

ADVANCED PARALLEL COMPUTING

LECTURE 03 - SNOOPING COHERENCE

Holger Fröning
holger.froening@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

Some material by Falsafi, Hardavellas, Nowatzky of EPFL, Northwestern, CMU

RECAP

Consistency vs. coherence (view from above the clouds)

Consistency is mainly about updates: a consistent view of all memory locations

Consistent global view of all memory locations

Coherence is mainly about reads: coherent view of individual cache lines

UMA vs. NUMA

Easy vs. scalable

Caches introduce coherence problem

Consistency problem is independent of using caches

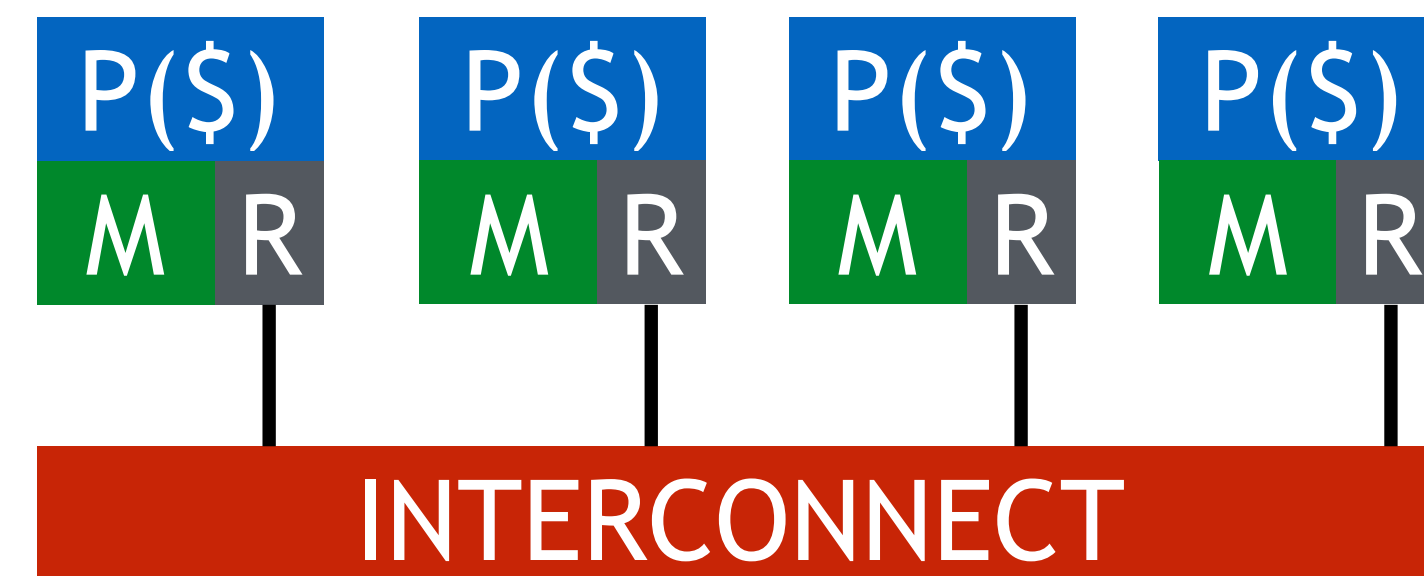
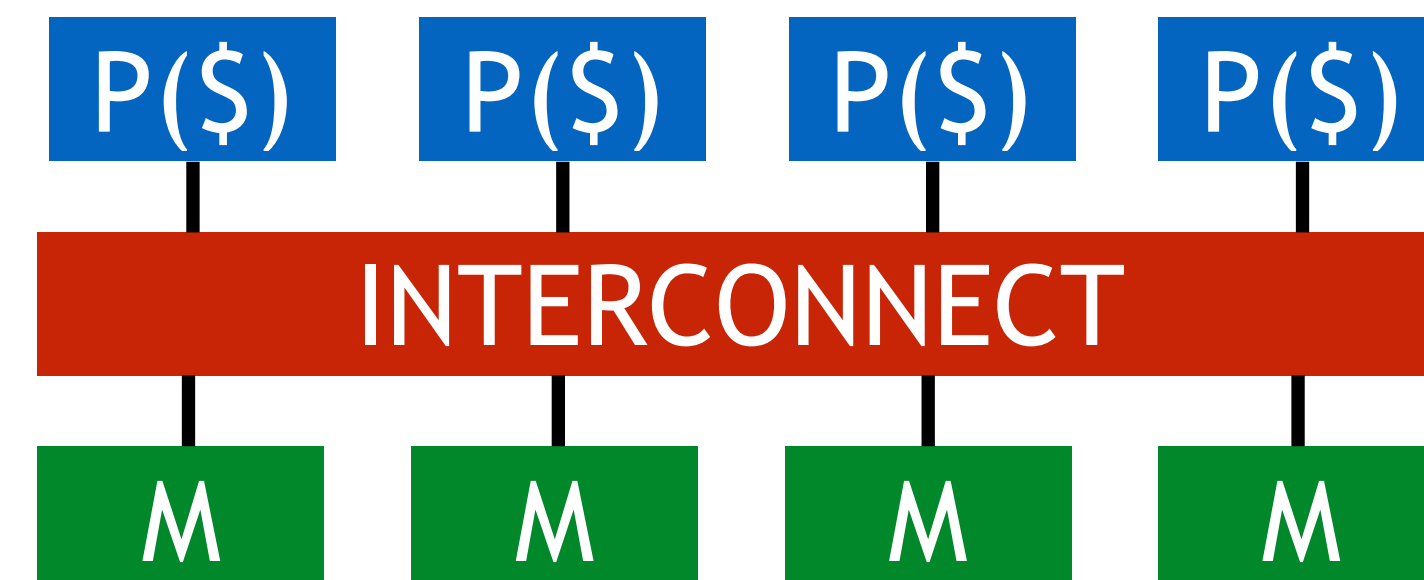
```
//initial all var = 0
```

Processor 0

```
A = 1;  
flag = 1;
```

Processor 1

```
while (flag == 0);  
print A;
```



CONSISTENCY VS COHERENCE

DEKKER'S ALGORITHM

Coherence problem?

Shared variables

Consistency problem?

Ordering of writes and
reads on different
addresses

```
//initial: all vars 0
```

P0

```
flag[0] = true;
while (flag[1] == true) {
    if (turn != 0) {
        flag[0] = false;
        while (turn != 0) {};
        flag[0] = true;
    }
}
```

...

```
turn = 1;
flag[0] = false;
```

P1

```
flag[1] = true;
while (flag[0] == true) {
    if (turn != 1) {
        flag[1] = false;
        while (turn != 1) {};
        flag[1] = true;
    }
}
```

...

```
turn = 0;
flag[1] = false;
```

DEKKER'S ALGORITHM

Software-only solution to the mutual exclusion problem

First known correct solution

Works as advertised under sequential consistency

No atomics!

Break-down to highlight consistency issues:

```
//initial A = B = 0
```

P0

```
A = 1;
```

```
retry:
```

```
if (B != 0) goto retry;
```

```
// enter critical section
```

P1

```
B = 1;
```

```
retry:
```

```
if (A != 0) goto retry;
```

```
// enter critical section
```

STORE BUFFERS

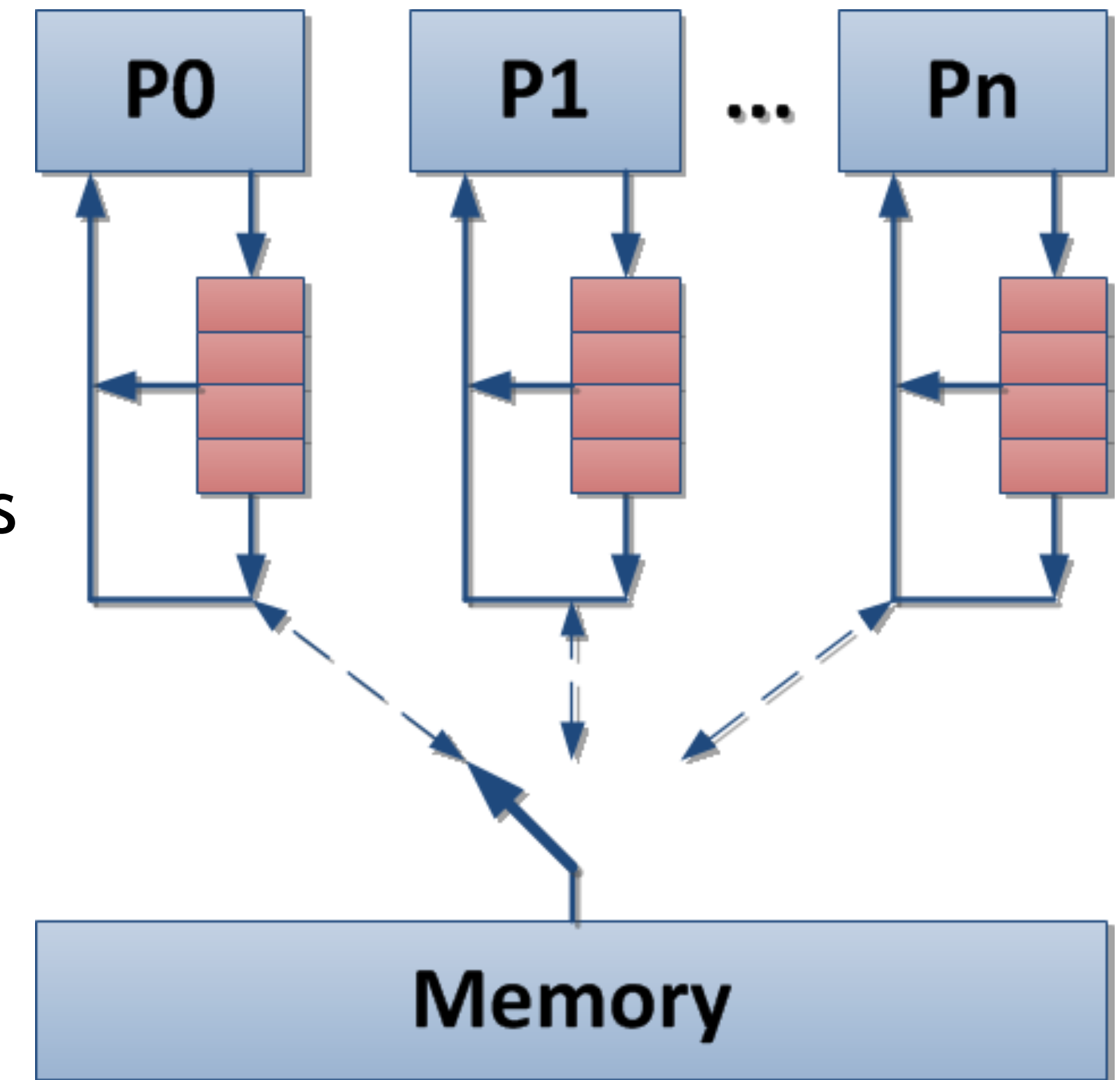
Store Buffers: used in modern processors

- Allow reads to bypass incomplete writes

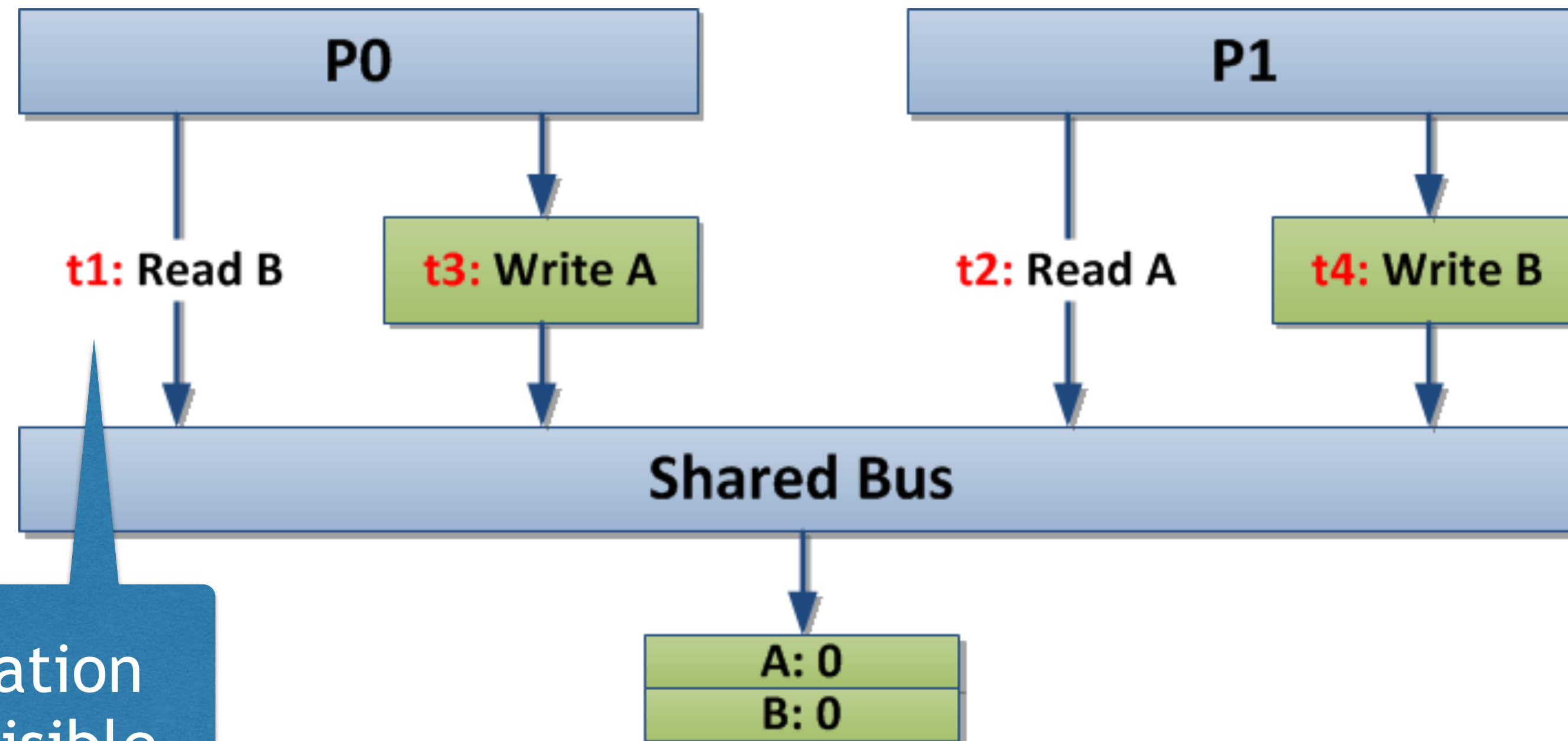
- Reads search store buffer for matching addresses (read own write early)

- Hides all latencies related to store misses

Can be flushed using *MFENCE*



DEKKER'S ALGORITHM WITH STORE BUFFERS



t: time when operation becomes globally visible

```
//initial A = B = 0
```

P0

```
A = 1;
```

```
retry:
```

```
if (B != 0) goto retry;
```

```
// enter critical section
```

P1

```
B = 1;
```

```
retry:
```

```
if (A != 0) goto retry;
```

```
// enter critical section
```

WHERE TO PUT THE MFENCE?

insert “MFENCE ();”

```
//initial: all vars 0
```

P0

```
flag[0] = true;
while (flag[1] == true) {
    if (turn != 0) {
        flag[0] = false;
        while (turn != 0) {};
        flag[0] = true;
    }
}
```

...

```
turn = 1;
flag[0] = false;
```

P1

```
flag[1] = true;
while (flag[0] == true) {
    if (turn != 1) {
        flag[1] = false;
        while (turn != 1) {};
        flag[1] = true;
    }
}
```

...

```
turn = 0;
flag[1] = false;
```


SEQUENTIAL CONSISTENCY

Not maintained by modern processor architectures

See example: Dekker's Algorithm

Users/programmers have to specify when ordering matters

=> Relaxed or weak consistency

Alternative: speculatively ignore ordering rules

Goal: combination of easy use (no burden on programmer) and performance of relaxed consistency

More on (relaxed) consistency later...

SNOOPING COHERENCE

RECAP: THE COHERENCE PROBLEM

Caches reduce average memory access latency

Mind the 3C of cache in-effectivity

Caches have to be kept coherent

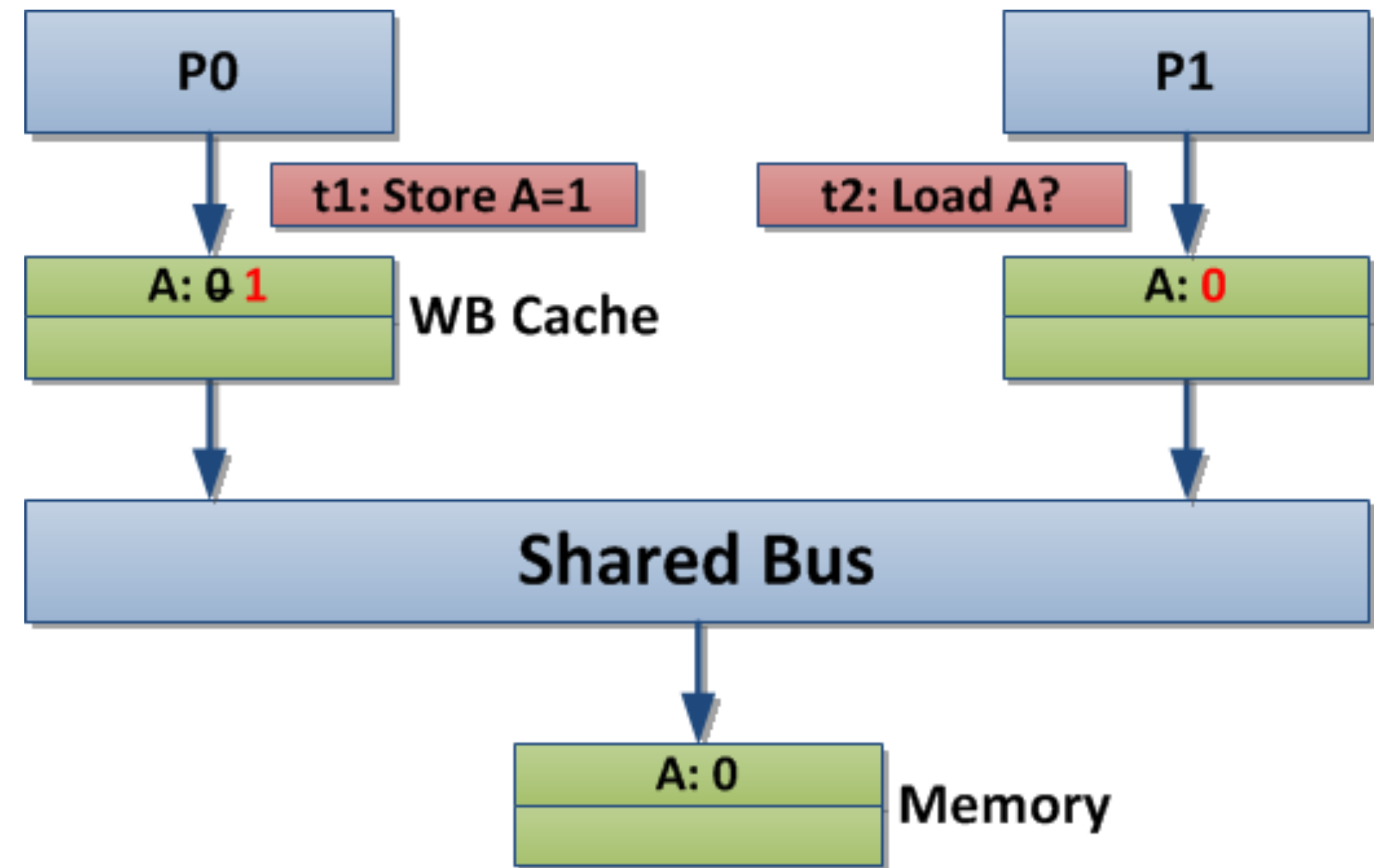
Ensure that all Ps see the same (most recent) value

Write-back (WB) policy

Coherence problem?

Write-through (WT) policy

Coherence problem?



APPROACHES TO CACHE COHERENCE

Software coherence

Blocks (cache lines, pages) are marked as cacheable/non-cacheable

Add “flush” and “invalidate” instructions

When are these needed?

Compiler or run-time system

Difficult to get perfect

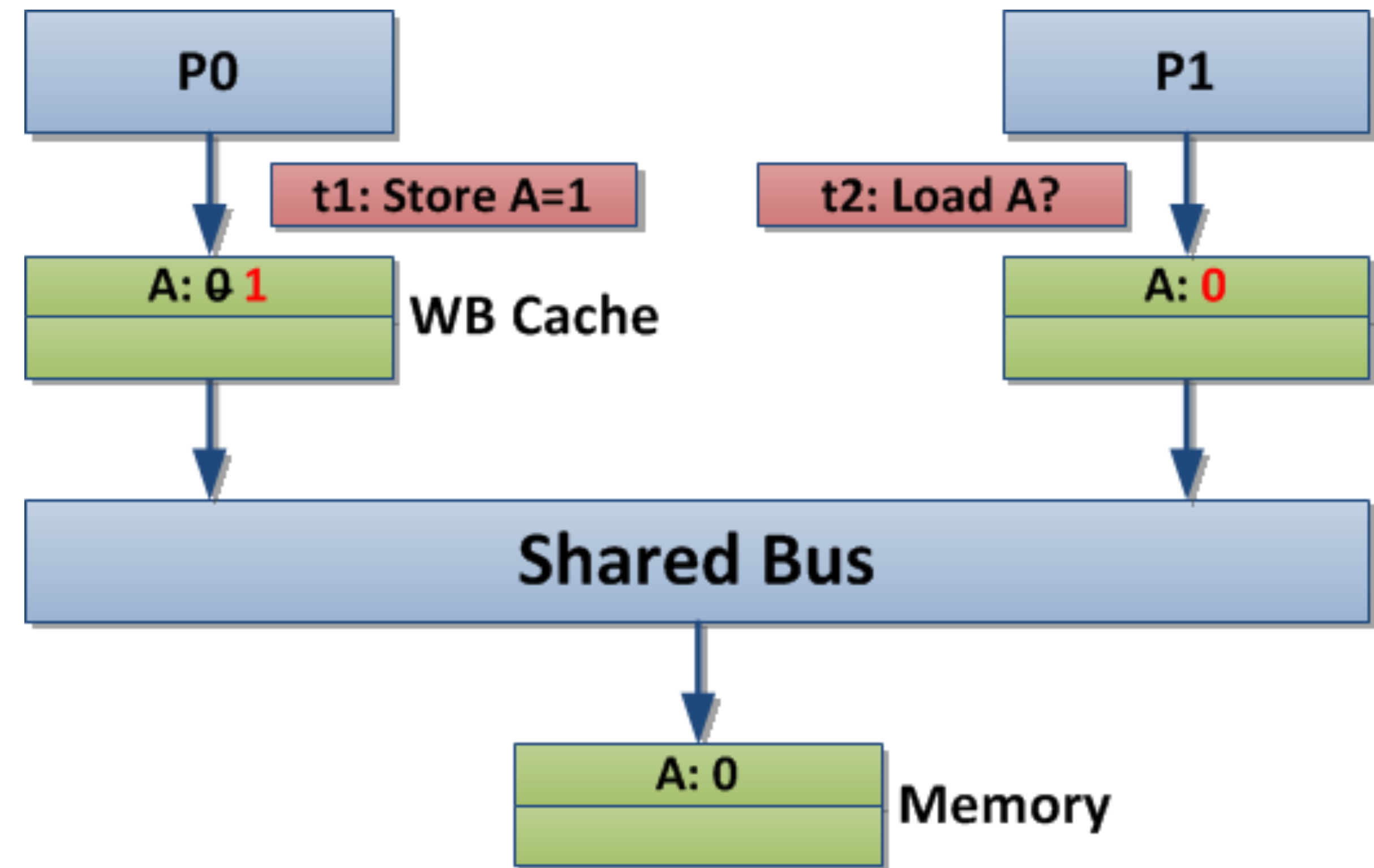
Memory aliasing

Will be reviewed later

Hardware coherence

Far more common

Focus of this lecture



WRITE-THROUGH: VALID-INVALID COHERENCE

Allows multiple readers, but writes must pass bus

=> Write-through, no-write-allocate cache

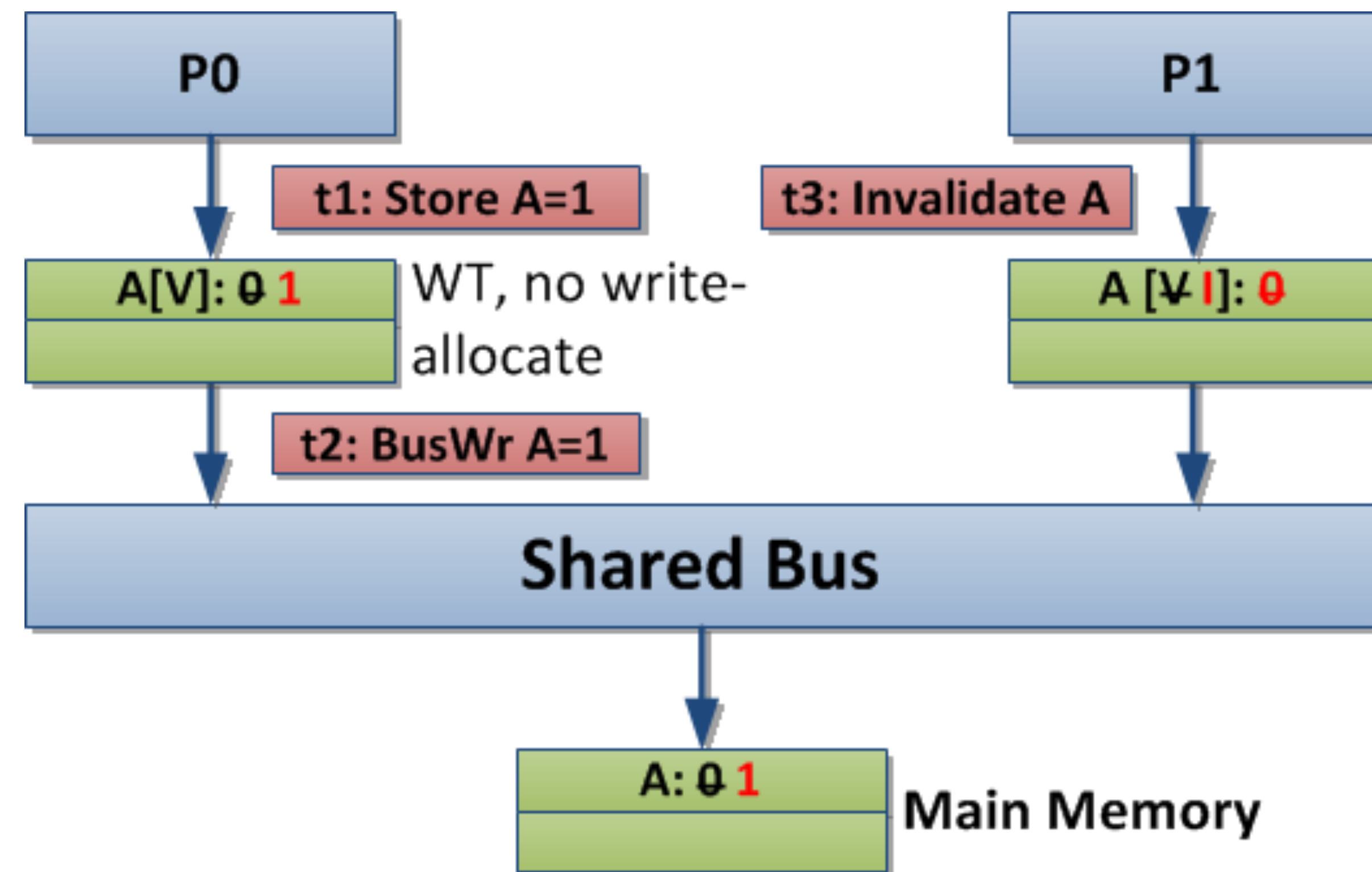
Write-invalidate coherence

=> All caches must monitor (“snoop”) bus traffic

Implementation using FSMs

What if two stores happen at the same point in time?

Can't happen - it's a bus

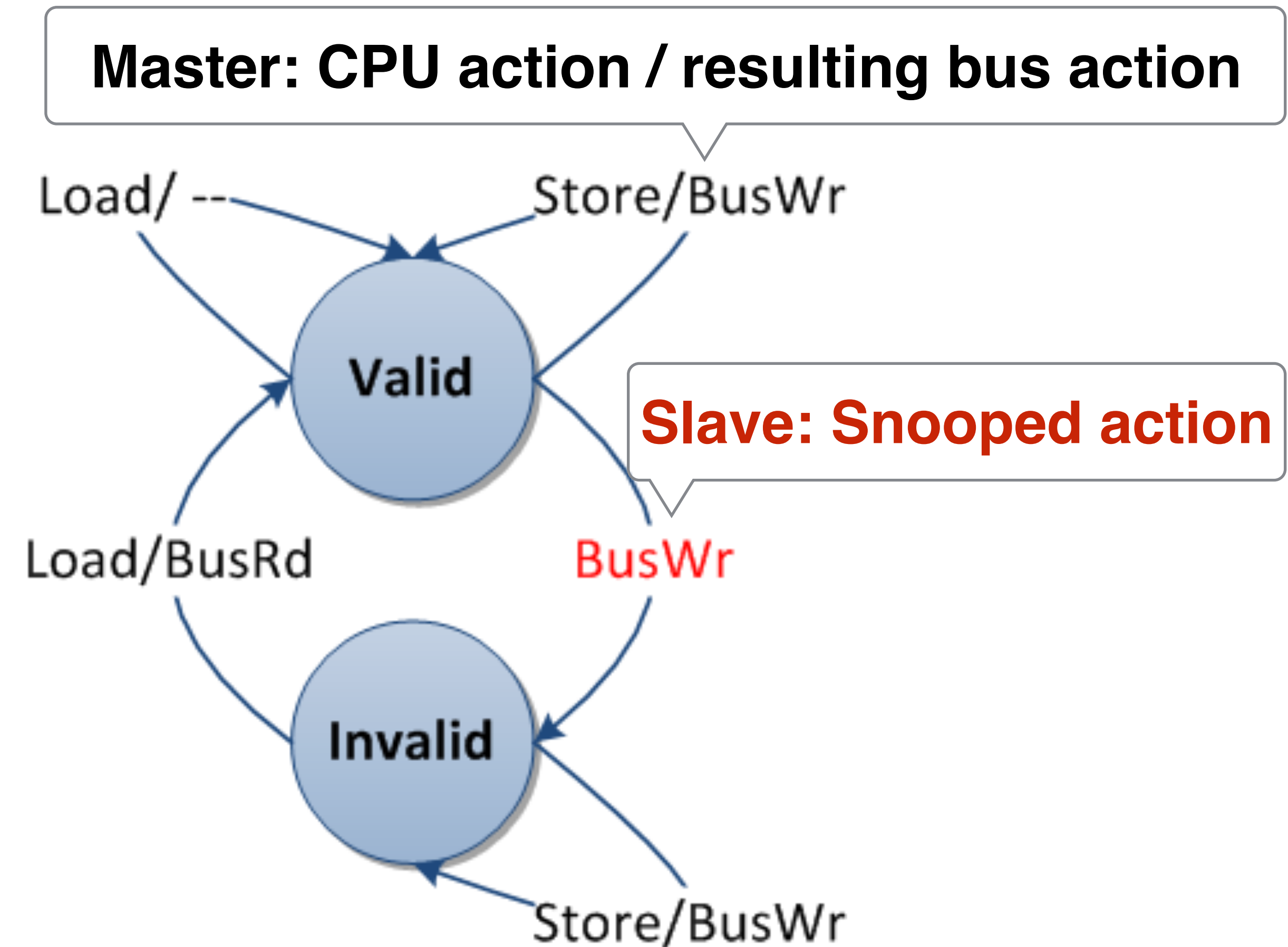


VALID-INVALID SNOOPING PROTOCOL

Actions

Load, Store
BusRd, BusWr

Write-through, no-write-allocate
1 bit of storage overhead per cache line



WRITE-THROUGH: WRITE-UPDATE COHERENCE

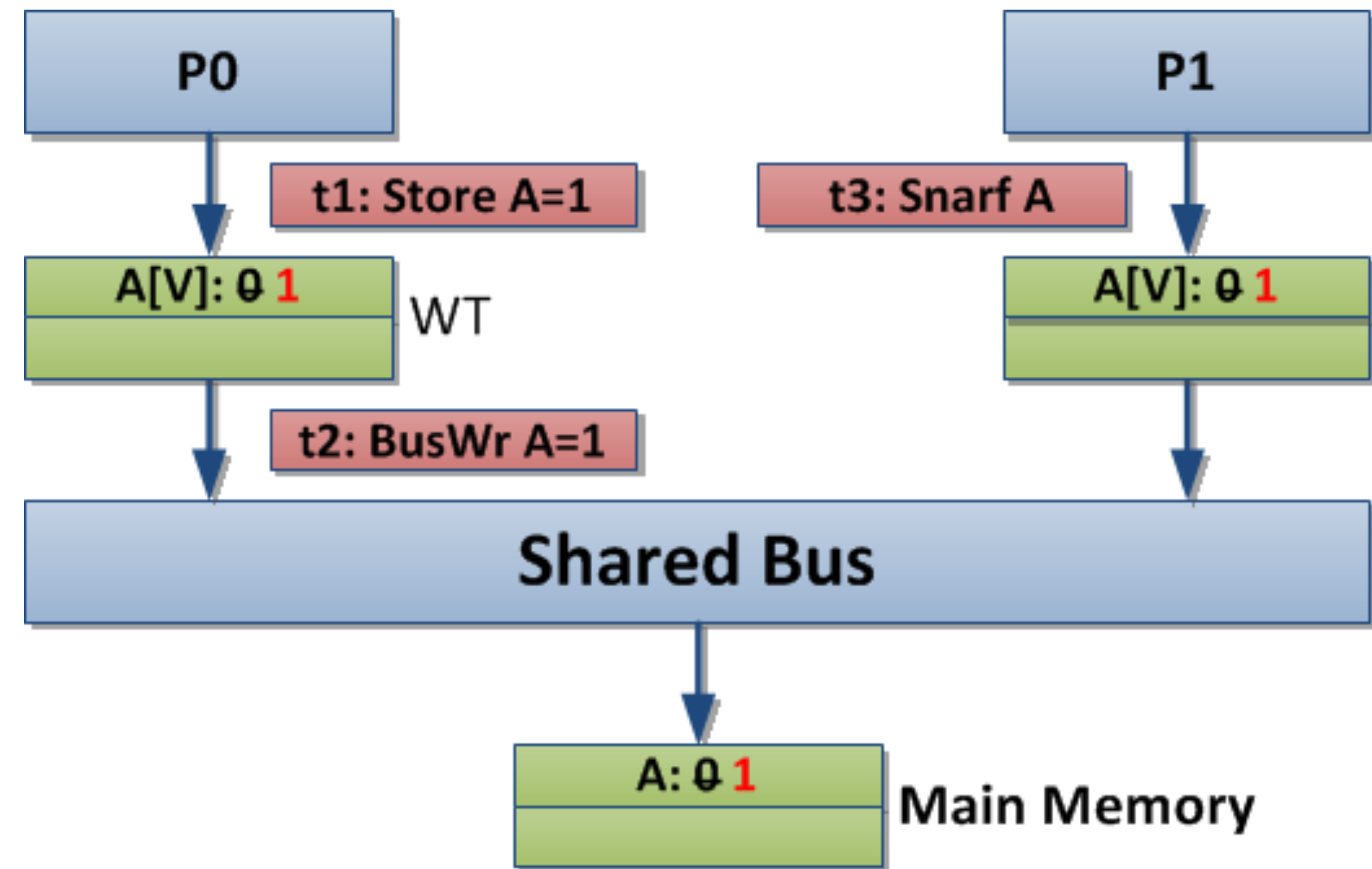
Updates instead of invalidations

“Snarfing”

Consider that 15% of all bus accesses are stores

= Tremendous bus and cache bandwidth required

=> In practice basically not used



WRITE-BACK

Write-back caches dramatically reduce bus write bandwidth

Key idea: add a new state indicating “ownership”

Exclusive copy: “owner” has the only replica of a cache block, it may update freely

Shared copy: multiple readers are ok, but ownership is required for writing

Need to find owner (if any) on read misses

Need to eventually update memory so writes are not lost

MODIFIED-SHARED-INVALID (MSI) PROTOCOL

MODIFIED-SHARED-INVALID (MSI) PROTOCOL

Based on VI: replace “valid” with “modified” and “shared”

Invalid: no copy

Shared: cache has a read-only copy, clean

Clean = memory is up-to-date

Modified: cache has writable (exclusive) copy, dirty

Dirty = memory is out-of-date



Ownership

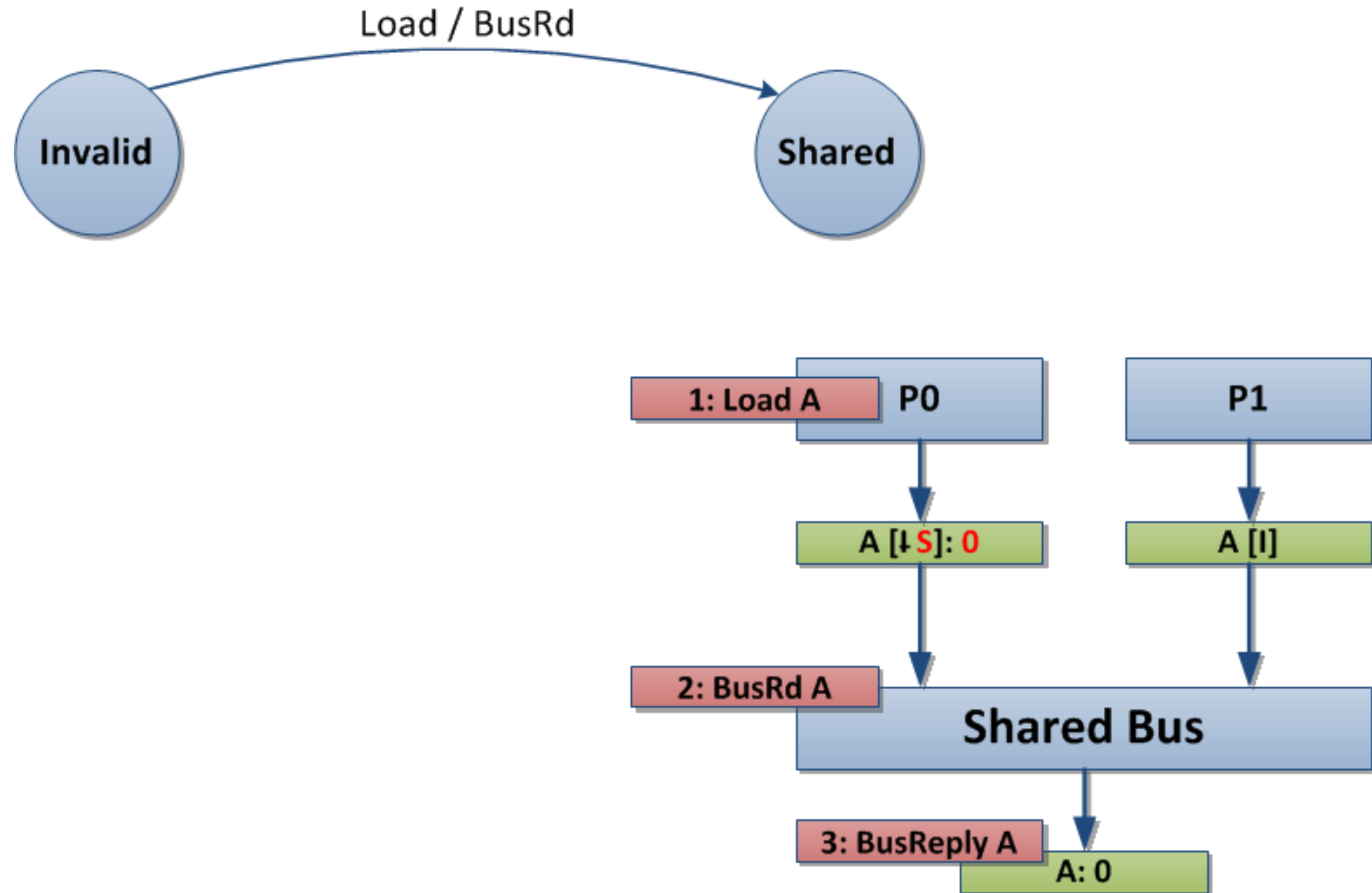
Actions: Load, Store, Evict

Bus messages: BusRd, BusRdX, BusInv, BusWb, BusReply

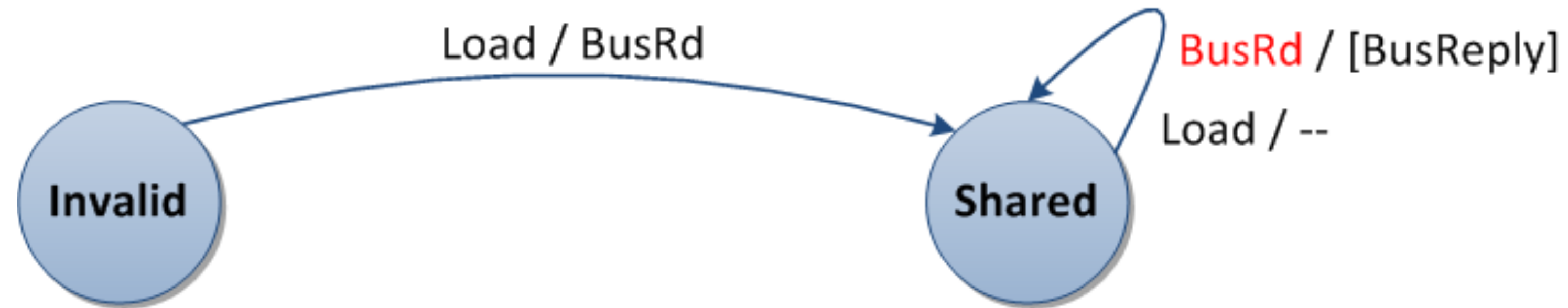
BusRdX: get exclusive access and the most recent copy of the data

Some can be combined

MODIFIED-SHARED-INVALID (MSI) PROTOCOL

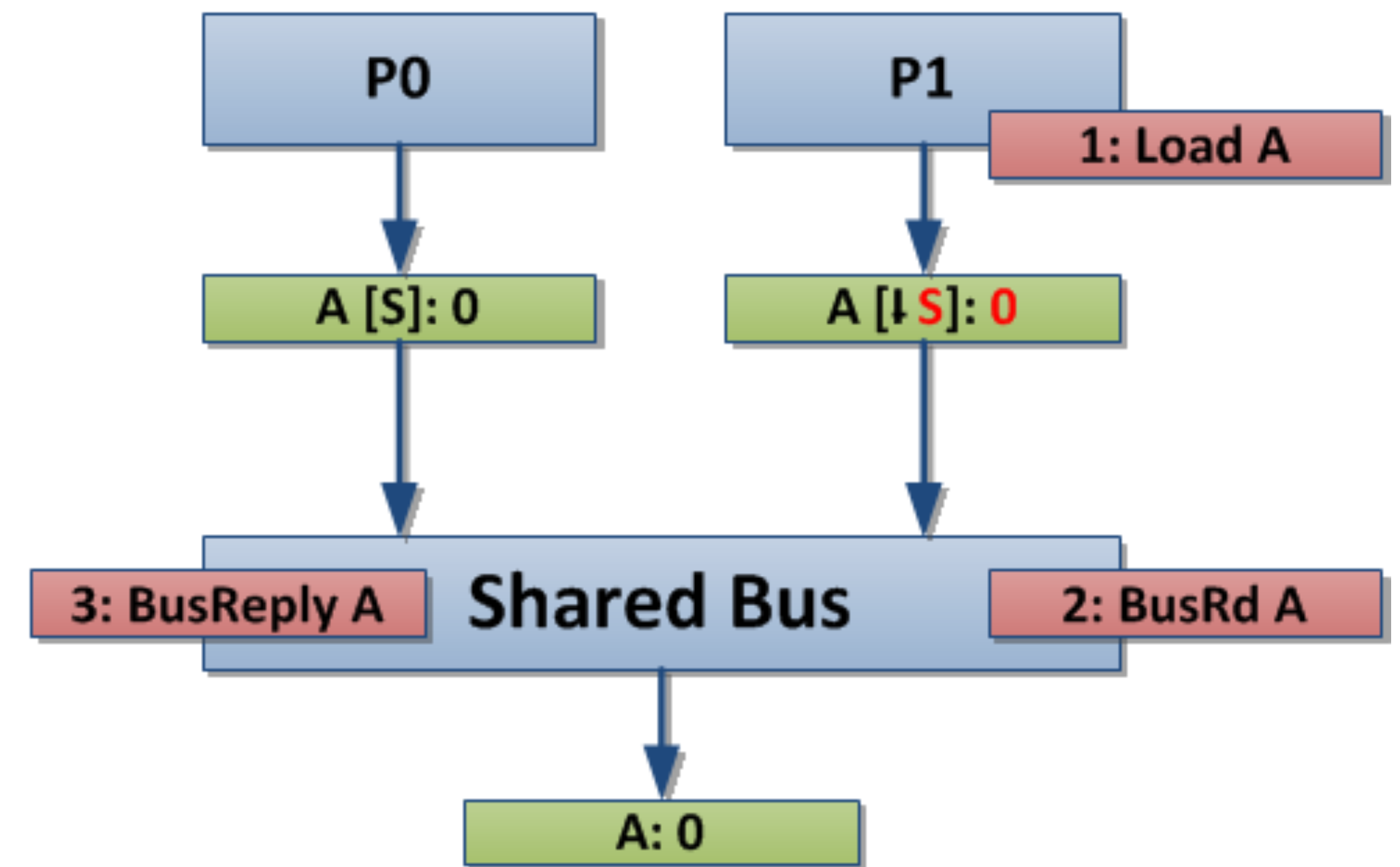


MODIFIED-SHARED-INVALID (MSI) PROTOCOL

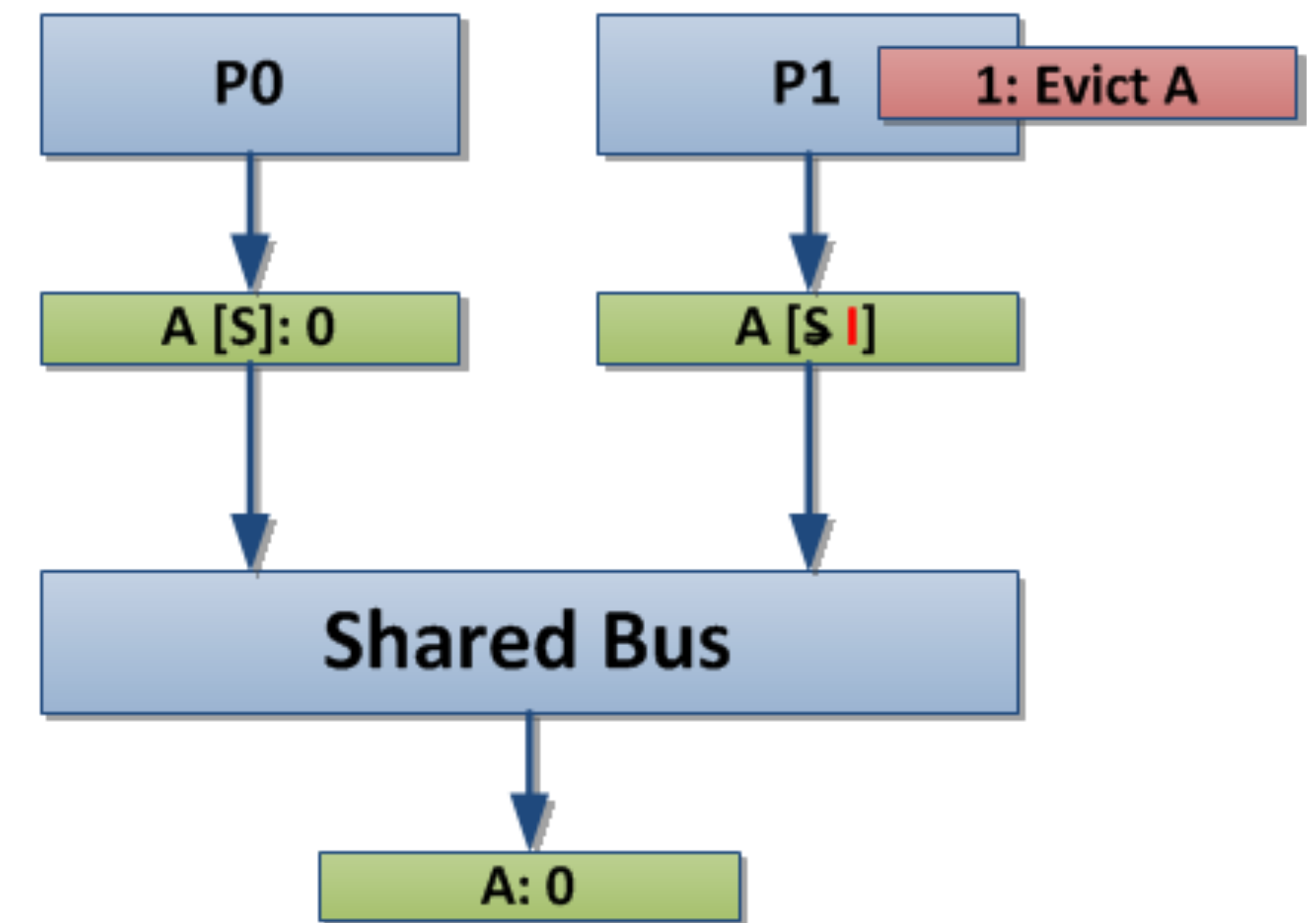
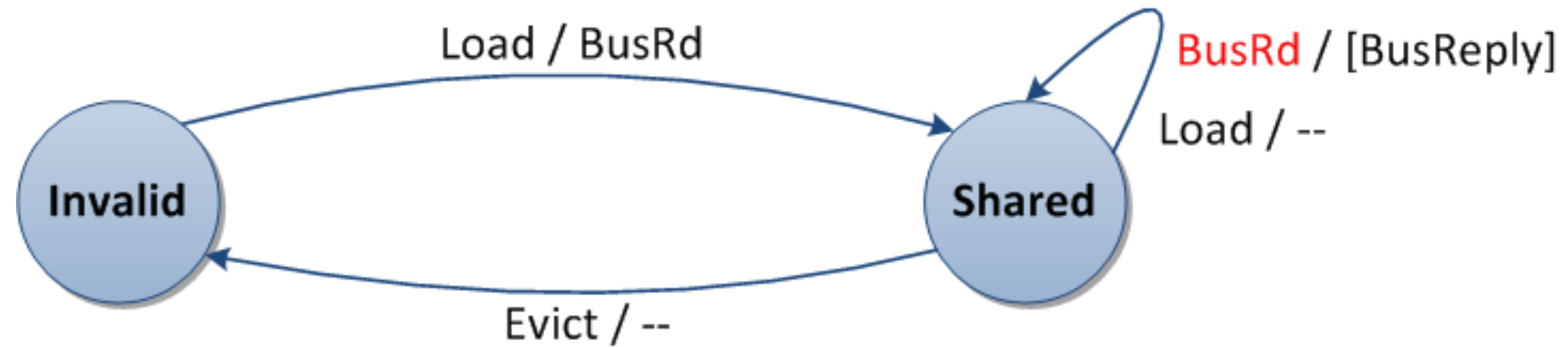


An answer from memory
would be sufficient.
Why answers P0?

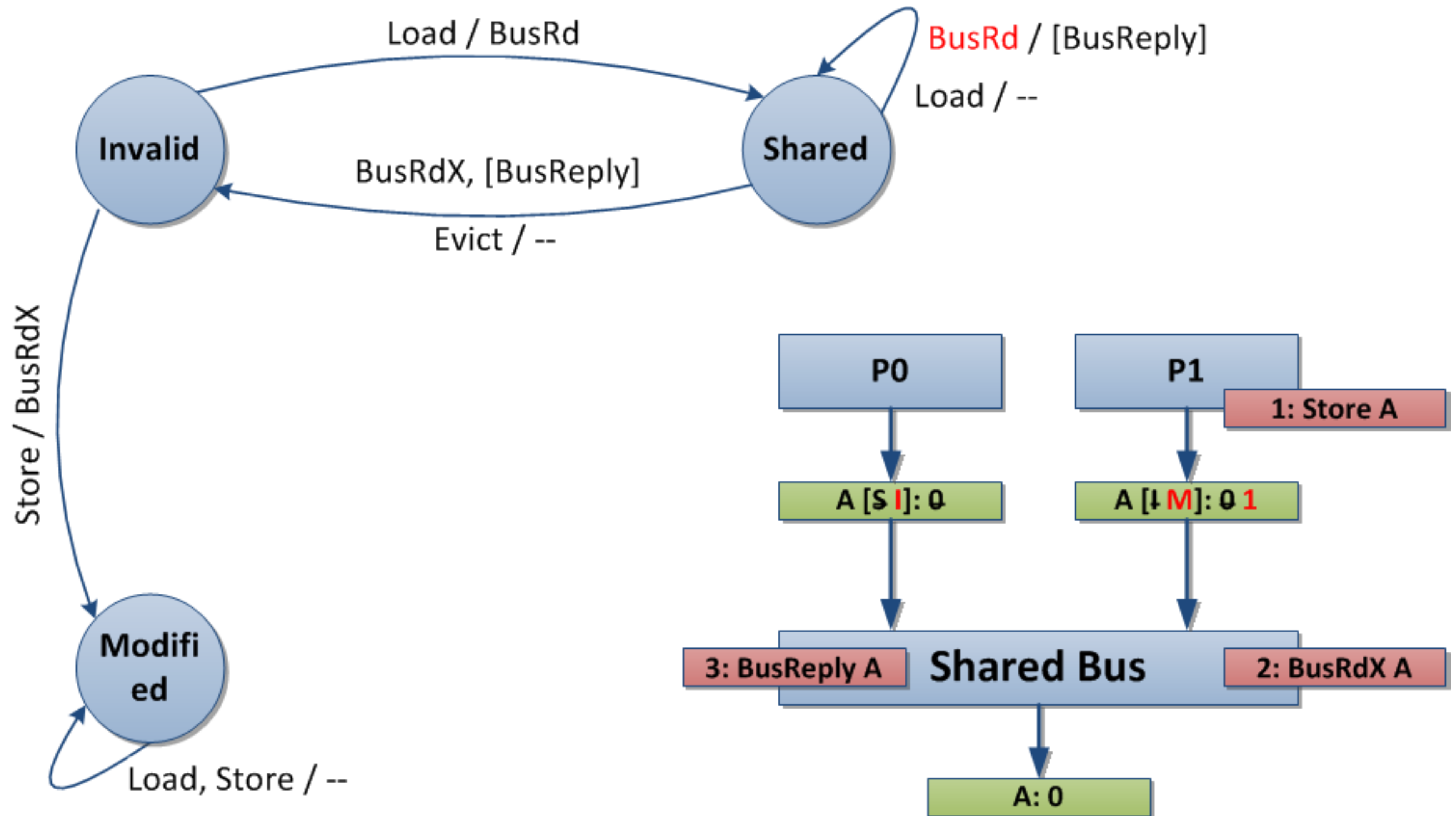
Cache controller answers
faster than memory
controller



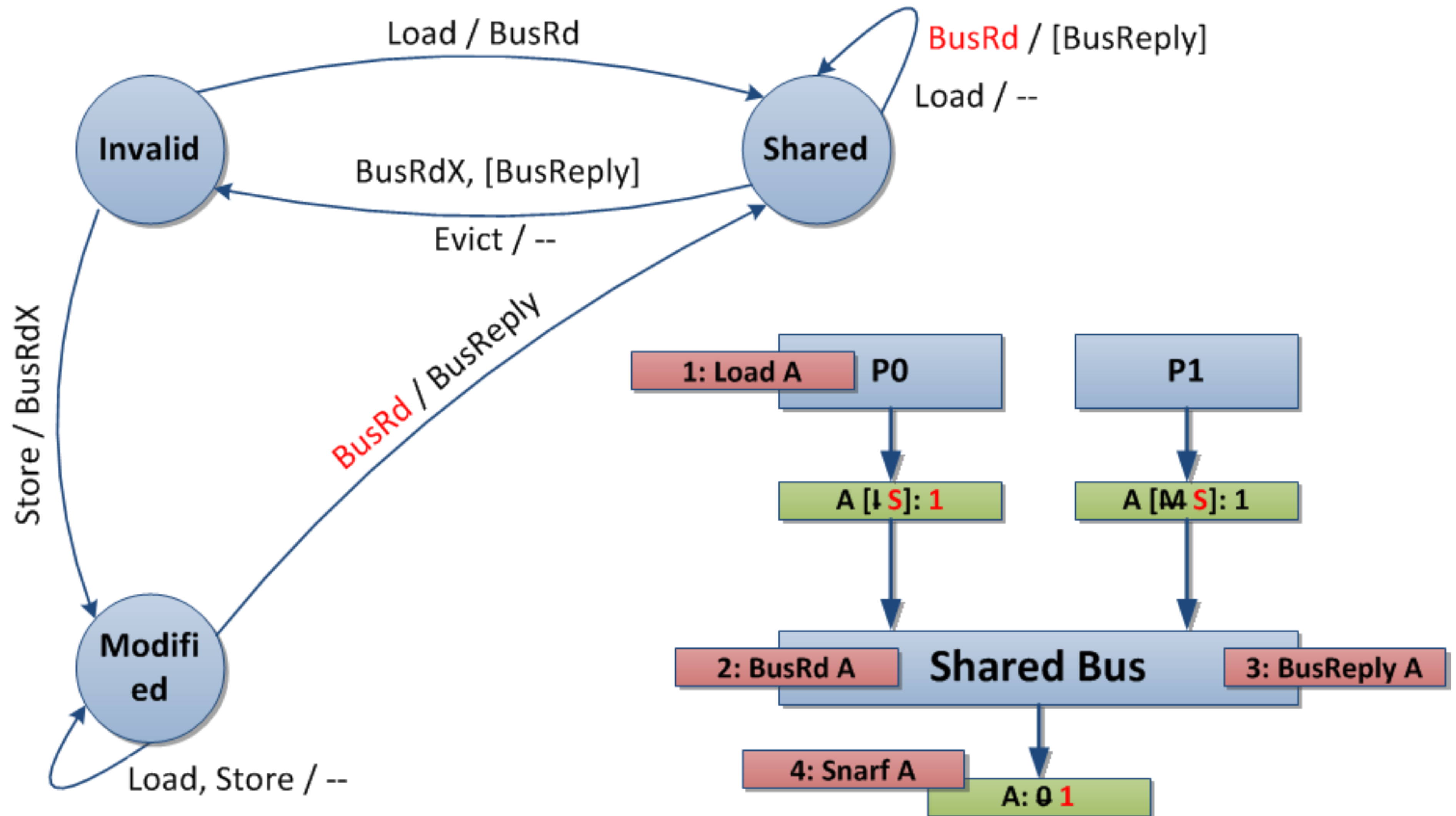
MODIFIED-SHARED-INVALID (MSI) PROTOCOL



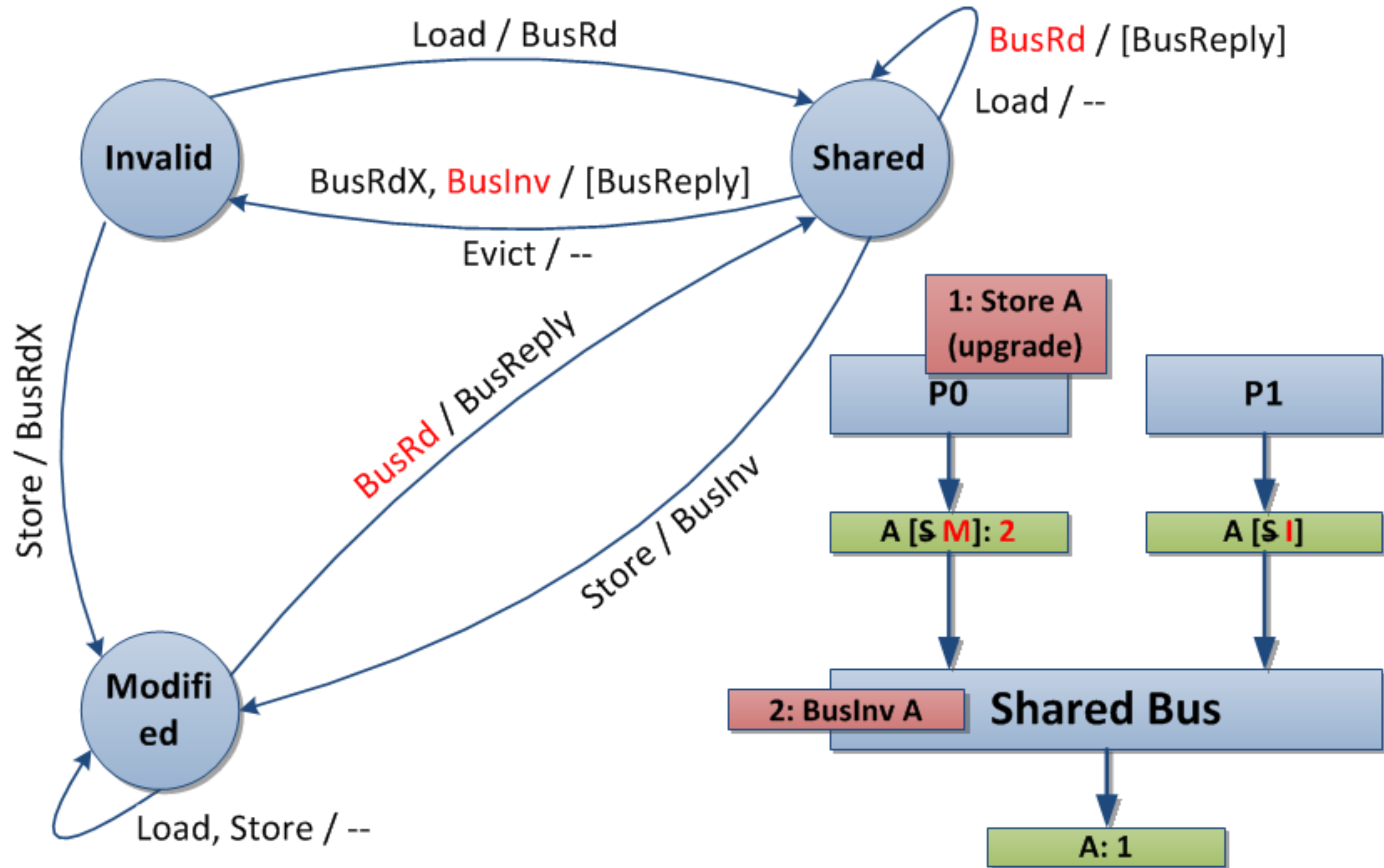
MODIFIED-SHARED-INVALID (MSI) PROTOCOL



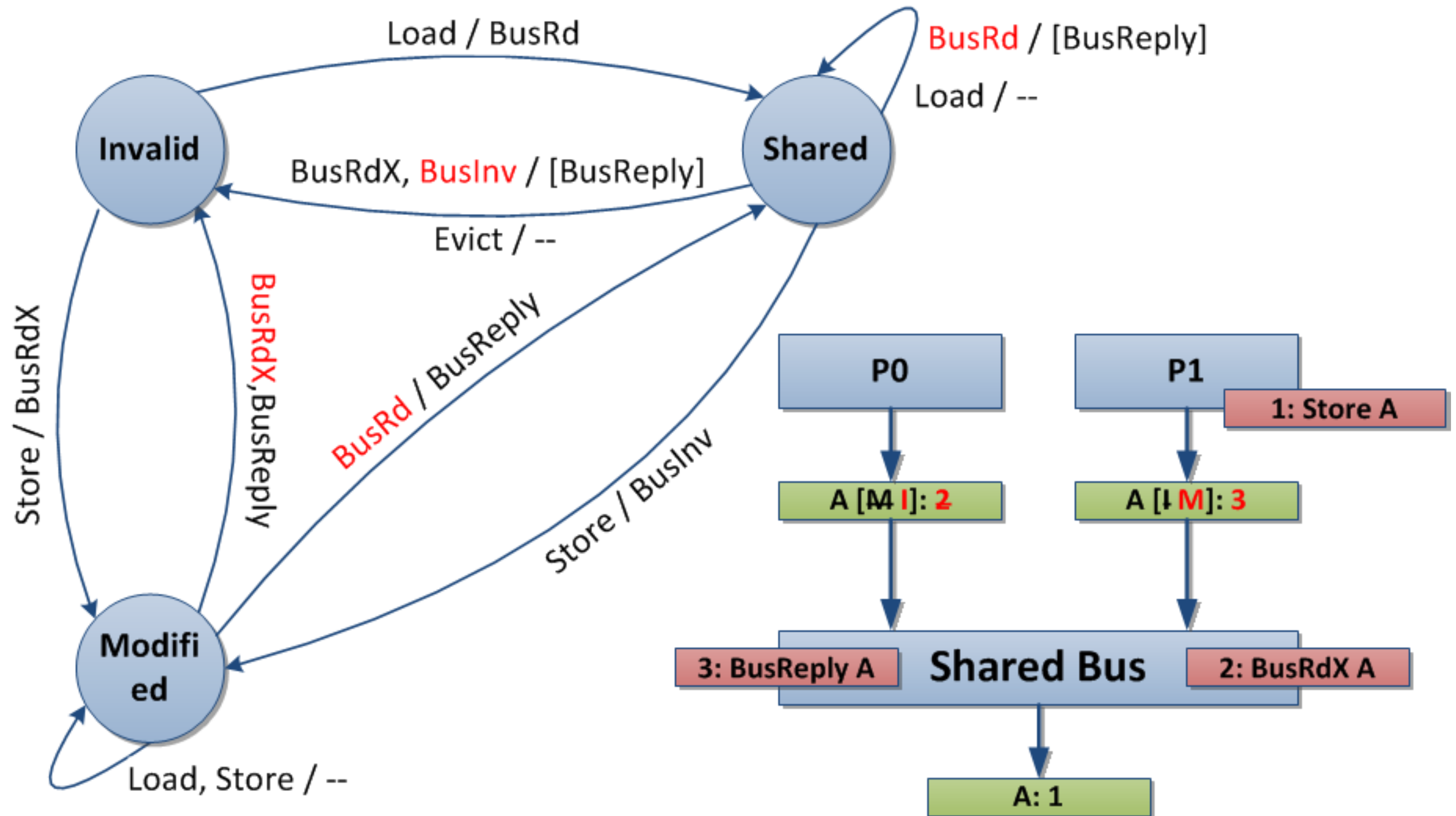
MODIFIED-SHARED-INVALID (MSI) PROTOCOL



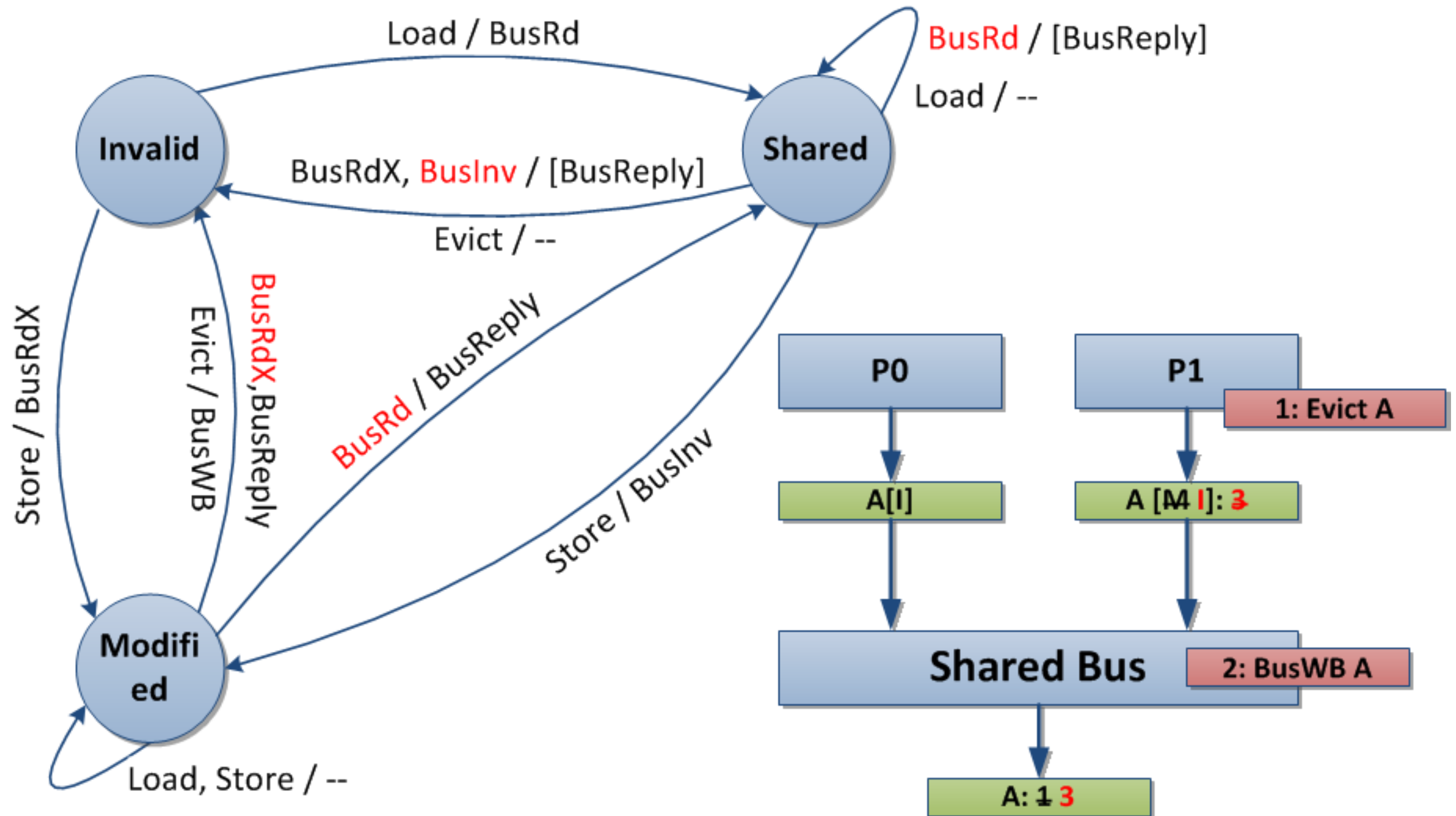
MODIFIED-SHARED-INVALID (MSI) PROTOCOL



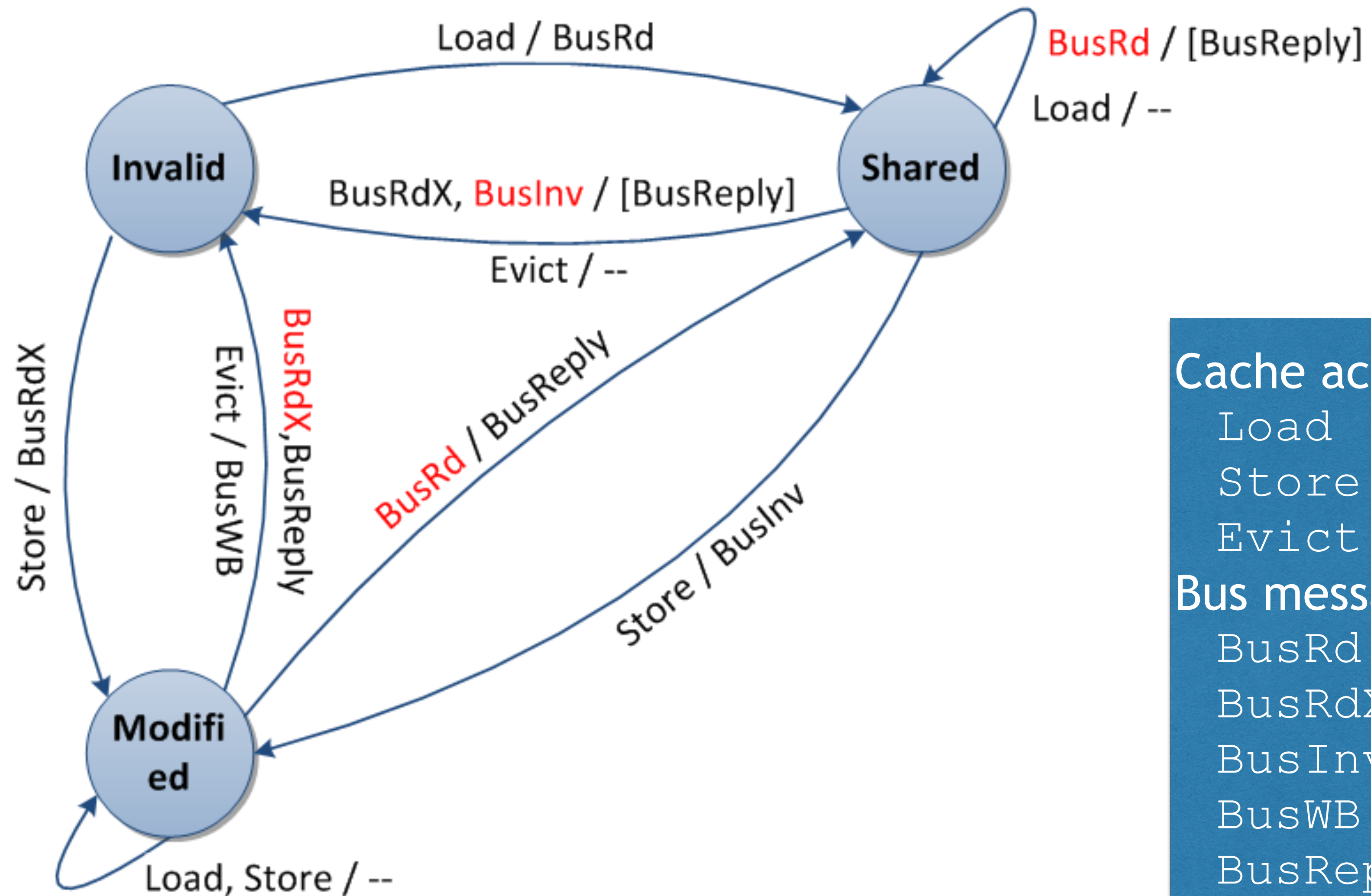
MODIFIED-SHARED-INVALID (MSI) PROTOCOL



MODIFIED-SHARED-INVALID (MSI) PROTOCOL



MODIFIED-SHARED-INVALID: SUMMARY



Cache actions

Load
Store
Evict

Bus messages

- BusRd
- BusRdX
- BusInv
- BusWB
- BusReply

PROTOCOL ISSUES & OPTIMIZATIONS

UPDATE VS. INVALIDATE

True sharing: multiple processes sharing the same part of a cache line

False sharing: unused part of a cache line causes sharing effects

- Cache lines are unnecessarily shared

- Performance degradation

Invalidation is bad when:

- Single producer and many consumers of data

Update is bad when:

- Multiple writes by one CPU before read by another

- Junk data accumulates in large caches (e.g., process migration)

COHERENCE DECOUPLING

HUH, CHANG, BURGER, SOHI @ ASPLOS2004

After invalidate, keep stale data around

On subsequent read, speculatively supply stale value

Confirm speculation with a normal read operation

Need a branch-prediction-like rewind mechanism

Completely solves false sharing problem

Not exactly: latency penalty addressed, but not the traffic overhead

Also addresses “silent”, “temporally-silent” stores

Silent stores: stores which write a value matching the one already stored at the memory location

Temporally silent stores: stores which change data at memory location only temporally, with another subsequent store restoring the old value

Update-like mechanisms to improve prediction

Examples of update heuristics in the paper

E.g., piggy-back value of 1st write on invalidation message

MESI PROTOCOL (AKA ILLINOIS)

MSI suffers from frequent read-upgrade sequences

Leads to two bus transactions, even for private blocks

Uniprocessors don't have this problem

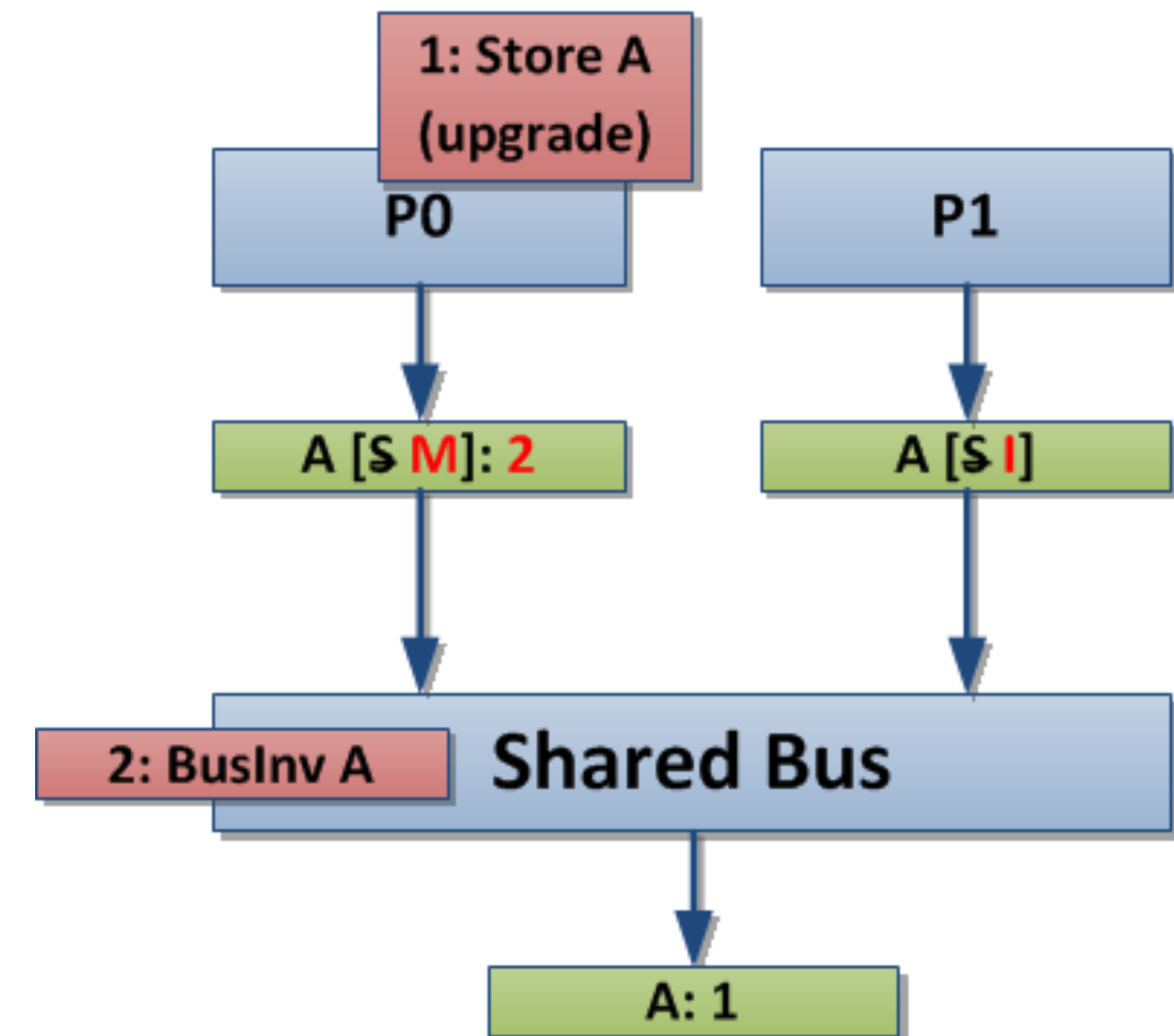
Solution: add an “**exclusive**” state

Exclusive - only one copy; writable; clean

Can detect exclusivity when only memory provides reply to a read

Upon store, transition to Modified to indicate data is dirty

No need for a $BUSWB$ from Exclusive



MESI PROTOCOL (AKA ILLINOIS)

Note that based on reply, different transitions are now possible

Invalid + Load

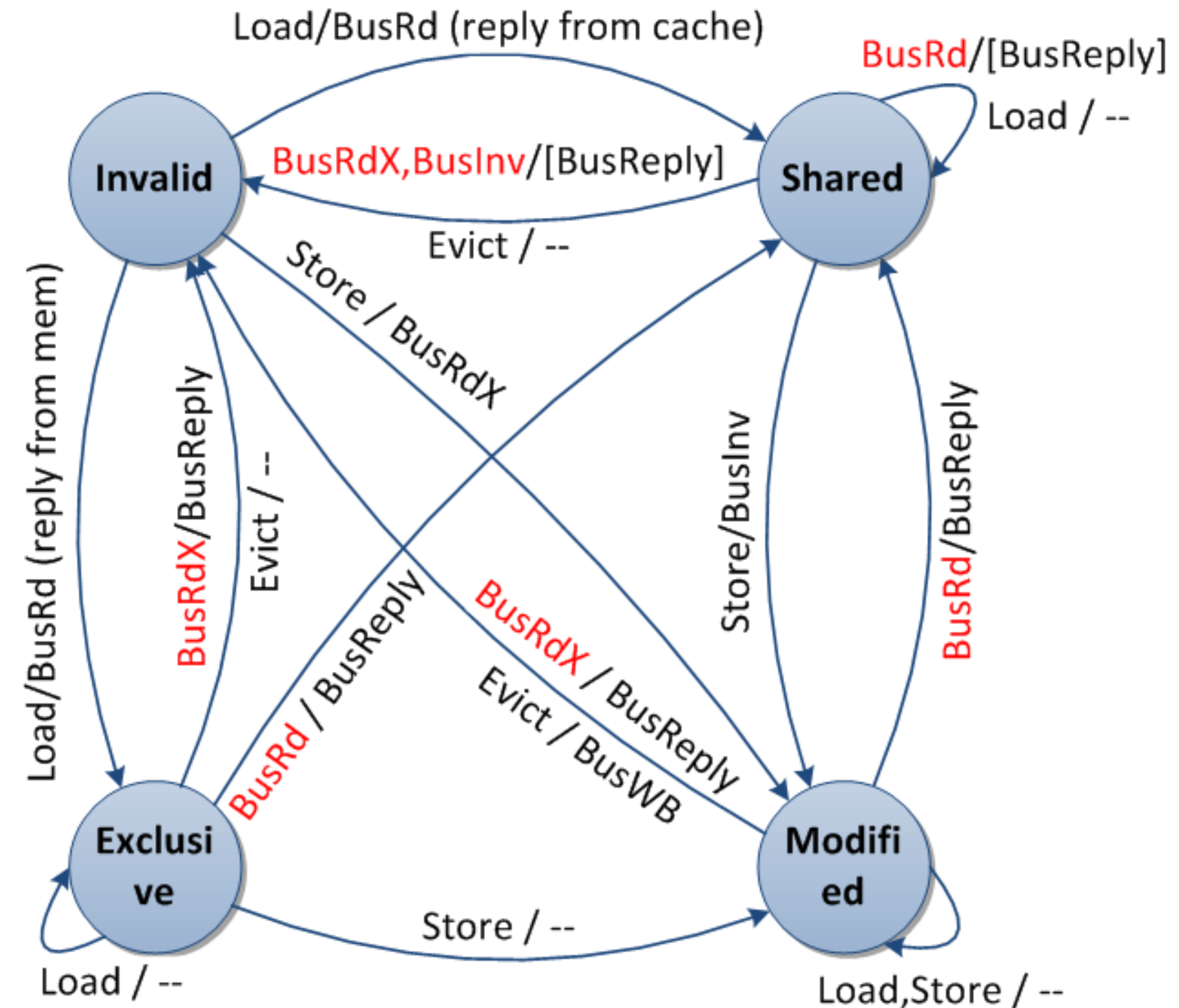
MESI must write back to memory on M->S transitions

Protocol allows “silent” evicts from shared state, thus dirty blocks otherwise might get lost

But, writebacks might be a waste of bandwidth

E.g., for subsequent stores

Common case in producer-consumer use models



MOESI PROTOCOL

Solution: add an “owned” state

Owned: shared, but dirty

Only one owner (others enter S)

Entered on M->S transition

Owner is responsible for writeback upon eviction

[Sweazey & Smith @ ISCA86]

States

M - modified (dirty)

O - owned (dirty but shared)

E - exclusive (clean unshared)

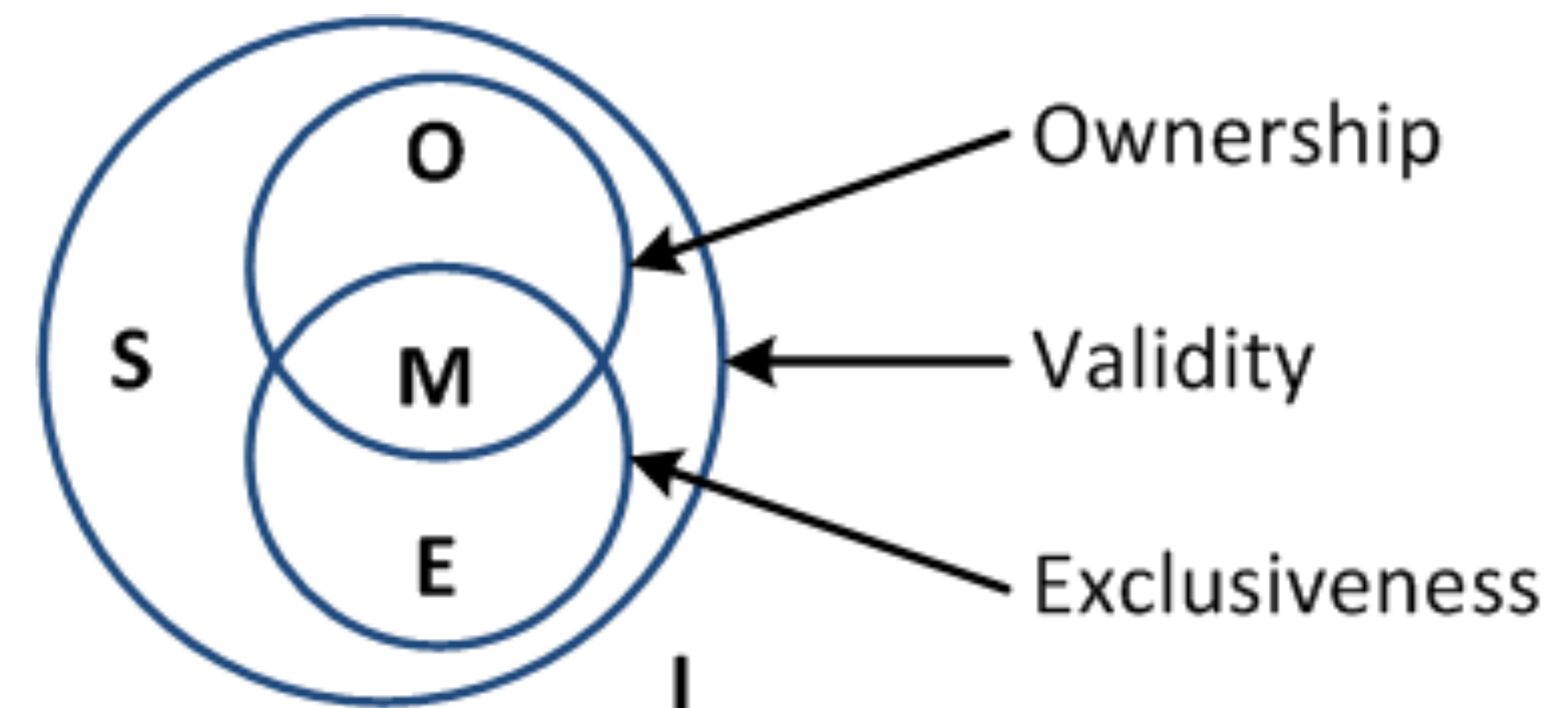
S - shared

I - invalid

All variants

MSI, MESI, MOSI, MOESI, MESIF (F=Forward)

F state: specialized form of the S state, indicates that a cache should act as a designated responder for a given CL. A cache line request will be responded to only by the cache holding the line in the F state => less responses



NON-ATOMIC STATE TRANSITIONS

Operations involve multiple actions

- Look-up cache tags

- Bus arbitration

- Check for write-back

Even if bus is atomic, overall set of actions is not atomic

=> Race conditions among multiple operations

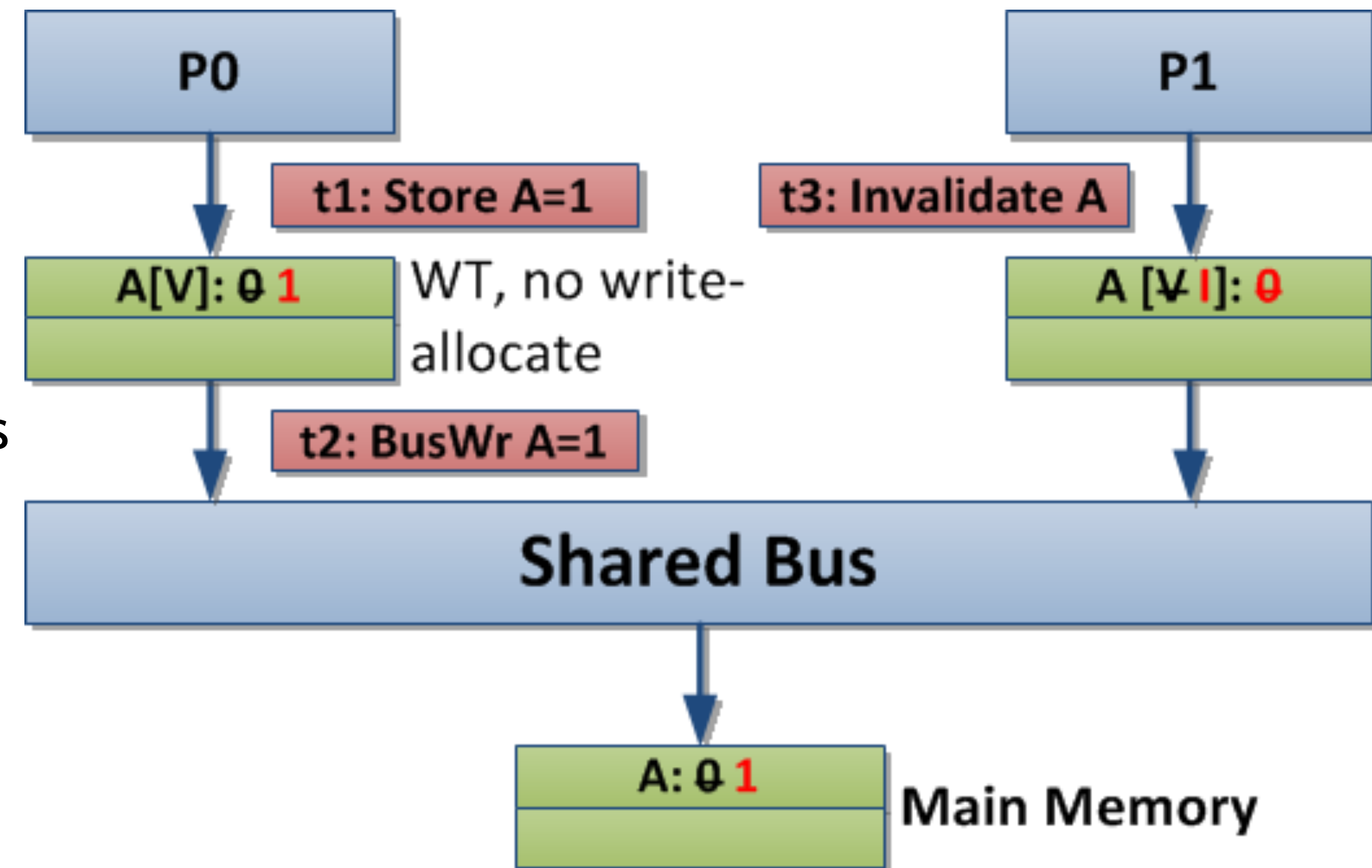
Suppose P0 and P1 attempt to write CL

Each decides to issue `BusUpgr` to allow S->M

Issues

- Handle requests for other blocks while waiting to acquire bus

- Must handle requests for this CL



SCALABILITY PROBLEMS OF SNOOPY COHERENCE

Prohibitive bus bandwidth

- Required bandwidth grows with number of Ps

- .. but available BW per bus is fixed

- Adding busses makes serialization/ordering hard

Prohibitive processor snooping bandwidth

- All caches do tag lookup when ANY processor accesses memory

- Inclusion limits this to LLC (last level cache), but still lots of lookups

Upshot: bus-based coherence doesn't scale beyond 8-16 Ps