

► Programmation parallèle

Cycle 2019-2020

ECE – Ing5 – Systèmes Embarqués

Alexandre Berne

aberne@inseec-edu.com

Etienne Hamelin

ehamelin@inseec-edu.com

► Le récap' de la semaine dernière

- Rappels: architecture d'un CPU,
- Quête historique de performance
- Taxonomie
 - SISD, SIMD, MIMD: *{Single/Multiple}-Instruction, {S/I}-Data*
- Analyse quantitative
 - Métriques, latence, *speedup*,
 - Loi d'Amdahl
- Risques associés à la concurrence
- Fonctionnement d'un cache
- TP PThreads



► Séance 2

Hiérarchie mémoire

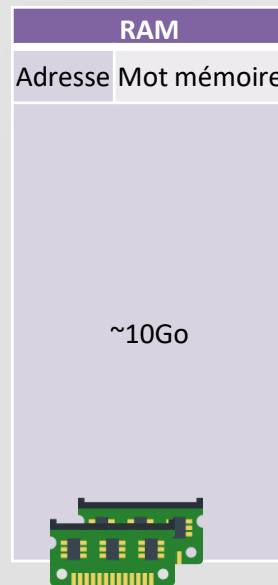
- **Le problème de mémoire**
- **Fonctionnement d'un cache**
- **Cohérence**



Comment fonctionne un cache ?

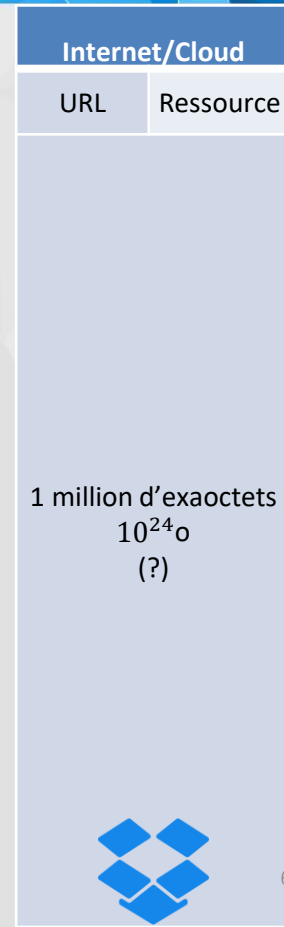
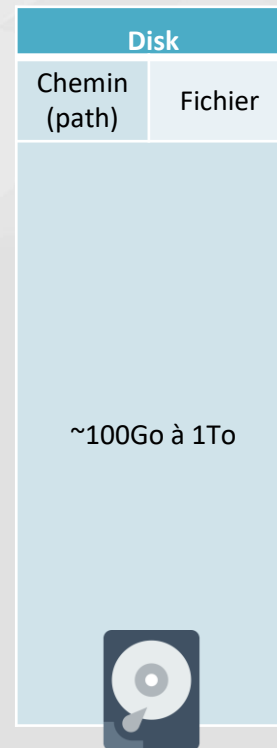
Séance 2

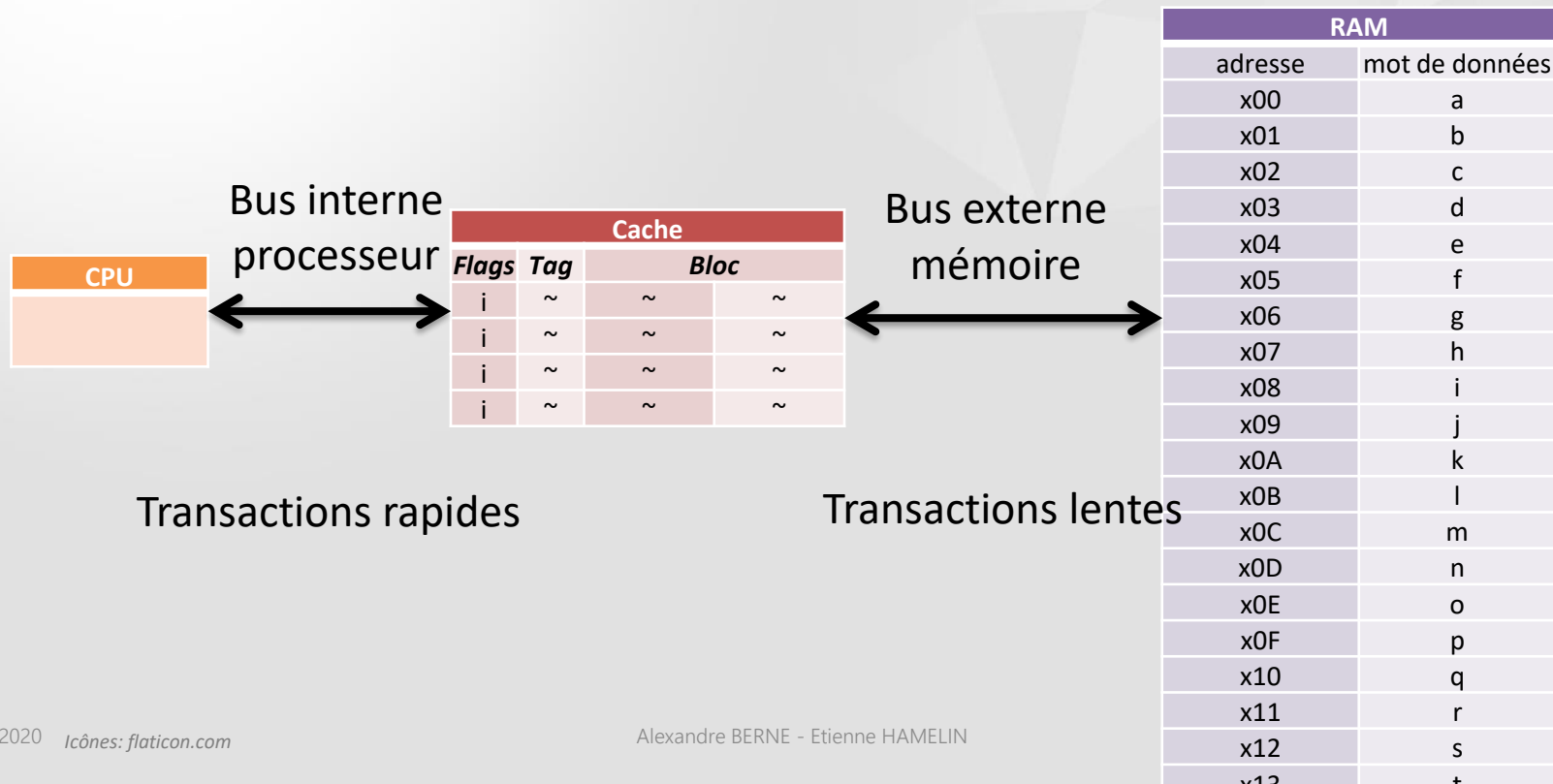
Principe de cache



Séance 2

Principe de cache





▶ Localité temporelle

- Adresse accédée récemment → probabilité d'être réutilisée bientôt
⇒ politiques de remplacement

▶ Localité spatiale

- Adresse accédée récemment → probabilité d'utiliser les adresses adjacentes
⇒ gestion par blocs

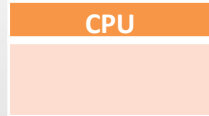
▶ Exemple

```
for (i=0; i < n; i++) {  
    for (j=0; j < n; j++) {  
        y[i] += a[i][j] * x[j];  
    }  
}
```


► Paramètres caractéristiques d'un cache

- Capacité du cache, taille de bloc
- Degré d'associativité
 - combien de choix ai-je pour placer un bloc de données?
- Lors des remplacements, quelle ligne dois-je évincer ?
 - politique de remplacement
- Comment propager les écritures?
 - politique d'écriture
- Comment gérer les accès concurrents?
 - politique de cohérence

Cache
direct-mapped
 4 lignes
 blocs de 2 octets



Cache			
Flags	Tag	Bloc	
i	~	~	~
i	~	~	~
i	~	~	~
i	~	~	~

RAM	
adresse	mot de données
x00	a
x01	b
x02	c
x03	d
x04	e
x05	f
x06	g
x07	h
x08	i
x09	j
x0A	k
x0B	l
x0C	m
x0D	n
x0E	o
x0F	p
x10	q
x11	r
x12	s
x13	t

1 bloc

Lecture

> read @x00 ?



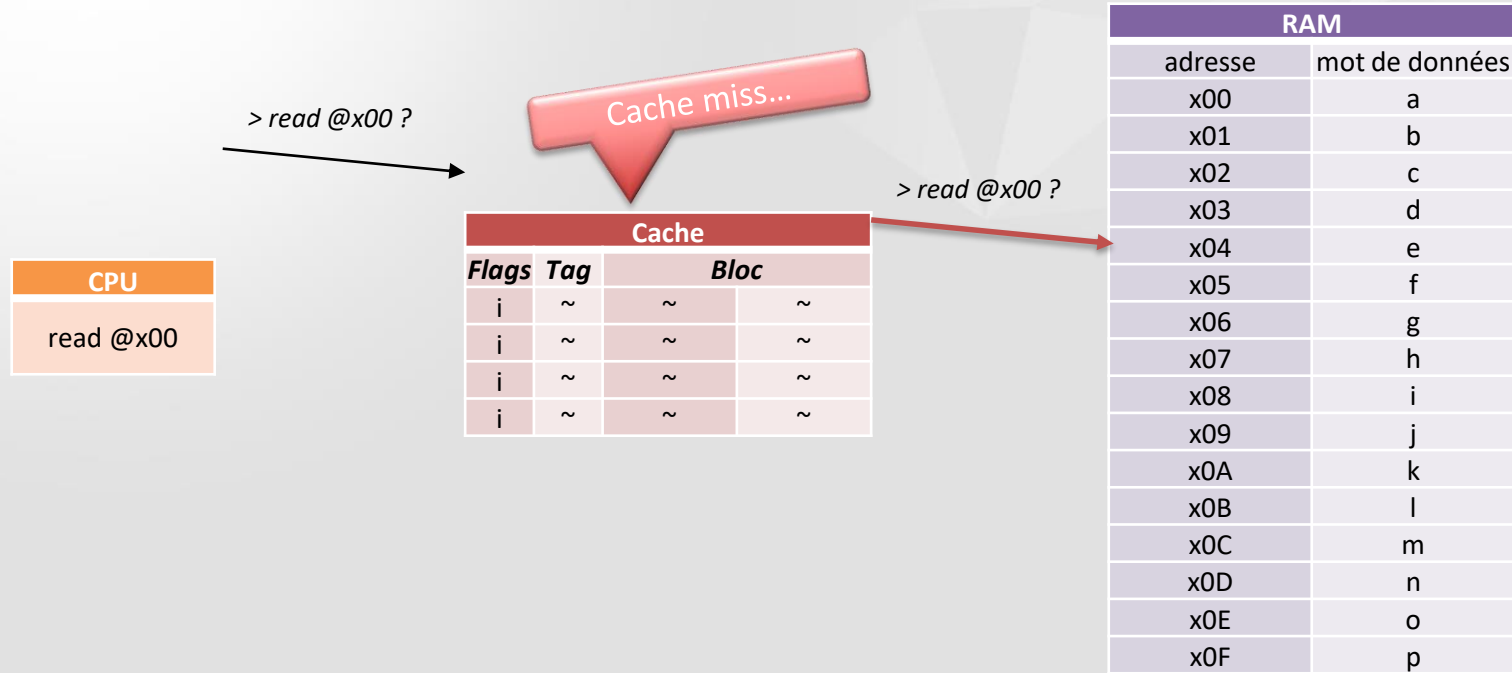
CPU
read @x00

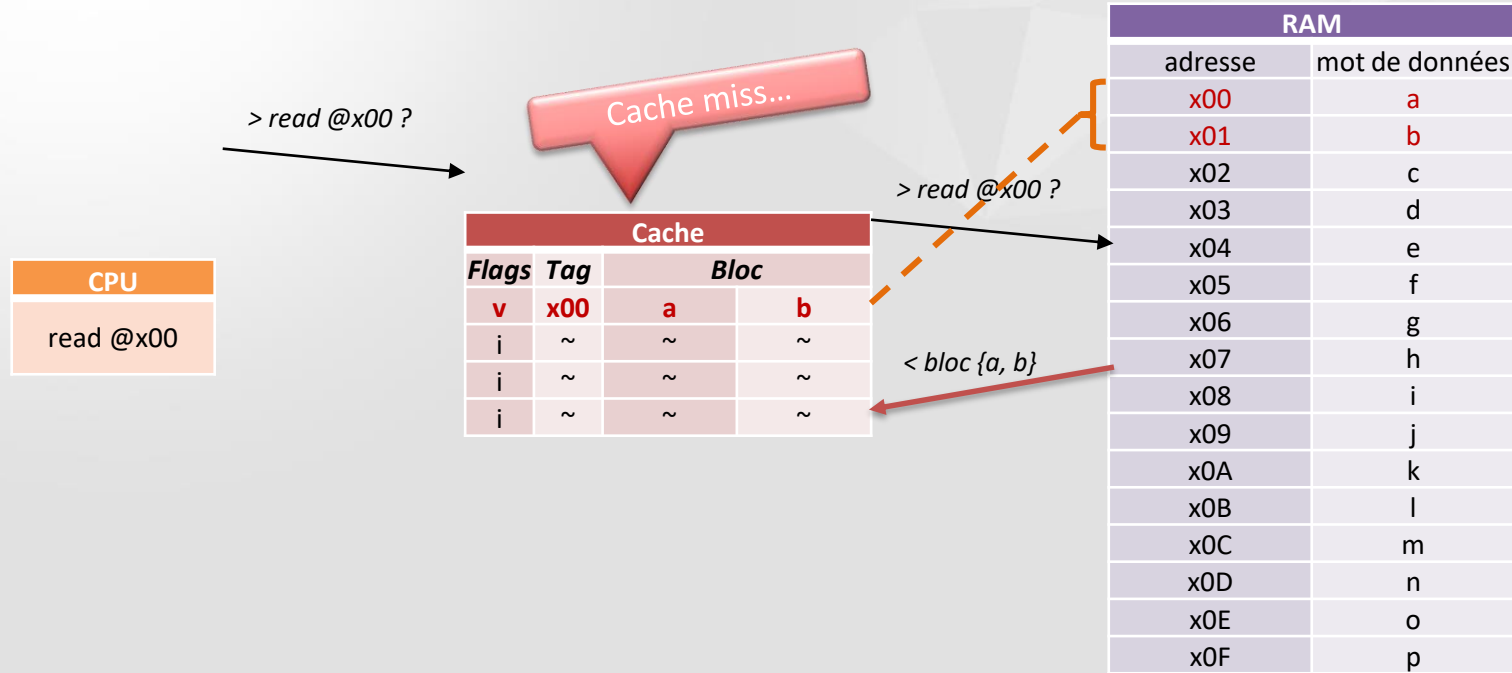
Cache			
Flags	Tag	Bloc	
i	~	~	~
i	~	~	~
i	~	~	~
i	~	~	~

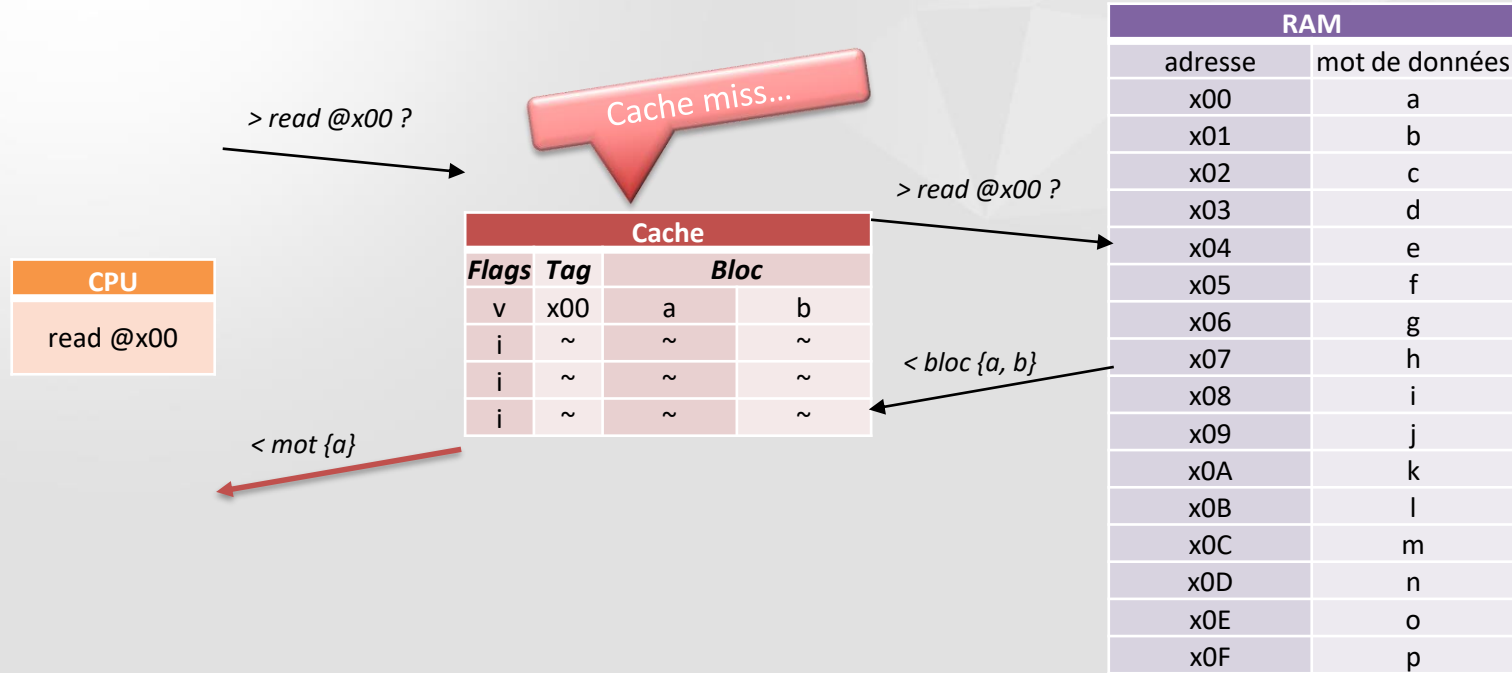
RAM	
adresse	mot de données
x00	a
x01	b
x02	c
x03	d
x04	e
x05	f
x06	g
x07	h
x08	i
x09	j
x0A	k
x0B	l
x0C	m
x0D	n
x0E	o
x0F	p

Séance 2

Fonctionnement d'un cache



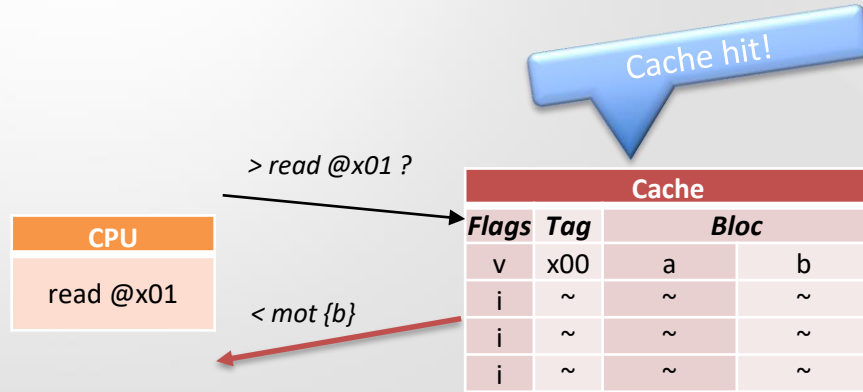




► Lecture / cache hit



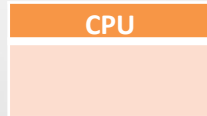
RAM	
adresse	mot de données
x00	a
x01	b
x02	c
x03	d
x04	e
x05	f
x06	g
x07	h
x08	i
x09	j
x0A	k
x0B	l
x0C	m
x0D	n
x0E	o
x0F	p



RAM	
adresse	mot de données
x00	a
x01	b
x02	c
x03	d
x04	e
x05	f
x06	g
x07	h
x08	i
x09	j
x0A	k
x0B	l
x0C	m
x0D	n
x0E	o
x0F	p

Séance 2

Fonctionnement d'un cache



Cache hit!

Cache			
Flags	Tag	Bloc	
v	x00	a	b
i	~	~	~
i	~	~	~
i	~	~	~

RAM	
adresse	mot de données
x00	a
x01	b
x02	c
x03	d
x04	e
x05	f
x06	g
x07	h
x08	i
x09	j
x0A	k
x0B	l
x0C	m
x0D	n
x0E	o
x0F	p

► Associativité

- *Direct-mapped*
 - 1 bloc de données ira toujours dans la même ligne de cache; **pas de choix**.
(associativité = 1, chaque ligne est un groupe)
- *Set-associative*
 - 1 bloc a le choix entre toutes les lignes (voies) d'un groupe (*set*)
 - Nb de voies = degré d'associativité
- *Fully-associative*
 - 1 bloc peut aller dans n'importe quelle ligne du cache
(cache = 1 groupe, associativité = nb de lignes de cache)

► Politique de remplacement

Quand on a le choix (~~direct-mapped~~):

- une ligne vide
- ou on remplacer une ligne déjà pleine...

Exemples de politiques:

- *Random*: n'importe laquelle
- FIFO (*first-in, first-out*): la ligne chargée depuis le plus longtemps
- LRU (*Least Recently Used*): on remplace la ligne restée inutilisée depuis longtemps

Remplacement de ligne

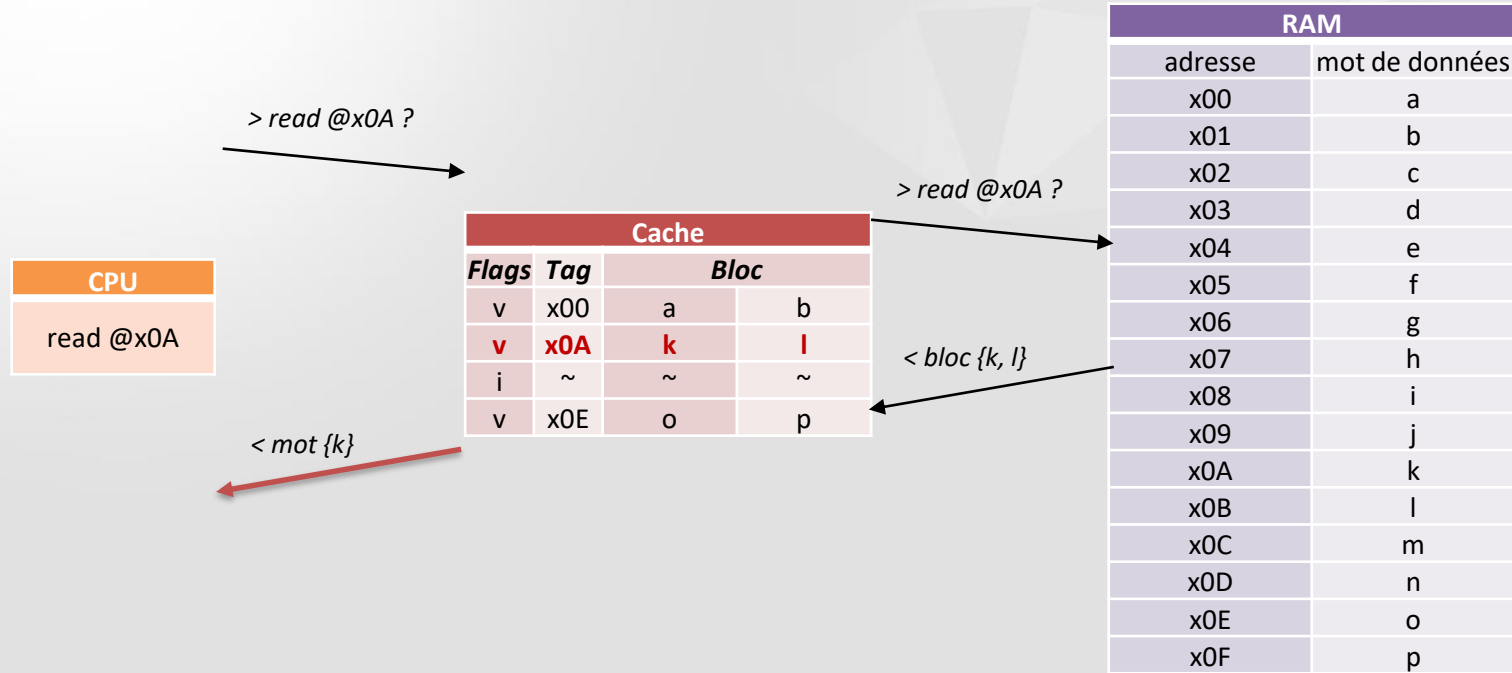
Cache *direct mapped*

> read @x0A ?

CPU
read @x0A

Cache			
Flags	Tag	Bloc	
v	x00	a	b
v	x02	c	d
i	~	~	~
v	x0E	o	p

RAM	
adresse	mot de données
x00	a
x01	b
x02	c
x03	d
x04	e
x05	f
x06	g
x07	h
x08	i
x09	j
x0A	k
x0B	l
x0C	m
x0D	n
x0E	o
x0F	p



Remplacement de ligne

Cache 2-way set associative, LRU

> read @x0A ?



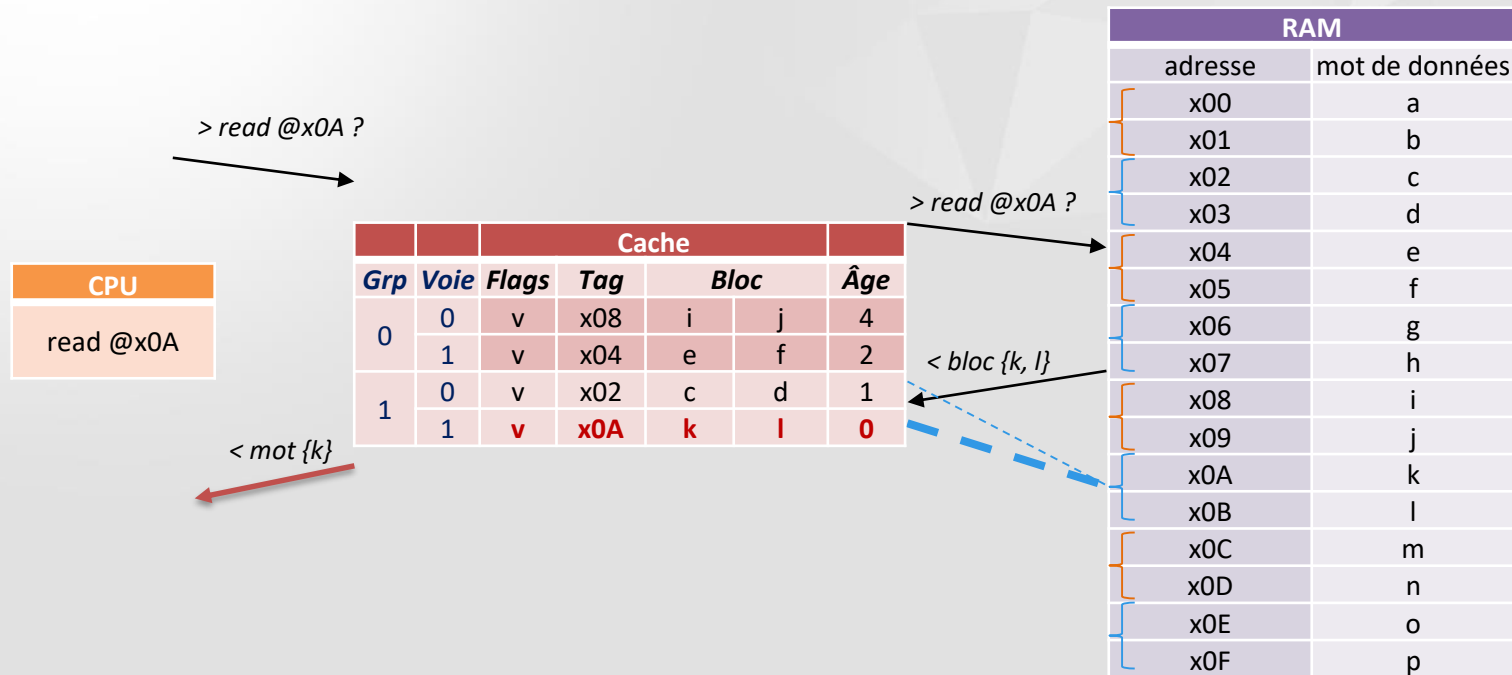
CPU
read @x0A

Cache						
Grp	Voie	Flags	Tag	Bloc		Âge
0	0	v	x08	i	j	3
	1	v	x04	e	f	1
1	0	v	x02	c	d	0
	1	v	x06	g	h	2

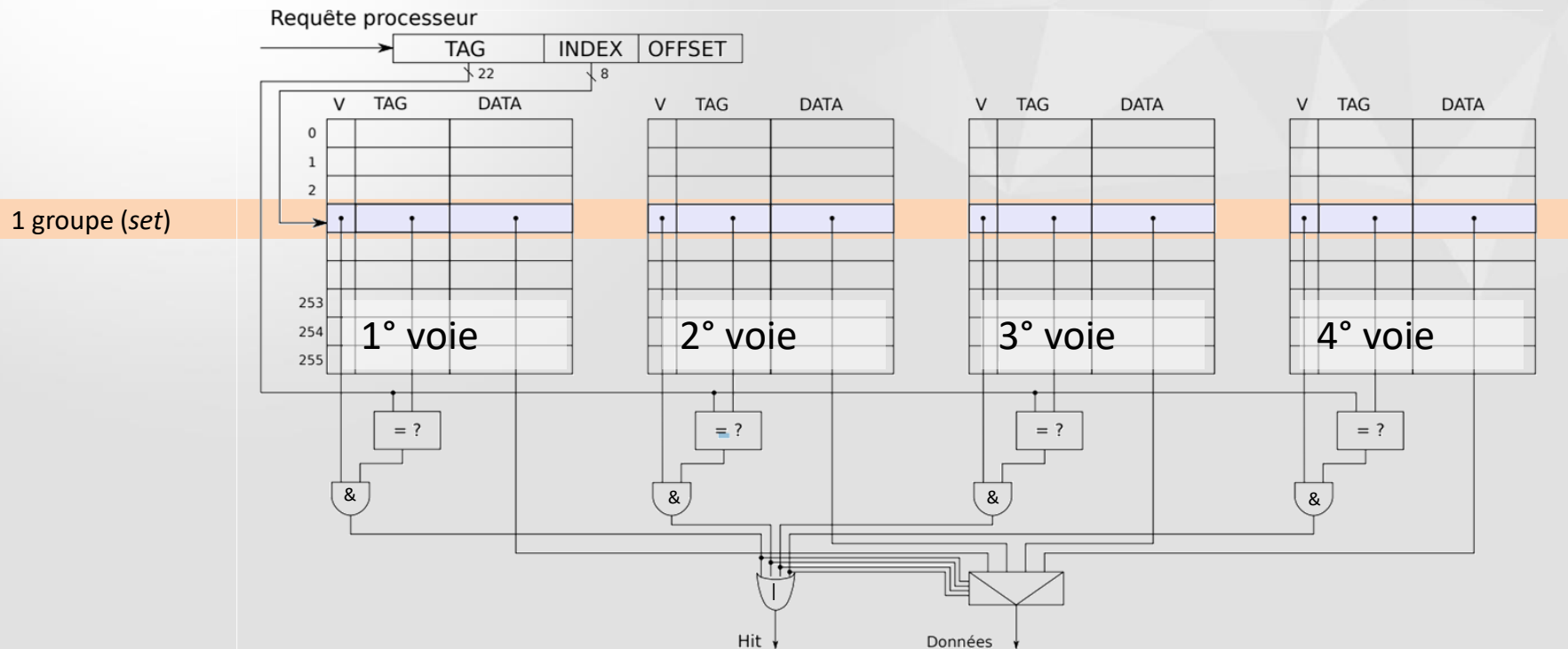
RAM	
adresse	mot de données
x00	a
x01	b
x02	c
x03	d
x04	e
x05	f
x06	g
x07	h
x08	i
x09	j
x0A	k
x0B	l
x0C	m
x0D	n
x0E	o
x0F	p



Associativité double
(2 voies par groupe),
2 groupes (sets).



Matériellement



Remplacement de ligne

Cache *fully associative*, LRU

CPU
read @x0A

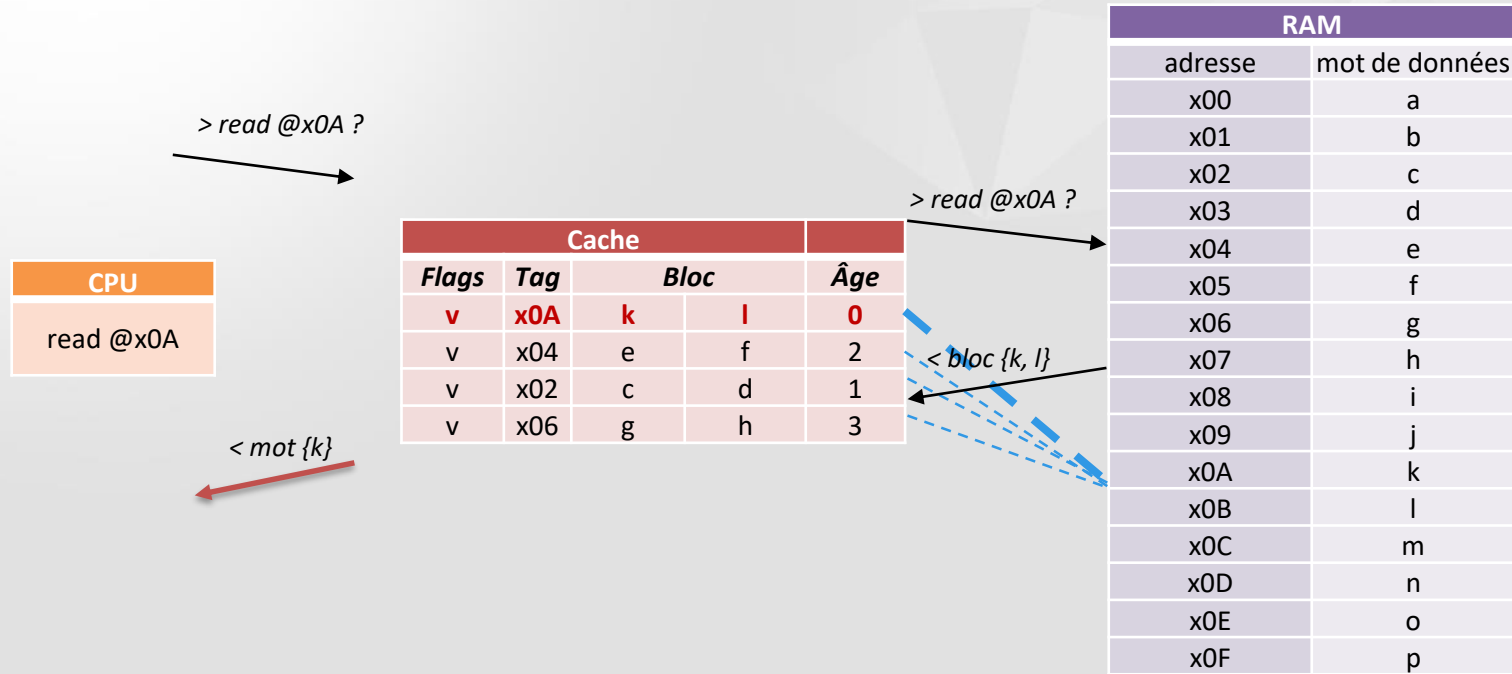
Cache				
Flags	Tag	Bloc		Âge
v	x08	i	j	3
v	x04	e	f	1
v	x02	c	d	0
v	x06	g	h	2

RAM	
adresse	mot de données
x00	a
x01	b
x02	c
x03	d
x04	e
x05	f
x06	g
x07	h
x08	i
x09	j
x0A	k
x0B	l
x0C	m
x0D	n
x0E	o
x0F	p



Séance 2

Fonctionnement d'un cache



► Politiques d'écriture

► *Write hit*: écriture sur une donnée présente en cache

- Ecriture RAM immédiate: *Write-through*
 - Dès qu'on écrit dans le cache, on propage vers la RAM
...mais beaucoup d'écritures inutiles sur le bus RAM...
- Ou bien écriture RAM différée: *Write-back*
 - On n'écrit que dans le cache; on mettra la RAM à jour lors de l'éviction de la ligne

► *Write miss*: écriture sur une donnée non cachée

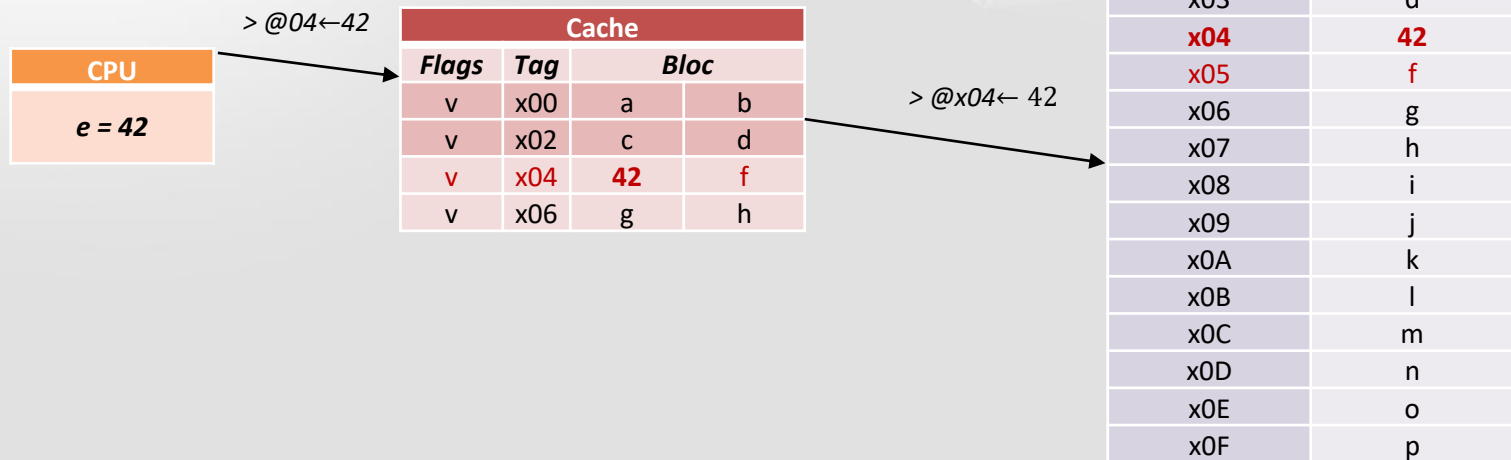
- Ecriture avec allocation: *Write-allocate*
 - On charge la ligne, et on la modifie (voir ci-dessus)
- Ecriture sans allocation: *Write-non-allocate*
 - On écrit dans la RAM, sans charger la ligne en cache

► Exemples

- CPU modernes: souvent write-back & write-allocate

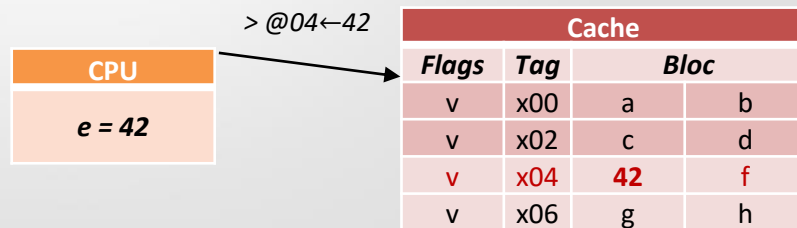
Politiques d'écriture

- Ecriture immédiate *write-through*



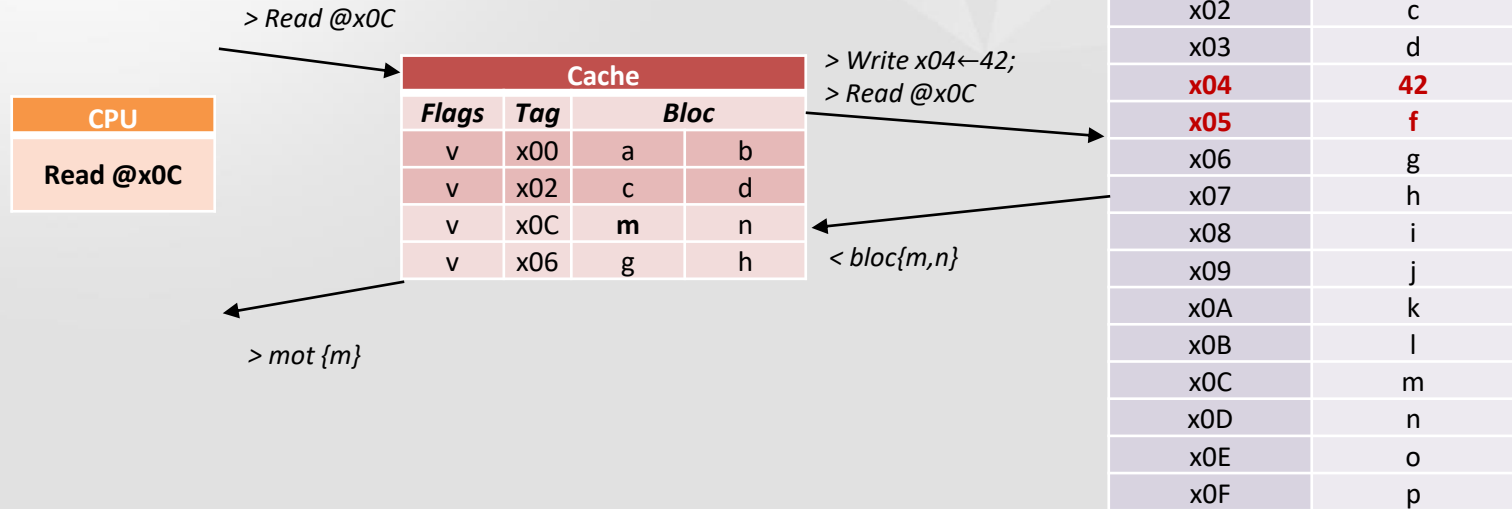
Politiques d'écriture

- Ecriture différée *write-back*



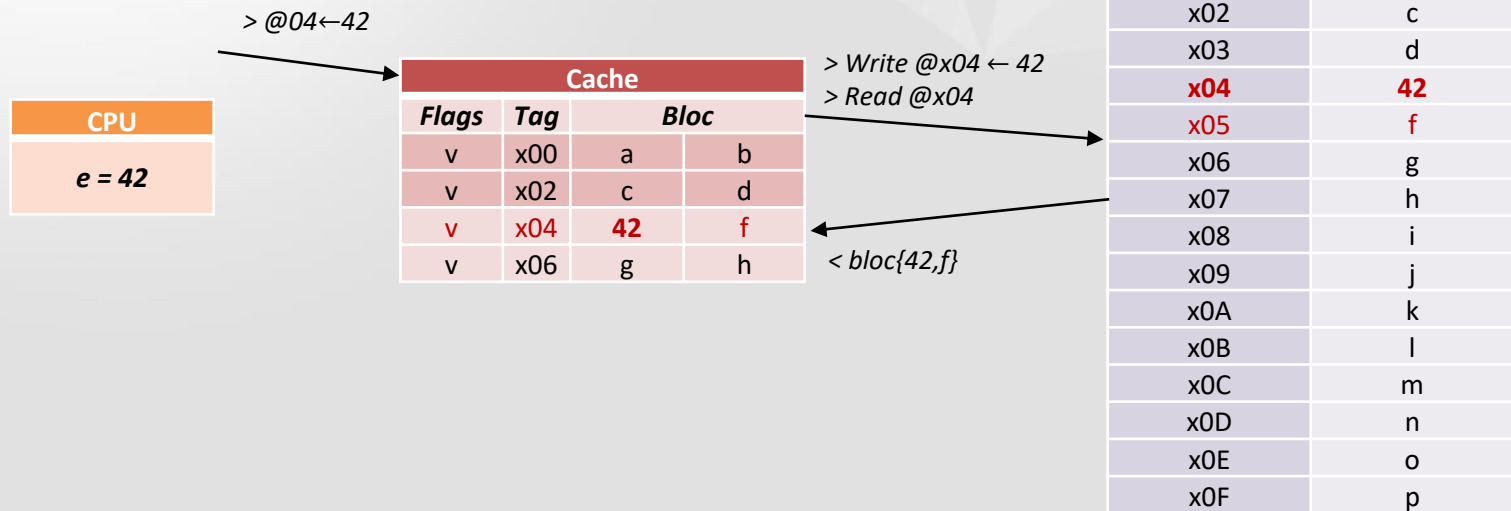
RAM	
adresse	mot de données
x00	a
x01	b
x02	c
x03	d
x04	e
x05	f
x06	g
x07	h
x08	i
x09	j
x0A	k
x0B	l
x0C	m
x0D	n
x0E	o
x0F	p

Quelques cycles plus tard:
au moment d'évincer la ligne, le bloc
est écrit en RAM



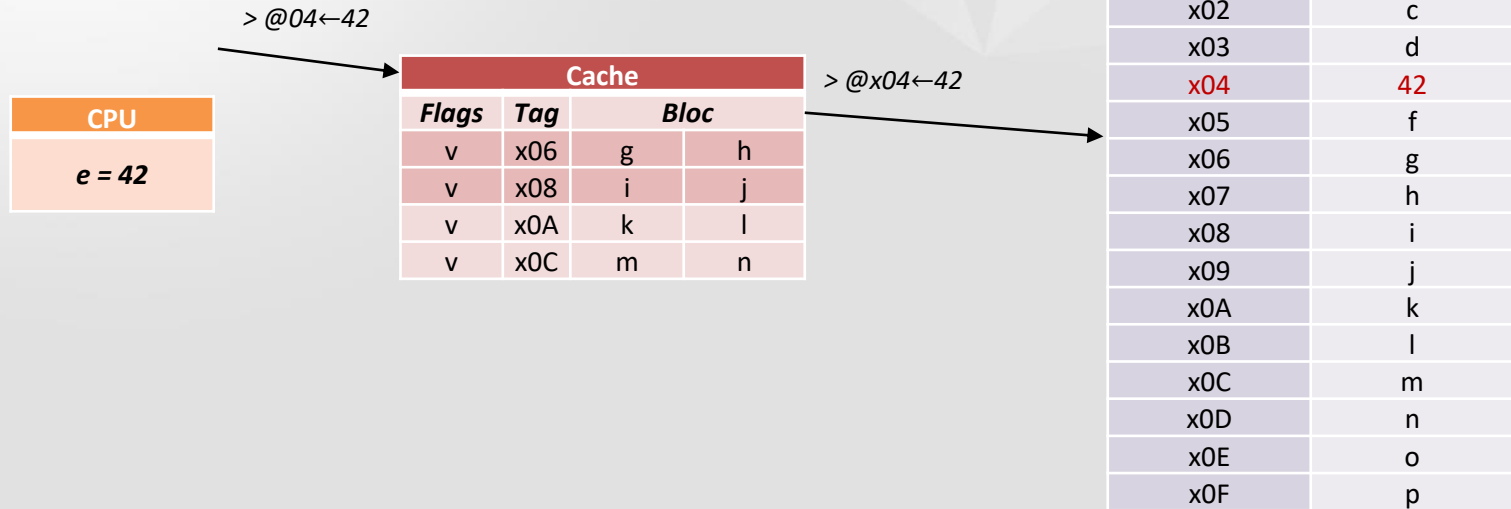
Write allocate

- Bloc n'est pas présent initialement dans le cache, alloué lors de l'écriture



Write non-allocate

- Bloc n'est pas présent initialement dans le cache, n'est pas alloué
- Forcément *write-through*



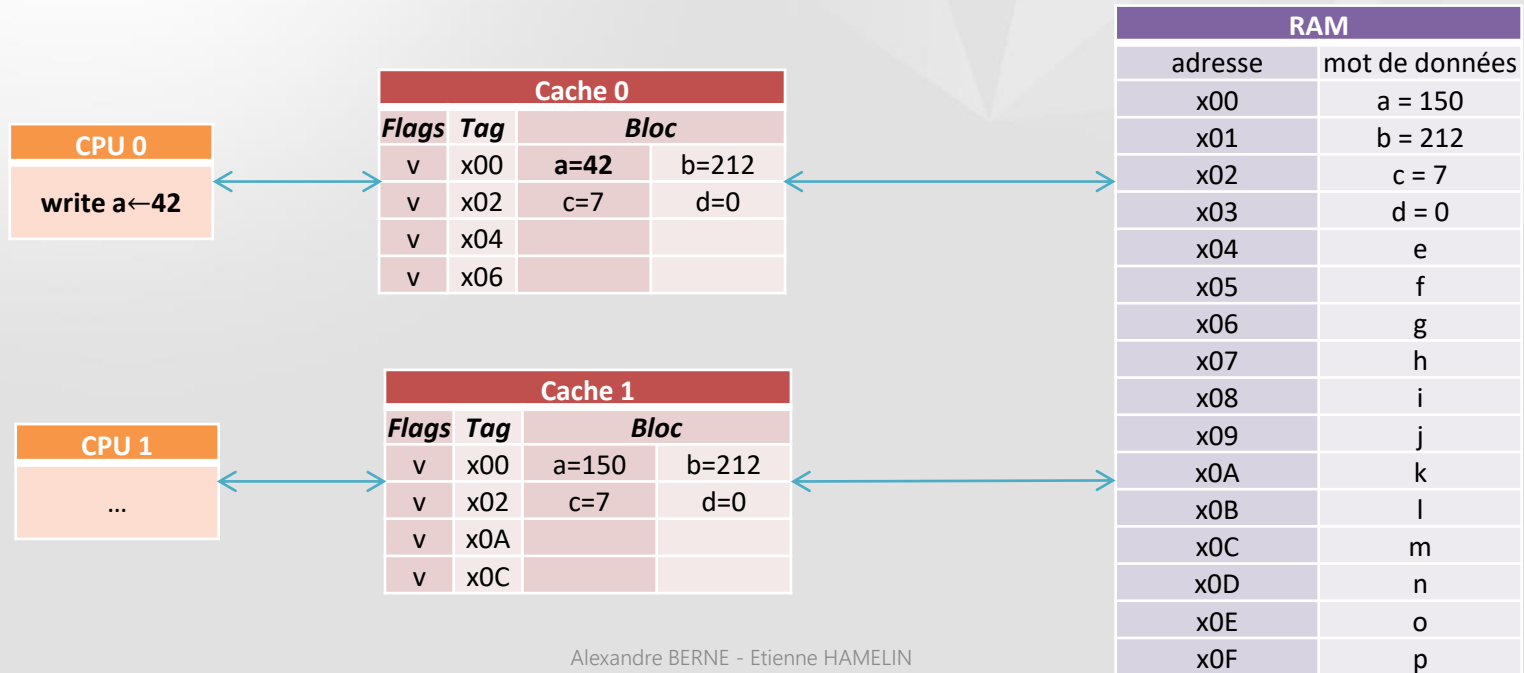
- ▶ 4 sources de défaut de cache (*cache miss*): les 4C
- ▶ *Compulsory miss*
 - Le premier accès à un bloc: inévitable !
- ▶ *Conflict miss*
 - utilisations multiples du même groupe (*set*), dépassant l'associativité
 - ⚠ alignement des données...
- ▶ *Capacity miss*
 - jeu de donnée dépasse la taille du cache \Rightarrow défaut quelle que soit l'associativité
- ▶ *Coherency miss*
 - Conflit avec d'autres processeurs: on va voir tout de suite



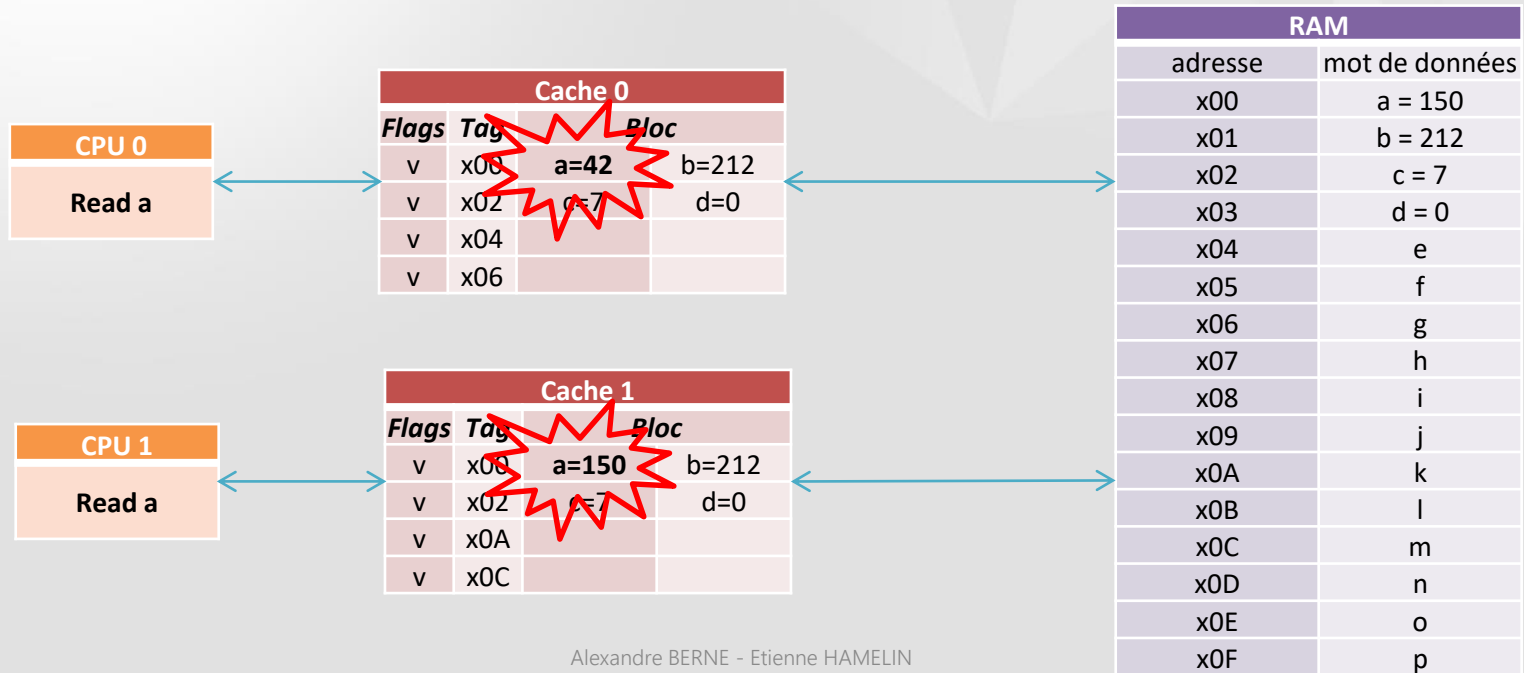
► Hiérarchie mémoire

Caches des circuits multicoeur

► Et en multicoeur maintenant!



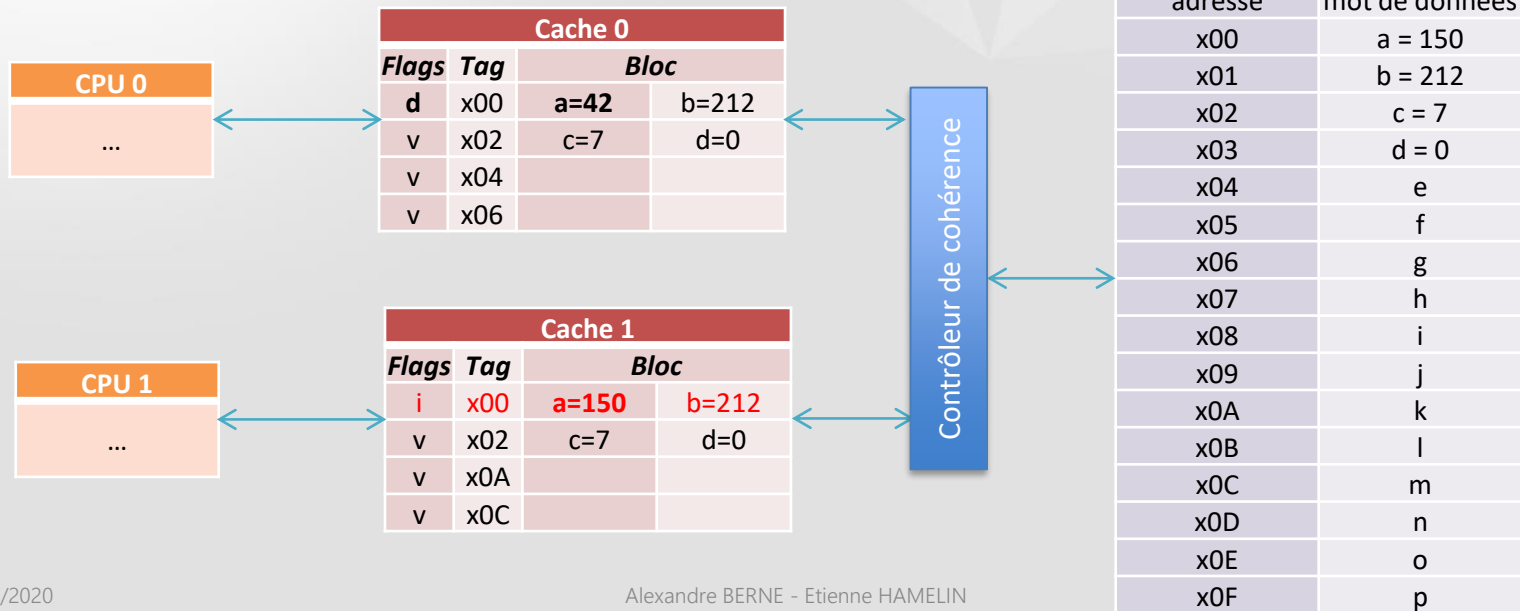
► Problème de cohérence



► Protocole *write-invalidate*

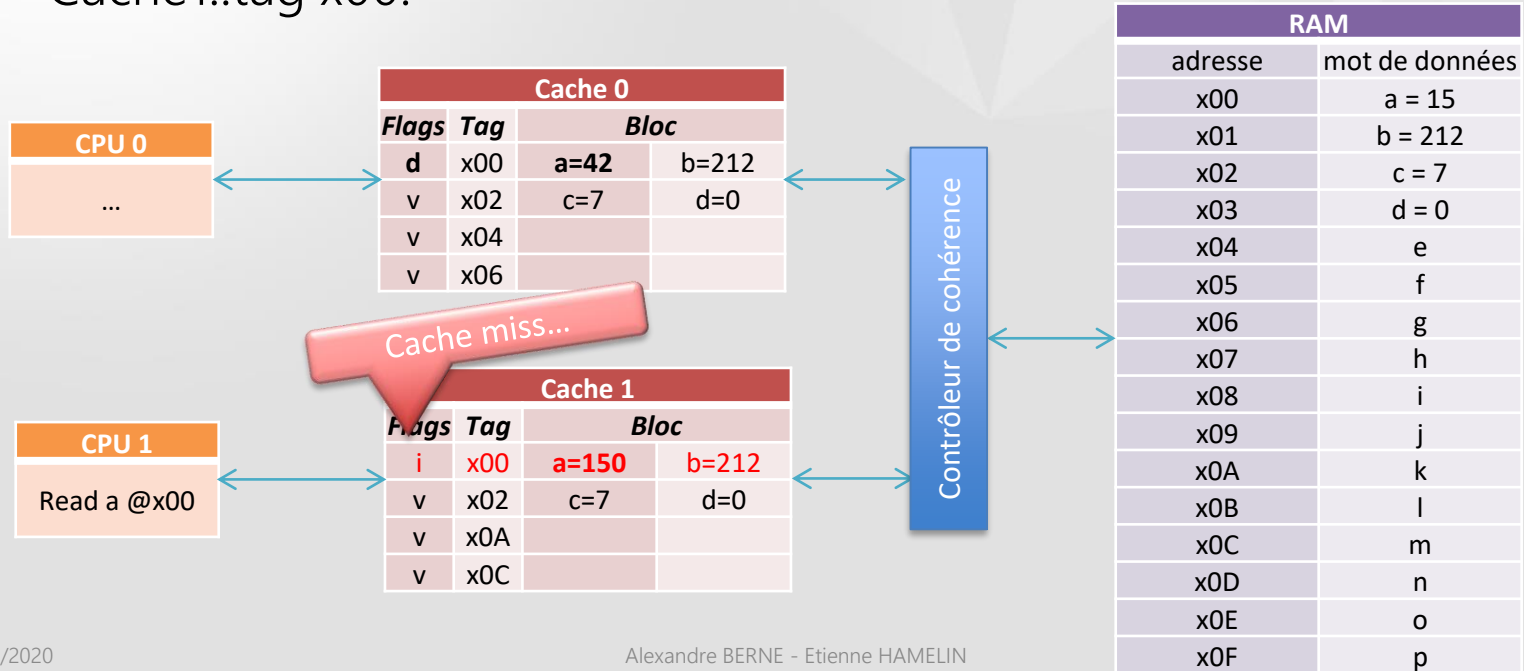
Ligne Cache0::tag x00 modifiée → flag *dirty*

Contrôleur de cohérence propage l'info, **invalide** la liche tag x00 de Cache1.




Ligne Cache0::tag x00 modifiée, on ajoute le flag *dirty*

Le contrôleur de cohérence propage l'info, et invalide la liche Cache1::tag x00.

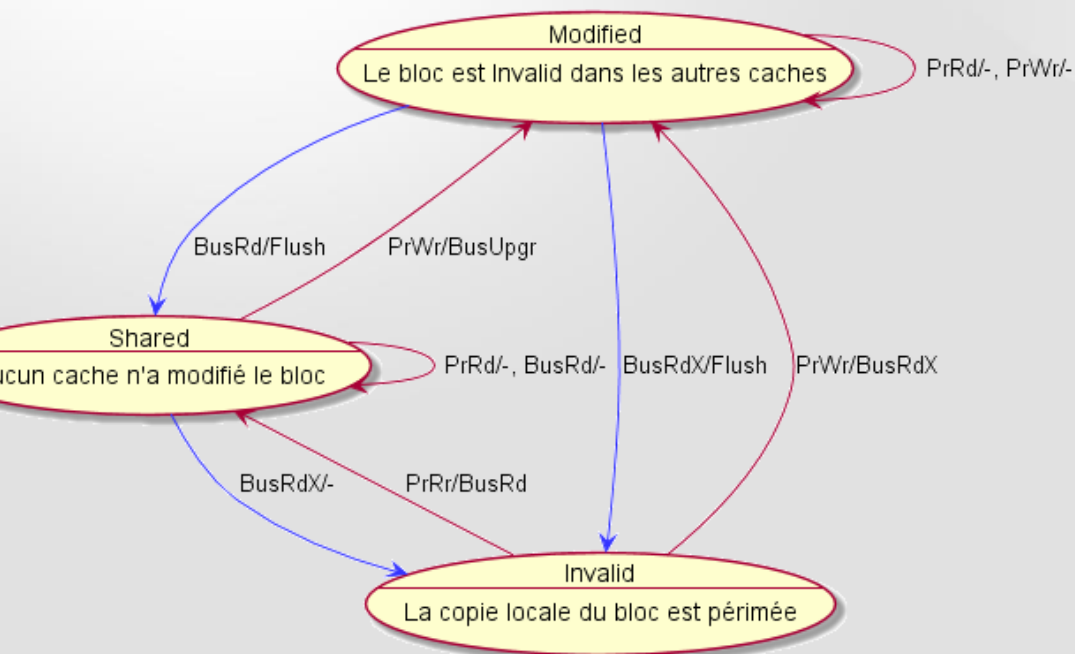


► Protocole MSI : un bloc de cache peut être dans l'état:

- 
- *Shared (v)*
 - le bloc est cohérent avec la RAM;
 - d'autres caches peuvent éventuellement avoir une copie
 - *Modified (d)*
 - le bloc a été modifié par le processeur local
 - la RAM et les autres caches sont périmés
 - *Invalid (i)*
 - le bloc a été modifié dans un autre cache; la copie locale est invalide.

► Variantes

- MESI: *Modified, Exclusive, Shared, Invalid*
- MOESI: *Modified, Owned, Exclusive, Shared, Invalid*



Note:

PrRd: lecture par le processeur propriétaire du cache

PrWr: écriture par le processeur

BusRd: demande sur le bus de lecture par un processeur

BusRdX: demande sur le bus de lecture exclusive pour écriture par un processeur

BusUpgr: demande de mise à jour sur le bus

Flush: écriture du bloc sur le bus

Propriétés essentielles:

► Cohérence

- À un même instant, 2 CPUs ne doivent pas considérer valides des valeurs différentes du même bloc

► Consistance

- Lectures & écritures concurrentes sur *plusieurs blocs*
- Consistance stricte: le cache doit faire « comme si » les écritures étaient instantanées. Coûteux...
- Consistance séquentielle: pour chaque processeur, l'ordre perçu est correct
- Outils: memory barriers

P0

```
flag0 = 1;  
if (flag1 == 0) {  
    /* section critique */  
}  
flag0 = 0;
```

P1

```
flag1 = 1;  
if (flag0 == 0) {  
    /* section critique */  
}  
flag1 = 0;
```

Algorithme de Dekker

P0

```
flag1 = 1;  
turn = 0;  
if (flag0 == 0 && turn == 0) {  
    /* section critique */  
}  
flag1 = 0;
```

P1

```
flag0 = 1;  
turn = 1;  
if (flag1 == 0 && turn == 1) {  
    /* section critique */  
}  
flag0 = 0;
```

► Le récap'

- Associativité: *direct mapped, set/fully associative*
- Politiques de remplacement: LRU, FIFO, random, ...
- Politiques d'écriture: *write hit (write-through, write-back), write miss (write-allocate, write-non-allocate)*
- Les sources de conflits (4C)