CS 33

Memory Hierarchy I

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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
- Static RAM (SRAM)
  - each cell stores a bit with a four- or six-transistor circuit
  - retains value indefinitely, as long as it is kept powered
  - relatively insensitive to electrical noise (EMI), radiation, etc.
  - faster and more expensive than DRAM
- Dynamic RAM (DRAM)
  - each cell stores bit with a capacitor; transistor is used for access
  - value must be refreshed every 10-100 ms
  - more sensitive to disturbances (EMI, radiation,…) than SRAM
  - slower and cheaper than SRAM

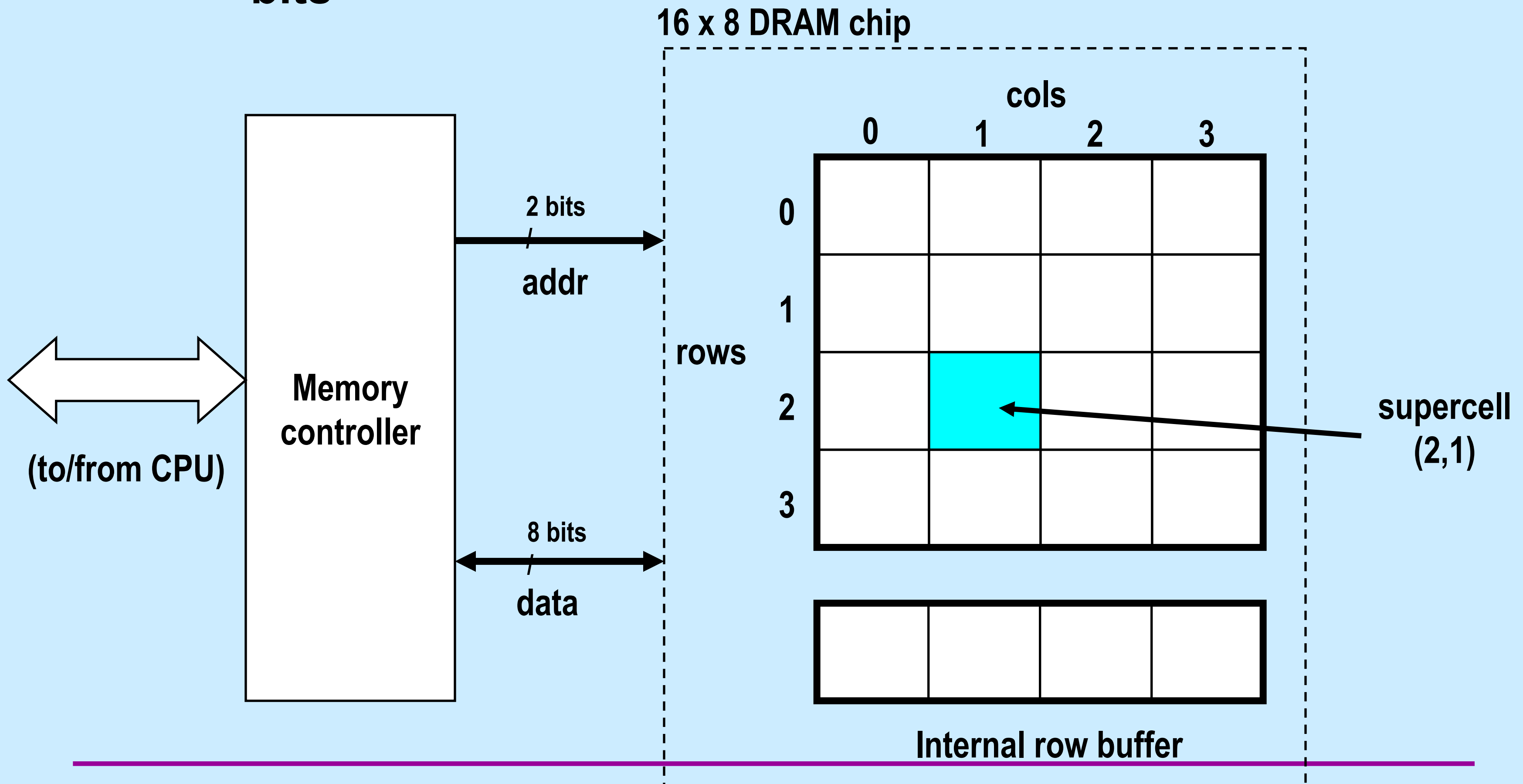
SRAM vs DRAM Summary

	Trans. per bit	Access time	Needs refresh?	Needs EDC?	Cost	Applications
SRAM	4 or 6	1X	No	Maybe	100x	Cache memories
DRAM	1	10X	Yes	Yes	1X	Main memories, frame buffers

- **EDC = error detection and correction**
  - **to cope with noise, etc.**

Conventional DRAM Organization

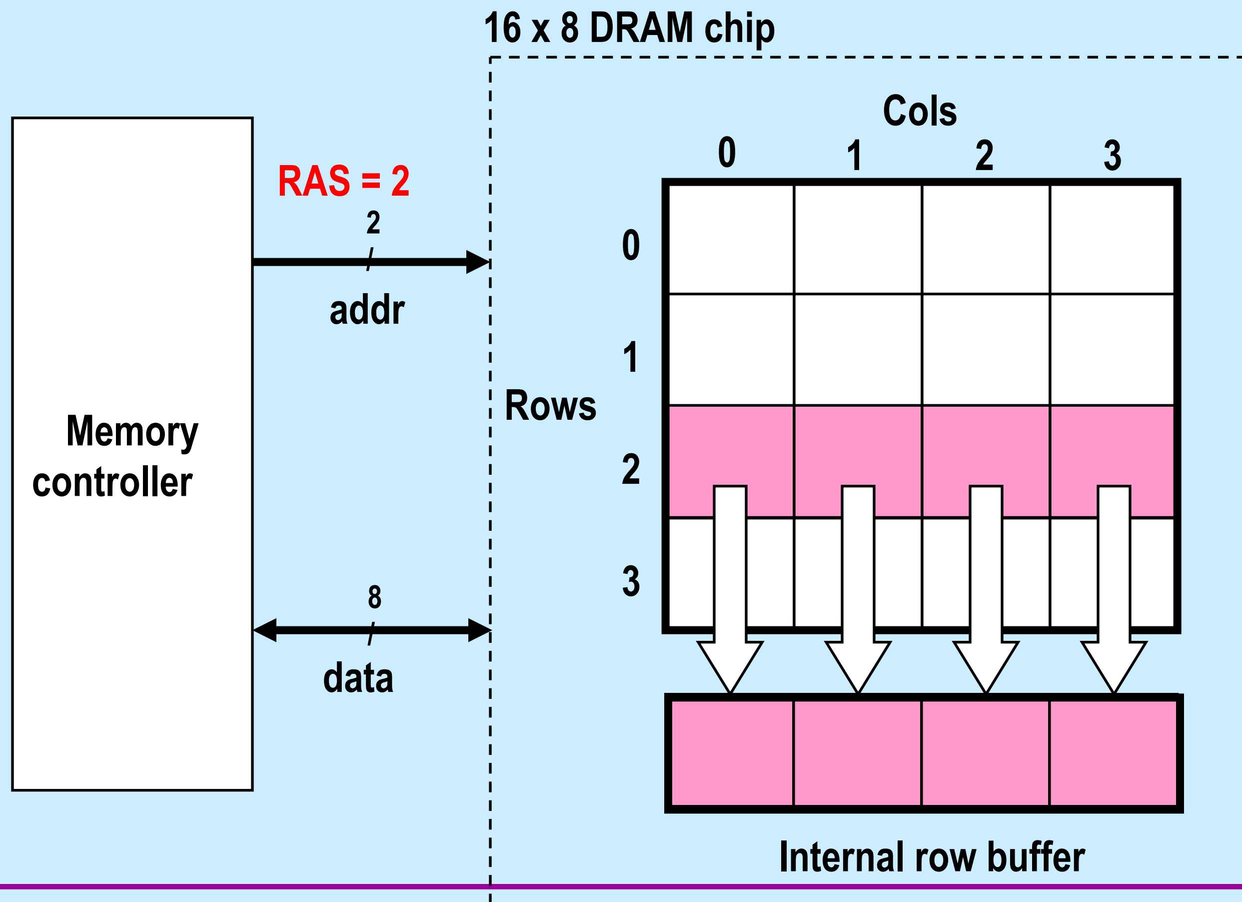
- **d x w DRAM:**
  - dw total bits organized as d **supercells** of size w bits



Reading DRAM Supercell (2,1)

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

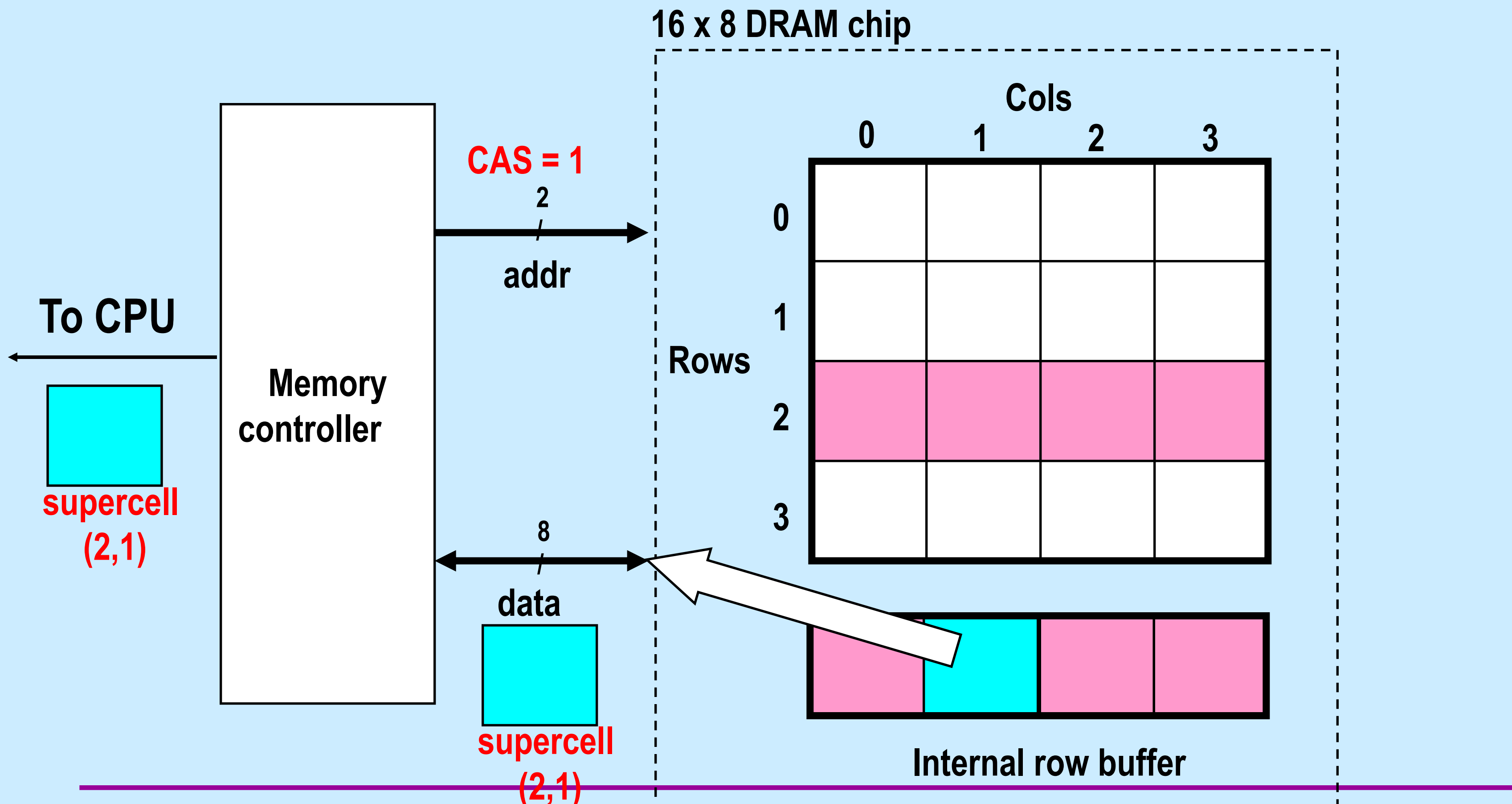
Step 1(b): row 2 copied from DRAM array to row buffer



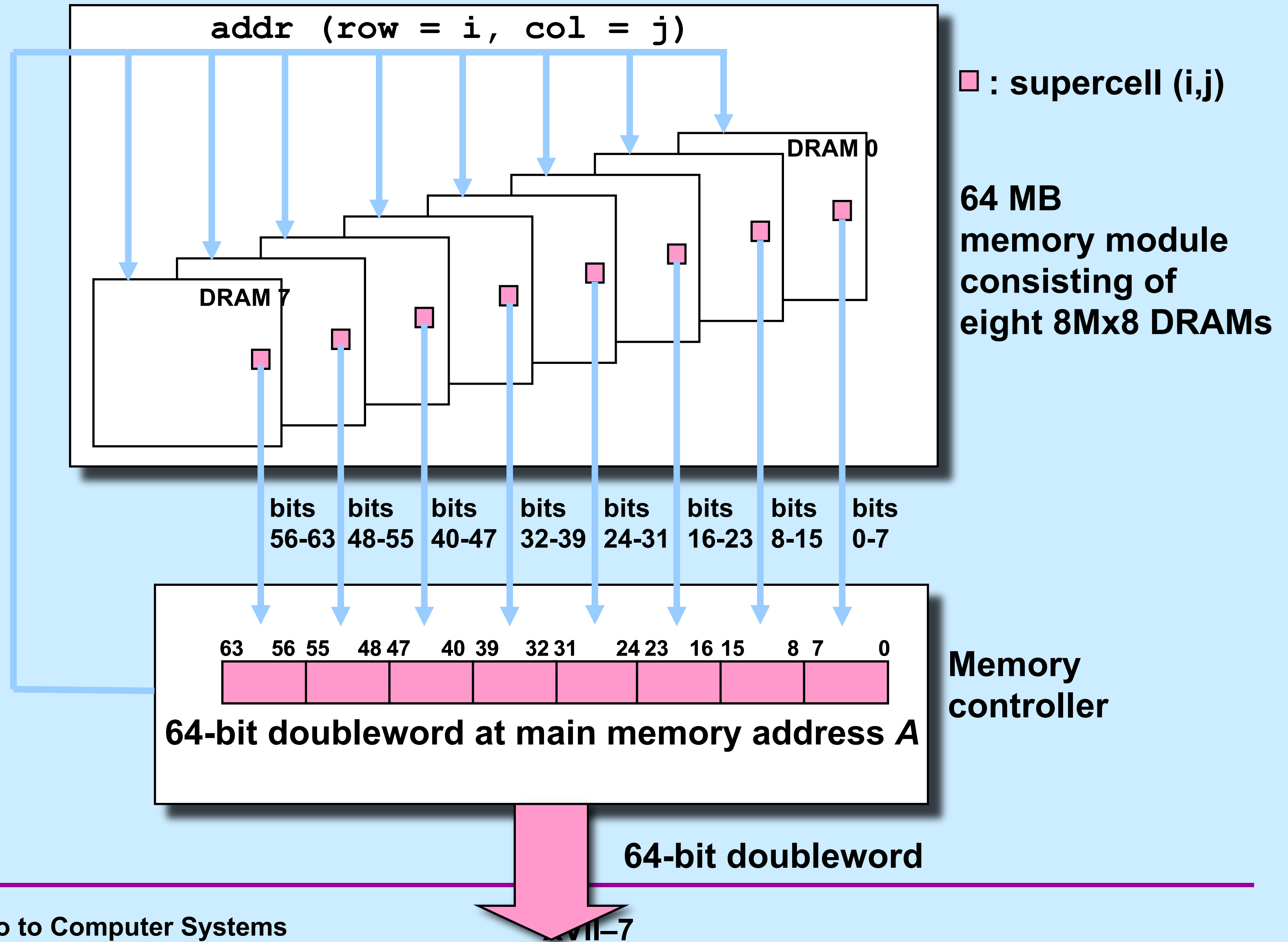
Reading DRAM Supercell (2,1)

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



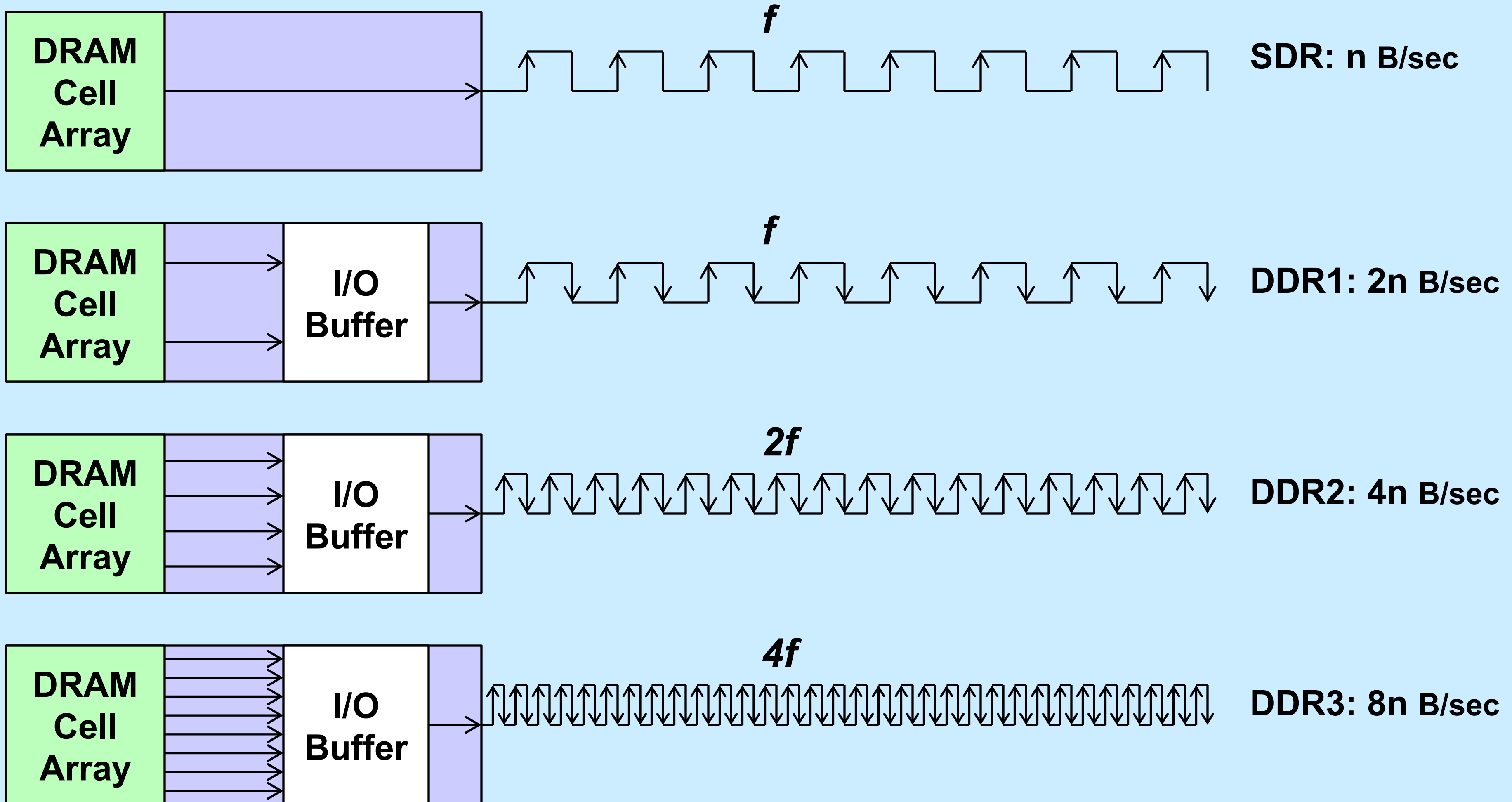
Memory Modules



Enhanced DRAMs

- **Basic DRAM cell has not changed since its invention in 1966**
  - commercialized by Intel in 1970
- **DRAMs with better interface logic and faster I/O:**
  - synchronous DRAM (SDRAM or SDR)
    - » uses a conventional clock signal instead of asynchronous control
    - » allows reuse of the row addresses (e.g., RAS, CAS, CAS, CAS)
  - double data-rate synchronous DRAM (DDR SDRAM)
    - » DDR1
      - twice as fast: 16 consecutive bytes xfr'd as fast as 8 in SDR
    - » DDR2
      - 4 times as fast: 32 consecutive bytes xfr'd as fast as 8 in SDR
    - » DDR3
      - 8 times as fast: 64 consecutive bytes xfr'd as fast as 8 in SDR

Enhanced DRAMs



Copyright © 2024 Thomas W. Doeppner. All rights reserved.

DDR4

- Memory transfer speed increased by a factor of 16 (twice as fast as DDR3)
  - no increase in DRAM Cell Array speed (same as SDR)
  - 16 times more data transferred at once
    - » 64 adjacent bytes fetched from DRAM
      - just like DDR3

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Quiz 2

A program is loading randomly selected bytes from memory. These bytes will be delivered to the processor on a DDR4 system at a speed that's n times that of an SDR system, where n is:

a) 8
b) 4
c) 2
d) 1

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

A Mismatch

- A processor clock cycle is ~0.3 nsecs
  - Older SunLab machines (Intel Core i5-4690) run at 3.5 GHz
- Basic operations take 1 – 10 clock cycles
  - .3 – 3 nsecs
- Accessing memory takes 70-100 nsecs
- How is this made to work?

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Caching to the Rescue

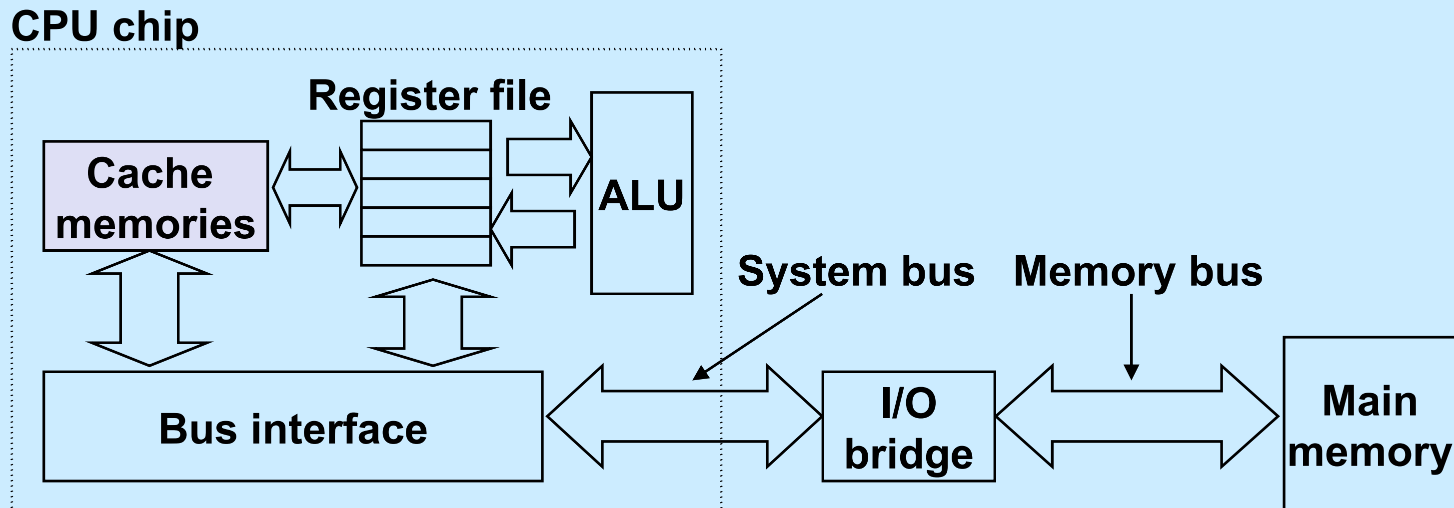
CPU

Cache

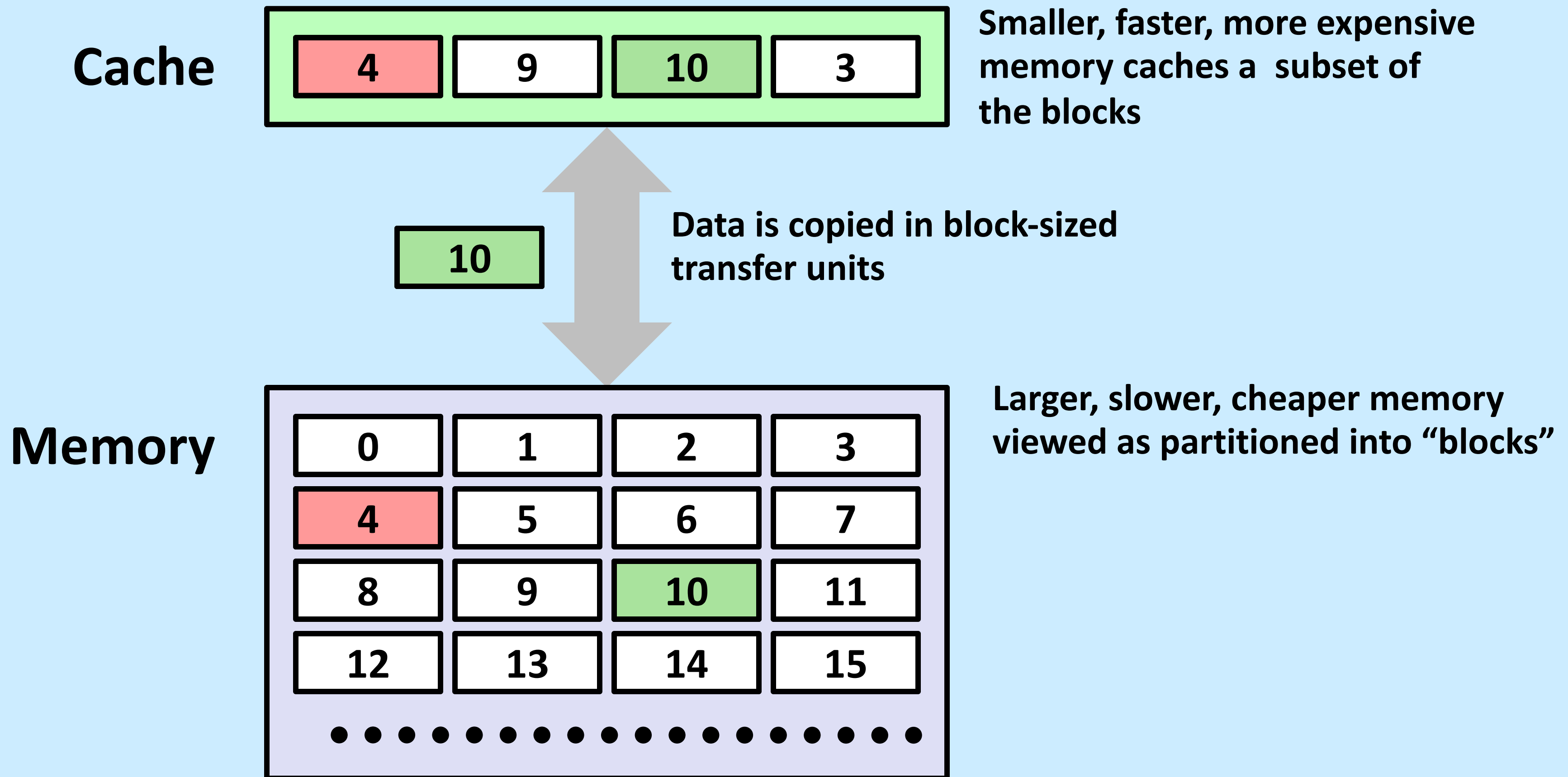
Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Cache Memories

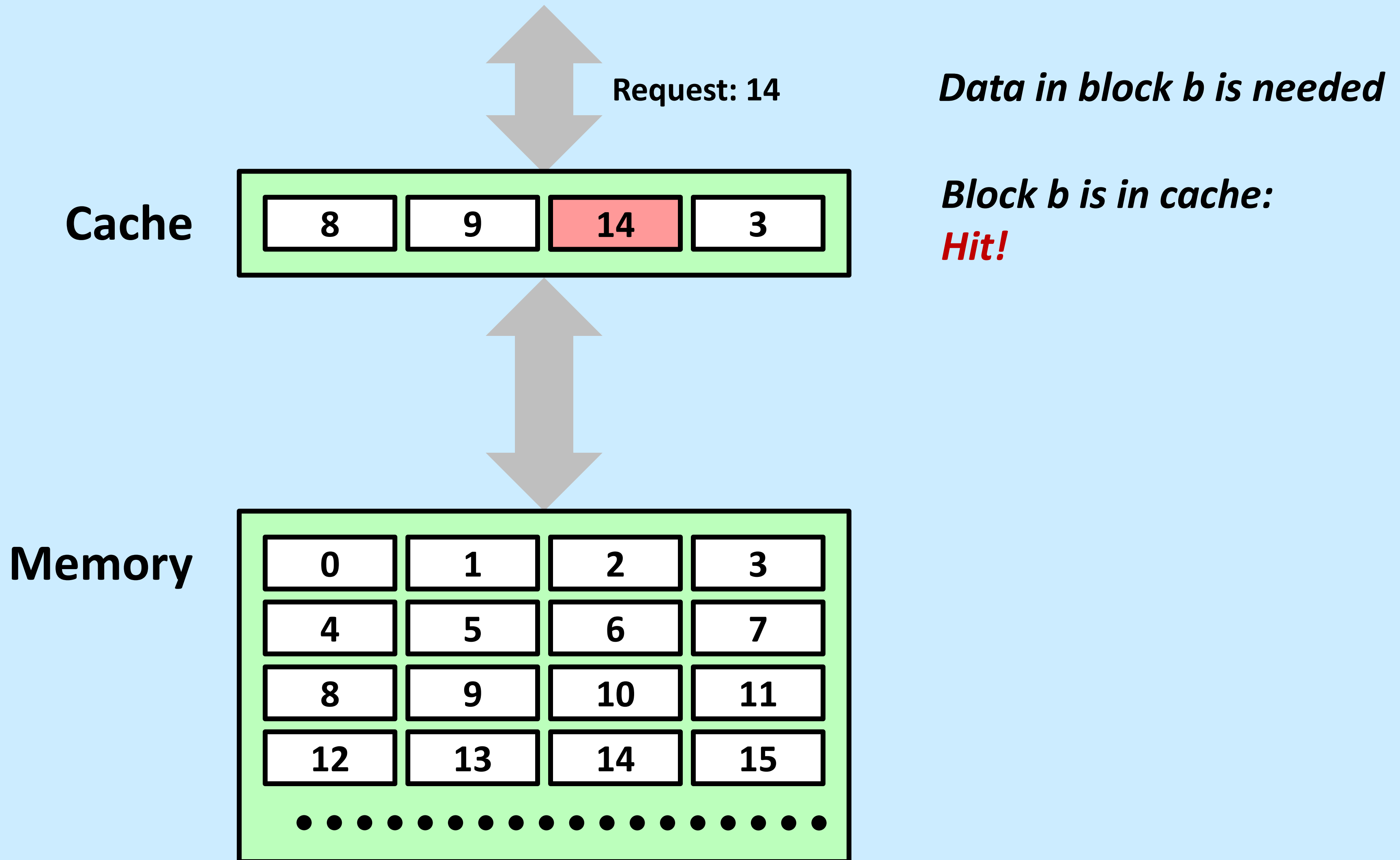
- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware
  - hold frequently accessed blocks of main memory
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory
- Typical system structure:



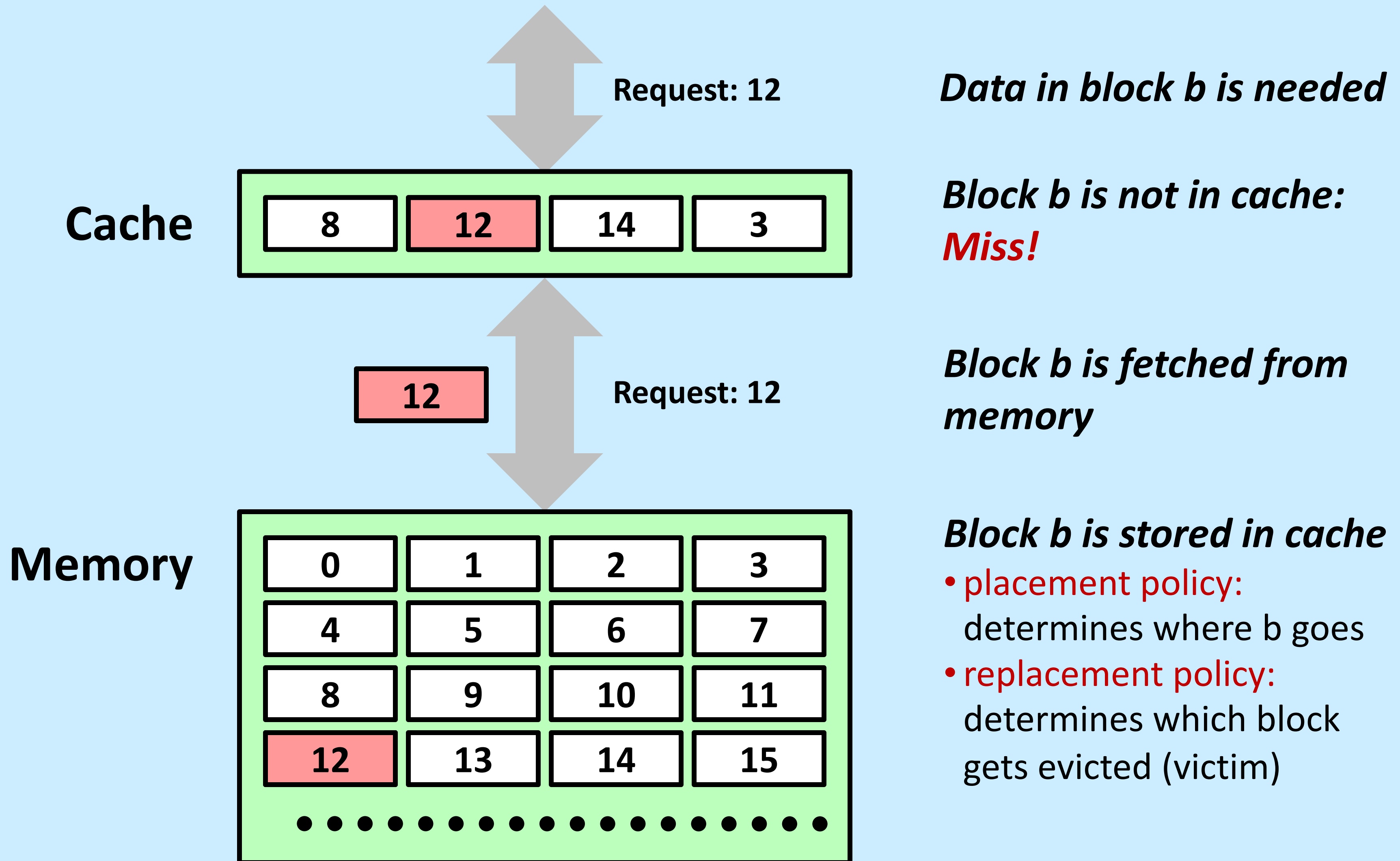
General Cache Concepts



General Cache Concepts: Hit



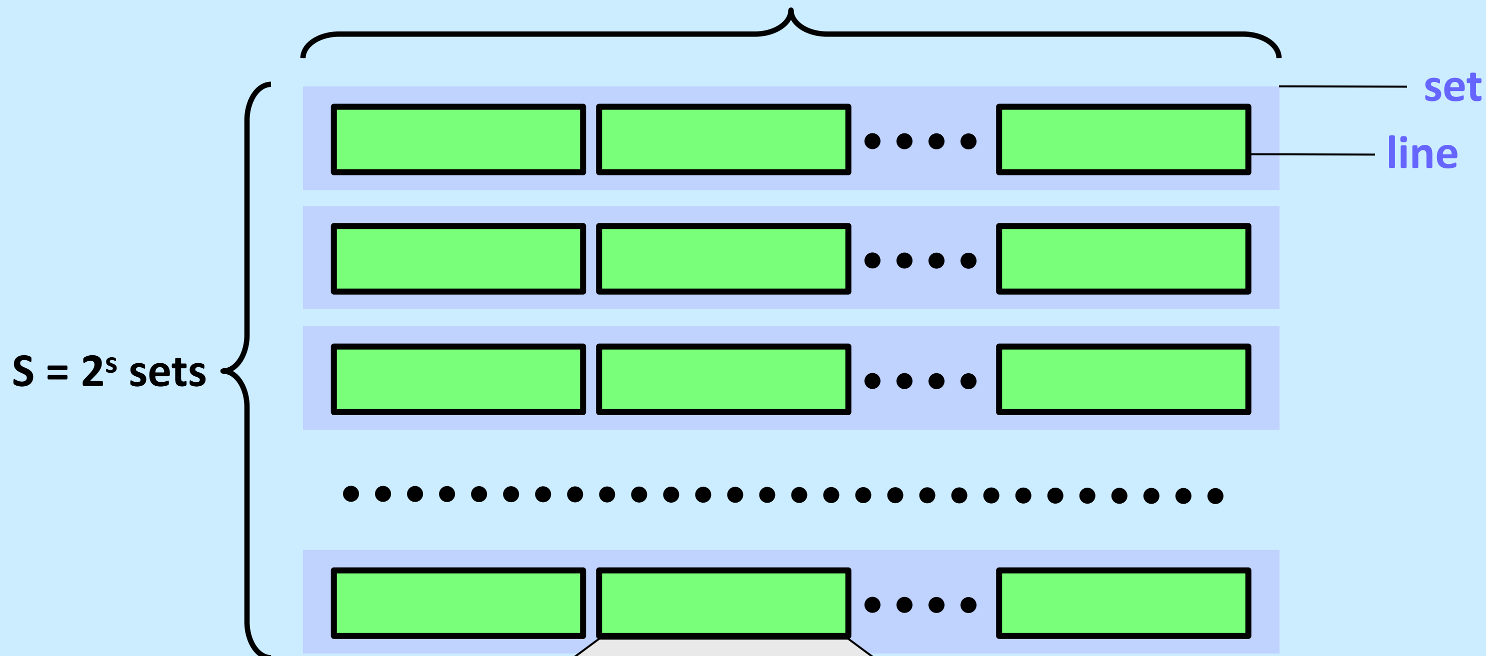
General Cache Concepts: Miss



General Caching Concepts: Types of Cache Misses

- **Cold (compulsory) miss**
  - cold misses occur because the cache is empty
- **Conflict miss**
  - most caches limit blocks to a small subset (sometimes a singleton) of the block positions in RAM
    - » e.g., block i in RAM must be placed in block (i mod 4) in the cache
  - conflict misses occur when the cache is large enough, but multiple data objects all map to the same cache 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

General Cache Organization (S, E, B)



v | tag | 0 | 1 | 2 | ······ | B-1

valid bit

B = 2^b bytes per cache block (the data)

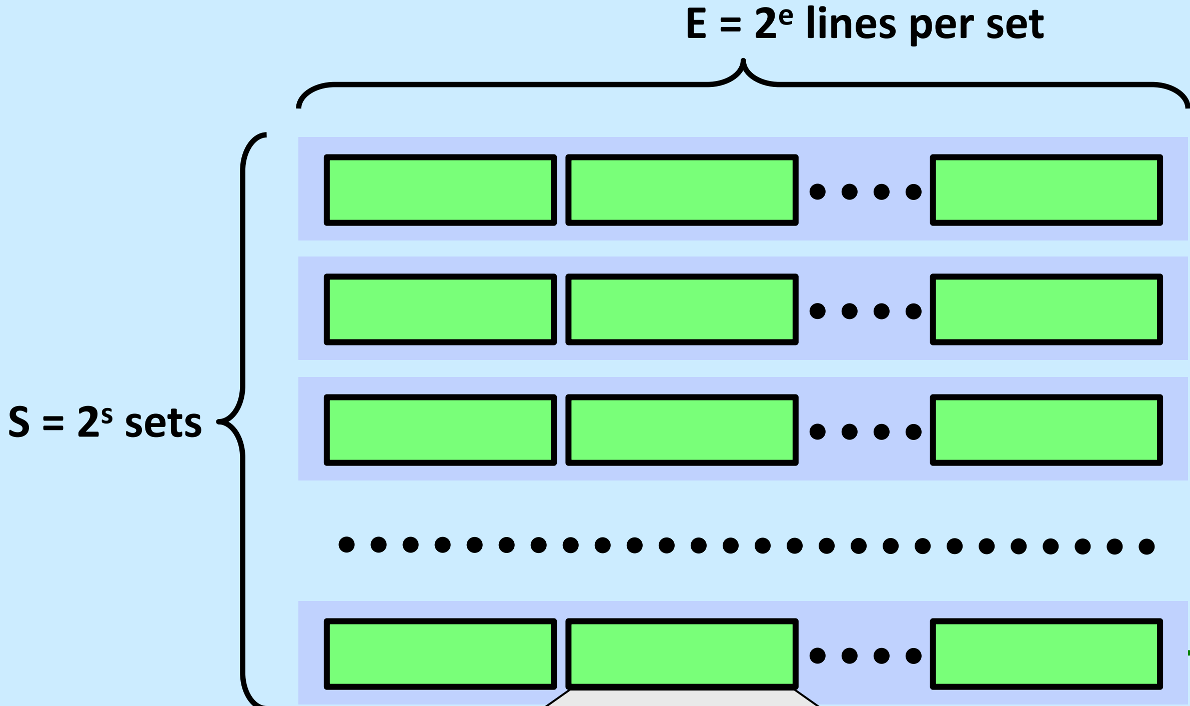
E = 2^e lines per set; S = 2^s sets

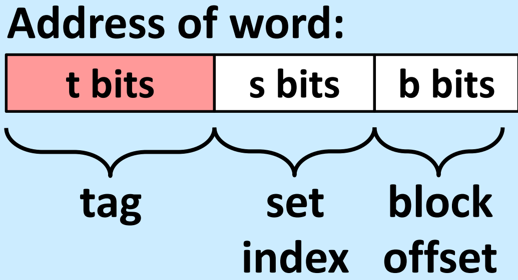
Cache size:

C = S x E x B data bytes

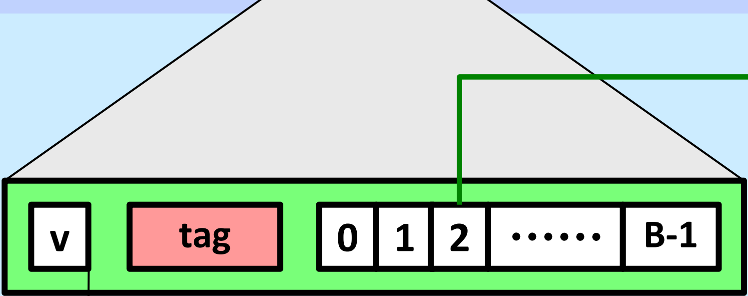
Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*





data begins at this offset



B = 2^b bytes per cache block (the data)

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 byte addresses, B=2 bytes/block,
S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

0	**[0000$_2$],**	miss
1	**[0001$_2$],**	hit
7	**[0111$_2$],**	miss
8	**[1000$_2$],**	miss
0	**[0000$_2$]**	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

A Higher-Level Example

Ignore the variables sum, i, j

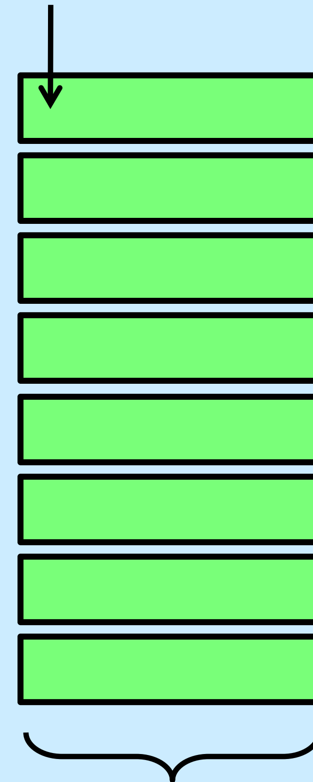
assume: cold (empty) cache,
a[0][0] goes here

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```



32 B = 4 doubles

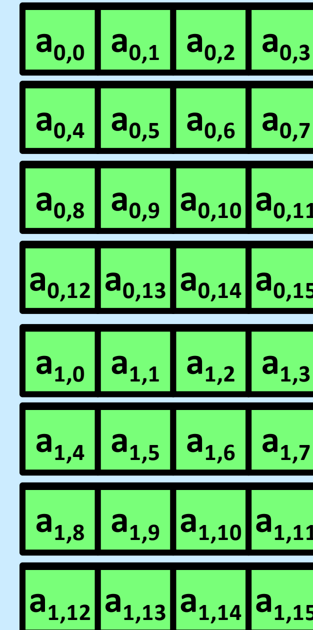
A Higher-Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```



32 B = 4 doubles

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

A Higher-Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; i++)
        for (i = 0; i < 16; j++)
            sum += a[i][j];
    return sum;
}
```

32 B = 4 doubles

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Conflict Misses: Aligned

```
double dotprod(double x[8], double y[8]) {
  double sum = 0.0;
  int i;

  for (i=0; i<8; i++)
    sum += x[i] * y[i];

  return sum;
}
```

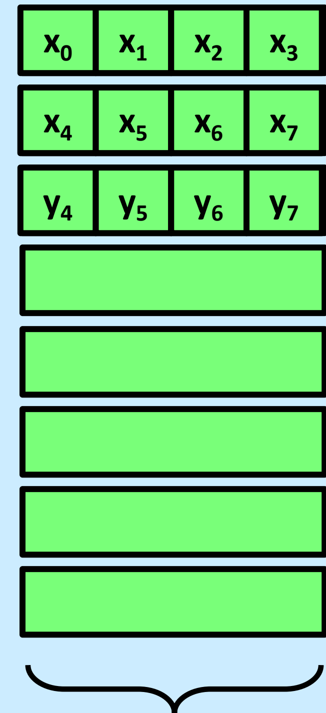
Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Different Alignments

```
double dotprod(double x[8], double y[8]) {
  double sum = 0.0;
  int i;

  for (i=0; i<8; i++)
    sum += x[i] * y[i];

  return sum;
}
```



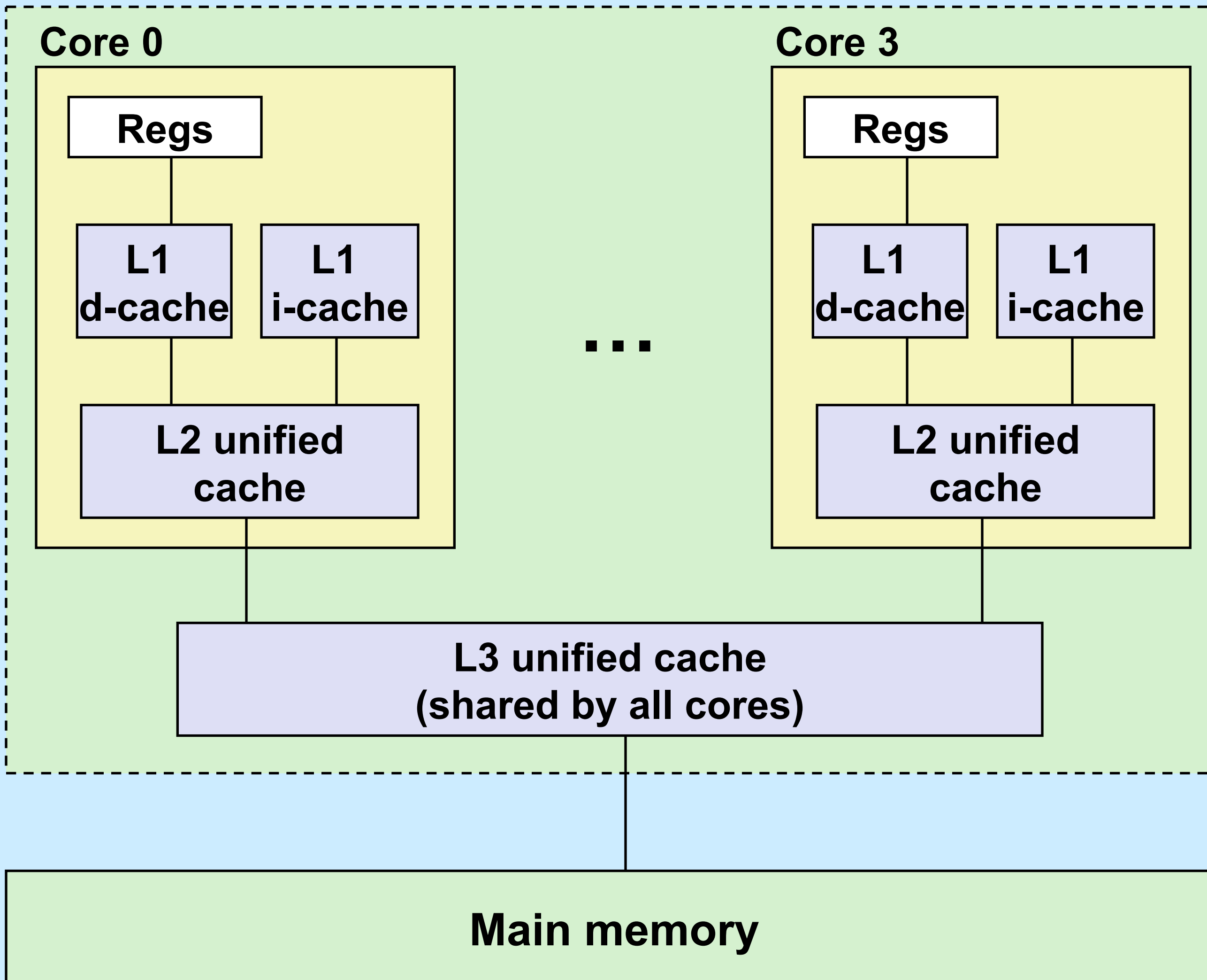
 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

CS 33

Exploiting Caches

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Intel Core i5 and i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches

What About Writes?

- Multiple copies of data exist:
  - L1, L2, main memory, disk
- What to do on a write-hit?
  - write-through (write immediately to memory)
  - write-back (defer write to memory until replacement of line)
    - » need a dirty bit (line different from memory or not)
- What to do on a write-miss?
  - write-allocate (load into cache, update line in cache)
    - » good if more writes to the location follow
  - no-write-allocate (writes immediately to memory)
- Typical
  - write-through + no-write-allocate
  - write-back + write-allocate

Accessing Memory

- **Program references memory (load)**
  - **if not in cache (*cache miss*), data is requested from RAM**
    - **fetched in units of 64 bytes**
      - **aligned to 64-byte boundaries (low-order 6 bits of address are zeroes)**
    - **if memory accessed sequentially, data is pre-fetched**
    - **data stored in cache (in 64-byte *cache lines*)**
      - **stays there until space must be re-used (least recently used is kicked out first)**
  - **if in cache (*cache hit*) no access to RAM needed**
- **Program modifies memory (store)**
  - **data modified in cache**
  - **eventually written to RAM in 64-byte units**

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Cache Performance Metrics

- **Miss rate**
  - fraction of memory references not found in cache (misses / accesses)
    = 1 – hit rate
  - typical numbers (in percentages):
    - » 3-10% for L1
    - » can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **Hit time**
  - time to deliver a line in the cache to the processor
    - » includes time to determine whether the line is in the cache
  - typical numbers:
    - » 1-2 clock cycles for L1
    - » 5-20 clock cycles for L2
- **Miss penalty**
  - additional time required because of a miss
    - » typically 50-200 cycles for main memory (trend: increasing!)

Hits vs. Misses

- Huge difference between hit and miss times
  - could be 100x, if just L1 and main memory
- 99% hit rate is twice as good as 97%!
  - consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - average access time:

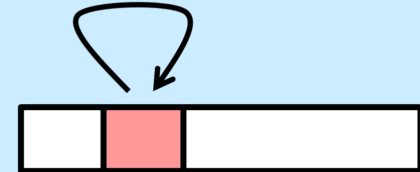
    97% hits: .97 * 1 cycle + 0.03 * 100 cycles ≈ 4 cycles

    99% hits: .99 * 1 cycle + 0.01 * 100 cycles ≈ 2 cycles

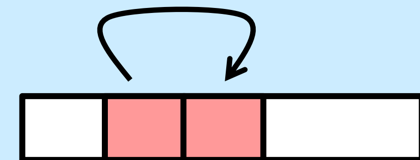
- This is why “miss rate” is used instead of “hit rate”

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**
  - reference array elements in succession (stride-1 reference pattern) — Spatial locality
  - reference variable `sum` each iteration — Temporal locality
- **Instruction references**
  - reference instructions in sequence. — Spatial locality
  - cycle through loop repeatedly — Temporal locality

Quiz 2

Does this function have good locality with respect to array `a`? The array a is MxN.

a) yes

b) no

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

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

Writing Cache-Friendly Code

- **Make the common case fast**
  - focus on the inner loops of the core functions
- **Minimize the misses in the inner loops**
  - repeated references to variables are good (temporal locality)
  - stride-1 reference patterns are good (spatial locality)

Matrix Multiplication Example

- **Description:**
  - multiply N x N matrices
    - » each element is a double
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination
    - » but may be able to hold in register

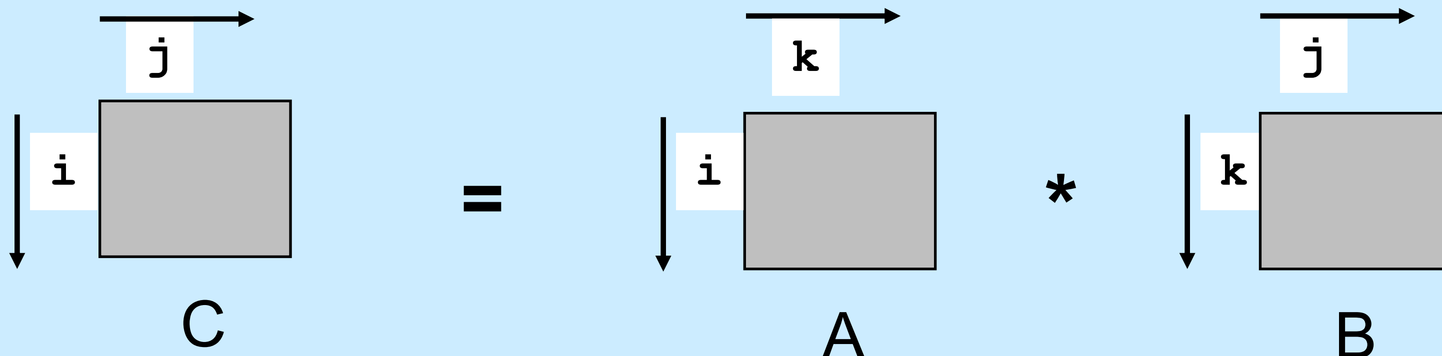
Variable `sum` *held in register*

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

Miss-Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 64B (big enough for eight doubles)
  - matrix dimension (N) is very large
  - cache is not big enough to hold multiple rows
- **Analysis method:**
  - look at access pattern of inner loop

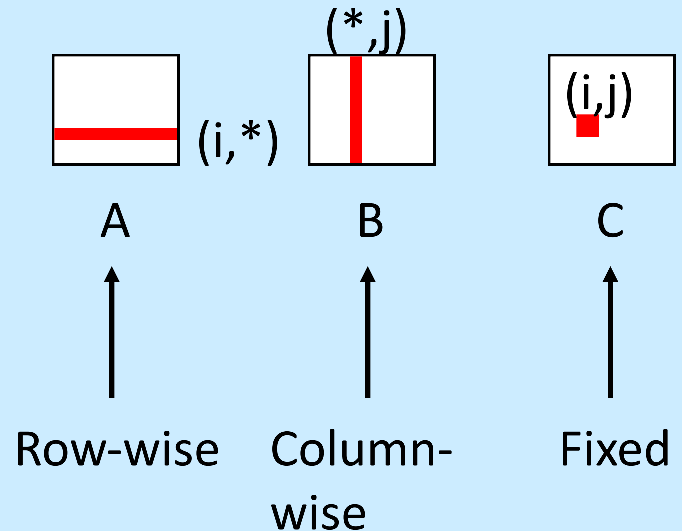


Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - ```
    for (i = 0; i < N; i++)
      sum += a[0][i];
    ```
  - accesses successive elements
  - if block size (B) > 8 bytes, exploit spatial locality
    - » compulsory miss rate = 8 bytes / Block
- Stepping through rows in one column:
  - ```
    for (i = 0; i < n; i++)
      sum += a[i][0];
    ```
  - accesses widely separated elements
  - no spatial locality!
    - » compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

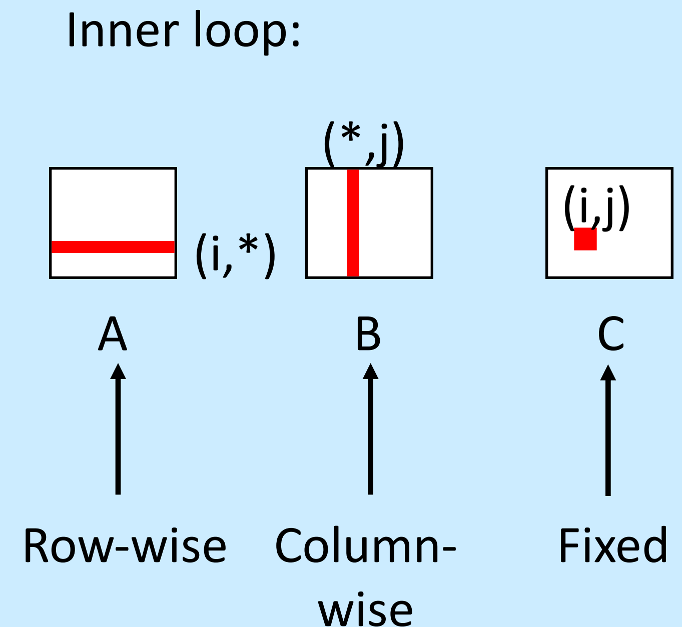


Misses per inner loop iteration:

A	B	C
0.125	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

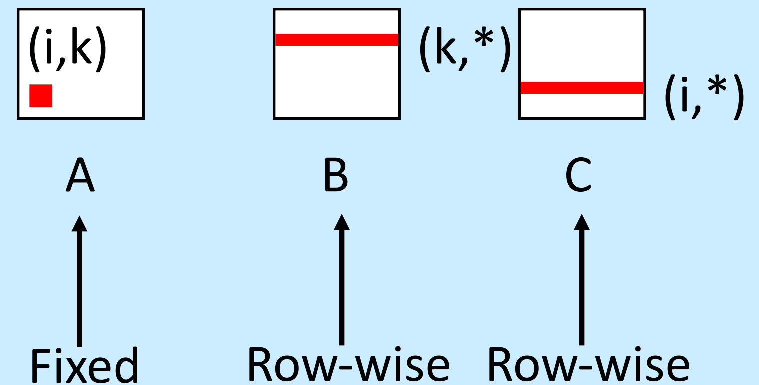


Misses per inner loop iteration:

A	B	C
0.125	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

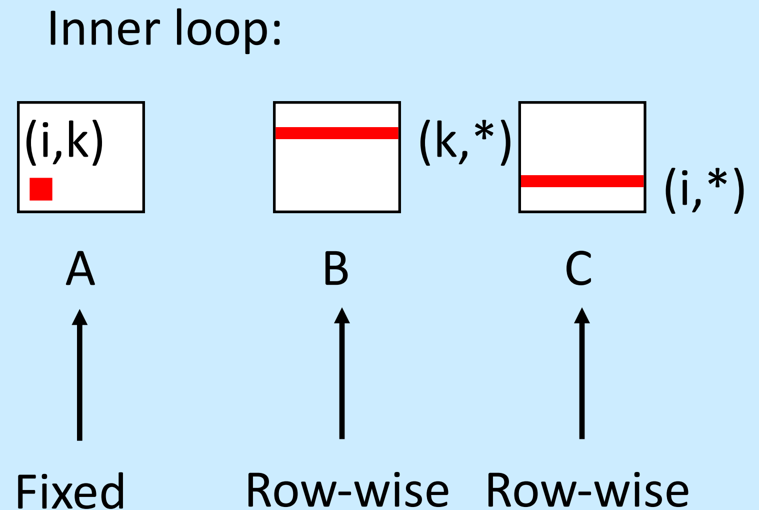


Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

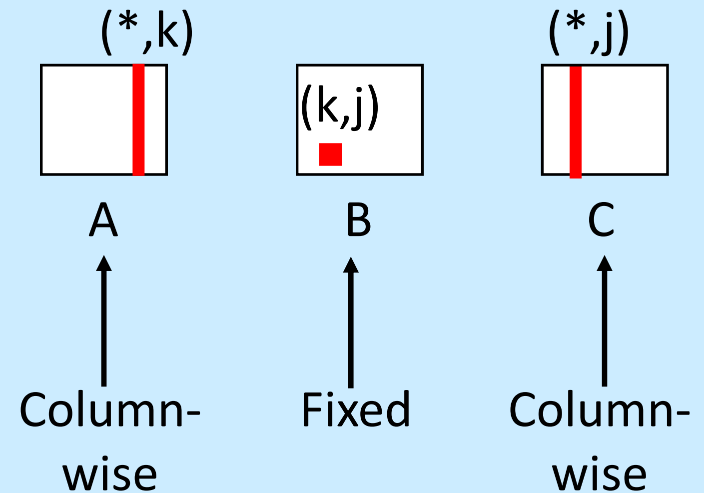


Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

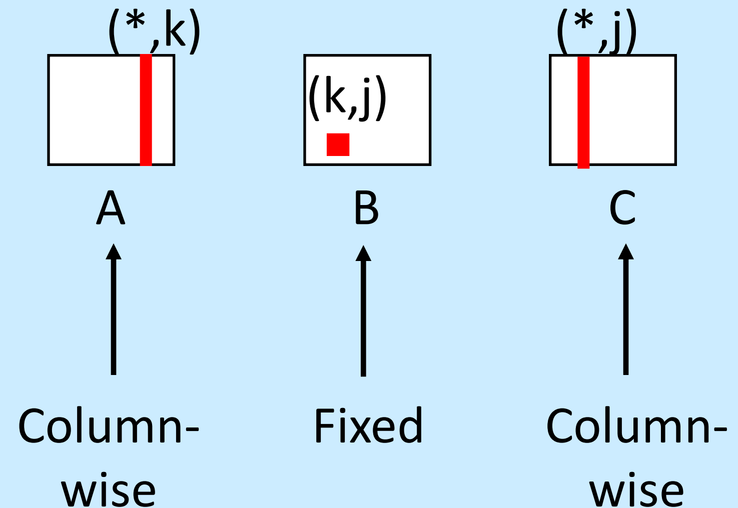


Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
  }
```

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.125**

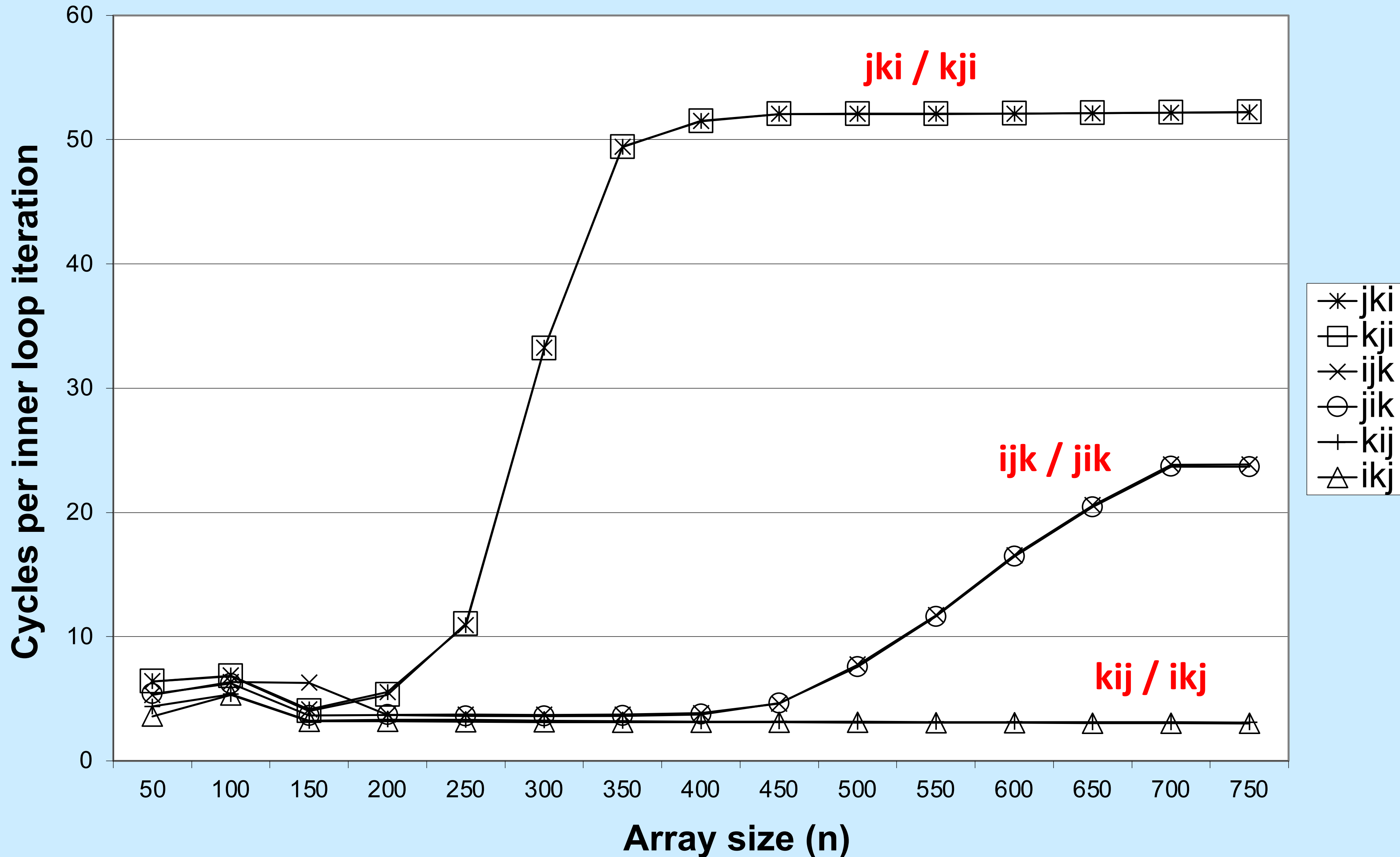
```
for (k=0; k<n; k++)
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
  }
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.25**

```
for (j=0; j<n; j++)
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

Core i7 Matrix Multiply Performance



In Real Life ...

- **Multiply two 1024x1024 matrices of doubles on core i5 machines (formerly in the Sunlab)**
  - **ijk**
    - » **4.185 seconds**
  - **kij**
    - » **0.798 seconds**
  - **jki**
    - » **11.488 seconds**

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Concluding Observations

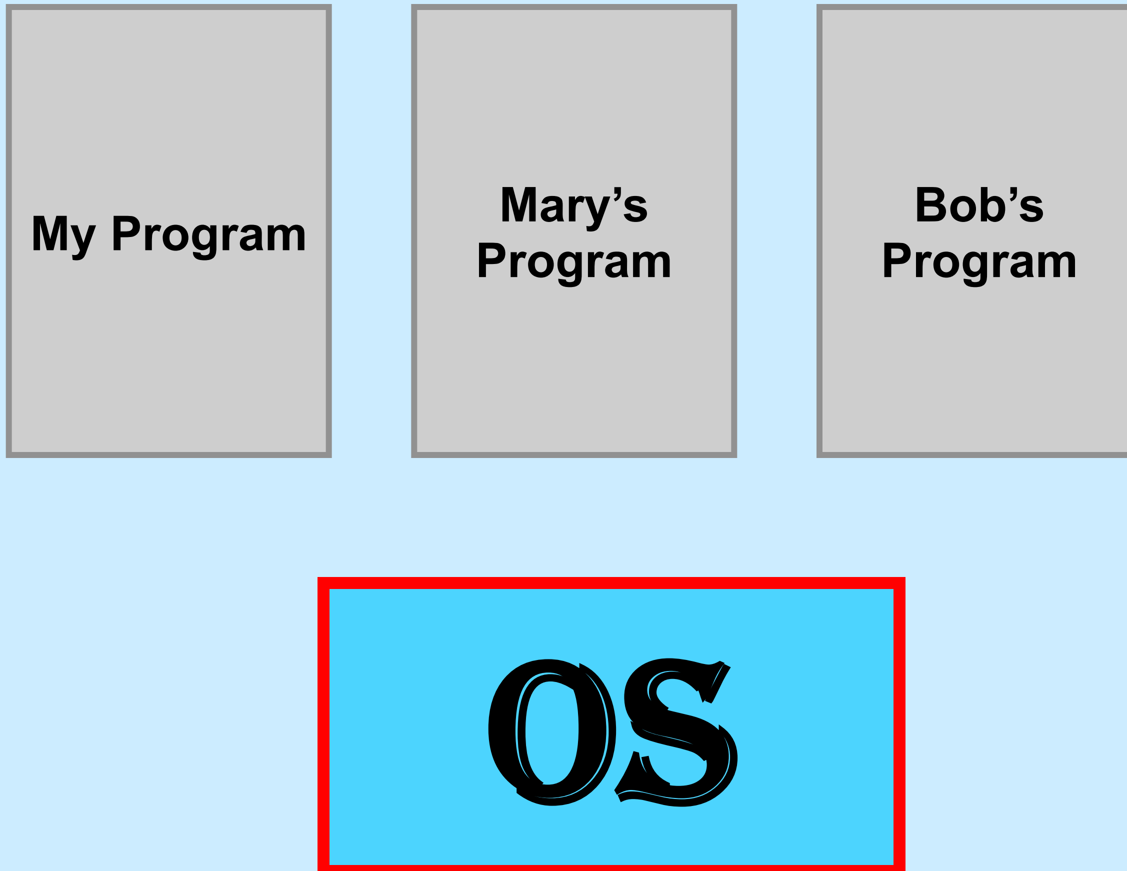
- Programmer can optimize for cache performance
  - organize data structures appropriately
- All systems favor "cache-friendly code"
  - getting absolute optimum performance is very platform specific
    - » cache sizes, line sizes, associativities, etc.
  - can get most of the advantage with generic code
    - » keep working set reasonably small (temporal locality)
    - » use small strides (spatial locality)

CS 33

Architecture and the OS

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

The Operating System



Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Processes

- Containers for programs
  - virtual memory
    - address space
  - scheduling
    - one or more threads of control
  - file references
    - open files
  - and lots more!

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Idiot Proof …

```
int main( ) {
  int i;
  int A[1];

  for (i=0; ; i++)
    A[rand()] = i;
}
```

Can I clobber Mary's program?

Mary's Program

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Fair Share

```
void runforever( ){
  while(1)
    ;
}

int main( ) {
  runforever();
}
```

Can I prevent Bob's program from running?

Bob's Program

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Architectural Support for the OS

- Not all instructions are created equal ...
  - non-privileged instructions
    - » can affect only current program
  - privileged instructions
    - » may affect entire system
- Processor mode
  - user mode
    - » can execute only non-privileged instructions
  - privileged mode
    - » can execute all instructions

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Which Instructions Should Be Privileged?

- I/O instructions
- Those that affect how memory is mapped
- Halt instruction
- Some others ...

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Who Is Privileged?

- No one
  - user code always runs in user mode
- The operating-system kernel runs in privileged mode
  - nothing else does
  - not even super user on Unix or administrator on Windows

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Entering Privileged Mode

- How is OS invoked?
  - very carefully ...
  - strictly in response to interrupts and exceptions
  - (booting is a special case)

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Interrupts and Exceptions

- Things don’t always go smoothly ...
  - I/O devices demand attention
  - timers expire
  - programs demand OS services
  - programs demand storage be made accessible
  - programs have problems
- Interrupts
  - demand for attention from external sources
- Exceptions
  - executing program requires attention

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Exceptions

- **Traps**
  - **"intentional" exceptions**
    - **execution of special instruction to invoke OS**
  - **after servicing, execution resumes with next instruction**
- **Faults**
  - **a problem condition that is normally corrected**
  - **after servicing, instruction is re-tried**
- **Aborts**
  - **something went dreadfully wrong ...**
  - **not possible to re-try instruction, nor to go on to next instruction**

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

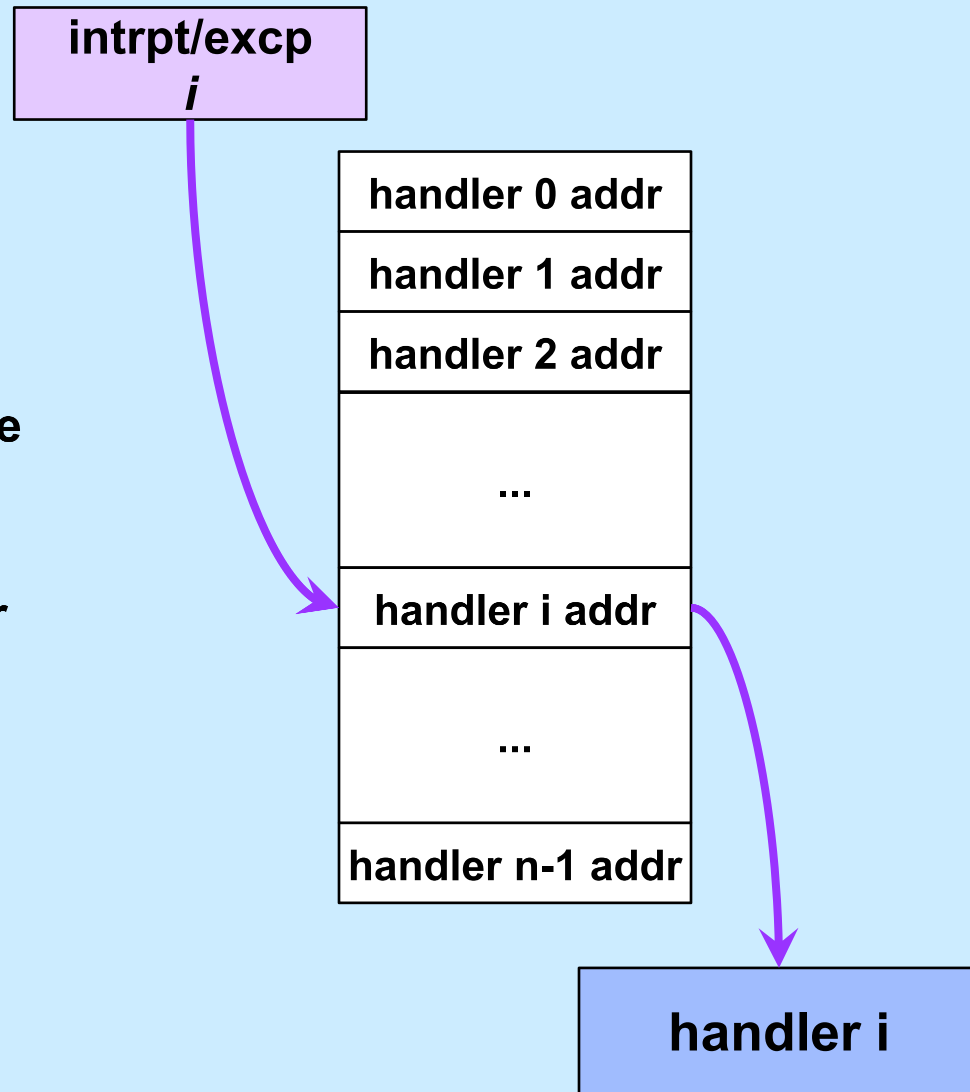
Actions for Interrupts and Exceptions

- When interrupt or exception occurs
  - processor saves state of current thread/process on stack
  - processor switches to privileged mode (if not already there)
  - invokes handler for interrupt/exception
  - if thread/process is to be resumed (typical action after interrupt)
    - » thread/process state is restored from stack
  - if thread/process is to re-execute current instruction
    - » thread/process state is restored, after backing up instruction pointer
  - if thread/process is to terminate
    - » it's terminated

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Interrupt and Exception Handlers

- Interrupt or exception invokes handler (in OS)
  - via interrupt and exception vector
    - » one entry for each possible interrupt/exception
      - contains
        - address of handler
  - code executed in privileged mode
    - » but code is part of the OS



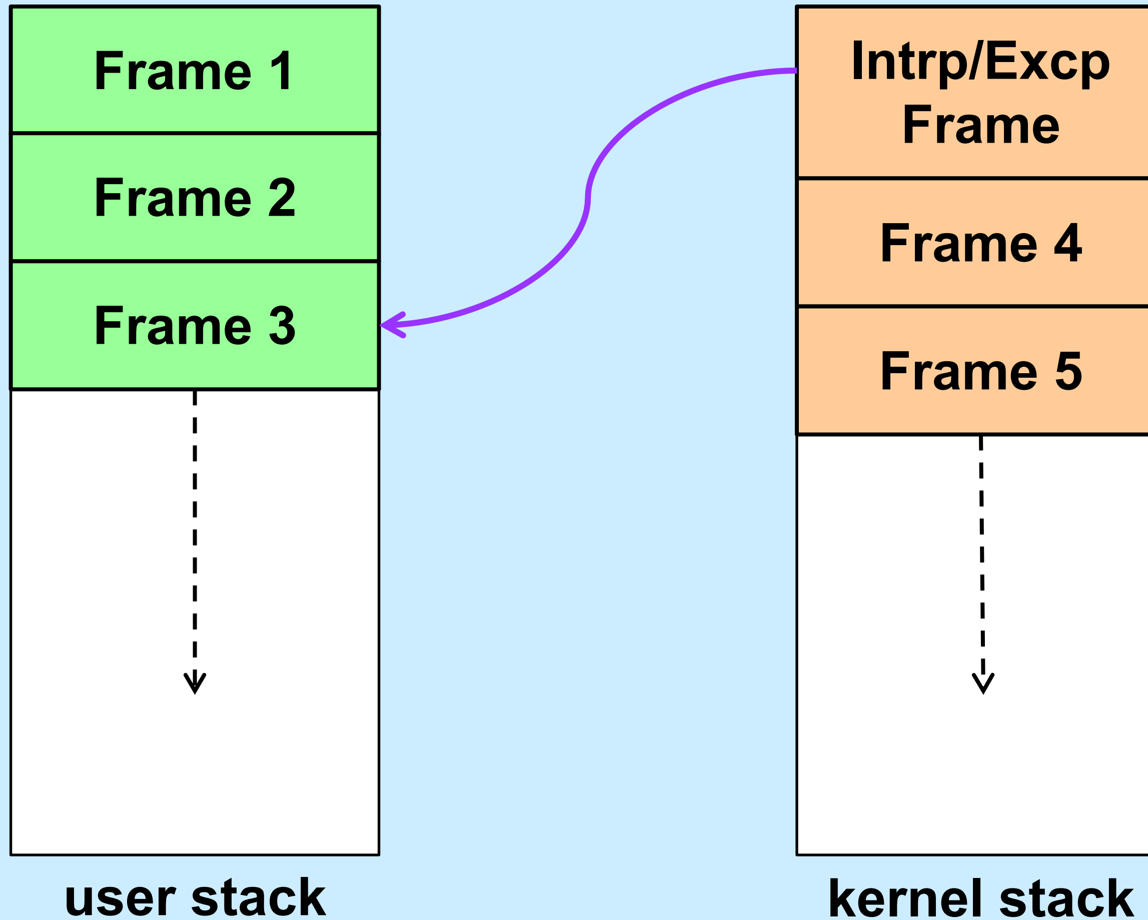
 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Entering and Exiting

- Entering/exiting interrupt/exception handler more involved than entering/exiting a function
  - must deal with processor mode
    - switch to privileged mode on entry
    - switch back to previous mode on exit
  - interrupted process/thread’s state is saved on separate kernel stack
  - stack in kernel must be different from stack in user program
    - why?

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

One Stack Per Mode



 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Quiz 3

If an interrupt occurs, which general-purpose registers must be pushed onto the kernel stack?

a) all
b) none
c) callee-save registers
d) caller-save registers

Copyright © 2024 Thomas W. Doeppner. All rights reserved.