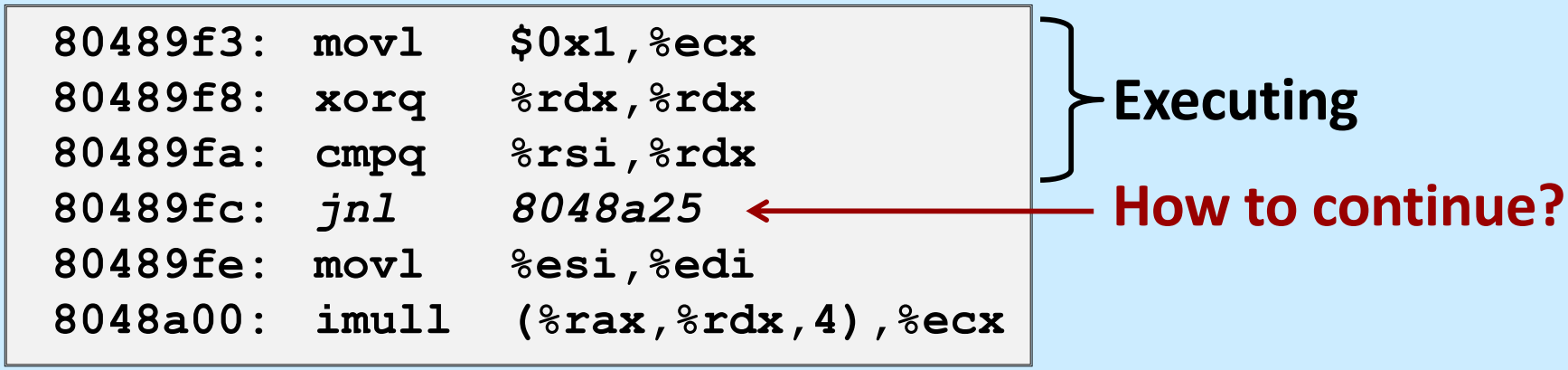
CS 33

Architecture and Optimization (3)

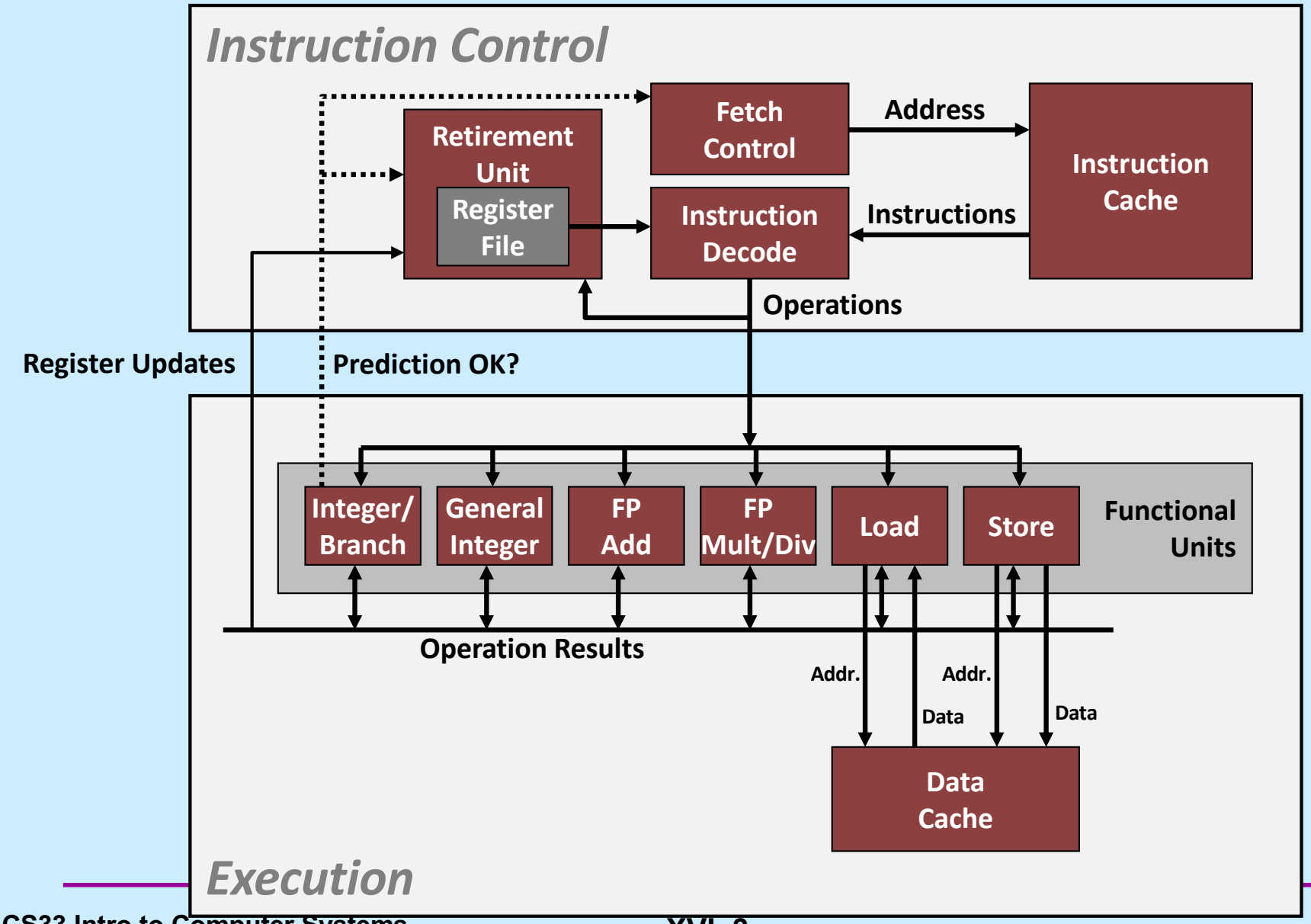
What About Branches?

- **Challenge**
  - **instruction control unit must work well ahead of execution unit to generate enough operations to keep EU busy**



  - **when it encounters conditional branch, cannot reliably determine where to continue fetching**

Modern CPU Design



Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
  - branch taken: transfer control to branch target
  - branch not-taken: continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit

```
80489f3:  movl   $0x1,%ecx
80489f8:  xorq   %rdx,%rdx
80489fa:  cmpq   %rsi,%rdx
80489fc:  jnl    8048a25
80489fe:  movl   %esi,%esi
8048a00:  imull  (%rax,%rdx,4),%ecx
```

Branch not-taken

Branch taken

```
8048a25:  cmpq   %rdi,%rdx
8048a27:  jl     8048a20
8048a29:  movl   0xc(%rbp),%eax
8048a2c:  leal   0xfffffff8(%rbp),%esp
8048a2f:  movl   %ecx,(%rax)
```

Branch Prediction

- Idea
  - guess which way branch will go
  - begin executing instructions at predicted position
    - » but don't actually modify register or memory data

```
80489f3:  movl   $0x1,%ecx
80489f8:  xorq   %edx,%edx
80489fa:  cmpq   %rsi,%rdx
80489fc:  jnl    8048a25
. . .
```

Predict taken

```
8048a25:  cmpq   %rdi,%rdx
8048a27:  jl     8048a20
8048a29:  movl   0xc(%rbp),%eax
8048a2c:  leal   0xffffffe8(%rbp),%esp
8048a2f:  movl   %ecx,(%rax)
```

Begin execution

Branch Prediction Through Loop

```
80488b1:   movl   (%rcx,%rdx,4),%eax
80488b4:   addl   %eax,(%rdi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 98

Predict taken (OK)

```
80488b1:   movl   (%rcx,%rdx,4),%eax
80488b4:   addl   %eax,(%rdi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 99

Predict taken (oops)

```
80488b1:   movl   (%rcx,%rdx,4),%eax
80488b4:   addl   %eax,(%rdi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

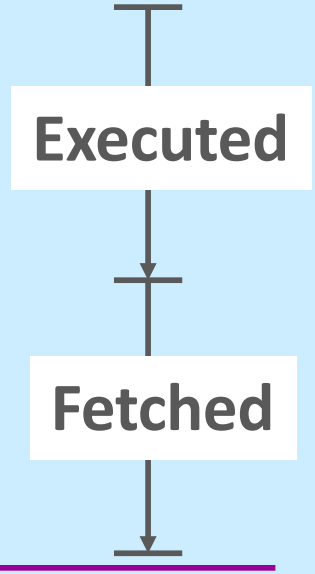
i = 100

Read invalid location

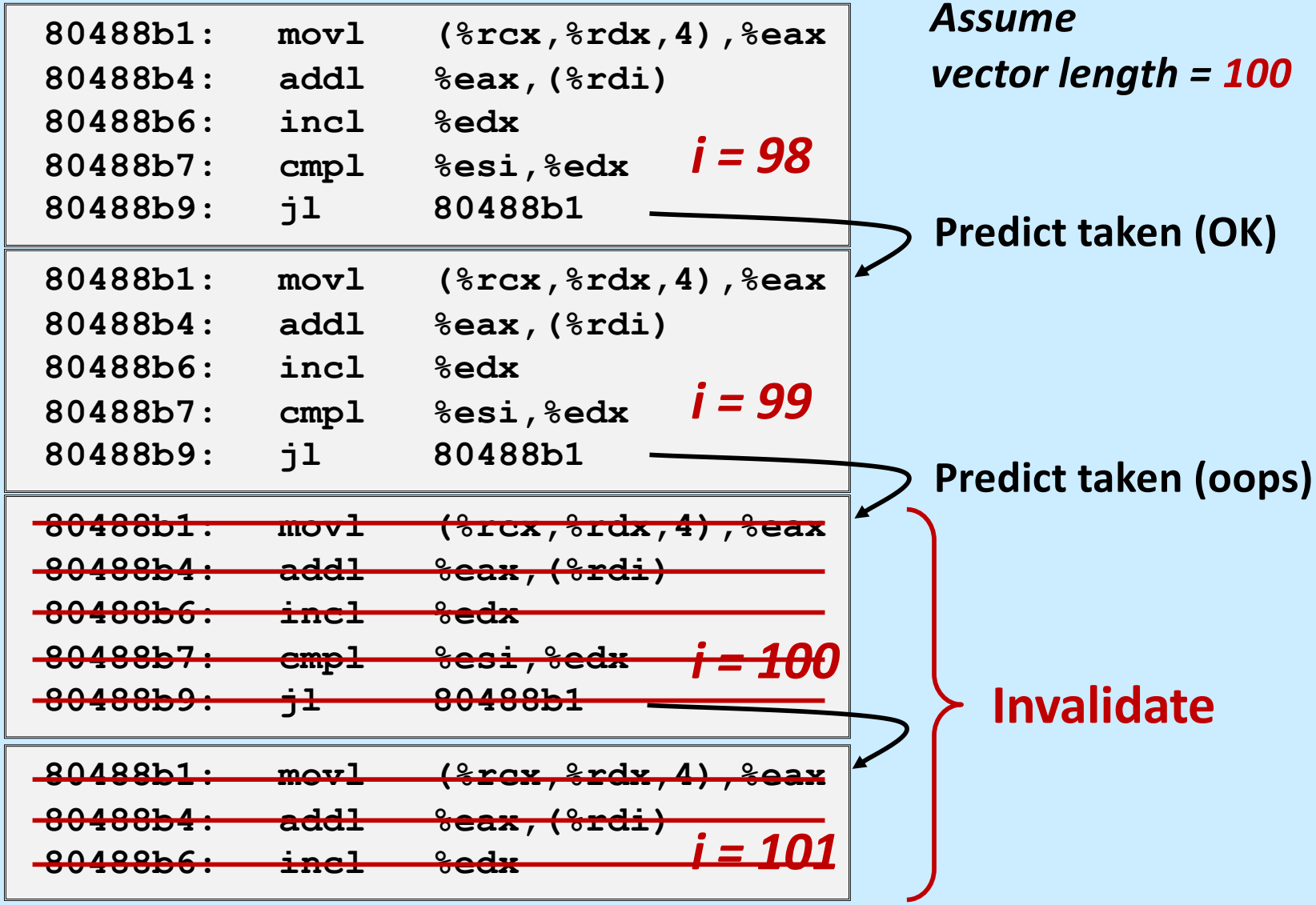
```
80488b1:   movl   (%rcx,%rdx,4),%eax
80488b4:   addl   %eax,(%rdi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 101

Assume vector length = 100



Branch Misprediction Invalidation



Branch Misprediction Recovery

```
80488b1:   movl   (%rcx,%rdx,4),%eax
80488b4:   addl   %eax,(%rdi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
80488bb:   leal   0xfffffffe8(%rbp),%esp
80488be:   popl   %ebx
80488bf:   popl   %esi
80488c0:   popl   %edi
```

i = 99

Definitely not taken

- **Performance Cost**
  - multiple clock cycles on modern processor
  - can be a major performance limiter

Latency of Loads

```
typedef struct ELE {
  struct ELE *next;
  long data;
} list_ele, *list_ptr;

int list_len(list_ptr ls) {
  long len = 0;
  while (ls) {
    len++;
    ls = ls->next;
  }
  return len;
}
```

```
# len in %rax, ls in %rdi

.L11:                       # loop:
  addq   $1, %rax           # incr len
  movq   (%rdi), %rdi       # ls = ls->next
  testq  %rdi, %rdi         # test ls
  jne    .L11               # if != 0
                            #   go to loop
```

- **4 CPE**

Clearing an Array ...

```
#define ITERS 100000000
void clear_array() {
  long dest[100];
  int iter;
  for (iter=0; iter<ITERS; iter++) {
    long i;
    for (i=0; i<100; i++)
      dest[i] = 0;
  }
}
```

- **1 CPE**

Store/Load Interaction

```
void write_read(long *src, long *dest, long n) {
  long cnt = n;
  long val = 0;

  while(cnt--) {
    *dest = val;
    val = (*src)+1;
  }
}
```

Store/Load Interaction

```
void write_read(long *src,
     long *dest, long n){
  long cnt = n;
  long val = 0;
  while(cnt--) {
    *dest = val;
    val = (*src)+1;
  }
}
```

```
long a[] = {-10, 17};
```

Example A: `write_read(&a[0],&a[1],3)`

	Initial	Iter. 1	Iter. 2	Iter. 3
cnt	3	2	1	0
a	–10 | 17	–10 | 0	–10 | –9	–10 | –9
val	0	–9	–9	–9

- **CPE 1.3**

Example B: `write_read(&a[0],&a[0],3)`

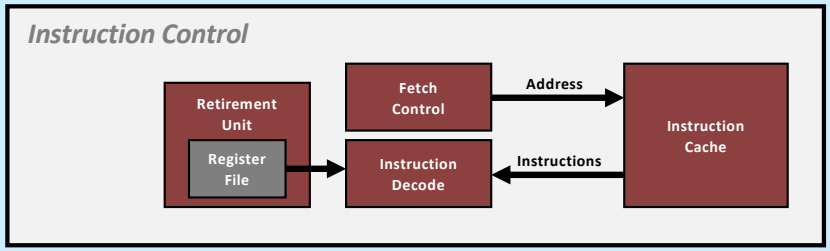
	Initial	Iter. 1	Iter. 2	Iter. 3
cnt	3	2	1	0
a	–10 | 17	0 | 17	1 | 17	2 | 17
val	0	1	2	3

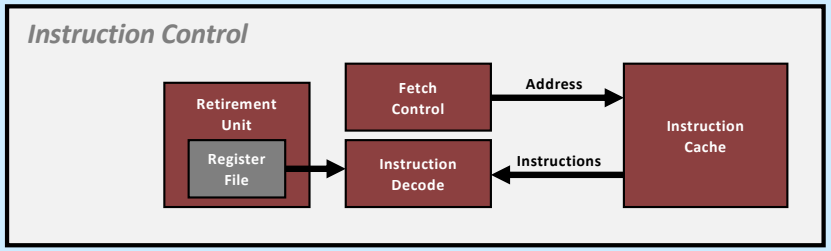
- **CPE 7.3**

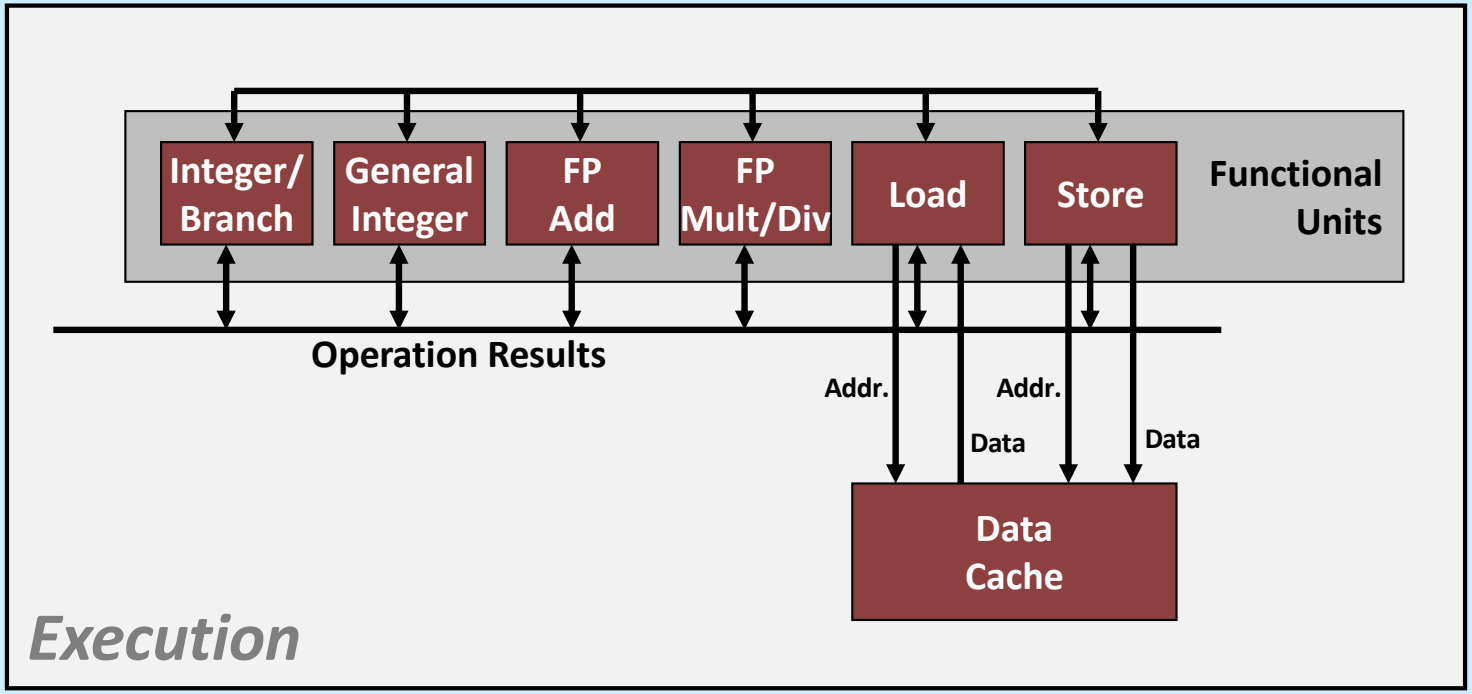
Getting High Performance

- Good compiler and flags
- Don't do anything stupid
  - watch out for hidden algorithmic inefficiencies
  - write compiler-friendly code
    - » watch out for optimization blockers: procedure calls & memory references
  - look carefully at innermost loops (where most work is done)
- Tune code for machine
  - exploit instruction-level parallelism
  - avoid unpredictable branches
  - make code cache friendly (covered soon)

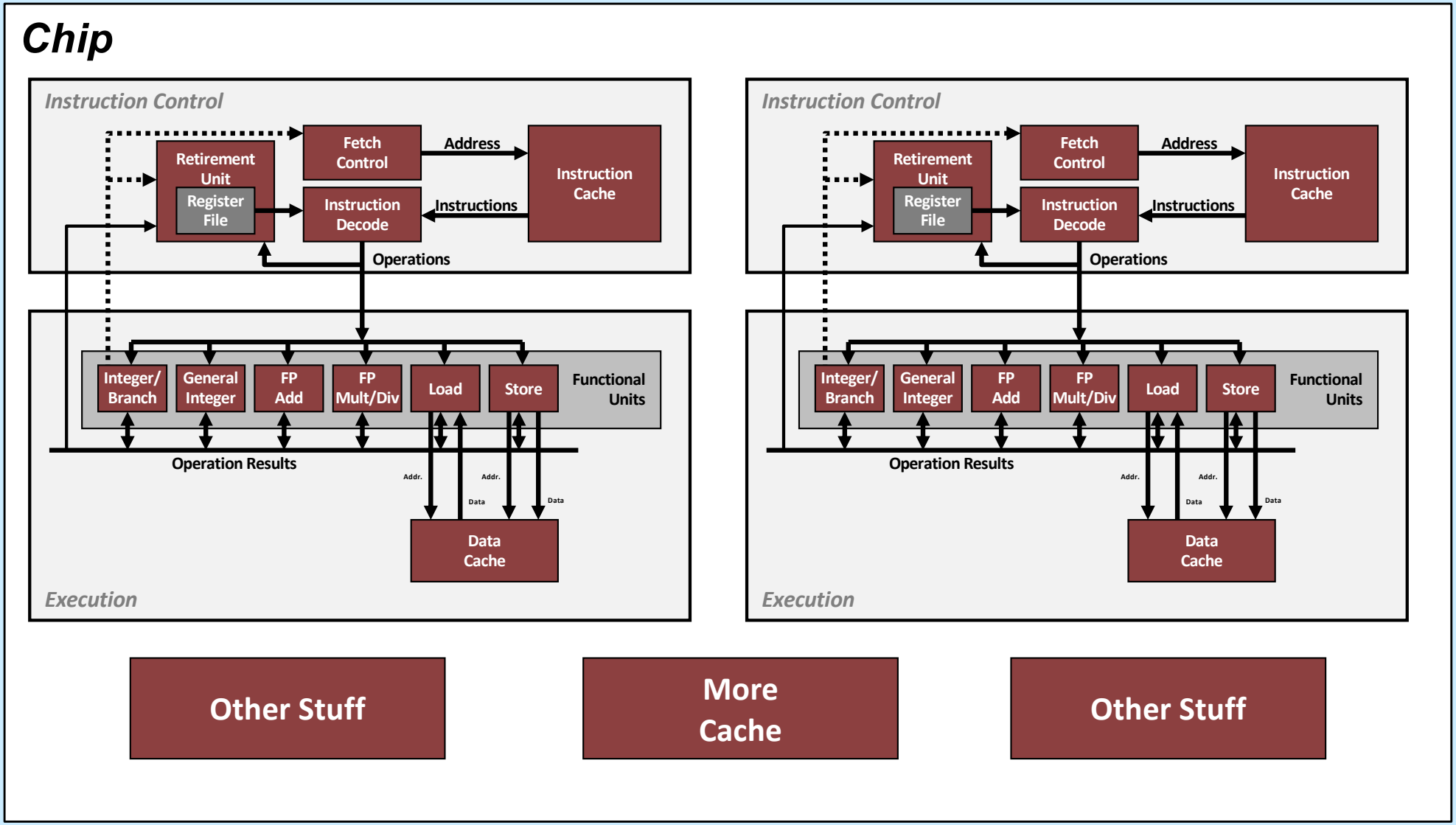
Hyper Threading







Multiple Cores



CS 33

Memory Hierarchy I

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

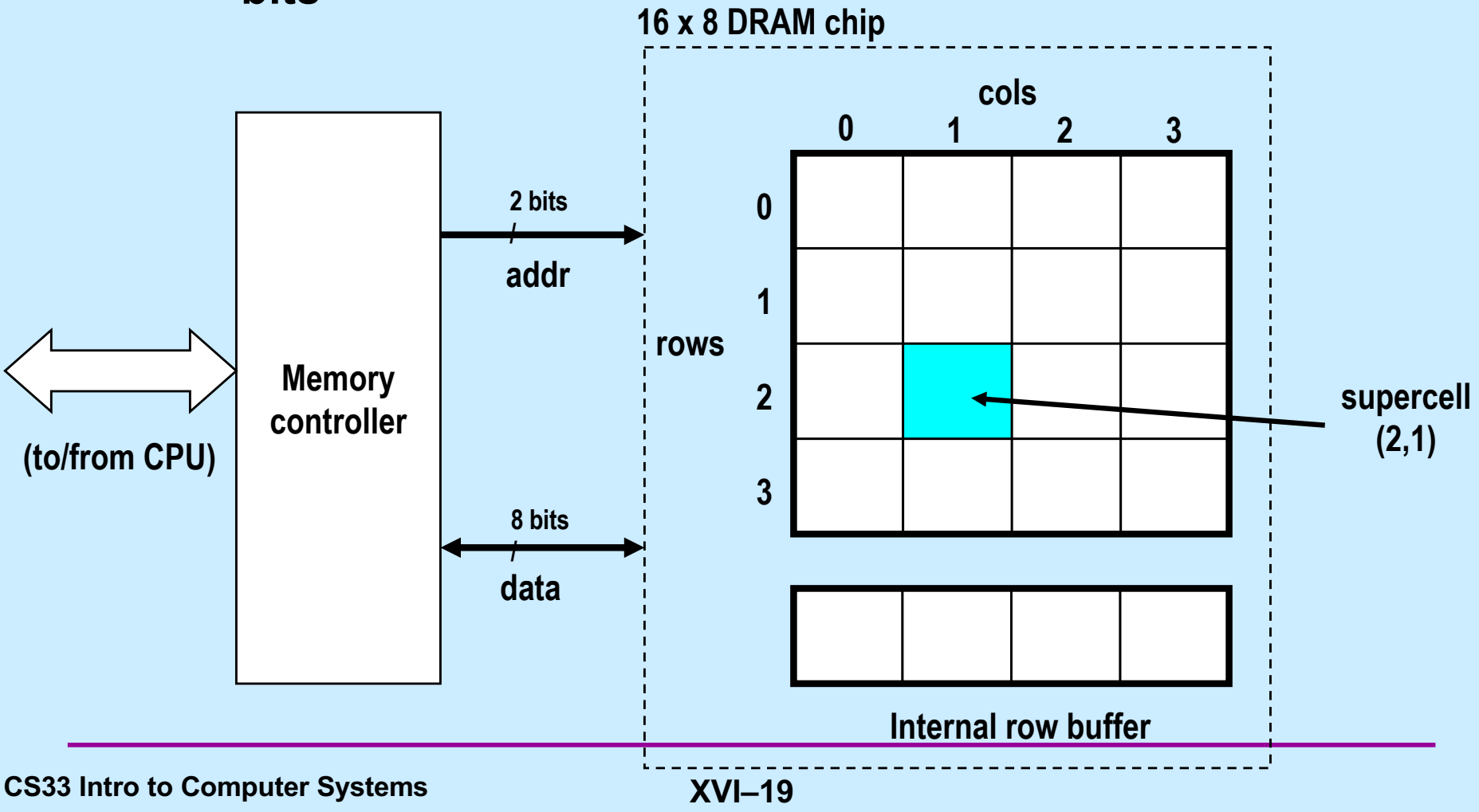
SRAM vs DRAM Summary

	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

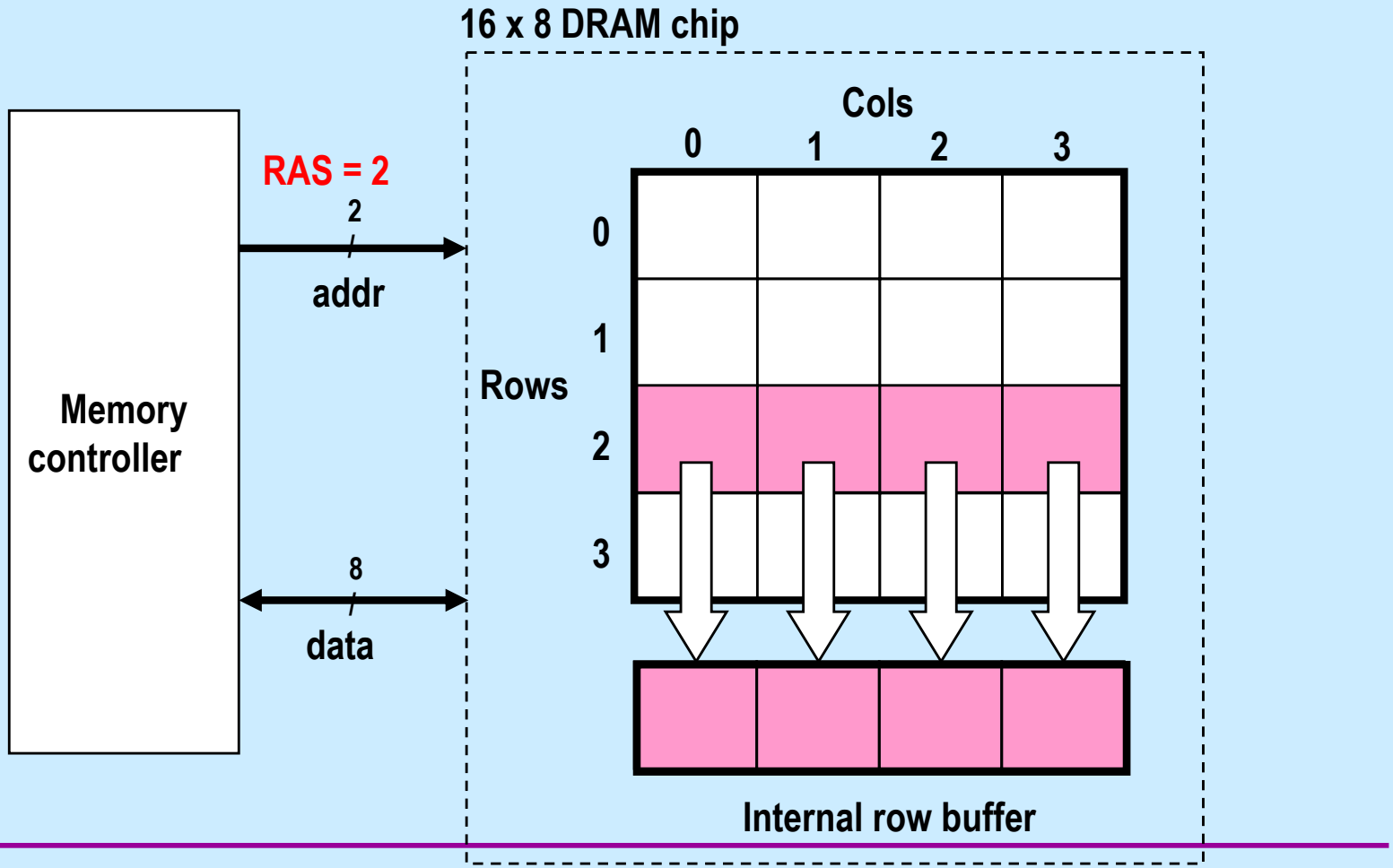
- d x w DRAM:
  - dw total bits organized as d supercells of size w bits



Reading DRAM Supercell (2,1)

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

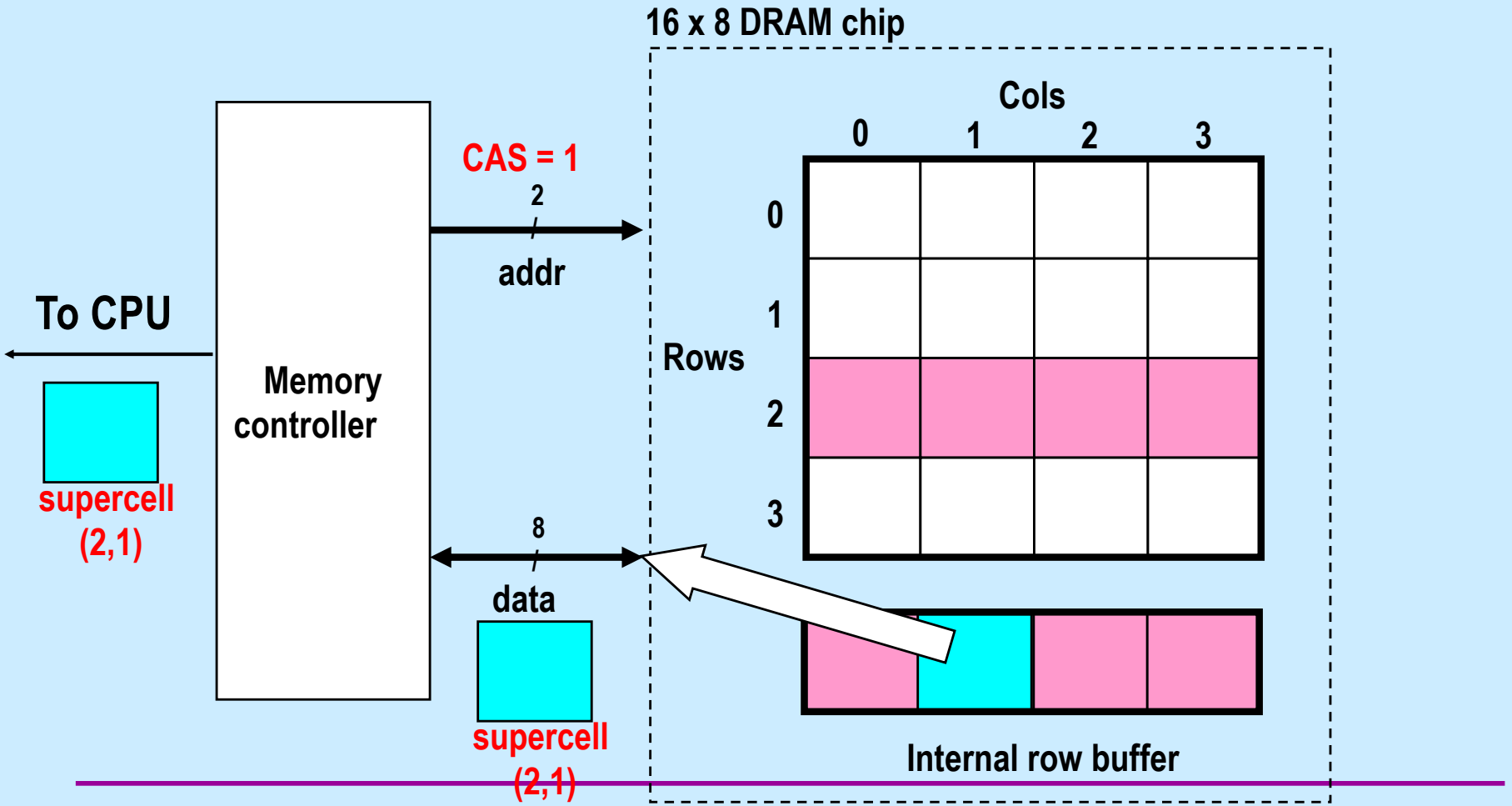
Step 1(b): row 2 copied from DRAM array to row buffer



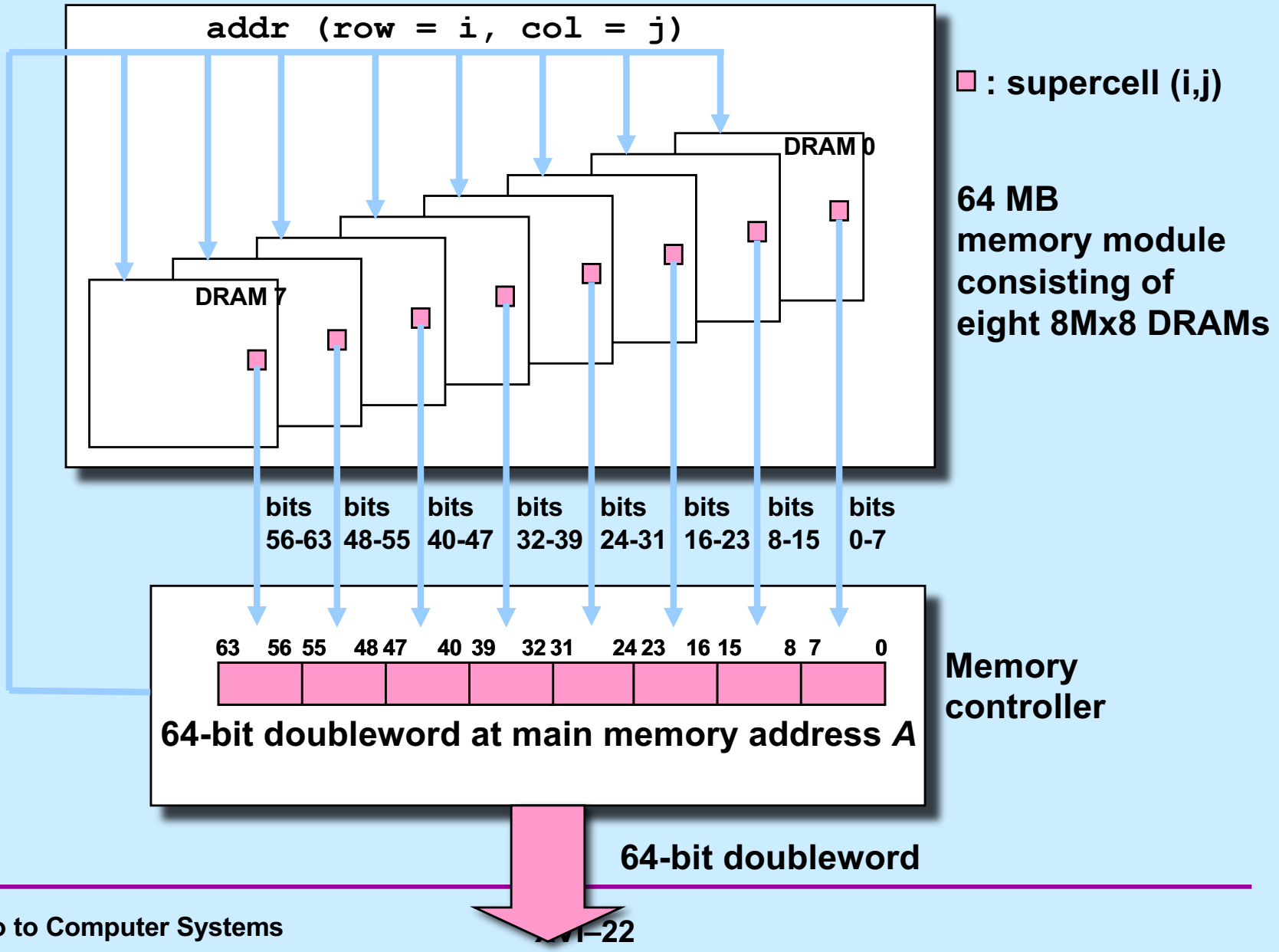
Reading DRAM Supercell (2,1)

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



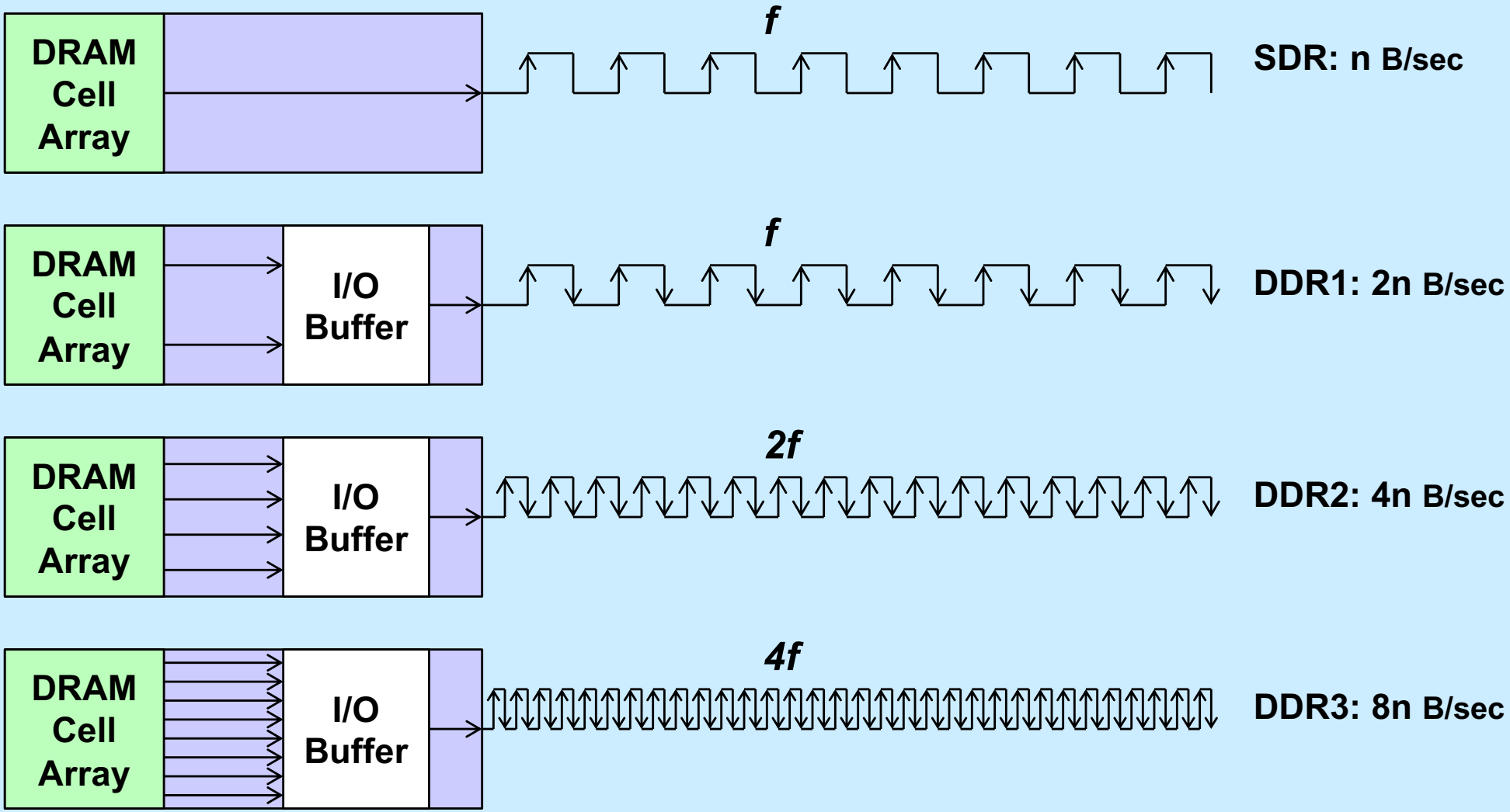
Memory Modules



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)
    - » 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
    - » DDR2
      - four times as fast
    - » DDR3
      - eight times as fast

Enhanced DRAMs



Summary

- Memory transfer speed increased by a factor of 8
  - no increase in DRAM Cell Array speed
  - 8 times more data transferred at once
    - » 64 adjacent bytes fetched from DRAM

Quiz 1

A program is loading randomly selected bytes from memory. These bytes will be delivered to the processor on a DDR3 system at a speed that's n times that of an SDR system, where n is:

a) 1

b) 2

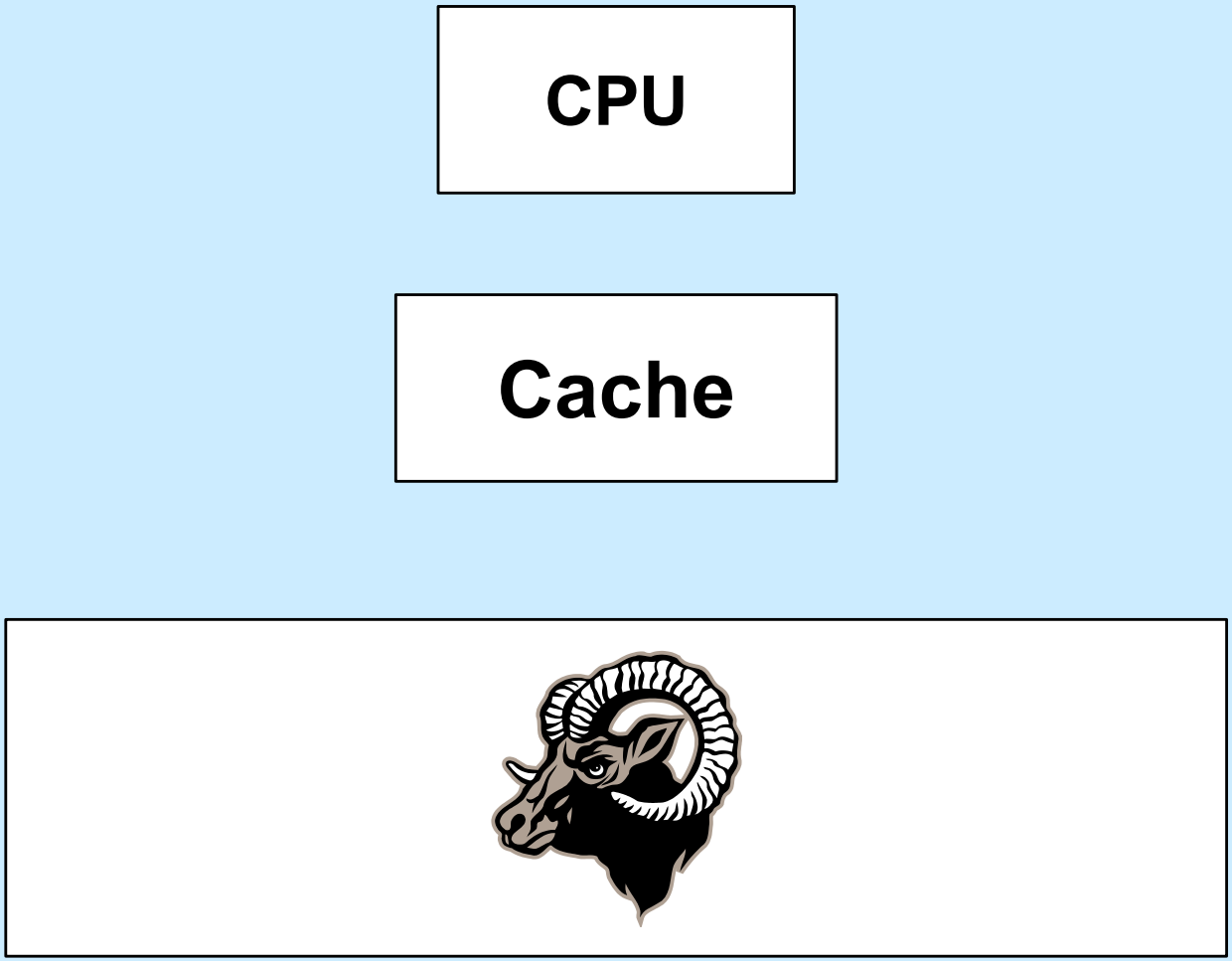
c) 4

d) 8

A Mismatch

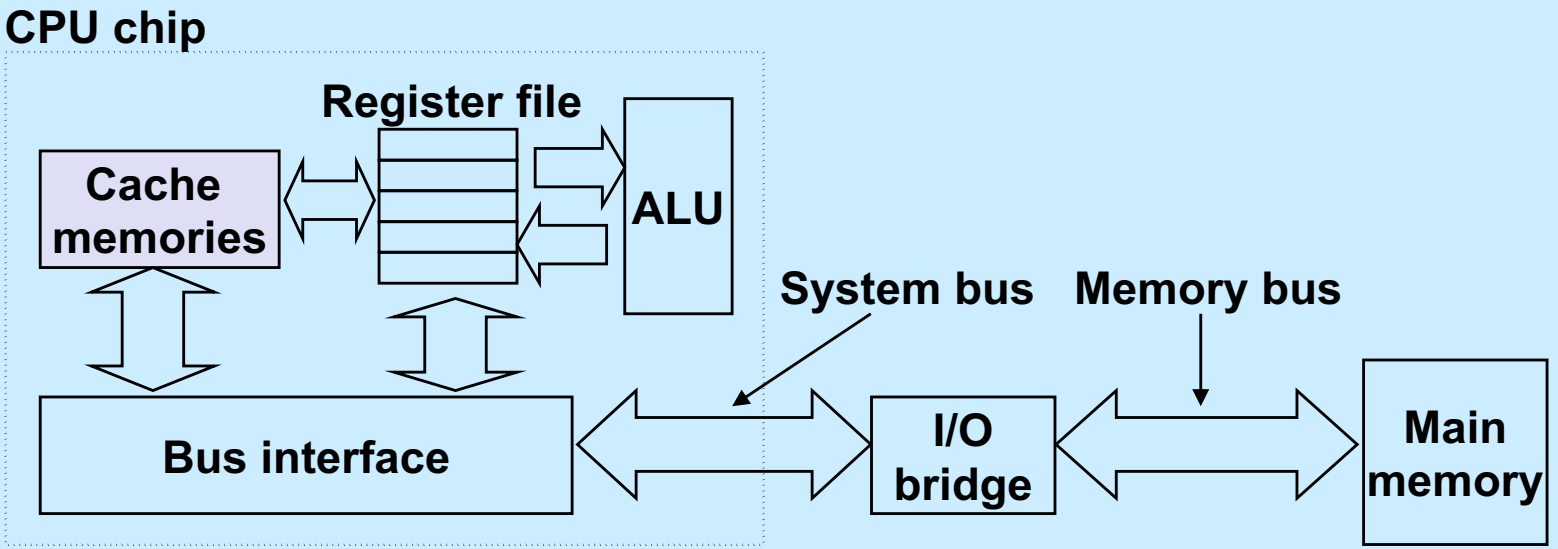
- A processor clock cycle is ~0.3 nsecs
  - SunLab machines (Intel Core i5-4690) run at 3.5 GHz
- Basic operations take 1 – 10 clock cycles
  - .3 – 3 nsecs
- Accessing memory takes 70-100 nsecs
- How is this made to work?

Caching to the Rescue

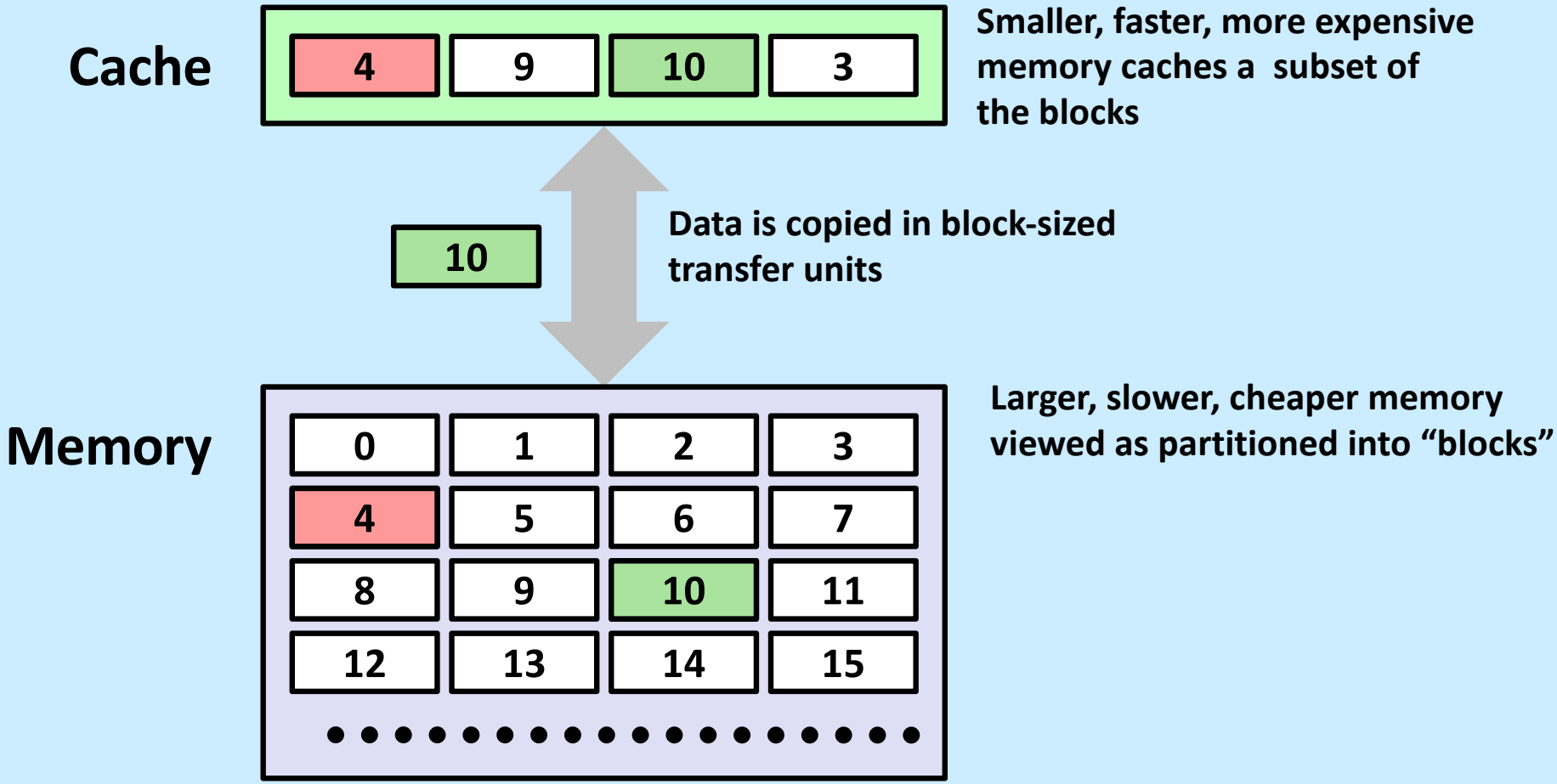


Cache Memories

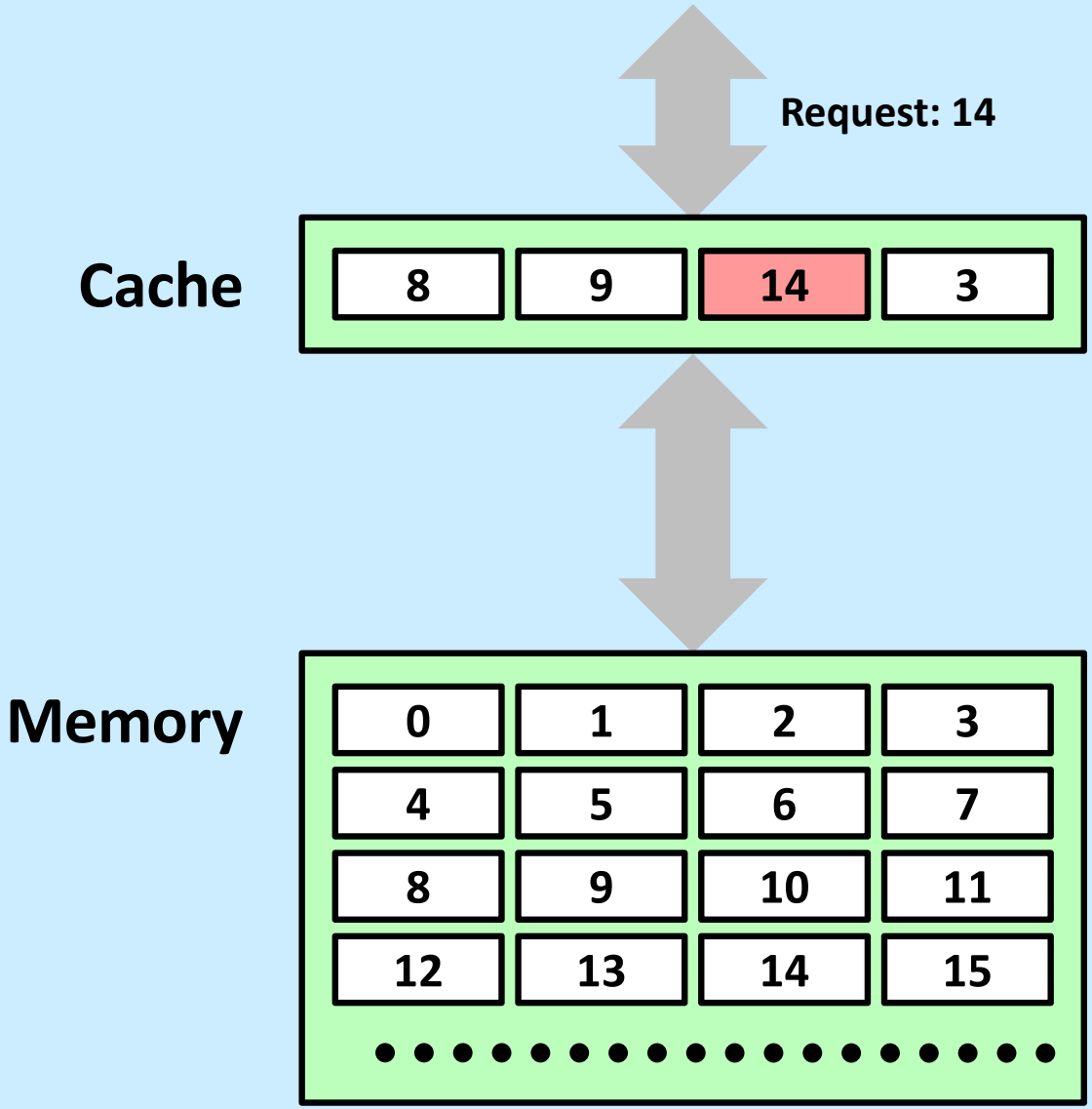
- **Cache memories are small, fast SRAM-based memories managed automatically in hardware**
  - **hold frequently accessed blocks of main memory**
- **CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory**
- **Typical system structure:**



General Cache Concepts



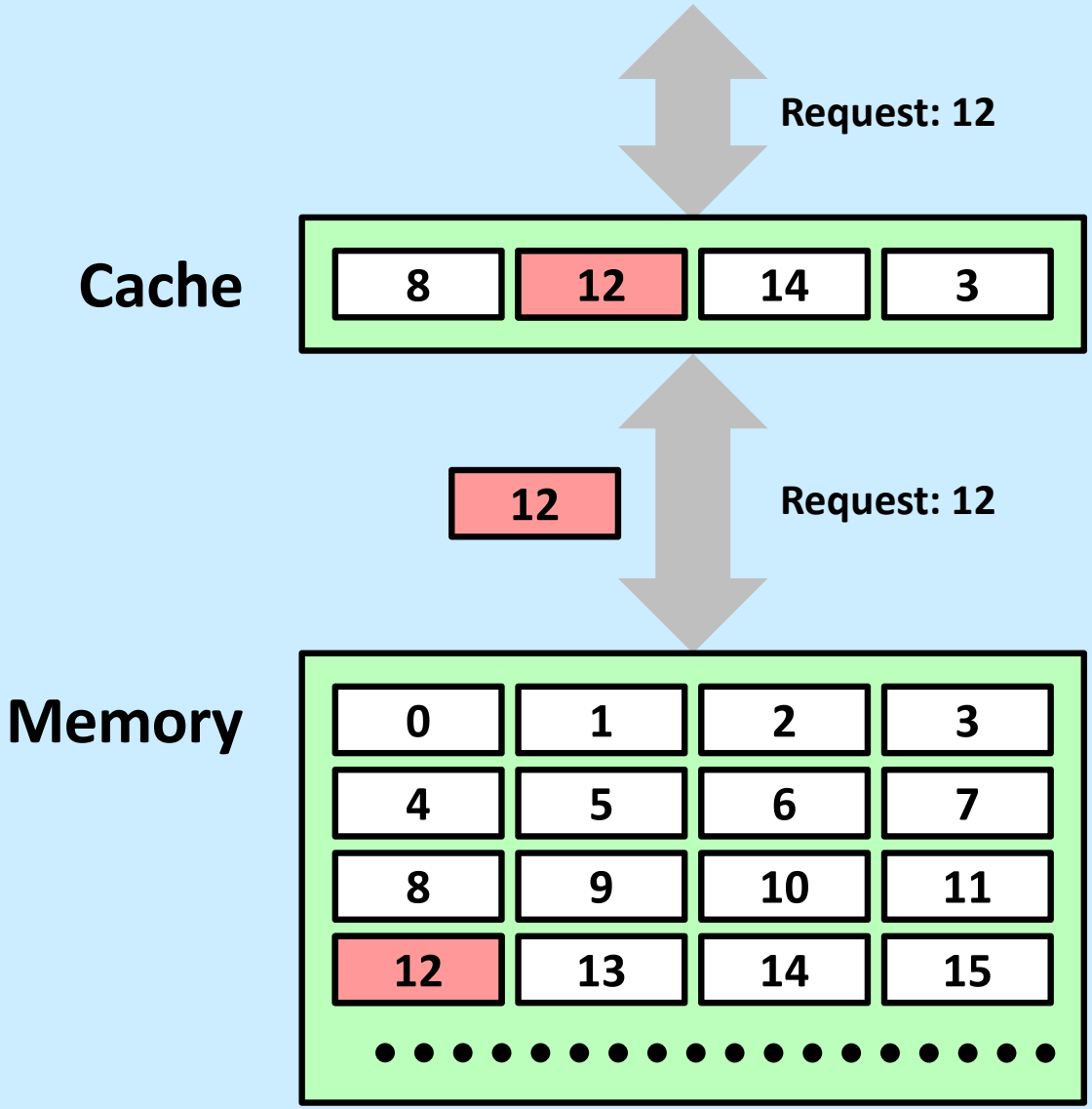
General Cache Concepts: Hit



Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

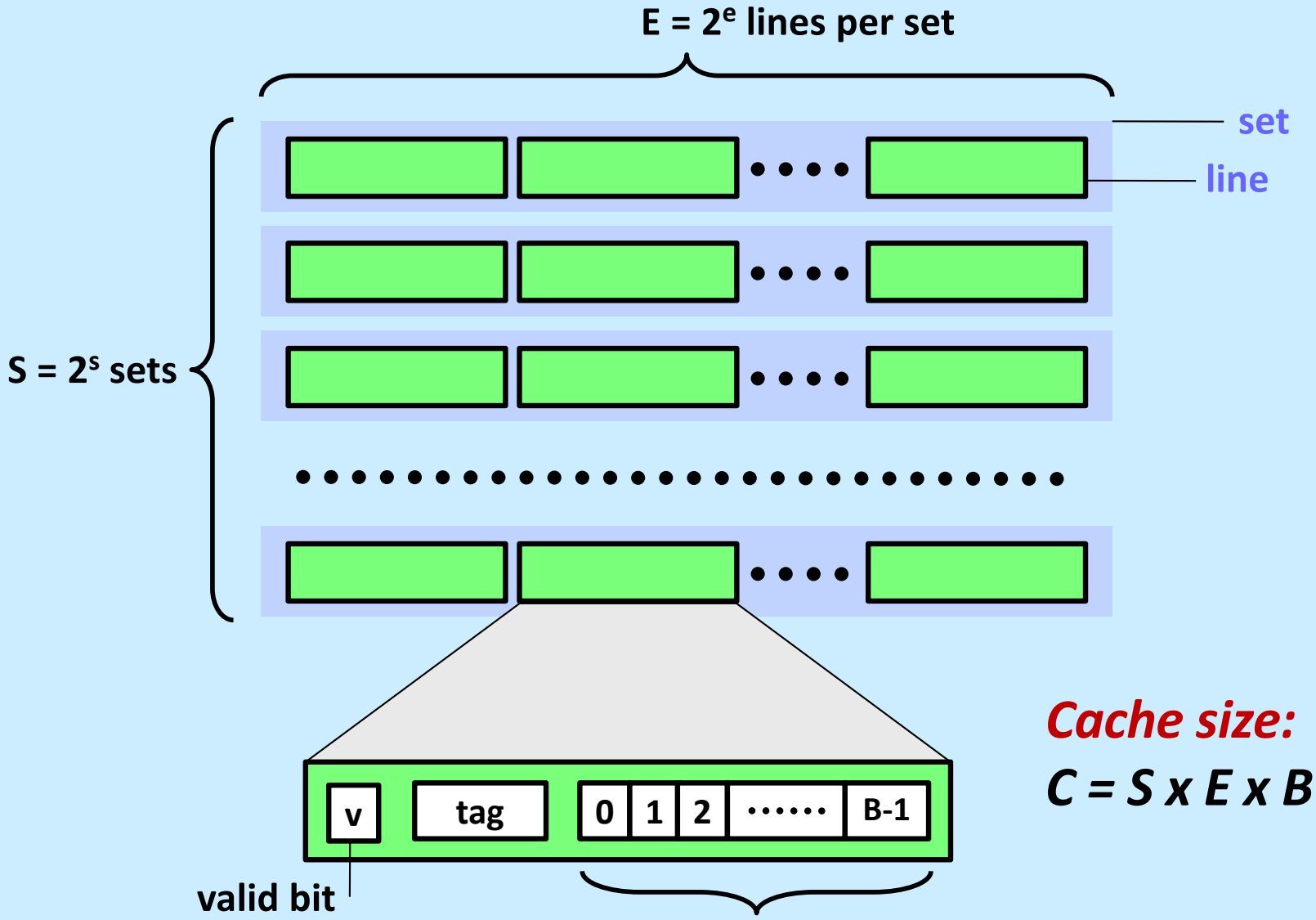
Block b is stored in cache

- placement policy: determines where b goes
- replacement policy: determines which block gets evicted (victim)

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)

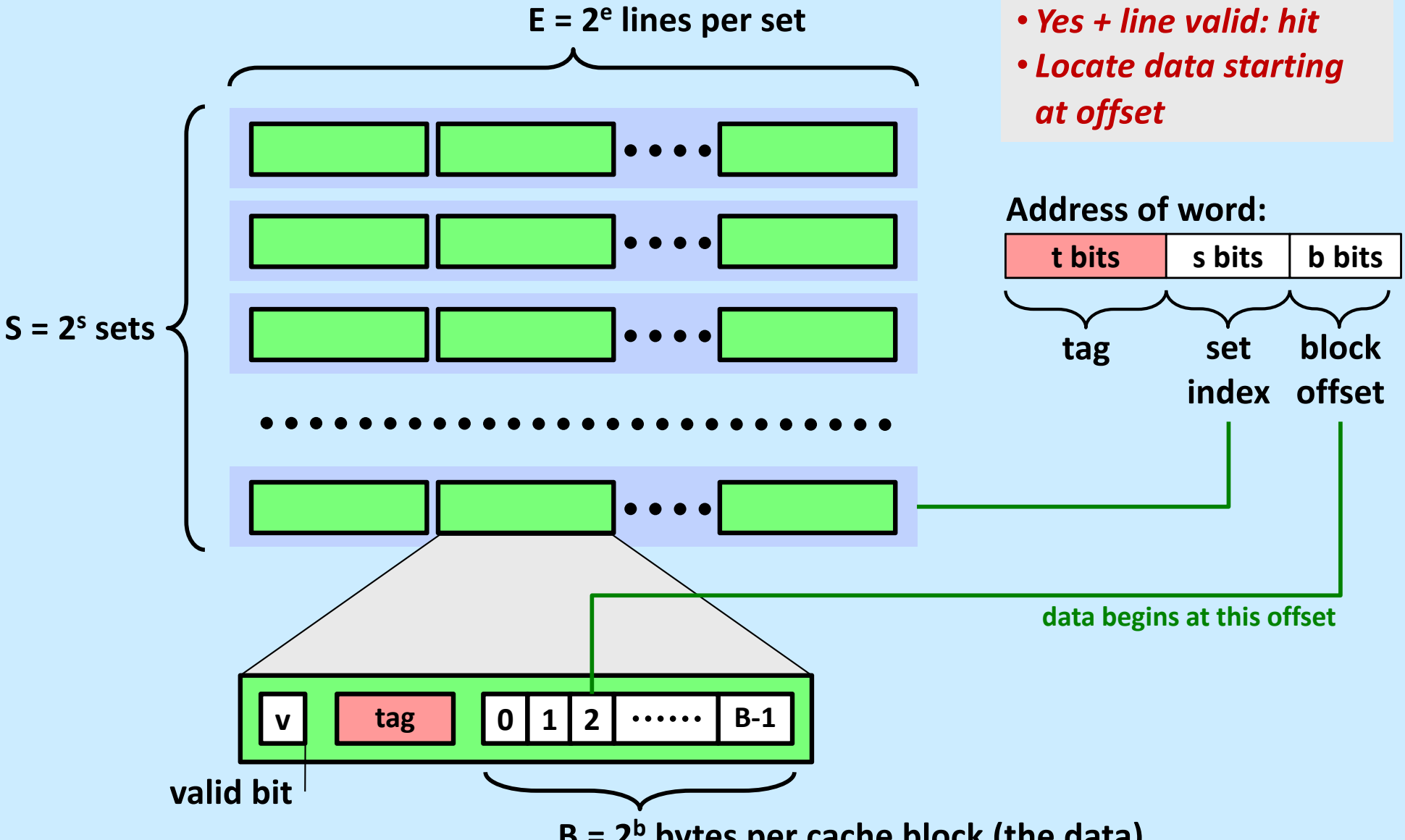


Cache size:

C = S x E x B data bytes

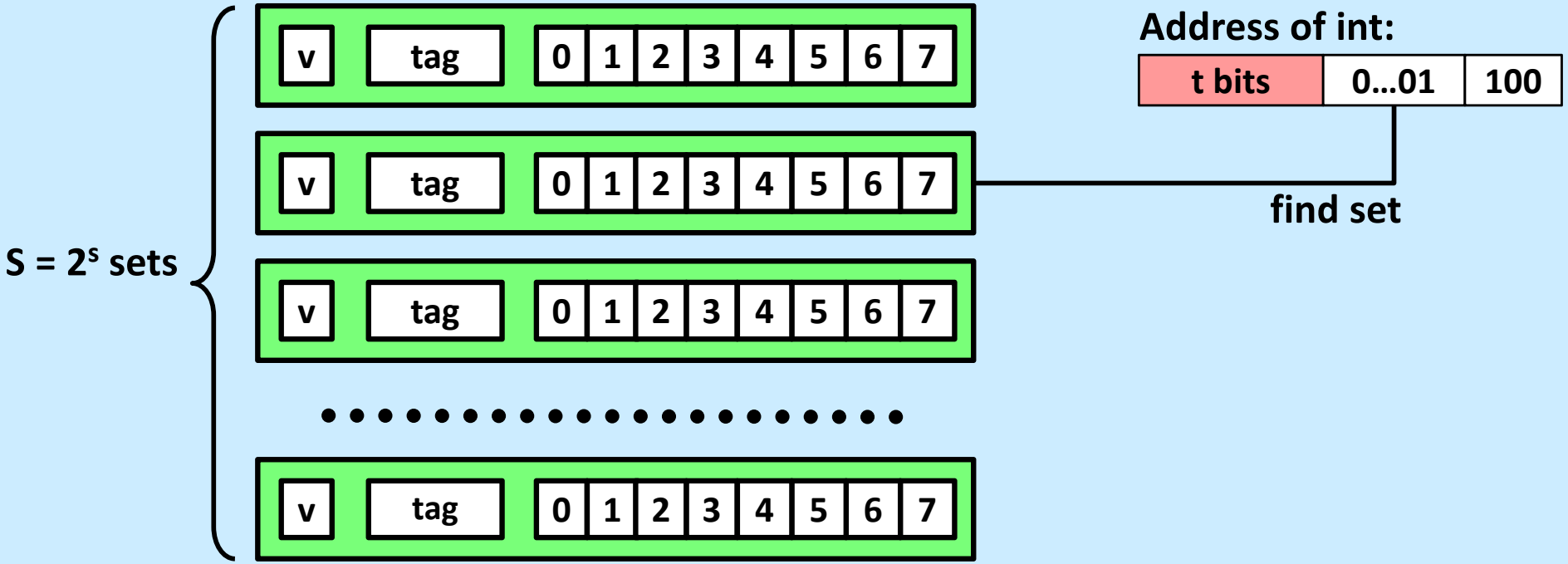
Cache Read

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



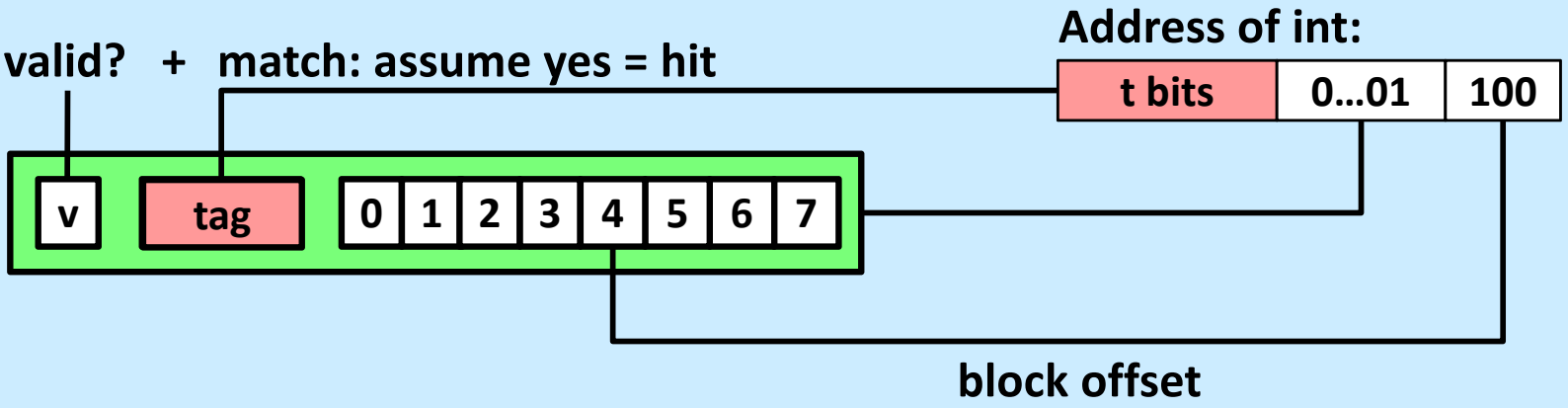
Example: Direct Mapped Cache (E = 1)

Direct mapped: one line per set
Assume: cache block size 8 bytes



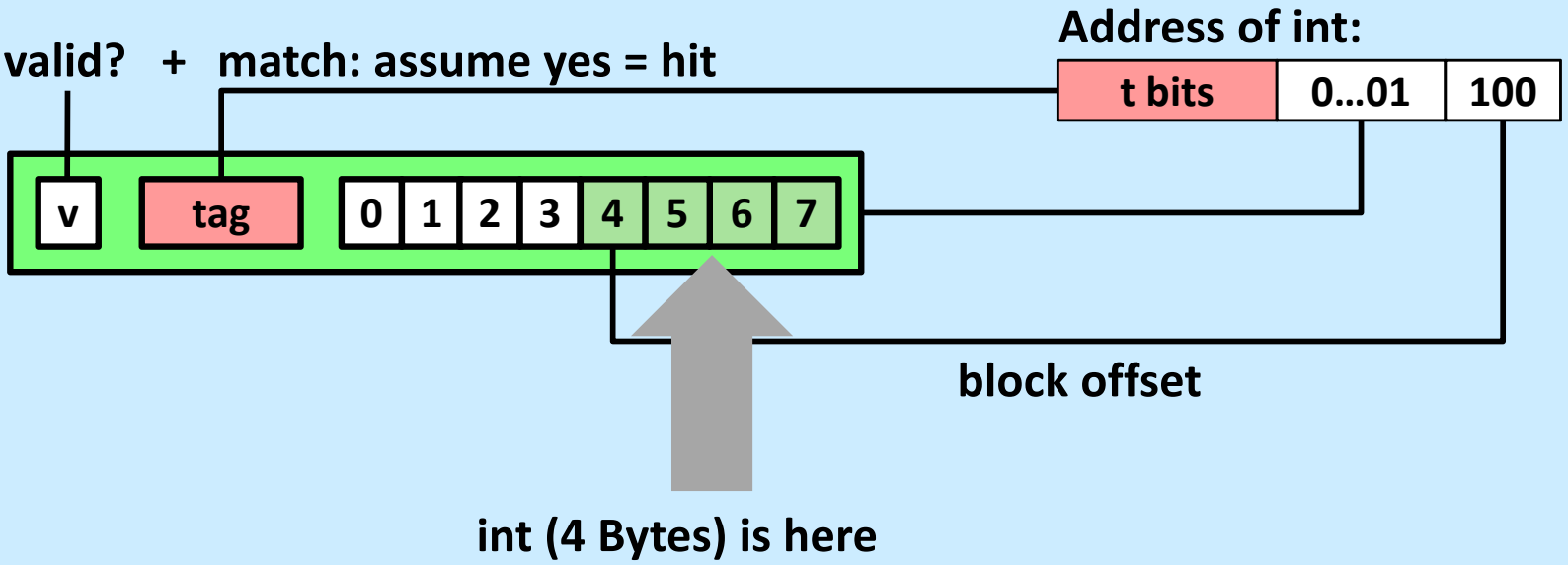
Example: Direct Mapped Cache (E = 1)

Direct mapped: one line per set
Assume: cache block size 8 bytes



Example: Direct Mapped Cache (E = 1)

Direct mapped: one line per set
Assume: cache block size 8 bytes



No match: old line is evicted and replaced

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	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	miss
0	**[0$\underline{00}$0$_2$]**	miss

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

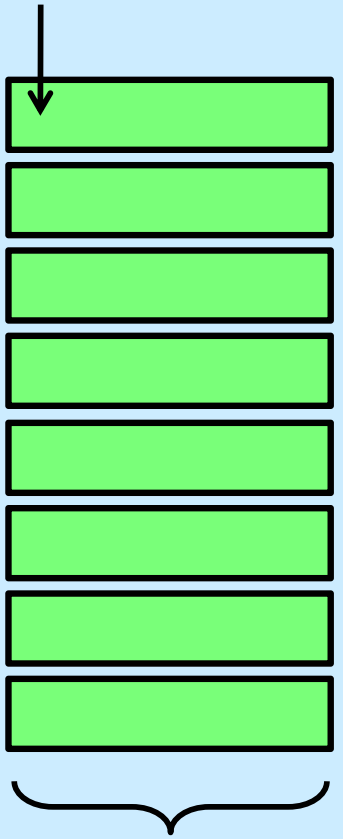
A Higher-Level Example

```
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;
}
```

Ignore the variables sum, i, j

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

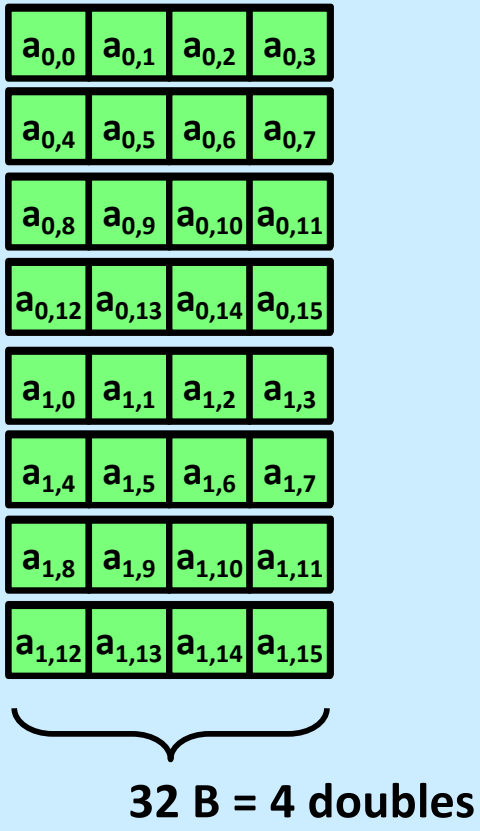


32 B = 4 doubles

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;
}
```

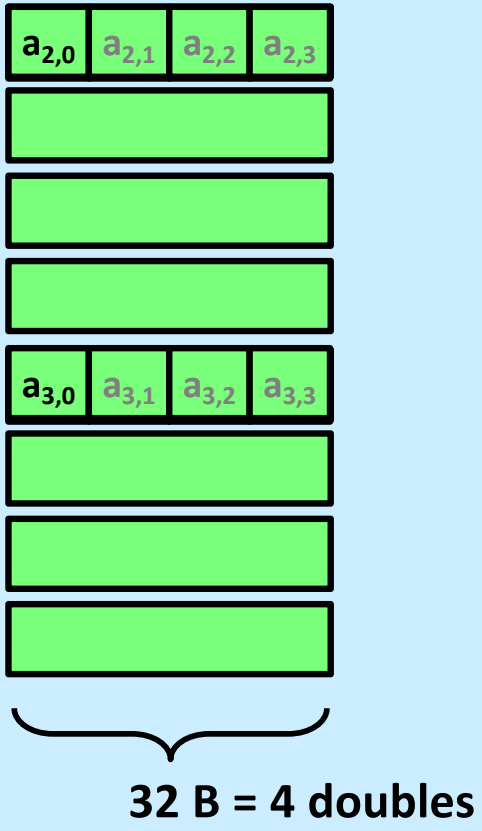
```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

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



A Higher-Level Example

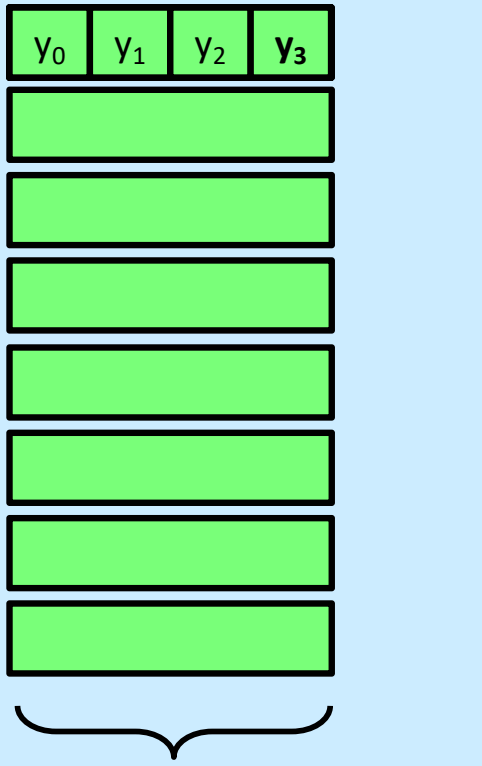
```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    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;
}
```



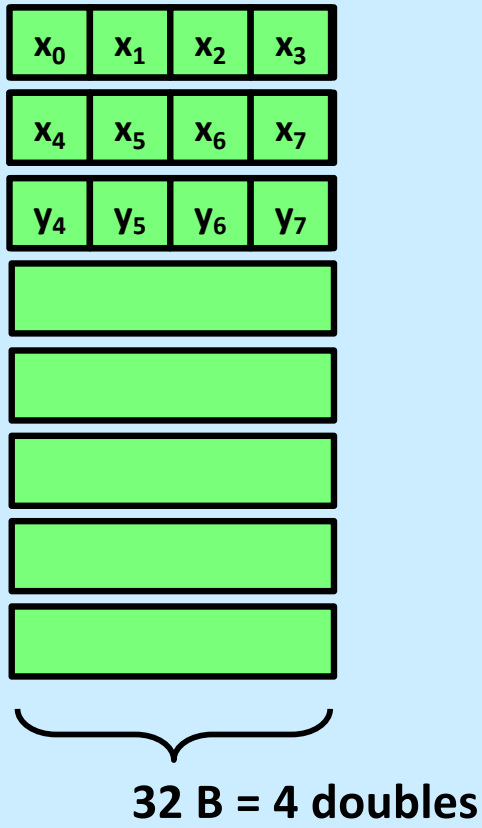
Conflict Misses: Aligned

```
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;
}
```



Different Alignments

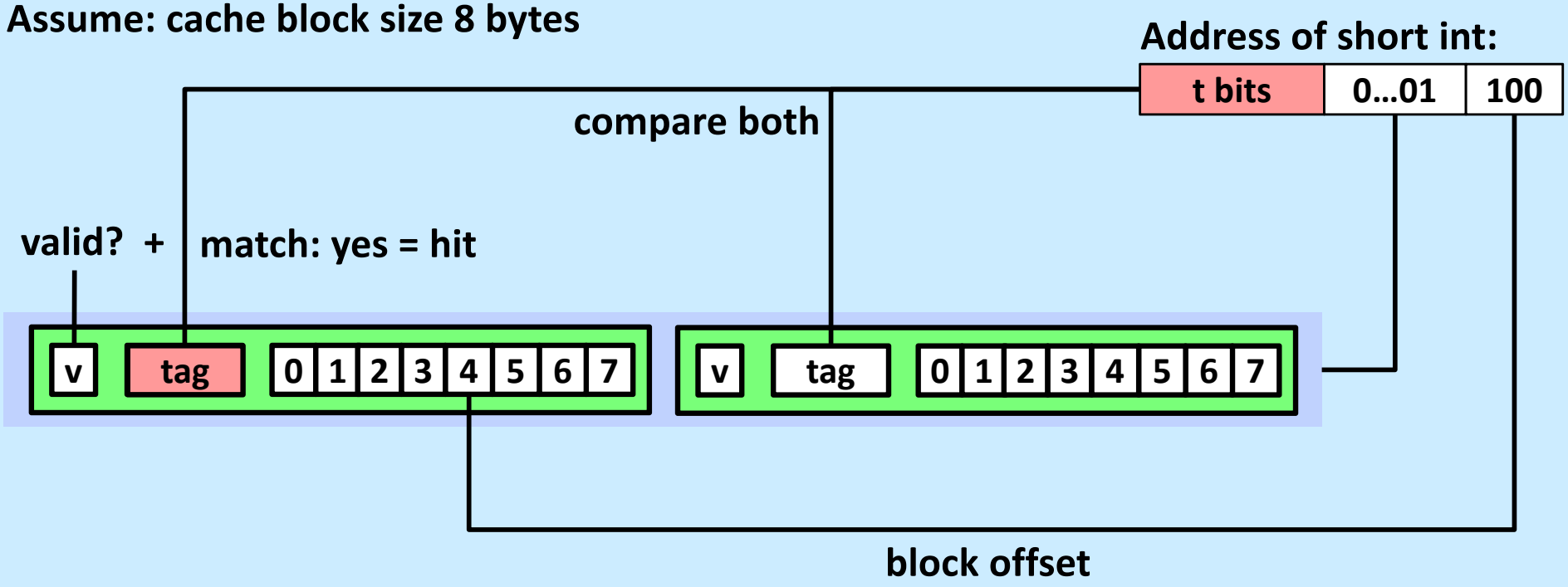
```
double dotprod(double x[8], double y[8]) {
  double sum = 0.0;
  int i;

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

  return sum;
}
```



E-way Set-Associative Cache (Here: E = 2)

E = 2: two lines per set

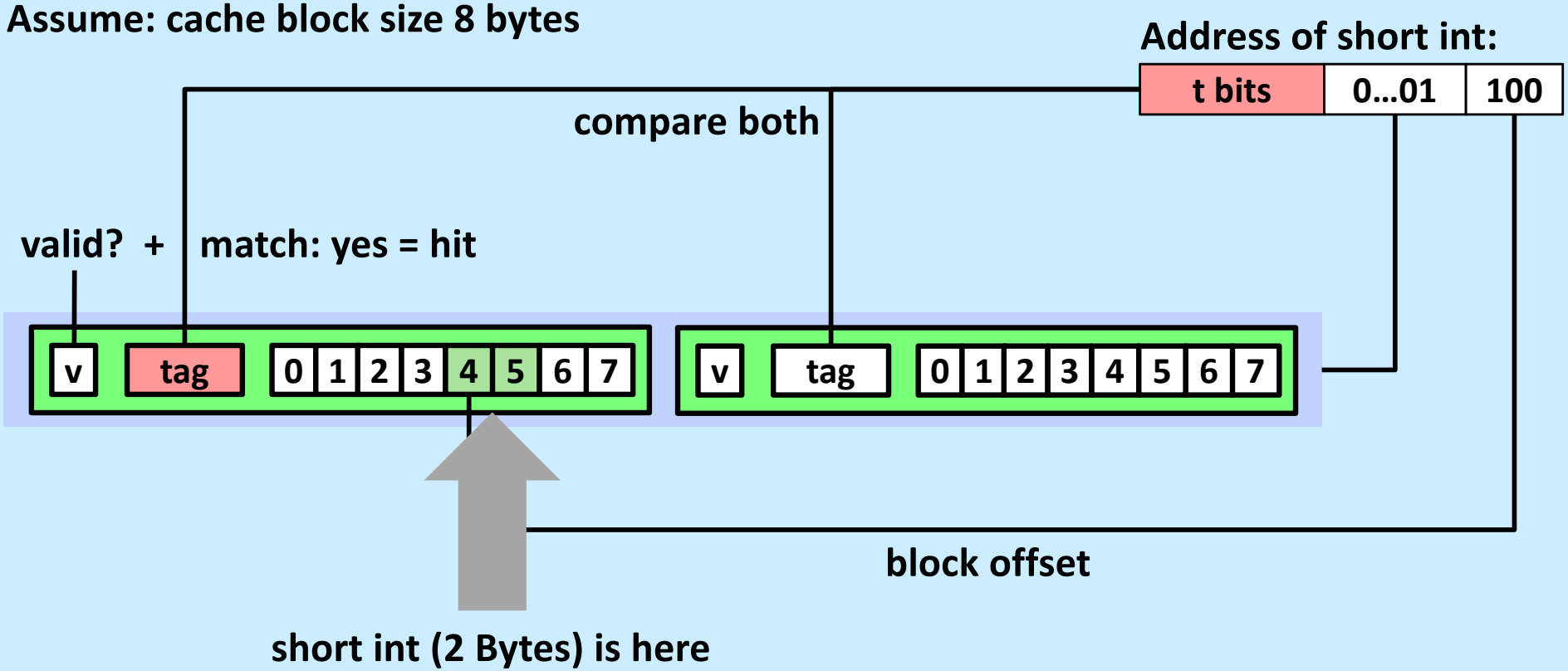
Assume: cache block size 8 bytes



E-way Set-Associative Cache (Here: E = 2)

E = 2: two lines per set

Assume: cache block size 8 bytes



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

Quiz 2

Address of int:

100	01	100

0	v	tag=0	0	0	0	0	1	1	1	1
1	v	tag=0	4	4	4	4	5	5	5	5
2	v	tag=2	8	8	8	8	9	9	9	9
3	v	tag=4	c	c	c	c	d	d	d	d

v	tag=2	2	2	2	2	3	3	3	3
v	tag=4	6	6	6	6	7	7	7	7
v	tag=3	a	a	a	a	b	b	b	b
v	tag=a	e	e	e	e	f	f	f	f

Given the address above and the cache contents as shown, what is the value of the *int* at the given address?

a) 1111
b) 3333
c) 4444
d) 7777

2-Way Set-Associative Cache Simulation

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

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 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$_2$]**	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

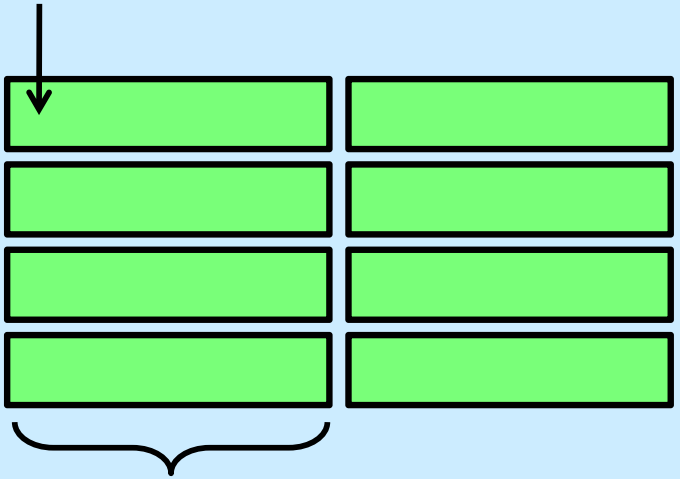
A Higher-Level Example

```
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_rows(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;
}
```

Ignore the variables sum, i, j

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



32 B = 4 doubles

A Higher-Level Example

Ignore the variables sum, i, j

```
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;
}
```

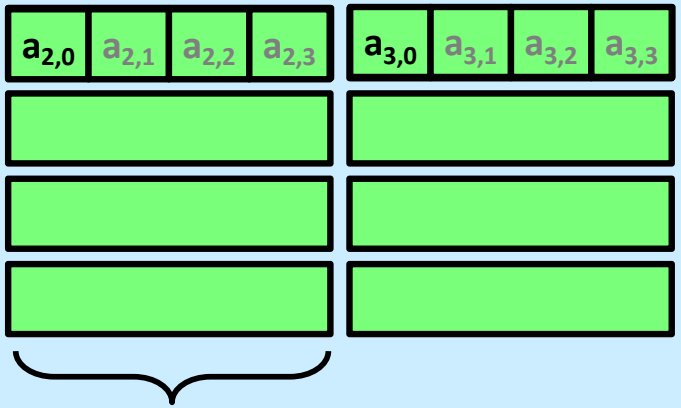
```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

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

A Higher-Level Example

Ignore the variables sum, i, j

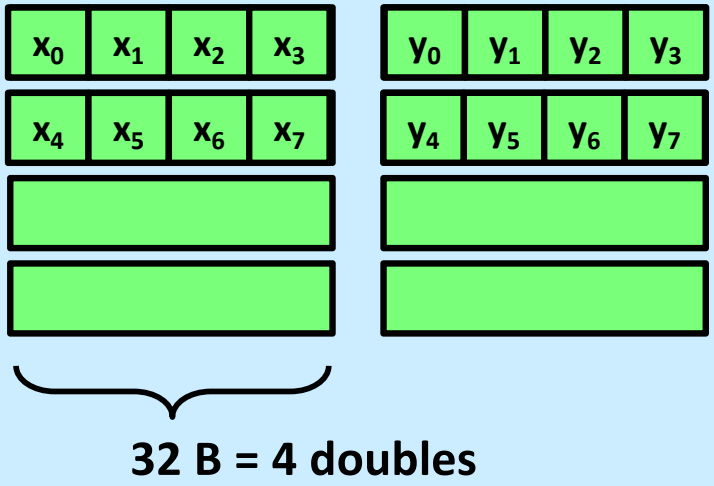
```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    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;
}
```

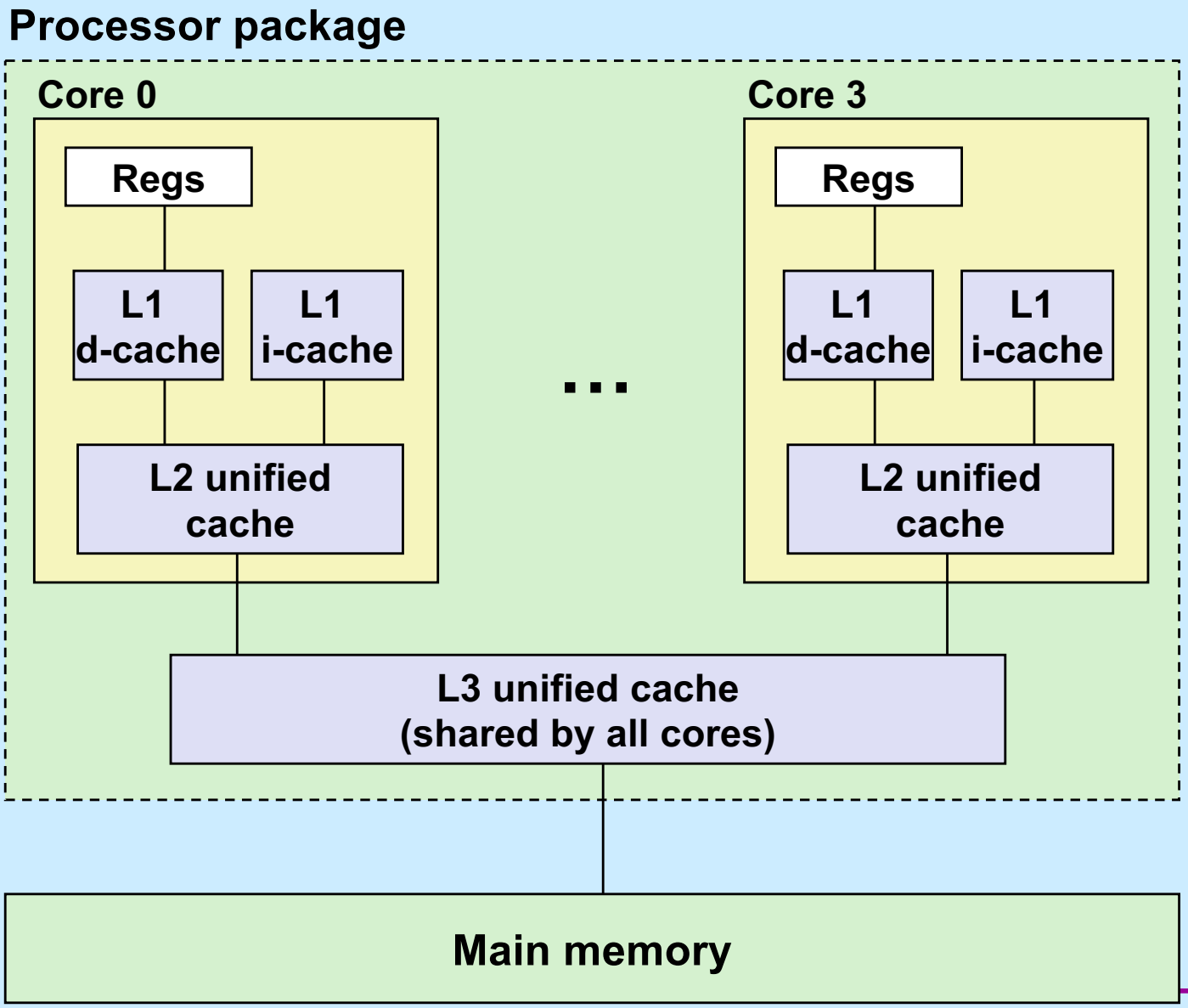


Conflict Misses

```
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;
}
```



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