Principles of Computer Architecture

CSE 240A

Fall 2024

Hadi Esmaeilzadeh

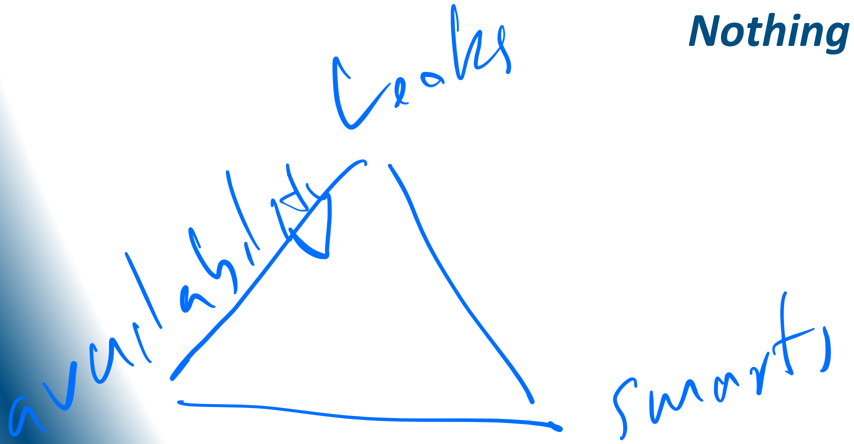
hadi@ucsd.edu

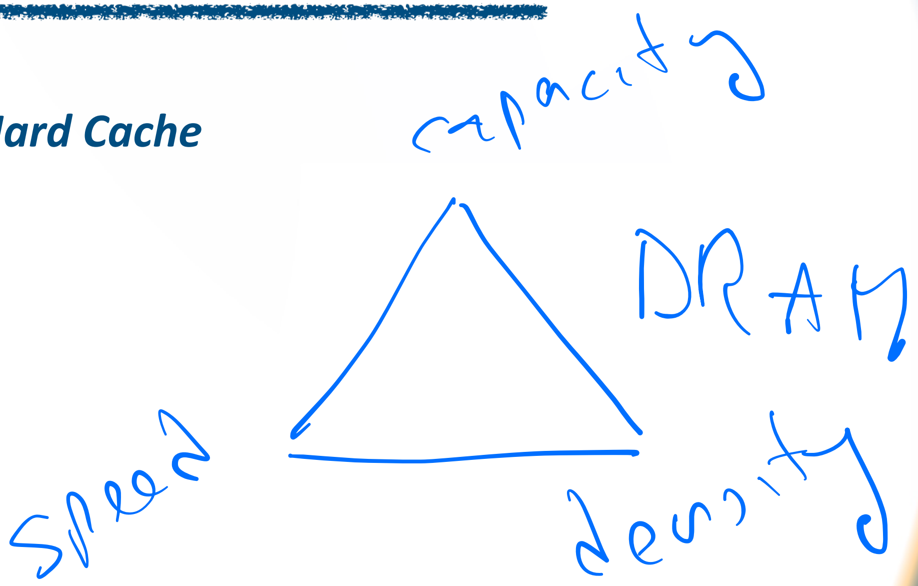
University of California, San Diego

Memory Subsystem Design
Caches – Part 1

or

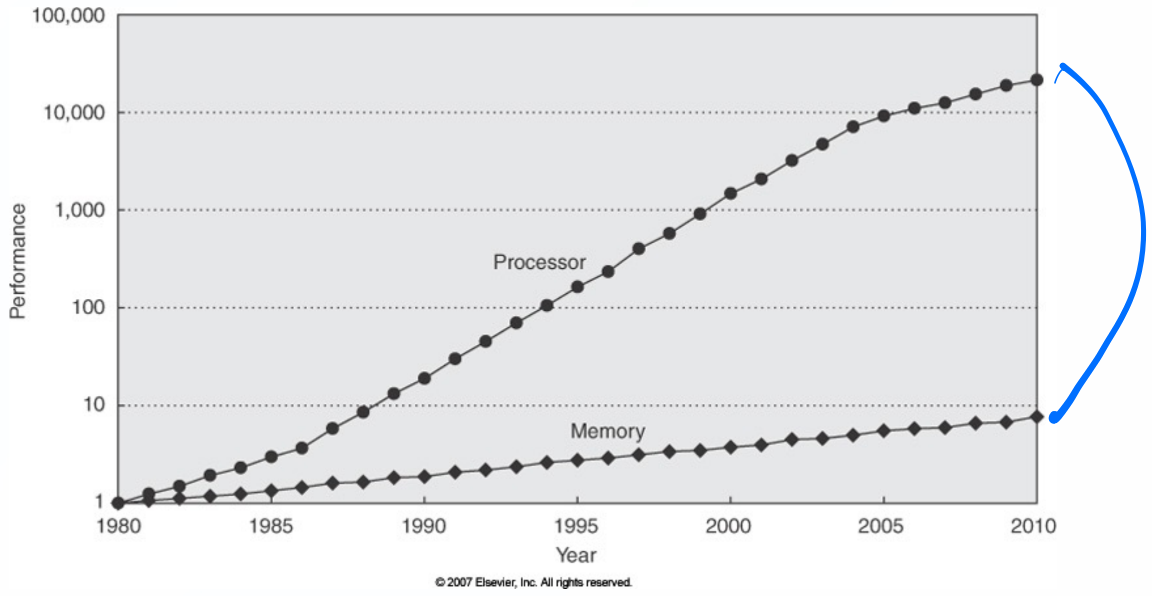
Nothing Beats Cold, Hard Cache





Who Cares about Memory Hierarchy?

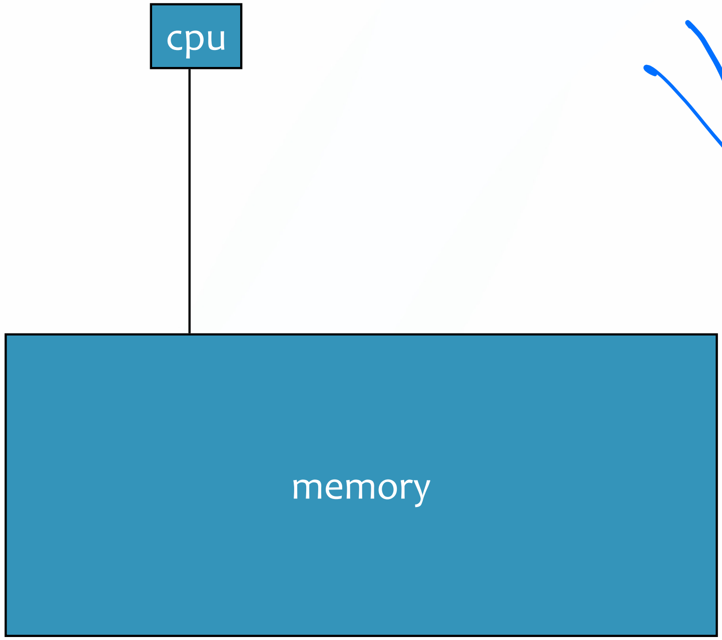
- Processor Only Thus Far in Course



1980: no cache in µproc;

1995 2-level cache, 60% trans. on Alpha 21164 µproc

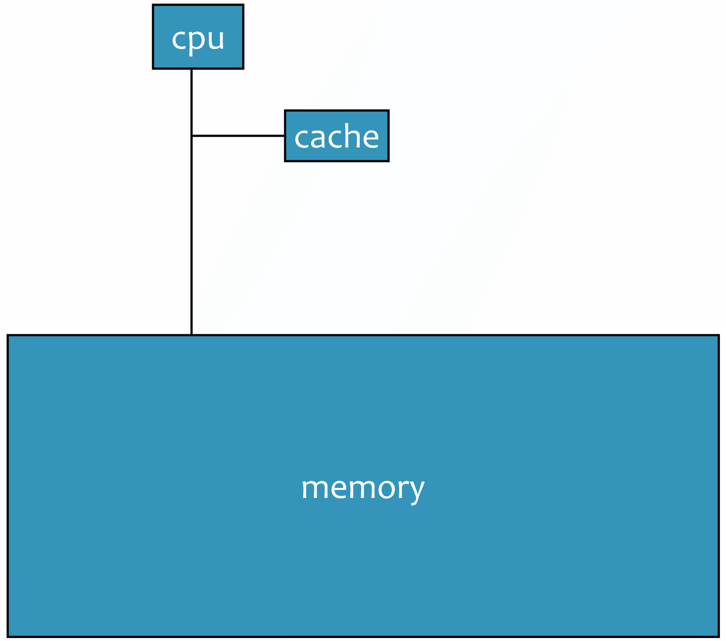
Memory Cache



Memory Wall

Value prediction

Memory Cache



- Can put small, fast memory close to processor.
- What do we put there?

Memory Locality

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

1,2,3,1,2,3,8,8,47,9,10,8,8...

Locality and cacheing

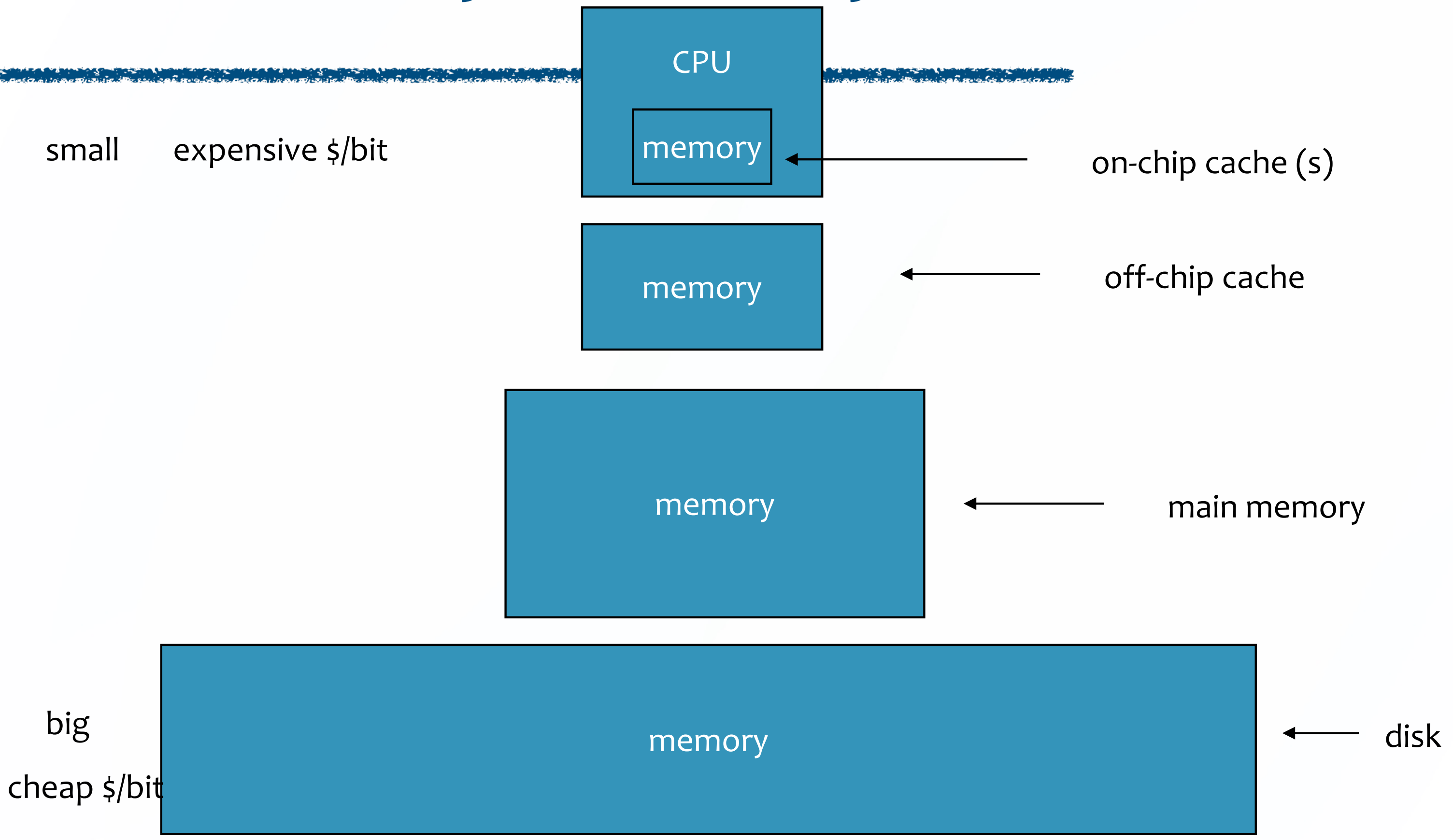
- Memory hierarchies exploit locality by *cacheing* (keeping close to the processor) data likely to be used again.
- This is done because we can build large, slow memories and small, fast memories, but we can't build large, fast memories.
- If it works, we get the illusion of SRAM access time with disk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

SRAM (static RAM) -- 5-20 ns access time, very expensive (onchip much faster -- < 1 ns)

DRAM (dynamic RAM) -- 60-100 ns, cheaper

disk -- access time measured in milliseconds, very cheap

A typical memory hierarchy

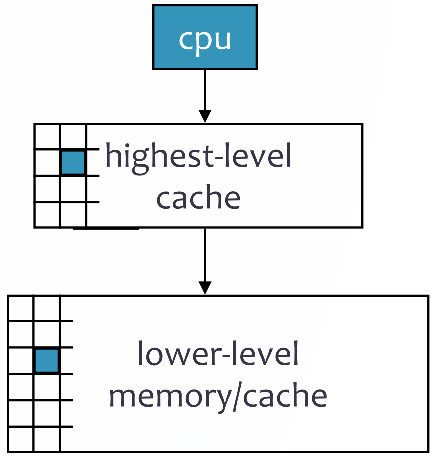


- *so then where is my program and data??*

The programmer has a logical view of memory that has little to do with reality.

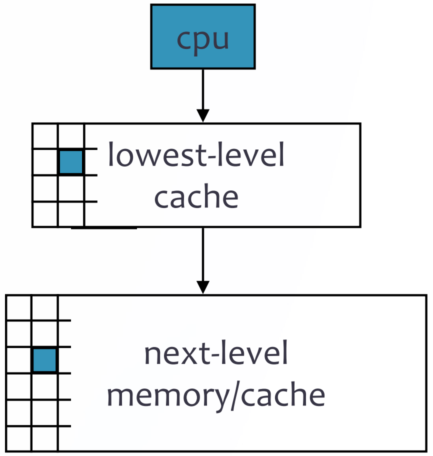
Cache Fundamentals

- *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 higher cache
- *miss penalty* -- time to move data from lower level to upper, then to cpu
- *hit ratio* -- percentage of time the data is found in the higher cache
- *miss ratio* -- (1 - hit ratio)



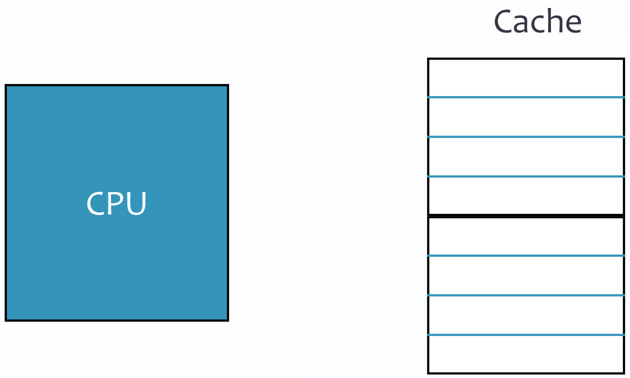
Cache Fundamentals, cont.

- *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.



Accessing a simple cache

- blocksize = 4 words (16 bytes), cache size = 2 blocks (32 bytes), associativity = full



CPU reads addresses 8, 0, 4, 0, 4, 8, 12, 16, 20,

write val 30 to address 12,

reads 8, 20, 28, 56, 20, 60, 12

		Memory
0000 00 00	0	63
0000 01 00	4	37
0000 10 00	8	4
0000 11 00	12	149
0001 00 00	16	12
0001 01 00	20	82
0001 10 00	24	2
0001 11 00	28	3
0010 00 00	32	4
0010 01 00	36	5
0010 10 00	40	17
0010 11 00	44	3245
0011 00 00	48	63
0011 01 00	52	37
0011 10 00	56	4
0011 11 00	60	149
0100 00 00	64	12
0100 01 00	68	82
0100 10 00	72	21
0100 11 00	76	92

Cache Characteristics

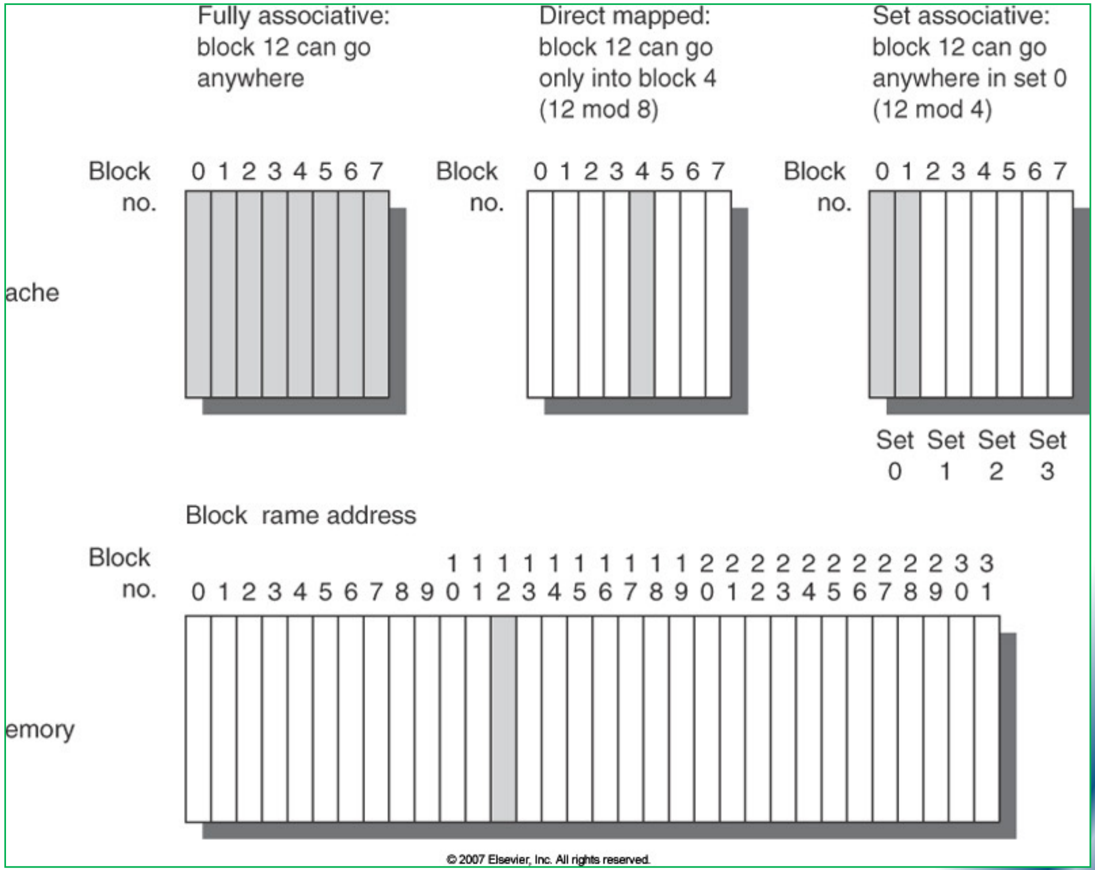
- Cache Organization (size, associativity, block size)
- Cache Access
- Cache Replacement
- Write Policy

Cache Organization: Where can a block be placed in the cache?

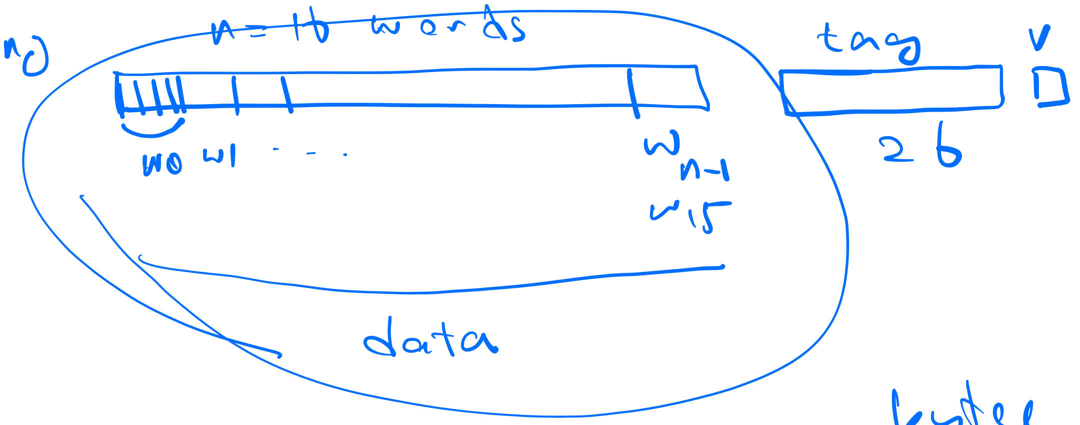
- Block 12 placed in 8-block cache:
  - Fully associative, direct mapped, n-way set associative
  - index = pointer to the set in the cache where a memory location might be cached

  (associativity = degree of freedom in placing a particular block of memory)

  (set = a collection of cache blocks with the same cache index)

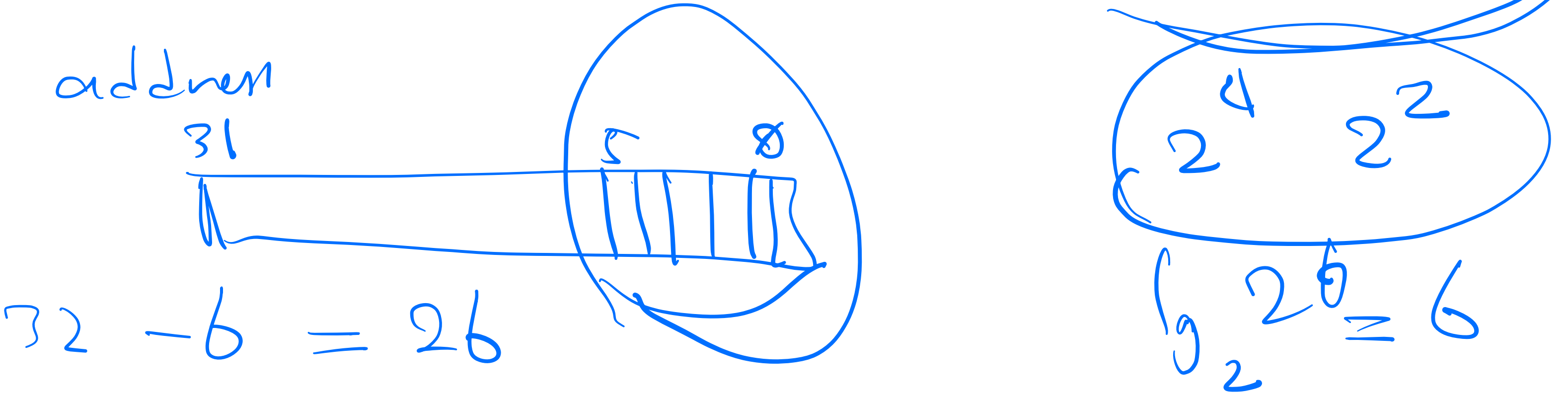

Fully associative: block 12 can go anywhere
Direct mapped: block 12 can go only into block 4 (12 mod 8)
Set associative: block 12 can go anywhere in set 0 (12 mod 4)
Block no.
0 1 2 3 4 5 6 7
ache
Set 0 Set 1 Set 2 Set 3
Block rame address
Block no.
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
emory
© 2007 Elsevier, Inc. All rights reserved.

Byte addressing



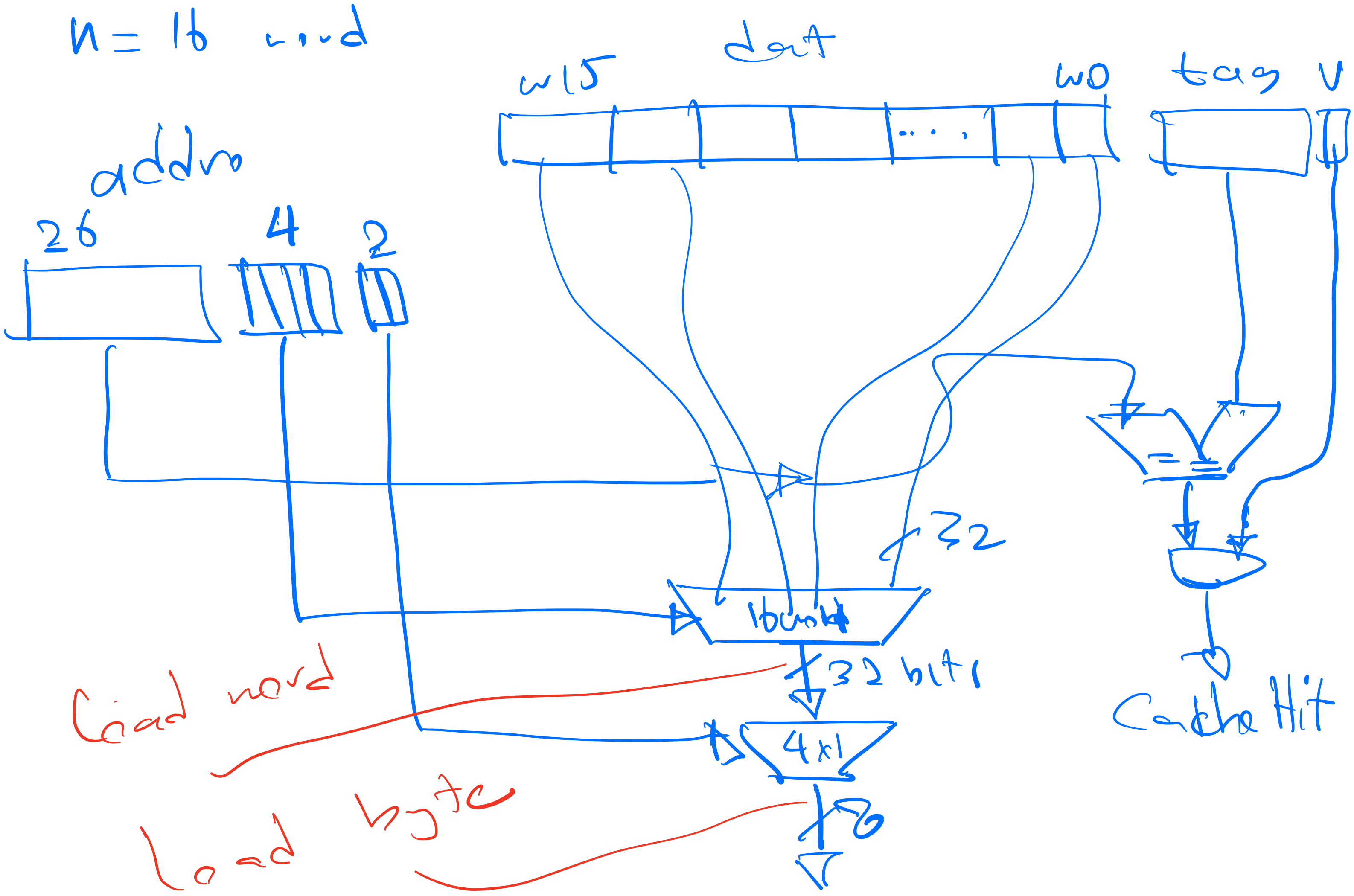
tag size

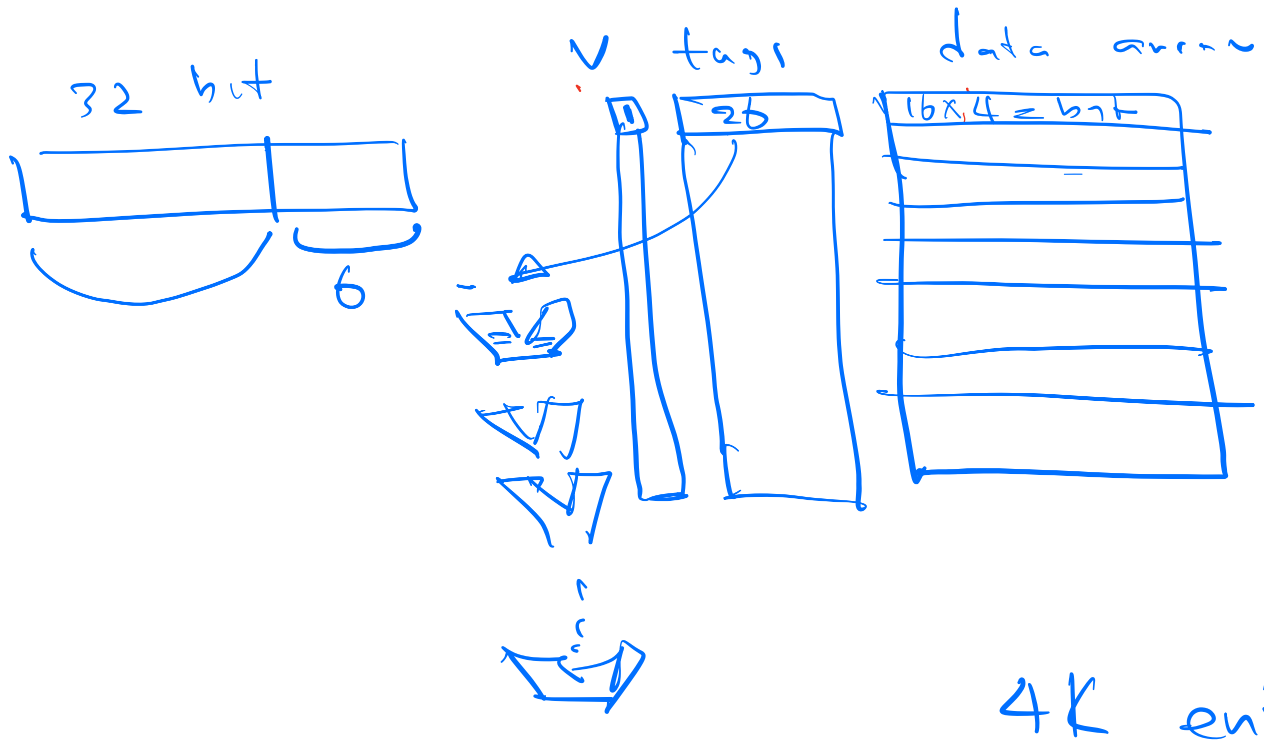
16 × 4 bytes



$32 - 6 = 26$

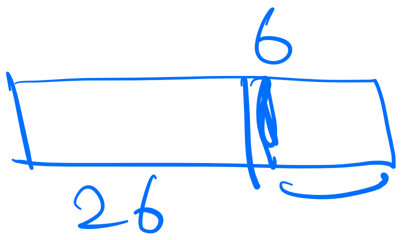
n = 16 word





16 word Cache

4K entries cache line of size 16 words



Total Cache Size

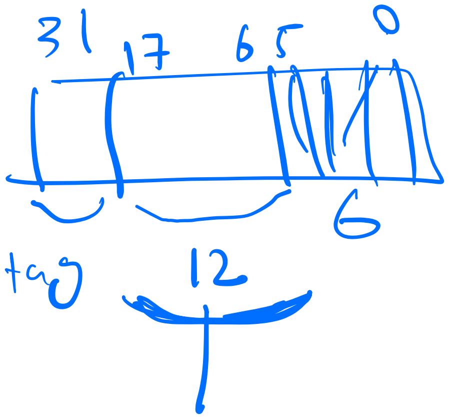
$$\underbrace{4 \times 1024 \times 16 \times 4}_{\text{data array}} + \underbrace{4 \times 1024 \times \frac{\overset{14}{\cancel{26}}}{32} \times 4}_{}$$

tag

$$+ \underbrace{4 \times 1024 \times \frac{1}{32} \times 4}_{\text{Valid}}$$

16 word caching

Byte Addressable

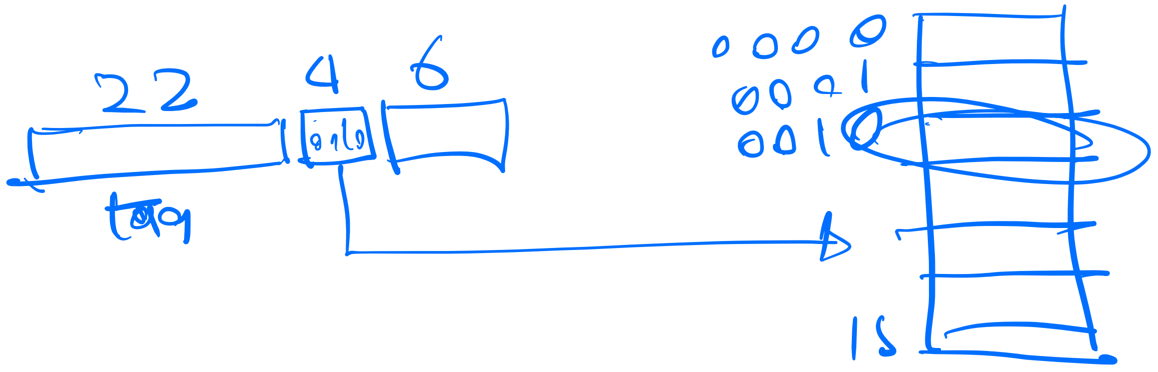


4 K Entries

$$\frac{\log_2 2^{2} \times 2^{10}}{2} = 12$$

Direct Map

$6+12 \rightarrow 32 - 12 = 14$ tag size



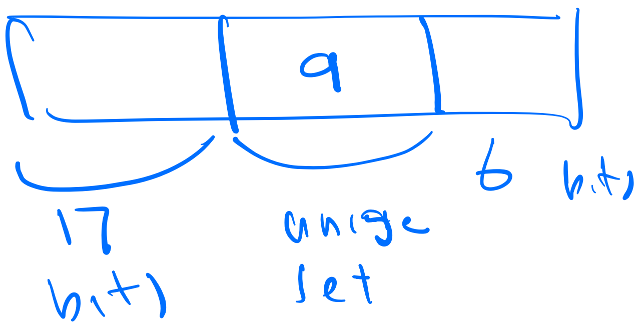
4 K entries

16 word cache line

Byte addressing

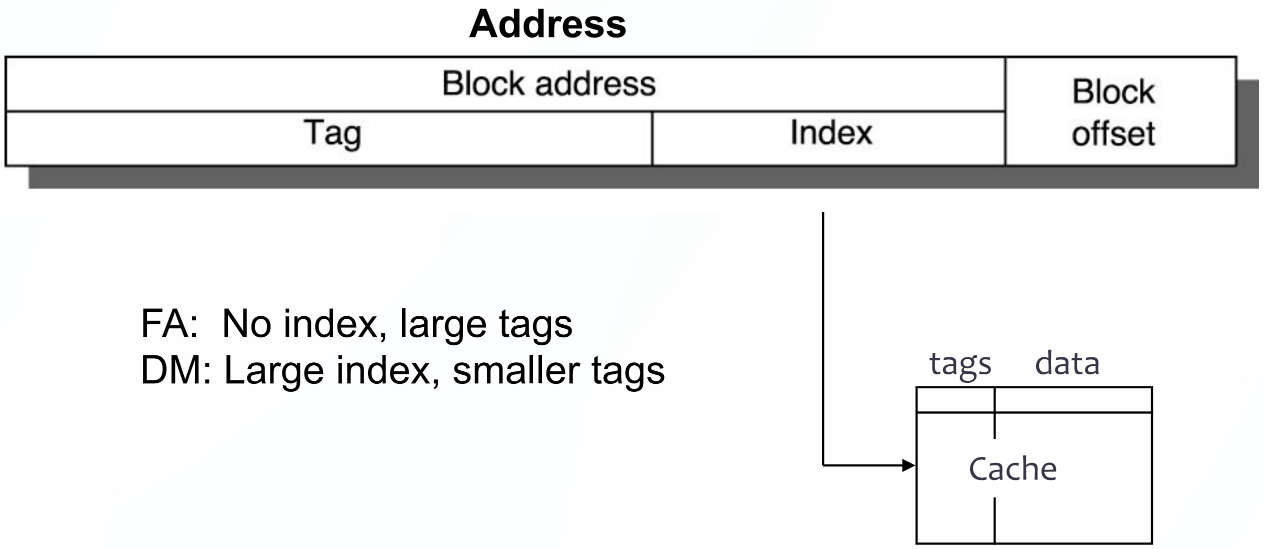
8-way set

$$\frac{2^2 \times 2^{10}}{\boxed{8 = 2^3}} = 2^9$$ unige sets



Cache Access: How Is a Block Found In the Cache?

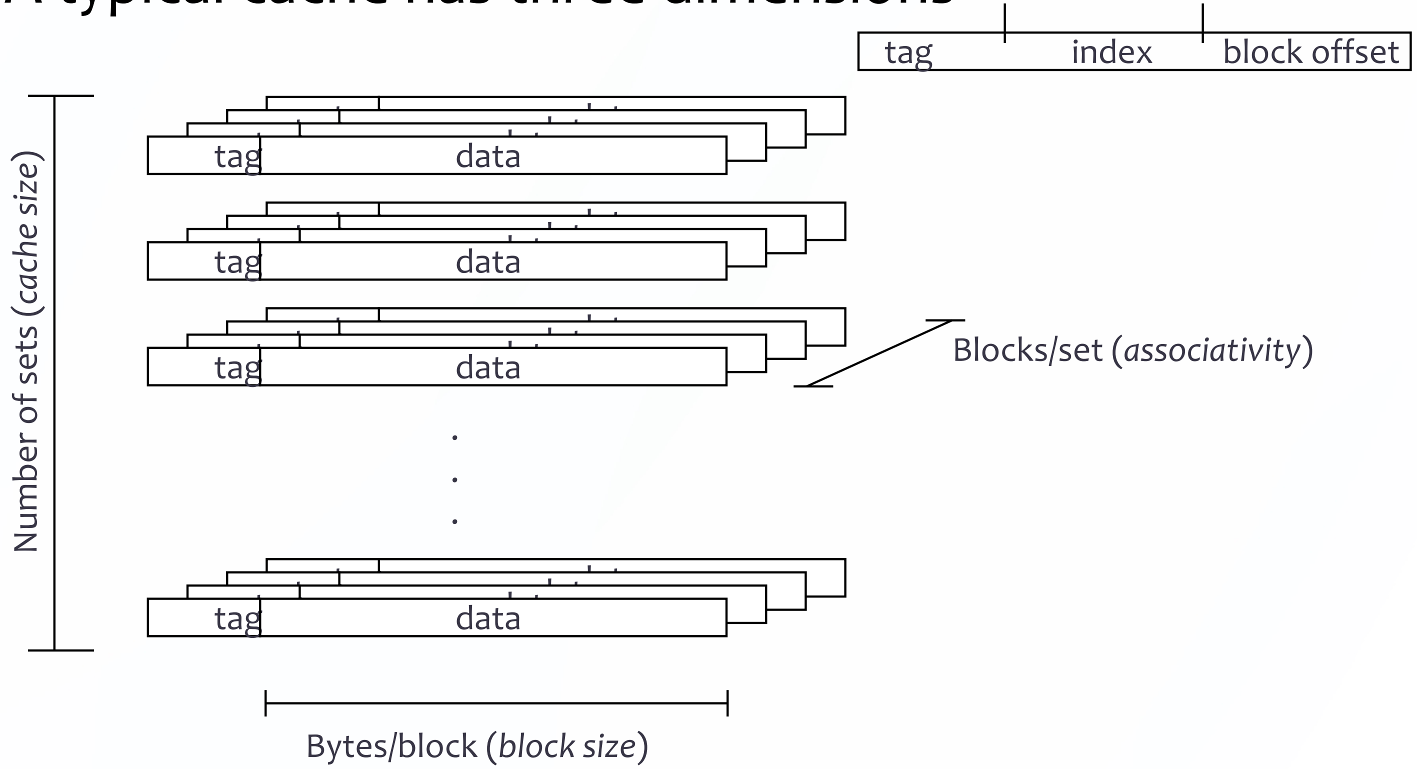
- Tag on each block
  - No need to check index or block offset
- Increasing associativity shrinks index, expands tag



FA: No index, large tags
DM: Large index, smaller tags

Cache Organization -- Overview

- A typical cache has three dimensions

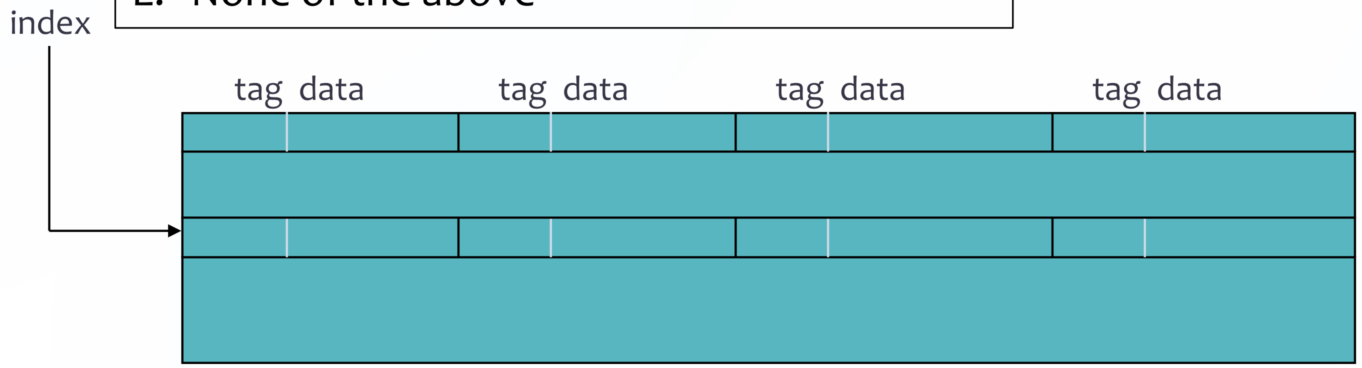

tag
index
block offset
tag
data
Number of sets (*cache size*)
Blocks/set (*associativity*)
Bytes/block (*block size*)

Cache Access

- 16 KB, 4-way set-associative cache, 32-bit address, byte-addressable memory, 16-byte cache blocks/lines

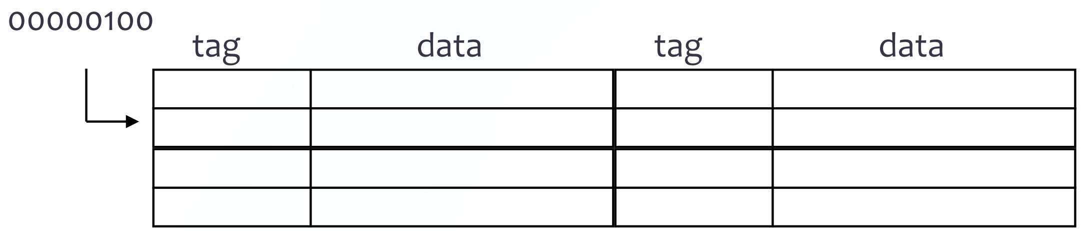
At what index would you find the word at address 0x200356A4?

A. 0x56
B. 0x6A
C. 0xA4
D. 0x56A
E. None of the above



A set-associative cache

address string:	
4	0000 01 00
8	0000 10 00
12	0000 11 00
4	0000 01 00
8	0000 10 00
20	0001 01 00
4	0000 01 00
8	0000 10 00
20	0001 01 00
24	0001 10 00
36	0010 01 00
4	0000 01 00
20	0001 01 00

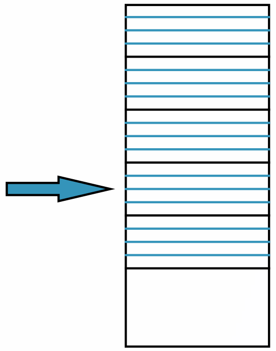


4 entries, each block holds one word, each word in memory maps to one of a set of *n* cache lines

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

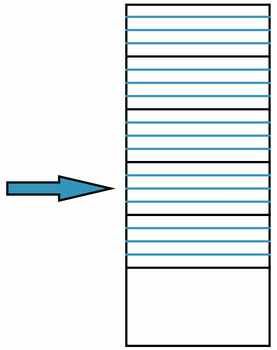
Which Block Should be Replaced on a Miss?

- Direct Mapped is Easy
- Set associative or fully associative:
  - longest till next use (ideal, impossible)
  - least recently used (best practical approximation)
  - pseudo-LRU (e.g., NMRU, NRU)
  - random (easy)
  - how many bits for LRU?



Which Block Should be Replaced on a Miss?

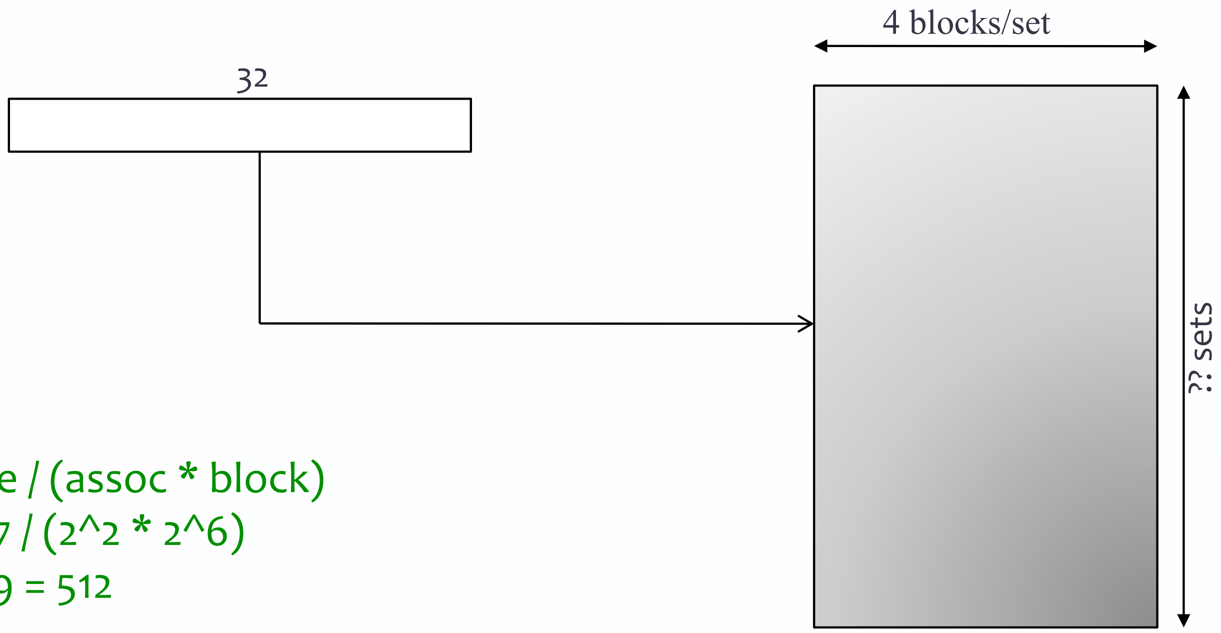
- Direct Mapped is Easy
- Set associative or fully associative:
  - longest till next use (ideal, impossible)
  - least recently used (best practical approximation)
  - pseudo-LRU (e.g., NMRU, NRU)
  - random (easy)
  - how many bits for LRU?



Associativity:	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Cache Access

- 32-bit address, and 128 KB cache with 64-byte blocks, 4-way set associative.



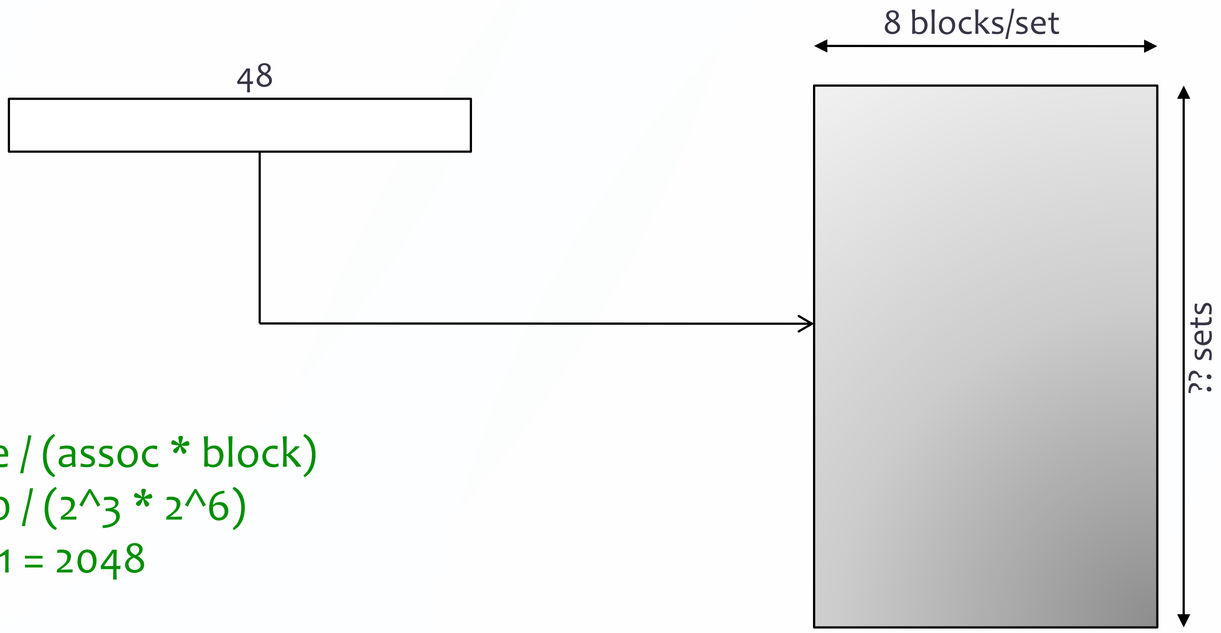
Sets = size / (assoc * block)

2^17 / (2^2 * 2^6)

Sets = 2^9 = 512

Cache Access

- 48-bit address, and 1 MB cache with 64-byte blocks, 8-way set associative.



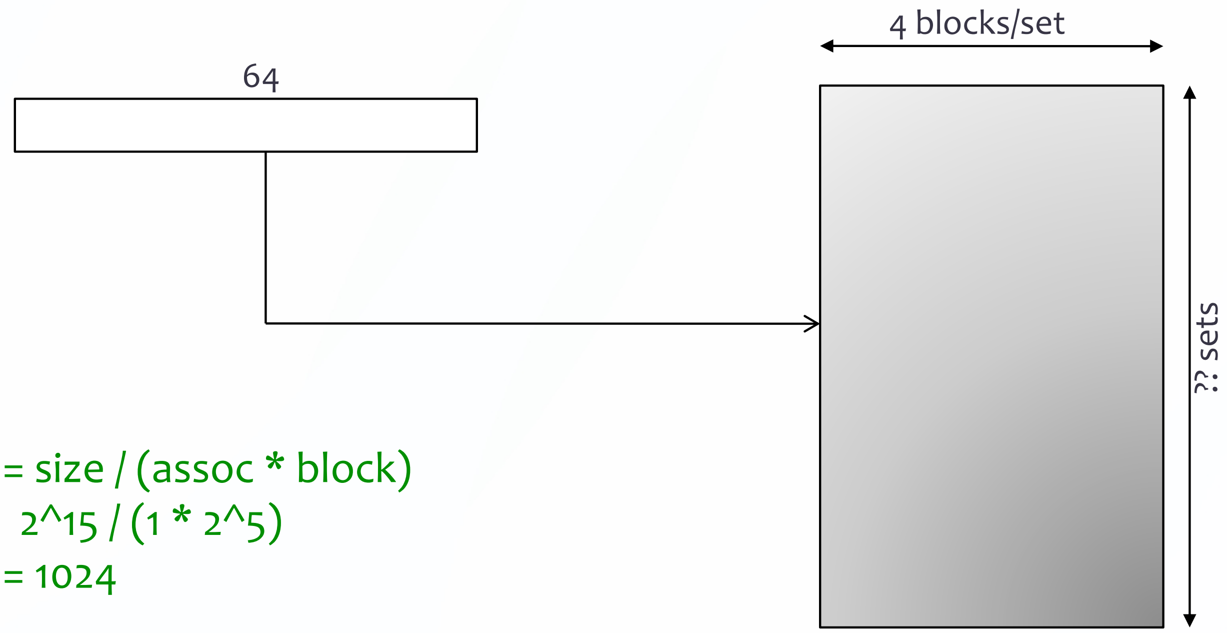
Sets = size / (assoc * block)

2^20 / (2^3 * 2^6)

Sets = 2^11 = 2048

Cache Access

- 64-bit address, and 32 KB cache with 32-byte blocks, direct-mapped.



Sets = size / (assoc * block)

2^15 / (1 * 2^5)

Sets = 1024

What Happens on a Write?

- *Write through*: The information is written to both the block in the cache and to the block in the lower-level memory.

What Happens on a Write?

- *Write through*: The information is written to both the block in the cache and to the block in the lower-level memory.
- *Write back*: The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?

What Happens on a Write?

- *Write through*: The information is written to both the block in the cache and to the block in the lower-level memory.
- *Write back*: The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each:
  - WT: read misses cannot result in writes (because of replacements)
  - WB: no writes of repeated writes

What Happens on a Write?

- *Write through*: The information is written to both the block in the cache and to the block in the lower-level memory.
- *Write back*: The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each:
  - WT: read misses cannot result in writes (because of replacements)
  - WB: no writes of repeated writes
- WT always combined with *write buffers* so that don’t wait for lower level memory

What happens on a write miss?

- *write-allocate* -- make room for the cache line in the cache, fetch rest of line from memory.

What happens on a write miss?

- *write-allocate* -- make room for the cache line in the cache, fetch rest of line from memory.
- *no-write-allocate* (write-around) -- write to lower levels of memory hierarchy, ignoring this cache.

What happens on a write miss?

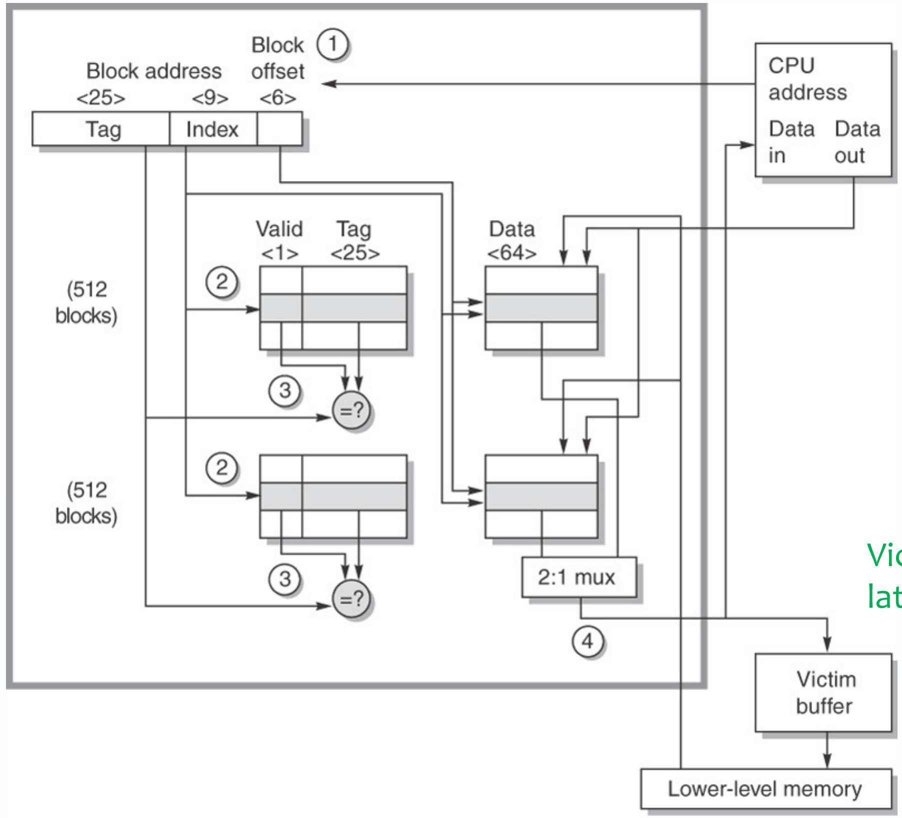
- *write-allocate* -- make room for the cache line in the cache, fetch rest of line from memory.
- *no-write-allocate* (write-around) -- write to lower levels of memory hierarchy, ignoring this cache.
- Tradeoffs?
- Which makes most sense for write-back?
- Which makes most sense for write-through?

Write Back -> write allocate
For write back , we miss in the caches, allocate the memory block, and then write ->
Then you have to keep dirty bits per individual byte if you don't bring the data from memory and change the word

Write Through -> no-write allocate
If you are doing write through, it is better to just go change the memory and use the write buffers.

21264 L1 Data Cache

- 64 KB, 64-byte blocks, 2-way set associative, ? blocks, ? sets
- write-back



© 2007 Elsevier, Inc. All rights reserved.

Separate Tag and Data Array

Victim cache later

Cache Organization: Separate Instruction and Data Caches?

Size	Instruction Cache	Data Cache	Unified Cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	1.99%
64 KB	0.15%	3.77%	1.35%
128 KB	0.02%	2.88%	0.95%

Why separate the caches?

Most are instructions. Different associativity. Not as much about miss rate as bandwidth. Need high bandwidth to each as both are accessed at the same time

Cache Performance

- CPU time = (CPU execution clock cycles + Memory stall clock cycles) x clock cycle time
- Memory stall clock cycles = Memory accesses x Miss rate x Miss penalty
    - Or, if you have more detail…
- Memory stall clock cycles = (Reads x Read miss rate x Read miss penalty + Writes x Write miss rate x Write miss penalty)
    - Or…
- Memory stall clock cycles = (Inst Cache Reads x IC miss rate x IC miss penalty + Data Cache accesses x DC miss rate x DC miss penalty)
    - Etc.

These are approximations –
assume you stall immediately when
you miss

Cache Performance

CPUtime = IC x ($CPI_{execution}$ + Memory stalls per instruction) x Clock cycle time

CPUtime = IC x ($CPI_{execution}$ + Mem accesses per instruction x Miss rate x Miss penalty) x Clock cycle time

(includes hit time as part of CPI)

(note, I will typicall call “memory stalls (cycles) per instruction” MCPI)

Cache Performance

- Instruction cache miss rate of 4%, data cache miss rate of 10%, Base CPI (no memory stalls) = 1.1, 20% of instructions are loads and stores, miss penalty = 12 cycles,

What is the CPI?

A. 1.436

B. 1.82

C. 2.78

D. None of the above

Inst Cache Miss Rate = 4% = 0.04

Data Cache Miss Rate = 10% = 0.10

Base CPI = 1.1

Load/Store = 20% = 0.20

Miss Penalty = 12

$$1.1 \underbrace{(1 + 0.04 \times 12)}_{\text{Inst Cache}} +$$

$$0.20 \times (1 + 0.10 \times 12)$$

Cache Performance

CPUtime = IC x (CPI$_{execution}$ + Memory stalls per instruction) x Clock cycle time

CPUtime = IC x (CPI$_{execution}$ + Mem accesses per instruction x Miss rate x Miss penalty) x Clock cycle time

(includes hit time as part of CPI)

(Alternate view of memory performance)

Average memory-access time = Hit time + Miss rate x Miss penalty (ns or clocks)

Improving Cache Performance

Average memory-access time = Hit time + Miss rate x Miss penalty (ns or clocks)

How are we going to improve cache performance??

1. Reduce Hit Time

2. Reduce Miss Rate

3. Reduce Miss Penalty

Reducing Misses

- Classifying Misses: 3 Cs
  - *Compulsory*—The first access to a block is not in the cache, so the block must be brought into the cache. These are also called *cold start misses* or *first reference misses*.
  - *Capacity*—If C is the size of the cache (in blocks) and there have been more than C unique cache blocks accessed since this cache was last accessed.
  - *Conflict*—Any miss that is not a compulsory miss or capacity miss must be a byproduct of the cache mapping algorithm. A conflict miss occurs because too many active blocks are mapped to the same cache set.

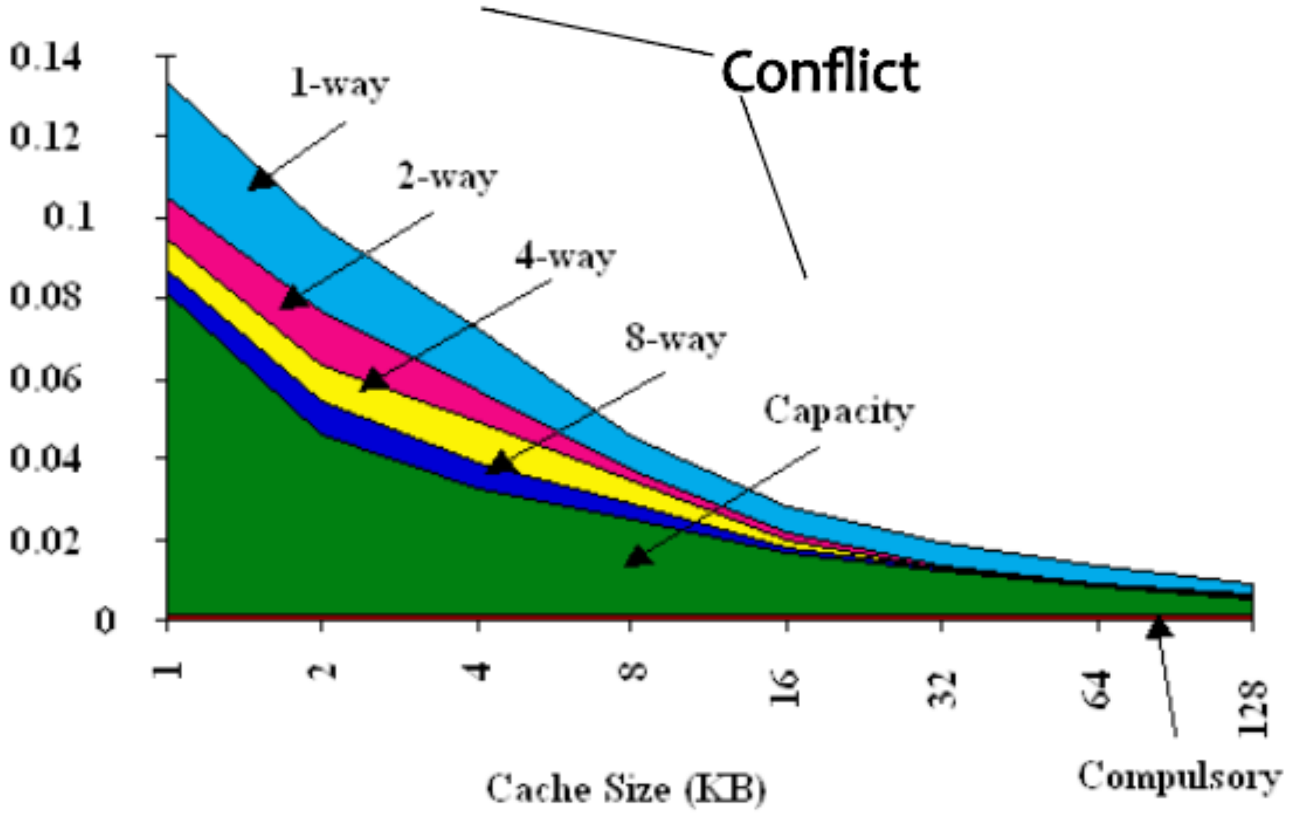
How To Measure

Misses in infinite cache

Non-compulsory misses in size X fully associative cache

Non-compulsory, non-capacity misses

3Cs Absolute Miss Rate



How To Reduce Misses?

- Compulsory Misses?
- Capacity Misses?
- Conflict Misses?
- What can the compiler do?

Caches, pt I: Key Points

- CPU-Memory gap is a major performance obstacle
- Caches take advantage of program behavior: locality
- Designer has lots of choices -> cache size, block size, associativity, replacement policy, write policy, ...
- Time of program still only reliable performance measure