ECE 485/585
Cache Basics

Portland State University
Mark G. Faust

Module:

Cache Basics

Topics

- An analogy
- Relative processor/memory performance
- Basic cache principles
  - Spatial and temporal locality
  - Direct mapped caches
    - Cache index
    - Hit/Miss handling
  - Set associative caches
  - Fully associative caches
  - Cache line replacement algorithms
- Cache performance metrics
- Vocabulary

Caching: Student Advising Analogy

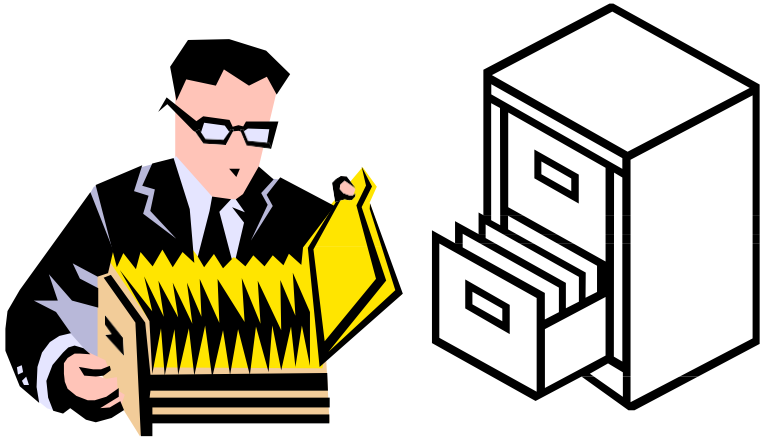
Thousands of student folders

Indexed by 9-digit student ID

Located down the hall – long walk

Space for 100 file folders at my desk

Located at my side – short access time



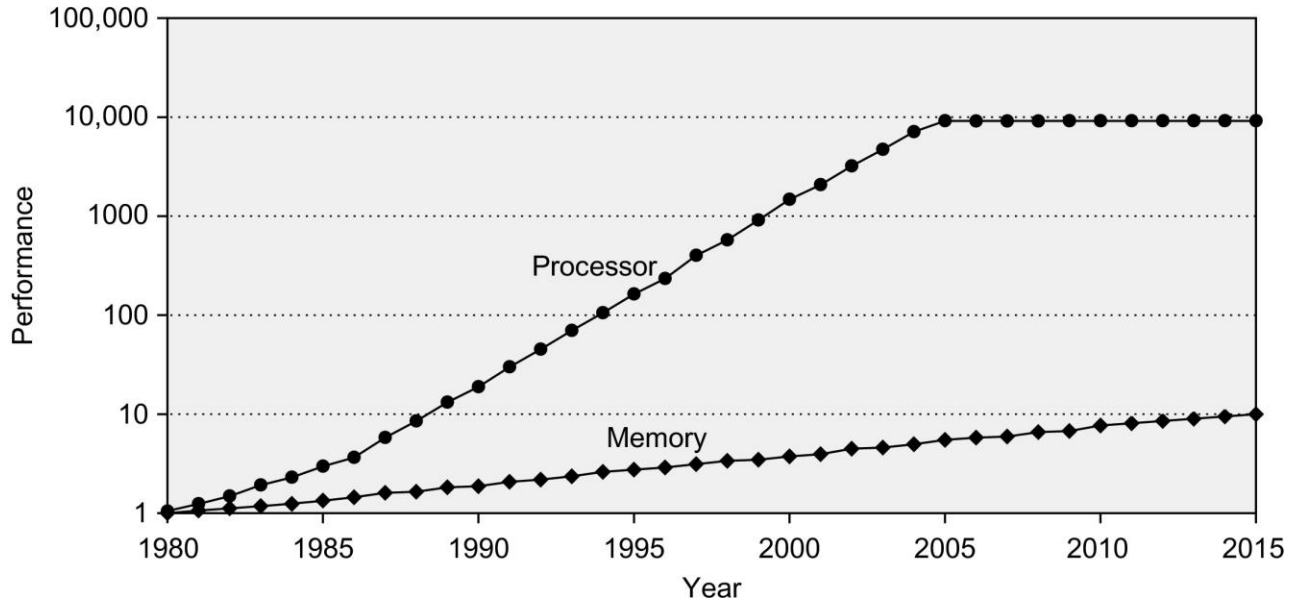
CPU/Memory Performance Gap

Year of introduction	Chip size	Row access strobe (RAS)		Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
		Slowest DRAM (ns)	Fastest DRAM (ns)		
1980	64K bit	180	150	75	250
1983	256K bit	150	120	50	220
1986	1M bit	120	100	25	190
1989	4M bit	100	80	20	165
1992	16M bit	80	60	15	120
1996	64M bit	70	50	12	110
1998	128M bit	70	50	10	100
2000	256M bit	65	45	7	90
2002	512M bit	60	40	5	80
2004	1G bit	55	35	5	70
2006	2G bit	50	30	2.5	60

Memory Technology Trends

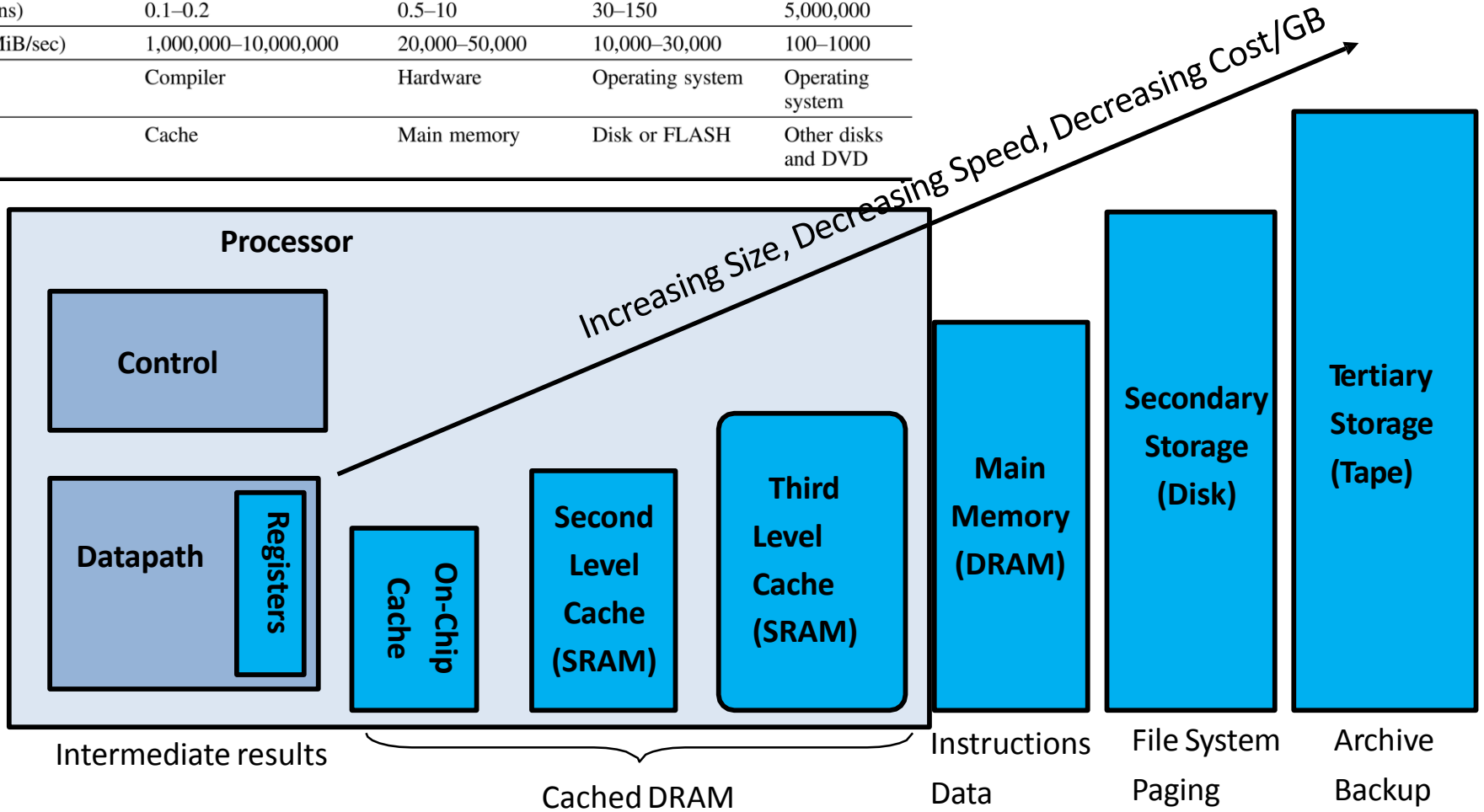
Production year	Chip size	DRAM type	Best case access time (no precharge)			Precharge needed
			RAS time (ns)	CAS time (ns)	Total (ns)	Total (ns)
2000	256M bit	DDR1	21	21	42	63
2002	512M bit	DDR1	15	15	30	45
2004	1G bit	DDR2	15	15	30	45
2006	2G bit	DDR2	10	10	20	30
2010	4G bit	DDR3	13	13	26	39
2016	8G bit	DDR4	13	13	26	39

Relative Performance Gains

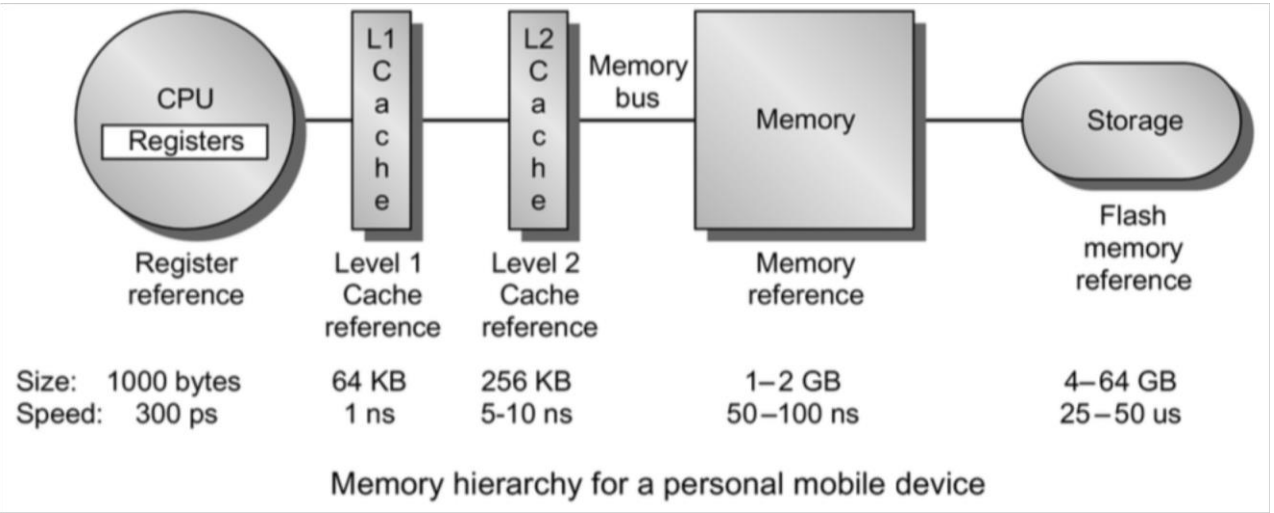


Memory Hierarchy

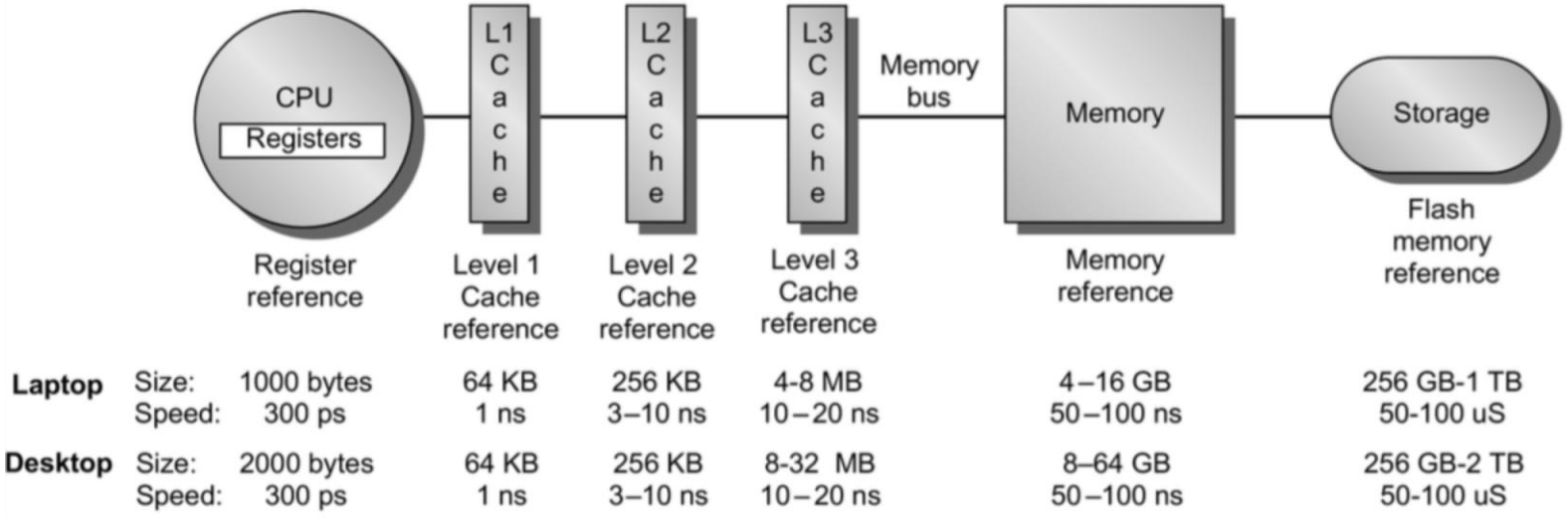
Level	1	2	3	4
Name	Registers	Cache	Main memory	Disk storage
Typical size	<4 KiB	32 KiB to 8 MiB	<1 TB	>1 TB
Implementation technology	Custom memory with multiple ports, CMOS	On-chip CMOS SRAM	CMOS DRAM	Magnetic disk or FLASH
Access time (ns)	0.1–0.2	0.5–10	30–150	5,000,000
Bandwidth (MiB/sec)	1,000,000–10,000,000	20,000–50,000	10,000–30,000	100–1000
Managed by	Compiler	Hardware	Operating system	Operating system
Backed by	Cache	Main memory	Disk or FLASH	Other disks and DVD



Memory Hierarchy

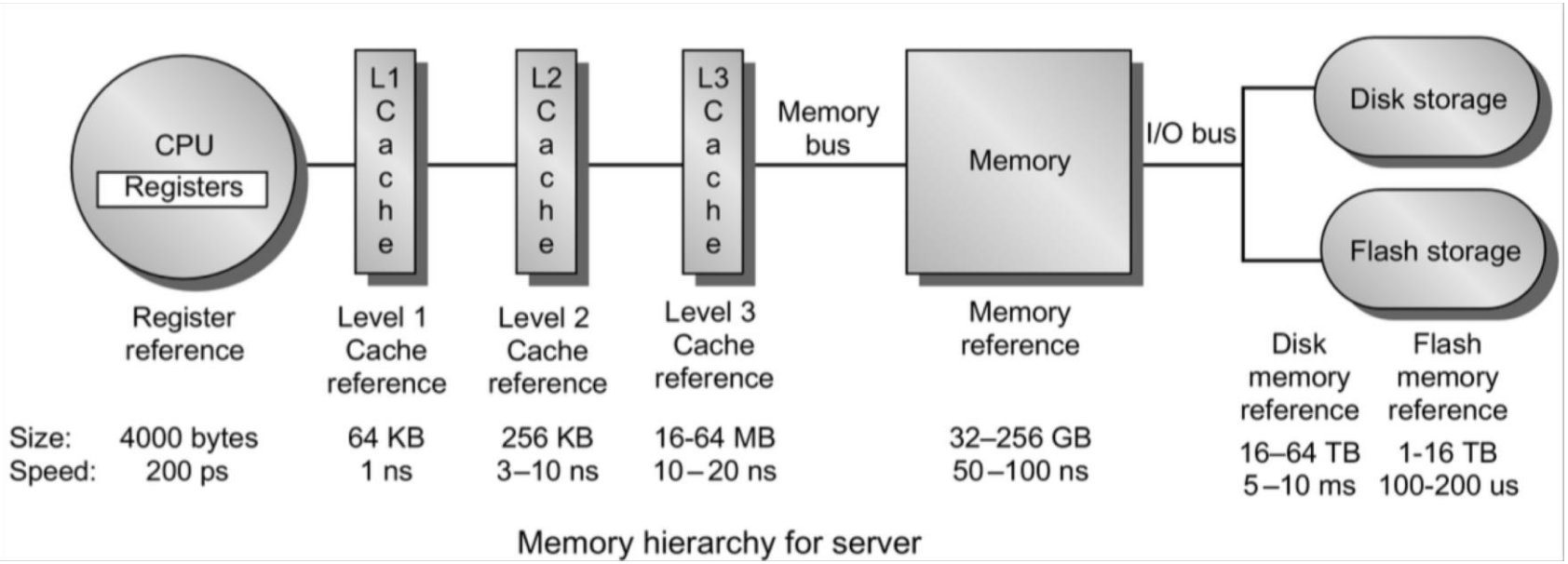


Memory hierarchy for a personal mobile device



Memory hierarchy for a laptop or a desktop

Memory Hierarchy



Memory hierarchy for server

Principle of locality

Essential for caches to provide any improvement. If data/instructions were only accessed once and addresses were scattered randomly, caches would provide no benefit.

Temporal locality: A referenced item (instruction or data) is likely to be referenced again soon.

Code: instructions in a frequently called function/subroutine, instructions in a loop

Data: commonly used variables, pointers to root of trees, heads of linked lists

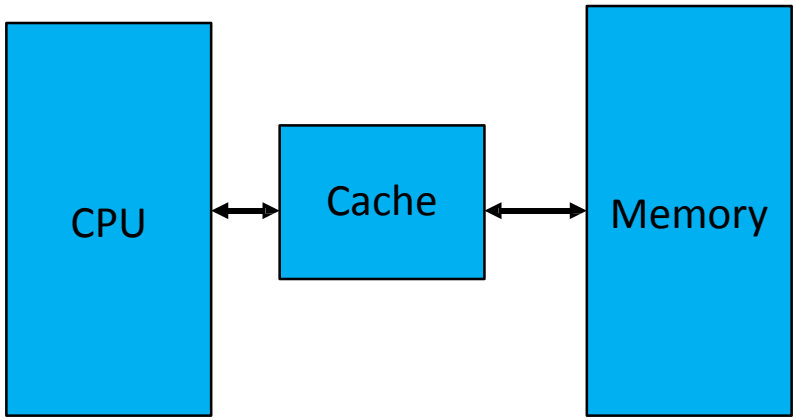
Spatial locality: Items near a referenced item are likely to be referenced soon

Code: in the absence of branches the next instruction is the next sequential instruction

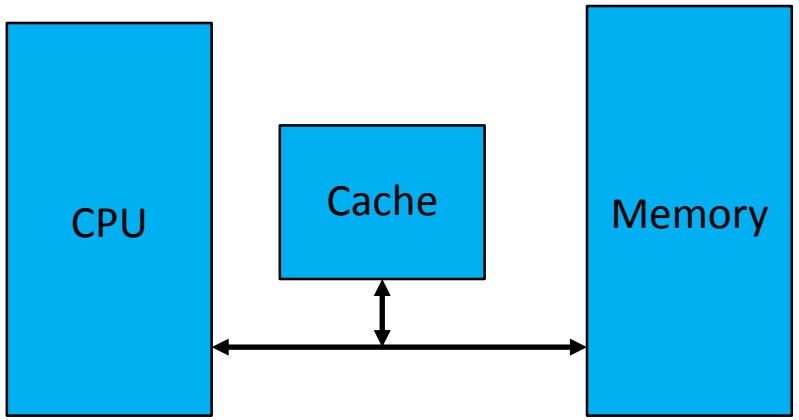
Data: frequently access all items in a vector/array/string sequentially

Look Through vs. Look Aside Cache

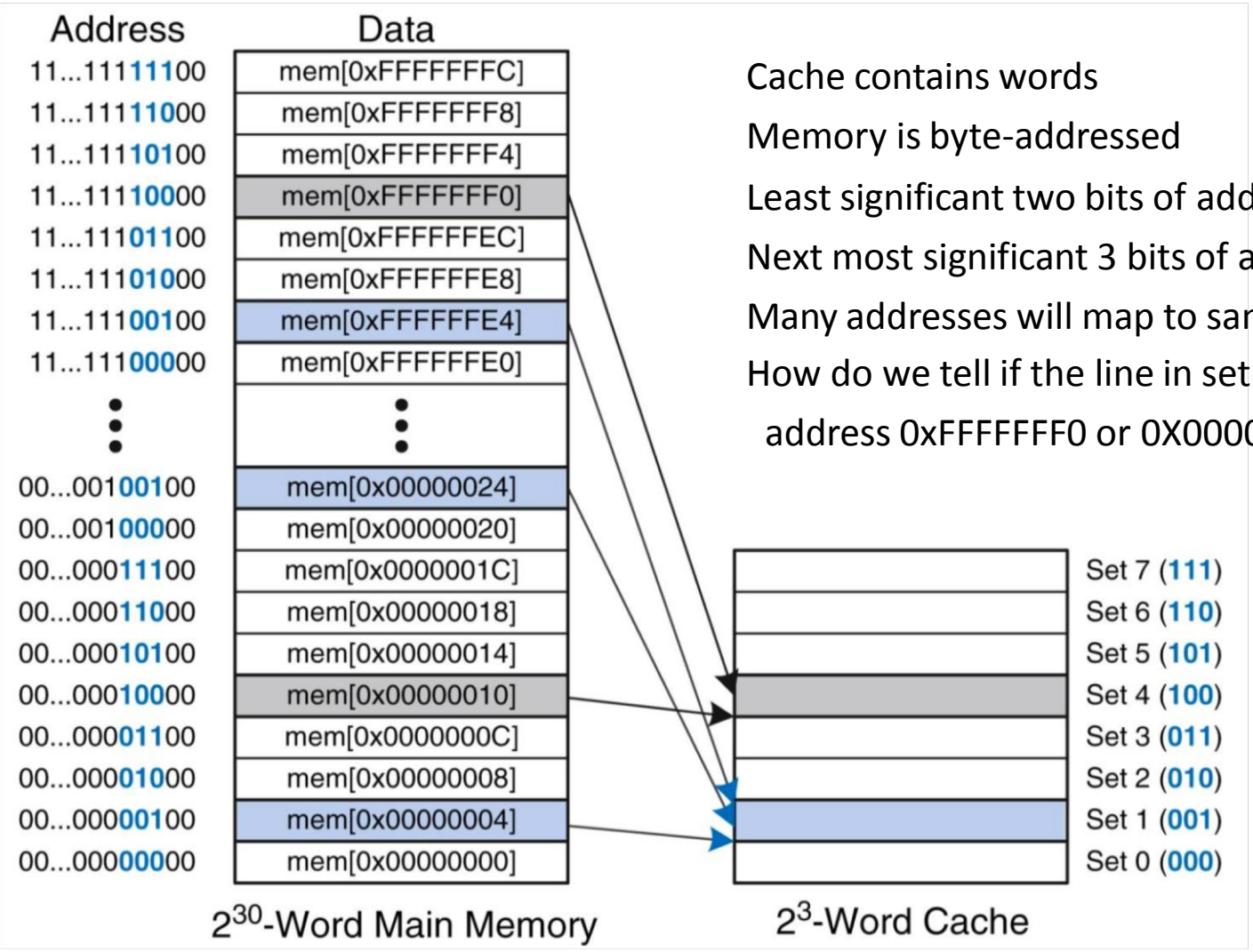
- Pros
  - Cache "filters" memory accesses
    - Fewer CPU/Memory cycles
    - Less bus contention
  - No CPU wait on writes (cache buffers)
- Cons
  - Increase in access time
  - Complexity?



- Pros
  - Faster response to cache miss
    - Memory cycle already begun
  - Simpler Design?
- Cons
  - More main memory bus traffic
  - Wasted memory bandwidth
  - Blocks other bus masters



Simple Direct Mapped Cache



Cache contains words

Memory is byte-addressed

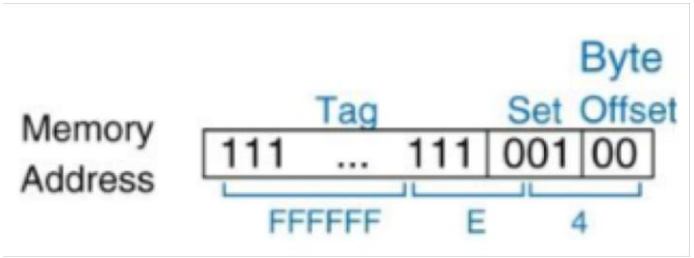
Least significant two bits of address are "byte select"

Next most significant 3 bits of address form "index"

Many addresses will map to same cache set (index)

How do we tell if the line in set 4 contains data for address 0xFFFFFFF0 or 0X00000010?

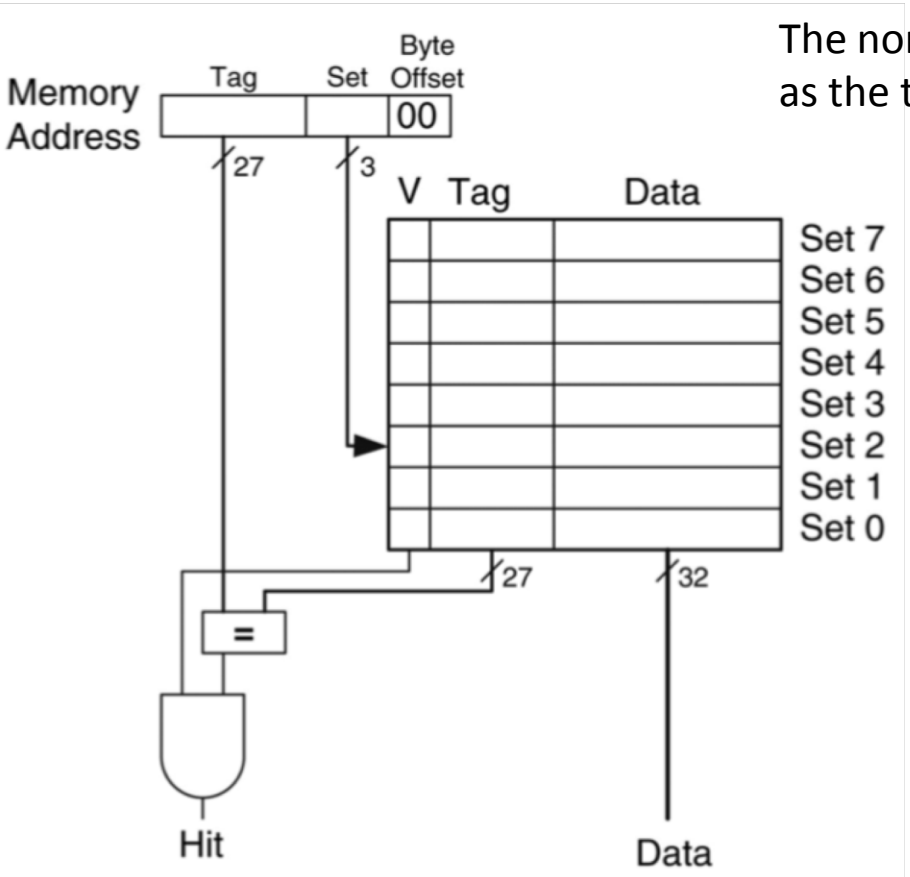
Tags

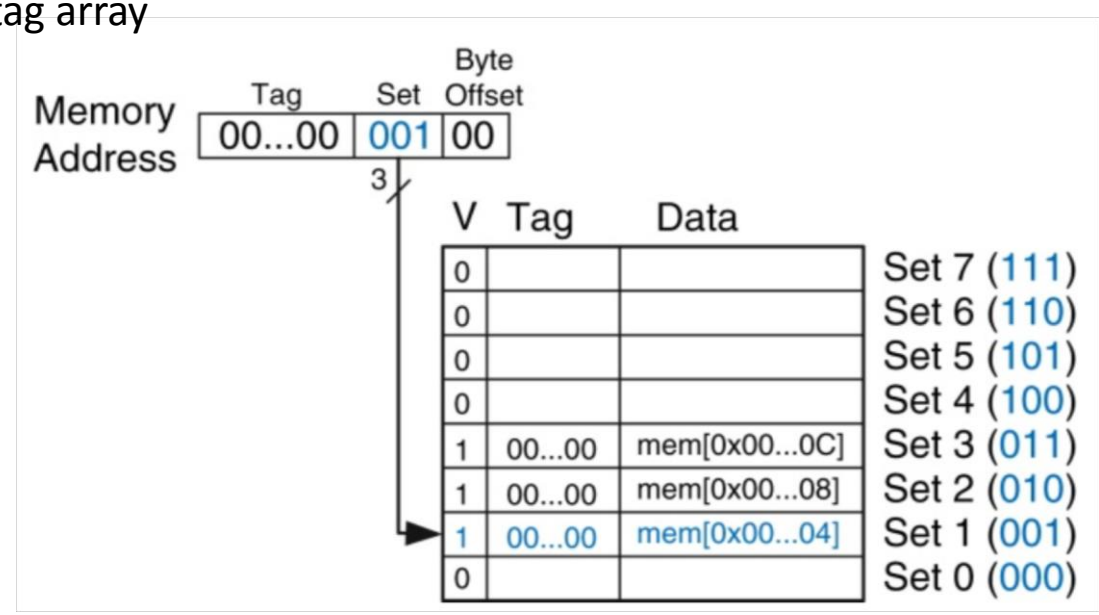


Storing the most significant bits remaining after set index and byte offset as a "tag" in the cache identifies which of many (2^{27} in this example) addresses the data is associated with.

Also need Valid bit to indicate whether the tag/data are valid (0 if unoccupied – doesn't matter what the tag bits are)

The non-Data portion of the cache (e.g. Tag, Valid) are known as the tag array





Cache line size

More efficient for cache lines to be larger

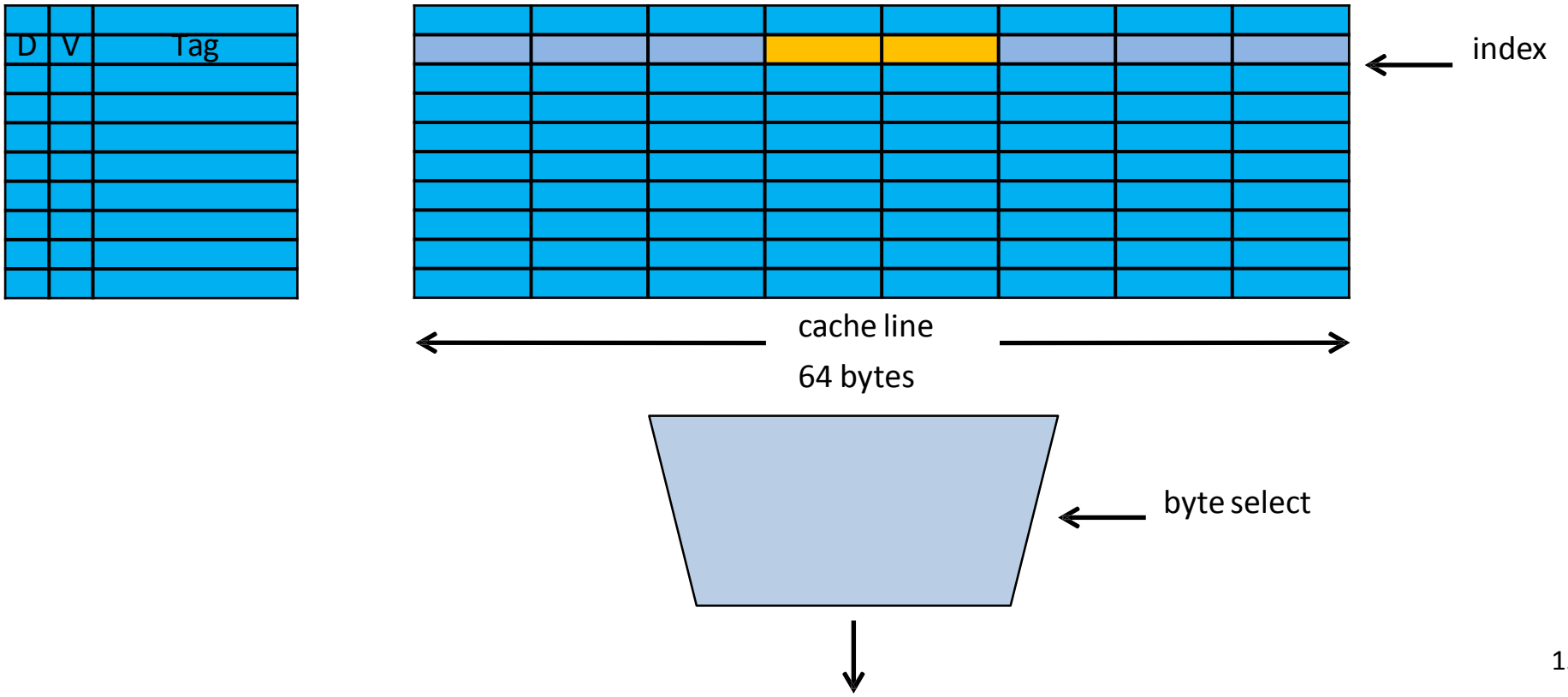
Recall DRAM burst access

DIMMs provide 64 bytes for each read

Spatial locality suggests that if CPU is requesting 4 bytes, the other 60 nearby are likely to be needed as well

Subsequent reference to any of the 64 bytes in the same cache line will result in a hit

If 64 byte cache line, byte select field becomes 6 bits, used as data selector to retrieve subset of cache line



Cache Reads

① Partition address into byte offset, index, tag fields

② if Valid[index] == 1

- ① If Tag[index] == tag ← cache hit
  - ① Deliver data to CPU
- ② If Tag[index] != tag ← cache miss (conflict)
  - ① stall CPU
  - ② Evict line
  - ③ Initiate memory request to read cache line
  - ④ Write Tag[index] ← tag
  - ⑤ deliver data to CPU

③ If Valid[index] == 0 ← cache miss (unoccupied)

- ① Stall CPU
- ② Initiate memory request to read cache line
- ③ Set Valid[index] ← 1
- ④ Write Tag[index] ← tag
- ⑤ Deliver data to CPU

Cache Writes

- Policy decisions for write hits
  - Write through
    - Write data to cache and to memory
    - Use a write buffer so CPU doesn't stall awaiting completion of DRAM write
  - Write back
    - Replace data in cache only
    - Write back to DRAM only when necessary
      - Cache "flush"
      - If the line needs to be evicted
    - Cache may have only correct copy of the data
    - Need to keep track of cache lines that have been written so that we know to write the data back to DRAM if/when they are evicted
    - Addition of "dirty" bit to tag array

Cache Writes

- Policy decisions for write misses
  - Write no allocate (or Write around)
    - Don't place the data in the cache
    - Rationale: successive writes without an intervening read unlikely
      - Saves cache line fill (DRAM read)
      - Avoid possibility of evicting a line likely to be needed later
  - Write allocate
    - Place the data in the cache
    - Requires a DRAM read! Why?

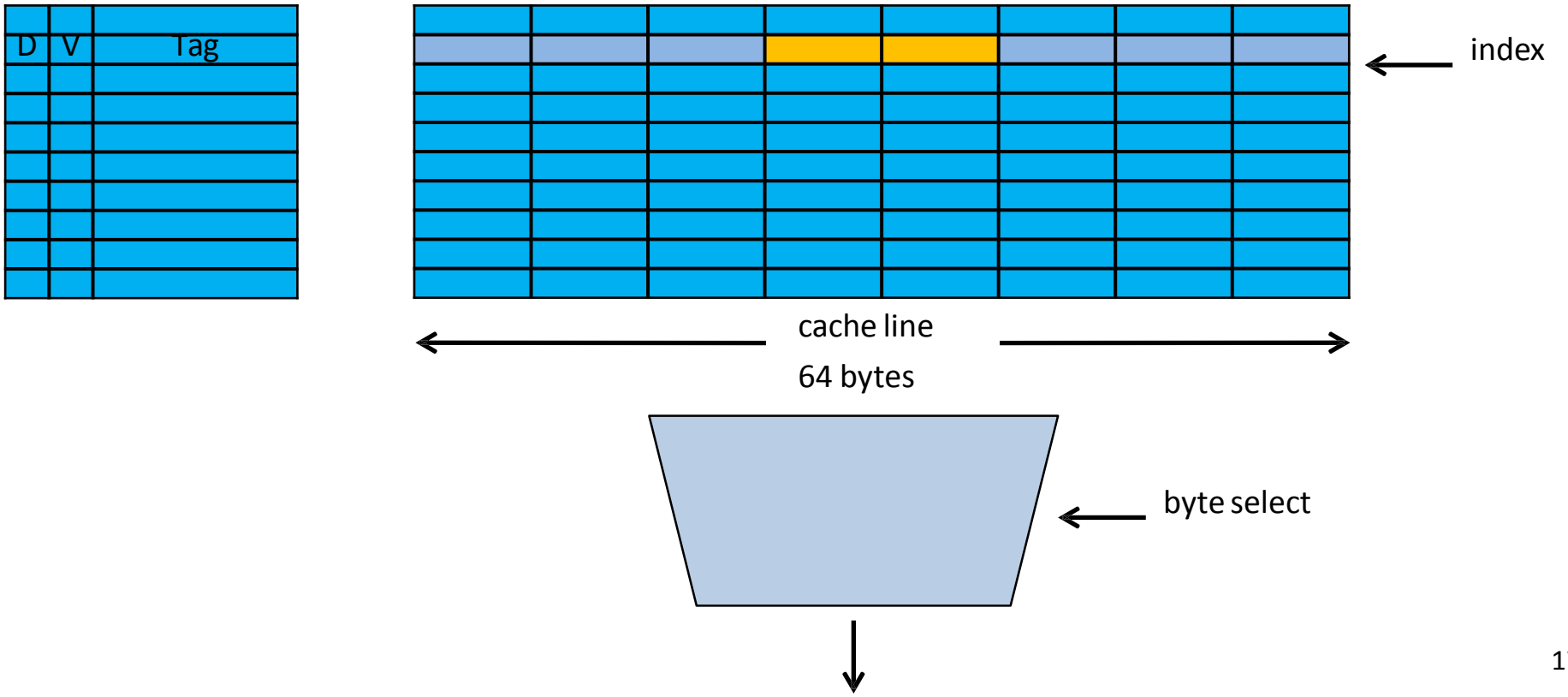
Write allocate

Can't just write the data to the cache (only a subset of the line)

What happens if subsequent read to the line

- Would be a hit (valid, tag)
- But line (except for the bytes previously written) wouldn't have valid data!

Have to read the entire cache line, then overwrite the bytes to be written



Cache Writes: write allocate, write back

① Partition address into byte offset, index, tag fields

② if Valid[index] == 1 ← cache hit

  ① If Tag[index] == tag

    ① Write bytes to cache line

  ② If Tag[index] != tag ← cache miss (conflict)

    ① stall CPU

    ② Evict line

      ① If Dirty[index] write back line to DRAM

    ③ Initiate memory request to read cache line

    ④ Write Tag[index] ← tag

    ⑤ Write bytes to cache line

③ If Valid[index] == 0 ← cache miss (unoccupied)

  ① Stall CPU

  ② Initiate memory request to read cache line

  ③ Set Valid[index] ← 1

  ④ Write Tag[index] ← tag

  ⑤ Write bytes to cache line

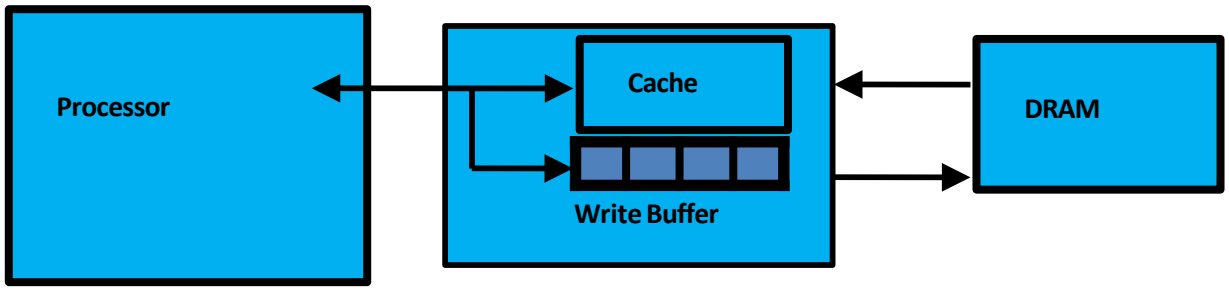
④ Dirty[index] ← 1

Evictions in WB Caches

- Can lead to interesting DRAM behavior depending upon policies
  - A CPU "read" request can cause a DRAM "write" followed by a DRAM "read"
    - Eviction of a dirty line
    - Read new line
  - A CPU "write" request can cause a DRAM "write" followed by a DRAM "read"
    - Eviction of a dirty line
    - Read new line (write allocate) into which data will be written in cache

Write Buffers

- Needed to avoid stalling processor while DRAM writes complete
  - Write through caches
  - Write no allocate
  - Write back: evictions
- Just a small depth FIFO
- Sometimes called a Posted Write Buffer (PWB)



What If…

- Cache miss (conflict)
- Evict the cache line A_1
- Read in newly requested line A_2, returning data to CPU
- Later CPU read to address in line A_1
- Cache miss (conflict)
- Evict the cache line A_2
- Later request to address in line A_1

- Problem known as "thrashing"
- Solution: N-way set associative cache
  - Allow a set to have multiple "ways"
  - Cache lines for addresses that map to same index reside in set at different ways
  - Direct mapped caches have associativity of 1 (only one place in a set for the line)

Associativity

Assuming same number of cache lines

Number of sets halved (number of index bits decreases by 1) as associativity doubles

Fully associative – no index bits!

One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

Example

Four 1-byte lines

Three cache configurations: direct mapped, 2-way set associative, fully associative

Addresses accessed in sequence: 0, 8, 0, 6, 8

No byte select bits

Direct mapped: 2 bits for index, 2 bits for tag

0 = 0000 → tag 0, set 0

8 = 1000 → tag 2, set 0

0 = 0000 → tag 0, set 0

6 = 0110 → tag 1, set 2

8 = 1000 → tag 2, set 0

Mem[x] indicates contents of memory location x

blue indicates a line just placed in the cache

red indicates a line just placed in the cache which evicted an existing line

black indicate a line already in the cache

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Example

Four 1-byte lines

Three cache configurations: direct mapped, 2-way set associative, fully associative

Addresses accessed in sequence: 0, 8, 0, 6, 8

No byte select bits

2-way set associative: 4 lines: 2 ways and 2 sets, 1 bit for index, 3 bits for tag

0 = 0000 → tag 0, set 0

8 = 1000 → tag 4, set 0

0 = 0000 → tag 0, set 0

6 = 0110 → tag 3, set 0

8 = 1000 → tag 4, set 0

Mem[x] indicates contents of memory location x

grey indicates a line just placed in the cache

red indicates a line just placed in the cache which evicted an existing line

black indicates a line already in the cache

green indicates a cache hit

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		
			way 0	way 1		

Example

Four 1-byte lines

Three cache configurations: direct mapped, 2-way set associative, fully associative

Addresses accessed in sequence: 0, 8, 0, 6, 8

No byte select bits

fully associative: 4 lines: 4 ways and 1 sets, 0 bits for index, 4 bits for tag

0 = 0000 → tag 0, set 0

8 = 1000 → tag 8, set 0

0 = 0000 → tag 0, set 0

6 = 0110 → tag 6, set 0

8 = 1000 → tag 8, set 0

Mem[x] indicates contents of memory location x

grey indicates a line just placed in the cache

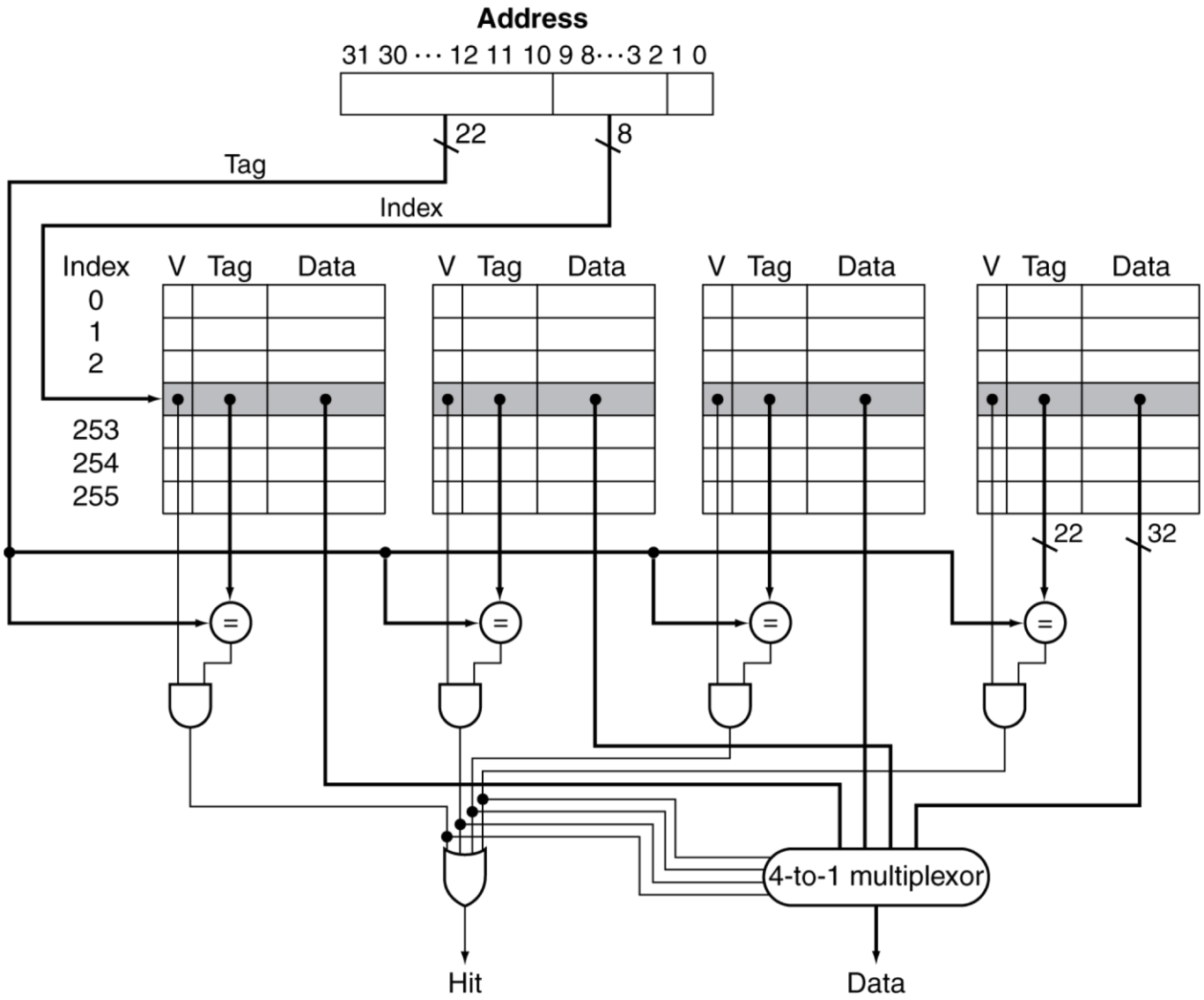
red indicates a line just placed in the cache which evicted an existing line

black indicates a line already in the cache

green indicates a cache hit

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	
			way 0	way 1	way 2	way 3

Set Associative Cache Organization



Replacement Policies

Direct mapped – no choice

Associative – first use non-valid entry, otherwise need to choose victim (way) for replacement

- Least Recently Used (LRU)
- Random
- First in, first out (FIFO)

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KiB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KiB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KiB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Need additional bits in tag array

How many bits needed to implement LRU for n-way set associative cache? per set? per line?

How many states are there?

Not enough to know current LRU way, need to maintain order from MRU to LRU and revise it

Example (4-way associative): Let 3<1<0<2 mean way 3 MRU, way 2 LRU. If cache miss and evict now, line in way 2 is victim. But what if there's cache hit to way 2 and then an eviction is necessary due to a subsequent miss? Ordering is now 2<3<1<0, so LRU is now way 0.

How many orderings of n ways are there? n!

How many bits needed to encode n!? $\lceil \log_2(n!) \rceil$ for each set

But state machine is messy and state encoding is arbitrary

Counter-based LRU implementation

n counters for each set, each counter requires $\lceil \log_2(n) \rceil$ bits

Each line's counter keeps track of its own relative position in MRU to LRU order (0 is MRU)

Each counter updated on every cache hit

Measuring Cache Performance

Some definitions

- Hit – desired item is in the cache
- Miss – desired item is not in the cache; need to request from next level (e.g. DRAM)
- Hit Rate – percentage of all cache accesses that are hits
- Miss Rate – percentage of all cache accesses that are misses (1 – Hit Rate)
- Hit Time – take to determine item is in the cache and return data
- Miss Time/Miss Penalty – time required to get item from next level (e.g. DRAM)

Cache performance requires looking at more than just the Miss Rate

One measure is Average Memory Access Time (AMAT)

Assuming Miss Time includes both time to determine it was a miss and the time required to get the data from the next level of the memory hierarchy:

Average Memory Access time (AMAT) = (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

If Miss Penalty only includes the time required to get the data from the next level of the memory hierarchy and Hit Time is the same time needed to determine there was a miss:

Average Memory Access time (AMAT) = (Hit Rate x Hit Time) + (Miss Rate x (Hit Time + Miss Time))

= Hit Time + (Miss Rate x Miss Time)

An Example

Assume

DRAM access time is 100 CPU cycles

Cache access time is 2 cycles (same time for hit or miss)

Hit rate of 90%

Memory access time without a cache = 100 cycles

Average Memory Access time (AMAT) = (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

= .90 x 2 + .10 x (2 + 102) cycles

= 12 cycles

if we assume miss time is 2 cycles to discover miss in L1 and 100 cycles to access DRAM

Since require two cycles to determine hit/miss (and return data if hit)

Average Memory Access time (AMAT) = Hit Time + (Miss Rate x Miss Time)

= 2 + .10 x 100 cycles

= 12 cycles

If we consider miss time just the additional penalty of accessing DRAM when a miss

Vocabulary

spatial locality: the locality principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon

temporal locality: the locality principle stating that if a data location is referenced then it will tend to be referenced again soon

memory hierarchy: the use of multiple levels of memories; as the distance from the processor increases, the size and access times of the memories increase

line/block: the minimum unit of information that can be present (or not) in a cache (not necessarily the minimum unit that can be *accessed*)

index: a portion of the address used to determine where in the cache a line would be located

set: all cache lines having the same index

cache miss: when the requested item is not in the cache

cache hit: when the requested item is in the cache

miss rate: the fraction of cache accesses that result in misses

hit rate: the fraction of cache access that result in hits

miss time/miss penalty: the time required to fetch a line/block into a level of the memory hierarchy from the next lower level, including the time to access the block, transfer it from one level to the other, insert it in the level experiencing the miss, and passing it to the requestor

Vocabulary

hit time: the time required to access a line/block in a level of the memory hierarchy, including the time needed to determine whether the access is a hit or miss

conflict: when two or more memory addresses map to the same cache index

direct-mapped cache: a cache in which each memory location is mapped to exactly one location in the cache

associative cache: a cache in which a memory location can be mapped to n locations (all with the same index); the cache is said to be n-way set associative

way: one of n locations in a set (lines having the same index) where a memory location is mapped in an associative cache

fully associative cache: a cache in which a memory location can be mapped to any location in the cache.

tag: the portion of the address that is used to disambiguate an address from other memory locations that map to the same index; also: the field in the cache where this information is stored

valid bit: a bit used to indicate that a cache line contains valid data (and associated tag)

write through: a cache policy in which a line written to the cache is also written to the next level in the hierarchy

write back: a cache policy in which a line is written back to the next level in the hierarchy only when the block is replaced (evicted) or flushed

evict: to cause a line to be removed from a cache, typically due to replacement

flush: to cause one or more (usually all) lines to be evicted from a cache

write allocate: a cache policy in which a line is read into the cache on a cache write miss

write no allocate: a cache policy in which a write miss does not cause the line to be read into the cache

Vocabulary

split cache: a scheme in which two separate, independent caches are used for instructions (i-cache) and data (d-cache)

unified cache: a scheme in which instructions and data reside in the same cache

(posted) write buffer: a buffer used to manage writes to the next lower level in the memory hierarchy; allows the writing cache to avoid waiting for the write to the next level to complete before continuing work

replacement policy: the policy used by a cache to determine which line to evict to make room to bring in a new line on a cache miss when all ways at the index are occupied

victim: the line chosen to be evicted by the replacement policy

LRU: least-recently used – a replacement policy in which the least recently used line in a set is evicted when a new line must be accommodated