Projet ASTRE

Modélisation et Vérification de systèmes concurrents

Tara Aggoun
21304625

Riwan Coëffic
21309062

Sorbonne Université
2024-2025

1- Etude du protocole

Cas monoprocesseur

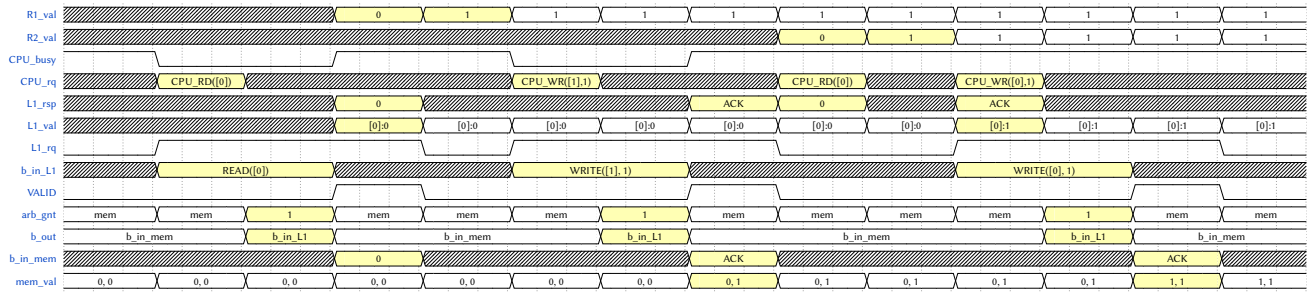
Nous nous plaçons dans un premier temps dans un contexte monoprocesseur. Le processeur dispose de trois registres (R1, R2, R3) et communique avec un cache L1 pouvant contenir un seul mot de 1 bit.
Le cache L1 peut générer des requêtes de lecture ou d'écriture vers la mémoire principale en passant par 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 supposons que les registres contiennent des valeurs indéterminées et que la mémoire contient la valeur 0 à ses deux adresses. L'arbitre donne accès à la mémoire sur le bus.

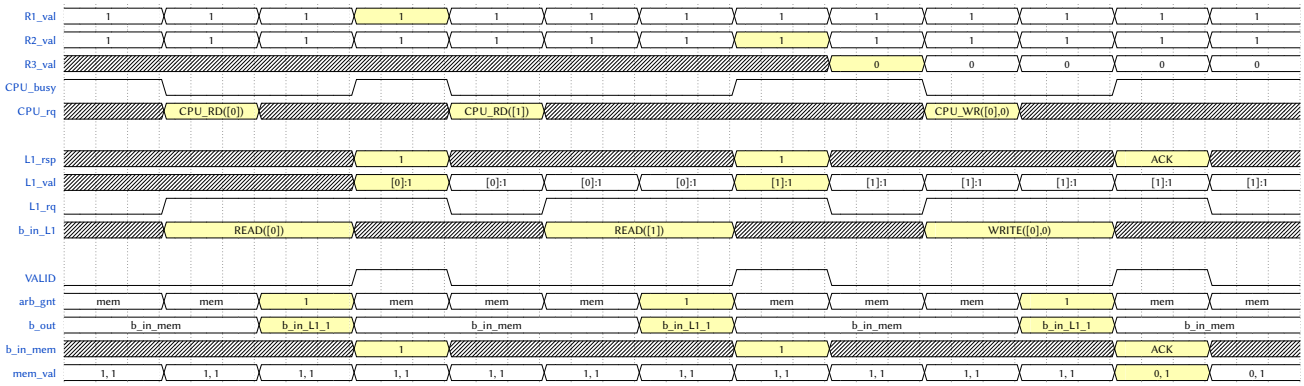
- Le processeur commence par effectuer l'instruction `ld R1, [0]` :
  - Le CPU envoie une requête de lecture `CPU_RD([0])` au cache L1.
  - Comme la donnée n'est pas présente en cache (`MISS`), le processeur est gelé (`CPU_busy = 0`) en attendant que la requête soit satisfaite.
  - Le cache L1 active le signal de requête `L1_rq`, écrit `READ([0])` dans `b_in_L1` et gèle en attendant la réponse de la mémoire.
  - L'arbitre accorde l'accès au cache L1 (`arb_gnt = 1`), transmettant `b_in_L1` sur `b_out`.
  - Au cycle suivant, la mémoire répond via `b_in_mem` en placant la valeur contenu dans l'adresse 0 et en activant `VALID`.
  - Le cache L1 se dégel, stocke la valeur récupérée, la transmet au processeur, puis désactive le signal `L1_rq`.
  - Le CPU redevient occupé (`CPU_busy = 1`) et enregistre la valeur dans `R1`.

  - Enfin, La mémoire désactive le signal `VALID` pour indiquer la fin de l'opération : l'arbitre peut maintenant choisir une nouvelle cible.
- L'instruction `add R1, R1, 1` se fait en un cycle, car ne demandant pas de transfert sur le bus. La valeur dans le registre `R1` est mise à jour.
- Le processeur exécute ensuite `st R1, [1]` :
  - Le CPU envoie une requête d'écriture `CPU_WR([1],1)` au cache L1.
  - En raison d'un `MISS` dans le cache, le processeur est gelé. Le cache L1 active `L1_rq` et écrit `WRITE([1],1)` dans `b_in_L1`.
  - L'arbitre accorde l'accès du bus au cache L1, transmettant `b_in_L1` sur `b_out`.
  - Au cycle suivant, la mémoire met à jour la valeur à l'addresse 1 et envoie un `ACK` au cache.
  - Le cache reçoit l'`ACK` via le bus et transmet un `ACK` au CPU.
- Le processeur exécute `ld R2, [0]` :
  - Il transmet la commande `CPU_RD([0])` au cache.
  - La valeur se trouvant déjà dans le cache (`HIT`), celui-ci répond immédiatement avec la valeur, en un cycle.
- L'instruction `add R2, R2, 1` s'effectue également en un cycle, car elle ne demande pas de transfert sur le bus. La valeur dans le registre `R2` est mise à jour.
- Enfin, le processeur exécute `st R2, [0]` :
  - Il envoie `CPU_WR([0],1)` au cache L1.
  - La donnée se trouvant déjà dans le cache (`HIT`), celui-ci modifie la valeur localement et renvoie un `ACK` au CPU, qui n'est donc pas gelé.
  - Le cache L1 active `L1_rq` pour transmettre la requête d'écriture à la mémoire via le bus.
  - La mémoire met à jour la valeur à l'adresse spécifiée et acquitte le cache, lui permettant de se dégeler.

Programme b

Entre l'exécution des deux programmes, le cache a été évicté. Les valeurs des registres (R1 = 1, R2 = 1) et de la mémoire ([0] = 1, [1] = 1) sont conservées.

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



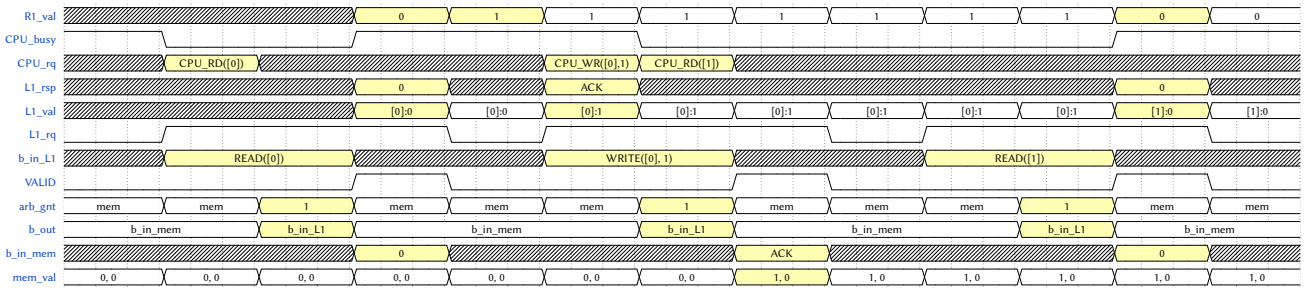
- Le processeur commence par exécuter l'instruction `ld R1, [0]`.
  - Le CPU envoie une requête de lecture `CPU_RD([0])` au cache L1.
  - Comme la donnée n'est pas présente en cache (`MISS`), une requête `READ([0])` est envoyée par le cache L1.

  - L'arbitre accorde l'accès au bus, et la mémoire répond avec la donnée au cycle suivant avec la donnée.
  - Le cache stocke la valeur, la transmet au CPU, et celui-ci stocke la valeur dans `R1`.
- Le processeur exécute `ld R2, [1]`.
  - Le CPU envoie une requête de lecture `CPU_RD([1])` au cache L1.
  - Comme la donnée n'est pas présente en cache (`MISS`), une requête `READ([1])` est envoyée par le cache L1.
  - L'arbitre accorde l'accès au bus, et la mémoire répond avec la donnée au cycle suivant.
  - Le cache stocke la nouvelle valeur (réécrivant par dessus l'ancienne valeur), la transmet au CPU, et celui-ci la stocke dans `R2`.
- L'instruction `add R3, R1, R2` se fait en un cycle, car ne demandant pas de transfert sur le bus. La valeur dans le registre `R3` est mise à jour.
- Le processeur exécute `st R3, [0]`.
  - Le CPU envoie une requête d'écriture `CPU_WR([0], 0)` au cache L1.
  - Comme la donnée n'est pas en cache (`MISS`), une requête `WRITE([0], 0)` est envoyée par le cache L1.
  - L'arbitre accorde l'accès au bus, et la mémoire met à jour la valeur à l'adresse 0 au cycle suivant, renvoyant un `ACK`,
  - Le cache reçoit l'`ACK` via le bus et transmet un `ACK` au CPU.

Programme c

Nous avons imaginé un programme pour observer l'enchaînement d'une requête d'écriture par le CPU sur une adresse dont la valeur est déjà stockée en cache L1, suivie d'une autre requête (lecture ou écriture). L'écriture étant asynchrone, le processeur n'est donc pas gelé, mais le cache l'est pendant l'opération.

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



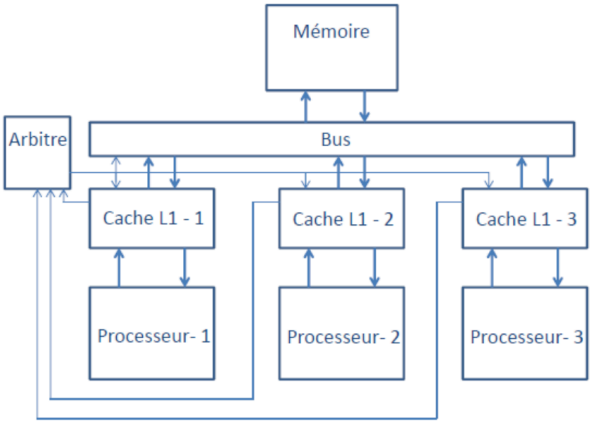
Nous faisons l'hypothèse que les deux adresses de la mémoire contiennent la valeur 0 et que nous ne savons pas ce qu'il y a dans les registres ni dans le cache.

- Le processeur commence par effectuer l'instruction `ld R1, [0]` :
  - Le CPU envoie une requête de lecture `CPU_RD([0])` au cache L1.
  - Comme la donnée n'est pas présente en cache (`MISS`), une requête `READ([0])` est envoyée par le cache L1.
  - L'arbitre accorde l'accès au bus, et la mémoire répond avec la donnée au cycle suivant.
  - Le cache stocke la valeur, désactive le signal `L1_rq`, la transmet au CPU, et celui-ci stocke la valeur dans `R1`.
- L'instruction `add R1, R1, 1` se fait en un cycle, car ne demandant pas de transfert sur le bus. La valeur dans le registre `R1` est mise à jour.

- 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 CPU, 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 CPU envoie une requête de lecture `CPU_RD([1])` au cache L1.
  - Le cache est en attente de la réponse de la mémoire pour l'écriture asynchrone et ne peut donc pas traiter immédiatement la requête du CPU.
  - Une fois que la mémoire acquitte la requête d'écriture, le cache peut alors traiter la requête de lecture.
  - Comme la valeur n'est pas présente en cache, il transmet une requête de lecture à la mémoire.
  - Après réception de la réponse, le cache transmet la valeur au CPU, permettant au cache et au CPU de se dégeler.

Cas multiprocesseur

On se place maintenant dans le contexte d'un système multiprocesseur, avec 3 processeurs, muni chacun de leur propre cache L1. Tous ont accès au bus, comme décrit dans l'image suivante:



Chaque processeur dispose du même jeu d'instructions, comme décrit dans la section mono-processeur.

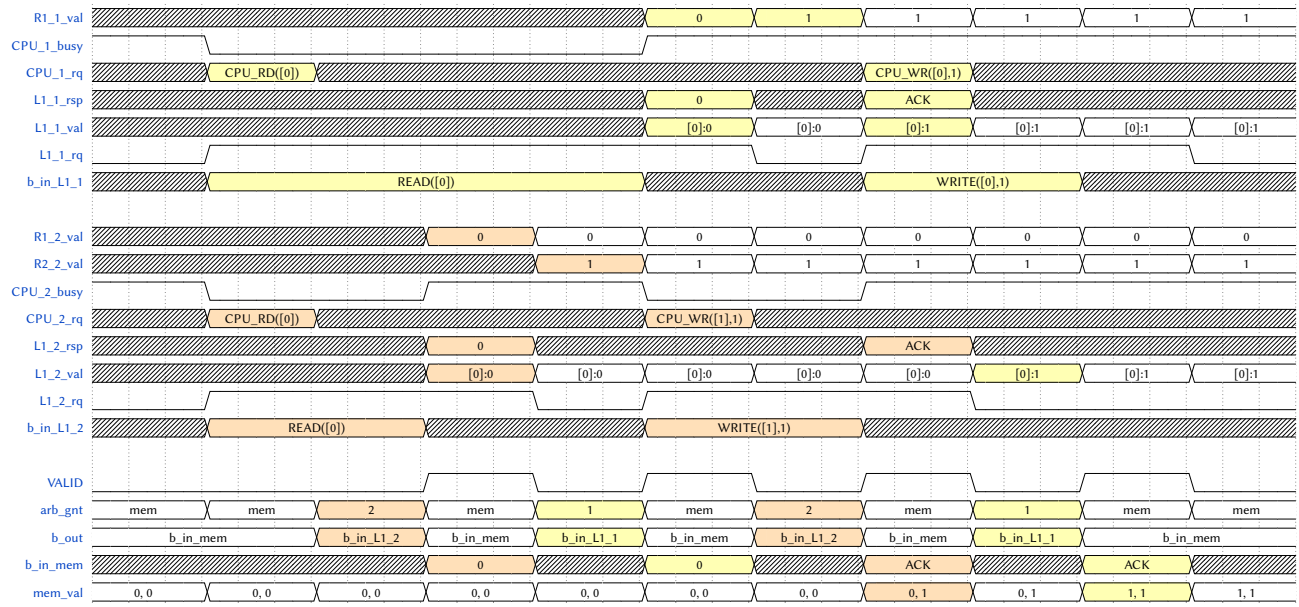
Programme a

Nous allons analyser les échanges d'informations générés par les programmes suivants sur les processeurs 1 et 2 (le processeur 3 étant 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 les deux adresses en mémoire contiennent la valeur 0.

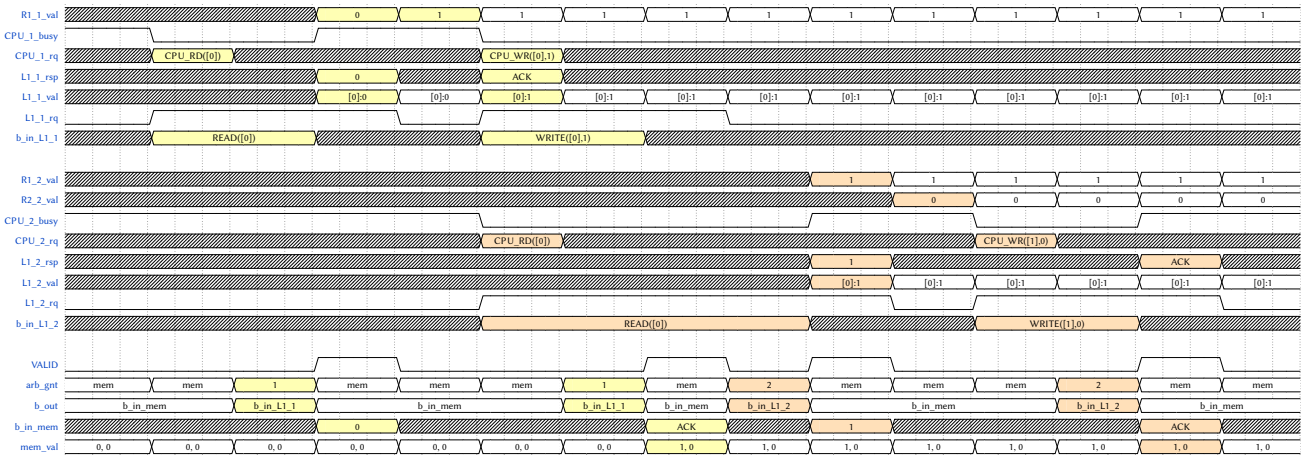
Cas 1



- On suppose que les deux processeurs exécutent simultanément l'instruction `ld R1, [0]` :
  - Chacun envoie `CPU_RD([0])` à son cache L1 respectif.
  - Il y a un `MISS` dans les deux caches.
  - Les caches activent leur signal `L1_rq` et écrivent la requête `READ([0])` dans `b_in_L1`.
  - L'arbitre a donc le choix entre les deux requêtes et selectionne le cache 2.
  - La mémoire répond au cycle suivant.
  - L'arbitre de bus choisit ensuite le cache 1 au cycle suivant.
- Les deux instructions `add R1, R1, 1` pour le `CPU 1` et `add R2, R1, 1` pour le `CPU 2` se font en un cycle, car elles ne nécessitent pas d'accès au bus.
- Comme l'arbitre de bus a permis au cache 2 de faire son `load` avant le cache 1, le CPU 2 effectue son écriture en premier.
- Une fois cette requête terminée, le cache 1 peut 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 CPU 1 commence et a le temps d'exécuter `ld R1, [0]` et `add R1, R1, 1` avant que le CPU 2 exécute `ld R1, [0]`.
- Ensuite, Le CPU 1 demande l'accès au bus pour l'instruction `st R1, [0]`, au même moment où le processus 2 demande à exécuter sa première instruction `ld R1, [0]`.
- L'arbitre de bus doit choisir entre les deux requêtes des caches et selectionne celle du cache 1.
- Le cache 2 pourra donc exécuter le reste de son programme une fois que la requête du cache 1 sera terminée.

À la fin de l'exécution, l'adresse 0 de la mémoire contient la valeur 1, tandis que l'adresse 1 reste avec la valeur 0. Le registre `R1` des deux processeurs contiennent 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.

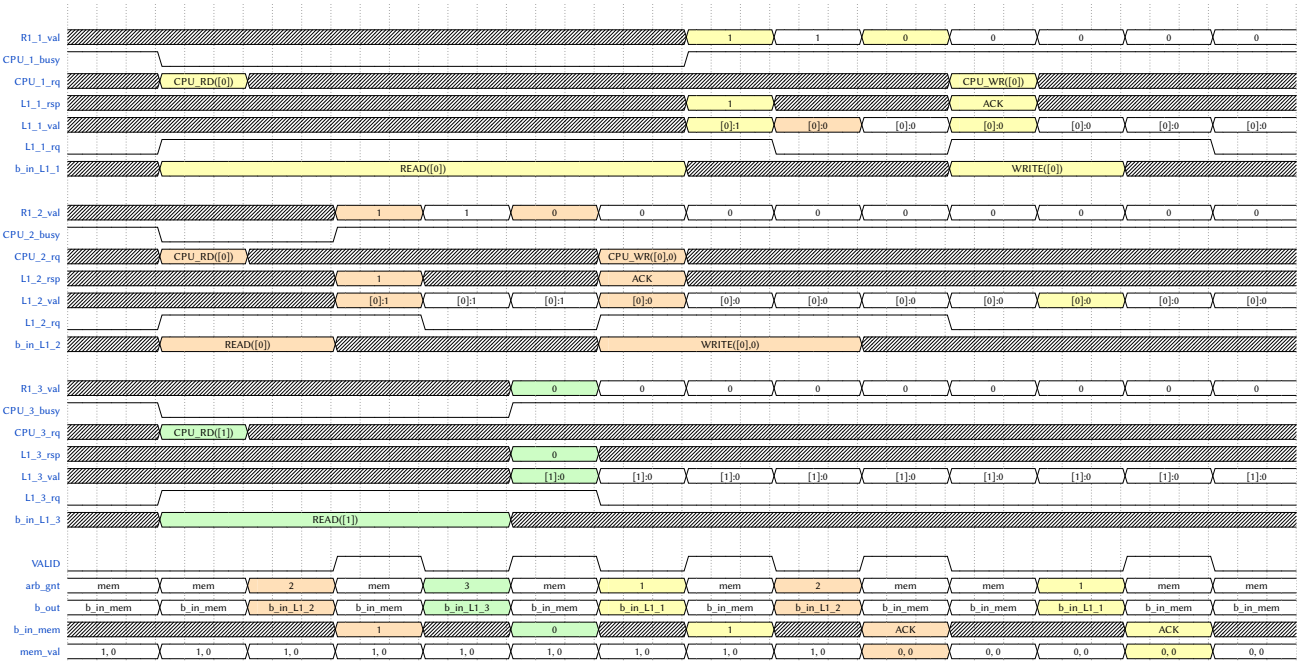
Programme b

Nous allons analyser les échanges d'informations produit pas les programmes suivants 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.
Nous supposons 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]`, tous simultanément.
  - Tout les caches `MISS`, envoyant chacun une requête de lecture.
  - L'arbitre choisit P2. La mémoire répond au cycle suivant avec la valeur 1 pour l'adresse 0, et le cache `L1_2` est mis à jour.
  - L'arbitre choisit ensuite P3. La mémoire répond au cycle suivant avec la valeur 0 pour l'adresse 1, et le cache `L1_3` est mis 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 vérifie que `R1` (valeur 1) n'a pas été modifié, et valide donc la prise de jeton en écrivant la valeur 0 en mémoire et rentre en section critique.
  - La mémoire répond au cycle suivant à P1 avec la valeur 1 pour l'addresse 0, le cache `L1_1` est mis à jour.
- L'arbitre choisit à nouveau P2. La requête d'écriture de P2 (valeur 0 pour l'adresse 0) est envoyé sur le bus.
  - Le snoopy du cache `L1_1` de P1 observe la transaction est met à jour son cache avec la valeur 0.
- Enfin, P1 envoie une requête d'écriture avec la valeur 0 pour l'addresse 0. Son registre `R1` n'ayant pas été mis à jour malgré le snoopy, il valide donc la prise de jeton en écrivant 0 en mémoire et rentre en section critique.

Problème de cohérence

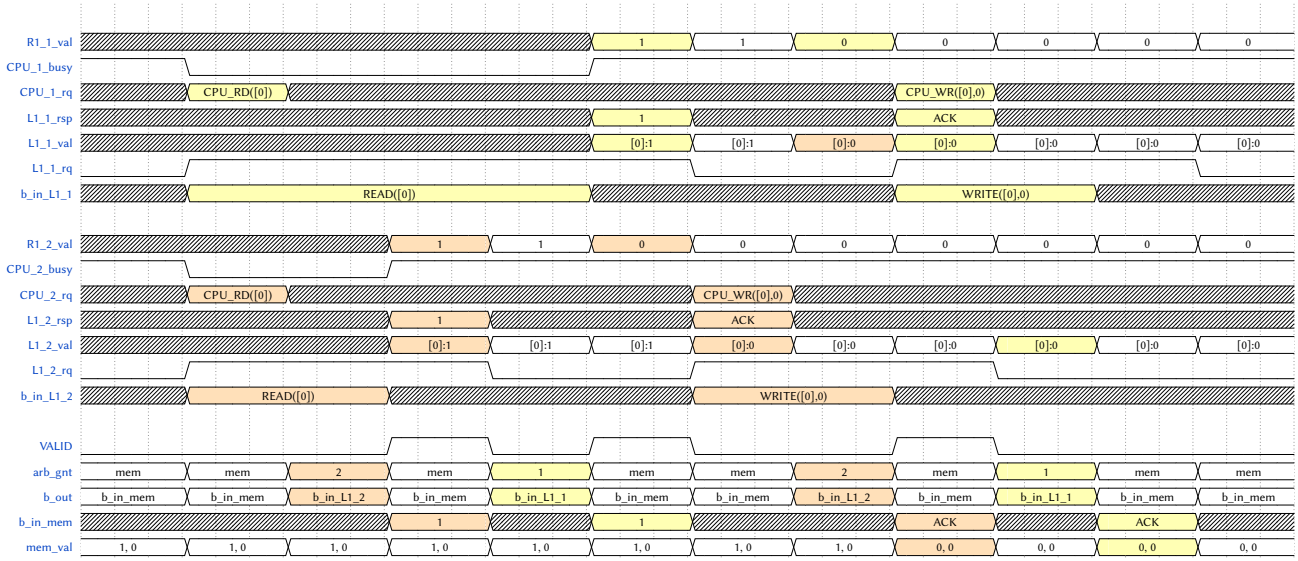
Nous observons que, malgré le méchanisme de snooping qui assure la cohérence au niveau des caches L1, la prise de jeton ici n'est pas exclusive. En effet, P1 et P2 rentrent en section critique en même temps.

Le problème ici vient du fait que le registre `R1` n'est pas été mis à jour lors du mécanisme de snooping, ce qui entraîne une incohérence. Une solution pourrait être de réaliser les comparaisons directement au niveau des caches, sans passer par les registres.

Cependant, cette solution n'est pas satisfaisante car elle ne résout pas d'autres problèmes de cohérence, illustré dans le cas 2.

Cas 2

Dans ce cas, nous ignorons P3 afin de nous concentrer sur les problèmes de concurrence entre P1 et P2.



- On suppose que P1 et P2 exécutent simultanément `ld R1, [0]`.

  - Les deux caches `MISS`, donc chaque cache envoie une requête de lecture.
  - L'arbitre choisit d'abord P2. La mémoire répond au cycle suivant avec la valeur 1 pour l'adresse 0, et le cache `L1_2` est mis à jour.
  - L'arbitre choisit P1. La mémoire répond au cycle suivant avec la valeur 1 pour l'adresse 0, le cache `L1_1` est mis à jour.
- P2 envoie une requête d'écriture avec la valeur 0 pour l'adresse 0. Il valide la prise de jeton en écrivant 0 en mémoire, après avoir vérifié que `R1` contient la valeur 1 (et non 0). P2 entre donc en section critique.
- L'arbitre choisit à nouveau P2, et la requête d'écriture (de la valeur 0 pour l'adresse 0) est transmise sur le bus.
  - Le snoopy dans le cache `L1_1` de P1 observe la transaction est met à jour le cache avec la valeur 0. Cependant, au moment de la mise à jour, P1 à déja effectué la comparaison et s'apprète à prendre le jeton, ce qui signifie qu'il entre aussi en section critique.
- P1 envoie alors une requête d'écriture avec la valeur 0 pour l'addresse 0. La transaction passe ensuite sur le bus et est enregistré en mémoire.

Problème de cohérence

Encore une fois, nous observons un problème de cohérence, car P1 et P2 rentrent en section critique en même temps. Il s'agit ici d'un problème de timing : P1 a déja décidé de rentrer en section critique avant que le mécanisme de snooping n'ait le temps de gèrer la cohérence entre les caches.

Nous proposons donc un mécanisme de **transaction atomique** pour permetre la gestion des operations READ, MODIFY, WRITE de façon atomique. Ce mécanisme permettrait d'éviter qu'un processus puisse lire une valeur en mémoire avant qu'elle ne soit modifiée, garantissant ainsi un accès exclusif. Cette approche est décrite plus en détail dans la section sur le mécanisme de garantie d'accès exclusif.

Intéret du méchanisme de SNOOP

Dans les deux exemples précédents, nous avons observé que le mécanisme de snooping garantit la cohérence entre les différents caches L1 des processeurs. Cela signifie que si P1 modifie la valeur contenue dans l'adresse 0 en mémoire, et que P2 a cette la valeur dans son cache L1, alors le cache L1 de P2 finira par être mis à jour à terme avec la valeur modifiée de la mémoire, même si P2 ne réalise aucune opération de lecture.

Mécanisme pour garantir un accès exclusif aux données partagées

Nous proposons l'ajout de l'instruction `swap Rx, [x]`, qui échange la valeur contenue dans le registre `Rx` avec celle située dans la mémoire à l'adresse `x`. Ce mécanisme de swap est suffisant dans notre cas, car il permet de gérer un verrou sur une donnée partagée, en n'utilisant que deux valeurs différentes, 0 et 1. L'objectif est de garantir qu'un seul processus puisse prendre le jeton et entrer en section critique. 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` prend la valeur 0.
- À partir du label `r0`, le processus échange la valeur de `R1` (0) avec la valeur stockée en mémoire à l'adresse 0 en utilisant l'instruction `swap`:

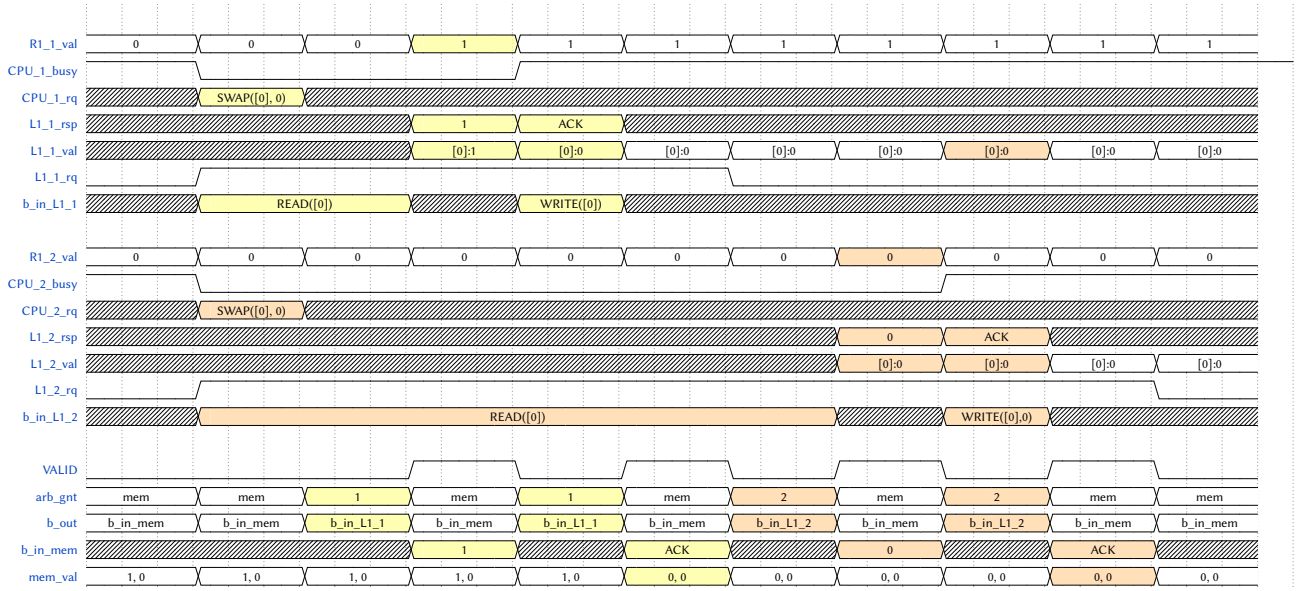
- Si la valeur en mémoire est 0, alors `R1` contient 0:
  - Cela signifie que le processus ne peut pas prendre le jeton car il est déja pris par un autre processus.
  - `R1 = 0`, le processus revient au label `r0`, et tente à nouveau de refaire l'échange.
- Si la valeur en mémoire est 1, alors `R1` contient 1:
  - Cela signifie que le processus peut prendre le jeton car il est libre, il peut donc rentrer en section critique.
  - `R1 != 0`, le processus continue en section critique.

Nous pouvons observer que le mécanisme de swap nous permet de tester et de prendre le jeton en une seule transaction atomique car:

- Si le jeton est déja prit, alors la mémoire contient 0, et l'échange écrit 0 par dessus cette valeur, ne modifiant pas vraiment la mémoire.
- Si le jeton n'est pas prit, alors la mémoire contient 1. Le processus écrit 0 dans la mémoire, ce qui rend impossible la prise de jeton par un autre processus, tant que celui-ci n'a pas écrit 1 à la fin de sa section critique.

Ce ménasime repose en pratique sur la **monopolisation du bus** par un seul processus pendant la durée d'une transaction de lecture suivit d'une transaction d'écriture.

Exemple d'exécution



- Nous ignorons la mise à 0 des registres, donc P1 et P2 exécutent simultanément l'instruction `swap R1, [0]` avec `R1` initialisé à 0.
- Tout les caches `MISS`, ils envoient donc une requête de lecture vers le bus. La cohérence au niveau des caches est assurée par le mécanisme de snooping, ce qui permet leur utilisation pour l'opération de swap.
- L'arbitre choisit d'abord P1, ce qui empêche P2 d'accéder au bus tant que la transaction de lecture puis d'écriture de P1 n'est pas terminée (tant que `L1_1_req` ne passe pas à 0).
- Une fois la transaction atomique de P1 terminée, l'arbitre choisit P2. Il effectue également une transaction atomique de lecture puis écriture.

Au final, nous observons que P1 peut entrer en section critique, car la valeur de son registre `R1` est égal à 1. En revanche, ce n'est pas possible pour R2, car son registre `R1` reste à 0. Nous démontrons donc que notre mécanisme de transaction atomique garantit l'atomicité de l'opération `swap`, résolvant ainsi les problèmes d'exclusivité observés précédemment.

Modélisation des composants

Dans nos modélisations, nous avons choisi de simplifier l'architecture afin de nous concentrer sur les échanges de données. Pour cela, nous ne modélisons ni les instructions ni les registres du processeur, mais uniquement les requêtes de lecture et d'écriture qu'il effectue vers son cache L1.

Réalisation d'une plate-forme monoprocesseur sans mémoire du cache

La réalisation NuSMV de cette partie se trouve dans le fichier `mono_proc_simple.smv`.

Définition des signaux d'interface des éléments

Mémoire

La mémoire communique avec les autres composants via le bus avec :
- `VALID` : Indique qu'une valeur significative est envoyée par la mémoire. Lorsque ce signal est à `true`, la mémoire émet une valeur sur le bus.
- `out` : Bus de réponse de la mémoire. Peut valoir `0` ou `1` lors d'une requête de lecture, et `ACK` lors d'une requête d'écriture.

Processeur (CPU)

Le processeur communique avec le cache via :
- `req` : Type de requête émise par le processeur. Peut prendre les valeurs :
  - `NONE` : Aucune requête.
  - `CPU_READ` : Demande de lecture.
  - `CPU_WRITE` : Demande d'écriture.
- `address` : Adresse pour la requête (lecture ou écriture). Peut valoir `0` ou `1`.
- `data` : Valeur de la donnée à écrire. Peut valoir `0` ou `1`.

Arbitre

L'arbitre gère les accès au bus entre le processeur, le cache et la mémoire via le signal `gnt`. Ce signal spécifie quel composant peut écrire sur le bus :
- `gnt` : Peut valoir `MEM` (mémoire) ou `L1_1` (cache L1).

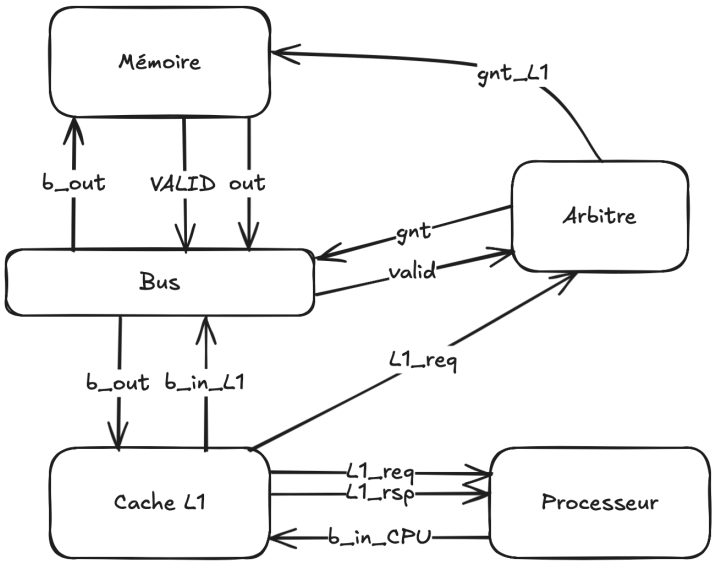
Bus

Le bus sert d'interface de communication entre les différents composants via :
- `VALID`: correspond au signal `VALID` de la mémoire.
- `address` : Adresse pour une lecture / écriture. Peut valoir `0` ou `1` (utilisé uniquement par le cache).
- `data` :
  - Si le cache envoie une requête : la valeur de la donnée à écrire (`0` ou `1`).
  - Si la mémoire répond : la valeur de la réponse (`0`, `1` ou `ACK`).
- `ctrl` : Type de requête envoyée sur le bus par le cache. Peut valoir `BUS_READ` (lecture) ou `BUS_WRITE` (écriture).

Cache L1

Le cache communique avec le processeur et les autres composants via :
- `rsp` : Réponse envoyée par le cache au processeur. Peut valoir `NONE`, `0`, `1` ou `ACK`.
- `address` : Adresse pour une lecture / écriture. Peut valoir `0` ou `1`.
- `data` : Valeur de la donnée à écrire. Peut valoir `0` ou `1`.

Définition des intéractions entre les composants



Nous avons restreint le signal `gnt` émis par l'arbitre vers la mémoire pour qu'il indique uniquement si un cache L1 envoie une requête sur le bus, on le nomme `gnt_L1`.
De plus, nous avons ajouté le signal `valid` entre le bus et l'arbitre. Sachant que la mémoire répond en un seul cycle, ce signal simplifie l'état interne de l'arbitre en évitant de devoir gérer un état supplémentaire indiquant que la mémoire est en cours de traitement.

Propriétés

Nous avons défini plusieurs propriétés qui servent à vérifier le bon fonctionnement de notre plateforme :

1. Une requête du processeur entraîne une requête du cache L1 et une réponse de la mémoire et du cache.
   - `SPEC AG ((cpu.req != NONE) -> AF(L1.req & AF(bus.valid & L1.rsp != NONE)))`
2. Le processeur émet une requête et l'arbitre accorde l'accès au bus au cache L1
   - `SPEC AG ((cpu.req != NONE & !cpu.busy) -> AF(arbiter.gnt = L1_1))`
3. Lorsqu'une requête est émise par le processeur et qu'une réponse lui a déjà été donnée, `L1_req` passe à faux pendant 1 cycle, puis revient à vrai.
   - `SPEC AG ((cpu.req != NONE & prev_valid) -> (!L1.req & AX(L1.req & AF(!L1.req))))`
4. Si une requête de lecture est effectuée à l'adresse 0, la mémoire répond avec la valeur de l'adresse 0.
   - `SPEC AG ((cpu.req = CPU_READ & cpu.address = 0) -> AF(memory.out = memory.data[0] & AF(L1.rsp = memory.data[0])))`
5. Si une requête de lecture est effectué par le CPU à l'adresse 0, le cache L1 fera également une requête de lecture à l'adresse 0.
   - `SPEC AG ((cpu.req = CPU_READ & cpu.address = 0) -> AF(L1.state = L1_READ & L1.address = 0))`
6. Si une requête d'écriture de la valeur 1 est effectuée à l'adresse 0, la mémoire doit contenir la valeur 1 à l'adresse 0.
   - `SPEC AG ((cpu.req = CPU_WRITE & cpu.address = 0 & cpu.data = 1) -> AF(memory.data[0] = 1))`
7. Si une requête d'écriture est effectuée, la mémoire renvoie un ACK au cache et le cache renvoie un ACK au CPU.
   - `SPEC AG ((cpu.req = CPU_WRITE) -> AF(memory.out = ACK & AF(L1.rsp = ACK)))`
8. Si une requête d'écriture de la valeur 0 est effectuée à l'adresse 0, le cache effectue également une requête d'écriture de la valeur 0 à l'adresse 0.
   - `SPEC AG ((cpu.req = CPU_WRITE & cpu.address = 0 & cpu.data = 0) -> AF(L1.state = L1_WRITE & L1.address = 0 & L1.data = 0))`

9. Si le signal VALID est vrai, le cache effectue une requête et celle-ci sera terminée au cycle suivant.
   - `SPEC AG (bus.valid -> (L1.req & AX(!L1.req)))`
10. Si le bus est accordé au L1, alors, avant et après le cycle où l'accès est accordé, le bus est à la mémoire.
    - `SPEC AG (AX(arbiter.gnt != MEM) -> (arbiter.gnt = MEM & AX(AX(arbiter.gnt = MEM))))`
11. Si une requête d'écriture de la valeur 1 à l'adresse 0 est suivie d'une requête de lecture, alors le CPU recevra la valeur 1.
    - `SPEC AG ((cpu.req = CPU_WRITE & cpu.address = 0 & cpu.data = 1) -> AX(AF((cpu.req = CPU_READ & cpu.address = 0) -> AX(AF(L1.rsp = 1)))))`
12. Si l'arbitre de bus accorde l'accès au cache, alors le bus utilise l'adresse et les données provenant de `b_in_L1_1`.
    - `SPEC AG ((arbiter.gnt = L1_1) -> (L1.address = bus.address & (L1.data = 1 -> bus.data = 1) & (L1.data = 0 -> bus.data = 0) & (L1.state = L1_READ -> bus.ctrl = BUS_READ) & (L1.state = L1_WRITE -> bus.ctrl = BUS_WRITE)))`
13. Si l'arbitre de bus accorde l'accès à la mémoire, alors le bus a le signal VALID et les données provenant de `b_in_mem`.
    - `SPEC AG ((arbiter.gnt = MEM & memory.valid) -> (bus.valid & (memory.out = bus.data))`

Réalisation d'une plate-forme monoprocesseur avec mémoire du cache

La réalisation NuSMV de cette partie se trouve dans le fichier `mono_proc_simple.smv`

Signaux d'interface

Nous avons utilisé la réalisation précédente, que nous avons modifiée pour permettre le stockage d'une valeur dans le cache.

Nous avons ajouté la modélisation du fil `address` entre la mémoire et le bus. Cette modification nous permet de tester, dans le cas d'une demande d'écriture, si la mémoire envoie un ACK en fonction de l'adresse. Si l'adresse correspond à une adresse déjà présente dans le cache, l'ACK n'est pas propagé au processeur. En revanche, si l'adresse ne correspond pas à celle du cache, l'ACK est propagé.

Propriétés

Nous avons ajusté les propriétés précédentes, car certaines ne convenaient plus. Par exemple, il n'est plus obligatoire de faire une requête à la mémoire lorsque la valeur est déjà présente dans le cache. C'est pour cela que certaines propriétés ont été remplacées par des variantes qui spécifient qu'un chemin avec requête vers la mémoire existe, mais qu'il n'est pas toujours obligatoire (EF au lieu de AF).
De plus, nous avons ajouté des spécifications pour vérifier que le cache avec la mémoire fonctionne correctement :

1. Si une requête de lecture est effectuée à l'adresse 0, alors le cache doit contenir la valeur de l'adresse 0.
   - `SPEC AG ((cpu.req = CPU_READ & cpu.address = 0) -> AF(L1.word_address = 0))`
2. **Lecture avec donnée présente dans le cache** : Si une requête de lecture est effectuée à l'adresse 0 et que le mot est présent dans le cache, alors la valeur du mot est renvoyée par le cache.
   - `SPEC AG ((cpu.req = CPU_READ & cpu.address = L1.word_address & !L1.req) -> (L1.rsp = L1.word_data))`

3. **Écriture avec donnée présente dans le cache** : Si une requête d'écriture du mot 1 est effectuée à l'adresse présente dans le cache, la réponse est reçue au même cycle. La valeur 1 sera stockée dans le cache et une nouvelle requête d'écriture sera lancée.
   - `SPEC AG ((cpu.req = CPU_WRITE & cpu.address = L1.word_address & cpu.data = 1 & !L1.req) -> (L1.rsp = ACK & AF(L1.word_data = 1 & L1.req)))`
4. **Écriture avec donnée non présente dans le cache** : Si une requête d'écriture est lancée à l'adresse d'un mot qui n'est pas présent dans le cache, le processeur sera gelé. Le cache lancera une requête d'écriture, et l'arbitre permettra au cache d'écrire sur le bus. Une réponse sera ensuite reçue à la fois du cache et de la mémoire.
   - `SPEC AG ((cpu.req = CPU_WRITE & cpu.address != L1.word_address & !cpu.busy) -> AF(L1.state = L1_WRITE & AF(arbiter.gnt = L1_1 & AF(bus.valid & L1.rsp = ACK))))`
5. **Écriture successive de 1 puis 0 à l'adresse 0** : Si une requête d'écriture de la valeur 1 est effectuée à l'adresse 0 avec le mot dans le cache, puis que la même requête avec la valeur 0 est effectué juste après, la première requête se fait en 1 cycle, puis la deuxième en plus de 1 cycle, et la mémoire écrira d'abord 1, puis 0.
   - `SPEC AG ((cpu.req = CPU_WRITE & cpu.address = 0 & L1.word_address = 0 & cpu.data = 1 & ! L1.req) ->`
     `(cpu.busy & AX((cpu.req = CPU_WRITE & cpu.address = 0 & cpu.data = 0)`
     `-> (!cpu.busy & AF(memory.data[0] = 1 & AF(memory.data[0] = 0))))))`
6. **Écriture suivit de lecture à l'adresse 0** : Si une requête d'écriture de la valeur 1 est effectuée à l'adresse 0 avec le mot dans le cache, puis une requête de lecture est effectuée à la même adresse, l'écriture se fait en 1 cycle, suivie de la lecture en plus de 1 cycle, et la valeur renvoyée par le cache L1 est 1.
   - `SPEC AG ((cpu.req = CPU_WRITE & cpu.address = 0 & L1.word_address = 0 & cpu.data = 1 & ! L1.req) ->`
     `(cpu.busy & AX((cpu.req = CPU_READ & cpu.address = 0)`
     `-> (!cpu.busy & L1.rsp = NONE & AF(L1.rsp = 1)))))`

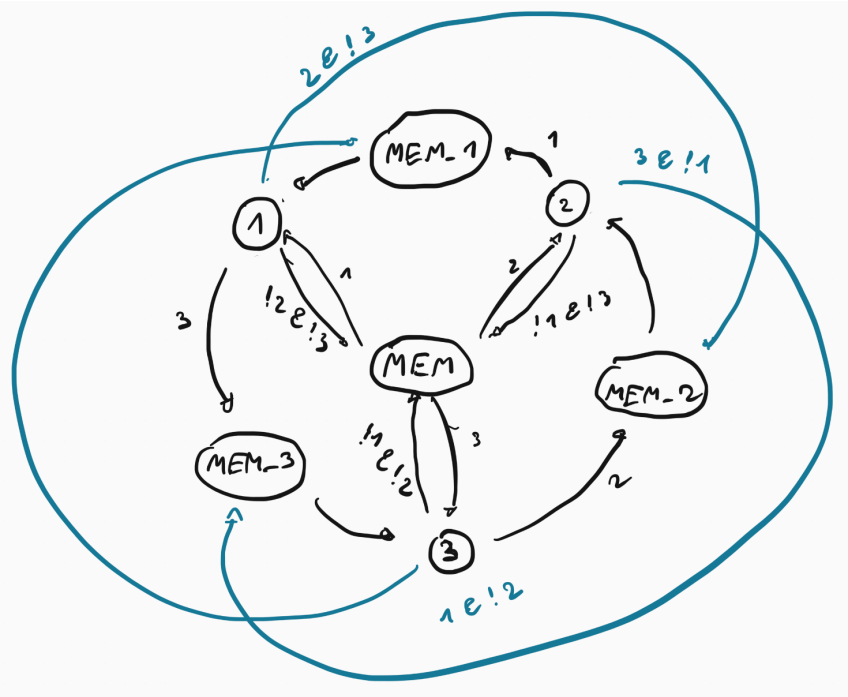
Réalisation d'une plate-forme multi processeurs

Conception d'un arbitre équitable

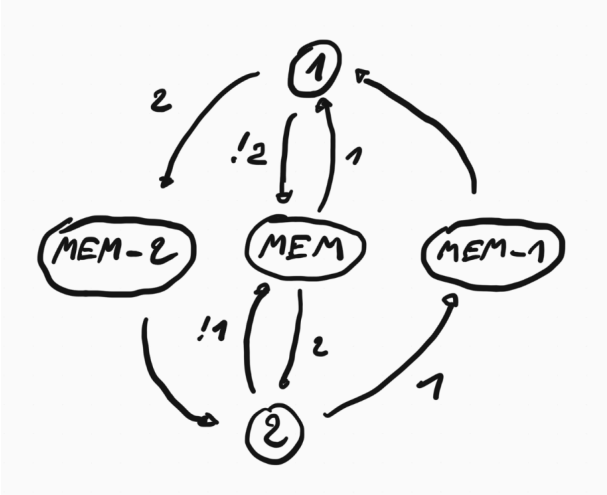
Dans un premier temps, nous avons défini un arbitre non équitable. Cependant, lors des simulations et de la vérification des propriétés avec NuSMV, des contre-exemples ont montré qu'un des processeurs pouvait rester bloqué en situation de famine et donc jamais être servi. Cela rendait nos spécifications incorrectes.

Pour corriger ce problème, nous avons défini un arbitre équitable. Cet arbitre garantit que tout les caches L1 seront sélectionnés s'ils émettent une requête. Ce fonctionnement est possible dans notre cas, de part le protocole, où un cache L1 garde son signal actif jusqu'à être sélectionné et que sa requête soit terminée.

L'implémentation se base sur l'ajout d'états intermédiaires, appelés `MEM_1`, `MEM_2` et `MEM_3`. Ces états représente un retour a l'état `MEM`, mais avec la garantie de choisir ensuite un cache L1 différent, assurant ainsi que leur requêtes soient prise en compte à terme. L'automate d'états est illustré dans la figure suivante.



Une description de l'automate pour deux processeurs est présentée dans la figure suivante. Ce choix a été nécessaire car la simulation avec trois processeurs étant trop lente en raison d'un phénomène d'explosion combinatoire. Ainsi, nous avons dans un premier temps décidé de vérifier en pratique nos spécifications sur une architecture à deux processeurs.



Malgrès cela, nous avons testé tout de même l'architecture à 3 processeurs en utilisant la simulation interactive avec l'utilisation de contraintes. Par exemple, en utilisant `simulate -r -k 20 -c "cpu_1.req = CPU_WRITE & cpu_1.data = 1"`, on peut forcer le premier processeur à faire une requête d'écriture, puis vérifier que `L1_1` est élu et que la mémoire est bien modifiée dans la trace résultante.

Dernier cache sélectionné

Nous avons ajouté une variable dans l'arbitre pour indiquer l'identifiant du dernier cache L1 sélectionné. Cela nous permet de déterminer, que lorsqu'une requête est reçue, la donnée est bien destinée au cache qui a émis la requête (en pratique `arb_gnt_me` dans chaque cache L1). Par exemple, si deux caches, `L1_1` et `L1_2`, émettent simultanément une requête de lecture, il est nécessaire de pouvoir déterminer à qui la réponse est envoyée.

Priorité avec le snoop

Lors de la modélisation du snoop, nous avons identifié un problème de priorité entre une requête d'écriture effectuée par le processeur et le mécanisme de snoop. En effet, ce problème survient lorsqu'une écriture est effectué sur un mot présent dans le cache simultanément qu'une requête d'écriture en cours sur le bus.

Dans ce cas, nous avons décidé de prioriser la requête du processeur car elle sera diffusé plus tard sur le bus, garantissant donc la cohérence du système.

Propriétés

Nous avons encore une fois ajusté les propriétés précédentes. Cette fois, nous avons modifié les spécifications pour qu'elles prennent en compte la présence de plusieurs processeurs. Par ailleurs, nous avons intégré les états ou l'arbitre laisse le bus à la mémoire (`MEM`, `MEM_1`, `MEM_2`, etc..) en ajoutant le signal `is_mem`, utilisé dans nos propriétés pour les simplifier.

Finalement, nous avons ajouté des spécifications pour vérifier l'absence de famine et le mécanisme de snoop :

1. **Equité** : Si les processeurs émettent une requête simultanément, il recevront tous une réponse à terme.
   - Pour 2 processeurs :
     - `SPEC AG (AG AF(AX (L1_1.req) & !L1_1.req & AX(L1_2.req) & !L1_2.req) -> (AG AF(L1_1.rsp != NONE) & AG AF(L1_2.rsp != NONE)))`
   - Pour 3 processeurs :
     - `SPEC AG (AG AF(AX (L1_1.req) & !L1_1.req & AX(L1_2.req) & !L1_2.req & AX(L1_3.req) & !L1_3.req) -> (AG AF(L1_1.rsp != NONE) & AG AF(L1_2.rsp != NONE) & AG AF(L1_3.rsp != NONE)))`
2. **Mécanisme de snoop** : Si une requête d'écriture est effectuée sur le bus, la valeur dans le cache L1 sera mise à jour.
   - `SPEC AG ((bus.ctrl = BUS_WRITE & cpu_1.req != CPU_WRITE & bus.address = L1_1.word_address & bus.data = 1) -> AX (L1_1.word_data = 1))`