ASTRE : Projet

1- Etude du protocole

Cas monoprocesseur

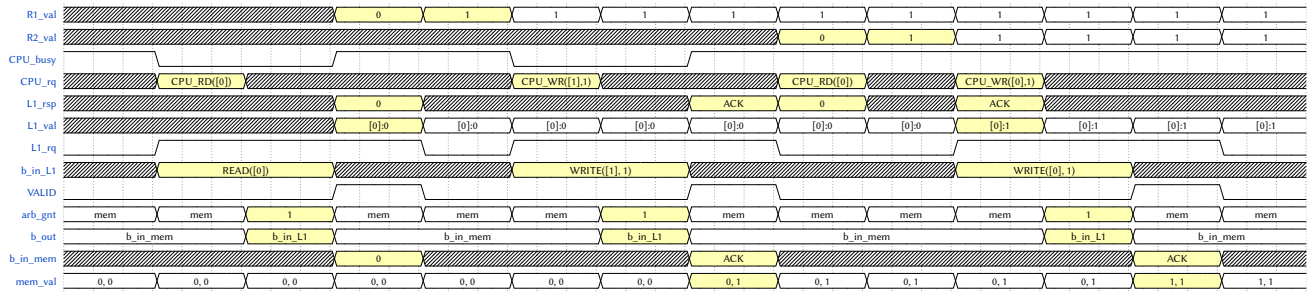
Nous nous plaçons dans un premier temps dans un contexte monoprocesseur. Le processeur dispose de trois registres (R1, R2, R3) et communique avec un cache L1 pouvant contenir un seul mot de 1 bit.
Le cache L1 peut générer des requêtes de lecture ou d'écriture vers la mémoire principale en passant par le bus. Ces requêtes ne sont transmises que lorsque le bus est disponible, sous le contrôle d'un arbitre de bus.
Le processeur dispose des instructions suivantes :

`ld`	load	`dec`	decrement by 1
`st`	store	`beq`	branch if equal
`add`	addition	`cmp`	comparaison

Programme a:

Nous allons étudié les transferts d'informations induits par l'exécution du programme :

```
P1a :
  ld R1, [0]
  add R1, R1, 1
  st R1, [1]
  ld R2, [0]
  add R2, R2,1
  st R2, [0]
```



Avant le début du programme, nous partons du principe que nous ne savons pas ce qui se trouve dans les registres, et que la mémoire contient 0 aux deux adresses. L'arbitre donne accès à la mémoire sur le bus.

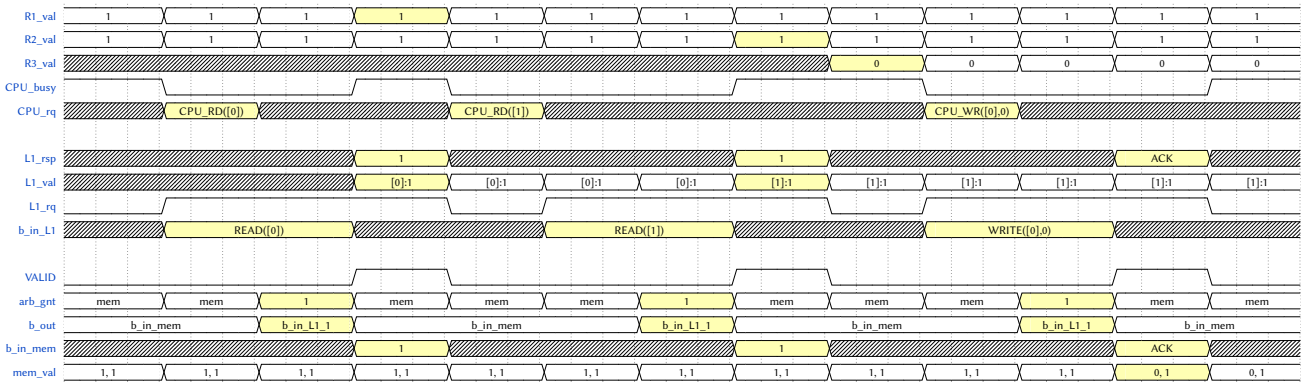
- Le processeur commence par effectuer l'instruction `ld R1, [0]` :
  - Le CPU envoie une requête `CPU_RD([0])` au cache.
  - Il y a un MISS dans le cache L1. Il gèle le processeur le temps d'exécuter la requête de lecture (`CPU_busy` passe à 0).
  - Le cache active le signal de requête `L1_rq` et écrit dans `b_in_L1` : `READ([0])` et géle en attente d'une réponse de la mémoire.
  - L'arbitre laisser la parole au processeur sur le bus et grant le processeur (`arb_gnt = 1`).
  - Le contenu de `b_in_L1` est écrit sur `b_out`.
  - Au cycle suivant, la mémoire répond sur `b_in_mem` avec la valeur à l'adresse 0 et active `VALID`.
  - Le cache dégele, stocke la valeur, la transmet au processeur, puis désactive le signal `L1_rq`.
  - Le CPU redevient occupé (`busy`) et enregistre la valeur dans `R1`.

    - La mémoire met `VALID` a 0 pour signaler qu'elle a fini et que l'arbitre peut choisir une nouvelle cible.
- L'instruction `add R1, R1, 1` se fait en un cycle, car elle ne demande pas de transfert sur le bus. Nous avons juste une mise à jour de la valeur dans le registre `R1`.
- Le processeur exécute ensuite `st R1, [1]` :
    - Le CPU envoie une requête d'écriture `CPU_WR([1],1)` au cache.
    - Il y a un MISS dans le cache. Le processeur est donc gelé. Le cache active `L1_rq` et écrit `WRITE([1],1)` dans `b_in_L1`.
    - Quand `VALID` est désactivé, l'arbitre de bus laisse la main au cache.
    - Au cycle suivant, la mémoire met à jour la valeur et envoie un `ACK` au cache.
    - Le cache reçoit l'`ACK` et répond par un `ACK` au CPU.
- Le processeur exécute `ld R2, [0]` :
    - Il transmet la commande `CPU_RD([0])` au cache.
    - Comme la valeur se trouve déjà en cache, celui-ci lui répond avec la valeur en un cycle.
- L'instruction `add R2, R2, 1` se fait en un cycle, car elle ne demande pas de transfert sur le bus. Il y a une mise à jour de la valeur dans le registre `R2`.
- Enfin, le processeur exécute `st R2, [0]` :
    - Il envoie `CPU_WR([0],1)` au cache L1.
    - La donnée se trouve en cache (HIT). Le cache modifie la valeur et envoie un `ACK` au processeur. Le processeur n'est donc pas gelé.
    - Le cache L1 active `L1_rq` et transmet la requête à la mémoire via le bus.
    - La mémoire effectue la modification et acquitte le cache pour qu'il puisse se dégeler.

Programme b:

Entre l'exécution des deux programme, on a évicté le cache. Les valeur des registre (R1 = 1, R2 = 1) et de la mémoire ([0] = 1, [1] = 1) sont gardé.

```
P1b :
ld R1, [0]
ld R2, [1]
add R3, R1, R2
st R3, [0]
```



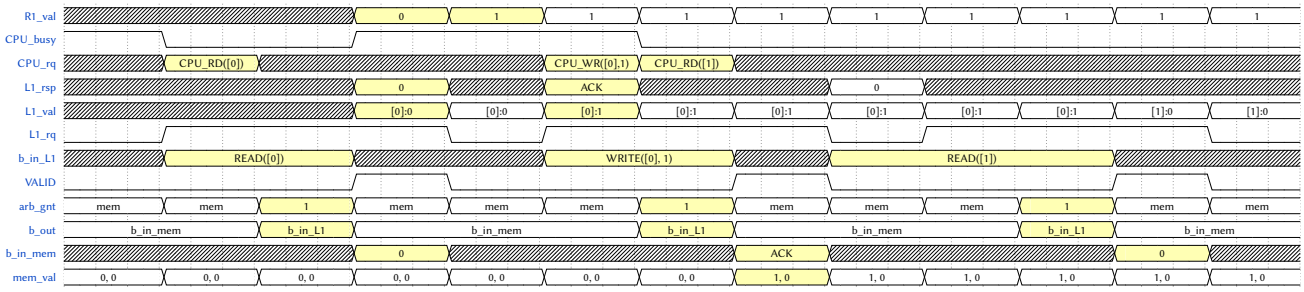
- Le processeur commence par exécuter l'instruction `ld R1, [0]`.
    - Le processeur envoie la requête `CPU_RD([0])` au cache.
    - Il y a un `MISS` dans le cache. Le cache envoie une requête READ([0]) à la mémoire.
    - Au cycle suivant, l'arbitre de BUS donne l'accès au BUS au processeur.
    - La mémoire reçoit la requête et répond au cycle suivant.
    - Le cache stocke la valeur et la transmet au processeur.
    - Le processeur stocke la valeur dans R1.

- Le processeur exécute `ld R2, [1]`.
  - Le processeur envoie la requête `CPU_RD([1])` au cache.
  - Il y a un `MISS` dans le cache. Le cache envoie une requête `READ([1])` à la mémoire.
  - Au cycle suivant, l'arbitre de BUS donne l'accès au BUS au processeur.
  - La mémoire reçoit la requête et répond au cycle suivant.
  - Le cache stocke la valeur et la transmet au processeur.
  - Le processeur stocke la valeur dans `R2`.
- L'instruction `add R3, R1, R2` se fait en un cycle, car elle ne demande pas de transfert sur le BUS. Nous avons juste une mise à jour de la valeur dans le registre `R3`.
- Le processeur exécute `st R3, [0]`.
  - Le processeur envoie une requête d'écriture `CPU_WR([0], 0)` au cache.
  - Il y a un MISS dans le cache. Il envoie donc une requête `WRITE([0], 0)` à la mémoire.
  - Au cycle suivant, l'arbitre de BUS donne l'accès au BUS au processeur.
  - La mémoire reçoit la requête, modifie la valeur et répond au cycle suivant avec un `ACK`.
  - Le cache reçoit le `ACK` et le transmet au processeur.

Programme c:

Nous avons imaginé un programme pour voir ce qui se passe quand le CPU lance une requete d'écriture sur une adresse ou la valeur est stocker en cache (l'écriture est asynchrone, le processeur n'est pas gelé mais le cache oui), suivie d'une requete de lecture/d'écriture.

```
P1b :
ld R1, [0]
add R1, R1, 1
st R1, [0]
ld R1, [1]
```



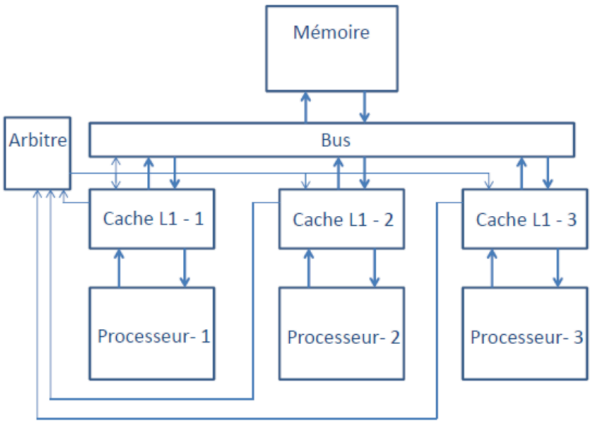
Nous faisons l'hypothèse qu'il y a 0 aux deux adresses dans la mémoire et que nous ne savons pas ce qu'il y a dans les registres ni dans le cache.

- Le processeur commence par effectuer l'instruction `ld R1, [0]` :
  - Le processeur envoie une requête `CPU_RD([0])` au cache.
  - Il y a un `MISS` dans le cache L1, qui envoie une requête à la mémoire.
  - L'arbitre laisse la parole au processeur sur le bus.
  - Au cycle suivant, la mémoire répond.
  - Le cache se dégele, stocke la valeur, la transmet au processeur, puis désactive le signal `L1_rq`.
  - Le processeur enregistre la valeur dans `R1`.
- L'instruction `add R1, R1, 1` se fait en un cycle, car elle ne demande pas de transfert sur le bus. Nous avons juste une mise à jour de la valeur dans le registre R1.
- L'instruction `st R1, [0]` se fait en un cycle pour le processeur, car il y a un `HIT` dans le cache.
  - Le cache modifie sa valeur localement, envoie un ACK au processeur, puis transmet la requête à la mémoire.
  - Pendant ce temps, le cache se gèle en attendant une réponse de la mémoire.
- Le processeur exécute ensuite l'instruction `ld R1, [1]` :

- Le processeur envoie au cache une requête `CPU_RD([1])`.
- Le cache est en attente de la réponse de la mémoire pour l'écriture asynchrone.
- Une fois que la mémoire acquitte la requête, le cache peut traiter la requête de lecture.
- Comme la valeur n'est pas présente dans le cache, il transmet une requête de lecture à la mémoire.
- Après réception de la réponse, le cache transmet la valeur au processeur.

Cas multiprocesseur

On se place maintenant dans le cas multiprocesseur, nous avons 3 processeur, muni chacun de leur cache, tous on accés au bus comme décrit sur l'image suivante:



Chaque processeur on leur jeu d'instruction comme décrit dans la partie monoprocesseur.

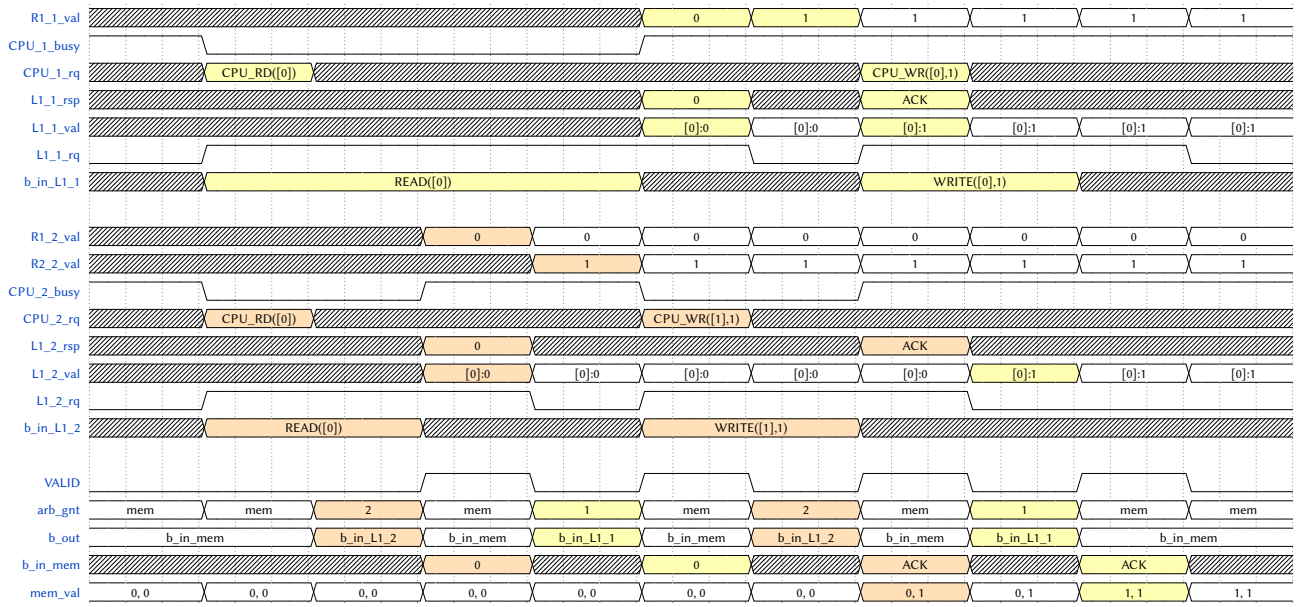
Programme a

Nous allons analyser les échanges d'informations produit pas les programme suivant sur les processeurs 1 et 2 (le 3 est inactif).

```
P1a :                   P2a :
ld R1, [0]              ld R1, [0]
add R1, R1, 1           ....
st R1, [0]              ....
....                    add R2, R1, 1
....                    st R2, [1]
```

Nous allons étudier deux ordonnancements différents pour ces programmes.
On suppose que dans la mémoire, les deux adresses contiennent la valeur `0`.

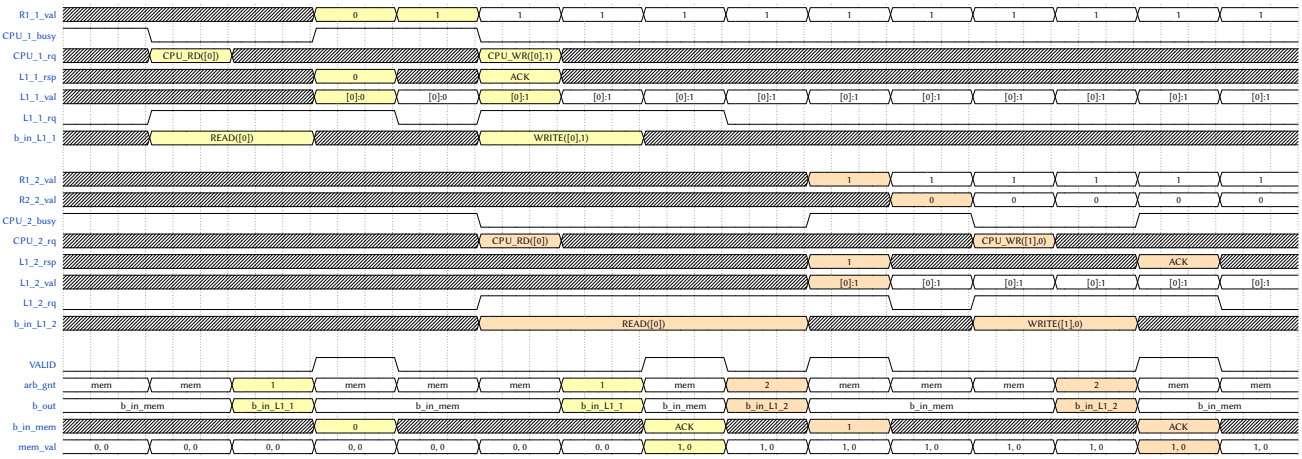
Cas 1



- On suppose que les deux processeurs exécutent simultanément l'instruction `ld R1, [0]` :
  - Ils envoient tous les deux `CPU_RD([0])` à leur cache respectif.
  - Il y a un `MISS` dans les deux caches.
  - Les caches activent leur signal `L1_rq` et écrivent dans `b_in_L1` la requête `READ([0])`.
  - L'arbitre a donc le choix entre les deux requêtes. Il choisit le cache 2.
  - La mémoire répond au cycle suivant.
  - L'arbitre de bus choisit le cache 1 au cycle suivant.
- Les deux instructions `add R1, R1, 1` sur le `CPU 1` et `add R2, R1, 1` sur le `CPU 2` se font en un cycle, car elles ne nécessitent pas de demande sur le bus.
- Comme l'arbitre de bus a décidé que le cache 2 fasse son load avant le cache 1 (qui est gelé), le processeur 2 demande de faire son écriture en premier.
- Lorsque la requête est terminée, le cache 1 pourra alors faire sa requête d'écriture.

À la fin de l'exécution, les deux adresses de la mémoire contiennent la valeur 1. Le registre R1 du processeur 1 contient la valeur 1, le registre R1 du processeur 2 contient la valeur 0, et le registre R2 du processeur 2 contient la valeur 1.

Cas 2



Ici, nous partons du principe que les deux processeurs ne commencent pas nécessairement en même temps.

- Le processus 1 commence et a le temps d'exécuter `ld R1, [0]` et `add R1, R1, 1` avant que le processus 2 demande à exécuter `ld R1, [0]`.
- Le processus 1 demande l'accès au bus pour l'instruction `st R1, [0]` au même moment que le processus 2 demande à exécuter sa première instruction.
- L'arbitre de bus a le choix entre les deux requêtes des caches et choisit celle du cache 1.
- Le cache 2 pourra donc exécuter le reste de son programme une fois la requête du cache 1 terminée.

À la fin de l'exécution, l'adresse 0 de la mémoire contient la valeur 1, tandis qu'à l'adresse 1, la valeur est 0. Le registre R1 des deux processeurs contient la valeur 1, et le registre R2 du processeur 2 contient la valeur 0.
Nous remarquons que, selon l'ordonnancement des instructions, les résultats peuvent être différents.

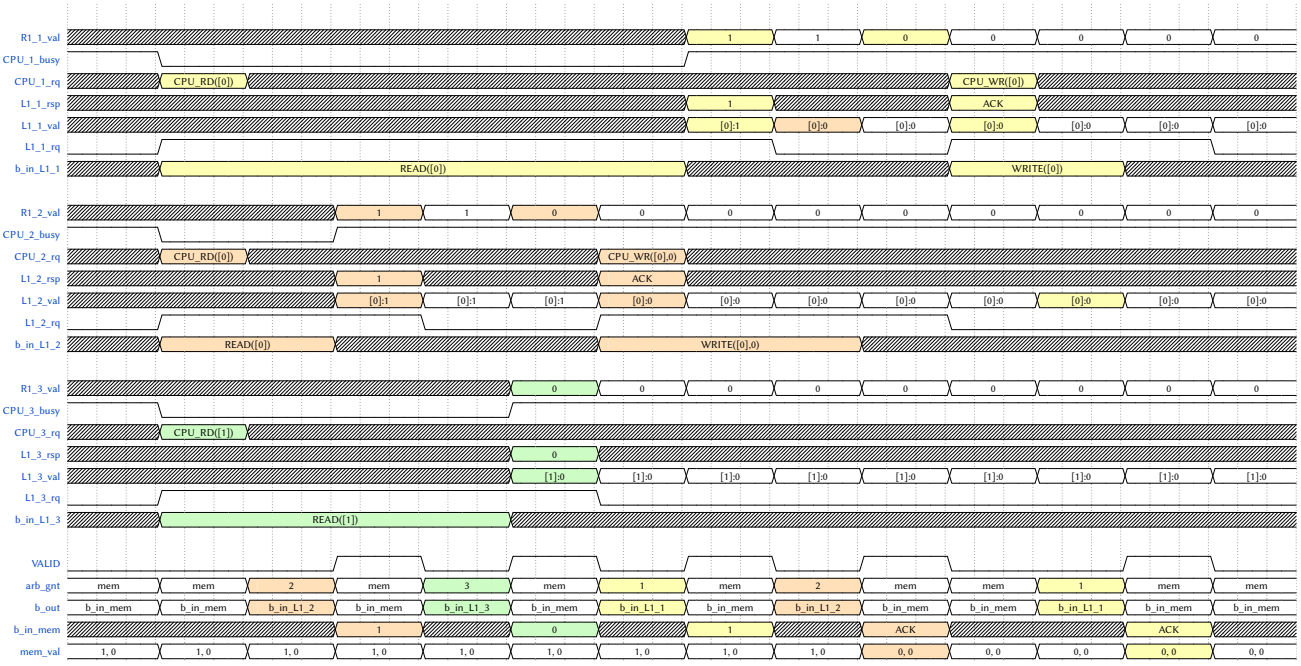
Programme b

Nous allons analyser les échanges d'informations produit pas les programme suivant sur les processeurs 1, 2 et 3.

```
P1b :                  P2b :                  P3b :
dbt: ld R1, [0]        dbt: ld R1, [0]             ....
     cmp R1, 0              cmp R1, 0
     beq dbt                beq dbt
     dec R1                 dec R1                 ld R1, [1]
     st R1, [0]             st R1, [0]
     <sc>                   <sc>
```

Nous allons étudier deux ordonnancements differents pour ces programmes.
On suppose que dans la mémoire, la première adresse contient la valeur 1 et la deuxième la valeur 0.

Cas 1



- On suppose que P1 et P2 exécutent `ld R1, [0]` et que P3 exécute `ld R1, [1]` simultanément.
  - Tout les caches MISS, donc tout les caches envoient une requête de lecture.
  - L'arbitre choisit P2, la mémoire répond le cycle d'après avec la valeur 1 pour l'adresse 0, le cache L1_2 est mit à jour.
  - L'arbitre choisit ensuite P3, la mémoire répond le cycle d'après avec la valeur 0 pour l'adresse 1, le cache L1_3 est mit à jour.
- L'arbitre finit par choisir P1, pendant ce temps P2 envoie une requête d'écriture avec la valeur 0 pour l'adresse 0. Il a vérifié les cycles d'avant que R1 (valeur 1) n'est pas 0, donc il valide la prise de token en écrivant 0 en mémoire.
  - La mémoire répond le cycle d'après à P1 avec la valeur 1 pour l'addresse 0, le cache L1_1 est mit à jour.
- L'arbitre choisit P2, La requête d'écriture de P2 de la valeur 0 pour l'adresse 0 passe sur le bus.
  - Le snoopy du cache L1_1 de P1 observe la transaction est update le cache avec la valeur 0.
- P1 envoie une requête d'écriture avec la valeur 0 pour l'addresse 0. Son registre R1 n'a pas été mit à jour malgrès le snoopy, il valide donc la prise de token en écrivant 0 en mémoire.

Problème de cohérence

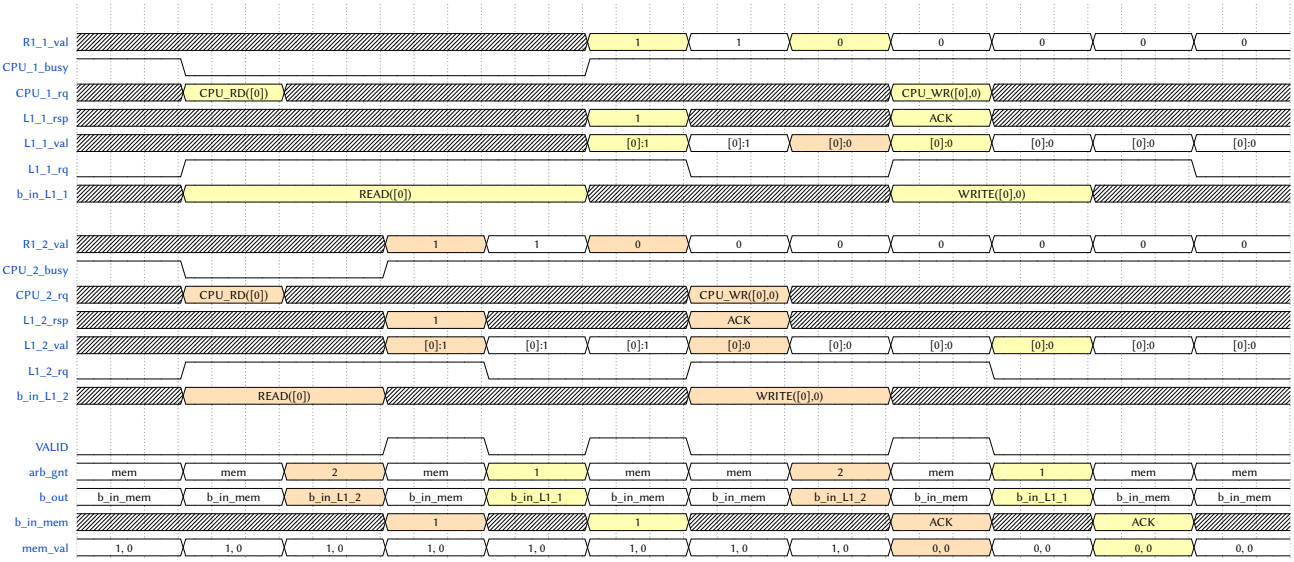
On peut observer que malgrès le méchanisme de snoop, qui nous permet une cohérence au niveau des caches L1, la prise de token ici n'est pas exclusive, car P1 et P2 rentrent en section critique en même temps.

Sachant qu'ici le problème vient du fait que le registre R1 n'est pas mit à jour lors du mécanisme de snoop, on pourrait faire les comparaisons directement au niveau du cache, sans passer par R1.

Cette solution n'est pas satisfaisante car elle ne prend pas en compte d'autre problèmes de cohérence illustré lors du cas 2.

Cas 2

Ici on ignore P3 car on s'intéresse aux problèmes de concurrence entre P1 et P2.



- On suppose que P1 et P2 exécutent `ld R1, [0]` simultanément.
  - Tout les caches MISS, donc tout les caches envoient une requête de lecture.
  - L'arbitre choisit P2, la mémoire répond le cycle d'après avec la valeur 1 pour l'adresse 0, le cache L1_2 est mit à jour.

- L'arbitre choisit P1, la mémoire répond le cycle d'après avec la valeur 1 pour l'adresse 0, le cache L1_1 est mit à jour.
- P2 envoie une requête d'écriture avec la valeur 0 pour l'adresse 0. Il valide la prise de token en écrivant 0 en mémoire (après avoir vérifié que R1 != 0).
- L'arbitre choisit P2, La requête d'écriture de P2 de la valeur 0 pour l'adresse 0 passe sur le bus.
  - Le snoopy du cache L1_1 de P1 observe la transaction est update le cache avec la valeur 0, mais P1 à déja effectué la comparaison et s'apprète à prendre le token.
- P1 envoie une requête d'écriture avec la valeur 0 pour l'addresse 0, la transaction passe ensuite sur le bus et en mémoire.

Problème de cohérence

Encore une fois on observe un problème de cohérence où P1 et P2 rentrent en section critique en même temps. Ici, on est sur un problème de timing car P1 à déja décider de rentrer en section critique avant que le mécanisme de snoop ne gère la cohérence.

Nous proposons donc un mécanisme de transaction atomique pour permetre la gestion de READ, MODIFY, WRITE de façon atomique afin d'éviter qu'un processus puisse lire une valeur en mémoire avant qu'elle soit modifiée, décrit dans la partie du méchanisme de garantie d'accès exclusif.

Intéret du méchanisme de SNOOP

On a pu observer dans les deux examples que le méchansime de snoop nous permet de garantir une cohérence au niveau des différents caches L1 des différents processeurs. Ce qui signifie que si P1 modifie la valeur contenue dans l'adresse 0 en mémoire, et que P2 contient la valeur dans son cache, alors le L1 de P2 aura a terme la valeur mit à jour de la mémoire meme si P2 ne fait aucune lecture.

Méchanisme pour garantir un accès exclusif aux donnée partagées

Nous proposons d'ajouter l'instruction `swap Rx, [x]`, qui échange la valeur dans le registre `Rx` avec la valeur dans la mémoire à l'address `x`. Un swap suffit dans notre cas car nous souhaitons géré un mécanisme de verrou avec seulement deux valeurs différentes. Le nouveau code ressemblerait à:

```
P1c :                    P2c :
dbt: cmp R1, 0          dbt: cmp R1, 0
     beq r0                   beq r0
     add R1, 1                add R1, 1
r0:  swap R1, [0]       r0:  swap R1, [0]
     cmp R1, 0                cmp R1, 0
     beq r0                   beq r0
     <sc>                     <sc>
```

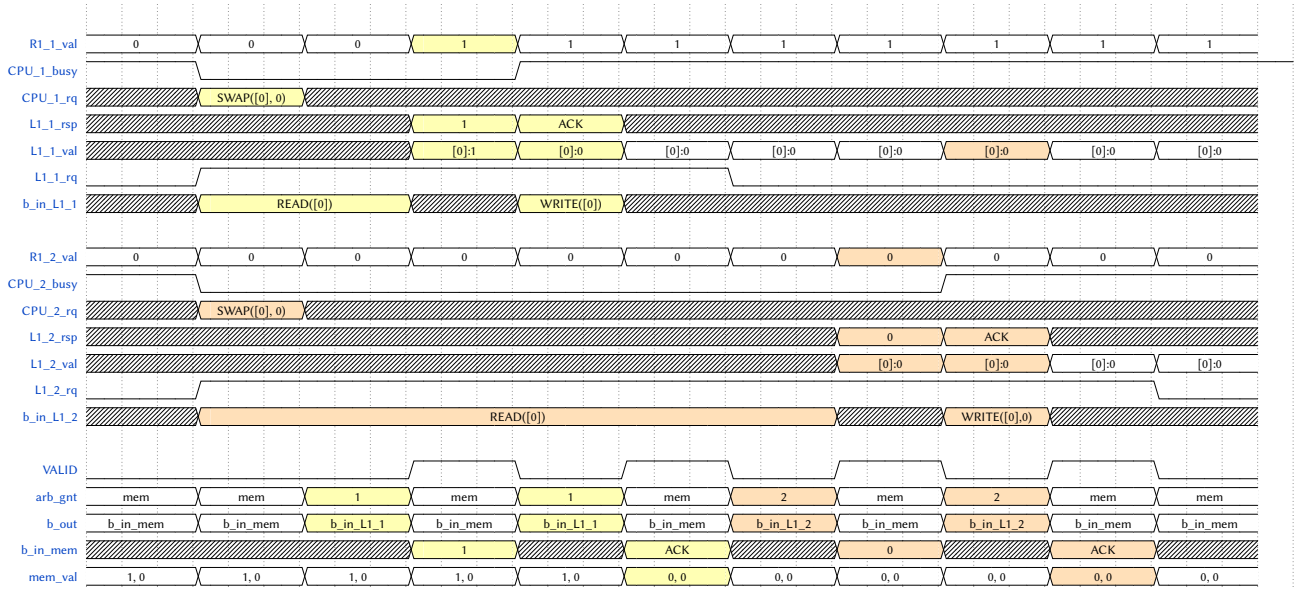
- Les trois premières instructions permettent de garantir que `R1` possède la valeur 0.
- À partir du label `r0`, on échange la valeur 0 de `R1` avec la valeur dans la mémoire à l'adresse 0:
  - Si la valeur en mémoire est 0, alors `R1` contient 0:
    - On ne peut pas prendre le token, car il est déja prit par un autre processus.
    - `R1 = 0`, donc on revient au label `r0`, et on refait le swap.
  - Si la valeur en mémoire est 1, alors `R1` contient 1:
    - On peut prendre le token, donc on peut rentre en section critique
    - `R1 != 0`, donc on continue en section critique.

On peut observer que le swap nous permet de tester et de prendre le token en une seule transaction car:
- Si le token est déja prit, alors la mémoire contient 0, et on écrit 0 par dessus.

- Si le token n'est pas encore prit, alors la mémoire contient 1, on écrit 0 par dessus, et donc plus personne ne pourra prendre le token tant qu'on écrit pas 1 a la fin de notre section critique.

On l'implémente en monopolisant le bus pour un processus donnée pour une transaction de lecture suivit d'une transaction d'écriture.

Exemple d'exécution



- On ignore la mise à 0 des registres, donc P1 et P2 exécutent `swap R1, [0]` simultanément avec `R1` à 0.
- Tout les caches MISS, donc ils envoient une requête de lecture vers le bus. On peut passer par les caches car la cohérence est assuré par le méchanisme de snoop.
- L'arbitre choisit P1, donc P2 ne pourra pas être choisit tant que L1_req ne passe pas à 0, donc tant que P1 n'a pas finit sa transaction de lecture puis écriture.
- L'arbitre choisit ensuite P2 à la fin de la transaction atomique de P1, il effectue aussi une lecture / écriture.

On finit par observer que P1 peut rentrer en section critique car son `R1` est égal à 1 mais que P2 ne peut pas, car son `R1` est égal à 0. Notre mécanisme de transaction atomique nous permet donc d'assurer l'atomicité de l'opération `swap`, ce qui règle les problèmes d'exclusivité observés précédement.