Caches and Memory Hierarchy: Review

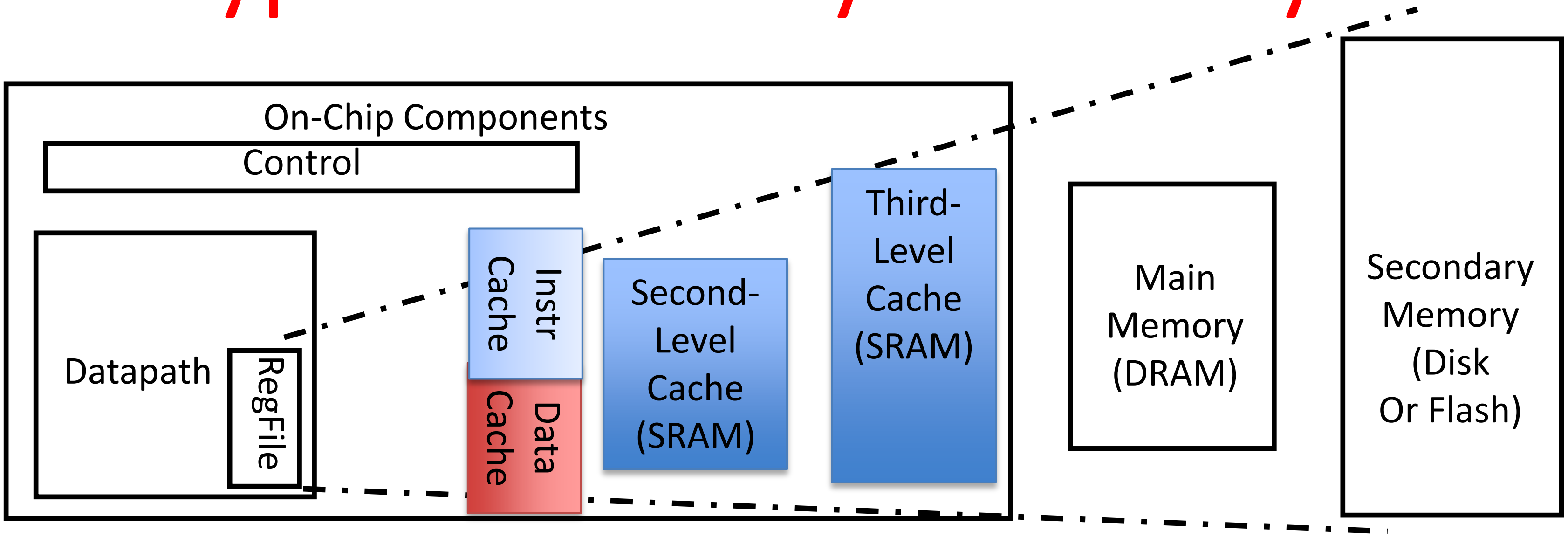
UCSB CS240A, Winter 2016

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



Speed (cycles):	½'s	1's	10's	100's	1,000,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost/bit:	highest	↔			lowest

- 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

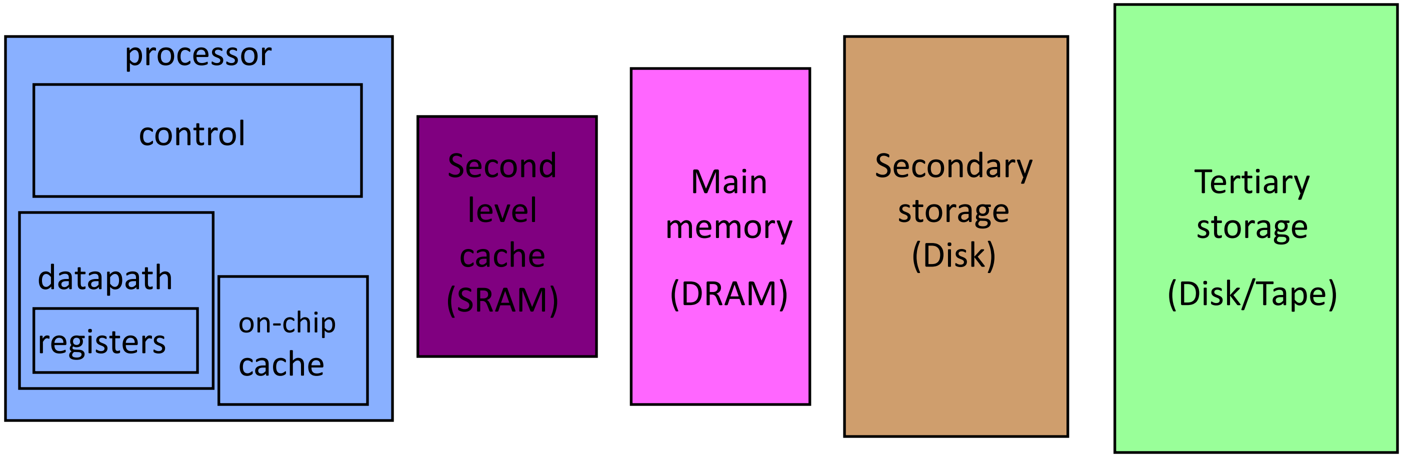
    (read, write, add, multiply, etc.)

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

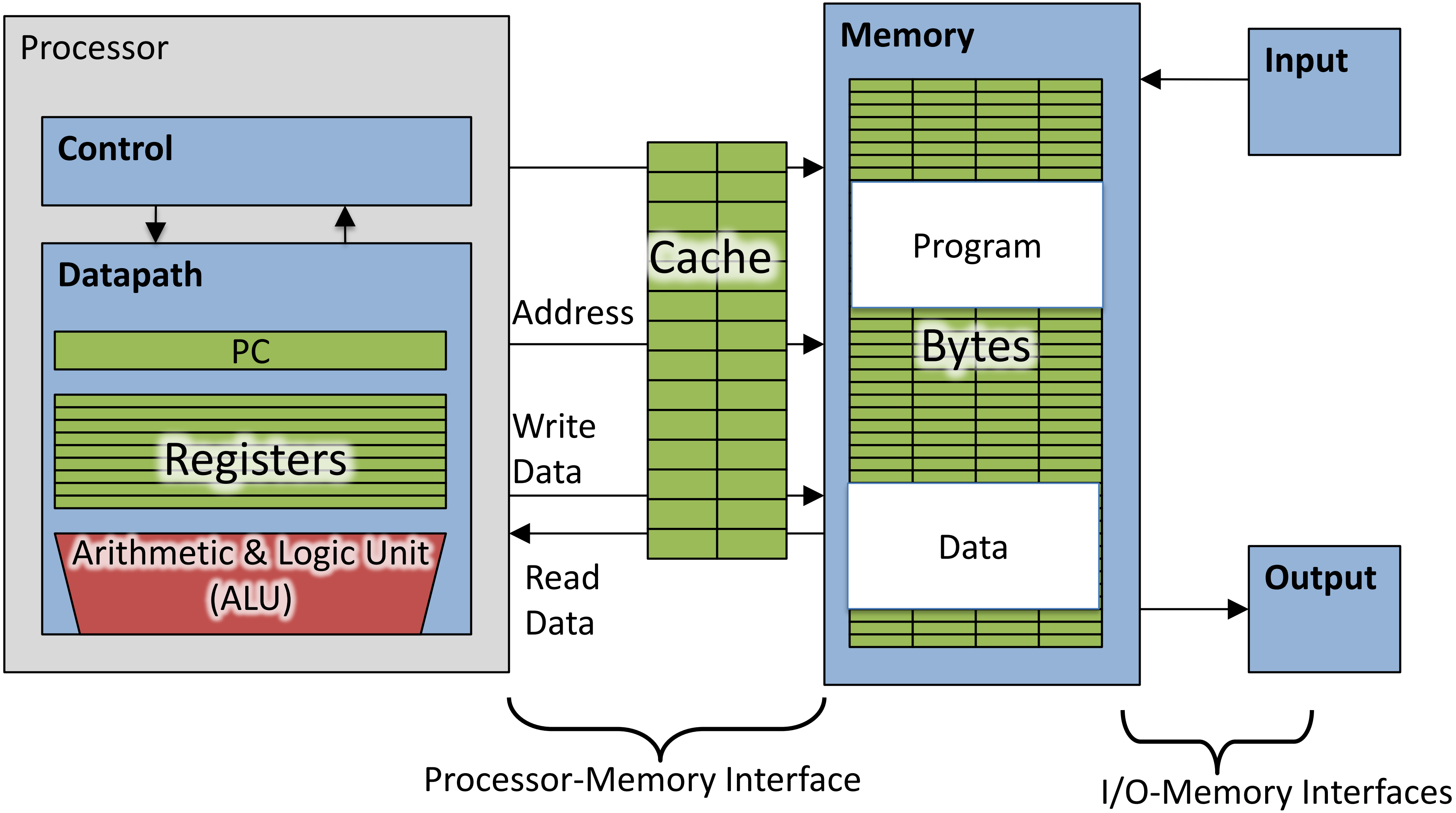
Memory Hierarchy

- 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	10ms	10sec
Size	KB	MB	GB	TB	PB

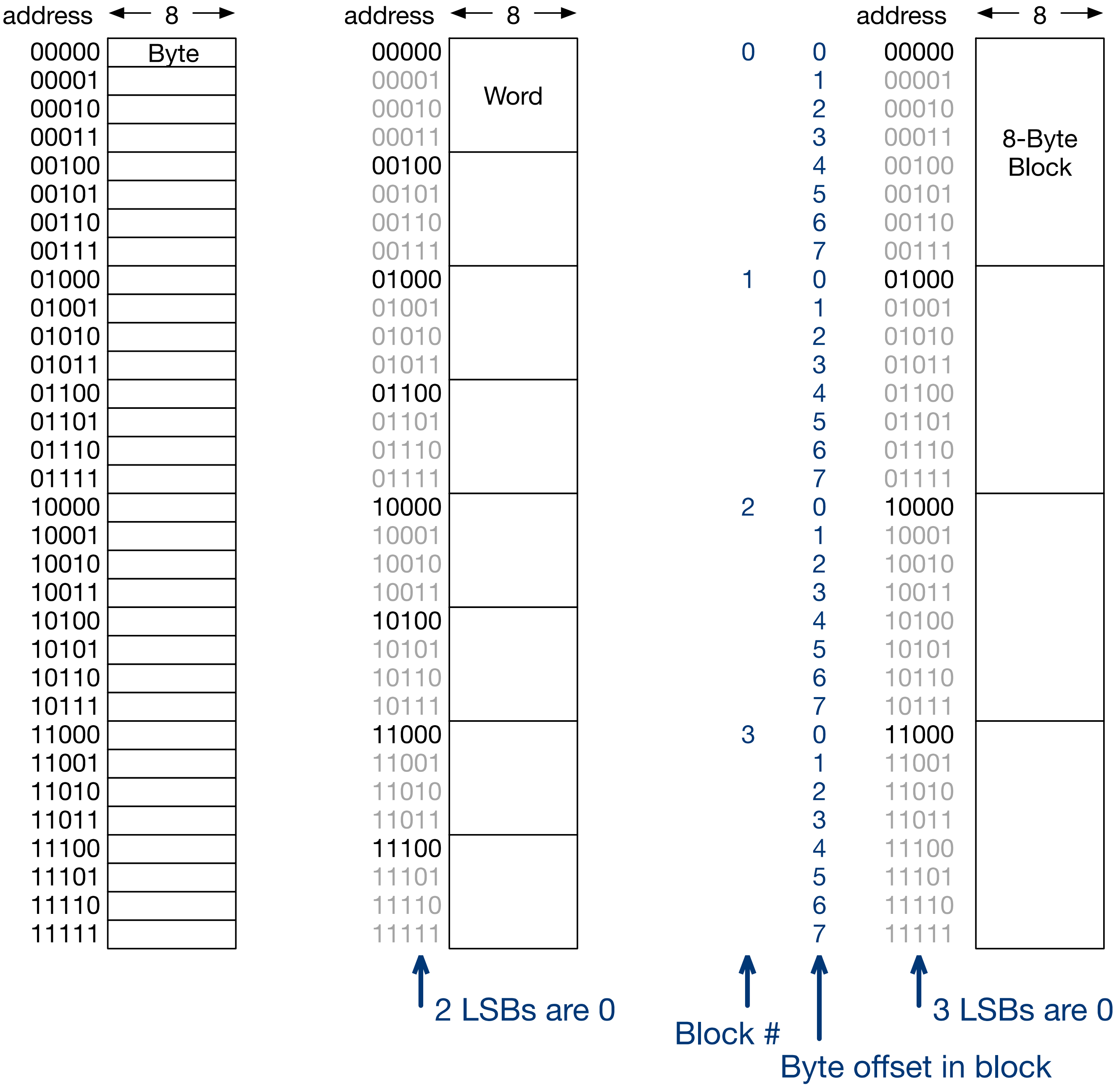
Review: Cache in Modern Computer Architecture



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



address	← 8 →	address	← 8 →			address	← 8 →
00000	Byte	00000	Word	0	0	00000	8-Byte Block
00001		00001			1	00001	
00010		00010			2	00010	
00011		00011			3	00011	
00100		00100			4	00100	
00101		00101			5	00101	
00110		00110			6	00110	
00111		00111			7	00111	
01000		01000		1	0	01000	
01001		01001			1	01001	
01010		01010			2	01010	
01011		01011			3	01011	
01100		01100			4	01100	
01101		01101			5	01101	
01110		01110			6	01110	
01111		01111			7	01111	
10000		10000		2	0	10000	
10001		10001			1	10001	
10010		10010			2	10010	
10011		10011			3	10011	
10100		10100			4	10100	
10101		10101			5	10101	
10110		10110			6	10110	
10111		10111			7	10111	
11000		11000		3	0	11000	
11001		11001			1	11001	
11010		11010			2	11010	
11011		11011			3	11011	
11100		11100			4	11100	
11101		11101			5	11101	
11110		11110			6	11110	
11111		11111			7	11111	

2 LSBs are 0

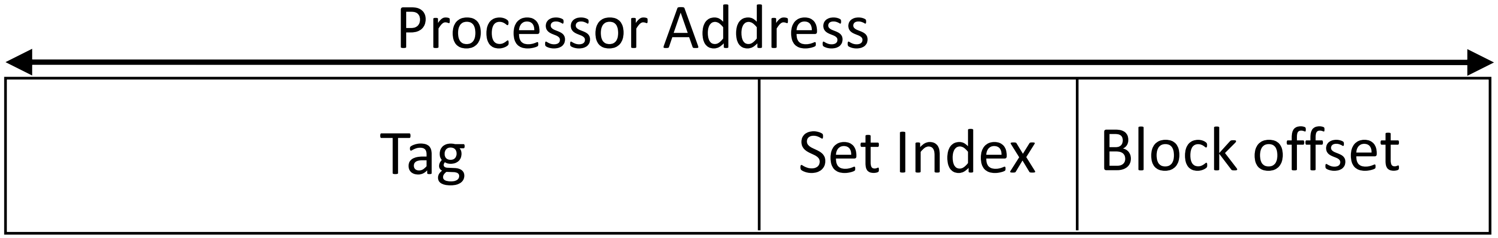
Block #

Byte offset in block

3 LSBs are 0

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



Processor Address

Tag

Set Index

Block offset

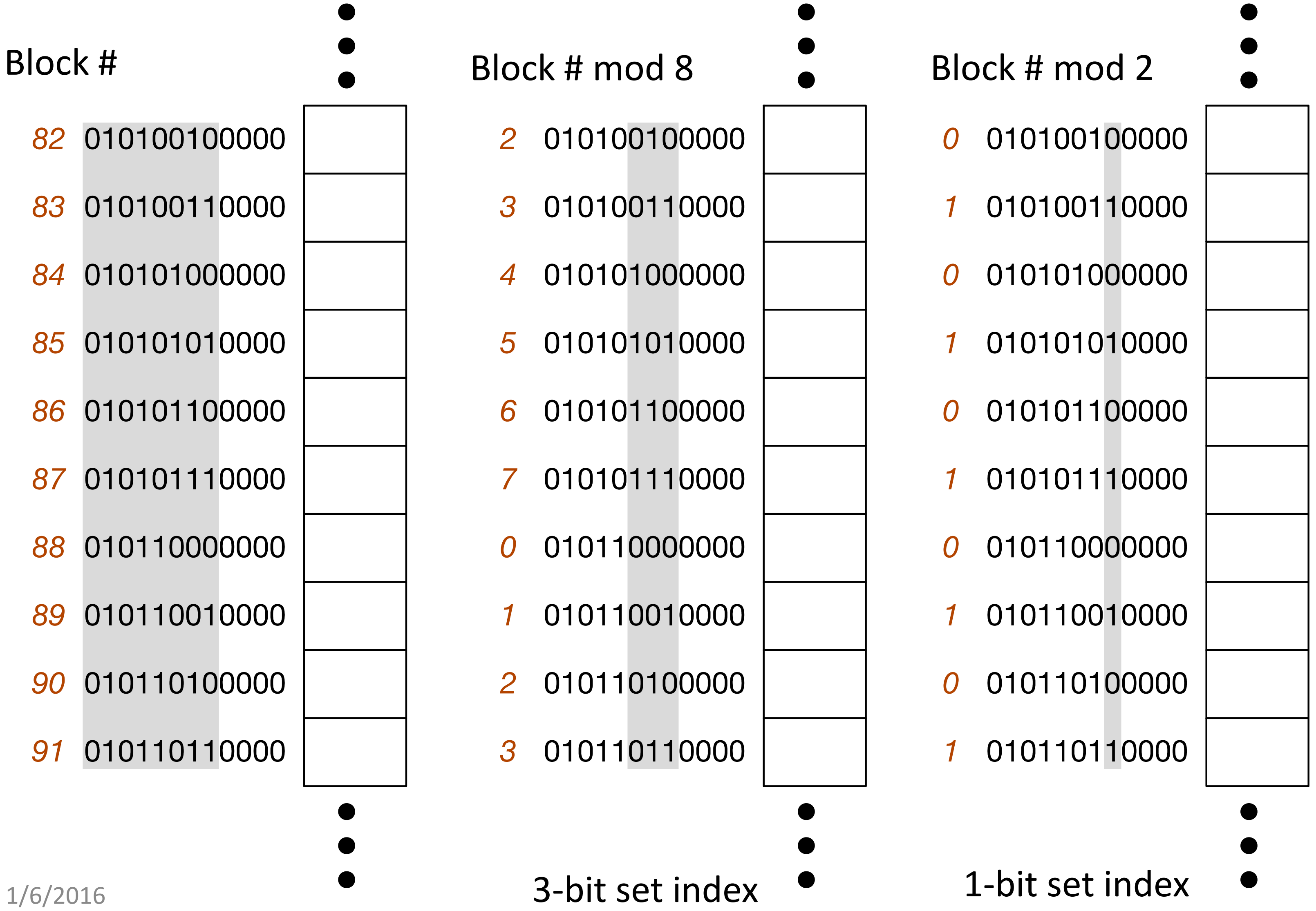
- Size of Tag = Address size – log(S) – log(B)

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.

Block number aliasing example

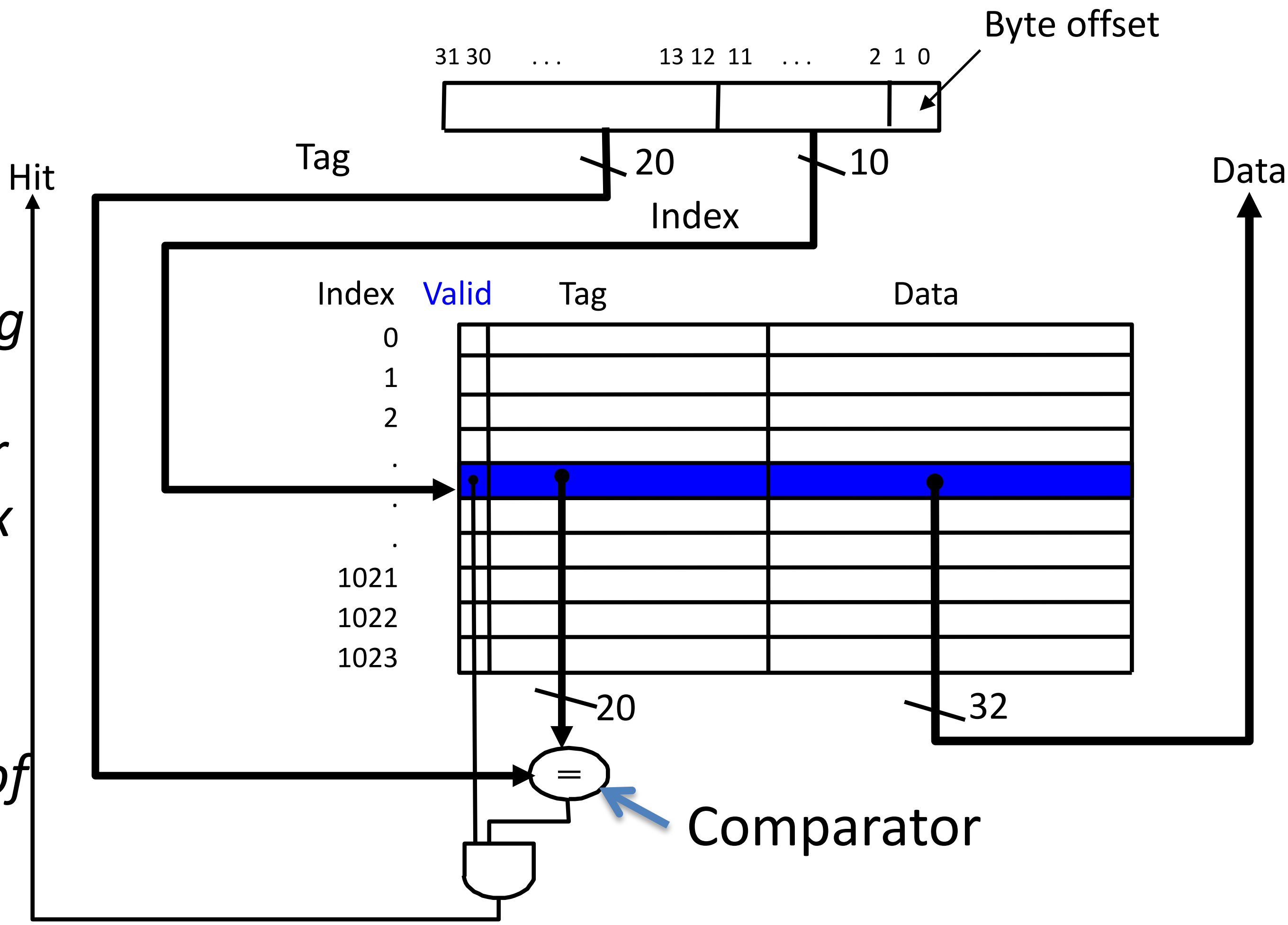
12-bit memory addresses, 16 Byte blocks



Block #		Block # mod 8		Block # mod 2	
82	010100100000	2	010100100000	0	010100100000
83	010100110000	3	010100110000	1	010100110000
84	010101000000	4	010101000000	0	010101000000
85	010101010000	5	010101010000	1	010101010000
86	010101100000	6	010101100000	0	010101100000
87	010101110000	7	010101110000	1	010101110000
88	010110000000	0	010110000000	0	010110000000
89	010110010000	1	010110010000	1	010110010000
90	010110100000	2	010110100000	0	010110100000
91	010110110000	3	010110110000	1	010110110000

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

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



Valid bit ensures something useful in cache for this index

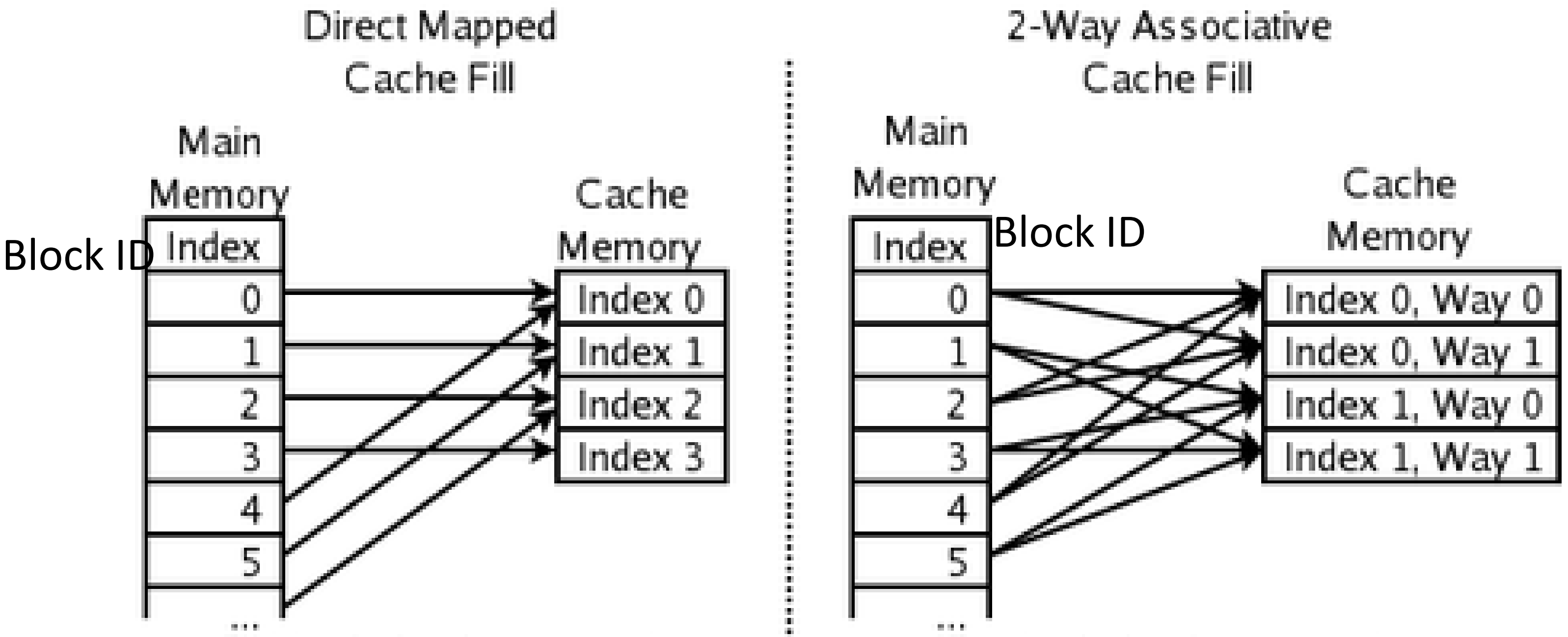
Compare Tag with upper part of Address to see if a Hit

Read data from cache instead of memory if a Hit

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

Cache Organizations

- “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

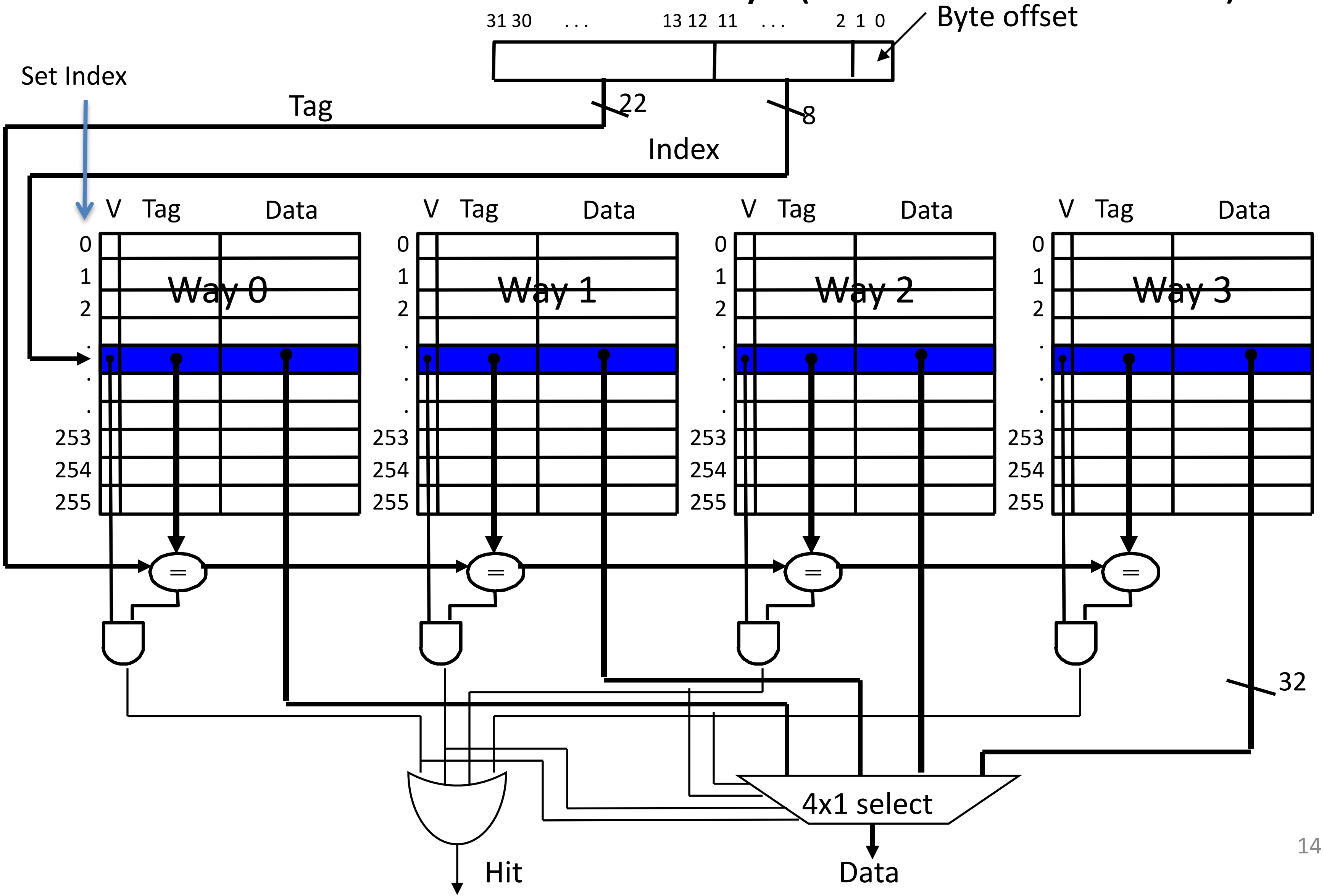


Each location in main memory can be cached by just one cache location.

Each location in main memory can be cached by one of two cache locations.

Four-Way Set-Associative Cache

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

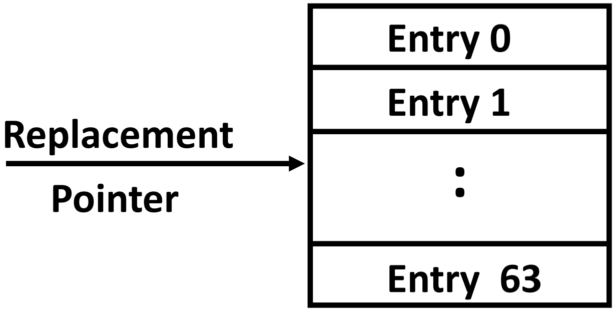


How to find if a data address in cache?

- 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 Stores with Write-Through

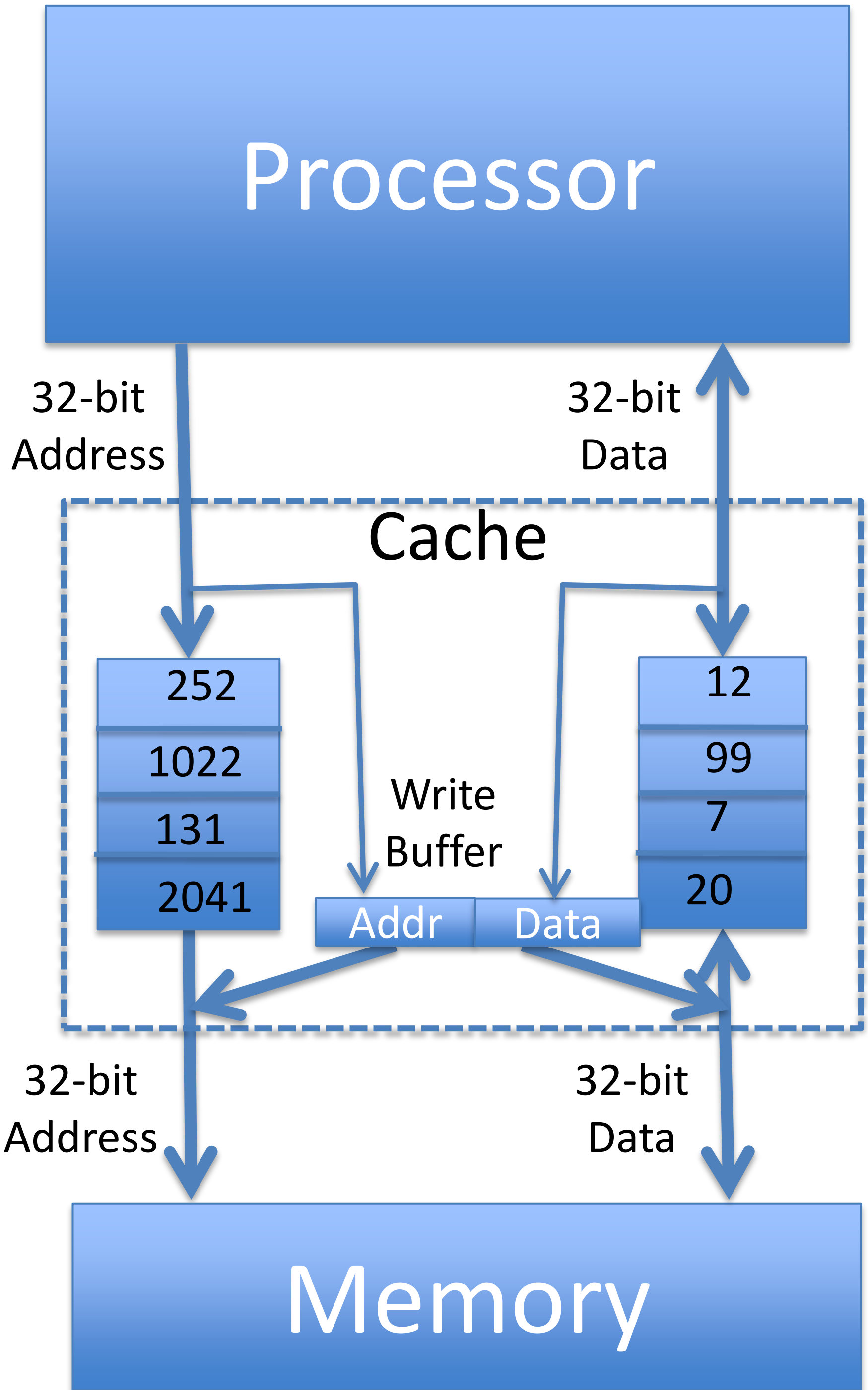
- Store instructions write to memory, changing values
- Need to make sure cache and memory have same values on writes: 2 policies

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

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?



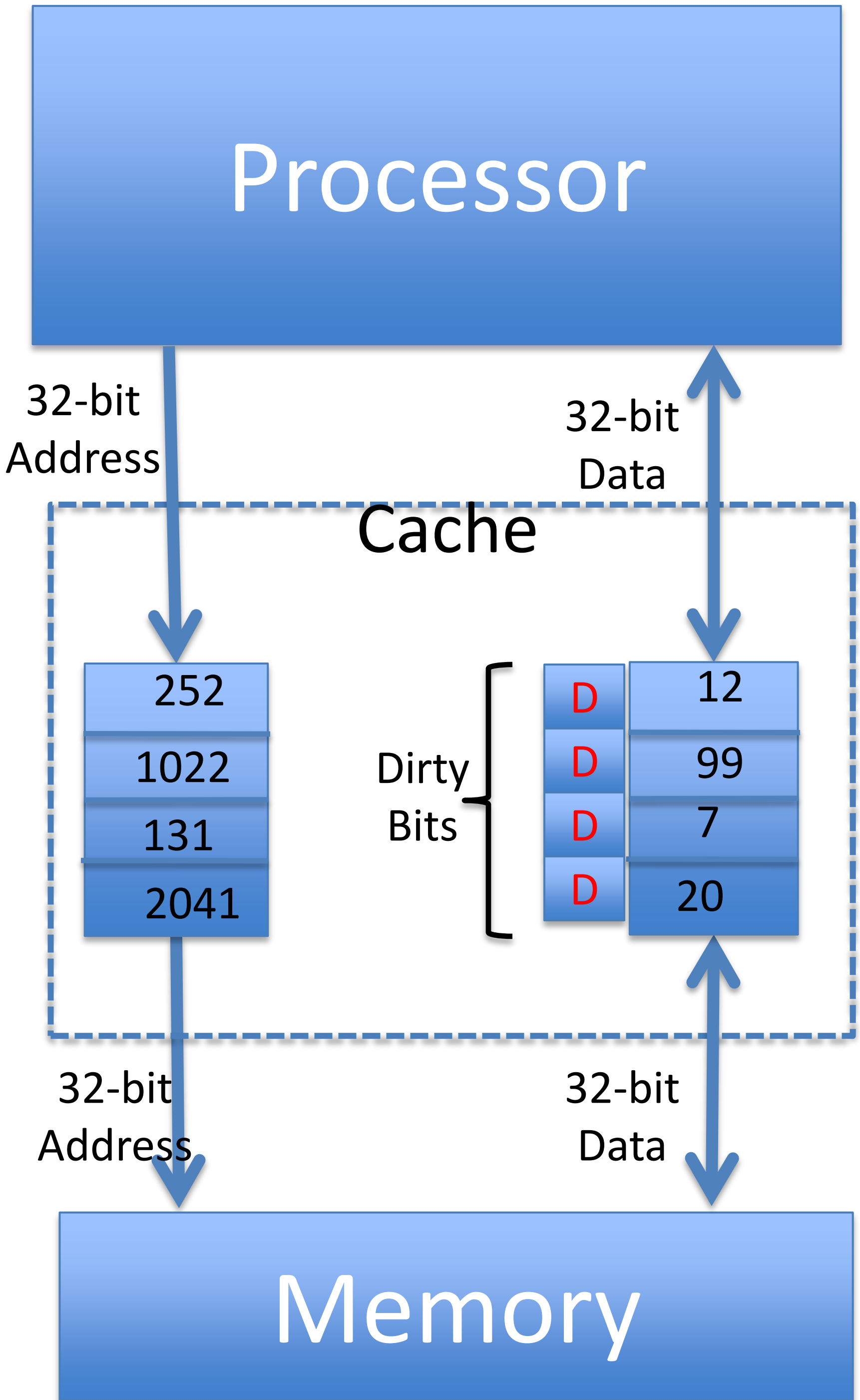
Handling Stores with Write-Back

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.