

Gerarchia di Memoria e Cache

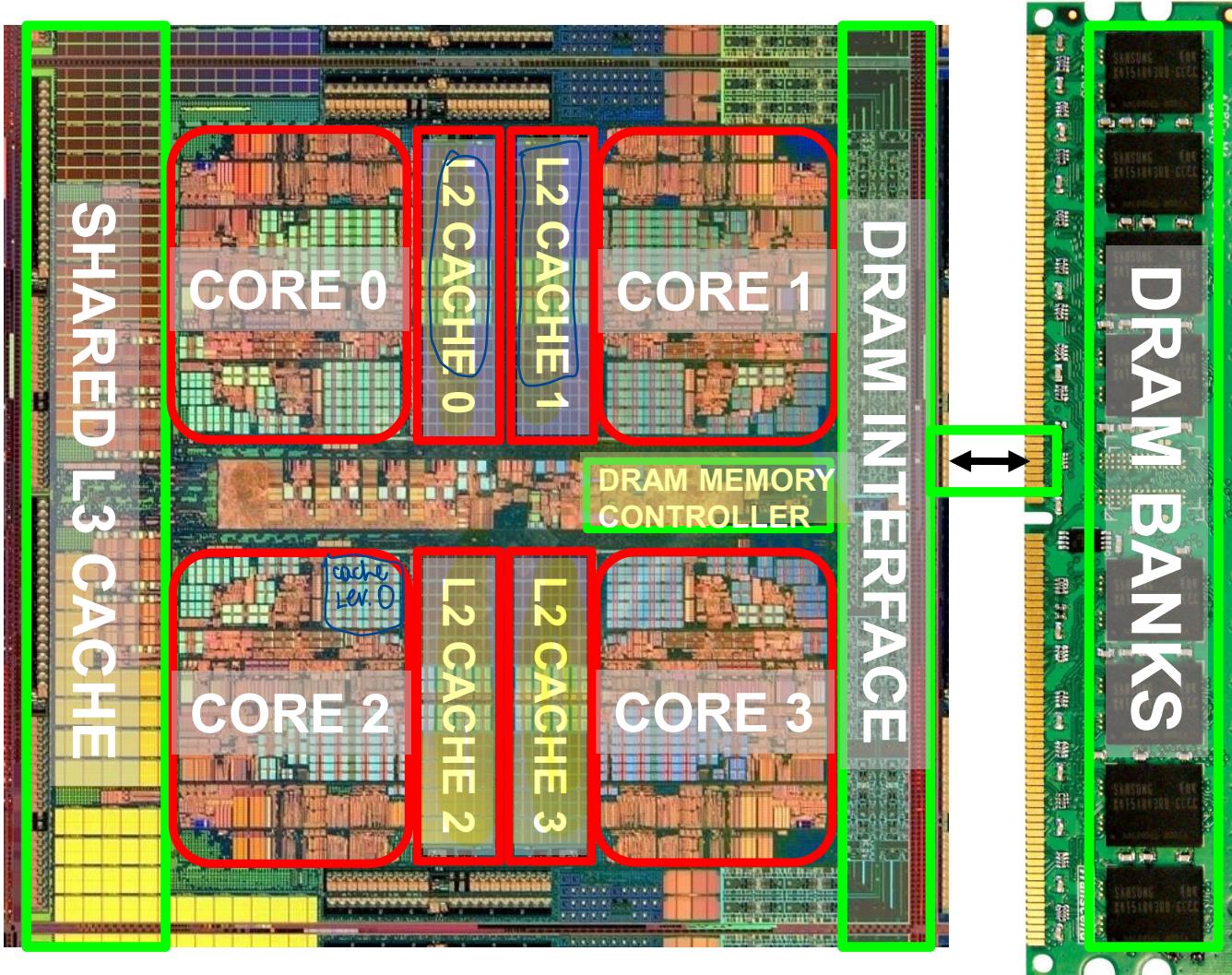
Andrea Bartolini – a.bartolini@unibo.it

Multi-Core Issues in Caching

- Miss:
- compulsorie
 - di conflitto

Caches in a Multi-Core System

CACHE: X EVITARE DA
ACCEDERE IN MEMORIA

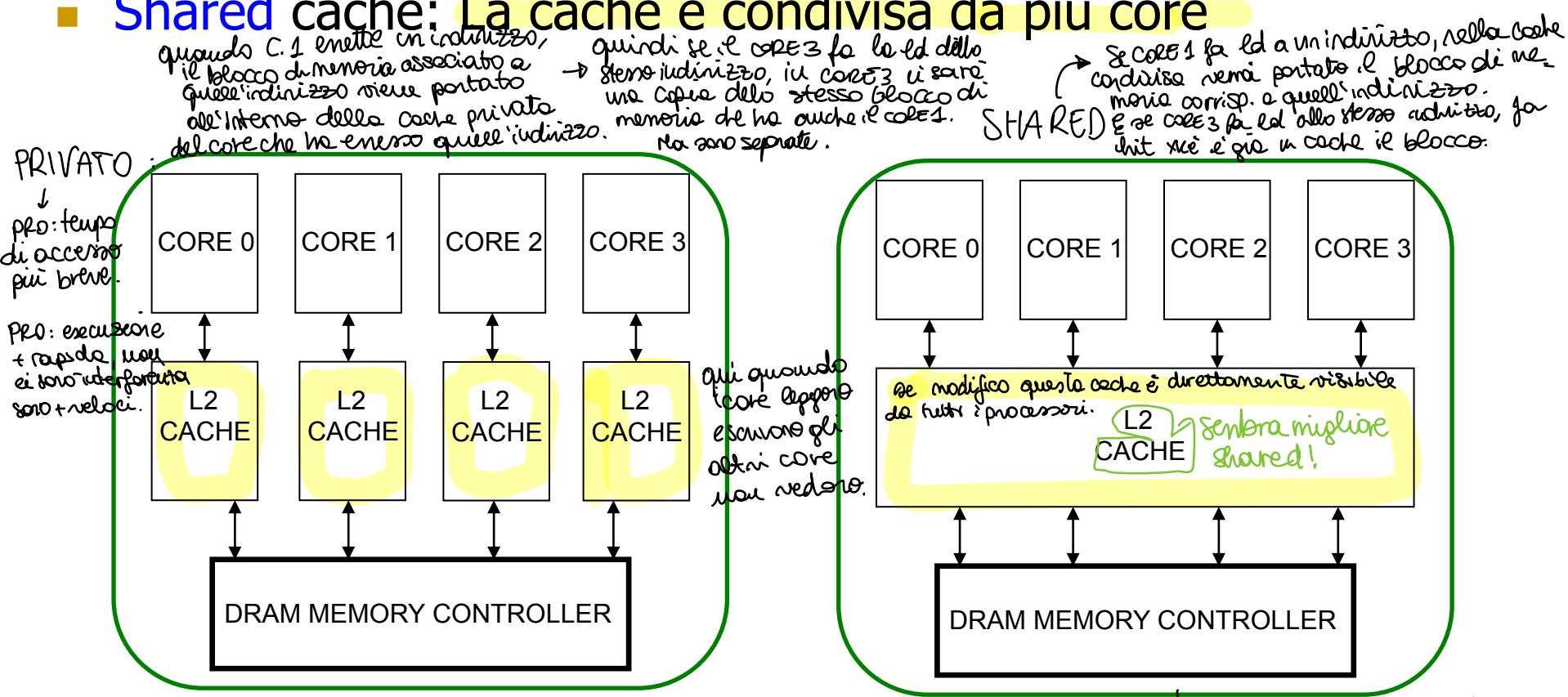


Caches in Multi-Core Systems

- L'efficienza della cache diventa ancora più importante in un sistema multi-core/multithread
 - Memory bandwidth è prioritaria
 - Lo spazio di cache è una risorsa limitata tra core/thread
- Come progettiamo le cache in un sistema multi-core?

Private vs. Shared Caches

- **Private cache:** la cache appartiene ad un core (un blocco condiviso può essere in più cache)
- **Shared cache:** La cache è condivisa da più core



Shared → contro: ha + copie dello stesso dato, ma ottimizza lo spazio
pro: può essere fisicamente suddivisa in più pezzi.

Tipicamente i primi livelli di cache sono privati, quelli più profondi sono shared.

Shared Caches Between Cores

■ Advantages:

- High effective capacity $\text{capacity}_{\text{effective}} >$
- Dynamic partitioning of available cache space
 - No fragmentation due to static partitioning
 - If one core does not utilize some space, another core can
- Easier to maintain coherence (a cache block is in a single location)

■ Disadvantages

- Slower access (cache not tightly coupled with the core)
- Cores incur conflict misses due to other cores' accesses
 - Misses due to inter-core interference
 - Some cores can destroy the hit rate of other cores
- Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)

Caching in Multiprocessors

- Caching not only complicates ordering of all operations...
 - A memory location can be present in multiple caches
 - Prevents the effect of a store or load to be seen by other processors → makes it difficult for all processors to see the same global order of (all) *memory operations*
- ... but it also complicates ordering of operations on a single memory location
 - A single memory location can be present in multiple caches
 - Makes it difficult for processors that have cached the same location to have the correct value of that location (in the presence of updates to that location)

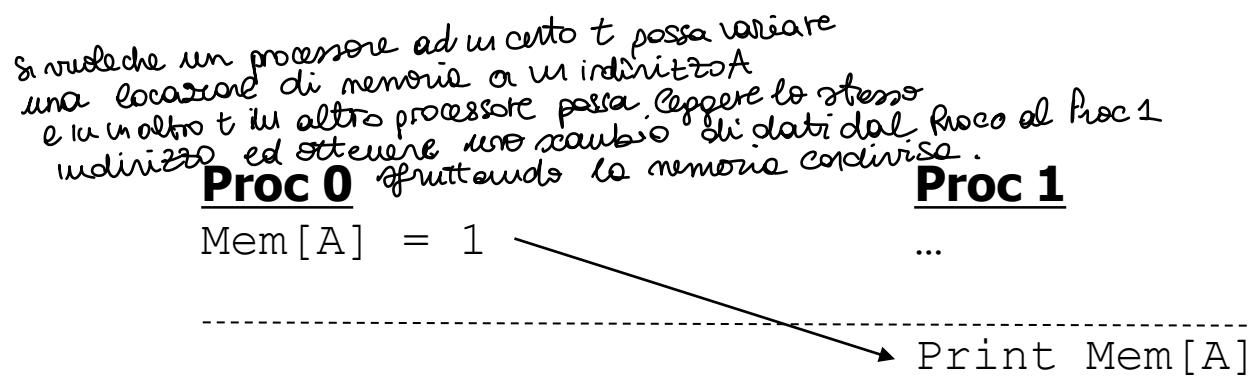
Memory Consistency vs. Cache Coherence

- **Consistency** is about ordering of **all memory operations** from different processors (i.e., to different memory locations)
 - **Global ordering** of accesses to *all* memory *locations*
- **Coherence** is about ordering of **operations** from different processors **to the same memory location**
 - **Local ordering** of accesses to *each* cache *block*

Cache Coherence

Shared Memory Model

- Many parallel programs communicate through *shared memory*
- Proc 0 writes to an address, followed by Proc 1 reading
 - This implies communication between the two



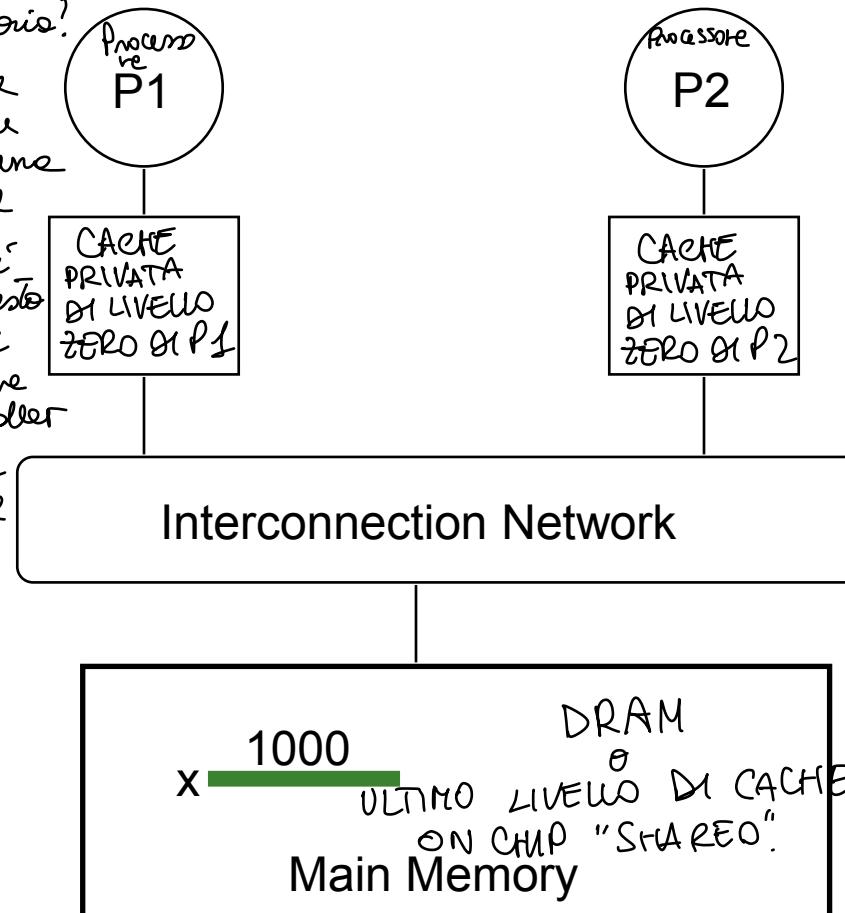
- Each read should receive the value last written by anyone
 - This requires synchronization (what does last written mean?)
- What if Mem[A] is cached (at either end)?

Cache Coherence

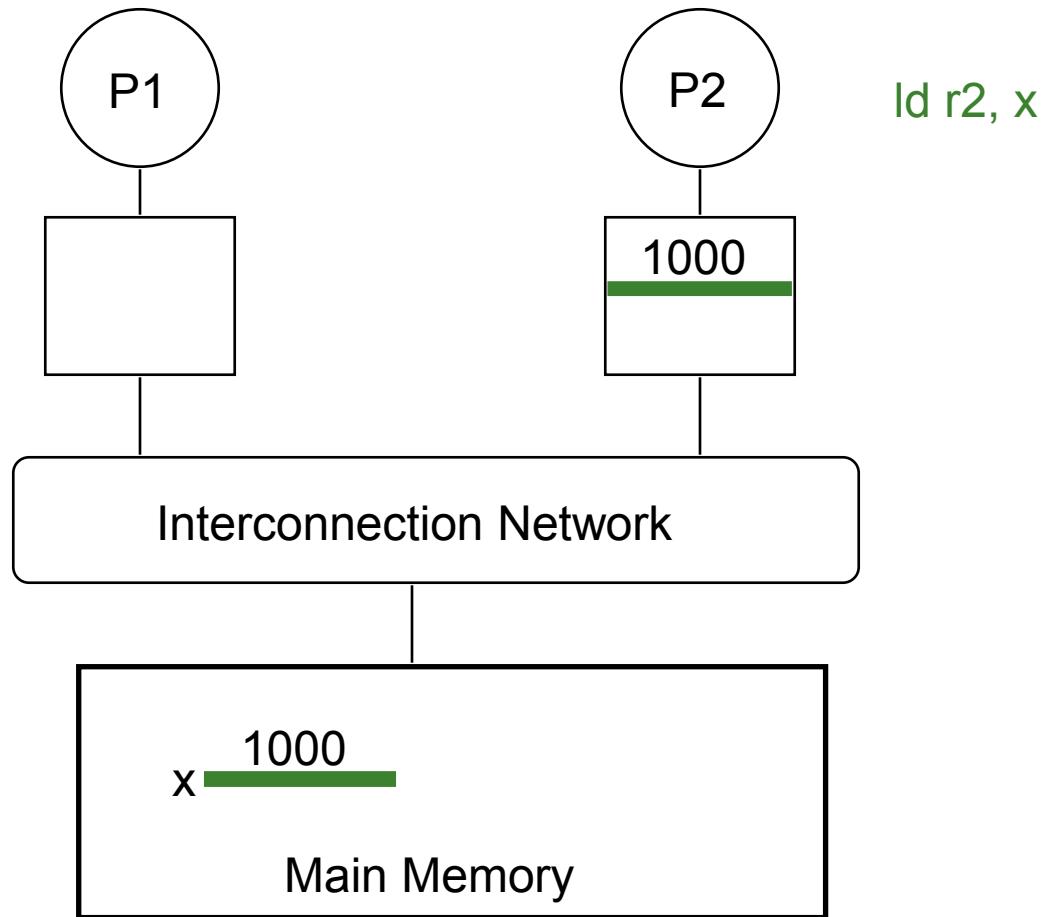
- Se più processori memorizzano nella cache lo stesso blocco, come fanno a garantire che tutti vedano uno stato coerente?

Se P1 e P2 fanno accesso allo stesso blocco di memoria, come si può garantire che tutti i processori vedono uno stato coerente della memoria?

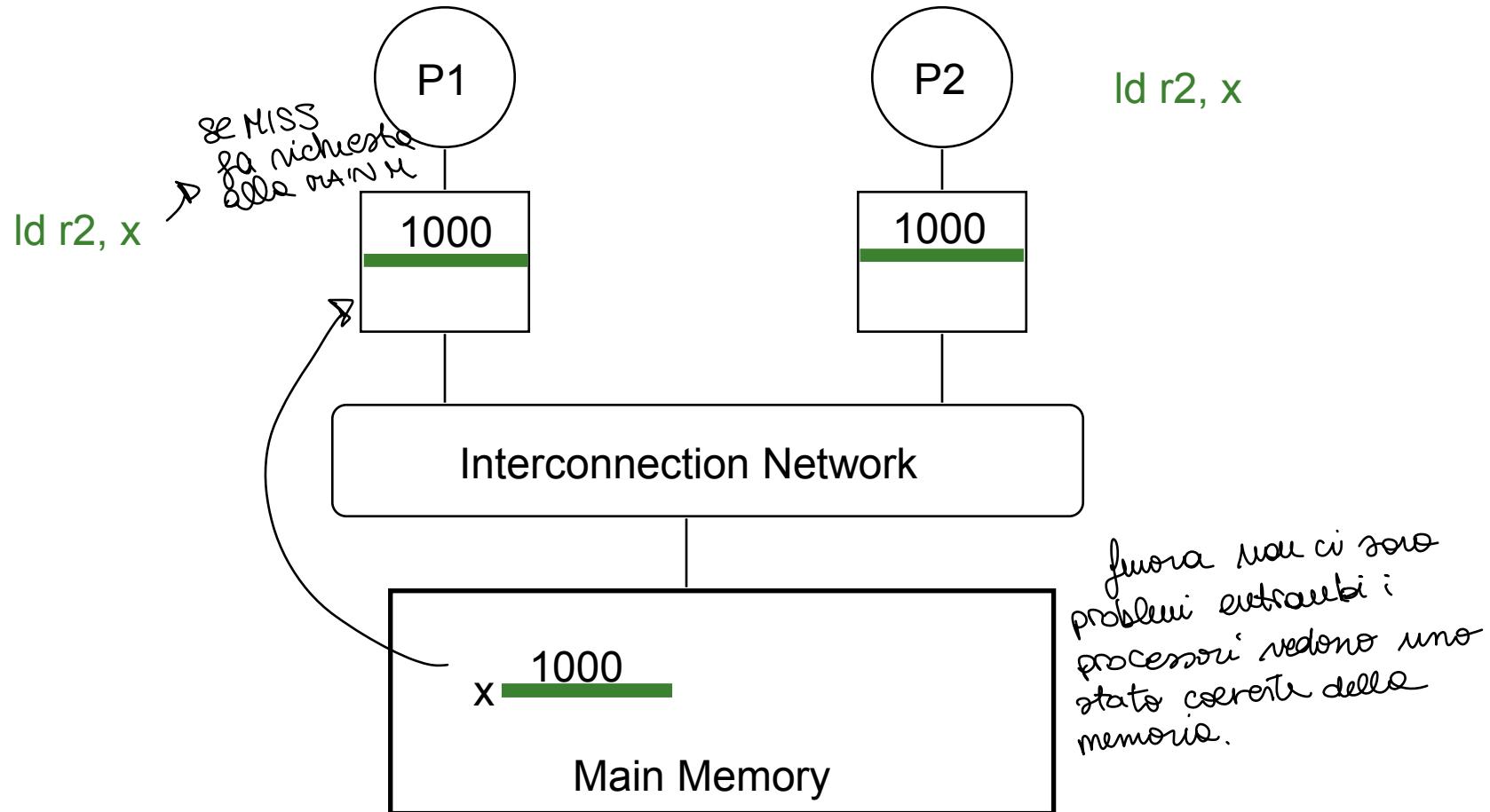
Se x e y P2 legge una variabile all'indirizzo x e la salva in un registro, quando si farà una lettura di y, il processore 2 enerterà sul bus degli indirizzi la lettura di un certo indirizzo. Lo richiesto da un altro al cache controller che verificherà se il dato è in cache e se c'è una miss il cache controller andrà alla main memory e consenirà nella cache privata il blocco di memoria.



The Cache Coherence Problem

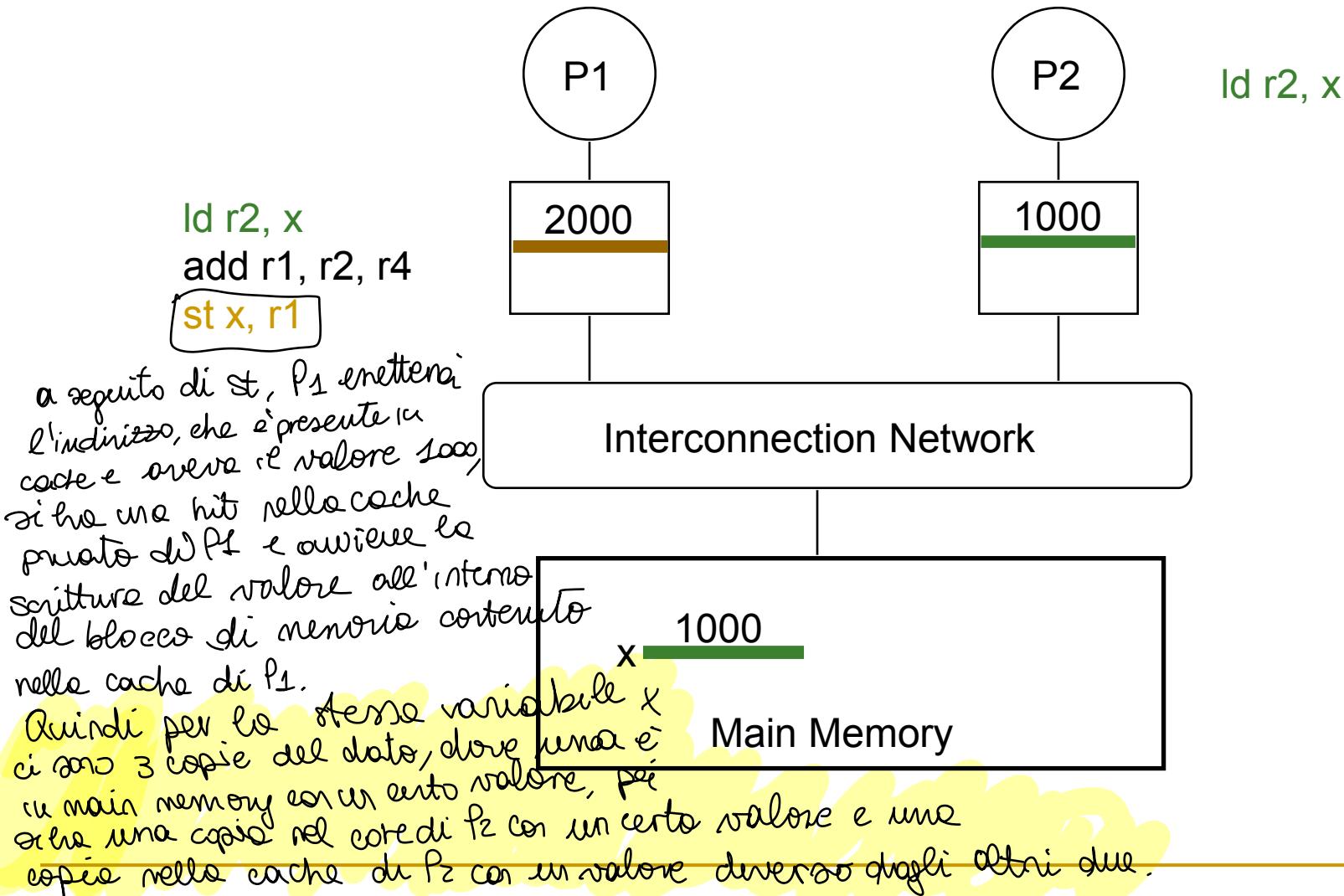


The Cache Coherence Problem



The Cache Coherence Problem

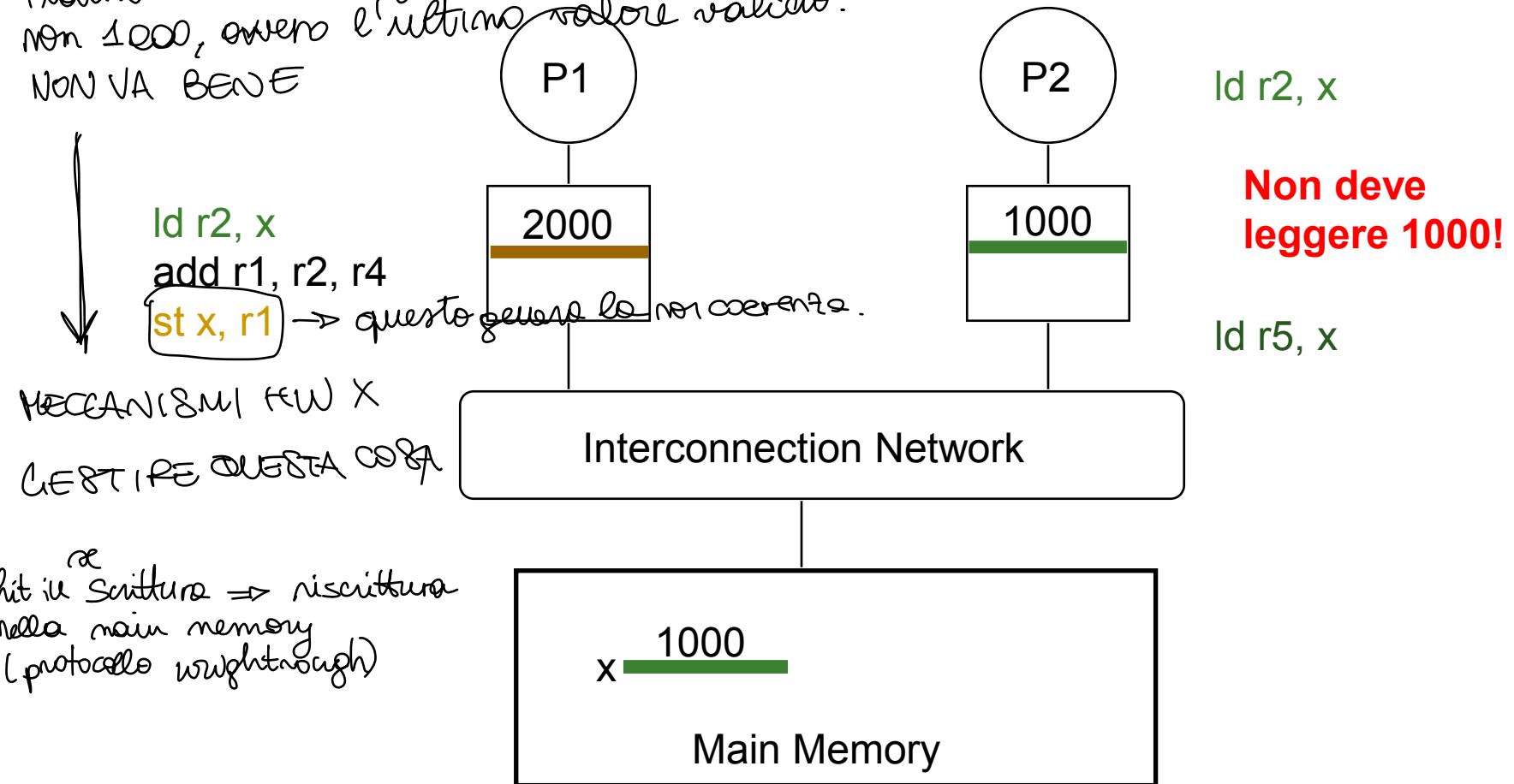
Il problema c'è quando uno dei due processori decide di modificare il valore della variabile x , e quindi fa una store, verso la variabile x , di un nuovo valore.



The Cache Coherence Problem

Il problema è se P2 vuole leggere di nuovo il valore associato a quell'indirizzo. Se P2 fa ld verso x, assumiamo che venga visto il valore 2000, non 1000, ovvero l'ultimo valore valido.

NON VA BENE



Cache Coherence: Whose Responsibility?

- Software
 - Can programmer ensure coherence if caches invisible to software?
 - **Coarse-grained:** Page-level coherence has overheads
 - Non-solution: Make shared locks/data non-cacheable
 - **A combination of non-cacheable and coarse-grained is doable**
 - **Fine-grained:** What if the ISA provided a cache flush instruction?
 - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache.
 - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches.
 - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.
- Hardware
 - **Greatly simplifies software's job**
 - One idea: Invalidate all other copies of block A when a core writes to A

(Non-)Solutions to Cache Coherence

- **No hardware based coherence** *fare cache shared*
 - Keeping caches coherent is software's responsibility
 - + Makes microarchitect's life easier
 - Makes average programmer's life much harder
 - need to worry about hardware caches to maintain program correctness?
 - Overhead in ensuring coherence in software (e.g., page protection, page-based software coherence, non-cacheable)
- **All caches are shared between all processors**
 - + No need for coherence
 - Shared cache becomes the bandwidth bottleneck
 - Very hard to design a scalable system with low-latency cache access this way

Maintaining Coherence

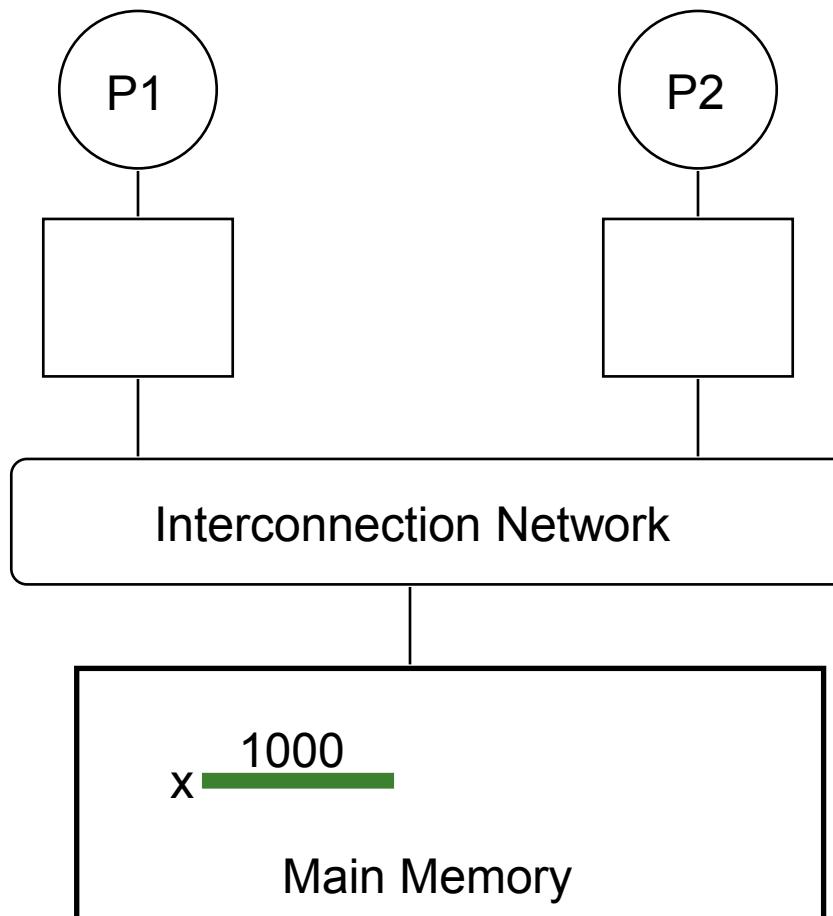
- Need to guarantee that all processors see a consistent value (i.e., consistent updates) for the same memory location *fare in modo che eventuali scritture ad una locazione di memoria fatte da un processore siano viste dagli altri processori.*
- Writes to location A by P0 should be seen by P1 (eventually), and all writes to A should appear in some order *Sequenzialità tra le varie scritture.*
Le multiple scritture di una certa locazione di memoria non possono avvenire contemporaneamente, c'è sempre una che avviene prima delle altre.
- Coherence needs to provide:
 - **Write propagation:** guarantee that updates will propagate
 - **Write serialization:** provide a consistent order seen by all processors for the same memory location
- Need a global point of serialization for this store ordering

scrittura: c'è bisogno di una sincronizzazione quando scrivo a un det. indirizzo.

lettura: possono avvenire in modo non sincronizzato.

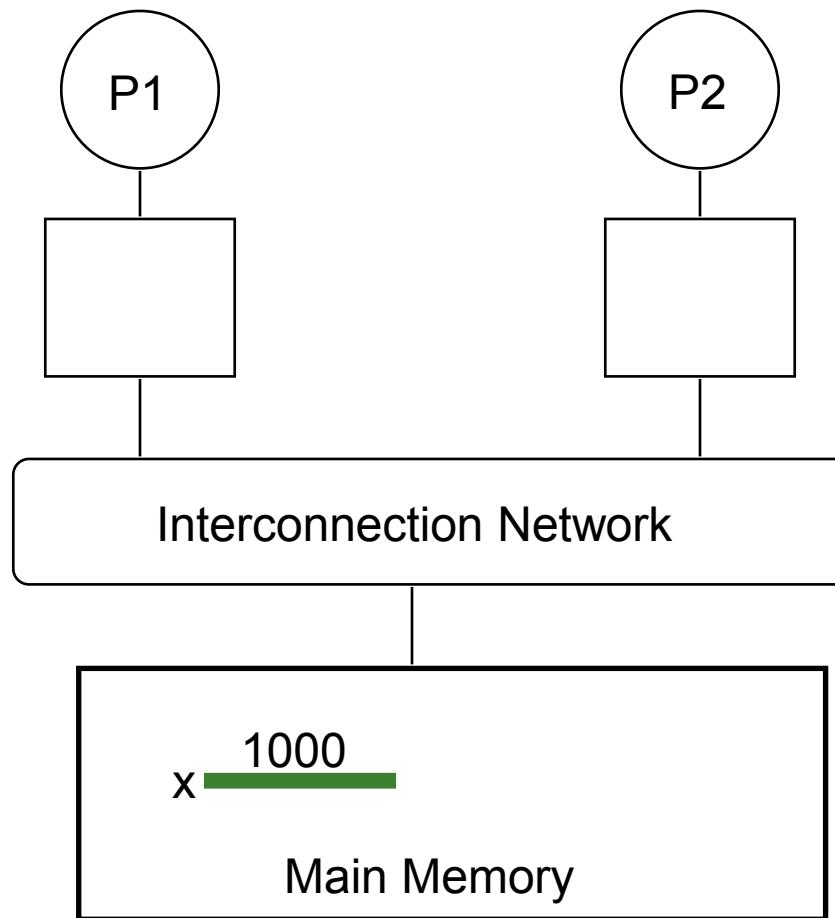
Cache Coherence

- Se più processori memorizzano nella cache lo stesso blocco, come fanno a garantire che tutti vedano uno stato coerente?



Cache Coherence

- Se più processori memorizzano nella cache lo stesso blocco, come fanno a garantire che tutti vedano uno stato coerente?



Hardware Cache Coherence

- Basic idea:

- A processor/cache broadcasts its write/update to a memory location to all other processors
 - Another cache that has the location either updates or invalidates its local copy

Quando un processore fa store faccio broadcast agli altri.

Gli altre cache a seguito di questa richiesta possono fare update della loro copia locale oppure invalidarla.

Necessariamente devono fare una delle due cose.

Coherence: Update vs. Invalidate

- How can we *safely update replicated data?*
 - Option 1 (Update protocol): ^{insieme alle comunicazioni di modifica} push an update to all copies ^{sarà fornito anche}
 - Option 2 (Invalidate protocol): ensure there is only one ^{detto} copy (local), update it ^{nuovo.}

- **On a Read:** \rightarrow se la cache locale non contiene il tag associato a quell'indirizzo dovrà estrarre una richiesta di lettura del dato.
 - If local copy is Invalid, put out request ^{lettura del dato.}
 - (If another node has a copy, it returns it, otherwise memory does) tutti gli altri invalidano le loro espie.

Coherence: Update vs. Invalidate (II)

■ On a Write:

- ❑ Read block into cache as before

Update Protocol: quando Scivo comunico a tutte le cache che ho un
quel dato

- ❑ Write to block, and simultaneously broadcast written data and address to sharers
- ❑ (Other nodes update the data in their caches if block is present)

Invalidate Protocol:

- ❑ Write to block, and simultaneously broadcast invalidation of address to sharers
- ❑ (Other nodes invalidate block in their caches if block is present)

Update vs. Invalidate Tradeoffs

- Which do we want?
 - Write frequency and sharing behavior are critical
- **Update** quando si modifica un dato si comunica, utilizzo di molte bande. Comunico indirizzo + dato da modificare
 - + If sharer set is constant and updates are infrequent, avoids the cost of invalidate-reacquire (broadcast update pattern)
 - If data is rewritten without intervening reads by other cores, updates would be useless
 - Write-through cache policy → bus becomes bottleneck
- **Invalidate** comunica solo indirizzo e non dato .
 - + After invalidation broadcast, core has exclusive access rights
 - + Only cores that keep reading after each write retain a copy
 - If write contention is high, leads to ping-ponging (rapid invalidation-reacquire traffic from different processors)

Two Cache Coherence Methods

- ❑ How do we ensure that the proper caches are updated?
 - Insieme di gli concorrenti dai cache controller di un determinato bus. Questo bus garantisce serializzazione.
 - ❑ **Snoopy Bus** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
 - core fino a 4-8 core sono troppo lento.
 - Bus-based, **single point of serialization for all memory requests**
 - Processors observe other processors' actions
 - ❑ E.g.: P1 makes “read-exclusive” request for A on bus, P0 sees this and invalidates its own copy of A
 - ⇒ BUS CONDIVISO IN CUI TUTTI I CACHE CONTROLLER E LA MAIN MEMORY SONO IN ASCIUTO. SUL BUS VENGONO COMUNICATI GLI INDIRIZZI.
 - ❑ **Directory** [Censier and Feautrier, IEEE ToC 1978]
 - **Single point of serialization per block**, distributed among nodes
 - Processors make explicit requests for blocks
 - Directory tracks which caches have each block
 - Directory coordinates invalidation and updates
 - ❑ E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1
- hp: spazio di ind a 32 bit
e blocchi di memoria a 32 byte, avremo allora 16 interne del directory una struttura dati con $2^{32}/2^5$ celle che contengono 1 bit per core che dicono se quelle cache hanno una copia esclusiva del dato + un extra bit che si dice se la copia è esclusiva o no.

Snoopy Cache Coherence

- Idea:
 - All caches “snoop” all other caches’ read/write requests and keep the cache block coherent
 - Each cache block has “coherence metadata” associated with it in the tag store of each cache
- Easy to implement if all caches share a common bus
 - Each cache broadcasts its read/write operations on the bus
 - Good for small-scale multiprocessors
 - What if you would like to have a 10,000-node multiprocessor?

Directory Based Coherence

- Idea: A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.
Le cache prima di effettuare una lettura o scrittura consultano il directory e vedono se c'è qualcuno che ha modificato il dato.
- An example mechanism:
In questo caso il directory comunicherà a quel qualcuno di invalidarlo e salvarlo nelle memorie
 - For each cache block in memory, store $P+1$ bits in directory
 - One bit for each cache, indicating whether the block is in cache
 - Exclusive bit: indicates that a cache has the only copy of the block and can update it without notifying others
che ha fatto richieste di accesso alle te datti corretto
 - On a read: set the cache's bit and arrange the supply of data
 - On a write: invalidate all caches that have the block and reset their bits
 - Have an “exclusive bit” associated with each block in each cache (so that the cache can update the exclusive block silently)

Write allocate : Miss in caso di scrittura si verifica se il blocco di memoria associato a quell' indirizzo è presente nella cache di livello zero, altrimenti si fa cache refill, ovvero si prende il blocco di memoria associato a quell'indirizzo e si porta nella cache.

Write around: Miss in caso di scrittura, si scrive direttamente la main memory senza portare il blocco di memoria associato a quel indirizzo all'interno della cache di livello più prossimo a noi.

Gestisco le Miss di scrittura non come le Write allocate, perché

1. Se scrivo un dato e non lo devo riusare, posso tollerare di avere bisogno di più ciclo per scriverlo nella main memory. se lo uso spesso ha senso metterlo in cache.
2. L'algoritmo di coerenza risulta più facile usando lo Write around, perché la main memory è corretta, la copia locale no, ma in una ottica di coordinazione è meglio che sia corretta la main memory. Se avessi gestito con Write allocate i casi di Miss in scrittura avrei avuto una copia locale corretta, ma diversa dalla main memory.

Se ho una hit in scrittura, la coerenza è più facile gestirla con writeback o writethrough ?

La seconda perché lo stato della main memory è coerente alla copia nella cache, quindi leggendo la main memory si trova sempre il dato corretto.

La combinazione migliore per la coerenza è writethrough e Write around, anche se non risolvono completamente, perché in caso di lettura si hanno copie locali e bisogna garantire che questa sia sempre coerente con la memoria.

Perché se leggo un dato e qualcuno dopo lo scrive, anche se usa Write tr, la memoria verrà variata ma questo non dice niente noi abbiamo un dato non coerente.

PROTOCOLLO VALID/INVALID

(1)

Se il core vuole modificare (scrivere) un dato, esiste una sola copia locale di quel dato nel nostro sistema. Questo garantisce la coerenza.

Se un core vuole modificare un dato, deve comunicare sul bus di sono questa intenzione e gli altri controllori di cache di livello zero, se hanno una copia locale di quel dato la devono invalidare.

Questo vuol dire che per realizzare questo protocollo bisogna aumentare i metadati associati al tag store di livello zero, con l'informazione se quella linea di cache è valida o invalida (valid bit?)

Inoltre questo processore gestisce le hit in scrittura tramite Write through , e le Miss in scrittura con Write around.

All'interno del bus di snoop vediamo viaggiare dei comandi di un bus read su un certo indirizzo e di bus Write di un certo indirizzo.

Le azioni emesse dal processore sono le operazioni di processor read e processor Write.

La macchina a Stati può avere come ingresso una professor Write e contestualmente deve emettere sul bus L informazione che sta facendo una bua Write di un certo indirizzo.

Se osserviamo una azione su un blocco di memoria che non è contenuto nella nostra cache, non dobbiamo fare niente, comunque prima va controllato se c'è l'ho in cache (hit) .

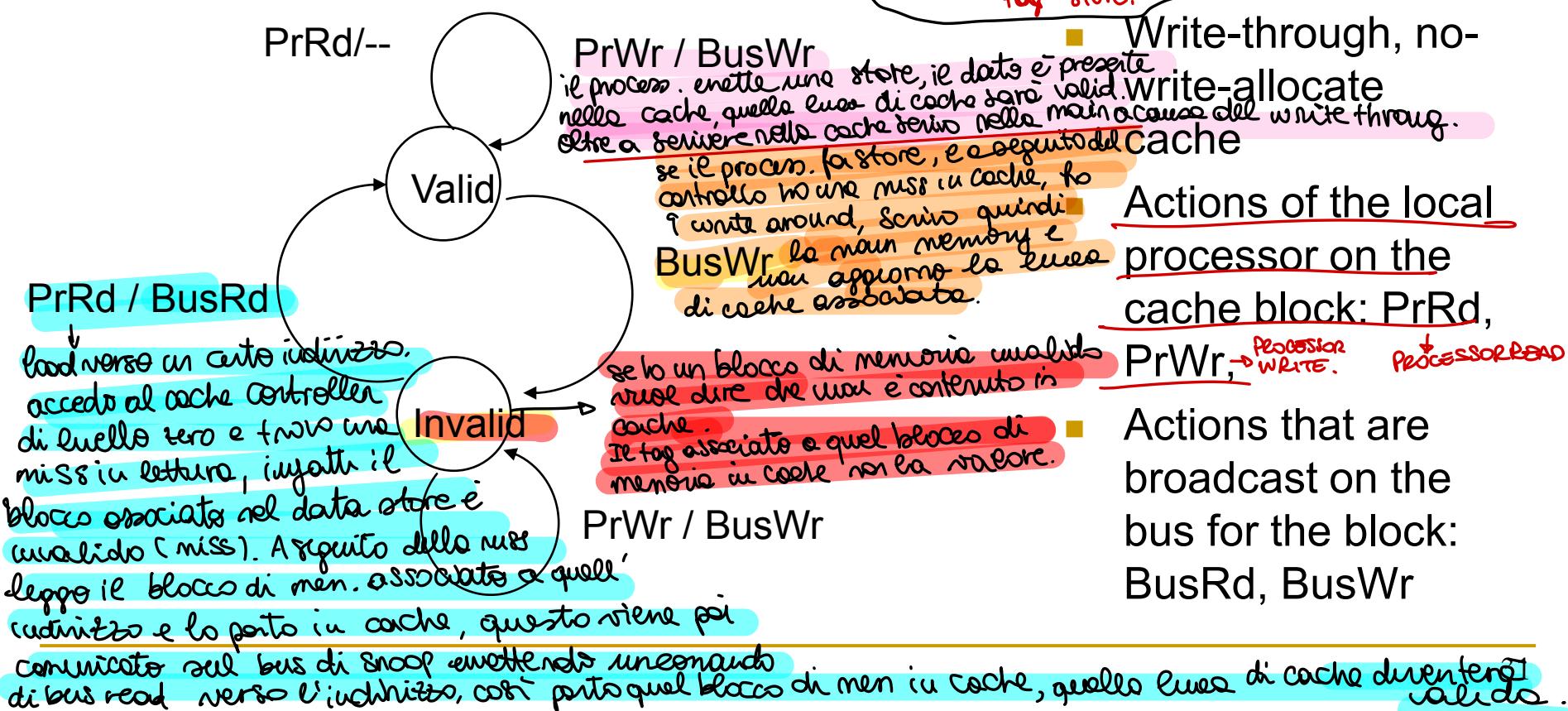
A Very Simple Coherence Scheme (VI)

2

- Caches “snoop” (observe) each other’s write/read operations. If a processor writes to a block, all others invalidate the block.
- A simple protocol:

Machinie è stato del controllore delle cache di livello zero di ciascun core.

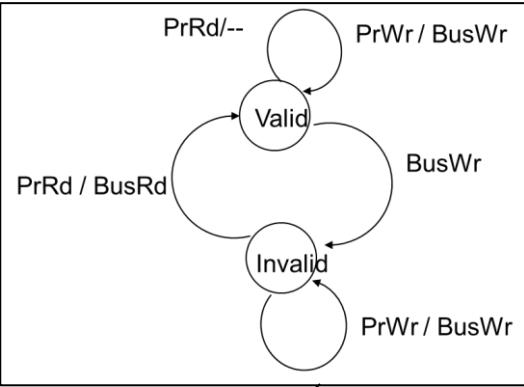
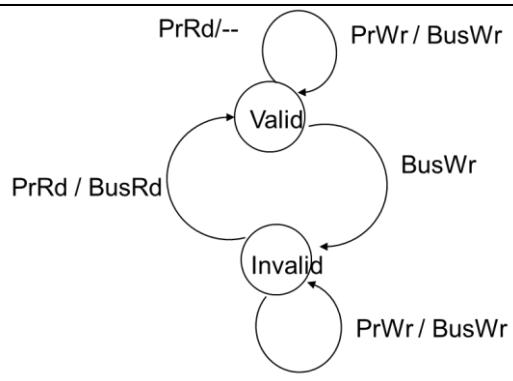
2 STATI per ciascun blocco di memoria nel data store e quindi x ciascun tag del tag store.



(3)

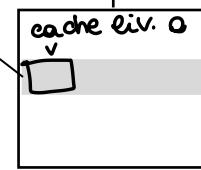
Ho un passaggio da valid a invalid quando osservo che qualcuno vuole scrivere un blocco di memoria di cui ho una copia locale, ovvero osservo sul bus di snoop che qualcuno ha emesso una richiesta di scrittura verso un indirizzo e a seguito di un controllo che faccio sulla presenza o meno di questo indirizzo nella mia cache ho una hit.
Quello che dovrò fare è invalidare quella linea di cache, così non esiste una copia locale non coerente con la main memory.

(VI) - Esempio

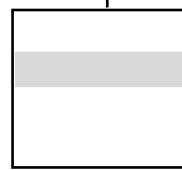


ciascuna linea di cache ha
 una macchina a stati con protocollo
 valid/invalid
Costo della coerenza:
 avere una macchina a stati e
 bit aggiuntivi in ogni linea di cache.

P1



P2

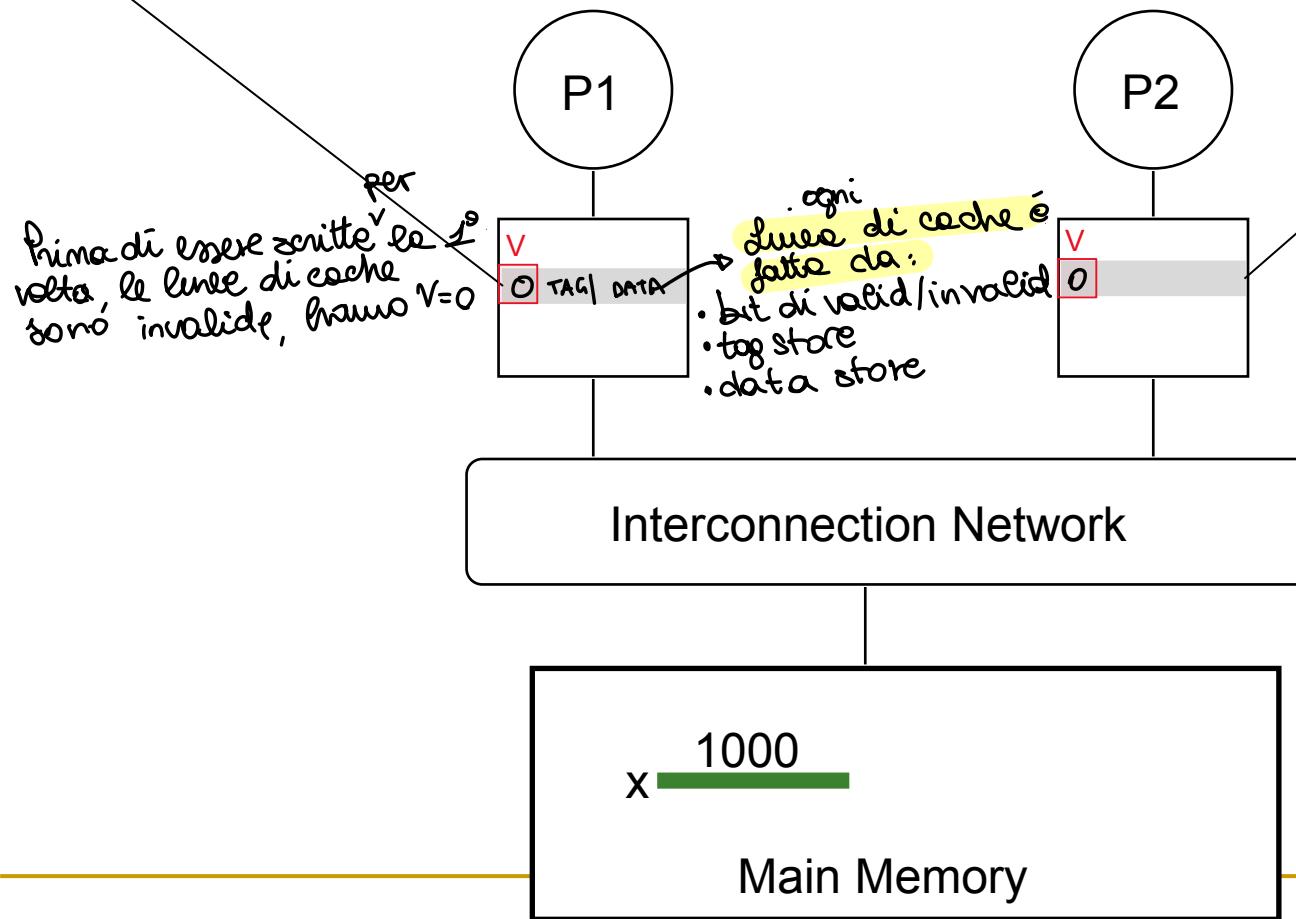
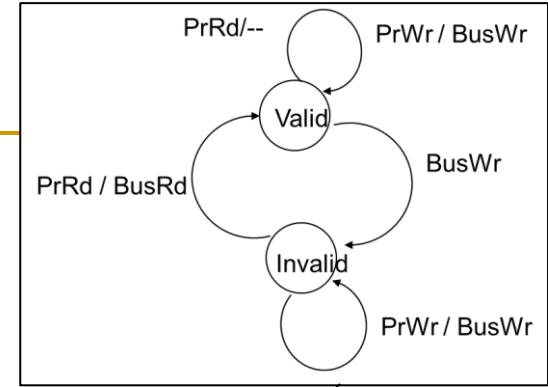
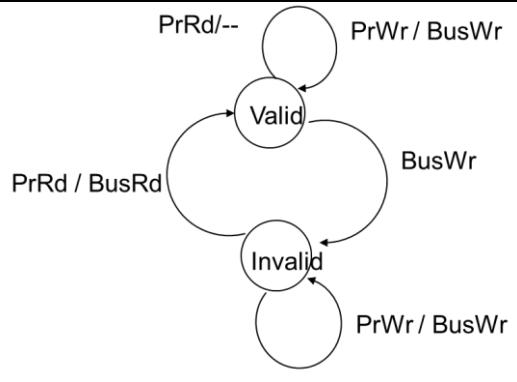


Interconnection Network
che realizza lo snoop bus

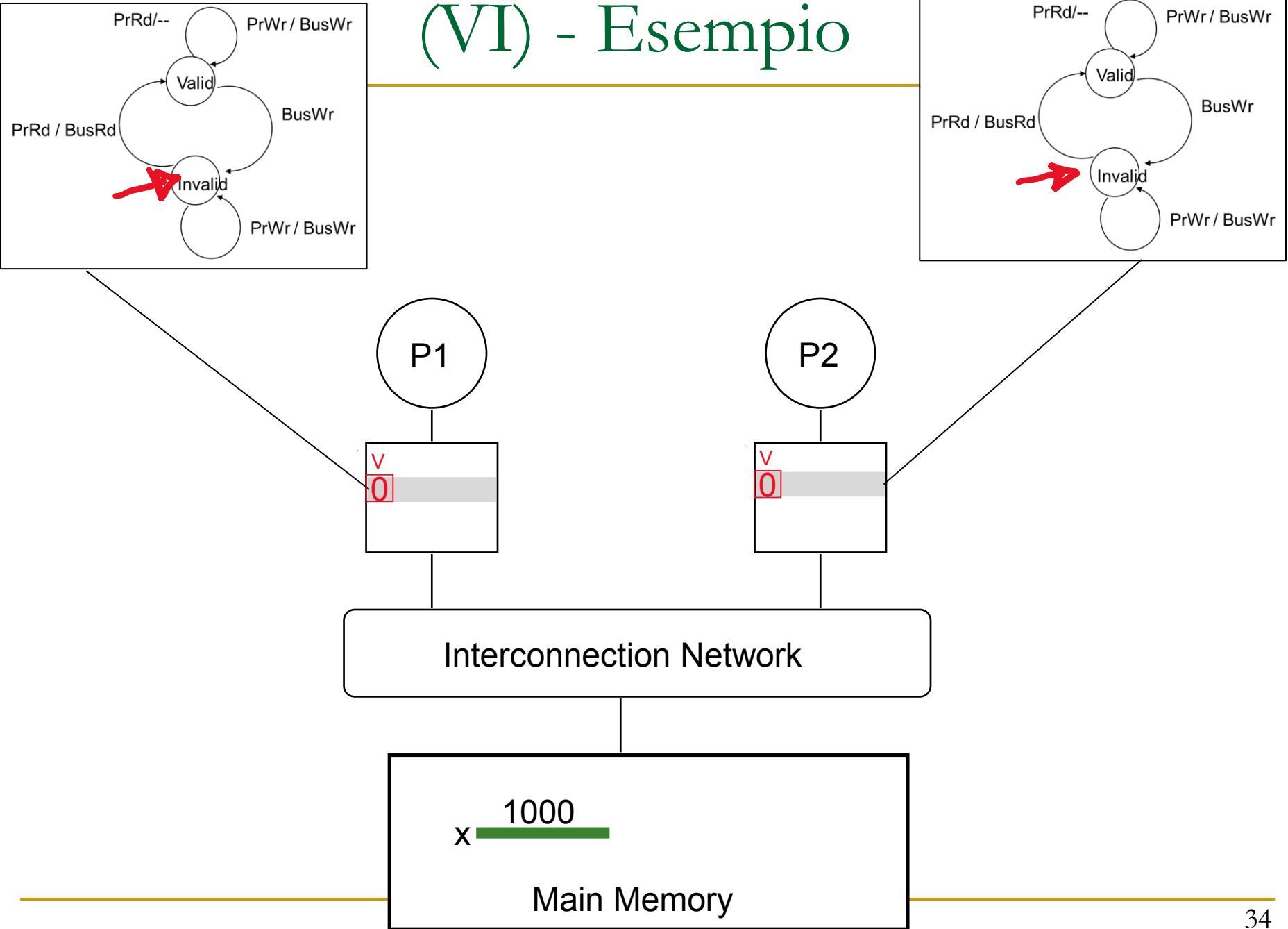
1000
 X

Main Memory

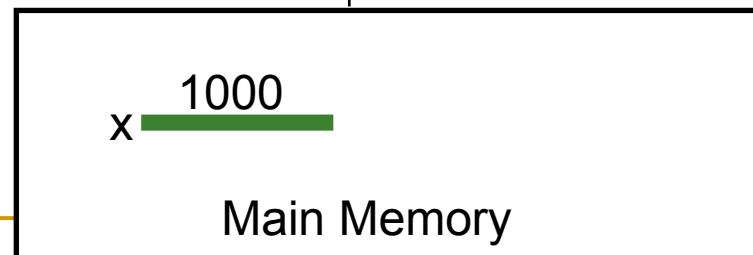
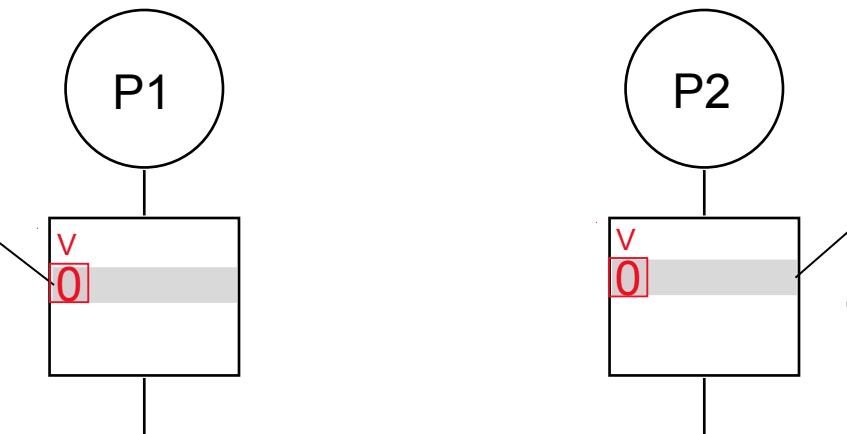
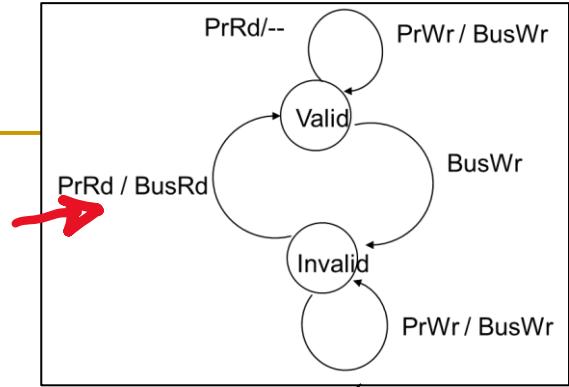
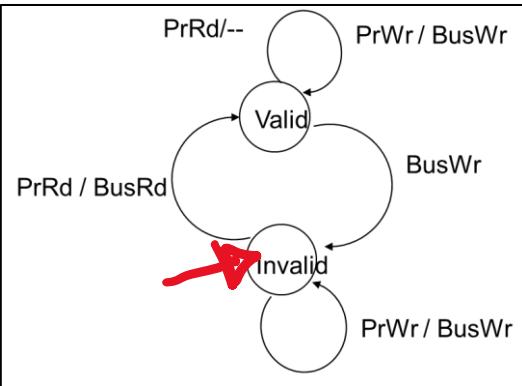
(VI) - Esempio



(VI) - Esempio

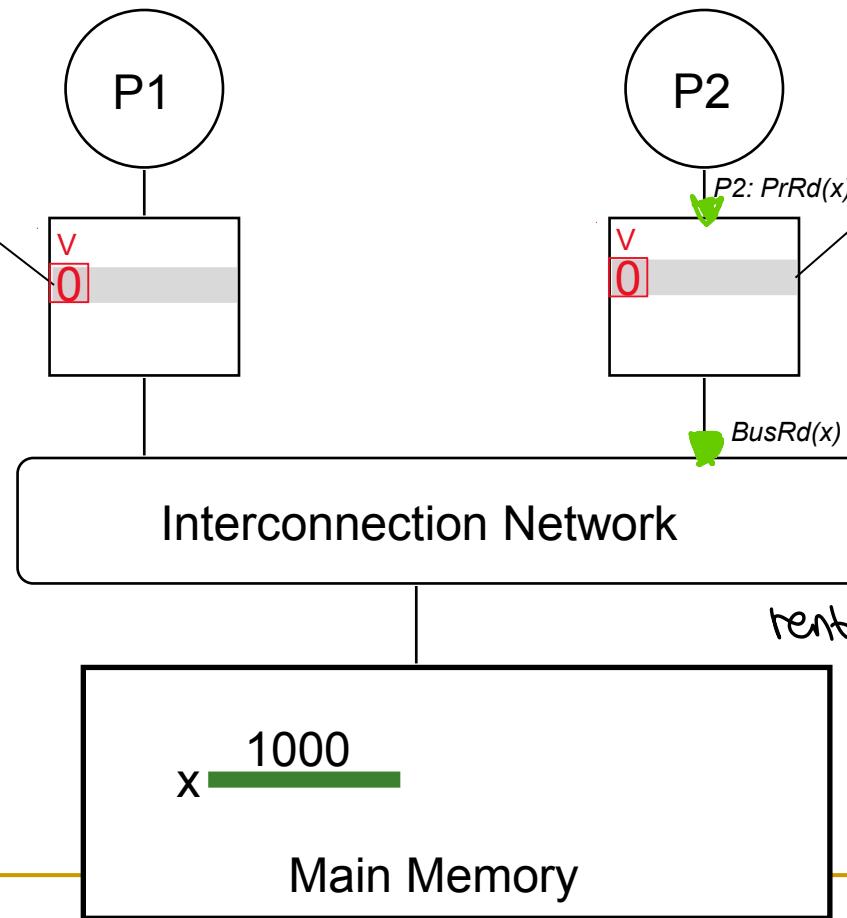
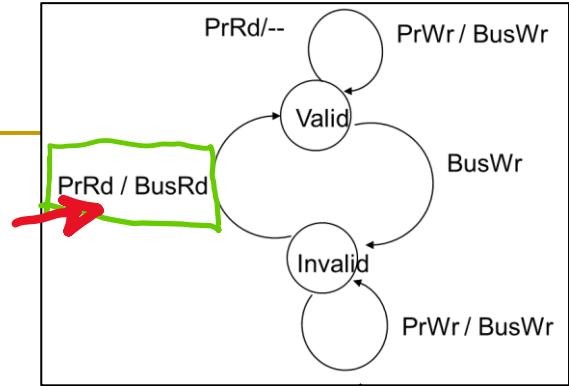
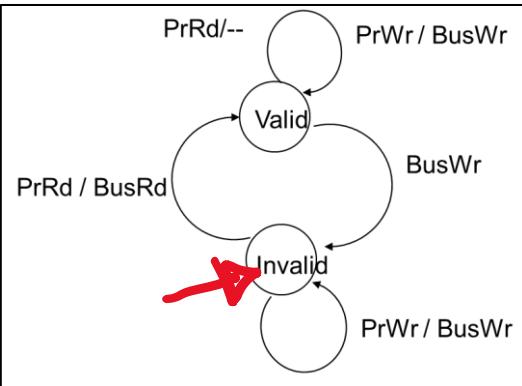


(VI) - Esempio



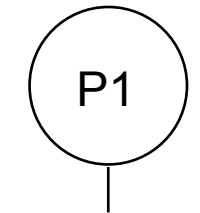
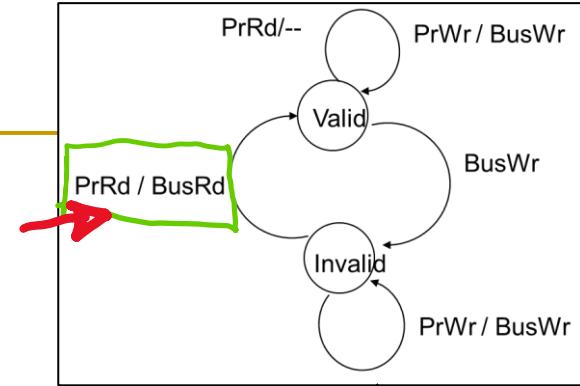
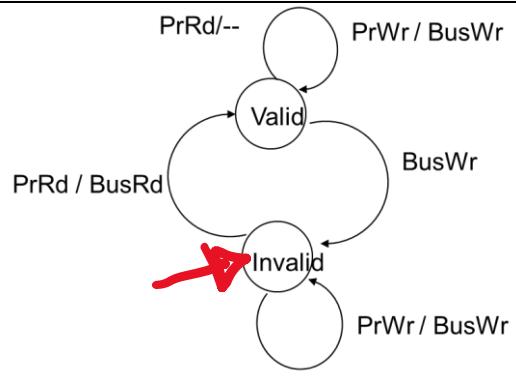
suppongo che P2 voglia fare un load da registro x ad R2.
 A seguito di questa operazione la load/store unit emette sul bus che connette il processore al cache controller una richiesta di lettura verso l'indirizzo x.
 Il cache controller verificherà se ha una hit / miss associata all'indirizzo, verificherà che sia una miss e emetterà sul bus di snoop un comando di bus read.

(VI) - Esempio



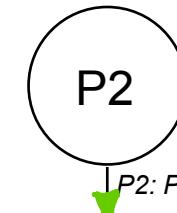
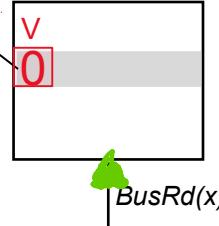
La bus read view
comunicate verso le
memorie e a tutti i
cache controller che deb-
bono essere monitorati coe-
rente con le cache di P2.

(VI) - Esempio

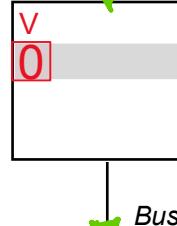


te cache contr. di P1
vede una richiesta
bus read verso l'
indirizzo X.

A seguito di questa richiesta il cache controllore del P1 deve vedere se l'indirizzo X è contenuto nella sua cache e verificare qual è lo stato del validity bit associato a quella linea di cache. Il valid bit è pari a zero, il cache controllore del processore uno non deve fare nulla quindi, perché dallo stato invalid non deve fare niente se osserva una bus read dal bus di snoop



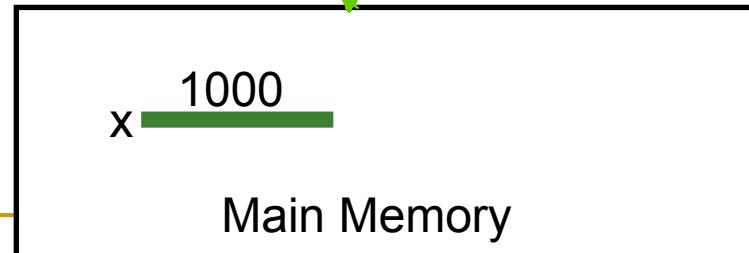
P2: PrRd(x)



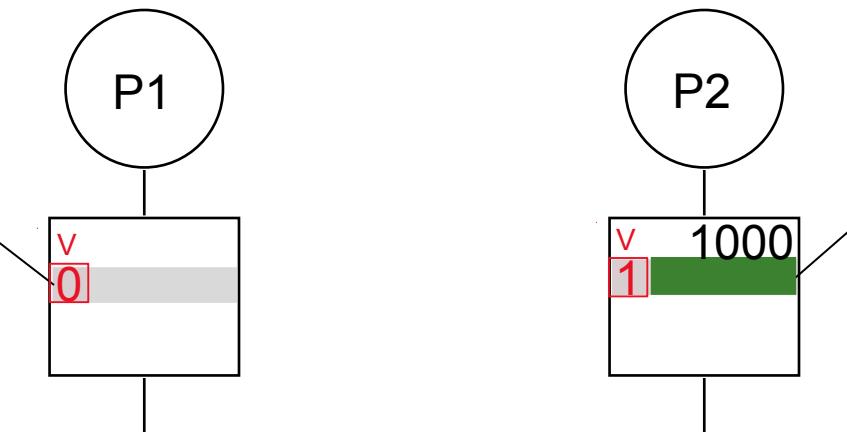
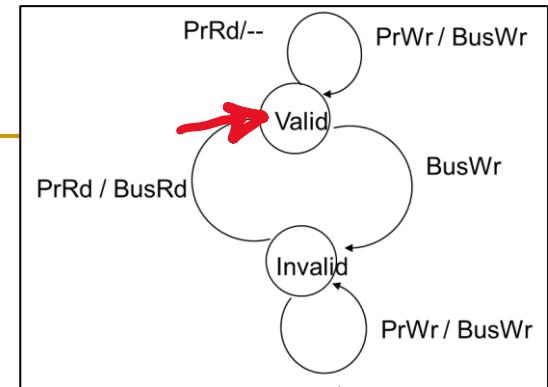
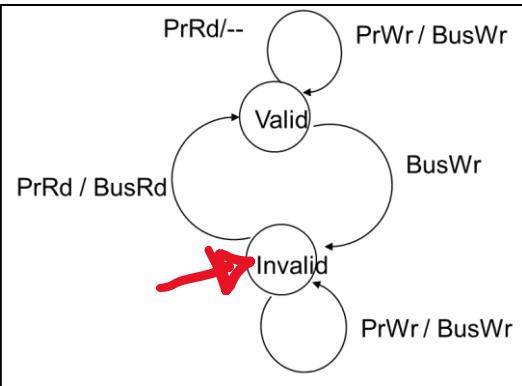
Id r2, x



BusRd(x)



(VI) - Esempio

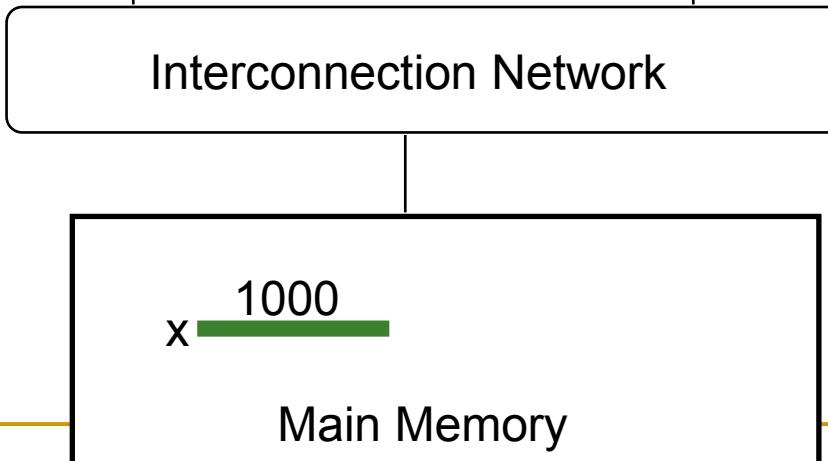


Id r2, x

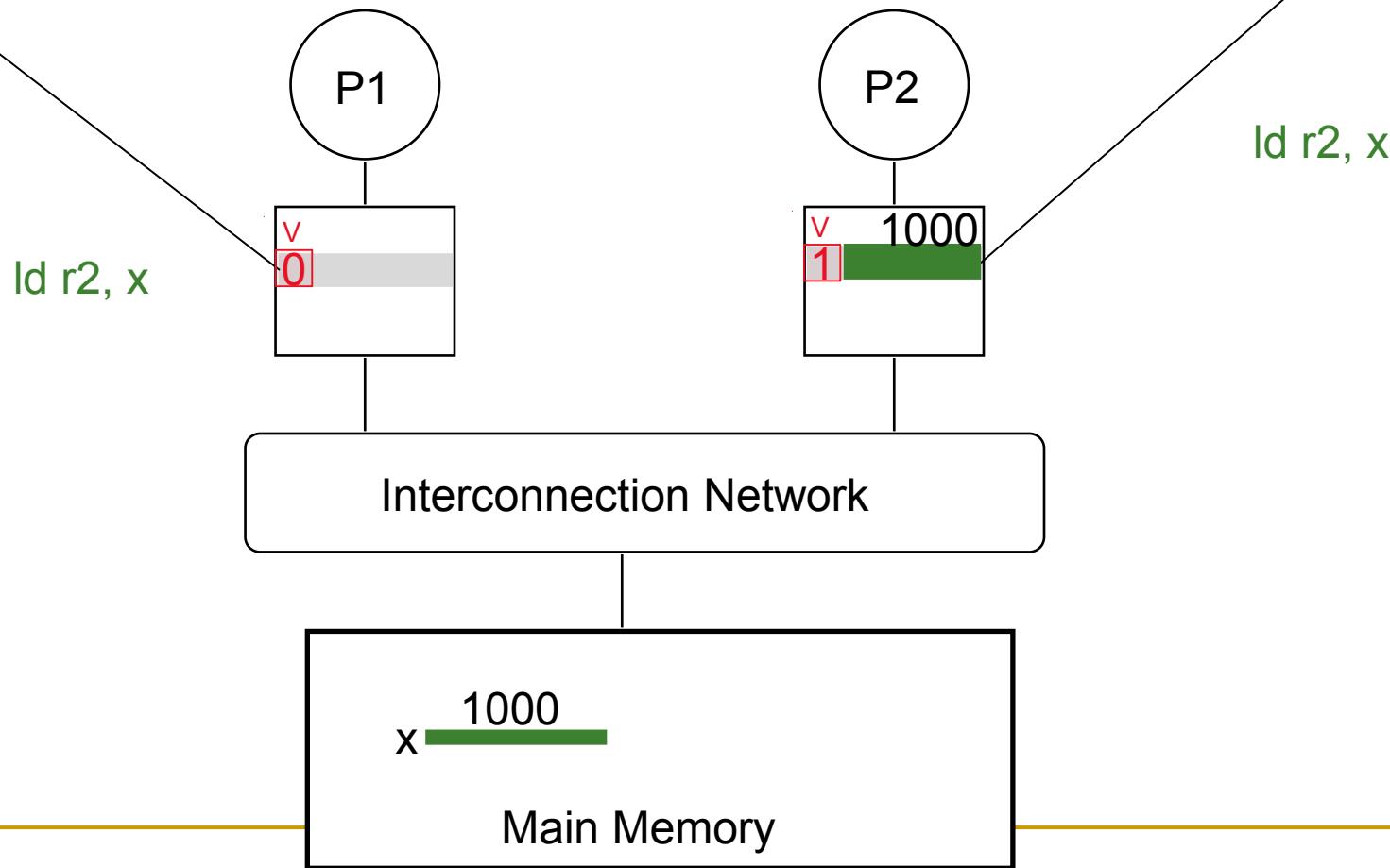
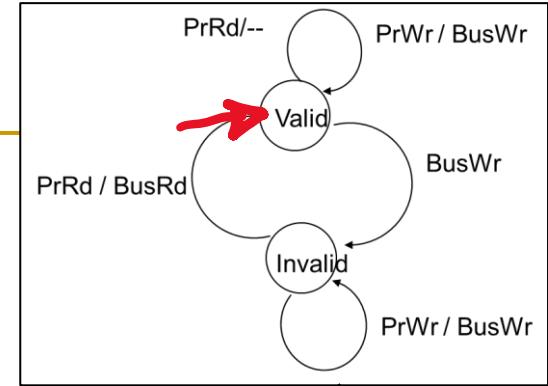
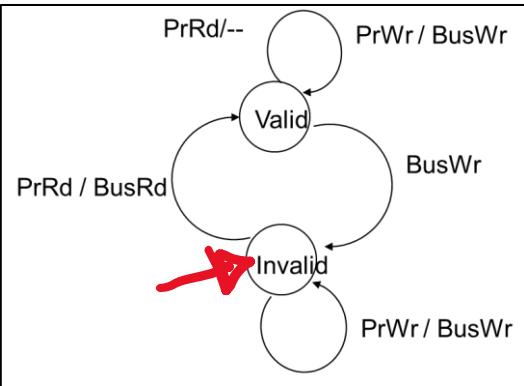
A seguito della bus read, il dato viene letto dalla memoria e portato nella cache privata del processore 2, che ora avrà bit valid pari a 1, perché ha una copia locale.

Ora se il P1 fa una lettura verso lo stesso indirizzo succede la stessa cosa, avendo P1 emetterà la richiesta di read, verrà verificata la hit o Miss e verrà fatta la richiesta al bus.

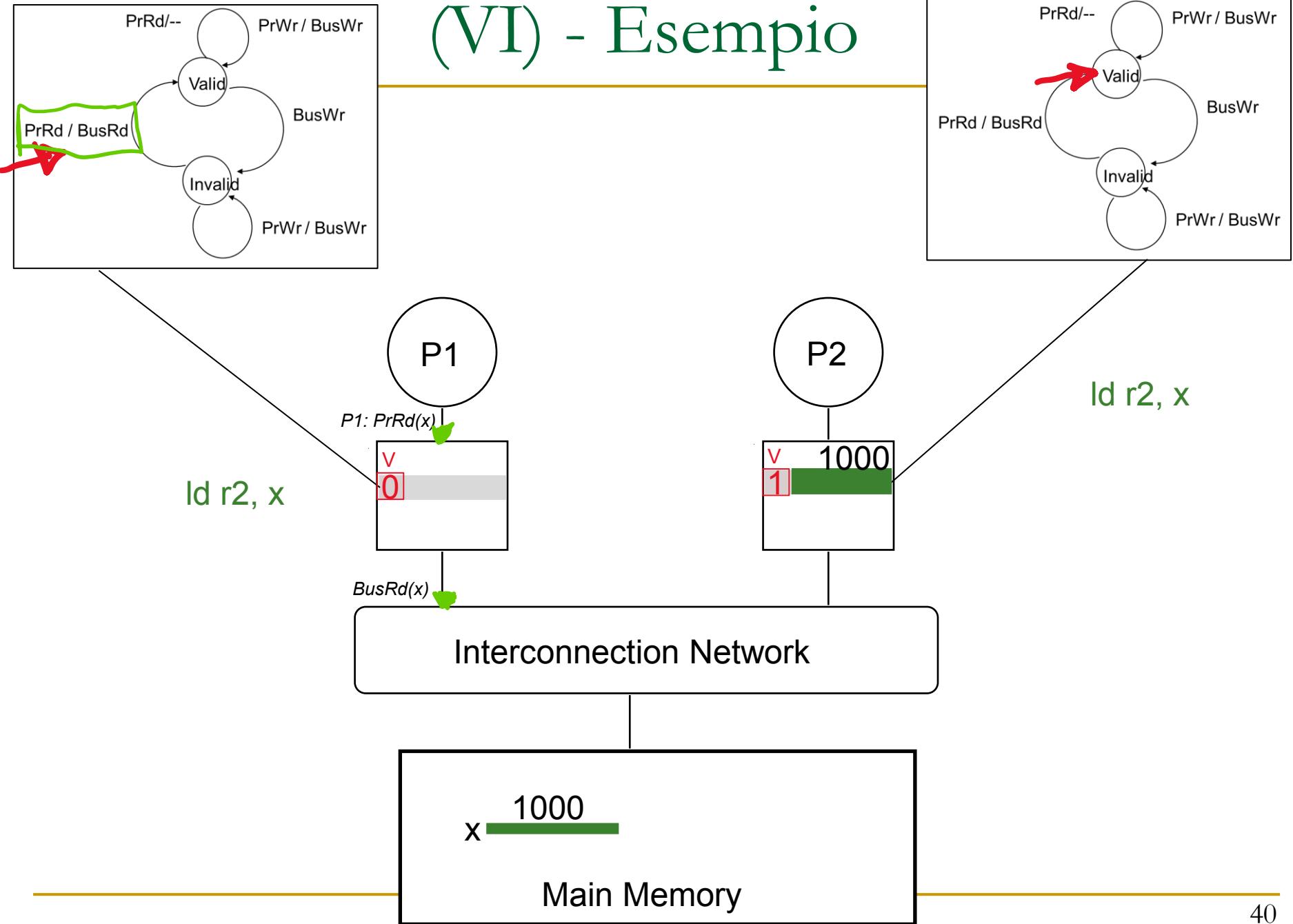
Succede che però p2 verificherà se ha una copia locale di quel dato, c'è l'ha e non deve fare nulla, la sua copia è coerente con la memoria.



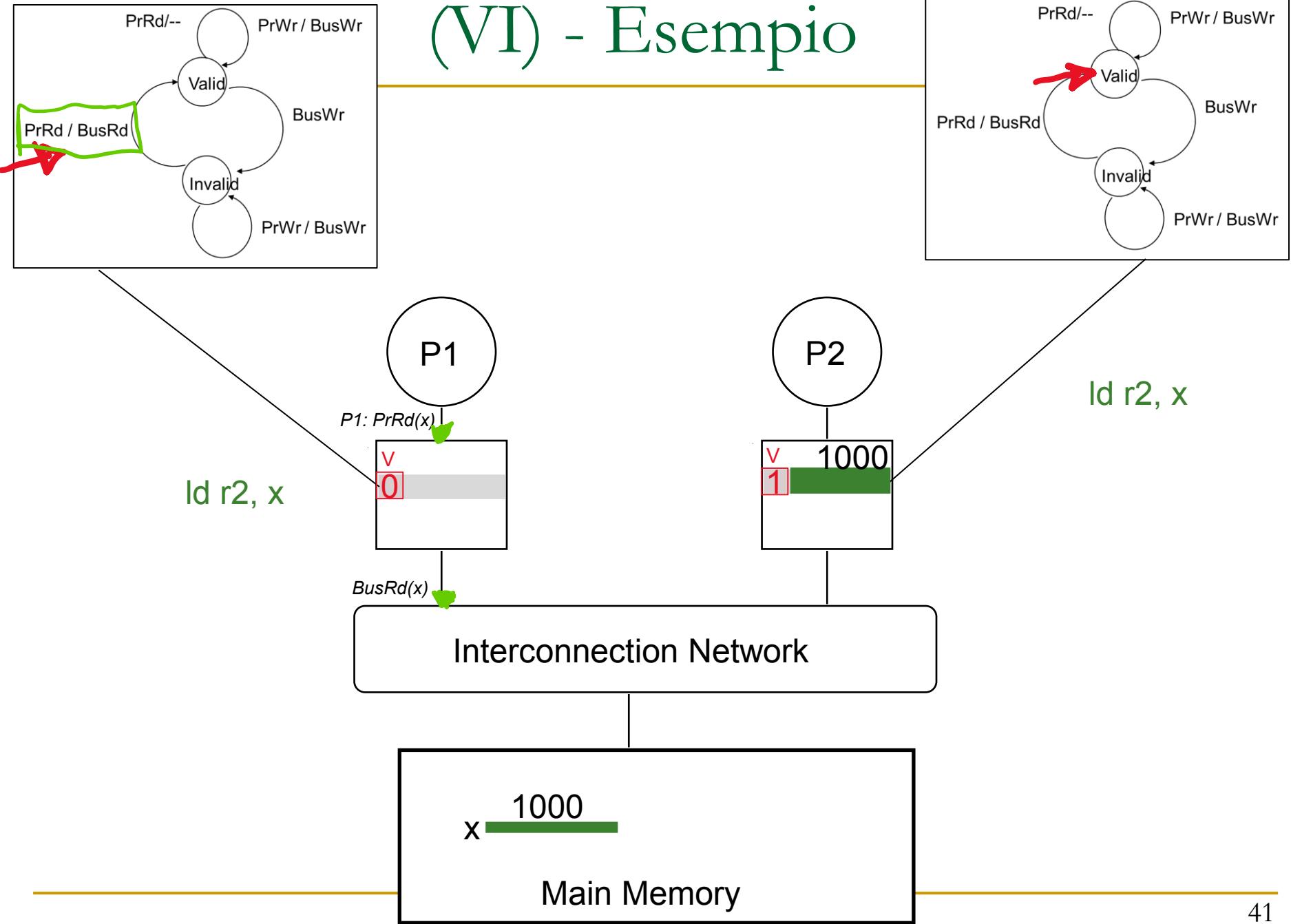
(VI) - Esempio



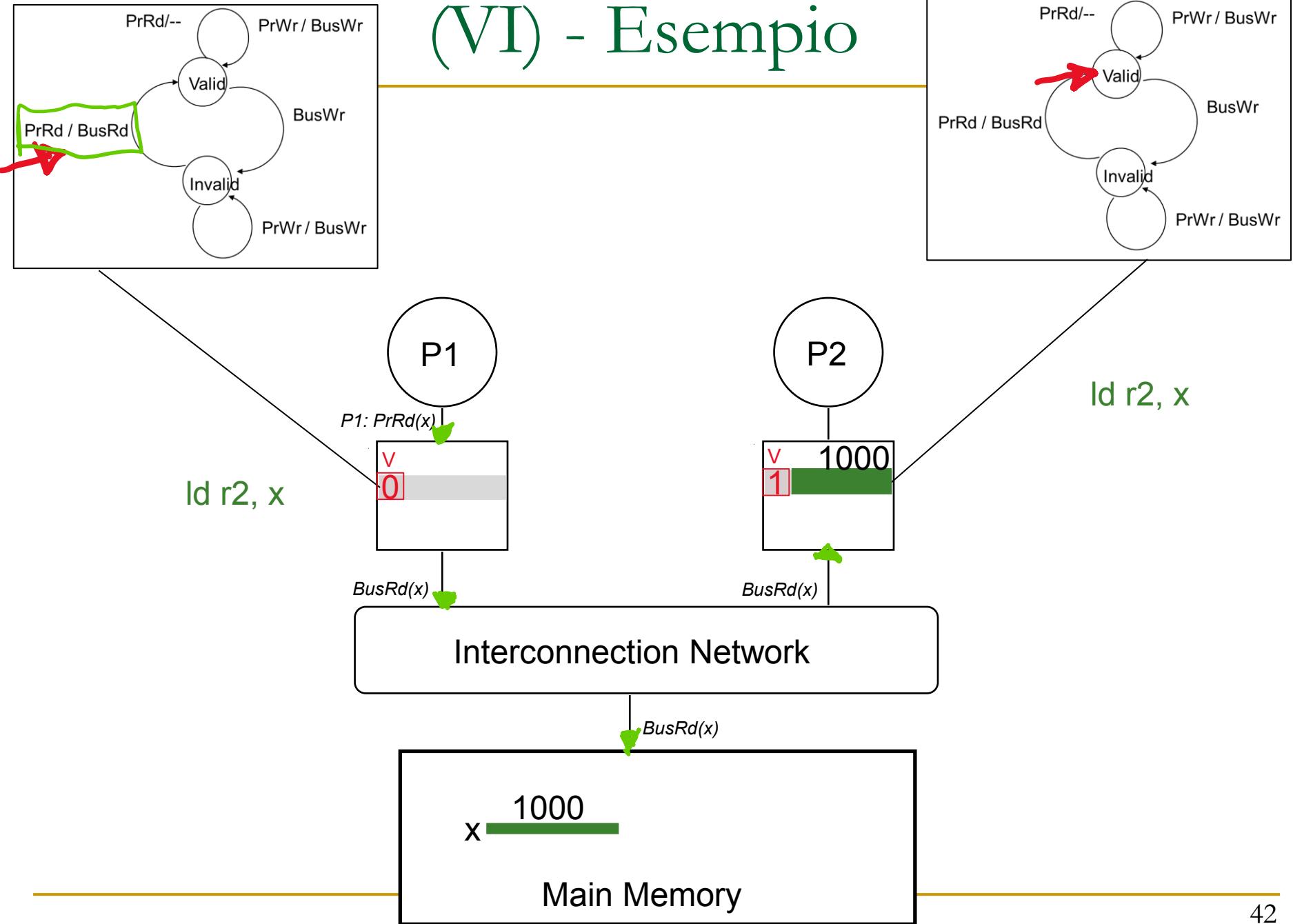
(VI) - Esempio



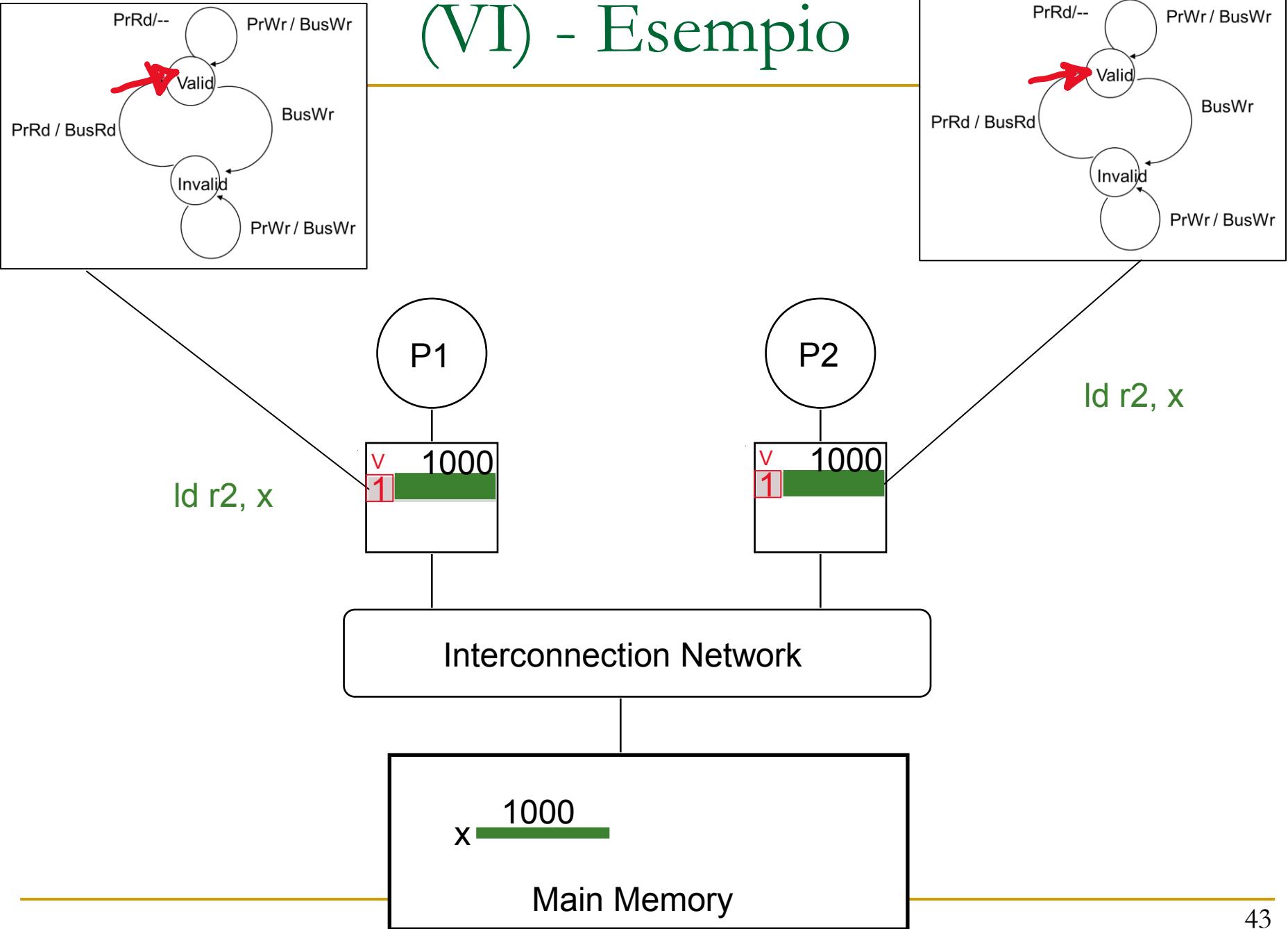
(VI) - Esempio



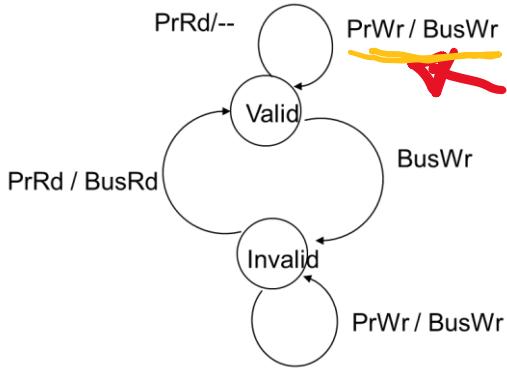
(VI) - Esempio



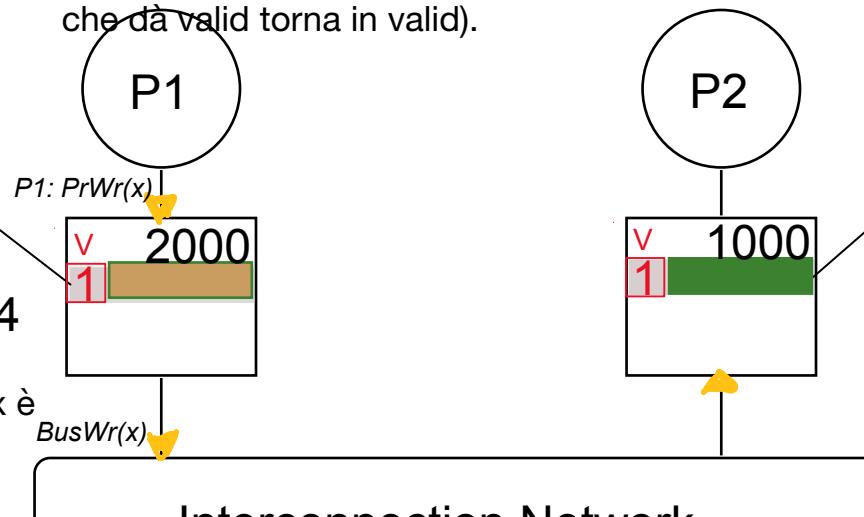
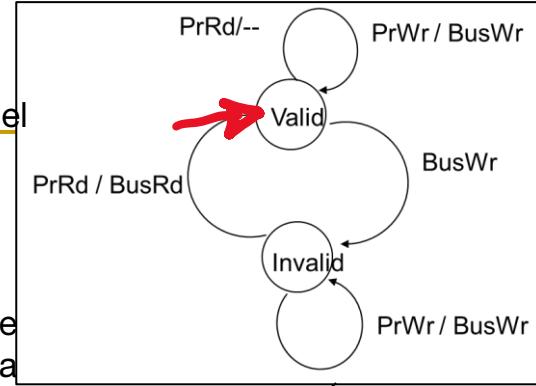
(VI) - Esempio



(VI) - Esempio



Se P1 fa un'operazione di store: la load store uniti del p1 emette la richiesta di scrittura verso quell'indirizzo al proprio cache controller, che verifica e ha una hit perché quel dato c'è già dentro e il cache controller di p1 si trova in stato valid, osserva in ingresso una richiesta di scrittura e quindi deve emettere sul bus di snoop una richiesta di Write verso quell'indirizzo. (si trova nella freccia che da valid torna in valid).

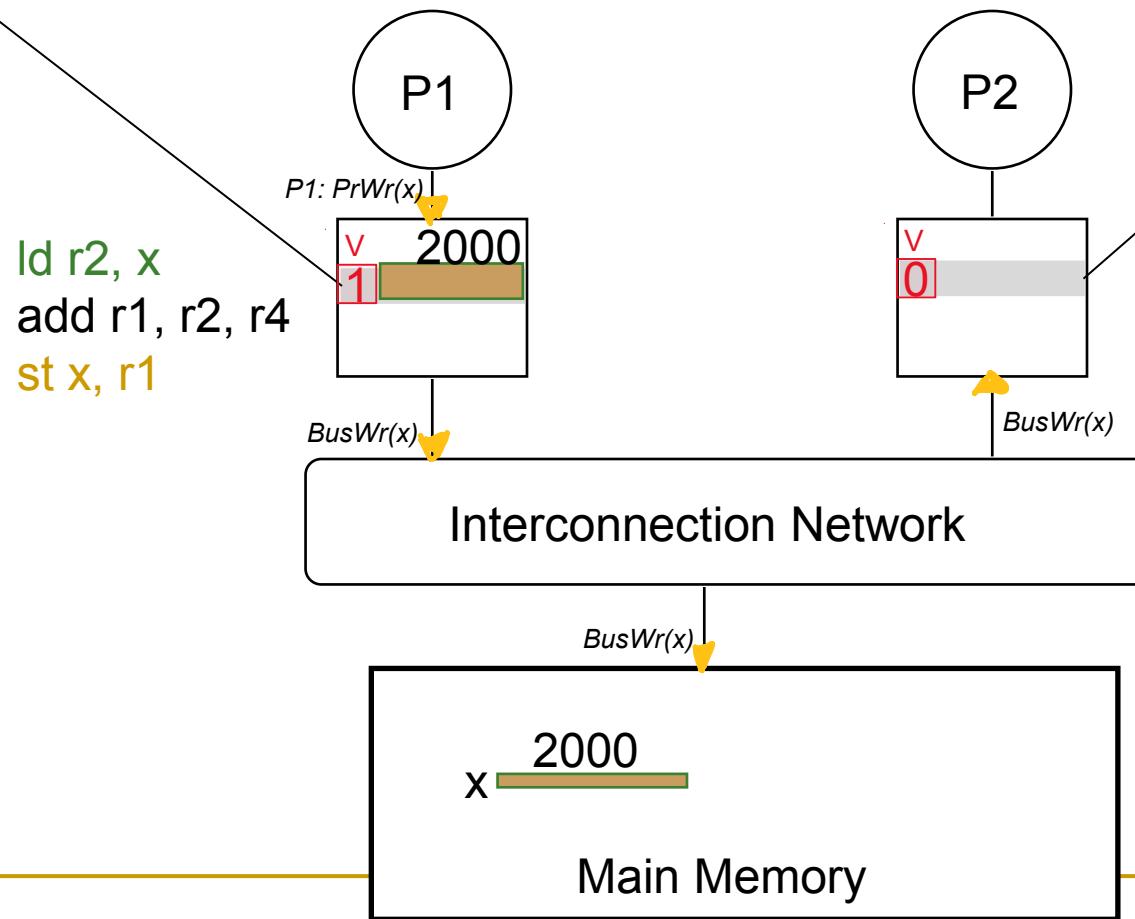
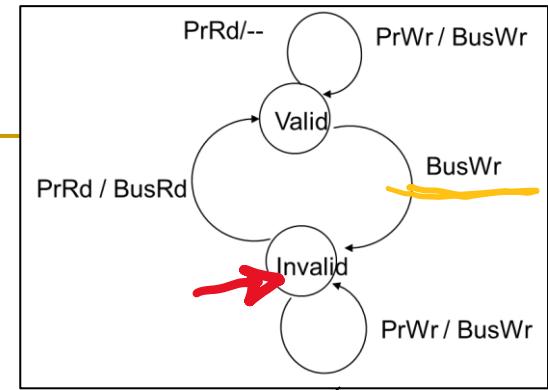
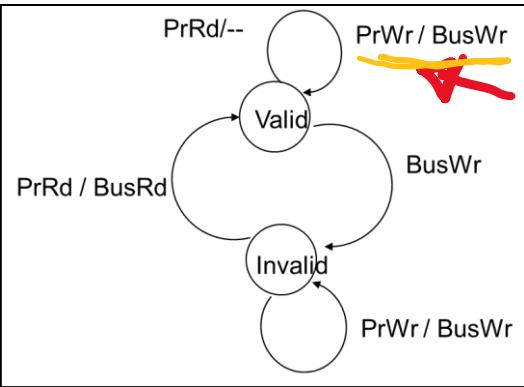


La buia Write verso l'indirizzo x è legata alla gestione della hit in caso di scrittura tramite protocollo Write through, cioè devo scriverlo anche negli altri livelli di memoria. Questa comunicazione viene inviata sia all' memoria che aggiorna il suo dato, ma anche al Cache controller del processore 2. Questo porta ad invalido la sua copia locale di X.

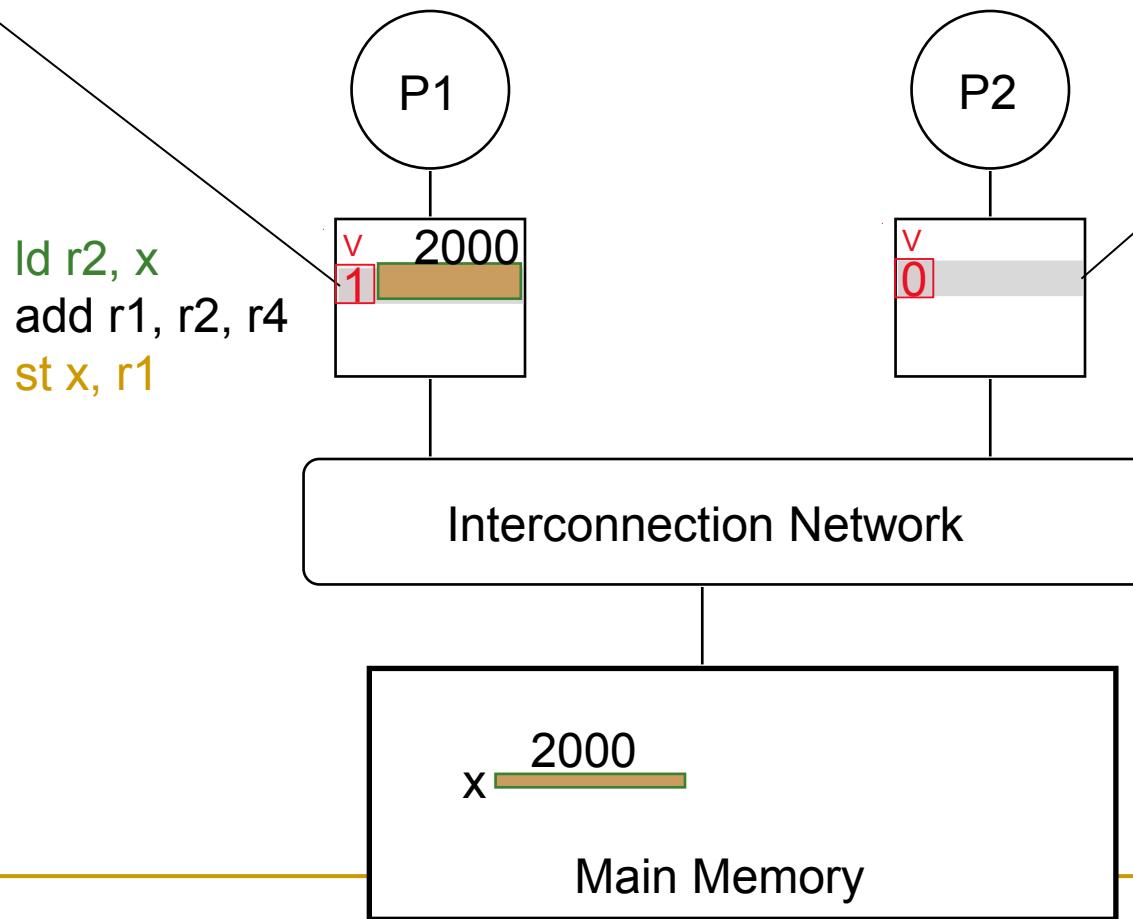
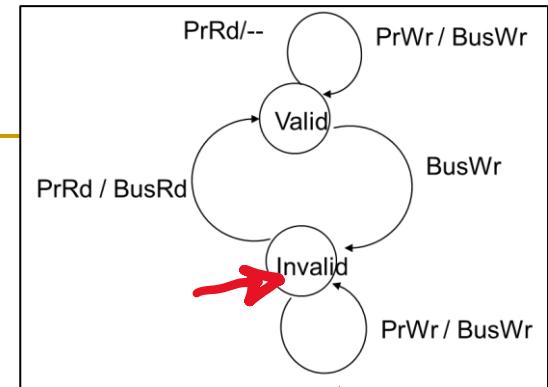
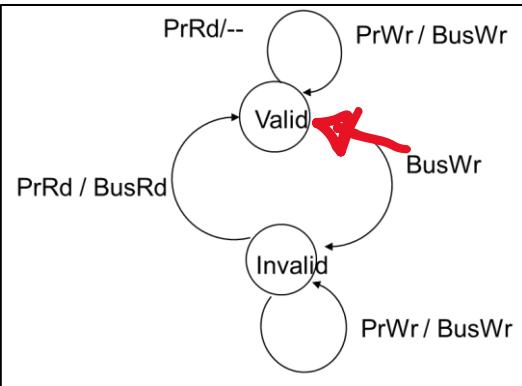
$Id\ r2, x$

$Id\ r2, x$
add r1, r2, r4
st x, r1

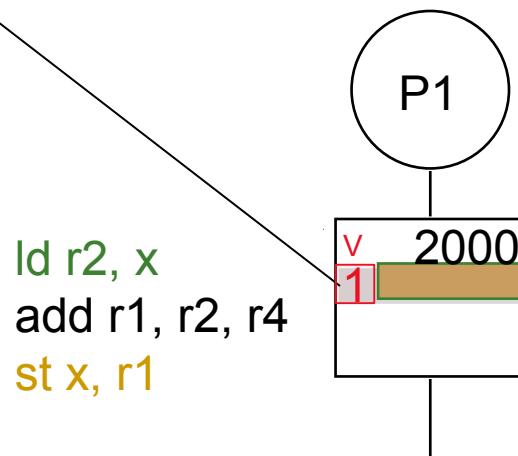
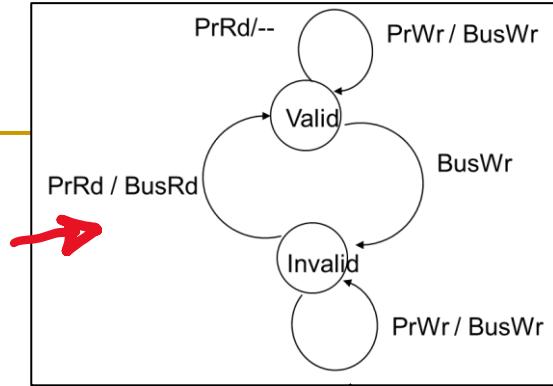
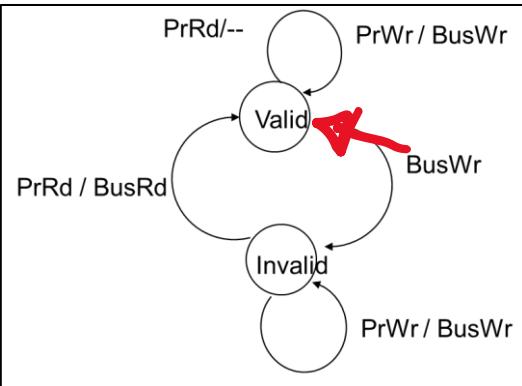
(VI) - Esempio



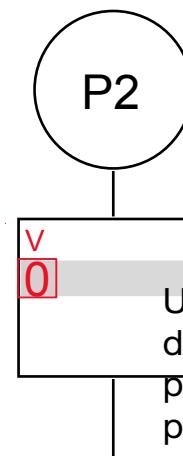
(VI) - Esempio



(VI) - Esempio

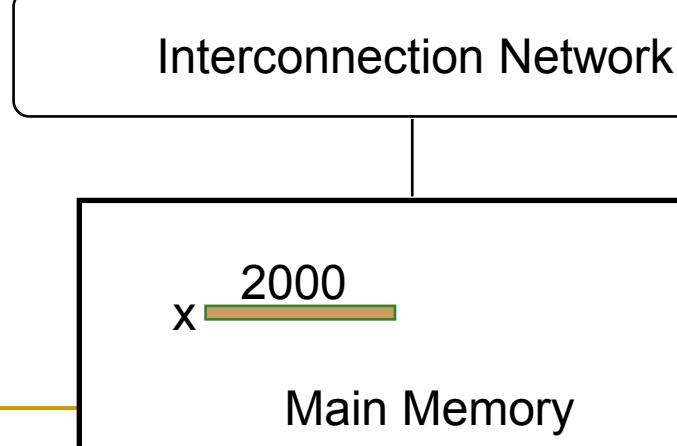


Id r2, x
add r1, r2, r4
st x, r1

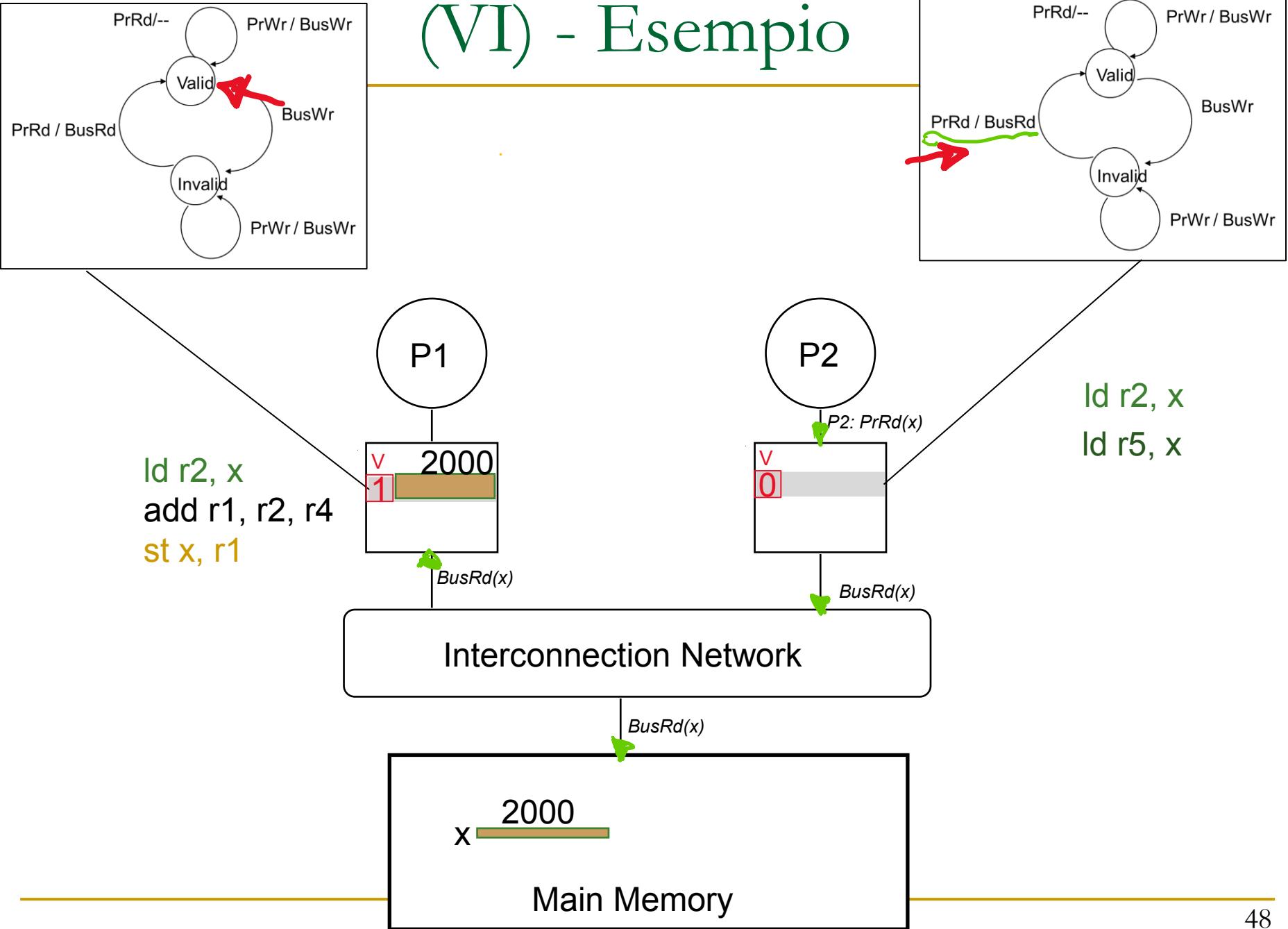


Id r2, x
Id r5, x

Una successiva lettura del dato da parte di p2 viene gestita come prima, ovvero il processore 2 non ha una copia locale , il p1 ha una copia locale coerente con la memoria, quindi per gestire la load di p2, questa darà origine a una richiesta da parte di p2 di una processor read all' indirizzo X, il cache controller verifica che non ha una copia locale del dato e emette una bus read verso l'interconnect. A seguito della bus read, questa viene comunicata a tutti i cache controller che devono rimanere coerenti e viene comunicata anche alla main memory che propaga il dato richiesto al cache controller.

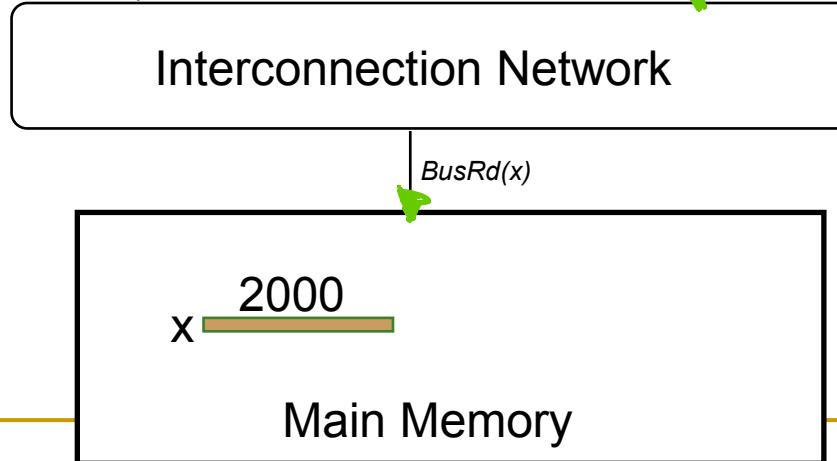
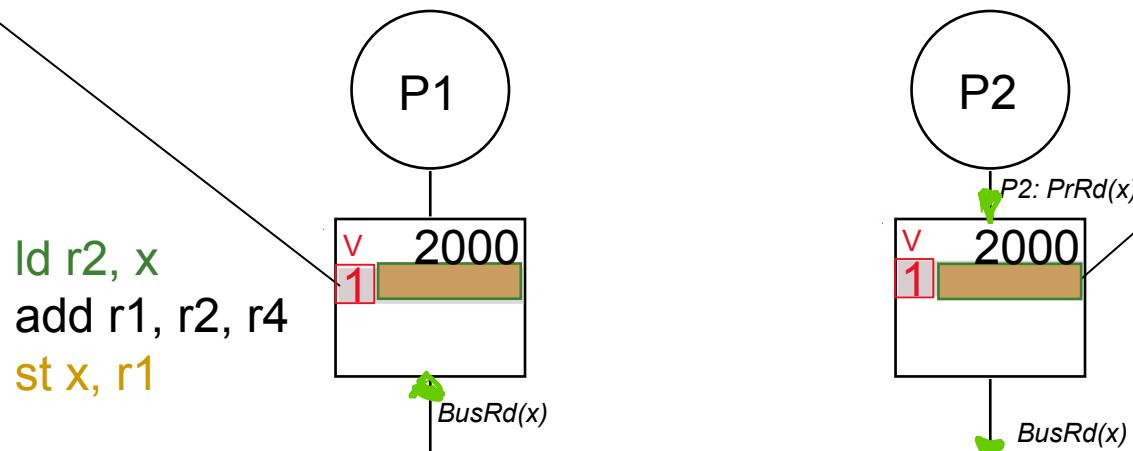
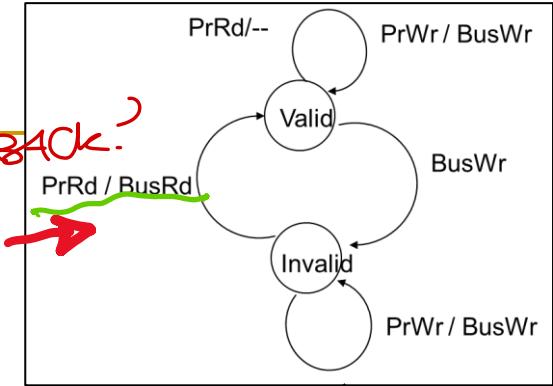
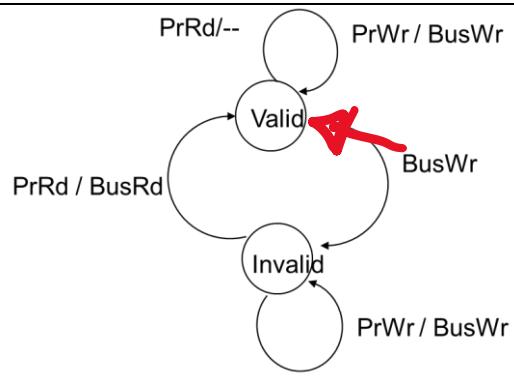


(VI) - Esempio



(VI) - Esempio

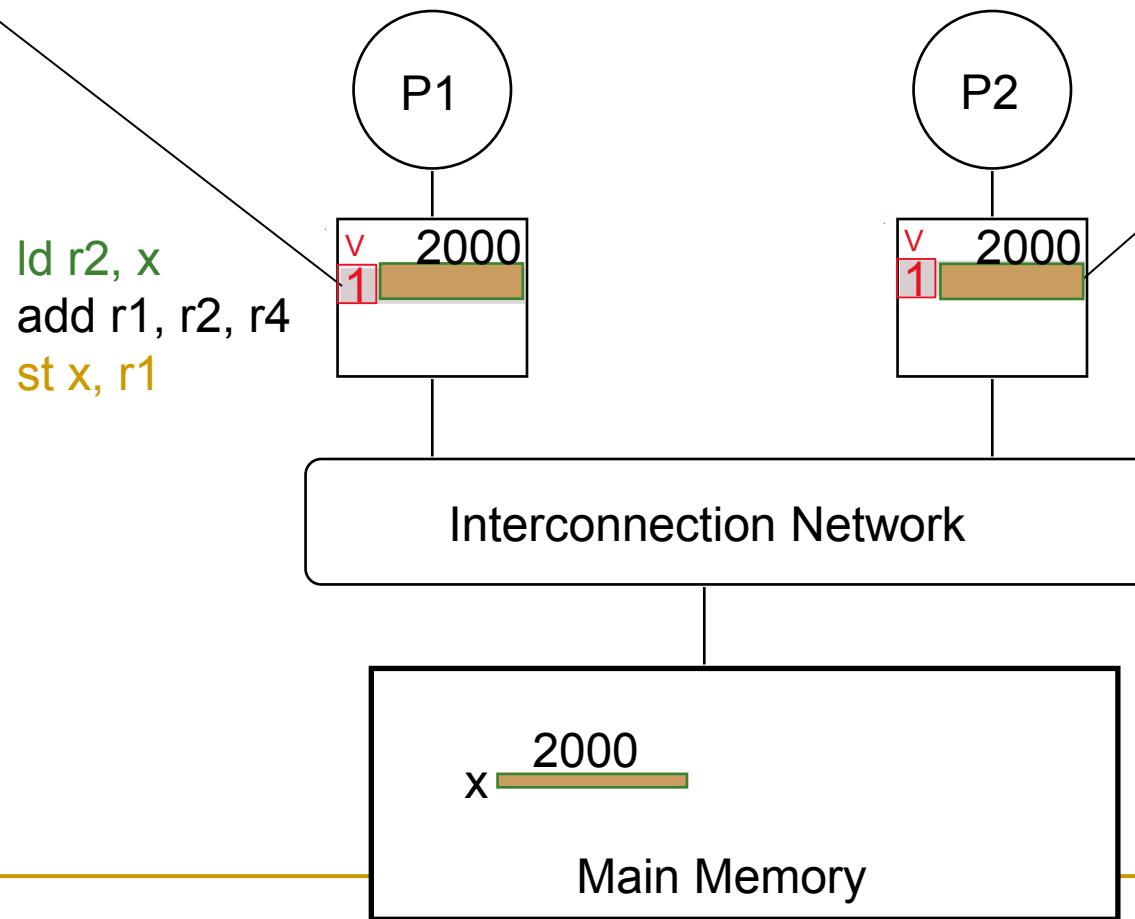
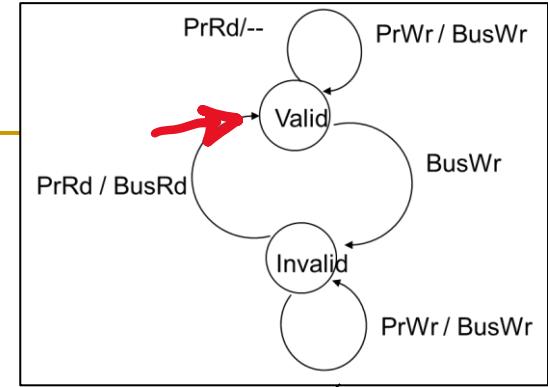
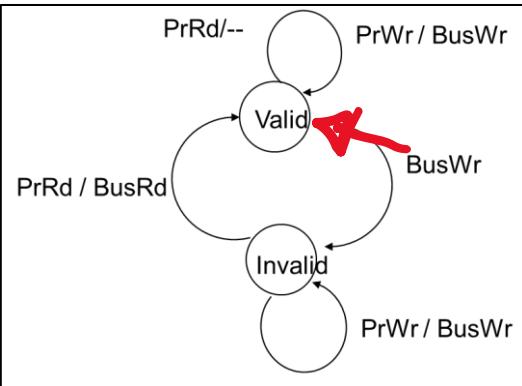
COSA SUCCEDE SE USO POLIZIA WRITE BACK?



Id r2, x
add r1, r2, r4
st x, r1

Id r2, x
Id r5, x

(VI) - Esempio



Extending the Protocol

- What if you want write-back caches?
 - We want a “modified” state

A More Sophisticated Protocol: MSI

- Extend metadata per block to encode three states:
 - **M**(odified): cache line is the only cached copy and is dirty
 - **S**(hared): cache line is one of potentially several cached copies and it is clean (i.e., at least one clean cached copy)
 - **I**(nvalid): cache line is not present in this cache

A giungo uno stato: MODIFICATO

XÉ MI PRECIO
PUNACIA?

- Read miss makes a *Read* request on bus, transitions to **S**
- Write miss makes a *ReadEx* request, transitions to **M** state
- When a processor snoops *ReadEx* from another writer, it must invalidate its own copy (if any)
- *S → M upgrade* can be made without re-reading data from memory (via *Invalidations*)

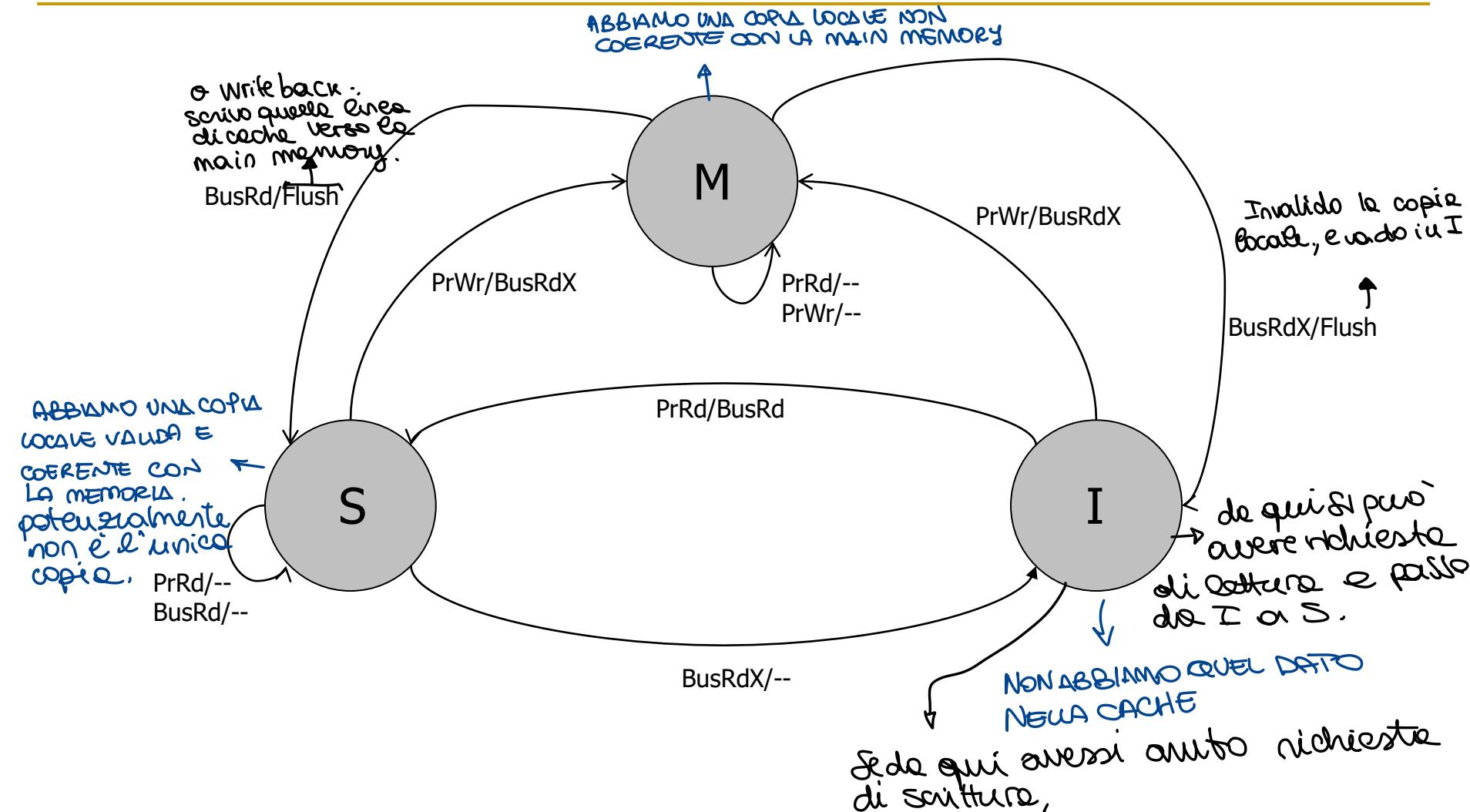
EXCLUSIVUS

Cose se voglio fare una modifica devo averne unica copia, koerentata.

Se il proc. fa store, il cache controller non comincia subito a scambiare ma comunica che vuole avere una copia esclusiva, per cui gli altri devono invalidarsi.

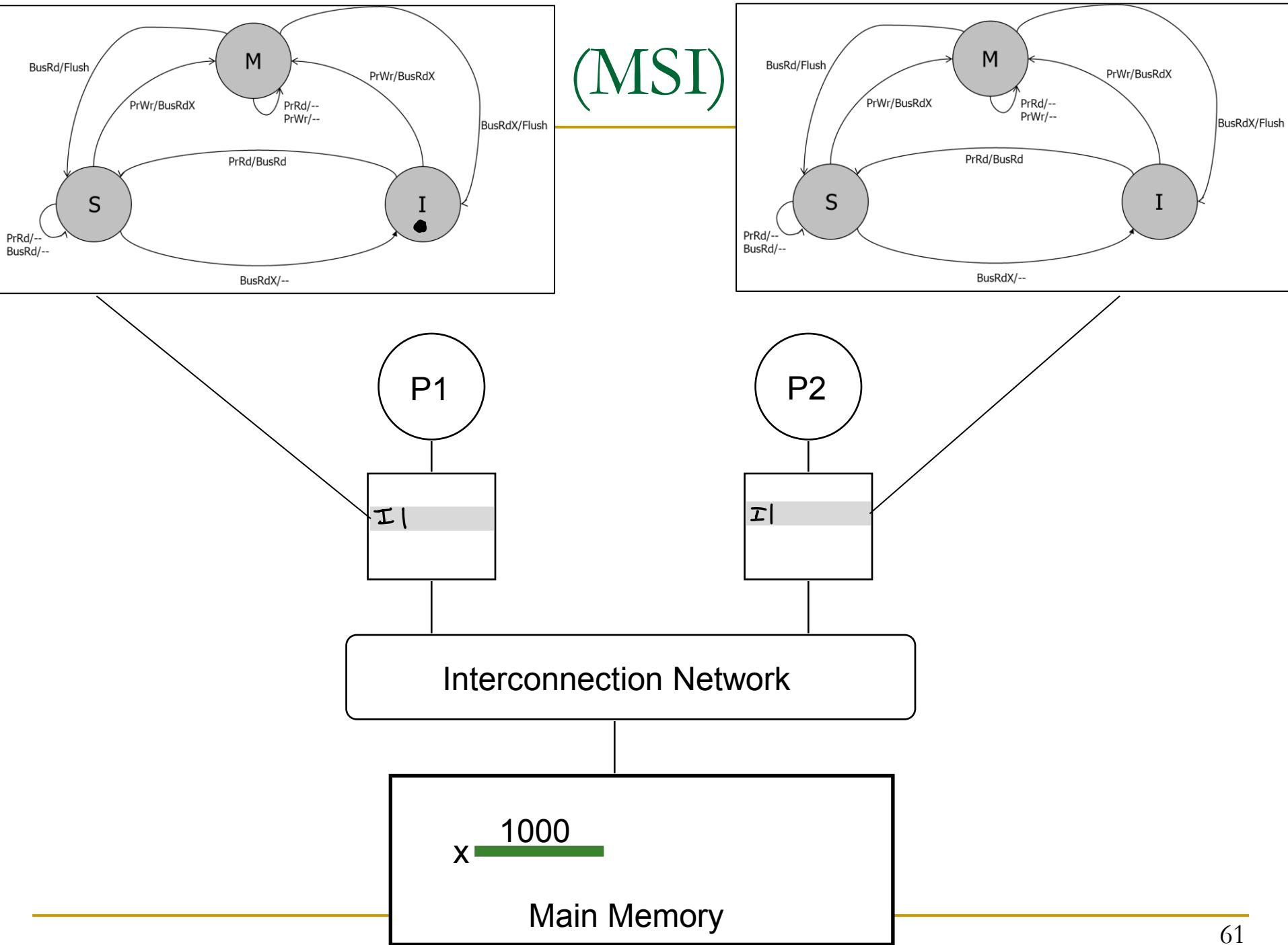
MSI State Machine

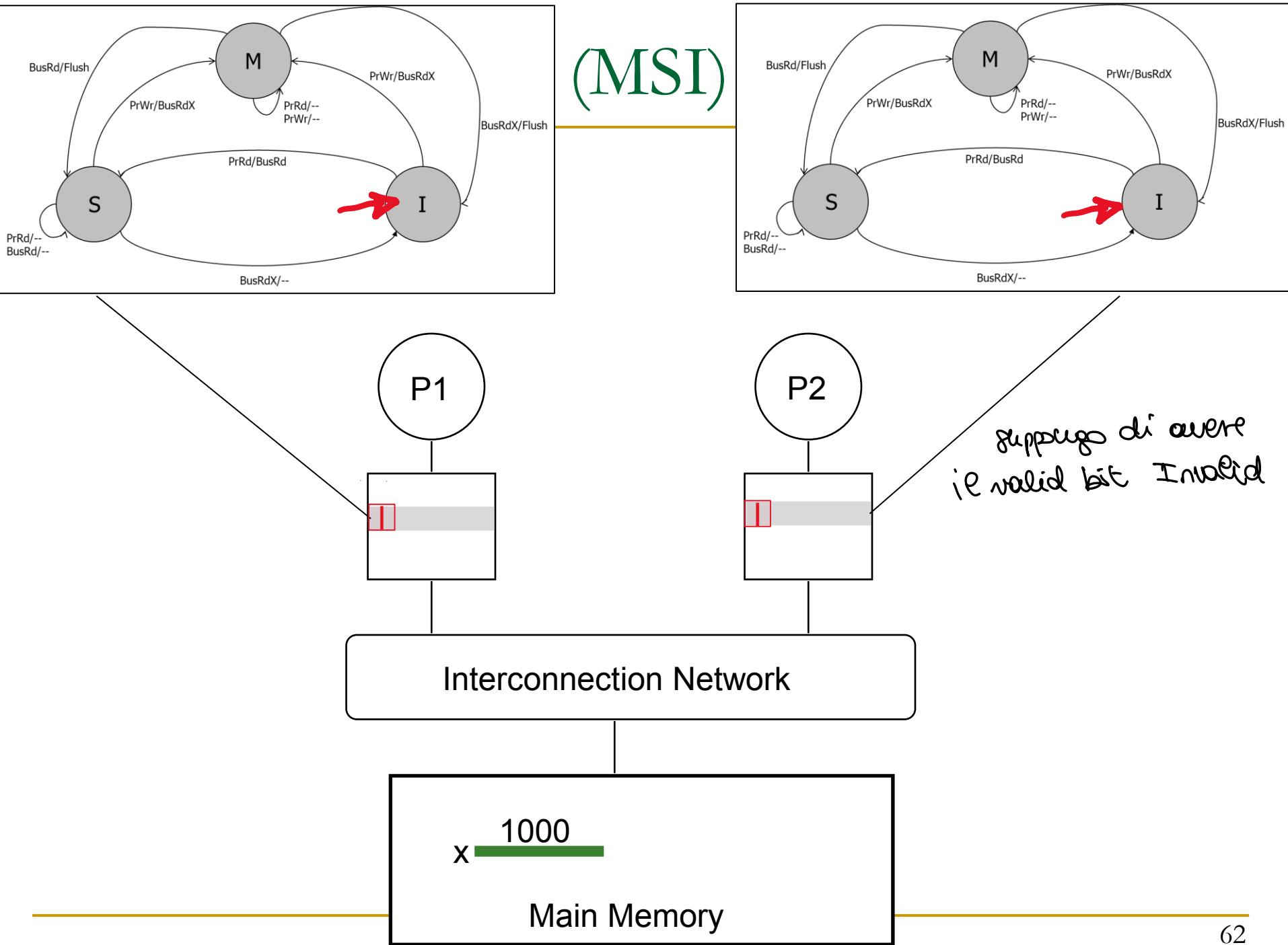
La linea di cache si può trovare in uno di questi tre stati: M/S/I.

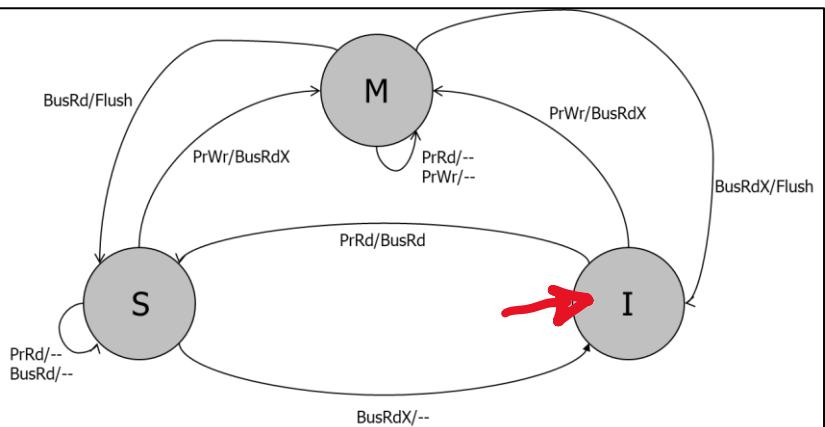


ObservedEvent/Action

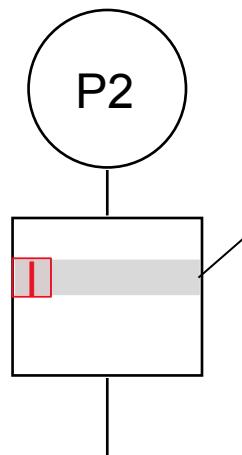
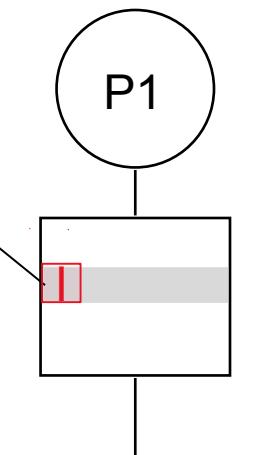
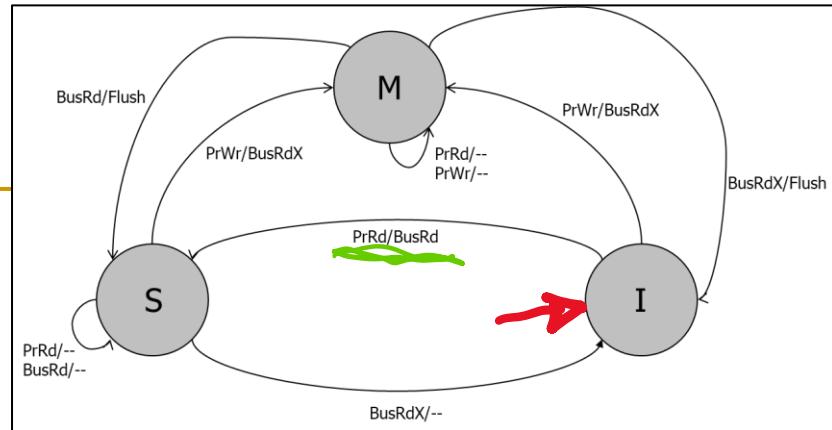
[Culler/Singh96]







(MSI)



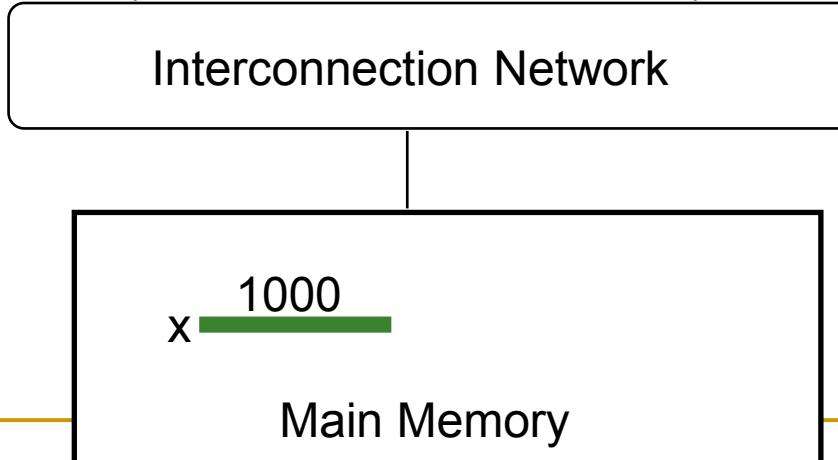
Id r2, x

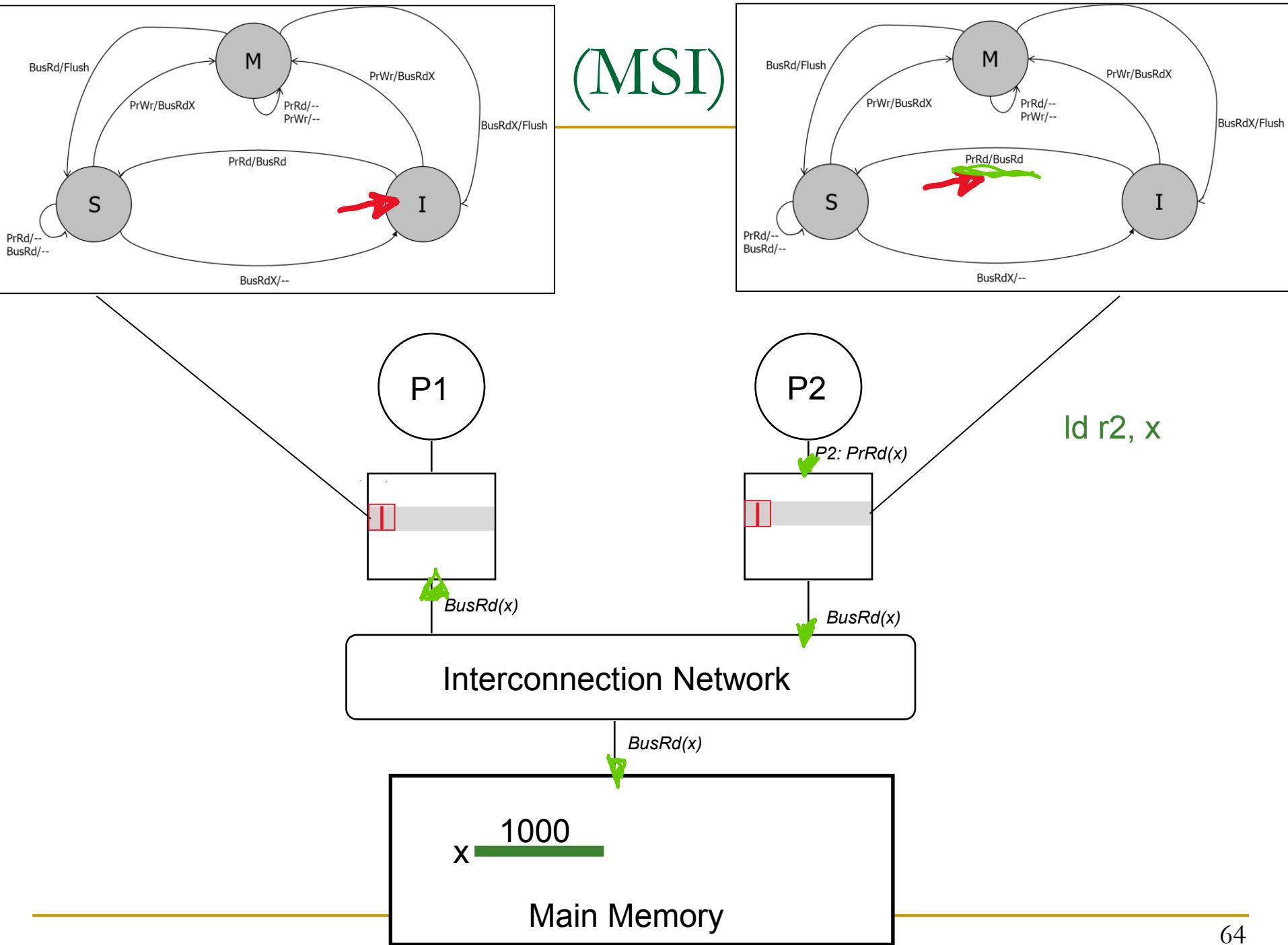
suppongo di fare una load verso R2.

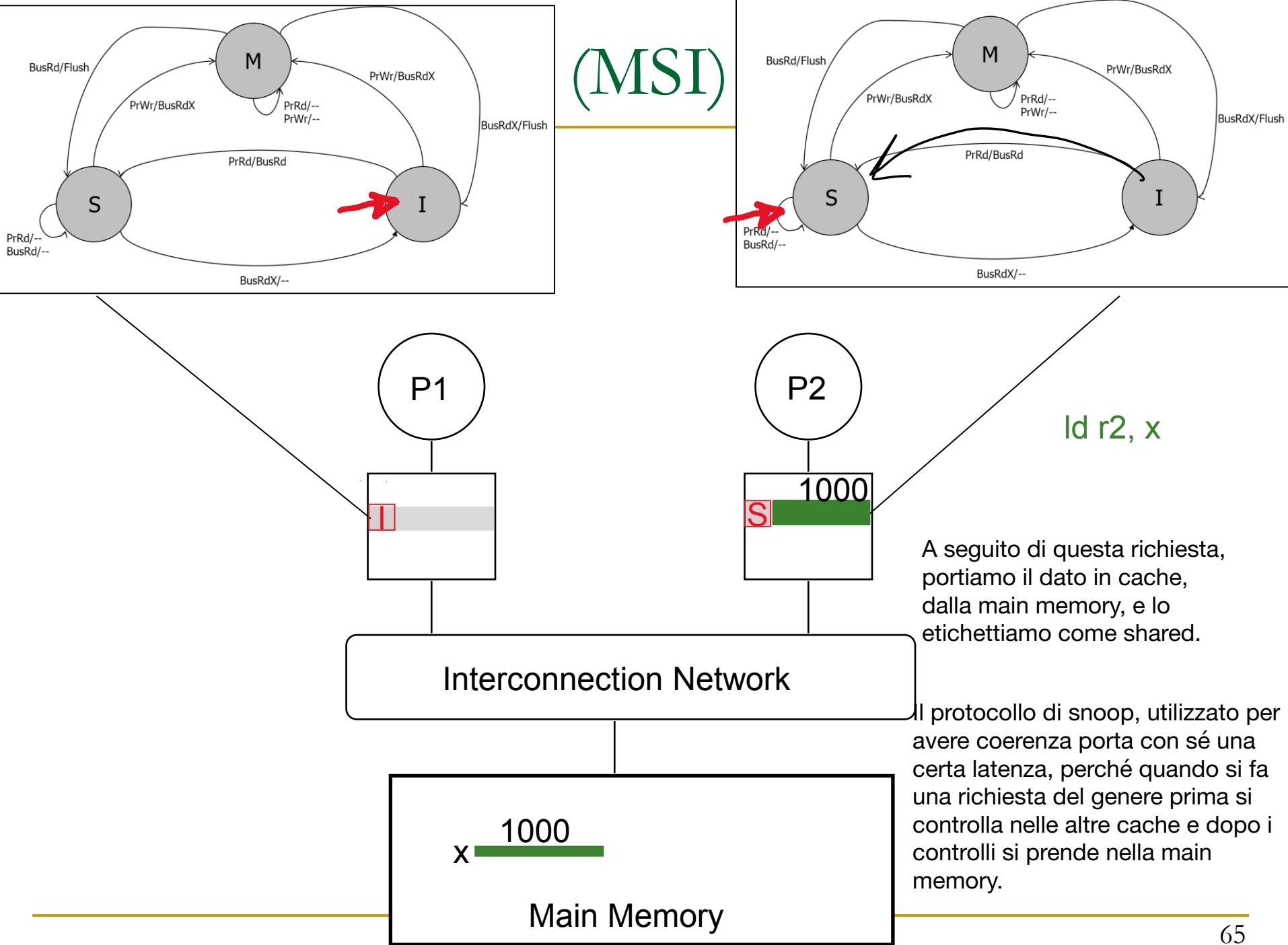
Siamo nello stato invalido, il p2 emetterà una richiesta processor read all' indirizzo X, questa verrà inoltrata alla logica di cache che risponderà con una Miss e propagherà verso il bus di snoop la richiesta di bus read non esclusiva dell' indirizzo X.

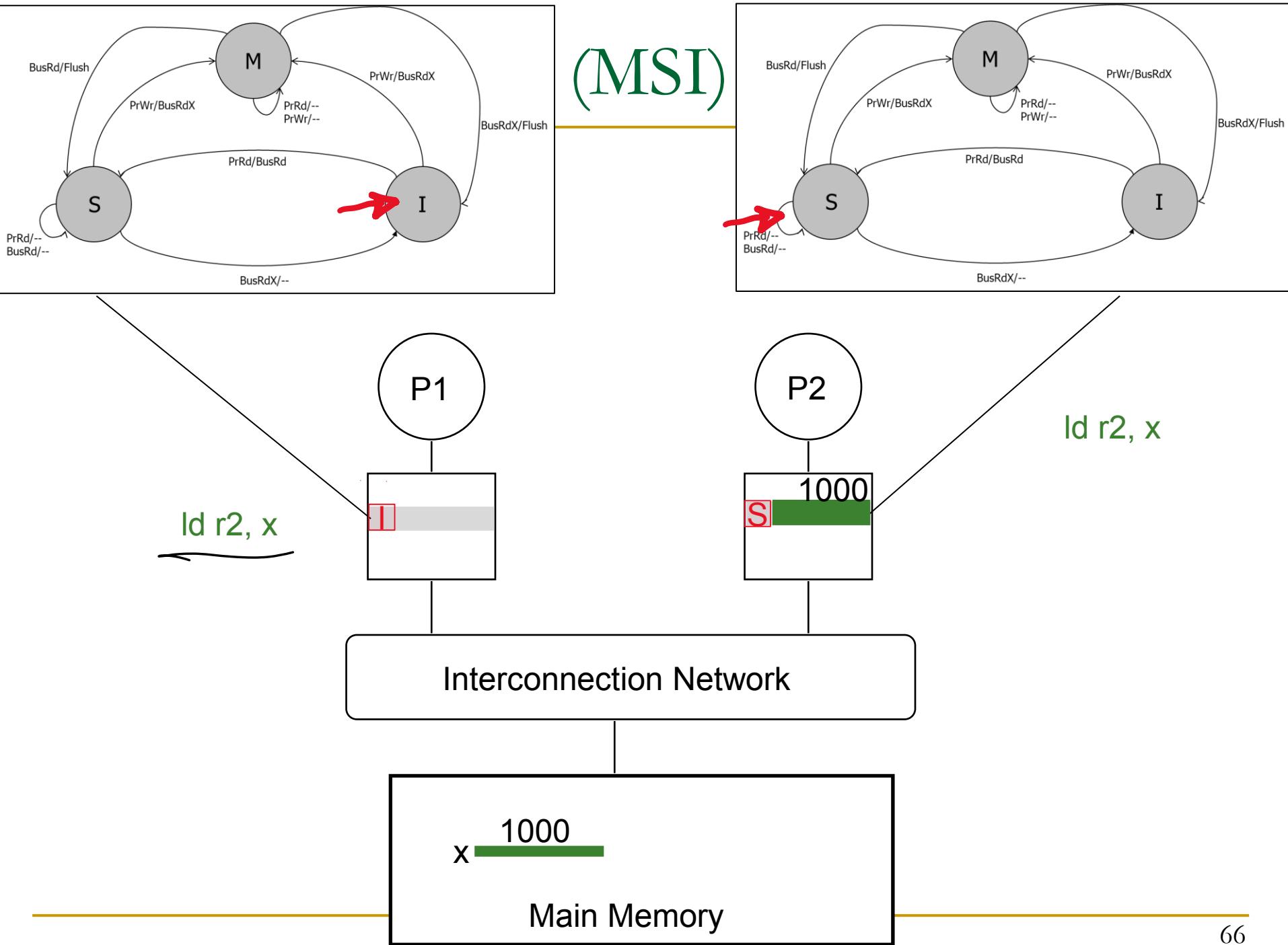
Questa bus read verrà comunicata alla main memory e agli altri cache controller.

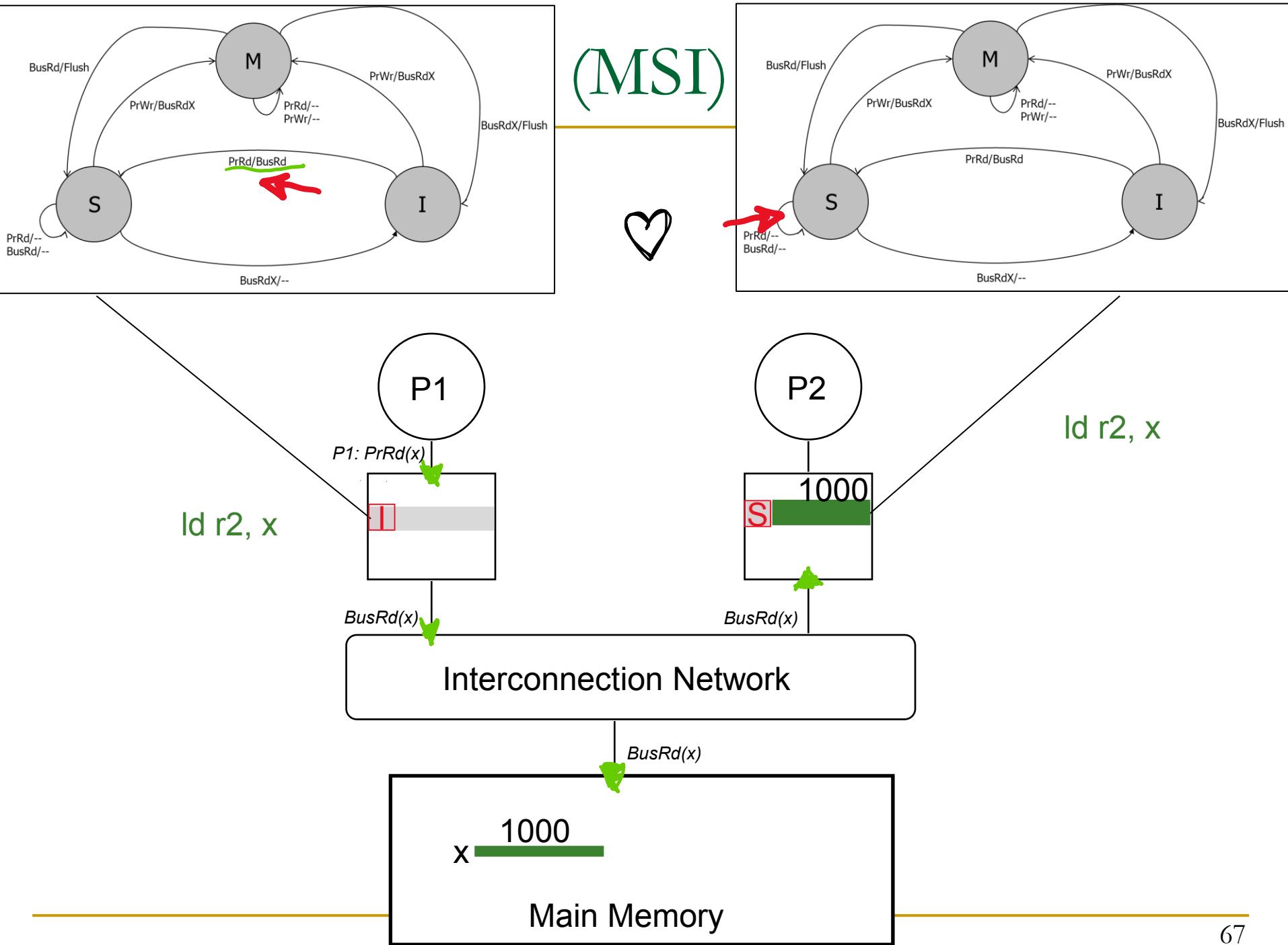
Prima si aspetta gli altri cache agent e poi nel caso si legge dalla main.

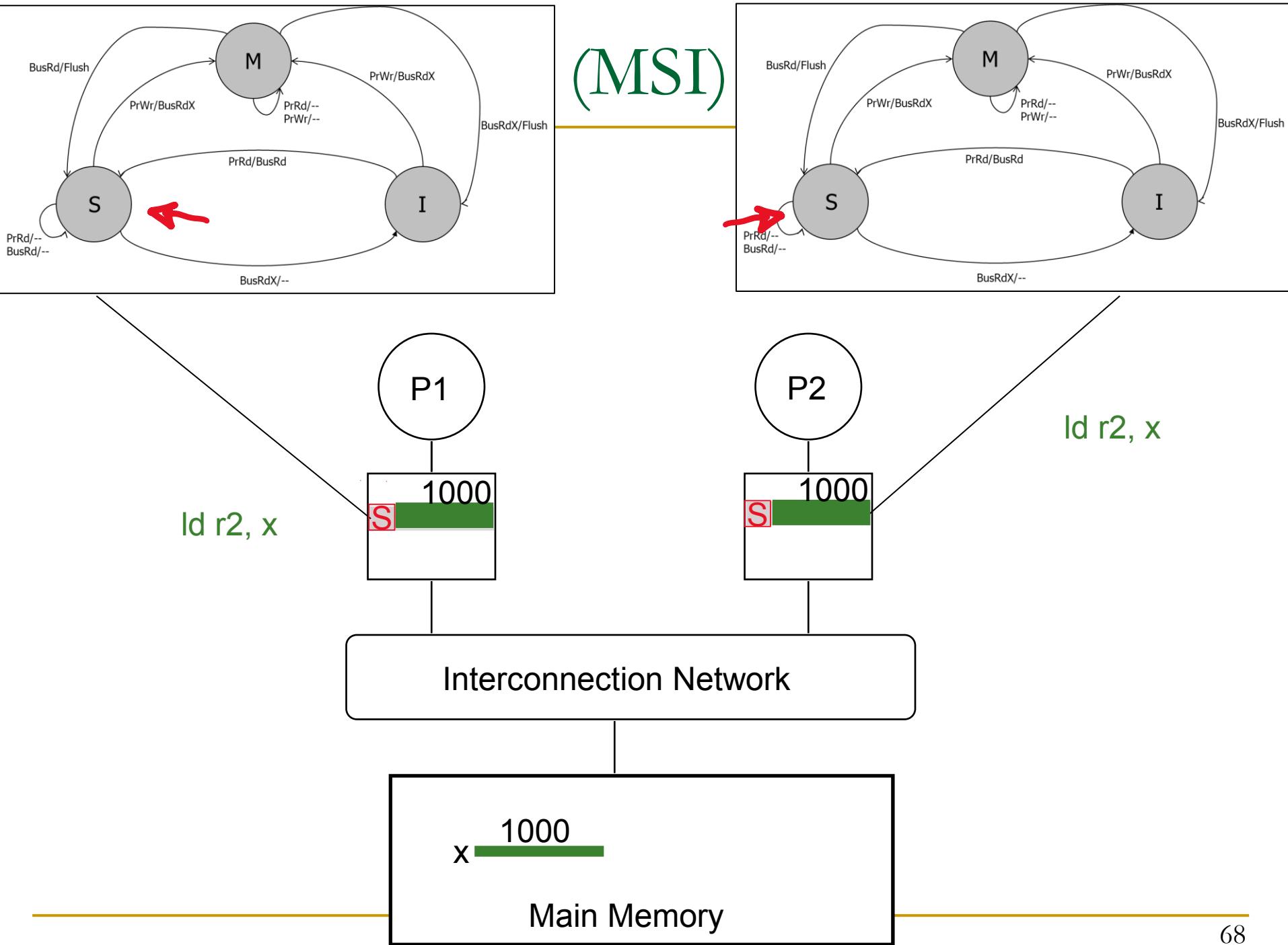


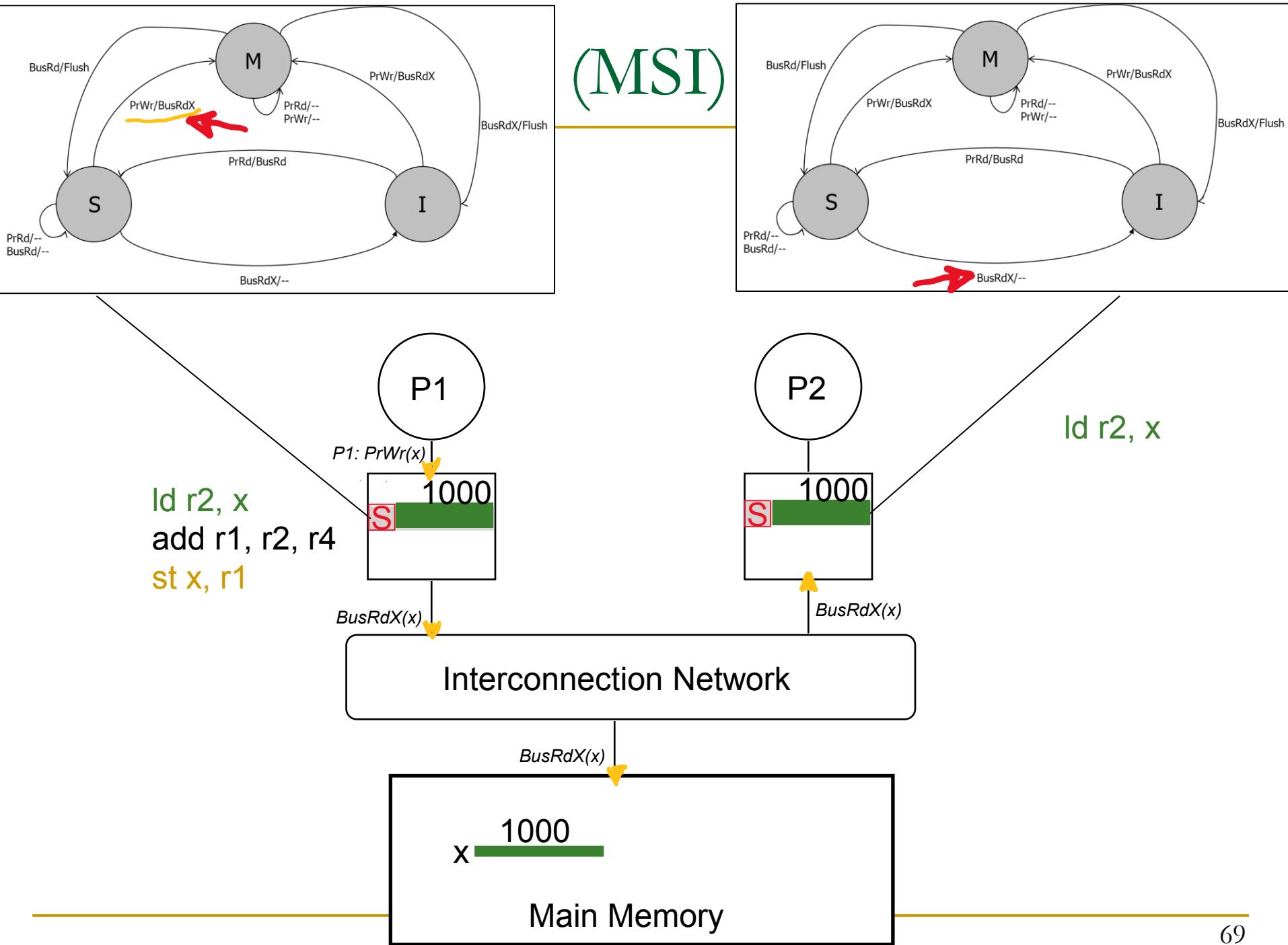


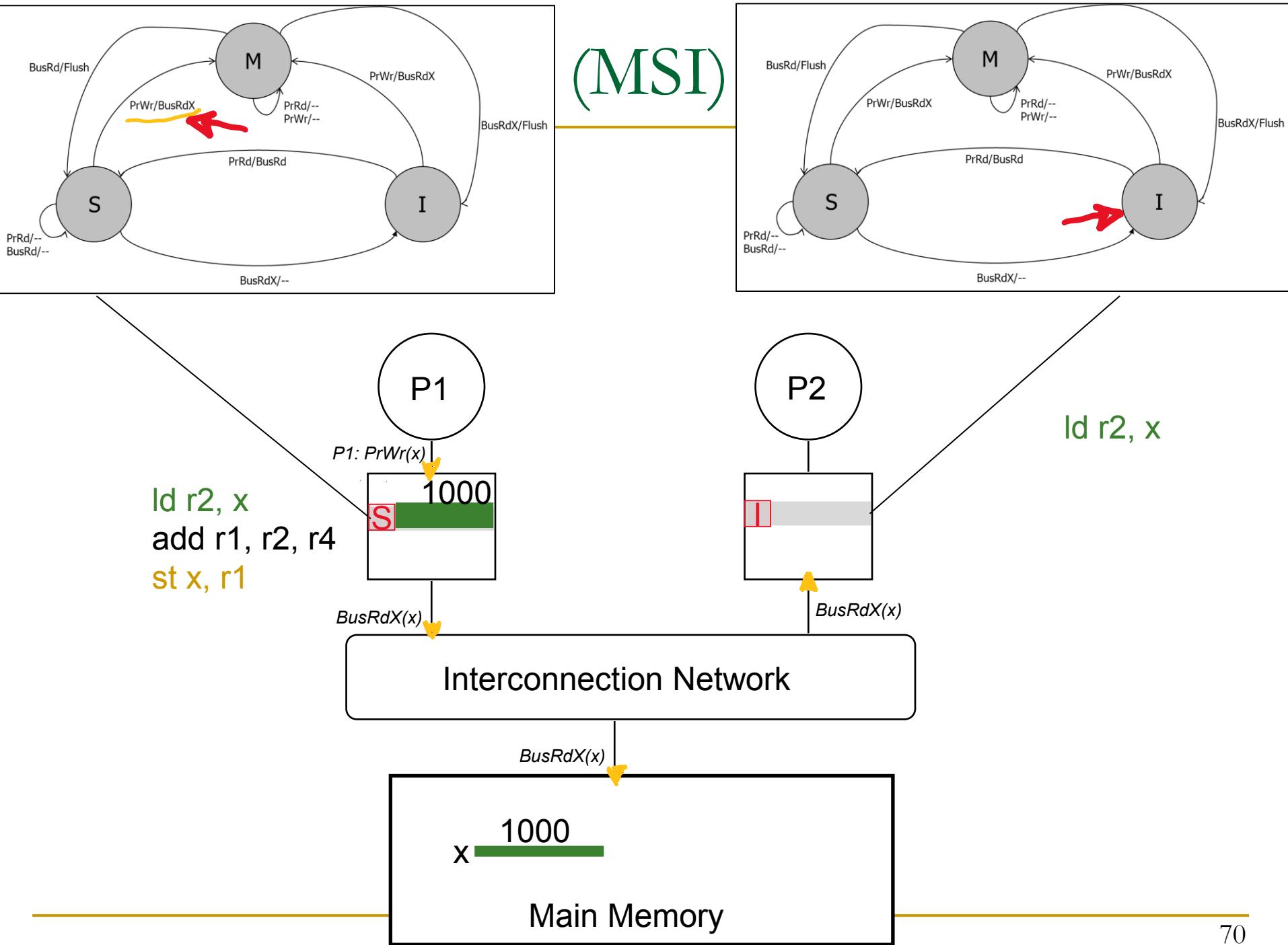


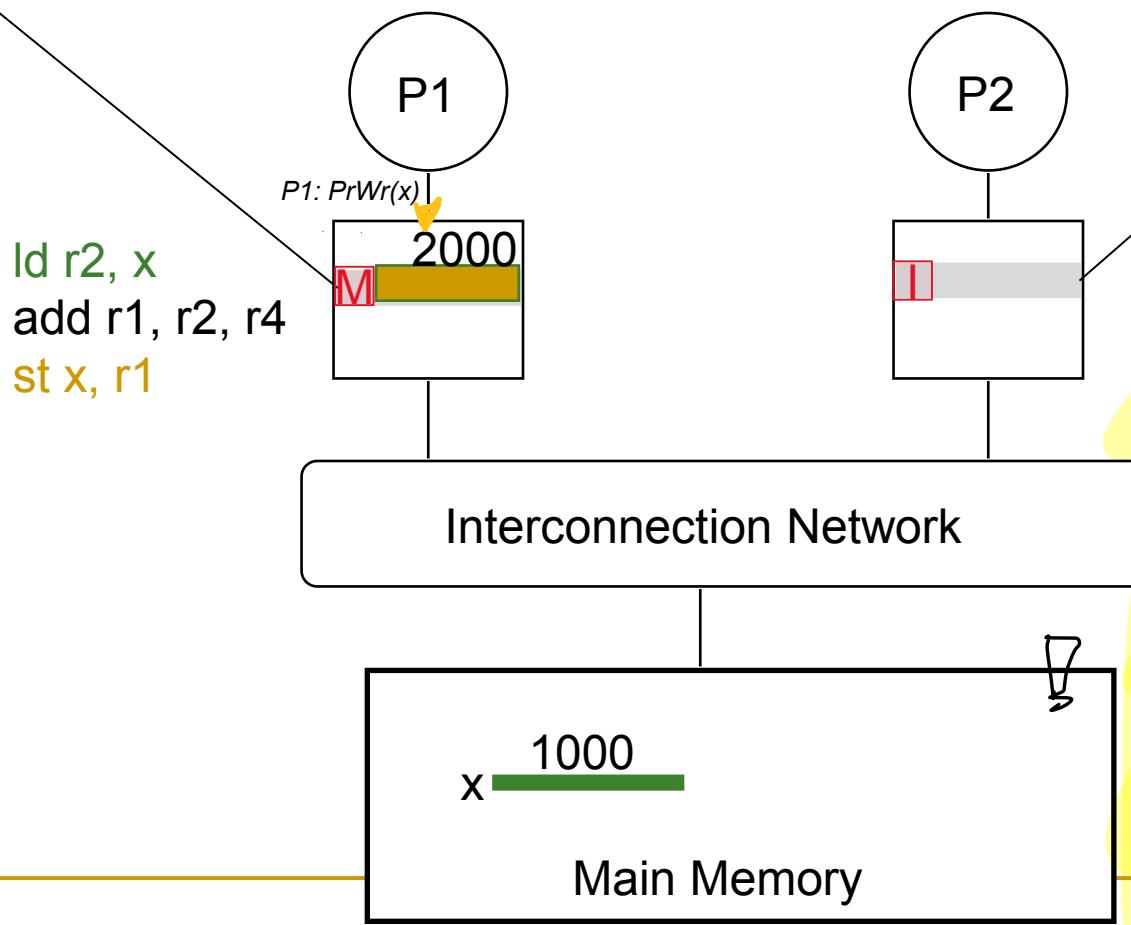
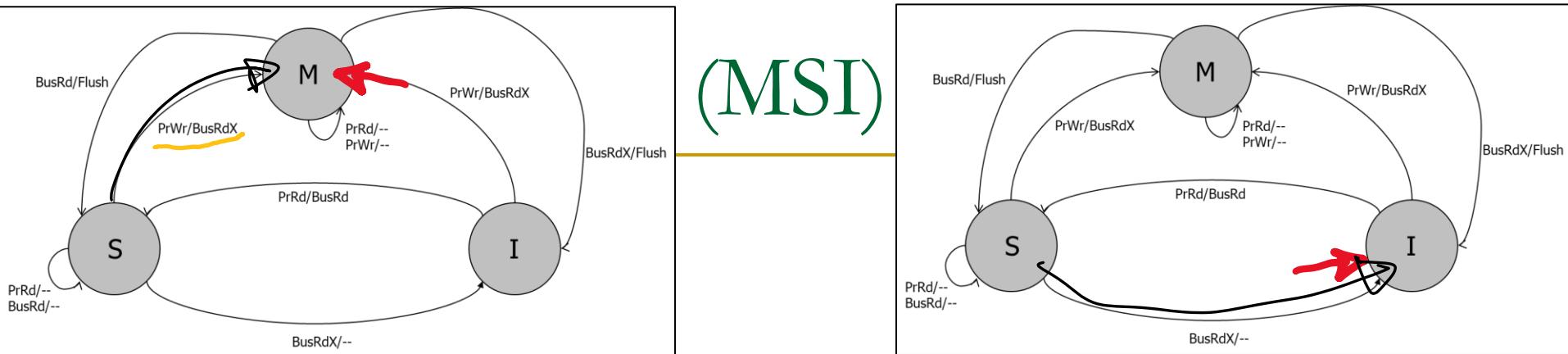








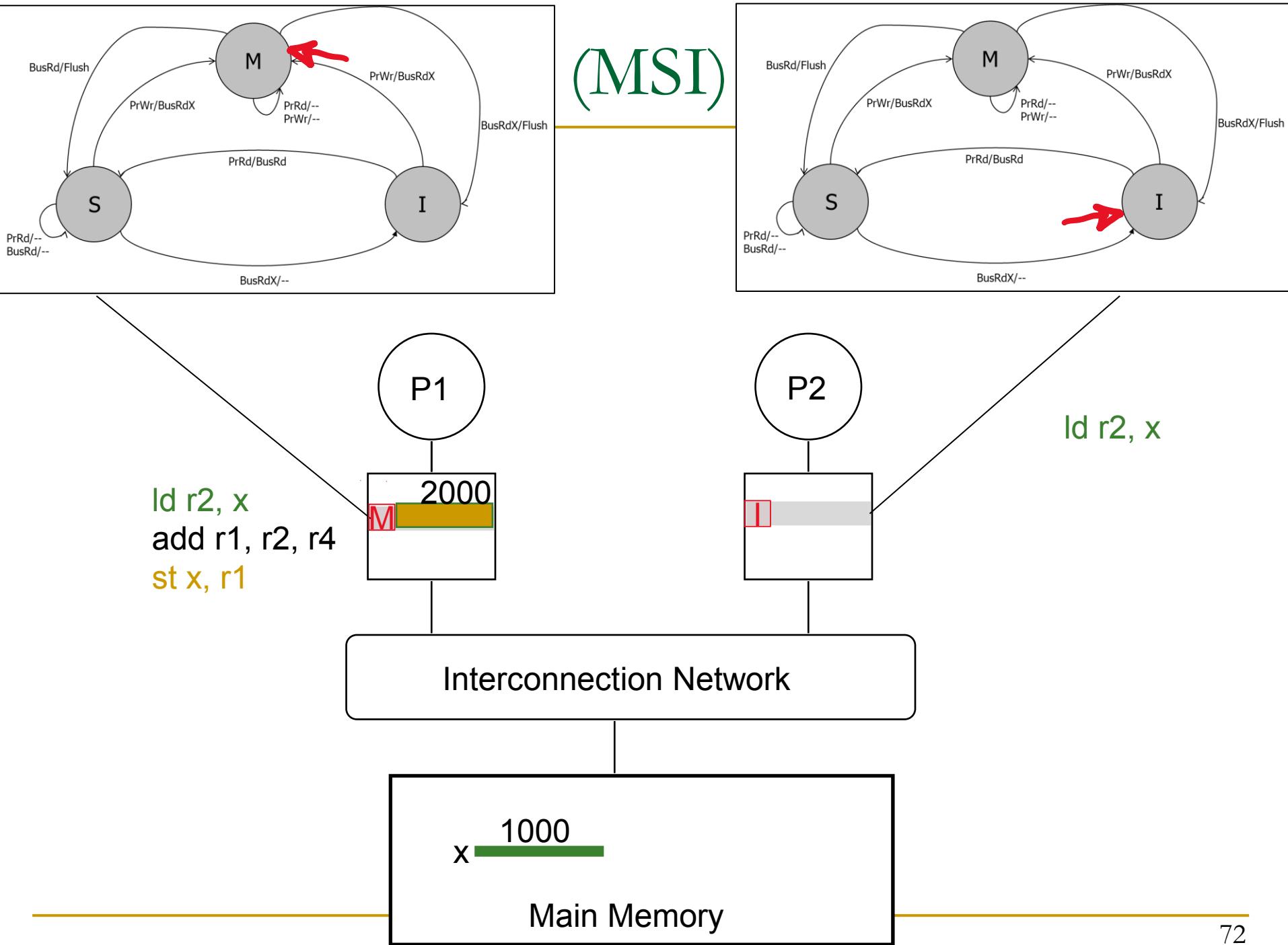


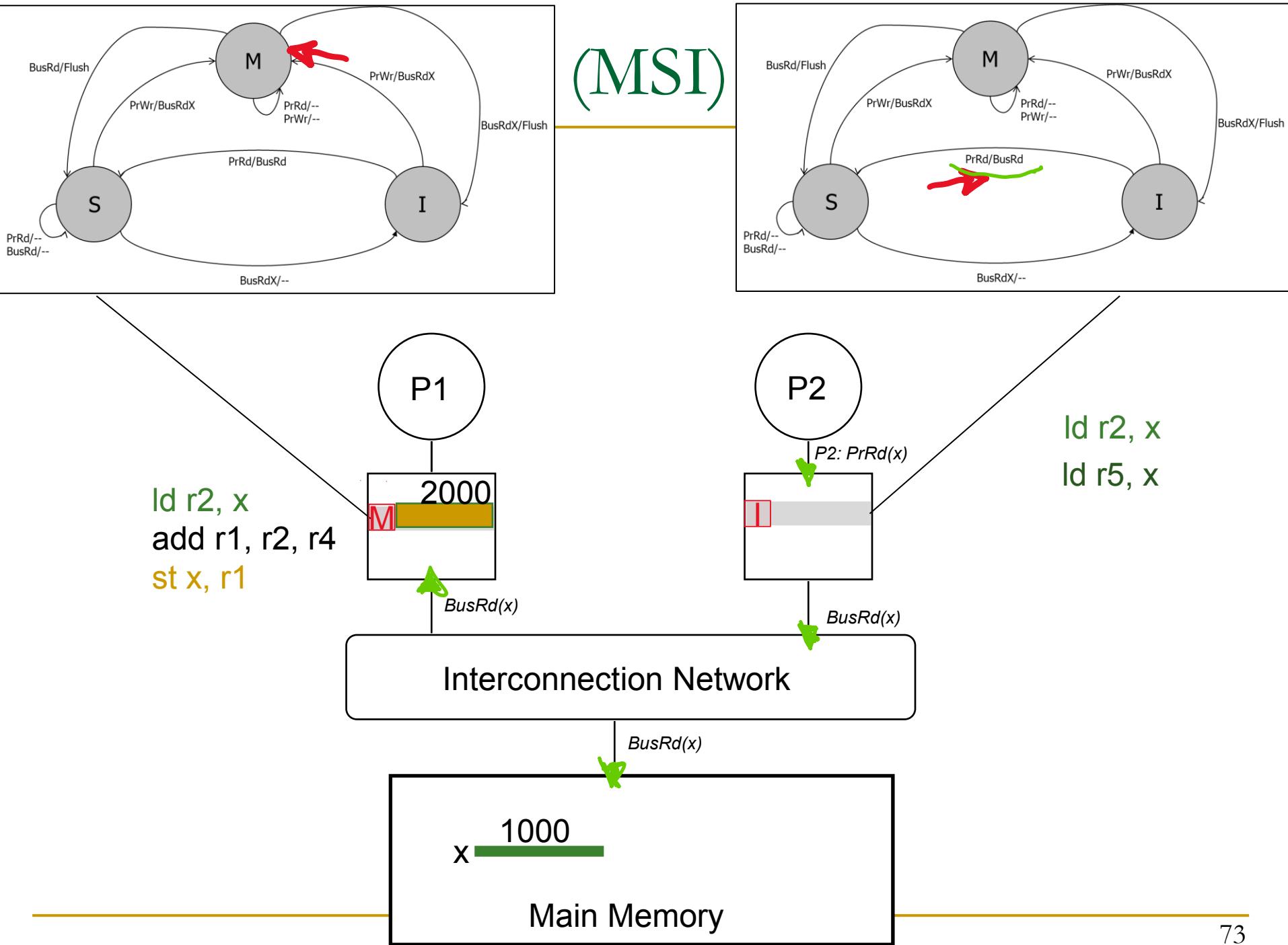


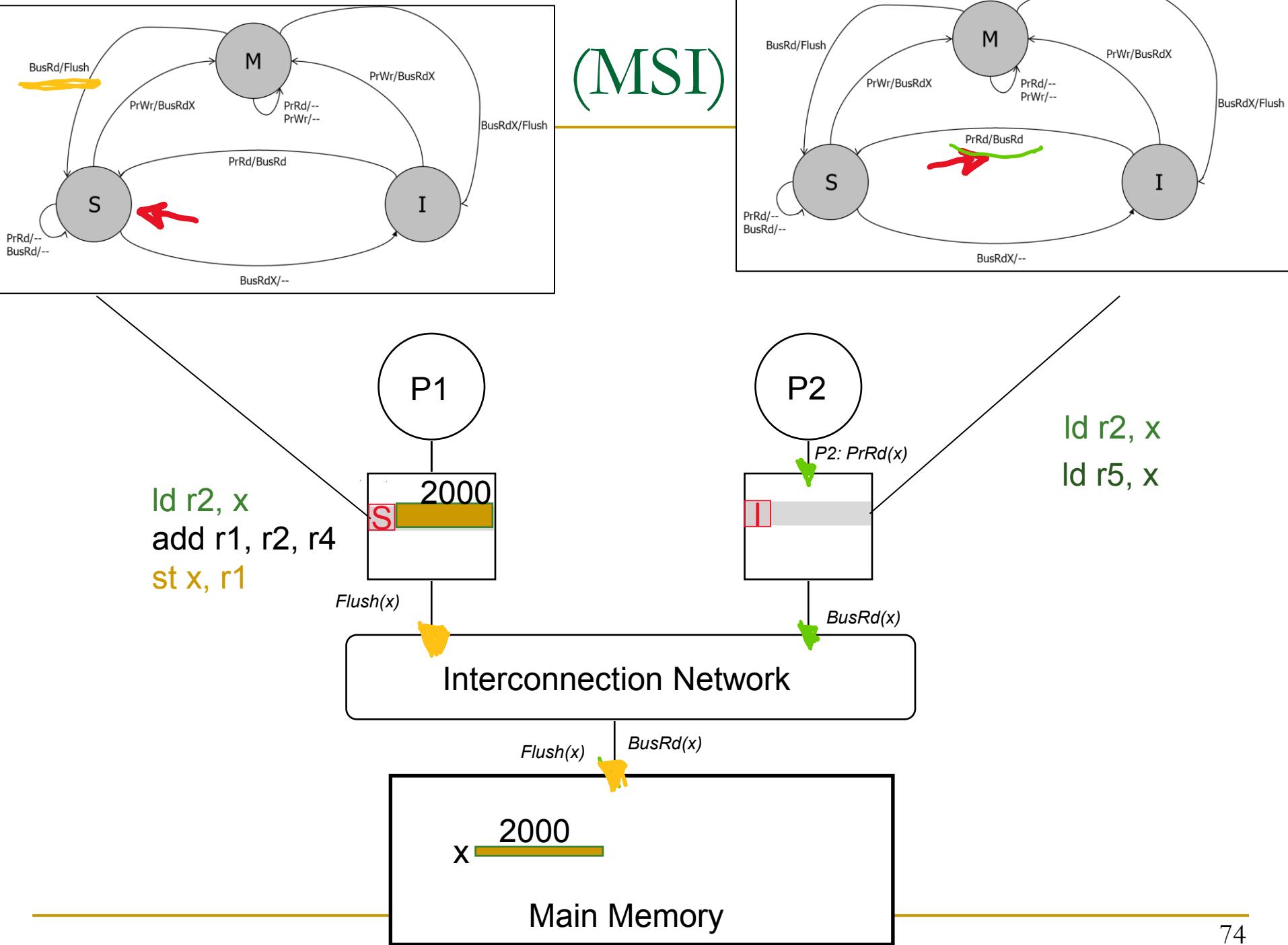
Id r2, x

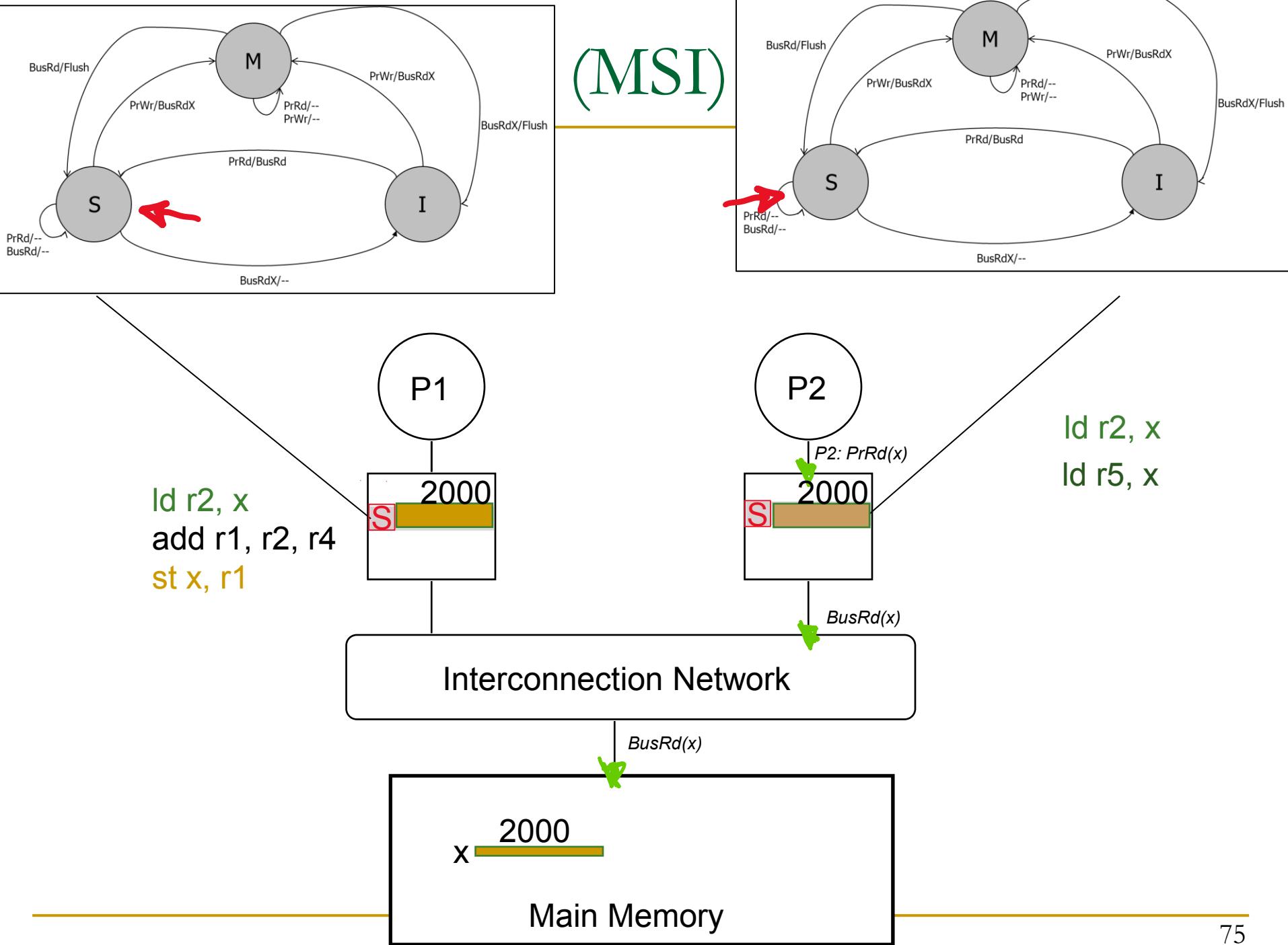
Se si fa una hit di una load in una cache in stato S costa poco in termini di latenza, se invece la load fa una Miss possono succedere due cose: o non esiste nessuna copia di quel dato modificato, e quindi vado a leggere direttamente dalla main memory, e pago solo la latenza di accesso alla main memory una volta.

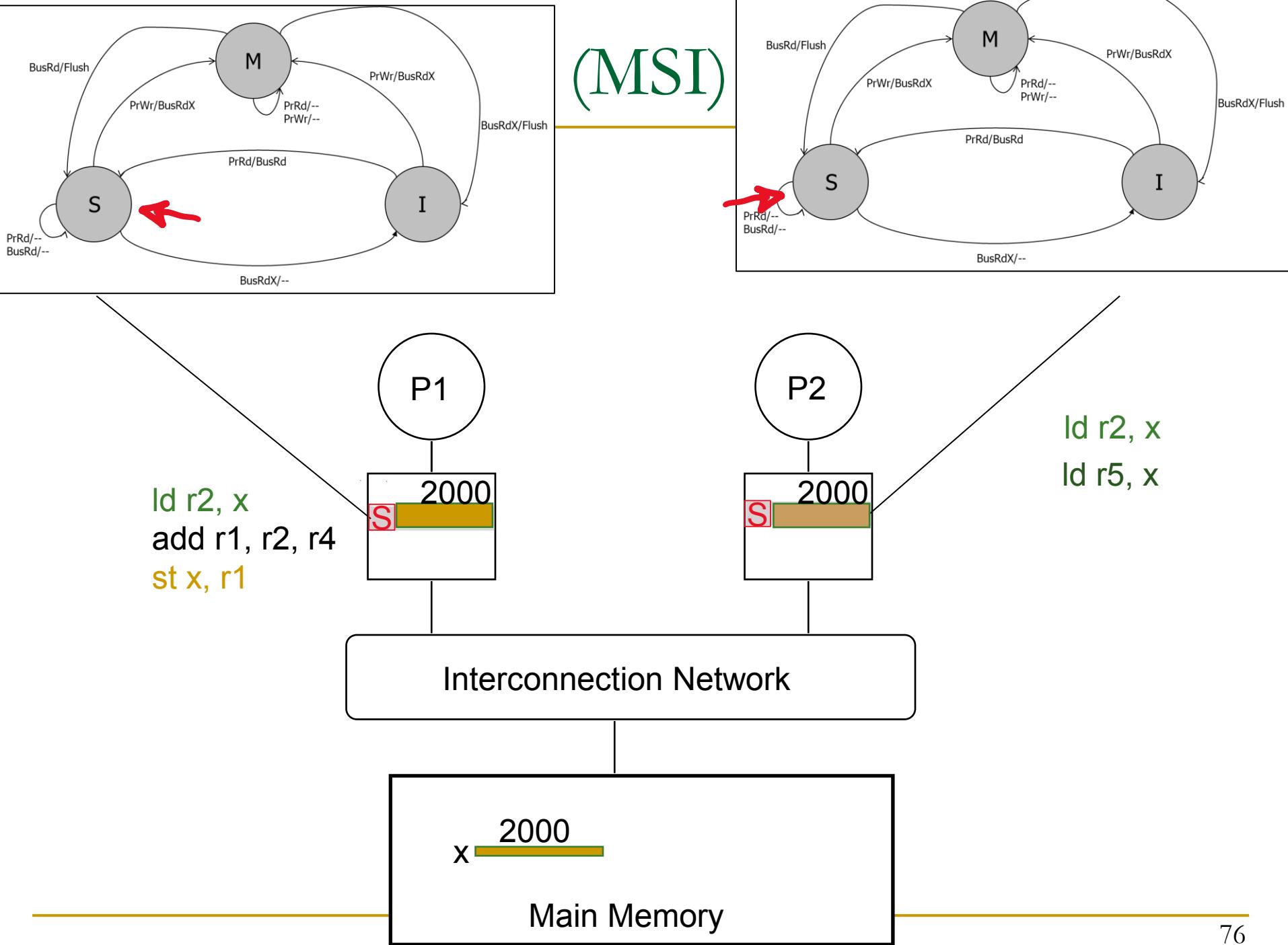
Se invece quel dato è presente in un'altra cache ed è in stato modified, bisogna aspettare che quel processore scriva in memoria, quindi si aspetta e poi dobbiamo andare a leggere dalla main memory, altra latenza.











Se si fa una hit di una load in una cache in stato S costa poco in termini di latenza, se invece la load fa una Miss possono succedere due cose: o non esiste nessuna copia di quel dato modificato, e quindi vado a leggere direttamente dalla main memory, e pago solo la latenza di accesso alla main memory una volta.

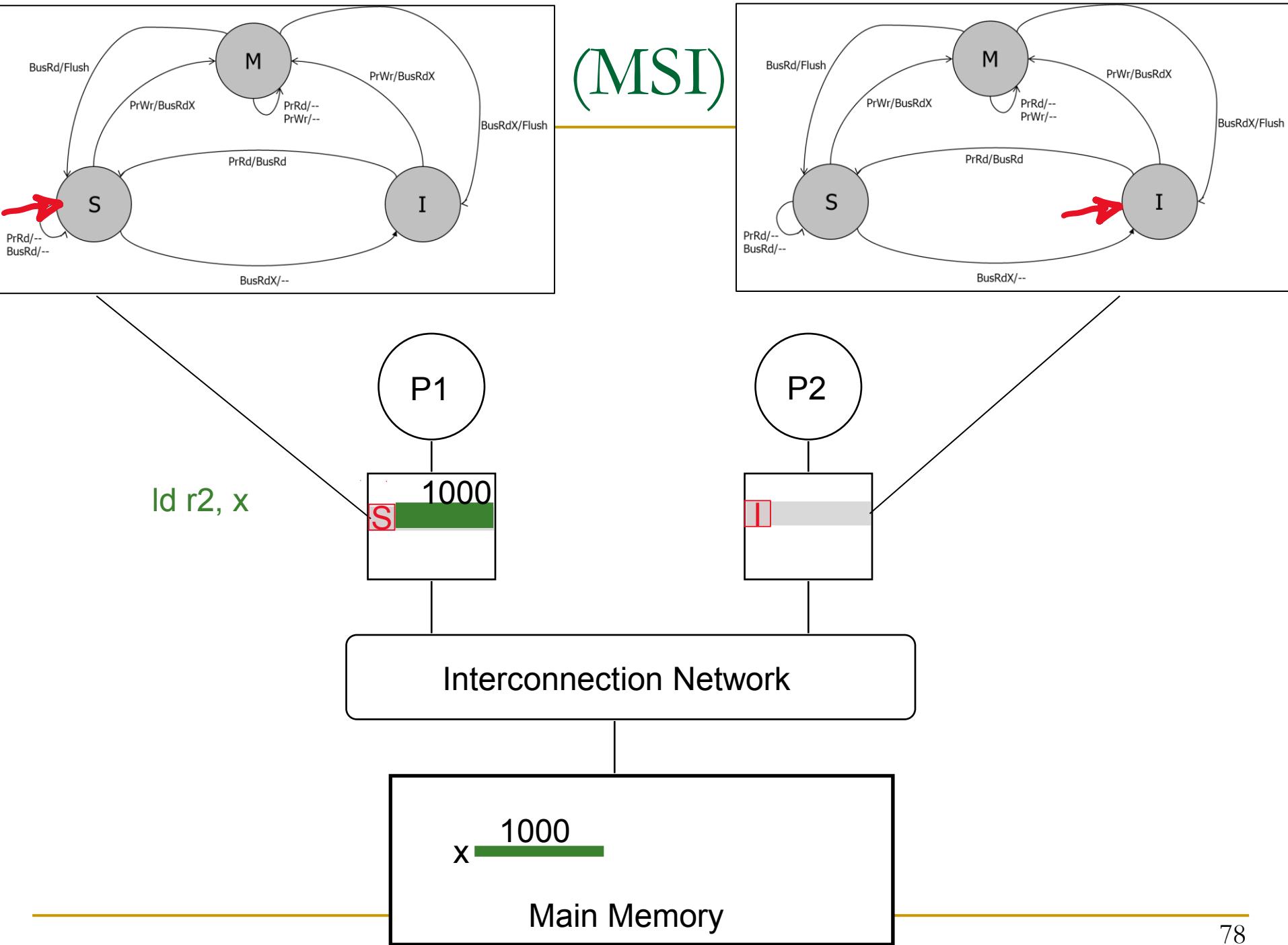
Se invece quel dato è presente in un'altra cache ed è in stato modified, bisogna aspettare che quel processore scriva in memoria, quindi si aspetta e poi dobbiamo andare a leggere dalla main memory, altra latenza.

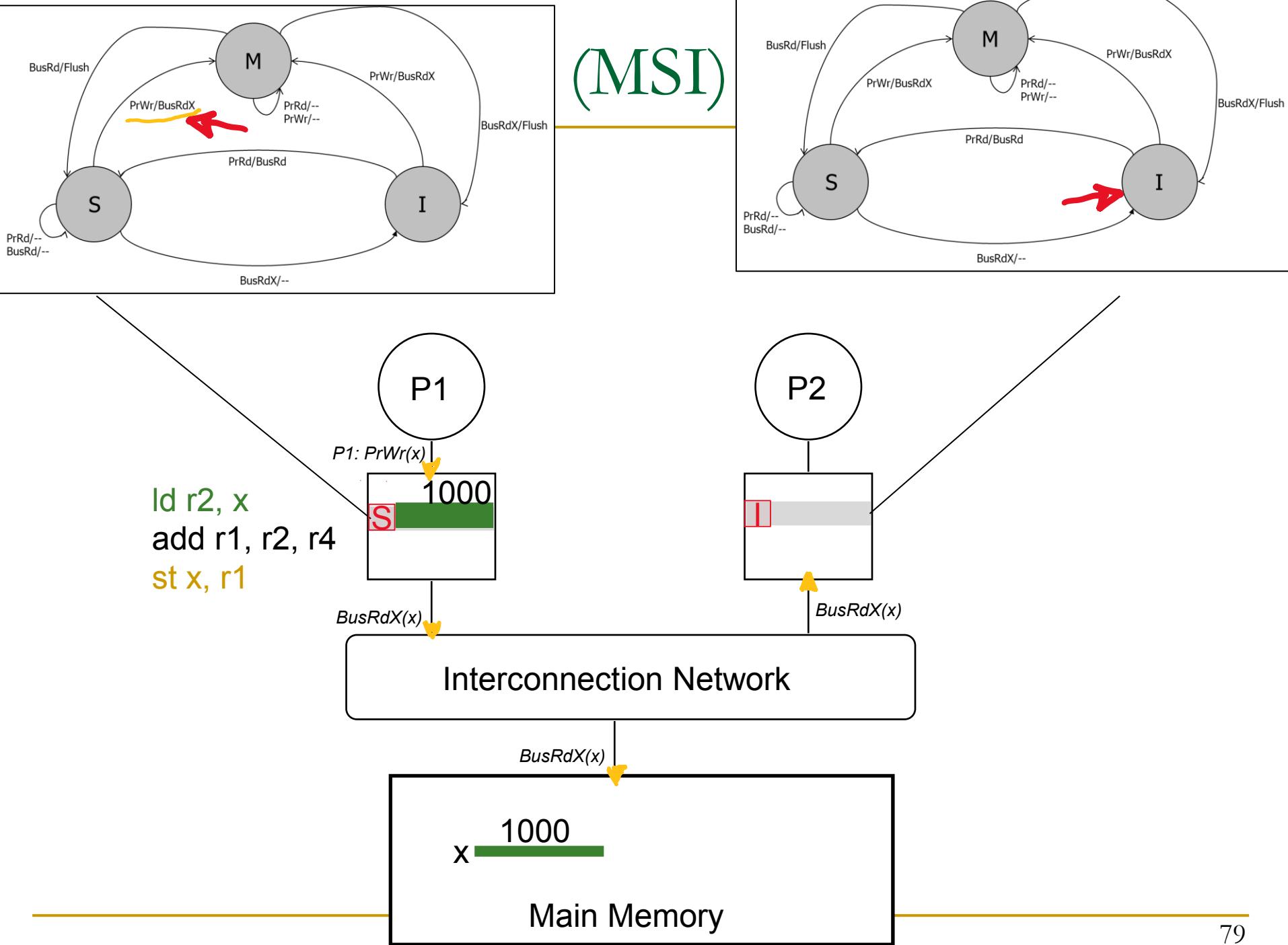
Quindi nella logica di Tommasoulo, se ho una architettura bloccante, uno stallo a seguito di una operazione di load può bloccare la pipeline per diversi cicli, anche centinaia.

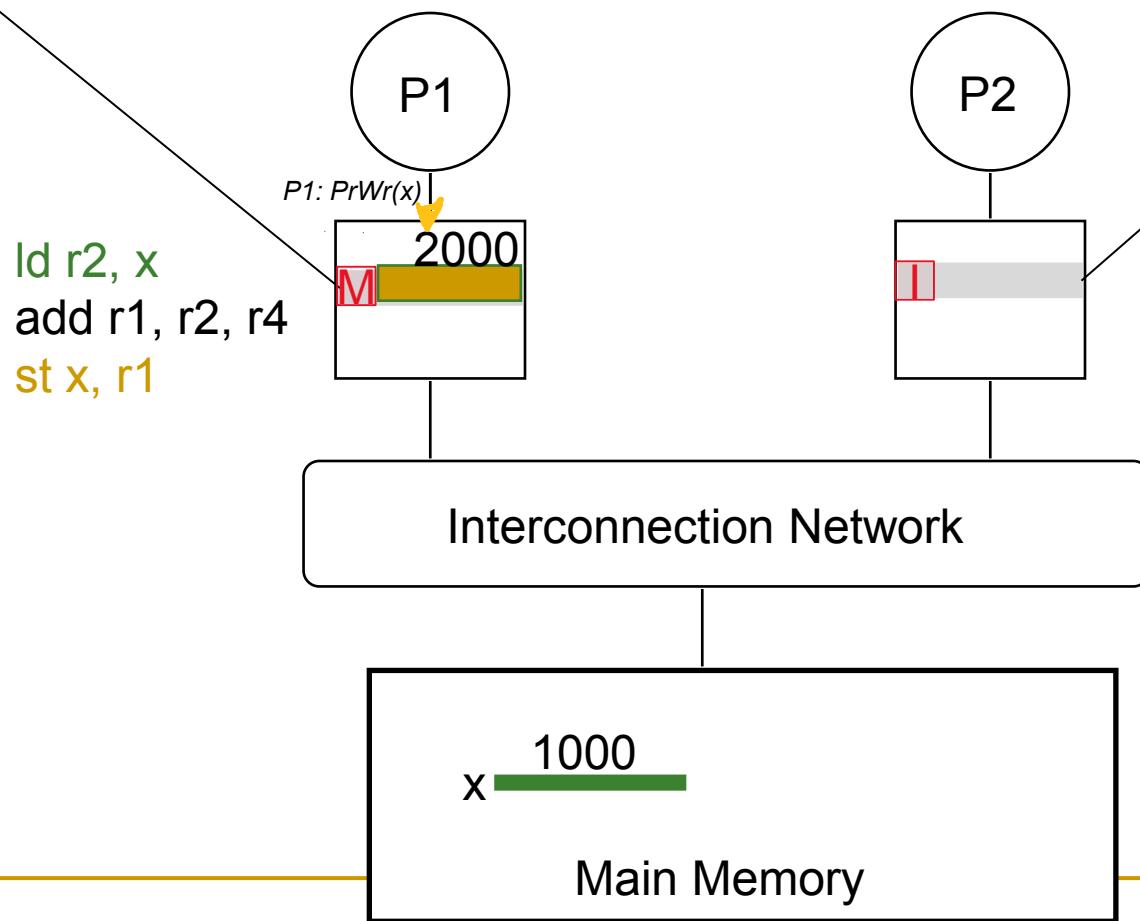
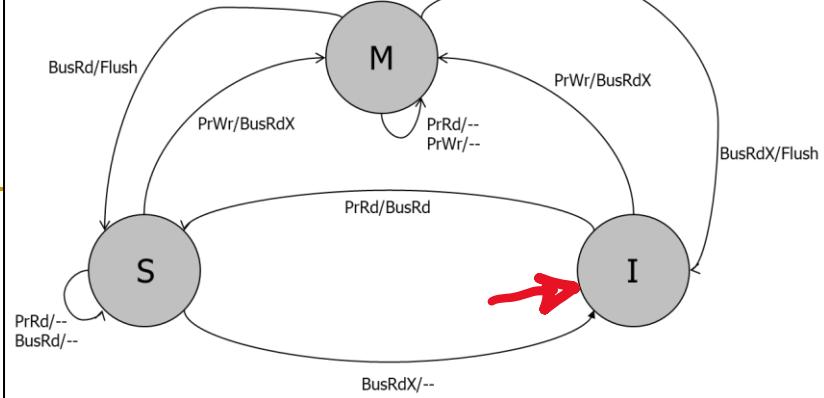
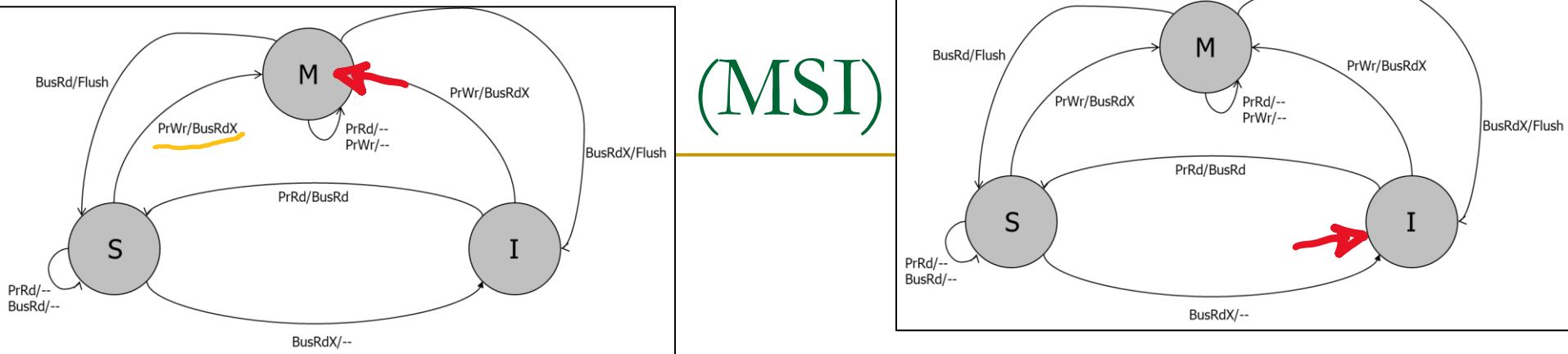
The Problem with MSI

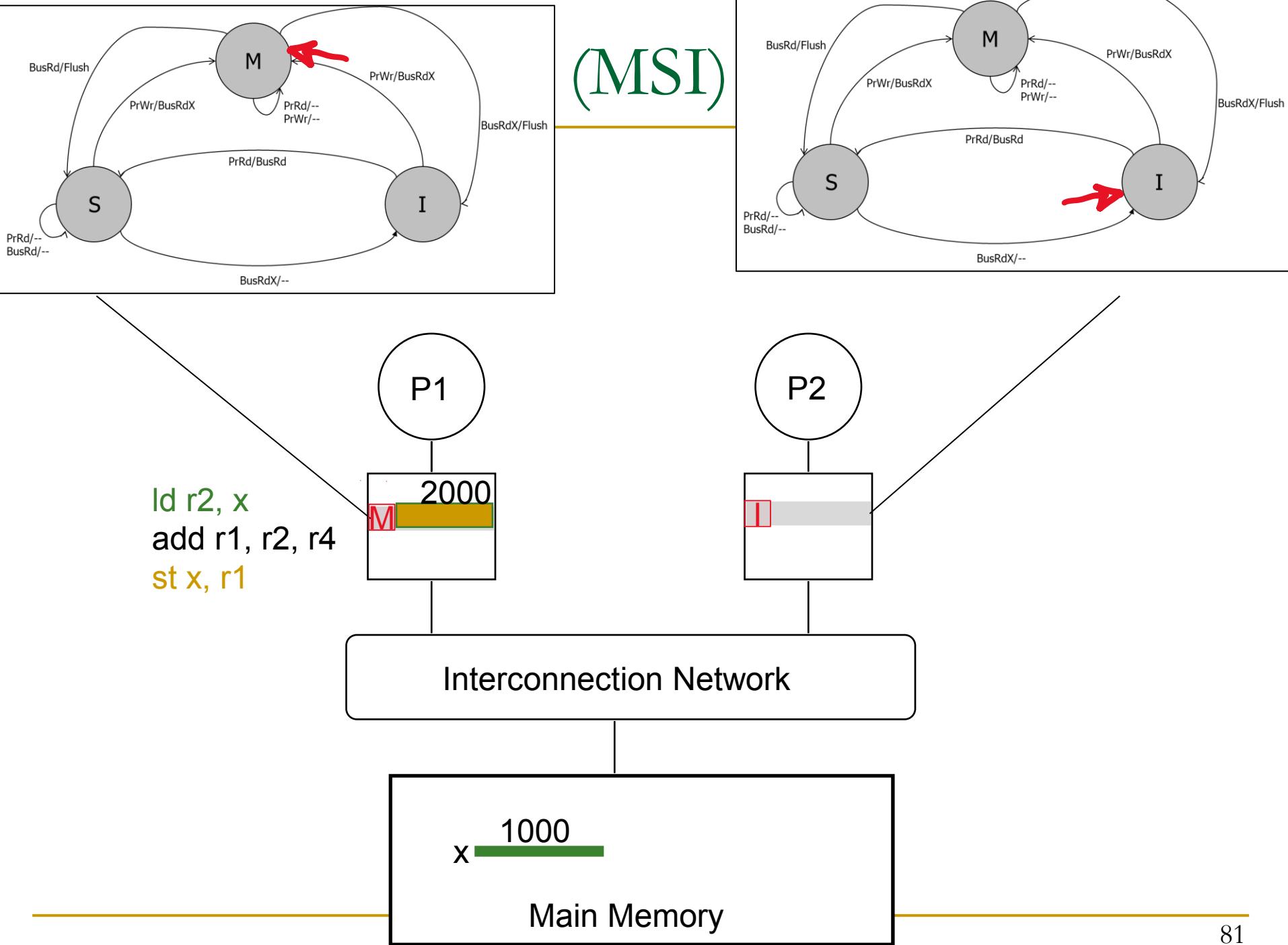
se voglio modificare un blocco di memoria deve comunicare agli altri che ha un read exclusive su quel blocco e invalidare eventuali copie locali.
Questo deve farlo e perciò deve fare almeno una copia locale in altre cache.

- A block is in no cache to begin with
- Problem: On a read, the block immediately goes to “Shared” state although it may be the only copy to be cached (i.e., no other processor will cache it)
- Why is this a problem?
 - Suppose the cache that reads the block wants to write to it at some point
 - It needs to broadcast “invalidate” even though it has the only cached copy!
 - *If the cache knew it had the only cached copy in the system, it could have written to the block without notifying any other cache → saves unnecessary broadcasts of invalidations*









The Solution: MESI

E: NUOVO STATO CHE DICE CHE E' L'UNICA COPIA LOCALE PUUTA DI UN DATO.

S: IN STATO SHARED QUANDO HOO UNA COPIA LOCALE CORRETTA MA NON SONO L'UNICO AD AVERLA.

- Idea: Add another state indicating **that this is the only cached copy and it is clean.**
 - *Exclusive state*
- Block is placed into the *exclusive* state if, during *BusRd*, no other cache had it
 - Wired-OR “shared” signal on bus can determine this:
snooping caches assert the signal if they also have a copy
- Silent transition *Exclusive*→*Modified* is possible on write!
- MESI is also called the *Illinois protocol*
 - Papamarcos and Patel, “**A low-overhead coherence solution for multiprocessors with private cache memories**,” ISCA 1984.

PRINCIPIO MESI

Faccio una load, ho una Miss, vuol dire che non ho una copia locale. Emetto quindi sul bus di snoop la richiesta di fare una bus read, quindi leggo una linea di cache associata ad un indirizzo.

Prima di capire in che stato andare, a partire da un invalid, dopo aver letto, bisogna capire se qualcun'altra cache ha una copia locale associata a quell'indirizzo. Se non esiste una copia locale andrò in E, se esiste una copia locale andrò in S.

Quindi mi aspetto un segnale di risposta nel bus di snoop che mi dica se esistono già copie locali per quell'indirizzo. Questo si chiama segnale di shared.

Quindi a seguito di una bus read leggo un segnale shared, che è un or di un segnale di shared che emetteranno tutti gli altri cache agent.

A seguito di questo segnale capisco se andare in S o in E.

La cosa buona è che si passa dallo stato esclusive a modified senza la necessità di comunicarlo a nessuno., perché tanto era esclusiva, nessun altro aveva quel dato quindi nessun altro deve essere informato.

Identicamente al msi, se mi trovo in S e il processore emette una store, prima di servirla dovrò emettere una richiesta sul bus di lettura esclusiva che invalida le altre copie e a quel punto posso modificare il dato e portarmi nello stato M. (M informa che ho L unica copia locale di quel dato e mi informa che è sporca, ovvero non corrisponde a quello che c'è nella main memory).

Se mi trovo in E, vuol dire che ho una copia locale che è coerente con la memoria, ma è l'unica copia, quindi se voglio modificarla, lo faccio portando la linea di cache in M, senza avvertire nessuno perché tanto ho la certezza che quella copia era L unica. Non esistono altre copie in altre cache che devo andare a invalidare.

E: ho copia di dato della main memory in cache ed è l'unica ed è pulita. Se osservo una richiesta di bus read, vuol dire che c'è qualcun altro che vuole fare una copia locale del dato, per cui devo portarmi in stato shared, perché esisterà un'altra copia locale di quel dato.

Posso anche passare direttamente la copia all'altra cache senza farla andare in memoria eventualmente.

Se sono in E e osservo una richiesta di bus read esclusiva, dovrò invalidare la copia locale perché c'è qualcun altro che vuole modificare quel dato quindi dovrò invalidare la mia copia locale e portare la mia Linea di cache allo Stato I.

La linea di cache ha due proprietà: essere sporca o pulita rispetto alla main memory e l'altra è essere l'unica copia o meno.

Nota: se ho l'unica copia del dato, ho un privilegio perché posso modificarlo senza dire a nessuno niente.

Se ho multiple copie del dato posso solo leggerlo, se esiste un unico copia del dato posso sia leggerlo che modificarlo.

Se riesco a garantire questo ho coerenza nel sistema multi core.

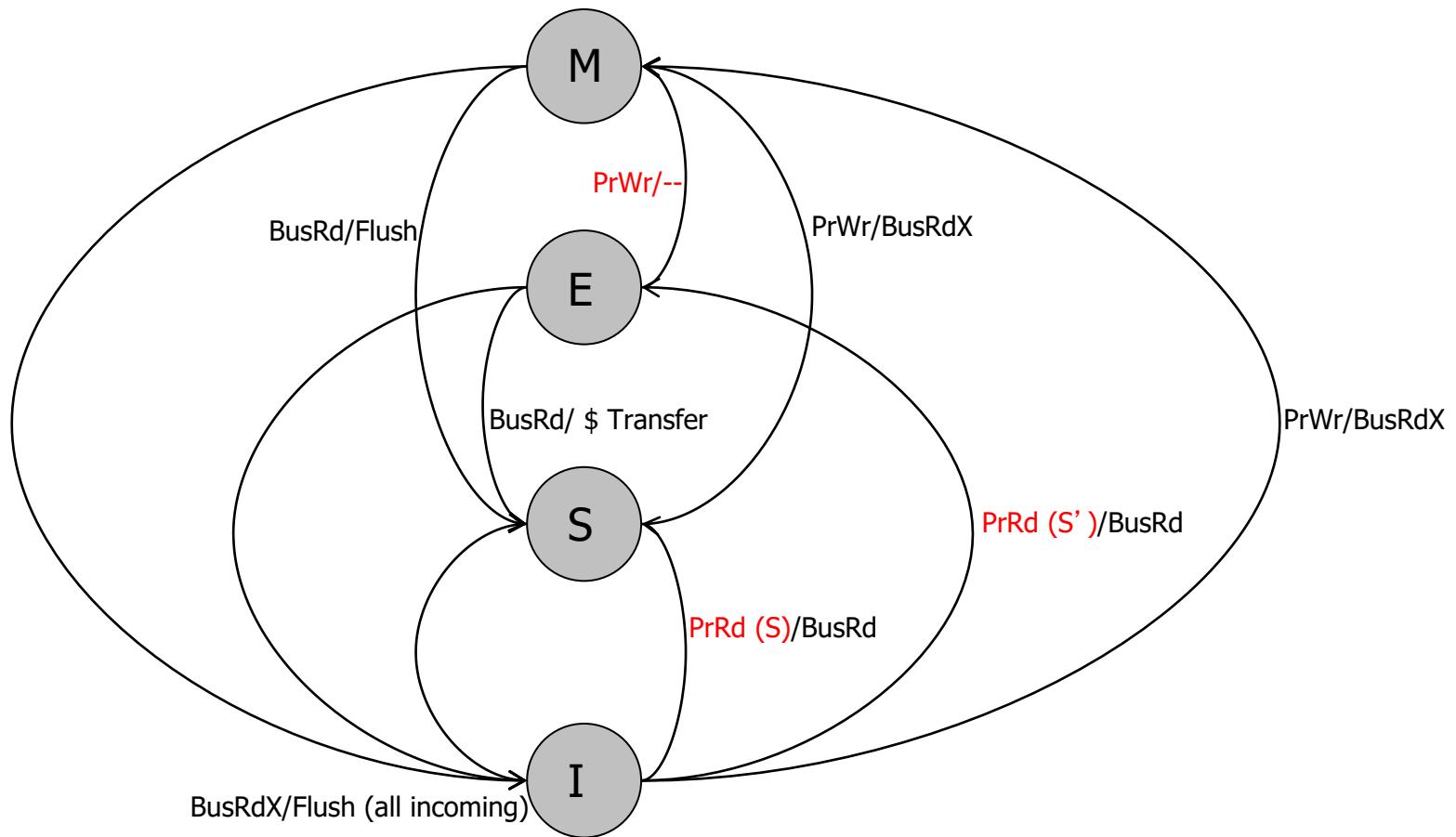
Se mi trovo in M, e osservo una richiesta di bus read (quindi qualcun altro vuole leggere quel blocco di memoria) o una bus read esclusiva (qualcun altro vuole modificare quel dato), devo fare flush verso la memoria , ovvero fare un ciclo di Write back , dove copio in memoria la linea dirty.

A questo punto posso andare in I o S, questo dipende dalla richiesta del bus di snoop.

Se quando ero in M mi è stato richiesto una lettura da un altro cache agent allora vado nello stato shared ma devo comunque prima copiare il dato verso la main memory, se invece osservo una richiesta di read esclusiva , devo aggiornare la memoria rispetto al valore che ho precedentemente modificato, quindi rendere la riga clean e poi la devo invalidare perché qualcun altro vuole avere l unica copia locale (ed essere in E).

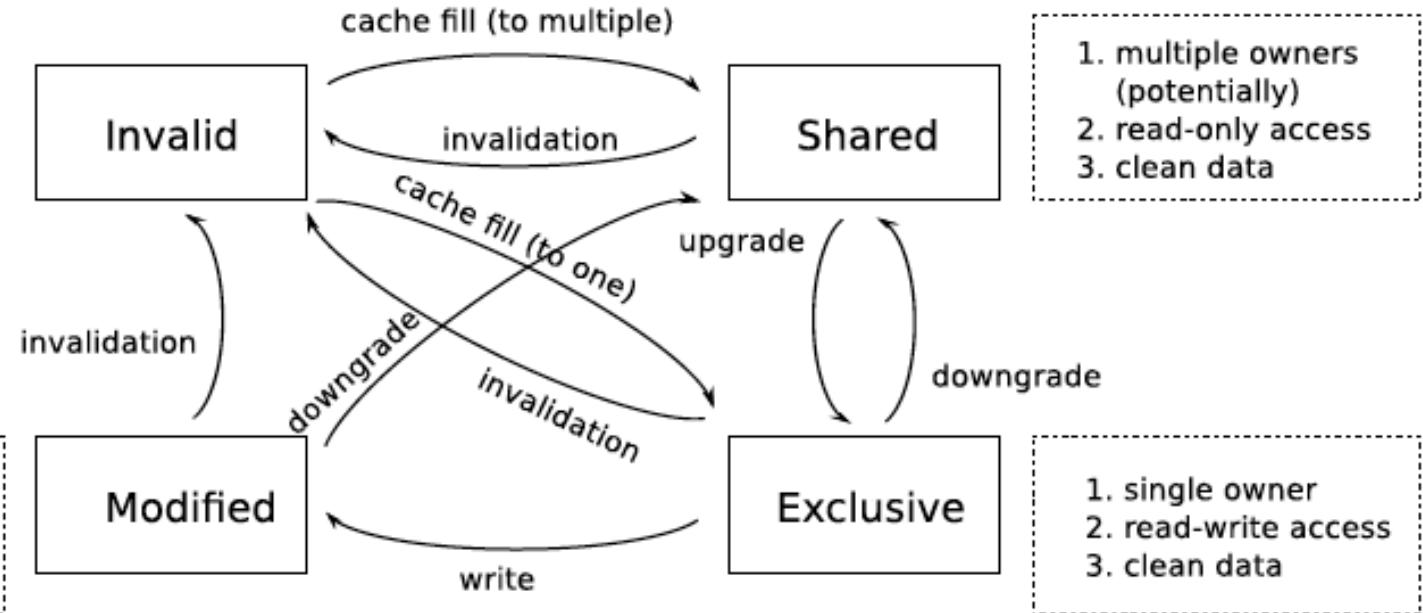
Per codificare l'informazione di in quale dei 4 stati si trova la riga di cache sono necessari 2 bit.

MESI State Machine



[Culler/Singh96]

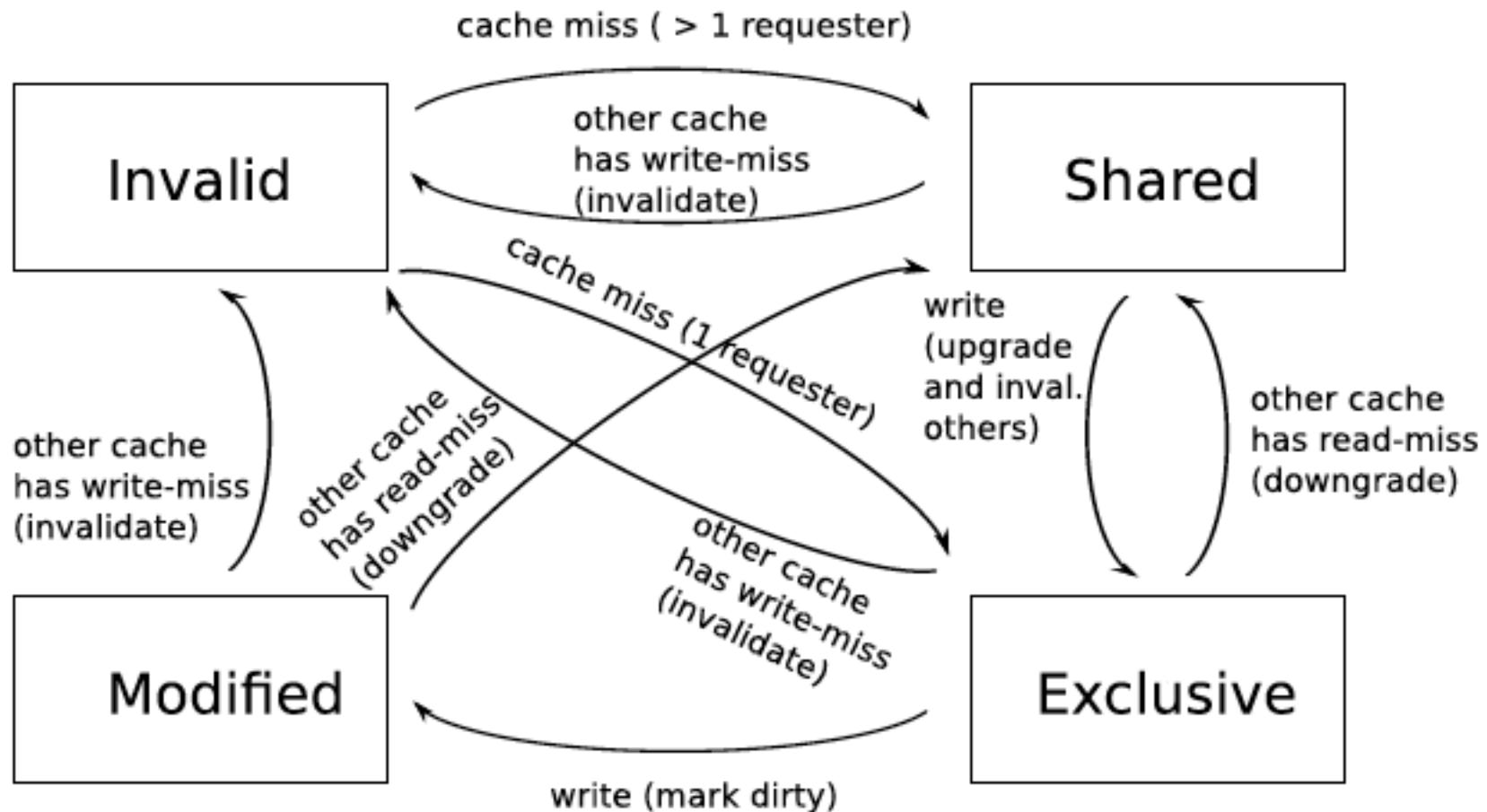
MESI State Machine



A transition from a single-owner state (Exclusive or Modified) to Shared is called a downgrade, because the transition takes away the owner's right to modify the data

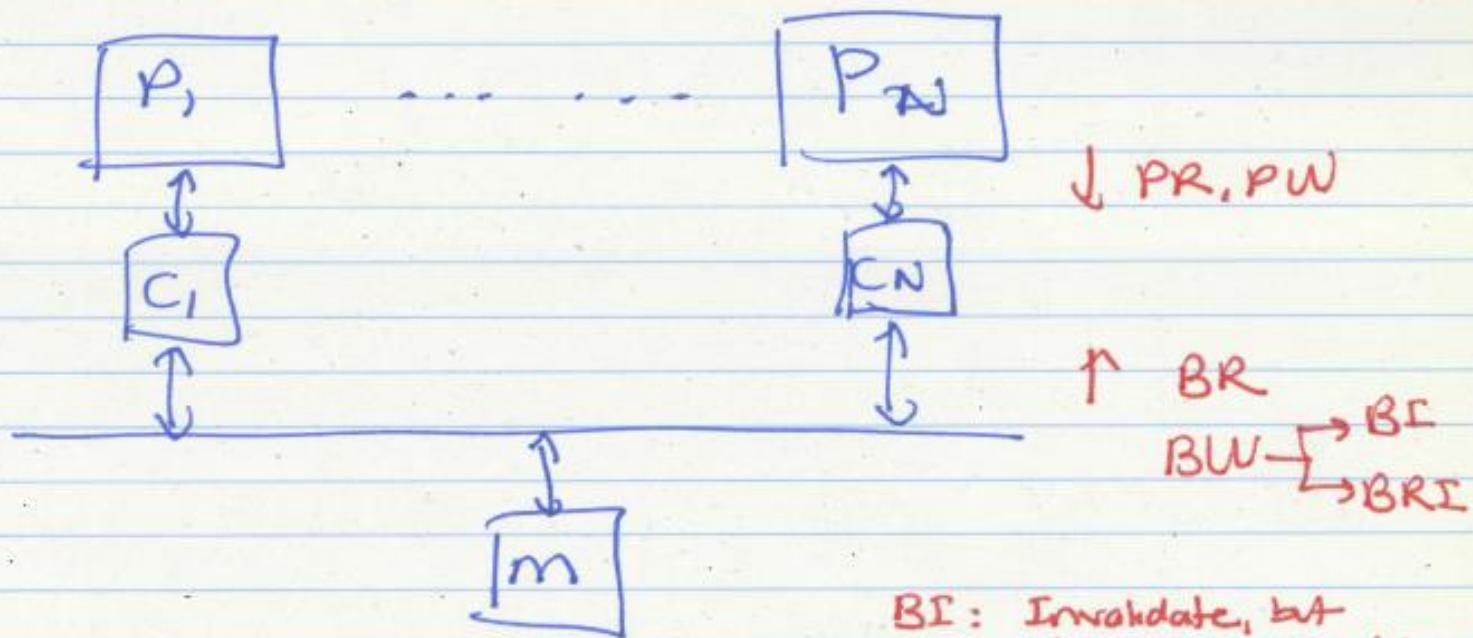
A transition from Shared to a single-owner state (Exclusive or Modified) is called an upgrade, because the transition grants the ability to the owner (the cache which contains the respective block) to write to the block.

MESI State Machine



Papamarcos & Patel, ISCA 1984

Illinois Protocol



BI: Invalidate, but
already have the data
(do not supply it)

BRI: Invalidate, but
also need the data
(supply it)

4 States

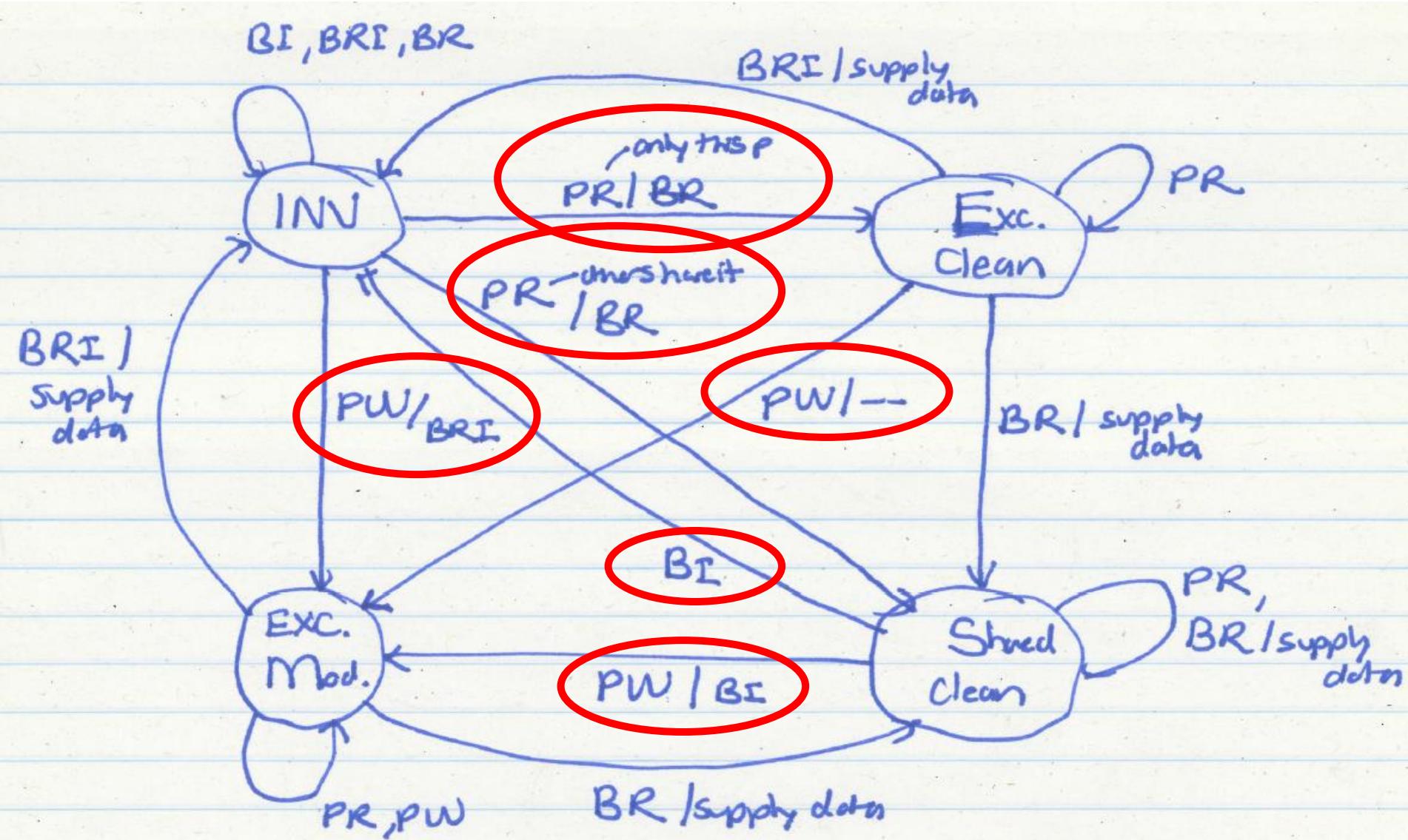
M: Modified (Exclusive copy, modified)

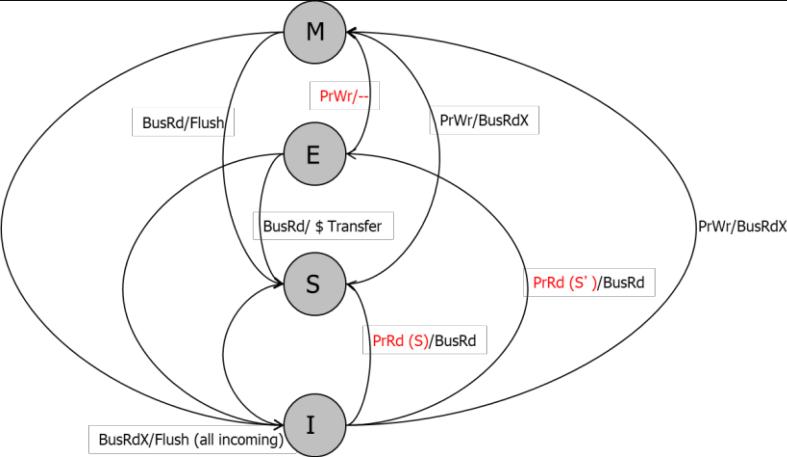
E: Exclusive (" ", clean)

S: Shared (Shared copy, clean)

I: Invalid

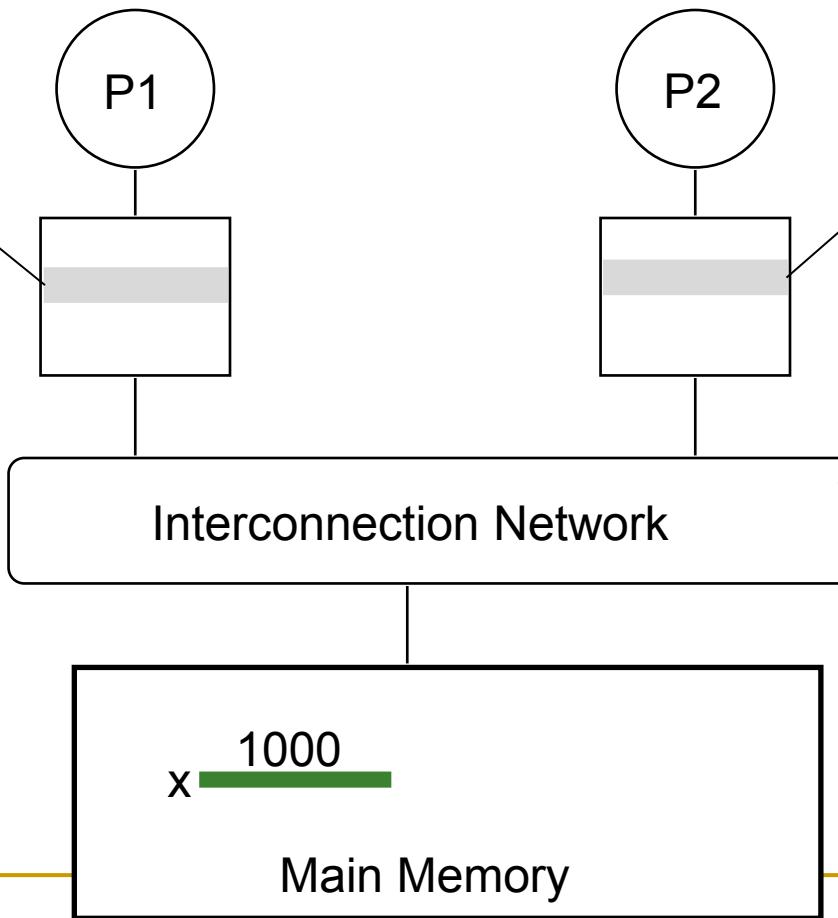
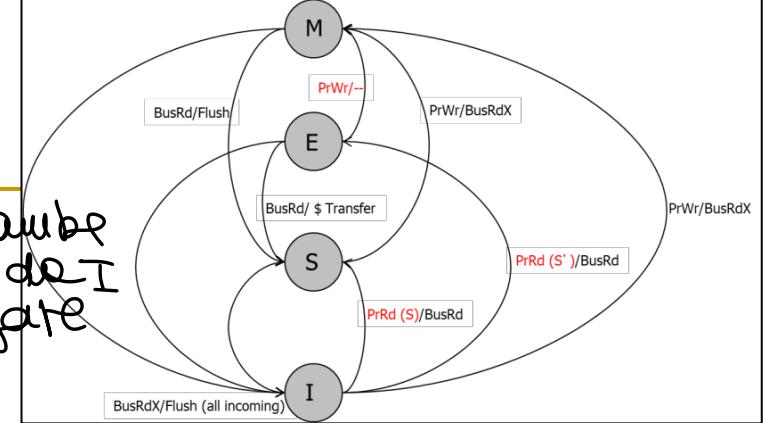
MESI State Machine



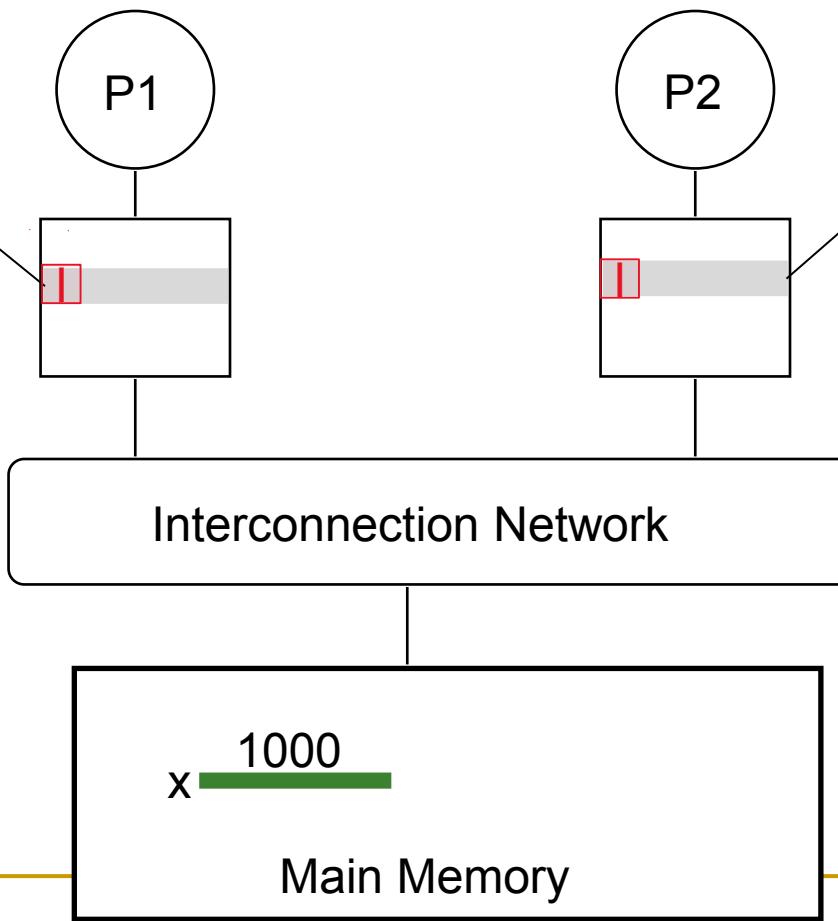
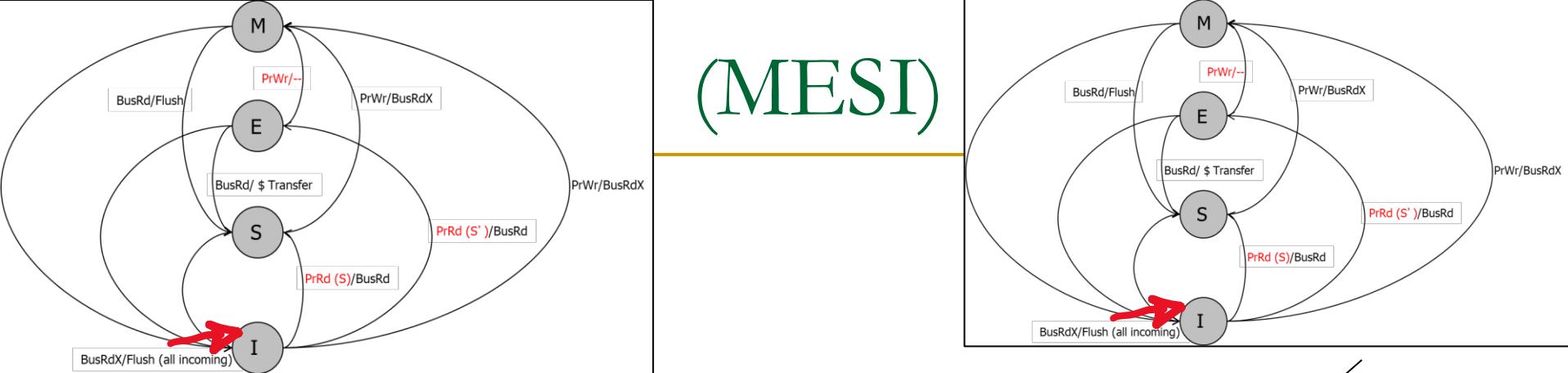


(MESI)

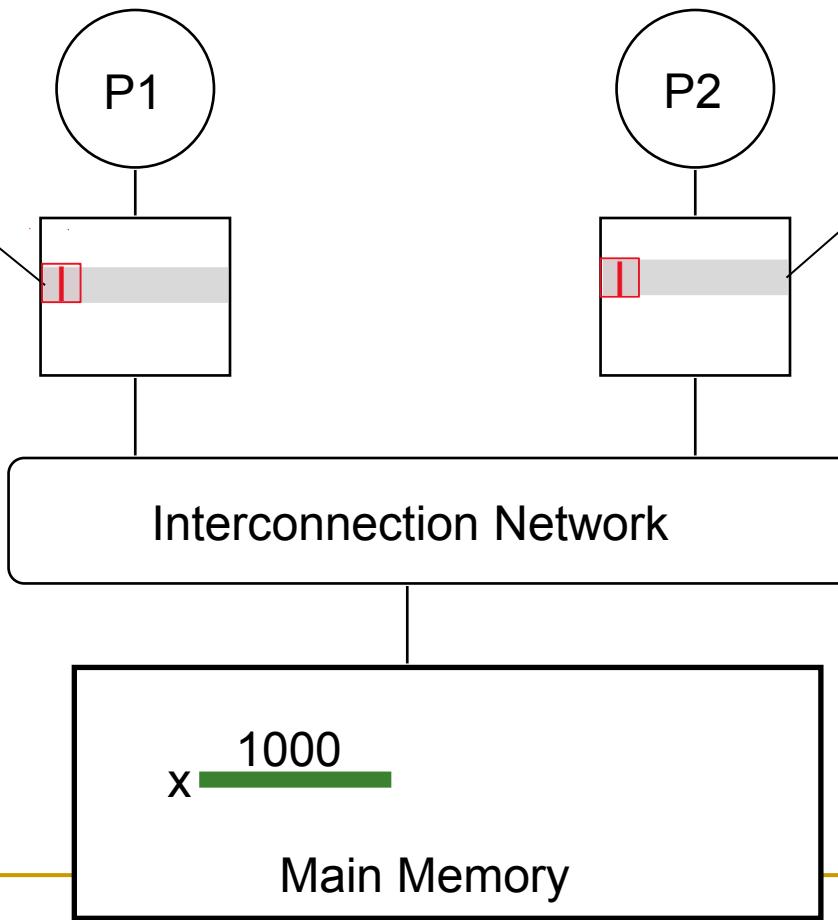
Assumo che entrambe le cache partano da I e che P2 voglia fare ld verso X,



(MESI)

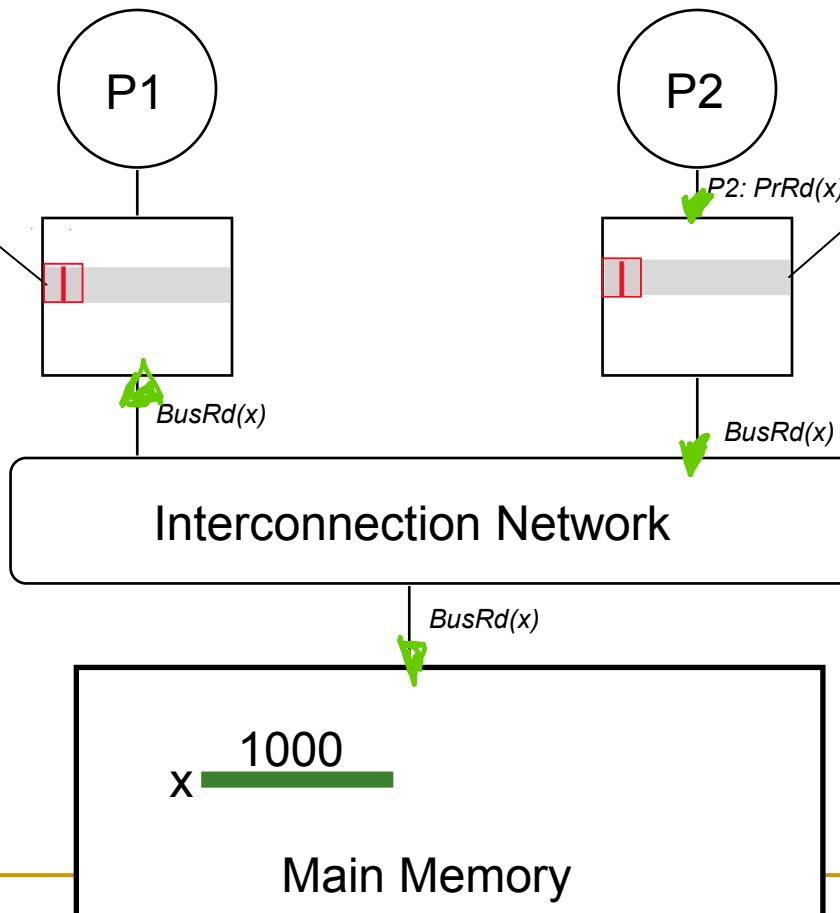
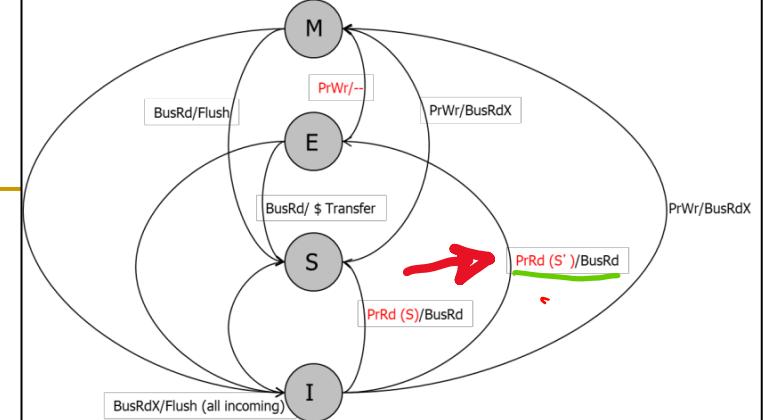
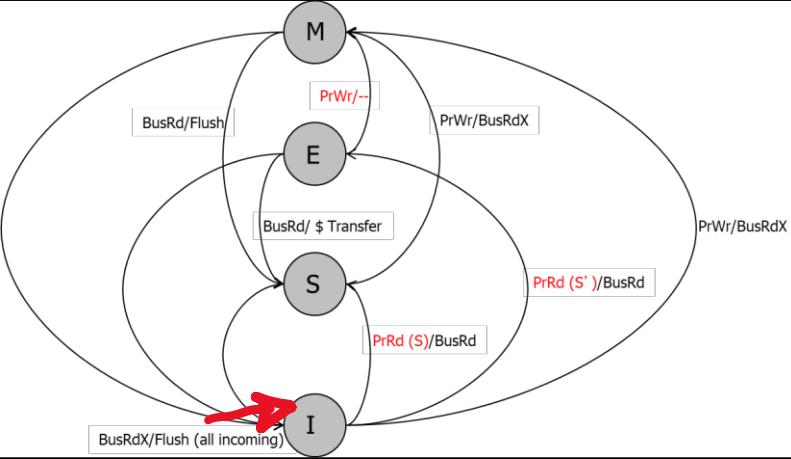


(MESI)



Id r2, x

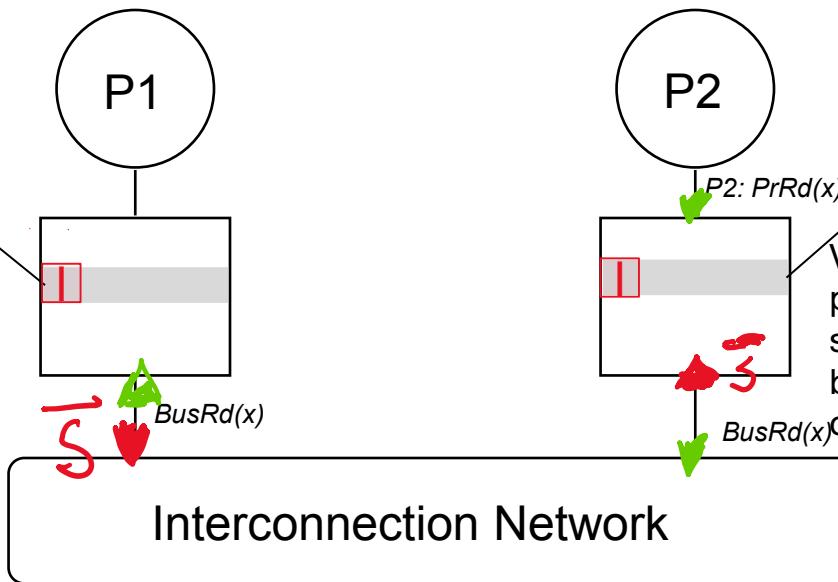
(MESI)



P2 comunica la richiesta bus read verso l'indirizzo x.

Prima di decidere in che stato andrà, gli altri cache agent ricevono la richiesta bus read relativa a quell'indirizzo, verificano se hanno una copia locale nella cache, se non c'è l'hanno come in questo caso, il P1 share un no. Quindi lo shared è negato.

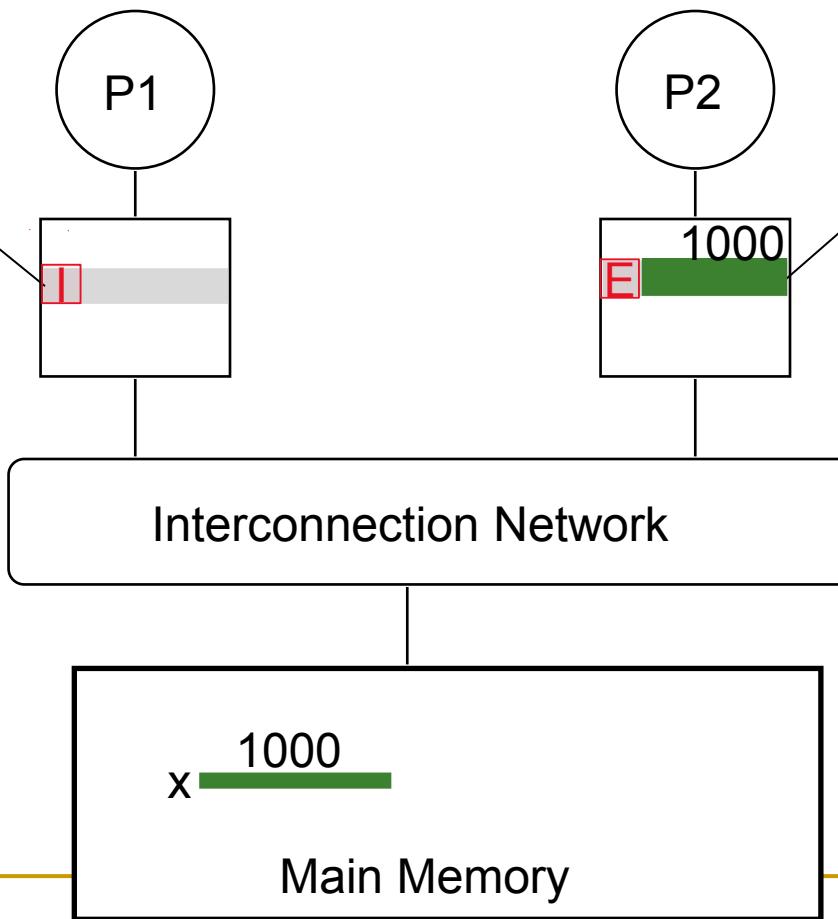
(MESI)



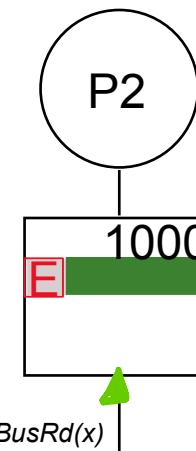
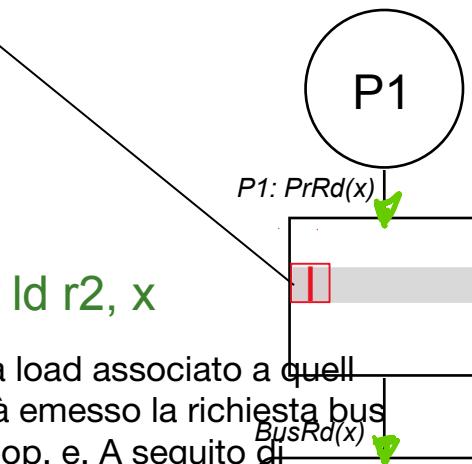
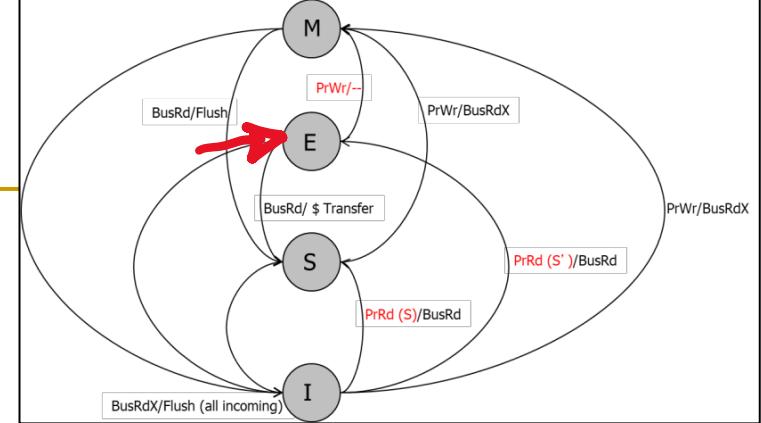
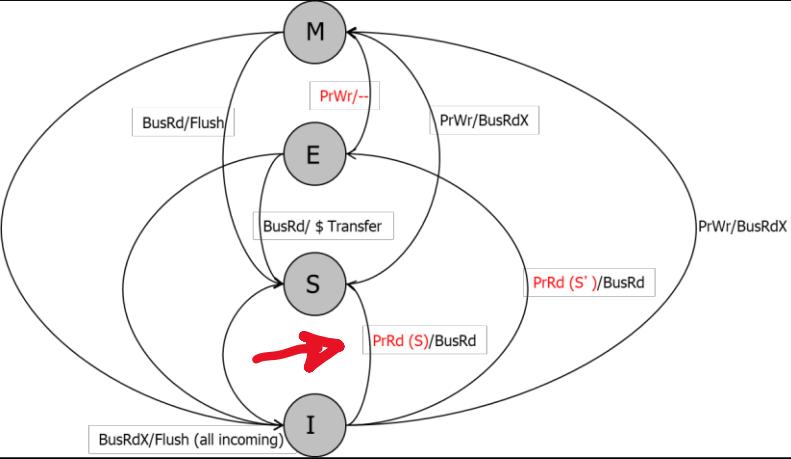
Id r2, x

Viene ricevuto not shared e quindi può passare direttamente allo stato exclusive, dopo aver letto il blocco di memoria associato a quell indirizzo.

(MESI)



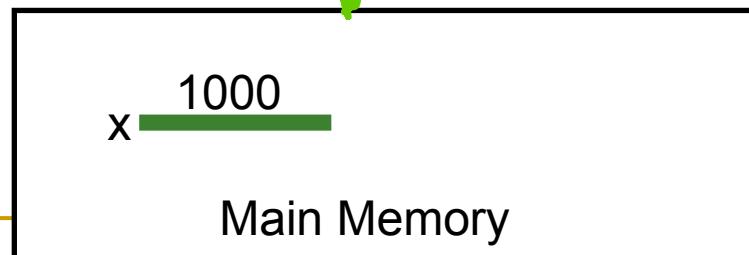
(MESI)



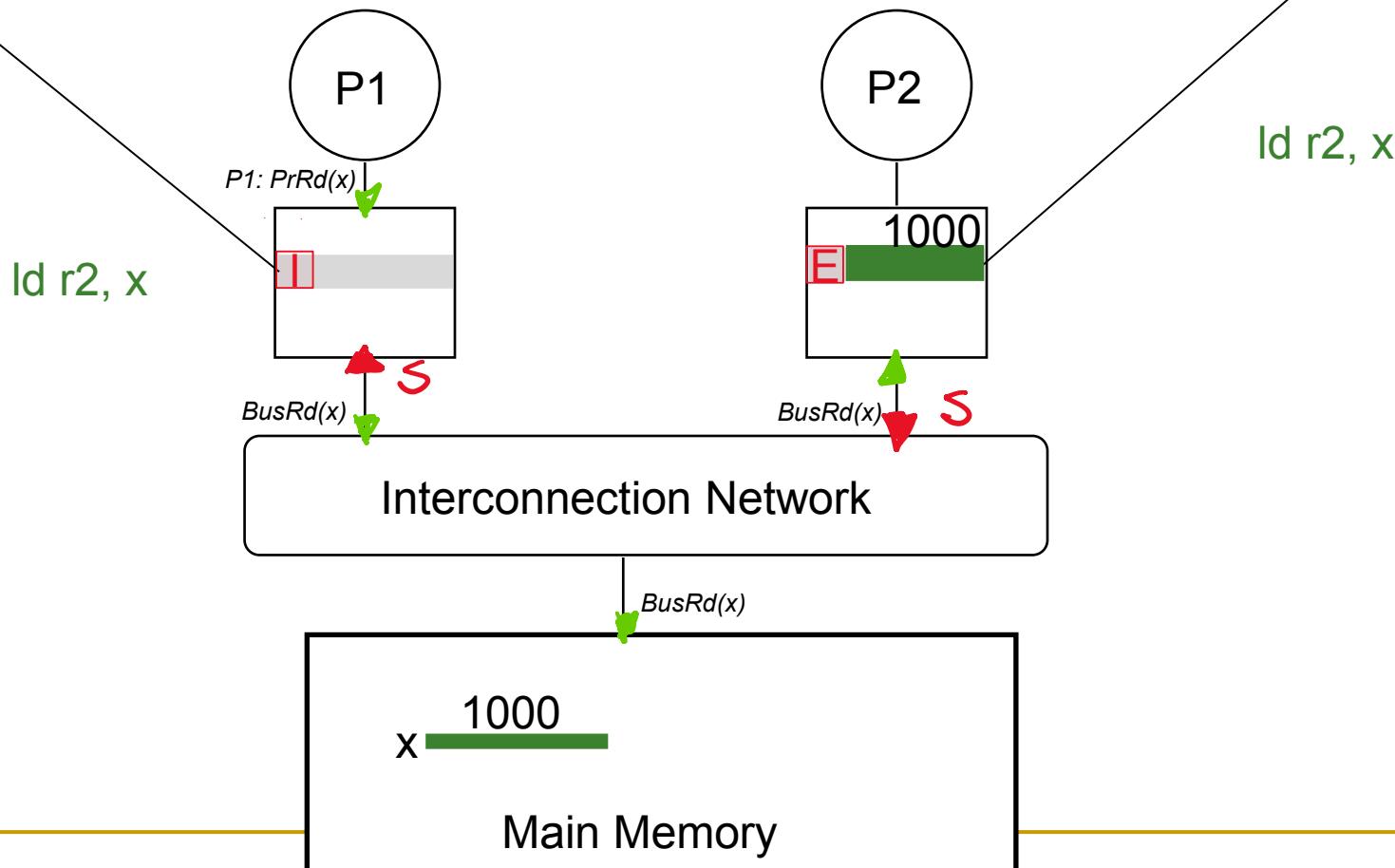
Id r₂, x

Se il p2 fa una load associato a quell indirizzo, verrà emesso la richiesta bus read sullo snoop, e. A seguito di questa richiesta il cache agent di p2 dichiara di avere una copia locale, quindi shared è vero, quindi a seguito della richiesta di processor read e shared vero va verso lo stadio S.

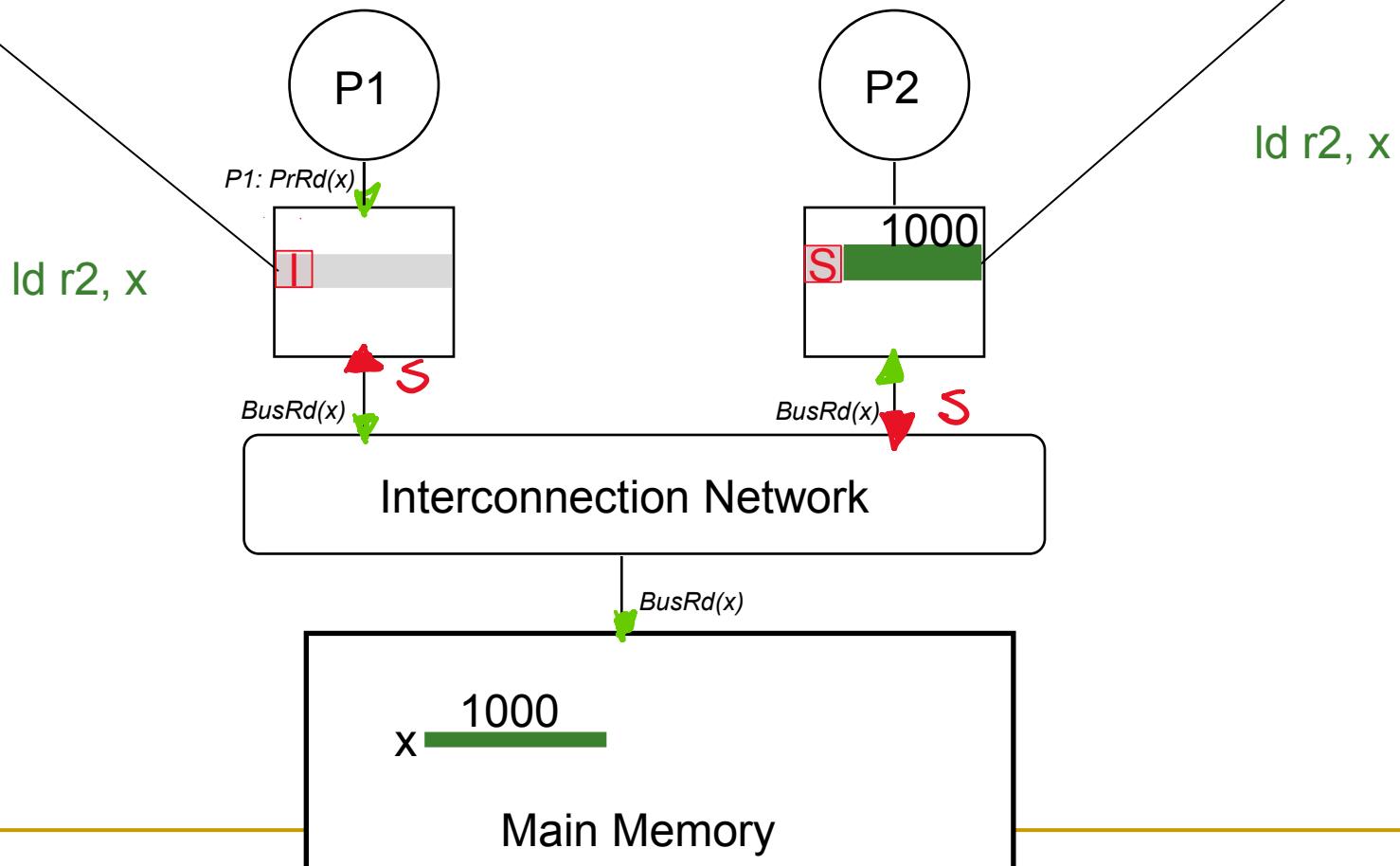
Interconnection Network



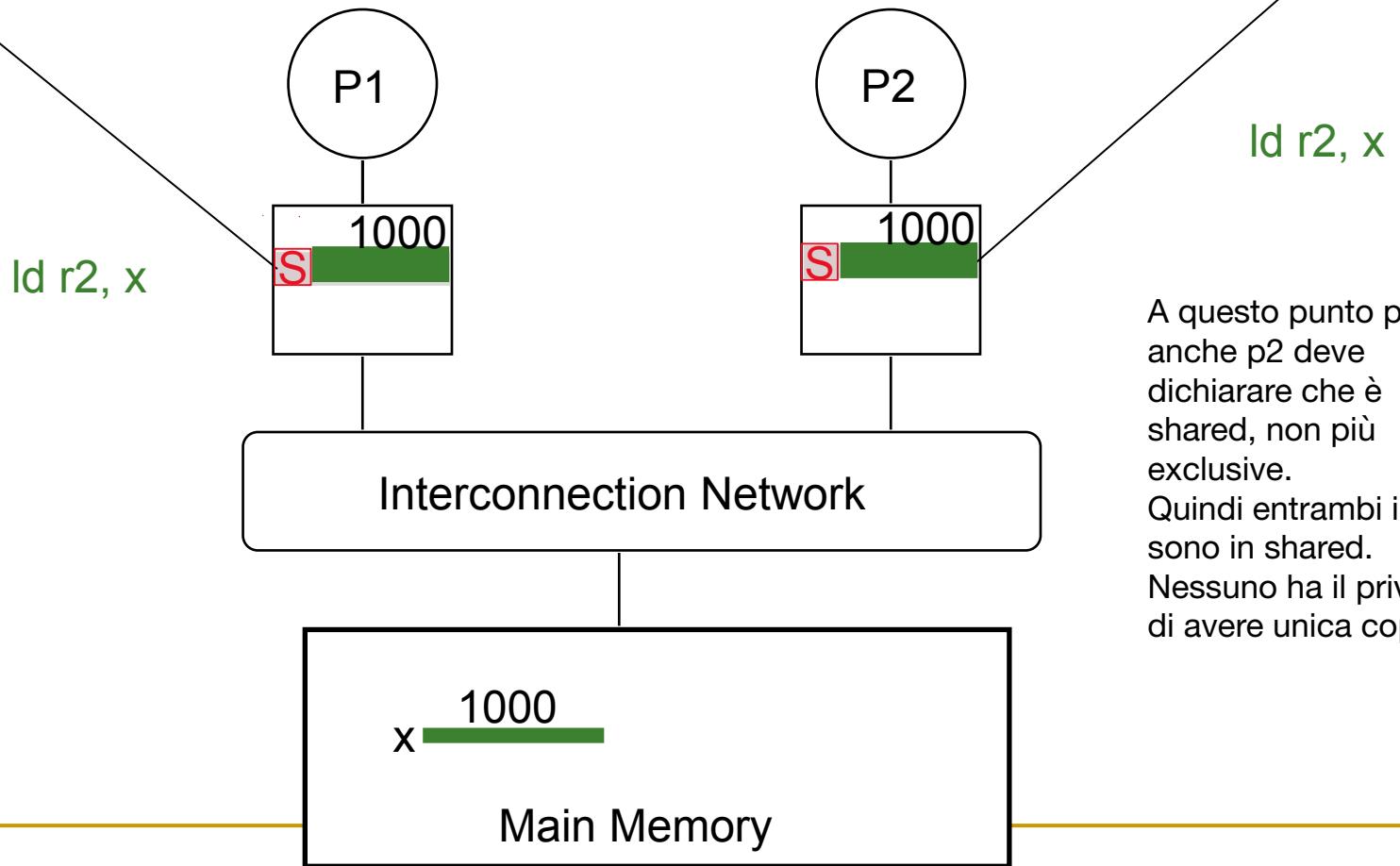
(MESI)



(MESI)

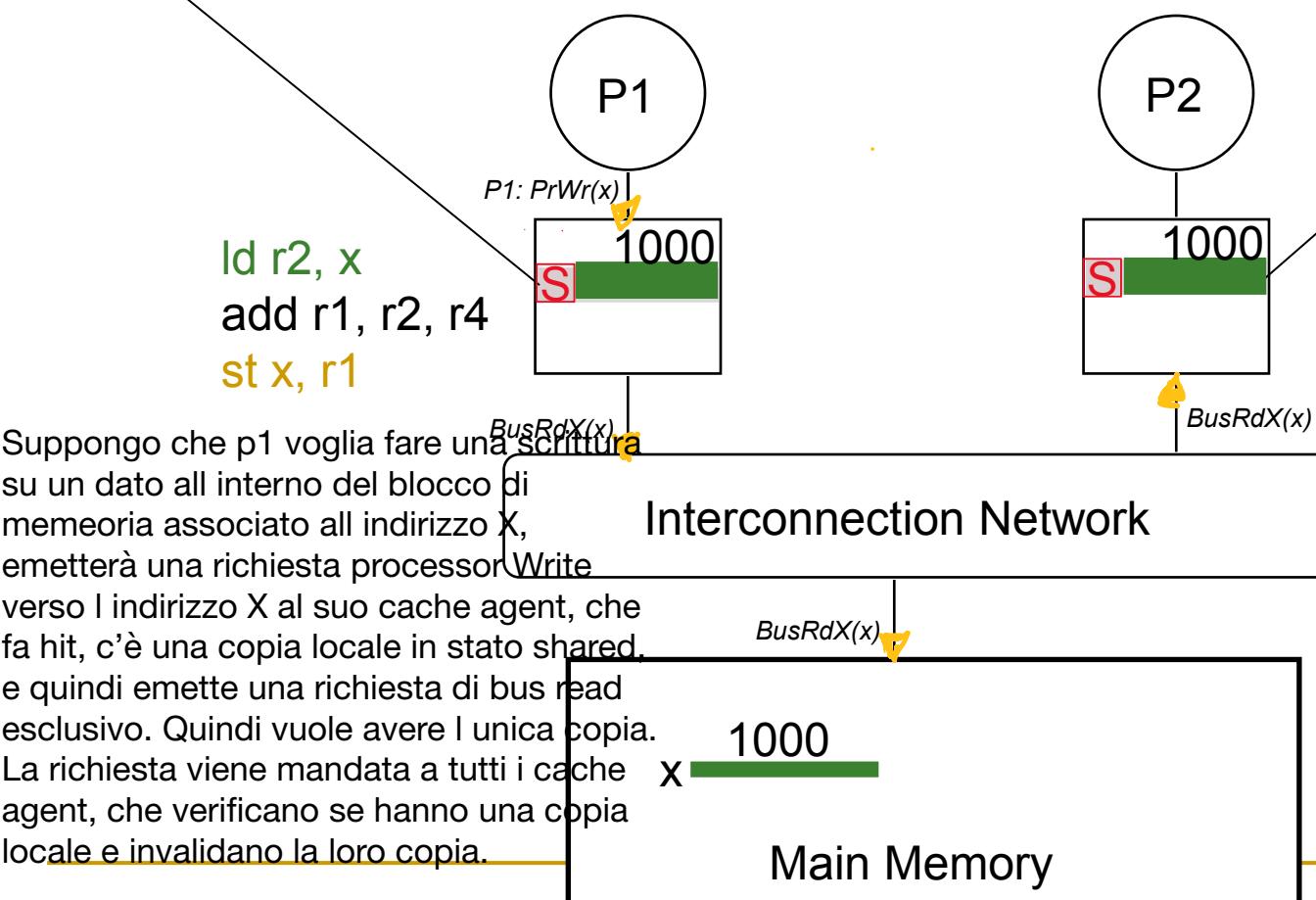


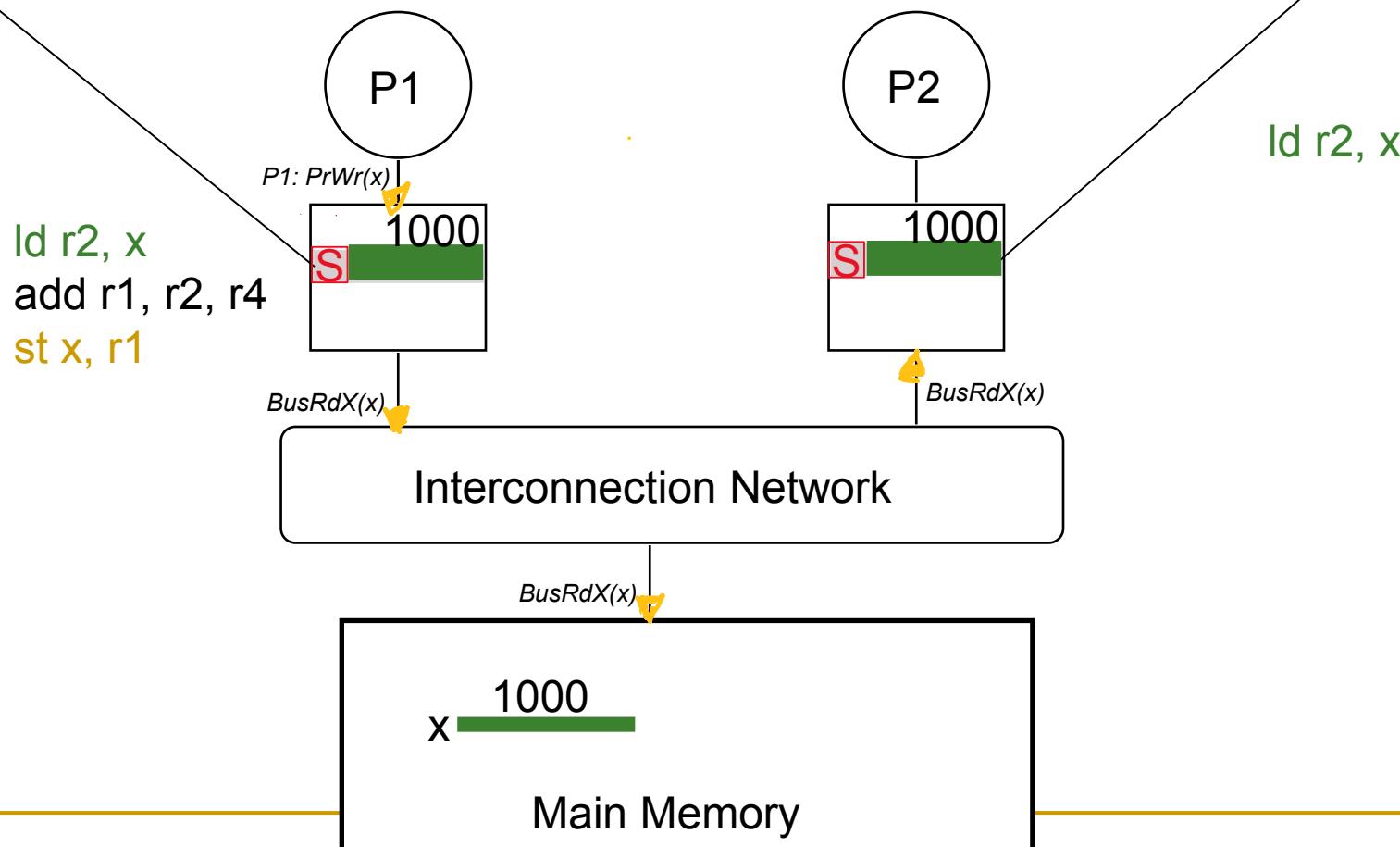
(MESI)



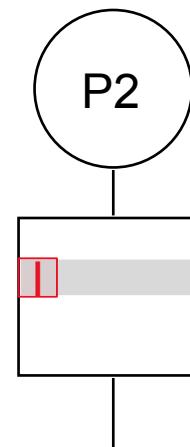
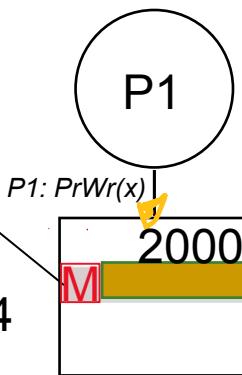
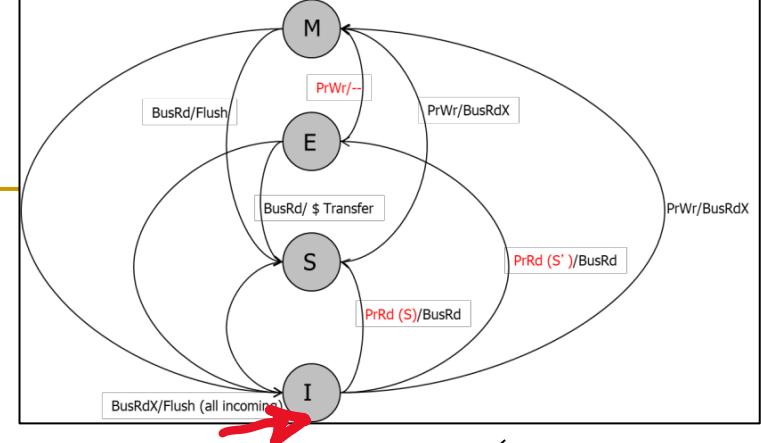
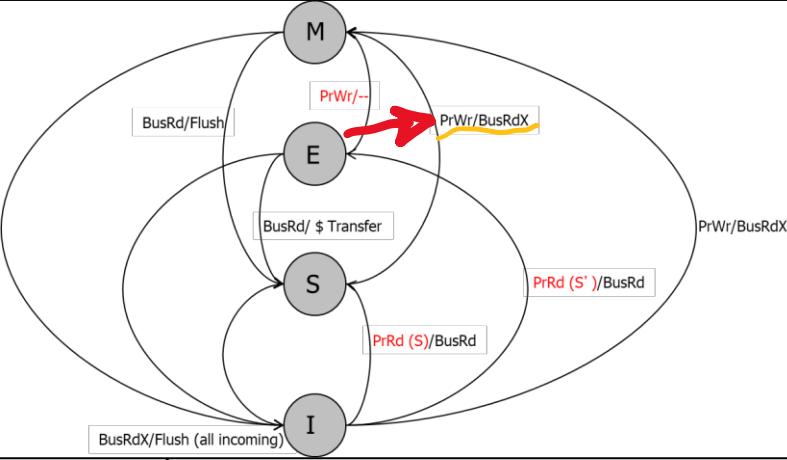


(MESI)





(MESI)



Id r2, x

Id r2, x
add r1, r2, r4

st x, r1

A questo punto la copia locale associata a X si trova in stato M, se p2 ora vuole leggere lo stesso dato, andrà a fare una richiesta di bus read, che viene comunicata a tutti i cache agent, p1 risponde dicendo che ha una copia locale dirty.

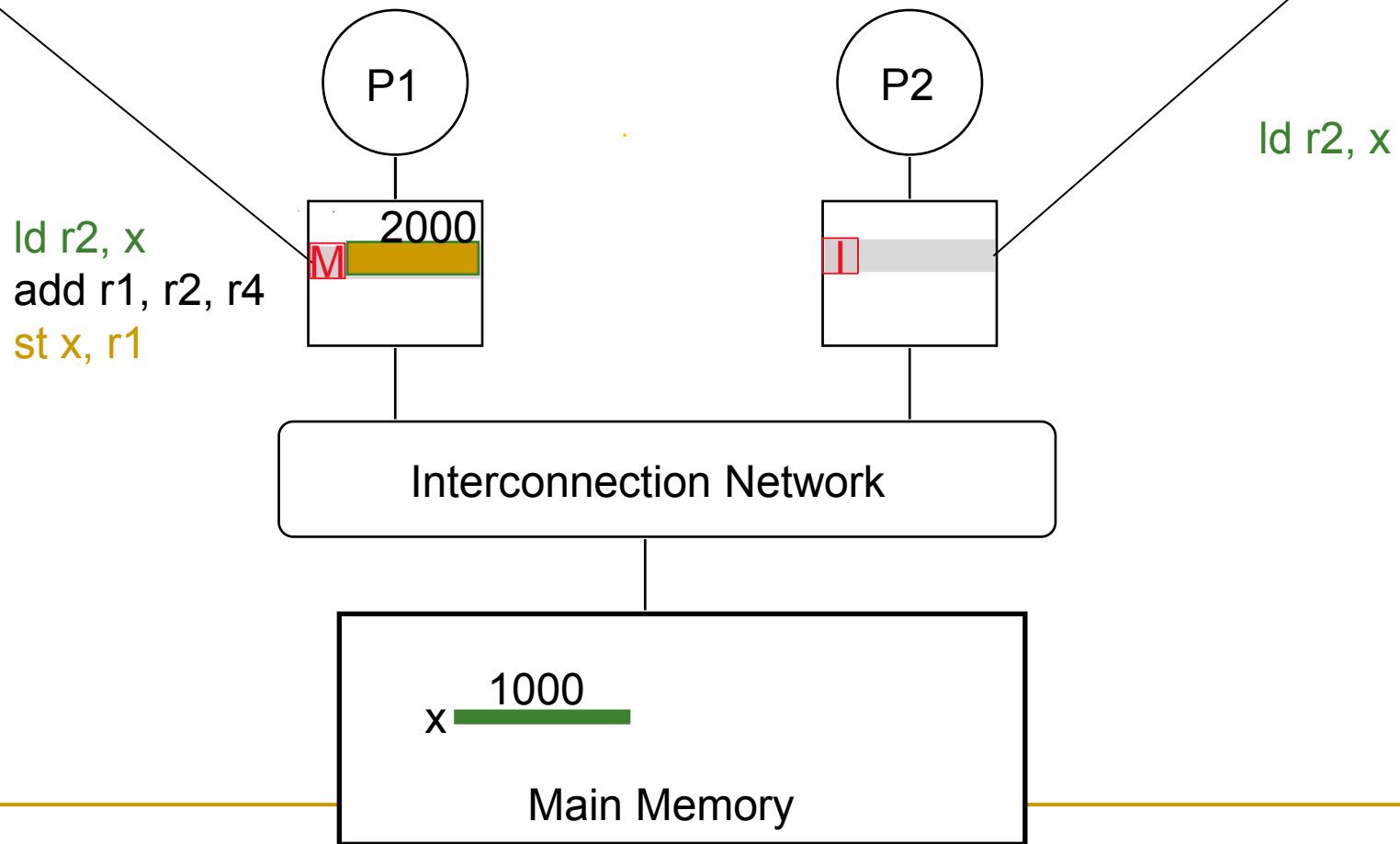
Ma prima di questo farà un ciclo di wb , scrivendola in memory, poi risponde al bus di snoop con il segnale shared dicendo che ha una copia locale. A questo punto il p1 può leggere il dato dalla main memory e portarsi in shared.

Interconnection Network

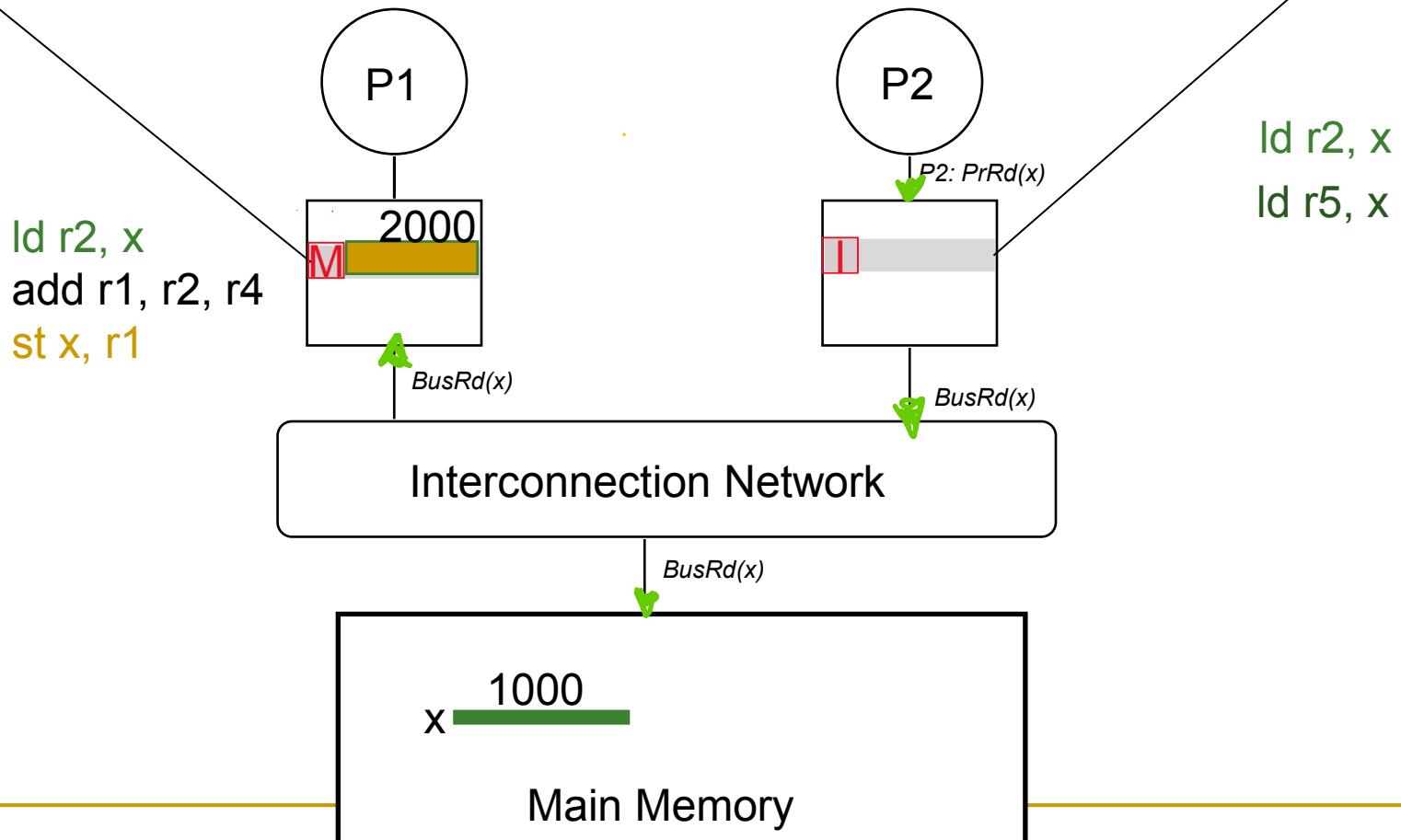
X 1000

Main Memory

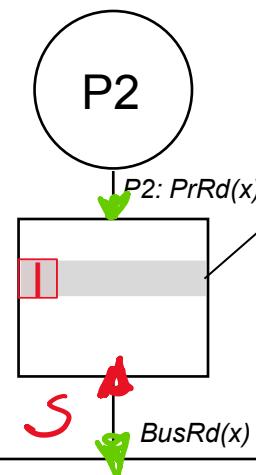
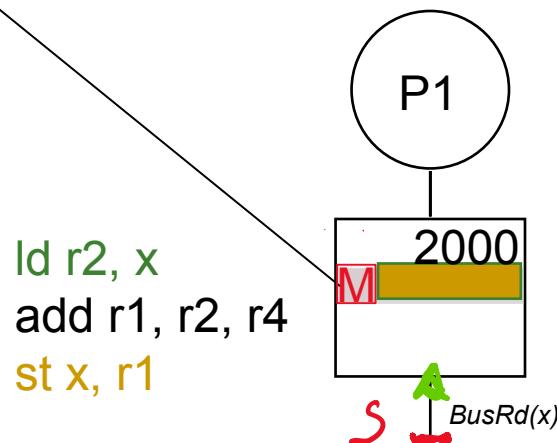
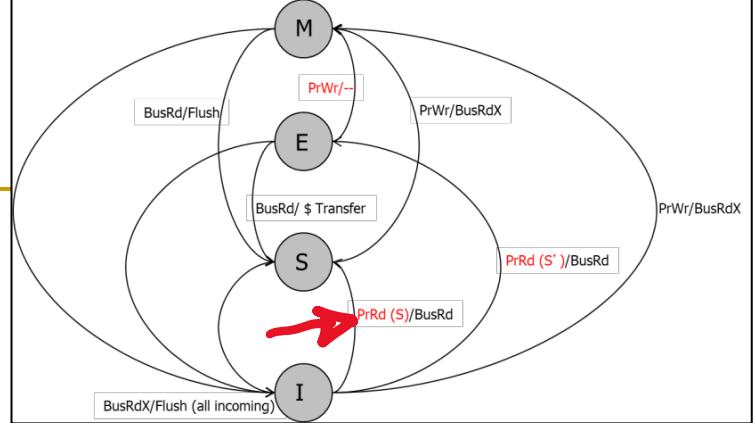
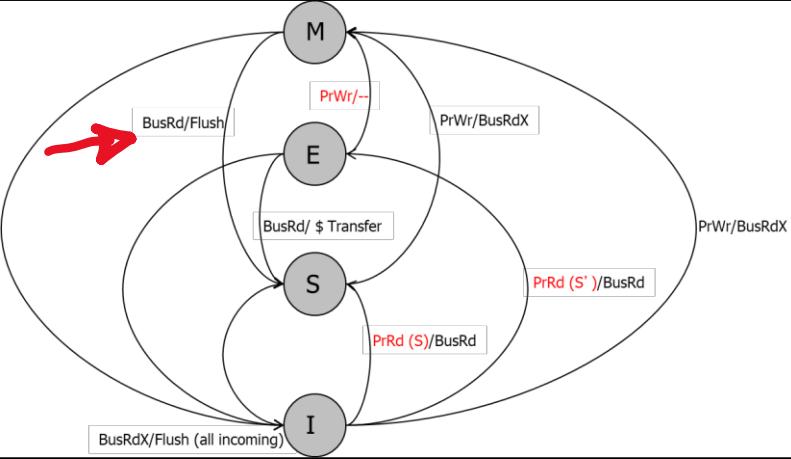
(MESI)



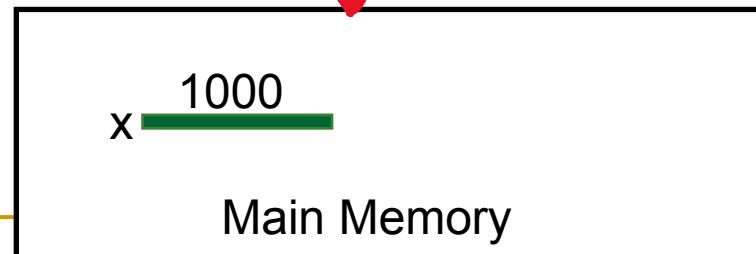
(MESI)



(MESI)



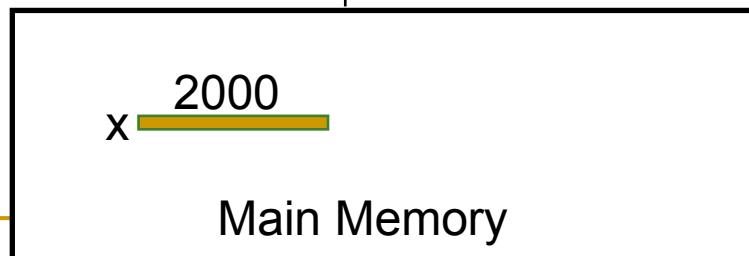
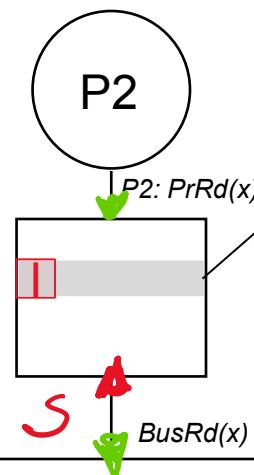
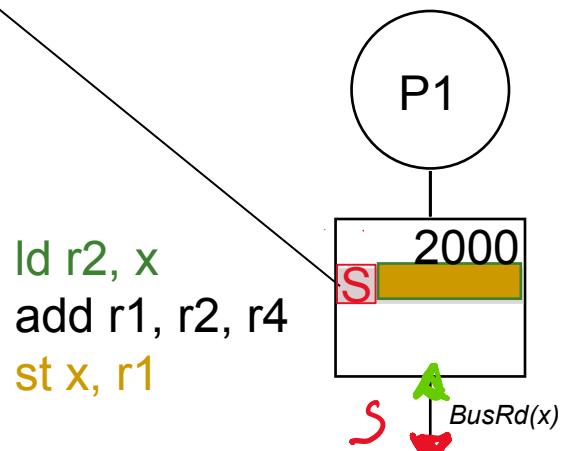
Interconnection Network



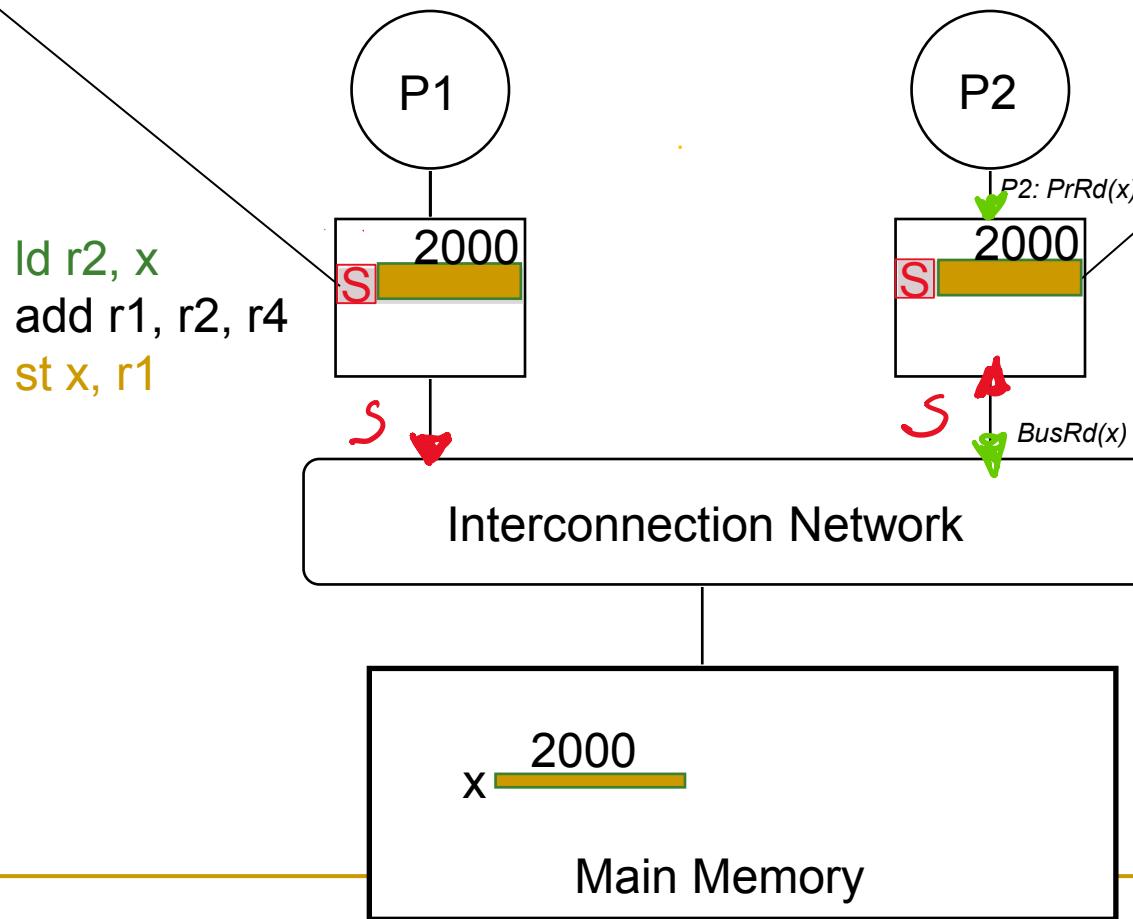
Id r2, x
Id r5, x

Id r2, x
add r1, r2, r4
st x, r1

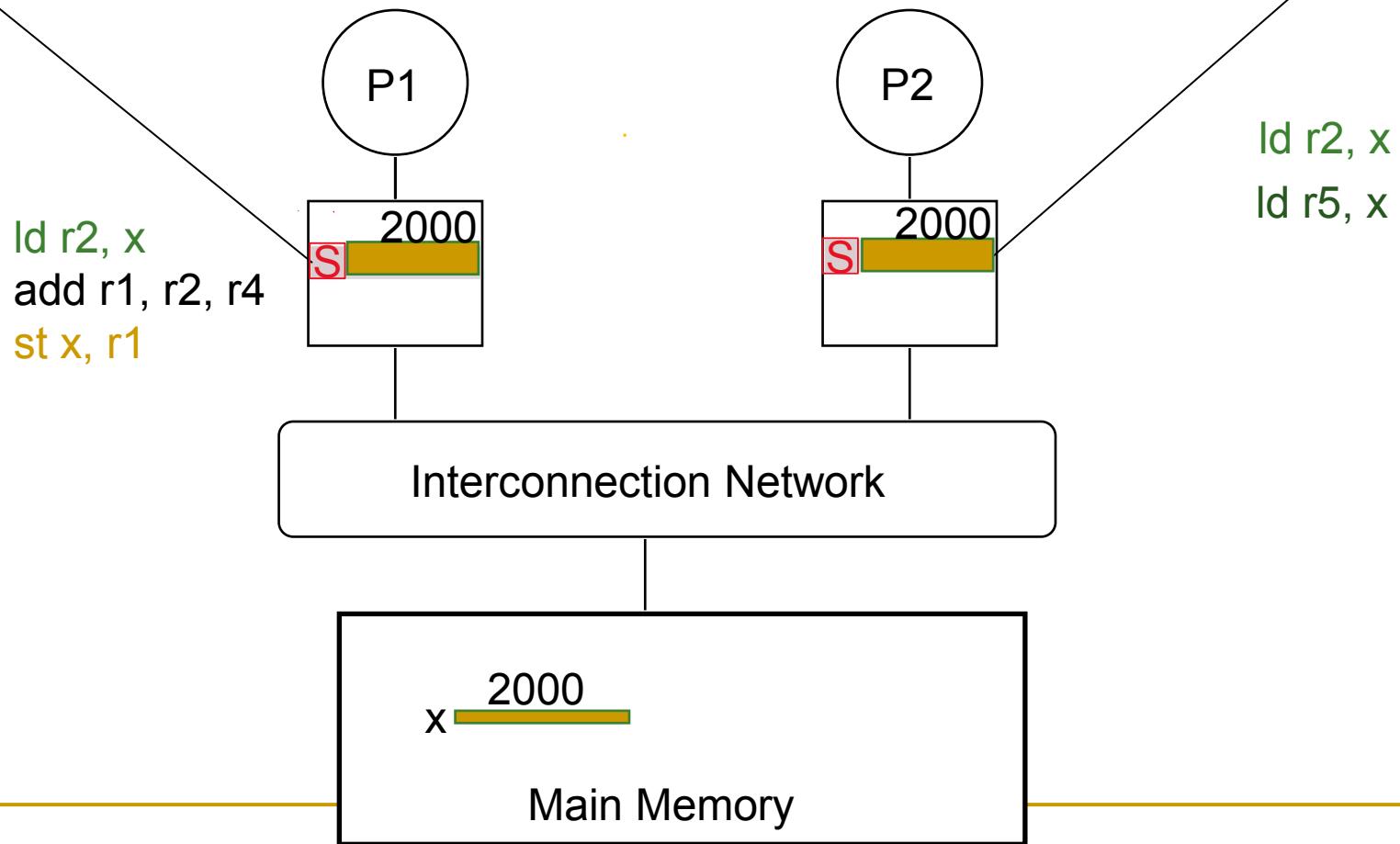
(MESI)

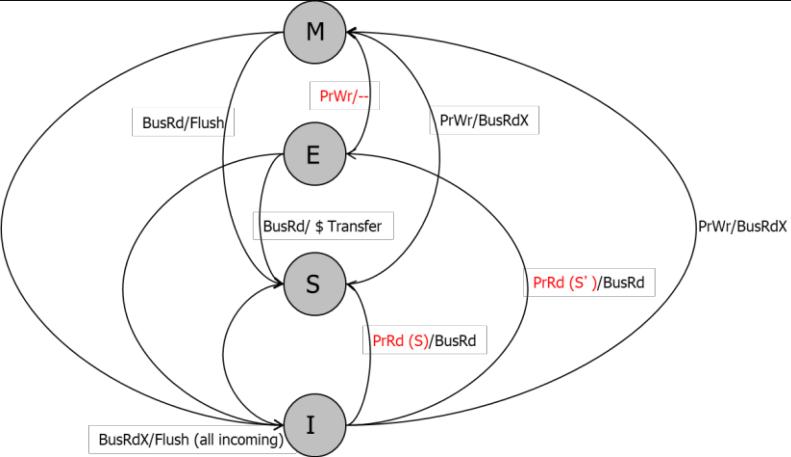


(MESI)



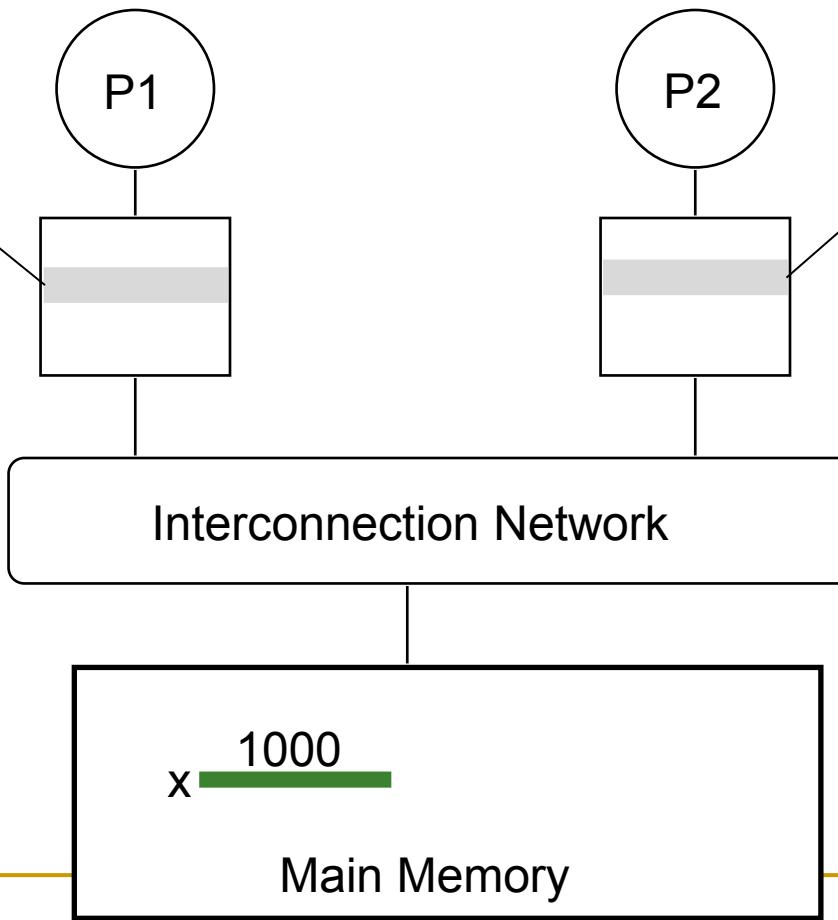
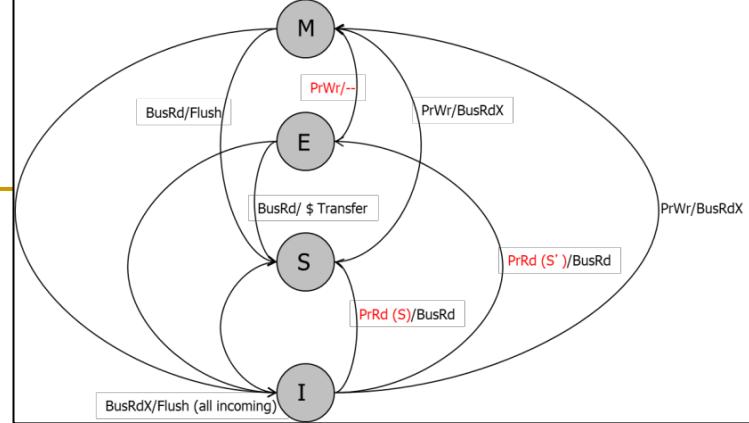
(MESI)



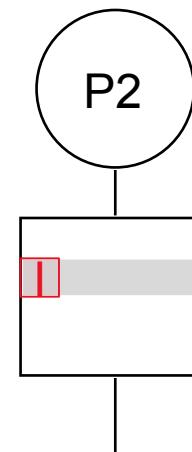
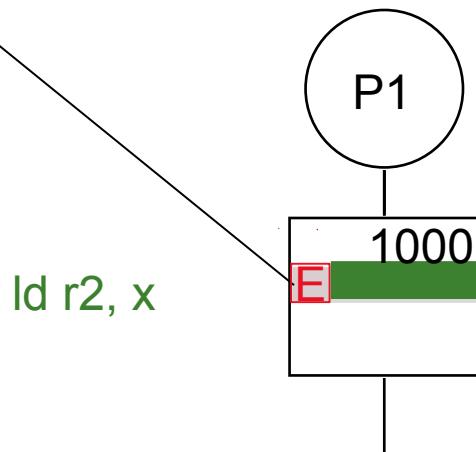


(MESI)

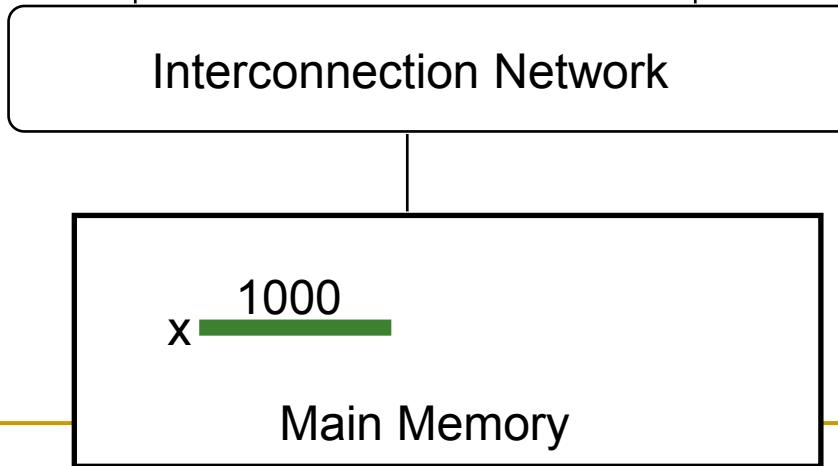
Altro caso: p1 ha una copia esclusiva del dato e vuole modificarla.



(MESI)

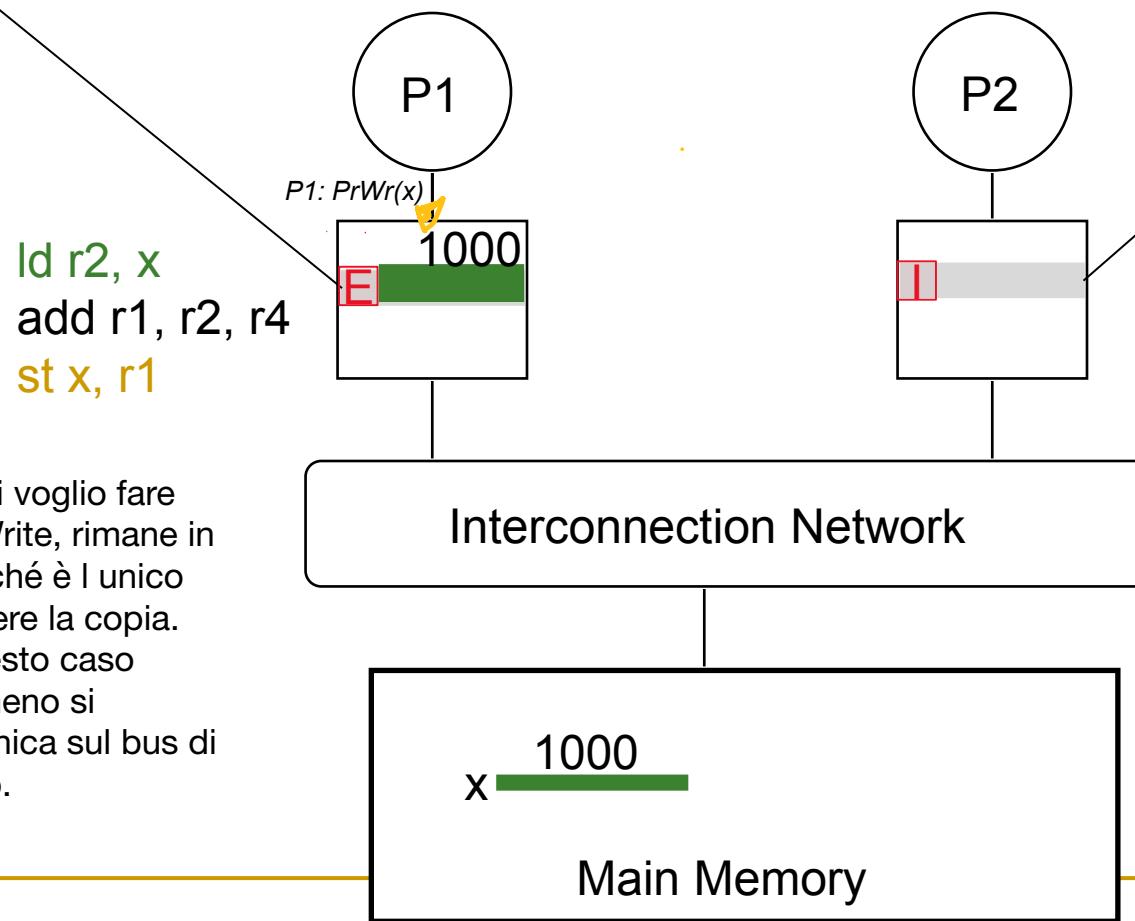


Non esistono copie,
p1 fa load, porta il
dato nella sua cache
e va in E,

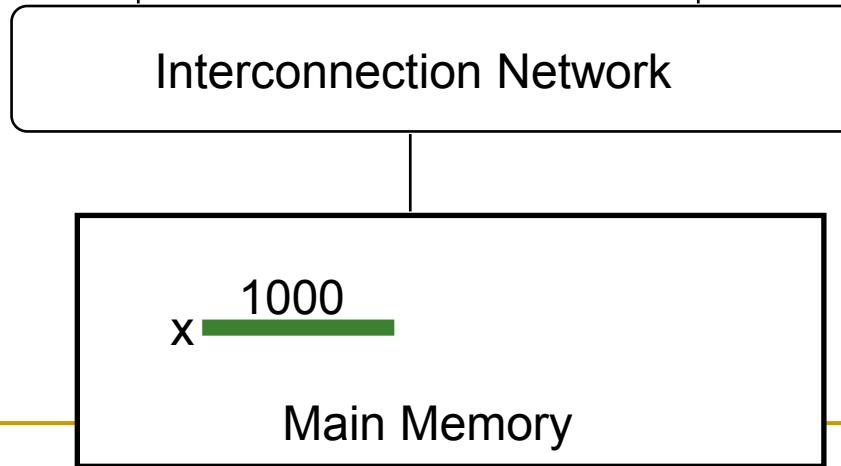
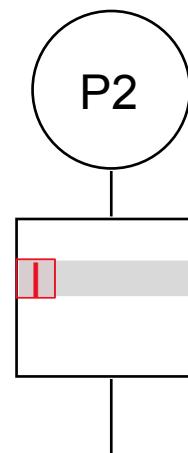
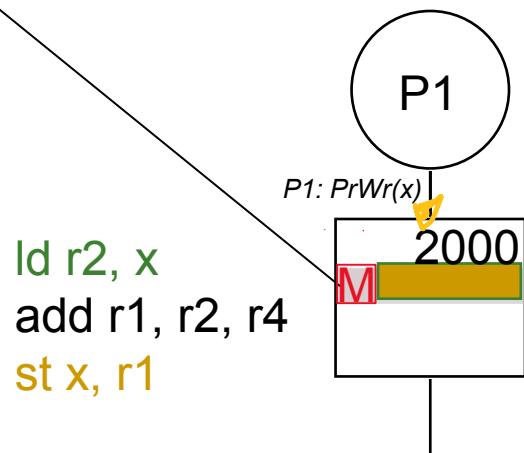
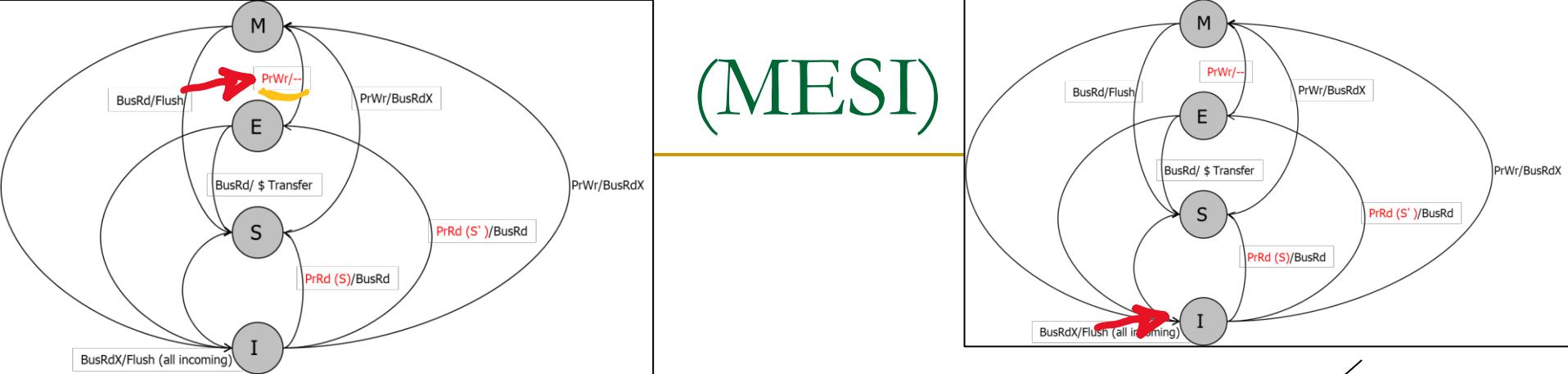


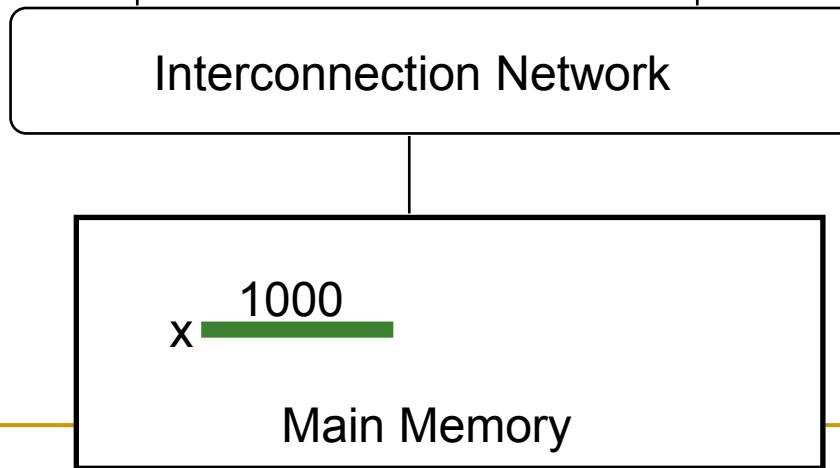
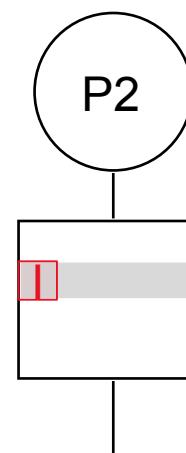
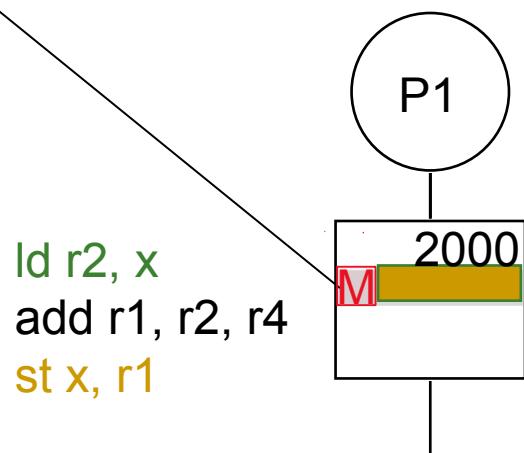


(MESI)



(MESI)





Snoopy Invalidation Tradeoffs

- Should a downgrade from M go to S or I?
 - S: if data is likely to be reused (before it is written to by another processor)
 - I: if data is likely to be not reused (before it is written to by another)
 - Cache-to-cache transfer
 - On a BusRd, should data come from another cache or memory?
 - Another cache
 - May be faster, if memory is slow or highly contended
 - Memory
 - Simpler: no need to wait to see if another cache has the data first
 - Less contention at the other caches
 - Requires writeback on M downgrade
 - Writeback on Modified->Shared: necessary?
 - One possibility: **Owner**(O) state (MOESI protocol)
 - One cache owns the latest data (memory is not updated)
 - Memory writeback happens when all caches evict copies
- ho passato il valore
aggiornato su un altro
cache agent, secca
aggiornare le memorie
x questo diventano
OWNER.*
- Quando la mia
copia viene invalidata,
dato sarà io owner a trascrivere il
valore nello main.*

The Problem with MESI

- Observation: Shared state requires the data to be clean
 - i.e., all caches that have the block have the up-to-date copy and so does the memory
- Problem: Need to write the block to memory when BusRd happens when the block is in Modified state
- Why is this a problem?
 - Memory can be updated unnecessarily → some other processor may want to write to the block again

Improving on MESI

- Idea 1: Do not transition from M→S on a BusRd. Invalidate the copy and supply the modified block to the requesting processor directly without updating memory
- Idea 2: Transition from M→S, but designate one cache as the owner (O), who will write the block back when it is evicted
 - Now “Shared” means “Shared and potentially dirty”
 - This is a version of the MOESI protocol

Tradeoffs in Sophisticated Cache Coherence Protocols

- The protocol can be optimized with more states and prediction mechanisms to
 - + Reduce unnecessary invalidates and transfers of blocks
- However, more states and optimizations
 - Are more difficult to design and verify (lead to more cases to take care of, race conditions)
 - Provide diminishing returns