Advanced Cache Topics

Cache Coherency & Consistency

- A must, to avoid using wrong data (e.g., MP, DMA)
- Solution:
  Pass information about changes between caches & memory
- The process in which one element notifies of its actions and others listen is called ***SNOOPING ****
- Idea:
  - Main memory is passive
  - When needed, caches distribute changes to memory or other caches
  - All caches listen to snoop messages and act when needed

*** Other schemes exist, but SNOOPing is the only practical scheme in use**

Write Through
Simple Coherency Protocol

- All memory writes are written to main memory, even if the line is in cache
- Each line has 2 states: valid/invalid
  - A line is invalidated if another processor changed the memory portion occupied by that line
- Activity:

Action	Cache	Bus	Other Caches
Read Hit	Read	---	---
Read Miss	Read	Line Fill	---
Write Hit	Write	Write	Snoop/INV
Write Miss	---	Write	Snoop/INV


Simple


Slow, lot of bus traffic (immediate writes, small chunks)

- Write buffers are used to avoid write latency issues

Write Back

- Distributed cache management
- Idea: Hold a line in the cache, even if modified, as long as other agents do not need it.
- Update content externally, only when another agent needs it
- "Agent" informs others of intentions:
  - Read a copy
  - Modify a copy
  - Broadcast modifications
- Others respond
  - Have a copy
  - Have a dirty copy

- Most common model: ***MESI*** (Modified, Exclusive, Shared, Invalid)

MESI

- **Assumptions**
  - **Data ownership model. Only one cache can have dirty data**
  - **Writes are typically not broadcasted (i.e. *data* is not transmitted). Cause data invalidation in other caches**
- **Bus activity**
  - **Master: "Intend to cache" (inquiry cycle)**
  - **Slave: "I have a copy" (Hit signal)**
    "I have a modified copy" (HitM signal)
- **Each cache line can be in one of the following states:**
  - **Modified: line resides exclusively in this cache only**
    Content is modified relative to memory
  - **Exclusive: line resides exclusively in this cache only**
    Content is same as memory
  - **Shared: line resides in this cache but may be shared with other**
    Content is same as memory
  - **Invalid: Line contains no valid memory copy**
- **The idea: modified/exclusive lines are "owned" by the cache**
  They can be changed w/o telling other caches
- **Attempt to access a non-owned line should be broadcasted**
  - **If owned by another: the owner should update the world and give up ownership**

MESI Cache Line States

In the data cache, a cache protocol known as MESI maintains consistency with caches of other processors and with an external cache. The data cache has two status bits per tag; so, each line can be in one of the states defined in Table 18. The state of a cache line can change as the result of either internal or external activity related to that line. In general, the operation of the MESI protocol is transparent to programs.

Cache Line State:	M Modified	E Exclusive	S Shared	I Invalid
This cache line is valid?	Yes	Yes	Yes	No
The memory copy is...	...out of date	...valid	...valid	—
Copies exist in caches of other processors?	No	No	Maybe	Maybe
A write to this line ...	...does not go to bus	...does not go to bus	...goes to bus and updates cache	...goes directly to bus

Table 18-1. MESI Cache Line States

MESI

Action	Current State	Next State	Bus Activity
Read	M	M	None (Read Hit)
Read	E	E	None (Read Hit)
Read	S	S	None (Read Hit)
Read	I	S / E	Line Fill (Read Miss) - Send Inq (intend to read) to other - Bring data - Implementation may decide if S / E *
Write	M	M	None (Write Hit)
Write	E	M	None (Write Hit)
Write	S	S / E	Write Through (Write Hit) - Send INV signal to other caches
Write	I	I	Write Through (Write Miss) - Send INV signal - Wait for line to be written by other - If write allocate then “read for ownership” (to E stage)

* e.g., S may be better if line exist in another cache, otherwise E

MESI (Cont)

Action	Current State	Next State	Bus Activity
Snoop Read	M	S	Write Back - Send Hit & HitM
Snoop Read	E	S	- Send Hit
Snoop Read	S	S	- Send Hit
Snoop Read	I	I	Do nothing
Snoop Write	M	I	Write Back - Send Hit/HitM - Invalidate owned copy
Snoop Write	E	I	Invalidate owned copy
Snoop Write	S	I	Invalidate owned copy
Snoop Write	I	I	Invalidate owned copy

MESI (Cont)

Variations (*not* Pentium):

- **Ownership Exchange**
  - ♦ **Allows Modified line to be exchanged between caches without write-back**
- **Write Allocate**
  - ♦ **Write miss causes line fill ("read for ownership"), followed as write hit to E line.**
  - \+ **Higher performance if line is used later**
  - – **Reduced if line is not used (e.g. zeroing big array)**
  - – **Must identify non-cacheable memory (e.g., video RAM)**
- **Handling 2 level caches is more complex. See later...**

X86 Caches

Processor	L1	L2	Comments
386	---	32K	82385
486	8K I+D	?	
Pentium®	8K I + 8K D	256K/512K	82491/82496
Pentium® MMX	16K I + 16K D	256K/512K	82491/82496
Pentium Pro®	8K I + 8K D	256K-1024K	On package
Pentium-II®	16K I + 16K D	512K	On SECC

- Pentium® internal caches
  - Data: 8K, 128 sets, 32b line, 2 Way, LRU, Write back/MESI (2 bits)
  - Inst: 8K, 128 sets, 32b line, 2 Way, LRU, I/V bits (1 bit)
- 3 port tags, 1 port data, banked cache
  - Data: 1 snoop, 2 U/V pipes
  - Inst: 1 snoop, 2 prefetch

Processor-Initiated Read Cycles

Present State	Pin Activity	Next State	Description
M	n/a	M	Read hit; data is provided to processor core by cache. No bus cycle is generated.
E	n/a	E	Read hit; data is provided to processor core by cache. No bus cycle is generated.
S	n/a	S	Read hit; data is provided to the processor by the cache. No bus cycle is generated.
I	CACHE# low AND KEN# low AND WB/WT# high AND PWT low	E	Data item does not exist in cache (MISS). A bus cycle (read) will be generated by the Pentium ® processor. This state transition will happen if WB/WT# is sampled high with first BRDY# or NA#.
I	CACHE# low AND KEN# low AND (WB/WT# low OR PWT high)	S	Same as previous read miss case except that WB/WT# is sampled low with first BRDY# or NA#.
I	CACHE# high OR KEN# high	I	KEN# pin inactive; the line is not intended to be cached in the Pentium processor.

NOTE: *Locked accesses to the data cache will cause the accessed line to transition to the Invalid state

The transition from I to E or S states (based on WB/WT#) happens only if KEN# is sampled low with the first of BRDY# or NA#, and the cycle is transformed into a LINE FILL cycle. If KEN# is sampled high, the line is not cached and remains in the I state.

Table 2-4. Data Cache State Transitions for UNLOCKED Pentium® Processor Initiated Read Cycles*

Processor-Initiated Write Cycles

Present State	Pin Activity	Next State	Description
M	n/a	M	Write hit; update data cache. No bus cycle generated to update memory.
E	n/a	M	Write hit; update cache only. No bus cycle generated; line is now MODIFIED.
S	PWT low AND WB/WT# high	E	Write hit; data cache updated with write data item. A write-through cycle is generated on bus to update memory and/or invalidate contents of other caches. The state transition occurs after the writethrough cycle completes on the bus (with the last BRDY#).
S	PWT low AND WB/WT# low	S	Same as above case of write to S-state line except that WB/WT# is sampled low.
S	PWT high	S	Same as above cases of writes to S state lines except that this is a write hit to a line in a writethrough page; status of WB/WT# pin is ignored.
I	n/a	I	Write MISS; a writethrough cycle is generated on the bus to update external memory. No allocation done.

NOTE: Memory writes are buffered while I/O writes are not. There is no guarantee of synchronization between completion of memory writes on the bus and instruction execution after the write. A serializing instruction needs to be executed to synchronize writes with the next instruction if necessary.

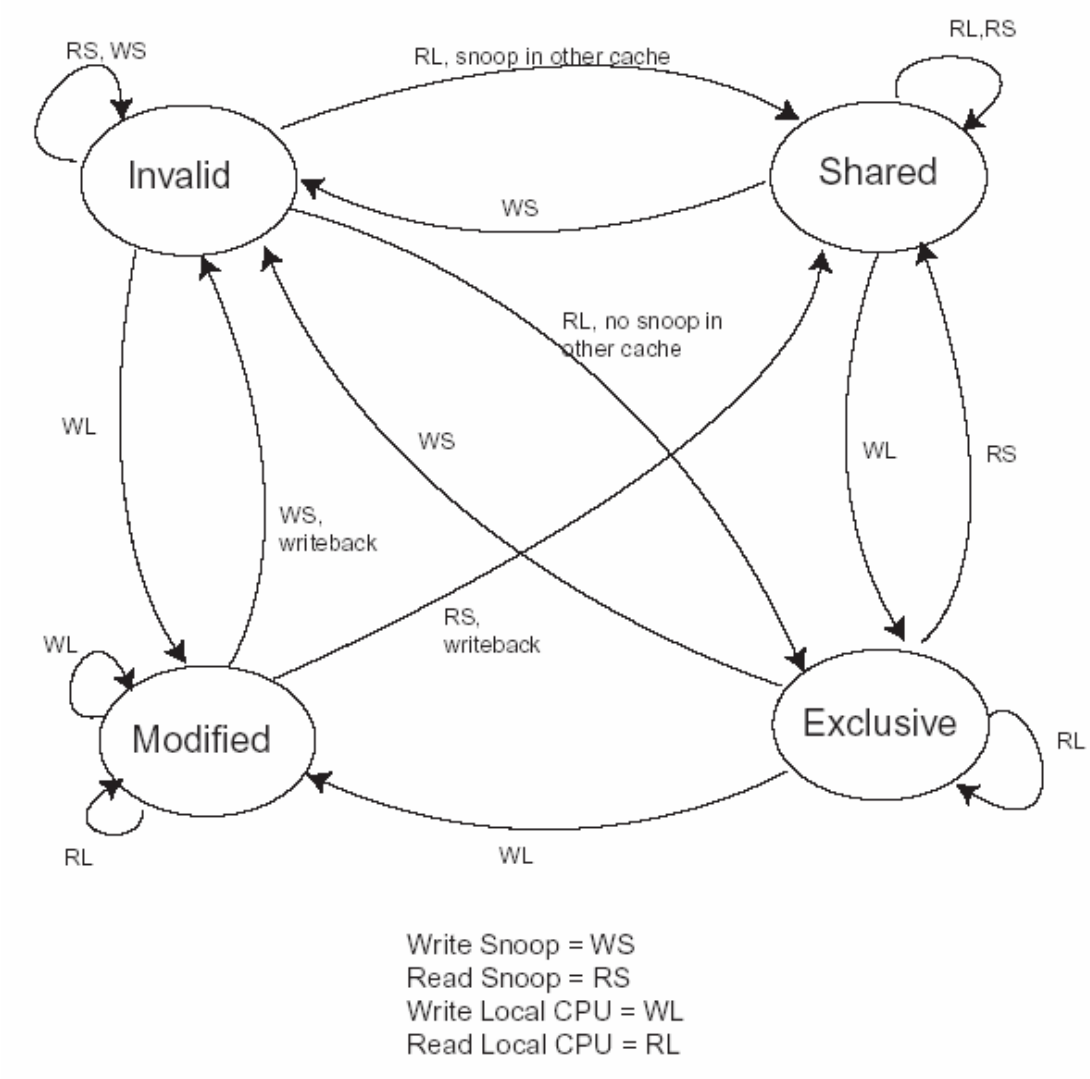
Table 2-5. Data Cache State Transitions for Pentium® Processor Initiated Write Cycles

Cache State Transitions During Inquire Cycles

Present State	Next State INV=1	Next State INV=0	Description
M	I	S	Snoop hit to a MODIFIED line indicated by HIT# and HITM# pins low. Pentium ® processor schedules the writing back of the modified line to memory.
E	I	S	Snoop hit indicated by HIT# pin low; no bus cycle generated.
S	I	S	Snoop hit indicated by HIT# pin low; no bus cycle generated.
I	I	I	Address not in cache; HIT# pin high.

Table 2-6. Cache State Transitions During Inquire Cycles

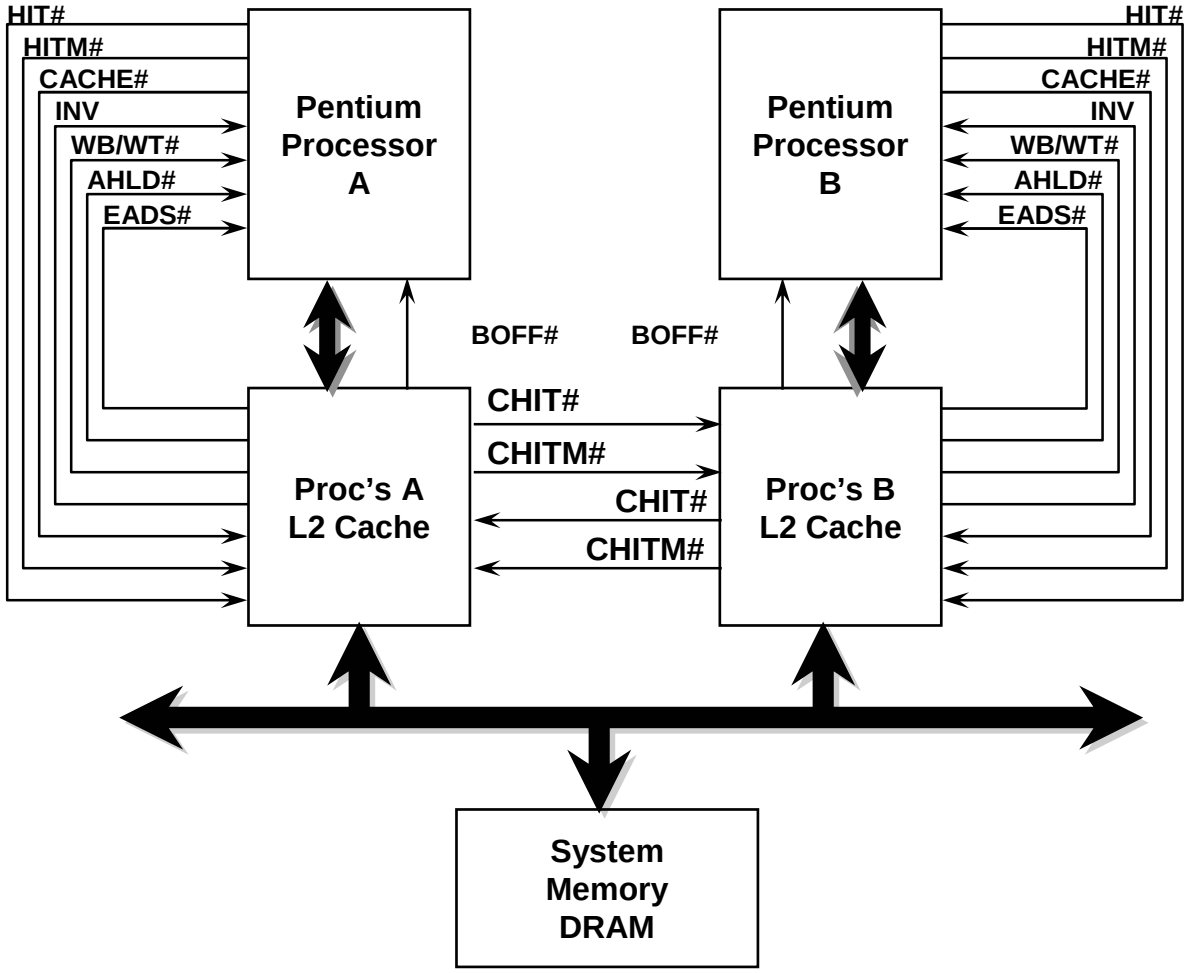
MESI Diagram



Cache Relevant Signals

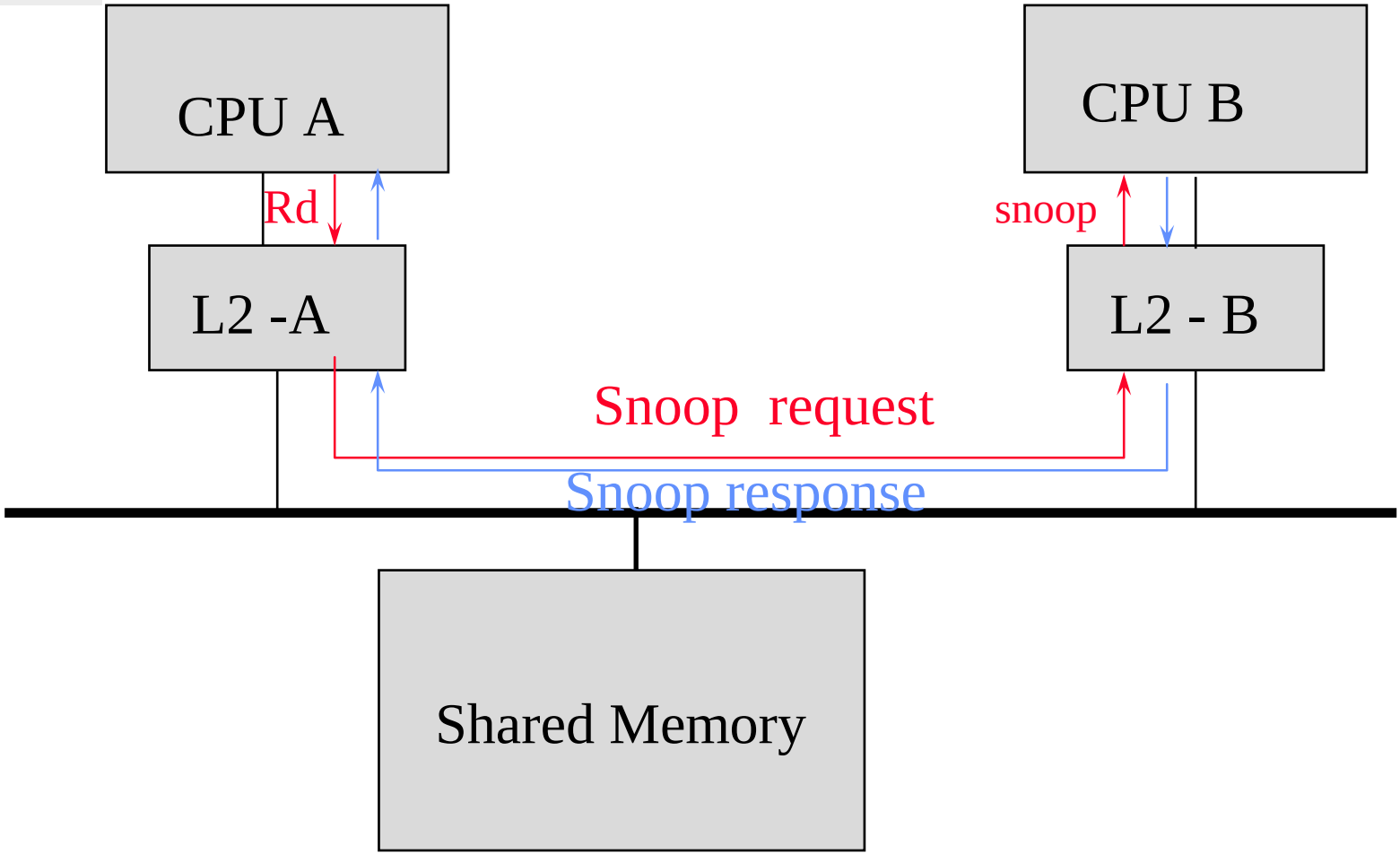
AHOLD	I	Address Hold
BOFF#	I	Start the aborted bus cycle from the beginning
CACHE#	O	Internal cacheability of the cycle
EADS#	I	Valid external address driven on address pins
EWBE#	I	External Write Buffers Empty (sync info)
FLUSH#	I	Force writeback of all modified lines
HIT#	O	Hit answer following an Inquire request
HITM#	O	Hit modify answer following an Inquire req.
INV	I	Inquire/Invalidate request
KEN#	I	Cache enable pin
PCD/PWT	O	Page Cache Disable / Page Write Through
WB/WT#	I	Write Back/Write Through cache line request

Pentium System - 2 Levels of Caches



Two Processors With Writeback Caches Using the MESI Model

L1/L2 organization
“Flow” of SNOOP cycles



L1/L2 communication

Case	L2	L1
1 First read, No partners	E	S
2 First write to line	M	E
3 Subsequent writes	M	M
4 Another processor read to same line	S	S

- Each line in L1 must be in L2 - the *inclusion property*
- L2 status represents the combined L1+L2 complex
- L1 status is relative to its L2 and always L1<=L2 (I,S,E,M=0,1,2,3)
  - Every CPU *M* or *E* line (dirty or exclusive) is *M* in L2.
  - Every L2 *M* line might be *I, S, E, M* in L1.
- All snoops into L2 are passed to L1
  - L2 does not really know L1 status!
  - But L2 should wait for L1 only if L2 is in M state

Process Synchronization

- Distinguish between *memory coherency* to *synchronization*
  - "If 2 processors increment the same cell, how to ensure X←X+2"
  - This is a synchronization problem, not a cache issue!
- Caches do not guarantee order!
  Semaphores should be used for exclusive access to X!
- In X86, XCHG can be used to implement a semaphore
  - XCHG has an implicit lock which makes it an atomic operation
  - Alternative: use explicit LOCK prefix on the instruction

```
get_x:
        MOV   EAX, 0
again:  XCHG  EAX, x_guard   / atomic - locked
        CMP   EAX, 0
        JZ    again          / until success
        .
        .
        MOV   EAX, X         / manipulate value
        INC   EAX            / ensure X is untouched
        MOV   X, EAX         / between operations.
        .
free_x: MOV   x_guard, 1
```