Architecture des Ordinateurs Avancée (L3)

Cours 2 - L'organisation de la mémoire

Carine Pivoteau [1]

1. En collaboration avec X. Goaoc

Retour sur le TP 1

L'important, c'est l'ANALYSE de ce que vous observez ! Faites preuve d'esprit critique et soyez précis dans vos arguments !

- Le test == entre flottants n'est pas fiable :
  - augmenter la précision n'améliore pas le taux de réussite...
  - ...mais réduit les écarts absolus et relatifs
  - différents langages $\rightarrow$ même combat.

- Le test =< entre flottants est fiable.
  $\triangleright$ On peut l'utiliser pour faire un **test d'égalité à précision fixée**.

- Le nombre, l'ordre et la précision des opérations ont beaucoup d'importance sur la propagation des erreurs. La seule façon de lutter, c'est d'en avoir conscience...

- Le calcul sur les puissances de 2 exact :)

La mémoire... Non, les mémoires !

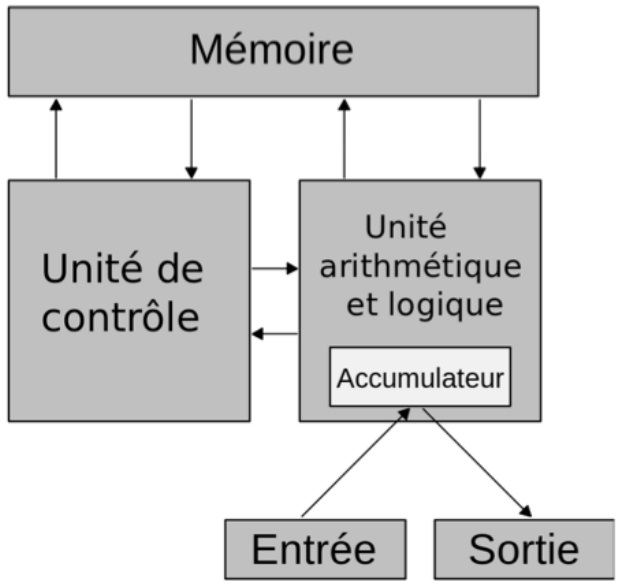
Goulôt de Von Neumann – J. Backus, 1977

La quantité de mémoire disponible augmente très vite, mais pas la capacité de transfert avec le processeur.

Il y a deux raison principales :

- quand la volume augmente, l'adressage devient plus compliqué ;
- l'emplacement physique de la mémoire s'éloigne du processeur.

Si le taux de transfert est plus lent que la vitesse de traitement du CPU, ce dernier perd du temps en attente d'information.



https://en.wikipedia.org/wiki/Von_Neumann_architecture

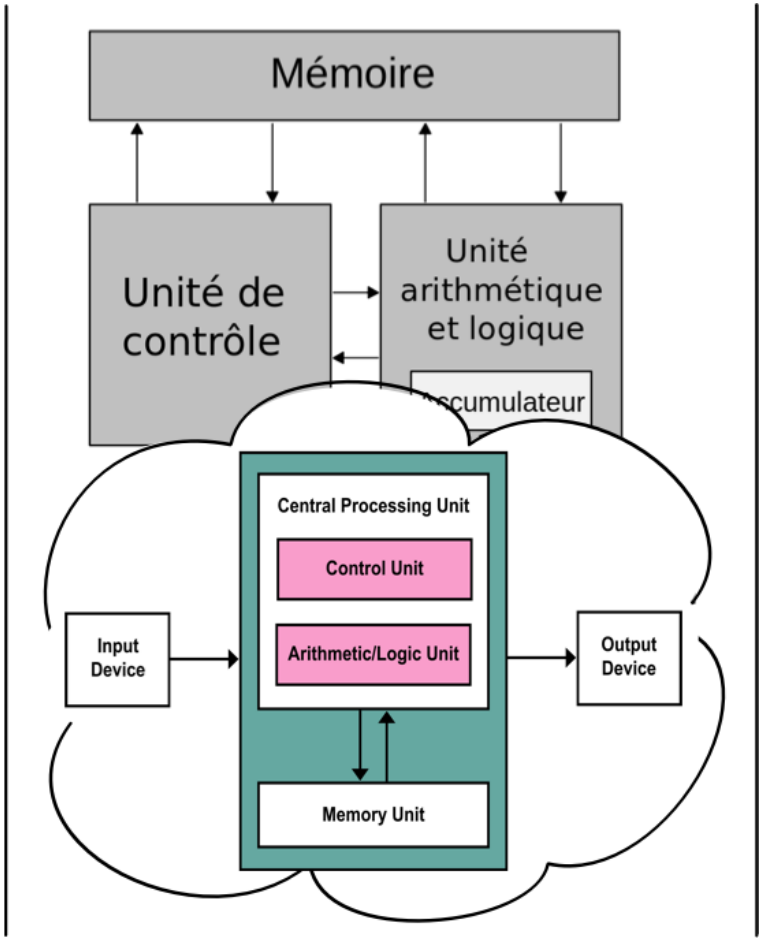
Goulôt de Von Neumann – J. Backus, 1977

La quantité de mémoire disponible augmente très vite, mais pas la capacité de transfert avec le processeur.

Il y a deux raison principales :

- quand la volume augmente, l'adressage devient plus compliqué ;
- l'emplacement physique de la mémoire s'éloigne du processeur.

Si le taux de transfert est plus lent que la vitesse de traitement du CPU, ce dernier perd du temps en attente d'information.



https://en.wikipedia.org/wiki/Von_Neumann_architecture

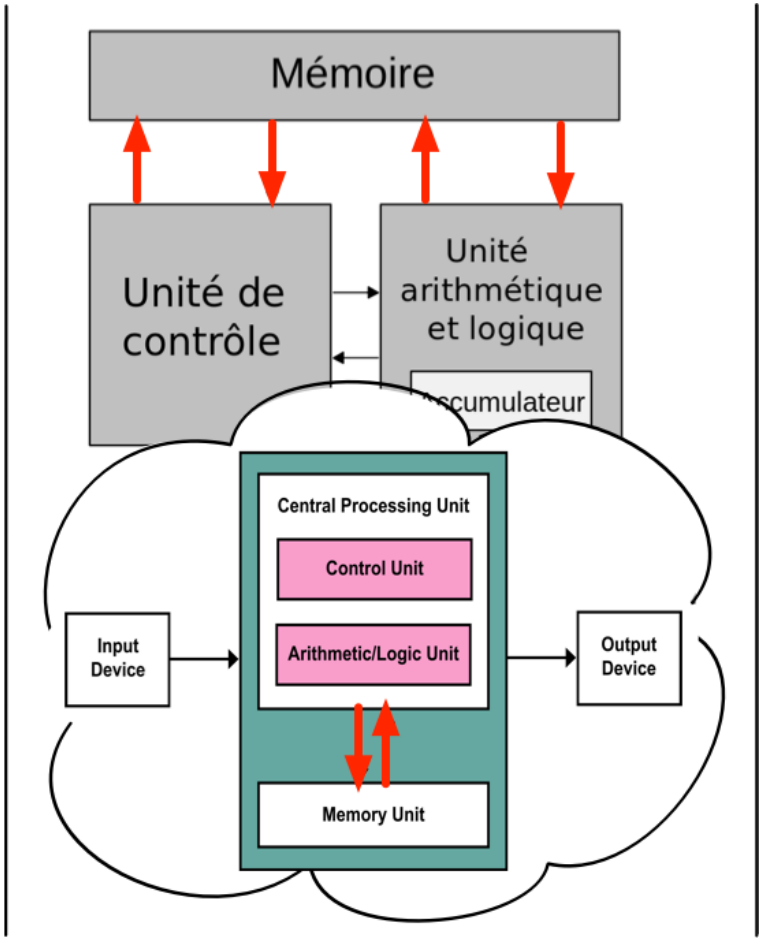
Goulôt de Von Neumann – J. Backus, 1977

La quantité de mémoire disponible augmente très vite, mais pas la capacité de transfert avec le processeur.

Il y a deux raison principales :

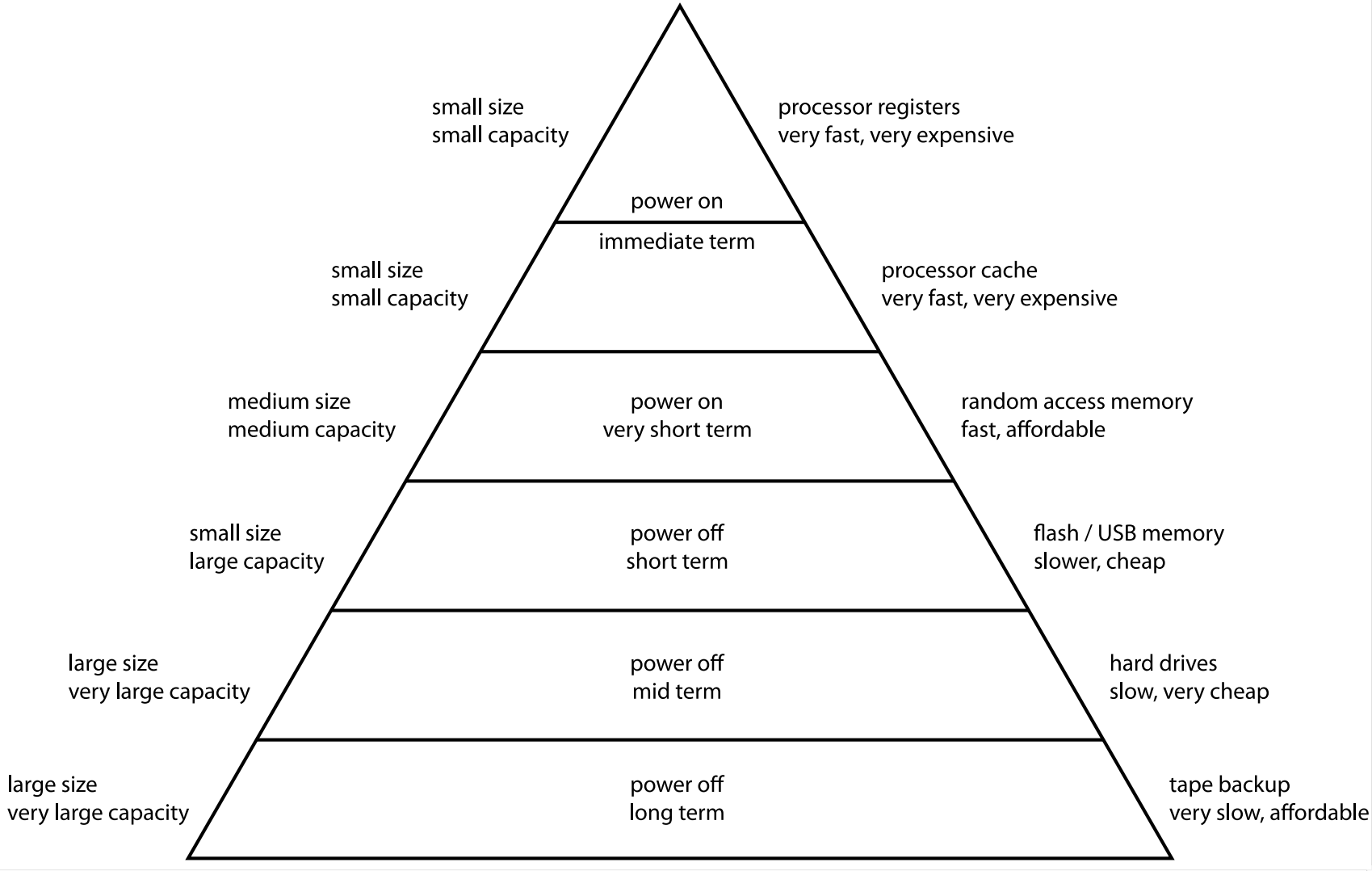
- quand la volume augmente, l'adressage devient plus compliqué ;
- l'emplacement physique de la mémoire s'éloigne du processeur.

Si le taux de transfert est plus lent que la vitesse de traitement du CPU, ce dernier perd du temps en attente d'information.



`https://en.wikipedia.org/wiki/Von_Neumann_architecture`

Computer Memory Hierarchy



source : `https://en.wikipedia.org/wiki/Memory_hierarchy`

La RAM (*Random-access memory*)

- On l'appelle ainsi car elle permet l'accès en lecture ou écriture **avec sensiblement le même temps de réponse** à n'importe quelle donnée.
- Ce résultat est obtenu par adressage direct, c'est à dire que l'on accède à une donnée directement grâce à son adresse. C'est le principe des tableaux manipulés par des pointeurs en C.
- Par opposition aux supports à accès séquentiel (des bandes magnétiques, par exemple). C'est le principe d'une liste chaînée.

Rappel : en C, un pointeur est une variable interprétée comme une adresse. Ainsi, déclarer `int* p` *crée une variable* `p` *dont la valeur est interprétée comme l'adresse d'une variable de type* `int`*. Pour ce cours, vous devez maîtriser l'arithmétique des pointeurs. Par exemple, comment la valeur de* `p` *change-t-elle lorsque l'on fait* `p++` *?*

Exo : le coût d'un accès mémoire est-il constant ?

On va essayer de répondre à cette question :

- Écrire un programme C qui
  - alloue un tableau d'entiers dont la taille n est prise en argument de la ligne de commande ;
  - remplit le tableau avec des 1.
- Ajouter une fonction qui calcule la somme de n cases du tableau espacées de 7 (avec un modulo quand on revient à 0).
- Ajouter une deuxième fonction qui calcule la somme de n cases du tableau espacées de 42 (avec modulo aussi).
- Dans le `main`, mesurer le temps d'exécution de chaque fonction.
- Compiler (sans optimisations) et lancer les programmes avec $n = 10^3, 10^4, 10^5, \ldots$
- Que constate-t-on ?
- Pouvez-vous deviner la taille du cache ?

(fichiers **demo-C2-1.c**, **demo-C2-001.c**)

Exo : le coût d'un accès mémoire est-il constant ?

On va essayer de répondre à cette question :

- Écrire un programme C qui
  - alloue un tableau d'entiers dont la taille n est prise en argument de la ligne de commande ;
  - remplit le tableau avec des 1.
- Ajouter une fonction qui calcule la somme de n cases du tableau espacées de 7 (avec un modulo quand on revient à 0).
- Ajouter une deuxième fonction qui calcule la somme de n cases du tableau espacées de 42 (avec modulo aussi).
- Dans le `main`, mesurer le temps d'exécution de chaque fonction.
- Compiler (sans optimisations) et lancer les programmes avec $n = 10^3, 10^4, 10^5, \ldots$
- Que constate-t-on ?
- **Pouvez-vous deviner la taille du cache ?**

(fichiers **demo-C2-1.c**, **demo-C2-001.c**)

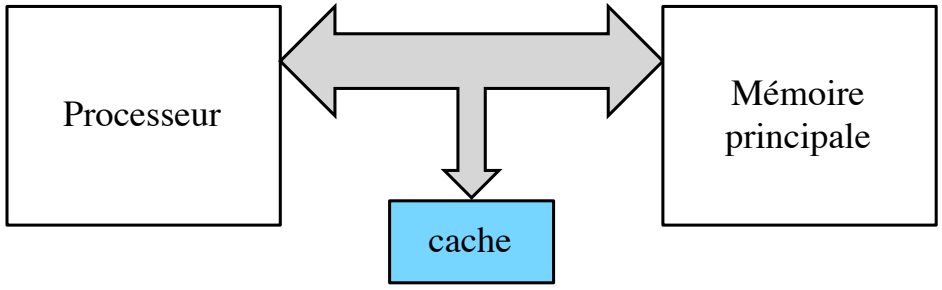
Système à deux niveaux mémoire...

Depuis plusieurs décennies, les ordinateurs utilisent des **mémoires hiérarchiques** pour améliorer l'accès des processeurs aux données :

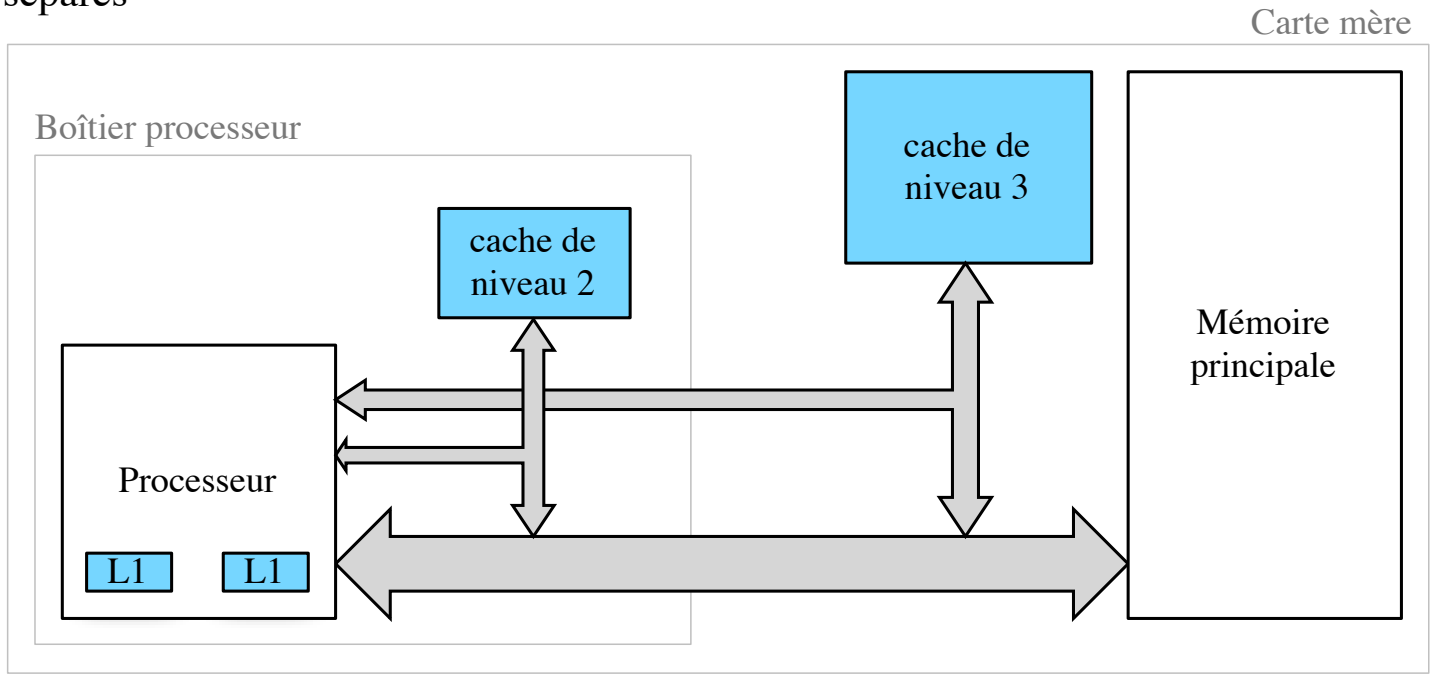
- Deux types de mémoire distincts :
  - l'une est grande mais lente (mémoire centrale) : elle stocke l'ensemble des données.
  - l'autre est rapide mais petite (**mémoire cache**) : elle *essaie* de garder une copie des données "les plus utilisées".
- Lorsque le processeur souhaite accéder à une donnée :
  - il la demande à la mémoire cache ;
  - si la mémoire cache dispose d'une copie de la donnée demandée, elle réponds directement au processeur ; c'est un **cache hit**.
  - si la mémoire cache ne dispose pas d'une copie de la donnée demandée, elle la récupère auprès de la mémoire centrale, puis répond directement au processeur ; c'est un **cache miss**.
  - Un **cache hit** est beaucoup plus rapide qu'un **cache miss** !

... voire plus.

Cache unique



Caches séparés



Différents niveaux de cache

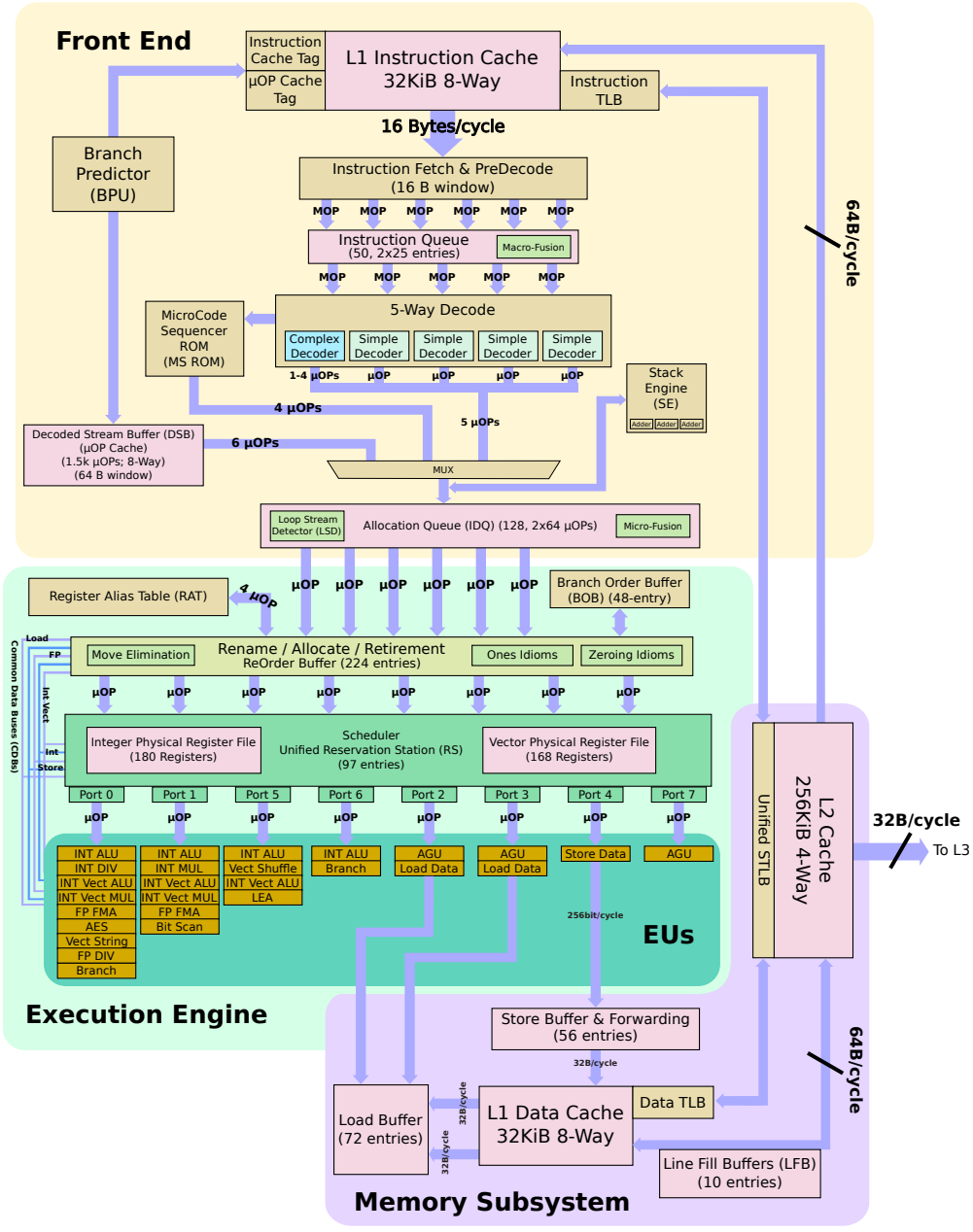
Le système mémoire d'une machine *typique* comporte 4 niveaux hiérarchiques : 3 niveaux de mémoire cache (L1, L2, L3) et la mémoire principale (RAM). Sur les architectures récentes :

- Le niveau **L1** comporte en général deux caches indépendants, un pour les instructions et l'autre pour les données.
- Le cache L1 d'*instructions* est utilisé par le CPU pour des accès mémoire correspondant à la lecture du programme à exécuter.
- Les caches **L1 et L2 sont internes à chaque cœur du processeur**, comme on peut le voir par exemple sur le diagramme de l'architecture Skylake.
- Le niveau **L3 est partagé** entre plusieurs cœurs.

Architecture Skylake
(Intel Core i3, i5, i7)

source :

`https://en.wikichip.org/wiki/skylake`



Panorama des mémoires

Il existe de nombreux supports mémoire, dont la fonction est de stocker puis restituer de l'information : RAM, disque dur (HDD, SSD), USB, CD ou DVD, mémoire cache, mais aussi bande magnétique, carte perforée, papier, ...
Ces supports diffèrent en plusieurs points : coût, taille, performances (réactivité et débit), permanence, durée de vie, fiabilité, ...

	débit max.	temps d'accès	$\sim$ taille
disque HDD	140 Mo/s.	$\sim$ 3-12 ms.	$<$ 50 To
disque SSD	600 Mo/s.	$\sim$ 0.1 ms.	$<$ 5 To
RAM	10 Go/s.	60-100 cycles ($\sim$ 50-60 ns.)	qq. Go
L2 cache	200 Go/s.	10 cycles ($\sim$ 3-5 ns.)	qq. Mo
L1 cache	700 Go/s.	4 cycles ($\sim$ 1-2 ns.)	100 Ko

Votre machine

- Caractéristiques du processeur : dans le fichier `/proc/cpuinfo`.
- Informations complémentaires sur la mémoire : dans le fichier `/proc/meminfo`.
- Informations précises sur le cache : dans le dossier `/sys/devices/system/cpu/cpu0/cache/`.
  - Différents sous-répertoires pour les différents niveau de cache.
  - On peut notamment y trouver la taille de chaque cache en octets.
  - Il y a aussi des informations plus précises sur la forme des ces caches (suite du cours).
- Et sur un mac : `sysctl -a`

Principe des tiroirs *(pigeonhole principle)*

Constat : la mémoire cache est plus petite que le mémoire centrale... donc on ne peut pas tout stocker en cache.

Comment choisit-on :

- quelle donnée y est stockée ?
- où est stockée cette donnée dans le cache ?

Principe des tiroirs *(pigeonhole principle)*

Constat : la mémoire cache est plus petite que le mémoire centrale... donc on ne peut pas tout stocker en cache.

Comment choisit-on :

- quelle donnée y est stockée ?
- où est stockée cette donnée dans le cache ?

... en faisant des compromis.

Principe de localité

À tout instant, un programme accède à une *petite* partie de son espace d'adressage ; 2 types de localité :

- **Temporelle** : Si un mot mémoire a été utilisé récemment, il a plus de chances d'être *réutilisé* (exemple : dans les boucles) ;
- **Spatiale** : Si un mot a été utilisé récemment, les mots avec des *adresses voisines* ont plus de chances d'être utilisés (exemple : pile, tableaux) ;

Idée : on place dans le **cache** (mémoire proche du processeur) :

- Les données les *plus récemment adressées* ;
- Les données en *blocs* (lignes de cache).

La ligne de cache

- L'unité de transfert entre mémoire centrale et cache est la **ligne de cache**, qui fait généralement 64 ou 128 octets.

  Le cache peut stocker un nombre fixe de lignes de cache, donc le chargement d'une nouvelle ligne de cache peut écraser une des lignes précédemment chargées.

  Le choix de cette ligne correspond à la *stratégie de pagination*.

- On aimerait bien que la ligne écrasée soit celle qui a été utilisée le moins récemment (*LRU* = *least recently used*), pour bénéficier de la localité temporelle... Mais comment faire cela à moindre coût ?

La ligne de cache

- L'unité de transfert entre mémoire centrale et cache est la **ligne de cache**, qui fait généralement 64 ou 128 octets.

  Le cache peut stocker un nombre fixe de lignes de cache, donc le chargement d'une nouvelle ligne de cache peut écraser une des lignes précédemment chargées.

  Le choix de cette ligne correspond à la *stratégie de pagination*.

- On aimerait bien que la ligne écrasée soit celle qui a été utilisée le moins récemment (*LRU* = *least recently used*), pour bénéficier de la localité temporelle... Mais comment faire cela à moindre coût ?

  On voit ça au prochain cours...

Exo : le parcours de matrice

Maintenant, on peux mieux comprendre pourquoi l'ordre de parcours d'une matrice est important.

- Écrire un programme C qui :
  - alloue une matrice d'entiers dont la taille $n \times n$ est prise en argument de la ligne de commande,
  - et la remplit avec des 1.
  - Ajouter deux fonctions qui calculent et la somme de ses cases :
    - une qui parcourt la matrice **en ligne et en colonne** ;
    - et l'autre **en colonne puis en ligne**.
  - Rajouter dans le `main` de quoi mesurer leur temps d'exécution.
- Lancer le programme avec $n = 100, 200, 400, \ldots$ et mesurer le temps d'exécution.
- Que constate-t-on ?

(fichiers **demo-C2-3.c**, **demo-C2-003.c**)

Même expérience en mesurant le nombre de cache-misses avec la librairie `PAPI` : `https://icl.utk.edu/papi/`.

```
pivoteau@aquitaine 14:18> ./a.out 100
lignes-colonnes L1 misses :      66748, L2 misses :        121, L3 misses :          0, n = 100 (39 K), 120146 cycles
colonnes-lignes L1 misses :      64360, L2 misses :         72, L3 misses :          0, n = 100 (39 K), 129133 cycles
pivoteau@aquitaine 14:18> ./a.out 100
lignes-colonnes L1 misses :      66803, L2 misses :         98, L3 misses :          1, n = 100 (39 K), 170604 cycles
colonnes-lignes L1 misses :      64437, L2 misses :        160, L3 misses :          0, n = 100 (39 K), 120389 cycles
pivoteau@aquitaine 14:18> ./a.out 100
lignes-colonnes L1 misses :      66771, L2 misses :        104, L3 misses :          0, n = 100 (39 K), 119079 cycles
colonnes-lignes L1 misses :      64221, L2 misses :        102, L3 misses :          0, n = 100 (39 K), 129961 cycles
pivoteau@aquitaine 14:18> ./a.out 100
lignes-colonnes L1 misses :      66766, L2 misses :        246, L3 misses :          0, n = 100 (39 K), 131200 cycles
colonnes-lignes L1 misses :      64328, L2 misses :        134, L3 misses :          0, n = 100 (39 K), 128486 cycles
pivoteau@aquitaine 14:18> ./a.out 100
lignes-colonnes L1 misses :      66772, L2 misses :        124, L3 misses :          1, n = 100 (39 K), 115063 cycles
colonnes-lignes L1 misses :      64301, L2 misses :        149, L3 misses :          0, n = 100 (39 K), 120651 cycles
pivoteau@aquitaine 14:18>
pivoteau@aquitaine 14:19> ./a.out 200
lignes-colonnes L1 misses :     257795, L2 misses :       5210, L3 misses :          3, n = 200 (156 K), 457076 cycles
colonnes-lignes L1 misses :     327288, L2 misses :      17406, L3 misses :          0, n = 200 (156 K), 509737 cycles
pivoteau@aquitaine 14:19> ./a.out 200
lignes-colonnes L1 misses :     257855, L2 misses :       3085, L3 misses :          2, n = 200 (156 K), 461686 cycles
colonnes-lignes L1 misses :     326238, L2 misses :      14374, L3 misses :          0, n = 200 (156 K), 480511 cycles
pivoteau@aquitaine 14:20>
pivoteau@aquitaine 14:20> ./a.out 400
lignes-colonnes L1 misses :    1015499, L2 misses :      36911, L3 misses :          0, n = 400 (625 K), 1770793 cycles
colonnes-lignes L1 misses :    9500524, L2 misses :    1511312, L3 misses :          0, n = 400 (625 K), 1981710 cycles
pivoteau@aquitaine 14:20>
pivoteau@aquitaine 14:20> ./a.out 800
lignes-colonnes L1 misses :    4039073, L2 misses :     152378, L3 misses :         18, n = 800 (2500 K), 7048863 cycles
colonnes-lignes L1 misses :   82196785, L2 misses :    6117841, L3 misses :        366, n = 800 (2500 K), 12061000 cycles
pivoteau@aquitaine 14:20>
pivoteau@aquitaine 14:20> ./a.out 1600
lignes-colonnes L1 misses :   16110815, L2 misses :     865140, L3 misses :     603912, n = 1600 (10000 K), 28999183 cycles
colonnes-lignes L1 misses :  339203167, L2 misses :   23458600, L3 misses :   11164875, n = 1600 (10000 K), 65197791 cycles
pivoteau@aquitaine 14:20> ./a.out 3200
lignes-colonnes L1 misses :   64285115, L2 misses :    3845834, L3 misses :    3109845, n = 3200 (40000 K), 116329491 cycles
colonnes-lignes L1 misses : 1556074723, L2 misses :  773858320, L3 misses :   62300150, n = 3200 (40000 K), 296527885 cycles
```

Est-ce qu'on peut en déduire la taille des lignes de cache ?

Question bonus : problèmes de cache vs. complexité

Au vu de ce qui précède, à votre avis, y a-t-il des cas où l'on a intérêt à utiliser une liste chaînée plutôt qu'un tableau ?

Question bonus : problèmes de cache vs. complexité

Au vu de ce qui précède, à votre avis, y a-t-il des cas où l'on a intérêt à utiliser une liste chaînée plutôt qu'un tableau ?

▷ Attention, le cache peut avoir un effet très important sur le temps d'exécution d'un programme... mais en terme de complexité, ça ne change rien... (ça ne modifie que les constantes).