

# 15-213

*“The course that gives CMU its Zip!”*

## **Main Memory and Caches** **Sept. 23, 2008**

### **Topics**

- **DRAM as main memory**
- **Locality of reference**
- **Caches**

# Announcements

## Exam Thursday (two days from now)

- In class
- See exams page on class website for info and old exams

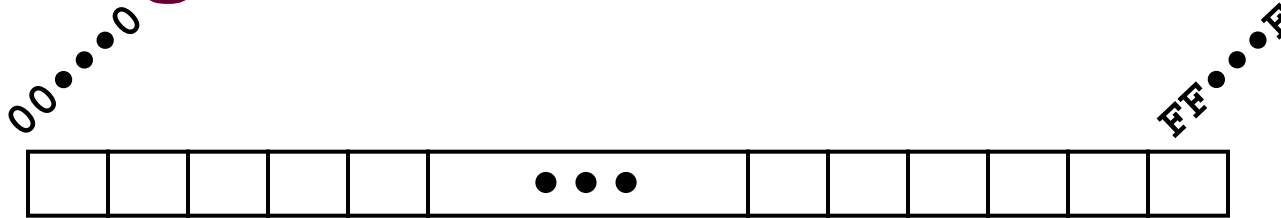
## Calculator policy

- Calculators will not be needed on the exam; hence forbidden

## Collaboration reminder

- Writing code together counts as “sharing code” - forbidden
- Talking through a problem can include pictures (not code)

# Byte-Oriented Memory Organization



- **Programs Refer to Virtual Memory Addresses**
  - Conceptually very large array of bytes
  - Actually implemented with hierarchy of different memory types
  - System provides address space private to particular “process”
    - Program being executed
    - Program can clobber its own data, but not that of others
- **Compiler + Run-Time System Control Allocation**
  - Where different program objects should be stored
  - All allocation within single virtual address space

# Simple Addressing Modes

- **Normal (R) Mem[Reg[R]]**

- Register R specifies memory address

```
movl (%ecx), %eax
```

- **Displacement D(R) Mem[Reg[R]+D]**

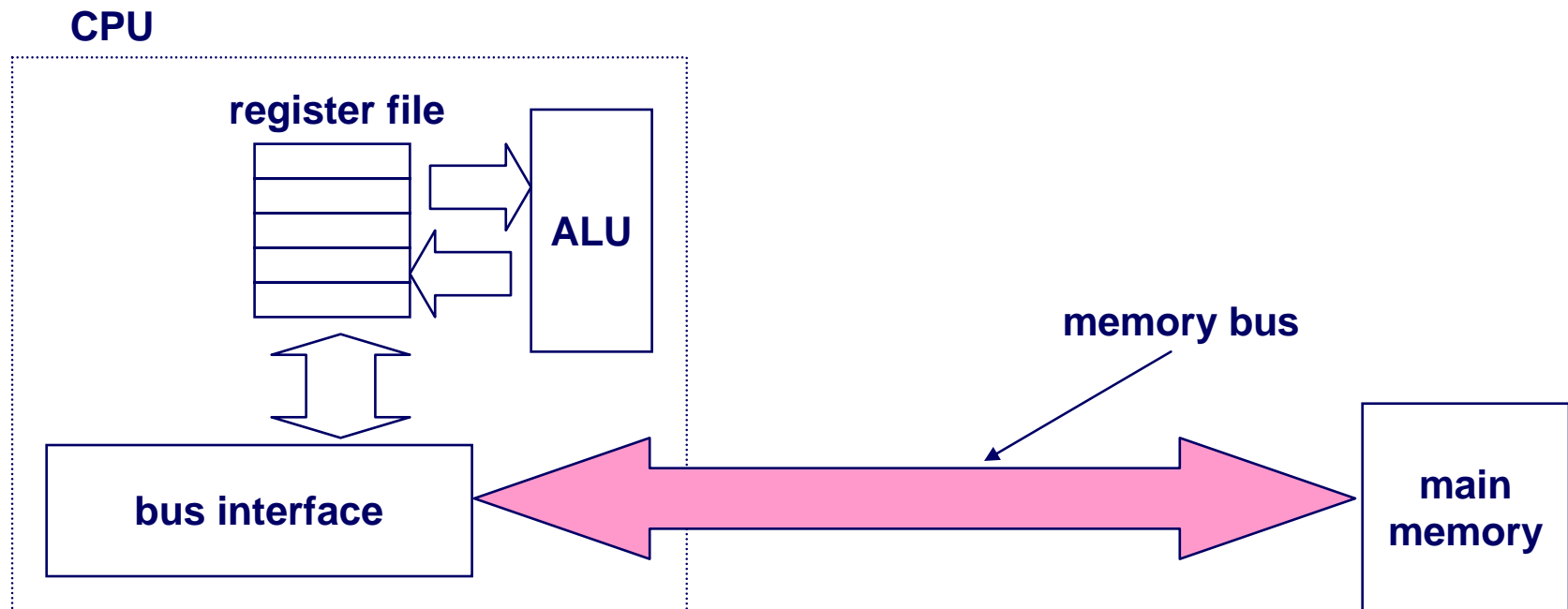
- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movl 8(%ebp), %edx
```

# Traditional Bus Structure Connecting CPU and Memory

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

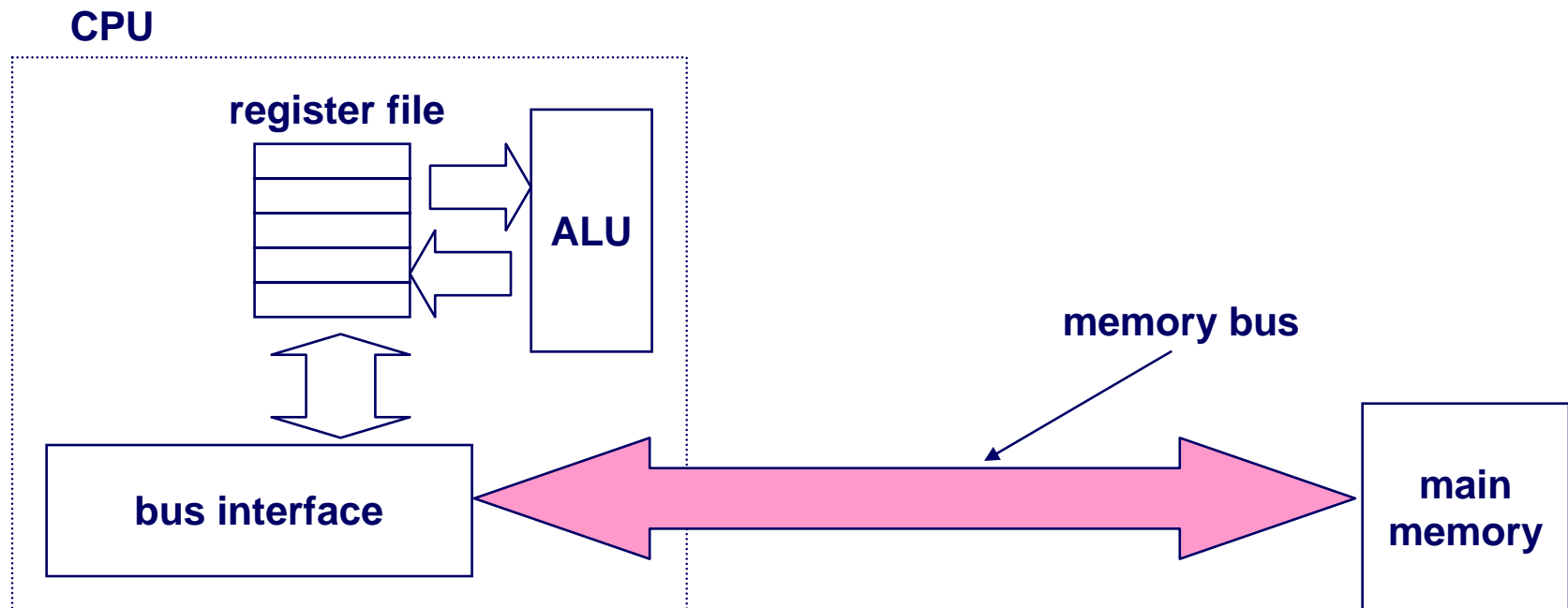
Buses are typically shared by multiple devices.



# Traditional Bus Structure Connecting CPU and Memory

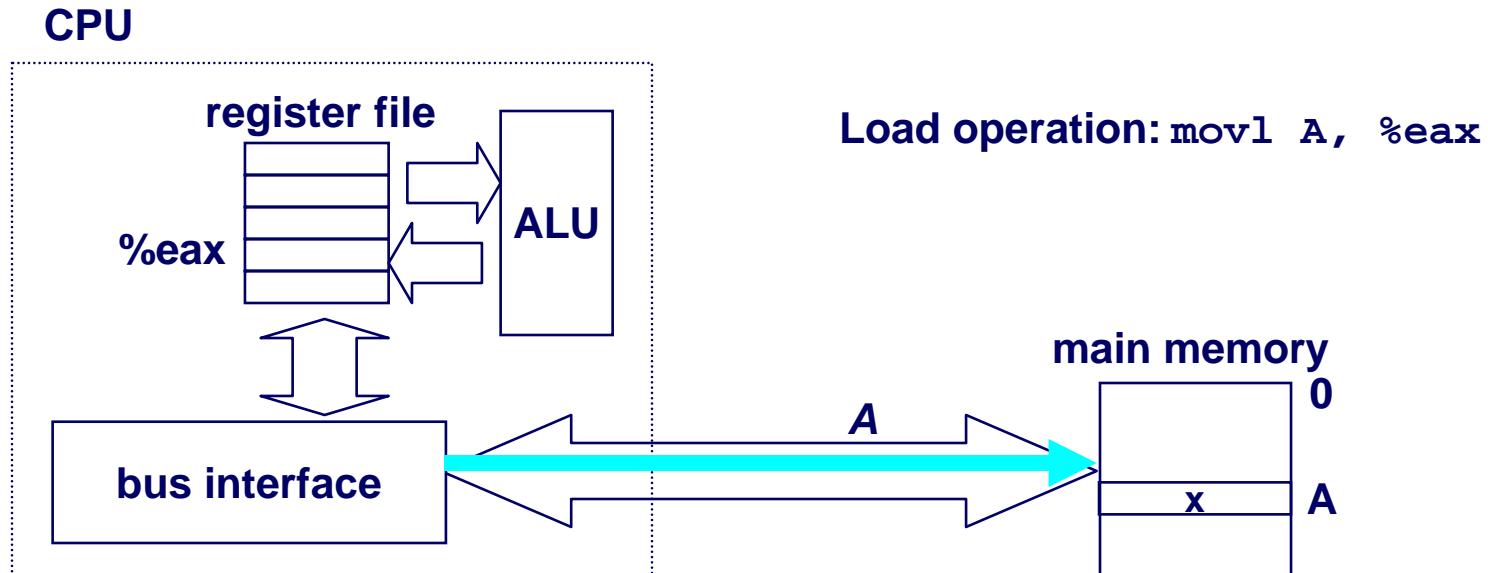
A **bus** is a collection of parallel wires that carry address, data, and control signals.

Buses are typically shared by multiple devices.



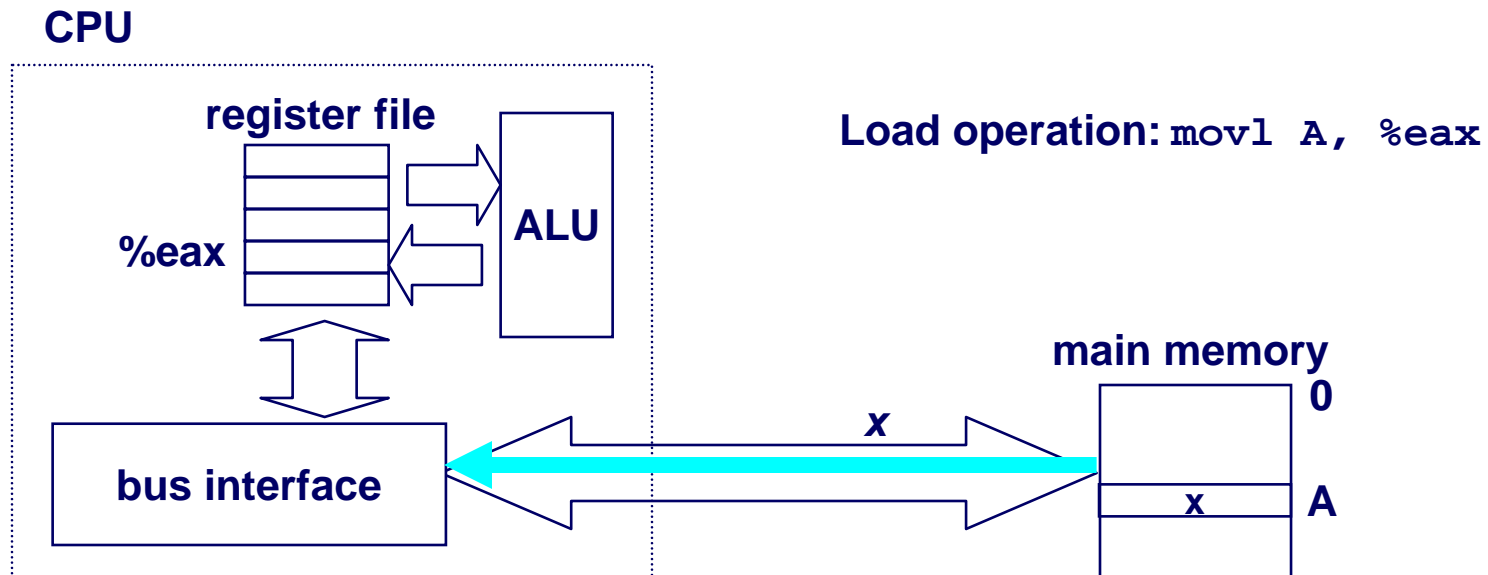
# Memory Read Transaction (1)

**Step 1: CPU places address A on the memory bus with signal indicating “read”**



# Memory Read Transaction (2)

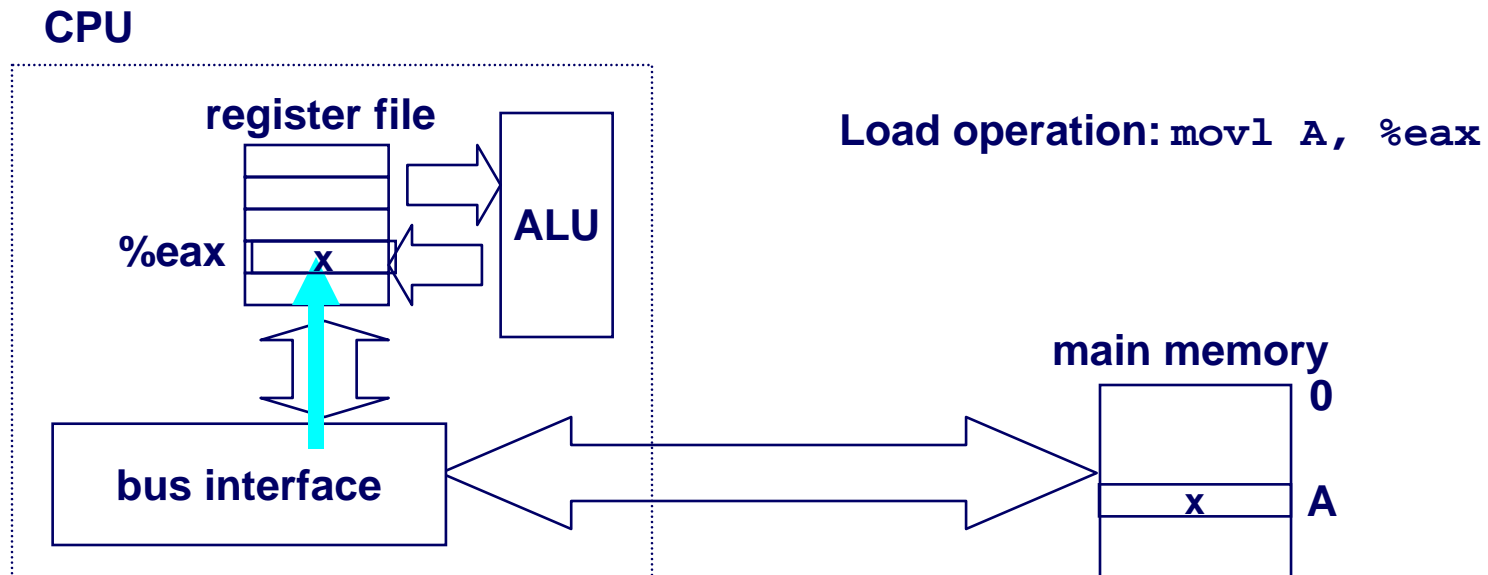
**Steps 2-4: Main memory reads A from the memory bus, retrieves word x, and places it on the bus**





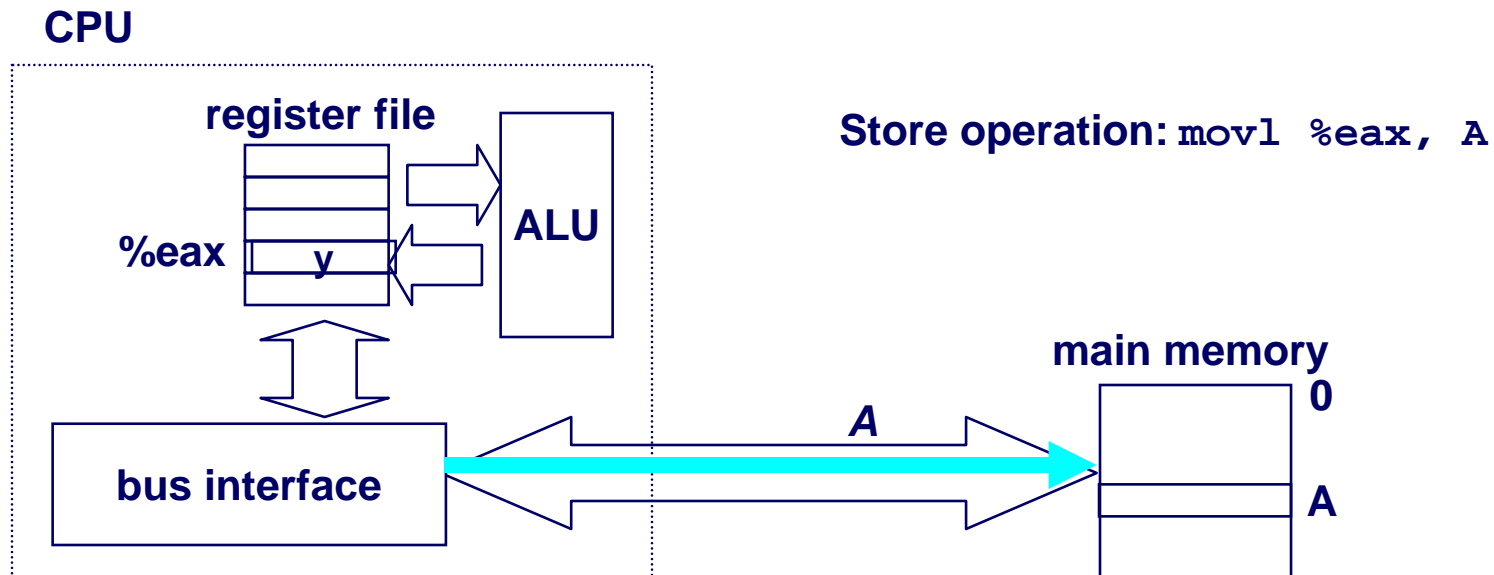
# Memory Read Transaction (3)

**Step 5: CPU reads word x from the bus and copies it into register %eax**



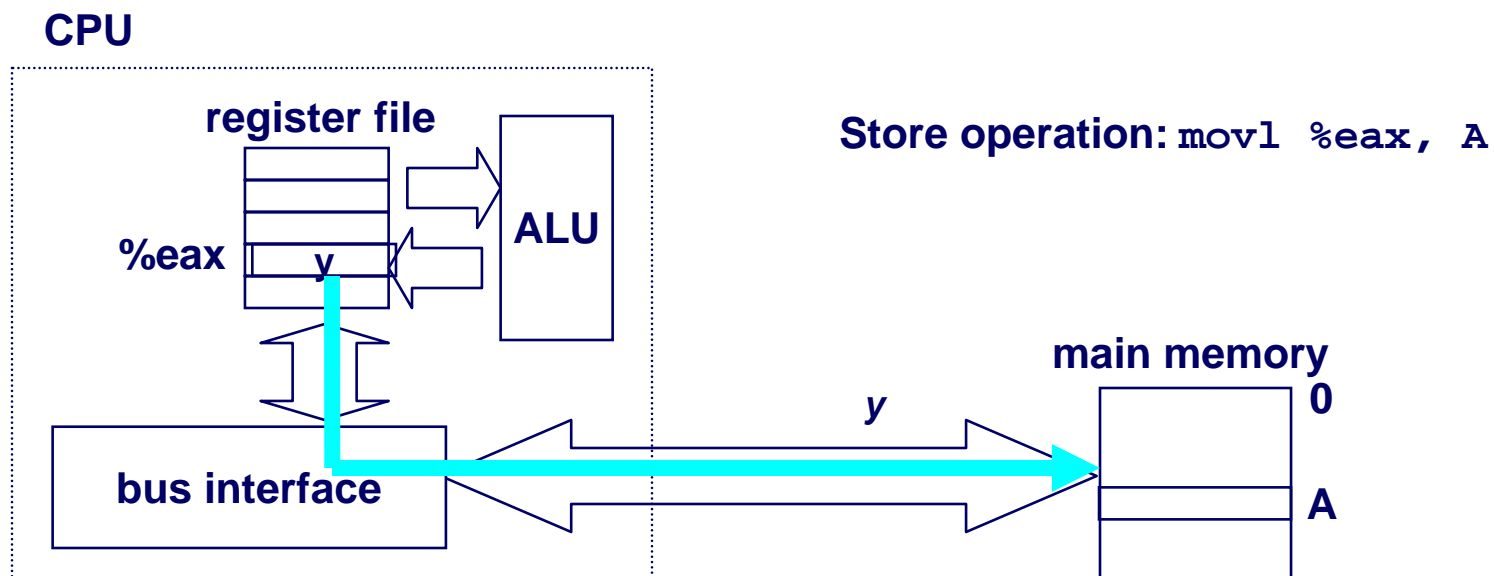
# Memory Write Transaction (1)

**Step 1: CPU places address A on the memory bus with signal indicating “write”**



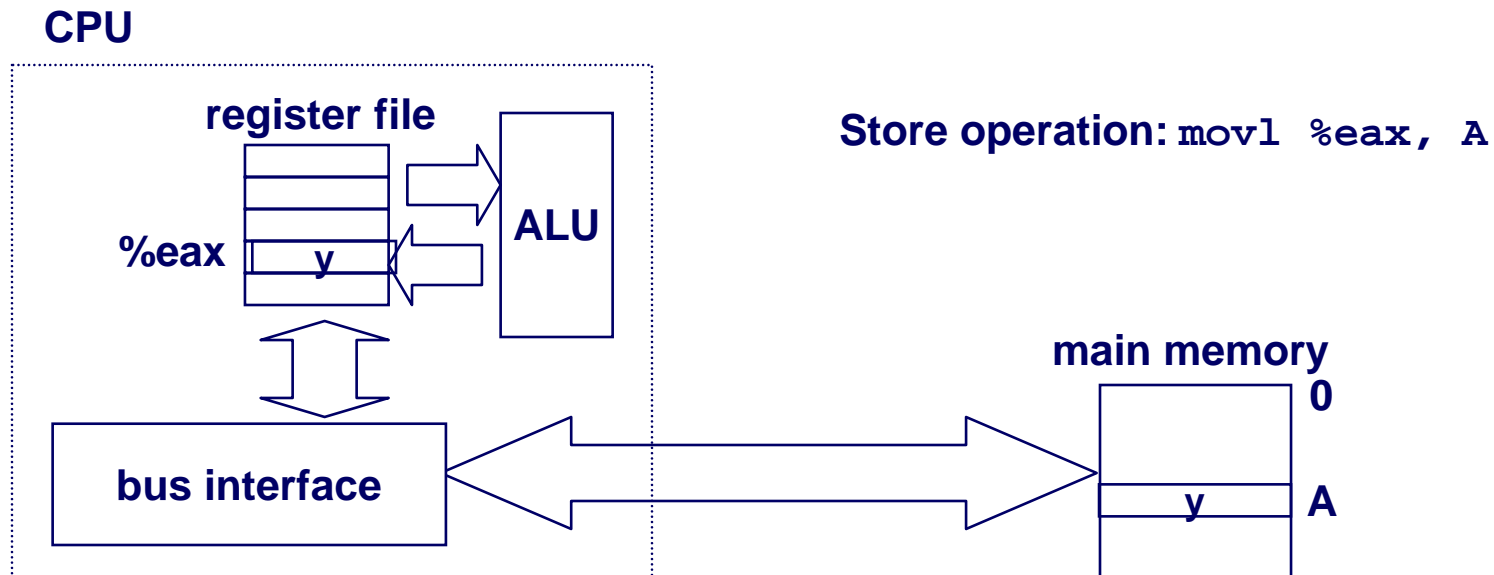
# Memory Write Transaction (2)

**Step 2: CPU places data word  $y$  on the memory bus**



# Memory Write Transaction (3)

**Steps 3-4: Main memory reads data word *y* from the bus and stores it at address *A***



# Random-Access Memory (RAM)

## Key features

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

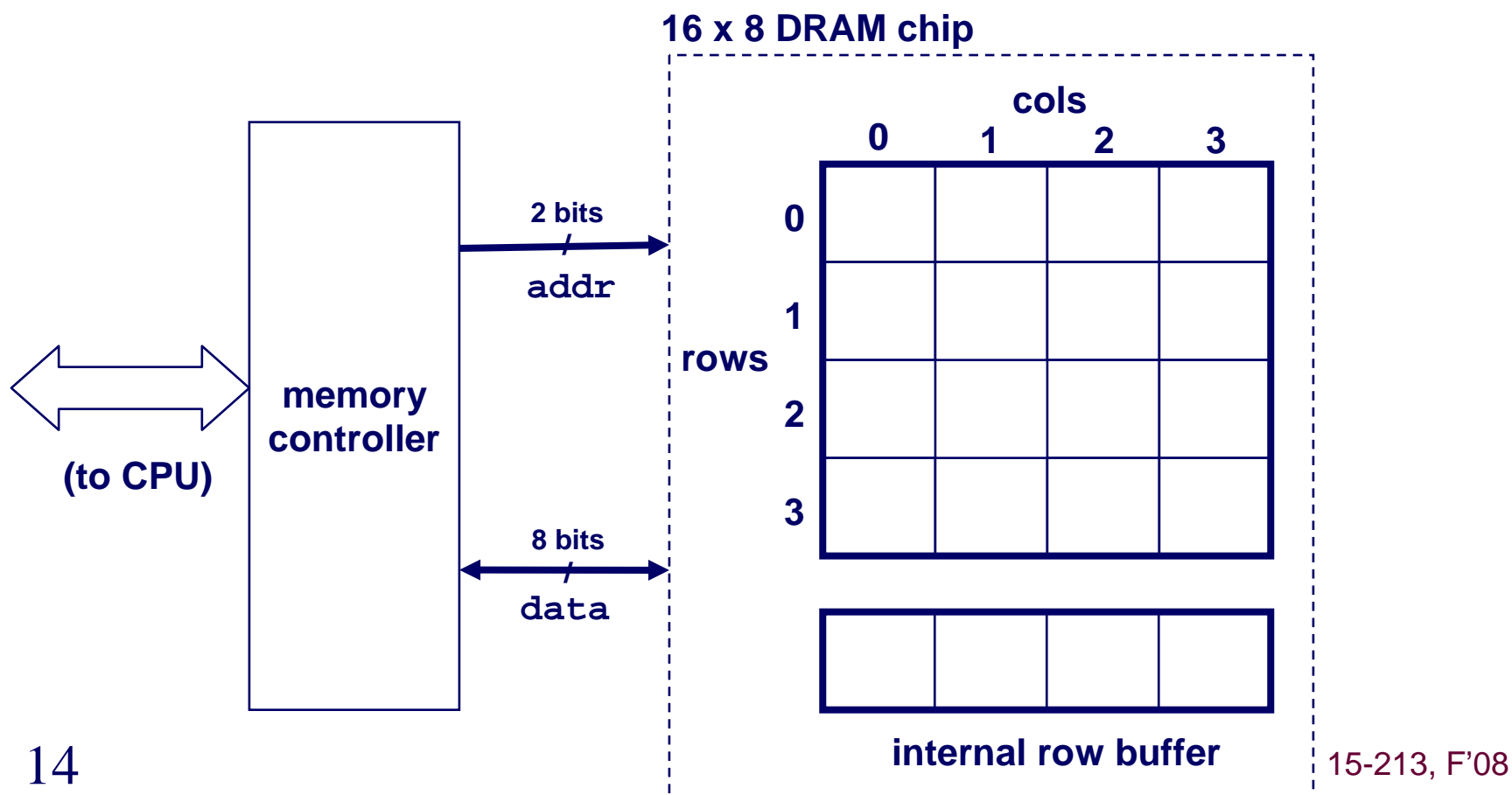
## Dynamic RAM (**DRAM**)

- Common technology for main memory
- Organized in two dimensions (rows and columns)
  - To access: select row then select column
  - Consequence: 2<sup>nd</sup> row access faster than different column/row
- Some technical details
  - Each cell stores bit with a capacitor
  - One transistor is used for access
  - Value must be refreshed every 10-100 ms

# Conventional DRAM Organization

## $d \times w$ DRAM:

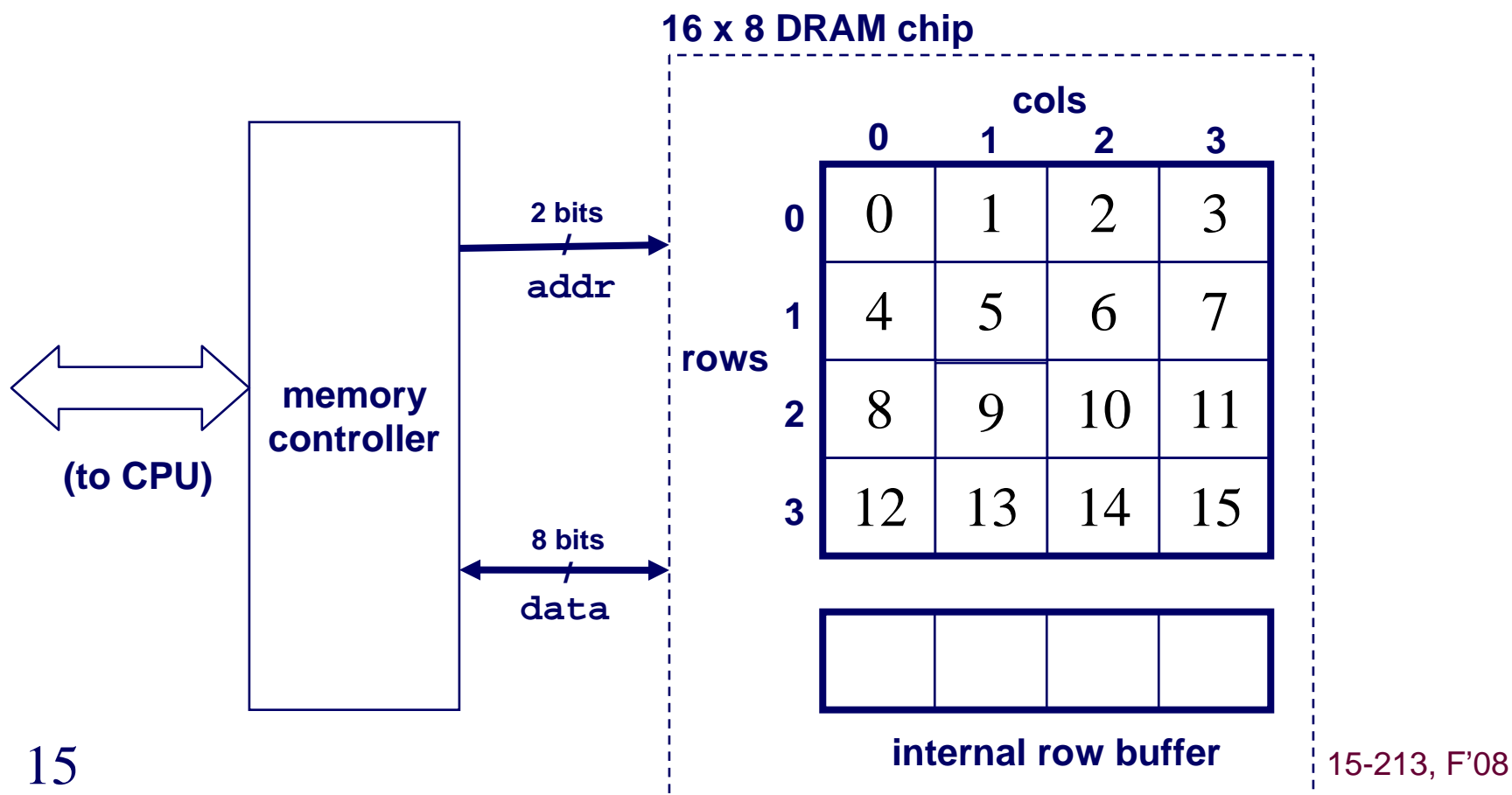
- $dw$  total bits organized as  $d$  **supercells** of size  $w$  bits



# Conventional DRAM Organization

## $d \times w$ DRAM:

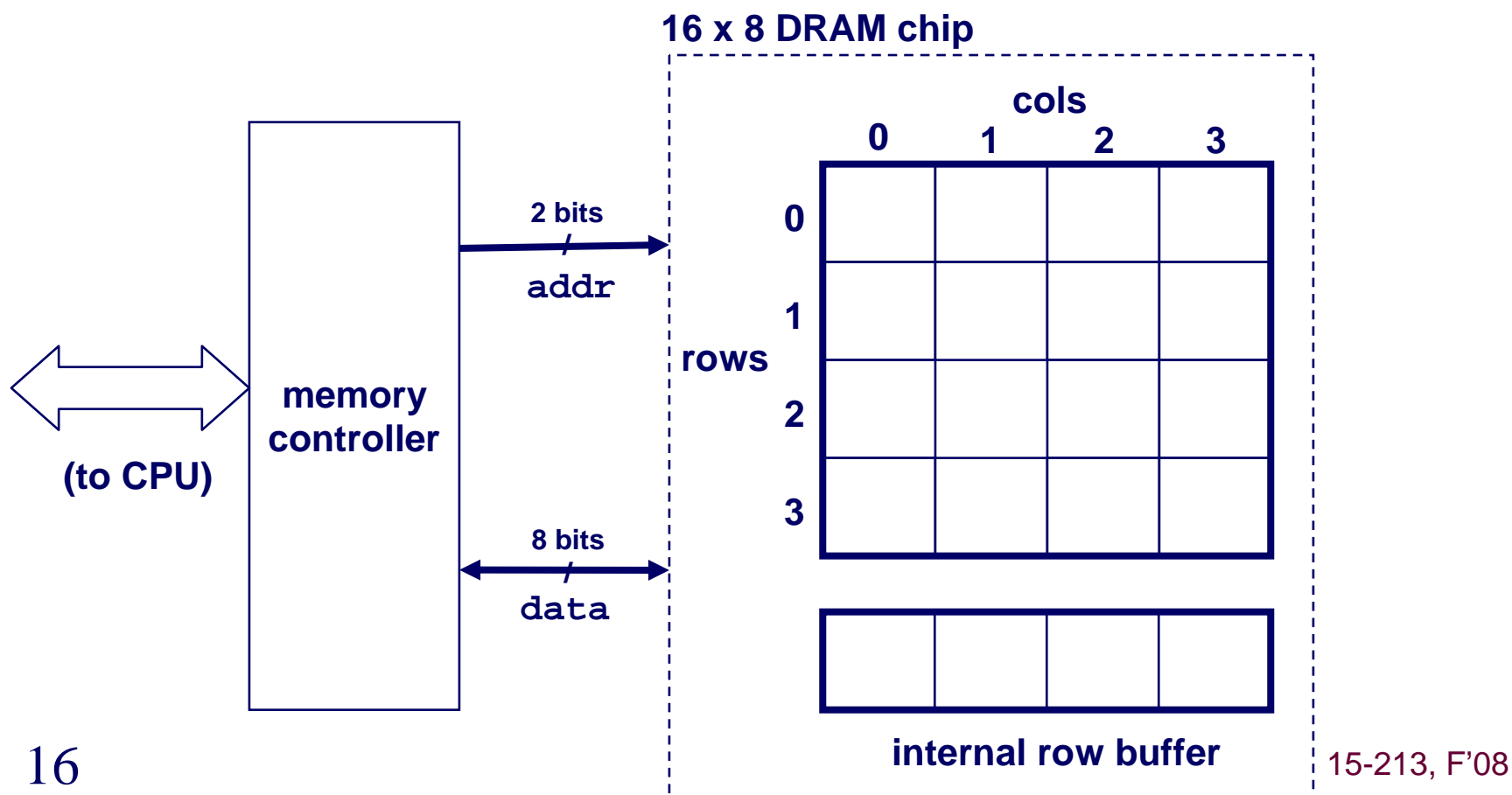
- $dw$  total bits organized as  $d$  **supercells** of size  $w$  bits



# Conventional DRAM Organization

## $d \times w$ DRAM:

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

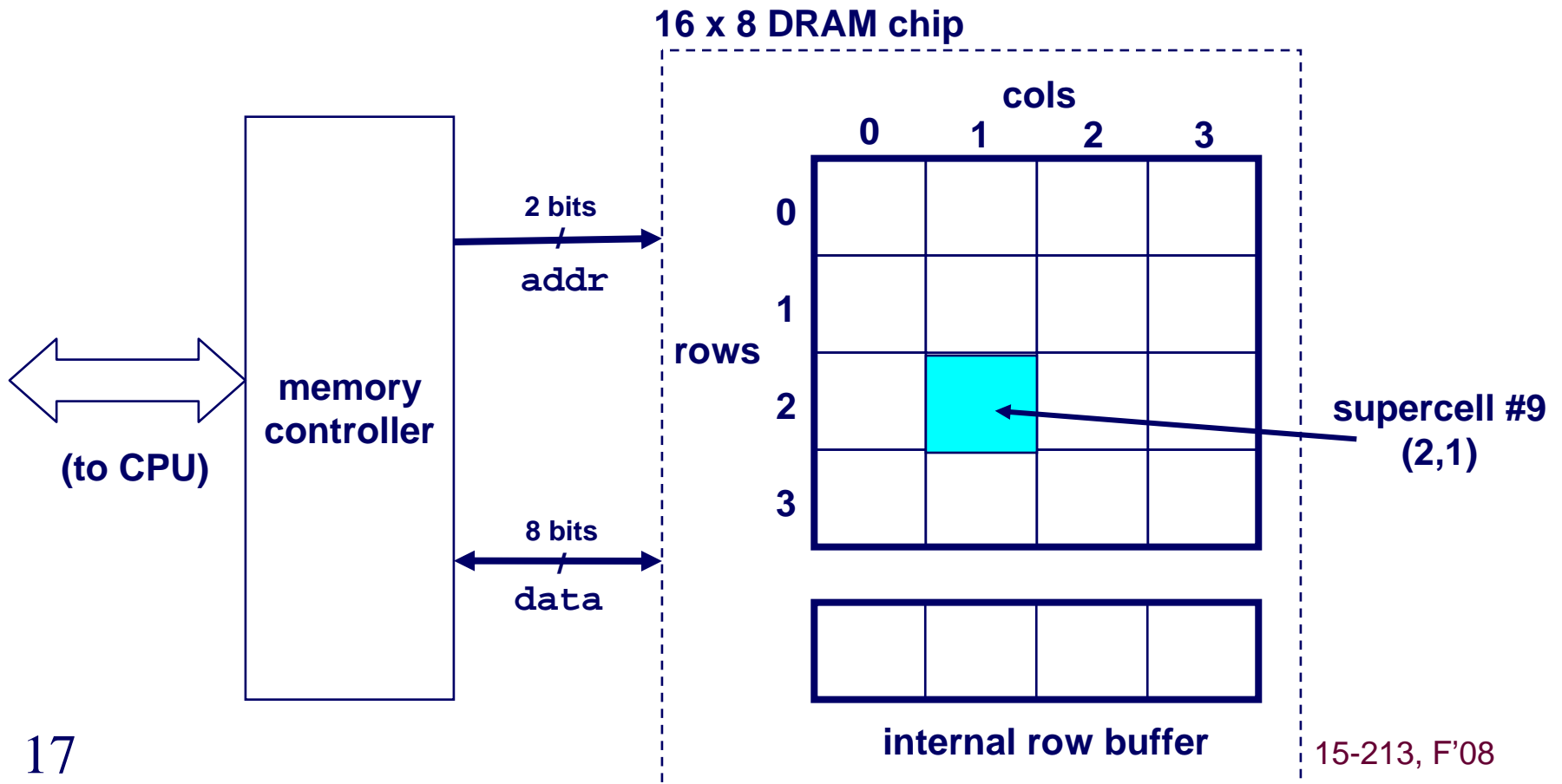




# Conventional DRAM Organization

## $d \times w$ DRAM:

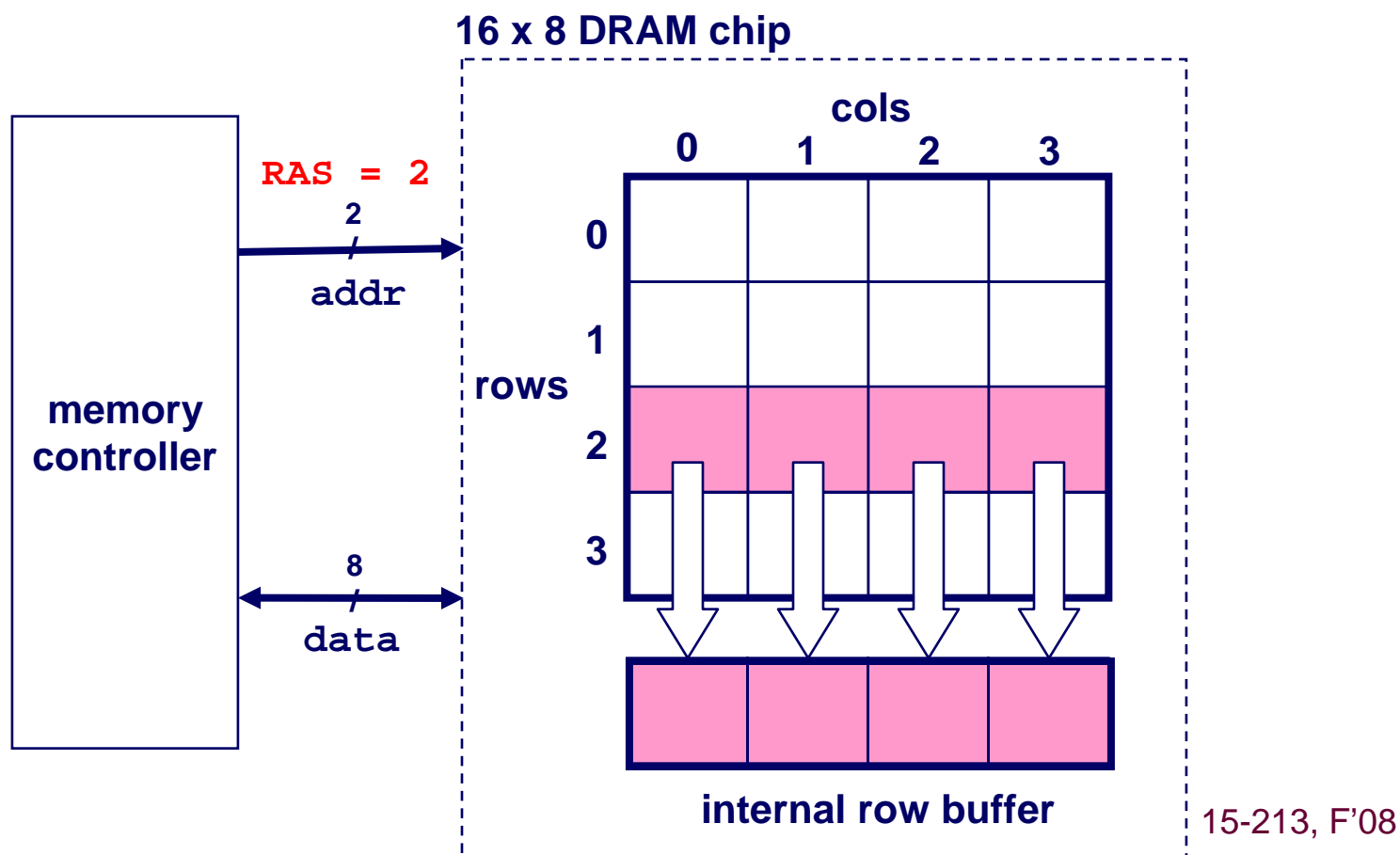
- $dw$  total bits organized as  $d$  **supercells** of size  $w$  bits



# Reading DRAM Supercell #9 = (2,1)

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

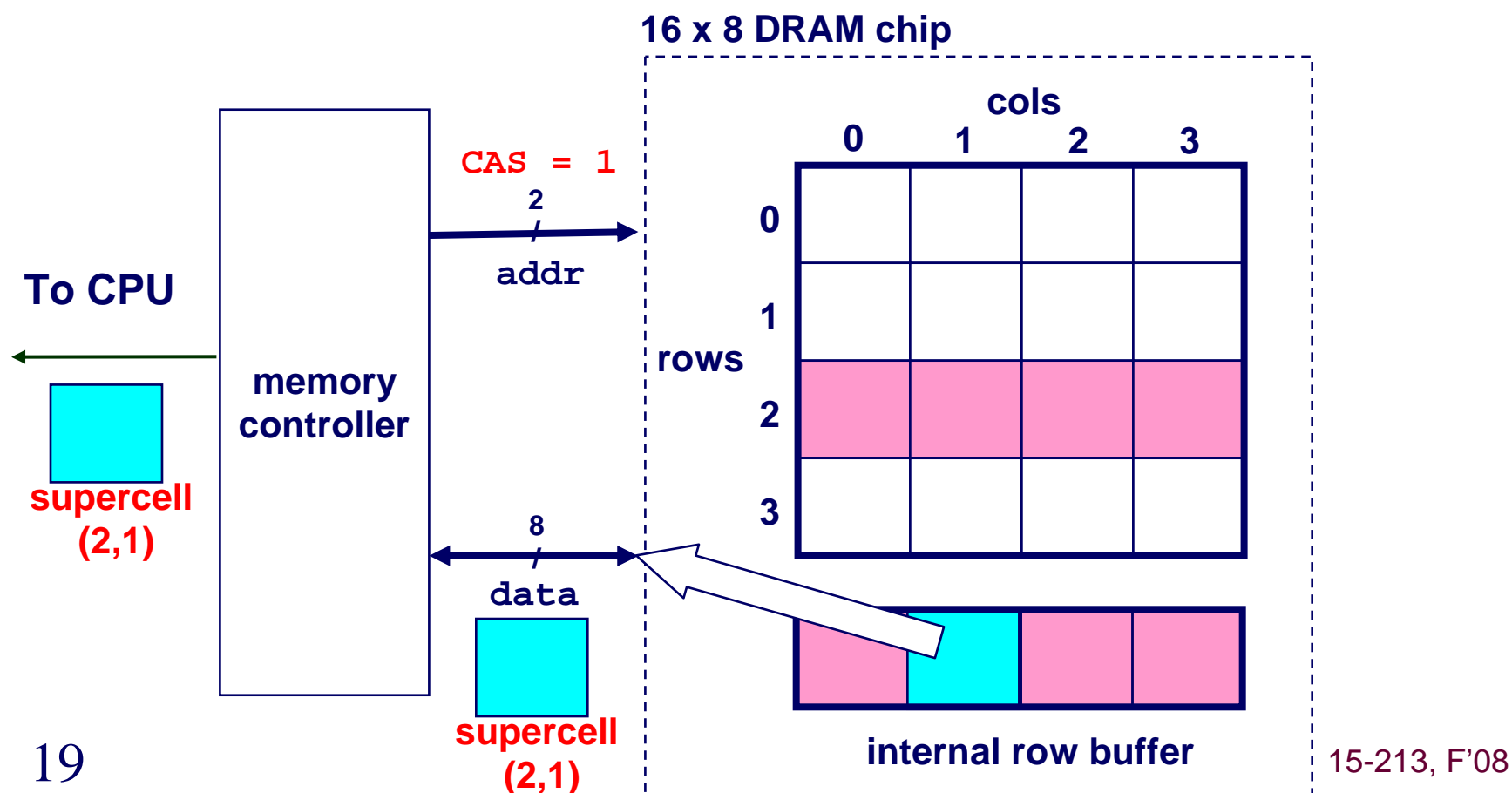
Step 1(b): Row 2 copied from DRAM array to row buffer



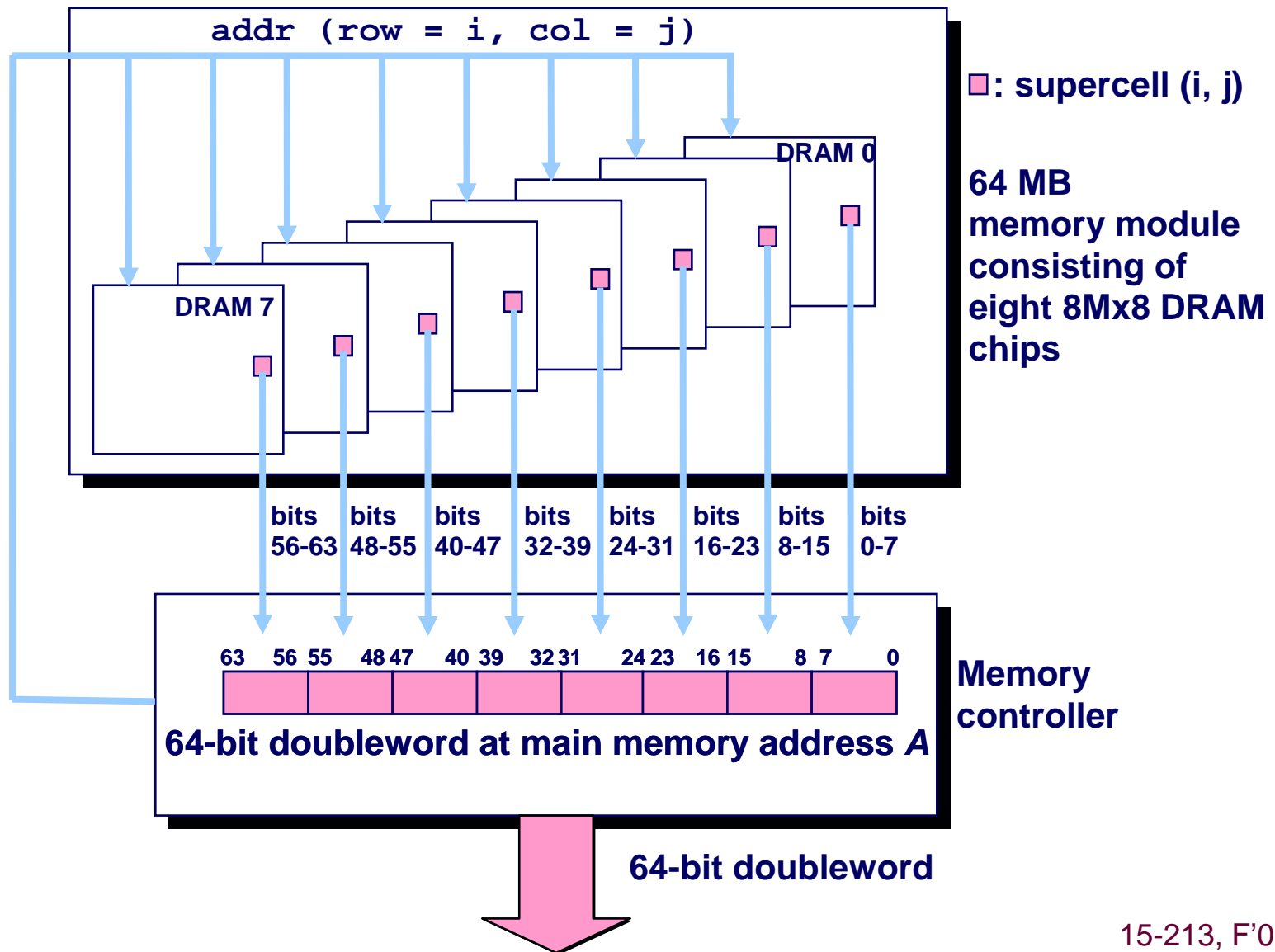
# Reading DRAM Supercell #9 = (2,1)

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



# Multi-chip Memory Modules



# Memory access is slow

## Observation: memory access is slower than CPU cycles

- A DRAM chip has an access time of 30-50ns
  - further, systems may need 3x longer or more to get the data from memory into a CPU register
- With sub-ns cycle times, 100s of cycles per memory access
  - and, the gap has been growing

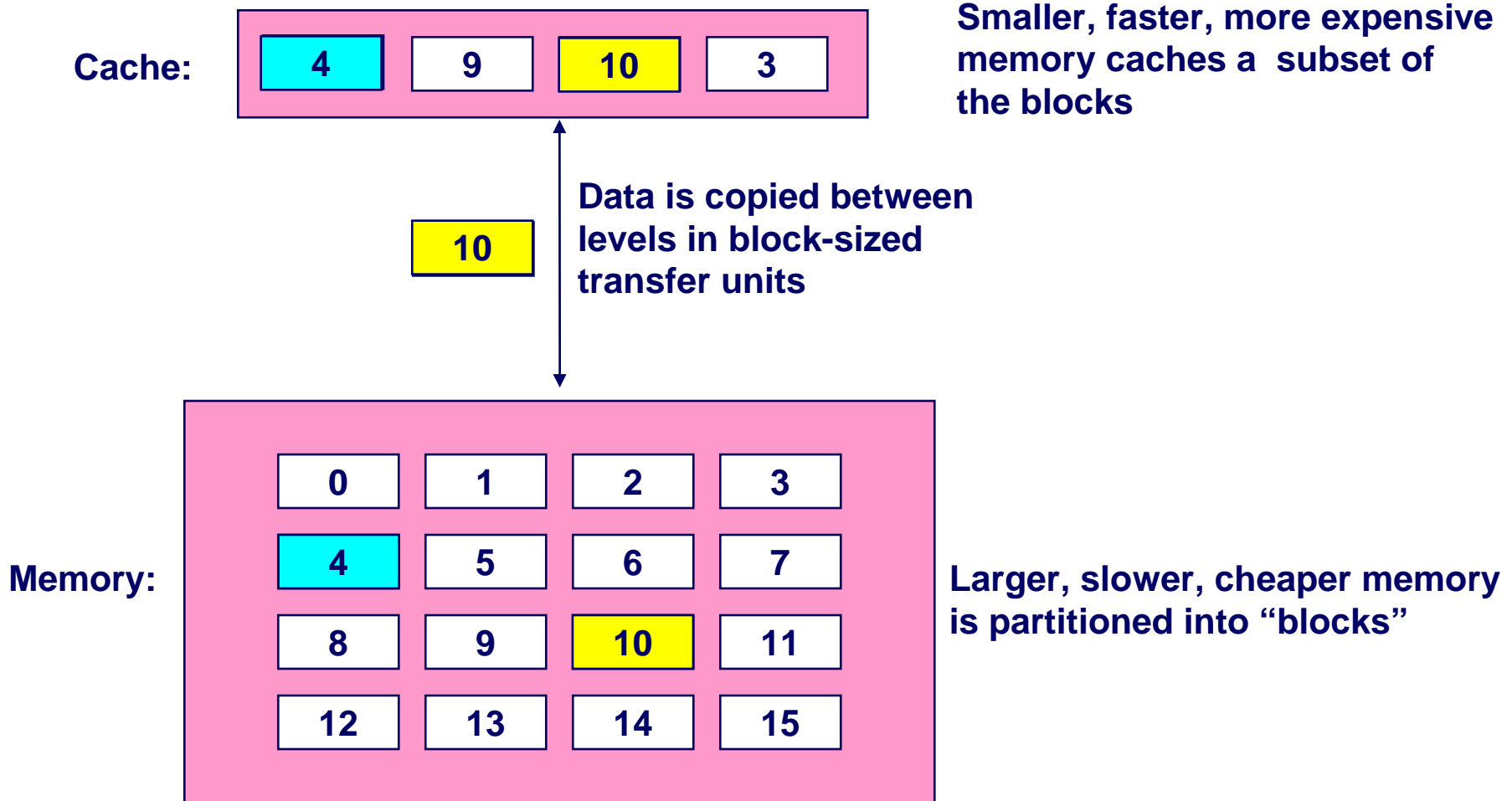
## Can't go to memory on every load and store

- approximately 1/3 of instructions are loads or stores

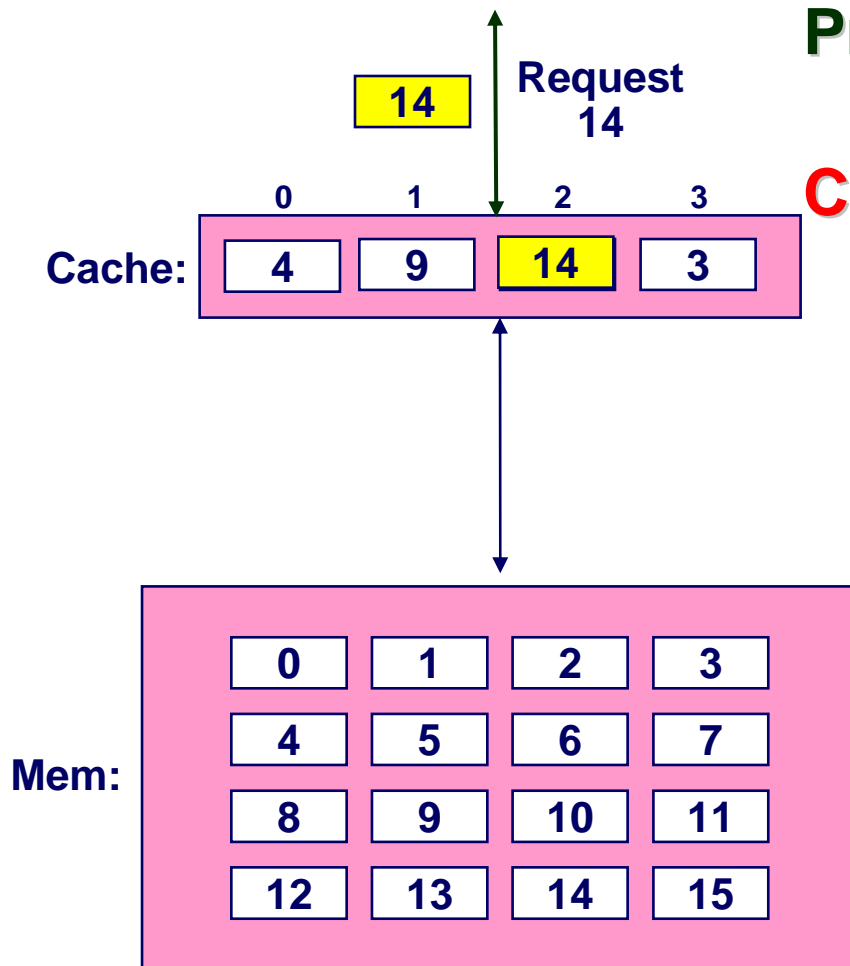
# Caches to the rescue

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

# General cache mechanics



# General Caching Concepts (hit)



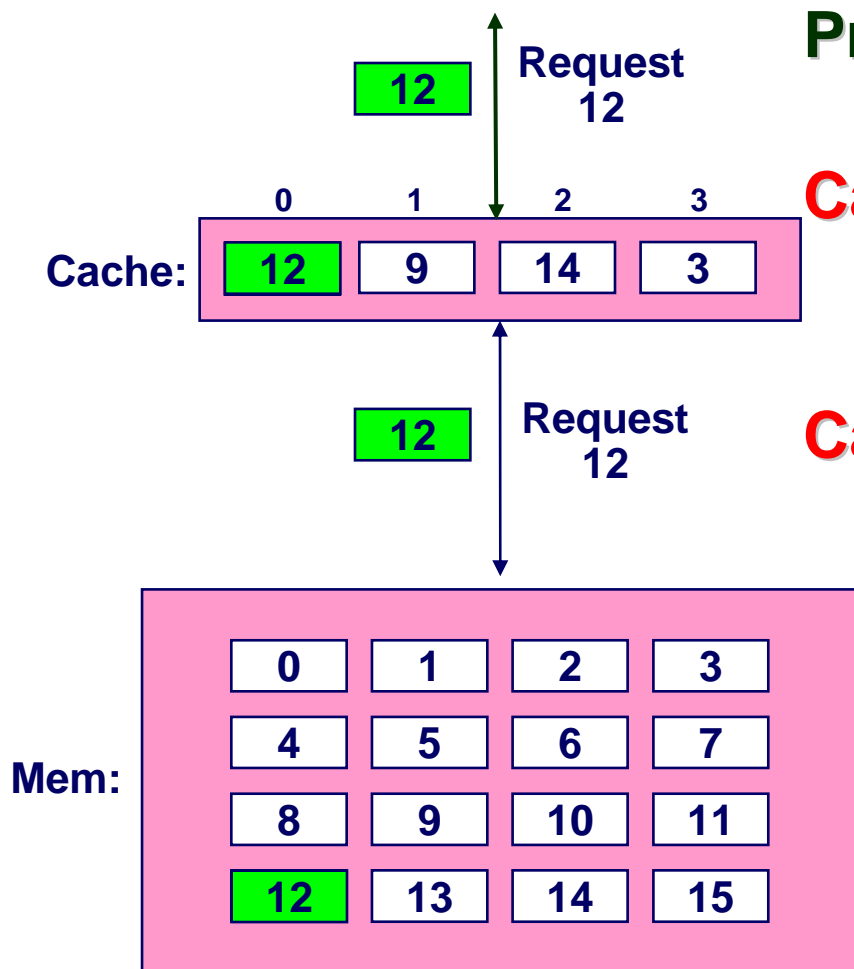
Program needs object d, which is stored in some block b

**Cache hit**

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



# General Caching Concepts (miss)



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

## Cache hit

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

## Cache miss

- b is not in cache, so must fetch it  
E.g., block 12
- If 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., slot  $\#(b \text{ mod } 4)$
  - Replacement policy:** which block should be evicted? E.g., LRU

# Types of cache misses

## Cold (compulsory) miss

- Cold misses occur on first accesses to given blocks

## Conflict miss

- Most hardware caches limit blocks to a small subset (sometimes a singleton) of the available cache slots
  - e.g., block  $i$  must be placed in slot  $(i \bmod 4)$
- Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
  - e.g., referencing blocks 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

# Types of cache misses

## Cold (compulsory) miss

- Cold misses occur on first accesses to given blocks

## Conflict miss

- Most hardware caches limit blocks to a small subset (sometimes a singleton) of the available cache slots
  - e.g., block  $i$  must be placed in slot  $(i \bmod 4)$
- Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
  - e.g., referencing blocks 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

# Locality: why caches work

## Principle of Locality:

- Programs tend to use data and instructions with addresses near or equal to those they have used recently
- **Temporal locality:** Recently referenced items are likely to be referenced again in the near future
- **Spatial locality:** Items with nearby addresses tend to be referenced close together in time

## Locality Example:

- **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**

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

# Locality Example #1

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 sum_array_rows(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 #2

**Question:** Does this function have good locality?

```
int sum_array_cols(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 #3

**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 sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

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

    return sum;
}
```

# Memory Hierarchies

**Some fundamental and enduring properties of hardware and software systems:**

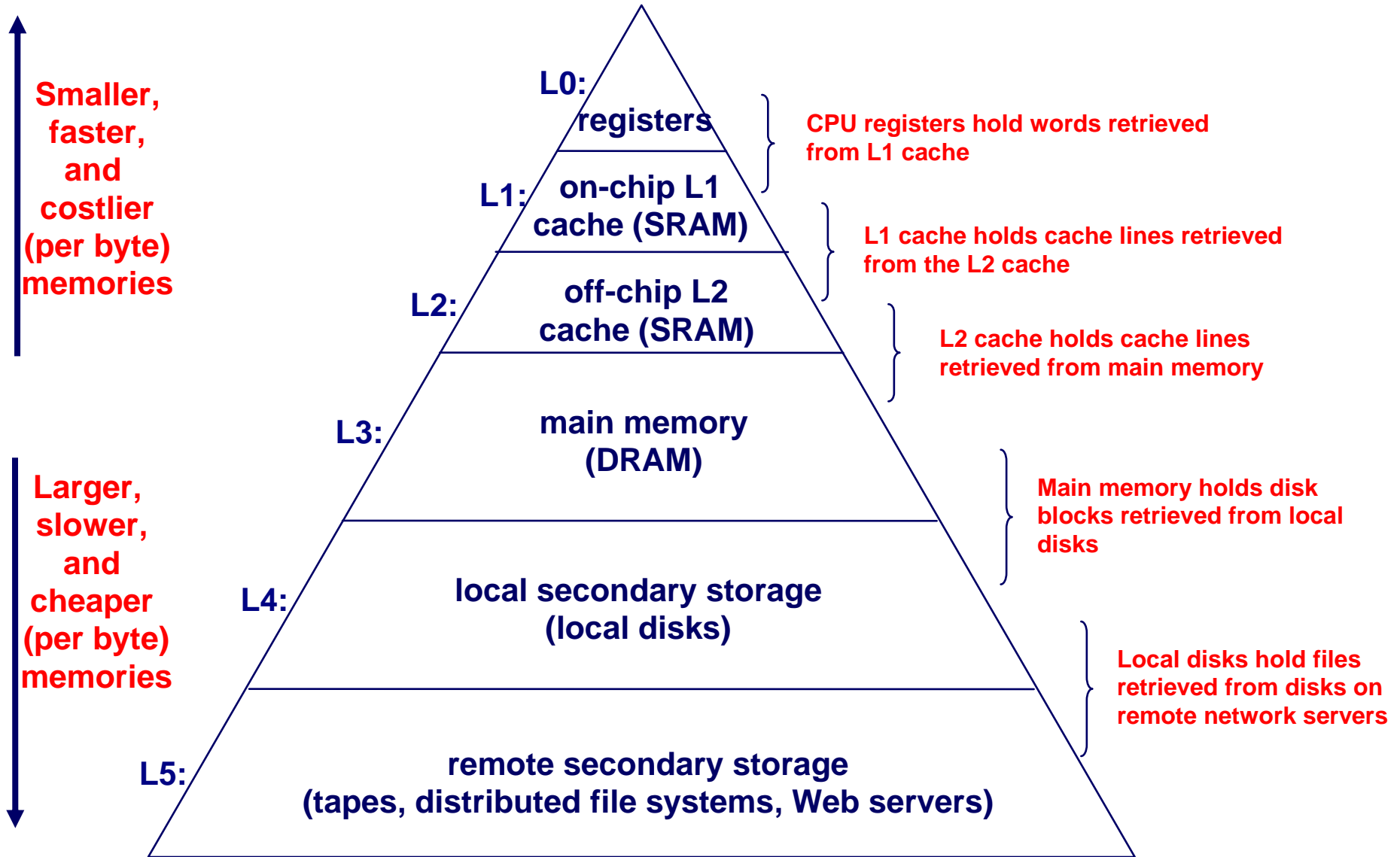
- **Faster storage technologies almost always cost more per byte and have lower capacity**
- **The gaps between memory technology speeds are widening**
  - **True of registers:DRAM, DRAM:disk, etc.**
- **Well-written programs tend to exhibit good locality**

**These properties complement each other beautifully**

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



# An Example Memory Hierarchy



# Caching is the core concept

## Fundamental idea of a memory hierarchy:

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

## Why do memory hierarchies work?

- Locality causes many accesses to be hits at level  $k$ 
  - More than its relative size would suggest
- Thus, many fewer accesses to level  $k+1$
- The storage at level  $k+1$  can be slower, larger and cheaper

**Net effect: A large pool of memory with the cost of cheap storage near the bottom, but the performance of the expensive storage near the top**

# Examples of Caching in the Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-byte words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes block	On-Chip L1	1	Hardware
L2 cache	64-bytes 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

# Summary

- The memory hierarchy is a fundamental consequence of maintaining the *random access memory* abstraction and practical limits on cost and power consumption
- Locality makes caching effective
- Programming for good *temporal* and *spatial* locality is critical for high performance
  - For caching and for row-heavy access to DRAM
- Trend: the speed gaps between levels of the memory hierarchy continue to widen
  - Consequence: inducing locality becomes even more important