UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 21: Caches

Taylor Johnson

Announcements and Outline

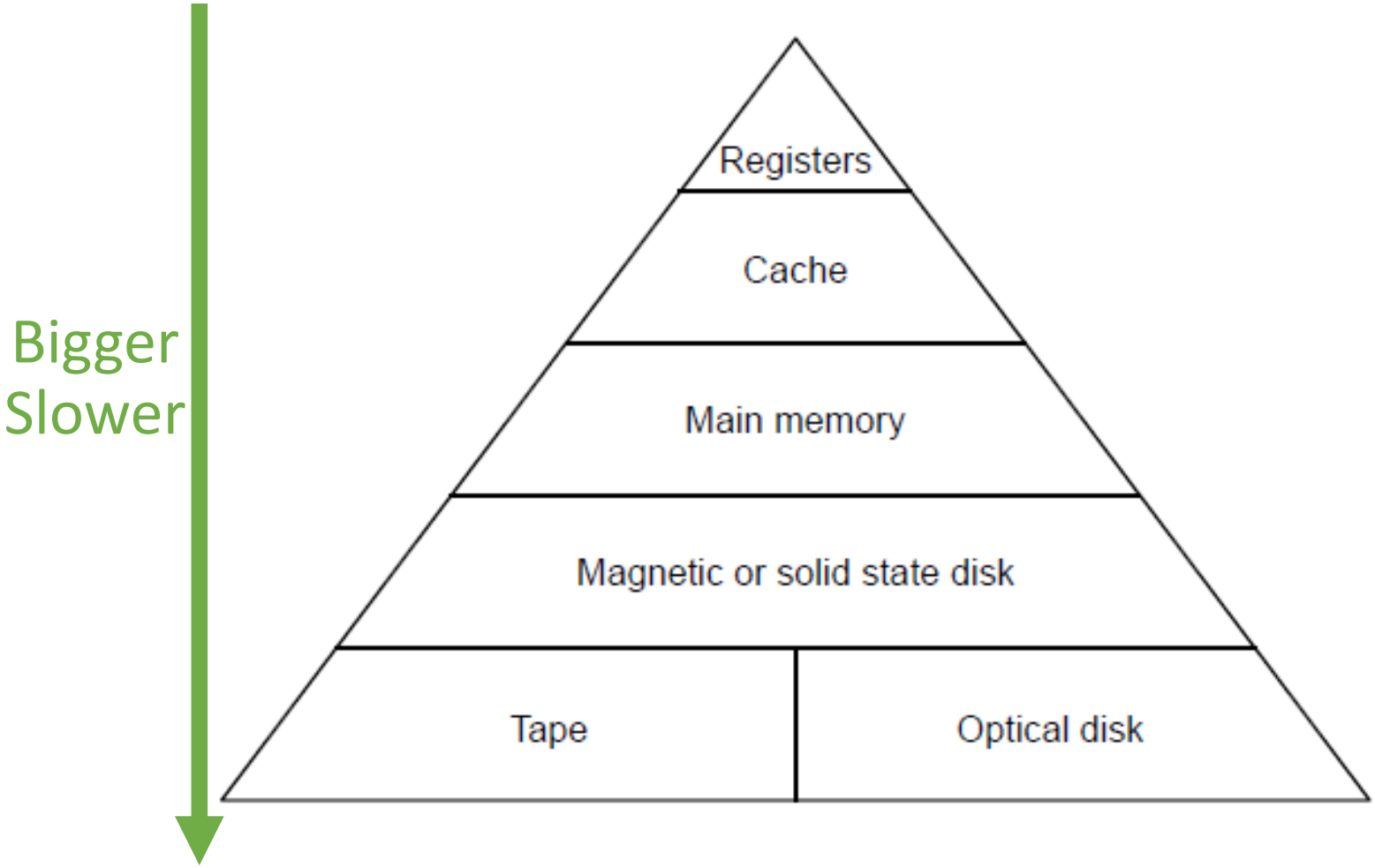
- Programming assignment 1 assigned, due 11/4 by midnight

- Review
  - Example Debugging UART Interaction with gdb
  - Memory Hierarchies
    - Registers
    - Caches
    - Main Memory
    - Storage
    - …
- Caches

Memory Speed and CPU Execution

- CPUs have always been faster than memories.
- This means that a **load** or **store** instruction, accessing the main memory, is much slower in practice than instructions accessing the ALU.
- How can we deal with this? Suppose we have a **load** instruction:
  - Let the **load** instruction go through the pipeline to fetch the data from memory to a register.
  - If any subsequent instruction tries to use that data before it has arrived, just stall (don't let that instruction proceed to the next pipeline step).

Memory Hierarchy



Levels of the Memory Hierarchy

- Registers: a few tens, accessible at full CPU speed.
  - 1 nanosecond or less.
- Cache, up to a few megabytes.
  - Accessible in a few CPU cycles.
- Main memory: 1GB-16GB for most personal PCs.
  - Access: 10 nanoseconds.
- Solid state drives: 128-512GB.
  - Access: about 100 nanoseconds.
- Magnetic disks (traditional hard drives): 1-4TB.
  - Access: 3-6 milliseconds (millions of nanoseconds).
- Optical disks, tapes: access can take seconds or more.

Memory Hierarchies

- We can have superfast expensive memory.
- We can also have slow cheap memory.
- We want lots of superfast cheap memory.
- We can get close to that goal, using a memory hierarchy.
  - At the top, a little bit of superfast expensive memory: CPU registers.
  - Each next level is somewhat slower, and somewhat cheaper.
- Because of the locality principle, the vast majority of memory accesses happen at the lower, faster levels.

Computing the Slowdown

- Suppose that:
  - 1 out of 5 instructions accesses memory.
  - Memory access is 5 CPU cycles.
- Then, on average, for each 5 instructions, we need:
  - ?? CPU cycles to execute the 4 instructions not accessing memory.
  - ?? CPU cycles to execute the 1 instruction accessing memory.
- In total, we need ?? CPU cycles per 5 instructions.
  - ??% slower than it would be if memory ran at the same speed as the CPU.

Computing the Slowdown

- Suppose that:
  - 1 out of 5 instructions accesses memory.
  - Memory access is 5 CPU cycles.
- Then, on average, for each 5 instructions, we need:
  - 4 CPU cycles to execute the 4 instructions not accessing memory.
  - 5 CPU cycles to execute the 1 instruction accessing memory.
- In total, we need 9 CPU cycles per 5 instructions.
  - 80% slower than it would be if memory ran at the same speed as the CPU.

Computing the Slowdown

- Suppose that:
  - 1 out of 5 instructions accesses memory.
  - Memory access is 50 CPU cycles.
- Then, on average, for each 5 instructions, we need:
  - ?? CPU cycles to execute the 4 instructions not accessing memory.
  - ?? CPU cycles to execute the 1 instruction accessing memory.
- In total, we need ?? CPU cycles per 5 instructions.
  - About ?? times slower than it would be if memory ran at the same speed as the CPU.

Computing the Slowdown

- Suppose that:
  - 1 out of 5 instructions accesses memory.
  - Memory access is 50 CPU cycles.
- Then, on average, for each 5 instructions, we need:
  - 4 CPU cycles to execute the 4 instructions not accessing memory.
  - 50 CPU cycles to execute the 1 instruction accessing memory.
- In total, we need 54 CPU cycles per 5 instructions.
  - About 11 times slower than it would be if memory ran at the same speed as the CPU.

load and Reordering

- This problem of slow memory access only occurs when subsequent instructions try to use the memory data that **load** is fetching.
  - Unfortunately this is very common.
- Instruction reordering can be used to try to put as many instructions between the **load** instruction and the instruction that tries to use the data fetched by **load**.
  - However, oftentimes that is not very useful. Why?

load and Reordering

- This problem of slow memory access only occurs when subsequent instructions try to use the memory data that **load** is fetching.
  - Unfortunately this is very common.
- Instruction reordering can be used to try to put as many instructions between the **load** instruction and the instruction that tries to use the data fetched by **load**.
  - However, oftentimes that is not very useful. Why?
  - The most common reason why an instruction fetches data from memory is that the next instructions need that data.

load vs. store

- **Store** instructions are not as big a problem as **load** instructions, in terms of hurting performance. Why?

load vs. store

- **Store** instructions are not as big a problem as **load** instructions, in terms of hurting performance. Why?
  - We typically store data back to memory when we do not want to use it anymore.
  - So, subsequent instructions typically do not need to refetch that data from memory.
  - A **store** instruction may need to wait until its data is ready, but that type of wait is much shorter than waiting for **load** to bring data from memory.

Remedy for Slow Memory: The Cache

- Designers of memory systems have to struggle to satisfy two conflicting goals:
  - We want lots of cheap memory.
  - We want memory to be as fast as the CPU.
- To improve performance, it is common to use a hybrid approach:
  - A large amount of slow and cheap memory.
  - A small amount of fast and expensive memory.
- This small, fast memory, is called a cache.

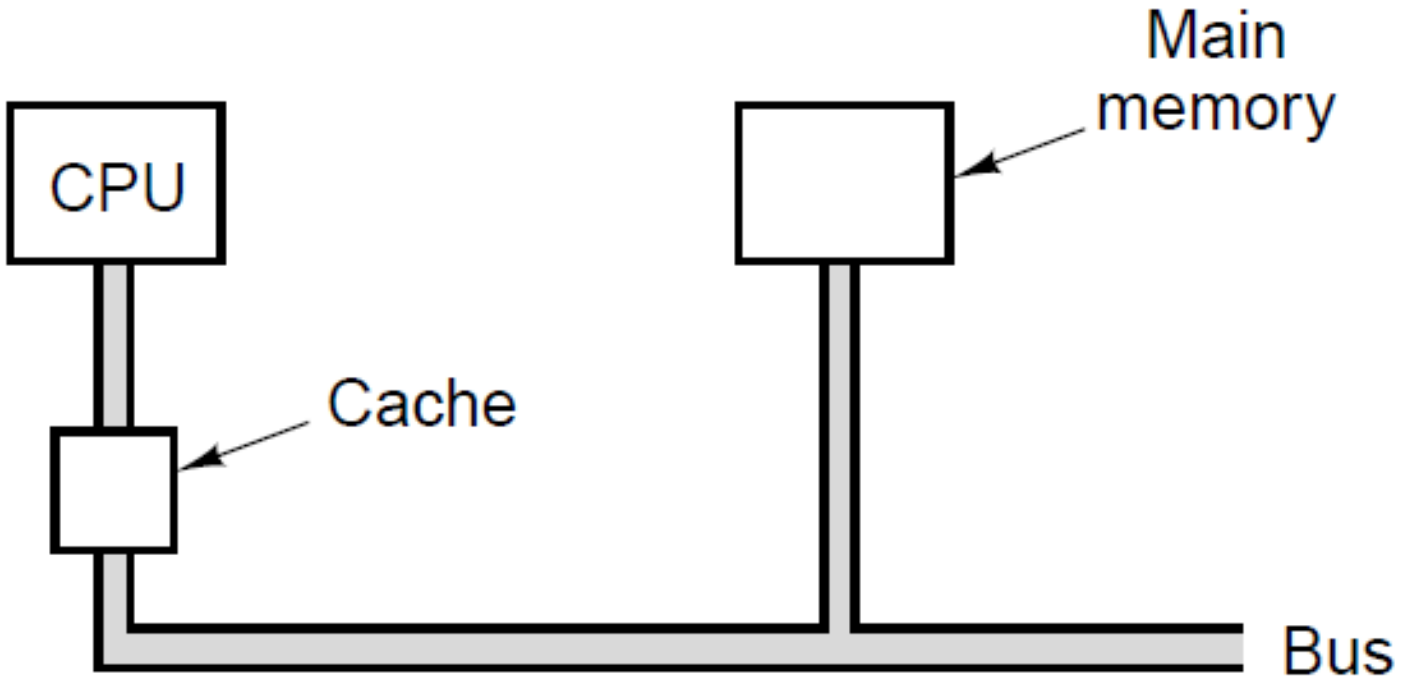
How the Cache Works

- If the CPU needs a memory word, it looks for it in the cache.
- If the word is found in the cache, proceed as normal.
- If the word is not found in the cache, get it from main memory, and store it in the cache.
  - Obviously, every time we store a word in the cache, some other word gets overwritten and is now only available in main memory.
- When will this approach improve performance, when will it hurt performance?

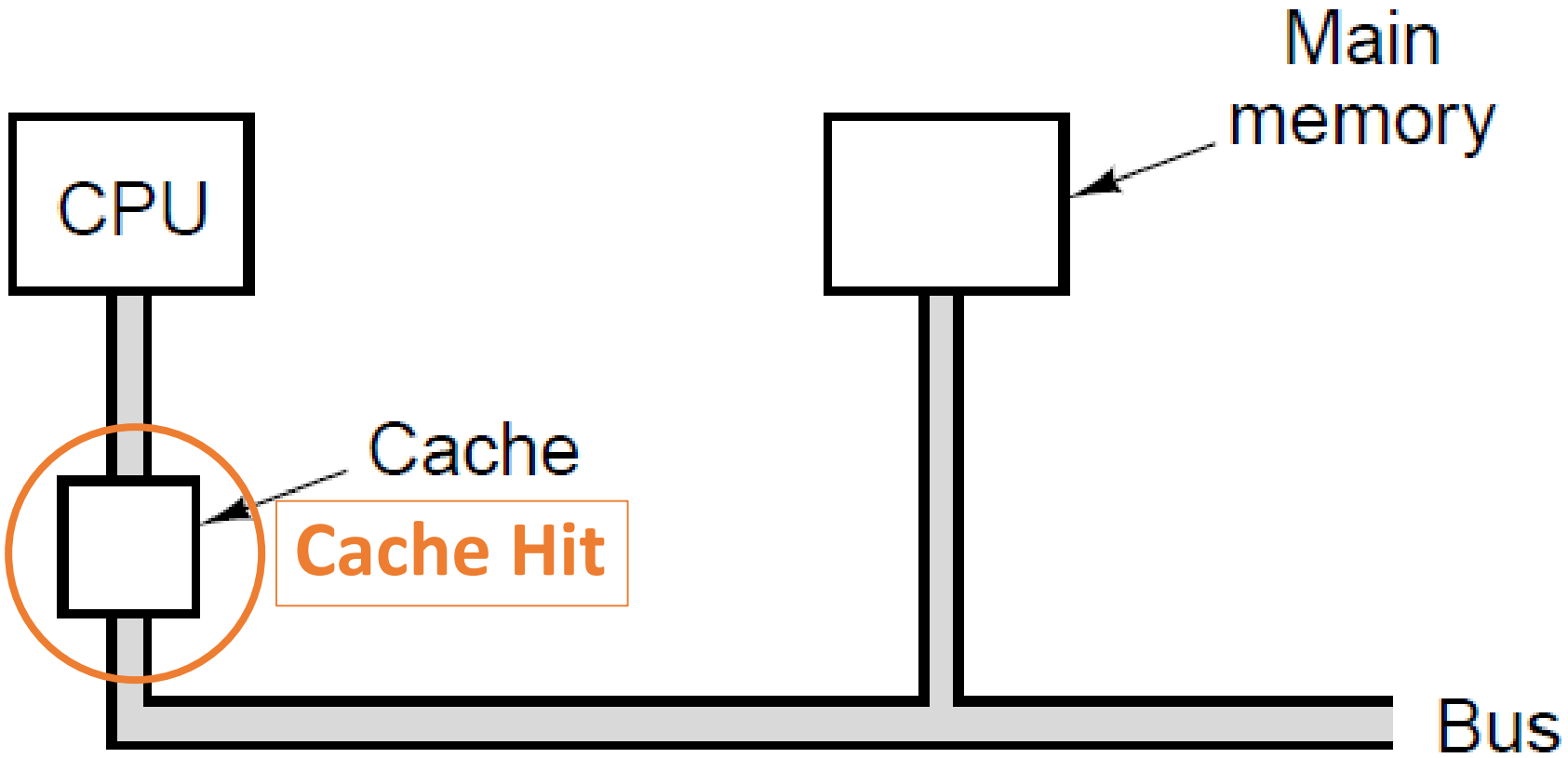
How the Cache Works

- If the CPU needs a memory word, it looks for it in the cache.
- If the word is found in the cache, proceed as normal.
- If the word is not found in the cache, get it from main memory, and store it in the cache.
  - Obviously, every time we store a word in the cache, some other word gets overwritten and is now only available in main memory.
- When will this approach improve performance, when will it hurt performance?
  - It depends on the percentage of times that the word we need is in the cache.

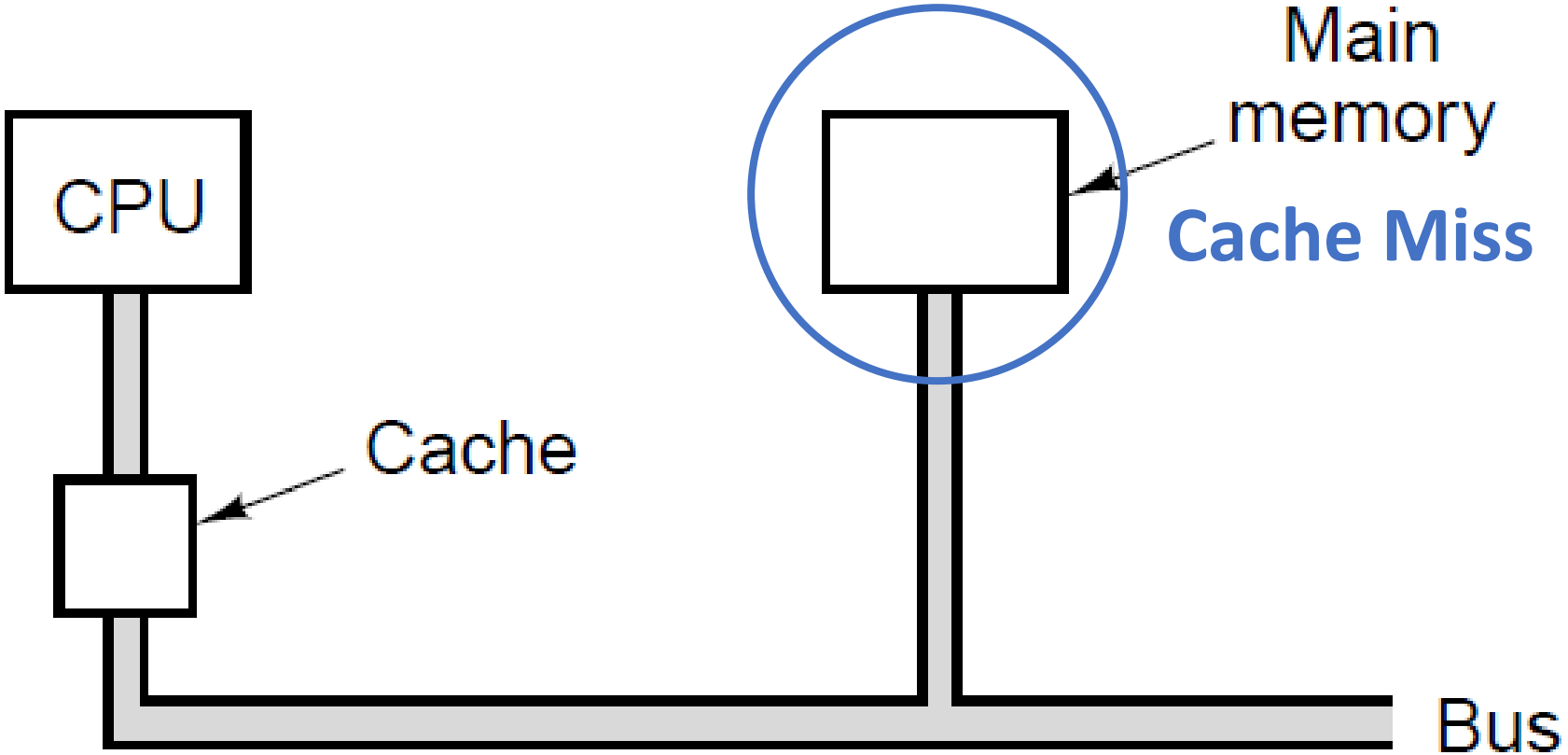
Cache Memory



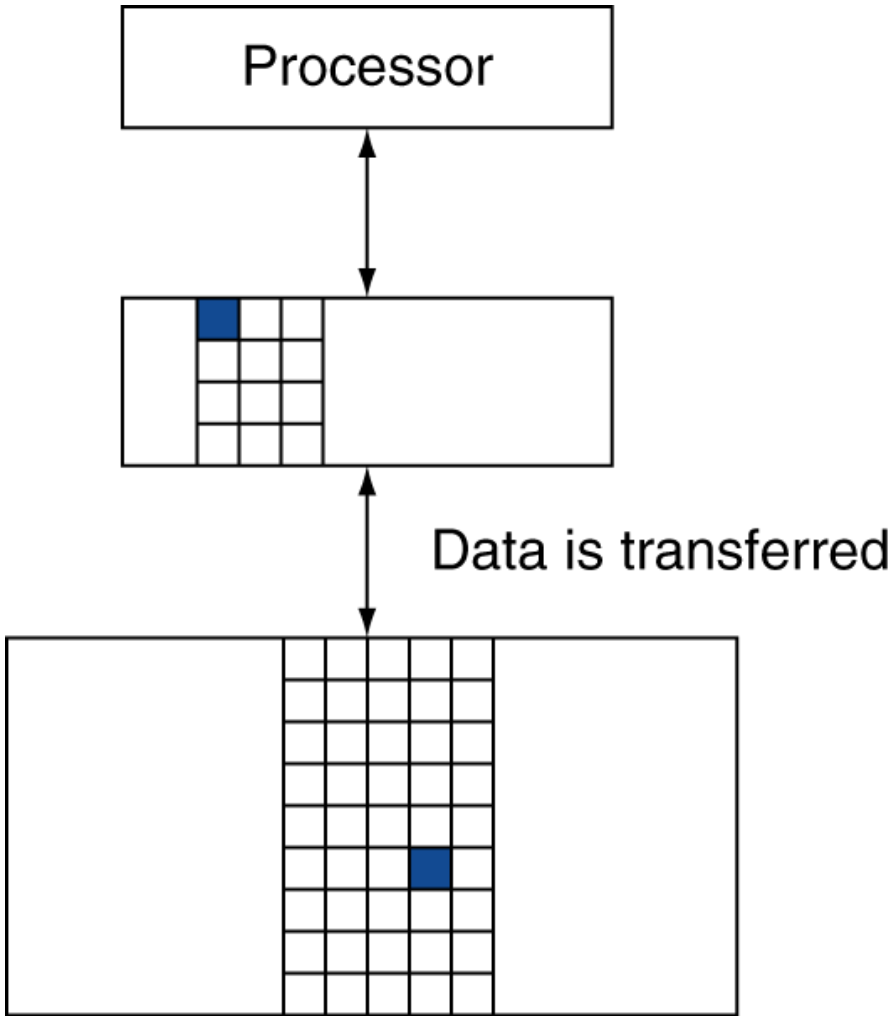
Cache Hit: find necessary data in cache



Cache Miss: have to get necessary data from main memory

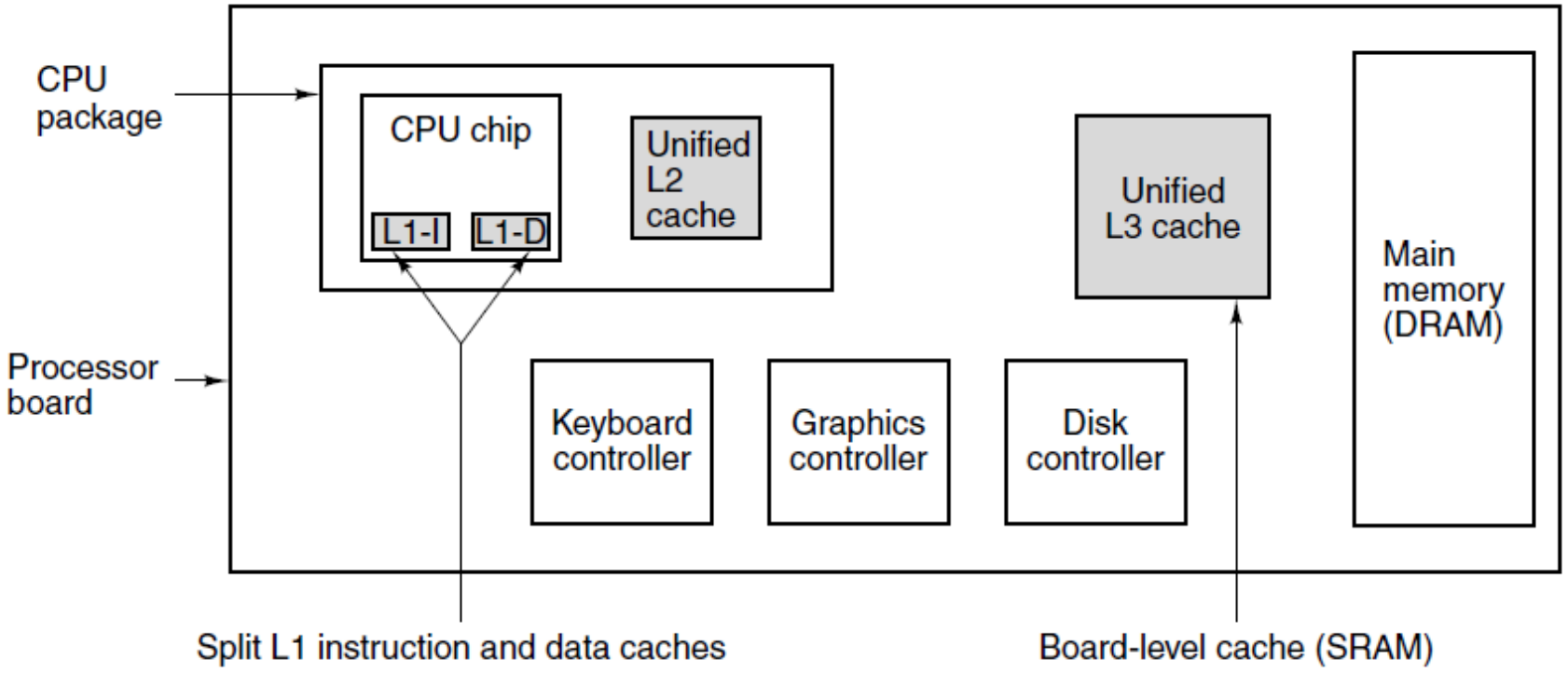


Memory Hierarchy Levels



- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses = 1 – hit ratio
  - Then accessed data supplied from upper level

More Detailed Cache Organization



Cache Terms

- Cache line: block of cells inside a cache
  - Usually store several words in a line (e.g., store 32 bytes on 32-bit word CPU)
- Cache hit: memory access finds value in cache
  - Antonym: cache miss: have to get it from main memory
- Spatial locality: likely we need data from addresses around one we're requesting (example: array operations)
- Mean access time: C + (1 – H) * M
  - C: cache access time
  - M: main memory access time (usually M >> C, e.g., M > 100 * C)
  - H: hit ratio: probability to find a value in the cache
  - miss ratio: 1 – H
- Time cost of cache miss: C + M memory access time

Cache Design Criteria

- Cache size
  - Bigger cache is more effective, but slower to access and more expensive
- Cache line size
  - Example: 16KB cache divides into
    - 1024 lines of 16 bytes
    - 2048 lines of 8 bytes
    - Etc.
- Cache organization
  - How to keep track of which memory words are in cache?
  - Keep both data and instructions in same cache?

Best and Worst Case

- If almost all words the CPU needs are in the cache, then the average time of accessing memory is close to the time it takes to access the cache.
- If almost all words the CPU needs are NOT in the cache, then the average time of accessing memory is even worse than the time it takes to access main memory
  - Because, before we even access main memory, we need to check the cache.

Quantifying Memory Access Speed

- Let:
  - mean_access_time be the average time it takes for the CPU to access a memory word.
  - C be the average time it takes for the CPU to access a memory word **if that word is currently in the cache**.
  - M be the average time it takes for the CPU to access a word in main memory (i.e., **not in the cache**).
  - H be the **hit ratio**:the fraction of times that the memory word the CPU needs is in the cache.
- mean_access_time = C + (1 – H) M
- If H is close to 1:
- If H is close to 0:

Quantifying Memory Access Speed

- Let:
  - mean_access_time be the average time it takes for the CPU to access a memory word.
  - C be the average time it takes for the CPU to access a memory word **if that word is currently in the cache**.
  - M be the average time it takes for the CPU to access a word in main memory (i.e., **not in the cache**).
  - H be the **hit ratio**:the fraction of times that the memory word the CPU needs is in the cache.
- mean_access_time = C + (1 – H) M
- If H is close to 1: mean_access_time ≅ C.
- If H is close to 0: mean_access_time ≅ C + M.

Quantifying Memory Access Speed

- mean_access_time = C + (1 – H) M
- If H is close to 1: mean_access_time ≅ C.
  - If the hit ratio is close to 1, then almost all memory accesses are handled by the cache, so the time it takes to access main memory does not affect the average much.
- If H is close to 0: mean_access_time ≅ C + M.
  - If the hit ratio is close to 0, then almost all memory accesses are handled by the main memory. In that case, the CPU:
    - First tries to access the word in the cache, which takes time C.
    - The word is not found in the cache, so the CPU then accesses the word from memory, which takes time M.

The Locality Principle

- In typical programs, memory accesses are not random.
- If we access memory address A, it is likely that the next memory address to be accessed will be close to A.
- More generally, the memory references made in any short time interval tend to use only a small fraction of the total memory.
  - This observation is called the **locality principle**.

Principle of Locality

- Programs access a small proportion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU

Using the Locality Principle

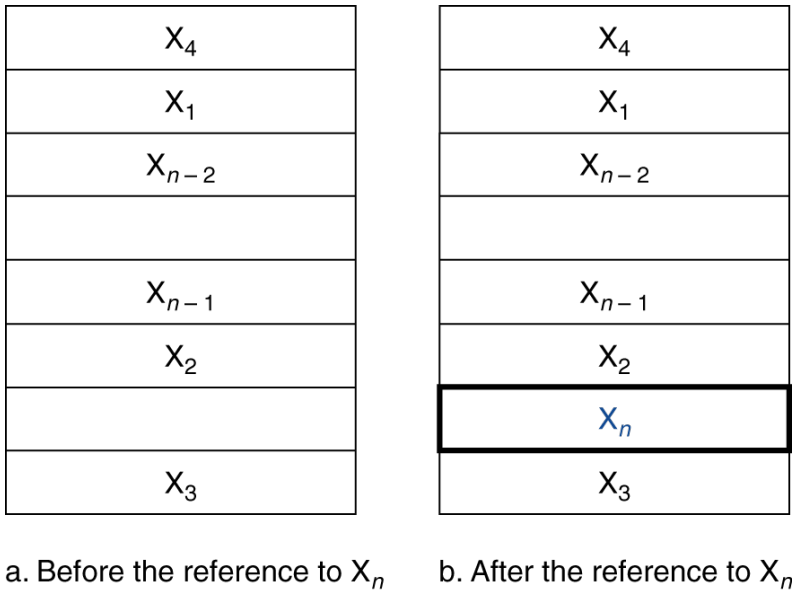
- How do we use the locality principle?
- If we need a word and that word is not in the cache:
  - Bring to the cache not only that word, but also several of its neighbors, since they are likely to be accessed next.
- How do we determine which neighbors to load?
  - We divide memories and caches into fixed-sized blocks called **cache lines**.
  - When a cache miss occurs, the entire cache line for that word is loaded into the cache.

Cache Design Optimization

- In designing a cache, several parameters must be determined, oftentimes experimentally.
  - Size of cache: bigger caches lead to better performance, but are more expensive.
  - Size of cache line:
    - 1 word is too small.
    - Setting the cache line to be equal to the cache size is probably too large.
    - Not clear where the optimal value in between is, but simulations can help determine that.

Cache Memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
- Given accesses X_1, …, X_{n-1}, X_n

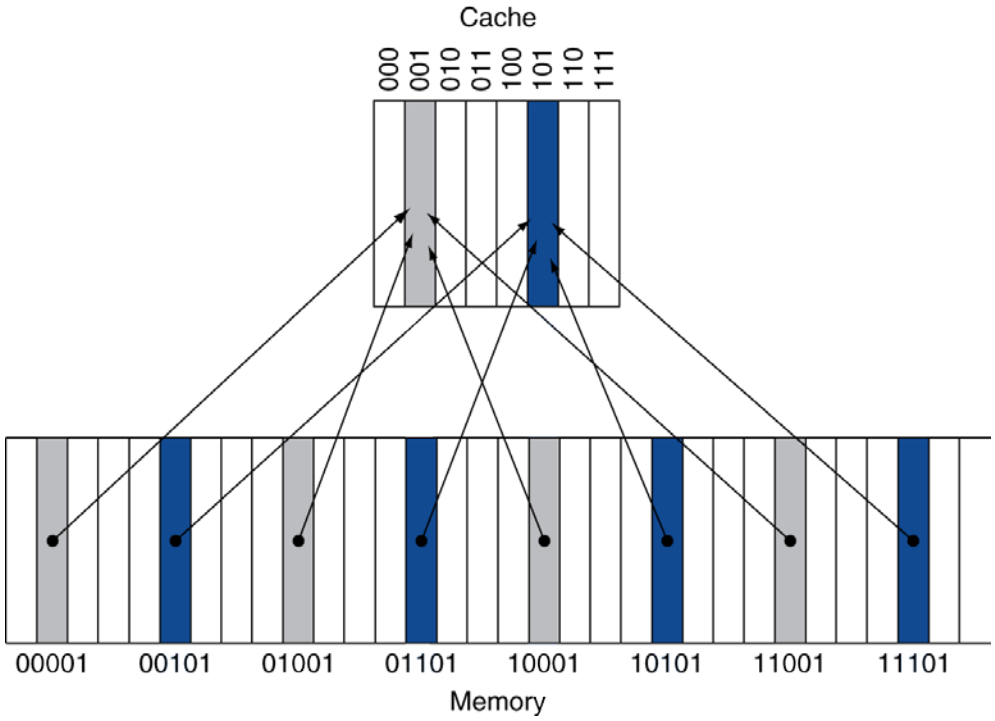


a. Before the reference to X_n b. After the reference to X_n

- How do we know if the data is present?
- Where do we look?

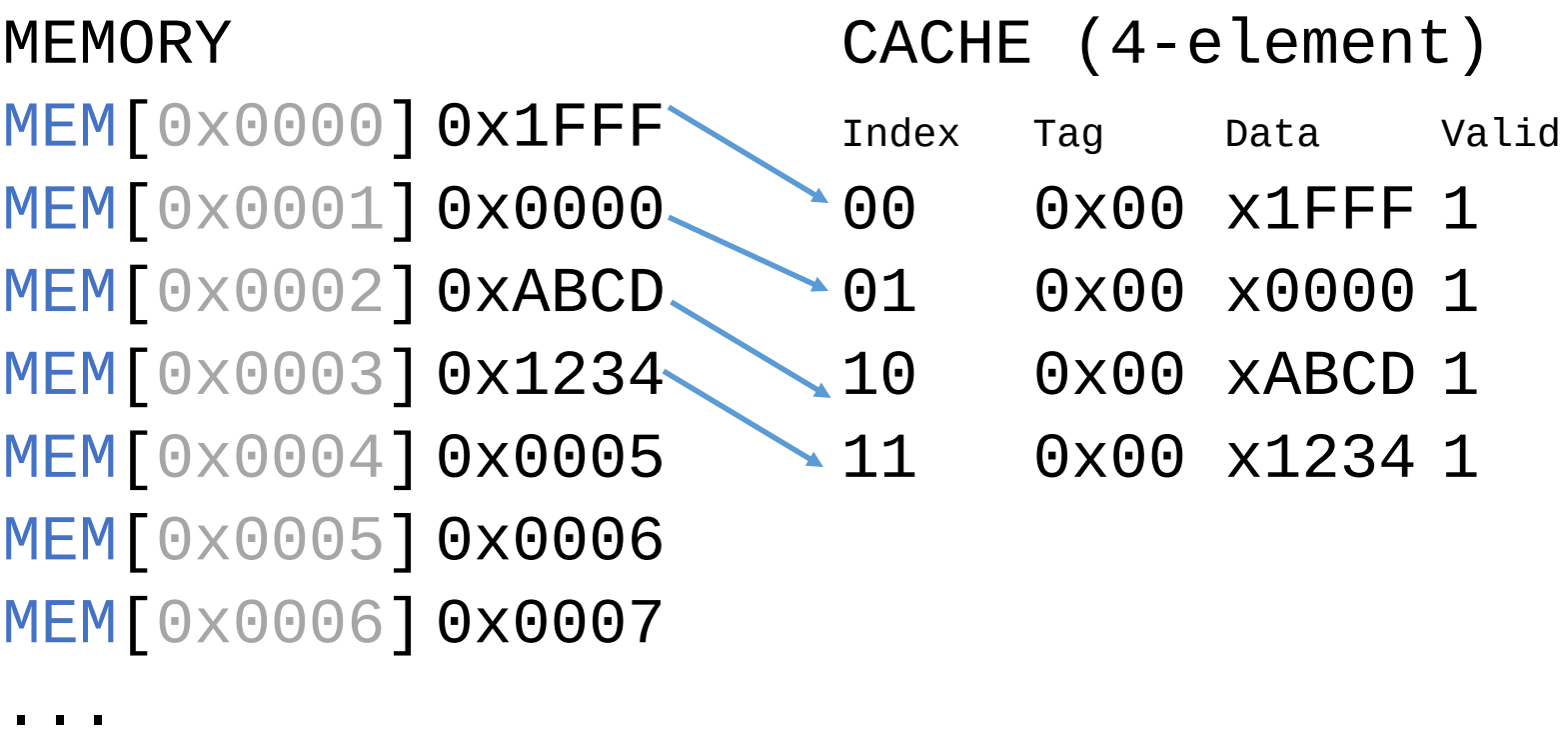
Direct-Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)

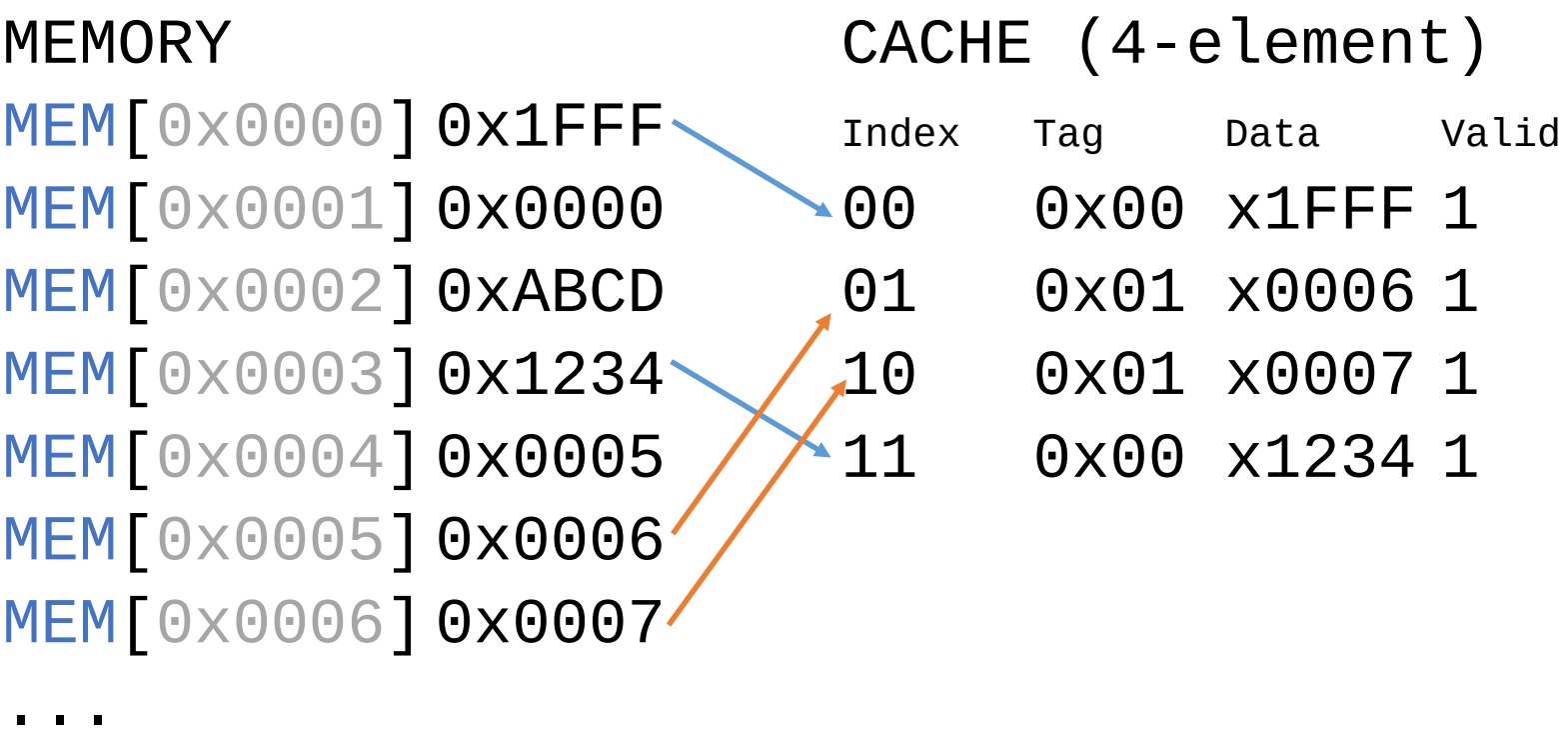


- #Blocks is a power of 2
- Use low-order address bits

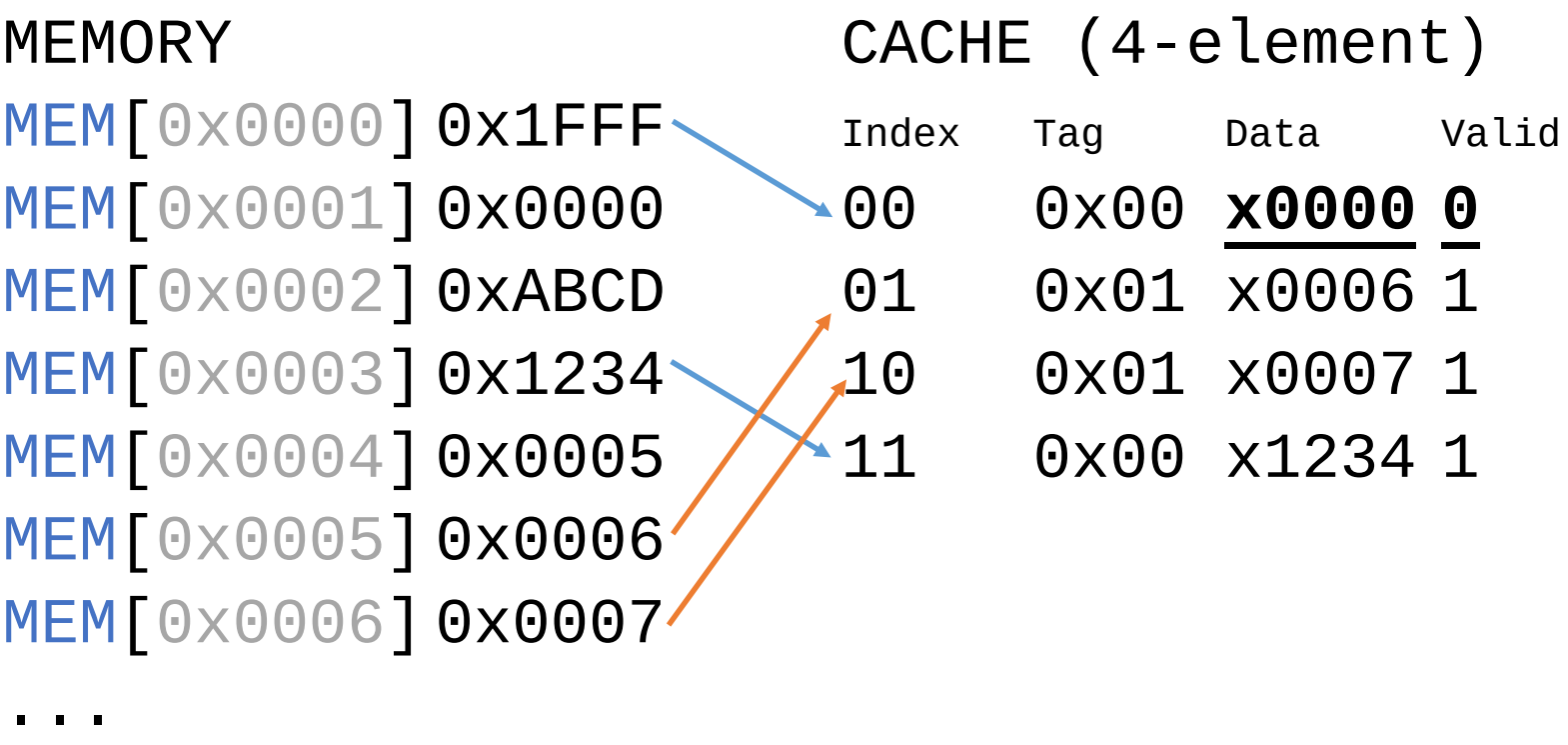
Direct-Mapped Caches



Direct-Mapped Caches



Direct-Mapped Caches



Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - **Index** = bottom bits of address
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the ***tag***
  - ***Memory Address = concatenating tag and index***
- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Direct-Mapped Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Direct-Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10 110]**
111	N		

Direct-Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

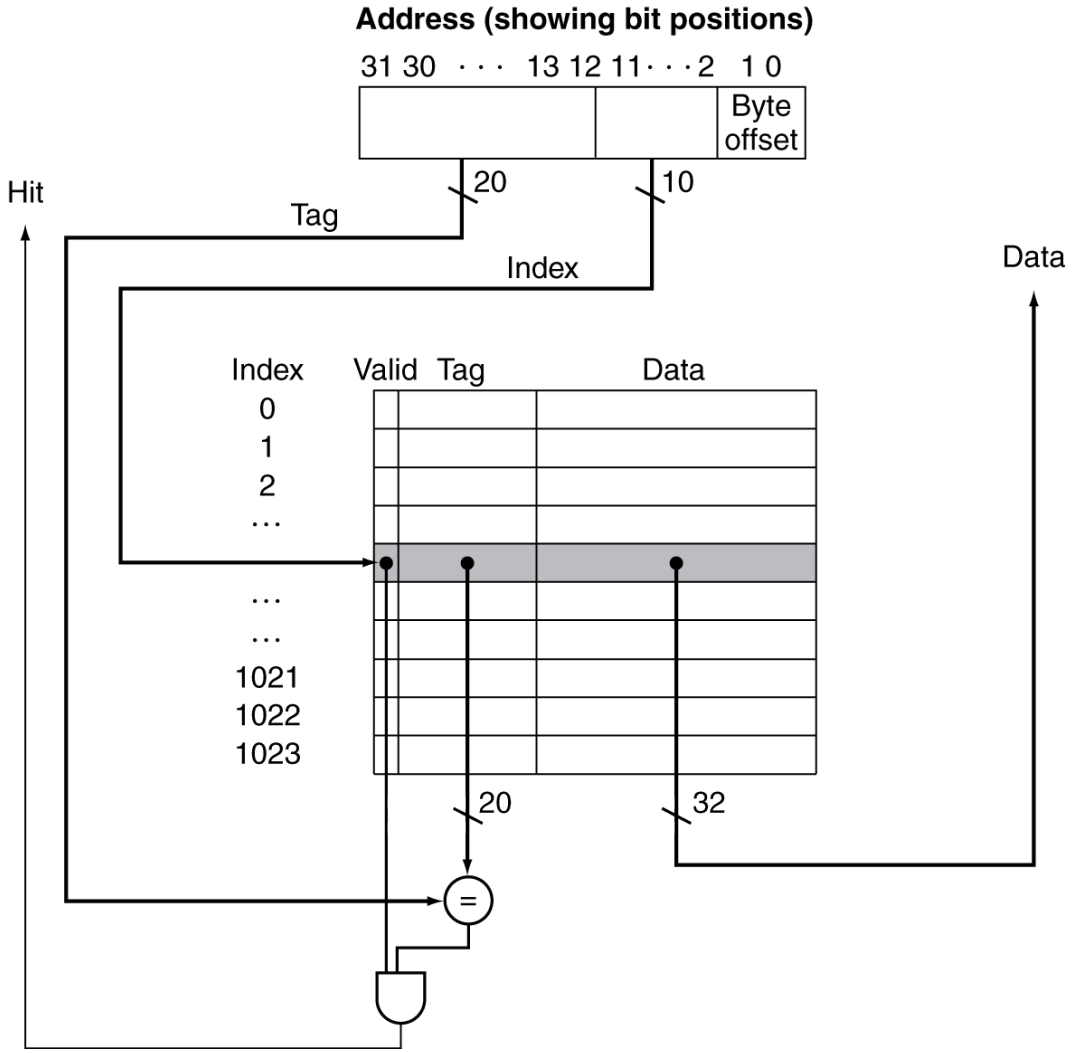
Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

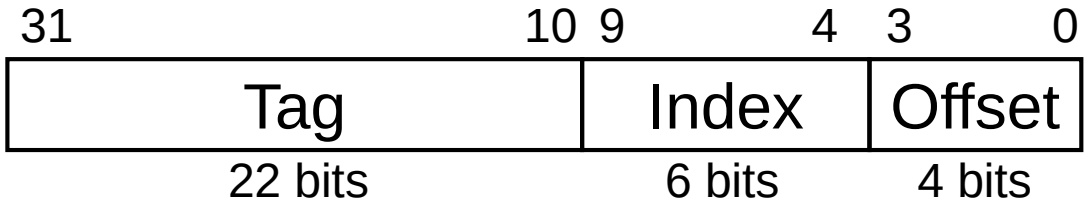
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Address Subdivision



Example: Larger Block Size

- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
- Block address = $\lfloor 1200/16 \rfloor = 75$
- Block number = 75 modulo 64 = 11



Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks ⇒ fewer of them
    - More competition ⇒ increased miss rate
  - Larger blocks ⇒ pollution
- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first

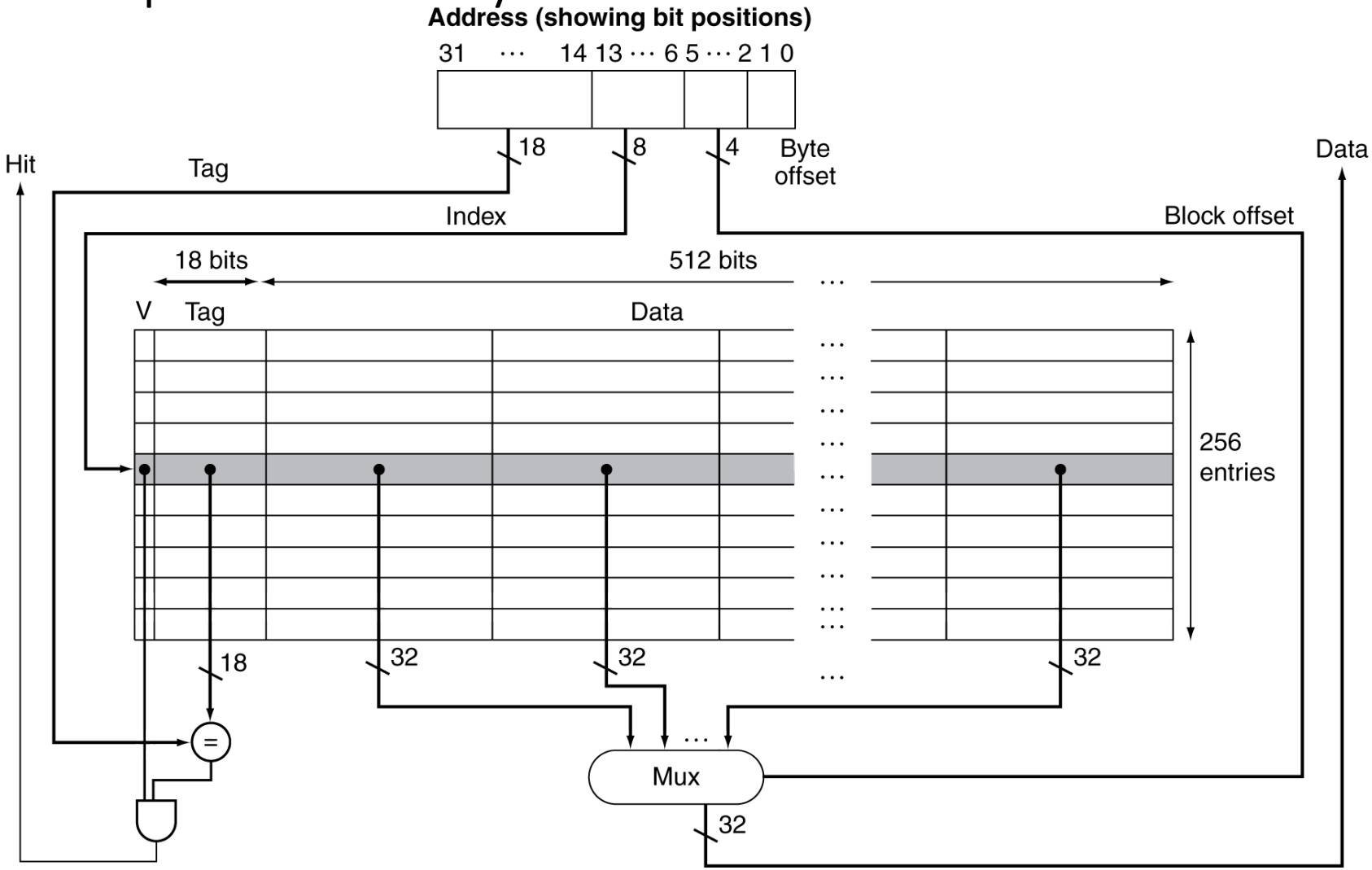
Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block

Example: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

Example: Intrinsity FastMATH



Main Memory Supporting Caches

- Use DRAMs for main memory
  - Fixed width (e.g., 1 word)
  - Connected by fixed-width clocked bus
    - Bus clock is typically slower than CPU clock
- Example cache block read
  - 1 bus cycle for address transfer
  - 15 bus cycles per DRAM access
  - 1 bus cycle per data transfer
- For 4-word block, 1-word-wide DRAM
  - Miss penalty = 1 + 4×15 + 4×1 = 65 bus cycles
  - Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle

Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Cache Performance Example

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: 0.02 × 100 = 2
  - D-cache: 0.36 × 0.04 × 100 = 1.44
- Actual CPI = 2 + 2 + 1.44 = 5.44
  - Ideal CPU is 5.44/2 =2.72 times faster

Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

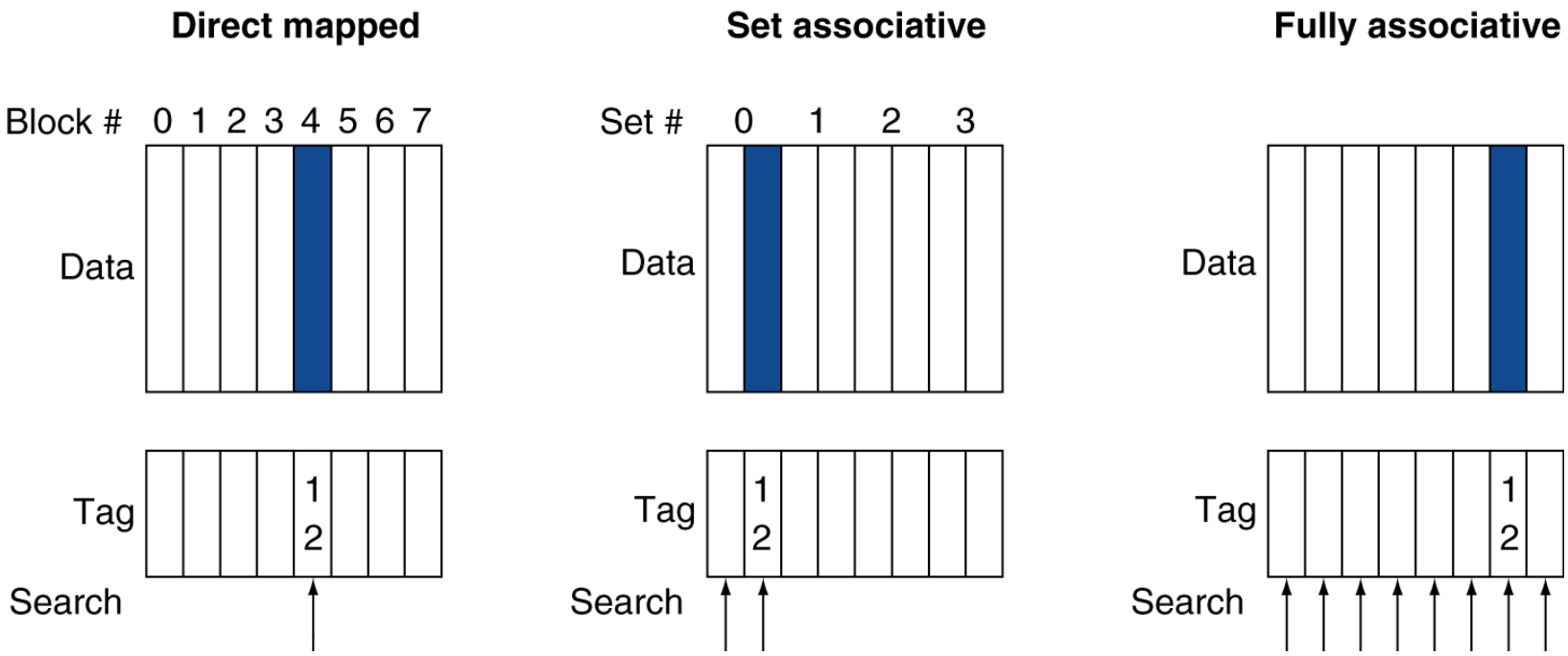
Performance Summary

- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can't neglect cache behavior when evaluating system performance

Associative Caches

- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

Associative Cache Example



Spectrum of Associativity

- For a cache with 8 entries

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

Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

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]	

Associativity Example

- 2-way set associative

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]		

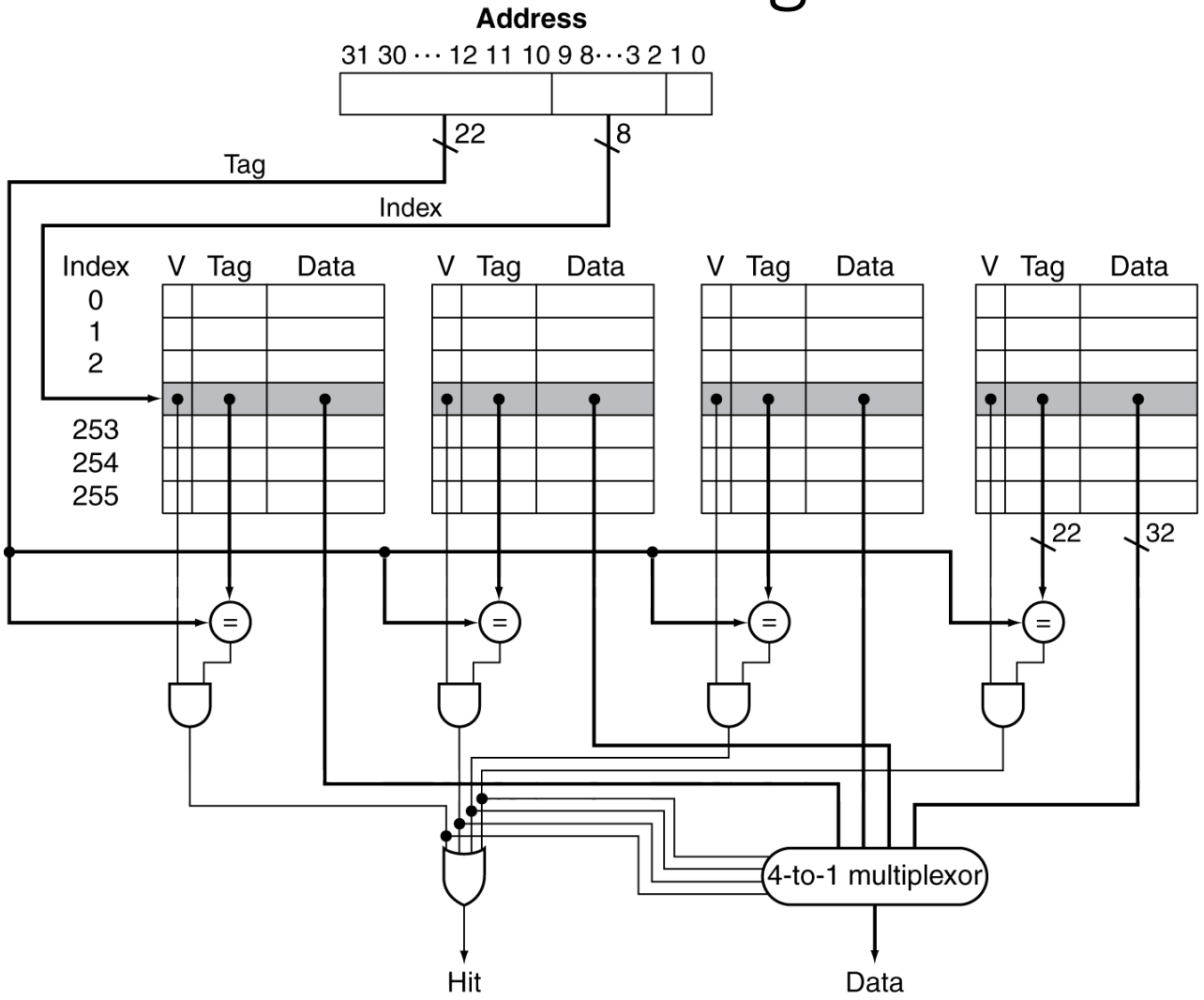
- Fully associative

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]	

How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Set Associative Cache Organization



Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity

Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 500 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- Performance ratio = 9/3.4 = 2.6

Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

Interactions with Advanced CPUs

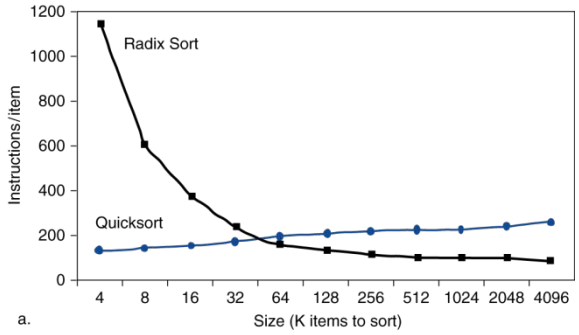
- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyse
  - Use system simulation

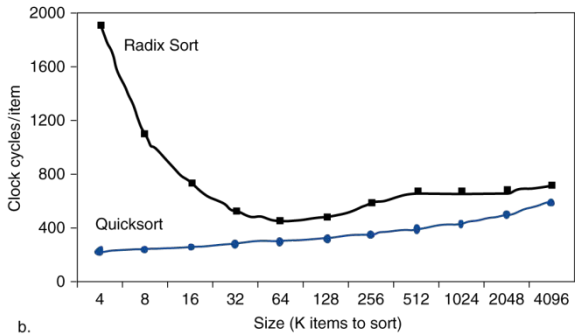
Summary

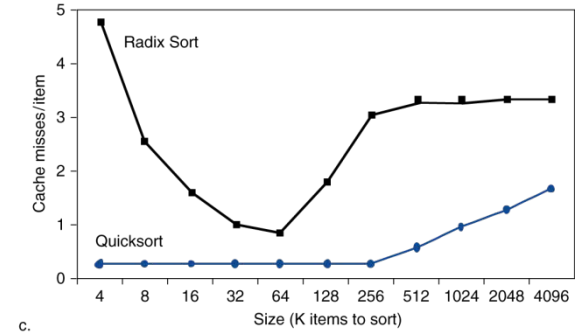
- Memory hierarchy
  - Cache
  - Main memory
  - Disk / storage
- Caches
  - Direct-mapped vs. associative
  - Tags
  - Indices
  - Valid bits
  - Write-back vs. write-through

Interactions with Software

- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory access







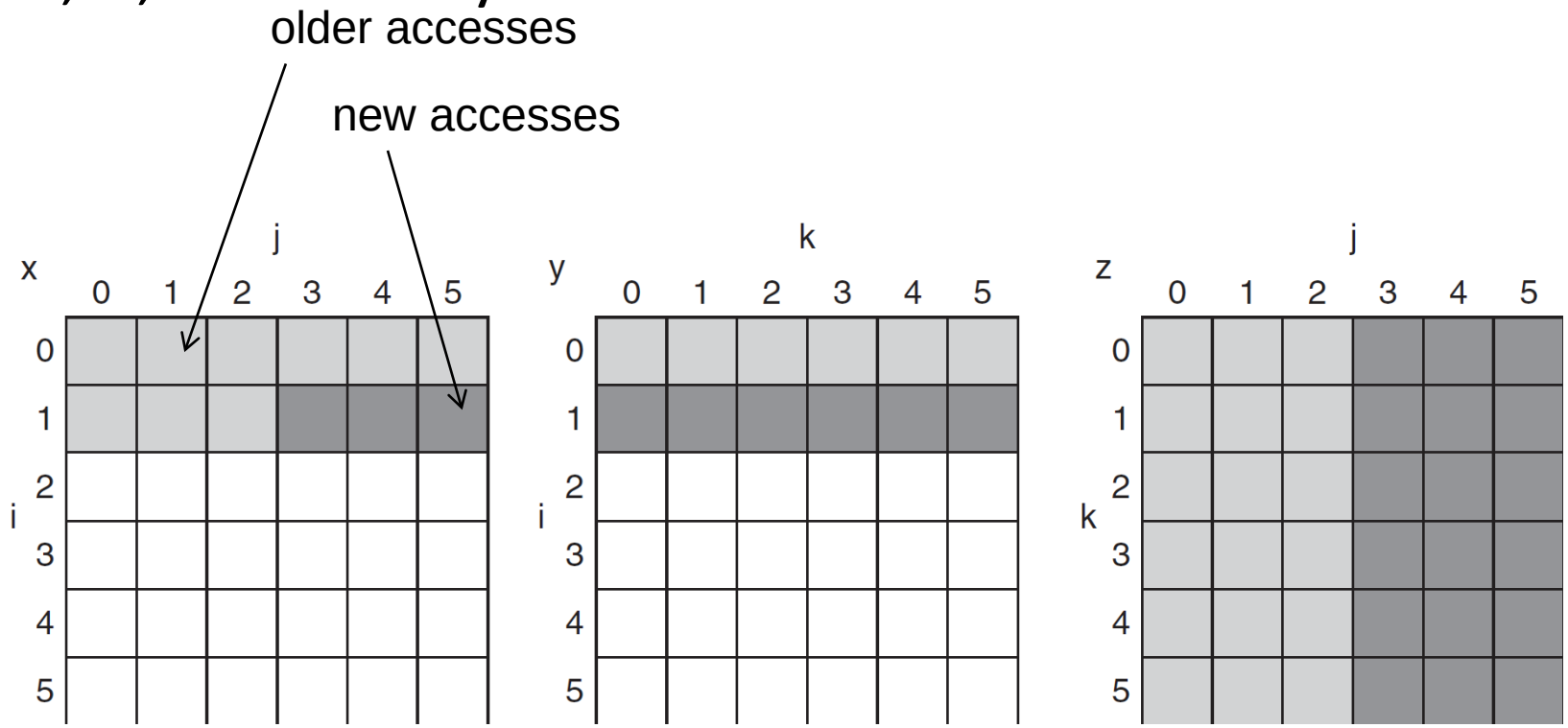
Software Optimization via Blocking

- Goal: maximize accesses to data before it is replaced
- Consider inner loops of DGEMM:

```
for (int j = 0; j < n; ++j)
{
  double cij = C[i+j*n];
  for( int k = 0; k < n; k++ )
    cij += A[i+k*n] * B[k+j*n];
  C[i+j*n] = cij;
}
```

DGEMM Access Pattern

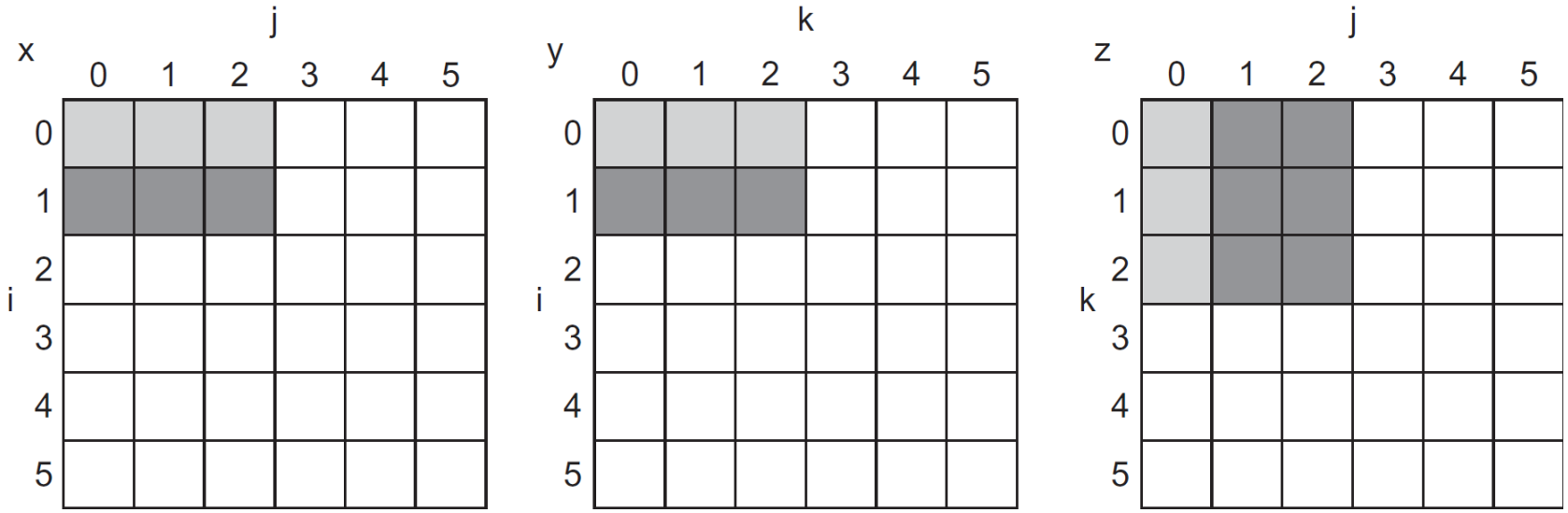
- C, A, and B arrays

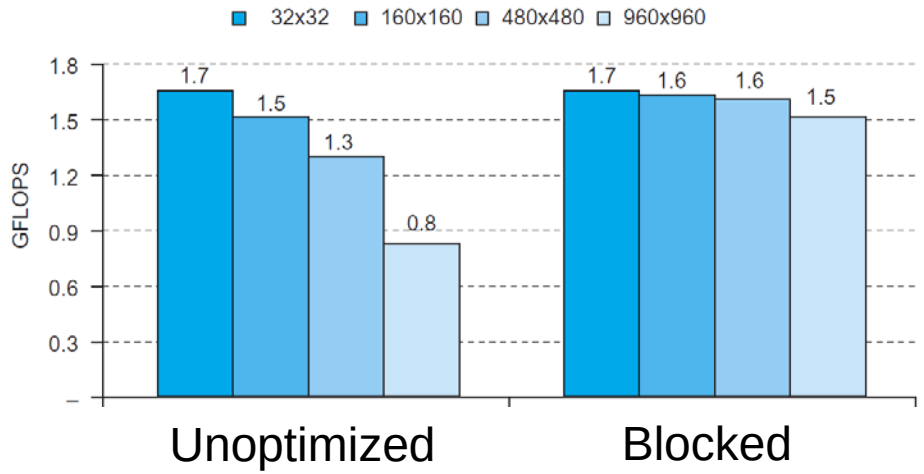


Cache Blocked DGEMM

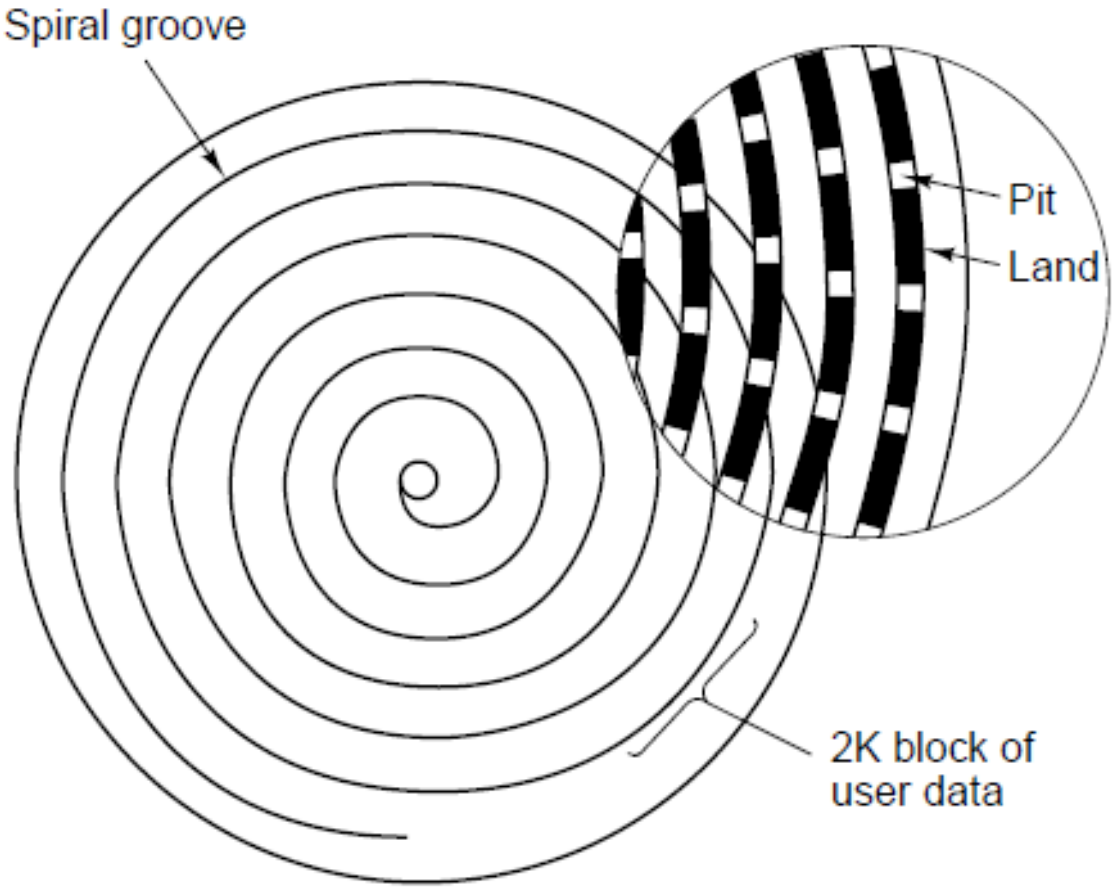
```
1 #define BLOCKSIZE 32
2 void do_block (int n, int si, int sj, int sk, double *A, double
3 *B, double *C)
4 {
5  for (int i = si; i < si+BLOCKSIZE; ++i)
6   for (int j = sj; j < sj+BLOCKSIZE; ++j)
7   {
8    double cij = C[i+j*n];/* cij = C[i][j] */
9    for( int k = sk; k < sk+BLOCKSIZE; k++ )
10    cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11   C[i+j*n] = cij;/* C[i][j] = cij */
12  }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16  for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17   for ( int si = 0; si < n; si += BLOCKSIZE )
18    for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19     do_block(n, si, sj, sk, A, B, C);
20 }
```

Blocked DGEMM Access Pattern

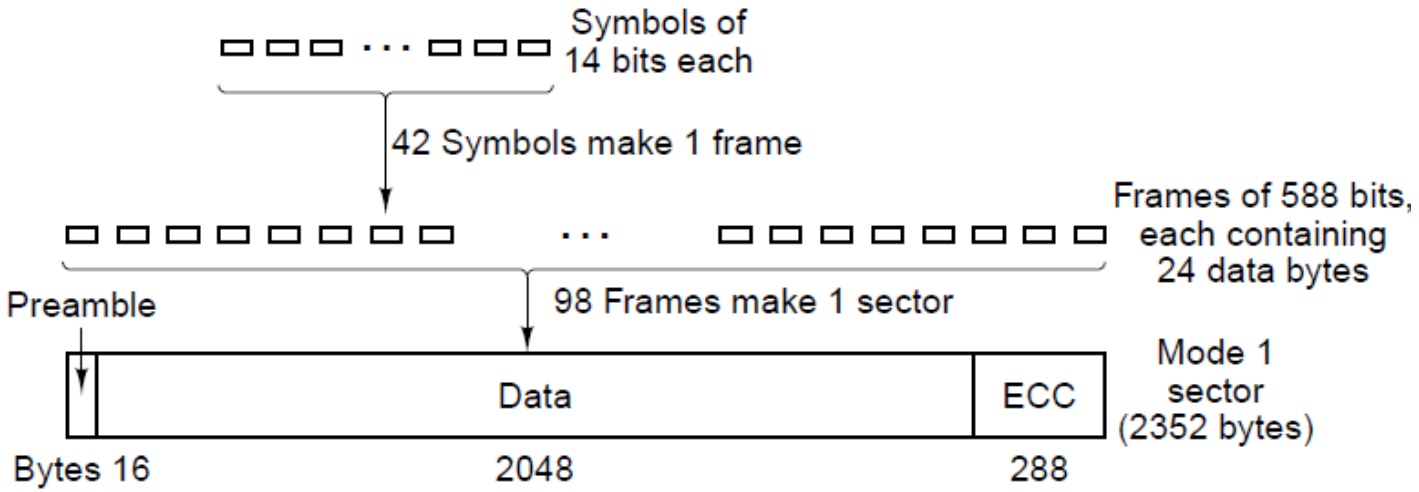




CDs

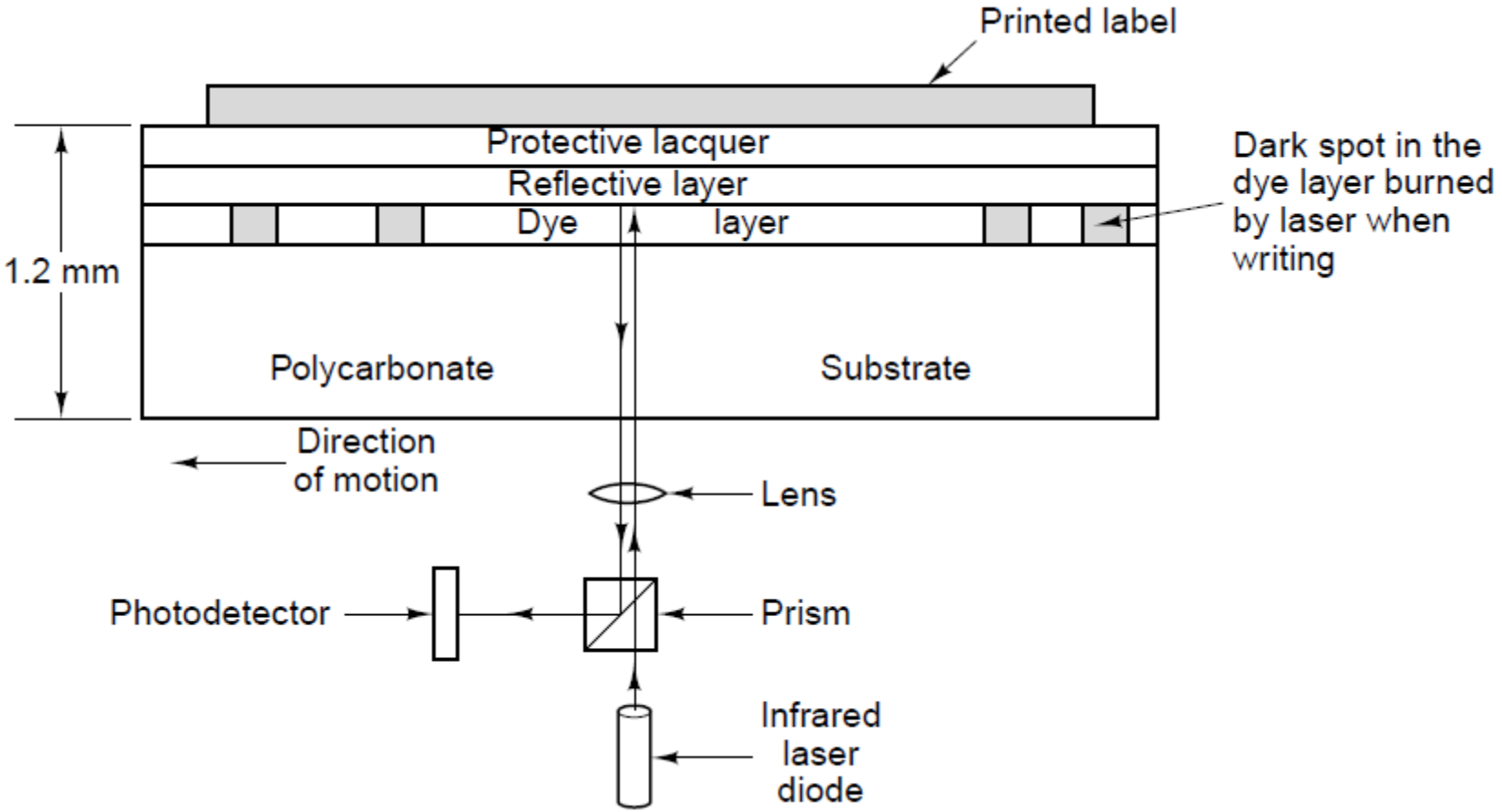


CDs



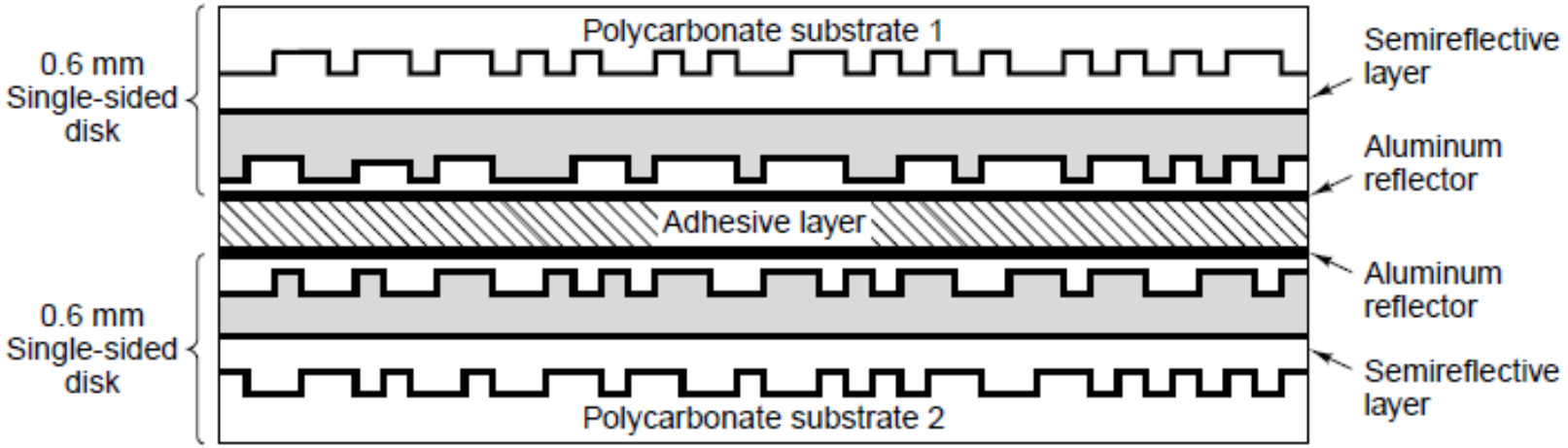
- Mode 1
  - 16 bytes preamble, 2048 bytes data, 288 bytes error-correcting code
  - Single Speed CD-ROM: 75 sectors/sec, so data rate: 75*2048=153,600 bytes/sec
  - 74 minutes audio CD: Capacity: 74*60*153,600=681,984,000 bytes ~=650 MB
- Mode 2
  - 2336 bytes data for a sector, 75*2336=175,200 bytes/sec

CD-R

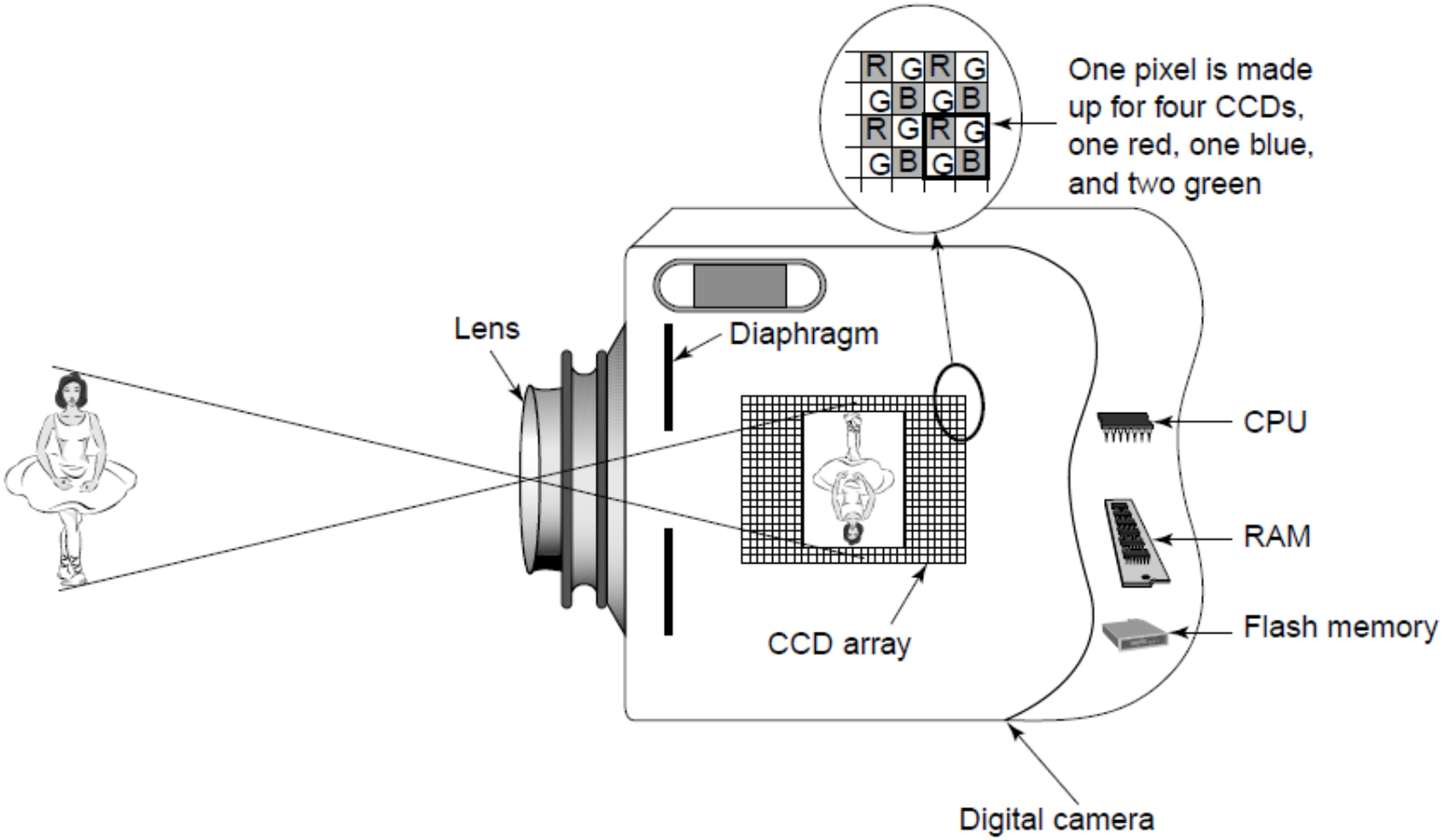

Printed label
Protective lacquer
Reflective layer
Dye layer
Dark spot in the dye layer burned by laser when writing
1.2 mm
Polycarbonate
Substrate
Direction of motion
Lens
Photodetector
Prism
Infrared laser diode

DVDs

- Single-sided, single-layer (4.7 GB)
- Single-sided, dual-layer (8.5 GB)
- Double-sided, single-layer (9.4 GB)
- Double-sided, dual-layer (17 GB)



Storing Images



Optical Disks

- Disks in this family include:
  - CDs, DVDs, Blu-ray disks.
- The basic technology is similar, but improvements have led to higher capacities and speeds.
- Optical disks are much slower than magnetic drives.
- These disks are a cheap option for write-once purposes.
  - Great for mass distribution of data (software, music, movies).
- CD capacity: 650-700MB.
  - Minimum data rate: 150KB/sec.
- DVD capacity: 4.7GB to 17GB.
  - Minimum data rate: 1.4MB/sec.
- Blu-ray capacity: 25GB-50GB.
  - Minimum data rate: 4.5MB/sec.

Optical Disk Capacities

- CD capacity: 650-700MB.
  - Minimum data rate: 150KB/sec.
- DVD capacity: 4.7GB to 17GB.
  - Minimum data rate: 1.4MB/sec.
  - Single-sided, single-layer: 4.7GB.
  - Single-sided, dual-layer: 8.5GB.
  - Double-sided, single-layer: 9.4GB.
  - Double-sided, dual-layer: 17GB.
- Blu-ray capacity: 25GB-50GB.
  - Minimum data rate: 4.5MB/sec.
  - Single-sided: 25GB.
  - Double-sided: 50GB.

Magnetic Disks

- Consists of one or more platters with magnetizable coating
- Disk head containing induction coil floats just over the surface
- When a positive or negative current passes through head, it magnetizes the surface just beneath the head, aligning the magnetic particles face right or left, depending on the polarity of the drive current
- When head passes over a magnetized area, a positive or negative current is induced in the head, making it possible to read back the previously stored bits
- Track
  - Circular sequence of bits written as disk makes complete rotation
  - Sector: Each track is divided into some sector with fixed length

Classical Hard Drives: Magnetic Disks

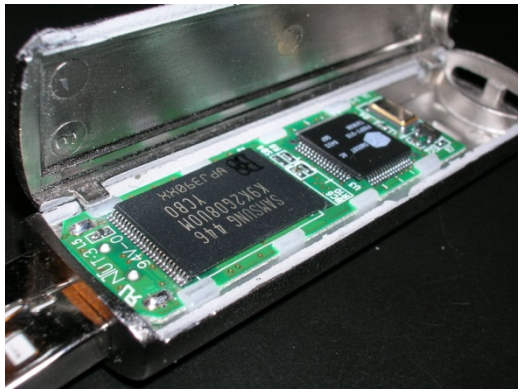
- A magnetic disk is a disk, that spins very fast.
  - Typical rotation speed: 5400, 7200, 10800 RPMs.
  - RPMs: rotations per minute.
  - These translate to 90, 120, 180 rotations per second.
- The disk is divided into rings, that are called **tracks**.
- Data is read by the **disk head**.
  - The head is placed at a specific radius from the disk center.
  - That radius corresponds to a specific track.
  - As the disk spins, the head reads data from that track.

Solid-State Drives

- A solid-state drive (SSD) is NOT a spinning disk. It is just cheap memory.
- Compared to hard drives, SSDs have two to three times faster speeds, and **~100nsec access time.**
- Because SSDs have no mechanical parts, they are well-suited for mobile computers, where motion can interfere with the disk head accessing data.
- Disadvantage #1: price.
  - Magnetic disks: pennies/gigabyte.
  - SSDs: one to three dollars/gigabyte.
- Disadvantage #2: failure rate.
  - A bit can be written about 100,000 times, then it fails.

Flash Storage

- Nonvolatile semiconductor storage
  - 100× – 1000× faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)

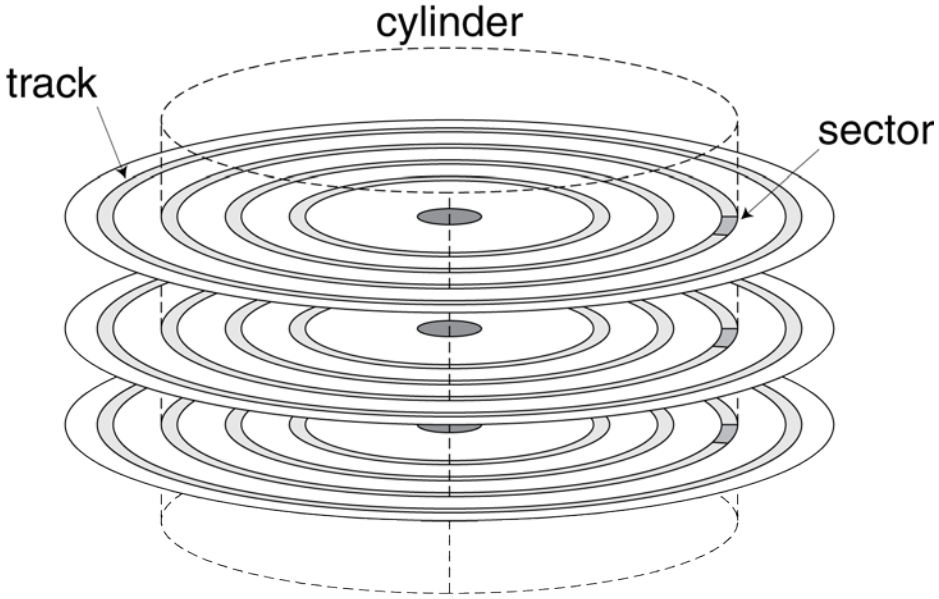


Flash Types

- NOR flash: bit cell like a NOR gate
  - Random read/write access
  - Used for instruction memory in embedded systems
- NAND flash: bit cell like a NAND gate
  - Denser (bits/area), but block-at-a-time access
  - Cheaper per GB
  - Used for USB keys, media storage, …
- Flash bits wears out after 1000's of accesses
  - Not suitable for direct RAM or disk replacement
  - Wear leveling: remap data to less used blocks

Disk Storage

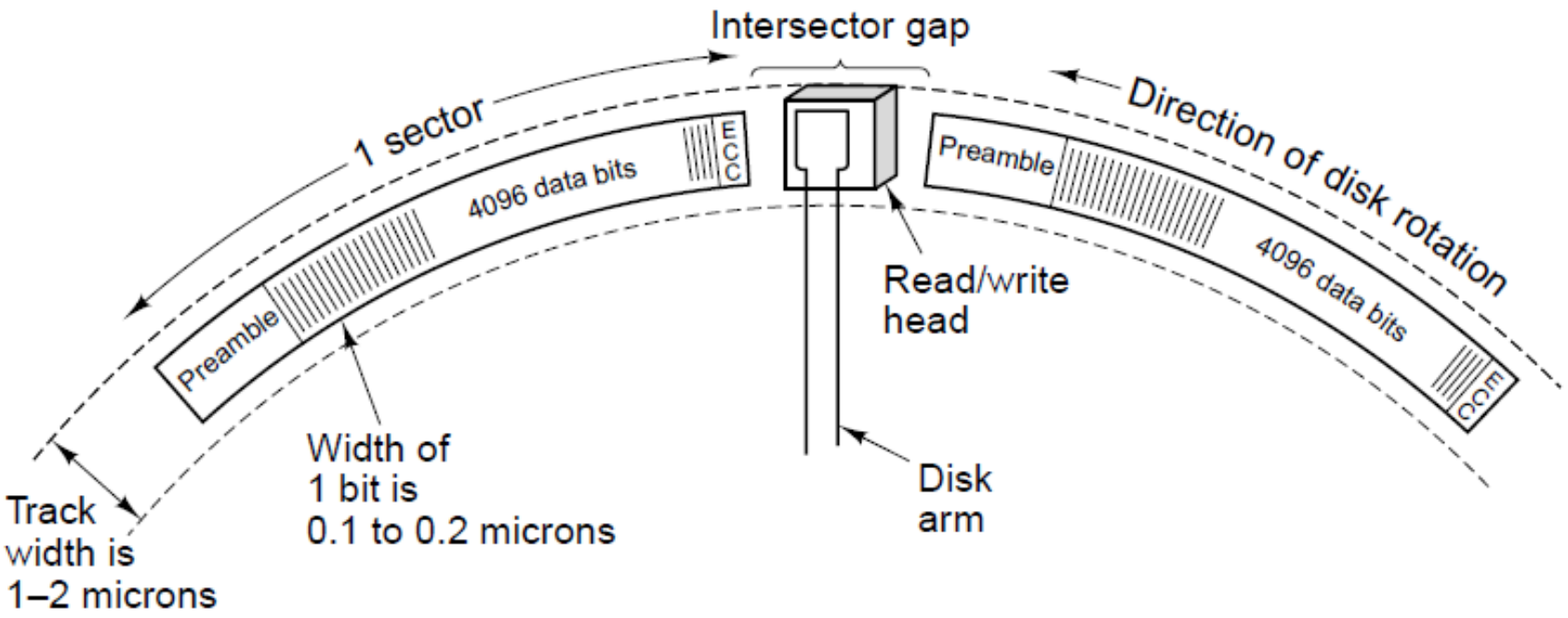
- Nonvolatile, rotating magnetic storage



Disk Tracks and Sectors

- A track can be 0.2µm wide.
  - We can have 50,000 tracks per cm of radius.
  - About 125,000 tracks per inch of radius.
- Each track is divided into fixed-length **sectors**.
  - Typical sector size: 512 bytes.
- Each sector is preceded by a **preamble**. This allows the head to be synchronized before reading or writing.
- In the sector, following the data, there is an error-correcting code.
- Between two sectors there is a small **intersector gap**.

Visualizing a Disk Track



A portion of a disk track. Two sectors are illustrated.

Disk Sectors and Access

- Each sector records
  - Sector ID
  - Data (512 bytes, 4096 bytes proposed)
  - Error correcting code (ECC)
    - Used to hide defects and recording errors
  - Synchronization fields and gaps
- Access to a sector involves
  - Queuing delay if other accesses are pending
  - Seek: move the heads
  - Rotational latency
  - Data transfer
  - Controller overhead

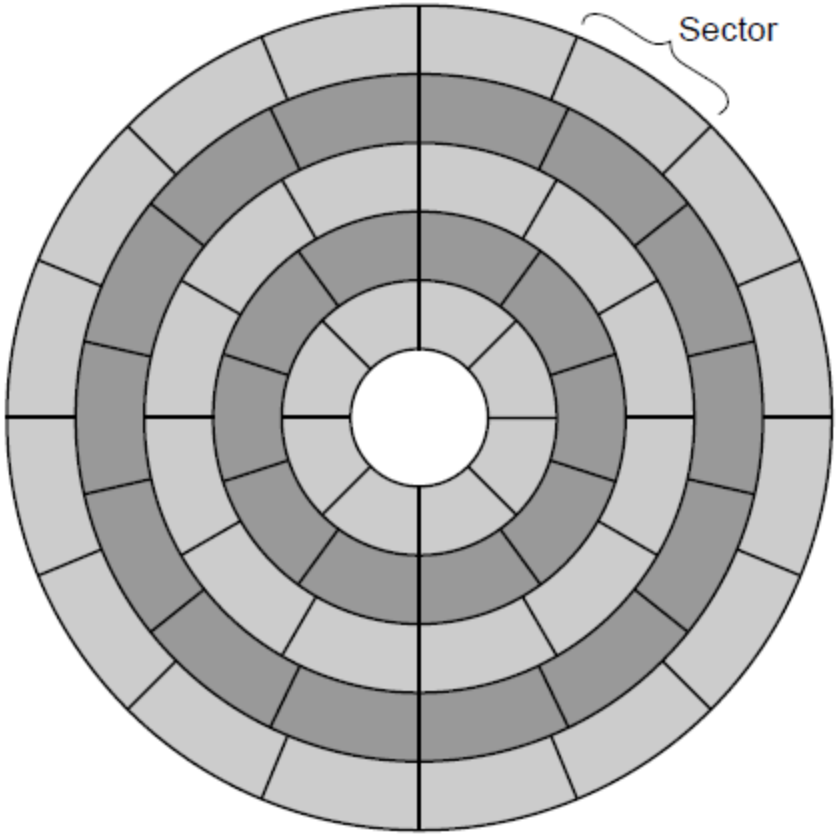
Disk Access Example

- Given
  - 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk
- Average read time
  - 4ms seek time
    + ½ / (15,000/60) = 2ms rotational latency
    + 512 / 100MB/s = 0.005ms transfer time
    + 0.2ms controller delay
    = 6.2ms
- If actual average seek time is 1ms
  - Average read time = 3.2ms

Disk Performance Issues

- Manufacturers quote average seek time
  - Based on all possible seeks
  - Locality and OS scheduling lead to smaller actual average seek times
- Smart disk controller allocate physical sectors on disk
  - Present logical sector interface to host
  - SCSI, ATA, SATA
- Disk drives include caches
  - Prefetch sectors in anticipation of access
  - Avoid seek and rotational delay

Magnetic Disk Sectors



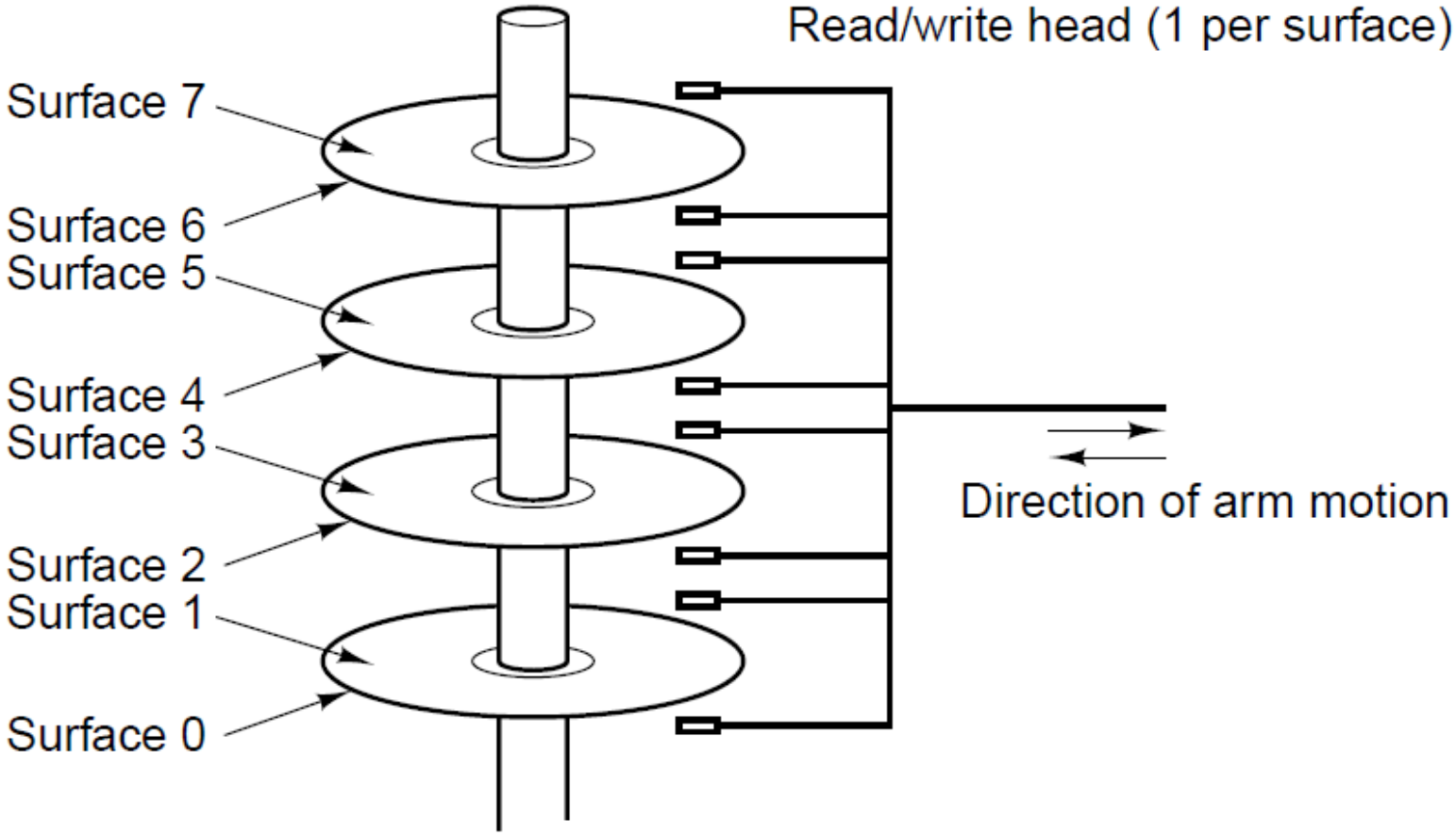
Measuring Disk Capacity

- Disk capacity is often advertized in unformatted state.
- However, **formatting** takes away some of this capacity.
  - Formatting creates preambles, error-correcting codes, and gaps.
- The formatted capacity is typically about 15% lower than unformatted capacity.

Multiple Platters

- A typical hard drive unit contains multiple platters, i.e., multiple actual disks.
- These platters are stacked vertically (see figure).
- Each platter stores information on both surfaces.
- There is a separate arm and head for each surface.

Magnetic Disk Platters



Cylinders

- The set of tracks corresponding to a specific radial position is called a **cylinder**.
- Each track in a cylinder is read by a different head.

Data Access Times

- Suppose we want to get some data from the disk.
- First, the head must be placed on the right track (i.e., at the right radial distance).
  - This is called **seek**.
  - Average seek times are in the 5-10 msec range.
- Then, the head waits for the disk to rotate, so that it gets to the right sector.
  - Given that disks rotate at 5400-10800 RPMs, this incurs an average wait of 3-6 msec. This is called **rotational latency**.
- Then, the data is read. A typical rate for this stage is 150MB/sec.
  - So, a 512-byte sector can be read in ~3.5 µsec.

Measures of Disk Speed

- **Maximum Burst Rate**: the rate (number of bytes per sec) at which the head reads a sector, **once the had has started seeing the first data bit**.
  - This excludes seeks, rotational latencies, going through preambles, error-correcting codes, intersector gaps.
- **Sustained Rate**: the actual average rate of reading data over several seconds, that includes all the above factors (seeks, rotational latencies, etc.).

Worst Case Speed

- Rarely advertised, but VERY IMPORTANT to be aware of if your software accesses the hard drive: **the worst case speed**.
- What scenario gives us the worst case?

Worst Case Speed

- Rarely advertised, but VERY IMPORTANT to be aware of if your software accesses the hard drive: **the worst case speed**.
- What scenario gives us the worst case?
  - Read random positions, one byte at a time.
  - To read each byte, we must perform a seek, wait for the rotational latency, go through the sector preamble, etc.
- If this whole process takes about 10 msec (which may be a bit optimistic), we can only read ???/sec?

Worst Case Speed

- Rarely advertised, but VERY IMPORTANT to be aware of if your software accesses the hard drive: **the worst case speed**.
- What scenario gives us the worst case?
  - Read random positions, one byte at a time.
  - To read each byte, we must perform a seek, wait for the rotational latency, go through the sector preamble, etc.
- If this whole process takes about 10 msec (which may be a bit optimistic), we can only read 100 bytes/sec.
  - More than a million times slower than the maximum burst rate.

Worst Case Speed

- Reading a lot of non-contiguous small chunks of data kills magnetic disk performance.
- When your programs access disks a lot, it is important to understand how disk data are read, to avoid this type of pitfall.

Disk Controller

- The disk controller is a chip that controls the drive.
  - Some controllers contain a full CPU.
- Controller tasks:
  - Execute commands coming from the software, such as:
    - READ
    - WRITE
    - FORMAT (writing all the preambles)
  - Control the arm motion.
  - Detect and correct errors.
  - Buffer multiple sectors.
  - Cache sectors read for potential future use.
  - Remap bad sectors.

IDE and SCSI Drives

- IDE and SCSI drives are the two most common types of hard drives on the market.
- The book goes into a lot of details about each of these types.
- We will skip that in this class.
  - We skip textbook sections 2.3.3 and 2.3.4.
- Just be aware that:
  - IDE drives are cheaper and slower.
    - Newer IDE drives are also called serial ATA or SATA.
  - SCSI drives are more expensive and faster.
- Most inexpensive computers use IDE drives.

RAID

- RAID stands for *Redundant Array of Inexpensive Disks.*
- RAID arrays are simply sets of disks, that are visible as a single unit by the computer.
  - Instead of a single drive accessible via a drive controller, the whole RAID is accessible via a RAID controller.
  - Since a RAID can look as a single drive, software accessing disks does not need to be modified to access a RAID.
- Depending on their type (we will see several types), RAIDs accomplish one (or both) of the following:
  - Speed up performance.
  - Tolerate failures of entire drive units.

RAID for Faster Speed

- Disk performance has not improved as dramatically as CPU performance.
- In the 1970s, average seek times on minicomputer disks were 50-100 msec.
- Now they have improved to 5-10 msec.
- The slow gains in performance have motivated people to look into ways to gain speed via parallel processing.

RAID-0

- RAID level 0: Improves speed via **striping**.
  - When a write request comes in, data is broken into strips.
  - Each strip is written to a different drive, in round-robin fashion.
  - Thus, multiple strips are written in parallel, effectively leading to faster speed, compared to using a single drive.
- Effect: most files are stored in a distributed manner: with different pieces of them stored on each drive of the RAID.
- When reading a file, the different pieces (strips) are read again in parallel, from all drives.

RAID-0 Example

- Suppose we have a RAID-0 system with 8 disks.
- What is the best case scenario, in which performance will be the best, compared to a single disk?

- Compared to a single disk, in the **best** case:
  - The write performance of RAID-0 is: ???
  - The read performance of RAID-0 is: ???
- What is the best case scenario, in which performance will be the best, compared to a single disk?

- Compared to a single disk, in the **worst** case:
  - The write performance of RAID-0 is: ???
  - The read performance of RAID-0 is: ???

RAID-0 Example

- Suppose we have a RAID-0 system with 8 disks.
- What is the best case scenario, in which performance will be the best, compared to a single disk?
  - Reading/writing large chunks of data, so striping can be exploited.
- Compared to a single disk, in the **best** case:
  - The write performance of RAID-0 is: 8 times faster than a single disk.
  - The read performance of RAID-0 is: 8 times faster than a single disk.
- What is the best case scenario, in which performance will be the best, compared to a single disk?
  - Reading/writing many small, unrelated chunks of data (e.g., a single byte at a time). Then, striping cannot be used.
- Compared to a single disk, in the **worst** case:
  - The write performance of RAID-0 is: the same as that of a single disk.
  - The read performance of RAID-0 is: the same as that of a single disk.

RAID-0: Pros and Cons

- RAID-0 works the best for large read/write requests.
- RAID-0 speed deteriorates into that of a single drive if the software asks for data in chunks of one strip (or less) at a time.
- How about reliability? A RAID-0 is **less** reliable, and more prone to failure than that of a single drive.
  - Suppose we have a RAID with four drives.
  - Each drive has a mean time to failure of 20,000 hours.
  - Then, the RAID has a mean time to failure that is ??? hours?

RAID-0: Pros and Cons

- RAID-0 works the best for large read/write requests.
- RAID-0 speed deteriorates into that of a single drive if the software asks for data in chunks of one strip (or less) at a time.
- How about reliability? A RAID-0 is **less** reliable, and more prone to failure than that of a single drive.
  - Suppose we have a RAID with four drives.
  - Each drive has a mean time to failure of 20,000 hours.
  - Then, the RAID has a mean time to failure that is only 5000 hours.
- RAID-0 is not a "true" RAID, no drive is redundant.

RAID-1

- In RAID-1, we need to have an even number of drives.
- For each drive, there is an identical copy.
- When we write data, we write it to both drives.
- When we read data, we read from either of the drives.
- NO STRIPING IS USED.
- Compared to a single disk:
  - The write performance is:
  - The read performance is:
  - Reliability is:

RAID-1

- In RAID-1, we need to have an even number of drives.
- For each drive, there is an identical copy.
- When we write data, we write it to both drives.
- When we read data, we read from either of the drives.
- NO STRIPING IS USED.
- Compared to a single disk:
  - The write performance is: twice as slow.
  - The read performance is: the same.
  - Reliability is: far better, drive failure is not catastrophic.

The Need for RAID-5.

- RAID-0: great for performance, bad for reliability.
  - striping, but no redundant data.
- RAID-1: bad for performance, great for reliability.
  - redundant data, no striping
- RAID-2, RAID-3, RAID-4: have problems of their own.
  - You can read about them in the textbook if you are curious, but they are not very popular.
- RAID-5: great for performance, great for reliability.
  - both redundant data and striping.

RAID-5

- Data is striped for writing.
- If we have N disks, we can process N-1 data strips in parallel.
- For every N-1 data strips, we create an Nth strip, called **parity strip**.
  - The k-th bit in the parity strip ensures that there is an even number of 1-bits in position k in all N strips.
- If any strip fails, its data can be recovered from the other N-1 strips.
- This way, the contents of an entire disk can be recovered.

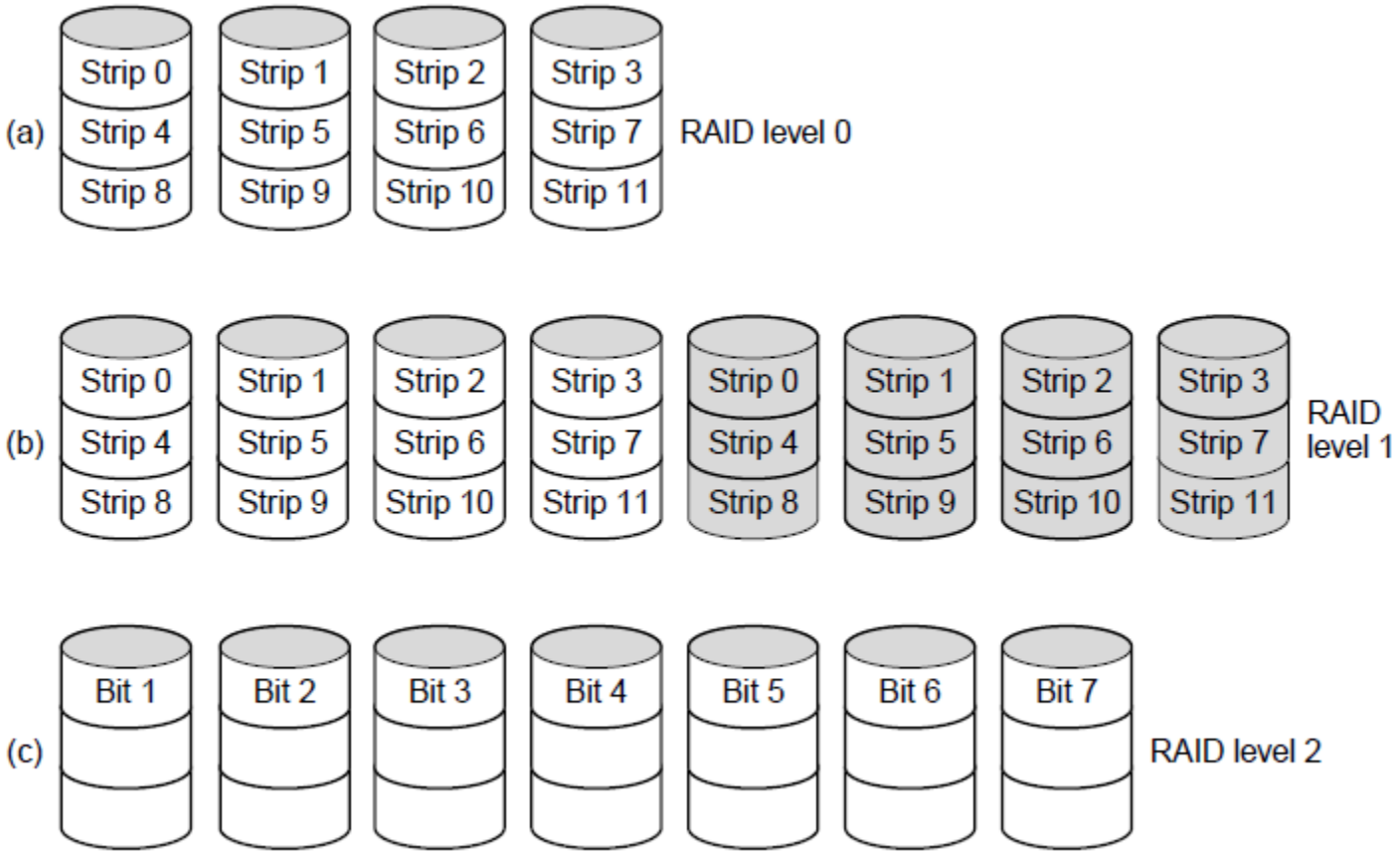
RAID-5 Example

- Suppose we have a RAID-5 system with 8 disks.
- Compared to a single disk, in the **best** case:
  - The write performance of RAID-5 is: ???
  - The read performance of RAID-5 is: ???
- Compared to a single disk, in the **worst** case:
  - The write performance of RAID-5 is: ???
  - The read performance of RAID-5 is: ???

RAID-5 Example

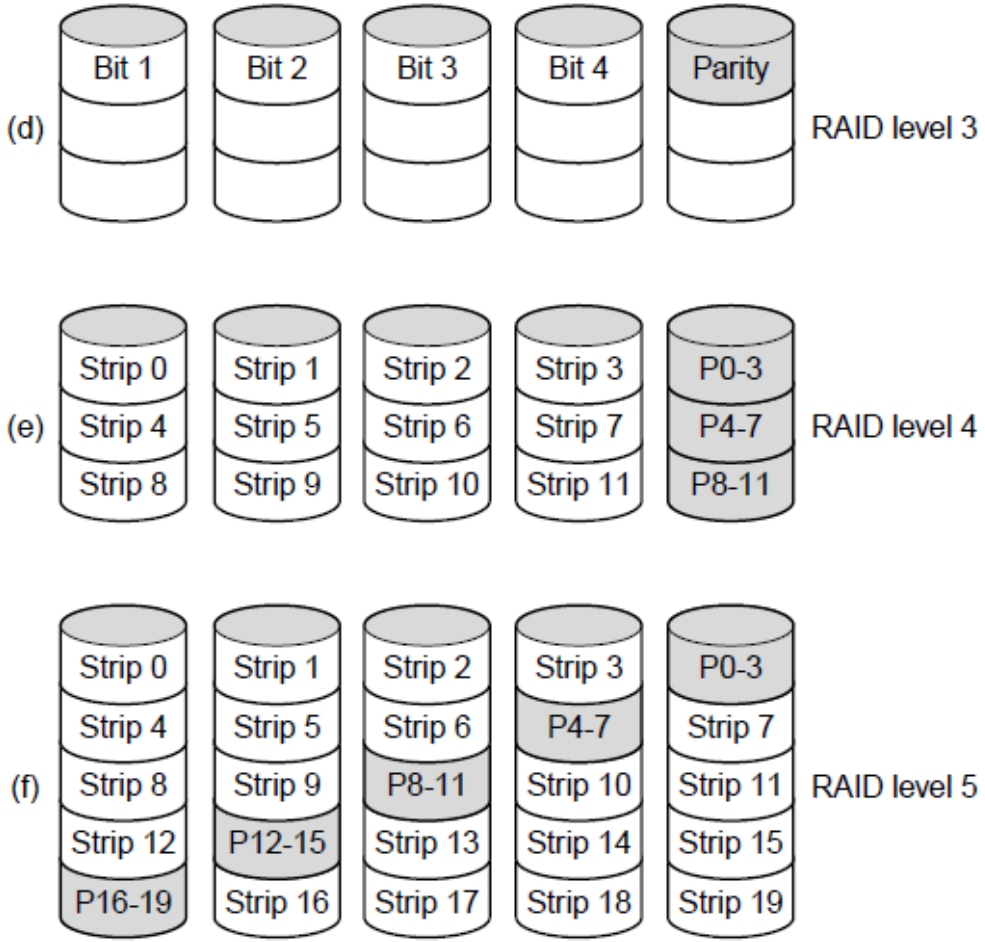
- Suppose we have a RAID-5 system with 8 disks.
- Compared to a single disk, in the **best** case:
  - The write performance of RAID-5 is: 7 times faster than a single disk. (writes non-parity data on 7 disks simultaneously).
  - The read performance of RAID-5 is: 7 times faster than a single disk. (reads non-parity data on 7 disks simultaneously).
- Compared to a single disk, in the **worst** case:
  - The write performance of RAID-5 is: the same as that of a single disk.
  - The read performance of RAID-5 is: the same as that of a single disk.
  - Why? Because striping is not useful when reading/writing one byte at a time.

RAID-0, RAID-1, RAID-2



RAID levels 0 through 5. Backup and parity drives are shown shaded.

RAID-3, RAID-4, RAID-5



RAID levels 0 through 5. Backup and parity drives are shown shaded.