18-447 Lecture 16:
Cache Design in Context
(Uniprocessor)
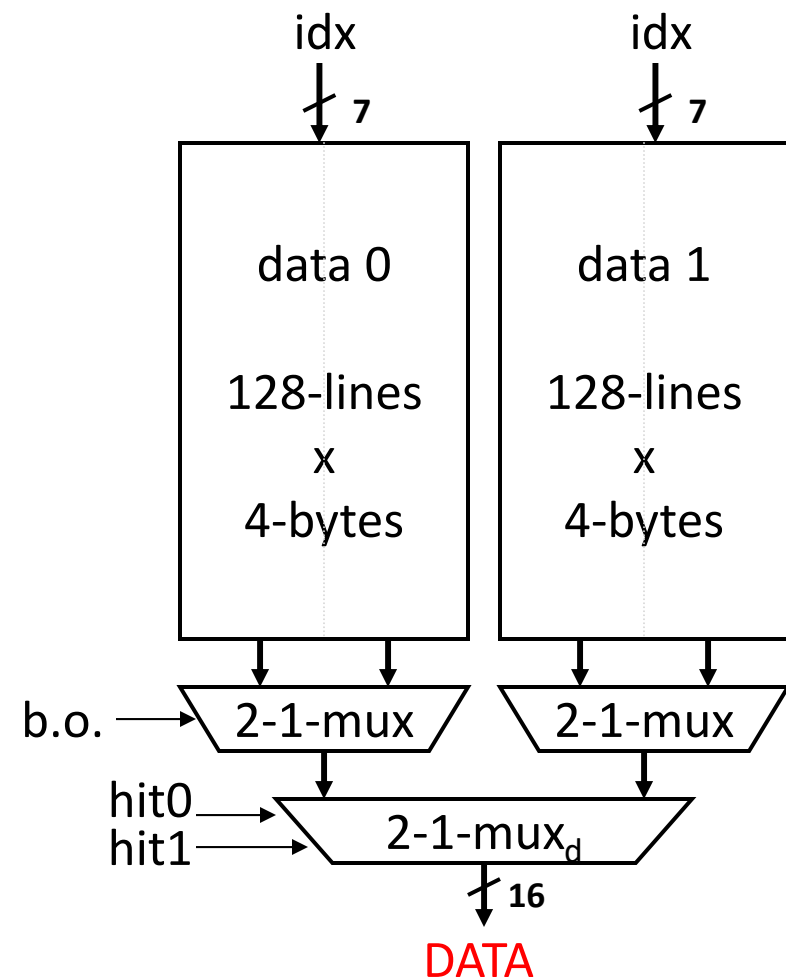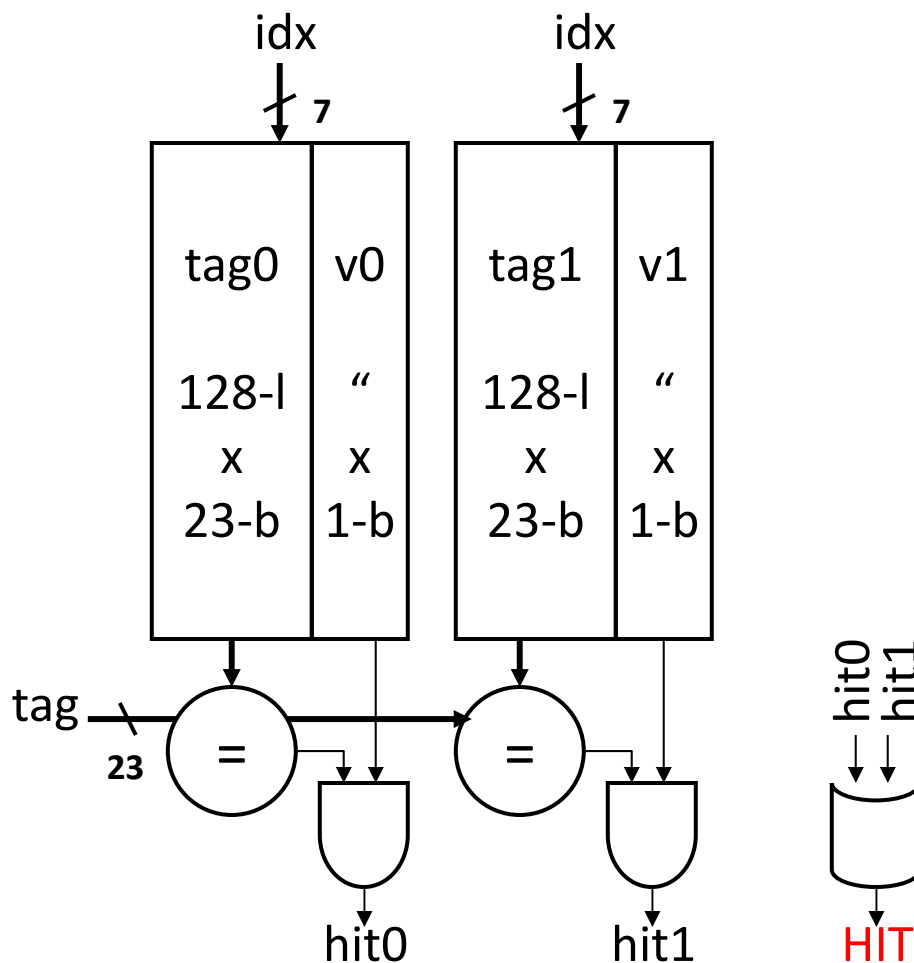
James C. Hoe

Department of ECE

Carnegie Mellon University
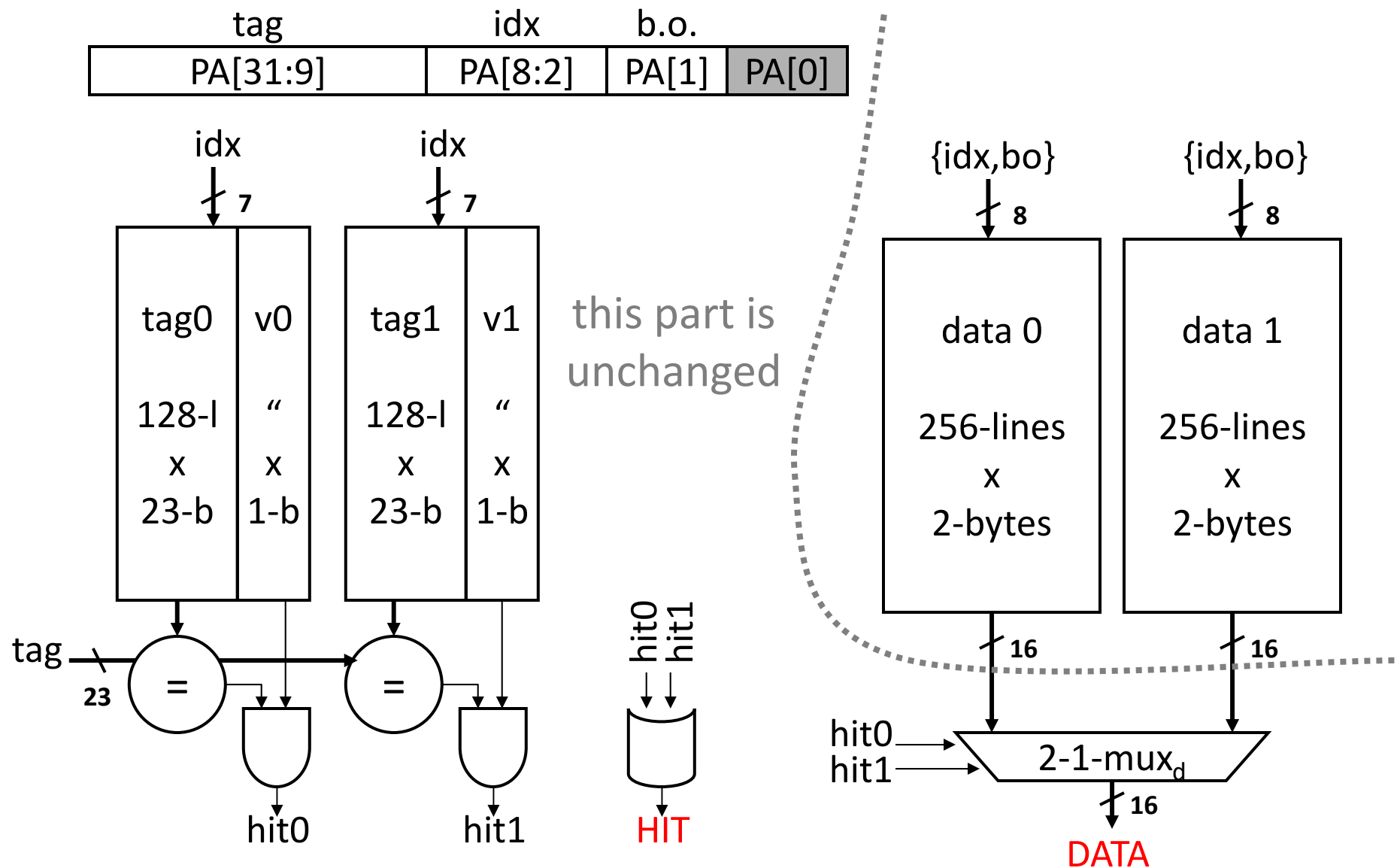
Housekeeping

- Your goal today
    - understand cache design and operation in context
    - focus on uniprocessor for now
- Notices
    - Lab 3, due Friday 4/9 noon
    - HW 4, due Monday 4/12 noon
    - Break Day Monday
- Readings
    - P&H Ch 5

M=2^{32}, a=2, C=1K, B=4, G=2: "textbook" solution

tag	idx	b.o.	
PA[31:9]	PA[8:2]	PA[1]	PA[0]

idx →7   idx →7   idx →7   idx →7

tag0	v0		tag1	v1		data 0	data 1
128-l x 23-b	" x 1-b		128-l x 23-b	" x 1-b		128-lines x 4-bytes	128-lines x 4-bytes

tag — 23 → =   =   hit0 / hit1

b.o. → 2-1-mux   2-1-mux

hit0 / hit1 → 2-1-mux$_d$ →16

hit0   hit1   **HIT**   **DATA**

Same cache parameters
but tune for "narrower" data <u>SRAM banks</u>

tag	idx	b.o.	
PA[31:9]	PA[8:2]	PA[1]	PA[0]

idx   idx   {idx,bo}   {idx,bo}

↓ 7   ↓ 7   ↓ 8   ↓ 8

tag0	v0		tag1	v1
128-l	"		128-l	"
x	x		x	x
23-b	1-b		23-b	1-b

this part is
unchanged

data 0		data 1
256-lines		256-lines
x		x
2-bytes		2-bytes

tag ⎯⎯

23   =   =

hit0 hit1

hit0   hit1   HIT

hit0 →
hit1 →   2-1-mux$_d$
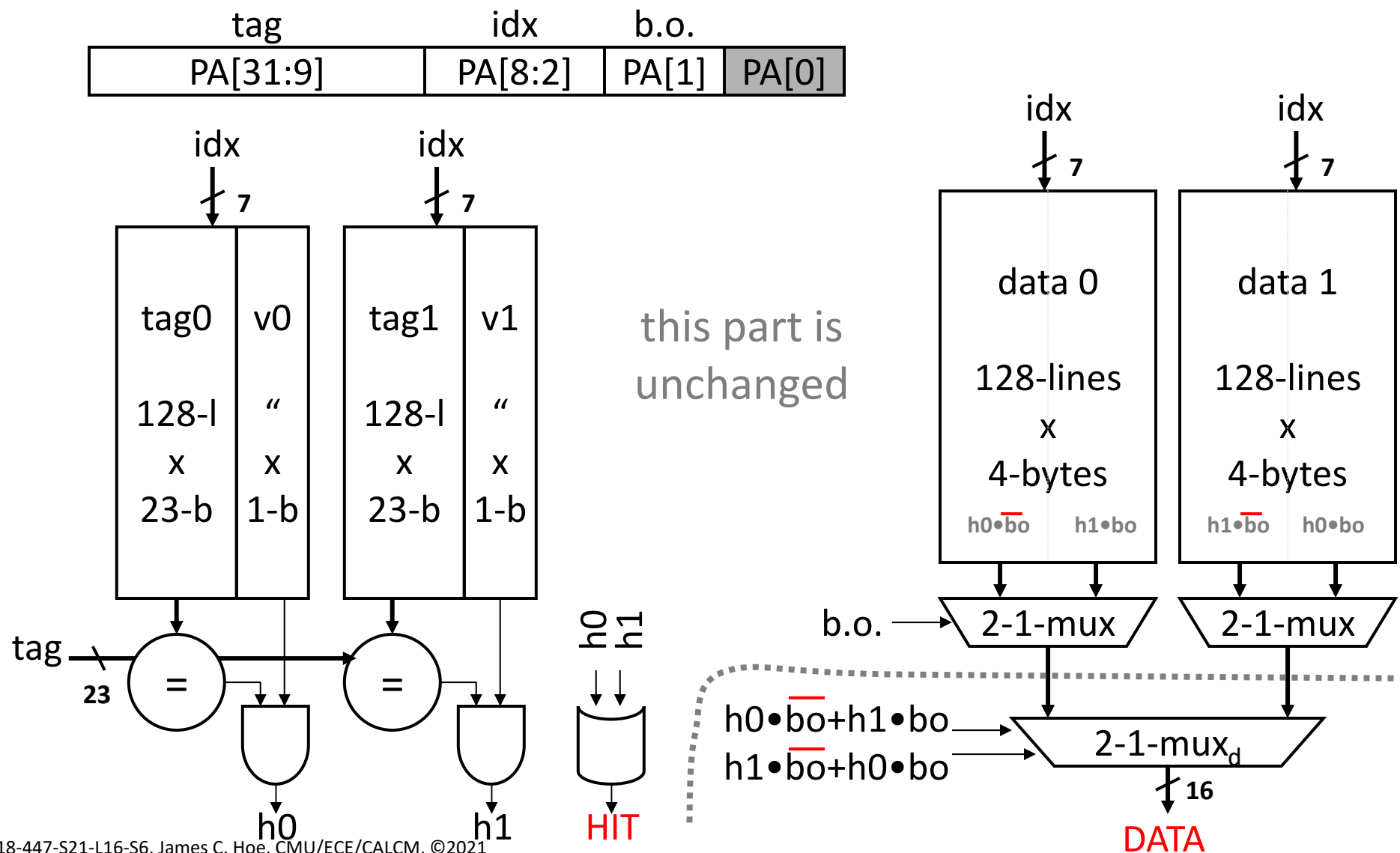
↓ 16

DATA

Can you play the same trick on the tag SRAMs?

Same cache parameters
but tune for "fatter" data SRAM banks

tag	idx	b.o.	
PA[31:9]	PA[8:2]	PA[1]	PA[0]

PA[8:3]   PA[8:3]

idx   idx

7   7

6   6

tag0	v0
128-l	"
x	x
23-b	1-b

tag1	v1
128-l	"
x	x
23-b	1-b

this part is unchanged

data 0

64-lines
x
8-bytes

data 1

64-lines
x
8-bytes

hit0 hit1

{PA[2],b.o.} → 4-1-mux   4-1-mux
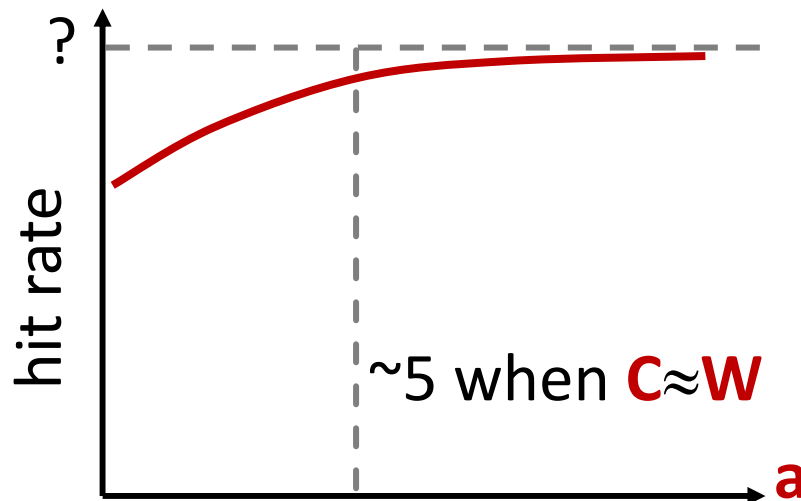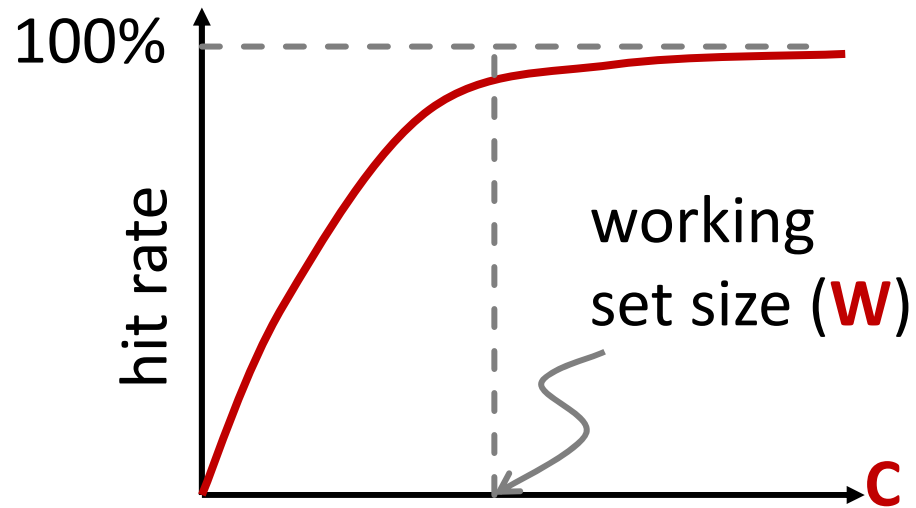
tag

23

= =

hit0
hit1 → 2-1-mux_d

16

hit0   hit1   HIT

DATA
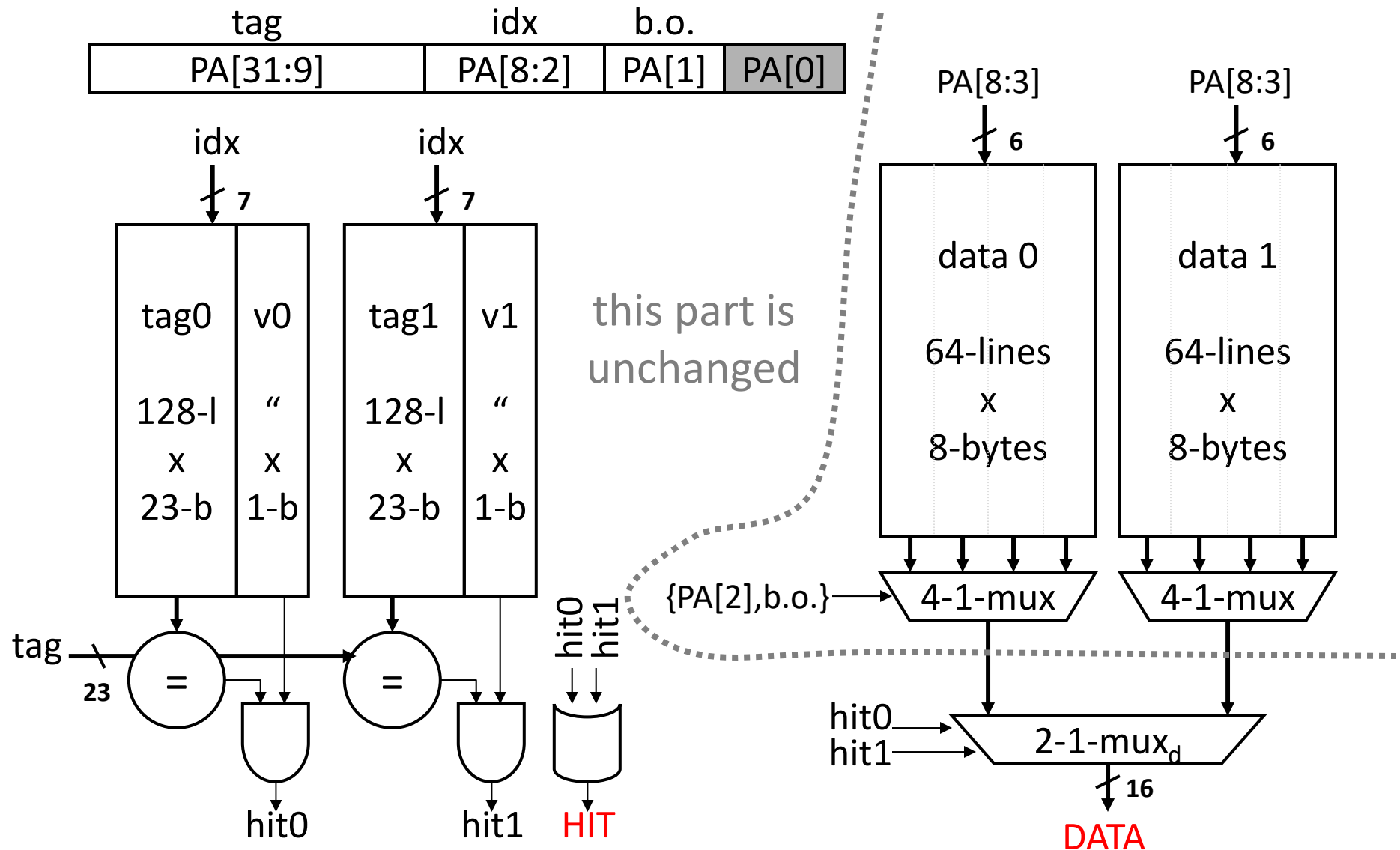
Can you play the same trick on the tag SRAMs?

Same cache parameters but each block frame is interleaved over 2 SRAM banks

tag	idx	b.o.	
PA[31:9]	PA[8:2]	PA[1]	PA[0]

idx

idx

idx

$\downarrow$ 7

$\downarrow$ 7

$\downarrow$ 7

$\downarrow$ 7

tag0	v0
128-l x 23-b	" x 1-b

tag1	v1
128-l x 23-b	" x 1-b

this part is unchanged

data 0
128-lines x 4-bytes
h0•$\overline{bo}$ h1•bo

data 1
128-lines x 4-bytes
h1•$\overline{bo}$ h0•bo

tag

=

=

23

h0 h1

b.o. → 2-1-mux   2-1-mux

h0•$\overline{bo}$+h1•bo →

h1•$\overline{bo}$+h0•bo →

2-1-mux$_d$

$\downarrow$ 16

h0

h1

HIT

DATA

aBC Rule of Thumb Cribsheet

100%

hit rate

working set size (**W**)

C

hit rate

?

hit rate

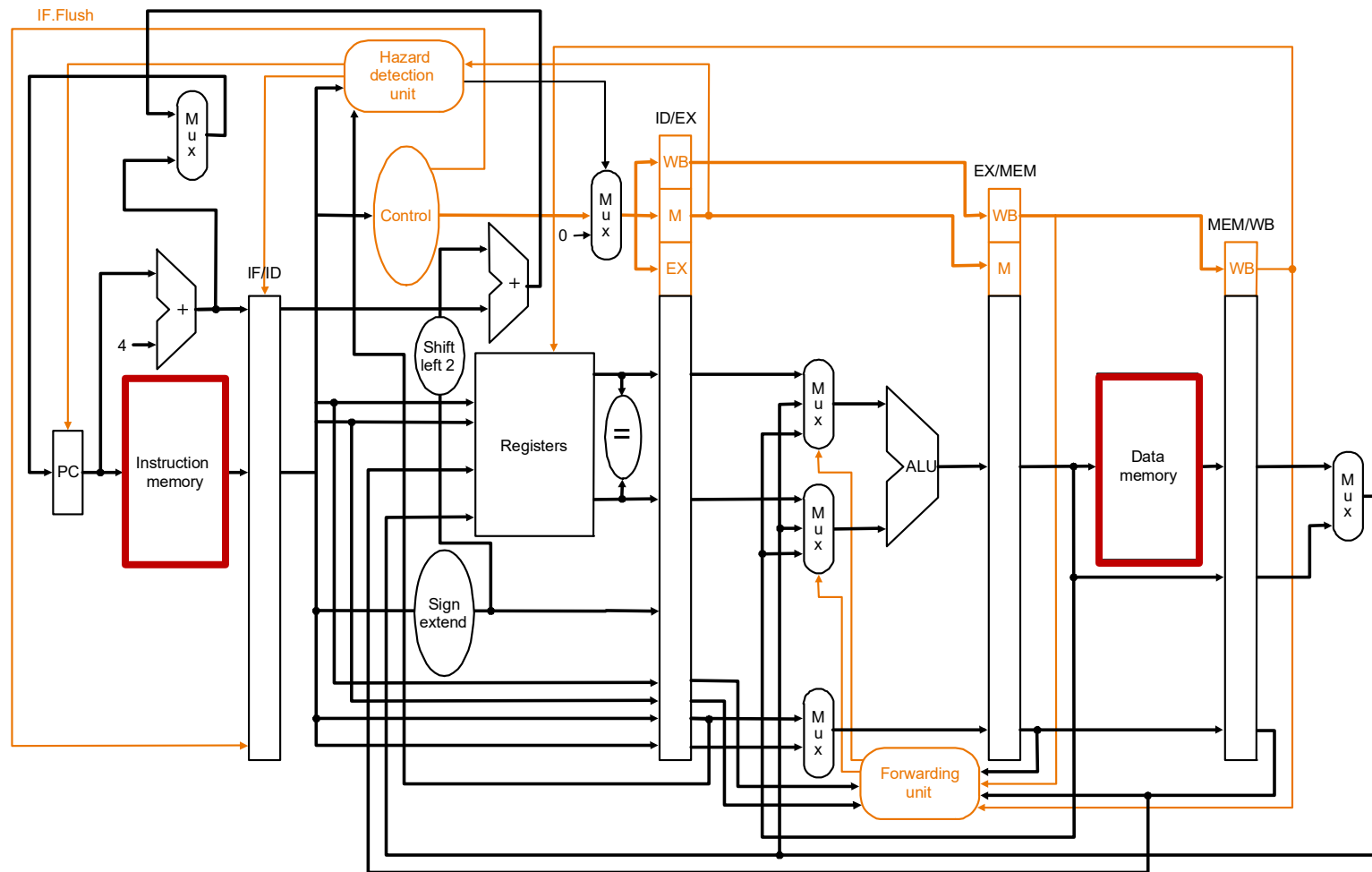~5 when **C≈W**

a

hit rate

B

available spatial locality
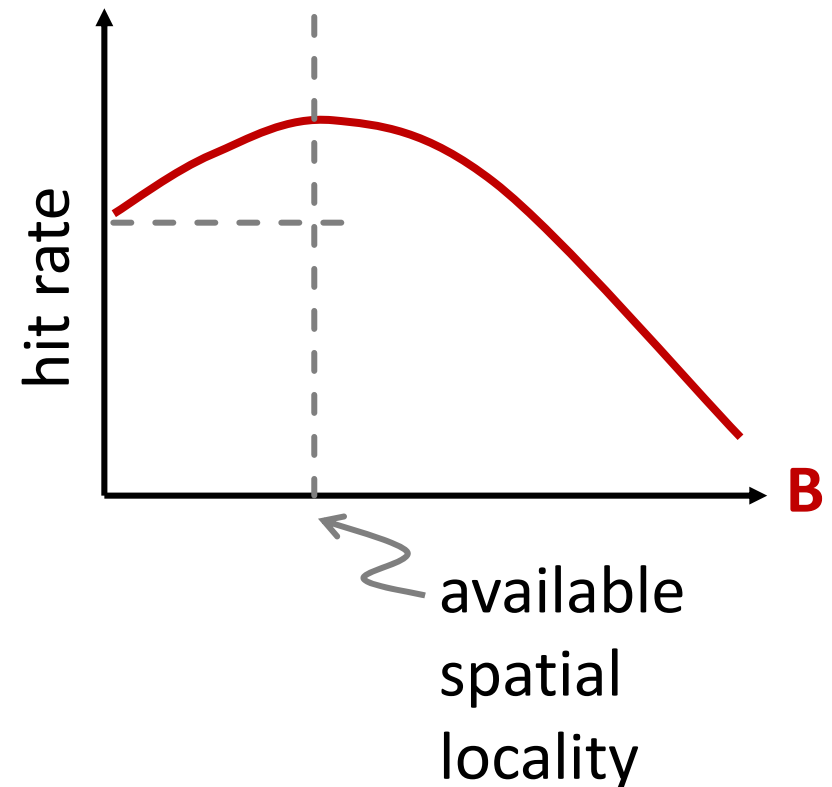
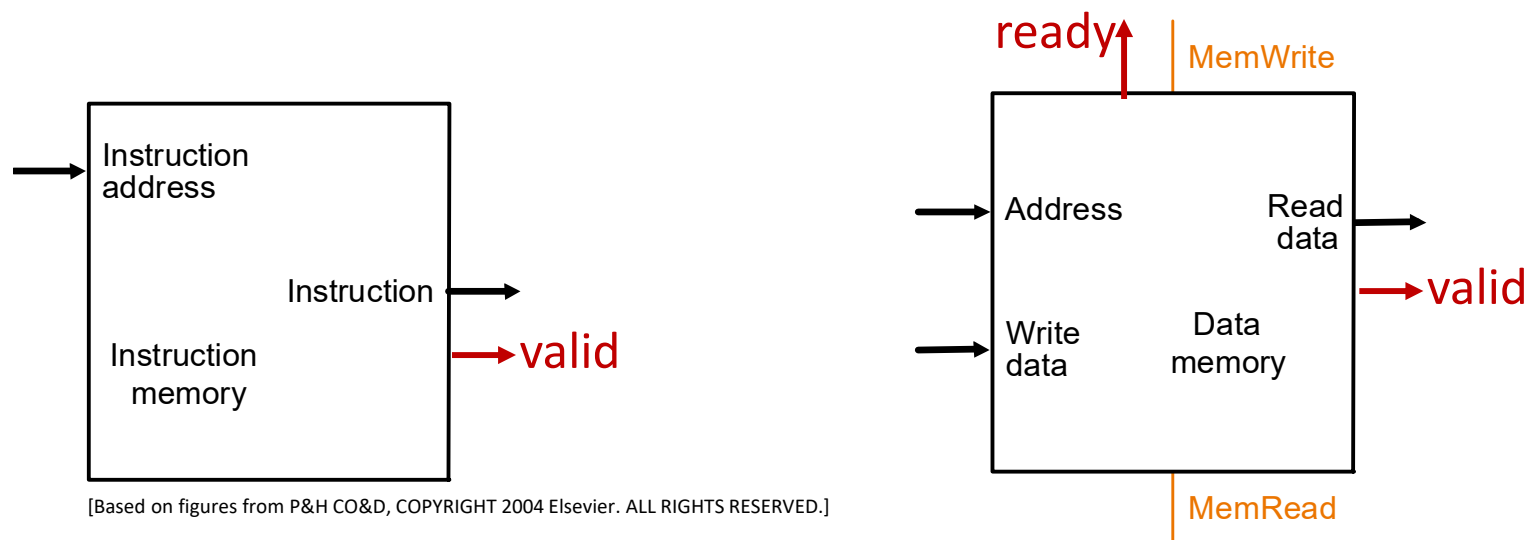For "typical" programs

The Cache and You
(simple, single core from Lab)

The Context

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Cache Interface for Dummies

ready ↑ MemWrite

Instruction address

Instruction

Instruction memory

→ valid

Address   Read data

Write data   Data memory

→ valid

MemRead

[Based on figures from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

- Like the magic memory
  - present address, R/W command, etc
  - result or update valid after a short/fixed latency
- Except occasionally, cache needs more time
  - will become valid/ready eventually
  - what to do with pipeline until then? Stall!!

Recall

Adding Caches to In-order Pipeline

- On I-fetch and LW assuming 1-cyc SRAM lookup
  - if hit, just like magic memory
  - if miss, stall pipeline until cache ready
- On SW also assuming 1-cycle SRAM lookup
  - if miss, stall pipeline until cache ready (must we??)
  - if hit, ???. . .
- For SW, need to check tag array to ascertain hit before committing to write data array
  - data array write happens in the next cycle
  - if SW is followed immediately by LW

$\Rightarrow$ **structural hazard** $\Rightarrow$ **stall**

Store Buffer

- Why stall when memory port is usually free?

- After tag array hit, buffer SW address and data until next free data array cycle (not needed by LW)

  – allow younger LW to execute (out-of-order)

  – must ensure SW target block not evicted

- Memory dependence and forwarding

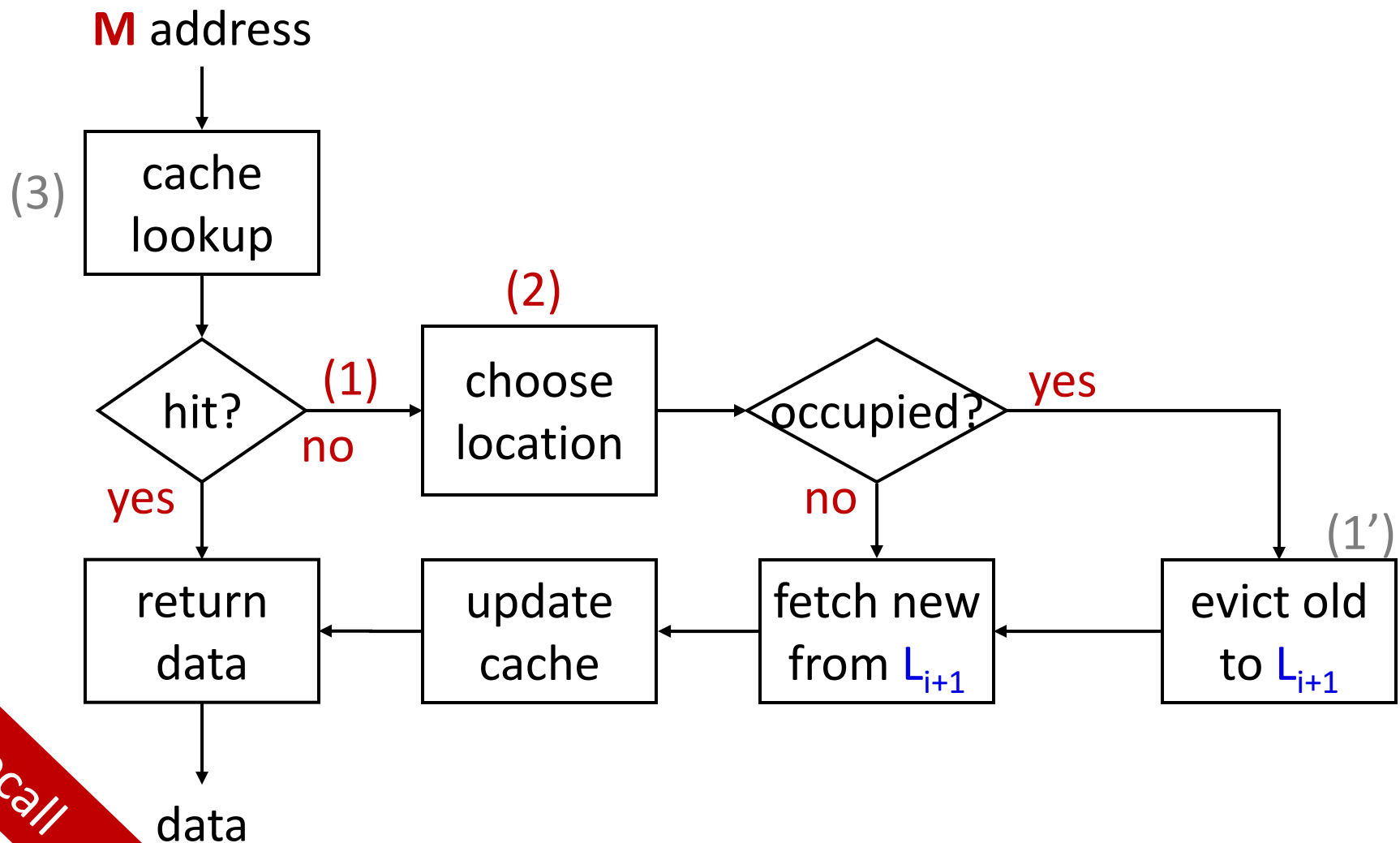 – younger LW must check against pending SW-addresses in store buffer (CAM) for RAW dependence
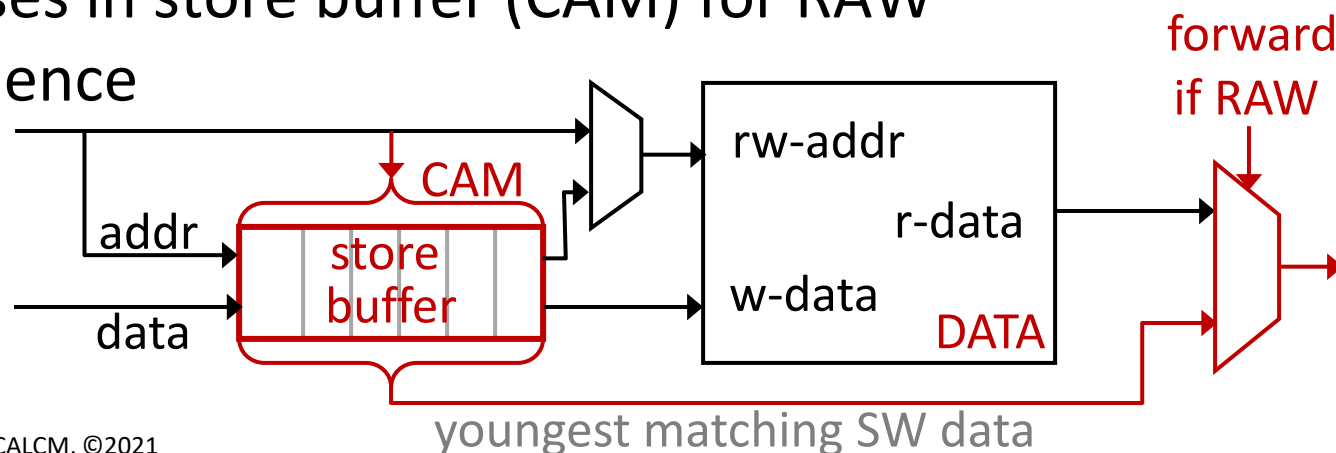
Must wait for a miss? (uniprocessor)

- In-order pipeline must stall for LW-miss
- Younger instructions can move ahead of SW-miss
  - except LW to same address; if so, stall or forward
  - additional SW-misses to same and different addr's can be "completed" from pipeline's view
- Modern out-of-order execution supports non-blocking miss handling for both LW and SW
  - too expensive to stall (CPU/memory speed gap)
  - significant complexity in
    - detecting and resolving memory dependencies
    - constructing precise exception state

Details and more details when building a cache for real

Basic "Cache Controller" (demand-driven version)

M address

(3) cache lookup

hit?

(1) no → choose location **(2)**

occupied? — yes

yes

no

return data ← update cache ← fetch new from L_{i+1} ← evict old to L_{i+1} (1')

data

Recall

Write-Through Cache

- On write-hit in L_i, should L_{i+1} be updated?

- If yes, write-through

  - simple management (discard on replacement)

  - external agents (DMA and other proc's) see up-to-date values in DRAM

- With write-through, on a write-miss, should a cache block be allocated in L_i (aka write-allocate)?

- Write-through to DRAM not viable today

  3.0GHz, IPC=2, 10% SW, ~8byte/SW $\Rightarrow$ ~5GB/sec

  L1 write-through to L2 still useful

Write-Back Cache

- Hold changes in L_i until block is displaced to L_{i+1}
  - on read or write miss, entire block is brought into L_i
  - LWs and SWs hit in L_i until replacement
  - on replacement, L_i copy written back out to L_{i+1}

    adds latency to load miss stall

- "Dirty" bit optimization
  - keep per-block status bit to track if a block has been modified since brought into L_i
  - if not dirty, no write-back on replacement

- What if a DMA device wants to read a DRAM location with a dirty cached copy?

    How to find out? How to access?

Write-Back Cache and DMA

- DRAM not always up-to-date if write-back

- DMA should see up-to-date value (aka, cache coherent)

- Option 1: SW flushes whole cache or specific blocks before programming DMA

- Option 2: cache monitors bus for external requests

  - ask request to a dirty location to "retry"

 - write out dirty copy before request is repeated

Proc

Cache

c[A]={drty, X'}

DMA

M[A]=X

DRAM
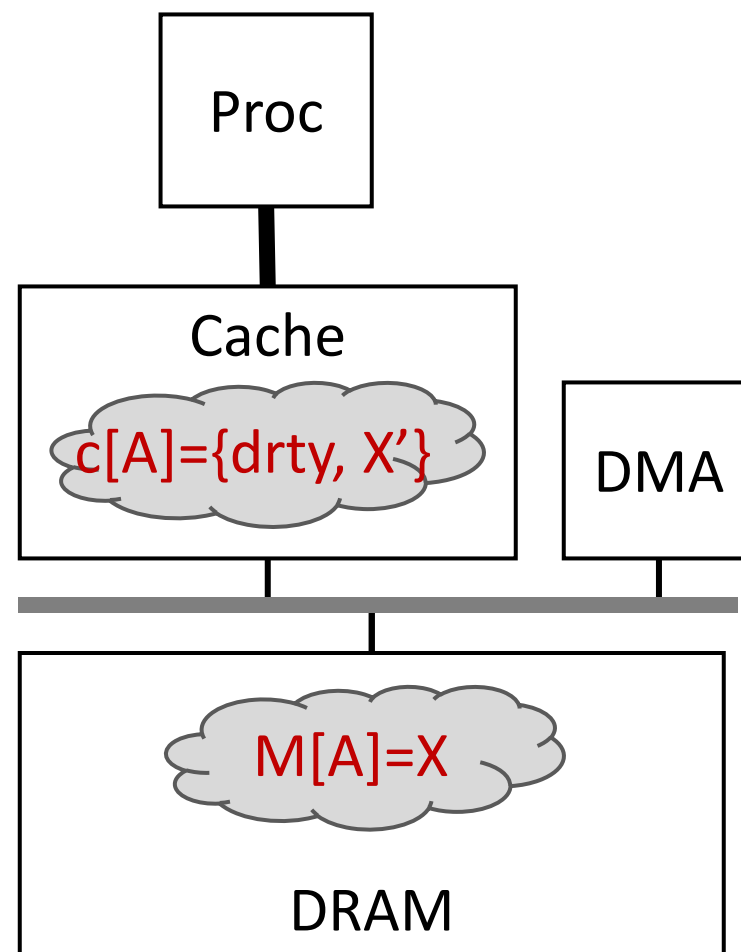
Idempotency and Side-effects

- Loading from real memory location M[A] should return most recent value stored to M[A]

  $\Rightarrow$ writing M[A] once is the same as writing M[A] with same value multiple times in a row

  $\Rightarrow$ reading M[A] multiple times returns same value
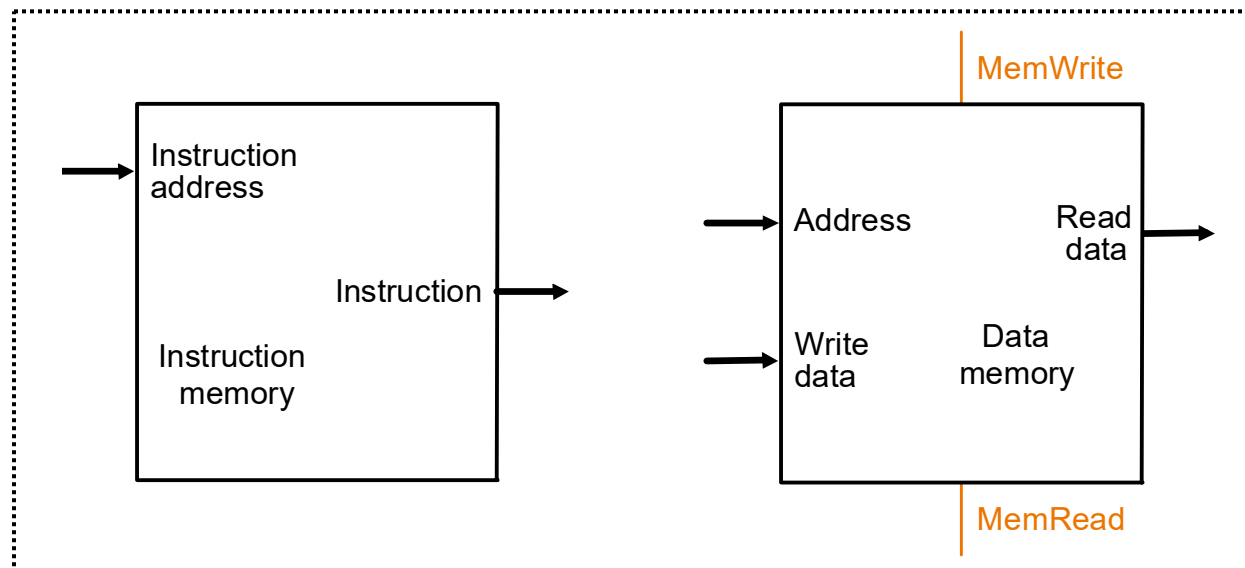
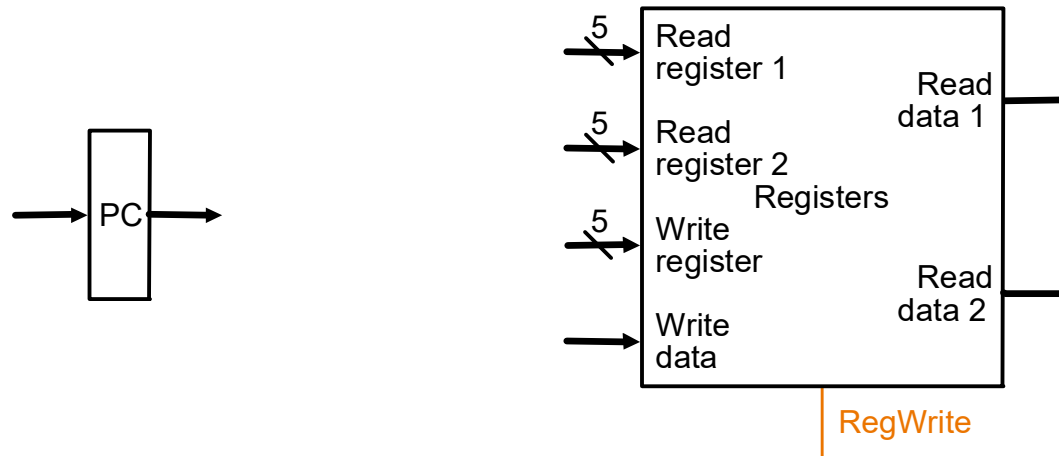 This is why memory caching works!!

- LW/SW to mmap locations can have side-effects

  – reading/writing mmap location can imply commands and other state changes

  – e.g., a mmap device that is a FIFO

    • SW to 0xffff0000 pushes value

    • LW from 0xffff0000 returns popped value

0xffff0000

FIFO
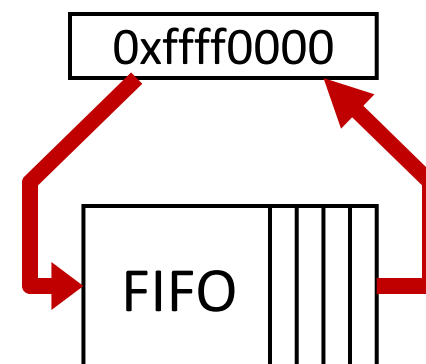
What happens if 0xffff0000 is cached?

Recall

Program Visible State
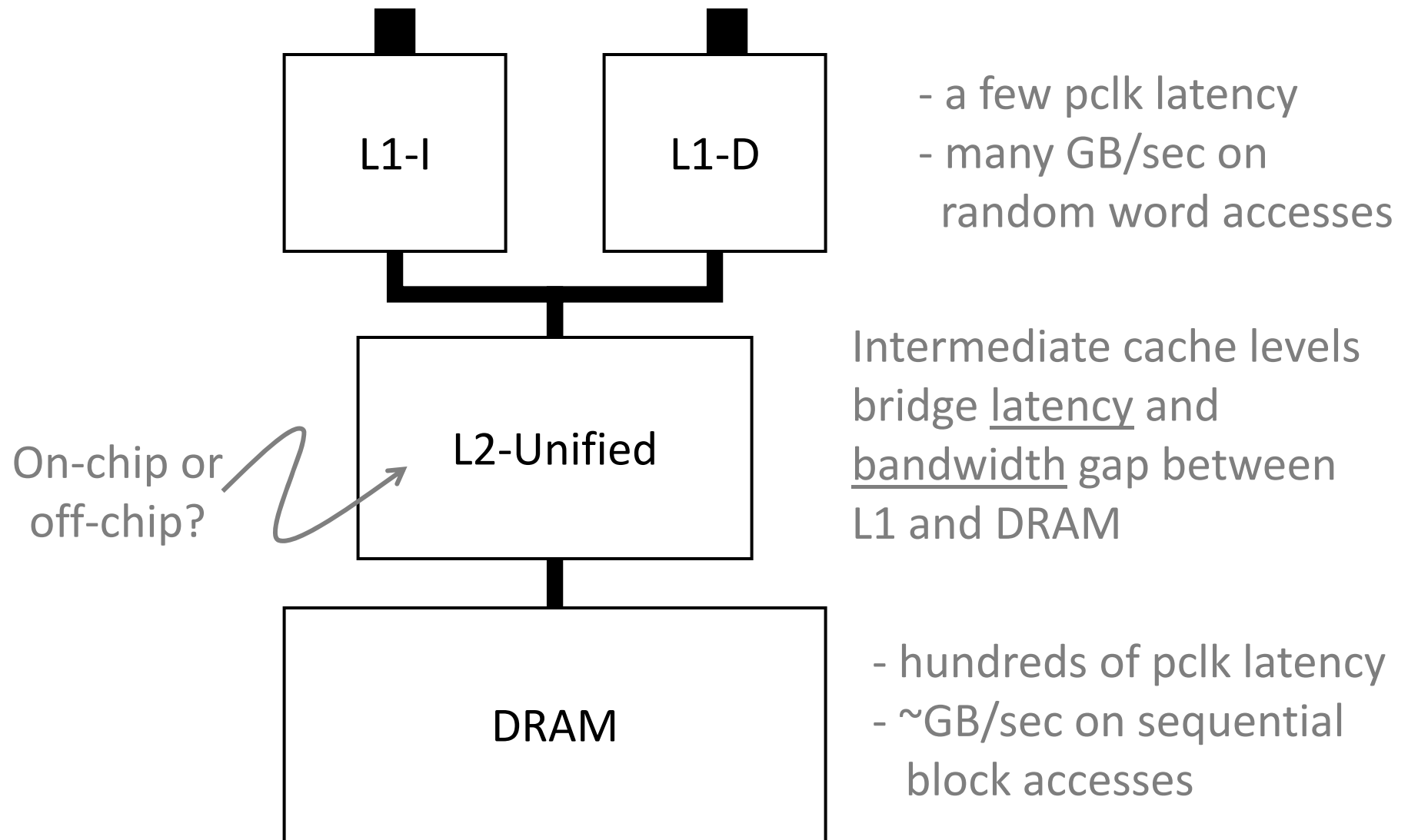## (aka Architectural State)

Recall

Harvard vs Princeton Architecture

- Historically
  - "Harvard" referred to Aiken's Mark series with separate instruction and data memory
  - "Princeton" referred to von Neumann's unified instruction and data memory
- Contemporary usage: split vs unified "caches"
- L1 I/D caches commonly split and asymmetrical
  - double bandwidth and no-cross pollution on disjoint I and D footprints
  - I-fetch smaller footprint, high-spatial locality and read-only $\Rightarrow$ I-cache smaller, simpler

    what about self-modifying code?
- L2 and L3 are unified for simplicity

Multi-Level Caches

- a few pclk latency
- many GB/sec on
  random word accesses

Intermediate cache levels bridge <u>latency</u> and <u>bandwidth</u> gap between L1 and DRAM

On-chip or off-chip?

- hundreds of pclk latency
- ~GB/sec on sequential
  block accesses

aBC of Multi-Level Cache Design

- Upper-level caches (L1)
  - small **C**: upper-bound by SRAM access time
  - smallish **B**: upper-bound by **C/B** effects
  - **a**: required to counter **C/B** effects
- Lower-level caches (L2, L3, etc.)
  - large **C**: upper-bound by chip area
  - large **B**: to reduce tag storage overhead
  - **a**: upper bound by complexity and speed
- New very large (10s MB) on-chip caches are distributed structures for multicores
  - same basic notions of ways and sets
  - but they don't look or operate anything like "textbook"

Modern Last-Level Cache (LLC)
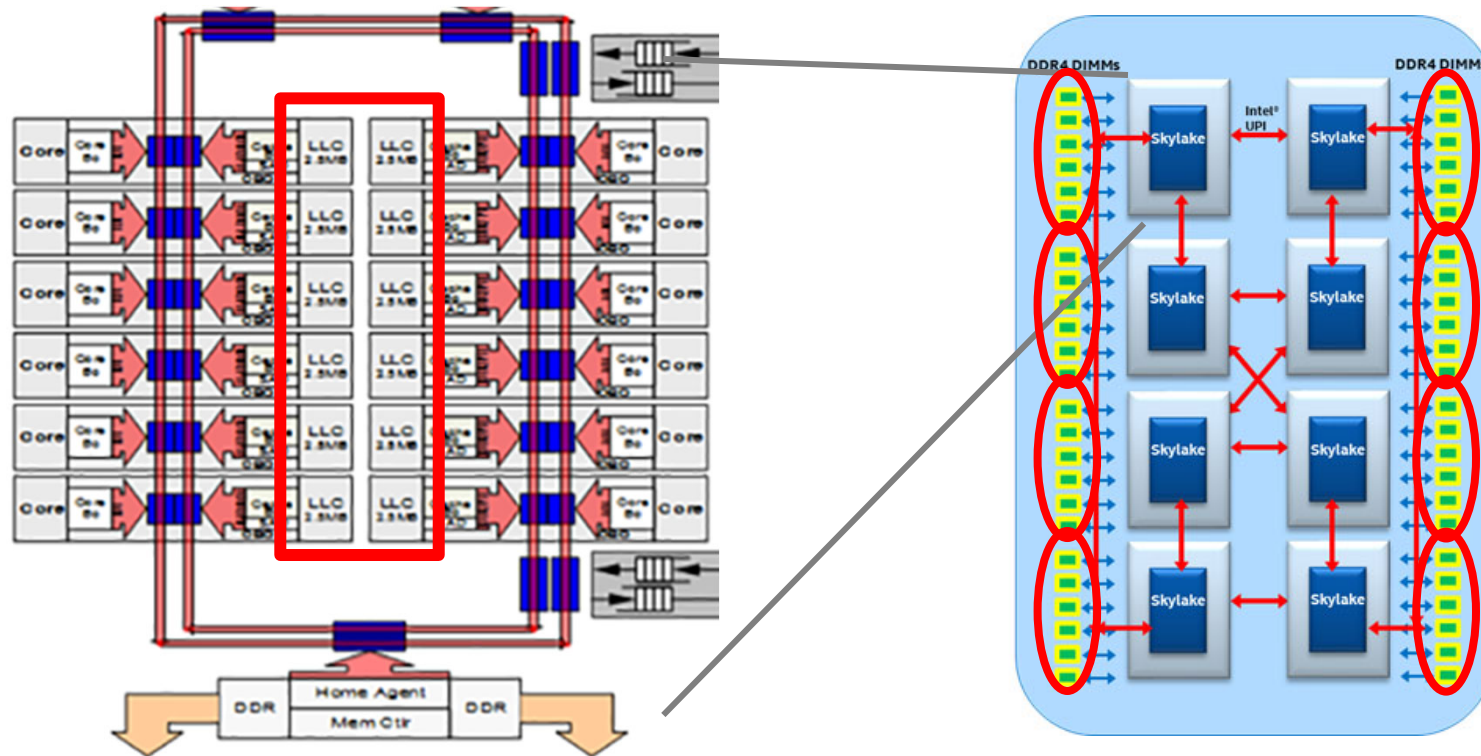
- Disaggregated, asynchronous structure; shared by all cores within a socket

- Hold, fast "coherent" copies of local and remote DRAM locations
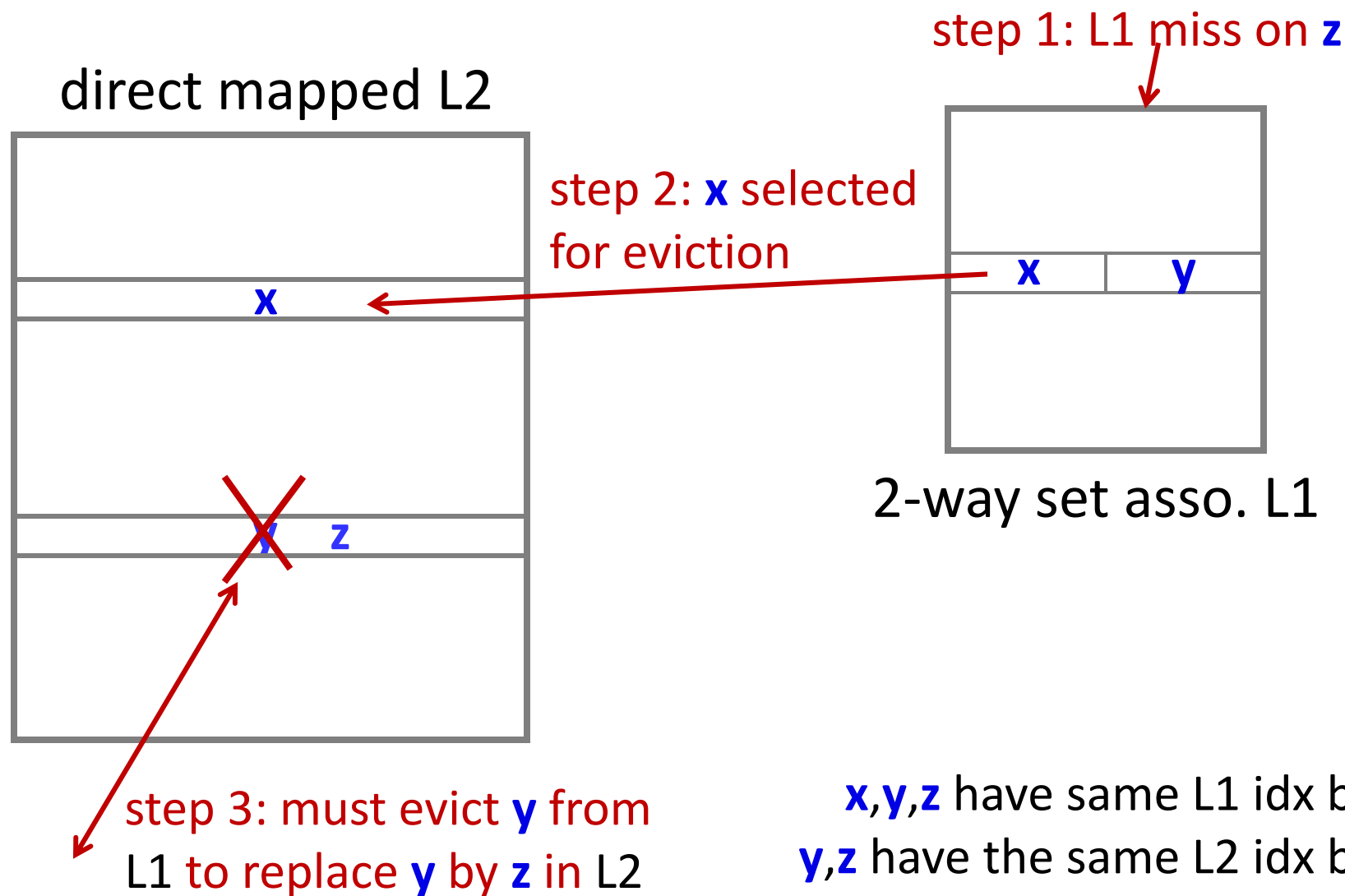
Departure from classic uniproc. hierarchy

Inclusion Principle

- Classically, L_i contents is always a subset of L_{i+1}
  - if an address is important enough to be in L_i, it must be important enough to be in L_{i+1}
  - external agents (DMA and other proc's) only have to check the lowest level to know if an address is cached—do not need to consume L1 bandwidth
- Inclusion no longer taken as a given
  - nontrivial to maintain if L_{i+1} has lower associativity
  - too much redundant capacity in multicore with many per-core L_i and shared L_{i+1}

Inclusion Violation Example

step 1: L1 miss on **z**

direct mapped L2

step 2: **x** selected
for eviction

x

y **z**

2-way set asso. L1

x   **y**

step 3: must evict **y** from
L1 to replace **y** by **z** in L2

x,**y**,**z** have same L1 idx bits
y,**z** have the same L2 idx bits
x,{**y**,**z**} have different L2 idx bits

Aside: Victim "Cache"

- High-associativity is an expensive solution to avoid conflicts by a few stray addresses

- Augment a low-associative main cache with a very small but fully associative victim cache

  - blocks evicted from main cache is first held in victim cache

  - if an evicted block is referenced again soon, it is returned to main cache

  - if an evicted block doesn't get referenced again, it will eventually be displaced from victim cache to next level

Plays a different role outside of standard memory hierarchy stacking

Aside: Software-Assists

- Separate "temporal" vs "non-temporal" hierarchy
  - exposed in the ISA (e.g., Intel IA64 below)
  - load and store instructions include **hints** about where to cache on a cache miss
  - **"hint"** only so implementation could support a subset or none of the levels and actions

temporal

L1   L2   L3

Main
Memory

non-temporal-L1

NT
L1

NT
L2

NT L3

non-temporal-L2

non-temporal-All
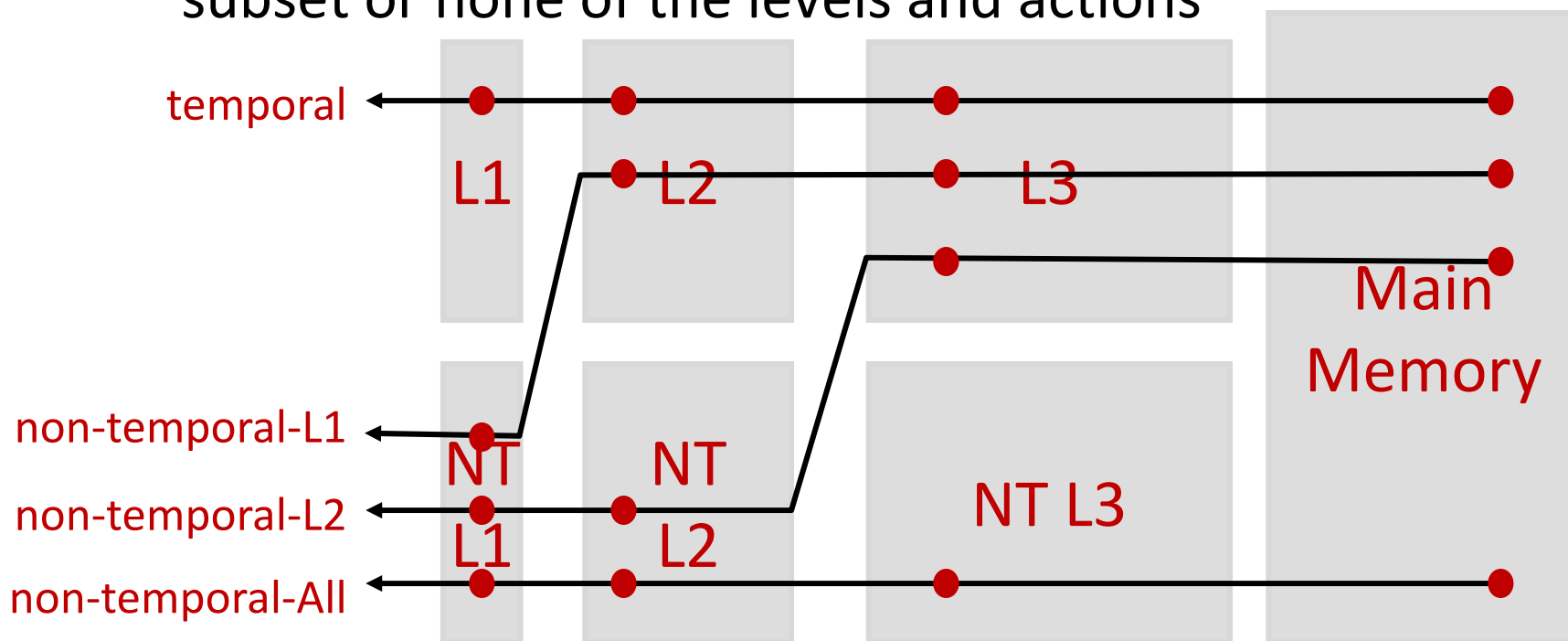
Test yourself

Optional Reading: "Measuring Cache and TLB Performance and Their Effect on Benchmark Run Times," Saavedra and Smith, 1995.

What cache is in your computer?

- How to figure out what cache configuration is in your computer
    - capacity (**C**), associativity (**a**), and block-size (**B**)
    - number of levels

- The presence or lack of a cache should not be detectable by functional behavior of software

- But you could tell if you measured execution time to infer the number of cache misses

Capacity Experiment: assume 2-power C

- For increasing **R**ange = 1,2,4,8,16,…
  - allocate a buffer of size **R**
  - repeatedly {read every byte in buffer in sequence}
  - measure average read time in steadystate
- Analysis
  - for small **R**≤**C**, expect all reads to hit
  - for large **R**>**C**, expect reads to miss and detect corresponding jump in memory access time
- If continuing to increase **R**, read time jumps again when buffer size spills out to next cache level

Warning: timing won't be perfect when you try this

Block Size Experiment: knowing C

- Allocate a buffer of size **R** >> **C**

- For increasing **S**=1,2,4,8….,
  - repeatedly {read every **S**'th byte in buffer in sequence}
  - measure average read time in steadystate

- Analysis
  - since **R**>>**C**, expect first read to a block to miss when revisiting a block
  - reads to same block in same round should hit
  - expect increasing average read time for increasing **S** until **S**≥**B** (no reuse in block)

Associativity Experiment: knowing C

- For increasing **R**, where **R** is a multiple of **C**
  - allocate a buffer of size **R**
  - repeatedly {read every **C**'th byte in buffer in sequence}
- Analysis
  - all **R**/**C** references map to the same set
  - for small **R** s.t. (**R**/**C**)≤**a**, expect all reads to hit
  - for large **R** s.t. (**R**/**C**)>**a**, expect some reads to miss since touching more addresses than ways

note: 100% cache miss if LRU is used

How to detect associativity for lower-level caches?

Know your cache

- What else can you tell?
    - write-back vs write-through/write-allocate
    - unified vs. split design
    - I-cache C, B, a
    - t_i
    - replacement policy of associative caches
- Same mental exercise is required to control cache use in performance tuning

Caveat: experiments may not predict behaviors exactly for modern CPUs with virtual memory, complex hierarchies, and prefetchers