

4. Les Processeurs actuels.

Pipelining. Prédiction de branchement. Caches. Processeurs superscalaires

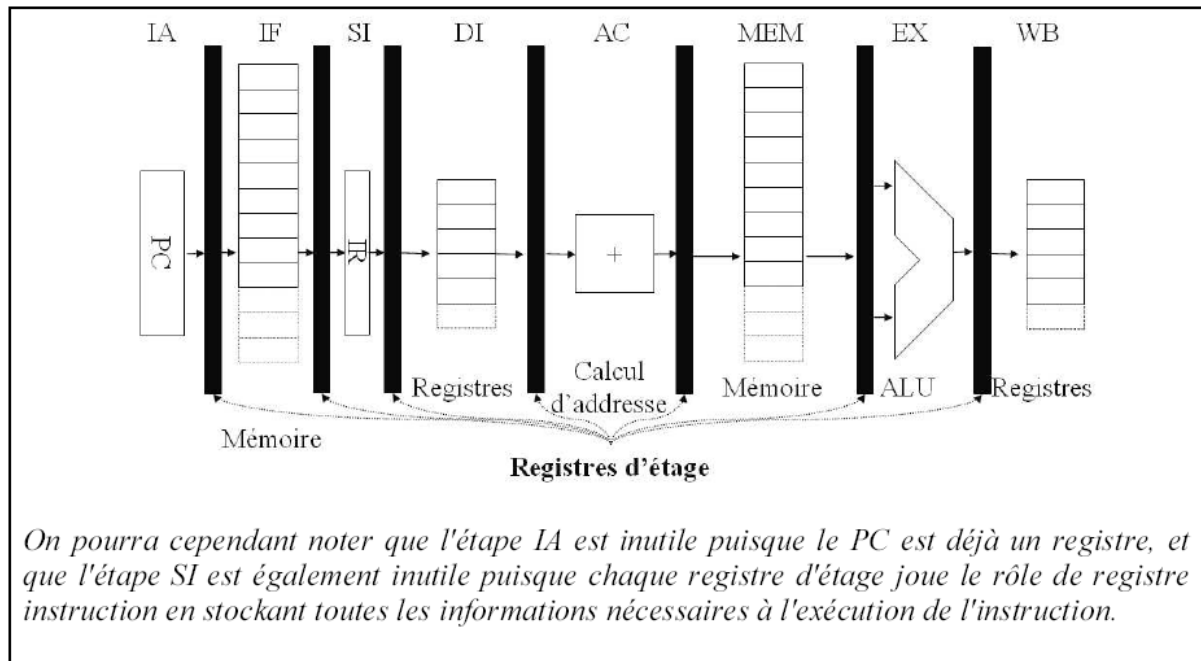
1. Pipelining

L'exécution d'une instruction est décomposée en une succession d'étapes, et chaque étape correspond à l'utilisation d'un des composants du processeur. Lorsqu'une instruction se trouve dans l'une des étapes, les composants associés aux autres étapes ne sont pas utilisés. Le fonctionnement d'un processeur simple est donc inefficace. L'exécution pipelinée des instructions permet d'améliorer l'efficacité d'un processeur : on charge une première instruction dans la première étape, et au cycle suivant, cette instruction passe dans la seconde étape, on charge alors une seconde instruction dans la première étape, et ainsi de suite... En régime permanent, il peut y avoir une instruction en cours d'exécution dans chacune des étapes, et donc chacun des composants du processeur peut être utilisé à chaque cycle, l'efficacité est maximale. Le temps d'exécution d'une instruction n'est pas réduit, en revanche, le débit de sortie des instructions est considérablement augmenté : jusqu'à une instruction exécutée par cycle ; sur un processeur comportant cinq étapes d'exécution, pipeliner l'exécution permet donc de multiplier la performance par cinq.

Implémentation du pipeline. Pour obtenir une version pipelinée d'un processeur comme le LC-2, on peut physiquement découper le processeur en étape/étage en séparant les composants les uns des autres à l'aide de registres, appelés **registres d'étage**. Chaque composant traitant une instruction différente des autres composants, il doit disposer de l'ensemble des informations nécessaires à l'exécution de l'étape pour cette instruction (opcode, bits de contrôle, opérandes...). Le contrôle devient alors décentralisé au niveau de chaque étage. Pipeliner un processeur ayant pour but d'accroître sa performance, on utilise un contrôle câblé plutôt qu'un contrôle microprogrammé.

Exemple. Pour le LC-2, on peut par exemple découper l'exécution d'une instruction en 8 étapes, comme indiqué ci-dessous :

1. Envoi adresse instruction (IA)
2. Chargement instruction (IF)
3. Stockage instruction (SI)
4. Décodage; lecture des opérandes (DI)
5. Calcul d'adresse (AC)
6. Accès mémoire (ME)
7. Exécution (EX)
8. Ecriture du résultat (WB)



Dans les processeurs récents, l'exécution pipelinée a un autre avantage : puisqu'il est possible de découper l'exécution d'une instruction en étapes sans affecter la performance, bien au contraire, on peut réduire la durée d'une étape en augmentant le nombre d'étapes. Cela permet de réduire le temps de cycle du processeur, et donc d'augmenter le débit de sortie des instructions. On va voir cependant que plus le pipeline est long, et plus le nombre de cas où il n'est pas possible d'atteindre la performance maximale est élevé.

Il existe trois principaux cas où la performance d'un processeur pipeliné peut être dégradée ; ces cas de dégradation de performance sont appelés des **aléas** (*pipeline hazard*) :

- aléas structurels
- aléas de données
- aléas de contrôle

Lorsqu'un aléa se produit, cela signifie qu'une instruction ne peut continuer à progresser dans le pipeline. Pendant un ou plusieurs cycles, l'instruction va rester bloquée dans un étage du pipeline, mais les instructions situées plus en avant pourront continuer à s'exécuter jusqu'à ce que l'aléa ait disparu. Les étages du pipeline vacants sont appelés des « bulles » de pipeline, une bulle correspondant en fait à une instruction NOP (*No OPeration*). En pratique, le registre d'un étage bloqué émet effectivement une instruction NOP à la place de l'instruction bloquée.

Exemple. On suppose que Instr2 est bloquée pendant deux cycles à cause d'un aléa.

	IA	IF	SI	DI	AC	MEM	EX	WB
Inst 1								
Inst 2					●	●		

Aléas structurels. Un aléa structurel correspond au cas où deux instructions ont besoin d'utiliser la même ressource du processeur.

Exemple. Dans le pipeline du LC-2, il peut y avoir un aléa structurel entre une instruction load/store et une instruction quelconque en cours de chargement : en effet, l'instruction load/store utilise la mémoire pour lire ou écrire une donnée, tandis que l'autre instruction doit également être chargée depuis la mémoire.

Inst.	0	1	2	3	4	5
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM
ADD R1,R1,#5		IA	IF	SI	DI	AC
STR R1,R0,#30			IA	IF	SI	DI
ADD R0,R0,#1				IA	IF	SI
ADD R3,R0,R2					IA	IF

Conflit de ressource

On peut résoudre cet aléa en utilisant une mémoire séparée pour les données et les instructions ; dans les processeurs actuels, c'est ce que l'on fait à l'aide des caches : il y a un cache pour les données et un cache pour les instructions.

Aléas de données. Un aléa de données intervient lorsqu'une instruction produit un résultat, et qu'une instruction suivante utilise ce résultat avant qu'il ait pu être écrit dans le banc de registres.

Exemple.

LDR R1 ← R0, #30
ADD R1 ← R1, #5

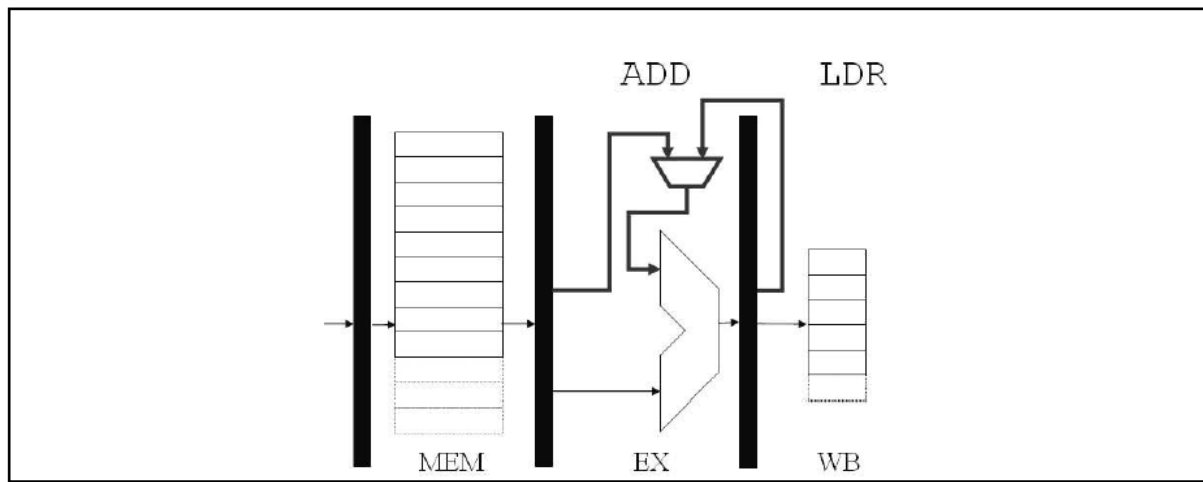
→ R₁ contient la valeur attendue par LDR

Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM	EX	WB					
ADD R1,R1,#5		IA	IF	SI	●	●	●	●	DI	AC	MEM	EX	WB

L'instruction ADD ne peut effectuer son étage DI (pendant lequel on récupère les registres sources dans le banc de registres), le registre R1 produit par l'instruction LDR n'étant disponible qu'à la fin de l'étage WB de cette instruction.

Cependant, il est souvent inefficace d'attendre qu'une donnée ait été écrite dans le banc de registres avant de l'utiliser. En effet, la donnée est souvent disponible plus tôt ; ainsi, la donnée est disponible à la fin de l'étage MEM dans l'exemple ci-dessus, et l'instruction ADD a réellement besoin de la donnée au début de l'étage EX seulement. On peut donc réduire l'impact des aléas de données en créant des chemins supplémentaires entre les différents étages du pipeline afin de passer les données plus rapidement d'une instruction à l'autre. Ce mécanisme s'appelle le **forwarding**. Dans l'exemple ci-dessus, il permet d'éliminer les quatre cycles de gel du pipeline.

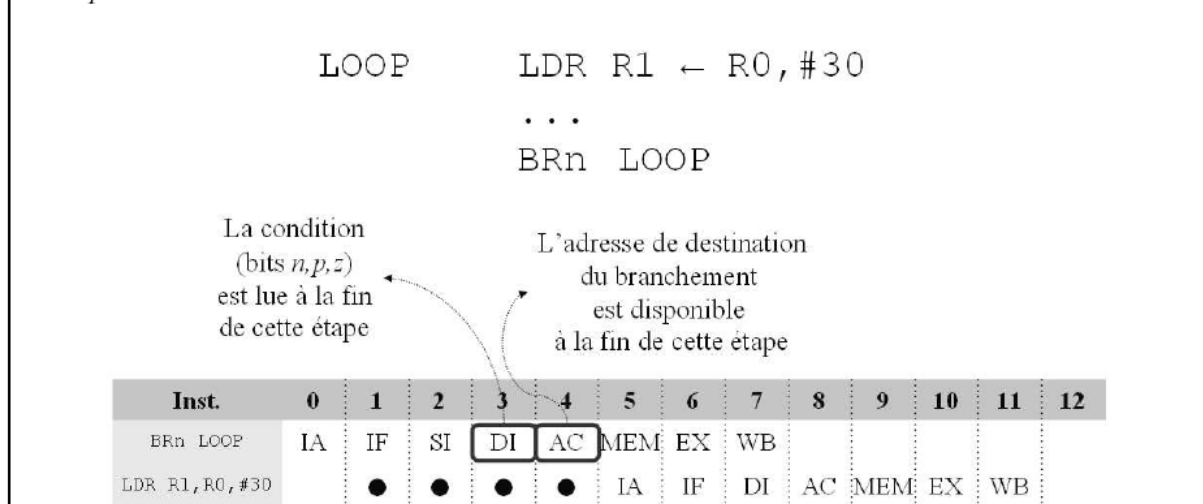
Exemple. Chemin de données à rajouter dans l'exemple ci-dessus pour implémenter le forwarding.



Aléas de contrôle. Un aléa de contrôle se produit à chaque fois qu'une instruction de branchement est exécutée. Lorsqu'une instruction de branchement est chargée, il faut normalement attendre de connaître l'adresse de destination du branchement pour pouvoir charger l'instruction suivante. Or, cette information n'étant en général connue que plusieurs cycles après le chargement de l'instruction, il est nécessaire de bloquer le pipeline.

Dans le cas d'un branchement conditionnel, il faut également connaître la valeur de la condition (branchement pris ou non pris).

Exemple. Aléa de contrôle dans le LC-2.

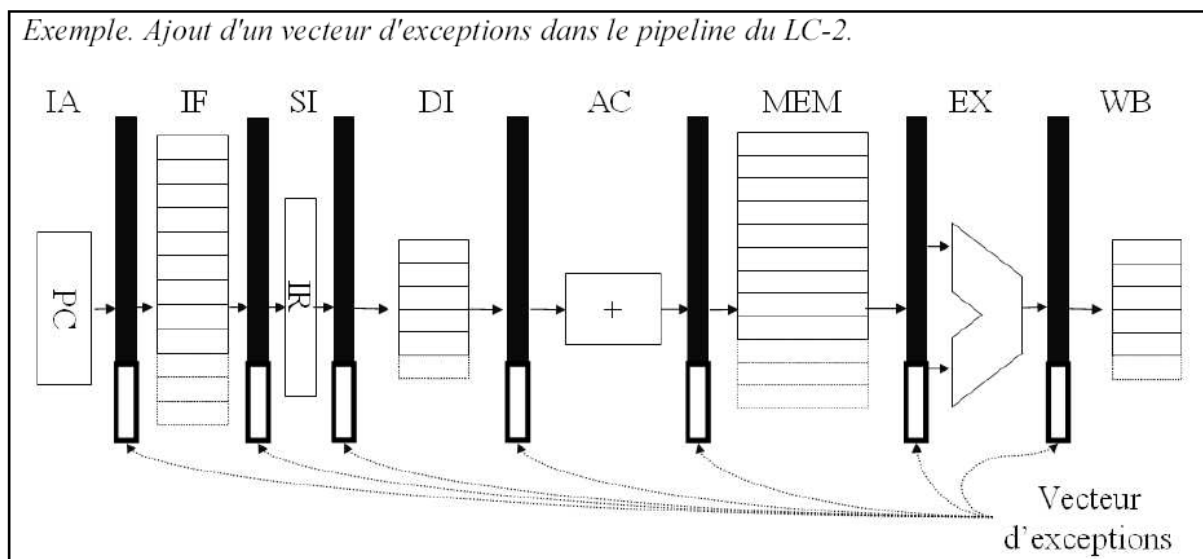


Pipeline et exceptions. Lorsque le programme doit s'interrompre brutalement (faute dans l'exécution du programme ou interruption provoquée par le système d'exploitation), il faut pouvoir éventuellement redémarrer le programme précisément à l'endroit où il s'est arrêté, c'est-à-dire exactement à l'instruction assembleur où l'exception s'est produite ; cette propriété du couple architecture/système d'exploitation est appelée une « exception précise ». Or, dans un processeur pipeliné, lorsqu'une exception intervient pour une instruction dans l'une de ses étapes d'exécution, il y a souvent plusieurs instructions précédentes qui n'ont pas terminé leur exécution. Si le programme est redémarré à l'instruction qui a provoqué l'exception, son exécution sera incorrecte puisque les instructions précédentes n'auront pas été exécutées entièrement.

Il est donc nécessaire de retarder la gestion des exceptions jusqu'au moment où l'on sait que toutes les instructions précédentes ont terminé de s'exécuter. Pour ce faire, on ajoute aux registres d'étage un « vecteur d'exception » qui contient un bit par exception possible. Lorsqu'une instruction produit une exception, elle met à 1 le bit correspondant, et ne peut plus modifier l'état du processeur (ni écrire en mémoire, ni écrire dans le banc de registres), i.e., elle est transformée en NOP. L'exception ne sera traitée que lorsque l'instruction arrive dans l'étage final, e.g., WB dans le LC-2. À ce moment-là, les instructions précédentes auront terminé de s'exécuter.

En pratique, la présence d'instructions multi-cycles, comme les instructions de calcul flottant, rend l'implémentation des exceptions précises encore plus complexe.

Exemple. Ajout d'un vecteur d'exceptions dans le pipeline du LC-2.

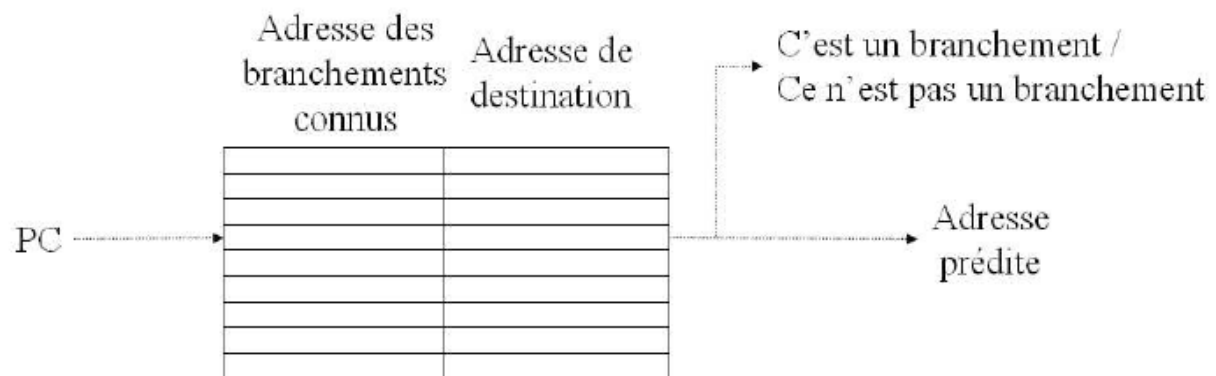


2. Prédiction de branchement

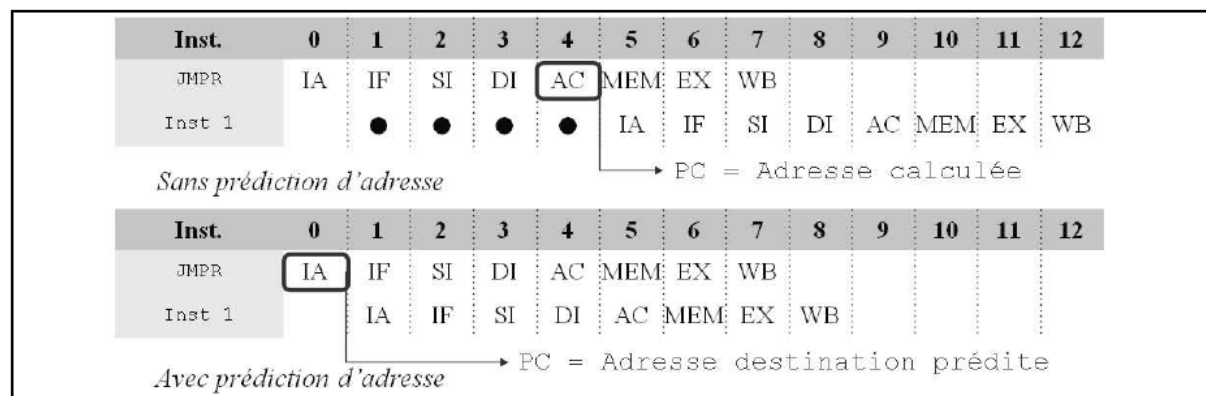
En raison de l'allongement de la taille du pipeline (liée à l'augmentation de la fréquence), le délai entre le moment où l'on charge une instruction de branchement et le moment où l'on connaît l'adresse de destination du branchement et la valeur de la condition tend à s'allonger considérablement (20 cycles sur le Pentium 4). Comme, en moyenne, il y a une instruction de branchement toutes les cinq instructions, la dégradation de performances induites par le délai de branchement serait intolérable. Aussi, depuis plusieurs années, les processeurs comportent des mécanismes de **prédiction de branchement**, destinés à prédire l'adresse de destination de branchement et la valeur de la condition pour les branchements conditionnels.

Prédiction d'adresse. La prédiction d'adresse est souvent la plus fiable parce que la plupart des instructions de branchement ont des adresses de destination fixe. Pour effectuer cette prédiction d'adresse, on ajoute au processeur une table (*Branch Target Buffer*) indexée par le PC des instructions et qui contient les adresses de destination des branchements. Lorsqu'on exécute une instruction de branchement, on stocke dans la table son adresse de destination à l'emplacement déterminé par son PC (approximativement, PC modulo la taille de la table). Lorsque l'on charge une instruction, i.e., que l'on envoie l'adresse contenue dans le PC à la mémoire, on envoie également le PC dans cette table. La table peut être lue très rapidement, contrairement à la mémoire, et en un cycle on sait si l'instruction en cours de chargement est un branchement, et si c'est le cas, on dispose également de son adresse de destination. Dès le cycle suivant, on peut donc envoyer à la mémoire l'adresse de l'instruction de destination et éviter tout gel du pipeline.

Ce mécanisme de prédiction fonctionne moins bien pour les instructions RETURN (RET dans le LC-2) car une procédure peut être appelée de plusieurs endroits différents, et l'adresse de destination de cette instruction de branchement est donc variable.

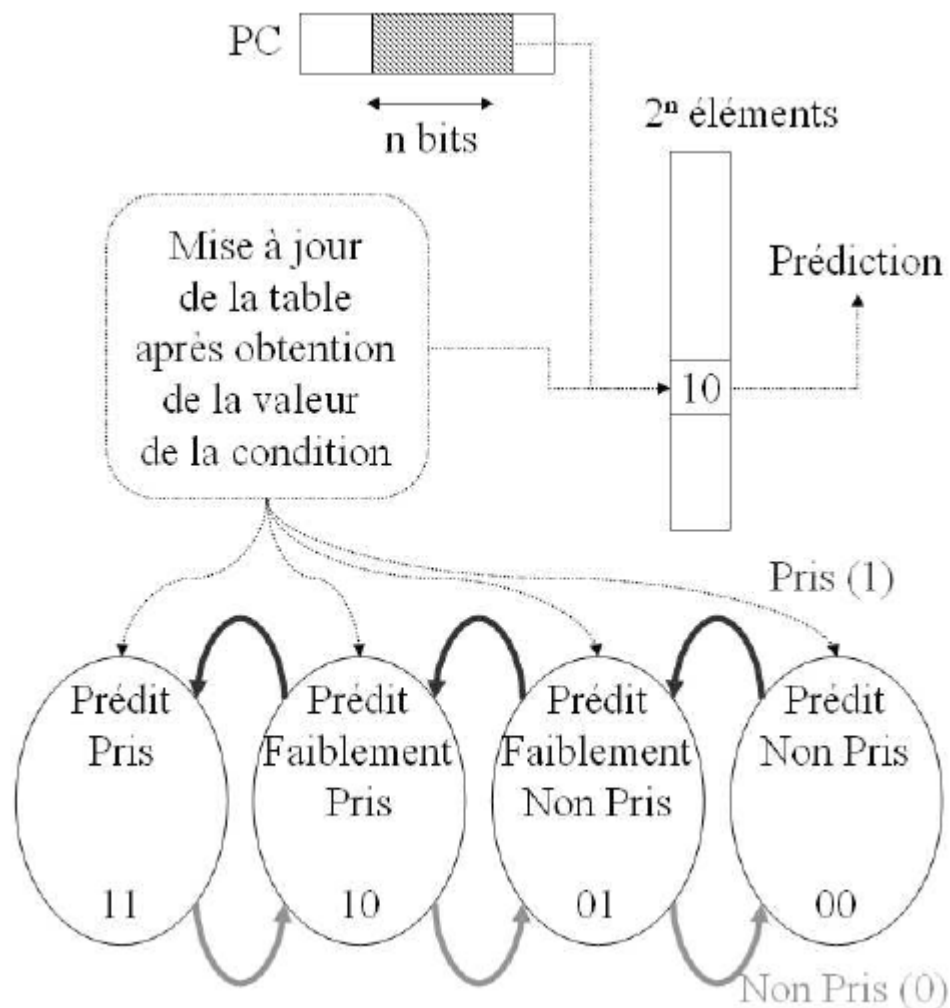


Exemple. Dans le cas du LC-2 pipeliné, la prédiction d'adresse permet d'éviter jusqu'à quatre cycles de pénalité.



Prédiction de condition. La prédiction de la condition d'un branchement conditionnel est nettement plus complexe et plus aléatoire parce que la valeur de la condition peut changer fréquemment d'une exécution du branchement à l'autre. Les mécanismes de prédiction existants ont pour but d'analyser le comportement du branchement pour détecter un éventuel comportement régulier. Les mécanismes de prédiction les plus simples effectuent une analyse uniquement locale. Comme pour la prédiction d'adresse, on dispose d'une table indexée par le PC de l'instruction de branchement. Chaque entrée de la table contient un automate à quatre états (2 bits) qui indique la prédiction courante : *pris*, *faiblement pris*, *faiblement non pris*, *non pris*. À chaque fois que ce branchement conditionnel est exécuté, une fois que sa condition est connue, l'automate est mis à jour.

Cette table est utilisée de la façon suivante. En plus de la table de prédiction d'adresse mentionnée ci-dessus, le PC de chaque instruction à charger est également envoyé à la table de prédiction de condition (*Branch Prediction Table*). On lit alors la valeur de la condition, et si la table de prédiction d'adresse indique que l'instruction en cours de chargement est un branchement conditionnel, on utilisera cette valeur pour déterminer si le branchement est prédit pris ou non pris.



Plus le pipeline du processeur est long, plus le délai de branchement est grand et plus la qualité de la prédiction de branchement doit être élevée. Pour améliorer la précision de la prédiction de branchement, les nouveaux prédicteurs ne se contentent pas d'analyser le comportement local du branchement mais également le comportement des branchements conditionnels précédemment exécutés. En effet, le comportement des branchements conditionnels est parfois corrélé au chemin du graphe de flot de contrôle suivi par le programme.

Exemple.

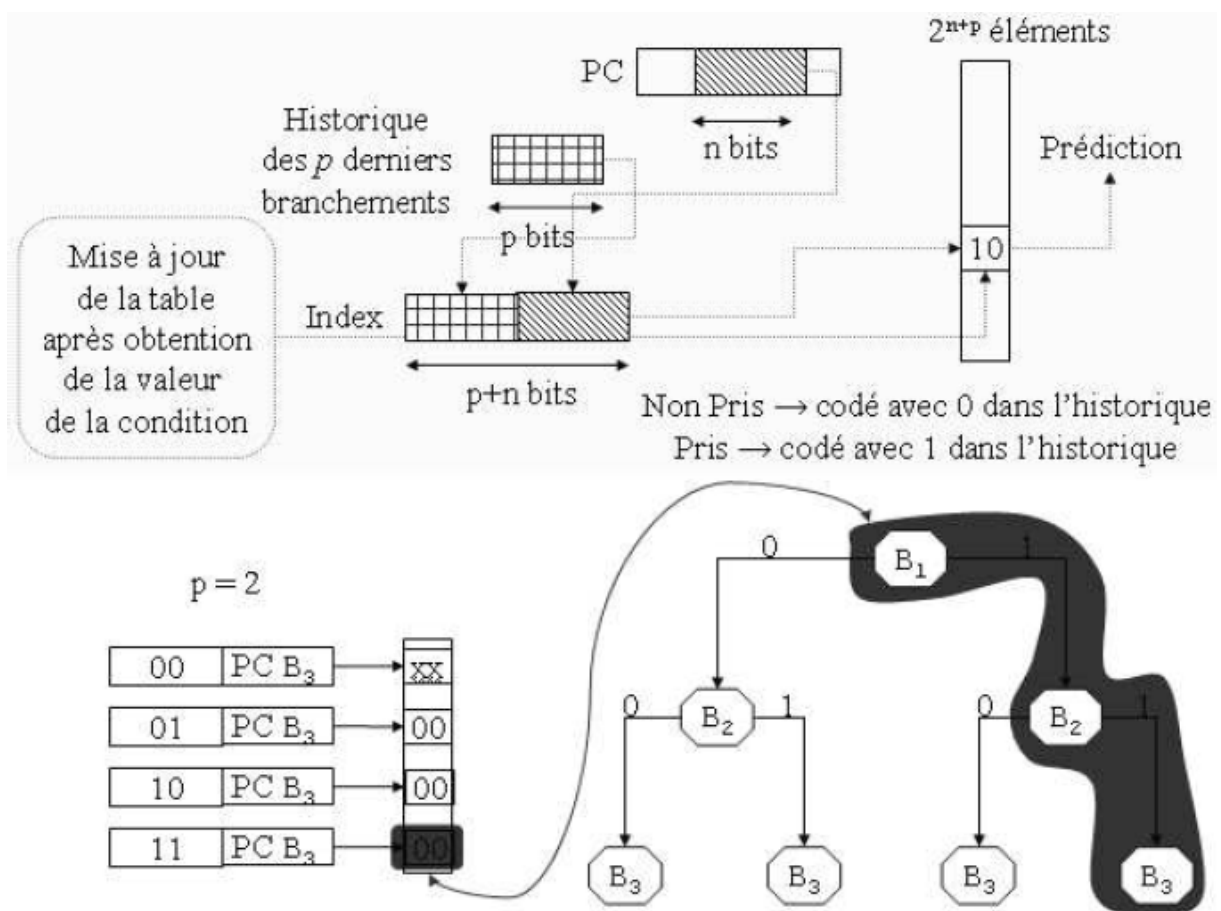
```

if (a == 1) a = 0;      /* Branchement B1 */
...
if (b == 0) b = 1;     /* Branchement B2 */
...
if (a == b) ...;       /* Branchement B3 */

```

La valeur de la condition du dernier branchement conditionnel est en partie déterminée par le comportement des deux branchements précédents.

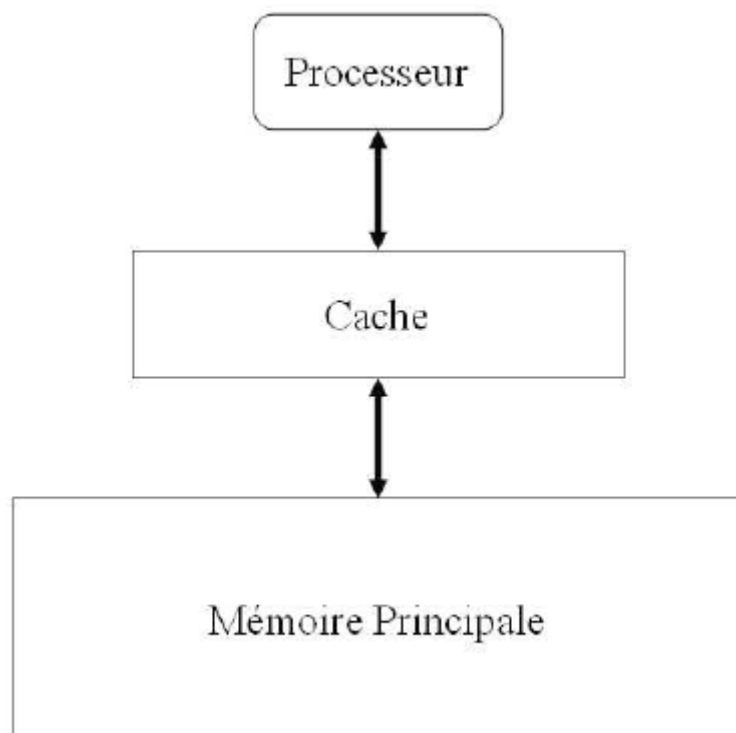
Pour ce faire, les processeurs intègrent un registre à décalage donnant la valeur de la condition des N derniers branchements conditionnels (*Branch History Register*). La table de prédiction de condition est alors indexée non pas seulement par le PC, mais par une concaténation du PC et de ce registre. Intuitivement, cela signifie que l'on distingue le comportement local du branchement selon le comportement des branchements conditionnels précédents.



3. Caches

Depuis de nombreuses années l'écart entre la performance du processeur et celle de la mémoire tend à s'accroître. Cet accroissement s'est accéléré avec l'avènement des processeurs RISC, plus à même d'exploiter rapidement l'évolution de la technologie. Les composants mémoire bénéficient des mêmes progrès de la technologie, mais le décodage de l'adresse et le chargement des lignes communes permettant de lire ou d'écrire les différentes cellules d'une mémoire sont des étapes difficiles à accélérer. En conséquence, le temps d'accès à la mémoire décroît moins vite que le temps de cycle du processeur. Aujourd'hui, charger une information depuis la mémoire requiert de plusieurs dizaines à plusieurs centaines de cycles selon les processeurs. Sachant qu'il y a en moyenne une instruction d'accès à la mémoire toutes les trois instructions, en l'absence de mécanismes permettant de masquer cette latence d'accès à la mémoire, la performance des processeurs serait beaucoup plus faible qu'elle ne l'est aujourd'hui.

Depuis le début des années 80, le principal mécanisme architectural permettant de masquer la latence de la mémoire est le **cache**. Le principe est de placer une mémoire rapide entre le processeur et la mémoire principale. Cette rapidité, liée à une technologie différente de la mémoire principale (SRAM au lieu de DRAM) et une taille réduite, s'accompagne d'un coût plus élevé (6 transistors au lieu de 1 par cellule). Le principe du cache est très simple : le processeur n'a pas conscience de sa présence et lui envoie toutes ses requêtes comme s'il s'agissait de la mémoire principale. Soit la donnée ou l'instruction requise est présente dans le cache et elle est alors renvoyée immédiatement au processeur, soit elle n'est pas dans le cache, et le contrôleur du cache envoie alors une requête à la mémoire principale, puis renvoie la donnée au processeur et la stocke en même temps dans le cache. Le premier cas s'appelle un **succès** de cache (*cache hit*), tandis que le second cas s'appelle un **défaut** de cache (*cache miss*). En cas de défaut, le cache n'apporte bien sûr aucun gain, la performance du cache est donc entièrement liée au taux de succès.



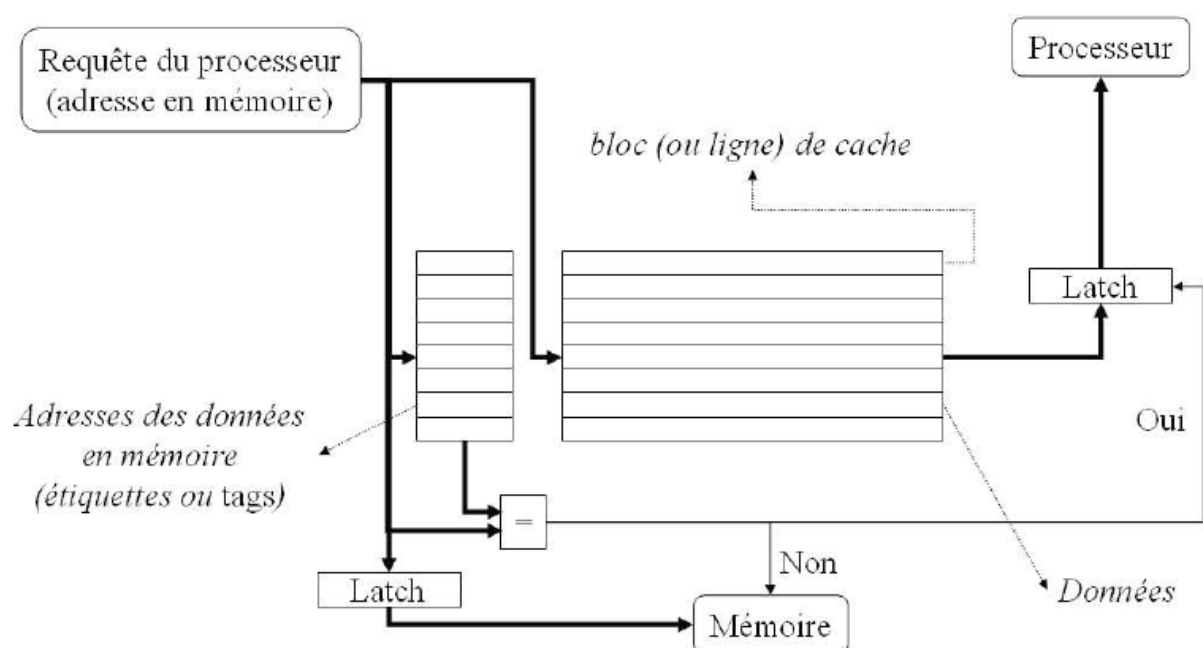
En pratique, on observe fréquemment des taux de succès moyens de l'ordre de 80 à 90 %. Cette performance s'explique par les fortes propriétés de **localité** des programmes. Il existe deux types de localités : la **localité temporelle**, et la **localité spatiale**. Une définition intuitive de la localité temporelle est la suivante : si une donnée située à une adresse A en mémoire est référencée, elle a une forte probabilité d'être référencée à nouveau dans un court intervalle de temps. Et pour la localité spatiale : si une donnée située à une adresse A en mémoire est référencée, il y a une forte probabilité de référencer une donnée située à une adresse voisine dans un court intervalle de temps.

Exemple. On peut considérer l'exemple du produit matrice-vecteur.

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        y[i] = y[i] + a[i][j] * x[j]
    }
}
```

- $y[i]$: propriétés de localités temporelle et spatiale.
- $a[i][j]$: propriétés de localité spatiale.
- $x[j]$: propriétés de localité temporelle et spatiale.

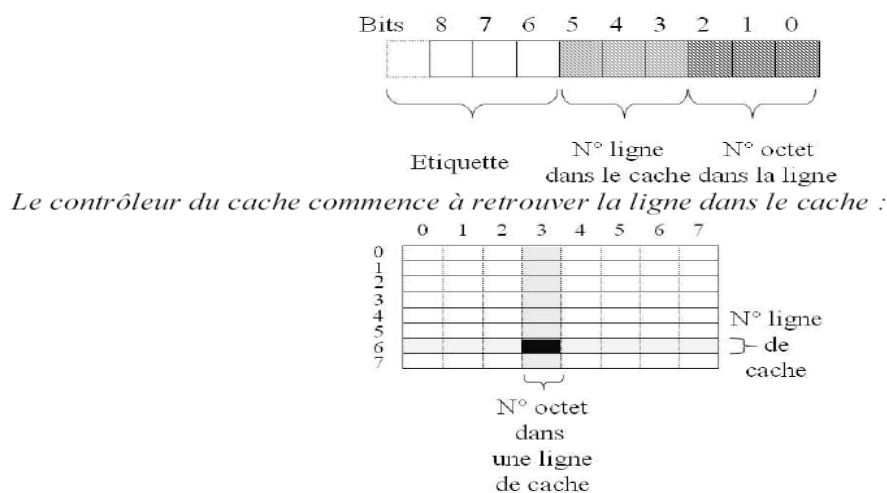
Implémentation du cache. La taille du cache étant nettement inférieure à celle de la mémoire, le cache ne pourra contenir qu'une partie des données de la mémoire. En conséquence, la position d'une donnée dans le cache ne correspond donc plus à son adresse, contrairement à ce qui se passe en mémoire principale. Avec chaque donnée stockée dans le cache, il est donc nécessaire de conserver également son adresse. Aussi, un cache comporte deux sous-composants principaux : la table des étiquettes et la table des données. La table des étiquettes stocke les adresses des données, et la table des données stocke bien sûr les données elles-mêmes.



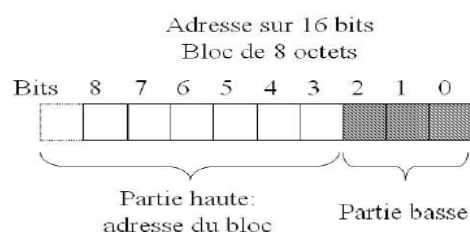
Lorsque le processeur envoie au cache l'adresse de la donnée requise, le contrôleur du cache compare cette adresse à celles stockées dans le banc d'étiquettes. Si la donnée est présente, elle est renvoyée au processeur et sinon il y a défaut de cache. Afin de réduire le temps de cette comparaison, le contrôleur de cache utilise une politique de placement simple : grossièrement, une donnée située à une adresse A est placée dans la case du cache $A \bmod \text{Taille du Cache}$. En d'autres termes, il existe un seul emplacement possible pour une donnée dans le cache, et il y a donc une seule comparaison à effectuer lors d'une requête du processeur (on verra que cette contrainte peut avoir un impact sur la performance du cache et on peut la relâcher au prix d'une complexité supérieure de l'implémentation du cache). La simple présence d'une donnée dans le cache permet d'exploiter la localité temporelle. Pour exploiter la localité spatiale, en cas de défaut de cache, on ne charge pas seulement la donnée requise mais cette donnée et plusieurs des données voisines. Les données chargées simultanément forment une **ligne** ou un **bloc** du cache (les deux termes sont utilisés indifféremment).

Plus précisément, lorsque le processeur fait une requête il envoie deux informations : l'adresse du premier octet de la donnée requise et le nombre d'octets à charger. L'adresse est alors décomposée en trois parties par le contrôleur du cache : les bits de poids faible correspondent à l'emplacement de l'octet à l'intérieur d'une ligne du cache ; les bits suivants donnent le numéro de la ligne du cache (l'index des tables) ; les bits de poids fort restants correspondent donc à l'adresse de la ligne (tout comme on avait défini la notion «d'adresse de page» pour le système exploitation), et ce sont ces bits qui sont stockés dans la table des étiquettes.

Exemple. On considère un cache de 64 octets agencé en huit lignes de huit octets chacune. L'adresse d'une requête du processeur se décompose ainsi :



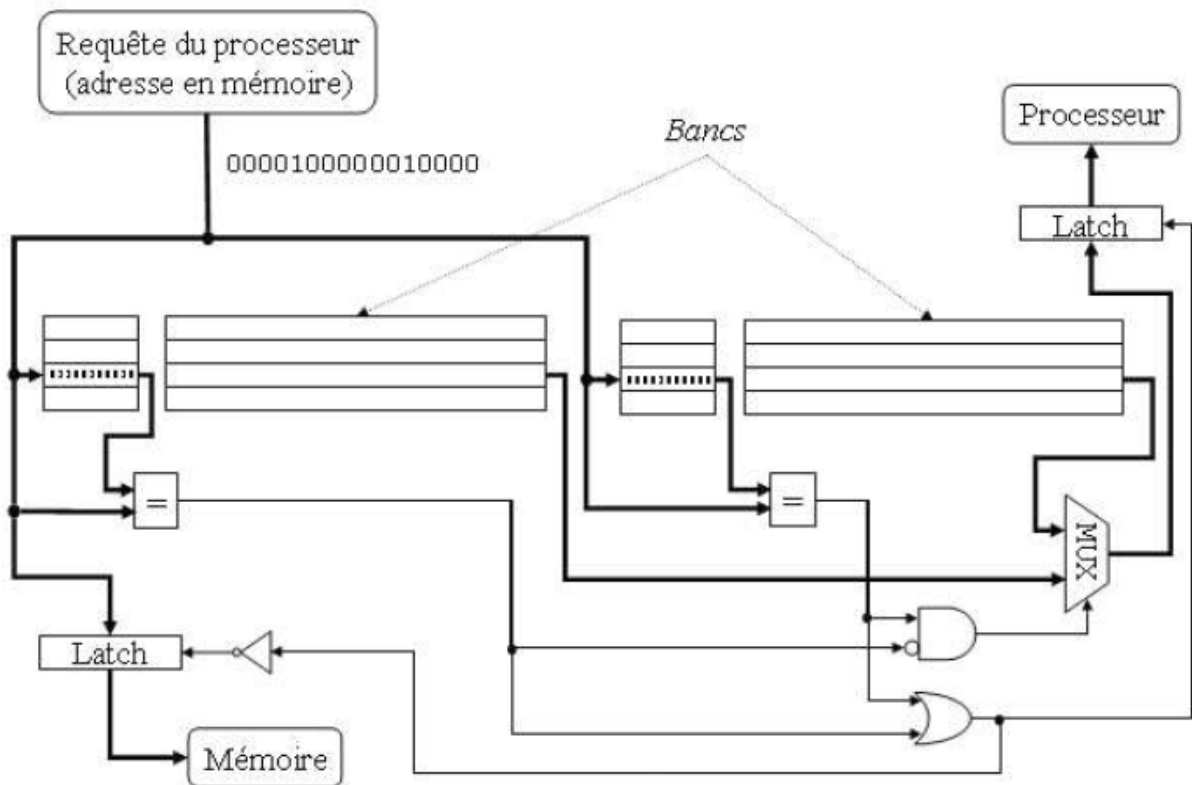
puis l'octet dans la ligne :



Exemple:

0...010100000	} Même ligne de cache
0...010100111	
0...010101000	} Lignes distinctes, adresses consécutives

Associativité. En raison de la politique de placement dans les caches, il peut arriver que deux données soient en compétition pour la même ligne de cache bien que les autres lignes ne soient pas utilisées. Il suffit pour cela que les bits de l'adresse permettant de déterminer la ligne du cache soient les mêmes pour les deux données. On parle alors de **conflit de cache**. Pour limiter le nombre de conflits de cache, plusieurs processeurs utilisent des caches **associatifs**. Dans un cache associatif, on divise la table des données et celle des étiquettes en n bancs. La politique de placement dans un banc est la même que dans un cache, mais une donnée peut se trouver dans n'importe où le quel des n bancs. On dit alors que le degré d'associativité du cache est n (*n-way associative cache*).

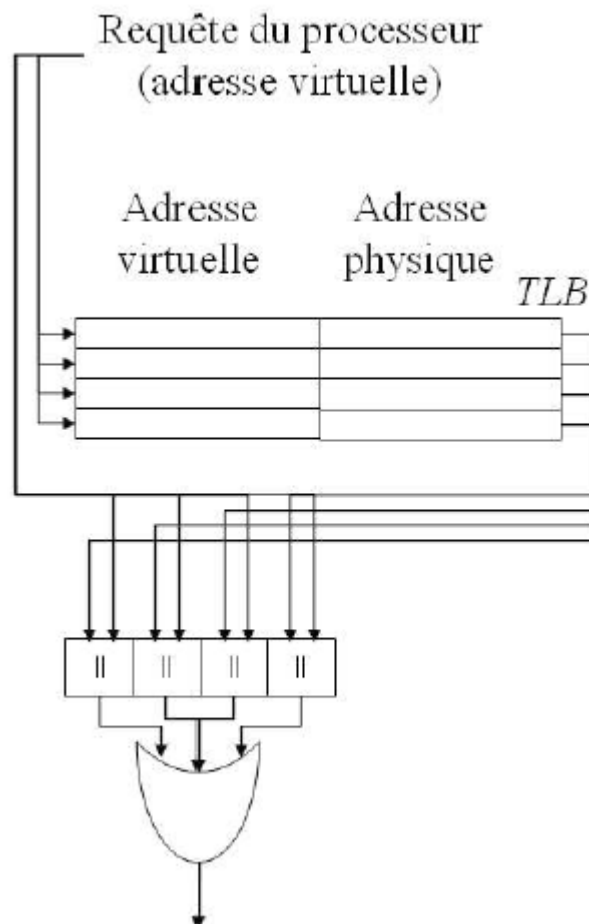


La politique de placement dans un cache est donc légèrement plus complexe : lorsque l'on charge une nouvelle ligne, il faut déterminer dans quel banc la placer. Les caches associatifs comportent donc une politique de remplacement destinée à choisir la donnée à éjecter du cache. Les politiques les plus classiques sont : *Random* (choix aléatoire), *LRU* (*Least Recently Used* : on choisit la ligne la plus anciennement utilisée), *Pseudo-LRU* (on n'éjecte pas la ligne la plus récemment utilisée, puis on fait un choix aléatoire entre les autres lignes).

Translation Lookaside Buffer (Cache de traduction d'adresses)

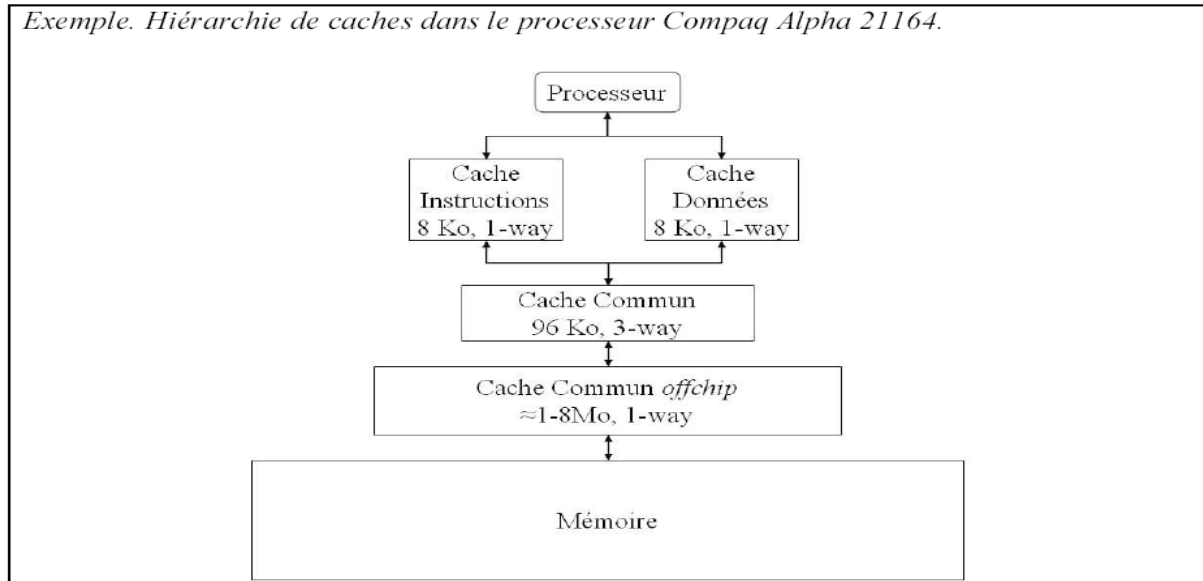
TLB. Le processeur utilisant des adresses virtuelles, et les données étant placées à des adresses physiques en mémoire principale, le processeur accède en général au cache à l'aide d'adresses virtuelles, tandis que les données peuvent être rechargées dans le cache à l'aide d'adresses physiques, i.e., les étiquettes correspondent à des adresses physiques.

Une façon astucieuse de procéder, utilisée dans plusieurs caches, est de dimensionner les bancs du cache de façon à ce qu'ils soient inférieurs à la taille d'une page. Les bits utilisés pour chercher le numéro d'octets dans la ligne et le numéro de ligne dans le cache sont alors les mêmes pour l'adresse physique et pour l'adresse virtuelle, ils correspondent aux bits utilisés pour chercher le numéro d'octets dans la page. Le contrôleur du cache peut alors commencer l'accès au cache et récupérer les étiquettes tandis que le processeur obtient une traduction en adresse physique de son étiquette virtuelle. Pour accélérer cette traduction et éviter d'aller la chercher en mémoire pour chaque instruction load/store, la plupart des processeurs intègrent un TLB (*Translation Lookaside Buffer*) qui n'est rien d'autre qu'un cache de traductions d'adresses ; il contient donc les adresses virtuelles et physiques des pages les plus récemment et fréquemment utilisées.



Hiérarchie de caches. La différence de performance entre processeur et mémoire continuant à s'accroître, la plupart des processeurs récents intègrent non pas un cache, mais une hiérarchie de caches dans la taille augmente et la vitesse décroît au fur et à mesure que l'on se rapproche de la mémoire principale.

Exemple. Hiérarchie de caches dans le processeur Compaq Alpha 21164.

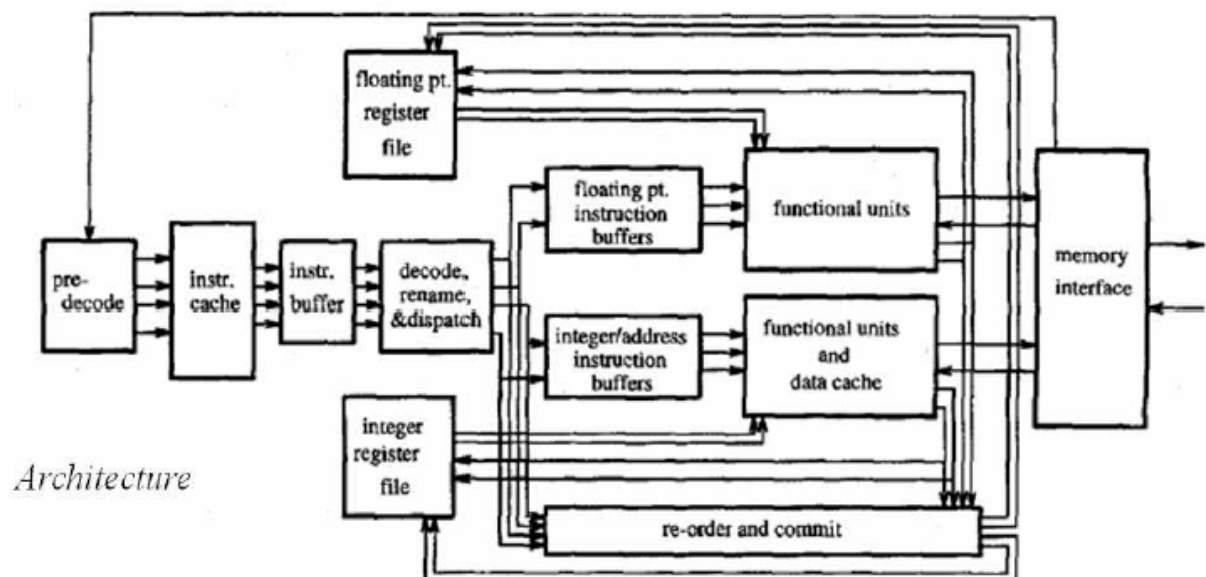


4. Processeurs superscalaires

Une façon majeure de gagner en performance, au-delà de l'exécution pipelinée des instructions, est d'exécuter en parallèle plusieurs instructions. La plupart des processeurs haute-performance ont aujourd'hui cette capacité, on parle de processeurs **superscalaires**. Pour réaliser un processeur superscalaire, il faut:

- Assurer un flux d'instructions suffisant.
- Déterminer quelles instructions peuvent s'exécuter en parallèle.
- Passer les données entre les instructions (le résultat d'une instruction i est l'opérande d'une instruction j).
- Disposer de plusieurs unités de calcul en parallèle.

La plupart des processeurs superscalaires ont une structure similaire à celle indiquée dans la figure ci-dessous :



Parallélisme entre instructions. Bien que notre façon de raisonner soit essentiellement séquentielle, une fois le programme traduit en assembleur, on constate qu'il y a un degré important de parallélisme entre instructions (*ILP : Instruction-Level Parallelism*).

Exemple de parallélisme entre instructions.

<pre> for (i=0; i<last; i++) { if (a[i] > a[i+1]) { temp = a[i]; a[i] = a[i+1]; a[i+1] = temp; change++; } } </pre>	<pre> L2: move r3,r7 #r3->a[i] lw r8,(r3) #load a[i] add r3,r3,4 #r3->a[i+1] lw r9,(r3) #load a[i+1] ble r8,r9,L3 #branch a[i]>a[i+1] move r3,r7 #r3->a[i] sw r9,(r3) #store a[i] add r3,r3,4 #r3->a[i+1] sw r8,(r3) #store a[i+1] add r5,r5,1 #change++ L3: add r6,r6,1 #i++ add r7,r7,4 #r4->a[i] blt r6,r4,L2 #branch i<last </pre>
---	---

Chargement des instructions. Une des principales difficultés des processeurs superscalaires actuels est de charger des instructions à un rythme suffisant pour alimenter le pipeline. Le principal écueil est la présence de nombreux branchements. Pour limiter leur impact sur le flux des instructions, on utilise plusieurs techniques : avant tout, les techniques de prédiction de conditions et d'adresses mentionnées ci-dessus, mais aussi le chargement anticipé des instructions depuis le cache des instructions dans un tampon de préchargement, et également l'insertion de bits de prédécodage dans le cache d'instructions afin de déterminer très rapidement la nature de l'instruction (branchement conditionnel, branchement inconditionnel, instruction n'influant pas sur le flot de contrôle).

Décodage des instructions. Une fois les instructions chargées, la première étape consiste à déterminer les dépendances entre les instructions, plus exactement entre les registres lus et écrits par les instructions. Cette analyse permettra de déterminer quelles instructions peuvent s'exécuter en parallèle. Elle permet également d'éliminer les fausses dépendances entre les instructions liées à la réutilisation des registres ; cette dernière action s'accompagne du renommage des registres.

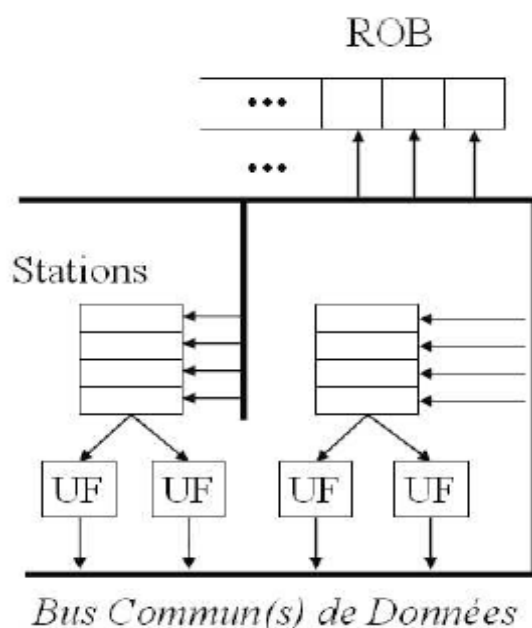
Exemple. Les instructions move r3 et lw r8 ci-dessous ne peuvent s'exécuter en parallèle en raison d'une dépendance sur le registre r3. Par ailleurs, le fait que l'instruction move r3 utilise le même registre de destination que l'instruction add r3 va ralentir la fin de l'exécution de cette seconde instruction en raison de la réutilisation du registre r3.

<pre> L2: move r3,r7 lw r8,(r3) add r3,r3,4 lw r9,(r3) ble r8,r9,L3 </pre>	
---	--

Renommage des registres et tampon de réordonnement. Le nombre de registres que peuvent manipuler les instructions assembleur est limité par le jeu d'instructions. Or la taille des processeurs permet aujourd'hui de disposer de beaucoup plus de registres. Comme ces registres additionnels ne peuvent pas être exhibés à l'utilisateur par le biais du jeu d'instructions, les architectures de processeurs superscalaires effectuent un renommage à la volée des registres. On parle alors de registres logiques, ceux du jeu d'instructions et du programme, et de registres physiques, ceux contenus dans le processeur.

En pratique, on ne dispose pas toujours de registres physiques supplémentaires mais d'un **tampon de réordonnement** (*ROB : ReOrder Buffer*, le *ROB* est une FIFO) qui permet à la fois de mémoriser l'ordre des instructions dans le programme (on verra plus tard pourquoi cette fonctionnalité est nécessaire) et de fournir l'équivalent de registres physiques additionnels. Outre le *ROB*, le processeur contient également un banc de registres correspondant aux registres logiques. A chaque instruction chargée est associée une entrée dans le *ROB*. Le registre logique de destination de l'instruction est remplacé par le numéro d'une entrée du *ROB*: on renomme le registre. En outre, la valeur produite par l'instruction est également stockée dans le *ROB*, et une fois qu'une instruction a terminé son exécution et sort du *ROB*, sa valeur est écrite dans son registre logique de destination, dans le banc de registres. Une table indique où se trouve la valeur courante d'un registre logique (dans le banc de registres ou dans le *ROB*), et lors du chargement d'une instruction, ses opérandes sont donc soit dans le *ROB*, soit dans le banc de registres.

Exécution. Une fois ces étapes préliminaires effectuées, on envoie l'instruction pour exécution. Si l'un de ses opérandes n'est pas encore disponible, ou si aucune unité fonctionnelle n'est libre, on envoie l'instruction dans une des **stations de réservation** associée à l'unité fonctionnelle. Ces stations de réservation forment une file d'attente devant l'unité fonctionnelle et espionnent en permanence les bus de sortie des unités fonctionnelles ; dès qu'une station de réservation repère que l'instruction dont elle attend le résultat vient de terminer son exécution, elle se saisit du résultat, et si l'unité fonctionnelle est libre, l'instruction en attente commence son exécution. Implicitement, les instructions vont s'exécuter dans l'ordre de disponibilité de leurs opérandes : à l'intérieur du processeur, il s'agit donc d'un modèle d'exécution de type *Dataflow* et non de type *Von Neumann*.



Fin de l'exécution. Après exécution de l'instruction, le résultat est donc propagé aux stations de réservation par le biais de bus communs de données et est également stocké dans le ROB. Le ROB étant une FIFO, les instructions en sortent dans l'ordre du programme ; vu de l'extérieur, un processeur superscalaire se comporte donc bien comme un processeur de type *Von Neumann*. De même qu'il est possible de charger plusieurs instructions à la fois, en un seul cycle, il est possible de sortir plusieurs instructions à la fois du ROB. Lorsqu'une instruction sort du ROB, son résultat est écrit dans le banc de registres.

Une des principales raisons pour lesquelles on force les instructions à terminer dans l'ordre est la nécessité d'implémenter des exceptions précises : lorsque l'on traite une exception, en fin de pipeline, on doit s'assurer que toutes les instructions précédentes ont bien terminé de s'exécuter. Sachant qu'une instruction *store* n'est effectivement envoyée à la mémoire que lorsqu'elle sort du ROB, on constate donc que l'état du processeur (registres et mémoires) est bien modifié dans l'ordre spécifié par les instructions du programme assembleur.

5. Conclusion

La complexité des processeurs haute-performance est telle, aujourd'hui, qu'il est difficile d'écrire un programme qui en exploite réellement la performance maximale. Il existe une différence très importante, et surtout croissante, entre la performance maximale d'un processeur et sa performance soutenue. En outre, il est de plus en plus difficile de dimensionner un processeur (d'en accroître la capacité et la performance) au fur et à mesure de l'évolution de la technologie. Il est donc possible que, dans les années à venir, on assiste à une modification relativement profonde du paradigme d'exécution des processeurs haute-performance.
