CPU Caches
Part 1

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Administrivia

- Exam tomorrow!
  - 7-9 pm
  - Email on room assignments (for in person) will be sent out today
  - If you are feeling sick, fill out the exam accommodation form to request an on-line exam
- 10, two sided, hand written cheat sheets allowed
- We will provide the "Green sheet" and an ASCII table
  - You can print out in advance or view online for on-line exams

Reminder: Pipelining Hazards

- A hazard is a situation that prevents starting the next instruction in the next clock cycle
- ***Structural*** hazard
  - A required resource is busy (e.g. needed in multiple stages)
- ***Data*** hazard
  - Data dependency between instructions
  - Need to wait for previous instruction to complete its data read/write
- ***Control*** hazard
  - Flow of execution depends on previous instruction

How To Deal With Structural Hazards?

- When two different pipeline stages need the same resource
  - E.G. Memory
- Just Throw Money At It!
  - Duplicate resources

How To Deal With Data Hazards

- Where the subsequent instruction needs the result of an "in flight" previous instruction
  - `lw t0 0(t4)`
  - `addi t0 t0 0x42`
- Reduce the frequency of the hazards:
  - "Forwarding": Take the result from a subsequent stage rather than a register file
- Stall the subsequent instruction(s)
  - Just repeat the second (and subsequent) instructions
  - And insert a "bubble" (noop) into the pipeline
- In classic RISC-V 5-stage pipeline:
  - Only data hazard which requires a stall are when you use data from a load
  - And only a single cycle

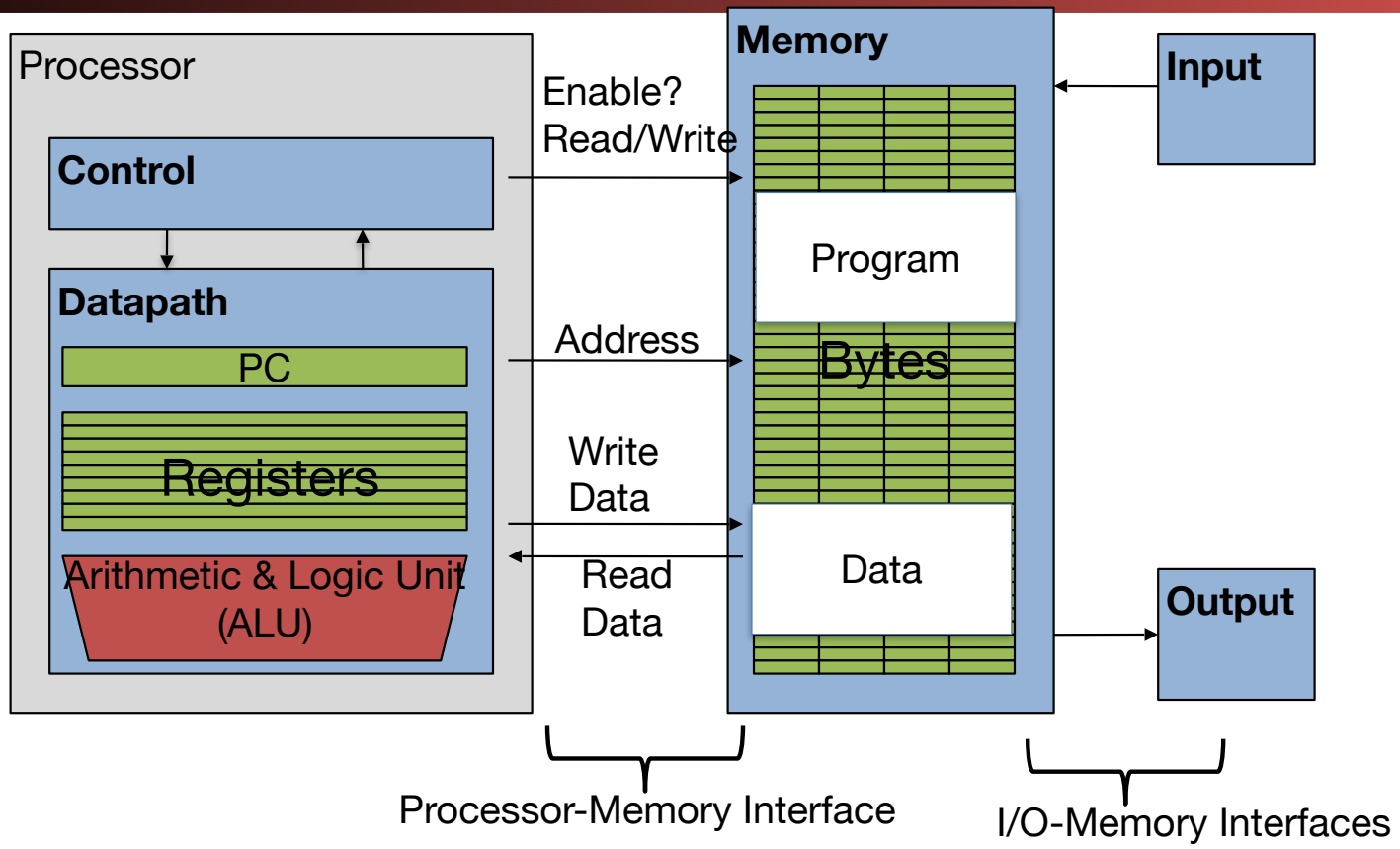
How To Deal With Control Hazards

- Control hazard: when a control flow change happens "unexpectedly"
  - The subsequent in-flight instructions ***must not*** execute:

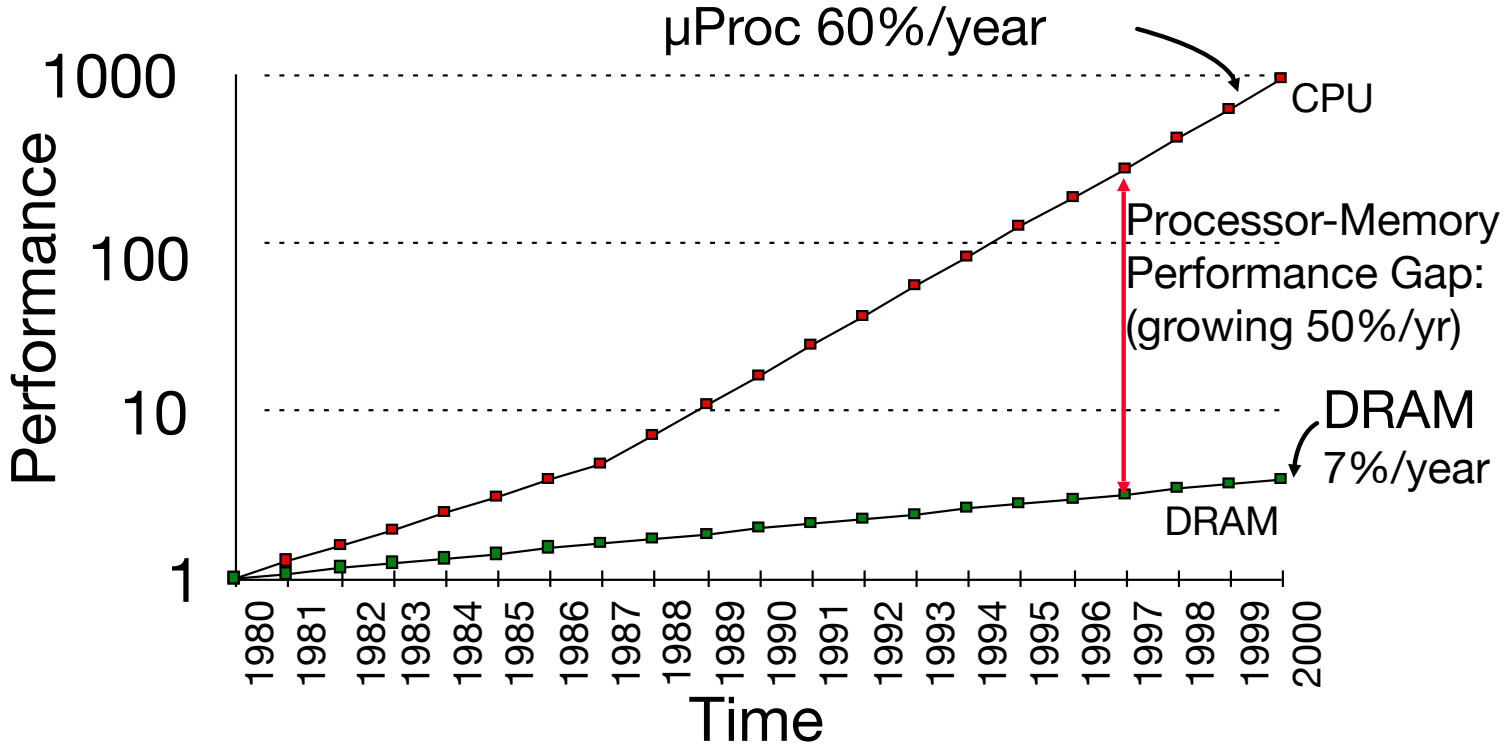
```
beq t0 t1 foo # Branch taken, but processor didn't guess this
add t0 t2 t3
xor t3 t4 t5
...
foo:
add t0 t4 t5
```

- Reduce the frequency of the hazard:
  - Better "Branch prediction": the processor guesses whether a branch or jump is likely to be taken or not
    - And, if taken, where the new location will be
- Kill the in-flight instructions:
  - Replace with bubbles/noop
- Classic RISC-V 5-stage pipeline:
  - Mispredicted branch requires killing two improperly fetched instructions

Components of a Computer



Processor-DRAM latency gap



1980 microprocessor executes ~one instruction in same time as DRAM access
2020 microprocessor (theoretically) executes ~25,000 instructions in same time as DRAM access

Berkeley EECS ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Slow DRAM access could have disastrous impact on CPU performance!

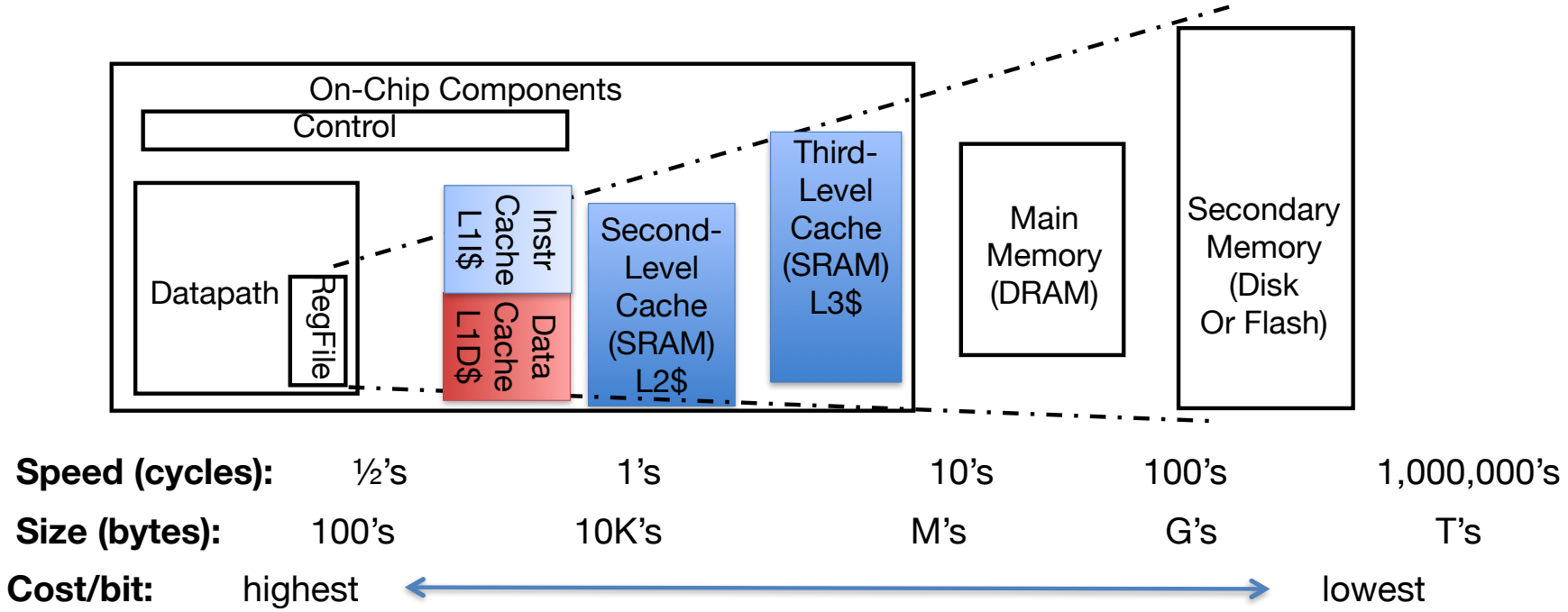
The numbers today...

- Nick's Desktop:
  - AMD Ryzen 9 3900X: 12 cores, each at 3.8 GHz
  - DDR4 at 3 GHz with 15-17-17-35 latency
- Accessing a single random memory location:
  - 70ns in real world testing
- Potential compute:
  - 12 (cores) * 8 (uOPs issuable/cycle) * 70 (ns) * 3.8 (clocks/ns) ~= 25,000 possible instructions!!!!
  - Even if just 1 core and a CPI of 1 (so RISC-V 5 stage pipeline)... That is ***still*** >250 instructions worth of work!

What to do: Library Analogy

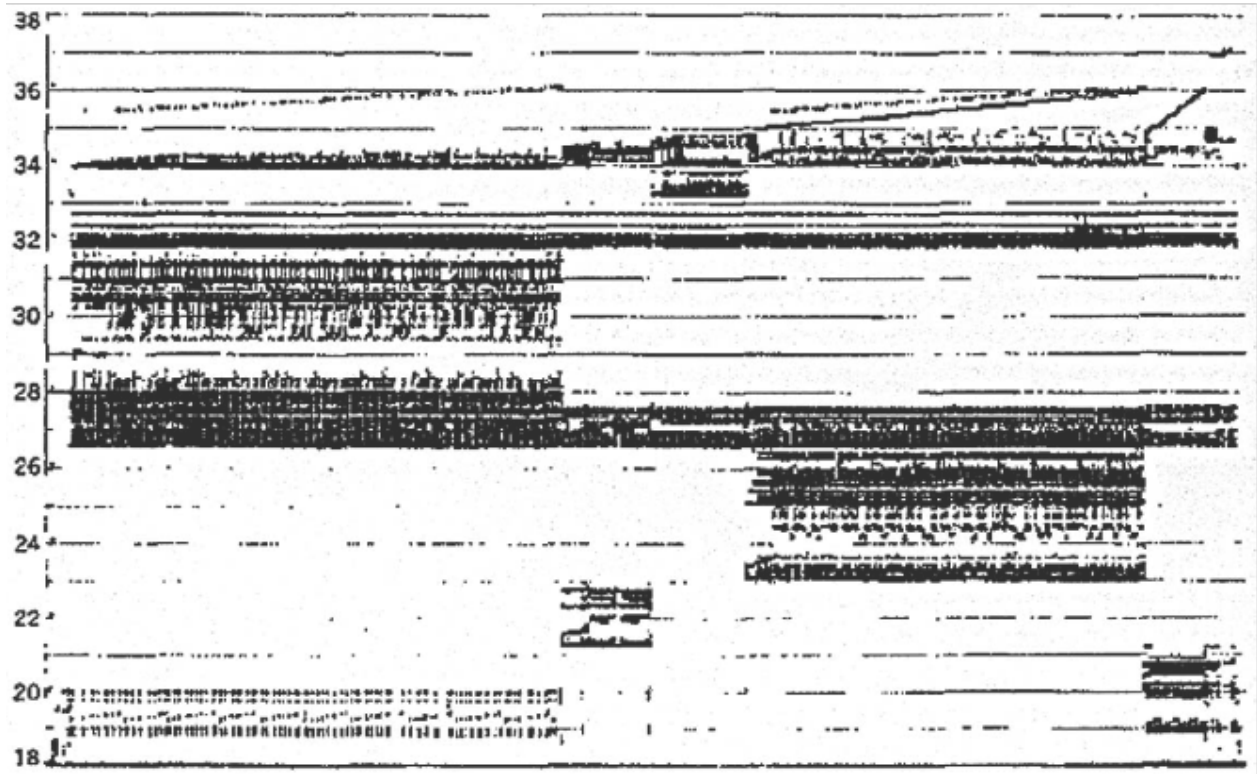
- Want to write a report using library books
  - E.g., writing about the works of John Steinbeck:
    Want to understand why on cheating cases, Nick says "A man ought to shoot his own dog"
- Go to Doe library, look up relevant books, fetch from stacks, and place on desk in library
  - Hint, start with "Of Mice and Men"
- 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



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

Real Memory Reference Patterns

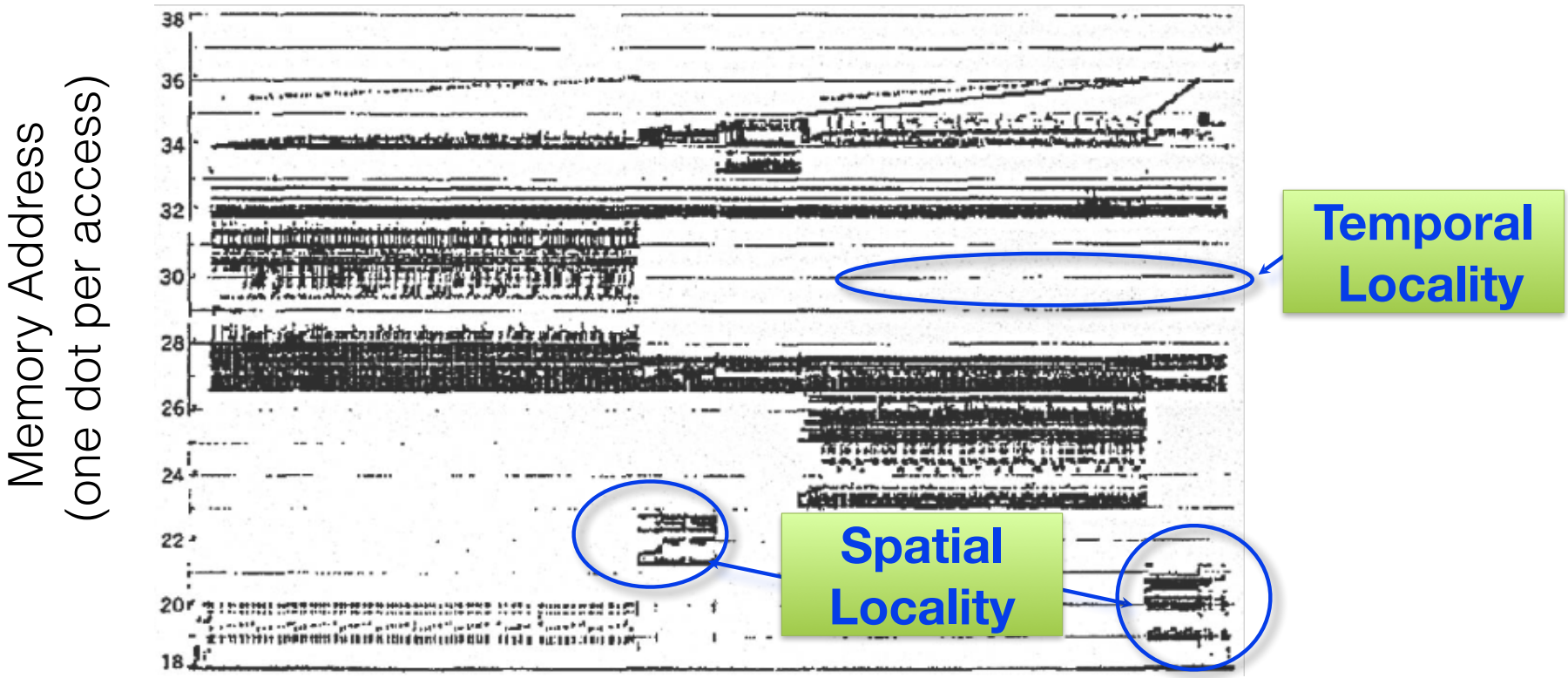


Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

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 Steinbeck 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

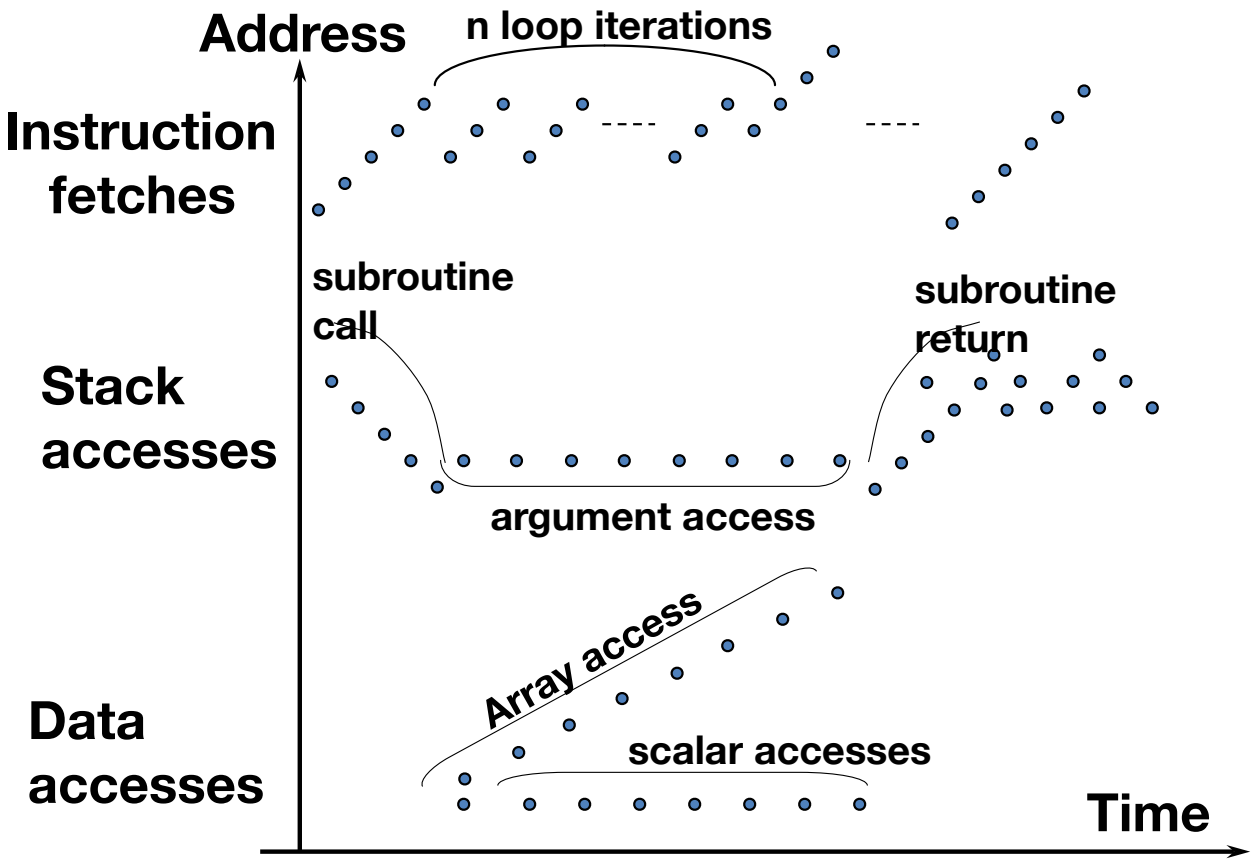


Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

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?

Good Memory Reference Patterns



And the Bane of Locality: Pointer Chasing...

- We all love linked lists, trees, objects, etc...
  - Easy to append onto and manipulate...
- But they exhibit **poor** locality
  - Every time you follow a pointer it is to an unrelated location:
    No spacial reuse from previous pointers
  - And if you don't chase the pointers again you don't get temporal reuse either
- An O(lg(n)) search time in a balanced binary tree...
  - Does you no good if each step in the search takes 40 clock cycles!

Modern Language Alternatives: slices and maps

- Why modern languages tend to do things a bit differently. For example, ***go*** has "slices" and "maps":
- Slice, an easy to append to view into an array
  - Only copies on append when you overwhelm the size
- Map, a hash table implementation
  - But without nearly so much pointer chasing, tends to be a tuned implementation

Cache Philosophy

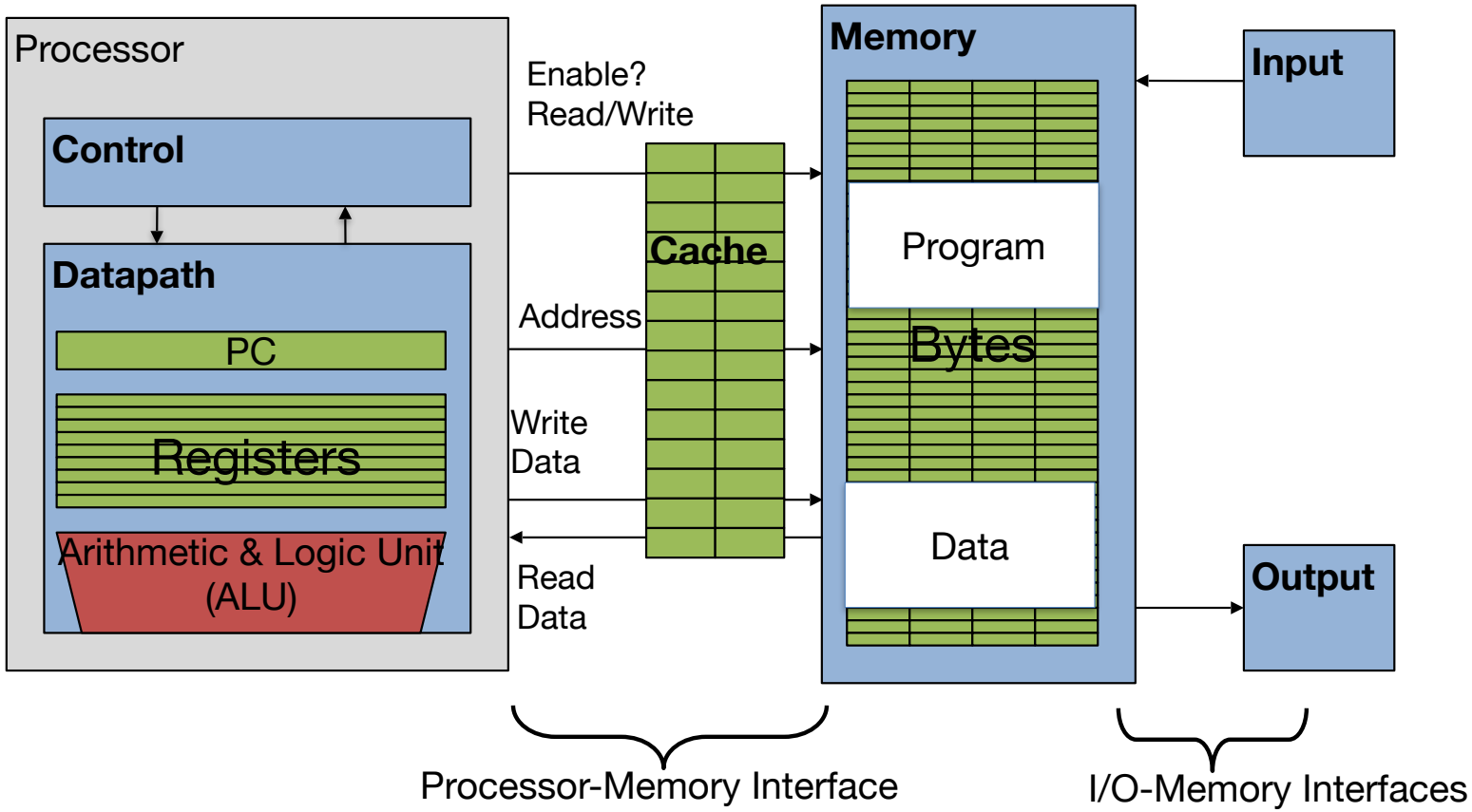
- Programmer-invisible hardware mechanism to give ***illusion*** of speed of fastest memory with size of largest memory
  - Works fine (mostly) even if programmer has no idea what a cache is
    - However, performance-oriented programmers today sometimes "reverse engineer" cache design to design data structures to match cache
    - And modern programming languages try to provide storage abstractions that provide flexibility while still caching well
- Does have limits: When you overwhelm the cache your performance may drop off a cliff...

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

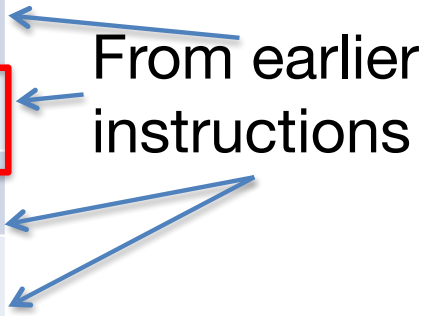
2. Processor loads 99 into register t0

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

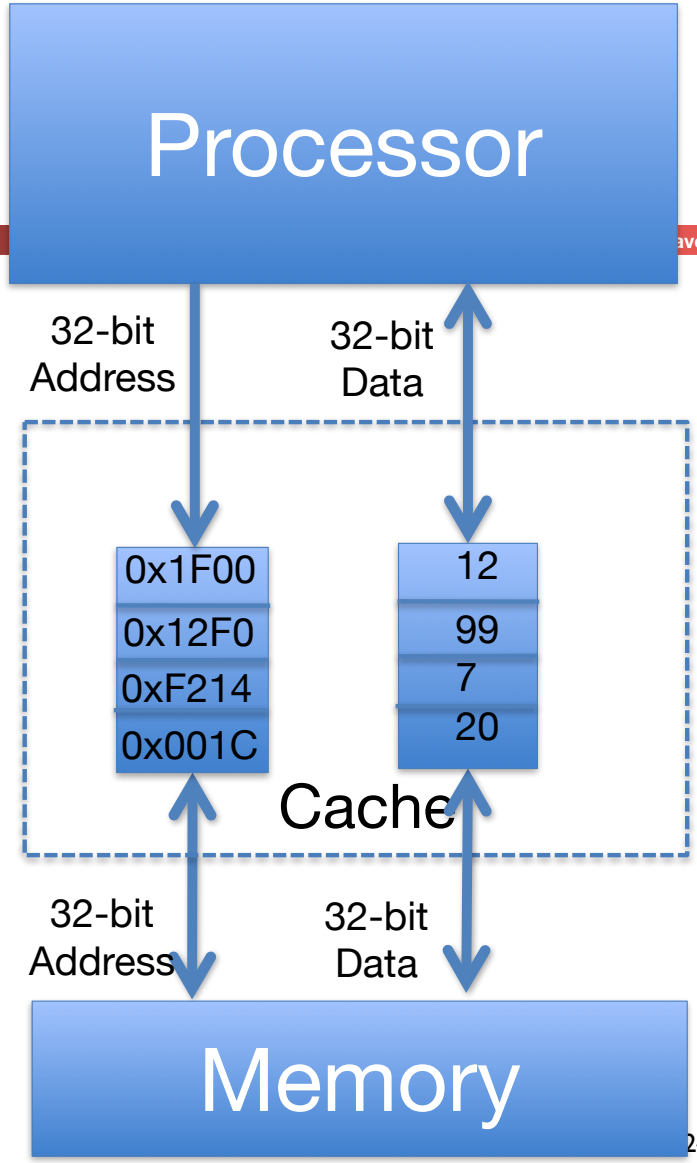
  0x12F0 placed in tag next to data from memory (99)

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



Anatomy of a 16 Byte Cache, 4 Byte Block

- Operations:
  - Cache Hit
  - Cache Miss
  - Refill cache from memory
  - Flush (empty) the cache
- 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

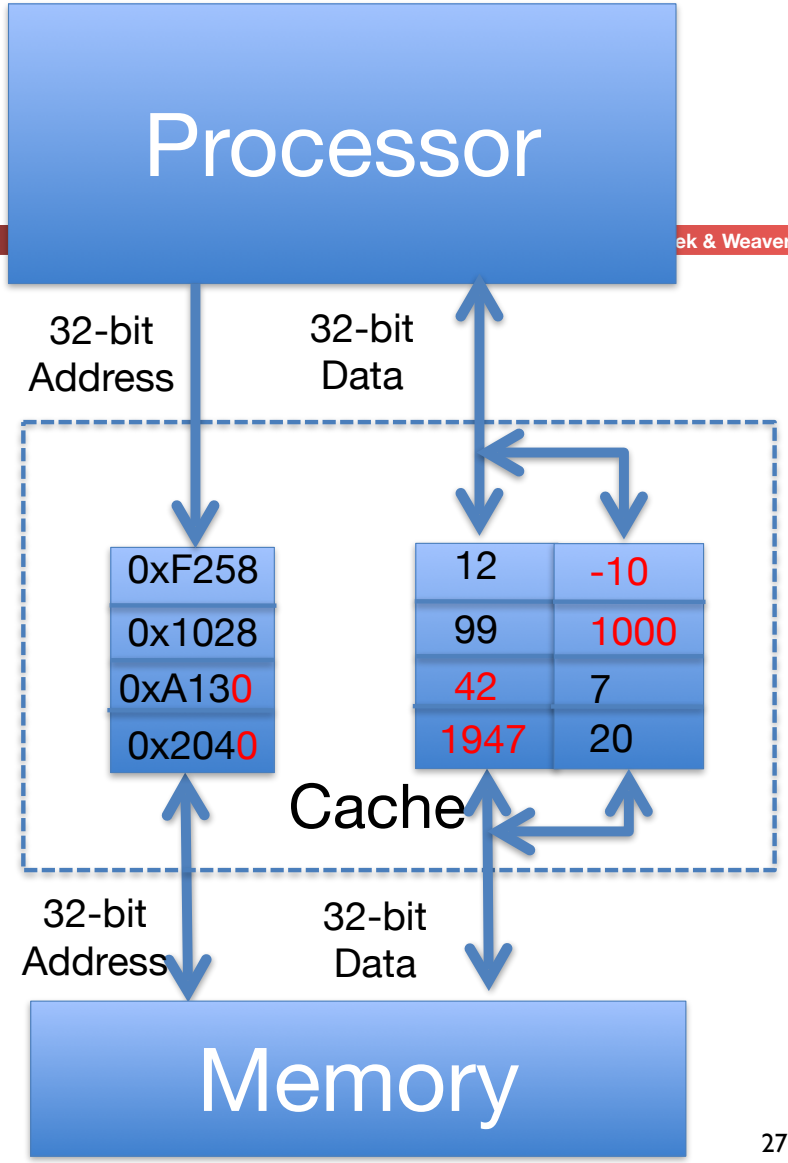
- Blocks must be aligned in pairs, otherwise could get same word twice in cache

⇒Tags only have even-numbered words

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

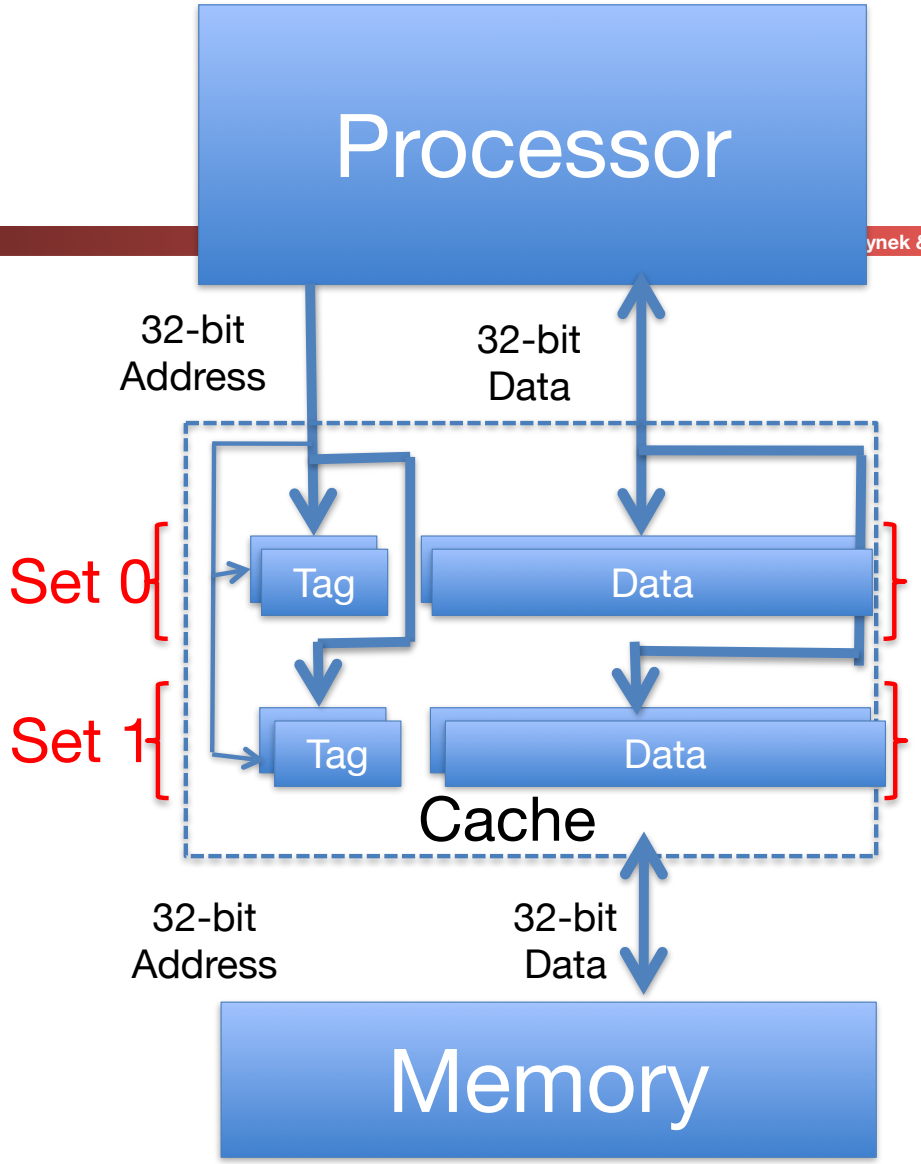
⇒Tags, comparators can be narrower

- Can get hit for either word in block



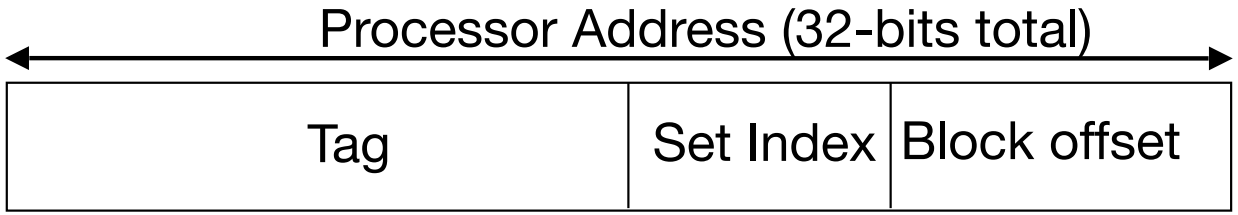
Hardware Cost of Cache

- 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 = $\log_2$ (number of sets)
- Size of Tag = Address size – Size of Index – $\log_2$ (number of bytes/block)

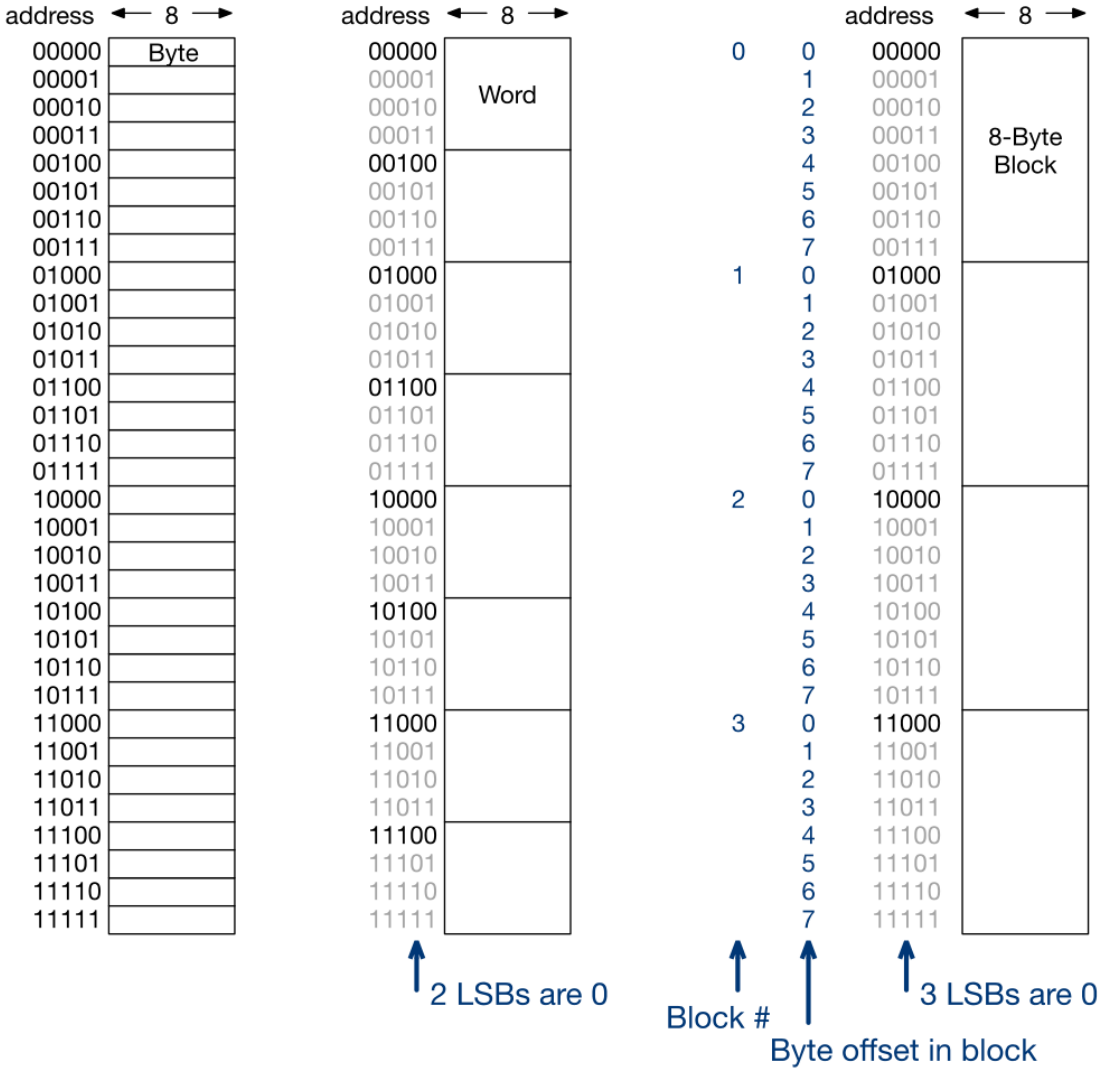
What is limit to number of sets?

- 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

Cache Names for Each Organization

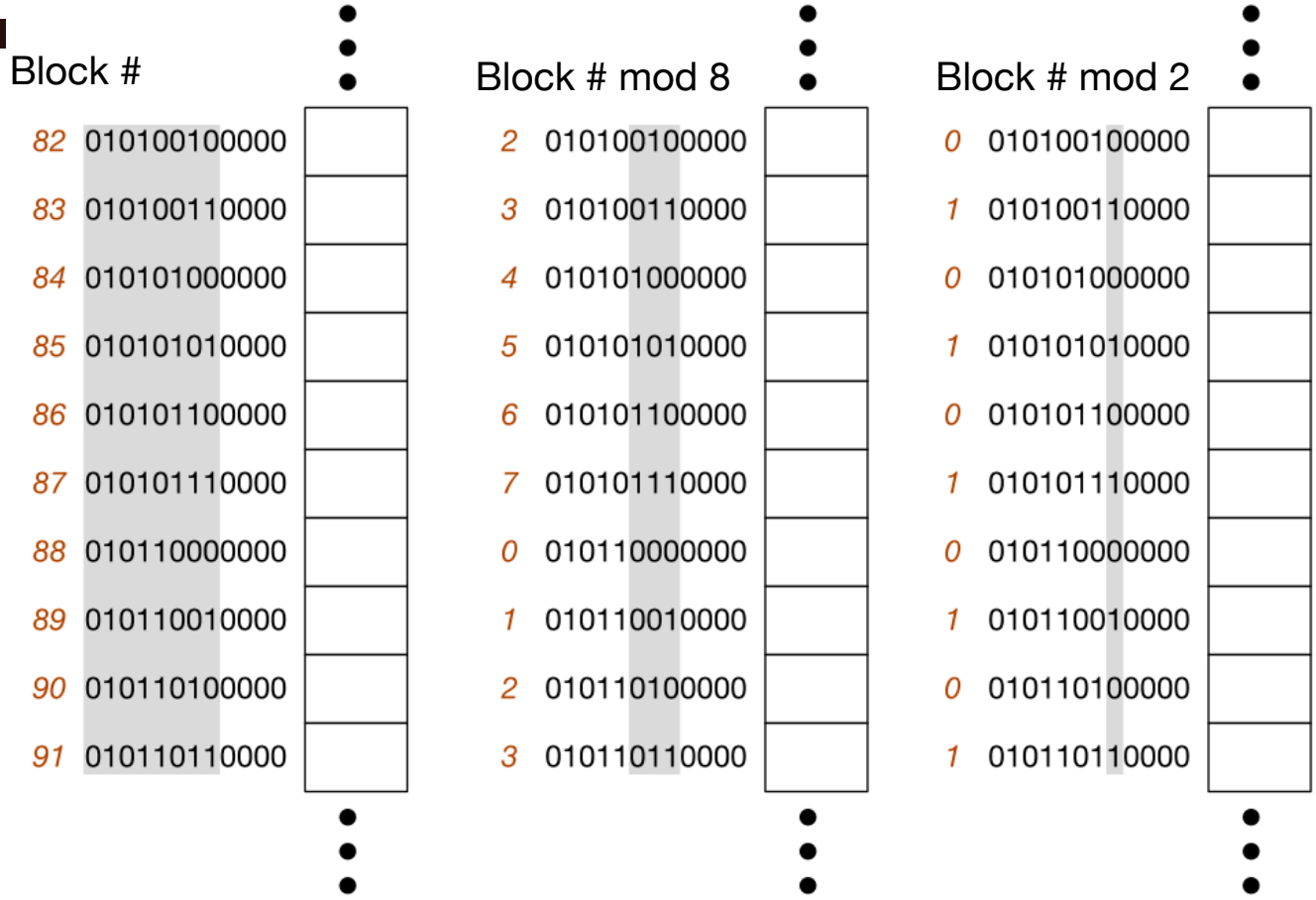
- "Fully Associative": Block can go anywhere
  - First design in lecture
  - Note: No Index field, but 1 comparator/block
- "Direct Mapped": Block goes one place
  - Note: Only 1 comparator
  - Number of sets = number blocks
- "N-way Set Associative": N places for a block
  - Number of sets = number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

Memory Block-addressing example

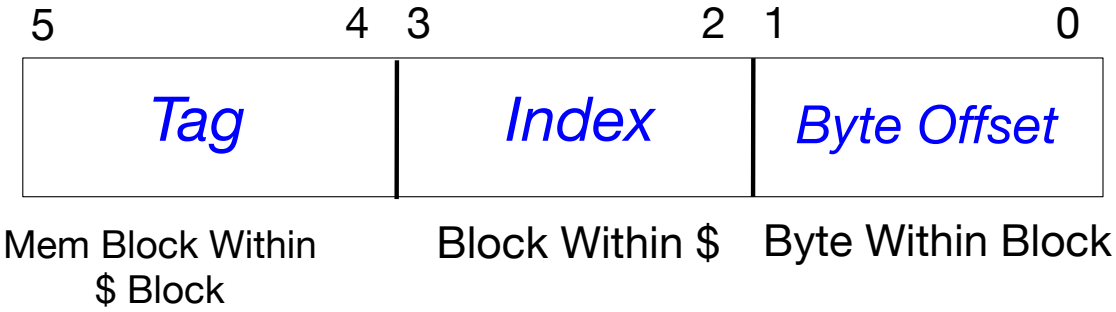


Block number aliasing example

12-bit memory addresses, 16 Byte blocks

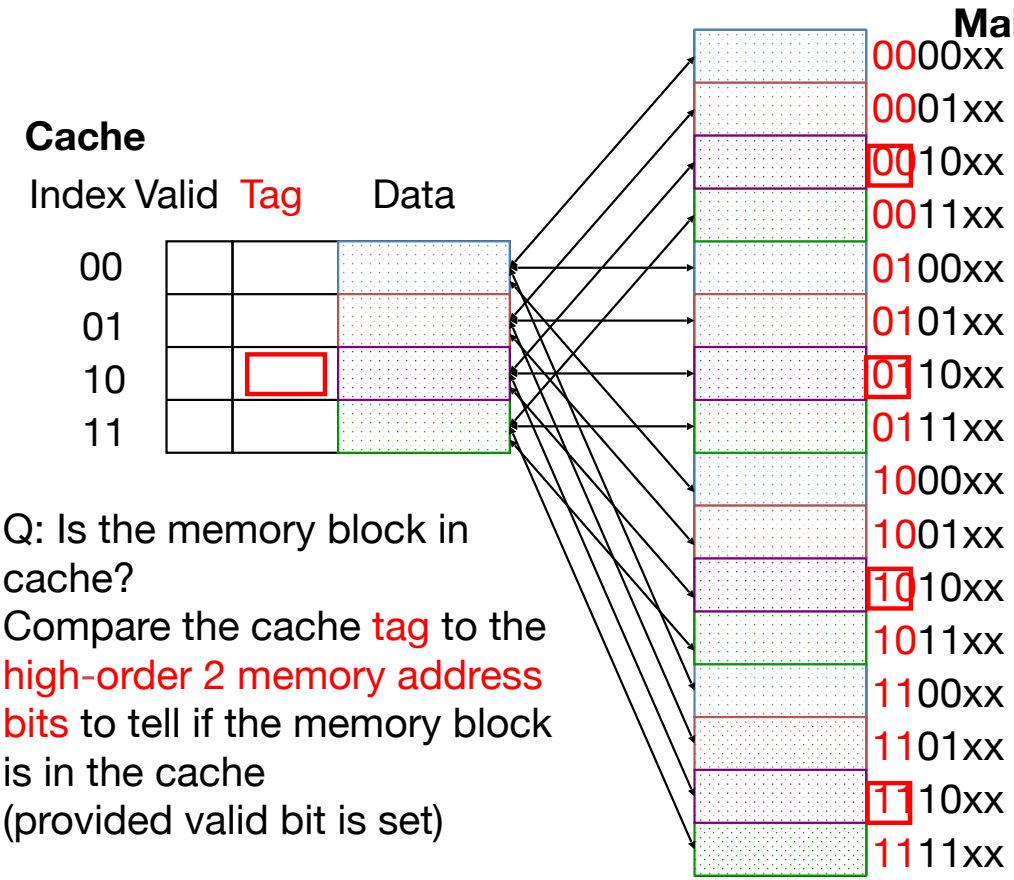


Direct Mapped Cache Ex: Mapping a 6-bit Memory Address



- 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: 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)

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)

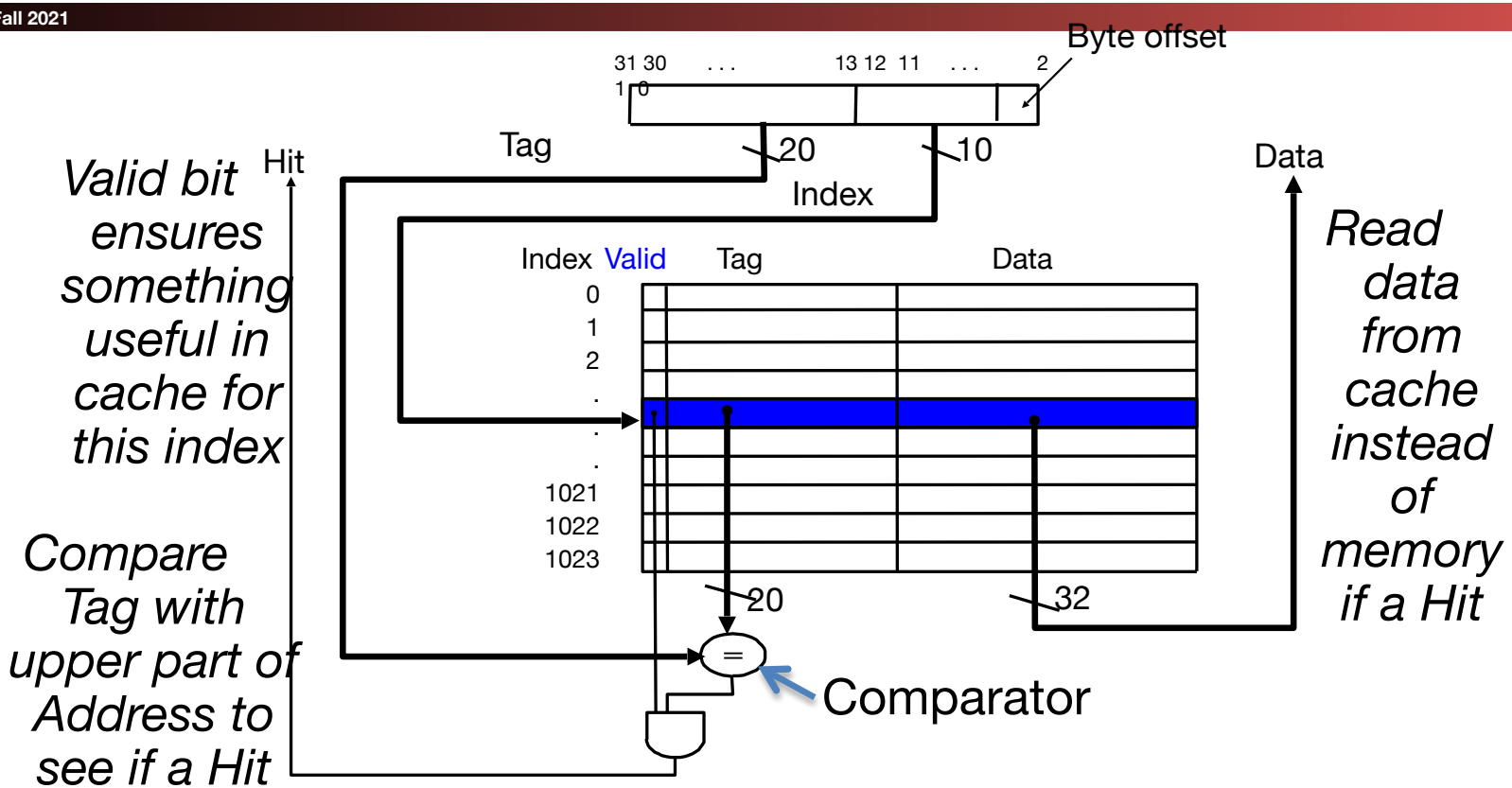
One More Detail: Valid Bit

- 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

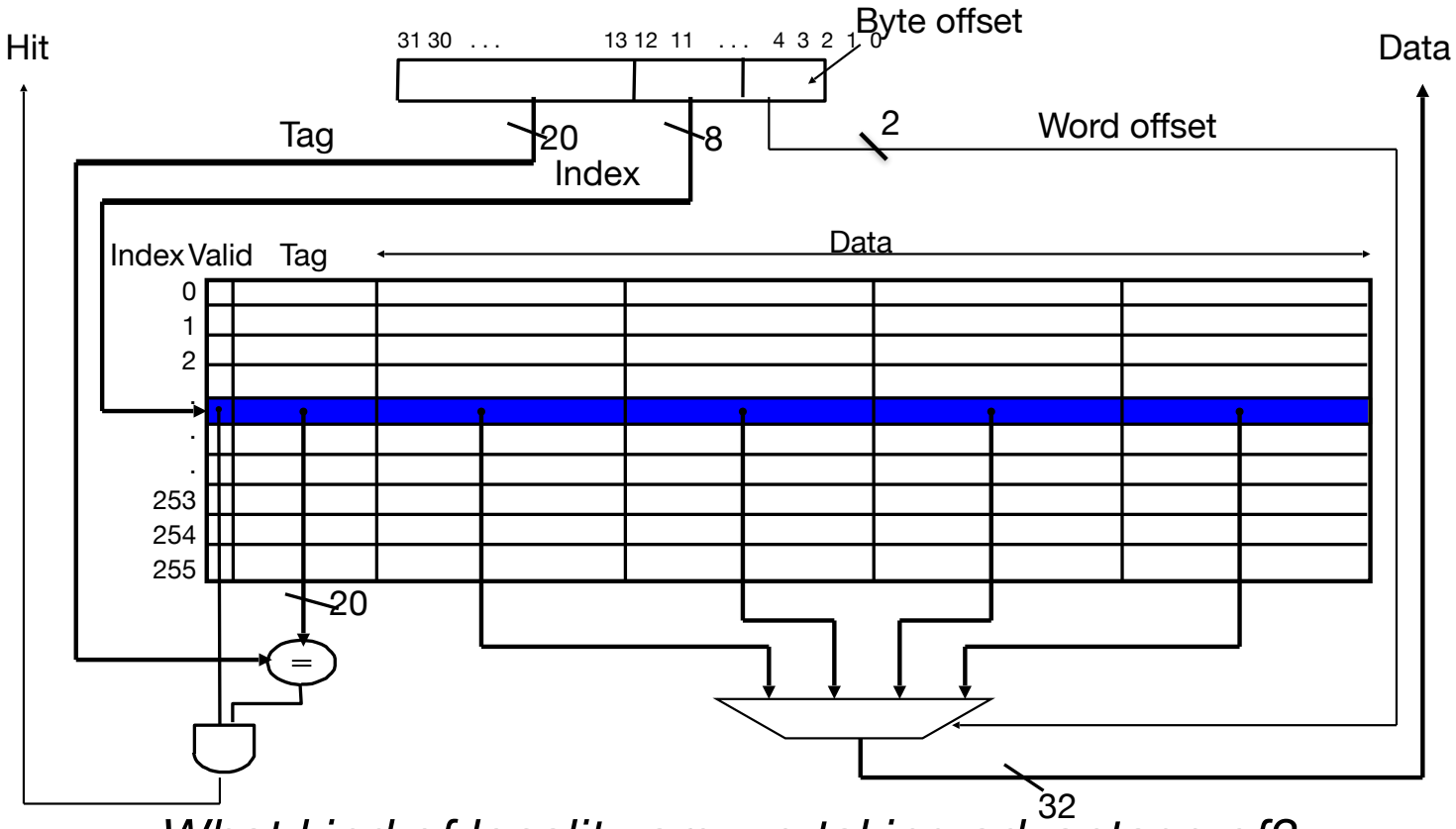
1 => cache hit, if processor address = tag

Direct-Mapped Cache Example: 1 word blocks, 4KB data



What kind of locality are we taking advantage of?

Multiword-Block Direct-Mapped Cache: 16B block size, 4 kB data



What kind of locality are we taking advantage of?

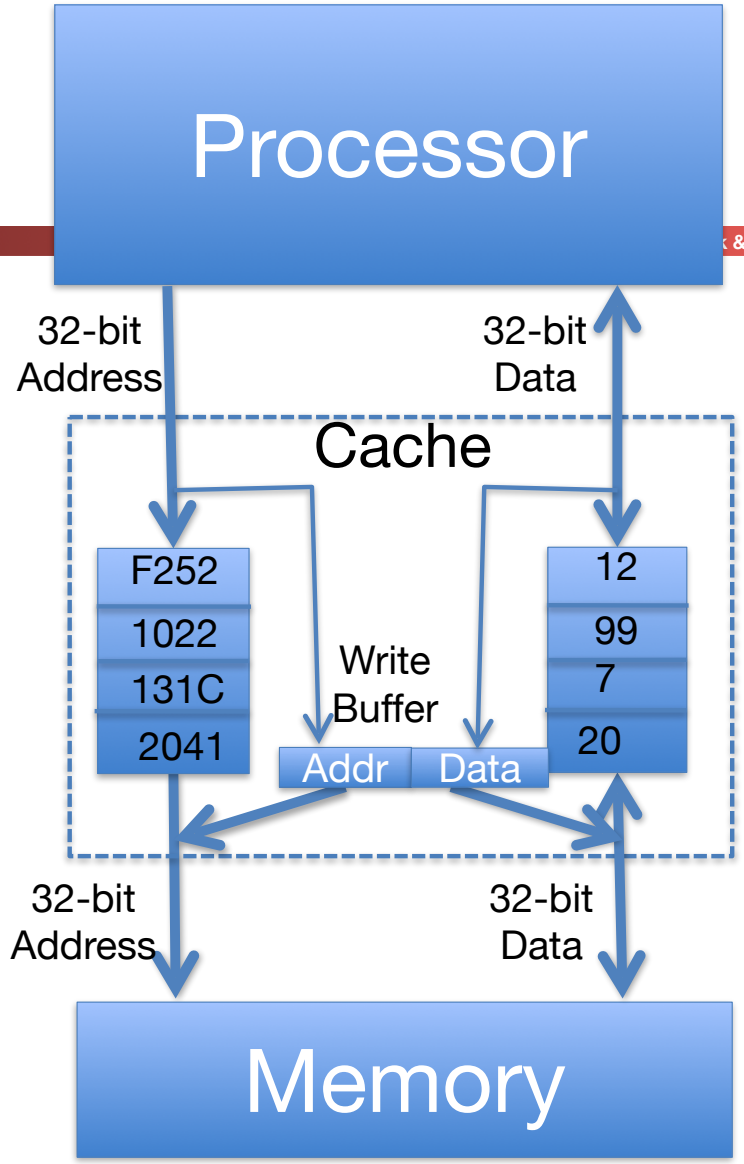
Handling Stores with Write-Through

- 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?



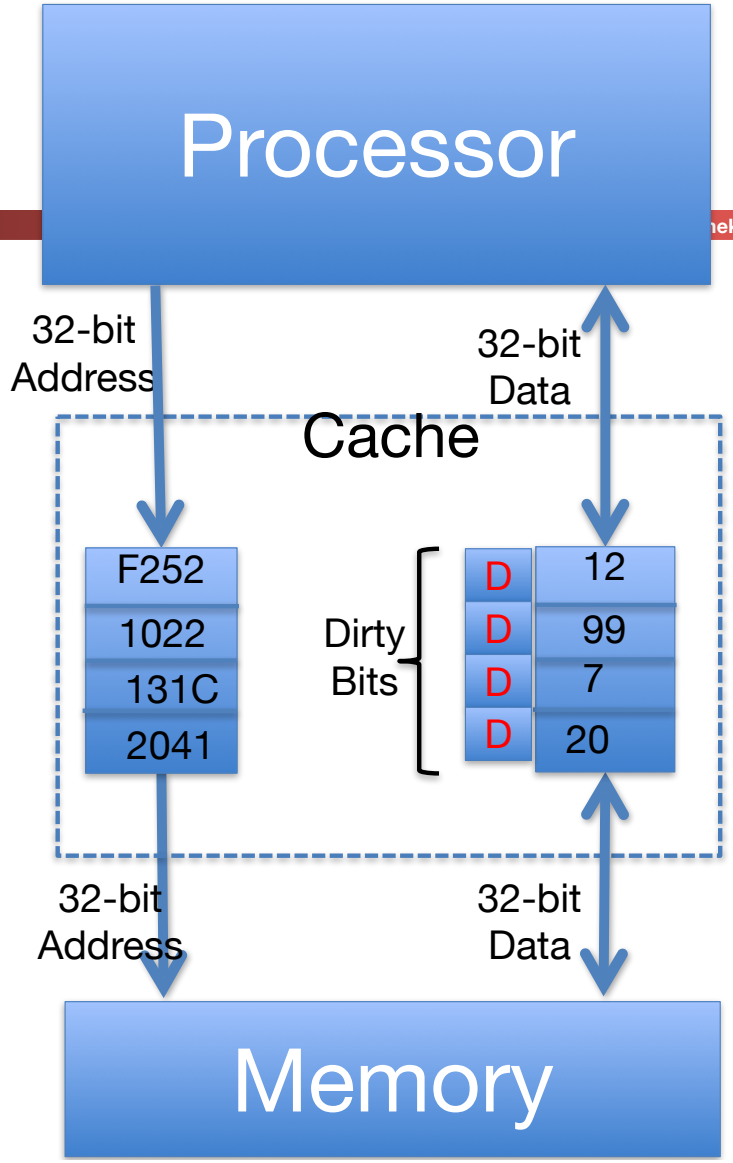
Handling Stores with Write-Back

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
    - EG, you do a lot of writes to the stack
  - 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

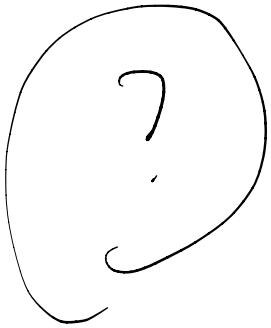
AMAT = Time for a hit
+ Miss rate × Miss penalty

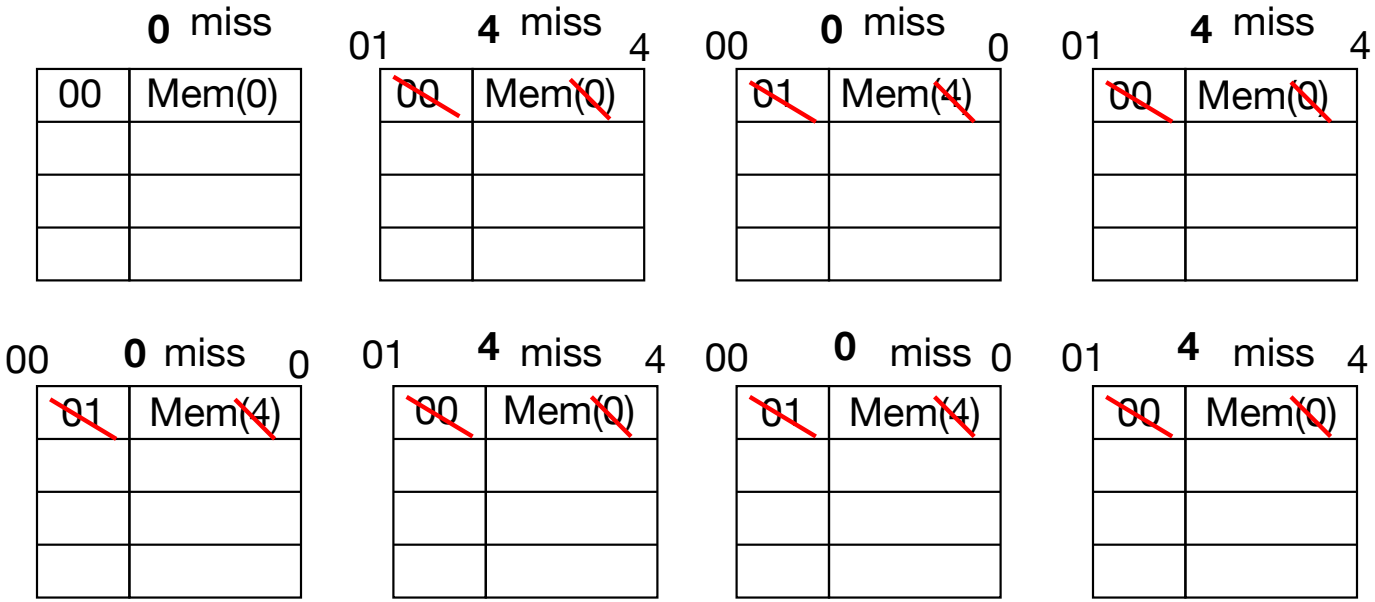
one block, one set

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

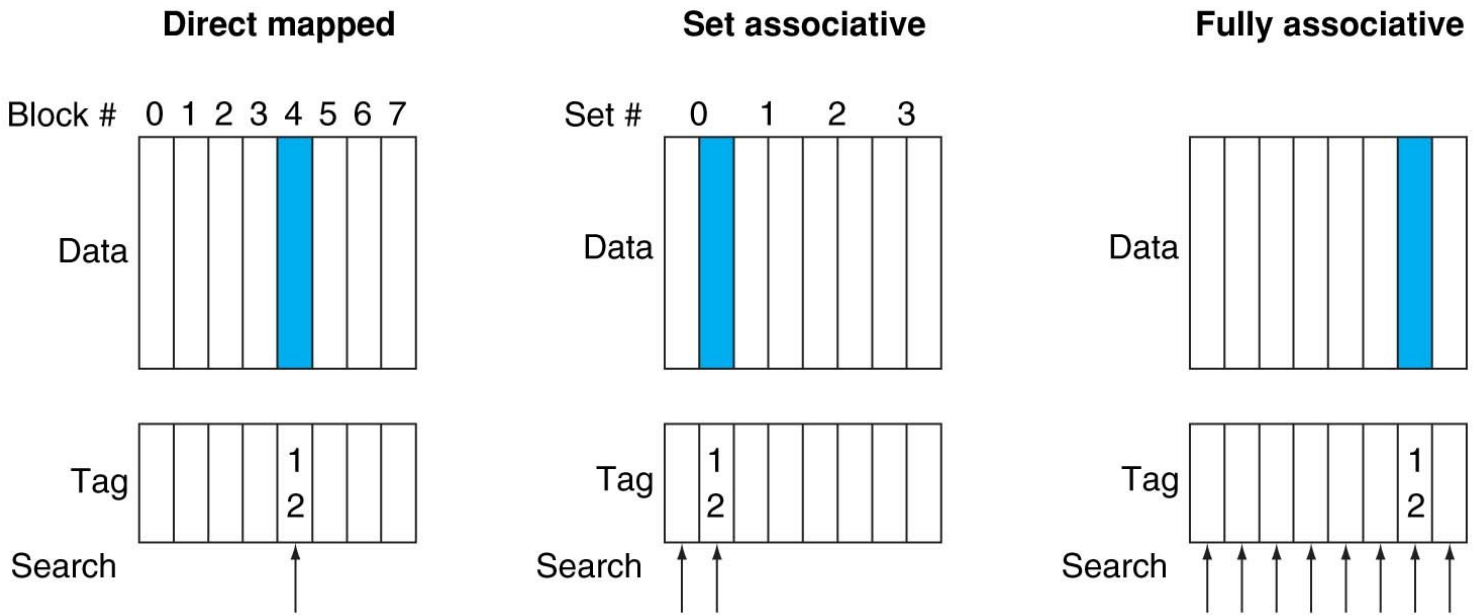




- 8 requests, 8 misses

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

Alternative Block Placement Schemes



- DM placement: mem block 12 in 8 block cache: only one cache block where mem block 12 can be found—(12 modulo 8) = 4
- SA placement: four sets x 2-ways (8 cache blocks), memory block 12 in set (12 mod 4) = 0; either element of the set
- FA placement: mem block 12 can appear in any cache blocks

Example: 4-Word 2-Way SA $ Same Reference String

- Consider the main memory address reference string

0 4 0 4 0 4 0 4

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

0 miss

000	Mem(0)

4 miss

000	Mem(0)
010	Mem(4)

0 hit

000	Mem(0)
010	Mem(4)

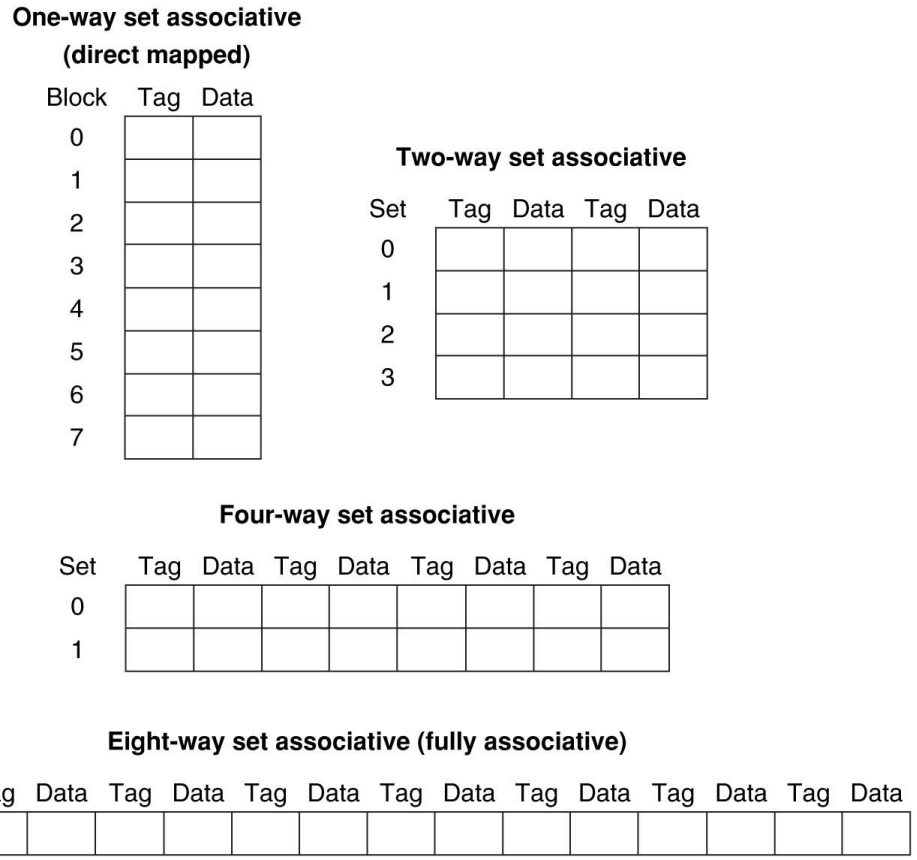
4 hit

000	Mem(0)
010	Mem(4)

  - 8 requests, 2 misses
- Solves the ping-pong effect in a direct-mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!

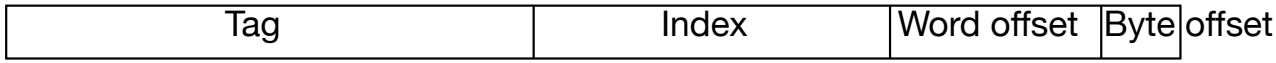
Different Organizations of an Eight-Block Cache

Total size of $ in blocks is equal to *number of sets* × *associativity*. For fixed $ size and fixed block size, increasing associativity decreases number of sets while increasing number of elements per set. With eight blocks, an 8-way set-associative $ is same as a fully associative $.



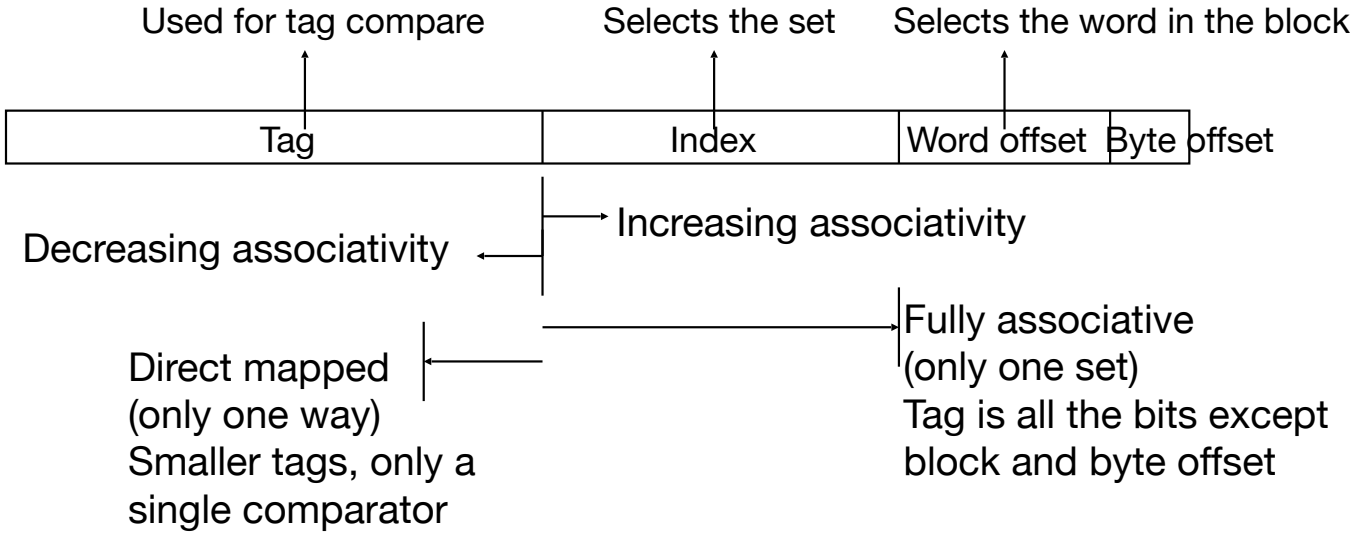
Range of Set-Associative Caches

- For a fixed-size cache and fixed block size, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit



Range of Set-Associative Caches

- For a *fixed-size* cache and fixed block size, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit



Total Cache Capacity =

of blocks in each set

Associativity × # of sets × block_size

Bytes = blocks/set × sets × Bytes/block

$C = N \times S \times B$

Tag	*Index*	*Byte Offset*

address_size = tag_size + index_size + offset_size
= tag_size + $\log_2(S)$ + $\log_2(B)$

Total Cache Capacity =

Associativity × # of sets × block_size

Bytes = blocks/set × sets × Bytes/block

$C = N \times S \times B$

address_size = tag_size + index_size + offset_size
= tag_size + $\log_2$(S) + $\log_2$(B)

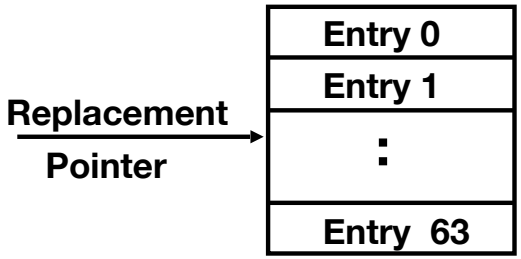
Clicker Question: C remains constant, S and/or B can change such that
C = 2N * (SB)' => (SB)' = SB/2
Tag_size = address_size – ($\log_2$(S') + $\log_2$(B')) = address_size – $\log_2$(SB)'
= address_size – $\log_2$(SB/2)
= address_size – ($\log_2$(SB) – 1)

Costs of Set-Associative Caches

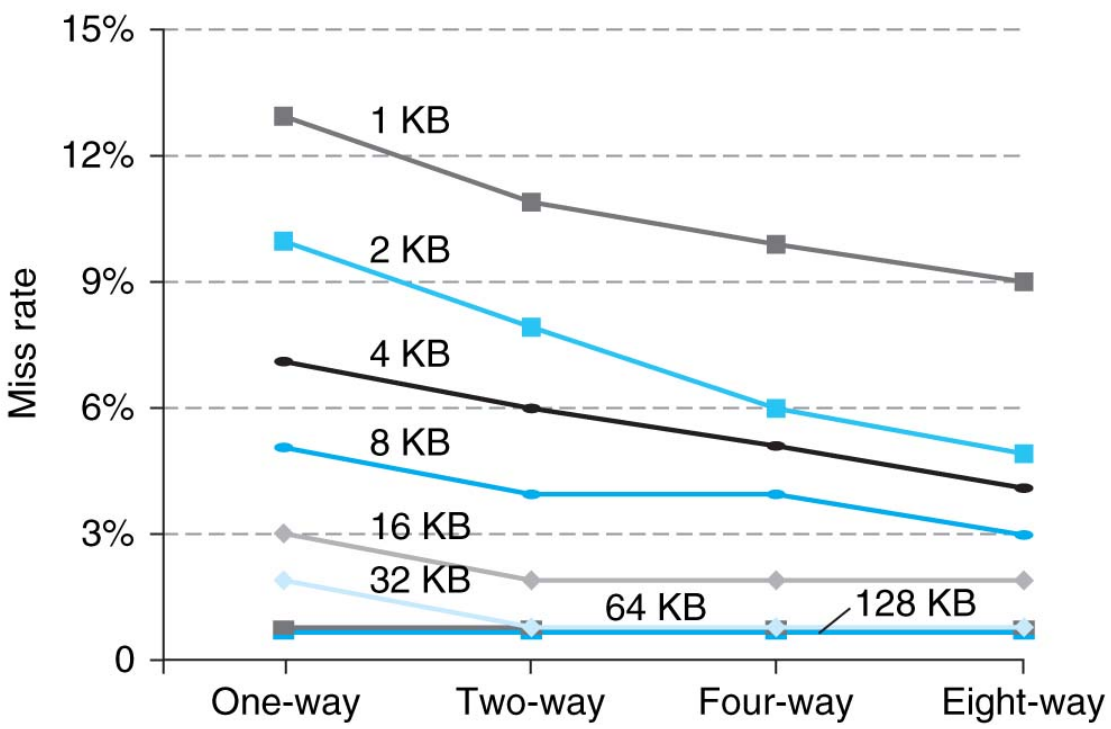
- N-way set-associative cache costs
  - N comparators (delay and area)
  - MUX delay (set selection) before data is available
  - Data available after set selection (and Hit/Miss decision). DM $: block is available before the Hit/Miss decision
    - In Set-Associative, not possible to just assume a hit and continue and recover later if it was a miss
- When miss occurs, which way's block selected for replacement?
  - ***Least Recently Used*** (LRU): one that has been unused the longest (principle of temporal locality)
    - Must track when each way's block was used relative to other blocks in the set
    - For 2-way SA $, one bit per set → set to 1 when a block is referenced; reset the other way's bit (i.e., "last used")

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)



Benefits of Set-Associative Caches



- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

Sources of Cache Misses (3 C's)

冷启动.

- *Compulsory* (cold start, first reference):
  - 1st access to a block, not a lot you can do about it.
    - If running billions of instructions, compulsory misses are insignificant
- *Capacity*:
  - Cache cannot contain all blocks accessed by the program
    - Misses that would not occur with infinite cache
- *Conflict* (collision):
  - Multiple memory locations mapped to same cache set
    - Misses that would not occur with ideal fully associative cache