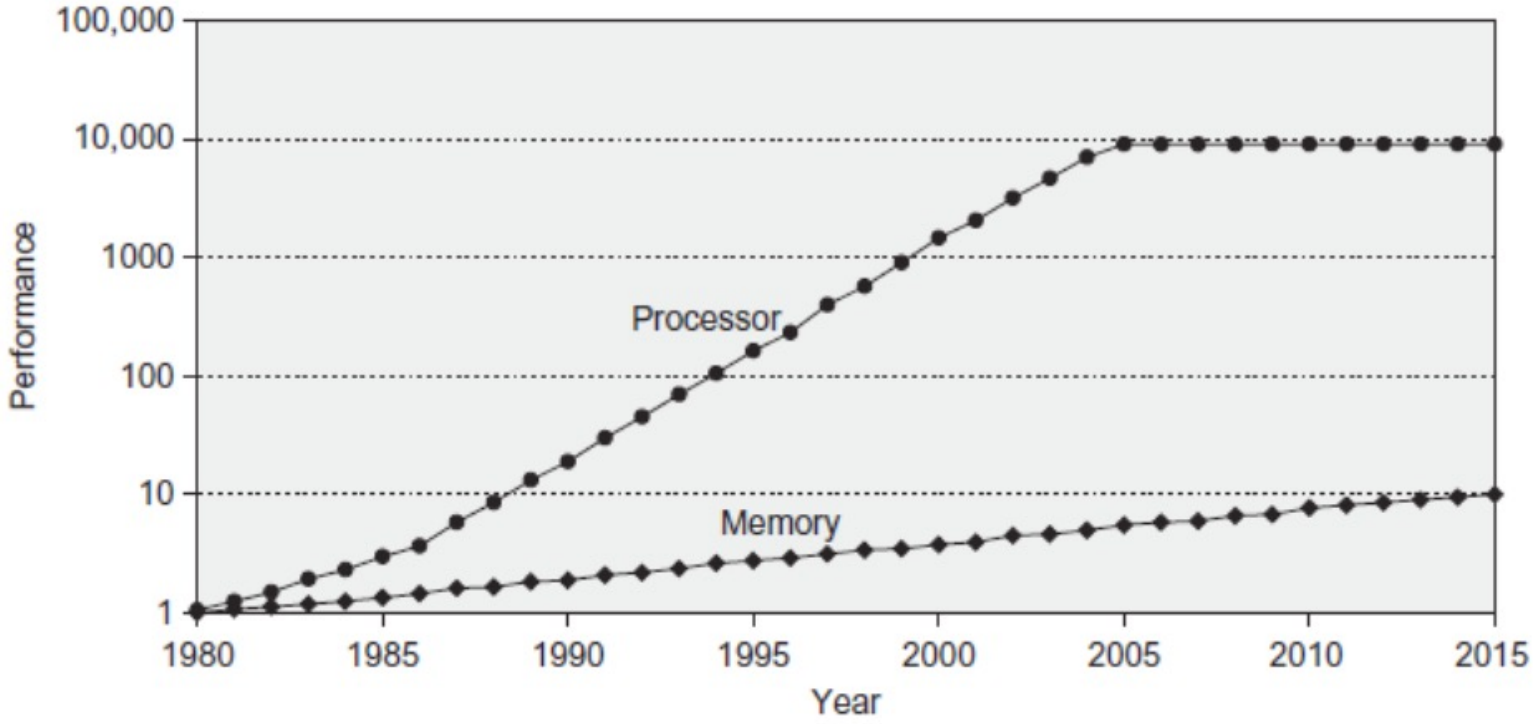
La mémoire cache

Raymond Namyst
Pierre-André Wacrenier

Pourquoi a-t-on besoin des caches ?

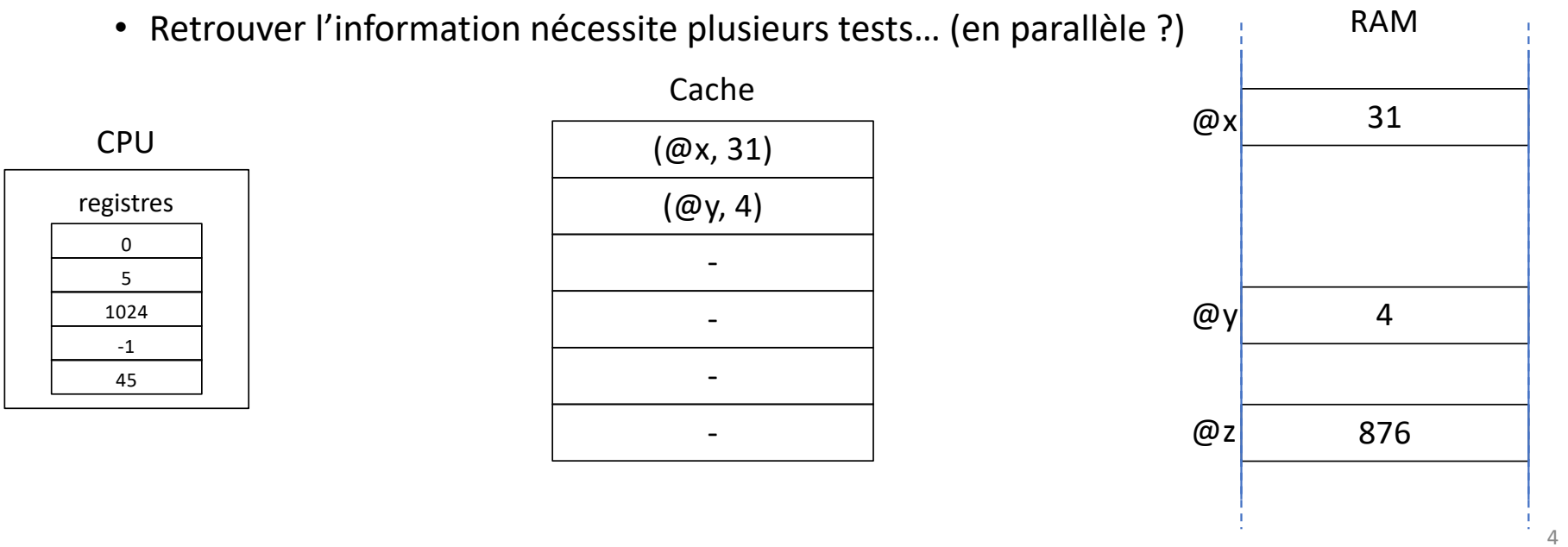


De l’utilité d’une mémoire de proximité

- Comment obtenir les performances promises par les processeurs ?
  - Plus une mémoire est rapide plus elle est chère
  - Plus une mémoire est petite plus son temps de réponse peut être faible
- Localité temporelle
  - Beaucoup d’accès répétés à des mêmes variables, de façon proche dans le temps
    - Boucles
- Localité spatiale
  - Concentration des accès sur des données proches
    - Structures, tableaux, variables locales
- Règle 90/10 « 90% de l'exécution se déroule dans 10% du code »

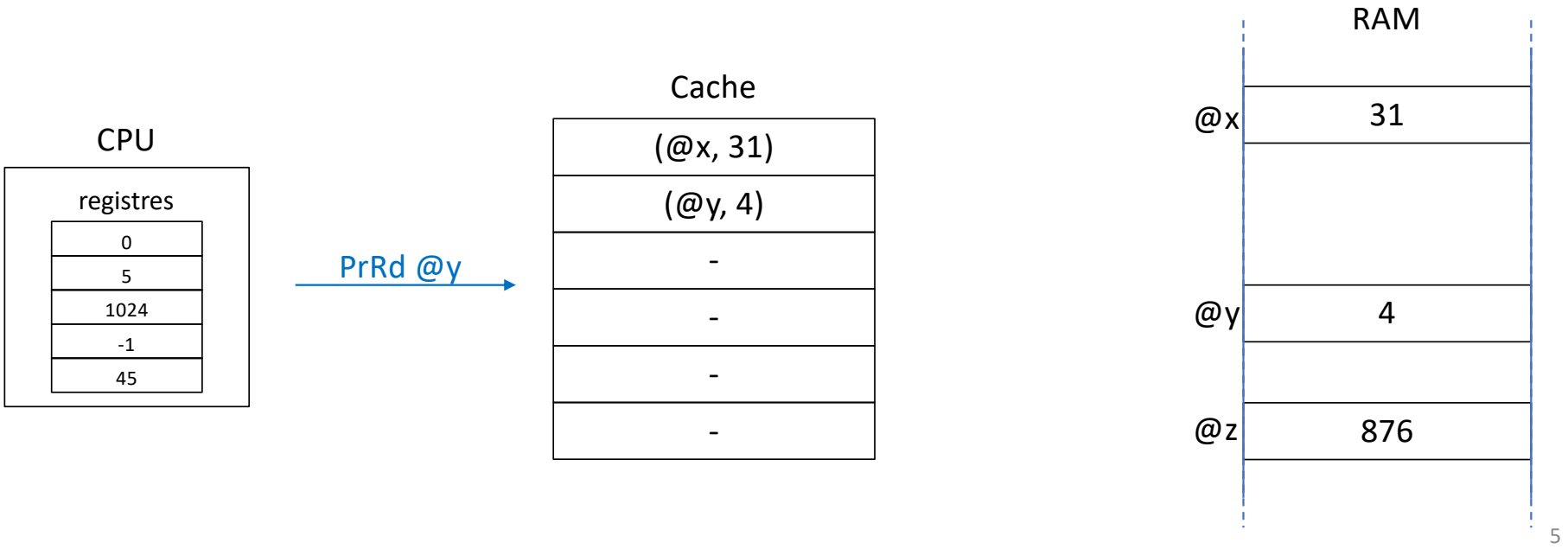
En première approximation…

- Le cache conserve les données les plus « utiles »
  - On peut imaginer qu'il stocke des couples (adresse, contenu)
  - Son fonctionnement est différent de la RAM
    - Retrouver l'information nécessite plusieurs tests… (en parallèle ?)



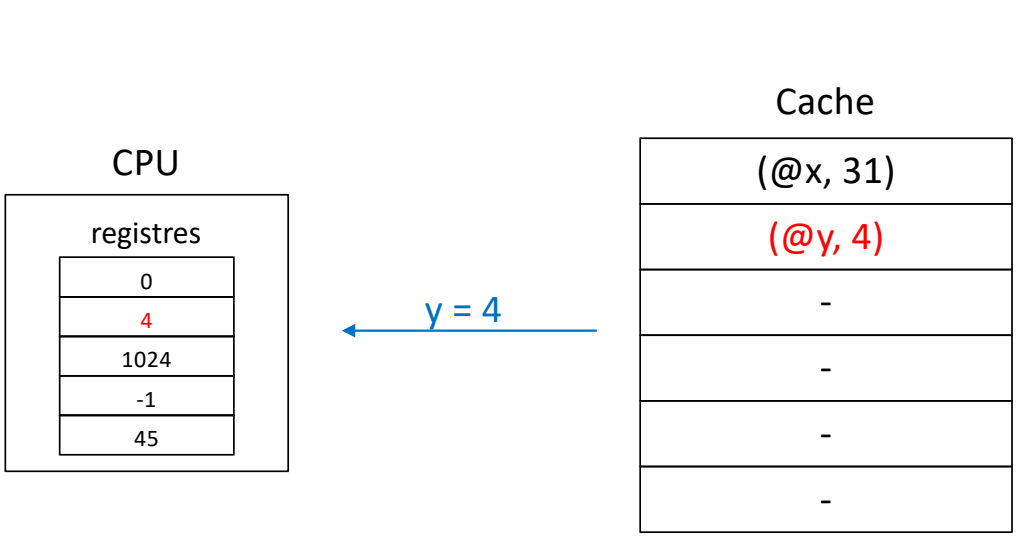
En première approximation…

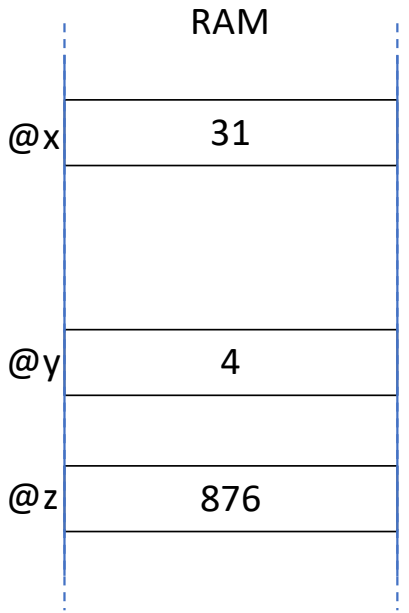
- Lorsque le CPU émet une requête de lecture
  - Le cache l'intercepte et répond s'il possède la donnée



En première approximation…

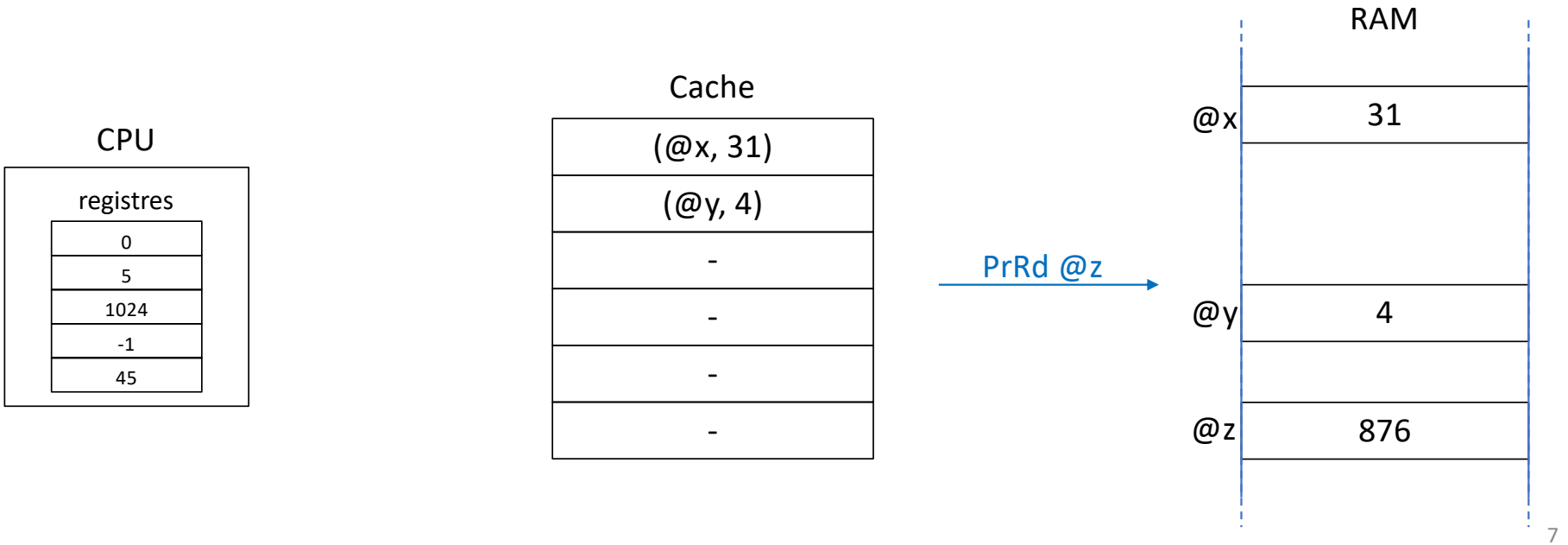
- Lorsque le CPU émet une requête de lecture
  - Le cache l'intercepte et répond s'il possède la donnée
  - C'est un *cache hit*





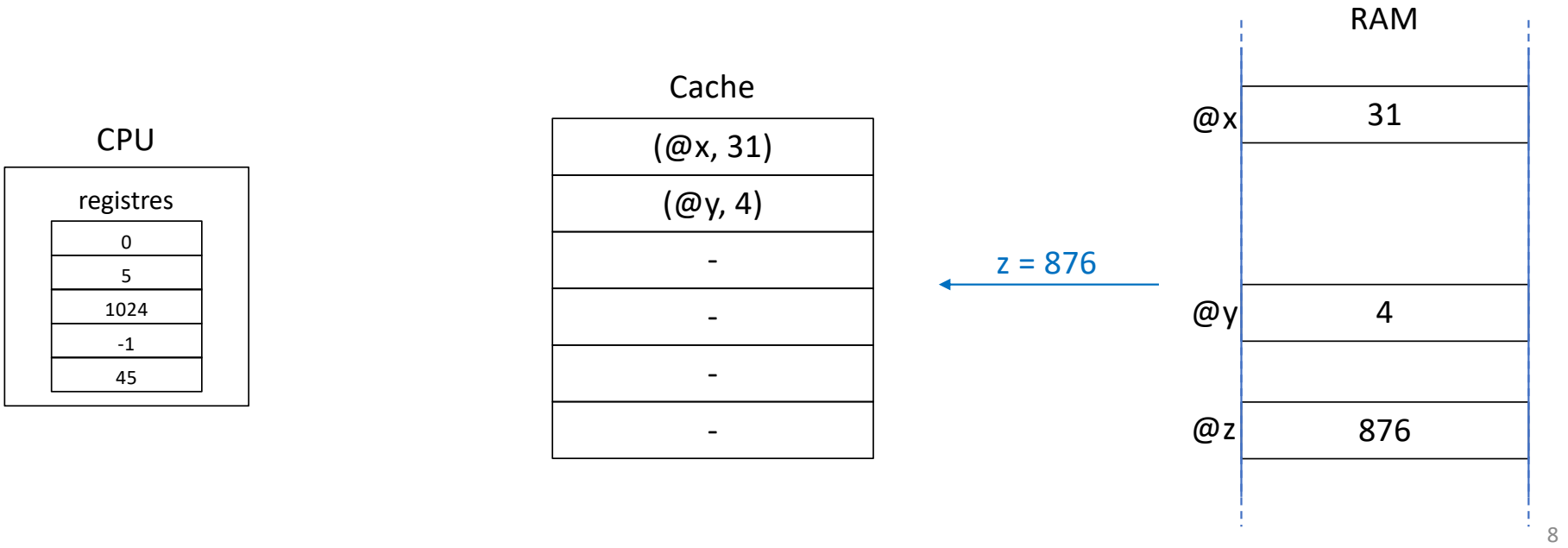
En première approximation…

- Lorsque le CPU émet une requête de lecture
  - La requête est acheminée vers la RAM si le cache ne possède pas la donnée
  - C'est un *cache miss*



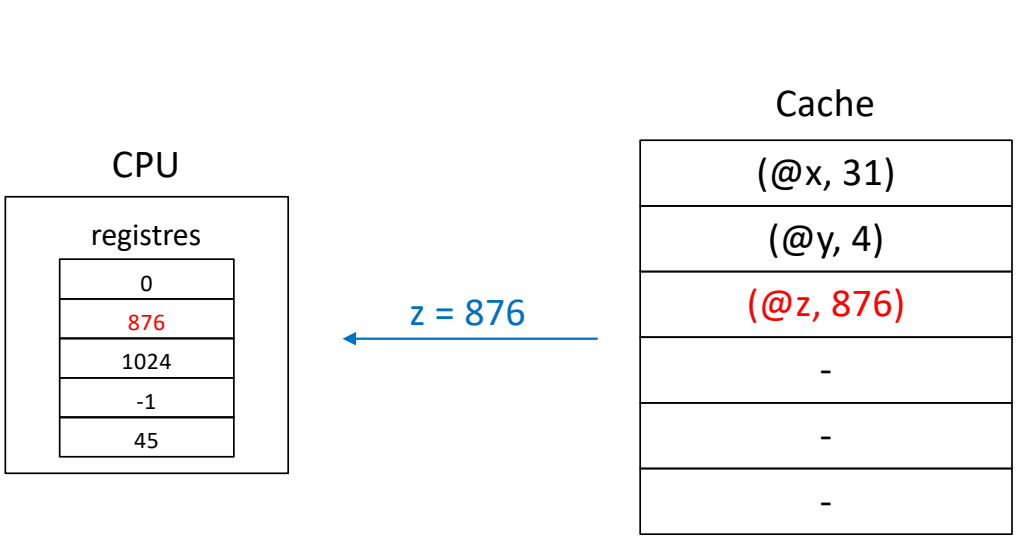
En première approximation…

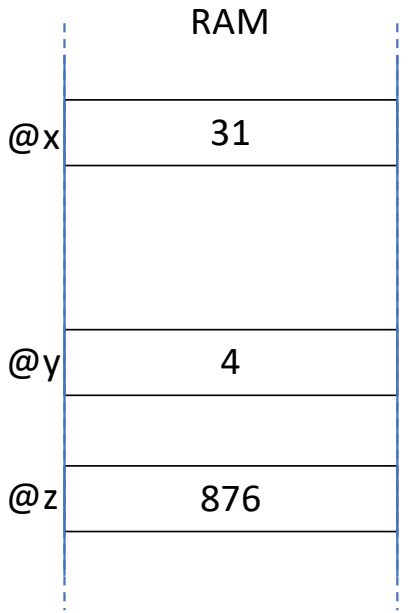
- Lorsque le CPU émet une requête de lecture
  - La requête est acheminée vers la RAM si le cache ne possède pas la donnée
  - C'est un *cache miss*



En première approximation…

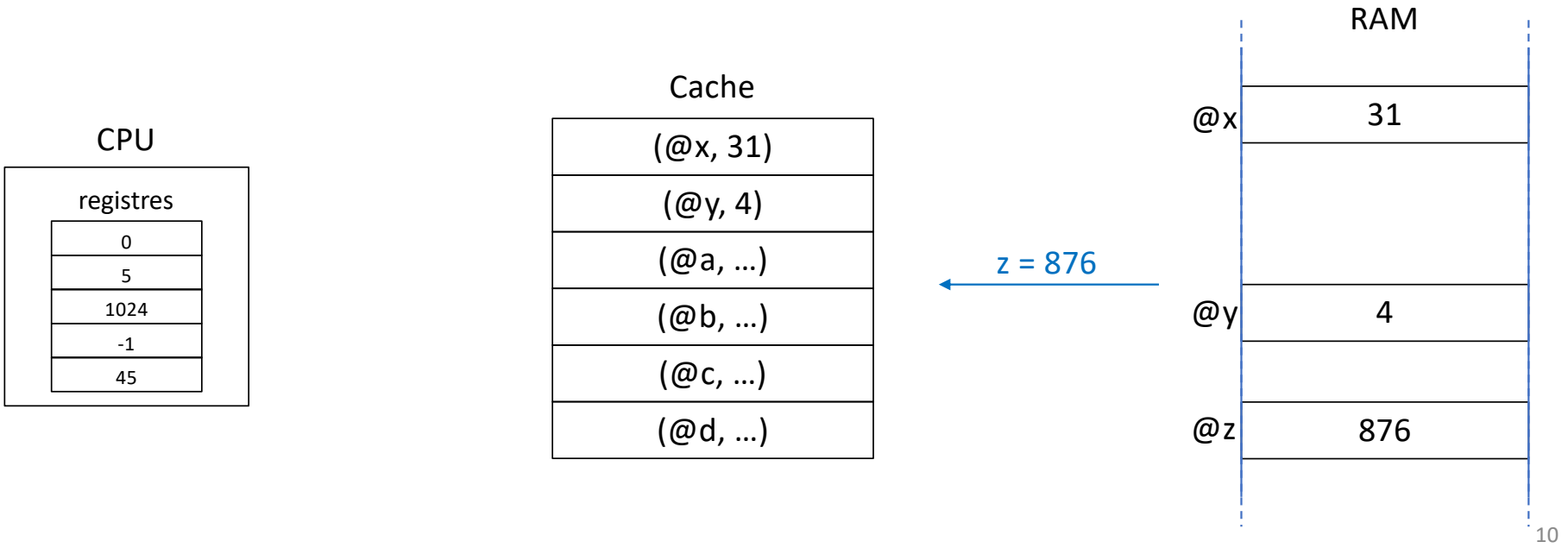
- Lorsque la RAM répond
  - Le cache conserve une copie au passage





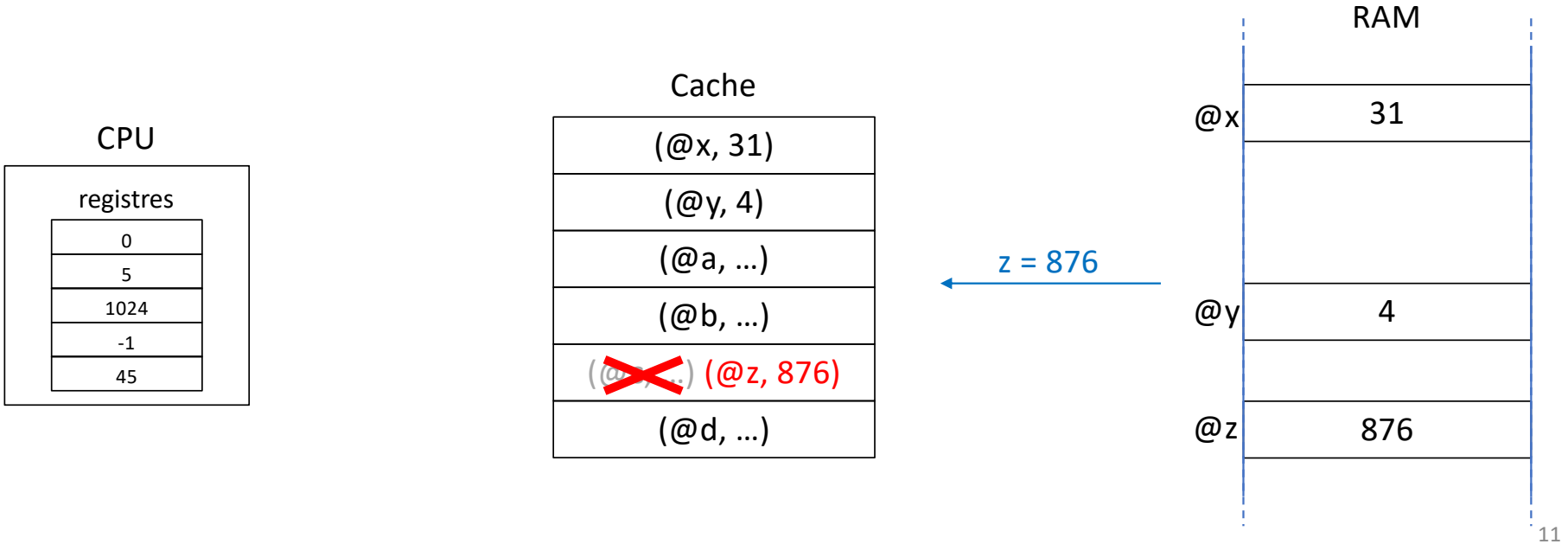
En première approximation…

- Lorsque la RAM répond mais que le cache est plein…
  - On fait quoi ??



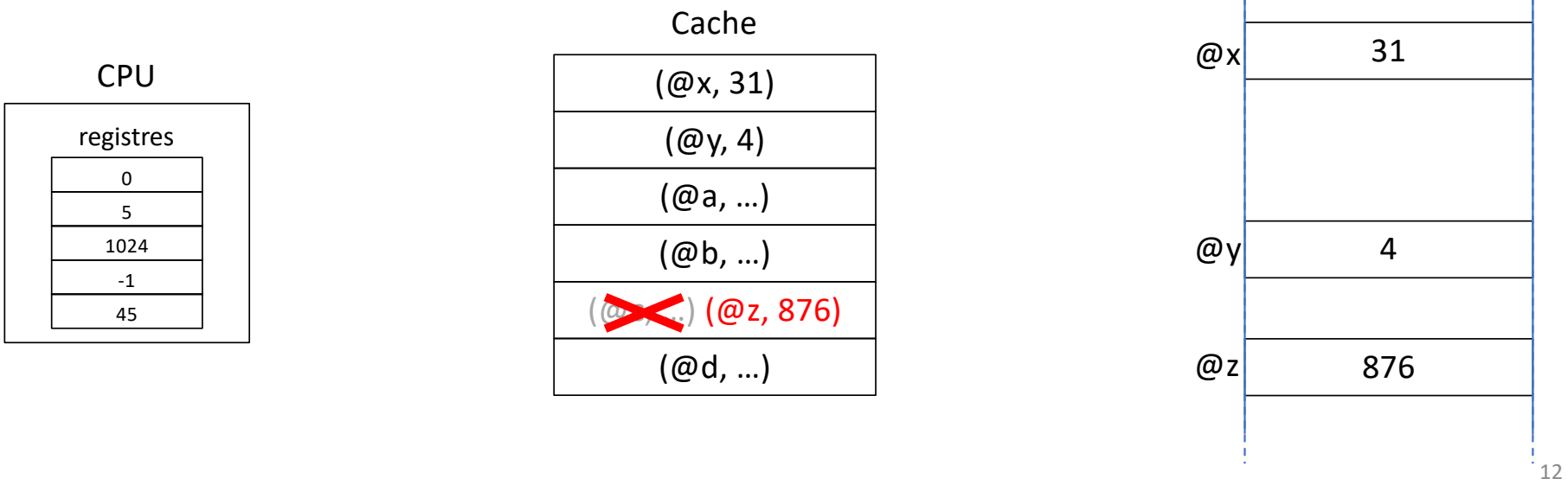
En première approximation…

- Lorsque la RAM répond mais que le cache est plein…
  - La donnée la moins récemment accédée est évincée pour faire de la place
  - Stratégie LRU (*Least Recently Used*, cf principe de localité temporelle)



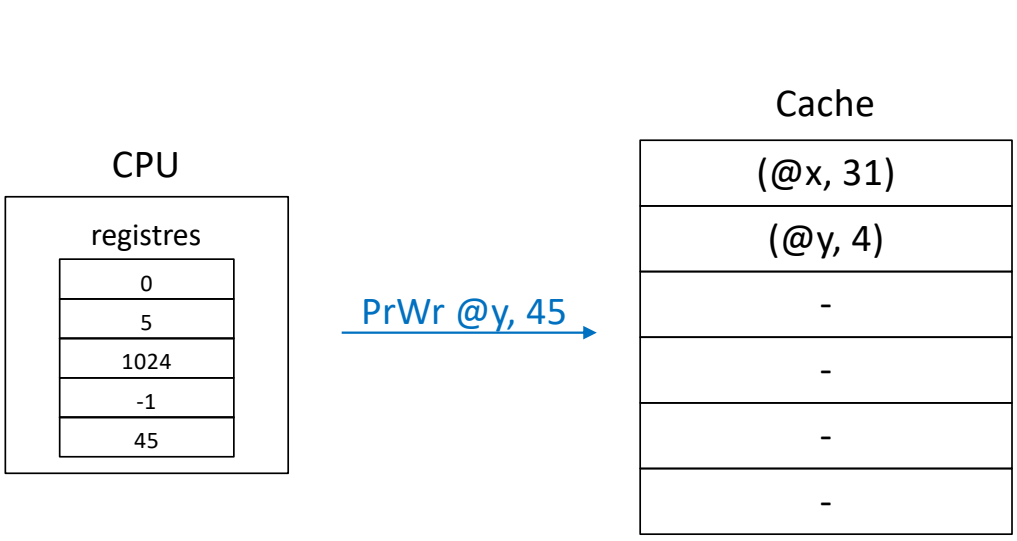
En première approximation…

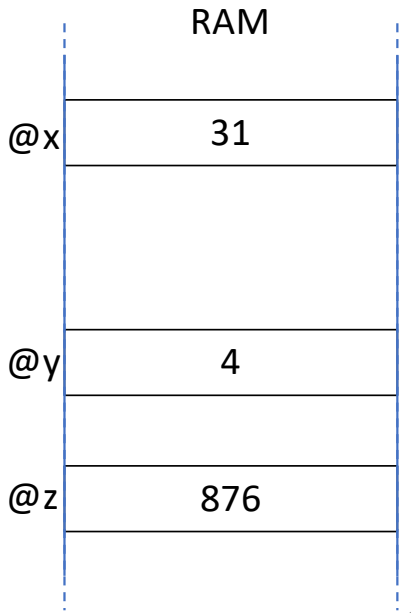
- Eviction du cache
  - En toute rigueur, il faudrait mémoriser la date du dernier hit pour chaque donnée
  - En pratique : approximation de LRU avec bit d'accès



En première approximation…

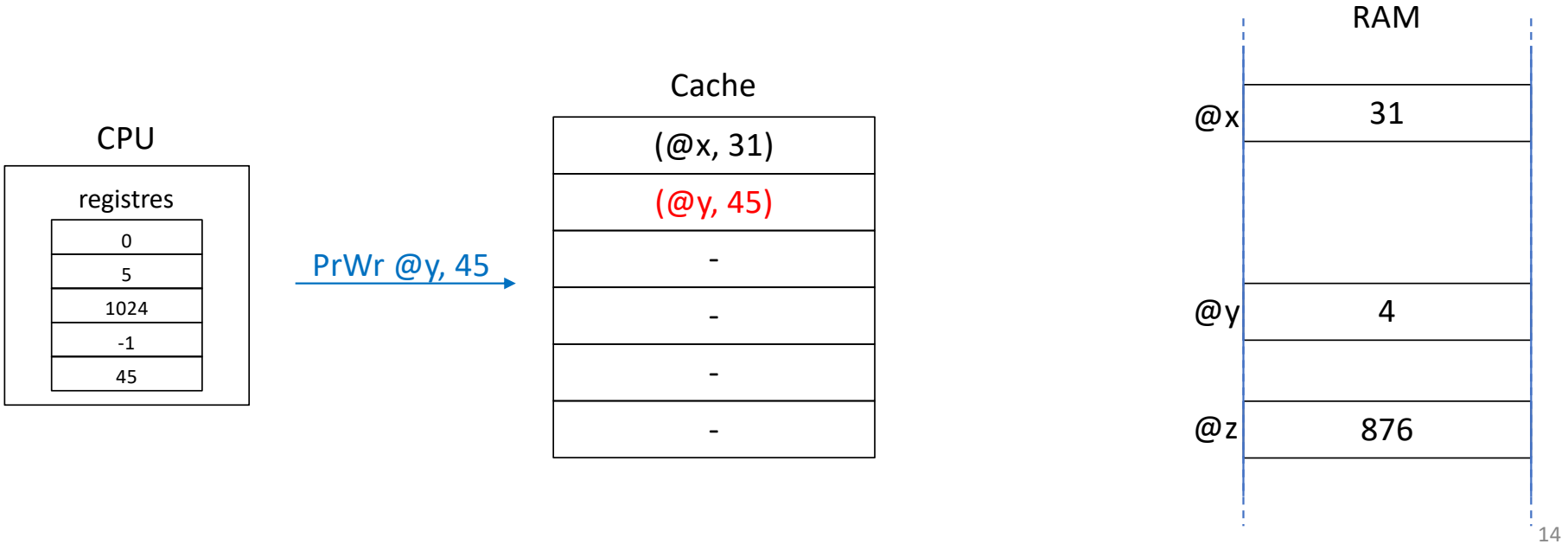
- Lorsque le CPU émet une requête d'écriture
  - Et que le cache possède la donnée…





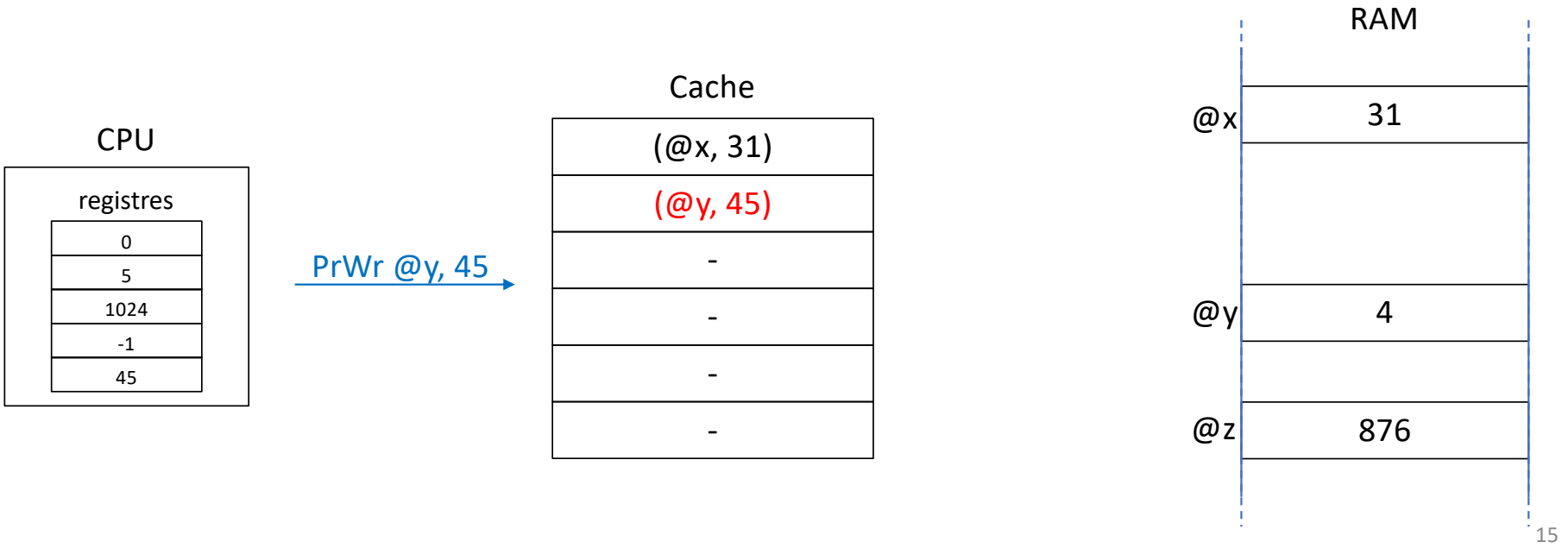
En première approximation…

- Lorsque le CPU émet une requête d'écriture
  - Et que le cache possède la donnée… le cache se met à jour
  - La RAM est mise à jour (*write through*) ou pas (*write back*, comme ici)



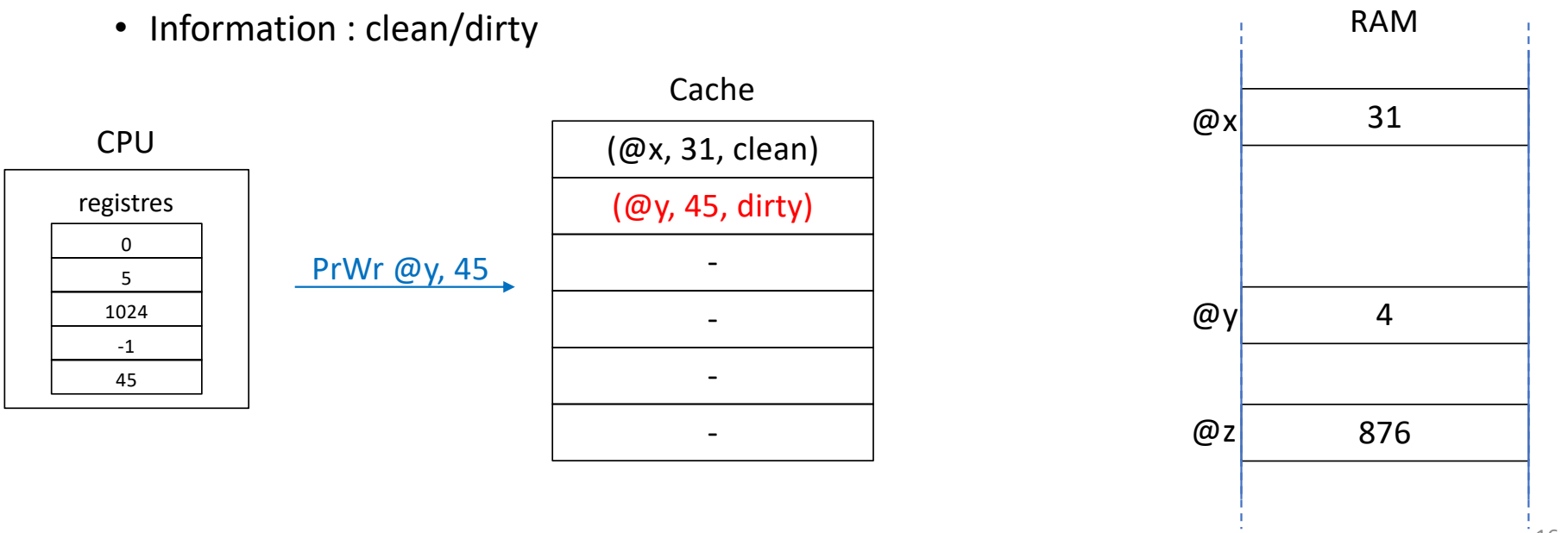
En première approximation…

- Lorsque le CPU émet une requête d'écriture
  - Et que le cache possède la donnée… le cache se met à jour
  - Problème : la RAM possède une valeur obsolète !



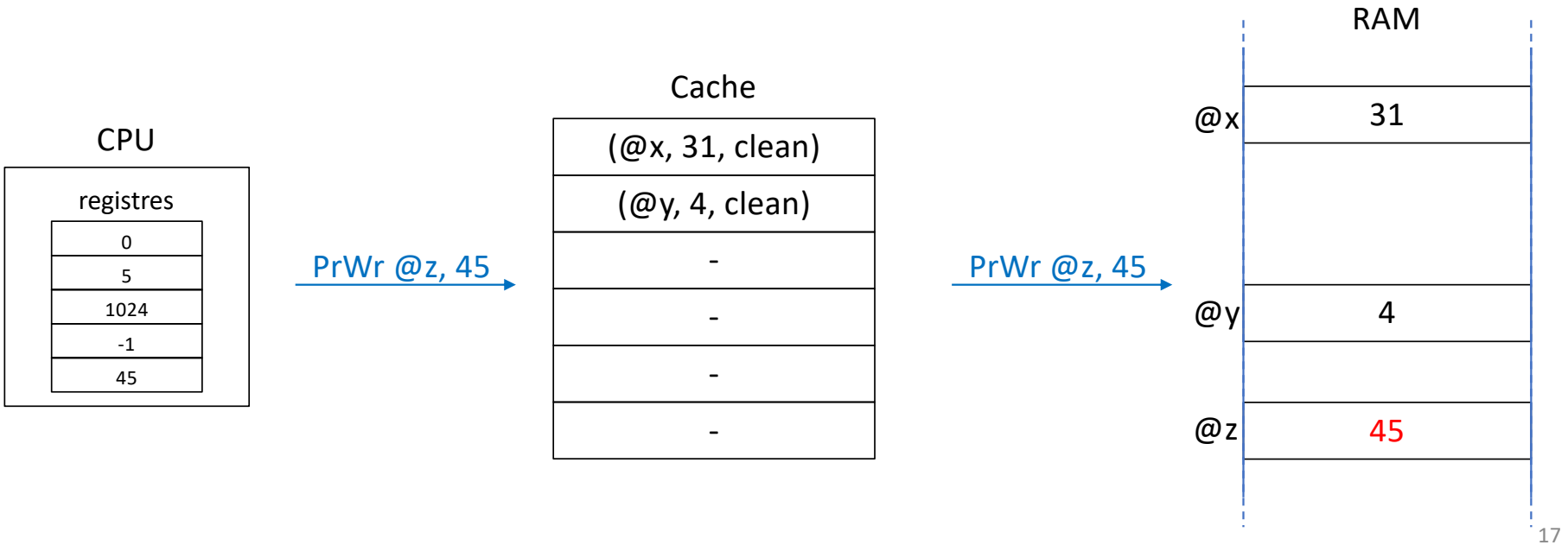
En première approximation…

- Lorsque le CPU émet une requête d'écriture
  - Et que le cache possède la donnée… le cache se met à jour
  - *write back* : la mise à jour de la RAM s'effectuera lors de l'éviction
    - Information : clean/dirty



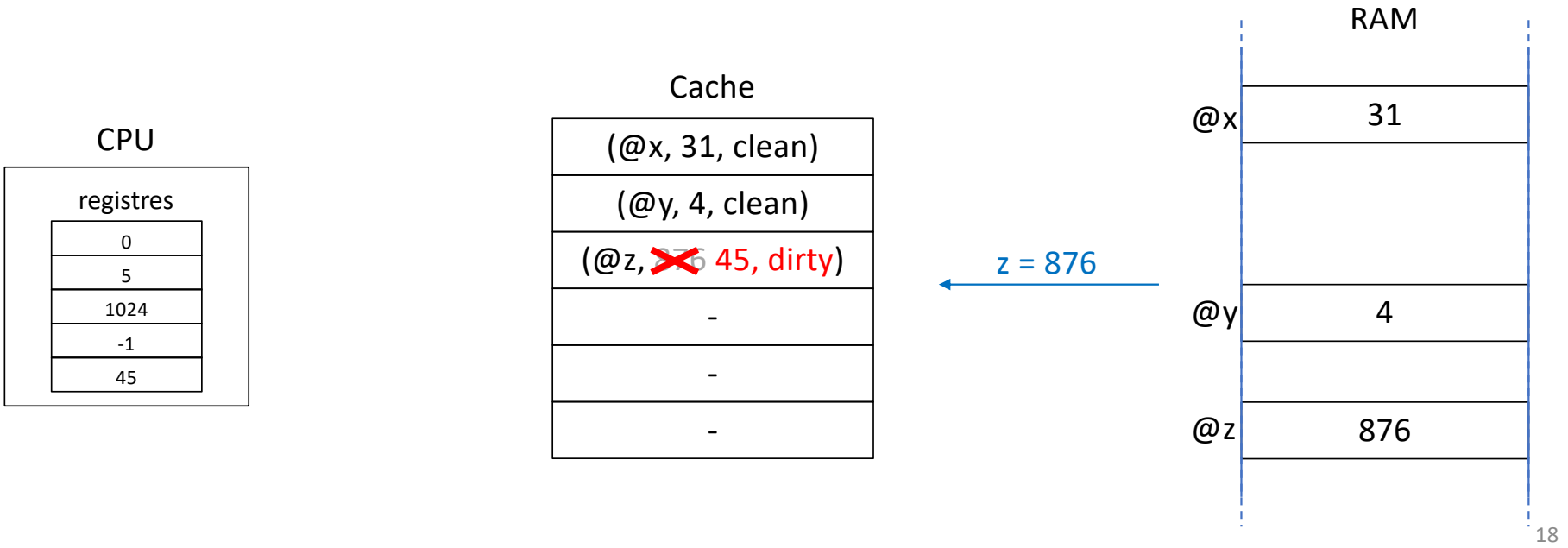
En première approximation…

- Lorsque le CPU émet une requête d'écriture
  - Et que le cache ne possède pas la donnée…
  - Soit le cache fait suivre à la mémoire (L1)

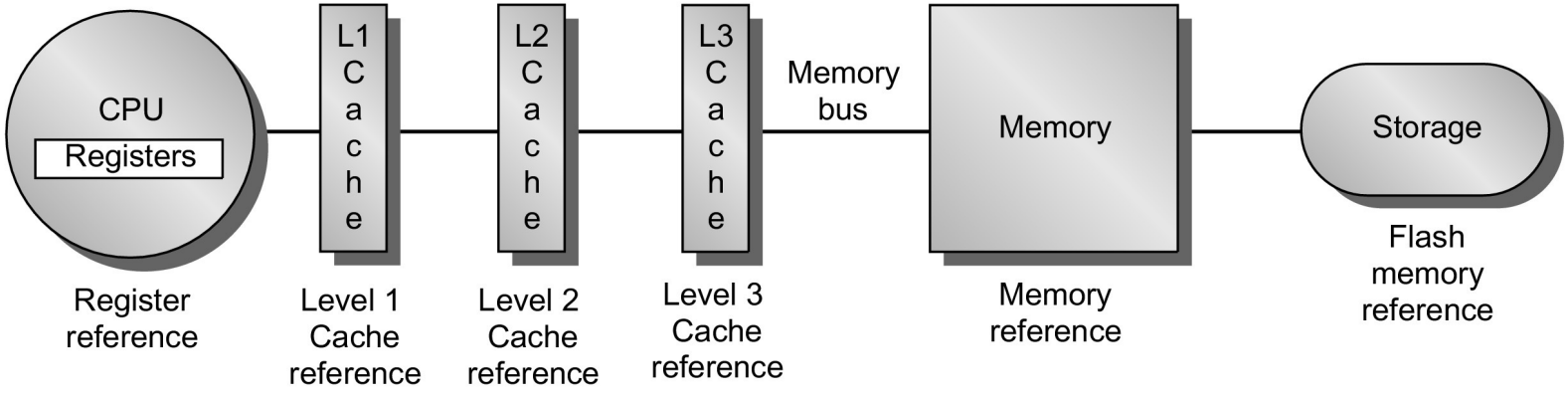


En première approximation…

- Lorsque le CPU émet une requête d'écriture
  - Et que le cache ne possède pas la donnée…
  - Soit il précharge la donnée depuis la RAM (nous verrons pourquoi plus tard)



Hiérarchie mémoire d'un ordinateur



Laptop	Size:	1000 bytes	64 KB	256 KB	4-8 MB	4–16 GB	256 GB-1 TB
	Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	50-100 uS
Desktop	Size:	2000 bytes	64 KB	256 KB	8-32 MB	8–64 GB	256 GB-2 TB
	Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	50-100 uS

(B) Memory hierarchy for a laptop or a desktop

Les caches sont souvent inclusifs (i.e. le cache L2 inclut le contenu du L1)

Localité spatiale

- Forte probabilité qu'une donnée soit accédée prochainement…
  - Si elle est à proximité d'une donnée accédée récemment
    - Éléments de tableaux, champs de structures, etc.
- Les caches mémorisent des « lignes » (typiquement 64 octets)
  - Sur une architecture 64bits, une ligne de cache de 64 octets contient 8 mots
    - 8 double / long
    - 16 float / int

Cache

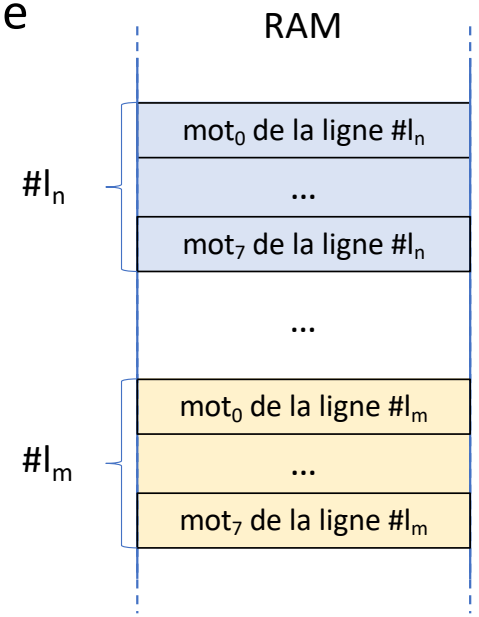
(#l_n, [mot_0, mot_1, …, mot_7], clean/dirty)
(#l_m, [mot_0, mot_1, …, mot_7], clean/dirty)
-
-
-
-

La mémoire est une séquence de lignes

- La RAM ne travaille qu'avec des lignes
  - Lorsque le CPU réclame un mot absent du cache, c'est toute la ligne de cache qui le contient qui est chargée depuis la RAM vers le cache

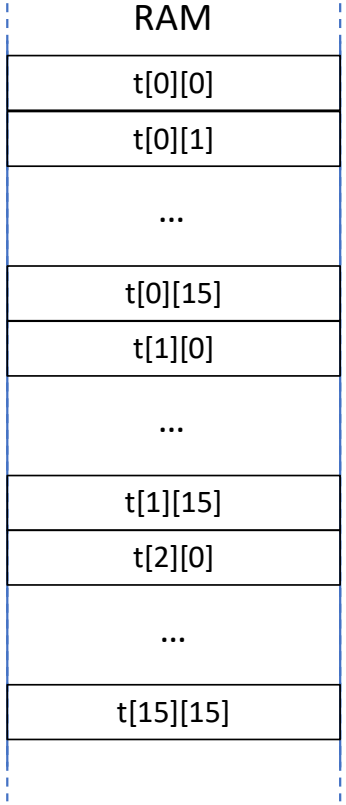
Cache

(#l_n, [mot_0, mot_1, …, mot_7], clean/dirty)
(#l_m, [mot_0, mot_1, …, mot_7], clean/dirty)
-
-
-
-



De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire de façon *row-major*
  - Exemple :
    float t [16][16];
  - t occupe 16 x 16 x 4 octets (1 KB)



De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire de façon « *row-major* »
  - Exemple :

```
float t [16][16];

for (int j = 0; j < 16; j++)
  for (int i = 0; i < 16; i++)
    sum += t[i][j];
```

  - Avec un cache de 8 lignes
  - j=0, i=0..7 : le cache est rempli !

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]
#l_{n+1}:	t[1][0], t[1][1], …, t[1][15]
#l_{n+2}:	t[2][0], t[2][1], …, t[2][15]
#l_{n+3}:	t[3][0], t[3][1], …, t[3][15]
#l_{n+4}:	t[4][0], t[4][1], …, t[4][15]
#l_{n+5}:	t[5][0], t[5][1], …, t[5][15]
#l_{n+6}:	t[6][0], t[6][1], …, t[6][15]
#l_{n+7}:	t[7][0], t[7][1], …, t[7][15]

De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire de façon « *row-major* »
  - Exemple :

```
float t [16][16];

for (int j = 0; j < 16; j++)
  for (int i = 0; i < 16; i++)
    sum += t[i][j];
```

  - Avec un cache de 8 lignes
  - j=0, i=8 : première éviction ☹

Cache

#l_{n+8}:	t[8][0], t[8][1], …, t[8][15]
#l_{n+1}:	t[1][0], t[1][1], …, t[1][15]
#l_{n+2}:	t[2][0], t[2][1], …, t[2][15]
#l_{n+3}:	t[3][0], t[3][1], …, t[3][15]
#l_{n+4}:	t[4][0], t[4][1], …, t[4][15]
#l_{n+5}:	t[5][0], t[5][1], …, t[5][15]
#l_{n+6}:	t[6][0], t[6][1], …, t[6][15]
#l_{n+7}:	t[7][0], t[7][1], …, t[7][15]

De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire de façon « *row-major* »
  - Exemple :

```
float t [16][16];

for (int j = 0; j < 16; j++)
  for (int i = 0; i < 16; i++)
    sum += tab[i][j];
```

  - Avec un cache de 8 lignes
  - J=0, i=9 : seconde éviction ☹☹

Cache

#l_{n+8}:	t[8][0], t[8][1], …, t[8][15]
#l_{n+9}:	t[9][0], t[9][1], …, t[9][15]
#l_{n+2}:	t[2][0], t[2][1], …, t[2][15]
#l_{n+3}:	t[3][0], t[3][1], …, t[3][15]
#l_{n+4}:	t[4][0], t[4][1], …, t[4][15]
#l_{n+5}:	t[5][0], t[5][1], …, t[5][15]
#l_{n+6}:	t[6][0], t[6][1], …, t[6][15]
#l_{n+7}:	t[7][0], t[7][1], …, t[7][15]

De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire de façon « *row-major* »
  - Exemple :
    float t [16][16];

    for (int j = 0; j < 16; j++)
      for (int i = 0; i < 16; i++)
        sum += t[i][j];

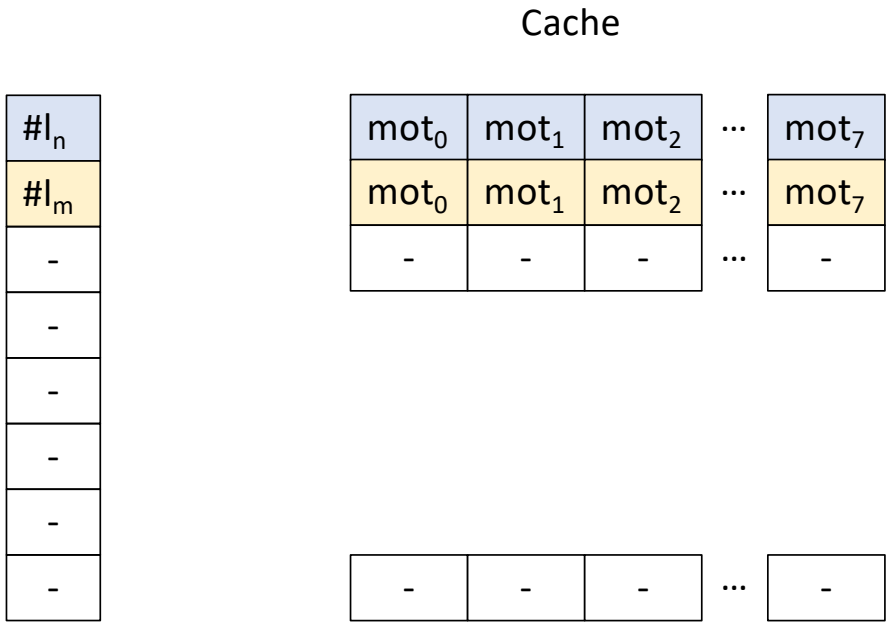
  - En fait, chaque accès va désormais provoquer une éviction…
  - Si possible, il faut parcourir t en row-major !

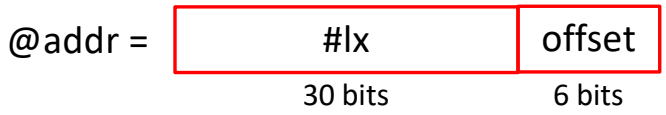
Cache

#l_{n+8}:	t[8][0], t[8][1], …, t[8][15]
#l_{n+9}:	t[9][0], t[9][1], …, t[9][15]
#l_{n+2}:	t[2][0], t[2][1], …, t[2][15]
#l_{n+3}:	t[3][0], t[3][1], …, t[3][15]
#l_{n+4}:	t[4][0], t[4][1], …, t[4][15]
#l_{n+5}:	t[5][0], t[5][1], …, t[5][15]
#l_{n+6}:	t[6][0], t[6][1], …, t[6][15]
#l_{n+7}:	t[7][0], t[7][1], …, t[7][15]

Réalisation matérielle des caches

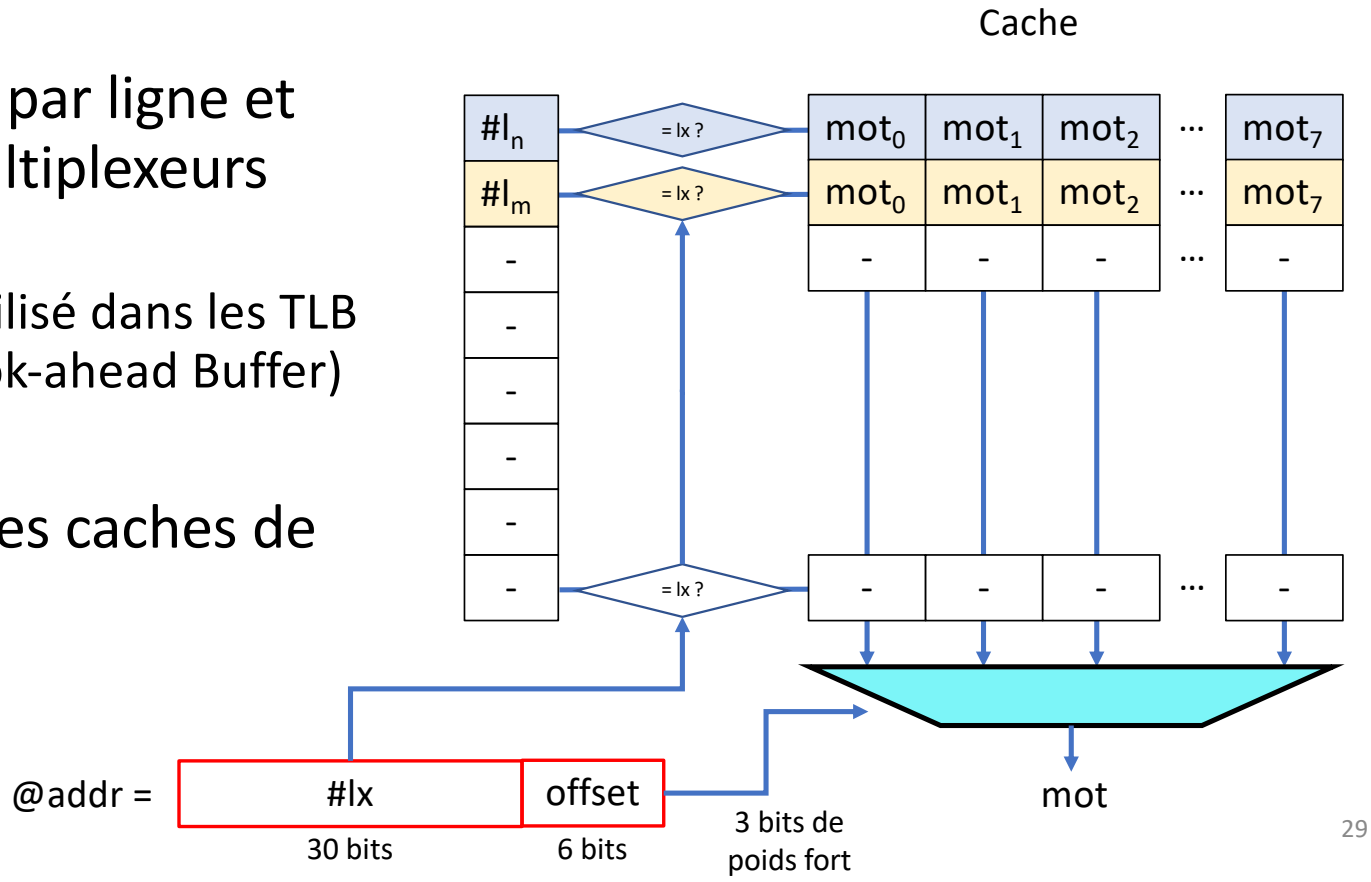
- Lorsque le CPU réclame le contenu de l'adresse @addr
  - Le cache détermine le numéro de ligne qui contient la donnée
    - #lx = @addr >> 6
  - Pour savoir très rapidement si cette ligne est présente, il faut effectuer toutes les comparaisons en parallèle





Fully Associative Cache

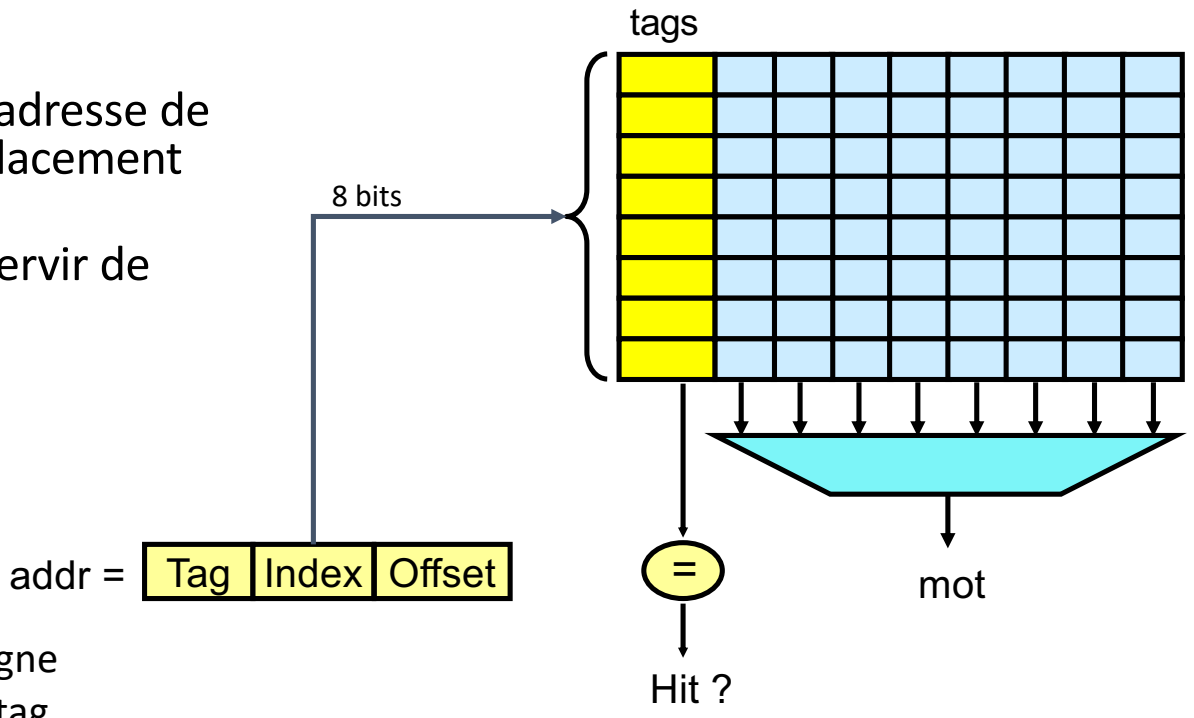
- Un comparateur par ligne et beaucoup de multiplexeurs
  - Très cher
  - Usuellement utilisé dans les TLB (Translation Look-ahead Buffer)
- Trop complexe des caches de grande taille…



Direct-Mapped Cache

- Idée
  - Les bits de poids faible de l'adresse de ligne prédéterminent l'emplacement dans le cache
  - Les bits de poids fort vont servir de « tag » (clé de hachage)

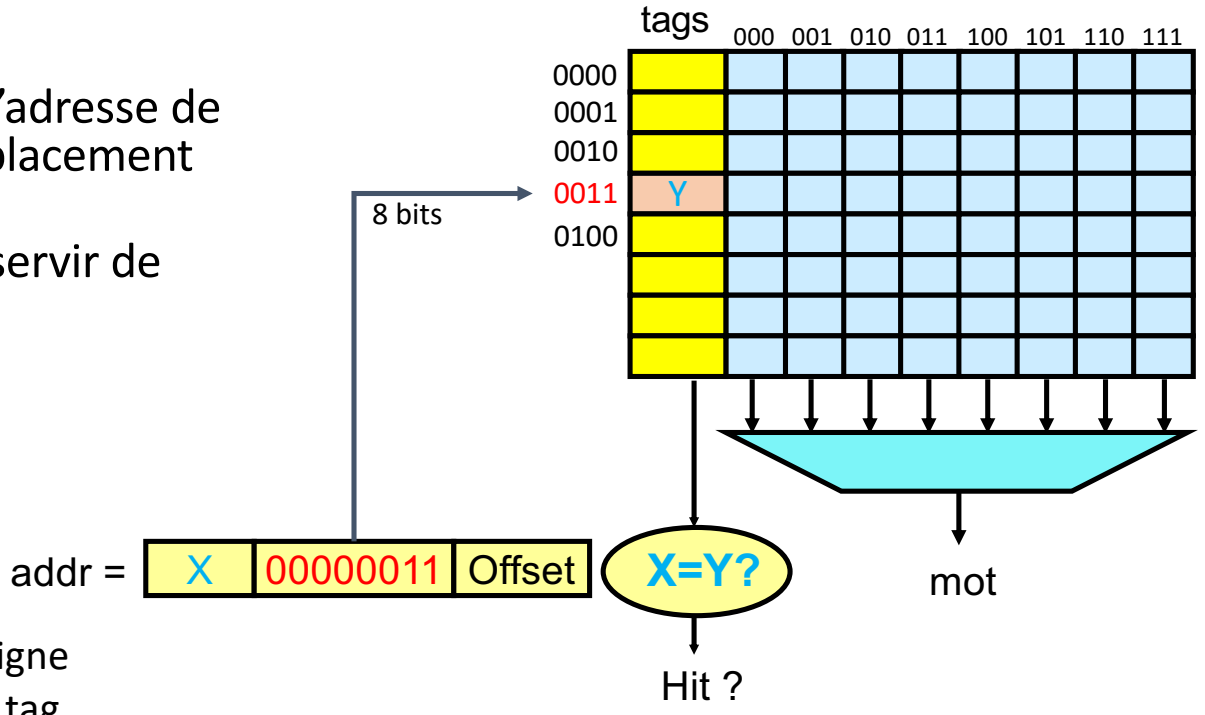
- Exemple
  - Cache de 16 KB
    - 256 x 64 octets
    - Index sur 8 bits pour déterminer le numéro de ligne
    - 36 – 6 – 8 = 22 bits pour le tag



Direct-Mapped Cache

- Idée
  - Les bits de poids faible de l'adresse de ligne prédéterminent l'emplacement dans le cache
  - Les bits de poids fort vont servir de « tag » (clé de hachage)

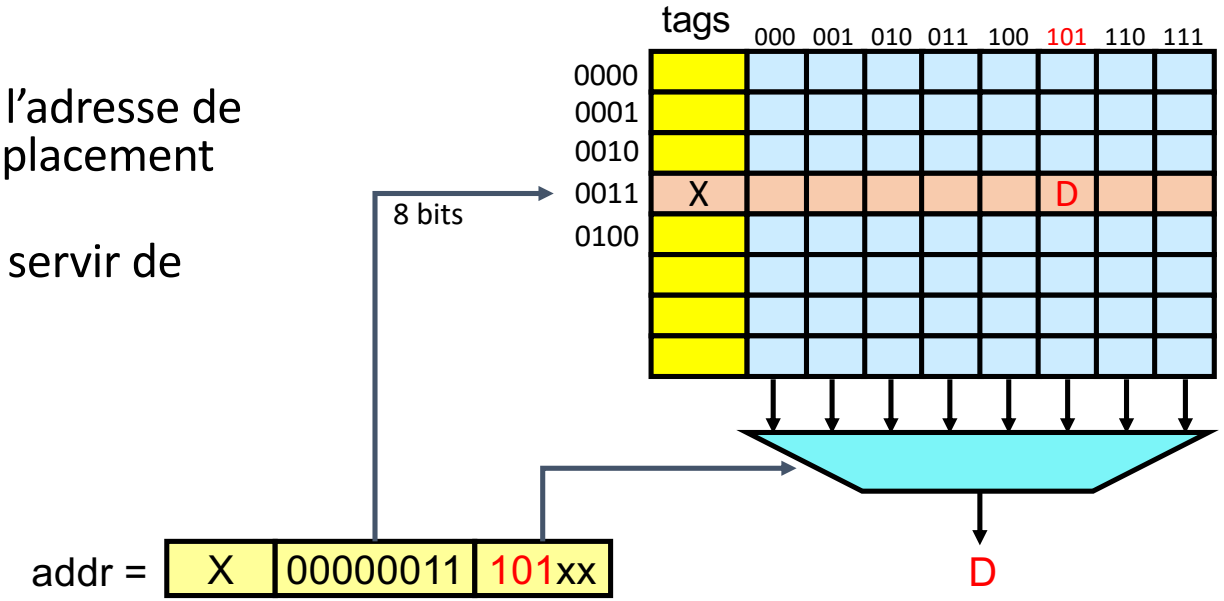
- Exemple
  - Cache de 16 KB
    - 256 x 64 octets
    - Index sur 8 bits pour déterminer le numéro de ligne
    - 36 – 6 – 8 = 22 bits pour le tag



Direct-Mapped Cache

- Idée
  - Les bits de poids faible de l'adresse de ligne prédéterminent l'emplacement dans le cache
  - Les bits de poids fort vont servir de « tag » (clé de hachage)

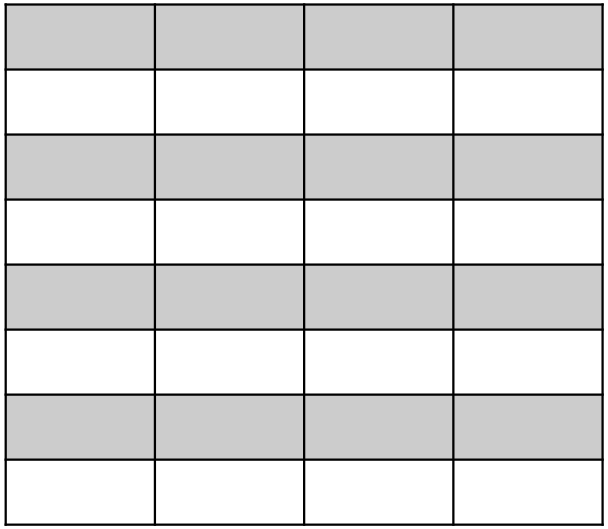
- Exemple
  - Cache de 16 KB
    - 256 x 64 octets
    - Index sur 8 bits pour déterminer le numéro de ligne
    - 36 – 6 – 8 = 22 bits pour le tag



Direct-Mapped Cache

a = 1 ?

Cache direct 4 mots / lignes



Direct-Mapped Cache

a = 1
→ Lire la ligne de cache

Cache direct 4 mots / lignes

12	43	122	45

Chargement de la ligne de a

Direct-Mapped Cache

a = 1

→ Lire la ligne de cache

→ Écrire dans la ligne de cache

Cache direct 4 mots / ligne

12	1	122	45

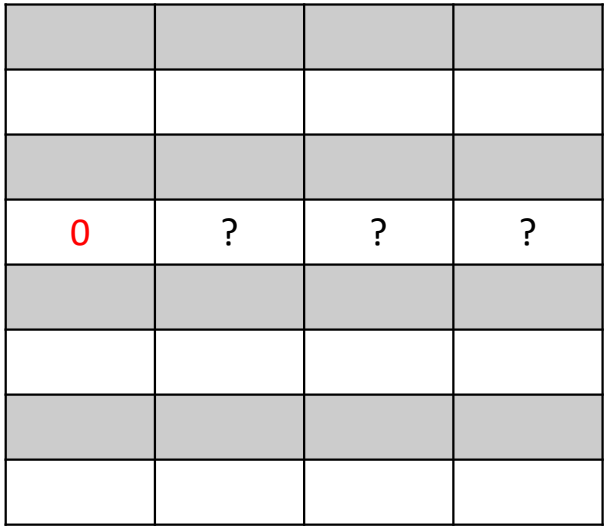
écriture de a

Direct-Mapped Cache

```
#define N 8192
long vec [N];

for (i=0; i<N; i++)
        vec [i] = i;
```

Cache direct 4 mots / ligne



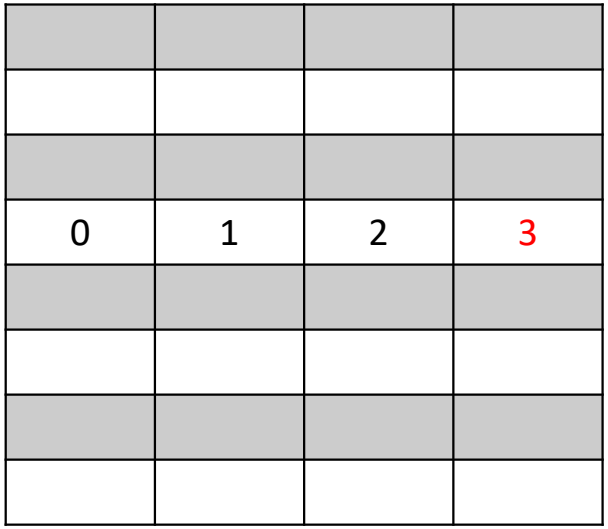
i = 0,
1 défaut

Direct-Mapped Cache

```
#define N 8192
long vec [N];

for (i=0; i<N; i++)
        vec [i] = i;
```

Cache direct 4 mots / ligne



i = 3,
1 défaut

Direct-Mapped Cache

```
#define N 8192
long vec [N];

for (i=0; i<N; i++)
        vec [i] = i;
```

Cache direct 4 mots / ligne

0	1	2	3
4	?	?	?

i = 4,
2 défauts

Direct-Mapped Cache

```
#define N 8192
long vec [N];

for (i=0; i<N; i++)
        vec [i] = i;
```

Cache direct 4 mots / ligne

0	1	2	3
4	5	6	7
8	?	?	?

i = 8,
3 défauts

Direct-Mapped Cache

```
#define N 8192
long vec [N];

for (i=0; i<N; i++)
        vec [i] = i;
```

Cache direct 4 mots / ligne

20	21	22	23
24	25	26	27
28	29	30	31
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

i = 31,
8 défauts

Direct-Mapped Cache

```
#define N 8192
long vec [N];

for (i=0; i<N; i++)
        vec [i] = i;
```

Cache direct 4 mots / ligne

20	21	22	23
24	25	26	27
28	29	30	31
32	?	?	?
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

i = 32,
9 défauts,
1 écriture

Direct-Mapped Cache

- Note: avec un *stride* c'est pire

```
#define N 8192
long vec [N];

for (i=0; i<N; i += 8)
        vec [i] = i;
```

Cache direct 4 mots / ligne

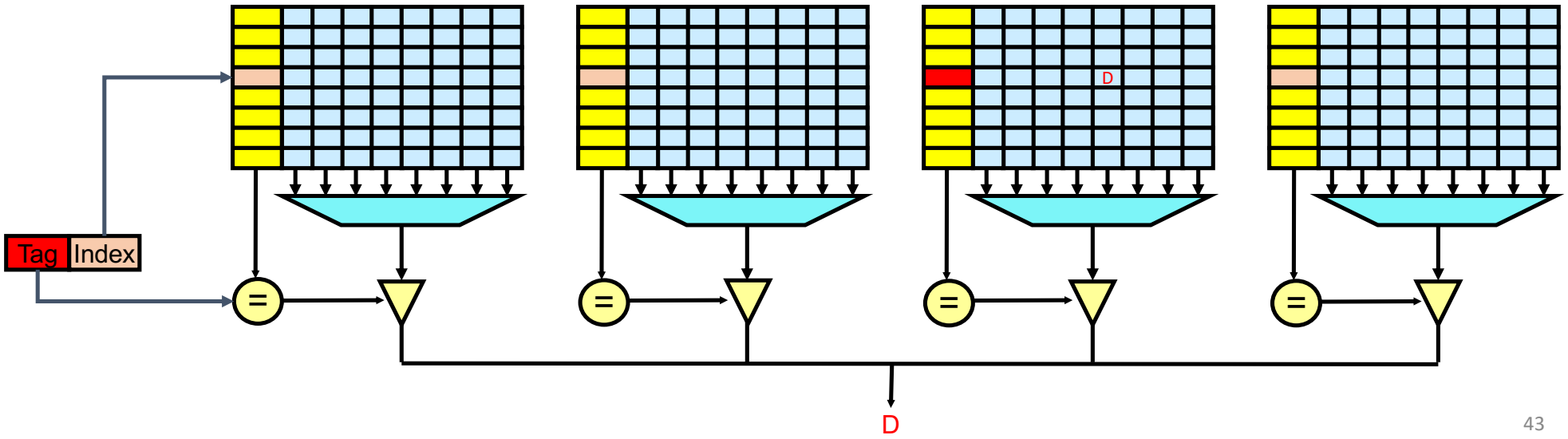
?	?	?	?
24	?	?	?
?	?	?	?
0	?	?	?
?	?	?	?
8	?	?	?
?	?	?	?
16	?	?	?

- Parcours en colonne, tableaux de structures…

Premier défaut pour i = 32,
à la 5^{e} itération

N-way associative Cache

- On utilise N sous-caches directs (= bancs)
  - N lignes de même index (mais de tag différent) peuvent co-exister
  - Exemple: 4 voies x 128 lignes x 64 octets = 4 bancs de 8 BKB = 32 KB



N-way associative Cache

- Exemple : core i7
  - Cache L1D du Intel core i7
    - 32KB, 8 voies, 64 lignes de 64 octets par voie
    - 8 bancs de 4 KB
    - Index = 6 bits
  - Cache L2
    - 256KB, 8 voies, 512 lignes de 64 octets par voie
    - 8 bancs de 32 KB
    - Index = 9 bits
  - Cache L3
    - 8MB, 16 voies, 8192 lignes de 64 octets par voie
    - 16 bancs de 512 KB
    - Index = 13 bits

N-way associative Cache

- Exemple : core i7
  - Cache L1D du Intel core i7
    - 32KB, 8 voies, 64 lignes de 64 octets par voie
    - 8 bancs de 4 KB

    - Si la variable x est dans le cache, alors
      - (&x + 1*4096),
      - (&x + 2*4096)
      - …
      - (&x + 7*4096) pourront entrer dans le cache, chacune dans un banc différent

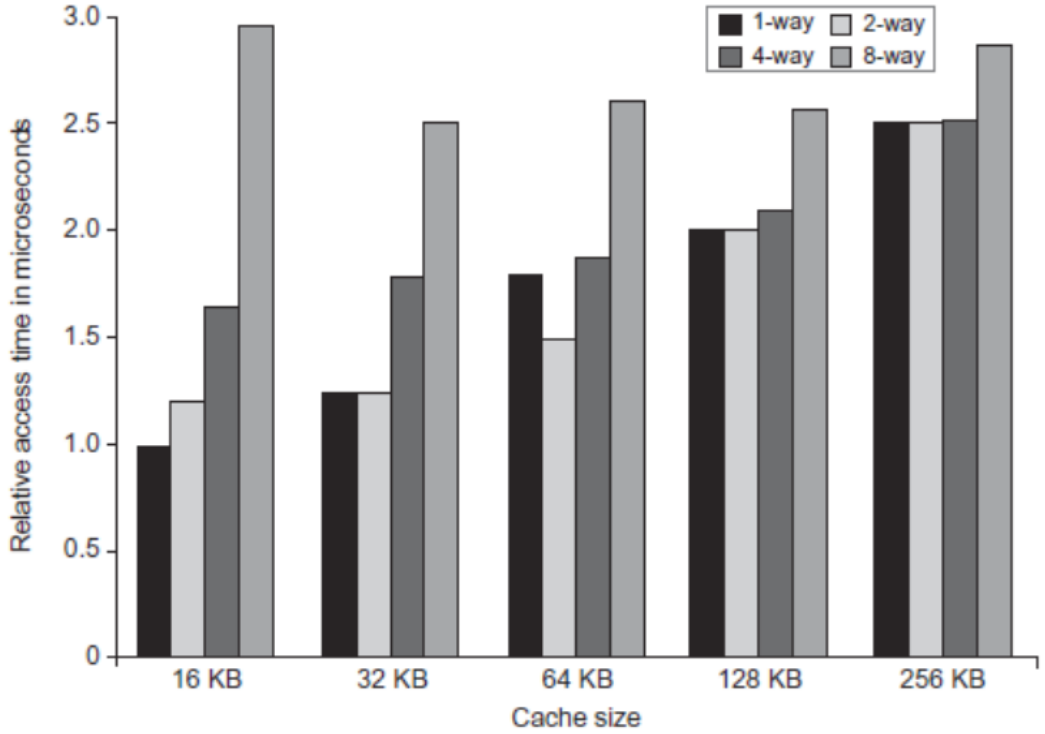
      - Un accès à (&x + 8*4096) provoquera l'éviction d'une des huit lignes
      - Attention aux strides multiples de 4KB !

N-way associative Cache

- Finalement, les conflits surgissent moins vite avec une associativité élevée
  - Exemple : 16 voies pour de nombreux caches L3

- Pourquoi ne pas augmenter franchement l'associativité et proposer 128 ou 1024 voies ?
  - Parce qu'en augmentant le nombre de bancs et en réduisant le nombre de lignes par banc, on augmente le nombre de comparateurs et la complexité des multiplexeurs
  - À l'extrême, on tendrait vers un cache *full associative* !

L1 Size and Associativity



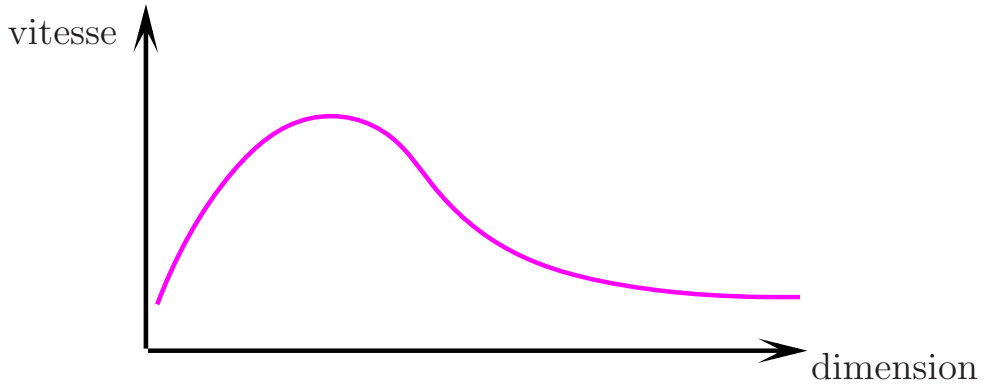
Access time vs. size and associativity

Optimisation de l'utilisation du cache

- Minimiser le nombre de défauts de cache
  - Réutiliser les données chargées dans le cache
    - Adapter les structures de données
    - Aligner les données

- Techniques de pavage (tiling)

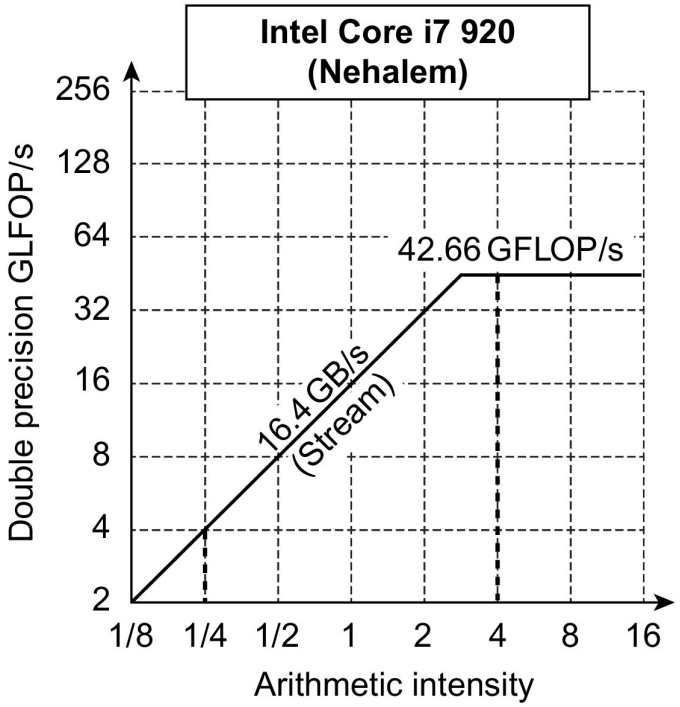
```
for( I =; ...; I+=TI)
    for( J =; ...; J+=TJ)
        for( i = I; i < I + TI ; i++)
            for( j = J; j < J + TJ ; j++)
```



Optimisation de l'utilisation du cache

- Types de défauts de cache (*cache misses*)
  - ***Cold miss***
    - Premier accès à une variable
    - Regrouper des variables (structures, tableaux)
  - ***Capacity miss***
    - Le cache est complet
    - Revoir la localité,
    - Travailler plus intensivement sur une plus petite zone de données (cf tuiles)
  - ***Conflict miss***
    - Le cache n'est pas complet mais son associativité est trop faible.
    - Revoir l'alignement des données
    - Regrouper des variables (structures, tableaux)

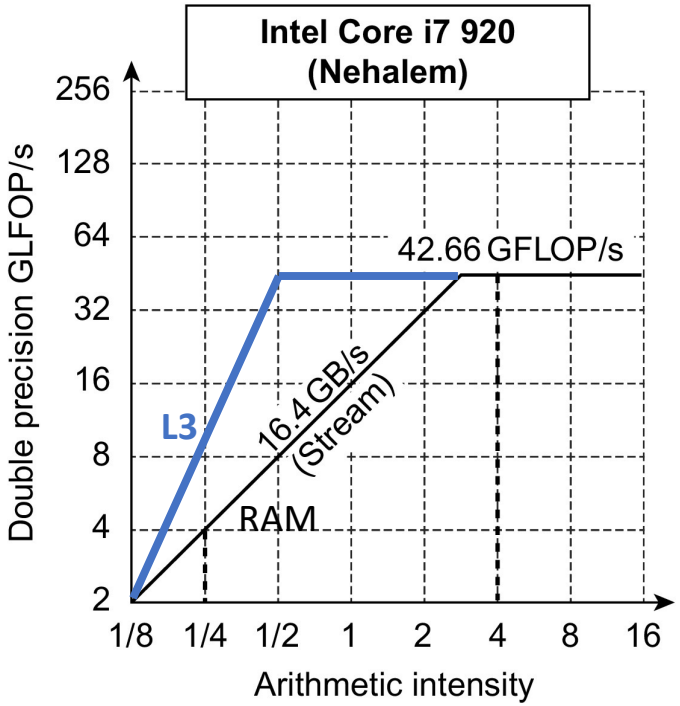
Roofline model



Hennessy & Pattersson

- Performance en fonction du nombre d'opérations par octet
- Exemple : pour bénéficier de la puissance de calcul du core i7 il faut :
  - 3 opérations / octet
  - 12 opérations / float
  - 24 opérations / double

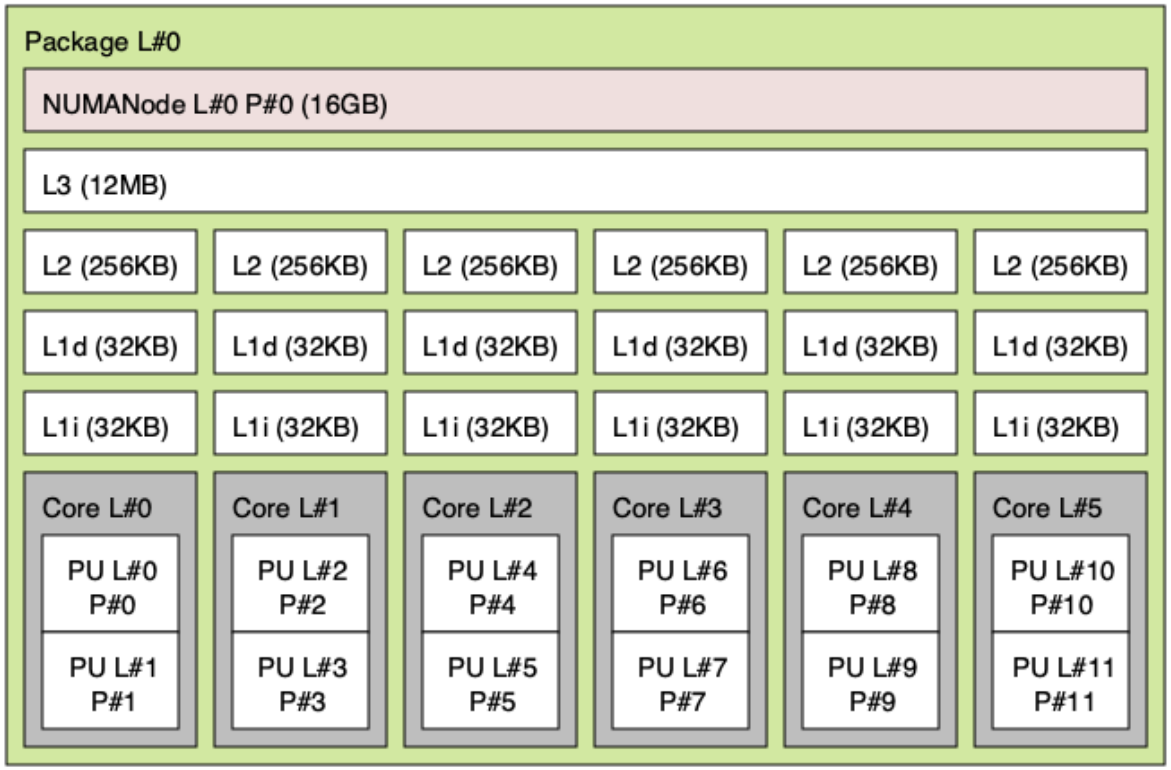
Roofline model



Hennessy & Pattersson

- Performance en fonction du nombre d'opérations par octet
- Si la donnée est dans le cache L3, ½ opération par octet suffit !

Caches et multicœurs

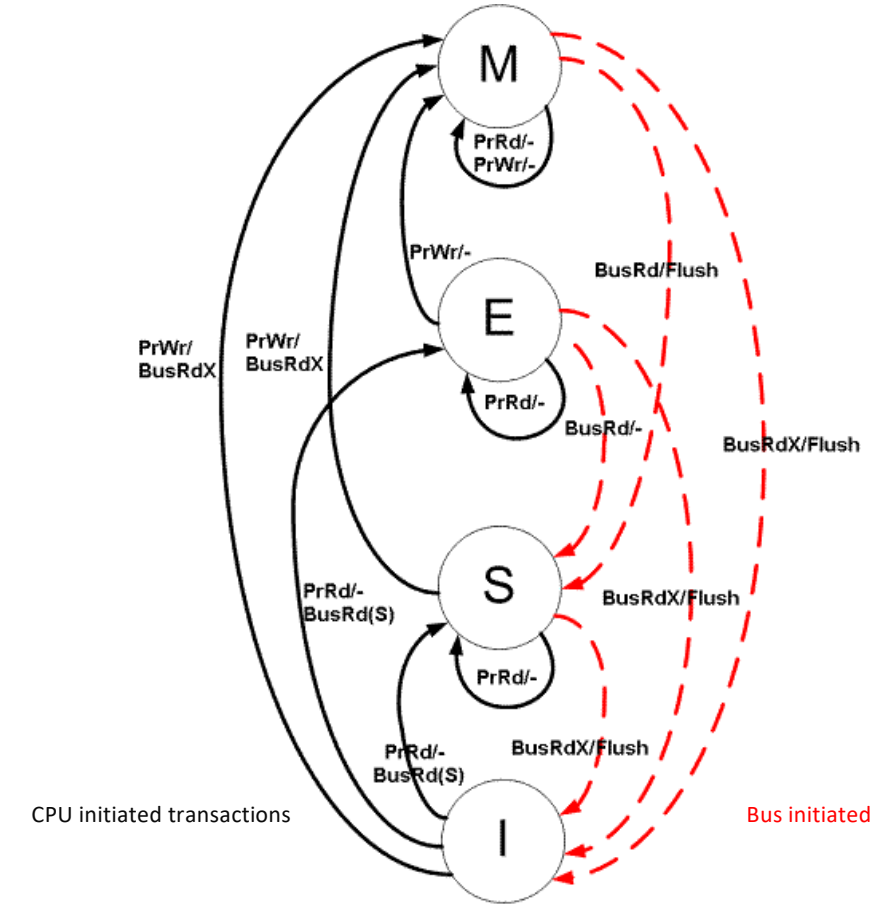


Caches et multicœurs

- La mémoire et les caches forment une hiérarchie
  - Très variable d'une architecture/d'un constructeur à l'autre
- Certains caches sont privés, d'autres partagés par un sous-ensemble (éventuellement tous) les CPU
- Il faut garantir la cohérence des données en présence de copies !
  - De nombreux protocoles de cohérence ont été proposés
    - MSI, MESI, MOESI, MESIF, etc.
    - On associe un état à chaque ligne du cache qui permet d'optimiser les échanges entre les caches et la mémoire

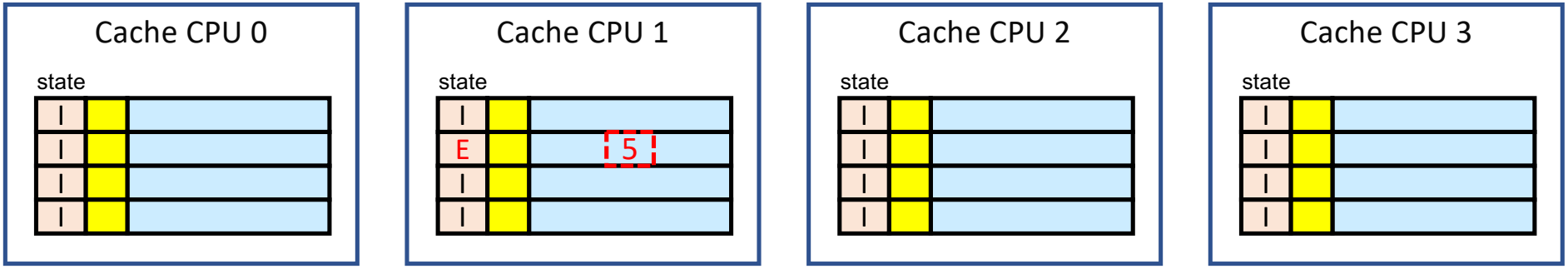
Caches et multicœurs

- Le protocole MESI
  - Chaque ligne de cache peut être dans l'un de ces 4 etats :
    - M (modified)
      - La ligne n'est présente que dans le cache actuel, et est *dirty*
    - E (exclusive)
      - La ligne n'est présente que dans le cache actuel, et est *clean*
    - S (shared)
      - La ligne est peut-être présente dans d'autres caches, et est *clean*
    - I (invalid)
      - La ligne ne contient pas de données



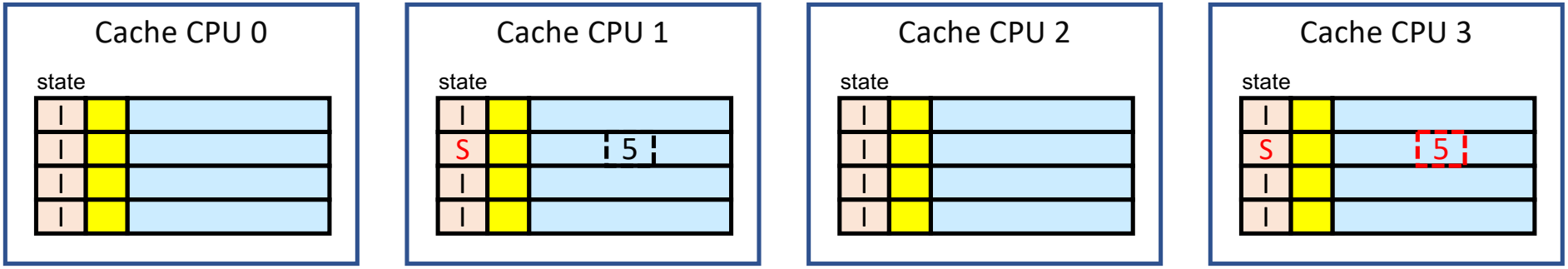
Caches et multicœurs

- CPU_1: PrRd (x) (qui vaut 5 en RAM)
  - Le signal BusRd est envoyé sur le bus
  - Les autres caches répondent qu'ils ne possèdent pas de copie
  - La ligne de cache vient de la RAM, et son état est *Exclusive*



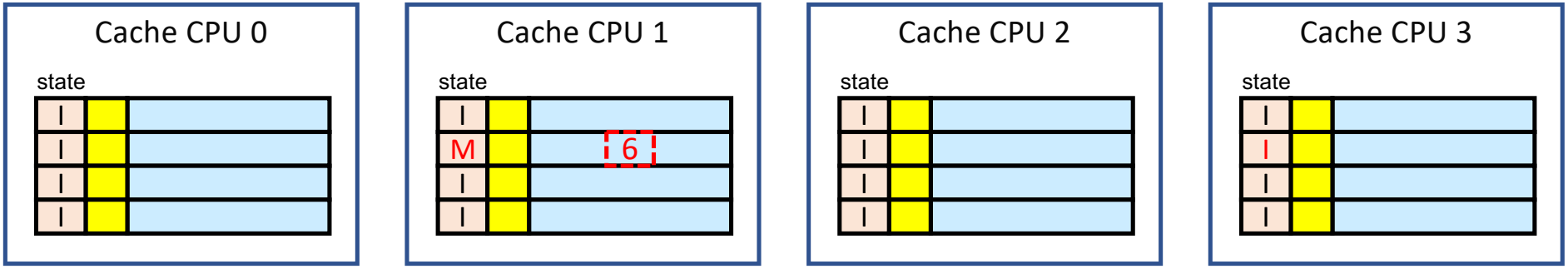
Caches et multicœurs

- CPU_3: PrRd (x)
  - Le signal BusRd est envoyé sur le bus
  - CPU_1 envoie la ligne de cache à CPU_3
  - L'état de la ligne passe à *Shared* pour CPU_1 et CPU_3



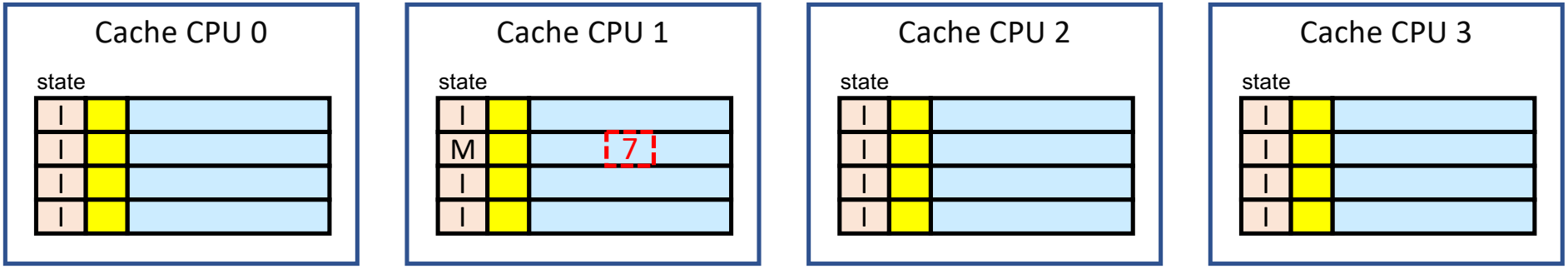
Caches et multicœurs

- CPU_1: PrWr (x, 6)
  - Le signal BusRdX est envoyé sur le bus
  - CPU_3 invalide sa copie
  - L'état de la ligne passe à *Modified* pour CPU_1



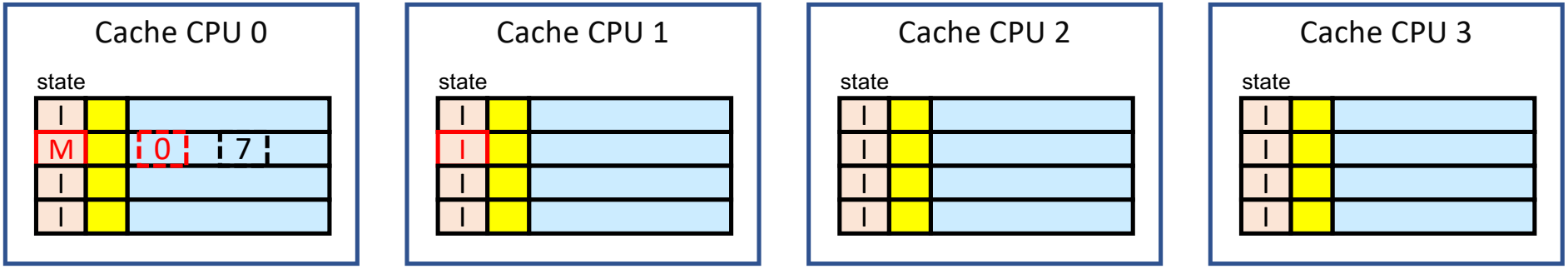
Caches et multicœurs

- CPU_1: PrWr (x, 7)
  - Aucun signal n'est envoyé sur le bus
  - L'état de la ligne reste à *Modified*



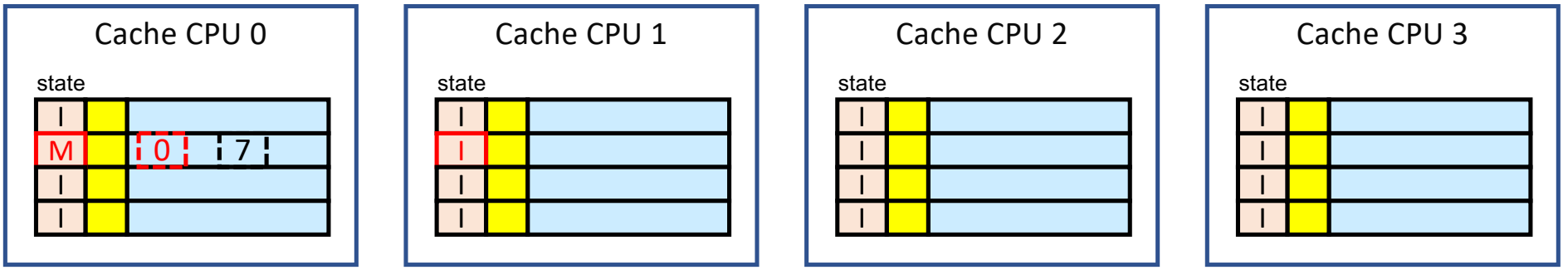
Caches et multicœurs

- CPU_0: PrWr (y, 0), y se trouvant dans la même ligne de cache que x
  - Le signal BusRdX est envoyé sur le bus
  - CPU_1 envoie la ligne à CPU_0 puis invalide sa copie
  - L'état de la ligne passe à *Modified* pour CPU_0



Caches et multicœurs

- Si CPU_1 modifie x, puis CPU_0 modifie y, etc.
  - On assiste à un ping-pong entre les caches
  - Il s'agit d'une situation de faux-partage !
    - Introduction de padding nécessaire



Caches et multicœurs

- Si CPU_1 modifie x, puis CPU_0 modifie y, etc.
  - On assiste à un ping-pong entre les caches
  - Il s'agit d'une situation de faux-partage !
    - Introduction de *padding* pour séparer les données

- À la lumière de ce constat, on peut se poser des questions sur l'efficacité d'un code tel que :

```
float tab [N];

#pragma omp parallel for schedule(static, 1)
      for (int i = 0; i < N; i++)
            tab [i] = f (i);
```