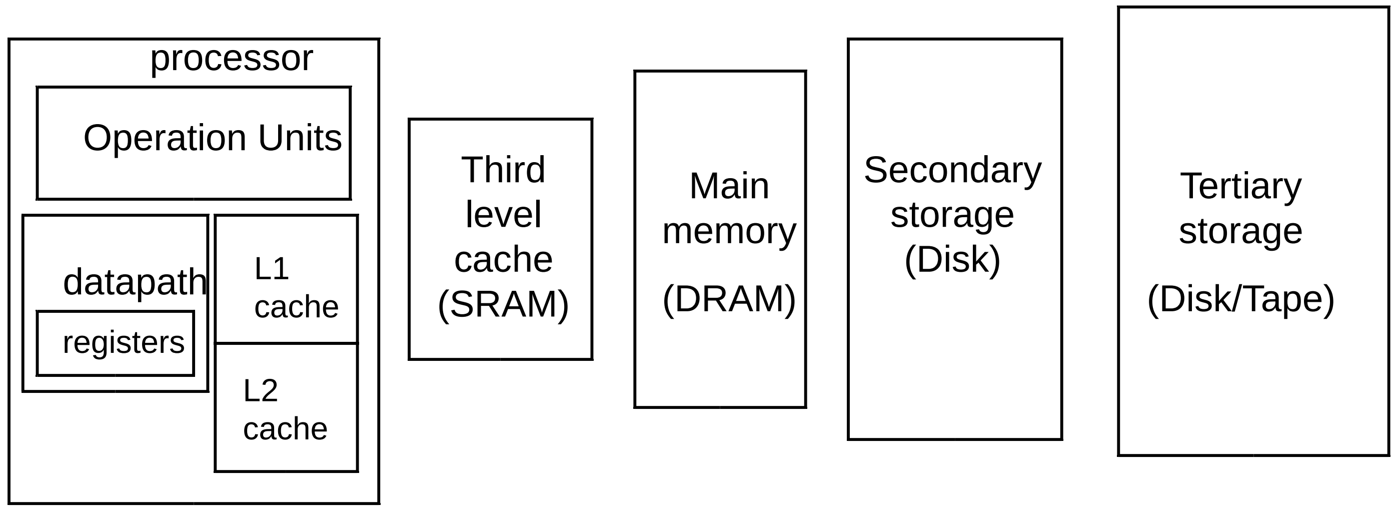
departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Concurrency and Parallelism
(Concorrência e Paralelismo – CP 11158)

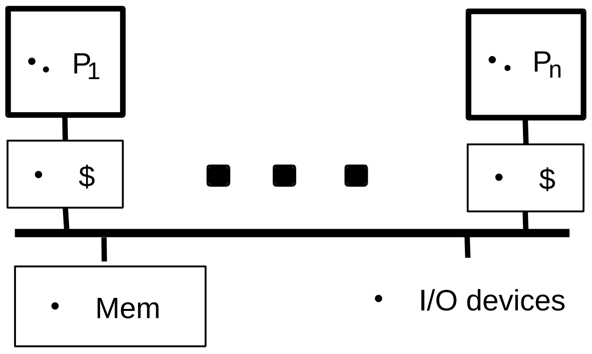
Lecture 4
— Cache Coherence—

Memory Hierarchy

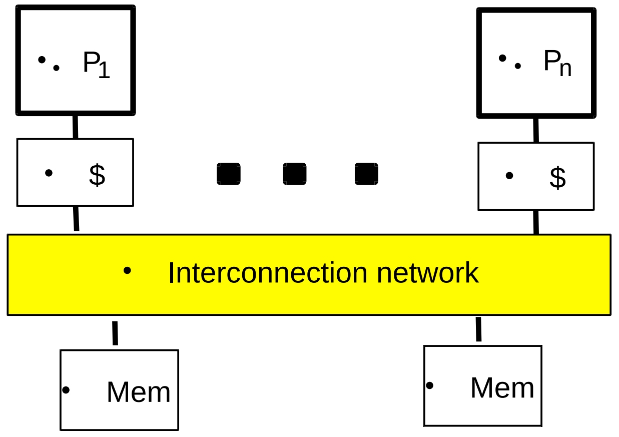
- Most programs have a high degree of locality in their accesses
  - Spatial locality: accessing things nearby previous accesses
  - Temporal locality: accessing an item that was previously accessed
- Memory Hierarchy tries to exploit locality



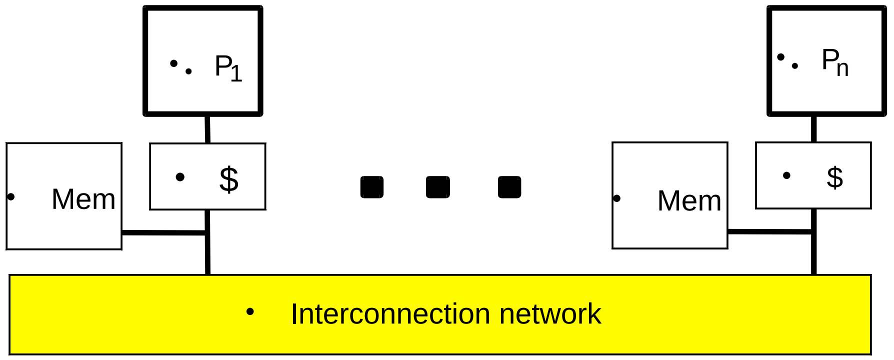
Shared Memory Organizations



- Bus-based Shared Memory



- Dance Hall (UMA)

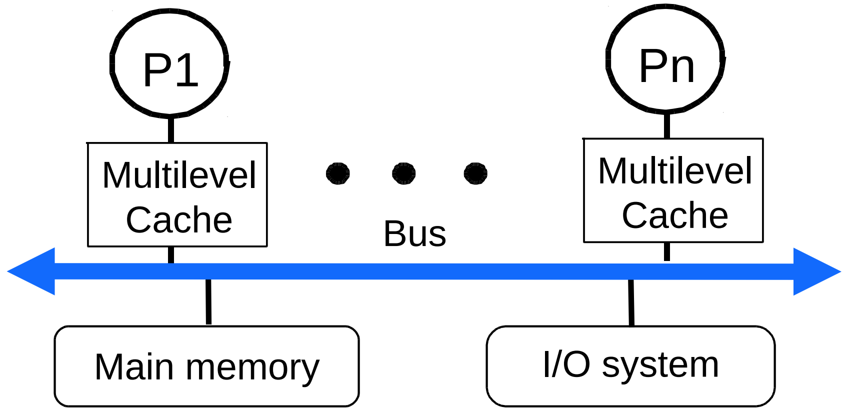


- Distributed Shared Memory (NUMA)

Bus-Based Symmetric Multiprocessors

- Symmetric access to main memory from any processor
- An important architecture until very recently
  - Building blocks for larger systems; arriving to desktop
- Attractive as throughput servers and for parallel programs

  - Uniform access via loads/stores
  - Automatic data movement and coherent replication in caches
  - Cheap and powerful extension to uniprocessors



- Normal uniprocessor mechanisms to access data
  - Key is extension of memory hierarchy to support multiple processors

Caches are Critical for Performance

- Reduce average **latency**
  - Main memory access costs from 100 to 1000 cycles
  - Caches can reduce latency to few cycles
- Reduce average **bandwidth** and demand to access main memory
  - Reduce access to shared bus or interconnect

•Automatic migration of data

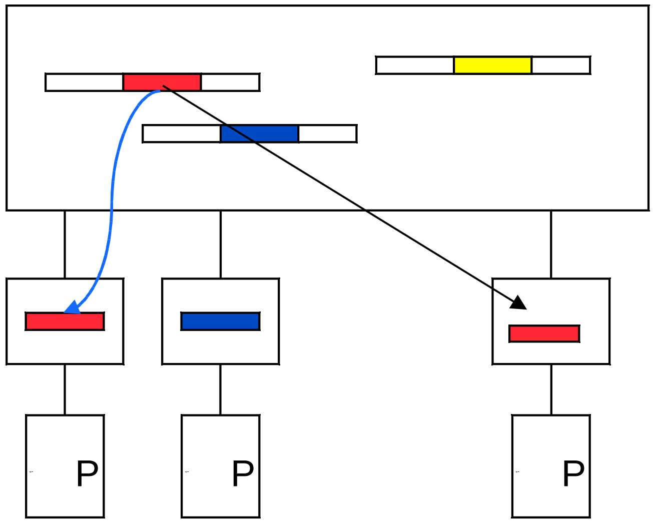
•Data is moved closer to processor

•Automatic replication of data

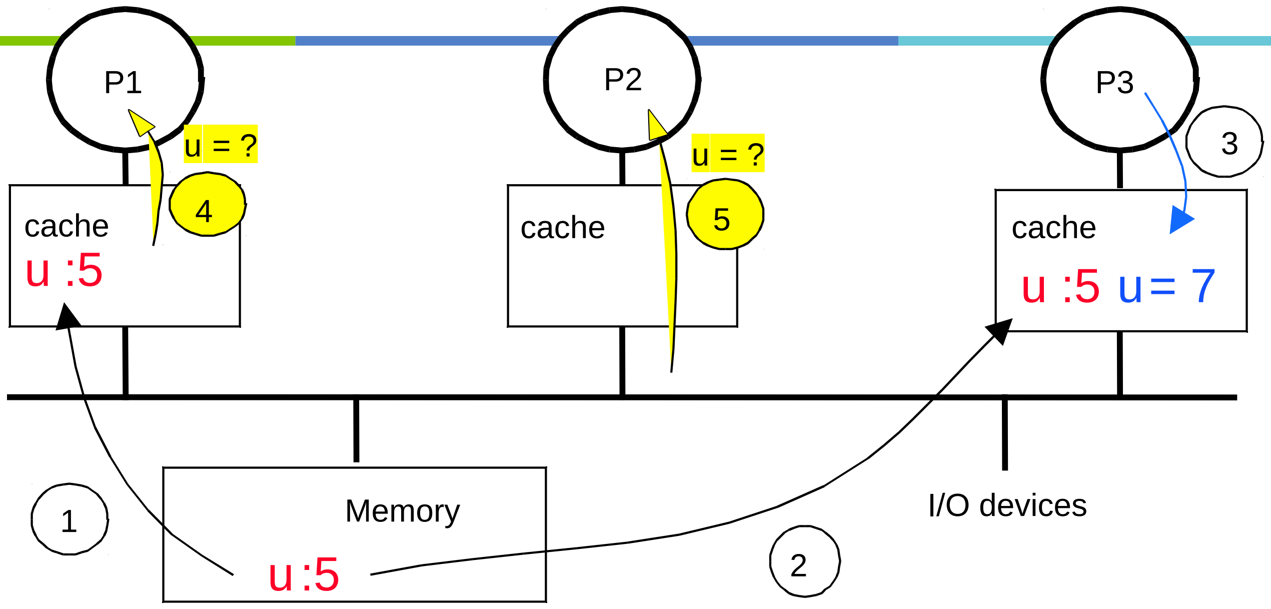
•Shared data is replicated upon need

•Processors can share data efficiently

•But private caches create a problem



Example on Cache Coherence Problem



- Processors see **different** values for *u* after event 3
- With write back caches …
  - Processes accessing main memory may see **stale** (old incorrect) value
  - Value written back to memory depends on sequence of cache flushes
- Unacceptable to programs, and frequent!

Caches and Cache Coherence

- Private processor caches create a problem
  - Copies of a variable can be present in multiple caches
  - A write by one processor may not become visible to others
    - » They'll keep accessing stale value in their caches

  -> *Cache coherence* problem
- What do we do about it?
  - Organize the memory hierarchy to make it go away
  - Detect and take actions to eliminate the problem

What to do about Cache Coherence?

- Organize the memory hierarchy to make it go away
  - Remove private caches and use a shared cache
    - A switch is needed ⇒ added cost and latency
    - Not practical for a large number of processors
- Mark segments of memory as **uncacheable**
  - Shared data or segments used for I/O are not cached
  - Private data is cached only
  - We loose performance
- Detect and take actions to eliminate the problem
  - Can be addressed as a basic hardware design issue
  - Techniques solve both multiprocessor as well as I/O cache coherence

Intuitive Coherent Memory Model

- Caches are supposed to be transparent
- What would happen if there were no caches?
  - All reads and writes would go to main memory
  - Reading a location should return **last value written** by any processor
- What does **last value written** mean in a multiprocessor?
  - All operations on a **particular location** would be **serialized**
  - All processors would **see the same access order** to a particular location
    - If they bother to read that location
- Interleaving among memory accesses from different processors
  - Within a processor ⇒ program order on a given memory location
  - Across processors ⇒ only constrained by explicit synchronization

Formal Definition of Memory Coherence

- A memory system is coherent if there exists a serial order of memory operations on each memory location *X*, such that …

  1. A read by any processor *P* to location *X* that follows a write by processor *Q* (or *P*) to *X* returns the **last written value** if no other writes to *X* occur between the two accesses

  2. Writes to the same location *X* are **serialized**; two writes to same location *X* by any two processors are seen in the same order by all processors

- Two properties

  - **Write propagation**: writes become visible to other processors

  - **Write serialization**: writes are seen in the same order by all processors

Hardware Coherency Solutions

- **Bus Snooping Solution**
  - Send all requests for data to all processors
  - Processors snoop to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is in processors
  - Works well with bus (natural broadcast medium)
  - Dominates for small scale multiprocessors (most of the market)
- **Directory-Based Schemes**
  - Keep track of what is being shared in one logical place
  - Distributed memory $\Rightarrow$ distributed directory
  - Send point-to-point requests to processors via network
  - Scales better than Snooping and avoids bottlenecks

Hardware Coherency Solutions

- Write-through: the data is written both into the cache and passed on to the next lower level in the memory hierarchy
- Write-back: the data is written only into the first level cache. Only when the line is replaced, the data is transferred to the next level in memory hierarchy

Write-through VS Write-back

- Write-through protocol is simple
  - Every write is observable
- However, every write goes on the bus
  - Only one write can take place at a time in any processor
- Uses a lot of bandwidth!
- Write-back caches absorb most writes as cache hits
  - But write hits don’t go on bus – need more sophisticated protocols

Hardware Coherency Solutions

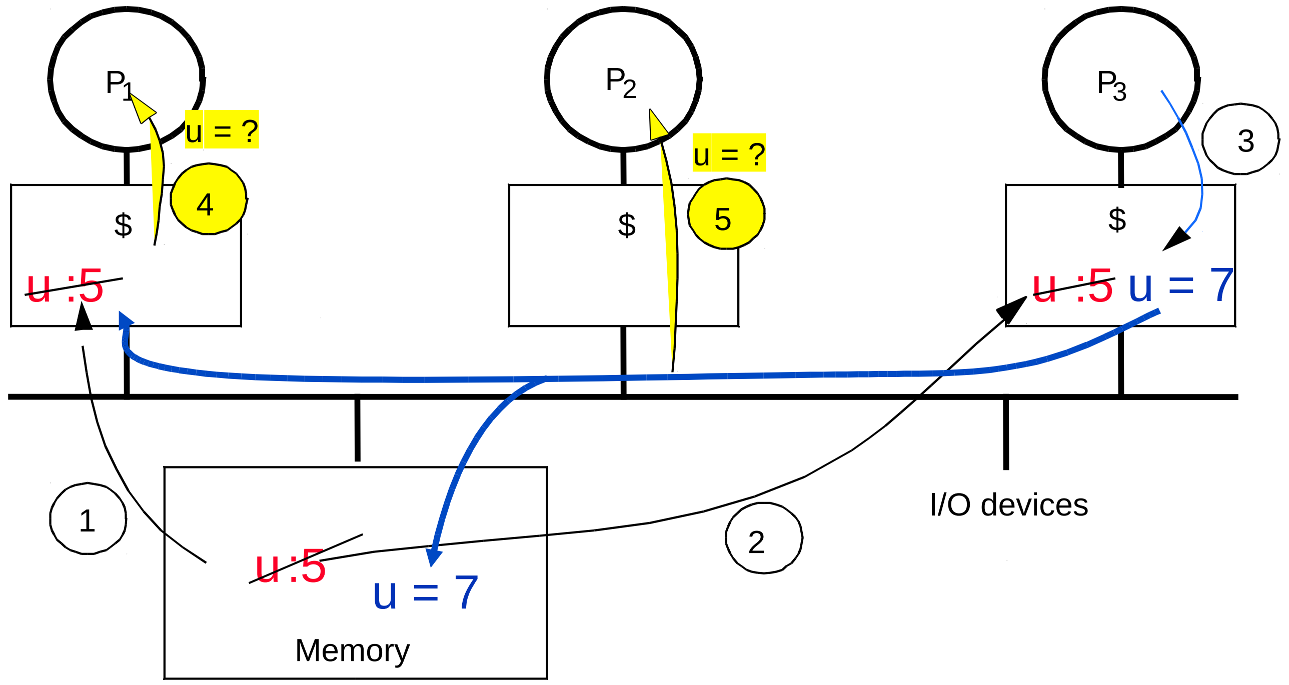
- Write-invalidate: a processor gains exclusive access of a block before writing by invalidating all other copies
- Write-update: when a processor writes, it updates other shared copies of that block

Invalidate VS Update

- Basic question of program behavior:
  - Is a block written by one processor later read by others before it is overwritten?
- Invalidate.
  - yes: readers will take a miss
  - no: multiple writes without addition traffic
    - also clears out copies that will never be used again
- Update.
  - yes: avoids misses on later references
  - no: multiple useless updates

⇒ Need to look at program reference patterns and hardware complexity

Example of Write-through Invalidate



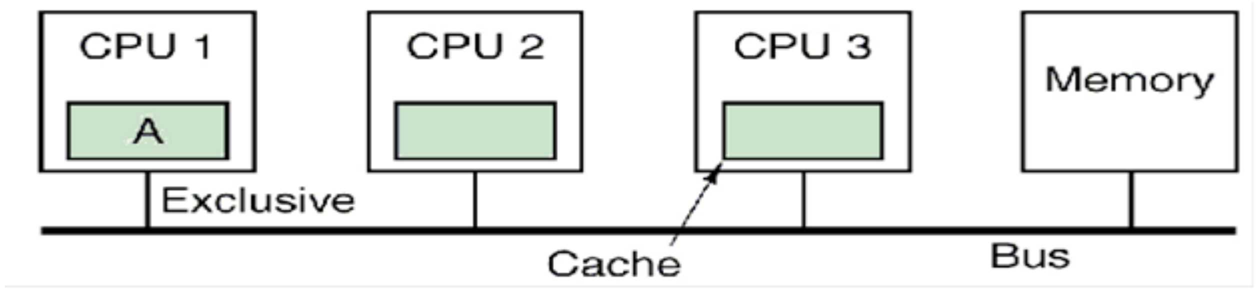
- At step 4, an attempt to read *u* by P1 will result in a cache miss
  - Correct value of *u* is fetched from memory

- Similarly, correct value of *u* is fetched at step 5 by P2

MESI (write-invalidate, write-back)

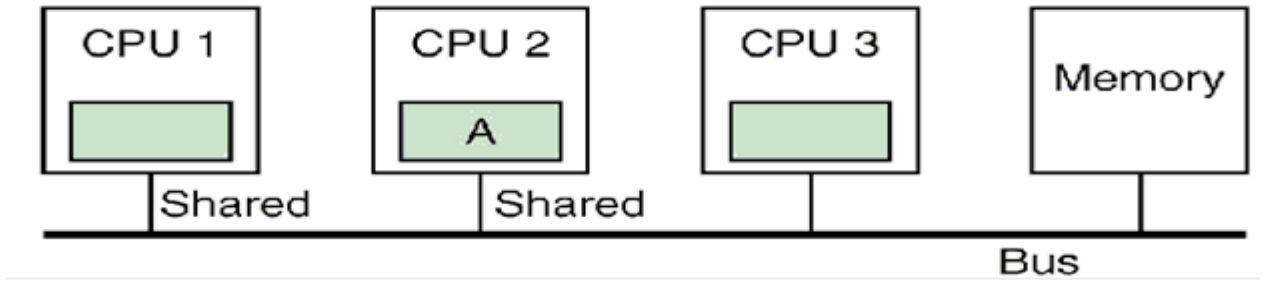
- **M: Modified**
  - Only this cache has copy and is modified
  - Main memory copy is stale
- **E: Exclusive** or *exclusive-clean*
  - Only this cache has copy which is not modified
  - Main memory is up-to-date
- **S: Shared**
  - More than one cache may have copies, which are not modified
  - Main memory is up-to-date
- **I: Invalid**
- Know also as Illinois protocol
  - First published at University of Illinois at Urbana-Champaign
  - Variants of MESI protocol are used in many modern microprocessors

MESI Illustrated (step 1)



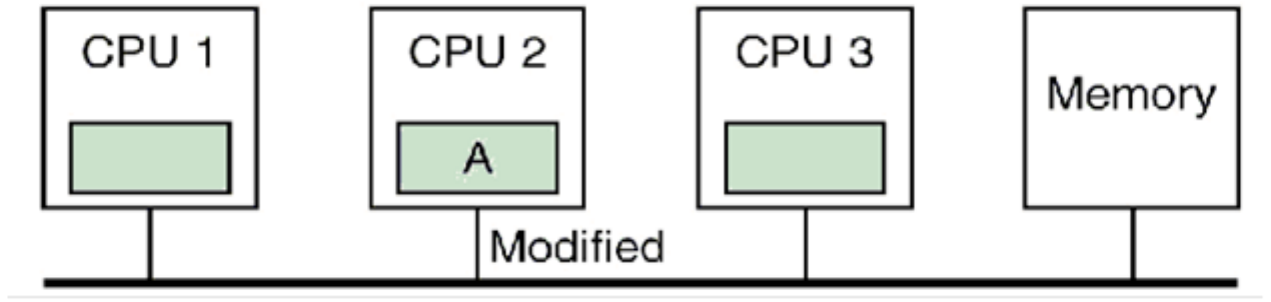
- When the multiprocessor is turned on, all cache lines are marked invalid.
- CPU 1 requests block A from the shared memory.
- It issues a BR (Bus Read) for the block and gets its copy.
- The cache line containing block A is marked Exclusive.
- Subsequent reads to this block access the cached entry and not the shared memory.
- Neither CPU 2 nor CPU 3 respond to the BR.

MESI Illustrated (step 2)



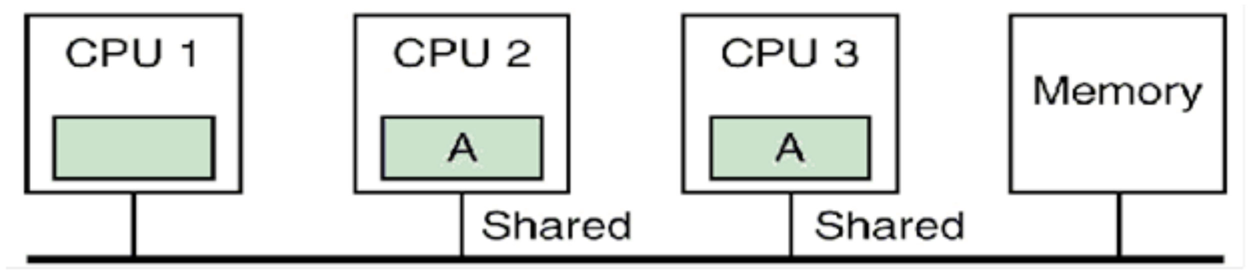
- CPU 2 requests the same block. The snoop cache on CPU 1 notes the request and CPU 1 broadcasts “Shared”, announcing that it has a copy of the block.
- Both copies of the block are marked as shared.
- This indicates that the block is in two or more caches for reading and that the copy in the shared primary memory is up to date.
- CPU 3 does not respond to the BR.

MESI Illustrated (step 3)



- Suppose that CPU 2 writes to the cache line it is holding in its cache. It issues a BU (Bus Upgrade) broadcast, marks the cache line as Modified, and writes the data to the line.
- CPU 1 responds to the BU by marking the copy in its cache line as Invalid.
- CPU 3 does not respond to the BU.
- Informally, CPU 2 can be said to “own the cache line”.

MESI Illustrated (step 4)

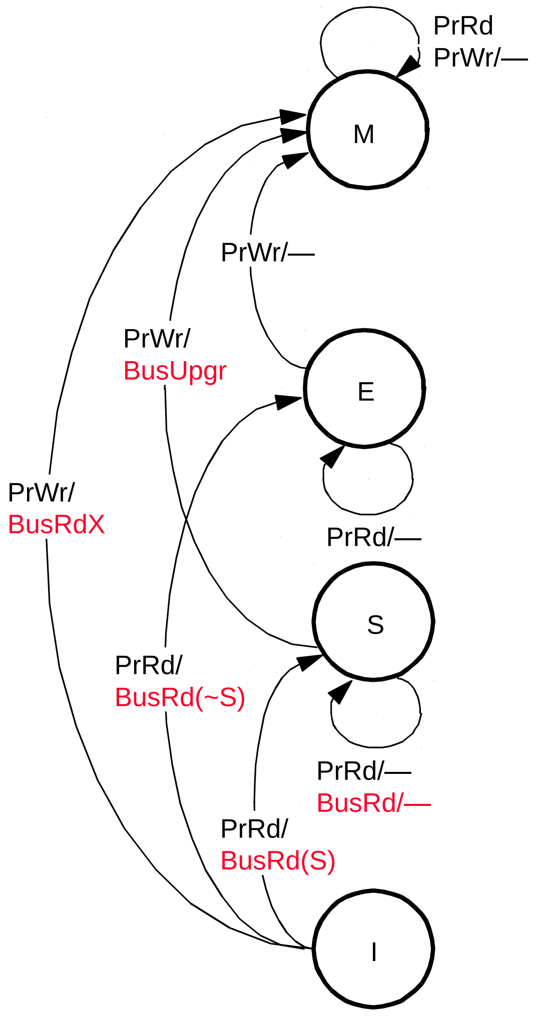


- Now suppose that CPU 3 attempts to read block A .
- For CPU 1, the cache line holding that block is marked as Invalid. CPU 1 does not respond to the BR (Bus Read).
- CPU 2 has the cache line marked as Modified. It asserts the signal “Dirty” on the bus, writes the data in the cache line back to the shared memory, and marks the line “Shared”.
- Informally, CPU 2 asks CPU 3 to wait while it writes back the contents of its modified cache line to the shared primary memory. CPU 3 waits and then gets a correct copy. The cache line in each of CPU 2 and CPU 3 is marked as Shared.

MESI State Transition Diagram

❖ **Processor Read**
- Causes a BusRd on a read miss
- BusRd(**S**) => shared line asserted
  - Valid copy in another cache
  - Goto state *S*
- BusRd(**~S**) => shared line not asserted
  - No cache has this block
  - Goto state *E*
- No bus transaction on a read hit

❖ **Processor Write**
- Promotes block to state *M*
- Causes BusRdX / BusUpgr for states ***I* / *S***
  - To invalidate other copies
- No bus transaction for states ***E* and *M***

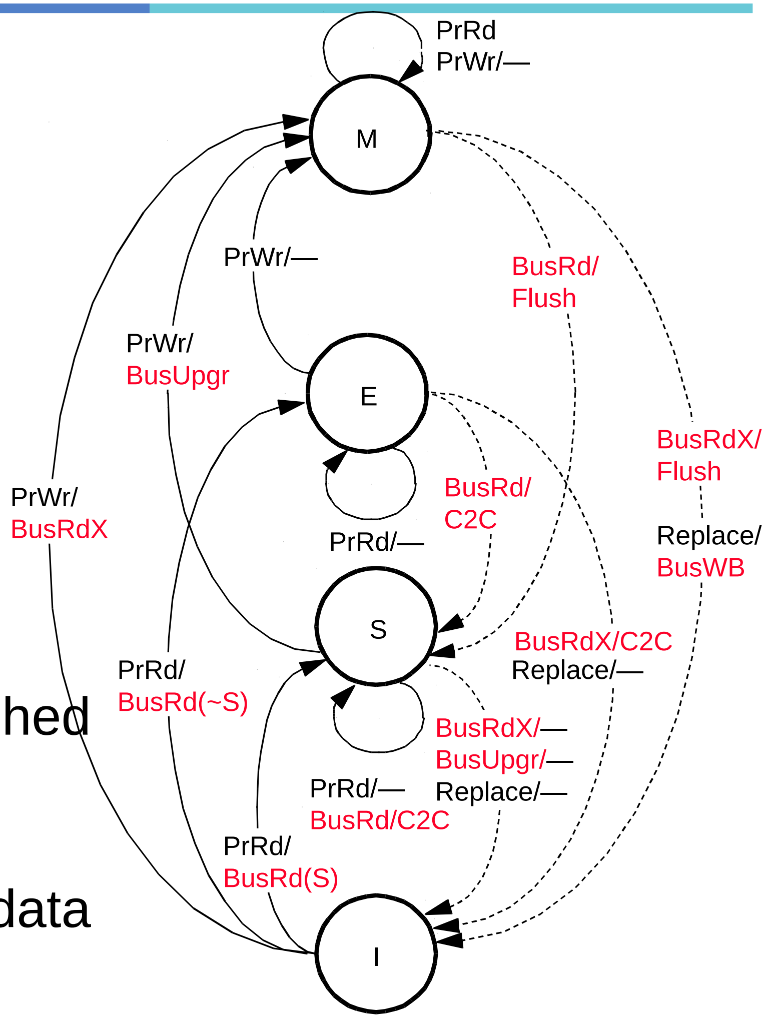


MESI State Transition Diagram – cont’d

❖**Observing a BusRd**
- Demotes a block from *E* to *S* state
  - Since another cached copy exists
- Demotes a block from *M* to *S* state
  - Will cause modified block to be flushed
  - Block is picked up by requesting cache and main memory

❖**Observing a BusRdX or BusUpgr**
- Will invalidate block
- Will cause a modified block to be flushed

❖Cache-to-Cache (**C2C**) Sharing
- Supported by original Illinois version
- Cache rather than memory supplies data


PrRd
PrWr/—
M
PrWr/—
BusRd/
Flush
PrWr/
BusUpgr
E
BusRdX/
Flush
PrWr/
BusRdX
BusRd/
C2C
PrRd/—
Replace/
BusWB
S
BusRdX/C2C
Replace/—
PrRd/
BusRd(~S)
BusRdX/—
BusUpgr/—
PrRd/—
Replace/—
BusRd/C2C
PrRd/
BusRd(S)
I