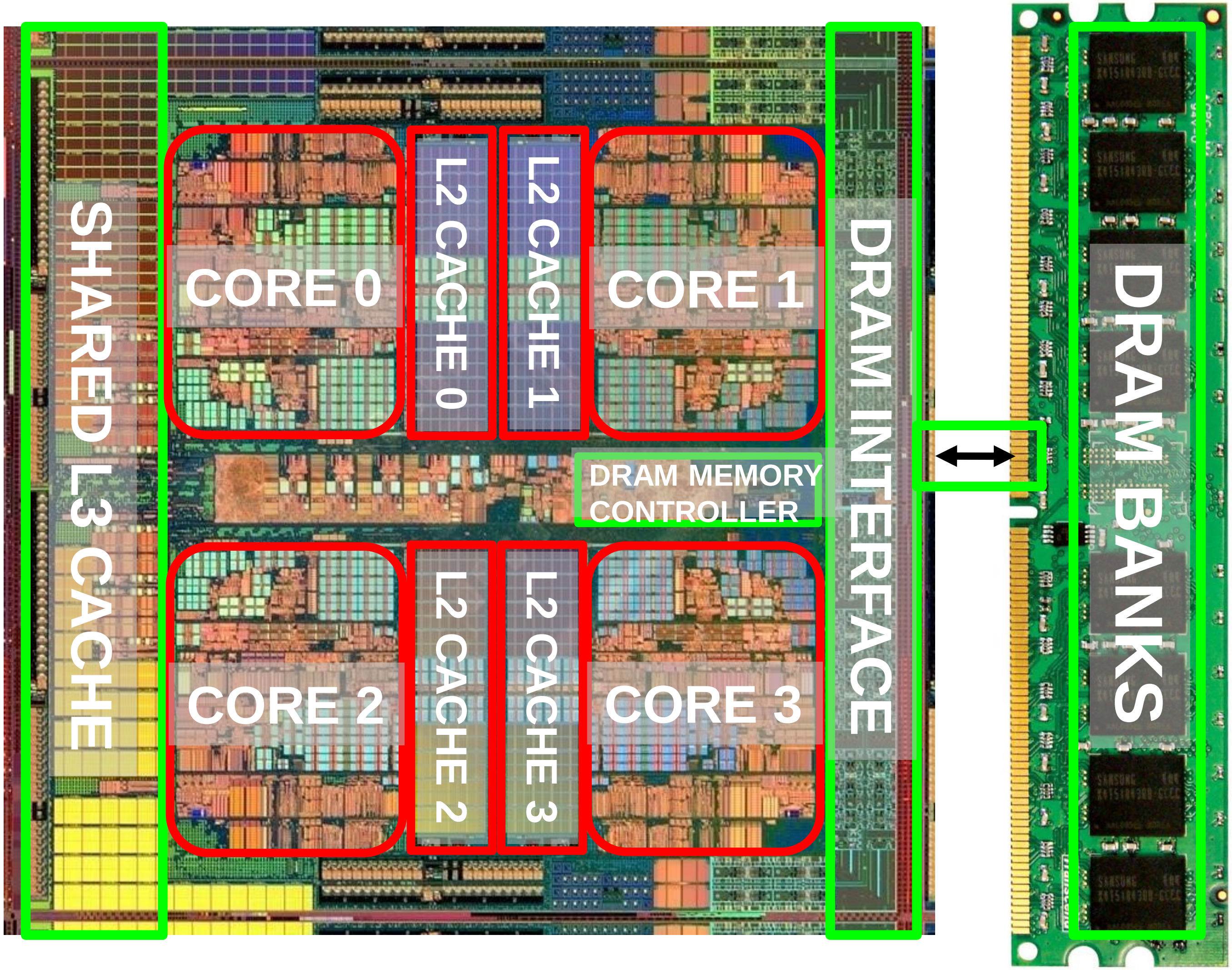
Gerarchia di Memoria e Cache

Andrea Bartolini – a.bartolini@unibo.it

Derivate da slides Onur Mutlu, Tulio Salmon Cinotti

Multi-Core Issues in Caching

Caches in a Multi-Core System



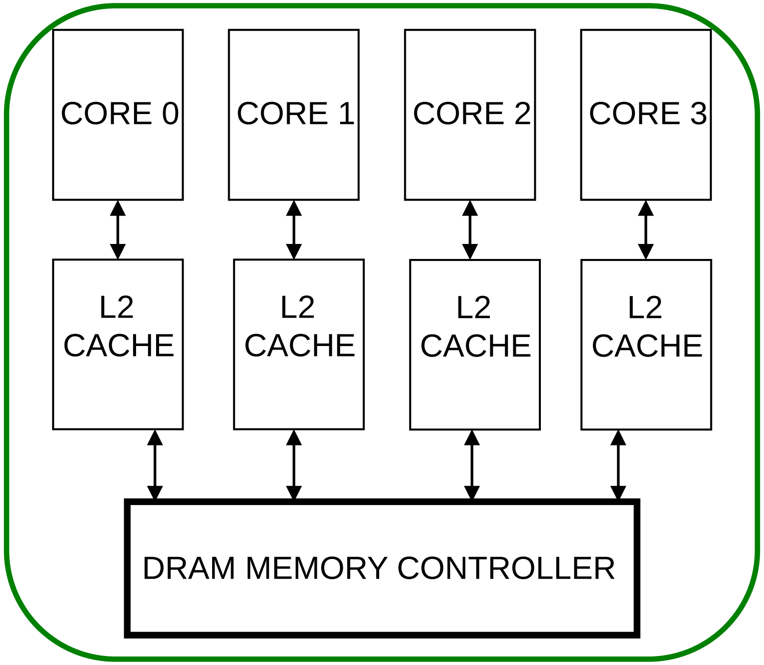
Caches in Multi-Core Systems

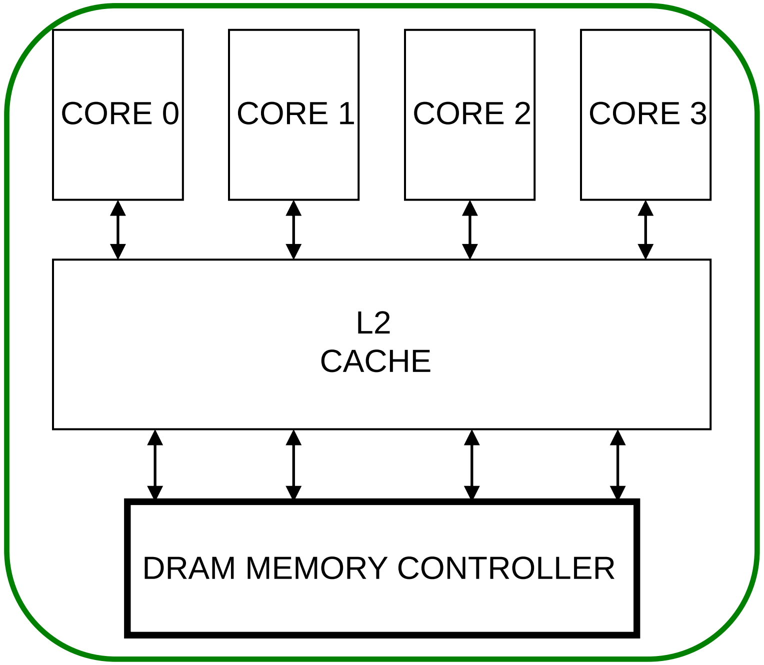
- L'efficienza della cache diventa ancora più importante in un sistema multi-core/multithread
  - Memory bandwidth è prioritaria
  - Lo spazio di cache è una risorsa limitata tra core/thread

- 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 Caches Between Cores

- Advantages:
  - High effective capacity
  - 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

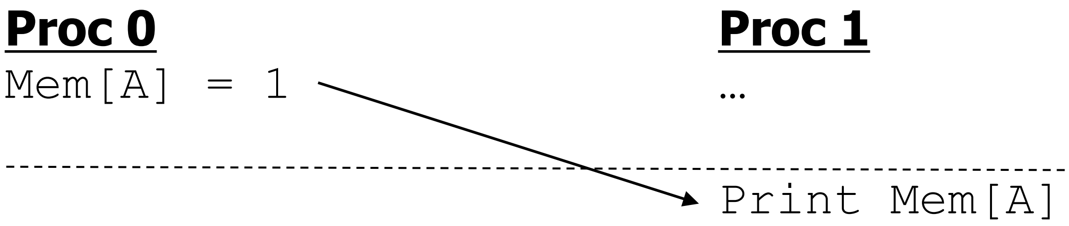
- Consistency is about ordering of all memory operations from different processors (i.e., to different memory locations)
  - Global ordering of accesses to *all* memory *locations*

- 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



Proc 0

`Mem[A] = 1`

Proc 1

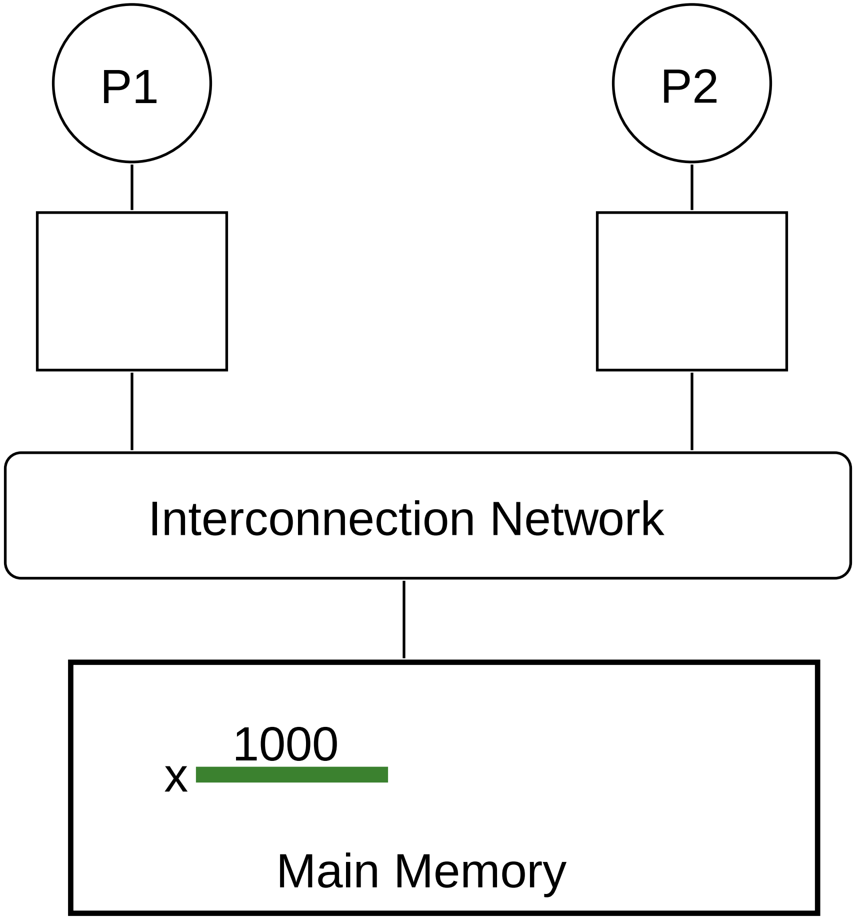
…

`Print Mem[A]`

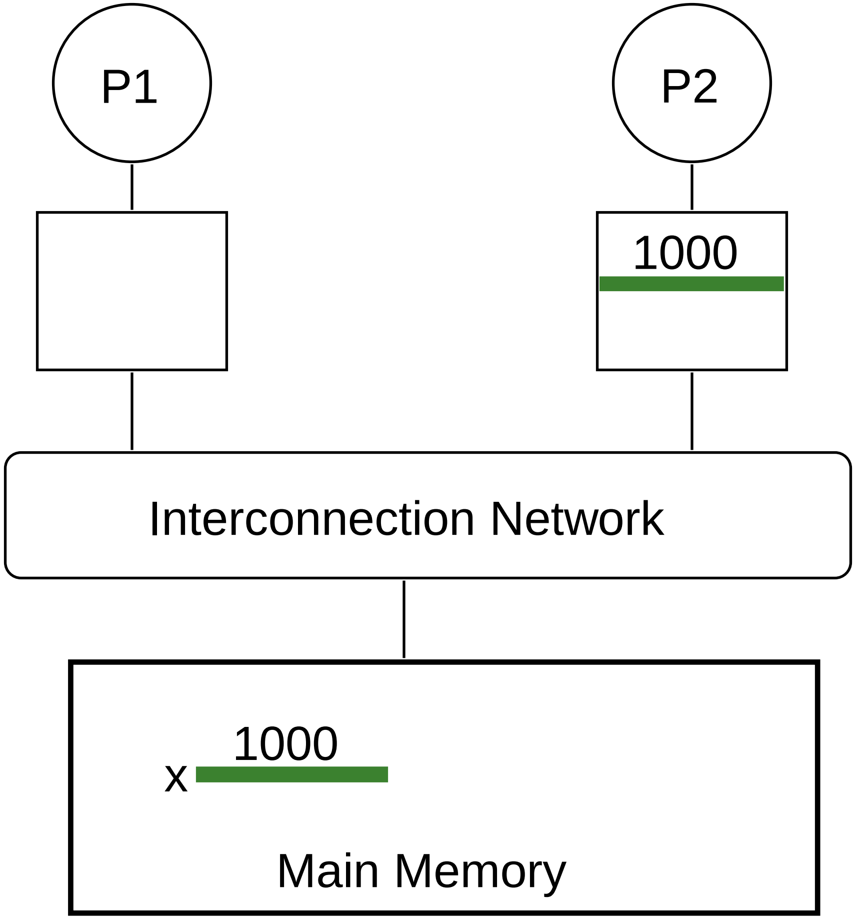
- Each read should receive the value last written by anyone
  - This requires synchronization (what does last written mean?)
- What if Mem[A] is cached (at either end)?

Cache Coherence

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

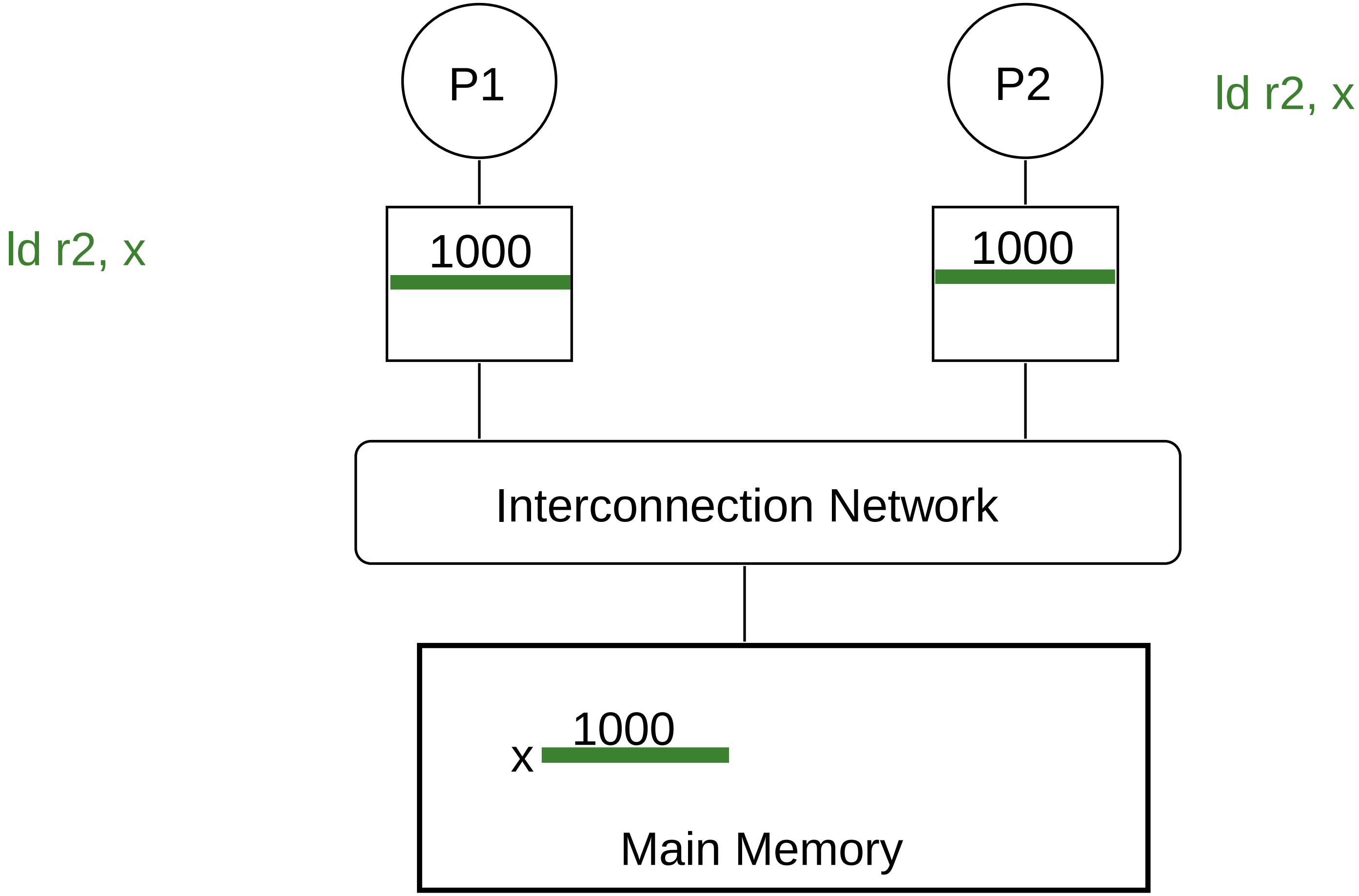


The Cache Coherence Problem



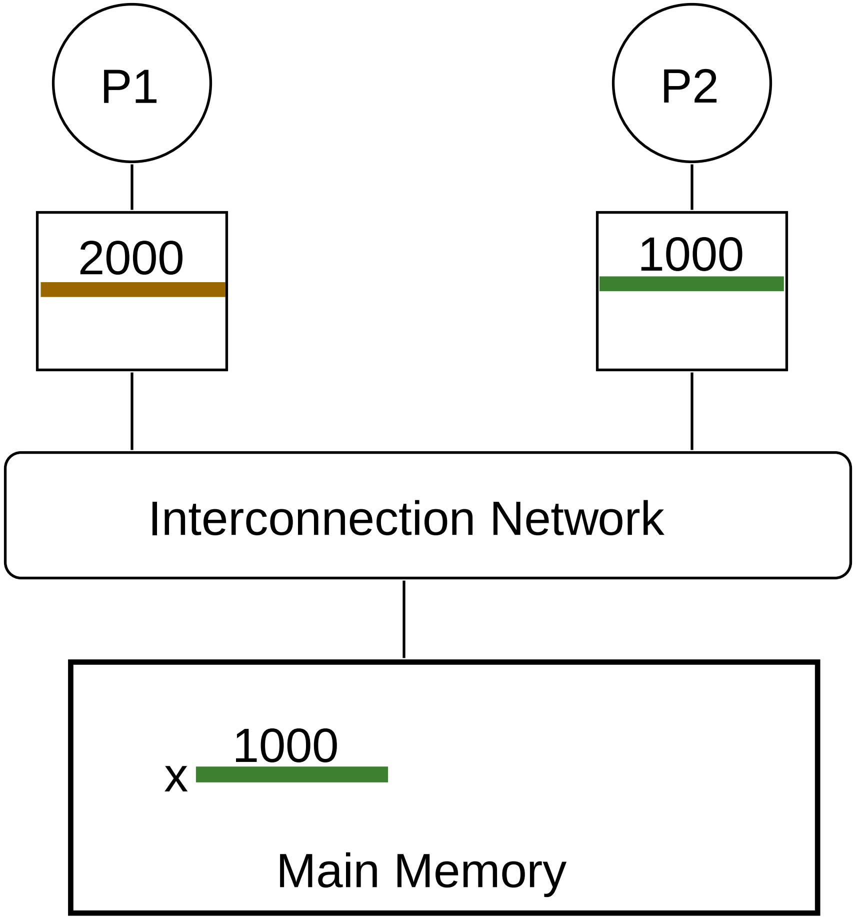
ld r2, x

The Cache Coherence Problem



The Cache Coherence Problem

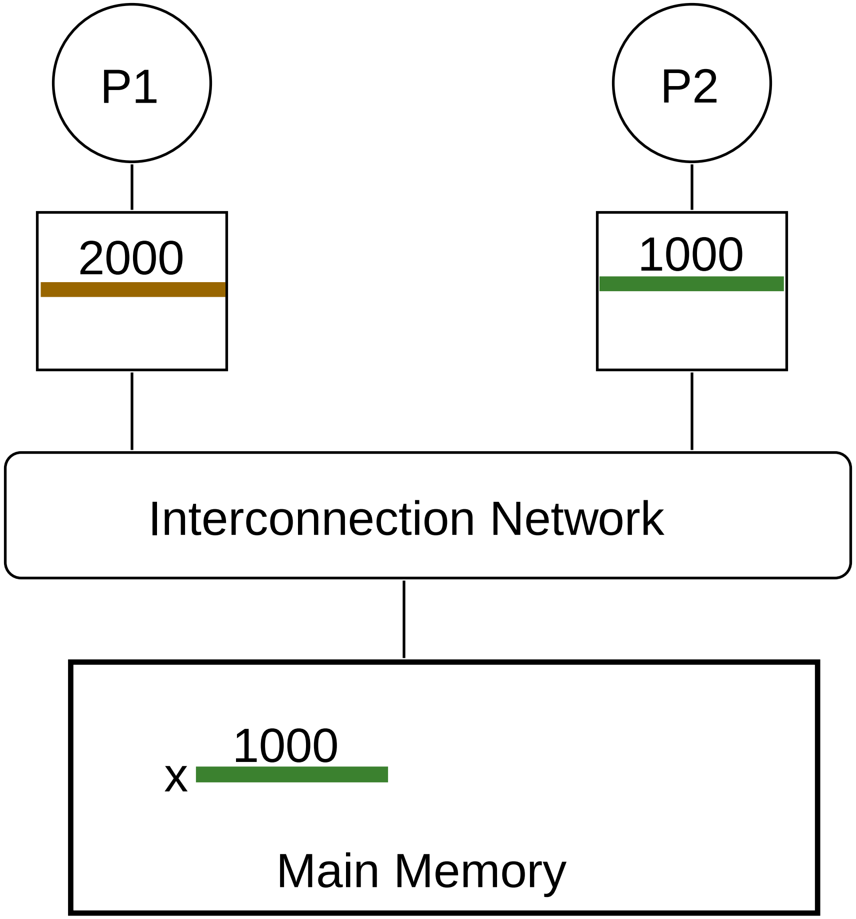
ld r2, x
add r1, r2, r4
st x, r1



ld r2, x

The Cache Coherence Problem

ld r2, x
add r1, r2, r4
st x, r1



ld r2, x

Non deve leggere 1000!

ld r5, x

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
  - 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

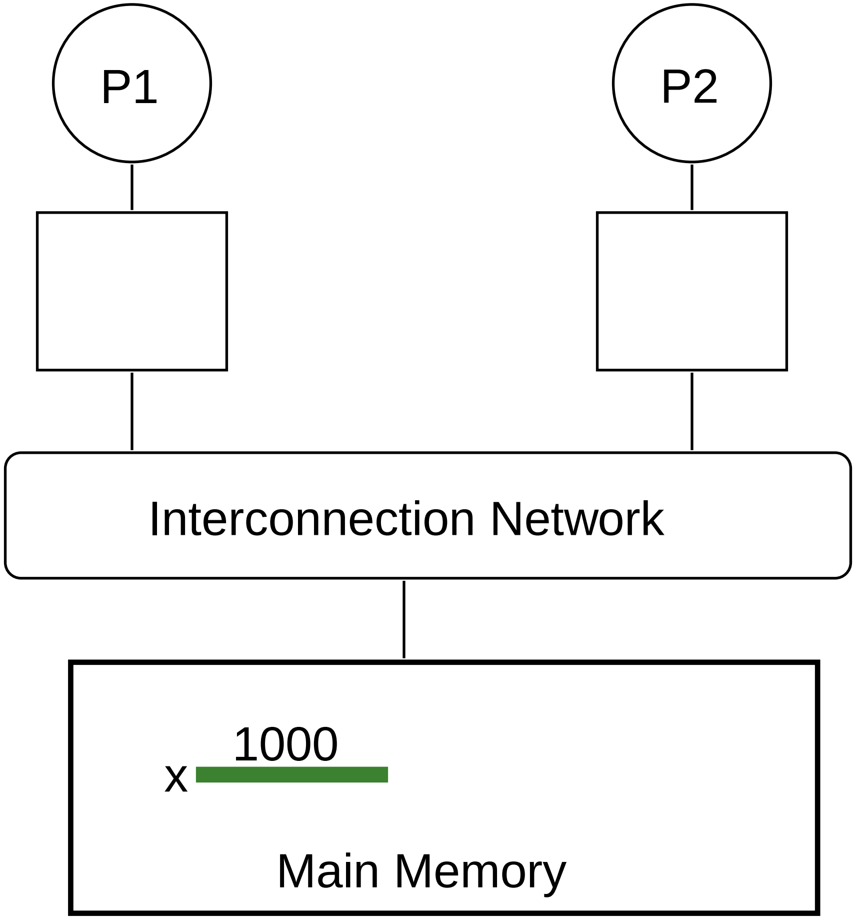
- Writes to location A by P0 should be seen by P1 (eventually), and all writes to A should appear in some order

- 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

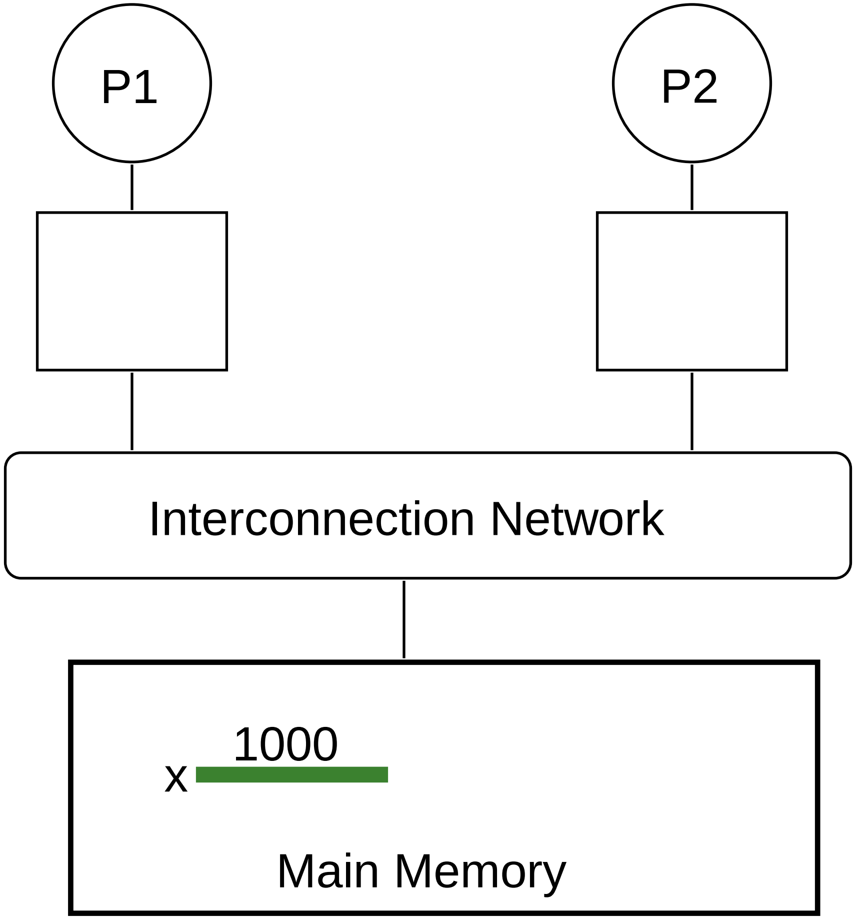
Cache Coherence

- 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

Coherence: Update vs. Invalidate

- How can we *safely update replicated data?*
  - Option 1 (Update protocol): push an update to all copies
  - Option 2 (Invalidate protocol): ensure there is only one copy (local), update it

- **On a Read:**
  - If local copy is Invalid, put out request
  - (If another node has a copy, it returns it, otherwise memory does)

Coherence: Update vs. Invalidate (II)

- **On a Write:**
  - Read block into cache as before

Update Protocol:

  - 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**
  - + 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**
  - + 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?

- **Snoopy Bus** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
  - 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

- **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

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?

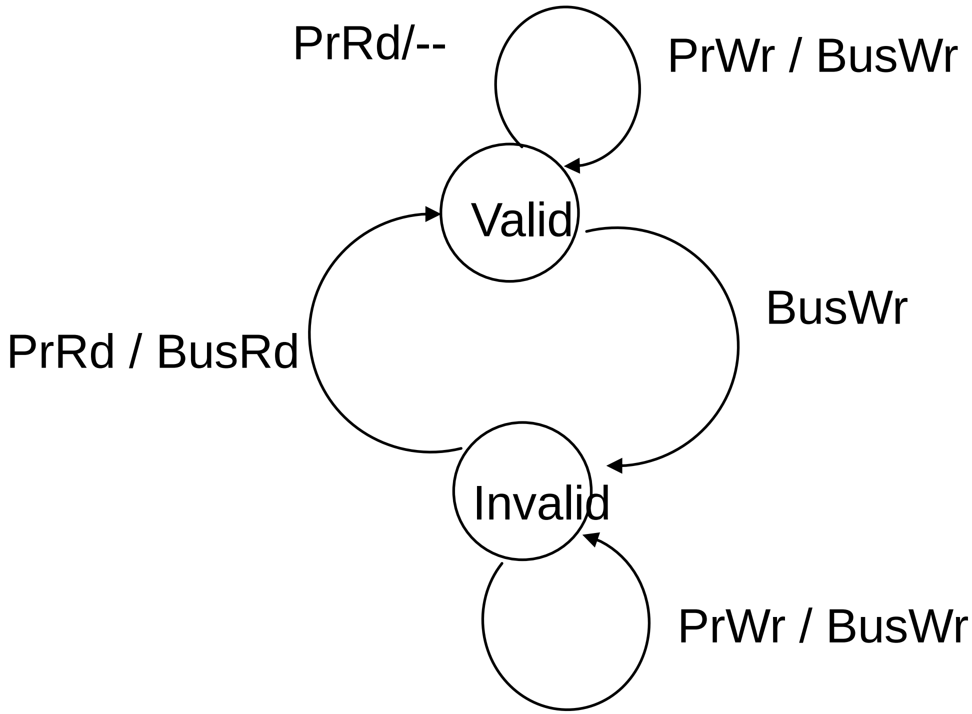
Directory Based Coherence

- Idea: A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.

- An example mechanism:
  - 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
  - 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)

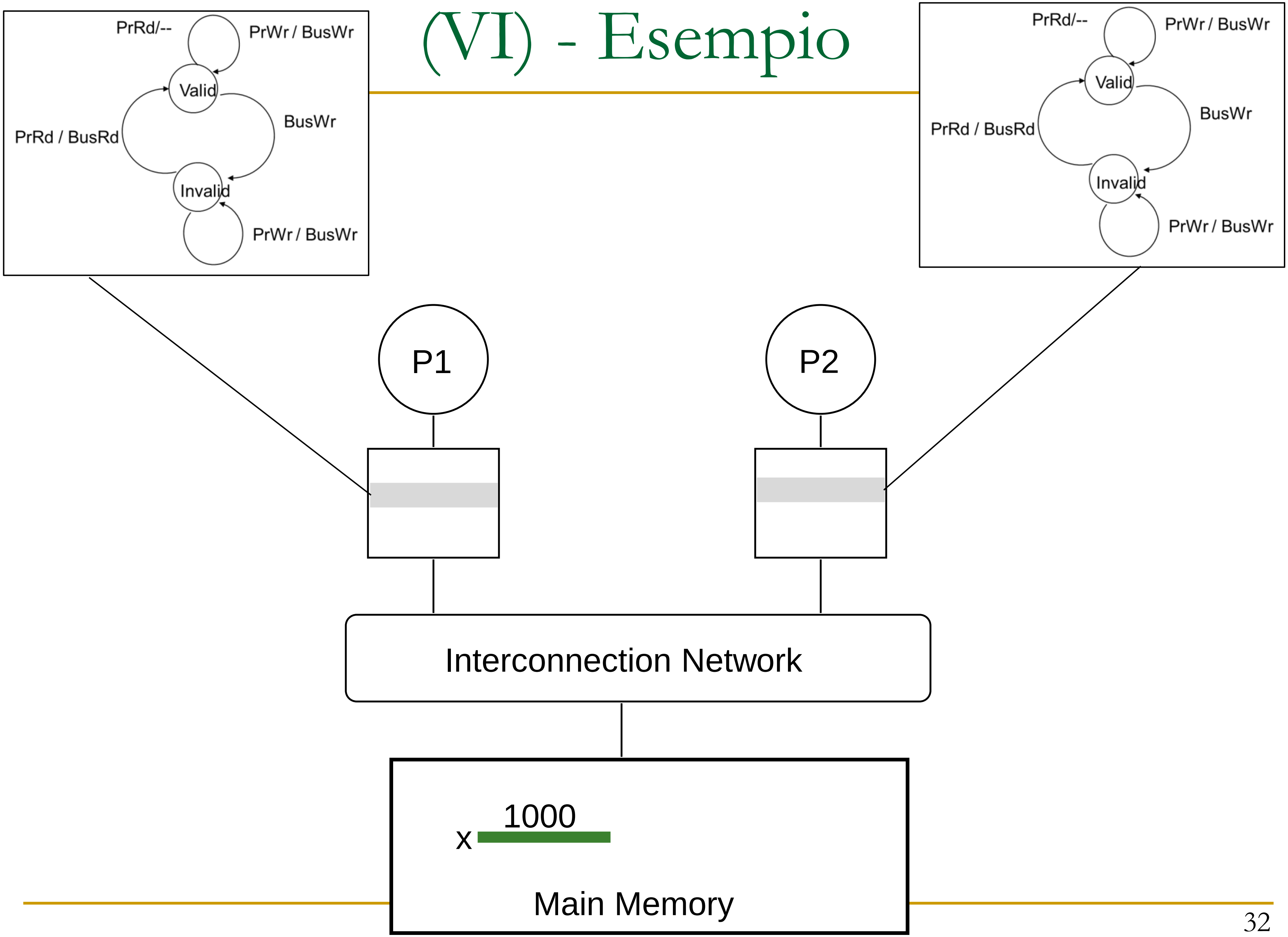
A Very Simple Coherence Scheme (VI)

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

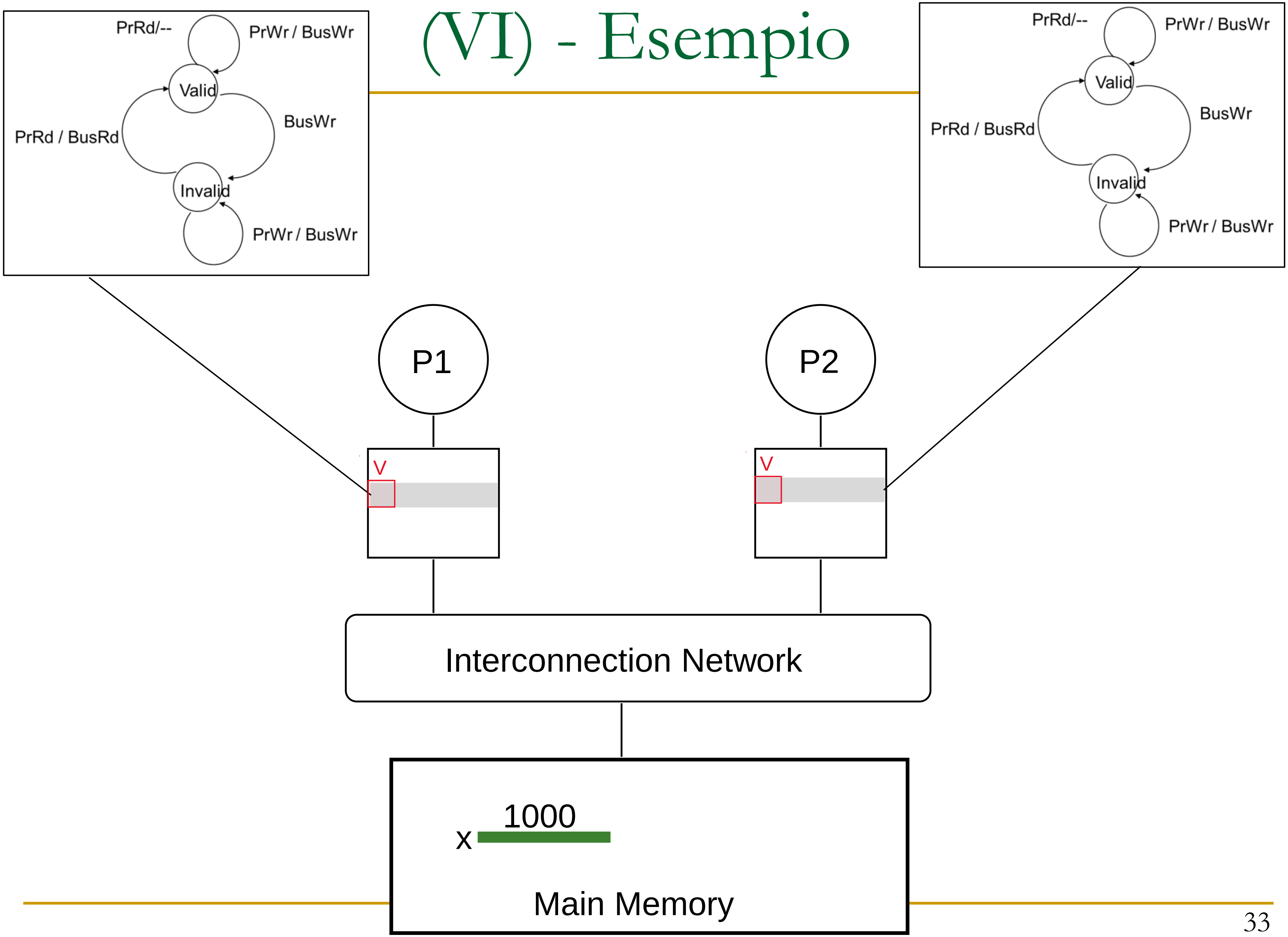


- Write-through, no-write-allocate cache
- Actions of the local processor on the cache block: PrRd, PrWr,
- Actions that are broadcast on the bus for the block: BusRd, BusWr

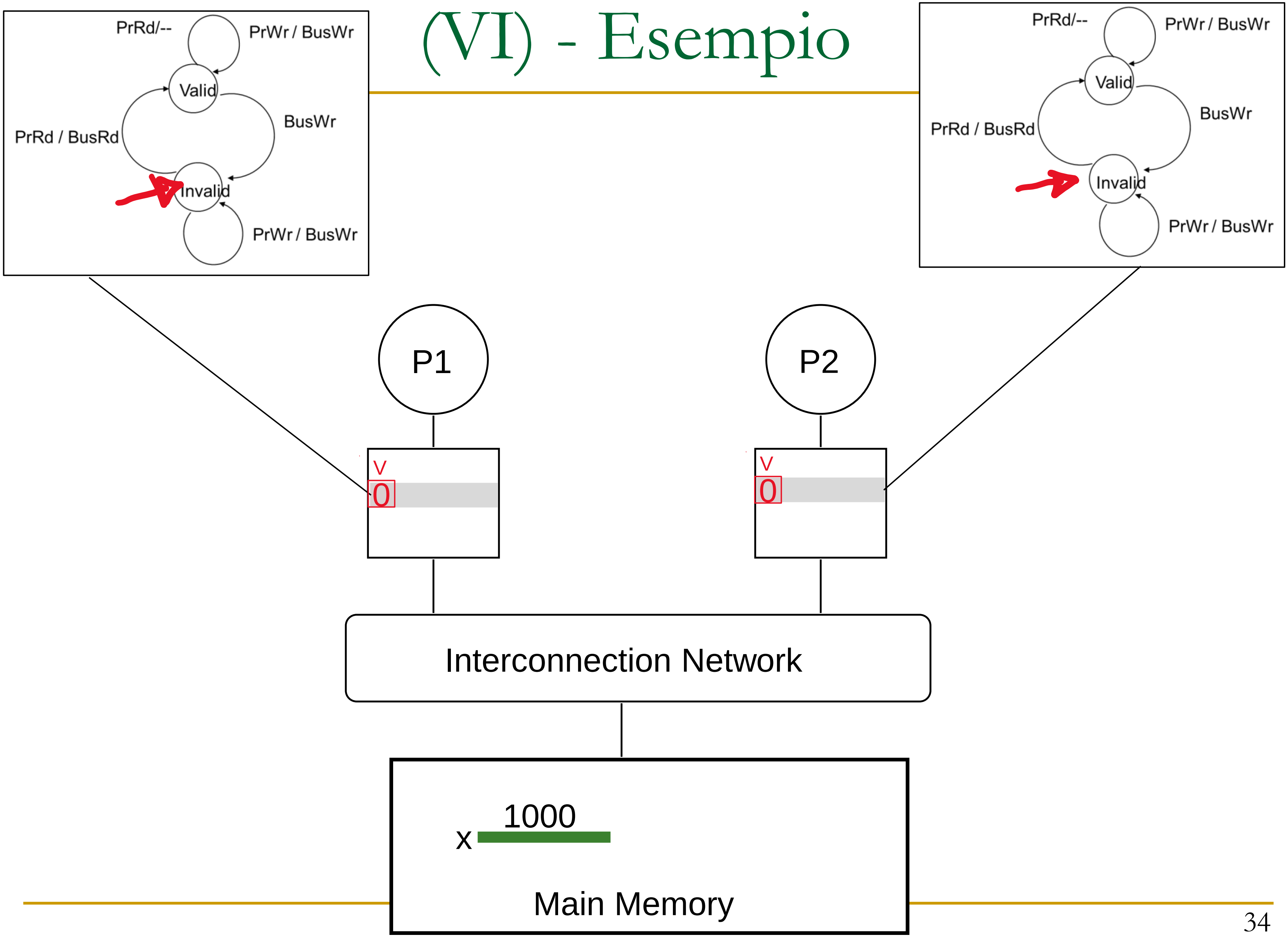
(VI) - Esempio


PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr
PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr
P1
P2
Interconnection Network
1000
x
Main Memory

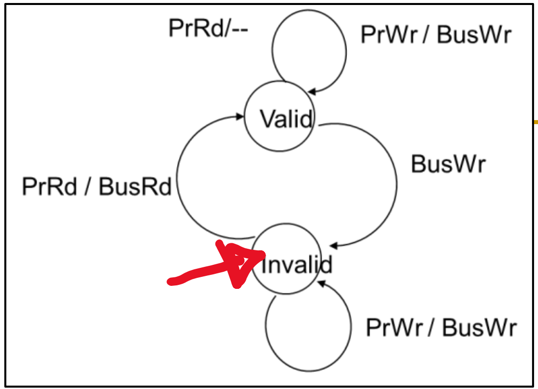
(VI) - Esempio

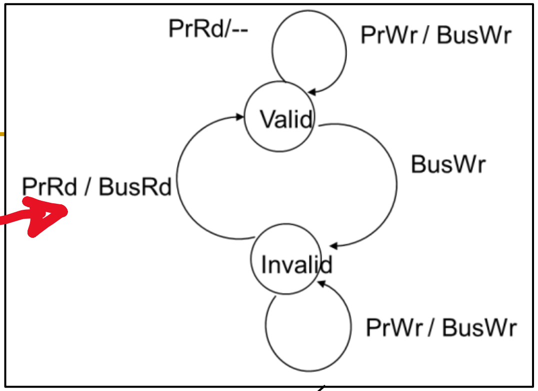

PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr
PrRd/--
PrWr / BusWr
Valid
PrRd / BusRd
BusWr
Invalid
PrWr / BusWr
P1
P2
V
V
Interconnection Network
x
1000
Main Memory

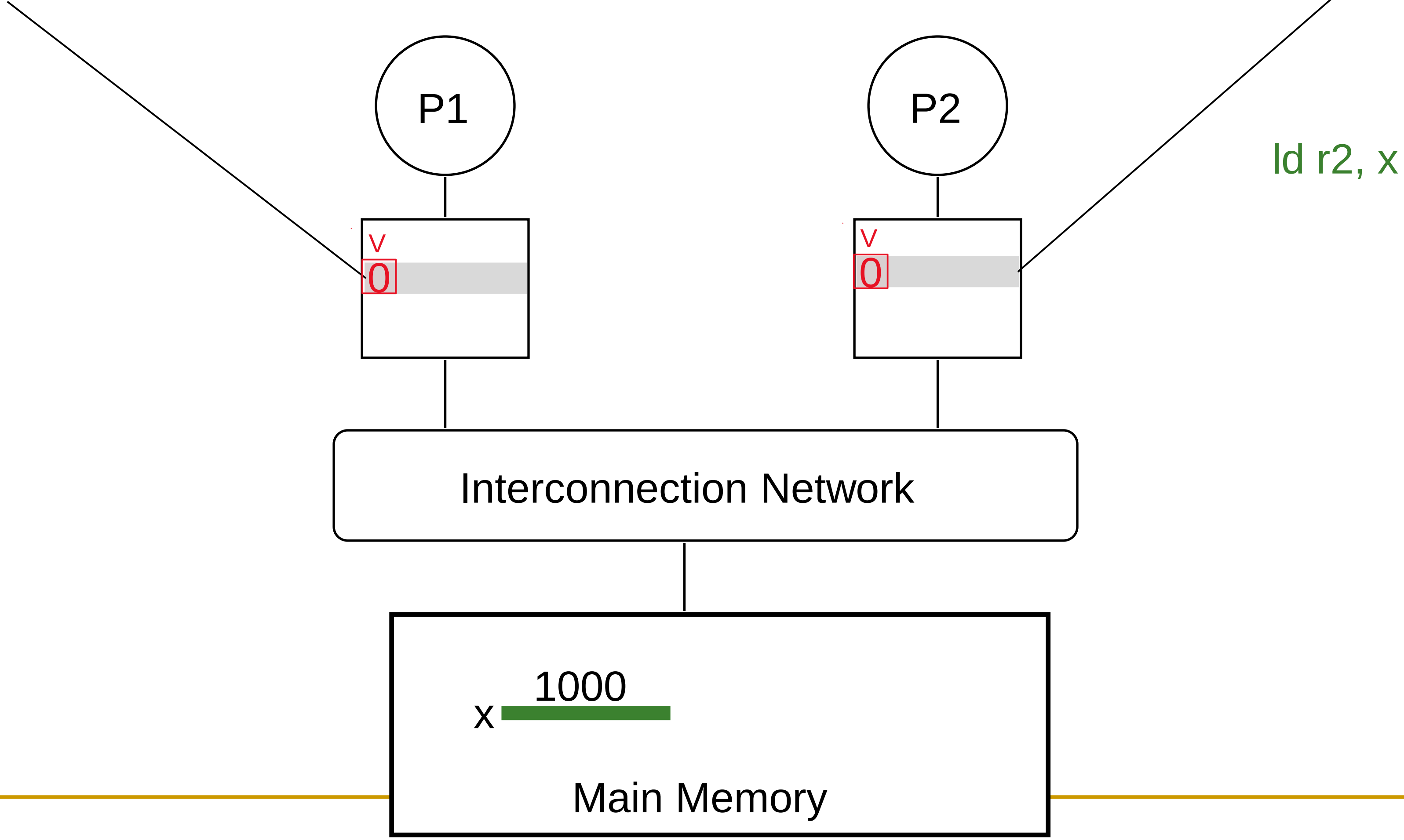
(VI) - Esempio


PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr
PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr
P1
P2
V
0
V
0
Interconnection Network
x
1000
Main Memory

(VI) - Esempio

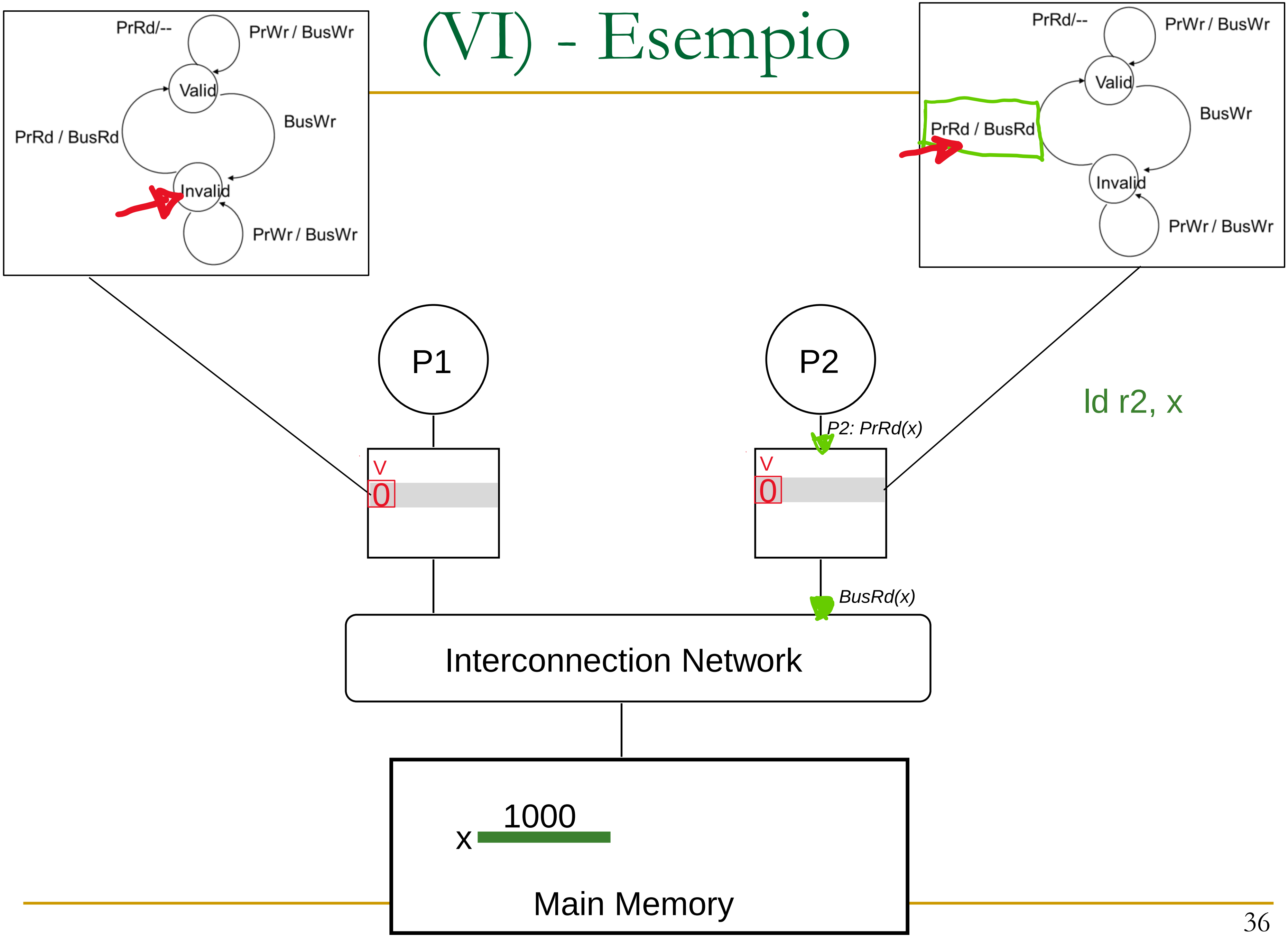






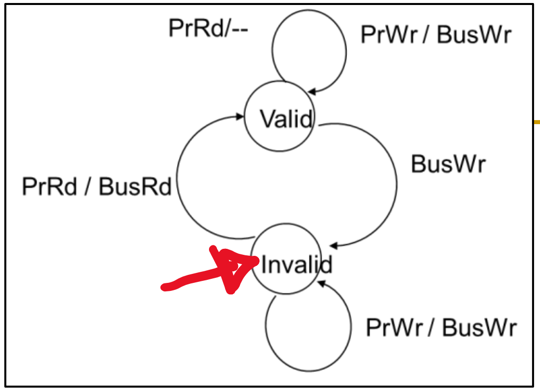
ld r2, x

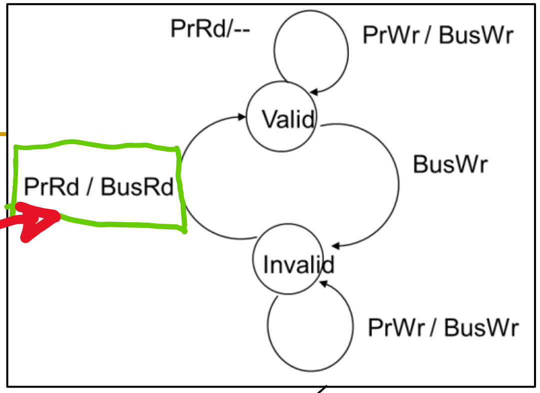
(VI) - Esempio

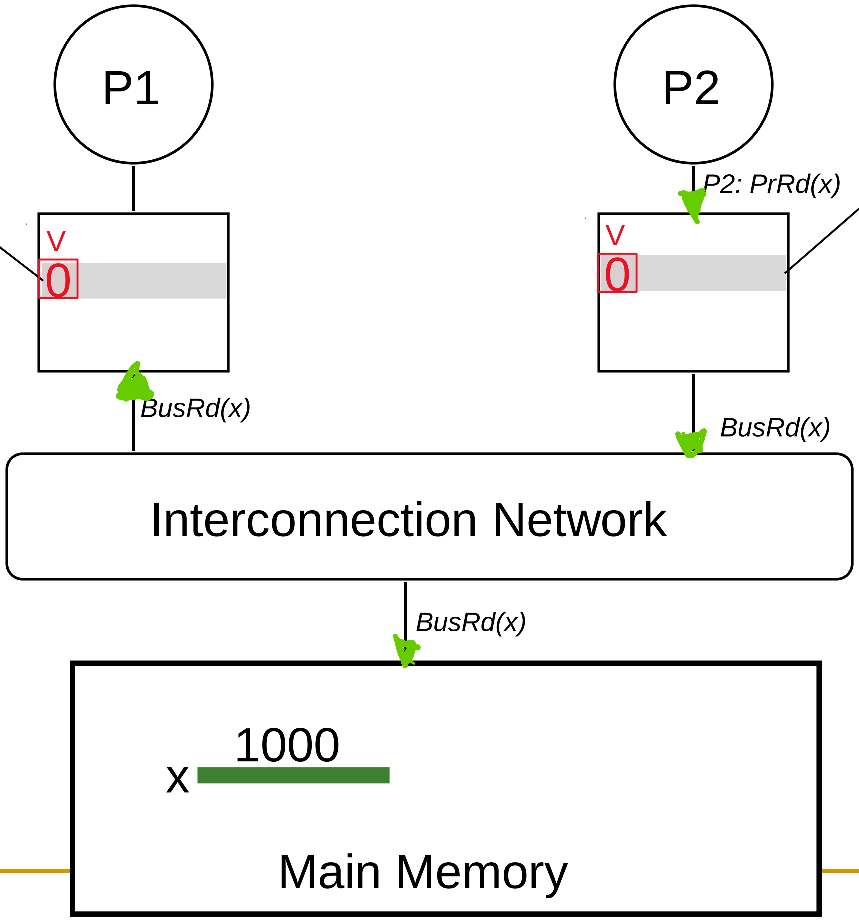


ld r2, x

(VI) - Esempio

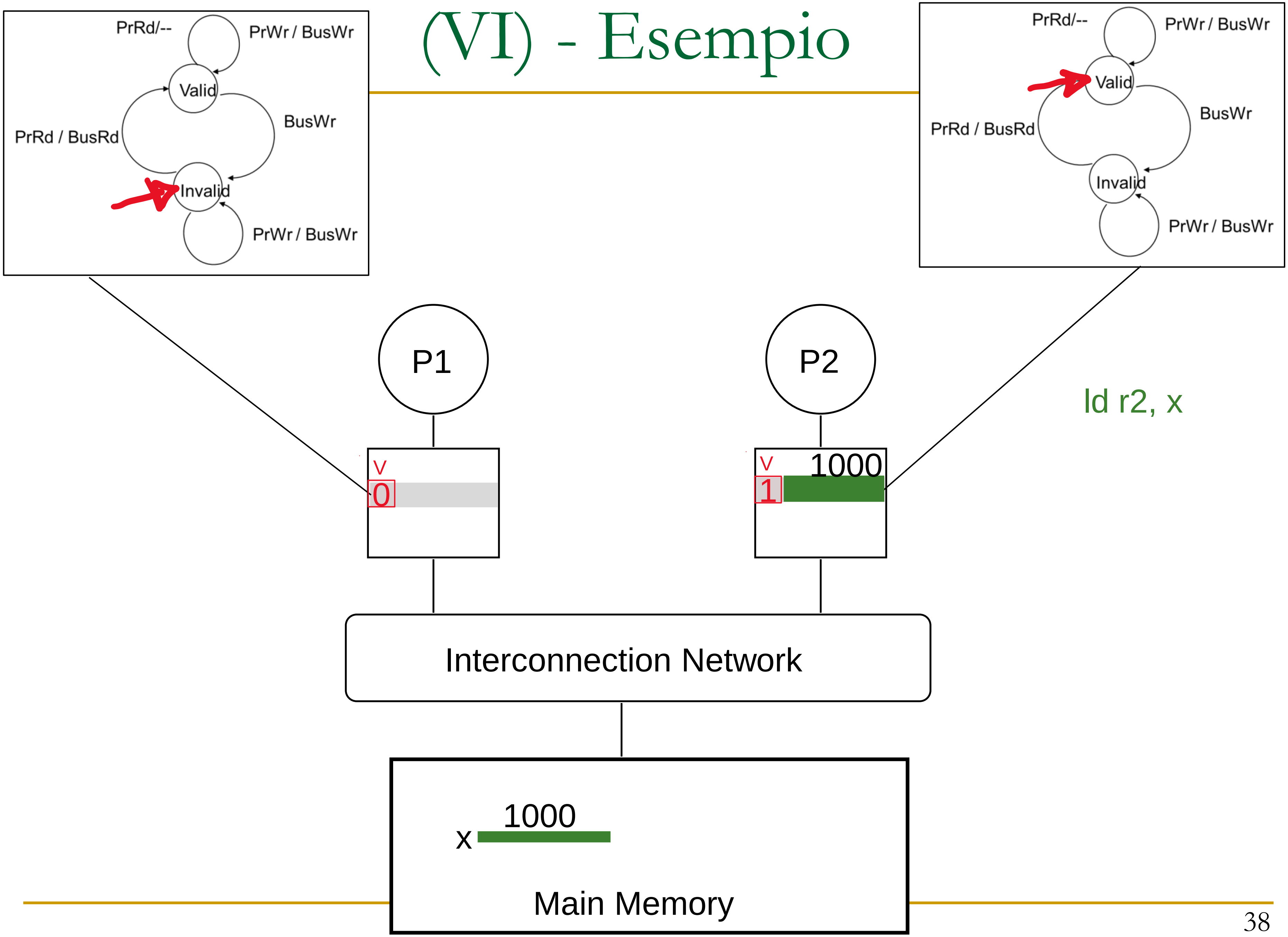






ld r2, x

(VI) - Esempio



PrRd/--

PrWr / BusWr

Valid

BusWr

PrRd / BusRd

Invalid

PrWr / BusWr

PrRd/--

PrWr / BusWr

Valid

BusWr

PrRd / BusRd

Invalid

PrWr / BusWr

P1

P2

ld r2, x

V

0

V

1

1000

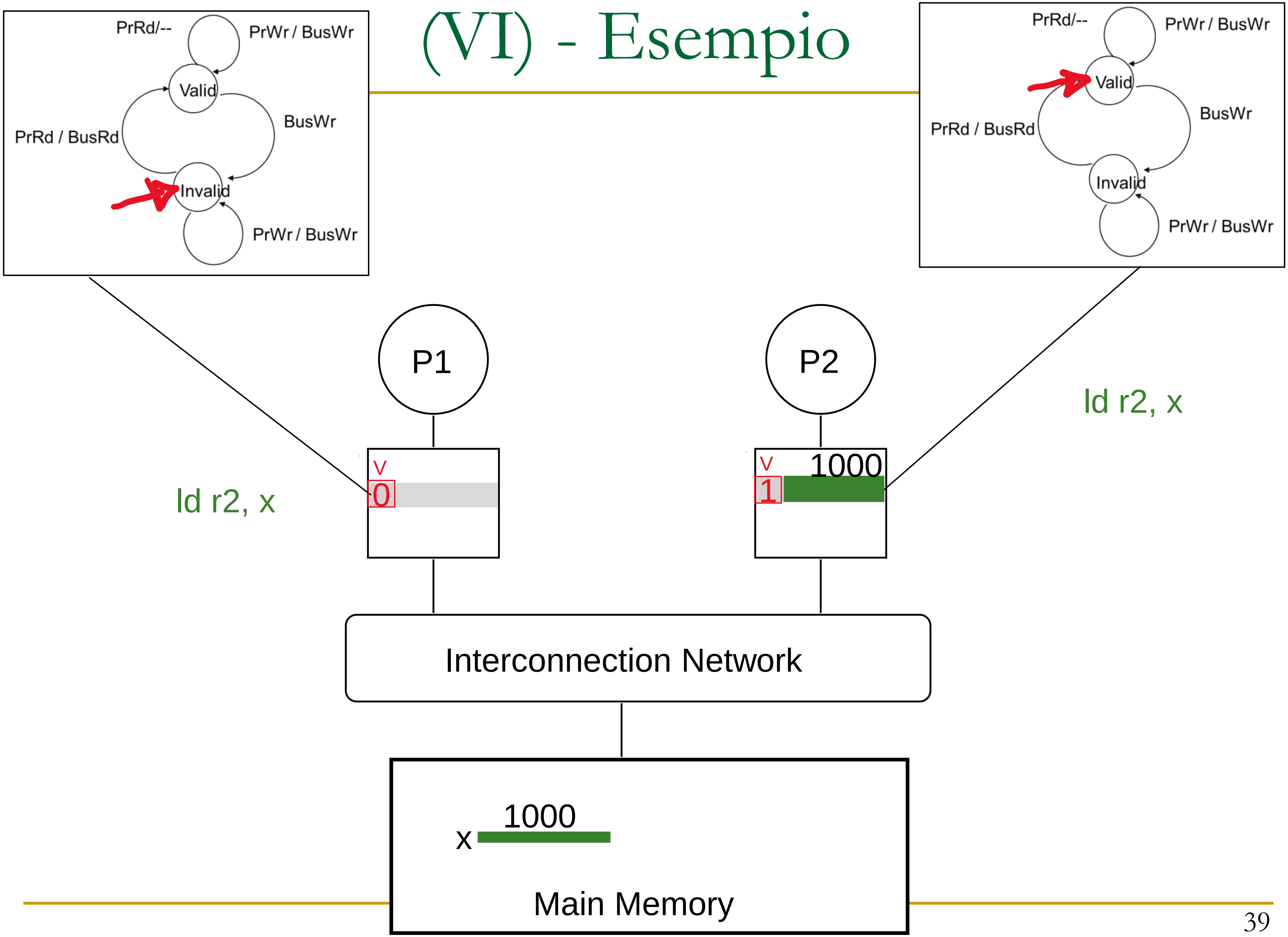
Interconnection Network

x

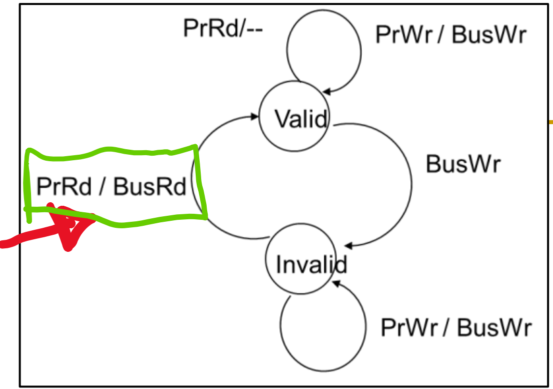
1000

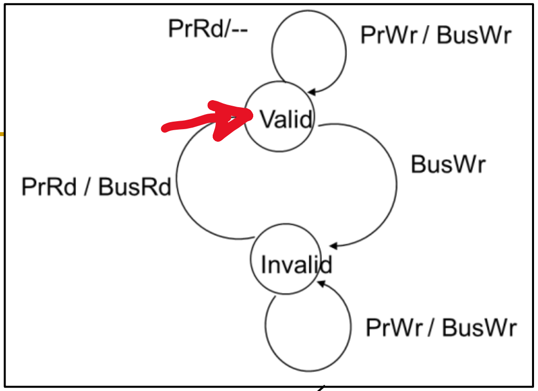
Main Memory

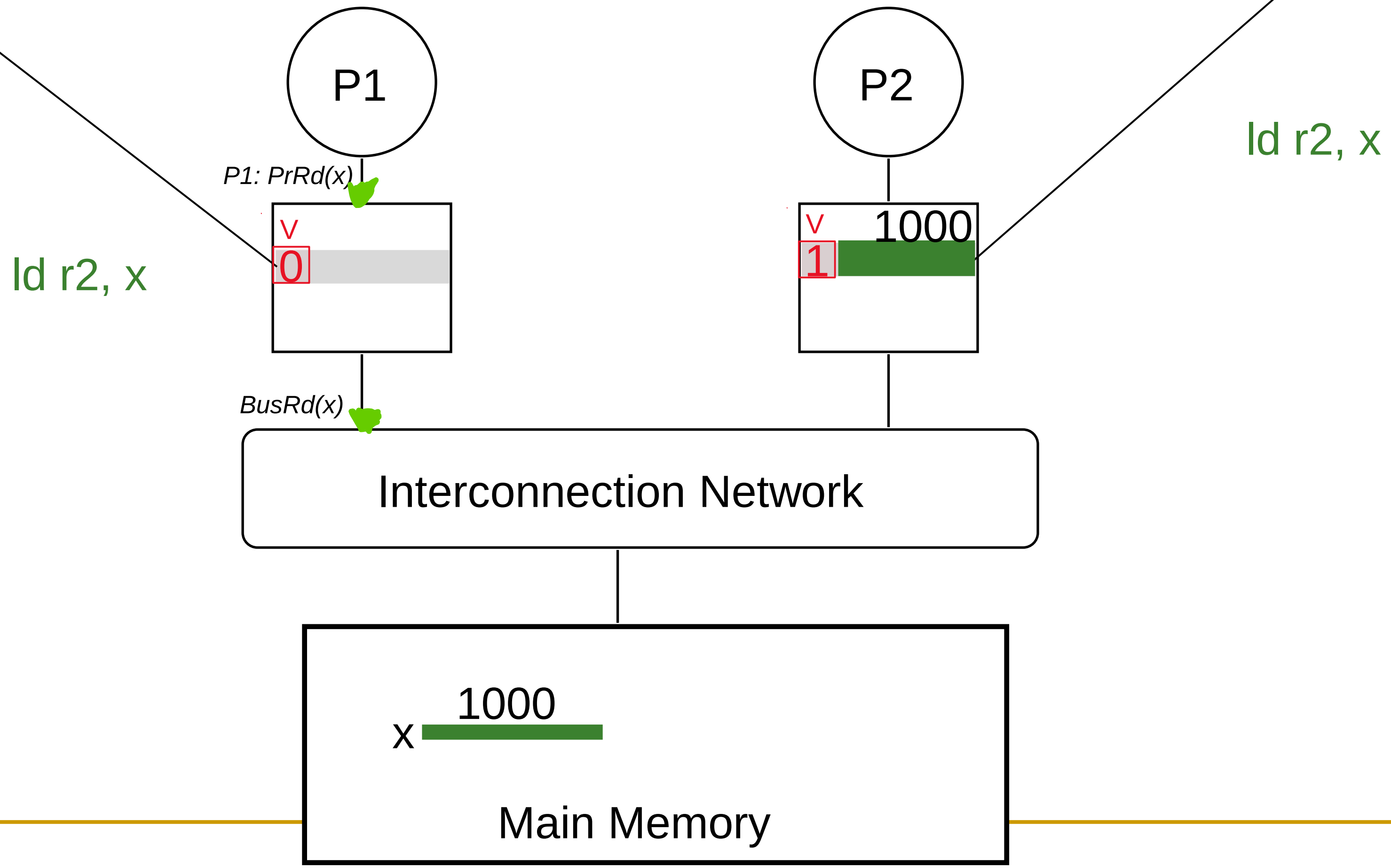
(VI) - Esempio


PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr
PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr
P1
P2
ld r2, x
ld r2, x
V
0
V
1
1000
Interconnection Network
x
1000
Main Memory

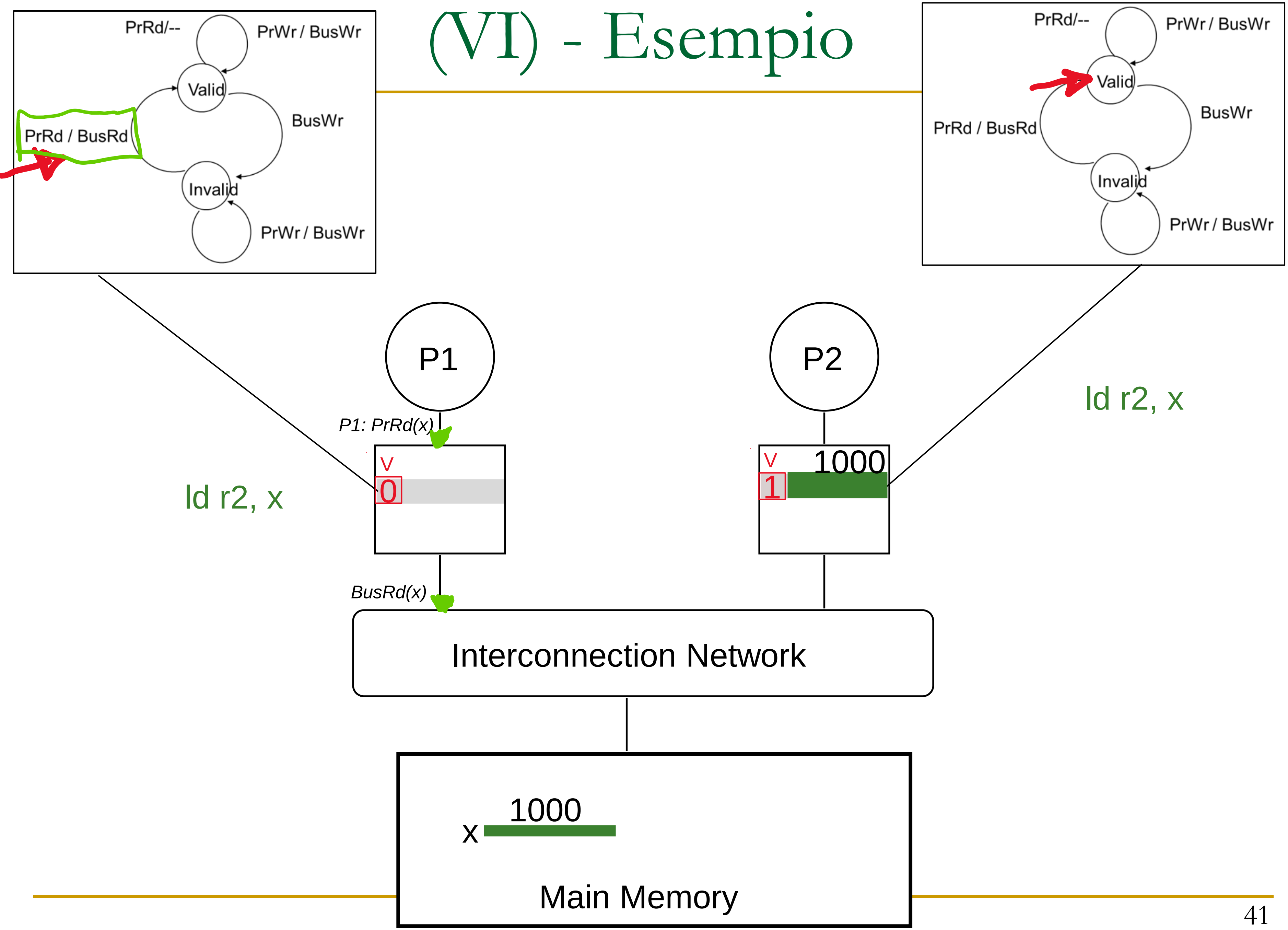
(VI) - Esempio


PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr


PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr


P1
P2
ld r2, x
ld r2, x
P1: PrRd(x)
V
0
V
1
1000
BusRd(x)
Interconnection Network
x
1000
Main Memory

(VI) - Esempio



PrRd/--

PrWr / BusWr

Valid

BusWr

PrRd / BusRd

Invalid

PrWr / BusWr

PrRd/--

PrWr / BusWr

Valid

BusWr

PrRd / BusRd

Invalid

PrWr / BusWr

P1

P2

ld r2, x

P1: PrRd(x)

V

0

ld r2, x

V

1

1000

BusRd(x)

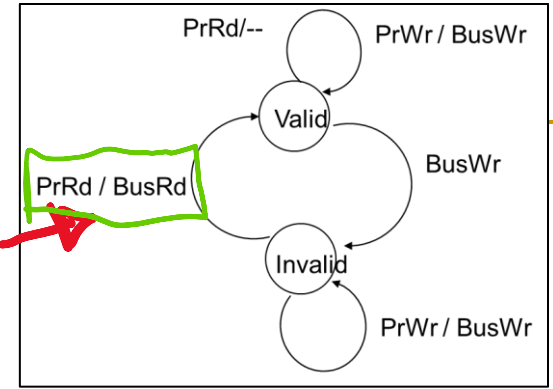
Interconnection Network

x

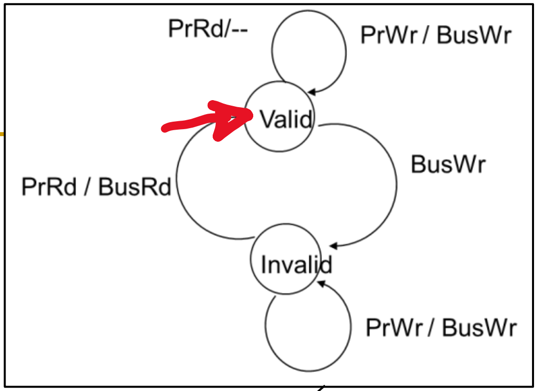
1000

Main Memory

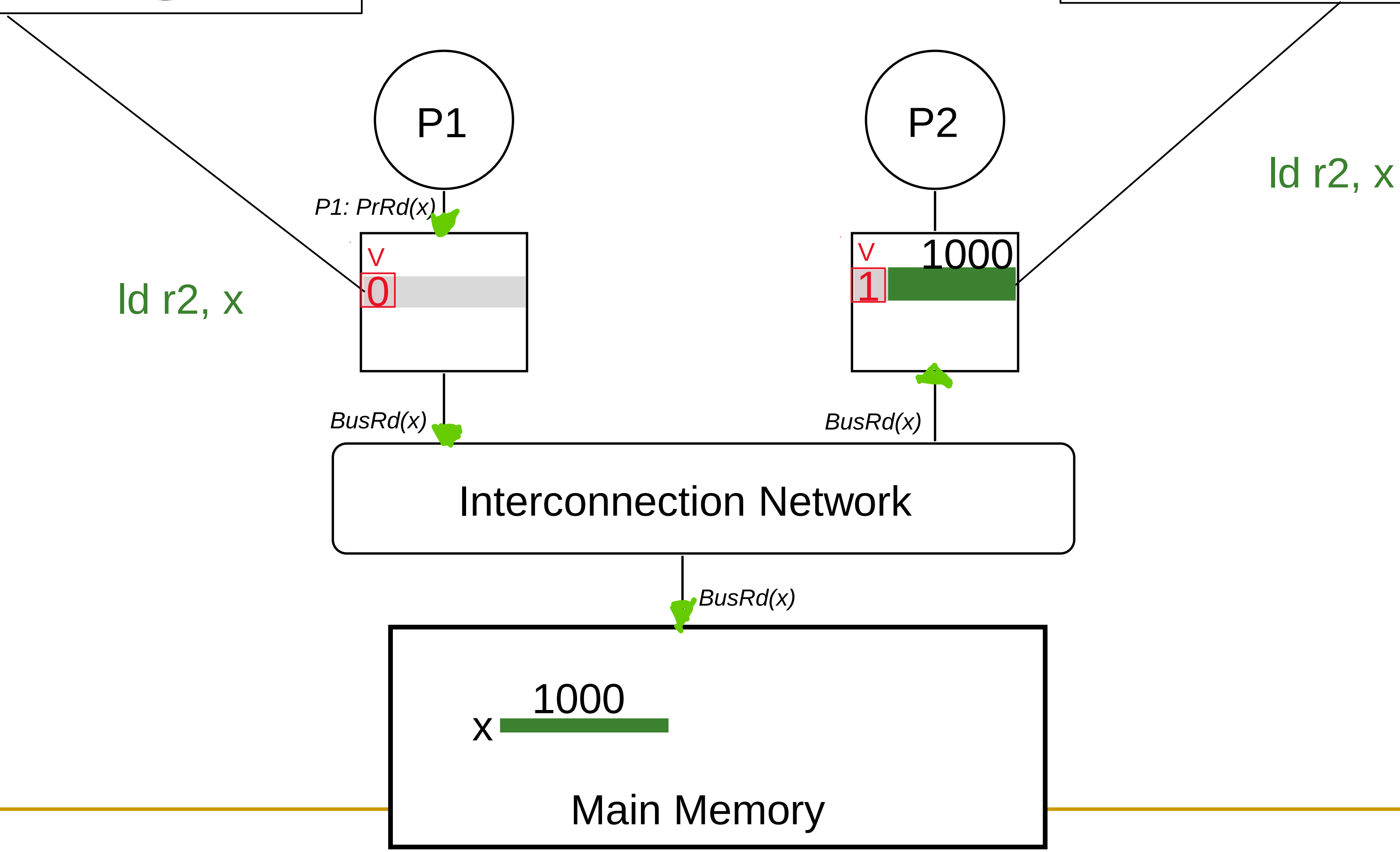
(VI) - Esempio



PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr

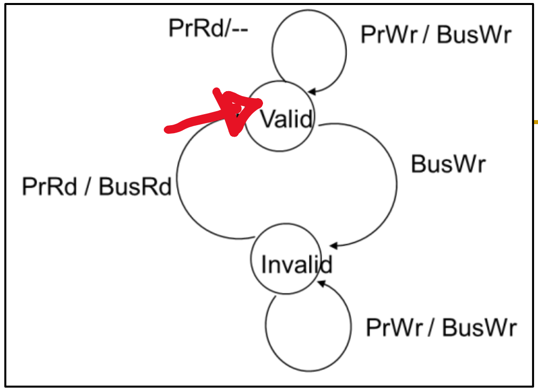


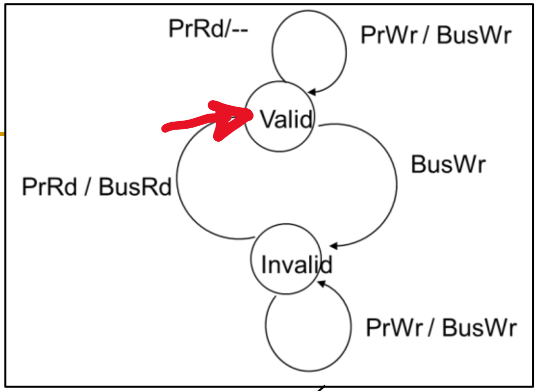
PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr

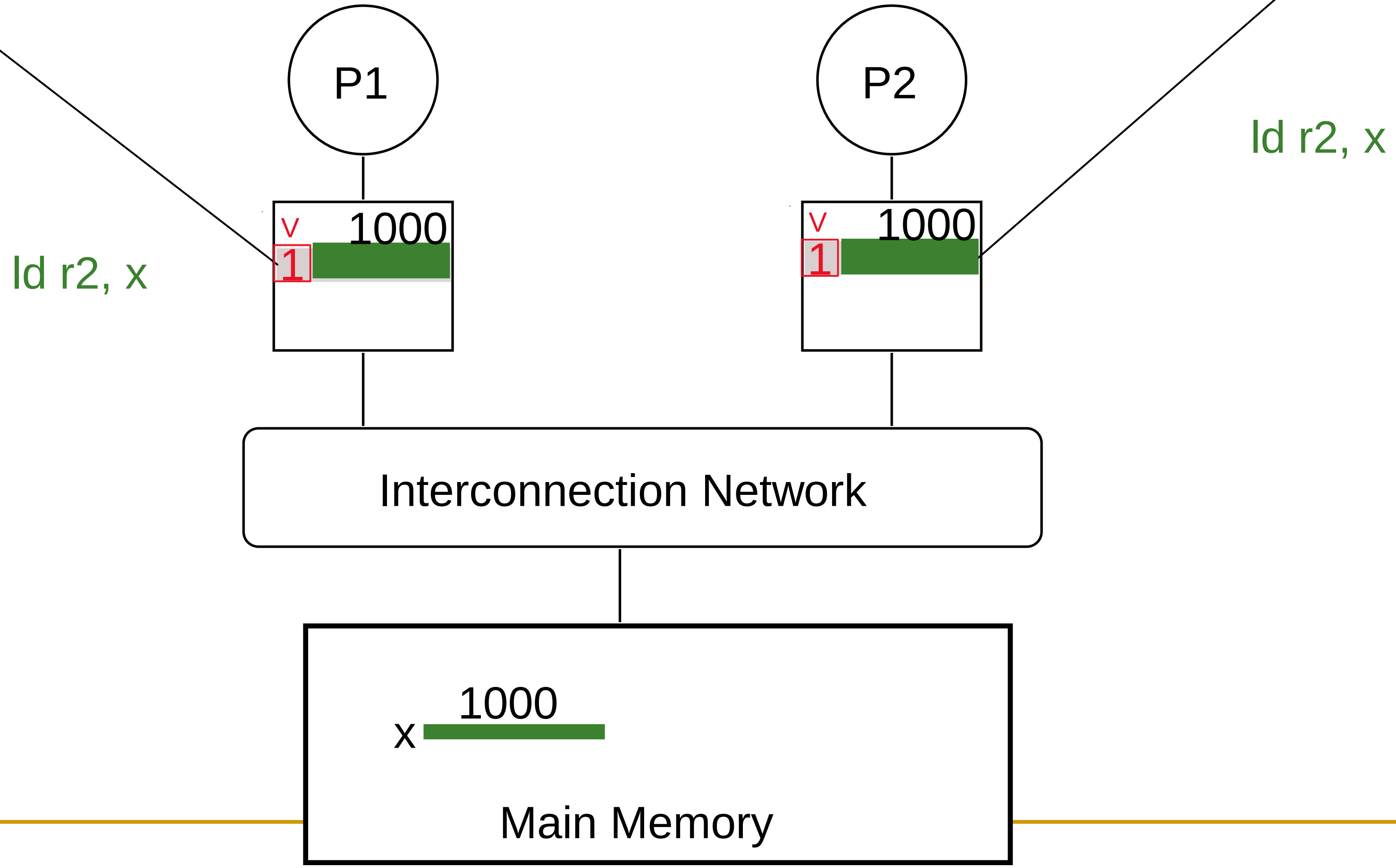


P1
P2
P1: PrRd(x)
ld r2, x
V
0
V
1
1000
ld r2, x
BusRd(x)
BusRd(x)
Interconnection Network
BusRd(x)
x
1000
Main Memory

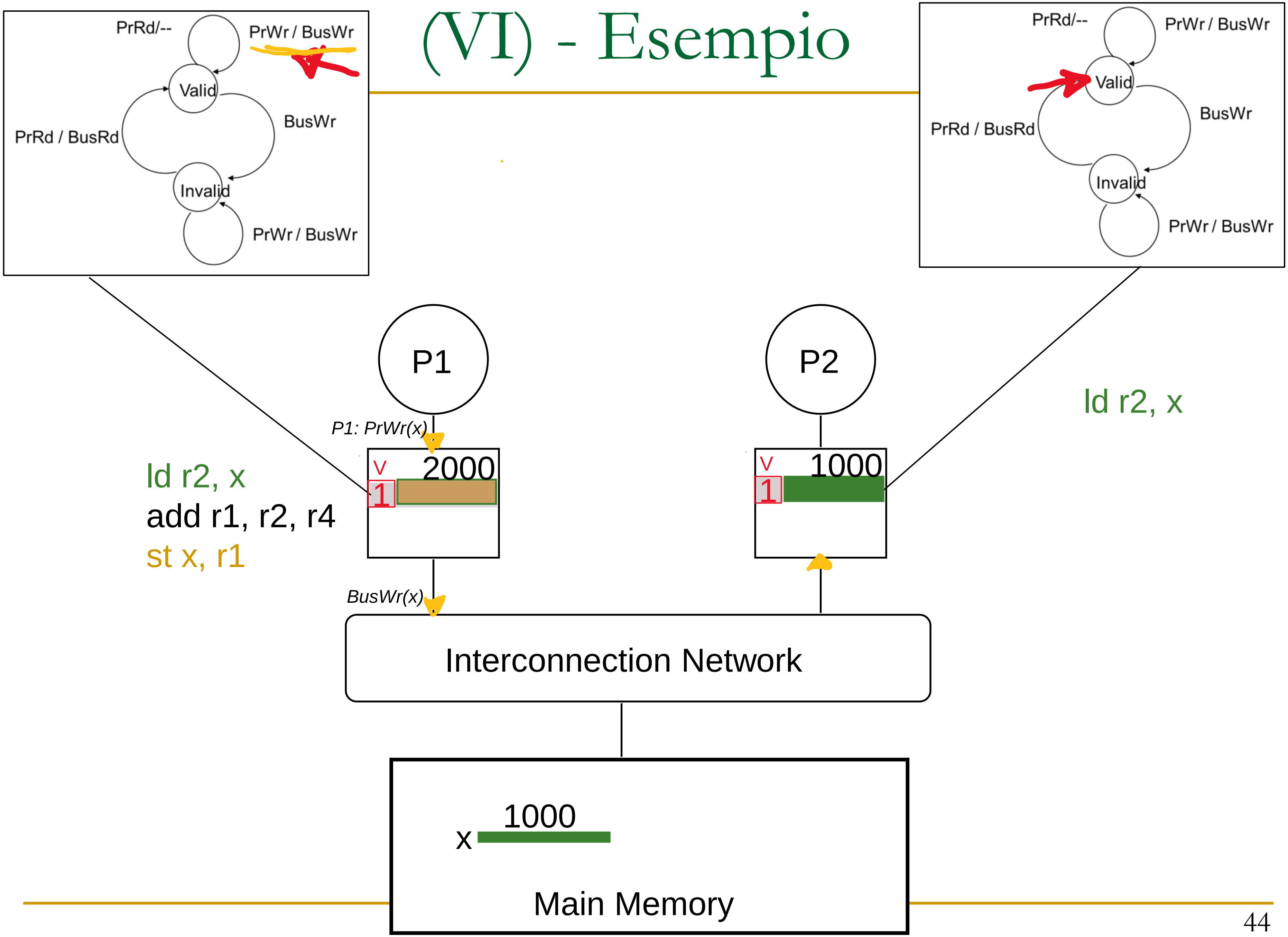
(VI) - Esempio


PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr


PrRd/--
PrWr / BusWr
Valid
BusWr
PrRd / BusRd
Invalid
PrWr / BusWr


P1
P2
ld r2, x
ld r2, x
V
1
1000
V
1
1000
Interconnection Network
x
1000
Main Memory

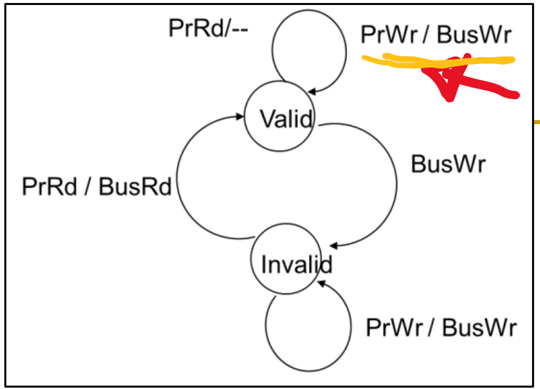
(VI) - Esempio

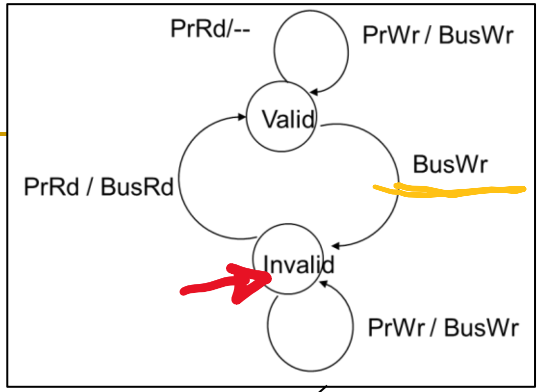


ld r2, x
add r1, r2, r4
st x, r1

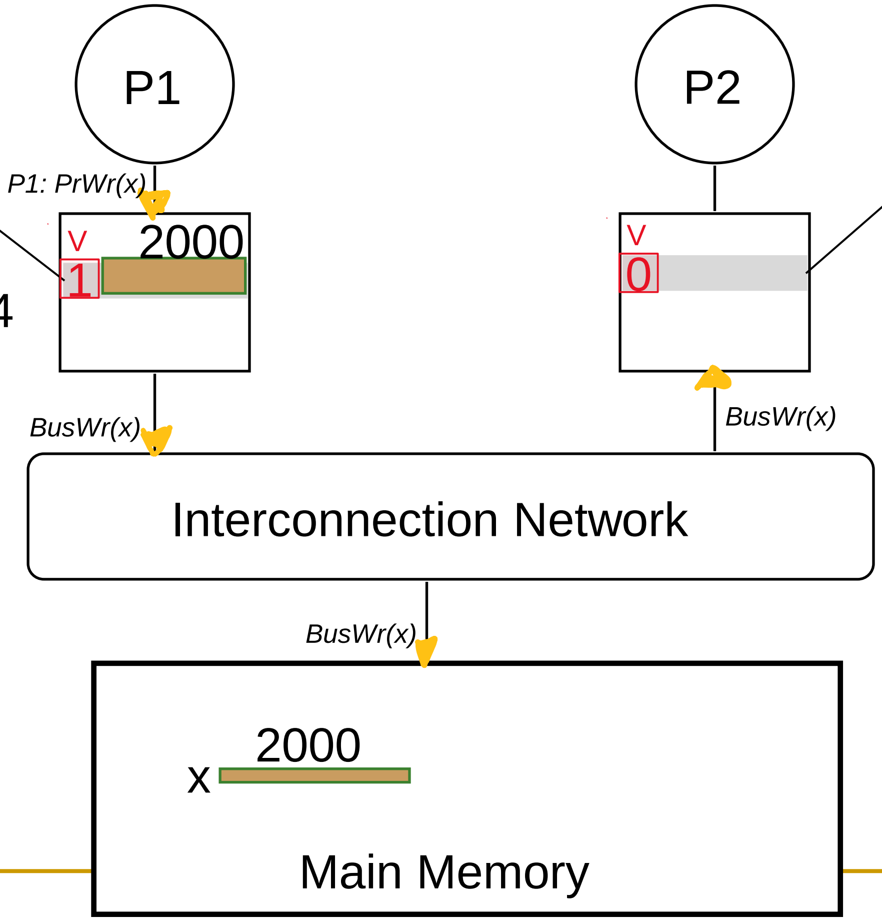
ld r2, x

(VI) - Esempio



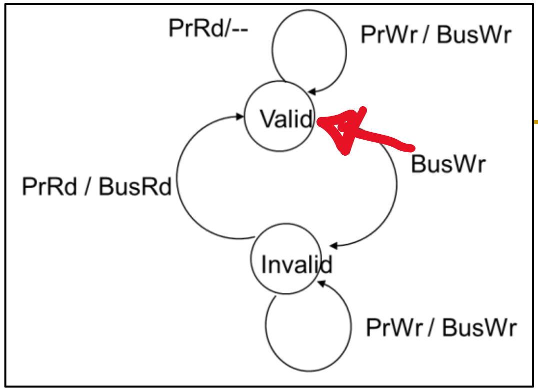


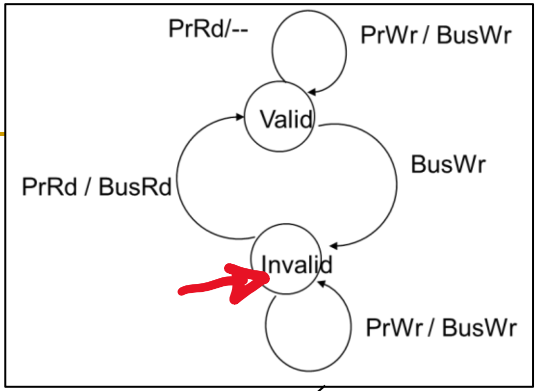
ld r2, x
add r1, r2, r4
st x, r1



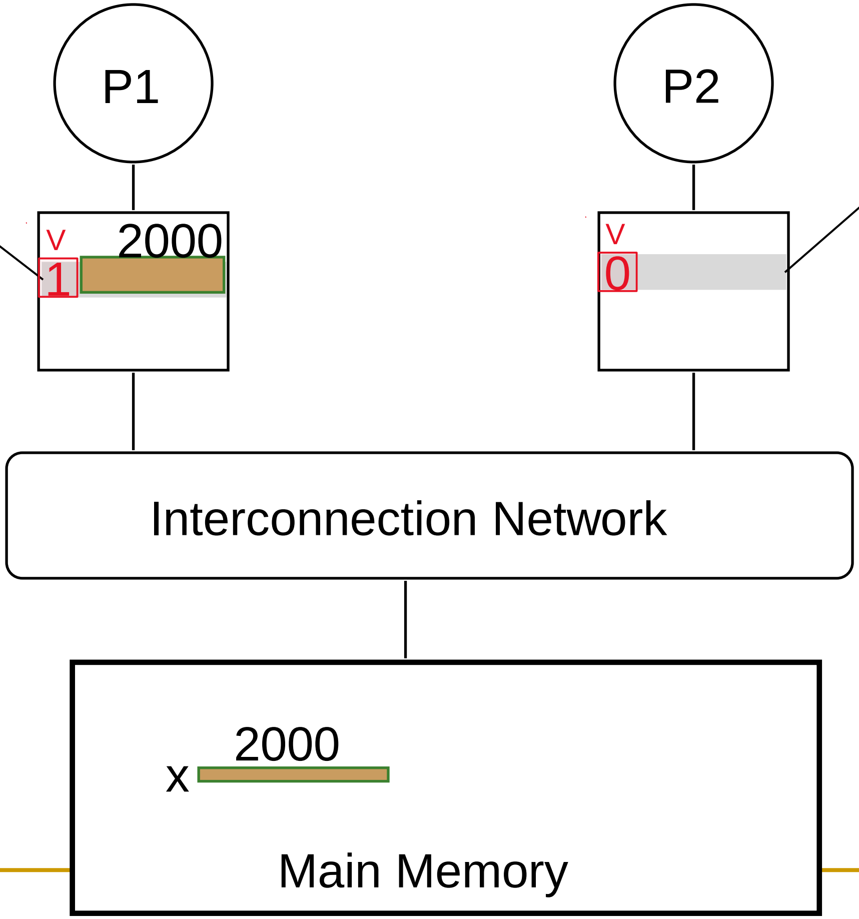
ld r2, x

(VI) - Esempio



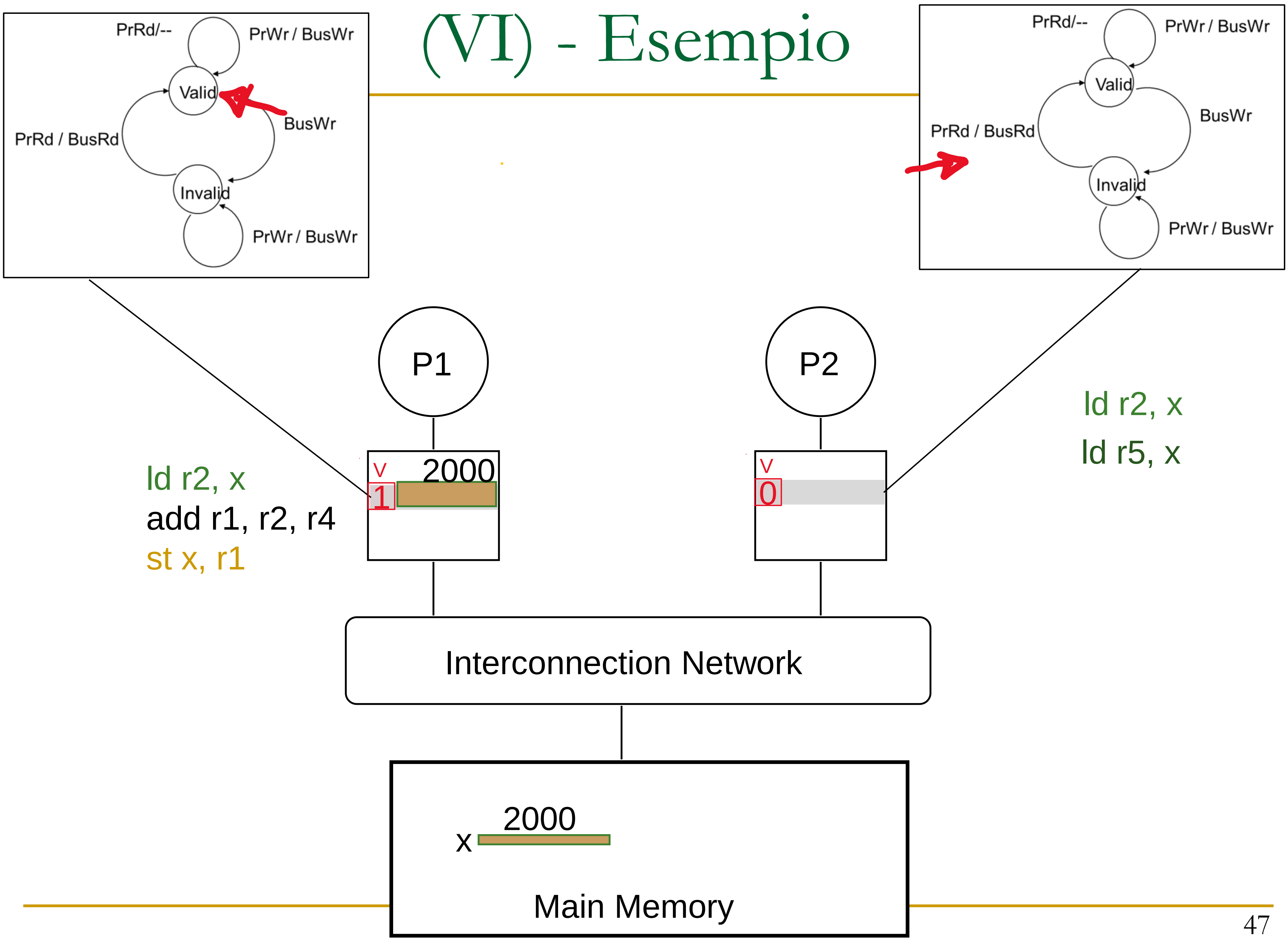


ld r2, x



ld r2, x
add r1, r2, r4
st x, r1

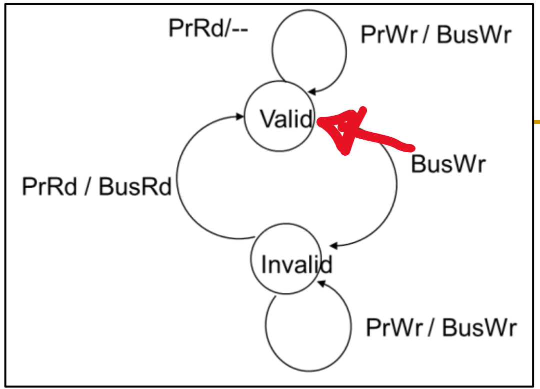
(VI) - Esempio

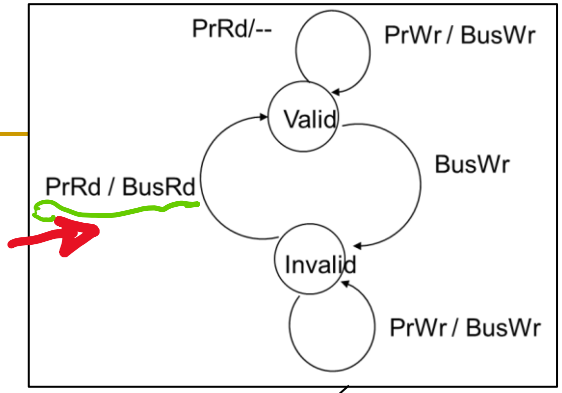


ld r2, x
add r1, r2, r4
st x, r1

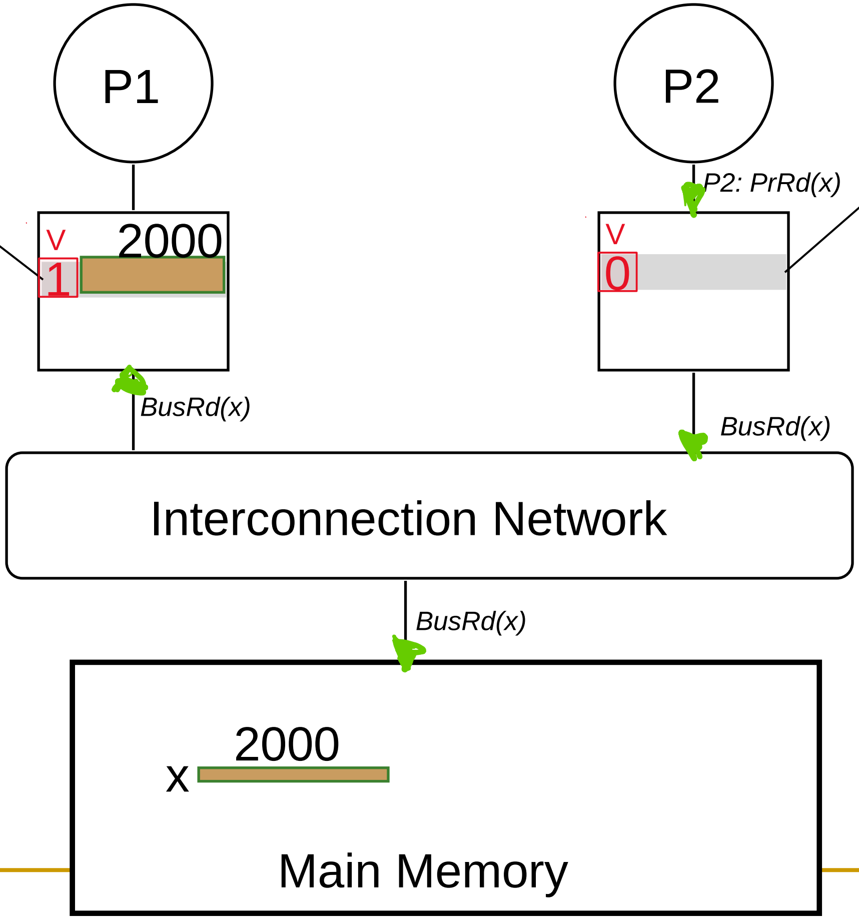
ld r2, x
ld r5, x

(VI) - Esempio



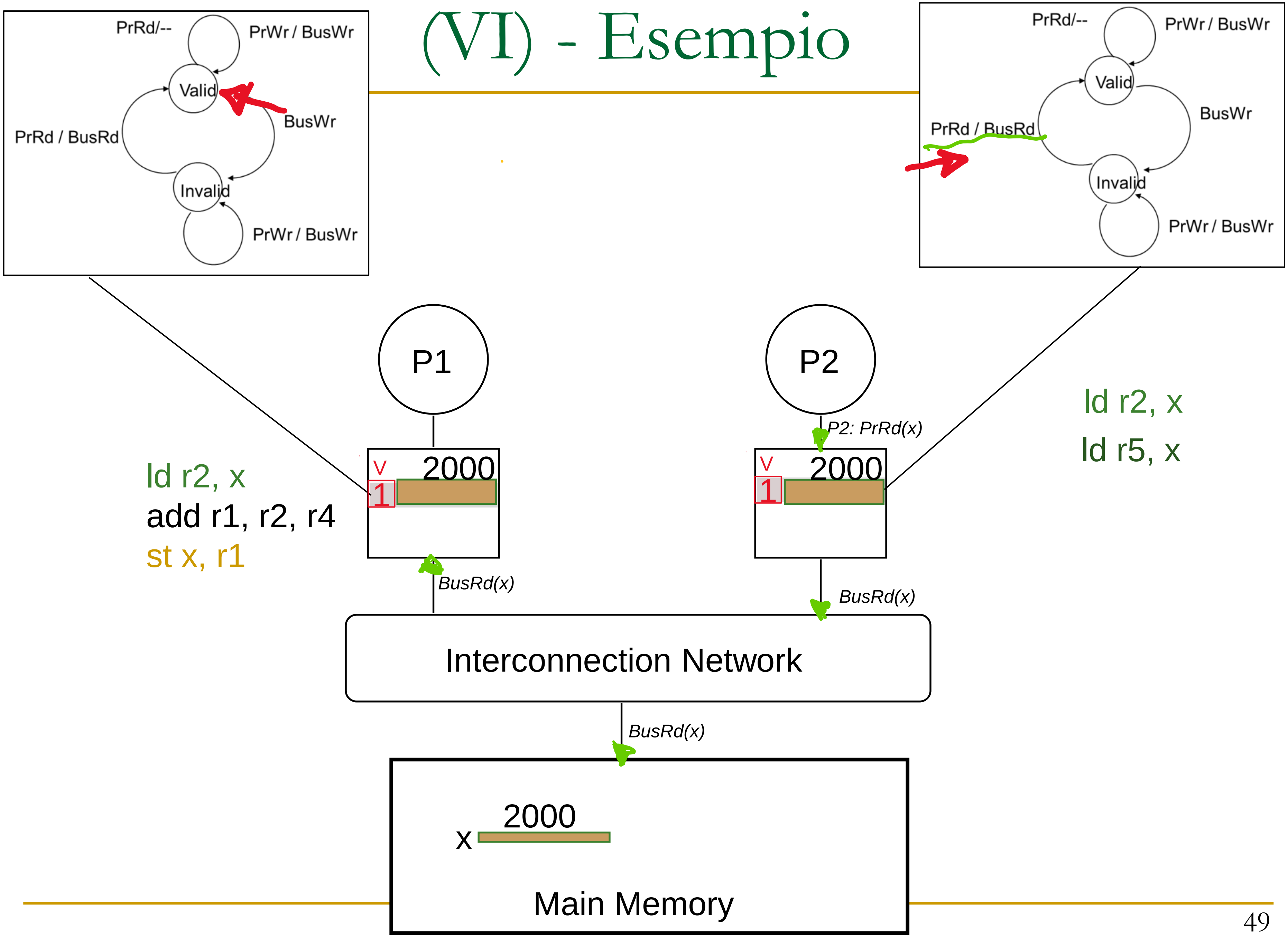


ld r2, x
add r1, r2, r4
st x, r1



ld r2, x
ld r5, x

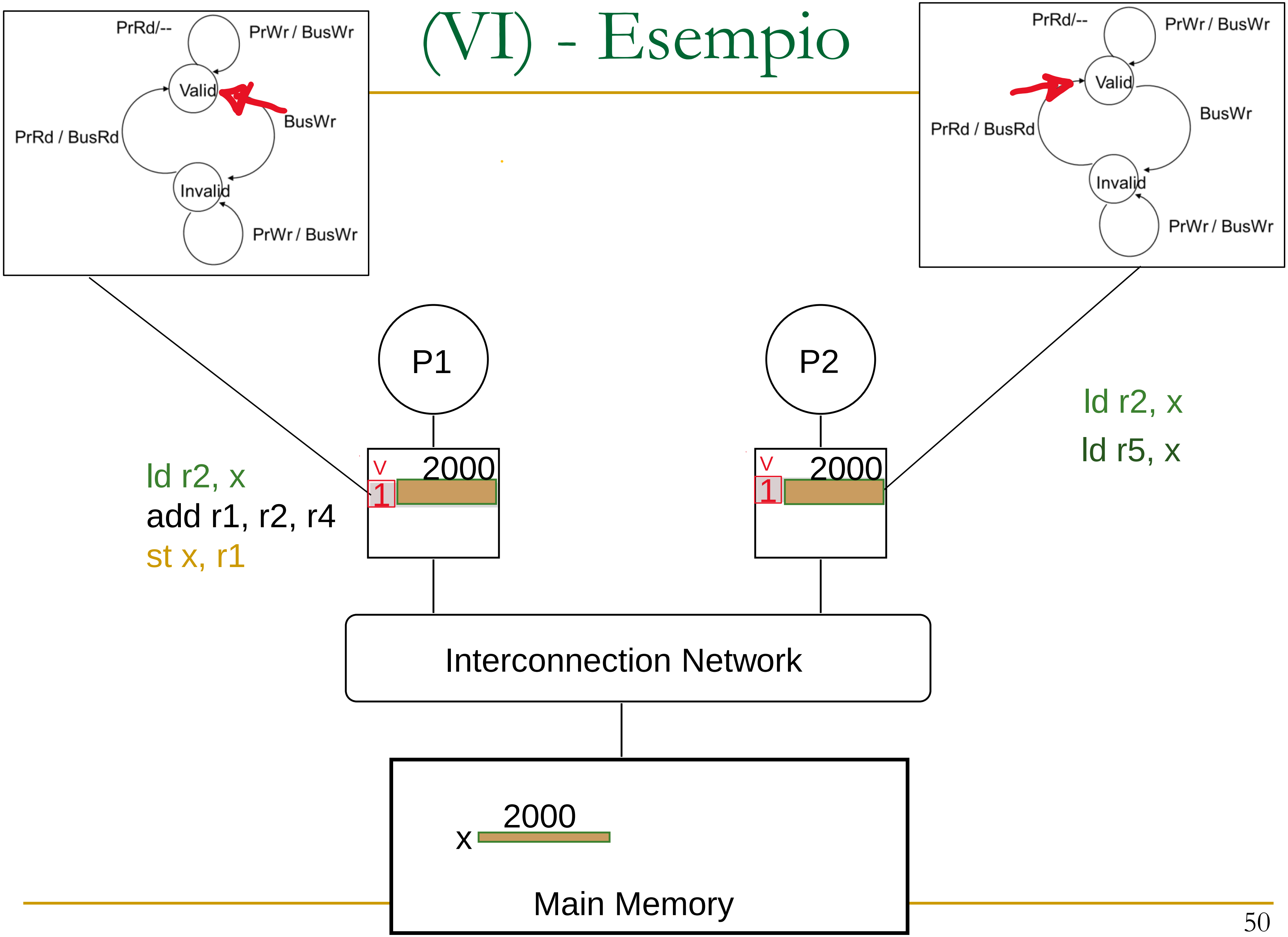
(VI) - Esempio



```
ld r2, x
add r1, r2, r4
st x, r1
```

```
ld r2, x
ld r5, x
```

(VI) - Esempio



ld r2, x
add r1, r2, r4
st x, r1

ld r2, x
ld r5, x

Extending the Protocol

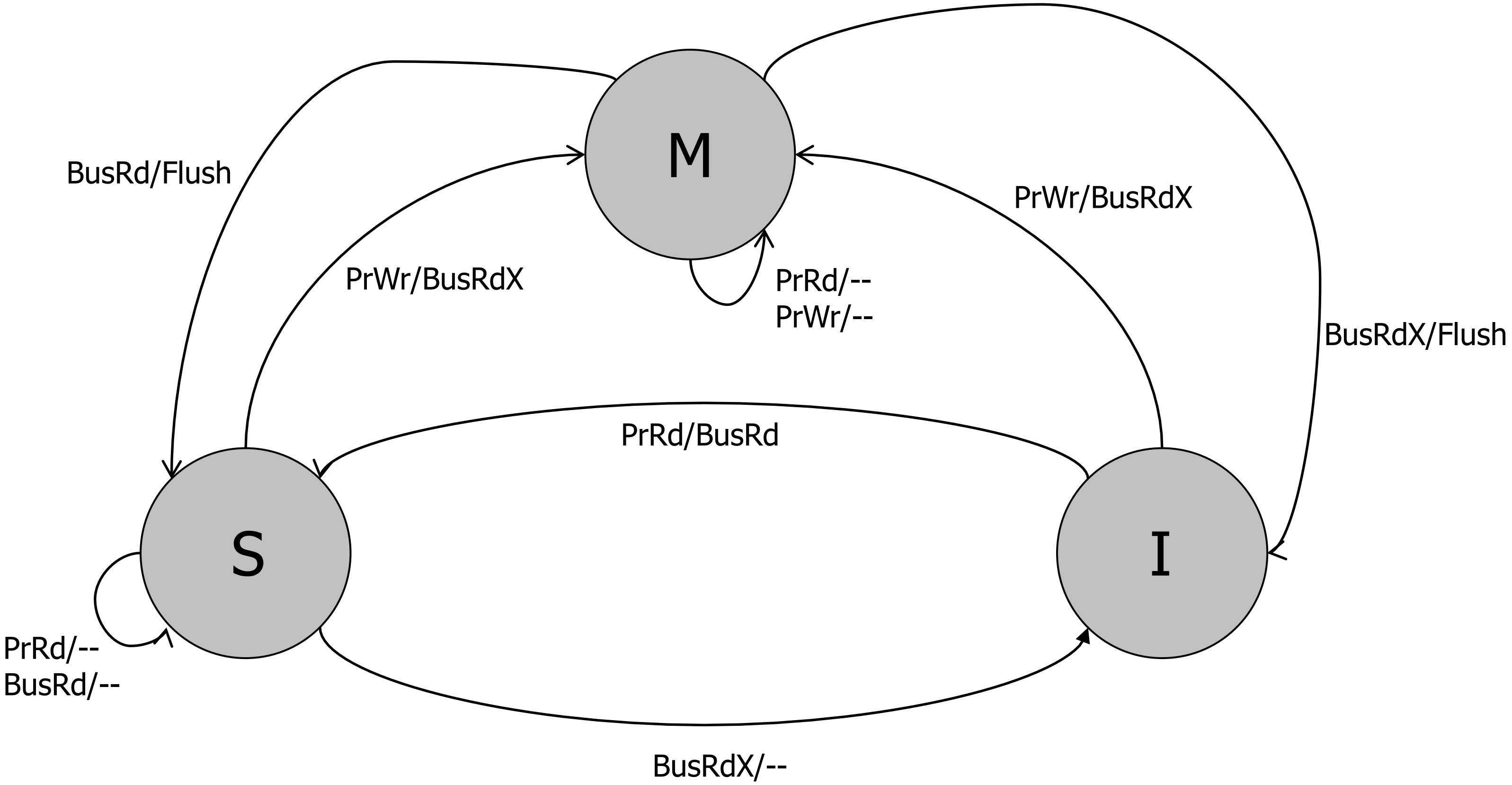
- What if you want write-back caches?
  - We want a “modified” state

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

- 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)*

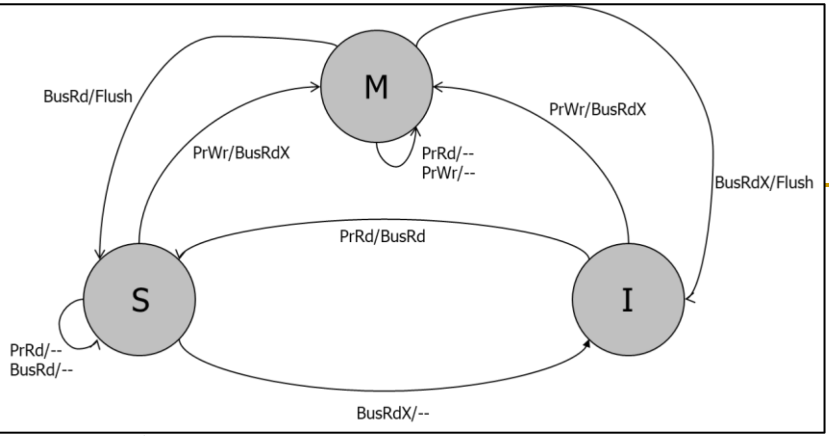
MSI State Machine

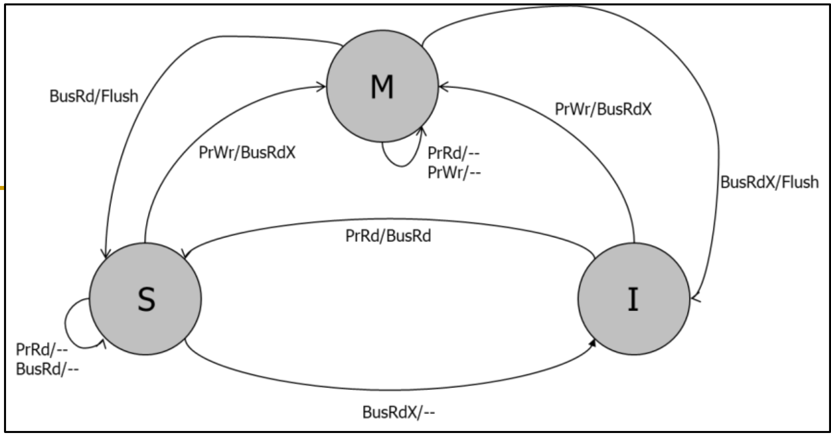


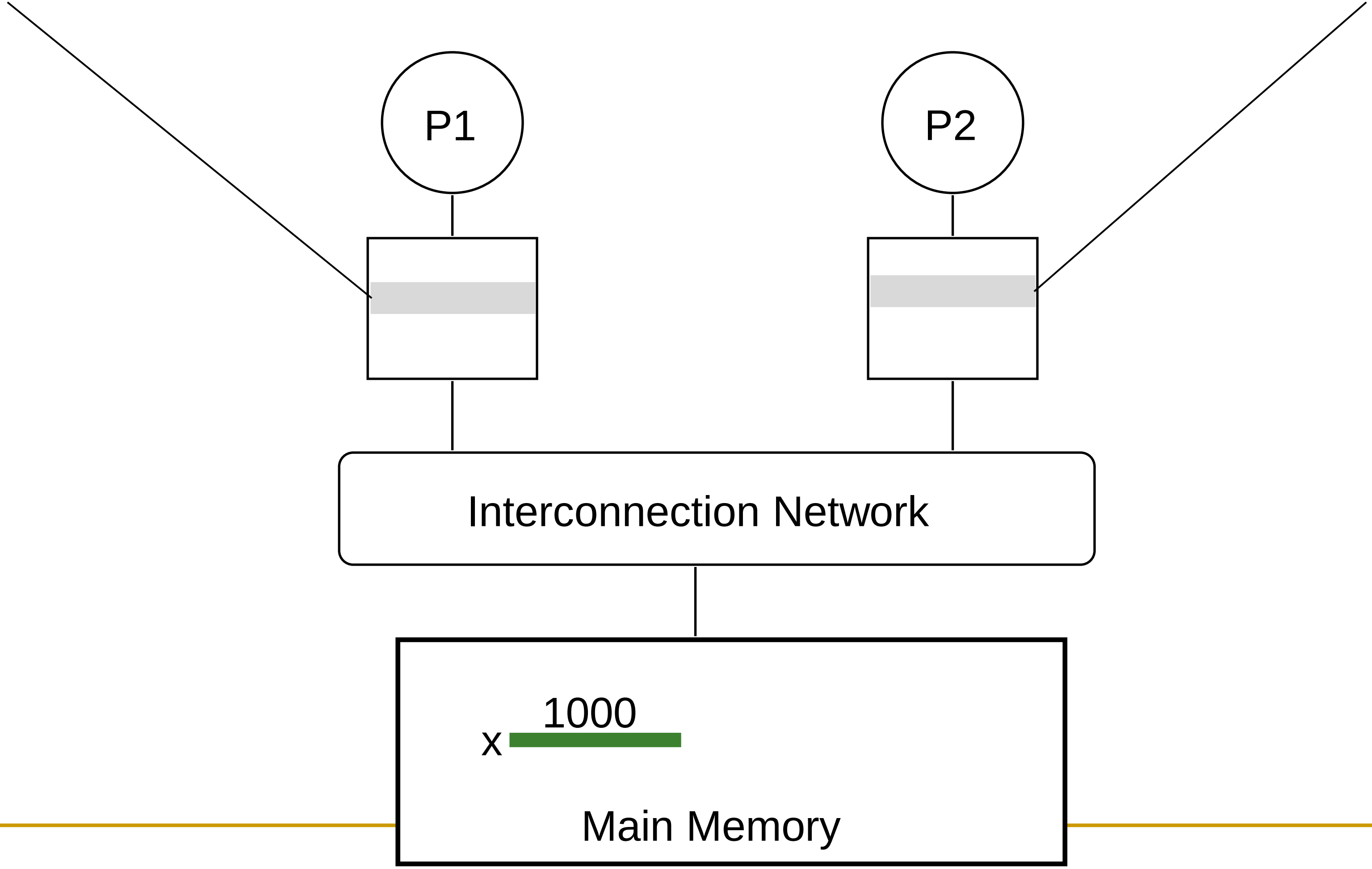
ObservedEvent/Action

[Culler/Singh96]

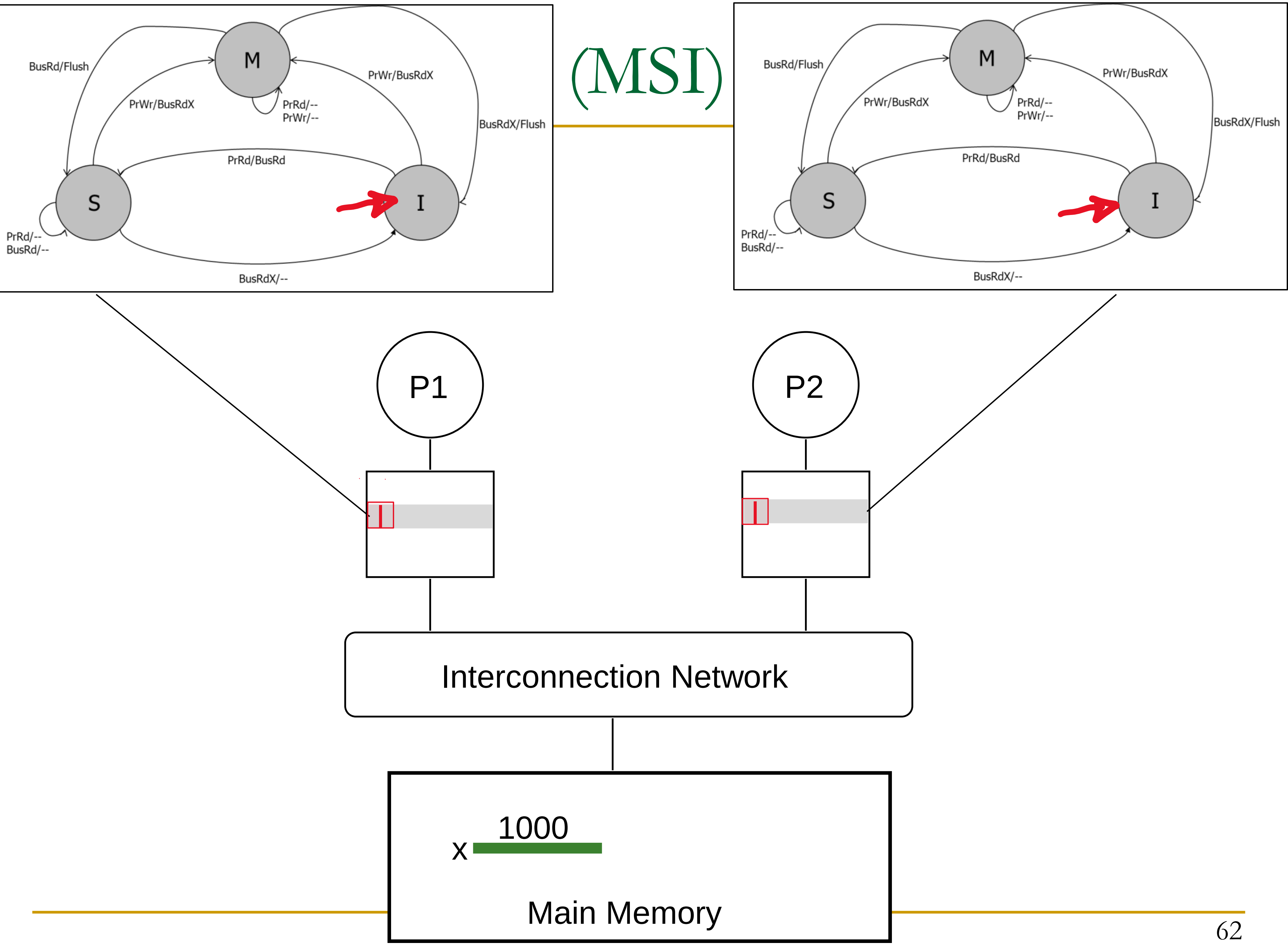
(MSI)


M
BusRd/Flush
PrWr/BusRdX
PrRd/--
PrWr/--
PrWr/BusRdX
BusRdX/Flush
PrRd/BusRd
S
I
PrRd/--
BusRd/--
BusRdX/--

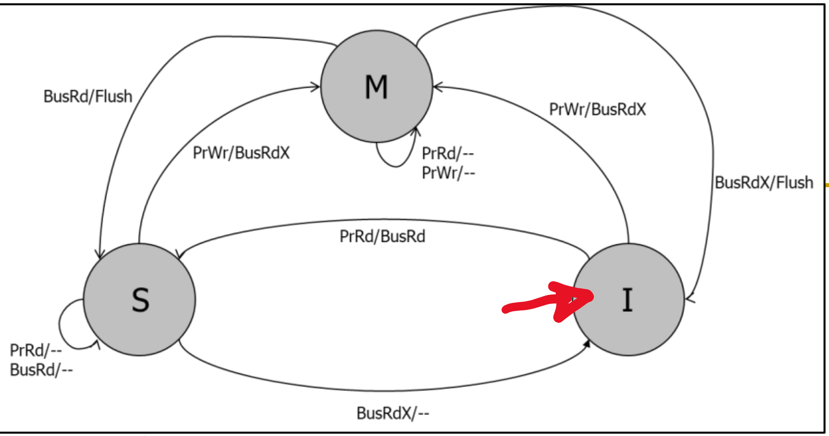

M
BusRd/Flush
PrWr/BusRdX
PrRd/--
PrWr/--
PrWr/BusRdX
BusRdX/Flush
PrRd/BusRd
S
I
PrRd/--
BusRd/--
BusRdX/--


P1
P2
Interconnection Network
x
1000
Main Memory

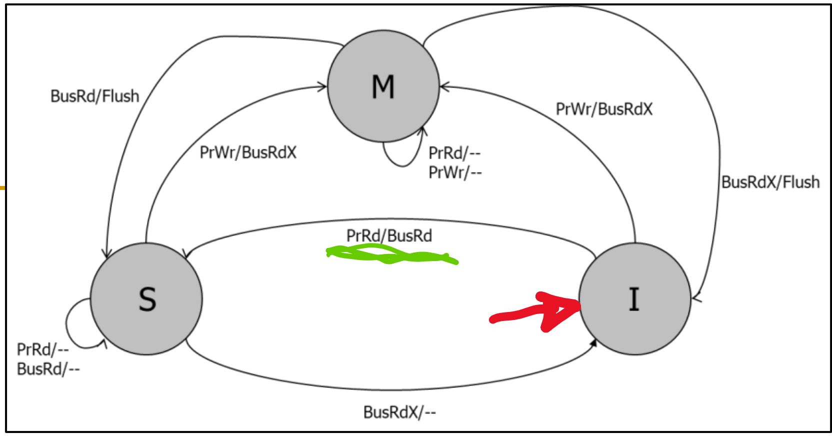
(MSI)

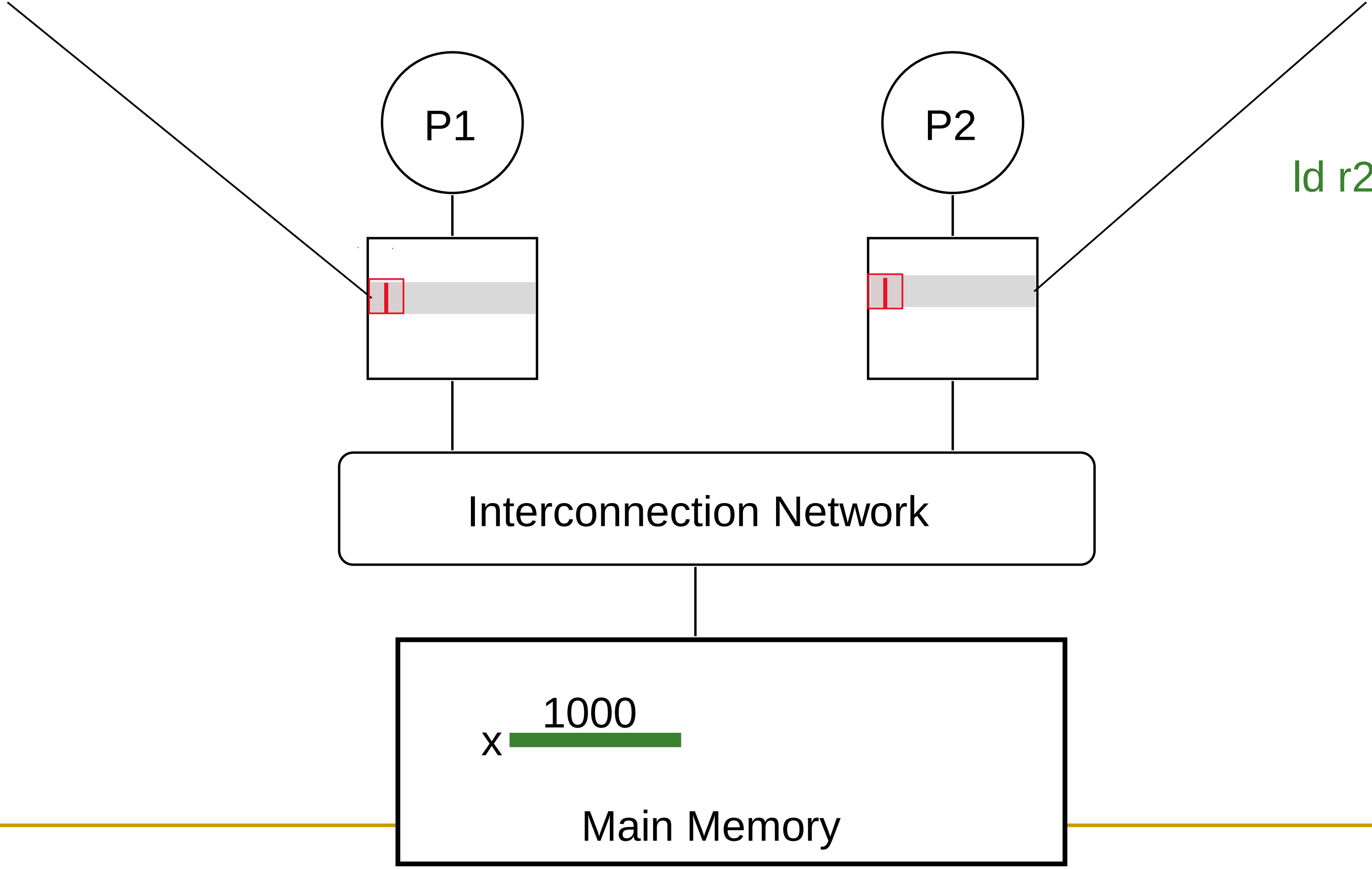


BusRd/Flush
M
PrWr/BusRdX
PrRd/--
PrWr/--
PrWr/BusRdX
BusRdX/Flush
PrRd/BusRd
S
I
PrRd/--
BusRd/--
BusRdX/--
BusRd/Flush
M
PrWr/BusRdX
PrRd/--
PrWr/--
PrWr/BusRdX
BusRdX/Flush
PrRd/BusRd
S
I
PrRd/--
BusRd/--
BusRdX/--
P1
P2
I
I
Interconnection Network
x
1000
Main Memory

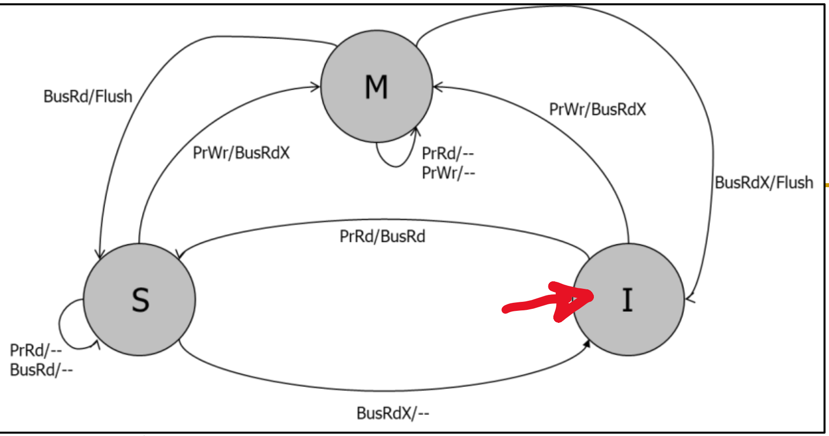

BusRd/Flush
M
PrWr/BusRdX
PrRd/--
PrWr/--
PrWr/BusRdX
BusRdX/Flush
PrRd/BusRd
S
I
PrRd/--
BusRd/--
BusRdX/--

(MSI)

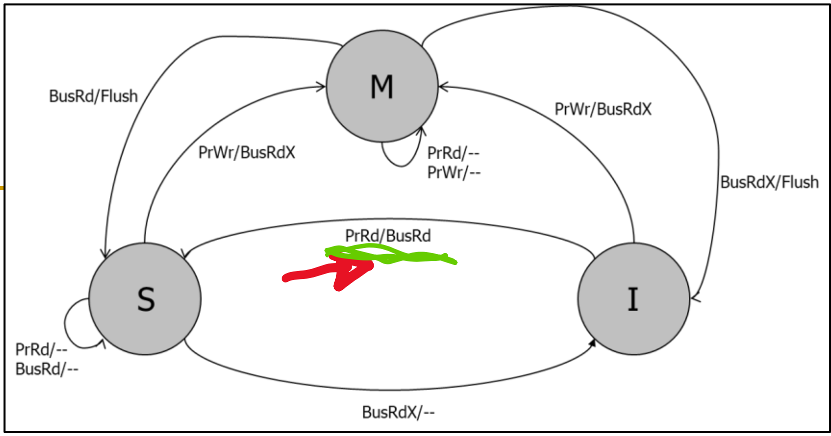

BusRd/Flush
M
PrWr/BusRdX
PrRd/--
PrWr/--
PrWr/BusRdX
BusRdX/Flush
PrRd/BusRd
S
I
PrRd/--
BusRd/--
BusRdX/--

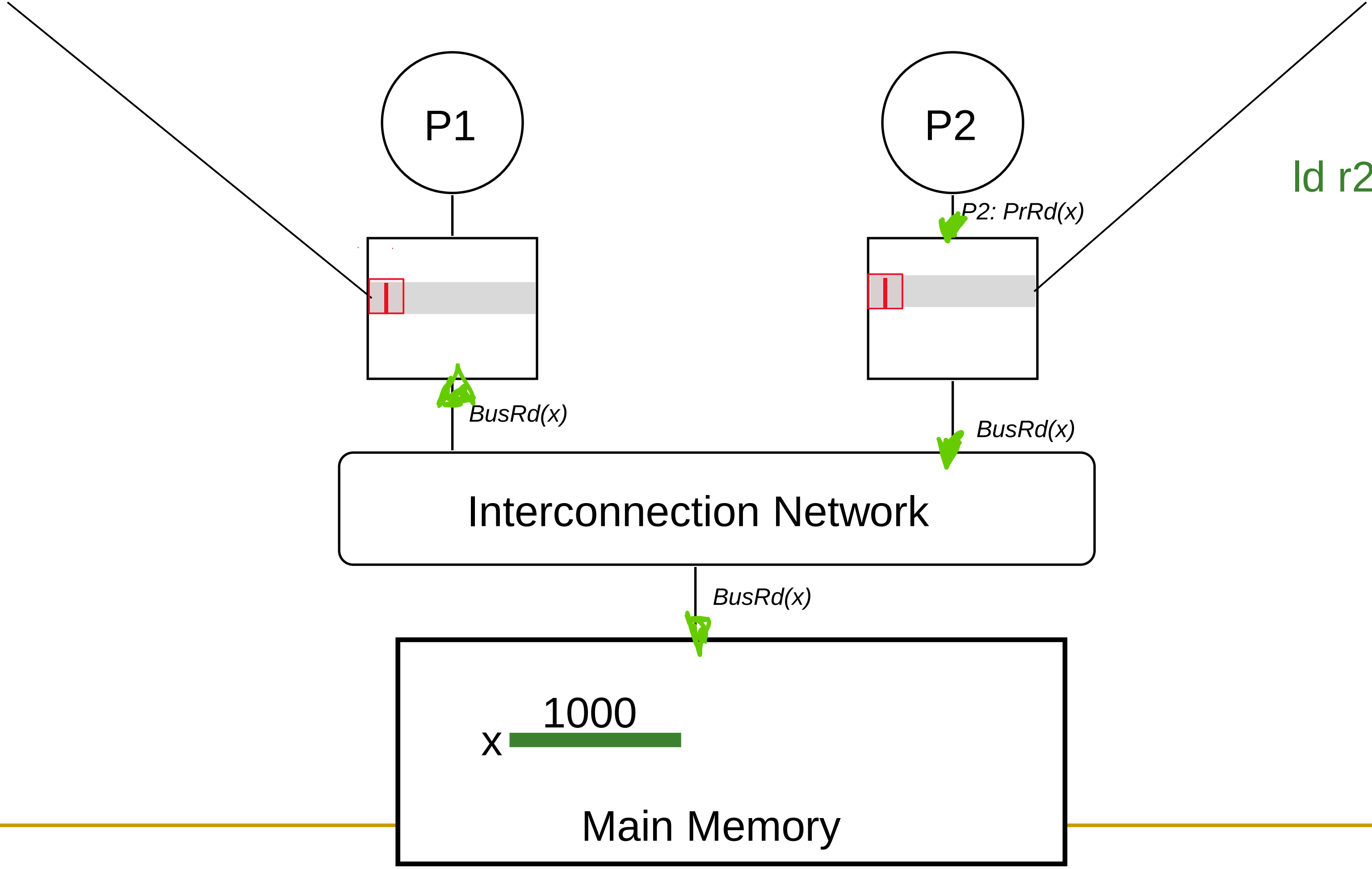

P1
P2
I
I
Interconnection Network
x
1000
Main Memory

ld r2, x


BusRd/Flush
M
PrWr/BusRdX
PrWr/BusRdX
PrRd/--
PrWr/--
BusRdX/Flush
PrRd/BusRd
S
I
PrRd/--
BusRd/--
BusRdX/--

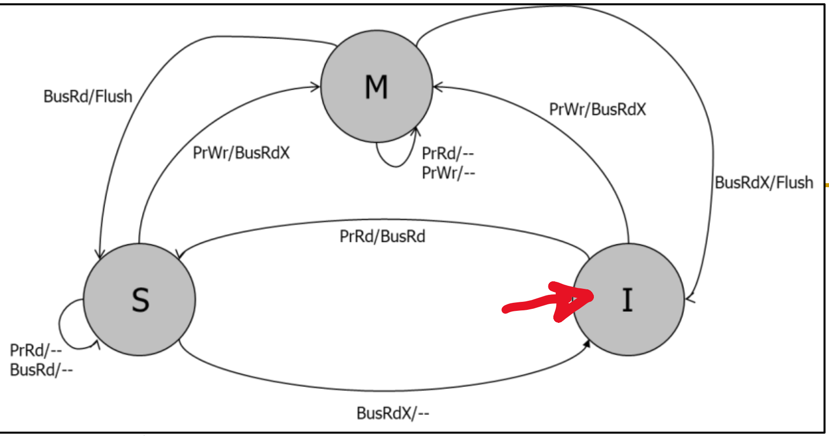
(MSI)

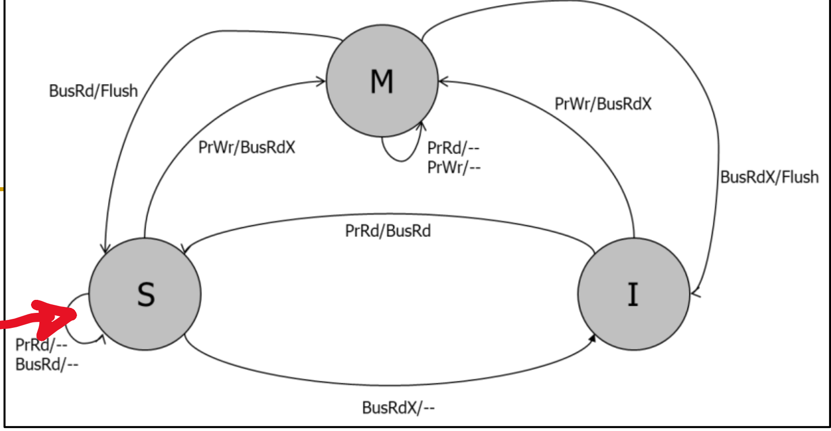

BusRd/Flush
M
PrWr/BusRdX
PrWr/BusRdX
PrRd/--
PrWr/--
BusRdX/Flush
PrRd/BusRd
S
I
PrRd/--
BusRd/--
BusRdX/--

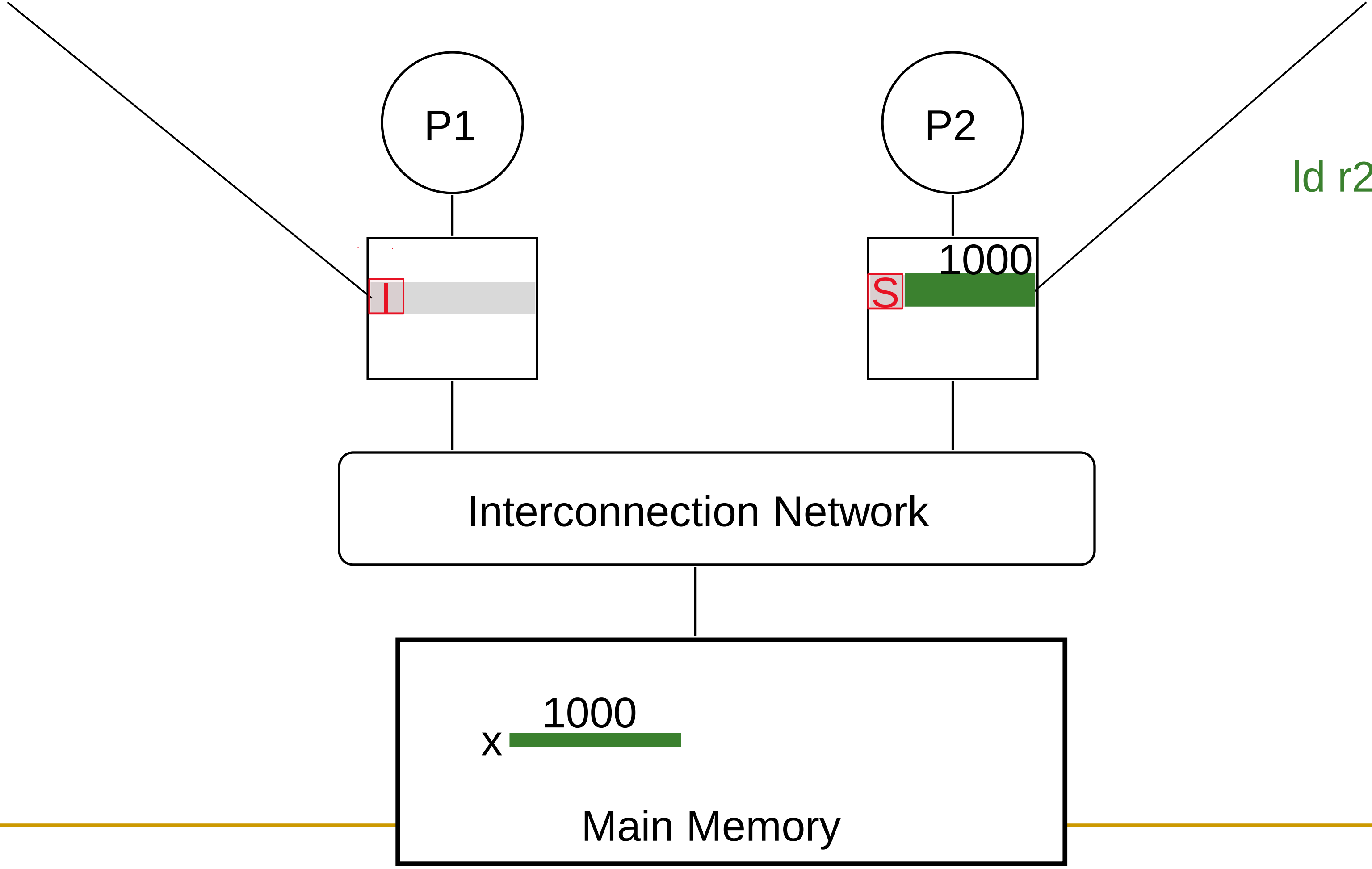

P1
P2
P2: PrRd(x)
I
I
BusRd(x)
BusRd(x)
Interconnection Network
BusRd(x)
x
1000
Main Memory

ld r2, x

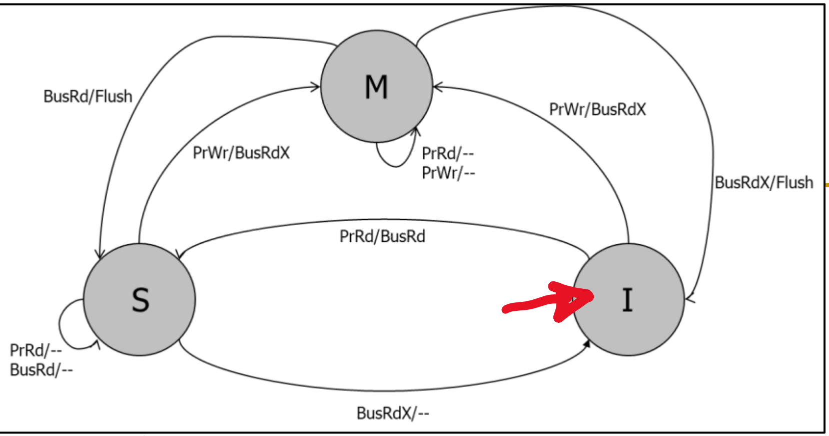
(MSI)


M
S
I
BusRd/Flush
PrWr/BusRdX
PrRd/--
PrWr/--
PrWr/BusRdX
BusRdX/Flush
PrRd/BusRd
PrRd/--
BusRd/--
BusRdX/--

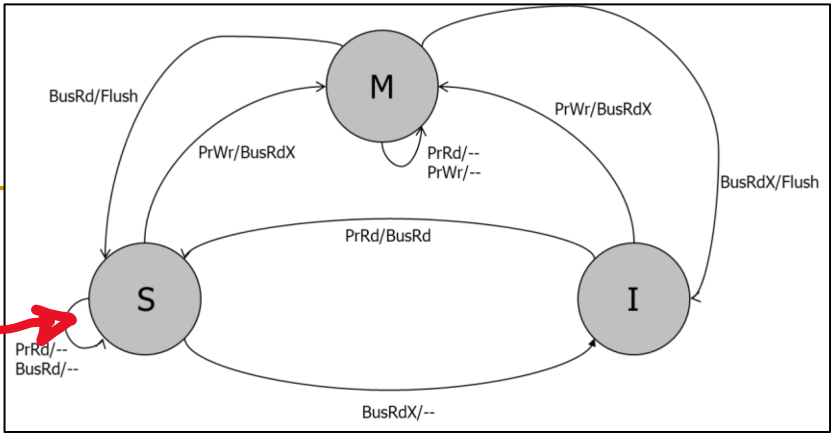

M
S
I
BusRd/Flush
PrWr/BusRdX
PrRd/--
PrWr/--
PrWr/BusRdX
BusRdX/Flush
PrRd/BusRd
PrRd/--
BusRd/--
BusRdX/--

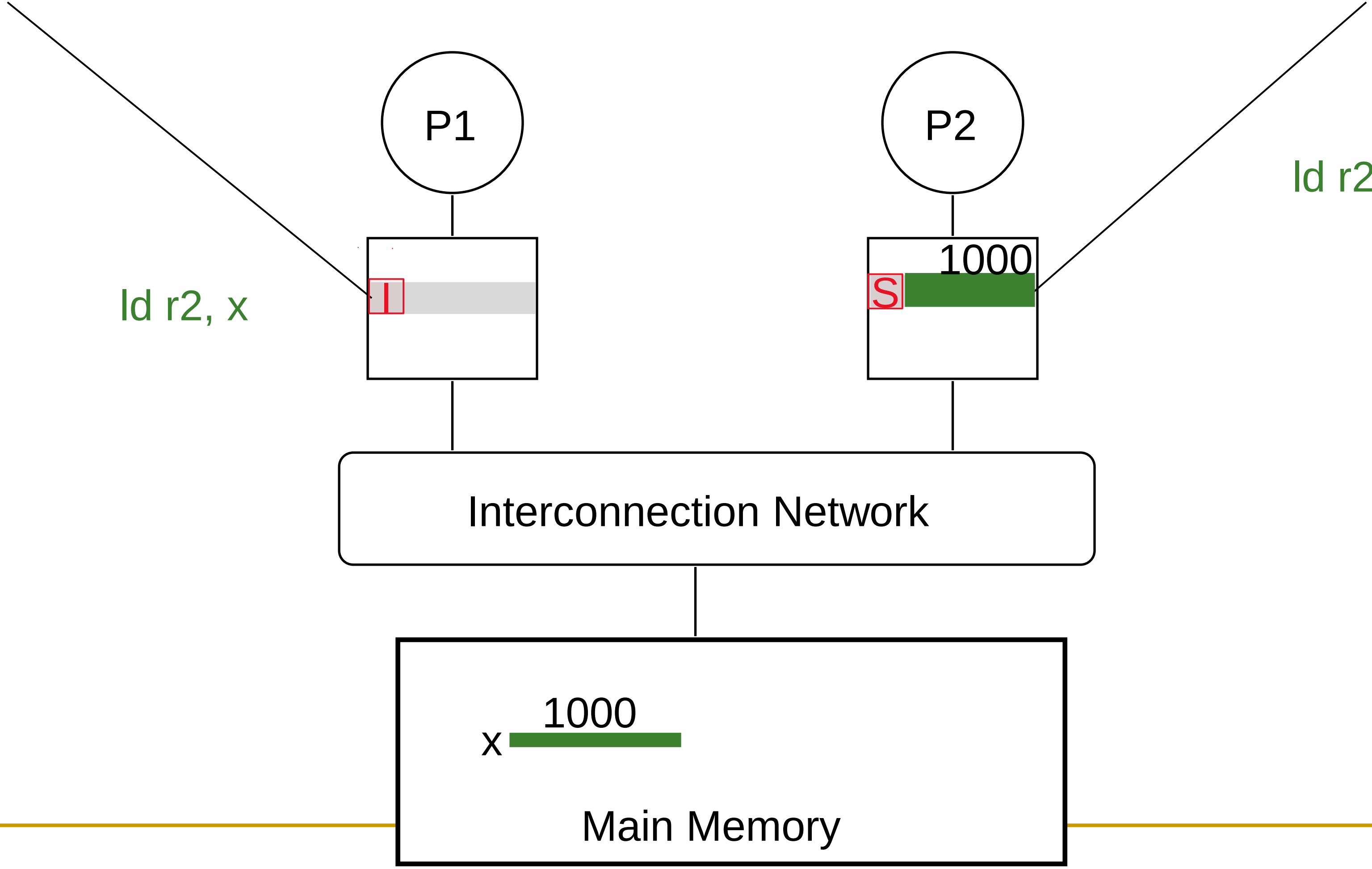

P1
P2
I
S
1000
Interconnection Network
x
1000
Main Memory

ld r2, x



(MSI)





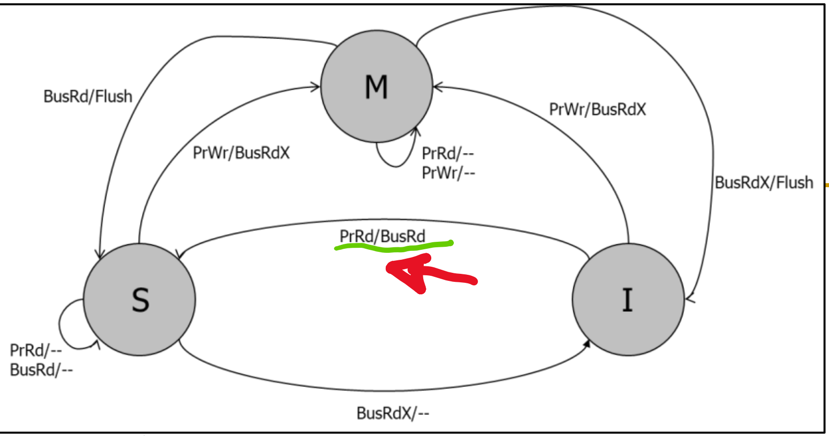
ld r2, x

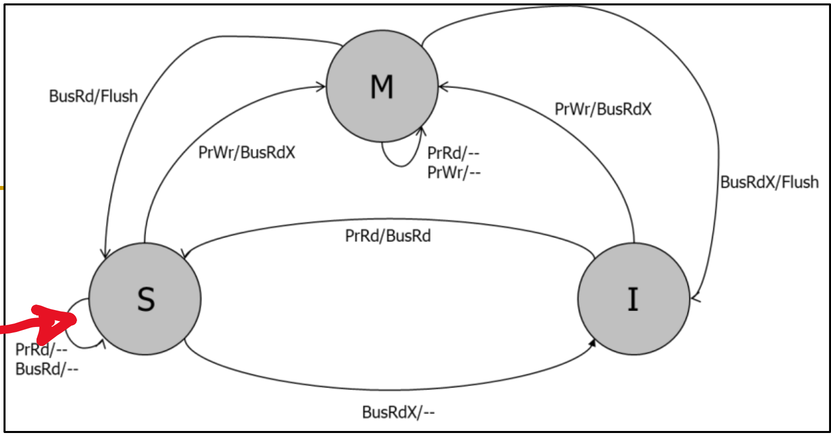
ld r2, x

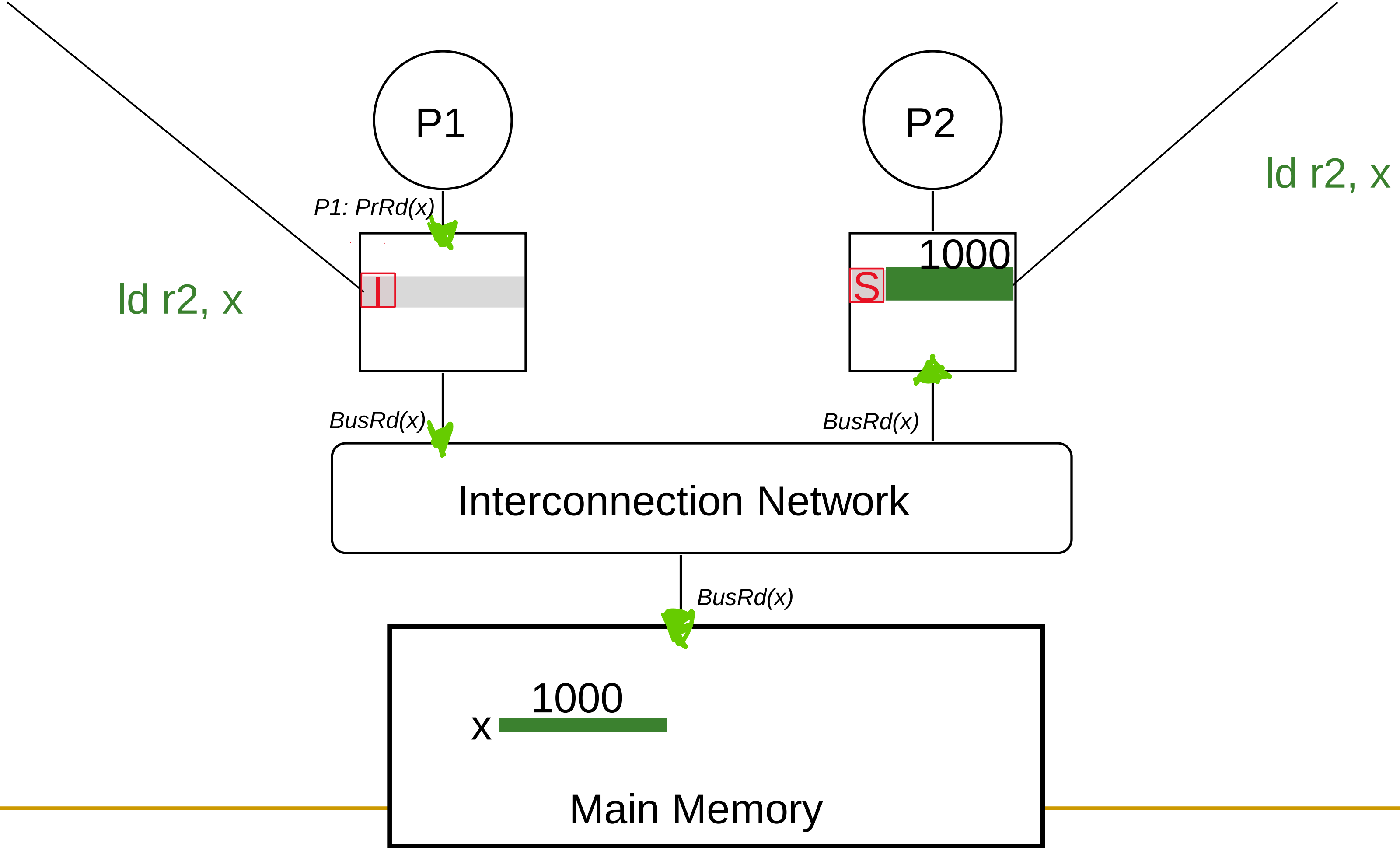
Interconnection Network

Main Memory

(MSI)







P1

P2

ld r2, x

ld r2, x

P1: PrRd(x)

I

S

1000

BusRd(x)

BusRd(x)

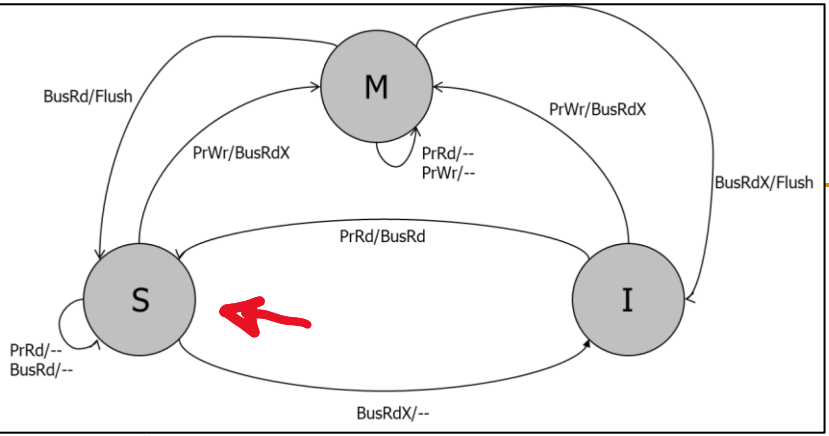
Interconnection Network

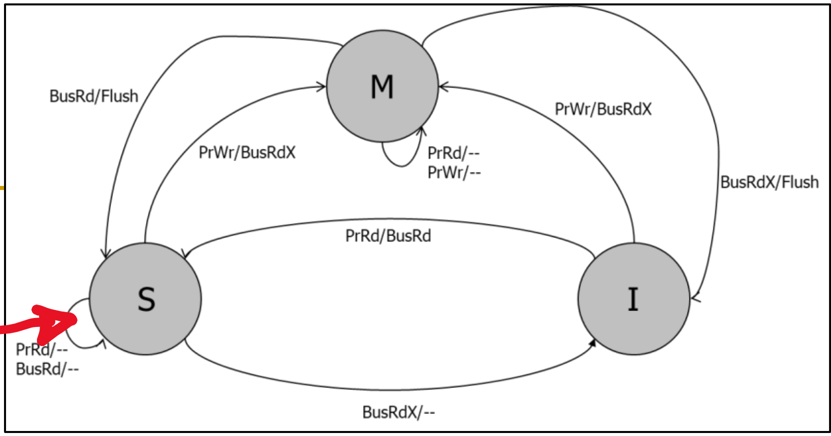
BusRd(x)

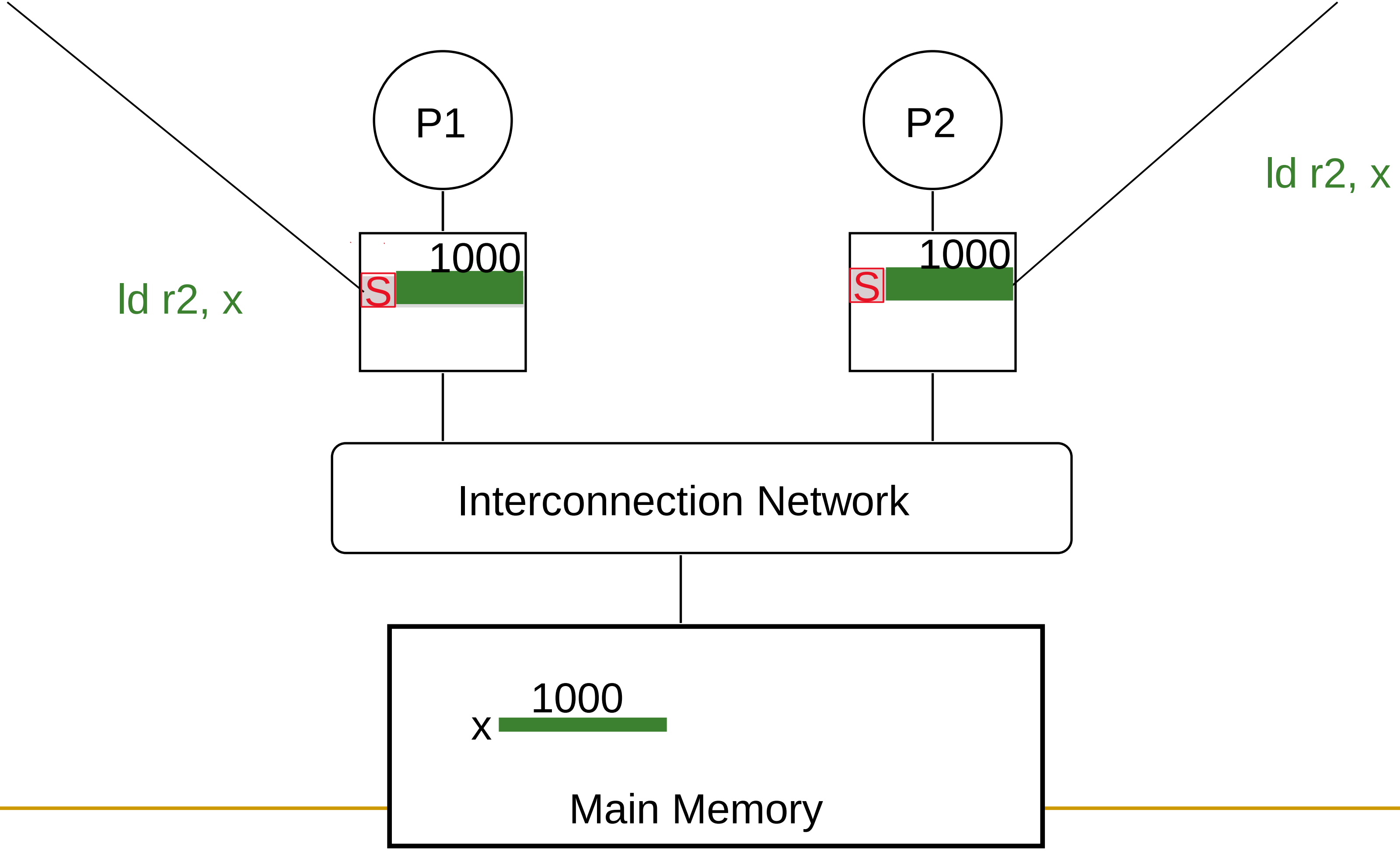
x 1000

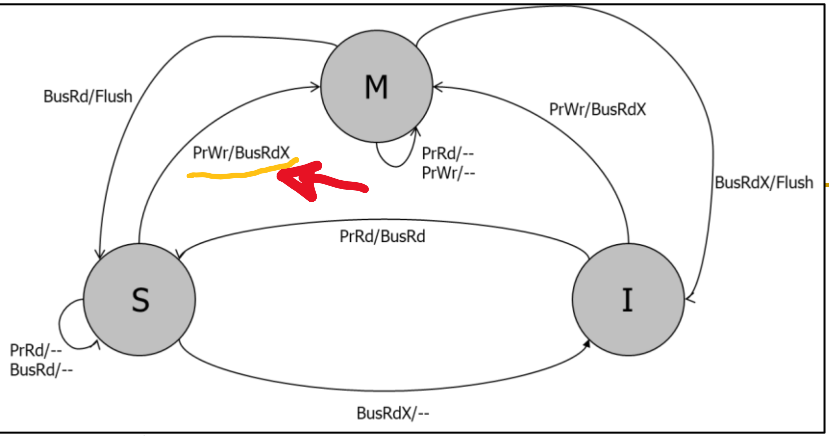
Main Memory

(MSI)

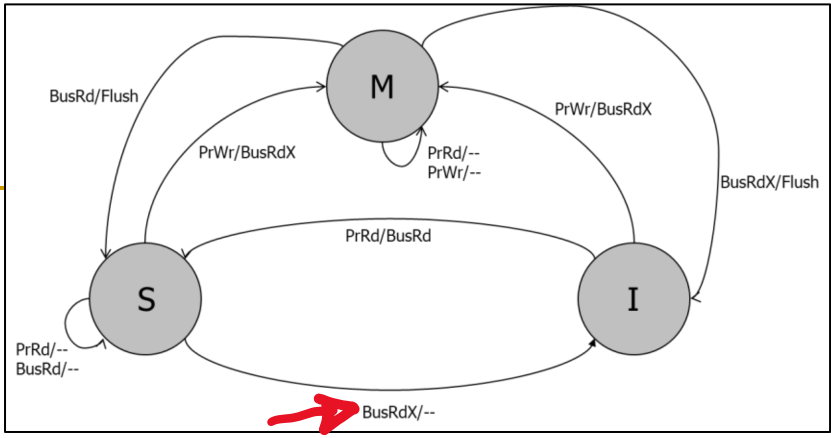

M
S
I
BusRd/Flush
PrWr/BusRdX
PrWr/BusRdX
PrRd/--
PrWr/--
BusRdX/Flush
PrRd/BusRd
PrRd/--
BusRd/--
BusRdX/--


M
S
I
BusRd/Flush
PrWr/BusRdX
PrWr/BusRdX
PrRd/--
PrWr/--
BusRdX/Flush
PrRd/BusRd
PrRd/--
BusRd/--
BusRdX/--

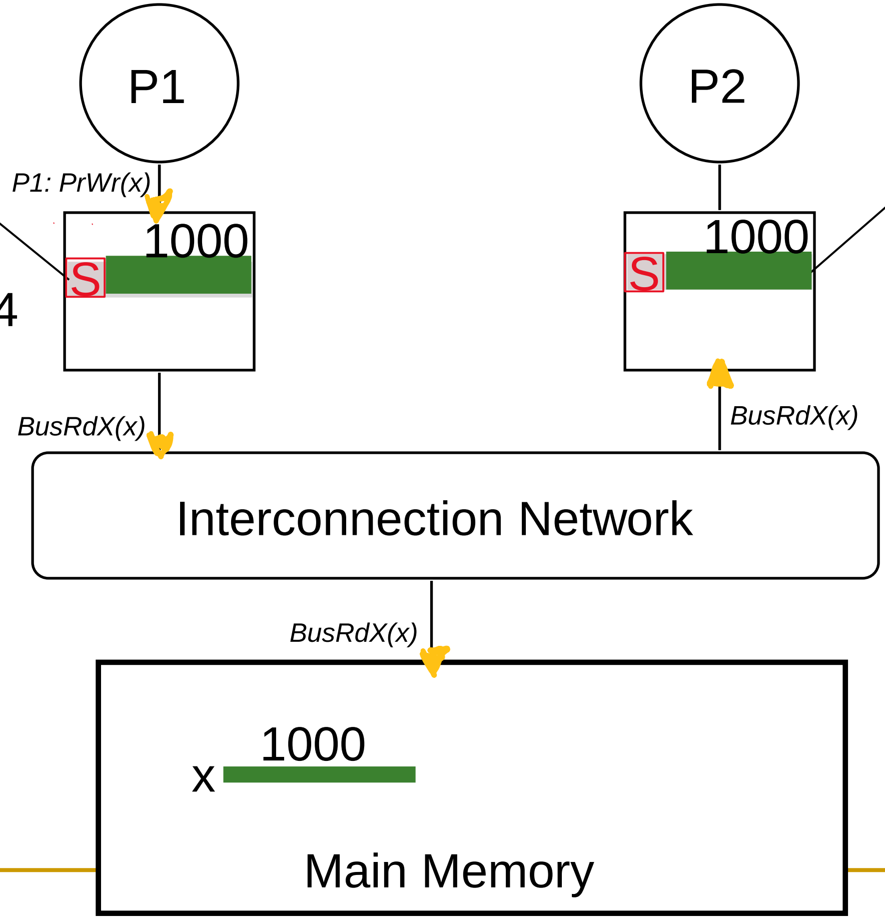

P1
P2
1000
S
1000
S
ld r2, x
ld r2, x
Interconnection Network
x
1000
Main Memory



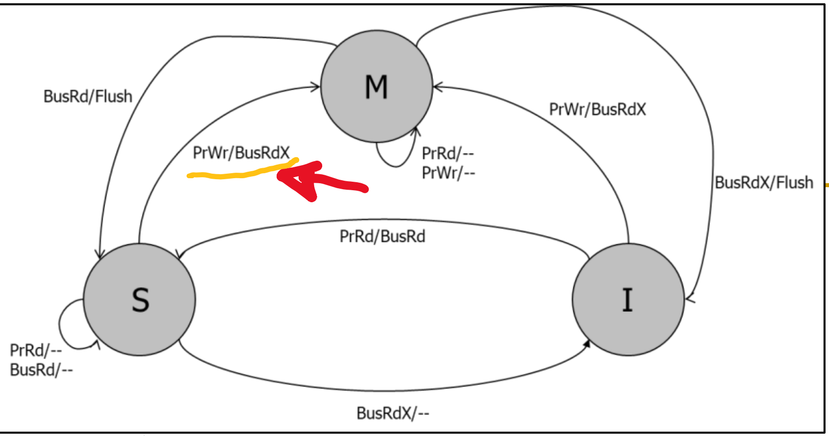
(MSI)



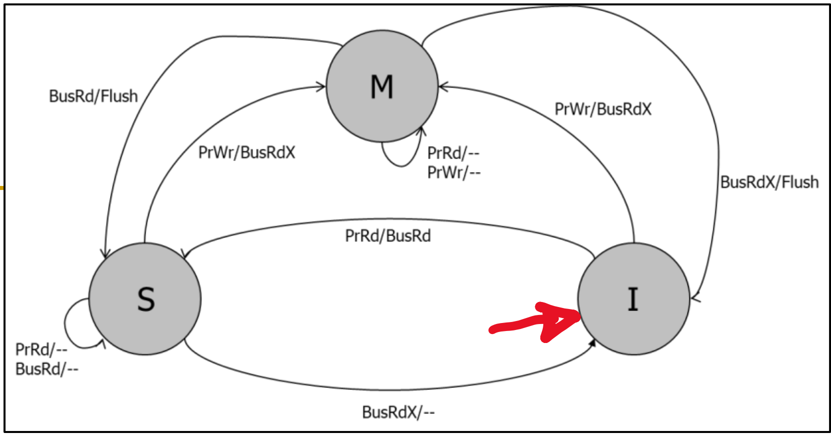
ld r2, x
add r1, r2, r4
st x, r1



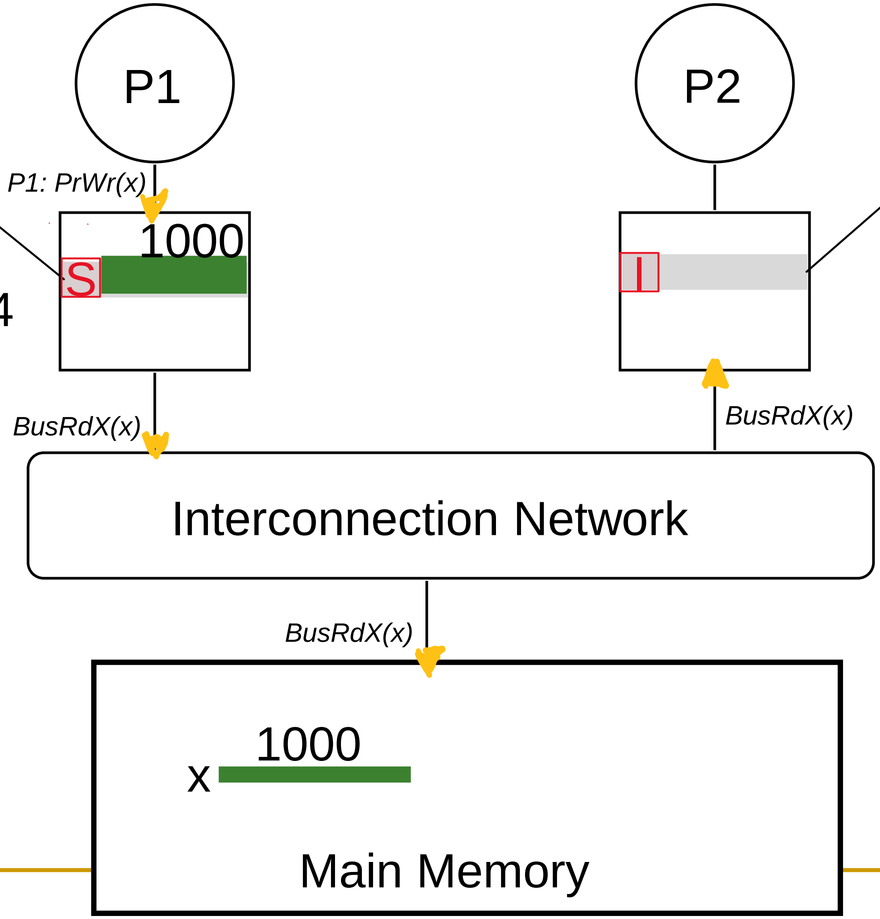
ld r2, x



(MSI)

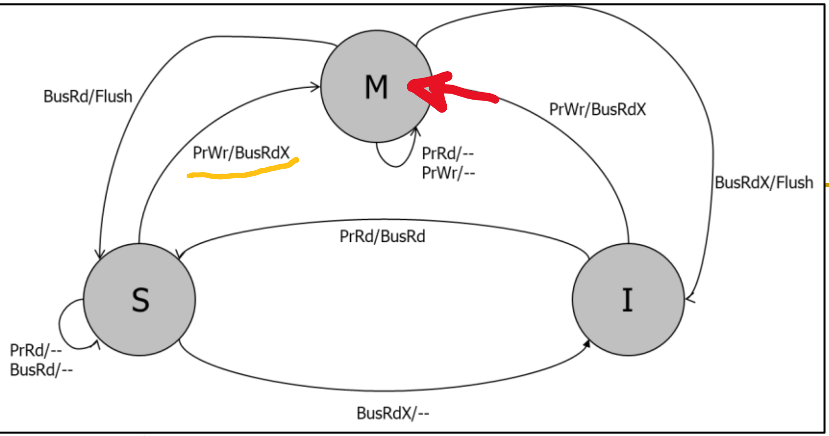


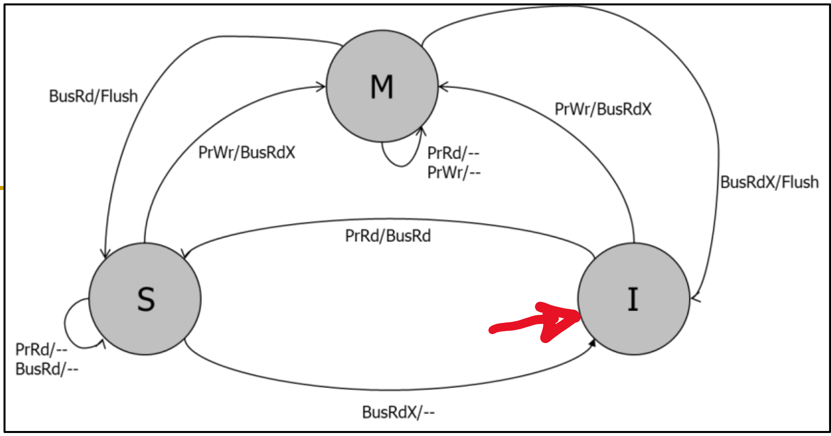
```
ld r2, x
add r1, r2, r4
st x, r1
```



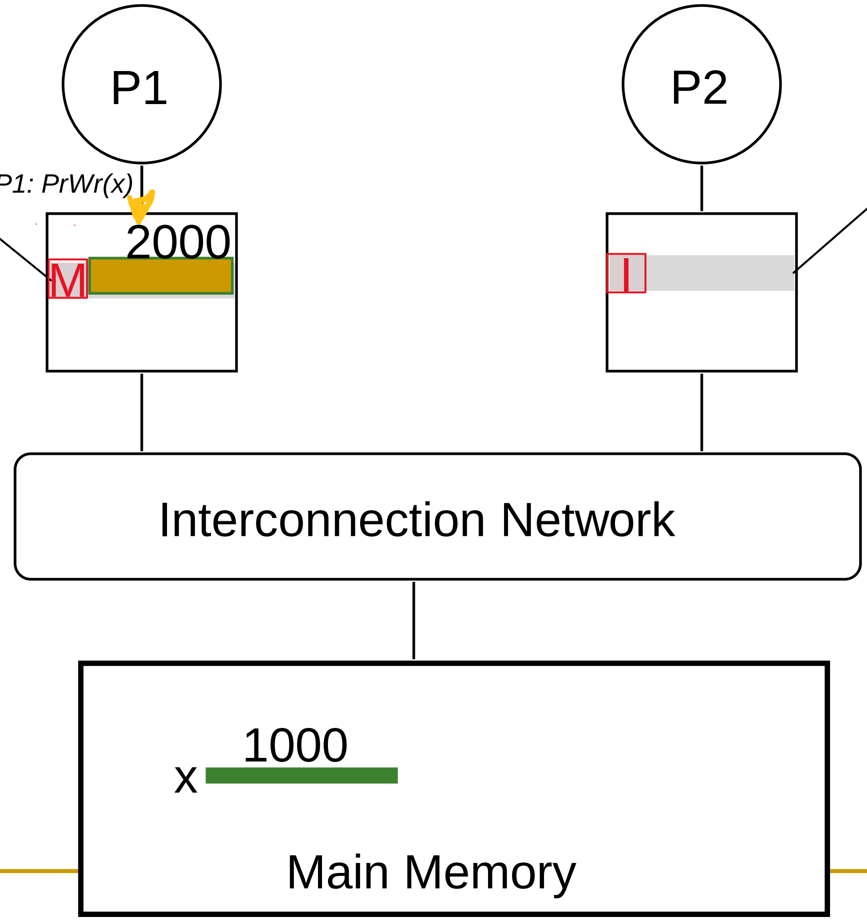
```
ld r2, x
```

(MSI)

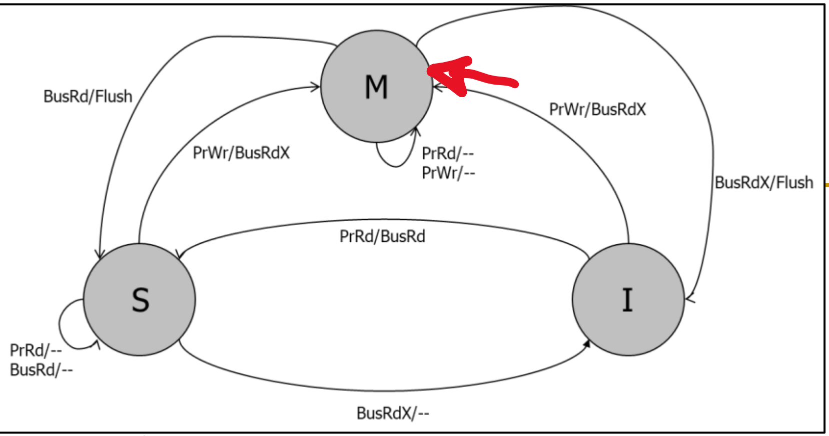




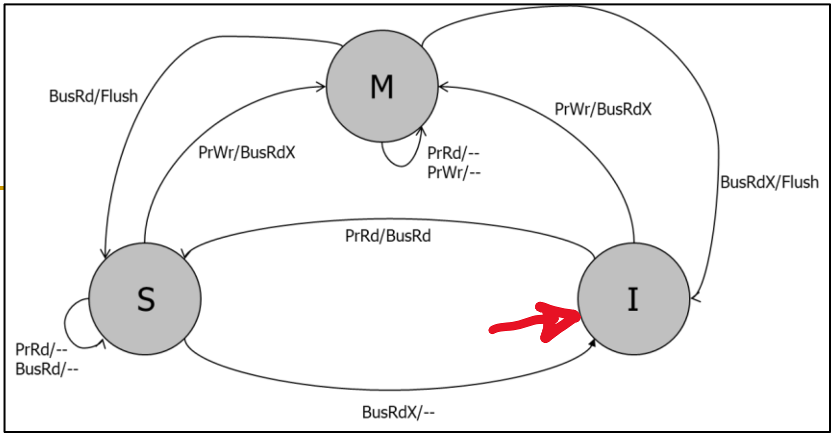
ld r2, x
add r1, r2, r4
st x, r1



ld r2, x



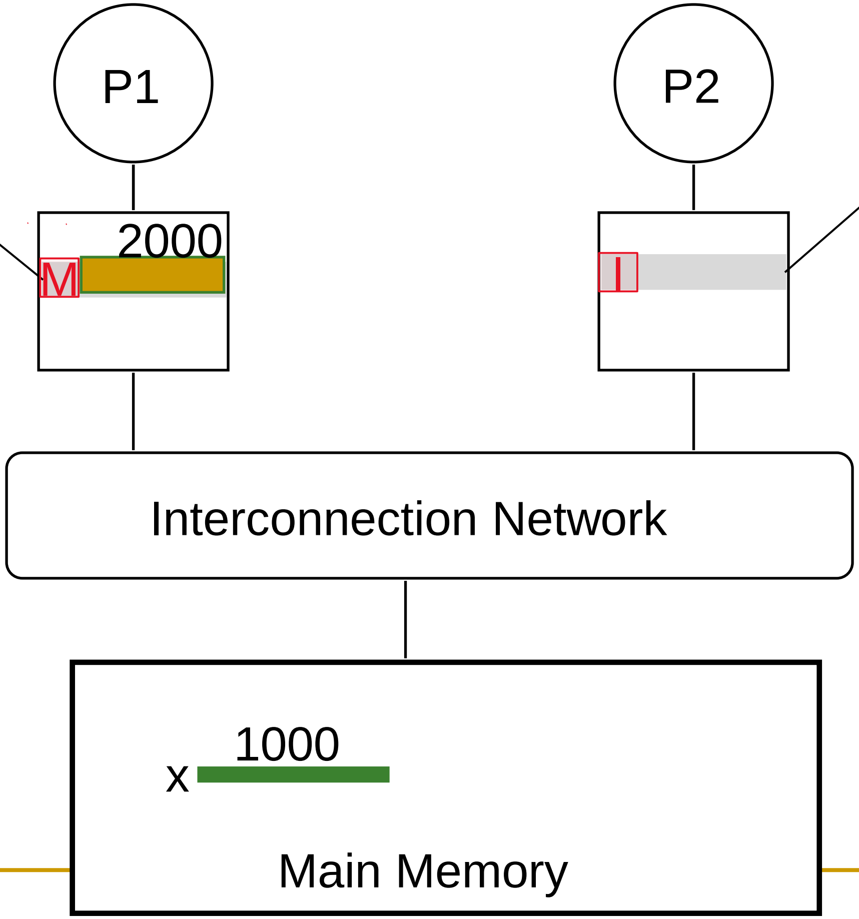
(MSI)



ld r2, x

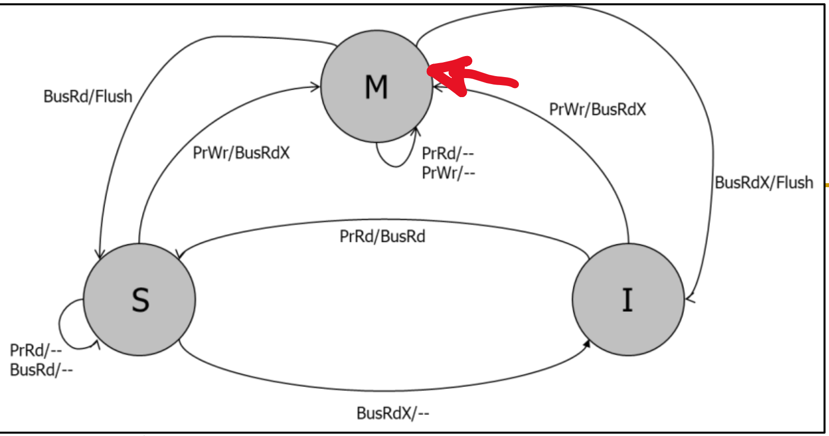
add r1, r2, r4

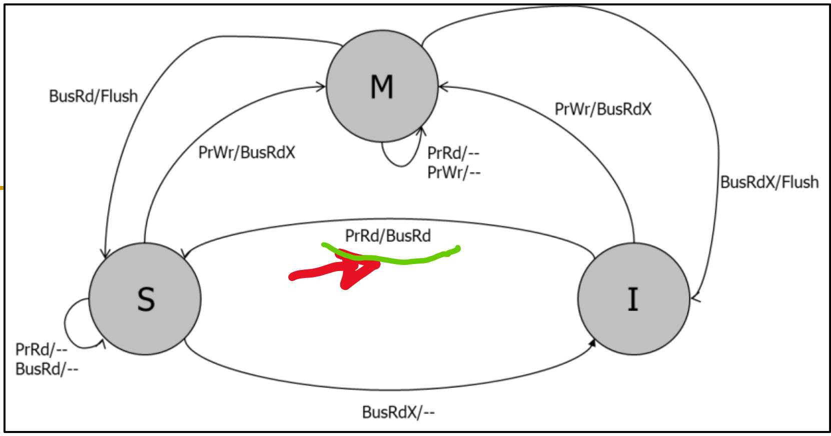
st x, r1

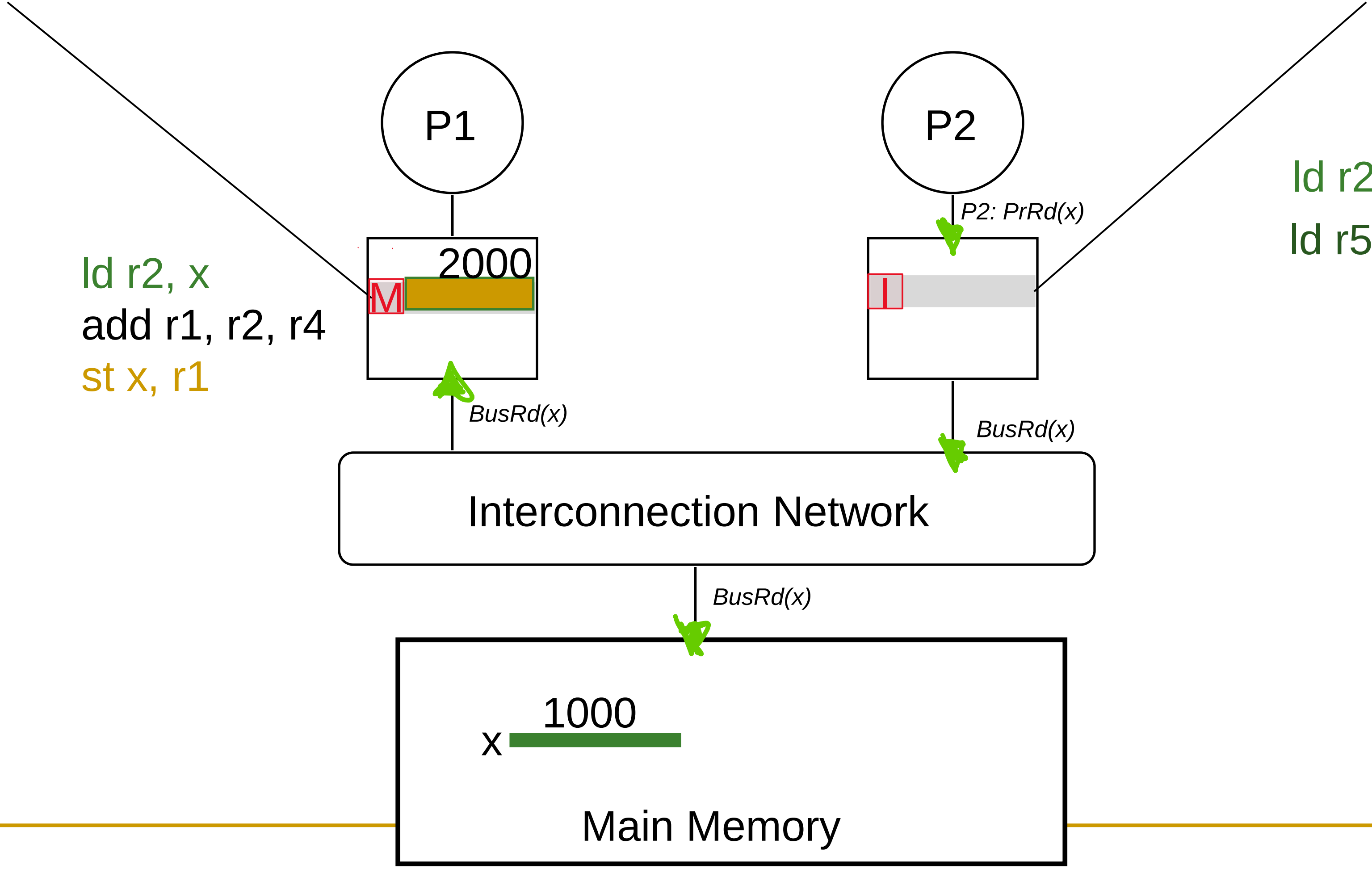


ld r2, x

(MSI)

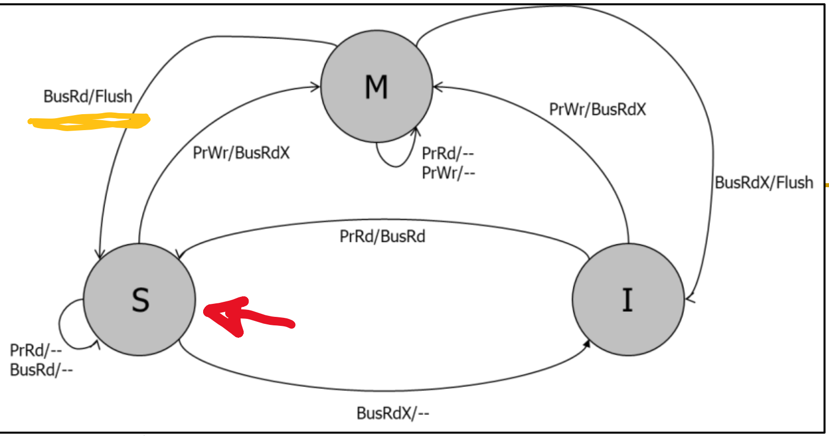




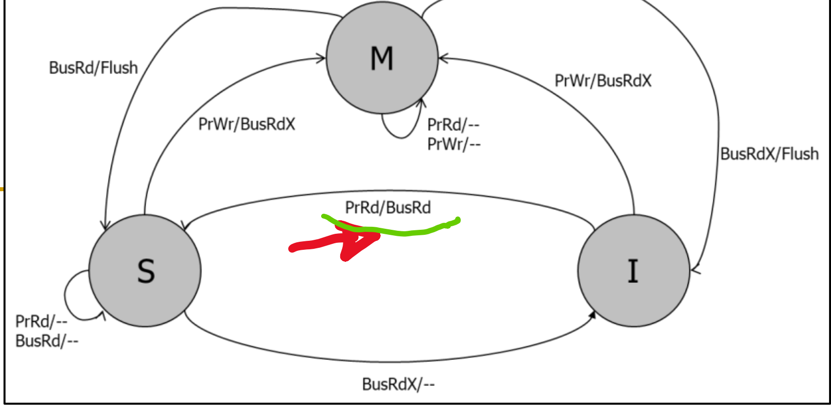


ld r2, x
add r1, r2, r4
st x, r1

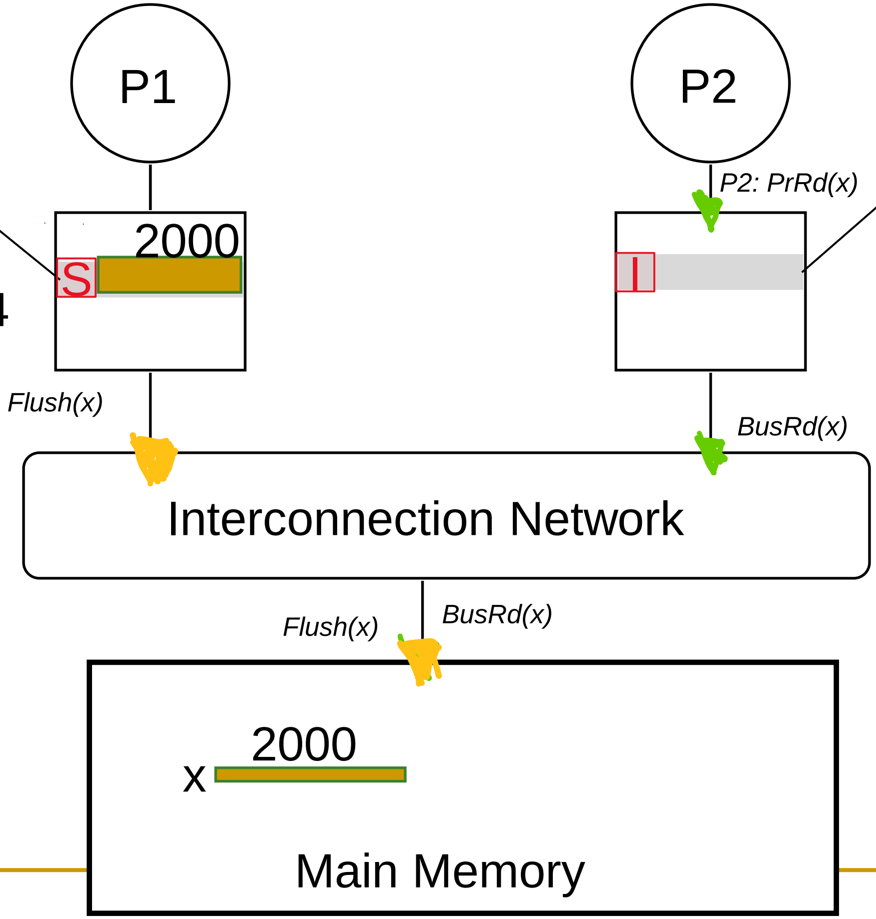
ld r2, x
ld r5, x



(MSI)

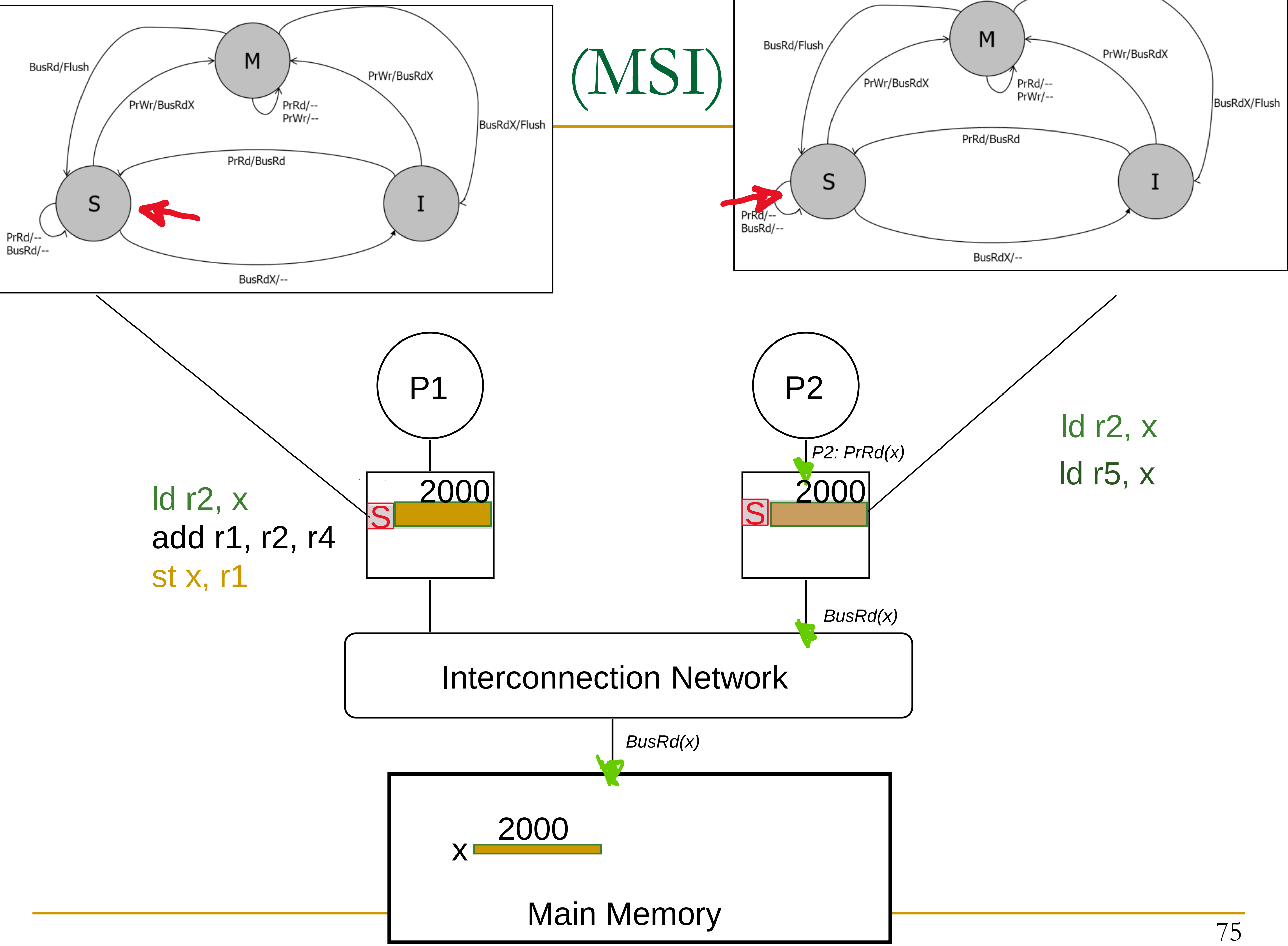


ld r2, x
add r1, r2, r4
st x, r1



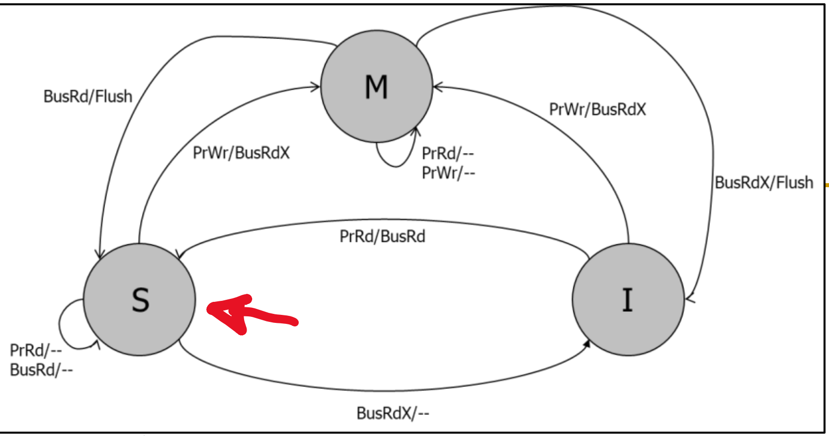
ld r2, x
ld r5, x

(MSI)

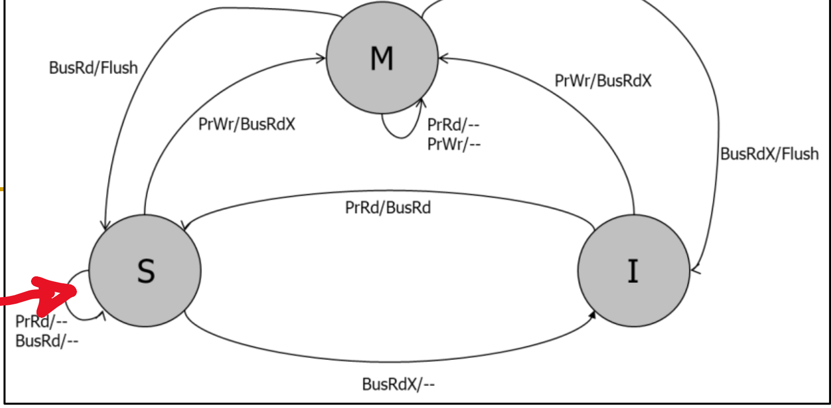


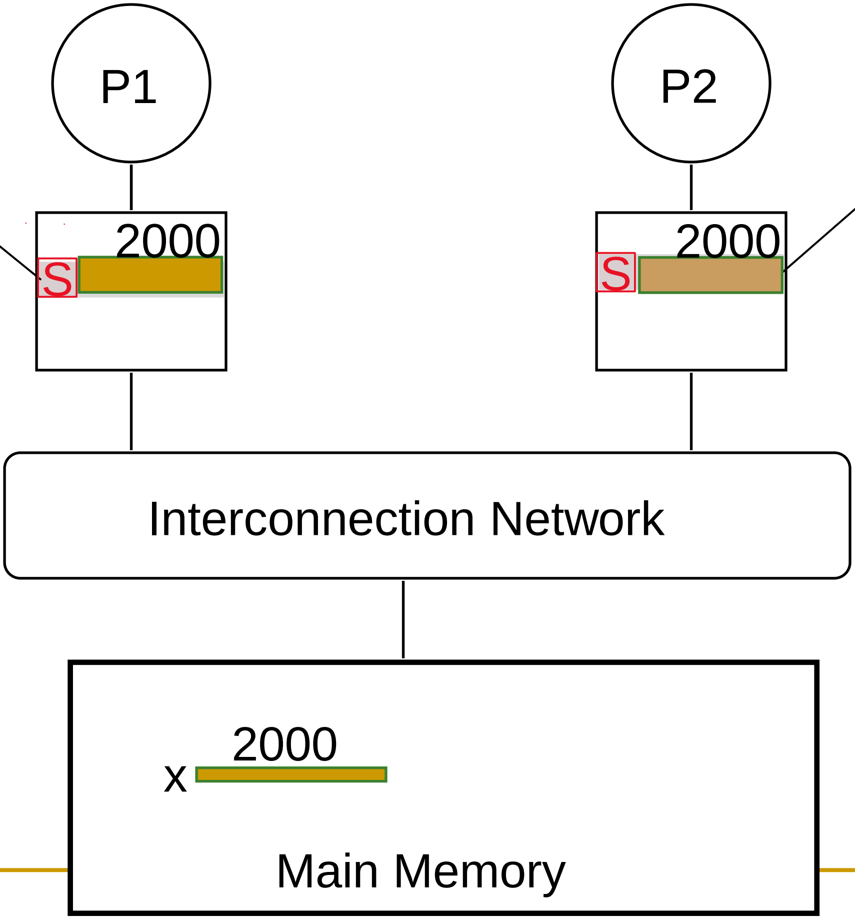
ld r2, x
add r1, r2, r4
st x, r1

ld r2, x
ld r5, x



(MSI)





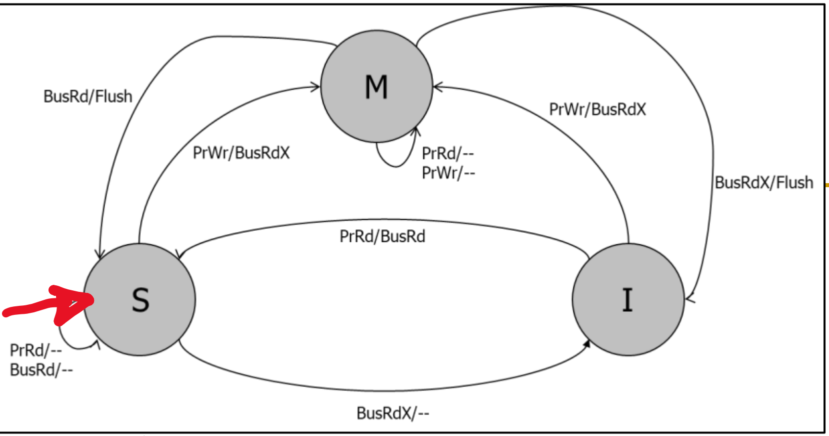
ld r2, x
add r1, r2, r4
st x, r1

ld r2, x
ld r5, x

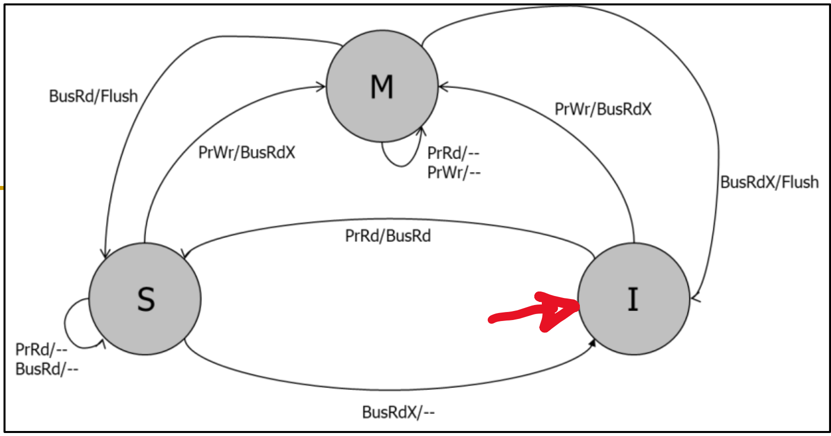
The Problem with MSI

- 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)

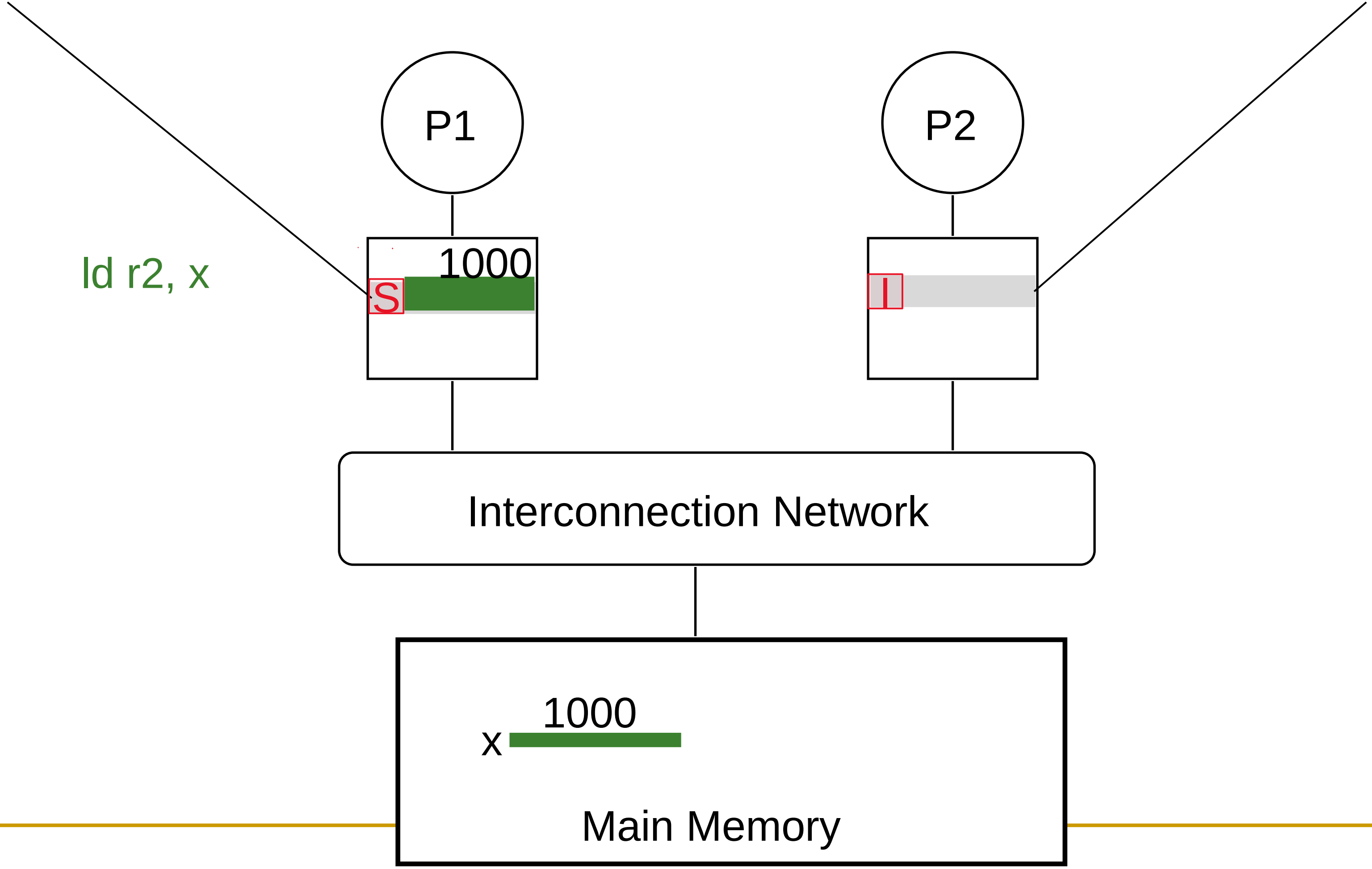
- Why is this a problem?
  - Suppose the cache that reads the block wants to write to it at some point
  - It needs to broadcast "invalidate" even though it has the only cached copy!
  - *If the cache knew it had the only cached copy in the system,* it could have written to the block without notifying any other cache → saves unnecessary broadcasts of invalidations

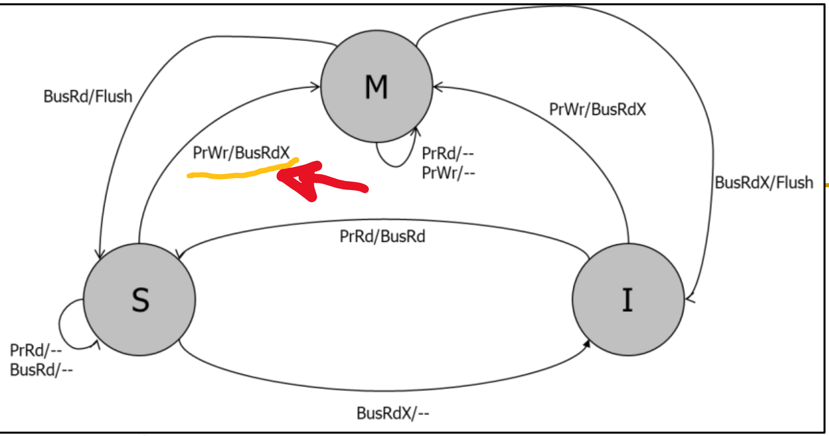

BusRd/Flush
M
PrWr/BusRdX
PrRd/--
PrWr/--
PrWr/BusRdX
BusRdX/Flush
PrRd/BusRd
S
I
PrRd/--
BusRd/--
BusRdX/--

(MSI)

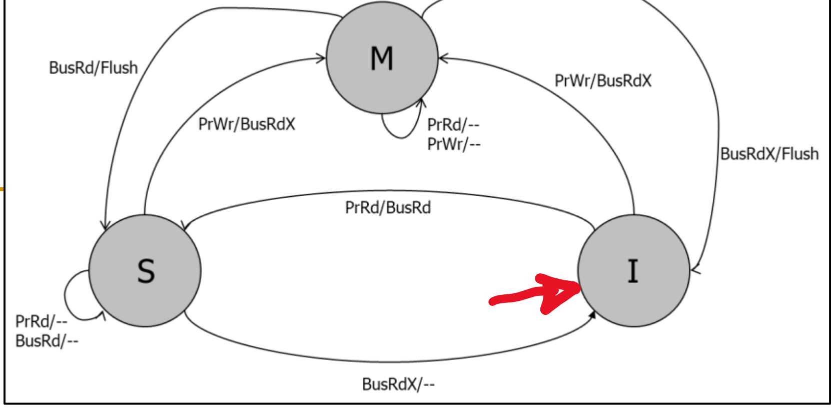

BusRd/Flush
M
PrWr/BusRdX
PrRd/--
PrWr/--
PrWr/BusRdX
BusRdX/Flush
PrRd/BusRd
S
I
PrRd/--
BusRd/--
BusRdX/--

ld r2, x

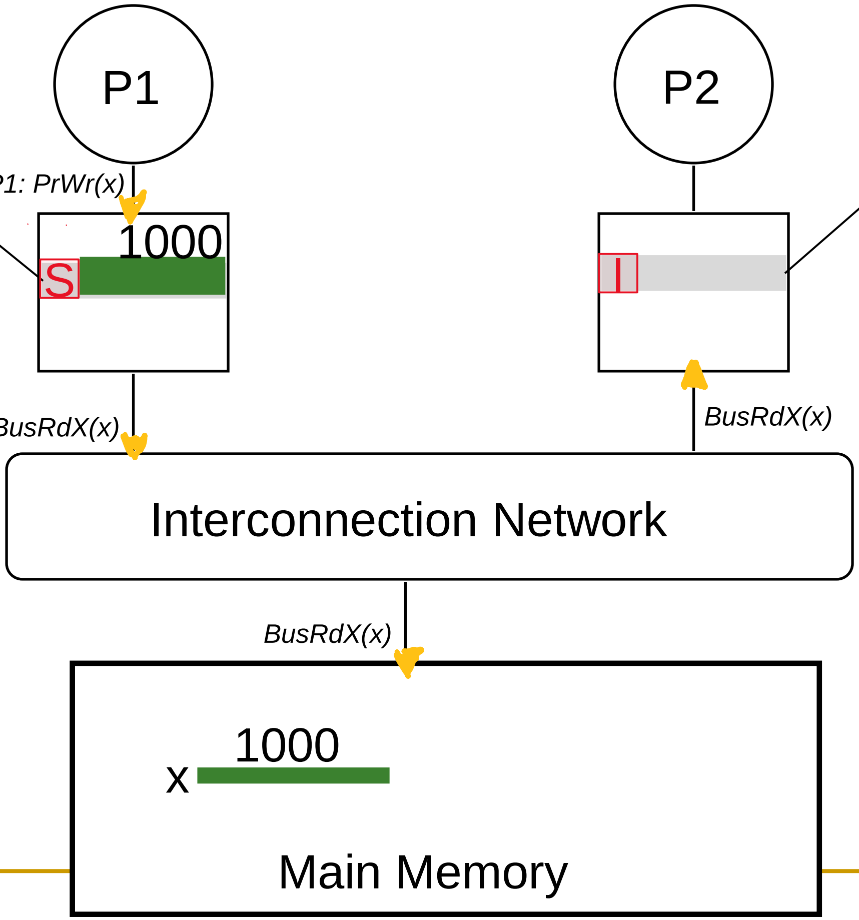

P1
P2
1000
S
I
Interconnection Network
x
1000
Main Memory



(MSI)



```
ld r2, x
add r1, r2, r4
st x, r1
```



P1

P2

P1: PrWr(x)

1000

S

I

BusRdX(x)

BusRdX(x)

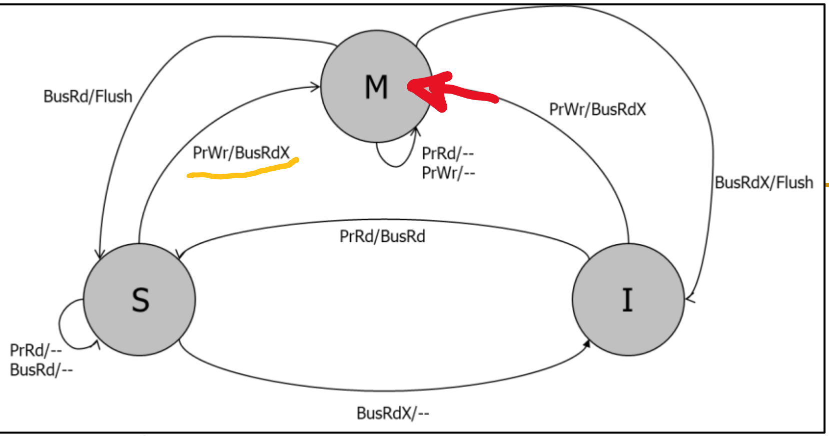
Interconnection Network

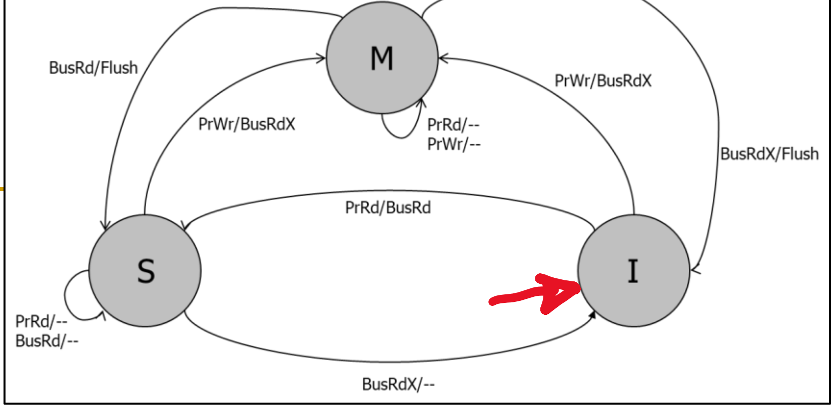
BusRdX(x)

x 1000

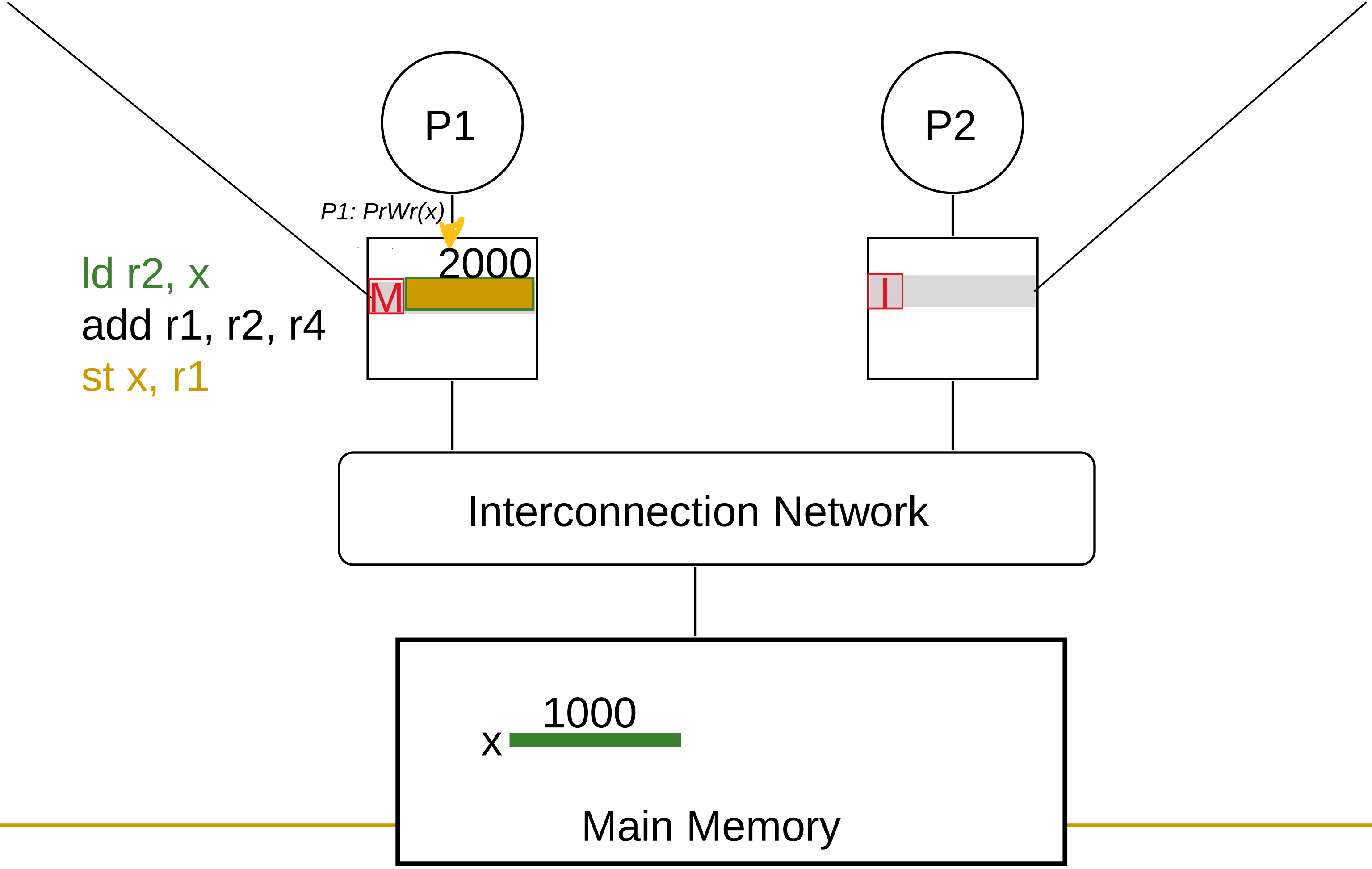
Main Memory

(MSI)

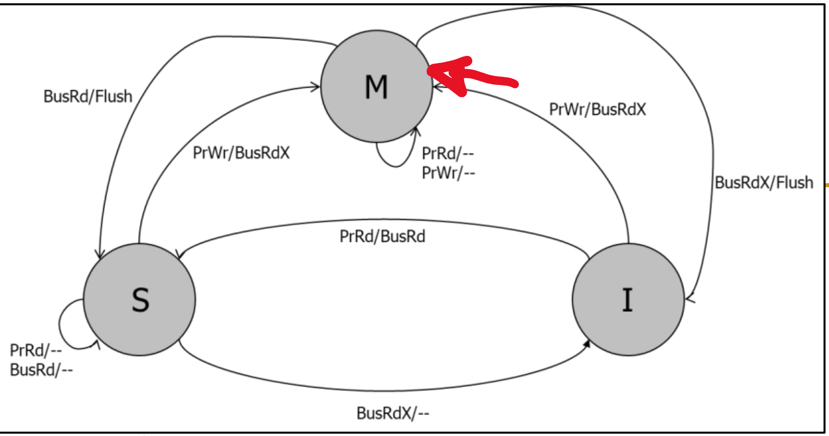


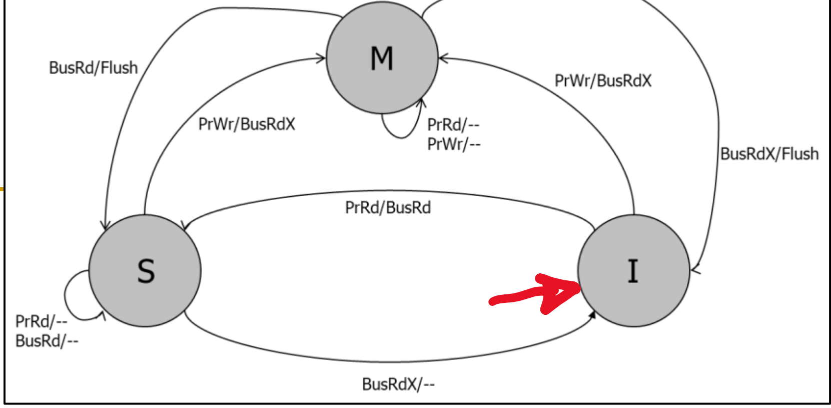


ld r2, x
add r1, r2, r4
st x, r1

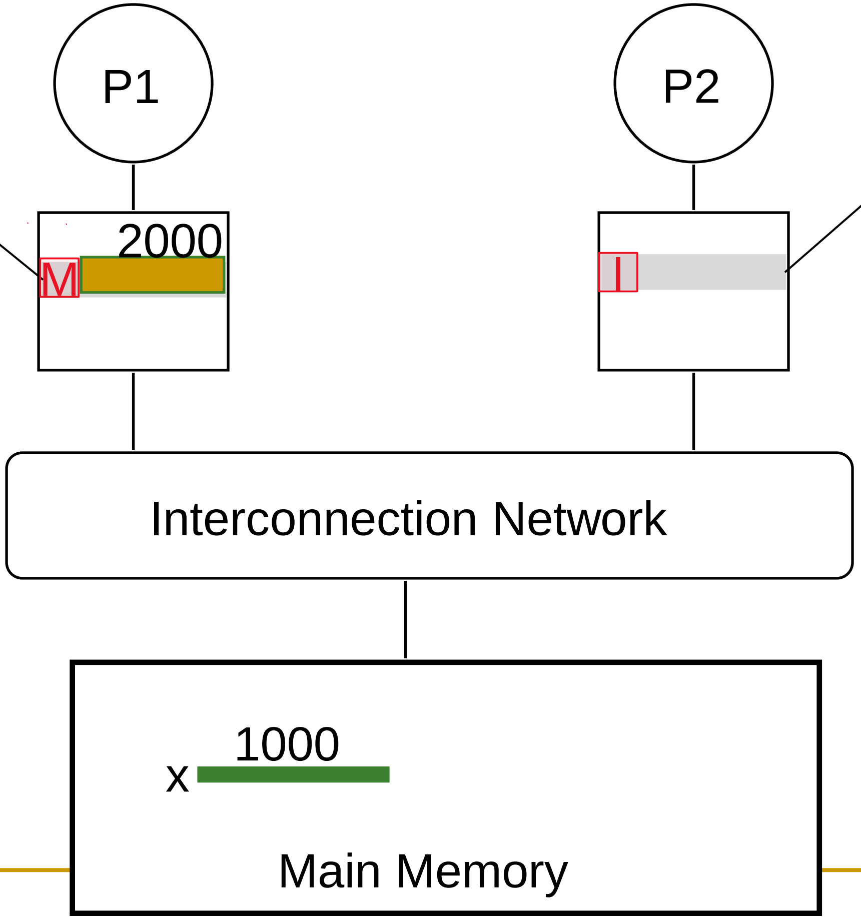


(MSI)





ld r2, x
add r1, r2, r4
st x, r1



The Solution: MESI

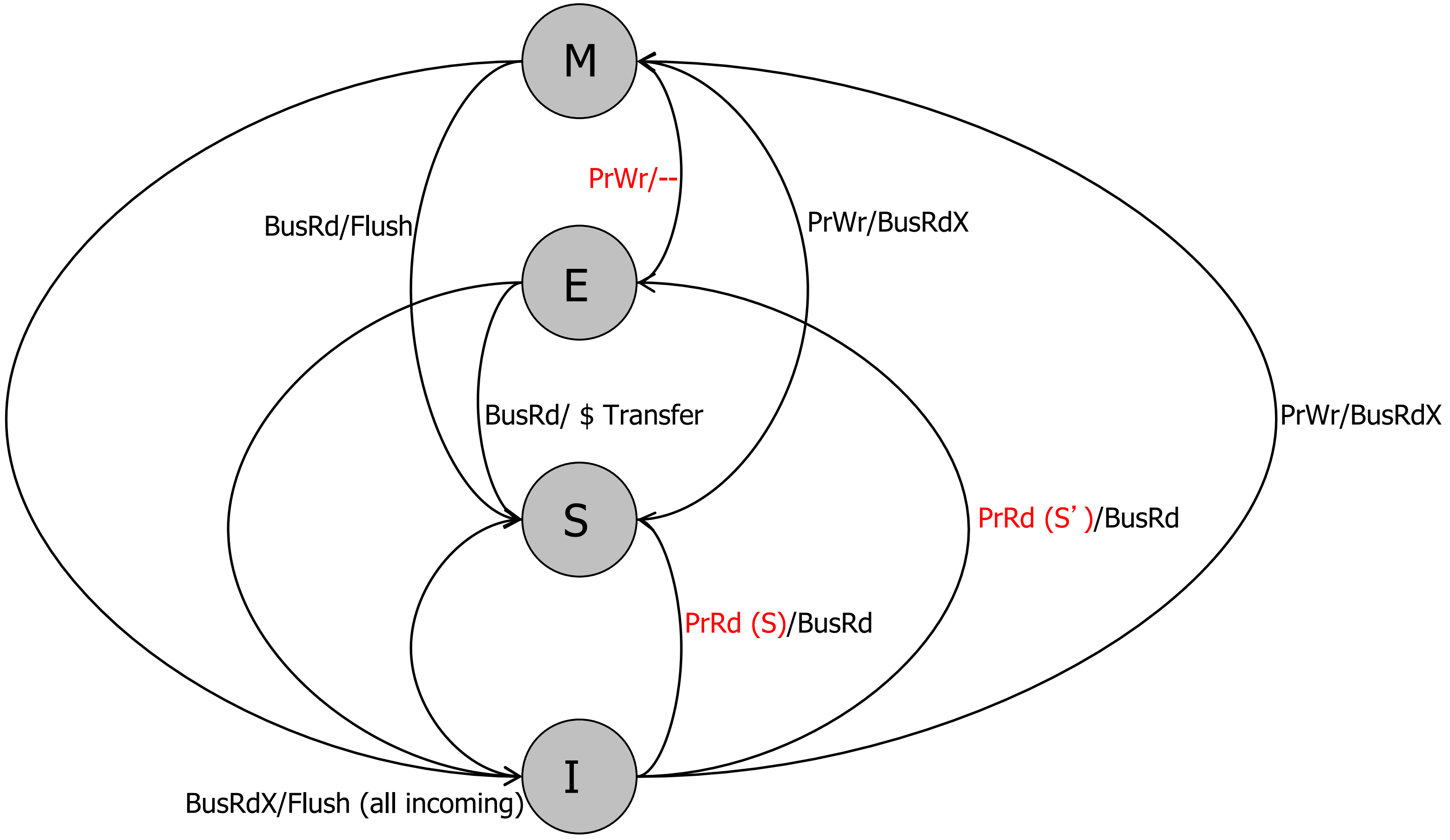
- Idea: Add another state indicating that this is the only cached copy and it is clean.
  - *Exclusive* state

- 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!

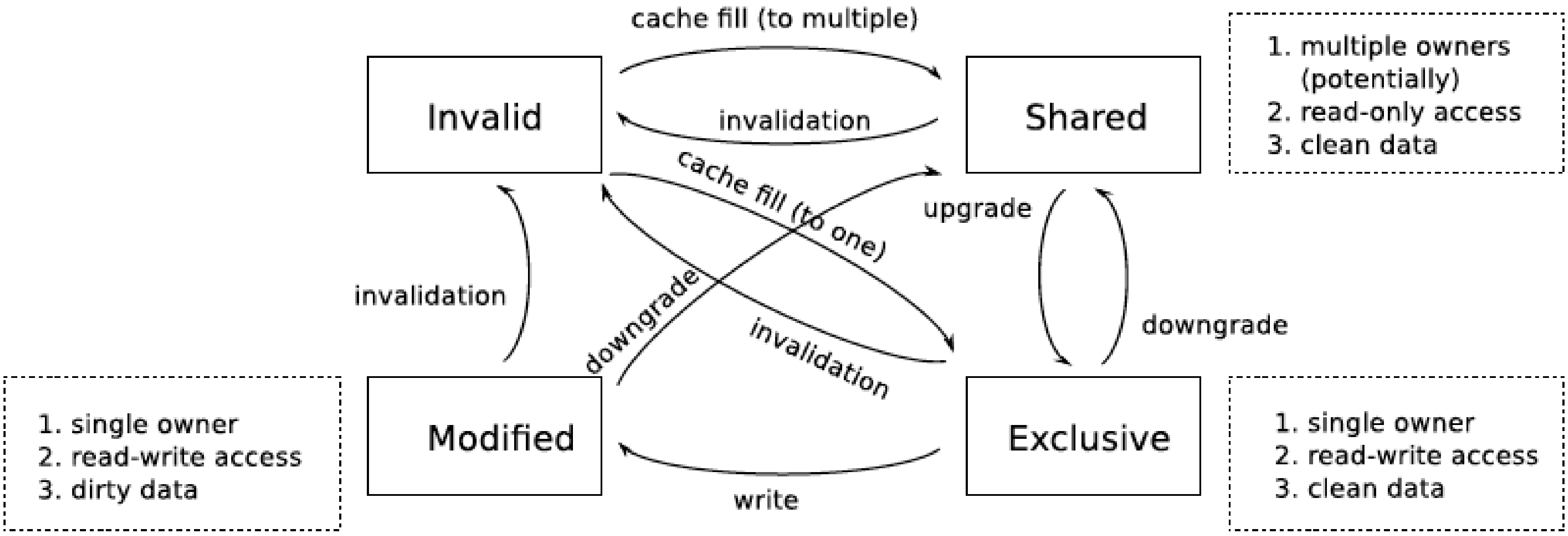
- MESI is also called the *Illinois protocol*
  - Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.

MESI State Machine



[Culler/Singh96]

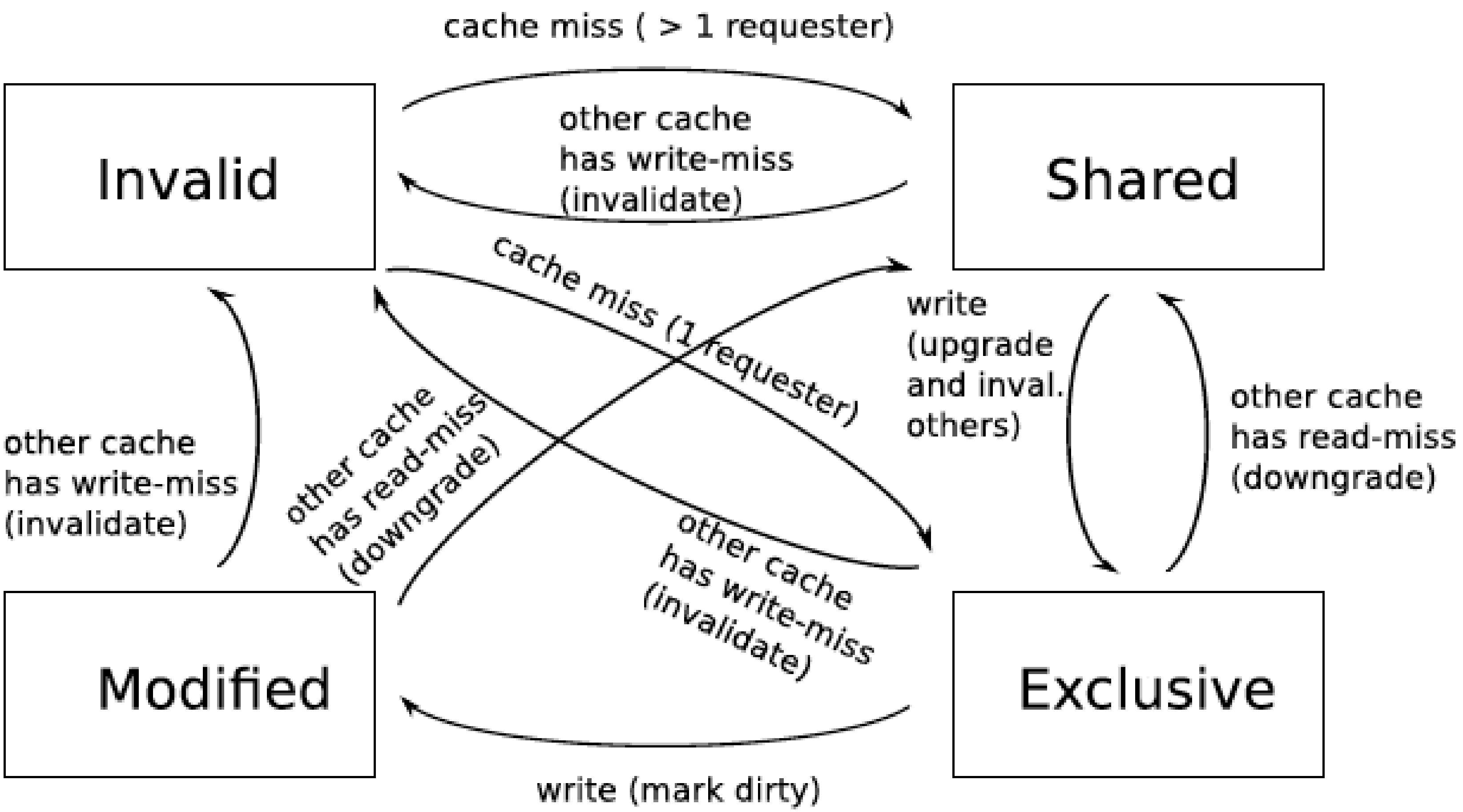
MESI State Machine



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.

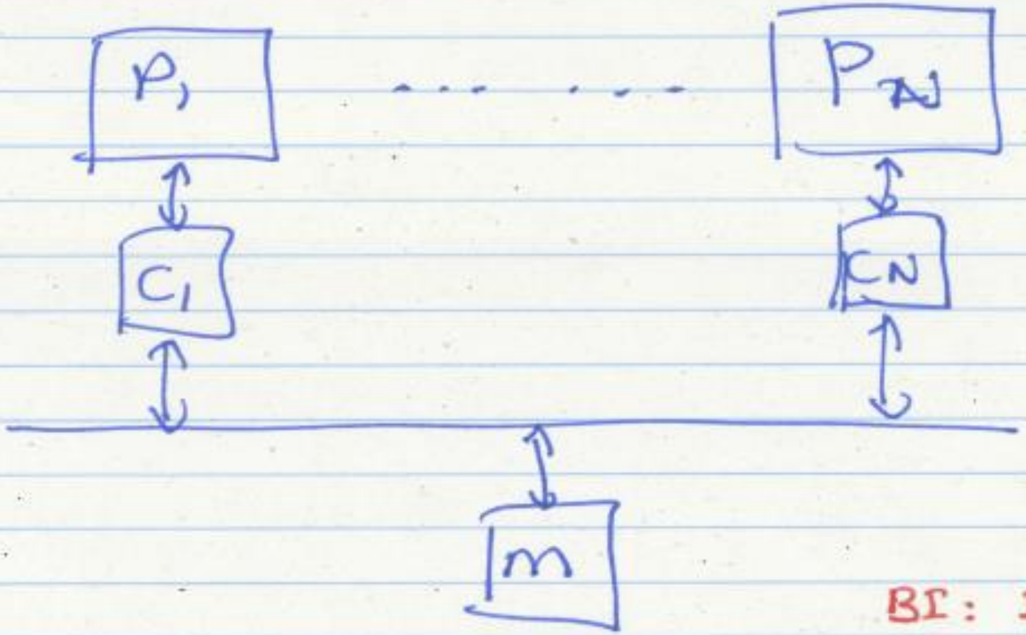
MESI State Machine



Invalid
Shared
Modified
Exclusive
cache miss (> 1 requester)
other cache has write-miss (invalidate)
cache miss (1 requester)
write (upgrade and inval. others)
other cache has read-miss (downgrade)
other cache has write-miss (invalidate)
other cache has read-miss (downgrade)
other cache has write-miss (invalidate)
write (mark dirty)

Papamarcos & Patel, ISCA 1984

Illinois Protocol



↓ PR, PW

↑ BR

BW → BI
BW → BRI

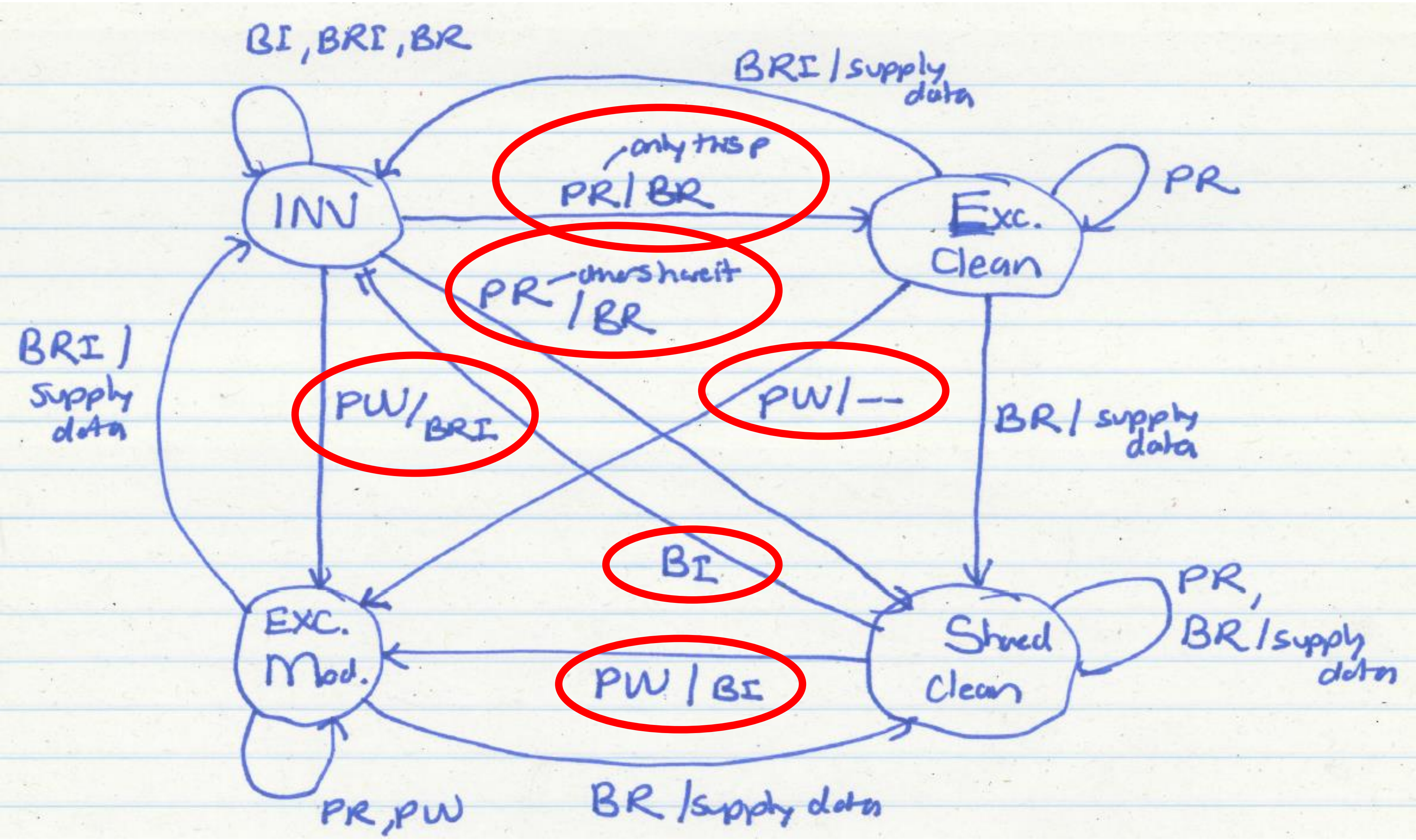
BI : Invalidate, but
already have the data
(do not supply it)

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

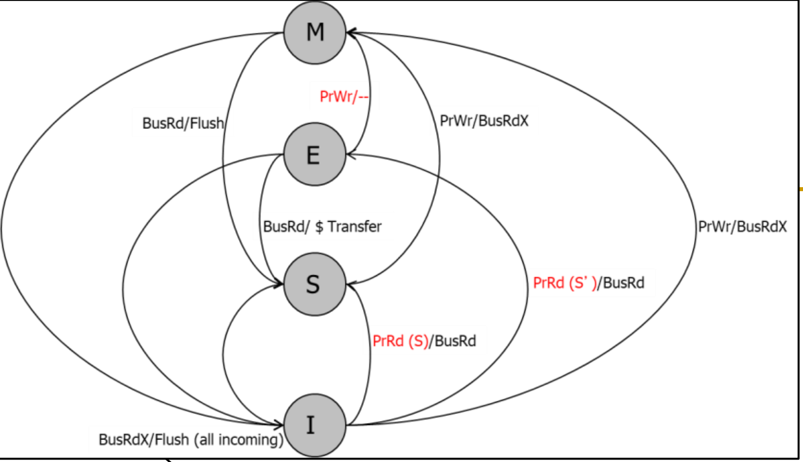
4 States

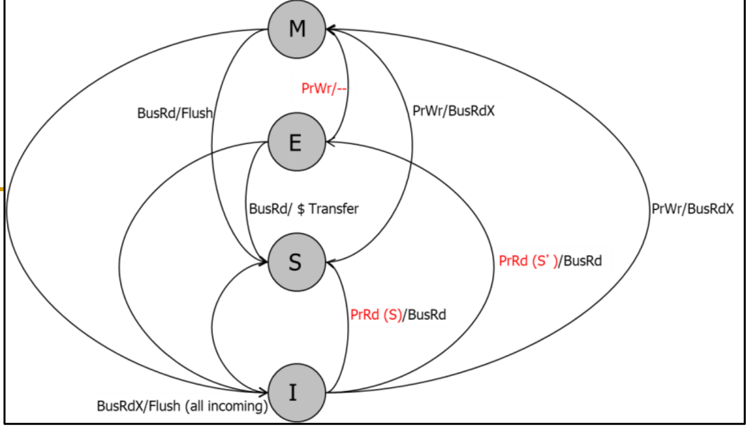
M : Modified (Exclusive copy, modified)
E : Exclusive (" " , clean)
S : Shared (Shared copy, clean)
I : Invalid

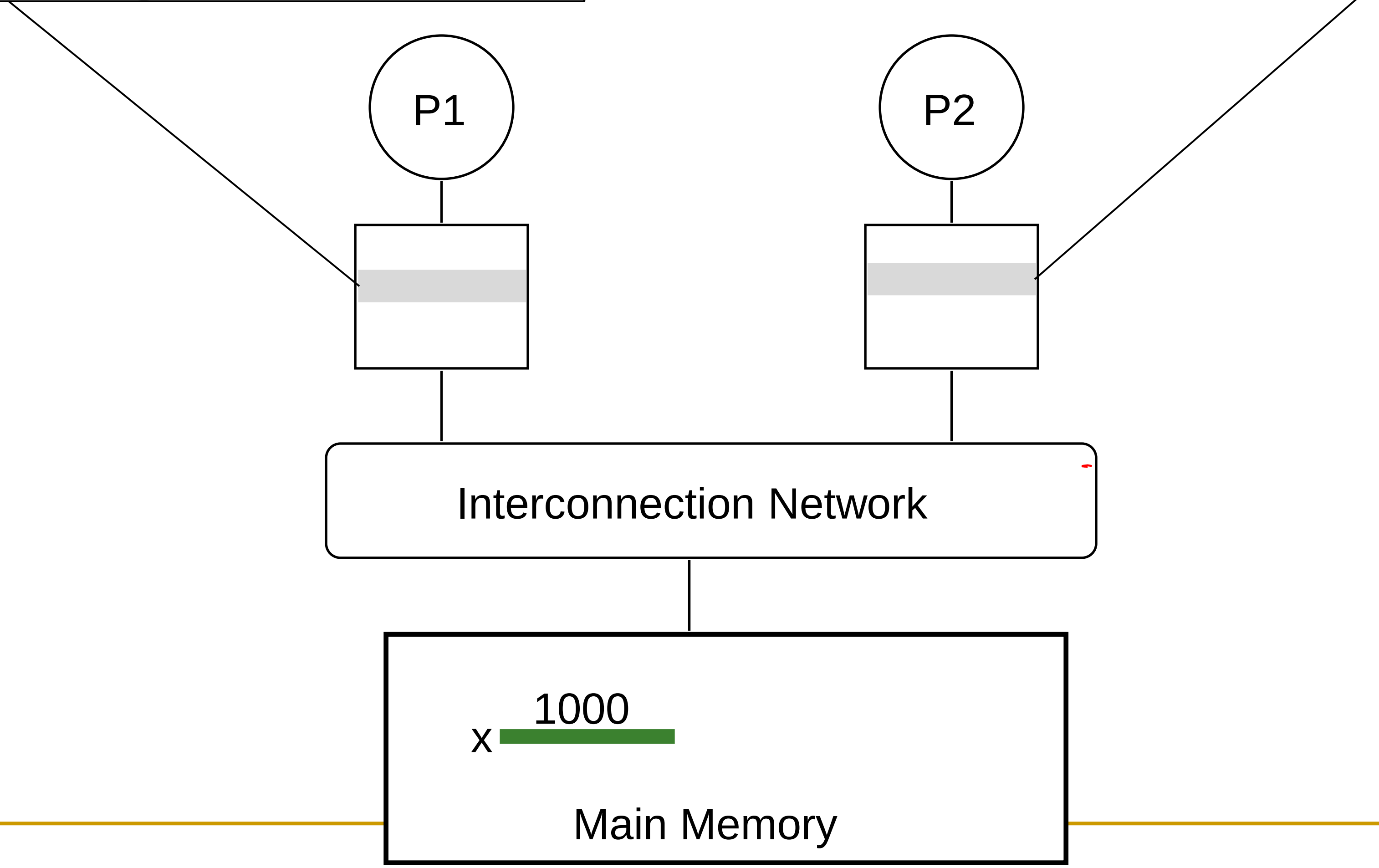
MESI State Machine



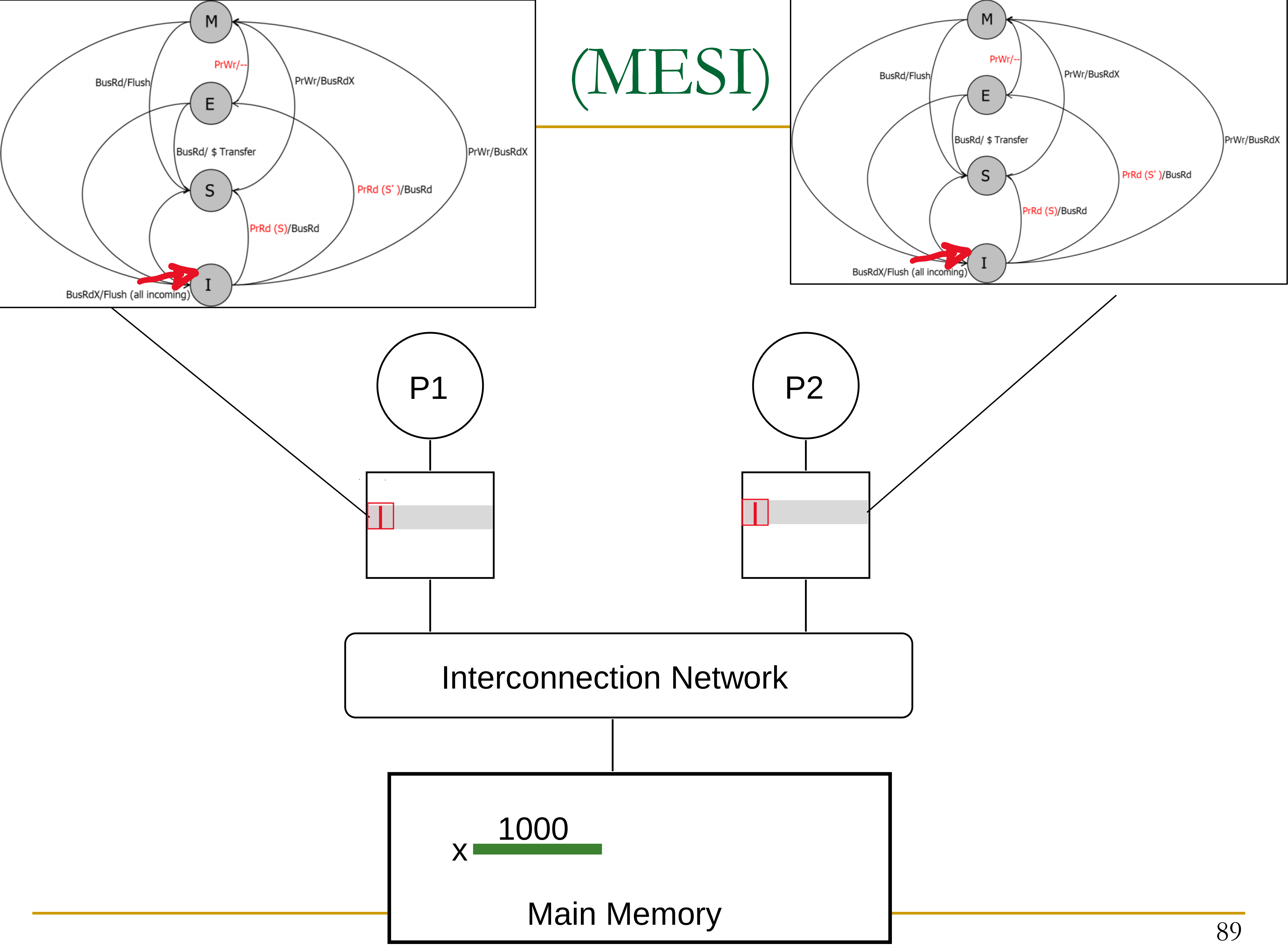
(MESI)


M
E
S
I
PrWr/--
BusRd/Flush
PrWr/BusRdX
BusRd/ $ Transfer
PrWr/BusRdX
PrRd (S')/BusRd
PrRd (S)/BusRd
BusRdX/Flush (all incoming)

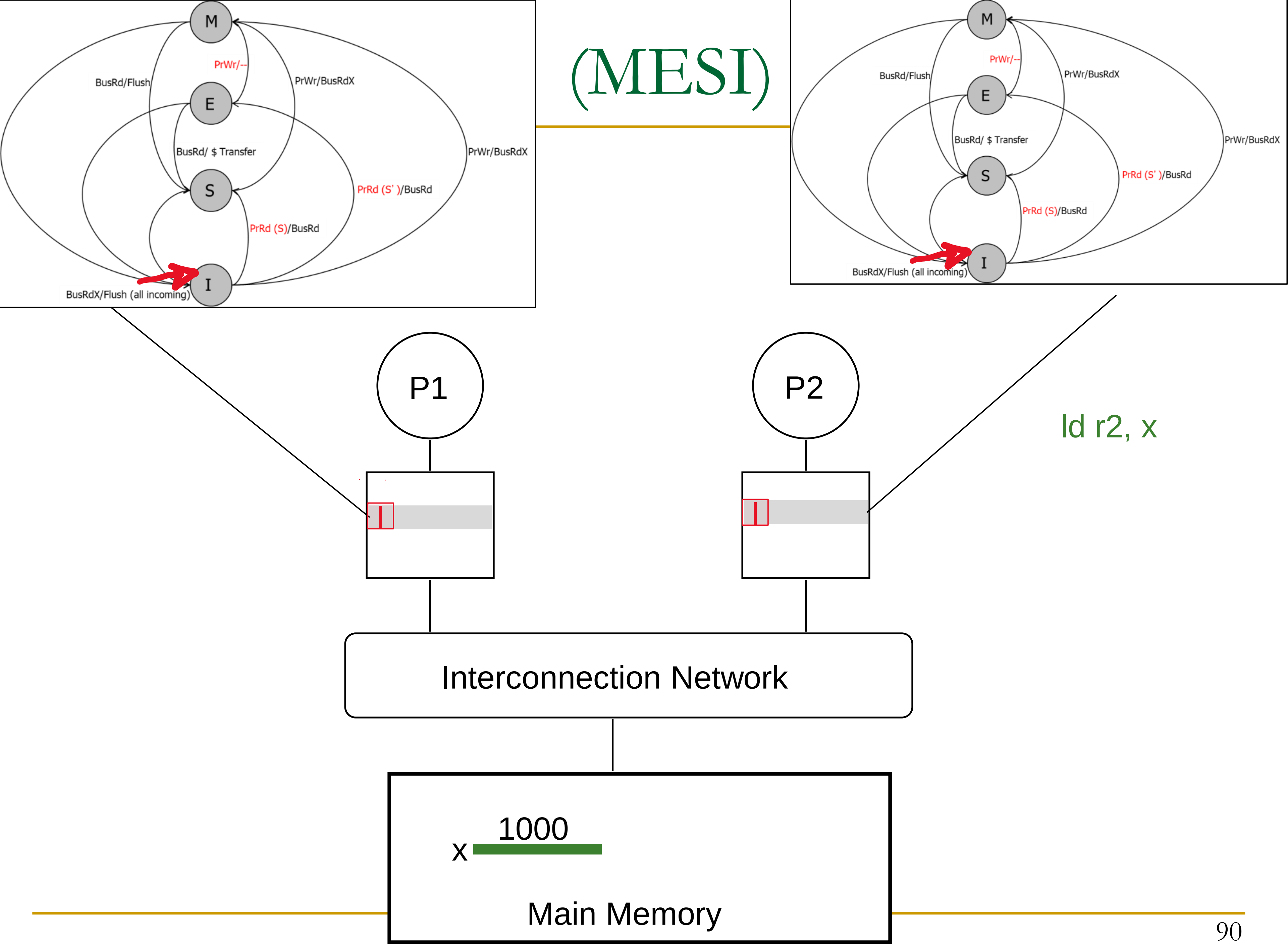

M
E
S
I
PrWr/--
BusRd/Flush
PrWr/BusRdX
BusRd/ $ Transfer
PrWr/BusRdX
PrRd (S')/BusRd
PrRd (S)/BusRd
BusRdX/Flush (all incoming)


P1
P2
Interconnection Network
x
1000
Main Memory

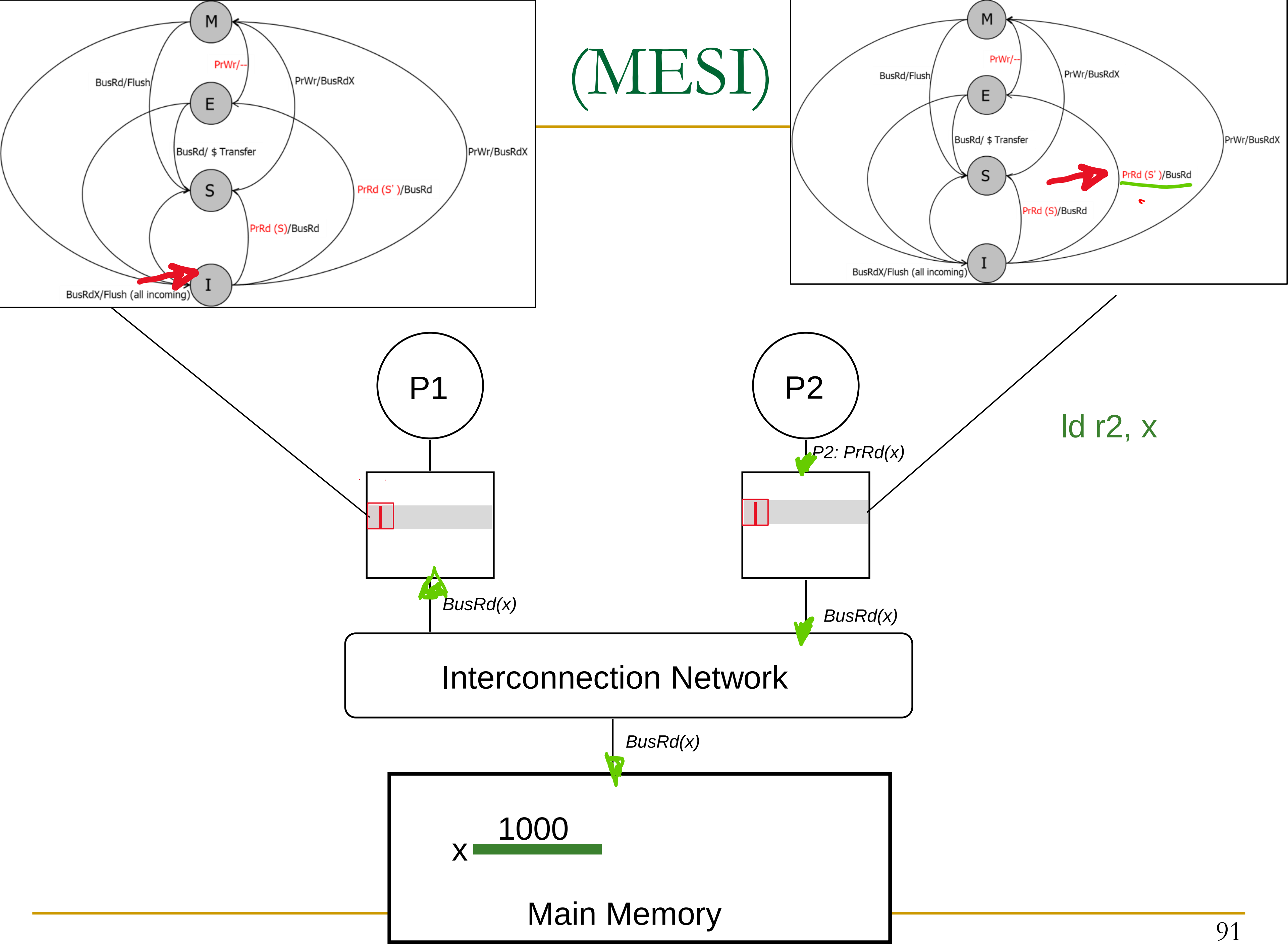
(MESI)


M
E
S
I
PrWr/--
BusRd/Flush
PrWr/BusRdX
BusRd/ $ Transfer
PrWr/BusRdX
PrRd (S')/BusRd
PrRd (S)/BusRd
BusRdX/Flush (all incoming)
P1
P2
I
I
Interconnection Network
X
1000
Main Memory

(MESI)


M
PrWr/--
BusRd/Flush
PrWr/BusRdX
E
BusRd/ $ Transfer
PrWr/BusRdX
S
PrRd (S')/BusRd
PrRd (S)/BusRd
I
BusRdX/Flush (all incoming)
M
PrWr/--
BusRd/Flush
PrWr/BusRdX
E
BusRd/ $ Transfer
PrWr/BusRdX
S
PrRd (S')/BusRd
PrRd (S)/BusRd
I
BusRdX/Flush (all incoming)
P1
P2
ld r2, x
I
I
Interconnection Network
x
1000
Main Memory

(MESI)



ld r2, x

P2: PrRd(x)

BusRd(x)

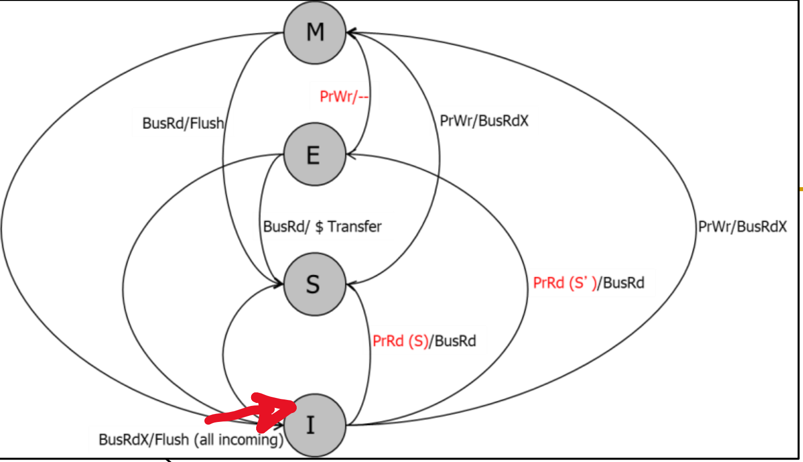
BusRd(x)

Interconnection Network

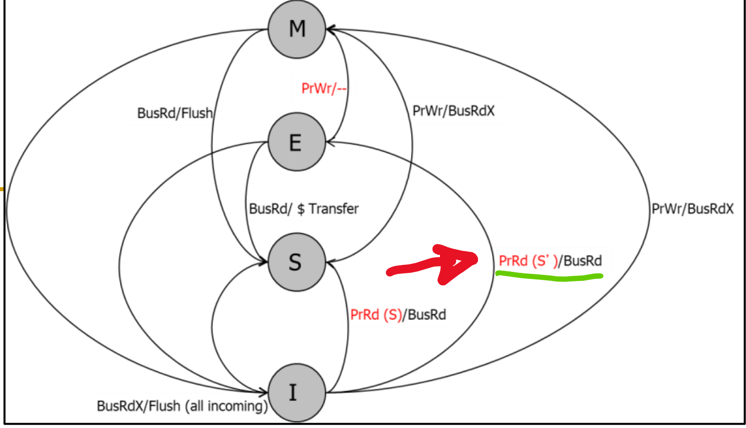
BusRd(x)

x 1000

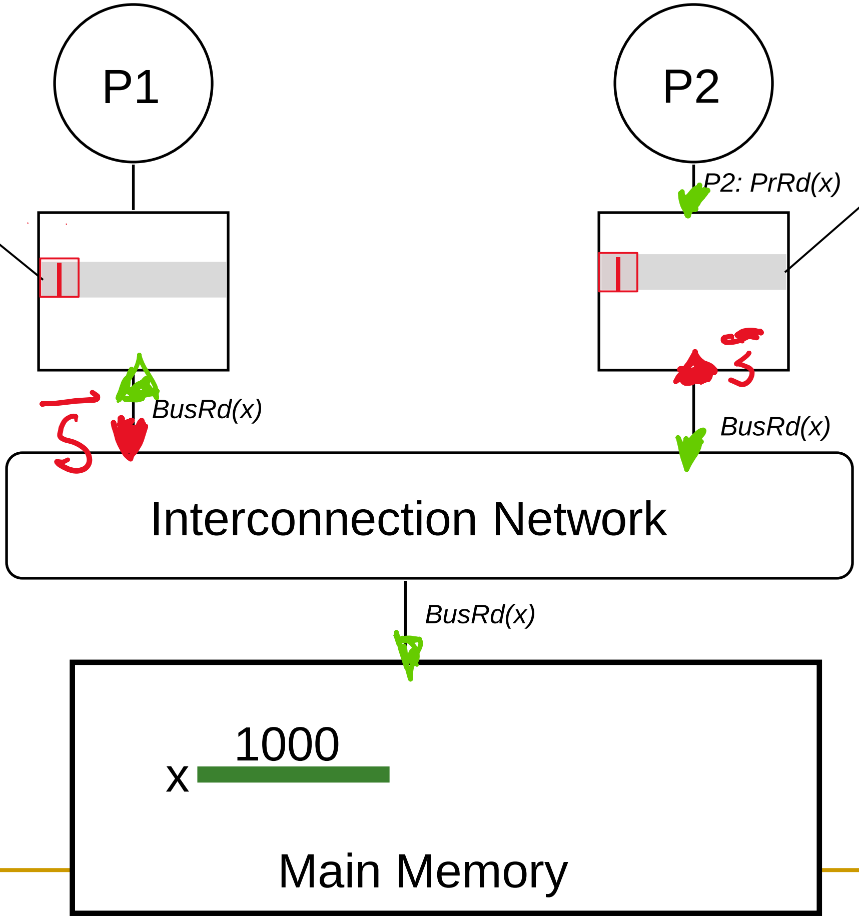
Main Memory



(MESI)



ld r2, x


P1

P2

P2: PrRd(x)

I

I

BusRd(x)

BusRd(x)

S

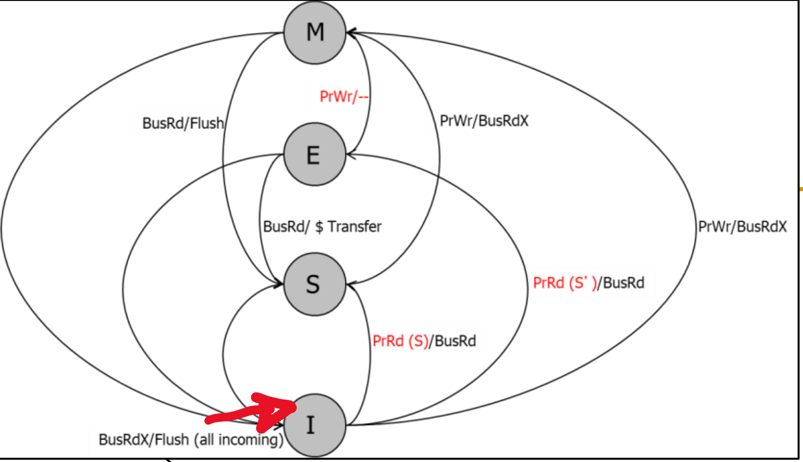
S

Interconnection Network

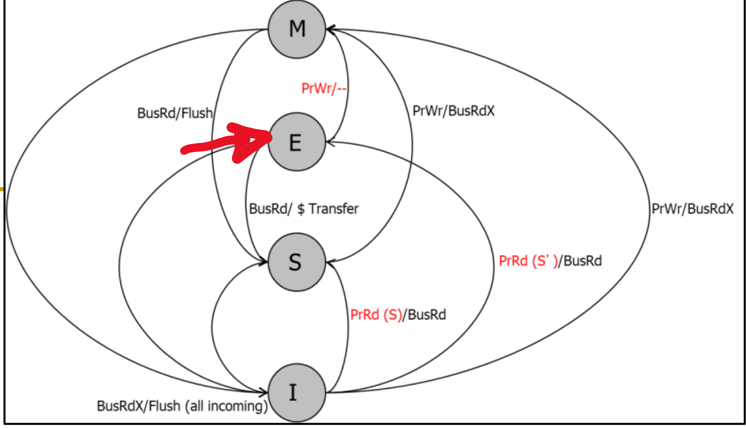
BusRd(x)

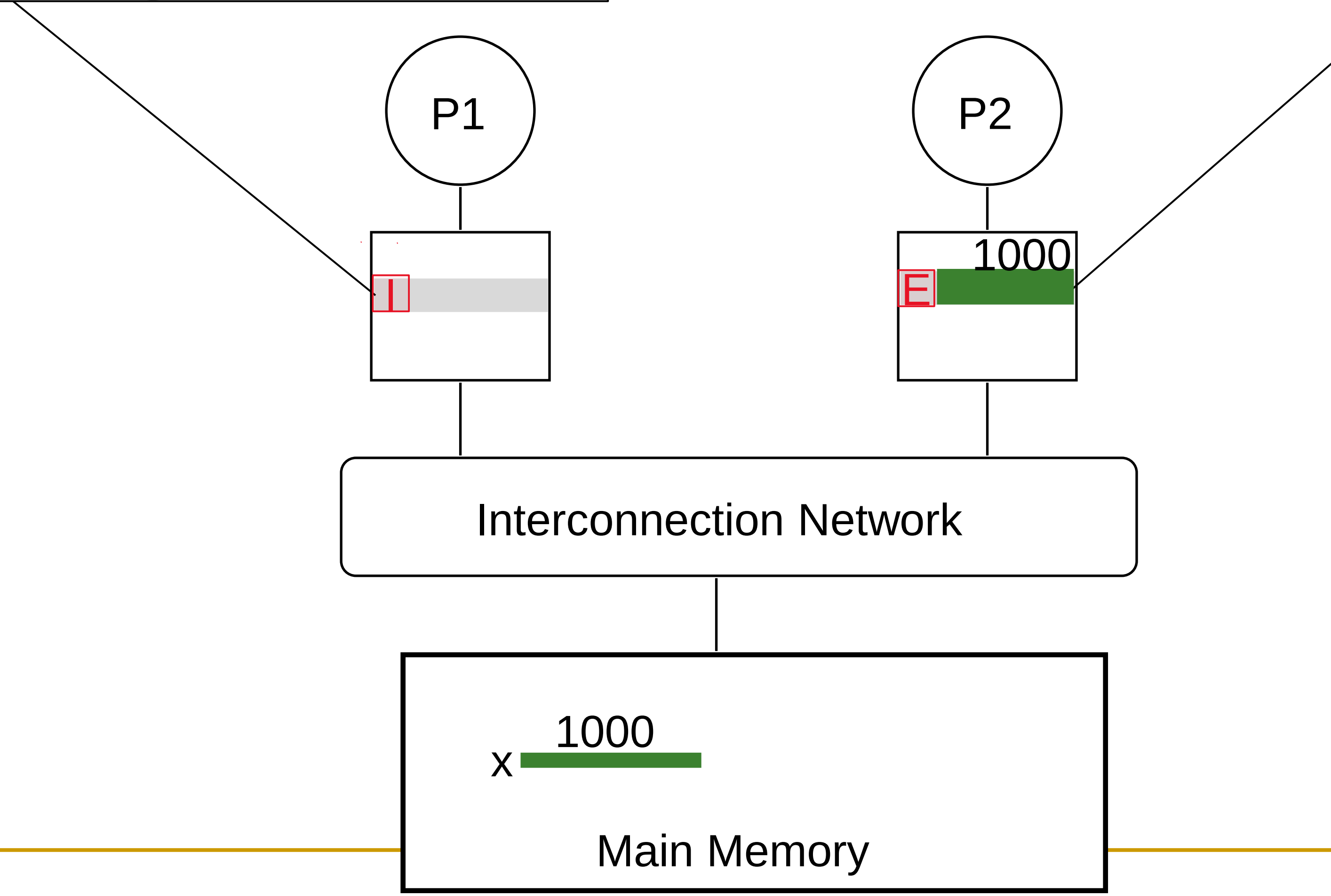
x 1000

Main Memory


M
E
S
I
PrWr/--
BusRd/Flush
PrWr/BusRdX
BusRd/ $ Transfer
PrWr/BusRdX
PrRd (S')/BusRd
PrRd (S)/BusRd
BusRdX/Flush (all incoming)

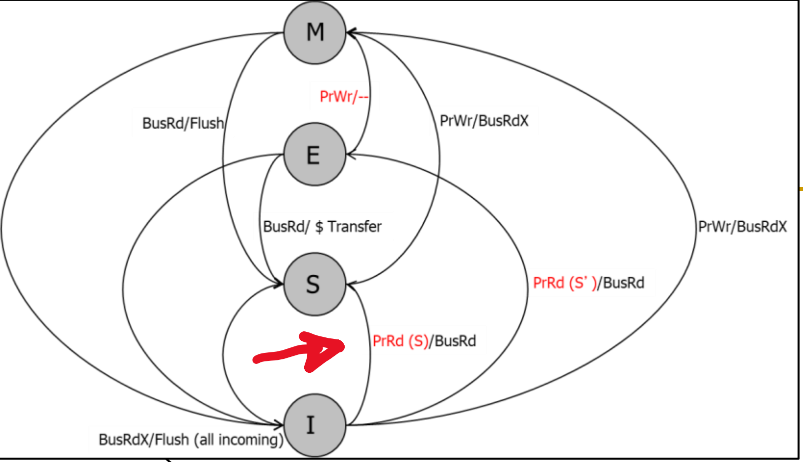
(MESI)

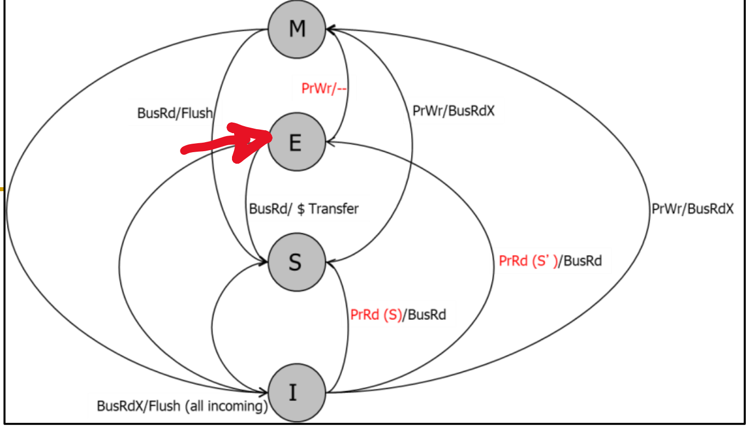

M
E
S
I
PrWr/--
BusRd/Flush
PrWr/BusRdX
BusRd/ $ Transfer
PrWr/BusRdX
PrRd (S')/BusRd
PrRd (S)/BusRd
BusRdX/Flush (all incoming)

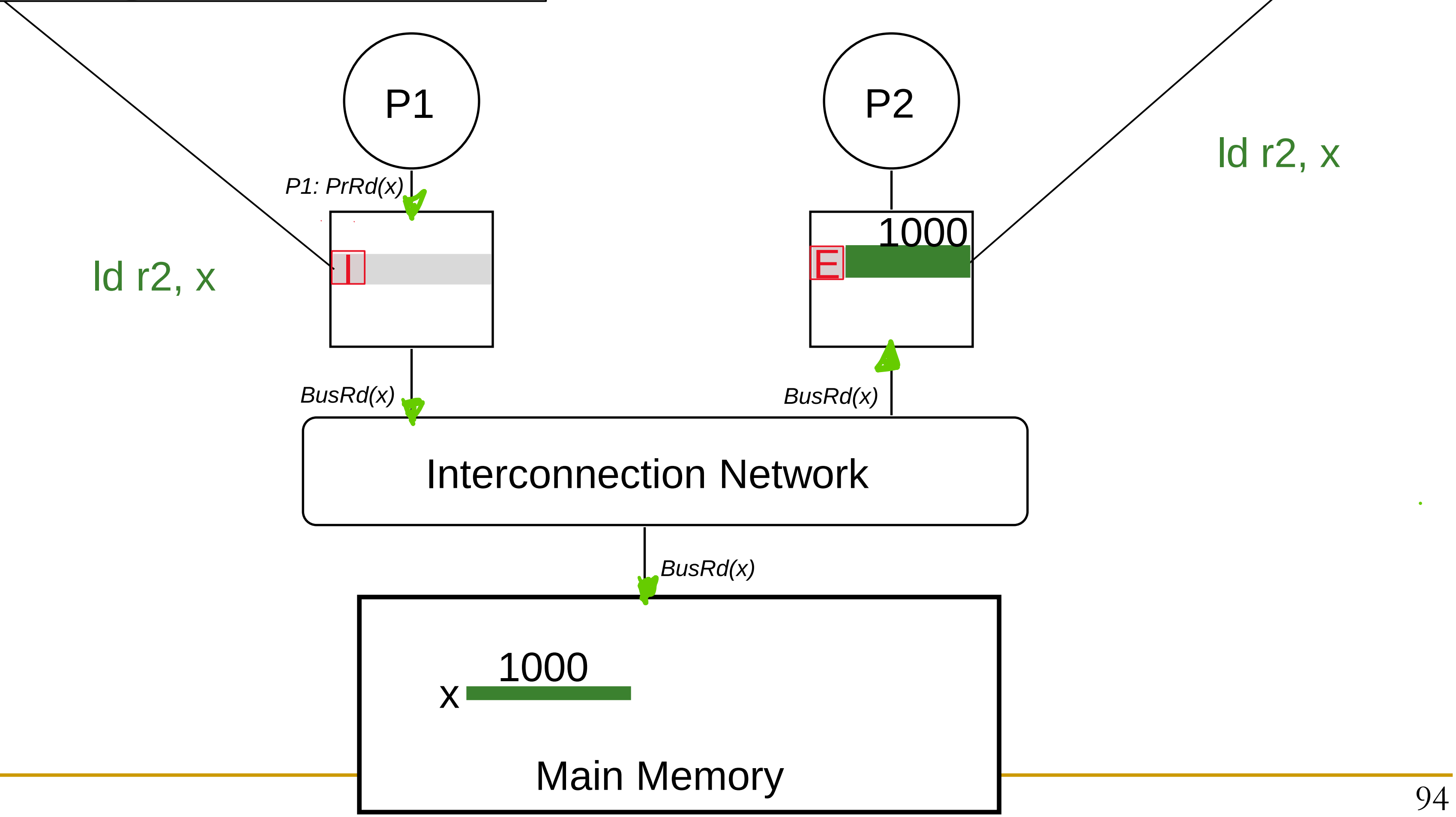

P1
P2
I
E
1000
Interconnection Network
x
1000
Main Memory

ld r2, x

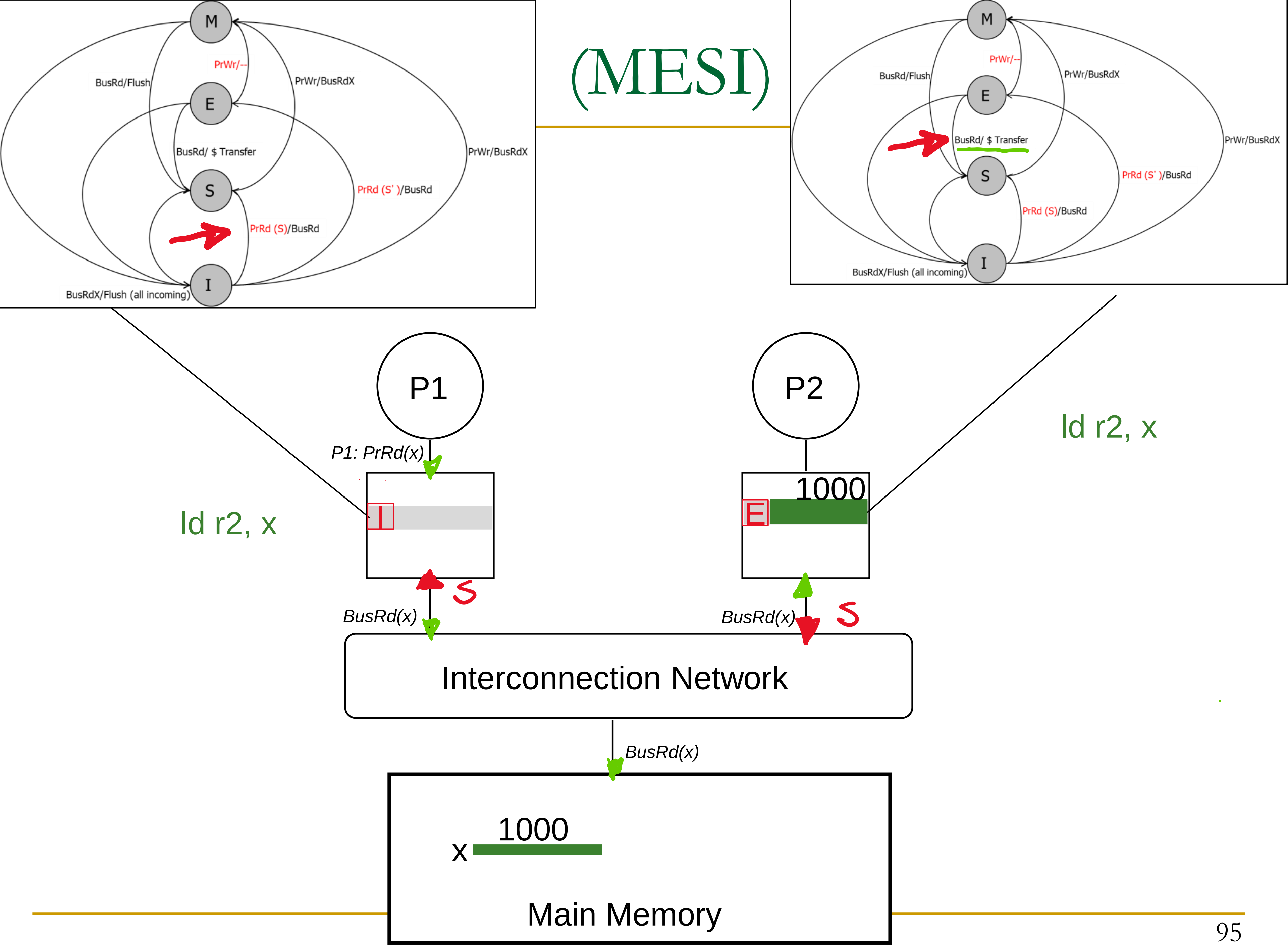
(MESI)


M
E
S
I
PrWr/--
BusRd/Flush
PrWr/BusRdX
BusRd/ $ Transfer
PrWr/BusRdX
PrRd (S')/BusRd
PrRd (S)/BusRd
BusRdX/Flush (all incoming)

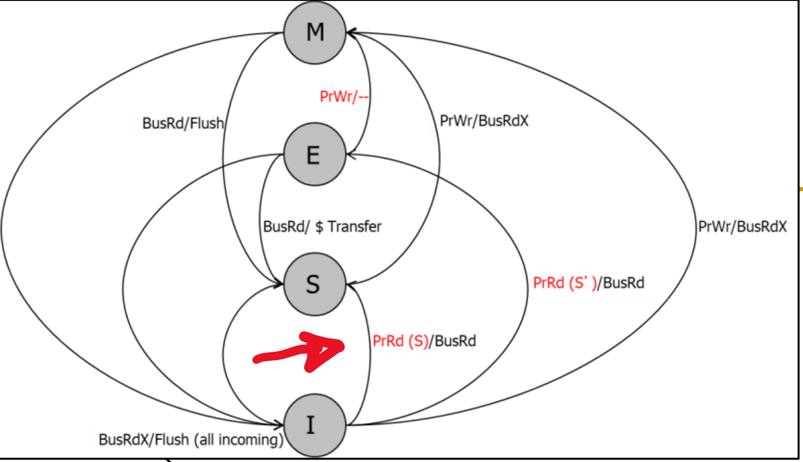

M
E
S
I
PrWr/--
BusRd/Flush
PrWr/BusRdX
BusRd/ $ Transfer
PrWr/BusRdX
PrRd (S')/BusRd
PrRd (S)/BusRd
BusRdX/Flush (all incoming)

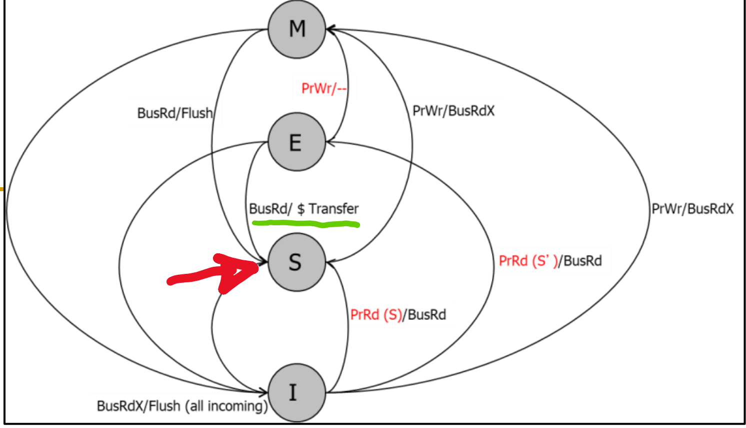

P1
P2
ld r2, x
ld r2, x
P1: PrRd(x)
I
1000
E
BusRd(x)
BusRd(x)
Interconnection Network
BusRd(x)
x
1000
Main Memory

(MESI)

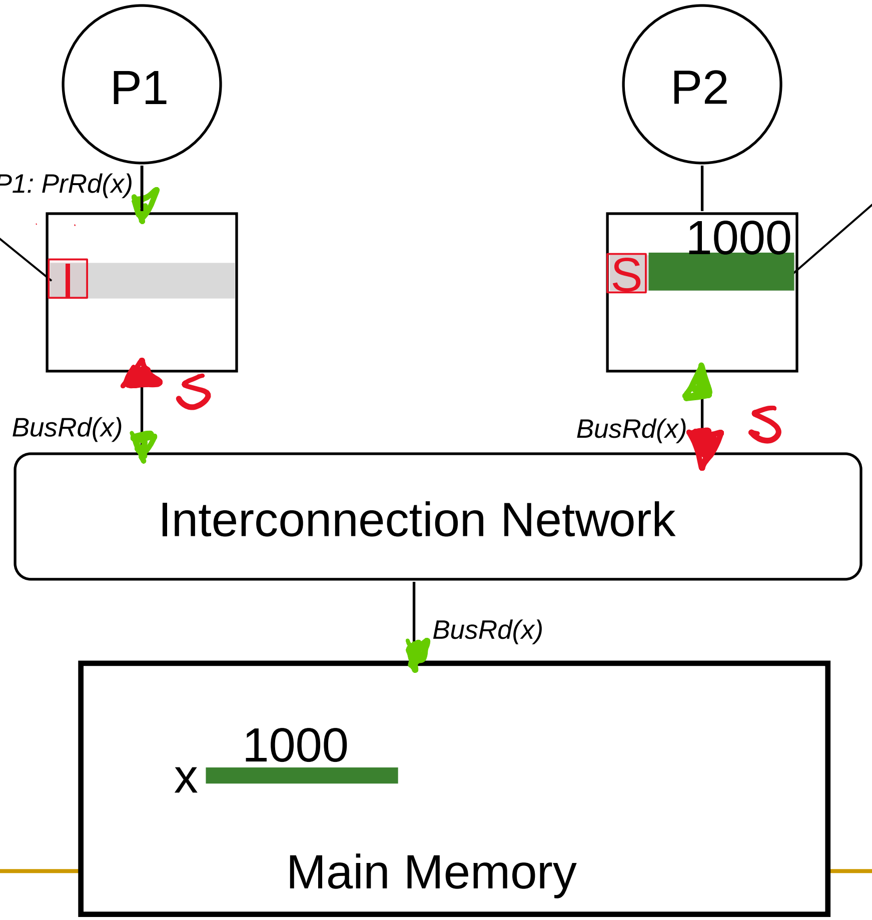

M
E
S
I
PrWr/--
BusRd/Flush
PrWr/BusRdX
BusRd/ $ Transfer
PrWr/BusRdX
PrRd (S')/BusRd
PrRd (S)/BusRd
BusRdX/Flush (all incoming)
M
E
S
I
PrWr/--
BusRd/Flush
PrWr/BusRdX
BusRd/ $ Transfer
PrWr/BusRdX
PrRd (S')/BusRd
PrRd (S)/BusRd
BusRdX/Flush (all incoming)
P1
P2
ld r2, x
ld r2, x
P1: PrRd(x)
I
1000
E
S
S
BusRd(x)
BusRd(x)
Interconnection Network
BusRd(x)
x
1000
Main Memory

(MESI)



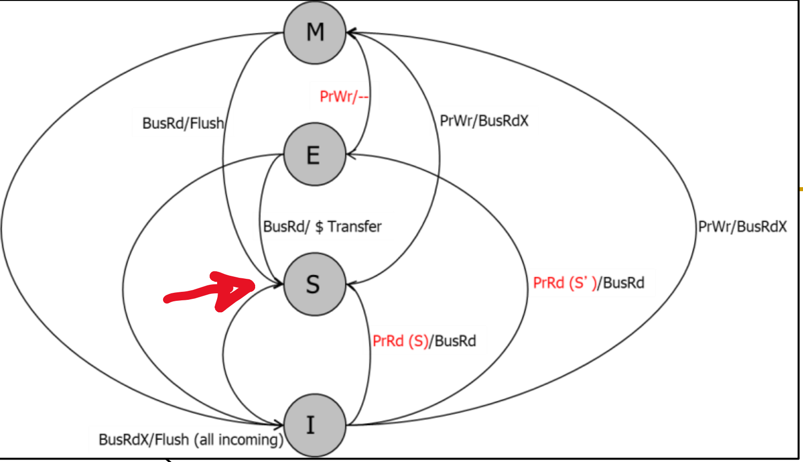


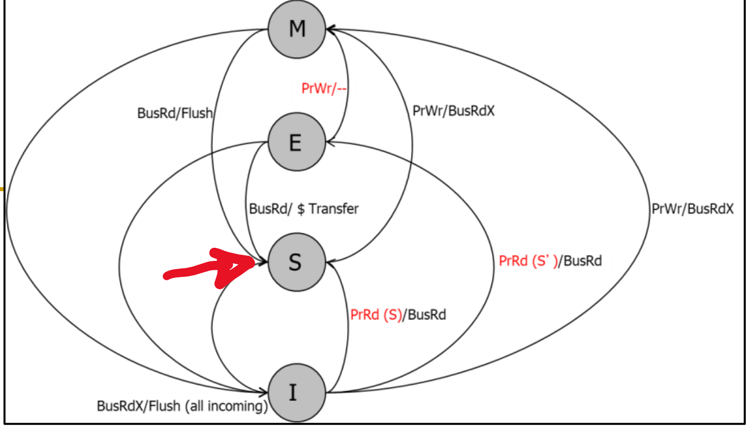
ld r2, x

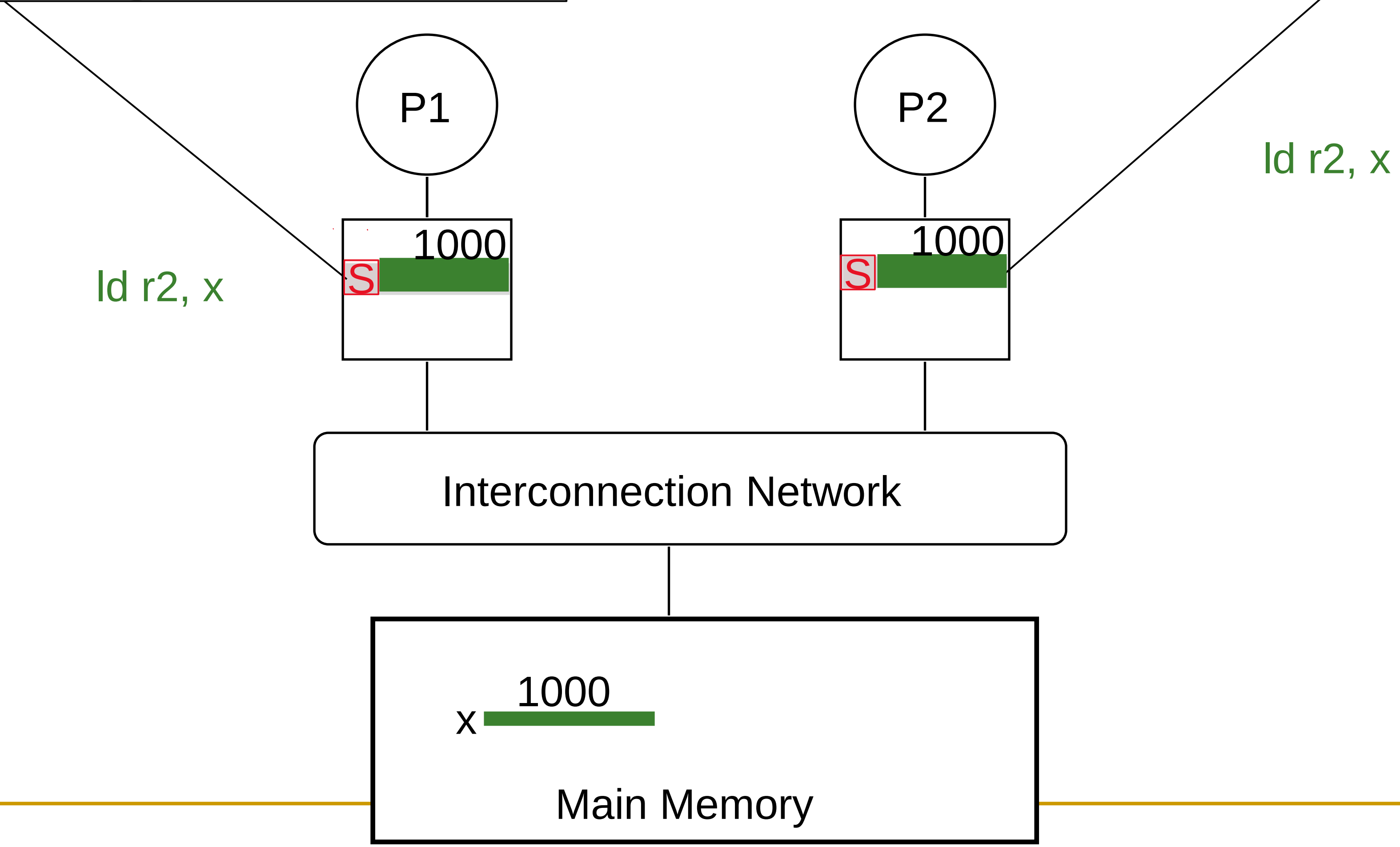


ld r2, x

(MESI)







P1

P2

ld r2, x

ld r2, x

S 1000

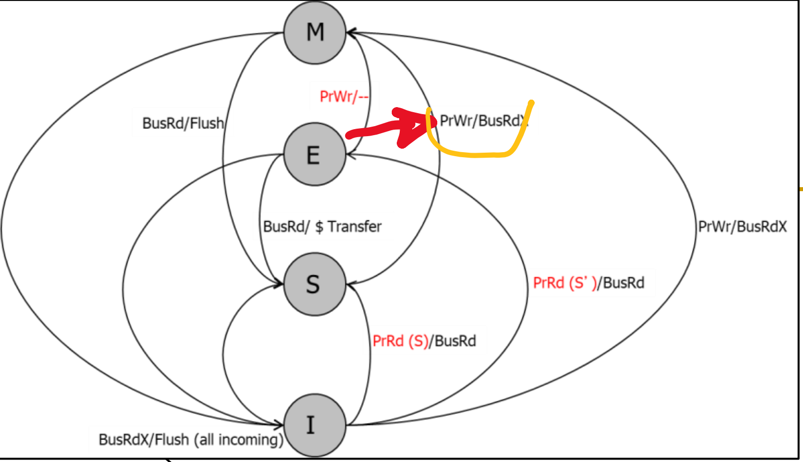
S 1000

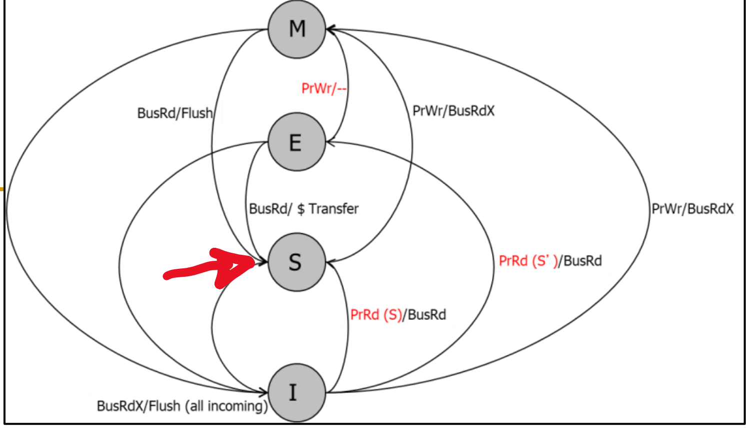
Interconnection Network

x 1000

Main Memory

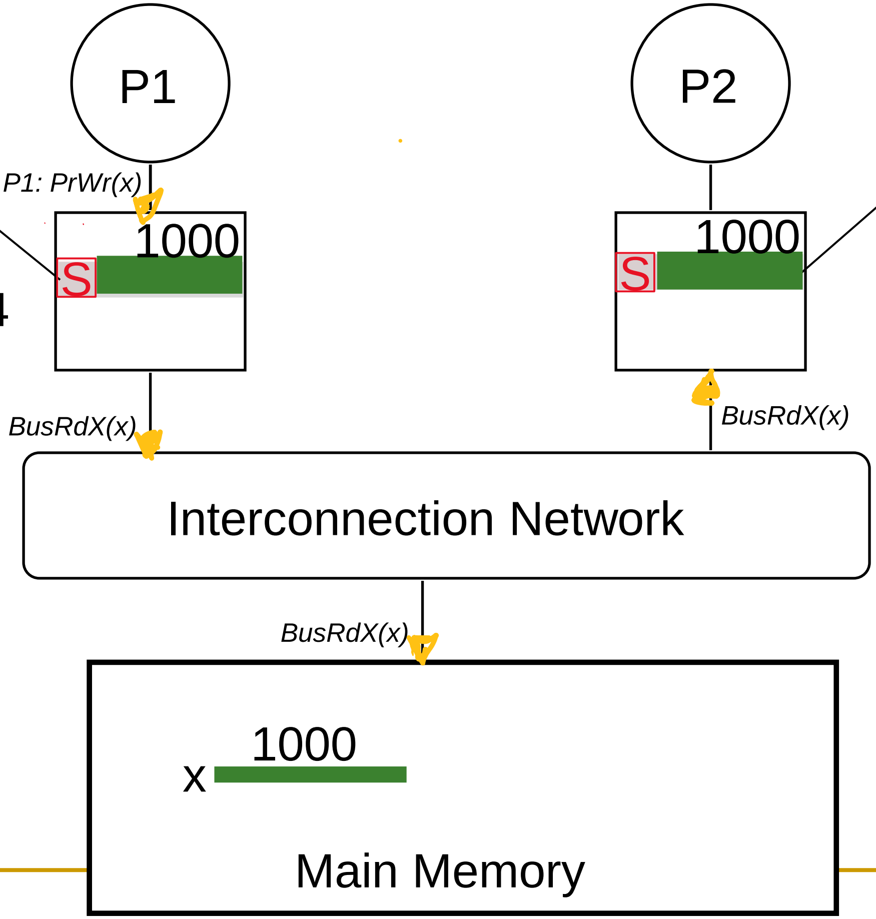
(MESI)



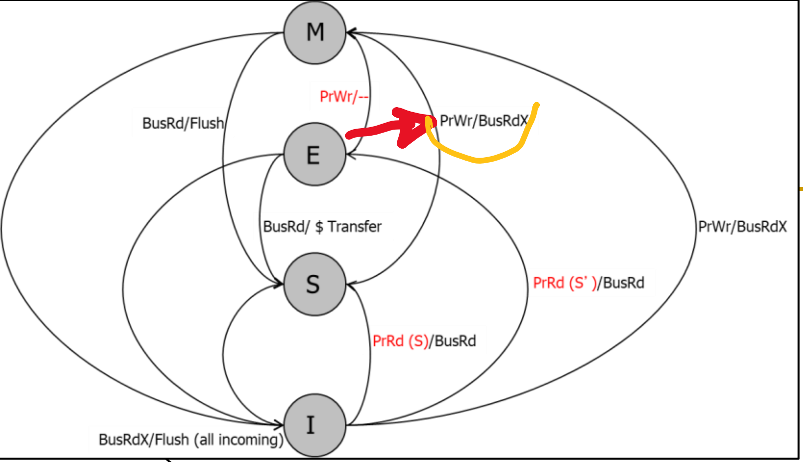


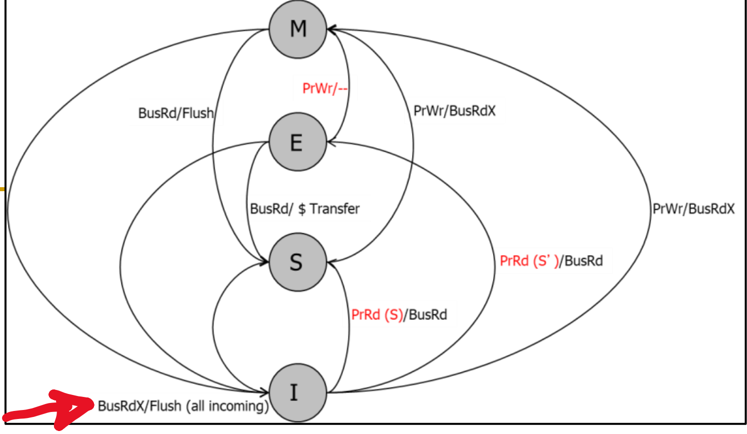
ld r2, x
add r1, r2, r4
st x, r1

ld r2, x



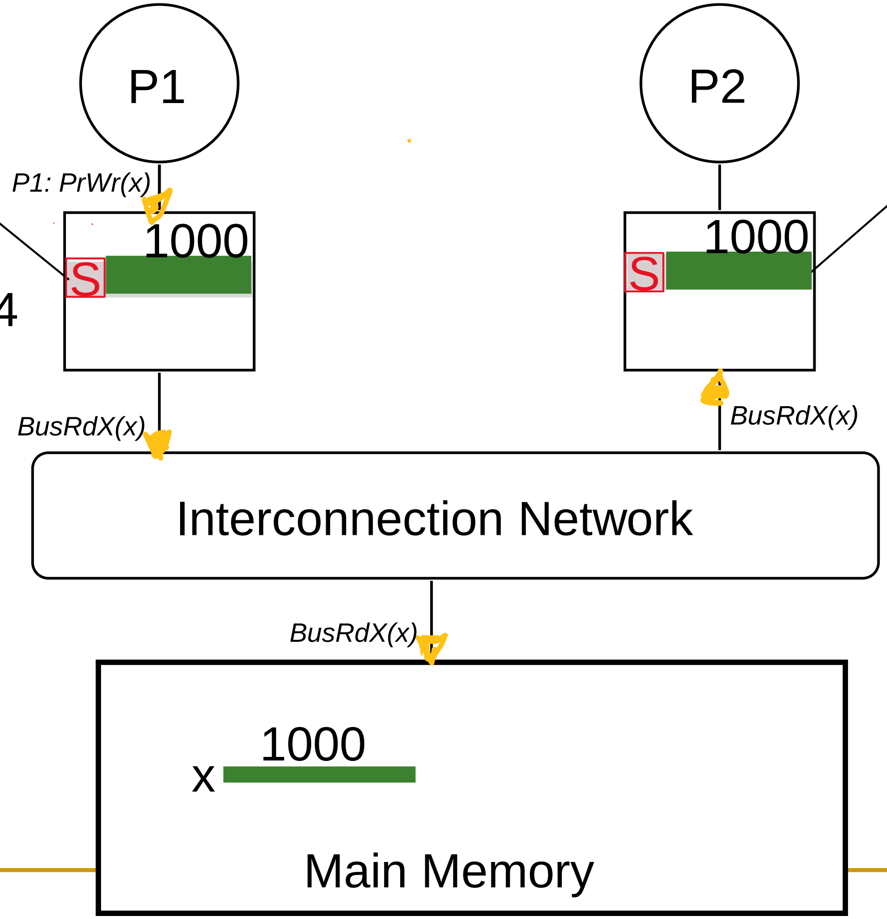
(MESI)





ld r2, x
add r1, r2, r4
st x, r1

ld r2, x



P1

P2

P1: PrWr(x)

S 1000

S 1000

BusRdX(x)

BusRdX(x)

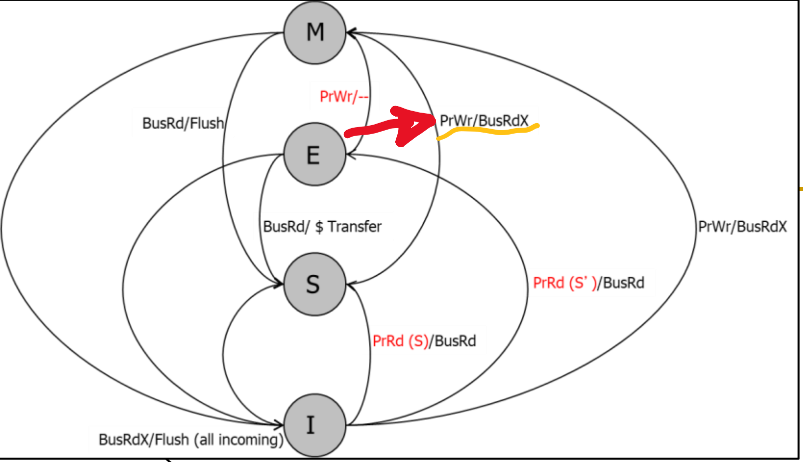
Interconnection Network

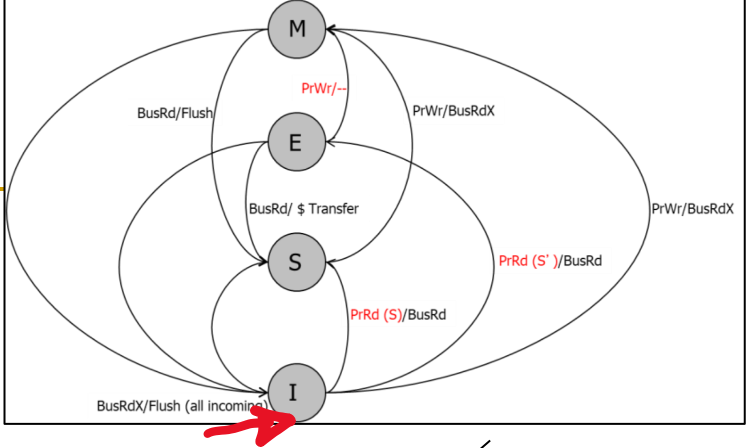
BusRdX(x)

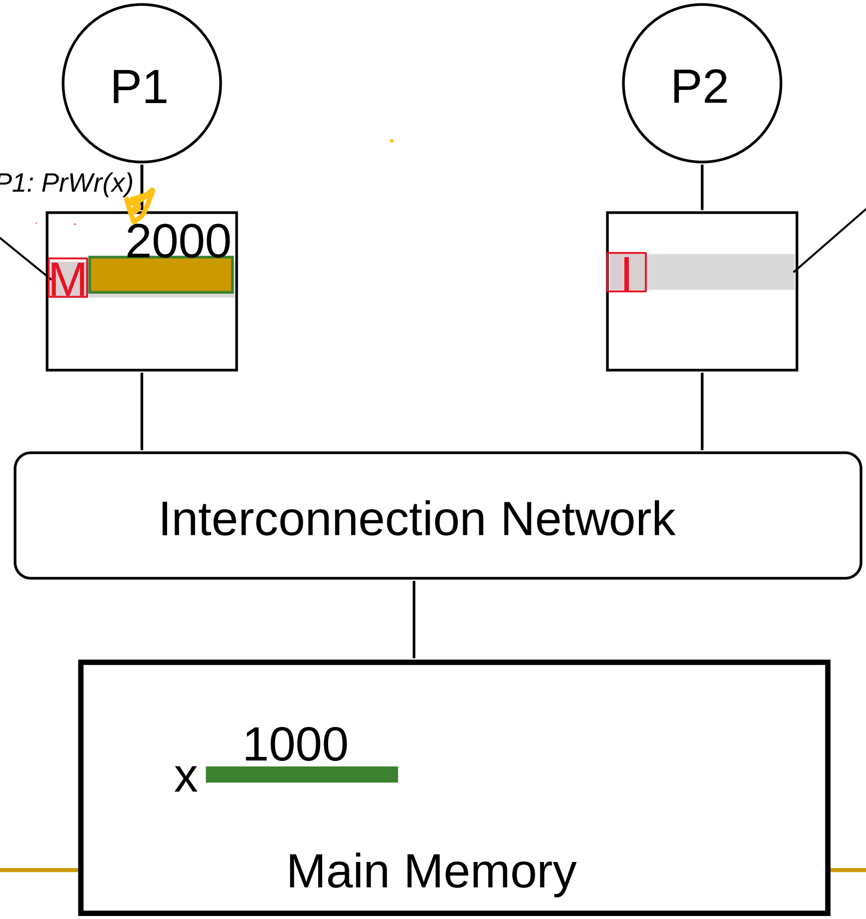
x 1000

Main Memory

(MESI)



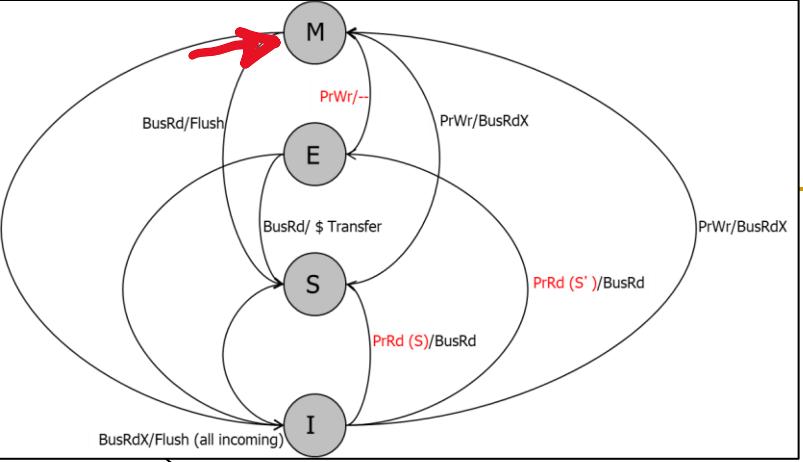


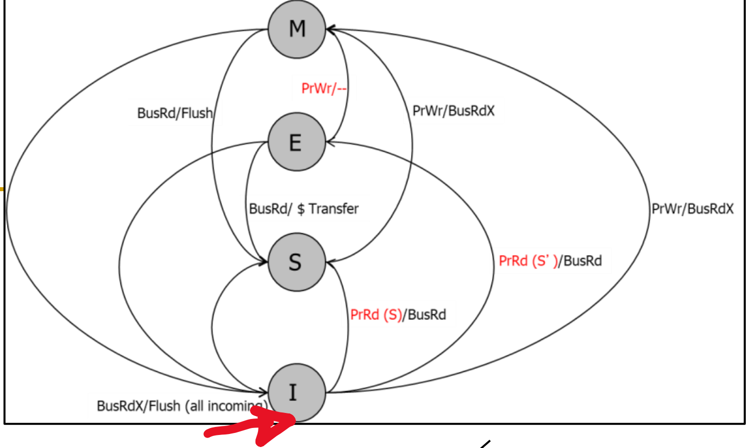


ld r2, x
add r1, r2, r4
st x, r1

ld r2, x

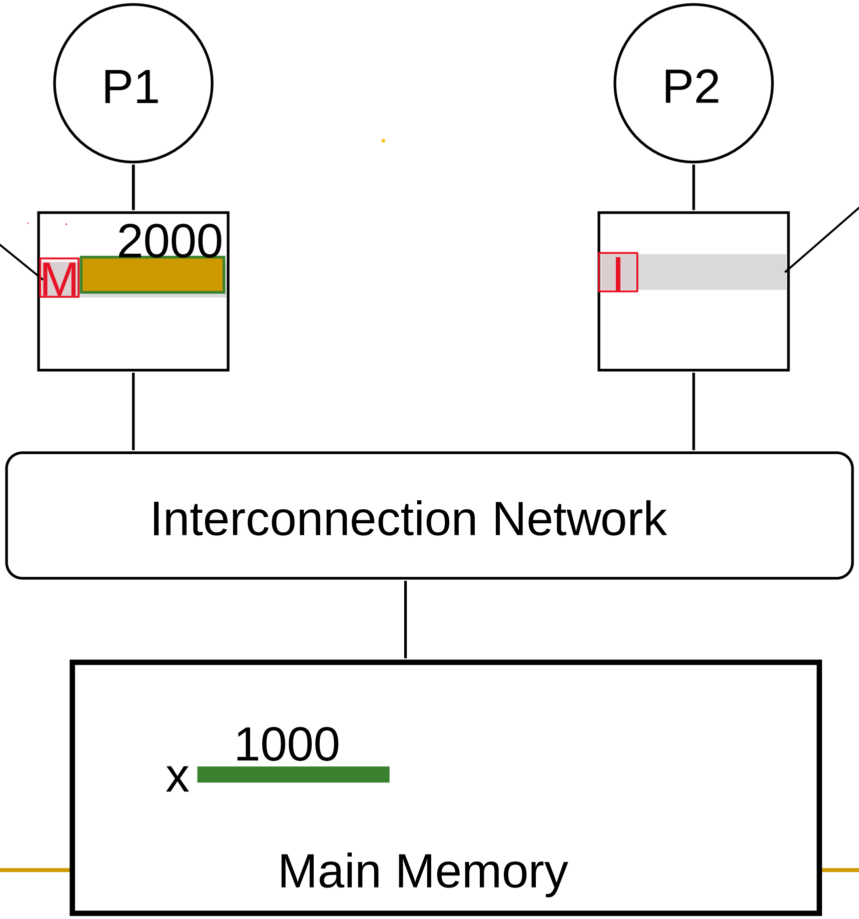
(MESI)



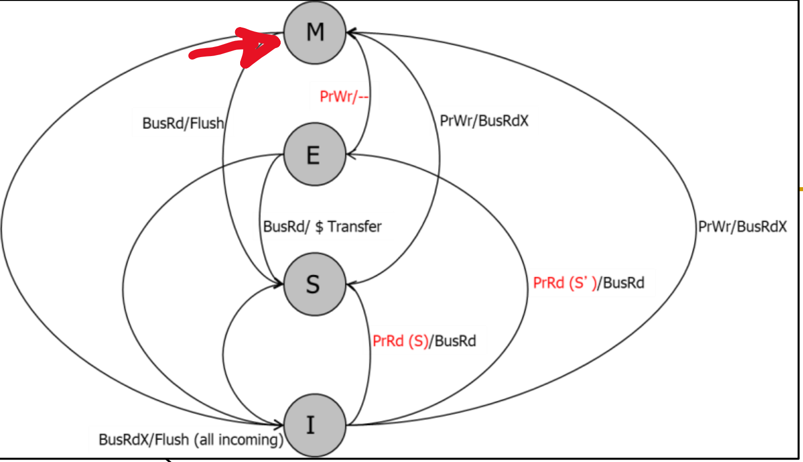


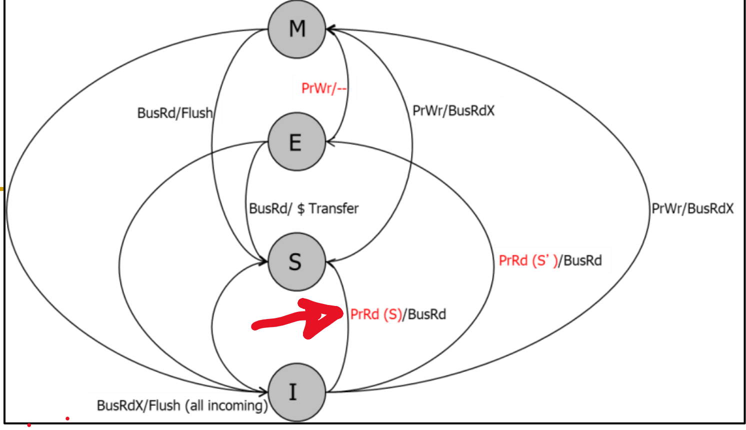
ld r2, x
add r1, r2, r4
st x, r1

ld r2, x

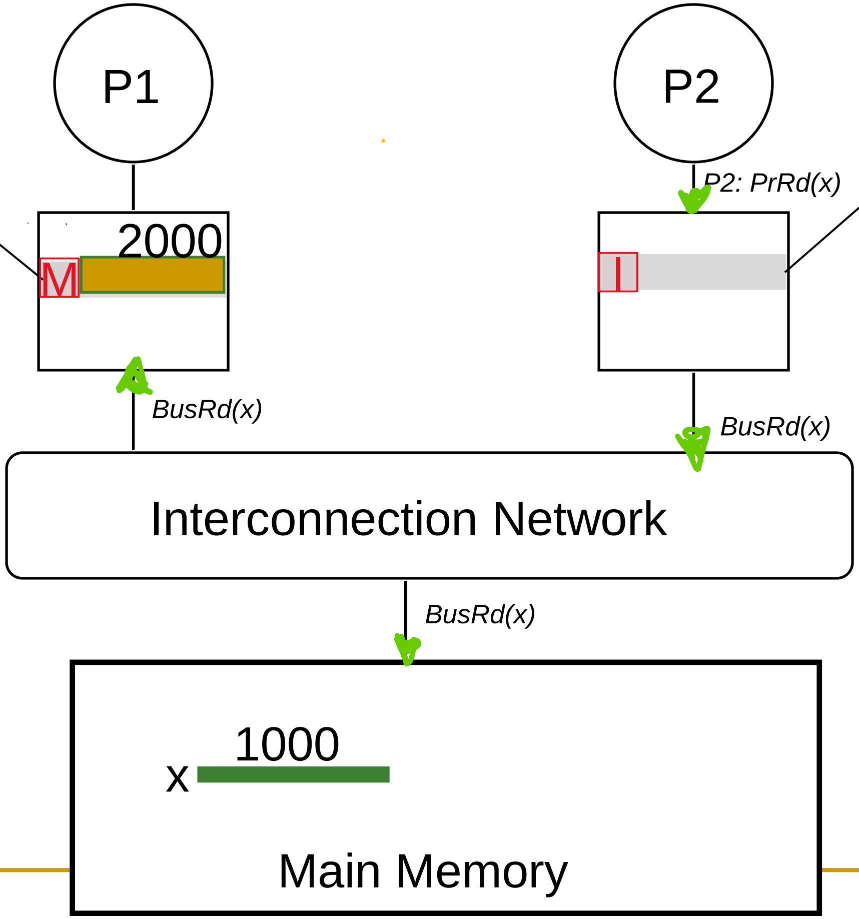


(MESI)



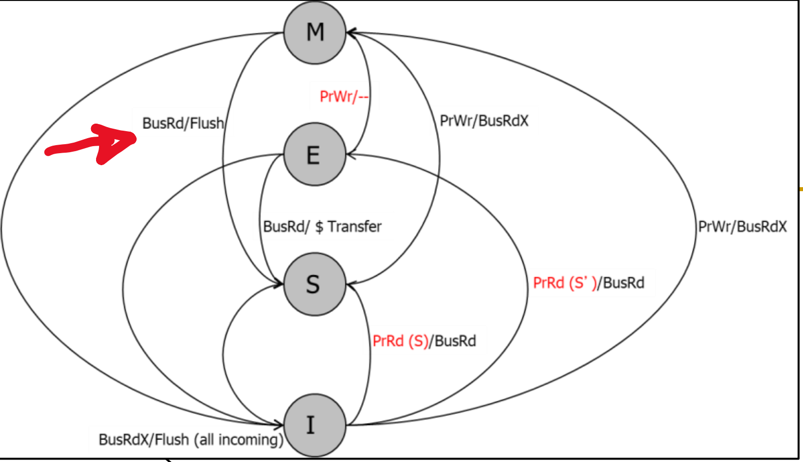


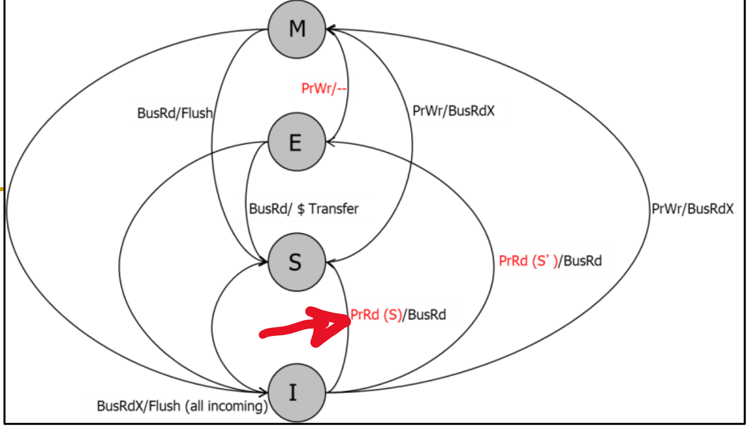
ld r2, x
add r1, r2, r4
st x, r1



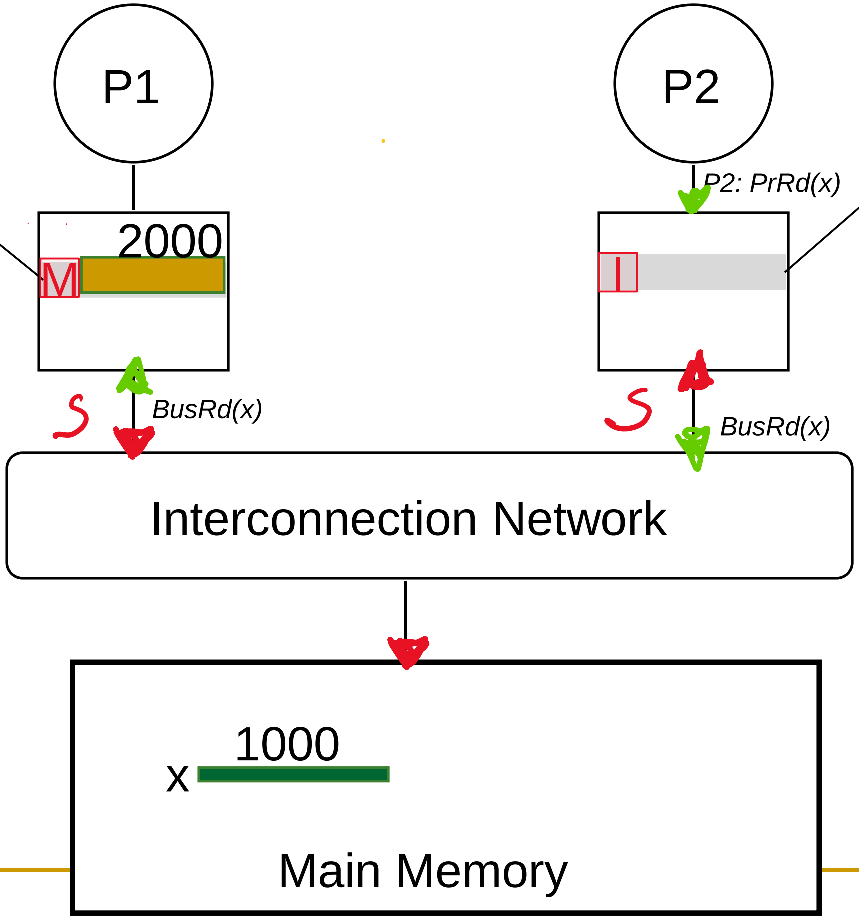
ld r2, x
ld r5, x

(MESI)

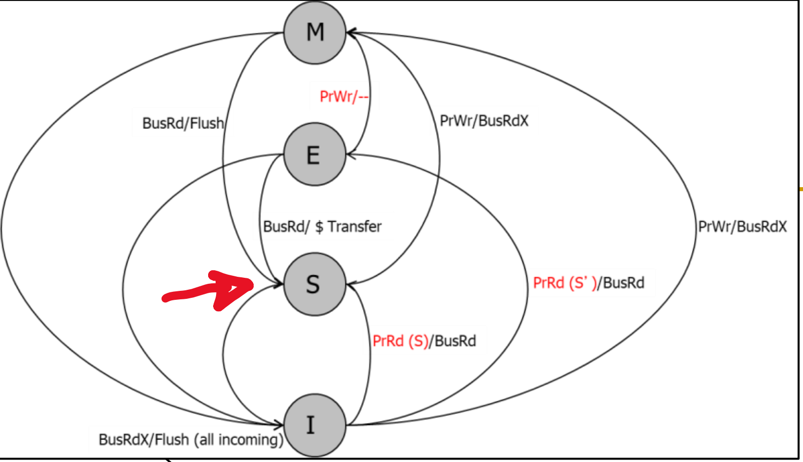




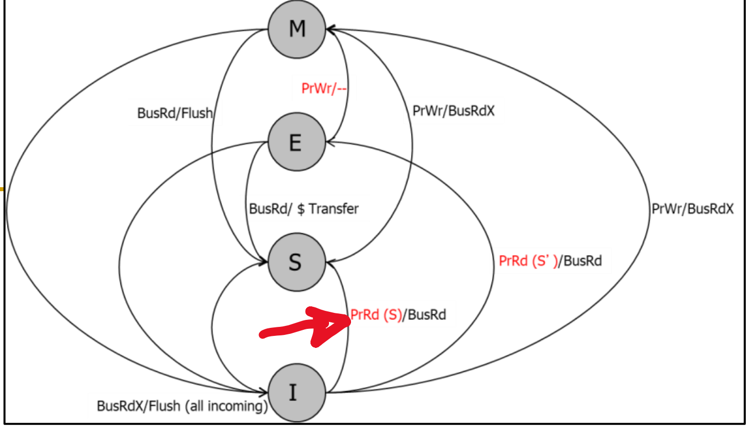
ld r2, x
add r1, r2, r4
st x, r1



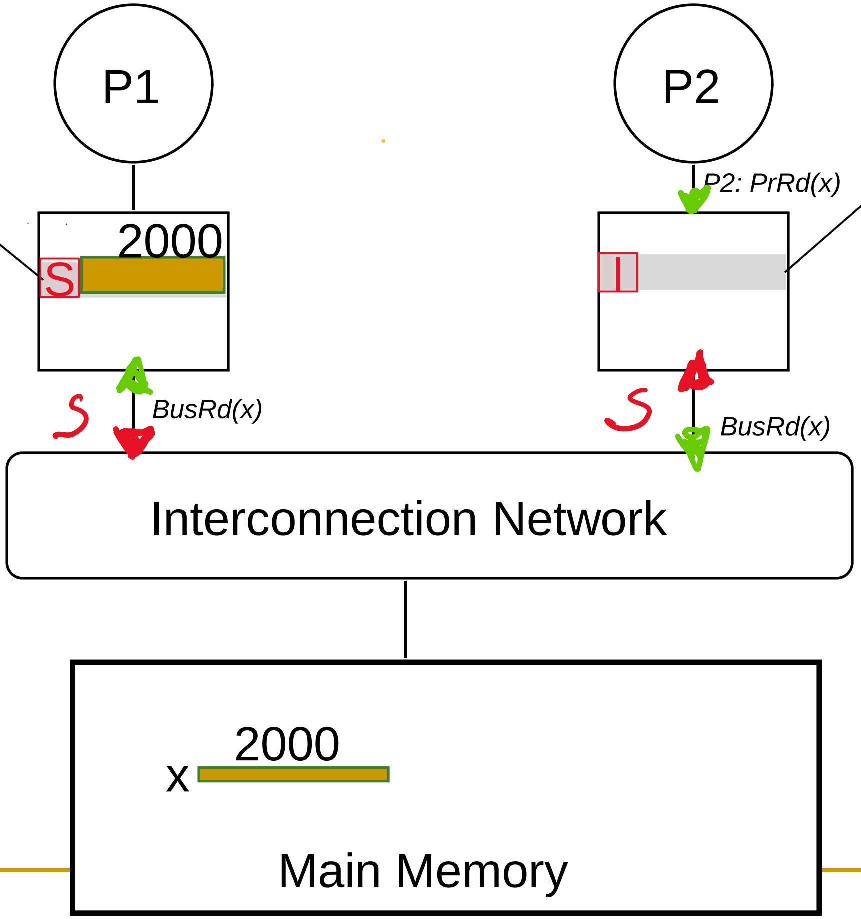
ld r2, x
ld r5, x



(MESI)

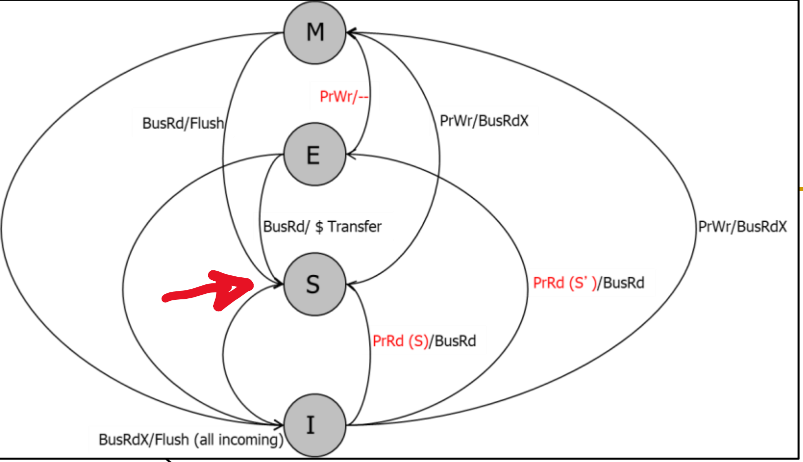


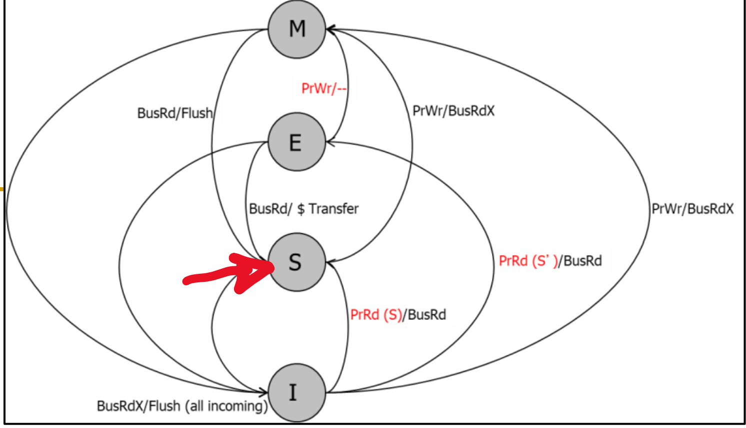
ld r2, x
add r1, r2, r4
st x, r1



ld r2, x
ld r5, x

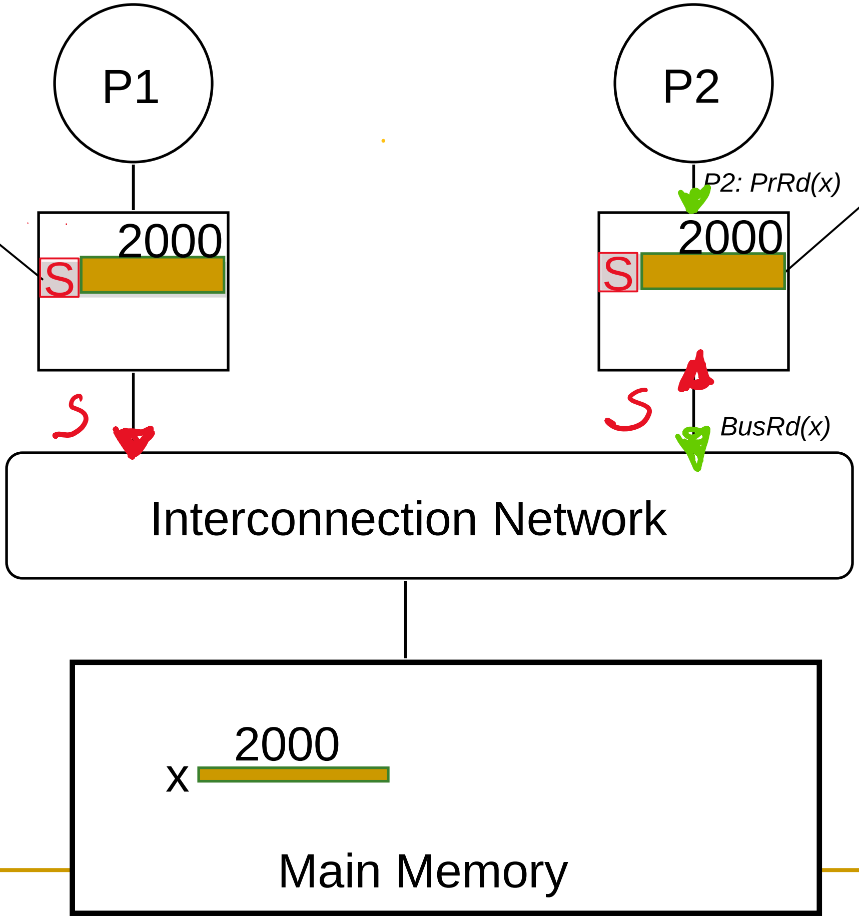
(MESI)





ld r2, x

ld r5, x


P1

P2

P2: PrRd(x)

2000

S

2000

S

S

S

BusRd(x)

Interconnection Network

x 2000

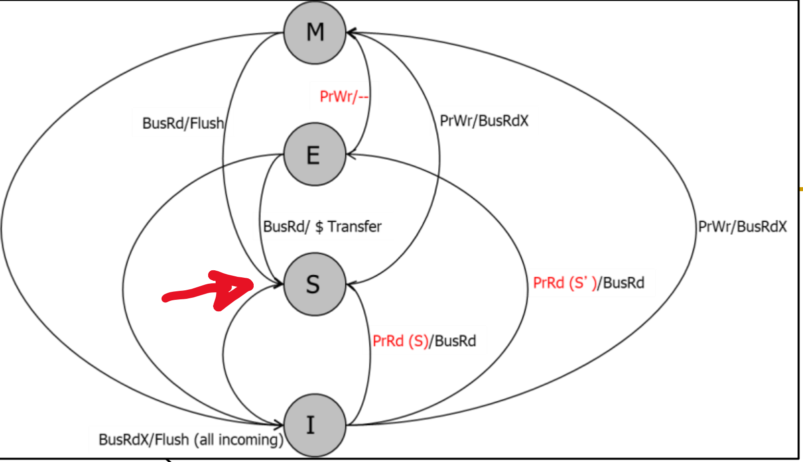
Main Memory

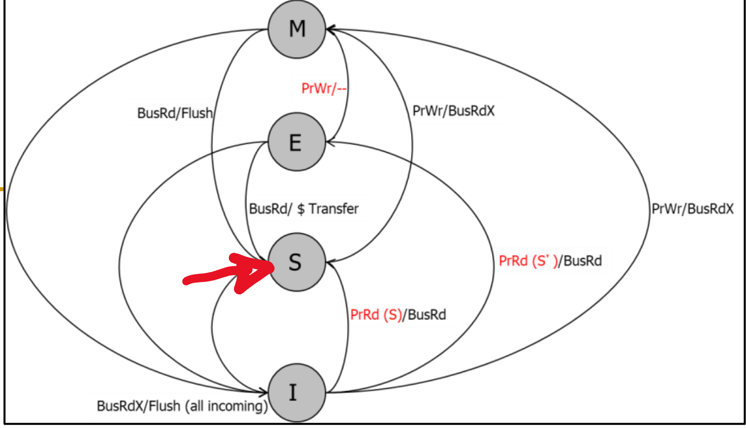
ld r2, x

add r1, r2, r4

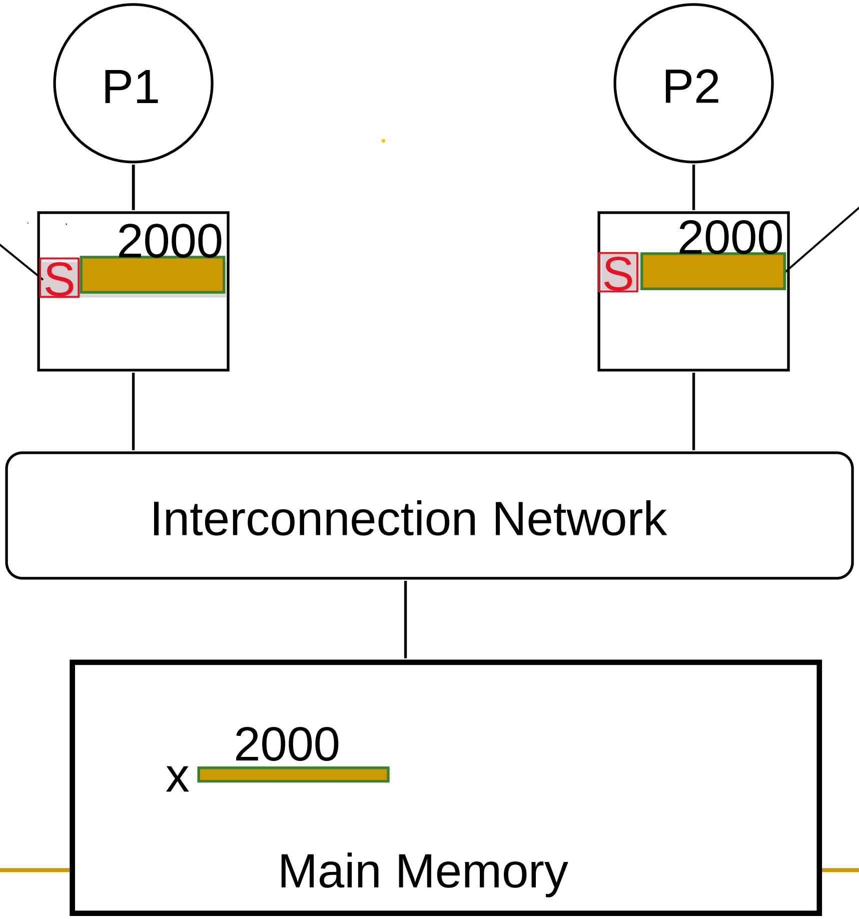
st x, r1

(MESI)

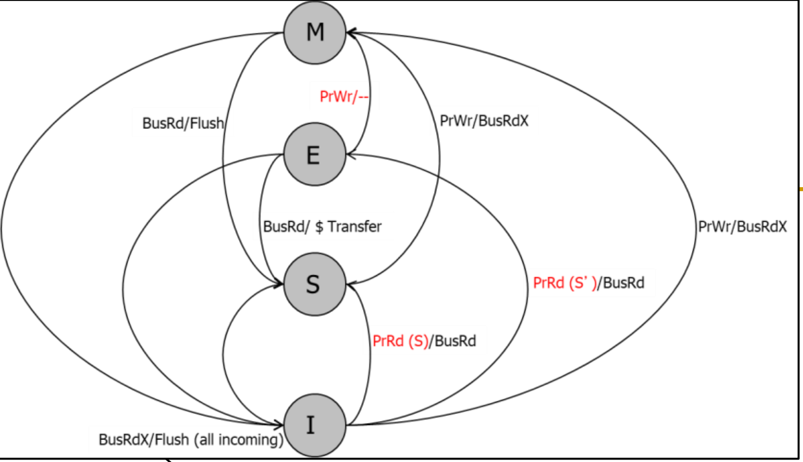




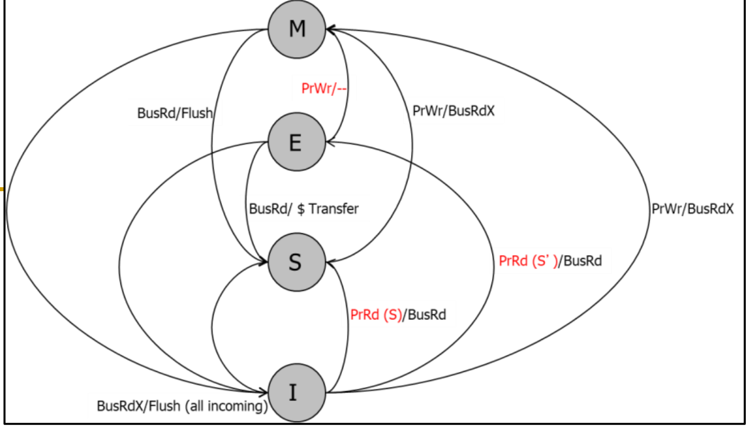
ld r2, x
add r1, r2, r4
st x, r1

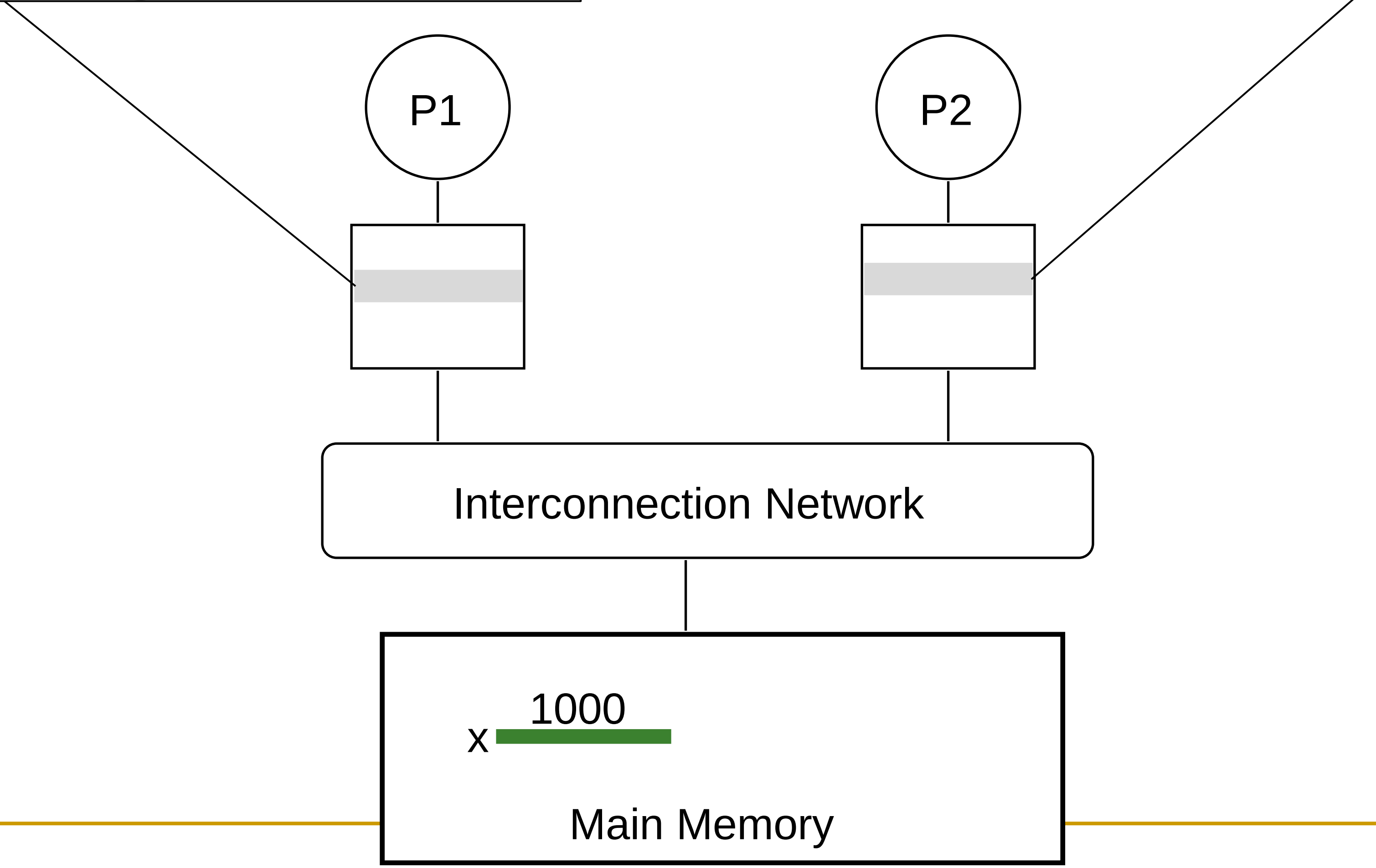


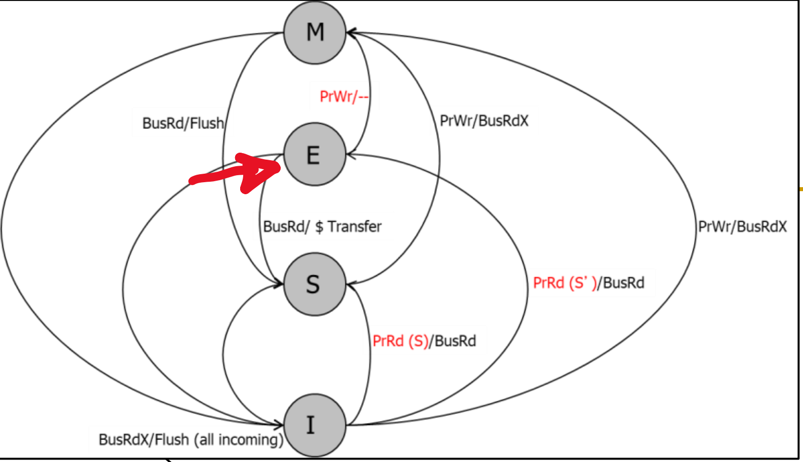
ld r2, x
ld r5, x


M
PrWr/--
BusRd/Flush
PrWr/BusRdX
E
BusRd/ $ Transfer
PrWr/BusRdX
S
PrRd (S')/BusRd
PrRd (S)/BusRd
I
BusRdX/Flush (all incoming)

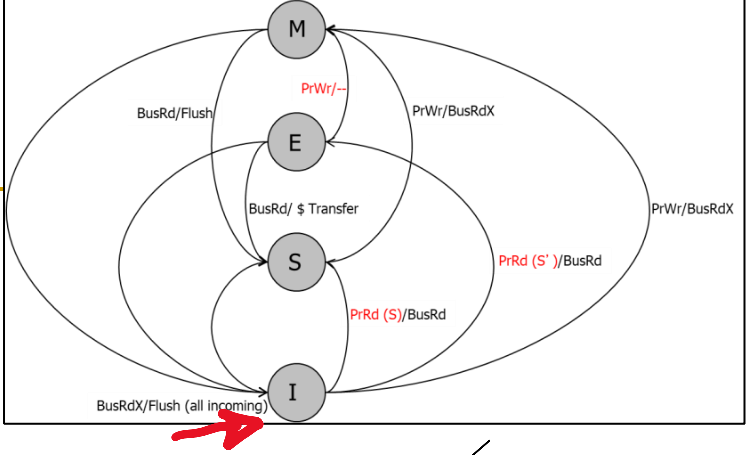
(MESI)


M
PrWr/--
BusRd/Flush
PrWr/BusRdX
E
BusRd/ $ Transfer
PrWr/BusRdX
S
PrRd (S')/BusRd
PrRd (S)/BusRd
I
BusRdX/Flush (all incoming)

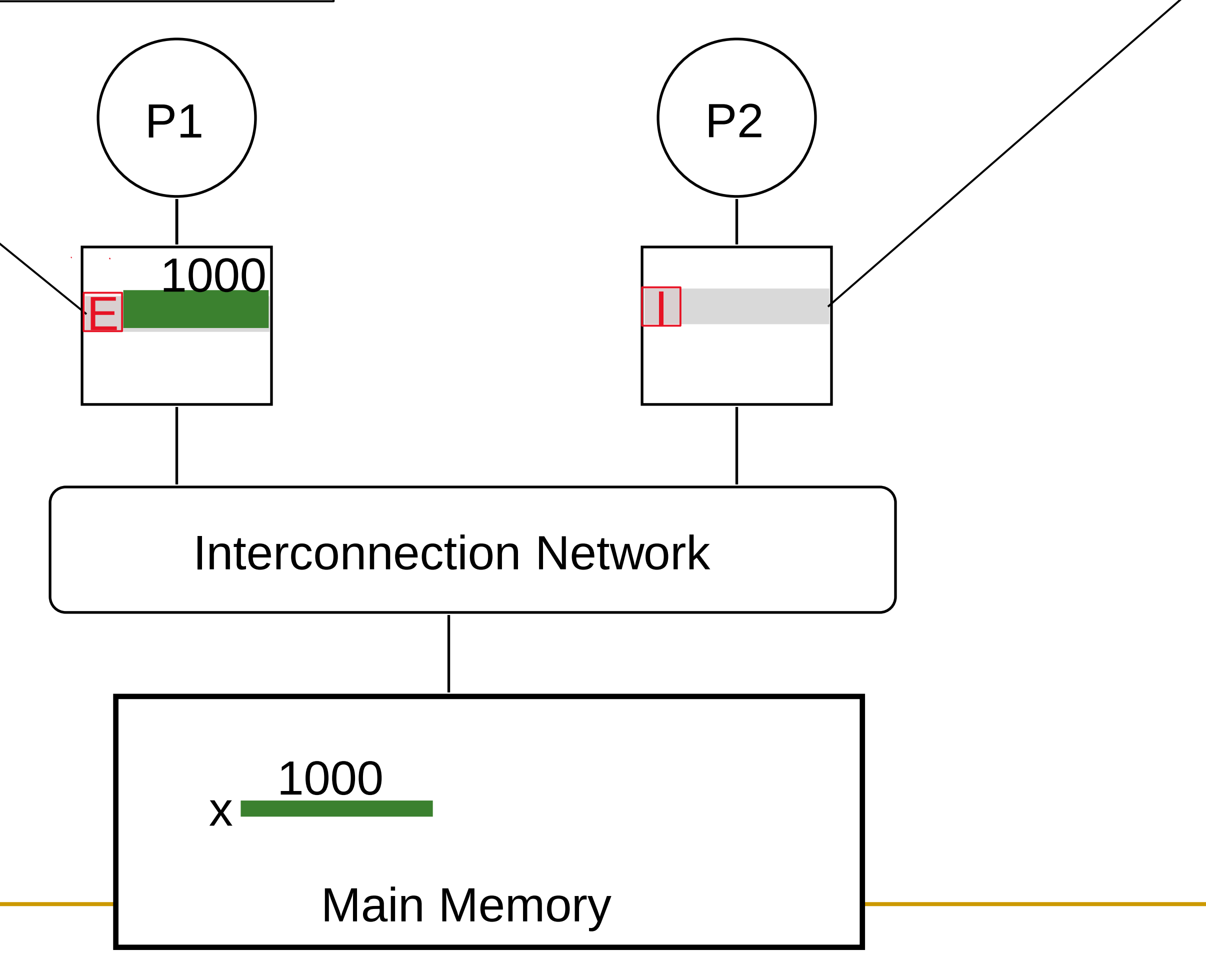

P1
P2
Interconnection Network
x
1000
Main Memory


M
E
S
I
PrWr/--
BusRd/Flush
PrWr/BusRdX
BusRd/ $ Transfer
PrWr/BusRdX
PrRd (S')/BusRd
PrRd (S)/BusRd
BusRdX/Flush (all incoming)

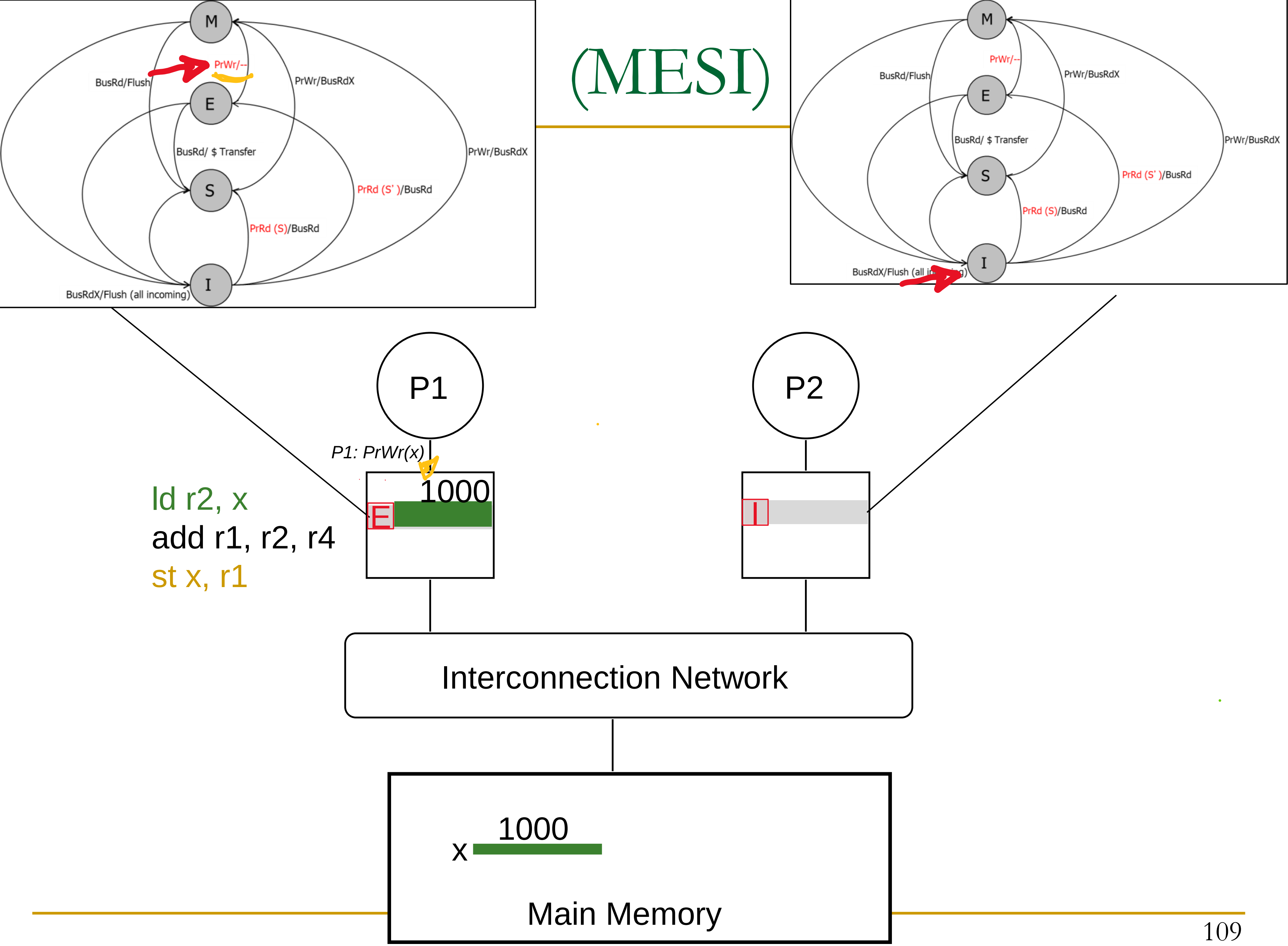
(MESI)


M
E
S
I
PrWr/--
BusRd/Flush
PrWr/BusRdX
BusRd/ $ Transfer
PrWr/BusRdX
PrRd (S')/BusRd
PrRd (S)/BusRd
BusRdX/Flush (all incoming)

ld r2, x

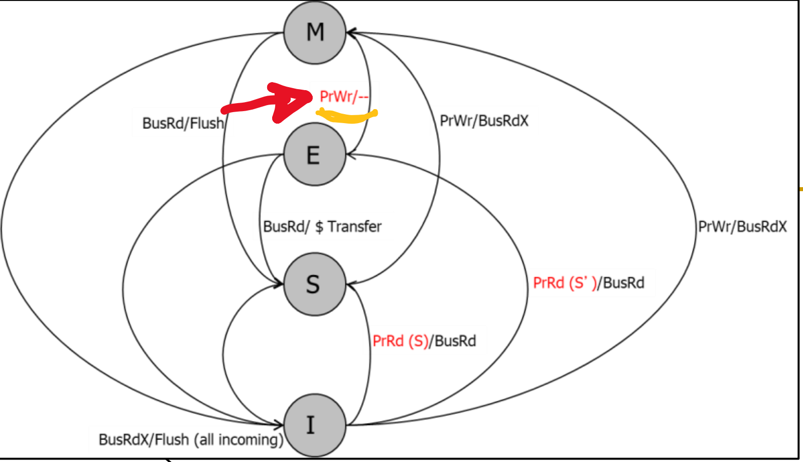

P1
P2
1000
E
I
Interconnection Network
x
1000
Main Memory

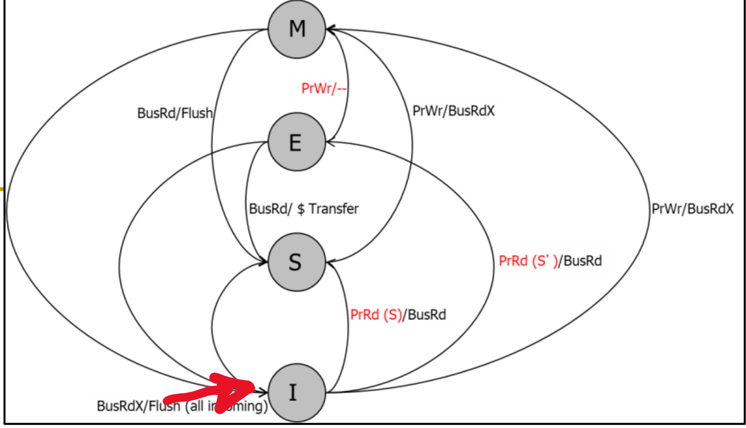
(MESI)



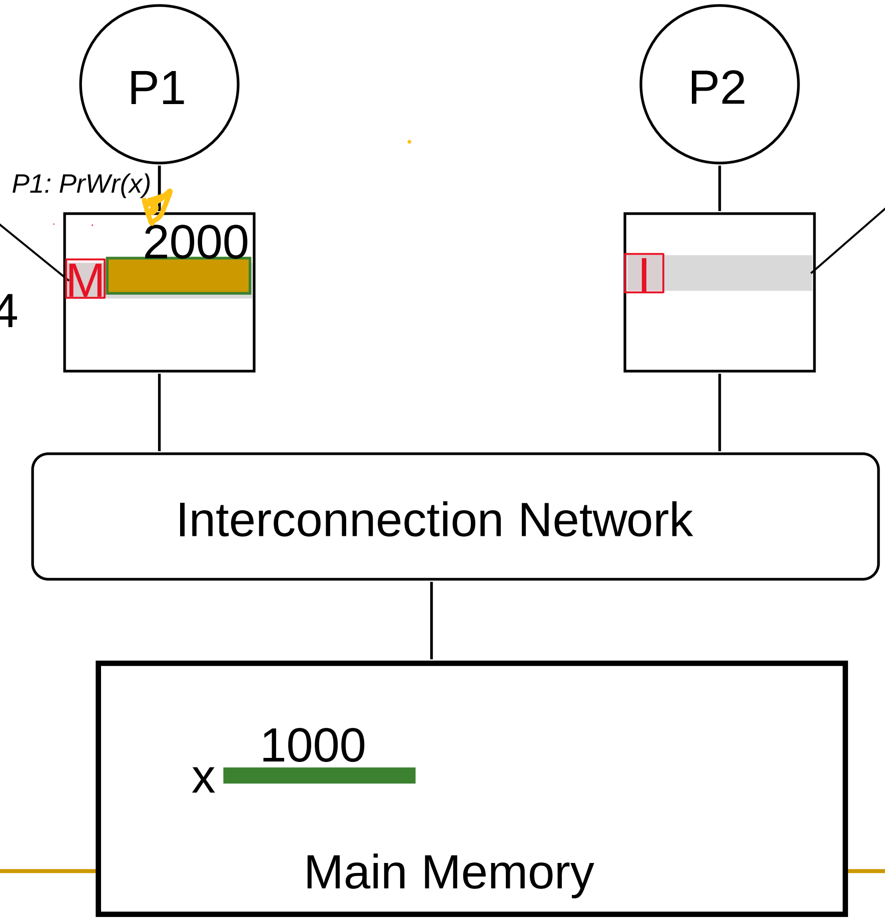
ld r2, x
add r1, r2, r4
st x, r1

(MESI)

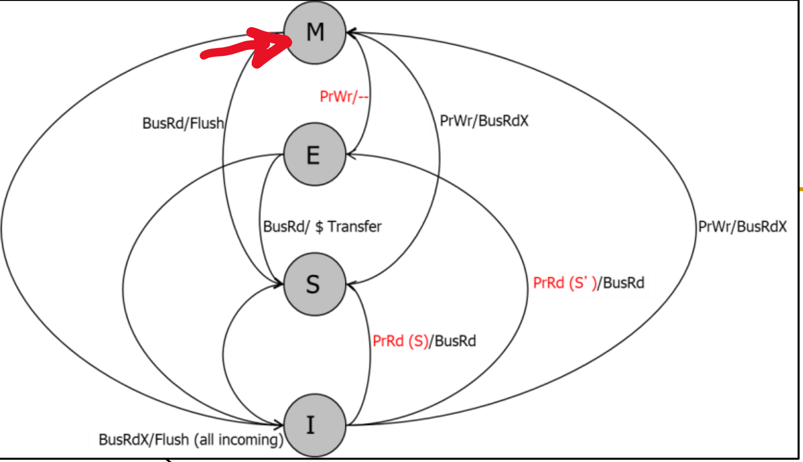


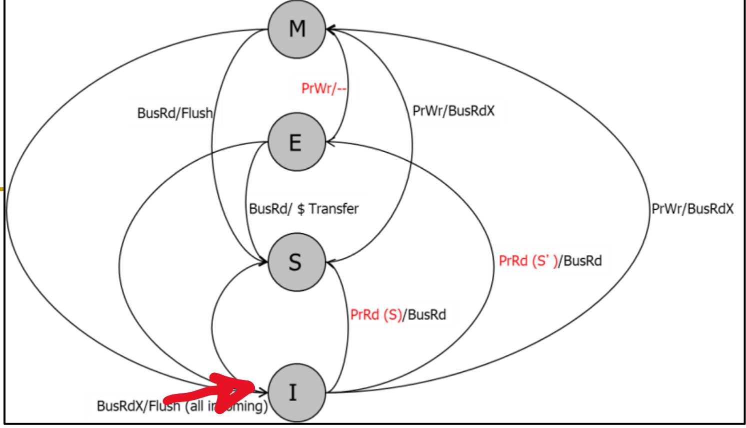


ld r2, x
add r1, r2, r4
st x, r1

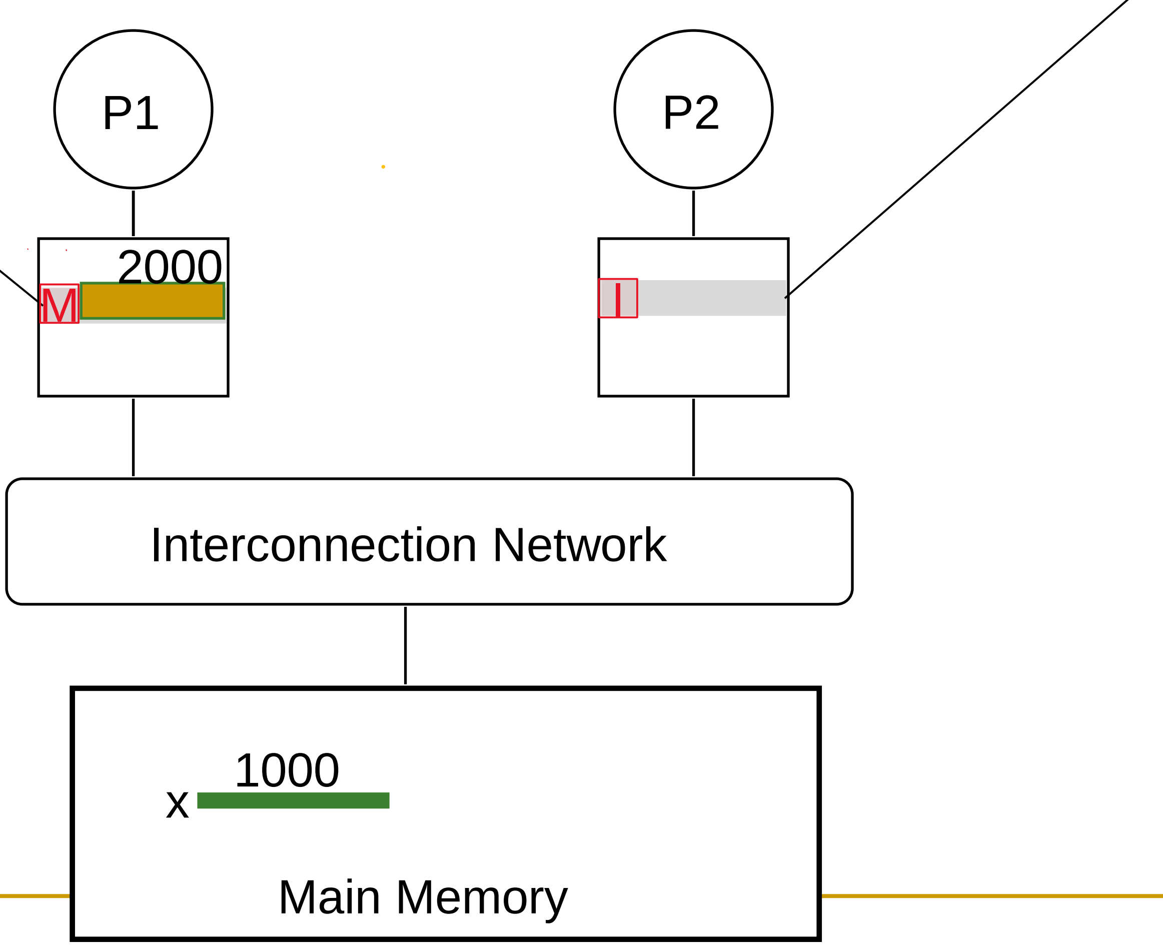


(MESI)





ld r2, x
add r1, r2, r4
st x, r1



P1

P2

2000

M

I

Interconnection Network

x 1000

Main Memory

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

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