Un peu au delà du modèle de Von Neumann...

Dans ce cours, nous ferrons plusieurs zooms sur des ajouts au modèle de Von Neumann permettant d'améliorer les performances des ordinateurs : la *hiérarchie mémoire*, le principe des *mémoires caches*, et le *parallélisme d'instructions*.

Plan

Mémoires secondaires

La taille de la mémoire des ordinateurs continue de s'accroître, mais elle sera toujours insuffisante : nous développerons toujours de nouvelles applications capables de la saturer...

De plus, il s'agit d'une mémoire volatile, perdue à l'extinction de la machine...

Cela motive l'utilisation de *mémoires secondaires*, pour compléter la mémoire centrale :

- les supports magnétiques : les disques durs.
- les disques à lecture/écriture optiques : CD-ROM (*Compact Disc ROM*), CD-RW (*CD ReWritable*), DVD (*Digital Versatil Disc*)...
- la mémoire flash : « clefs usb », les disques SSD (*Solid State Drive*)...

Les mémoires peuvent être caractérisées par :

- *le temps d'accès* : temps nécessaire à l'écriture ou à la lecture d'une donnée.
- *le débit* : quantité de données qu'il est possible de transférer sur, ou de charger depuis le support par unité de temps.

Pour comprendre les différences entre tous ces supports, on considère le tableau suivant (seuls les ordres de grandeurs comptent ici) :

support	**temps d'accès**	**débit**	**capacité**
registres	1 ns		≈kio
mémoire RAM	5-60 ns	1-20 Gio/s	≈Gio
disques durs	3-20 ms	10-320 Mio/s	≈Tio
CD	120 ms	1-8 Mio/s	650 Mio
DVD	140 ms	2-22 Mio/s	4.6-17 Gio

Rappel : nano $= 10^{-9}$, micro $= 10^{-6}$, milli $= 10^{-3}$.

Le temps d'accès à la RAM est environ 10 fois plus long que pour les registres !

Pour compenser cette lenteur, une mémoire rapide est placée entre les registres de l'UCT et la mémoire centrale : c'est la *mémoire cache*, qui sert au stockage temporaire de données : les mots les plus souvent utilisés y sont conservés.

support	**temps d'accès**	**débit**	**capacité**
registres	1 ns		≈kio
cache	2-3ns		≈Mio
mémoire RAM	5-60 ns	1-20 Gio/s	≈Gio
disques durs	3-20 ms	10-320 Mio/s	≈Tio
CD	120 ms	1-8 Mio/s	650 Mio
DVD	140 ms	2-22 Mio/s	4.6-17 Gio

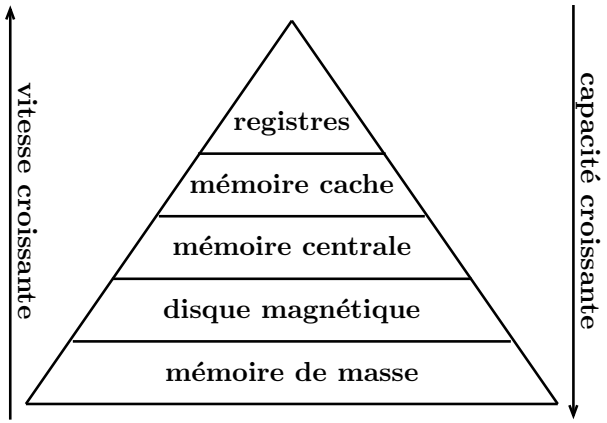
Rappel : nano $= 10^{-9}$, micro $= 10^{-6}$, milli $= 10^{-3}$.

Notion de hiérarchie mémoire

La conception de la mémoire pose les questions suivantes : « combien ? », « à quelle vitesse ? », « à quel prix ? » Il y a un compromis à trouver :

- plus la mémoire est rapide, plus son coût par bit est élevé ;
- plus la capacité est grande, plus les temps d'accès sont importants.

Une solution est de mettre en place une *hiérarchie mémoire* : Les mémoires rapides, de faibles capacité et très coûteuses sont secondées par des mémoires plus lentes, mais de plus grande capacité et à faible coût. On simule ainsi une mémoire de grande capacité, performante en moyenne, pour un coût raisonnable.



Un exemple

Supposons que le processeur a accès à deux niveaux de mémoire :

- Niveau 1 : temps d'accès de 0.01 μs ;
- Niveau 2 : temps d'accès de 0.1 μs.

Si un mot est accédé alors qu'il est stocké au niveau 1, le processeur y accède directement. S'il est au niveau 2, il doit d'abord être chargé au niveau 1, puis vers le processeur (on suppose pour simplifier que le temps pour déterminer si une donnée se trouve ou non au niveau 1 est nul).

Si par exemple 95% des accès se font au niveau 1, le temps d'accès moyen sera :

$$\begin{aligned} 0.95 \times 0.01\mu\text{s} + 0.05 \times (0.01\mu\text{s} + 0.1\mu\text{s}) &= 0.0095\mu\text{s} + 0.0055\mu\text{s} \\ &= 0.015\mu\text{s}. \end{aligned}$$

Le temps d'accès moyen est donc plus proche de 0.01μs que de 0.1μs.

Plan

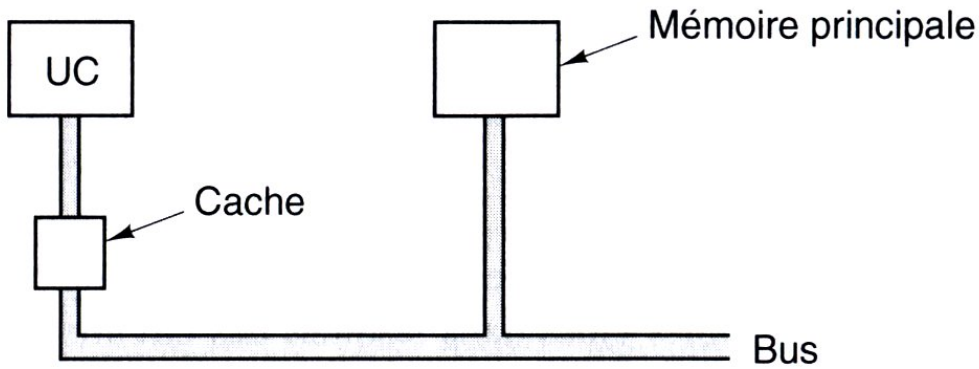
Mémoire cache

Nous allons décrire deux organisations possibles d'une mémoire cache

- cache à correspondance directe,
- cache associatif à n voies,

en nous appuyant essentiellement sur un exemple.

Pour fixer les idées, on suppose qu'un seul niveau de cache vient se placer entre la mémoire centrale et l'UCT :



Quand l'UCT réalise un accès à une donnée en mémoire :

- soit la donnée est présente dans le cache, et l'UCT y accède rapidement.
- soit la donnée est absente du cache, et elle doit d'abord y être chargée.

Cache à correspondance directe

Un ordinateur dispose d'une mémoire centrale de 2^{32} o, adressable par octets.

La mémoire principale est divisée en blocs de taille fixe, appelés *lignes de cache*.

- Chaque ligne comporte 2^5 o $= 32$ o.
- La mémoire se décompose en 2^{27} lignes de caches.

Chaque ligne peut être identifiée par un entier sur 27 bits.

On dispose d'une *mémoire cache* de 64 kio composée de $2048 = 2^{11}$ *entrées* de 32 o. Chaque *entrée* peut stocker temporairement une ligne de cache.

Comment utiliser ces 2^{11} entrées comme espace de stockage temporaire pour les 2^{27} lignes de cache de la mémoire centrale ?

D'après l'adresse d'une donnée en mémoire centrale, on doit pouvoir :

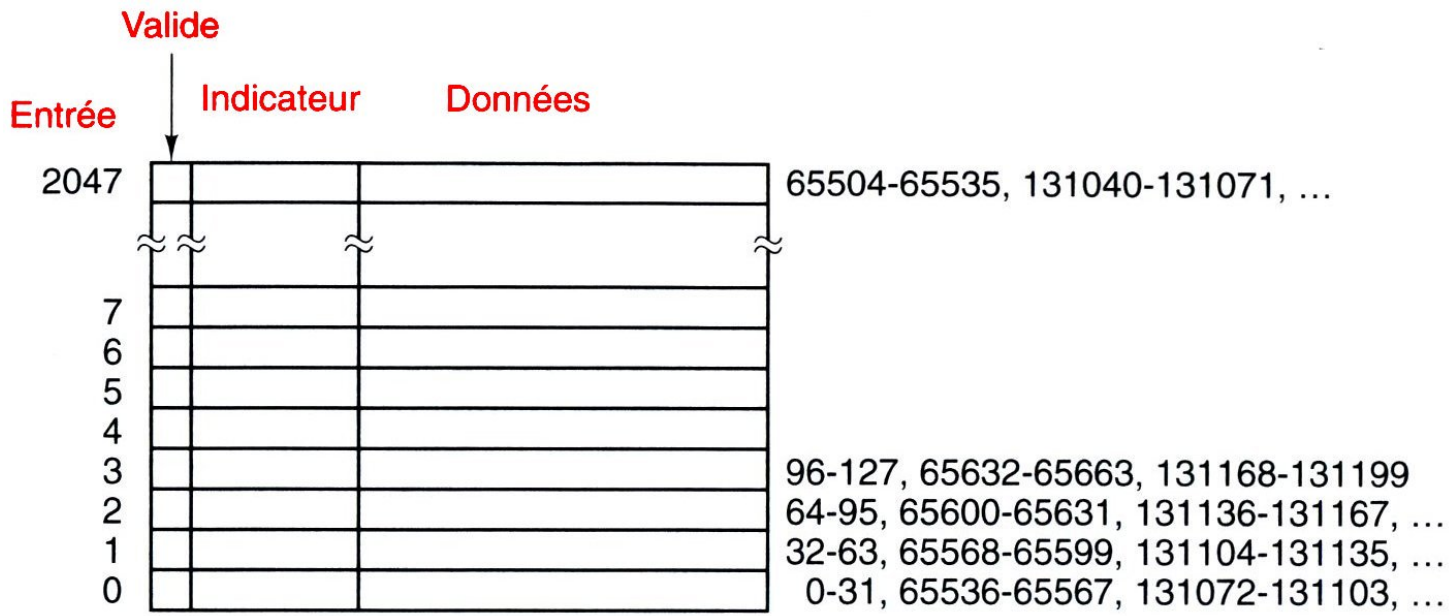
- déterminer à quelle ligne de cache elle appartient ;
- dire dans quelle entrée elle doit être rangée dans la mémoire cache.

On a 2^{11} entrées et 2^{27} lignes de cache. On décompose les adresses ainsi :

16 bits (31-16)	11 bits (15-5)	5 bits (4-0)
INDICATEUR	ENTREE	OCTET

- Les 27 bits formés de INDICATEUR et ENTREE identifient chaque ligne.
- Les 11 bits du champ ENTREE identifient une entrée du cache.
- Les 5 bits d'OCTET identifient l'un des 32 o d'une ligne ou d'une entrée.
- Toutes les lignes
  - ayant même ENTREE seront rangées dans la même entrée du cache.
  - ayant même INDICATEUR sont consécutives en mémoire centrale.

La mémoire cache est organisée comme ceci :



- Une ligne est stockée dans l'entrée correspondant à son champ ENTREE.
- Le champ Indicateur correspond au champ INDICATEUR des adresses, et permet d'identifier la ligne de cache d'où proviennent les données.
- Le bit Valide indique si l'entrée contient des données valides (initialisé à 0).
- Le champ Données contient la copie d'une ligne de cache.

Quand le processeur doit *lire* un octet à une adresse donnée en mémoire :

- le champ ENTREE de l'adresse indique une entrée du cache ;
- si Valide=1, et INDICATEUR=Indicateur, il y a *cache hit* ; l'octet peut être lu dans l'entrée du cache d'indice ENTREE, et sa position dans l'entrée est indiquée par OCTET. *Cela épargne un accès à la mémoire centrale.*
- si Valide=0 ou INDICATEUR≠Indicateur, il y a *cache miss* ; la ligne accédée est copiée de la mémoire vers le cache.

Pour les écritures : *comment assurer la cohérence entre les données du cache et celles de la mémoire ?* Il existe deux statégies d'écriture :

- *immédiate* : mise à jour simultanée du mot dans le cache et la mémoire principale ; on perd l'intérêt du cache pour les écritures...
- *différée* : si la ligne écrite est présente dans le cache, seul le cache est mis à jour. La mémoire n'est mise à jour que lorsque la ligne est évincée : le champ Valide sert à indiquer si la ligne a été écrite depuis son chargement.

Le système de cache à correspondance directe que nous venons de décrire place des lignes de caches consécutives dans des entrées consécutives du cache : 64 kio de données contiguës peuvent être stockées dans le cache.

Si on effectue des accès à tous les éléments d'un tableau de 65536 octets, on aura juste un *cache miss* par ligne de cache occupée par le tableau.

Par contre, deux lignes dont les adresses différent d'un multiple de 64 kio ne peuvent être stockées en même temps dans le cache, car elles ont le même champs ENTREE.

Si un programme effectue un accès à une adresse X, puis aux adresses $X + 65536$, $X + 2 \times 65536, \ldots$ alors il provoquera à chaque fois un *cache miss*, d'où de moins bonnes performances (cela revient à ne pas avoir de cache).

Les concepteurs des caches sont partis du principe que, souvent, le programmeur respectera le *principe de localité*, qui se décompose en deux parties :

- *localité temporelle :* une adresse mémoire ayant fait l'objet d'un accès récent sera de nouveau utilisée prochainement.

- *localité spatiale :* les adresses mémoires « proches » d'une adresse qui vient d'être utilisée sont susceptibles d'être utilisées aussi.

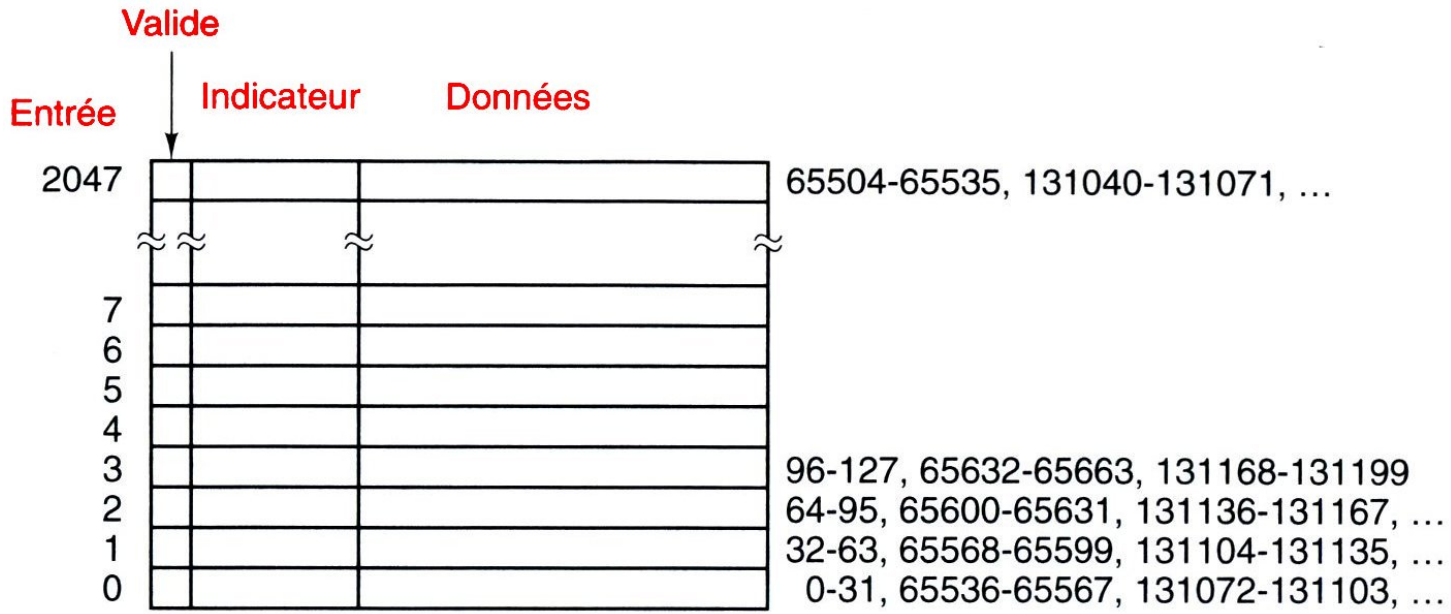
La tâche du programmeur est donc :

- de choisir des algorithmes favorisant le principe de localité,
- de programmer ces algorithmes selon le principe de localité,

afin d'améliorer les performances de ses programmes.

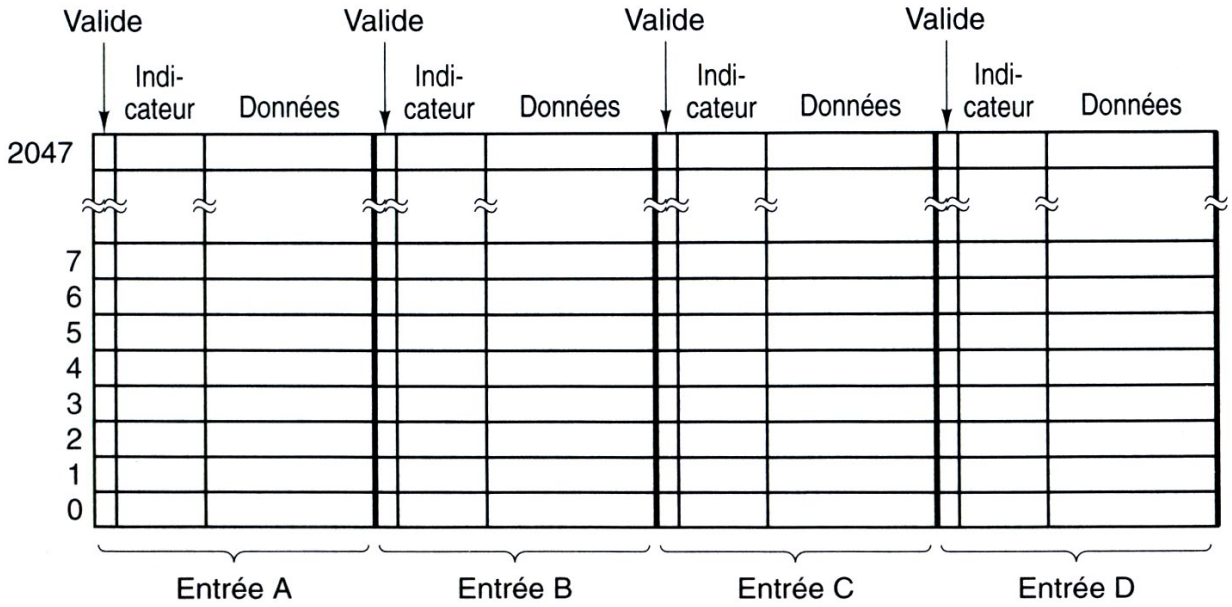
Cache associatif à n entrées

Dans un cache direct figuré ci-dessous, si un programme effectuent des accès intensifs aux adresses 0 et 65536, il y aura constamment un conflit sur l'entrée 0.



Une solution consiste à autoriser $n \geq 2$ lignes pour chaque entrées du cache.

On obtient un cache associatif à n voies (ici $n = 4$) :



Cela semble régler une partie du problème... Mais lorsqu'une nouvelle entrée doit être insérée dans le cache, quelle ligne doit en être évincée ?

Un algorithme assez efficace est l'algorithme *LRU* (*Least Recently Used*) : on évince la ligne qui n'a pas été utilisée depuis le plus longtemps.

D'autres algos existent, mais pas de solution miracle : on arrive toujours à trouver un programme provoquant « beaucoup » de *cache miss*. Mais on obtient nécessairement une amélioration des performances moyennes du cache.

Plan

Technique du pipeline

Supposons que l'on parvienne à découper l'exécution de toutes les instructions du jeu d'instructions en *4* phases (un seul cycle d'instruction commun), et que chaque phase occupe un cycle d'horloge de UCT :

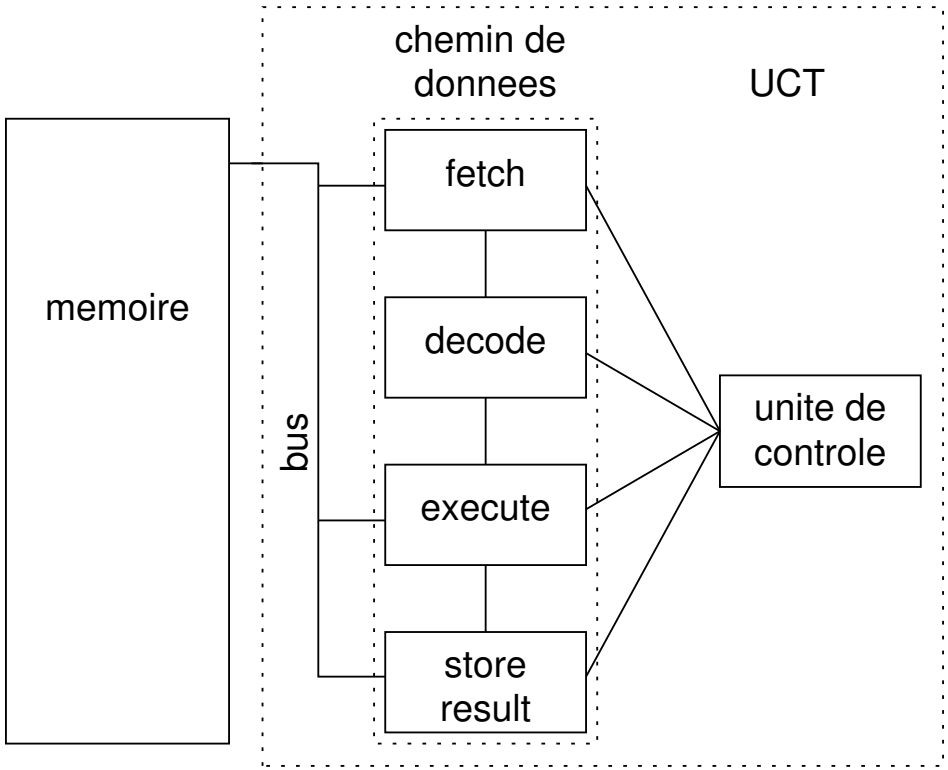
1. *FE*, FETCH : chargement de l'instruction depuis la mémoire centrale.
   ↪ phase commune à toutes les instructions.
   ↪ chargement de l'instruction à l'adresse donnée par le **PC** dans **IR**.
   ↪ incrémentation de **PC** en vue de l'exécution de l'instruction suivante.

2. *DE*, DECODE : décodage de l'instruction par l'unité de contrôle.
   ↪ phase commune à toutes les instructions.
   ↪ décision des phases suivantes en fonction de l'opcode.

3. *EX*, EXECUTE : exécution l'instruction à proprement parler.

4. *SR*, STORE RESULT : écriture des résultats en mémoire centrale.

Dans notre cas, voici une organisation possible pour un processeur mettant en place l'exécution du cycle d'instruction de la façon indiquée :

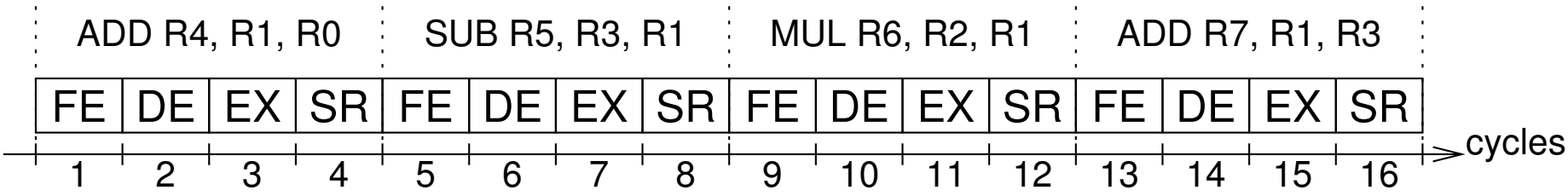


Supposons d'abord que lors du cycle d'instruction l'UC active successivement : FE (fetch), DE (decode), EX (execute), SR (store result).

On considère le programme suivant, donné en langage d'assemblage :

```
ADD R4, R1, R0      ; R4 <- R1 + R0
SUB R5, R3, R1      ; R5 <- R3 - R1
MUL R6, R2, R1      ; R6 <- R2 * R1
ADD R7, R1, R3      ; R7 <- R1 + R3
```

L'exécution de ce morceau de programme dans notre chemin de données sera :

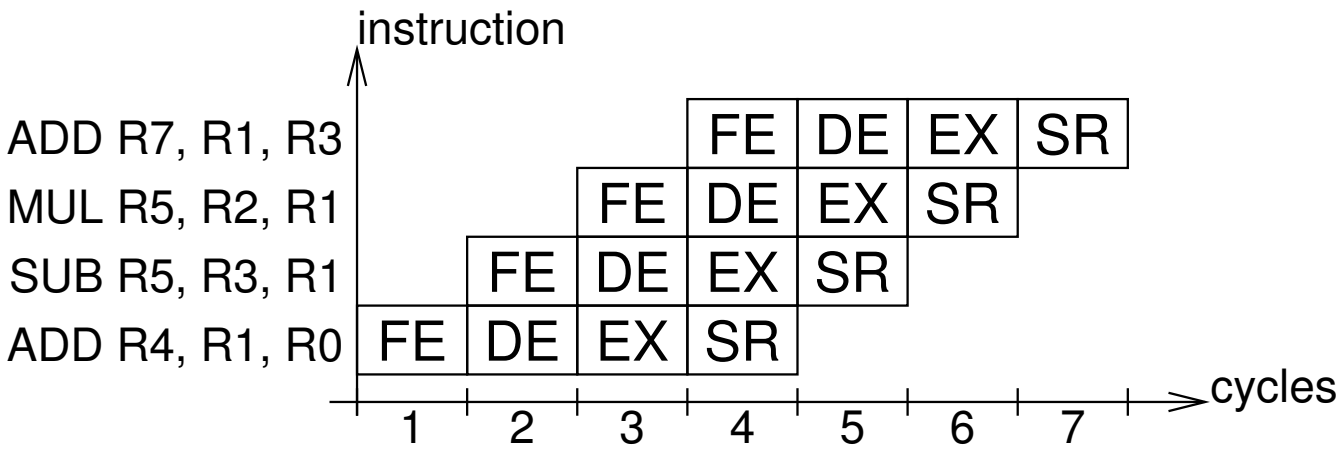


Les instructions sont exécutées les unes à la suite des autres, puisque de toute façon l'UC n'active jamais deux unités du chemin de données pendant le même cycle... Le programme s'exécute au total en 16 cycles.

Maintenant, *supposons que l'UC active les 4 unités du chemin de données à chaque cycle, créant ainsi un pipeline à 4 étages* :

- Au cycle 1, la phase FE de l'ins. 1 est traitée.
- Au cycle 2, la phase DE de l'ins. 1, et la phase FE de l'ins. 2 sont traitées.
- ...

On peut représenter l'exécution du programme sur le chronogramme suivant :



Désormais, le programme s'exécute en 7 cycles !

Plus généralement, on dit que le chemin de données forme un *pipeline à $k \geq 2$ étages* lorsque :

- les instructions possédent un cycle d'instruction commun de k phases,
- le chemin de données se décompose en k unités fonctionnelles, chacune traitant indépendamment l'une des phases du cycle d'instruction,
- les unités du chemin de données sont activées à chaque cycle par l'UC.

Idéalement, une instruction termine son exécution à chaque cycle d'horloge : on peut vérifier que $n \geq 1$ instructions s'exécutent dans ce cas en $n + k - 1$ cycles.

Tous les processeurs « généralistes » produits depuis le milieux des années 1980 sont pipelinés :

- Intel Pentium 2 (1997) : 14 étages.
- Intel Pentium 3 (1999) : 10 étages.
- Intel Pentium 4 (2000) : 20 à 31 étages (différentes versions).

Le problème des dépendances entre instructions

Cependant, il faut nuancer : il n'est pas toujours possible de terminer l'exécution d'une instruction à chaque cycle. Il y a plusieurs raisons à cela :

- Il n'est pas réaliste de supposer que les instructions d'accès à la mémoire vont toutes être réalisées en k cycles. On a vu que la latence de l'accès à la mémoire peut être variable (cache-hit / cache-miss).

- Il existe entre les instructions des *dépendances de données* : une instruction j dépend d'une instruction i si l'instruction i produit un résultat qui est un opérande source de l'instruction j. Par exemple :

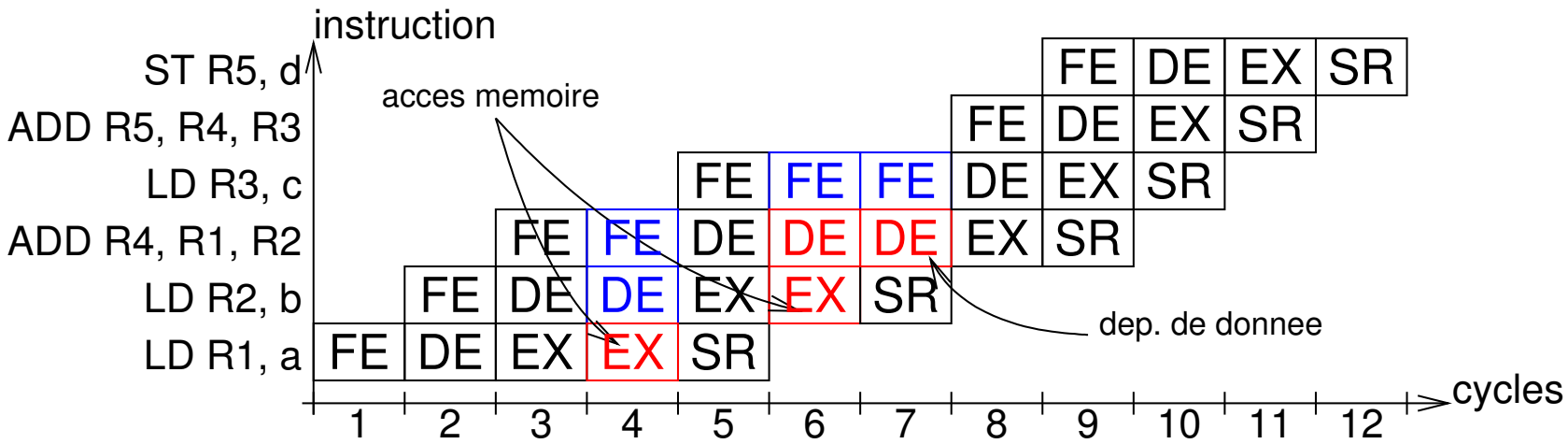
```
ADD R2, R0, R1
ADD R3, R2, R1
```

  La seconde instruction ne pourra pas commencer sa phase de EX dans le pipeline tant que la première n'aura pas passé la phase SR.

Exemple : On considère le programme suivant :

```
LD R1, a              ; R1 <- mem[a]
LD R2, b              ; R2 <- mem[b]
ADD R4, R1, R2        ; R4 <- R1+R2
LD R3, c              ; R3 <- mem[c]
ADD R5, R4, R3        ; R5 <- R4+R3
ST R5, d              ; mem[d] <- R5
```

On suppose que la phase EX d'une instruction commence au cycle qui suit la phase SR de l'instruction dont elle dépend :



Dans le cas le plus favorable, on peut exécuter 6 instructions en 9 cycles dans ce pipeline : dans cet exemple, 12 sont nécessaires.

Il existe d'autres types de dépendances entre instructions :

- *Dépendance de nom :* une dépendance de nom existe lorsque deux instructions utilisent le même nom (registre ou emplacement mémoire), mais qu'il n'y a pas de circultation de données entre les deux.
  $\hookrightarrow$ Les dépendances de nom peuvent être supprimées à l'aide d'une technique appelée *renommage* :

```
LD R1, a          LD R1, a
ADD R1, R1, 1     LD R2, b
ST R1, aa         ADD R1, R1, 1
LD R1, b          ADD R2, R2, 1
ADD R1, R1, 1     ST R1, aa
ST R1, bb         ST R2, bb
```

  Le renommage peut être réalisé :
  - par le programmeur,
  - à la compilation ou a l'assemblage,
  - par le processeur lui-même (renommage de registre).

- *Dépendance de contrôle :* une telle dépendance existe lorsque l'exécution d'une instruction est conditionnée par une instruction de branchement.

```
add:        ADD R2,R0,0
loop:       NOT R3,R1
            ADD R3,R3,1
            ADD R3,R2,R3
            BRp endloop   ; branchement de sortie de boucle
            LDR R3,R2,0
            ADD R3,R3,1
            STR R3,R2,0
            ADD R2,R2,1
            BR loop       ; branchement vers le début de boucle
endloop:    RET
```

$\hookrightarrow$ Les dépendances de contrôle ne peuvent pas être supprimées, et peuvent ralentir l'exécution des instructions dans le pipeline.

  - Une solution simple est que le pipeline cesse de charger de nouvelles instructions tant que l'instruction de branchement n'est pas traitée.
  - Une technique appelée *prédiction de branchement* permet cependant d'améliorer le traitement des branchements conditionnels dans le cas moyen.

Parallélisme d'instruction

Dans la sous-section précédente, on vient de voir que :

- on ne peut pas faire se chevaucher très bien l'exécution de deux instructions présentant des dépendances entre-elles ;
- un ensemble d'instructions indépendantes peut s'exécuter très efficacement dans un pipeline.

En fait, deux instructions indépendantes peuvent être exécutées *en parallèle* si le chemin de données du processeur le permet.

Dans les programmes courants, il existe de nombreuses instructions indépendantes les unes des autres : on parle de *parallélisme d'instruction*.

L'exécution pipelinée permet d'exploiter une partie de ce parallélisme d'instruction pour améliorer les performances de l'exécution des programmes.

Plan

Dans ce cours :

- nous avons donné une idée de l'organisation de la *mémoire centrale* d'un ordinateur, et montré comment elle peut être secondée par des mémoires secondaires via une organisation hiérarchique.

- nous avons vu que la *mémoire cache* permet de diminuer les pénalités lors des accès à la mémoire, si le programmeur respecte le principe de localité.

- nous avons vu comment le *parallélisme d'instruction* peut être exploité. D'autres techniques sont aussi utilisées : exécution superscalaire, instructions exploitant le parallélime de données, et processeurs multicœurs.