# QUESTIONS sur le TP9 / Synchronisation dans les applications parallèles coopératives

## Q1) Pourquoi le mécanisme de synchronisation par bascule SET/RESET ne nécessite-t-il pas d'instructions assembleur spéciales, autres que les instruction *lw* et sw *?*

Dans ce mécanisme, il y a un seul producteur, un seul consommateur, et le tampon de communication ne peut contenir qu'une seule donnée. Le tampon ne peut donc être que dans deux états : PLEIN ou VIDE, et ce tampon a donc toujours un seul propriétaire (le producteur quand il est vide / le consommateur quand il est plein).

La conséquence est qu'il ne peut pas exister d'accès concurrents en écriture sur ce tampon puisque seul le propriétaire (unique) du tampon a le droit de modifier l'état du tampon. L'inconvénient de ce mécanisme est qu'il ne peut pas exister de parallélisme entre la tâche productrice et la tâche consommatrice : il y a toujours une des deux tâches qui doit attendre l'autre.

## Q2) Pourquoi la fonction atomic_increment(), utilisée en particulier par la fonction de prise du verrou à ticket, a-t-elle absolument besoin d'instructions assembleur spéciales qui garantissent une *lecture_puis_écriture_atomique_à_la_même_adresse* ?

Pour incrémenter un compteur stocké à une adresse X (par exemple un distributeur de ticket), il faut exécuter au moins trois instructions assembleur : (i) lire la valeur courante à l'adresse X : *lw*, (ii) incrémenter cette adresse : *add*, (iii) écrire la valeur à l'adresse X : *sw*. Cette séquence peut être exécutée de façon concurrente par deux tâches A et B ce qui peut donner la séquence suivante pour les transactions entrelacées sur le bus: lw_A, puis lw_B, puis sw_A, puis sw_B. Cette séquence donnera la même valeur de ticket aux deux tâches A et B, ce qui est un résultat faux.

On a donc absolument besoin d'un mécanisme qui garantit a chaque tache qu'il n'y a pas eu d'autre accès en écriture à l'adresse X entre sa propre lecture, et sa propre écriture à l'adresse X. C'est cette garantie qu'apporte les deux instructions *ll* et *sc* du MIPS32 .

## Q3) Dans le mécanisme *ll/sc* l'instruction ll(X) effectue une lecture de la donnée stockée en mémoire à l'adresse X, et en plus, met sous surveillance de l'adresse X (appelée réservation). L'instruction *sc(X)* effectue une écriture conditionnelle à l'adresse X, et en plus, retourne un Booléen indiquant si l'écriture est un succès ou un échec. Quels sont les événements qui mettent fin à la surveillance de l'adresse X *?*

Lorsqu'une adresse X a été mise sous surveillance, par une (ou plusieurs) instruction(s) *ll(X)*, n'importe quelle écriture à l'adresse X casse la réservation, quel que soit l'écrivain. On ne distingue donc pas les instructions *sw(X)* et *sc(X)*. Le plus souvent c'est le premier concurrent qui exécute une instruction *sc(X)* qui gagne le concours et casse la réservation.

## Q4) == Comment les instructions *ll* et *sc* sont-elles implémentées dans le matériel matériel ?

Conceptuellement, la mise sous surveillance d'une adresse X devrait être réalisée par le contrôleur mémoire, qui centralise toutes les requêtes de lecture ou d'écriture.

Mais en pratique, dans les architectures à base de bus permettant le scoop (c'est à dire la possibilité pour n'importe quel maître de surveiller ce que font les autres maîtres), l'enregistrement des requêtes de mise sous surveillance peur être distribuée dans chacun des contrôleurs de cache L1, et c'est ce qui est fait par le contrôleur de cache L1 utilisé dans l'UE Multi, comme cela peut être vérifié en analysant le comportement des 4 automates contenus dans ce composant matériel.

Plus précisément, l'exécution d'une instruction *ll(X)* par un coeur se traduit par une requête de type LL vers l'automate DCACHE_FSM du contrôleur de cache. Cet automate enregistre l'adresse X dans un registre spécifique LLSC_ADDR, ainsi que la validité de cette requête dans une bascule LLSC_VALID, envoie une requête de lecture non cachable vers l'automate PIBUS_FSM, et gèle le coeur en attendant la réponse de la mémoire. Si l'automate SNOOP_FSM détecte qu'un autre coeur effectue une écriture à une adresse X enregistrée dans les registres LLSC_ADDR et LLSC_VALID, celle ci est invalidée. Par ailleurs, l'exécution d'un instruction *sc(X)* par un coeur se traduit par une requête de type LL vers l'automate DCACHE_FSM du contrôleur de cache. Si il existe une requête LLSC valide dans les registres

LLSC_ADDR et LLSC_VALID, c'est un succès, et cet automate envoie une requête d'écriture vers l'automate PIBUS_FSM. Sinon, l'automate DCACHE_FSM retourne directement une réponse d'échec au coeur demandeur.

### Q5) L'instruction assembleur *sync* empêche le re-ordonnancement des instructions par le processeur lui-même. Plus précisément toutes les instructions de lecture ou d'écriture en mémoire placées avant l'instruction *sync* sont garanties être effectivement exécutée AVANT l'exécution de la première instruction de lecture/écriture placée après l'instruction sync. Comment l'instruction assembleur *sync* est-elle implémentée dans le matériel ?

Pour ce qui concerne le coeur MIPS32 utilisé dans l'UE Multi, ce coeur exécute les instructions du code binaire en respectant strictement l'ordre des instructions (au contraire d'autres implémentations qui autorisent une exécution dans le désordre.

Pour ce qui concerne le contrôleur de cache L1 utilisé dans l'UE Multi, il garantit que les requêtes de lectures de données qui font MISS ne seront jamais traitées avant les requêtes d'écritures qui sont en attente dans le tampon d'écritures postées. Pour traiter une instruction assembleur *sync*, l'automate DCACHE_FSM qui traite les requêtes concernant le cache de données peut donc se contenter de geler (c'est à dire faire attendre) le coeur jusqu'à ce que toutes les requêtes d'écriture en attente dans le tampon d'écritures postées aient effectivement été traitées et acquittées par l'automate PIBUS_FSM.

*Last modified on 9 Jun 2020, 22:31:00*