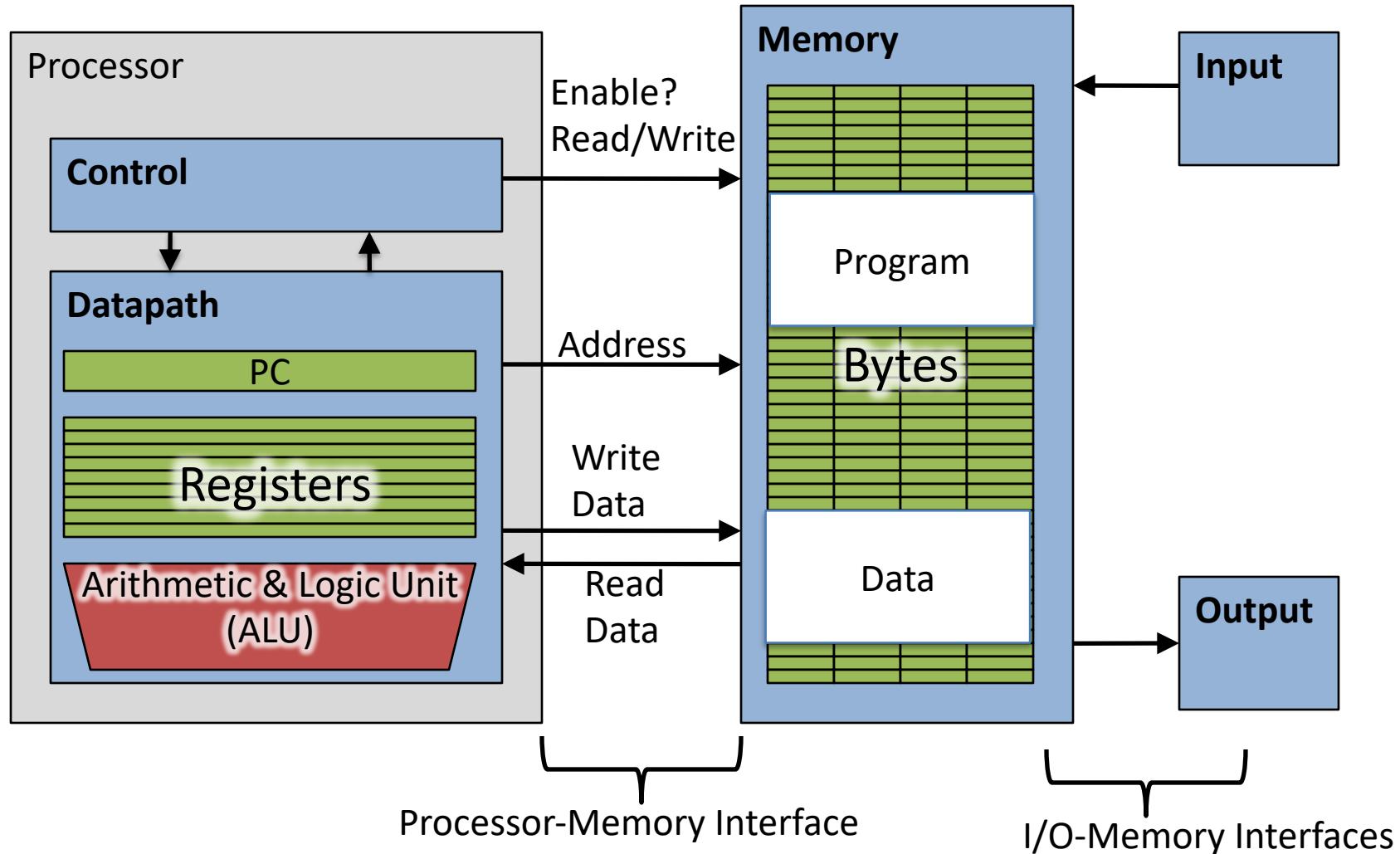


# Architecture des Processseurs

Gestion de la mémoire

# Components of a Computer



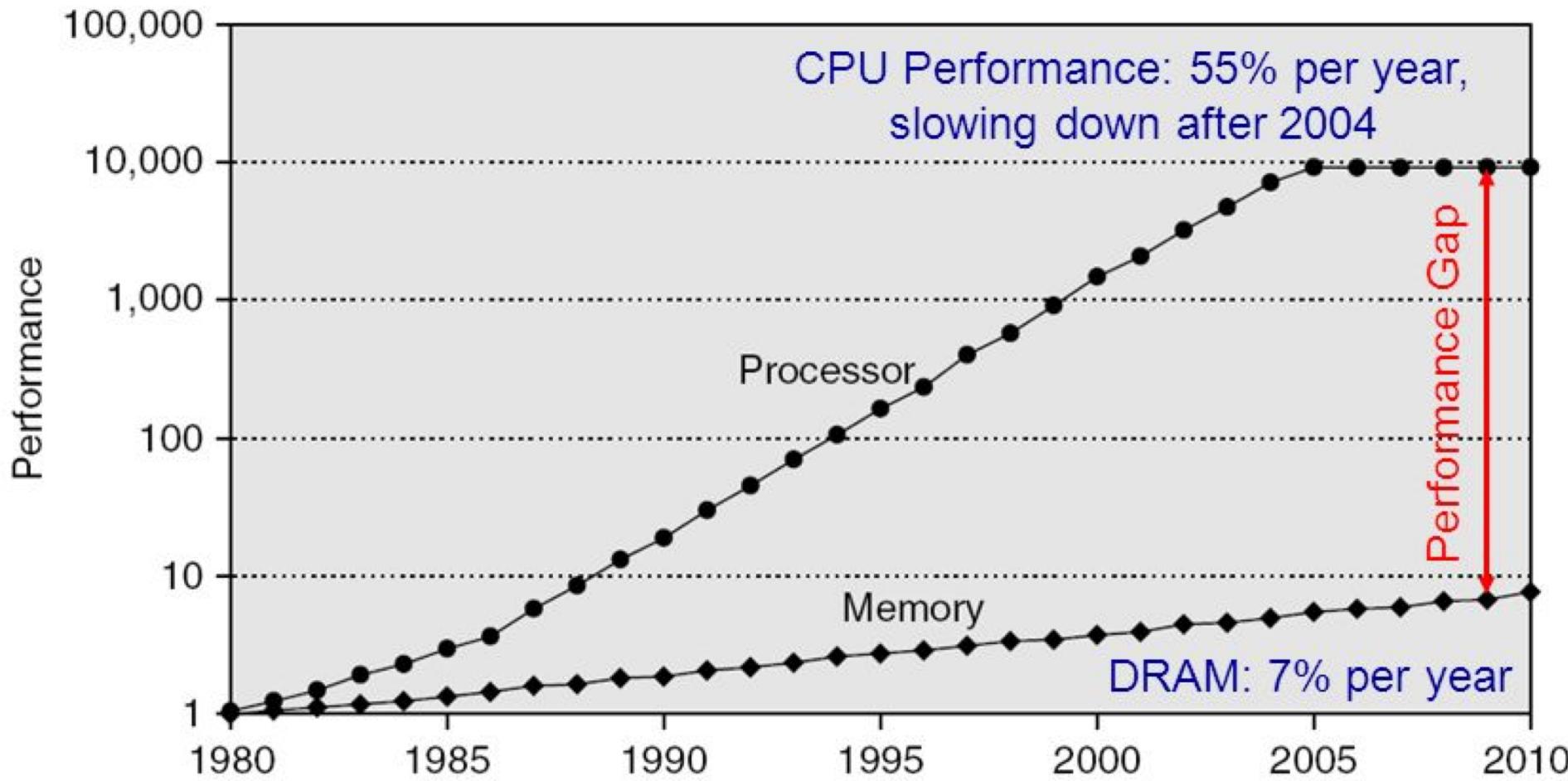
# Performance

□ Le temps passé à attendre une réponse de la mémoire (**suspension mémoire** ou **attente mémoire**) a un impact fondamental sur le temps d'exécution d'un programme:

- Temps d'exécution = (# cycles d'exécution + # cycles d'attente mémoire) × Temps de cycle
- La **pénalité d'accès** est le temps (nombre des cycles) nécessaire pour transférer une donnée de la mémoire au processeur.
- Cycles d'attente mémoire = # accès × pénalité d'accès
- Pénalité d'accès = # instructions × # accès par instruction × Pénalité d'accès

□ Le **cache** est un moyen de réduire la pénalité d'accès.

# Processeur vs DRAM (Latence)



En 1980 un microprocesseur exécute une instruction pendant un accès en DRAM

**La lenteur des accès DRAM a un impact très fort sur les performances du système!**

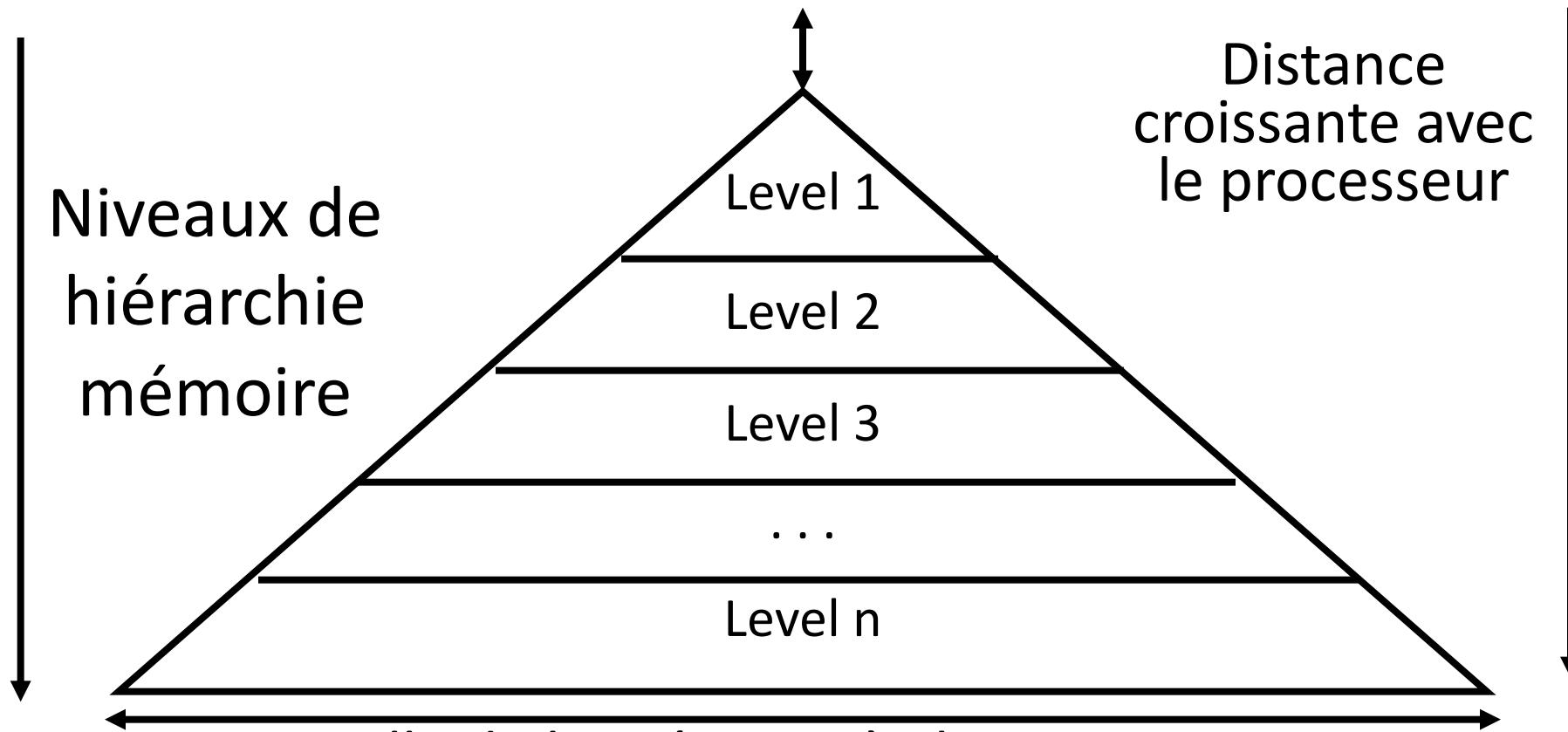
En 2017 un microprocesseur exécute 1000 instructions pendant un accès en DRAM

# Que faire pour lutter contre les problèmes de latence?

- Vous devez faire une étude bibliographique sur un sujet
  - Vos recherches à la B.U. vous indiquent une dizaine de livres intéressants sur le sujet
  - Pb la BU est à l'autre bout de la ville et l'enregistrement d'un livre à la BU est très long...
  - Qu'est ce que vous faites?
    - Vous allez une seule fois à la BU, prenez le max de livres autorisé, les déposez sur votre bureau et vous si vous avez besoin d'y accéder l'accès sera rapide.
    - C'est exactement le fonctionnement des mémoires caches...
    - Quelque problème (comment optimiser le max? comment choisir les livres dont la probabilité d'être utiles est maximale?)

# Hiérarchie Mémoire

Processeur

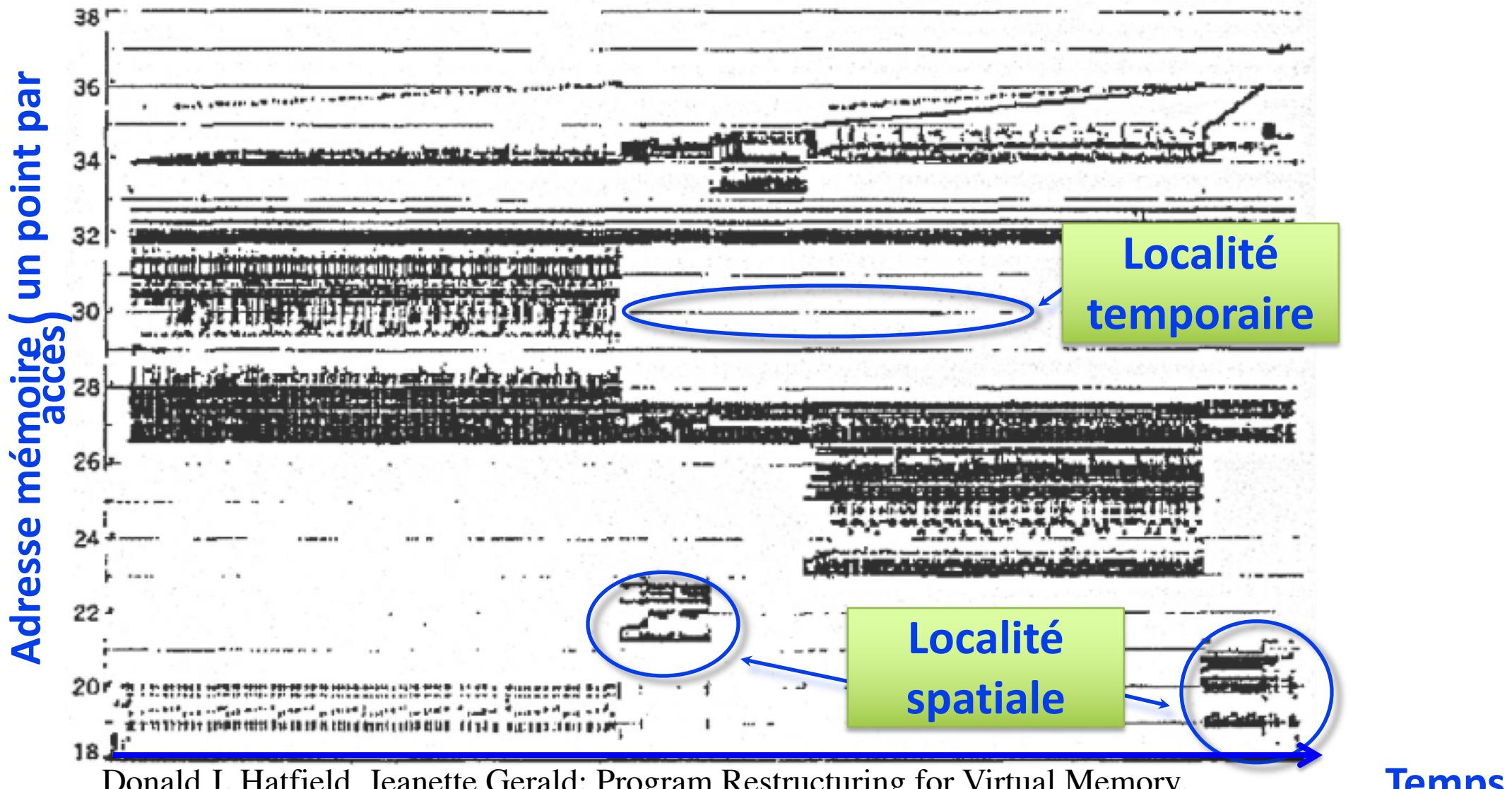


Taille de la mémoire à chaque niveau  
*Plus on s'éloigne du processeur plus le bit de mémoire est chère et plus son temps d'accès est long*

# Principes de localité

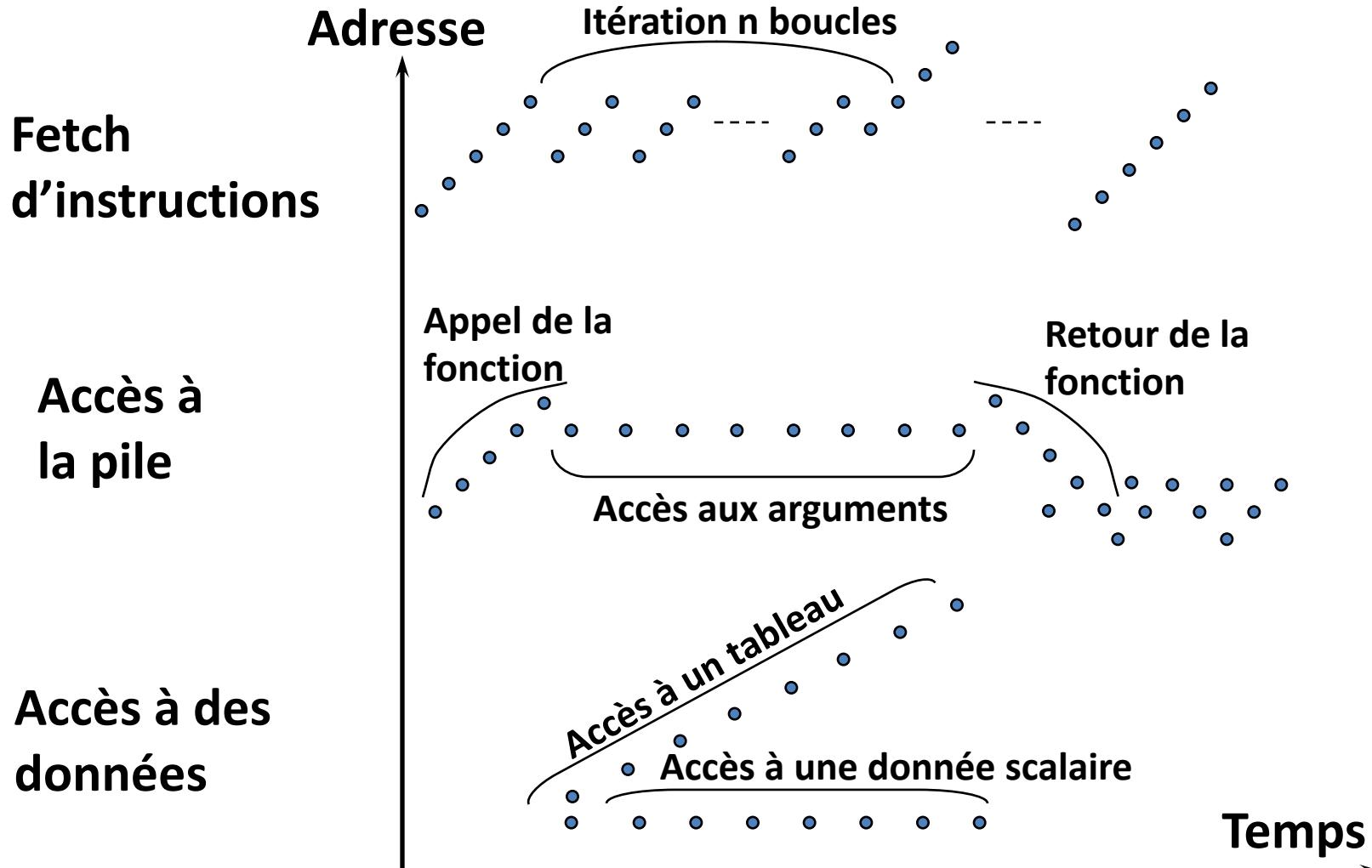
- Observation: les références aux données et surtout aux instructions ne sont pas, d'habitude, indépendantes. Les programmes ont tendance à réutiliser les données et les instructions qu'ils ont utilisées récemment.
- **Localité spatiale**: les éléments dont les adresses sont proches les unes des autres auront tendance à être référencés dans un temps rapproché (p.ex. instructions, images).
- **Localité temporelle**: les éléments auxquels on a eu accès récemment seront probablement accédés dans un futur proche (p.ex. boucles).

# Cartographie des accès mémoire



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory.  
IBM Systems Journal 10(3): 168-192 (1971)

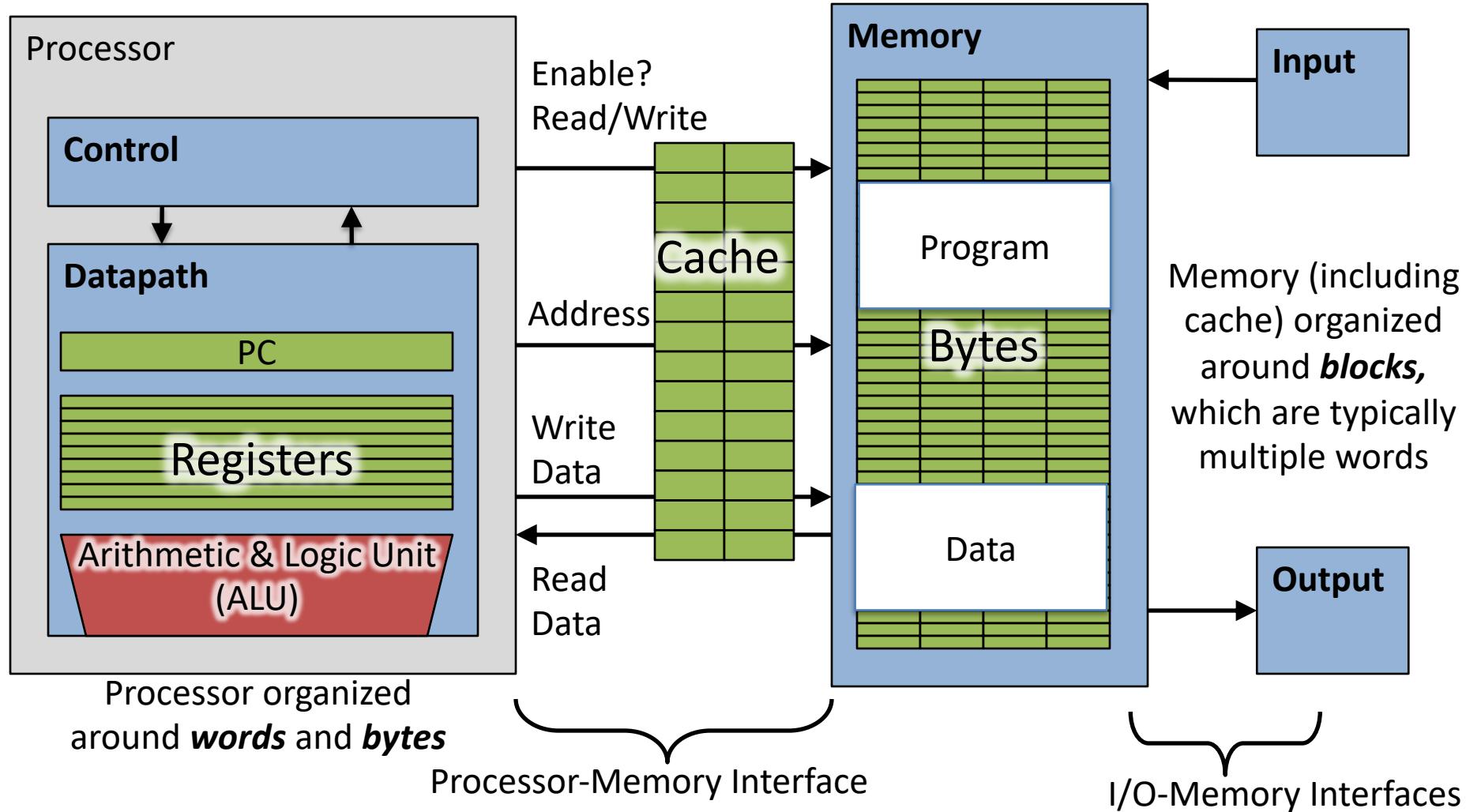
# Memory Reference Patterns



# Accès Mémoire sans cache

- Instruction Load word :  $lw \ t0, 0(t1)$
- $t1$  contient  $1022_{ten}$ ,  $\text{Mem}[1022] = 99$ 
  1. Le processeur génère l'adresse  $1022_{ten}$  vers la mémoire
  2. Le mot contenu en mémoire à l'adress $1022_{ten}$  est lu
  3. La mémoire (interface) envoie la valeur 99 vers le processeur
  4. Le processeur charge la valeur 99 dans le registre  $t0$

# Si on rajoute une mémoire cache...



# Questions

À cause de la localité spatiale, les données et les instructions sont transférées de la mémoire principale au cache en petits blocs de 2-8 mots mémoire. Plusieurs questions s'imposent:

- Où peut-on placer un bloc dans le cache?
  - Placement de bloc
- Comment trouver un bloc s'il est présent dans le cache?
  - Identification de bloc
- Quel bloc doit être remplacé en cas d'échec?
  - Remplacement de bloc
- Qu'arrive-t-il lors d'une écriture?
  - Stratégie d'écriture

# Accès Mémoire avec un cache

- Instruction Load word :  $lw\ t0, 0(t1)$
- $t1$  contient  $1022_{ten}$ ,  $\text{Mem}[1022] = 99$
- Cache: Le processeur génère la requête d'accès à l'adresse  $1022_{ten}$  à la mémoire cache
  1. Le contrôleur de cache vérifie si une copie de la donnée se trouvant en mémoire à l'adresse  $1022_{ten}$  est présente dans le cache
    - 2a. Si la donnée est présente (**Hit**): le cache renvoie la valeur 99 au processeur
    - 2b. Si non présente (**Miss**): la mémoire principale est accédée à l'adresse 1022 to
      - I. Le mot contenu en mémoire à l'adress  $1022_{ten}$  est lu
      - II. La valeur lue est envoyée vers le cache
      - III. Le cache enregistre cette nouvelle valeur
      - IV. Le cache envoie la valeur (99) au processeur
  2. Le processeur charge la valeur 99 dans le registre  $t0$

# Les Tags

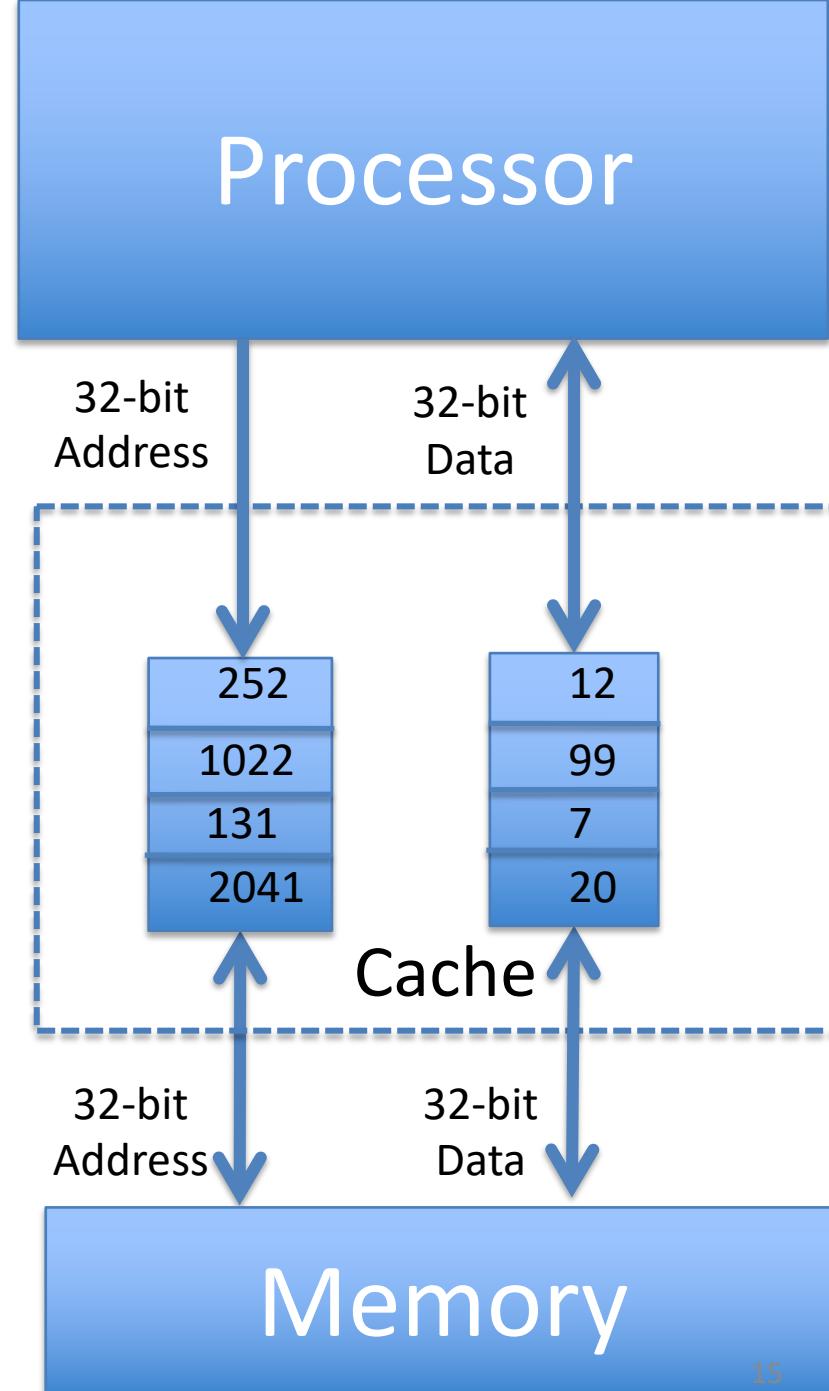
- Le contrôleur de cache a besoin d'un moyen de savoir si une donnée de la mémoire principale est présente dans le cache (hit ou miss)
- Lors d'un miss, on stocke l'adresse du bloc chargée dans la mémoire cache dans le contrôleur de cache
  - Ici 1022 est stocké dans le tag associé à la valeur 99

Tag	Data
252	12
1022	99
131	7
2041	20

Données en cache des précédents load/store

# Mémoire cache de 16 Octets avec des blocs de 4 octets

- 3 Opérations:
  1. Cache Hit
  2. Cache Miss
  3. Chargement du cache avec les données de la mémoire principale.
- Le cache a besoin des tags pour savoir s'il s'agit d'un hit ou d'un miss.
  - Les 4 tags sont comparés



# Remplacement

- Imaginons que le processeur accède à l'adresse 511 qui contient la donnée 11.
- Aucun tag ne correspond, on doit donc supprimer une ligne du cache pour charger ce nouveau bloc.
  - Quel bloc remplacer?

Tag	Data
252	12
1022	99
131	7
2041	20

# Remplacement

- Imaginons que le processeur accède à l'adresse 511 qui contient la donnée 11.
- Aucun tag ne correspond, on doit donc supprimer une ligne du cache pour charger ce nouveau bloc.
  - Quel bloc remplacer?
- On remplace la “victims” avec le nouveau bloc de l'adresse 511

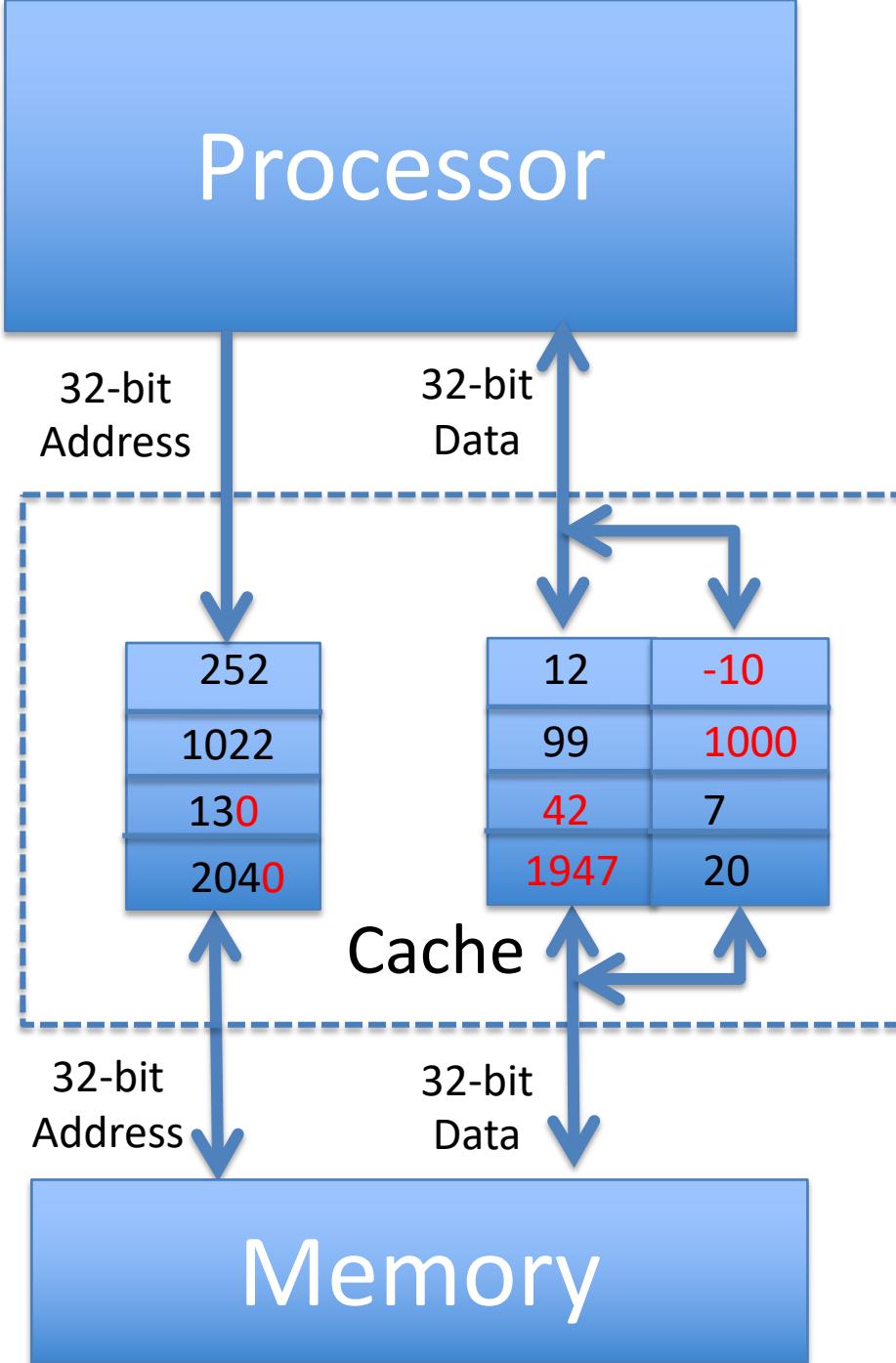
Tag	Data
252	12
1022	99
511	11
2041	20

# Gestion des Tags

- Les mots stockés en mémoire sont alignés donc l'adresse binaire d'un mot se termine toujours par  $00_{\text{two}}$
- Il n'est donc pas nécessaire de stocker les 2 derniers bits dans le tag
  - Cela simplifie le contrôleur et nécessite moins de capacité de stockage.

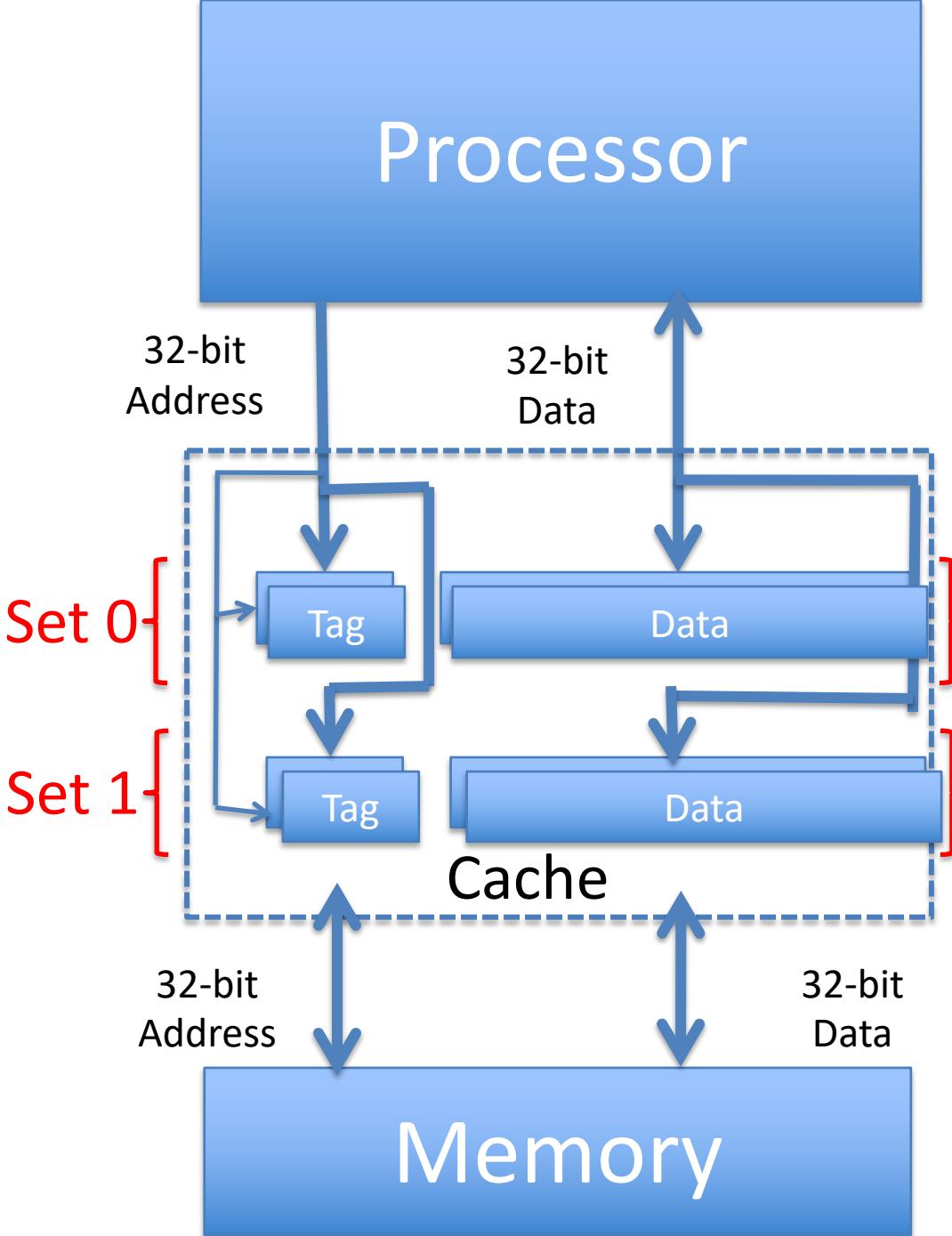
# Cache de 32 octets avec des blocs de 8 octets

- Les 3 derniers bits de l'adresse du bloc valent toujours  $000_{\text{two}}$
- On a un hoit si on accede un des deux mots du bloc

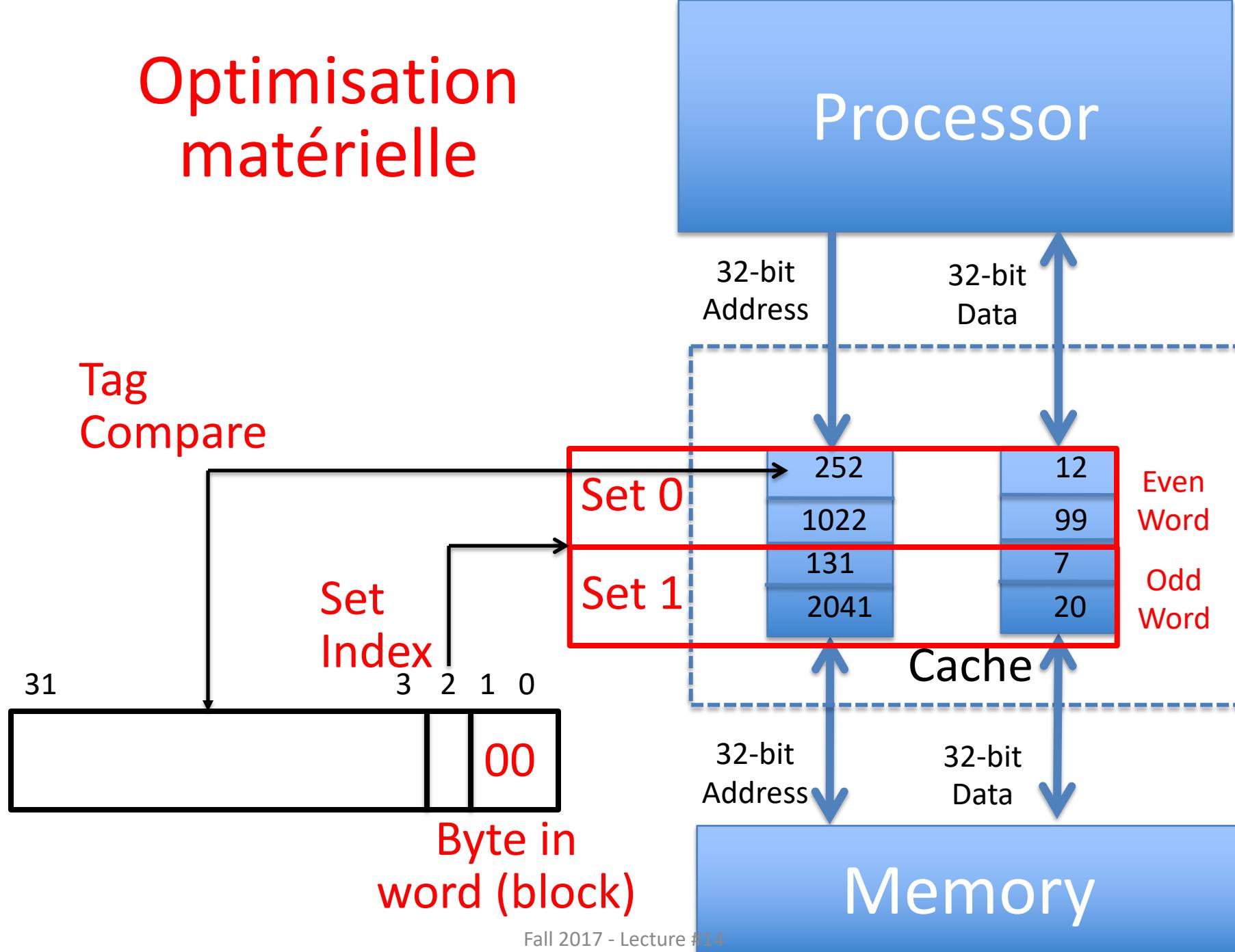


# Optimisation matérielle

- Il faut comparer tous les tags du cache avec l'adresse
- Les comparateurs ont un cout non négligeable.
- Une optimisation: utilisation de deux ensembles “sets” de données pour limiter les comparaisons.
- On utilise un bit de l'adresse pour sélectionner un ensemble.
- On compare l'adresse uniquement avec les tags de l'ensemble
- On peut généraliser cette approche à plus d'ensembles.

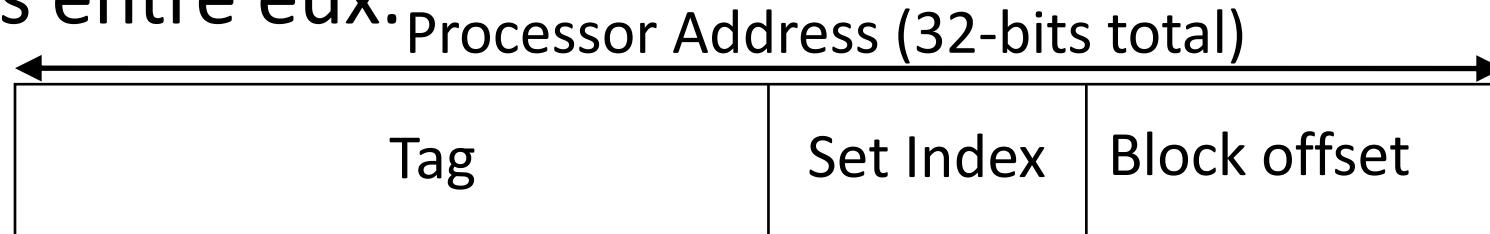


# Optimisation matérielle



# Quels champs de l'adresse utilise le contrôleur de cache?

- **Block Offset**: adresse de l'octet dans le bloc
- **Set Index**: indique quel ensemble (set)
- **Tag**: les bits restant de l'adresse qui permettent de différencier les blocs entre eux.



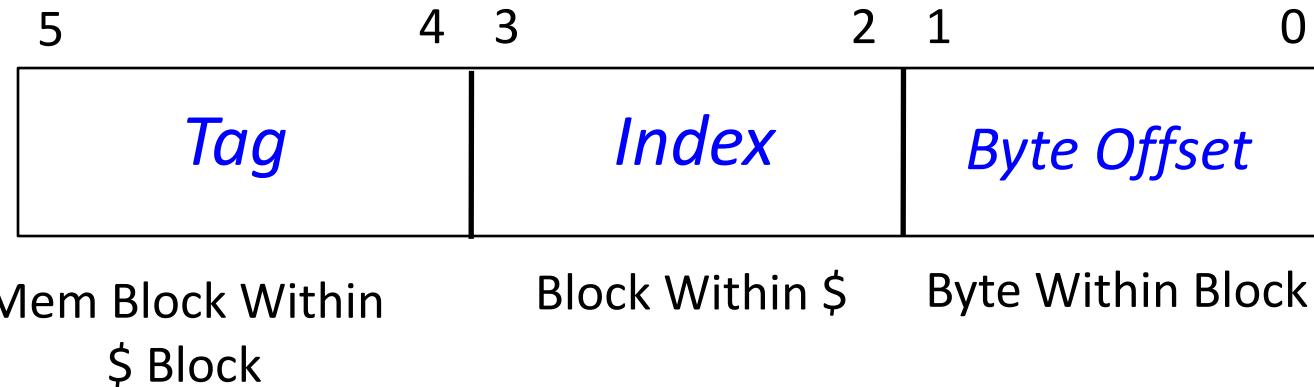
- Taille de l'index=  $\log_2(\text{number of sets})$
- Taille du tag= Address size – Size of Index  
–  $\log_2(\text{number of bytes/block})$

# Qu'est ce qui limite le nombre d'ensembles?

- Pour un nombre total de blocs, on réduit le nombre de comparaisons si on a plus de deux ensembles.
- Une limite: Si on a autant d'ensemble que de bloc=> seulement un bloc par ensemble, une seule comparaison à effectuer!
- C'est ce qu'on appelle les caches à correspondance directe

Tag	Index	Block offset
-----	-------	--------------

# Cache à correspondance directe: Mémoire avec un bus d'adresse de 6 bits



- Taille des blocs 4 octets
- Les blocs de la mémoire cache et de la mémoire principale ont toujours la même taille
- # blocs en mémoire >> # blocs en cache
  - 16 blocs en mémoire = 16 words = 64 bytes => 6 bits pour addresser tous les bytes
  - 4 blocs en cache, 4 bytes (1 word) par bloc
  - 4 blocs de la mémoire peuvent être contenu dans la cache
- Dans quel ensemble peut se trouver un bloc de la mémoire en cache: c'est le rôle de l'index
- Quel bloc de la mémoire principale est dans le cache : information donnée par le tag

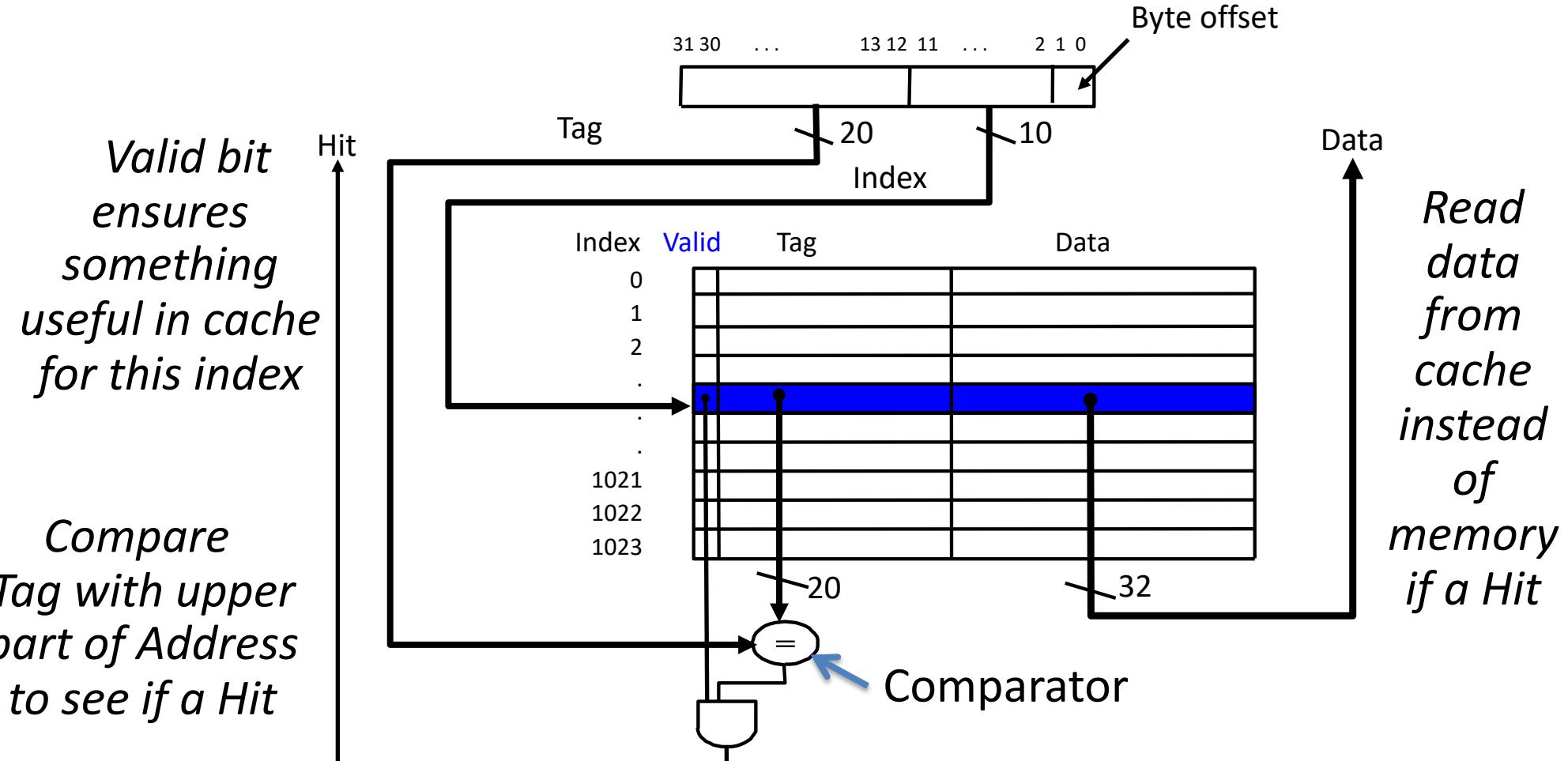
# Bit de Validité

- Lorsqu'on démarre un nouveau programme, le cache ne contient pas des informations valides pour ce programme.
- On a besoin d'indiquer cette information
- C'est le rôle du bit de validité
  - 0 => cache miss, même si on a le bon tag
  - 1 => cache hit, si on le bon tag

# Les différentes architectures de mémoire cache

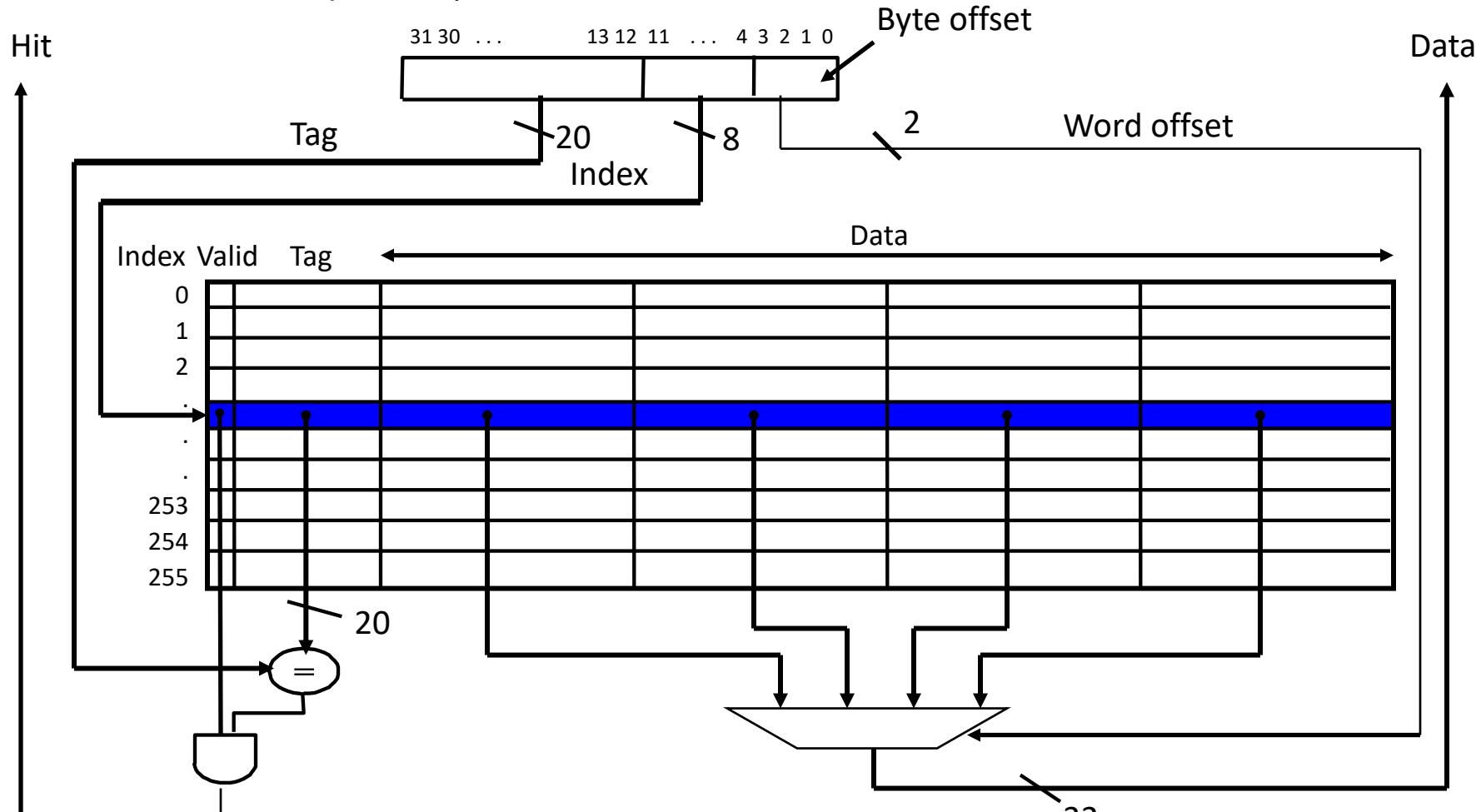
# Mémoire Cache à correspondance directe

- One word blocks, cache size = 1K words (or 4KB)



# Multiword-Block Direct-Mapped Cache

- Four words/block, cache size = 1K words



*What kind of locality are we taking advantage of?*

# Les différentes structures de mémoire cache

- Totalement associatif “**Fully Associative**”: un bloc peut être placé n’importe où dans la mémoire cache
  - Il n’y a donc pas d’index



- Cache à correspondance directe “**Direct Mapped**”: un bloc ne peut aller que dans un emplacement du cache (en fonction de l’index)
  - Le nombre d’ensemble dans la mémoire cache est le nombre de blocs possible en cache



Le nombre de bits pour coder l’index est  $\log_2(\text{nombre de blocs en caches})$

- Associatif par ensemble de N “**N-way Set Associative**”: Il y a N emplacements disponibles pour un bloc en mémoire cache.
  - Le nombre d’ensemble en mémoire cache est : nombre de blocs en cache/N

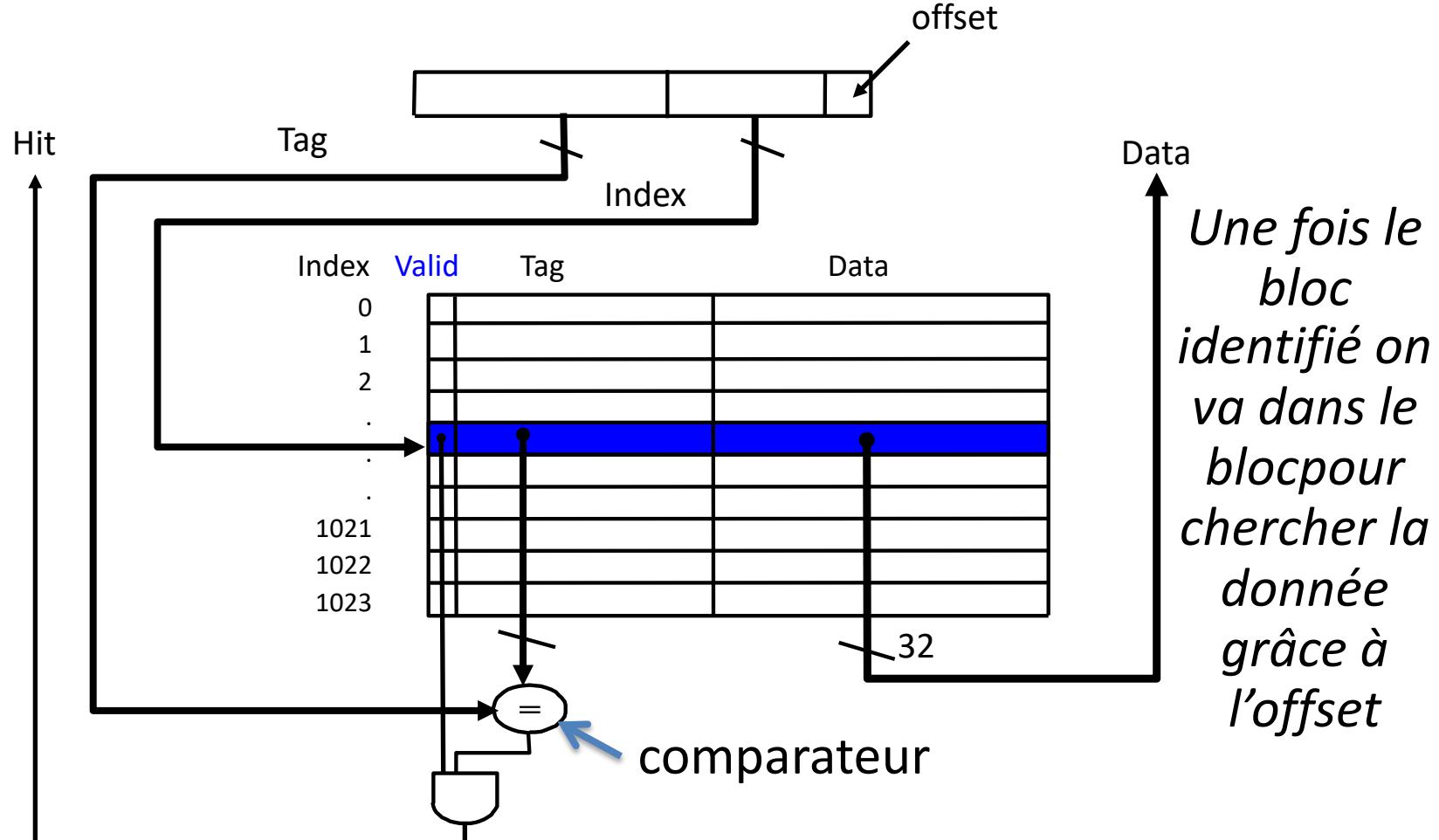


– Le nombre de bits pour coder l’index est  $\log_2(\text{nombre de blocs en caches}/N)$

# Cache à correspondance directe

*Bit de validité  
indique si le bloc du  
cache contient des  
données valides*

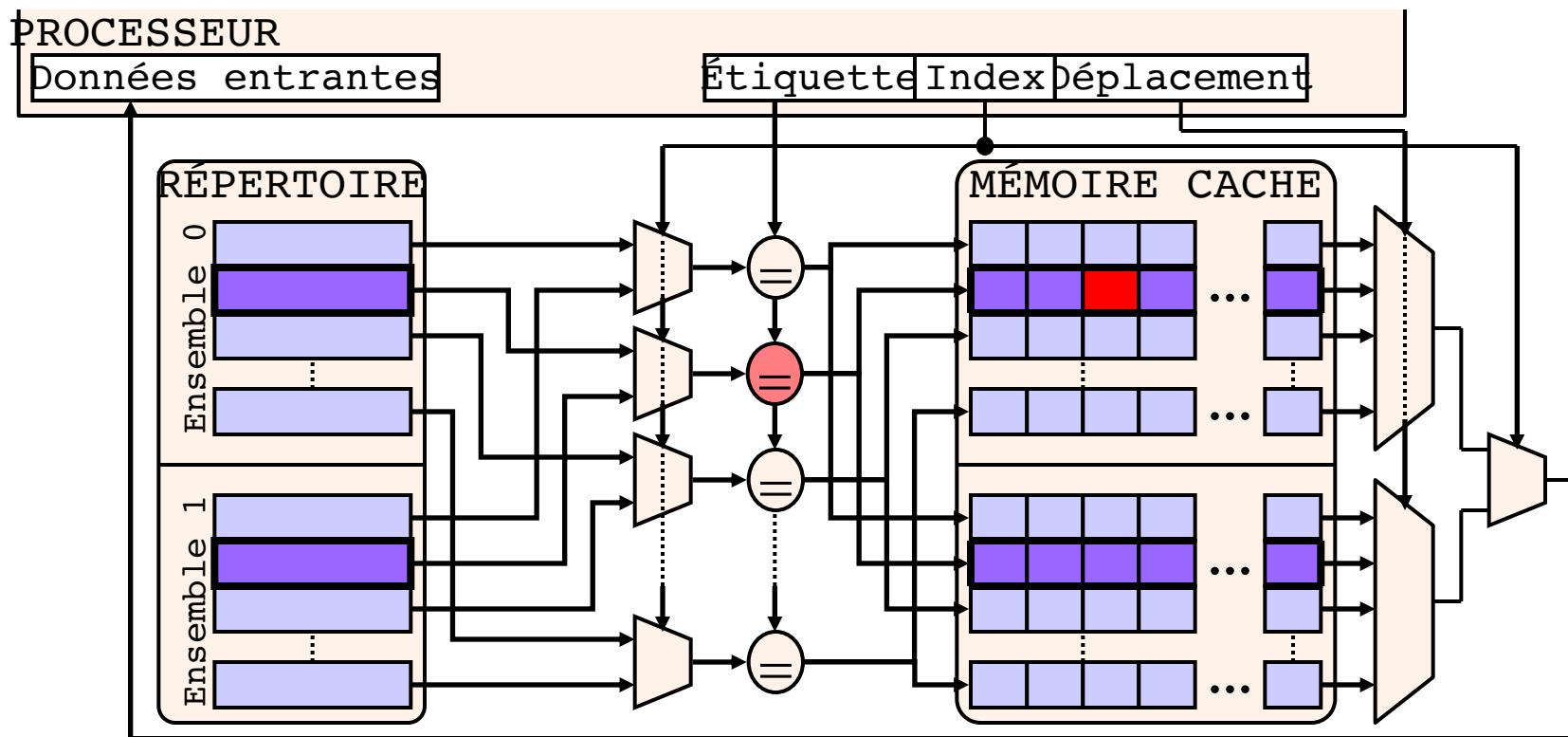
*On compare le  
tag pour savoir  
si le bloc en  
cache  
correspond à  
celui qu'on  
recherche*



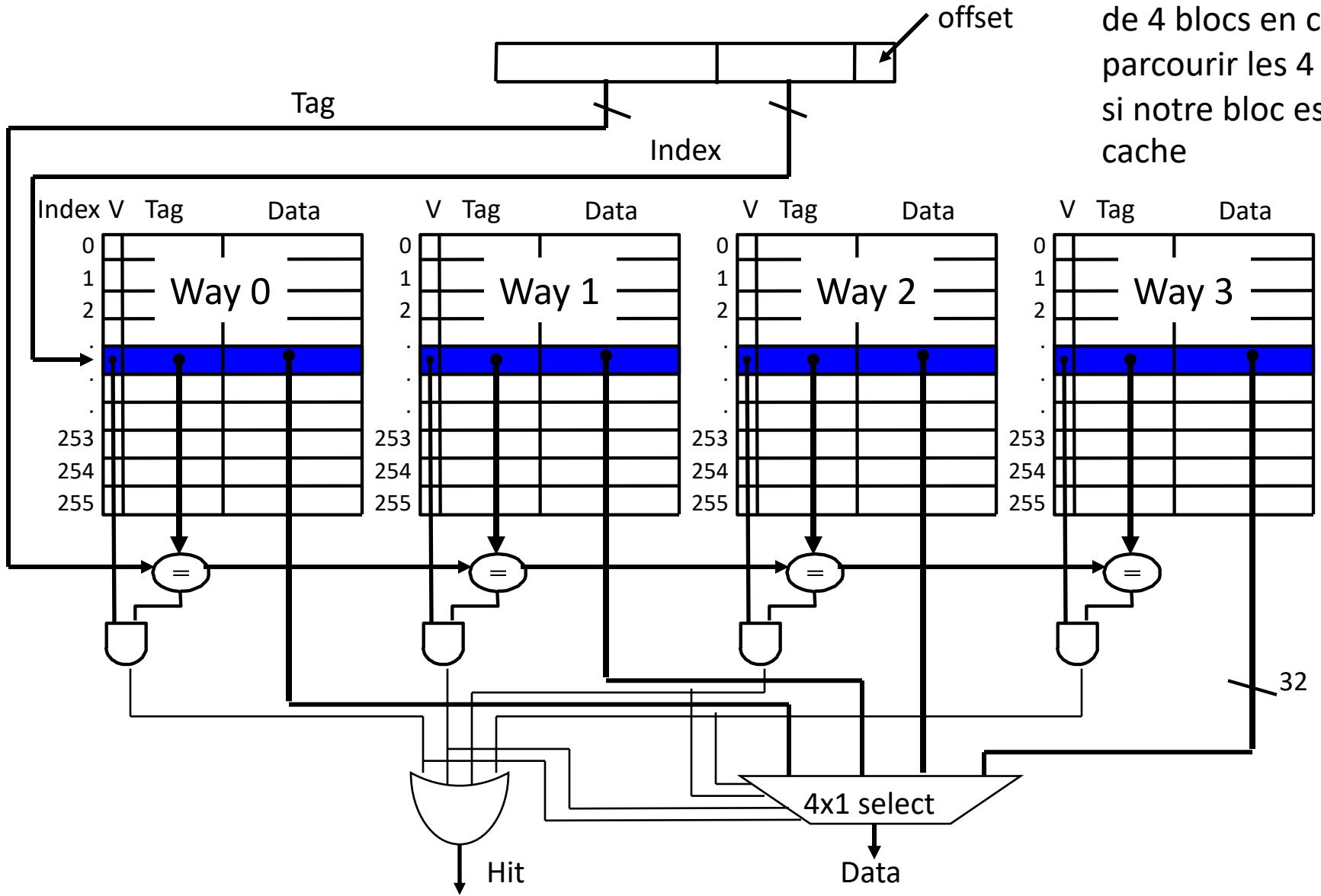
*Une fois le  
bloc  
identifié on  
va dans le  
bloc pour  
chercher la  
donnée  
grâce à  
l'offset*

# Identification de bloc

- Pour un cache associatif par ensemble, une adresse cherchée par le processeur peut être partagée en 3 champs:  
le **déplacement (offset)** est l'adresse dans le bloc du mot cherché, l'**index** identifie l'ensemble, et l'**étiquette (tag)** est la partie de l'adresse du bloc utilisée pour la comparaison.



# Cache associative par assemblage de 4



Un index identifie un ensemble de 4 blocs en cache, il faut donc parcourir les 4 tags pour savoir si notre bloc est présent en cache

# Cache totalement associatif

Alem. Principale

## Mem. Cache Totalement associative

dimanche 20 septembre 2020 20:48

Un bloc de la mémoire centrale peut être stocké n'importe où dans la mémoire cache



Le contrôleur de cache calcule le tag à partir de l'adresse et parcourt le répertoire de la mémoire cache pour voir si celui-ci est présent dans la cache.

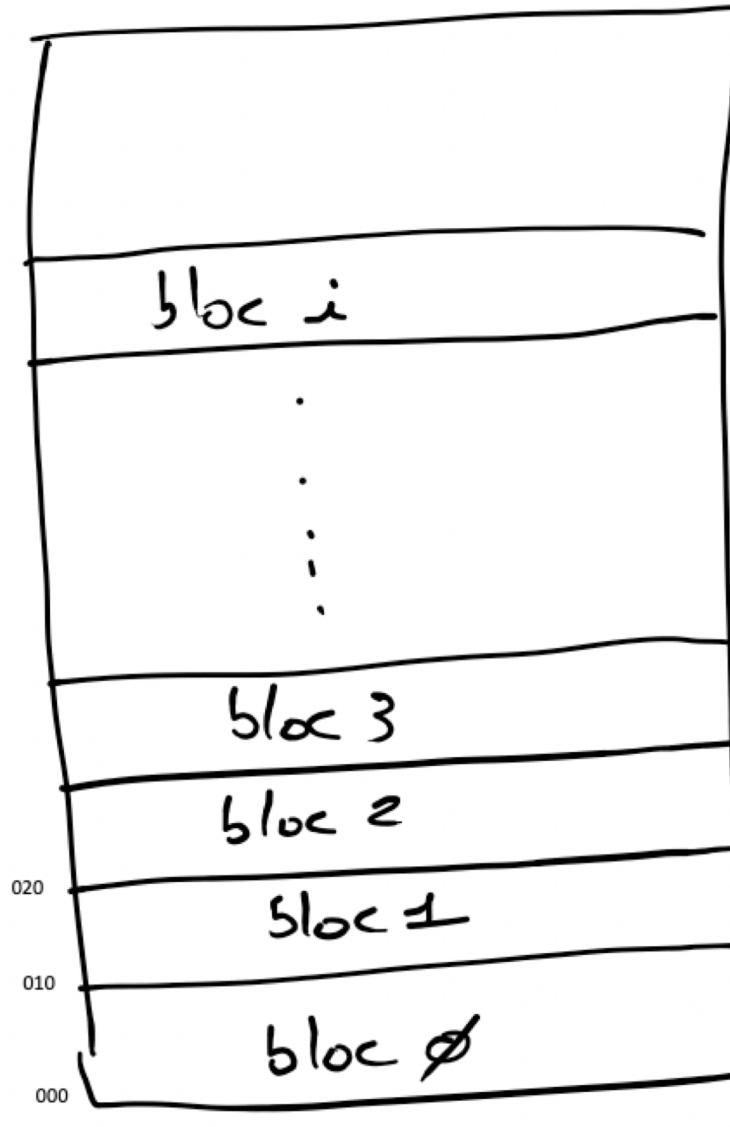
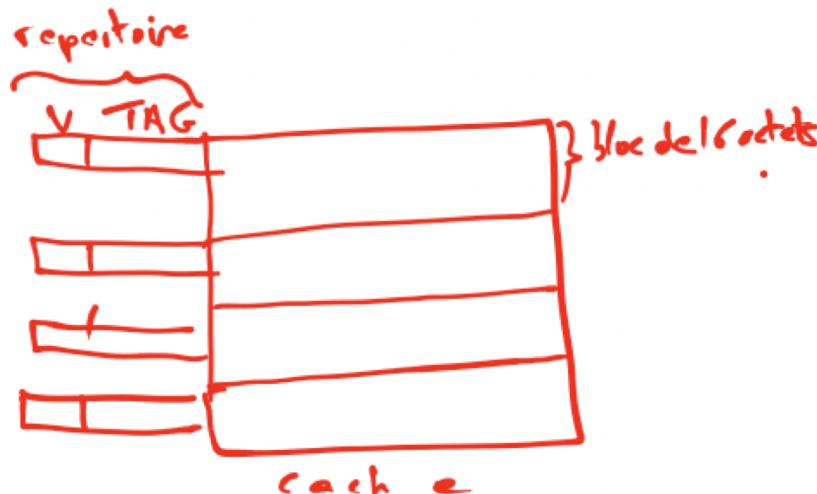
Dans l'exemple on a des blocs de 16 octets

L'offset est donc codé sur 4 bits et le reste de l'adresse est le tag (ce qui correspond aussi au numéro de bloc en mémoire centrale)

Si le processeur demande l'adresse 014 (en hexa)

Alors le tag vaut 01 et l'offset vaut 4

On va parcourir le répertoire pour voir si le tag (01) est présent, si oui (et que le bit v=1) alors on ira dans le bloc de la mémoire cache à l'offset 4 pour récupérer la donnée



# Remplacement de bloc

Quand un échec intervient, le contrôleur de cache doit choisir un bloc à remplacer par la donnée désirée.

Pour les caches à correspondance directe, ce choix est obligatoire. Pour les caches à associativité totale ou par ensemble, trois algorithmes sont courants:

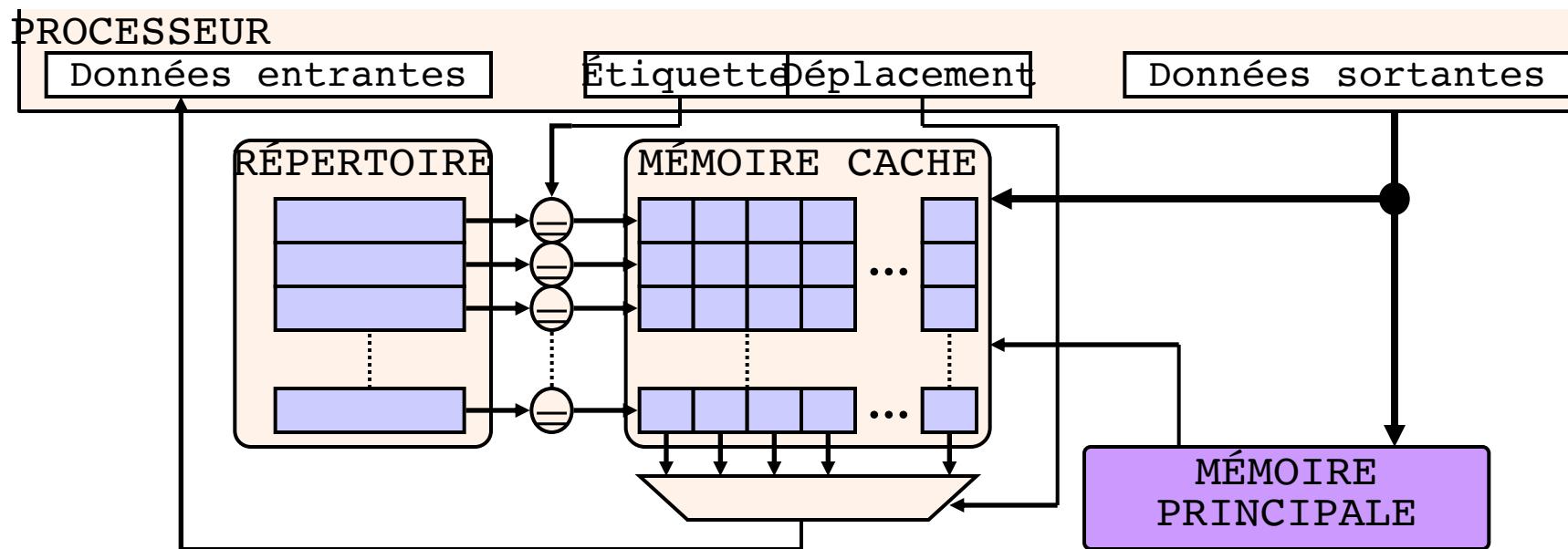
- Le hasard.
- Pas le dernier (NLU pour *not last used*).
- Le plus ancien (LRU pour *least recently used*).

Le LRU est la stratégie la plus efficace, mais aussi la plus coûteuse à réaliser.

# Stratégie d'écriture I

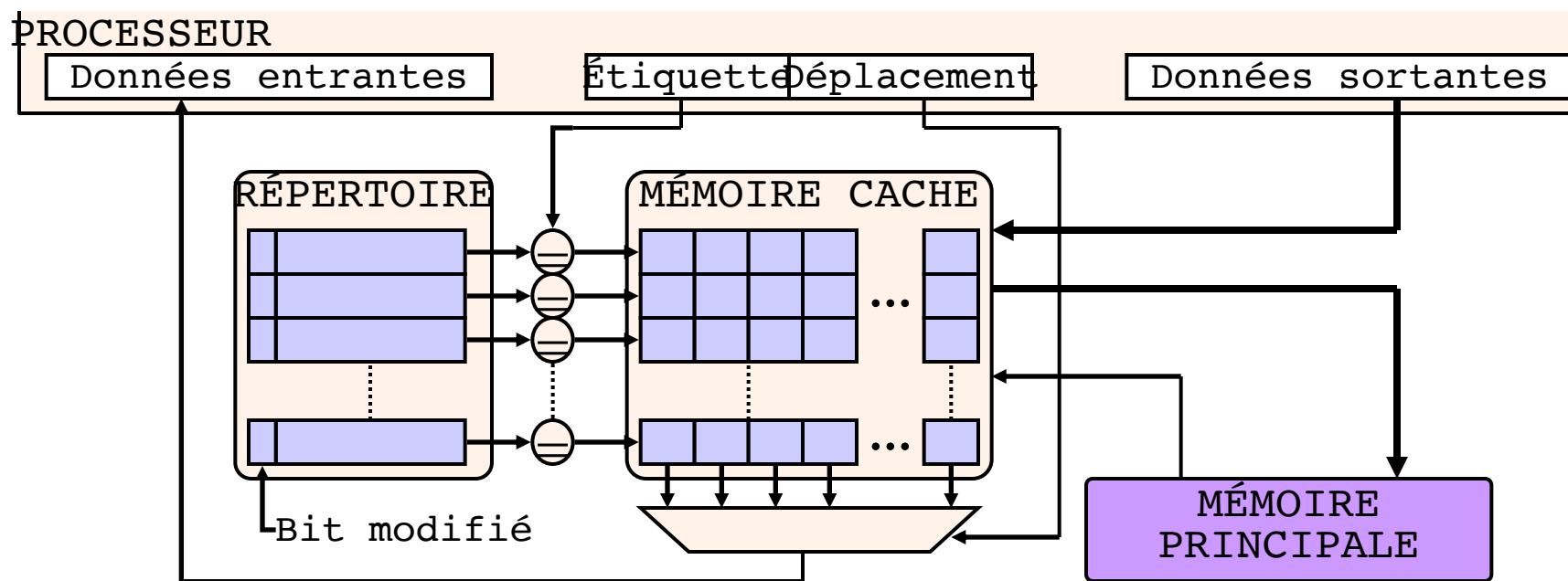
La stratégie d'écriture est fondamentale pour la performance du **cache de données** (environ 25% des accès à ce cache sont des écritures). Deux stratégies sont couramment employées:

- L'**écriture simultanée** ou **rangement simultané (write-through)** consiste à écrire simultanément dans le bloc de cache et dans la mémoire principale.



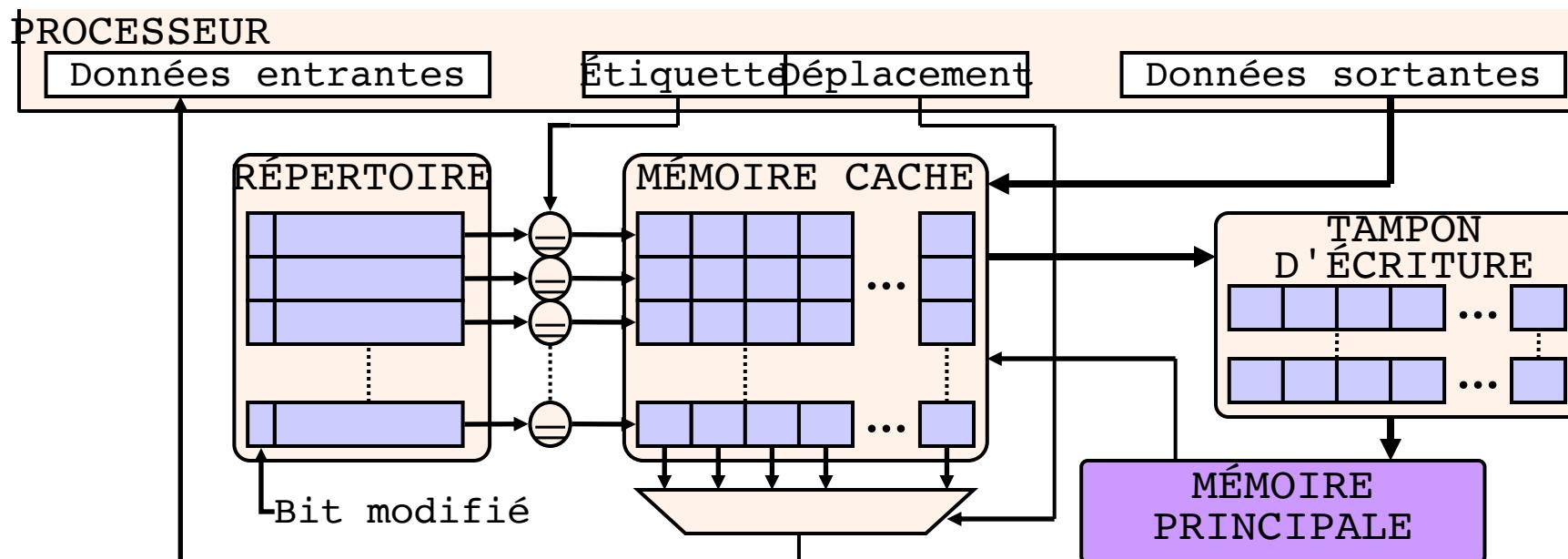
# Stratégie d'écriture II

- La **réécriture ou recopie ou rangement (write-back ou copy-back)** consiste à écrire uniquement dans le bloc du cache. Le bloc modifié du cache est recopié en mémoire principale uniquement quand il est remplacé. Un **bit modifié (modified bit)** dans le répertoire du cache indique si le bloc a été modifié.



# Stratégie d'écriture III

La recherche de données en mémoire principale lors d'un échec de lecture dans le cache est **prioritaire** par rapport à l'écriture. Cette dernière peut donc être **suspendue** en utilisant un **tampon d'écriture (write buffer)**.



Cependant, les tampons d'écriture compliquent les choses car ils peuvent contenir des **valeurs modifiées** d'un bloc qui fait l'objet d'un échec en lecture.

# Amélioration de la performance

- Le processeur cherche un mot d'abord dans le cache. En cas d'**échec** (*cache miss*), le mot est cherché en mémoire et **son bloc est copié dans le cache pour un accès futur**. En cas de **succès** (*cache hit*), la mémoire principale **n'est pas accédée**.
- Temps d'exécution = Temps de cycle  $\times$  (# cycles d'exécution + # cycles d'attente mémoire)
- # cycles d'attente mémoire = # accès mémoire  $\times$  (Pénalité d'accès + Taux d'échec  $\times$  Pénalité d'échec)
- Pour améliorer cette performance, il y a trois moyens:
  - Réduire le taux d'échec
  - Réduire la pénalité d'échec
  - Réduire le temps de l'accès réussi au cache

# Réduire le taux d'échec

- Échec obligatoire (ou de première référence): lors de son premier accès, un bloc n'est sûrement pas dans le cache.
- Échec de capacité: si le cache ne peut contenir tous les blocs nécessaires pendant l'exécution d'un programme, des blocs seront écartés puis rappelés plus tard.
- Échec de conflit (ou de collision ou d'interférence): dans des caches associatifs par ensemble ou à correspondance directe, un bloc peut être écarté et rappelé plus tard si trop de blocs doivent être placés dans son ensemble.

# Réduire le taux d'échec I

Une solution: augmenter la taille des blocs pour mieux exploiter la localité spatiale des références. En même temps, des blocs plus gros peuvent augmenter les échecs de conflit et même les échecs de capacité si le cache est petit. Plusieurs paramètres architecturaux définissent la taille optimale des blocs.

Une associativité plus élevée peut améliorer les taux d'échec. Des lois empiriques sont généralement appliquées. Par exemple, l'associativité à 8 blocs est aussi efficace que l'associativité totale pour les tailles de cache plus courantes. Malheureusement, une associativité plus élevée ne peut être obtenue qu'au prix d'un temps d'accès réussi plus élevé (dû à une logique de contrôle plus compliquée).

## Réduire le taux d'échec II

Une manière d'améliorer les taux d'échec sans affecter la fréquence d'horloge du processeur est de lire à l'avance les éléments avant qu'ils soient demandés par le processeur. Données et instructions peuvent être lues à l'avance, soit directement dans le cache, soit dans un tampon externe.

Cette **lecture anticipée** est fréquemment faite **par matériel**. Par exemple, le processeur DEC Alpha AXP 21064 lit deux blocs lors d'un échec en lecture des instructions: le bloc demandé et le suivant. Ce dernier est placé dans un **tampon de flux d'instructions**.

Un tampon capable de stocker un seul bloc d'instructions peut éviter de 15% à 25% des échecs.

# Réduire le taux d'échec III

Une alternative à la lecture anticipée matérielle est que le **compilateur** introduise des **instructions de préchargement** pour aller chercher la donnée **avant** qu'elle ne soit nécessaire. Il y a plusieurs types de lecture anticipée:

- La **lecture anticipée des registres**, qui charge la valeur dans un registre.
- La **lecture anticipée du cache**, qui charge la valeur uniquement dans le cache et pas dans les registres.

Cette prélecture n'a de sens que si le processeur peut travailler pendant que la donnée lue à l'avance est en cours de lecture. Ceci suppose que le cache peut continuer à fournir des informations au processeur pendant qu'il attend l'arrivée de la donnée anticipée (**cache non bloquant**).

# Réduire le taux d'échec IV

Le compilateur peut réduire les taux d'échec du cache en effectuant des **optimisations sur le code**.

Un exemple parmi les plus courants:

- **Échange de boucles**: certains programmes ont des boucles imbriquées qui accèdent aux données de manière non-séquentielle. Le simple échange de l'ordre d'imbrication des boucles peut faire correspondre l'ordre d'accès des données à leur ordre de rangement (localité spatiale).

# Réduire la pénalité d'échec I

Cette solution, qui ne nécessite pas de matériel supplémentaire, est basée sur l'observation que le CPU a juste besoin d'un mot d'un bloc à la fois.

- Le **redémarrage précoce**: aussitôt que le mot demandé du bloc arrive, il est envoyé au CPU qui redémarre.
- Le **mot critique en premier**: le mot demandé est cherché en mémoire et envoyé au CPU d'abord, ensuite le restant du bloc est copié dans la cache pendant que la CPU continue l'exécution; cette solution est aussi appelée **lecture enroulée**. Généralement, ces techniques ne sont profitables que pour de très grosses tailles de blocs.

# Réduire la pénalité d'échec II

Est-ce qu'il faut accélérer le cache pour suivre la vitesse croissante du CPU ou agrandir le cache pour suivre la taille croissante de la mémoire principale? Les deux!

En ajoutant un **deuxième niveau de cache** entre le cache originel et la mémoire, le premier niveau peut être assez petit pour correspondre au temps de cycle du CPU, alors que le second niveau peut être assez grand pour capturer beaucoup d'accès qui iraient à la mémoire principale, en réduisant ainsi la pénalité d'échec effective.

Les **paramètres** du cache de second niveau sont donc différents de ceux du cache de premier niveau: il peut être **plus lent** mais aussi **plus grand**, permettant un **plus haut degré d'associativité** ainsi qu'une **taille de blocs plus grande**.

# Réduire le temps d'accès I

Une partie importante du temps d'accès réussi au cache est le temps nécessaire pour comparer le contenu du répertoire de cache à l'adresse, fonction de la taille du cache.

Un dispositif matériel plus petit est plus rapide, et un **cache petit** améliore certainement le temps d'accès réussi (il est aussi critique de garder un cache suffisamment petit pour qu'il puisse tenir dans le même circuit que le processeur).

Pour la même raison, il est important de garder un **cache simple**, p.ex. en utilisant la correspondance directe.

Ainsi, la pression pour un **cycle d'horloge rapide** encourage la réalisation de caches petits et simples pour les **caches primaires**.

## Réduire la temps d'accès II

Même un cache petit et simple doit prendre en compte la traduction d'adresse virtuelle venant du CPU en adresse physique pour accéder à la mémoire principale.

Une solution est d'**éviter la traduction d'adresse durant l'indexation du cache (cache virtuel)**, puisque les accès réussis sont beaucoup plus courants que les échecs.

Les caches virtuels sont plus rapides par rapport aux **caches physiques**, mais introduisent de nombreuses complications (p.ex., plusieurs adresses virtuelles peuvent référencer la même adresse physique, ce qui peut conduire à plusieurs copies de la même donnée dans un cache virtuel et causer des erreurs lorsqu'une de ces copies est modifiée).