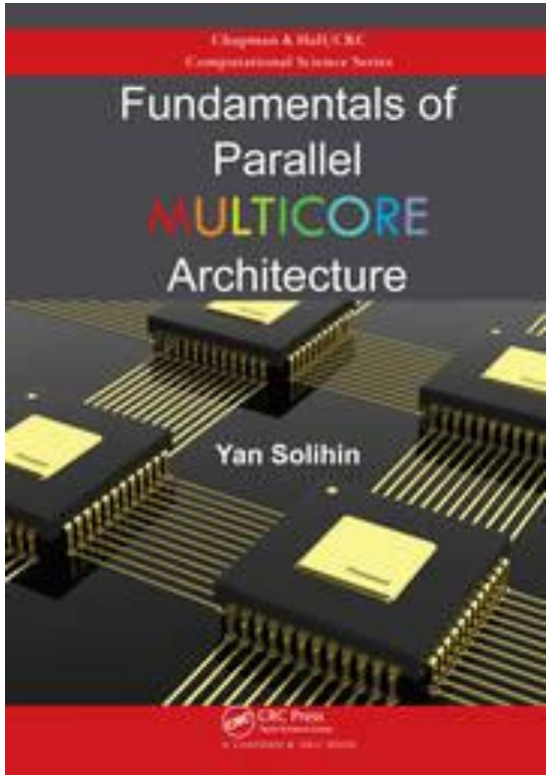
TU WIEN Informatics

E2. Advanced Computer Architecture

Cache Coherency

Daniel Mueller-Gritschneder

Sources



Multi-Cores.

Literature: **Yan Solihin : Fundamentals of Parallel Multicore Architecture, 2015**

Online from TU Wien library:
https://catalogplus.tuwien.at/permalink/f/8j3js/UTW_alma51112913160003336

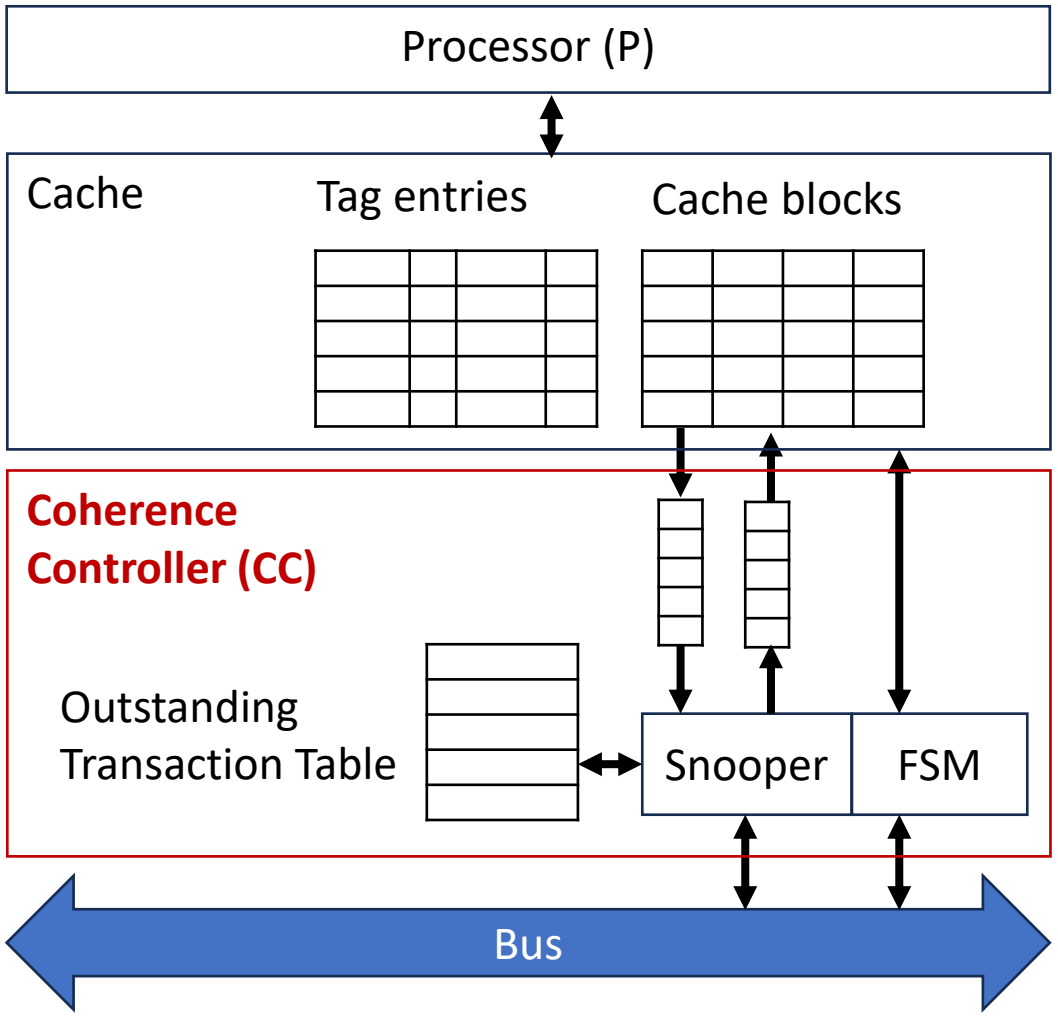
Write propagation & Transaction serialization

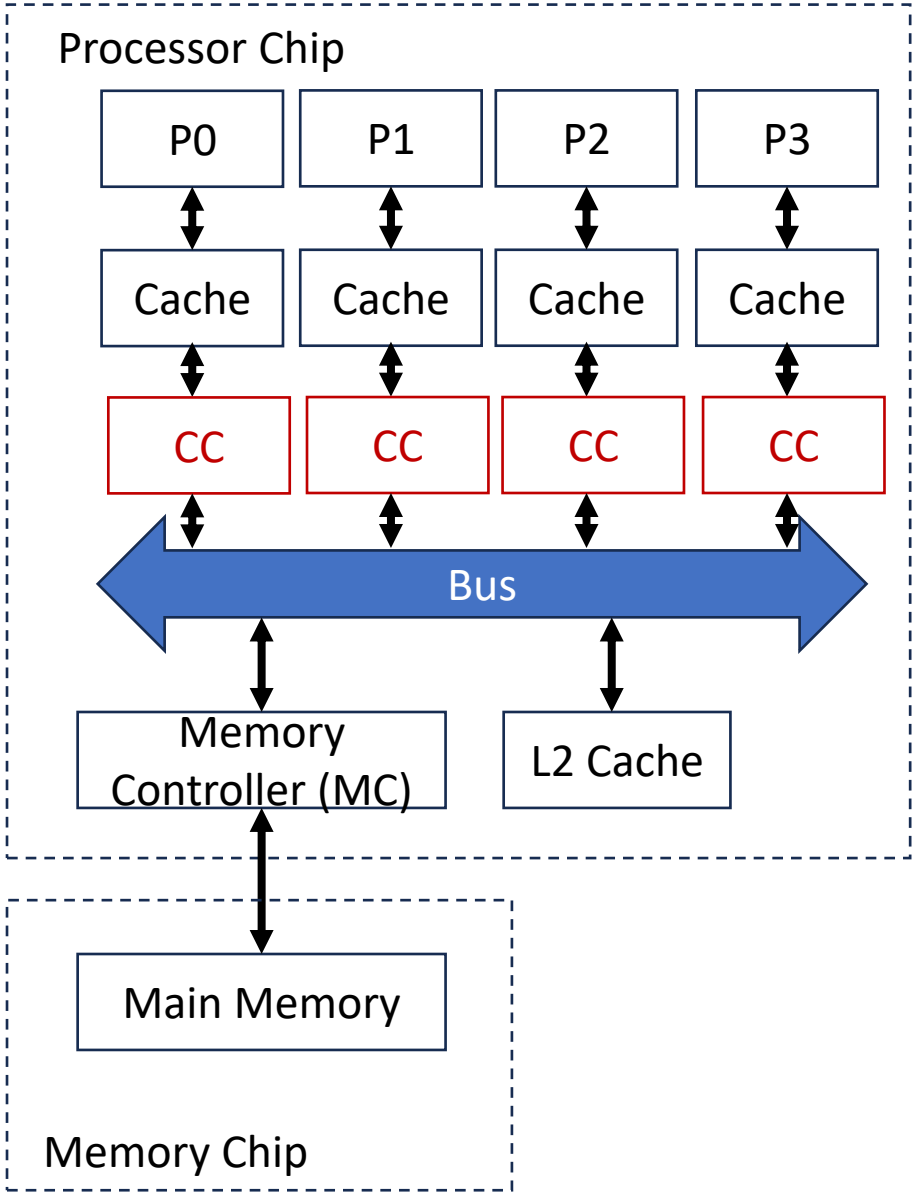
- Cache coherence : Support for coherent view of data values in multiple caches

- Requires:
  - *Write propagation: P*ropagate changes in one cache to other caches
  - *Transaction serialization:* Multiple operations (reads or writes) to a single memory location are seen in the same order by all processors

E2.1 Cache Controllers

Coherence Controller





Outstanding transaction table & Snooper

- Outstanding transaction table
  - In the split transaction bus, multiple requests to different addresses can be placed on the bus even when the oldest request has not obtained its data.
  - keeps track of bus transactions that have not completed.

- Bus snooper.
  - Snoops each bus transaction
  - checks the cache tag array to see if it has the block that is involved in the transaction
  - checks the current state of the block (if the block is found)
  - changes the state of the block
  - New state of block -> a finite state machine (FSM) implementing the cache coherence protocol
  - Data that is sent out is placed in a queue called the write back buffer

E2.3 Coherence Protocol for Write Through Caches

Coherence Protocol for Write Through Caches - Requests

- The simplest cache coherence: write through caches.
- Requests from the processor side, as well as from the bus side are snooped by the snooper.

Processor requests to the cache include:

1. **PrRd**: processor-side request to read to a cache block.
2. **PrWr**: processor-side request to write to a cache block.

Snooped requests to the cache include:

1. **BusRd**: snooped request that indicates there is a read request to a block made by another processor.
2. **BusWr**: snooped request that indicates there is a write request to a block made by another processor. In the case of a write through cache, the BusWr is a write through to the main memory performed by another processor.

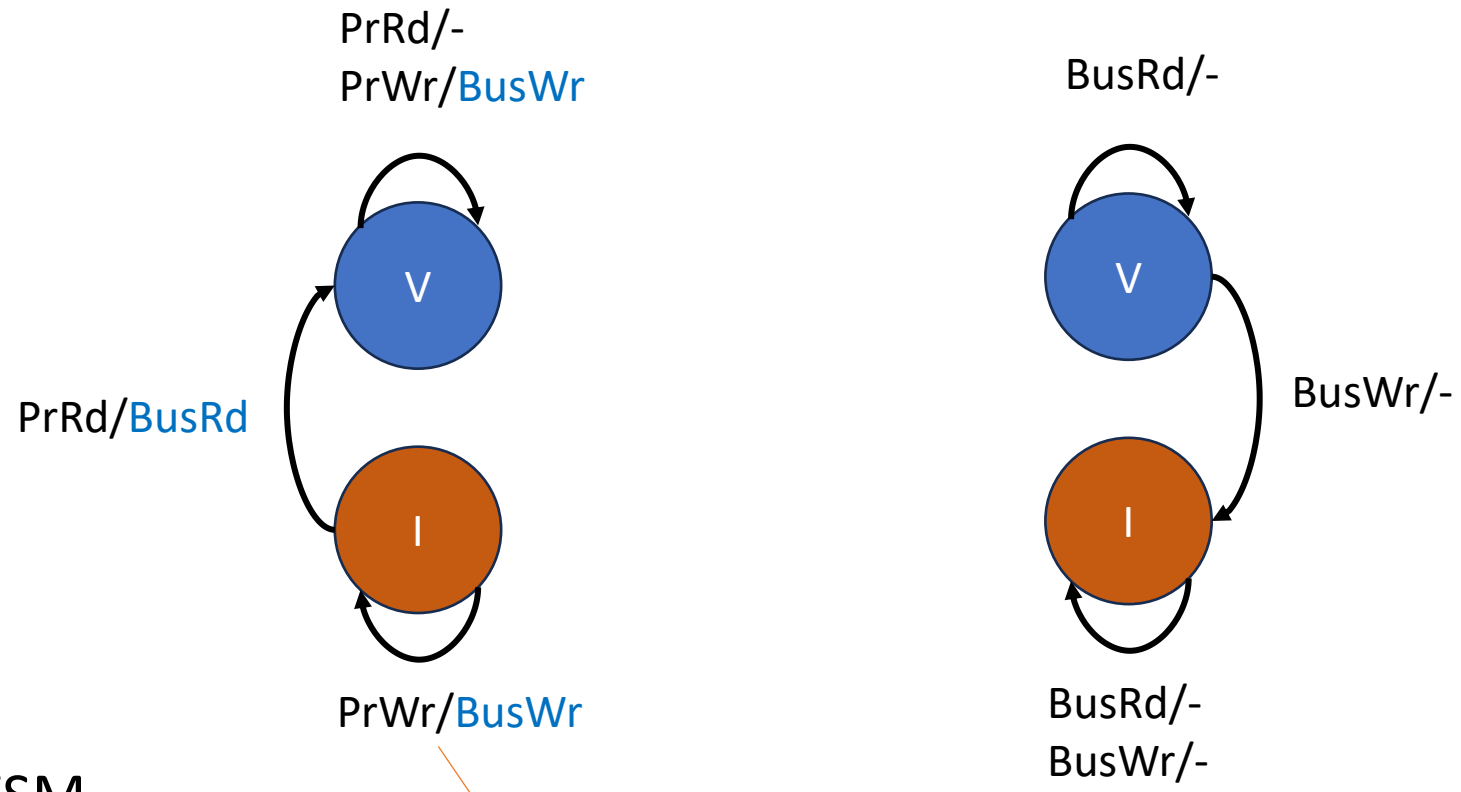
Coherence Protocol for Write Through Caches – Cache Block States

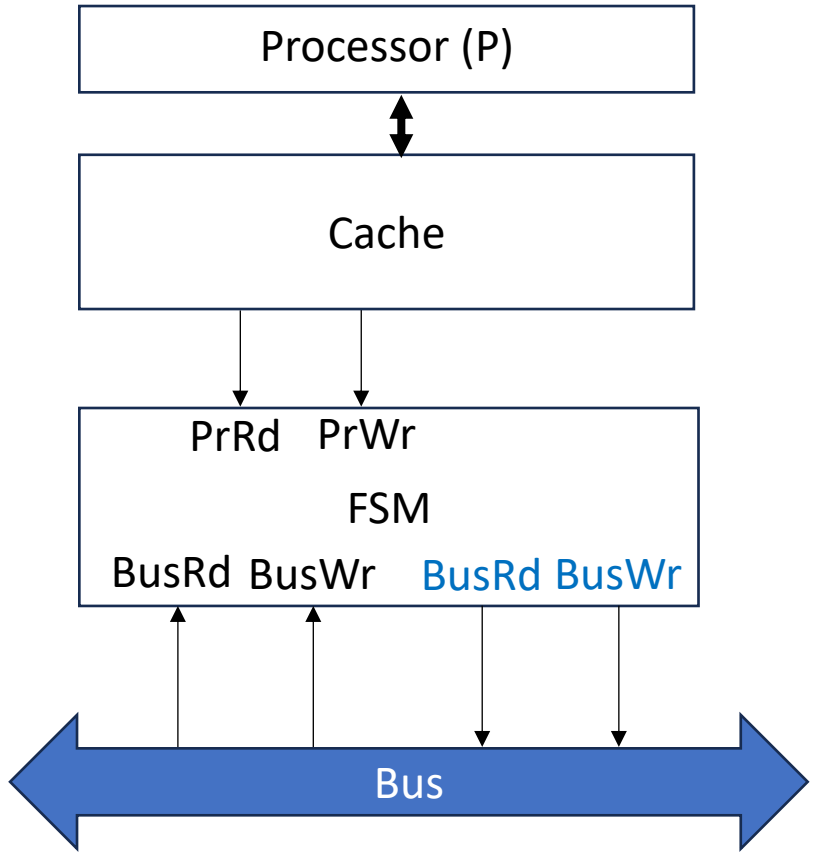
Each cache block has an associated state which can have one of the following values:

1. **Valid** (V): the cache block is valid and *clean*, meaning that the cached value is the same with that in the lower-level memory component (in this case the main memory).
2. **Invalid** (I): the cache block is invalid. Accesses to this cache block will generate cache misses.

FSM for Coherence Protocol for Write Through Caches – Snooper FSM

- Processor Side Request
- Bus Side Request





The processor in the book uses a write around (write no-allocate) policy so the value is directly updated in the memory and not fetched to the cache (remains invalid)

Cache Coherency Write through - Example 2

Rx or Wx, where R stands for a read request, W stands for a write request, and x stands for the ID of the processor making the request

	Request	P1	P2	P3	Bus Request	Data Transfer
0	Initially	–	–	–	–	–
1	R1	V	–	–	BusRd	Mem -> P1‘s cache
2	W1	V	–	–	BusWr	P1‘s cache -> Mem (Write through)
3	R3	V	–	V	BusRd	Mem -> P3‘s cache
4	W3	I	–	V	BusWr	P3‘s cache -> Mem (Write through)
5	R1	V	–	V	BusRd	Mem -> P1‘s cache
6	R3	V	–	V	-	-
7	R2	V	V	V	BusRd	Mem -> P2‘s cache

E2.3 MSI Protocol with Write Back Caches

MSI Protocol with Write Back Caches - Requests

In the MSI protocol, processor requests to the cache include:

1. **PrRd**: processor-side request to read from a cache block.
2. **PrWr**: processor-side request to write to a cache block.

Bus-side requests include:

1. **BusRd**: snooped request that indicates there is a read request to a cache block made by another processor.
2. **BusRdX**: snooped request that indicates there is a *read exclusive* (write) request to a cache block made by another processor.
3. **Flush**: snooped request that indicates that an entire cache block is written back to the main memory by another processor.

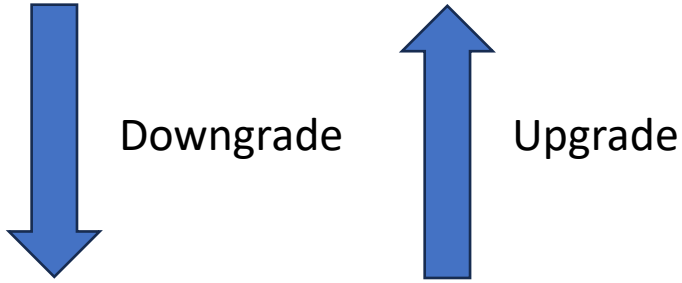
MSI Protocol with Write Back Caches - Cache Block States

Each cache block has an associated state which can have one of the following values:

1. **Modified** (M): the cache block is valid in only one cache, and the value is (likely) different than the one in the main memory. This state extends the meaning of the dirty state in a write back cache for a single processor system, except that now it also implies exclusive ownership. Whereas dirty means the cached value is potentially different than the value in the main memory, modified means both the cached value is potentially different than the value in the main memory, and it is cached only in one location.
2. **Shared** (S): the cache block is valid, potentially *shared* by multiple processors, and is clean (the value is the same as the one in the main memory). The shared state is similar to the valid state in the coherence protocol for write through caches.
3. **Invalid** (I): the cache block is invalid (either not cached, or cached but outdated).

MSI Protocol with Write Back Caches – State Transistions

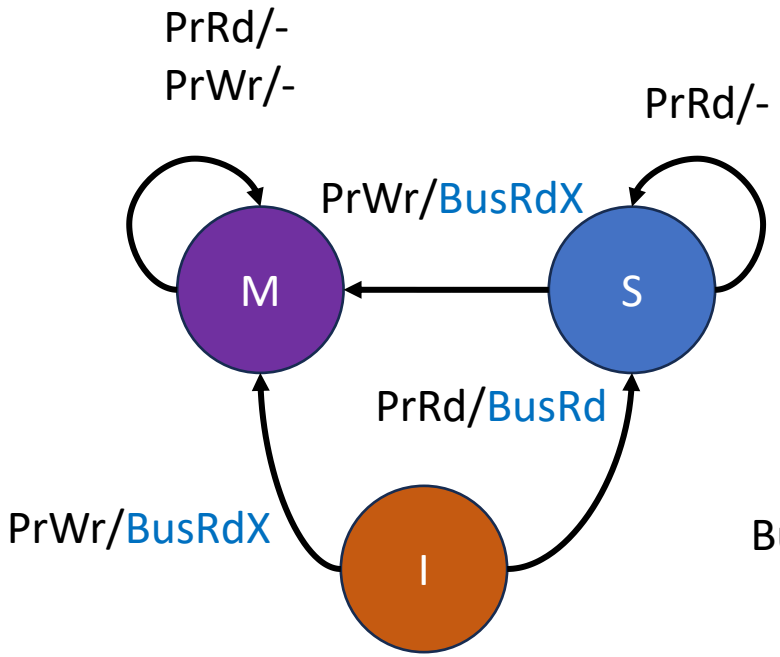
- Modified state (M): read/write permission
- Shared state (S): read/no-write permission
- Invalid state (I): no-read/no-write permission



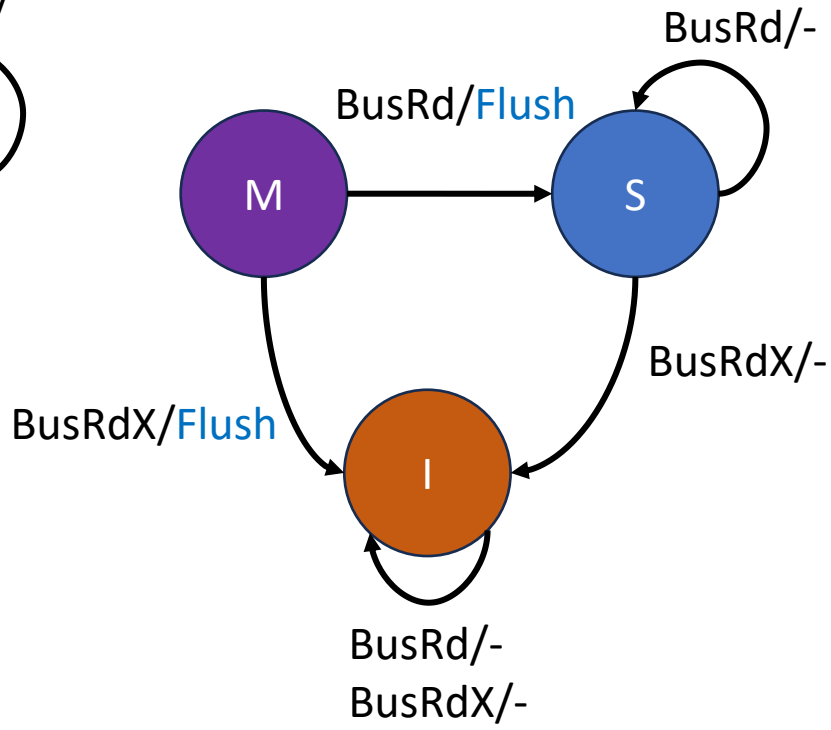
- Intervention: Downgrade to S state
- Invalidation: Downgrade to I state

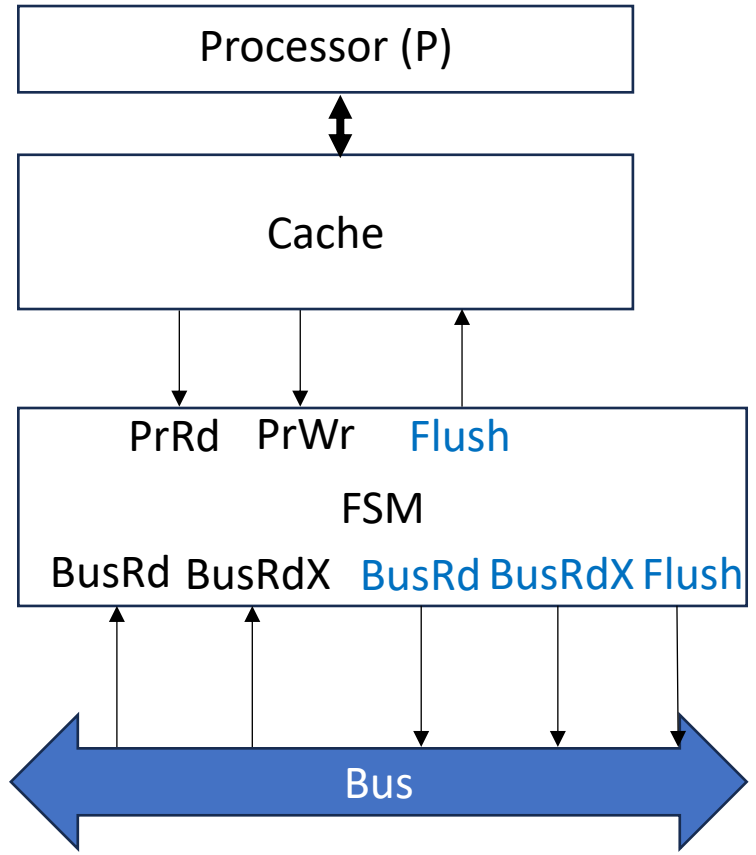
MSI Protocol with Write Back Caches - Snooper FSM

- Processor Side Request



- Bus Side Request





Cache Coherency MSI - Example

- Multi-threaded execution (MSI):

Thread 0 (P0):

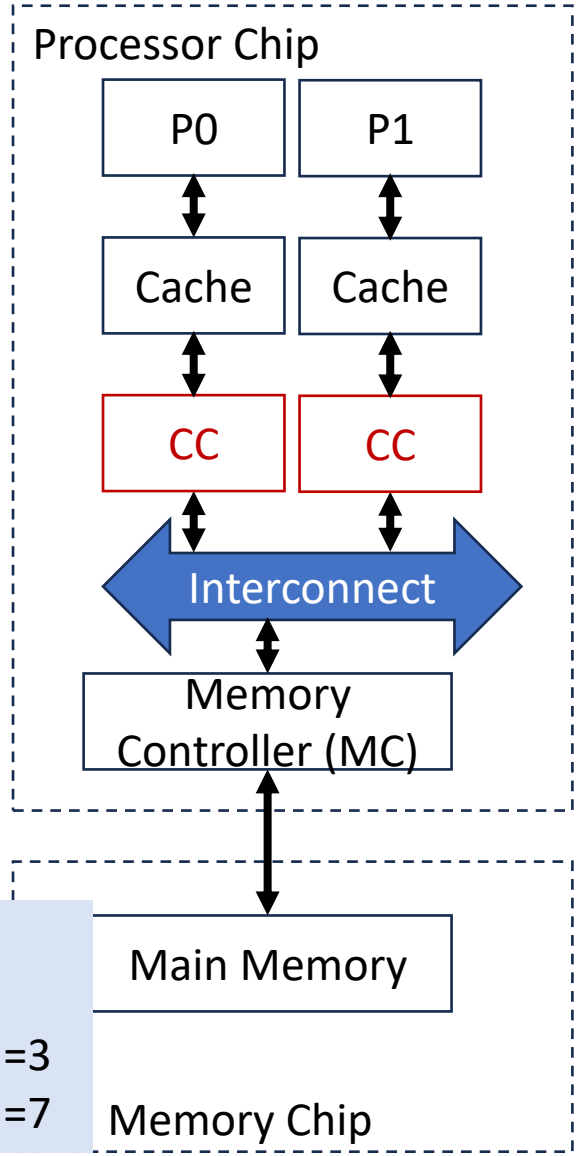
```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```

Thread 1 (P1):

```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```



0x1000: ?
….
0x2000: a[0]=3
0x2004: a[1]=7

Cache Coherency MSI - Example

- Multi-threaded execution (MSI):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

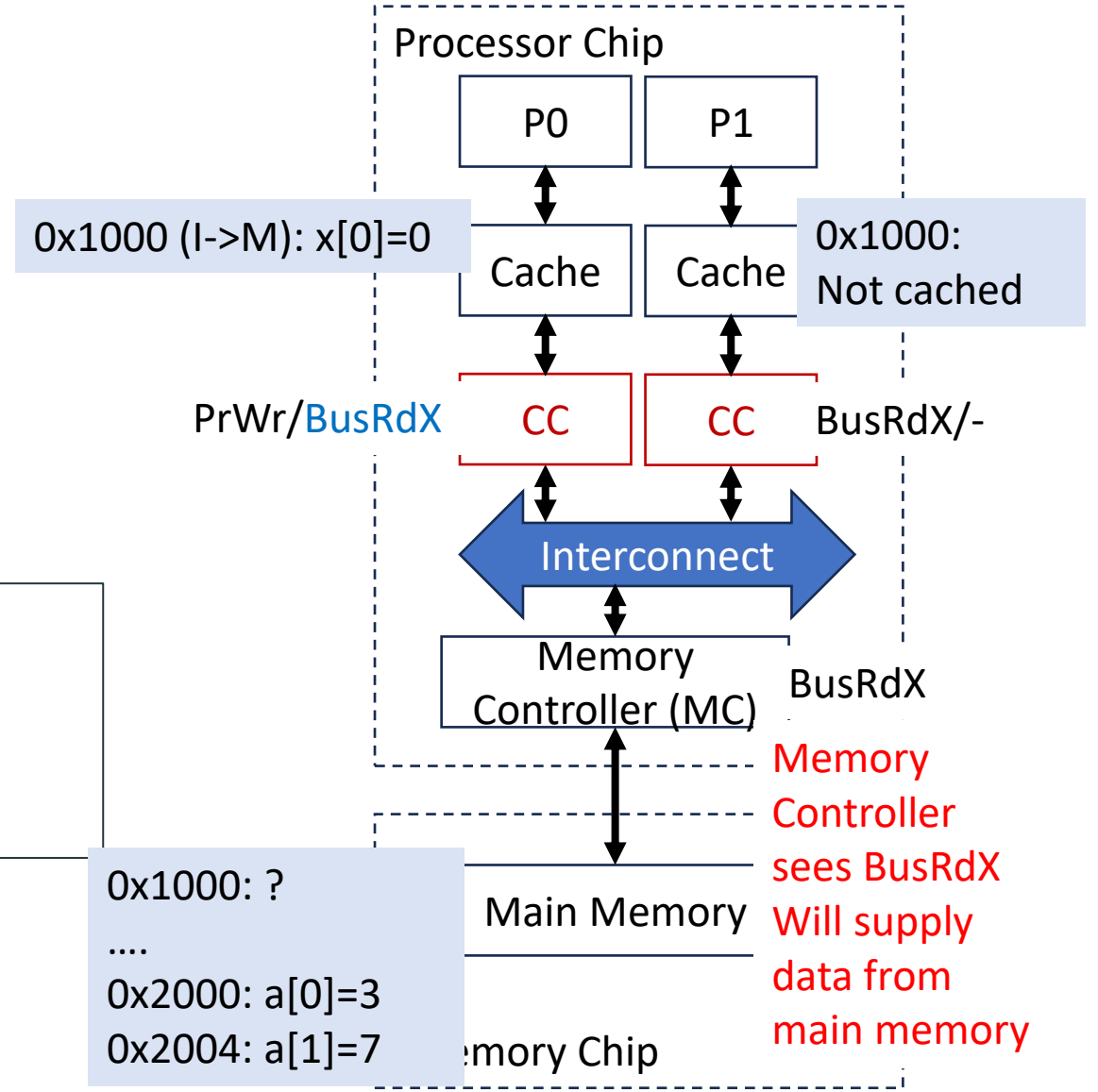
Write x[0]=0, miss,
fetch from Mem & write

```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Thread 1 (P1):

```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```



Cache Coherency MSI - Example

- Multi-threaded execution (MSI):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

Write x[0]=0, miss,
fetch from mem & write

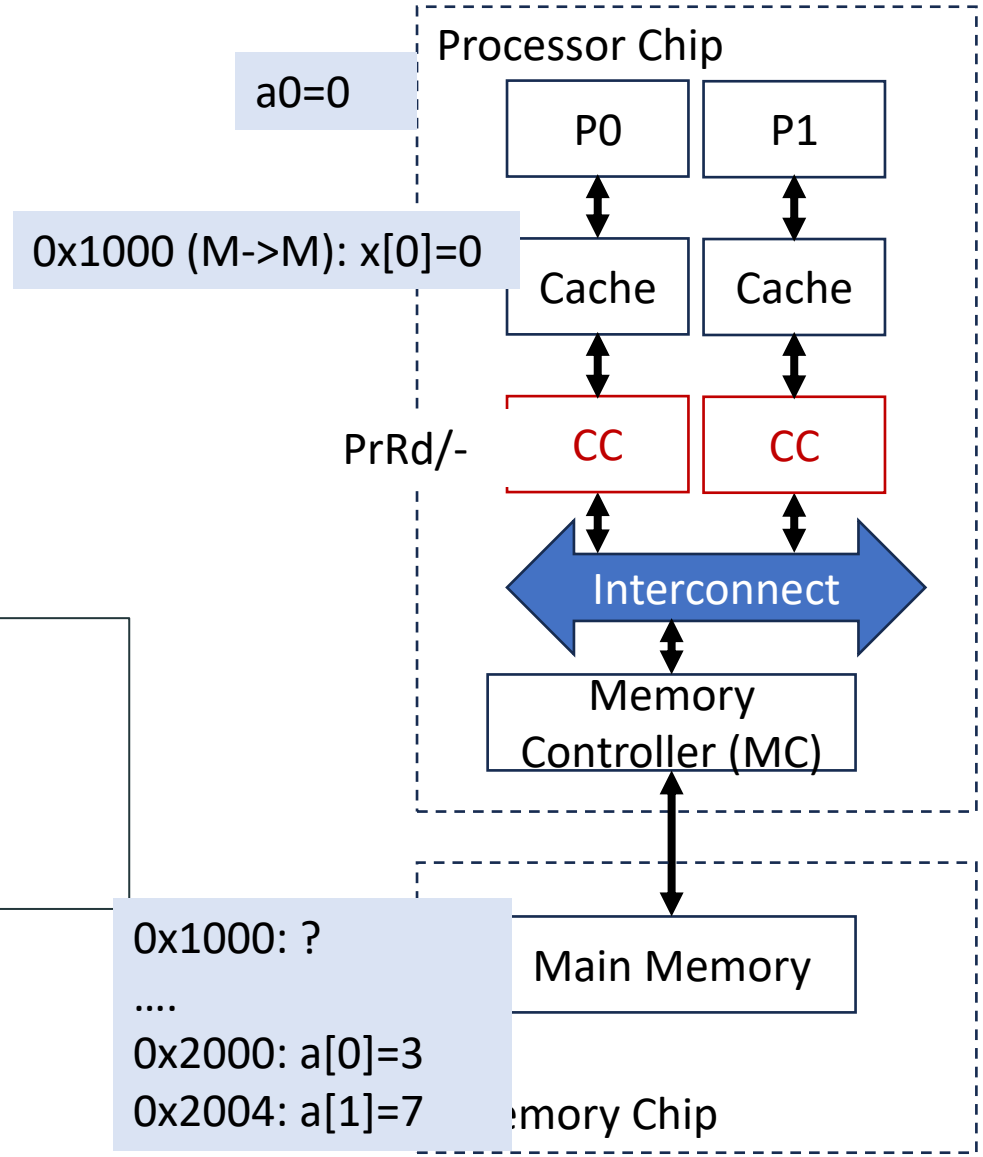
```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=0, hit

Thread 1 (P1):

```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```



Cache Coherency MSI - Example

- Multi-threaded execution (MSI):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

Write x[0]=0, miss,
fetch from mem & write

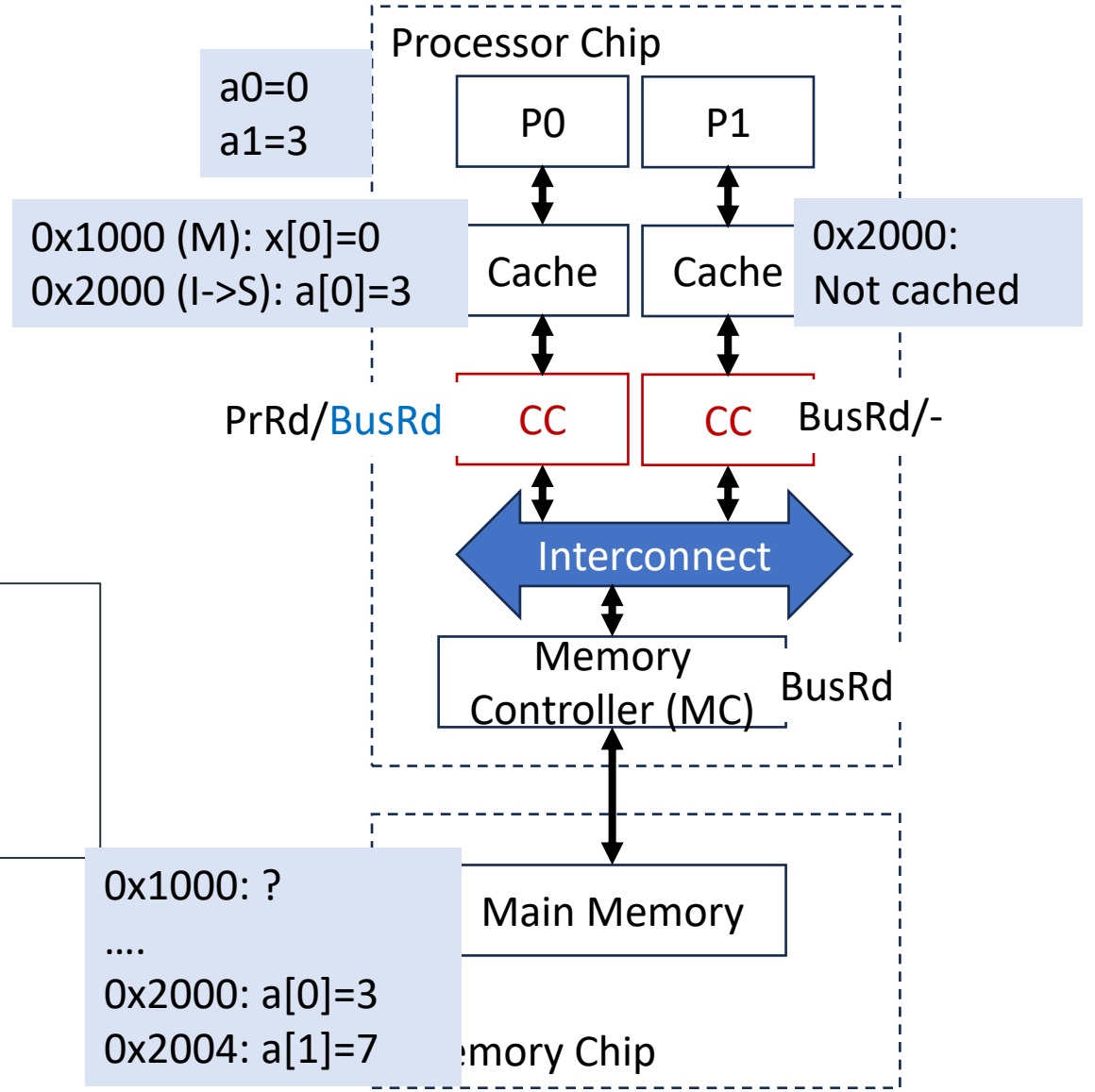
```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=0, hit
Read a[0]=3, miss -> fetch

```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```

Thread 1 (P1):

```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```



Cache Coherency MSI - Example

- Multi-threaded execution (MSI):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

Write x[0]=0, miss,
fetch from mem & write

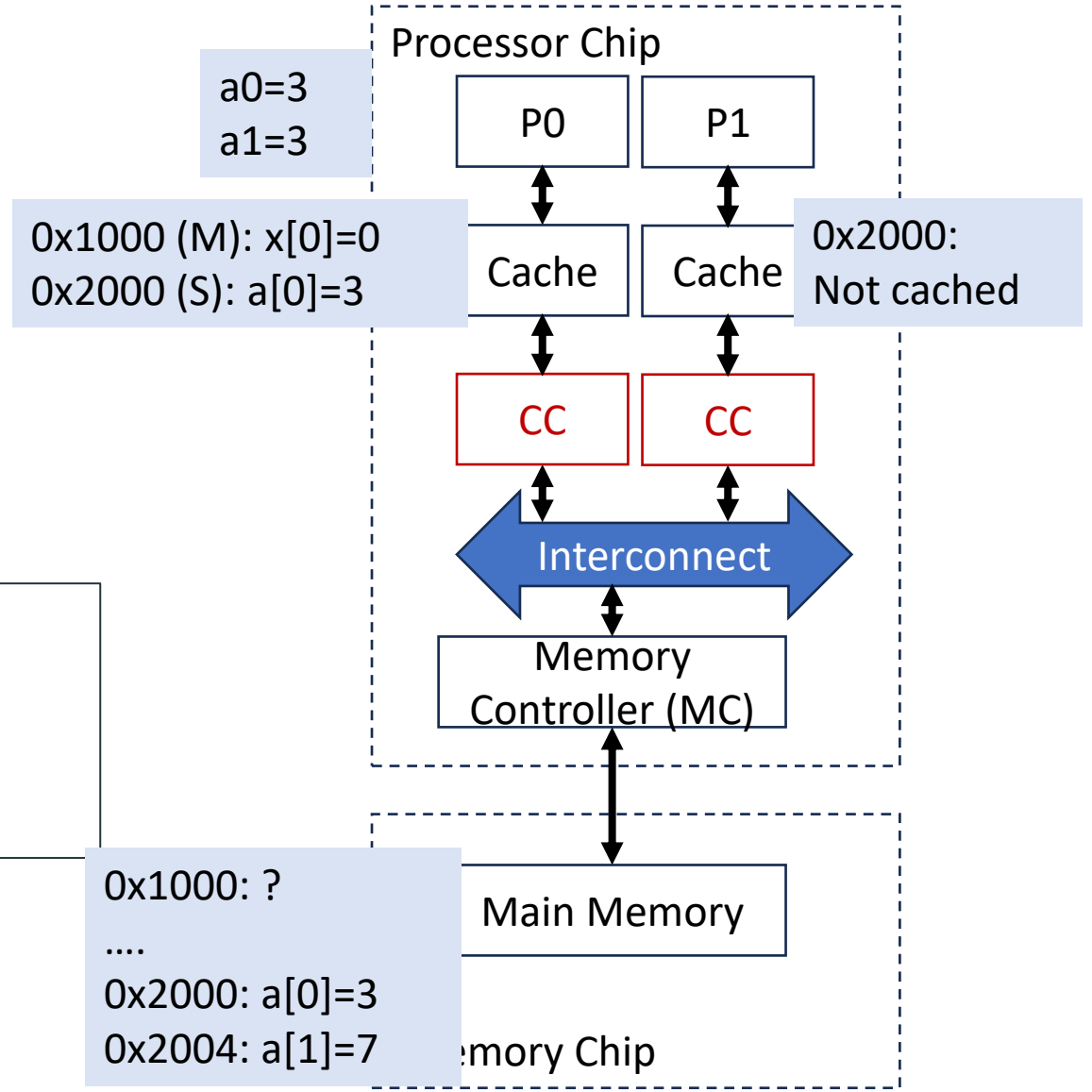
```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=0, hit
Read a[0]=3, miss -> fetch
x[0]=x[0]+a[0]

```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```

Thread 1 (P1):

```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```



Cache Coherency MSI - Example

- Multi-threaded execution (MSI):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

Write x[0]=0, miss,
fetch from mem & write

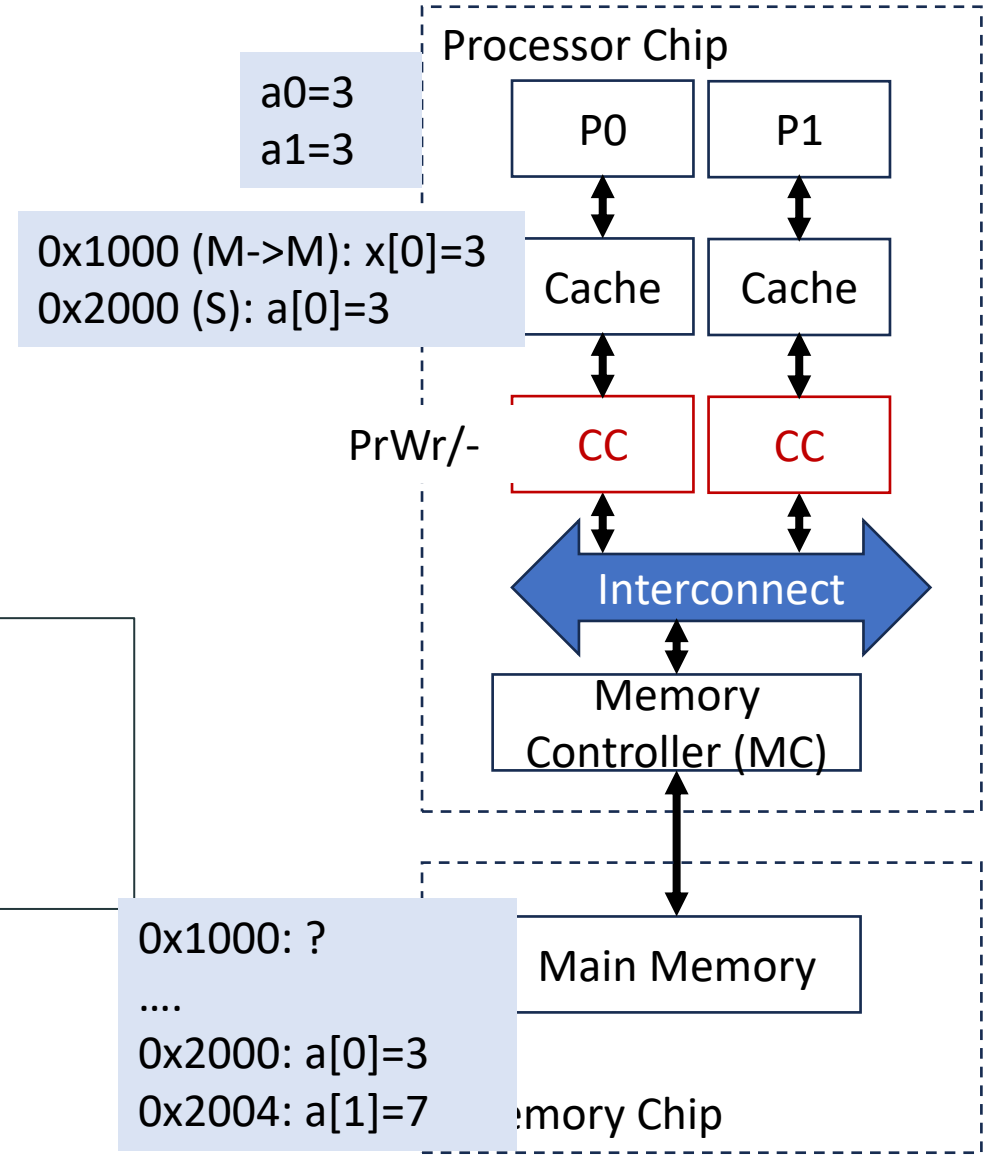
```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=0, hit
Read a[0]=3, miss -> fetch
x[0]=x[0]+a[0]
Write x[0]=3, hit

```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```

Thread 1 (P1):

```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```



Cache Coherency MSI - Example

- Multi-threaded execution (MSI):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

Write x[0]=0, miss,
fetch from mem & write

```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=0, hit
Read a[0]=3, miss -> fetch
x[0]=x[0]+a[0]
Write x[0]=3, hit

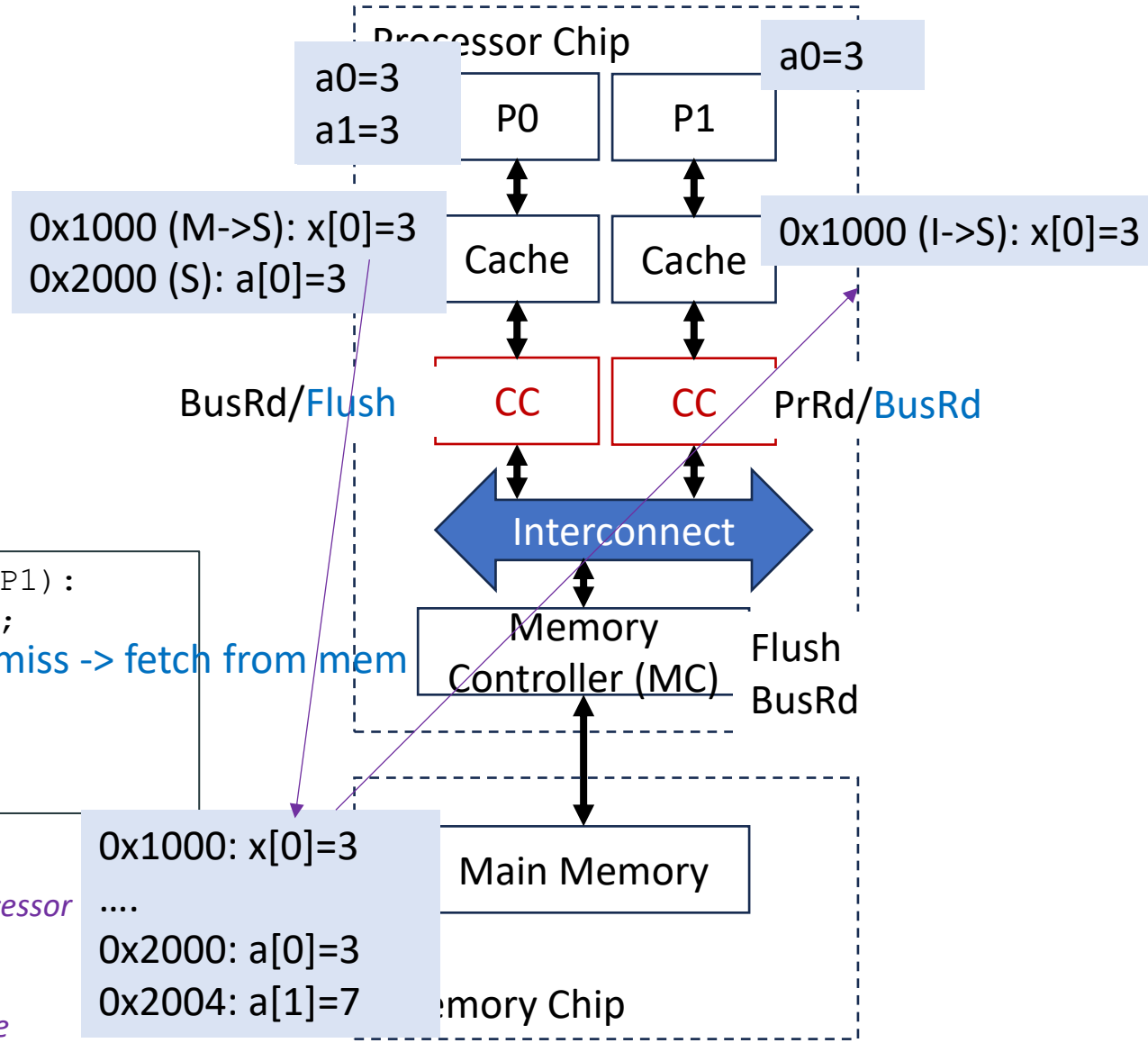
```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```

Thread 1 (P1):

```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0] miss -> fetch from mem

Flush: Value is written back to memory and read by other processor cache
Data transfer:
P0's cache -> Mem -> P1's cache



Cache Coherency MSI - Example

- Multi-threaded execution (MSI):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

Write x[0]=0, miss,
fetch from mem & write

```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=0, hit
Read a[0]=3, miss -> fetch
x[0]=x[0]+a[0]
Write x[0]=3, hit

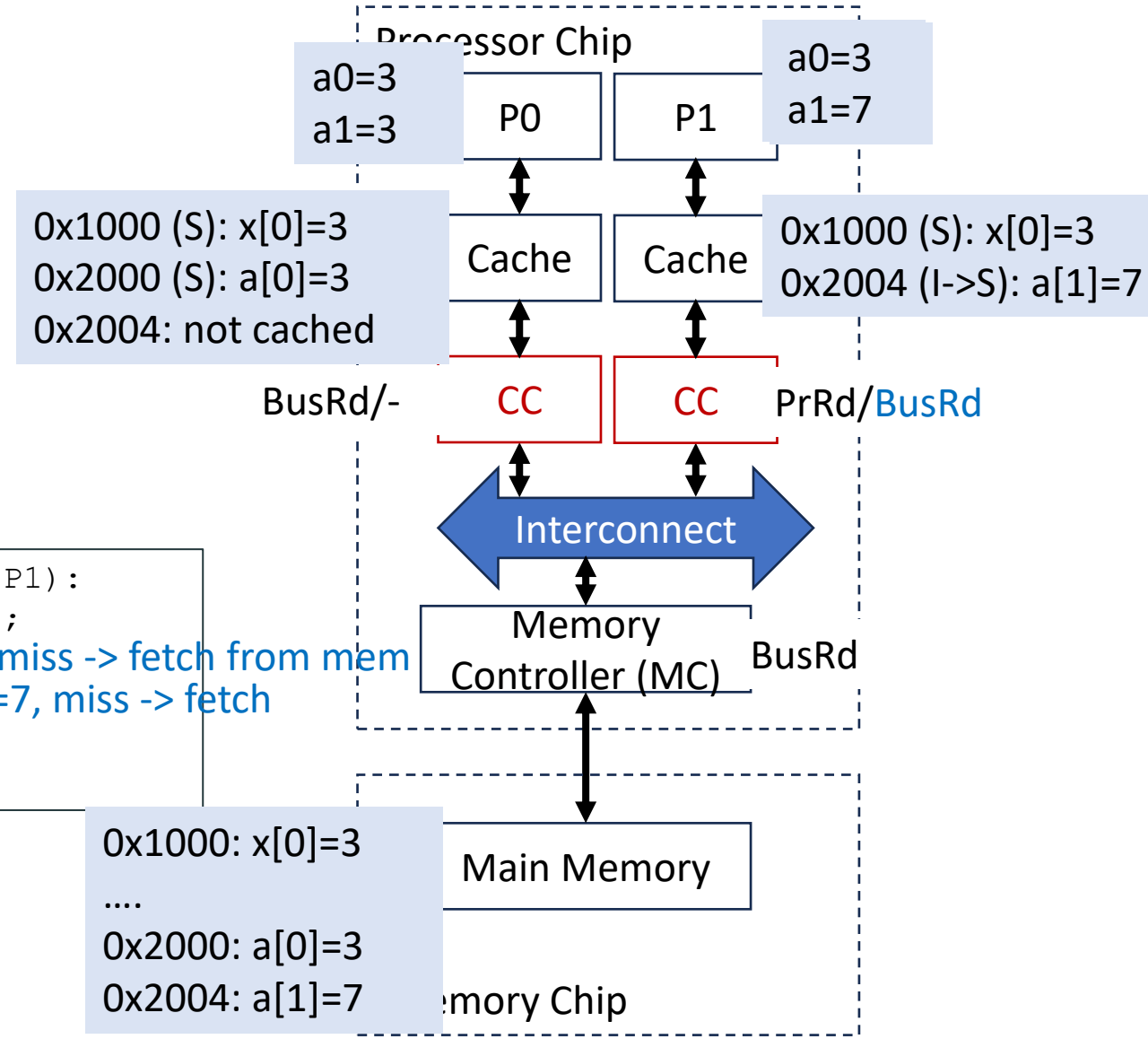
```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```

Thread 1 (P1):

```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0] miss -> fetch from mem

Read a[1]=7, miss -> fetch



Cache Coherency MSI - Example

- Multi-threaded execution (MSI):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

Write x[0]=0, miss,
fetch from mem & write

```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

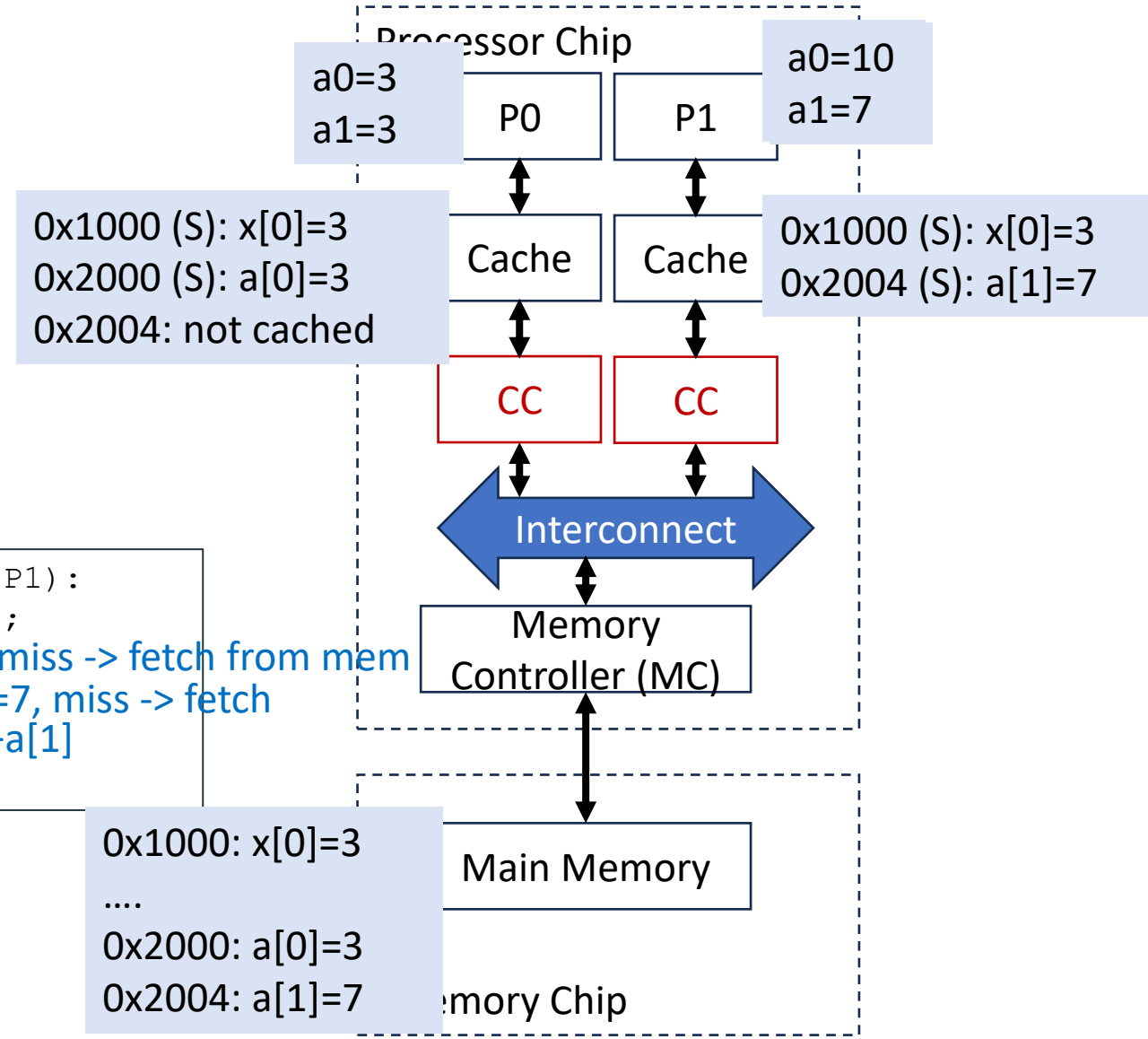
Read x[0]=0, hit
Read a[0]=3, miss -> fetch
x[0]=x[0]+a[0]
Write x[0]=3, hit

```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```

Thread 1 (P1):

```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0] miss -> fetch from mem
Read a[1]=7, miss -> fetch
x[0]=x[0]+a[1]



Cache Coherency MSI - Example

- Multi-threaded execution (MSI):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

Write x[0]=0, miss,
fetch from mem & write

```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

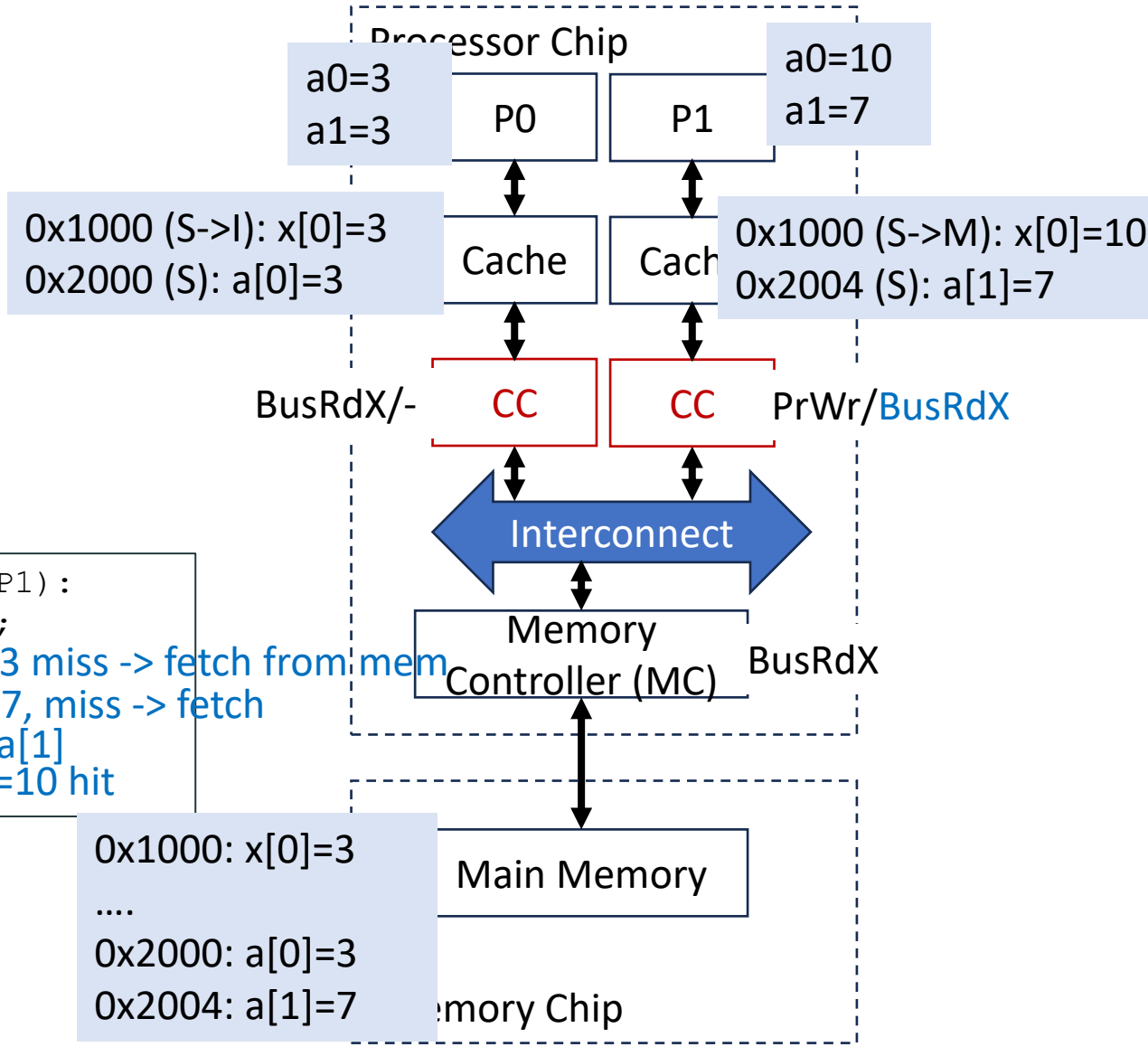
Read x[0]=0, hit
Read a[0]=3, miss -> fetch
x[0]=x[0]+a[0]
Write x[0]=3, hit

```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```

Thread 1 (P1):

```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=3 miss -> fetch from mem
Read a[1]=7, miss -> fetch
x[0]=x[0]+a[1]
Write x[0]=10 hit



Cache Coherency MSI - Example

- Multi-threaded execution (MSI):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

Write x[0]=0, miss,
fetch from mem & write

```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=0, hit
Read a[0]=3, miss -> fetch
x[0]=x[0]+a[0]
Write x[0]=3, hit

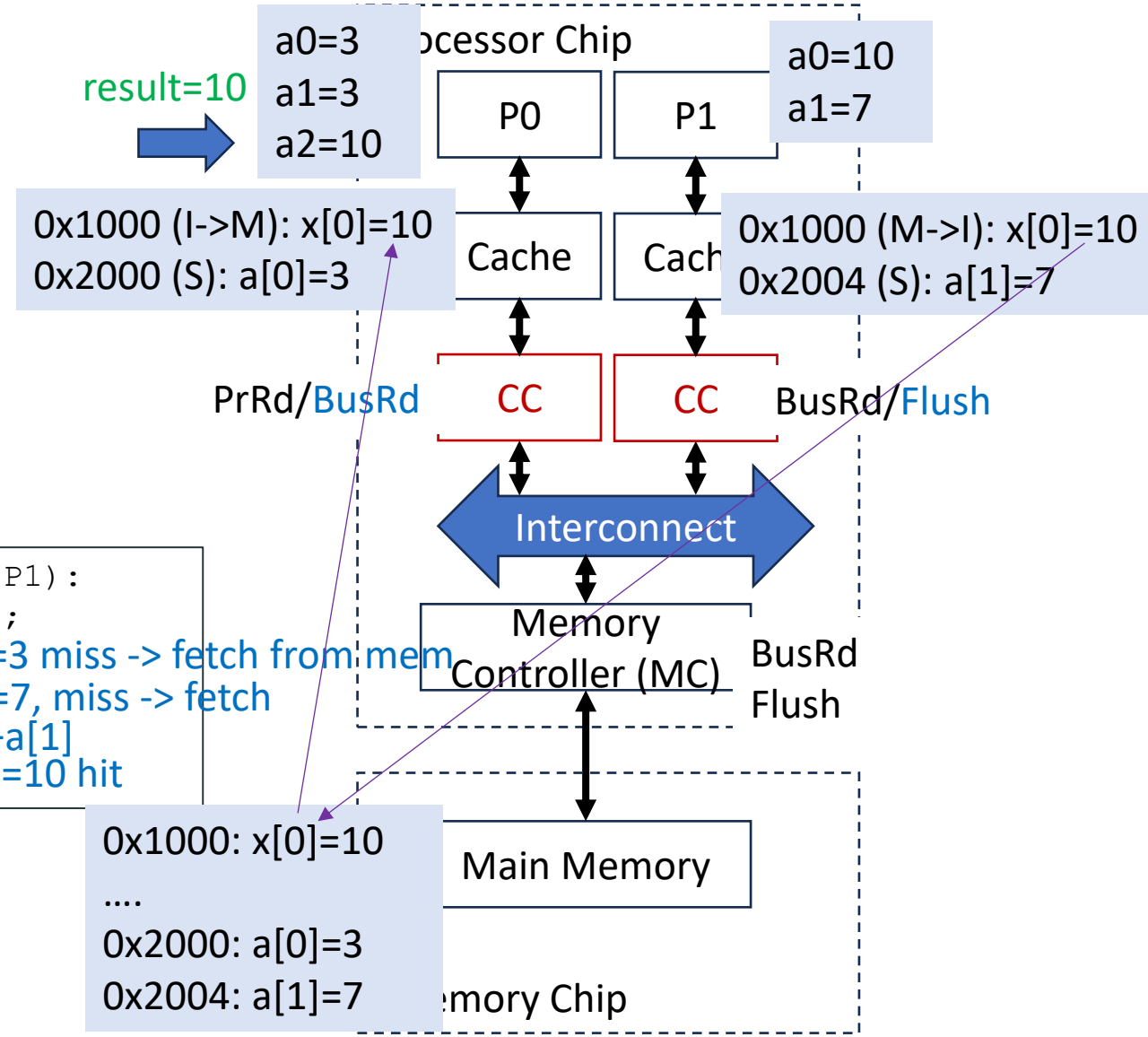
```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```

Read x[0]=10, miss

Thread 1 (P1):

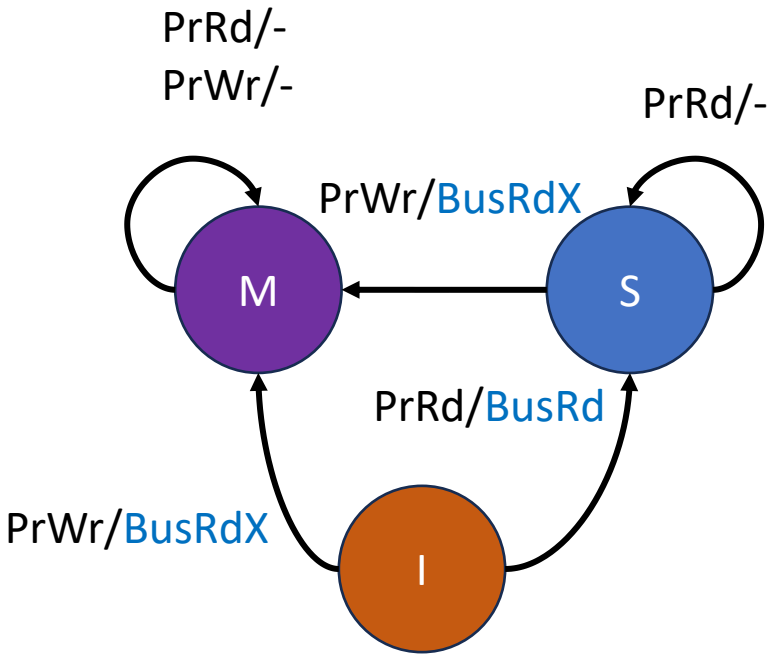
```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=3 miss -> fetch from mem
Read a[1]=7, miss -> fetch
x[0]=x[0]+a[1]
Write x[0]=10 hit



MSI Protocol with Write Back Caches – Processor Side Request

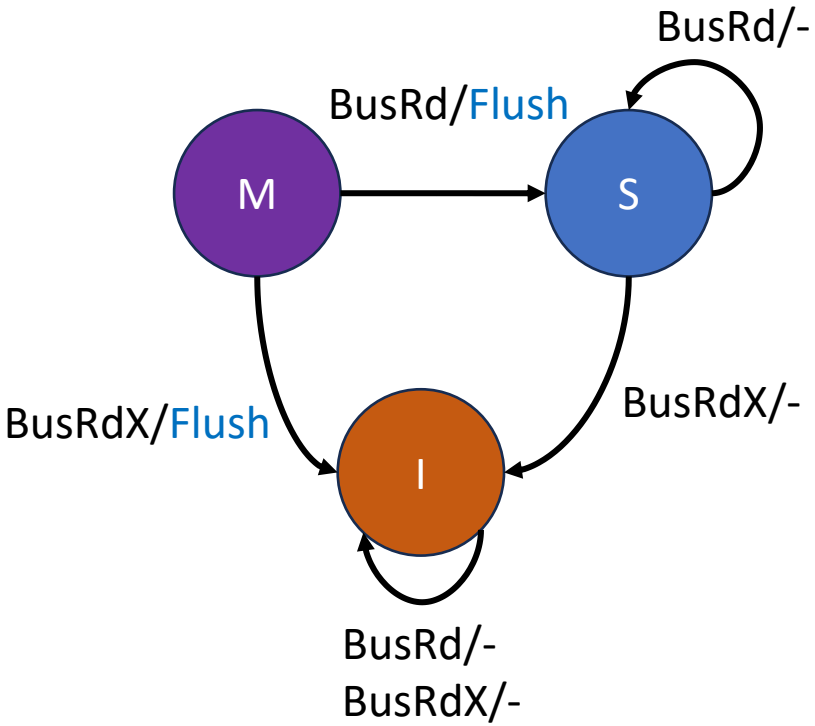
- Processor Side Request



- In invalid state (I):
  - Processor read request (PrRd):
    - Cache miss occurs
    - To load the data into the cache, a BusRd is posted on the bus
    - Fetching block from memory -> Set state to S
  - Processor write Request (PrWr):
    - posts a BusRdX request on the bus
    - Other caches will invalidate their cached copies
    - Fetching block from memory -> Set state to M
    - Processor can update the block
- In shared state (S):
  - Processor read request (PrRd):
    - Block already cached -> provide value to processor
    - No bus transaction
  - Processor write Request (PrWr):
    - Block already cached
    - posts a BusRdX request on the bus
    - Other caches will invalidate their cached copies
    - Processor can update the block in its own cache
- In modified state (M):
  - Processor read request (PrRd) & Processor write Request (PrWr)
    - No change in state

MSI Protocol with Write Back Caches – Bus Side Request

- Bus Side Request



- In invalid state (I):
  - Bus read request (BusRd, BusRedX):
    - No change in state as block can be ignored (not cached or invalid)
- In shared state (S):
  - Bus read request (BusRd):
    - Another cache is fetching the block for read
    - No state change
  - Exclusive bus read request (BusRdX):
    - Another processor is fetching the block for write
    - Invalide our copy
- In modified state (M):
  - Bus read request (BusRd): - **Intervention**
    - Another cache is fetching the block for read and has a miss
    - Flush the block to the other cache and to the memory (clean sharing)
    - Move the shared state (our copy is still up to date)
  - Exclusive bus read request (BusRdX):
    - Another cache is fetching the block for read and has a miss
    - Flush the block to the other cache and to the memory (clean sharing)
    - Invalidate our copy

Cache Coherency MSI - Example 2

Rx or Wx, where R stands for a read request, W stands for a write request, and x stands for the ID of the processor making the request

	Request	P1	P2	P3	Bus Request	Data Transfer
0	Initially	–	–	–	–	–
1	R1	S	–	–	BusRd	Mem -> P1‘s cache
2	W1	M	–	–	BusRdX	Mem -> discarded
3	R3	S	–	S	BusRd	P1’s -> Mem (flush) -> P3‘s cache
4	W3	I	–	M	BusRdX	Mem -> discarded
5	R1	S	–	S	BusRd	P3’s -> Mem (flush) -> P1‘s cache
6	R3	S	–	S	–	–
7	R2	S	S	S	BusRd	Mem -> P2‘s cache

MSI Protocol with Write Back Caches – Drawback

- Drawback with the MSI protocol:
  - For each read-then-write sequence two bus transactions are involved:
    - a BusRd to fetch the block into the shared state,
    - and a BusRdX to invalidate other cached copies.
  - Example:
    - P1 has a copy due to read request 1 (R1), The BusRdX is useless for request 2 (W1) since cache of P1 does not need a copy from memory as no other cache has this block
    - The memory controller will still supply the value even though the cache does not need it (discards!) because it does not know that cache of P1 already has a copy and only wants to upgrade from S to M).
  - **Unnecessary BW to memory!**

	Request	P1	P2	P3	Bus Request	Data Transfer
0	Initially	–	–	–	–	–
1	R1	S	–	–	BusRd	Mem -> P1‘s cache
2	W1	M	–	–	BusRdX	Mem -> discarded
3	R3	S	–	S	BusRd	P1’s -> Mem (flush) -> P3‘s cache
4	W3	I	–	M	BusRdX	Mem -> discarded

Cache Coherency MSI - Example

- Multi-threaded execution (MSI):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

Write x[0]=0, miss, fetch from mem & write

```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=0, hit
Read a[0]=3, miss -> fetch

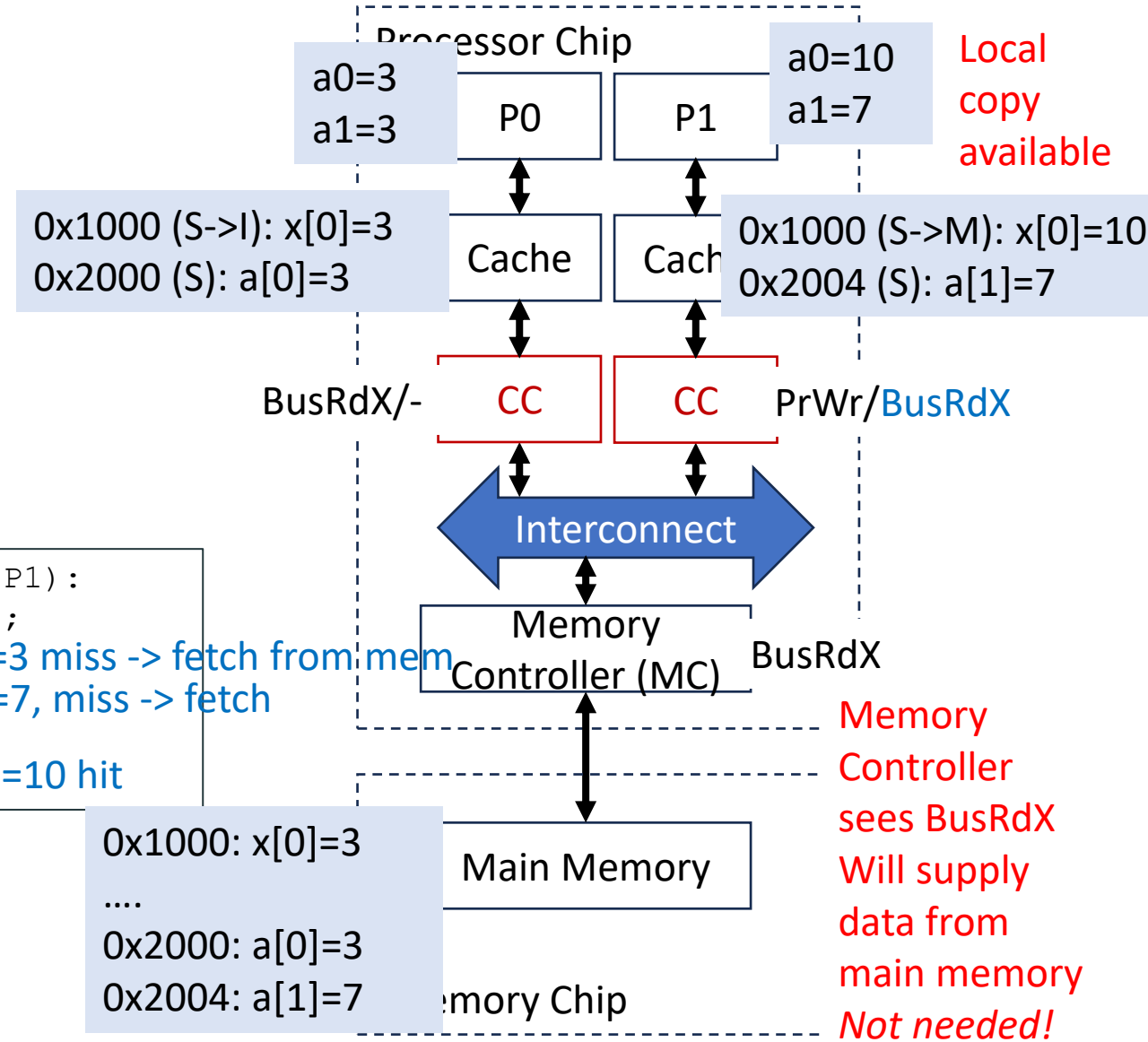
Write x[0]=3, hit

```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```

Thread 1 (P1):

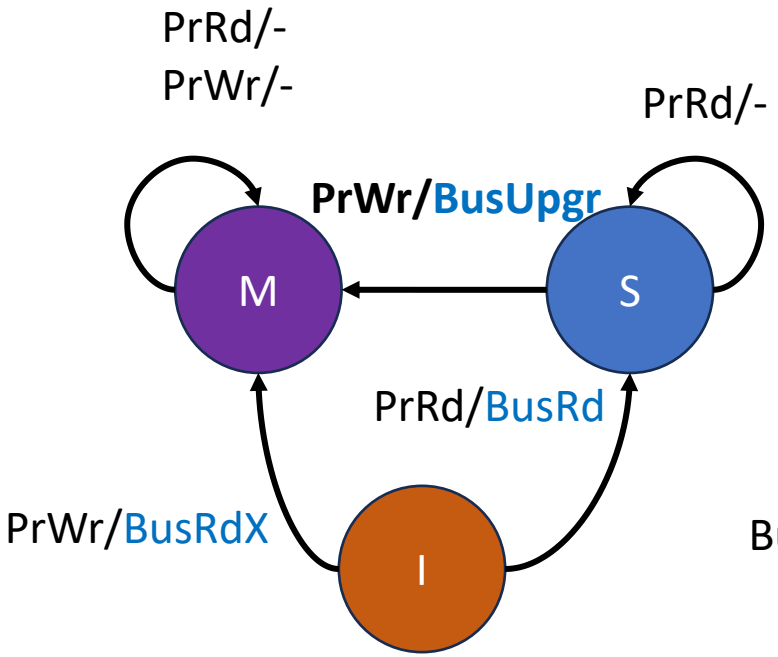
```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=3 miss -> fetch from mem
Read a[1]=7, miss -> fetch
Write x[0]=10 hit

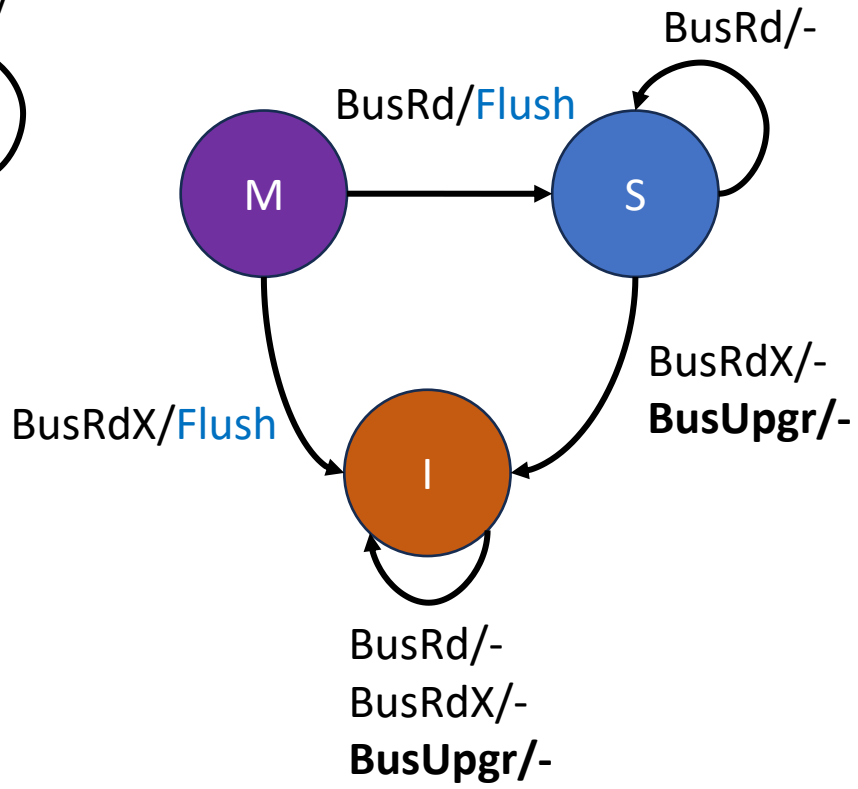


MSI Protocol with Write Back Caches and BusUpgr - Snooper FSM

- Processor Side Request



- Bus Side Request



MSI Protocol with Write Back Caches and BusUpgr

- New bus request called a bus upgrade (*BusUpgr*).
  - If a cache already has a valid copy of the block and only needs to upgrade its permission from S to M, it posts a BusUpgr instead of BusRdX.
  - On the other hand, if it does not have the block in the cache and needs the memory or another cache to supply it, it posts a BusRdX.
  - The memory controller responds differently in these two cases: Ignores the BusUpgr, but fetches the block when it snoops a BusRdX.

	Request	P1	P2	P3	Bus Request	Data Supplier
0	Initially	–	–	–	–	–
1	R1	S	–	–	BusRd	Mem -> P1‘s cache
2	W1	M	–	–	BusUpgr	-
3	R3	S	–	S	BusRd	P1’s -> Mem (flush) -> P3‘s cache
4	W3	I	–	M	BusUpgr	-

Cache Coherency MSI with BusUgr - Example

- Multi-threaded execution (MSI with BusUgr):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

Write x[0]=0, miss, fetch from mem & write

```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=0, hit
Read a[0]=3, miss -> fetch

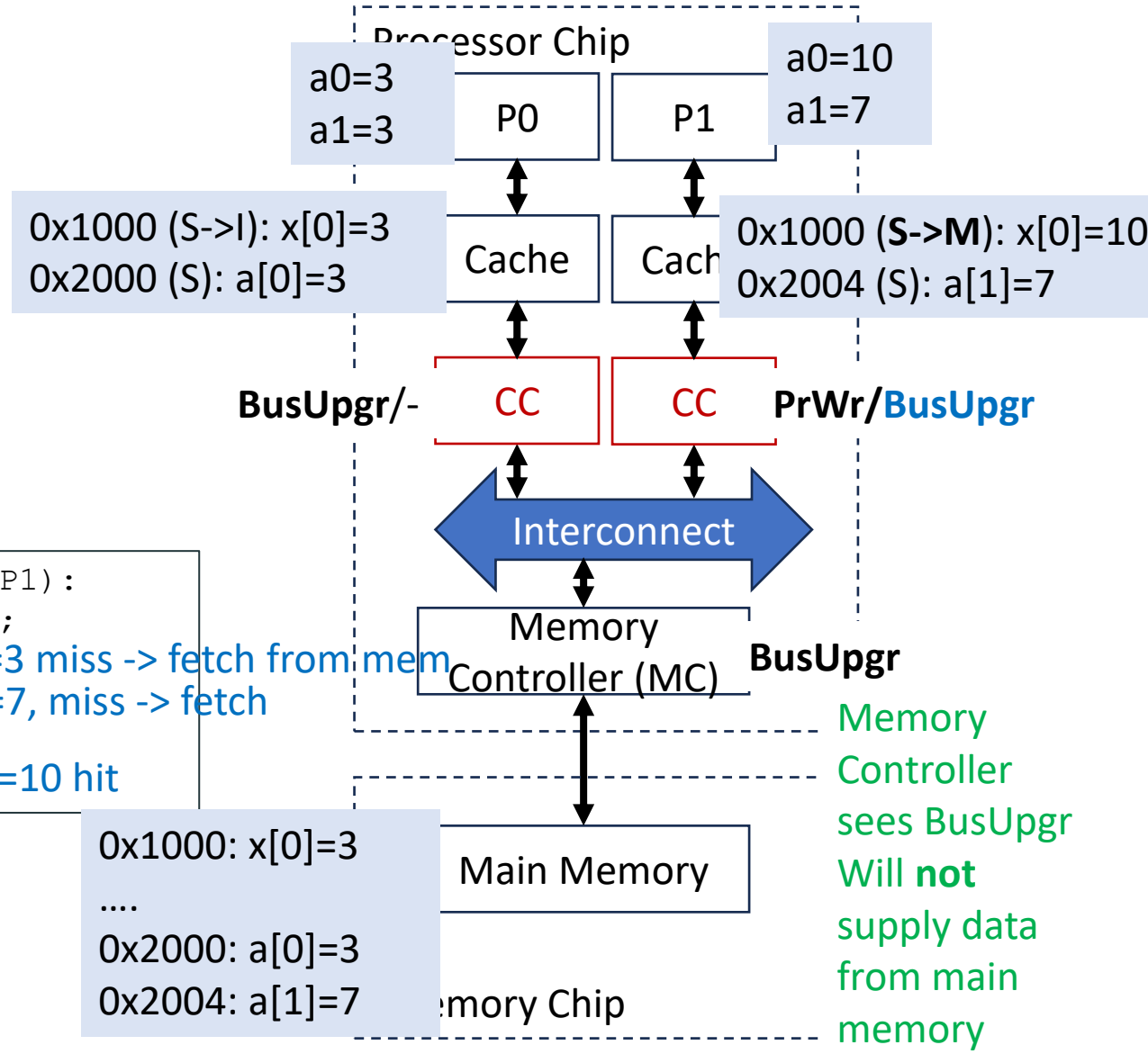
Write x[0]=3, hit

```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```

Thread 1 (P1):

```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=3 miss -> fetch from mem
Read a[1]=7, miss -> fetch
Write x[0]=10 hit



E2.4 MESI Protocol with Write Back Caches

MESI Protocol with Write Back Caches - Requests

In the MESI protocol, processor requests to the cache include:

1. **PrRd**: processor-side request to read from a cache block.
2. **PrWr**: processor-side request to write to a cache block.

Bus-side requests include:

1. **BusRd**: snooped request that indicates there is a read request to a cache block made by another processor.
2. **BusRdX**: snooped request that indicates there is a *read exclusive* (write) request to a cache block made by another processor which does not already have the block.
3. **BusUpgr**: snooped request that indicates that there is a write request to a cache block that another processor already has in its cache.
4. **Flush**: snooped request that indicates that an entire cache block is written back to the main memory by another processor.
5. **FlushOpt**: snooped request that indicates that an entire cache block is posted on the bus in order to supply it to another processor. We refer to such an optional block flush as *cache-to-cache transfer*.

MESI Protocol with Write Back Caches - Cache Block States

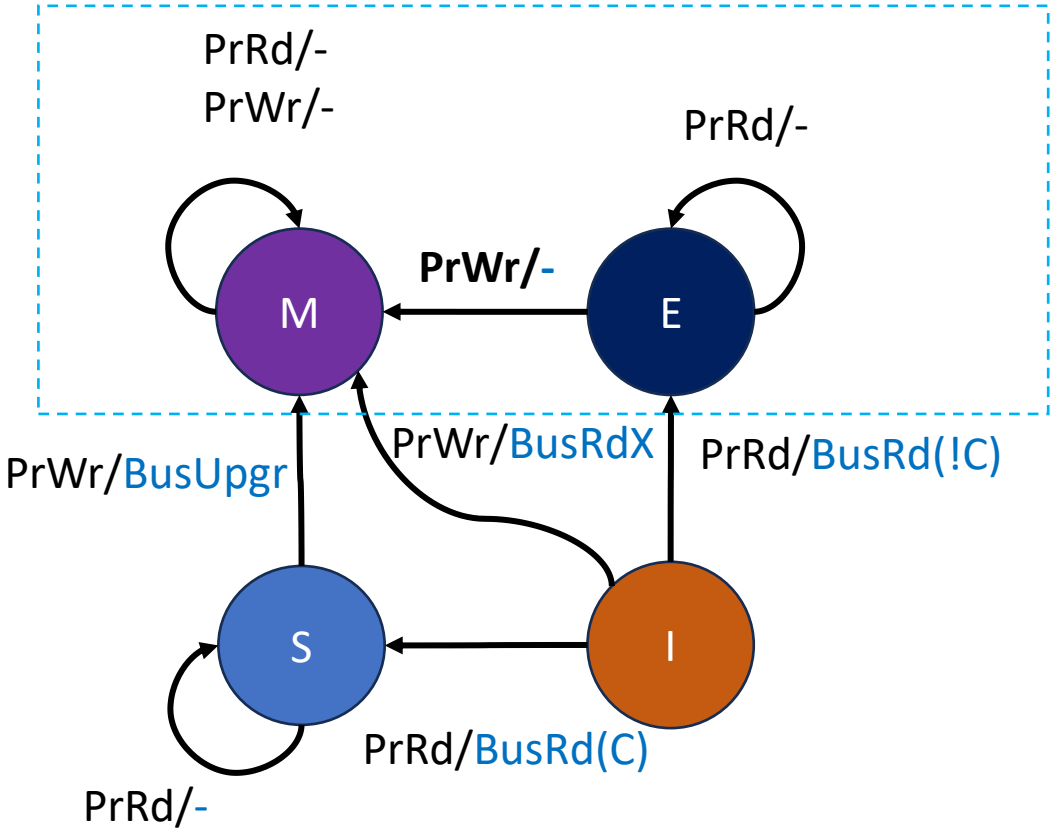
Each cache block has an associated state which can have one of the following values:

1. **Modified** (M): the cache block is valid in only one cache, and the value is (likely) different than the one in the main memory. This state has the same meaning as the dirty state in a write back cache for a single processor system.
2. **Exclusive** (E): the cache block is valid, clean, and *only resides in one cache*.
3. **Shared** (S): the cache block is valid, clean, but may reside in multiple caches.
4. **Invalid** (I): the cache block is invalid.

New signal C at Snooper (is high if any processor has a copy of the cache block)

MESI Protocol with Write Back Caches - Snooper FSM

- Processor Side Request

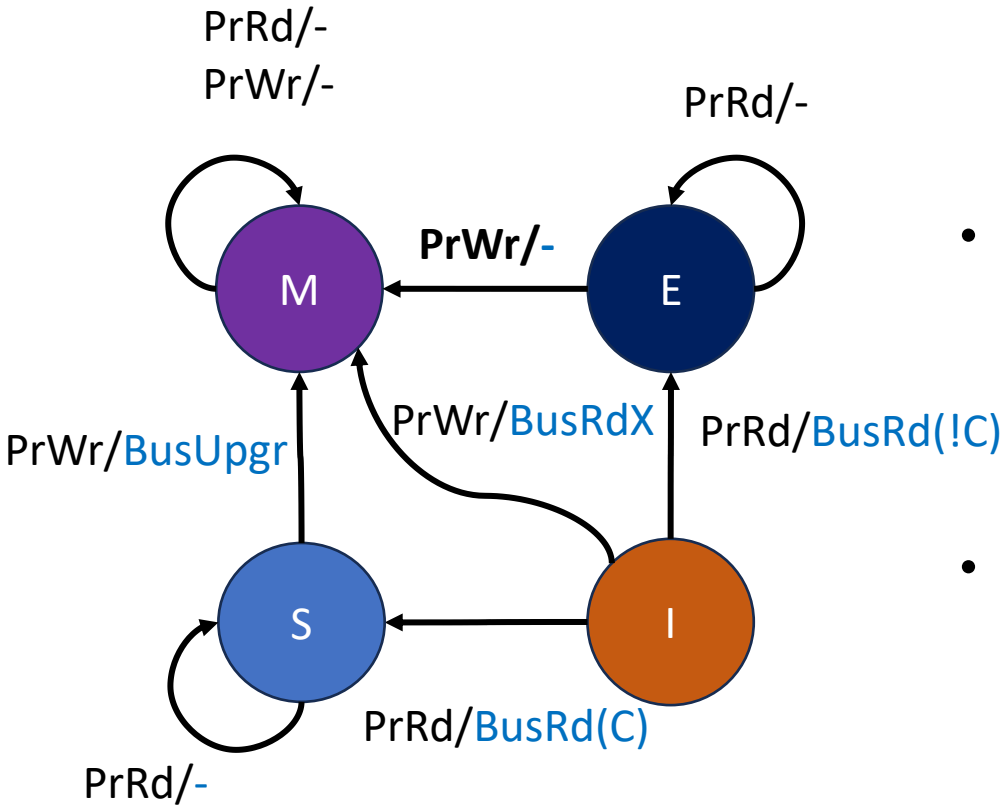


MESI: Keeps track if processor has the data exclusively:

- Often threads operate on private data that would either be in exlusive (E) or modified (M) state.
- For this private data no bus signaling is required (see blue box)
- Bus signalling always incurs performance overheads as other CCs and memory controller need to react.

MESI Protocol with Write Back Caches - Snooper FSM

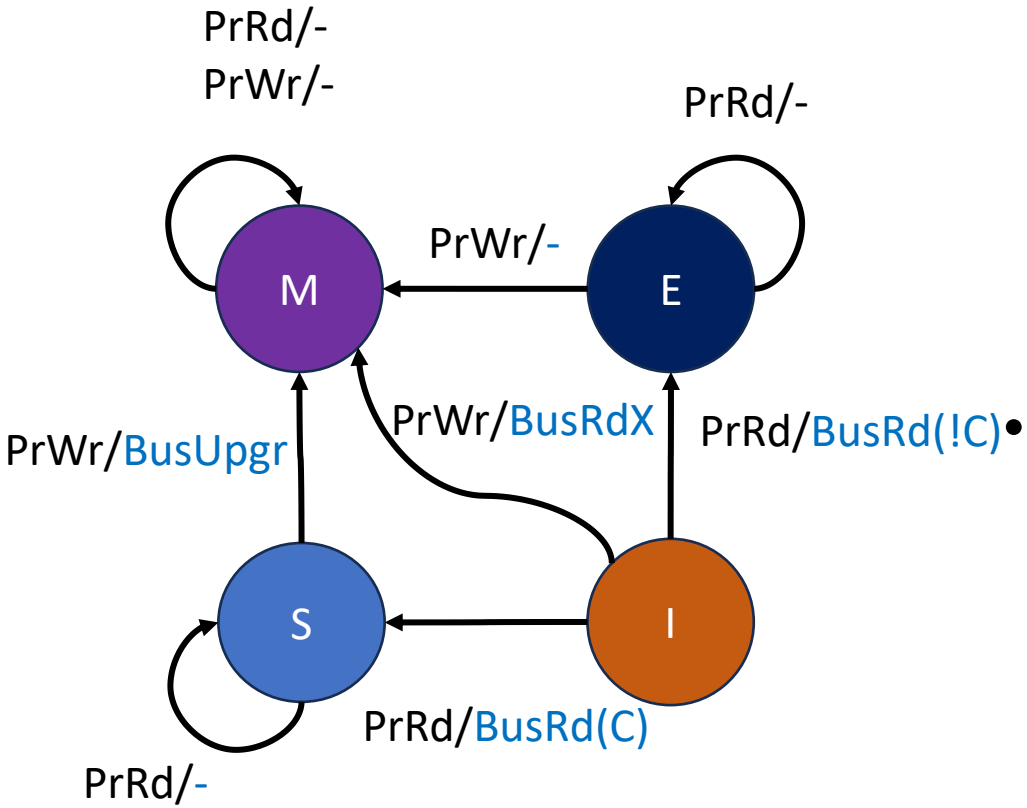
- Processor Side Request



- In invalid state (I):
  - Processor read request, other processor has cache block (PrRd(C)):
    - Cache miss occurs
    - To load the data into the cache, a BusRd is posted on the bus
    - Other processors indicate with C that they have a copy in cache
    - Fetching block from other cache (FlushOpt) -> Set state to S
  - Processor read request, no other processor has cache block (PrRd(!C)):
    - Cache miss occurs
    - To load the data into the cache, a BusRd is posted on the bus
    - Other processors indicate with C that they do not have a copy in cache
    - Fetching block from memory -> Set state to E
  - Processor write Request (PrWr):
    - posts a BusRdX request on the bus
    - Other caches will invalidate their cached copies, possibly flush to mem
    - Fetching block from memory -> Set state to M
    - Processor can update the block

MESI Protocol with Write Back Caches - Snooper FSM

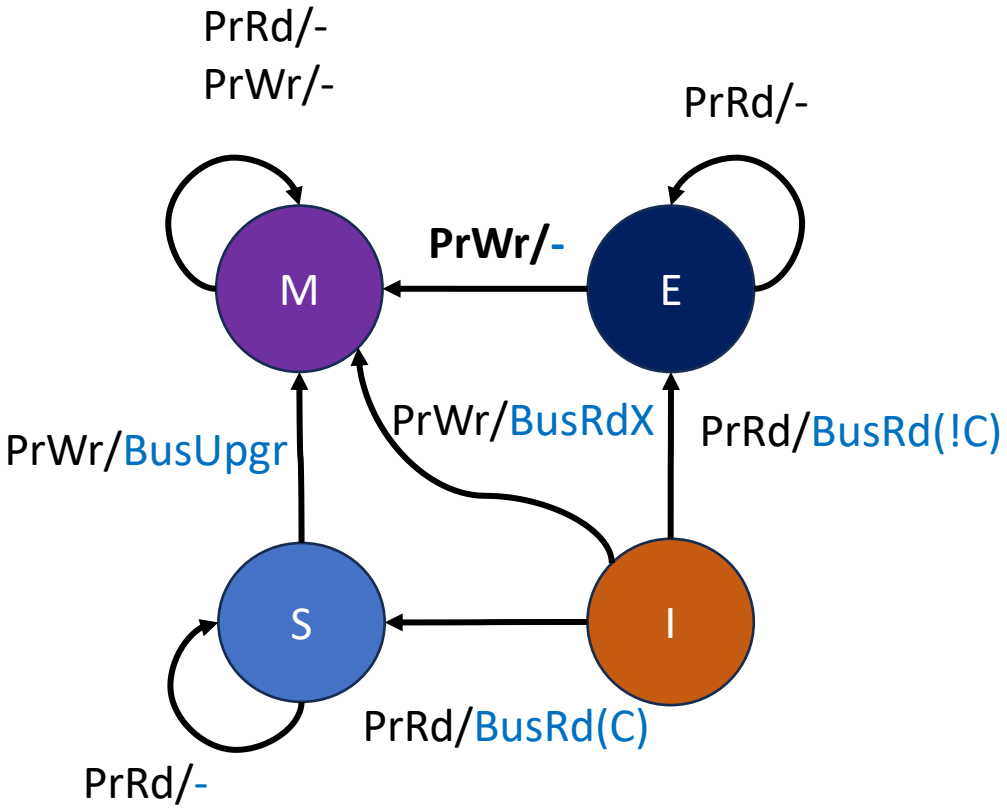
- Processor Side Request



- In shared state (S):
  - processor read request (PrRd):
    - Block already cached -> provide value to processor
    - No bus transaction
  - Processor write Request (PrWr):
    - Block already cached
    - posts a BusUpgr request on the bus
    - Other caches will invalidate their cached copies
    - Processor can update the block in its own cache
- In modified state (M):
  - processor read request (PrRd) & Processor write Request (PrWr)
    - No change in state

MESI Protocol with Write Back Caches - Snooper FSM

- Processor Side Request

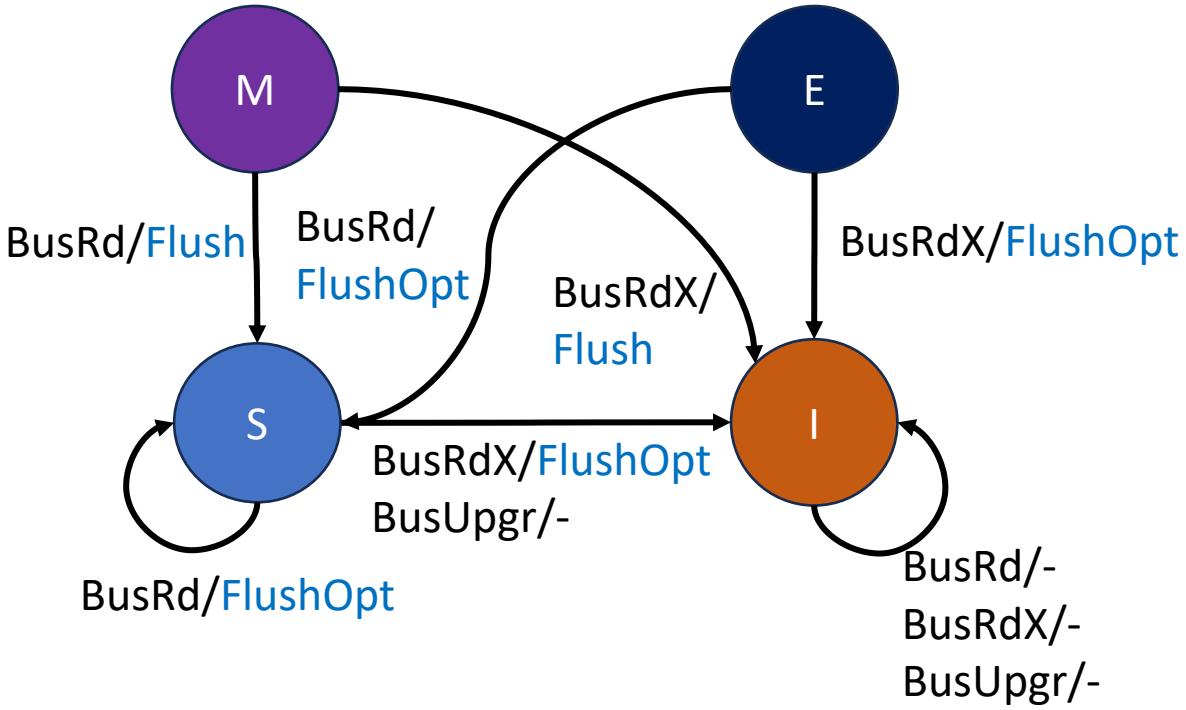


- In exclusive state (E):
  - processor read request (PrRd):
    - Block already cached -> provide value to processor
    - No bus transaction
  - **Processor write Request (PrWr):**
    - **Block already cached**
    - **No other processor has copy, no need to send bus message**
    - **Processor can update the block in its own cache**

    One major advantage of MESI!

MESI Protocol with Write Back Caches - Snooper FSM

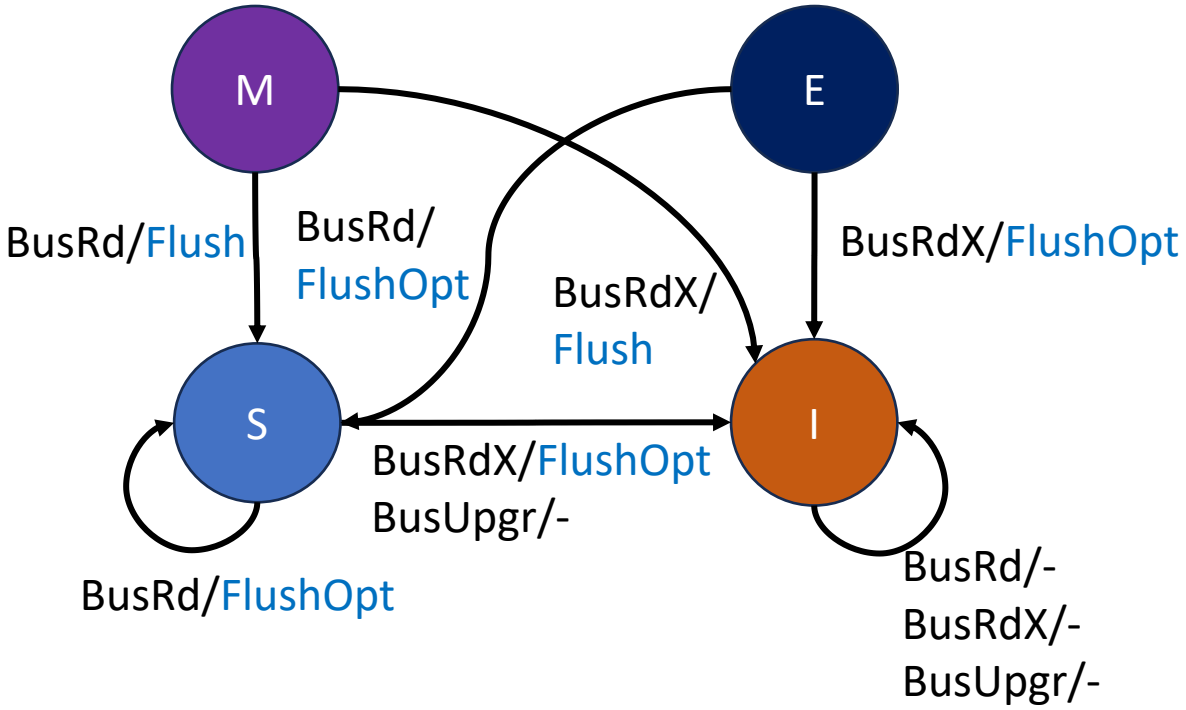
- Bus Side Request



- In invalid state (I):
  - Bus read request (BusRd, BusRdX,BusUpgr):
    - No change in state as block can be ignored (not cached or invalid)
- In shared state (S):
  - Bus read request (BusRd):
    - Another cache is fetching the block for read
    - FlushOpt to allow a cache-to-cache transfer, as value is same as in memory
    - No state change
  - Exclusive bus read request (BusRdX):
    - Another processor is fetching the block for write
    - FlushOpt to allow a cache-to-cache transfer, as value is same as in memory
    - Invalide our copy
  - Bus upgrade request (BusUpgr):
    - Another processor is fetching the block for write; but has a local copy
    - Invalide our copy

MESI Protocol with Write Back Caches - Snooper FSM

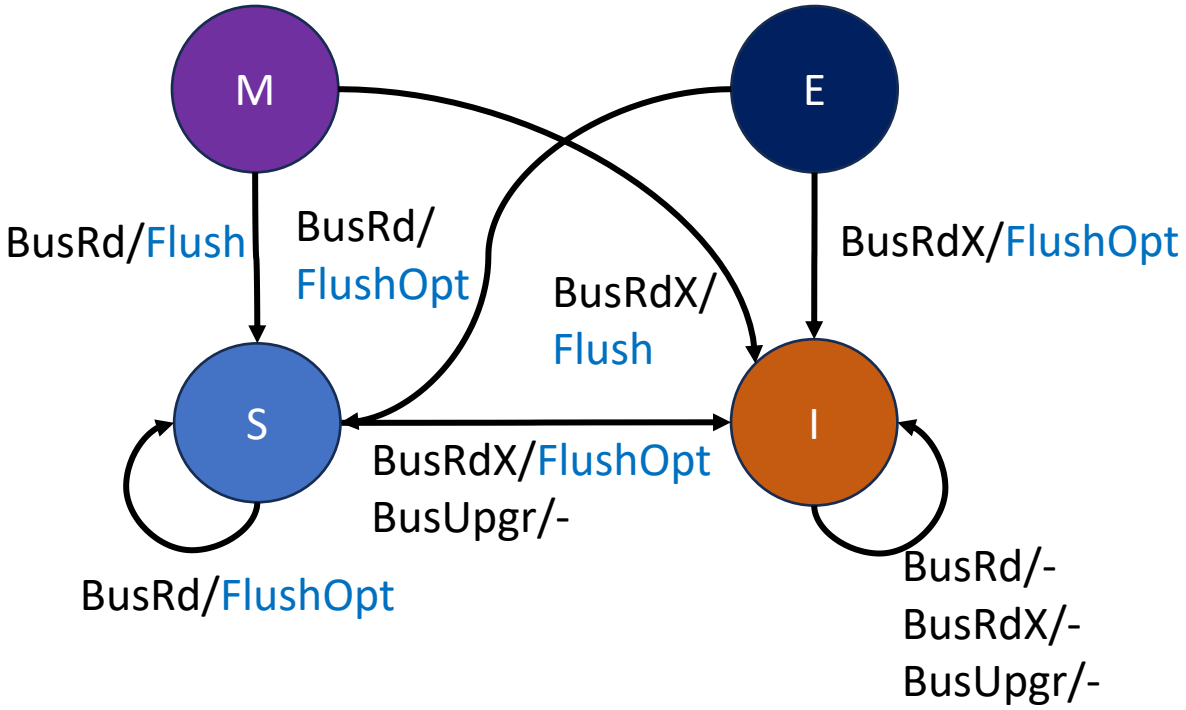
- Bus Side Request



- In modified state (M):
  - Bus read request (BusRd):
    - Another cache is fetching the block for read and has a miss
    - Flush the block to the other cache and to the memory (clean sharing)
    - Move to the shared state (our copy is still up to date)
  - Exclusive bus read request (BusRdX):
    - Another cache is fetching the block for write and has a miss
    - Flush the block to the other cache and to the memory (clean sharing)
    - Invalidate our copy

MESI Protocol with Write Back Caches - Snooper FSM

- Bus Side Request



- In exclusive state (E):
  - Bus read request (BusRd):
    - Another cache is fetching the block for read and has a miss
    - FlushOpt to allow a cache-to-cache transfer, as value is same as in memory
    - Move the shared state (our copy is still up to date)
  - Exclusive bus read request (BusRdX):
    - Another cache is fetching the block for write and has a miss
    - FlushOpt to allow a cache-to-cache transfer, as value is same as in memory
    - Invalidate our copy

Cache Coherency MESI - Example

- Multi-threaded execution (MESI):

Thread 0 (P0):

```
//Line 1 Thread 0 (P0): x[0]=0
SW zero,0(t0)
```

Write x[0]=0, miss,
fetch from mem & write

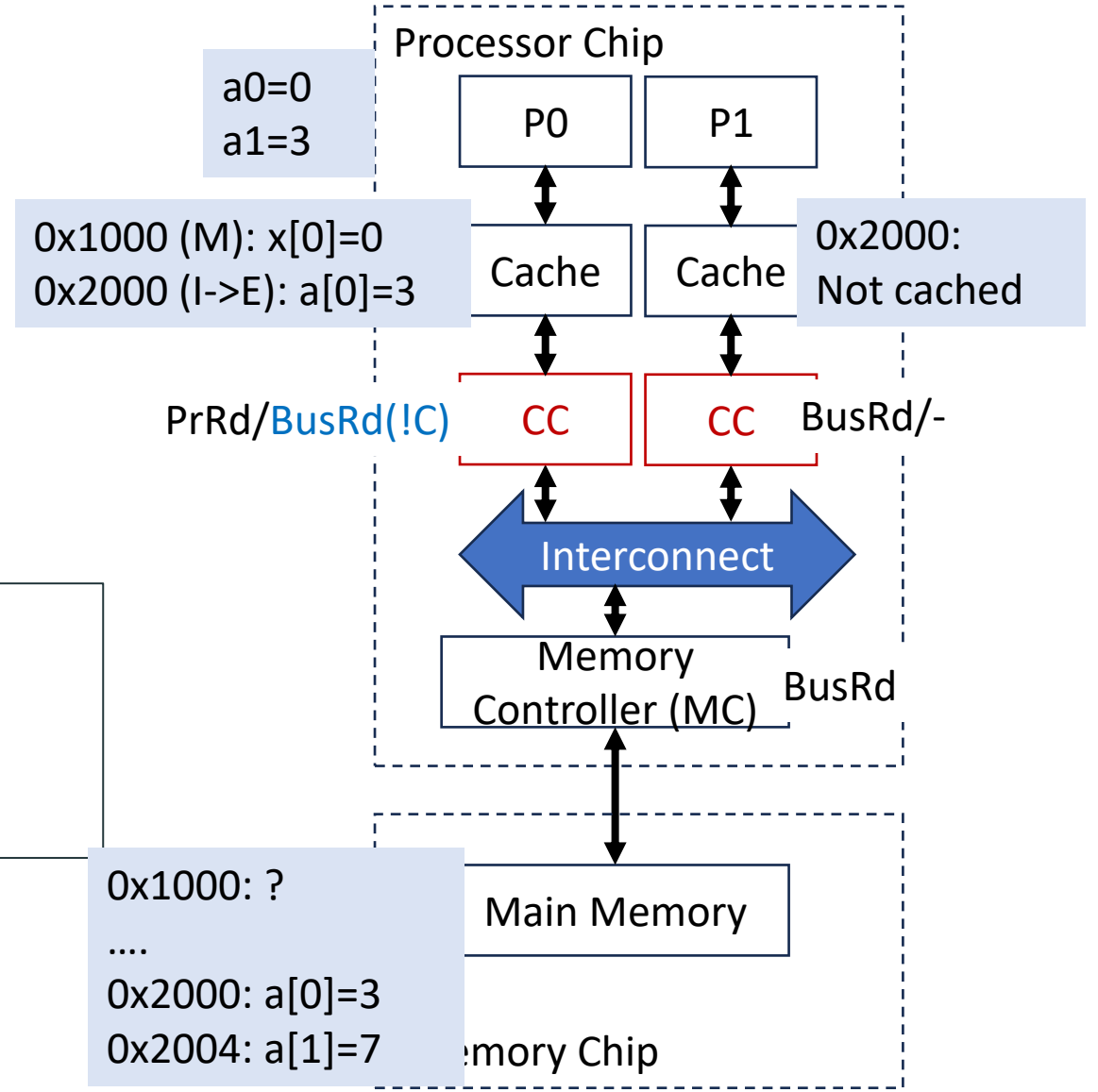
```
//Line 5: Thread 0 (P0):
// x[0] = x[0] + a[0];
LW a0,0(t0)
LW a1,0(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

Read x[0]=0, hit
Read a[0]=3, miss -> fetch

Thread 1 (P1):

```
// Line 5: Thread 1 (P1):
// x[0] = x[0] + a[1];
LW a0,0(t0)
LW a1,4(t1)
ADD a0,a0,a1
SW a0,0(t0)
```

```
// Line 8 Thread (P0):
// result=x[0]
LW a2,0(t0)
```



Example

Rx or Wx, where R stands for a read request, W stands for a write request, and x stands for the ID of the processor making the request

	Request	**P1**	**P2**	**P3**	**Bus Request**	**Data Transfers**
0	Initially	–	–	–	–	–
1	R1	**E**	–	–	BusRd	Mem -> P1's cache
2	W1	M	–	–	–	–
3	R3	S	–	S	BusRd	P1's cache -> Mem (flush) -> P3's cache
4	W3	I	–	M	BusUpgr	–
5	R1	S	–	S	BusRd	P3's cache -> Mem (flush) -> P1's cache
6	R3	S	–	S	–	–
7	R2	S	S	S	BusRd	**P1/P3's cache ->** P2's cache (flushOpt)[+]

[+]Clean Sharing: the block in memory is the same as in all three caches (request 5 wrote it to the memory)

Comparison MSI vs. MESI

- Compared to the MSI protocol, the MESI protocol does not reduce the bandwidth usage on the bus, but it does reduce the bandwidth use to the main memory due to the cache-to-cache transfers (FlushOpt).

➢ Bandwidth to the main memory is often a bottleneck when there is a lot of processors connected to the same memory (*known as the Memory wall!*).

- Additionally, MESI keeps track of data that is exclusive to the thread (threads often operate on private data, not all data is shared). No bus signaling required for this private data.

E2.5 MOESI Protocol with Write Back Caches

MOESI Protocol with Write Back Caches - Requests

In the MOESI protocol, processor requests to the cache include:

1. **PrRd**: processor-side request to read to a cache block.
2. **PrWr**: processor-side request to write to a cache block.

Bus-side requests include:

1. **BusRd**: snooped request that indicates there is a read request to a cache block made by another processor.
2. **BusRdX**: snooped request that indicates there is a *read exclusive* (write) request to a cache block made by another processor which does not already have the block.
3. **BusUpgr**: snooped request that indicates that there is a write request to a cache block that another processor already has in its cache.
4. **Flush**: snooped request that indicates that an entire cache block is placed on the bus by a processor to facilitate a transfer to another processor's cache. (*Different from MESI!, not to memory, closer to FlushOpt in MESI!)*
5. **FlushOpt:** snooped request that indicates that an entire cache block is posted on the bus in order to supply it to another processor. (*We refer to it as FlushOpt because unlike Flush which is needed for write propagation correctness, FlushOpt is implemented as a performance enhancing feature that can be removed without impacting correctness*.)
6. **FlushWB**: snooped request that indicates that an entire cache block is written back to the main memory by another processor, and it is not meant as a transfer from one cache to another.

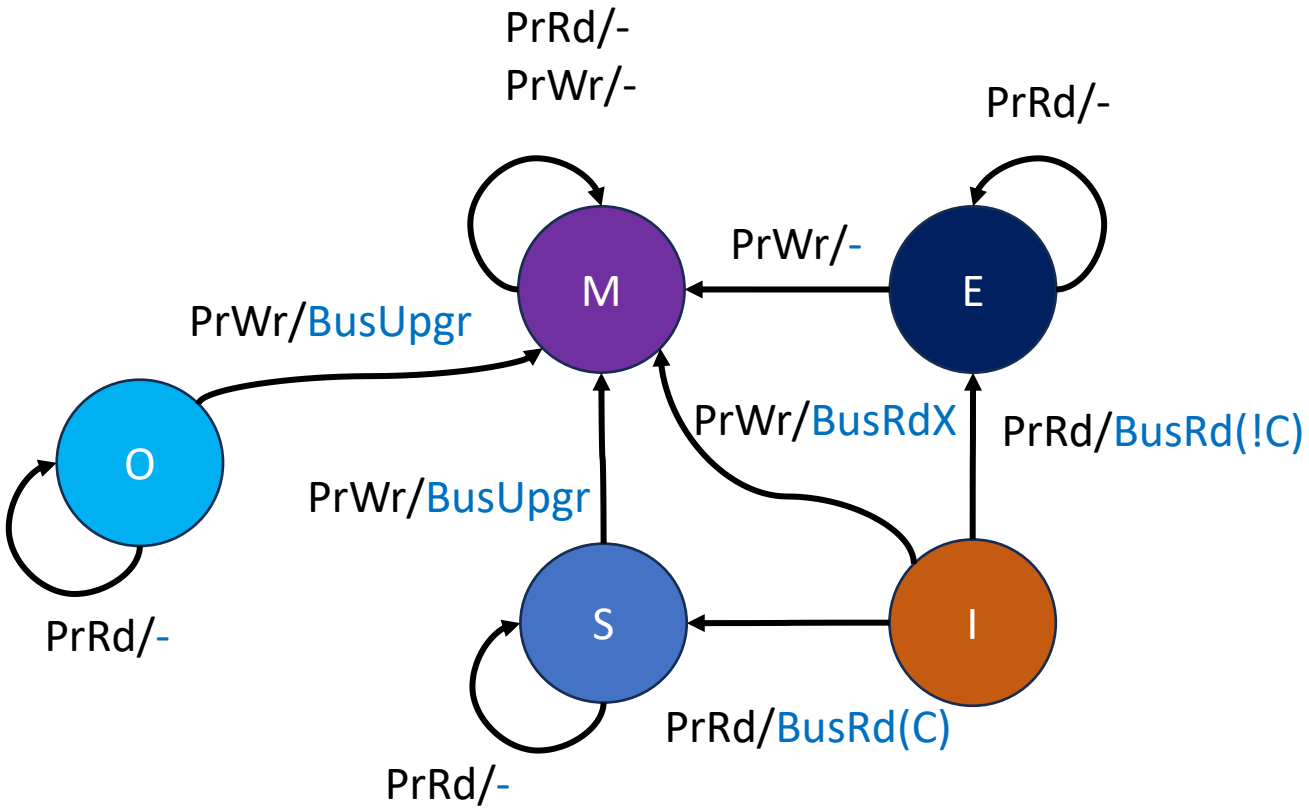
MOESI Protocol with Write Back Caches - Cache Block States

Each cache block has an associated state which can have one of the following values:

1. **Modified** (M): the cache block is valid in only one cache, and the value is (likely) different than the one in the main memory. This state has the same meaning as the dirty state in a write back cache for a single processor system, except that now it also implies exclusive ownership.
2. **Owned** (O): the cache block is valid, possibly dirty, and may reside in multiple caches. However, when there are multiple cached copies, there can only be one cache that has the block in owned state, other caches should have the block in state shared.
3. **Exclusive** (E): the cache block is valid, clean, and *only resides in one cache*.
4. **Shared** (S): the cache block is valid, *possibly dirty*, but may reside in multiple caches.
5. **Invalid** (I): the cache block is invalid.

MOESI Protocol with Write Back Caches - Snooper FSM

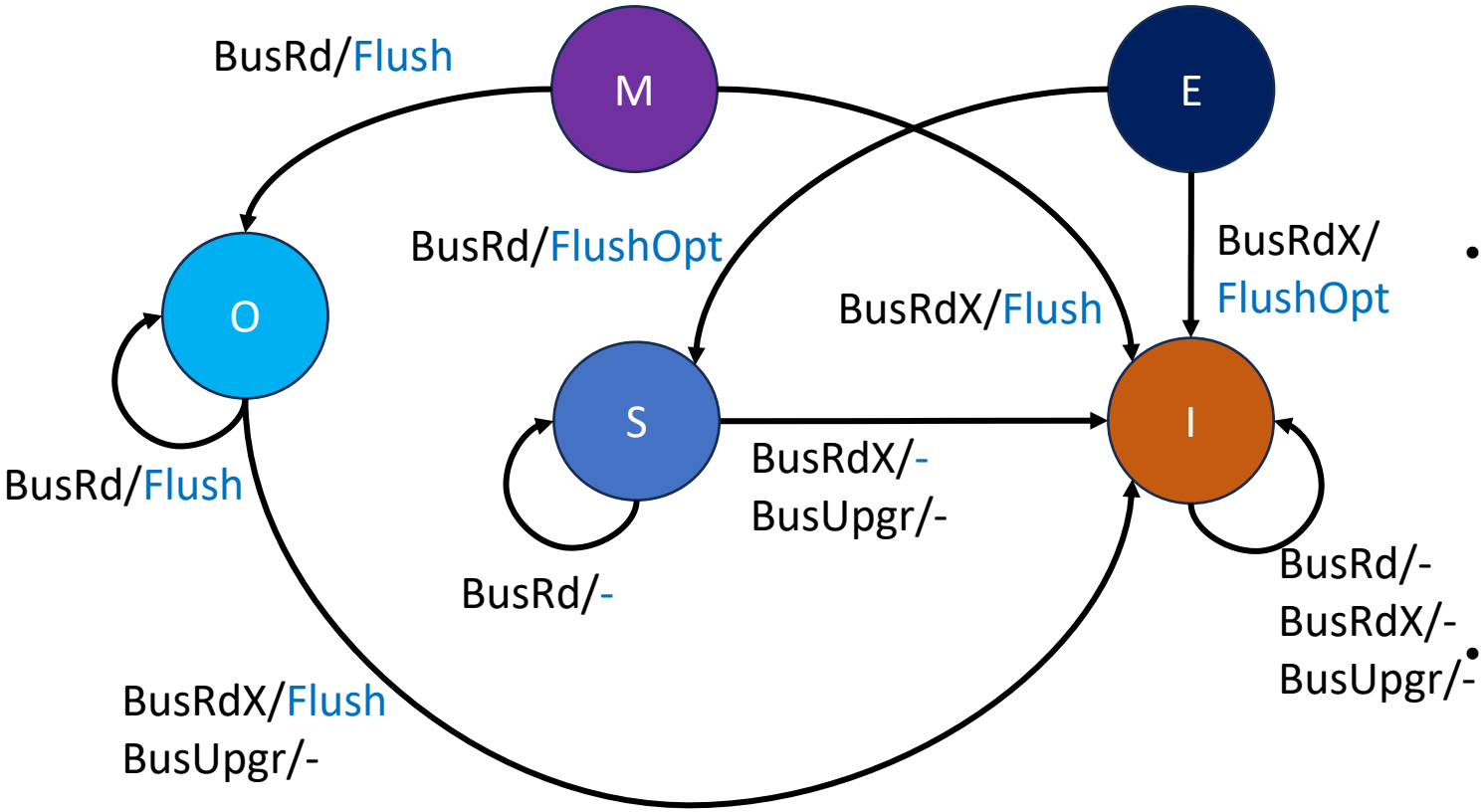
- Processor Side Request



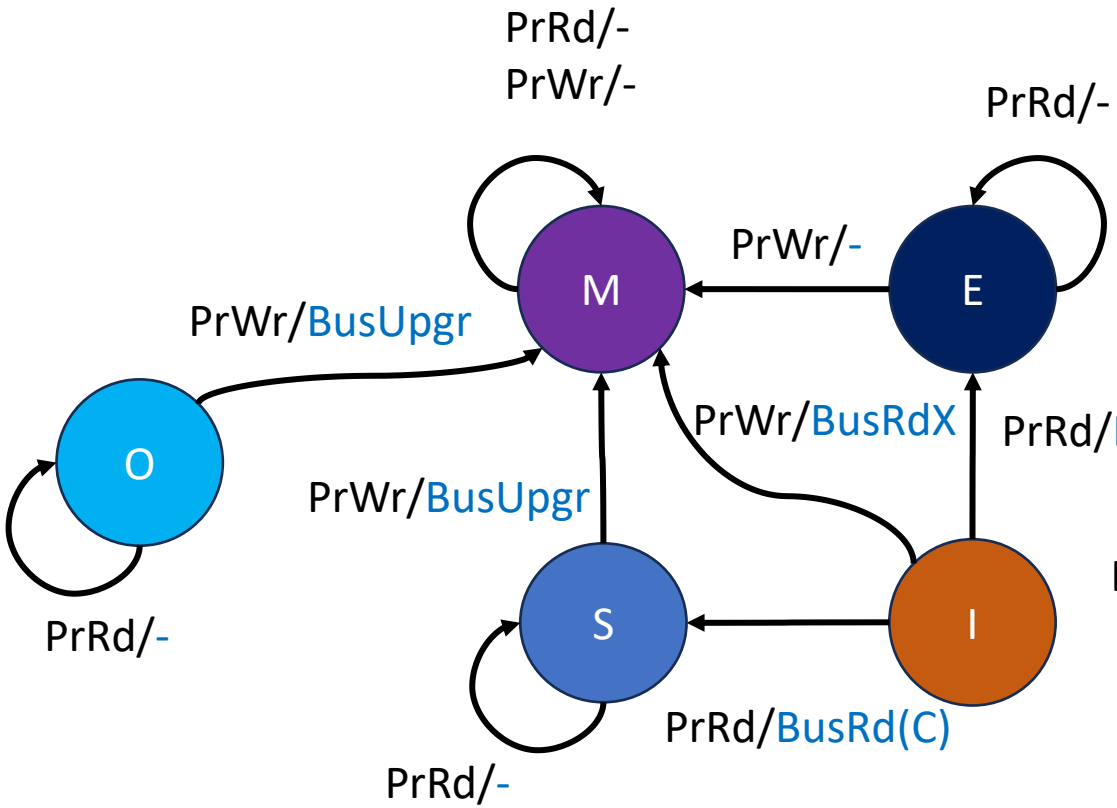
- In Owned State(O):
  - Processor read request (PrRd):
    - Block already cached -> provide value to processor
  - Processor write Request (PrWr):
    - Block already cached
    - posts a BusUpgr request on the bus
    - Other caches will invalidate their cached copies
    - Processor can update the block in its own cache

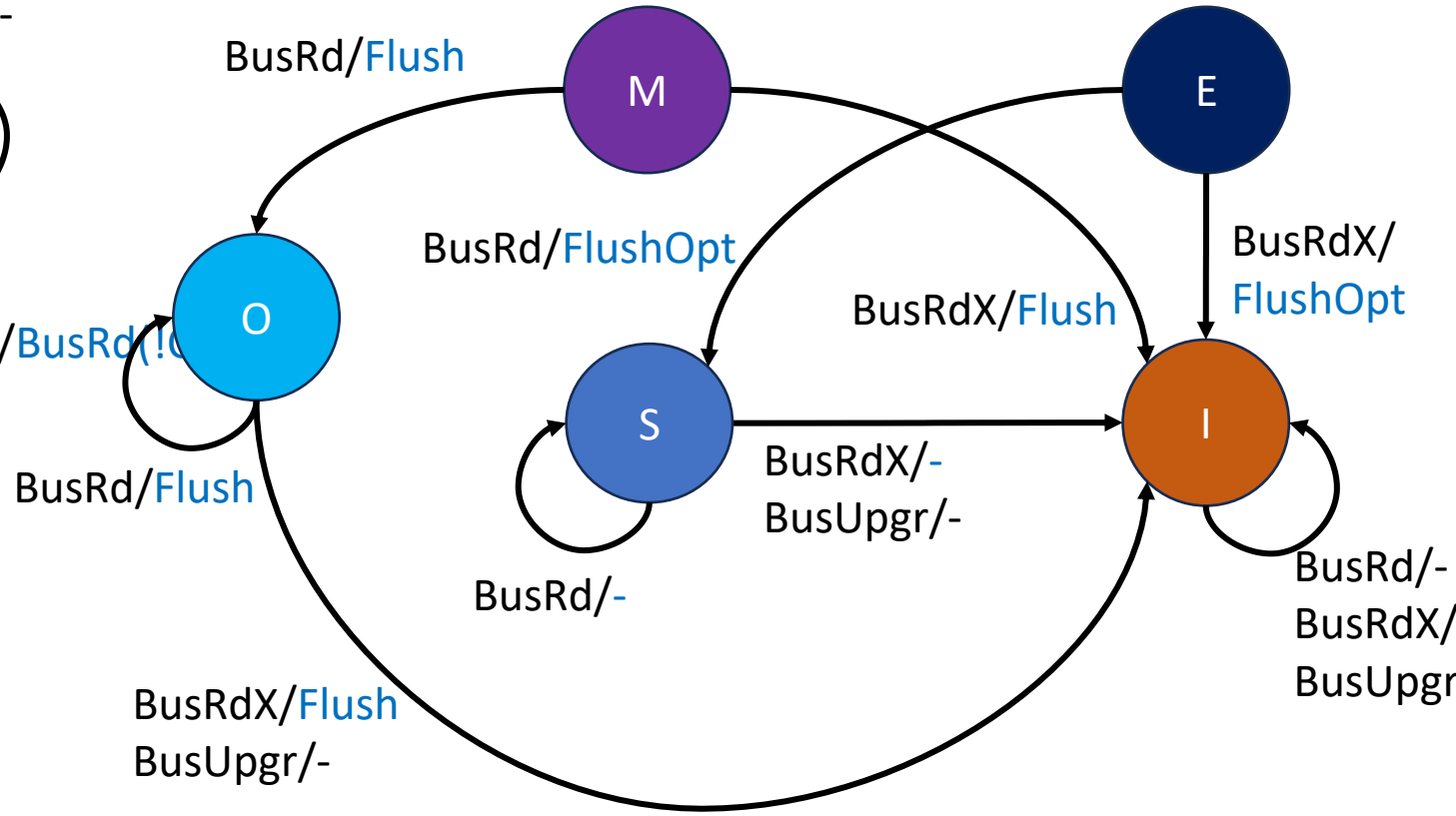
MOESI Protocol with Write Back Caches - Snooper FSM

- Bus Side Request



- In Owned State(O):
  - Bus read request (BusRd):
    - Block is owned by own cache and possibly dirty
    - Other cache fetches block for read
    - Flush block to other processor (cache to cache transfer, possibly **dirty sharing**)
    - Own cache remains to be owner
  - Exclusive bus write request (BusRdX):
    - Block is owned by own cache and possibly dirty
    - Other cache fetches block for write
    - Flush block to other processor (cache to cache transfer, possibly **dirty sharing**)
    - Invalidate own cache block, loose ownership
  - Bus Upgrade Request (BusUpgr):
    - Other cache fetches block for write and has own up-to-date copy
    - Invalidate own cache block, loose ownership


PrRd/-
PrWr/-
PrRd/-
M
PrWr/-
E
PrWr/BusUpgr
O
PrWr/BusRdX
PrRd/BusRd(!C
PrWr/BusUpgr
S
I
PrRd/-
PrRd/BusRd(C)
PrRd/-


BusRd/Flush
M
E
BusRd/FlushOpt
BusRdX/
FlushOpt
BusRdX/Flush
O
S
I
BusRd/Flush
BusRdX/-
BusUpgr/-
BusRd/-
BusRd/-
BusRdX/
BusUpgr
BusRdX/Flush
BusUpgr/-

MOESI - Example

- The owner cache supplies the block
- As the owner could have modified the block, it may differ from the block in memory during transfer (dirty sharing)
- **The memory controller (MC) only writes the block to memory during a flushWB, the flushWB is coming from the owner (see next slide)**.

	Request	P1	P2	P3	Bus Request	Data Transfer
0	Initially	–	–	–	–	–
1	R1	E	–	–	BusRd	Mem -> P1's cache
2	W1	M	–	–	–	–
3	R3	**O**	–	S	BusRd	P1's cache -> P3's cache (flush)*
4	W3	I	–	M	BusUpgr	–
5	R1	S	–	**O**	BusRd	P3's cache -> P1's cache (flush)*
6	R3	S	–	**O**	–	–
7	R2	S	S	**O**	BusRd	**P3's cache** -> P2's cache (flush)* +

- * Dirty sharing: the copy of the block in the main memory has not seen the updates in request 2 and 4
- + MC: Writing the block also to memory as P3 was the owner (O) of block when it was flushed.

MOESI – FlushWB

- The owner (O) keeps track of the latest version on each block and supplies it.
- Dirty sharing: The memory may not have up-to-date copy.

- flushWB's role:
  - If the owner evicts the cache block, then it needs to be written back to the main memory (**this is the FlushWB**), it is not in the FSM as it is not caused by a read/write of this cache block, but by another block causing the evict.
  - There is no owner after that but other caches may still have block in shared state (transfer of owner can be implemented)

MOESI – FlushOpt

- FlushOpt‘s role:
  - FlushOpt is happening when downgrading from Exclusive (E) to Shared (S) or invalid (I)
  - As a key characteristic, MOESI fetches blocks from the owner
  - If the block is in E state, it is not marked as „owned“
  - Yet, as optimization feature flushOpt indicates that the block is supplied by the cache having it in „E“ state and not by the memory in a clean sharing cache-to-cache transfer
  - This is not needed for correctness (write propagation) as the block could also be supplied by the memory (clean sharing, memory has a valid copy)

MOESI vs. MESI

- MOESI allows for dirty sharing:
  - Less memory traffic, Faster transfers (cache to cache)
  - But with L2 cache, the effect may be less important, as L2 to L1 may still be fast.
- MOESI needs 3 bit per cache line to store state, MESI only 2 bit

- MESI, MOESI:
  - Open question: When several blocks have clean cache block in shared state – Who supplies block?

E2.6 Further Protocols

Further protocols

- MESIF (by Intel): MESI with a forwarding state (used as designated supplier when several caches share a clean block), but no dirty sharing such as MOESI

- MSI, MESI; MOESI: Invalidation-based protocols
- Alternative are update-based protocols, e.g. Dragon protocol (see book)

Summary

Where we are

- Multi-cores with shared memory
- Cache coherency protocols

- Next: Memory consistency & Synchronization mechanisms