

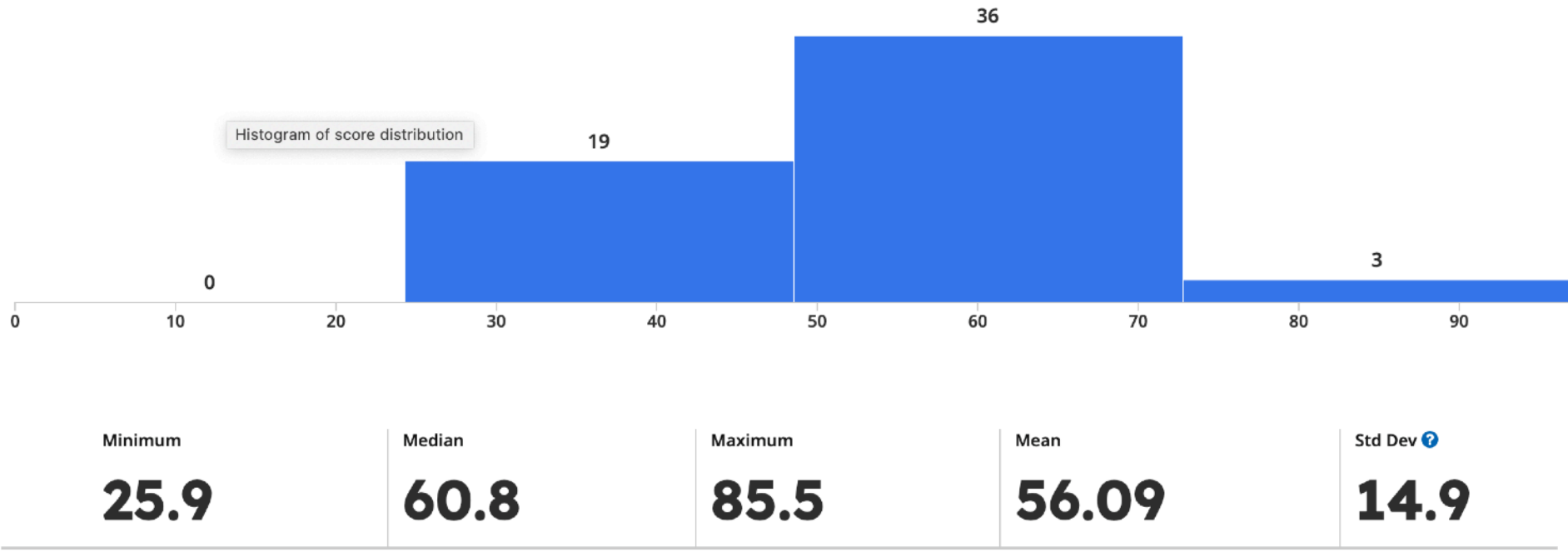
# **CSC 252: Computer Organization**

## **Spring 2023: Lecture 16**

Instructor: Yuhao Zhu

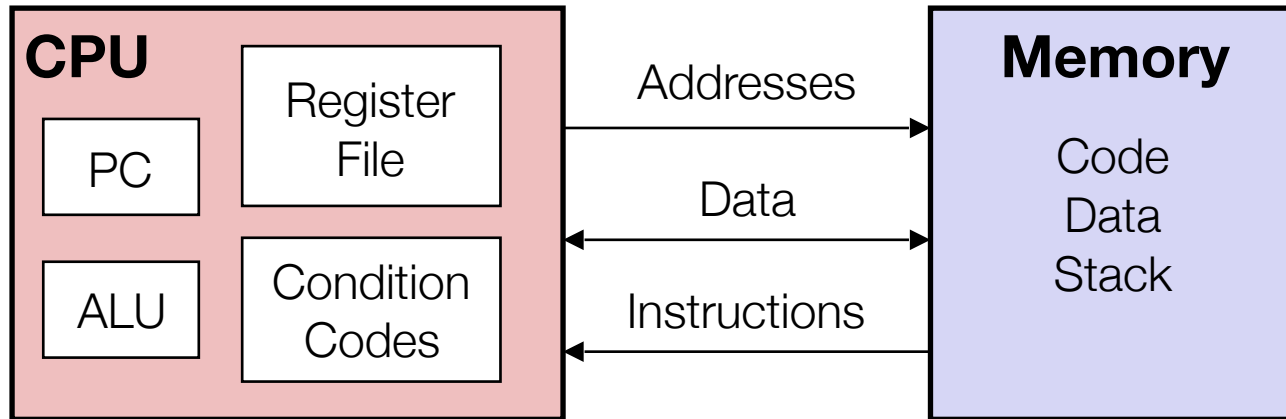
Department of Computer Science  
University of Rochester

# Announcements



- If you have doubts about your midterm grade, calm down and talk to the TAs.
- Will post the solution today.

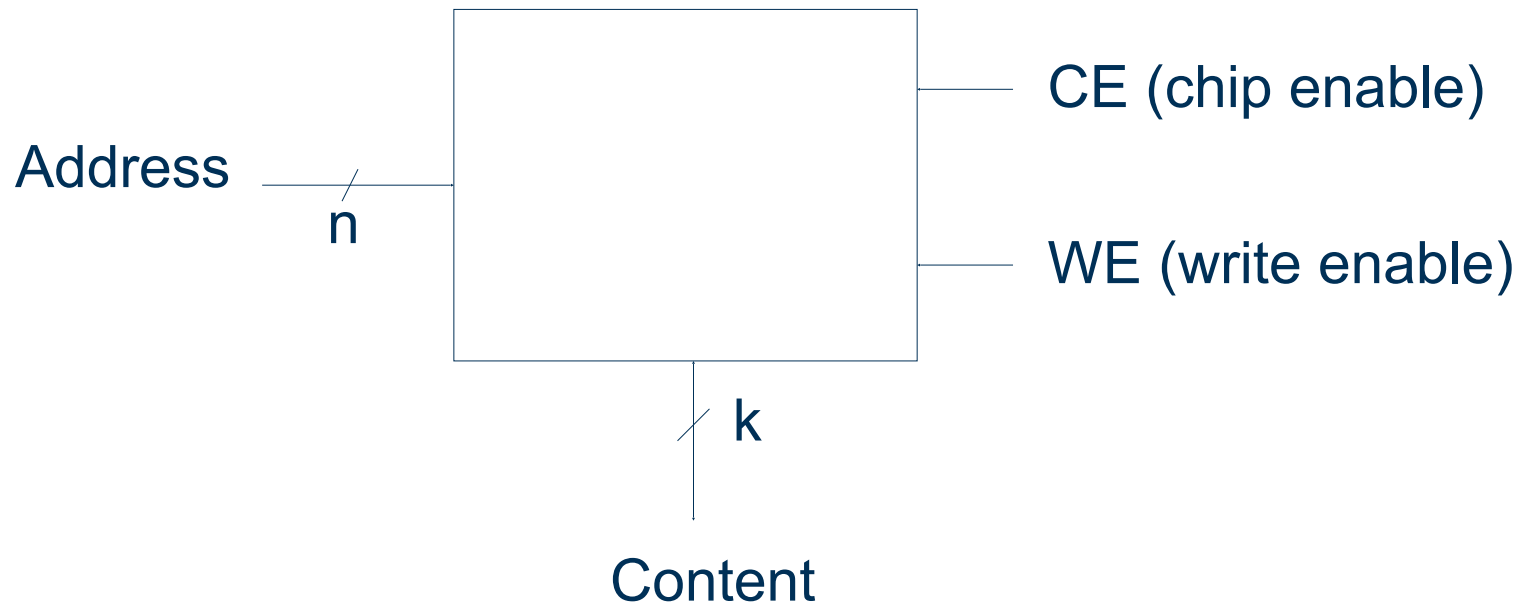
# So far in 252...



- We have been discussing the CPU microarchitecture
  - Single Cycle, sequential implementation
  - Pipeline implementation
  - Resolving data dependency and control dependency
- What about memory?
  - Sometimes called the main memory to differentiate it from other storage units inside the CPU such as the register file.
  - What is a programmer's view of memory?

# Abstract View of RAM

- Random access memory
- Given an arbitrary address, a RAM should return data at that address
- Think of it as a hardware implementation of an array
- What are data structures that do not support random accesses?



# Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

# The Problem

- Ideal memory's requirements oppose each other

# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower

# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location



# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive

# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: Flip-flop vs. SRAM vs. DRAM vs. Disk vs. Tape

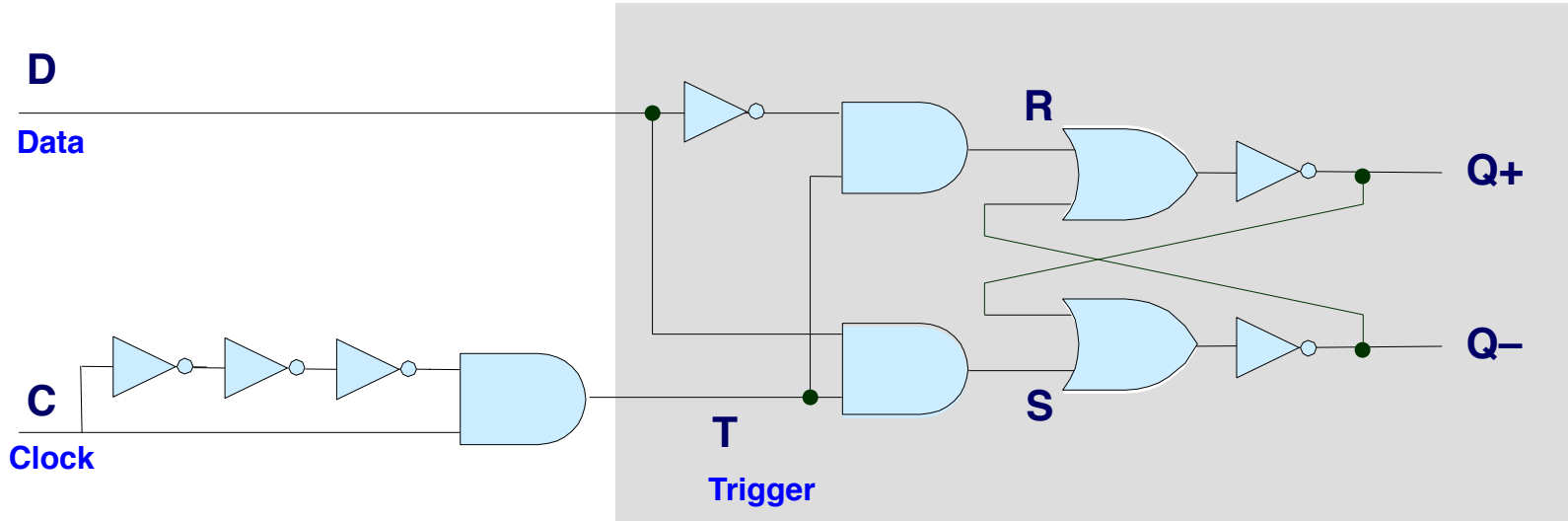
# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: Flip-flop vs. SRAM vs. DRAM vs. Disk vs. Tape
- Higher bandwidth is more expensive

# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: Flip-flop vs. SRAM vs. DRAM vs. Disk vs. Tape
- Higher bandwidth is more expensive
  - Need more ports, higher frequency, or faster technology

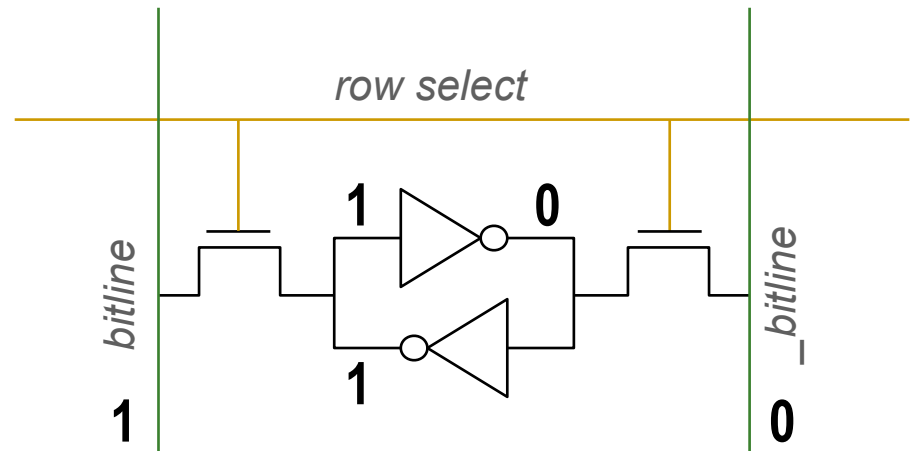
# Memory Technology: D Flip-Flop (DFF)



- Very fast
- Very expensive to build
  - 6 NOT gates (2 transistors / gate)
  - 3 AND gates (3 transistors / gate)
  - 2 OR gates (3 transistors / gate)
  - 27 transistors in total for just one bit!!
- Usually used to build the register file, not the main memory

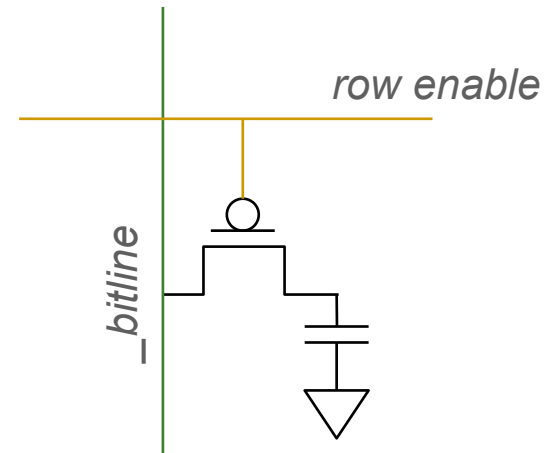
# Memory Technology: SRAM

- Static random access memory
- Random access means you can supply an arbitrary address to the memory and get a value back
- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the “cell”
  - 4 transistors for storage
  - 2 transistors for access
  - 6 transistors in total per bit



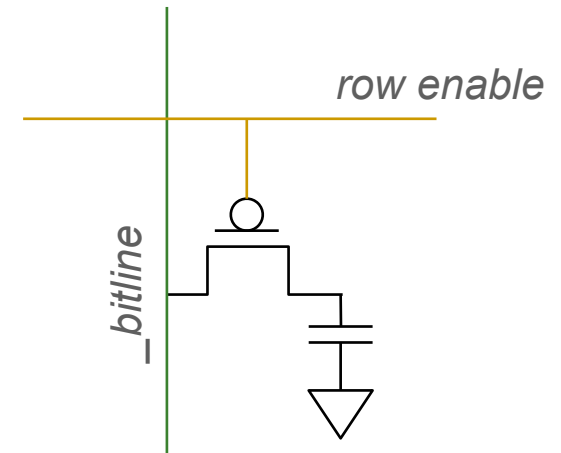
# Memory Technology: DRAM

- Dynamic random access memory



# Memory Technology: DRAM

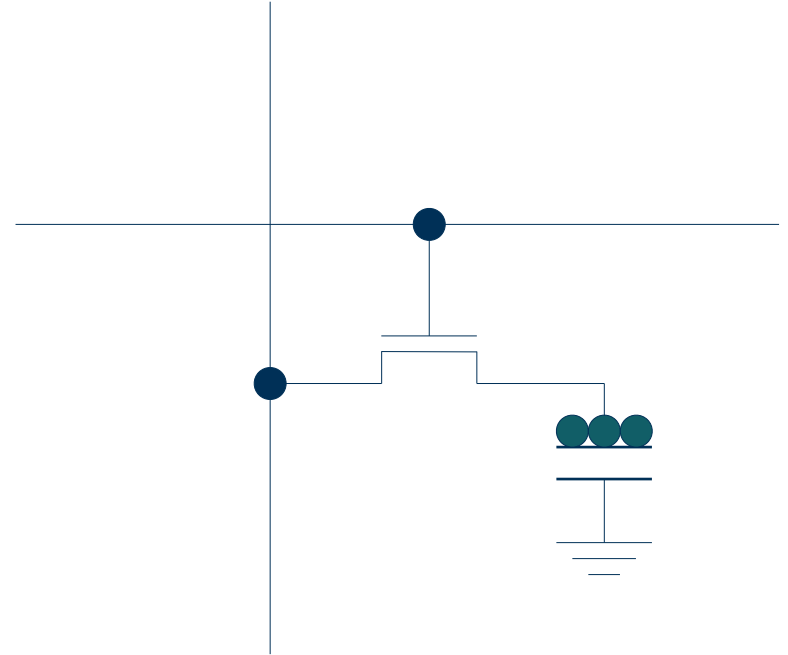
- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor





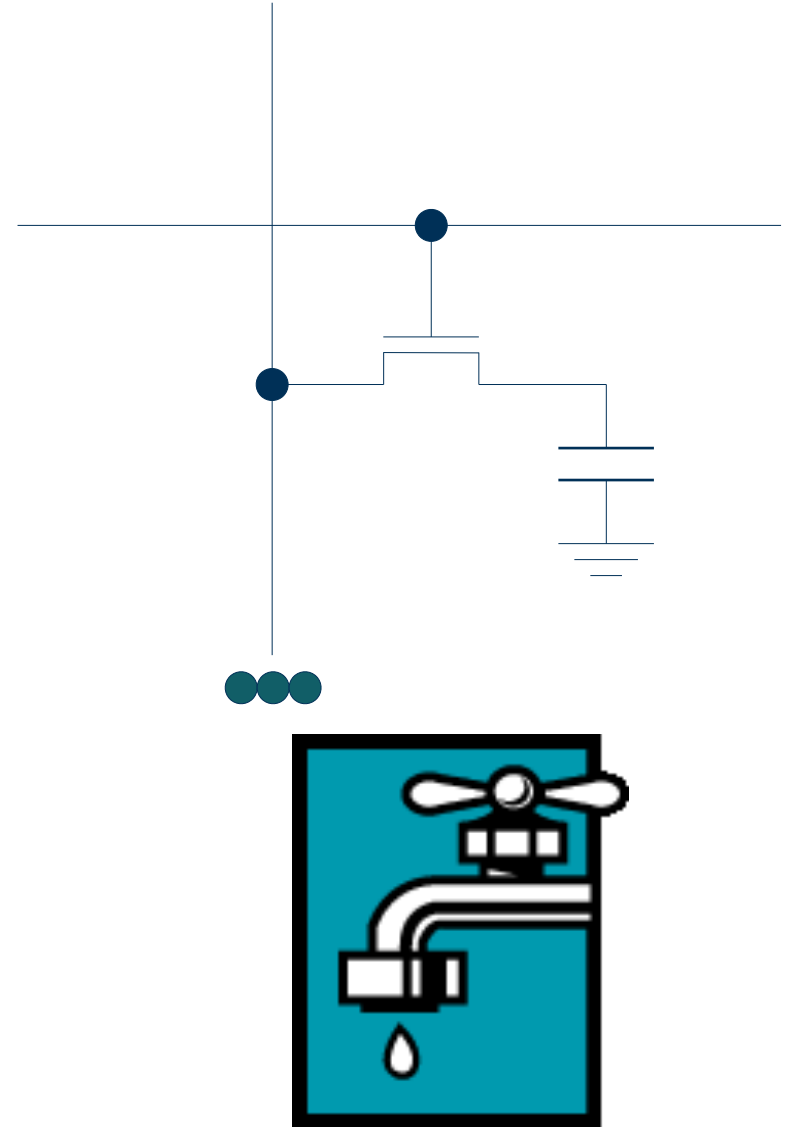
# DRAM Cell

- Capacitors will leak!
  - DRAM cell loses charge over time
  - DRAM cell needs to be **refreshed** periodically.
  - Refresh takes time and power.  
When refreshing can't read the data. A major issue, lots of research going on to reduce the refresh overhead.



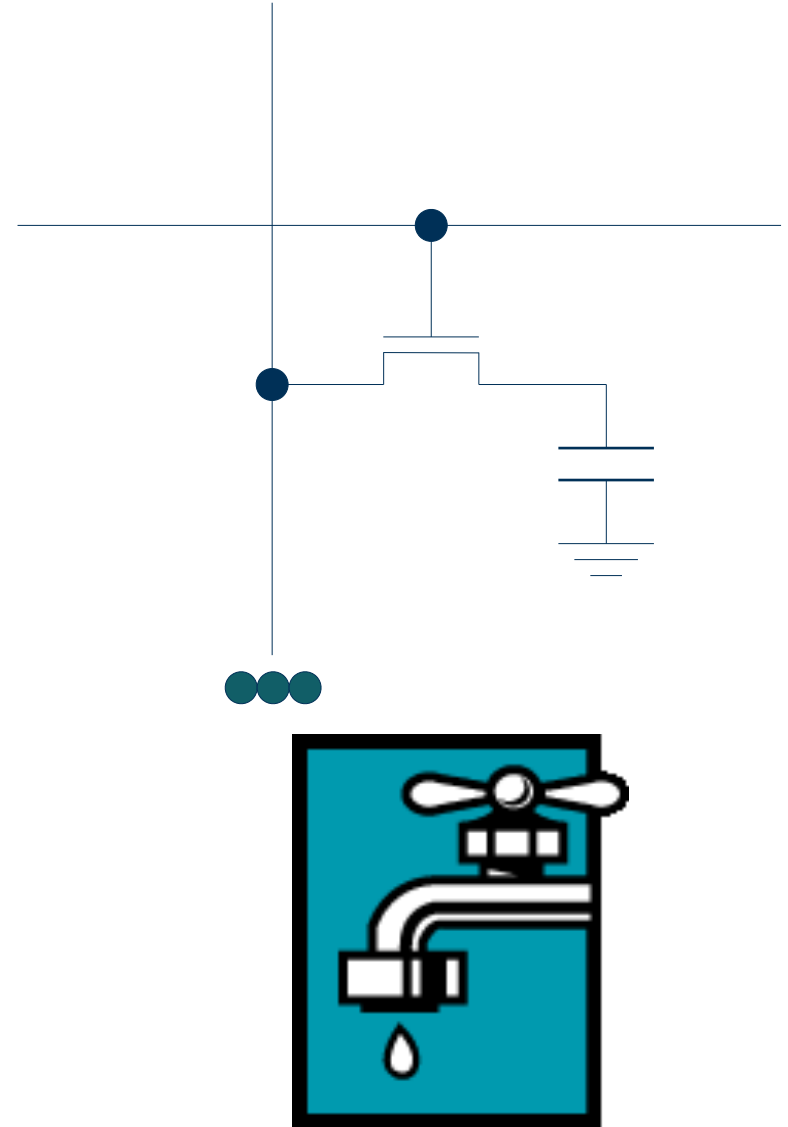
# DRAM Cell

- Capacitors will leak!
  - DRAM cell loses charge over time
  - DRAM cell needs to be **refreshed** periodically.
  - Refresh takes time and power.  
When refreshing can't read the data. A major issue, lots of research going on to reduce the refresh overhead.



# DRAM Cell

- Capacitors will leak!
  - DRAM cell loses charge over time
  - DRAM cell needs to be **refreshed** periodically.
  - Refresh takes time and power.  
When refreshing can't read the data. A major issue, lots of research going on to reduce the refresh overhead.



# Latch vs. DRAM vs. SRAM

- DFF
  - Fastest
  - Low density (27 transistors per bit)
  - High cost
- SRAM
  - Faster access (no capacitor)
  - Lower density (6 transistors per bit; there are designs w/ fewer Ts)
  - Higher cost
  - No need for refresh
  - Manufacturing compatible with logic process (no capacitor)
- DRAM
  - Slower access (capacitor)
  - Higher density (1 transistor + 1 capacitor per bit)
  - Lower cost
  - Requires refresh (power, performance, circuitry)
  - Manufacturing requires putting capacitor and logic together

# Nonvolatile Memories

- DFF, DRAM and SRAM are volatile memories
  - Lose information if powered off.

# Nonvolatile Memories

- DFF, DRAM and SRAM are volatile memories
  - Lose information if powered off.
- Nonvolatile memories retain value even if powered off
  - Flash (~ 5 years)
  - Hard Disk (~ 5 years)
  - Tape (~ 15-30 years)
  - DNA (centuries)

# Nonvolatile Memories

- DFF, DRAM and SRAM
  - Lose information if power is lost
- Nonvolatile memories retain information
  - Flash (~ 5 years)
  - Hard Disk (~ 5 years)
  - Tape (~ 15-30 years)
  - DNA (centuries)

---

## Rewriting Life

---

# Microsoft Has a Plan to Add DNA Data Storage to Its Cloud

Tech companies think biology may solve a looming data storage problem.

by Antonio Regalado    May 22, 2017

**Based on early research involving the storage of movies and documents in DNA**, Microsoft is developing an apparatus that uses biology to replace tape drives, researchers at the company say.

Computer architects at Microsoft Research say the company has formalized a goal of having an operational storage system based on DNA

# Nonvolatile Memories

- DFF, DRAM and SRAM are volatile memories
  - Lose information if powered off.
- Nonvolatile memories retain value even if powered off
  - Flash (~ 5 years)
  - Hard Disk (~ 5 years)
  - Tape (~ 15-30 years)
  - DNA (centuries)
- Uses for Nonvolatile Memories
  - Firmware (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,...)
  - Files in Smartphones, mp3 players, tablets, laptops
  - Backup



# Summary of Trade-Offs

- Bigger is slower
  - Flip-flops/Small SRAM, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec
- Faster is more expensive (dollars and chip area)
  - SRAM, < 10\$ per Megabyte
  - DRAM, < 1\$ per Megabyte
  - Hard Disk < 1\$ per Gigabyte
- Other technologies have their place as well
  - PC-RAM, MRAM, RRAM

# Hardware Implementation of Large RAMs

- How do we read data from a RAM?

# Hardware Implementation of Large RAMs

- How do we read data from a RAM?
  - Given an address, how does it return the data at the address?

# Hardware Implementation of Large RAMs

- How do we read data from a RAM?
  - Given an address, how does it return the data at the address?
- Simplest solution: read data **at all addresses**, and then use a MUX to select the data based on the address.

# Hardware Implementation of Large RAMs

- How do we read data from a RAM?
  - Given an address, how does it return the data at the address?
- Simplest solution: read data **at all addresses**, and then use a MUX to select the data based on the address.
  - This is how the register file is implemented.

# Hardware Implementation of Large RAMs

- How do we read data from a RAM?
  - Given an address, how does it return the data at the address?
- Simplest solution: read data **at all addresses**, and then use a MUX to select the data based on the address.
  - This is how the register file is implemented.
- Downsides?

# Hardware Implementation of Large RAMs

- How do we read data from a RAM?
  - Given an address, how does it return the data at the address?
- Simplest solution: read data **at all addresses**, and then use a MUX to select the data based on the address.
  - This is how the register file is implemented.
- Downsides?
  - It reads all the data at the same time; very inefficient (slow and power hungry) if the RAM is big.

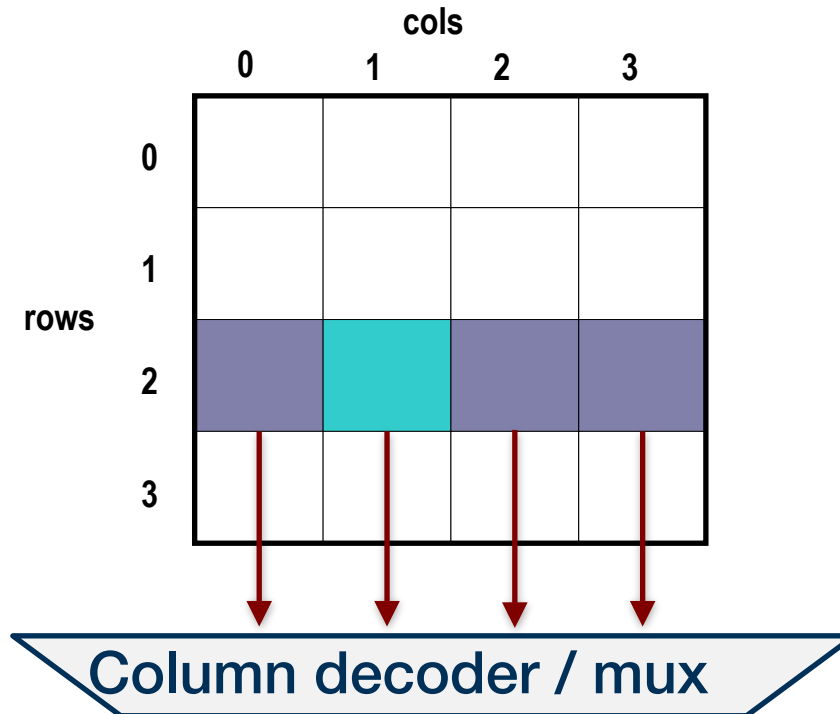
# Hardware Implementation of Large RAMs

- How do we read data from a RAM?
  - Given an address, how does it return the data at the address?
- Simplest solution: read data **at all addresses**, and then use a MUX to select the data based on the address.
  - This is how the register file is implemented.
- Downsides?
  - It reads all the data at the same time; very inefficient (slow and power hungry) if the RAM is big.
  - Register file is at most hundreds of bytes; main memory could be hundreds of GBs!



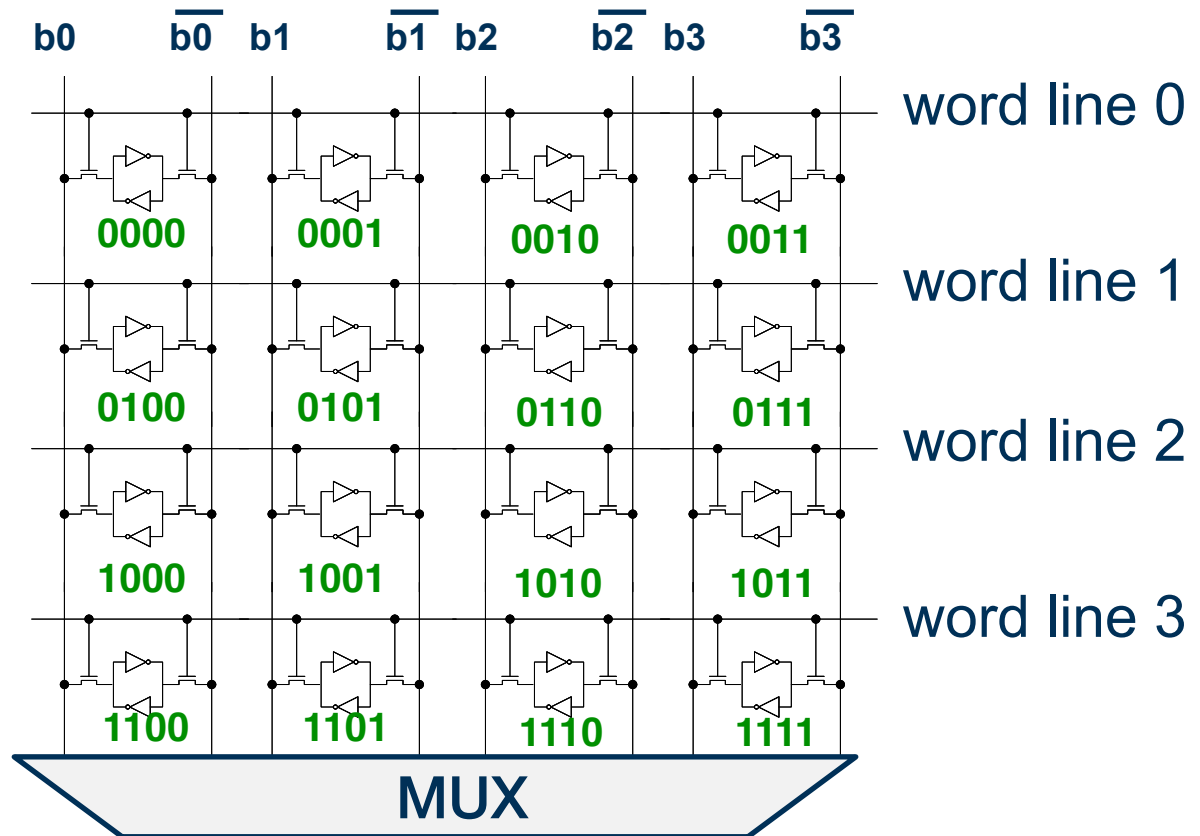
# Hardware Implementation of Large RAMs

- Organize a RAM as a 2D array.
- We first select a row to activate, and read all the data in that row. No other rows' data will be read.
- Then use a MUX to select the data from the activated row.



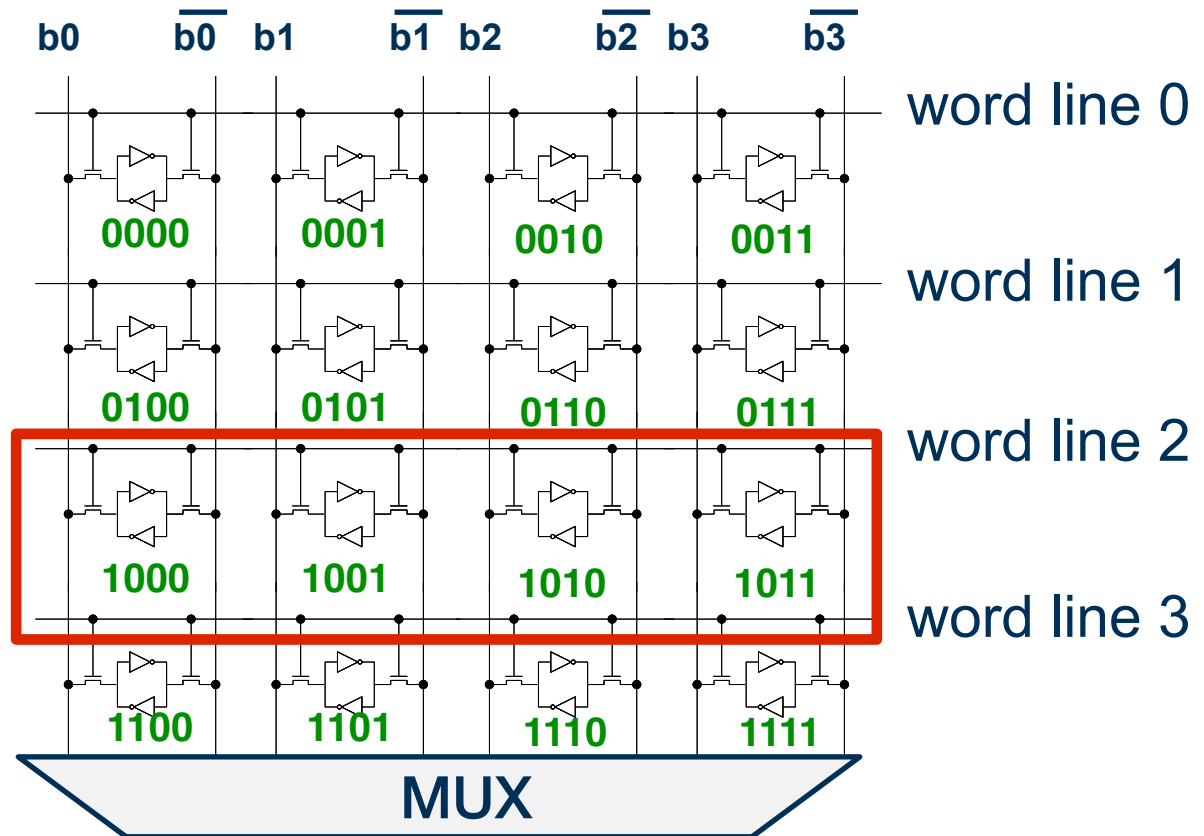
# An SRAM Array (So Far)

- Each cell stores data at a particular address.

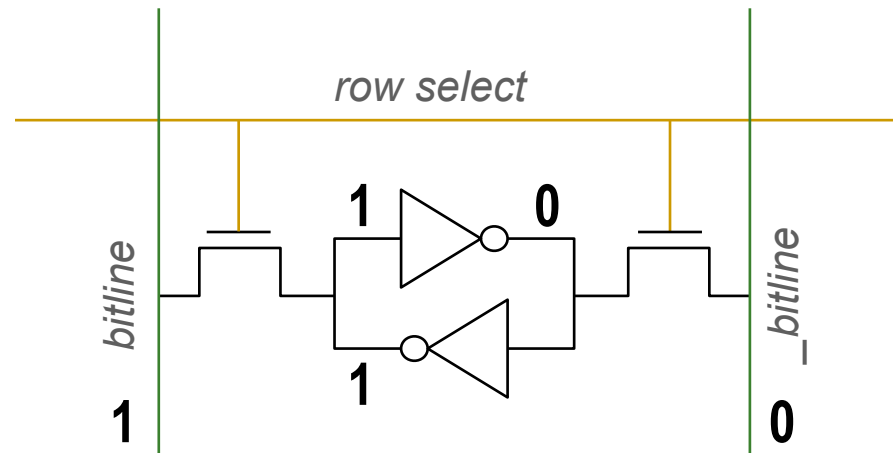


# An SRAM Array (So Far)

- Each cell stores data at a particular address.
- How do we make sure we activate only one row?

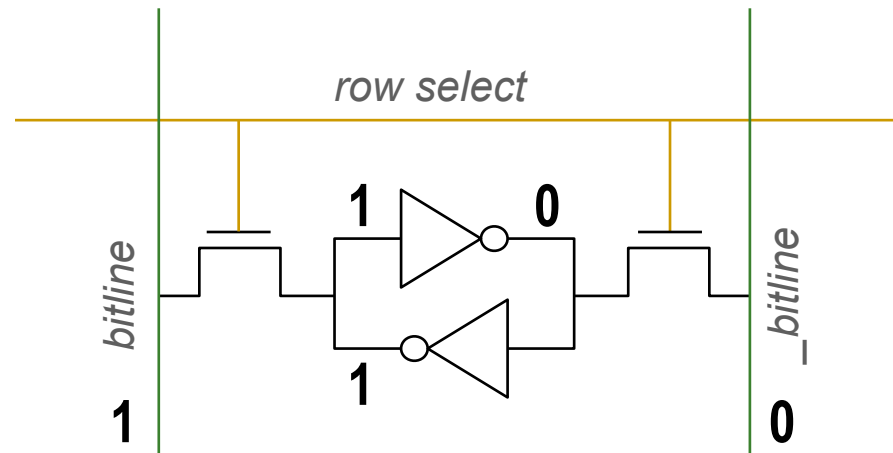


# Recall: (De)Activating an SRAM Cell



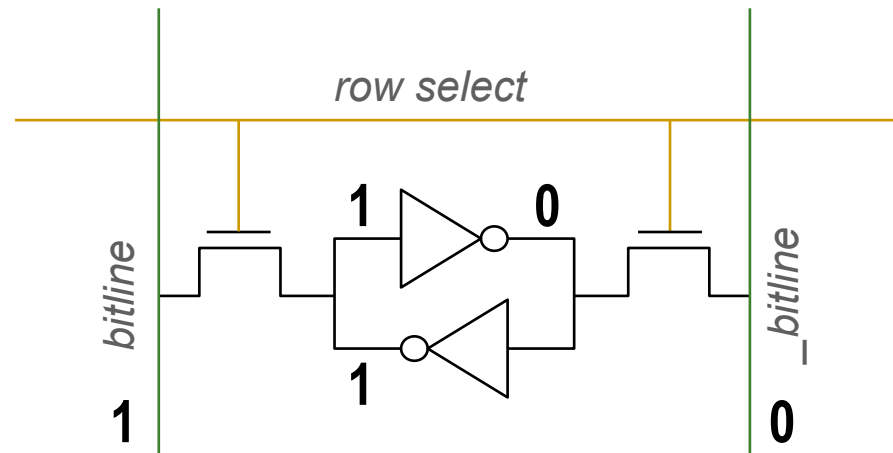
# Recall: (De)Activating an SRAM Cell

- If the row select signal is 1, the two NMOS transistors act as two closed switches. We can read the cell data.

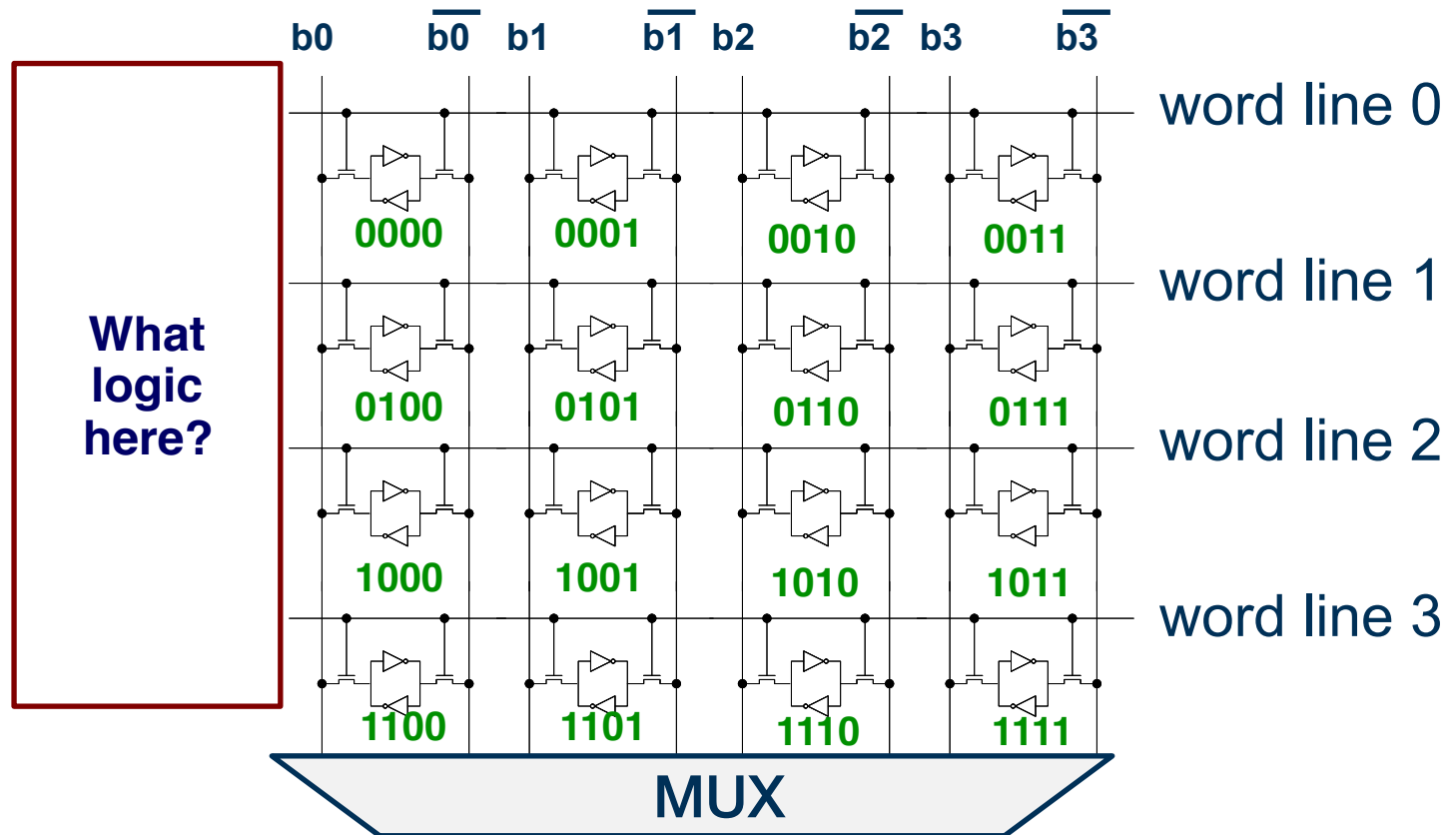


# Recall: (De)Activating an SRAM Cell

- If the row select signal is 1, the two NMOS transistors act as two closed switches. We can read the cell data.
- If the row select signal is 0, the two NMOS transistors act as two open switches, deactivating the access to the cell.



# (De)Activating an SRAM Row



# Recall: Decoder

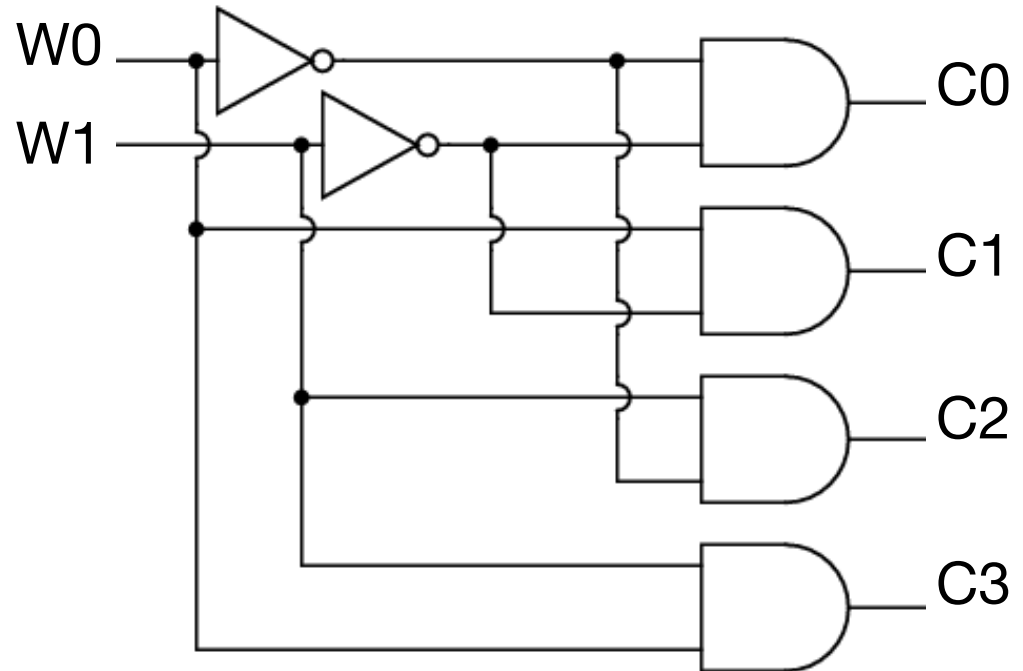
W1	W0	C3	C2	C1	C0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

$$C0 = !W1 \ \& \ !W0$$

$$C1 = !W1 \ \& \ W0$$

$$C2 = W1 \ \& \ !W0$$

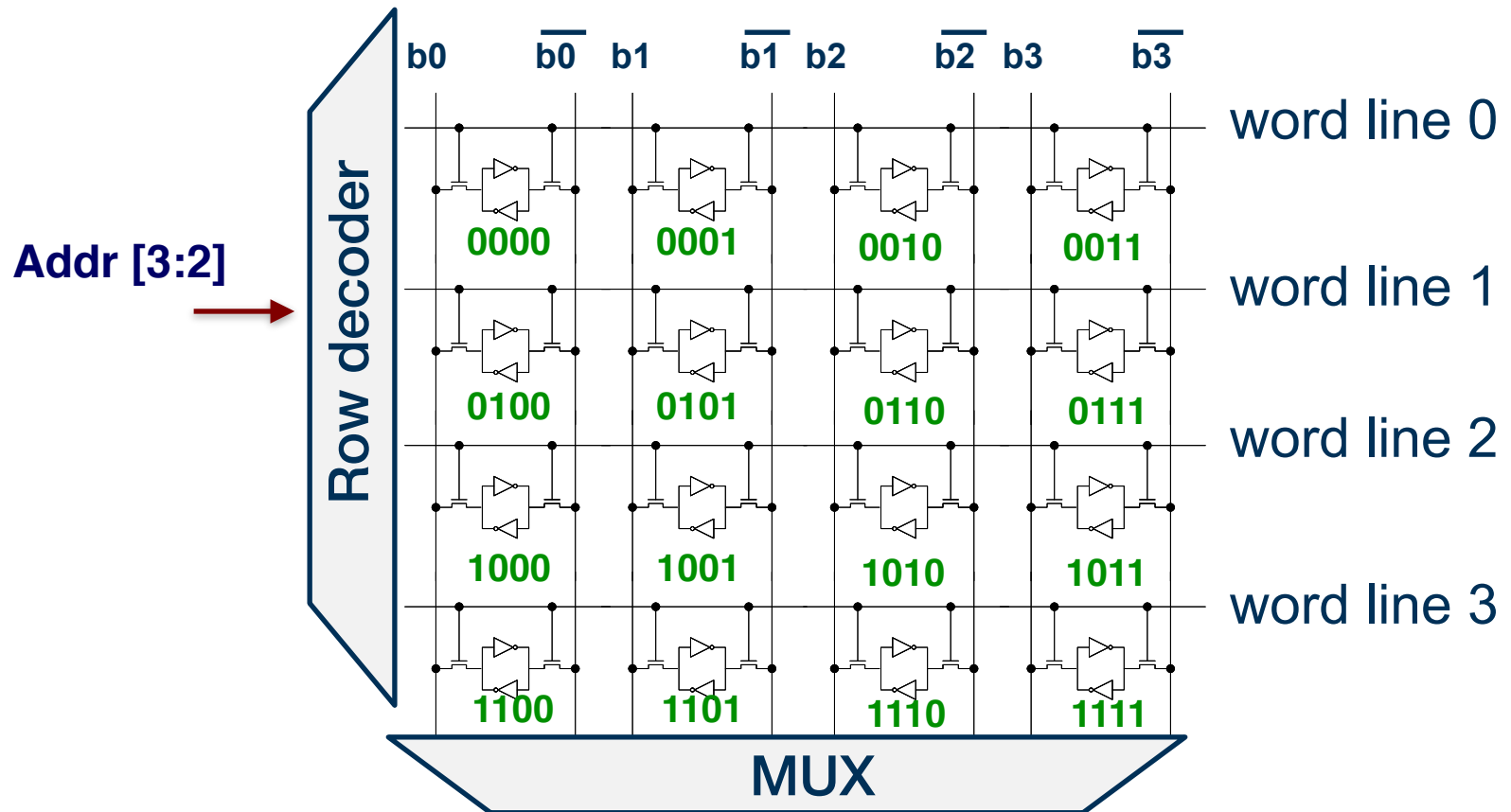
$$C3 = W1 \ \& \ W0$$



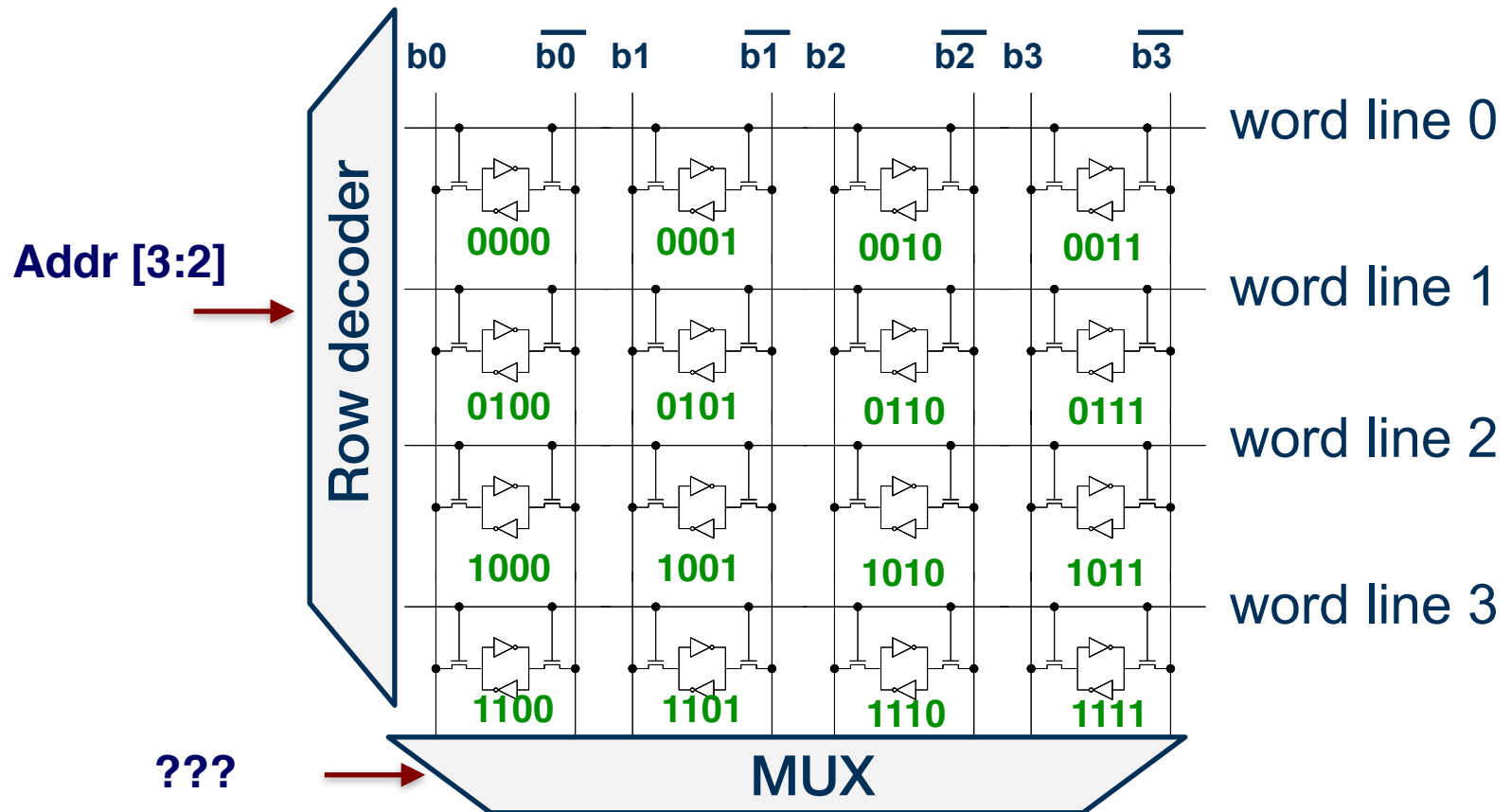


# (De)Activating an SRAM Row

- Use a 2-to-4 decoder.
- The input to the decoder is the first two bits in the address.

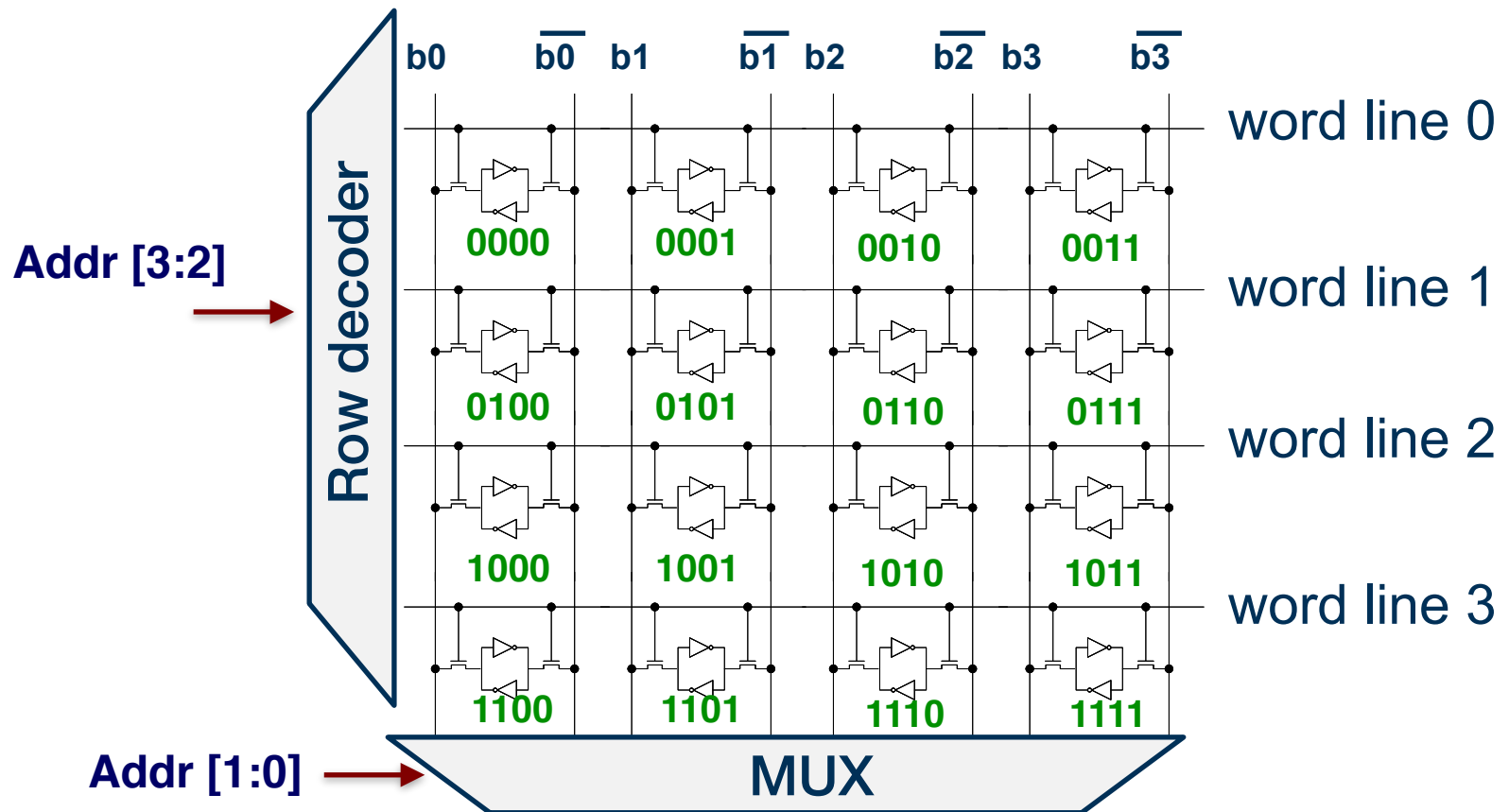


# How to Select From a Row?



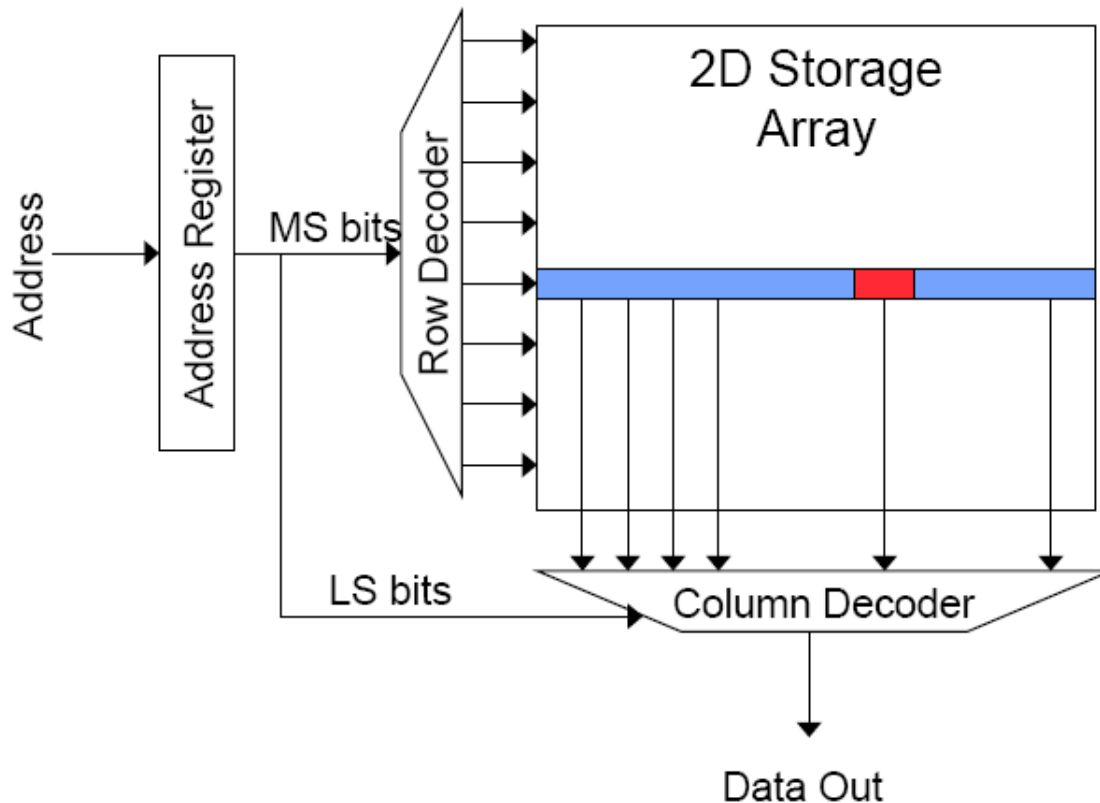
# How to Select From a Row?

- Use the last two bits in the address to operate the MUX.



# Hardware Implementation of Large RAMs

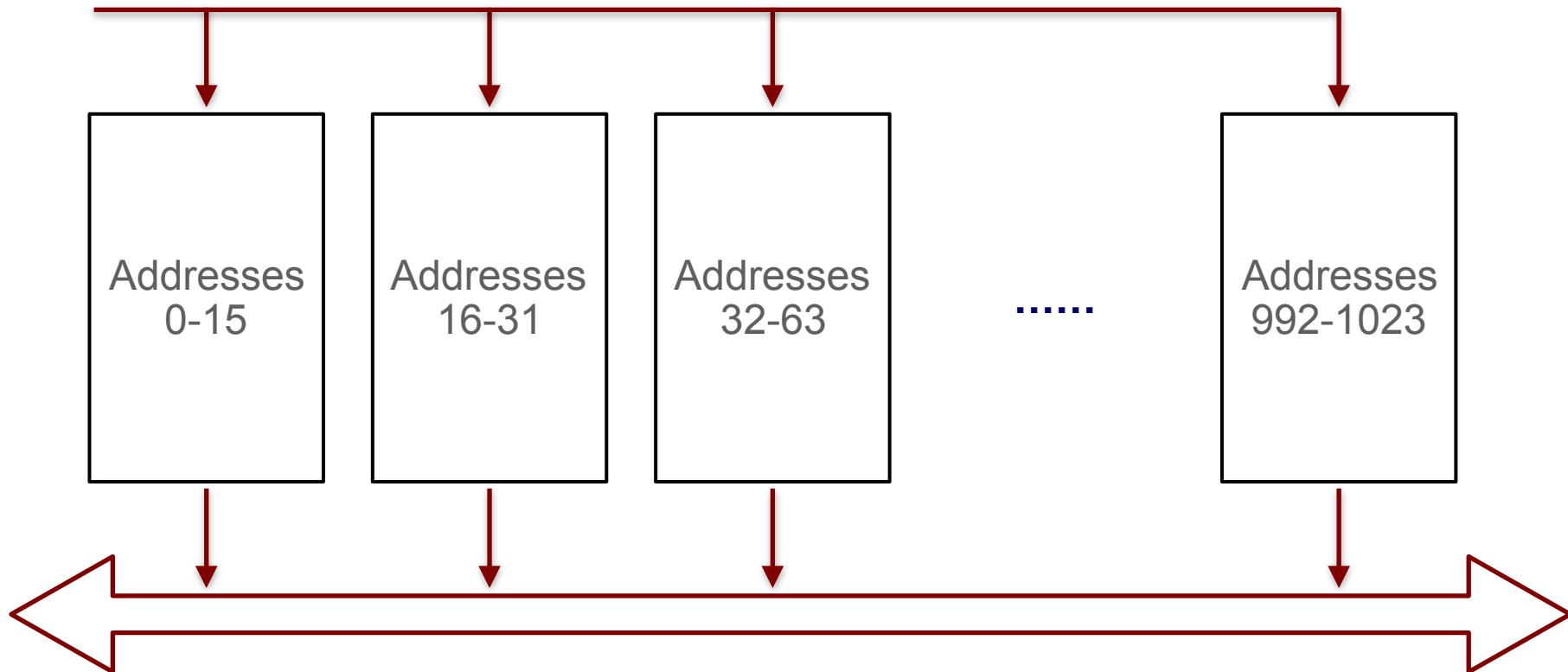
- A RAM is usually organized as a 2D array; both DRAM and SRAM.
- An address split into two: row address and column address.
  - The row address activates a row, and the column address selects data from a row.



# What If a Row is Still too Big?

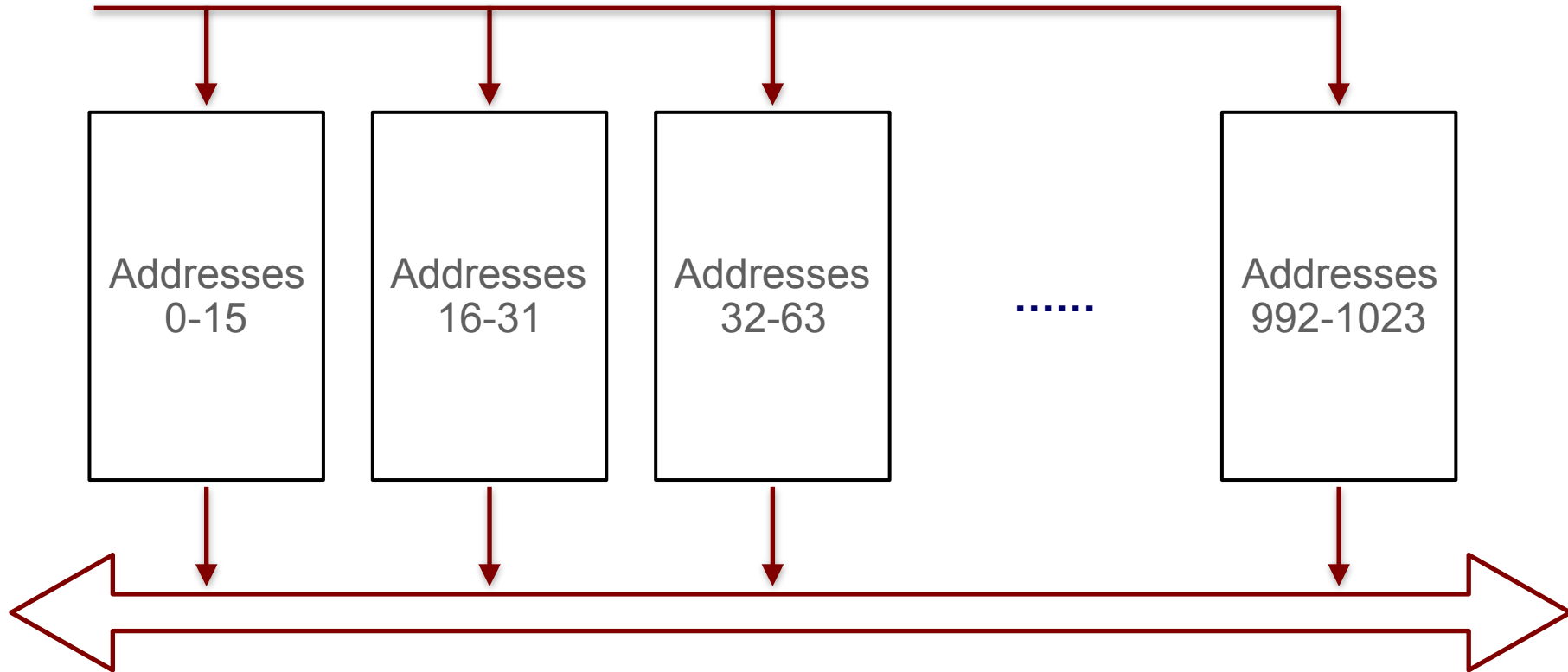
# What If a Row is Still too Big?

Chip Enable (CE)



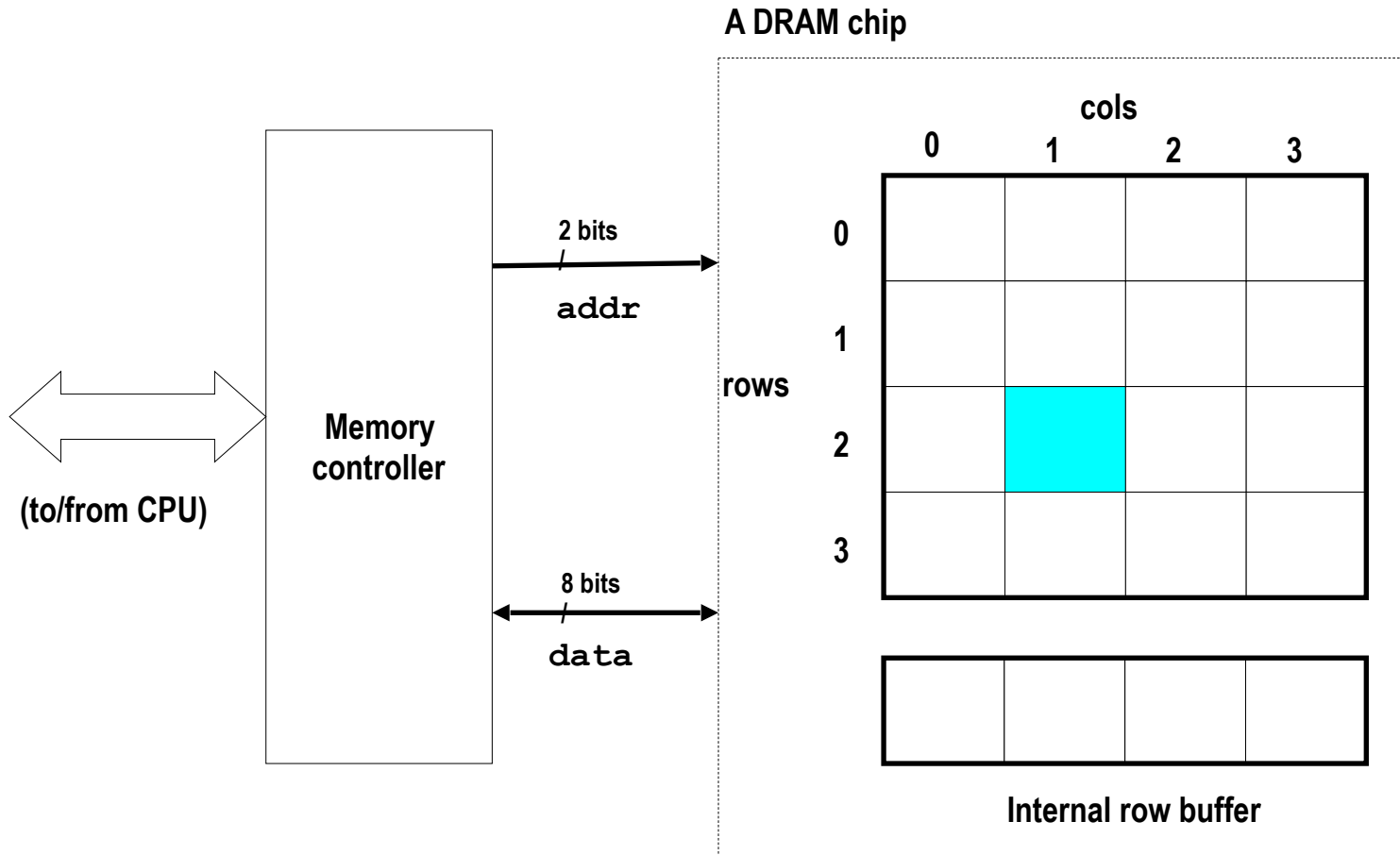
# What If a Row is Still too Big?

## Chip Enable (CE)



- What data go into the CE signal? How does CE disable and enable a chip?

# Aside: DRAM Chip Organization

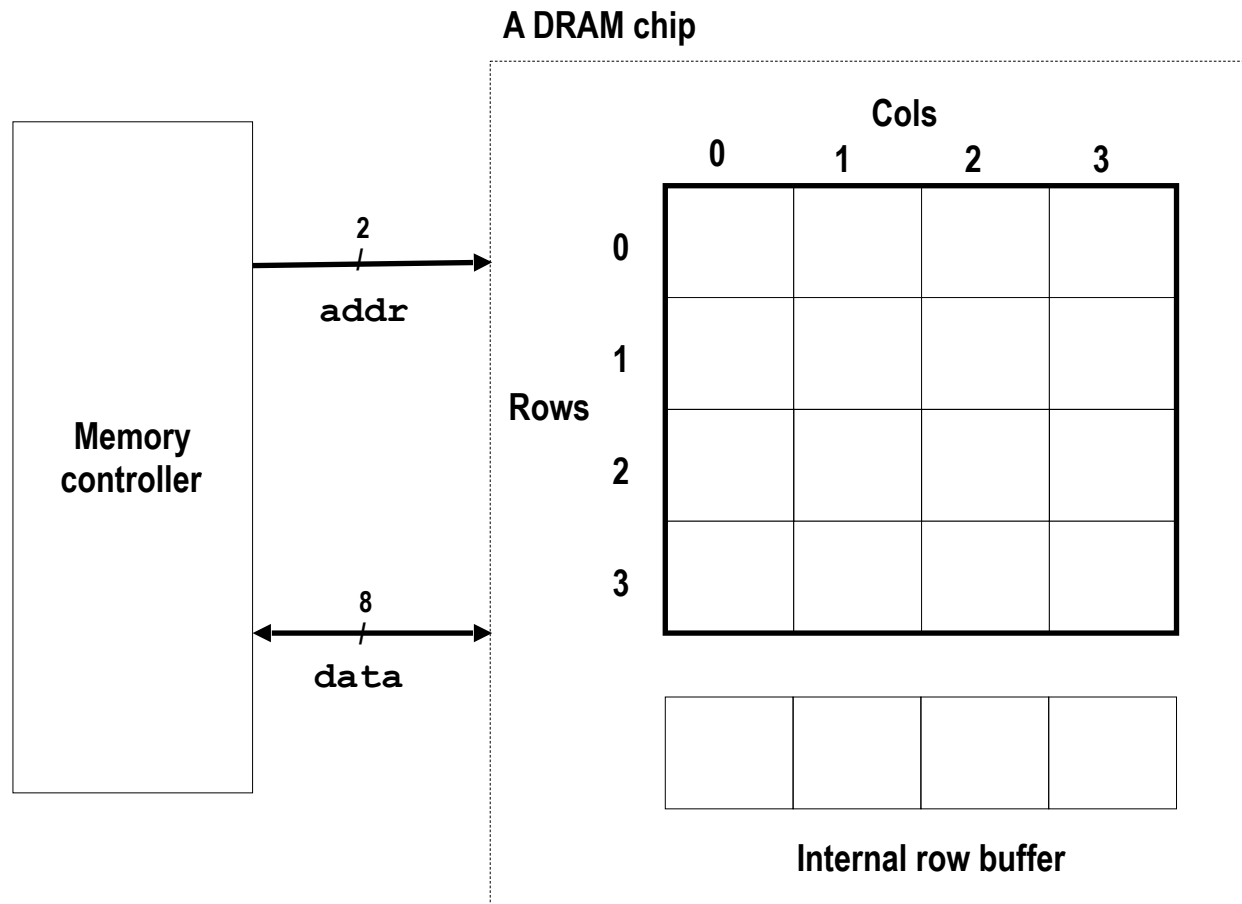




# Aside: Reading DRAM Cell (2,1)

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

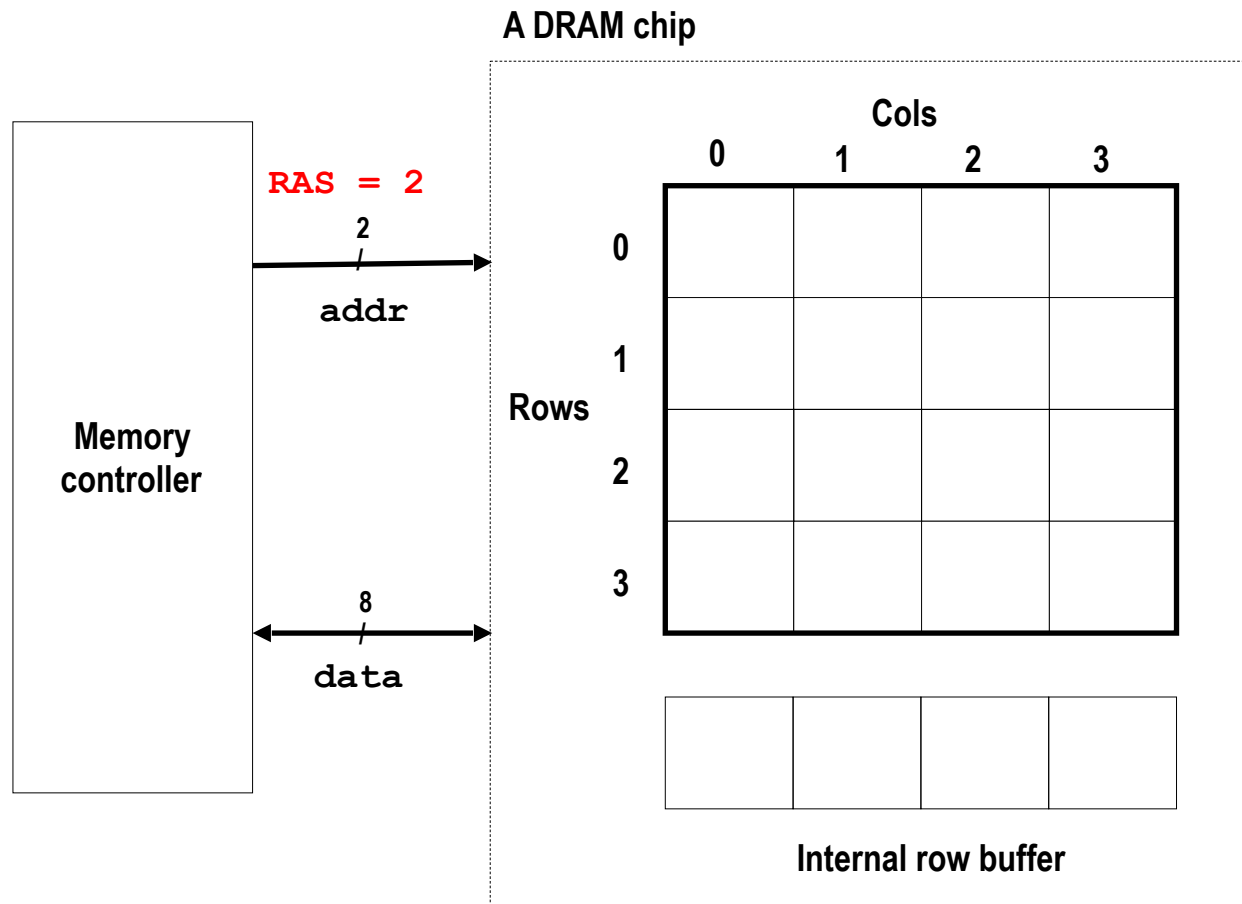
Step 1(b): Row 2 copied from DRAM array to row buffer.



# Aside: Reading DRAM Cell (2,1)

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

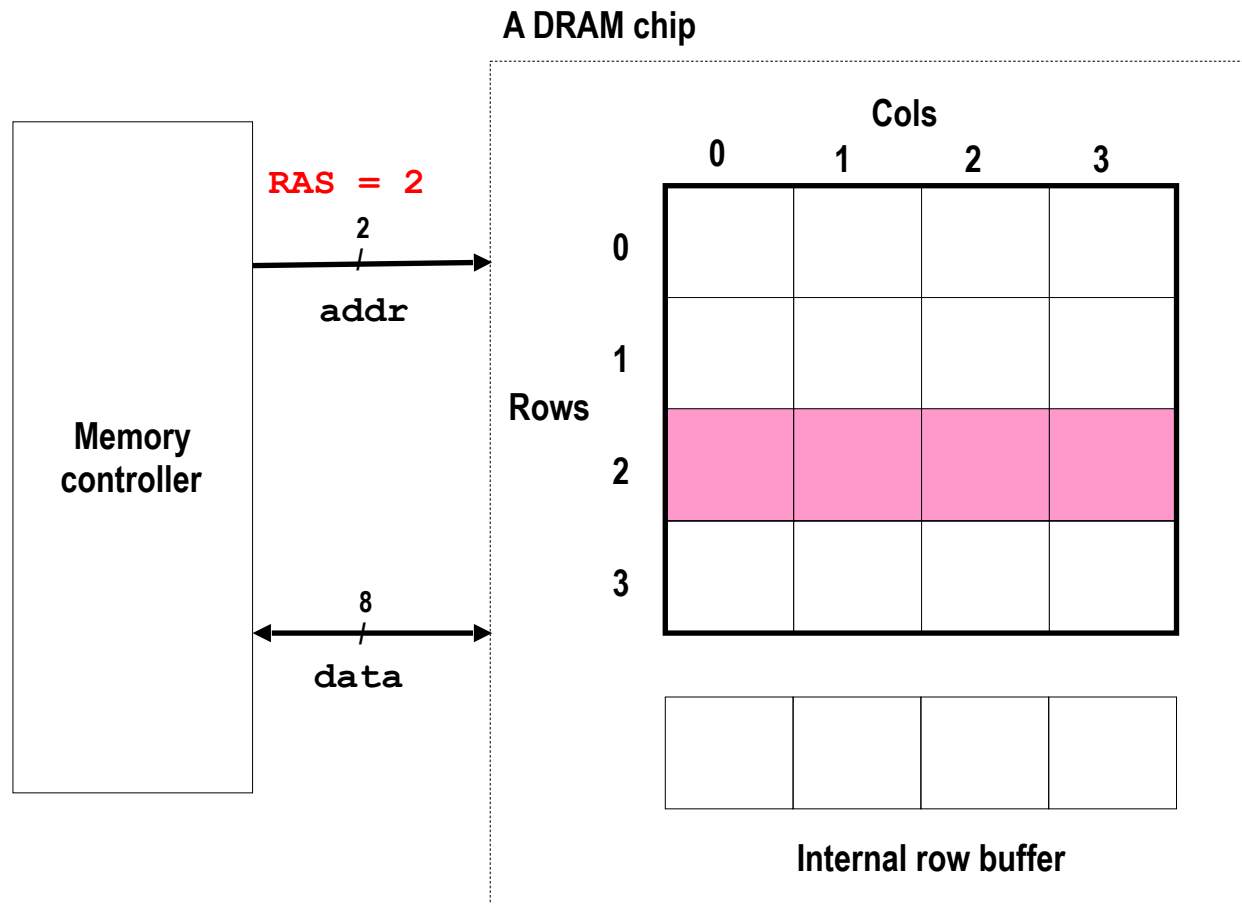
Step 1(b): Row 2 copied from DRAM array to row buffer.



# Aside: Reading DRAM Cell (2,1)

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

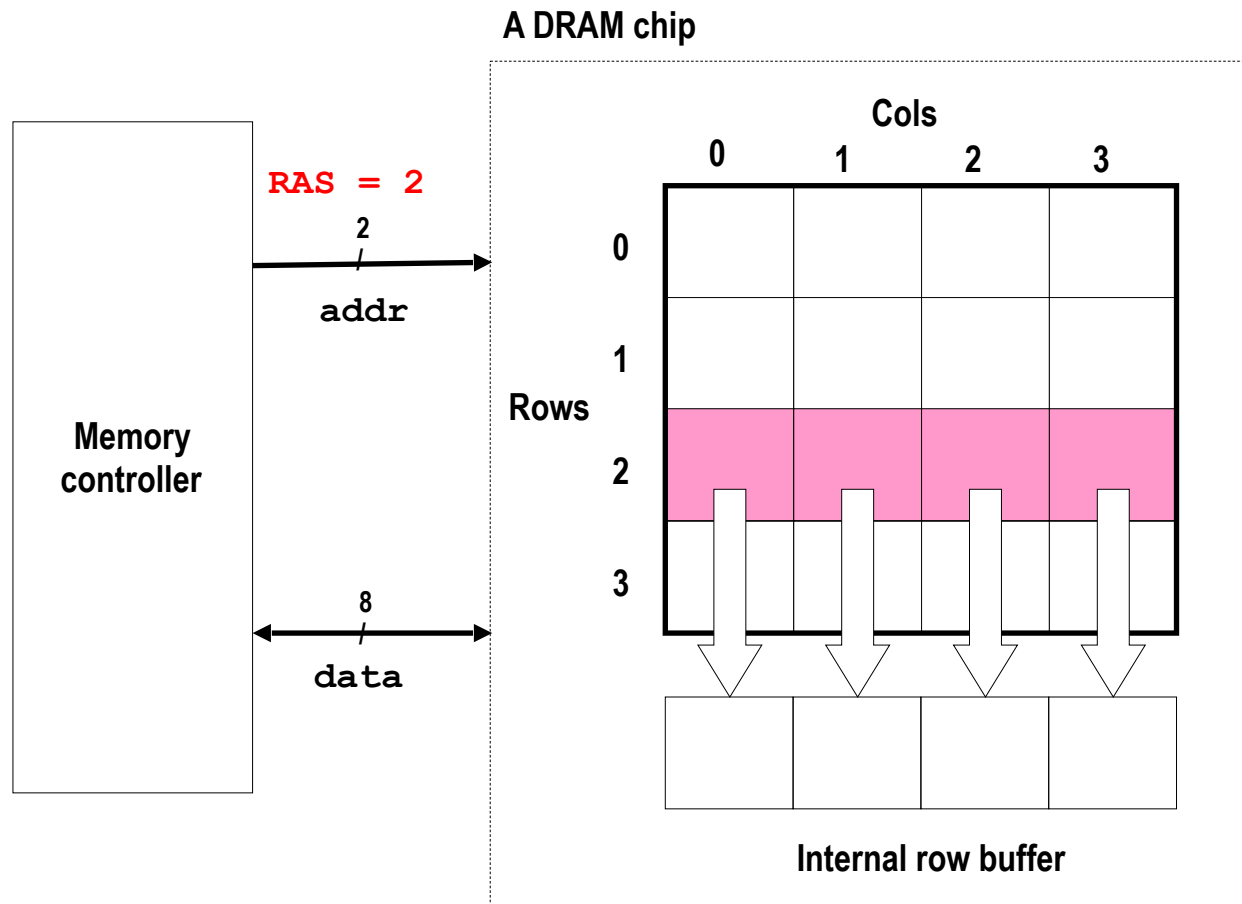
Step 1(b): Row 2 copied from DRAM array to row buffer.



# Aside: Reading DRAM Cell (2,1)

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

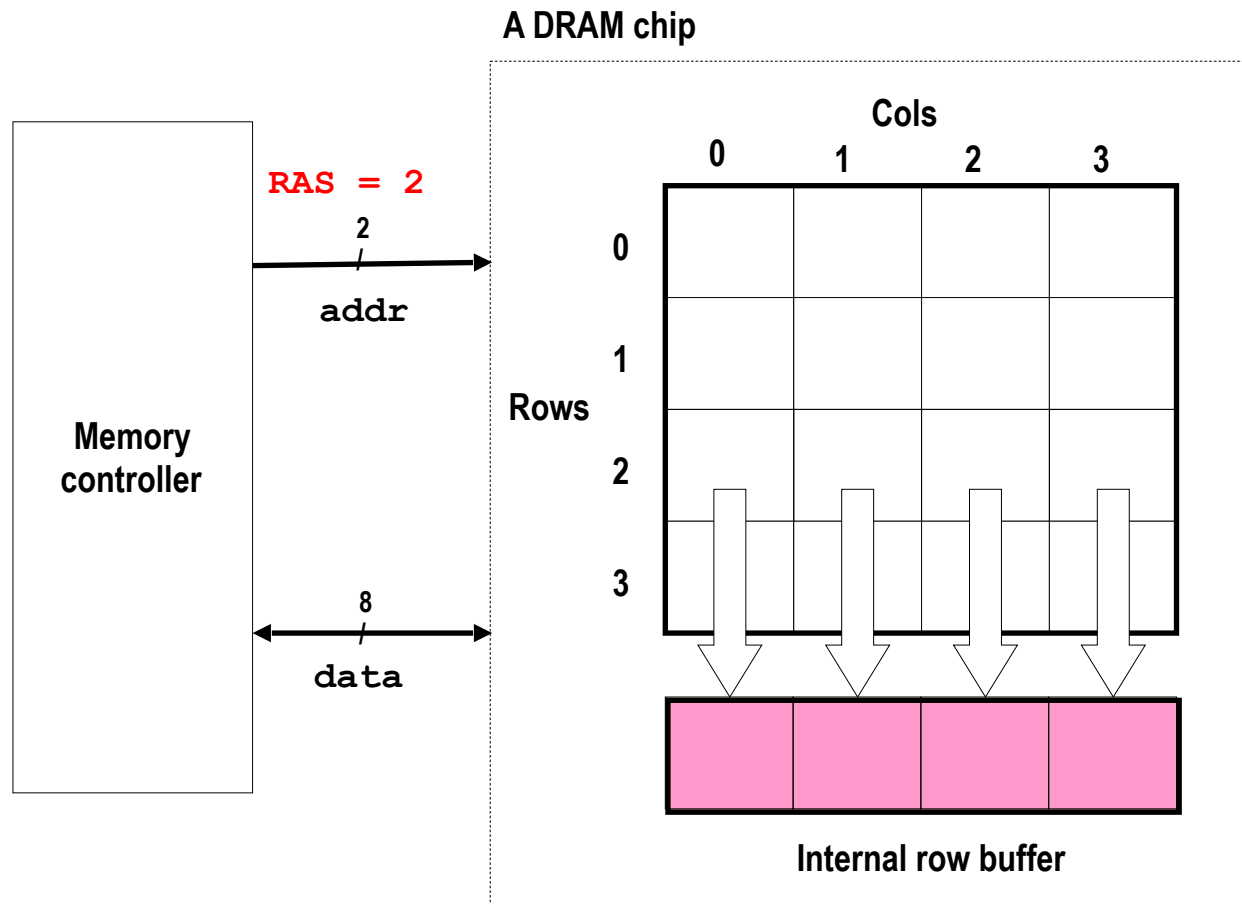
Step 1(b): Row 2 copied from DRAM array to row buffer.



# Aside: Reading DRAM Cell (2,1)

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

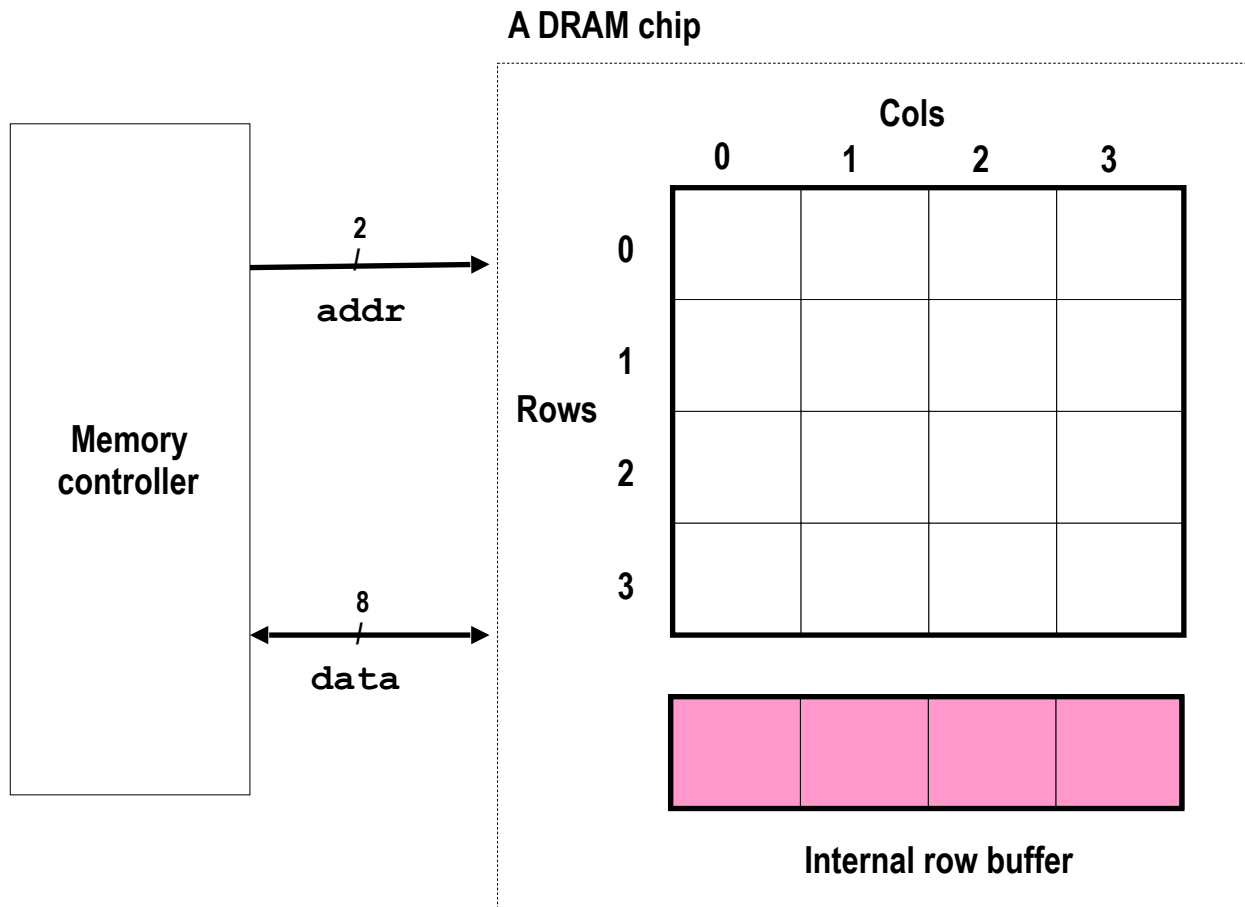
Step 1(b): Row 2 copied from DRAM array to row buffer.



# Aside: Reading DRAM Cell (2,1)

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

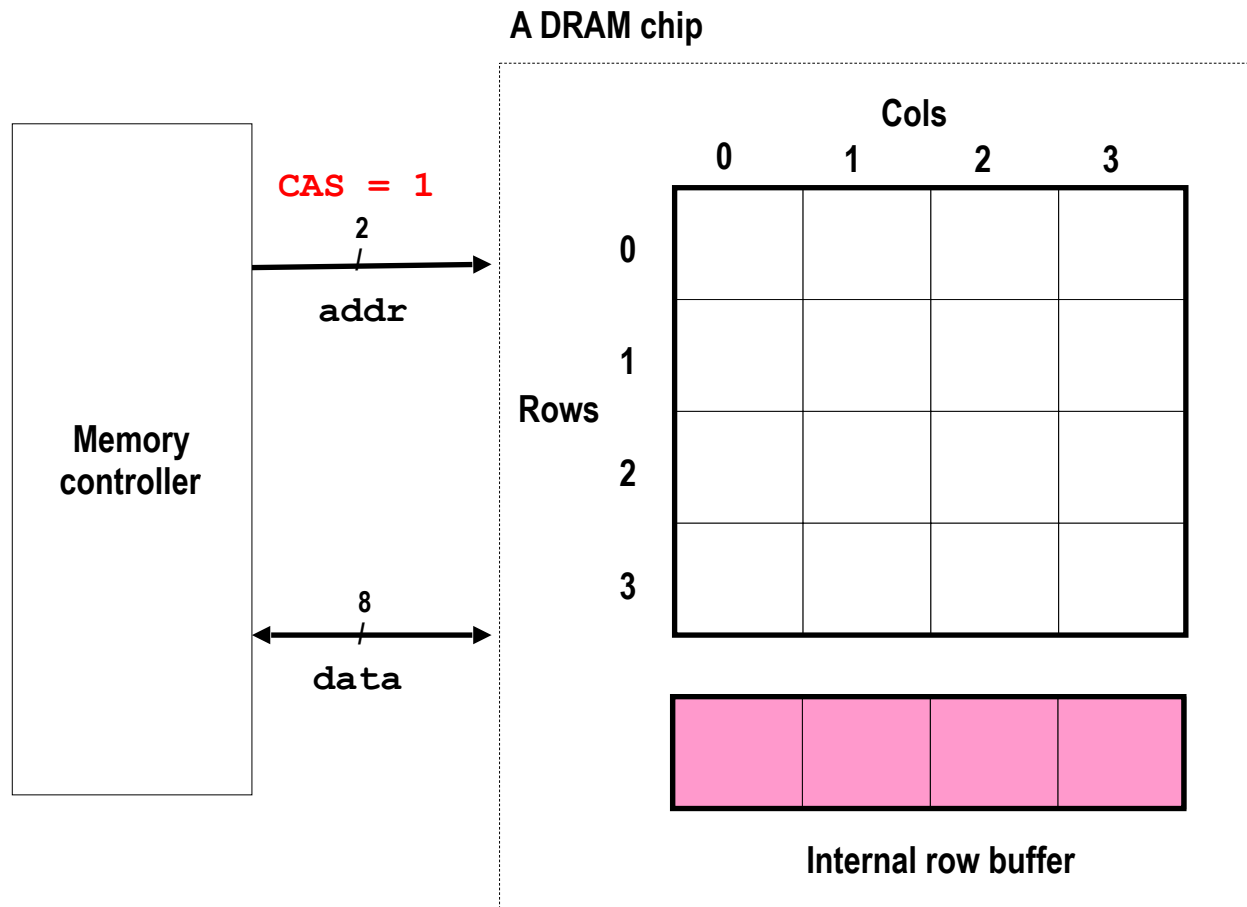
Step 2(b): Cell (2,1) copied from buffer to data lines, and eventually back to the CPU.



# Aside: Reading DRAM Cell (2,1)

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

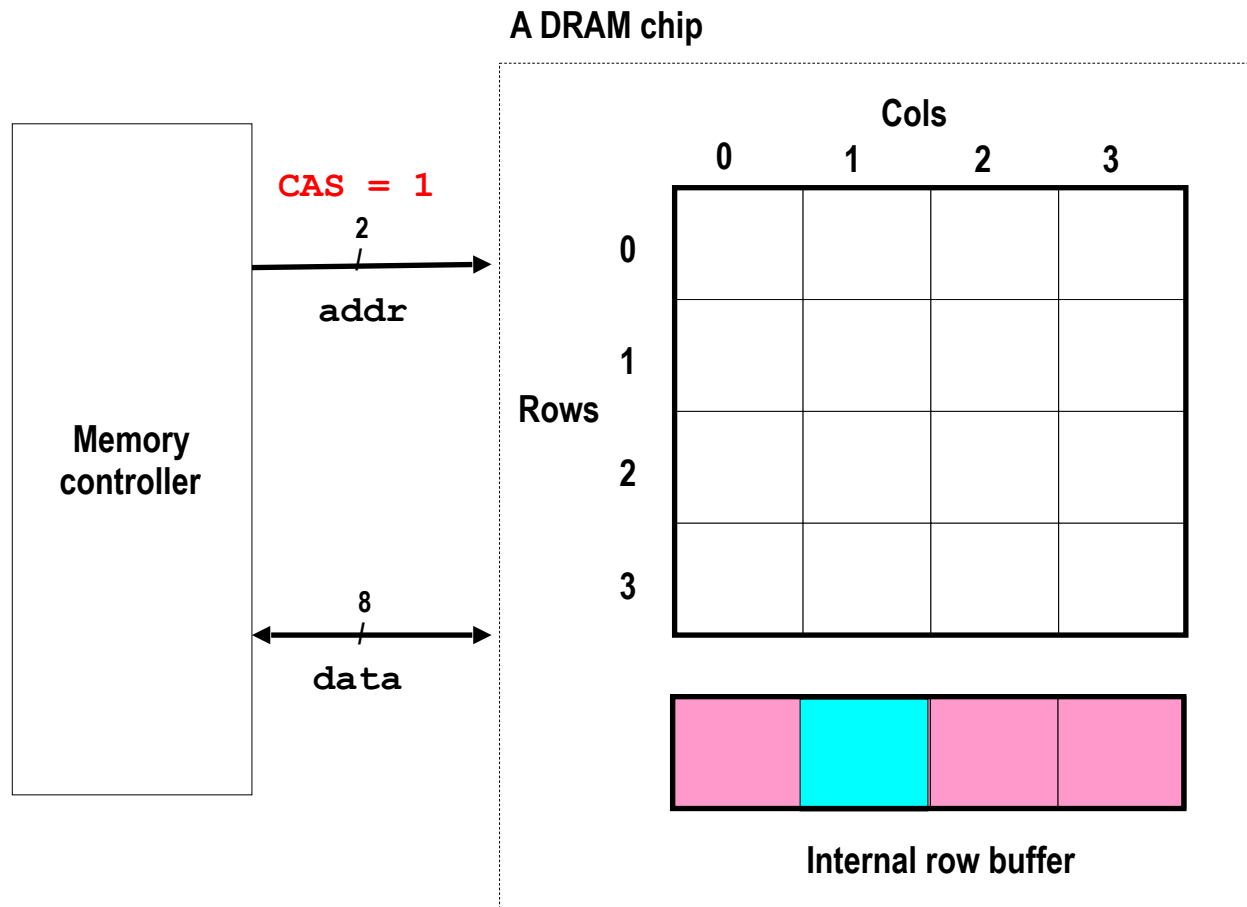
Step 2(b): Cell (2,1) copied from buffer to data lines, and eventually back to the CPU.



# Aside: Reading DRAM Cell (2,1)

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

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

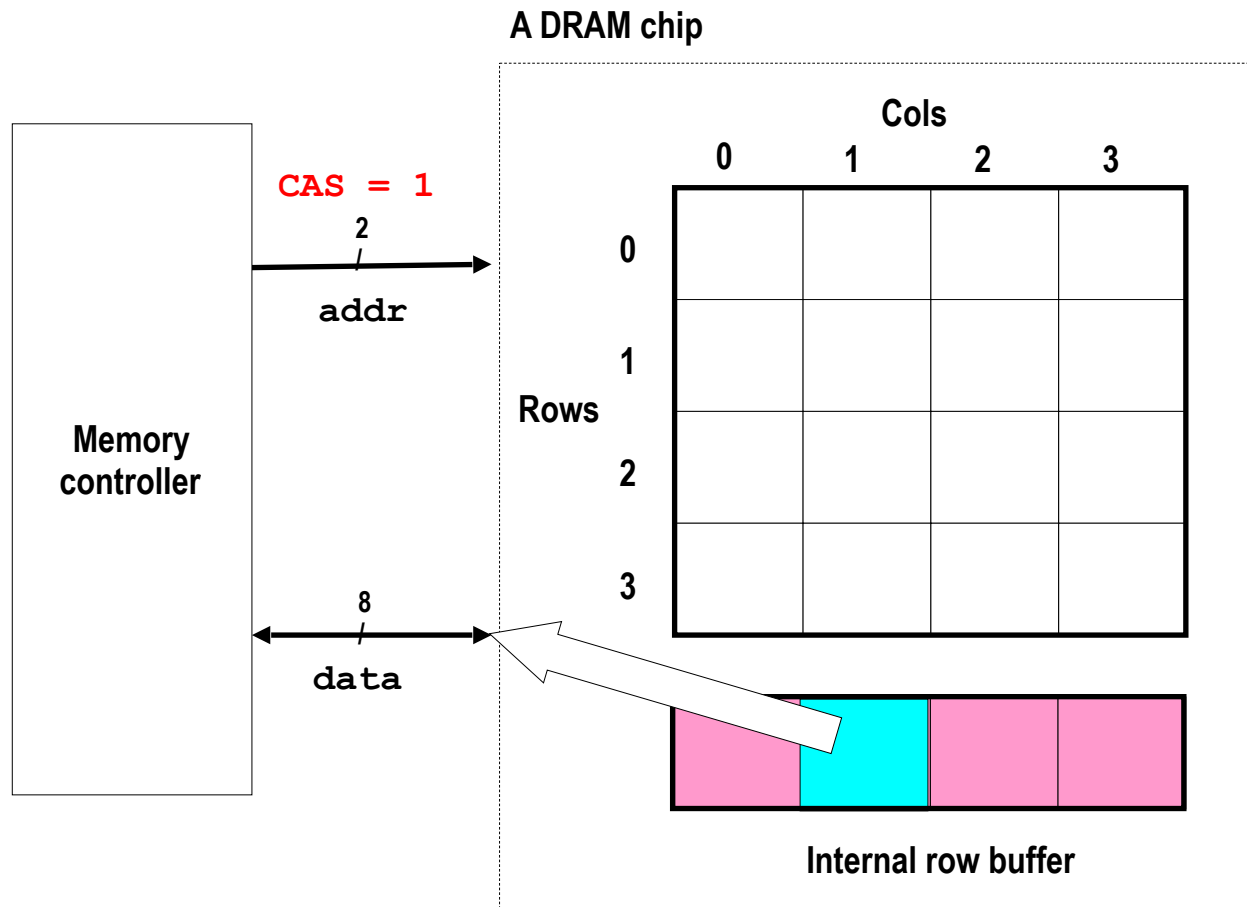




# Aside: Reading DRAM Cell (2,1)

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

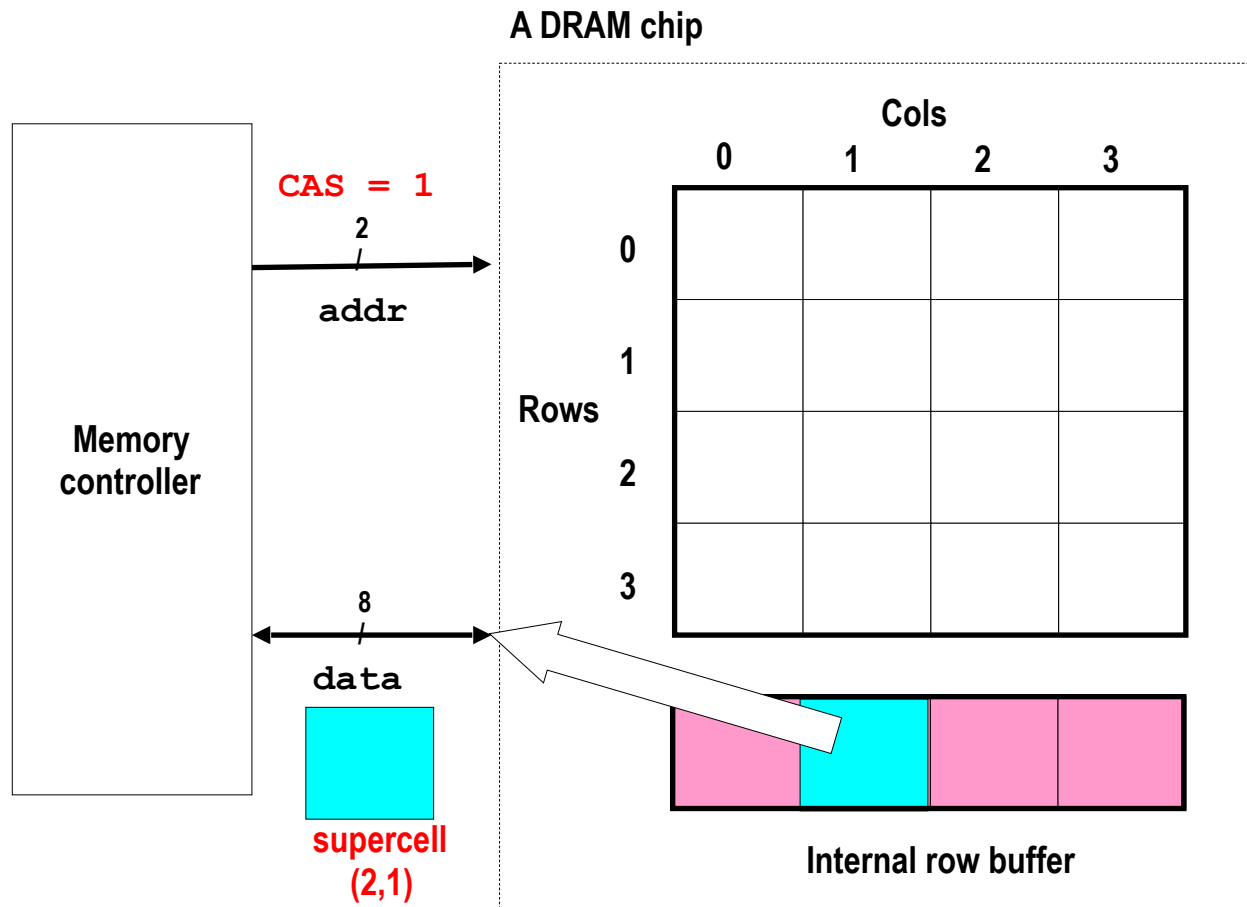
Step 2(b): Cell (2,1) copied from buffer to data lines, and eventually back to the CPU.



# Aside: Reading DRAM Cell (2,1)

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

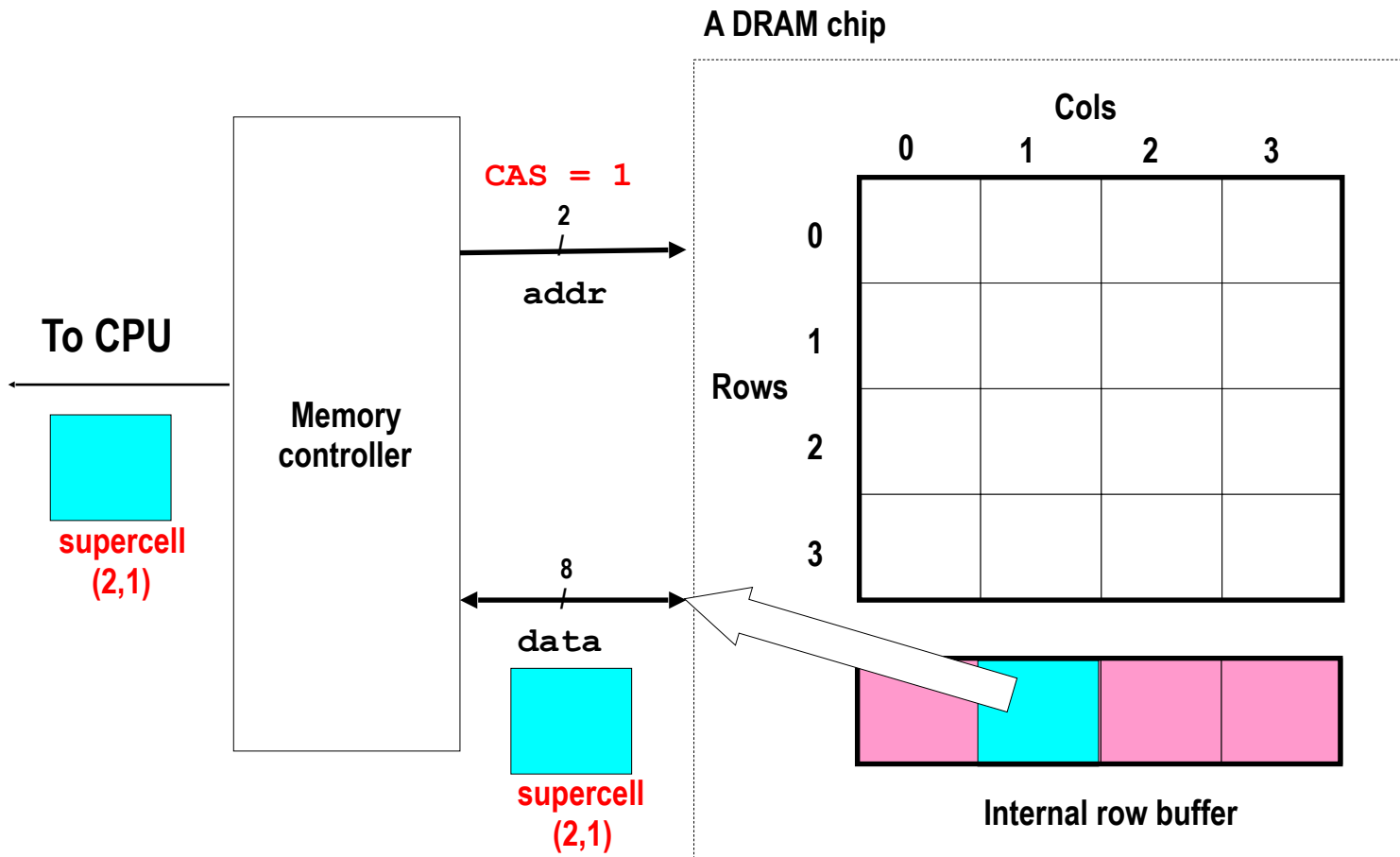
Step 2(b): Cell (2,1) copied from buffer to data lines, and eventually back to the CPU.



# Aside: Reading DRAM Cell (2,1)

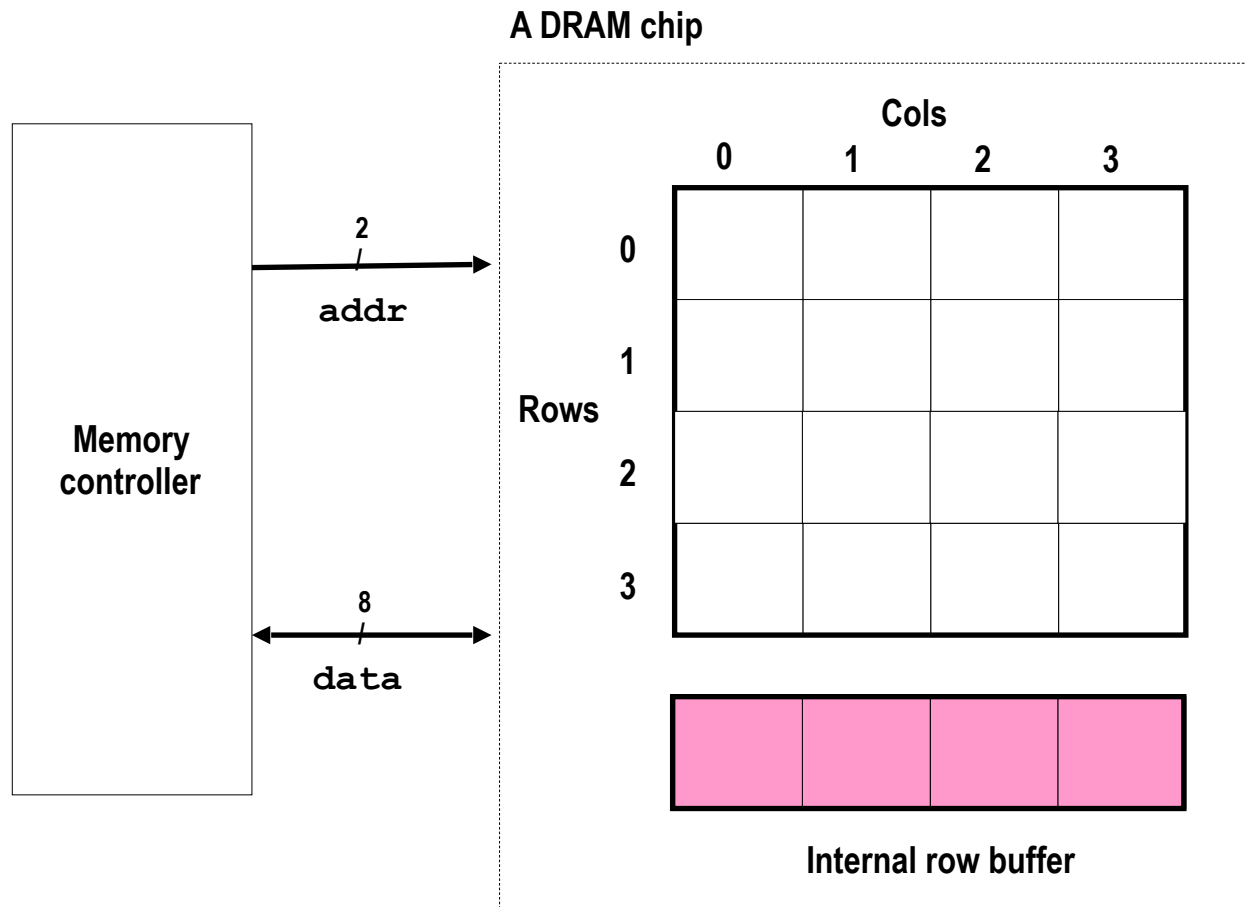
Step 2(a): Column access strobe (**CAS**) selects column 1.

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



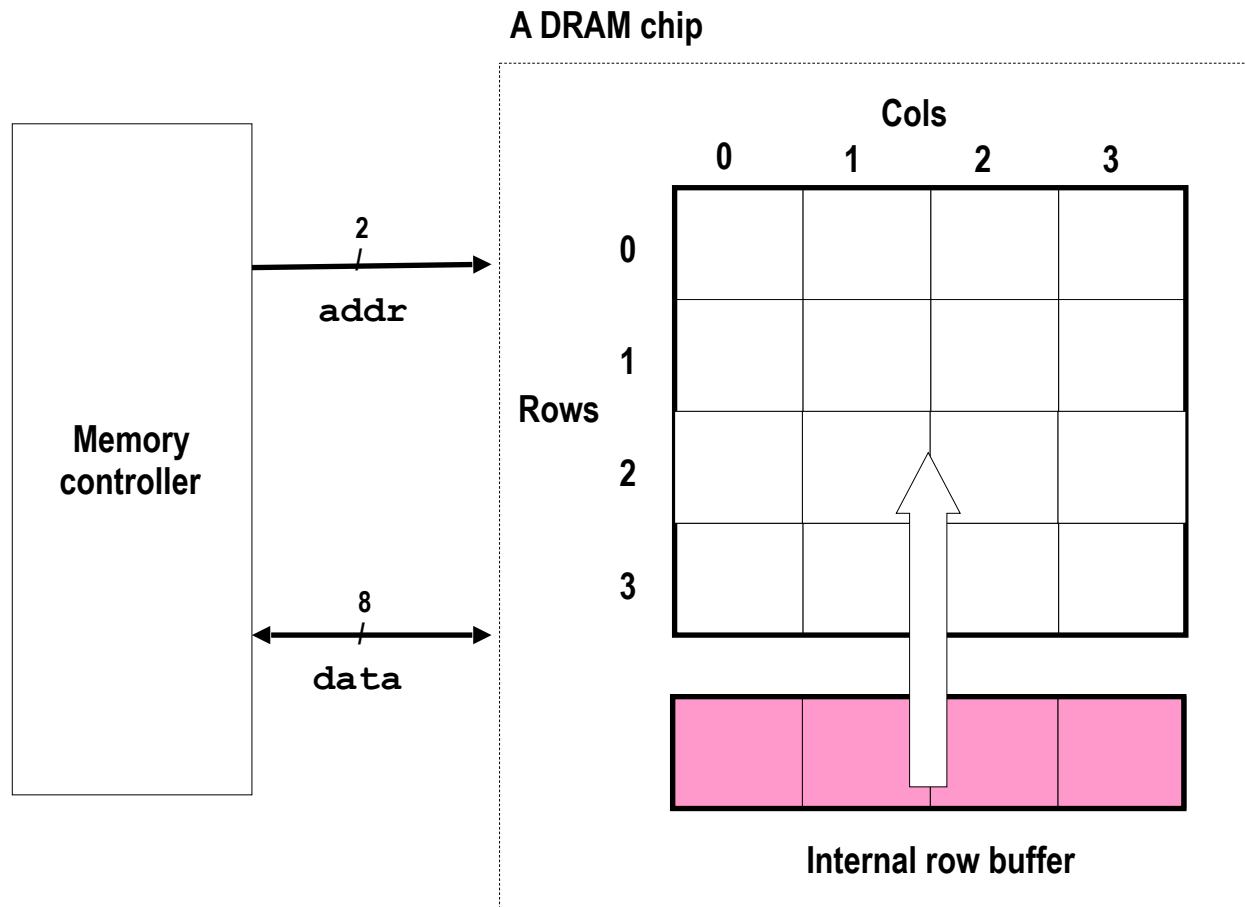
# Aside: Reading DRAM Cell (2,1)

Step 3: A sense amplifier amplifies and regenerates the bitline and refresh the cells. A DRAM controller must periodically read each row within the allowed refresh time (10s of ms) to restore charge.



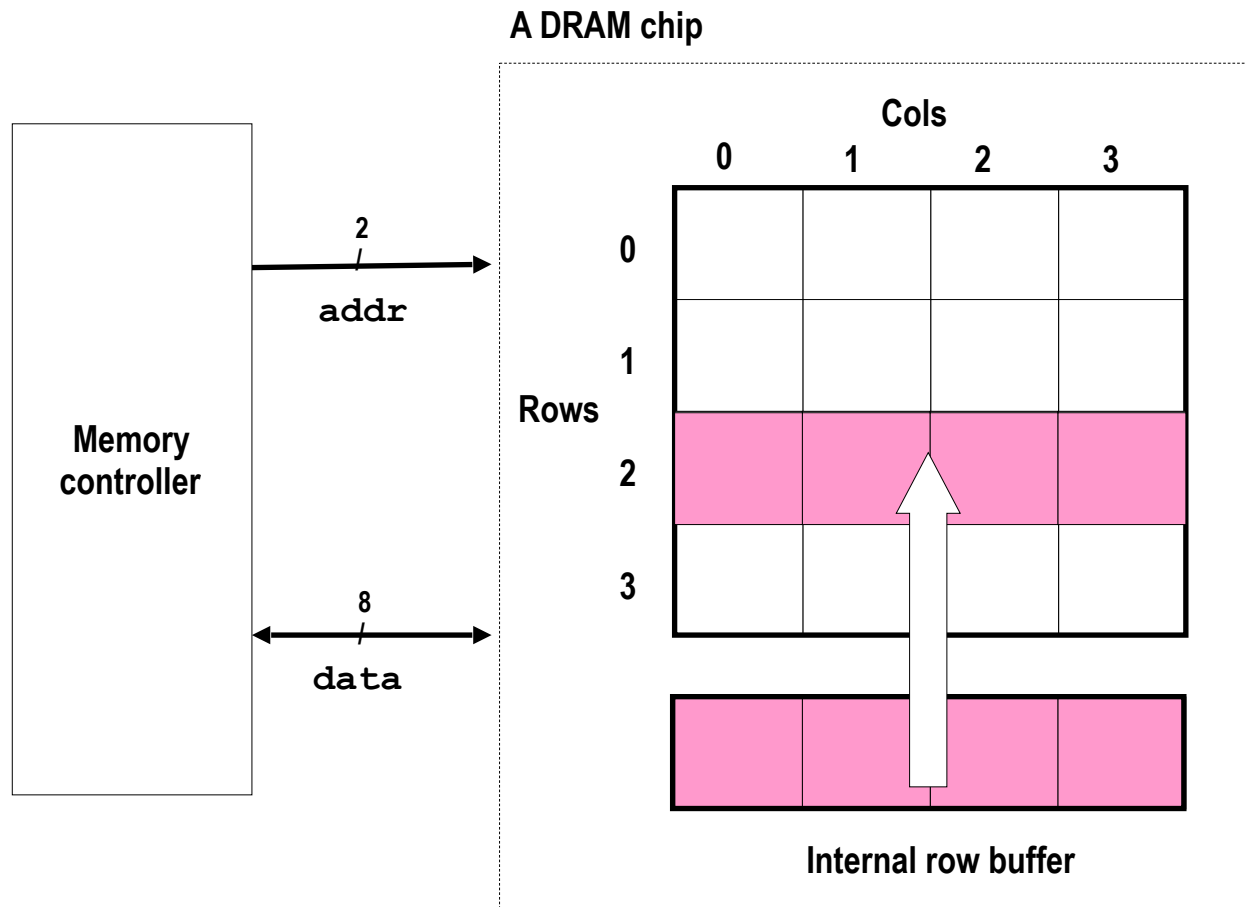
# Aside: Reading DRAM Cell (2,1)

Step 3: A sense amplifier amplifies and regenerates the bitline and refresh the cells. A DRAM controller must periodically read each row within the allowed refresh time (10s of ms) to restore charge.



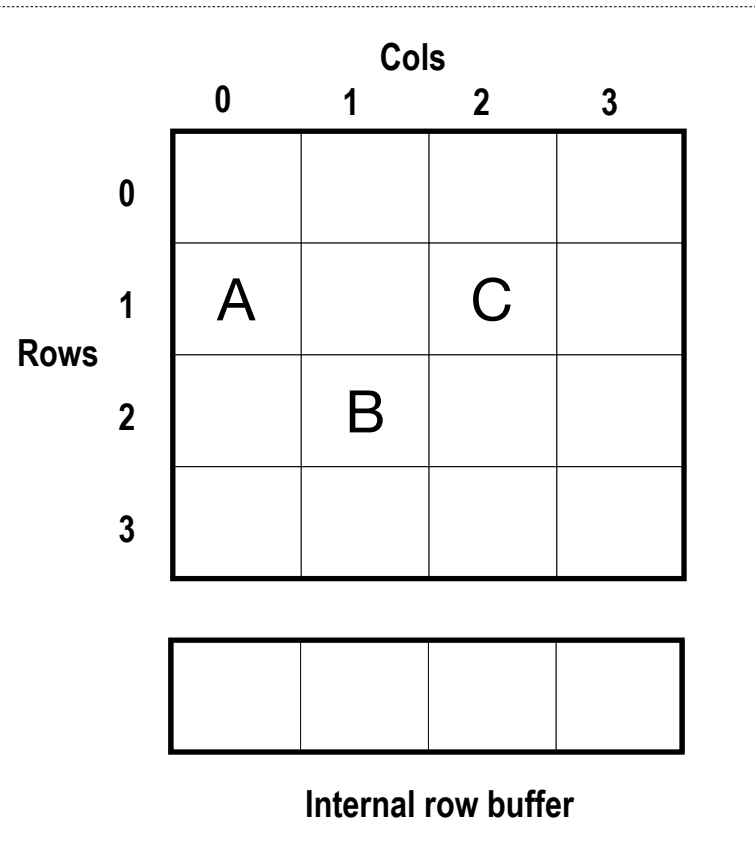
# Aside: Reading DRAM Cell (2,1)

Step 3: A sense amplifier amplifies and regenerates the bitline and refresh the cells. A DRAM controller must periodically read each row within the allowed refresh time (10s of ms) to restore charge.



# Aside: DRAM Scheduling

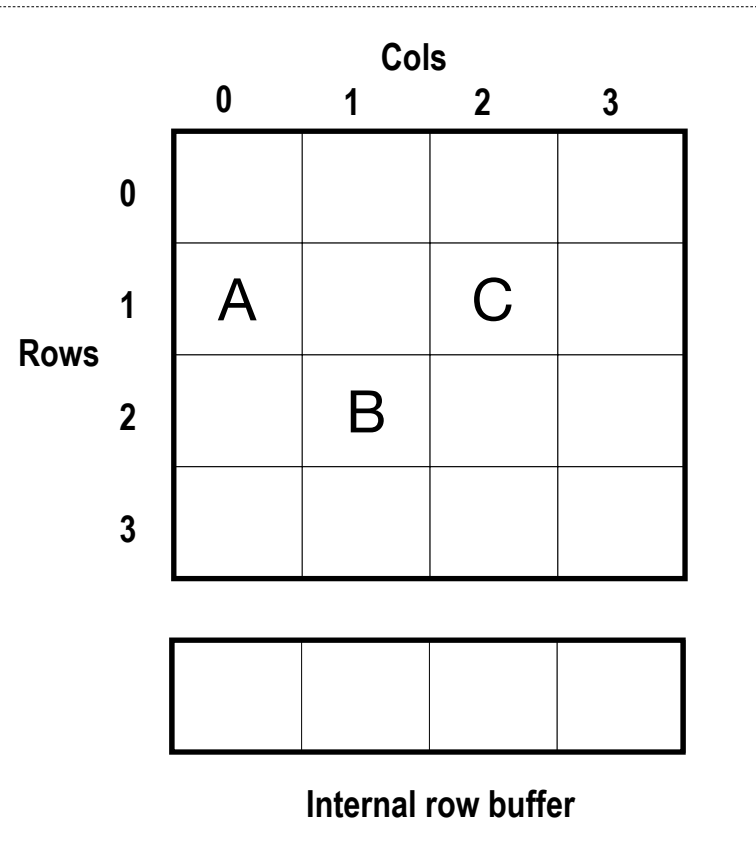
16 x 8 DRAM chip



# Aside: DRAM Scheduling

- Assume the following memory accesses:  
A, B, C

16 x 8 DRAM chip

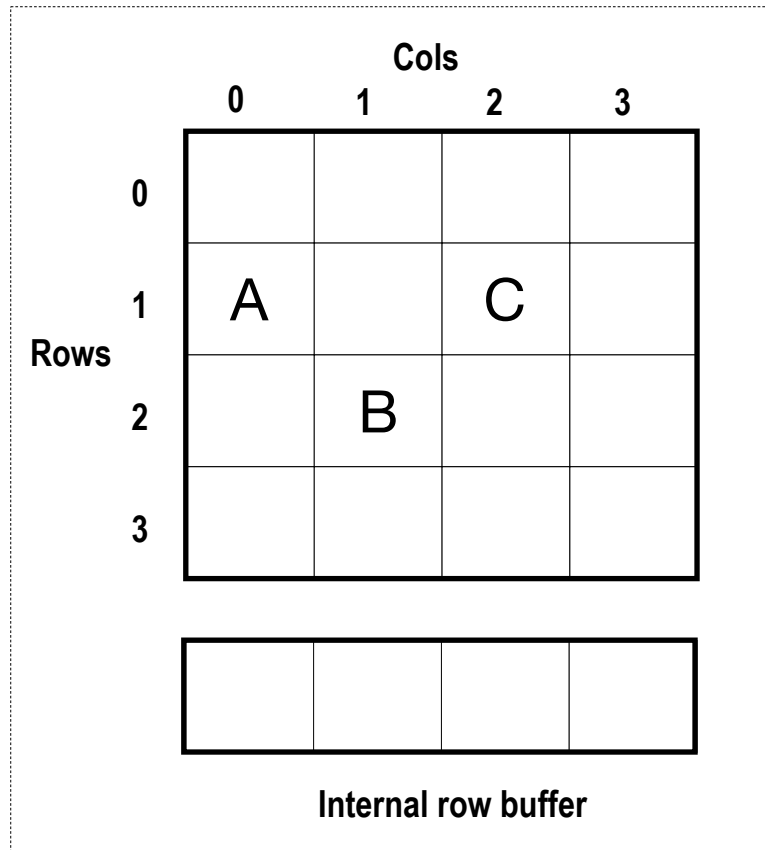




# Aside: DRAM Scheduling

- Assume the following memory accesses:  
A, B, C
- Which one is faster?
  - A  $\rightarrow$  B  $\rightarrow$  C
  - A  $\rightarrow$  C  $\rightarrow$  B

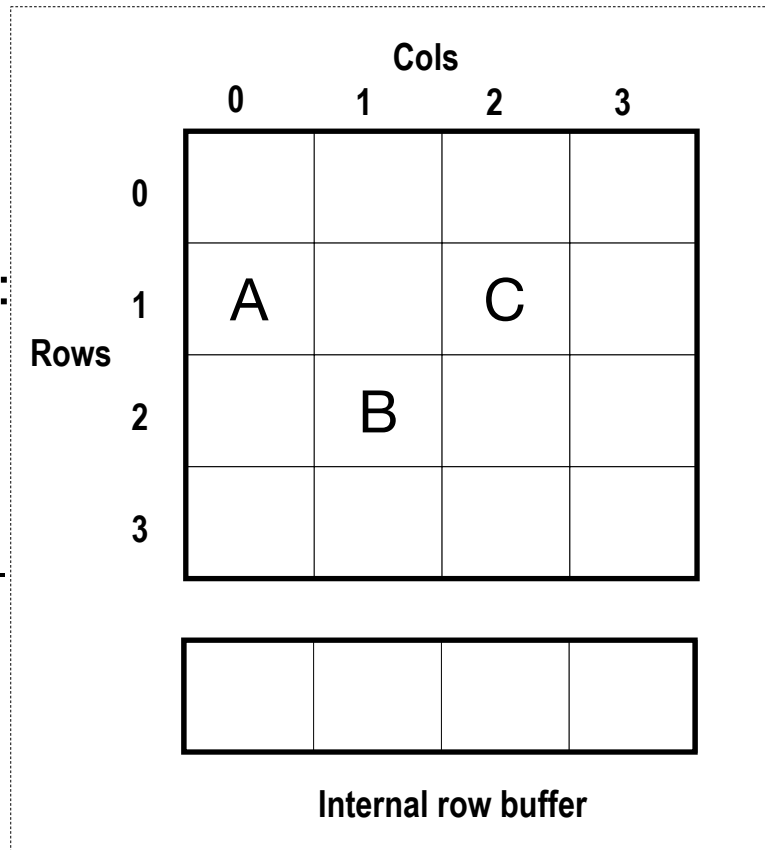
16 x 8 DRAM chip



# Aside: DRAM Scheduling

- Assume the following memory accesses:  
A, B, C
- Which one is faster?
  - A  $\rightarrow$  B  $\rightarrow$  C
  - A  $\rightarrow$  C  $\rightarrow$  B
- Most common memory scheduling policy:  
FR-FCFS
  - First-ready, first-come-first-serve
  - Prioritize addresses to data that is already in the row buffer; otherwise first-come-first-serve

16 x 8 DRAM chip



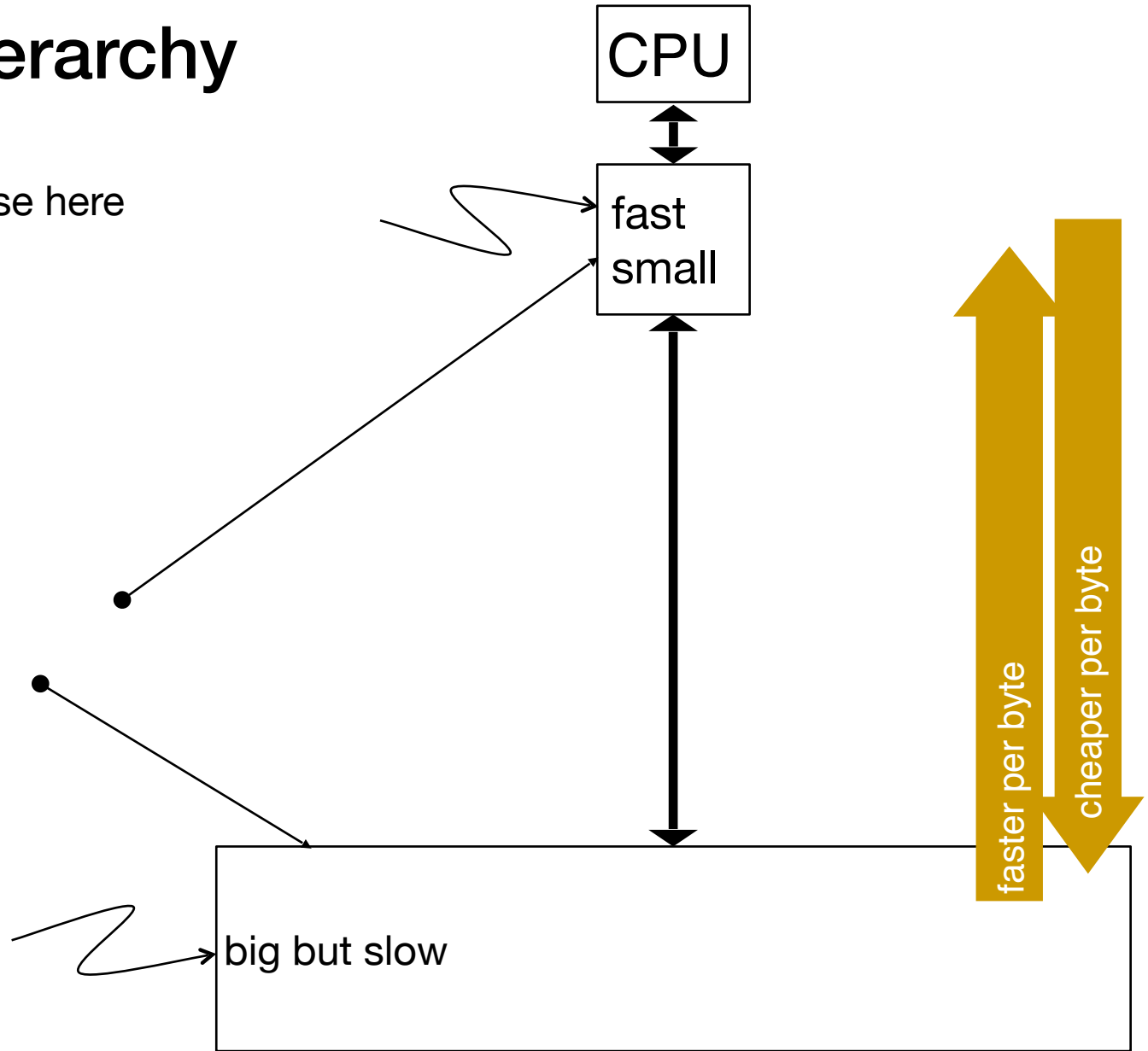
# We want both fast and large Memory

- But we cannot achieve both with a single level of memory
- Idea: Memory Hierarchy
  - Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor)
  - Key: manage the data such that most of the data the processor needs in the near future is kept in the fast(er) level(s)

# Memory Hierarchy

move what you use here

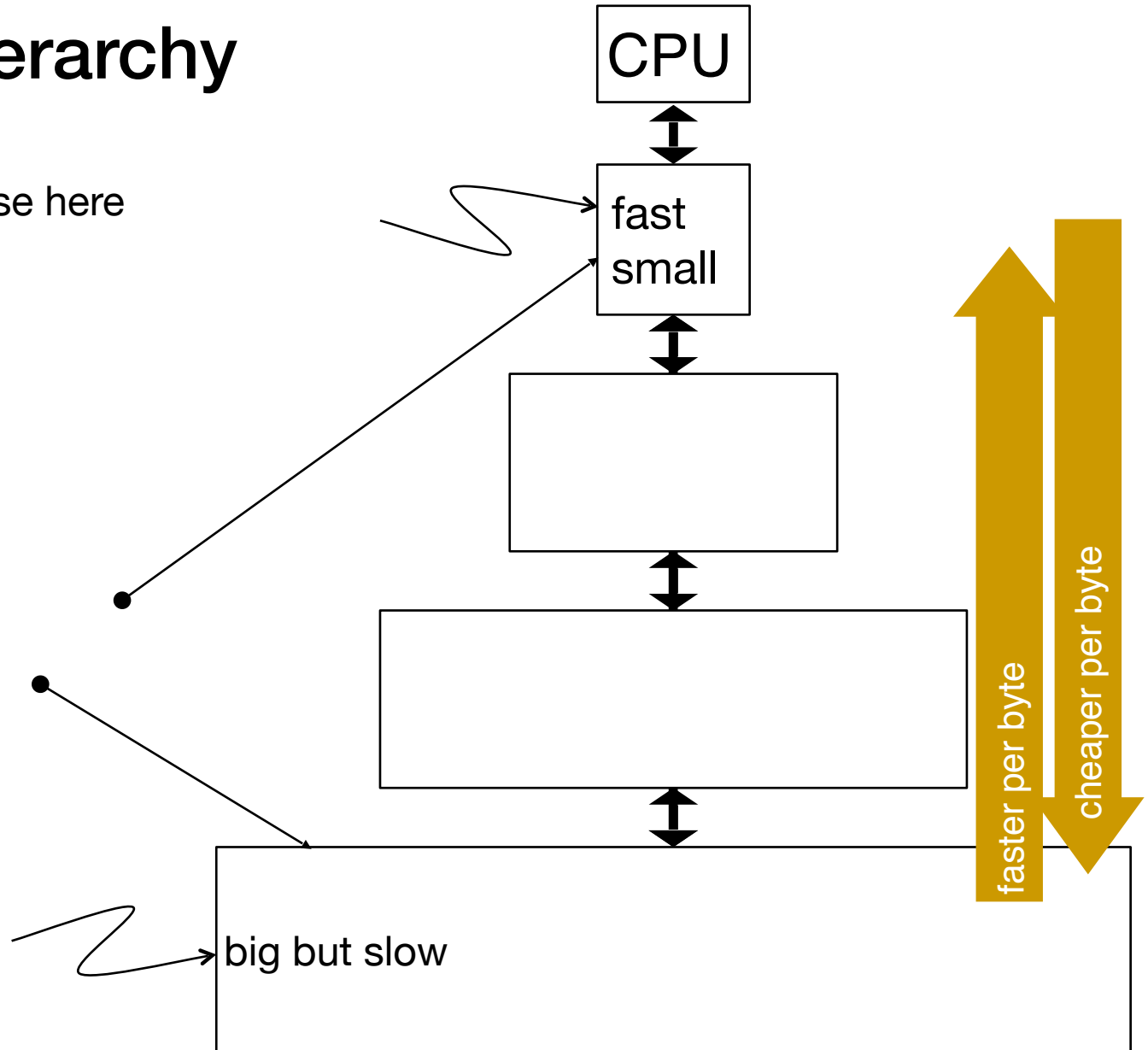
backup  
everything  
here



# Memory Hierarchy

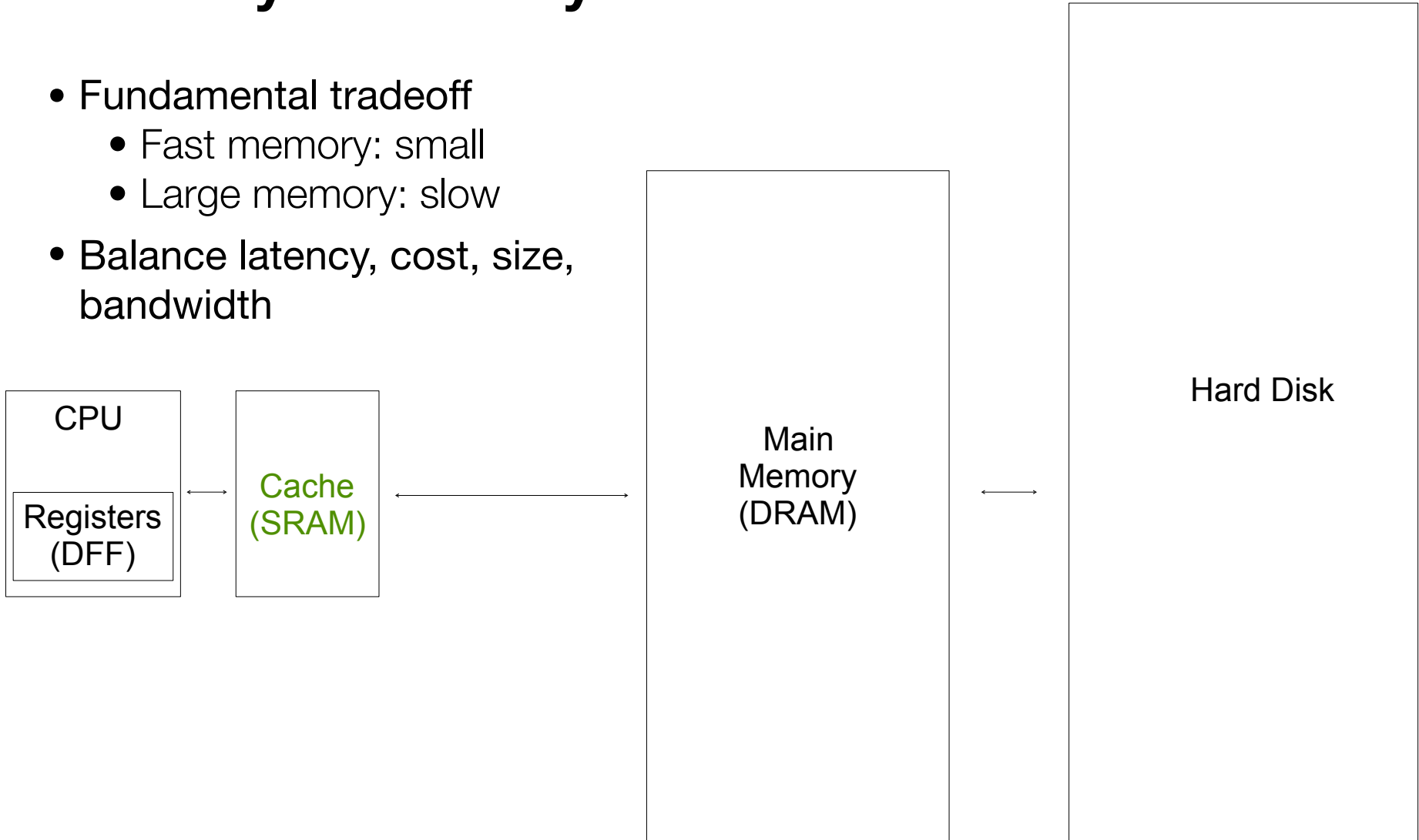
move what you use here

backup  
everything  
here



# Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow
- Balance latency, cost, size, bandwidth



# A Modern Memory Hierarchy

Register File (DFF)  
32 words, sub-nsec

---

L1 cache (SRAM)  
~32 KB, ~nsec

L2 cache (SRAM)  
512 KB ~ 1MB, many nsec

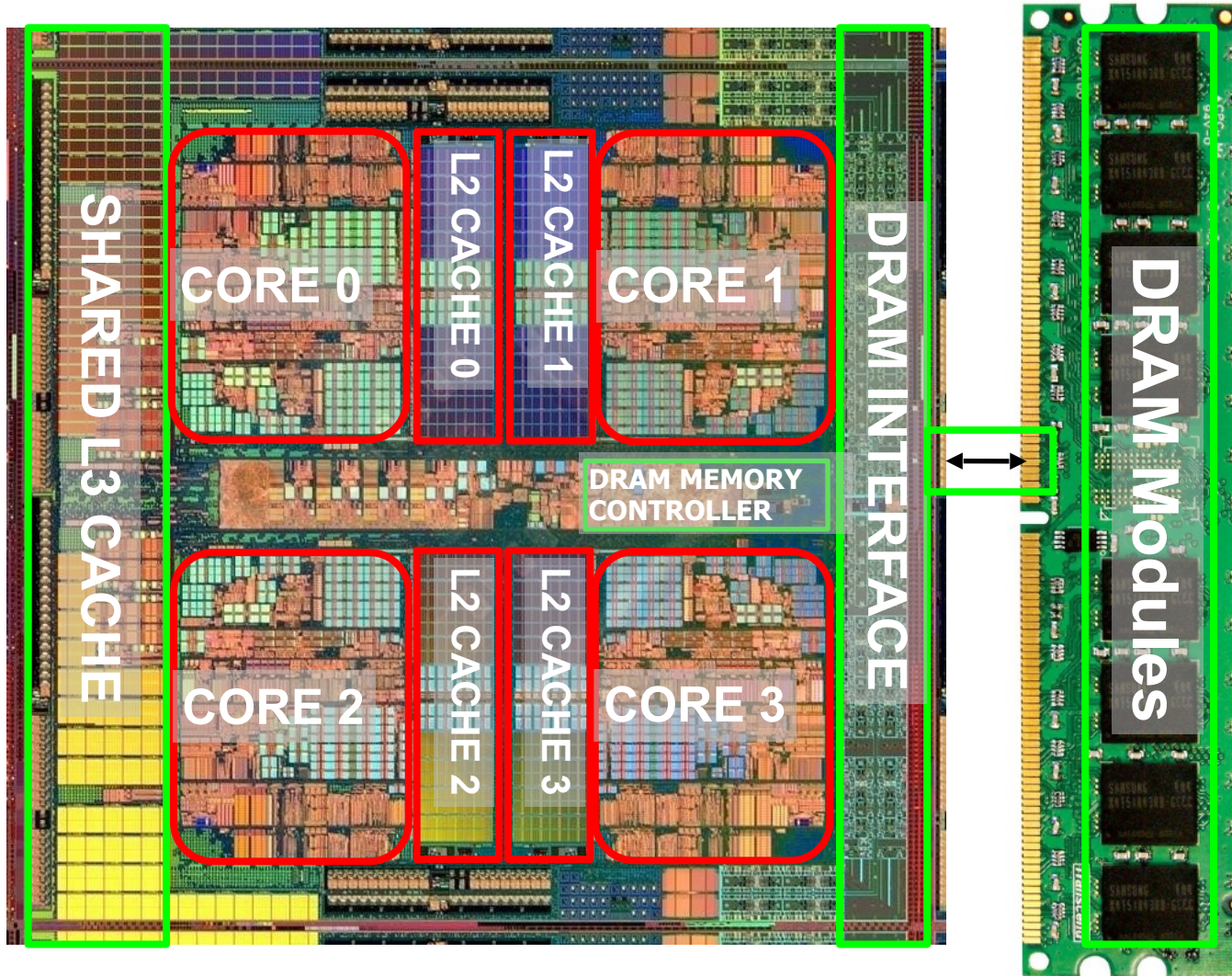
L3 cache (SRAM)  
.....

Main memory (DRAM),  
GB, ~100 nsec

---

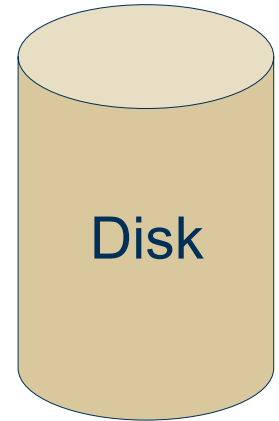
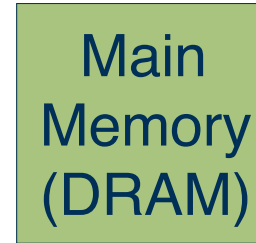
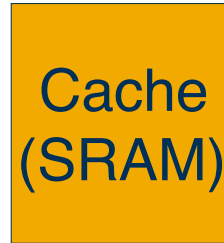
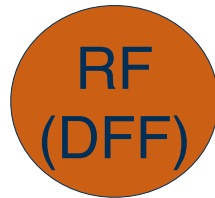
Hard Disk  
100 GB, ~10 msec

# Memory in a Modern System





# How Things Have Progressed

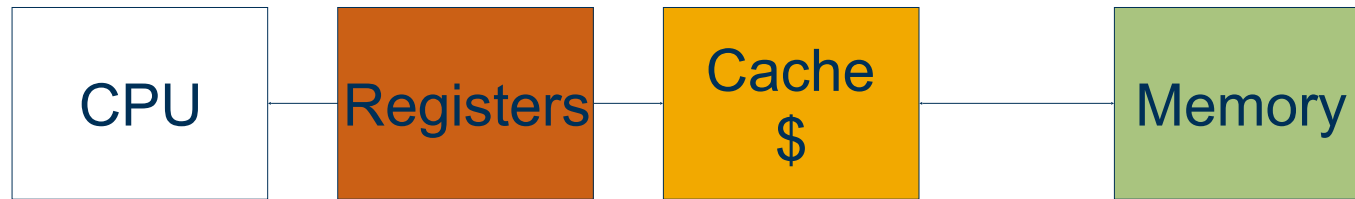


<b>1995 low-mid range</b> <small>Hennessy &amp; Patterson, Computer Arch., 1996</small>	<b>200B</b> <b>5ns</b>	<b>64KB</b> <b>10ns</b>	<b>32MB</b> <b>100ns</b>	<b>2GB</b> <b>5ms</b>
<b>2009 low-mid range</b> <small><a href="http://www.dell.com">www.dell.com</a>, \$449 including 17" LCD flat panel</small>	<b>~200B</b> <b>0.33ns</b>	<b>8MB</b> <b>0.33ns</b>	<b>4GB</b> <b>&lt;100ns</b>	<b>750GB</b> <b>4ms</b>
<b>2015 mid range</b>	<b>~200B</b> <b>0.33ns</b>	<b>8MB</b> <b>0.33ns</b>	<b>16GB</b> <b>&lt;100ns</b>	<b>256GB</b> <b>10us</b>

# How to Make Effective Use of the Hierarchy

- Fundamental question: how do we know what data to put in the fast and small memory?
- Answer: ensure most of the data the processor needs **in the near future** is kept in the fast(er) level(s)
- How do we know what data will be needed in the future?
  - Do we know **before** the program runs?
    - If so, programmers or compiler can place the right data at the right place
  - Do we know only **when** the program runs?
    - If so, only the hardware can effectively place the data

# How to Make Effective Use of the Hierarchy



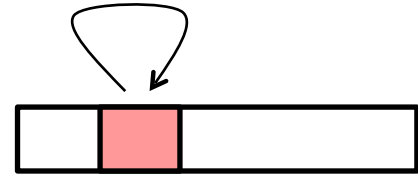
- Modern computers provide both ways
- Register file: programmers explicitly move data from the main memory (slow but big DRAM) to registers (small, very fast)
  - `movq (%rdi), %rdx`
- **Cache**, on the other hand, is automatically managed by hardware
  - Sits between registers and main memory, “invisible” to programmers
  - The hardware automatically figures out what data will be used in the near future, and place in the cache.
  - How does the hardware know that??

# Locality: An Empirical Observation

- **Principle of Locality:** Programs tend to use the same data over and over again, and tend to access data next to each other.

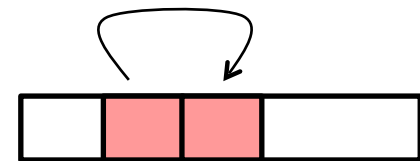
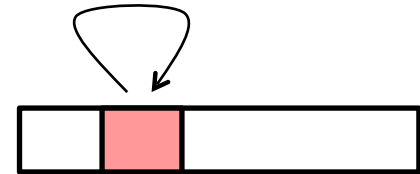
# Locality: An Empirical Observation

- **Principle of Locality:** Programs tend to use the same data over and over again, and tend to access data next to each other.
- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future



# Locality: An Empirical Observation

- **Principle of Locality:** Programs tend to use the same data over and over again, and tend to access data next to each other.
- **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

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

- Data references
  - **Spatial** Locality: Reference array elements in succession (stride-1 reference pattern)
  - **Temporal** Locality: Reference variable sum each iteration.
- Instruction references
  - **Spatial** Locality: Reference instructions in sequence.
  - **Temporal** Locality: Cycle through loop repeatedly.

# Use Locality to Manage Memory Hierarchy

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

- Exploiting temporal locality:
  - If a piece of data is recently accessed, very likely it will be needed again, so moved it to cache.
- Exploiting spatial locality:
  - When moving a piece of data from the memory to the cache, move its adjacent data to the cache as well.



# The Bookshelf Analogy

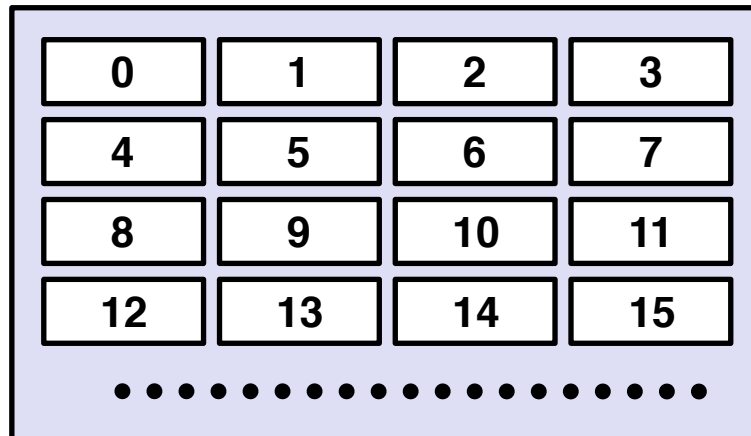
- Book in your hand
  - Desk
  - Bookshelf
  - Boxes at home
  - Library
- 
- Recently-used books tend to stay on desk, because you will likely use it again.
    - Comp Org. books, books for classes you are currently taking
  - Organize books in the shelf such that adjacent books are mostly accessed around the same time

# Cache Illustrations

CPU



Memory  
(big but slow)

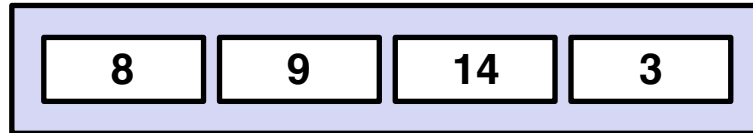


# Cache Illustrations

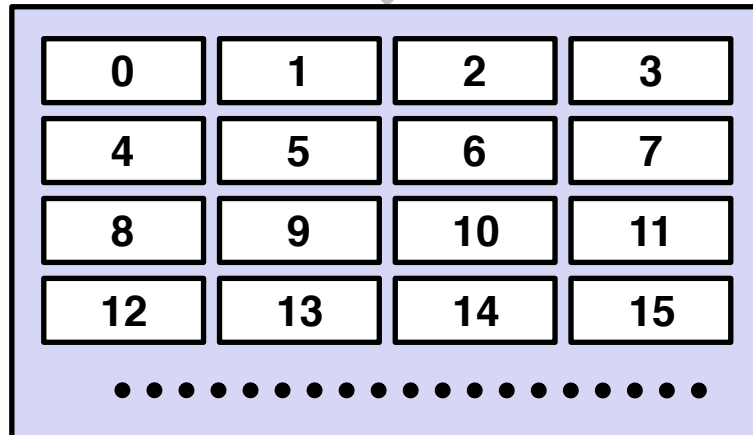
CPU



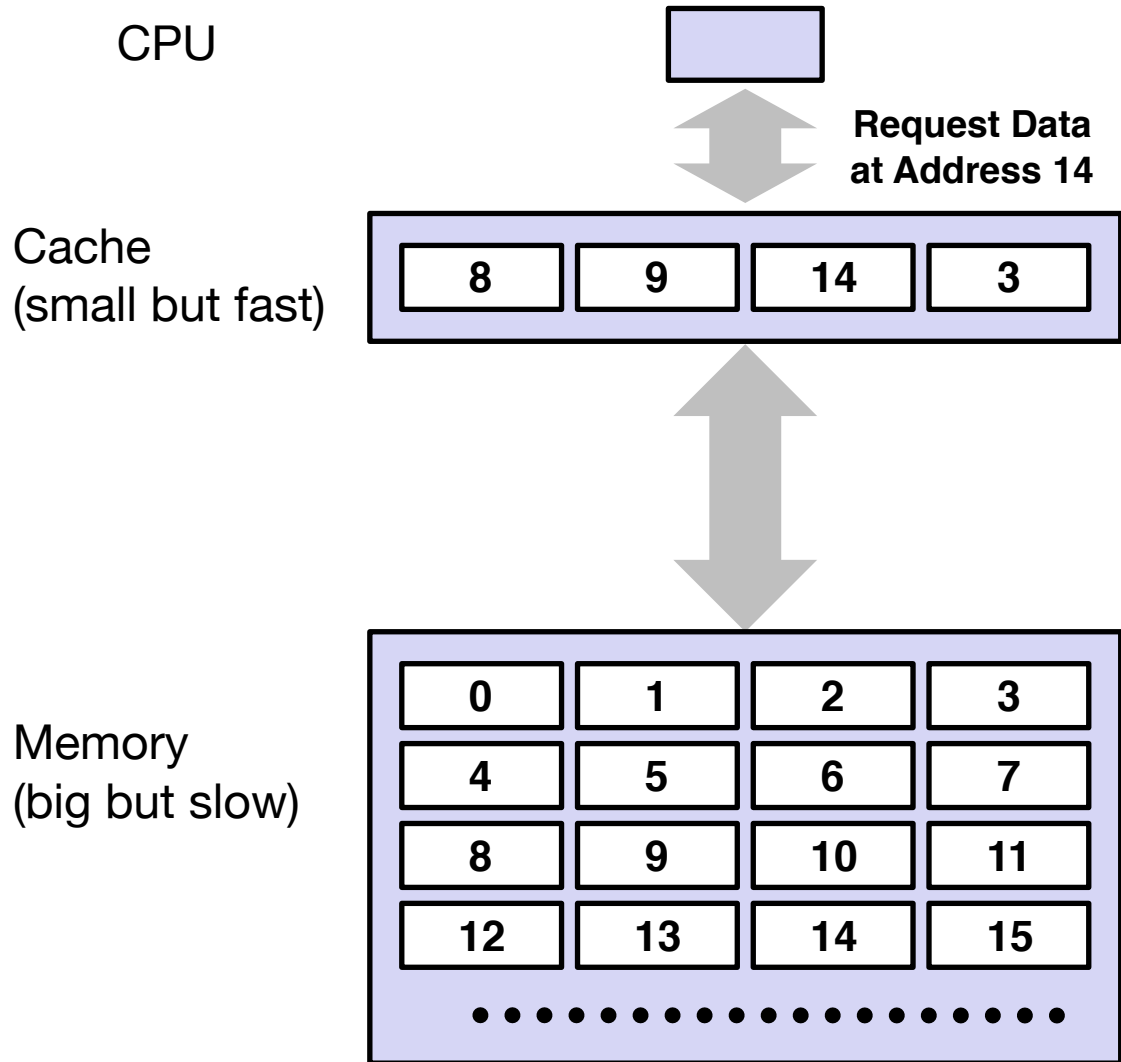
Cache  
(small but fast)



Memory  
(big but slow)

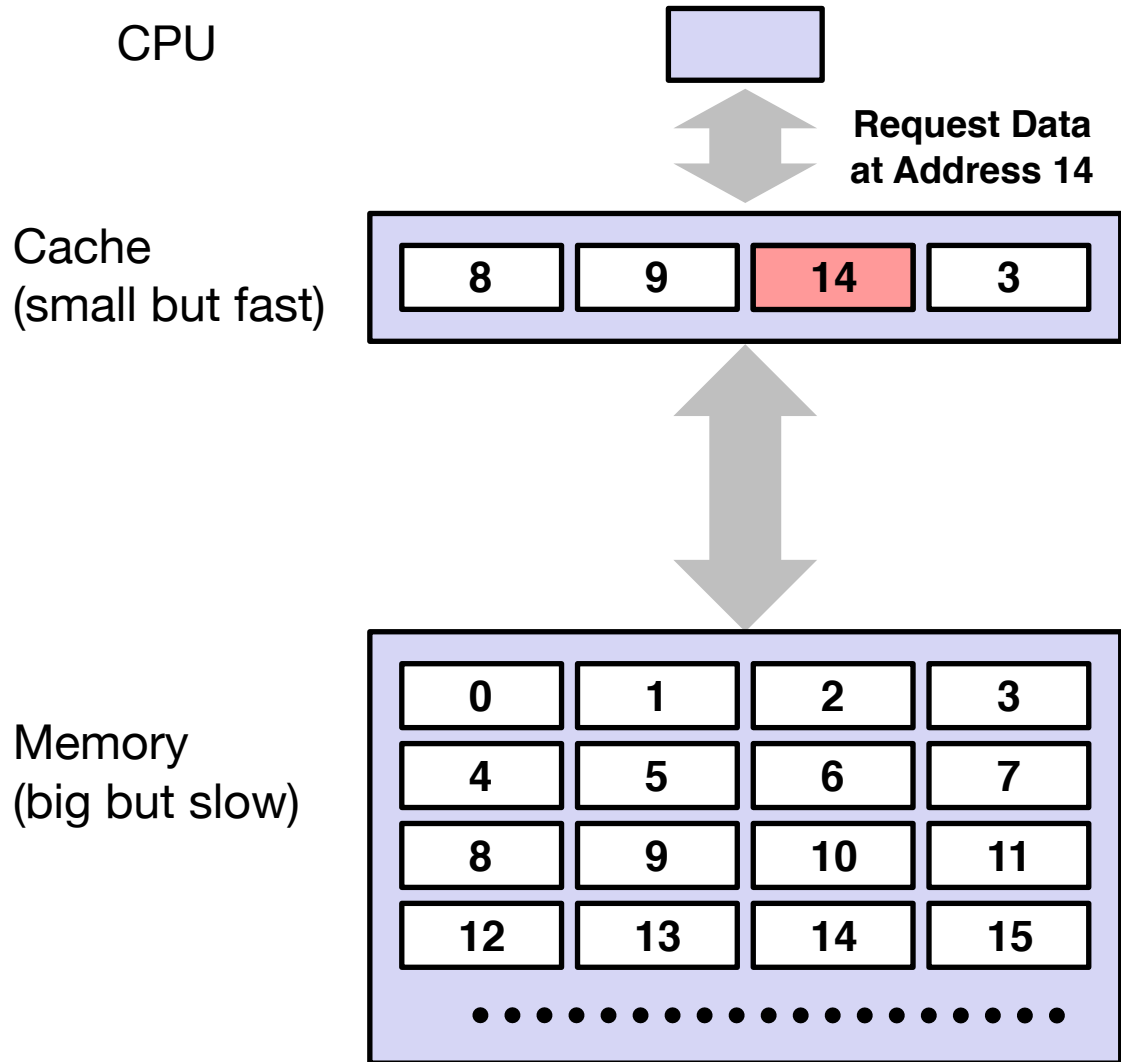


# Cache Illustrations



*Data in address b is needed*

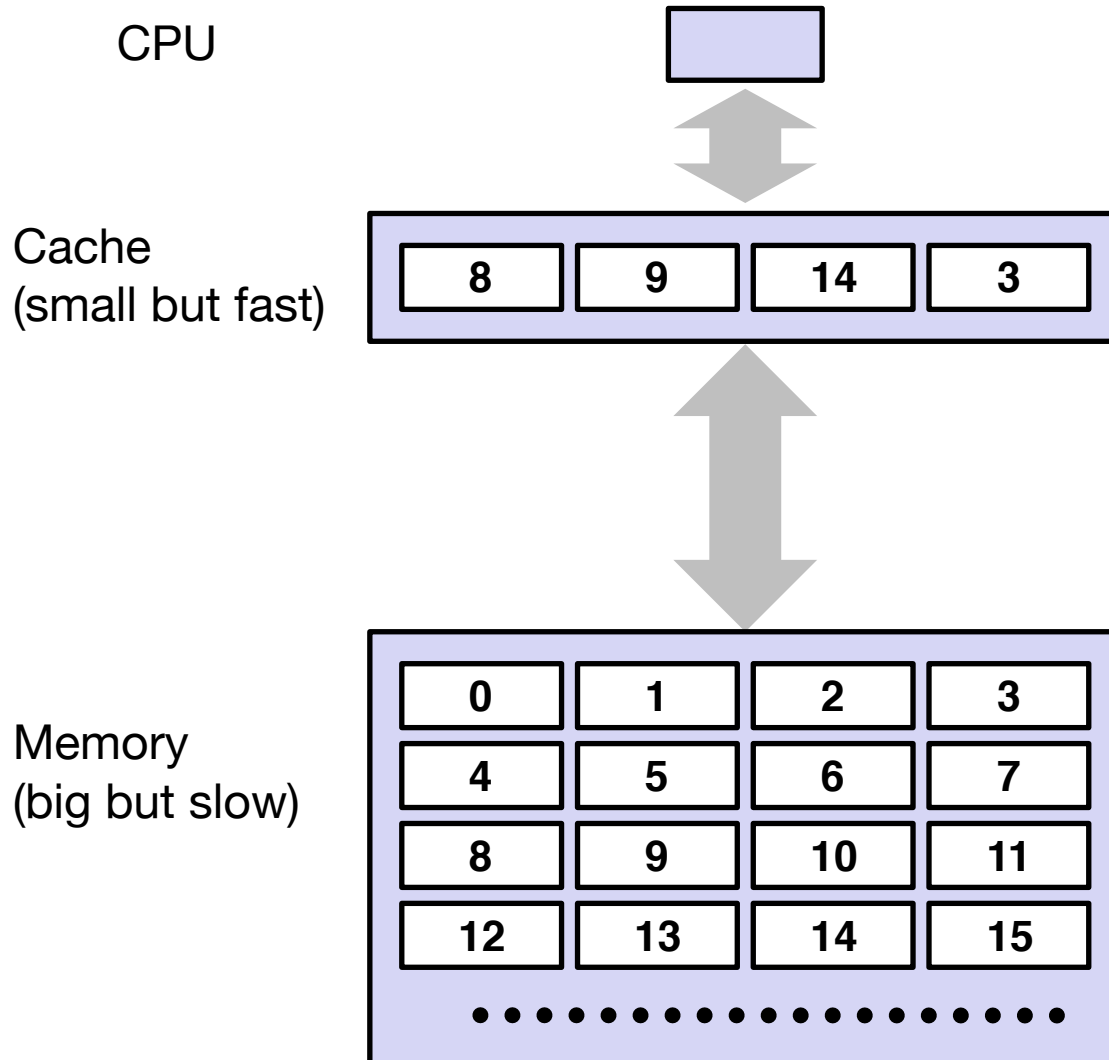
# Cache Illustrations



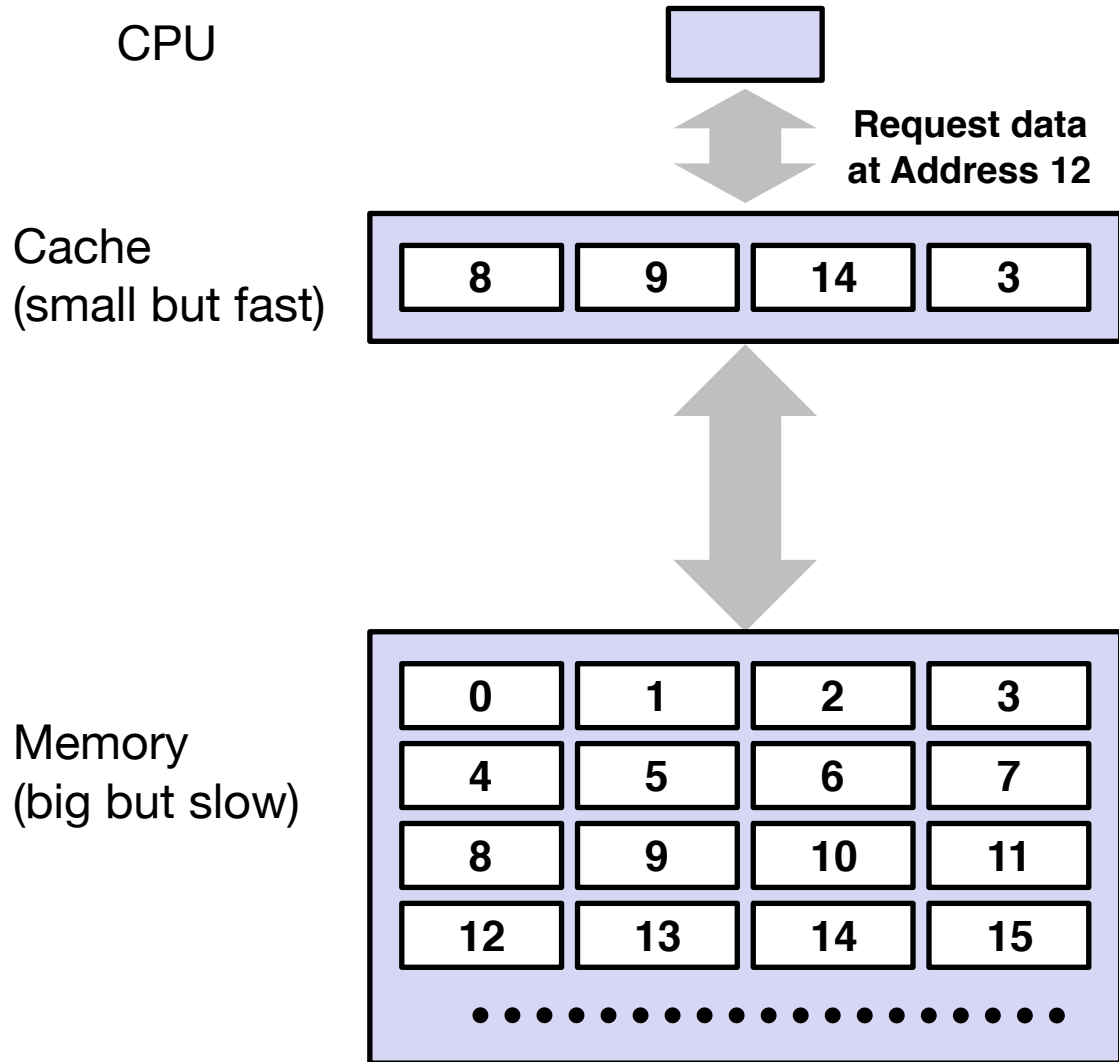
*Data in address  $b$  is needed*

*Address  $b$  is in cache: **Hit!***

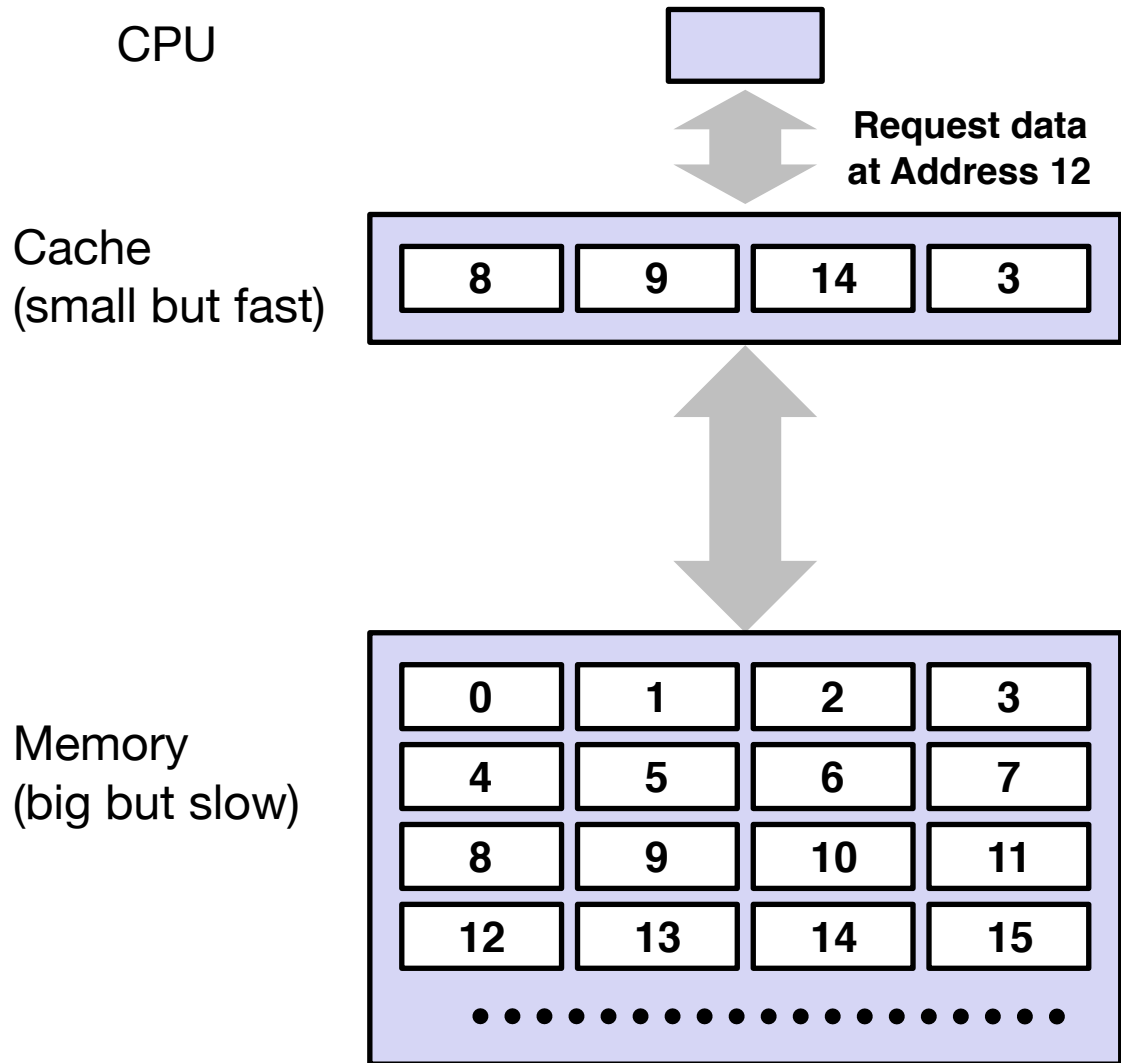
# Cache Illustrations



# Cache Illustrations



# Cache Illustrations

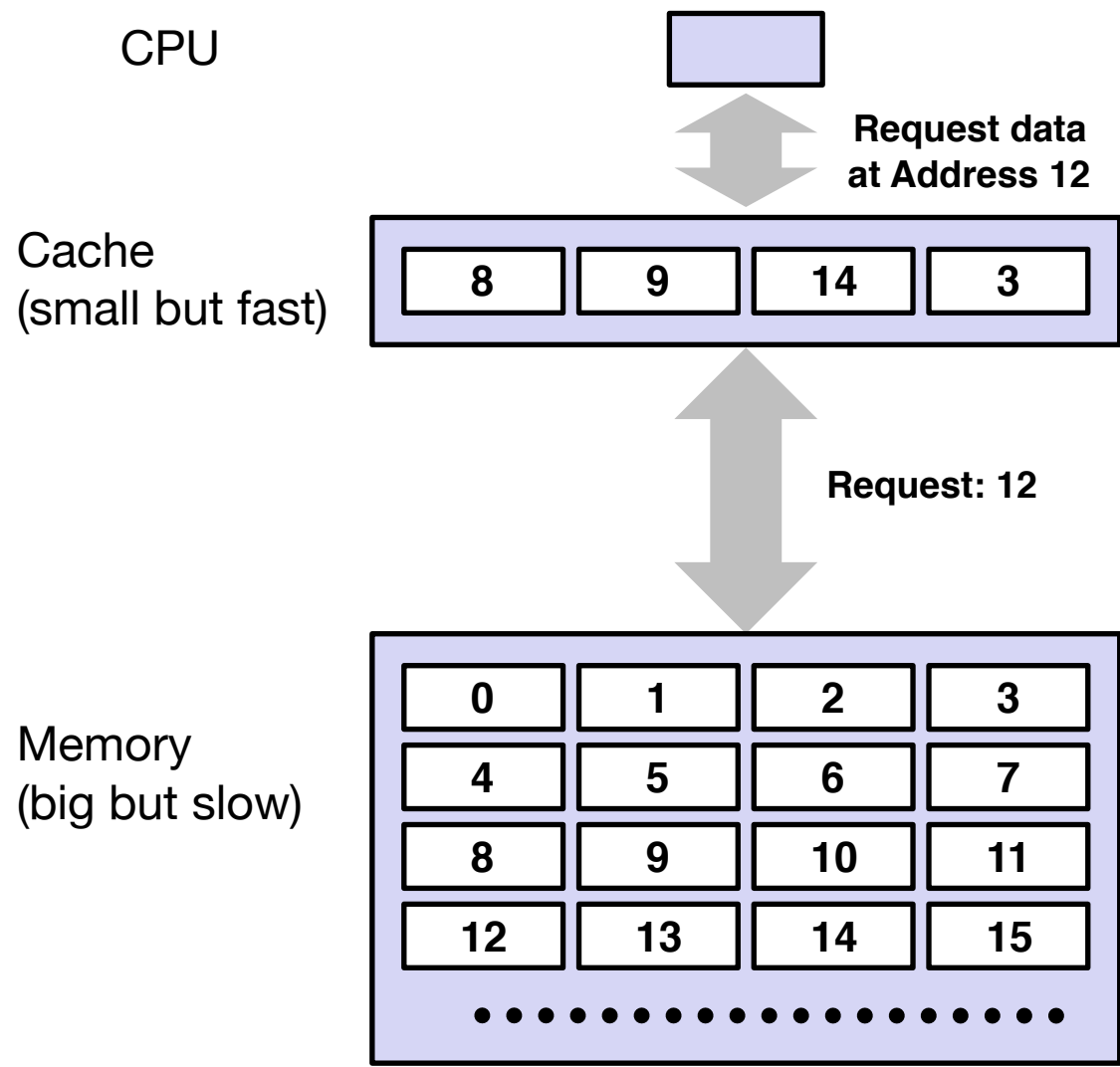


*Data in address  $b$  is needed*

*Address  $b$  is not in  
cache: **Miss!***



# Cache Illustrations

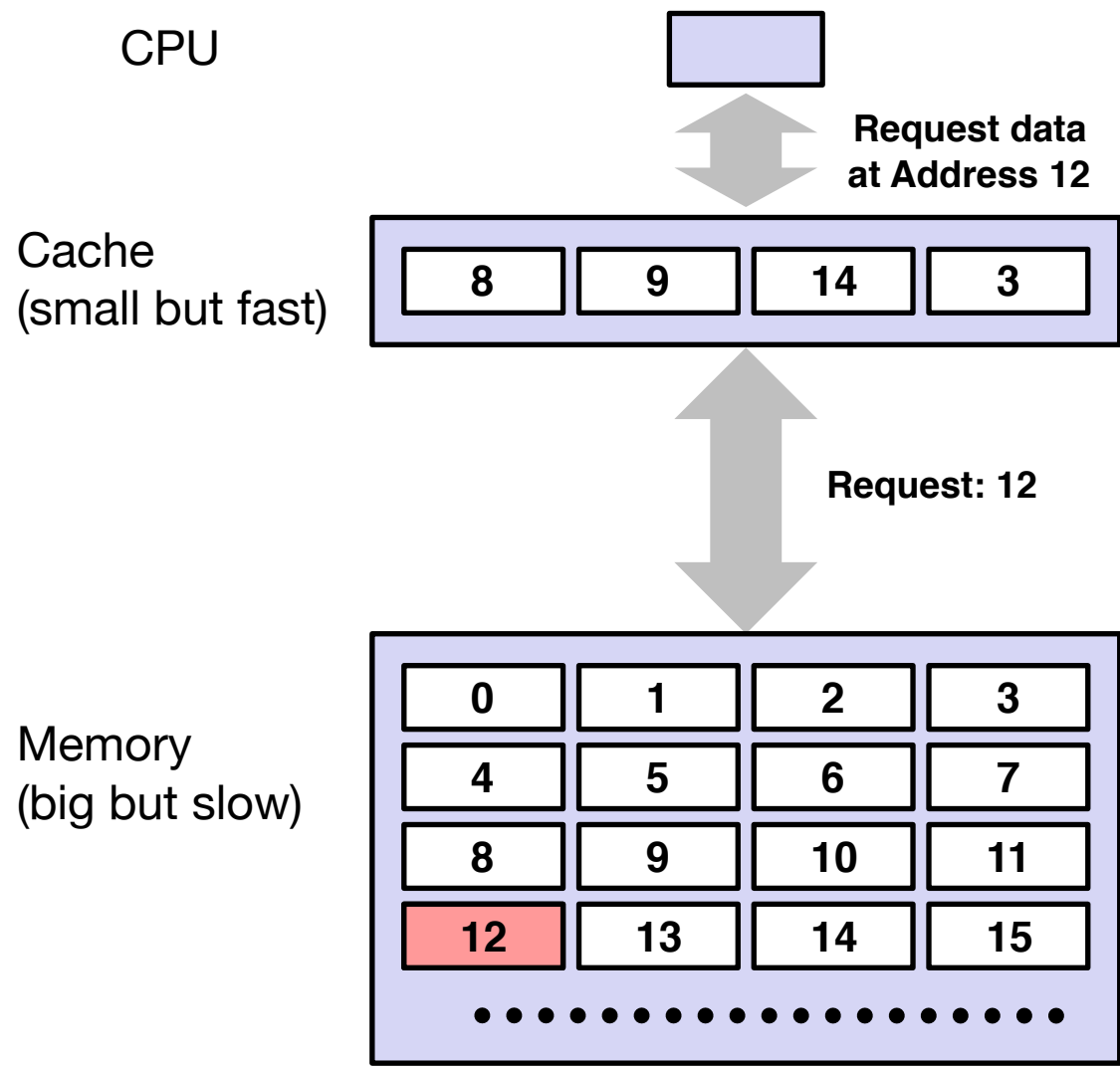


*Data in address b is needed*

*Address b is not in cache: **Miss!***

*Address b is fetched from memory*

# Cache Illustrations

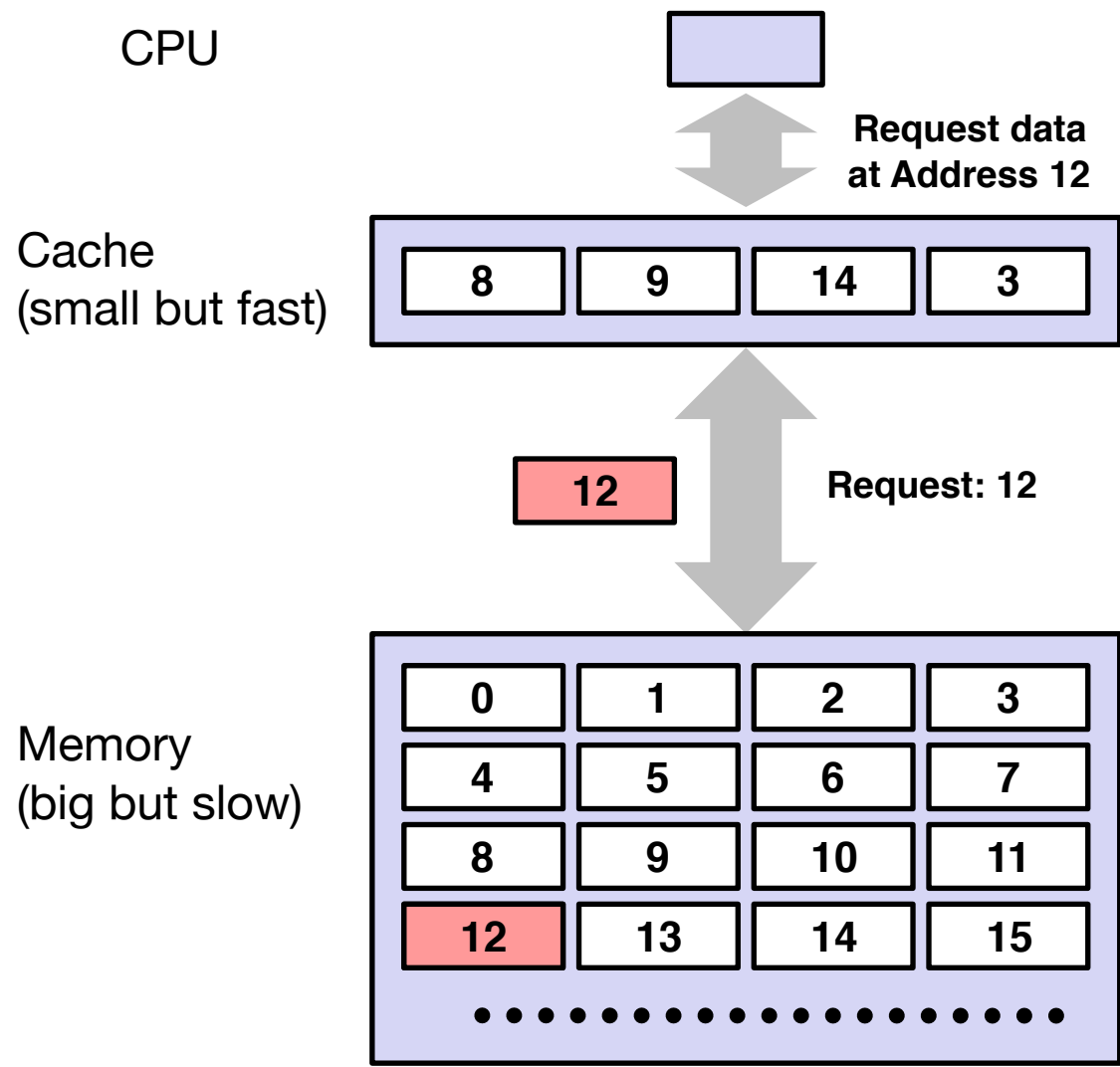


*Data in address b is needed*

*Address b is not in  
cache: **Miss!***

*Address b is fetched from  
memory*

# Cache Illustrations

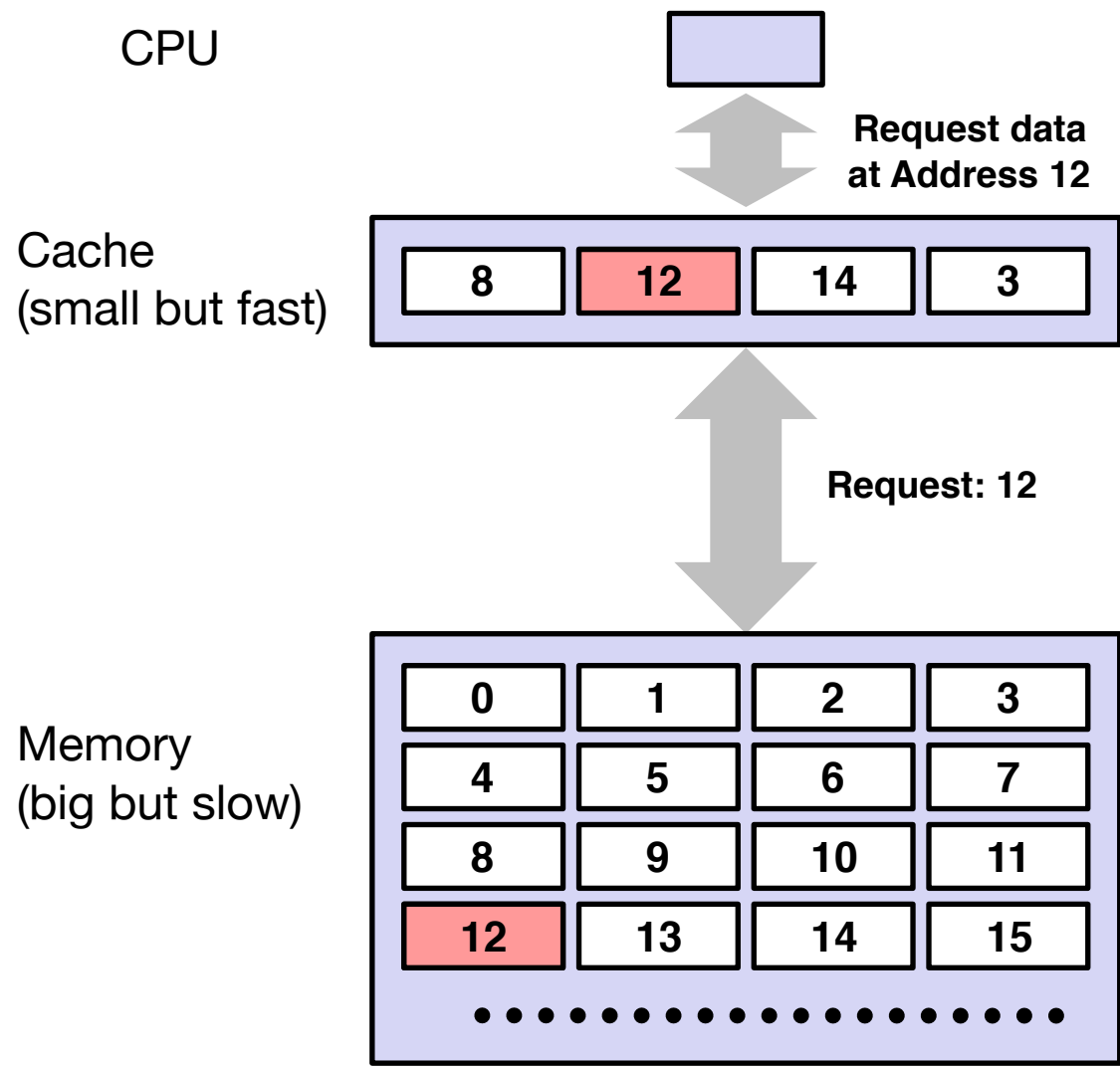


*Data in address b is needed*

*Address b is not in  
cache: **Miss!***

*Address b is fetched from  
memory*

# Cache Illustrations



*Data in address b is needed*

*Address b is not in cache: **Miss!***

*Address b is fetched from memory*

*Address b is stored in cache*