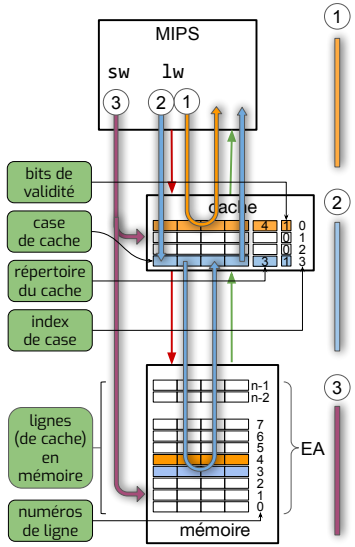
ALMO

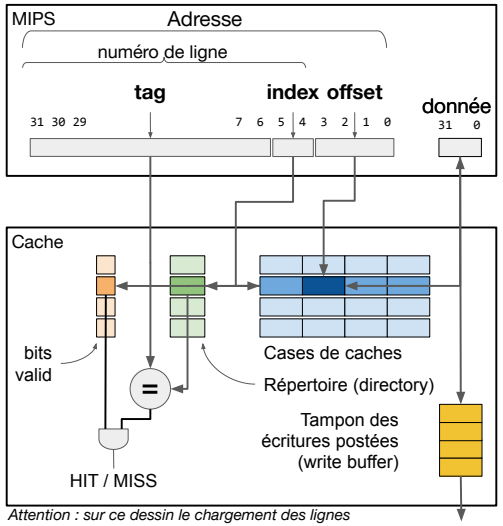
Caches (suite)

Principe de fonctionnement du cache



1. Lecture d'une donnée déjà présente dans le cache
   1. Le MIPS exécute une instruction `lw`
   2. Le cache détermine le numéro de ligne où se trouve la donnée à partir de son adresse
   3. La ligne est déjà dans le cache (
   4. La donnée est lue dans le cache et rendue au MIPS
2. Lecture d'une donnée absente du cache
   3. La ligne est absente du cache
   4. Le MIPS est gelé
   5. La ligne est demandée à la mémoire
   6. La ligne est rangée dans le cache
   7. La donnée est lue dans le cache et rendue au MIPS
3. Écriture d'une donnée (politique ***write through***)
   1. Le MIPS exécute une instruction `sw`
   2. Le cache détermine le numéro de ligne où se trouve la donnée à partir de son adresse
   3. Si la ligne est déjà dans le cache, elle est mise à jour
   4. La donnée est envoyée vers la mémoire

Schéma du cache à correspondance directe



Attention : sur ce dessin le chargement des lignes depuis la mémoire n'est pas représenté

Il y a trois tableaux dans le cache :

1. Les cases de cache (en bleu).
   Ces cases contiennent les lignes de cache.
2. Le répertoire (en vert)
   Ce répertoire contient autant d'entrée que de cases dans le cache (il y a une entrée par case). Chaque entrée contient le numéro de la ligne (ou juste le tag) contenu dans la case du cache correspondante.
3. Les bits de validités (en orange clair)
   Ce répertoire contient autant d'entrée que de cases dans le cache (il y a une entrée par case). Lorsqu'une entrée est à 1, c'est que la case correspondante contient une ligne, sinon elle n'en contient pas.

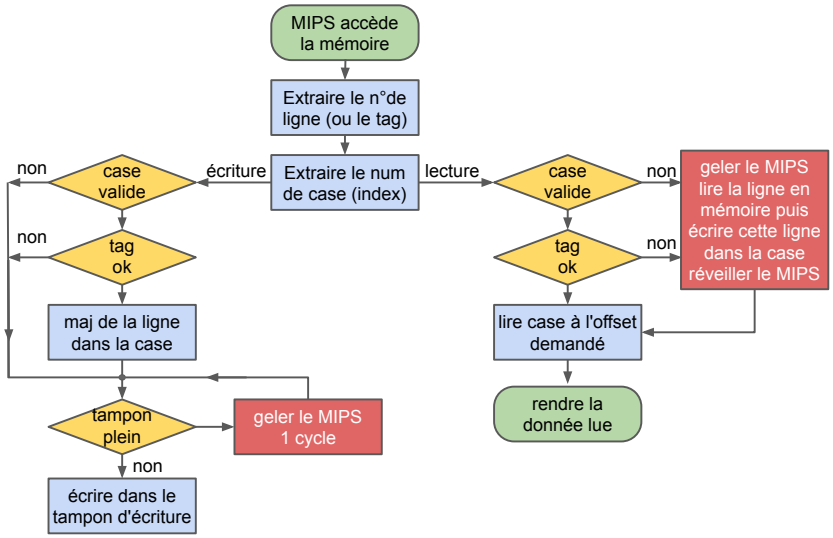
Et aussi un tampon d'écriture (en orange)
Ce tampon contient les écritures (adresses et données) en attente d'être envoyées en mémoire

Vocabulaire

- Ligne de cache
  - Une ligne de cache est un segment d'adresses de 2^n octets dans l'espace d'adressage (8, 16, 32, 64 ou 128 octets, cela dépend du cache).
    C'est la quantité minimale de données ou d'instructions lues par un cache.
- Case de cache
  - Un cache contient des copies de lignes de cache venant de la mémoire.
  - Ces lignes de cache sont rangées dans des cases de cache.
  - Une ligne de cache est un contenu, alors qu'une case de cache est un contenant
- Correspondance directe
  - Le numéro de case (index) utilisé pour ranger une ligne est déterminé en prenant les k bits de poids faible du numéro de ligne, il y a donc 2^k cases dans le cache.
- Répertoire
  - Le cache range dans un répertoire les numéros de ligne de cache qu'il possède.
  - Il y a autant d'entrée dans le répertoire qu'il y a de cases dans le cache.
  - Pour un cache à correspondance directe, on peut sauver seulement le tag de ligne
  - Le tag de ligne est égal au numéro de ligne divisé par l'index (shift right de k bits)
- Bit de validité
  - Au début, les cases de cache sont toutes vides.
  - Elles se remplissent avec des lignes de cache au fur et à mesure des lectures.
  - Il y a un bit de validité par case de cache indiquant si la case contient une ligne.

il y a une explication détaillée juste après

Algorithme de comportement du cache



Dans le répertoire : tag ou numéro de ligne ?

- Le cache stocke des copies de lignes de cache dans ses cases.
- Pour chaque case contenant une copie de ligne, le cache doit savoir quel est le numéro de ligne de la copie, c'est-à-dire d'où elle vient dans l'espace d'adressage.
- Dans un cache à correspondance directe, ce sont les bits de poids faible du numéro de ligne qui servent à désigner les numéros de cases. Ces bits définissent l'index. Ce choix impose que le nombre de cases soit une puissance de 2.
- Toutes les lignes qui ont un index 0 vont dans la case 0, toutes les lignes qui ont un index 1 vont dans la case, etc.
- En conséquence, il n'est pas nécessaire de stocker les bits d'index dans le répertoire puisqu'on les connaît déjà, c'est le numéro de la case. Autrement dit, la case X du répertoire, contient une ligne dont les derniers bits contiennent la valeur X.
- En résumé:
  Ce qu'on doit enregistrer dans le répertoire, c'est le numéro de ligne,
  MAIS, dans un cache à correspondance directe, on peut ne stocker que le tag,
  et ainsi, économiser de la mémoire dans le répertoire.
- Pour un cache de 16ko et des lignes de 64o ⇒ 16ko/64 = 256 cases ⇒ index de 8bits
  On économise 256o, c'est beaucoup, car le répertoire est une mémoire rapide.

Évincement de ligne

- Lors d'un MISS, le cache doit aller chercher la ligne manquante dans la mémoire pour la copier dans une case du cache
- Dans le cas d'un cache à correspondance directe, le numéro de la case choisie est imposé (ce sont les n-bits de poids faible du numéro de ligne)
- Dans un cache N-associatif ou full-associatif, le cache doit choisir une case avec soin. Il doit éviter de choisir une case contenant une ligne très utilisée.
- Pour savoir si une ligne est très utilisée, il faut mettre des compteurs d'usage dans le répertoire, permettant d'exécuter des algorithmes de type *Last Recently Used* (LRU). Nous n'allons pas détailler ce type d'algorithme.
- La case choisie par l'algorithme LRU est nommée "case victime" et la ligne présente dans cette case est dite "évincée".
- On a un "évincement de ligne" à chaque fois qu'une ligne est retirée du cache.

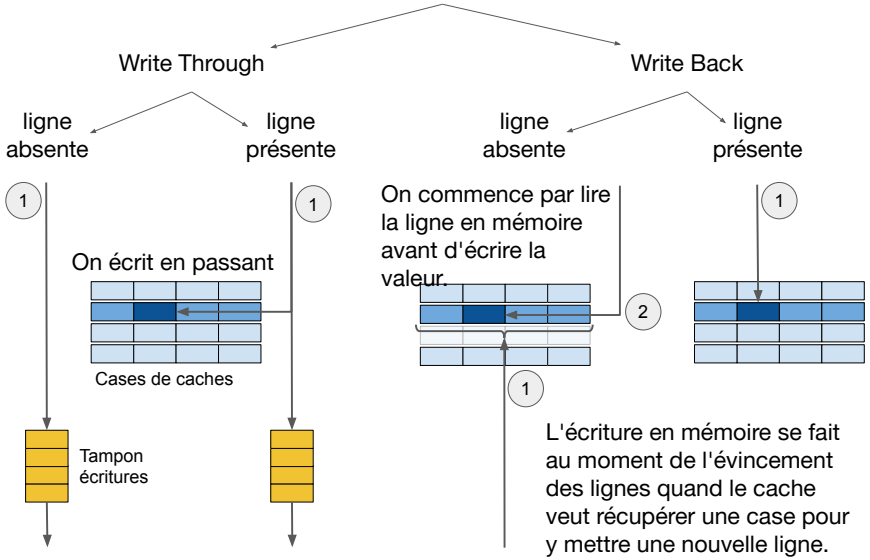
Comportement du cache pour les écritures

Il existe deux types de comportement pour les écritures

1. Cache Write-Through (WT)
   Les écritures se font toujours dans la mémoire,
   et si la ligne est déjà présente dans le cache, elle est mise à jour.
   Ainsi la mémoire contient toujours la copie la plus à jour des données.
2. Cache Write-Back (WB)
   Lorsqu'on écrit dans une ligne, si elle est absente du cache, elle est chargée depuis la mémoire, puis les écritures se font dans le cache.
   Ainsi, c'est le cache qui contient la copie la plus à jour des données.
   Les lignes modifiées sont copiées en mémoire au moment de leur évincement, uniquement si elles ont été écrites.

Les caches WB réduisent beaucoup le nombre d'écriture sur le bus,
mais ils sont plus complexes, car il y a plus d'états à gérer pour chaque ligne.

Write Through vs Write Back



Mesure de la performance d'un processeur

- La performance d'un processeur se mesure en Cycles Par Instruction : `CPI`
  C'est à dire le nombre de cycles nécessaires à l'exécution d'une instruction.
- Dans un système mémoire parfait, la mémoire répond toujours à tous les accès en lecture et en écriture pour les instructions et les données.
- Le `CPI` dans un système mémoire parfait, nous le nommons `CPI0`.
- Le MIPS32 a une architecture à exécution pipelinée.
  Il est capable de lire une nouvelle instruction à chaque cycle,
- Pour autant, le `CPI0` n'est pas égal à 1 car il y a des dépendances entre les instructions obligeant à introduire des bulles dans le pipeline (une bulle est un cycle où on ne lit pas une nouvelle instruction).
- Le `CPI0` va dépendre du programme et des caractéristiques du MIPS (vous verrez ça en TME). Typiquement, `CPI0 = 1.2`
- Le `CPI` réel va être augmenté à cause des MISS de cache qui gèle le MIPS.
  $\Rightarrow CPI = CPI0 + \Delta CPI_{ins} + \Delta CPI_{data}$

Calcul du CPI réel

- L'augmentation du CPI (Δ`CPI`) est due à deux choses
  1. Le taux de MISS
     c'est-à-dire le pourcentage de requête d'accès à la mémoire qui provoque un MISS (pour les instructions et les données).
     Ce taux dépend de la taille du cache et du programme

     $$\text{Taux\_de\_miss} = \frac{\text{nombre de requêtes avec MISS}}{\text{nombre de requêtes totales}}$$

     Typiquement le taux_de_miss_instruction < 2% (voire très < 2%)
     le taux_de_miss_data < 5%
  2. Le coût du MISS
     Ce coût dépend de l'architecture, du bus, du nombre de MIPS, de la taille des autres niveaux de cache, de la longueur des lignes, etc.

     Typiquement coût du MISS = 10 à 100 cycles.
- L'augmentation du CPI (Δ`CPI`) est simplement le produit du taux par le coût
  `ΔCPI = Taux_de_miss * Cout_du_miss`

Influence des écritures sur le CPI

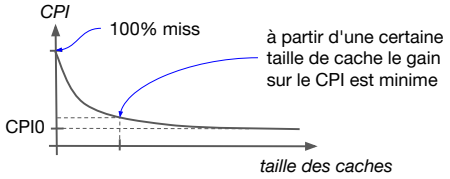
- En première approximation, les écritures n'ajoutent rien au `CPI`,
  à condition que les deux conditions suivantes soient remplies :
  1. La durée moyenne entre deux instructions d'écritures dans le programme est supérieure à la durée d'une écriture en mémoire
  2. Le tampon d'écriture contient assez de cases pour absorber les rafales d'écriture
- Nous supposons que ces conditions sont remplies et que les écritures ne gèlent pas le MIPS.
  En d'autres termes, quand le MIPS exécute une instruction d'écriture en mémoire, il y a toujours de la place dans le tampon des écritures

Influence de la taille des caches sur le CPI

- Nous avons vu que le CPI dépend du taux de MISS et de son coût.
- On suppose, ici, que les écritures n'influencent pas le ΔCPI.
  - $CPI = CPI0 + \Delta CPI_{ins} + \Delta CPI_{data}$
  - $\Delta CPI = Taux\_de\_miss * Cout\_du\_miss$
- Le taux de MISS
  - dépend de la taille du cache
    ⇒ plus il est grand plus ce taux baisse
  - du comportement de l'application
    ⇒ plus l'application exploite la localité spatiale ou temporelle, plus le taux baisse. L'idéal, ce sont des boucles sur des tableaux…

Ce graphe représente la forme de la courbe du CPI en fonction de la taille des caches.

La vraie courbe dépend de tous les caches et de l'application, c'est juste l'idée qui nous importe



Exemple

On considère un cache de données de premier niveau write-through à correspondance directe, d'une capacité totale de 8Kio. La ligne de cache a une largeur de 32 octets (8 mots de 32 bits). Les adresses sont sur 32 bits.

Le but de l'exercice est d'analyser le remplissage de ce cache de données L1, puis d'estimer le nombre de cycles nécessaires à l'exécution d'un programme. On suppose que le cache de données est initialement vide. On considère la fonction suivante :

On suppose que le tableau X est à l'adresse 0x10010000, et qu'il est passé à la fonction f.

```
int32_t X[512][8];
 ...
void f(int32_t X[256][8]) {
      register int32_t i, j;
      for (j = 7; j >= 0; j -= 1) {
          for (i = 0; i < 256; i += 1) {
              X[i][j] = X[i][j] / 2;
          }
      }
 }
```

1. *Donner le nombre de bits des champs offset, index et étiquette d'une adresse.*
2. *Donner, dans l'ordre, les 3 premiers éléments de X lus par la fonction f.*
3. *Quels sont les éléments de X qui peuvent occuper les mots du cache suivants :*
   — mot 0 de case d'index 0 ;
   — mot 3 de la case d'index 19.
4. *Représenter la(les) case(s) valide(s) du cache après 3 itérations (1 case contient 1 ligne), en précisant les indexes (seules 3 cases sont représentées), et en mettant dans les parties « Mot x » les éléments de X présents (p. ex : X[12][3]).*

Exemple

1. *Donner le nombre de bits des champs offset, index et étiquette d'une adresse*

   offset :
   index :
   tag :

2. *Donner, dans l'ordre, les 3 premiers éléments de X lus par la fonction f.*

Exemple

1. *Donner le nombre de bits des champs offset, index et étiquette d'une adresse*

   offset : log2(32) = 5
   index : log2(8*1024/32) = 8
   tag : 32 - 8 -5 = 19

2. *Donner, dans l'ordre, les 3 premiers éléments de X lus par la fonction f.*

Exemple

1. *Donner le nombre de bits des champs offset, index et étiquette d'une adresse*

   offset : log2(32) = 5
   index : log2(8*1024/32) = 8
   tag : 32 - 8 -5 = 19

2. *Donner, dans l'ordre, les 3 premiers éléments de X lus par la fonction f.*

   A chaque tour de boucle on lit un mot d'une nouvelle ligne.
   X[0][7], X[1][7], X[2][7]
   Ça ne semble pas optimal.

Exemple

3. *Quels sont les éléments de X qui peuvent occuper les mots du cache suivants : mot 0 de case d'index 0 ; et mot 3 de la case d'index 19.*

   X est à l'adresse 0x10010000
   0b0001.0000.0000.0001.0000.0000.0000.0000.
   On voit que X est alignée sur une ligne (offset == 0)
   On voit que la première ligne de X sera rangée dans la case 0 (index == 0)
   La première ligne de cache du tableau contient :
   X[0][0] X[0][1] … X[0][7] ça fait 8 mots
   La dernière case de cache contiendra donc (si cette ligne est lue)
   X[255][0] X[255][1] … X[255][7]
   Le cache ne peut pas contenir tout le tableau X
   ⇒ la case du tableau X[256][0] sera aussi rangée dans la case 0.

   ⇒ mot 0 de case 0 : X[0][0] et X[256][0]
   mot 3 de la case 19 : X[19][3] et X[19+256][3]

Exemple

3. *Quels sont les éléments de X qui peuvent occuper les mots du cache suivants : mot 0 de case d'index 0 ; et mot 3 de la case d'index 19.*

Exemple

4. *Représenter la(les) case(s) valide(s) du cache après 3 itérations (1 case contient 1 ligne), en précisant les indexes (seules 3 cases sont représentées), et en mettant dans les parties « Mot x » les éléments de X présents (p. ex : X[12][3]).*

Index	Adresse du Mot 0	Mot 7	Mot 6	Mot 5	Mot 4	Mot 3	Mot 2	Mot 1	Mot 0
0									
1									
2									

Exemple

4. *Représenter la(les) case(s) valide(s) du cache après 3 itérations (1 case contient 1 ligne), en précisant les indexes (seules 3 cases sont représentées), et en mettant dans les parties « Mot x » les éléments de X présents (p. ex : X[12][3]).*

Index	Adresse du Mot 0	Mot 7	Mot 6	Mot 5	Mot 4	Mot 3	Mot 2	Mot 1	Mot 0
0	0x10010000	**X[0][7] /2**	X[0][6]	X[0][5]	X[0][4]	X[0][3]	X[0][2]	X[0][1]	X[0][0]
1									
2									

On lit X[0][7] mais le cache va chercher la ligne entière contenant cette donnée

Exemple

5. *Calculer, en le justifiant, le nombre de miss de données rencontrées lors de l'exécution de cette fonction.*

6. *Donner le taux de miss sur le cache de données pour cette fonction.*

7. *La compilation de ce code a produit une boucle de 9 instructions pour la boucle interne (qui met donc 9 cycles à s'exécuter en présence d'un système mémoire parfait `CPI0=1`), et 3 en plus pour la boucle externe. On néglige l'effet des miss sur le cache d'instructions. Si un miss de données coûte 17 cycles, calculer le nombre de cycles total, puis calculer le nombre moyen de cycles par élément.*

Exemple

4. *Représenter la(les) case(s) valide(s) du cache après 3 itérations (1 case contient 1 ligne), en précisant les indexes (seules 3 cases sont représentées), et en mettant dans les parties « Mot x » les éléments de X présents (p. ex : X[12][3]).*

Index	Adresse du Mot 0	Mot 7	Mot 6	Mot 5	Mot 4	Mot 3	Mot 2	Mot 1	Mot 0
0	0x10010000	**X[0][7] /2**	X[0][6]	X[0][5]	X[0][4]	X[0][3]	X[0][2]	X[0][1]	X[0][0]
1	0x10010020	**X[1][7] /2**	X[1][6]	X[1][5]	X[1][4]	X[1][3]	X[1][2]	X[1][1]	X[1][0]
2	0x10010040	**X[2][7] /2**	X[2][6]	X[2][5]	X[2][4]	X[2][3]	X[2][2]	X[2][1]	X[2][0]

Exemple

5. *Calculer, en le justifiant, le nombre de miss de données rencontrées lors de l'exécution de cette fonction.*

```
Le tableau est parcouru "à l'envers" mais on ne lit que la moitié
du tableau, il n'y a pas de collision de lignes dans les cases.
Il y a un miss par ligne ⇒ 256 MISS.
Si on avait lu tous le tableau ⇒ 512 * 8 MISS.
```

6. *Donner le taux de miss sur le cache de données pour cette fonction.*

7. *La compilation de ce code a produit une boucle de 9 instructions pour la boucle interne (qui met donc 9 cycles à s'exécuter en présence d'un système mémoire parfait `CPI0=1`), et 3 en plus pour la boucle externe. On néglige l'effet des miss sur le cache d'instructions. Si un miss de données coûte 17 cycles, calculer le nombre de cycles total, puis calculer le nombre moyen de cycles par élément.*

Exemple

5. *Calculer, en le justifiant, le nombre de miss de données rencontrées lors de l'exécution de cette fonction.*

```
Le tableau est parcouru "à l'envers" mais on ne lit que la moitié
du tableau, il n'y a pas de collision de lignes dans les cases.
Il y a un miss par ligne ⇒ 256 MISS.
Si on avait lu tous le tableau ⇒ 512 * 8 MISS.
```

6. *Donner le taux de miss sur le cache de données pour cette fonction.*

```
256 * 8 lectures  = 2048
256 miss                    ⇒ taux de MISS = 256/2048 = 12.5%
```

7. *La compilation de ce code a produit une boucle de 9 instructions pour la boucle interne (qui met donc 9 cycles à s'exécuter en présence d'un système mémoire parfait `CPI0=1`), et 3 en plus pour la boucle externe. On néglige l'effet des miss sur le cache d'instructions. Si un miss de données coûte 17 cycles, calculer le nombre de cycles total, puis calculer le nombre moyen de cycles par élément.*

Problème de cohérence

Définition de la Cohérence

Deux choses (idées, ondes, objets) sont cohérentes,
si elles évoluent exactement de la même manière.

En informatique (wikipédia)

La cohérence est la capacité pour un système à refléter sur la copie d'une donnée les modifications intervenues sur d'autre copies de cette donnée.

Pour les caches

- Les caches contiennent des copies de lignes de cache,
- S'il y a plusieurs cœurs MIPS dans un processeur chacun a ses caches.
- Une même ligne de cache peut alors être copiées dans plusieurs caches.
- Si l'un des MIPS modifie sa copie, alors il y a une perte de cohérence entre les copies de la ligne de cache.
- Si rien n'est fait, certains cache peuvent contenir des copies anciennes de ligne de cache.

Exemple

5. *Calculer, en le justifiant, le nombre de miss de données rencontrées lors de l'exécution de cette fonction.*

```
Le tableau est parcouru "à l'envers" mais on ne lit que la moitié
du tableau, il n'y a pas de collision de lignes dans les cases.
Il y a un miss par ligne ⇒ 256 MISS.
Si on avait lu tous le tableau ⇒ 512 * 8 MISS.
```

6. *Donner le taux de miss sur le cache de données pour cette fonction.*

```
256 * 8 lectures  = 2048
256 miss                    ⇒ taux de MISS = 256/2048 = 12.5%
```

7. *La compilation de ce code a produit une boucle de 9 instructions pour la boucle interne (qui met donc 9 cycles à s'exécuter en présence d'un système mémoire parfait `CPI0=1`), et 3 en plus pour la boucle externe. On néglige l'effet des miss sur le cache d'instructions. Si un miss de données coûte 17 cycles, calculer le nombre de cycles total, puis calculer le nombre moyen de cycles par élément.*

```
Nombre de cycles total = 256 * 8 * 9 + 8 * 3 + 256 * 17 = 22808
Il y a 2048 éléments,
soit une moyenne de 22808/2048 = 11.14 cyc/élé
```


Solution de principe logicielle

Dans le cas d'une solution logicielle, c'est le programme qui connaît les données partagées entre plusieurs coeurs et qui évite les situations problématiques

- Les problèmes de cohérence n'arrivent que si une ligne est copiée dans plusieurs caches ou si une donnée change en mémoire toute seule (si c'est un périphérique qui est responsable de ce changement).
- Le programme connaît les lignes qui sont touchées par ce problème et il peut décider de lui même de retirer ces lignes de son cache
  ⇒ il demande le *flush* total ou partiel.

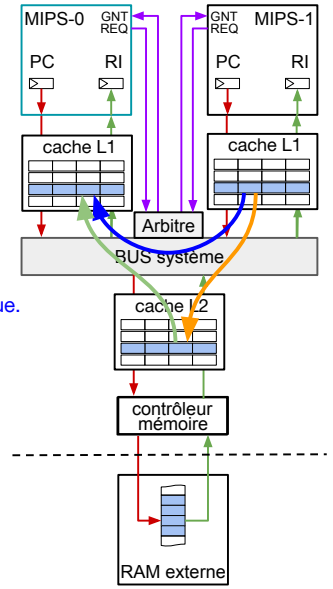
C'est ce que nous utiliserons sur notre plateforme.
Il faudra demander au système GIET de "*flusher*" le cache DATA

Solution de principe matérielle

Ce sont les caches eux-même qui détectent les incohérences dues aux copies multiples.

Hypothèse, c'est un cache L1 Write Through

- snooping based (Espionnage)
  - Toutes les écritures sont envoyées sur le bus vers la mémoire.
  - Les caches L1 peuvent *snooper (espionner)* le bus et s'ils voient des écritures dans des lignes dont ils ont une copie, il la mette à jour. flèche bleue.
- directory based (basé sur un répertoire des copies)
  - C'est cache L2 donne les copies de lignes aux caches L1. Il peut se souvenir pour chaque ligne dans quel cache L1, elle est copiée.
  - C'est lui qui est responsable des mises à jour.
  - C'est plus long et il y a une perte temporaire de cohérence, mais cela n'oblige pas les caches L1 de faire de l'espionnage, parfois c'est impossible.



En résumé, nous avons vu :

- Un rappel de la différence entre ligne de cache et case de cache
- Un rappel de la fonction du répertoire de cache
- Un rappel du schéma d'un cache à correspondance directe
- L'algorithme de comportement du cache en écriture et en lecture
- Pourquoi on peut ne stocker que le tag dans le répertoire
- Ce qu'est un évincement de ligne dans un cache
- La différence de principe entre caches Write-Through et Write-Back
- La définition de la performance d'un processeur (vitesse)
- L'influence de la taille des caches sur la performance
- Que les écritures en mémoire peuvent ne pas influencer la performance
- La définition de la cohérence de cache
- Les principes de solutions pour permettre la cohérence
- Un exemple d'exercice sur les caches

Prochaine séance : plongée au cœur du GIET

```
GIET
├── app
│   ├── stdio.c      : lib C minimaliste
│   └── stdio.h
└── sys
    ├── giet.s           : entrée noyau
    ├── sys_handler.c    : gest. syscalls
    ├── irq_handler.c    : gest. IRQ
    ├── exc_handler.c    : gest. exception
    ├── ctx_handler.c    : gest. tâches
    ├── common.c         : lib C kernel
    ├── drivers.c        : pilotes
    ├── common.h
    ├── hwr_mapping.h
    ├── sys_handler.h
    ├── irq_handler.h
    ├── exc_handler.h
    ├── ctx_handler.h
    └── drivers.h
```

architecture logicielle