Caches and Memory Hierarchy: Review

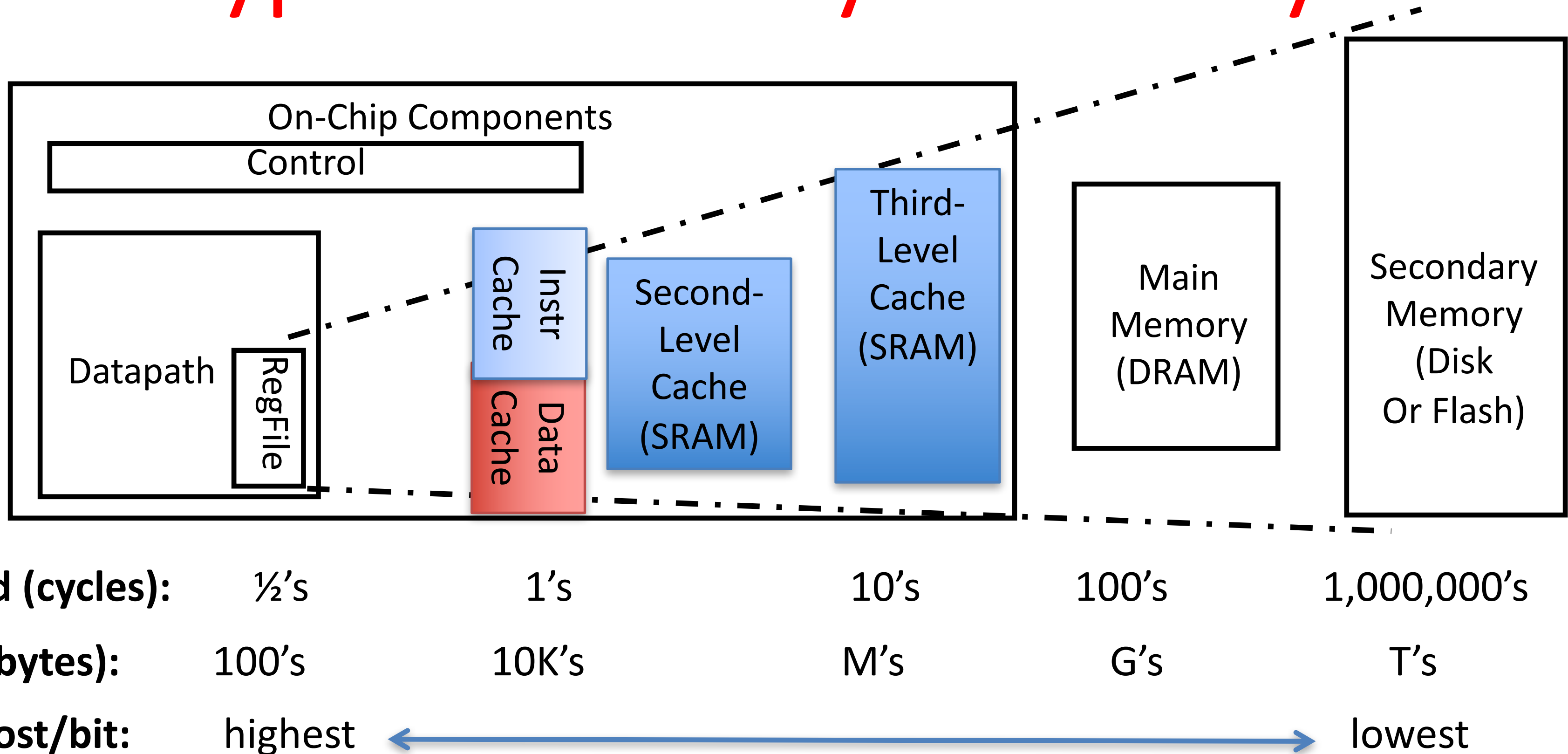
UCSB CS240A, Fall 2017

Motivation

- Most applications in a single processor runs at only 10-20% of the processor peak
- Most of the single processor performance loss is in the memory system
  - Moving data takes much longer than arithmetic and logic

- Parallel computing with low single machine performance is not good enough.
  - Understand high performance computing and cost in a single machine setting
- Review of cache/memory hierarchy

Typical Memory Hierarchy



- Principle of locality + memory hierarchy presents programmer with ≈ as much memory as is available in the *cheapest* technology at the ≈ speed offered by the *fastest* technology

Idealized Uniprocessor Model

- Processor names bytes, words, etc. in its address space
  - These represent integers, floats, pointers, arrays, etc.
- Operations include
  - Read and write into very fast memory called registers
  - Arithmetic and other logical operations on registers
- Order specified by program
  - Read returns the most recently written data
  - Compiler and architecture translate high level expressions into "obvious" lower level instructions

A = B + C ⇒

Read address(B) to R1
Read address(C) to R2
R3 = R1 + R2
Write R3 to Address(A)

  - Hardware executes instructions in order specified by compiler
- *Idealized Cost*
  - Each operation has roughly the same cost

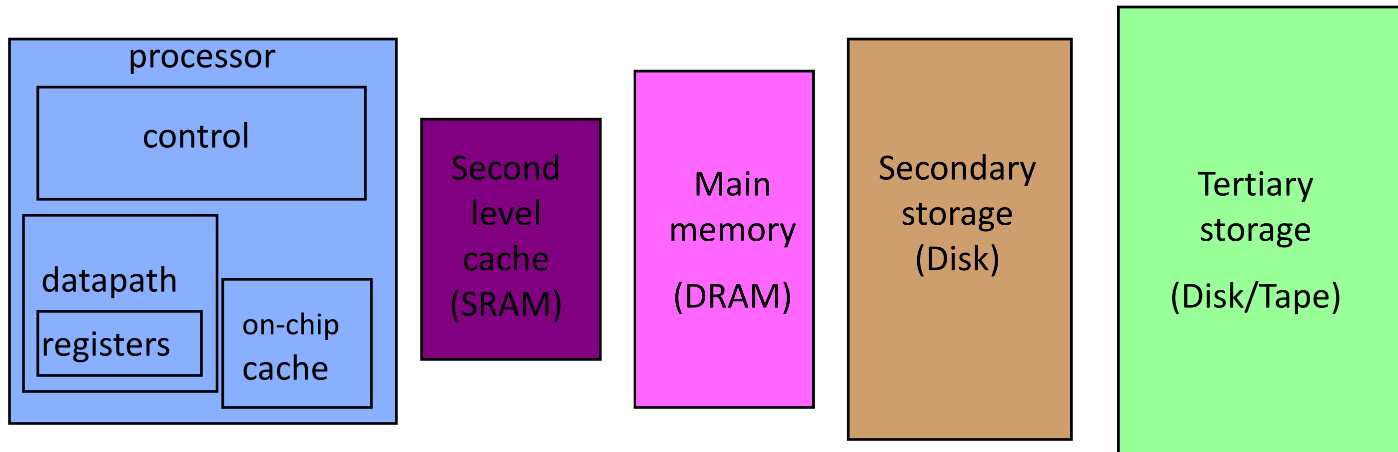
    (read, write, add, multiply, etc.)

Uniprocessors in the Real World

- **Real processors have**
  - **registers and caches**
    - **small amounts of fast memory**
    - **store values of recently used or nearby data**
    - **different memory ops can have very different costs**
  - **parallelism**
    - **multiple “functional units” that can run in parallel**
    - **different orders, instruction mixes have different costs**
  - **pipelining**
    - **a form of parallelism, like an assembly line in a factory**
- **Why is this your problem?**
    - **In theory, compilers and hardware “understand” all this and can optimize your program; in practice they don’t.**
    - **They won’t know about a different algorithm that might be a much better “match” to the processor**

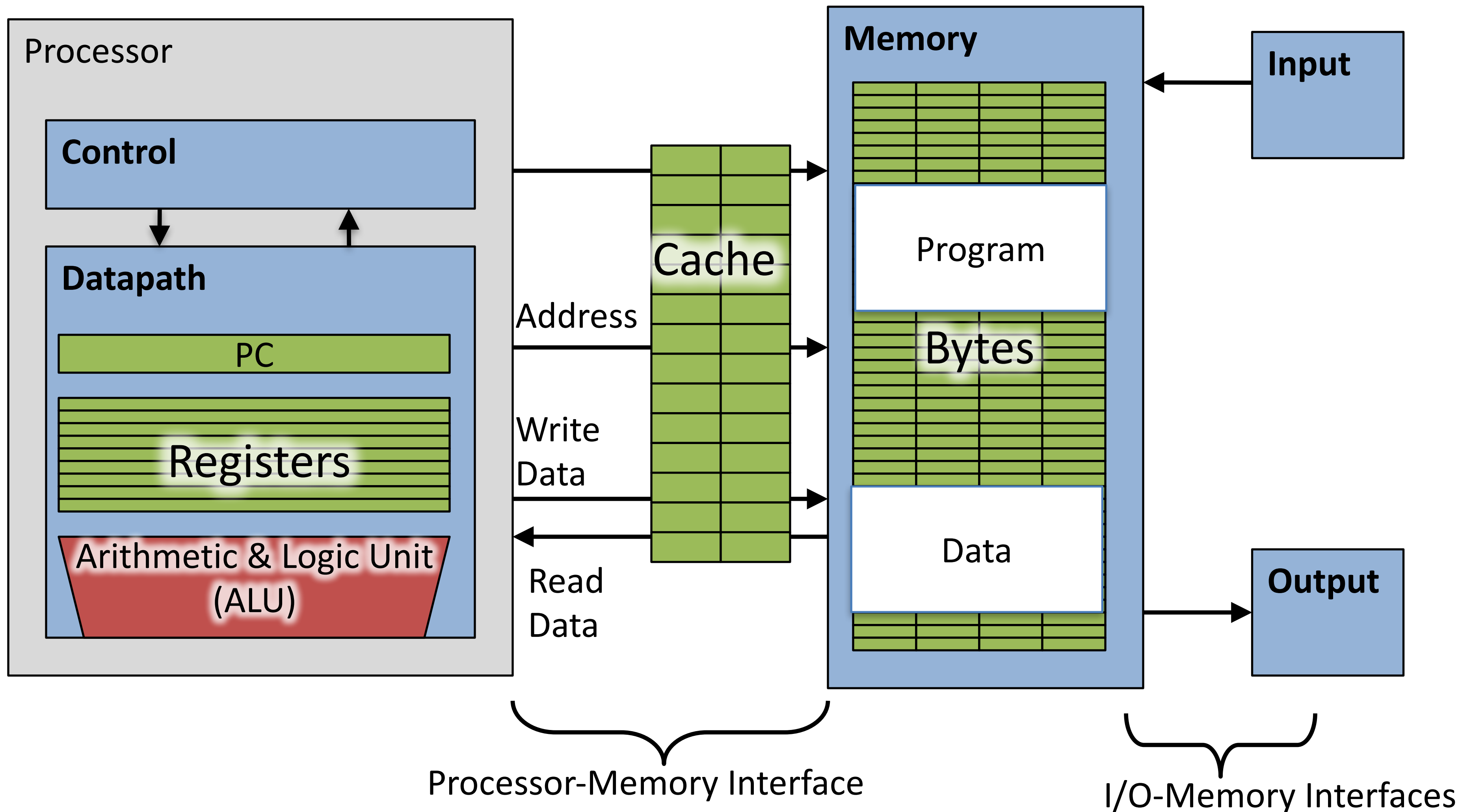
Memory Hierarchy

- Most programs have a high degree of locality in their accesses
  - **spatial locality:** accessing things nearby previous accesses
  - **temporal locality:** reusing an item that was previously accessed
- Memory hierarchy tries to exploit locality to improve average



Speed	1ns	10ns	100ns	1-10ms	10sec
Size	KB	MB	GB	TB	PB

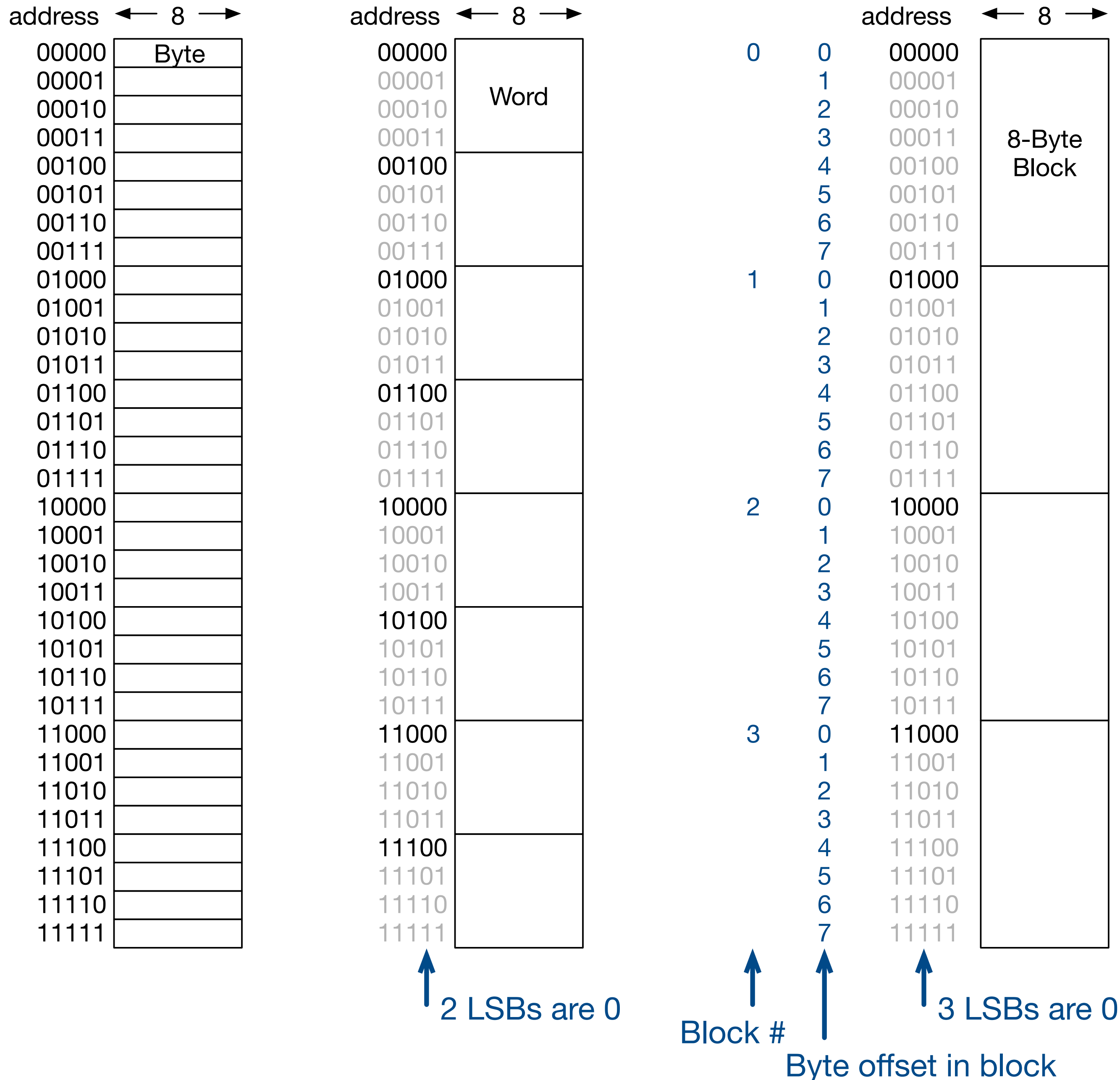
Review: Cache in Modern Computer Architecture



Cache Basics

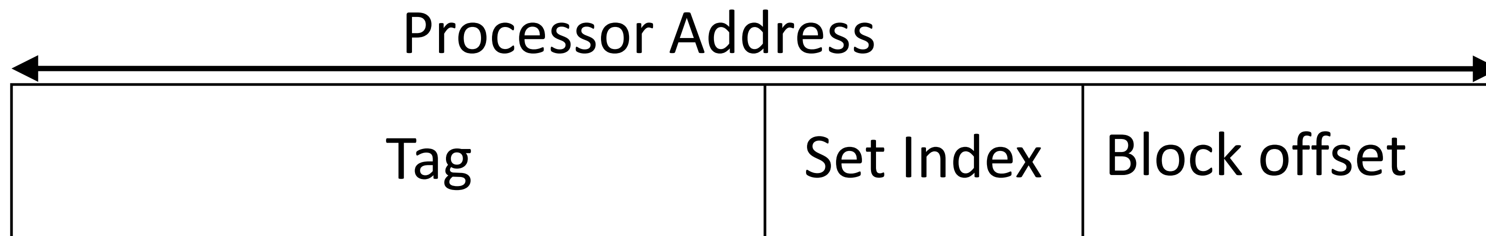
- **Cache is fast (expensive) memory which keeps copy of data in main memory; it is hidden from software**
  - **Simplest example: data at memory address xxxxx1101 is stored at cache location 1101**
- Memory data is divided into blocks
  - Cache access memory by a block (cache line)
  - Cache line length: # of bytes loaded together in one entry
- **Cache is divided by the number of sets**
  - **A cache block can be hosted in one set.**
- **Cache hit: in-cache memory access—cheap**
- **Cache miss: Need to access next, slower level of cache**

Memory Block-addressing example



Processor Address Fields used by Cache Controller

- Block Offset: Byte address within block
  - B is number of bytes per block
- Set Index: Selects which set. S is the number of sets
- Tag: Remaining portion of processor address



- Size of Tag = Address size – log(S) – log(B)

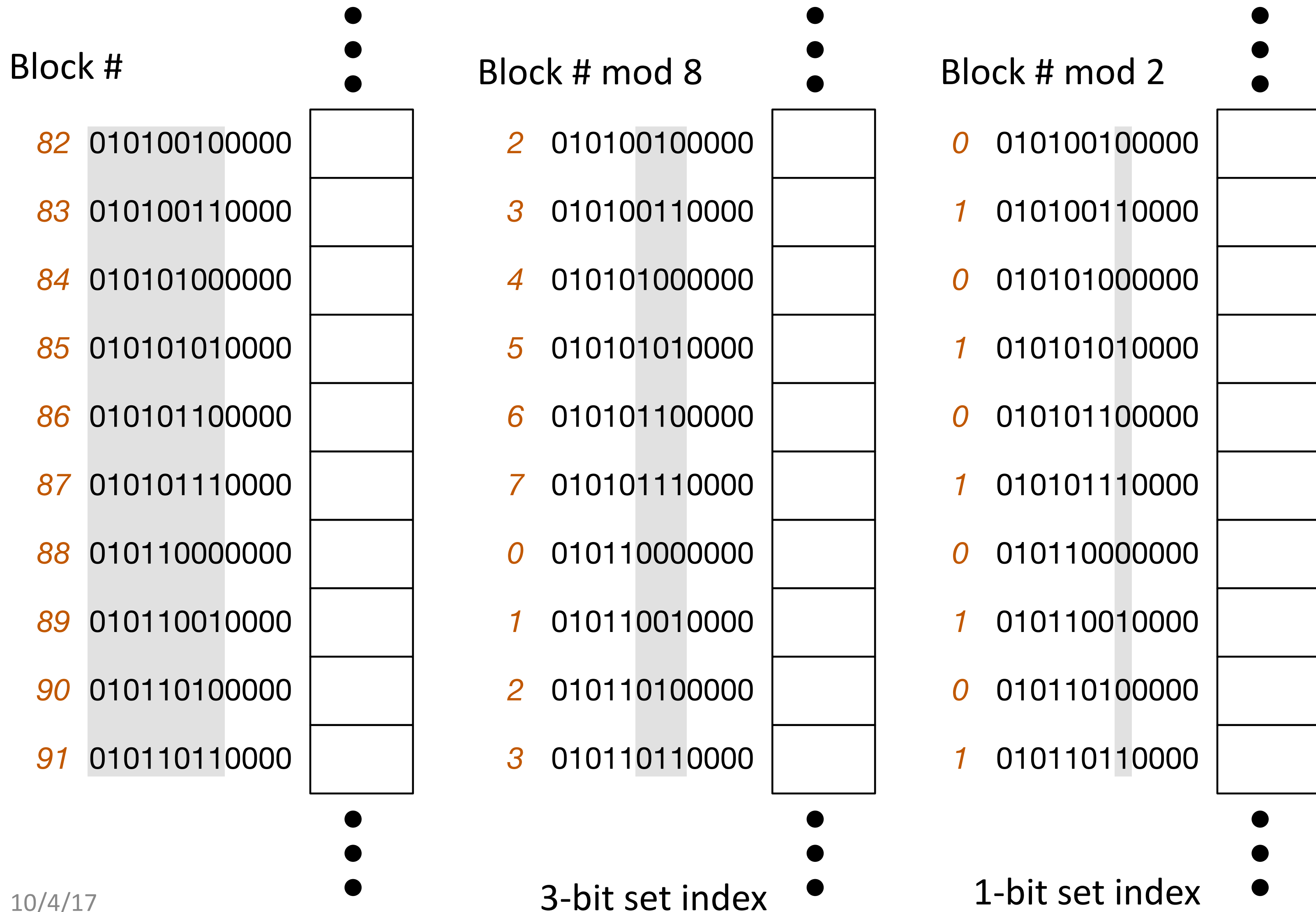
Cache Size C = Associativity N × *# of Set S* × *Cache Block Size B*

Example: Cache size 16K. 8 bytes as a block. → 2K blocks → If N=1, S=2K using 11 bits.

Associativity N *represents # items that can be held per set*

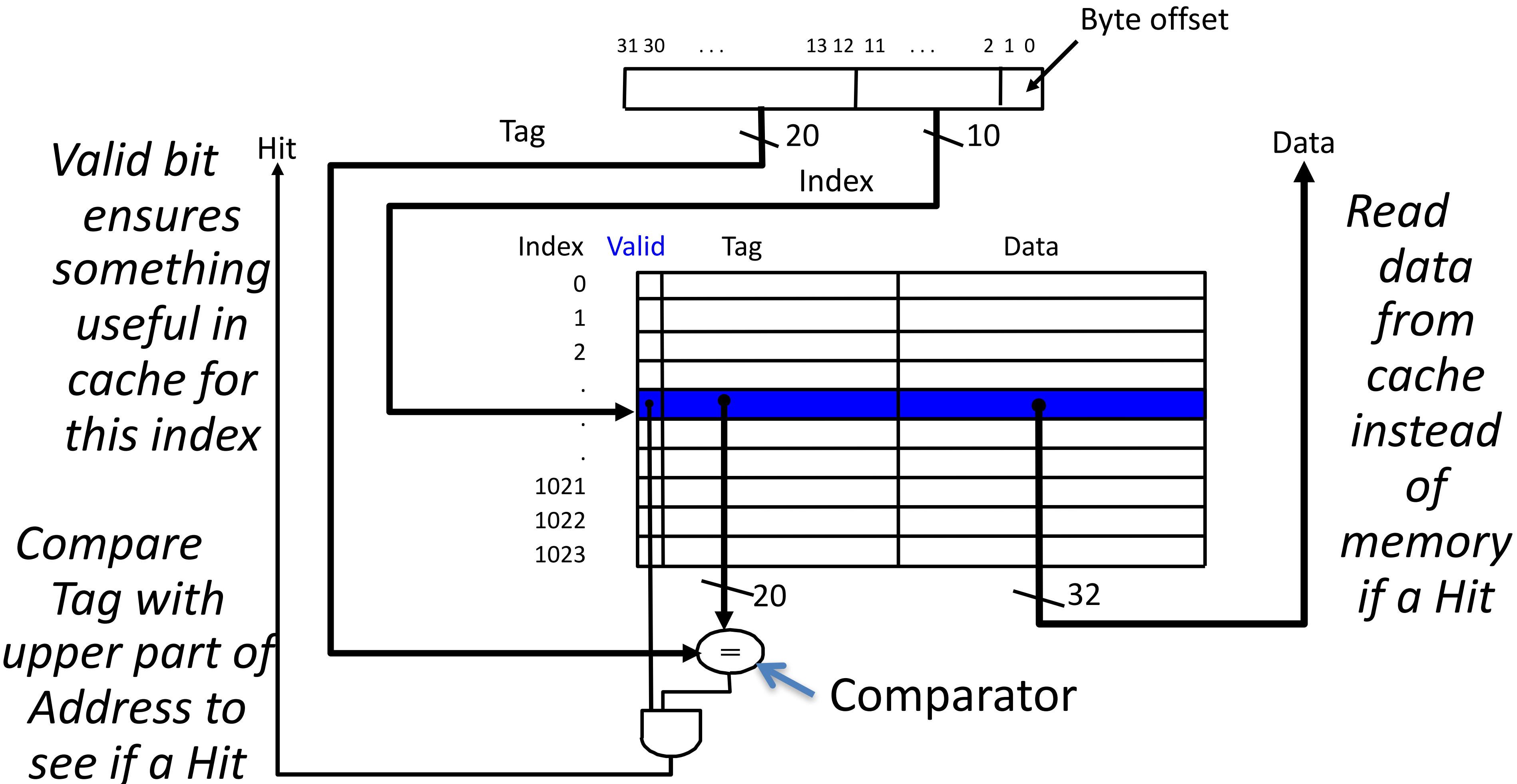
Block number aliasing example

12-bit memory addresses, 16 Byte blocks



Direct-Mapped Cache: N=1. S=Number of Blocks=2^{10}

- 4byte blocks, cache size = 1K words (or 4KB)



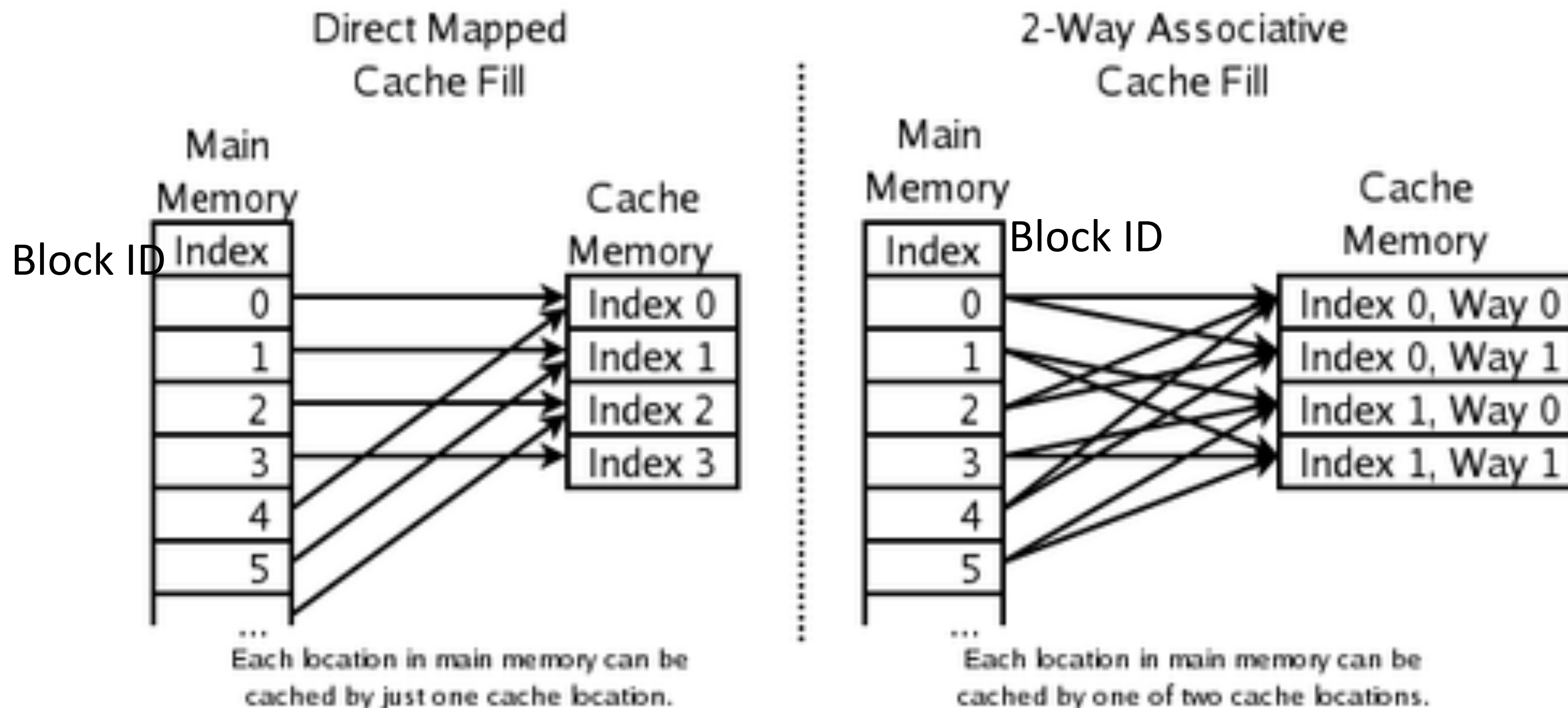
Cache Size C = Associativity N × *# of Set S* × *Cache Block Size B*

Cache Organizations

Cache Size C = N × *# of Set* S × *Size* B

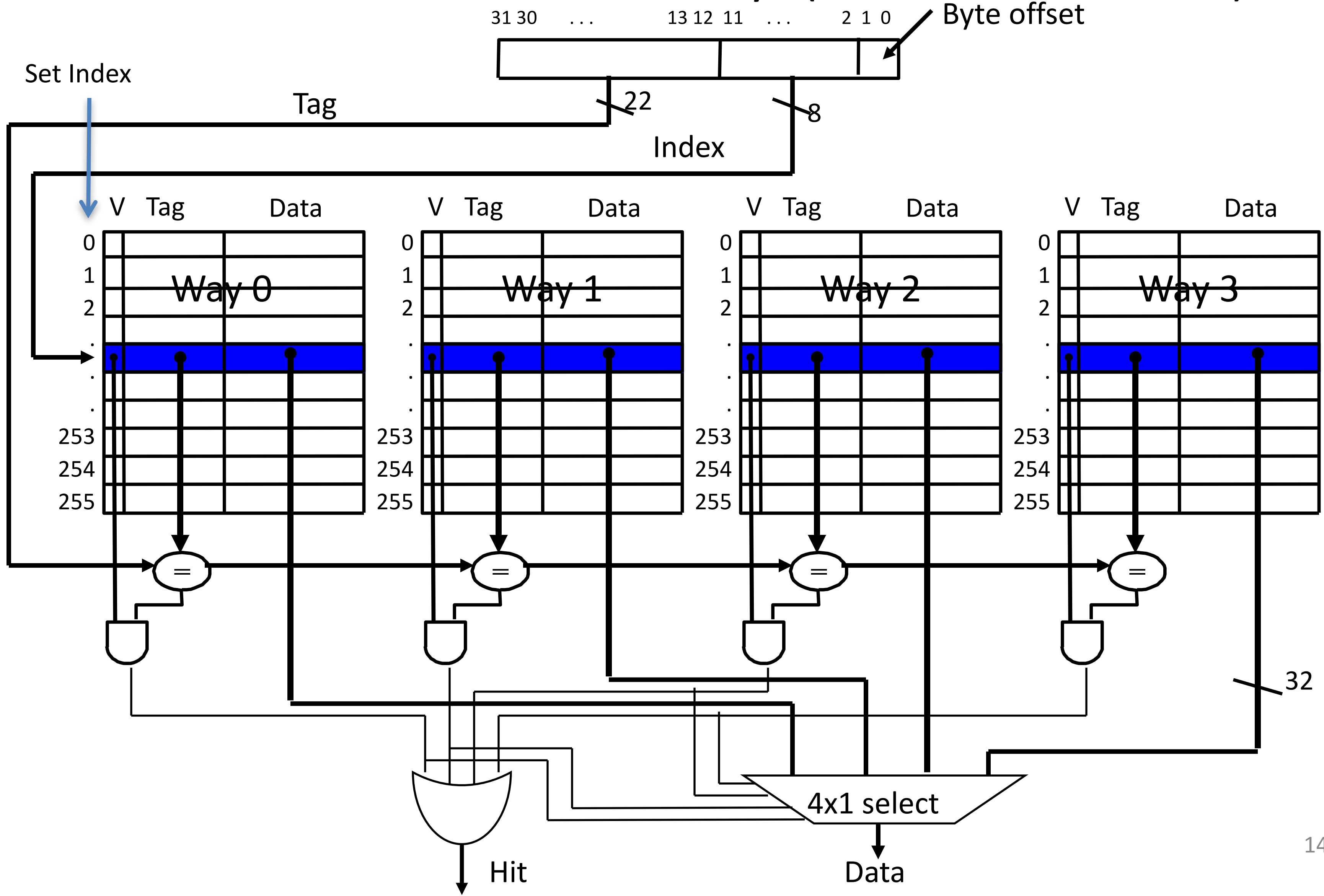
Associativity N *represents # items that can be held per set*

- "Fully Associative": Block can go anywhere
  - N= number of blocks. S=1
- "Direct Mapped": Block goes one place
  - N=1. S= cache capacity in terms of number of blocks
- "N-way Set Associative": N places for a block



Four-Way Set-Associative Cache

- 2^8 = 256 sets each with four ways (each with one block)



How to find if a data address in cache?

0b means binary number

- Assume block size 8 bytes →last 3 bits of address are offset.
- Set index 2 bits.
- Given address 0b1001011, where to find this item from cache?

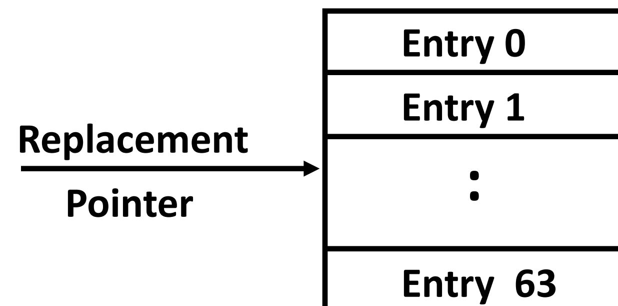
How to find if a data address in cache?

0b means binary number

- Assume block size 8 bytes →last 3 bits of address are offset.
- Set index 2 bits.
- 0b1001011 → Block number 0b1001.
- Set index 2 bits (mod 4)
  - Set number → 0b01.
- Tag = 0b10.
  - If directory based cache, only one block in set #1.
  - If 4 ways, there could be 4 blocks in set #1.
  - Use tag 0b10 to compare what is in the set.

Cache Replacement Policies

- **Random Replacement**
  - Hardware randomly selects a cache evict
- **Least-Recently Used**
  - Hardware keeps track of access history
  - Replace the entry that has not been used for the longest time
  - For 2-way set-associative cache, need one bit for LRU replacement
- Example of a Simple "Pseudo" LRU Implementation
  - Assume 64 Fully Associative entries
  - Hardware replacement pointer points to one cache entry
  - Whenever access is made to the entry the pointer points to:
    - Move the pointer to the next entry
  - Otherwise: do not move the pointer
  - (example of "not-most-recently used" replacement policy)

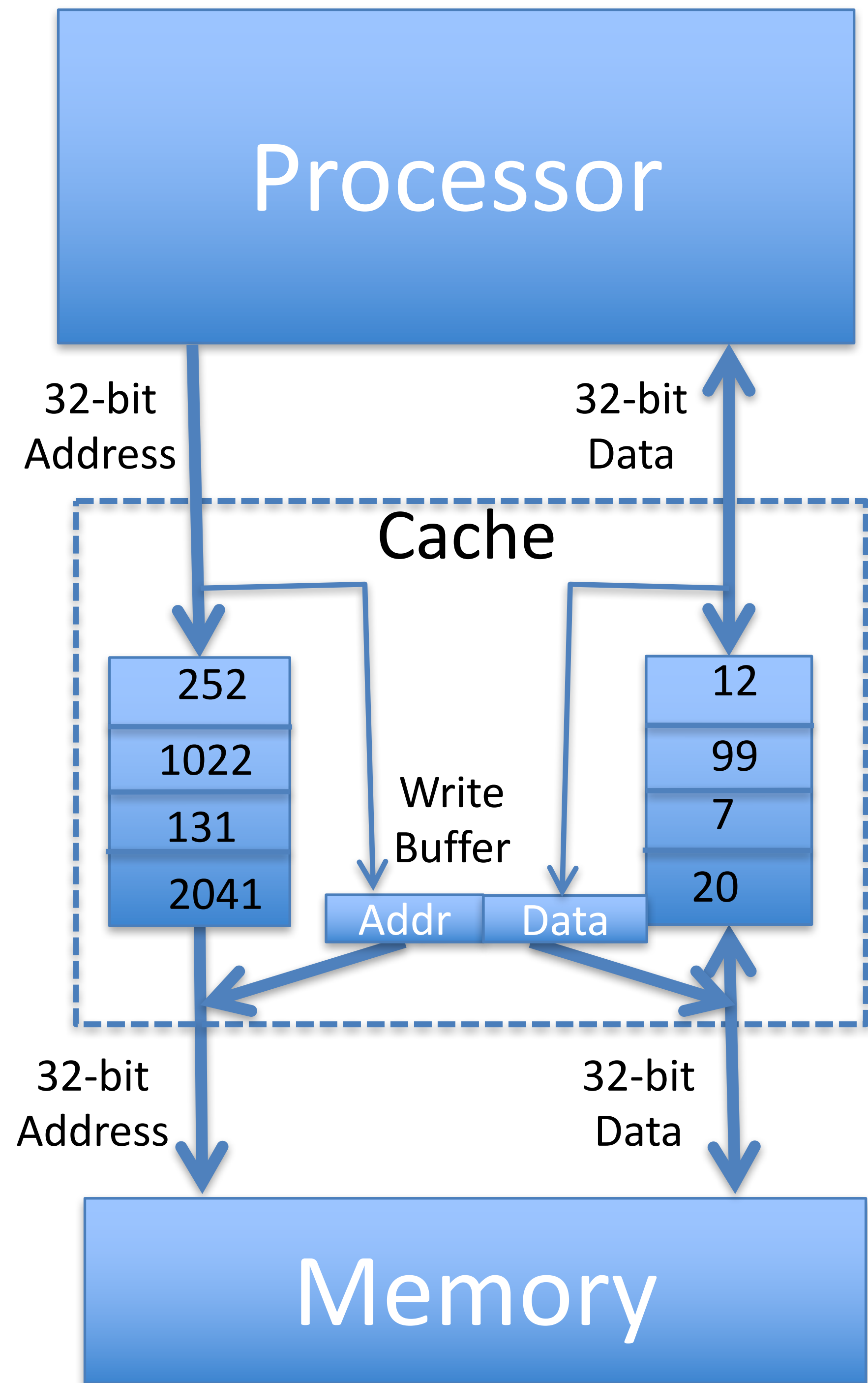


Handling Data Writing

- Store instructions write to memory, changing values
- Need to make sure cache and memory have same values on writes: 2 policies
  - 1) Write-through policy: write cache and write *through* the cache to memory
    - Every write eventually gets to memory
    - Too slow, so include Write Buffer to allow processor to continue once data in Buffer
    - Buffer updates memory in parallel to processor
  - 2) Write-back policy

Write-Through Cache

- Write both values in cache and in memory
- Write buffer stops CPU from stalling if memory cannot keep up
- Write buffer may have multiple entries to absorb bursts of writes
- What if store misses in cache?



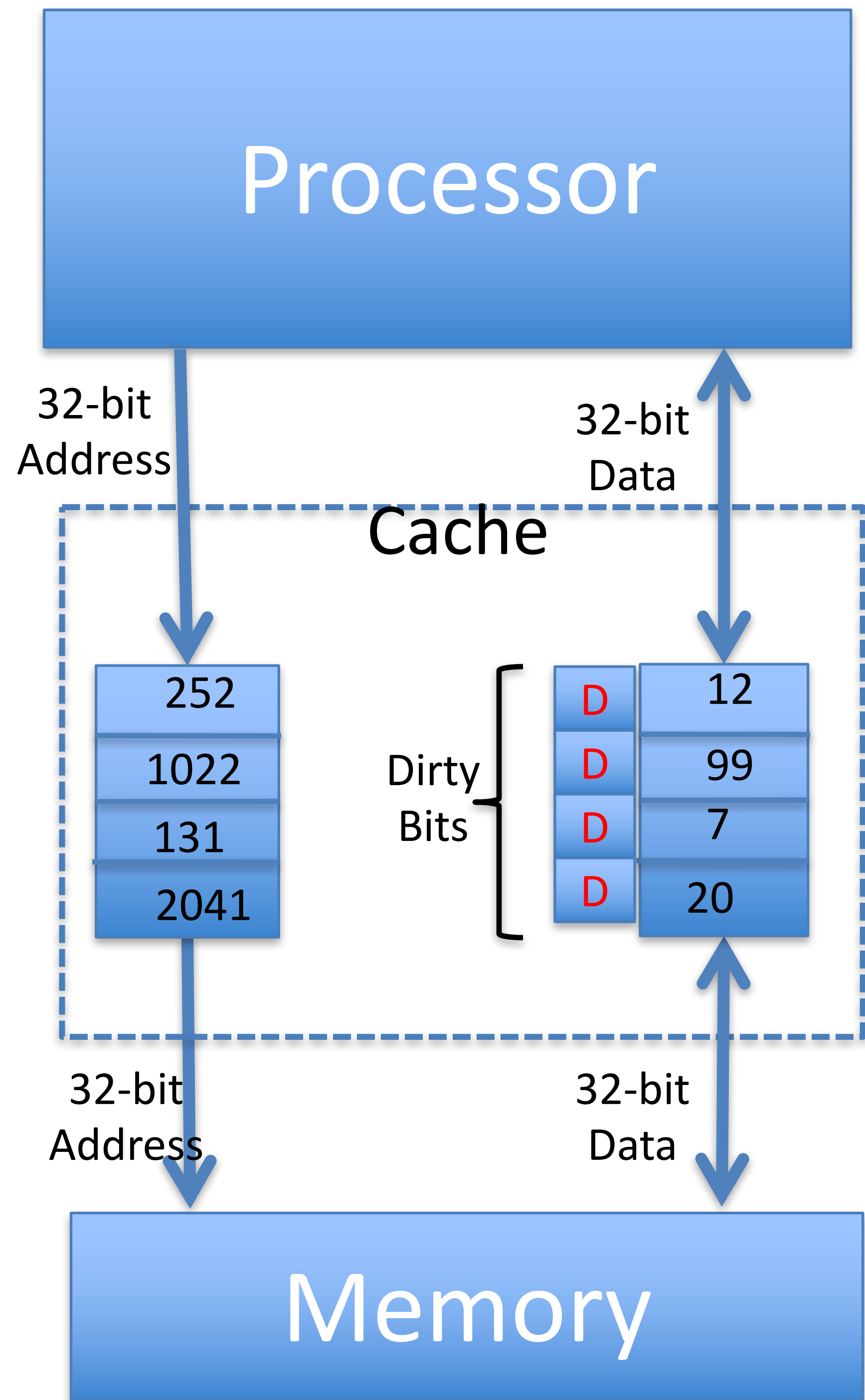
Handling Stores with Write-Back

2) Write-Back Policy: write only to cache and then write cache block *back* to memory when evict block from cache

- Writes collected in cache, only single write to memory per block
- Include bit to see if wrote to block or not, and then only write back if bit is set
  - Called "Dirty" bit (writing makes it "dirty")

Write-Back Cache

- Store/cache hit, write data in cache *only* & set dirty bit
  - Memory has stale value
- Store/cache miss, read data from memory, then update and set dirty bit
  - "Write-allocate" policy
- Load/cache hit, use value from cache
- On any miss, write back evicted block, only if dirty. Update cache with new block and clear dirty bit.



Write-Through vs. Write-Back

- Write-Through:
  - Simpler control logic
  - More predictable timing simplifies processor control logic
  - Easier to make reliable, since memory always has copy of data (big idea: Redundancy!)

- Write-Back
  - More complex control logic
  - More variable timing (0,1,2 memory accesses per cache access)
  - Usually reduces write traffic
  - Harder to make reliable, sometimes cache has only copy of data

Cache (*Performance*) Terms

- Hit rate: fraction of accesses that hit in the cache
- Miss rate: 1 – Hit rate
- Miss penalty: time to replace a block from lower level in memory hierarchy to cache
- Hit time: time to access cache memory (including tag comparison)

Average Memory Access Time (AMAT)

- Average Memory Access Time (AMAT) is the average time to access memory considering both hits and misses in the cache

AMAT = Time for a hit

+ Miss rate × Miss penalty

Given a 0.2ns clock, a miss penalty of 50 clock cycles, a miss rate of 2% per instruction and a cache hit time of 1 clock cycle, what is AMAT?

AMAT = 1 cycle + 0.02*50 = 2 cycles = 0.4ns.