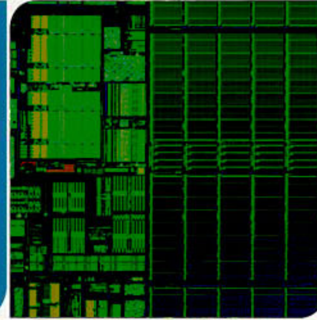
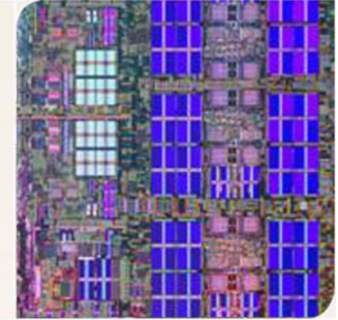
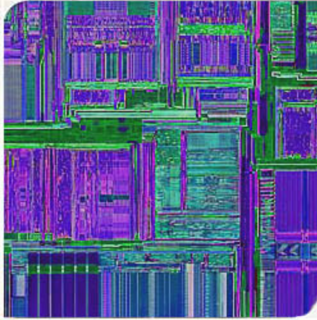
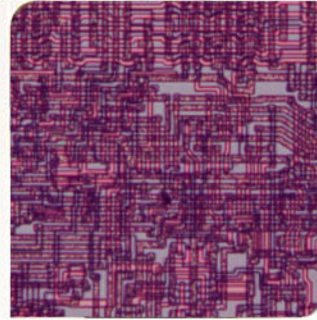
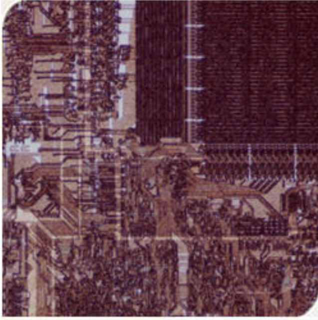
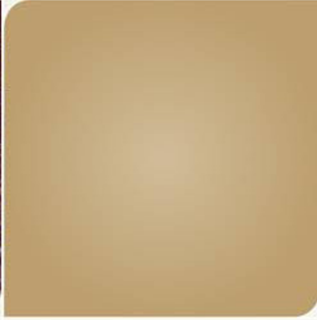





CS6290

Fall 2009

Prof. Hyesoon Kim

Thanks to Prof. Loh & Prof. Prvulovic

Problem

P1
LD MEM[A1]
ST MEM[A1] 20
$
A1: 20

P2
LD MEM[A1]
$
10 ? 20

P3
$

Main Memory

A1: 10 / A1: 20
A2: 20
A3: 39
A4: 17

No problem in writeback

P1
ST MEM[A1] 20
$
A1: 20

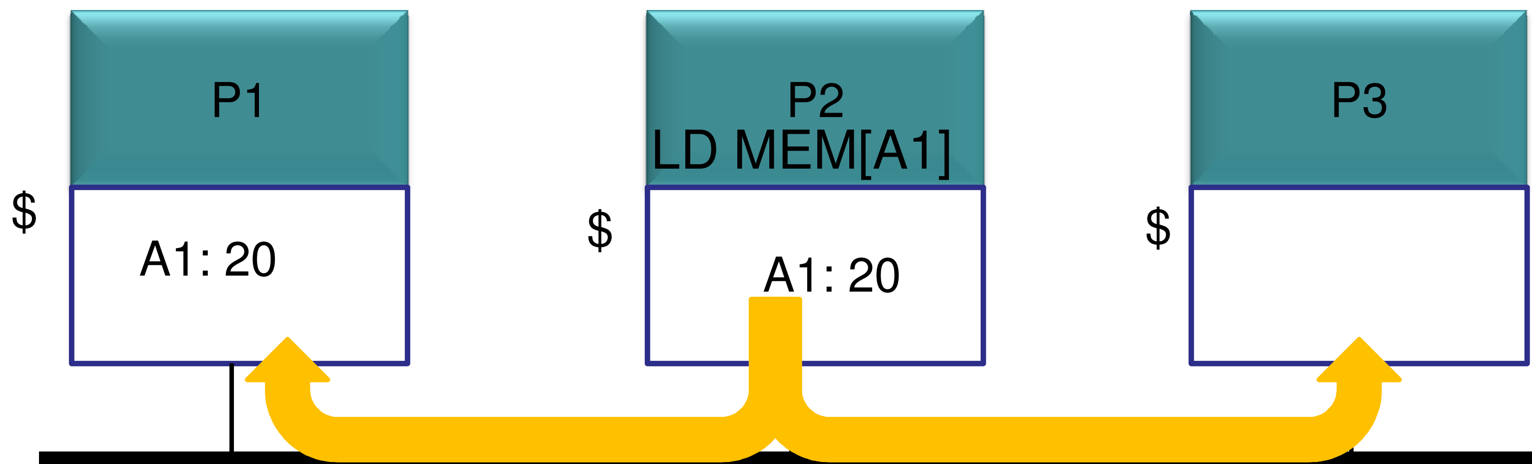
P2
LD MEM[A1]
$
A1: 20

P3
$

Main Memory

A1: 10
A2: 20
A3: 39
A4: 17

SNOOP



Main Memory

A1: 20
A2: 20
A3: 39
A4: 17

SNOOP

P1
$
A1: 20

P2
$
A1: 20

P3
ST MEM[A2]
$
A2: 30

Main Memory

A1: 20
A2: 20
A3: 39
A4: 17

Cache Coherence Problem

- Shared memory easy with no caches
  - P1 writes, P2 can read
  - Only one copy of data exists (in memory)
- Caches store their own copies of the data
  - Those copies can easily get inconsistent
  - Classic example: adding to a sum
    - P1 loads allSum, adds its mySum, stores new allSum
    - P1's cache now has dirty data, but memory not updated
    - P2 loads allSum from memory, adds its mySum, stores allSum
    - P2's cache also has dirty data
    - Eventually P1 and P2's cached data will go to memory
    - Regardless of write-back order, the final value ends up wrong

Cache Coherence Definition

- A memory system is coherent if
  1. A read R from address X on processor P1 returns the value written by the most recent write W to X on P1 if no other processor has written to X between W and R.
  2. If P1 writes to X and P2 reads X after a sufficient time, and there are no other writes to X in between, P2's read returns the value written by P1's write.
  3. Writes to the same location are serialized: two writes to location X are seen in the same order by all processors.

Cache Coherence Definition

- Property 1. preserves program order
  - It says that in the absence of sharing, each processor behaves as a uniprocessor would
- Property 2. says that any write to an address must eventually be seen by all processors
  - If P1 writes to X and P2 keeps reading X, P2 must eventually see the new value
- Property 3. preserves causality
  - Suppose X starts at 0. Processor P1 increments X and processor P2 waits until X is 1 and then increments it to 2. Processor P3 must eventually see that X becomes 2.
  - If different processors could see writes in different order, P2 can see P1's write and do its own write, while P3 first sees the write by P2 and then the write by P1. Now we have two processors that will forever disagree about the value of A.

Snooping

- Typically used for bus-based (SMP) multiprocessors
  - Serialization on the bus used to maintain coherence property 3
- Two flavors
  - Write-update (write broadcast)
    - A write to shared data is broadcast to update all copies
    - All subsequent reads will return the new written value (property 2)
    - All see the writes in the order of broadcasts
      One bus == one order seen by all (property 3)
  - Write-invalidate
    - Write to shared data forces invalidation of all other cached copies
    - Subsequent reads miss and fetch new value (property 2)
    - Writes ordered by invalidations on the bus (property 3)

Update vs. Invalidate

- A burst of writes by a processor to one addr
  - Update: each sends an update
  - Invalidate: possibly only the first invalidation is sent
- Writes to different words of a block
  - Update: update sent for each word
  - Invalidate: possibly only the first invalidation is sent
- Producer-consumer communication latency
  - Update: producer sends an update, consumer reads new value from its cache
  - Invalidate: producer invalidates consumer's copy, consumer's read misses and has to request the block
- Which is better depends on application
  - But write-invalidate is simpler and implemented in most MP-capable processors today

MSI Snoopy Protocol

- State of block B in cache C can be
  - Invalid: B is not cached in C
    - To read or write, must make a request on the bus
  - Modified: B is dirty in C
    - has the block, no other cache has the block, and C must update memory when it displaces B
    - Can read or write B without going to the bus
  - Shared: B is clean in C
    - C has the block, other caches have the block, and C need not update memory when it displaces B
    - Can read B without going to bus
    - ***To write, must send an upgrade request to the bus***

MSI Snoopy Protocol

Write miss for this block
Invalidate for this block
CPU read
Place read miss on bus
CPU read hit
Invalid
Shared
(read only)
CPU read miss
Place read miss on bus
Write-back block
Place write miss on bus
CPU write
CPU read miss
Write-back data; place read miss on bus
Read miss for block
Write-back block
Place invalidate on bus
CPU write
CPU write miss
Place write miss on bus
Write miss for block
Exclusive
(read/write)
CPU write hit
CPU read hit
CPU write miss
Write-back data
Place write miss on bus

Also called "modified"

My RD/WR:
My action

Someone else

© 2007 Elsevier, Inc. All rights reserved.

Cache to Cache transfers

- Problem
  - P1 has block B in M state
  - P2 wants to read B, puts a RdReq on bus
  - If P1 does nothing, memory will supply the data to P2
  - What does P1 do?
- Solution 1: abort/retry
  - P1 cancels P2's request, issues a write back
  - P2 later retries RdReq and gets data from memory
  - Too slow (two memory latencies to move data from P1 to P2)
- Solution 2: intervention
  - P1 indicates it will supply the data ("intervention" bus signal)
  - Memory sees that, does not supply the data, and waits for P1's data
  - P1 starts sending the data on the bus, memory is updated
  - P2 snoops the transfer during the write-back and gets the block

Cache-To-Cache Transfers

- Intervention works if some cache has data in M state
  - Nobody else has the correct data, clear who supplies the data
- What if a cache has requested data in S state
  - There might be others who have it, who should supply the data?
  - Solution 1: let memory supply the data
  - Solution 2: whoever wins arbitration supplies the data
  - Solution 3: A separate state similar to S that indicates there are maybe others who have the block in S state, but if anybody asks for the data we should supply it

MSI Protocol

- Three states:
  - Invalid
  - Shared (clean)
  - Modified (dirty)

MESI Protocol

- New state: exclusive
  - data is clean
  - but I have the only copy (except memory)
- Benefit: bandwidth reduction

Detecting Other Sharers

- Problem
  - P1 wants to read B, puts a RdReq on bus, receives data
  - How does P1 know whether to cache B in S or E state?
- Solution: “Share” bus signal
  - Always low, except when somebody pulls it to high
  - When P2 snoops P1’s request, it pulls “Share” to high
  - P1 goes to S if “Share” high, to E if “Share” low

MOESI

- M: Modified
  - I have the only copy, and it's dirty (memory is stale)
- O: Owned
  - I have the most up-to-date copy, others may have copies, too, but I am responsible for sourcing data
- E: Exclusive
  - I have the only copy (clean)
- S: Shared
  - Everyone has a clean copy (incl. memory)
- I: Invalid

Directory-Based Coherence

- Typically in distributed shared memory
- For every local memory block, local directory has an entry
- Directory entry indicates
  - Who has cached copies of the block
  - In what state do they have the block

Basic Directory Scheme

- Each entry has
  - One dirty bit (1 if there is a dirty cached copy)
  - A presence vector (1 bit for each node)
    Tells which nodes may have cached copies
- All misses sent to block's home
- Directory performs needed coherence actions
- Eventually, directory responds with data

Read Miss

- Processor P_k has a read miss on block B, sends request to home node of the block
- Directory controller
  - Finds entry for B, checks D bit
  - If D=0
    - Read memory and send data back, set P[k]
  - If D=1
    - Request block from processor whose P bit is 1
    - When block arrives, update memory, clear D bit, send block to P_k and set P[k]

Directory Operation

- Network controller connected to each bus
  - A proxy for remote caches and memories
    - Requests for remote addresses forwarded to home, responses from home placed on the bus
    - Requests from home placed on the bus, cache responses sent back to home node
- Each cache still has its own coherence state
  - Directory is there just to avoid broadcasts and order accesses to each location
    - Simplest scheme:
      If access A1 to block B still not fully processed by directory when A2 arrives, A2 waits in a queue until A1 is done

Shared Memory Performance

- Another “C” for cache misses
  - Still have Compulsory, Capacity, Conflict
  - Now have Coherence, too
    - We had it in our cache and it was invalidated
- Two sources for coherence misses
  - True sharing
    - Different processors access the same data
  - False sharing
    - Different processors access different data, ***but they happen to be in the same block***