert de la fig. 5.14. La représentation en table de vérité étant peu lisible, la table présente une représentation synthétique : par exemple, la colonne bus A code l'ensemble des barrières de bus vers le bus A; la colonne Wreg rassemble l'ensemble des commande d'écritures de registre ; on n'a pas représenté le décodage des numéros de registres. Un X signifie que la commande est indifférente : c'est le cas des circuits combinatoires lorsque leur résultat n'est pas mémorisé.

La ligne 1 correspond à la lecture de l'instruction. La ligne 2 correspond aux instructions UAL, en mode Reg-Reg, sauf l'instruction CMP (ligne 4). En effet, pour cette instruction, on n'ecrit pas le banc de registres, mais le registre de flags. La lignes 6 décrit le calcul d' adresse pour une instruction d'accès mémoire (premier cycle d'exécution) ; les lignes 7 et 8 l'accès mémoire respectivement pour le chargement et le rangement (deuxième cycle d'exécution).

Explicitons par exemple les commandes en relation avec le compteur de programme. L'état est codé sur 2 bits e_1e_0.

- MPC :

  C'est la commande de sélection du multiplexeur en entrée de PC. L'entrée 0 de ce multiplexeur reçoit la valeur de PC après incrémentation ; l'entrée 1 est connectée au bus R, donc reçoit le résultat d'un calcul d'adresse de branchement. Plusieurs fonctions sont possibles, la plus simple est :

  $$\mathrm{MPC} = e_1,$$

  qui réalise MPC = 0 dans l'état $0 (e_0 = e_1 = 0$, lecture de l'instruction)

	Etat	codop	cond	UAL	ES	Bus A	Bus B	Bus R
1	0	X	X	X	X	PC	Rien	Mem
2	1	UAL Reg-Reg	X	OP	X	Ra	Rb	UAL
3	1	UAL Reg-Imm	X	OP	16	Ra	EXS	UAL
4	1	CMP Reg-Reg	X	OP	X	Ra	Rb	UAL
5	1	CMP Reg-Imm	X	OP	16	Ra	EXS	UAL
6	1	LD/ST Reg-Reg	X	OP	X	Ra	Rb	UAL
7	1	LD/ST Reg-Imm	X	OP	16	Ra	EXS	UAL
8	2	LD	X	X	X	X	X	Mem
9	2	ST	X	X	X	Rd	X	X
10	1	Bcond	cond=True	+	22	PC	EXS	UAL

	Etat	codop	cond	Wreg	MPC	MAd	Mem
1	0	X	X	WRI, WPC	Bus R	Bus A	Lect
2	1	UAL Reg-Reg	X	WRd,WF	X	X	rien
3	1	UAL Reg-Imm	X	WRd,WF	X	X	rien
4	1	CMP Reg-Reg	X	WF	X	X	rien
5	1	CMP Reg-Imm	X	WF	X	X	rien
6	1	LD/ST Reg-Reg	X	WRAdM	X	X	rien
7	1	LD/ST Reg-Imm	X	WRAdM	X	X	rien
8	2	LD	X	WRd	X	RadM	Lect
9	2	ST	X	rien	X	RAdM	Ecr
10	1	Bcond	cond=True	WPC	Bus R	X	rien

Table 5.5: Le PLA des commandes de la machine DER99/NP

et MPC = 1 dans l'état 1, qui est celui où le calcul d'adresse est effectué.

- WPC

  C'est la commande d'écriture du registre PC. D'après la table 5.5, on a WPC = 1 pour

  - la lecture de l'instruction, soit $e_0 e_1 = 00$.
  - ou bien, l'exécution d'une instruction de branchement, donc $e_1 e_0 = 01$ et codop = 11xxx, et que la condition est vraie. La valeur booléenne c associée à la condition est calculée à partir des drapeaux. On a donc :

$$\mathrm{WPC} = \overline{e_1}.\overline{e_0} + \overline{e_1}.e_0.\mathrm{RI}_{31}.\mathrm{RI}_{30}.c$$

5.3.2 Performance

Le décodage introduit une contrainte supplémentaire sur le temps de cycle : les commandes doivent être établies à leur valeur correcte avant que les résultats (par exemple de l'UAL) puissent être échantillonés. En réalité, les commandes doivent être elles-même échantillonées dans des registres, pour être synchronisées dans le temps avec le chemin de données. Sans entrer dans les détails techniques, retenons que le décodage ajoute une contrainte significative sur le temps de cycle.

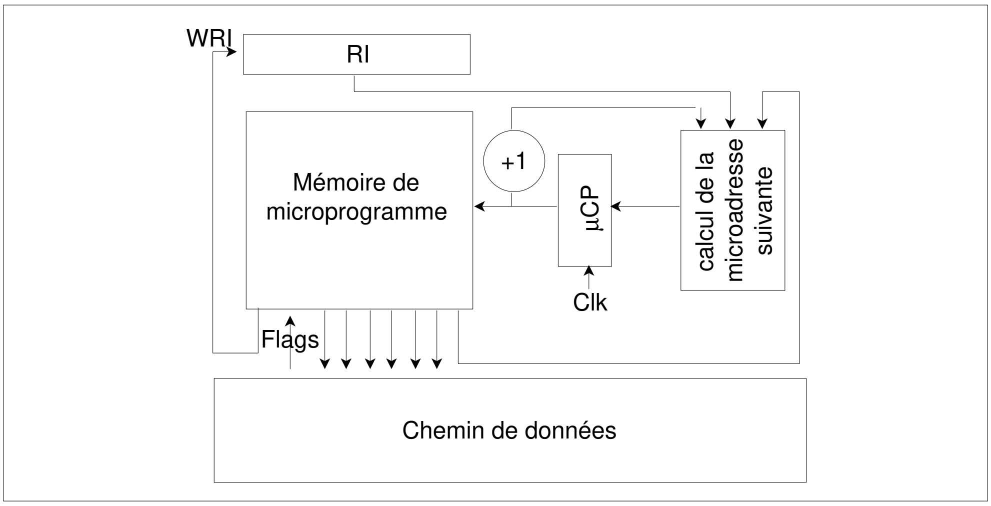


Figure 5.17: Principe de la microprogrammation.

La technique du contrôle câblé est bien adaptée à une microarchitecture où le comportement des instructions est régulier. Dans notre exemple, les instructions consomment toutes deux cycles, sauf les chargements et rangements ; le format est orthogonal, et il n'existe que deux modes d'adressage mémoire. Ainsi, seul le détail des commandes varie. Plus généralement, cette propriété est vraie pour les microarchitectures qui implémentent des architectures RISC. En effet, bien que considérablement plus complexes, ces architectures ont une taille et un format d'instruction fixes.

5.3.3 Microprogrammation

L'autre technique de contrôle est la microprogrammation. Elle a été largement utilisée dans les machines plus anciennes, notamment lorsque le jeu d'instructions est complexe, avec des instructions de longueur variable et des formats complexes. Son schéma de principe est donné fig. 5.17. Dans cette technique, les fils de contrôle du chemin de données correspondent aux sorties (bits) d'une microinstruction qui est contenue dans une mémoire de microprogramme. Le contrôle des actions élémentaires d'une instruction machine est donc réalisé par une suite de microinstructions exécutées par une micromachine, constituée d'un microcompteur de programme et d'une mémoire. Le séquencement des microinstructions est fonction du registre instruction, de l'état du chemin de données, et des informations " partie microadresse suivante" de la microinstruction.

La microprogrammation permet un séquencement plus complexe de l'exécution des instructions : si les instructions demandent en général plusieurs cycles, avec des parties qui sont fréquemment identiques, des branchements dans le microprogramme permettent un gain d'espace et de temps de développement.

5.4 Conclusion

Ce chapitre a présenté une microarchitecture qui est l'une des réalisations possibles d'une architecture logicielle très simple, de type RISC : longueur et format d'instruction fixes. On a vu comment cette microarchitecture peut être construite en organisant autour de trois bus un petit nombre de composants matériels. Soulignons que, malgré sa simplicité, cette microarchitecture est réaliste : l'implémenter à partir d'une bibliothèque de composants ne demanderait que des précisions de détail.

En revanche, cette micro-architecture n'est pas efficace : tous les microprocesseurs actuels sont capables d'exécuter plusieurs instructions par cycle, alors que DER99/NP a un débit inférieur à 0,5 instruction par cycle.

Pour obtenir un meilleur débit, il faut reconsidérer radicalement l'organisation de l'exécution des instructions, en introduisant un principe nouveau, le pipeline.

Chapitre 6

Microarchitecture pipelinée

Les architectures logicielles dominantes sont de type RISC, avec deux caractéristiques fondamentales :

- format d'instructions fixe et simple ;
- accès mémoire limité aux instructions de chargement et rangement

Ces architectures s'opposent aux architectures à jeu d'instructions complexe, dont le paradigme est le jeu d'instructions x86, avec leurs instructions de longueur variable, et où la plupart des instructions peuvent adresser des opérandes mémoire.

Ce chapitre explique la relation fondamentale entre le modèle d'architecture logicielle RISC et un mode d'exécution particulier, le *pipeline*. La propriété essentielle du pipeline est de *d'exécuter une instruction à chaque cycle*, avec un temps de cycle qui est essentiellement celui de l'UAL.

Les premières sections de ce chapitre vont définir le mode d'exécution pipeliné, et montrer qu'il permet d'obtenir un débit de une instruction par cycle. Dans la suite, les contraintes de l'exécution pipelinées seront utilisées pour montrer qu'elles imposent une architecture logicielle RISC. Ensuite, on montrera que le pipeline ne réalise pas naturellement la sémantique séquentielle usuelle des programmes en langage machine, car plusieurs instructions s'exécutent simultanément. Les techniques utilisées pour réaliser quand même cette sémantique séquentielle sont présentés dans les sections suivantes. Enfin, on présentera sommairement les architectures pipelinées superscalaires, qui permettent d'exécuter plusieurs instructions par cycle.

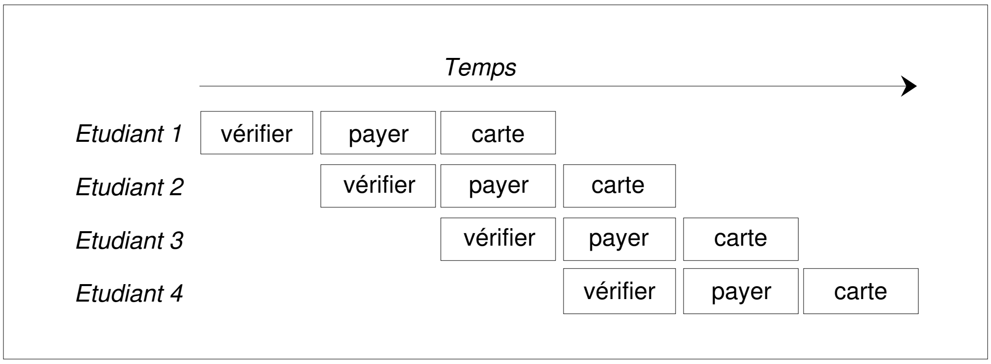


Figure 6.1: Principe du pipeline

6.1 Principe du pipeline

Le principe du pipeline est celui de la chaîne de montage :

- fractionner la tâche en sous-tâches de durées égales, appelées *étages* ;
- exécuter simultanément les différentes sous-tâches de plusieurs tâches.

Par exemple, lors de son inscription, un étudiant doit effectuer trois actions : faire vérifier des diplômes, puis payer, enfin recevoir sa carte. Ces trois actions dépendent l'une de l'autre, et doivent donc être effectuées en séquence. En admettant que chaque action consomme une unité de temps, le temps total pour chaque étudiant est trois unités, et ne peut pas être amélioré. Si chaque étudiant est traité complètement par un seul employé d'administration, le débit est 1/3 d'étudiant par unité de temps.

Avec des ressources plus importantes, et si beaucoup d'étudiants doivent s'incrire, le *débit* du système peut être triplé (fig 6.1) : pendant que le premier étudiant effectue le paiement, un deuxième étudiant entre dans le système et voit ses diplômes verifiés. Au temps suivant, trois étudiants occupent chacun un poste de traitement. Au temps suivant, le premier étudiant est sorti du système ; à partir de là, un étudiant sort du système à chaque unité de temps ; le débit est donc d'un étudiant par unité de temps, bien que le temps individuel vécu par chaque étudiant reste trois unités de temps.

Plus généralement, un système pipeliné est caractérisé par deux paramètres : T, la durée individuelle d'un étage et l, le nombre d'étages du pipeline.

La *latence* L du pipeline est la durée de l'exécution complète d'une tâche :

$$L = lT$$

Le *débit* d du pipeline dépend du nombre n de tâches à exécuter. Le temps nécessaire à l'exécution pipelinée de n tâches est le temps de l'exécution de la première, L, plus le temps nécessaire pour terminer les $n-1$ suivantes :

$$d(n) = \frac{n}{L + (n-1)T};$$

On a

$$\lim_{n \to +\infty} d(n) = T^{-1}.$$

On voit apparaître le résultat essentiel : pour un grand nombre de tâches, le débit ne dépend pas de la latence, mais de la durée individuelle de chaque étage ; donc le débit optimal sera atteint en fractionnant aussi finement que possible la tâche individuelle en sous-tâches.

Le fractionnement a des limites. L'impression d'une carte ne peut guère être décomposée ; le fractionnement peut augmenter légèrement le temps de traitement total (la latence), par exemple du temps nécessaire à transmettre l'information d'un employé administratif au suivant dans le système ; enfin le fractionnement demande une augmentation des ressources disponibles, dans notre exemple passer d'un à trois employés.

6.2 Le pipeline classique pour l'exécution des instructions

6.2.1 Tâches élémentaires

Considérons d'abord l'exécution des instructions les plus fréquentes d'une architecture RISC, les instructions arithmétiques et logiques. Elles nécessitent plusieurs actions élémentaires :

- lecture de l'instruction depuis la mémoire ;
- mise à jour du compteur de programme pour qu'il contienne l'adresse de l'instruction suivante ;
- décodage de l'instruction ;
- lecture des opérandes depuis le banc de registres ;

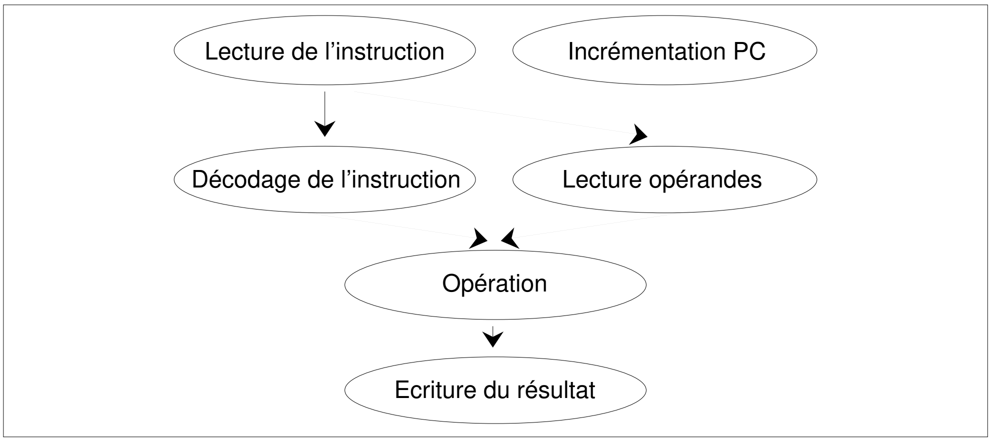


Figure 6.2: Actions élementaires de l'exécution d'une instruction UAL

- opération sur les opérandes ;
- ecriture du résultat.

Toutes ces actions ne s'effectuent pas nécessairement en séquence. La mise à jour de CP peut s'effectuer en parallèle avec la lecture de l'instruction, comme on l'a vu qu chapitre précedent : il suffit de disposer de registres opaques. Plus subtilement, la lecture des opérandes peut s'effectuer en parallèle avec le décodage de l'instruction ; en effet, le décodage des numéros des registres opérandes sources est très rapide, puisque ces numéros occupent un emplacement fixe dans l'instruction ; a fortiori, un opérande immédiat est directement disponible dans le registre RI. Au contraire, le reste du décodage peut être plus complexe. Il est donc possible d'effectuer simultanément ces deux opérations. En revanche, on ne peut ni lire les opérandes, ni décoder l'instruction avant de posséder l'instruction dans un registre du processeur. La fig. 6.2 résume les dépendances entres les actions successives sous forme d'un graphe : un sommet représente une action, un arc la contrainte de séquence.

6.2.2 Pipeline pour les instructions UAL

A cette étape, on peut proposer le pipeline de la fig. 6.3. Idéalement, **ce pipeline assure un débit de 1 instruction par cycle**, ce qui est une modification radicale par rapport à l'architecture non pipelinée du chapitre

LI	Lecture de l'instruction et mise à jour de CP
DI	Décodage des instructions et lecture des opérandes
EX	Exécution dans l'UAL
RR	Rangement du résultat dans le banc de registres

Figure 6.3: Pipeline pour les instructions UAL

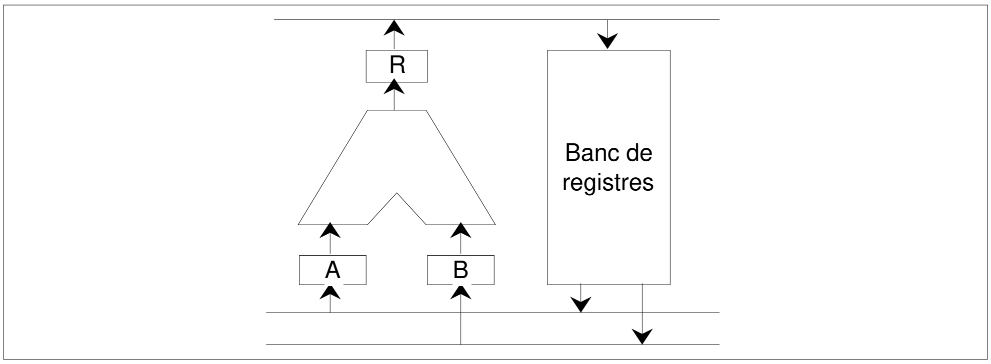


Figure 6.4: Chemin de données pipeliné pour les instructions UAL

précédent. En revanche, le temps d'exécution d'une instruction individuelle est de 4 cycles.

Pour comparer ces performances avec celles de la version non pipelinée, il faut maintenant comparer les temps de cycle des deux versions, donc définir une microarchitecture pipelinée et les contraintes qui lui sont associées. La durée de l'étape la plus longue déterminera le temps de cycle T_c, qui détermine à son tour la durée de tous les étages, puisqu'ils doivent être tous d'égale durée.

Chaque étage doit être accompagné au minimum d'un ensemble de registres internes qui mémorisent les informations calculées dans de cet étage. Par exemple, la notion de lecture des opérandes n'a de sens qu'en ajoutant deux registres A et B en entrée de l'UAL. De même, l'UAL n'a pas de capacité de mémorisation ; il faut donc ajouter un registre R en sortie de l'UAL. Il s'agit de registres internes, donc non visibles à l'utilisateur. Le chemin de données pour les instructions UAL aura donc l'allure de la fig. 6.4. On verra plus loin le détail d'une microarchitecture pipelinée réalisant le jeu d'instruction DE99.

En utilisant les mêmes circuits que dans le chapitre précédent, cette

LI	Lecture de l'instruction et mise à jour de CP
DI	Décodage des instructions et lecture des opérandes
EX	Exécutionde l'opération arithmétique dans l'UAL ou calcul de l'adresse mémoire
MEM	Accès mémoire pour les instructions LD et ST
RR	Rangement du résultat dans le banc de registres

Figure 6.5: Pipeline entier à cinq étages

microarchitecture n'augmente pas, et en fait diminue légèrement, le temps de cycle. En notant T_c^p le temps de cycle dans la version pipelinée et T_c^n le temps de cycle dans la version non pipelinée :

$$T_c^p = T_{\mathrm{UAL}} + T_{\mathrm{Reg}},$$

alors que

$$T_c^n = T_{\mathrm{UAL}} + T_{\mathrm{BReg}}.$$

Le débit asymptotique idéal en instructions UAL est donc $1/T_c^p$ instructions par seconde. En revanche, le temps d' exécution d'une instruction individuelle a été considérablement augmenté : il est passé de T_c^n à $4T_c^p$. **Le pipeline n'est donc efficace qu'alimenté régulièrement en instructions**. Les problèmes liés à l'alimentation du pipeline seront étudiés en 6.5.

6.2.3 Pipeline et accès mémoire

Comme dans le chapitre précédent, les mémoires considérées ici sont des mémoires idéales, dont le temps d'accès est du même ordre que le temps de traversée de l'UAL, et qui correspond dans la réalité à un cache de premier niveau.

L'accès mémoire étant équivalent en temps à une opération UAL, il faut introduire un étage supplémentaire dans le pipeline pour les instructions mémoire. On obtient alors le pipeline de la fig. 6.5.

Deux solutions architecturales sont alors possibles. Toutes les instructions peuvent exécuter toutes les phases du pipeline ; dans ce cas, les instructions UAL ne font rien dans la phase MEM. Ou bien, le pipeline est de longueur variable, les instructions UAL se déroulant en quatre étapes et les instructions mémoire en cinq. C'est plutôt la première solution qui

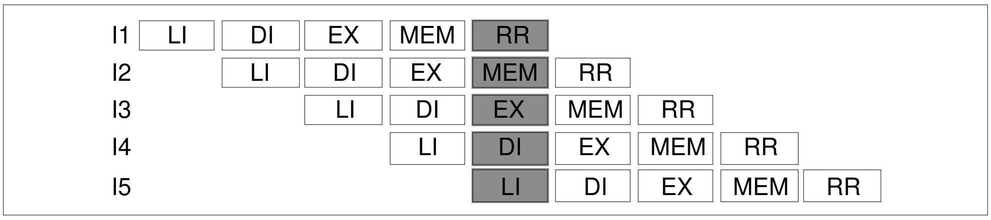


Figure 6.6: Le pipeline classique à cinq étages.

est généralement retenue dans les microprocesseurs actuels : elle simplifie le contrôle du pipeline.

L'introduction d'un étage supplémentaire ne modifie pas le débit, mais ajoute un cycle à la latence, qui devient égale à $5T_c$.

Les instructions de branchement seront traitées en 6.6.

6.3 Les contraintes du pipeline

Dans l'exemple de l'inscription des étudiants, le pipeline imposait d'augmenter les ressources de traitement. Il en est de même pour une micro-architecture.

Avec le pipeline défini ci-dessus, cinq instructions sont simultanément en cours de traitement (fig. 6.6). Chaque étage doit donc disposer exclusivement de toutes les ressources qui lui sont nécessaires. Cette contrainte modifie profondément la micro-architecture, comme on le verra en 6.7. Soulignons seulement maintenant quelques conséquences générales.

6.3.1 Ressources matérielles

Un opérateur par opération

Dans une machine non pipelinée, on peut utiliser le même opérateur (l'UAL) pour la mise à jour de CP, le calcul de l'adresse de l'opérande mémoire ou le calcul de l'adresse de branchement, puis l'opération sur les opérandes, puisque ces différentes opérations interviennent sur des phases successives. Avec un fonctionnement en pipeline, il faut un opérateur par opération pour permettre le fonctionnement simultané. Le nombre exact d'opérateurs nécessaires dépend du nombre d'étages choisi pour le pipeline.

Pour le pipeline classique, il faudra au minimum une UAL, pour les opérations de la phase EX, et un incrémenteur pour CP, dans la phase DI.

Si nous choisissons d'effectuer le calcul de l'adresse de branchement dans la phase EX, ces ressources sont suffisantes.

Accès multiples au banc de registres

En mode d'adressage base + index, une instruction de rangement (ST Rd, Ra, Rb) demande une lecture registre supplémentaire, celle de Rd. Cette lecture va se dérouler, soit à l'étage DI de l'instruction ST, soit pendant un autre étage ; mais celui-ci sera simultané avec l'étage DI d'une autre instruction, qui consomme déjà les deux lectures du banc de registres. Le mode d'adressage base + index impose donc un banc de registres qui autorise trois lectures simultanées. C'est la raison pour laquelle certaines architectures (MIPS, Alpha) ne disposent que du mode d'adressage base + déplacement.

Cache instructions et données séparés

Si l'instruction I_2 est une instruction mémoire, la lecture de l'instruction I_5 (phase LI) crée un conflit avec la phase MEM de l'instruction I_2 : par définition, un circuit mémoire ne permet qu'un seul accès par cycle. La solution la plus simple consiste à disposer de deux caches séparés pour les instructions et les données. Cette organisation a reçu le nom d'*architecture Harvard*.

On verra plus en détail le fonctionnement des caches dans le chapitre 7. Indiquons seulement ici que l'architecture Harvard ne contredit pas le modèle d'exécution de Von-Neumann : instructions et données résident bien dans le même espace d'adressage.

6.3.2 Format d'instruction fixe et simple - Cache d'instruction

Le pipeline d'exécution des instructions implique qu'à chaque cycle, une instruction complète est lue à partir de la mémoire, dans l'étage LI. Pendant qu'elle s'exécute dans les étages suivants, les instructions qui la suivent dans le programme sont à leur tour lues. Ce mécanisme régulier serait contradictoire avec des instructions de longueur variable, nécessitant plusieurs accès mémoire. L'instruction a donc une taille fixe et son format est limité par la taille d'un mot mémoire.

La lecture d'une instruction par cycle n'est pas possible à partir de la mémoire principale, qui est un ordre de grandeur plus lente que le processeur. Un cache d'instructions interne au processeur est donc obligatoire.

D'autre part, le décodage doit être rapide, pour ne pas pénaliser le temps de cycle. Un format orthogonal simple le permet, ce qui autorise alors un contrôle câblé.

6.3.3 Architecture chargement-rangement

La validité de l'approche RISC impose, comme on vient de le voir, un cache d'instructions interne. Pourquoi ne pas inclure aussi un cache de données interne, et autoriser les accès mémoire dans les instructions arithmétiques ? Si le budget en transistors des premiers RISC ne le permettait pas, tous les microprocesseurs actuels incluent un cache de données. Cependant, ils respectent strictement le modèle chargement-rangement.

Le comportement des caches est par définition non prédictible : la donnée peut être présente dans le cache (*succès*), le temps d'accès est alors de un cycle ; mais la donnée peut aussi être absente (*échec*), et la pénalité sur le temps d'accès peut alors être de plusieurs temps d'accès à la mémoire principale. Dans les architectures RISC, la gestion de cet aléa est limitée aux instructions d'accès mémoire. En forçant les instructions UAL à opérer sur des registres, on garantit qu'elles s'exécuteront sans problème lié à la hiérarchie mémoire. On verra dans la partie consacrée aux dépendances de données (6.5.1) que cet avantage est décisif.

6.4 CISC et RISC

Les architectures à jeu d'instructions complexe (CISC) ont été conçus à partir années 1960 sur la base des données technologiques de l'époque.

- Les compilateurs ne savent pas bien utiliser les registres.
- Les microinstructions s'exécutent beaucoup plus vite que les instructions ; en effet, dans les années 60, les premières résident dans une mémoire à semiconducteurs, alors que les secondes résident dans une mémoire à tore. Même dans la suite, l'écart entre temps d'accès mémoire et temps de cycle processeur reste important.
- La mémoire centrale est très limitée en taille ; il est donc important de diminuer la taille du programme.

Ces caractéristiques ont conduit à privilégier les modes mémoire-mémoire et les modes mémoire-registre, avec des instructions complexes, utilisant un nombre important de modes d'adressage. Ceci a conduit à des instructions de longueur variable, pour laquelle le contrôle microprogrammé est le plus efficace. Les architectures les plus représentatives de cette classe sont l'IBM 360 et 370, le VAX-11 et les séries Intel 80x86 et Motorola 680x0. L'IBM 360 et le Vax ont en commun un grand nombre d'instructions, des instructions de taille variable (avec une grande amplitude de variation), plusieurs modèles d'exécution et des mémoires de microprogramme de taille conséquente. On peut remarquer que les modes utilisés par l'architecture Intel 80x86 sont moins complexes que ceux du VAX-11, ce qui a permis la réalisation de processeurs CISC performants (80486, Pentium) en dépit du handicap du jeu d'instructions CISC.

L'évolution des performances des circuits intégrés et les progrès de la technologie des compilateurs ont conduit à remettre en cause les axiomes de la période précédente, notamment à partir d'une étude quantitative de l'utilisation des jeux d'instructions des machines CISC.

Ces études ont été menées principalement par les créateurs des architectures RISC John Hennessy et David Patterson. Une synthèse de ces travaux est disponible dans un ouvrage [7], qui a très fortement marqué la conception d'architecture, en imposant une discipline d'approche quantitative. Les résultats sont très clairs : les modes d'adressage sophistiqués et les instructions complexes sont rarement utilisés. Pire, certaines instructions complexes sont plus longues à exécuter que leur émulation par une séquence d'instructions plus simples.

D'autre part, la technologie des compilateurs a progressé. En particulier, l'adaptation d'algorithmes classiques de coloriage de graphe leur permet d'allouer efficacement les registres.

Surtout, il n'y a plus de différence significative entre les performances des mémoires utilisée pour les microprogrammes et celles des mémoires caches, utilisées pour les instructions et les données les plus fréquemment utilisées.

Ceci a conduit au début des années 1980 à l'introduction du concept de machine à jeu d'instruction réduit (RISC), qui se fonde essentiellement sur un jeu d'instructions permettant une exécution pipeline efficace, grâce des instructions de longueur fixe, avec des formats simples faciles à décoder. La simplicité du décodage diminue drastiquement la fraction de la surface du processeur dédiée au contrôle.

Initialement, l'espace libéré par la mémoire de microprogramme était dédié à l'implantation d'un banc de registres qui fournissait beaucoup plus de registres que les microprocesseurs CISC. Ce grand banc de registres est crucial pour permettre au compilateur de conserver les variables vivantes en

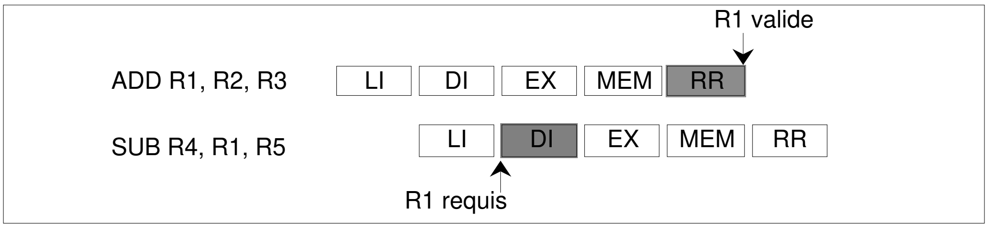


Figure 6.7: Un exemple de dépendances de données

registre, sans devoir les écrire ou les lire en mémoire à chaque accès.

6.5 Les limitations du pipeline.

Le débit idéal d'une instruction par cycle n'est possible que si les instructions sont essentiellement indépendantes ; elles peuvent alors s'exécuter simultanément Lorque leur sémantique est réellement séquentielle, le pipeline peut être pénalisé. Cette section et la suivante étudient ces limitations, et quelques solutions classiques pour les contourner.

6.5.1 Aléas de données.

Le terme d'*aléa de données* correpond, dans le cas des microprocesseurs, aux dépendances de données en théorie de la compilation, lorqu'elles portent sur des **registres**.

La fig. 6.7 en donne un exemple : la première instruction écrit le registre R1, et l'instruction suivante le lit. Si initialement Ri =i, on doit obtenir $2 + 3 - 5 = 0$ dans R4. Or, l'instruction ADD lit le registre R1 dans son étage DI, donc au cycle 3. L'instruction SUB n'écrira son résultat dans R1 qu'au cycle 5. La valeur lue est donc la valeur antérieure du registre R1, soit 1, et le résultat est -1. Bien évidemment, ce comportement n'est pas acceptable : le processeur doit fournir le résultat qui correspond à la sémantique du programme !

Le problème peut être un peu allégé, mais non résolu, en disposant d'un banc de registres rapide. Le banc de registres écrit sur le front descendant de l'horloge, et la valeur est disponible en sortie du banc de registres avant le front montant suivant (fig. 6.8). Comme l'étage RR n'effectue pas de calcul, mait seulement un transfert entre registres, ceci n'impose pas de contrainte

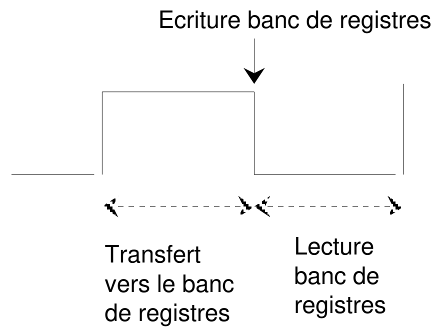


Figure 6.8: Ecriture avant lecture dans le banc de registres

critique sur le temps de cycle. A partir de maintenant, on supposera toujours ce fonctionnement.

Pour que le problème d'aléa de données n'intervienne pas, il faudrait que les deux instructions en dépendance soient séparées dans le programme en langage-machine par au moins 2 instructions (fig. 6.9).

Instructions UAL

La dépendance de donnée n'est pas intrinsèque, mais provient de la structure du pipeline : la valeur nécessaire à l'étage EX de l'instruction ADD est disponible au début de cet étage (fig 6.10). Pour utiliser cette valeur, il faut modifier la micro-architecture en introduisant le mécanisme d'*anticipation* décrit fig. 6.11. Dans le cas de deux instructions consécutives en dépendance, le résultat de l'opération UAL est réinjecté en entrée des registres A et B ; dans le cas où les deux instructions en dépendance sont séparées par une instruction, le contenu du registre R est choisi. Dans les deux cas, le contenu du bus en provenance du banc de registre n'est pas considéré. Les multiplexeurs en entrée des registres A et B permettent de choisir l'opérande source.

La complexité matérielle supplémentaire du chemin de données reste modeste. Cependant, la partie contrôle doit traquer en permanence les dépendances : elle compare, dans l'étage de décodage d'une instruction i, les numéros du (format Reg-Imm) ou des (format Reg-Reg) registres sources de l'instruction i avec le numéro du registre destination des deux instructions précédentes ; en fonction du résultat de cette comparaison, la commande

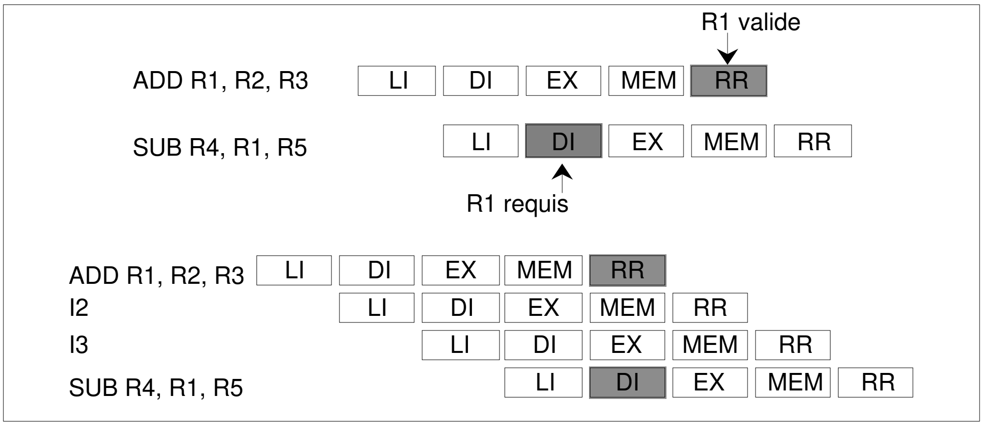


Figure 6.9: Aléas de données et écriture avant dans le banc de registres

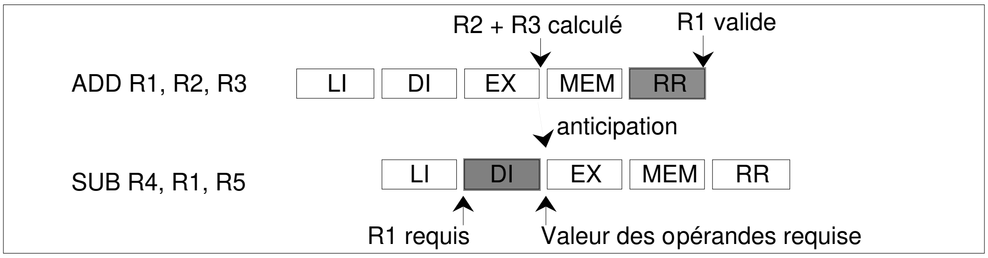


Figure 6.10: Anticipation et pipeline

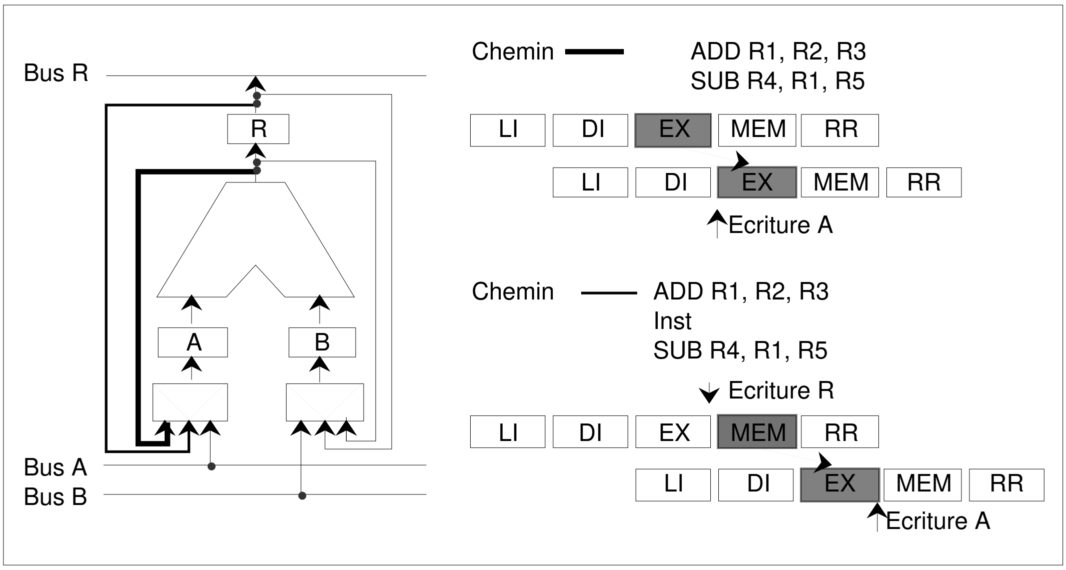


Figure 6.11: Anticipation et pipeline

adéquate est envoyée aux multiplexeurs.

C'est l'architecture chargement-rangement qui rend possible un traitement simple des aléas de données, qui découlent du pipeline, en le limitant aux registres. En effet, il n'est pas possible en général de décider à la compilation si deux références mémoire pointent ou non la même case mémoire. Si ceci devait être décidé à l'exécution, la partie contrôle devrait comparer les résultats des calculs d'adresse, tout en gérant la hiérarchie mémoire.

Autres instructions

Le problème des aléas de données ne se limite pas aux instructions UAL : il concerne tous les cas où le registre destination d'une instruction est source pour une instruction suivante.

Les instructions qui produisent une valeur sont les instructions UAL et les chargements. Les instructions qui consomment une valeur sont :

- les instructions UAL et les instructions d'accès mémoire du point de vue du calcul d'adresse (par exemple, dans l'instruction ST Rd, Ra, Imm, on considère seulement Ra) ;
- les instructions de rangement, pour le registre rangé ;

Inst source Inst. destination	UAL	Chargement

UAL ou
calcul d'adresse
	ADD R1, R2, R3
	SUB R4, R1, R5
	latence 1
		LD R1, 0(R2)
		SUB R4, R1, R5
		latence 1

Rangement	ADD R1, R2, R3 ST R1, 0(R4) latence 2	LD R1, 0(R2) ST R1, 0(R4) latence 1

Table 6.1: Aléas de données pour les instructions entières de DER99

- les instructions de branchement, pour les registres de déplacement.

Le comportement des instructions UAL et d'accès mémoire a été complètement spécifié ci-dessus (fig. 6.6). Les instructions de branchement seront traitées en 6.6. La table. 6.1 décrit l'ensemble des aléas de données possibles pour le pipeline classique et le jeu d'instruction DER99. La table contient un exemple de chaque dépendance, et la latence *avec la meilleure anticipation possible*. La latence est égale à $1 + N_c$, où N_c est le nombre de cycles qui doivent séparer deux instructions pour qu'elles s'exécutent correctement ; une latence 1 signifie que les instructions peuvent être immédiatement consécutives. La fig. 6.12 explicite les anticipations.

- source = instruction UAL ; destination = instruction UAL ou mémoire pour l'adresse : le mécanisme d'anticipation permet une latence 1.
- source = instruction UAL ; destination = rangement pour la donnée. La donnée n'est requise qu'à l'étage Mem. Il est donc possible de créer, en entrée de la mémoire, un mécanisme d'anticipation analogue à celui en entrée de l'UAL. Une latence 1 est donc réalisable. Notons qu'un seul niveau d'anticipation est requis.
- source = chargement ; destination = instruction UAL ou mémoire pour l'adresse. Ici, la valeur est produite pendant le cycle où elle serait requise, donc trop tard pour être écrite dans une registre (fig. 6.13). En outre, le temps d'accès au cache est bien de l'ordre de grandeur du temps de cycle : il n'est donc pas possible d'utiliser la méthode lecture avant écriture du banc de registres. Il n'est donc pas possible de dérouler normalement le pipeline. Tous les processeurs actuels contrôlent cetet situation par matériel : la partie contrôle insère une

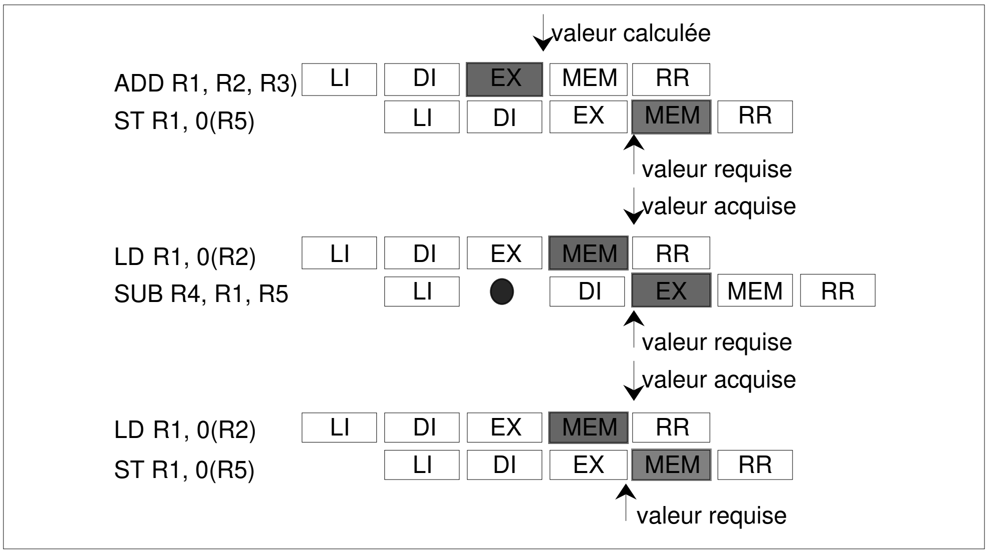


Figure 6.12: Anticipations et suspensions

suspension dans le pipeline. L'instruction consommatrice est bloquée jusqu'à ce que la donnée soit disponible. Dans la fig. 6.13, on a supposé qu'une anticipation était implémentée pour acheminer la donnée vers l'étage EX dès que possible ; si ce n'était pas le cas, il faudrait introduire deux suspensions au lieu d'une.

- source = chargement ; destination = rangement pour la donnée. La donnée n'est requise qu'à l'étage MEM, donc après le chargement effectif. En étendant le mécanisme d'anticipation en entrée de la mémoire à un deuxième niveau, aucune suspension n'est nécessaire et une latence 1 est réalisable.

Performances

L'introduction d'une suspension décale tout le flot d'instructions (fig. 6.13). Chaque instruction suspendue coûte un cycle supplémentaire. Si m est le pourcentage d'instructions de chargement qui créent une dépendance, le débit asymptotique en instructions par cycles devient :

$$\text{IPC} = \frac{1}{1+m}.$$

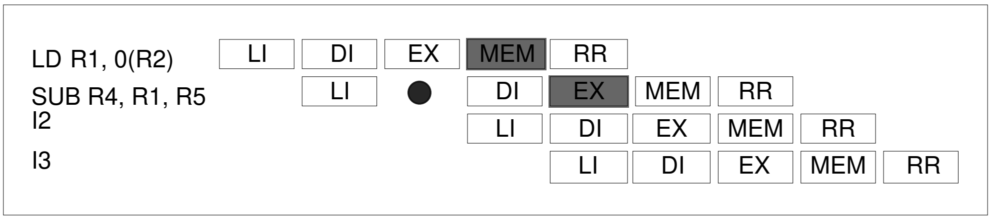


Figure 6.13: Décalage des instructions dû à une suspension

Pour éviter cette pénalité, le compilateur peut réordonner les instructions. Reprenons une fois de plus l'exemple d'une boucle de parcours de tableau (fig. 6.14).

```
int s, X[N] ;
for(i = 0 ; i < N ; i++)
   s = s + X[i] ;
```

Figure 6.14: Cumul d'un tableau

En supposant qu'initialement R1 = @X[0], R5 = N, et R3 = @X[N-1], le corps de la boucle prend un cycle de retard à chaque itération (on ne tient pas compte des problèmes liés au branchement, qui seront étudiés plus loin) :

Cycle	Instruction			
1	bcl :	LD	R10, 0(R1)	
3		ADD	R2, R2, R10	; alea R10 donc suspension
4		ADD	R1, R1, 4	
5		CMP	R1, R3	
6		BLE	bcl	

La performance de ce code est, en notant P le nombre d'additions par cycle :

$$P^1 = \frac{1}{6} = 0,17 \text{ et IPC}^1 = \frac{5}{6} = 0,83.$$

La suspension peut être avantageusement remplacée par l'incrémentation d'adresse, en modifiant l'ordre des instructions :

Cycle	Instruction			
1	bcl :	LD	R10, R1	
2		ADD	R1, R1, 4	
3		ADD	R2, R2, R10	; la latence 2 est respectee
4		CMP	R1, R3	
5		BL	bcl	

La performance devient

$$P^2 = \frac{1}{5} = 0,2 \text{ et IPC}^2 = 1.$$

Avec un jeu d'instruction plus puissant, l'optimisation est un peu plus difficile. Si le jeu d'instruction dispose d'un mode d'adressage autoincrémenté (LDU, STU) et d'un branchement sur compteur (BCTR), l'incrémentation d'adresse et le test disparaissent (le registre compteur avec la valeur de N) :

Cycle	Instruction			
1	bcl :	LDU	R10, 4(R1)	
3		ADD	R2, R2, R10	; alea R10
4		BCTR	bcl	

La performance devient

$$P^3 = \frac{1}{4} = 0,25 \text{ et IPC}^3 = \frac{3}{4} = 0,75.$$

Ici, la débit de calcul est amélioré, mais le débit d'instruction dégradé. L'enrichissement du jeu d'instruction n'est donc pas complètement exploité.

Le déplacement d'instruction n'est pas possible parce que le corps de boucle ne contient pas assez d'instructions. La technique classique pour résoudre ce problème est le *déroulement de boucle*. En langage C, en supposant que $N = 2M$, le code correspondrait à celui de la fig. 6.15.

Le code compilé en déroulant la boucle et en déplaçant la deuxième instruction de chargement est :

Cycle	Instruction		
1	bcl :	LDU	R10, 4(R1)
2		LDU	R11, 4(R1)
3		ADD	R2, R2, R10
4		ADD	R2, R2, R11
5		BCTR	bcl

```
int s, X[N] ;
for(i = 0 ; i < M ; i= i+2) {
   s = s + X[i] ;
   s = s + X[i+1] ;
}
```

Figure 6.15: Déroulement du cumul d'un tableau

La performance devient

$$P^3 = \frac{2}{5} = 0,4 \text{ et } \mathrm{IPC}^3 = 1.$$

De même que les compilateurs éliminent les chargements et rangements inutiles, ils déroulent les boucles et réordonnent les instructions sans modification du code source. La seule intervention de l'utilisateur se situe au niveau des options de compilation qui contrôlent l'optimisation. Ces options sont spécifiques de chaque compilateur et de chaque machine cible. Soulignons encore une fois que l'exploitation efficace des processeurs actuels impose l'emploi systématique des options d'optimisation.

Le déroulement de boucle peut créer un surcoût. Dans l'exemple précédent, N est supposé pair, et sa valeur connue à la compilation (donc une constante). En réalité, les compilateurs déroulent les boucles trois fois, donc obtiennent un corps de boucle formé de quatre corps de la boucle initiale. Un avantage supplémentaire est d'éliminer trois instructions de contrôle sur quatre. L'inconvénient est que si N est remplacé par une variable n, dont la valeur n'est pas connue à la compilation, le compilateur doit ajouter un code de traitement des itérations restantes, qui a en général la structure décrite fig. 6.16. Si le nombre d'itérations est petit, l'exécution de la boucle optimisée sera beaucoup plus lente que celle de la boucle non optimisée !

6.6 Aléas de contrôle

Les instructions qui modifient l'exécution séquentielle des instructions sont les sauts, les branchements et les appels de procédure. La sémantique séquentielle impose que la séquence syntaxique de la fig. 6.17 produise l'une des deux exécutions I1 ; I2 ou J1 ; J2, suivant la condition. On va voir

```
r = n % 4 ;
m = n / 4 ;
pour i = 0 , m-1 faire
   < code ordonnance des iterations 4i, 4i+1, 4i+2, 4i+3 >
ffaire
si r = 3 { iteration 3 ; r = r-1 ;}
si r = 2 { iteration 2 ; r = r-1 ;}
si r = 1 { iteration 1}
```

Figure 6.16: Structure de contrôle associée à une boucle déroulée trois fois

```
        Bcond cible
        I1
        I2
        ...
cible : J1
        J2
```

Figure 6.17: Structure conditionnelle générique

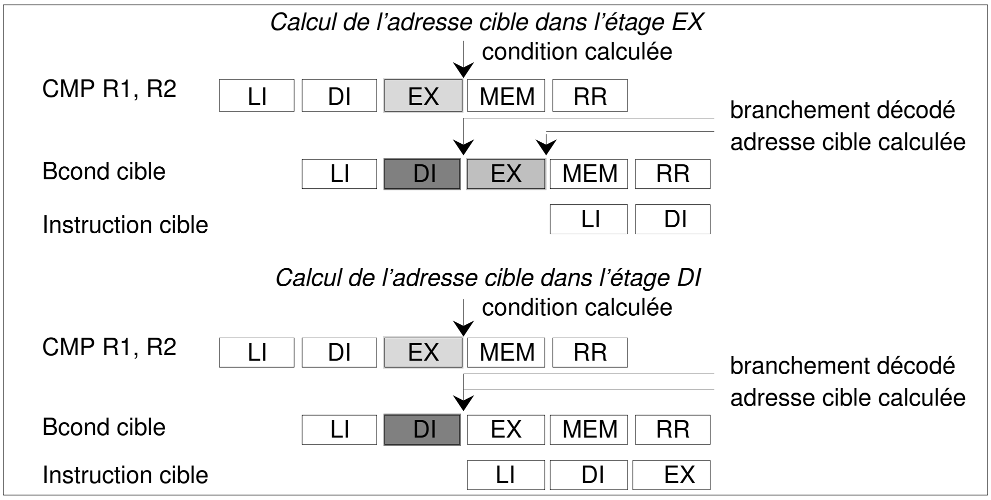


Figure 6.18: Origines des aléas de contrôle

que réaliser cette sémantique n'est pas simple pour une microarchitecture pipelinée. On parle alors d'*aléa de contrôle*.

6.6.1 Les différents types d'aléas de contrôle

La fig. 6.18 illustre les trois problèmes posés au pipeline par les ruptures de séquence.

Décodage des sauts et branchements

Il faut au minimum avoir décodé l'instruction pour savoir qu'il s'agit d'une instruction de saut ou branchement. La nature de l'instruction ne peut être connue avant l'étage DI du pipeline, ce qui veut dire que l'instruction cible du saut ou branchement ne peut être acquise (étage LI) qu'au cycle $c+2$, si l'instruction saut ou branchement commence au cycle c.

Calcul de l'adresse cible

Pour lire l'instruction cible du branchement, il faut évidemment connaître son adresse. Dans le cas d'un saut (PC $\leftarrow$ Registre ou Imm), l'adresse contenue dans un registre est obtenue dans la phase DI. Dans le cas d'un branchement avec déplacement immédiat, (PC $\leftarrow$ PC + Imm), il faut ajouter le contenu de PC et le déplacement. Les deux opérandes sont disponibles à la fin de l'étage LI, puisque l'immédiat est contenu dans RI ; l'addition peut donc s'effectuer dans l'étage DI. Mais ceci impose de disposer d'un additionneur spécifique. Si l'additionneur de l'UAL était requis, il y aurait conflit de ressources avec l'étage EX d'une autre instruction. Dans ce cas, le branchement peut être effectué à la fin du cycle DI, c'est à dire l'instruction cible commencer au cycle $c + 2$ si le branchement commence au cycle c.

Si on utilise l'UAL pour calculer l'adresse de branchement au lieu d'un additionneur spécifique, alors on obtient celle-ci à la fin de la phase EX, et l'instruction cible commence au cycle $c + 3$ si le branchement commence au cycle c.

Calcul des conditions

Il faut calculer la condition pour les sauts et les branchements conditionnels. Le traitement de la condition dépend de la manière dont sont organisés les instructions conditionnelles. Si le résultat des comparaisons est rangé dans un registre code condition, il n'est pas disponible avant la fin de la phase EX de l'instruction qui positionne la condition. L'instruction cible peut donc commencer au cycle $c + 2$.

Un cas plus critique peut intervenir avec des instructions de branchement conditionnel plus général, permettant de tester directement une condition entre deux registres, et d'effectuer un branchement si le booléen correspondant est vrai. Une telle instruction a une syntaxe du style Bcond Ri, Rj, déplacement, et la condition de branchement est Ricond Rj. Le calcul du booléen Ricond Rj n'est pas réalisable en un demi-cycle d'horloge, puisqu'il correspond à une opération UAL. Rallonger le temps de cycle d'horloge pour une seule instruction est évidemment une très mauvaise solution du point de vue efficacité globale de la machine. Le test de la condition intervient alors dans l'étage EX du branchement, donc l'instruction cible ne peut pas commencer avant le cycle $c + 3$.

6.6.2 Traitement des aléas de contrôle

Dans le meilleur cas, on peut obtenir l'adresse de branchement et la valeur de la condition à la fin de la phase DI. Il y a alors un délai de un cycle avant le démarrage de l'instruction cible. Ce délai est occupé par l'instruction qui suit syntaxiquement l'instruction de branchement (instruction I1 de la figure 6.17). Plus généralement, les cycles séparant une instruction de branchement de l'instruction cible sont appelés le *délai de branchement*. Dans la suite, on prend l'exemple d'un délai de branchement de 1.

Les codes non numériques peuvent présenter un très grand nombre de branchements. Réduire la pénalité associée a donc été l'objet de recherches très actives, qui ont fait considérablement évoluer les solutions en quelques années.

Annulation d'instruction

Une première technique consiste à annuler l'exécution en cours de l'instruction délai de branchement. Après au cycle $c + 2$, tout est résolu, et l'étage LI peut lire, soit I1, si le branchement n'est pas pris, soit J1, si le branchement est pris. Annuler l'exécution d'une instruction consiste à l'empêcher de modifier l'état visible de la machine : par exemple, en inhibant la commande d'écriture, l'écriture dans le banc de registres n'a pas lieu ; l'UAL peut être traversée, mais le registre de drapeaux ne doit pas être écrit. Ici, L'annulation peut commencer dès l'étage DI de l'instruction de branchement ; I1 effectue son étage LI, le pipeline arrive "à temps" pour bloquer l'incrémentation de PC.

La technique d'annulation inconditionnelle ne dépend pas de l'étage de calcul de la condition : tout branchement annule l'instruction suivante. Cette propriété rend sa réalisation matérielle facile, mais présente l'inconvénient de perdre systématiquement un cycle à chaque rupture de séquence. En particulier, si le branchement n'est pas pris (condition fausse), on annule I1 pour recommencer son exécution au cycle suivant.

Les sauts et branchements retardés

La seconde technique consiste, au lieu d'annuler l'exécution de l'instruction du délai de branchement, à l'exécuter systématiquement, en disant que l'effet d'un saut ou d'un branchement est *retardé* d'une instruction : l'instruction qui suit séquentiellement le saut ou le branchement est exécutée, avant d'exécuter l'instruction cible du branchement. Donc, le branchement retardé **modifie la sémantique des programmes en langage-machine**. Par exemple, R3 $\leftarrow$ Min(R1, R2) peut se coder par :

CMP R1, R2	
BL suite	; Branchement si R1 < R2
ADD R3 R1 R0	; delai de branchement, R3 $\leftarrow$ R1 **toujours** execute
	; le branchement prend effet ici

```
    ADD R3 R2 R0   ; R3 ← R2, si le branchement n'est pas pris ie R1 ≥ R2
suite.:.           ; execute a partir des deux chemins
```

Dans le cas général, le compilateur a pour tâche d'insérer une instruction adéquate après l'instruction de saut/branchement du code naturel. La solution toujours correcte est l'insertion d'une instruction neutre (NOP). la performance est alors identique à celle de l'annulation. Une solution plus efficace est l'insertion d'une instruction utile, extraite du reste du code. Par exemple, le code non optimisé du calcul précédent est :

```
    ADD R3 R1 R0   ; R3 ← R1 toujours execute
    CMP R1, R2
    BL   suite     ; Branchement si R1 < R2
    NOP            ; delai de branchement
                   ; le branchement prend effet ici
    ADD R3 R2 R0   ; R3 ← R2, si le branchement n'est pas pris ie R1 ≥ R2
suite.:.           ; execute a partir des deux chemins
```

et l'instruction ADD est déplacée pour prendre la place du NOP.

Cependant, il n'est pas toujours possible de trouver avant le branchement une instruction qui puisse être déplacée, par exemple si on a une chaîne d'instructions qui aboutissent au calcul de la condition. Une solution proposée par exemple par l'architecture SPARC réside dans les *branchements avec annulation*, qui combinent le branchement retardé avec l'annulation. Il s'agit d'un nouveau type d'instruction (dans la syntaxe SPARC Bxx,a) ; dans le cas particulier du SPARC, l'instruction qui suit syntaxiquement le branchement est exécutée si le branchement est pris, mais est annulée si le branchement n'est pas pris. On peut alors prélever une instruction dans le code cible du branchement pour l'insérer dans le délai de branchement. Cette instruction est particulièrement commode pour compiler les boucles (fig. 6.19).

6.7 Micro-architecture pipelinée

Ce paragraphe présente une microarchitecture qui implémente le jeu d'instruction DE99, sauf le mode d'adressage base + index. La fig. 6.20 présente la microarchitecture, et la table 6.2 sa description en langage transfert. Comme la micro-architecture non pipelinée, c'est une architecture à trois bus. La principale différence est l'introduction de registres internes, qui passent l'information entre les étages.

Code initial

```
bcl :   Inst A
        Inst B
        ...
        Bcond bcl
        NOP
```

Après optimisation

```
        inst A
bcl :   Inst B
        ...
        Bcond bcl
        Inst A
```

Figure 6.19: Optimisation d'une boucle pour un branchement avec annulation.

En particulier, le registre compteur de programme est implémenté par cinq registres CP_LI,CP_DI, CP_EX, CP_MEM, CP_RR. En effet, le compteur de programme peut être nécessaire pendant toute l'exécution de l'instruction, en particulier pour traiter les branchements et les exceptions ; d'autre part, le compteur de programme est modifié à chaque cycle ; il faut donc autant de registres que d'étages du pipeline pour conserver simultanément cinq compteurs de programmes différents. S'il n'y a pas de rupture de séquence, à chaque cycle, PC_LI est incrémenté et le contenu des registres est décalé d'un cran.

De même, le registre RI est implémenté par quatre registres, RI_DI, RI_EX, RI_MEM et RI_RR.

La microarchitecture réalise un délai de branchement de 1 instruction. Pour cela, un additionneur a été ajouté, qui peut effectuer soit +4, soit la somme de ses entrées. Les instructions de branchement sont en fait terminées après l'étage DI, sauf pour la recopie des registres d'état.

6.8 Instructions flottantes

L'exécution des opérations flottantes consomme plusieurs cycles. Pour les instructions flottantes, le pipeline comporte plusieurs étages d'exécution.

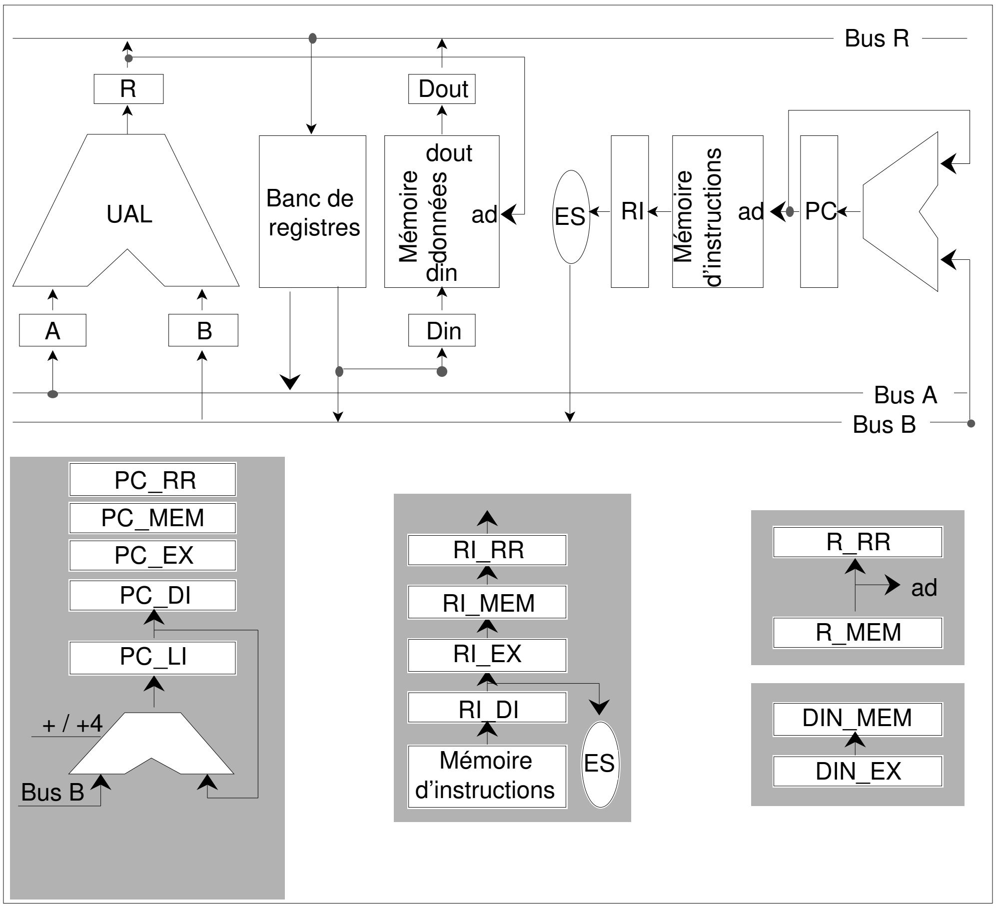


Figure 6.20: Microarchitecture pipelinée

<table>
<tr><th>Etage</th><th colspan="3">Transferts</th></tr>
<tr><td>LI</td><td colspan="3">RI_DI ← Mem[PC_LI] ; PC_DI ← PC_LI ; PC_LI ← PC_LI +4</td></tr>
<tr><td rowspan="2">DI</td><td colspan="3">RI_EX ← RI_DI ; PC_EX ← PC_DI</td></tr>
<tr><td>Format Reg-Reg
A ← Ra
B ← Rb</td><td>Format Reg-Imm
A ← Ra
B ← ES(Imm16)
si ST , DIN_EX ← Rd</td><td>Format Bcht

PC_LI ← PC_DI + ES(Imm22)</td></tr>
<tr><td rowspan="2">EX</td><td colspan="3">RI_MEM ← RI_EX ; PC_MEM ← PC_EX ; DIN_MEM ← DIN_EX</td></tr>
<tr><td colspan="3">Format Reg-Reg et Reg-Imm
R_MEM ← A OP B</td></tr>
<tr><td rowspan="2">MEM</td><td colspan="3">RI_RR ← RI_MEM ; PC_RR ← PC_MEM</td></tr>
<tr><td>Instructions UAL
R_RR ← R_MEM</td><td colspan="2">LD/ST
LD DOUT ← Mem[R_MEM]
ST Mem[R_MEM] ← DIN</td></tr>
<tr><td>RR</td><td>Instructions UAL
Rd ← R_RR</td><td colspan="2">LD
Rd ← DOUT</td></tr>
</table>

Table 6.2: Contrôle de la microarchitecture pipelinée

Toutefois, ce pipeline peut coexister avec le pipeline entier (instructions UAL, chargements/rangements, sauts et branchements) sans problème particulier, car les ressources sont disjointes : les opérations flottantes utilisent des opérateurs spécifiques (additionneur, multiplieur, éventuellement diviseur flottants), et un banc de registres flottants distinct du banc de registres entiers. L'exception est la multiplication entière, qui peut être effectuée dans le même multiplieur que la multiplication flottante.

Avec des instructions dont la phase d'exécution dure plusieurs cycles, le problème des dépendances de donnée est plus complexe, et peut conduire à un nouveau type d'aléa, l'aléa *écriture après écriture* (*WAW*, *Write after Write*). La fig. 6.21 en donne un exemple : la dernière valeur écrite dans F1 doit être le résultat de l'instruction LDD, et non celle de FMUL.

Le traitement le plus simple de ce type de dépendance est identique à celui des dépendances RAW : bloquer l'instruction dépendante jusqu'à ce que le résultat attendu ait été écrit dans le banc de registres. Il s'agit alors d'un contrôle centralisé, pour lequel un algorithme classique, l'algorithme de Tomasulo [7], a été défini à partir de tables de réservation. On verra plus loin des techniques plus avancées.

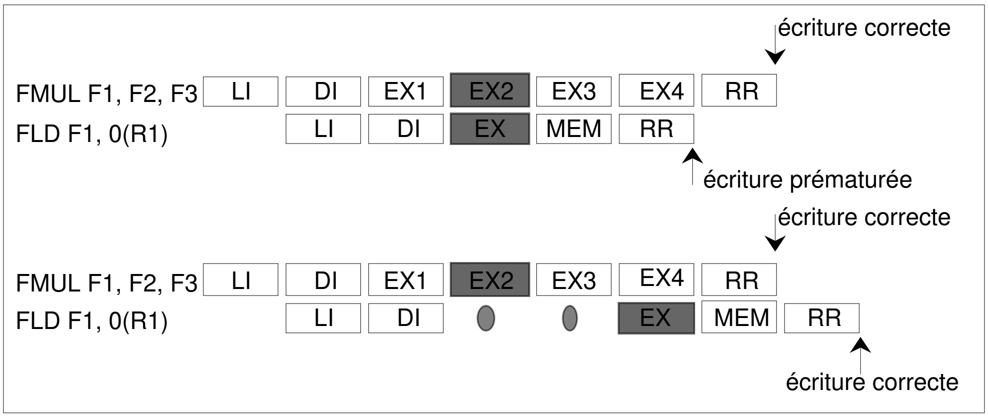


Figure 6.21: Aléa WaW avec instructions flottantes

6.9 Architectures superscalaires

Le principe du pipeline est d'exécuter simultanément les différentes phases d'une instruction. Le débit en instructions par cycle est au mieux 1. Le temps de cycle est déterminé par la durée commune de chaque étage, qui est elle même fixée par le temps de traversée du circuit combinatoire le plus long. La performance de l'architecture est alors bornée par la technologie de la circuiterie. Pour obtenir un débit supérieur à une instruction par cycle, il faut modifier l'architecture. Deux solutions sont possibles : le *superpipeline* et le *superscalaire*.

Une architecture superpipelinée fractionne chaque étage du pipeline, donc augmente le nombre d'étages du pipeline. Cette technique est largement employée, en particulier pour gérer la complexité croissante du contrôle et des accès mémoire. Elle a cependant des limites technologiques. Même s'il est possible de pipeliner l'UAL, par exemple 2 cycles, l'évolution de la circuiterie tend à diminuer proportionnellement plus les temps de propagation à l'intérieur de l'UAL que ceux liés aux registres ; par exemple, dans l'UltraSparcIII à 600 MHz, le surcoût registre est de l'ordre de 30% du temps de cycle). Le surcoût introduit par le tamponnement indispensable entre deux étages de pipeline tend à annuler les gains d'un superpipeline trop fin.

Le principe d'une architecture superscalaire est d'exécuter simultanément plusieurs instructions, en pipelinant les blocs d'instructions (fig. 6.22). Le

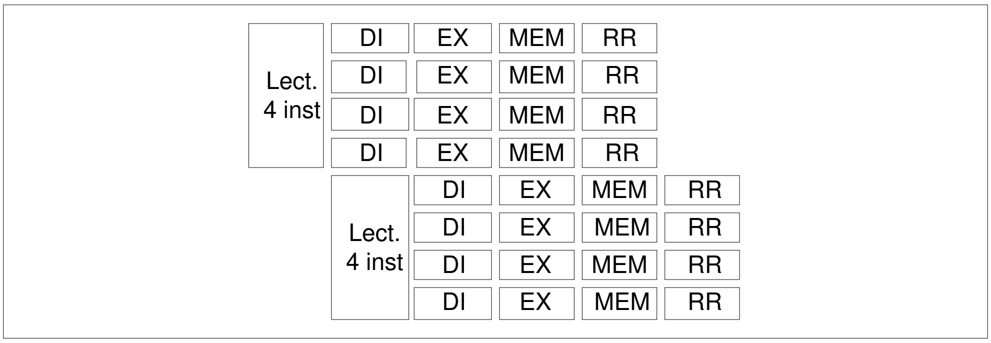


Figure 6.22: Exécution superscalaire de degré 4

débit d'une architecture superscalaire de degré n (blocs de n instructions) est n instructions par cycle, le temps de cycle restant borné par la technologie. Typiquement, en 99, le degré de superscalaire est 4 ou 8. L'efficacité des architectures superscalaires est conditionné par deux facteurs.

- Le matériel : les ressources nécessaires à l'exécution simultanée de n instances du même étage doivent pouvoir être intégrées dans le microprocesseur.

- L'application : n instructions ne peuvent s'exécuter simultanément que si elles sont indépendantes. On mesure cette propriété par l'ILP (*Instruction Level Parallelism*), qui est le nombre moyen d'instructions susceptibles de s'exécuter simultanément pour une application.

L'exploitation efficace de l'architecture superscalaire demande de modifier profondément l'ensemble des techniques matérielles et logicielles de traitement des dépendances. Globalement, l'architecture se composera de plusieurs blocs fonctionnels, chacun d'entre eux participant à ce traitement. La fig. 6.23 donne l'exemple de l'organisation architecturale de l'UltraSparcIII.

6.9.1 Dépendances de ressources

Le premier problème évident est la nécessité d'acquérir simultanément n instructions. Il est résolu par un cache permettant la lecture simultanée de plusieurs mots à chaque cycle, et un bus large entre le cache et l'unité d'instructions. L'augmentation de la taille des caches tend par ailleurs à demander plus d'un cycle pour effectuer un accès complet. En revanche, cet accès peut naturellement se diviser en deux étages de pipeline.

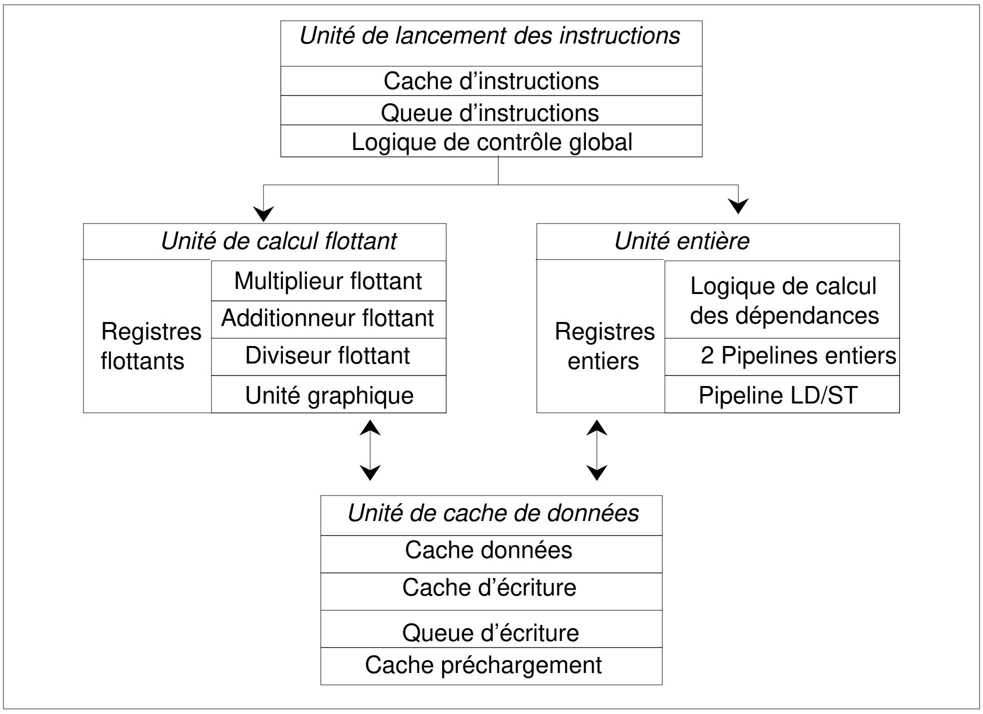


Figure 6.23: Architecture de l'UltraSparcIII

Le nombre d'additionneurs, UAL, etc. dépend du degré de superscalaire, et détermine la composition admissible du groupe d'instructions s'exécutant en parallèle. Par exemple, l'UltraSparcIII autorise

- 4 instructions entières dont
  - 2 instructions UAL,
  - 1 LD ou ST,
  - 1 branchement ;
- 2 instructions flottantes,
- 2 instructions graphiques.

6.9.2 Dépendances de données

Comme dans les architectures simplement pipelinées, les dépendances de données sont gérées en matériel. Le contrôle doit alors traquer les dépendances à l'intérieur d'un bloc d'instructions et entre les instructions de blocs différents. La complexité du contrôle croît donc quadratiquement en fonction de n. L'exécution pipelinée stricte, en présence d'une dépendance de données, pénalise gravement le superscalaire :

- l'anticipation n'est pas possible entre instructions du même bloc lorsque la dépendance porte sur le même étage ; par exemple, ADD R1 R2 R3 puis ADD R4 R1 R2.
- entre instructions de deux blocs consécutifs pour lesquels aucune anticipation n'est possible (LD R1 (R2) dans le bloc i, ADD R3 R3 R1 dans le second bloc $i+1$), l'élimination de la suspension par déplacement de code demande au compilateur d'insérer un bloc complet de n instructions indépendantes de celles du bloc i entre les deux blocs.

Exécution dans le désordre

Pour limiter les conséquences des dépendance de données, les instructions qui suivent une instruction suspendue, mais qui ne souffrent pas de la dépendance, sont autorisées à progresser.

Avec l'exécution dans le désordre, le cycle où le résultat d'une instruction devient disponible ne peut plus être déterminé simplement en matériel lors du décodage de l'instruction. Les algorithmes de contrôle centralisés évoqués en 6.8 deviennent donc inapplicables. Le contrôle doit être décentralisé au niveau de chaque unité fonctionnelle (fig. 6.24). Pour cela, le pipeline s'exécute dans l'ordre jusqu'à une étape, appelée *lancement de l'instruction*, qui peut être l'étage DI ou un étage inséré entre DI et EX. Ensuite, les instructions sont envoyés chacune dans une file d'attente associée à l'unité fonctionnelle qui la concerne (FADD vers l'additionneur flottant, FMUL vers le multiplieur flottant, etc.). Une logique de contôle associée à la file d'attente scrute le bus de résulat, sur lequel transistent les résulats étiquetés (taggés) par le numéro de registre destination. Lorsque le résultat attendu passe sur le bus, sa valeur est enregistrée dans les champs opérandes de l'instruction en attente, et utilisé lorsque l'instruction accède à l'opérateur. L'ensemble formé par la file d'attente et le contrôle associé est appelé une *station de réservation.*

Renommage de registres

Une conséquence de l'exécution dans le désordre est de rendre beaucoup plus fréquents les aléas WAW : une instruction bloquée peut écrire son reg-

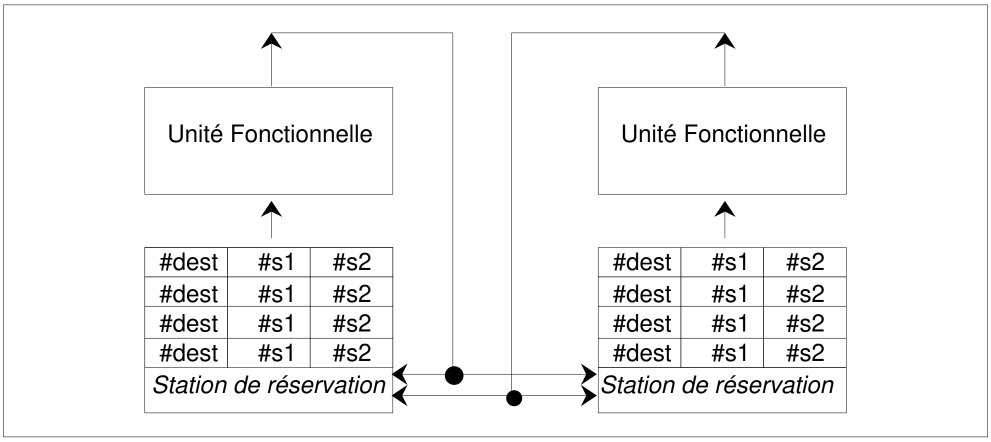


Figure 6.24: Principe des stations de réservation

istre destination après une instruction qui la précédait syntaxiquement (aléa *WAW*, Write After Read). Il est bien sûr possible d'introduire des suspensions pour gérer ces nouveaux aléas : une instruction qui doit écrire le même registre qu'une instruction bloquée est bloquée à son tour. Le recours systématique à cette solution anulerait rapidement les gains liés à l'exécution dans le désordre. La solution généralement retenue est le *renommage de registres.*

Les registres physiques réels sont plus nombreux que les registres logiques. Lorsque le contrôle détecte une dépendance WAW, l'écriture s'effectue dans deux registres physiques différents. L'écriture finale dans les banc de registres qui représente le banc de registres logiques est contrainte à s'effectuer dans l'ordre syntaxique du programme. Ceci conduit à introduire un étage supplémentaire dans le pipeline, correspondant à l'écriture finale, et appelée *complétion* (*complete*).

6.9.3 Aléas de contrôle

Les aléas de contrôle sont également beaucoup plus pénalisants pour les architectures superscalaires : il est très difficile de remplir le délai de branchement lorsqu'il doit être formé non plus d'une instruction, mais de quatre ou huit. En outre, pour les branchements conditionnels avec annulation, une mauvaise prédiction conduit à annuler n instructions si la prédiction du compilateur n'est pas exacte. Enfin, le compilateur ne dispose pas d'assez

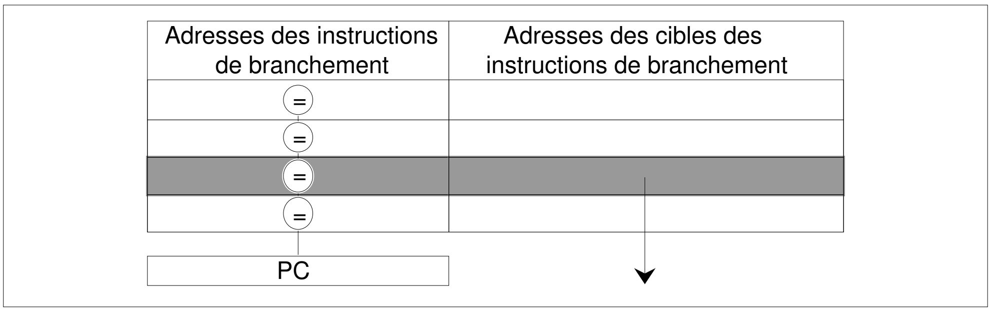


Figure 6.25: Principe du cache de branchements

d'informations pour prédire efficacement le comportement des branchements conditionnels. Les principales techniques développées pour le superscalaire sont les caches d'adresse de branchement, qui suppriment le délai de branchement, et la prédiction dynamique de branchement, qui améliore les performances de prédiction.

Caches d'adresses de branchement

Considérons le cas d'un branchement inconditionnel, ou d'un branchement conditionnel prédit pris. Le délai de branchement est dû au décodage de l'instruction combiné avec le calcul de l'adresse cible, l'ensemble s'effectuant au mieux dans l'étage DI.

La technique des caches d'adresses de branchement consiste à mémoriser les adresses cibles de saut ou branchement correspondant aux derniers sauts ou branchements effectués. Pour chaque instruction, l'adresse de l'instruction de branchement est comparée aux adresses dans le cache de branchement (fig. 6.25) pendant l'étage LI. Si l'adresse est présente, cela signifie qu'il s'agit d'une instruction de rupture de séquence, et l'adresse cible de la prochaine instruction à exécuter est disponible pour le cycle suivant. On obtient ainsi un délai de branchement nul ; cependant, une instruction de branchement inconditionnel ou prédit pris conduit à annuler les instructions suivantes dans le bloc contenant l'instruction de branchement.

Tampon de prédiction de branchement

La prédiction dynamique de branchement fait découler le comportement attendu du branchement de son comportement lors d'une exécution antérieure des instructions de branchement. Dans le cas le plus simple, on trouve un prédicteur 1 bit. Une table, réalisée en matériel contient un bit par entrée. Lorsqu'une instruction de branchement s'exécute, les poids faibles de l'adresse de branchement servent à indexer la table, et le bit à prédire le résultat du branchement. Il n'y a donc pas besoin d'attendre le résultat du calcul de la condition. Un branchement pris positionne le prédicteur à *pris*, le branchement suivant sera prédit pris et l'adresse de branchement prélevée dans le cache d'adresse de branchements ; un branchement non pris positionne le prédicteur à *non pris*, le prochain branchement sera prédit non pris et l'adresse du bloc d'instruction suivante est prise en séquence. Si la prédiction est erronée, le bloc d'instruction qui a commencé à s'exécuter est annulé.

La qualité de la prédiction peut être améliorée de deux façons. D'abord, l'indexation par les bits de poids faibles ne vérifie pas qu'il s'agit du "bon" branchement ; en augmentant la taille de la table, la probabilité de cet évènement diminue. On a montré qu'un résultat quasi-optimal est obtenu avec des tables de l'ordre de 4K entrées. D'autre part, on peut nuancer le comportement du prédicteur. La fig. 6.26 montre comment fonctionne un prédicteur 2 bits. Le prédicteur a quatre états. S'il se trouve dans l'un des états faiblement ou fortement pris, le branchement est prédit pris. Le résultat réel de la condition, donc du branchement, détermine les transitions de l'automate. Des études exhaustives ont montré qu'un prédicteur à plus de 2 bits n'améliorait pas significativement la qualité de prédiction.

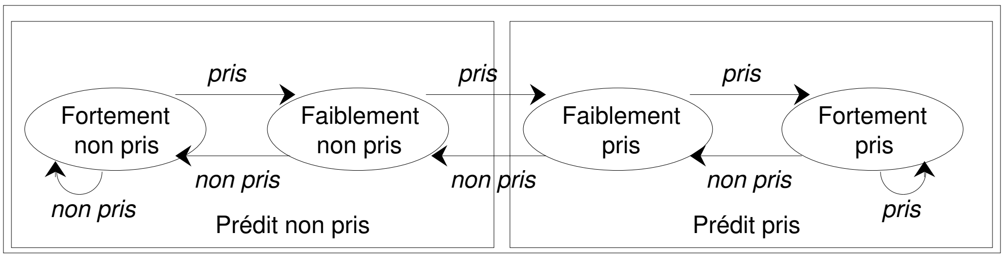


Figure 6.26: Prédicteur 2 bits

Les pénalités peuvent également être diminuées en demandant plus au cache d'instructions : certaines architectures effectuent simultanément la lecture des instructions cibles et de instructions en séquence.

Instructions prédiquées

Les instructions de rupture de séquence introduisent souvent une pénalité plus ou moins importante. D'autre part, elles limitent la taille du code que le compilateur peut réordonner : en pratique, les compilateurs travaillent sur les *blocs de base*, c'est à dire les fragments de code source compris entre deux instructions de rupture de séquence (corps de boucle, chaque branche d'une conditionnelle). Or, l'importance des optimisations de compilation est encore plus grande pour les architectures superscalaires que pour les architectures simplement pipelinées : le compilateur va réorganiser le code pour présenter des blocs d'instructions compatibles et sans dépendances.

Les instructions prédiquées introduisent la condition à l'intérieur de l'intruction. Par exemple, dans l'architecture Alpha, figure une instruction MOVxx. xx est une condition booléenne standard, qui porte sur le registre. Si la condition est vraie, l'instruction est exécutée, sinon elle est équivalente à un NOP. La pénalité est alors au maximum d'une instruction, et surtout, le code ne contient plus de branchements. Par exemple, le calcul dans R3 du minimum de deux variables contenues dans les registres R1 et R2 s'écrira

```
MOVA  R3, R1
CMP   R1, R2
MOVL  R3, R2
```

L'architecture IA-64 d'Intel a réalisé ce qui était proposé depuis une dizaine d'année par de nombreux travaux de recherche en architecture : un jeu d'instructions entièrement prédiqué.

6.10 Conclusion

L'évolution exponentielle de la technonologie des circuits intégrés modifie exponentiellement vite les conditions matérielles de la réalisation d'une architecture logicielle. Dans les années 80, l'essentiel de la complexité de la conception d'un microprocesseur était concentrée, hors les aspects proprement microélectroniques, dans la microprogrammation de la partie contrôle, donc dans des optimisations fines et locales. Actuellement, l'architecte doit organiser en une structure efficace un nombre considérable de sous-structures elles mêmes très complexes.

Chapitre 7

Hiérarchie mémoire : la mémoire centrale

7.1 Introduction

7.1.1 Motivation

La hiérachie mémoire est l'implémentation par des réalisations matérielles distinctes de l'espace d'adressage unique vu par l'architecture logicielle (fig. 7.1).

Les niveaux de la hiérarchie mémoire se différencient par plusieurs caractéristiques.

- Technologie de réalisation : semiconducteurs pour cache et mémoire principale ; magnétique ou optique pour les mémoires secondaire.
- Vitesse : temps d'accès croissant des caches vers les disques.
- Coût par bit : décroissant des caches vers les disques.

La table 7.1 donne quelques données chiffrées sur ces différents niveaux. Les prix des mémoires évoluent très brutalement ; les valeurs indiquées sont seulement des ordres de grandeur. La SRAM est la seule à suivre les performances du processeur ; mais réaliser en SRAM les 4GO d'espace adressable coûterait de 400K$ à 4M$. Cette solution n'est évidemment pas celle du marché de masse (de tels ordinateurs existent, pour le calcul numérique intensif). Même la DRAM est encore trop chère : les 4GO coûteraient 8000$.

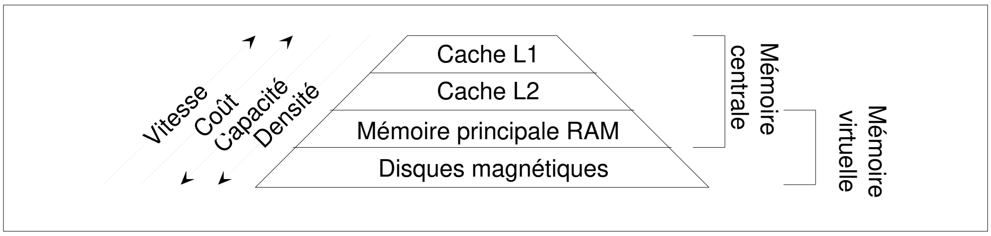


Figure 7.1: La hiérarchie mémoire

Type de mémoire	Temps d'accès typique	Prix par MOctet en $
SRAM	2 - 5 ns	100 - 1000
DRAM	10 ns	2
Disque magnétique	10 - 20 ms	0,01 - 0,1

Table 7.1: Caractéristiques de la hiérarchie mémoire en 99

La hiérarchie mémoire comporte logiquement deux niveaux fondamentaux :

- la mémoire centrale : les caches et la mémoire principale ;
- la mémoire virtuelle : la mémoire principale et les mémoires secondaires (disques)

Conceptuellement, chaque niveau de la hiérachie contient une copie d'un sous-ensemble du niveau supérieur, le niveau le plus bas étant le cache interne. L'accès à un niveau inférieur étant plus rapide que celui au niveau supérieur, l'accès à la copie offre ainsi l'illusion d'une mémoire rapide, à moindre coût.

Dans les deux cas, la gestion des copies entre niveaux doit être transparente à l'utilisateur final. Comme les temps d'accès au cache sont de l'ordre du temps de cycle, ce traitement est assuré complètement par le matériel. En revanche, pour la mémoire virtuelle, les temps d'accès aux disques autorisent une combinaison de traitement matériel et logiciel.

7.1.2 Le principe de localité

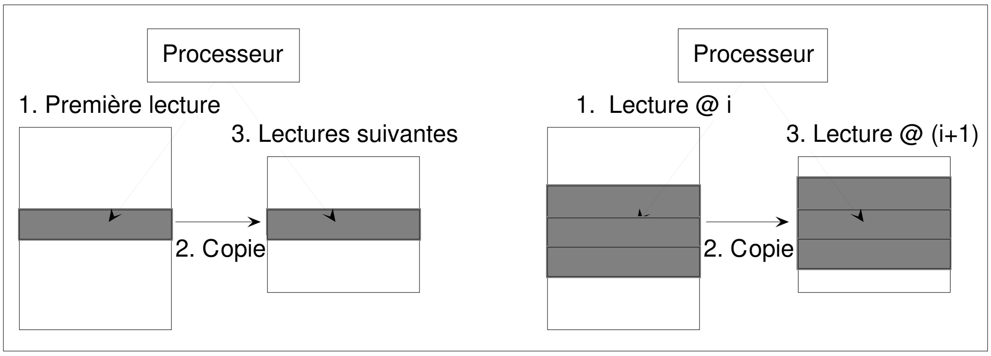


Figure 7.2: Localité temporelle et localité spatiale

La pertinence de l'approche hiérarchique n'est pas évidente. Elle fonctionne à cause d'une propriété fréquente, mais pas universelle, des programmes : la *localité*. On distingue deux types de localités : la localité *temporelle* et la localité *spatiale* (fig. 7.2).

La localité temporelle est liée au comportement dans le temps d'un programme. Lorsqu'une référence mémoire est effectuée, il y a de grandes chances qu'elle soit à nouveau effectuée très bientôt ; en effet, les programmes d'une durée significative sont constitués de boucles, et de parties annexes qui traitent les cas exceptionnels. De même, les données sont structurées et un programme accède souvent aux mêmes données. Si le mot mémoire référencé est copié depuis le niveau $n+1$ vers le niveau n lors du premier accès, il sera disponible avec les performances du niveau n pour les accès suivants.

La localité spatiale est une propriété de l'organisation des données ou des instructions en mémoire. Un accès mémoire rend probable un prochain accès aux adresses contigues : typiquement, la lecture d'une instruction à l'adresse n rend probable la lecture de l'instruction suivante en séquence, à l'adresse $n + 4$; pour les données, les adresses sont proches, en particulier dans le cas de parcours régulier d'un tableau.

La localité spatiale est exploitable d'abord à cause d'une caractéristique technologique commune aux DRAM et aux disques. La lecture d'un bloc de l mots à des adresses consécutives est beaucoup moins coûteuse en temps que p lectures indépendantes. Les causes sont différentes pour les disques et pour les DRAM, mais dans les deux cas, le temps nécessaire à la lecture de l mots est de la forme :

$$T(l) = t_s + lt_m,$$

où t_s est un temps de démarrage et t_m le temps unitaire d'accès à un mot. L'accès au premier mot d'un bloc coûtera donc le temps de lecture du bloc, pour effectuer sa copie du niveau $n+1$ vers le niveau n ; mais les autres mots du même bloc seront disponibles avec les performances du niveau n pour les accès suivants. La localité spatiale est aussi exploitable à cause des organisations du type bus large, qu'on verra en 7.6.

Définition 1 *Les transferts entre niveaux de la hiérachie mémoire s'effectuent donc toujours par* blocs d'adresses consécutives*. Au niveau de la mémoire centrale (caches et DRAM), l'unité de transfert est appelée* ligne *; au niveau de la mémoire virtuelle, l'unité de transfert est la* page*.*

Il n'est pas possible de poursuivre une étude commune de la mémoire centrale et de la mémoire virtuelle : les contraintes technologiques sont trop différentes. La suite de ce chapitre traitera donc uniquement de la mémoire centrale, et le chapitre suivant de la mémoire virtuelle.

7.1.3 Un exemple: le produit matrice-vecteur

L'algorithme le plus simple pour le produit matrice-vecteur est rappelé fig. 7.3. L'algorithme est écrit en C, donc les matrices sont rangées en mémoire lignes d'abord (RMO). L'accès à A et à Y au membre droit ($Y[j]$) offre donc une parfaite localité spatiale : les blocs successifs sont complètement utilisés. L'accès à Y en membre gauche offre également de la localité temporelle : l'accès à $Y[i]$ est répété au cours de la boucle j. En fait, un compilateur optimisant exploitera cette localité en effectuant l'accumulation dans un registre.

En Fortran, le code équivalent, c'est à dire la boucle i englobant la boucle j n'offrirait pas de localité spatiale : en effet, la convention pour les compilateurs Fortran est le placement colonnes d'abord (CMO). Les accès successifs à $A(i,j), A(i,j+1), \ldots A(i,N)$ correspondent donc à un pas mémoire ($N \times$ Taille de l'élément de tableau), ce qui en général beaucoup plus grand que la taille d'une ligne de cache. Chaque accès coûte donc le temps d'un accès au niveau supérieur (cache L2 ou DRAM). Dans ce cas particulier, la solution est simple, il suffit d'intervertir l'ordre des boucles (fig. 7.3(b)).

7.2 Fonctionnement du cache

Dans ce qui suit, on considère seulement deux niveaux, un cache et une mémoire principale ; les mêmes mécanismes réalisent le fonctionnement d'un

(a)

```
for (i = 0 ; i < N ; i++)
    for (j = 0 ; i < M ; j++)
        Y[i] = Y[i] + A[i][j]*Y[j]
```

(b)

```
do j = 1, M
   do i = 1, N
      Y(i) = Y(i) + A(i,j)*Y(j)
   end do
   end do
```

Figure 7.3: Produit matrice-vecteur naïf adapté à la hiérarchie mémoire (a) en C, (b) en Fortran

ensemble formé d'un cache L1 et d'un cache L2.

La hiérarchie mémoire cache-MP est un mécanisme transparent pour l'UC : elle envoie une adresse et reçoit une instruction ou une donnée. Le cache contient un sous-ensemble strict du contenu de la mémoire principale. Le fonctionnement du cache doit donc assurer d'une part la *correction* des accès, d'autre part la *cohérence* entre cache et mémoire principale.

7.2.1 Succès et échec cache

Le problème de correction consiste à assurer l'accès à l'adresse mémoire requise, qu'elle soit ou non présente dans le cache. Chaque entrée du cache contient essentiellement deux champs, adresse et donnée (fig. 7.4), plus des bits de contrôle. Le champ adresse contient le numéro de la ligne mémoire qui est actuellement présente dans cette entrée du cache. Le champ données contient la copie de cette ligne mémoire. Lors d'un accès mémoire, l'adresse requise est comparée à toutes ou certaines adresses des champs adresse du cache ; le détail de la correspondance est étudié section 7.2.

- Si l'adresse est trouvée dans le cache, il y a *succès* (hit).
- Si l'adresse n'est pas trouvée, il y a *échec*, ou *défaut* (miss), et il faut accéder à la mémoire principale.

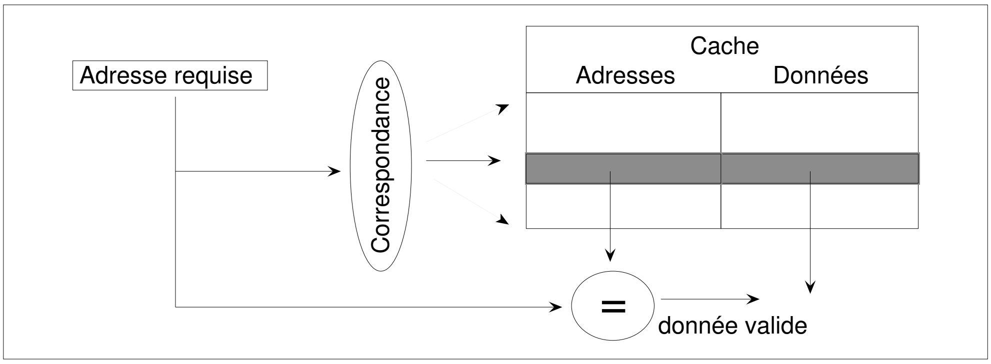


Figure 7.4: Principe de fonctionnement d'un cache

Important 1 *L'accès cache se traouve sur le chemin critique pour l'exécution des instructions.* Toute *lecture d'instruction accède au cache, pour savoir si l'instruction s'y trouve ;* toute *instruction d'accès mémoire accède au cache, pour tenter d'y trouver la donnée référencée. Autrement dit, l'accès* réussi *au cache doit tenir dans l'étage MEM du pipeline, au moins pour le cache L1. Ceci définit les contraintes temporelles pesant sur la réalisation du mécanisme de recherche.*

Le coût en temps du traitement d'un défaut de cache (*miss penalty*) est noté $T_{\text{défaut}}$ Le temps d'accès moyen à la mémoire est donc :

$$T_{\text{accès}} = T_{\text{accès cache}} + \text{taux d'échec} \times T_{\text{défaut}}$$

Cette analyse suppose que le processeur n'effectue aucun travail utile pendant le traitement du défaut. On verra en 7.6.2 qu'on peut faire mieux.

En pratique, comme la taille de la ligne est plusieurs fois celle du champ adresse (une ligne contient plus d'un mot), les champs adresses sont matériellement contenus dans une mémoire rapide qui a autant d'entrées que de lignes de cache.

7.2.2 Traitement des échecs en lecture

En cas d'échec en lecture, à cause du principe de localité, la ligne de cache qui contient le nouveau mot est copiée (*chargée*) dans le cache. Comme la mémoire principale est beaucoup plus grande que le cache, il est fréquent que la ligne nouvellement acquise vienne occuper une ligne du cache précédemment utilisée. On dit alors qu'il y a *conflit*, bien que celui-ci soit toujours résolu de la même façon : la nouvelle ligne écrase l'ancienne.

Le coût en temps du traitement d'un défaut de cache en lecture comprend donc le temps d'accès à la mémoire principale, plus le temps de transfert entre la mémoire principale et le cache, qui est proportionnel à la taille de la ligne de cache.

7.2.3 Les écritures

Il existe deux stratégies de traitement d'un succès en écriture : *l'écriture simultanée* (*write through*) et la *réécriture* (*write back*).

L'écriture simultanée écrit le mot dans le cache **et** dans la MP. L'écriture simultanée présente l'avantage d'assurer la cohérence maximale entre les contenus du cache et de la MP. En revanche, il peut il y avoir de nombreuses écritures inutiles en MP, lorsque des cases mémoire sont modifiées à chaque passage dans une boucle.

La réécriture consiste à écrire le mot uniquement dans le cache, en indiquant par un bit spécial (*dirty bit*) dans le cache que la ligne ayant été modifiée, son contenu est différent de celui du bloc correspondant de la MP. Lorsqu'une ligne doit être enlevée du cache (pour être remplacée par un autre), on regarde si elle a été modifiée. Si oui, on la recopie en MP. Sinon, la copie non modifiée est déjà en MP. L'avantage est de factoriser le coût des écriture en MP. Il y a deux inconvénients. D'une part, la pénalité d'échec est doublée. D'autre part, le contenu de la mémoire principale et celui du cache ne sont plus cohérents à chaque instant : le cache peut contenir une version plus récente que celle de la mémoire principale.

Il existe également deux stratégies de traitement d'un échec en écriture : l'écriture *allouée* (*write allocate*) consiste à d'abord charger le bloc manquant dans le cache, avant d'effectuer l'écriture. L'écriture *non allouée* (*no write allocate*) écrit uniquement dans la mémoire principale. En effet, la localité est nettement moins sensible pour les écritures, l'allocation n'est donc pas toujours la meilleure stratégie. Toutes les combinaisons write through / write back avec write allocate / no write allocate sont possibles. En général, l'écriture non allouée est plutôt associée avec l'écriture simultanée (on parie que l'écriture est un accès isolé) et l'écriture allouée avec la réécriture (on parie sur la localité). Dans les processeurs les plus récents, les stratégies en écriture sont paramétrables ; comme ce choix a des conséquences au niveau de l'architecture de l'ensemble de l'ordinateur, ce paramétrage fait partie des couches les plus basses du système et n'est en aucun cas accessible à l'utilisateur final.

7.2.4 En résumé

Les tables 7.2 et 7.3 résument les actions entreprises lors d'un succès ou d'un échec, suivant la stratégie adoptée pour un accès en écriture. Lors d'un succès en lecture, on lit simplement le mot dans le cache. Lors d'un échec, il faut au minimum transférer la ligne correspondante de la mémoire principale vers le cache et effectuer la lecture dans le cache.

	Lecture	Ecriture
Succès	Lire le mot dans le cache	Ecrire le mot dans le cache Ecrire le mot dans la MP
Echec	Charger la ligne Lire le mot dans le cache	Ecrire le mot en MP

Table 7.2: Cache en écriture simultanée avec écriture non allouée

	Lecture	Ecriture
Succès	Lire le mot dans le cache	Ecrire le mot dans le cache
Echec	Décharger la ligne si modifiée Charger la ligne Ecrire le mot dans le cache	Décharger la ligne si modifiée Charger la ligne Lire le mot dans le cache

Table 7.3: Cache en réécriture avec mémoire allouée

Le problème de cohérence est lié au fait que le cache et la MP peuvent contenir deux versions différentes pour la même adresse mémoire : le cache peut contenir une version plus récente dans le cas de la réécriture ; la cohérence est alors assurée par le microprocesseur, via le mécanisme de déchargement vu plus haut. Mais la MP peut aussi contenir une version plus récente, par exemple si un contrôleur d'E/S dans une architecture monoprocesseur, ou un autre processeur dans une architecture multiprocesseurs, a modifié le contenu d'une ligne de la MP qui est aussi présente dans le cache. Tous les microprocesseurs actuels intègrent le matériel nécessaire

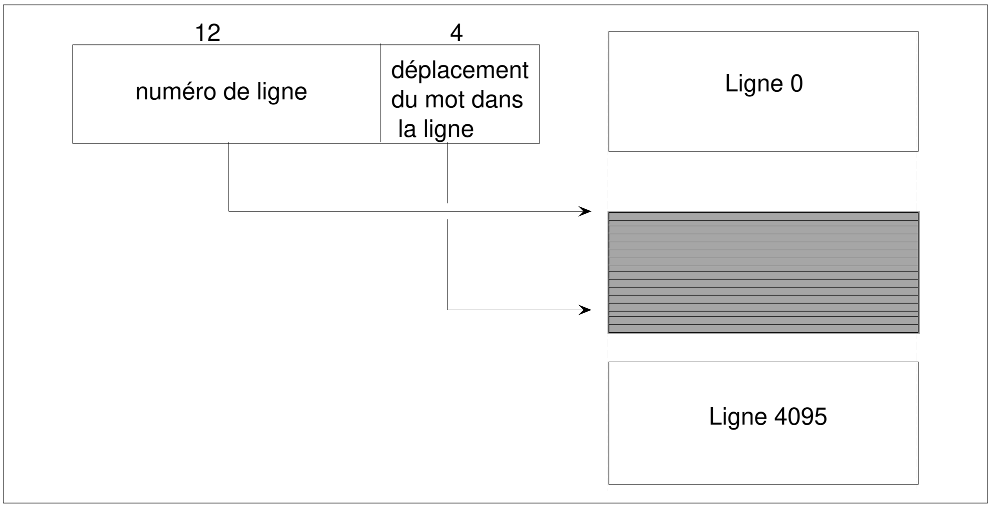


Figure 7.5: Décomposition d'une adresse mémoire

pour participer à une architecture multiprocesseurs, et les protocoles de bus qui assurent la cohérence par rapport aux E/S.

7.3 Correspondance entre le cache et la mémoire principale

Définition 2 *La correspondance entre les lignes du cache et celles de la MP définit à quel emplacement une ligne de la MP est chargée dans le cache. Le mécanisme de correspondance doit permettre de déterminer rapidement si une adresse mémoire correspond à une ligne présente dans le cache.*

Dans la suite, on note i le numéro d'une ligne de la mémoire principale, et N le nombre de lignes dans le cache. On utilisera l'exemple suivant : un cache de 2048 mots, et une mémoire principale de 64 K mots. La taille de la ligne est 16 mots. Le cache a donc $N = 128 = 2^7$ lignes, et la MP 4K $= 2^{12}$ lignes. La fig. 7.5 présente la décomposition d'une adresse : les 4 bits de poids faible donnent l'adresse du mot dans la ligne ; les 12 bits de poids fort donnent le numéro i de la ligne de la MP.

7.3.1 Correspondance directe

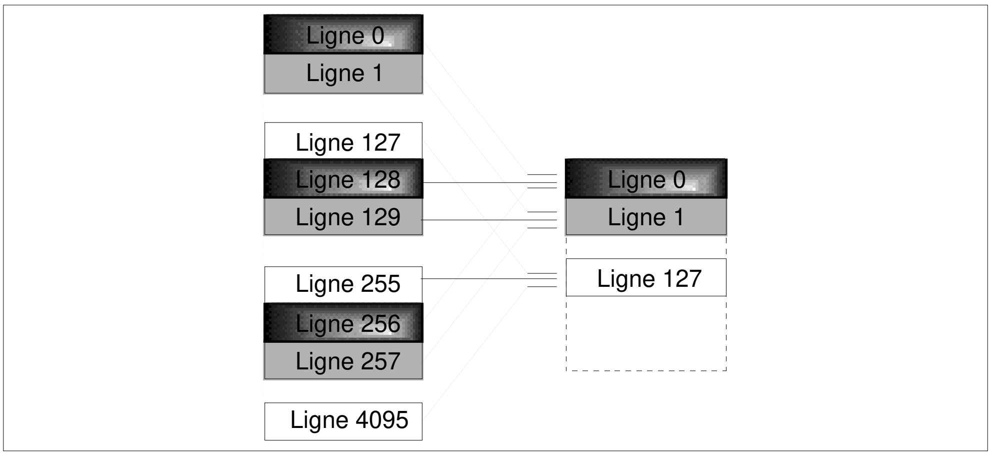


Figure 7.6: La correspondance directe

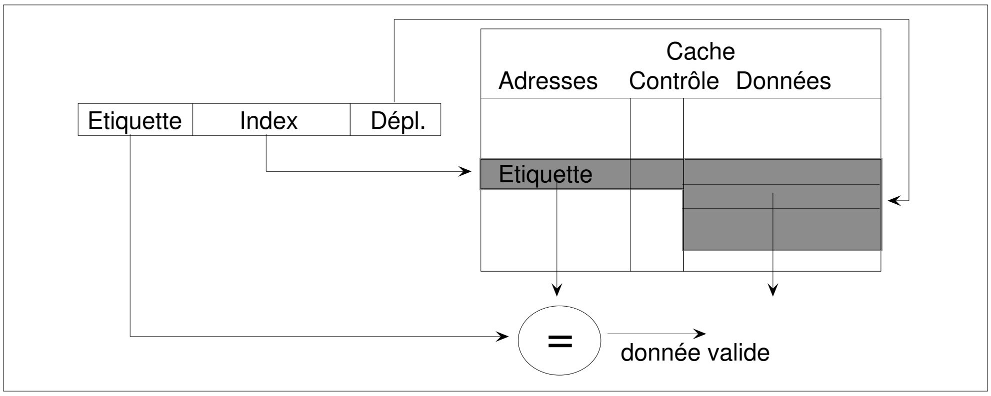


Figure 7.7: Cache à correspondance directe

Définition 3 *La correspondance directe associe à chaque ligne de la mémoire principale une unique ligne de cache : la ligne i de la MP va dans la ligne $j = i \bmod N$ du cache.*

Avec l'exemple, la ligne 0 du cache pourra contenir les lignes 0, 128, 256... de la MP ; la ligne 1 du cache pourra contenir les lignes 1, 129, 257... de la MP, etc. (fig. 7.6).

L'adresse mémoire se décompose alors en trois champs : *étiquette* e, *index* j et *déplacement* dans la ligne. On a :

$$i = e \times N + j.$$

Dans l'exemple, les 12 bits de i sont décomposés en 7 bits pour l'index et 5 pour l'étiquette.

Chaque entrée de la table contient l'étiquette e de la ligne actuellement présente (fig. 7.7). L'index est égal au numéro de la ligne du cache où doit être rangé la ligne de la MP. L'étiquette est comparée avec le champ étiquette de l'adresse requise. Il y a succès si les deux sont identiques. Une condition supplémentaire est que l'adresse soit significative : au démarrage du processeur, les contenus des champs étiquette et données sont complètement décorrélés, car aucun accès mémoire n'a encore eu lieu. Un *bit de validité* est positionné au premier chargement d'une ligne de cache.

La correspondance directe est rapide : une comparaison a un coût en temps quasi-négligeable. Mais elle n'exploite pas bien le cache : si le le programme n'accède qu'aux adresses des ligne 0 et 128, il se produira des collisions sur la ligne 0 du cache, alors que le reste du cache ne sert pas.

7.3.2 Correspondance associative

Pour exploiter complètement le cache, la solution évidente est la correspondance complètement associative : une ligne de la MP peut aller dans n'importe quelle ligne du cache.

L'étiquette d'une ligne de la MP est alors simplement son numéro : $e = i$; dans l'exemple, e est sur 12 bits.

Pour savoir si une adresse est présente dans le cache, il faut comparer l'étiquette avec toutes les étiquettes du cache. Le dispositif matériel, qui compare une valeur en entrée avec le contenu de toutes les valeurs d'une table, est en fait une mémoire associative à accès par le contenu, qui est beaucoup plus complexe qu'une mémoire normale à accès par adresse (fig. 7.8).

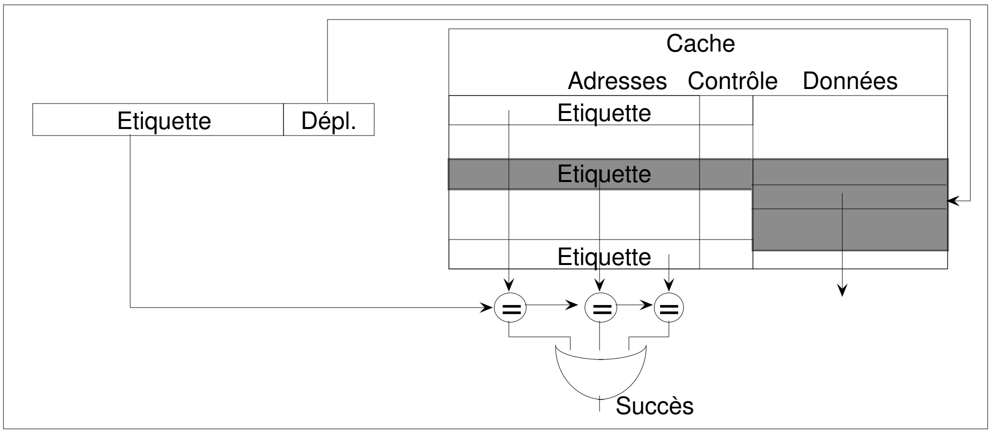


Figure 7.8: Cache complètement associatif

Cette méthode de correspondance n'est pas utilisée pour les caches contenant instructions et données. En effet, le temps de détection d'un succès ou d'un échec est approximativement logarithmique en la taille de la table, et devient donc prohibitif pour la taille des caches d'instructions ou de données. En revanche, la correspondance complètement associative est souvent utilisée pour des caches de petite taille, en particulier les les tampons de traduction d'adresses virtuelles en adresses réelles, comme on le verra au chapitre suivant.

7.3.3 Correspondance associative par ensemble

La correspondance associative par ensemble est intermédiaire entre les deux techniques précédentes. Les lignes du cache sont regroupées par ensembles, typiquement de 2 ou 4 lignes. Dans l'exemple, avec associativité par ensembles de 2 lignes (2-way), il y a 64 ensembles. La ligne i de la mémoire principale va dans une des lignes de l'ensemble $j = i \bmod E$, où E est le nombre d'ensembles. L'index est alors le numéro d'ensemble, sur 6 bits dans l'exemple. Il y a correspondance directe du point de vue ensembles, mais dans un ensemble donné, une ligne peut aller n'importe où. La fig. 7.9 illustre la correspondance avec deux lignes par ensemble, et la fig. 7.10 le mécanisme de détection.

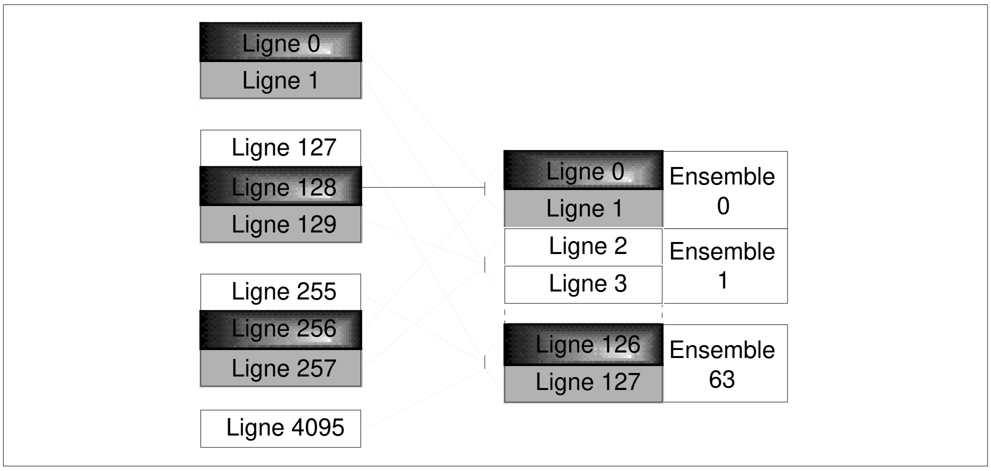


Figure 7.9: La correspondance associative par ensemble

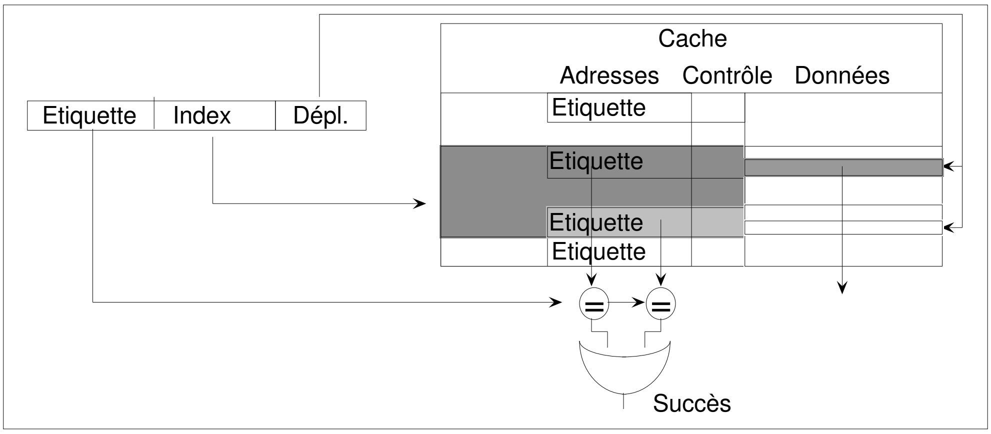


Figure 7.10: Cache associatif par ensemble

7.3.4 Algorithmes de remplacement

Avec la correspondance directe, il n'y a aucun choix lorsqu'il s'agit de remplacer une ligne ancienne du cache par une ligne nouvelle. Dans les autres techniques, plusieurs lignes sont candidates : pour des ensembles de 2 lignes, l'une quelconque des deux lignes peut être remplacée. Les trois algorithmes classiques sont :

- choisir au hasard
- remplacer la ligne la plus ancienne (Least Recently Used, LRU)
- remplacer la ligne la moins utilisée (Least Frequently Used)

Les deux dernières méthodes impliquent de gérer des compteurs pour déterminer l'ancienneté ou la fréquence d'utilisation des différents lignes du cache. Pour une associativité de degré supérieur à 2, l'algorithme le plus utilisé est le pseudo-LRU : un accès à une ligne positionne un bit d'accès récent pour la ligne. Pour un remplacement, la ligne victime est choisie au hasard parmi les autres lignes.

La table 7.4 donne quelques exemples d'organisation des caches. On notera que les caches L2 des processeurs Intel sont petits par rapport à ceux des UltraSparc. Il s'agit d'un choix spécifique d'Intel. Le cache L2 est réalisé encapsulé dans le même boîtier que le processeur. Pour le Pentium III, le cache L2 peut même être réalisé sur la même puce que le processeur, mais est alors limité à 256K.

7.4 Performances

Soit m_a le nombre moyen d'accès mémoire par instruction, m le taux d'échec cache, et p le nombre de cycles machines nécessaires pour traiter un défaut de cache. $T_{\text{défaut}} = p.T_c$ est le temps de traitement d'un défaut de cache. Le nombre de cycles machines par instruction CPI_m lié aux attentes mémoire que provoquent les défauts de cache, s'exprime sous la forme

$$\text{CPI}_m = m_a \times m \times p$$

Le temps d'exécution d'un programme, compte tenu des défauts de cache, s'écrit maintenant

$$T_e = \text{NI} \times (\text{CPI} + m_a.m.p)T_c.$$

La performance peut être améliorée :

Processeur	L1 Inst			L1 Données		
	taille	ligne	as.	taille	ligne	as.
Pentium II	16KO	32	4	16KO	32	4
Pentium III						
UltraSparc I et II	16KO	32	2	16KO	32	Dir.
UltraSparcIII	32 KO	32	4	64 KO	64	4

Processeur	L2		
	taille	ligne	as.
Pentium II	512		
Pentium III	256KO ou 512KO		8
UltraSparc I	512KO à 4MO		
UltraSparc II	512 KO à 16MO		
UltraSparcIII	4 à 16MO		

Table 7.4: Organisation des caches de quelques processeurs

- en diminuant le taux d'échec, soit en modifiant la structure du cache (matériel), soit en modifiant le programme (compilateur);
- en diminuant la pénalité d'échec, ce qui ne peut relever que du matériel.

Il est important de noter que la pénalité d'échec vue du processeur (en nombre de cycles machines) augmente lorsque le temps de cycle diminue. En effet, le temps de cycle processeur T_c dépend fondamentalement de la technologie utilisée pour le processeur et de l'organisation du pipeline des instructions, alors que le traitement de l'échec cache est corrélé au temps de cycle des mémoire DRAM T_m. Comme T_c diminue plus vite que T_m, il en résulte un coût croissant des échecs cache en cycles machine avec les progrès technologiques.

7.5 Performances : diminuer le taux d'échec

7.5.1 Origines des échecs

Les échecs cache peuvent résulter de trois causes :

- *démarrage* : aucune instruction ou donnée n'est présente dans le cache, et les premiers accès nécessitent les chargements de lignes depuis la mémoire principale.

- *capacité* : si le cache ne peut contenir tous les lignes nécessaires à l'exécution d'un programme, charger de nouvelles lignes impose de remplacer des lignes présentes dans le cache ; il y a échec de capacité si même un cache à correspondance complètement associative ne peut contenir toutes les données et instructions simultanément actives (*working set*) du programme.

- *conflit* : des lignes sont placées par le mécanisme de correspondance dans la même ligne du cache, ce qui provoque des remplacements et appels successifs, alors que d'autres lignes de cache étaient disponibles (mais non autorisées par le mécanisme de correspondance).

Considérons l'exemple suivant :

```
for(i = 0 ; i < n ; i++)
   s = s + X[i] + Y [i] ;
```

X et Y sont des tableaux de mots implantés respectivement à partir des adresses 0x1000 et 0x2000. La table 7.5 résume les défaut qui interviennent

	Corr. directe		2-assoc.	
itération	X	Y	X	Y
0	déf. dem.	déf. dem.	déf. dem.	déf. dem.
1 - 15	déf. conf.	déf. conf.	succès	succès
16	déf. dem.	déf. dem.	déf. dem.	déf. dem.

Table 7.5: Catégories de défauts de cache

pour l'exécution de ce programme. En correspondance directe (resp correspondance associative par ensemble de 2 lignes), X[0]-X[15] et Y[0]-Y[15] vont dans la ligne de cache (resp l'ensemble) 0, X[16]-X[31] et Y[16]-Y[31] vont dans la ligne de cache (resp l'ensemble) 1 etc. En correspondance directe, chaque accès à X chasse la ligne précédemment chargée pour Y et réciproquement. En correspondance associative, l'ensemble peut accueillir simultanément les deux lignes nécessaires à chaque itération.

7.5.2 Organisation du cache

Pour un programme donné, le taux d'échec est fonction de trois paramètres.

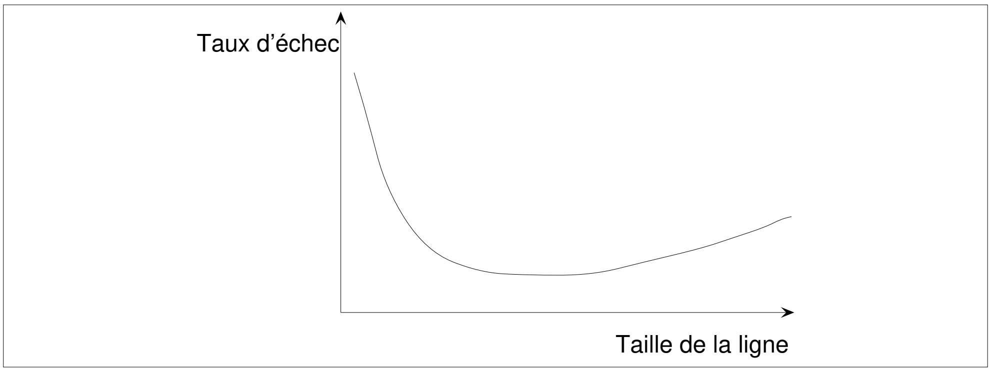


Figure 7.11: Taux d'échec en fonction de la taille de la ligne

- La taille du cache : elle détermine directement le nombre d'échecs de capacité ; elle est fonction de la technologie.
- La taille des lignes. La figure 7.11 montre l'évolution du taux d'échec en fonction de la taille des lignes pour une taille donnée de cache : pour des lignes très petites, il y a beaucoup d'échecs de démarrage ; puis le taux diminue avec la taille des lignes jusqu'au point où l'augmentation de la taille réduisant le nombre de lignes, les échecs de capacité et de conflit provoquent une remontée du taux d'échec.
- Le degré d'associativité : augmenter le nombre de lignes par ensemble réduit jusqu'à un certain point les échecs de conflit (fig. 7.5). Mais cette augmentation est limitée par la complication matérielle associée à la comparaison en parallèle des étiquettes de tous les lignes d'un ensemble.

Une règle empirique importante, appelée règle des 2:1, stipule qu'un cache à correspondance directe de taille N a environ le même taux d'échec qu'un cache de taille $N/2$ à correspondance associative par ensemble de deux lignes.

On trouvera dans [7] des statistiques détaillées sur les taux d'échec associés à ces trois causes en fonction de la taille du cache, de la taille des lignes et du degré d'associativité.

Pour limiter les effets des conflits sans pénaliser le temps de cycle, la technique de *cache des victimes* fournit un tampon contenant les lignes récemment rejetées, dans l'espoir qu'elles seront bientôt rechargées dans le cache.

7.5.3 Optimisations logicielles

On a vu en 7.1.3 qu'une modification assez mineure du programme de produit matrice-vecteur avait des conséquences majeures sur le taux d'échec. Ce comportement est particulièrement important pour le traitement numérique, qui travaille sur des structures de données matrice ou vecteurs, représentées par des tableaux.

Il est possible d'analyser à la compilation les accès aux tableaux, pour améliorer la localité d'un programme. Par exemple, la transformation du programme matrice-vecteur est connue sous le nom d'*échange de boucles*. Beaucoup d'autres techniques existent, en particulier le *blocage*. Le blocage consiste à réorganiser l'algorithme pour qu'il accède de façon répétitive à des blocs de données qui tiennent dans une ligne de cache. Le produit matrice-matrice en est l'exemple le plus simple.

Soient trois matrices $n \times n$ A, B et C. On veut effectuer $C = AB$. L'algorithme naïf, en C, est :

```
for (i=0 ; i<n ; i++)
    for (j=0 ; j<n ; j++)
        for (k=0 ; k<n ; k++)
            C[i][j] = C[i][j] + A[i][k]*B[k][j];
```

Dans l'analyse qui suit, on néglige les accès à la matrice C, qui sont typiquement en registre, et on se place dans le cas idéal d'un cache totalement associatif. Le produit matrice-vecteur demande $2n^3$ opérations et $2n^3$ accès aux données, mais seulement $2n^2$ accès mémoire distincts ; si le cache pouvait contenir les matrices A et B, il n'y aurait que des échecs de démarrage ; on n'effectuerait donc qu'une lecture en mémoire principale par donnée, soit un taux d'échec égal à $2n^2/2n^3 = 1/n$. L'*application* présente donc un bon potentiel de localité. Cependant, pour une application de taille significative, l'hypothèse que la taille du cache soit de l'ordre de n^2 n'est pas réaliste.

L'unité étant l'élément de tableau, soient c la taille du cache et l la taille de la ligne de cache ; on suppose que $c \approx 2n$, et $c << n^2$. Dans l'algorithme précédent, supposons i fixé.

– accès à B : 1 échec de démarrage à chaque accès, car on accède à B par colonnes.

- accès à A : 1 échec tout les l accès ; en effet, pour i fixé, il faut d'abord charger une première fois la ligne i de A, ce qui intervient à l'itération 0 en j. Mais aux itérations suivantes en j, les échecs sur B auront rempli le cache, donc éliminé les éléments de A déjà chargés. Il s'agit d'échecs de capacité, la taille du cache étant $2n$.

L'algorithme effectue n^3 accès à A, donc n^3/l défauts, et n^3 accès à B, donc n^3 défauts. Le taux d'échec est donc

$$m = \frac{1}{2}(1 + \frac{1}{l})$$

Donc, le taux d'échec est au moins 1/2, et n'est améliorable que marginalement par l'agrandissement de la ligne de cache.

Dans l'algorithme bloqué, les matrices sont découpées en blocs de taille b. Soit $p = n/b$. L'algorithme effectue simplement le produit de matrices par blocs :

```
for (I=0 ; I<p ; I++)
    for (J=0 ; J<p ; J++)
        for (K=0 ; K<p ; K++)
            matmuladd (C, A, B, I, J, K);
```

La procédure matmuladd (C, A, B, I, J, K) effectue le produit du bloc (I, K) de A par le bloc (K, J) de B et le cumule avec le bloc $C(I, J)$ de C.

On suppose que le cache peut contenir simultanément deux sous-matrices, ie. $c = 2b^2$, et $b \geq l$.

- accès à B : 1 échec de démarrage tout les l accès. Par exemple, l'accès à *B[0][0]* charge la ligne de cache contenant *B[0][1]*, ..., *B[0][l-1]* ; les accès suivants sont *B[1][0]* , ..., *B[b-1][0]*, qui chargent chacun la ligne de cache correspondante. Ensuite, les données chargées pourront être réutilisées.

- accès à A : 1 échec de démarrage tout les l accès, avec un fonctionnement analogue.

Le taux d'échec est alors

$$m = \frac{1}{l}.$$

Agrandir la ligne de cache diminue donc proportionnellement le taux d'échec.

Ce type d'optimisation logicielle est complexe à mettre en œuvre pour le compilateur : la taille optimale du bloc dépend de celle de la ligne de cache et de celle du cache. En outre, l'analyse précédente n'a pas pris en compte les échecs de conflit. Même s'il existe des compilateurs optimisants pour la hiérarchie mémoire (par exemple, le compilateur Fortran IBM effectue du blocage), la meilleure performance est souvent obtenue en utilisant des bibliothèques numériques spécialisées. En particulier, le produit matrice-matrice fait partie d'un ensemble de fonctions appelé BLAS (Basic Linear

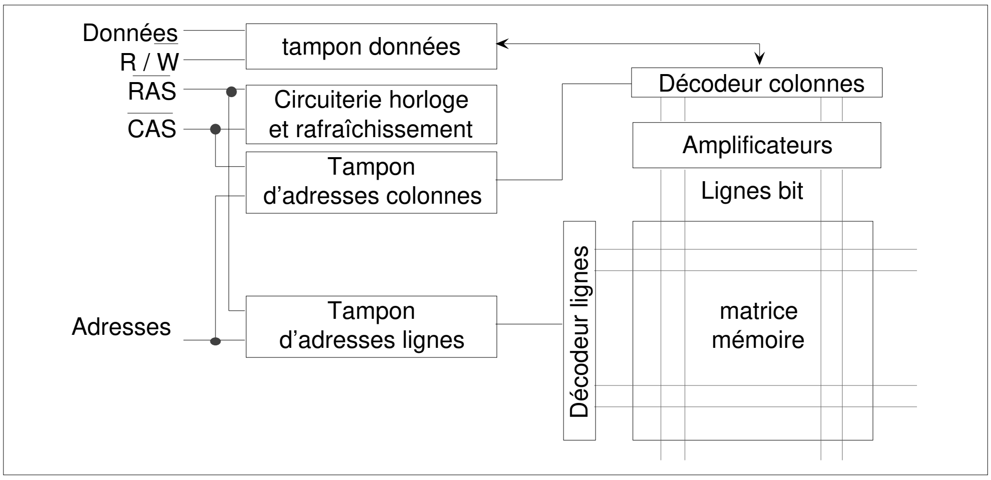


Figure 7.12: Organisation interne d'une DRAM

Algebra Subroutines), pour lesquelles il existe une version de spécification en langage de haut niveau [13], et des versions hautement optimisées en langage machine conçues par les constructeurs de la machine. Ces routines conjugent optimisation pour la hiérarchie mémoire et pour le pipeline.

7.6 Performance : le temps de rechargement

Le coût de traitement de l'échec dépend de deux facteurs : la capacité des circuits DRAM à effectuer des accès contigus d'une part, l'organisation de la mémoire et de la liaison entre le cache et la mémoire principale d'autre part.

7.6.1 Les circuits DRAM

Les circuits DRAMs sont caractérisés par leur taille, et leur organisation : la taille est la capacité totale du circuit, typiquement entre 1Mbits et 128 Mbits ; l'organisation est le nombre de bits adressés en parallèle. Par exemple, une mémoire 16 Mbits peut être organisée en 16Mbits$\times$1 (2^{14} adresses), ou 8Mbits$\times$2 (2^{13} adresses), ou 4Mbits$\times$4 (2^{12} adresses).

Les DRAM classiques

Un circuit DRAM est composé de cellules, qui contiennent chacune 1 bit de données. Dans un circuit $n \times 1$, les cellules sont organisées comme une matrice (grille à 2 dimensions), et sont repérées par un numéro de ligne et un numéro de colonne (fig. 7.12). Le numéro de lignes correspond aux poids forts de l'adresse, le numéro de colonne aux poids faibles. Dans un circuit $n \times p$ bits, on trouve p matrices identiques; une paire (numéro de ligne, numéro de colonne) repère alors p bits simultanément.

La caractéristique la plus importante de l'interface d'adressage des DRAM est que les entrées d'adresse sont *multiplexées* : les adresses de lignes et les adresses de colonnes sont présentées successivement sur les même lignes physiques. D'autre part, l'interface est asynchrone : elle ne reçoit pas de signal d'horloge externe. Le bus d'adresse comporte deux signaux de contrôle, $\overline{\text{RAS}}$ et $\overline{\text{CAS}}$ qui valident les adresses respectivement de ligne (row) et de colonne (column), sur lesquel la DRAM échantillone les composants de l'adresse. En lecture, lorsqu'une adresse de ligne est présentée, la totalité de la ligne est propagée le long des lignes bit vers les amplificateurs ; l'adresse de colonne sélectionne la donnée adéquate et la propage vers les sorties de données.

Originellement, les circuits DRAM multiplexaient les entrées d'adresse pour minimiser le nombre de plots, afin de diminuer le coût de fabrication, qui était dominé par le nombre de pattes du composant. En partie pour des raisons de compatibilité, cette contrainte a été maintenue, et eu des conséquences à long terme sur les architectures des DRAM. Les diverses DRAMs nouvelles qui sont apparues maintiennent pour la plupart une organisation interne identique, mais l'interface d'accès a été sophistiquée pour améliorer le débit, en particulier pour les accès consécutifs. On présentera ici trois classes de DRAMs : les FPM (Fast Page Mode), les EDO (Enhanced Data Out) et les SDRAM (Synchronous DRAM). On trouvera une présentation des organisations les plus récentes (Rambus, Direct Rambus) dans [4] et une comparaison de leurs performances dans [11].

Les FPM DRAMs

Les FPM DRAMs implémentent le mode page : l'adresse de ligne est maintenue constante, et les données de plusieurs colonnes sont lues depuis les amplificateurs (fig. 7.13). Les données d'une même ligne peuvent donc être lues plus rapidement que des données appartenant à deux lignes différentes.

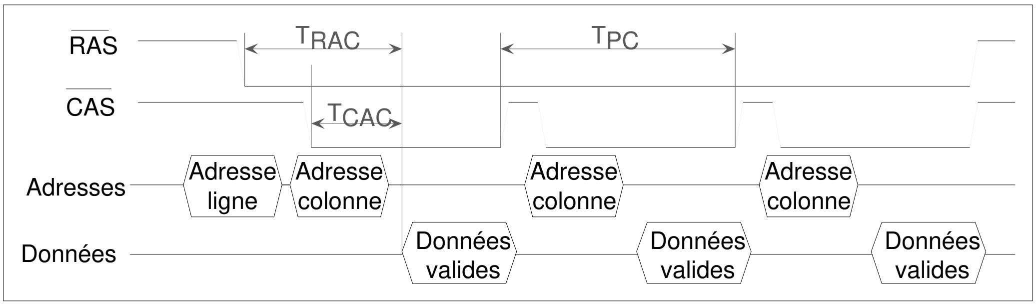


Figure 7.13: Diagramme temporel d'une FPM DRAM

Les EDO DRAMs

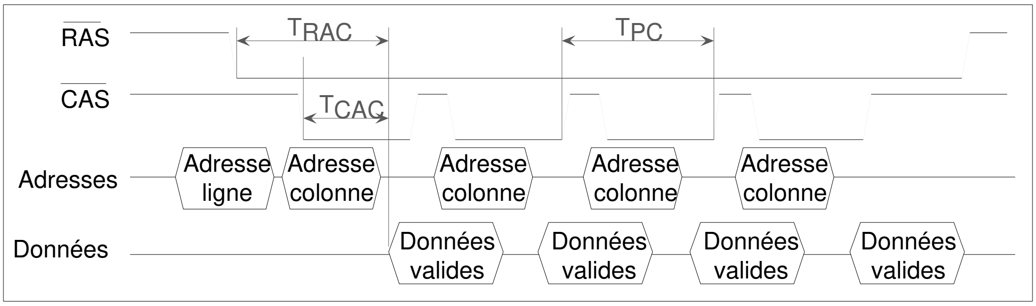


Figure 7.14: Diagramme temporel d'une EDO DRAM

Les EDO ajoutent un registre entre les amplificateurs et les sorties. Ce découplage permet au signal $\overline{\mathrm{CAS}}$ de descendre plus rapidement, donc un préchargement plus précoce (fig. 7.14). En outre, les données en sortie restent valides plus longtemps.

Les SDRAM

Les SDRAM sont synchrones : les échanges entre le CPU (ou le contrôleur mémoire) sont cadencées par une horloge commune. Le circuit contient un registre programmable, qui contient le nombre de données à délivrer par requète. Ainsi, une transaction permet de délivrer 2, 4, 8, ou même une ligne entière, à la fréquence de l'horloge, après l'adressage initial (fig. 7.15). Le temps d'accès à une colonne en régime permanent est le "Read cycle time" d'une SDRAM, et son inverse sa fréquence de fonctionnement. En

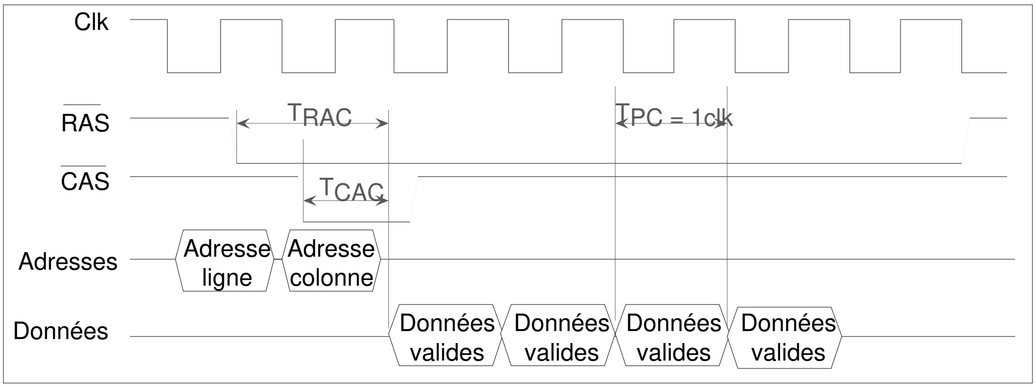


Figure 7.15: Diagramme temporel d'une SDRAM

outre, l'organisation interne utilise plusieurs bancs (2 ou 4), pour masquer le temps de préchargement.

La table 7.6 donne les paramètres temporels et d'organisation de trois DRAM 4M×16 des trois types précédents en 99. T_{RAC} (resp. T_{CAC}) est le délai entre l'abaissement du signal $\overline{\text{RAS}}$ (resp $\overline{\text{CAS}}$), et la disponibilité de la donnée ; T_{PC} est la période de lecture des données pour les accès consécutifs aux éléments d'une ligne. Dans les trois cas, la lecture de l mots consécutifs est bien de la forme :

$$T(l) = T_{\text{RAC}} + lT_{\text{PC}},$$

comme annoncé au début du chapitre. Par exemple, la lecture de 4 mots (de 16 bits) consécutifs s'effectue avec les débits suivants :

FPM : $r = \frac{8\times10^9}{60+4*40} = 36\text{MO}/s$

EDO : $r = \frac{8\times10^9}{60+4*25} = 50\text{MO}/s$

SDRAM : $r = \frac{8\times10^9}{40+4*10} = 100\text{MO}/s$

Type	Temps de cycle	T_{RAC} ligne	T_{CAC}	T_{PC}
FPM	115ns	60ns	15ns	40ns
EDO	104ns	60ns	15ns	25ns
SDRAMs	80ns	40ns	20ns	10ns

Table 7.6: Spécifications des trois types de DRAM

7.6.2 La liaison mémoire principale - cache

Indépendamment des propriétés des circuits mémoire, leur organisation pour constituer la mémoire principale peut permettre de diminuer le temps de chargement d'un ensemble de mots consécutifs.

Les circuit mémoire sont physiquement assemblés en modules, qui sont eux même logiquement assemblés en bancs.

Définition 4 *Un banc mémoire est un ensemble de circuits possédant une entrée d'adresse unique et délivrant un mot de la largeur du bus associé.*

Bus et mémoire large

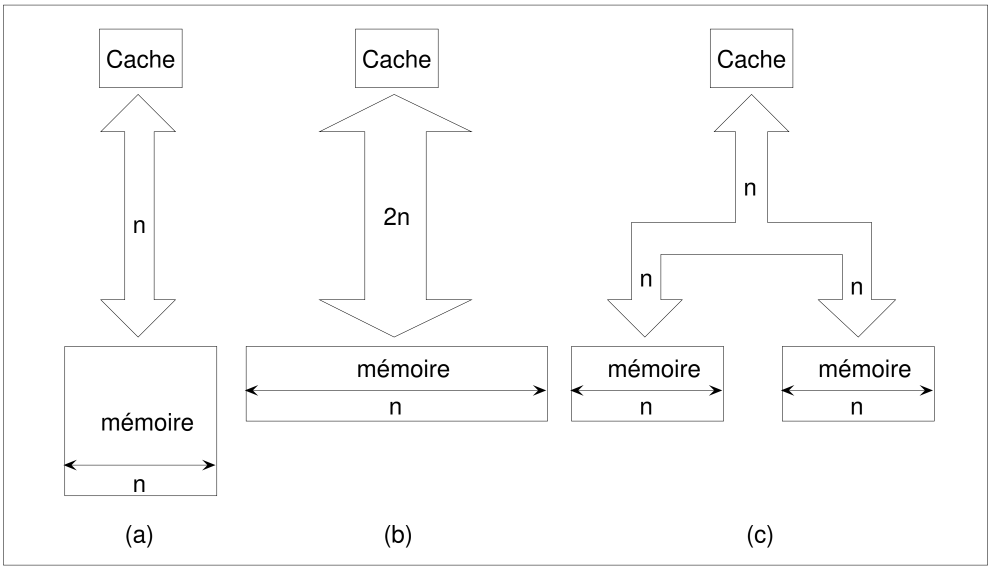


Figure 7.16: La mémoire et la liaison processeur-mémoire

Pour réaliser un banc mémoire d'une capacité donnée, on peut faire varier la largeur (taille du mot et du bus) et la hauteur (nombre de mots adressables) ; par exemple, un banc de 64MO peut être réalisé comme 16M fois un mot de 32 bits, ou 4M fois un mot de 64 bits etc. (fig. 7.16a et b).

Considérons un modèle de performances très simplifié : le temps d'envoi d'une adresse sur le bus est T_{Ad}, le temps de lecture d'un mot du banc est T_m, et le temps de transfert d'un mot sur le bus est T_b ; le bus n'est pas pipeliné, c'est à dire qu'une transaction adresse-lecture-transfert doit être terminée avant que la suivante puisse commencer. Enfin, le temps de lecture T_m ne tient pas compte du mode page, donc est identique pour tous les mots. Alors, le doublement de la taille du mot du banc divise par 2 le temps de chargement d'une ligne. Par exemple, pour une ligne de 16 octets (128 bits),

Taille bus et mot banc	Nombre d'accès	Chargement de la ligne
32 bits	4	$T = 4(T_{\mathrm{Ad}} + T_m + T_b)$
64 bits	2	$T = 2(T_{\mathrm{Ad}} + T_m + T_b)$
128 bits	1	$T = T_{\mathrm{Ad}} + T_m + T_b$

Cette variation a des limites : pour réalise chaque mot, il faut mettre en parallèle des circuits mémoire, qui doivent être adressé simultanément. Les limites technologiques (dissipation de chaleur et temps de propagation des signaux dans les modules, dispersions des signaux sur les bus) donnent actuellement un ordre de grandeur pour une valeur haute à 128 bits.

Mémoire entrelacée

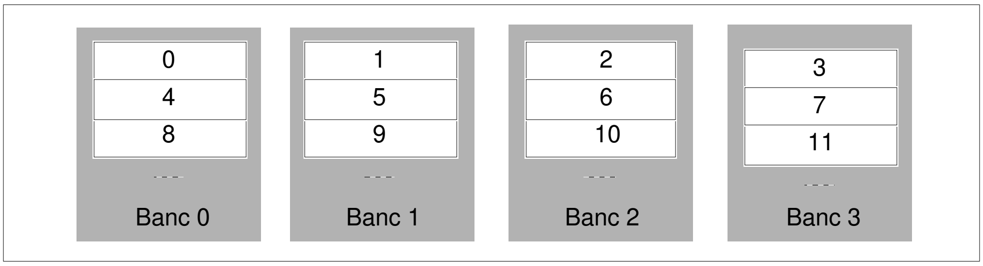


Figure 7.17: La mémoire entrelacée

Pour contourner ces limites, ou bien réaliser de bonnes performances avec un bus étroit, donc moins cher, ou encore des banc mémoires moins larges, la mémoire peut être organisée en plusieurs bancs entrelacés (fig. 7.16c). Les adresses consécutives sont placées sur des bancs différents (fig. 7.17) ; l'envoi d'une adresse permet d'obtenir autant de mots que de bancs, mais les données sont multiplexées sur le bus de données. Par exemple, avec un bus 128 bits, on peut entrelacer 2 bancs larges chacun de 8 octets, ou 4 bancs larges chacun de 4 octets, avec les performances pour la lecture d'une ligne de cache de 16 octets (128 bits) :

Nombre de bancs	Largeur banc	Chargement de la ligne
1	128 bits	$T = T_{\mathrm{Ad}} + T_m + T_b$
2	64 bits	$T = T_{\mathrm{Ad}} + T_m + 2T_b$
4	32 bits	$T = T_{\mathrm{Ad}} + T_m + 4T_b$

Avec un bus 128 bits à 100 Mhz, et les caractéristiques de la table 7.6, en prenant $T_m = T_{\mathrm{PC}}$ on peut utiliser un seul banc formé de 8 circuits SRAM, mais il faut entrelacer 4 bancs, formés chacuns de 8 circuits FPM, pour obtenir un débit de données compatible avec celui du bus.

Discussion

L'organisation à partir de circuits mémoire large semble la plus efficace. En fait, la présentation précédente a négligé d'une part le fait que l'accès à la mémoire et les transactions bus peuvent se recouvrir partiellement, et surtout l'influence du mode page. Typiquement, une organisation où le bus et la mémoire auraient la largeur d'une ligne des cache n'est pas raisonnable, car elle ne permet pas du tout d'exploiter ce mode page. Par exemple, en considérant un bus à 25MHz ($T_{\mathrm{Ad}} = T_b = 10$ns), pour une FPM DRAM, le premier accès coûte 40 + 60 + 40 = 140ns et les suivants 40ns ; un bus à 100 MHz et une SRAM, le premier accès coûte 10 + 40 + 10 = 60ns et les suivants 10ns, en supposant que les transferts d'adresse et de données suivants sont recouverts par les accès mémoire.

Taille bus et mot banc	Nombre d'accès	Chargement ligne FPM	Chargement ligne SDRAM
32 bits	4	$140 + 3 \times 40 = 260$ns	$60 + 3 \times 10 = 90$ns
64 bits	2	$140 + 2 \times 40 = 220$ns	$60 + 2 \times 10 = 80$ns
128 bits	1	140ns	60ns

Donc, pour l'organisation à base de FPM DRAM, le passage à 64 bits divise seulement par environ 1,2 le temps de rechargement (au lieu de 2), et le passage à 128 bits par 1,9 (au lieu de 4). Pour l'organisation à base de SDRAM, c'est encore pire. C'est pourquoi, dans les architectures réelles, la taille de la ligne de cache est toujours un multiple significatif de la largeur du bus.

Un autre problème important pour l'organisation de la mémoire est celui de l'incrément, c'est à dire la plus petite quantité de mémoire que l'organisation mémoire permet d'ajouter. Il est souhaitable, surtout sur les machines bas de gamme, que cet incrément ne soit pas trop grand. Soit à réaliser une mémoire de 128MO. On dispose de deux types de circuits : 64M$\times$ 1 et 16M$\times$ 4, et on veut comparer deux réalisations à un seul banc,

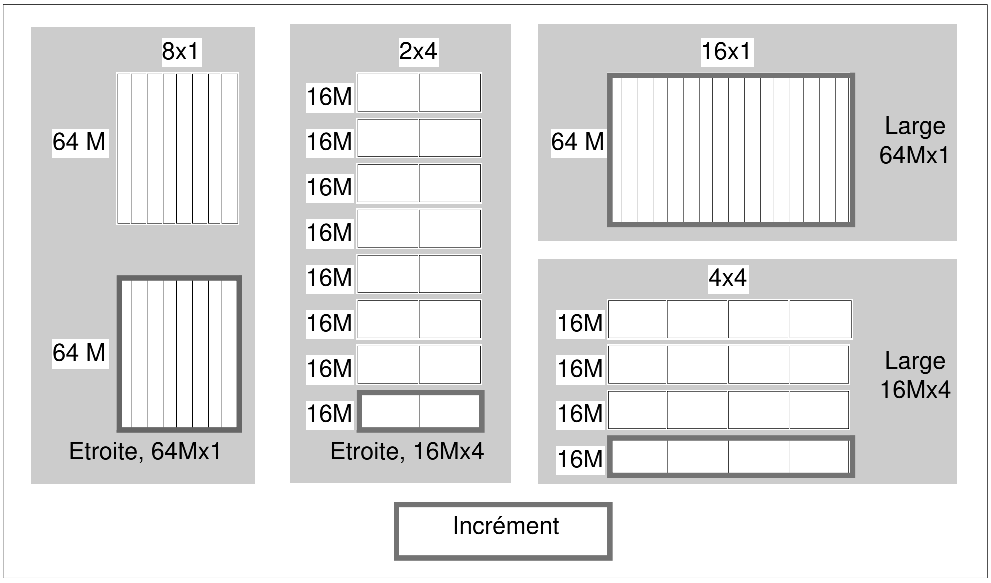


Figure 7.18: Incrément mémoire

l'une avec la largeur mémoire à 1 octet, l'autre avec la largeur mémoire à 2 octets (fig. 7.18). L'incrément minimum est décrit table 7.7.

Etroite		Large	
64M$\times$ 1	16M$\times$ 4	64M$\times$ 1	16M$\times$ 4
64MO	16MO	128MO	32MO

Table 7.7: Incrément minimum pour quatre organisations mémoire

On voit que l'incrément augmente en passant d'une organisation étroite à une organisation large, mais que l'utilisation de circuits plus larges, à capacité égale, permet de le diminuer. En revanche, les circuits plus larges sont plus coûteux, toujours à capacité égale. Les contraintes de coût limitent donc également l'utilisation de mémoires larges. On trouvera une discussion détaillée de l'évolution des circuits mémoires et de ses conséquences architecturales dans [16].

Masquer la pénalité d'échec

L'application du principe du pipeline à l'ensemble processeur-cache-mémoire centrale permet, à pénalité d'échec constante, d'effectuer une partie du rechargement pendant que le programme continue à se dérouler.

En réécriture, l'échec peut impliquer l'écriture de la ligne modifiée. La méthode la plus élémentaire est de disposer d'un tampon d'écriture : l'UC écrit dans le tampon, et celui-ci gère l'écriture effective dans la MP, au rythme de celle-ci. Comme il n'y a généralement pas d'écriture mémoire à chaque cycle d'horloge, un tampon de taille limitée est suffisant. En donnant la *priorité à la lecture sur l'écriture*, le programme peut continuer dès que la ligne est lue, pendant que l'écriture s'effectue.

Le *redémarrage précoce* consiste à lire la ligne mémoire par sous-ensembles. Le sous-ensemble contenant le mot demandé est lu d'abord ; l'exécution peut alors redémarrer, pendant que le reste de la ligne est lue.

On appelle *lecture non bloquante* une technique qui concerne en fait le pipeline : il s'agit d'appliquer le principe de l'exécution dans le désordre aux instructions de chargement ; les instructions qui n'utilisent pas la donnée lue s'exécutent pendant la lecture de la ligne de cache. La technique des *caches non bloquants* permet de limiter le débit demandé à la hiérachie mémoire, dans ce cas. Lorsque plusieurs instructions font un défaut de cache sur la même ligne, avant que celle-ci ait été rechargée, les requêtes sont regroupées, au niveau de la logique de contrôle de cache, et la transaction de défaut n'a lieu qu'une seule fois.

Ces diverses techniques augmentent la complexité du matériel, mais peuvent s'effectuer à débit constant de la mémoire principale.

Préchargement

Le préchargement consiste à anticiper une lecture depuis la mémoire principale vers le cache.

Dans le préchargement par matériel, un échec va déclencher la lecture, non seulement de la ligne requise l, mais aussi d'une autre ligne, typiquement la ligne suivante l', en pariant sur le principe de localité. La ligne l va dans le cache, la ligne l' dans un tampon du processeur. A l'échec suivant, si la ligne est trouvée dans le bloc, elle est transférée dans le cache, la requête vers la mémoire principale est annulée et le préchargement suivant est lancé. On obtient ainsi l'équivalent d'une augmentation de la taille de la ligne sans l'inconvénient d'augmenter le taux d'échec de conflit, puisque le cache n'est pas écrit au préchargement. Ce tampon peut être développé jusqu'à devenir un petit cache ; par exemple, dans l'UltraSparcIII, c'est un cache de 2KO 4-associatif.

Le préchargement par matériel augmente le débit demandé au bus, car une partie des lignes préchargées seront inutiles. Plusieurs jeux d'instruction rendent la hiérarchie cache-mémoire principale visible à l'utilisateur, en fournissant des instructions de préchargement. Par exemple, l'instruction *lfetch* de l'architecture IA-64 permet de forcer le chargement d'une ligne. Avec le préchargement par logiciel, le compilateur ou le concepteur du code en langage machine peut utiliser de façon plus efficace le débit de la liaison MP-cache. En revanche, le cache doit servir les accès aux données déjà présentes tout en recevant la ligne préchargée.

7.7 Conclusion

L'évolution technologique augmente exponentiellement l'écart entre le débit des processeurs et celui des circuits à la base de la mémoire principale. Une part croissante de la complexité des processeurs est donc dédiée aux techniques permettant de diminuer ou de masquer la latence d'accès aux données et aux instructions. Ces techniques comprennent l'intégration de caches de premier niveau toujours plus grands, la gestion de caches de second niveau, et l'implantation de multiples tampons d'écriture, caches supplémentaires intermédiaires (victimes, préchargement etc.). La complexité de la gestion de la cohérence entre les copies multiples d'une même donnée s'accroît en proportion. La gestion de la hiérachie mémoire est actuellement une partie du processeur au moins aussi cruciale que les unités de calcul.

Chapitre 8

Mémoire virtuelle

8.1 Introduction

Figure 8.1: Organisation globale de la mémoire virtuelle

La mémoire virtuelle est la réalisation d'un espace d'adressage de taille supérieure à celle de la mémoire centrale (fig. 8.1). Elle comporte deux aspects : *Hiérarchie mémoire* et *protection.*

- *Hiérarchie mémoire.* La mémoire virtuelle définit un mécanisme de transfert entre la mémoire principale (DRAM) et la mémoire secondaire (disque). En effet, l'espace physiquement adressable d'un processeur est défini par la largeur de son bus d'adresses, noté n dans la suite. Le coût des DRAM et l'encombrement interdisent de réaliser la totalité de l'espace d'adressage du processeur des boîtiers DRAM. La taille 2^m de la MP est donc plus petite que 2^n.

- *Protection et multiprogrammation* La protection intevient lorsque plusieurs processus partagent le processeur. Lorsqu'il est actif, chaque processus doit voir l'architecture comme s'il y était seul. L'espace d'adressage vu par le système, qui gère la commutation des processus, est donc plus grand que 2^n. Les processus doivent donc partager une ressource commune ; les mécanismes de protection permettent au système d'exploitation d'assurer que ce partage s'effectue sans interférence.

En outre, la mémoire virtuelle supporte la relogeabilité, car elle assure l'indépendance entre d'une part les les adresses dans un programme, celles que voit le processeur, et les adresses vues par les circuits mémoires ou les disques. Un programme peut ainsi être placé à tout endroit de la mémoire principale.

Historiquement, la mémoire virtuelle est apparue pour l'aspect hiérarchie mémoire. En effet, il fut un temps où la gestion des transferts entre disque et mémoire principale était à la charge du programme utilisateur. Le programmeur devait alors découper son programme en parties mutuellement exclusives (overlays) et gérer les transferts nécessaires entre MP et disques pour que le programme puisse s'exécuter dans la limite de la mémoire physique disponible. La mémoire virtuelle réalise ces transferts de façon transparente à l'utilisateur. Comme pour la hiérarchie cache-MC, les transferts entre mémoire virtuelle et mémoire centrale s'effectuent par fragments contigus de mémoire, qui sont appelés *pages*. A la différence de la hiérarchie cache-MC, le logiciel participe largement à la réalisation de la mémoire virtuelle.

Actuellement, les aspects de protection pour le support de la multiprogrammation sont aussi importants que l'aspect hiérarchie.

8.1.1 Organisation de principe

Adresse virtuelle et adresse physique

On appelle *adresse virtuelle* une adresse dans l'espace adressable du processeur ($0 \leq a_v \leq 2^n$). Les programmes calculent uniquement en adresses virtuelles : dans l'instruction LD R1, 4(R2), R2+4 définit une adresse

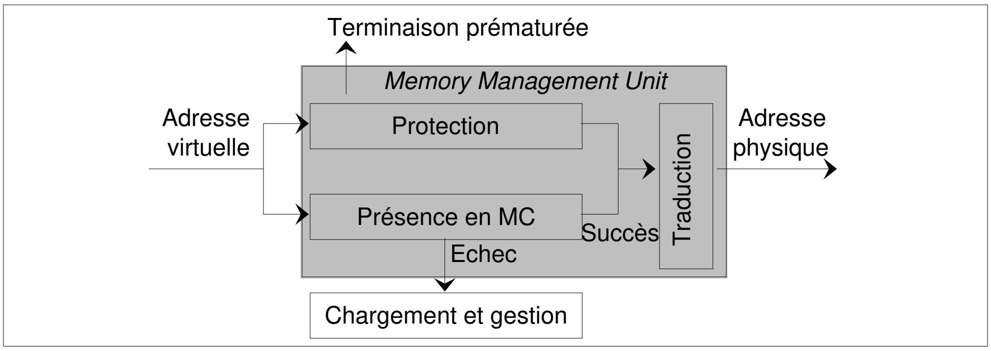


Figure 8.2: Support de la mémoire virtuelle : Description fonctionnelle

virtuelle ; de même, les instructions sont adressées virtuellement : le registre PC contient une adresse virtuelle.

On appelle *adresse physique* une adresse a_p dans l'espace mémoire physique : $0 \leq a_p \leq 2^m$. L'adresse physique est celle qui est présentée en entrée des boîtiers mémoire, pour sélectionner une case mémoire.

Le support de la mémoire virtuelle doit réaliser deux fonctions, *traduction*, et *gestion*. Lors d'un accès mémoire, l'adresse virtuelle doit être traduite en adresse physique. Cette traduction n'est possible immédiatement que si l'adresse virtuelle est présente en mémoire physique. Dans la suite, nous apellerons traduction à la fois le test de présence et la traduction effective.

Si l'adresse virtuelle est absente, la donnée correspondante doit être chargée à partir du disque, et la traduction recommencée. La mémoire virtuelle étant plus grande que la mémoire centrale, ceci peut entraîner le déchargement d'une page de la mémoire centrale vers le disque. La gestion concerne ces mouvements entre mémoire secondaire (disque) et mémoire centrale. La figure 8.2 schématise les opérations réalisées lors d'un accès mémoire.

Traduction.

La traduction est sur le chemin critique de l'exécution des instructions : elle intervient *à chaque lecture d'instruction*, et aussi lors de l'accès mémoire pour les instructions d'accès mémoire. La traduction détermine donc la durée des étages de pipeline pour la lecture des instructions et pour l'accès mémoire.

La traduction requiert donc un fort support matériel. Le matériel correspondant est appelé *Memory Management Unit* (MMU, Unité de gestion mémoire).

Gestion

Le coût d'un échec de traduction est par définition celui d'un accès disque, donc plusieurs centaines de milliers de cycles processeurs, avec un coût de démarrage fixe et élevé (cf. chapitre suivant). Ce coût élevé a trois conséquences.

- La pagination. L'unité de transfert entre MC et disque est un ensemble d'adresses contigües, de taille fixe, de l'ordre de quelque KO à quelques dizaines de KO. En effet, la structure du coût d'un accès disque permet d'exploiter la localité des références, en amortissant le coût de démarrage sur la taille du transfert. La localité des références motive la pagination (ou de la segmentation), exactement comme pour l'échange par blocs avec le cache, mais à une échelle plus globale : un programme au lieu d'un corps de boucle, la pile au lieu d'un fragment de tableau.

- Associativité totale : une page de la mémoire virtuelle peut occuper n'importe quelle page de la mémoire centrale. Le coût d'un accès disque justifie ce choix : d'une part, la mémoire physique doit être exploitée complètement, d'autre part le coût de la recherche associative n'est pas significatif par rapport au coût de l'accès disque lui-même.

- Gestion logicielle : l'algorithme de placement des pages virtuelles en mémoire centrale, et le contrôle du trafic entre MC et disque, sont réalisés par logiciel, pour la même raison. La mémoire centrale doit également être exploitée finement : la page à remplacer lors d'un défaut de capacité est choisie par un algorithme plus complexe, et qui prend en compte des paramètres plus détaillés que ce qui est réalisable par matériel.

8.2 Mécanismes de traduction

8.2.1 Table des pages

L'unité de transfert est la page. La traduction porte donc sur le numéro de page, d'une page virtuelle à une page physique. La correspondance entre numéro de page virtuelle et emplacement dans la hiérarchie (sur disque ou en MC) est enregistrée dans une table, appelée *table des pages*. En effet, cette correspondance est le résultat d'un algorithme complexe qui prend en compte toute l'histoire du fonctionnement de l'ordinateur depuis son dernier démarrage. Elle ne peut donc être décrite qu'explicitement, et pas par une fonction calculée.

Fonctionnellement, la table des pages est un tableau, de taille égale au nombre de pages virtuelles. Chaque entrée de ce tableau contient deux champs : numéro de page physique et contrôle. Si la page virtuelle index est en MC, le champ numéro de page physique contient le numéro de la page physique correspondante. Sinon, il contient l'adresse de la page sur disque, suivant l'organisation propre du système de disques du système. Le champ contrôle contient donc au minimum un bit de validité, qui est mis à vrai lorsqu'une page est chargée, et à faux lorsqu'elle est déchargée.

La traduction consiste à indexer cette table par le numéro de page virtuelle. La partie basse de l'adresse physique, appelée le *déplacement*, est invariante et donc n'intervient pas dans la traduction : elle définit l'adresse à l'intérieur de la page. Le succès est déterminé par la lecture du bit de validité. La concaténation du champ adresse physique et du déplacement fournit, en cas de succès, l'adresse physique délivrée sur le bus mémoire (fig 8.3).

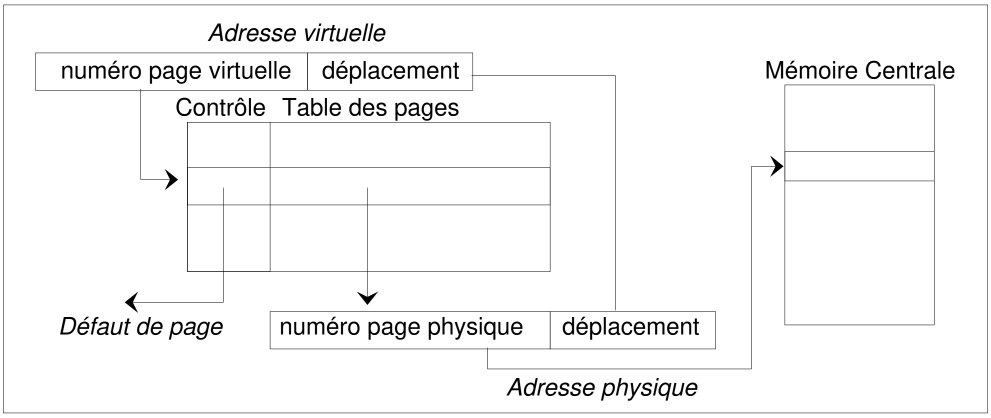


Figure 8.3: Traduction par accès à la table des pages.

Le coût d'un accès disque impose que les écritures en MC ne soient pas reflétées immédiatement sur le disque (analogie avec l'écriture différée). Le chargement d'une page virtuelle vers une page physique implique donc la recopie sur disque de la page remplacée si elle a été modifiée. Pour éviter des recopies inutiles, le champ contrôle contient également un bit de modification (équivalent du dirty bit des caches à réécriture), qui est positionné si la page est modifiée.

La table des pages sert aussi à implémenter la politique de remplacement. Elle suit en général l'algorithme LRU.

8.2.2 TLB

Organisation

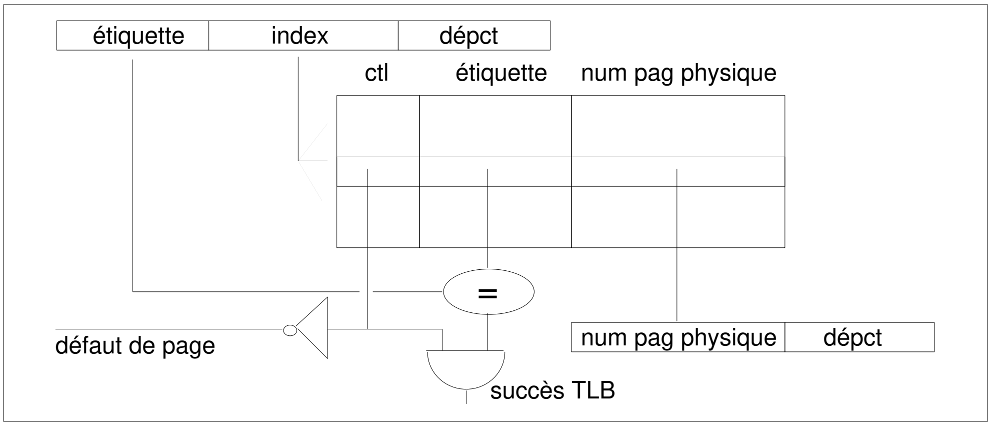


Figure 8.4: Tampon de traduction anticipée : organisation générale

La traduction doit être rapide, de l'ordre d'un cycle processeur. Or, la table des pages d'un processus est une grosse structure de données : pour un espace d'adressage virtuel de 32 bits, et des pages de 4KO, elle comprend 2^{20} entrées. En supposant que la table des pages réside en MP, l'accès à la table des pages impose au minimum deux accès à la MP : un pour lire le numéro de page physique, et un pour effectuer l'accès demandeur. On traitera plus loin le problème de savoir si la table des pages est adressée physiquement ou virtuellement.

L'accès à la table des pages doit donc s'effectuer à travers un cache. Le cache de la table des pages est un cache spécifique, indépendant du caches de données, appelé *Translation Lookaside Buffer* (TLB, tampon de traduction anticipée).

Chaque entrée du TLB contient plusieurs champs : l'étiquette, correspondant à l'adresse virtuelle originelle, et les champs correspondants de la table des pages (fig. 8.4). Cette entrée, correspondant à un élement de la table des pages, est appelé une PTE (page table entry). Les TLB ont en général de l'ordre d'une centaine d'entrées. Il n'y a pas de notion de ligne : chaque entrée de TLB est gérée individuellement.

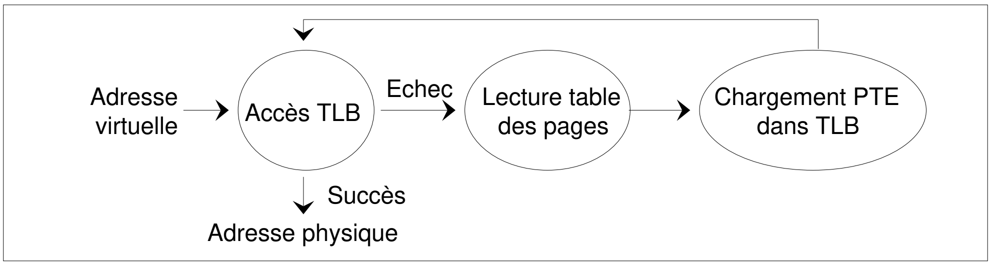


Figure 8.5: Succès et défaut de TLB

Si les informations de traduction sont dans le TLB, la traduction demande pas d'accès à la table des pages. Sinon, il y a *défaut de TLB* ; l'entrée correspondante du TLB est alors mise à jour à partir de la table des pages, et la traduction redémarrée (fig 8.5). Le point essentiel ici est que la traduction d'adresse ne se fait jamais directement à travers la table des pages, mais toujours à travers le TLB. *C'est donc le TLB qui est sur le chemin critique.*

Tradionnellement, les TLB sont complètement associatives, pour limiter le nombre de défauts. La pénalité d'un accès associatif, en temps et en consommation électrique, a crée une tendance à une organisation associative de degré 2 ou 4. De même, en cohérence avec les architectures Harvard, il existe souvent deux TLB séparés, pour instructions et données.

Traitement des défauts TLB

Tradionnellement, les défauts de TLB sont traités par matériel, pour limiter la pénalité de défaut. L'avantage du mécanisme matériel est de limiter la perturbation de l'exécution des instructions utilisateur. Typiquement, dans les architectures Pentium, qui utilisent la gestion en matériel, les instructions qui ne sont pas en dépendance avec l'accès mémoire continuent à s'exécuter.

L'inconvénient est que l'organisation de la table des pages est strictement figée, puisqu'elle doit être connue du matériel. L'OS n'a donc qu'un

Processeur	Associativité	Gestion	Taille TLB Inst (en PTE)	Taille TLB Don. (en PTE)
MIPS R10000	Complète	Logicielle	8	64
Dec Alpha 21164	Complète	Logicielle	48	64
Power PC 604	2-ensembles	Matérielle	128	128
UltraSparc I	Complète	Logicielle	64	64
Pentium II	4-ensembles	Matérielle	32	64

Table 8.1: Organisations de TLB

faible degré de liberté pour la gestion de la table des pages ; en outre, il devra intégrer dans sa définition l'architecture d'une implémentation particulière d'un microprocesseur particulier. Il est donc probable qu'il reflètera rapidement des contraintes matérielles périmées.

Plusieurs architectures actuelles ont choisi un traitement logiciel des défauts de TLB. L'intervention du matériel sur défaut de TLB se limite au déclenchement d'une exception, qui lance une routine système. Le coût de l'exception et de la routine est de quelques dizaines d'instructions, mais peut être beaucoup plus élevé si par exemple le code de la routine gérant l'exception n'est pas présent dans le cache ; en outre, ce type de traitement perturbe toujours massivement le pipeline. Le coût peut alors atteindre plusieurs centaines de cycles.

On verra en 8.4 le contenu du traitement d'un défaut de TLB.

La table 8.1 présente l'organisation des TLB de quelques processeurs.

8.2.3 Mémoire virtuelle et caches

Les caches de données et d'instruction sont indexés par, et contiennent, des fragments d'adresses : l'index est le numéro d'ensemble, le contenu est l'étiquette. S'agit-il d'adresses virtuelles ou physiques ? Les trois solutions existent, suivant que l'index et/ou l'étiquette sont extraites de l'adresse physique ou de l'adresse virtuelle.

Si étiquette et index sont extraites de l'adresse physique (cache *indexé physiquement*, fig. 8.6), le chemin critique additionne l'accès TLB et l'accès au cache, puisque la traduction (avec succès) doit préceder l'accès au cache (avec succès). Une solution est de pipeliner les deux accès. L'autre solution est d'imposer que l'index cache soit inclus dans le champ déplacement pour

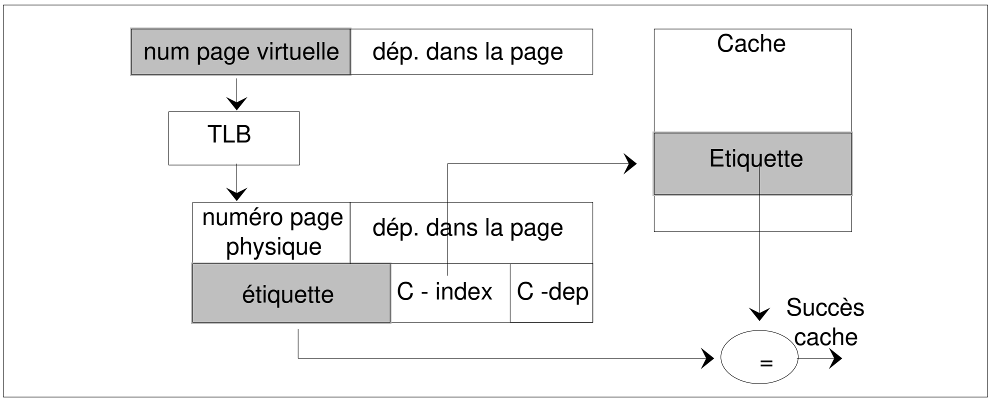


Figure 8.6: Cache indexé et adressé physiquement

la mémoire virtuelle. Comme le déplacement n'est pas traduit, l'accès au cache peut s'effectuer en parallèle à la traduction, et l'étiquette contenue dans le cache est comparée à l'étiquette résultat de la traduction (fig. 8.7). L'inconvénient est de limiter la taille du cache, qui doit être inférieure à la taille d'une page.

Si étiquette et index sont extraites de l'adresse virtuelle (fig. 8.8), la traduction et l'accès au cache sont complètement indépendantes. L'inconvénient est que les problèmes de cohérence des données ne peuvent plus être gérés complètement par matériel. En effet, dans certains cas, deux processus peuvent partager une page mémoire physique (cf 8.3). Deux adresses virtuelles différentes correspondent alors à la même adresse physique. Si le cache est indexé et adressé virtuellement, ces adresses virtuelles peuvent déterminer des adresses différentes dans le cache ; il peut exister alors deux instances de la même case mémoire dans le cache, avec un risque d'incohérence, si l'un des blocs cache est écrit. Ce problème est connu sous le nom de problème d'*alias* (aliasing). Par exemple, le problème d'alias existe sur l'UltraSparcI : le cache L2 peut contenir deux pages. Pour le résoudre, l'OS aligne toutes les pages susceptibles d'être des alias sur de frontières de 16MOctets, qui est le plus grand cache L2 supporté par l'architecture.

8.3 Mécanismes de protection

La mémoire virtuelle offre un support pour une protection entre utilisateurs indépendante de leur niveau hiérarchique.

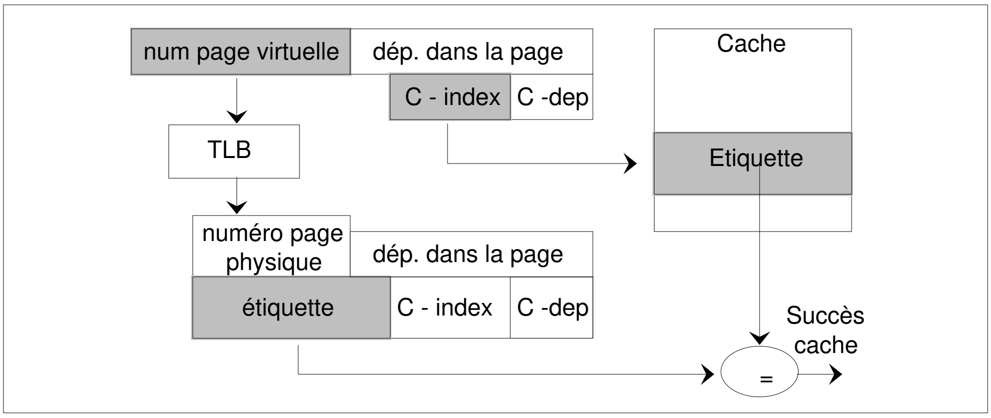


Figure 8.7: Cache indexé virtuellement, adressé physiquement

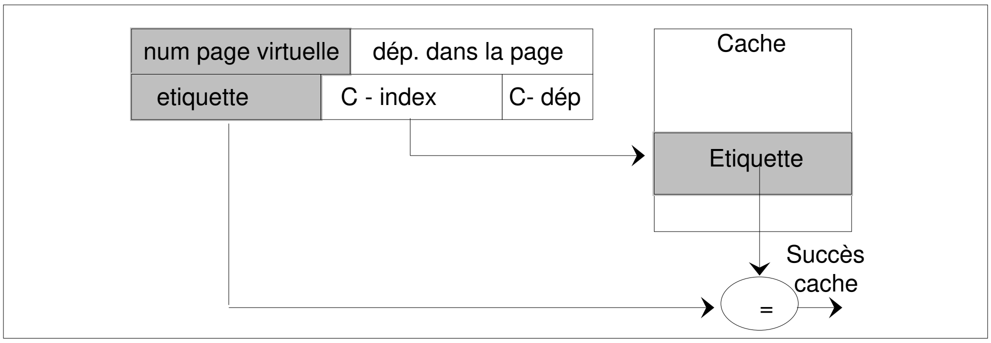


Figure 8.8: Cache indexé et adressé virtuellement

La protection a deux objectifs : isoler et partager. Il est évidemment nécessaire d'isoler les données et programmes appartenant à des utilisateurs différents. Plus précisément, il faut rendre impossible la corruption par autrui des informations : formellement, un processus peut être défini comme un ensemble de ressources mutuellement accessible, ie qui bénéficient de la même protection.

Inversement, le partage d'une zone mémoire est devenu une fonctionnalité standard des OS. On l'utilise par exemple dans la liaison dynamique. La communication inter-processus peut également être implémentée ainsi.

8.3.1 Protection

Il existe deux classes de méthodes, avec le support matériel correspondant : les *identifiants d'espace d'adressage* (address-space identifiers, ASIs), et la segmentation paginée. Dans le cas des identifiants d'espace d'adressage,

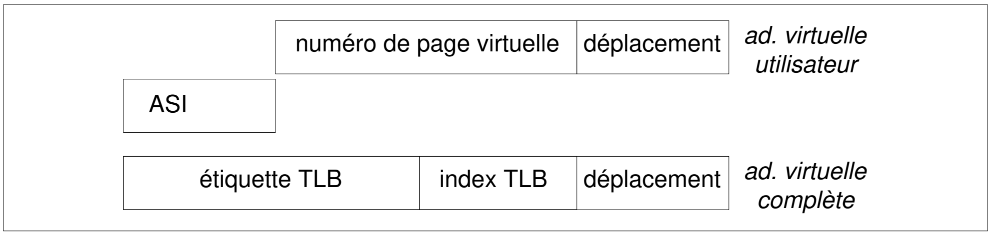


Figure 8.9: Extension de l'adresse virtuelle par ASI

un ASI unique est associé à chaque processus. Au changement de contexte, l'OS place l'ASI approprié dans un registre protégé (accessible uniquement en mode privilégié). L'adresse virtuelle étendue est alors la concaténation de l'ASI et de l'adresse virtuelle utilisateur (générée par le code). C'est cette adresse étendue que traduit le TLB (fig. 8.9). Les architectures MIPS, Alpha et SPARC utilisents les ASIs. Les Asis sont généralement de petite taille, 7 bits pour le 21164, 8 bits pour le R10000. Dans les implémentations

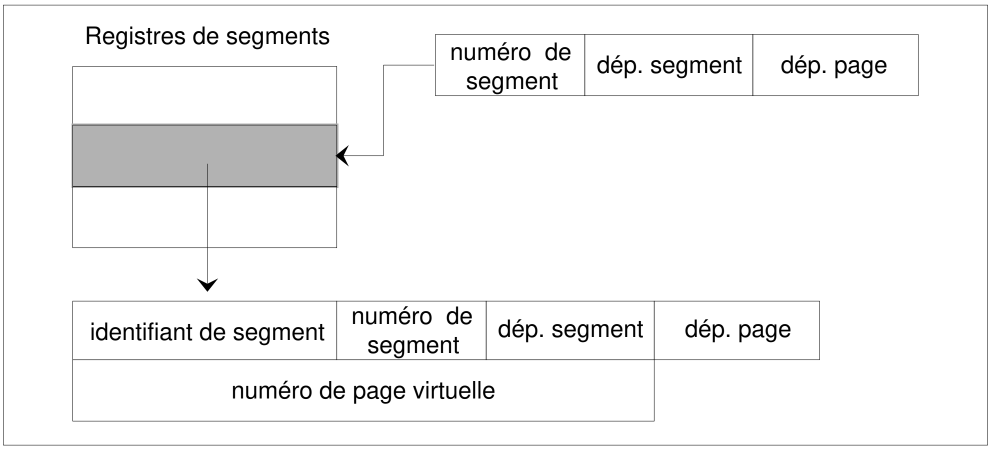


Figure 8.10: Segementation paginée

par segmentation paginée (Power, x86), la protection s'effectue avec une granularité plus fine. Chaque processus dispose de plusieurs identifiants de segments. Comme dans le schéma ASI, une adresse utilisateur est concaténée avec le numéro de segment pour fournir l'adresse virtuelle, mais le numéro de segment est obtenu à partir d'un ensemble de registres indexé par le numéro de segment (fig. 8.10). Par exemple, un utilisateur dispose de 16 segments de 256MO sur le PowerPC, qui réalisent l'espace d'adressage de 4GO correspondant aux 32 bits d'adresse. La fig. 8.11 montre la différence

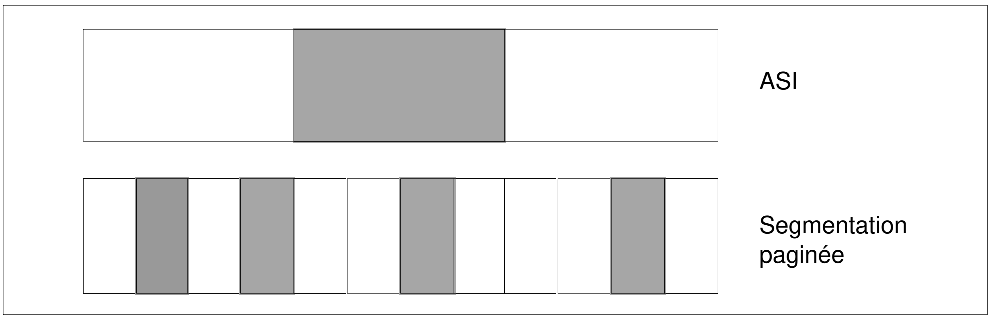


Figure 8.11: Plans mémoire par ASis et par segmentation paginée

des plans mémoire virtuelle entre ASIs et segmentation paginée : dans le premier cas, la mémoire virtuelle d'un processus est contigüe dans l'espace mémoire contenant tous les processus ; dans le second cas, elle est répartie.

8.3.2 Partage

Le mécanisme des ASIs interdit en principe le partage d'une page entre processus différents. Le partage peut cependant s'effectuer, soit par alias, soit en désactivant la protection. Dans le cas de l'alias, l'information de placement est dupliquée entre les tables des pages de deux processus ; autrement dit, deux adresses virtuelles étendues différentes pointent sur la même page en mémoire principale. On a vu ci-dessus les contraintes induites sur les caches.

L'autre solution consiste à indiquer que la vérification des ASIs ne doit pas avoir lieu : deux adresses virtuelles qui diffèrent par leurs ASIs sont considérées comme identiques lors de l'accès au TLB. Le R10000 implémente ce mécanisme à travers le bit Global du TLB, qui désactive la protection. Le problème est que la page devient alors visible par tous les processus. L'entrée correspondante du TLB doit alors être invalidée lorqu'un processus qui n'a pas accès à cette page devient actif. Les jeux d'instruction du microprocesseur proposent dans ce but une instruction qui permet d'invalider une entrée de TLB. Les paramètres de l'instruction sont l'ASI et le numéro de page virtuelle, et c'est bien sûr une instruction protégée.

La segmentation paginée offre un mécanisme de partage plus souple. L'OS peut affecter un segment de même numéro à deux processus, et conserver les autres numéros de segments distincts entre les deux processus et vis-à-vis des autres processus. Ainsi, les deux processus et eux seuls partagent ce segment et celui-là seulement (fig. 8.10).

8.4 Implémentations de la table des pages

Un défaut de TLB entraîne un accès à la table des pages, pour charger l'entrée de TLB requise. Cet accès doit être optimisé, au moins dans le cas où la page utilisateur est présente en MC. Attention : il faut bien distinguer *défaut de TLB*, qui est un défaut de cache de traduction, donc qui doit être traité rapidement, du *défaut de page*, qui demande un chargement à partir du disque, traitement qui relève des Entrées/Sorties. L'implémentation de la table des pages vise à optimiser le traitement des défauts de TLB.

Mais la table des pages d'un processus est une grosse structure de données : rappelons pour un espace d'adressage virtuel de 32 bits, et des pages de 4KO, elle comprend 2^{20} entrées. La table des pages ne peut donc pas résider en mémoire physique, ce qui implique qu'elle est elle même adressée virtuellement. On a donc un problème de démarrage. L'organisation de la table des pages peut suivre deux schémas : *hiérarchique*, ou *inverse*.

8.4.1 Organisation hiérarchique

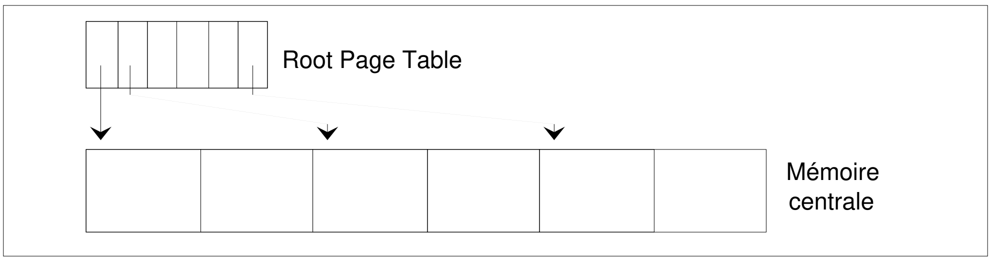


Figure 8.12: Organisation hiérarchique de la table des pages

L'organisation hiérarchique repose sur la pagination de la table des pages. Une table des pages racine (Root Page Table, RPT) contient l'adresse physique (en MC ou sur disque) du début de chaque page de la table des pages, et les bits de contrôle (fig 8.12). La RPT est résidente en MC, ce qui résout le problème de démarrage. Dans l'exemple, en supposant qu'une entrée de la table des pages occupe 8 octets, les 2^{20} entrées occupent $\frac{2^{23}}{2^{12}} = 2K$ pages, la RPT aura donc 2K entrées.

Si la RPT est elle-même trop grande pour être résidente en MC, ce qui est particulier toujours le cas avec des adresses sur 64 bits, le fonctionnement hiérarchique peut être itéré ; par exemple, le 21164 a 4 niveaux. Les performances sont alors fonction de la traversée de la hiérachie, qui peut être *descendante* ou *montante*

Traversée descendante

La fig. 8.13 montre le déroulement d'un accès hiérarchique descendant, pour 2^{20} pages, et une hiérarchie à deux niveaux (la hiérarchie minimale). Les 10

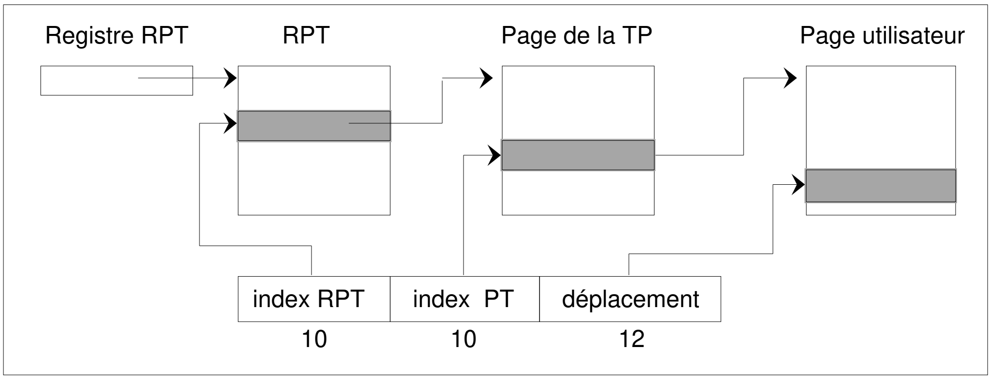


Figure 8.13: Traversée descendante de la table des pages

bits de poids fort de l'adresse virtuelle indexent la RPT, dont l'adresse de départ en MC est contenue dans un registre spécialisé. Si la page référencée de la table des pages est en mémoire, les 10 bits suivants indexent la page de la table des pages.

A partir de l'accès à la RPT, les adresses physiques sont directement disponibles. La traduction de l'adresse virtuelle qui a fait un défaut de TLB ne passe donc pas par le TLB. Le TLB n'est donc pas utilisé dans le traitement du défaut de TLB. Mais l'accès à la table des pages peut lui-même provoquer un défaut de page, donc demander que la page soit chargée ou créee.

Cet algorithme simple permet une réalisation matérielle. En revanche, il exige autant de références mémoire que de niveaux dans la hiérarchie. Il est implanté dans les architectures x86.

Traversée montante

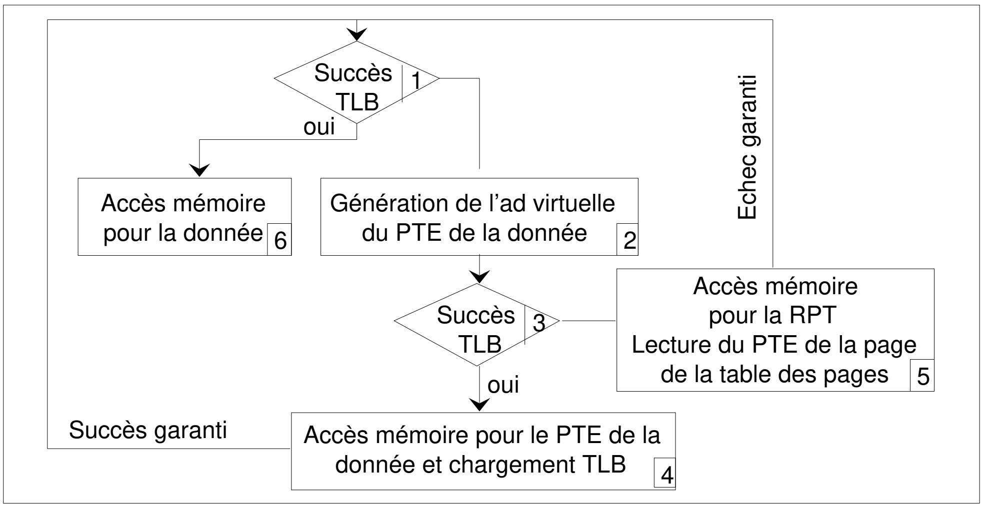


Figure 8.14: Algorithme d'accès à une page en traversée montante

La traversée montante permet de résoudre fréquemment un défaut TLB avec une seule référence mémoire, et de recourir à une traversée descendante sinon. Les architectures MIPS et Alpha l'utilisent

La table des pages est contigüe en mémoire virtuelle. Le numéro de page de l'adresse virtuelle indexe donc directement la table des pages. L'adresse résultante étant une adresse virtuelle, doit être traduite via le TLB. Si cette traduction réussit, l'entrée utilisateur manquante de TLB peut être chargée et l'accès mémoire initial être redémarré. Sinon, l'entrée de TLB correspondante à la page manquante de la table des pages est chargée à partir de la RPT, donc en utilisant directement des adresses physiques. Le chargement de l'entrée du TLB pour la page utilisateur est redémarré et réussit, puis l'accès utilisateur peut être redémarré, et réussit. La fig. 8.14 montre les étapes de l'algorithme : dans le cas où la PTE de la page demandée de la table des pages est dans le TLB, les étapes sont 1 - 2 - 3 - 4 - 6 ; dans le cas où il fait lui même un défaut de TLB, les étapes sont 1 - 2 - 3 - 5 - 1 -

2 - 3 - 4 - 6. Ce schéma complexe requiert une gestion logicielle, avec deux routines différentes pour gérer un défaut de TLB utilisateur et un défaut de TLB demandant un accès à la RPT.

8.4.2 Table des pages inverse

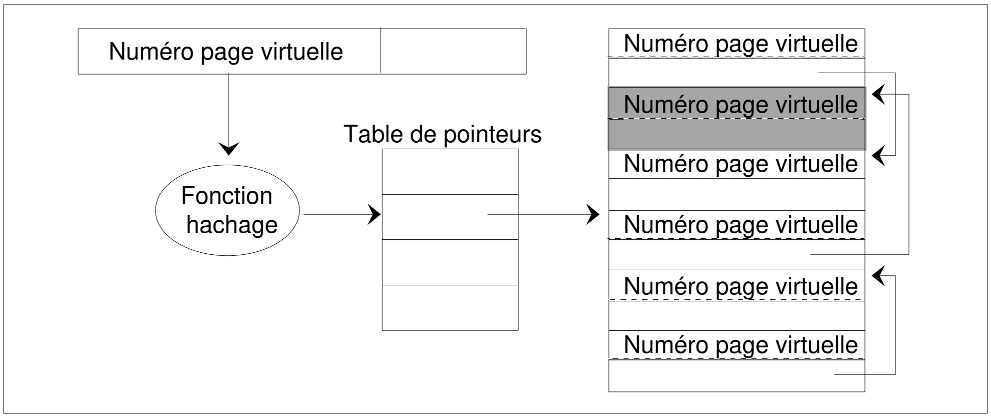


Figure 8.15: Traversée montante de la table des pages

Le traitement rapide des défauts de TLB n'a de sens que pour les pages de la mémoire virtuelles chargées en mémoire centrale. Or, les problèmes du schéma hiérarchique proviennent précisément de la taille potentiellement excessive de la table des pages, qui décrit la totalité de la mémoire virtuelle. Le problème est particulièrement sensible pour un espace d'adressage 64 bits.

La table des pages inverse échange les rôle des entrées et des contenus par rapport à une table des pages classique : l'index est le numéro de page physique en MP, le contenu est le numéro de page virtuelle (fig. 8.15). Lors d'un défaut de TLB, la table des pages fonctionne comme une mémoire associative, pour trouver l'entrée dont le contenu est le numéro de page virtuelle utilisateur. Le numéro de page physique recherché est l'indice de cette entrée.

L'avantage est la dimension de la table des pages, qui est proportionnelle à celle de la mémoire centrale. Cette compacité rend également possible un traitement matériel des défauts de TLB.

La taille de la table des pages rend évidemment impossible l'exploration simultanée de toutes les entrées ; l'exploration séquentielle de la table n'est pas non plus acceptable. Le numéro de page virtuelle est donc haché pour indexer la table des pages. Le mécanisme le plus simple pour résoudre les collisions (deux adresses virtuelles produisent le même index) ajoute dans chaque entrée un pointeur sur l'entrée suivante dans la chaîne de collision. Le service d'une entrée de TLB conduit donc éventuellement à suivre une longue chaîne de pointeurs.

8.5 Conclusion

La mémoire virtuelle traverse tous les niveaux d'un système informatique, de l'utilisateur à la micro-architecture. Même si la hiérarchie est logiquement transparente, l'utilisateur doit être conscient de la différence fondamentale en performances entre un accès disque et un accès en mémoire principale. La micro-architecture du pipeline prend en compte les temps d'accès au TLB pour éventuellement pipeliner les accès aux caches. Le chargement dynamique des programmes repose sur le partage de pages.

Le niveau du système d'exploitation est aussi très fortement impliqué. On a vu qu'il est responsable de l'algorithme de remplacement, et qu'il utilise les mécanismes de protection pour la protection et le partage des processus. Dans les microprocesseurs moderne, le système d'exploitation doit tenir compte de l'interaction entre l'ensemble des niveaux de la hiérarchie mémoire, par exemple pour le problème de l'alias cache. Il devient ainsi difficile de définir des abstractions portables du matériel.

Chapitre 9

Bus et Entrées-Sorties

9.1 Introduction

L'unité centrale et la mémoire principale communiquent avec d'autres sources ou destinations d'information : des disques, des réseaux, des capteurs (souris), des imprimantes, d'autres processeurs, etc. Le terme d'Entrées-Sorties rassemble tous les aspects liées au traitement de ces interactions. La fig. 9.1 présente l'organisation de principe de quelques unes de ces interactions.

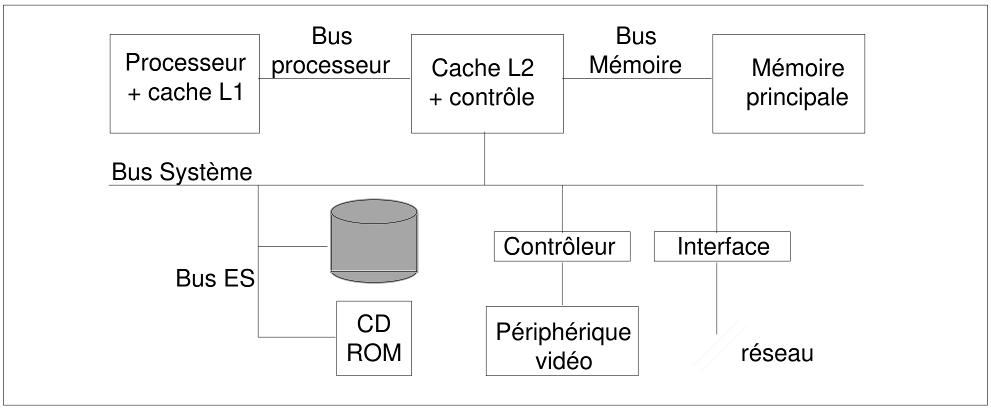


Figure 9.1: Processeur, mémoire et Entrée-Sorties

La plupart des interfaces avec le monde extérieur ne sont pas purement numériques, mais comprennent des dispositifs électro-mécaniques ou électroniques analogiques. On distinguera donc trois niveaux : le *périphérique*, qui est l'interface finale, par exemple le disque, le clavier, la souris ; le *contrôleur*, formé de l'ensemble des circuits numériques de contrôle du périphérique, qui peut varier d'un circuit très simple à un microprocesseur et sa mémoire (le terme de contrôleur est réserve dans la littérature technique à des circuits de contrôle simples ; nous étendons ici cette terminologie) ; les *bus*, qui transmettent l'information entre circuits numériques.

9.2 Les disques

9.2.1 Organisation

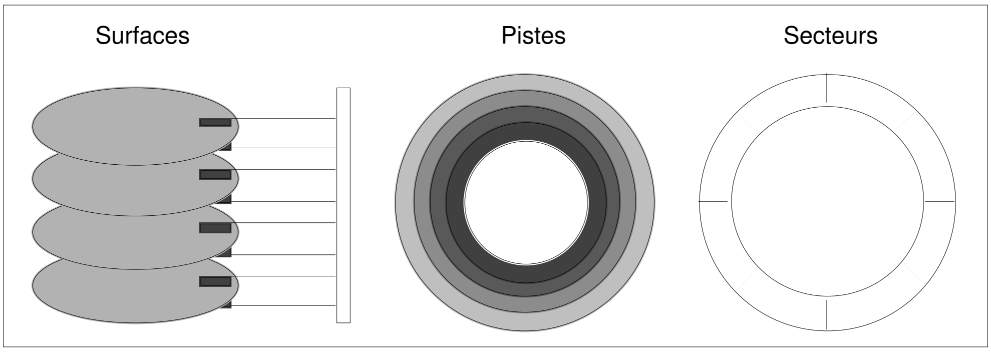


Figure 9.2: Organisation d'un disque

Un disque (fig. 9.2) est constitué d'un ensemble de *surfaces* ou *plateaux* tournant autour d'un axe de rotation. Chaque surface est divisée en *pistes* concentriques, chacune de ces pistes étant découpée en *secteurs*. L'unité de lecture/écriture est le secteur. Les caratéristiques typiques des disques sont :

- vitesse de rotation 3600 à 10000 tours par minute ;
- secteur 512 octets ; 32 à 128 secteurs par piste ;
- 500 à 2000 pistes par surface ;
- 2 à 20 surfaces
- capacité en Gigaoctets

Le passage d'une piste à la suivante peut s'effectuer par têtes fixes ou mobiles. Dans le système à têtes fixes, il y a une tête de lecture écriture au dessus de chaque piste ; le passage d'une piste à une autre se fait par commutation électronique. Avec les unités à tête mobile, il y déplacement mécanique de la tête de lecture écriture d'une piste à une autre.

Pour tenir compte de ces caractéristiques, et minimiser les déplacements ou commutation des têtes de lecture, les adresses disques sont structurées comme l'indique la fig. 9.3. Les enregistrements successifs sont sur des secteurs successifs, puis sur des surfaces successives : ils sont donc situés sur un cylindre. Ce n'est qu'ensuite que l'on change de numéro de piste, ce qui implique un déplacement des têtes.

Unité de disque	Numéro de piste	Numéro de surface	Numéro de secteur

Figure 9.3: Organisation des adresses disque

9.2.2 Accès aux disques

L'accès aux disques se traduit par l'envoi du numéro de piste, qui provoque la recherche de la piste, puis l'envoi du numéro de surface et du numéro de secteur, qui provoque l'attente du passage du début du secteur de la bonne surface sous la tête de lecture-écriture. Ensuite, le disque fournit les octets (en lecture) ou écrit les octets (en écriture) à un débit constant. Ce n'est pas le processeur qui contrôle le transfert de chaque octet vers la mémoire, mais un contrôleur, qui s'interface sur le bus système. L'accès aux disques comprend donc quatre composantes :

- La latence du contrôleur de disques, 1 à 2 ms.

- Le temps de recherche : c'est le positionnement de la tête au dessus de la piste adéquate. En moyenne sur toutes les positions possibles, il est de l'ordre de 10 à 20 ms ; dans le cas optimal, il est réduit au quart.

- La latence de rotation : en moyenne, c'est 1/2 tour de disque, soit 8,3 à 5,6 ms.

- Le temps de transfert d'un secteur, qui est séquentiel, avec un débit de 2 à 4 MOctet/s, soit au plus 0,25 ms

Le temps d'accès moyen à un secteur est donc, en prenant les valeurs les plus fortes, 2 + 20 + 8,3 + 0,25 = 30 ms ; en revanche, s'il y a localité, il est 2 + 5 + 8,3 + 0,25 = 15 ms.

En résumé, l'accès aux disques a deux caractéristiques essentielles : d'une part, la latence est de trois ordres de grandeur supérieure au temps de cycle du processeur ; d'autre part, elle est très sensible à la localité.

9.3 Processeurs d'E/S

9.3.1 Un exemple très simple

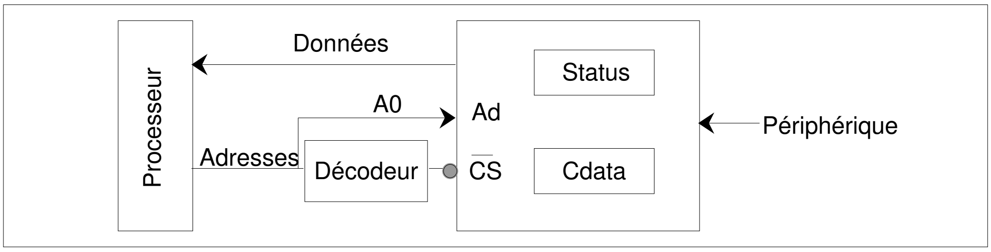


Figure 9.4: Connexion Interface-processeur

Considérons l'exemple d'une interface qui doit recevoir depuis un périphérique, et transmettre au processeur des données octets par octet. L'interface est connectée directement au bus adresses et données du processeur ; cette organisation, typique des architectures à base de microprocesseurs 8 bits, n'est plus d'actualité, mais illustre les fonctionnalités élémentaires requises.

L'interface contient un registre de données *Cdata*, d'un octet, qui recueille la donnée depuis le périphérique. Le contrôle des transferts implique que l'interface puisse indiquer au processeur le moment où le transfert depuis le périphérique est terminé, donc où un nouvel octet est disponible : lorsque l'interface reçoit un octet depuis le périphérique, elle positionne le bit *CRdy* de son registre d'état *Status*. Le processeur vient lire le registre d'état, et lit l'octet si $CRdy = 1$.

Du point de vue de l'UC, il faut donc disposer d'un moyen d'adresser les registres *Cdata* et *CRdy*. La technique la plus généralement utilisée consiste à réserver dans l'espace d'adressage un certains nombres d'adresses. Ces adresses sont associées aux registres de l'interface en décodant le bus d'adresse. Supposons que l'adresse de *Status* est ETAT, que celle de *Cdata* est DATA, et que $CRdy = Status_1$. Le code de la lecture d'un octet est alors :

bcl:	LDB	R1, ETAT	; R1 $\leftarrow$ *Status*
	ANDI	R1, R1, 2	; $CRdy = 1$?
	BZ	bcl	; octet non disponible
	LB	R1, DATA	Lecture le l'octet
	...		

Dans cet exemple, le processeur interroge l'interface jusqu'à ce qu'elle soit disponible. Cett technique est appelée *polling* On verra plus loin qu'il est possible d'affranchir le processeur de cette tâche, en autorisant l'interface à signaler sa disponiblité.

Compliquons légèrement le problème, en supposant que l'interface doive aussi pouvoir recevoir un octet du processeur, pour transmission à un périphérique. Le registre *Status* doit alors contenir un autre bit, pour signaler que l'interface a terminé la transmission précédente, et est donc prête à recevoir un nouvel octet. Le sens de la transmission doit aussi être précisé, donc l'interface doit disposer d'un signal d'entrée $\mathrm{RD}/\overline{\mathrm{WR}}$. Enfin, le processeur doit signaler à l'interface à quel moment il a écrit l'octet sur le bus de données, d'où un signal $\overline{\mathrm{DS}}$ (Data Strobe). L'organisation temporelle de ces divers signaux forme le protocole de bus, qu'on verra dans la section 9.4.

9.3.2 Processeurs d'E/S

Dans une machine monoprocesseur, les opérations d'E/S sont en dernière analyse contrôlées par l'UC. La première question est de savoir à quel niveau se fait ce contrôle. L'UC réalise-t-elle le contrôle fin des opérations d'E/S, ou délègue-t-elle le travail à des opérateurs ou processeurs spécialisés dont elle initialise le travail et contrôle la fin d'exécution ? La connexion directe vue ci-dessus est un exemple de la première solution. La tendance constante est de déléguer un maximum de fonctionnalités à des processeurs autres. On peut de ce point de vue distinguer deux types de fonctionnalités : les traitements spécialisés fortement liés aux E/S, typiquement les traitement postscript et les traitements graphiques, et les traitements de transfert de données purs, par exemple la lecture d'une page disque.

9.4 Les bus

9.4.1 Fonctionnalités

Les transferts d'information entre le processeur, la mémoire et les différents contrôleurs d'Entrée-Sorties s'effectuent à travers des dispositifs de communication appelés *bus*. Un bus comprend en génaral trois sous-ensembles : les lignes d'adresses, qui sélectionnent l'unité réceptrice, les lignes de données et les lignes de contrôle.

Contrairement à un bus interne, un bus externe est beaucoup plus qu'un ensemble de fils électriques passifs. On a vu dans le cas des bus internes du processeur qu'un bus ne pouvait être commandé que par un organe à la fois. Pour les bus externes au processeur, qui sont ceux dont il est question ici, le problème est le fonctionnellement le même. La différence majeure est qu'il n'y a pas de contrôle centralisé naturel : chaque organe (par exemple le processeur, ou un disque) peut vouloir prendre le bus. Une transaction bus comprend donc deux étapes, d'abord l'arbitrage entre plusieurs demandeurs potentiels, puis le transfert de données. L'ensemble des mécanismes d'arbitrage et de transfert est appelé le *protocole de bus.*

Le processeur est connecté au monde extérieur par un bus propriétaire (du constructeur), dont le protocole est spécifique du processeur. Les bus d'E/S, par exemple le bus PCI ou le bus SCSI (aux extrémités du spectre de performances), interconnectent les dispositifs d'entrée/sortie au bus processeur. Ces bus système peuvent accepter des dispositifs de constructeurs variés, et dont les débits et latences sont très différents. Leur protocole est donc défini publiquement, par des standards.

9.4.2 Arbitrage

On appelle *maître* tout organe capable d'accéder au bus, c'est à dire d'en prendre le contrôle en émettant des requêtes. Certains organes, comme la mémoire, sont par nature esclave. A l'envoi d'adresses (et de données en cas d'écriture), et de commandes de lecture ou d'écriture, elle effectue l'action demandée. Le processeur agit toujours comme maître. D'autres contrôleurs, peuvent également être maîtres. L'existence de plusieurs maîtres potentiels implique la nécessité d'arbitrage pour l'accès au bus. Les protocoles utilisables doivent concilier les objectifs de contraintes de priorité de service (urgence respective des requêtes des différents organes) et d'équité (que tout organe, même le moins prioritaire, puisse être servi).

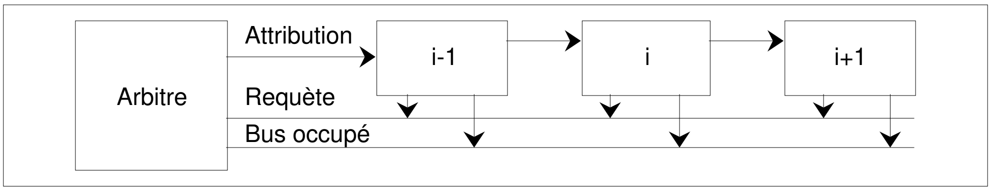


Figure 9.5: Arbitrage par chaînage

Arbitrage par chaînage

L'arbitrage par chaînage (daisy chain) est illustré par la fig. 9.5. Les différents candidats au bus envoient une requête en activant le fil Requête. Lorsque le maître du bus a fini d'utiliser le bus, il désactive le signal Bus Occupé. L'arbitre envoie alors une transition sur la ligne Attribution. Si l'organe de rang i n'est pas demandeur du bus, il fait passer la transition sur la ligne Attribution pour l'organe de rang $i + 1$. S'il est demandeur, il bloque la propagation de cette transition et active le signal Bus Occupé.

Dans la version simple, ce dispositif implique un ordre de priorité fixe, décroissant lorsque i augmente. Une priorité tournante, où le dernier servi devient le moins prioritaire, peut être obtenue si l'on impose qu'un maître qui vient de libérer le bus ne puisse l'obtenir à nouveau avant d'avoir vu le fil Requête passer à l'état inactif. Comme tous les candidats au bus maintiennent la ligne Requête dans l'état actif, un maître venant d'utiliser le bus ne pourra l'obtenir à nouveau avant que tous les autres candidats aient d'abord été servis.

Arbitrage avec un arbitre centralisé

Tous les candidats potentiels au bus transmettent leur requête via un fil spécifique à un arbitre unique qui connaît donc les candidats à l'utilisation au bus et leur priorité respective. Les attributions du bus par l'arbitre utilisent également des lignes spécifiques.

Arbitrage avec un mécanisme décentralisé

Les requêtes du bus se font à l'aide de plusieurs lignes. L'ensemble de ces lignes constituant un niveau de priorité. Par exemple, avec 4 fils de requête, il est possible d'avoir 16 niveaux de requêtes, de 0 (pas de requê te) à 15 (requête de priorité maximale). De cette manière, tout candidat au bus peut comparer son niveau de requête au niveau de requête présente sur le bus, et savoir s'il peut devenir le maître du bus.

9.4.3 Transfert de données

La deuxième grande caractéristique d'un protocole de bus est le type de référence temporelle. Dans un bus synchrone, l'émetteur et le récepteur partagent une horloge. Dans un bus asynchrone, l'émetteur et le récepteur ont chacun des références de temps distinctes et incomparables.

Bus synchrone

Lors d'une lecture, le maître place l'adresse sur les fils adresses sur un cycle d'horloge et l'ordre de lecture sur les fils commandes. Les adresses sont décodées par tous les esclaves et la lecture effectuée. Le mot lu est placé sur les fils données n cycles bus après l'initialisation du transfert par le maître, avec $n \geq 1$ (n dépend du temps de cycle bus, de la longueur du bus, du temps d'accès à l'esclave, etc.). De même, lors d'une écriture dans un esclave, le maître place simultanément adresses et données sur les fils correspondant et l'ordre d'écriture sur les fils commande. Dans tous les cas, toutes les opérations dans les différents organes, maître et esclaves, sont synchronisés par l'horloge bus.

La fig. 9.6 montre l'exemple du bus POWERPath-2 de SGI. Il utilise un bus de données de 256 bits, et un bus d'adresse de 40 bits ; il est cadencé à 47,6MHz.

Toutes les transactions de ce bus utilisent exactement cinq cycles bus. Les deux premiers cycles sont utilisés pour l'arbitrage. Les requêtes des différents organes sont envoyés lors du premier cycle, et un arbitrage décentralisé attribue le bus à un maître lors du second cycle. Les adresses et la commande sont envoyées dans le troisième cycles. Le quatrième cycle est consacré au décodage des adresses par les esclaves. Le cinquième cycle signale la fin de transaction par un acquittement, et commence le transfert des données. Lorsqu'il n'y a pas d'acquittement, toute la transaction doit être recommencée. L'exemple de la fig. 9.6 montre la possibilité de lire quatre données d'adresses successives en une seule transaction.

En outre, le bus POWERPath-2 est pipeliné : la phase de requête de la transaction $i+1$ commence pendant la phase de transmission des données de la phase i. Le bus peut ainsi assurer un débit soutenu de 1,2 GO/s (32 octets $\times$ 47,6MHz).

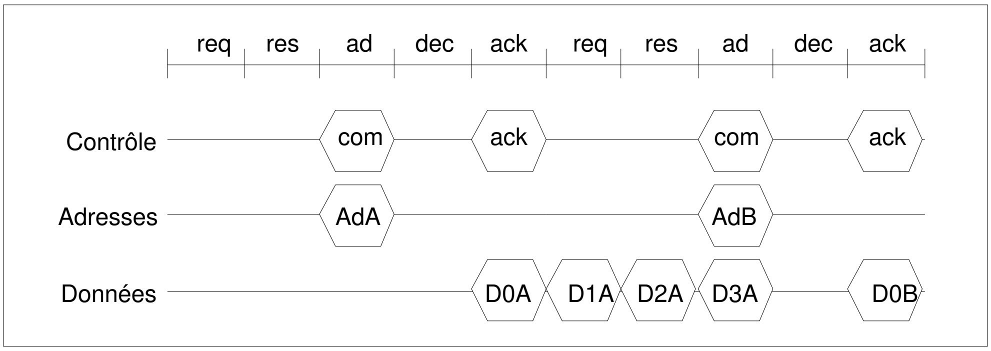


Figure 9.6: Diagramme temporel du bus POWERPath-2

Les bus synchrones sont bien adaptés pour des composants fonctionnant à la des horloges pas trop différentes, comme par exemple l'unité centrale et la mémoire. L'inconvénient fondamental est que les dispersions d'horloge, qui augmentent avec la longueur des connexions, imposent des longueurs relativement faibles, limitées à 50 cm, surtout si l'on veut fonctionner à fréquence élevée. Dans le cas d'organes où des composants lents doivent coexister avec des composants rapides, les bus asynchrones sont obligatoires. C'est le cas également pour des connexions longues.

9.4.4 Bus asynchrone

Pour un bus asynchrone, le maître et l'esclave doivent se synchroniser, typiquement par un protocole de requète/acquittement dont le support est une ou plusieurs lignes de contrôle. Nous considérons un exemple synthétique, d'un bus où adresse et données sont multiplexées. Le contrôle comporte trois lignes :

- ReqL : demande une lecture ; l'adresse est placée simultanément sur le bus ;
- DonneePrete : signale qu'un mot est disponible sur le bus de données.
- Ack : acquittement.

La fig. 9.7 détaille le protocole pour une lecture ; l'arbitrage n'est pas décrit. Le maître et l'esclave partagent le bus adresses/données, et ne doivent donc pas le positionner simultanément. Les signaux de contrôle sont par convention actifs à l'état haut. Le principe du protocole est que les lignes

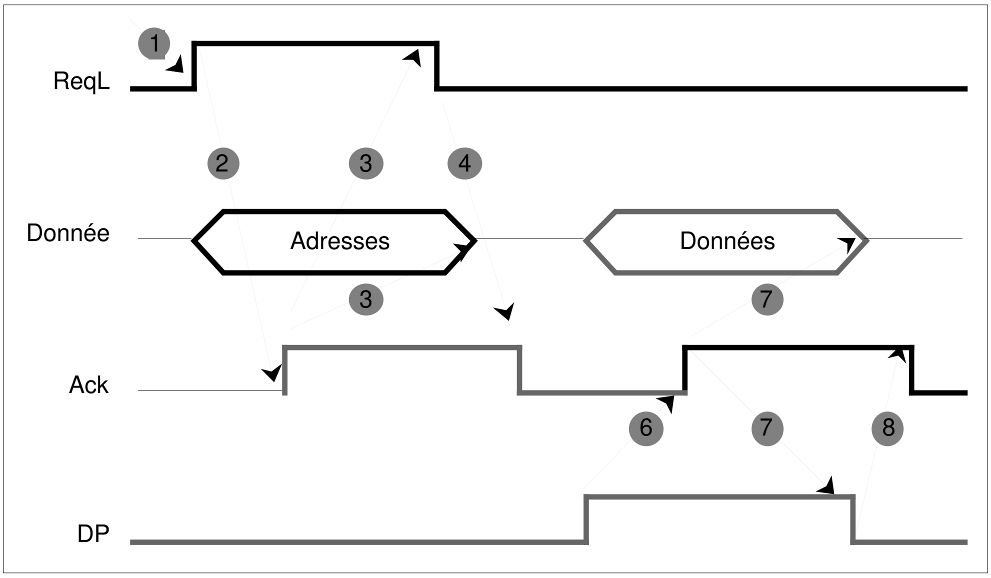


Figure 9.7: Diagramme temporel d'une lecture sur bus asynchrone

communes (données/adresses et Ack) doivent être mises à l'état inactif par l'organe qui les a activées antérieurement. Son correspondant doit donc lui envoyer un signal pour lui indiquer qu'il a vu le signal actif, et qu'il n'en a plus besoin.

- 1. Le maître demande la lecture : il lève le signal ReqL et place l'adresse sur le bus.
- 2. L'esclave voit ReqL et acquitte en activant Ack.
- 3. le maître voit l'acquittement, abaisse ReqL et libère le bus données/adresses.
- 4. L'esclave voit que le maître a vu l'acquittement (ReqL à l'état bas), et descend Ack.
- 5. Quand l'esclave a la donnée prète , il la place sur le bus données/adresses et active DonneePrete.
- 6. Le maître voit DonneePrete ; il lit le bus ; quand il a terminé, il acquitte
- 7. L'esclave voit Ack, libère DonneePrete et le bus données/adresses

- 8 Le maître voit DonneePrete à l'état bas, et descend Ack.

9.4.5 Le bus PCI

Le bus PCI (Perpheral Component Interconnect) est une spécification d'Intel, dont la version 1.0 a été définie en Juin 92 (V2.1 en 95). L'importance économique du marché des PC en a fait un standard de fait, géré par un consortium de fabricants de circuits (PCI Special Interest Group). C'est l'exemple type d'un bus système.

Le bus PCI est synchrone, à 33 ou 66MHz. Les bus données et adresses sont multiplexés, sur 32 bits, éventuellement 64 bits. Des transferts vectorisés (burst) sont possible : le maître transmet une adresse et un nombre d'octets, et le periphérique placera successivement sur le bus les octets requis. Pour un transfert de grande taille, on atteint ainsi un débit asymptotique de 4 octets par cycle à 33MHZ, soit 132MO/s, dans la configuration la plus basse (à multiplier par 2 en 66MHz, ou en 64 bits). En revanche, le bus PCI n'est pas pipeliné : adresse et données se succèdent sur le bus multiplexé. On voit donc qu'il s'agit plutôt d'un bus destiné à des transactions massives.

Un transfert comporte une phase d'adressage, qui dure 1 cycle, suivie d'une phase de transfert. On étudie le cas d'une lecture.

- *Cycle 1.* Le maître place une adresse sur le bus adresses/données ; cette adresse identifie le périphérique, et l'adresse de départ des données à transférer. Le maître définit le type de la transaction via les lignes de contrôle C/BE, et descend le signal Frame pour signaler la présence d'une transaction.

- *Cycles suivants* Les périphériques décodent l'adresse. La cible se signale en descendant le signal DEVSEL. Il faut noter que la cible doit échantilloner l'adresse, puisqu'elle contient aussi l'adresse locale. La cible dispose de 6 cycles pour descendre le signal DEVSEL, faute de quoi la transaction est annulée. Plus précisément, toutes les interfaces connectées sur le bus, sauf une, doivent décoder effectivement les adresses et répondre au cycle 2, 3 ou 4. Une interface, typiquement le bridge vers un bus d'extension, peut réclamer les adresses non décodées, et descendre DEVSEL au cours des cycles 5, 6 et 7.

Des transferts complexes sont rendus possibles par les lignes C/BE. A partir du cycle 2, et pendant tout le transfert, le maître identifie par cette ligne les octets à transférer, par exemple 0000 pour les quatre octets à partir de l'adresse de départ, 0001 pour les trois plus forts, etc.

9.4.6 Amélioration des performances

Les débits des bus sont fonction de la nature synchrone ou asynchrone du protocole, et des durées des différentes opérations du bus. Elles peuvent être améliorées par un certain nombre de techniques.

Augmentation de la largeur du bus

Un phénomène typique de l'évolution récente est l'augmentation de la largeur des données transmises. Comme les bus entre cache et mémoire principale ont souvent à transférer des blocs de caches, l'élargissement du bus données permet de transférer plusieurs mots par cycle bus, réduisant ainsi le nombre de cycles bus nécessaires. C'est l'une des techniques utilisées pour diminuer la pénalité d'échec cache.

Transferts de blocs

Les temps de transfert de blocs de caches ou de blocs disques peuvent être réduits si le bus peut transférer plusieurs mots d'adresses successives en une suite ininterrompue de cycles bus sans envoyer une adresse ou libérer le bus entre chaque cycle bus. On a vu cette technique dans le cas du bus PowerPath-2.

Transactions éclatées ou entrelacées

Une transaction sur un bus peut n'utiliser le sous-bus contrôle ou données ou adresses que pendant une partie du temps que dure la transaction globale. C'est le cas par exemple pour les lectures mémoire lorsque le temps d'accès de la mémoire est de plusieurs cycles bus. Avec un protocole où le bus est bloqué pendant toute la durée de la transaction, le bus attend entre le moment où il a transmis l'adresse mémoire et le moment où la donnée mémoire est disponible sur le bus. En éclatant la transaction en ses différentes étapes où le bus est effectivement utilisé, on peut libérer le bus lorsqu'il n'est pas utilisé, et entrelacer plusieurs transactions. On peut ainsi améliorer le débit effectif du bus, à condition que le bus et organes connectés puissent gérer plusieurs transactions entrelacées.

9.5 Traitement des Entrées/Sorties

9.5.1 Typologie

Les deux techniques fondamentales de contrôle des transferts correspondent à deux attitudes possibles du contrôleur.

- *interrogation* (*polling*). Le contrôleur est passif, et attend que l'UC vienne l'interroge par lecture du mot d'état.

- *interruption.* L'UC délègue une tâche au contrôleur, et continue l'exécution en cours ; lorsque le contrôleur a terminé l'E/S, il demande à l'UC d'intervenir, ce qui interrompt l'exécution en cours.

Le choix entre les deux techniques dépend du taux d'utilisation du processeur requis pour la gestion de l'évènement d'E/S. Considérons l'exemple d'un processeur à 100MHz, et de trois périphériques :

- une souris, qu'il suffit d'interroger 30 fois par seconde ; on a donc 30 évènements par seconde.

- une disquette, qui transmet par paquets de 16 bits, avec un débit de 50KO/s ; on a donc $50.10^3/2 = 25.10^3$ évènements par seconde.

- un contrôleur de disque, qui peut transmettre par 32 bits, avec un débit de 2MO/s ;on a donc $2.10^6/4 = 5.10^5$ évènements par seconde

En supposant que le traitement d'un évènement consomme 100 cycles processeur, on peut calculer la fraction du temps processeur utilisée par la gestion directe .

- souris : $\frac{30.10^2}{100\times 10^6} = 0,003\%$

- disquette : $\frac{25.10^5}{100\times 10^6} = 2,5\%$

- disque :$\frac{5.10^7}{100\times 10^6} = 50\%$

La gestion par polling de la souris est donc négligeable, alors que celle du disque n'est pas envisageable.

9.5.2 Mécanismes d'interruption

Comme on l'a vu dans les chapitres architecture logicielle et pipeline, le jeu d'instructions et la micro-architecture fournissent le suuport pour gérer les situations exceptionnelles automatiquement et de façon transparente par rapport à l'utilisateur final. Les mécanismes matériels d'interface dédiés aux interruptions doivent donc permettre qu'un contrôleur d'E/S se signale à l'UC, et fournisse les informations nécessaires au traitement de l'interruption. Les informations liées aux interruptions sont naturellement convoyées à partir des contrôleurs d'E/S par les bus d'E/S et le bus système, et font partie du protocole de bus. La présentation simplifiée ci-dessous interface directement les périphériques sur l'UC.

Reconnaissance des interruptions

Un ou plusieurs fils de requêtes transmettent à l'UC les requêtes d'interruption. Deux organisations sont possibles : une seule entrée de requête externe, appelée généralement INT (pour interrupt), ou bien plusieurs entrées, qui codent le niveau de priorité de la requête. Certains processeurs disposent d'un signal d'accusé de réception de la requête d'interruption (INTA), pour signaler aux interfaces de gestion des interruptions la prise en compte d'une requête. D'autres processeurs accusent réception par une écriture via les bus adresses et données.

Pour les processeurs n'ayant qu'une seule entrée pour les requêtes externes se pose le problème de transmettre par un seul fil des requètes potentiellement simultanées. D'autre part, l'UC n'a pas de moyen direct de connaître quel contôleur d'E/S a émis une requête d'interruption. Elle sait simplement qu'il y a une requête. Ce double problème peut être réglé par une interface matérielle spécialisée.

L'interface matérielle spécialisée (fig. 9.8) est adressable par l'UC : celle-ci peut lire (mot d'état) ou écrire (mot de contrôle) des registres de cette interface. Les requêtes individuelles des périphériques sont transmises au contrôleur d'interruptions. Celui-ci va transmettre la requête à l'UC via le fil de requête INT. L'UC répond, lorsqu'elle est prête à traiter la requête, par un accusé de réception (INTA). Dans le cas de requêtes multiples, le contrôleur établit une priorité entre les requêtes, selon plusieurs protocoles possibles établis par programmation du contrôleur : ordre de priorité fixe entre les requêtes, ordre de priorité tournante (le dernier servi devient le moins prioritaire). Le choix d'un type de priorité dépend de la nature des

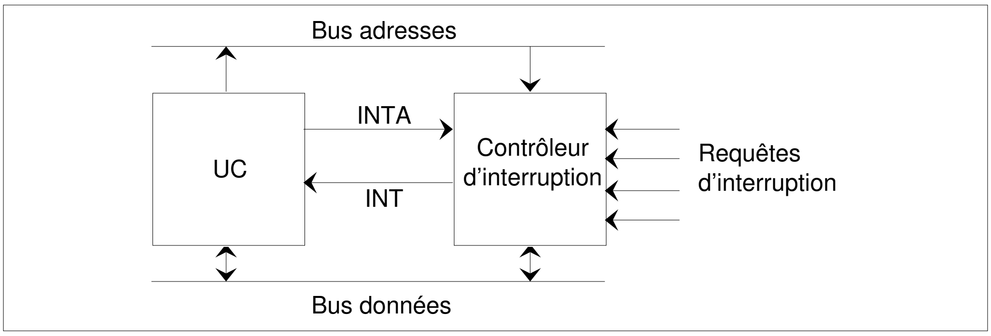


Figure 9.8: Contrôleur de priorités

périphériques : périphériques avec contraintes temporelles très différentes, ou périphériques de même type.

Vectorisation des interruptions

Le contrôleur de priorité présenté dans le paragraphe précédent permet également de traiter le problème restant : transmettre, lors de la prise en compte de la requête par l'UC (INTA), l'information sur le périphérique ayant émis la requête. En fait, l'UC a besoin de connaître l'adresse de début de la procédure de traitement de la requête. Lors de la réception de INTA, le contrôleur va transmettre à l'UC, via le bus donnée, une information correspondant au numéro de la requête d'interruption qui sera, directement ou à travers une transformation simple, l'adresse mémoire du pointeur qui contient l'adresse de début de la procédure de traitement. Ce mécanisme est appelé *vectorisation des interruptions*. Les adresses de début des procédure de traitement constituent un vecteur d'interruption situé dans une zone fixe de la mémoire. Le contrôleur d'interruption transmet à l'UC l'indice du vecteur permettant de traiter une interruption particulière.

9.6 Interruptions et exceptions

Dans cette partie, on considère le problème plus général des situations exceptionnelles, qui remettent en cause le fonctionnement normal de l'UC lorsqu'elle exécute un programme.

Phase de l'instruction	Situation exceptionnelle
Lecture Instruction	Défaut de page Violation de protection mémoire Accès mémoire non aligné
Décodage de l'instruction	Code opération illégal ou non défini
Accès Mémoire	Défaut de page Violation de protection mémoire Accès mémoire non aligné
Exécution de l'Instruction	Erreur arithmétique

Table 9.1: Situations exceptionnelles lors de l'exécution d'une instruction

9.6.1 Les situations exceptionnelles

Les situations exceptionnelles peuvent provenir d'un événement interne à l'exécution du programme ou d'un événement externe, indépendant de l'exécution du programme en cours, typiquement une requête en provenance d'un contrôleur d'E/S. Les situations exceptionnelles ont des noms différents selon les constructeurs de machines : on peut trouver les termes d'exceptions, fautes, interruptions. On utilisera dans la suite le terme d'interruption pour désigner toutes ces situations exceptionnelles et leur traitement.

Situations internes

Les situations exceptionnelles internes peuvent provenir, soit de problèmes insurmontables, soit d'un requête du programme. Dans le premier cas, l'instruction qui provoque l'exception est appelée instruction *fautive* (faulting) Les principaux problèmes insurmontables sont liés aux étapes du pipeline comme décrit en table 9.1.

.

Défaut de page

Il y a défaut de page lors d'un accès mémoire où l'adresse référencée n'est pas dans une page physiquement présente dans la mémoire principale. Il faut donc initialiser un transfert de la page du disque vers la mémoire principale. Après ce transfert, l'exécution de l'instruction fautive peut redémarrer.

Violation de protection mémoire

L'accès mémoire demandé n'est pas autorisé, compte tenu des droits d'accès indiqués dans la table des pages. Dans ce cas, la violation de protection mémoire est signalée, et il ne peut y avoir reprise de l'instruction fautive.

Accès mémoire non aligné

Si l'architecture matérielle impose l'alignement, l'instruction qui effectue un accès non alignée ne peut être exécutée. En revanche, si l'accès mémoire non aligné est admis (x86), l'accès mémoire non aligné n'est pas une situation exceptionnelle.

Codes opération illégaux ou non définis

Lorsque PC contient un code qui ne correspond pas à une instruction, la partie contrôle ne peut décoder l'instruction, qui est irrécupérable. la cause la plus fréquente, dans un programme compilé, est une erreur de programmation qui conduit le compteur de programme à pointer sur une zone de données. Une situation exceptionnelle du même type est la tentative d'exécution d'une instruction protégée en mode normal.

Erreurs arithmétiques

On a vu au chapitre 3 que les erreurs arithmétiques ne constituent pas nécessairement une situation exceptionnelle. En particulier, dans la norme IEEE 754 pour le traitement des flottants, le comportement par défaut est qu'il ne s'agit pas d'une situation exceptionnelle. Mais la norme recommande que l'erreur puisse être considérée comme une situation exceptionnelle. Le choix relève de l'environnement de programmation (options de compilation) ou d'appels de librairies système par l'utilisateur.

Génération par programme

Le programme utilisateur fait appel au mécanisme de gestion des situations exceptionnelles pour effectuer des tâches particulières. L'utilisateur appelle une procédure du système d'exploitation à l'aide d'une instruction machine appelée interruption logicielle, par exemple l'instruction `sc` (system call) en PowerPC.

Situations externes

Les situations externes sont également de natures diverses, depuis l'impossibilité de fonctionnement, jusqu'à la demande d'intervention à partir d'organes externes : contrôleurs de périphériques, autres processeurs. *Impossibilité de fonctionnement*

La coupure d'alimentation, résultant par exemple d'une coupure d'alimentation secteur, ou une panne matérielle de l'UC, sont des causes d'impossibilité de fonctionnement. Si la panne matérielle est sans issue, sauf cas de systèmes possédant des opérateurs redondants et susceptibles de reconfiguration dynamique, la coupure d'alimentation est une situation exceptionnelle qui peut être reconnue et traitée si la machine utilisée possède des mémoires à accès aléatoires (RAM) non volatiles. Il est alors possible, dans la courte période entre la détection de la panne, et le moment où les circuits ne sont plus opérationnels parce que l'alimentation n'est plus suffisante, d'exécuter les quelques centaines d'instructions pour sauvegarder en mémoire non volatile l'essentiel du contexte en cours d'exécution.

La remise à zéro de l'UC est un cas particulier de situation externe de ce type. Il y a impossibilité de fonctionnement ultérieur, puisque l'action d'une commande extérieure (RAZ) force la réinitialisation de l'UC. Cependant, cette situation est particulière puisqu'il n'y a pas traitement de la cause par le mécanisme général de traitement des interruptions, mais par le mécanisme spécifique de réinitialisation, par exécution d'une procédure de démarrage contenue dans une mémoire ROM spécifique.

Demande d'intervention externe

C'est une requête provenant de l'extérieur de l'UC, et nécessitant un certain traitement : les requêtes provenant des contrôleurs de périphériques pour gérer les entrées-sorties en sont un exemple typique ; mais ces requêtes externes peuvent aussi provenir d'autres UC, dans un contexte multiprocesseurs, et sont un moyen général de dialogue entre différents maîtres.

Caractéristiques

Redémarrage ou arrêt

Lorsque le l'interruption permet de résoudre le problème, l'instruction ayant provoqué la faute est redémarrée. Pour permettre le redémarrage, il faut sauvegarder le contexte du programme interrompu, puis le restituer après traitement. Dans le cas où la situation exceptionnelle est irréparable, le programme doit être arrêté immédiatement ; cependant le contexte peut aussi être sauvegardé dans ce cas, pour autoriser l'écriture sur un fichier (`core` dans le contexte UNIX) de l'état du processeur et de la mémoire ; ce fichier peut être utilisé par un déboggueur pour identifier la cause de l'erreur.

Synchrone ou asynchrone

Lorsque la situation exceptionnelle résulte de l'exécution d'une instruction du programme, elle intervient de manière synchrone par rapport à l'horloge du processeur. Par contre, elle survient de manière asynchrone lorsqu'elle est provoquée par un événement extérieur, totalement indépendant de l'horloge du processeur.

Situation	Redémarrage ou Arrêt	Synchrone ou asynchrone	Masquable par l'utilisateur	Prise en compte
Défaut de page	Redémarrage	S	NM	Pendant
Violation mémoire	Arrêt	S	NM	Pendant
Accès non aligné	Arrêt	S	M	Pendant
Code op. illégal	Arrêt	S	NM	Pendant
Erreur arithmétique		S	M	Pendant
Appel système	Redémarrage	S	NM	Entre
Alimentation	Arrêt	A	NM	Pendant
Requête E/S	Redémarrage	A	NM	Entre

Table 9.2: Caractéristiques des situations exceptionnelles

Interruption masquable ou non par l'utilisateur

Certaines situations exceptionnelles doivent absolument être traitées, par exemple un défaut de page. D'autres peuvent être ignorées ou traitées, au choix de l'utilisateur : c'est le cas des erreurs arithmétiques lorsqu'elles sont considérées comme des situations exceptionnelles.

Occurrence de l'interruption entre instructions ou pendant une instruction

Cette caractéristique distingue les situations exceptionnelles qui interviennent pendant l'exécution d'une instruction et empêchent la poursuite de l'exécution, des interruptions qui interviennent entre les instructions. Ces dernières provoquent, soit un traitement sur le programme en cours comme le déboggage en pas à pas, soit un traitement sur une requête extérieure qui intervient entre deux instructions du programme en cours.

9.6.2 Gestion des situations exceptionnelles

La gestion des interruptions comporte trois étapes :

- prise en compte de l'interruption
- sauvegarde du contexte
- traitement de l'interruption

Le traitement de l'interruption consiste à exécuter une procédure spécifique, permettant ensuite de redémarrer l'exécution de l'instruction "fautive" dans le cas d'exceptions permettant le redémarrage après traitement, ou d'informer l'utilisateur, dans le cas d'exceptions fatales. Les procédures de traitement d'interruption sont généralement accédées via une table de pointeurs, situés dans une zone définie de la mémoire, réservée au système d'exploitation, et accessible via le mode d'adressage direct. Un numéro est associé à chaque cause d'interruption, qui permet d'accéder à une entrée de la table des pointeurs. Les pointeurs contiennent l'adresse de début des procédures de traitement des interruptions.

Prise en compte des interruptions

La prise en compte des interruptions est plus délicate. : à quel moment sont testés les cas possibles d'interruptions ? Comment prendre en compte les requêtes extérieures asynchrones par rapport au fonctionnement du processeur ? Que faire en présence de causes multiples d'interruptions ? Quelle priorité définir ?

Pour un processeurs non pipeliné, les causes d'interruption possibles sont examinées à la fin de chaque instruction. Le processeur examine les causes possibles dans l'ordre de priorité décroissante. En examinant les éventualités d'interruption à un moment précis (fin d'instruction), le processeur réalise une synchronisation de fait des requêtes d'E/S qui interviennent de manière asynchrons.

Pour un processeur pipeliné, plusieurs instructions s'exécutent simultanément, donc l'ordre des interruptions n'est pas nécéssairement le même que celui des instruction. Par exemple, considérons deux instructions successives :

I1	LI	DI	EX	MEM	RR	
I2		LI	DI	EX	MEM	RR

I2 peut produire une interruption de défaut de page dans l'étage LI, avant que I1 produise une interruption d'erreur arithmétique dans la phase EX.

Deux options sont possibles.

- La prise en compte des interruptions dans l'ordre séquentiel des instructions : toute interruption d'une instruction i est traitée avant une interruption résultant d'une instruction i+1. Il suffit de tester l'éventualité d'une interruption dans une phase donnée, comme le début de la phase Rangement du résultat qui ne peut pas provoquer d'interruption.

- L'autre approche consiste à prendre en compte les interruptions dans l'ordre où elles apparaissent, ce qui signifie qu'une interruption liée à une instruction i+1 peut être traitée avant l'interruption i. Le système d'exploitation doit reconnaître l'instruction qui a interrompu et, après traitement, faire redémarrer l'instruction qui a provoqué l'interruption.

La plupart des processeurs pipelinés traitent les interruptions dans l'ordre séquentiel des instructions.

D'autre part, les requêtes externes asynchrones doivent donc être testées à chaque cycle d'horloge, et non plus à la fin de chaque instruction.

Sauvegarde et restitution du contexte

Après prise en compte de l'interruption, le processeur doit sauvegarder le contexte du programme interrompu. Les questions essentielles sont : que doit on sauvegarder, comment et où sauvegarder ?

Avec les processeurs non pipelinés, le contexte à sauvegarder est relativement simple : il comprend le compteur de programme, le mot d'état du processeur, et l'ensemble des registres contenant des informations significatives du programme en cours d'exécution.

L'exécution simultanée de plusieurs instructions rend difficile le gel du pipe-line dans un état précis, notamment compte tenu des instructions arithmétiques flottantes dont la partie exécution prend plusieurs cycles. Si le processeur permet que les instructions se terminent dans un ordre différent de l'ordre où elles ont commencé, le problème est encore plus compliqué. On appelle *interruptions précises* la situation où il est possible de geler l'état du pipe-line dans un état précis, de telle sorte que toutes les instructions avant l'instruction ayant causé l'interruption soient exécutées jusqu'à la fin, et l'instruction ayant provoqué l'interruption et les suivantes soient redémarrées si l'interruption le permet. On appelle *interruptions imprécises* la situation où il est impossible de geler le pipe-line dans un état précis, mais où l'on possède suffisamment d'informations pour redémarrer l'exécution des instructions.

Interruptions multiples et priorités

Nous avons signalé l'existence de nombreuses causes possibles d'interruptions, provenant soit d'exceptions internes au processeur, soit de requêtes externes. Ces interruptions peuvent intervenir simultanément ou de façon imbriquée. Le problème est relativement simple en ce qui concerne l'ordre de traitement des interruptions intervenant simultanément, en affectant à chaque type d'interruption une priorité. A chaque cycle d'horloge, le processeur examinant les l'existence ou non d'exceptions dans l'ordre des priorités décroissantes, l'exception la plus prioritaire est forcément traitée en premier.

L'attitude à adopter lorsqu'une exception intervient en cours de traitement d'une interruption est moins évident. Faut-il ou non interrompre la procédure de traitement en cours ? Les stratégies sont variables selon les processeurs. La technique la plus simple consiste à doter le processeur d'instructions d'autorisation (ex STI pour Set Interrupt) et d'inhibition (ex : CLI pour Clear Interrupt) des interruptions. Avec ces instructions, il est toujours possible d'interdire la prise en compte d'interruptions dans une section critique insérée entre les deux instructions CLI et STI dans une procédure de traitement des interruptions. Une section de procédure de traitement de haute priorité ne pourra donc être interrompue par une exception de priorité plus faible. La technique la plus élaborée consiste à doter le processeur d'un certain nombre de niveaux de priorité codés sur un certain nombre de bits. Dans l'architecture SPARC, le niveau de priorité est codé sur 4 bits, de 0000 (pas de requête d'interruptions) à 1111 (interruption non masquable). Avec cette stratégie, le processeur a à tout instant un certain niveau de priorité correspondant à la priorité de la tâche en cours d'exécution (Dans le CY7C101 qui est une implémentation de l'architecture SPARC, ce niveau est contenu dans le champ Processor Interrupt Level du registre d'état du processeur). Lorsqu'il n'y a aucune requête, le processeur travaille au niveau 0. Lorsqu'une exception ou une requête externe intervient, le niveau de requête est comparé au niveau d'exécution en cours. Si la requête est de niveau supérieur, elle est prise en compte et le processeur passe à ce niveau de priorité. Lorsque la procédure de traitement est terminée, le processeur exécute la requête en attente de niveau immédiatement inférieur et passe à ce niveau de priorité. Lorsqu'une requête est de niveau inférieur au niveau correspond à l'exécution en cours, elle reste en attente, et sera prise en compte lorsque le processeur aura terminé toutes les exécutions de priorités supérieures.

Bibliographie

Avertissement La bibliographie qui suit comprend deux parties. La première correspond à des ouvrages qui sont accessibles au niveau de ce cours. La seconde contient des articles de synthèse publiés dans des revues spécialisées de niveau recherche. Ils ne sont donc pas accessible à tous les étudiants et ne sont cités que pour information.

Première partie

[1] *ARIANE 5 Flight 501 Failure, Report by the Inquiry Board.* http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html

[2] http://www.asc-inc.com/ascii.html

[3] D. Etiemble. *Architecture des microprocesseurs RISC.* Dunod, collection A2I, 92

[4] D.Etiemble. *Les mémoires à semi-conducteurs : composants et organisation.* Coll. Les techniques de l'Ingénieur.

[5] D. Etiemble. *Fondements du matériel.* Polycopié de Licence d'Informatique.

[6] D. Patterson, J. Hennessy. *Organisation et conception des ordinateurs.* Dunod 94 Traduction de *Computer Organization and Design : The Harware/Software Interface*, Morgan-Kauffman 94.

[7] J. Hennessy, D. Patterson. *Architecture des ordinateurs : une approche quantitative, Deuxième édition.* Thompson Publishing France, 96. Traduction de *Computer Architecture. A Quantitative Approach.* McGrawHill 96

[8] http://unicode.org

[9] Mary Lou Nohr. *Understanding ELF Objetcs Files and Debugging Tools.* Prentice Hall, 94.

Deuxième partie

[10] *IEEE Transactions on Computers.* Octobre 96.

[11] V. Cuppu et al. A Performance comparison of contemporary DRAM architectures. *Proc. 26th ISCA.* Mai 99.

[12] D. Goldberg. What every computer scientist should know about floating point arithmetics. *ACM Computing Surveys*, 23:1, pp 5-48, Mars 91.

[13] BLAS (Basic Linear Algebra Subprograms) http://www.netlib.org/blas/index.html

[14] B.L. Jacob et T.N. Mudge. Virtual Memory Issues of Implementation. *IEEE Computer* 31:6, pp 33-43, Juin 98.

[15] B.L. Jacob et T.N. Mudge. Virtual Memory in Contemporary Microprocessors. *IEEE Micro* 18:4, pp 60-75 , Juillet 98.

[16] D.Patterson et al. A case for intelligent RAM : IRAM. *IEEE Micro.* Avril 97.