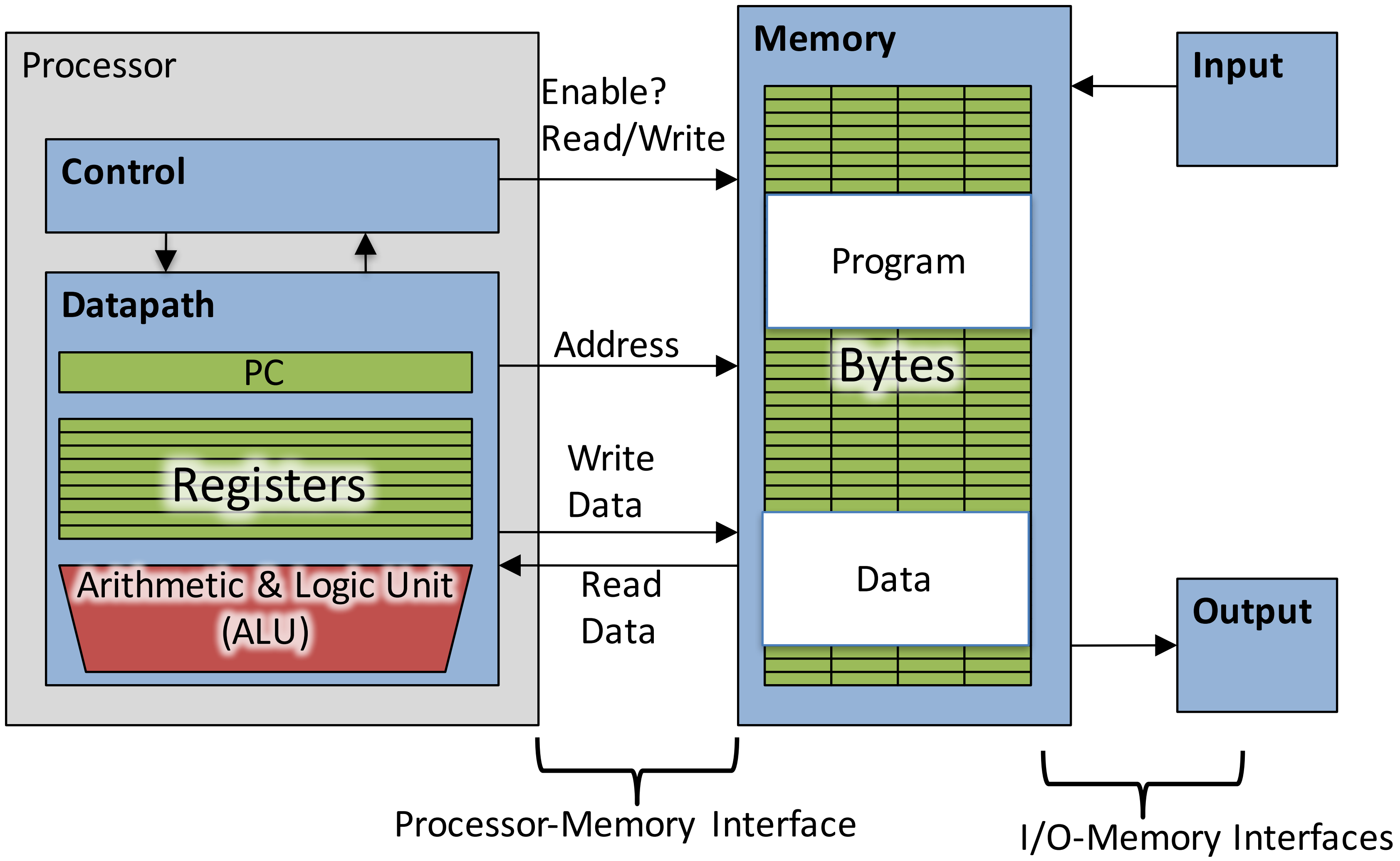
CS 61C: Great Ideas in Computer Architecture (Machine Structures) Caches Part 1

Instructors:

Nicholas Weaver & Vladimir Stojanovic

http://inst.eecs.berkeley.edu/~cs61c/

Components of a Computer

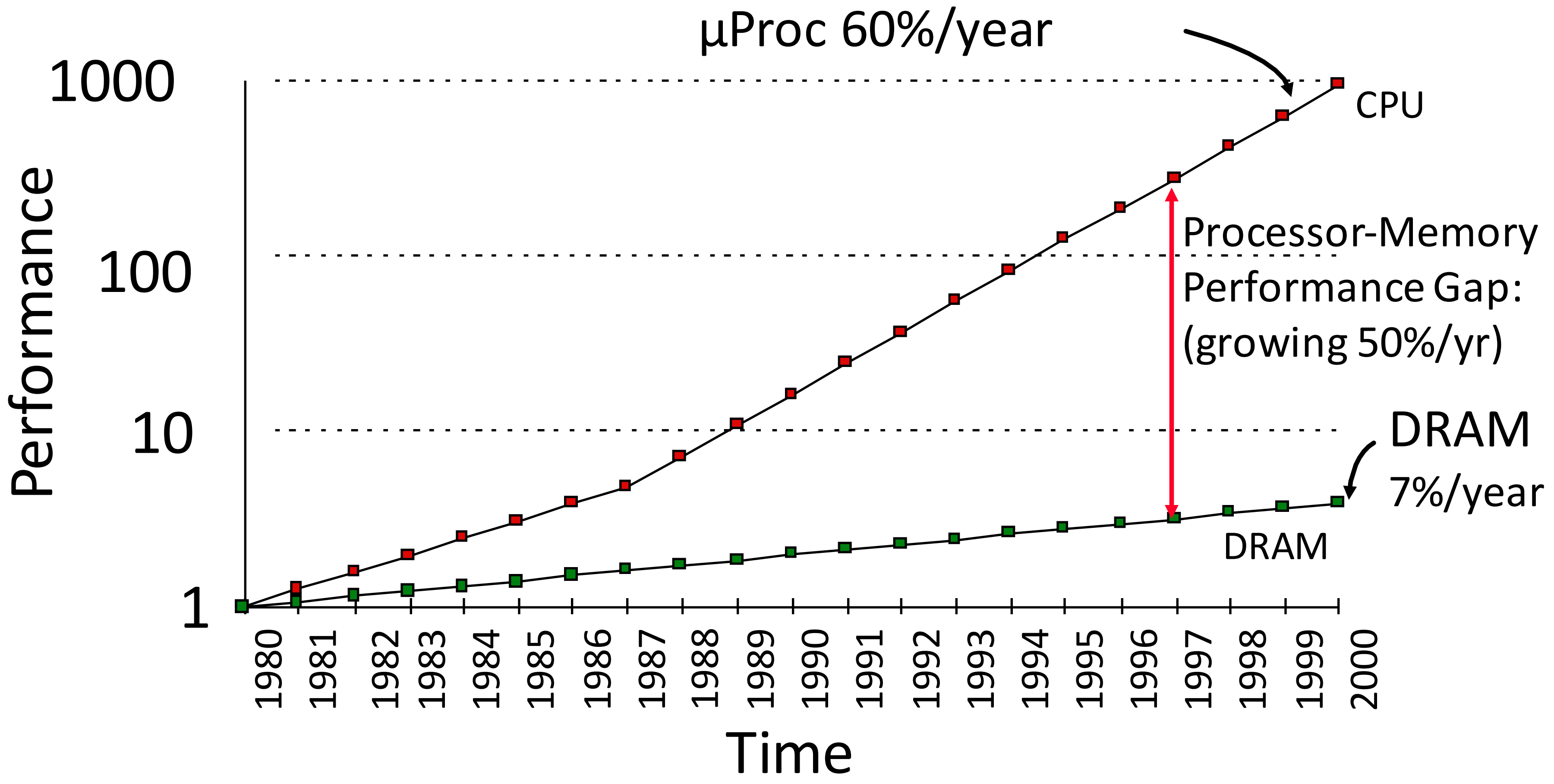


Problem: Large memories slow? Library Analogy

- Finding a book in a large library takes time
  - Takes time to search a large card catalog – (mapping title/author to index number)
  - Round-trip time to walk to the stacks and retrieve the desired book.
- Larger libraries makes both delays worse
- Electronic memories have the same issue, *plus* the technologies that we use to store an individual bit get slower as we increase density (SRAM versus DRAM versus Magnetic Disk)

However what we want is a large yet fast memory!

Processor-DRAM Gap (latency)



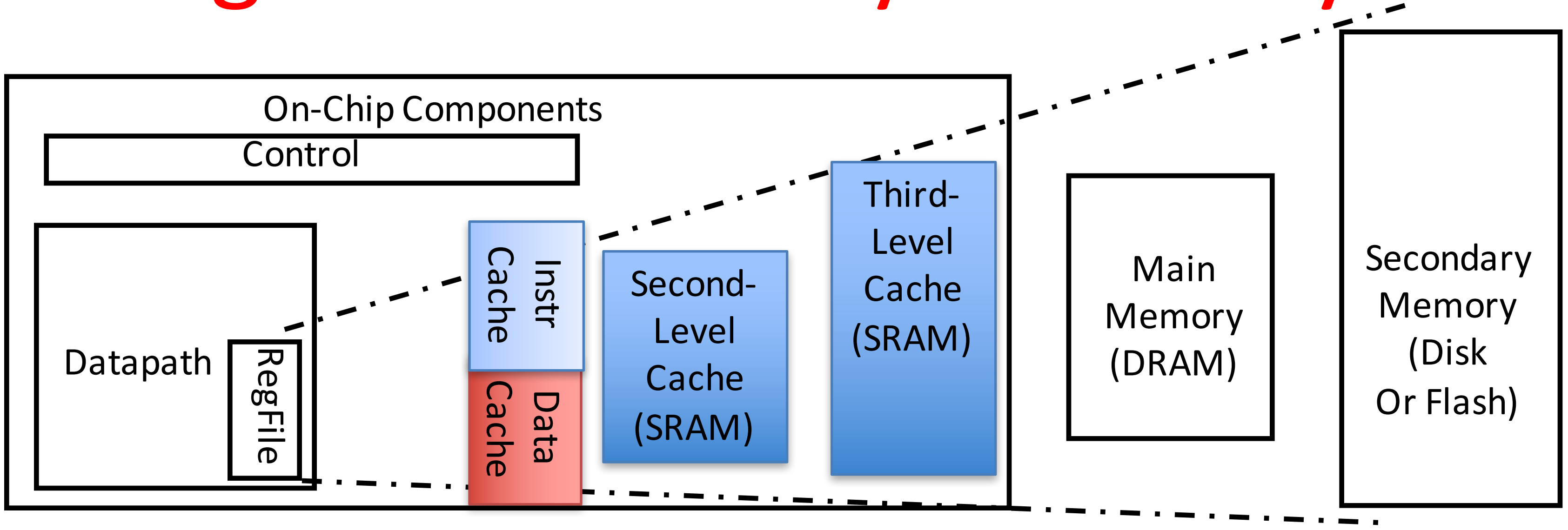
1980 microprocessor executes ~one instruction in same time as DRAM access
2015 microprocessor executes ~1000 instructions in same time as DRAM access

Slow DRAM access could have disastrous impact on CPU performance!

What to do: Library Analogy

- Want to write a report using library books
  - E.g., works of J.D. Salinger
- Go to Doe library, look up relevant books, fetch from stacks, and place on desk in library
- If need more, check them out and keep on desk
  - But don't return earlier books since might need them
- You hope this collection of ~10 books on desk enough to write report, despite 10 being only 0.00001% of books in UC Berkeley libraries

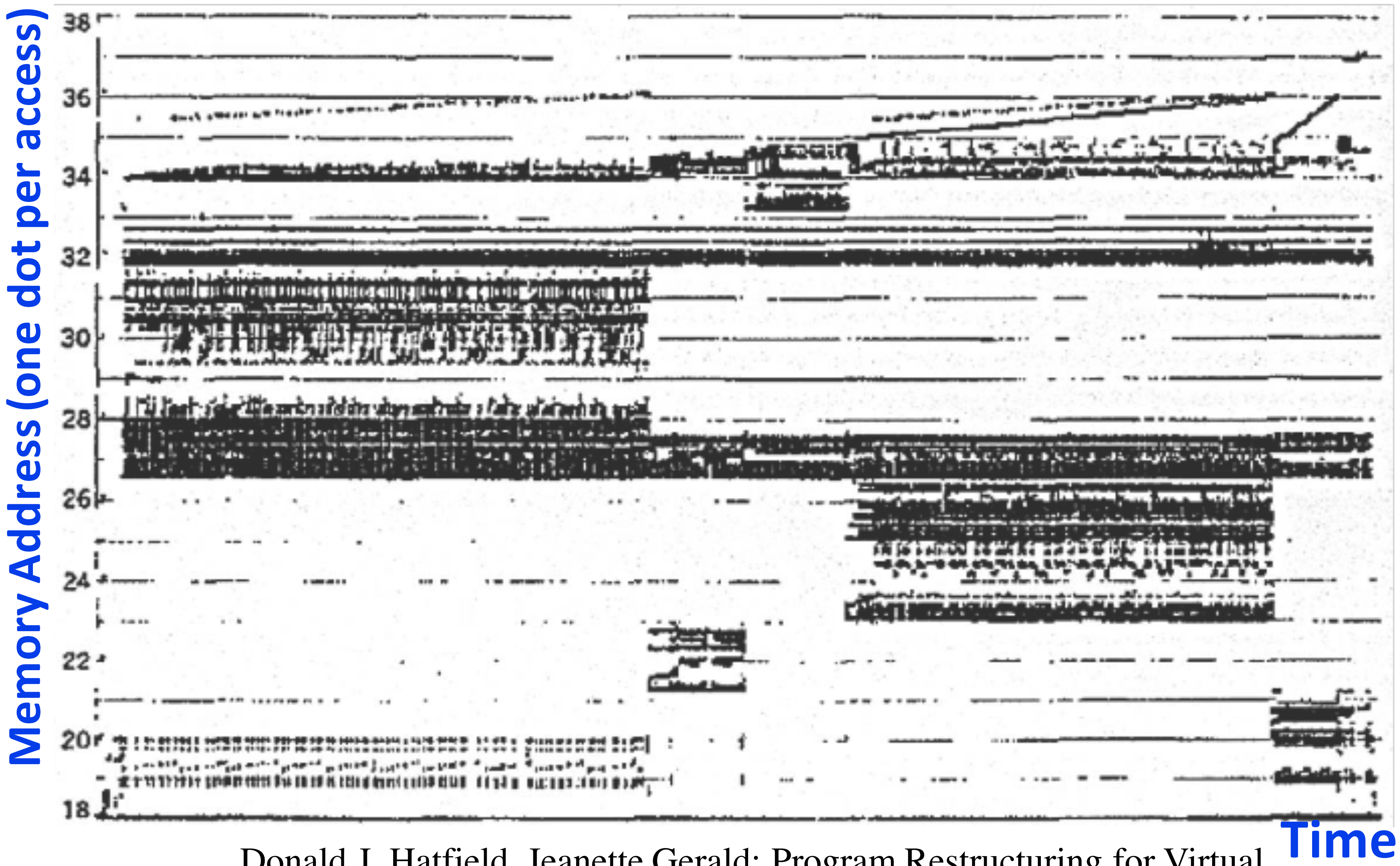
Big Idea: Memory Hierarchy



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

Real Memory Reference Patterns

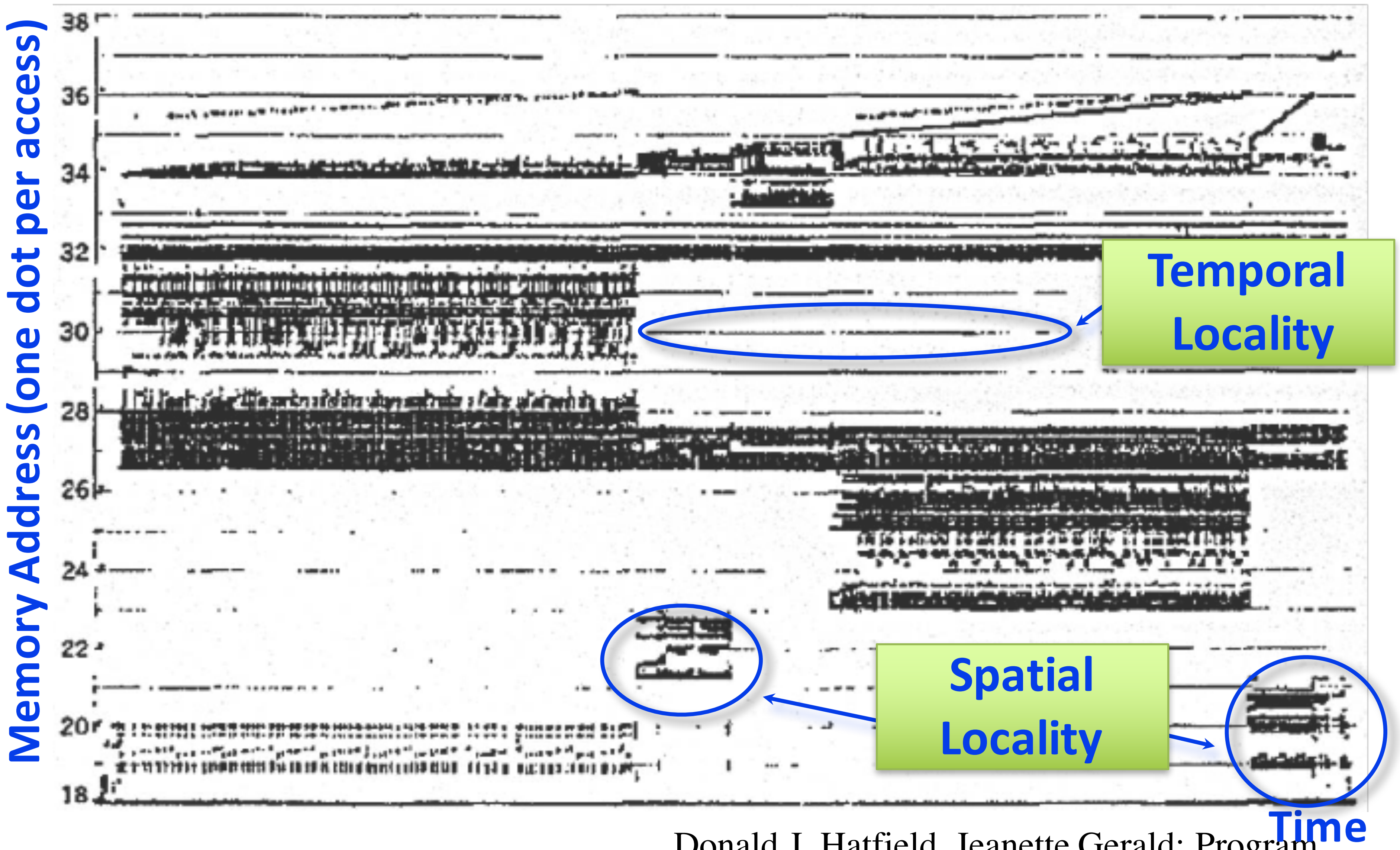


Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Big Idea: Locality

- *Temporal Locality* (locality in time)
  - Go back to same book on desktop multiple times
  - If a memory location is referenced, then it will tend to be referenced again soon
- *Spatial Locality* (locality in space)
  - When go to book shelf, pick up multiple books on J.D. Salinger since library stores related books together
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon

Memory Reference Patterns

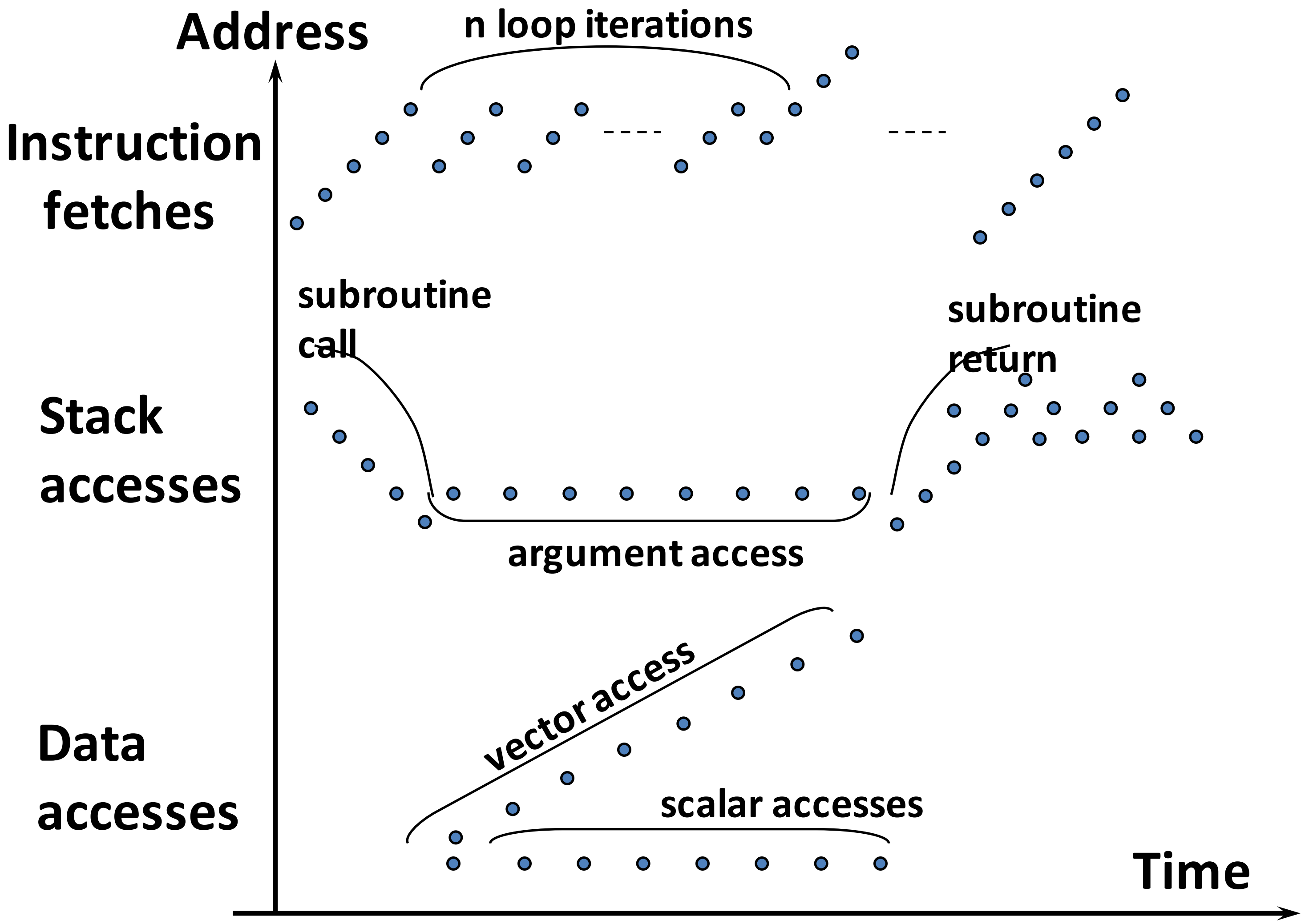


Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Principle of Locality

- *Principle of Locality*: Programs access small portion of address space at any instant of time (spatial locality) and repeatedly access that portion (temporal locality)
- What program structures lead to temporal and spatial locality in instruction accesses?
- In data accesses?

Memory Reference Patterns



Cache Philosophy

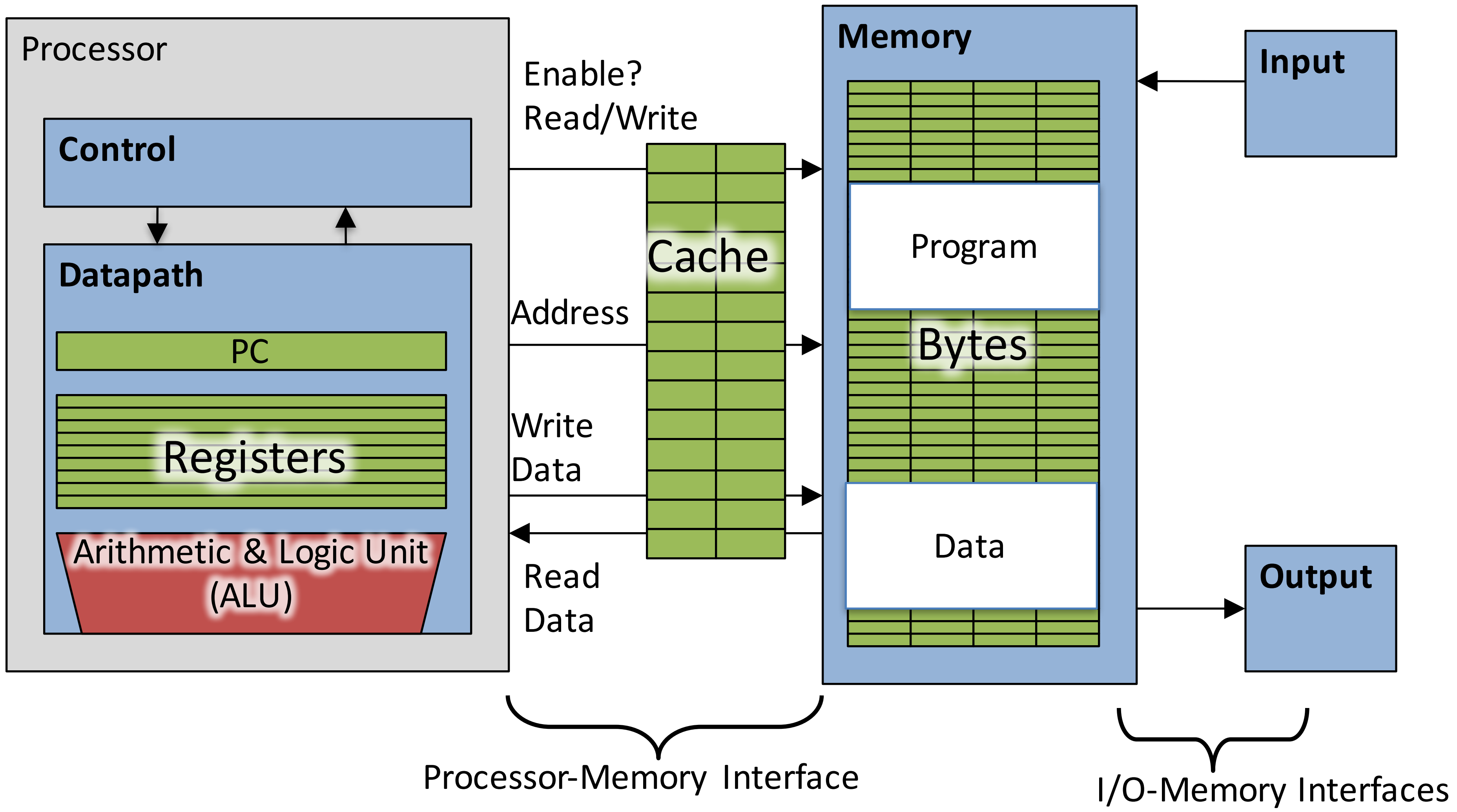
- Programmer-invisible hardware mechanism to give illusion of speed of fastest memory with size of largest memory
  - Works fine even if programmer has no idea what a cache is
    - However, performance-oriented programmers today sometimes “reverse engineer” cache design to design data structures to match cache

Memory Access without Cache

- Load word instruction: `lw $t0,0($t1)`
- $t1 contains 0x12F0, Memory[0x12F0] = 99

1. Processor issues address 0x12F0 to Memory
2. Memory reads word at address 0x12F0 (99)
3. Memory sends 99 to Processor
4. Processor loads 99 into register $t0

Adding Cache to Computer



Memory Access with Cache

- Load word instruction: `lw $t0,0($t1)`
- $t1 contains 0x12F0, Memory[0x12F0] = 99
- With cache: Processor issues address 0x12F0 to Cache
  1. Cache checks to see if has copy of data at address 0x12F0
     - 2a. If finds a match (Hit): cache reads 99, sends to processor
     - 2b. No match (Miss): cache sends address 0x12F0 to Memory
       - I. Memory reads 99 at address 0x12F0
       - II. Memory sends 99 to Cache
       - III. Cache replaces word which can store 0x12F0 with new 99
       - IV. Cache sends 99 to processor
  2. Processor loads 99 into register $t0

Administrivia

- Fill In

Clicker Question…

- Consider the following statements
  - 1: The J instructions in MIPS have a delay slot
  - 2: JAL records PC + 4 into $ra on MIPS with a delay slot
  - 3: The location where to jump to on a JR is known in the ID stage
- Which are true?
  - A) None
  - B) 1, 3
  - C) 1, 2
  - D) 2, 3
  - E) 1, 3

Cache "Tags"

- Need way to tell if have copy of location in memory so that can decide on hit or miss
- On cache miss, put memory address of block as "tag" of cache block

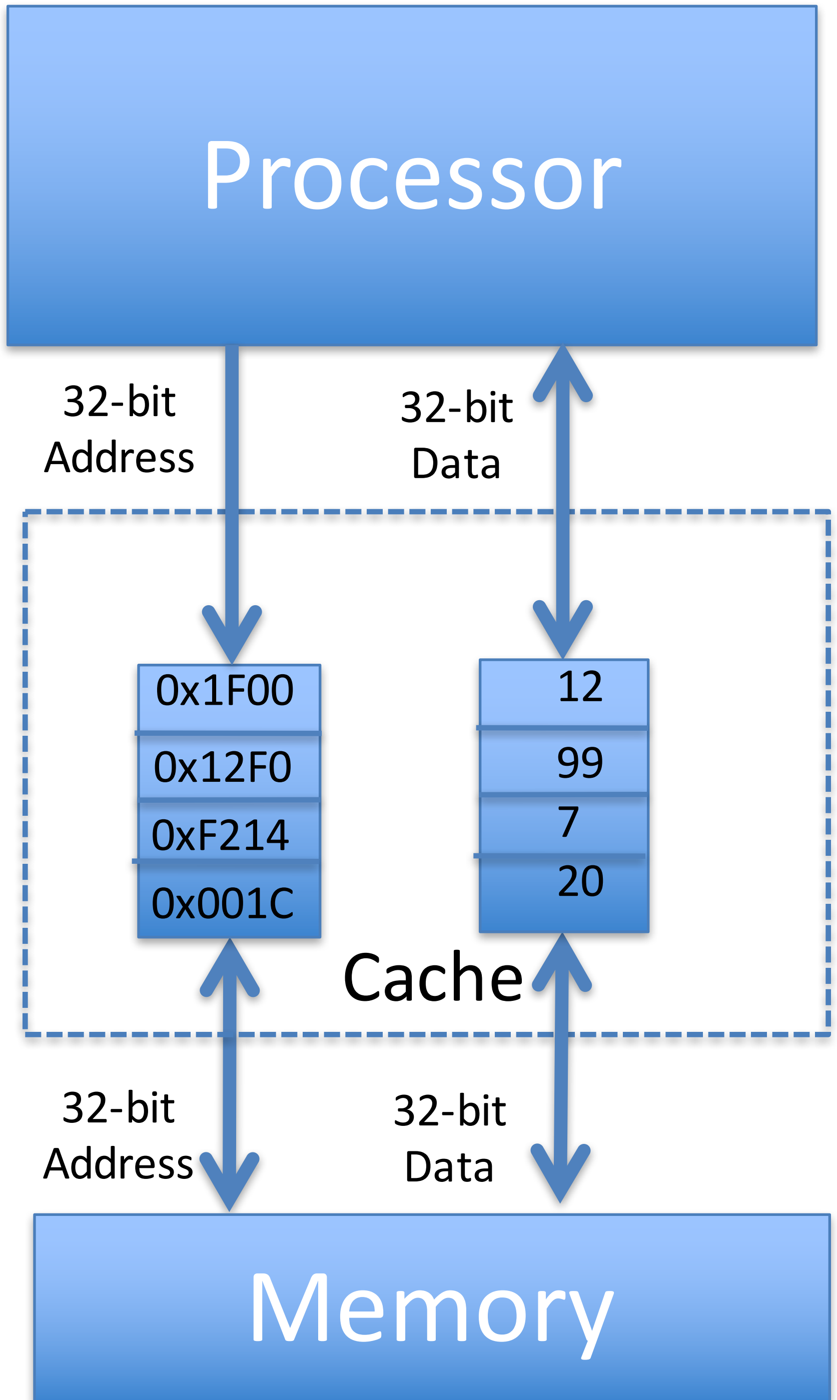
  1022 placed in tag next to data from memory (99)

Tag	Data
0x1F00	12
0x12F0	99
0xF214	7
0x001c	20

From earlier instructions

Anatomy of a 16 Byte Cache, 4 Byte Block

- Operations:
  1. Cache Hit
  2. Cache Miss
  3. Refill cache from memory
- Cache needs Address Tags to decide if Processor Address is a Cache Hit or Cache Miss
  - Compares all 4 tags
  - "Fully Associative cache" Any tag can be in any location so you have to check them all



Cache Replacement

- Suppose processor now requests location 0x050C, which contains 11?
- Doesn't match any cache block, so must "evict" one resident block to make room
  - Which block to evict?
- Replace "victim" with new memory block at address 0x050C

Tag	Data
0x1F00	12
0x12F0	99
0x050C	11
0x001C	20

Block Must be Aligned in Memory

- Word blocks are aligned, so binary address of all words in cache always ends in 00_{two}
- How to take advantage of this to save hardware and energy?
- Don't need to compare last 2 bits of 32-bit byte address (comparator can be narrower)

=> Don't need to store last 2 bits of 32-bit byte address in Cache Tag (Tag can be narrower)

Anatomy of a 32B Cache, 8B Block

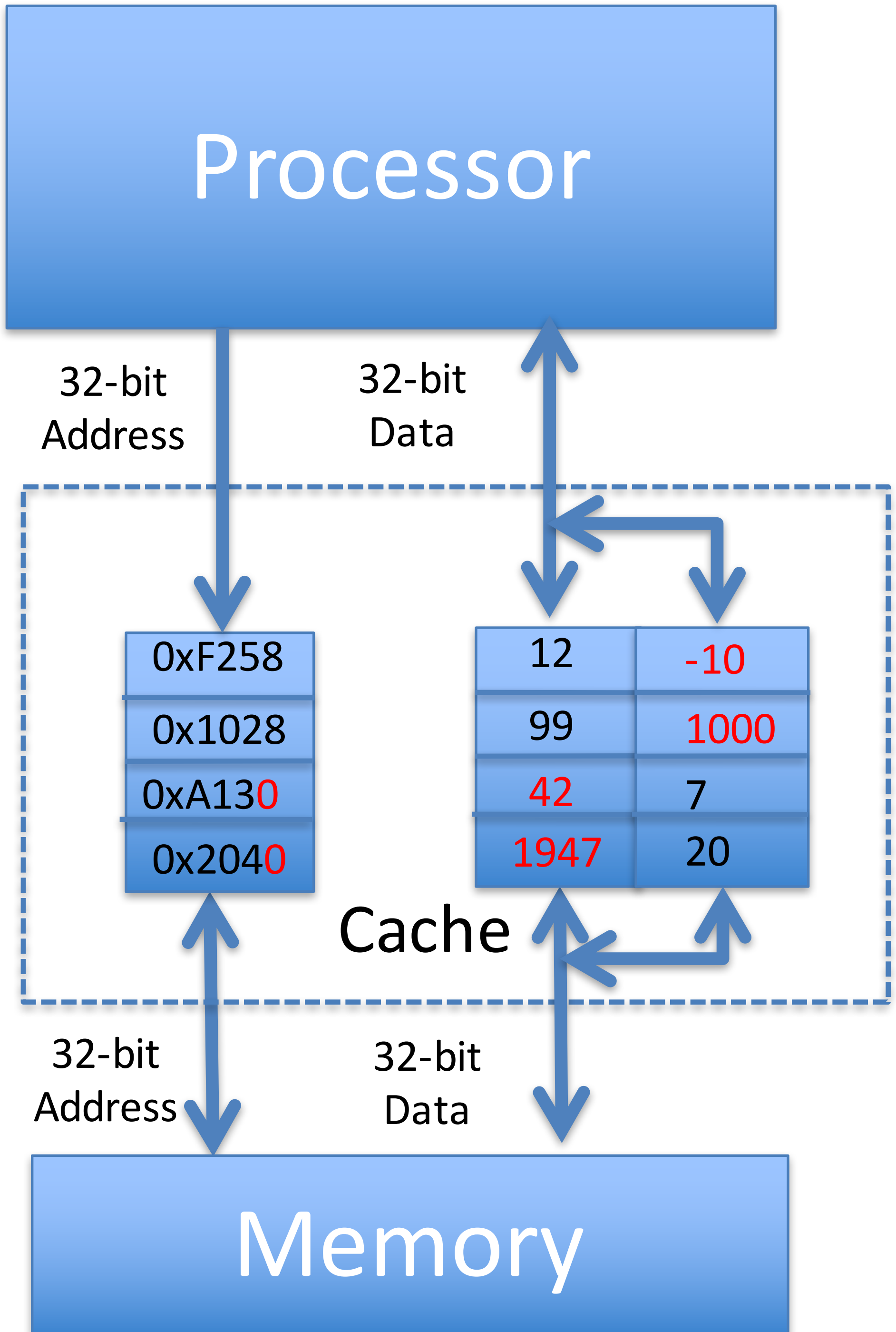
- Blocks must be aligned in pairs, otherwise could get same word twice in cache

⇒Tags only have even-numbered words

⇒ Last 3 bits of address always 000_{two}

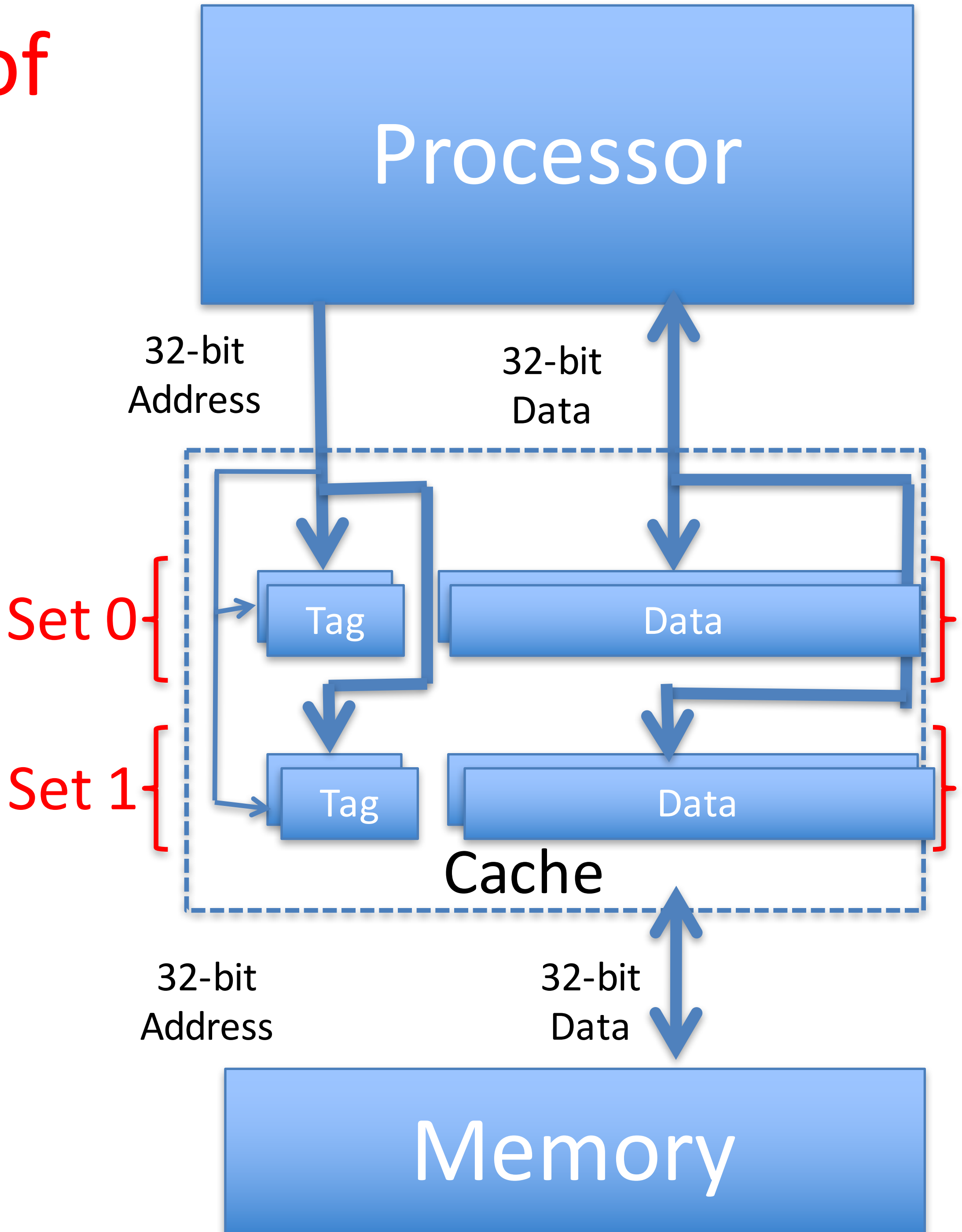
⇒Tags, comparators can be narrower

- Can get hit for either word in block



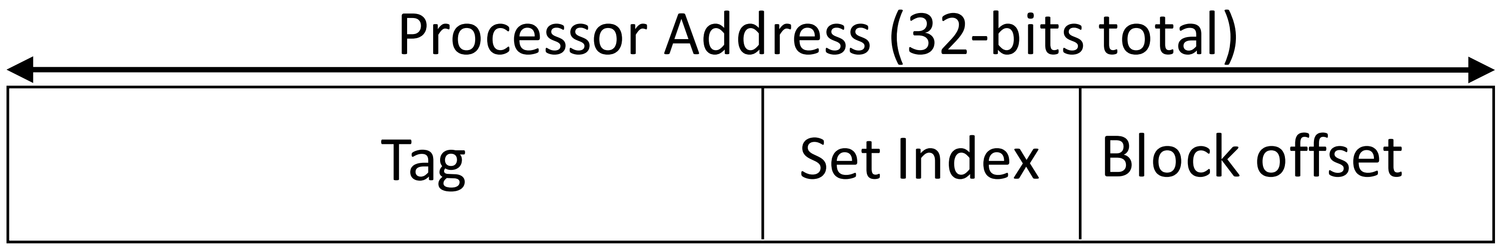
Hardware Cost of Cache

- Need to compare every tag to the Processor address
- Comparators are expensive
- Optimization: use 2 “sets” of data with a total of only 2 comparators
- 1 Address bit selects which set (ex: even and odd set)
- Compare only tags from selected set
- Generalize to more sets



Processor Address Fields used by Cache Controller

- Block Offset: Byte address within block
- Set Index: Selects which set
- Tag: Remaining portion of processor address



- Size of Index = log2 (number of sets)
- Size of Tag = Address size – Size of Index – log2 (number of bytes/block)

What is limit to number of sets?

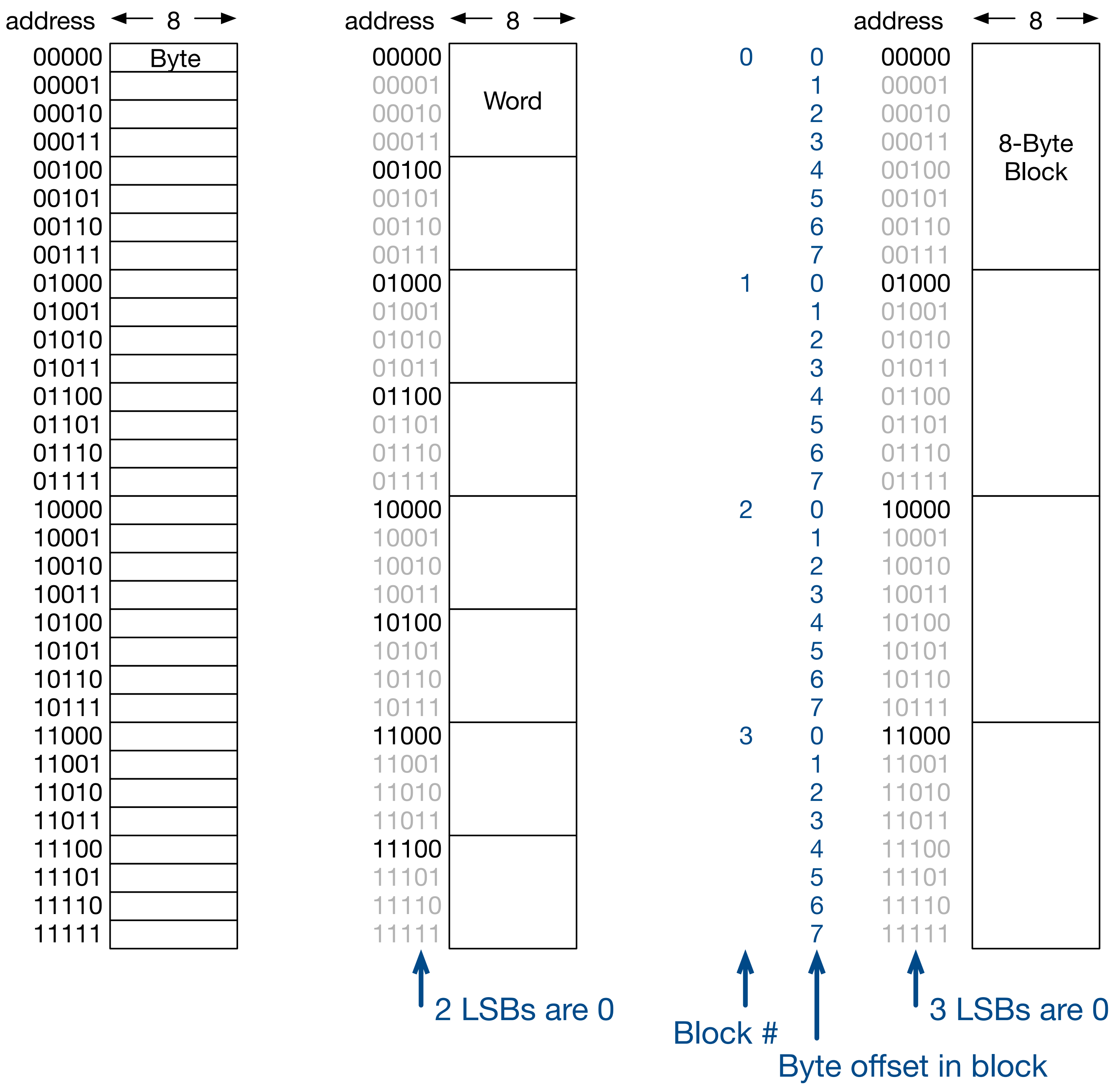
- For a given total number of blocks, we can save more comparators if have more than 2 sets
- Limit: As Many Sets as Cache Blocks => only one block per set – only needs one comparator!
- Called "Direct-Mapped" Design

Tag	Index	Block offset

Cache Names for Each Organization

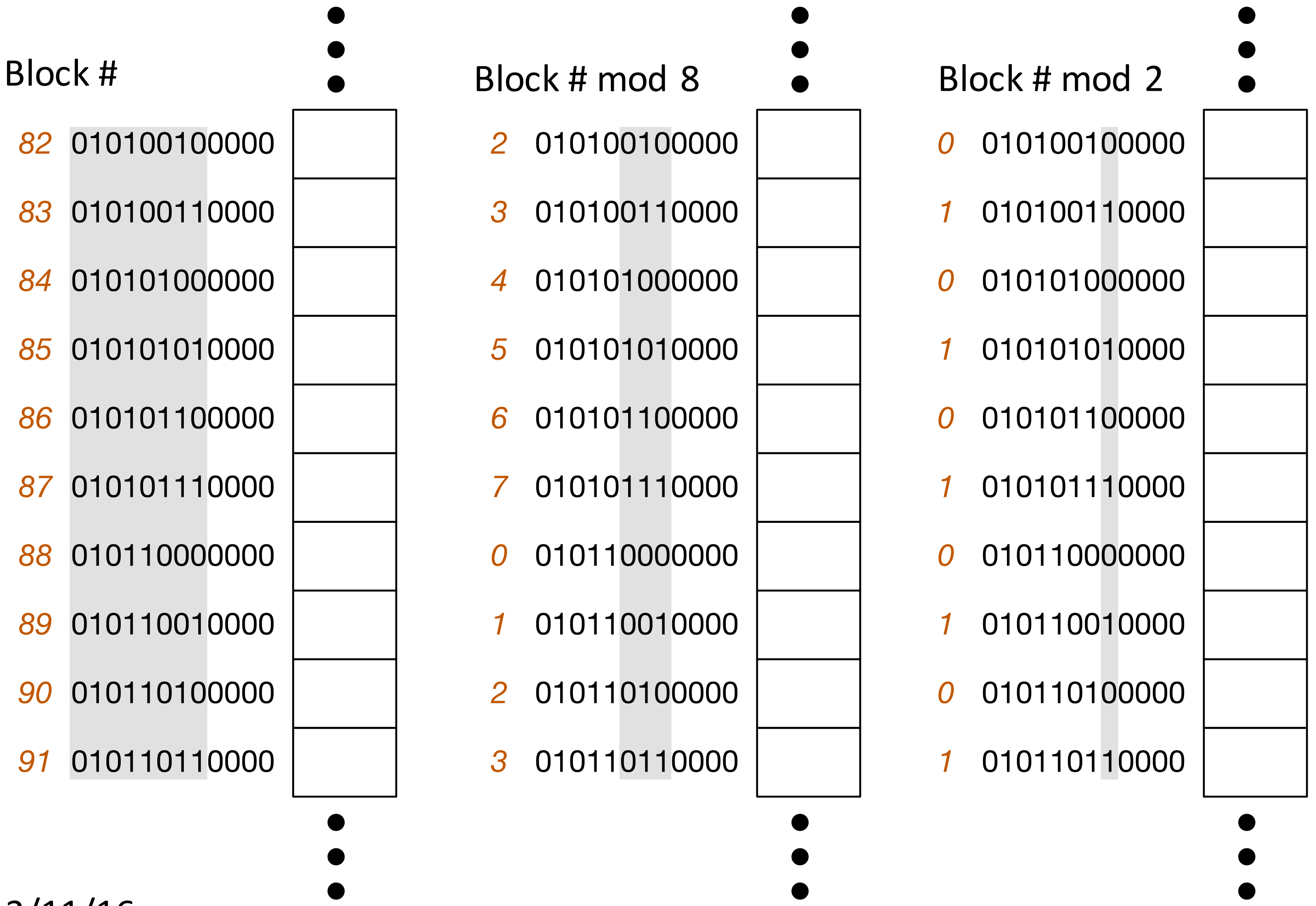
- "Fully Associative": Block can go anywhere
  - First design in lecture
  - Note: No Index field, but 1 comparator/block
- "Direct Mapped": Block goes one place
  - Note: Only 1 comparator
  - Number of sets = number blocks
- "N-way Set Associative": N places for a block
  - Number of sets = number of blocks / N
  - N comparators
  - ***Fully Associative: N = number of blocks***
  - ***Direct Mapped: N = 1***

Memory Block-addressing example



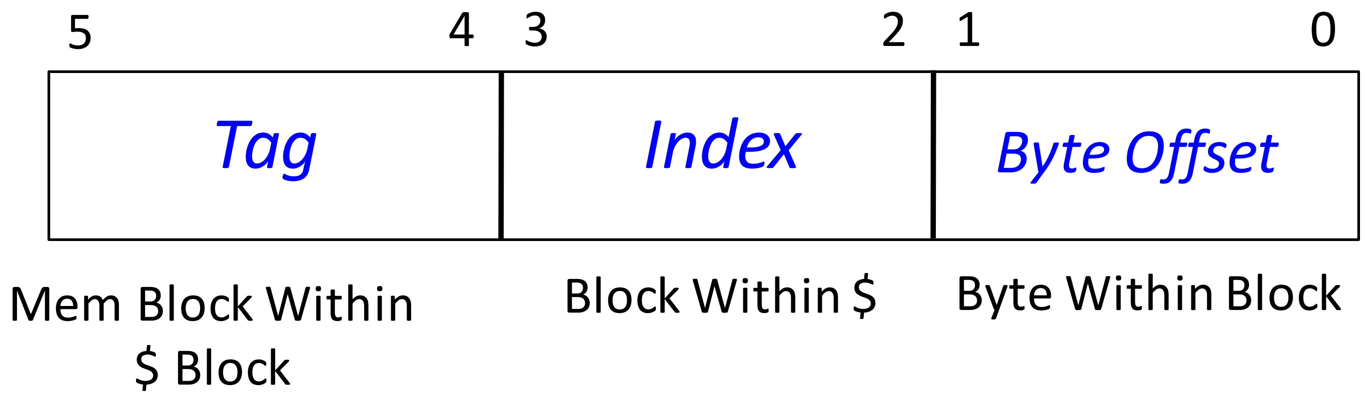
Block number aliasing example

12-bit memory addresses, 16 Byte blocks



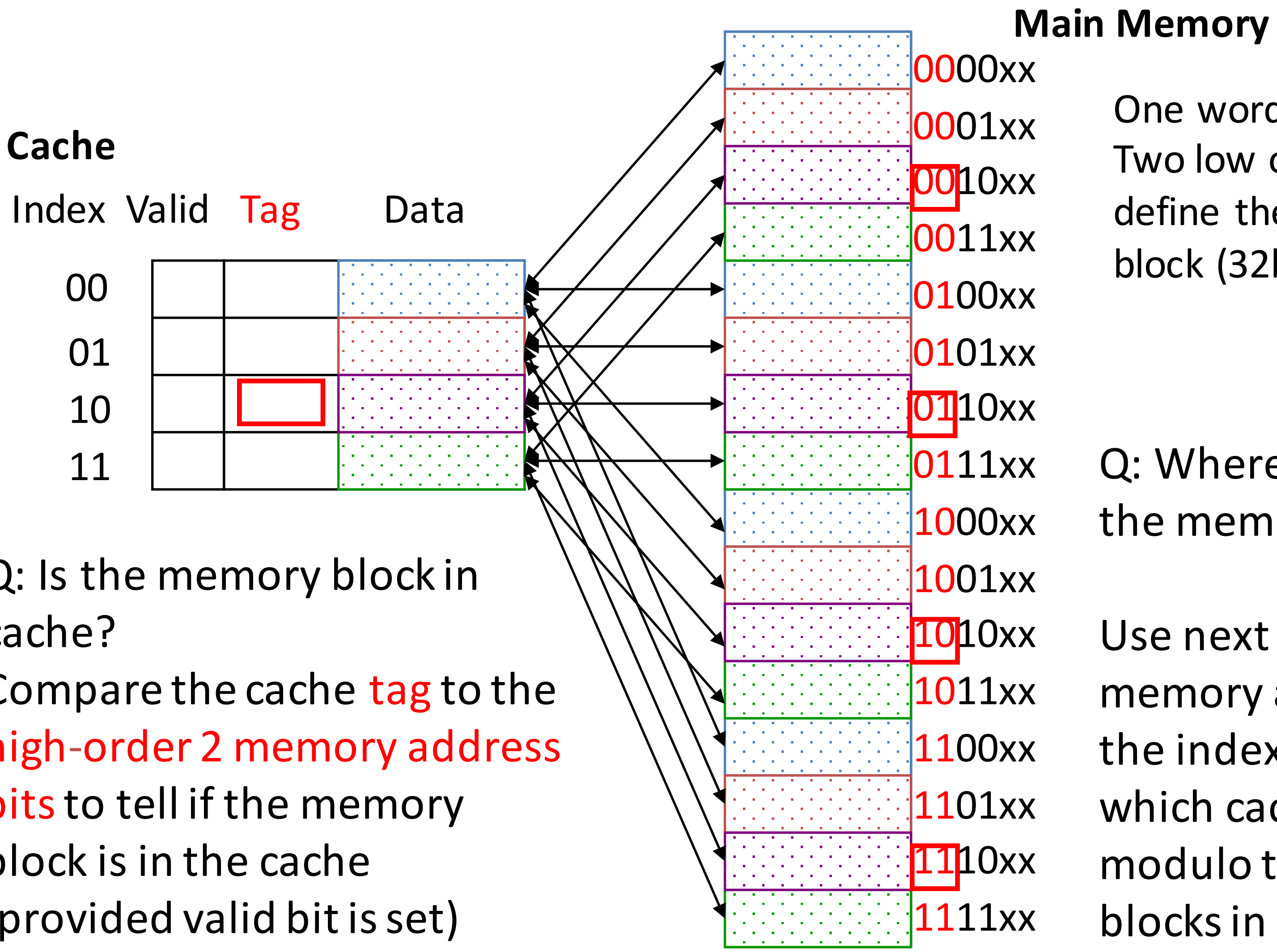
Block #		Block # mod 8		Block # mod 2	
82	010100100000	*2*	010100100000	*0*	010100100000
83	010100110000	*3*	010100110000	*1*	010100110000
84	010101000000	*4*	010101000000	*0*	010101000000
85	010101010000	*5*	010101010000	*1*	010101010000
86	010101100000	*6*	010101100000	*0*	010101100000
87	010101110000	*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 Ex: Mapping a 6-bit Memory Address



- In example, block size is 4 bytes (1 word)
- Memory and cache blocks always the same size, unit of transfer between memory and cache
- # Memory blocks >> # Cache blocks
  - 16 Memory blocks = 16 words = 64 bytes => 6 bits to address all bytes
  - 4 Cache blocks, 4 bytes (1 word) per block
  - 4 Memory blocks map to each cache block
- Memory block to cache block, aka *index*: middle two bits
- Which memory block is in a given cache block, aka *tag*: top two bits

Caching: A Simple First Example



One word blocks
Two low order bits (xx) define the byte in the block (32b words)

Q: Where in the cache is the mem block?

Use next 2 low-order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

Q: Is the memory block in cache?
Compare the cache tag to the high-order 2 memory address bits to tell if the memory block is in the cache (provided valid bit is set)

One More Detail: Valid Bit

- When start a new program, cache does not have valid information for this program
- Need an indicator whether this tag entry is valid for this program
- Add a “valid bit” to the cache tag entry

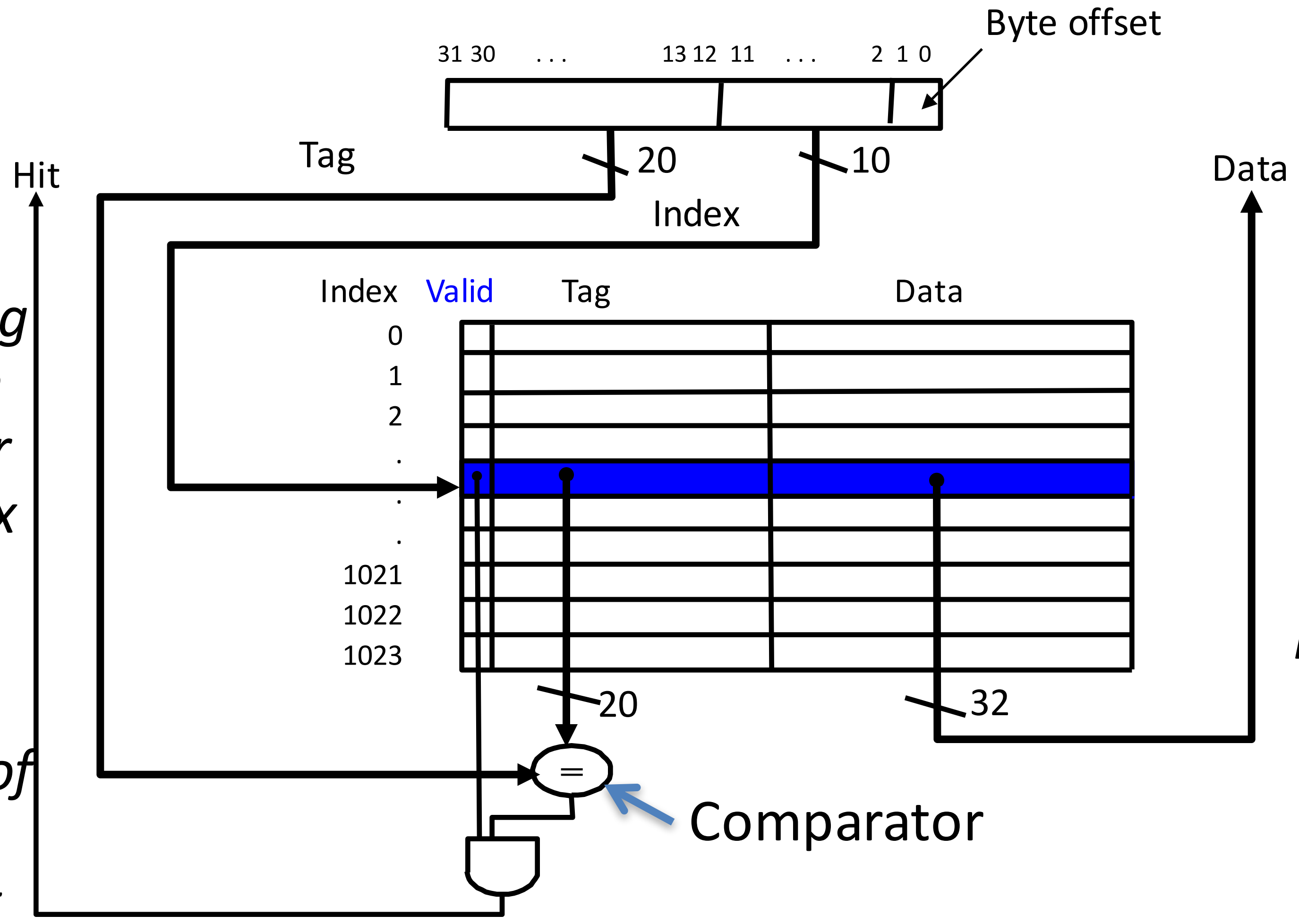
  0 => cache miss, even if by chance, address = tag

  1 => cache hit, if processor address = tag

Direct-Mapped Cache Example

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

Valid bit ensures something useful in cache for this index



Comparator

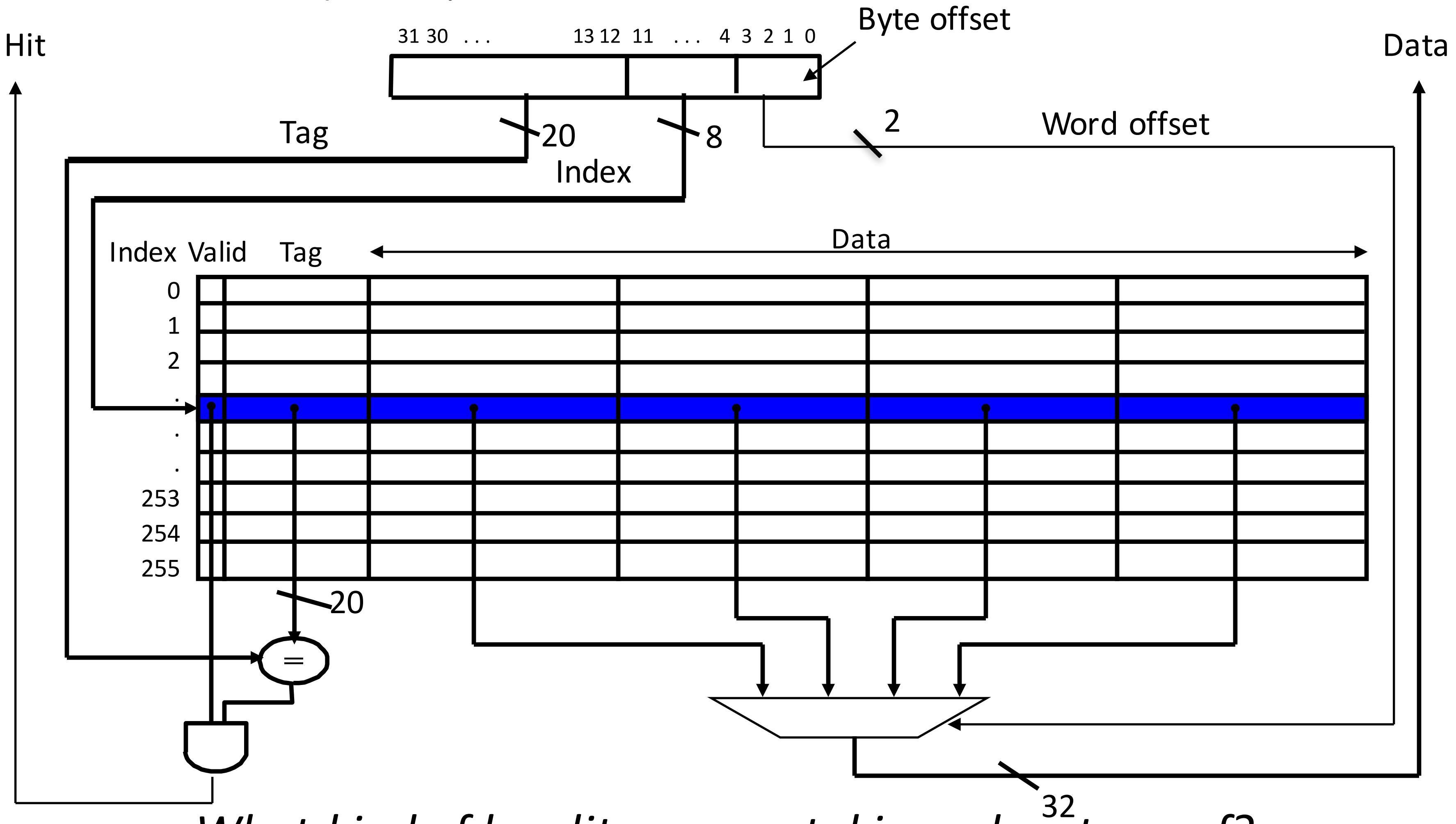
Compare Tag with upper part of Address to see if a Hit

Read data from cache instead of memory if a Hit

What kind of locality are we taking advantage of?

Multiword-Block Direct-Mapped Cache

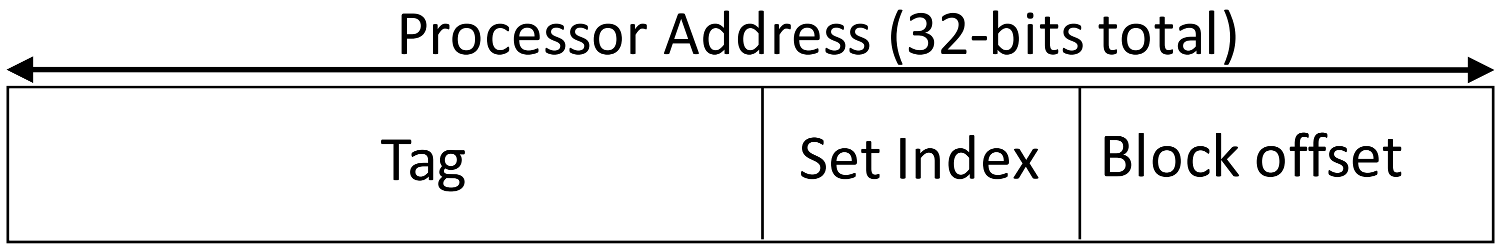
- Four words/block, cache size = 1K words



What kind of locality are we taking advantage of?

Processor Address Fields used by Cache Controller

- Block Offset: Byte address within block
- Set Index: Selects which set
- Tag: Remaining portion of processor address



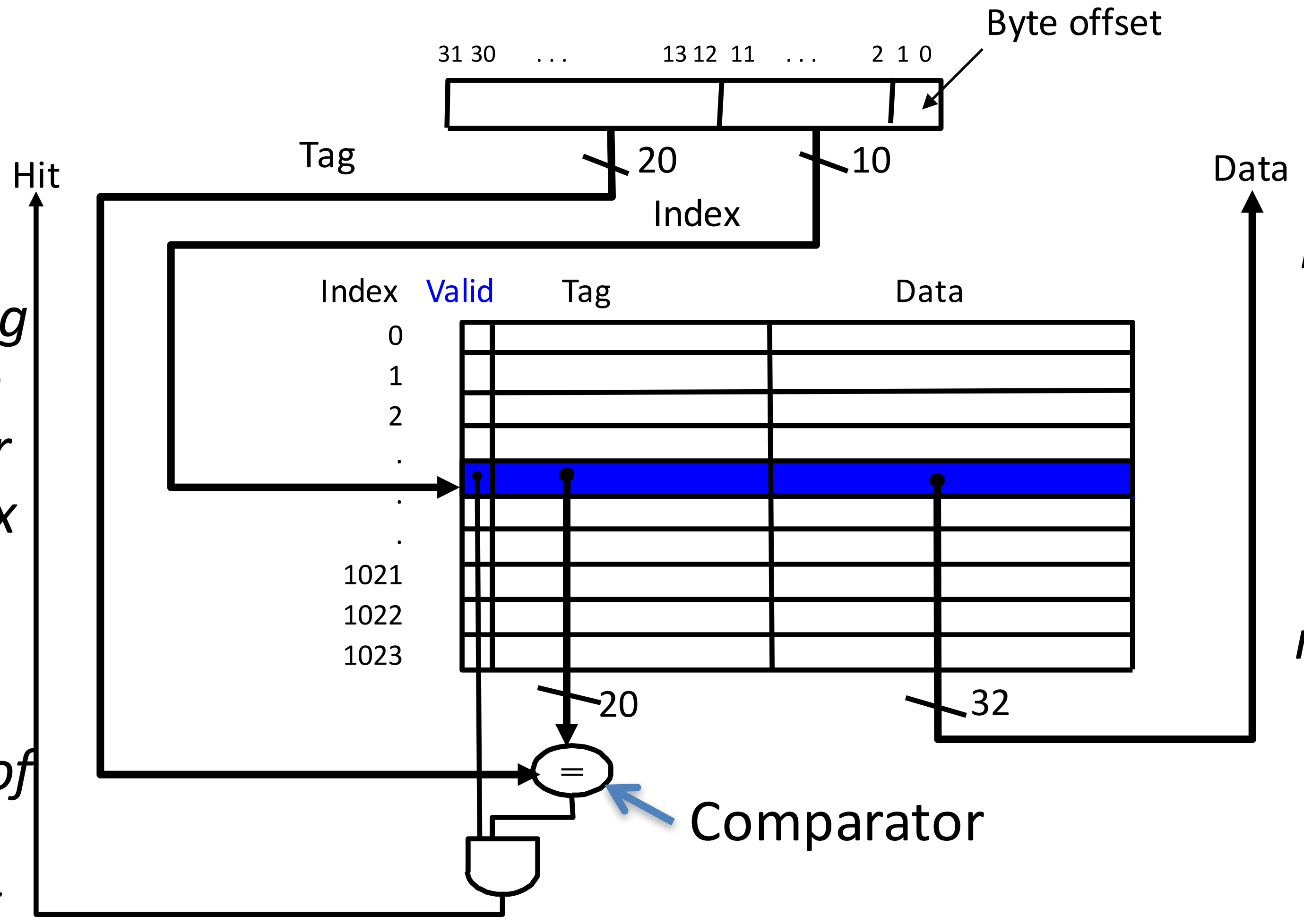
- Size of Index = log2 (number of sets)
- Size of Tag = Address size – Size of Index – log2 (number of bytes/block)

Direct-Mapped Cache Review

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

Valid bit ensures something useful in cache for this index

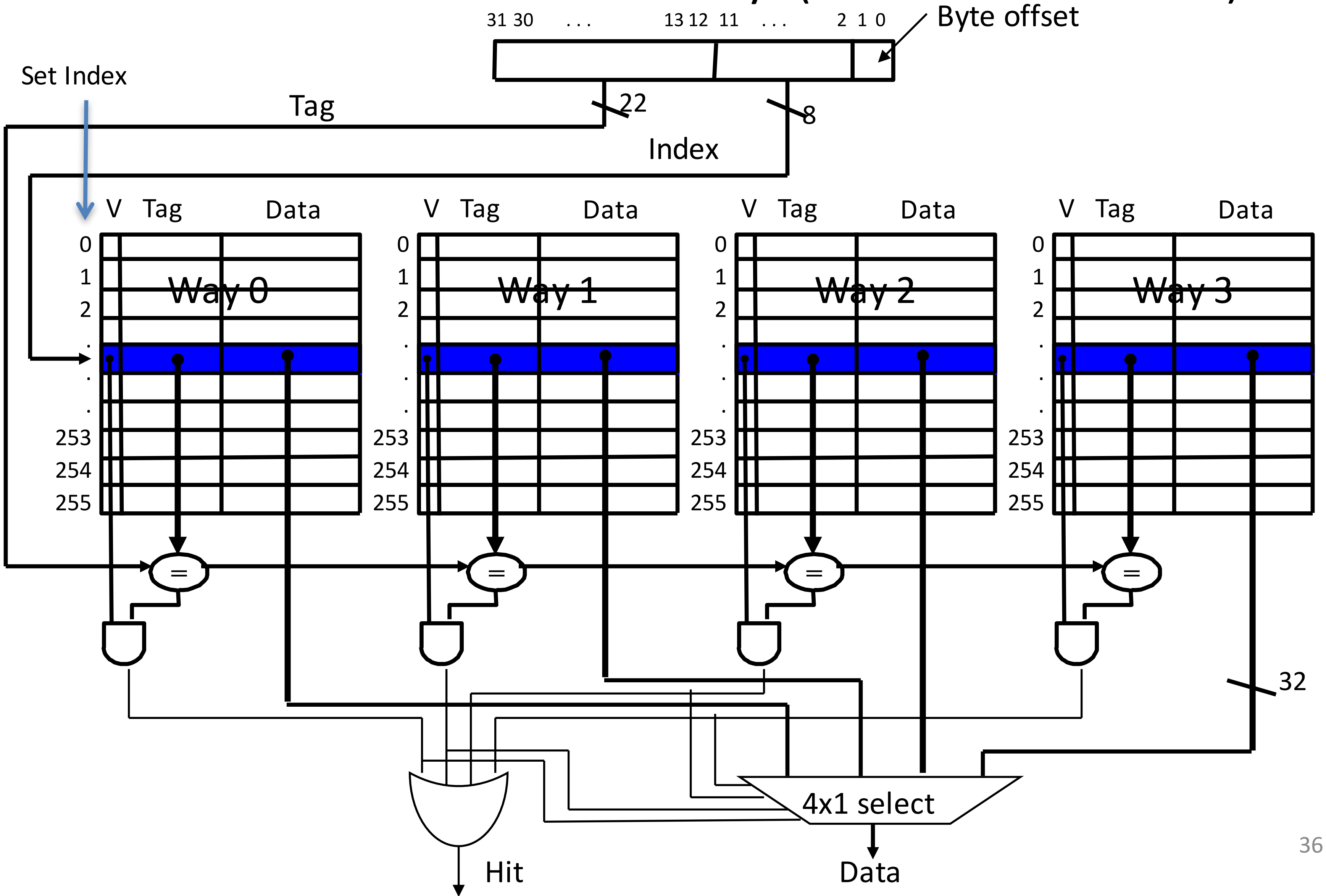
Compare Tag with upper part of Address to see if a Hit



Read data from cache instead of memory if a Hit

Four-Way Set-Associative Cache

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



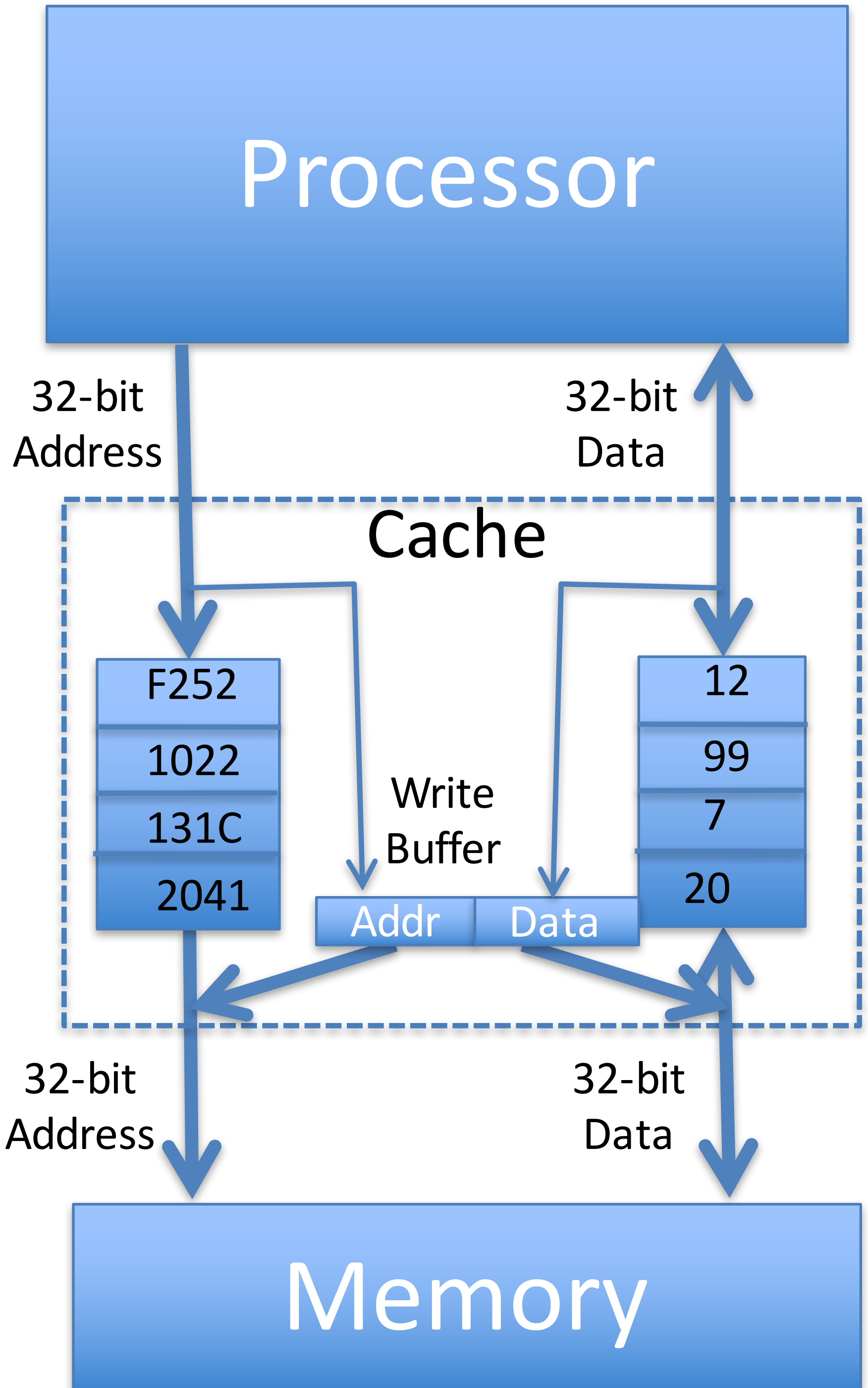
Handling Stores with Write-Through

- 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?

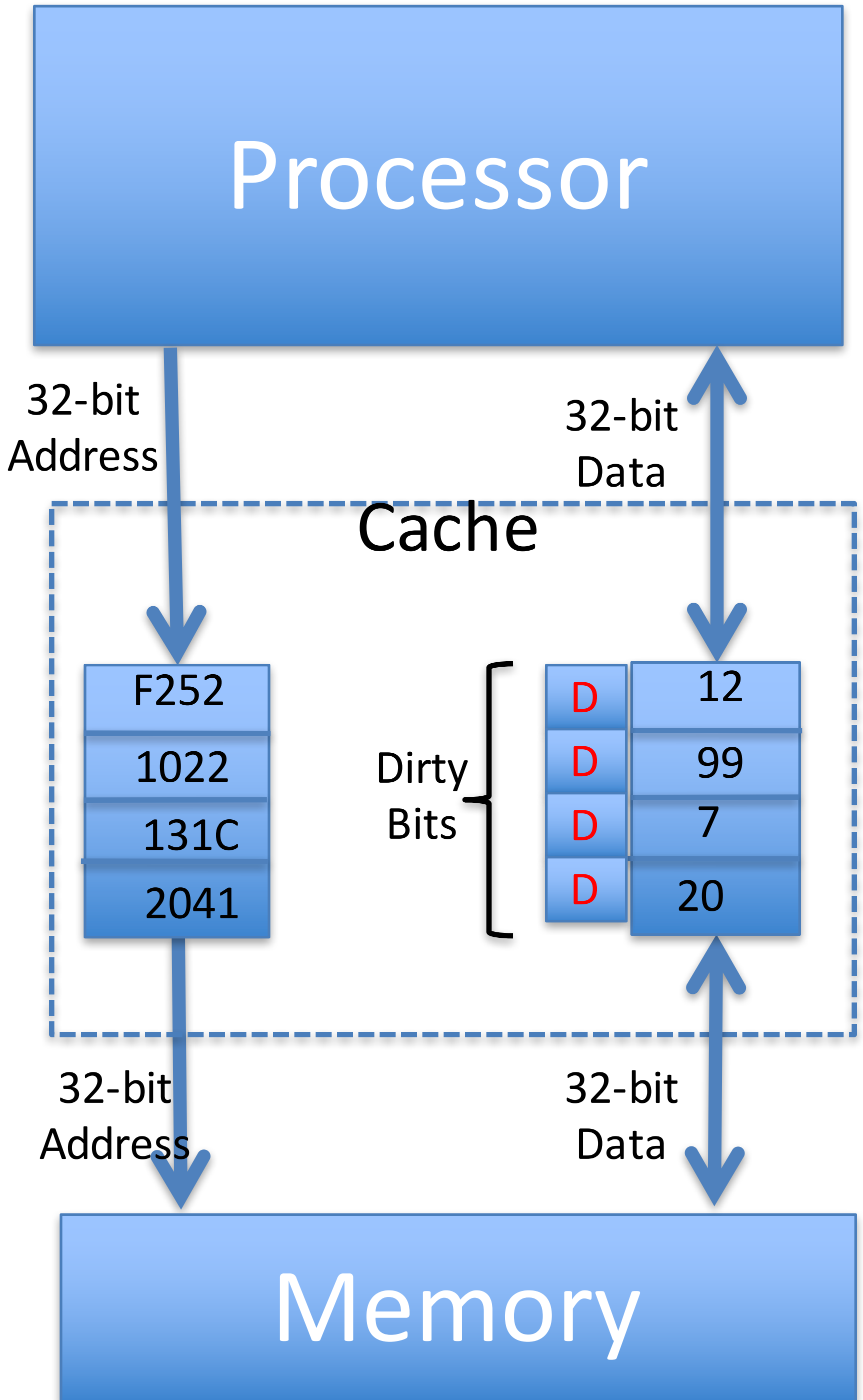


Handling Stores with Write-Back

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
- 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

Write Policy Choices

- Cache hit:
  - **write through**: writes both cache & memory on every access
    - Generally higher memory traffic but simpler pipeline & cache design
  - **write back**: writes cache only, memory `written only when dirty entry evicted
    - A dirty bit per line reduces write-back traffic
    - Must handle 0, 1, or 2 accesses to memory for each load/store
- Cache miss:
  - **no write allocate**: only write to main memory
  - **write allocate** (aka fetch on write): fetch into cache
- Common combinations:
  - write through and no write allocate
  - write back with write allocate

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)

- Abbreviation: "$" = cache (A Berkeley innovation!)

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

Clickers/Peer instruction

AMAT = Time for a hit + Miss rate x Miss penalty

Given a 200 psec clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache hit time of 1 clock cycle, what is AMAT?

- ☐ A: ≤200 psec
- ☐ B: 400 psec
- ☐ C: 600 psec
- ☐ D: ≥ 800 psec

Example: Direct-Mapped Cache with 4 Single-Word Blocks, Worst-Case Reference String

- Consider the main memory address reference string of word numbers: 0 4 0 4 0 4 0 4

Start with an empty cache - all blocks initially marked as not valid

0

4

0

4

0

4

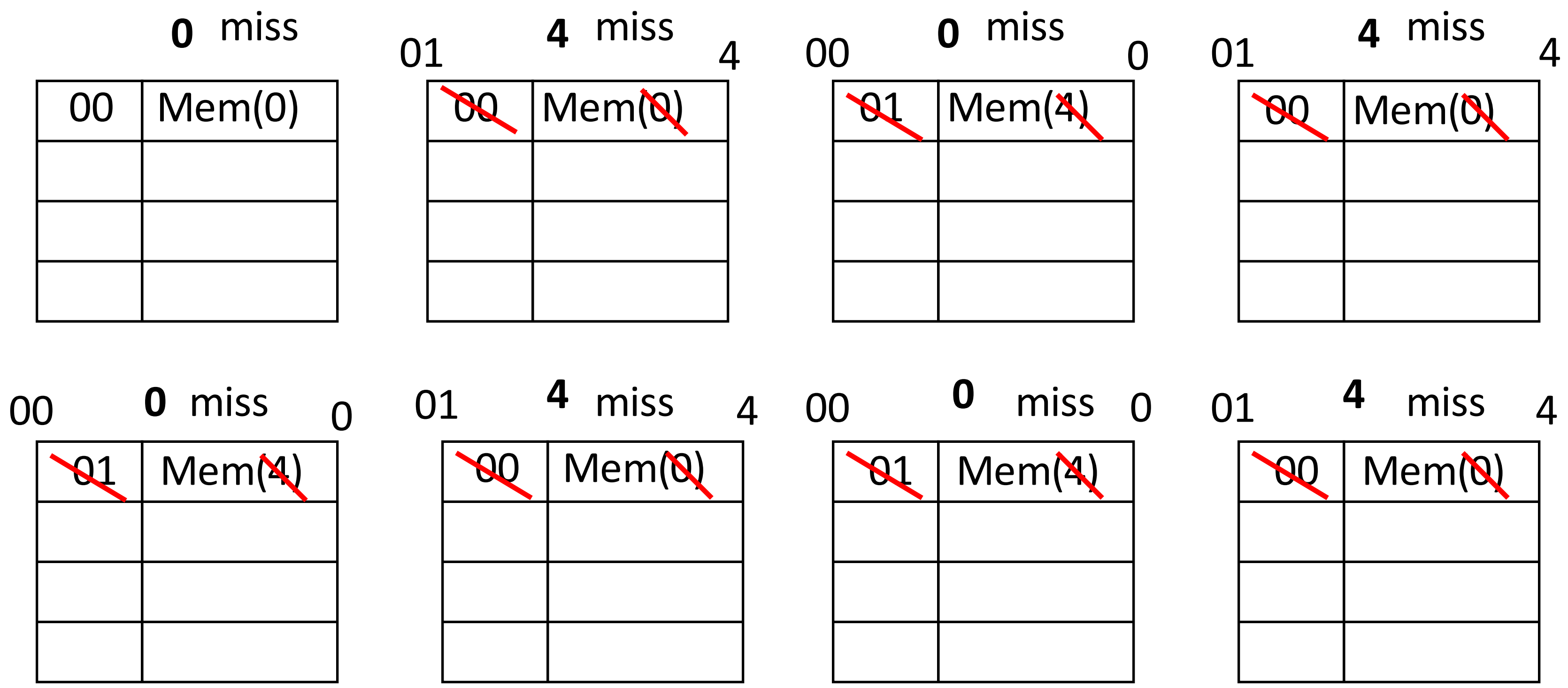
0

4

Example: Direct-Mapped Cache with 4 Single-Word Blocks, Worst-Case Reference String

- Consider the main memory address reference string of word numbers: 0 4 0 4 0 4 0 4

Start with an empty cache - all blocks initially marked as not valid



- 8 requests, 8 misses

- Ping-pong effect due to conflict misses - two memory locations that map into the same cache block