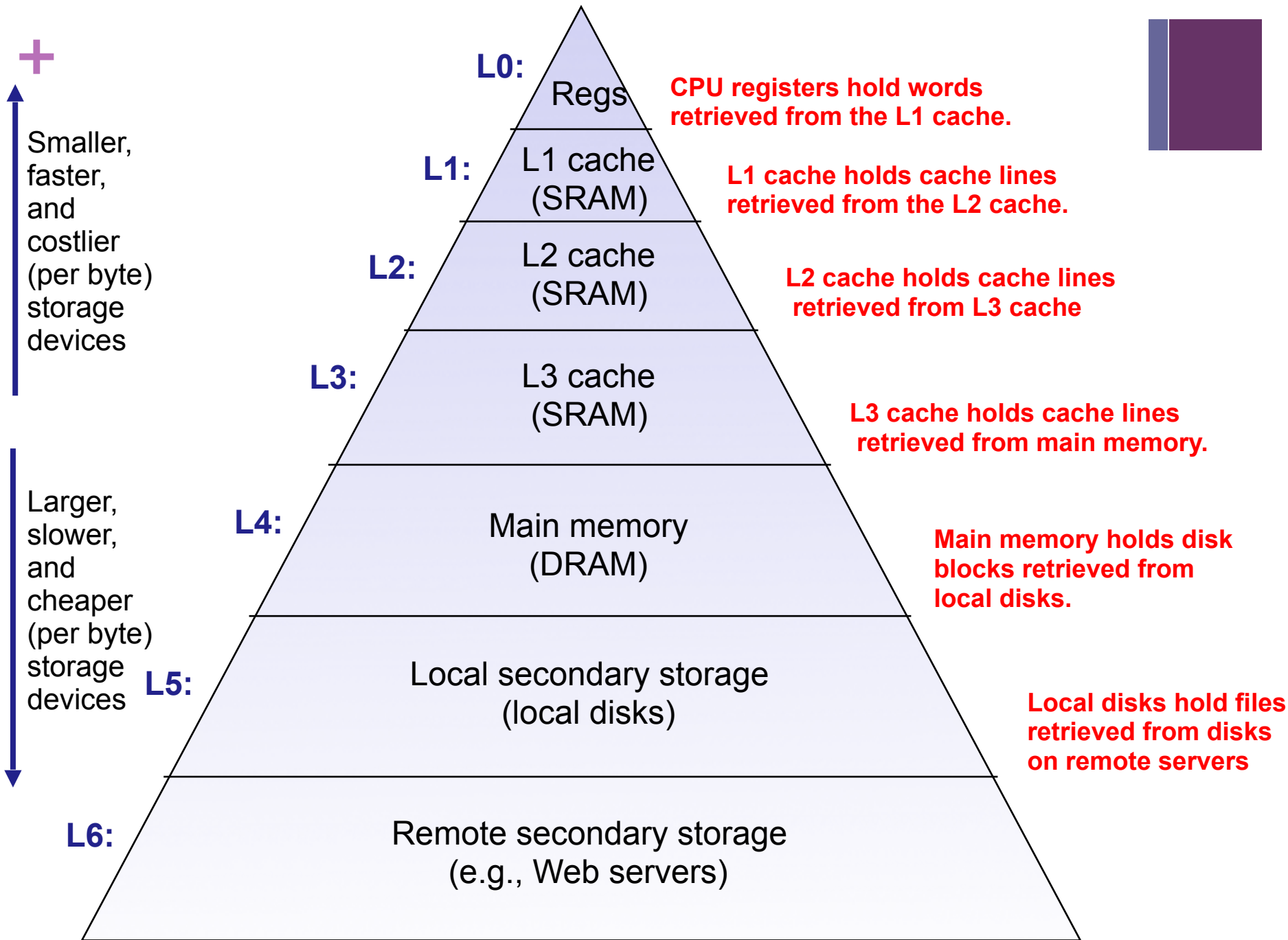


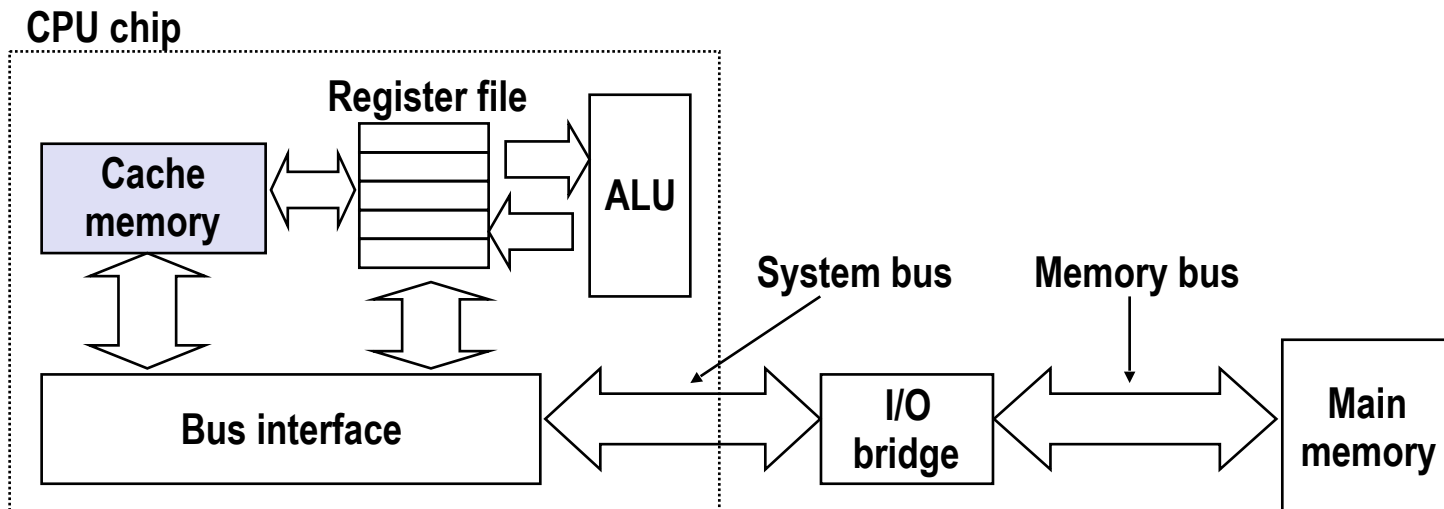


Cache Memories



+ Cache Memories

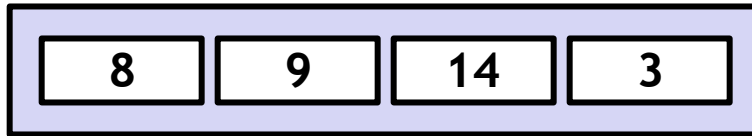
- Cache memories are small, fast SRAM-based memories managed automatically in hardware
 - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:



+ General Cache Concept



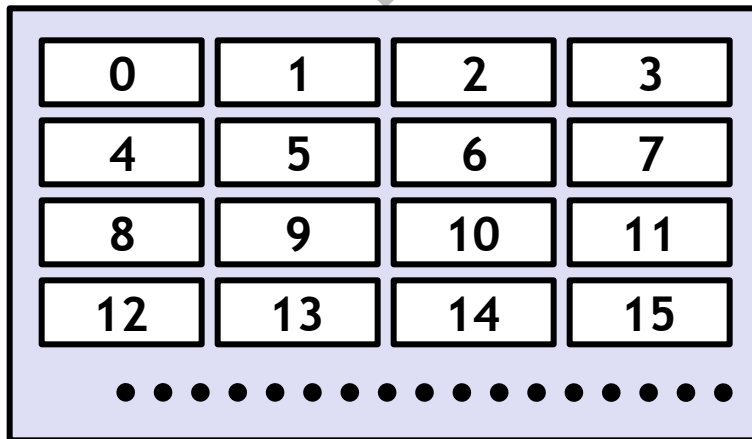
Cache



Smaller, faster, more expensive memory caches a subset of the blocks

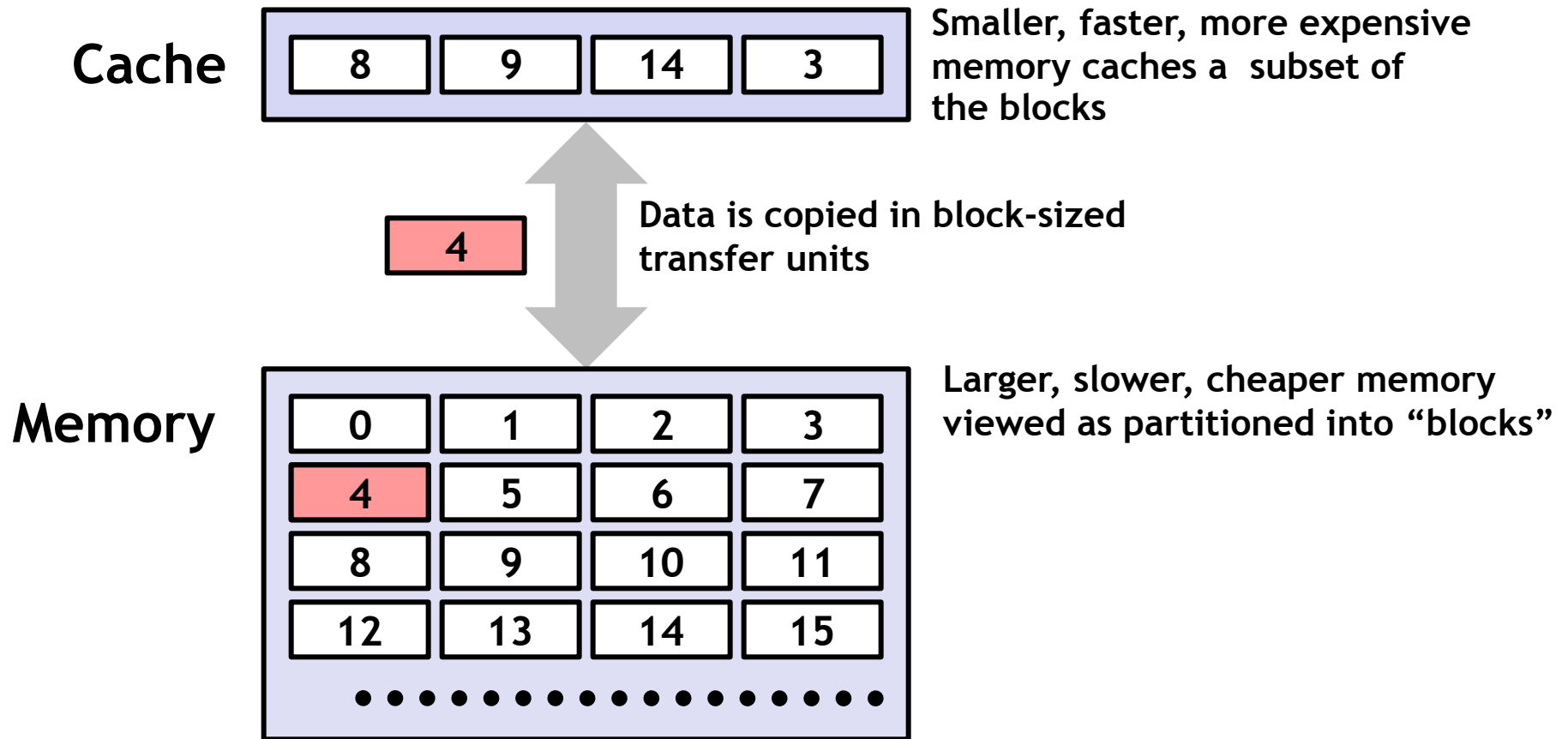


Memory



Larger, slower, cheaper memory viewed as partitioned into “blocks”

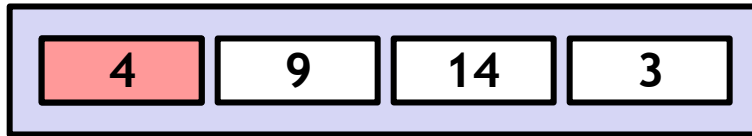
+ General Cache Concept



+ General Cache Concept



Cache

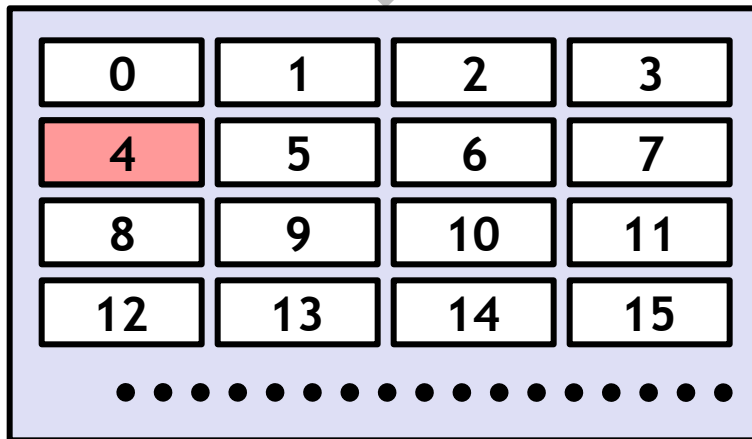


Smaller, faster, more expensive memory caches a subset of the blocks



Data is copied in block-sized transfer units

Memory

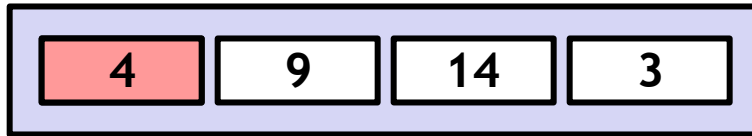


Larger, slower, cheaper memory viewed as partitioned into “blocks”

+ General Cache Concept



Cache

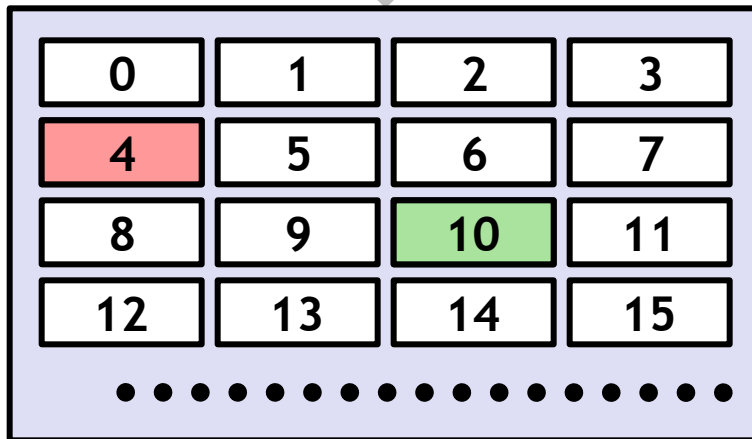


Smaller, faster, more expensive memory caches a subset of the blocks



Data is copied in block-sized transfer units

Memory

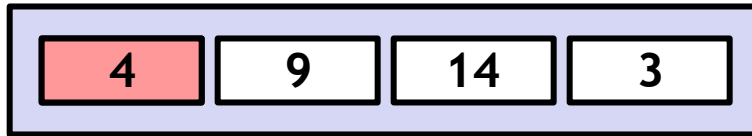


Larger, slower, cheaper memory viewed as partitioned into “blocks”

+ General Cache Concept



Cache

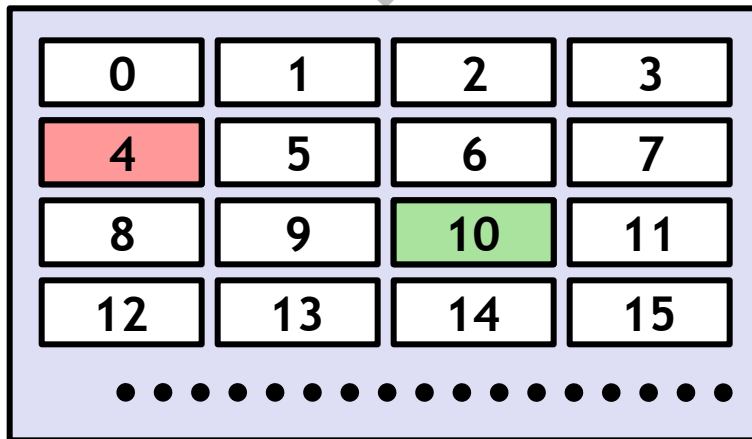


Smaller, faster, more expensive memory caches a subset of the blocks



Data is copied in block-sized transfer units

Memory

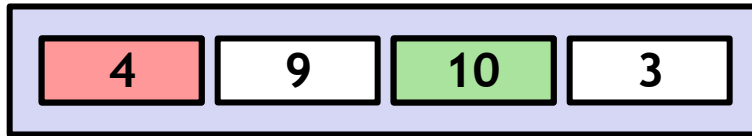


Larger, slower, cheaper memory viewed as partitioned into “blocks”

+ General Cache Concept



Cache

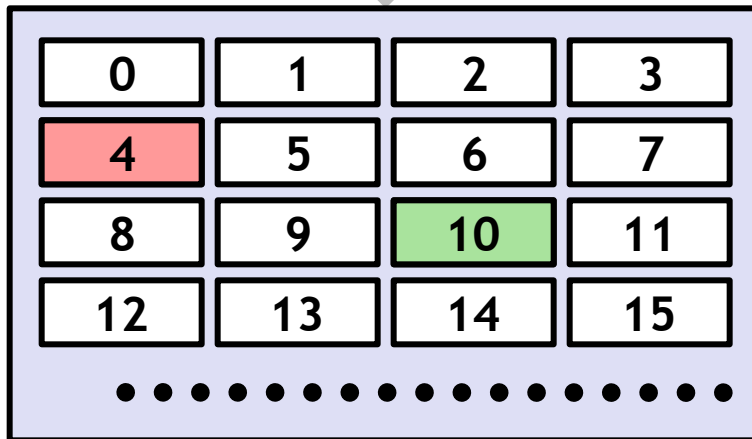


Smaller, faster, more expensive memory caches a subset of the blocks



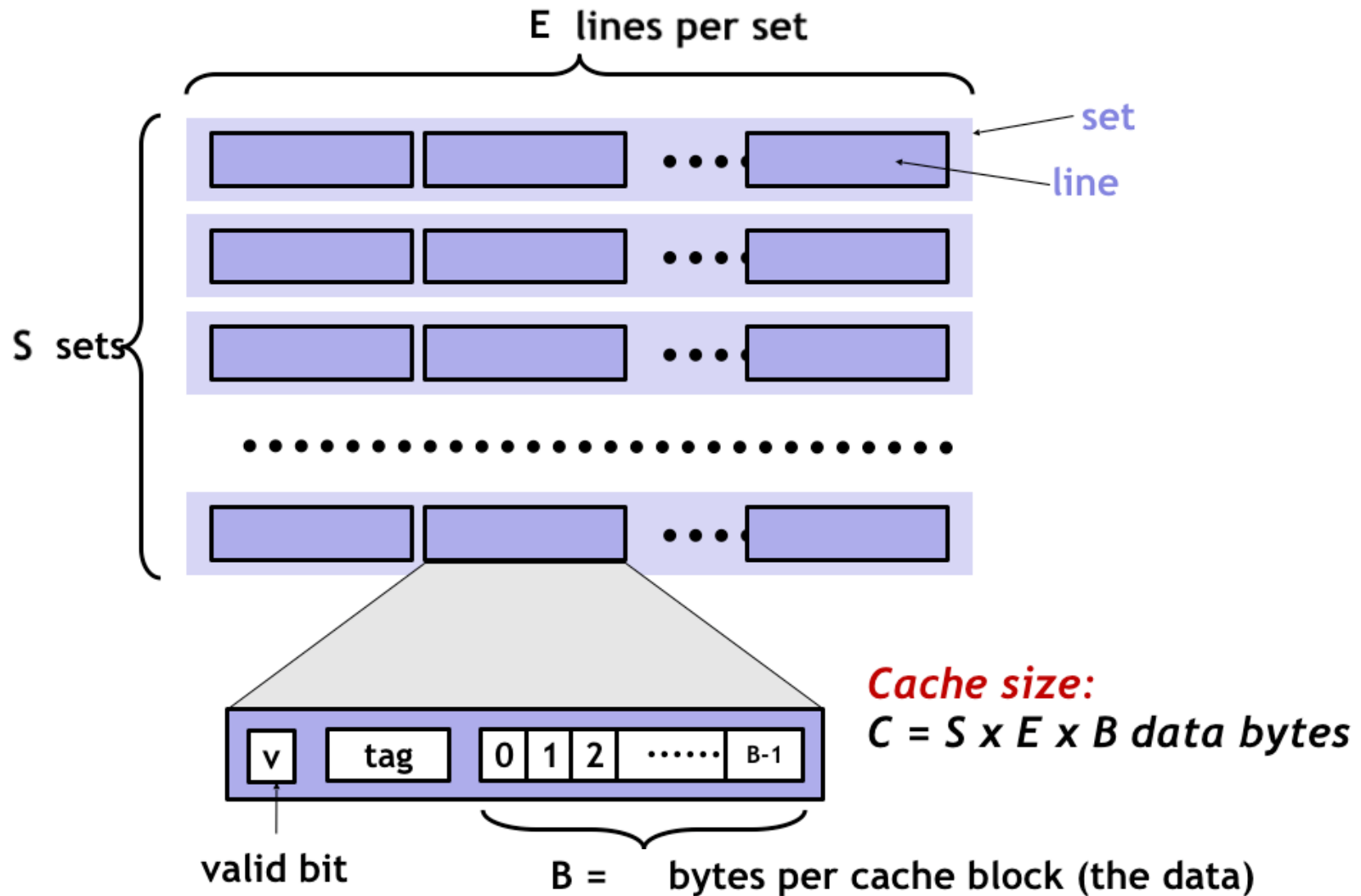
Data is copied in block-sized transfer units

Memory

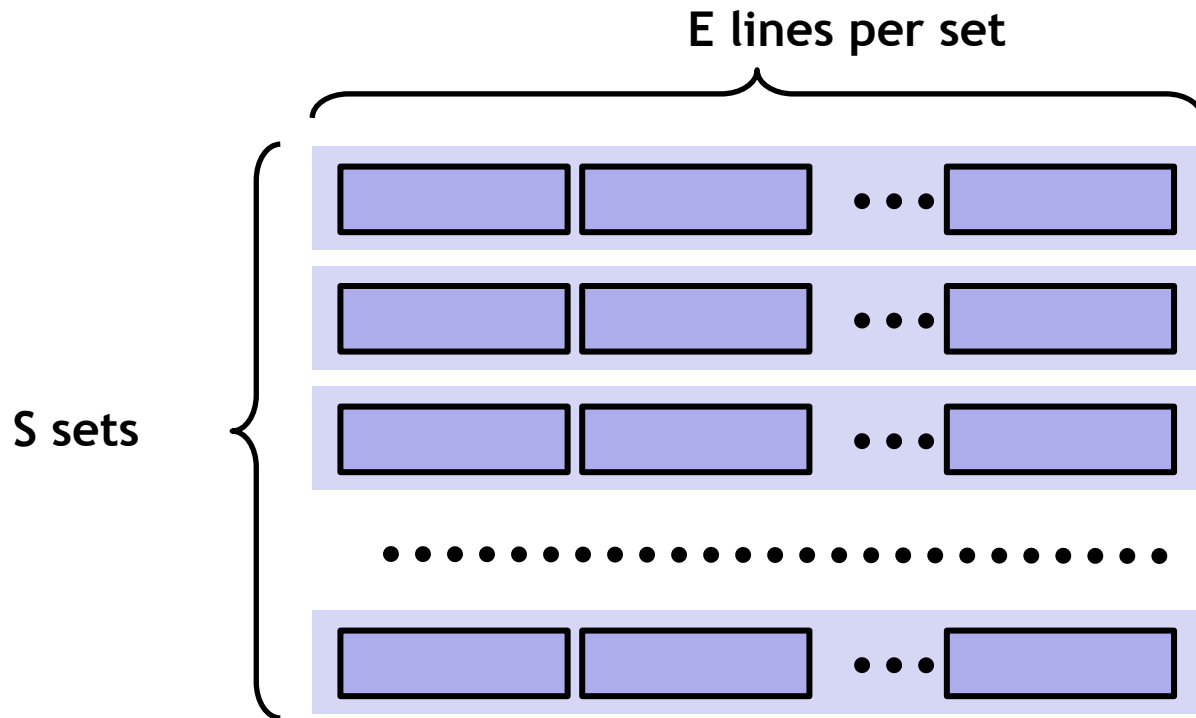
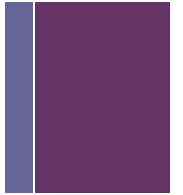


Larger, slower, cheaper memory viewed as partitioned into “blocks”

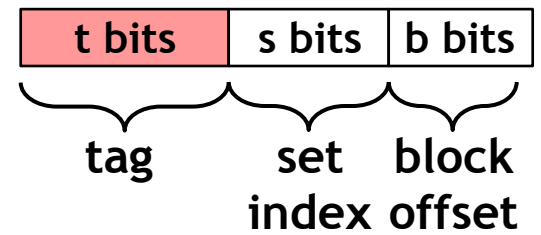
+ General Cache Organization



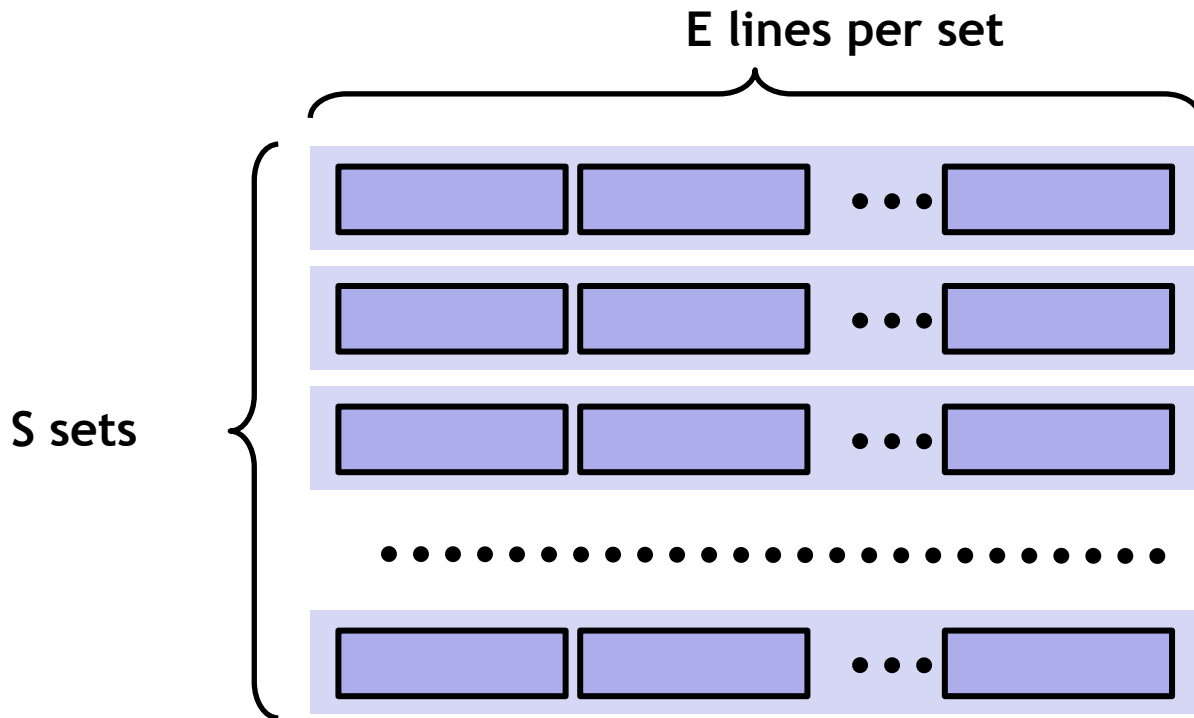
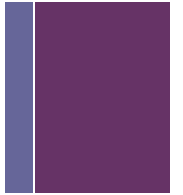
+ Cache Read



Address of word:

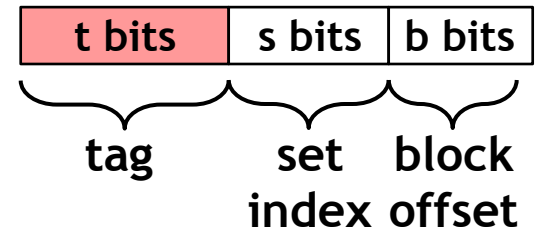


+ Cache Read

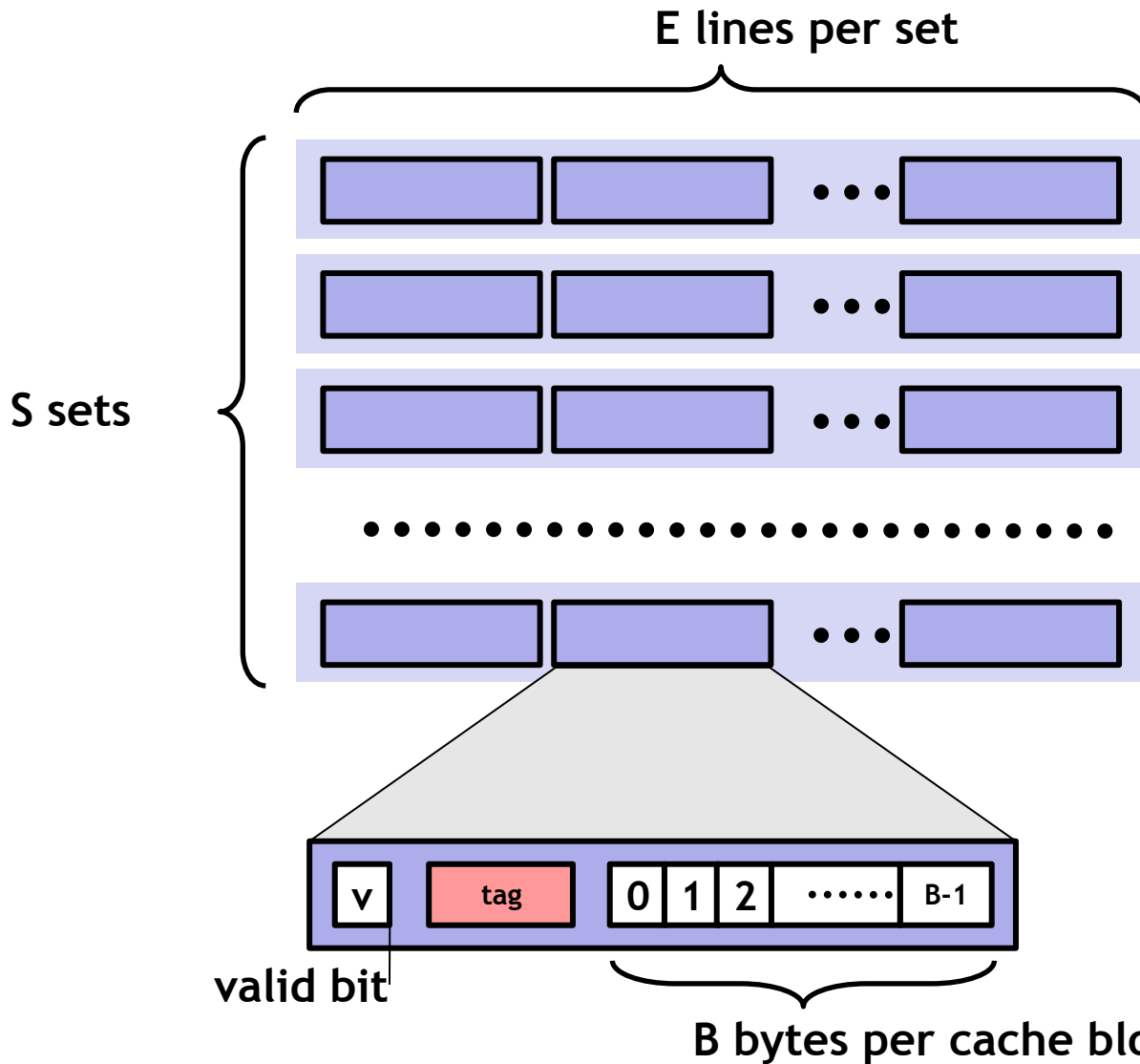
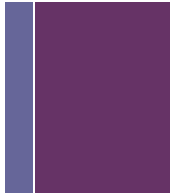


• *Locate set*

Address of word:

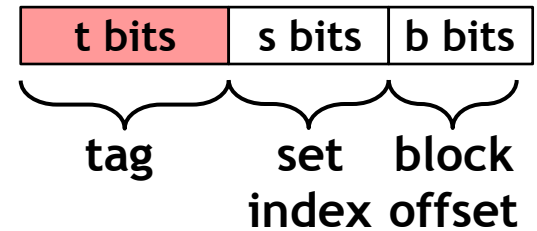


+ Cache Read

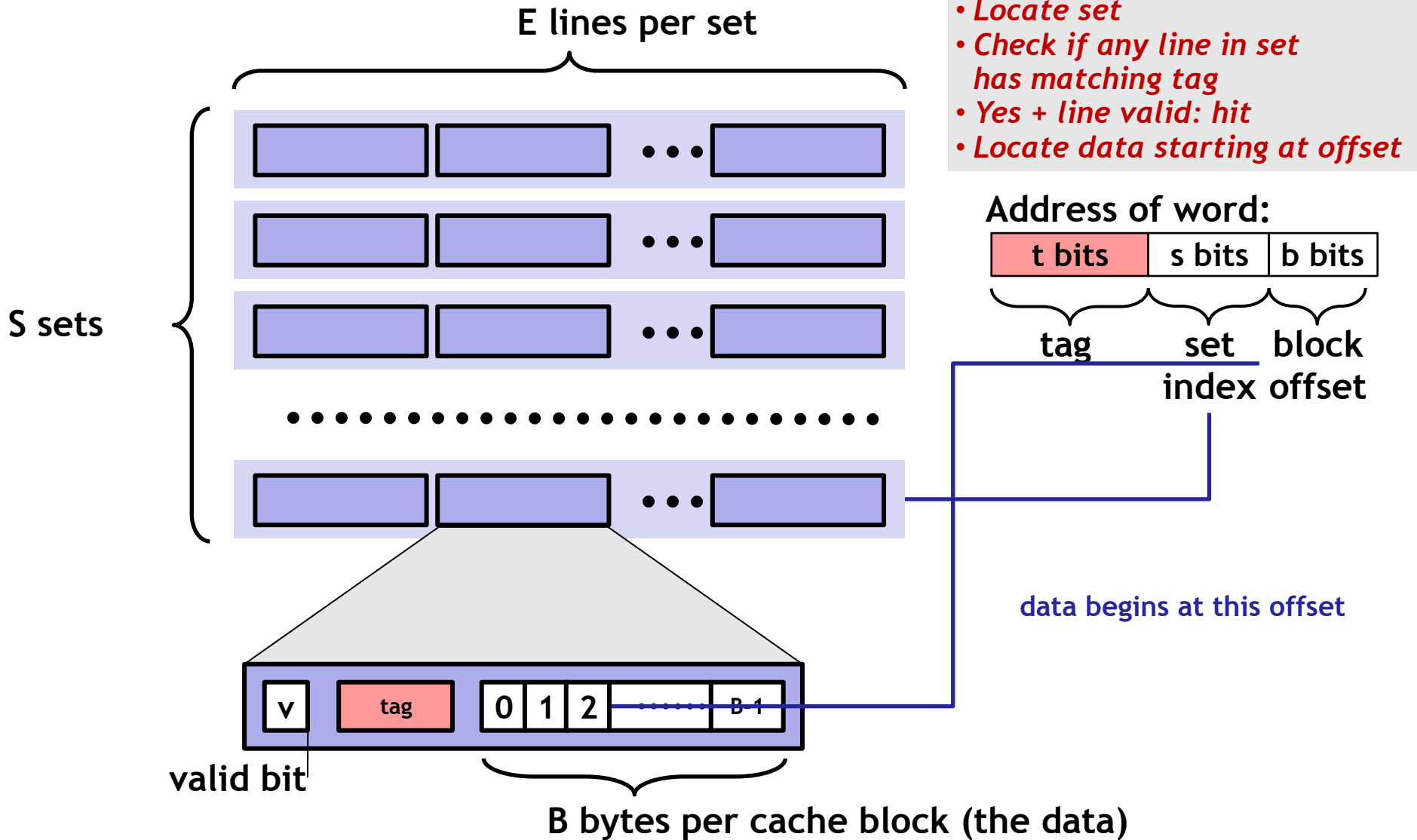
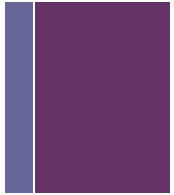


- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*

Address of word:



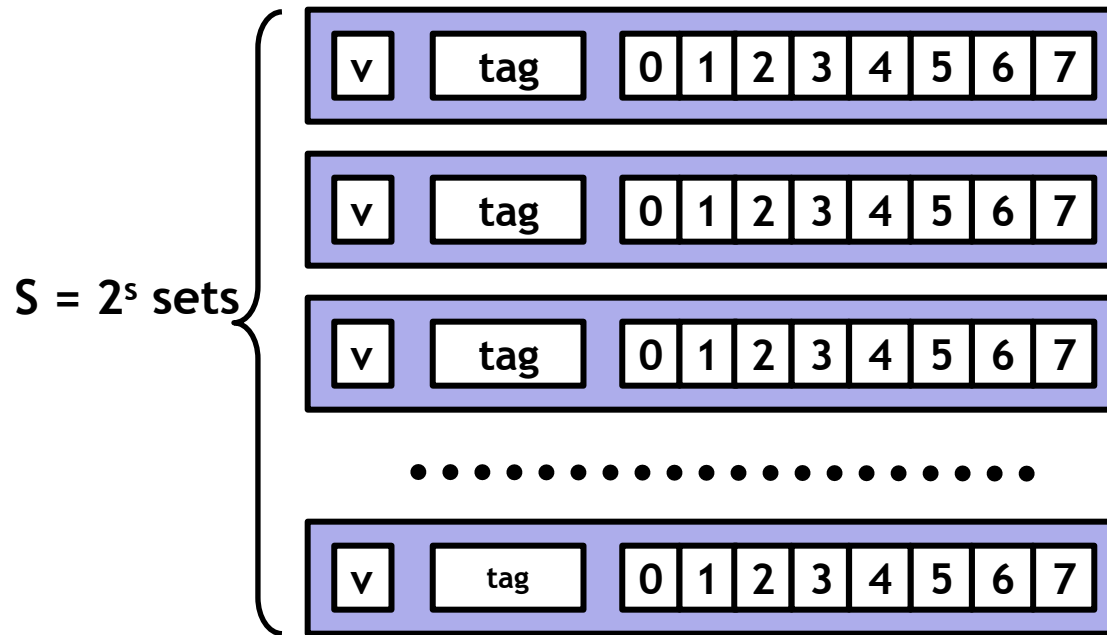
+ Cache Read



+ Direct Mapped Cache ($E = 1$)



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



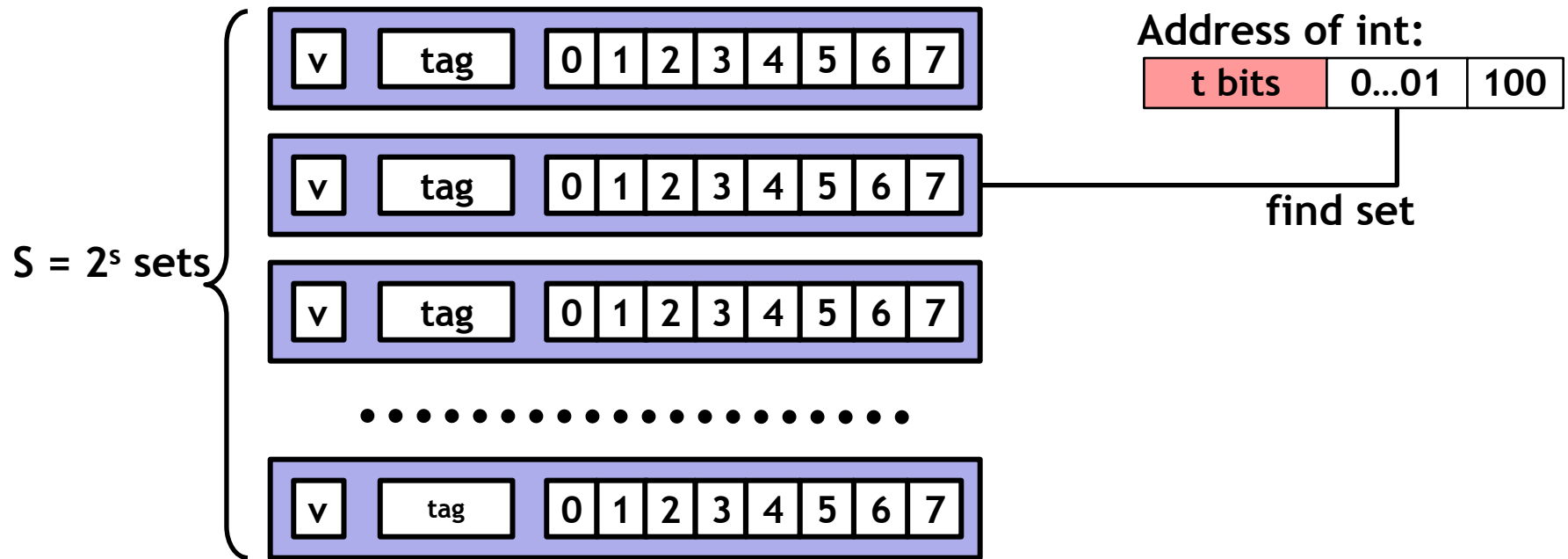
Address of int:



+ Direct Mapped Cache ($E = 1$)



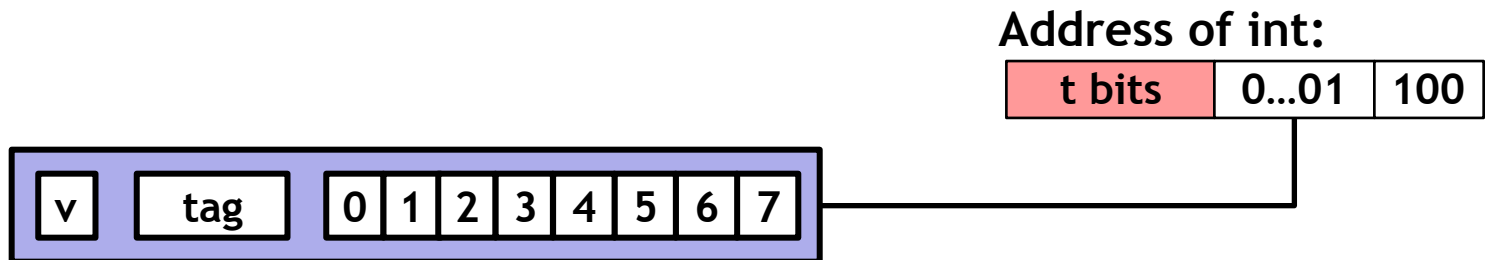
Direct mapped: One line per set
Assume: cache block size 8 bytes



+ Direct Mapped Cache ($E = 1$)



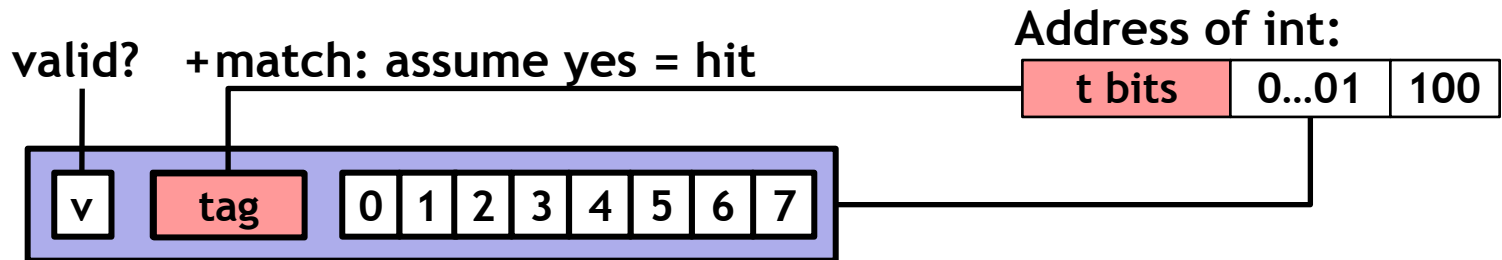
Direct mapped: One line per set
Assume: cache block size 8 bytes



+ Direct Mapped Cache ($E = 1$)



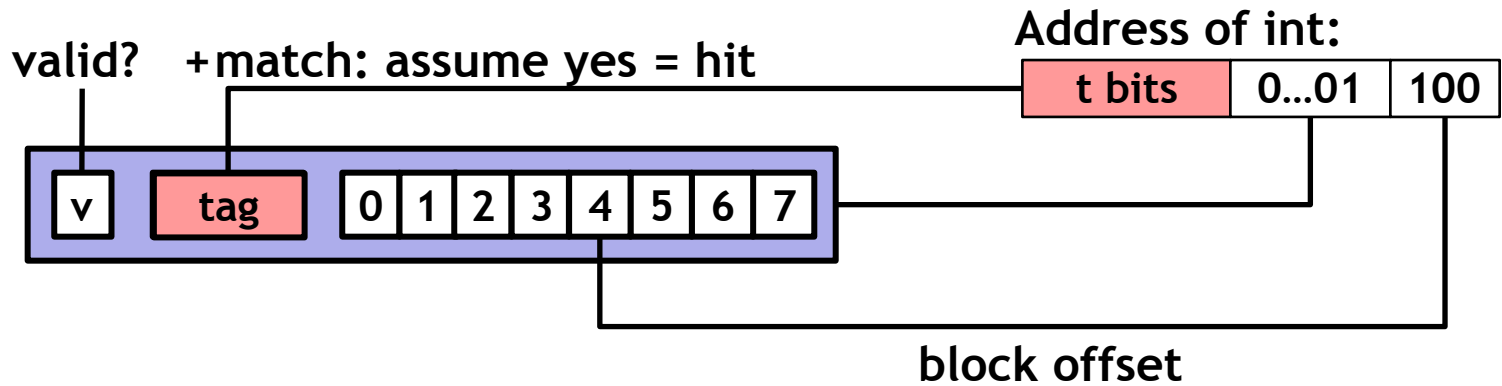
Direct mapped: One line per set
Assume: cache block size 8 bytes



+ Direct Mapped Cache ($E = 1$)



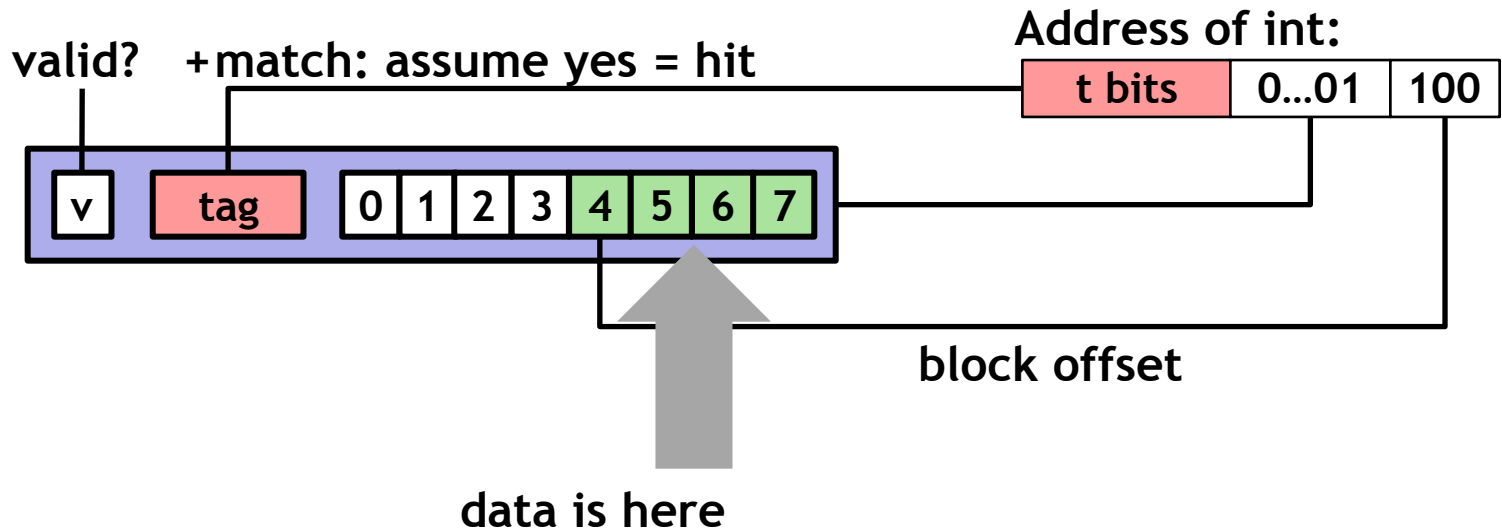
Direct mapped: One line per set
Assume: cache block size 8 bytes



+ Direct Mapped Cache ($E = 1$)



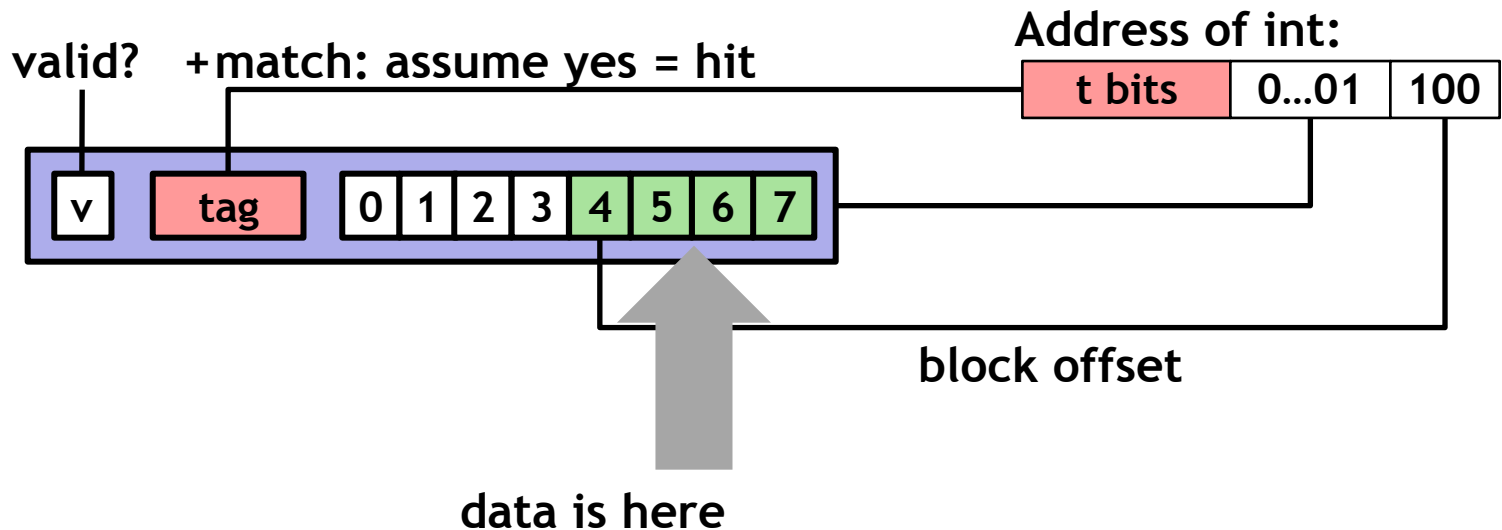
Direct mapped: One line per set
Assume: cache block size 8 bytes



+ Direct Mapped Cache ($E = 1$)



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



If tag doesn't match: old line is evicted and replaced

+ Direct-Mapped Cache Simulation



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

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0 [0000₂],
1 [0001₂],
7 [0111₂],
8 [1000₂],
0 [0000₂]

	v	Tag	Block
Set 0	0	?	?
Set 1			
Set 2			
Set 3			

+ Direct-Mapped Cache Simulation



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

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	
7	[<u>0111</u> ₂],	
8	[<u>1000</u> ₂],	
0	[<u>0000</u> ₂]	

	v	Tag	Block
Set 0	0	?	?
Set 1			
Set 2			
Set 3			

+ Direct-Mapped Cache Simulation



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

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	
7	[<u>0111</u> ₂],	
8	[<u>1000</u> ₂],	
0	[<u>0000</u> ₂]	

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

+ Direct-Mapped Cache Simulation



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

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	
8	[<u>1000</u> ₂],	
0	[<u>0000</u> ₂]	

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

+ Direct-Mapped Cache Simulation



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

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	
0	[<u>0000</u> ₂]	

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

+ Direct-Mapped Cache Simulation



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

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	
0	[<u>0000</u> ₂]	

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

+ Direct-Mapped Cache Simulation



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

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	miss
0	[<u>0000</u> ₂]	

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

+ Direct-Mapped Cache Simulation



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

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	miss
0	[<u>0000</u> ₂]	

	v	Tag	Block
Set 0	1	1	M[8-9]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

+ Direct-Mapped Cache Simulation



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

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	miss
0	[<u>0000</u> ₂]	miss

	v	Tag	Block
Set 0	1	1	M[8-9]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

+ Direct-Mapped Cache Simulation



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

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	[<u>0000</u> ₂],	miss
1	[<u>0001</u> ₂],	hit
7	[<u>0111</u> ₂],	miss
8	[<u>1000</u> ₂],	miss
0	[<u>0000</u> ₂]	miss

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

+ E-way Set Associative Cache (E = 2)

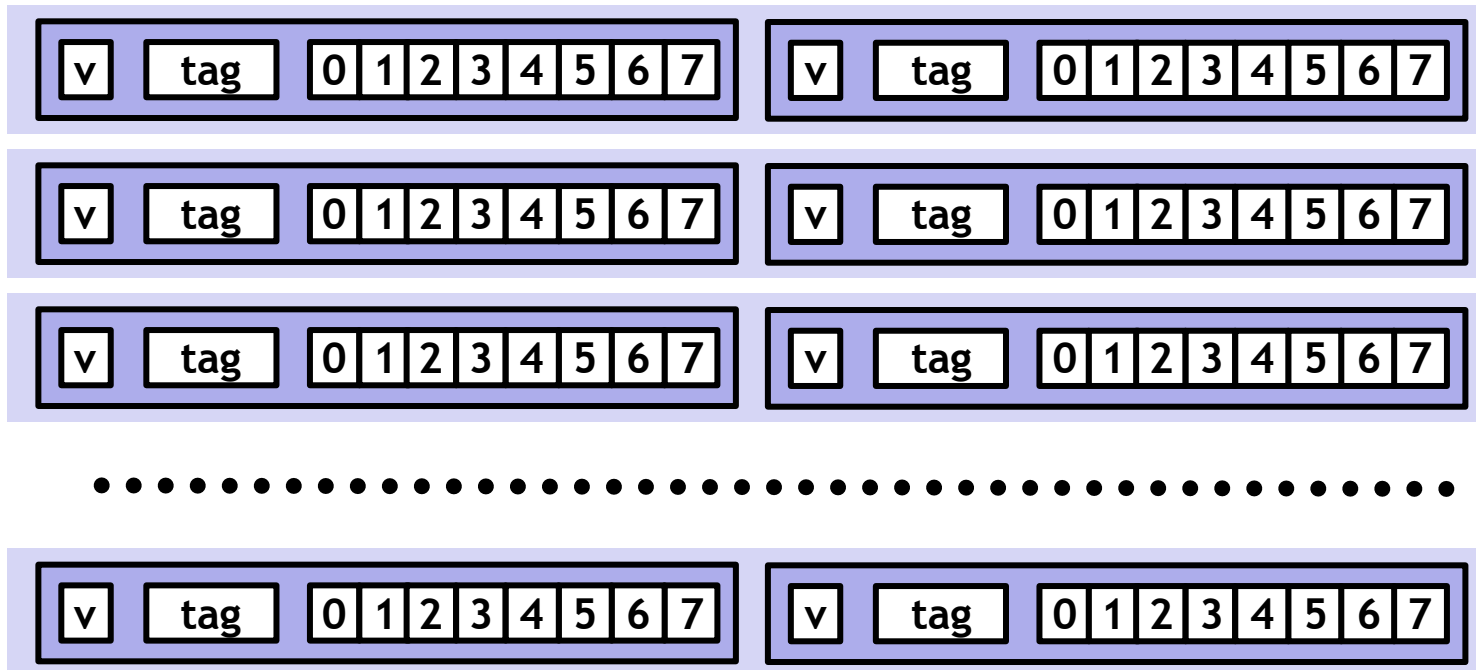


E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:

t bits	0...01	100
--------	--------	-----



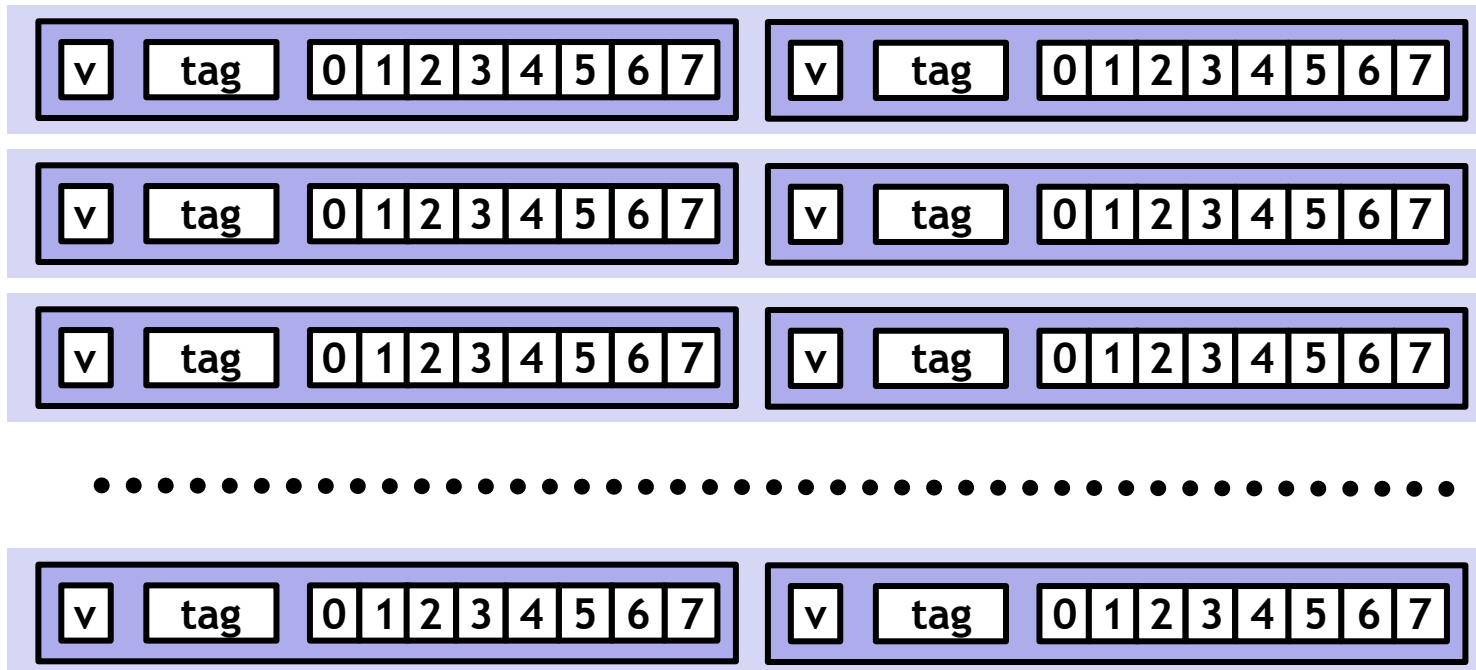
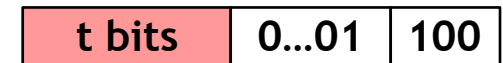
+ E-way Set Associative Cache (E = 2)



E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:



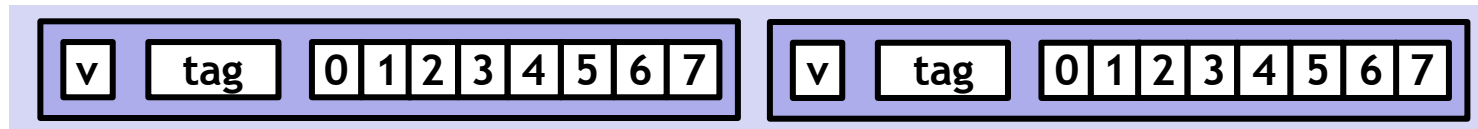
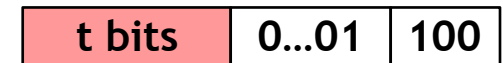
+ E-way Set Associative Cache (E = 2)



E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:

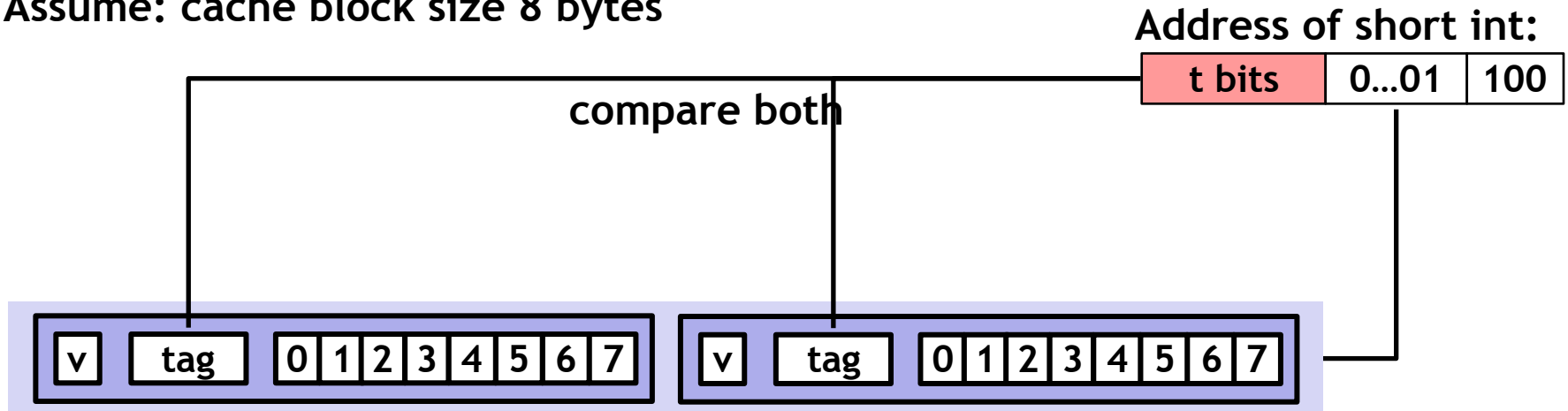


+ E-way Set Associative Cache (E = 2)



E = 2: Two lines per set

Assume: cache block size 8 bytes

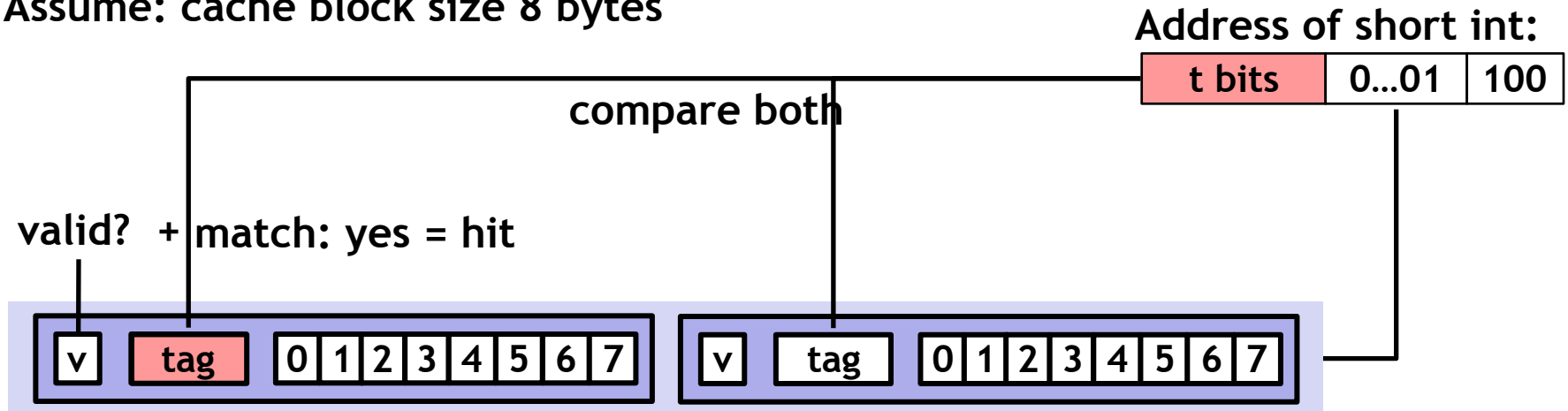


+ E-way Set Associative Cache (E = 2)



E = 2: Two lines per set

Assume: cache block size 8 bytes

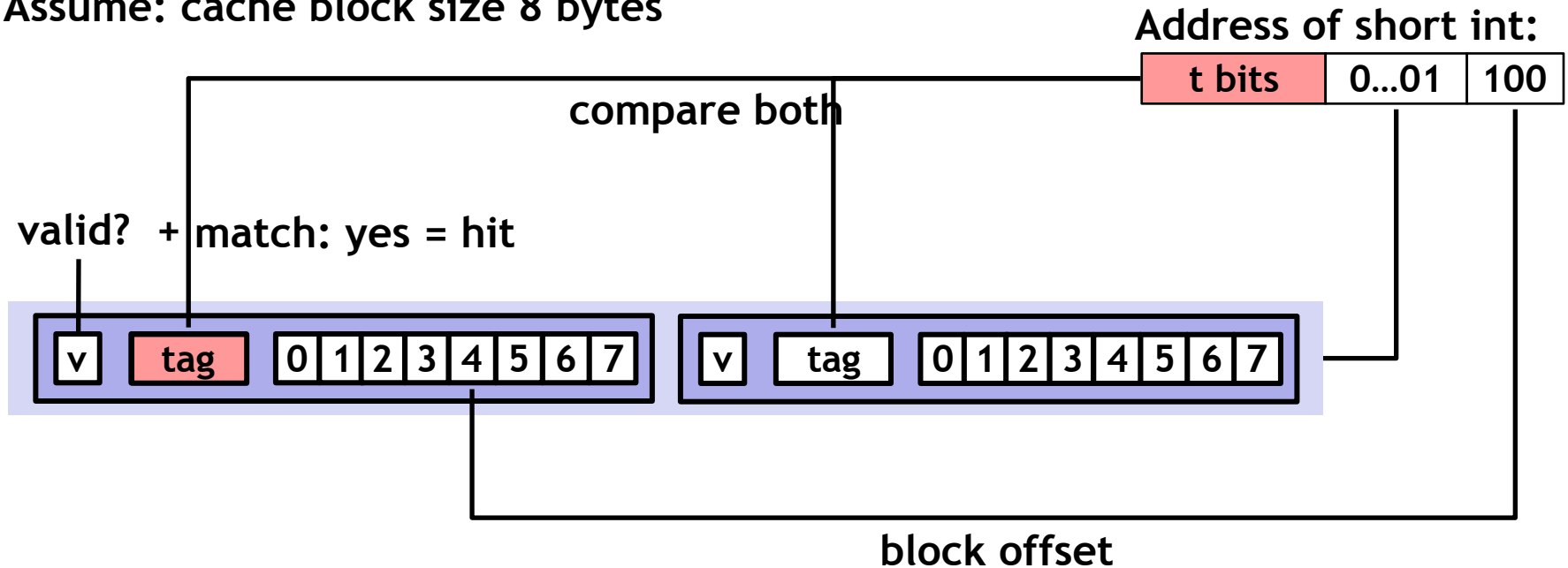


+ E-way Set Associative Cache (E = 2)



E = 2: Two lines per set

Assume: cache block size 8 bytes

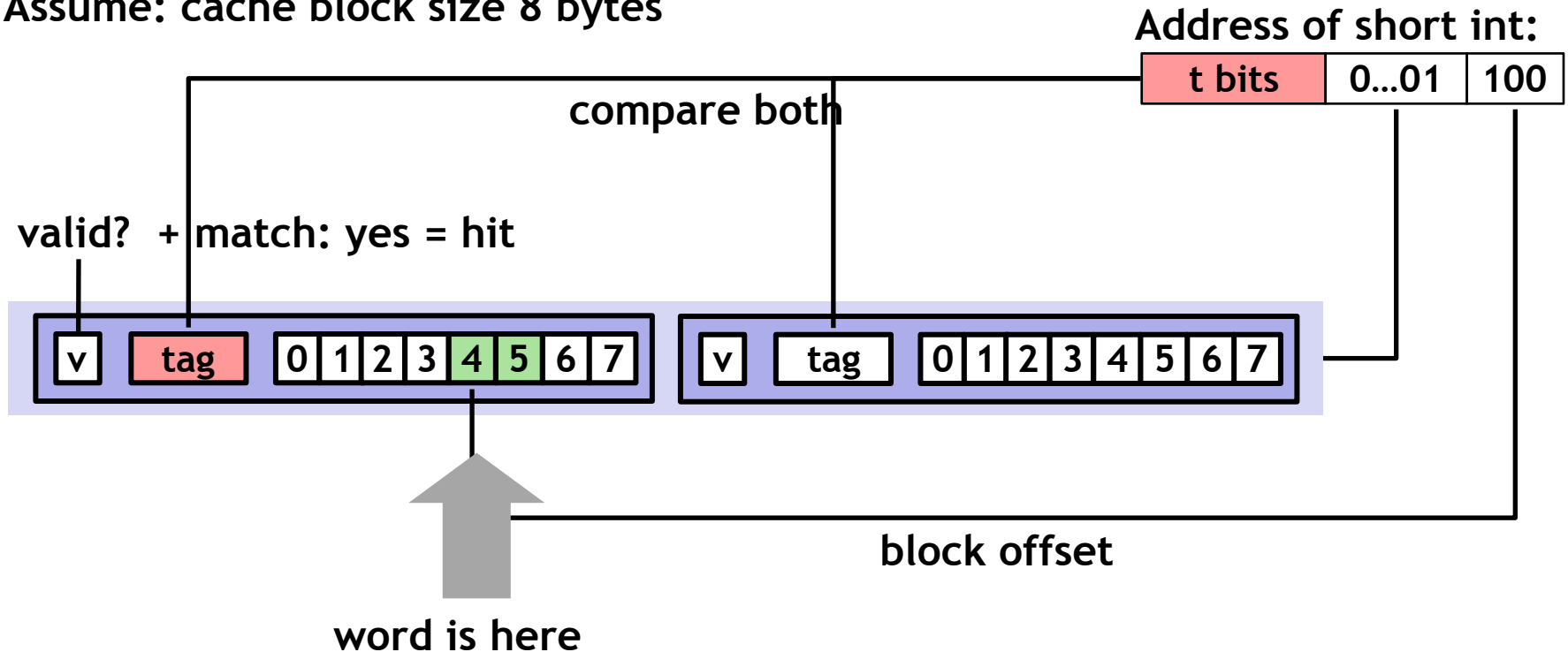


+ E-way Set Associative Cache (E = 2)



E = 2: Two lines per set

Assume: cache block size 8 bytes

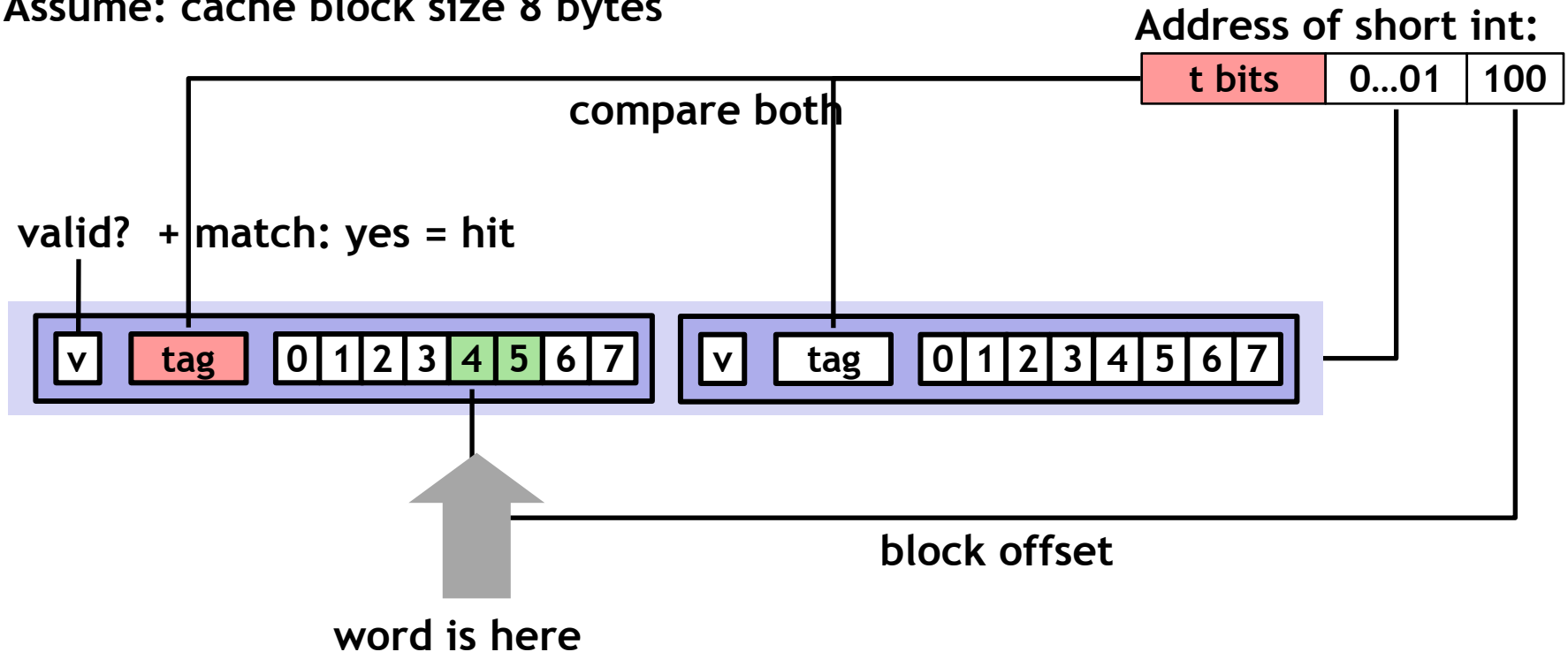


+ E-way Set Associative Cache (E = 2)



E = 2: Two lines per set

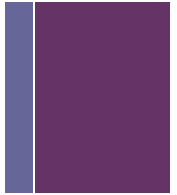
Assume: cache block size 8 bytes



No match:

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

+ 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 [0000₂],
1 [0001₂],
7 [0111₂],
8 [1000₂],
0 [0000₂]

	v	Tag	Block
Set 0	0	?	?
	0		
Set 1	0		
	0		

+ 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 [0000₂], miss
1 [0001₂],
7 [0111₂],
8 [1000₂],
0 [0000₂]

	v	Tag	Block
Set 0	0	?	?
	0		
Set 1	0		
	0		

+ 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 [0000₂], miss
1 [0001₂],
7 [0111₂],
8 [1000₂],
0 [0000₂]

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	0		
	0		

+ 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	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[01 <u>1</u> 1 ₂],	
8	[10 <u>0</u> 0 ₂],	
0	[00 <u>0</u> 0 ₂]	

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	0		
	0		

+ 2-Way Set Associative Cache Simulation



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

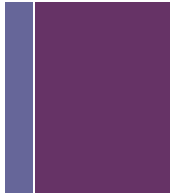
M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[01 <u>1</u> 1 ₂],	miss
8	[10 <u>0</u> 0 ₂],	
0	[00 <u>0</u> 0 ₂]	

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	0		
	0		

+ 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	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[01 <u>1</u> 1 ₂],	miss
8	[10 <u>0</u> 0 ₂],	
0	[00 <u>0</u> 0 ₂]	

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

+ 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	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[01 <u>1</u> 1 ₂],	miss
8	[10 <u>0</u> 0 ₂],	miss
0	[00 <u>0</u> 0 ₂]	

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

+ 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	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[01 <u>1</u> 1 ₂],	miss
8	[10 <u>0</u> 0 ₂],	miss
0	[00 <u>0</u> 0 ₂]	

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

+ 2-Way Set Associative Cache Simulation



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

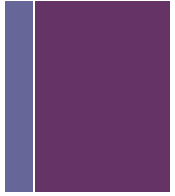
M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	[00 <u>0</u> 0 ₂],	miss
1	[00 <u>0</u> 1 ₂],	hit
7	[01 <u>1</u> 1 ₂],	miss
8	[10 <u>0</u> 0 ₂],	miss
0	[00 <u>0</u> 0 ₂]	hit

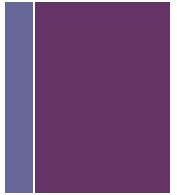
	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

+ What about writes?



- **Multiple copies of data exist:**
 - L1, L2, L3, Main Memory, Disk
- **What to do on a write-hit?**
 - *Write-through* (write immediately to memory, expensive)
 - *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 straight to memory, does not load into cache)
- **Typical**
 - Write-back + Write-allocate (i.e. update cache until eviction)

+ Cache Performance Metrics



- **Miss Rate**

- Fraction of memory references not found in cache (misses / accesses) = $1 - \text{hit rate}$
- Typical numbers:
 - 3-10% for L1
 - $< 1\%$ for L2, depending on size, etc.

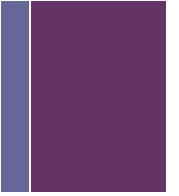
- **Hit Time**

- Time to deliver a line in the cache to the processor
 - includes time to determine whether the line is in the cache
- Typical numbers:
 - 4 clock cycle for L1
 - 10 clock cycles for L2

- **Miss Penalty**

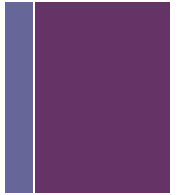
- Additional time required because of a miss
 - typically 50-200 cycles for main memory

+ Let's think about those numbers



- **Huge difference between a hit and a miss**
- **Would you believe 99% hits is twice as good as 97%?**
 - Consider:
 - cache hit time of 1 cycle
 - miss penalty of 100 cycles
 - Average access time:

+ Let's think about those numbers



- **Huge difference between a hit and a miss**
- **Would you believe 99% hits is twice as good as 97%?**
 - Consider:
cache hit time of 1 cycle
miss penalty of 100 cycles
 - Average access time:
97% hits: $1 \text{ cycle} + 0.03 * 100 \text{ cycles} = \mathbf{4 \text{ cycles}}$
99% hits: $1 \text{ cycle} + 0.01 * 100 \text{ cycles} = \mathbf{2 \text{ cycles}}$
- **This is why “miss rate” is used instead of “hit rate”**



Writing Cache Friendly Code

+ Matrix Multiplication Example



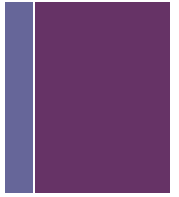
■ Description:

- Multiply $N \times N$ matrices
- Matrix elements are doubles (8 bytes)
- $O(N^3)$ total operations
- N reads per source element
- N values summed per destination

```
/* 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;
    }
}
```

Variable *sum*
held in register

+ Miss Rate Analysis for Matrix Multiply

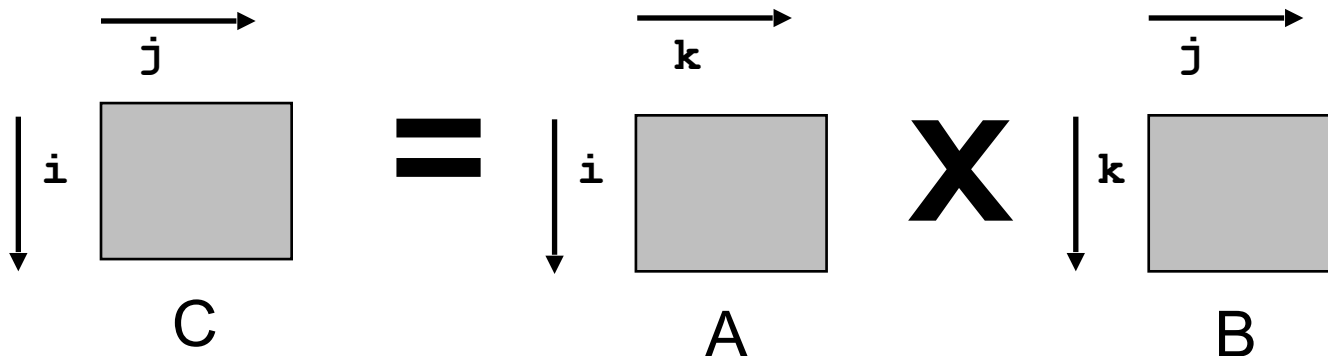


- **Assume:**

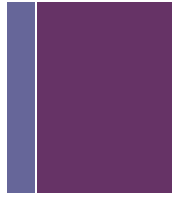
- Block size = 32B (big enough for 4 doubles)
- Matrix dimension (N) is “very large”
- Cache is not even big enough to hold multiple rows

- **Analysis Method:**

- Look at access pattern of inner loop



+ Layout of C Arrays in Memory (review)



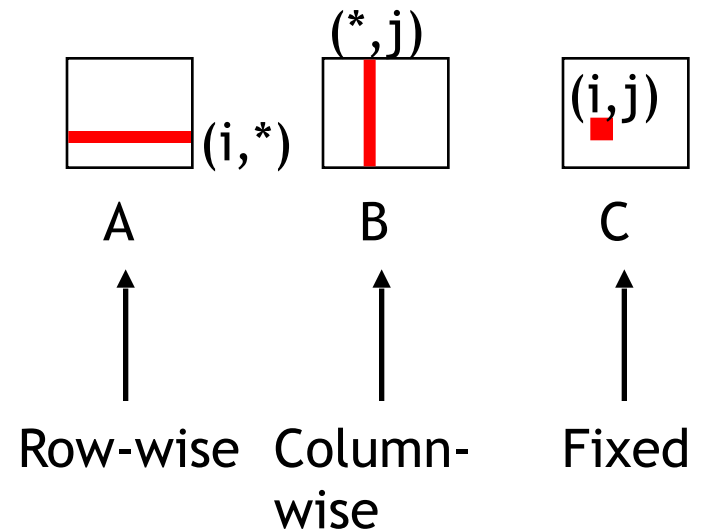
- **C arrays allocated in row-major order**
 - each row in contiguous memory locations
- **Stepping through columns in one row:**
 - for ($i = 0; i < N; i++$)
 $\text{sum} += a[0][i];$
 - accesses successive, contiguous elements
 - if block size $> \text{sizeof}(a_{ij})$ bytes, exploit spatial locality
 - miss rate = $\text{sizeof}(a_{ij}) / \text{block size}$
- **Stepping through rows in one column:**
 - for ($i = 0; i < n; i++$)
 $\text{sum} += a[i][0];$
 - accesses distant elements (stride-rowsize pattern)
 - no spatial locality!
 - 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;
    }
}
```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

double[2][2]

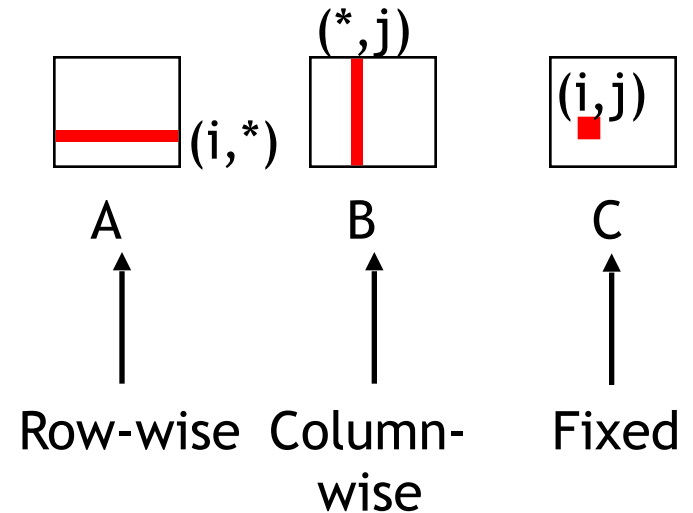
```
sum += A[0][0] * B[0][0]
sum += A[0][1] * B[1][0]
sum += A[0][0] * B[0][1]
sum += A[0][1] * B[1][1]
sum += A[1][0] * B[0][0]
sum += A[1][1] * B[1][0]
sum += A[1][0] * B[0][1]
sum += A[1][1] * B[1][1]
```

+ 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
    }
}
```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

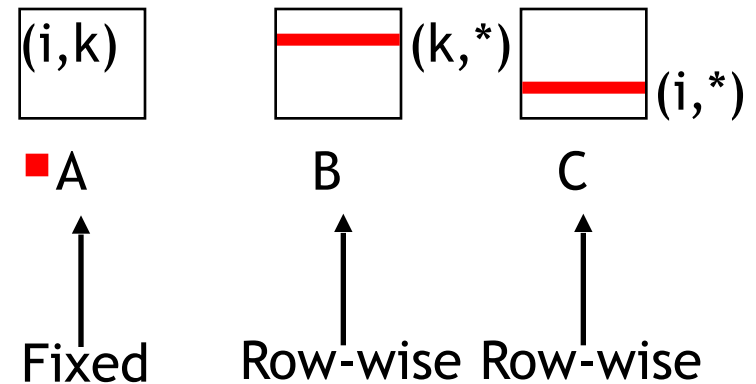
Effectively same as ijk

+ 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];
    }
}
```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

double[2][2]

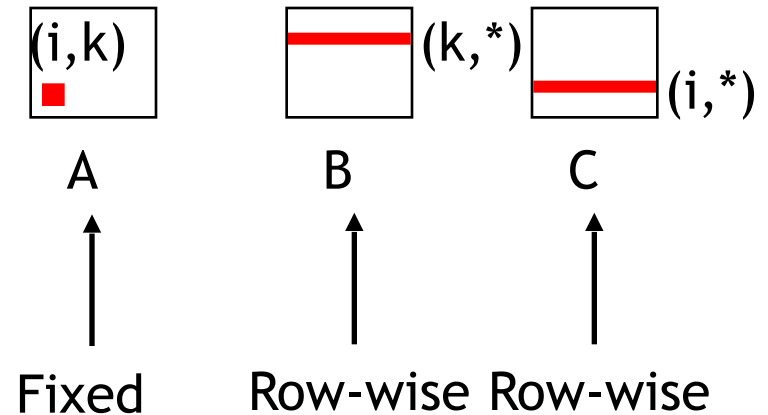
```
C[0][0] += r * B[0][0]
C[0][1] += r * B[0][1]
C[1][0] += r * B[0][0]
C[1][1] += r * B[0][1]
C[0][0] += r * B[1][0]
C[0][1] += r * B[1][1]
C[1][0] += r * B[1][0]
C[1][1] += r * B[1][1]
```

+ 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];
    }
}
```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

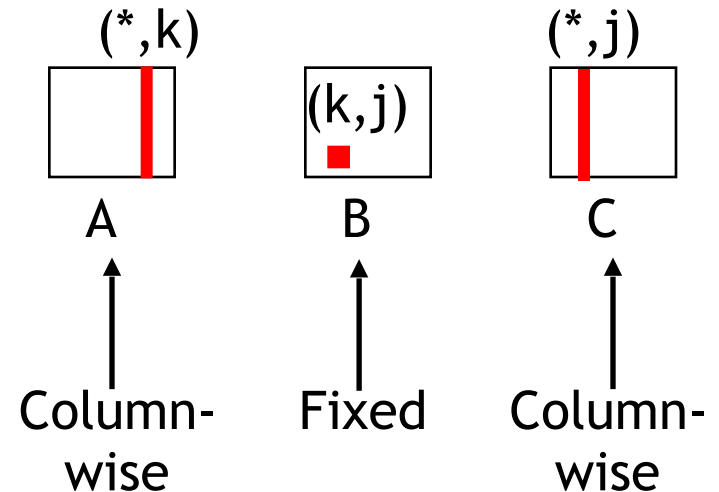
Effectively same as kij

+ 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;
    }
}
```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

double[2][2]

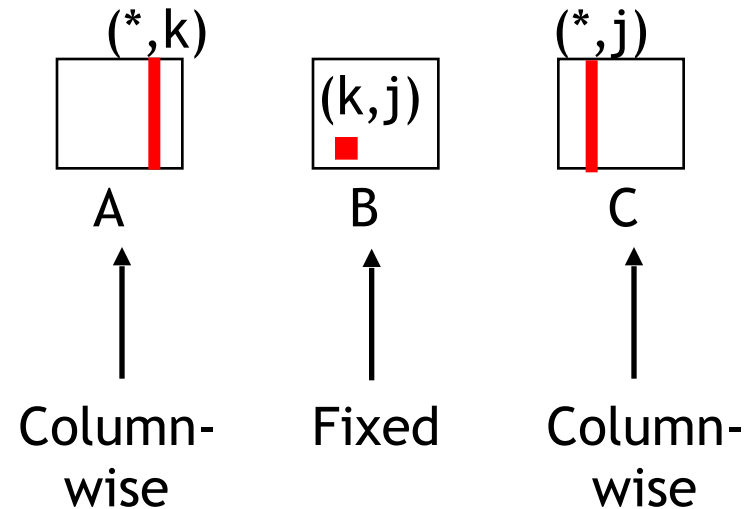
```
C[0][0] += A[0][0] * r
C[1][0] += A[1][0] * r
C[0][1] += A[0][1] * r
C[1][1] += A[1][1] * r
```

+ 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;
    }
}
```

Inner loop:

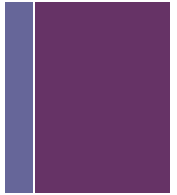


Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Effectively same as jki

+ 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 = **1.25**

```
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):

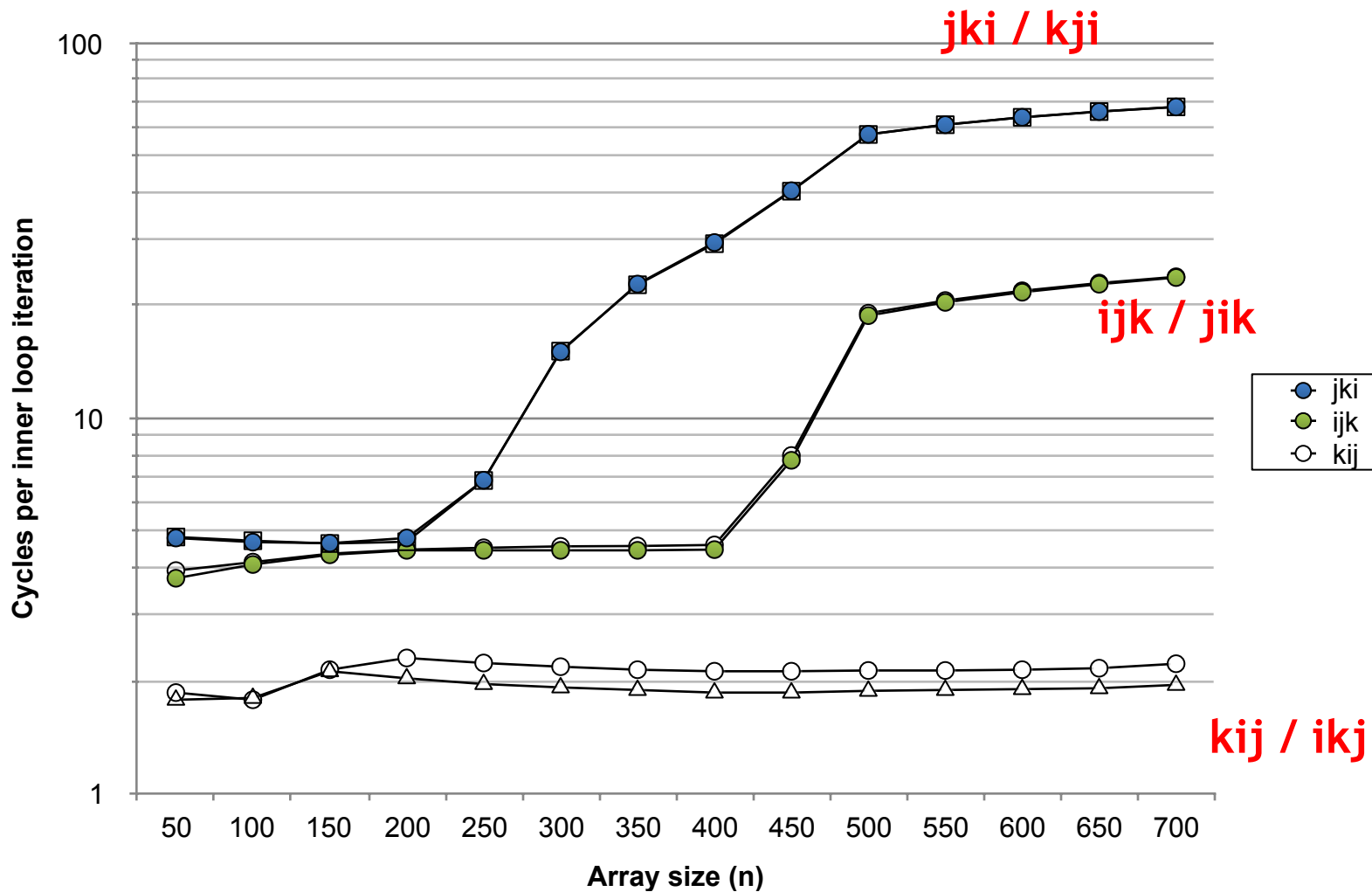
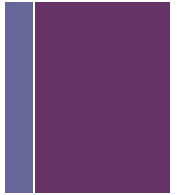
- 2 loads, 1 store
- misses/iter = **0.5**

```
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;  
    }  
}
```

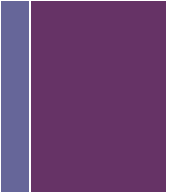
jki (& kji):

- 2 loads, 1 store
- misses/iter = **2.0**

+ Matrix Multiply Performance



+ Cache Summary



- **Cache memories can have significant performance impact**
- **You can write your programs to exploit this!**
 - Focus on the inner loops, where bulk of computations and memory accesses occur.
 - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
 - Will be next lab.