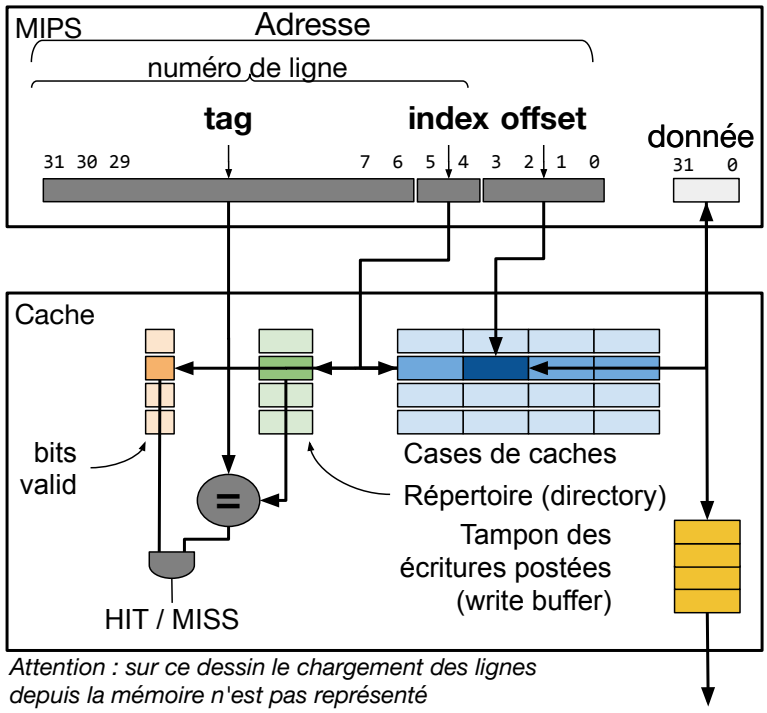
Cache L1 - performance

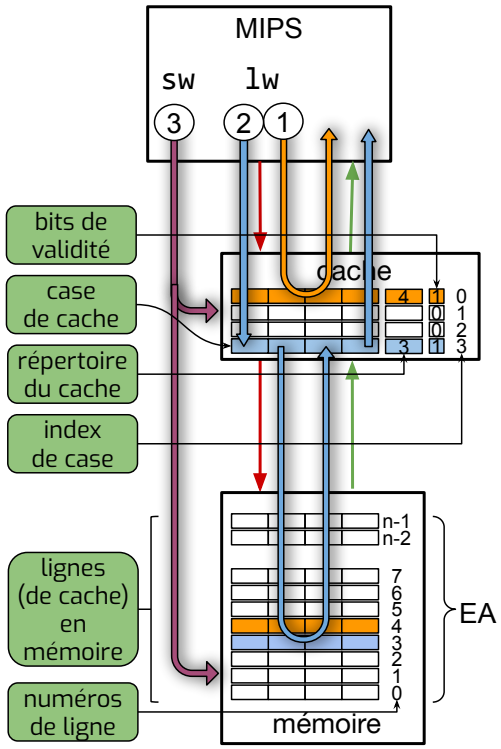
LU3IN031 Architecture des ordinateurs - 2
Matériel et Logiciel
B4

franck.wajsburt@lip6.fr
V4

Schémas Cache L1 - Direct Mapped*

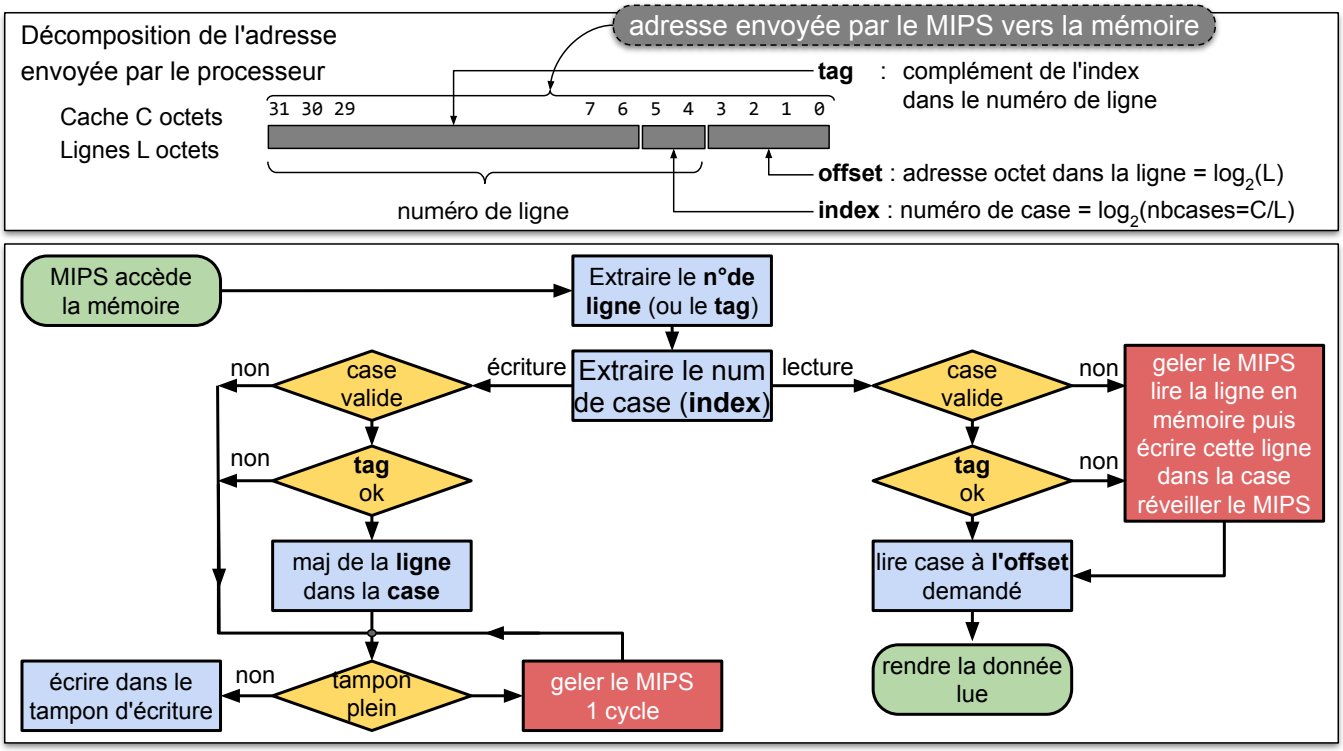


Attention : sur ce dessin le chargement des lignes depuis la mémoire n'est pas représenté



** à correspondance directe entre le numéro de ligne et le numéro de case*

Comportement du cache



Comportement du cache pour les écritures

Il existe deux types de comportement pour les écritures

1. Cache Write-Through (WT) *(c'est celui que nous utilisons)*

   Les écritures se font toujours dans la mémoire,
   et si la ligne est déjà présente dans le cache, alors elle est mise à jour.

   Ainsi la mémoire contient toujours la copie la plus à jour des données.

2. Cache Write-Back (WB)

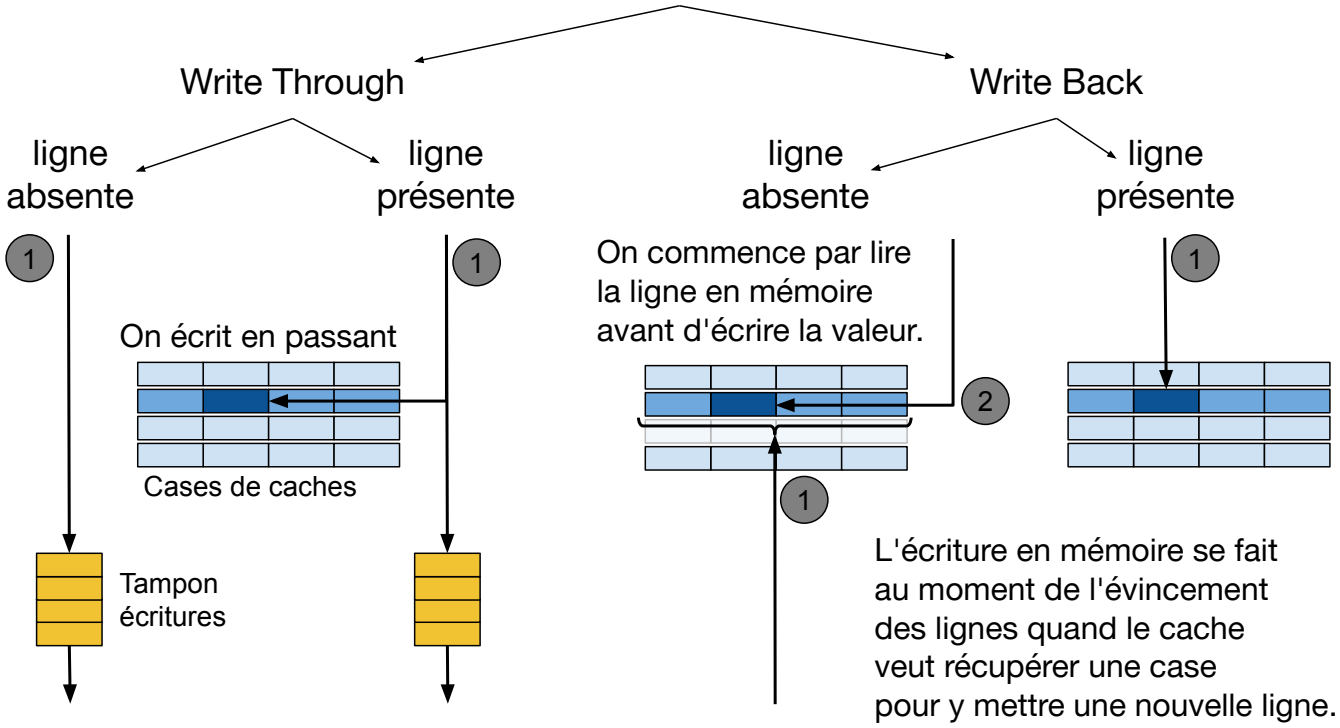
   Lorsqu'on écrit dans une ligne, si elle est absente du cache, elle est chargée depuis la mémoire, puis les écritures se font dans le cache.

   Ainsi, c'est le cache qui contient la copie la plus à jour des données.

   Les lignes modifiées sont copiées en mémoire uniquement au moment de leur évincement et uniquement si elles ont été modifiées (on dit *dirty*).

Les caches WB réduisent beaucoup le nombre d'écritures 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



Objectifs de la séance

Les caches de niveau L1 sont indispensables si les instructions et les données sont à plusieurs cycles du processeur, mais combien de temps est-ce qu'on gagne ? Et, s'il y a plusieurs processeurs et donc plusieurs caches, est-ce que ça cause des problèmes ?

- Gain en performance du MIPS dû au cache L1
- Exemple : exercice guidé
- Problème liés à la cohérence des caches

Gain en performance

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 (HIT à 100%), nous le nommons `CPI0`.
- Le MIPS32 de la plateforme a une architecture à exécution pipelinée.
  Il est donc 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
  Le `CPI` réel va être augmenté à cause des MISS de cache qui gèle le MIPS.

  → $CPI = CPI0 + \Delta CPI_{ins} + \Delta CPI_{data}$

Estimation du CPI0

Toutes les instructions n'ont pas la même durée, car elles dépendent des instructions voisines, il y a en effet des dépendances entre les instructions .

Par exemple, pour le MIPS (cela dépend aussi de l'implémentation)

- un `lw $rt, imm($rs)` dure 1 ou 2 cycles en fonction de l'usage de `$rt` dès l'instruction suivante ou non : `1` cycle si `$rt` n'est pas utilisé immédiatement, `2` cycles sinon.
  Le compilateur fait ce qu'il peut pour éviter ce cas, supposons qu'il y arrive 3 fois sur 4.
- une rupture de séquence dure `2` cycles quelle qu'elle soit

Il faut donc connaître le pourcentage moyen de chaque type d'instructions (hypothèse)

- Opérations entre registres (`add`, `or`, `lui`, etc.) `50%`
- Branchements (`bne`, `jal`, `jr`, etc.) `20%`
- Lecture de données (`lw`, `lh`, `lbu`, etc.) `20%`
- Écriture de données (`sw`, `sh`, `sb`, etc.) `10%`

La durée moyenne du `CPI0` est simplement une somme pondérée des durées des types par les %

→ `CPI0 = 1*50% + 2*20% + (1*75% + 2*25%) * 20% + 1*10% = 1.25 cycles`

Calcul du CPI réel

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

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

     Typiquement `taux_de_miss_instruction` `< 2% (voire très < 2%)`
     `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 `cout_du_MISS = 10 à 100 cycles`
- L'augmentation du CPI (`ΔCPI`) est simplement le produit du taux par le coût

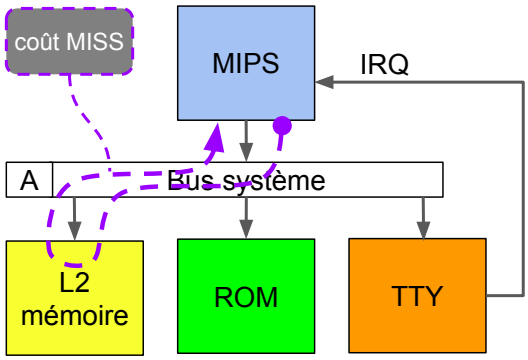
  `ΔCPI = Taux_de_miss * Cout_du_miss`

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

Estimation du coût des MISS

Quand le cache ne dispose pas de la donnée ou de l'instruction demandée, il doit aller la chercher dans la mémoire et il doit geler le processeur pendant ce temps.



1. il doit demander le bus à l'arbitre sauf s'il est le seul initiateur, mais c'est rare et envoyer la requête.
   → supposons 3 cycles
2. Le cache L2 doit aller chercher la ligne. C'est très variable, si la ligne demandée par le L1 est dans le L2, alors quelques cycles suffisent, si la ligne n'est pas dans le L2, p.e. 200 cycles ! Il faut déterminer un coût moyen, qui dépend de la taille du L2 et du type de programme
   → supposons 25 cycles
3. Il faut déplacer la ligne entre le cache L2 et le cache L1, cela dépend de la longueur de la ligne et du débit du bus système. Si le bus peut transférer 1 mot/c et que la ligne fait 4 mots,
   → supposons 4 cycles au mieux.
4. Enfin, il faut mettre à jour le cache L1 et redémarrer le MIPS
   → supposons 3 cycles

→ Avec ces hypothèses : `cout_du_miss = 3 + 25 + 4 + 3 = 35 cycles`

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 + ΔCPI`$_{ins}$ `+ ΔCPI`$_{data}$
  - `ΔCPI = Taux_de_miss * Cout_du_miss` (pour chaque cache `ins` et `data`)
- 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…

→ Avec les hypothèses précédentes

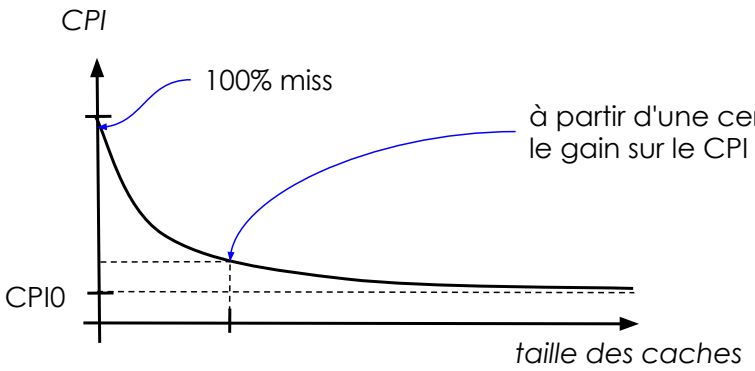
```
CPI = 1.25 + 35*2% + 35*20%*5% = 2.3 cycles
```


Influence de la taille des caches sur le CPI

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, vous allez pouvoir le constater en TP en changeant le nombre de cases des caches. Comprenez que c'est normalement impossible de changer la taille des caches parce que c'est du matériel et que le matériel ne peut être "créé" par un programme, mais ici, il suffira de changer la valeur d'un argument du simulateur du SoC 😊



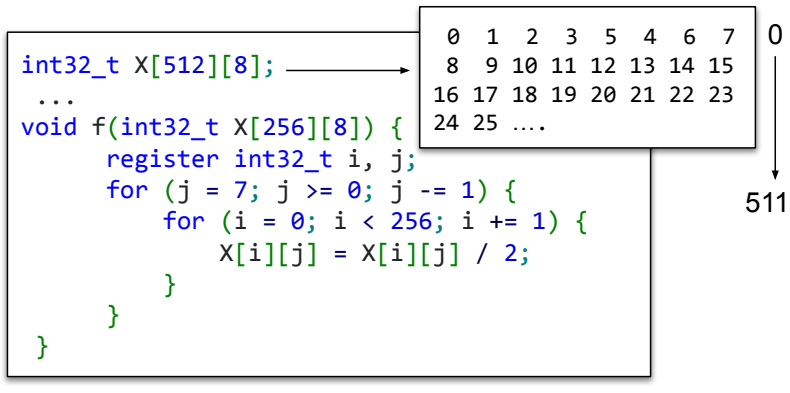
Exemple

Etude du remplissage et du gain en perf

On considère un cache de données de premier niveau write-through à correspondance directe, d'une capacité totale de 8 kibi. 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.



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]).*

Remplissage du cache

Cache L1 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 font 32 bits.

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

   offset :

   index :

   tag :

```
int32_t X[512][8]
```

```
 0  1  2  3  5  4  6  7
 8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 ….
```

2. *Donner, dans l'ordre, les 3 premiers éléments de X lus par 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;
          }
      }
 }
```


Exemple

Cache L1 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 font 32 bits.

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

```
int32_t X[512][8]
```

```
 0  1  2  3  5  4  6  7
 8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 ….
```

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.

```
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;
          }
      }
 }
```

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.*

Il y a plusieurs méthodes pour répondre à cette question
La plus simple, c'est de regarder dans quelle cases du cache et dans quel mot de cette ligne est rangé X[0][0]
Pour cela, il suffit de regarder à quelle adresse est placé X[][]
Ensuite, on peut déduire des choses puisqu'on sait comment X occupe l'espace

```
X est à l'adresse  0x10010000
```

```
int32_t X[512][8]
```

```
 0  1  2  3  5  4  6  7
 8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 ....
```


Remplissage du cache

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]
```

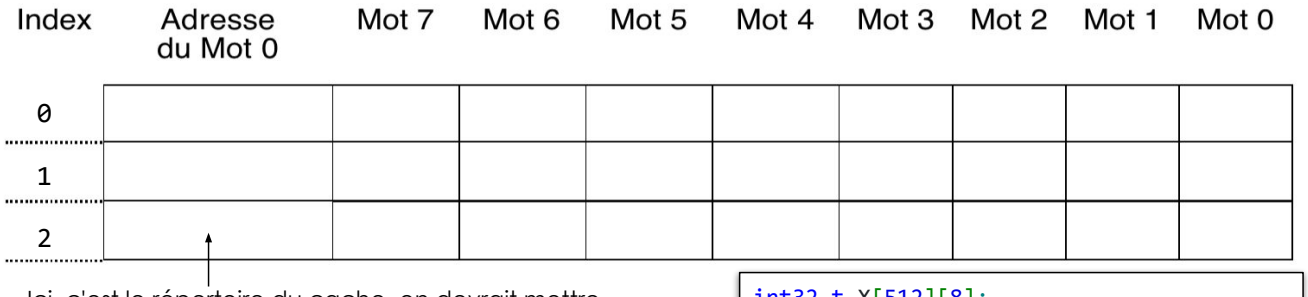
Le cache de 256 cases ne peut pas contenir tout le tableau X

⇒ 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]
```

Remplissage du cache

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]).*



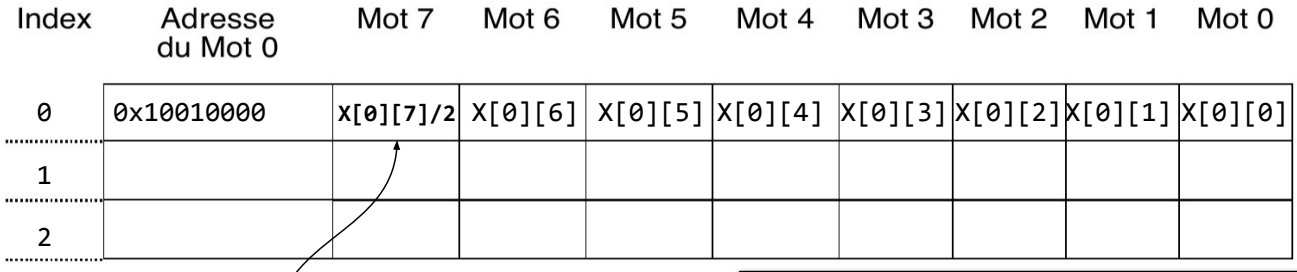
Ici, c'est le répertoire du cache, on devrait mettre les numéros de tag, voire les numéros de ligne mais plus simplement, on demande ici les numéros de ligne

```
Le tableau X
est à l'adresse 0x10010000,
```

```
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;
          }
      }
 }
```


Remplissage du cache

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]).*



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

```
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;
          }
      }
 }
```

Remplissage du cache

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]

```
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;
          }
      }
 }
```


Gain en performance

5. *Calculer 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.*

```
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;
          }
      }
 }
```

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.*

Gain en performance

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

   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 tout le tableau ⇒ 512 * 8 = 4096 MISS !

```
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;
          }
      }
 }
```

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%
   Si on avait lu tout le tableau ⇒ 100% de MISS

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.*

Gain en performance

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

   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 tout le tableau ⇒ 512 * 8 = 4096 MISS !

```
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;
          }
      }
 }
```

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%
   Si on avait lu tout le tableau ⇒ 100% de MISS

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 = 8 * ((256 * 9) + 3) + (256 * 17) = 22808
   Il y a 2048 éléments, soit une moyenne de 22808/2048 = 11.14 cycles/élément
   SI on avait lu tout le tableau = 8 * ((512 * 9) + 3) + (4096 * 17) = 106520 → 52 cycles/élément

   On est 5 fois plus lent → c'est un effet de cache

Problème de cohérence des caches

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'autres copies de cette donnée.

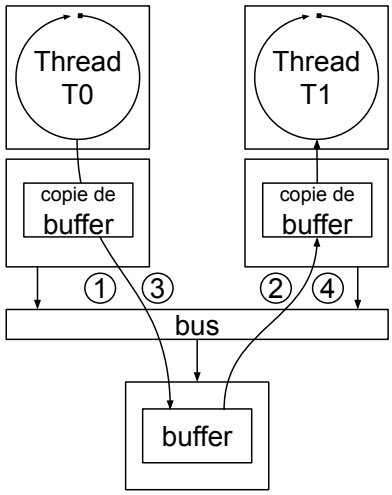
Pour les caches

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

Solution de principe logicielle

Dans le cas d'une solution logicielle, c'est le programme qui connaît les données partagées entre plusieurs cœurs 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 (à cause d'un périphérique initiateur)
- Le programme connaît les lignes qui sont touchées par ce problème car elles sont dans des buffers partagés et il peut décider de lui même de retirer ces lignes de son cache
  ⇒ il demande le *flush* total ou partiel de son cache avec lecture



Exemple :

1. T0 lit écrit **buffer** (ici, il l'a lu aussi mais c'est pas obligatoire)
2. T1 lit buffer
3. T0 écrit buffer à nouveau
4. T1 lit buffer mais l'ancienne version (dépend des évincements)

Idée

T1 sait que sa copie peut être obsolète → T1 demande au cache de son MIPS d'invalider les lignes de cache appartenant à `buffer` **Avant** de lire le buffer

Évincement de lignes (`hcpua.s`)

L'idée est d'invalider une adresse par ligne de cache du buffer avec l'instruction : `cache`

dcache_buf_invalidate()
dcache_invalidate()

registre du copro0 $16,1 contenant la longueur de ligne de cache

l'instruction : ext extrait un champs de bits dans un registre

l'instruction : cache invalide la ligne contenant l'adresse en argument

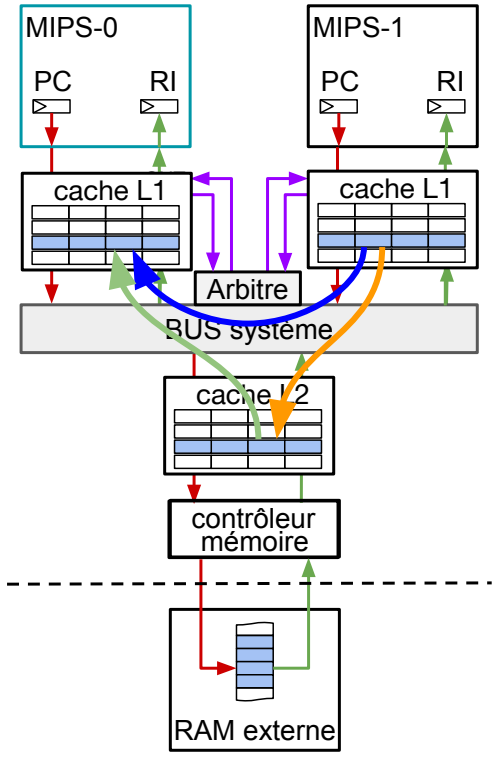
```
//https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00090-2B-MIPS32PRA-AFP-06.02.pdf 238
//https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol2.pdf p85
.globl dcache_buf_invalidate // ---------- void dcache_buf_invalidate (void *buf, unsigned size);
.globl dcache_invalidate // -------------- void dcache_invalidate (void *addr);
dcache_buf_invalidate:
    mfc0    $6, $16, 1          // $16,1 =conf reg. si DL=bits(12:10) dcache_line_size=2<<DL
    ext     $2, $6, 10, 3       // $2 = DL bits(12:10) of config register $16,1
    li      $6, 2               // 0x2
    sll     $6, $6, $2          // dcache_line_size = $6 <-- 2 << DL
    beqz    $2, dcache_inval_end // $2==0 means no cache
dcache_inval_loop:
    cache   17, 0($4)           // 17 == L1-D cache invalidate
    subu    $5, $5, $6          // size   -= dcache_line_size
    addu    $4, $4, $6          // buffer += dcache_line_size
    bgtz    $5, dcache_inval_loop // loop if size > 0
dcache_invalidate:
    cache   17, 0($4)           // invalidate the last line
    jr      $31
dcache_inval_end:               // do not erase these lines, it is to use
    jr      $31                 // the delayed slot at the end of dcache_invalidate
```

Solution de principe matérielle

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

Hypothèse, le cache L1 à une politique Write Through

- snooping based (Espionnage)
  - Toutes les écritures sont envoyées 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 sur le schéma)
- directory based (basé sur un répertoire des copies)
  - C'est le cache L2 qui donne les copies de lignes aux caches L1. Il peut alors se souvenir dans quel cache L1, il a copié chaque ligne.
  - C'est lui qui est responsable des mises à jour (flèche verte)
  - 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 car parfois c'est impossible à cause du BUS système.

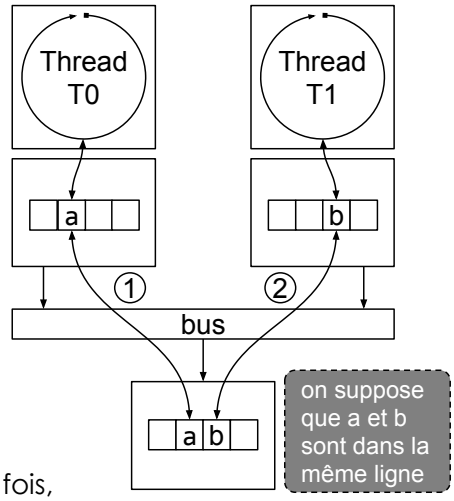


Problème du faux partage

Si le SoC dispose de plusieurs MIPS et que l'application utilise plusieurs threads s'exécutant sur des processeurs différents, il y a des risques de trafic de cohérence inutile lorsque deux variables globales sont dans la même ligne.

Ici, T0 et T1 utilisent 2 variables différentes dans la même ligne dès que l'un écrit dans sa variable, le mécanisme de gestion de la cohérence va mettre à jour, ou invalider, les autres copies, c'est inutile, mais le mécanisme ne peut pas le savoir.

```
int a, b ;
void * T0 (void * arg) {
      [...]
  ①   a = a + ... ;
      [...]
}
void * T1 (void * arg) {
      [...]
  ②   b = b + ... ;
      [...]
}
```



C'est un faux partage, pour l'éviter on peut demander d'aligner les variables globales sur des lignes de cache, mais il faut connaître la taille de la ligne de cache au moment de la compilation

→ `int a, b __attribute__((align(CACHELINESIZE))) ;`

Pour les allocations de mémoire dynamique, on alignera à chaque fois, on ne sera pas obligé de connaître la longueur de ligne à la compilation, on lira le reg. `c0_16,1`

Pour conclure

- Le CPI (Clock Per Instruction) est une mesure de la performance des processeurs
  - *Notez que pour les processeurs superscalaires qui exécutent plusieurs instructions par seconde, on utilise plutôt l'IPC (Instruction Per Cycle) pour avoir un nombre positif*
- Les caches de niveau 1 ont une influence importante sur le CPI
  - Le compilateur a un rôle important, s'il ne tient pas compte des **effets de cache**, il peut réduire le CPI
- Le CPI dépend de beaucoup de facteurs et il est fluctuant, il dépend :
  - de l'architecture du processeur et des dépendances entre instructions
  - de taux de MISS qui dépend des programmes exécutés
  - du coût en cycles des MISS qui dépends de l'activité des autres processeurs qui vont gêner l'accès au bus et à la mémoire
- La présence de plusieurs caches ou de plusieurs initiateurs entraîne un problème de cohérence des copies de lignes dans les caches,
  - On peut contourner le problème par logiciel (en flushant les caches)
  - On peut résoudre le problème en maintenant la cohérence des copies par matériel.
  - **On peut aussi éliminer le problème en configurant le cache pour que certains segments de l'espace d'adressage soit non cachés (aucun accès n'utilisent le cache !)**
- Nous avons vu qu'il est problématique de mettre deux variables globales dans la même ligne parce que cela peut mettre en œuvre le mécanisme de cohérence pour rien.

En TME

1. Calcul du CPI en fonction du cache, du programme et du SoC
2. Mesure expérimentale du CPI en fonction de la taille des caches