

# **CSC 252: Computer Organization**

## **Spring 2019: Lecture 16**

Instructor: Yuhao Zhu

Department of Computer Science  
University of Rochester

### **Action Items:**

- **Assignment 4 is out**

# Announcements

- Lab 4 is out.
- You will see run-to-run variation. We will take average when grading.

17	18	19	20	<b>21</b> <b>Today</b>	22	23
24	25	26	27	28	29	30
31	Apr 1 <b>Due</b>	2	3	4	5	6

# About Code Optimization

- Three entities can optimize the program: programmer, compiler, and hardware
- The best thing a programmer can do is to pick a good algorithm. Compilers/hardware can't do that in general.
- Algorithm choice decides overall complexity (big O), compiler/hardware decides the constant factor in the big O notation
- Quicksort:  $O(n \log n) = K * n * \log(n)$
- Bubblesort:  $O(n^2) = K * n^2$
- Compiler and hardware implementations decide the K.

# About Code Optimization

- From a programmer's perspective:
  - What you know: the functionality/intention of your code; the inputs to the program; all the code in the program
  - What you might not know: the hardware details.
- From a compiler's perspective:
  - What you know: all the code in the program (except library code); the hardware details.
  - What you might not know: the inputs to the program; the intention of the code
- From the hardware's perspective:
  - What you know: the hardware details; some part of the code
  - What you might not know: the inputs to the program; the intention of the code
- The different perspectives indicate that different entities have different responsibilities, limitations, and advantages in optimizing the code

# Today: Optimizing Code Transformation

- Things that programmer/compiler can do
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Exploiting hardware microarchitecture
- Things that compilers can't do but programmers can do

# Code Motion

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
void set_row(double *a, double *b,  
            long i, long n)  
{  
    long j;  
    for (j = 0; j < n; j++)  
        a[n*i+j] = b[j];  
}
```



```
long j;  
int ni = n*i;  
for (j = 0; j < n; j++)  
    a[ni+j] = b[j];
```

# Compiler-Generated Code Motion (-O1)

```
void set_row(double *a, double *b,  
            long i, long n)  
{  
    long j;  
    for (j = 0; j < n; j++)  
        a[n*i+j] = b[j];  
}
```

```
long j;  
long ni = n*i;  
double *rowp = a+ni;  
for (j = 0; j < n; j++)  
    *rowp++ = b[j];
```

```
set_row:  
    testq    %rcx, %rcx                # Test n  
    jle     .L1                        # If 0, goto done  
    imulq   %rcx, %rdx                # ni = n*i  
    leaq    (%rdi,%rdx,8), %rdx        # rowp = A + ni*8  
    movl    $0, %eax                  # j = 0  
.L3:                                           # loop:  
    movsd   (%rsi,%rax,8), %xmm0       # t = b[j]  
    movsd   %xmm0, (%rdx,%rax,8)       # M[A+ni*8 + j*8] = t  
    addq    $1, %rax                  # j++  
    cmpq    %rcx, %rax                # j:n  
    jne     .L3                        # if !=, goto loop  
.L1:                                           # done:  
    rep ; ret
```

# Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  - $16 * x \quad \text{-->} \quad x \ll 4$
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products



# Common Subexpression Elimination

- Reuse portions of expressions
- GCC will do this with `-O1`

3 multiplications:  $i*n$ ,  $(i-1)*n$ ,  $(i+1)*n$

```
/* Sum neighbors of i,j */
up = val[(i-1)*n + j ];
down = val[(i+1)*n + j ];
left = val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;
```



```
leaq 1(%rsi), %rax # i+1
leaq -1(%rsi), %r8 # i-1
imulq %rcx, %rsi # i*n
imulq %rcx, %rax # (i+1)*n
imulq %rcx, %r8 # (i-1)*n
addq %rdx, %rsi # i*n+j
addq %rdx, %rax # (i+1)*n+j
addq %rdx, %r8 # (i-1)*n+j
```

1 multiplication:  $i*n$

```
long inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```



```
imulq %rcx, %rsi # i*n
addq %rdx, %rsi # i*n+j
movq %rsi, %rax # i*n+j
subq %rcx, %rax # i*n+j-n
leaq (%rsi,%rcx), %rcx # i*n+j+n
```

# Exploiting Instruction-Level Parallelism

- Hardware can execute multiple instructions in parallel
  - Pipeline is a classic technique
- Performance limited by control/data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers are generally good at exploiting hardware because it has better knowledge about the hardware than programmers.
  - But often compilers cannot make these transformations due to e.g., lack of associativity and distributivity in floating-point arithmetic, not knowing the accuracy requirements of a program, etc.

# Baseline Code

```
for (i = 0; i < length; i++) {  
    t = t * d[i];  
    *dest = t;  
}
```

**.L519:**

```
imulq (%rax,%rdx,4), %ecx  
addq $1, %rdx      # i++  
cmpq %rdx, %rbp   # Compare length:i  
jg .L519          # If >, goto Loop
```

← Real work

← Overhead

# Loop Unrolling

```
long limit = length-1;
long i;
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x = (x * d[i]) * d[i+1];
}

/* Finish any remaining elements */
for (; i < length; i++) {
    x = x * d[i];
}
*dest = x;
```

- Perform 2x more useful work per iteration
- Reduce loop overhead (comp, jmp, index dec, etc.)
- Reduce branch stalls (if the hardware doesn't have good branch predictors)

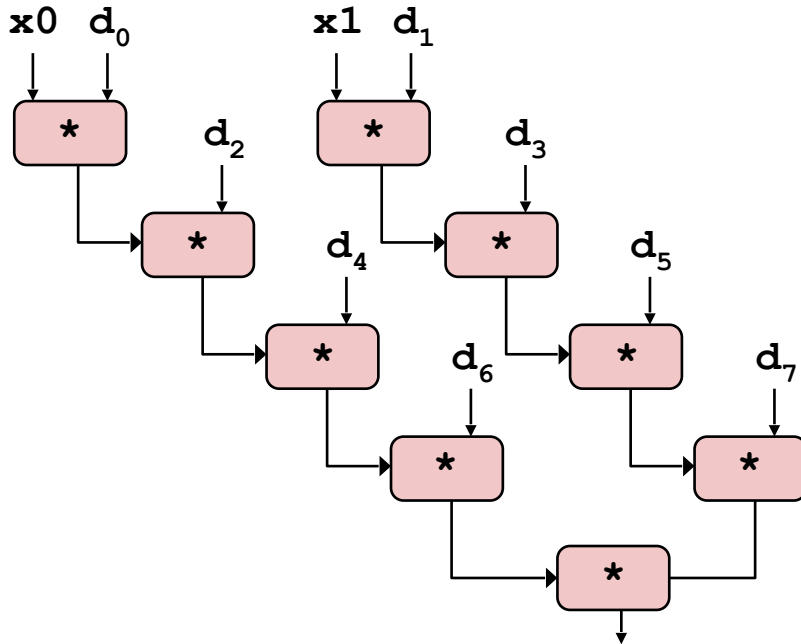
# Loop Unrolling with Separate Accumulators

```
long limit = length-1;
long i;
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x0 = x0 * d[i];
    x1 = x1 * d[i+1];
}

/* Finish any remaining elements */
for (; i < length; i++) {
    x0 = x0 * d[i];
}
*dest = x0 * x1;
```

# Separate Accumulators

```
x0 = x0 * d[i];  
x1 = x1 * d[i+1];
```



- What changed:
  - Two independent “streams” of operations
  - Reduce data dependency: good for pipelining.

# Today: Optimizing Code Transformation

- Things that programmer/compiler can do
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Exploiting hardware microarchitecture
- Things that compilers can't do but programmers can do

# A Trivial Example

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```



# A Trivial Example

- As a programmer, if you know what  $x$  and  $y$  will be, say 5, you could direct return the results 23769.8 without having to the computation

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

# A Trivial Example

- As a programmer, if you know what  $x$  and  $y$  will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

# A Trivial Example

- As a programmer, if you know what  $x$  and  $y$  will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea
- Except...Profile-guided optimizations:

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

# A Trivial Example

- As a programmer, if you know what  $x$  and  $y$  will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea
- Except...Profile-guided optimizations:
  - Run the code multiple times using some sample inputs, and observe the values of  $x$  and  $y$  (statistically).

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

# A Trivial Example

- As a programmer, if you know what  $x$  and  $y$  will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea
- Except...Profile-guided optimizations:
  - Run the code multiple times using some sample inputs, and observe the values of  $x$  and  $y$  (statistically).
  - If let's say 99% of the time,  $x = 2$  and  $y = 5$ , what could the compiler do then?

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

# A Trivial Example

- As a programmer, if you know what  $x$  and  $y$  will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea
- Except...Profile-guided optimizations:
  - Run the code multiple times using some sample inputs, and observe the values of  $x$  and  $y$  (statistically).
  - If let's say 99% of the time,  $x = 2$  and  $y = 5$ , what could the compiler do then?

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

```
float foo(int x, int y)
{
    if (x == 2 && y == 5) return 23769.8;
    else return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

# A Less Trivial Example: Procedure Calls

- Procedure to Convert String to Lower Case

```
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

# Calling Strlen

```
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

- **Strlen performance**
  - Has to scan the entire length of a string, looking for null character.
  - $O(N)$  complexity
- **Overall performance**
  - $N$  calls to strlen
  - Overall  $O(N^2)$  performance



# Improving Performance

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

```
void lower(char *s)
{
    size_t i;
    size_t len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

# Optimization Blocker: Procedure Calls

```
void lower(char *s)
{
    size_t i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

```
size_t total_lencount = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    total_lencount += length;
    return length;
}
```

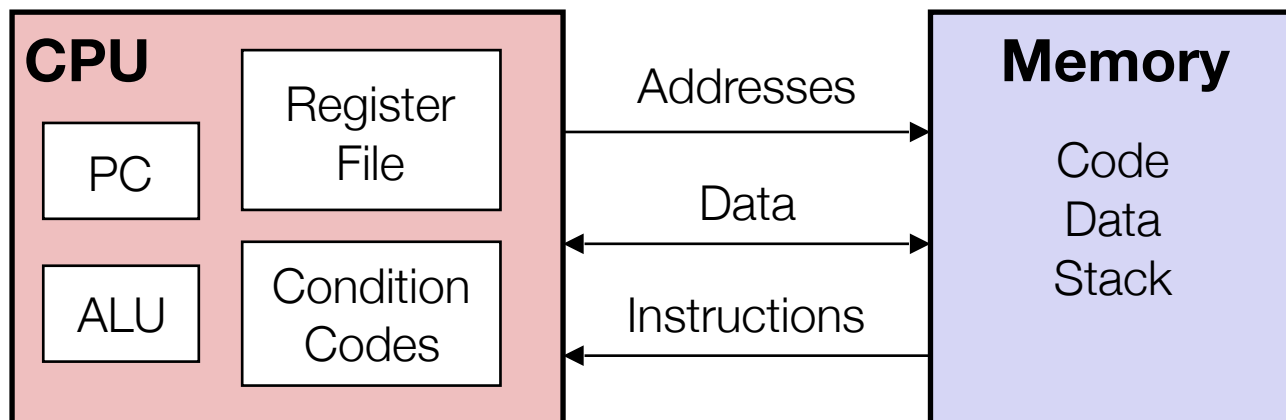
Why couldn't compiler move `strlen` out of loop?

- Procedure may have side effects, e.g., alters global state each time called
- Function may not return same value for given arguments

# Optimization Blocker: Procedure Calls

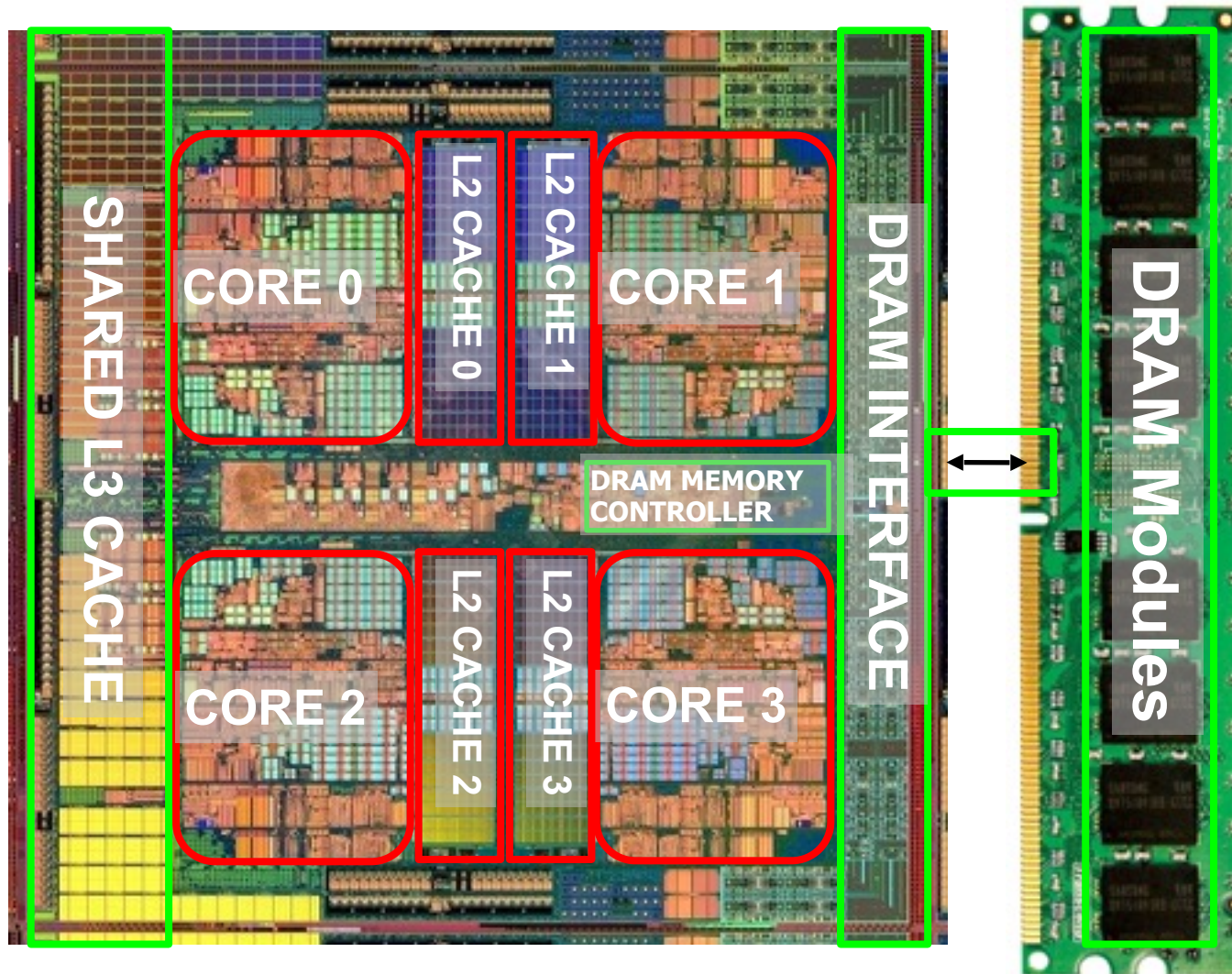
- Most compilers treat procedure call as a black box
  - Assume the worst case, weak optimizations near them
  - There are interprocedural optimizations (IPO), but they are expensive
  - Sometimes the compiler doesn't have access to source code of other functions because they are object files in a library. Link-time optimizations (LTO) comes into play, but are expensive as well.
- Remedies:
  - Use of inline functions
    - GCC does this with `-O1`, but only within single file
  - Do your own code motion

# 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?

# Memory in a Modern System



# 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

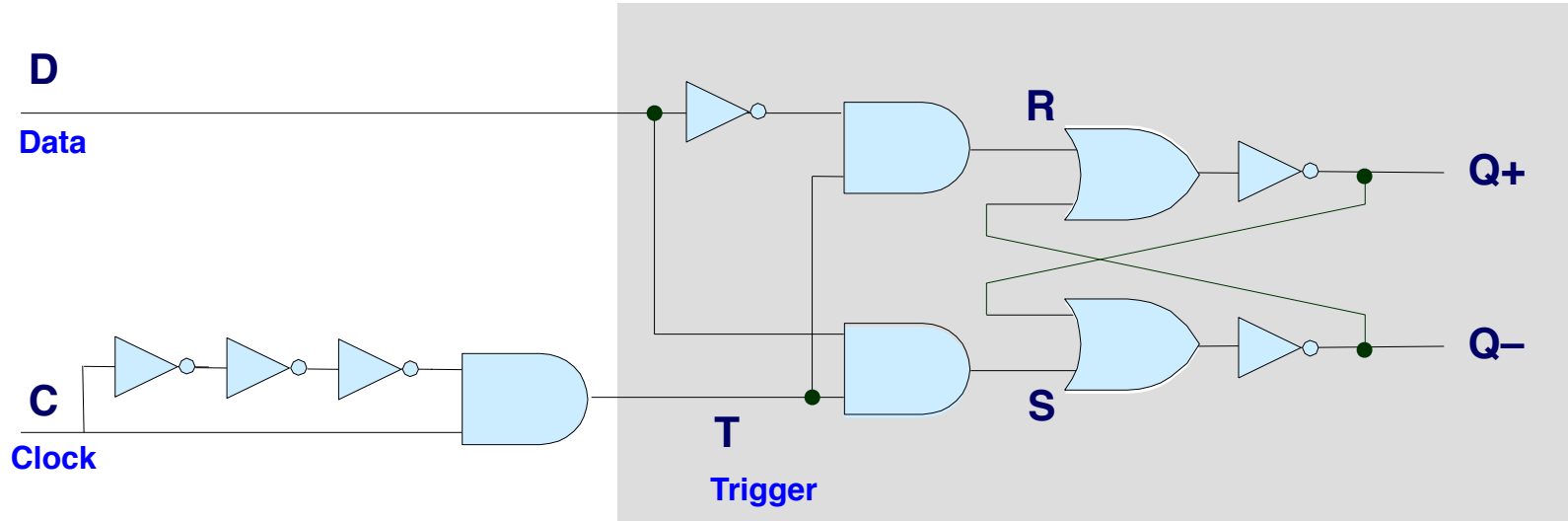
# The Problem

- 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

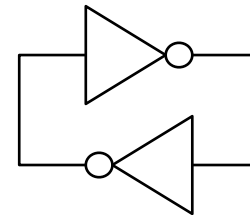
- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: Flip-flop vs. SRAM vs. DRAM vs. Disk vs. Tape
- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency,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

# Memory Technology: D Flip-Flop (DFF)



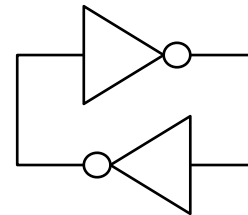
- Very fast
- Very expensive to build
  - 6 NOT gates (2 transistors / gate)
  - 3 AND gates (3 transistors / gate)
  - 2 OR gates (3 transistors / gate)
  - 27 transistors in total for just one bit!!

# Memory Technology: SRAM



# Memory Technology: SRAM

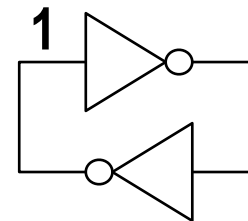
- Static random access memory
- Random access means you can supply an arbitrary address to the memory and get a value back
- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the “cell”
  - 4 transistors for storage
  - 2 transistors for access
 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





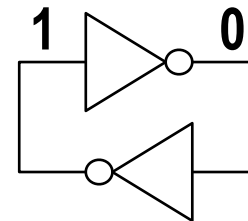
# Memory Technology: SRAM

- 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



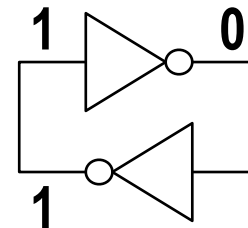
# Memory Technology: SRAM

- 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



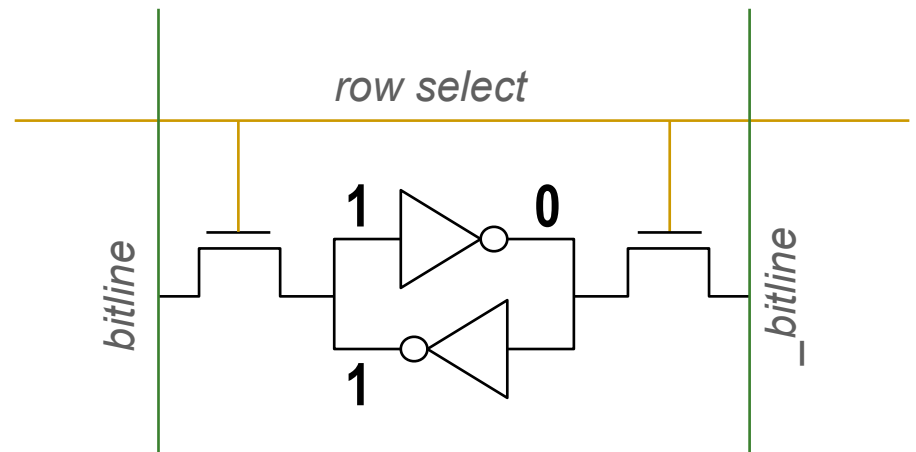
# Memory Technology: SRAM

- 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



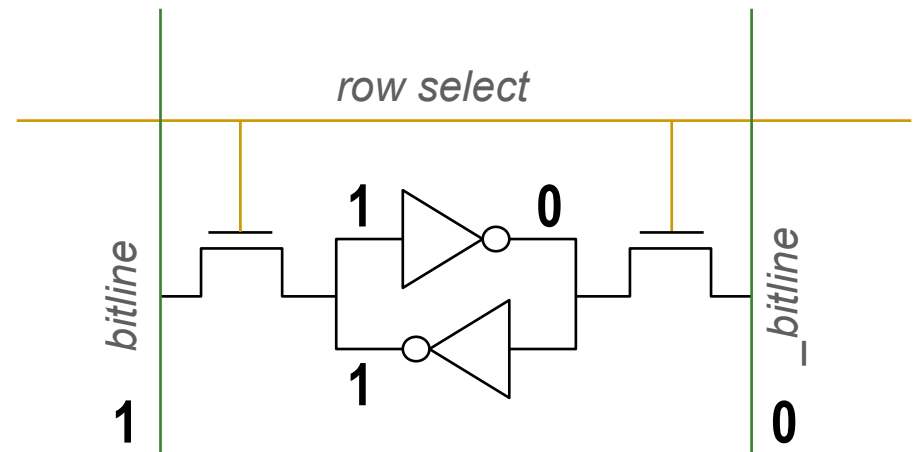
# Memory Technology: SRAM

- 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

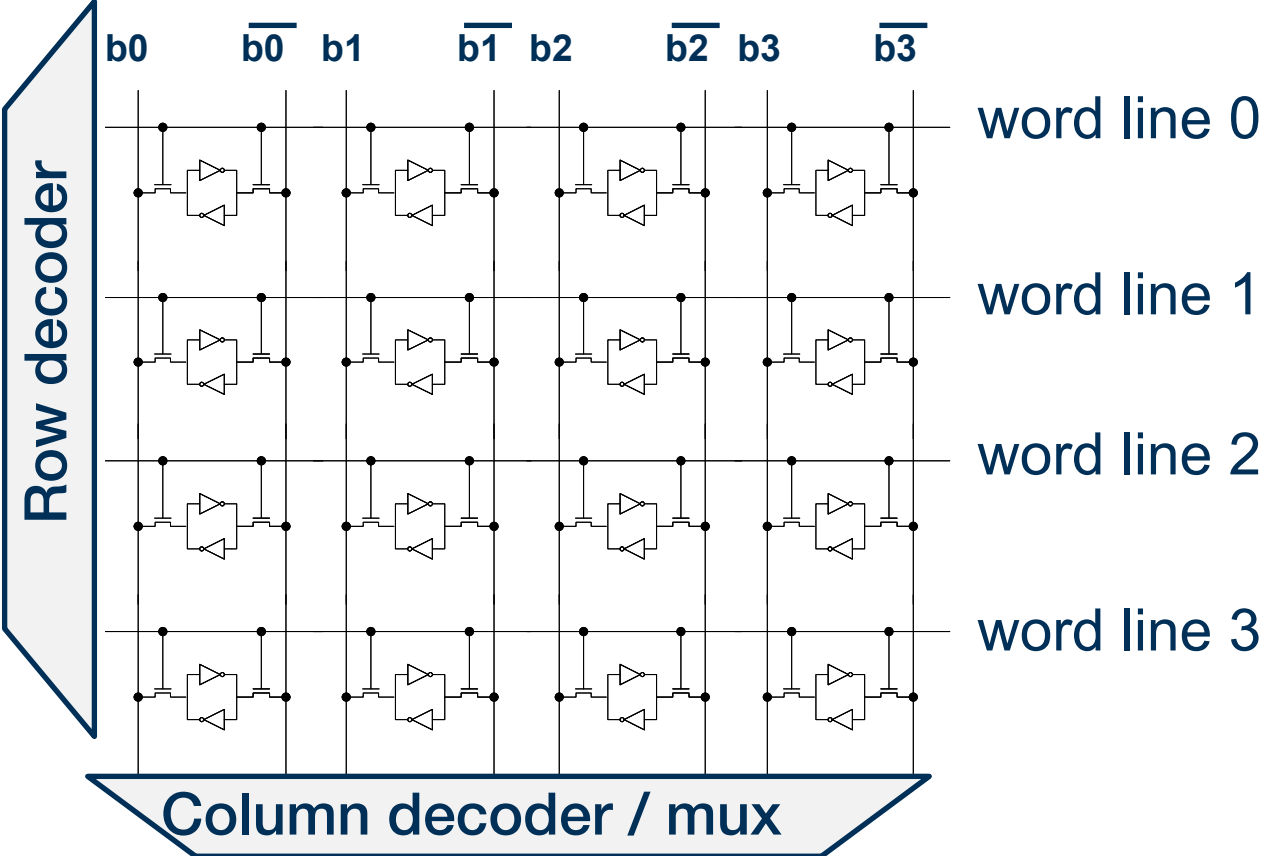


# Memory Technology: SRAM

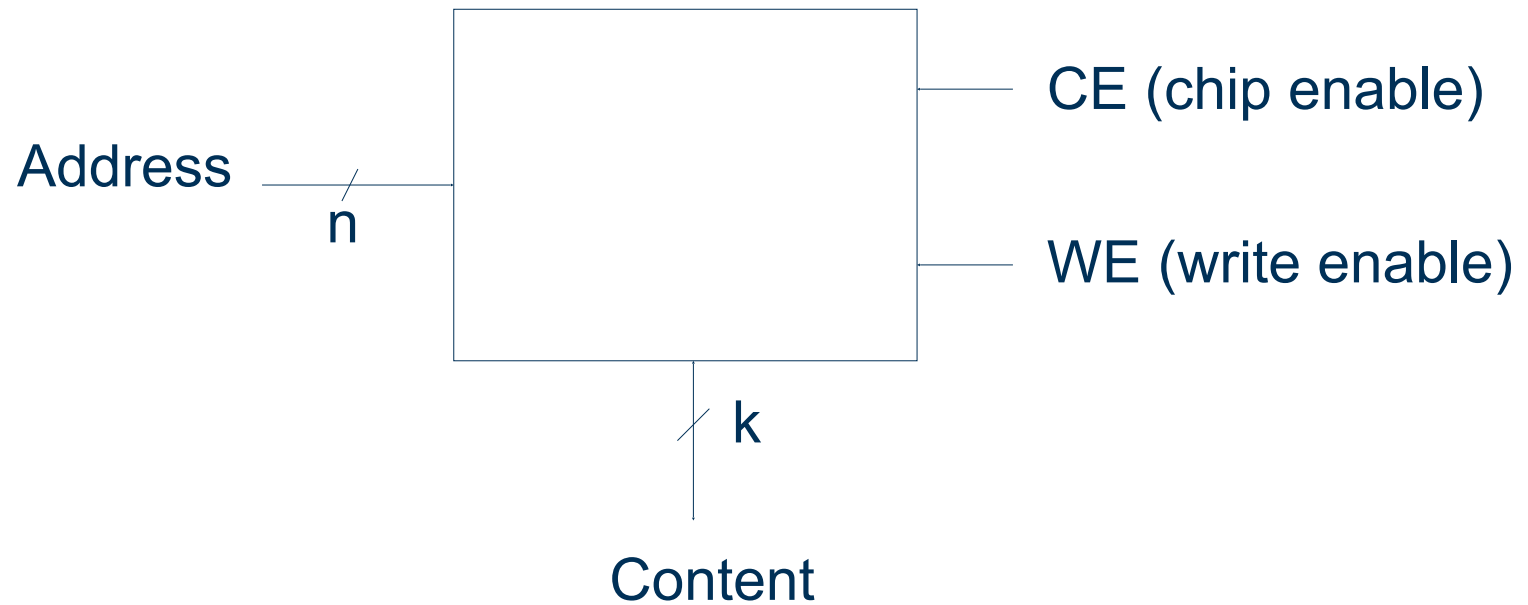
- Static random access memory
- Random access means you can supply an arbitrary address to the memory and get a value back
- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the “cell”
  - 4 transistors for storage
  - 2 transistors for access
 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



# SRAM Array



# Abstract View of SRAM



# Memory Technology: DRAM

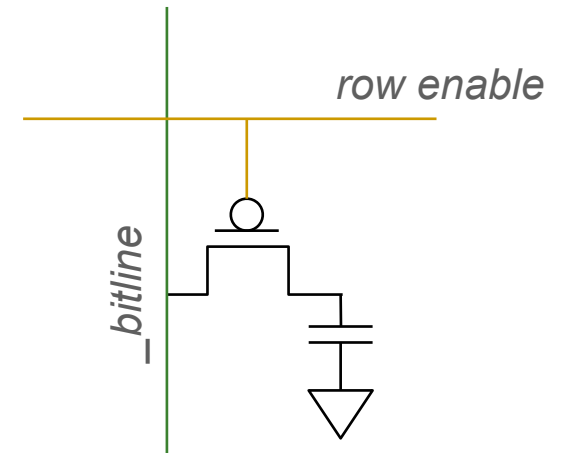
- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor





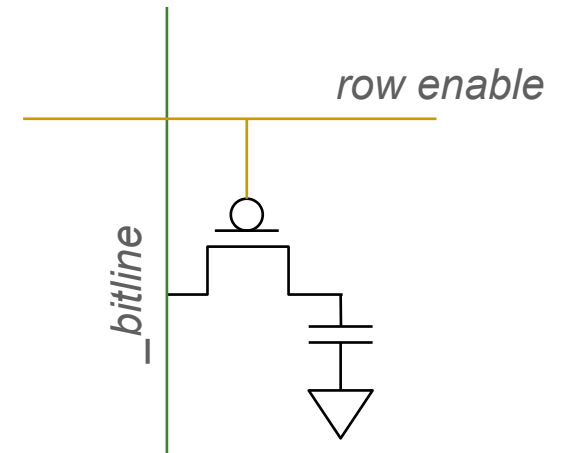
# Memory Technology: DRAM

- 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
- Capacitors will leak!



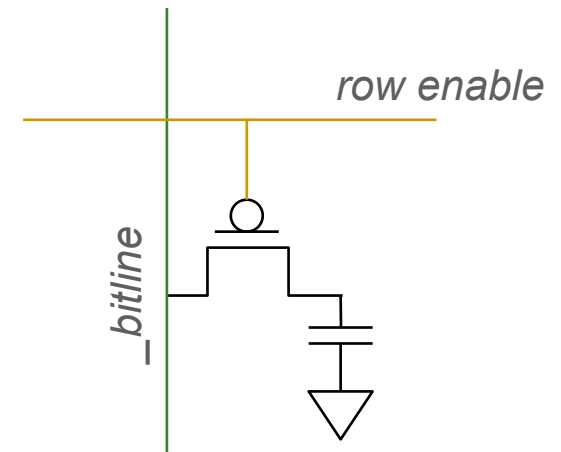
# Memory Technology: DRAM

- 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
- Capacitors will leak!
  - DRAM cell loses charge over time



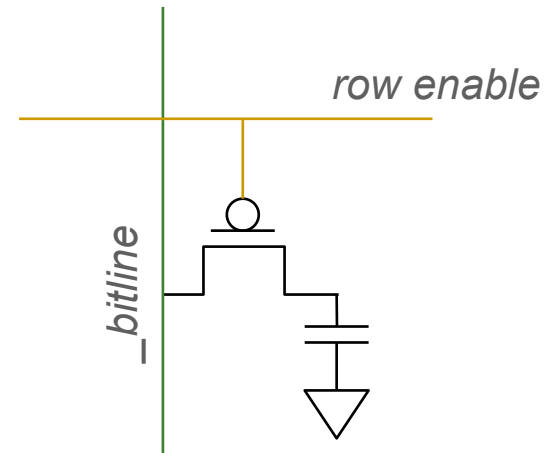
# Memory Technology: DRAM

- 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
- Capacitors will leak!
  - DRAM cell loses charge over time
  - DRAM cell needs to be **refreshed**.



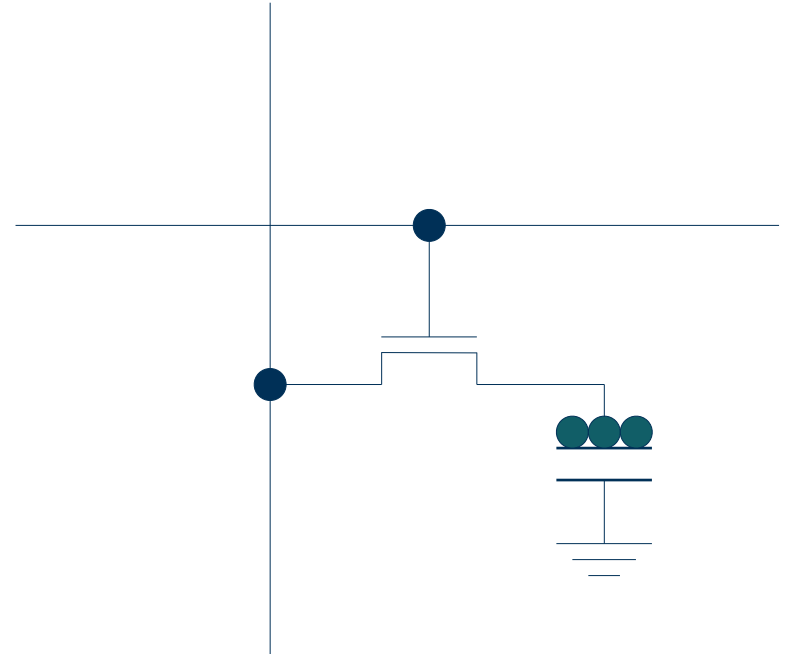
# Memory Technology: DRAM

- 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
- Capacitors will leak!
  - DRAM cell loses charge over time
  - DRAM cell needs to be **refreshed**.
  - 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.



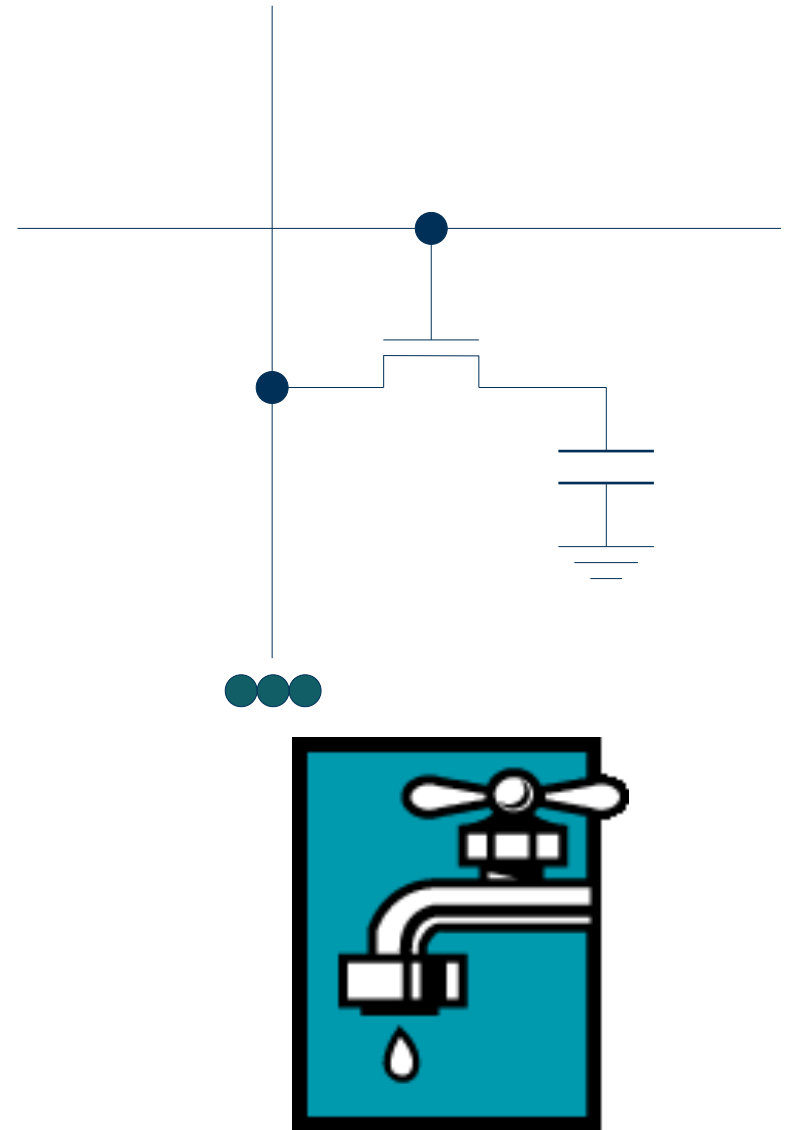
# DRAM Cell

- Capacitor holding value leaks, eventually you will lose information (everything turns to 0)
- How do you maintain the values in DRAM?
  - Refresh periodically
  - A major source for power consumption in DRAM



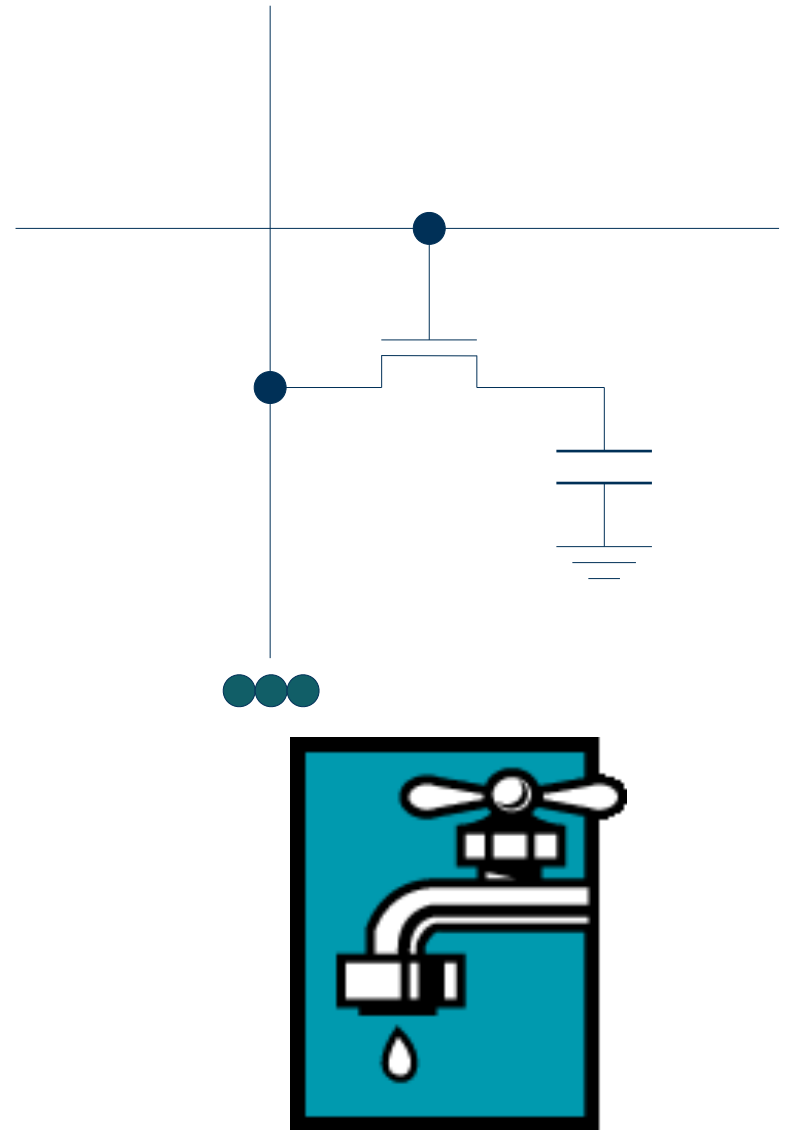
# DRAM Cell

- Capacitor holding value leaks, eventually you will lose information (everything turns to 0)
- How do you maintain the values in DRAM?
  - Refresh periodically
  - A major source for power consumption in DRAM



# DRAM Cell

- Capacitor holding value leaks, eventually you will lose information (everything turns to 0)
- How do you maintain the values in DRAM?
  - Refresh periodically
  - A major source for power consumption in DRAM



# 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)
  - 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

# 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

# The Problem

- **Bigger is slower**
  - SRAM, 512 Bytes, 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

# 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)
  - ensure most of the data the processor needs in the near future is kept in the fast(er) level(s)
- Question: How do we know what kind of data processors would use in the near future?

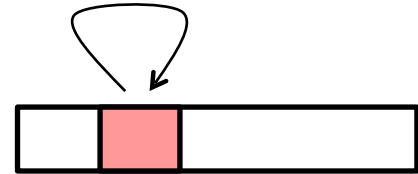
# Locality

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently



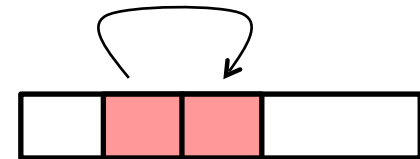
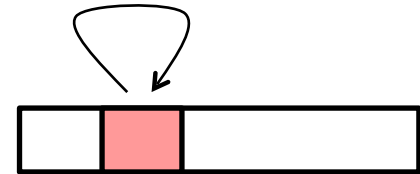
# Locality

- **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



# Locality

- **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

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

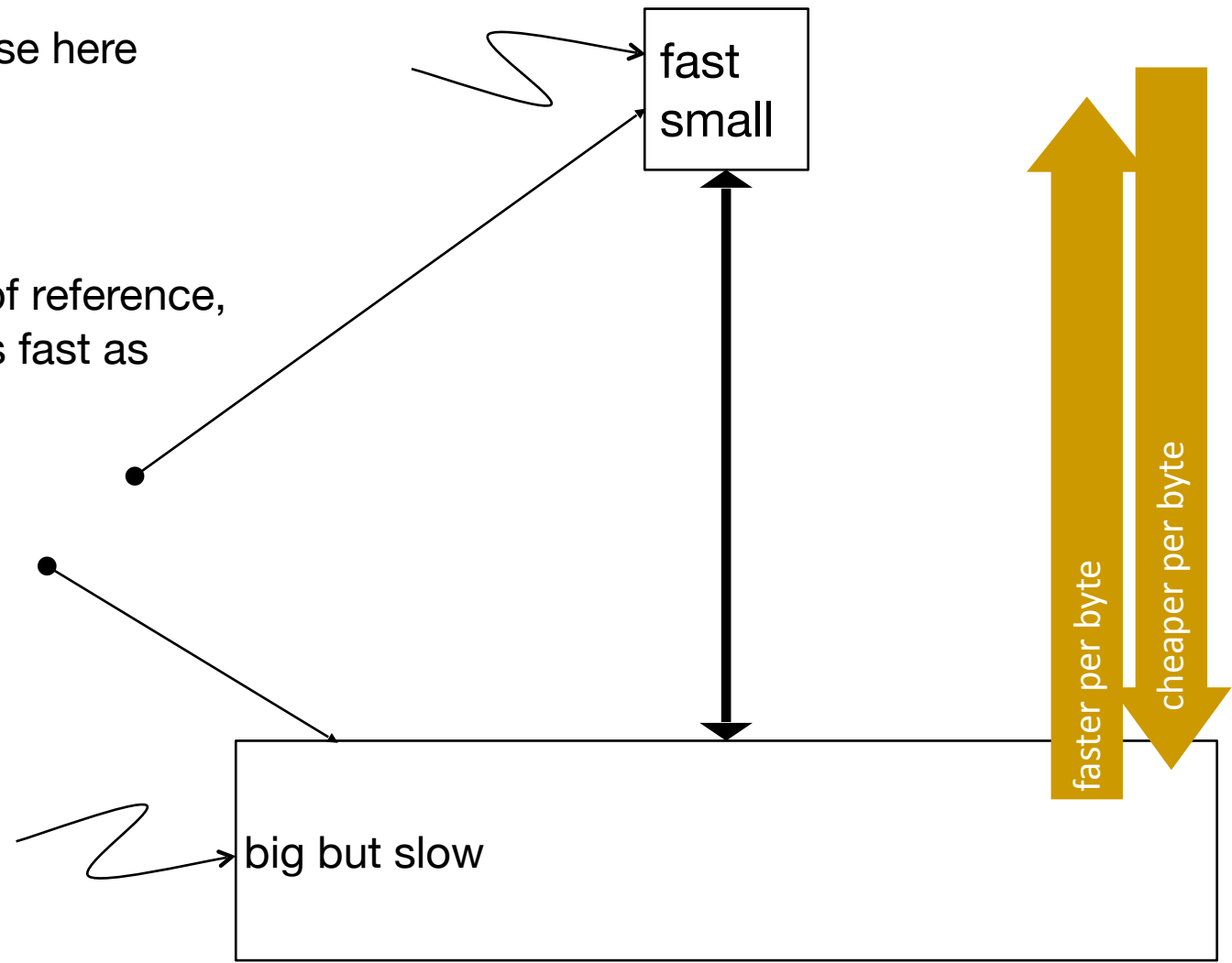
- Data references
  - **Spatial** Locality: Reference array elements in succession (stride-1 reference pattern)
  - **Temporal** Locality: Reference variable sum each iteration.
- Instruction references
  - **Spatial** Locality: Reference instructions in sequence.
  - **Temporal** Locality: Cycle through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

# Memory Hierarchy

move what you use here

With good locality of reference,  
memory appears as fast as  
and as large as

backup  
everything  
here

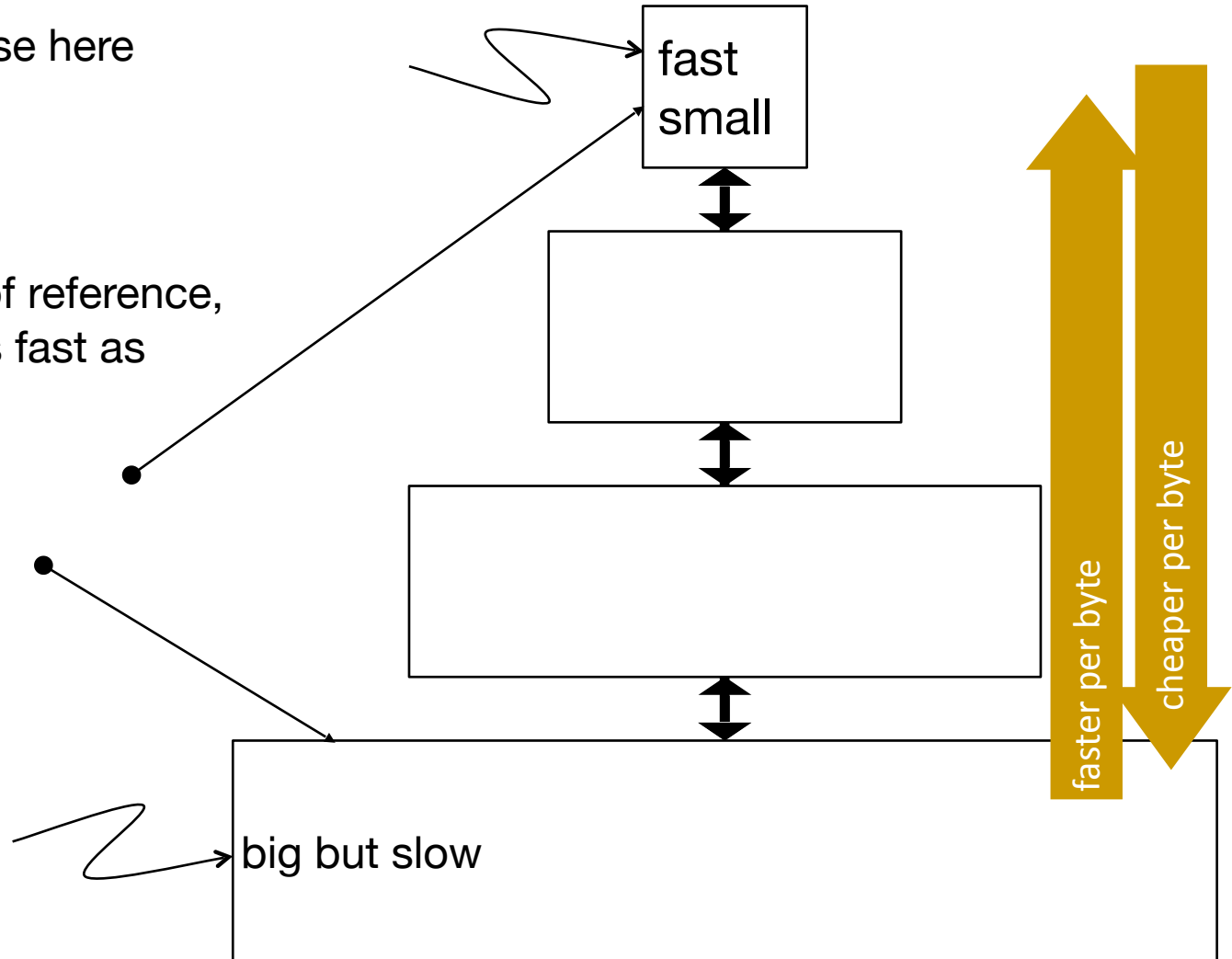


# Memory Hierarchy

move what you use here

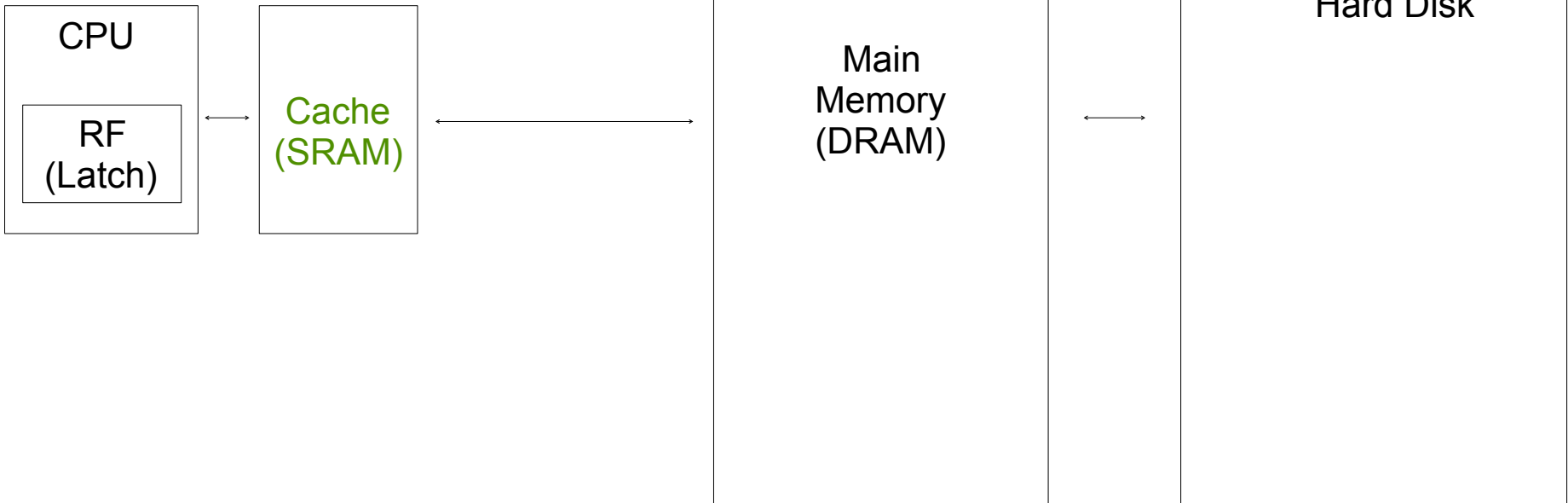
With good locality of reference,  
memory appears as fast as  
and as large as

backup  
everything  
here



# Memory Hierarchy

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



# The Bookshelf Analogy

- Book in your hand
- Desk
- Bookshelf
- Boxes at home
- Boxes in library
  
- Recently-used books tend to stay on desk
  - Comp Org. books, books for classes you are currently taking
    - Until the desk gets full
- Adjacent books in the shelf needed around the same time

# 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

---

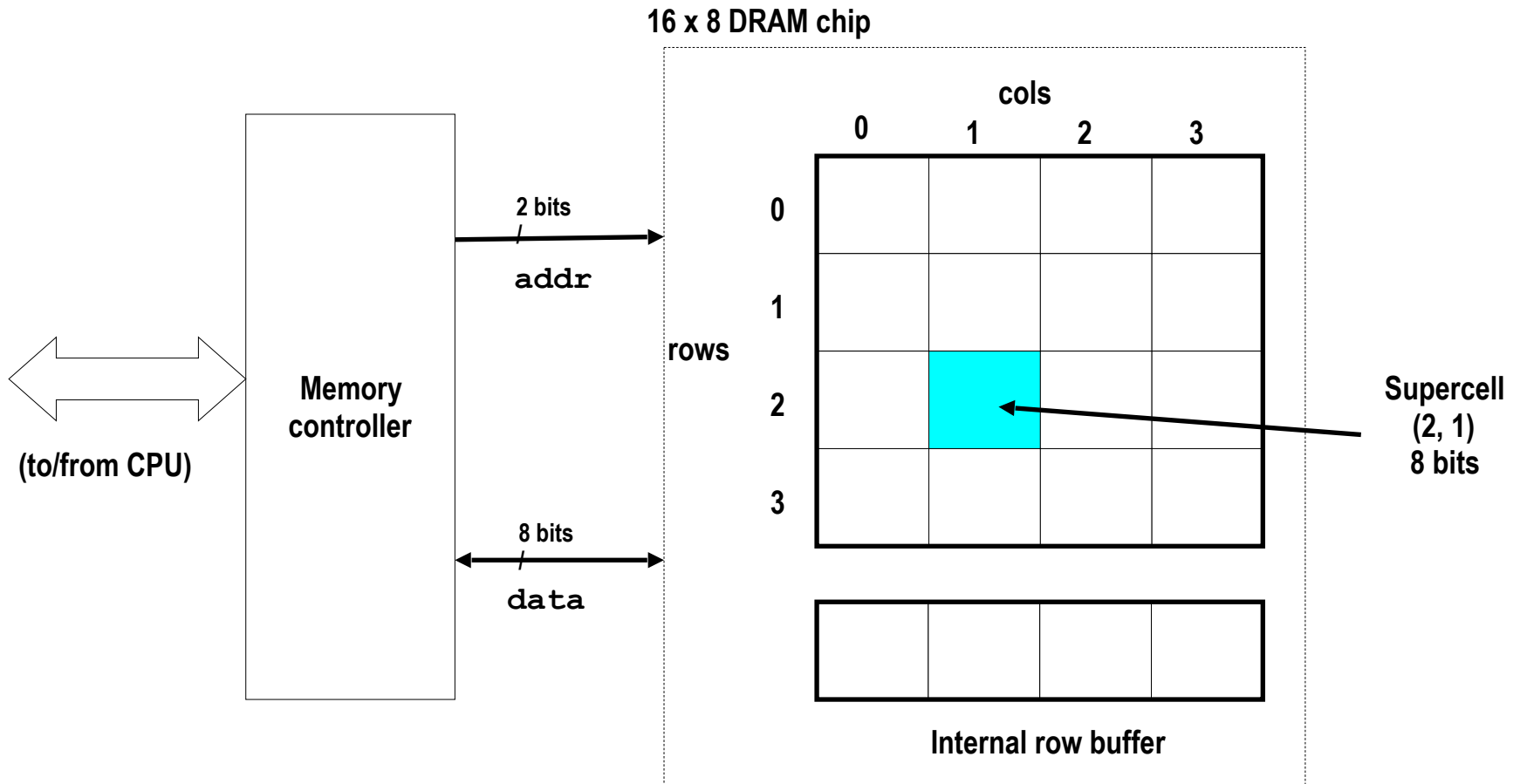
Hard Disk  
100 GB, ~10 msec



# This Module (3 Lectures)

- Today: Memory Hierarchy Overview
  - Trade-offs between different memory technologies
  - Exploiting locality to get the best of all worlds
  - SRAM/DRAM Hardware Basics
- Cache
- Memory-oriented Program Optimizations

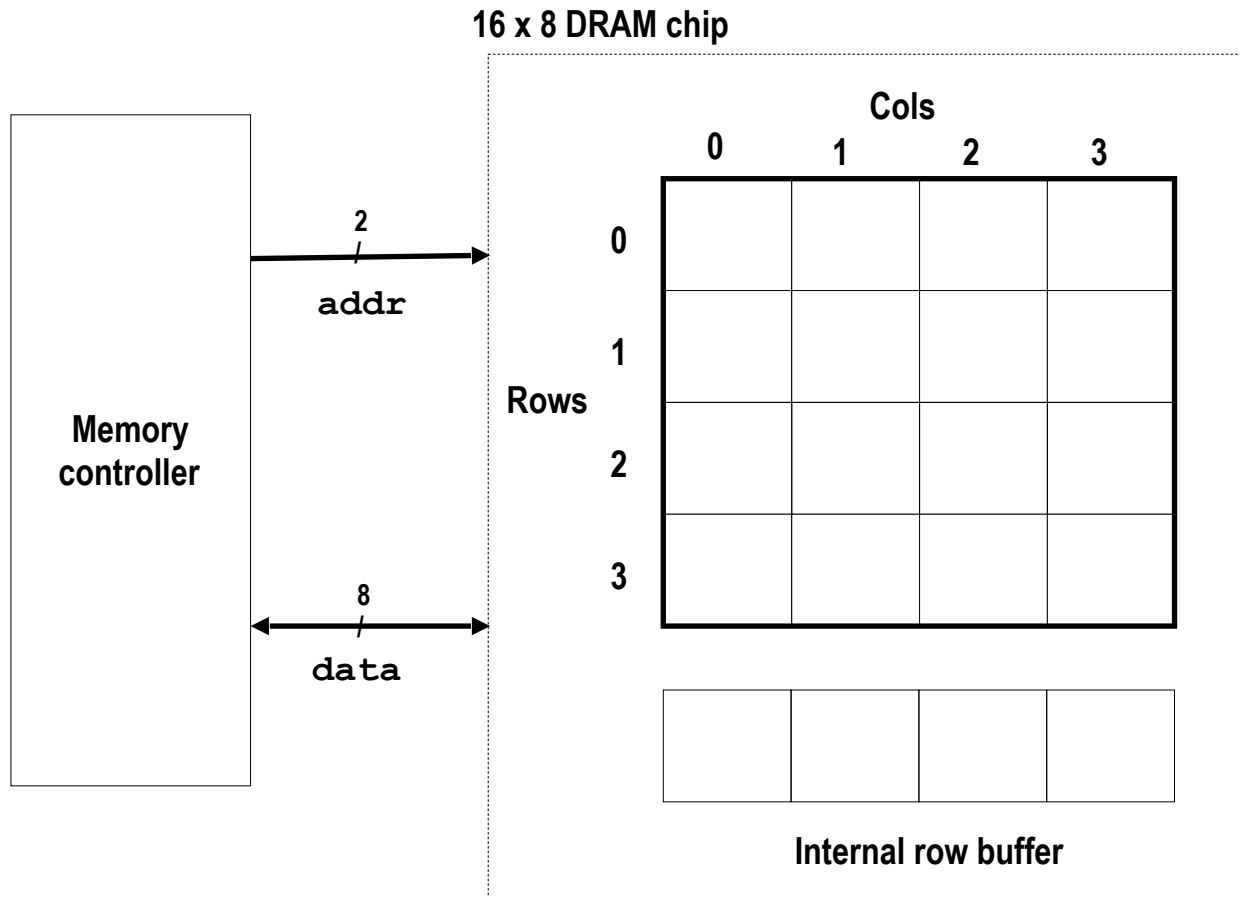
# DRAM Chip Organization



# Reading DRAM Supercell (2,1)

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

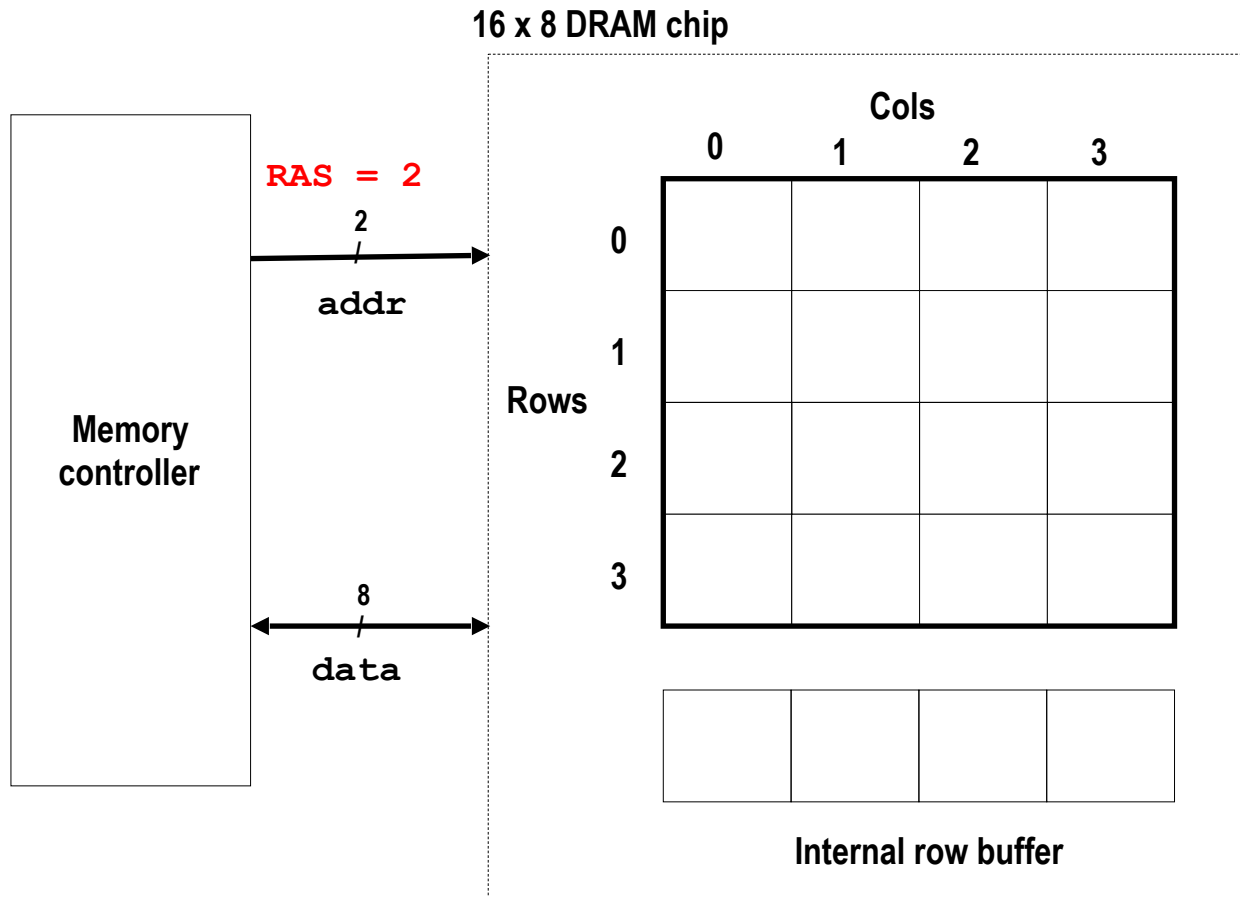
Step 1(b): Row 2 copied from DRAM array to row buffer.



# Reading DRAM Supercell (2,1)

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

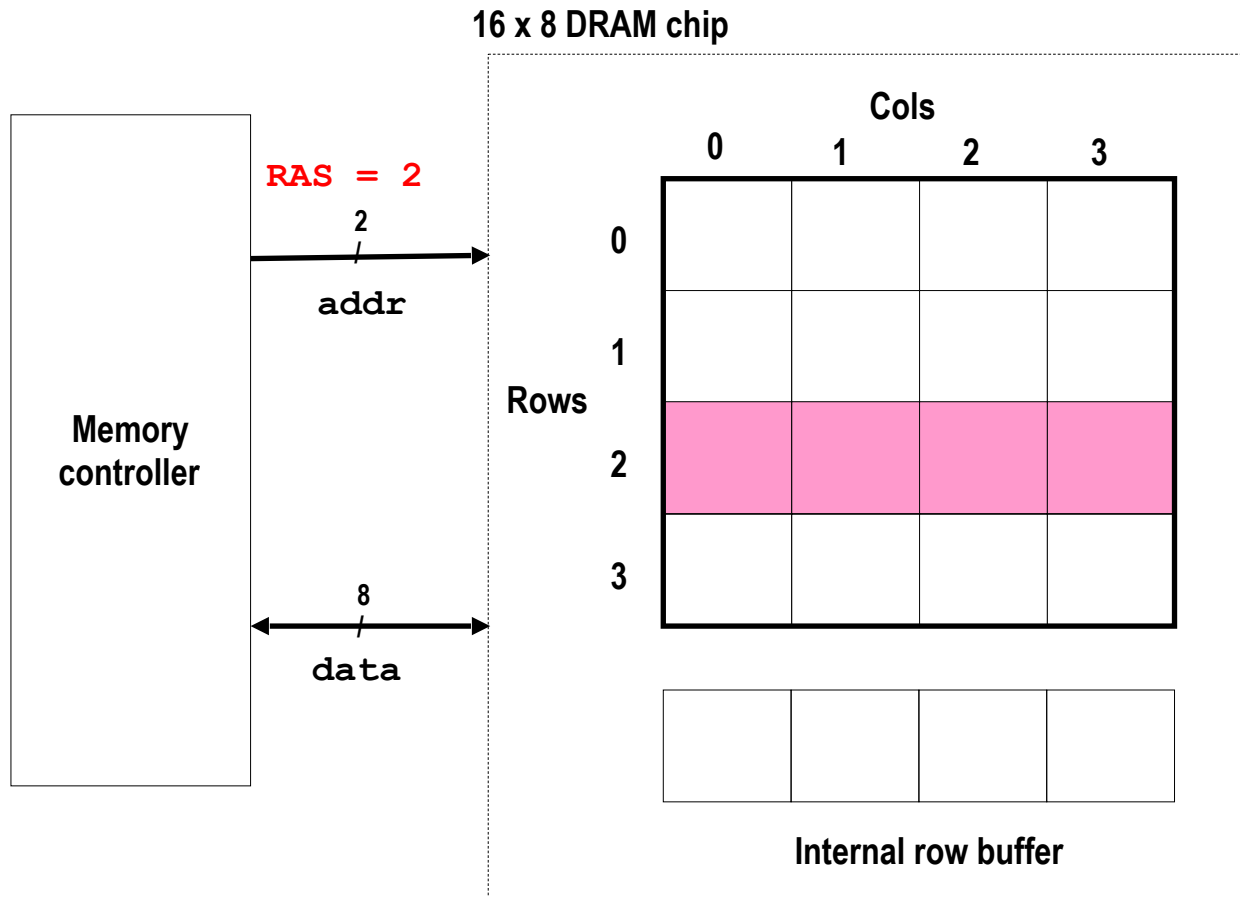
Step 1(b): Row 2 copied from DRAM array to row buffer.



# Reading DRAM Supercell (2,1)

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

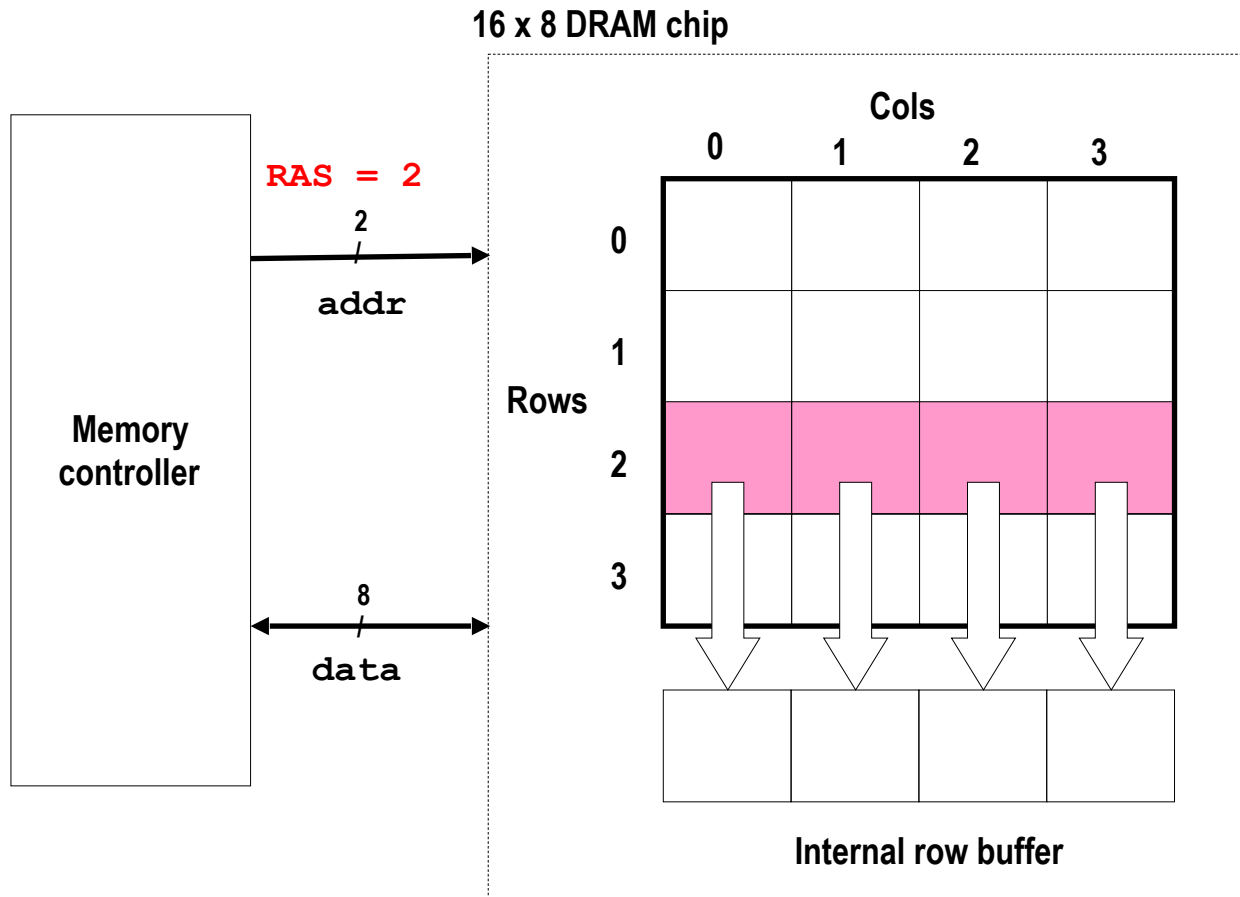
Step 1(b): Row 2 copied from DRAM array to row buffer.



# Reading DRAM Supercell (2,1)

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

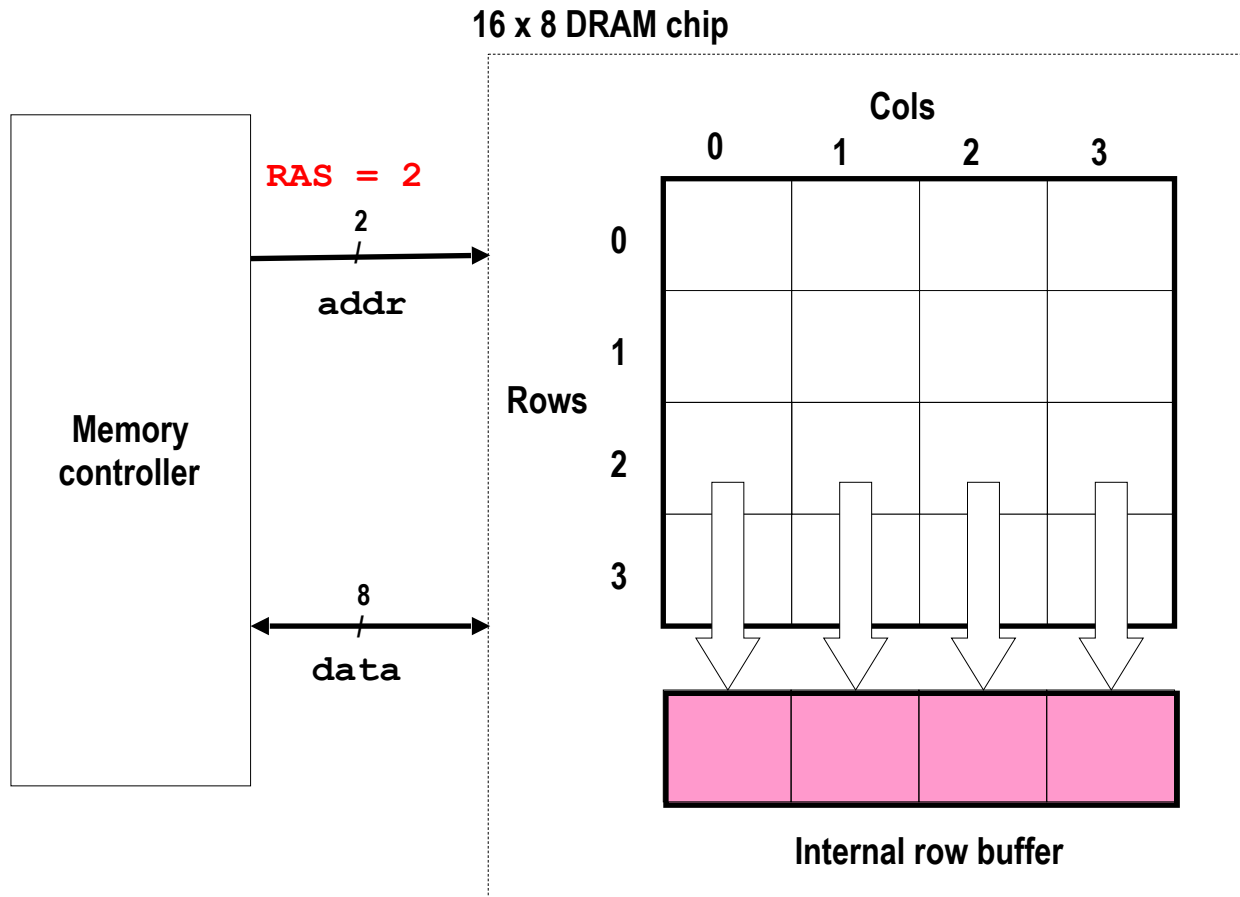
Step 1(b): Row 2 copied from DRAM array to row buffer.



# Reading DRAM Supercell (2,1)

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

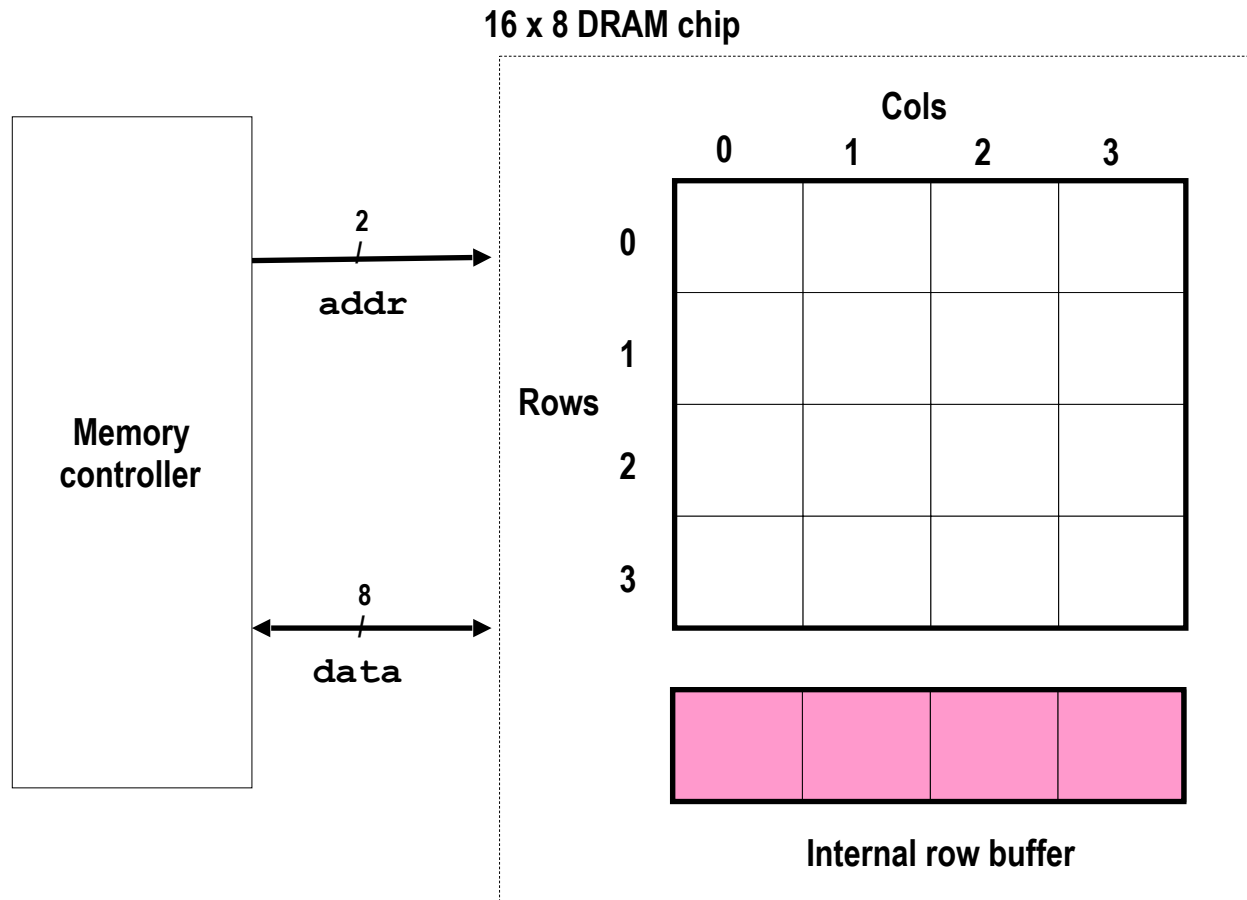
Step 1(b): Row 2 copied from DRAM array to row buffer.



# 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.

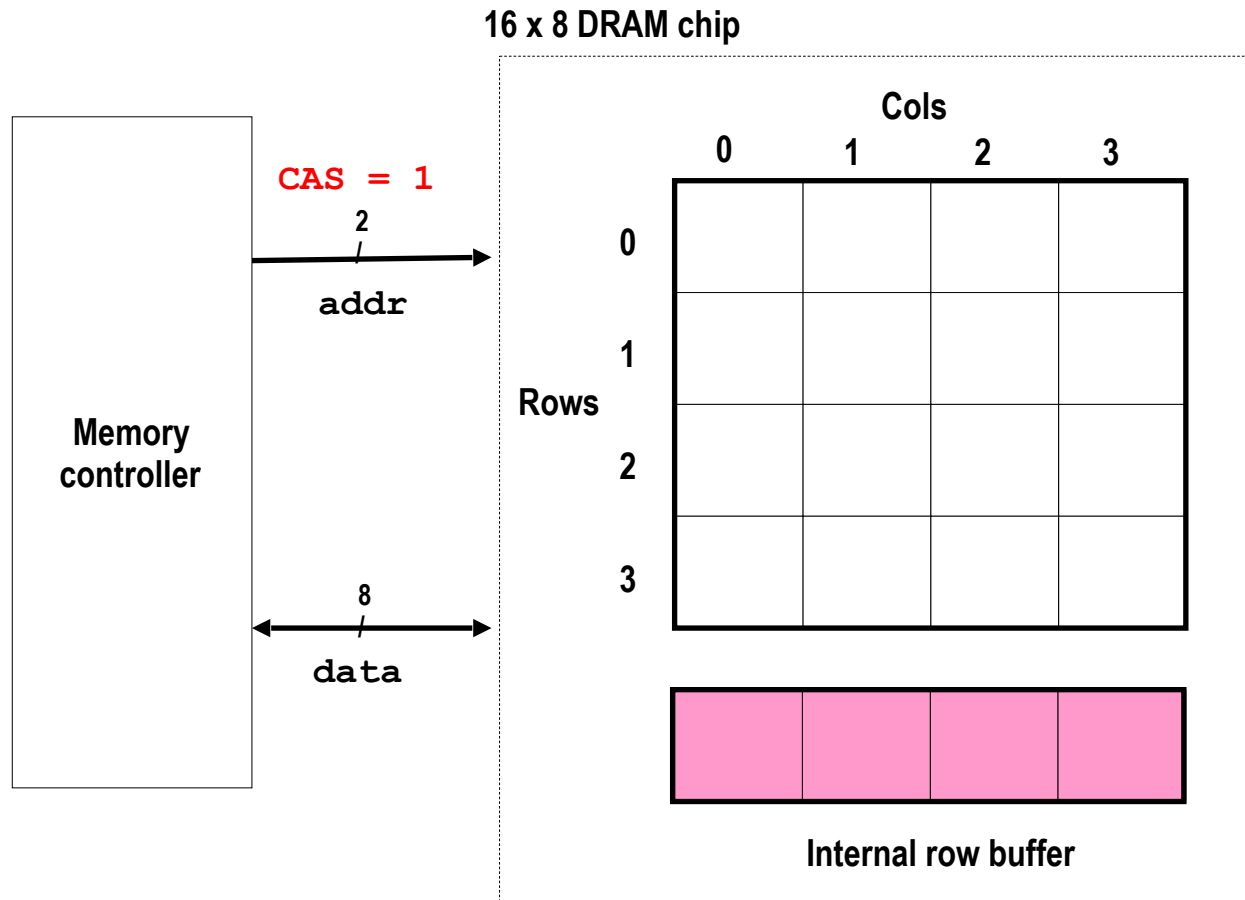




# Reading DRAM Supercell (2,1)

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

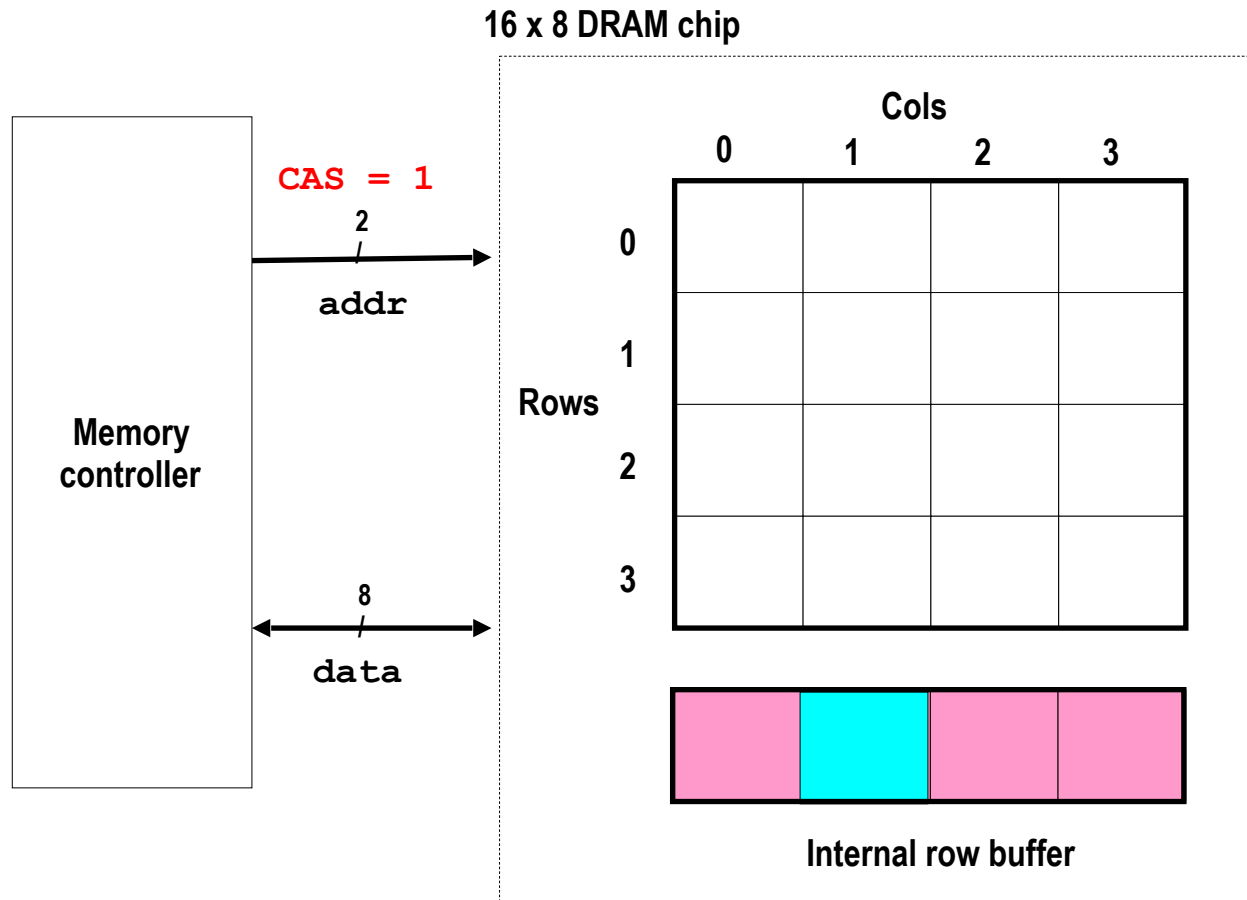
Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.



# Reading DRAM Supercell (2,1)

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

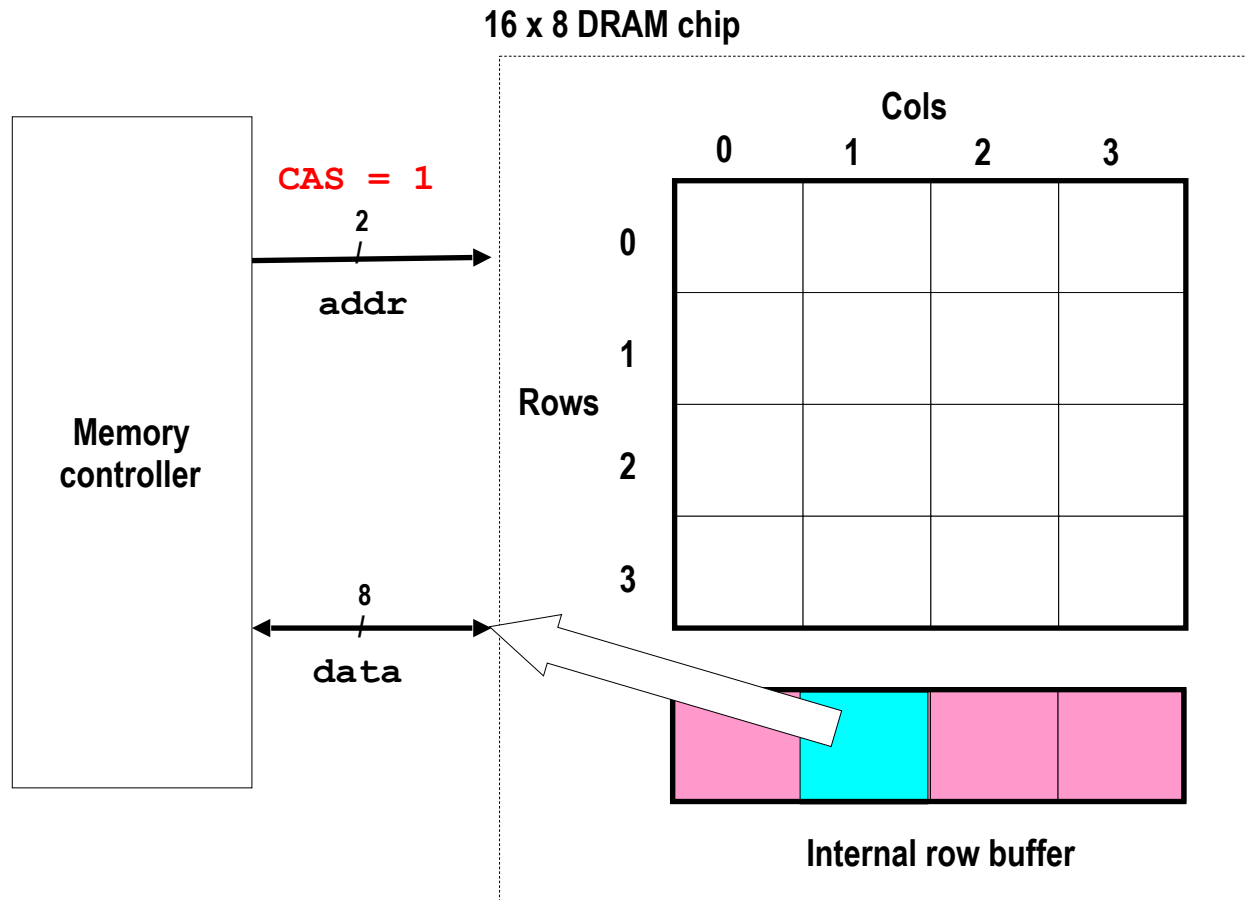
Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.



# Reading DRAM Supercell (2,1)

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

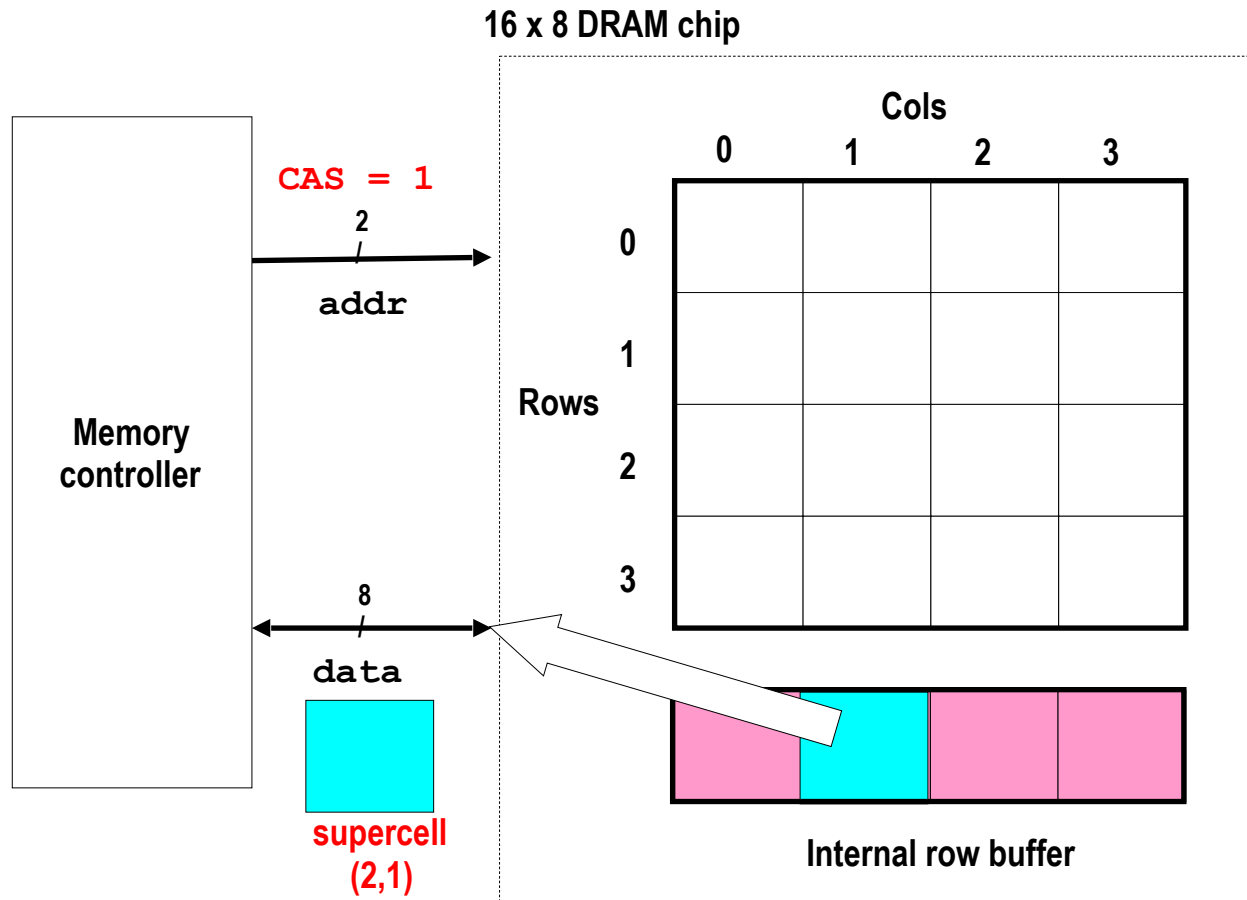
Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.



# Reading DRAM Supercell (2,1)

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

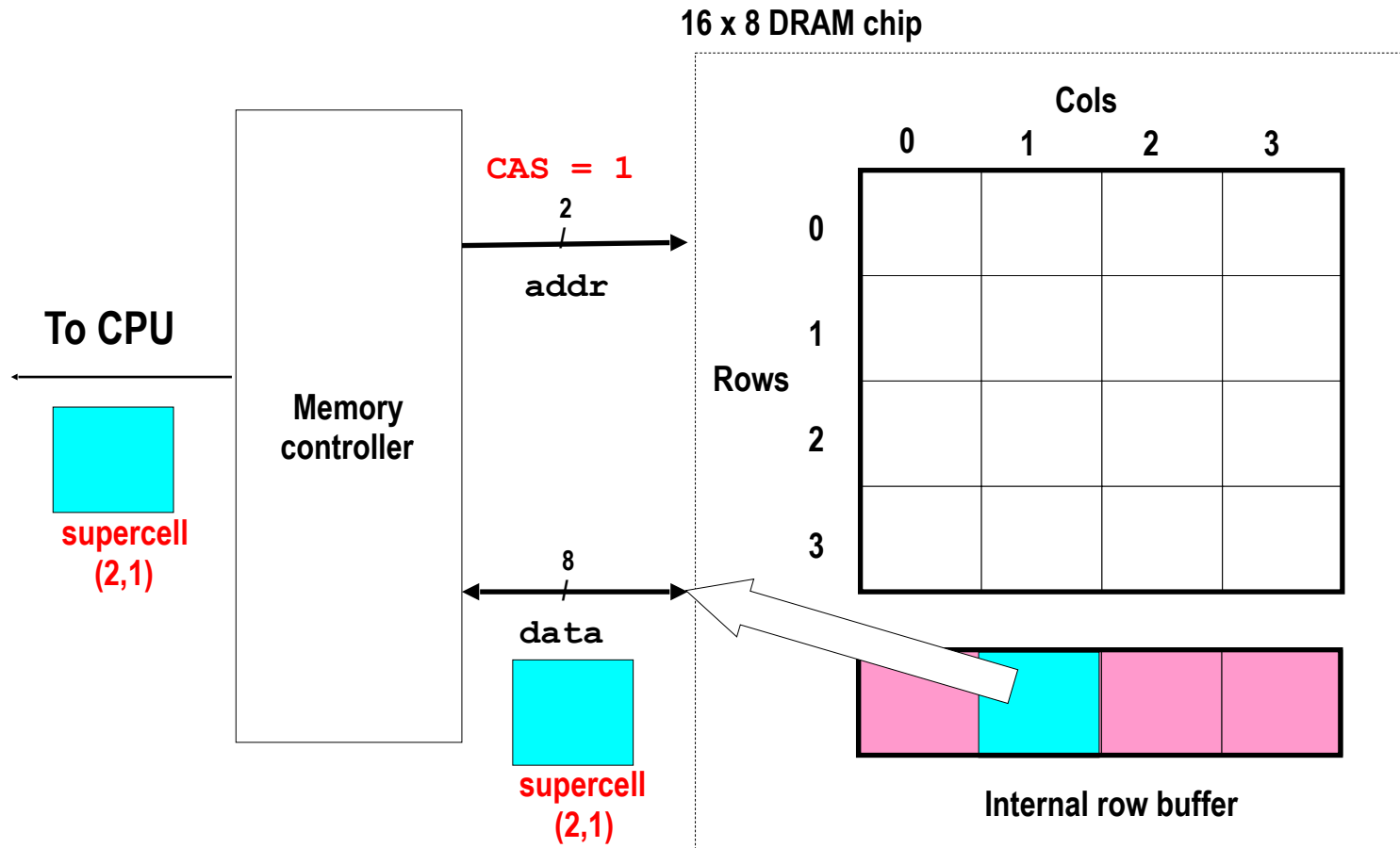
Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.



# Reading DRAM Supercell (2,1)

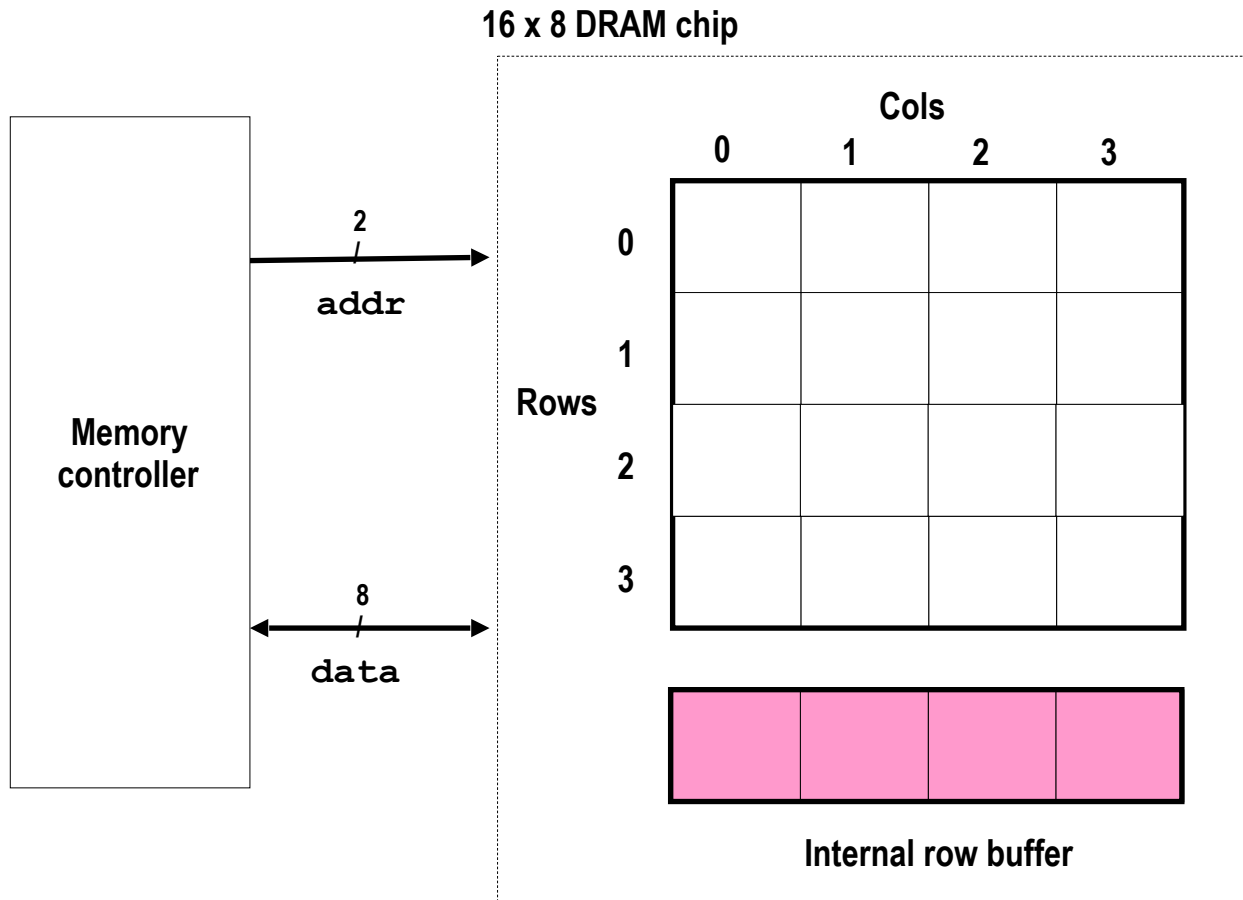
Step 2(a): Column access strobe (**CAS**) selects column 1.

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



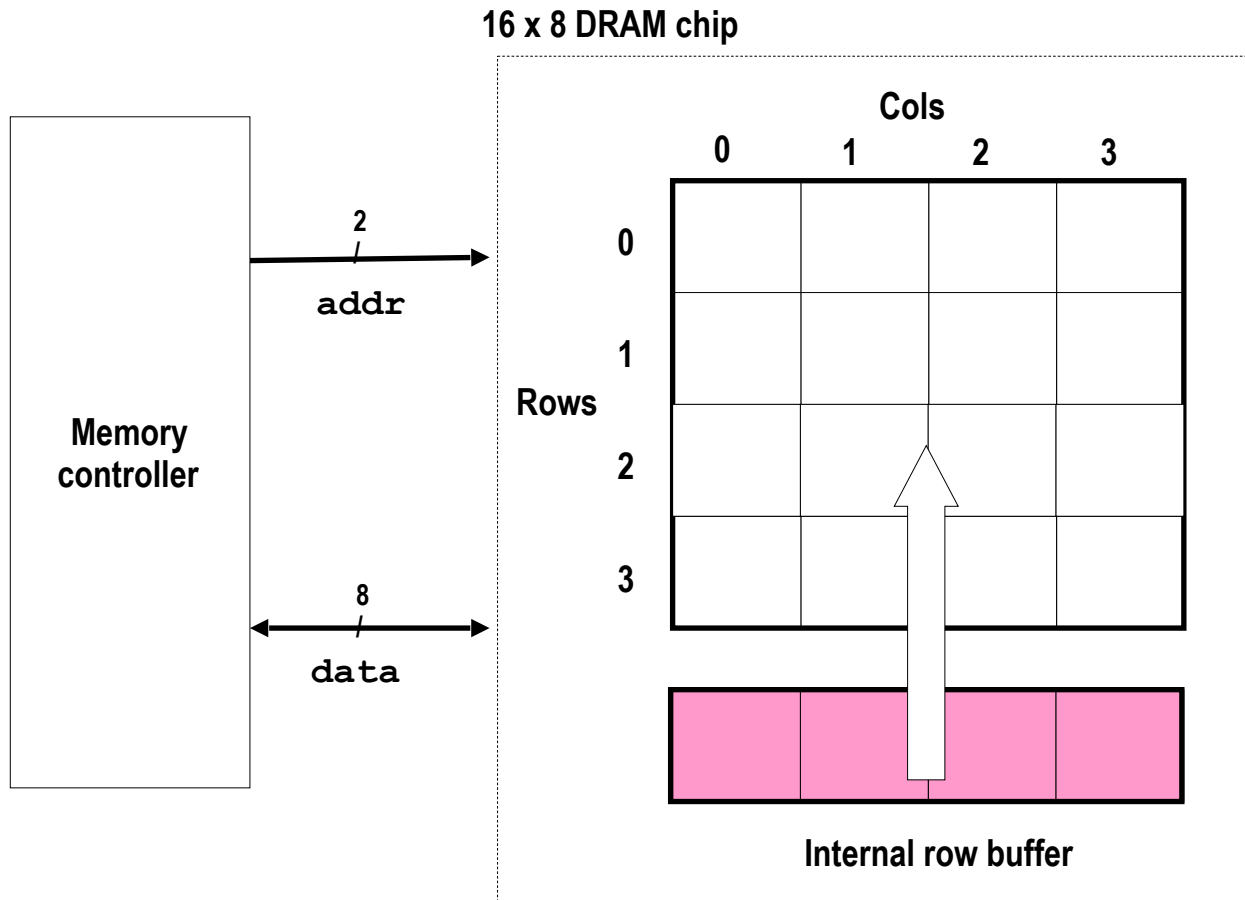
# Reading DRAM Supercell (2,1)

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.



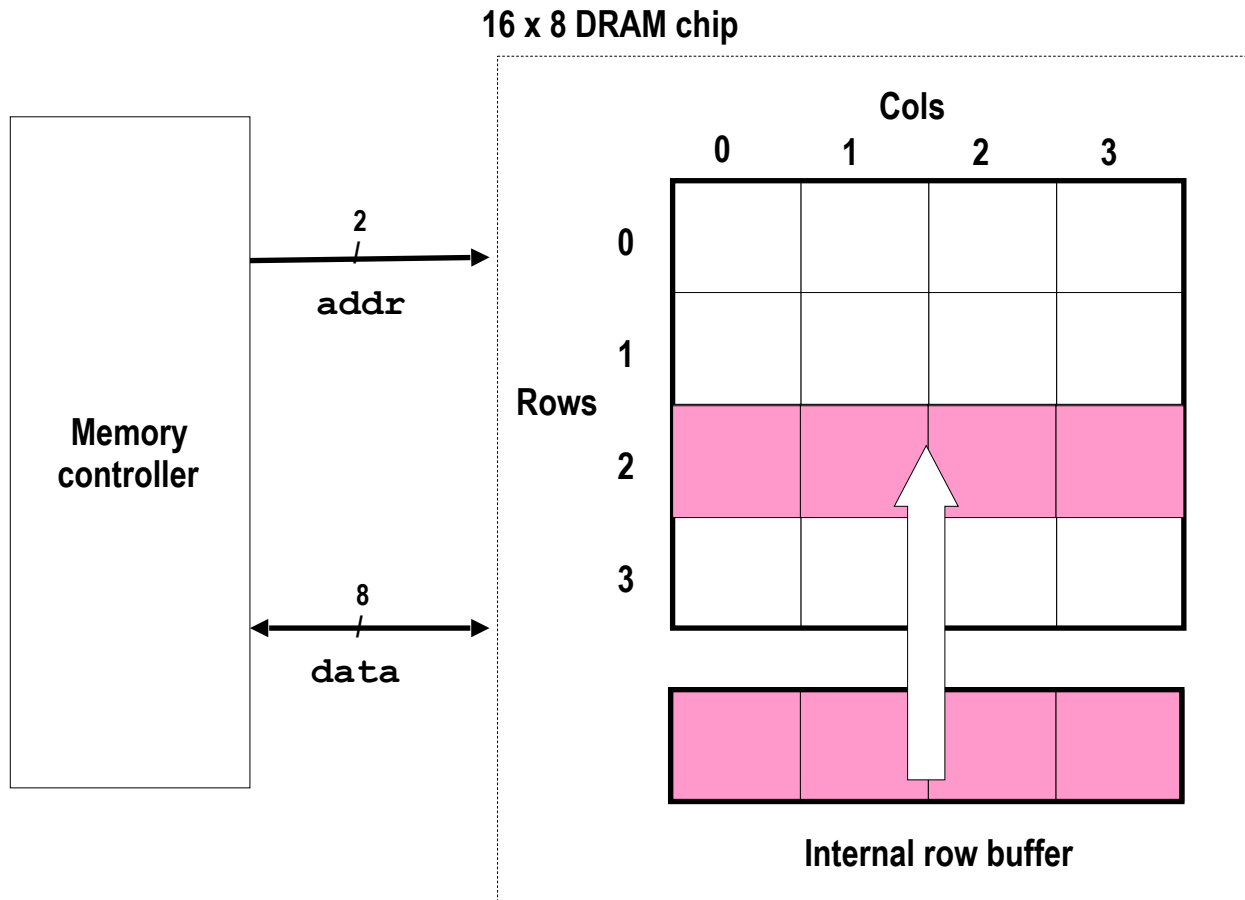
# Reading DRAM Supercell (2,1)

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.



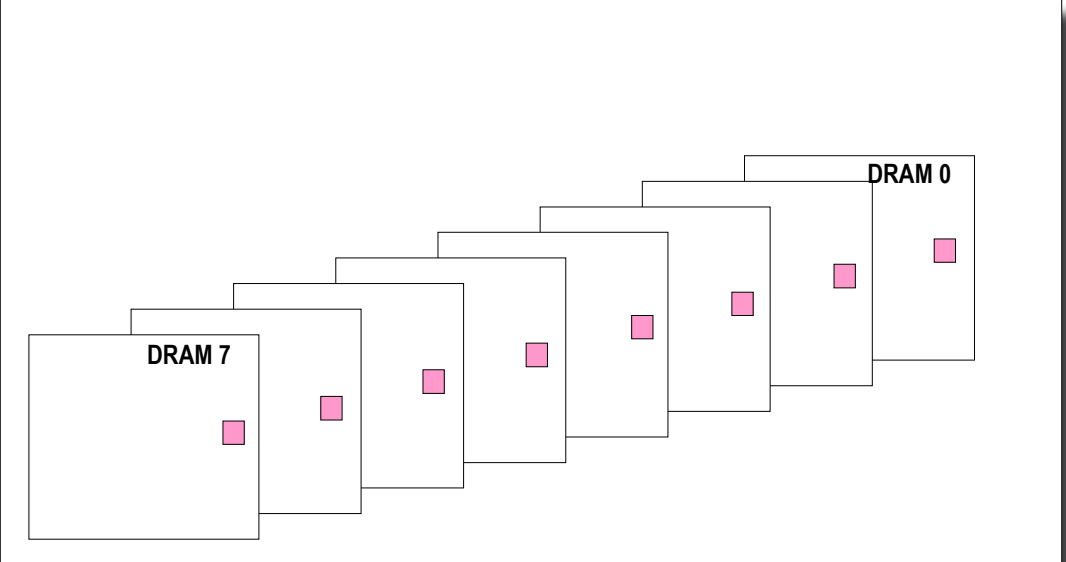
# Reading DRAM Supercell (2,1)

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.





# Memory Modules



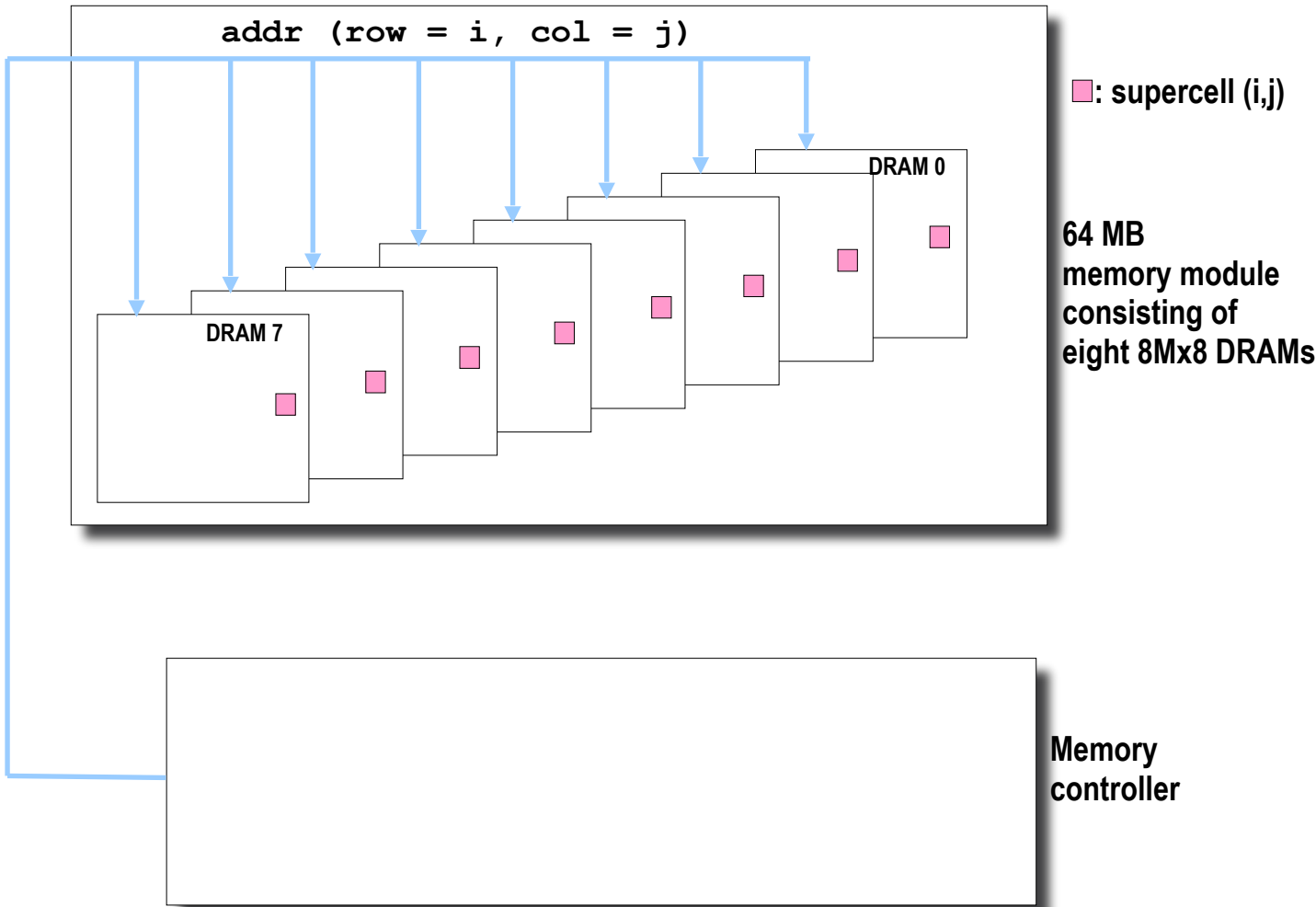
■: supercell (i,j)

64 MB  
memory module  
consisting of  
eight 8Mx8 DRAMs

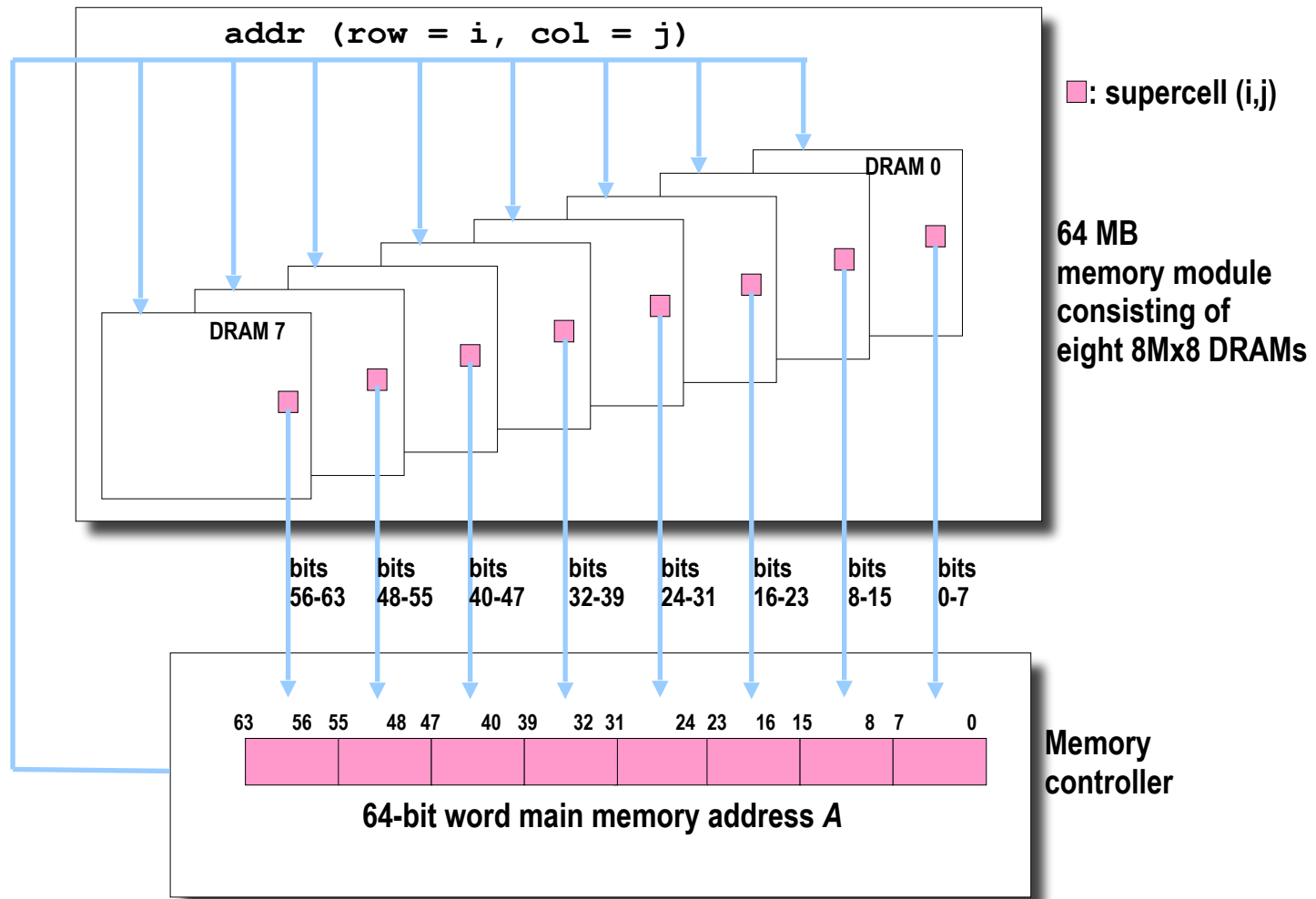


Memory  
controller

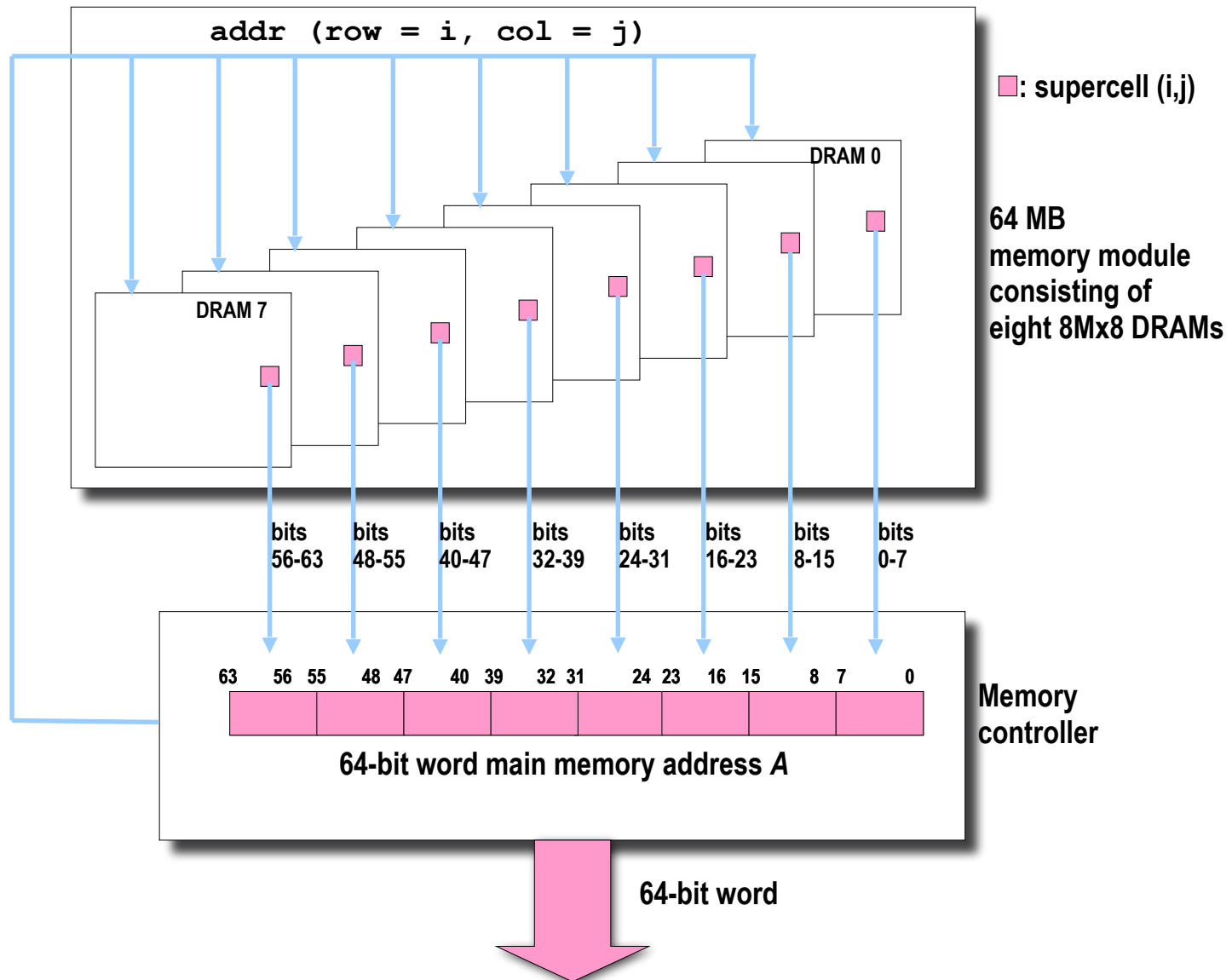
# Memory Modules



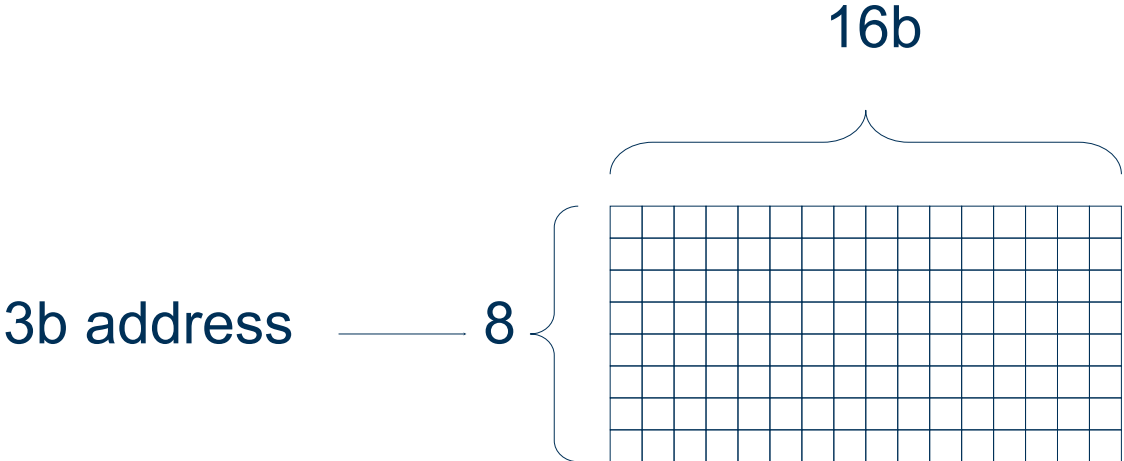
# Memory Modules



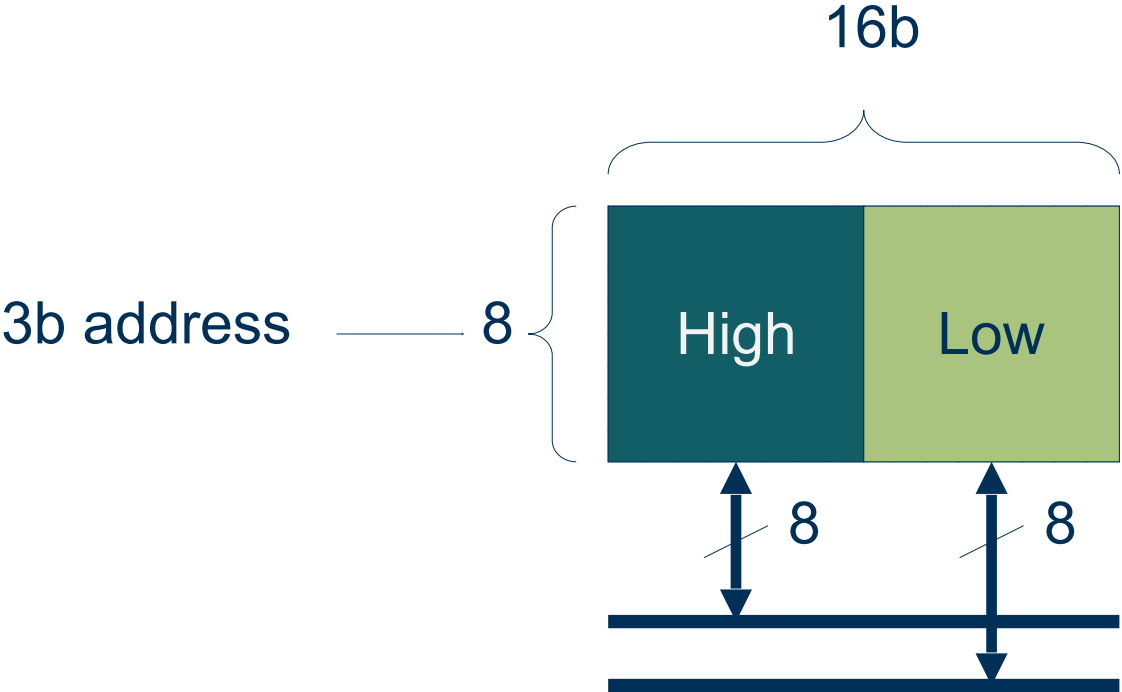
# Memory Modules



# Memory Layout Across Two Chips

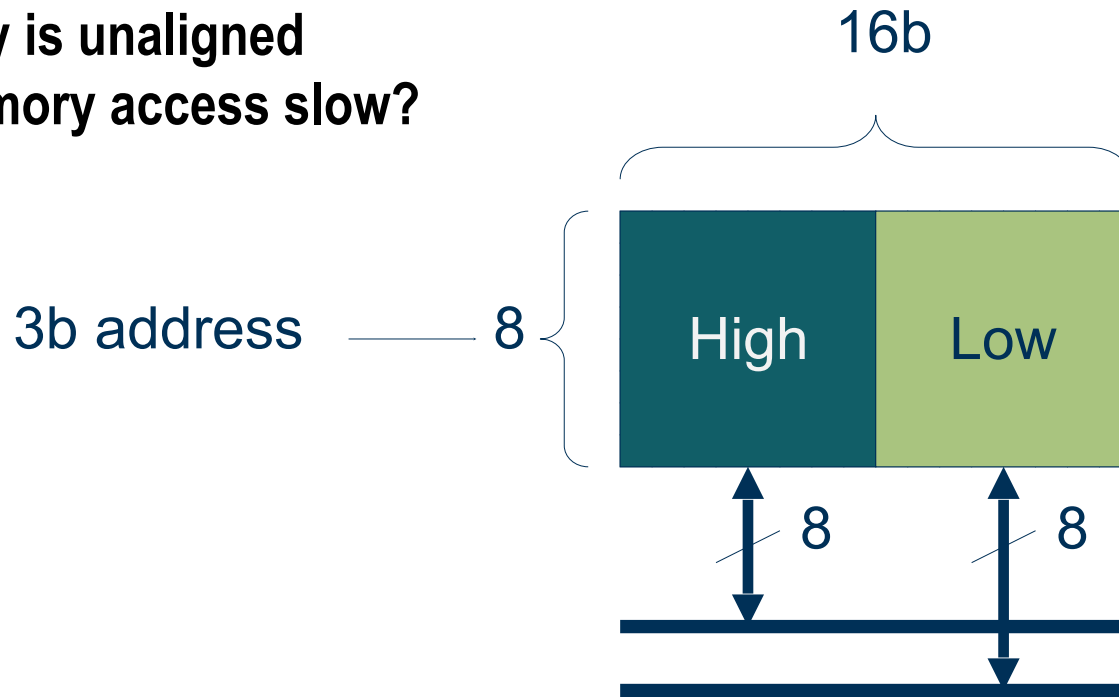


# Memory Layout Across Two Chips



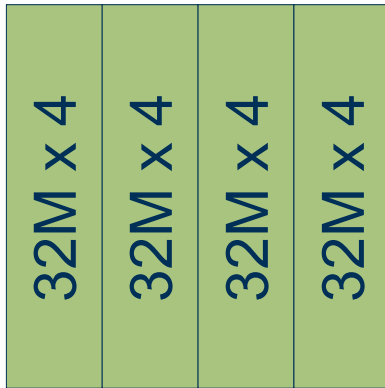
# Memory Layout Across Two Chips

Why is unaligned memory access slow?



# Narrower RAMs Enable Greater Capacity

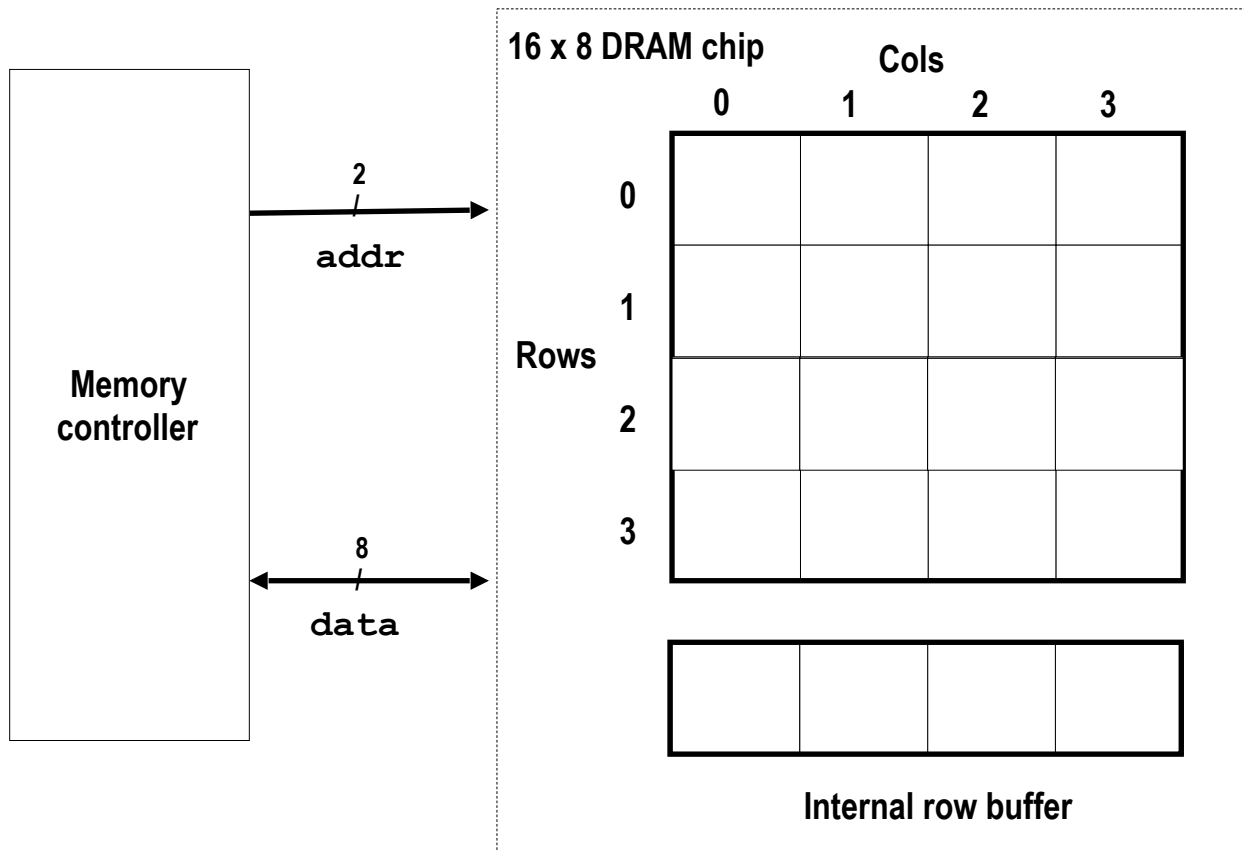
- Given Constant Total Width (pins)
- Multiple smaller chips are more reliable than one big chip
  - Yield rate issue





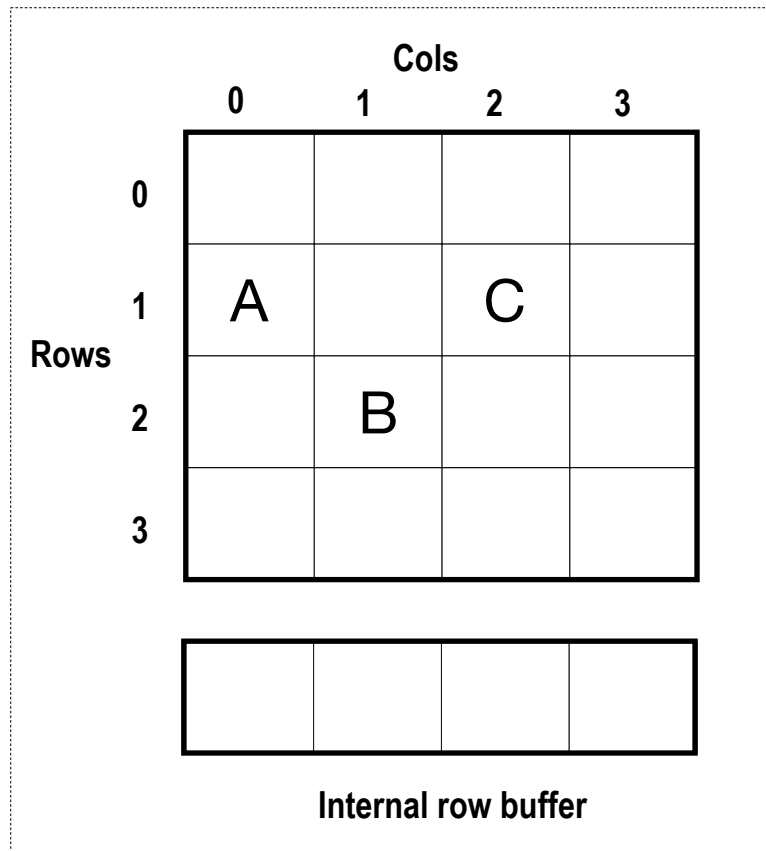
# Why Split Address into Row and Column?

- +: Reduce the number of address pins
- +: Also allow reading multiple columns within the same row
- -: Send address in two steps, increase latency



# Memory Scheduling

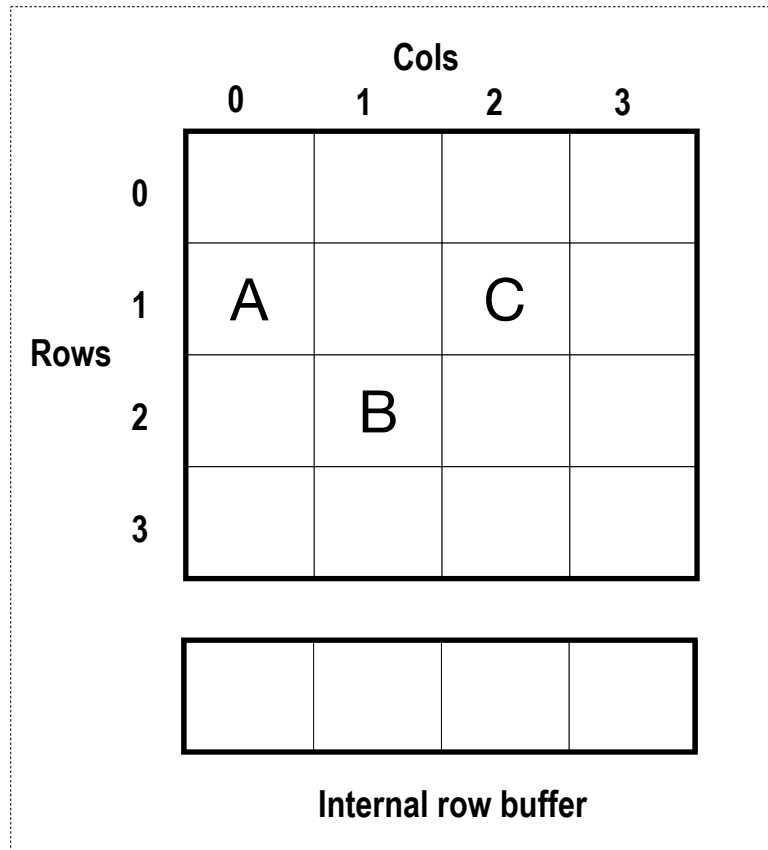
16 x 8 DRAM chip



# Memory Scheduling

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

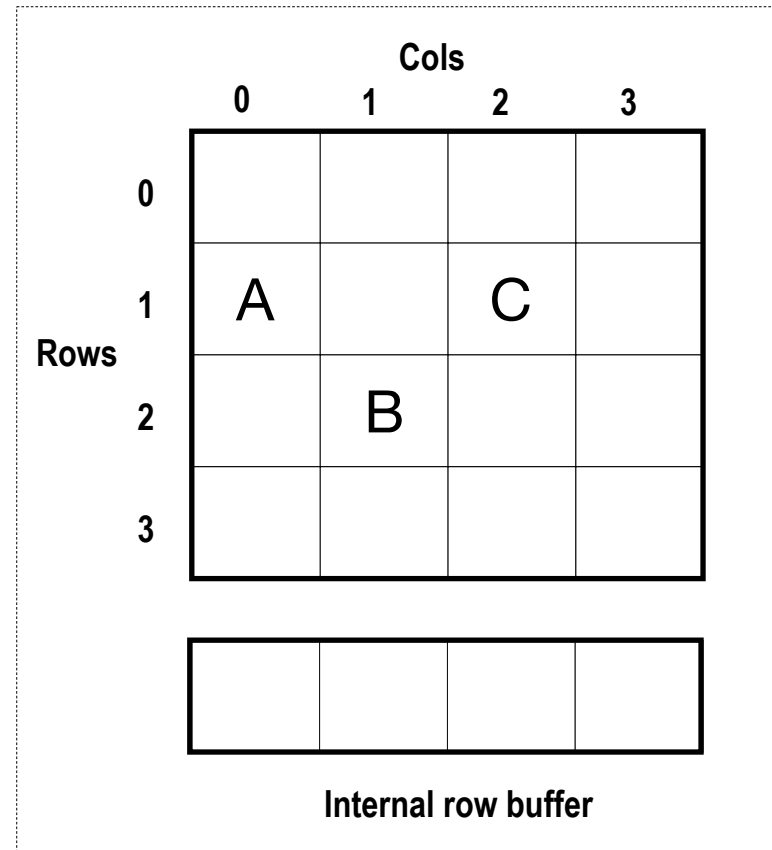
16 x 8 DRAM chip



# Memory Scheduling

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

16 x 8 DRAM chip



# Memory Scheduling

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

