# Memory Subsystem Design

or

***Nothing Beats Cold, Hard Cache***

# Finally, telling the truth about Memory

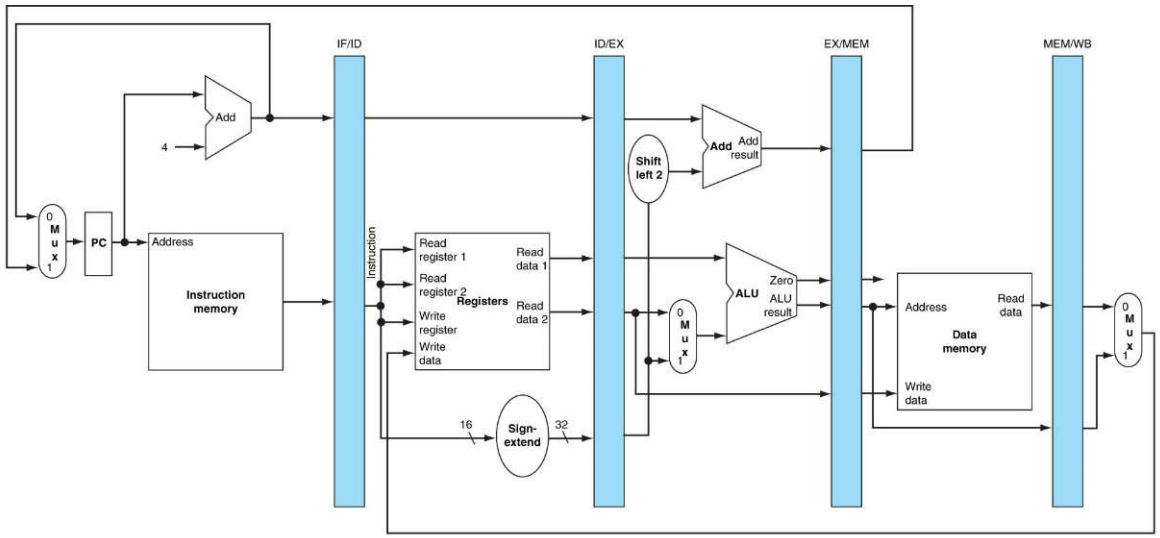
- Up to this point, we've been assuming memory can be accessed in a single cycle.
- In fact, that was true once (a long time ago…). But CPU cycle time has decreased rapidly, while memory access time has decreased very little.
- In modern computers, memory latency can be in the neighborhood of 250-500 cycles!

# The truth about memory latency

- So then what is the point of pipelining, branch prediction, etc. if memory latency is 300 cycles?
- Keep in mind, 20% of instructions are loads and stores, and we fetch (read inst memory) *every* instruction.

```
lw  R4, 1000(R2)   IF ID EX M - - - - - - - - -  …  - - - - - - - WB
lw  R8, 200(R4)       IF  ID  B - - - - - - - - -      - - - - - - - -ID EX M- - - - - - - - - …  - - - - - - - - - WB
add  R10, R8, R10        IF  B - - - - - - - - -       - - - - - - - -    ID  B - - - - - - - - -       - - - - - - - -ID EX M
```

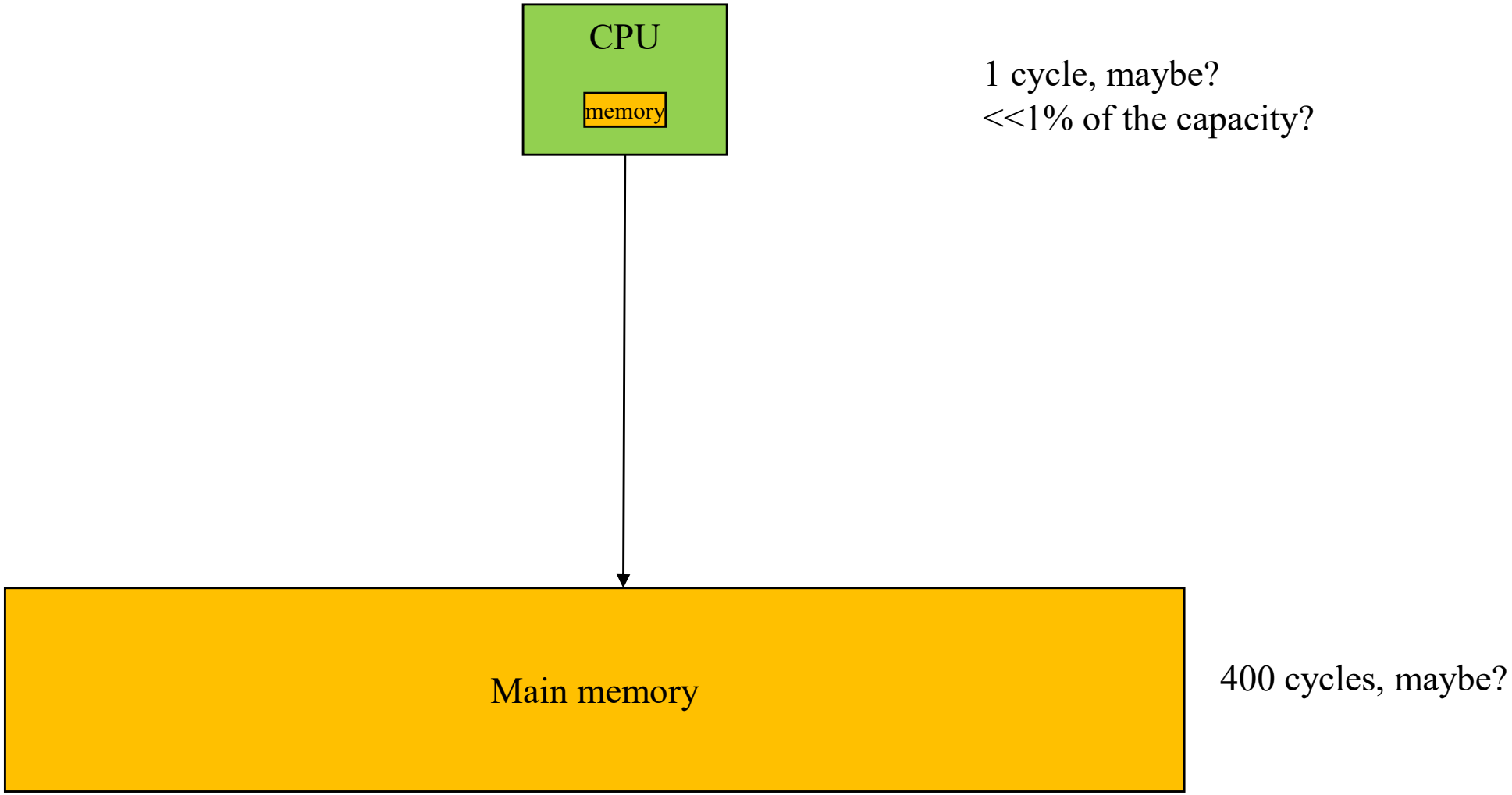
CPI = ~ ??



# But wait…

- That is assuming DRAM technology, which is necessary for large main memories (multiple gigabytes, for example)
- But we can design much smaller (capacity) memories using SRAM, even on chip.
- If we still want to access it in a cycle, it should be KB, not MB or GB.

# So what can I do with this?



# Ideas?

# Memory Locality

- Memory hierarchies take advantage of ***memory locality***.
- *Memory locality* is the principle that future memory accesses are *near* past accesses.
- Memories take advantage of two types of locality
  - -- near in time => we will often access the same data again very soon
  - -- near in space/distance => our next access is often very close to our last access (or recent accesses).

(this sequence of addresses exhibits both temporal and spatial locality)
1,2,3,1,2,3,8,8,47,9,10,8,8...

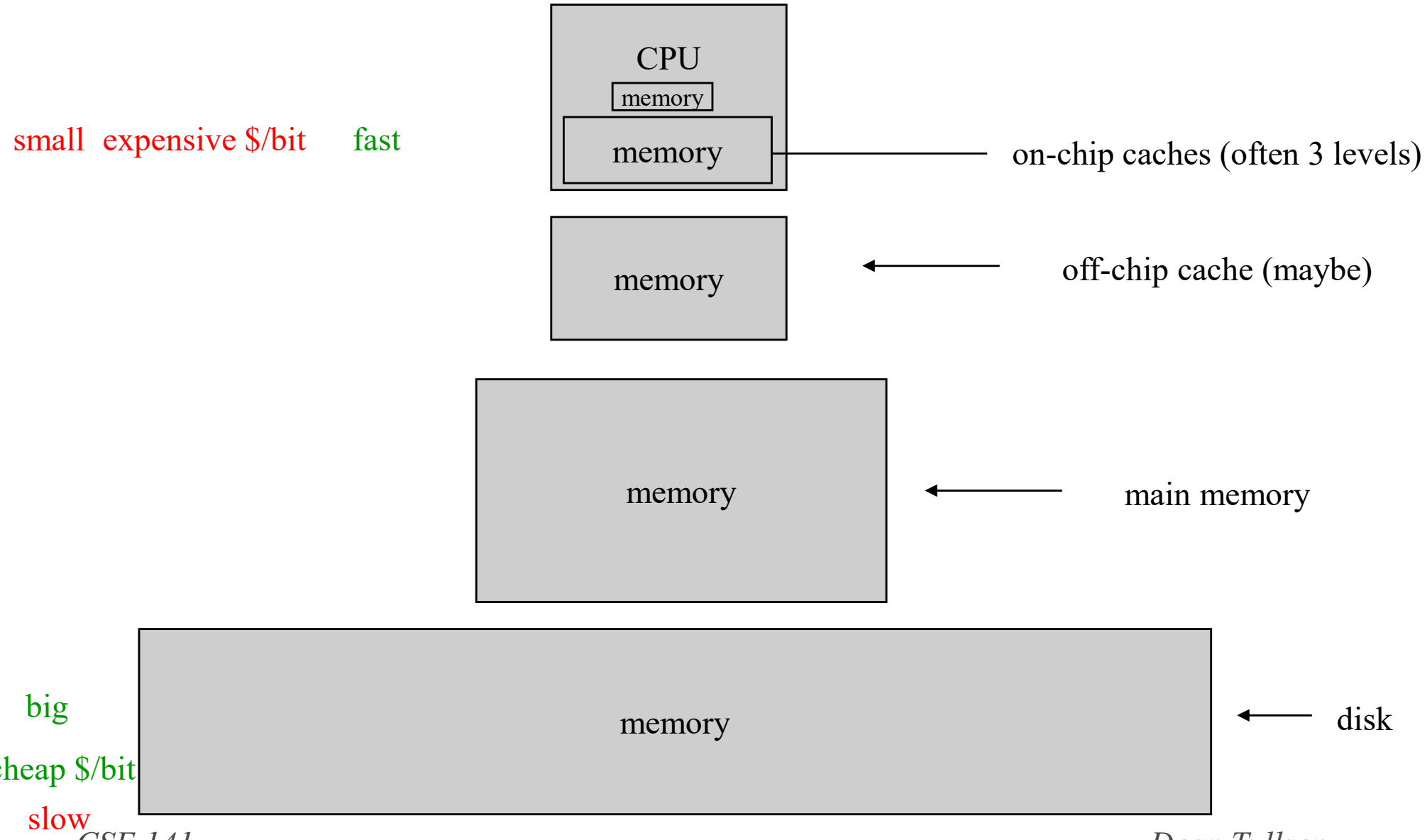
# Locality and cacheing

- Memory hierarchies exploit locality by *cacheing* (keeping close to the processor) data likely to be used again.
- This is done because we can build large, slow memories and small, fast memories, but we can't build large, fast memories.
- If it works, we get the illusion of SRAM access time with disk capacity

**SRAM access times are ~1ns at cost of ~$500 per Gbyte.**
**DRAM access times are ~60ns at cost of ~$10 per Gbyte.**

**Disk access times are 5 to 20 million ns at cost of $.20 to $2 per Gbyte.**

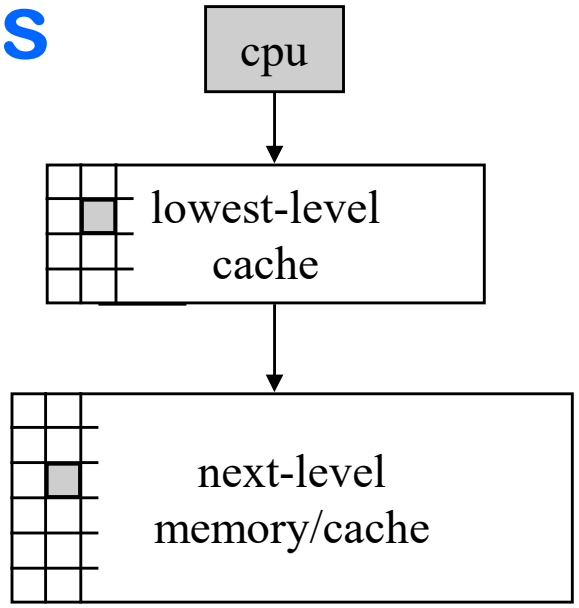
# A typical memory hierarchy






•*so then where is my program and data??*

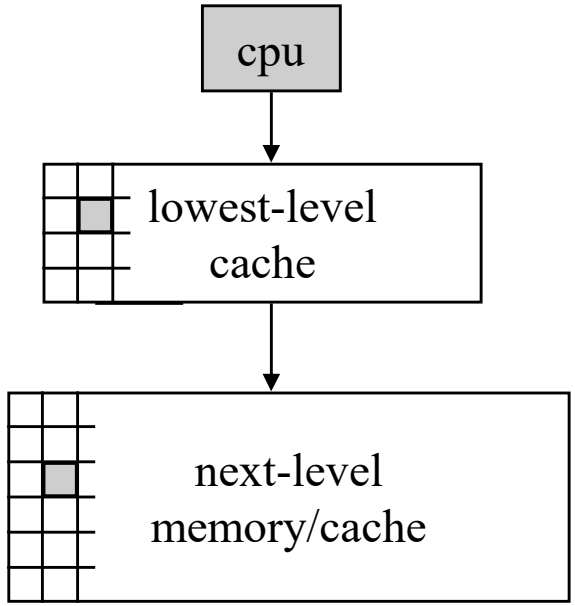
# Cache Fundamentals



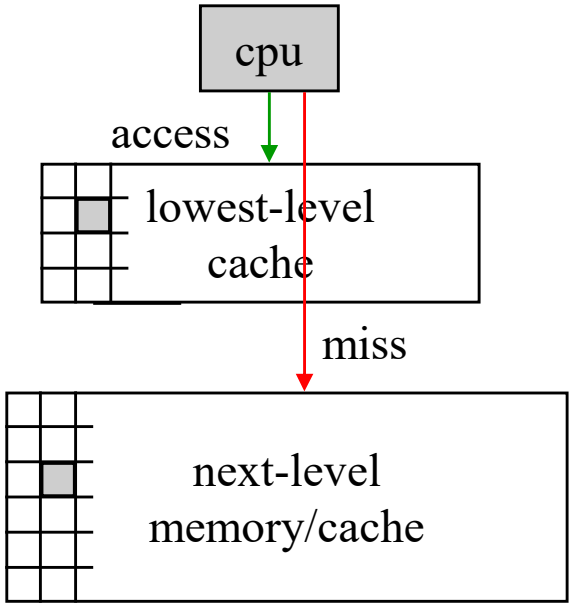

- *cache hit* -- an access where the data is found in the cache.
- *cache miss* -- an access which isn't
- *hit time* -- time to access the cache
- *miss penalty* -- time to move data from further level to closer
- *hit ratio* -- fraction of accesses where the data is found in the cache
- *miss ratio* -- (1 - hit ratio)

# Cache Fundamentals, cont.

- *cache block size* or *cache line size*– the amount of data that gets transferred on a cache miss.
- *instruction cache* -- cache that only holds instructions.
- *data cache* -- cache that only caches data.
- *unified cache* -- cache that holds both.



# Cacheing Issues




On a memory access -

- How do I know if this is a hit or miss?

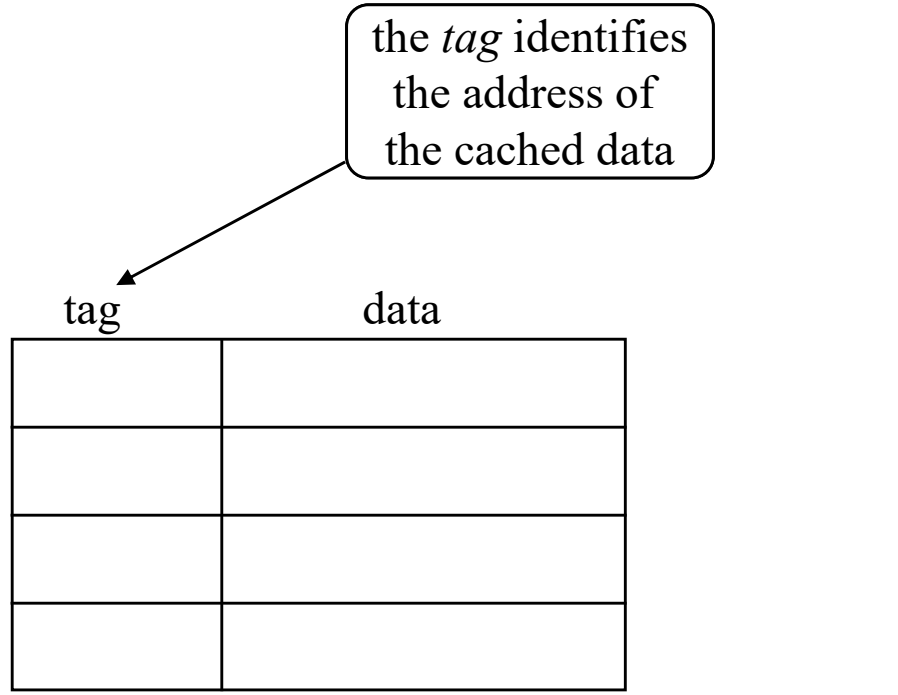
On a cache miss -

- where to put the new data?
- what data to throw out?
- how to remember what data this is?

# A simple cache

address string:

| | |
|---|---|
| 4 | 00000100 |
| 8 | 00001000 |
| 12 | 00001100 |
| 4 | 00000100 |
| 8 | 00001000 |
| 20 | 00010100 |
| 4 | 00000100 |
| 8 | 00001000 |
| 20 | 00010100 |
| 24 | 00011000 |
| 12 | 00001100 |
| 8 | 00001000 |
| 4 | 00000100 |



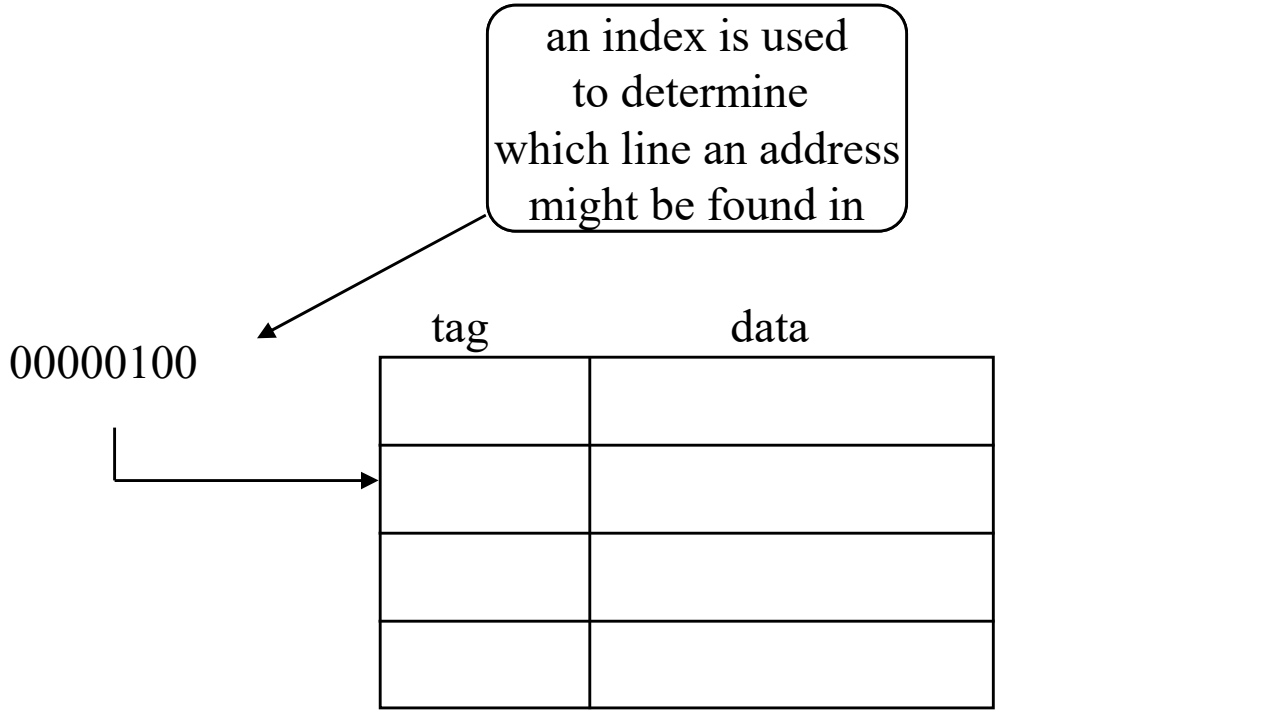

4 entries, each block holds one word, any block can hold any word.

- A cache that can put a line of data anywhere is called _____
- The most popular replacement strategy is *LRU* (  ).

# A simpler cache

address string:

| | |
|---|---|
| 4 | 00000100 |
| 8 | 00001000 |
| 12 | 00001100 |
| 4 | 00000100 |
| 8 | 00001000 |
| 20 | 00010100 |
| 4 | 00000100 |
| 8 | 00001000 |
| 20 | 00010100 |
| 24 | 00011000 |
| 12 | 00001100 |
| 8 | 00001000 |
| 4 | 00000100 |



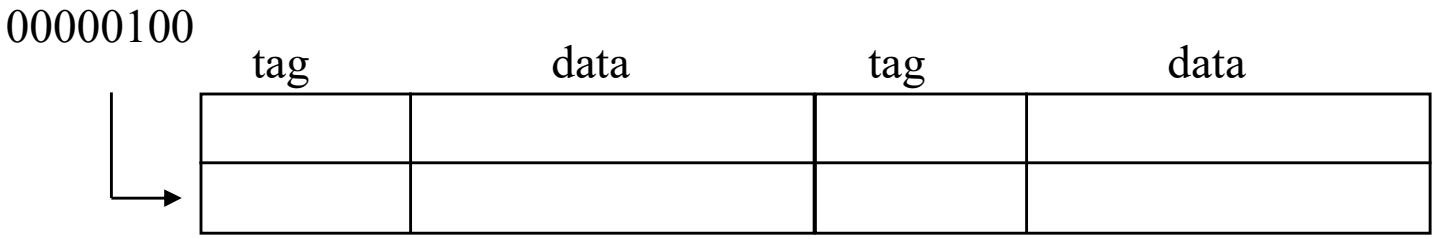

4 entries, each block holds one word, each word in memory maps to exactly one cache location.

- A cache that can put a line of data in exactly one place is called _____.
- Advantages/disadvantages vs. fully-associative?

# A set-associative cache

address string:

| | |
|---|---|
| 4 | 00000100 |
| 8 | 00001000 |
| 12 | 00001100 |
| 4 | 00000100 |
| 8 | 00001000 |
| 20 | 00010100 |
| 4 | 00000100 |
| 8 | 00001000 |
| 20 | 00010100 |
| 24 | 00011000 |
| 12 | 00001100 |
| 8 | 00001000 |
| 4 | 00000100 |



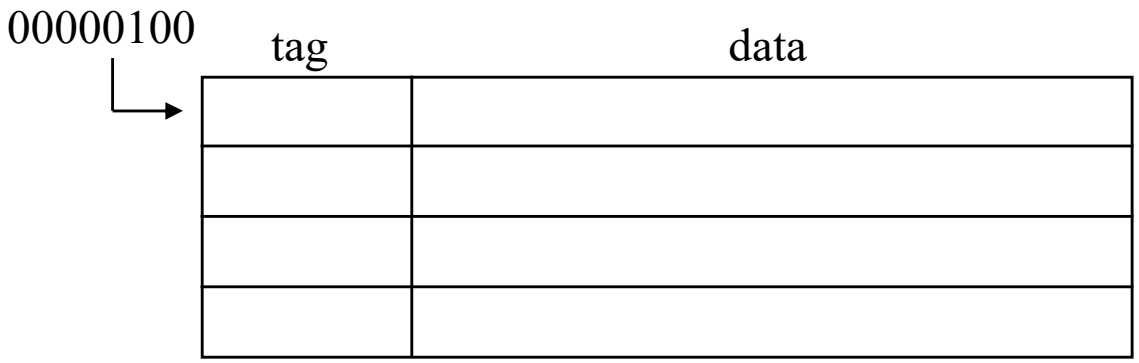

4 entries, each block holds one word, each word in memory maps to one of a set of *n* cache lines

- A cache that can put a line of data in exactly *n* places is called *n-way set-associative*.
- The cache lines/blocks that share the same index are a cache _____.

# Longer Cache Blocks

address string:

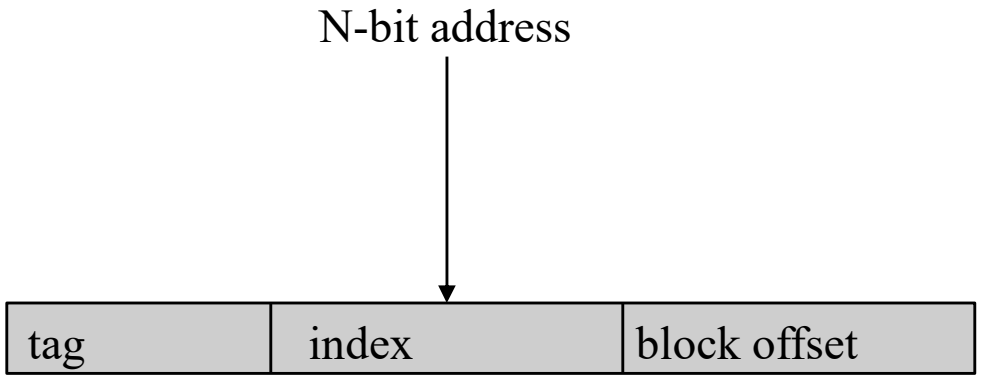
| | |
|---|---|
| 4 | 00000100 |
| 8 | 00001000 |
| 12 | 00001100 |
| 4 | 00000100 |
| 8 | 00001000 |
| 20 | 00010100 |
| 4 | 00000100 |
| 8 | 00001000 |
| 20 | 00010100 |
| 24 | 00011000 |
| 12 | 00001100 |
| 8 | 00001000 |
| 4 | 00000100 |

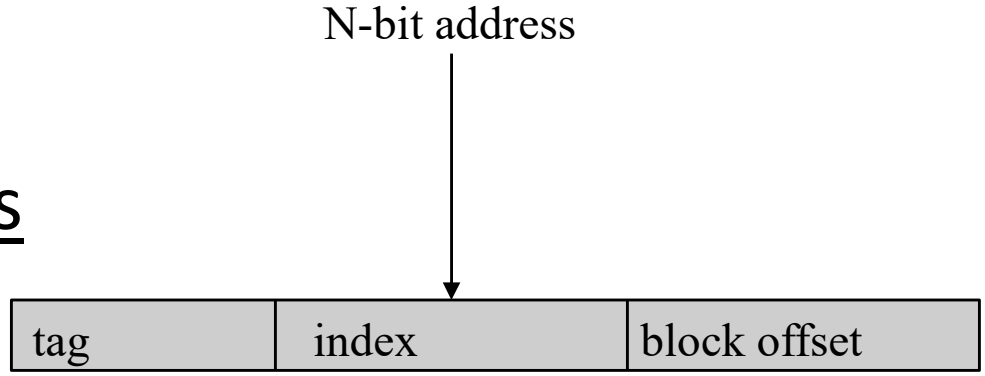



4 entries, each block holds two words, each word in memory maps to exactly one cache location (this cache is twice the total size of the prior caches).

- Large cache blocks take advantage of *spatial locality*.
- Too large of a block size can waste cache space.
- Longer cache blocks require less tag space

# Using the address…






# Equations

All "sizes" are in bytes

1. $\log_2$(block_size)
2. $\log_2$(cache_size /(assoc*block_size))
3. N – $\log_2$(cache_size/assoc)

| Selection | # tag bits | # index bits | # block offset bits |
|---|---|---|---|
| A | 3 | 2 | 1 |
| B | 1 | 2 | 3 |
| C | 1 | 3 | 2 |
| D | 2 | 1 | 3 |
| E | None of the above | | |

# Descriptions of caches

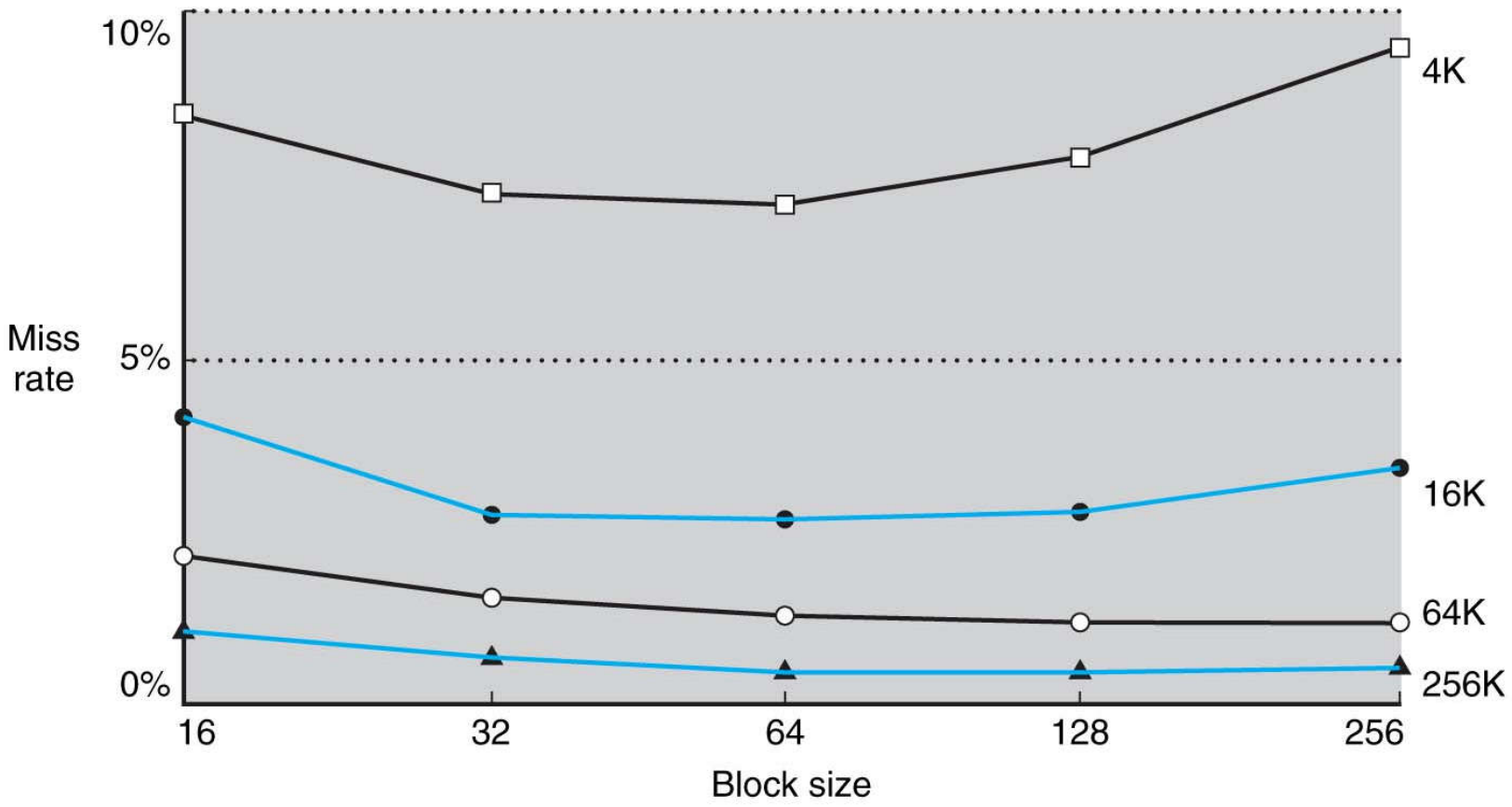
1. Exceptional usage of the cache space in exchange for a slow hit time
2. Poor usage of the cache space in exchange for an excellent hit time
3. Reasonable usage of cache space in exchange for a reasonable hit time

| Selection | Fully-Associative | 4-way Set Associative | Direct Mapped |
|---|---|---|---|
| A | 3 | 2 | 1 |
| B | 1 | 3 | 2 |
| C | 1 | 2 | 3 |
| D | 3 | 2 | 1 |
| E | None of the above | | |

# Back to Block Size

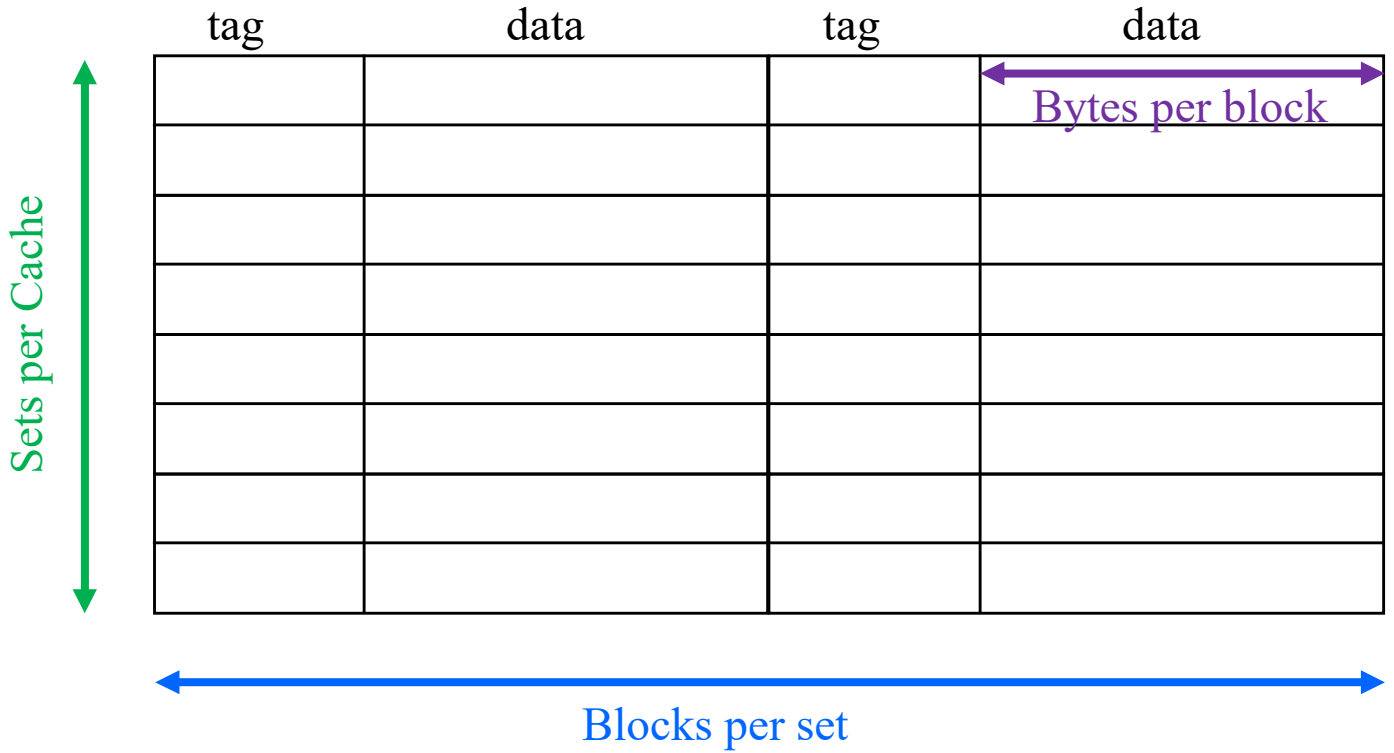
- If block size increases spatial locality, should we just make the cache block size really, really big????

# Block Size and Miss Rate



# Cache Parameters

Cache size = Number of sets * block size * associativity



# Cache Parameters

Cache size = Number of sets * block size * associativity

-128 blocks, 32-byte block size, direct mapped, size = ?

-128 KB cache, 64-byte blocks, 512 sets, associativity = ?

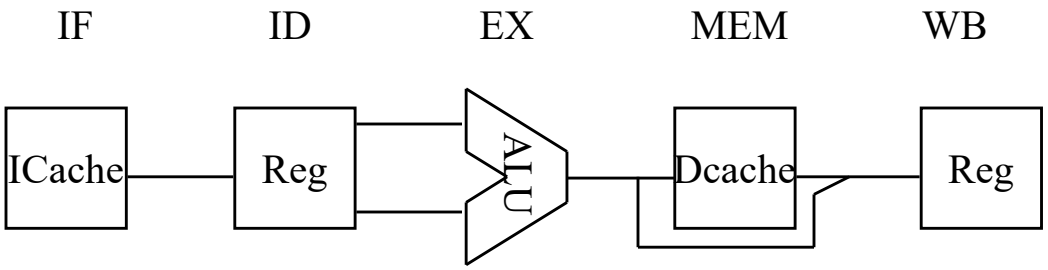
(always keep in mind “cache size” only counts the data storage)

# Handling a Cache Access

1. Use index and tag to access cache and determine hit/miss.
2. If hit, return requested data.
3. If miss, select a cache block to be replaced, and access memory or next lower cache (possibly stalling the processor).

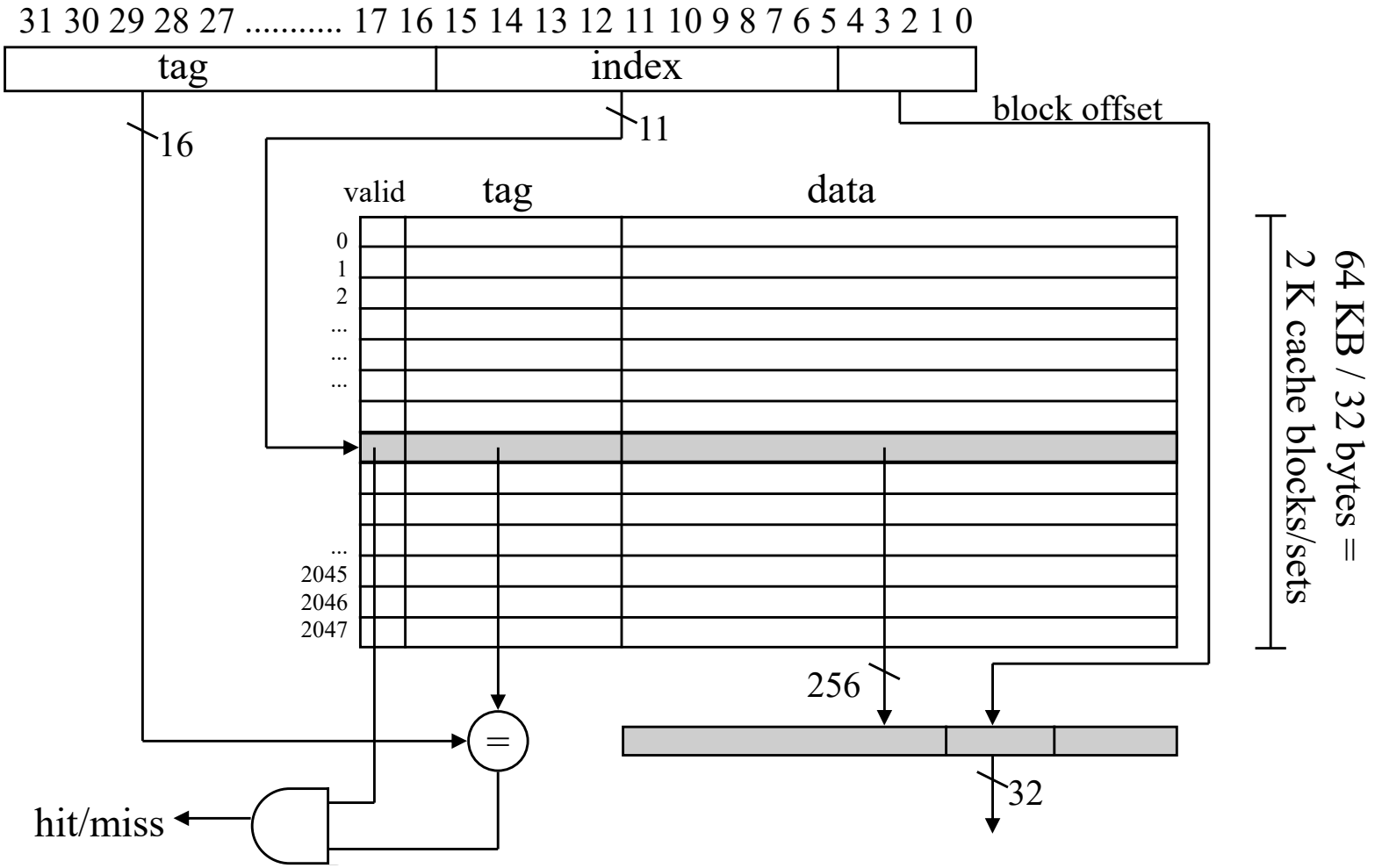
   -load entire missed cache line into cache

   -return requested data to CPU (or higher cache)
4. If next lower memory is a cache, goto step 1 for that cache.



# Accessing a Sample Cache

- 64 KB cache, direct-mapped, 32-byte cache block size

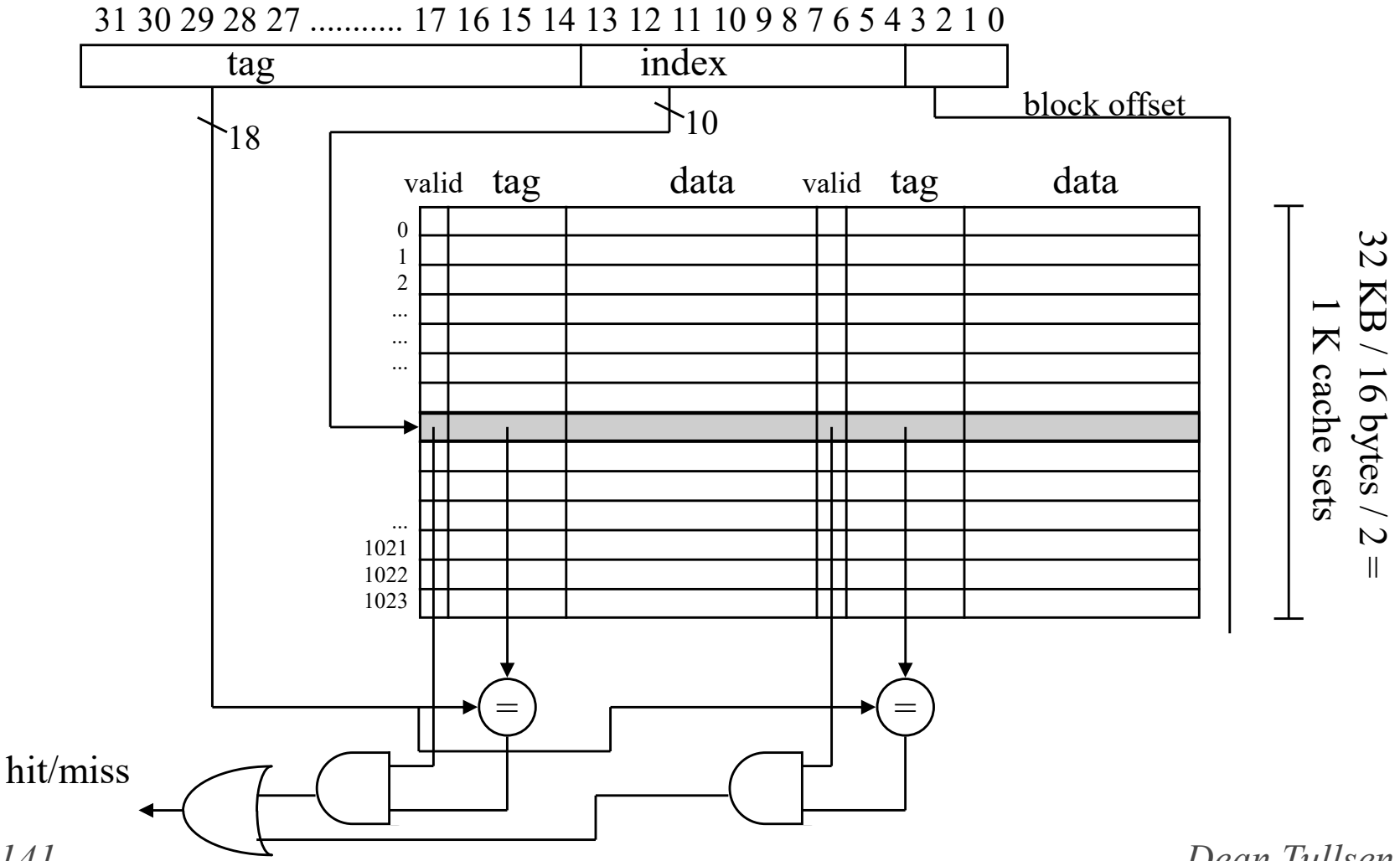





e.g., assume address 0b01100100111001010000101110101100

# Accessing a Sample Cache

- 32 KB cache, 2-way set-associative, 16-byte block size



# Associative Caches

- Higher hit rates, but...

- longer access time (longer to determine hit/miss, more muxing of outputs)
- more space (longer tags)
  - 16 KB, 16-byte blocks, dm, tag = ?
  - 16 KB, 16-byte blocks, 4-way, tag = ?

# Dealing with Stores

- Stores must be handled differently than loads, because...
  - they don't necessarily require the CPU to stall.
  - they change the content of cache/memory (creating memory *consistency* issues)
  - may require a and a write to memory to complete

# Policy decisions for stores

- Keep memory and cache identical?
  - => all writes go to both cache and main memory
  - => writes go only to cache. Modified cache lines are written back to memory when the line is replaced.
- Make room in cache for store miss?
  - *write-allocate* => on a store miss, bring written line into the cache
  - *write-around* => on a store miss, ignore cache

# Dealing with stores

- On a store hit, write the new data to cache. In a *write-through* cache, write the data immediately to memory. In a *write-back* cache, mark the line as dirty.
- On a store miss, initiate a cache block load from memory for a write-allocate cache. Write directly to memory for a write-around cache.
- On any kind of cache miss in a write-back cache, if the line to be replaced in the cache is dirty, write it back to memory.

# Reminder: Handling stalls in the ET equation

- Should be incorporated into CPI (ie CPI = BCPI + …)

- Eg, we've already seen, or at least discussed: LHSPI (load hazard), BHSPI (branch hazard)

# Cache Performance

**CPI = BCPI + MCPI**

- BCPI = base CPI, which means the CPI assuming perfect memory
- MCPI = the memory CPI, the number of cycles (per instruction) the processor is stalled waiting for memory.

**MCPI = accesses/instruction * miss rate * miss penalty**

- this assumes we stall the pipeline on both read and write misses, that the miss penalty is the same for both, that cache hits require no stalls.
- If the miss penalty or miss rate is different for Inst cache and data cache (common case), then

MCPI = I$ accesses/inst*I$MR*I$MP + D$ acc/inst*D$MR*D$MP

# And putting it, again, all together…

- Can generalize this formula further for other stalls:

- CPI = BCPI + DHSPI + BHSPI + MCPI
  - DHSPI = data hazard stalls per instruction
  - BHSPI = branch hazard stalls per instruction.

# Cache Performance

- Instruction cache miss rate of 4%, data cache miss rate of 10%, BCPI = 1.0, 20% of instructions are loads and stores, miss penalty = 12 cycles, CPI = ?

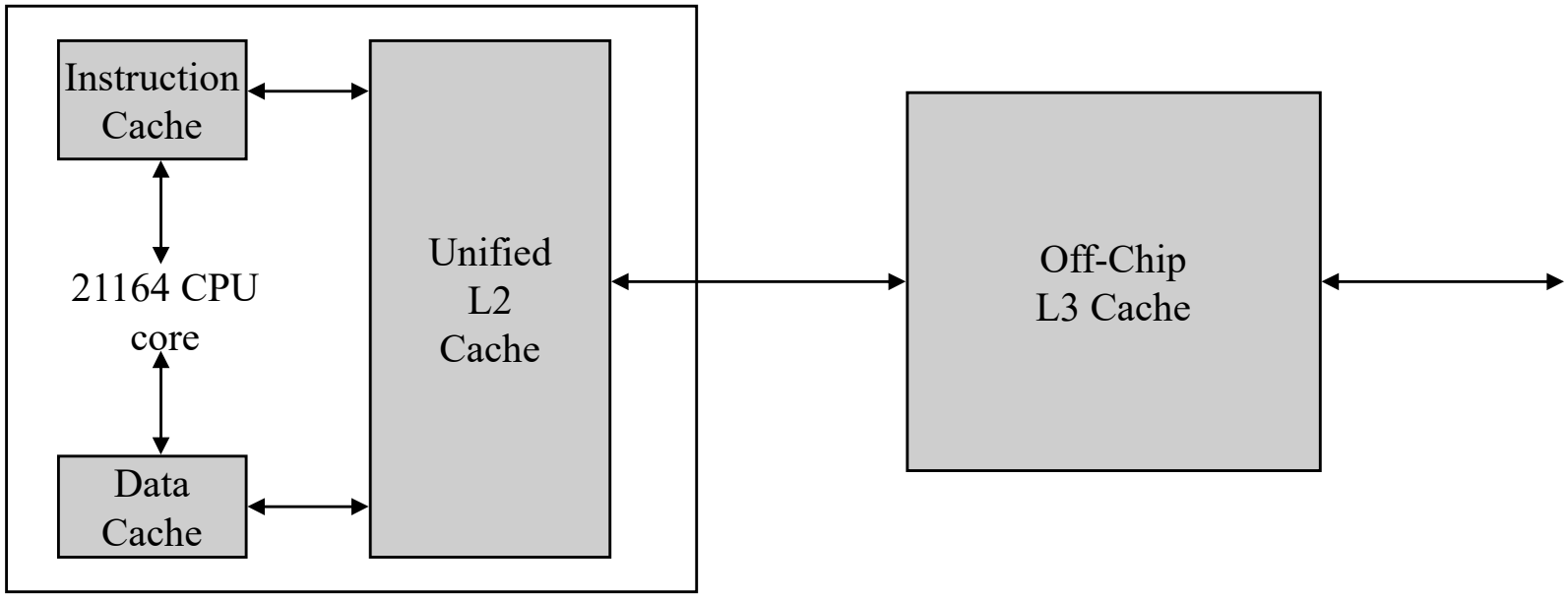
# Cache Performance

- Unified cache, 25% of instructions are loads and stores, BCPI = 1.2, miss penalty of 10 cycles. If we improve the miss rate from 10% to 4% (e.g. with a larger cache), how much do we improve performance?

# Cache Performance

- BCPI = 1, miss rate of 8% overall, 20% loads, miss penalty 20 cycles, never stalls on stores. What is the speedup from doubling the cpu clock rate?

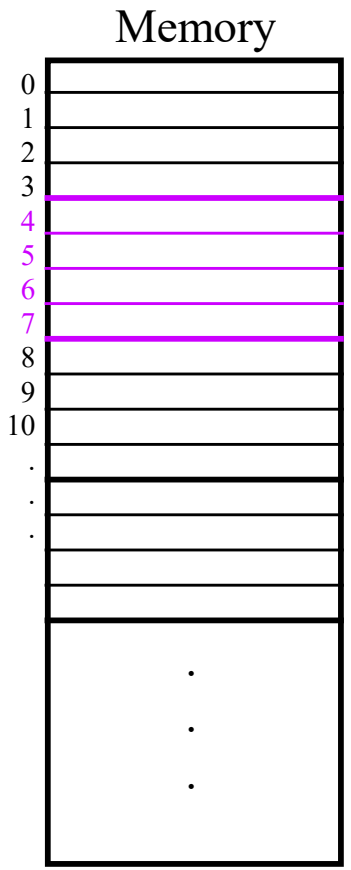
# Example -- DEC Alpha 21164 Caches




- ICache and DCache -- 8 KB, DM, 32-byte lines
- L2 cache -- 96 KB, ?-way SA, 32-byte lines
- L3 cache -- 1 MB, DM, 32-byte lines

# Cache Alignment

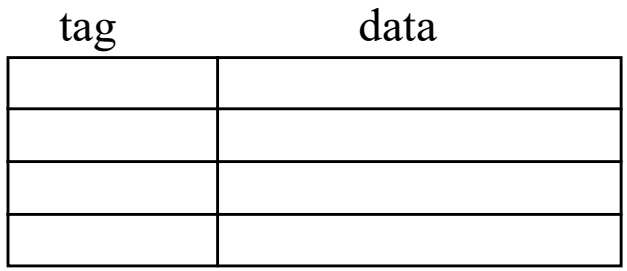

- The data that gets moved into the cache on a miss are all data whose addresses share the same tag and index (regardless of which data gets accessed first).
- This results in
  - no overlap of cache lines
  - easy mapping of addresses to cache lines (no additions)
  - data at address X always being present in the same location in the cache block (at byte *X mod blocksize*) if it is there at all.
- Think of main memory as organized into cache-line sized pieces (because in reality, it is!).



# Three types of cache misses

- Compulsory (or cold-start) misses
  - first access to the data.
- Capacity misses
  - we missed only because the cache isn't big enough.
- Conflict misses
  - we missed because the data maps to the same line as other data that forced it out of the cache.

| address string: | |
|---|---|
| 4 | 00000100 |
| 8 | 00001000 |
| 12 | 00001100 |
| 4 | 00000100 |
| 8 | 00001000 |
| 20 | 00010100 |
| 4 | 00000100 |
| 8 | 00001000 |
| 20 | 00010100 |
| 24 | 00011000 |
| 12 | 00001100 |
| 8 | 00001000 |
| 4 | 00000100 |



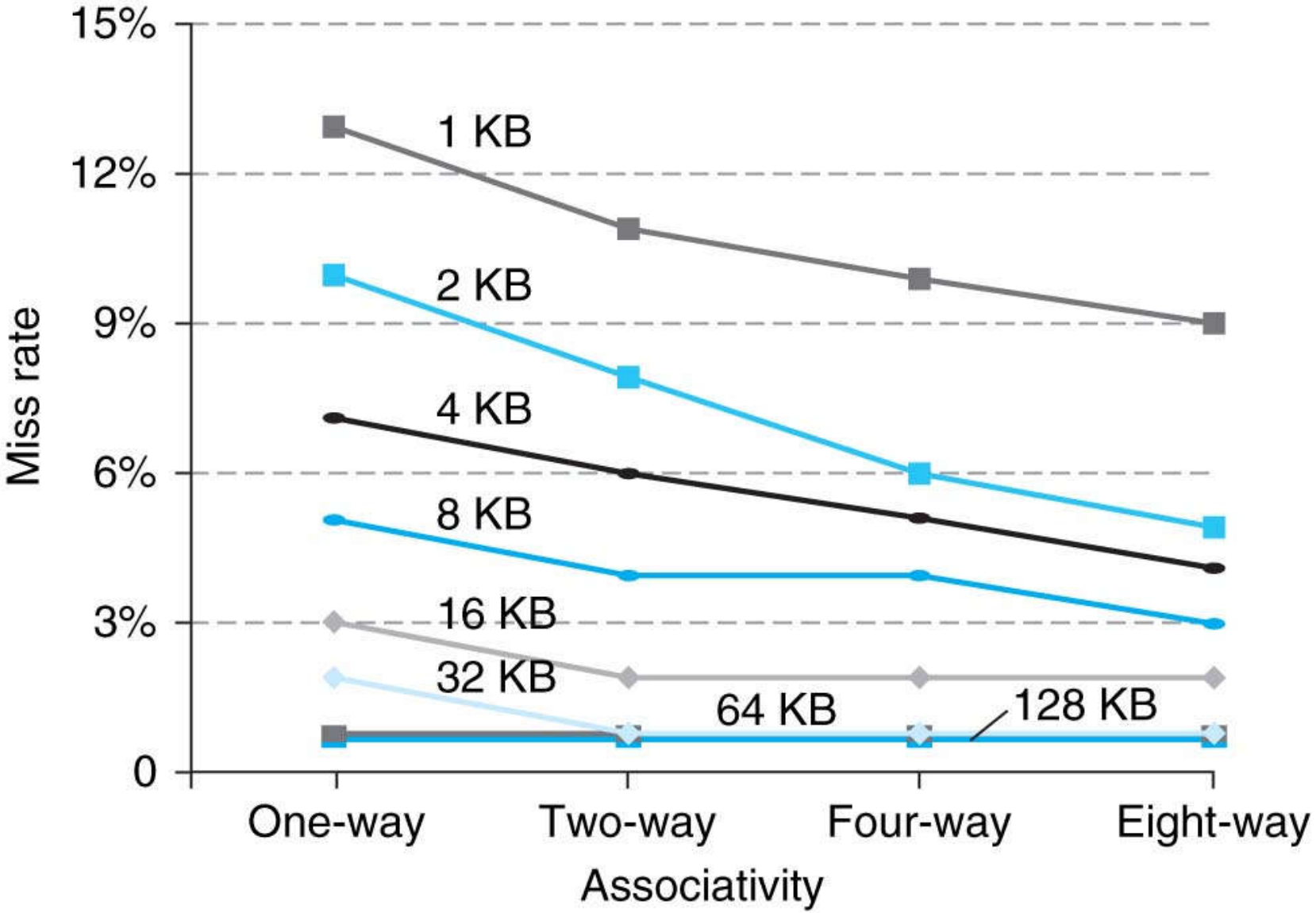

DM cache

- Suppose you experience a cache miss on a block (let's call it block A). You have accessed block A in the past. There have been precisely 1027 different blocks accessed between your last access to block A and your current miss. Your block size is 32-bytes and you have a 64KB cache. What kind of miss was this?

# So, then, how do we decrease…

- Compulsory misses?
- Capacity misses?
- Conflict misses?

# Cache Associativity



# LRU replacement algorithms

- only needed for associative caches
- requires one bit for 2-way set-associative, 8 bits (per set, 2/line) for 4-way, 24 bits for 8-way.
- can be emulated with log n bits (NMRU)
- can be emulated with *use* bits for highly associative caches (like page tables)

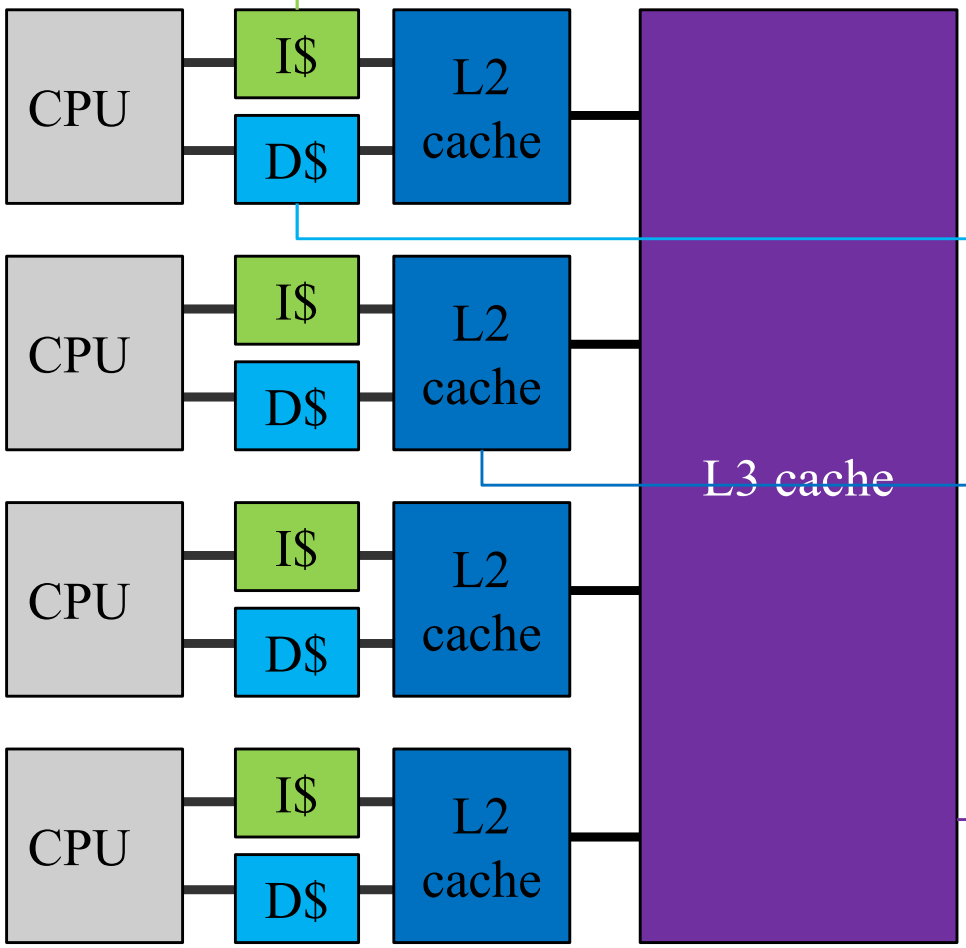
- Modern CPUs (which tend to have high-associativity caches) appear to use "pseudo-LRU" (better than NMRU, worse than true LRU), that requires less recordkeeping, lookups, and comparisons by treating the set as a binary tree.

# Caches in Current Processors

- Not long ago, they were DM at lowest level (closest to CPU), associative further away. Today they are *less* associative near the processor (4-8), and *more* associative farther away (up to 16).
- split I and D close (L1) to the processor (for throughput rather than miss rate), unified further away (L2 and beyond).
- write-through and write-back both common, but never write-through all the way to memory.
- 64-byte and 128-byte cache lines common.

- Non-blocking
  - processor doesn't stall on a miss, but only on the use of a miss (if even then)
  - this means the cache must be able to keep track of multiple outstanding accesses, even multiple outstanding misses.

# Intel Skylake




Instruction Cache
-32 KB, 8-way (Nehalem 4-way)
-64-byte line

Data Cache
-32 KB, 8-way
-64-byte line
-write-back, write-allocate
-4 cycle latency!

Unified L2 Cache
-256 KB, 4-way (Nehalem 8-way)
-64-byte line
-write-back, write-allocate
-12 cycle latency

Shared, unified L3 Cache
-8 MB, 16-way (for 4 cores)
-64-byte line
-write-back, write-allocate
-42 cycle latency

Memory latency = 42 cycles (L3 hit time) + 51ns
= 246 cycles at 4 GHz

# Key Points

- Caches give illusion of a large, cheap memory with the access time of a fast, expensive memory.
- Caches take advantage of memory locality, specifically temporal locality and spatial locality.
- Cache design presents many options (block size, cache size, associativity, write policy) that an architect must combine to minimize miss rate and access time to maximize performance.