Memory System Design:

Memory Hierarchy

Virendra Singh

Associate Professor

Computer **A**rchitecture and **D**ependable **S**ystems **L**ab

Department of Electrical Engineering

Indian Institute of Technology Bombay

http://www.ee.iitb.ac.in/~viren/

E-mail: viren@ee.iitb.ac.in

EE-739: Processor Design

Lecture 19 (28 Feb 2013)

CADSL

Memory Hierarchy Basics

- Four Basic Questions
  - Where can a block be placed in the upper level?
    - Block Placement
  - How a block found if it is in the upper level?
    - Block Identification
  - Which block should be replaced on miss
    - Block Replacement
  - What happens on write
    - Write Strategy

Replacement

- Cache has finite size
  - What do we do when it is full?
- Analogy: desktop full?
  - Move books to bookshelf to make room
- Same idea:
  - Move blocks to next level of cache

Replacement

- How do we choose *victim*?
- Several policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used)
  - NMRU (not most recently used)
  - Pseudo-random (yes, really!)
- Pick victim within *set* where a = *associativity*
  - If a <= 2, LRU is cheap and easy (1 bit)
  - If a > 2, it gets harder
  - Pseudo-random works pretty well for caches

Write Policy

- Memory hierarchy
  - 2 or more copies of same block
    - Main memory and/or disk
    - Caches

- What to do on a write?
  - Eventually, all copies must be changed
  - Write must *propagate* to all levels

Write Policy

- Easiest policy: *write-through*
- Every write propagates directly through hierarchy
  - Write in L1, L2, memory, disk (?!?)
- Why is this a bad idea?
  - Very high bandwidth requirement
  - Remember, large memories are slow
- Popular in real systems only to the L2
  - Every write updates L1 and L2
  - Beyond L2, use *write-back* policy

Write Policy

- Most widely used: *write-back*
- Maintain *state* of each line in a cache
  - Invalid – not present in the cache
  - Clean – present, but not written (unmodified)
  - Dirty – present and written (modified)
- Store state in tag array, next to address tag
  - Mark dirty bit on a write
- On eviction, check dirty bit
  - If set, write back dirty line to next level
  - Called a *writeback* or *castout*

Write Policy

- Complications of write-back policy
  - Stale copies lower in the hierarchy
  - Must always check higher level for dirty copies before accessing copy in a lower level
- Not a big problem in uniprocessors
  - In multiprocessors: *the cache coherence problem*
- I/O devices that use DMA (direct memory access) can cause problems even in uniprocessors
  - Called coherent I/O
  - Must check caches for dirty copies before reading main memory

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss

Tag Array

Tag0	Tag1	LRU
		0
		0
		0
		0

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss

Tag Array

Tag0	Tag1	LRU
		0
		0
10		1
		0

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit

Tag Array

Tag0	Tag1	LRU
		0
		0
10		1
		0

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss

Tag Array

Tag0	Tag1	LRU
		0
		0
10		1
11		1

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss

Tag Array

Tag0	Tag1	LRU
10		1
		0
10		1
11		1

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss

Tag Array

Tag0	Tag1	LRU
10	11	0
		0
10		1
11		1

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss
Load 0x11	010001	0/0 (lru)	Miss/Evict

Tag Array

Tag0	Tag1	LRU
01	11	1
		0
10		1
11		1

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

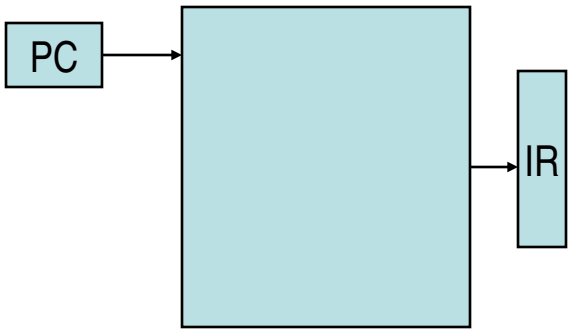
Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss
Load 0x11	010001	0/0 (lru)	Miss/Evict
Store 0x29	101001	2/0	Hit/Dirty

Tag Array

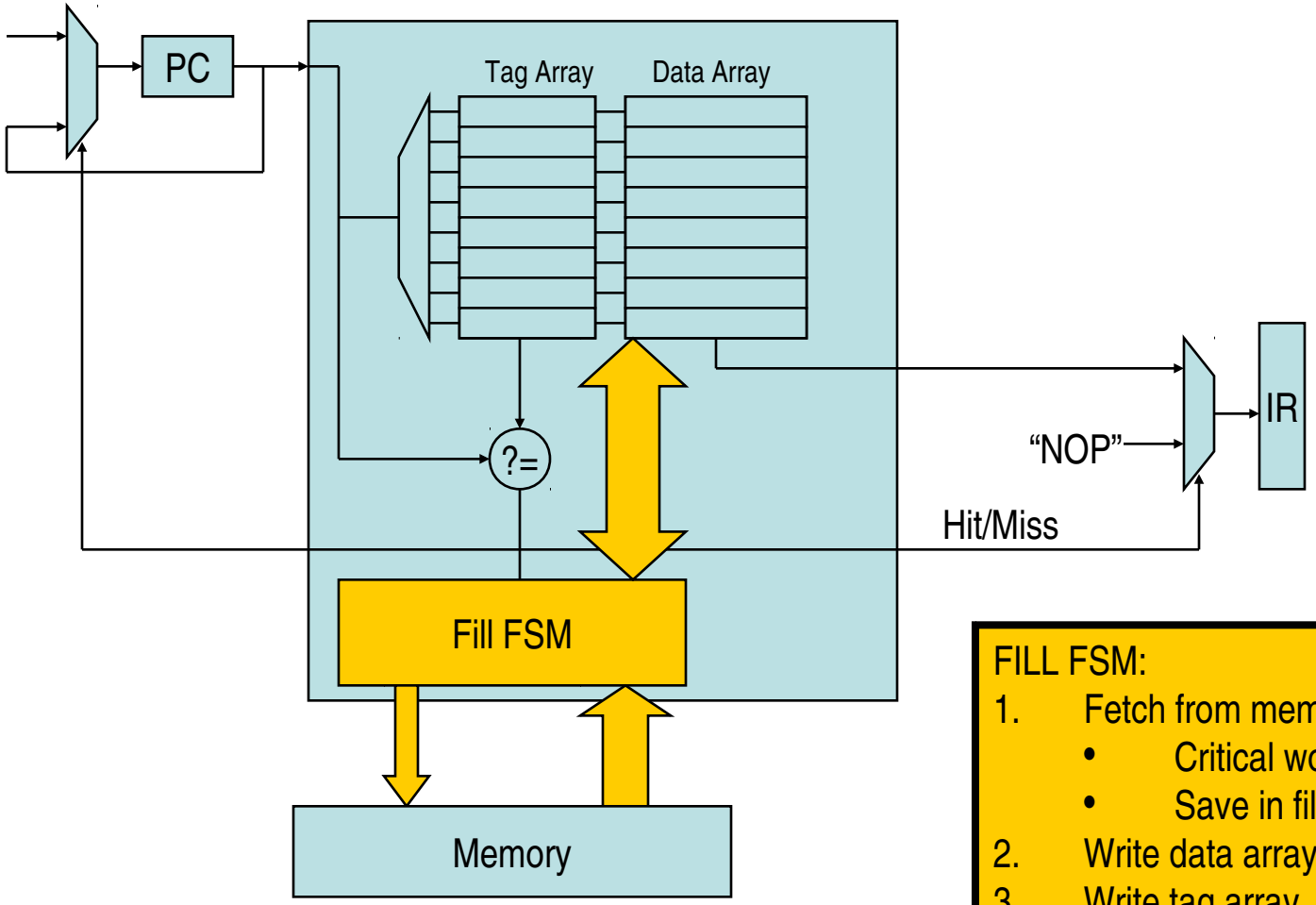
Tag0	Tag1	LRU
01	11	1
		0
10 d		1
11		1

Caches and Pipelining

- Instruction cache
  - No writes, so simpler
- Interface to pipeline:
  - Fetch address (from PC)
  - Supply instruction (to IR)
- What happens on a miss?
  - Stall pipeline; inject nop
  - Initiate cache fill from memory
  - Supply requested instruction, end stall condition



I-Caches and Pipelining



FILL FSM:

1. Fetch from memory
   - Critical word first
   - Save in fill buffer
2. Write data array
3. Write tag array
4. Miss condition ends

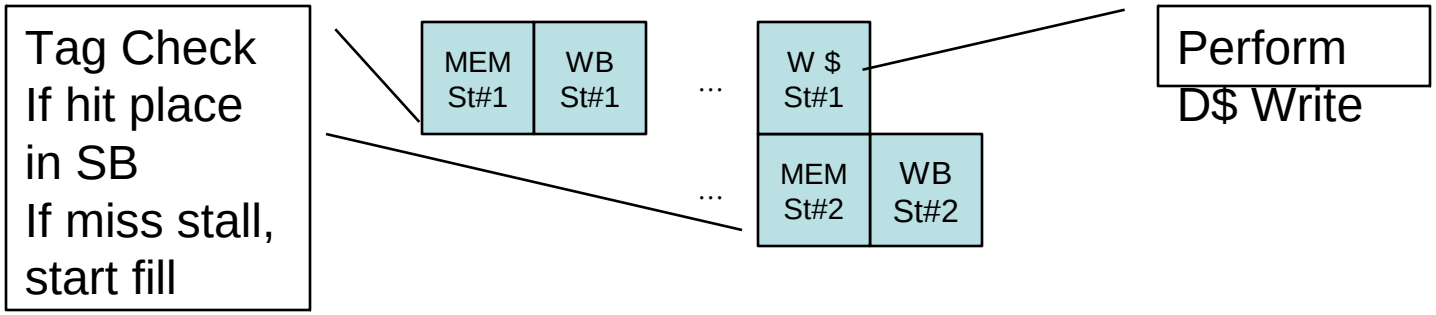
D-Caches and Pipelining

- Pipelining loads from cache
  - Hit/Miss signal from cache
  - Stalls pipeline or inject NOPs?
    - Hard to do in current real designs, since wires are too slow for global stall signals
  - Instead, treat more like branch misprediction
    - Cancel/flush pipeline
    - Restart when cache fill logic is done

D-Caches and Pipelining

- Stores more difficult
  - MEM stage:
    - Perform tag check
    - Only enable write on a hit
    - On a miss, must not write (data corruption)
  - Problem:
    - Must do tag check and data array access sequentially
    - This will hurt cycle time or force extra pipeline stage
    - Extra pipeline stage delays loads as well: IPC hit!

Solution: Pipelining Writes



- Store #1 performs tag check only in MEM stage
  - <value, address, cache way> placed in store buffer (SB)
- When store #2 reaches MEM stage
  - Store #1 writes to data cache
- In the meantime, must handle RAW to store buffer
  - Pending write in SB to address A
  - Newer loads must check SB for conflict
  - Stall/flush SB, or forward directly from SB
- Any load miss must also flush SB first
  - Otherwise SB D$ write may be to wrong line
- Can expand to >1 entry to overlap store misses