

18-447 Lecture 16: Cache Design in Context (Uniprocessor)

James C. Hoe

Department of ECE

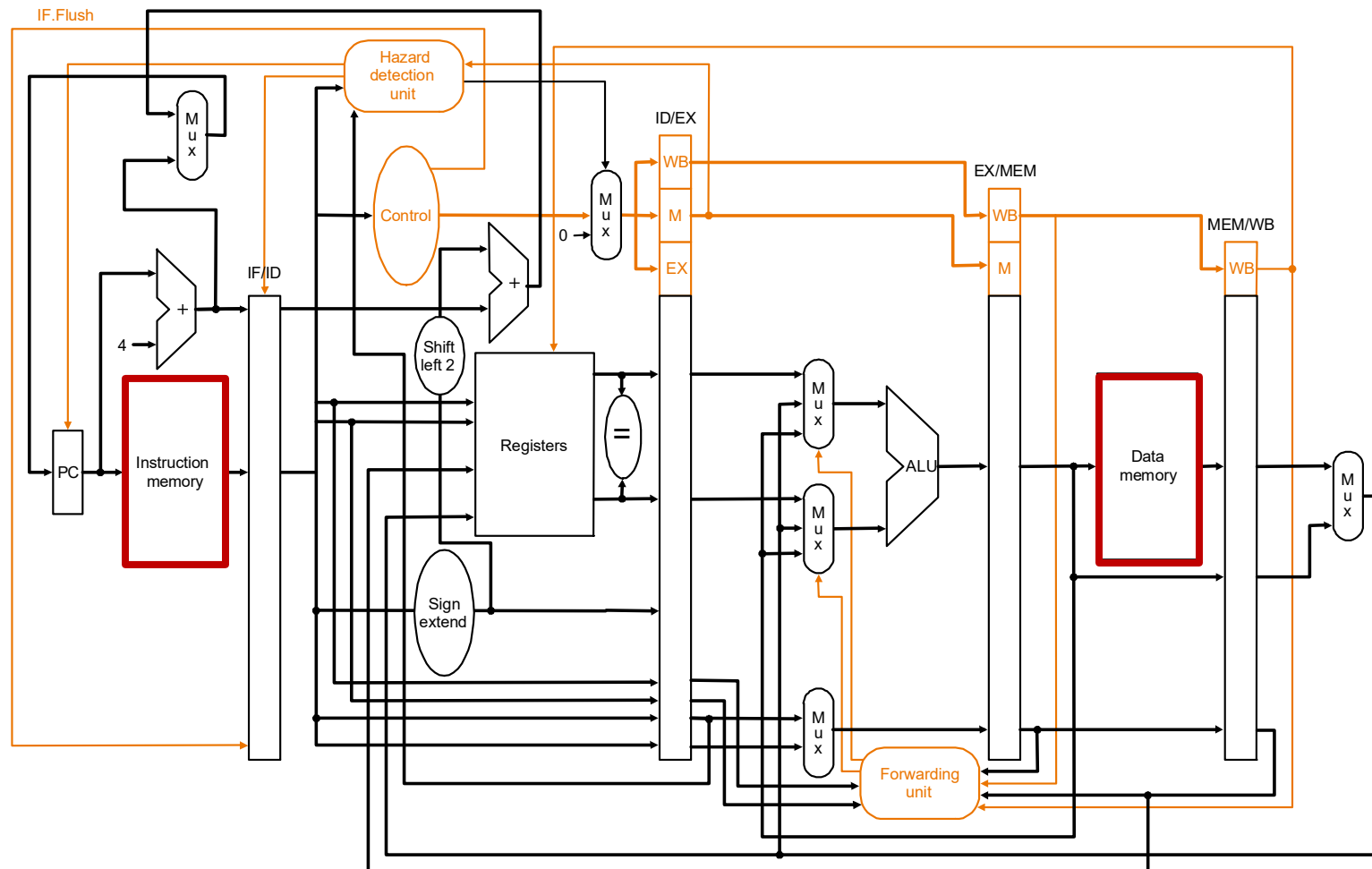
Carnegie Mellon University

Housekeeping

- Your goal today
 - understand cache design and operation in context
 - focus on uniprocessor for now
- Notices
 - Lab 3, due week of 3/30 (extended)
 - HW 4, due Friday 4/3
 - Midterm 2, online during class time, Mon, 4/6;
4-min dress rehearsal, Mon, 3/30
- Readings
 - P&H Ch 5

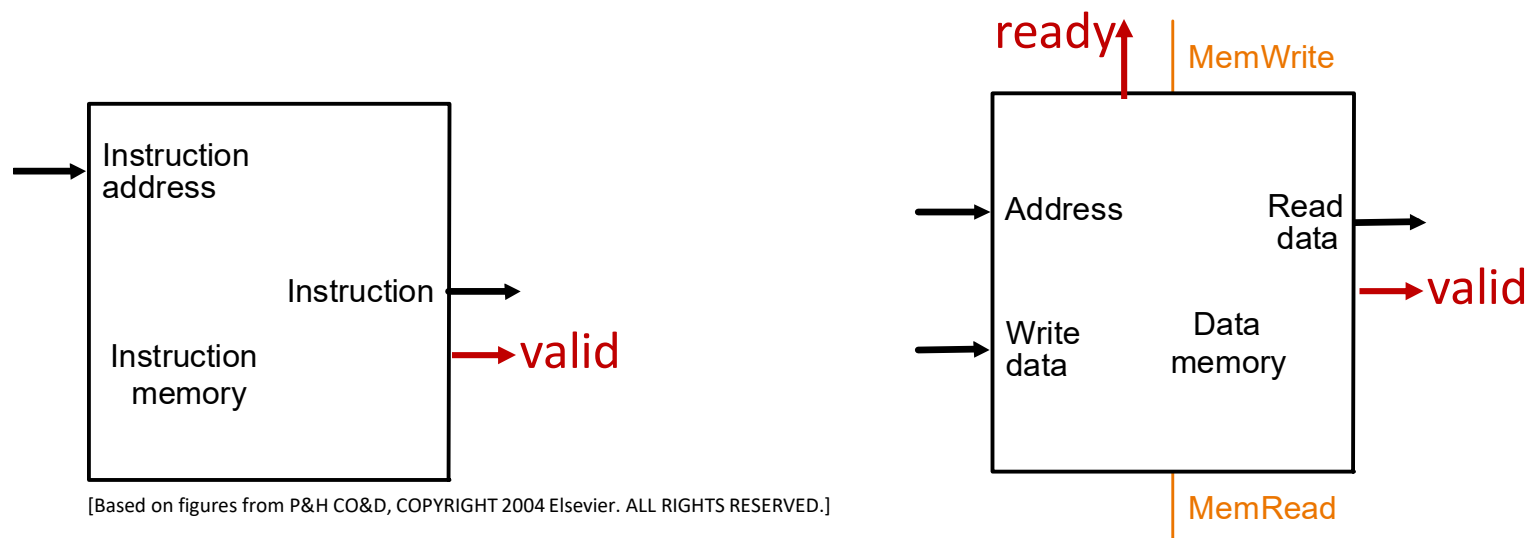
The Cache and You **(simple, single core from Lab)**

The Context



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

Cache Interface for Dummies



- 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

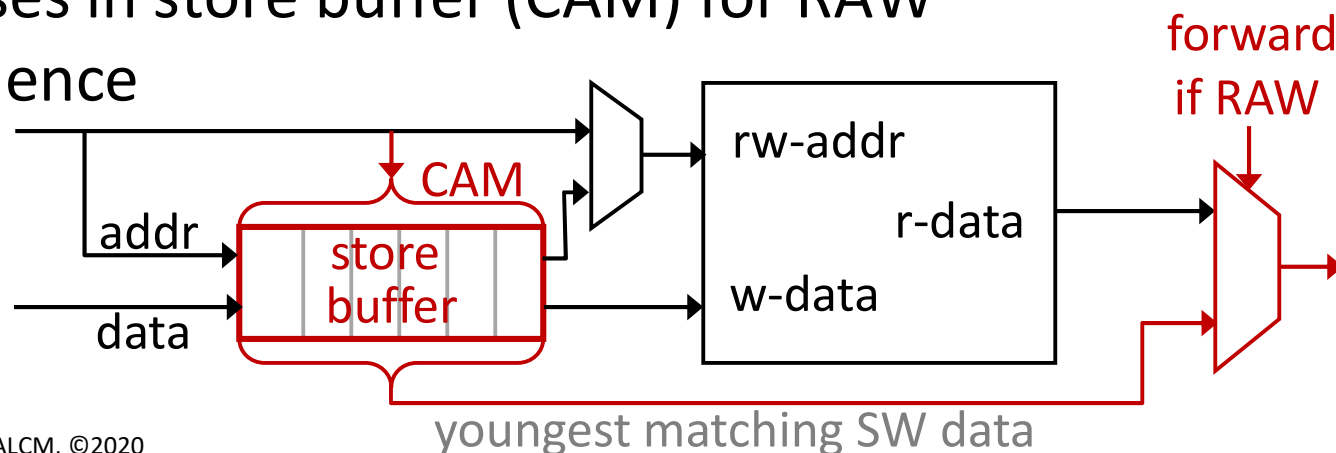
Adding Caches to In-order Pipeline

- 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 bank to ascertain hit before committing to write data bank
 - data bank write happens in the next cycle
 - if SW is followed immediately by LW

⇒ **structural hazard** ⇒ **stall**

Store Buffer

- Why stall when memory port is usually free?
- After tag bank hit, buffer SW address and data until next free data bank cycle
 - 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

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

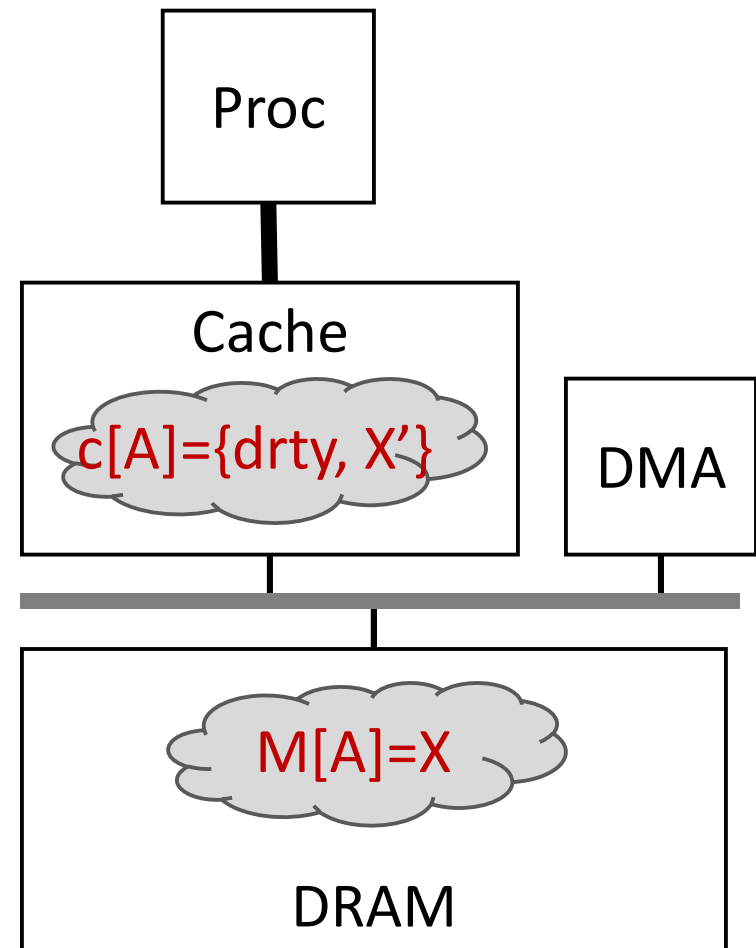
Write-Back Cache

- 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

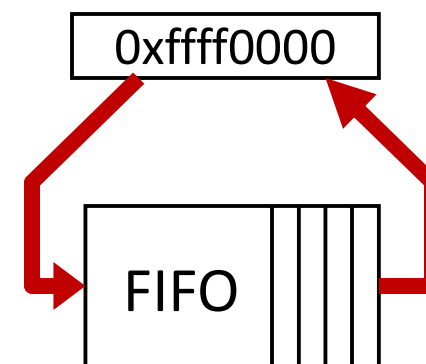


Cache and mmio

- Loading from real memory location $M[A]$ should return most recent value stored to $M[A]$
 - ⇒ writing $M[A]$ once is the same as writing $M[A]$ with same value multiple times in a row
 - ⇒ reading $M[A]$ multiple times returns same value

This is why memory caching works!!

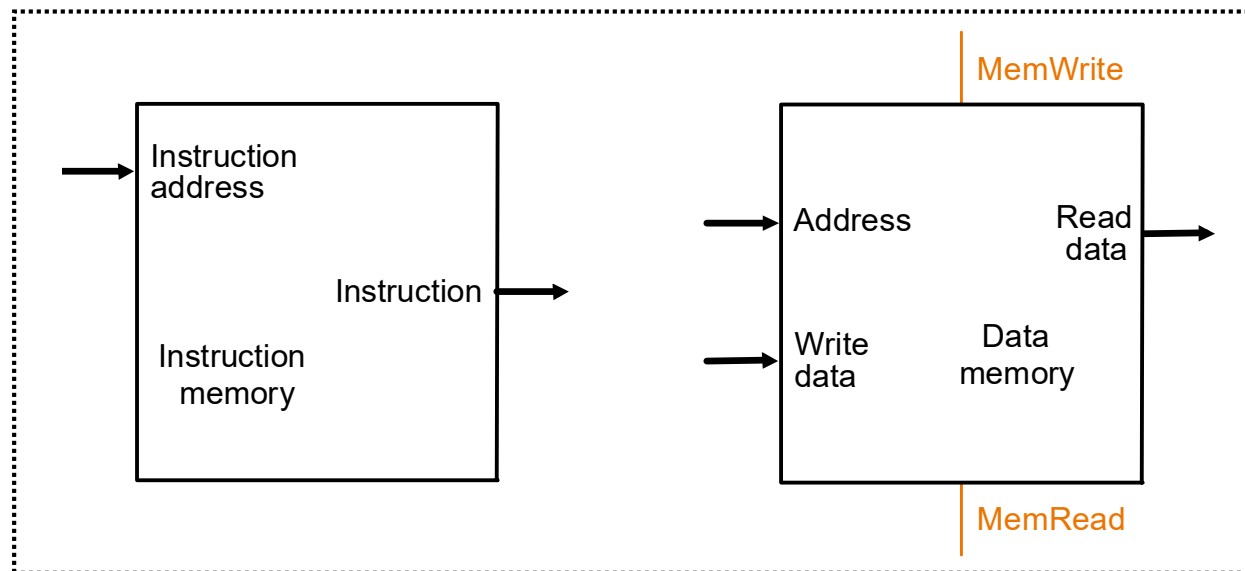
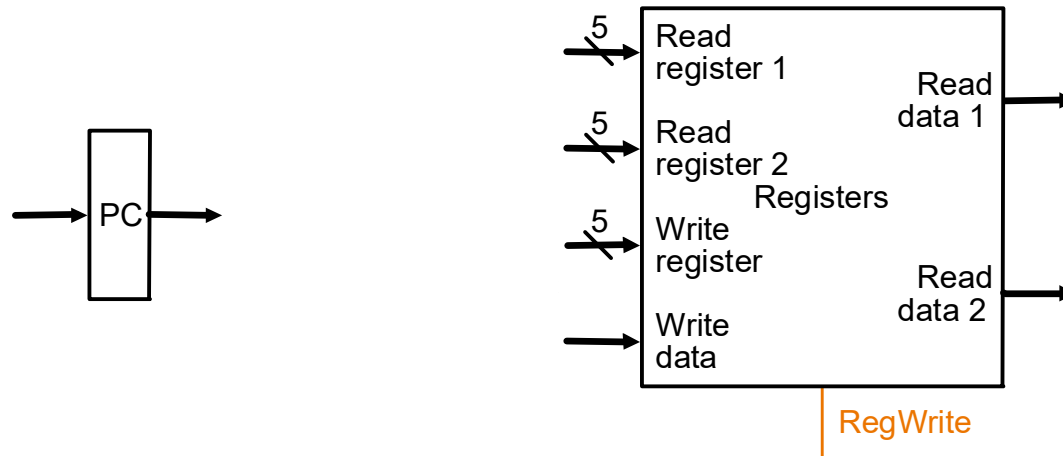
- LW/SW to mmap locations can have side-effects
 - reading/writing mmap location can imply commands and other state changes
 - consider a FIFO example
 - SW to 0xffff0000 pushes value
 - LW from 0xffff0000 returns popped value



What happens if 0xffff0000 is cached?

Recall L13

Program Visible State (aka Architectural State)



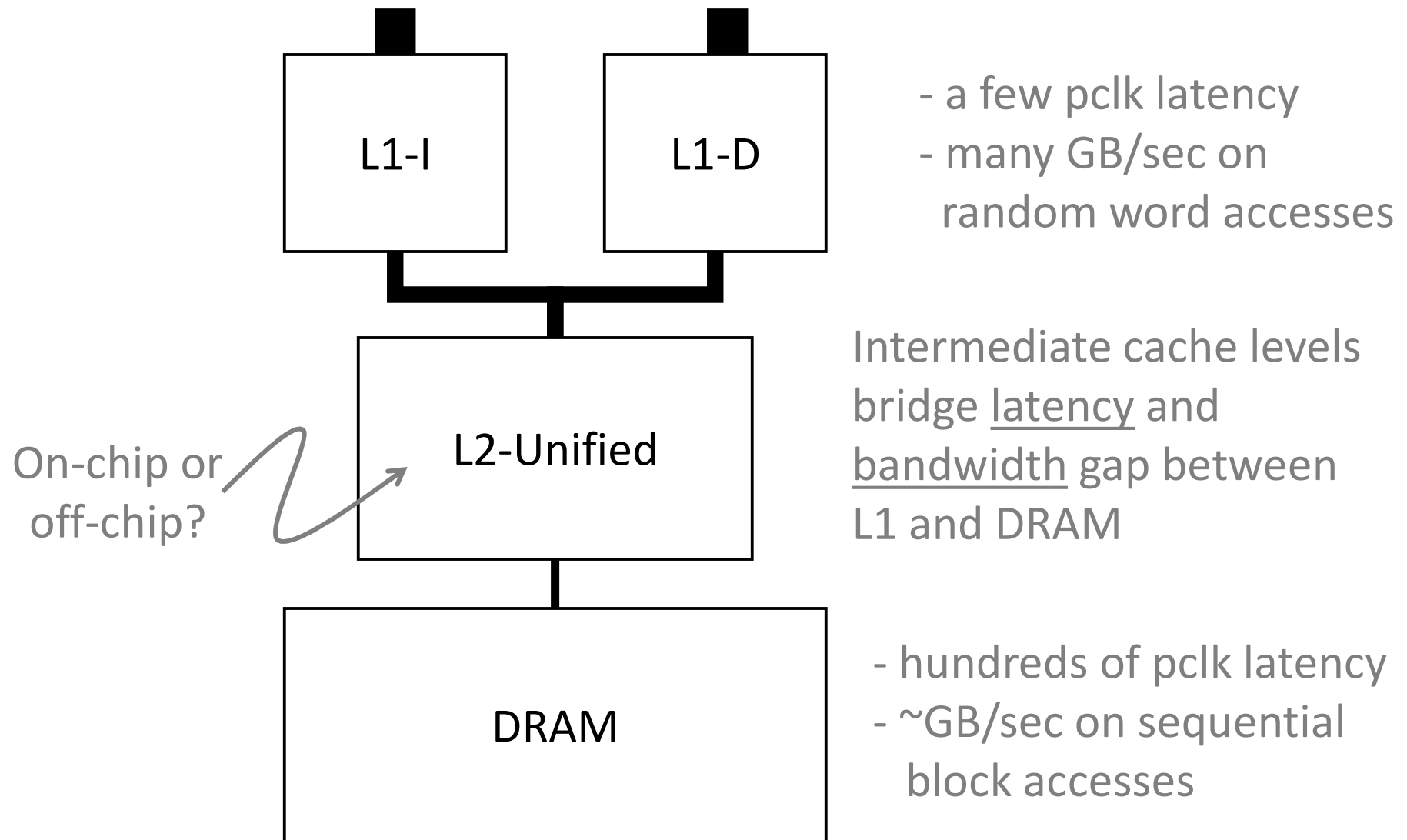
Recall

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

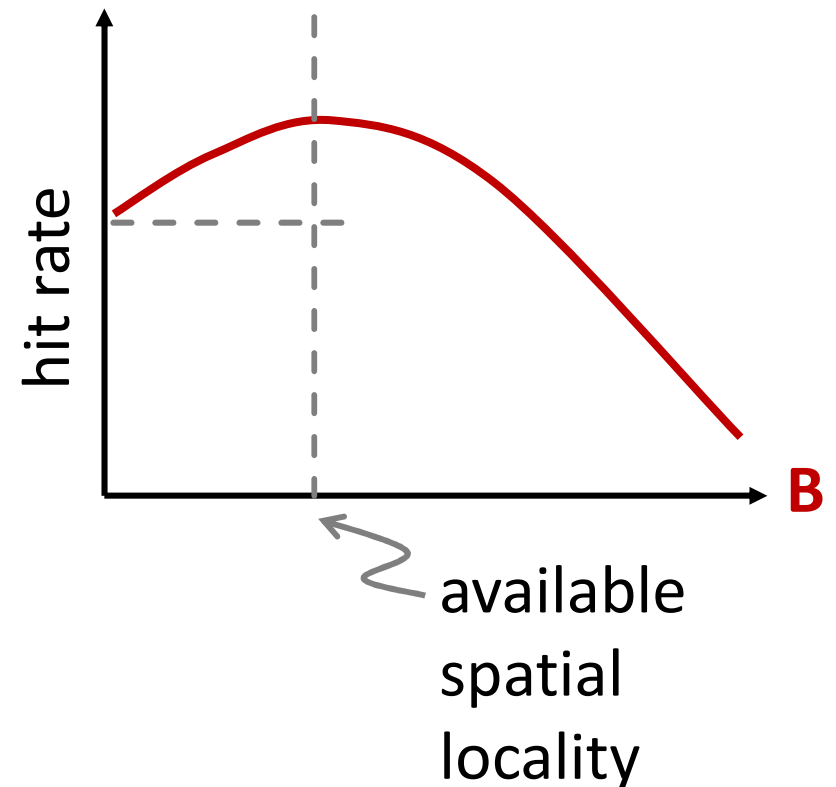
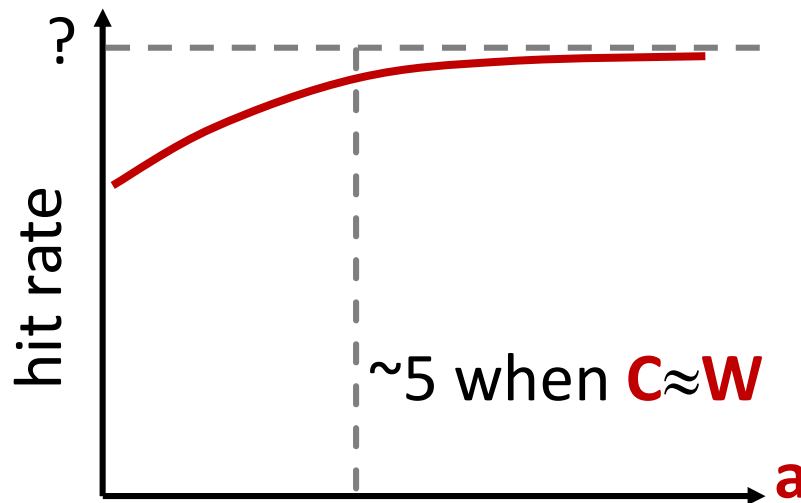
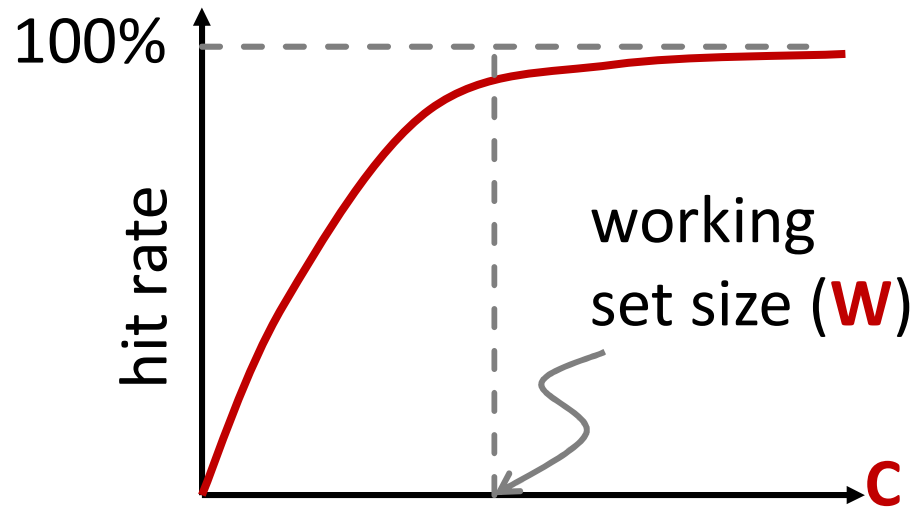
Multi-Level Caches



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”

aBC Rule of Thumb Cribsheet

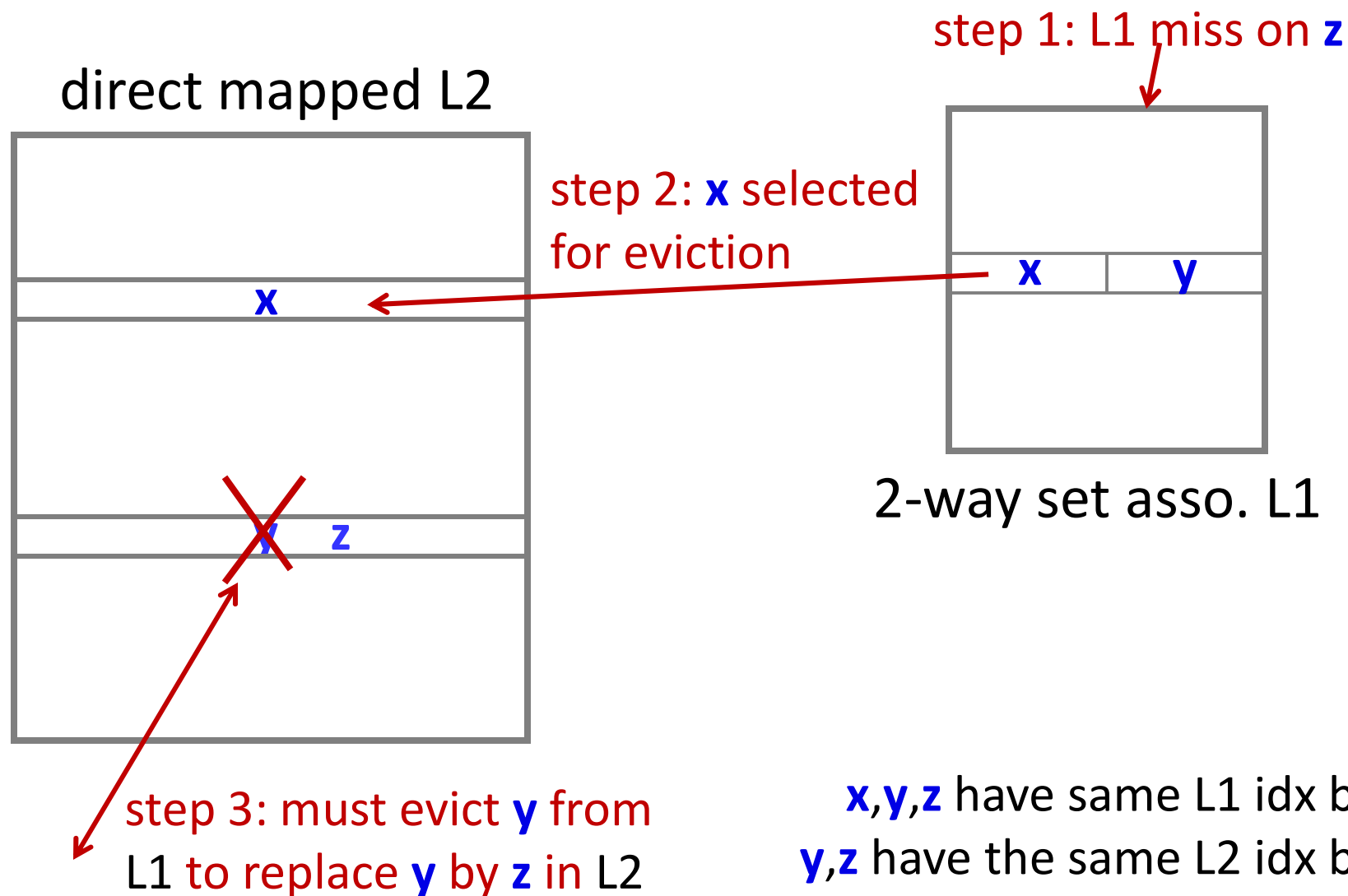


For “typical” programs

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 still common but no longer 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



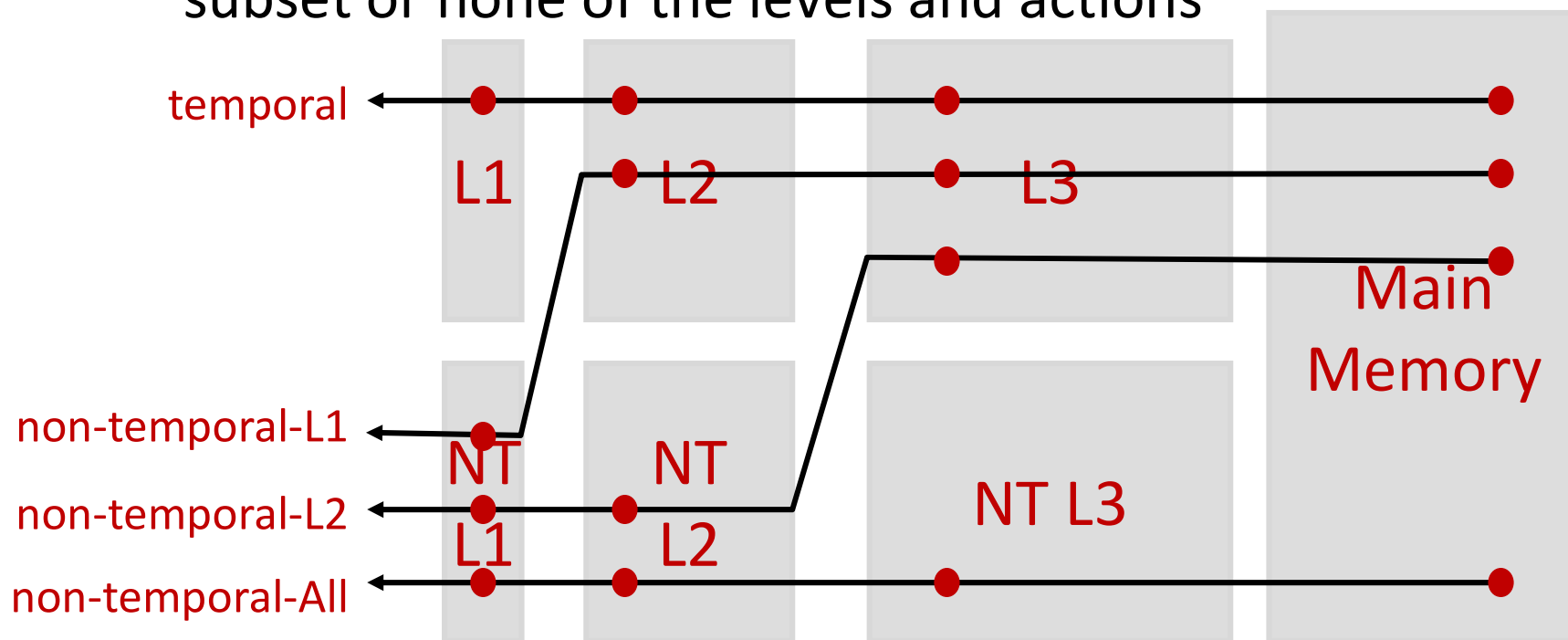
Aside: Victim “Cache”

- 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



Test yourself

Optional Reading: “Measuring Cache and TLB Performance and Their Effect on Benchmark Run Times,” Saavedra and Smith, 2095.

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 range = 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 $R \leq C$, 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 $R \gg C$
- For increasing $S=1,2,4,8,\dots$,
 - repeatedly read every S 'th byte in buffer in sequence
 - measure average read time in steadystate
- Analysis
 - since $R \gg C$, 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 $S \geq B$ (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. $(R/C) \leq a$, 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 in performance tuning

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