Architecture des ordinateurs

Licence Informatique - Université de Provence

Jean-Marc Talbot

jtalbot@cmi.univ-mrs.fr

Optimisation : pipeline

Pipeline (I)

L'exécution d'une instruction est décomposée en plusieurs étapes utilisant :

- des parties différentes du chemin de données
- des parties différentes de l'unité de contrôle/commande

De nombreuses unités sont donc **inactives** si on exécute qu'une seule instruction à la fois.

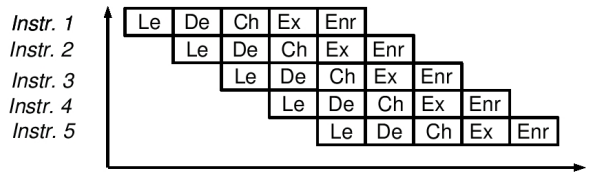
$\Longrightarrow$ paralléliser l'exécution des instructions : utilisation d'un pipeline

Pipeline (II)

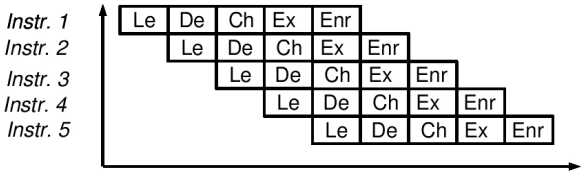
Exemple simplifié : Instructions de type R

Sous-opérations de l'exécution d'une instruction :

- LE : lecture de l'instruction en mémoire
- DE : décodage de l'instruction
- CH : chargement des registres de UAL avec le contenu des registres source
- EX : exécution du calcul
- ENR : enregistrement du résultat dans le registre destination



Pipeline (III)



Grande augmentation des performances

- Sans : exécution séquentielle de 2 instructions en 10 cycles
- Avec : exécution parallèle de 5 instructions en 9 cycles

Gain théorique car nombreux problèmes d'**aléas**

En pratique, autour de 12/15 étages dans un pipeline.

Aléas dans un pipeline

Le fonctionnement idéal du pipeline se base sur le fait que

- chacune des sous-opérations utilise des parties différentes du chemin de données
- les instructions exécutées les unes à la suite des autres sont indépendantes
- L'instruction devant être exécutée après celle en cours d'exécution est la suivante en mémoire (ou peut être facilement déterminée)

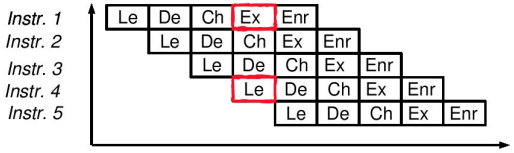
En pratique, aucune de ces conditions n'est vérifiée ; la violation d'une de ces conditions s'appelle un **aléa**

- **aléa structurel** : des parties du chemin de données doivent être utilisés simultanément par plusieurs étage du pipeline
- **aléa de données** : le calcul d'une valeur à un étage du pipeline nécessite une valeur non encore calculée
- **aléa de contrôle** : l'instruction suivante dépend d'une valeur calculée

Aléas structurels

La réalisation du chemin de données interdit certaines combinaisons d'opérations

Exemple : LE et EX accède tous les deux à la mémoire en cas de exécution d'une instruction de lecture écriture.



Solutions :

- attendre que l'unité soit disponible en retardant l'exécution : peu efficace
- dupliquer les différentes unités du chemin de données : ici, accès mémoire : découpage en 2 parties du cache L1
  - LE accède à la partie "instruction"
  - EX accède à la partie "données"

Aléas de données (I)

```
addi $1, $2, 12
multi $3, $1, 2
```

Problème : la lecture de la valeur de `$1` (Ch) de `multi $3, $1, 2` précède l'écriture de `$1` (Enr) dans `addi $1, $2, 12`.

Solution :

- arrêter le calcul de `$3` tant que `$1` n'est pas connu

- changer l'ordre d'exécution des instructions : **réordonnancement** (réalisé soit à la compilation, soit par le processeur à la volée)

```
addi $1, $2, 12          addi $1, $2, 12
multi $3, $1, 2    ⟹    li $5, 4
li $5, 4                 add $4, $5, $6
add $4, $5, $6           multi $3, $1, 2
```

Aléas de données (II)

Solution (suite) :

- On court-circuite le pipeline (et l'exécution normale des instructions) en plaçant le contenu du registre de sortie de UAL directement dans un des ses registres d'entrée.

  $\Longrightarrow$ modification du chemin de données

  $\Longrightarrow$ nécessité d'empiler les résultats de sortie de l'ALU si on veut traiter des aléas de données sur plusieurs étages.

Aléas de contrôle

```
sub $1, $1, $2
bne $1, $0, Suite
add $3, $3, $1
```

Problème :

L'instruction à charger après celui de `sub $1, $1, $2` dépend de la valeur de `$1` après l'exécution de celle-ci.

Solutions :

- attendre que le résultat de l'opération soit connue : peu efficace (ici, le chargement de l'instruction (LE) ne peut se faire qu'après l'écriture dans `$1`
- réaliser une prédiction de branchement et commencer le calcul avec celle-ci

Prédiction de branchement (I)

- Prédiction statique :
  - on suppose que le test réalisé est faux : pas de saut (Intel 486)
  - on suppose que le saut est effectué si c'est un saut arrière
  - le compilateur choisit le sens au moment de la production du code
- Prédiction dynamique :
  des informations concernant les branchements du programme sont stockés au cours de l'exécution et utilisées pour réaliser les prédictions de branchement futurs.

Prédiction de branchement (II)

La qualité d'une méthode de prédiction est donnée par

- le nombre de succès de la prédiction
- l'efficacité de cet algorithme

ATTENTION : en cas de prédiction erronée, le pipeline doit être vidé.

Optimisation : mémoire cache

Mémoire cache (I)

Le processeur a besoin en permanence des données à traiter et des instructions à exécuter

Mais la mémoire centrale ne peut fournir raisonnablement ces données à la vitesse dont le processeur en a besoin

Solution : utiliser une mémoire plus rapide entre la mémoire centrale et le processeur

Mémoire cache ou **cache**

Mémoire cache (II)

cache : mémoire d'accès rapide stockant une copie des données en petite quantité choisies parmi les données stockés dans une dispositif mémoire plus lent.

Conceptuellement, les registres servent de cache à la mémoire cache qui sert de cache à la mémoire centrale qui sert de cache à la mémoire de masse.

ATTENTION : Le mécanisme de cache entre la mémoire centrale et une mémoire de masse est du domaine du système et donc, logiciel.

Mémoire cache (III)

La mémoire cache doit rester **petite** :

- adressage limité pour rester efficace en temps d'accès et débit
- coût important

Problème : elle ne peut stocker un programme et toutes ses données.

Il faut décider :

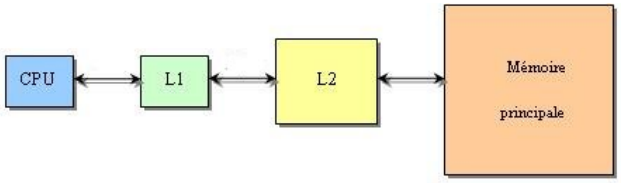
- quelle doit être la taille du cache, le nombre de niveau de caches, ...
- quoi mettre dans le cache et quand

Mémoire cache : niveaux de cache (I)

La mémoire cache sert à combler la différence de vitesse entre le processeur et la mémoire centrale.

La mémoire cache se divise généralement en deux parties

- une mémoire cache de niveau 1 (L1) au sein du processeur et allant à sa vitesse
- une mémoire cache de niveau 2 (L2) de taille supérieure au niveau 1 mais d'accès plus lent.



Mémoire cache : niveaux de cache (II)

2 relations possibles pour les niveaux L1 et L2

- Cache inclusif :
  - le contenu de L1 est également dans L2 (Le contenu de L1 est inclus dans L2) (L1 contient une copie de L2)
  - L2 est une copie partielle de la mémoire centrale et L1 est une copie partielle de L2
  - taille de la mémoire cache = taille de L2
- Cache exclusif :
  - le contenu de L1 n'est pas présent dans L2 (Les contenus de L1 et L2 sont exclus mutuellement)
  - L2 contient le trop-plein de L1 : quand L1 est plein, on transvase vers L2 et inversement si une données redevient intéressante.
  - taille de la mémoire cache = taille de L1 + taille de L2

Mémoire cache : niveaux de cache (II)

Comparaison

- Cache inclusif :
  - ⊕ Cache L2 plus performant
  - ⊖ Taille totale plus faible
  - ⊖ Taille de L2 ne doit pas être trop petite par rapport à celle de L1
- Cache exclusif :
  - ⊕ Cache plus grand au total
  - ⊕ L2 de taille quelconque
  - ⊖ la non-duplication des données entre L1 et L2 est très coûteuse : L2 est moins performant
- cache inclusif : Intel Pentium - Intel Xeon
- cache exclusif : AMD K8 - AMD Duron

Mémoire cache : réalisation (I)

La mémoire cache est de type SRAM

La mémoire centrale stocke de manière contiguë des données, les adresses se suivent.
Les mémoires caches doivent utiliser un principe différent, car les mots qu'elles vont stocker ont des adresses quelconques, qui ne se suivent pas forcément.

Les mémoires caches sont des **mémoires associatives**.

Mémoire cache : réalisation (II)

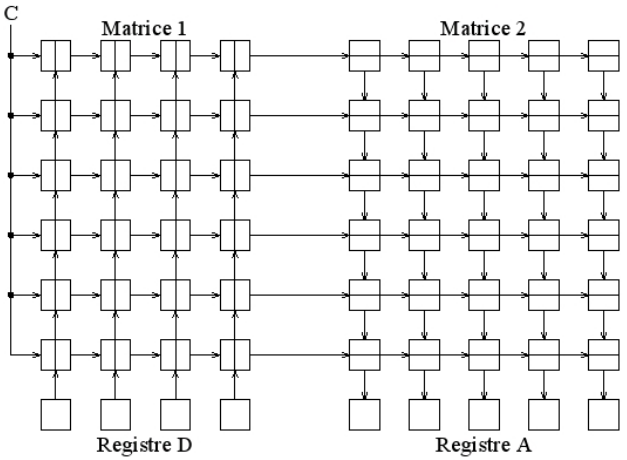
mémoire associative :

un **descripteur** ou **clef** est associé(e) à chacune des informations présentes dans la mémoire.

La recherche dans la mémoire se fait par ce descripteur :

- le descripteur à rechercher est comparé simultanément à tous ceux présents dans la mémoire.
- en cas d'égalité avec un descripteur de la mémoire, l'information associée à celui-ci est activée.

Mémoire cache : réalisation (III)



- la matrice 2 contient les informations
- la matrice 1 remplace le décodeur (d'adresses)

Fonctionnement de la mémoire cache (I)

L'accès au cache doit être transparent pour le processeur :

- Le processeur envoie une demande de lecture à une adresse de la mémoire centrale et reçoit la donnée (qui devrait se trouver) à cette adresse en mémoire
- Une demande d'écriture d'une donnée à une certaine adresse de la mémoire centrale, puis de lecture à cette même adresse retourne la même donnée.

Les accès à la mémoire sont corrects du point de vue du processeur

Fonctionnement de la mémoire cache (II)

Principe de fonctionnement :

1. Le processeur demande une donnée à une certaine adresse de la mémoire centrale

2. Le cache vérifie s'il possède la donnée de cette adresse :
   a) **succès :** la donnée est présente dans le cache et elle est transmise au processeur
   b) **défaut de cache :** la donnée n'est pas présente dans le cache ; celui-ci la demande au cache de niveau supérieur ou à la mémoire centrale, la stocke une fois obtenue et la transmet au processeur.

Fonctionnement de la mémoire cache (III)

Augmentation des performances :

- Augmentation de la taille du cache
  - augmente le taux de succès
  - mais augmente le temps d'accès au cache
- Augmentation du nombre de niveaux de cache
  - gain significatif pour le passage de 1 à 2 niveaux
  - gain faible pour le passage de 2 à 3 niveaux
  - gain insignifiant ou nul au dessus

Gestion du contenu de la mémoire cache (I)

Le cache (de niveau 1) est souvent divisé en deux parties

- une partie "donnée" qui stocke des données nécessaire à l'exécution du programme
- une partie "instruction" qui contient des instructions du programme en cours d'exécution

Que doit on mettre dans/retirer de la mémoire cache et quand ?

Gestion du contenu de la mémoire cache (II)

principe de localité :

- *localité temporelle :* une donnée manipulée à l'instant t aura de grandes chances d'être manipulée dans un futur proche
- *localité spatiale :* si une donnée d'adresse d est manipulée l'instant t alors des données d'adresses proches ont de fortes chances d'être manipulées conjointement.

Dans le cas d'instructions, si une instruction est exécutée alors ses suivantes en mémoire ont de fortes chances d'être exécutées dans un avenir proche.

Gestion du contenu de la mémoire cache (III)

principe de localité :

```
for (i = 0; i < n; i++)
    somme +=A[i];
```

- localité spatiale : $A[i], A[i+1], A[i+2], ...$
- localité temporelle : n, A, i

Gestion du contenu de la mémoire cache (IV)

pre-fetching : chargement en avance des données/instructions dont le processeur va avoir besoin.

Permet d'augmenter le taux de succès du cache

Les algorithmes de pre-fetching sont basés sur le principe de localité

- localité temporelle : garder dans le cache les dernières données manipulées par le programme.
- localité spatiale : charger en avance les données/instructions contiguës à une donnée/instruction référencée.

Gestion du contenu de la mémoire cache (V)

Remplacement d'informations dans le cache

La lecture ou l'écriture pour le processeur à une adresse non présente dans le cache nécessite son chargement dans ce dernier

Nécessite d'enlever des données présentes dans le cache pour y mettre ces nouvelles.

Quel choix des données à ôter ?

- Random
- LRU
- LFU

Gestion du contenu de la mémoire cache (VI)

Remplacement d'informations dans le cache

- Remplacement aléatoire (Random) :
  - simple à mettre en œuvre
  - peu efficace car on peut supprimer des données très accédées.
- Remplacement de la plus ancienne "non utilisée"
  - LRU : Last Recently Used
  - Nécessite des compteurs associés aux données
- Remplacement de la moins utilisée
  - LFU : Least Frequently Used
  - Nécessite également des compteurs

Cohérence des données entre cache et mémoire (I)

Lecture/écriture dans la mémoire centrale via le cache

- Le cache contient une partie des données de la mémoire centrale : risque d'incohérence entre les données
- opérations de lecture/écriture se réalise sur le cache : gérer les répercussions sur la mémoire centrale
- opérations de lecture : pas de modifications des données, donc **cohérence entre le cache et la mémoire**
- opérations d'écritures : modification du cache, donc **incohérence entre le cache et la mémoire**

Cohérence des données entre cache et mémoire (II)

Cohérence cache/mémoire lors d'une opération d'écriture

2 Méthodes :

- Ecriture simultanée (**write-through**)
  La donnée écrite dans le cache est simultanément écrite en mémoire
  - ⊕ cache et mémoire sont cohérentes à chaque instant
  - ⊖ écriture plus lente car accès à la mémoire centrale
- Ecriture différée (**write-back**)
  La donnée est modifiée uniquement dans le cache, la répercussion est faite en mémoire uniquement lorsque la donnée est sortie du cache
  - ⊕ limitation des accès à la mémoire en écriture
  - ⊖ problèmes si d'autres éléments accède à la mémoire centrale (périphérique en mode DMA - Direct Memory Acces)

Correspondance lignes cache/mémoire (I)

La mémoire cache contient des **lignes** de mots de la mémoire centrale.

Exemple :

- Lignes de cache de 32 octets
- Mémoire cache de 512 Ko : 16384 lignes
- Mémoire centrale de 128 Mo
  - doit être gérée par les 512 Ko de cache et ses 16384 lignes.
  - 1 ligne du cache / 8192 octets (256 lignes) de la mémoire centrale.

Quelle est la relation entre les lignes du cache et celle de la mémoire centrale ?

Correspondance lignes cache/mémoire (II)

Trois méthodes pour gérer la correspondance entre lignes du cache et lignes de la mémoire centrale :

- Correspondance directe (Direct mapping)
- Correspondance associative totale (Fully associative mapping)
- Correspondance associative par ensemble (N-way set associative mapping)

Correspondance directe (I)

Une ligne mémoire ne peut aller que dans une ligne du cache

L lignes de cache

- la ligne d'adresse j de la mémoire centrale est gérée par la ligne i du cache
  - $i = j \mod L$
  - A partir de l'adresse d'une ligne mémoire, on sait dans quelle ligne du cache elle doit se trouver.
- une ligne de cache étant partagée par plusieurs lignes de la mémoire centrale, on garde donc l'information sur la donnée effectivement présente dans le cache

Correspondance directe (II)

- ⊕ On sait immédiatement où aller chercher la ligne (accès rapide)
- ⊖ Nombreux défauts de cache conflictuels si on accède à des lignes de la mémoire centrale qui correspondent toutes à la même ligne du cache, tandis que d'autres lignes du cache ne sont pas utilisées.
- ⊖ performances médiocres (taux de succès : 60-80 %)

Correspondance associative totale

Chaque ligne de la mémoire peut se trouver dans n'importe quelle ligne du cache.

Le cache contient le numéro de ligne de le mémoire centrale de chacune des informations.

- ⊕ grande souplesse d'utilisation permettant d'augmenter le nombre de succès (90-95 % de succès)
- ⊖ temps de comparaison plus long car adresse complète (temps d'accès plus long)

Correspondance associative par ensemble

Combinaison des deux méthodes précédentes pour pallier leur défaut respectif.

- le cache est divisé en ensembles de lignes, chaque ensemble contenant N lignes.
- Chaque ligne de la mémoire centrale est affectée à un ensemble

$$\textit{ensemble} = \textit{numéro de ligne} \mod \textit{nombre d'ensembles}$$

- A l'intérieur d'un ensemble, correspondance associative totale.