Cache Coherency in Multiprocessor Systems

The ***Modified Exclusive Shared Invalid*** (MESI) algorithm for cache coherency.

MESI State	Definition
Modified (M)	The line is valid in the cache and in only this cache. The line is modified with respect to system memory—that is, the modified data in the line has not been written back to memory.
Exclusive (E)	The addressed line is in this cache only. The data in this line is consistent with system memory.
Shared (S)	The addressed line is valid in the cache and in at least one other cache. A shared line is always consistent with system memory. That is, the shared state is shared-unmodified; there is no shared-modified state.
Invalid (I)	This state indicates that the addressed line is not resident in the cache and/or any data contained is considered not useful.

Note that:

- Exclusive may also be called CleanExclusive
- Modified may also be called DirtyExclusive

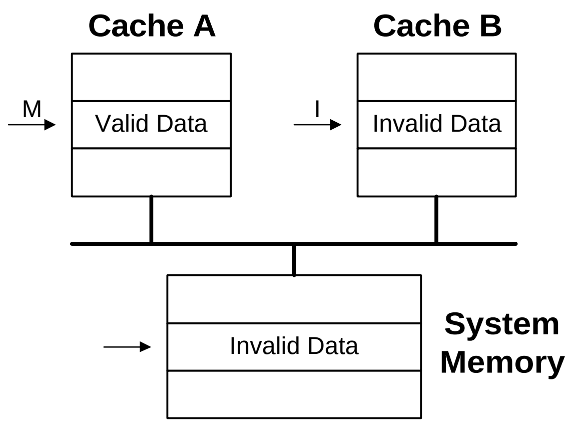
Some processors add a fifth state for ***Shared Modified*** and call it the ***MOESI*** protocol. The caches with the shared modified state update each other's lines with current data, but do not write it back to main memory.

The five MOESI states are defined as:

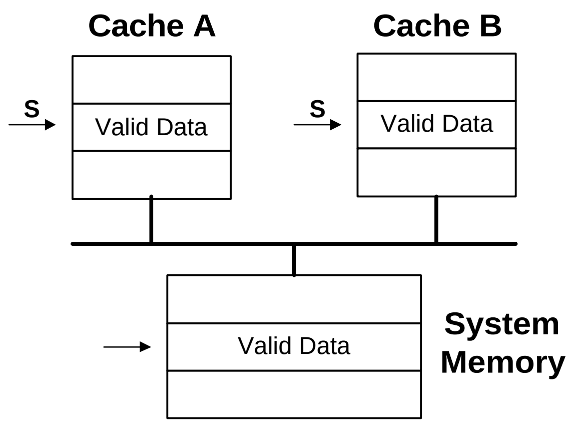
- Exclusively Modified (M)
- Shared Modified (O)
- Exclusive Clean (E)
- Shared Clean (S)
- Invalid (I)

Picture the MESI cache states:

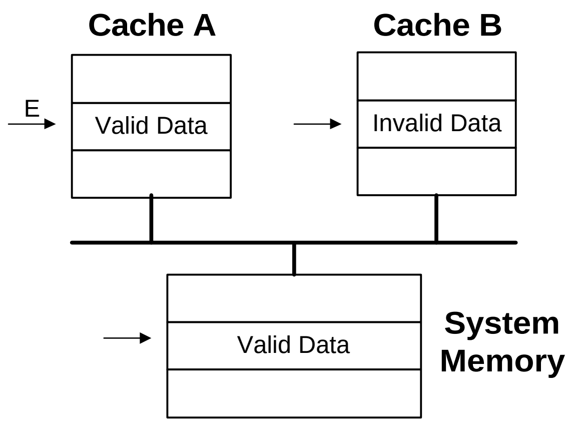
Modified in Cache A



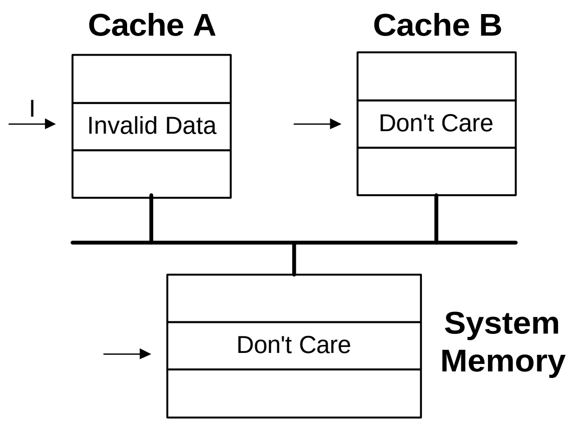
Shared in Cache A



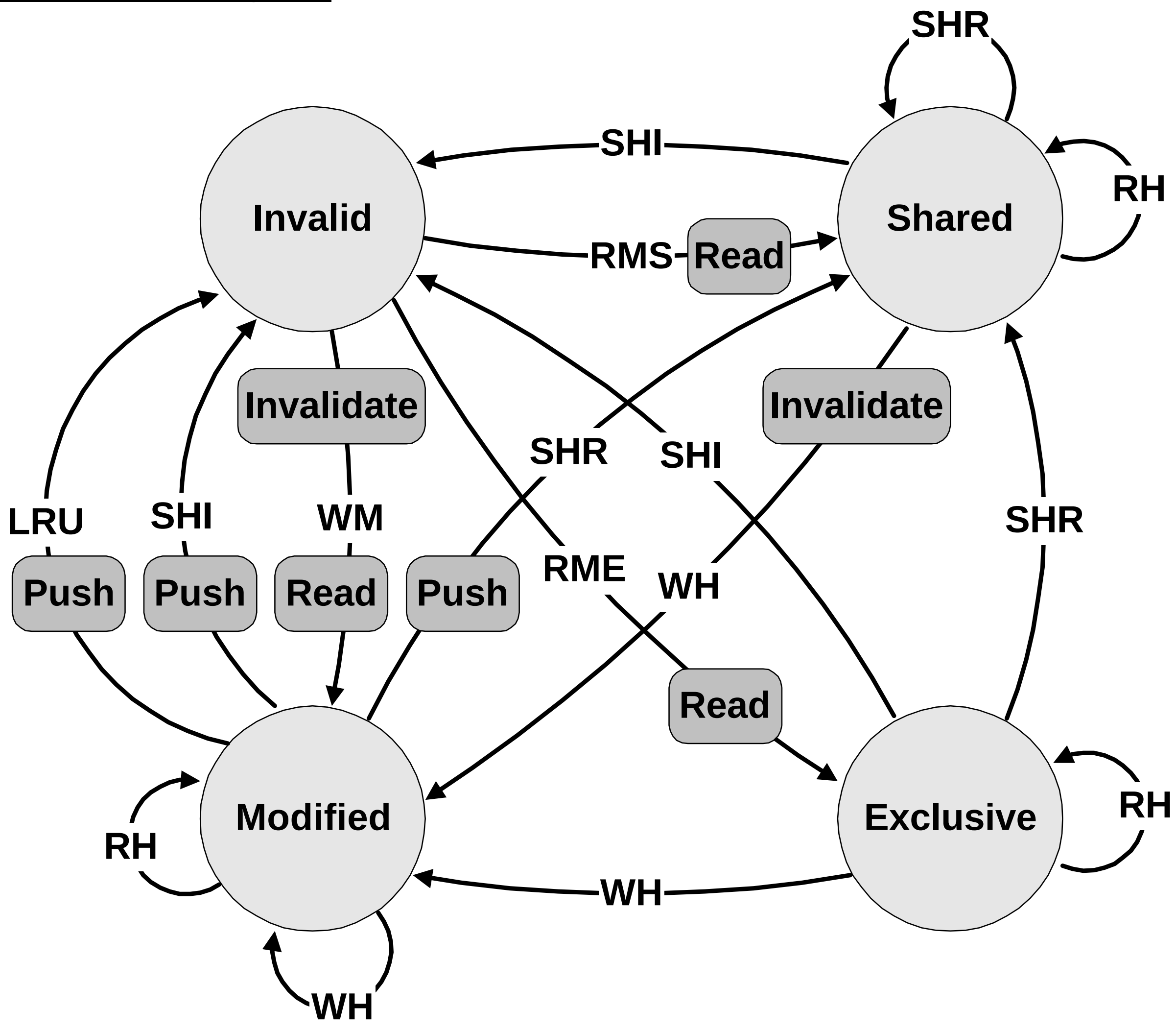
Exclusive in Cache A



Invalid in Cache A



MESI State Diagram



Events:

RH	= Read Hit
RMS	= Read miss, shared
RME	= Read miss, exclusive
WH	= Write hit
WM	= Write miss
SHR	= Snoop hit on read
SHI	= Snoop hit on invalidate
LRU	= LRU replacement

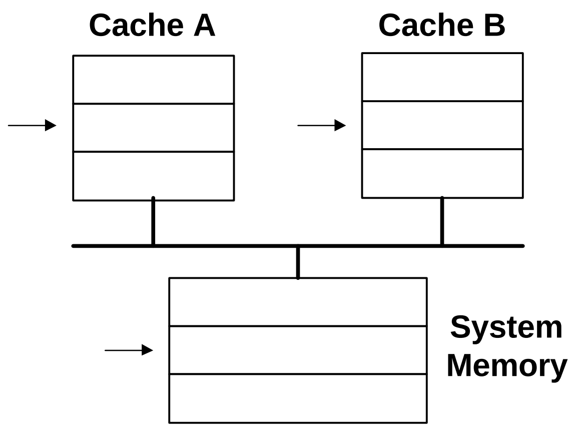
Bus Transactions:

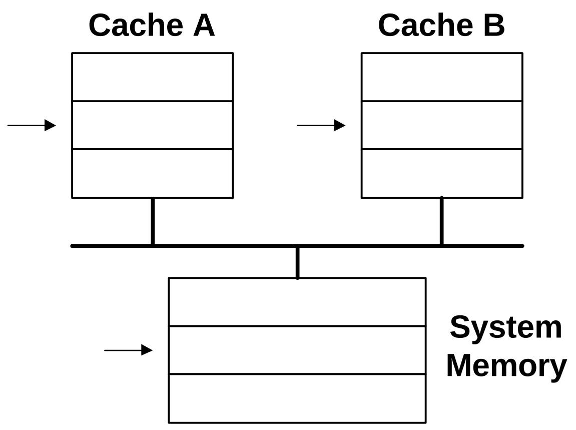
Push	= Write cache line back to memory
Invalidate	= Broadcast invalidate
Read	= Read cache line from memory

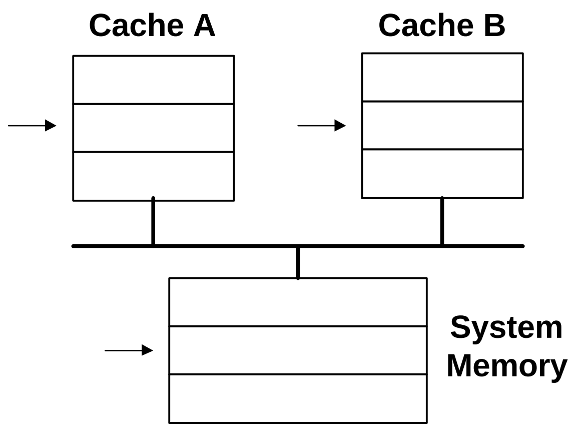
MESI State Table

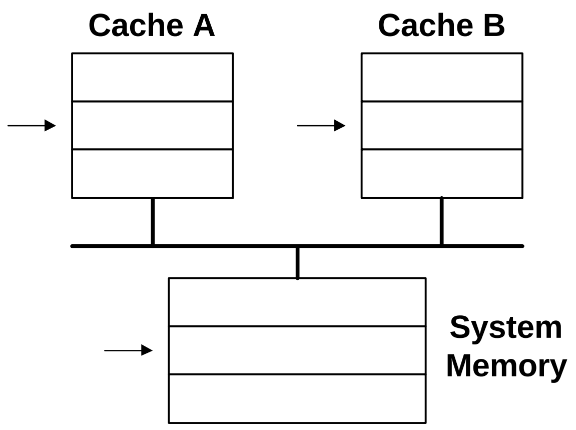
State	Event	Action	Next State
Invalid	Read miss, shared (cache copies exist)	Read cache line	Shared
	Read miss, exclusive (no cache copies exist)	Read cache line	Exclusive
	Write miss	• Broadcast invalidate • Read cache line • Modify cache line	Modified
Shared	Read hit		Shared
	Write hit	Broadcast invalidate	Modified
	Snoop hit on read		Shared
	Snoop hit on invalidate	Invalidate cache line	Invalid
Exclusive	Read hit		Exclusive
	Write hit		Modified
	Snoop hit on read		Shared
	Snoop hit on invalidate	Invalidate cache line	Invalid
Modified	Read hit		Modified
	Write hit		Modified
	Snoop hit on read	Write cache line back to memory	Shared
	Snoop hit on invalidate	Write cache line back to memory	Invalid
	LRU Replacement	Write cache line back to memory	Invalid

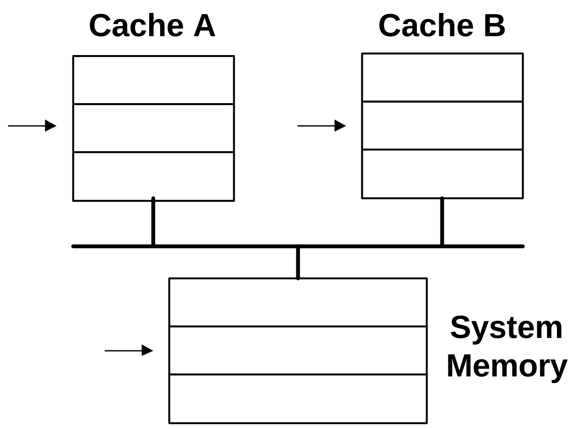
.

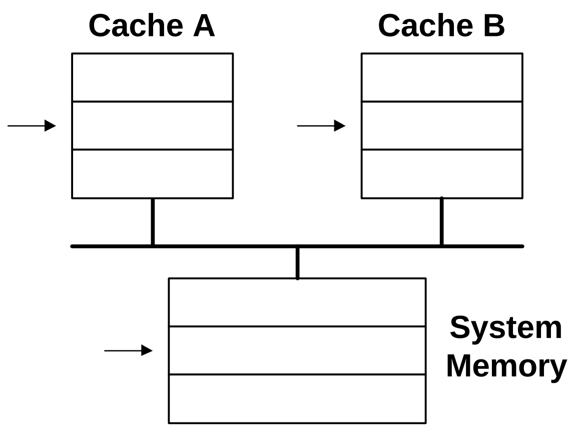

Cache A
Cache B
System Memory


Cache A
Cache B
System Memory


Cache A
Cache B
System Memory


Cache A
Cache B
System Memory


Cache A
Cache B
System Memory


Cache A
Cache B
System Memory

<u>The synchronization problem</u>, Stone pp 366-369 and section 7.2

<u>A *race* condition:</u>

<u>Processor A</u>	**<u>Processor B</u>**
R3 ← Memory(SharedLoc)	R5 ← Memory(SharedLoc)
R3 ← R3 + 300	R5 ← R5 + 200
Memory(SharedLoc) ← R3	Memory(SharedLoc) ← R5

<u>The solution:</u>
Provide a ***Lock*** or ***Semaphore*** for Memory(SharedLoc). The lock instruction must provide for an ***<u>atomic</u>*** read-modify-write in the hardware.

<u>Processor A</u>	**<u>Processor B</u>**
Lock SpinLock1 (success)	Lock SpinLock1 (fail)
R3 ← Memory(SharedLoc)	Lock SpinLock1 (fail)
R3 ← R3 + 300	Lock SpinLock1 (fail)
Memory(SharedLoc) ← R3	Lock SpinLock1 (fail)
Unlock SpinLock1	Lock SpinLock1 (success)
	R5 ← Memory(SharedLoc)
	R5 ← R5 + 200
	Memory(SharedLoc) ← R5
	Unlock SpinLock1

Actually, the Lock instruction would be implemented as (for example) Stone's ***Test-and-set*** (TS) instruction (p.424).

Test-and-set MemoryLoc – in one memory cycle:

- read out the contents of the addressed word & set CC
- write back all 1's

Define lock = any memory location (e.g. SpinLock1)
    if the memory location = 0, the lock is unlocked
    if the memory location = 1, the lock is locked

```
Lock:   TEST-AND-SET SpinLock1      Unlock:   STORE SpinLock1, 0
        BRANCH not-zero Lock
        Continue with locked code
```

What happens to the caches of processors A and B, and to the memory bus during this sequence?

Assume both coming from Invalid state:

Processor A	**Processor B**
Write miss on TS LockLoc	
Broadcast Invalidate	Write miss on TS LockLoc
Read cache line	Broadcast invalidate
Modify cache line with TS	--- wait for OK from owner
Marks cache line modified	
Recognizes snoop invalidate	
Writes back cache line	
Marks cache line invalid	
Send OK to proceed	Receives OK to proceed
continues with locked code	Read cache line
	Modify cache line with TS
	Marks cache line modified
	Spins on TS instruction in cache
Sometime later	. . . .
Write miss on store 0 into LockLoc	
Broadcast invalidate	
--- wait on OK from owner	Recognizes snoop invalidate
	Writes back cache line
	Marks cache line invalid
Receives OK to proceed	Send OK to proceed
Read cache line	Write miss on TS LockLoc
Modify cache line with store 0	Broadcast invalidate
Marks cache line modified	--- wait for OK from owner
Recognizes snoop invalidate	
Writes back cache line	
Marks cache line invalid	
Send OK to proceed	Receives OK to proceed
	Read cache line
	Modify cache line with TS
	Marks cache line modified
	continues with locked code

Can you imagine what happens if there are three or more processors contending for the lock?

Solutions to cache thrashing on synchronization locks:

- Use normal non-cached memory locations for locks
  ⇒ Still saturates the memory bus

- Provide a special, high-speed non-cached memory for locks
  ⇒ Also need a special bus to access it

- Provide a special synchronization processor to handle locks
  ⇒ Best solution for high-performance

The memory consistency problem – Stone section 6.5 (p.392)

♦ We optimize execution of a processor by using several execution units (superscalar)

♦ Some instructions are executed out of order, but the ***processor ensures correct result-ordering*** where explicit dependencies exist.

♦ The LOCK and UNLOCK instructions must be defined to have explicit dependencies ***on all other instructions***:

⇒ The LOCK instruction must ensure that NO instructions following it have started prior to its completion (Stone's ACQUIRE).

⇒ The UNLOCK instruction must ensure that ALL instructions preceding it have completed prior to its beginning execution (Stone's RELEASE).

♦ These orderings must be maintained ***across all processors and memories***. See Stone page 396-401 for the discussion on ***weak consistency*** and ***release consistency***.

Strong Consistency

- ***All*** instructions must have globally consistent ordering.

Weak Consistency

- ***Synchronizing*** instructions must have globally consistent ordering with all instructions.

Acquire Consistency

- The ***acquire*** (lock) instruction must be globally complete before any following instruction begins.

Release Consistency

- The ***release*** (lock) instruction must not begin before all preceding instructions are globally complete.

From the MIPS R10000 User Manual

The R10000 processor ***behaves*** as if strong ordering is implemented, although it does not actually execute all memory operations in strict program order.

In the R10000 processor, store operations remain ***pending*** until the store instruction is ready to graduate. Thus, stores are executed in program order, and memory values are precise following any exception.

For improved performance however, cached load operations my occur in any order, subject to memory dependencies on pending store instructions. To maintain the appearance of strong ordering, the processor detects whenever the reordering of a cached load might alter the operation of the program, backs up, and then re-executes the affected load instructions.

Specifically:

- ***Whenever a primary data cache block is invalidated*** due to an external coherency request, its index is compared with all outstanding load instructions.
- If there is a match and the load has been completed, the load is prevented from graduating.
- When it is ready to graduate, ***the entire pipeline is flushed***, and the processor is restored to the state it had before the load was decoded.

Since the R10000 processor ***behaves*** as if it implemented strong ordering, a suitable system design allows the processor to be used to create a shared-memory multiprocessor system with strong ordering.