Institut
Mines-Télécom

Multi-processeurs, multi-cœurs, cohérence mémoire et cache

Intervenant : Thomas Robert

Parallélisme inter instructions

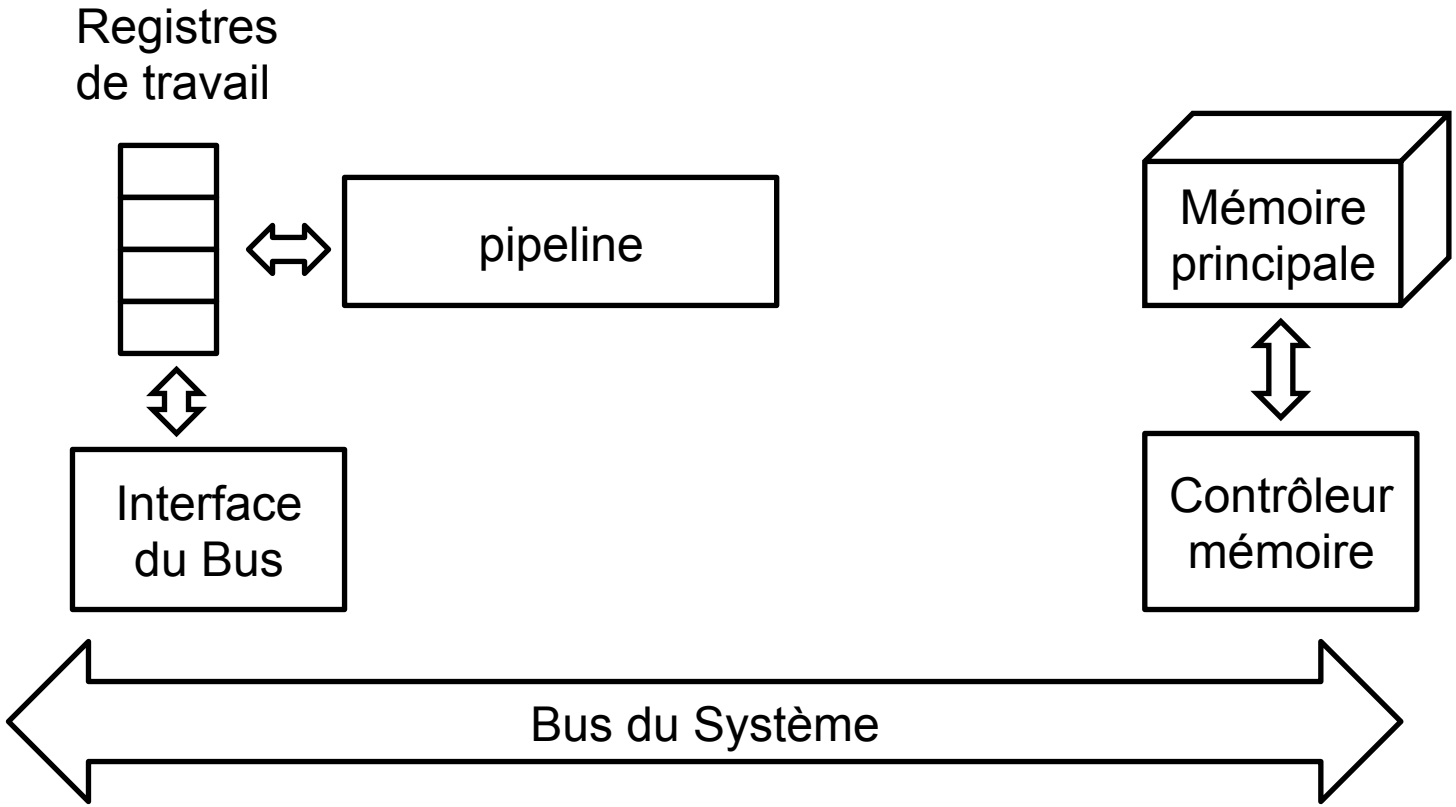
- **Exécution : une séquence d'instructions appliquées à un 1 ensemble de registres + emplacements mémoire**
- **1 instruction =**
  - succession d'étapes => 1 début, 1 fin
  - Parfois avec des lectures/écritures de la mémoire
  - Parfois avec des lectures écritures de registres
- **Objectif des contraintes sur l'ordre d'exécution :**

Maintenir une sémantique séquentielle équivalente

C1 Ordre de terminaison identique à l'ordre d'entrée dans le pipeline

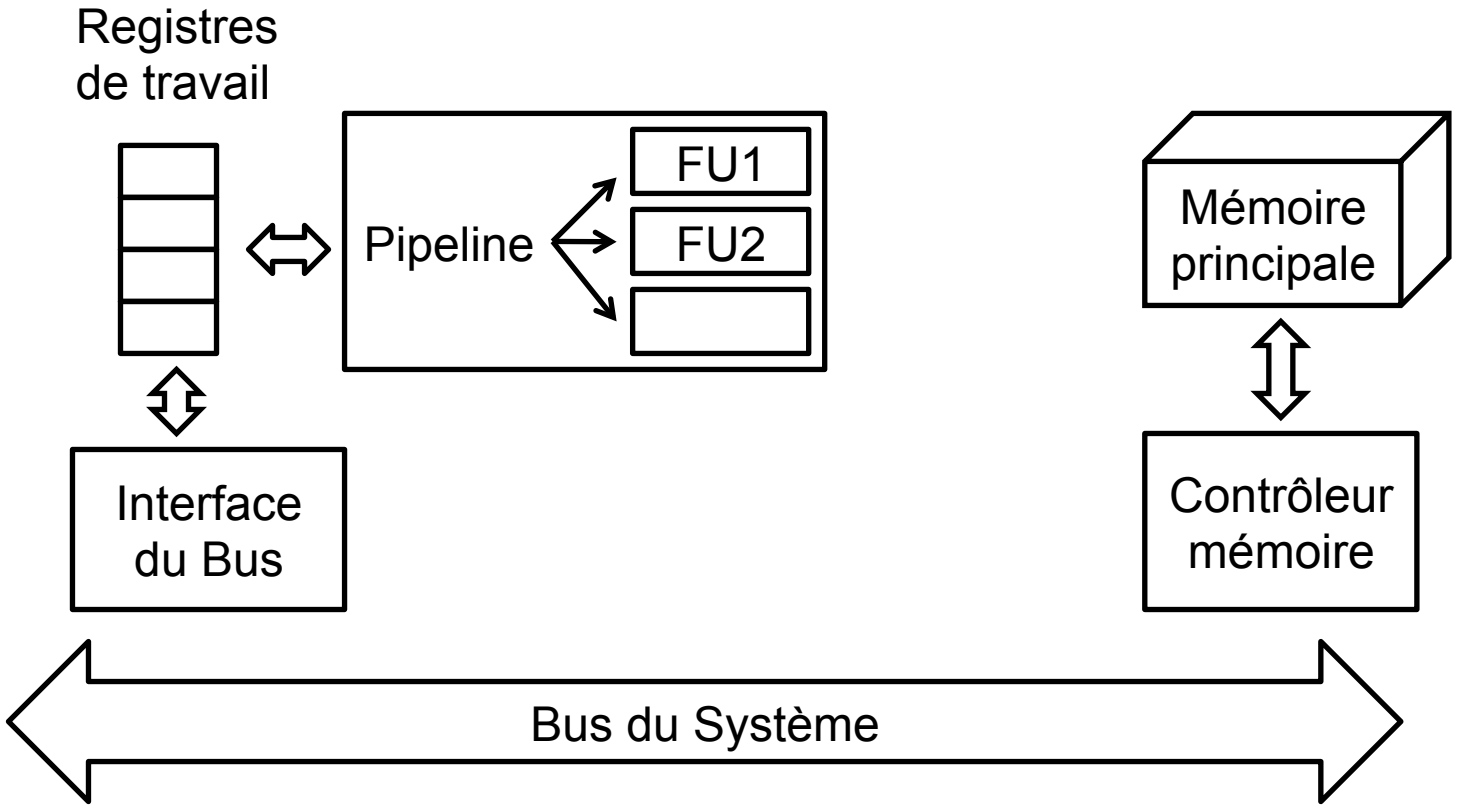
C2 Ordre des accès mémoire identique à celui des instructions

Architecture Pipeline simple (TD2)



C1 + C2

Architecture Monoprocesseur Superscalaire



C2

Classification des processeurs (Flynn)

- **Intérêt : une manière classique d'identifier le comportement d'un processeur**
  - Single Instruction Single Data (SISD) : uni-processeur (1 pipeline)
  - Single instruction Multiple Data (SIMD) : GPU
  - Multiple instruction Single Data (MISD) : processeur dédié aux application très « critiques » (e.g. ordinateurs de bord satellites)
  - Multiple instruction Multiple Data (MIMD): Multi cœurs / multi processeurs

- **Dans la suite nous verrons le dernier cas + le rôle de la mémoire**

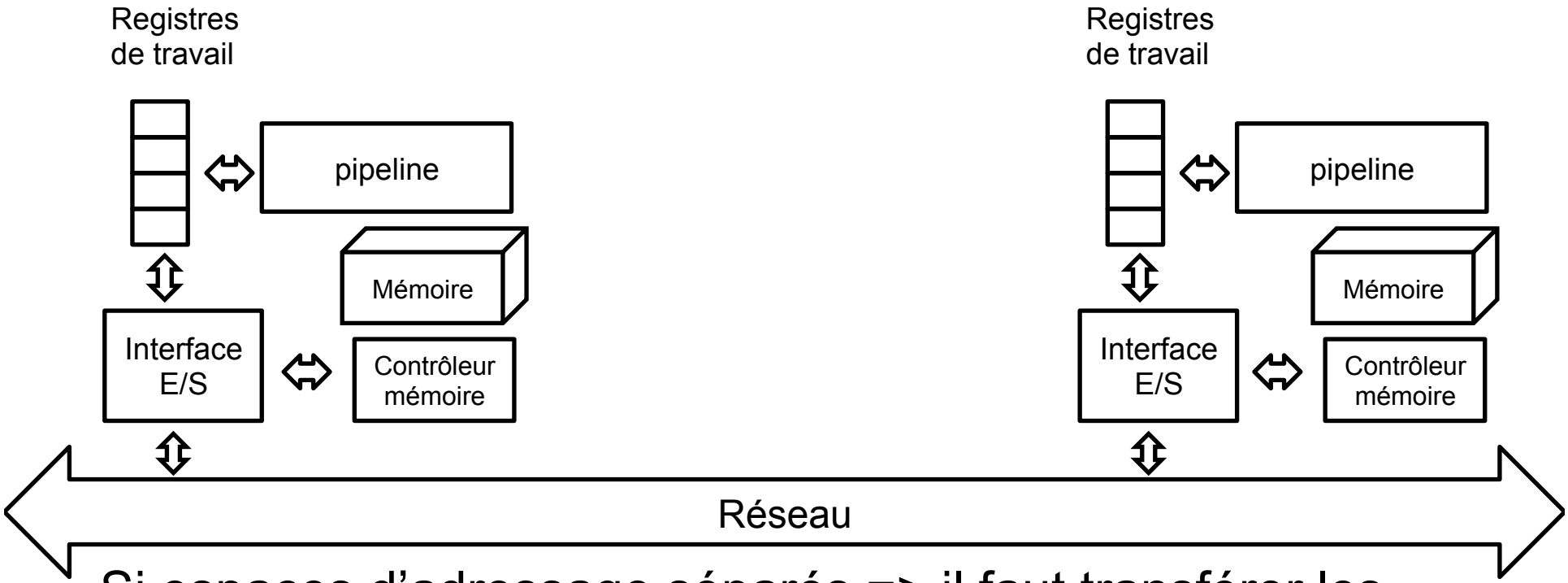
Parallélisme niveau Thread (matériel)

- **Activité = N x (séquence d'instructions + état)**
- **3 Problèmes différents**
  1. Maintenir N états d'exécutions
  2. Échanger des données et assurer la synchronisation
  3. Partager (répartir) les ressources communes matérielles

Solutions matérielles :

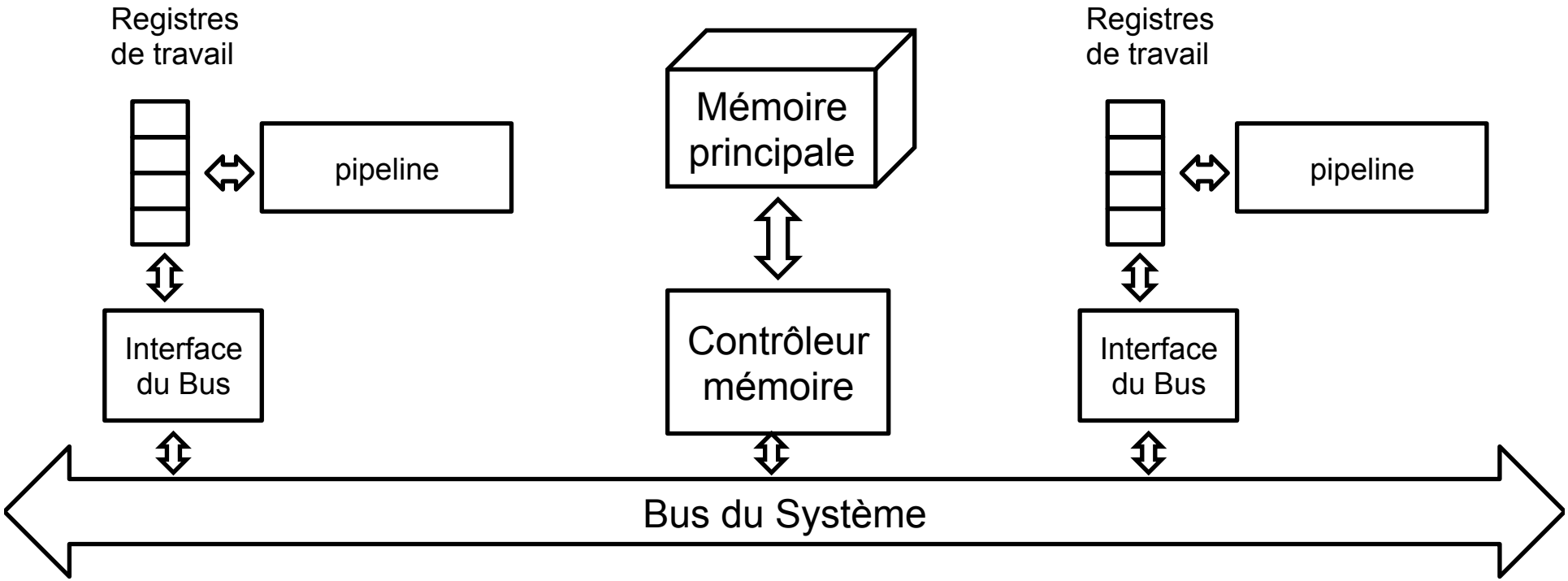
- **Réponse 1 : a) dupliquer les registres (à minima),**
  b) dupliquer les pipelines + registres + …
- **Réponse 2 : la suite de ce cours**
- **Réponse 3 : non traité ici**

Architecture Multi processeur (I)
mémoire distribuée espace d'adresses séparés



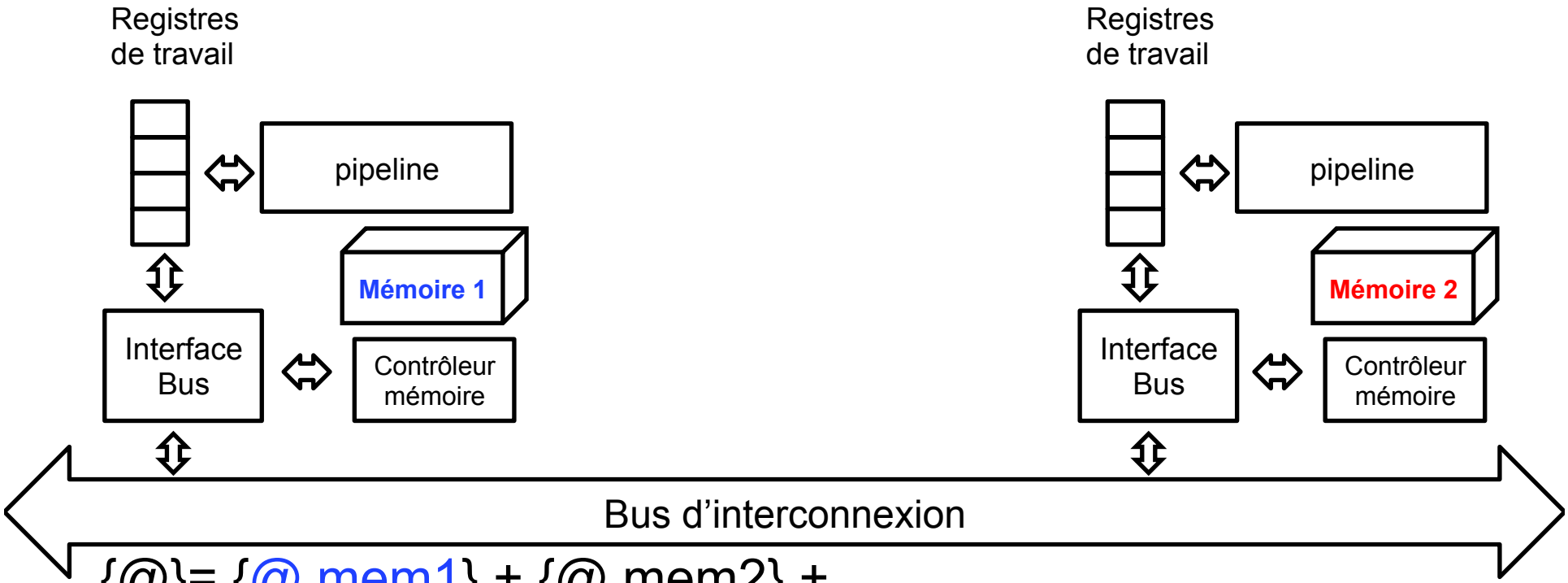
Si espaces d'adressage séparés => il faut transférer les données via un réseau (implique un protocole d'échange)

Architecture Multi processeur (II)
Mémoire centralisée / espace d'adressage partagé



Quelles interactions avec la mémoire ?
1 espace d'adressage :
chaque @ = 1 contenu accessible par tous

Architecture Multi processeur (III)
mémoire distribuée/espace d'adresses partagé



{@}= {@ mem1} + {@ mem2} + …

Architecture dite NUMA : non uniform memory access

Généralisation possible : N bus interconnectés par des ponts

Le problème des accès concurrents à la mémoire

- **Rappel Principe du data race (en général)**
  - 2 activités séquentielles utilisant X comme variable partagée
  - X lue au début de chaque séquence
  - X utilisée pour stocker le résultat

  S: lire X, calculer F(X), écrire X ; S' : lire X, calcul g(X), écrire X

  Si X vaut X0 au début => 4 valeurs possibles:
  f(X0), g(X0), f(g(X0), g(f(X0))
- **Problème concret pour l'entrelacement des instructions (e.g. processeur RISC)**

  A=A+4; devient ld r3,@A; add r3,4,r4; sw r4,@A;

  => 3 instructions pour lire incrémenter et mémoriser A

Synchronisation : test & set (TAS) (Rappel BCI)

- **Variable d'@ X : nb de « programmes » en section critique**
- **Effet test and set : tsl rd, @1**

  Rd← [@1] ; @1← 1 ;
- **Usage classique (naif avec attente active)**

```
tsl r1, X
cmp X,0
bnez loop;
<<section critique>>
sw 0, X:
```

- **Implémentation pour architecture avec 1 bus (nouveau) :**
  - Naif : vérrouiller le bus pendant la résolution du TAS (accès exclusif)
  - Moins Naif : mémoriser les TAS en cours (avec @).

Synchronisation & opération atomiques

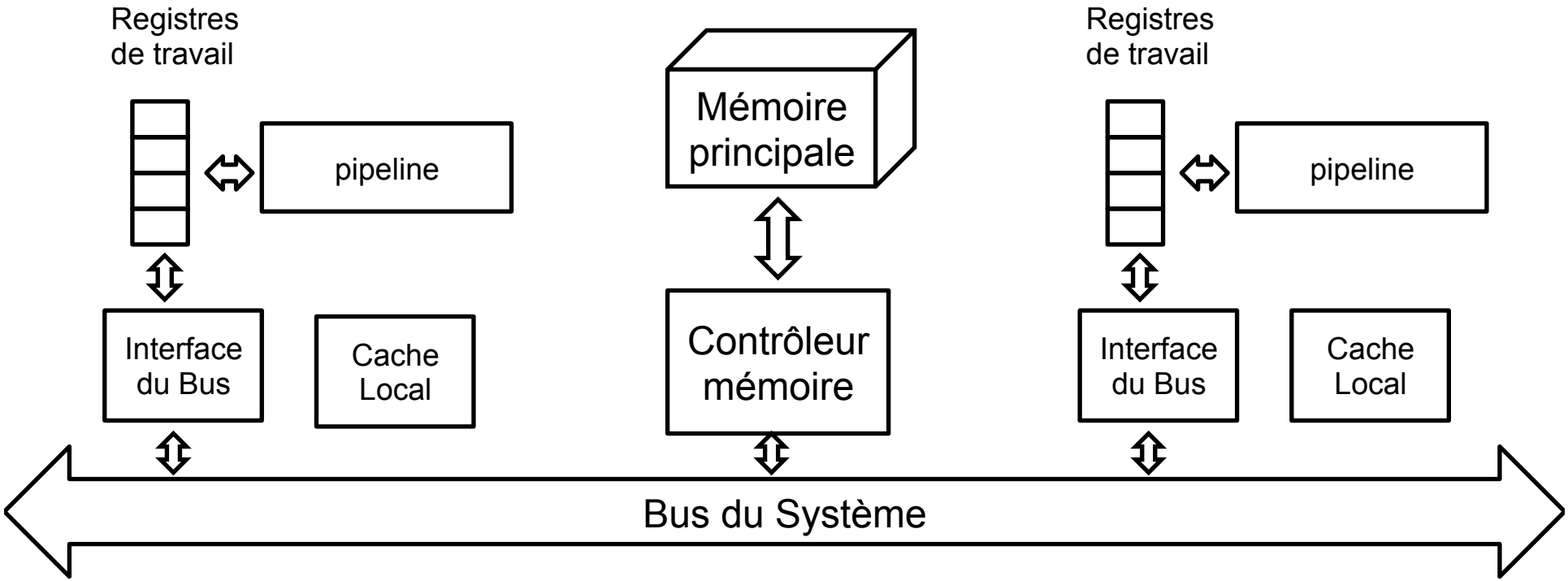
- **Objectifs :**
  - **Éviter des calculs incohérents**
  - **Optimiser l'usage des ressources**
- **Pb n°1 éviter les calculs incohérents**
- **Solution niveau jeu d'instruction**
  = opérations atomiques de lecture / écriture de la mémoire
- **A retenir : + d'une primitive => usages différents**
  - Test and set (1 @)
  - Compare and swap (2 @)
  - Fetch and add
  - Load-link / store conditional
- **PB :**
  hiérarchie mémoire => 1 @ stockée en plusieurs endroits

Rappel sur le fonctionnement du cache : politique d'écriture

- **Unité de stockage dans un cache == un bloc**
- **Si cache de donnée : accès en lecture et écriture**
- **Politique d'écriture différée (write-back)**
  - Les écritures sont stockées dans le cache
  - Les valeurs sont mises à jour en mémoire lorsque le contenu du cache doit en sortir

- **Politique d'écriture synchrone (write through)**
  - Les écritures sont stockées dans le cache
  - Elles sont propagées en mémoire

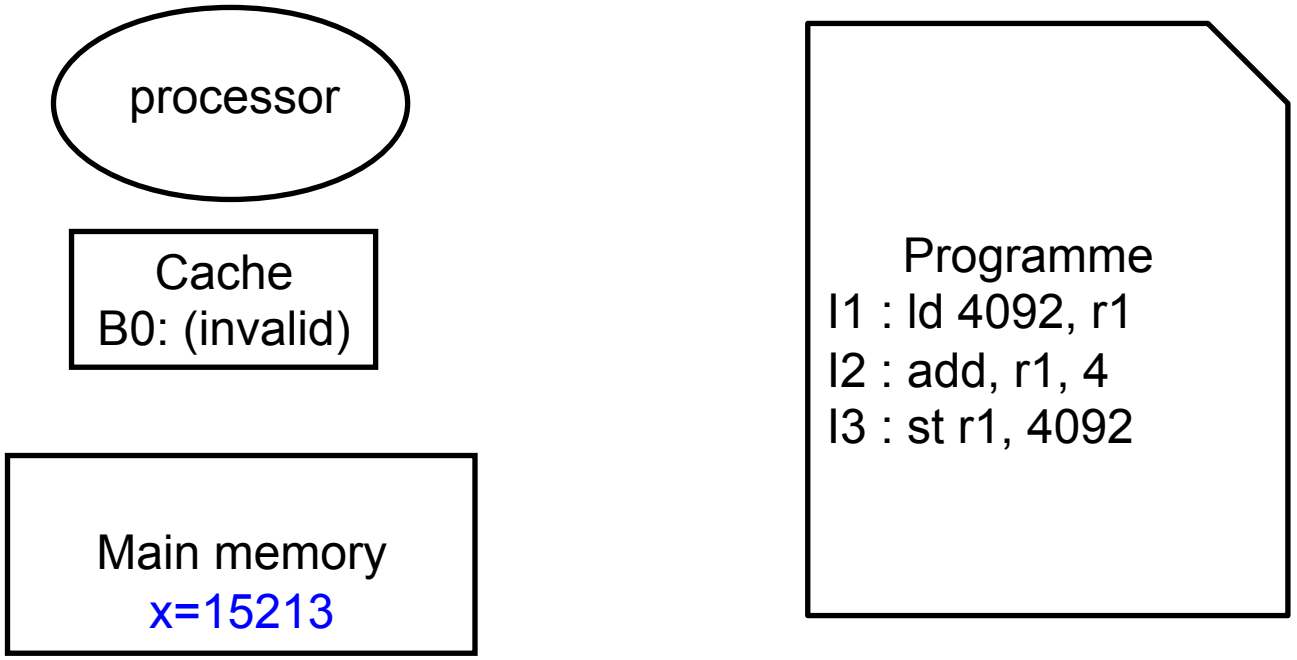
Architecture Multi processeur mémoire partagée avec cache : gestion des copies



Quels problèmes pose les caches ?

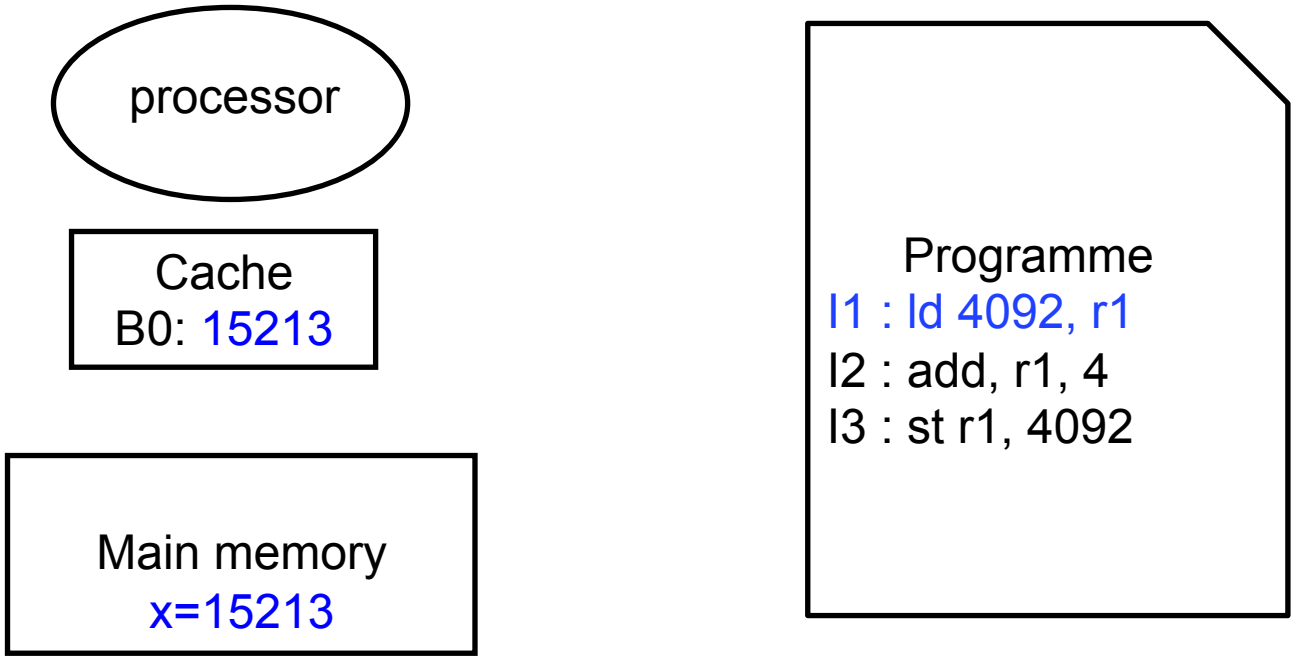
Fonctionnement d'un cache
séquence lire calculer écrire (read modify write)

- **Soit x à l'adresse 4092 + cache direct mapped tel que x va dans dans l'entrée numéro 0 (B0), x occupe tout B0**



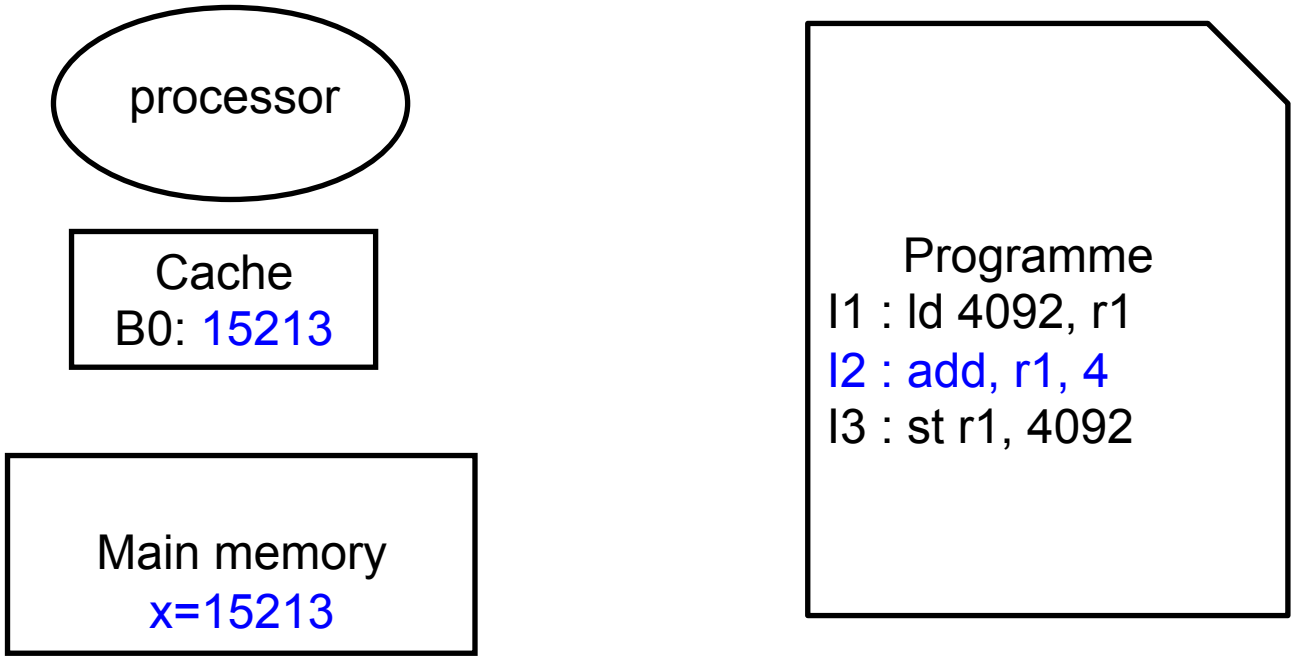
Fonctionnement d'un cache
séquence lire calculer écrire (read modify write)

- **Instructions : ld load, st store**
- **ld déclenche un cache miss qui met à jour L0**



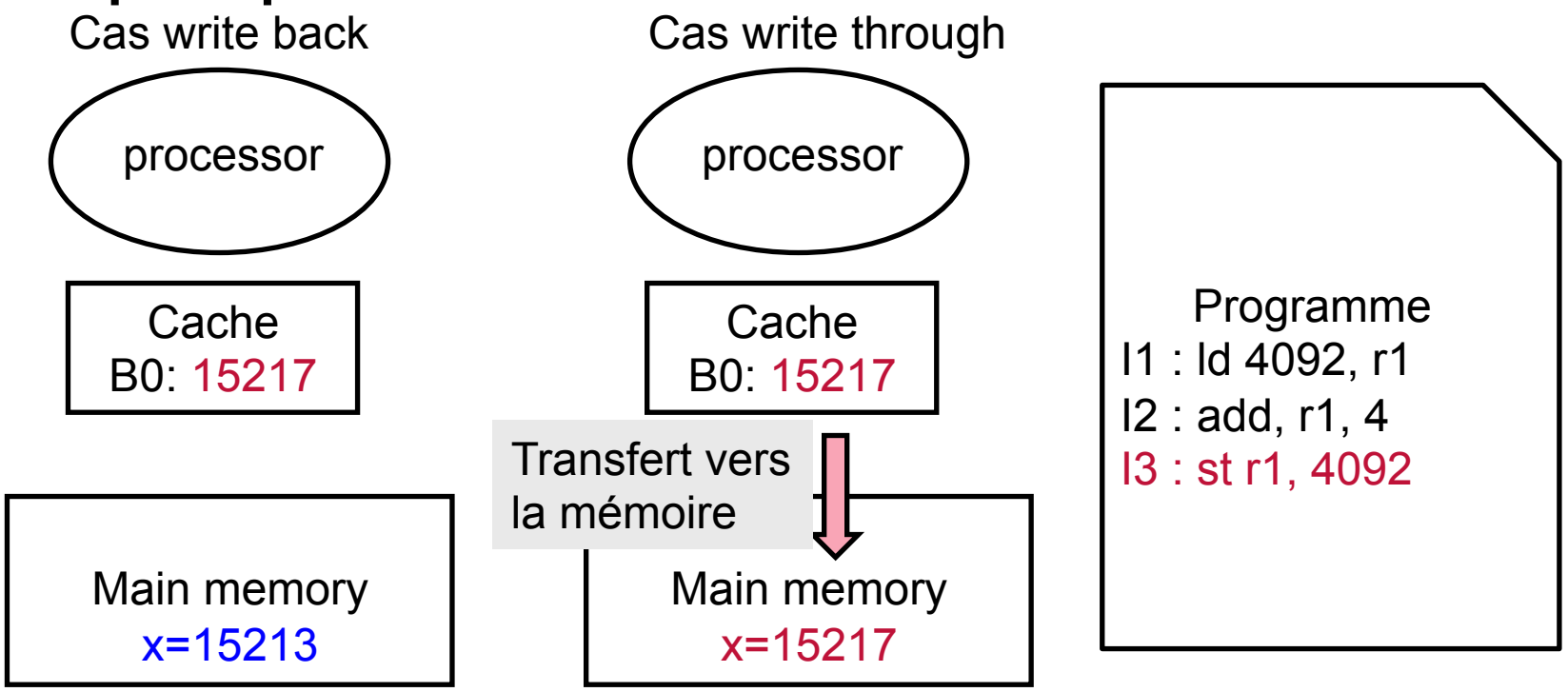
Fonctionnement d'un cache
séquence lire calculer écrire (read modify write)

- **L'instruction add n'a pas d'effet sur le cache**

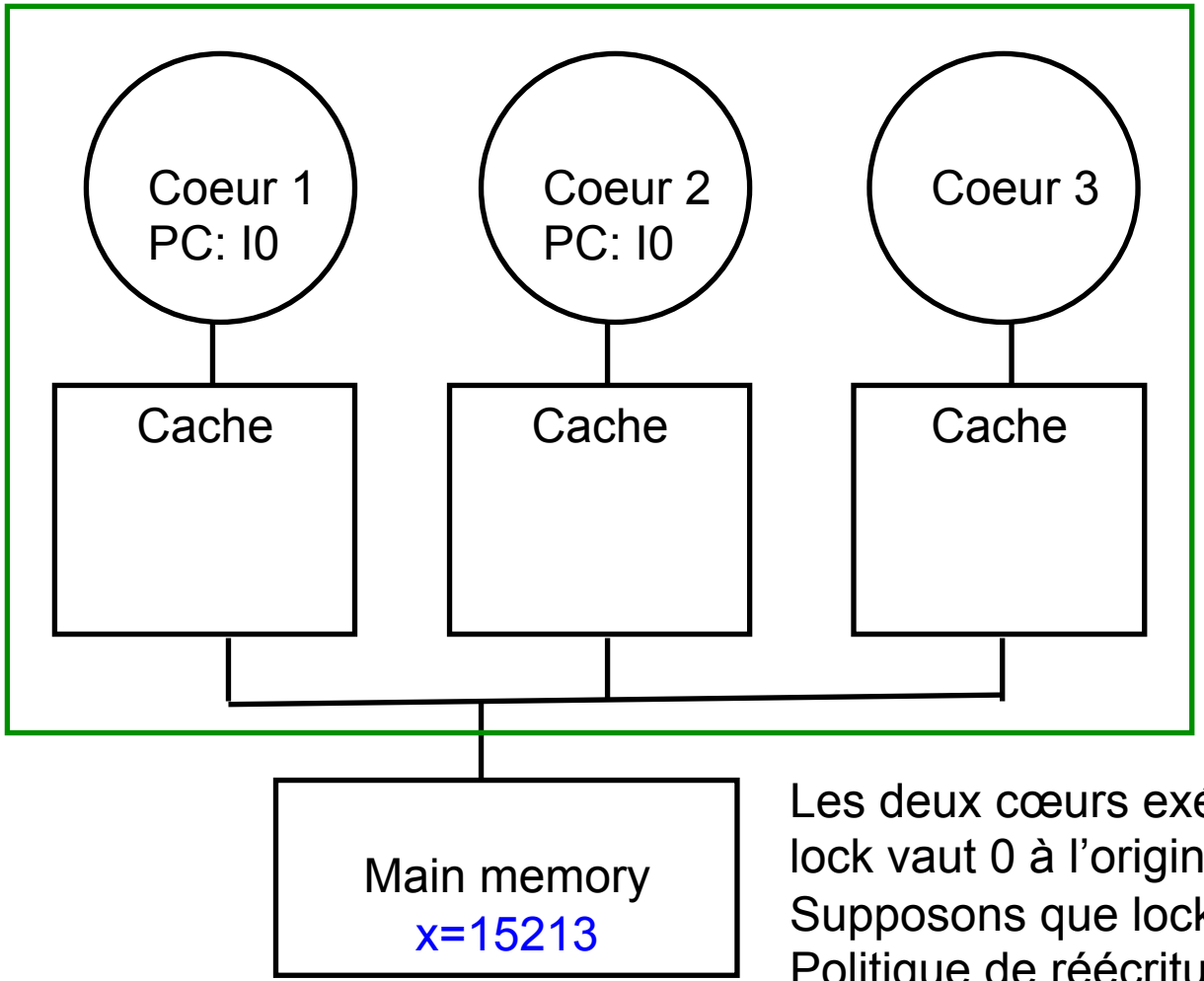


Fonctionnement d'un cache
séquence lire calculer écrire (read modify write)

- **L'instruction st a un effet différent en fonction de la politique**



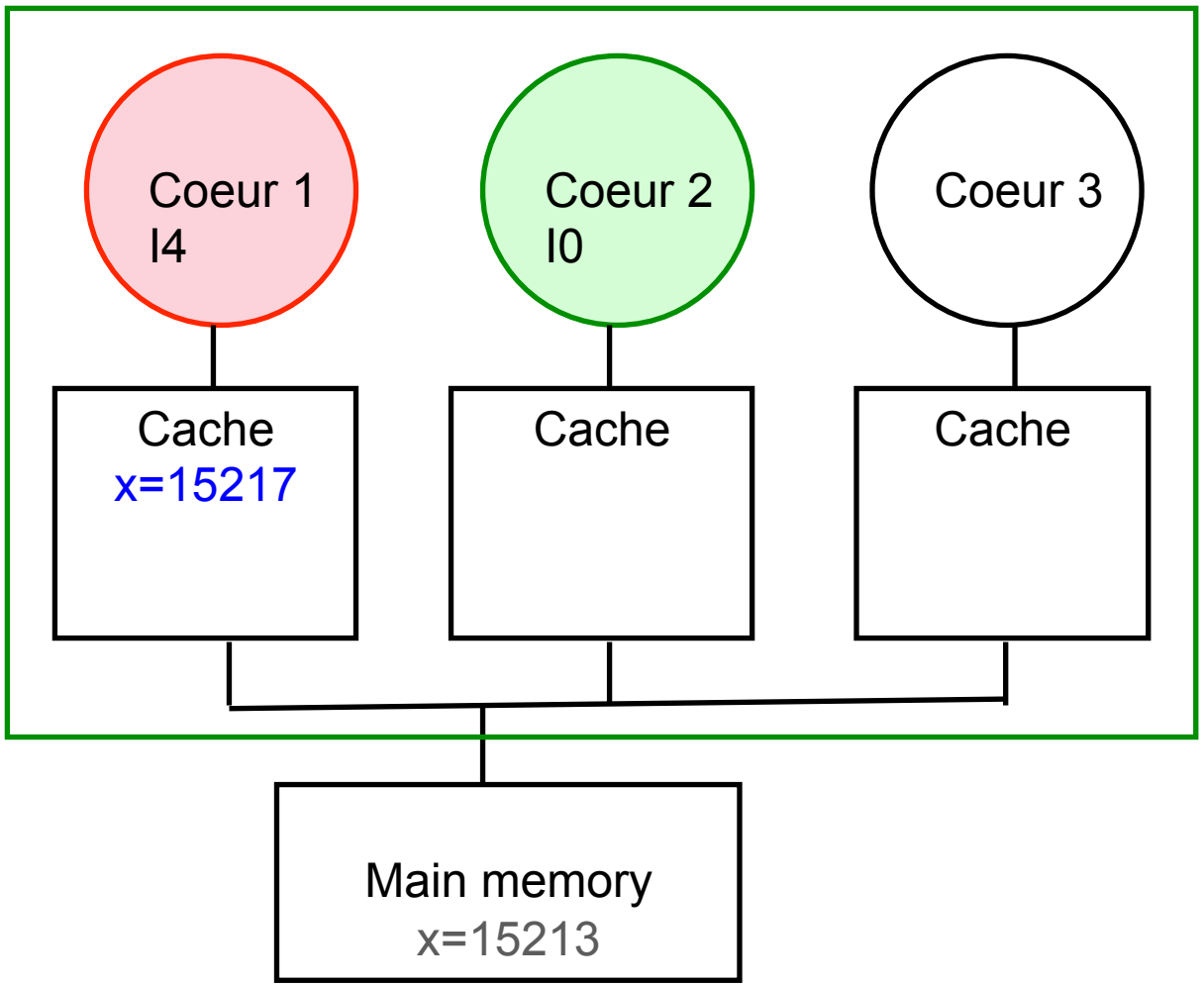
Problème de la cohérence du cache (1) : Avoir un data race alors qu'on utilise des verrous




```
@x = 4092
Programme
I0 : tsl r1, lock
i1 : bnz I0 // jump if ≠0
I2 : ld 4092, r1
I3 : add, r1, 4
I4 : st r1, 4092
I5 : st r0, lock ; //r0=0
I6 : jmp I0
```

Les deux cœurs exécutent *Programme*
lock vaut 0 à l'origine,
Supposons que lock ne va pas dans le cache
Politique de réécriture **: write-back**

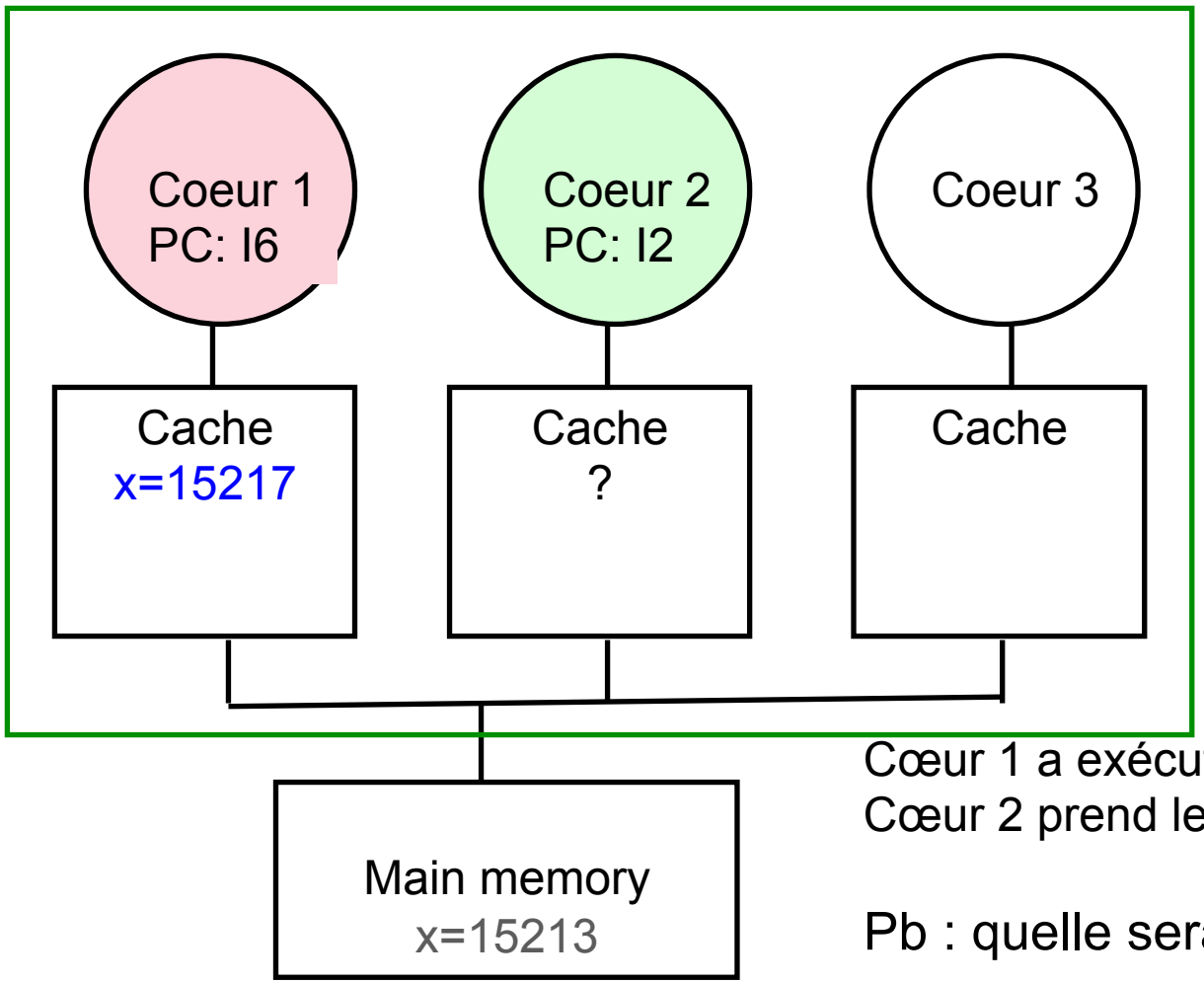
Problème de la cohérence du cache (2) : Avoir un data race alors qu'on utilise des verrous

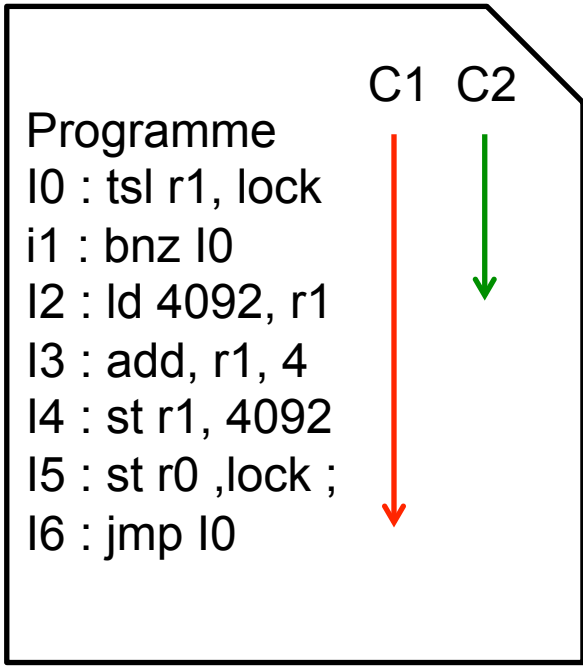


C1 C2

Programme
I0 : tsl r1, lock
i1 : bnz I0
I2 : ld 4092, r1
I3 : add, r1, 4
I4 : st r1, 4092
I5 : st r0 ,lock ; //r0=0
I6 : jmp I0

Problème de la cohérence du cache (3) : Avoir un data race alors qu'on utilise des verrous

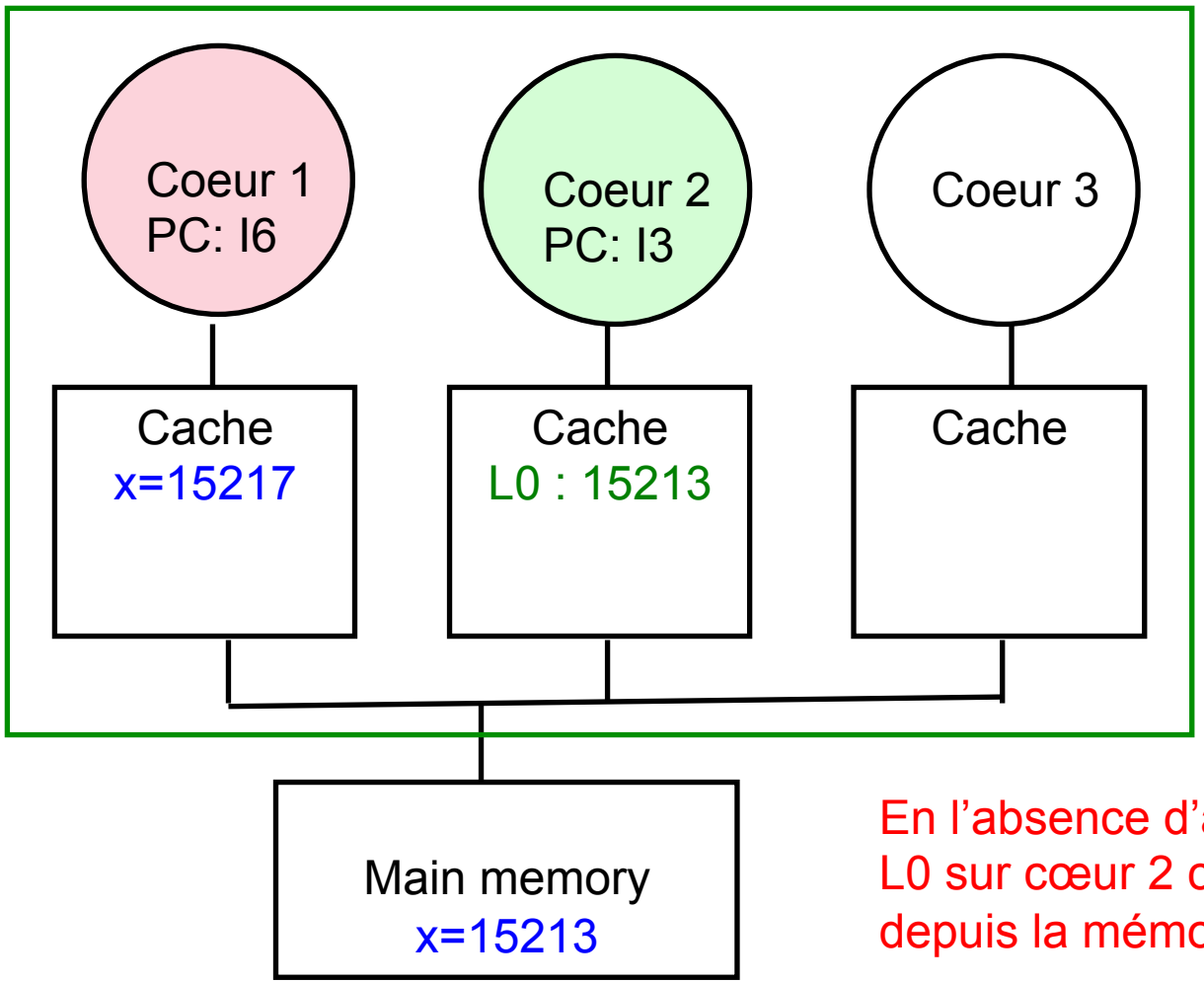




Cœur 1 a exécuté I5, lock est à 0
Cœur 2 prend le verrou et exécute I2

Pb : quelle sera la valeur chargée ?

Problème de la cohérence du cache (1) : Avoir un data race alors qu'on utilise des verrous




```
Programme
I0 : tsl r1, lock
i1 : bnz I0
I2 : ld 4000, r1
I3 : add, r1, 4
I4 : st r1, 4000
I5 : st r0 ,lock ;
I6 : jmp I0
```

En l'absence d'action spécifique
L0 sur cœur 2 charge la valeur
depuis la mémoire

Problème de la cohérence de cache
Est ce mieux en write through

- **Hypothèse : 2 cœurs C1, C2 avec chacun un cache permettant de mettre en cache X**
- **Identification du problème :**
  - 2 cœurs avec une donnée identique au contenu mémoire
  - Ecriture de X sur cœur C1 => X dans le cache de C1 avec pour valeur V et et idem en mémoire
  - Lecture, puis écriture de X sur cœur C2 => X dans le cache de C2 avec pour valeur V' et la mémoire contient x=V'
  - Le contenu du cache de C1 est **périmé** !!! (vaut V≠contenu mémoire)

Cependant si C1 accède à X, il y aura cache hit et utilisation de l'ancienne valeur

Problème et modèle de cohérence

- **Définition : problème de cohérence de cache**

Violation de la sémantique d'une activité parallèle causée par une combinaison de valeurs différentes entre un emplacement mémoire et ses différentes copies dans les caches.

- **Définition : modèle de cohérence (consistency model)**

Contraintes sur l'ordre dans lequel les écritures en mémoire sont réalisées.

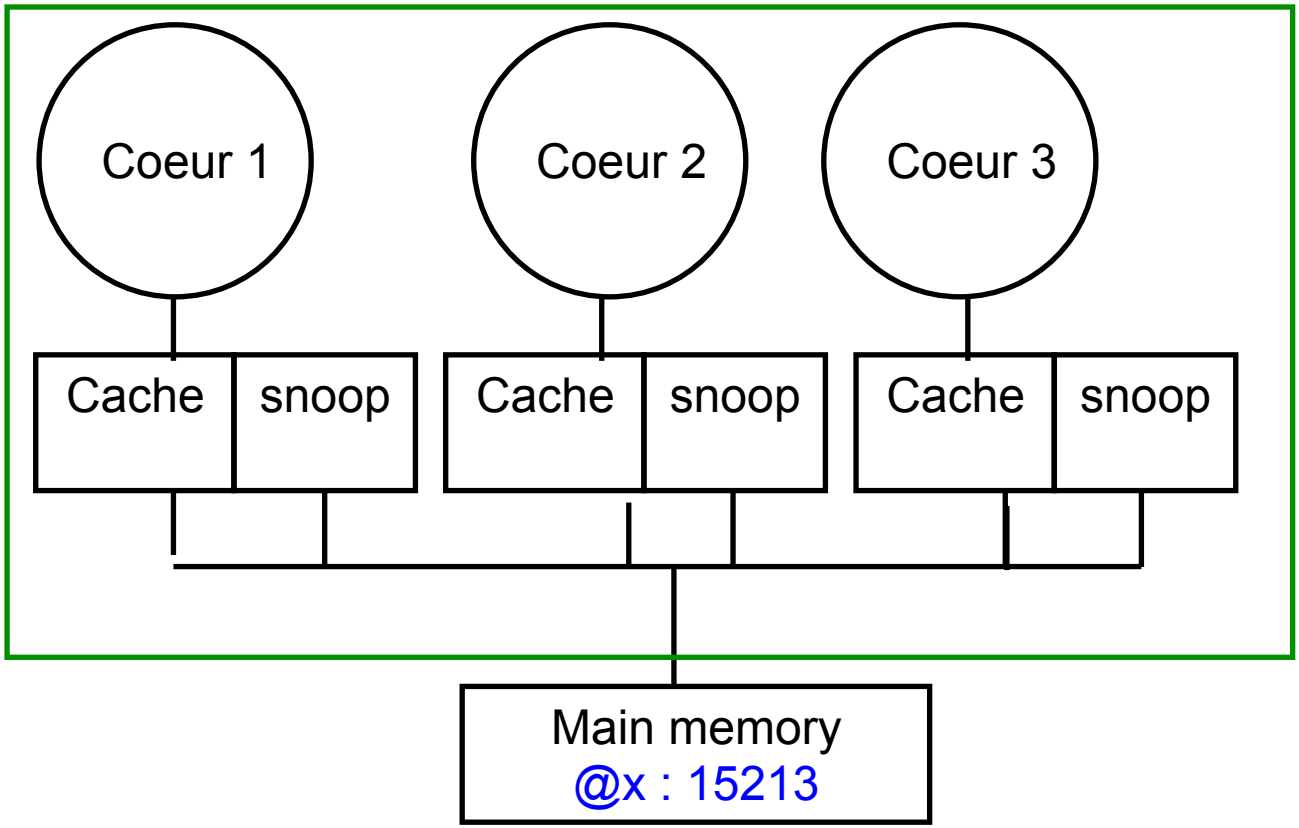
Cas des caches : contraintes sur les transferts entre caches et mémoire

- **Modèle de référence : cohérence séquentielle**

Les écritures d'une même séquence d'instructions sont réalisées dans l'ordre. Les écritures de deux séquences d'instructions exécutée sur deux cœurs différents sont entrelacées.

Mise en œuvre d'un modèle de cohérence
Le protocole de cohérence

- **Solution très populaire : scrutation (snooping)**



Informations circulant sur B :
- Adresse + type d'accès

Objectif déterminer les blocs partageant leur contenu (i.e. tag + index)

Solution :
- observer les accès en lecture pour détecter le Partage
- Mettre à jour son état pour éviter les problèmes de cohérence

Protocole de scrutation : état des blocs

- **Il peut y avoir plusieurs caches contenant un même bloc si sa valeur est identique à la mémoire.**
- **Si une opération modifie un bloc dans un cache,**

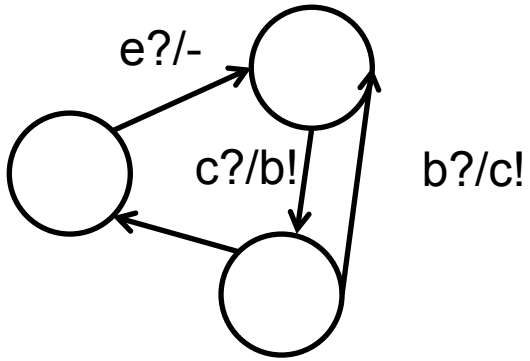
Choix 1 ce dernier doit informer les autres que leur « copie » est dépassée (invalidation)

Choix 2 ce dernier diffuse la mise à jour du bloc (update)

- **Comportement détaillé à définir pour chaque événement (Read/Write x Hit/Miss)**
- **Pb : le protocole occupe le bus commun**
  - Invalidation => read/write miss
  - Mise à jour => transfère du bloc modifié

Syntaxe des automate communicants

- **Pré-requis : notation graphique d'un automate fini**
- **Idée : identifier ce qui est observé émis, arcs = obs/action**
  - Si e est émis/exécuté : e!
  - Si e est reçu / observé : e?



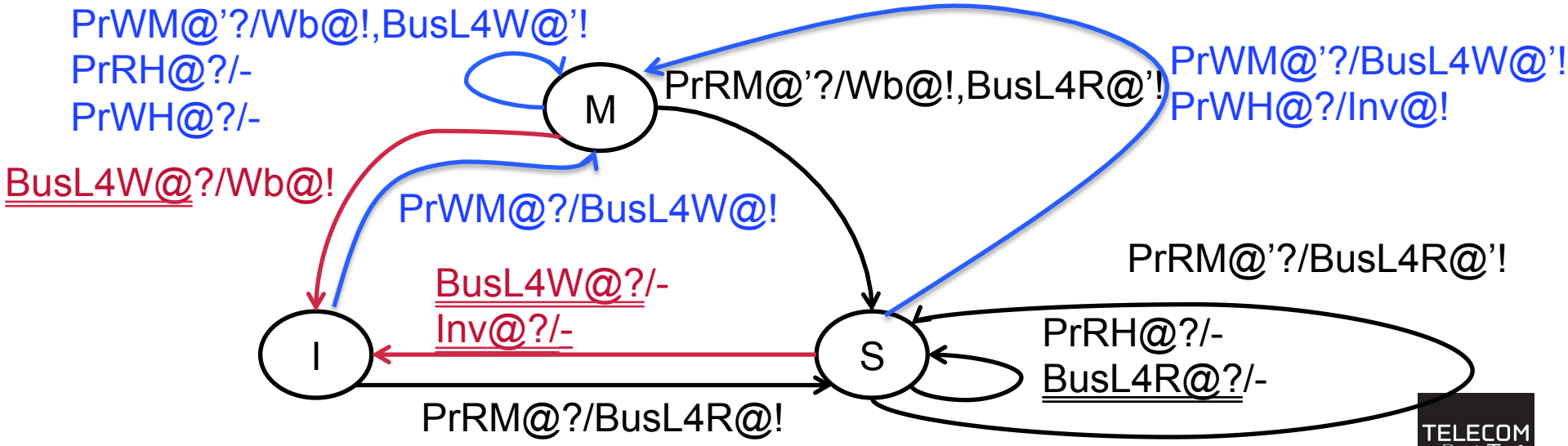
- **Note : modèle du cache = réactif => toujours 1 observation**

Événements observés / exécutés

- **Les messages sur le bus entre processeurs et mémoire**
  - Requête de lecture suite à un read miss: BusL4R (load for Read)
  - Requête de lecture suite à un write miss: BusL4W (load for Write)
  - Signal d'invalidation : Inv
  - Diffusion du bloc mis à jour (Write back) : Wb
- **Les événements observés par le cache lors d'une demande d'accès du processeur (read/write x hit /miss)**
  - Read hit : PrRH
  - Read miss : PrRM
  - Write hit : PrWH
  - Write Miss : PrWM
- **Remarque : un cache émet sur ou observe le bus pas les deux ! Il peut emettre 2 messages (e.g. Wb puis BusL4X)**

Protocole d'invalidation de ligne (MSI) pour cache write-back

- **Idée : autoriser 1 seul écrivain ou (exclusif) autant de lecteurs que nécessaire**
- **Etat d'un bloc : étiquette + contenu + état de partage**
  - Modified : contenu bloc ≠ mémoire suite à une écriture (écrivain)
  - Shared : contenu bloc = mémoire (lecteurs)
  - Invalid : le contenu de ce bloc ne doit pas être utilisé
- **Hyp : étq ~@, @'≠@ adresse quelconque**



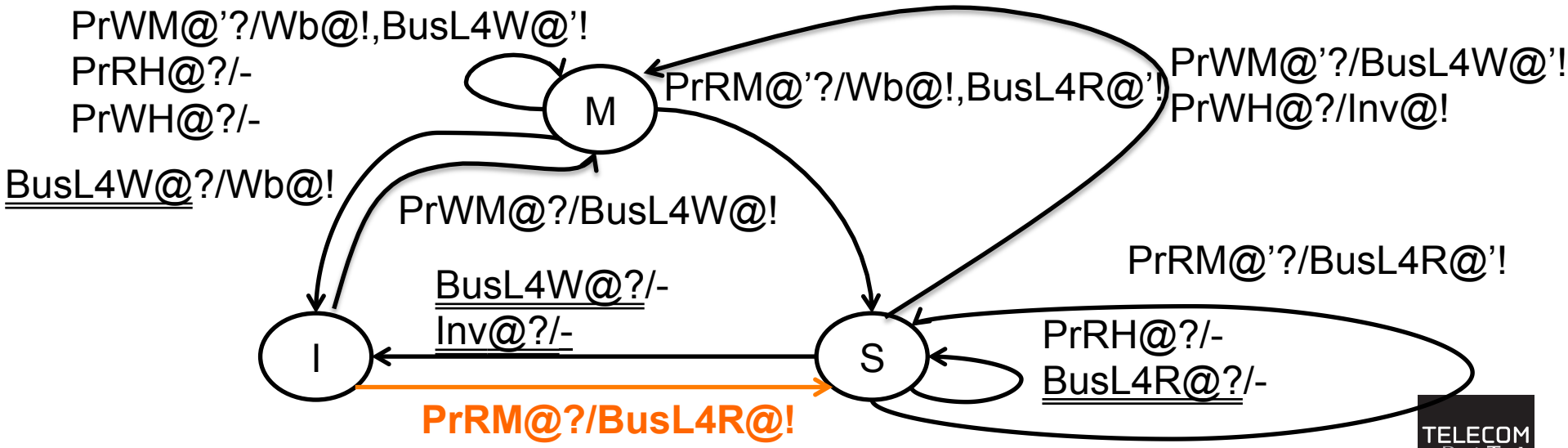
Protocole d'invalidation de ligne (MSI) pour cache write-back

Taille bloc : 4 octets, adressage par mots de 2 octets

	Cache1	État	Cache2	état
B0	Etq=4096, Val=FA010002	S	Etq=0, Val=00000000	I

@	Val
…	
4096	FA01
4097	0002
4098	B001
…	
8192	0001

C1 charge 4096 dans r1



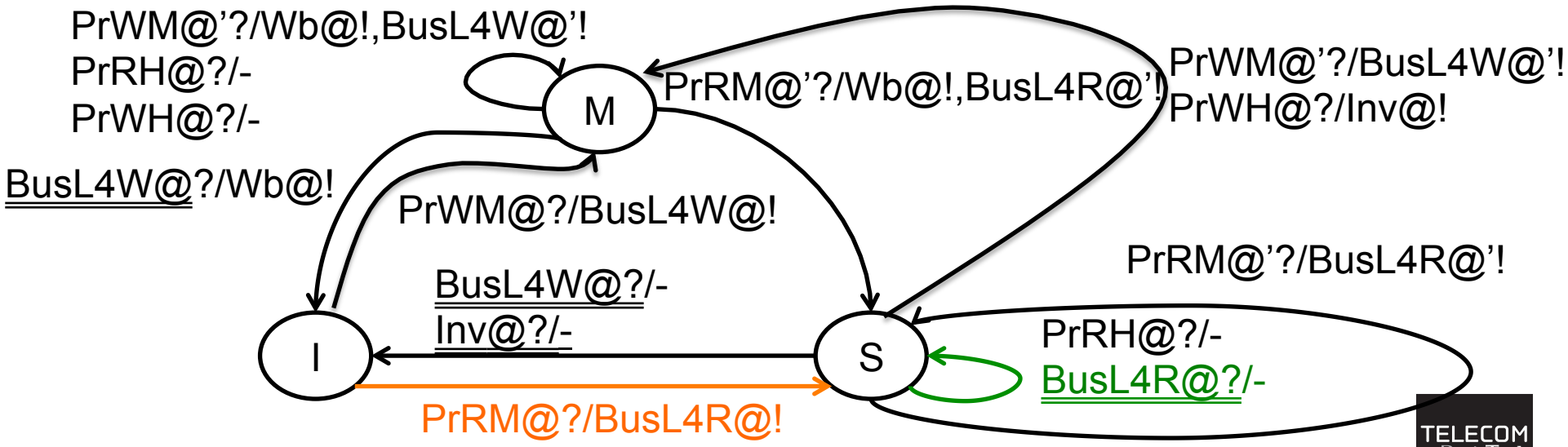
Protocole d'invalidation de ligne (MSI) pour cache write-back

Taille bloc : 4 octets, adressage par mots de 2 octets

	Cache1	État	Cache2	état
B0	Etq=4096, Val=FA010002	S	Etq=4096, Val=FA010002	S

@	Val
…	
4096	FA01
4097	0002
4098	B001
…	
8192	0001

C2 charge 4096 dans r1



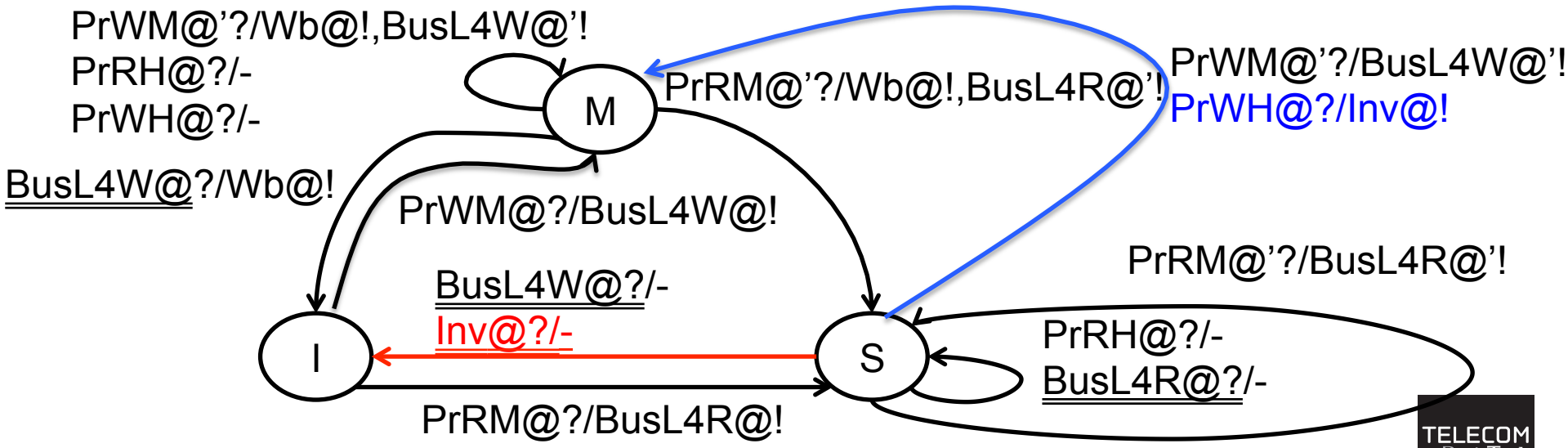
Protocole d'invalidation de ligne (MSI) pour cache write-back

Taille bloc : 4 octets, adressage par mots de 2 octets

	Cache1	État	Cache2	état
B0	Etq=4096, Val=FFFF0002	**M**	Etq=4096, Val=FA010002	S

@	Val
…	
4096	FA01
4097	0002
4098	B001
…	
8192	0001

C1 écrit FFFF à l'adresse 4096



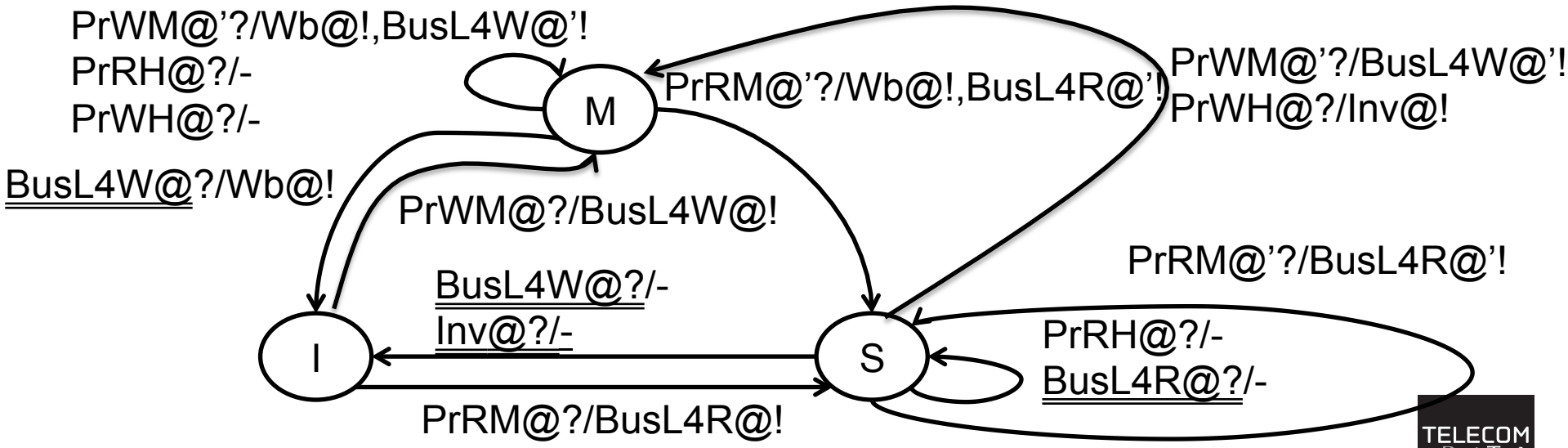
Protocole d'invalidation de ligne (MSI)
Faux partage

Taille bloc : 4 octets, adressage par mots de 2 octets

Et si C1 avait écrit FFFF à l'adresse 4097

	Cache1	État	Cache2	état
B0	?	?	?	?

@	Val
…	
4096	FA01
4097	0002
4098	B001
…	
8192	0001



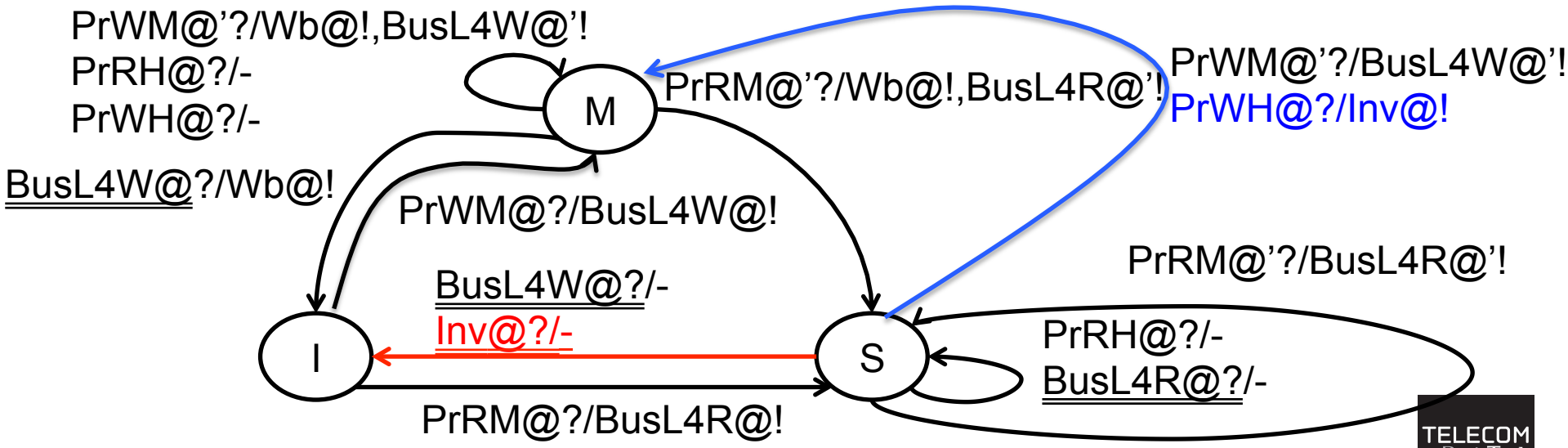
Protocole d'invalidation de ligne (MSI)
Faux partage

Taille bloc : 4 octets, adressage par mots de 2 octets

Et si C1 avait écrit FFFF à l'adresse 4097

	Cache1	État	Cache2	état
B0	Etq=4096, Val=FA01FFFF	**M**	Etq=4096, Val=FA010002	S

@	Val
…	
4096	FA01
4097	0002
4098	B001
…	
8192	0001



Observation et Conclusion

- **Parallélisme + hiérarchie mémoire = pb difficile**
- **Objectif n° 1 faire des calculs corrects**
- **Notion abordée en TD : conséquence faux partage**
- **Protocole de cohérence => usage du bus**
  - Pour signaler les changements d'état
  - Pour transférer les mises à jour des blocs
- **Pour plus de détails & autres protocoles Chap 9 : Computer Organization and Design: The Hardware/Software Interface, by D. Patterson, J. Hennessy**

http://www.cs.nccu.edu.tw/~whliao/cod2005/09HP.pdf

Approche alternative :
déplacement explicite des données !