CMSC 313
COMPUTER ORGANIZATION
&
ASSEMBLY LANGUAGE
PROGRAMMING

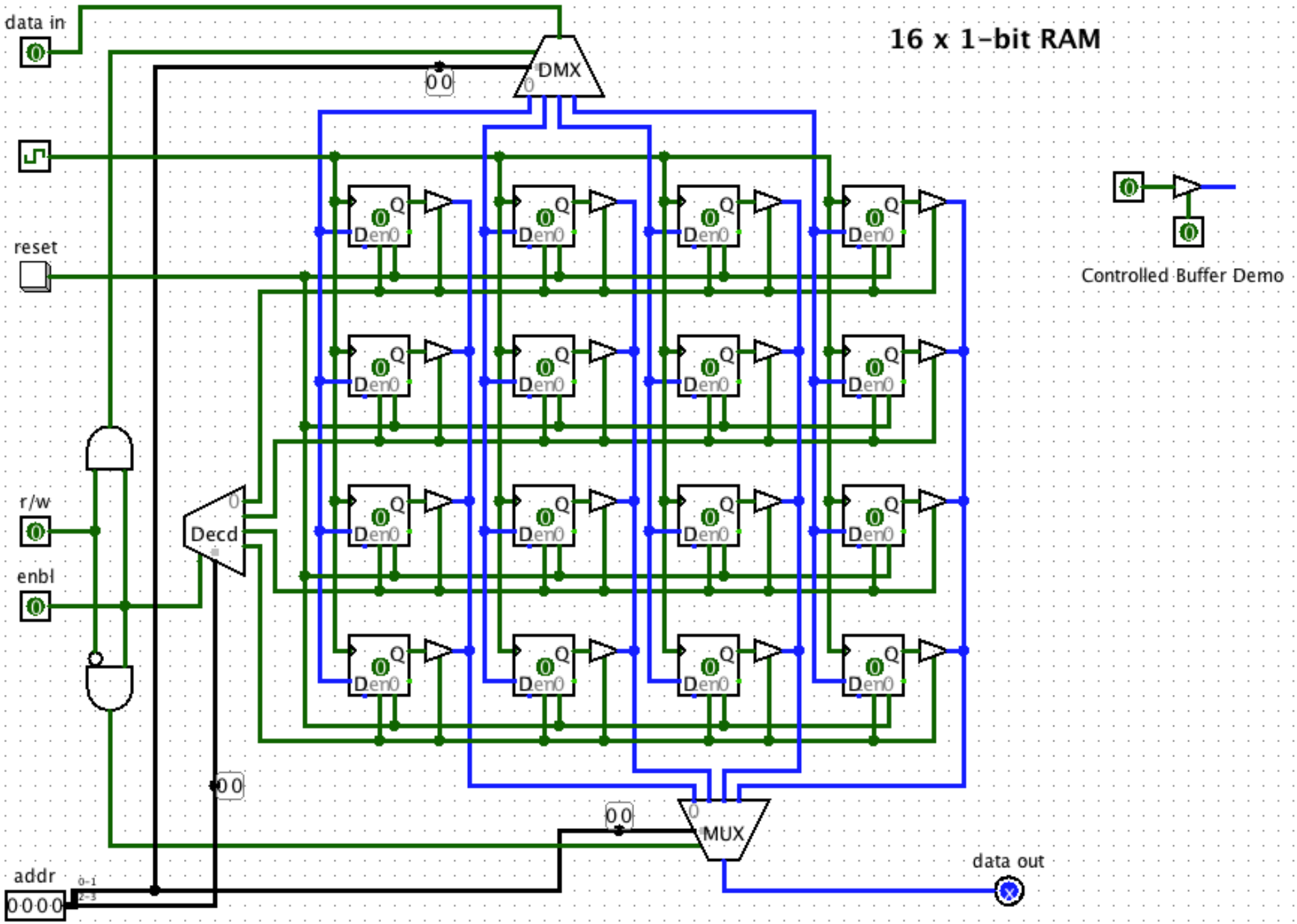
LECTURE 26, FALL 2012

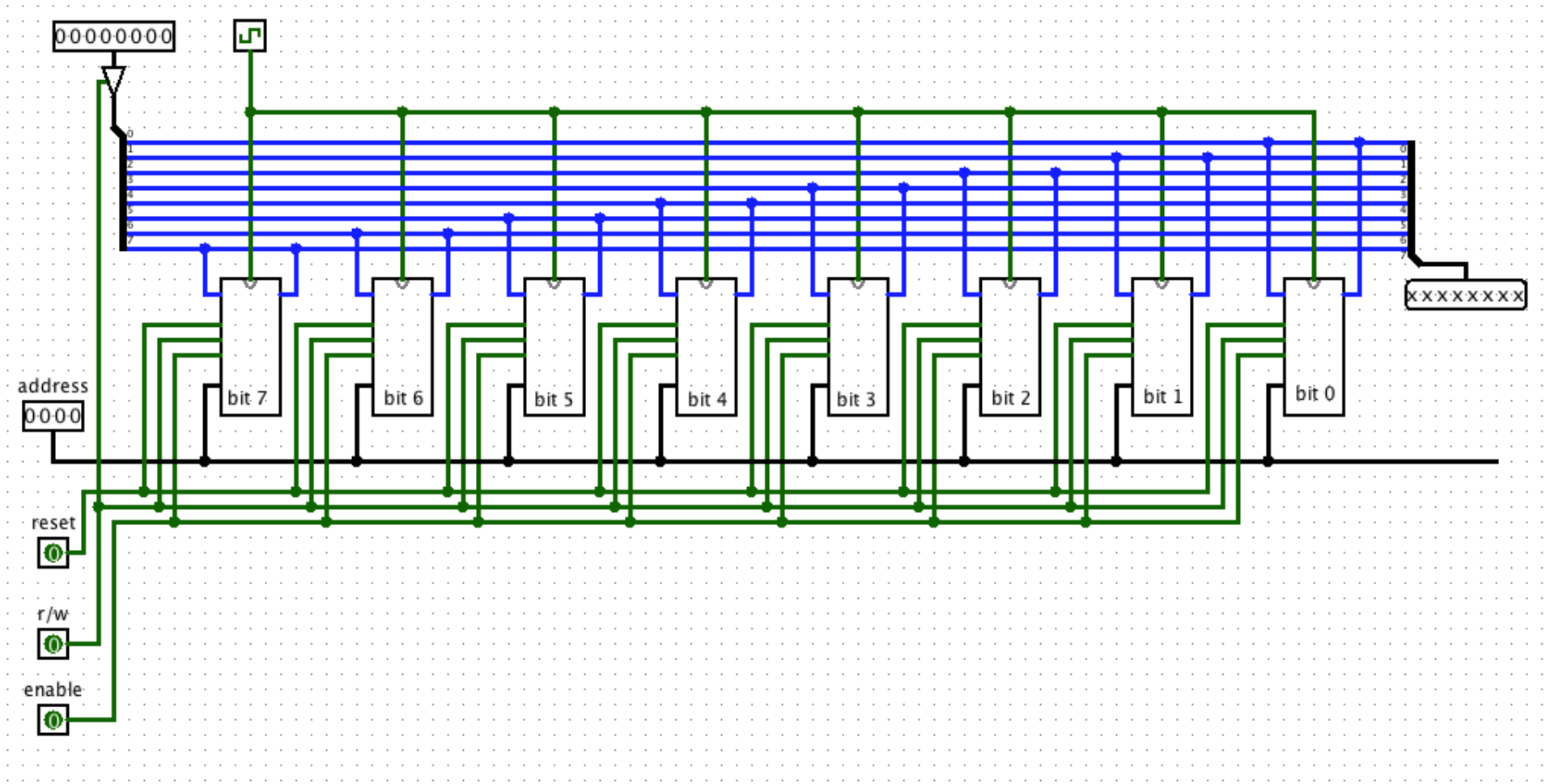
TOPICS TODAY

- Homework 5
- RAM in Circuits
- Memory Hierarchy
- Storage Technologies (RAM & Disk)
- Caching

HOMEWORK 5

RAM IN CIRCUITS

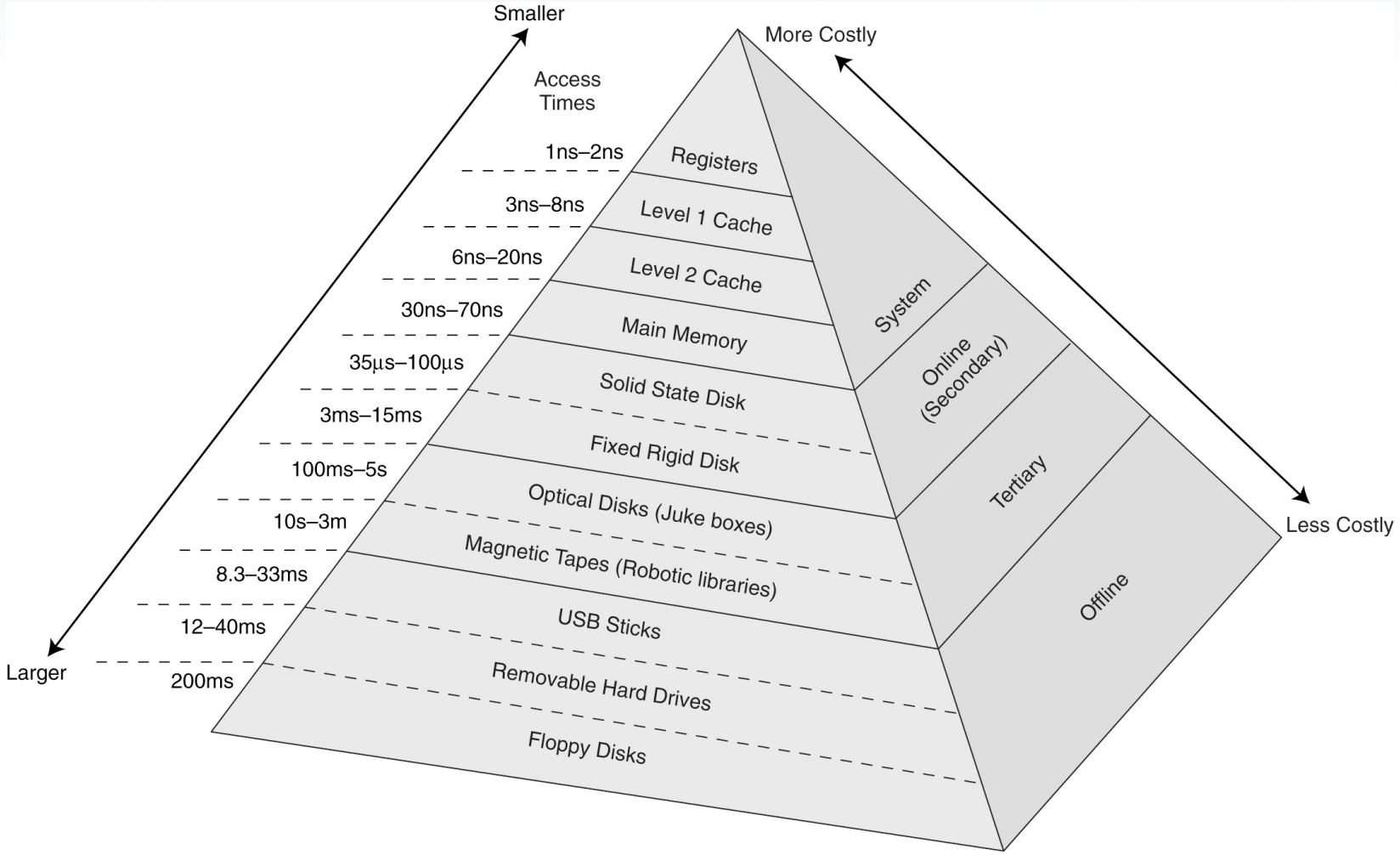

16 x 1-bit RAM
data in
reset
r/w
enbl
addr
0000
DMX
Decd
MUX
data out
Controlled Buffer Demo


00000000
xxxxxxxx
address
0000
bit 7
bit 6
bit 5
bit 4
bit 3
bit 2
bit 1
bit 0
reset
r/w
enable

MEMORY HIERARCHY

6.3 The Memory Hierarchy

- This storage organization can be thought of as a pyramid:



STORAGE TECHNOLOGIES

Random-Access Memory (RAM)

Key features

- RAM is packaged as a chip.
- Basic storage unit is a cell (one bit per cell).
- Multiple RAM chips form a memory.

Static RAM (SRAM)

- Each cell stores bit with a six-transistor circuit.
- Retains value indefinitely, as long as it is kept powered.
- Relatively insensitive to disturbances such as electrical noise.
- Faster and more expensive than DRAM.

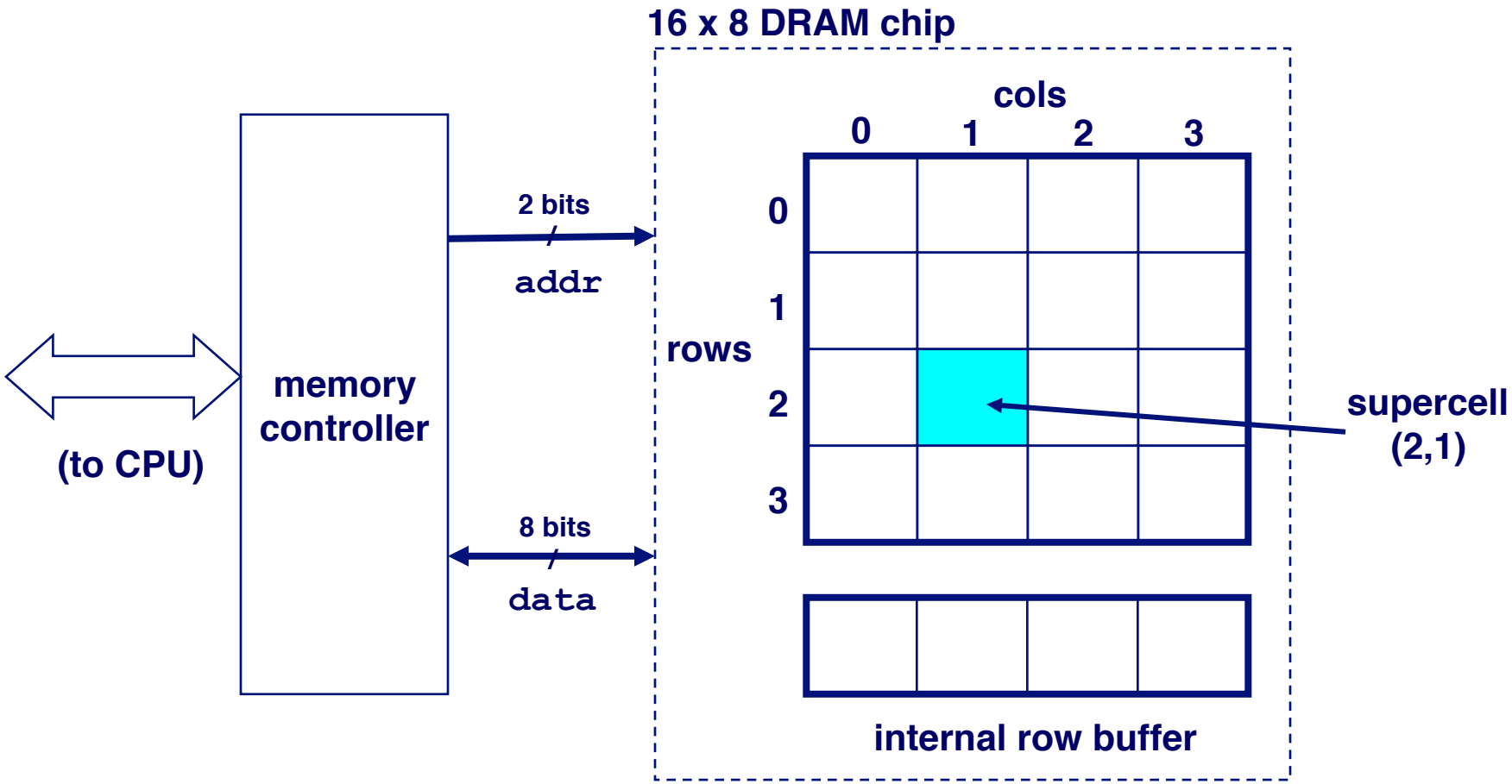
Dynamic RAM (DRAM)

- Each cell stores bit with a capacitor and transistor.
- Value must be refreshed every 10-100 ms.
- Sensitive to disturbances.
- Slower and cheaper than SRAM.

Conventional DRAM Organization

d x w DRAM:

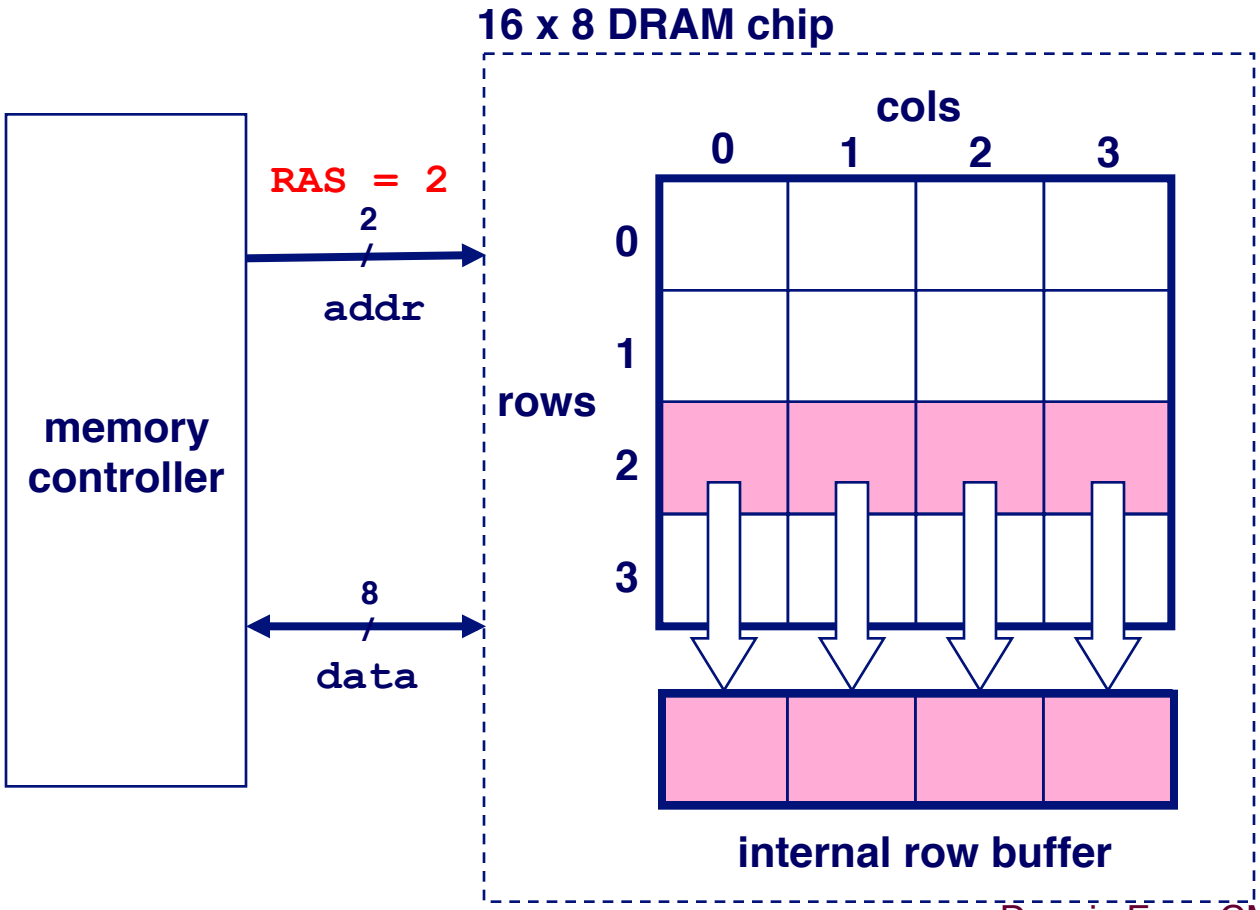
- dw total bits organized as d supercells of size w bits



Reading DRAM Supercell (2,1)

Step 1(a): Row access strobe (RAS) selects row 2.

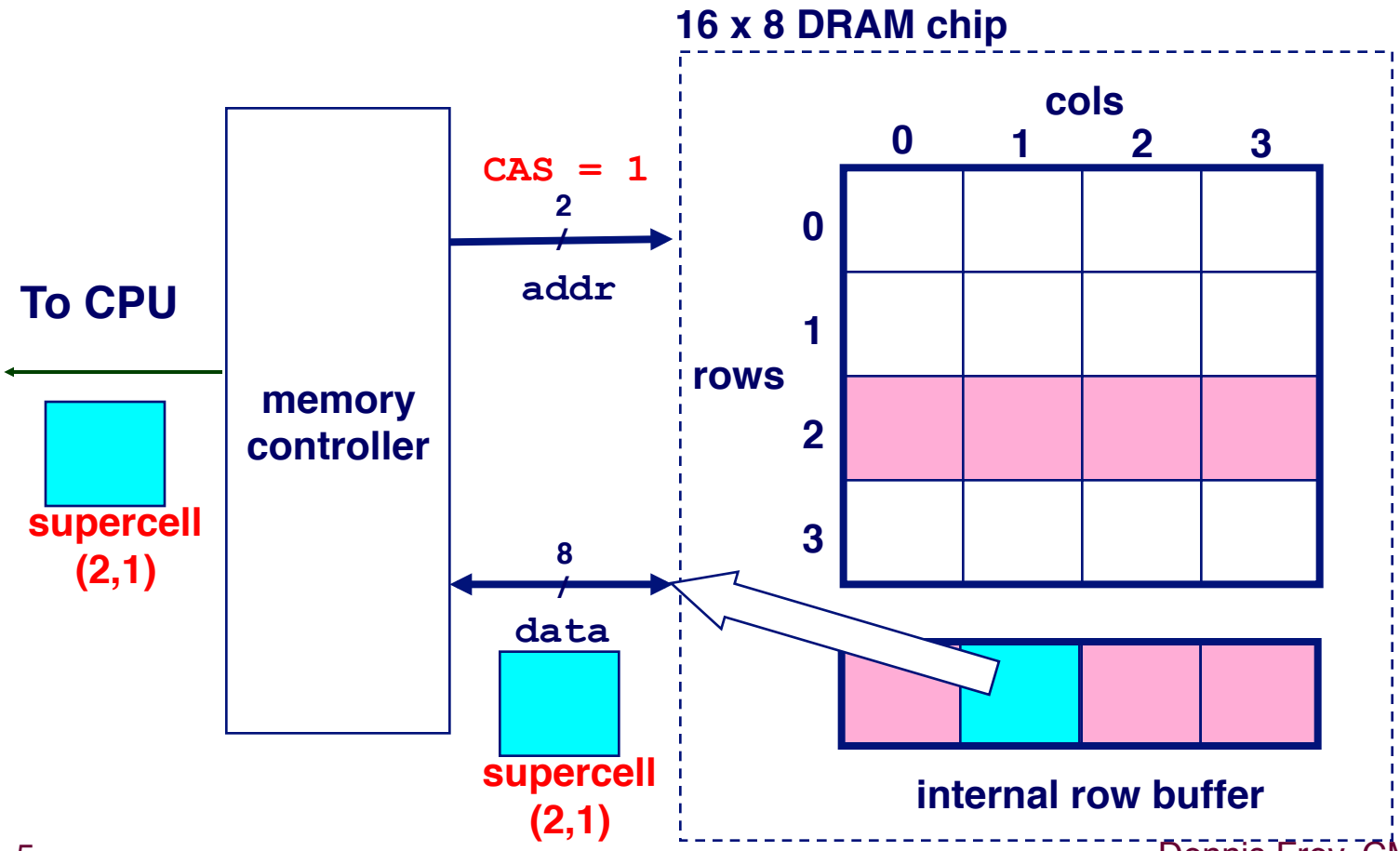
Step 1(b): Row 2 copied from DRAM array to row buffer.



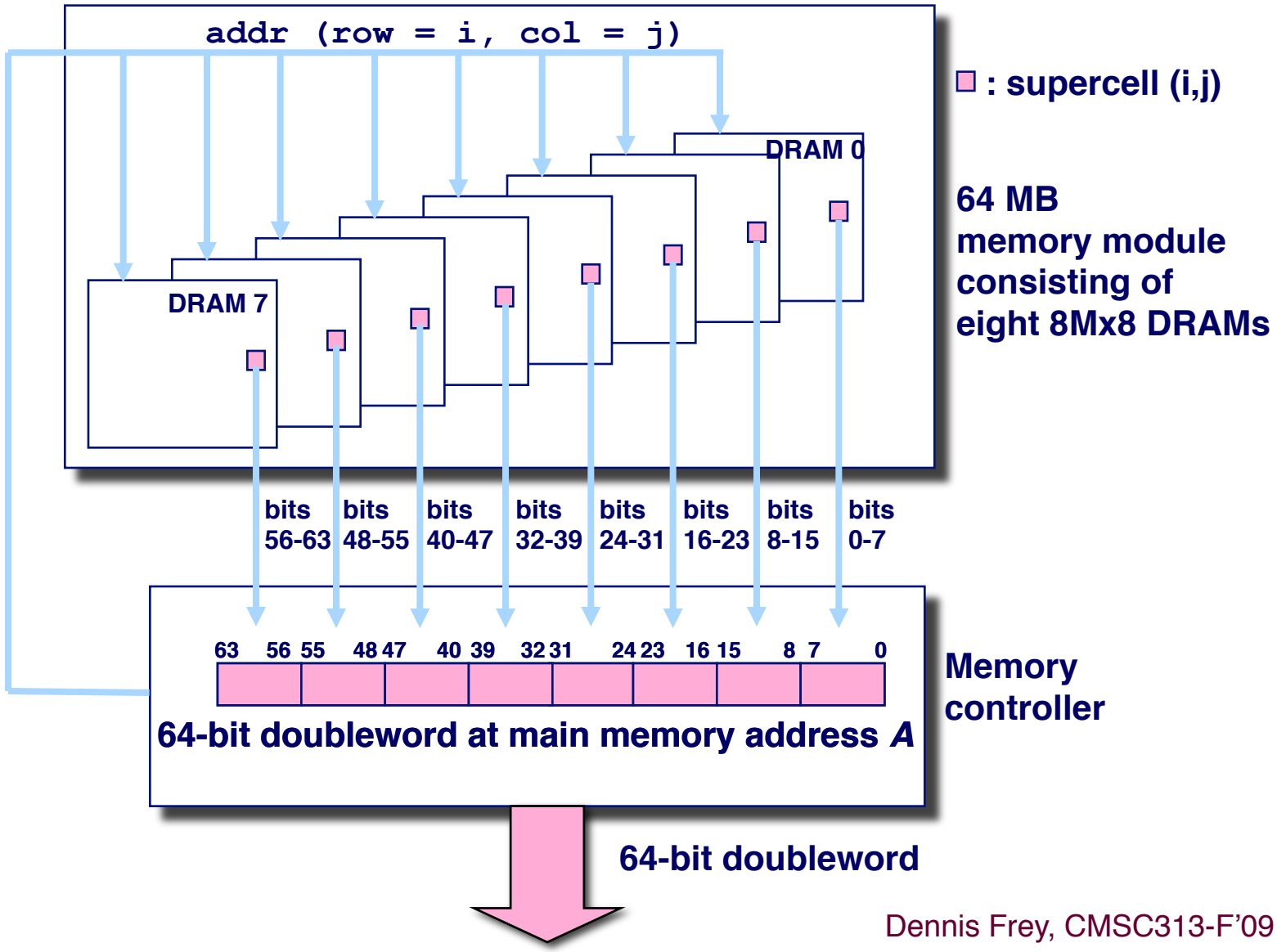
Reading DRAM Supercell (2,1)

Step 2(a): Column access strobe (CAS) selects column 1.

Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.



Memory Modules



Nonvolatile Memories

DRAM and SRAM are volatile memories

- Lose information if powered off.

Nonvolatile memories retain value even if powered off.

- Generic name is read-only memory (ROM).
- Misleading because some ROMs can be read and modified.

Types of ROMs

- Programmable ROM (PROM)
- Eraseable programmable ROM (EPROM)
- Electrically eraseable PROM (EEPROM)
- Flash memory

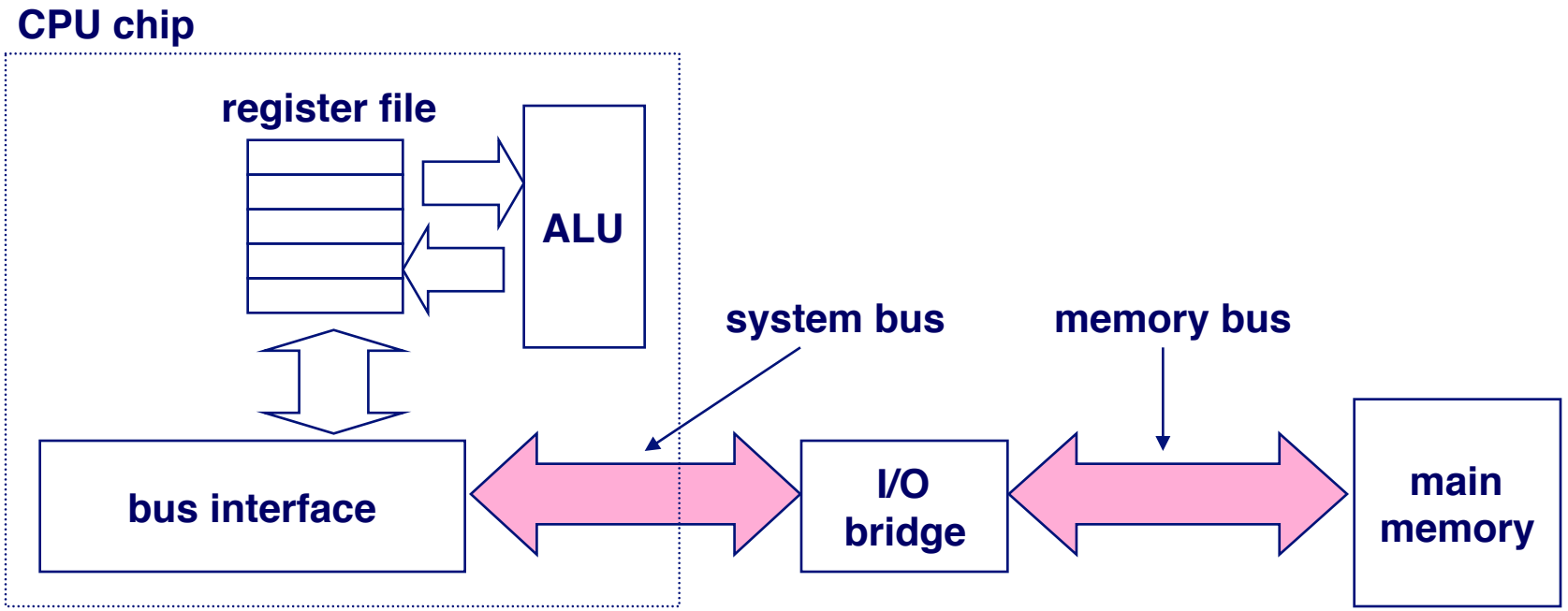
Firmware

- Program stored in a ROM
  - Boot time code, BIOS (basic input/ouput system)
  - graphics cards, disk controllers.

Typical Bus Structure Connecting CPU and Memory

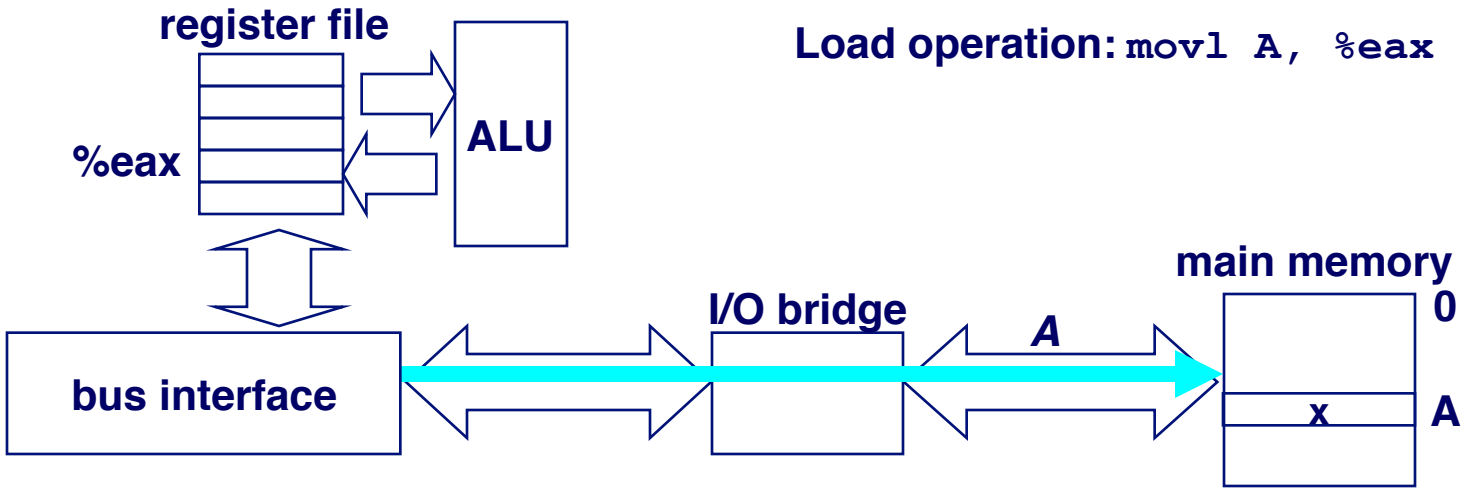
A bus is a collection of parallel wires that carry address, data, and control signals.

Buses are typically shared by multiple devices.



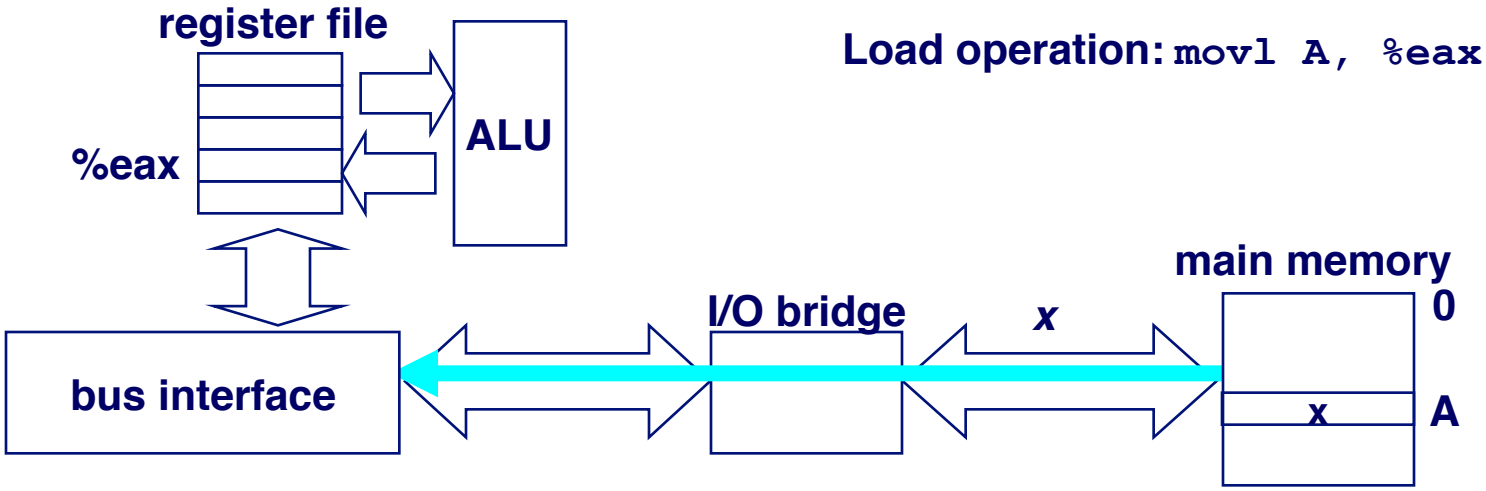
Memory Read Transaction (1)

CPU places address A on the memory bus.



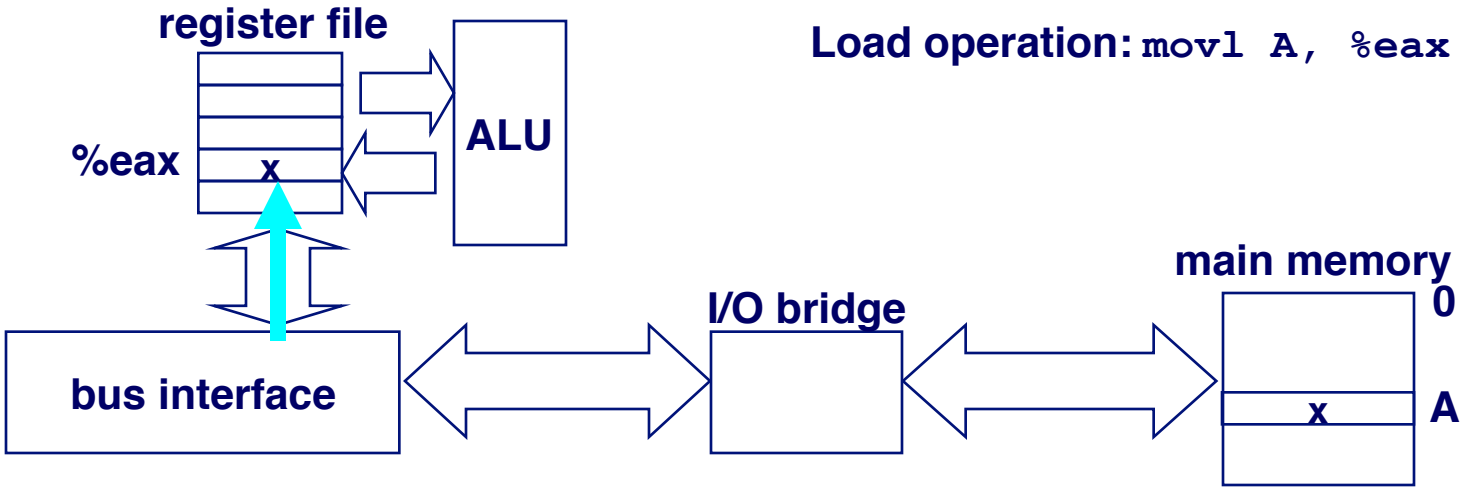
Memory Read Transaction (2)

Main memory reads A from the memory bus, retrieves word x, and places it on the bus.



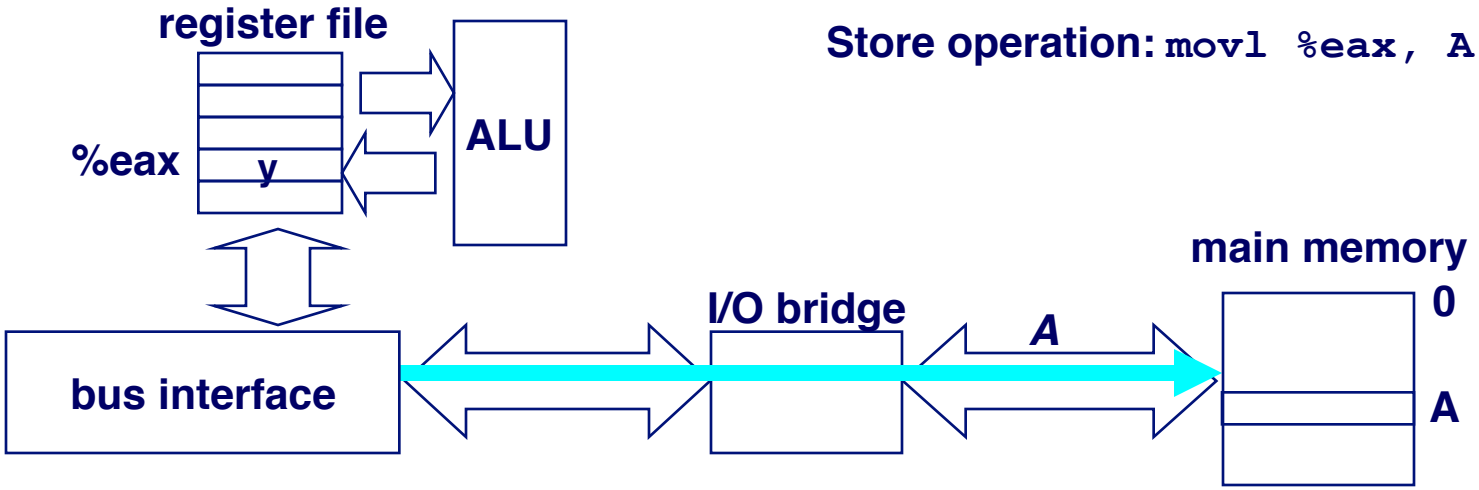
Memory Read Transaction (3)

CPU reads word x from the bus and copies it into register %eax.



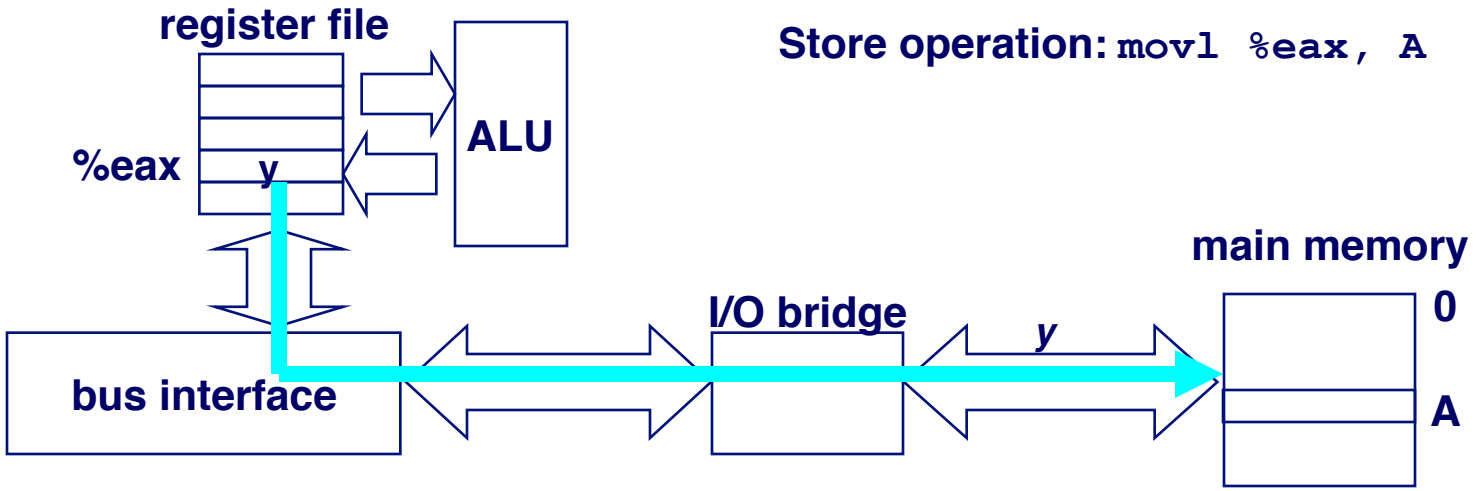
Memory Write Transaction (1)

CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.



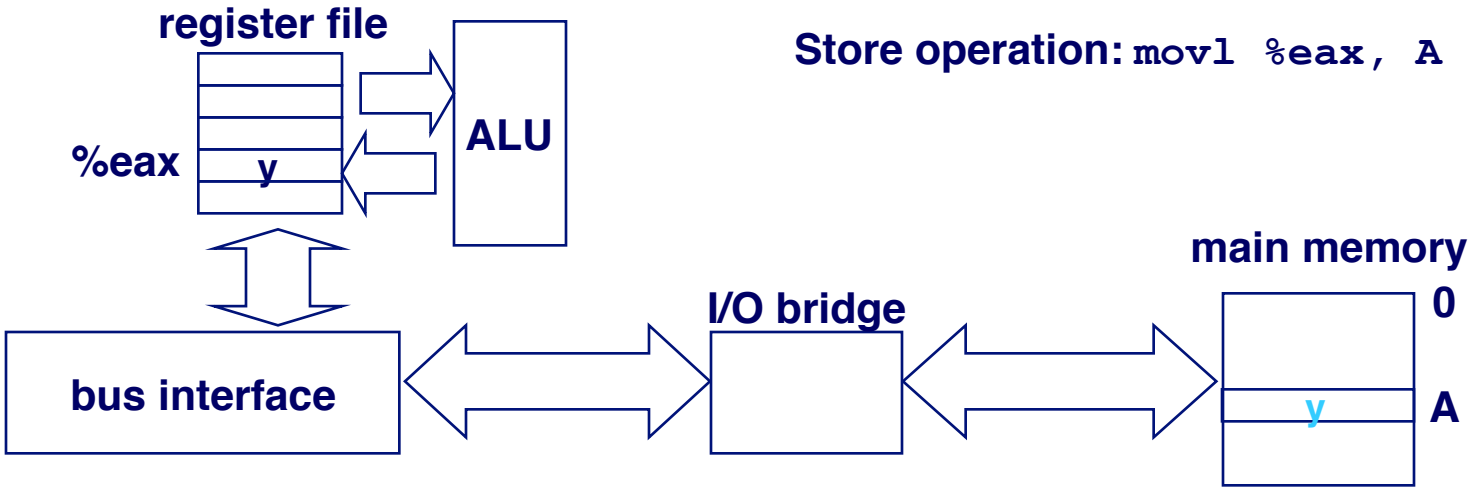
Memory Write Transaction (2)

CPU places data word y on the bus.



Memory Write Transaction (3)

Main memory reads data word y from the bus and stores it at address A.

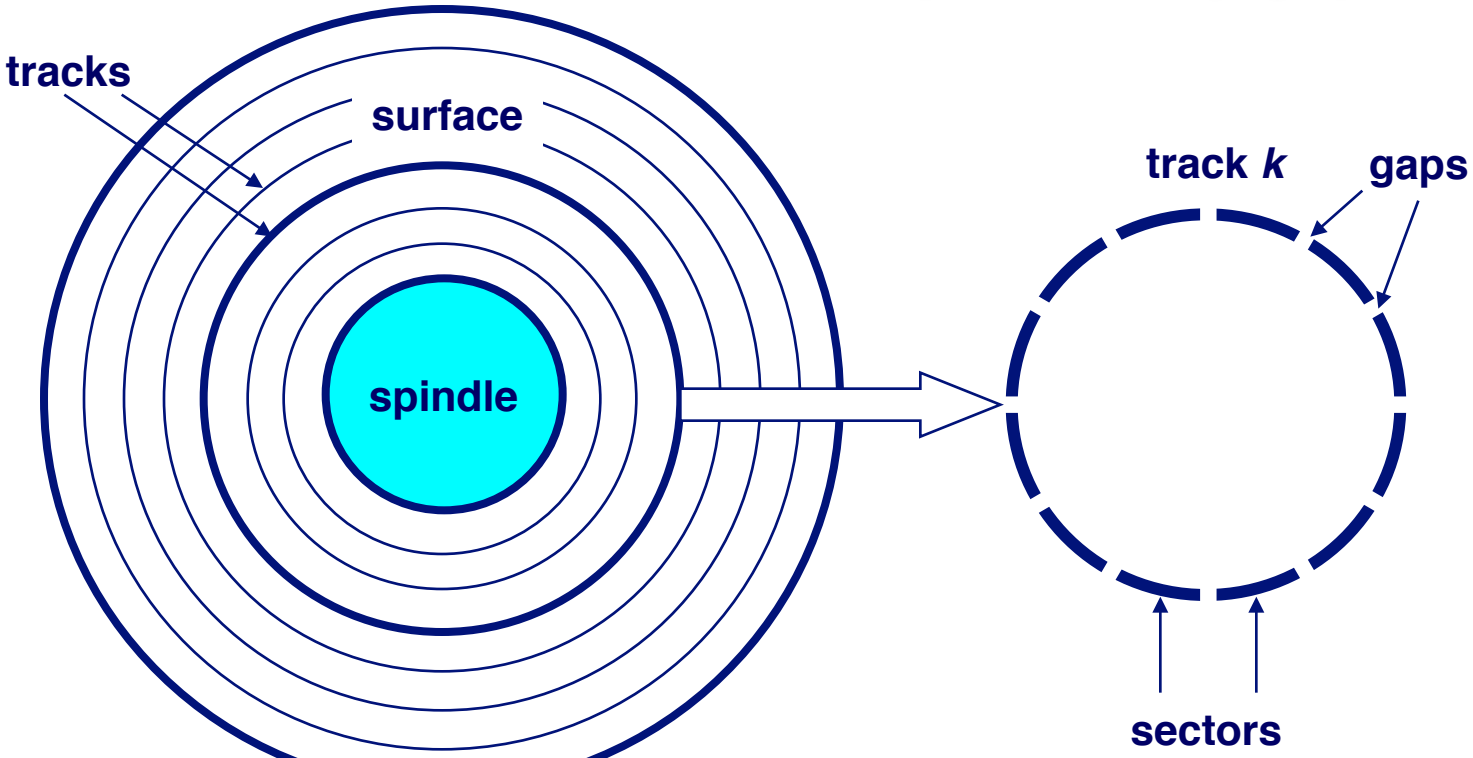


Disk Geometry

Disks consist of platters, each with two surfaces.

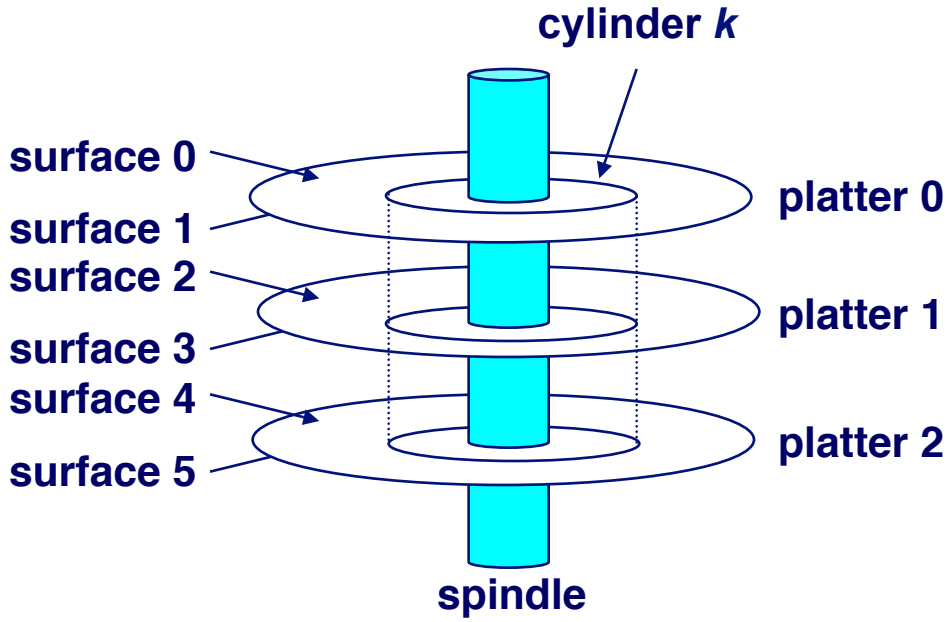
Each surface consists of concentric rings called tracks.

Each track consists of sectors separated by gaps.



Disk Geometry (Muliple-Platter View)

Aligned tracks form a cylinder.



Disk Capacity

Capacity: maximum number of bits that can be stored.

- Vendors express capacity in units of gigabytes (GB), where 1 GB = 10^9 bytes.

Capacity is determined by these technology factors:

- **Recording density** (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.
- **Track density** (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.
- **Areal density** (bits/in^2): product of recording and track density.

Modern disks partition tracks into disjoint subsets called recording zones

- Each track in a zone has the same number of sectors, determined by the circumference of innermost track.
- Each zone has a different number of sectors/track

Computing Disk Capacity

Capacity = (# bytes/sector) x (avg. # sectors/track) x (# tracks/surface) x (# surfaces/platter) x (# platters/disk)

Example:

- **512 bytes/sector**
- **300 sectors/track (on average)**
- **20,000 tracks/surface**
- **2 surfaces/platter**
- **5 platters/disk**

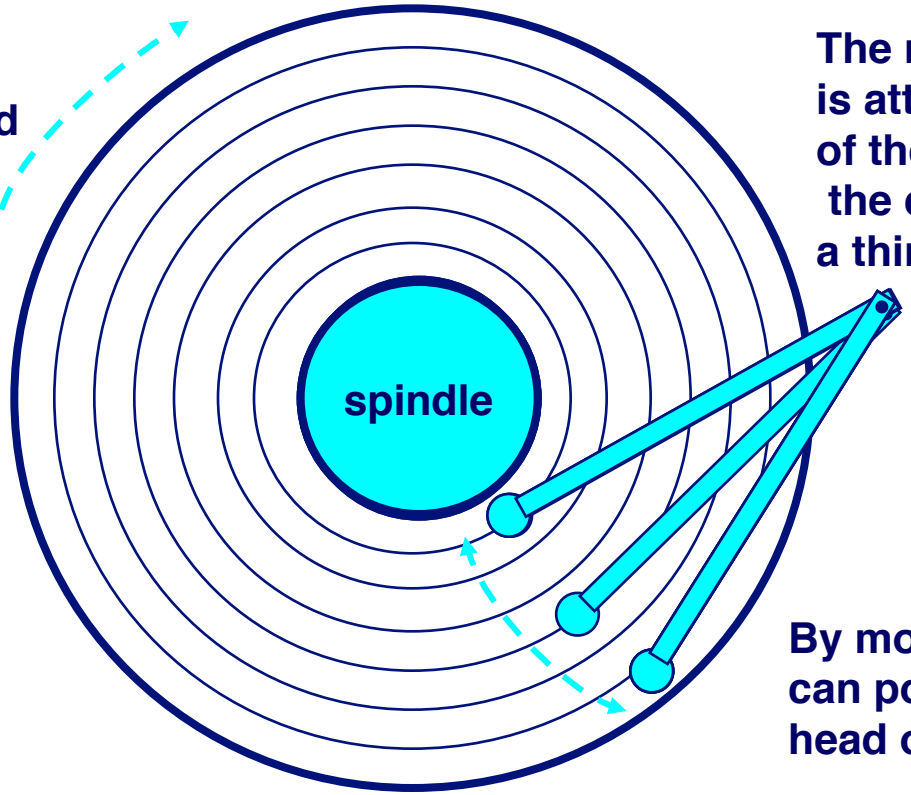
Capacity = 512 x 300 x 20000 x 2 x 5

= 30,720,000,000

= 30.72 GB

Disk Operation (Single-Platter View)

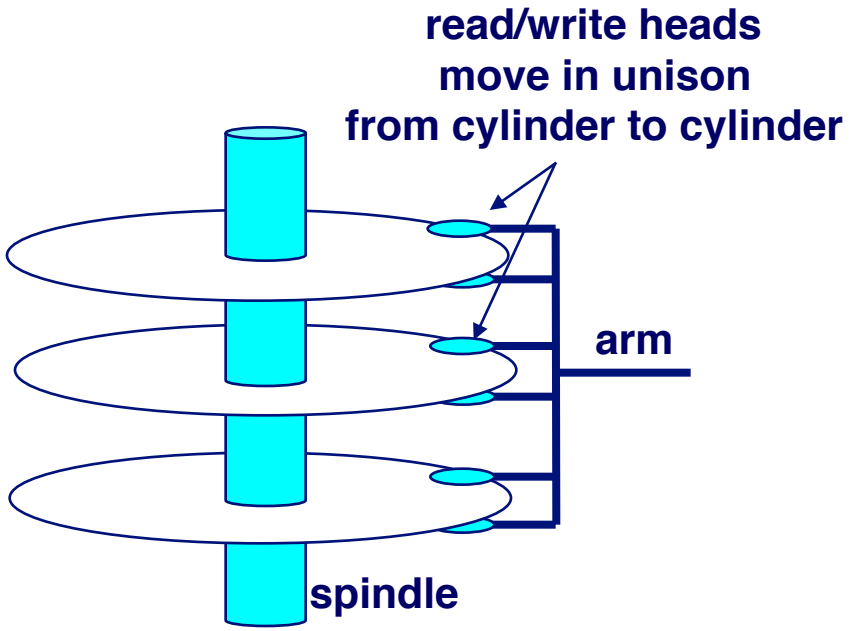
The disk surface spins at a fixed rotational rate



The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.

Disk Operation (Multi-Platter View)



Disk Access Time

Average time to access some target sector approximated by :

- $T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$

Seek time ($T_{avg\ seek}$)

- Time to position heads over cylinder containing target sector.
- Typical $T_{avg\ seek}$ = 9 ms

Rotational latency ($T_{avg\ rotation}$)

- Time waiting for first bit of target sector to pass under read/write head.
- $T_{avg\ rotation}$ = 1/2 x 1/RPMs x 60 sec/1 min

Transfer time ($T_{avg\ transfer}$)

- Time to read the bits in the target sector.
- $T_{avg\ transfer}$ = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min.

Disk Access Time Example

Given:

- Rotational rate = 7,200 RPM
- Average seek time = 9 ms.
- Avg # sectors/track = 400.

Derived:

- Tavg rotation = 1/2 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms.
- Tavg transfer = 60/7200 RPM x 1/400 secs/track x 1000 ms/sec = 0.02 ms
- Taccess = 9 ms + 4 ms + 0.02 ms

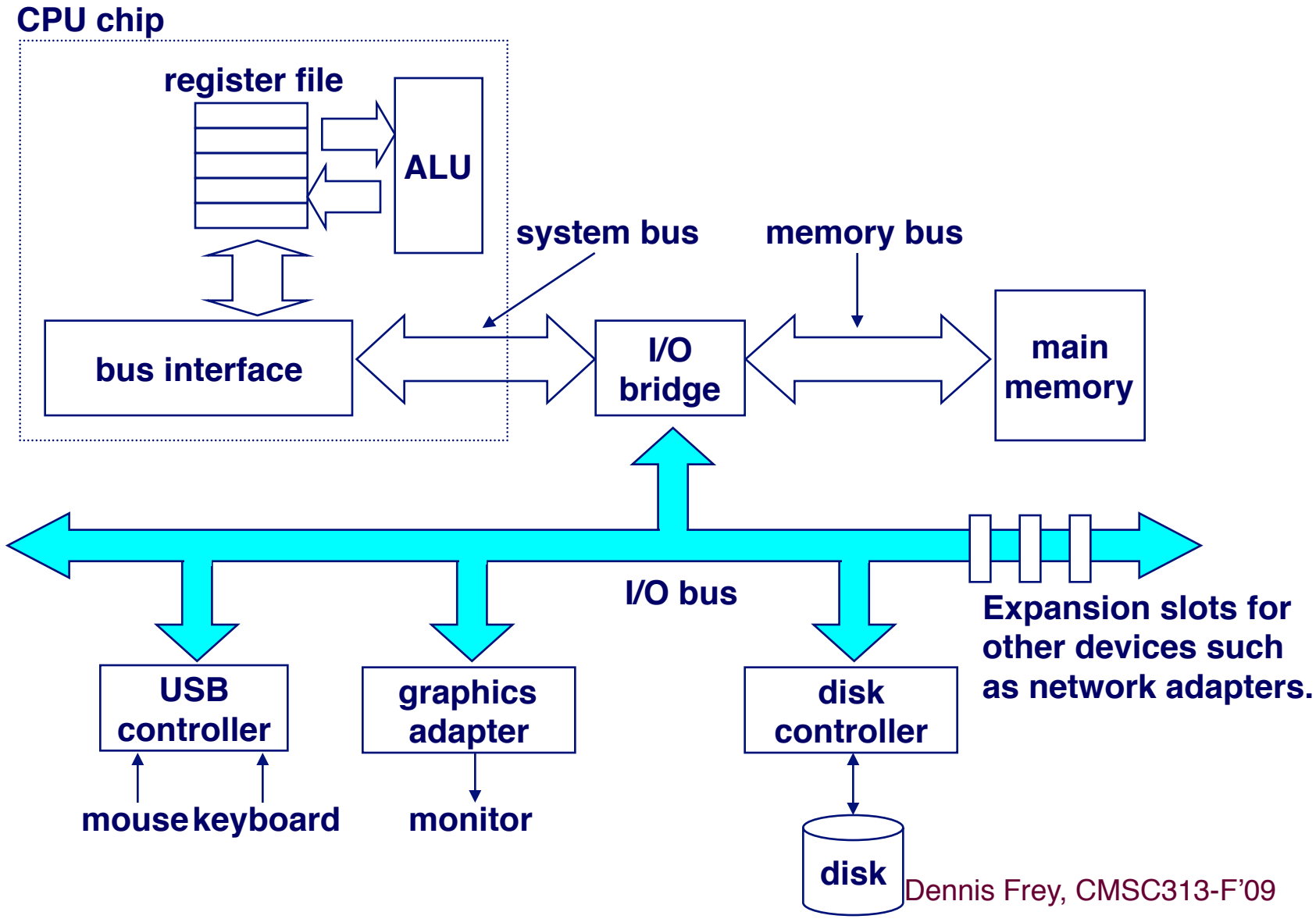
Important points:

- Access time dominated by seek time and rotational latency.
- First bit in a sector is the most expensive, the rest are free.
- SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
  - Disk is about 40,000 times slower than SRAM,
  - 2,500 times slower then DRAM.

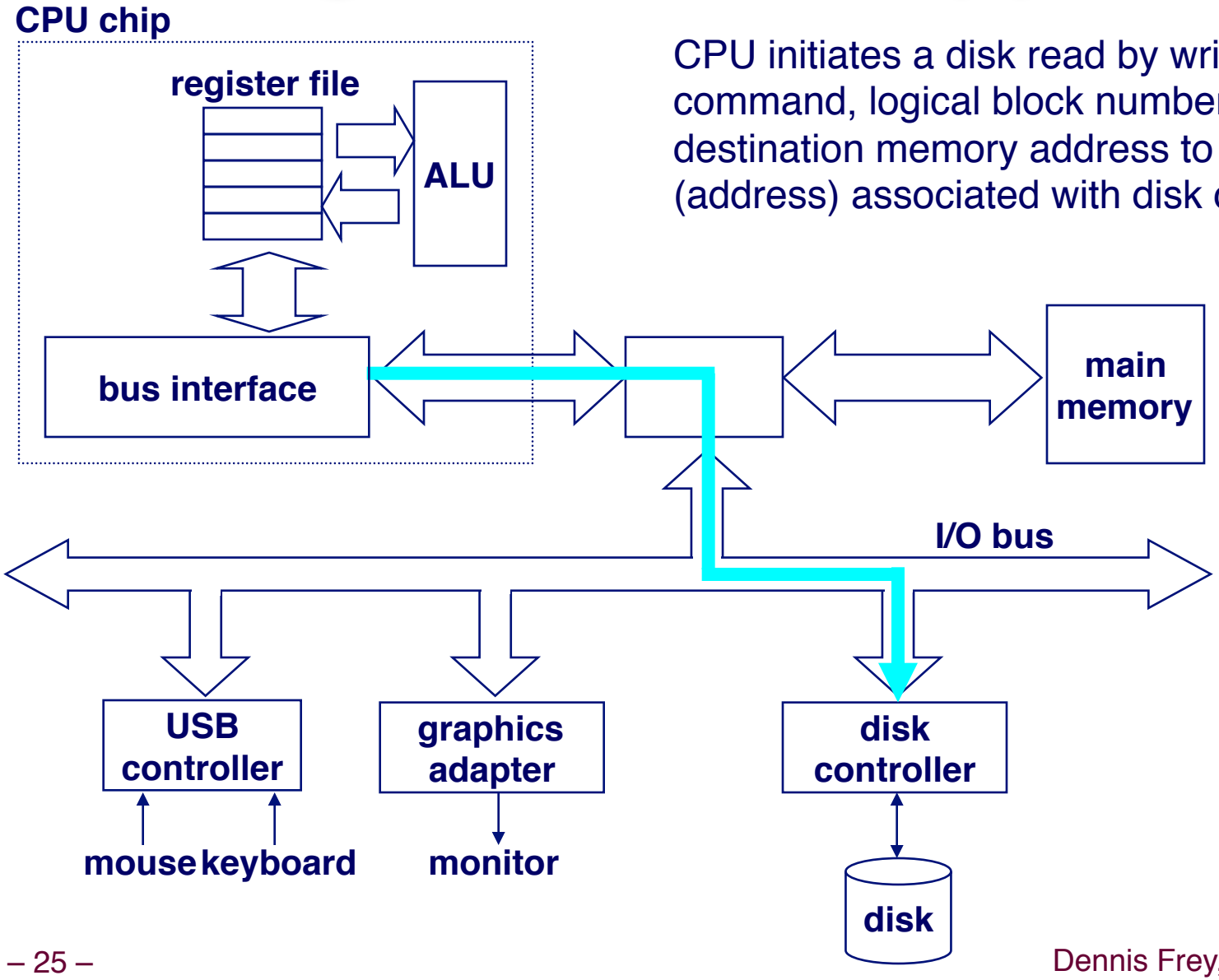
Logical Disk Blocks

- Modern disks present a simpler abstract view of the complex sector geometry:
  - The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2, ...)
- Mapping between logical blocks and actual (physical) sectors
  - Maintained by hardware/firmware device called disk controller.
  - Converts requests for logical blocks into (surface,track,sector) triples.
- Allows controller to set aside spare cylinders for each zone.
  - Accounts for the difference in "formatted capacity" and "maximum capacity".

I/O Bus

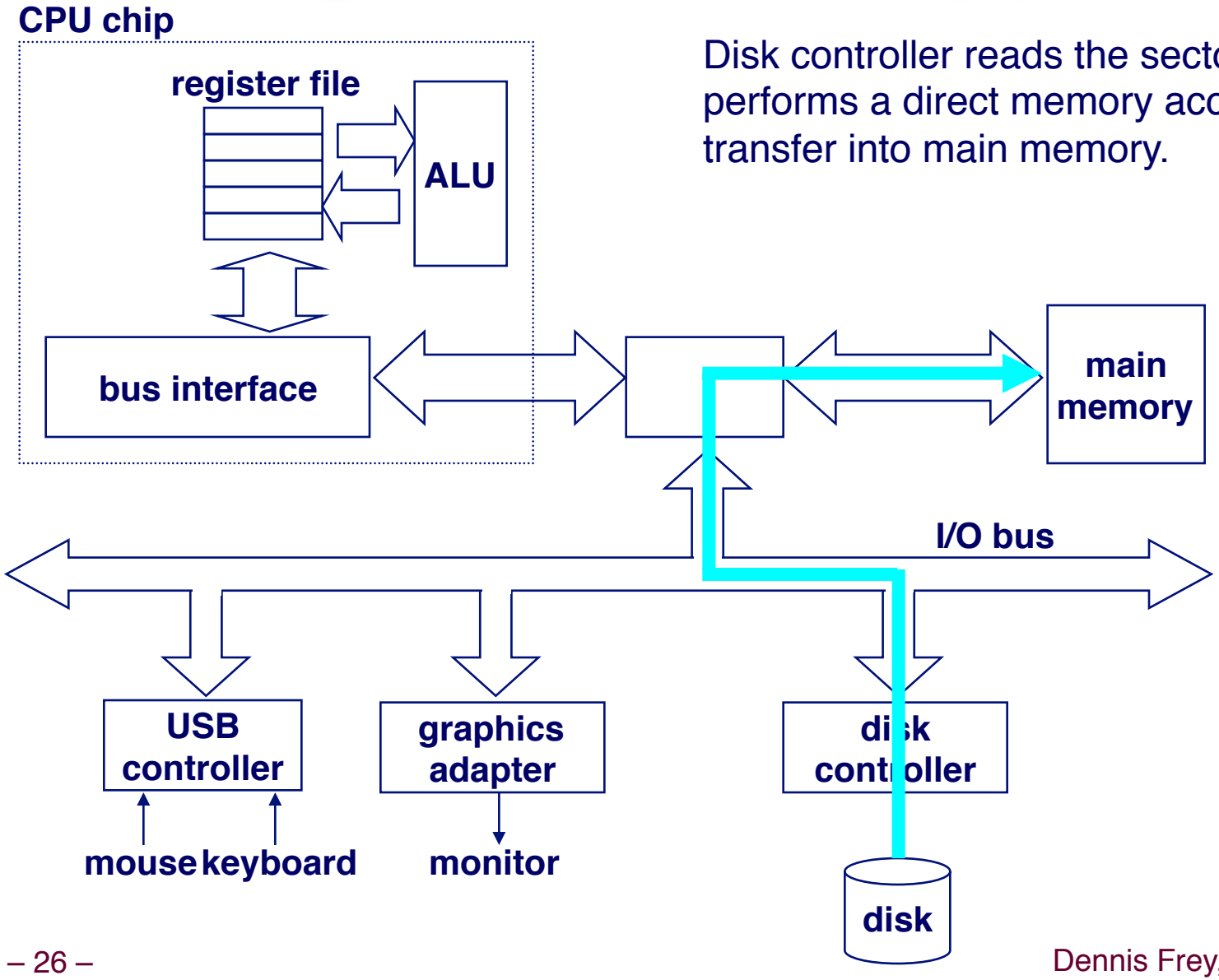

CPU chip
register file
ALU
system bus
memory bus
bus interface
I/O bridge
main memory
I/O bus
Expansion slots for other devices such as network adapters.
USB controller
graphics adapter
disk controller
mouse
keyboard
monitor
disk

Reading a Disk Sector (1)



CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

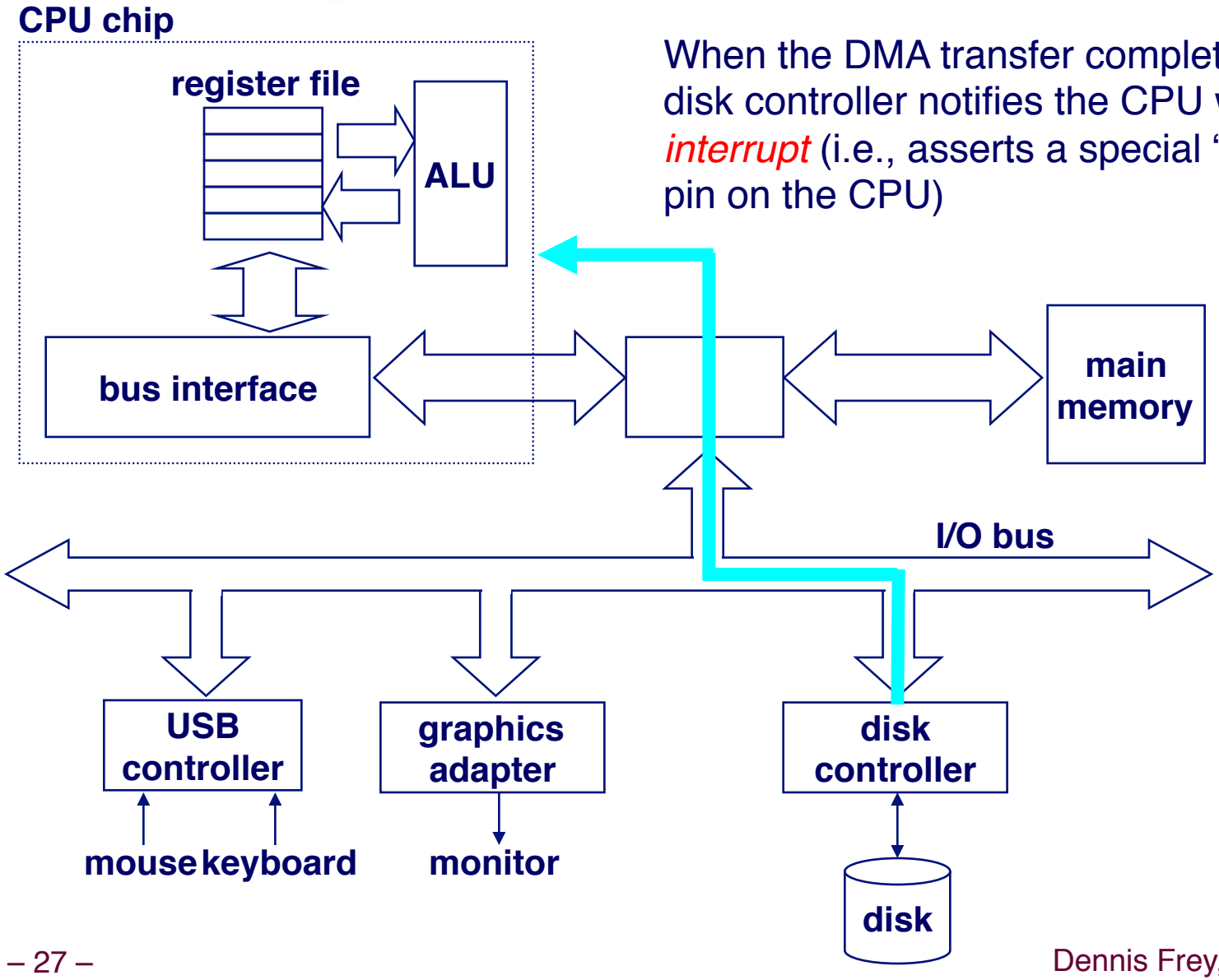
Reading a Disk Sector (2)



Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

Reading a Disk Sector (3)

When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special "interrupt" pin on the CPU)



CACHING

Locality

Principle of Locality:

- **Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.**
- **Temporal locality: Recently referenced items are likely to be referenced in the near future.**
- **Spatial locality: Items with nearby addresses tend to be referenced close together in time.**

Locality Example:

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data**
  - Reference array elements in succession (stride-1 reference pattern): **Spatial locality**
  - Reference `sum` each iteration: **Temporal locality**
- **Instructions**
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

Locality Example

Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

Question: Does this function have good locality?

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum
}
```

Locality Example

Question: Does this function have good locality?

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum
}
```

Locality Example

Question: Can you permute the loops so that the function scans the 3-d array `a[]` with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sumarray3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum
}
```

Memory Hierarchies

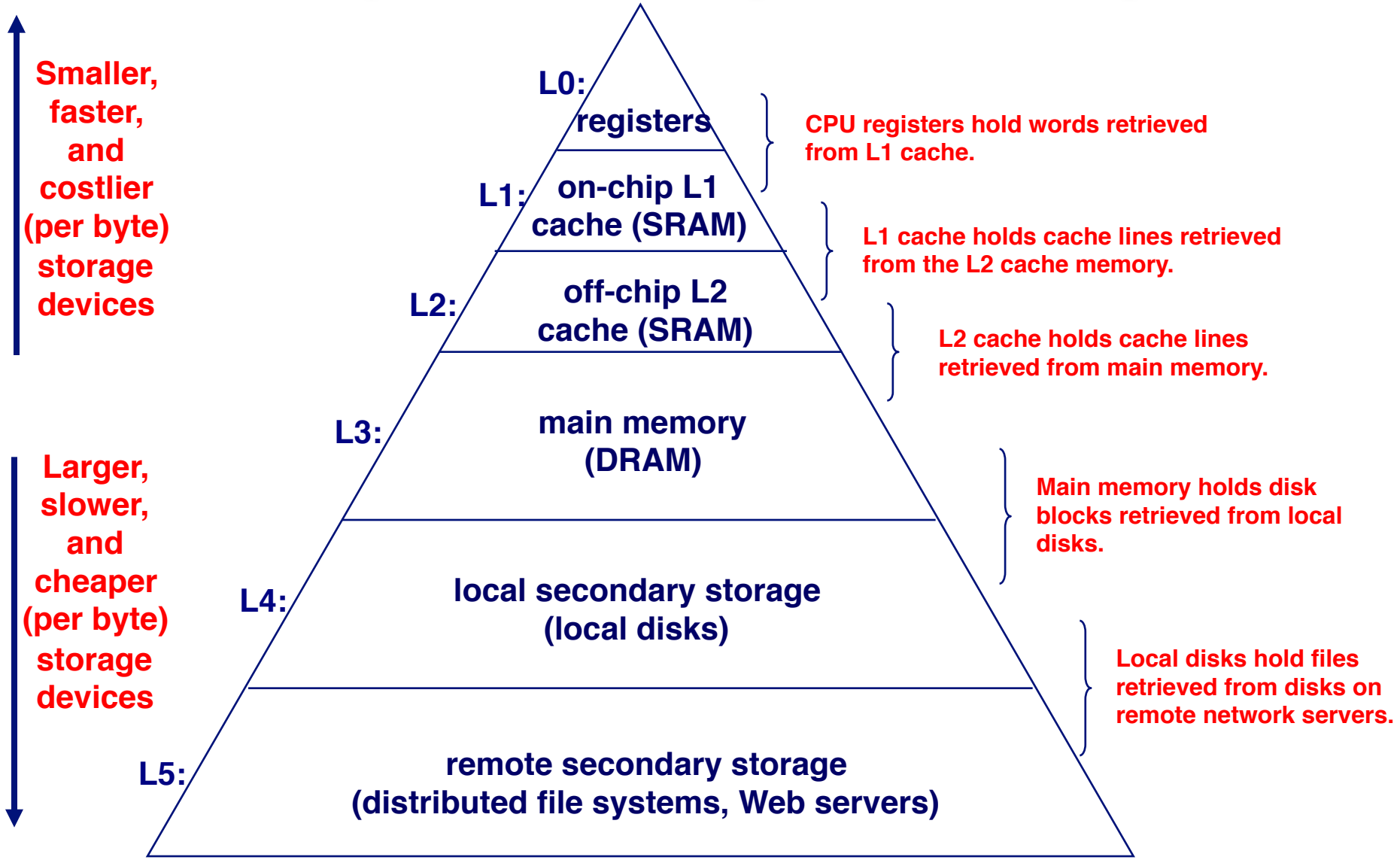
Some fundamental and enduring properties of hardware and software:

- **Fast storage technologies cost more per byte and have less capacity.**
- **The gap between CPU and main memory speed is widening.**
- **Well-written programs tend to exhibit good locality.**

These fundamental properties complement each other beautifully.

They suggest an approach for organizing memory and storage systems known as a memory hierarchy.

An Example Memory Hierarchy



Caches

Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

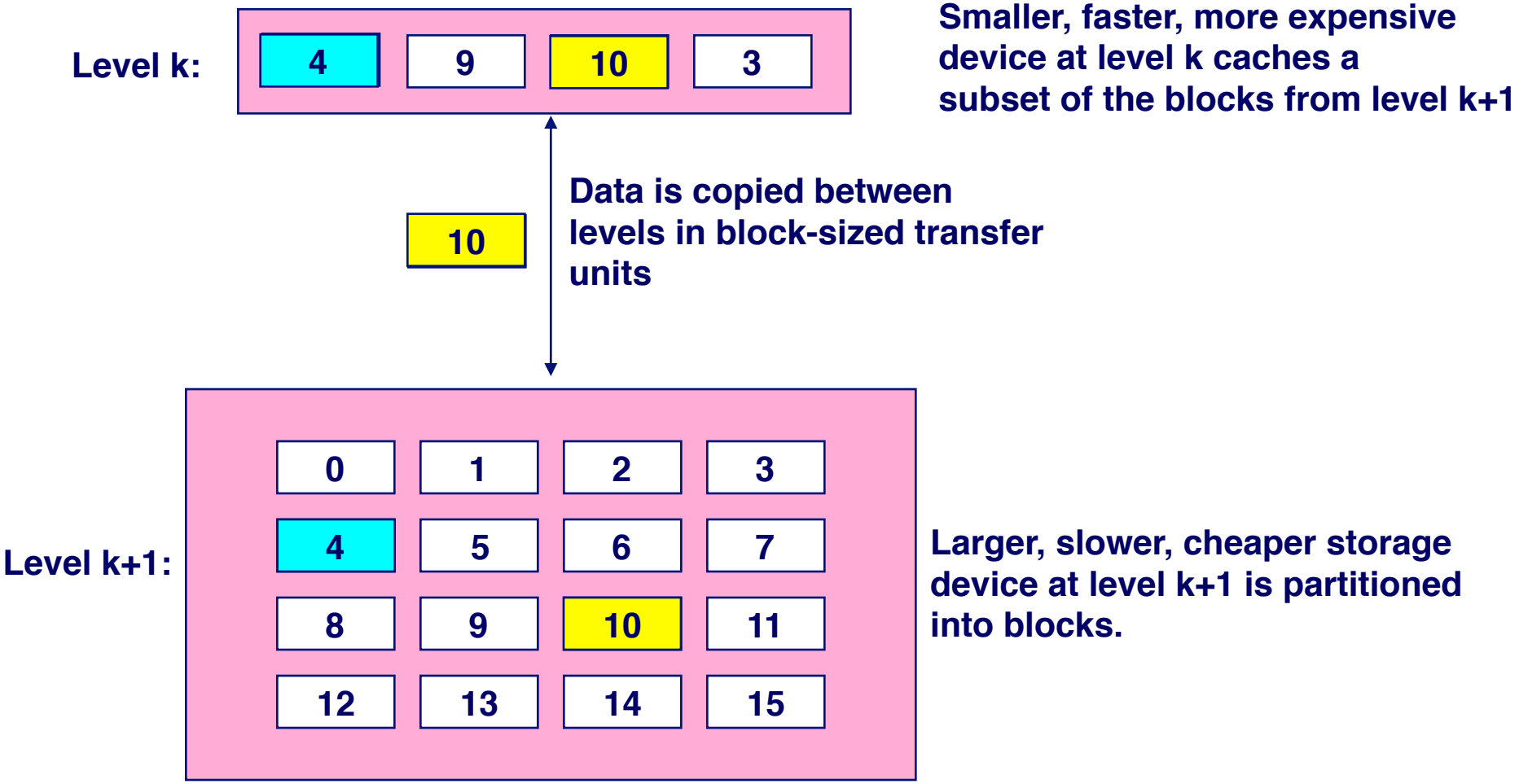
Fundamental idea of a memory hierarchy:

- For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.

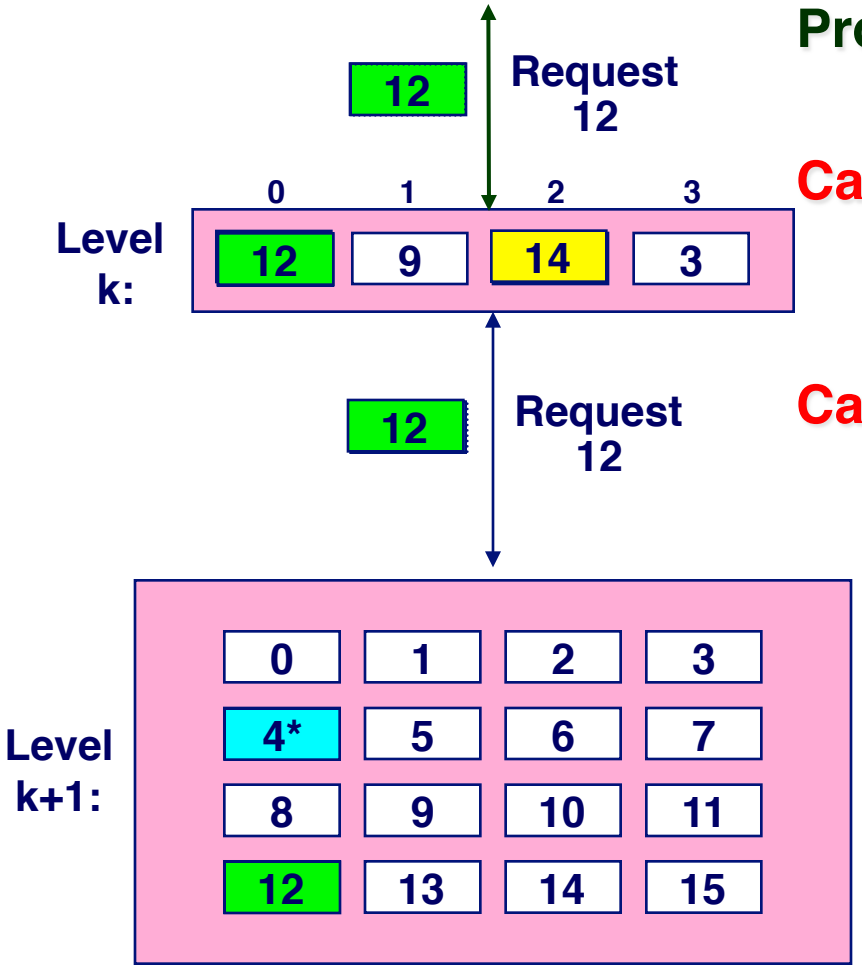
Why do memory hierarchies work?

- Programs tend to access the data at level k more often than they access the data at level k+1.
- Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
- Net effect: A large pool of memory that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Caching in a Memory Hierarchy



General Caching Concepts



Program needs object d, which is stored in some block b.

Cache hit

- Program finds b in the cache at level k. E.g., block 14.

Cache miss

- b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
- If level k cache is full, then some current block must be replaced (evicted). Which one is the "victim"?
  - **Placement policy:** where can the new block go? E.g., b mod 4
  - **Replacement policy:** which block should be evicted? E.g., LRU

General Caching Concepts

Types of cache misses:

- **Cold (compulsary) miss**
  - Cold misses occur because the cache is empty.
- **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
  - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k+1.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
  - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

Examples of Caching in the Hierarchy

Cache Type	What Cached	Where Cached	Latency (cycles)	Managed By
Registers	4-byte word	CPU registers	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	32-byte block	On-Chip L1	1	Hardware
L2 cache	32-byte block	Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	Main memory	100	Hardware +OS
Buffer cache	Parts of files	Main memory	100	OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

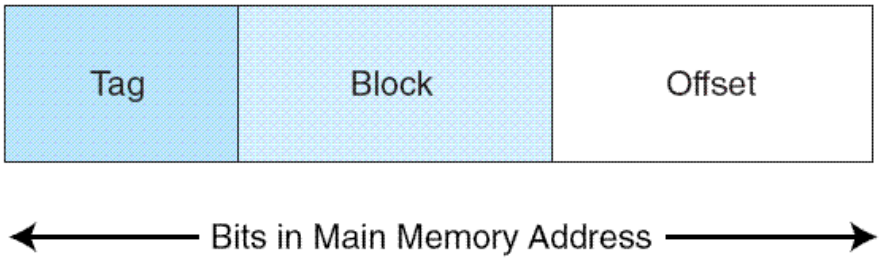
DIRECT MAPPING

6.4 Cache Memory

- The purpose of cache memory is to speed up accesses by storing recently used data closer to the CPU, instead of storing it in main memory.
- Although cache is much smaller than main memory, its access time is a fraction of that of main memory.
- Unlike main memory, which is accessed by address, cache is typically accessed by content; hence, it is often called *content addressable memory*.
- Because of this, a single large cache memory isn't always desirable-- it takes longer to search.

6.4 Cache Memory

- The "content" that is addressed in content addressable cache memory is a subset of the bits of a main memory address called a *field*.
  - Many blocks of main memory map to a single block of cache. A *tag* field in the cache block distinguishes one cached memory block from another.
  - A *valid bit* indicates whether the cache block is being used.
  - An *offset field* points to the desired data in the block.



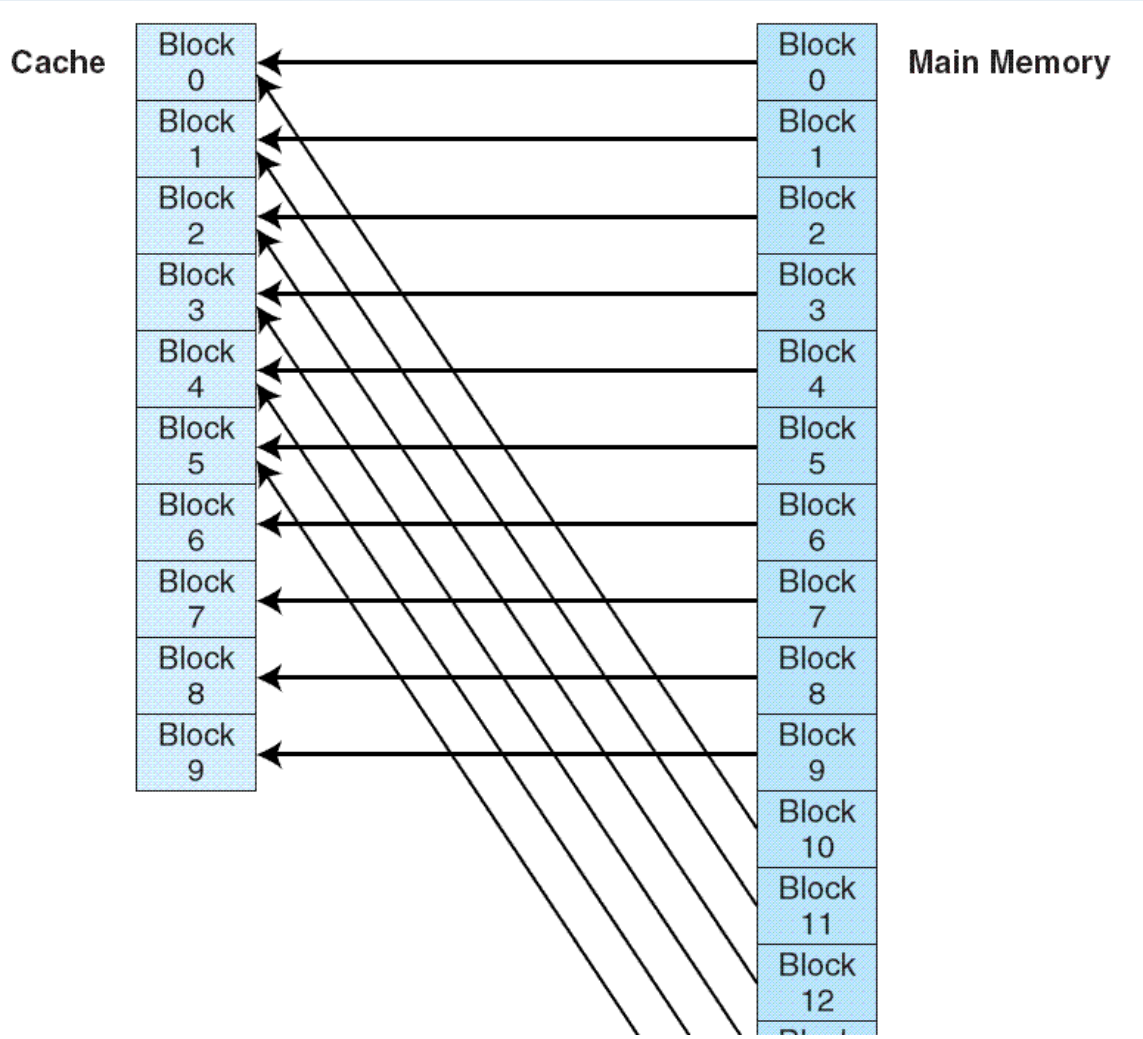
6.4 Cache Memory

- The simplest cache mapping scheme is *direct mapped cache*.
- In a direct mapped cache consisting of *N* blocks of cache, block *X* of main memory maps to cache block *Y* = *X* mod *N*.
- Thus, if we have 10 blocks of cache, block 7 of cache may hold blocks 7, 17, 27, 37, . . . of main memory.

The next slide illustrates this mapping.

6.4 Cache Memory

- With direct mapped cache consisting of *N* blocks of cache, block *X* of main memory maps to cache block *Y* = *X* mod *N*.

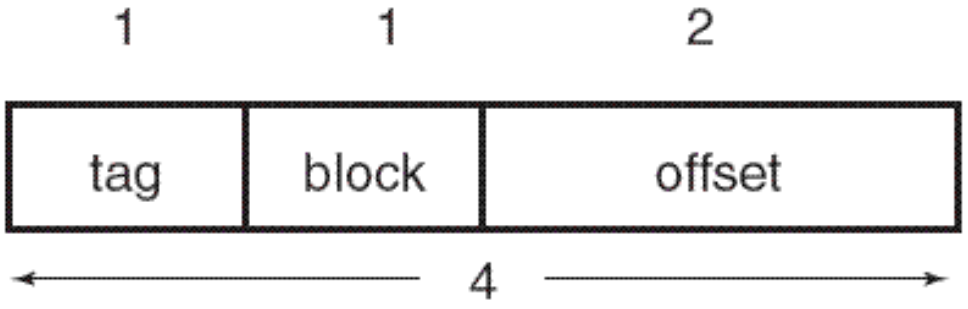


6.4 Cache Memory

- EXAMPLE 6.1 Consider a word-addressable main memory consisting of four blocks, and a cache with two blocks, where each block is 4 words.
- This means Block 0 and 2 of main memory map to Block 0 of cache, and Blocks 1 and 3 of main memory map to Block 1 of cache.
- Using the tag, block, and offset fields, we can see how main memory maps to cache as follows.

6.4 Cache Memory

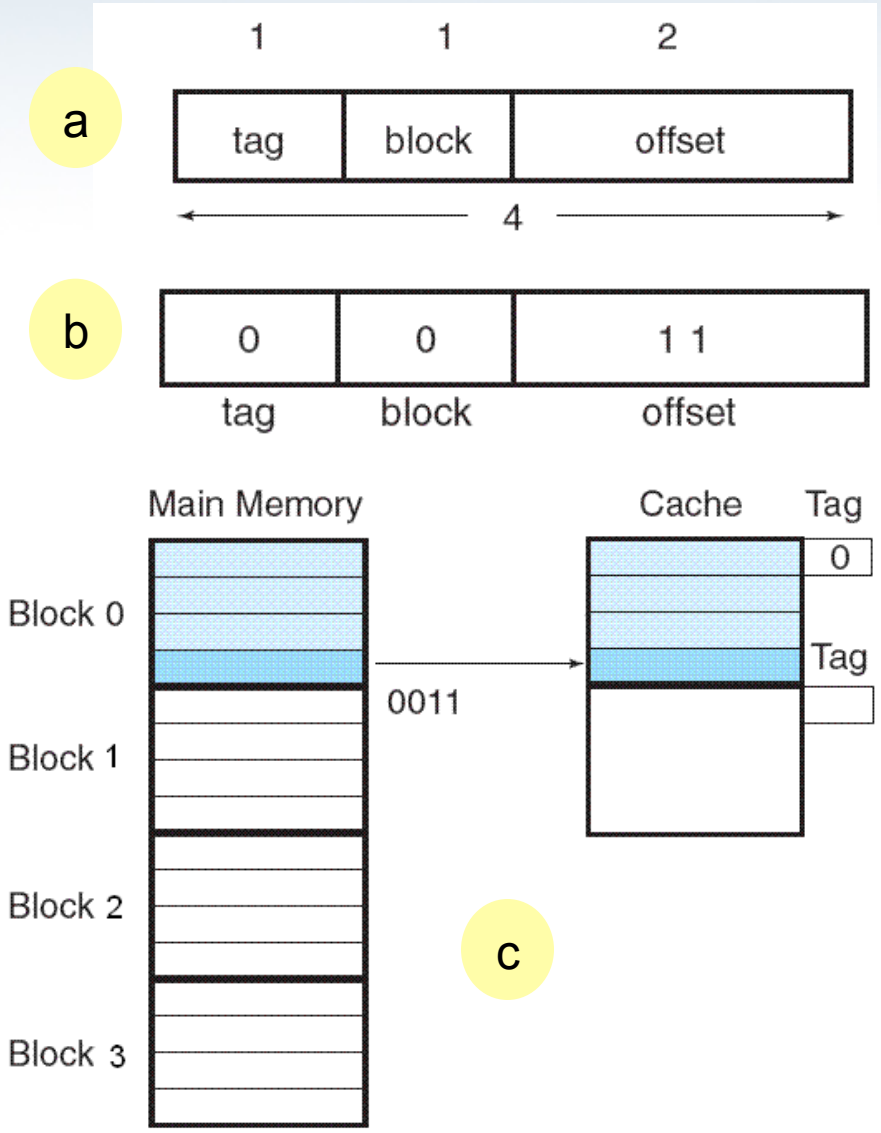
- EXAMPLE 6.1 Consider a word-addressable main memory consisting of four blocks, and a cache with two blocks, where each block is 4 words.
  - First, we need to determine the address format for mapping. Each block is 4 words, so the offset field must contain 2 bits; there are 2 blocks in cache, so the block field must contain 1 bit; this leaves 1 bit for the tag (as a main memory address has 4 bits because there are a total of 2^4=16 words).



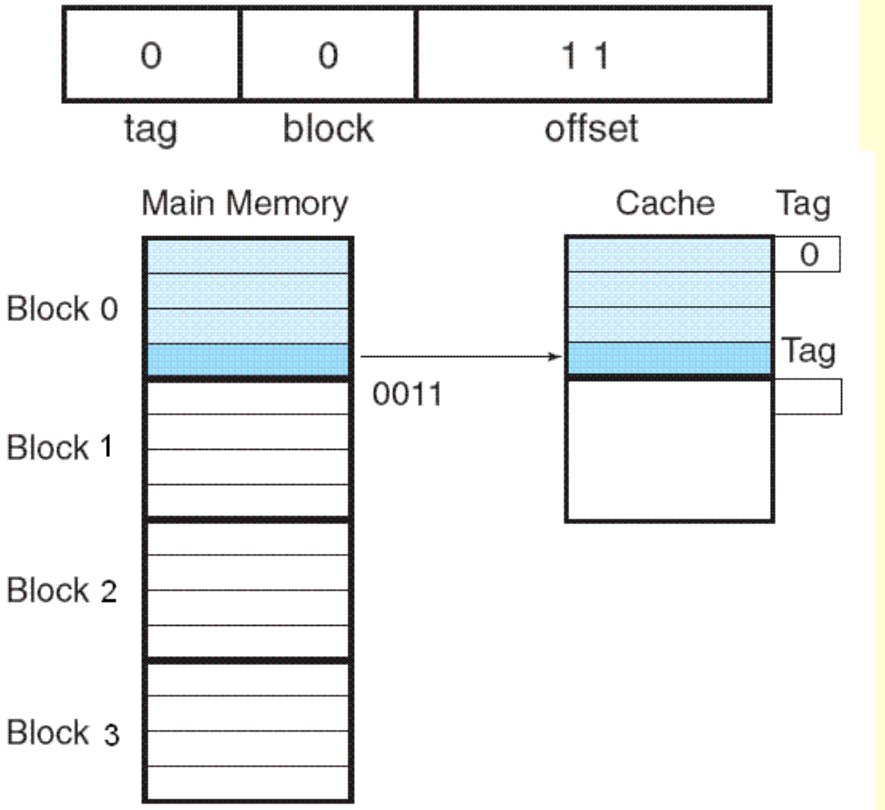
6.4 Cache Memory

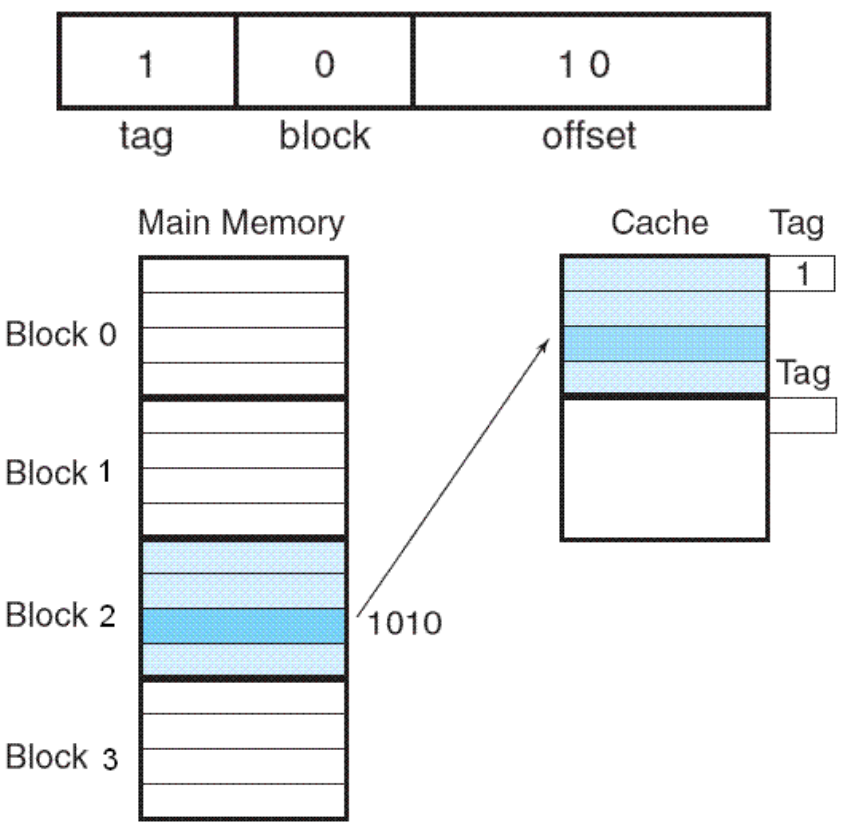
- EXAMPLE 6.1 Cont'd
  - Suppose we need to access main memory address 3_{16} (0011 in binary). If we partition 0011 using the address format from Figure a, we get Figure b.
  - Thus, the main memory address 0011 maps to cache block 0.
  - Figure c shows this mapping, along with the tag that is also stored with the data.

The next slide illustrates another mapping.



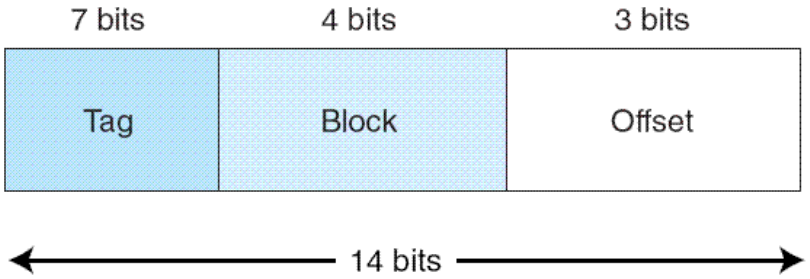
6.4 Cache Memory





6.4 Cache Memory

- EXAMPLE 6.2 Assume a byte-addressable memory consists of 2^{14} bytes, cache has 16 blocks, and each block has 8 bytes.
  - The number of memory blocks are: $\frac{2^{14}}{2^3} = 2^{11}$
  - Each main memory address requires14 bits. Of this 14-bit address field, the rightmost 3 bits reflect the offset field
  - We need 4 bits to select a specific block in cache, so the block field consists of the middle 4 bits.
  - The remaining 7 bits make up the tag field.



6.4 Cache Memory

- In summary, direct mapped cache maps main memory blocks in a modular fashion to cache blocks. The mapping depends on:
- The number of bits in the main memory address (how many addresses exist in main memory)
- The number of blocks are in cache (which determines the size of the block field)
- How many addresses (either bytes or words) are in a block (which determines the size of the offset field)

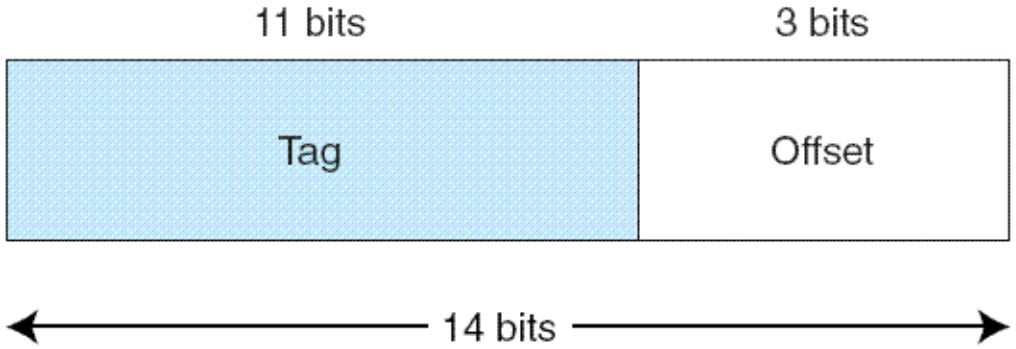
FULLY ASSOCIATIVE MAPPING

6.4 Cache Memory

- Suppose instead of placing memory blocks in specific cache locations based on memory address, we could allow a block to go anywhere in cache.
- In this way, cache would have to fill up before any blocks are evicted.
- This is how *fully associative* cache works.
- A memory address is partitioned into only two fields: the tag and the word.

6.4 Cache Memory

- Suppose, as before, we have 14-bit memory addresses and a cache with 16 blocks, each block of size 8. The field format of a memory reference is:



- When the cache is searched, all tags are searched in parallel to retrieve the data quickly.
- This requires special, costly hardware.

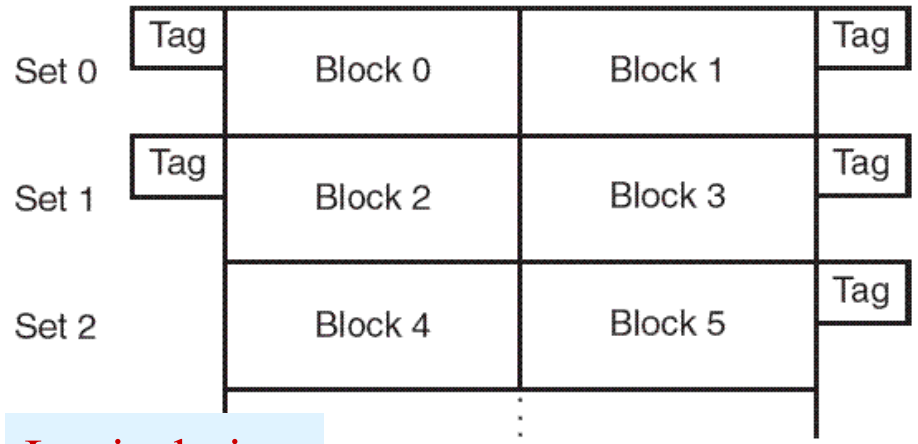
SET ASSOCIATIVE MAPPING

6.4 Cache Memory

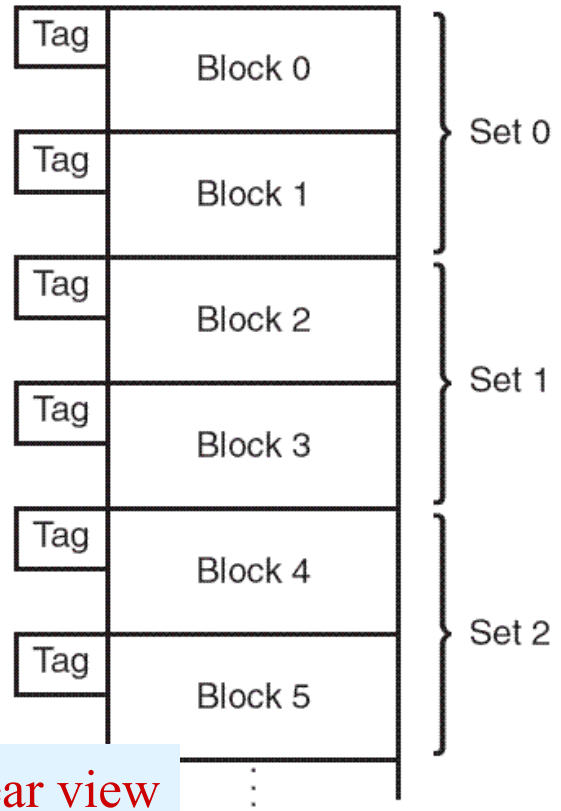
- Set associative cache combines the ideas of direct mapped cache and fully associative cache.
- An *N*-way set associative cache mapping is like direct mapped cache in that a memory reference maps to a particular location in cache.
- Unlike direct mapped cache, a memory reference maps to a set of several cache blocks, similar to the way in which fully associative cache works.
- Instead of mapping anywhere in the entire cache, a memory reference can map only to the subset of cache slots.

6.4 Cache Memory

- The number of cache blocks per set in set associative cache varies according to overall system design.
  - For example, a 2-way set associative cache can be conceptualized as shown in the schematic below.
  - Each set contains two different memory blocks.



Logical view



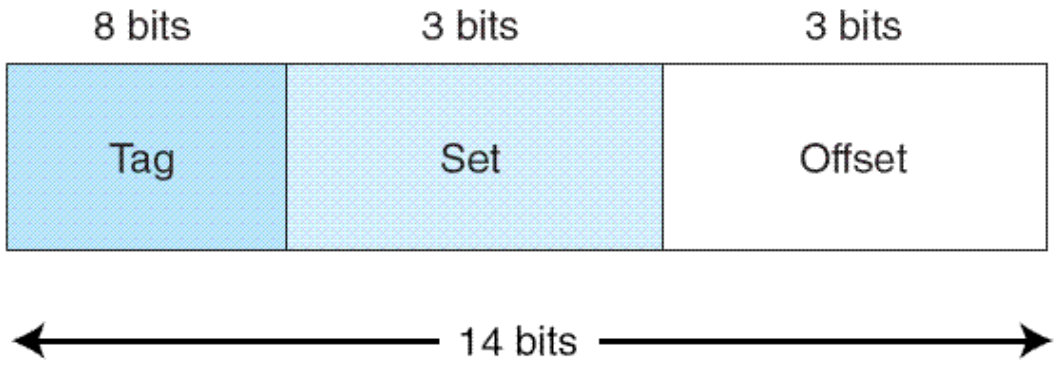
Linear view

6.4 Cache Memory

- In set associative cache mapping, a memory reference is divided into three fields: tag, set, and offset.
- As with direct-mapped cache, the offset field chooses the word within the cache block, and the tag field uniquely identifies the memory address.
- The set field determines the set to which the memory block maps.

6.4 Cache Memory

- EXAMPLE 6.5 Suppose we are using 2-way set associative mapping with a word-addressable main memory of 2^{14} words and a cache with 16 blocks, where each block contains 8 words.
  - Cache has a total of 16 blocks, and each set has 2 blocks, then there are 8 sets in cache.
  - Thus, the set field is 3 bits, the offset field is 3 bits, and the tag field is 8 bits.



CACHING POLICIES

- **Cache replacement policy**
  - For fully associative and set associative mapping
  - Which cache block gets kicked out?
  - Some schemes: first-in first-out, least recently used, ...

- **Cache write policy**
  - Write through: always write to main memory
  - Write back: write to main memory when replaced

NEXT TIME

- **Virtual Memory**