CS151B
Computer Systems Architecture

Week 8 Discussion
3/2/2018

Logistics

- HW6 due next Friday

Agenda

- Pipelining
- Caches

Pipelining

- Two primary methods of increasing ILP
  - Go deeper: super-pipelining
  - Go wider: multiple issue

- Multiple issue
  - Static multiple issue: VLIW
  - Dynamic instruction scheduling: superscalar, OoO…

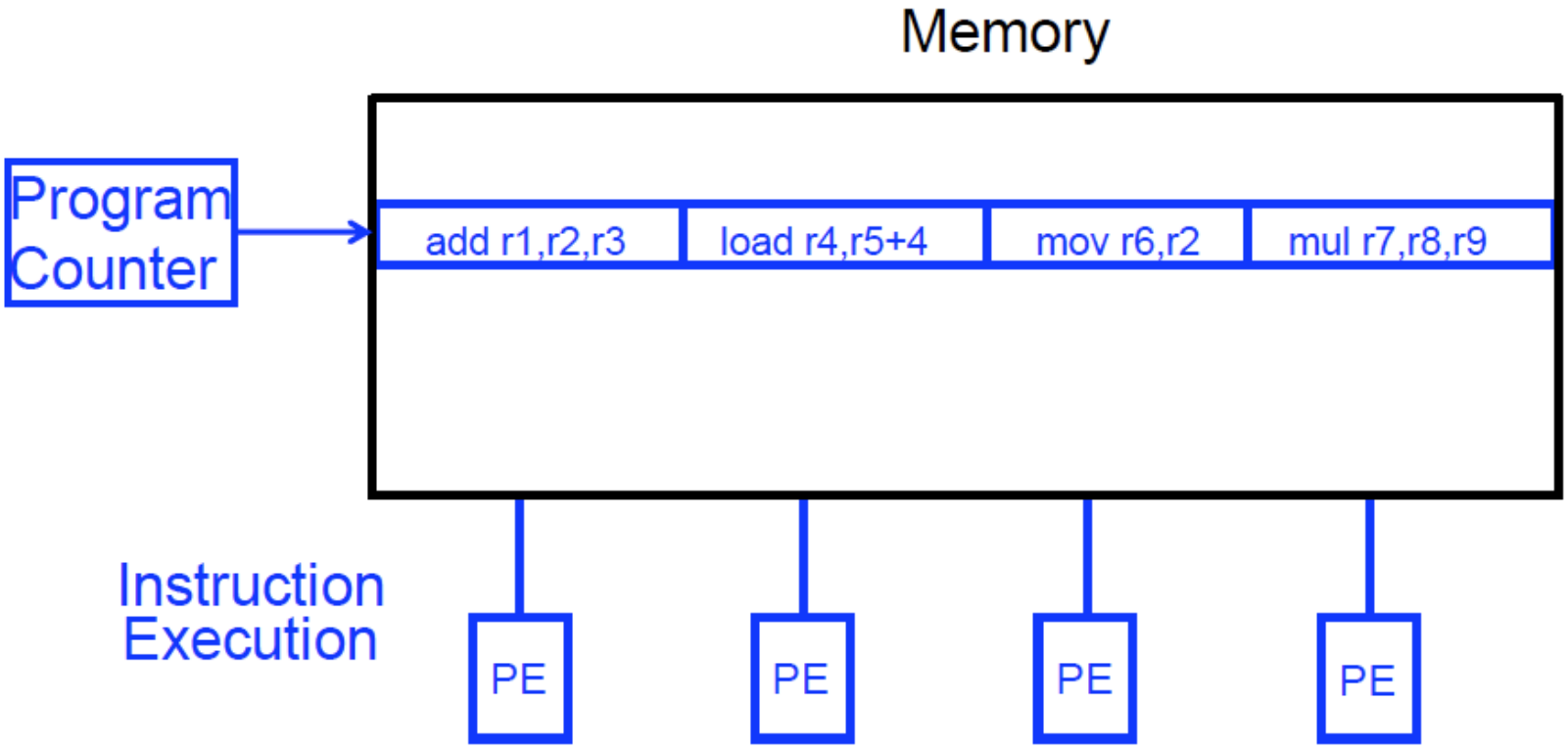
VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler
  - Packed instructions can be logically unrelated

- Idea: **Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction**

- Traditional Characteristics
  - Multiple functional units
  - Each instruction in a bundle executed in **lock step**
  - Instructions in a bundle **statically aligned** to be directly fed into the functional units

VLIW Concept



- Fisher, "**Very Long Instruction Word architectures and the ELI-512**," ISCA 1983.
  - ELI: Enormously longword instructions (512 bits)

VLIW Philosophy

- Philosophy similar to RISC (simple instructions and hardware)
  - Except multiple instructions in parallel
- RISC (John Cocke, 1970s, IBM 801 minicomputer)
  - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
    - And, to reorder simple instructions for high performance
  - Hardware does little translation/decoding → very simple
- VLIW (Fisher, ISCA 1983)
  - Compiler does the hard work to find instruction level parallelism
  - Hardware stays as simple and streamlined as possible
    - Executes each instruction in a bundle in lock step
    - Simple → higher frequency, easier to design

VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
- Solely-compiler approach of VLIW has several downsides that reduce performance
  - -- Too many NOPs (not enough parallelism discovered)
  - -- Static schedule intimately tied to microarchitecture
    - -- Code optimized for one generation performs poorly for next
  - -- No tolerance for variable or long-latency operations (lock step)

- ++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
  - Enable code optimizations
- ++ VLIW successful when parallelism is easier to find by the compiler (traditionally embedded markets, DSPs)

OoO Execution (Dynamic Scheduling)

- Idea: **Move the dependent instructions out of the way of independent ones (s.t. independent ones can execute)**
  - Rest areas for dependent instructions: Reservation stations

- Monitors the source values of each instruction in the resting area
- When all source values of an instruction are available, dispatch the instruction
  - Instructions dispatched in **dataflow (not control-flow) order**

- Benefit
  - **Latency tolerance**: allows independent instructions to execute and complete in the presence of a long latency operation

OoO Execution (Dynamic Scheduling)

- Smith and Sohi, "**The Microarchitecture of Superscalar Processors**," Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - OoO execution and superscalar concepts

Chapter 4: The Processor

- Single-cycle datapath
- Pipelining
- Issues in pipelining
  - Data hazards
    - EX/MEM forwarding
    - load-use penalty
  - Control hazards
    - Branch prediction
    - Misprediction penalty
- ILP: VLIW, loop unrolling

Caches, Caches, Caches

Review: Caching Basics

- **Block (line):** Unit of storage in the cache
  - Memory is logically divided into cache blocks that map to locations in the cache
- When data referenced
  - HIT: if in cache, use cached data instead of accessing memory
  - MISS: if not in cache, bring block into cache
- Some important cache design decisions
  - Placement: where and how to place/find a block in cache?
  - Replacement: what data to remove to make room in cache?
  - Granularity of management: large, small, uniform blocks?
  - Write policy: what do we do about writes?
  - Instructions/data: do we treat them separately?

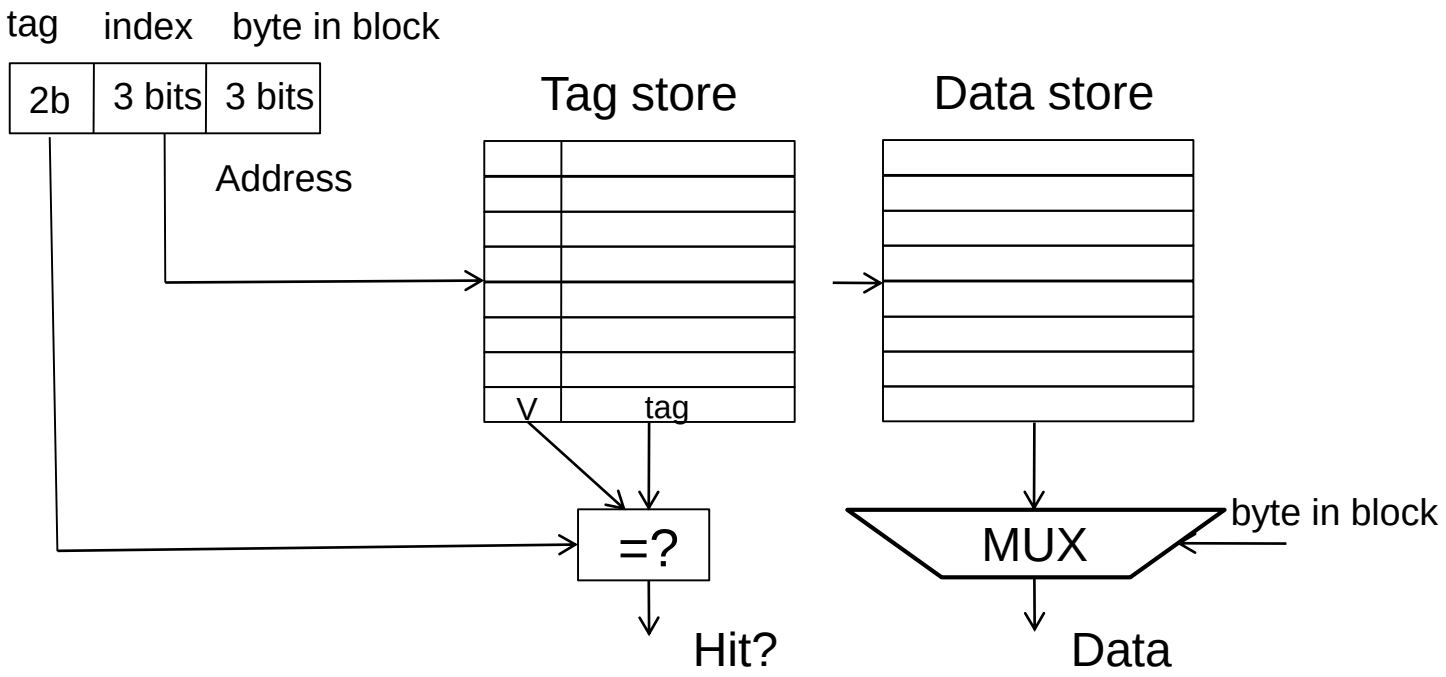
Direct-Mapped Caches

- **Direct-mapped cache**: two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time
  - One index -> one entry

- Can lead to 0% hit rate if more than one block accessed in an interleaved manner map to the same index
  - Assume addresses A and B have the same index bits but different tag bits
  - A, B, A, B, A, B, A, B, … -> conflict in the cache index
  - All accesses are **conflict misses**

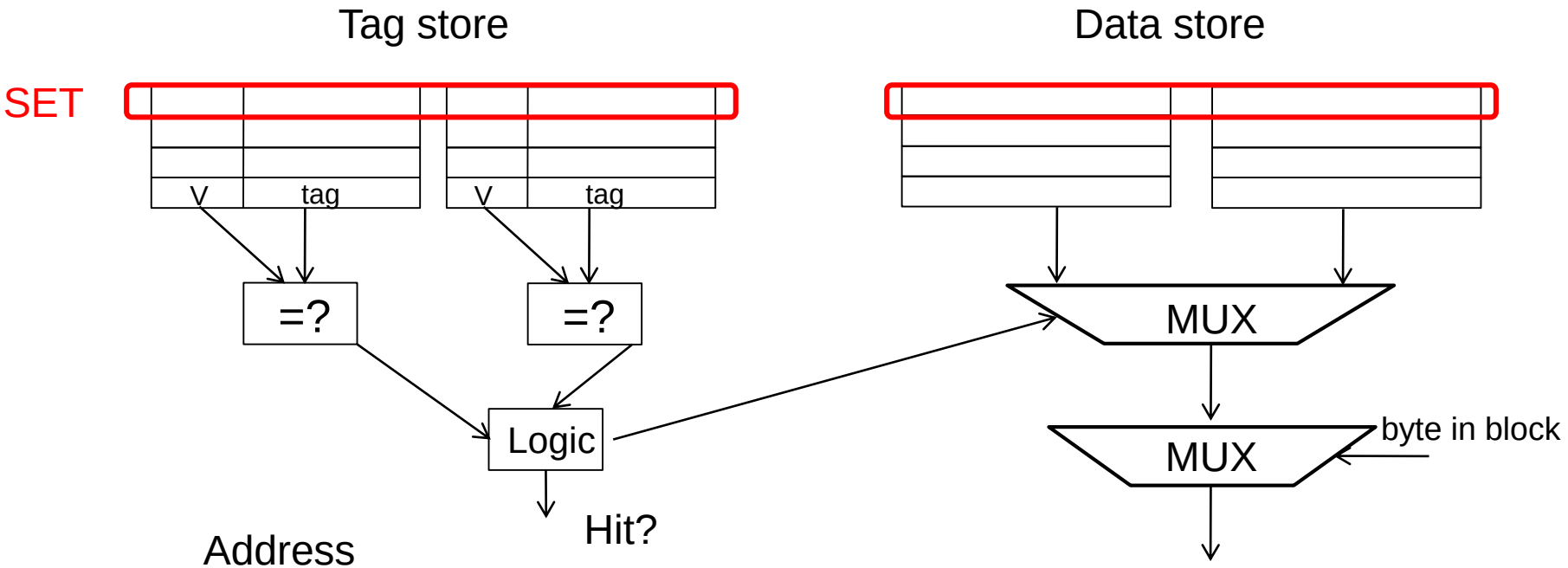
Direct-Mapped Cache

- Assume cache: 64 bytes, 8-byte blocks -> 8 blocks
  - Direct-mapped: a block can go to only one location
  - Addresses with same index contend for the same location



Set Associativity

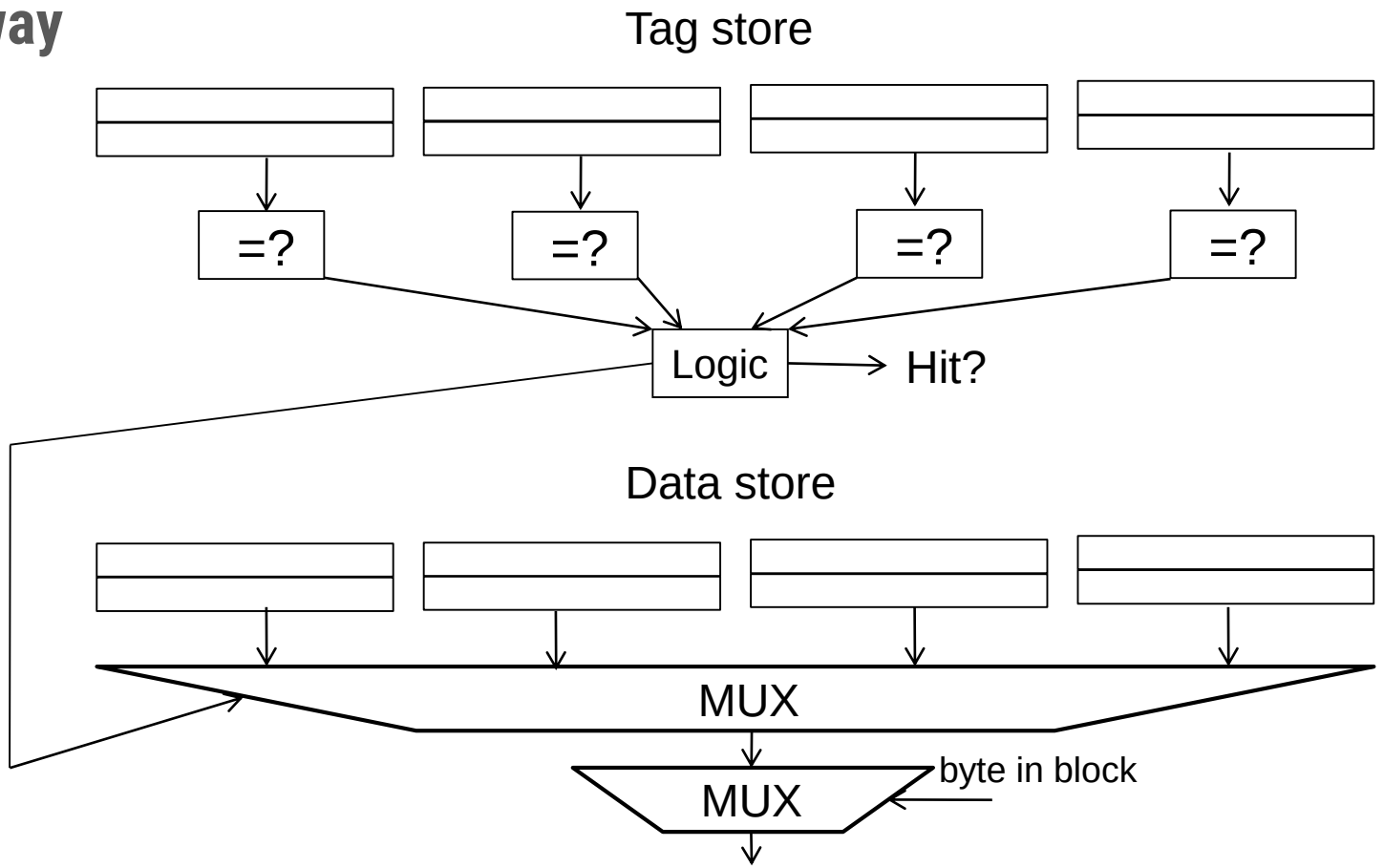
- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks



Key idea: Associative memory within the set
+ Accommodates conflicts better (fewer conflict misses)
-- More complex, slower access, larger tag store

Higher Associativity

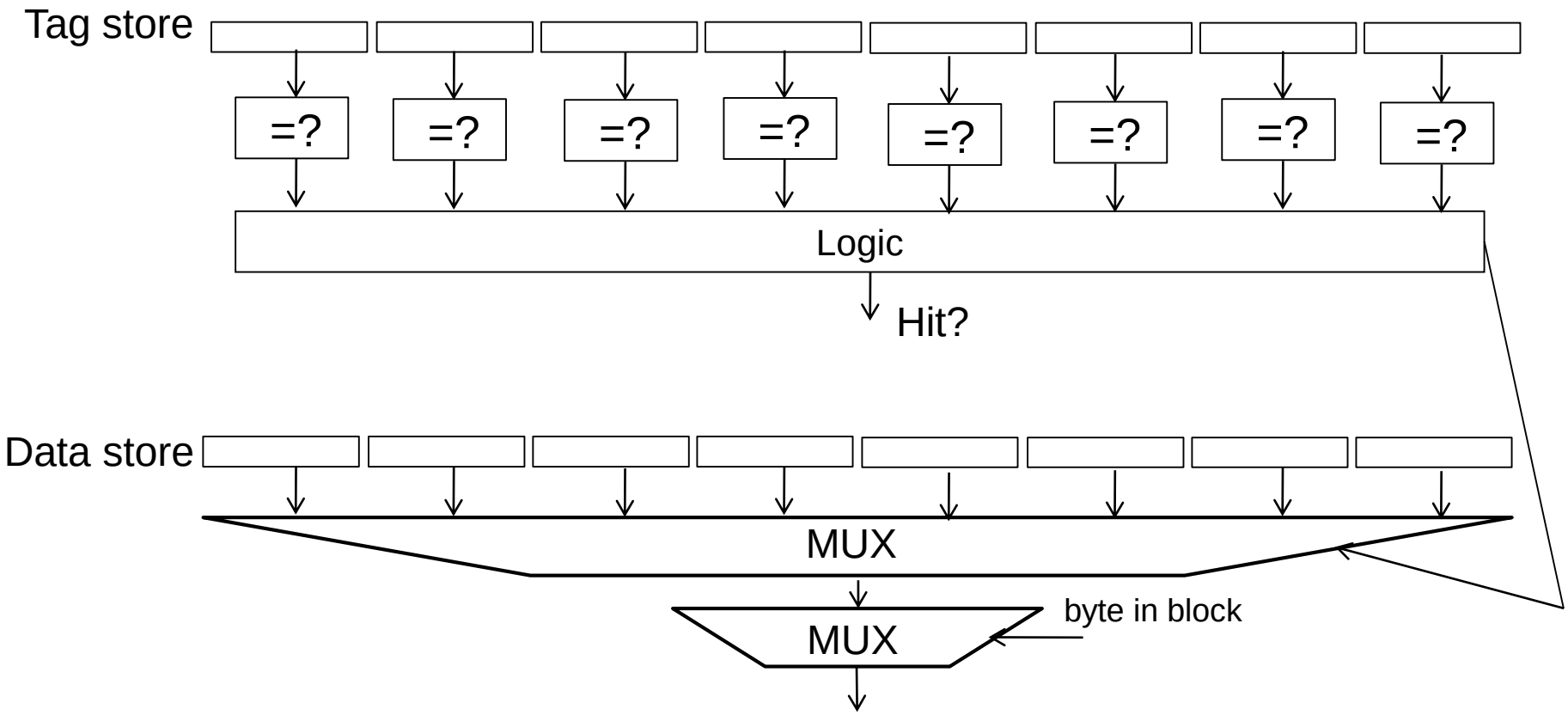
- 4-way



- + Likelihood of conflict misses even lower
- -- More tag comparators and wider data mux; larger tags

Full Associativity

- Fully associative cache
  - A block can be placed in any cache location



Handling Write Hits

- **When do we write the modified data in a cache to the next level?**
  - Write through: At the time the write happens
  - Write back: When the block is evicted

- Write-back
  - Can consolidate multiple writes to the same block before eviction
    - Potentially saves bandwidth between cache levels + saves energy
  - Need a bit in the tag store indicating the block is “dirty/modified”

- Write-through
  - Simpler
  - All levels are up to date. Consistency: Simpler cache coherence because no need to check lower-level caches
  - More bandwidth intensive; no coalescing of writes

Handling Write Misses

- **Do we allocate a cache block on a write miss?**
  - Allocate on write miss: Yes
  - No-allocate on write miss: No

- Allocate on write miss
  - Can consolidate writes instead of writing each of them individually to next level
  - Simpler because write misses can be treated the same way as read misses
  - Requires transfer of the whole cache block

- No-allocate
  - Conserves cache space if locality of writes is low (potentially better cache hit rate)

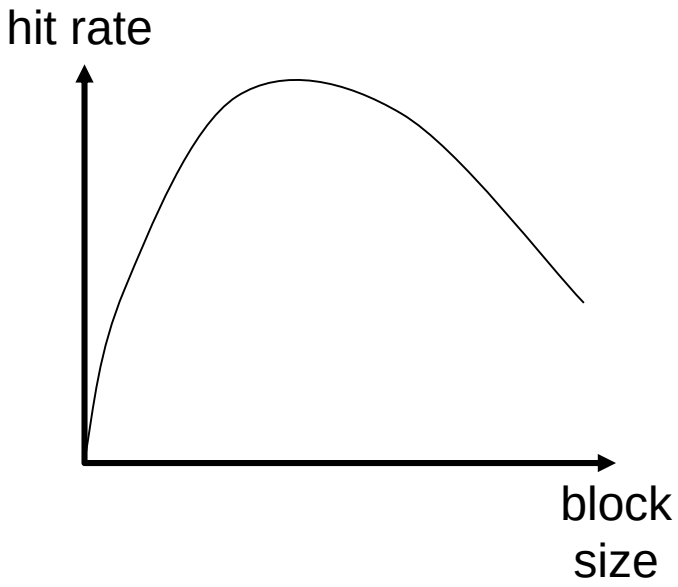
Cache Size Tradeoffs

- Cache size: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn't exploit temporal locality well
  - useful data replaced often

- Working set: the whole set of data the executing application references
  - Within a time interval

Block Size Tradeoffs

- Block size is the data that is associated with an address tag



- Too small blocks
  - don't exploit spatial locality well
  - have larger tag overhead
- Too large blocks
  - too few total # of blocks -> less temporal locality exploitation
  - waste of cache space and bandwidth/energy if spatial locality is not high

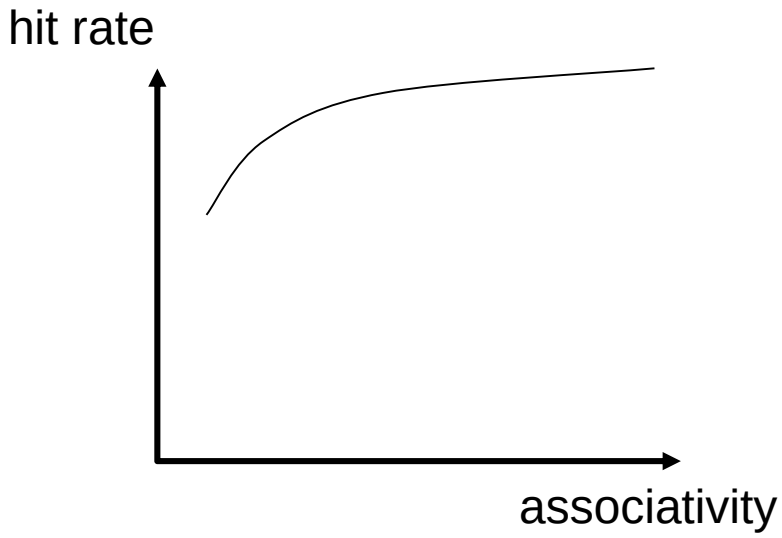
Associativity Tradeoffs

- How many blocks can map to the same index (or set)?

- Larger associativity
  - lower miss rate, less variation among programs
  - diminishing returns, higher hit latency

- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches

- Power of 2 associativity required?



Slides Credit

- Onur Mutlu