Architecture avancée

Alain MÉRIGOT

Université Paris Saclay

Architecture avancée

Les processeurs actuels augmentent leurs performances grace à plusieurs mécanismes :

- le parallélisme d'instructions qui permet d'améliorer l'exécution d'une instruction et/ou de permettre l'exécution d'instructions simultanée
- le parallélisme de données, et notamment les extension dites SIMD qui permettent en une seule instruction de transformer plusieurs données de manière similaire.
- la possibilité de basculer rapidement d'un *thread* à un autre (processeurs *multithreads*).
- l'utilisation de plusieurs processeurs simultanés, avec une mémoire partagée (processeurs *multicores*).

Le parallélisme d'instructions

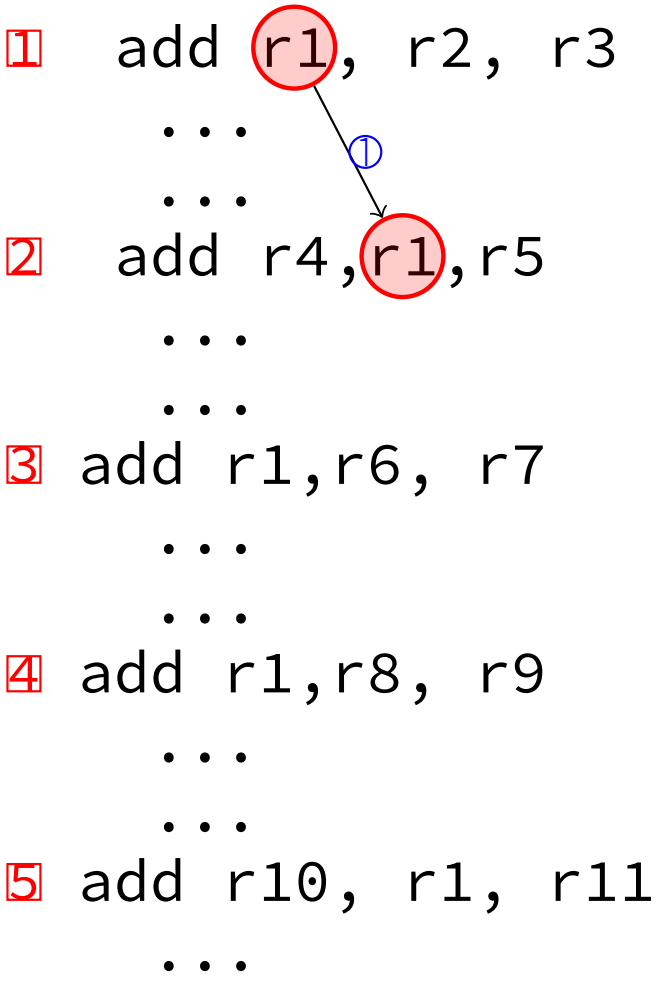
La parallélisation d'instructions repose sur une analyse des *dépendances* entre instructions.
Une dépendance reflète la séquentialité, le fait qu'une instruction doive s'exécuter *avant* une autre pour que le programme soit correct.
Ces mécanismes peuvent ensuite être utilisées :

- par un compilateur qui va modifier l'ordre des instructions pour améliorer l'exécution
- par un processeur qui va modifier *en cours d'exécution* l'ordonnancement des instructions (*exécution dans le désordre*)
- par un processeur qui va exécuter *plusieurs instructions* simultanément. *Processeurs superscalaires*
- par un processeur capable d'éxécuter des instructions en parallèle, la parallélisation étant faite préalablement par un compilateur. Processeurs VLIW

Différents types de dépendances de données

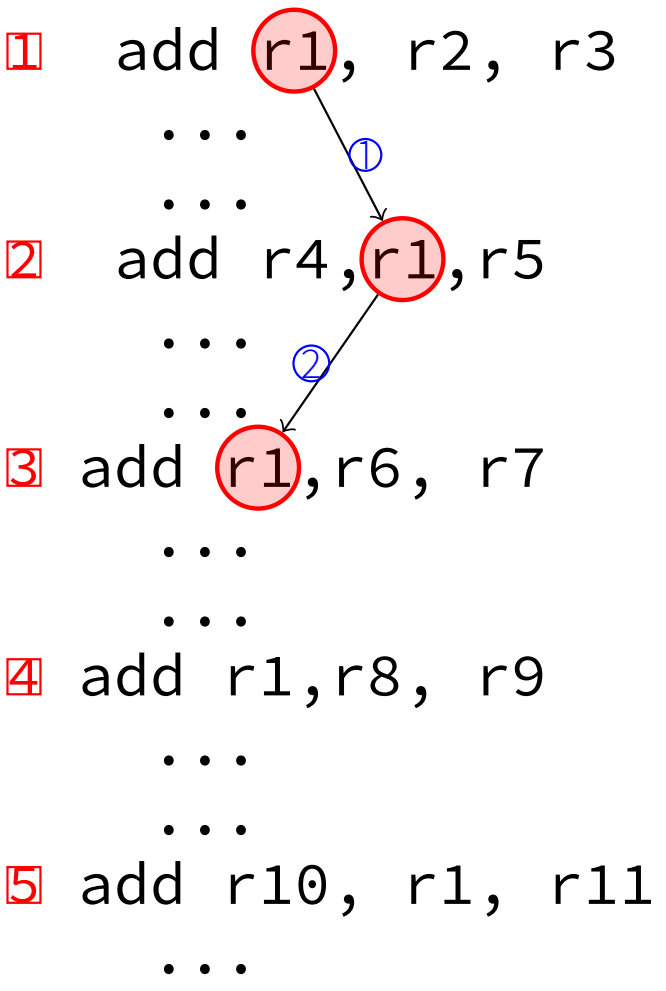
```
1  add r1, r2, r3
     ...
     ...
2  add r4,r1,r5
     ...
     ...
3 add r1,r6, r7
     ...
     ...
4 add r1,r8, r9
     ...
     ...
5 add r10, r1, r11
     ...
```

Différents types de dépendances de données



① L'information produite par l'instruction 1 dans `r1` est utilisée par l'instruction 2. C'est une vraie dépendance qui doit être respectée **dépendance de données**

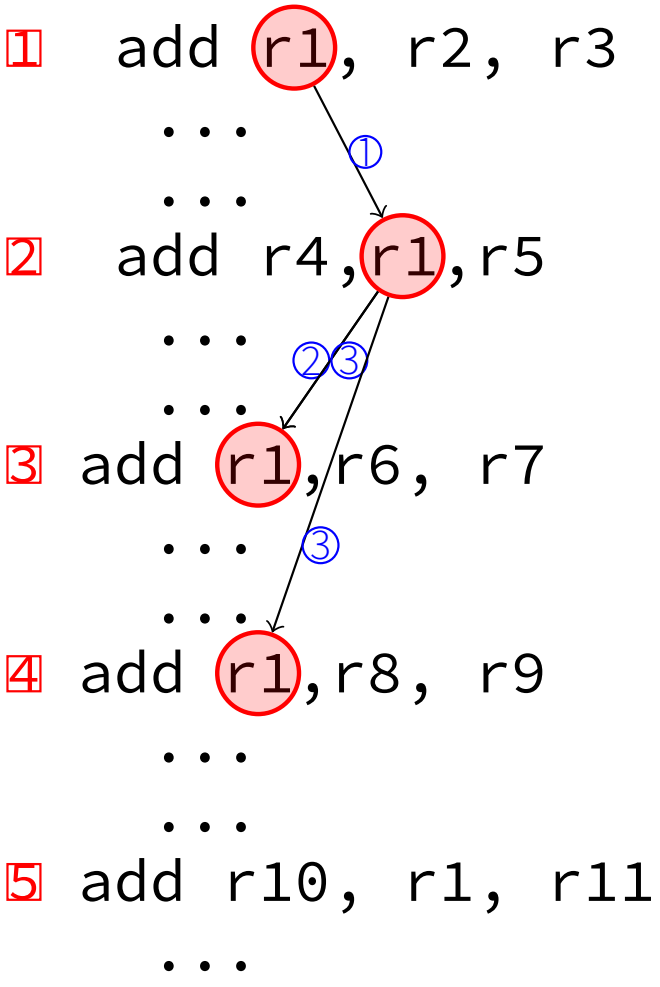
Différents types de dépendances de données



① L'information produite par l'instruction 1 dans `r1` est utilisée par l'instruction 2. C'est une vraie dépendance qui doit être respectée **dépendance de données**

② Le registre `r1` est écrit par l'instruction 3 et 2 doit lire son contenu *avant*. Par contre, il suffit d'utiliser un autre registre que `r1` pour que le problème disparaisse. **antidépendance** ou **dépendance de nom**

Différents types de dépendances de données

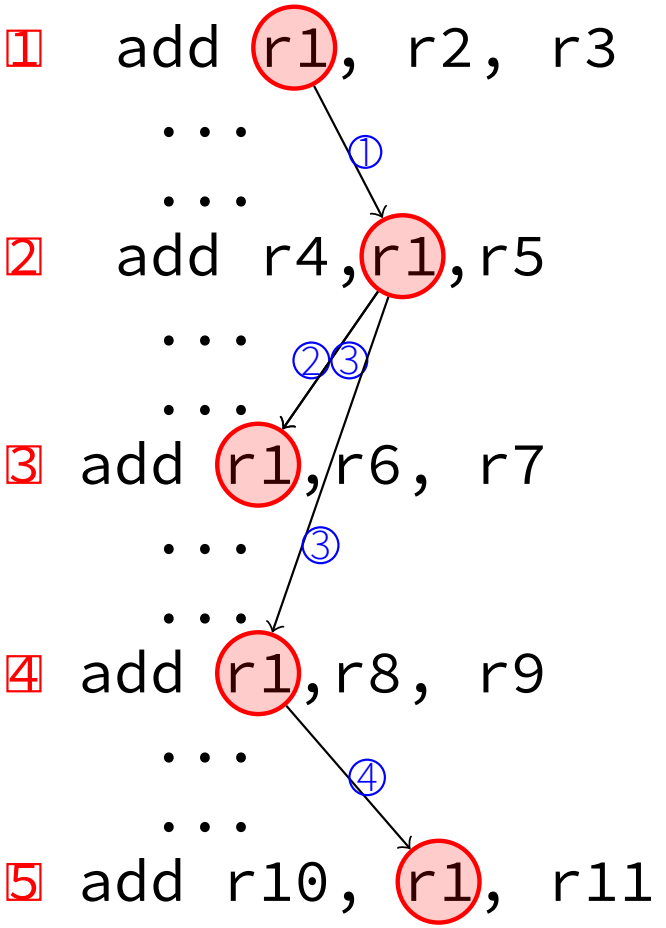


① L'information produite par l'instruction 1 dans `r1` est utilisée par l'instruction 2. C'est une vraie dépendance qui doit être respectée **dépendance de données**

② Le registre `r1` est écrit par l'instruction 3 et 2 doit lire son contenu *avant*. Par contre, il suffit d'utiliser un autre registre que `r1` pour que le problème disparaisse. **antidépendance** ou **dépendance de nom**

③ Les instructions 3 et 4 écrivent successivement le registre `r1`. L'ordre des écritures doit être respecté. **dépendance de sortie**

Différents types de dépendances de données



① L'information produite par l'instruction 1 dans `r1` est utilisée par l'instruction 2. C'est une vraie dépendance qui doit être respectée **dépendance de données**

② Le registre `r1` est écrit par l'instruction 3 et 2 doit lire son contenu *avant*. Par contre, il suffit d'utiliser un autre registre que `r1` pour que le problème disparaisse. **antidépendance** ou **dépendance de nom**

③ Les instructions 3 et 4 écrivent successivement le registre `r1`. L'ordre des écritures doit être respecté. **dépendance de sortie**

④ A nouveau une **dépendance de données** entre 4 et 5.

Réordonnancement d'instructions

On suppose qu'un `ld` suivi de l'utilisation du registre chargé depuis la mémoire provoque une suspension du pipeline d'un cycle (et même beaucoup plus, en cas de défaut de cache et/ou de pipeline profond).

Programme à exécuter

```
ld r1, 0(r2)
add r2, r1, r3
ld r4, 0(r5)
add r6, r4, r7
```

Sans optimisation

```
ld r1, 0(r2)
....susp...
add r2, r1, r3
ld r4, 0(r5)
....susp...
add r6, r4, r7
```

Présence de suspensions du pipeline 6 cy.

Après réordonnancement

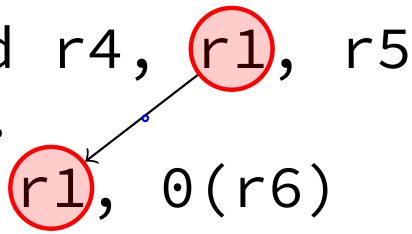
```
ld r1, 0(r2)
ld r4, 0(r5)
add r2, r1, r3
add r6, r4, r7
```

La permutation des instructions 3 et 4 ramène le temps à 4 cycles.

Dépendances de nom

Soit le programme ci-dessous

```
add r1, r2, r3
...
add r4, r1, r5
...
ld r1, 0(r6)
...
add r7, r1, r8
```



Dépendance de nom du fait de l'utilisation du même registre `r1`

Il suffit de *renommer* le (deuxième) registre `r1` en `r9`

```
add r1, r2, r3
...
add r4, r1, r5
...
ld r9, 0(r6)
...
add r7, r9, r8
```

La dépendance de nom a disparu sans que cela change le programme.

Le renommage est fait automatiquement par la plupart des processeurs. Ils disposent en interne de beaucoup plus de registres que dans le jeu d'instruction et à chaque nouvelle écriture dans un registre il est renommé en fonction des registres disponibles.

Prise en compte des aléas de contrôle

Pour limiter les aléas de contrôle, il y a deux mécanismes :

Prédiction de branchement On analyse le comportement des instructions de branchement et quand on retombe sur une instruction de rupture de séquence, on va supposer qu'elle aura le même comportement (même adresse de branchement).

Exécution spéculative On peut exécuter des instructions *postérieures* à un branchement, sans connaitre le résultat du branchement. Les calculs effectués de manière *spéculative* ne seront recopiés dans les registres que quand on connaitra effectivement le comportement de l'instruction (branchement pris ou exécution en séquence).

Exécution d'instructions en parallèle

Processeurs superscalaires Le processeur analyse *dynamiquement* les dépendances et exécute en parallèle 2–6 instructions (1–2 inst entières, 2 flottantes, un accès mémoire, un branchement). La quasi totalité des processeurs généralistes actuels sont superscalaires. En pratique 2–3 instr en //.

Processeurs VLIW (*very long instruction word*) On arrive à augmenter le nombre d'instructions parallèles en faisant une analyse **par le compilateur** qui génère un code avec des instructions déjà parallélisées. 8–16 inst en //. Actuellement surtout pour processeurs de traitement de signal et serveurs.

Processeurs *multithread*

En cas d'attente longue pour une instruction (défaut de cache), on bascule sur un autre fil d'exécution (*thread*).

Permet de masquer les attentes de données en mémoire, si le programme comprend plusieurs *threads*.

Nécessite plusieurs jeux de registres sur le processeur.

Dans certains processeurs superscalaires (Pentium IV et suivants), on peut même exécuter **en parallèle** des instructions appartenant à des *threads* différents.

Parallélisme de données

Dans de nombreux cas, un processeur est amené à effectuer un traitement identique sur un ensemble de données (image, son, etc).
Pour améliorer le temps d'exécution dans cette situation, de nombreux processeurs possèdent des *extensions SIMD* (*single instruction, multiple data*).

- On considère qu'un registre de 64 bits contient en fait 8 données de 8 bits.
- une instruction SIMD va opérer en parallèle sur ces 8 données (reconfiguration de l'ALU)

Il existe différentes configurations possibles des registres 64 bits (8x8, 4x16, 2x32) et les versions actuelles supportent des données flottantes et même un parallélisme très important (AVX512 16x32bits).
Instructions diverses de compactage/décompactage, instructions *masquées*, mouvements de données (permutations, etc).

Multicoeurs

‘

Actuellement plusieurs processeurs par circuits (multicoeurs)

- Progrès des technologies d’intégration
- Limitations des progrès en architecture des processeurs (parallélisme d’instruction)
- Problèmes de consommation

Actuellement

- ordinateurs “grand public” 4 à 8 coeurs
- serveurs 8 à 16
- systèmes embarqués jusqu’à 256 !

Multicoeurs (cont.)

Un ordinateur multicoeur permet d'exécuter

- plusieurs process sur les différents coeurs
- plusieurs *threads* dans un process (parallèlisation d'un même programme sur plusieurs coeurs)

Le parallélisme permet d'améliorer la rapidité d'exécution d'un programme, mais il nécessite une réécriture complète des applications.

Les différents coeurs travaillent sur les mêmes données en mémoire centrale. (modèle à *mémoire partagée*)

Par contre, chacun a sa propre mémoire cache. Nécessité de mécanismes assurant la *cohérence des caches*