# Architecture

François Trahay

module CSC4508/M2

Avril 2018

# Plan du document

1 Introduction . . . . . . . . . . 3
2 Processeur séquentiel . . . . . . . . . . 5
3 Pipeline . . . . . . . . . . 6
4 Parallel Processing . . . . . . . . . . 14
5 Hiérarchie mémoire . . . . . . . . . . 19

# 1 Introduction

Pourquoi ce cours ?

- Comprendre ce qui se passe dans la partie “hardware” de la pile d’exécution
- Pour écrire des programmes adaptés aux machines modernes

# 1.1 Loi de Moore

1965 – 2005

- Loi de Moore (1965): le nombre de transistors des micro processeurs double tous les deux ans
- La finesse de gravure du processeur diminue
- La fréquence d'horloge augmente

–> Augmentation des performances du processeur

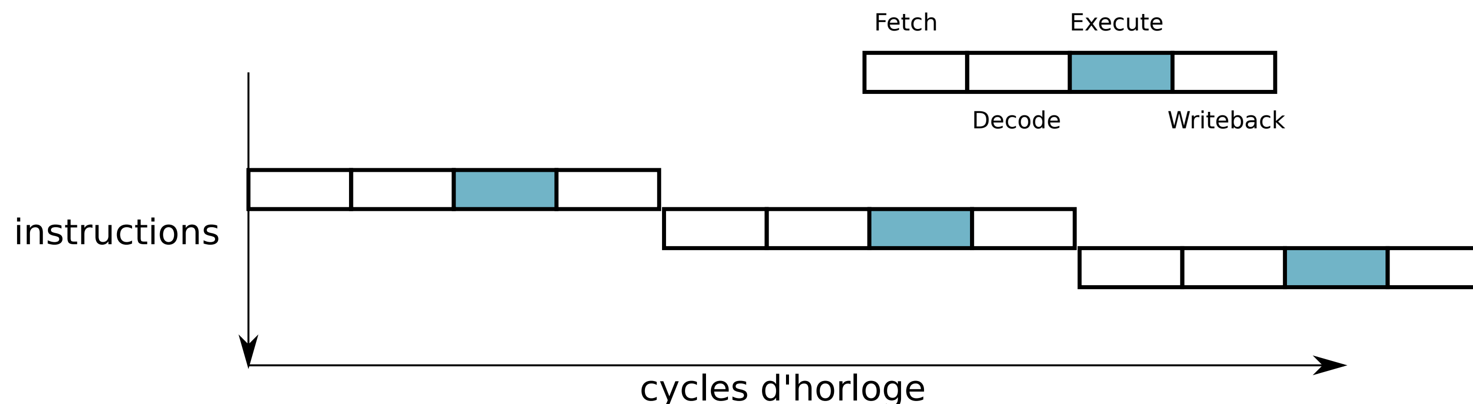
Depuis 2005

- La finesse de gravure continue de diminuer (mais moins vite)
- La fréquence d'horloge n'augmente plus
  - Dissipation thermique dépend de la fréquence, du nombre de transistors, de la taille des transistors
  - Si on diminue la taille des transistors, il faut réduire la fréquence

# 2 Processeur séquentiel

- Une instruction nécessite N étapes
  - *Fetch*: chargement de l'instruction depuis la mémoire
  - *Decode*: identification de l'instruction
  - *Execute*: exécution de l'instruction
  - *Writeback*: stockage du résultat
- Chaque étape est traitée par un circuit du processeur
- La plupart des circuits n'est pas utilisée à chaque étape

→ Une instruction est exécutée tous les N cycles



# 3 Pipeline

- Pipeline d'instructions
- À chaque étape, plusieurs circuits sont utilisés

→ Une instruction est exécutée à chaque cycle

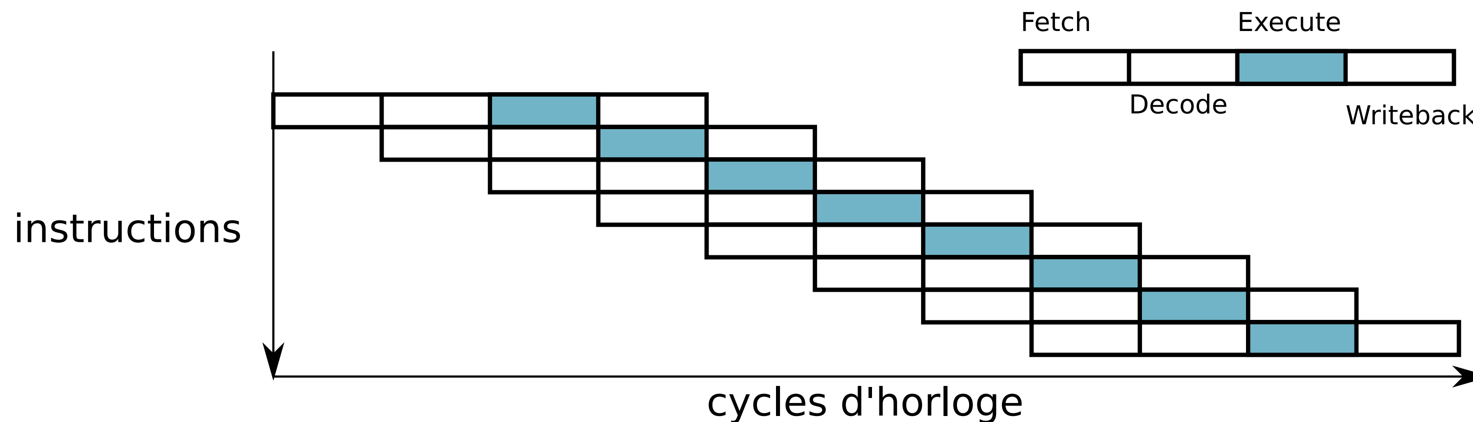



Figure 1: Exécution d'instructions sur un processeur avec *pipeline*

## 3.1 Micro architecture d’un pipeline

- Chaque étage du pipeline est implémenté par un ensemble de portes logiques
- Étage *Execute*: un sous-circuit par type d’opération (unité fonctionnelle)

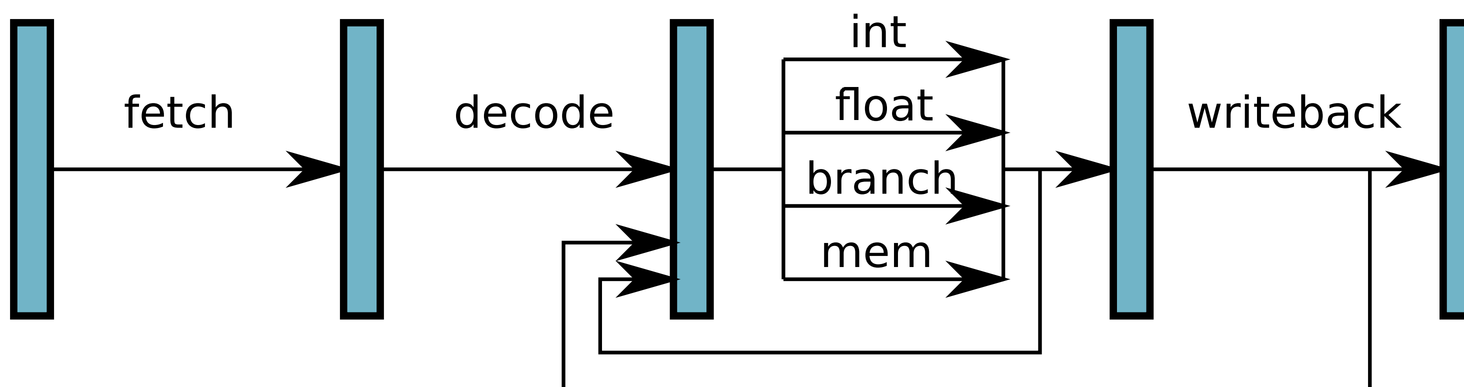



Figure 2: Micro-architecture d’un *pipeline*

## 3.2 Processeurs superscalaires

- Utilisation des différentes unités fonctionnelles simultanément

→ plusieurs instructions exécutées simultanément !

- Nécessité de charger et décoder plusieurs instructions simultanément

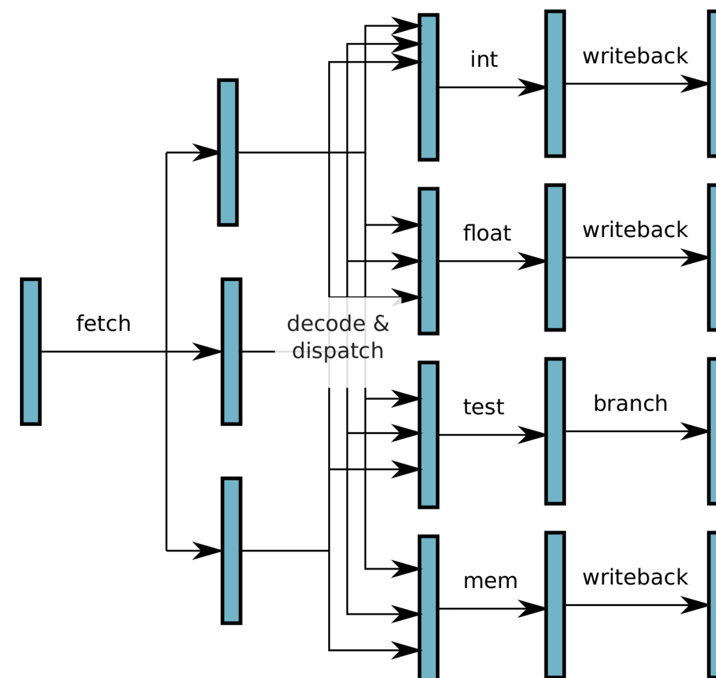
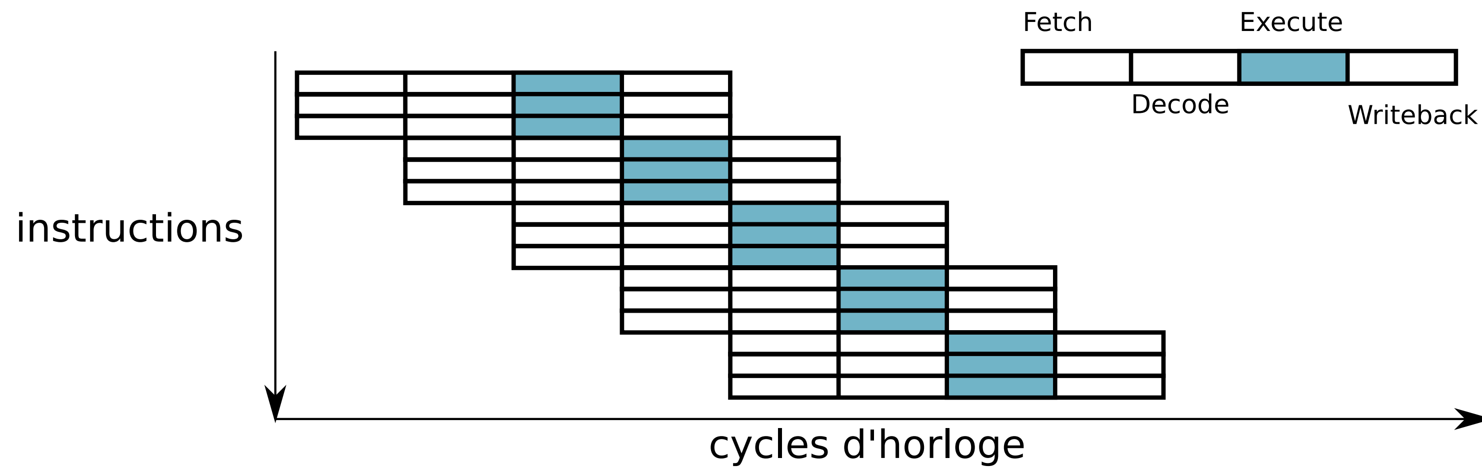



Figure 3: Micro-architecture d'un processeur *superscalaire*

# 3.3 Processeurs superscalaires



## 3.4 Dépendance entre instructions

Limitations du superscalaire:

- il ne doit pas y avoir de dépendance entre les instructions exécutées simultanément.
  - Exemple d'instructions non parallélisables

```
a = b * c;
d = a + 1;
```

- Degré de parallélisme des instructions : *Instruction Level Parallelism* (ILP)
- Les instructions exécutées en parallèles doivent utiliser des unités fonctionnelles différentes

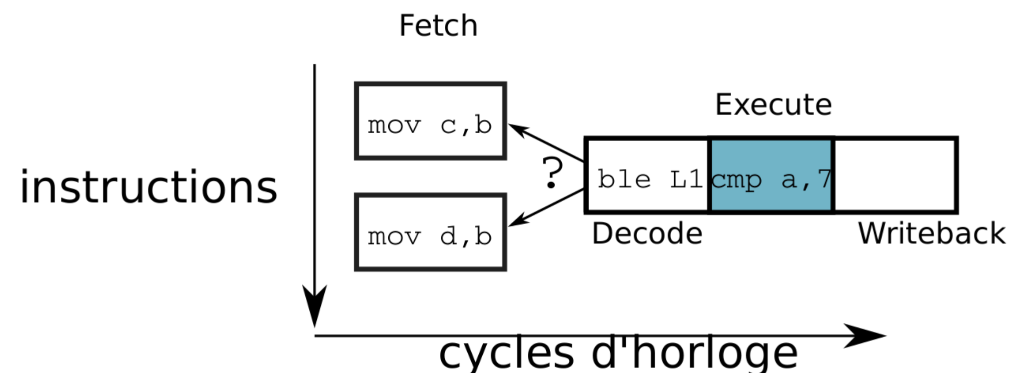
## 3.5 Gestion des branchements

- Comment remplir le pipeline quand les instructions contiennent des branchements conditionnels ?

```
     cmp a, 7     ; a > 7 ?
     ble L1
     mov c, b     ; b = c
     br L2
L1:  mov d, b     ; b = d
L2:  ...
```

- En cas de mauvais choix: il faut “vider” le pipeline

  → perte de temps



## 3.6 Prédiction de branchement

- Le processeur implémente un algorithme de prédiction
- Idée générale:
  - Pour chaque branchement, on se rappelle des résultats précédents

```
0x12 loop:
          ...
0x50      inc eax
0x54      cmpl eax, 10000
0x5A      jl loop
0x5C end_loop:
          ...
```

| *addr* | *branch history* |
|---|---|
| 0x23 | 0011 |
| 0x42 | 1000 |
| 0x5A | 1111 |
| 0x7E | 0000 |

## 3.7 Instructions vectorielles

- De nombreuses applications fonctionnent en mode *Data Parallelism*
- *Single Instruction, Multiple Data* (SIMD): une même opération appliquée à un ensemble de données

```
for(i=0; i<size; i++) {
    C[i] = A[i] * B[i];
}
```

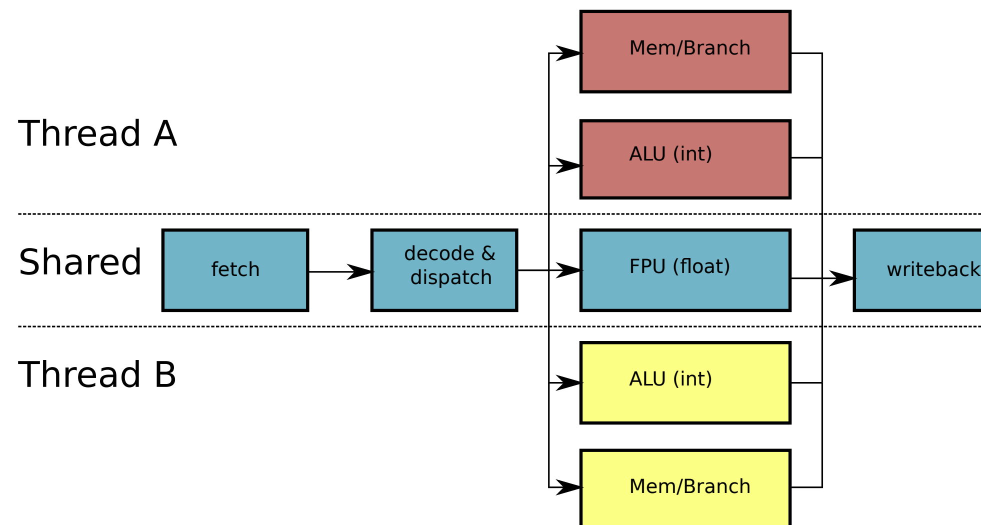
- Exemple: image, scientific computing
- Utilisation d'*instructions vectorielles* (MMX, SSE, AVX, ...)
  - instructions spécifiques à un type de processeur

```
for(i=0; i<size; i+= 8) {
    *pC = _mm_mul_ps(*pA, *pB);
    pA++; pB++; pC++;
}
```

# 4 Parallel Processing

# 4.1 Hyperthreading / SMT

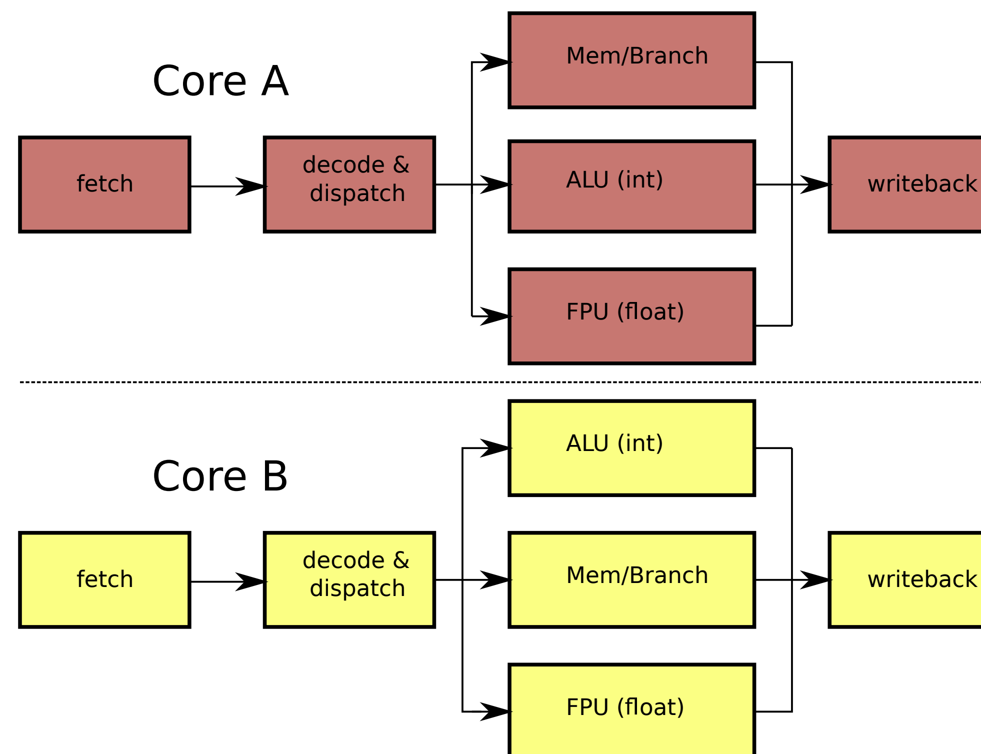
- Problème du superscalaire/vectoriel:
  - Il faut que l'application ait suffisamment de parallélisme à exploiter
  - Il y a d'autres applications qui attendent d'avoir le CPU
- Simultaneous Multi-Threading (SMT):
  - Modifier un processeur superscalaire pour l'exécution de plusieurs threads
  - Duplication de certains circuits
  - Mise en commun de certains circuits



# 4.2 Processeurs multi-cœurs

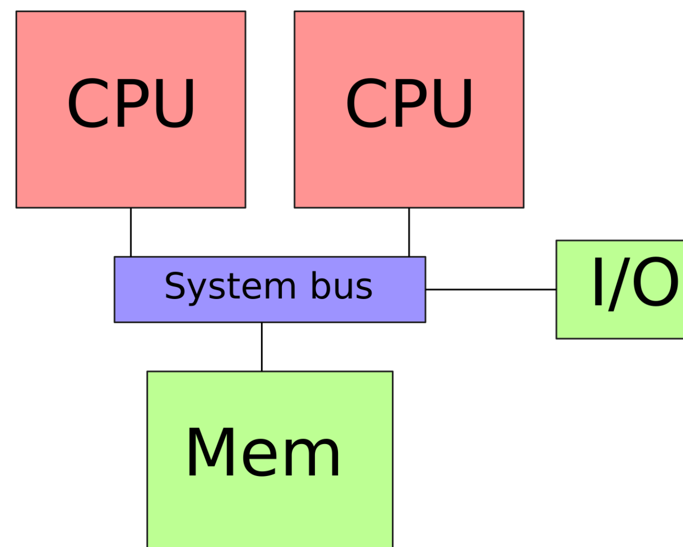
- Scalabilité du SMT limitée
  - *dispatcher* partagé
  - FPU partagé

→ Duplication de tous les circuits



# 4.3 Architectures SMP

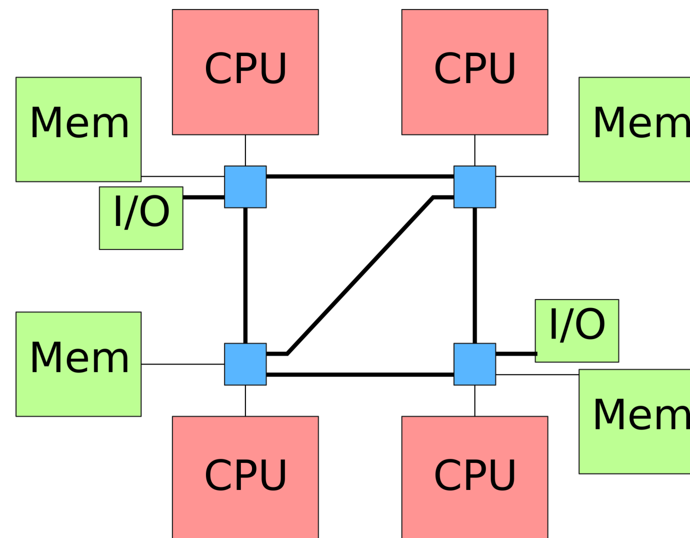
- *Symmetric Multi-Processing*
- Assemblage de processeurs sur une carte mère
- Les processeurs se partagent le bus système
- Les processeurs se partagent la mémoire
- Problème de scalabilité : contention pour l'accès au bus



# 4.4 Architectures NUMA

- *Nœuds* NUMA connectés par un réseau rapide
- Cohérence mémoire entre les processeurs
- Accès privilégié au *banc mémoire* local
- Accès possible (avec un surcoût) aux *bancs mémoire* situés sur les autres nœuds

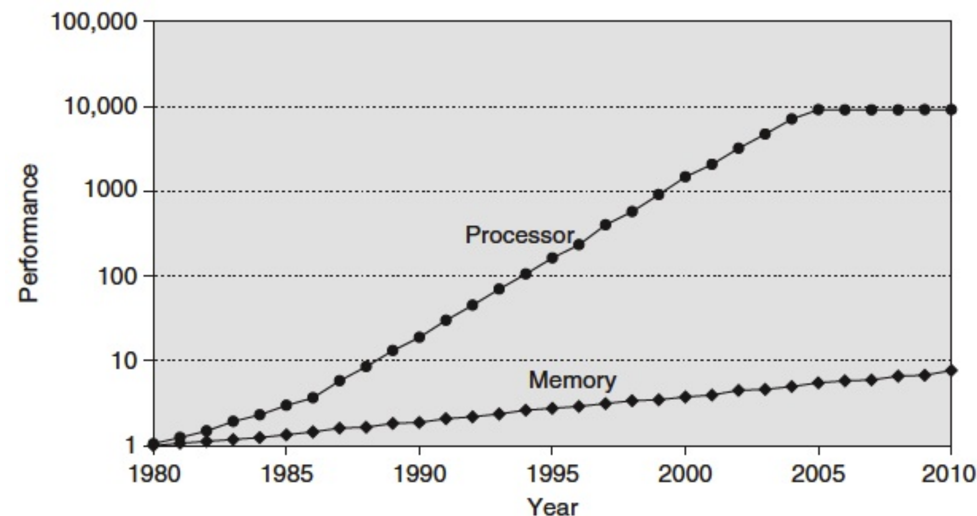
→ *Non-Uniform Memory Architecture*



# 5 Hiérarchie mémoire

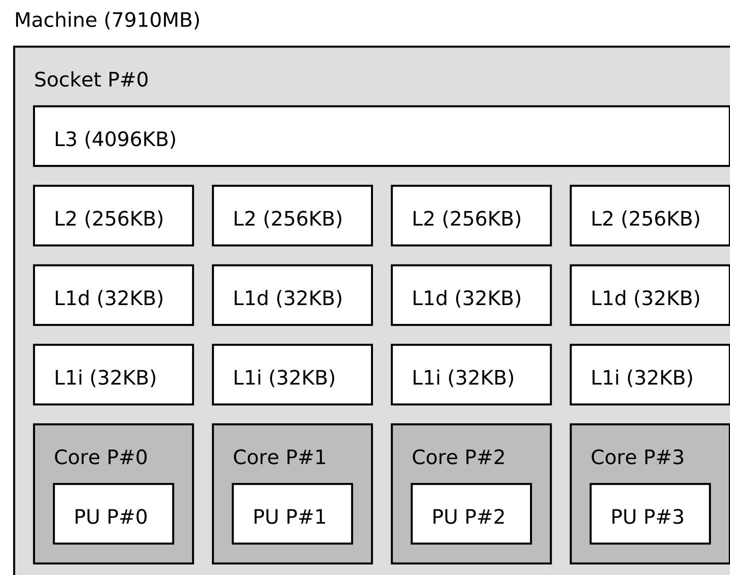
# 5.1 Enjeux

- Jusqu'en 2005: augmentation des perfs des CPUs: 55%/an
- Depuis 2005: augmentation du nombre de cœur par processeur
- Augmentation des perfs de la mémoire : 10%/an
- Ce sont les accès mémoire qui sont coûteux: *Memory Wall*
- Il faut des mécanismes pour améliorer les performances de la mémoire



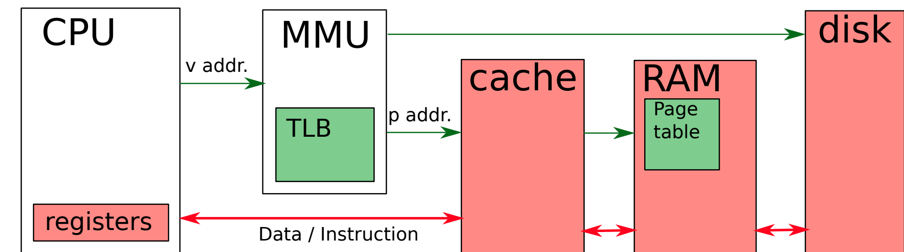
# 5.2 Caches

- Accès à la mémoire (RAM) très coûteux (env. 60 ns – env. 180 cycles)
- Pour accélérer les accès mémoire, utilisation de mémoire cache rapide :
  - Cache L1: très petite capacité (typiquement : 64 Ko), très rapide (env. 4 cycles)
  - Cache L2: petite capacité (typiquement : 256 Ko), rapide (env. 10 cycles)
  - Cache L3: grande capacité (typiquement : entre 4Mo et 20Mo), lent (env. 40 cycles)
- Accès au disque dur (SWAP) très coûteux: env. 40 ms (150 $\mu s$ sur un disque SSD)



# 5.3 Memory Management Unit (MMU)

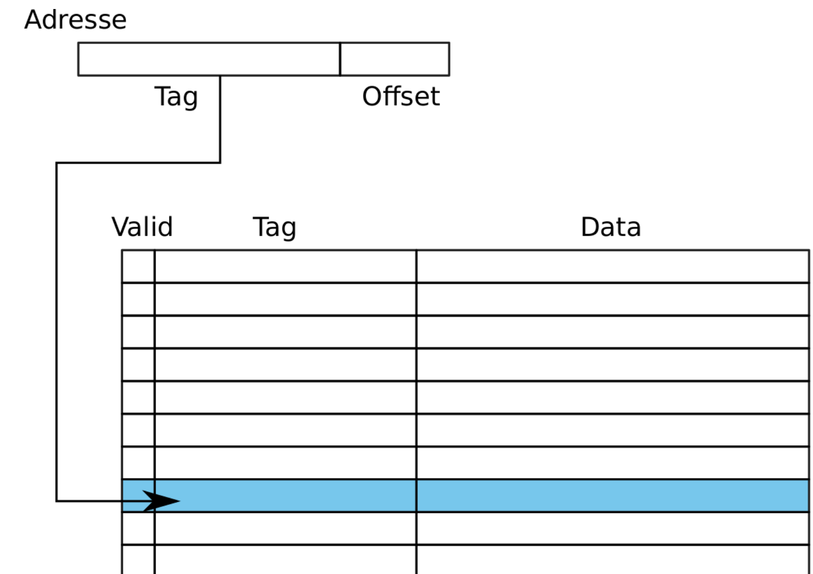
- Traduit les adresses mémoire virtuelles en adresses physiques
- Cherche dans le TLB (Translation Lookaside Buffer), puis dans la table des pages
- Une fois l'adresse physique trouvée, demande les données au cache/mémoire



## 5.3.1 *Fully-associative* caches

- Cache = tableau à N entrées
- À chaque référence, recherche de *Tag* dans le tableau
    - Si trouvé (*cache hit*) et Valid=1 : accès à la *ligne de cache Data*
    - Sinon (*cache miss*) : accès RAM
- Problème: nécessite de parcourir tout le tableau

→ Principalement utilisé pour les petits caches (ex: TLB)

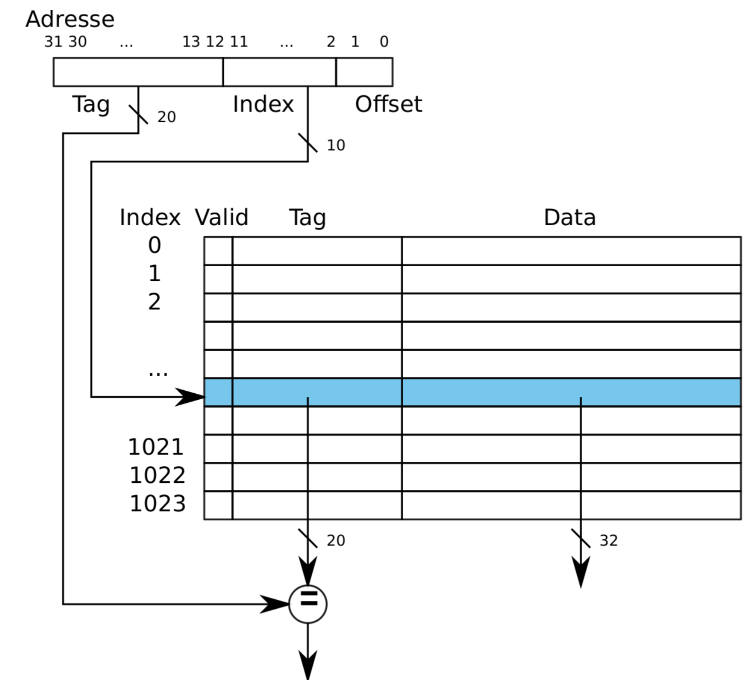


## 5.3.2 *Direct-mapped* caches

- Utilisation des bits de poids faible de l'adresse pour trouver l'*index* de l'entrée dans le cache
- Comparaison du *Tag* (bits de poids fort) de l'adresse et de l'entrée.

→ Accès direct à la ligne de cache

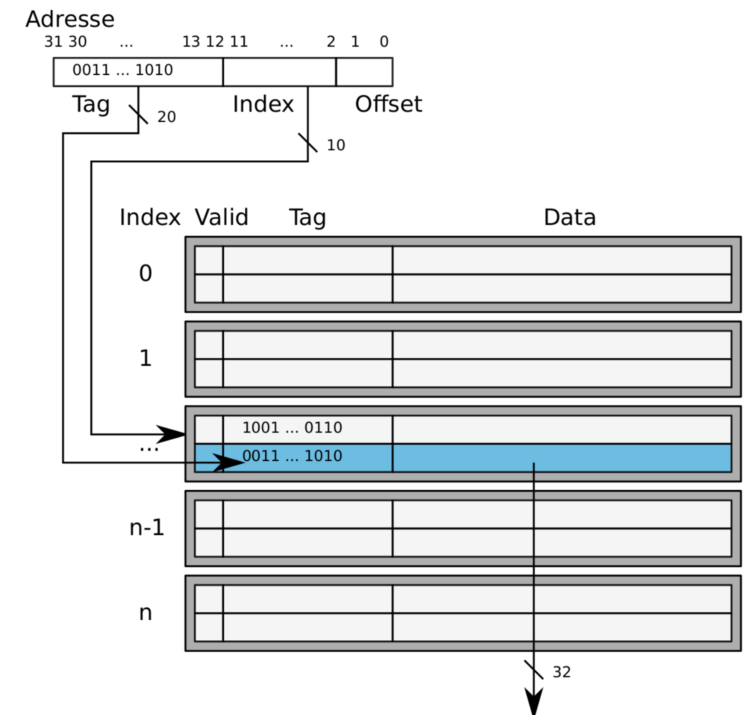
- Attention : risque de collision
  - exemple: *0x12345***67**8 et *0xbff72***67**8



## 5.3.3 *Set-associative* caches

- *Index* pour accéder à un *set* de $K$ lignes de cache
- Recherche du *Tag* parmi les adresses du set

→ Cache associatif K-voies (*K-way associative cache*)



## 5.3.4 Cohérence de cache

- Que faire si 2 threads accèdent à une même zone mémoire ?
  - En lecture : réplication dans les caches locaux
  - En écriture : nécessité d'invalider les données dans les autres caches
    - *Cache snooping*: le cache envoie un message qui invalide les autres caches

## Bibliographie du chapitre