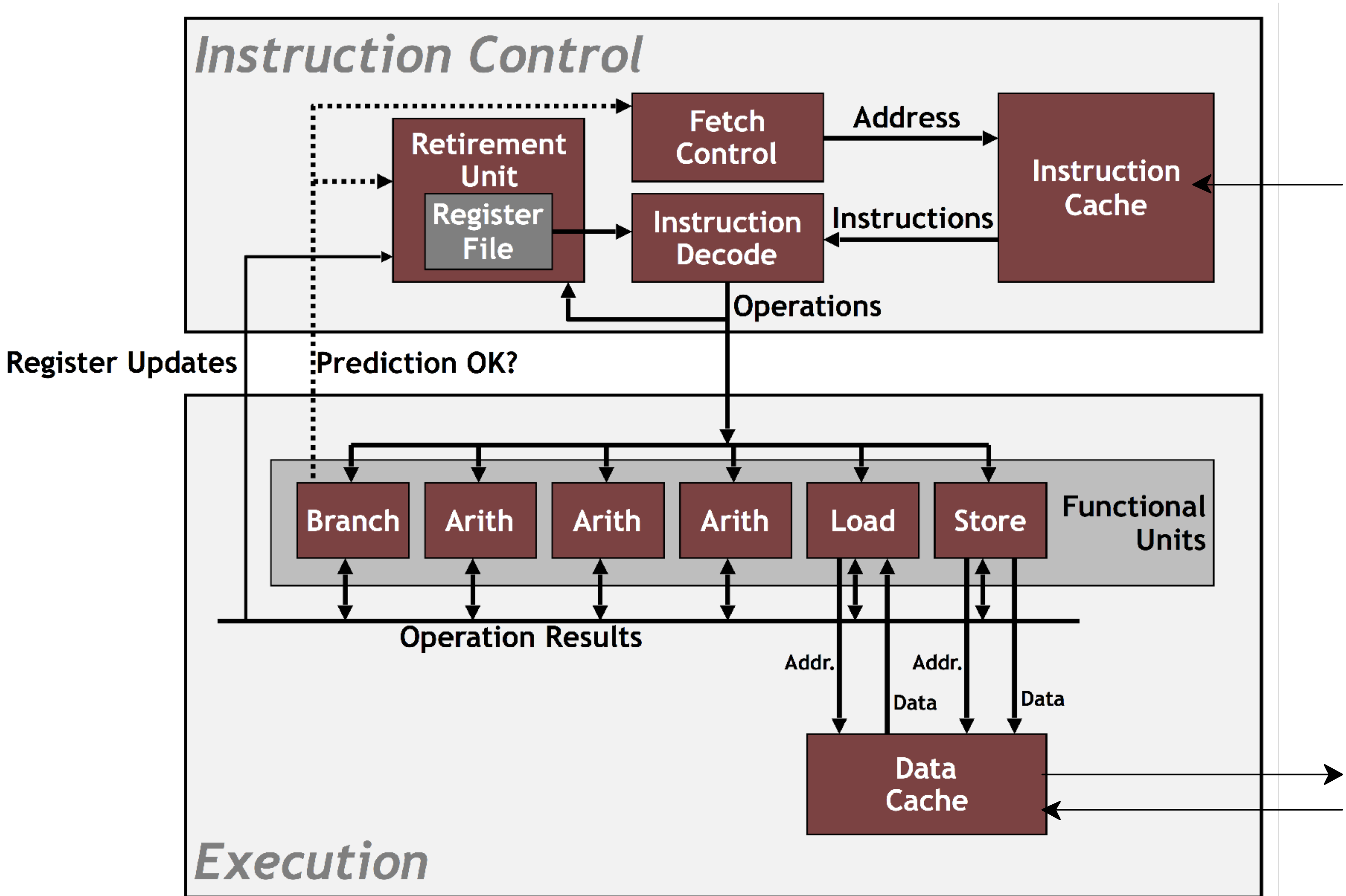
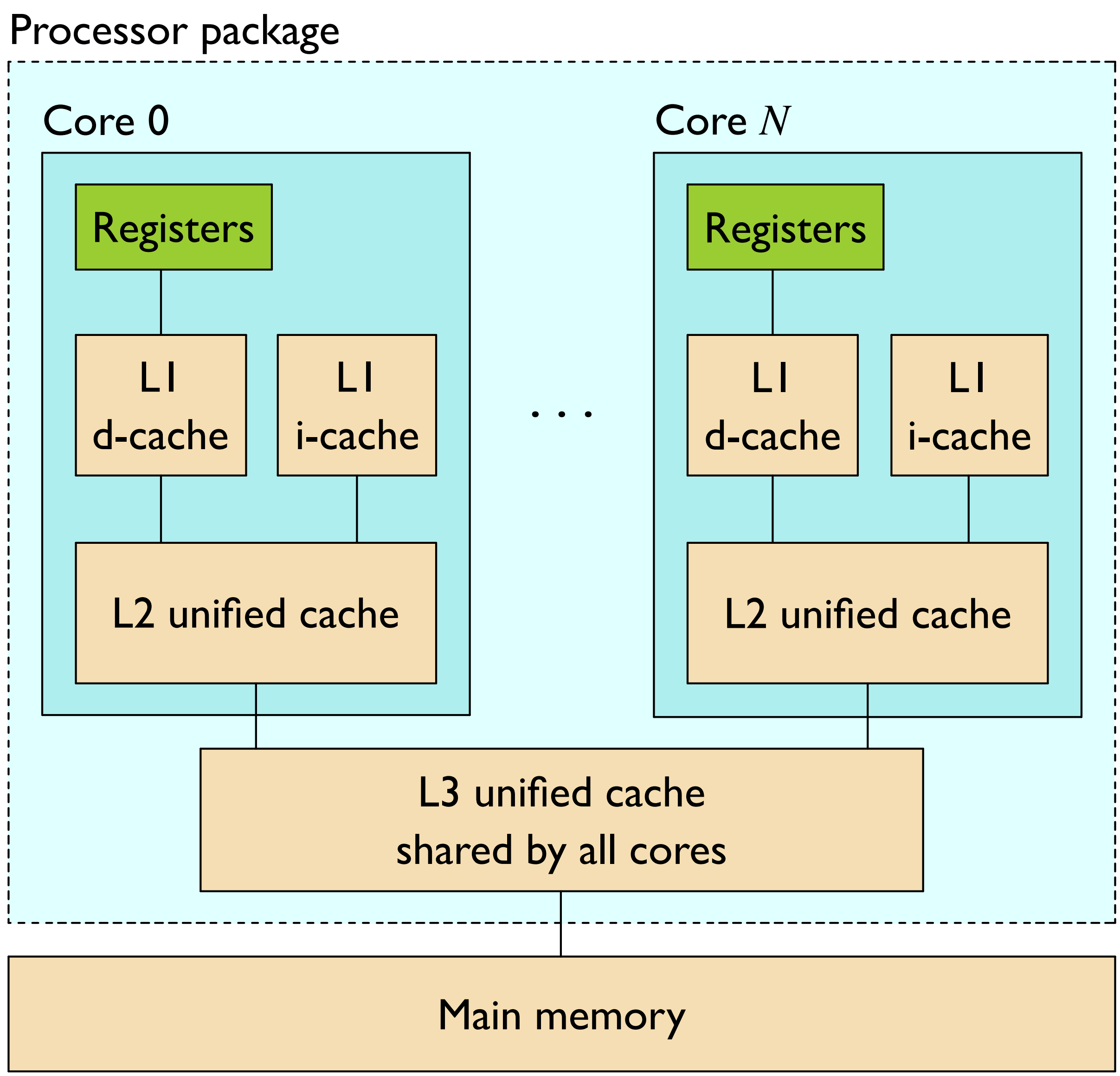
Modern CPU



Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Editio

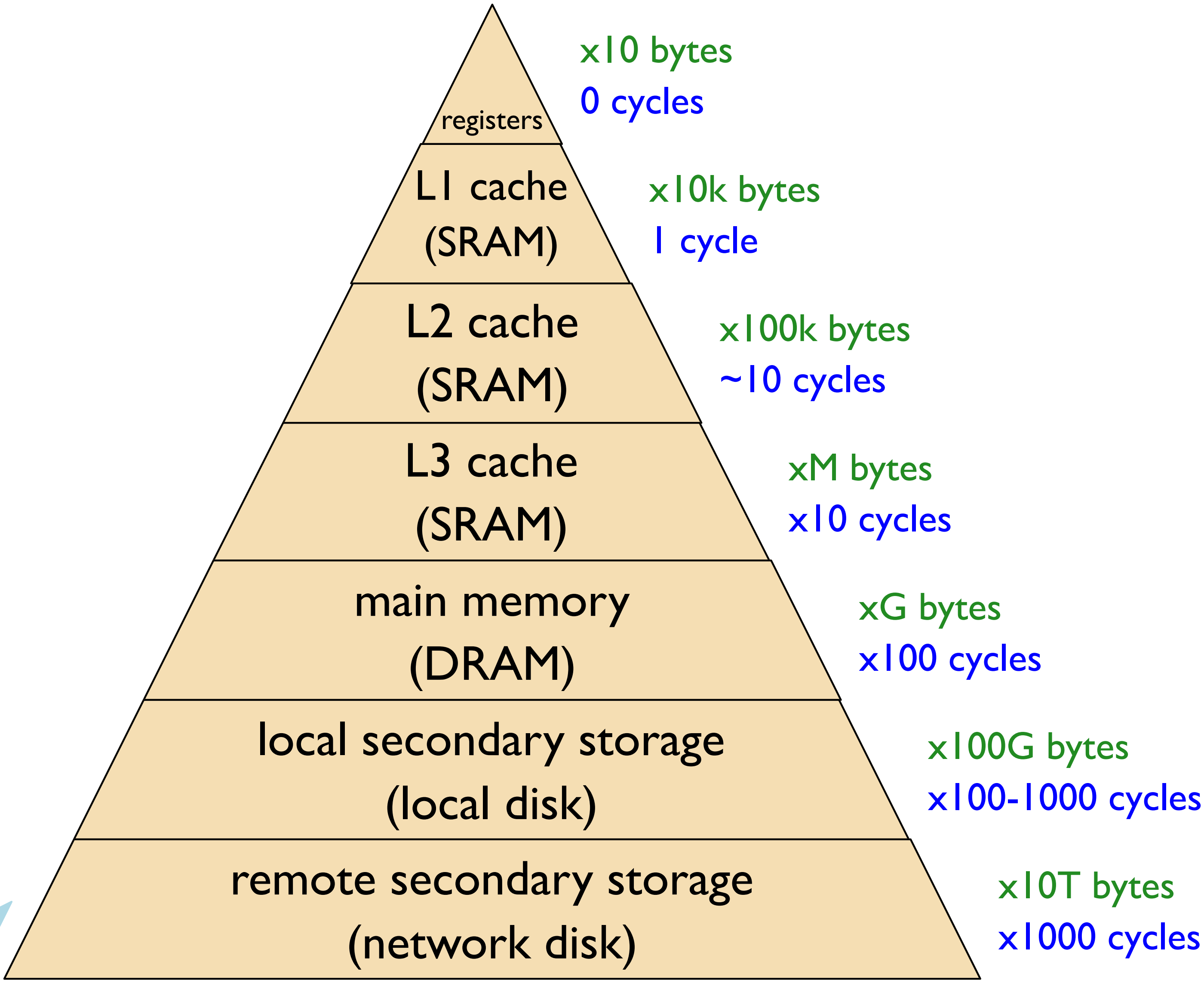
Cache Hierarchy



Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Memory Hierarchy

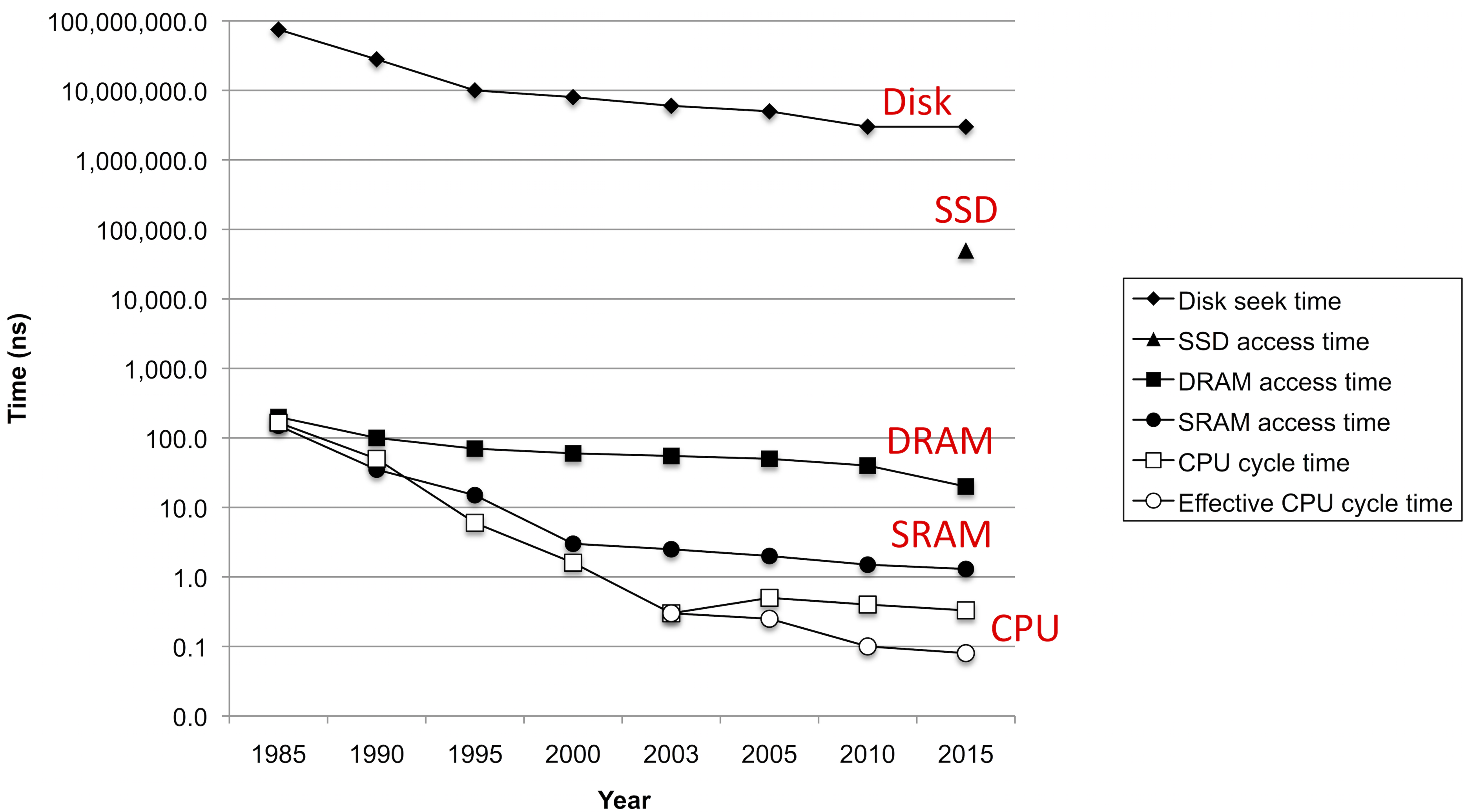
Smaller,
faster,
nearer to
processor,
costly per
byte



Bigger,
slower,
further from
processor,
cheaper per
byte

Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

The CPU–Memory Gap



Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Memory Access Times

2^{30} random accesses of an **`int`** spread across...

64B	0.8s
128B	0.8s
...	
32kB	0.8s
64kB	0.83s
128kB	1.2s
256kB	1.7s
512kB	1.9s
1MB	2.3s
2MB	2.5s
4MB	3.9s
8MB	7.2s
16MB	7.5s

Memory Access Times

2^{30} random accesses of an
`int` spread across...

64B	0.8s
128B	0.8s
...	
32kB	0.8s
64kB	0.83s
128kB	1.2s
256kB	1.7s
512kB	1.9s
1MB	2.3s
2MB	2.5s
4MB	3.9s
8MB	7.2s
16MB	7.5s

L1 cache: 32kB
4-5 cycles

L2 cache: 256kB
12 cycles

L3 cache: 3MB
36 cycles

Memory: 8GB
~100 cycles

Memory Access Times

2^{30} sequential accesses of an `int` spread across...

64B	0.8s
128B	0.8s
...	
32kB	0.8s
64kB	0.83s
128kB	0.88s
256kB	0.91s
512kB	0.91s
1MB	0.95s
2MB	0.95s
4MB	0.99s
8MB	1.07s
16MB	1.08s

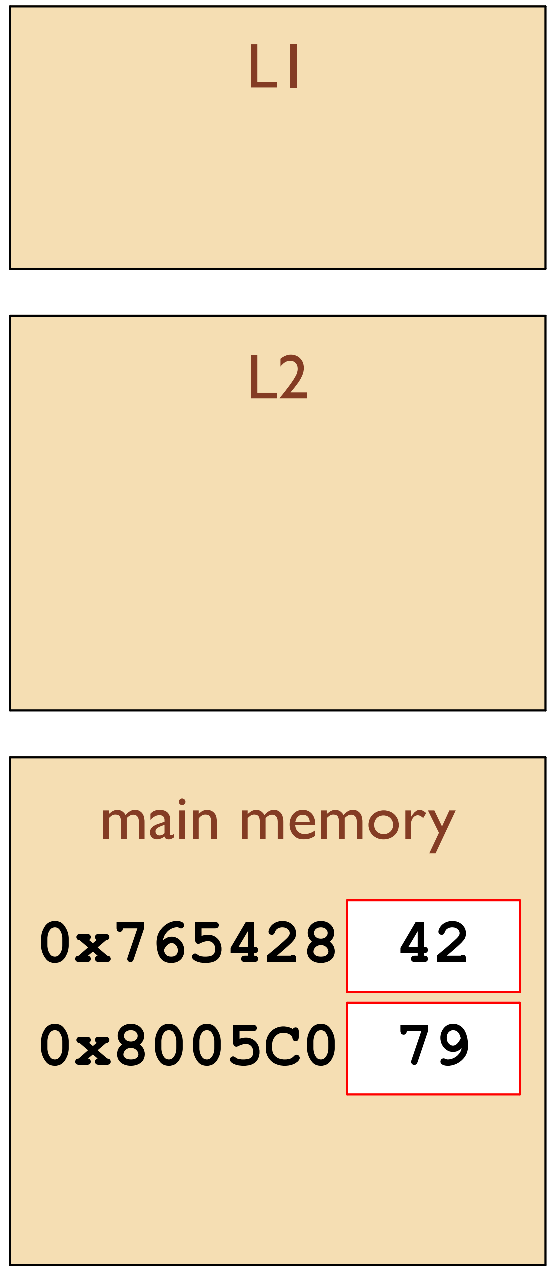
L1 cache: 32kB
4-5 cycles

L2 cache: 256kB
12 cycles

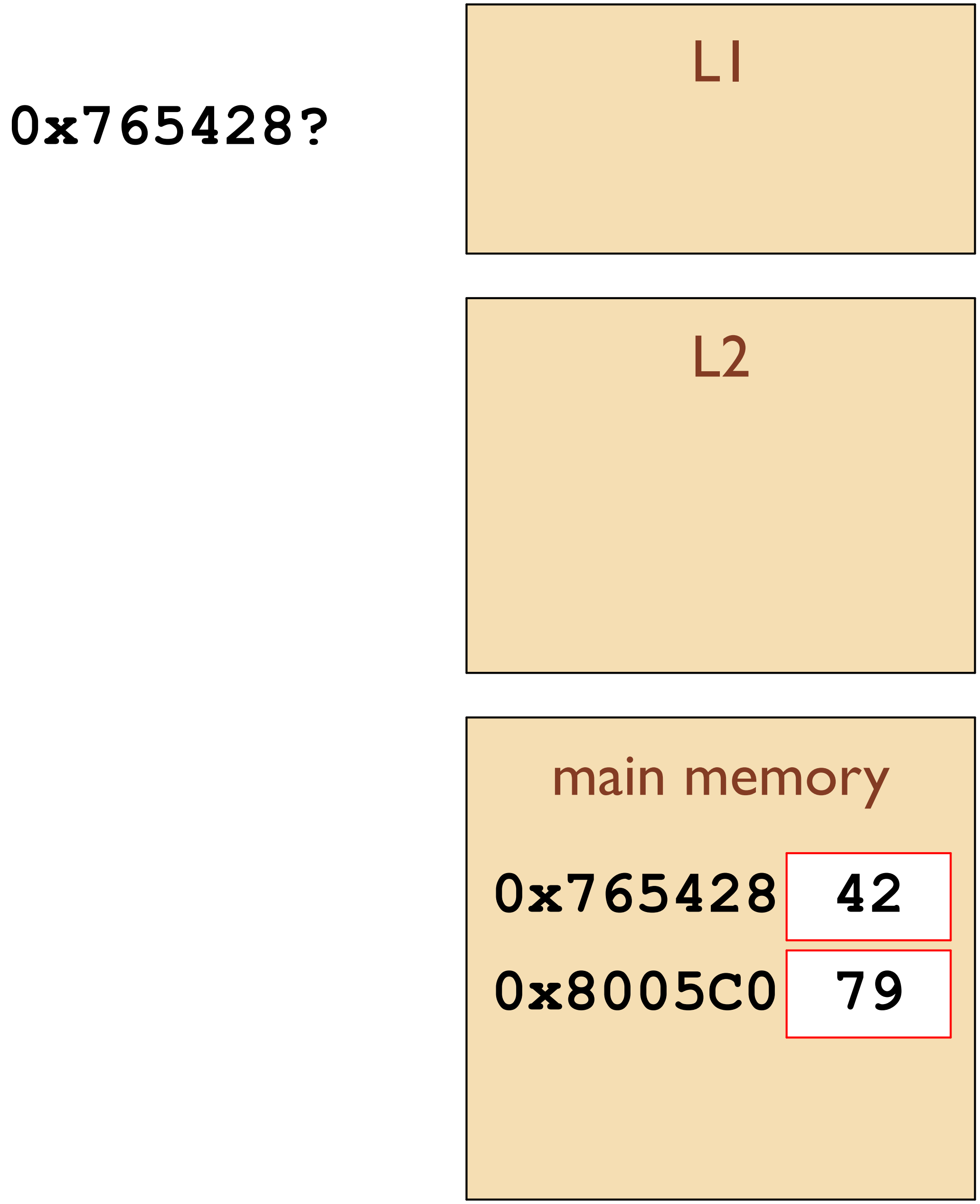
L3 cache: 3MB
36 cycles

Memory: 8GB
~100 cycles

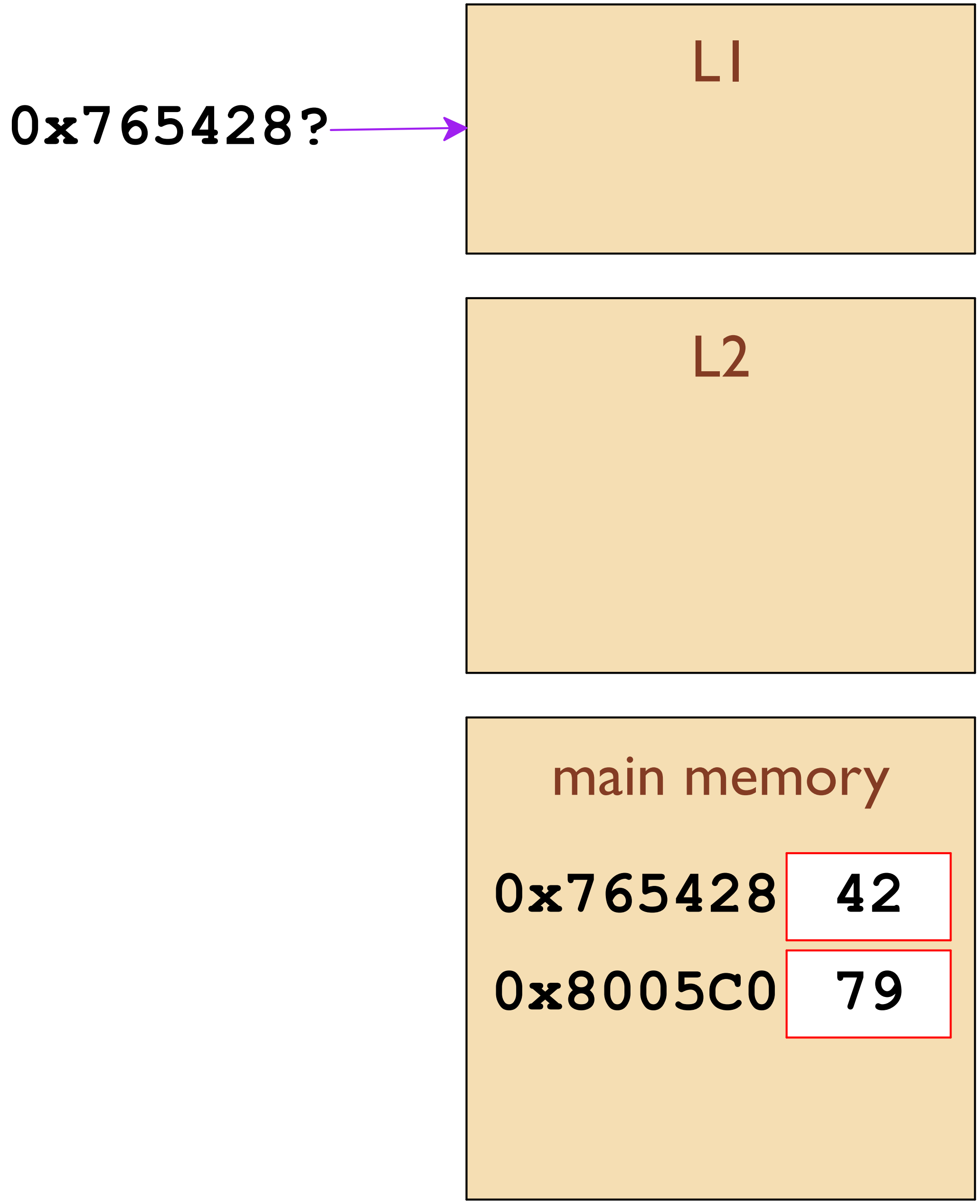
Cache Lookup



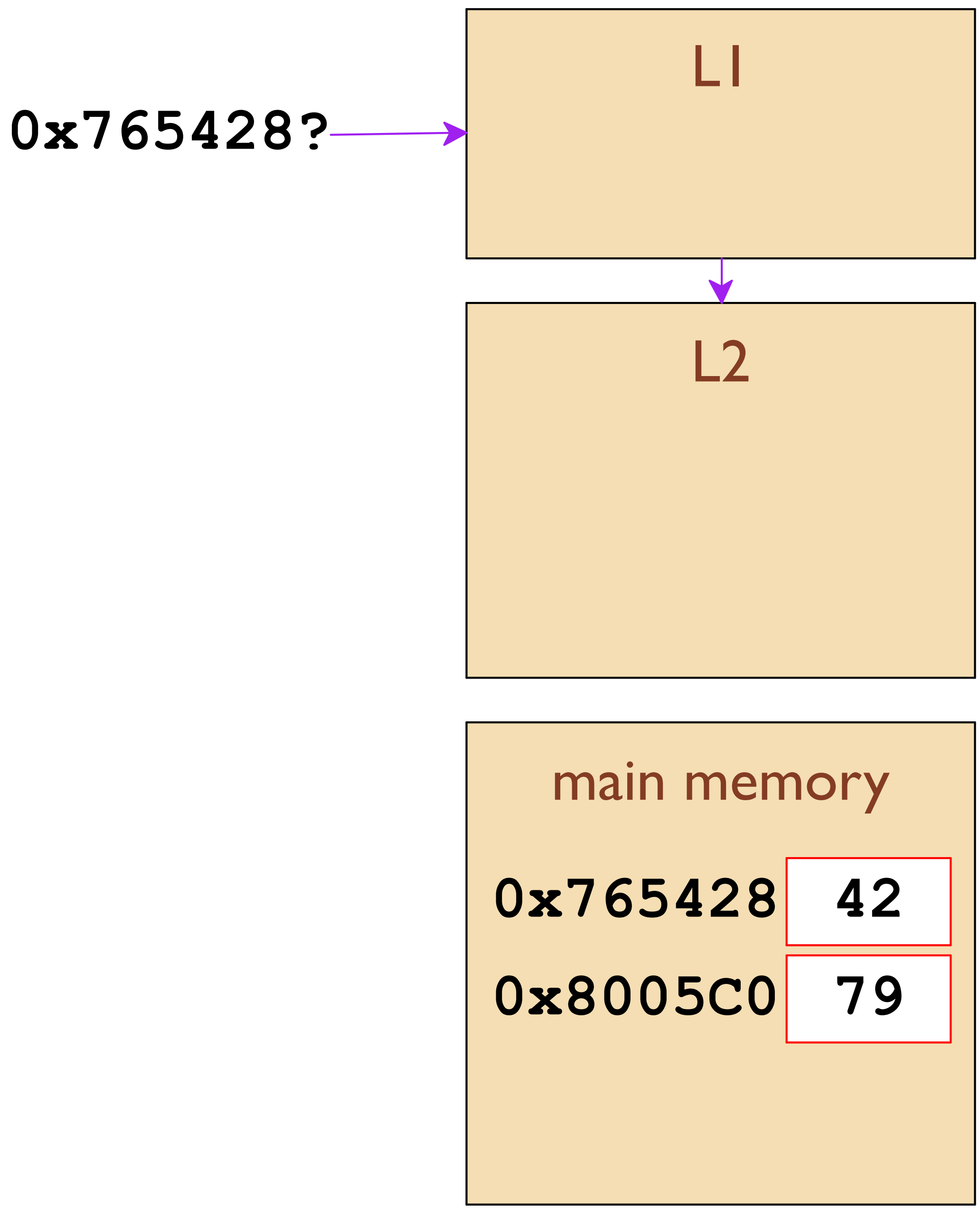
Cache Lookup



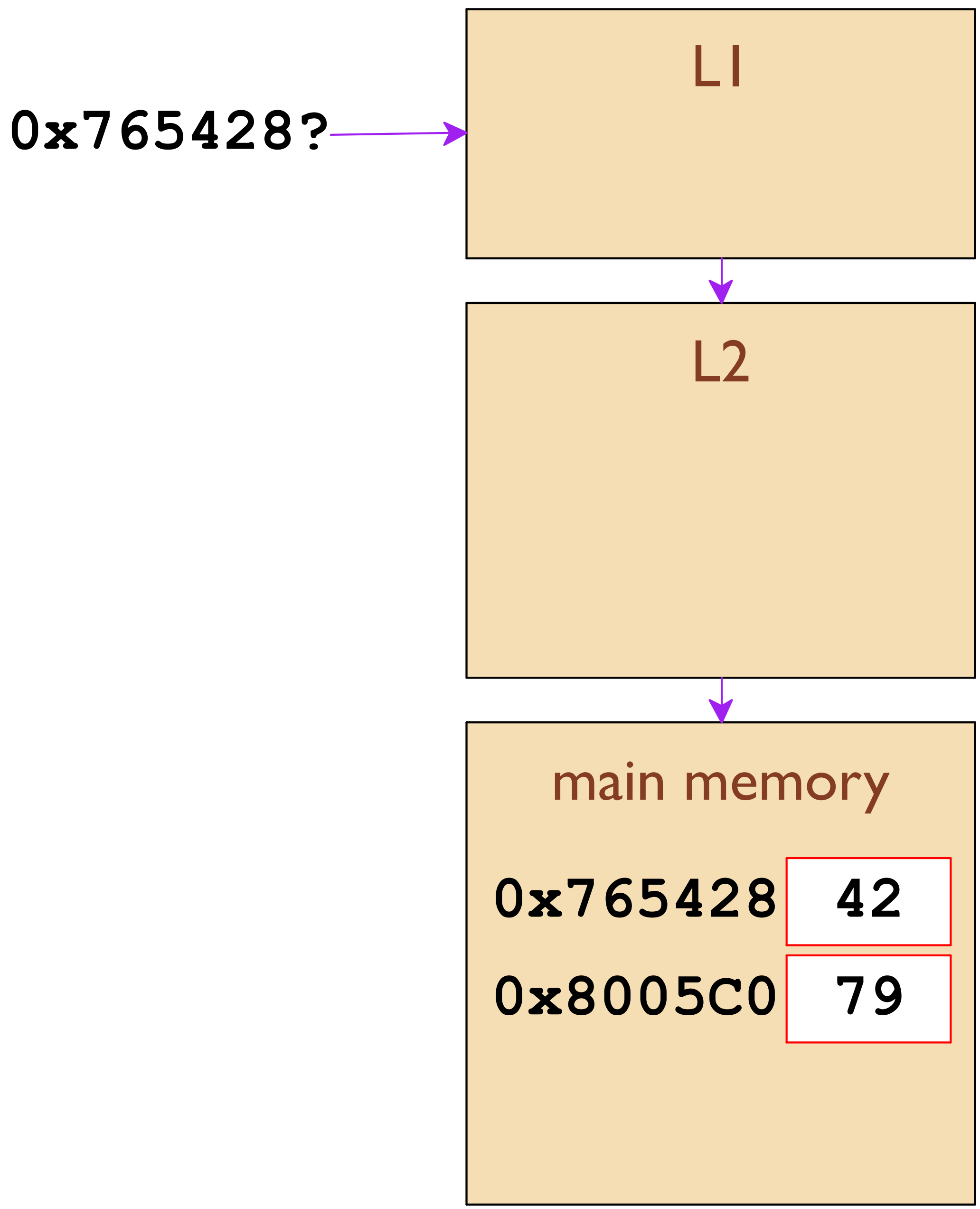
Cache Lookup



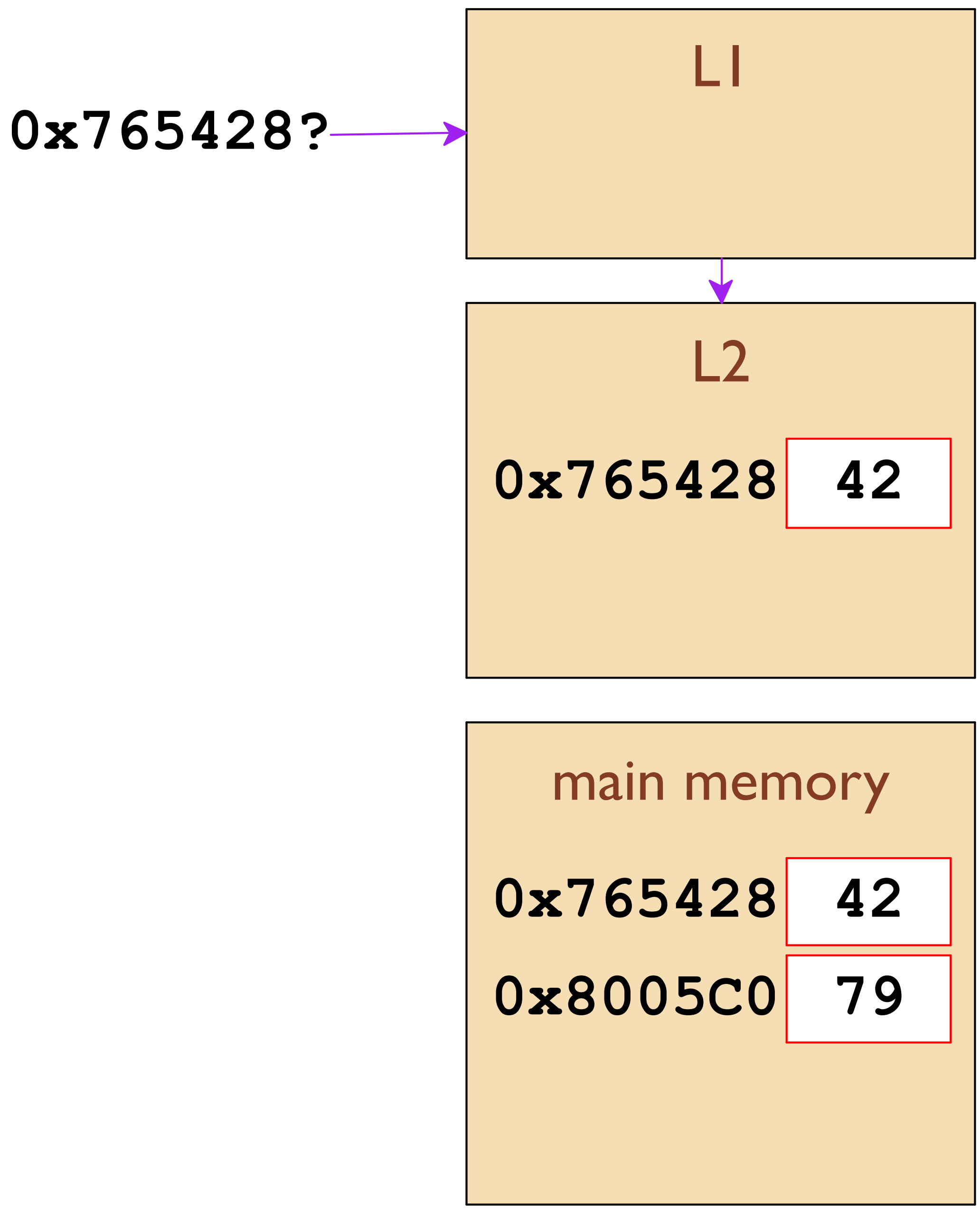
Cache Lookup



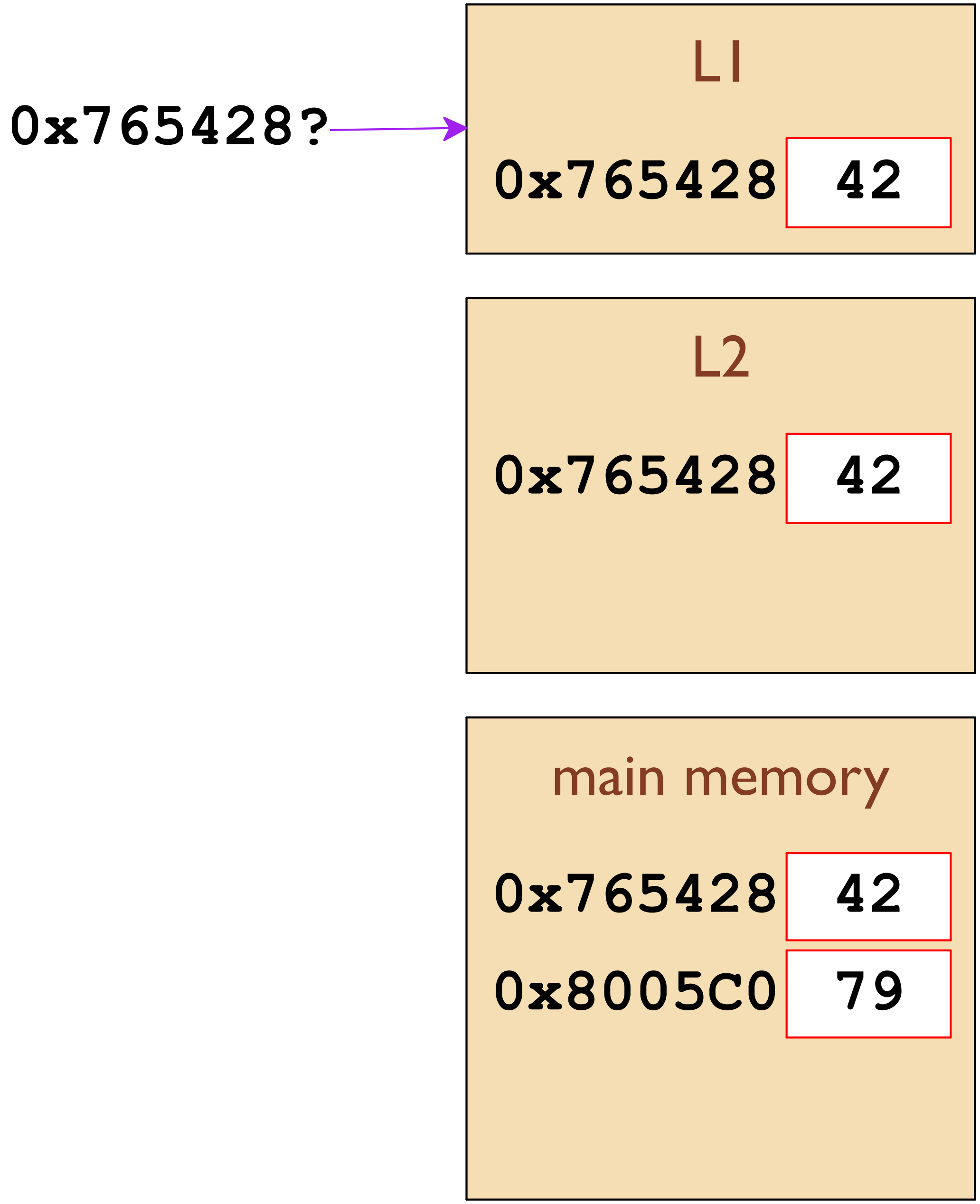
Cache Lookup



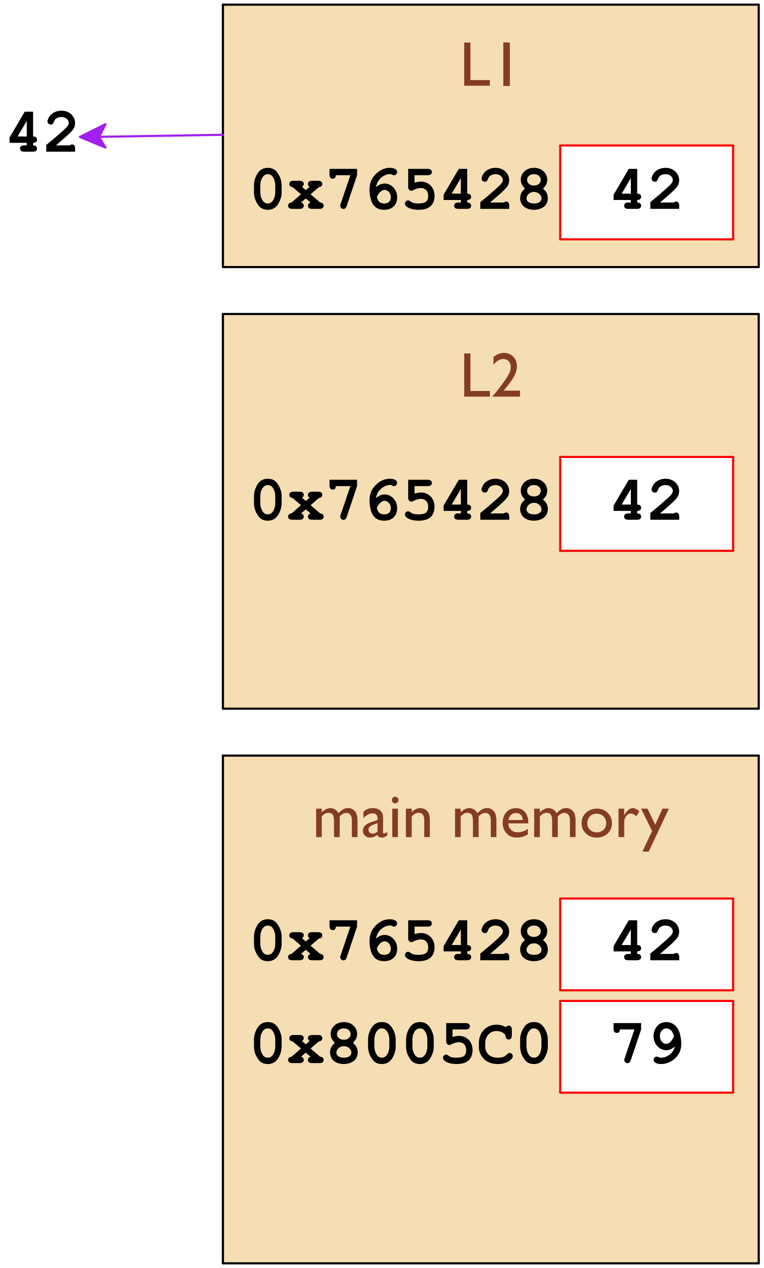
Cache Lookup



Cache Lookup



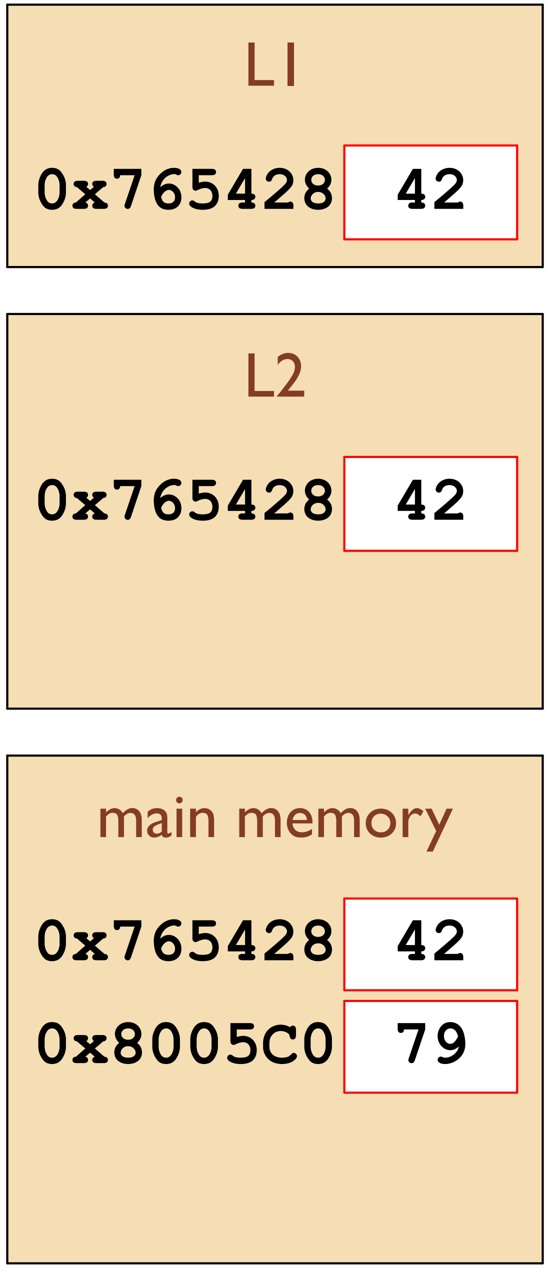
Cache Lookup



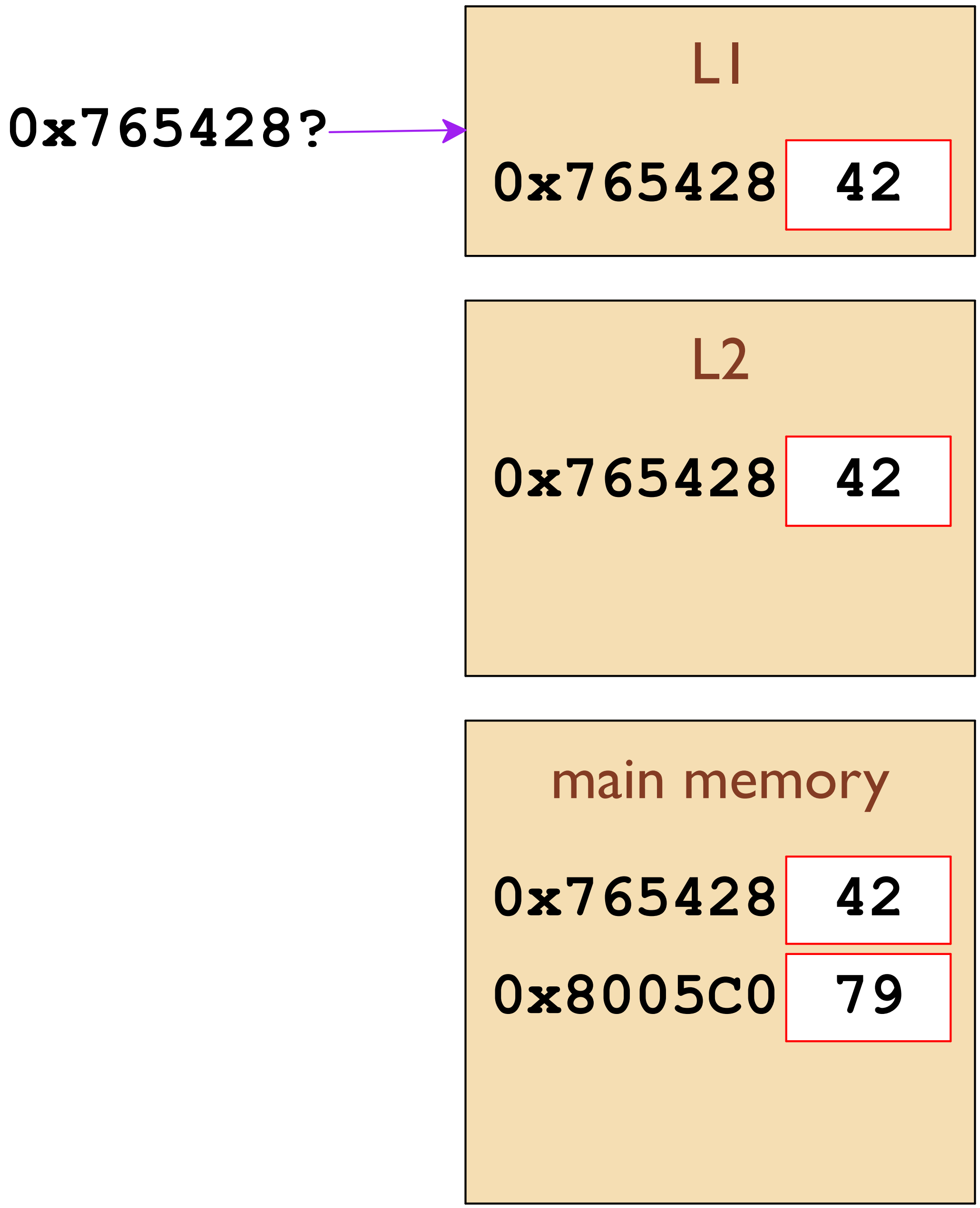
Cache Lookup

`0x765428?`

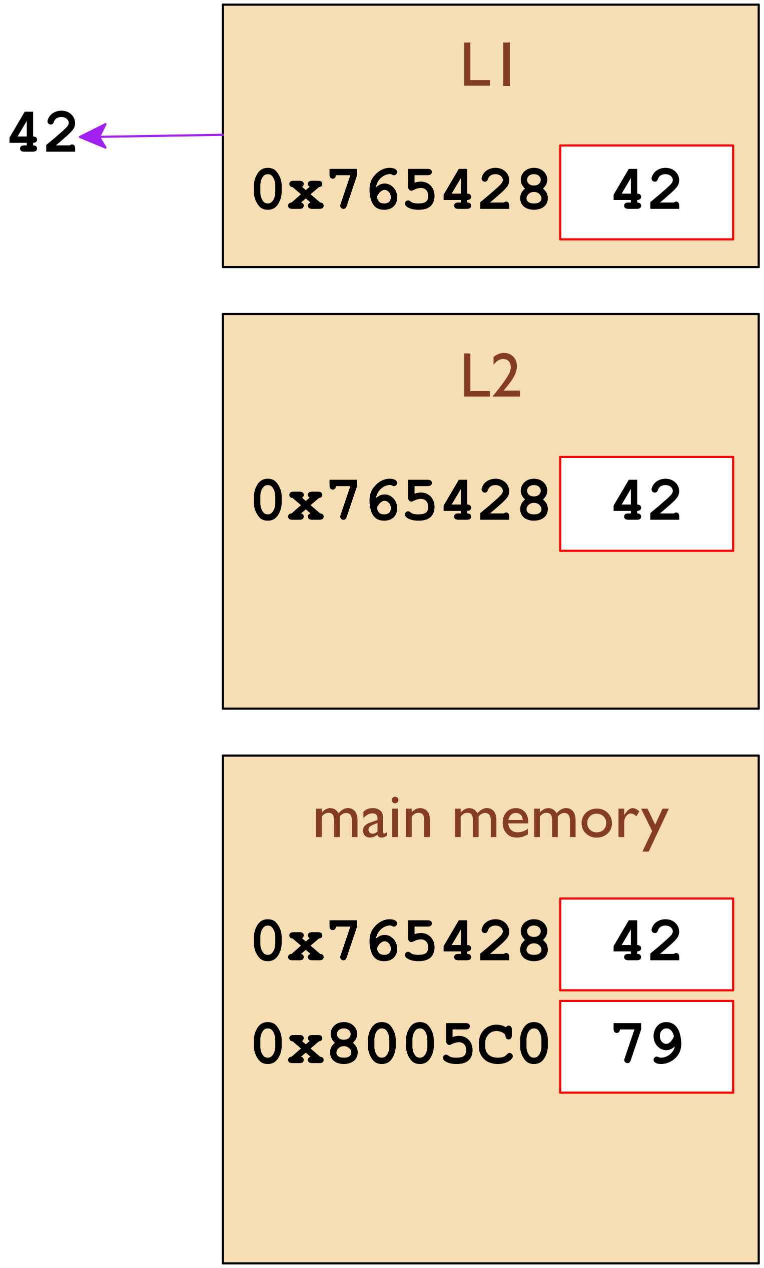
temporal locality



Cache Lookup



Cache Lookup



Cache Lookup

`0x8005C0?`

L1

`0x765428`	42

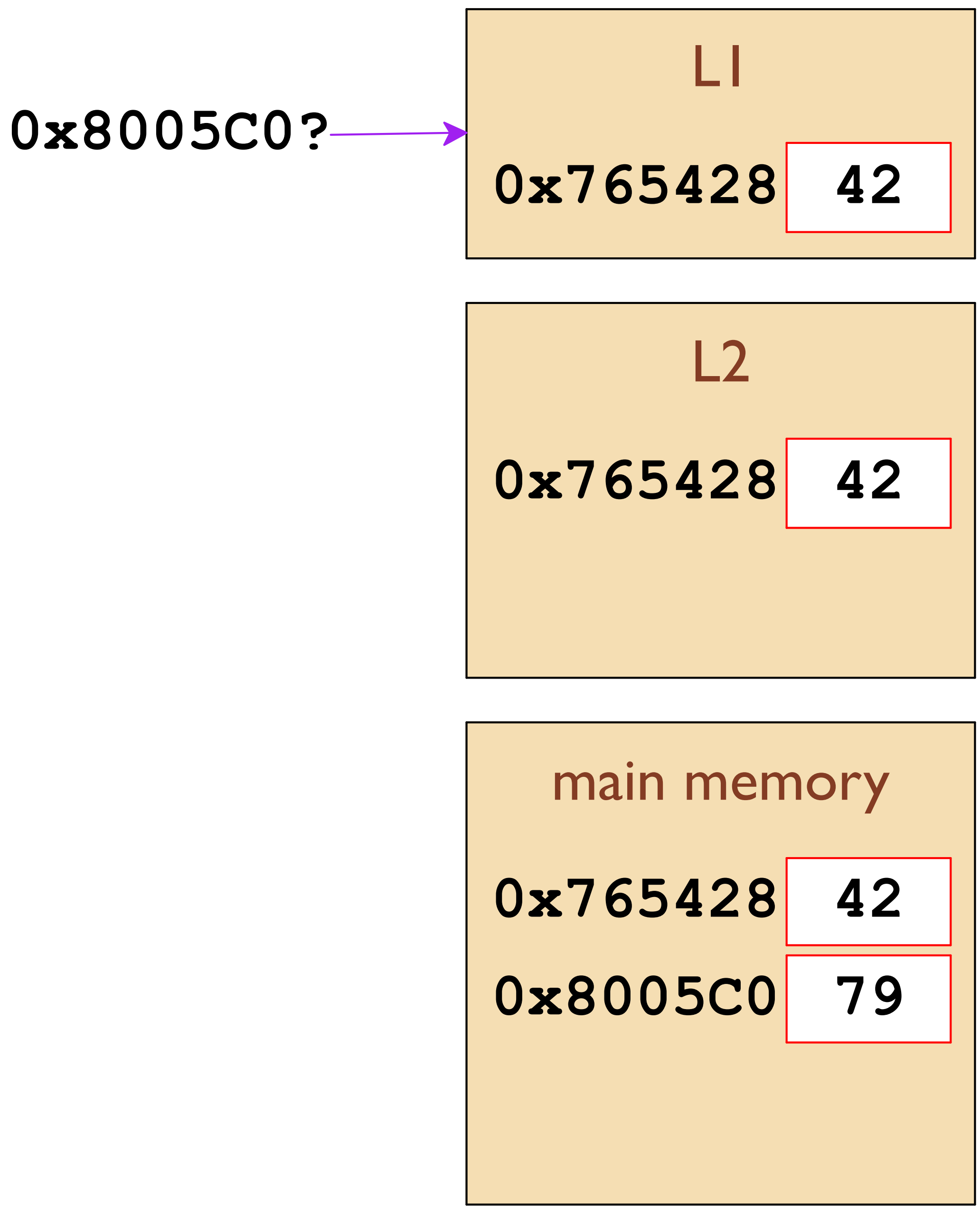
L2

`0x765428`	42

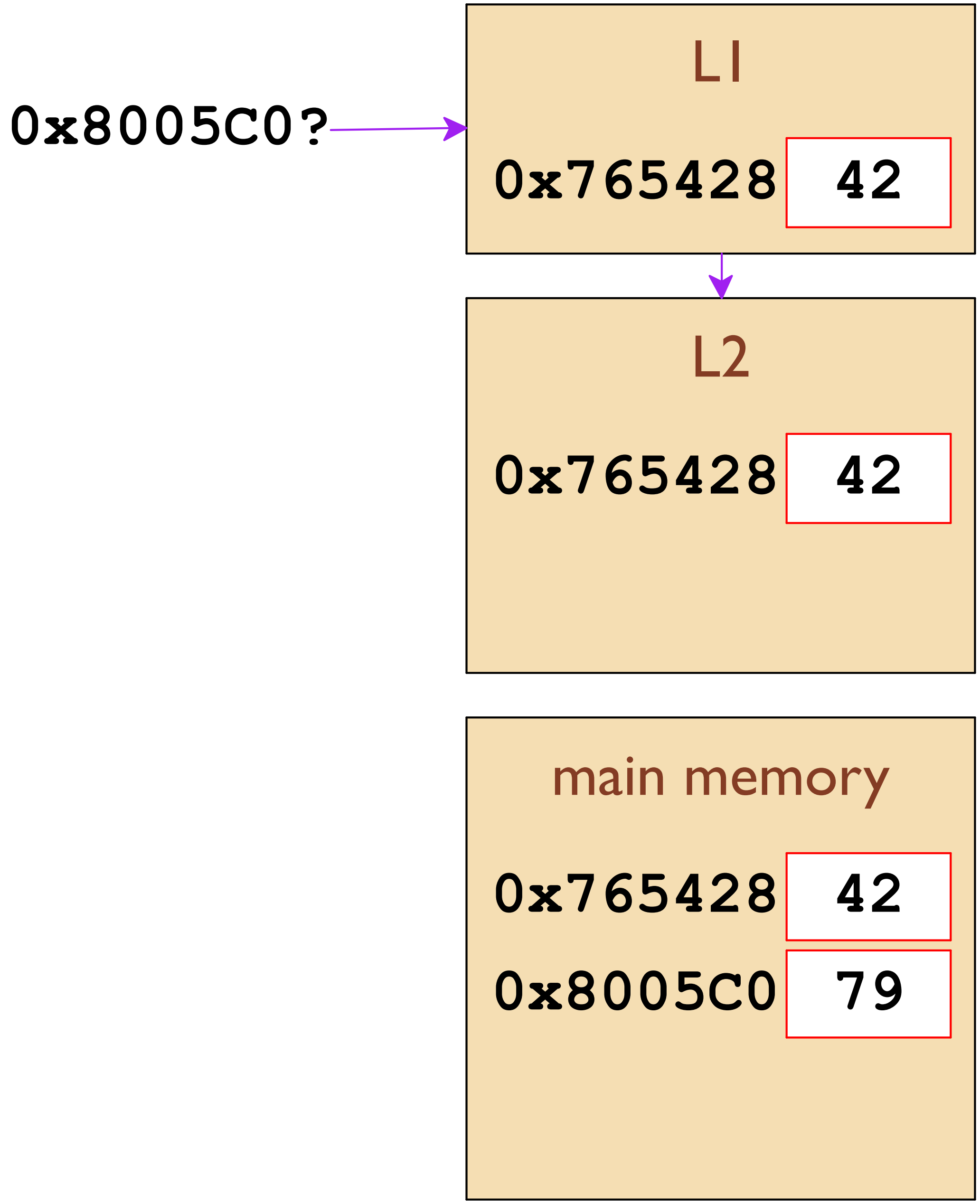
main memory

`0x765428`	42
`0x8005C0`	79

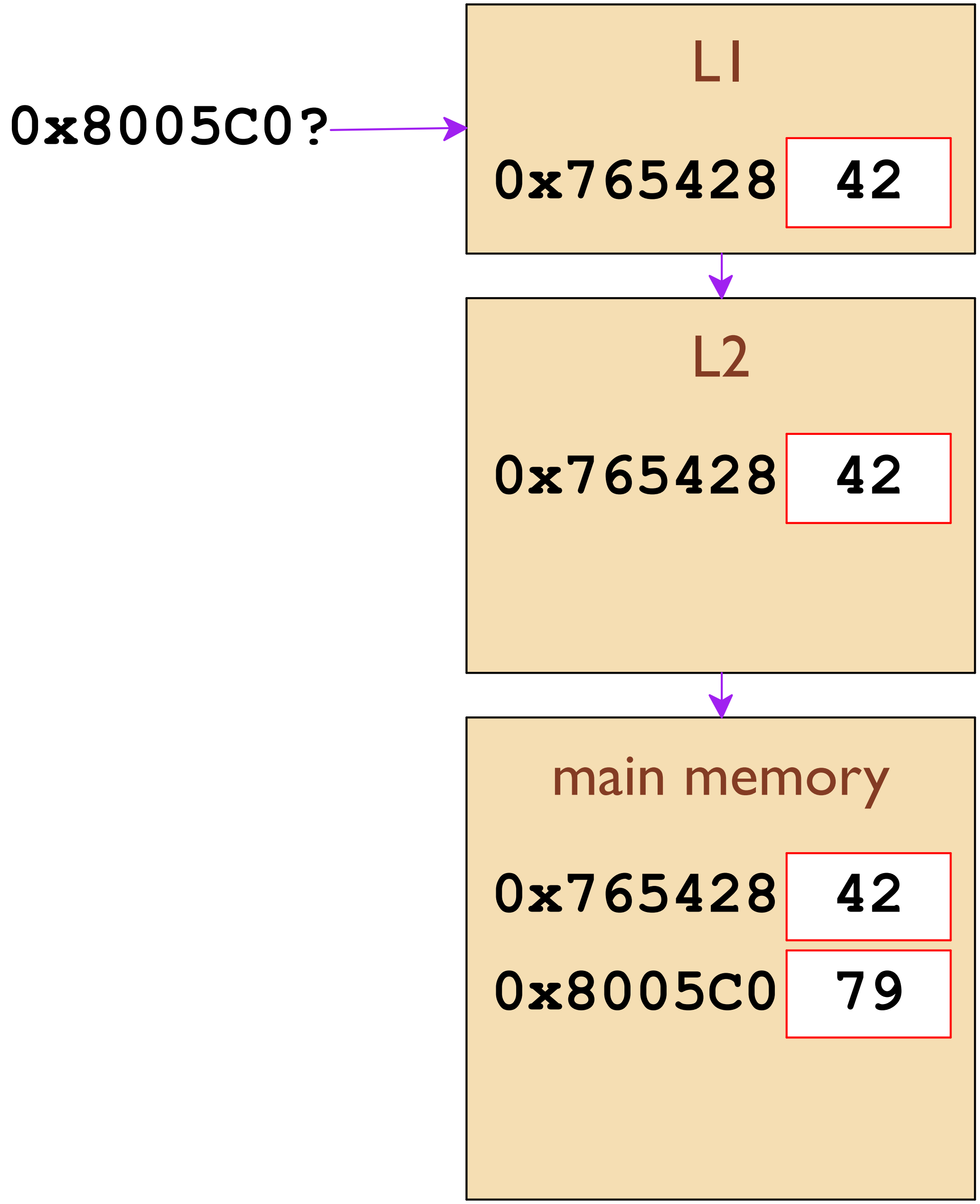
Cache Lookup



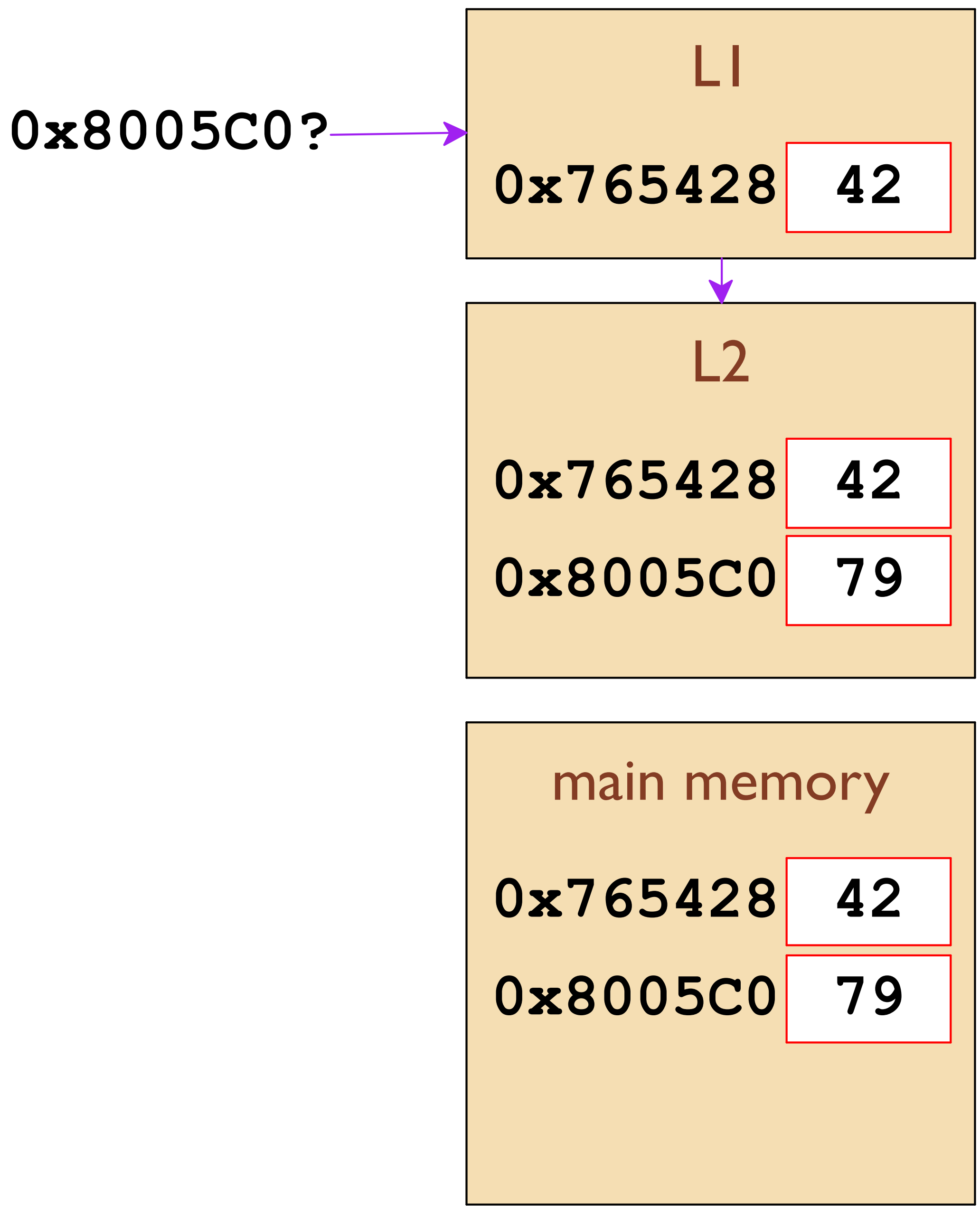
Cache Lookup



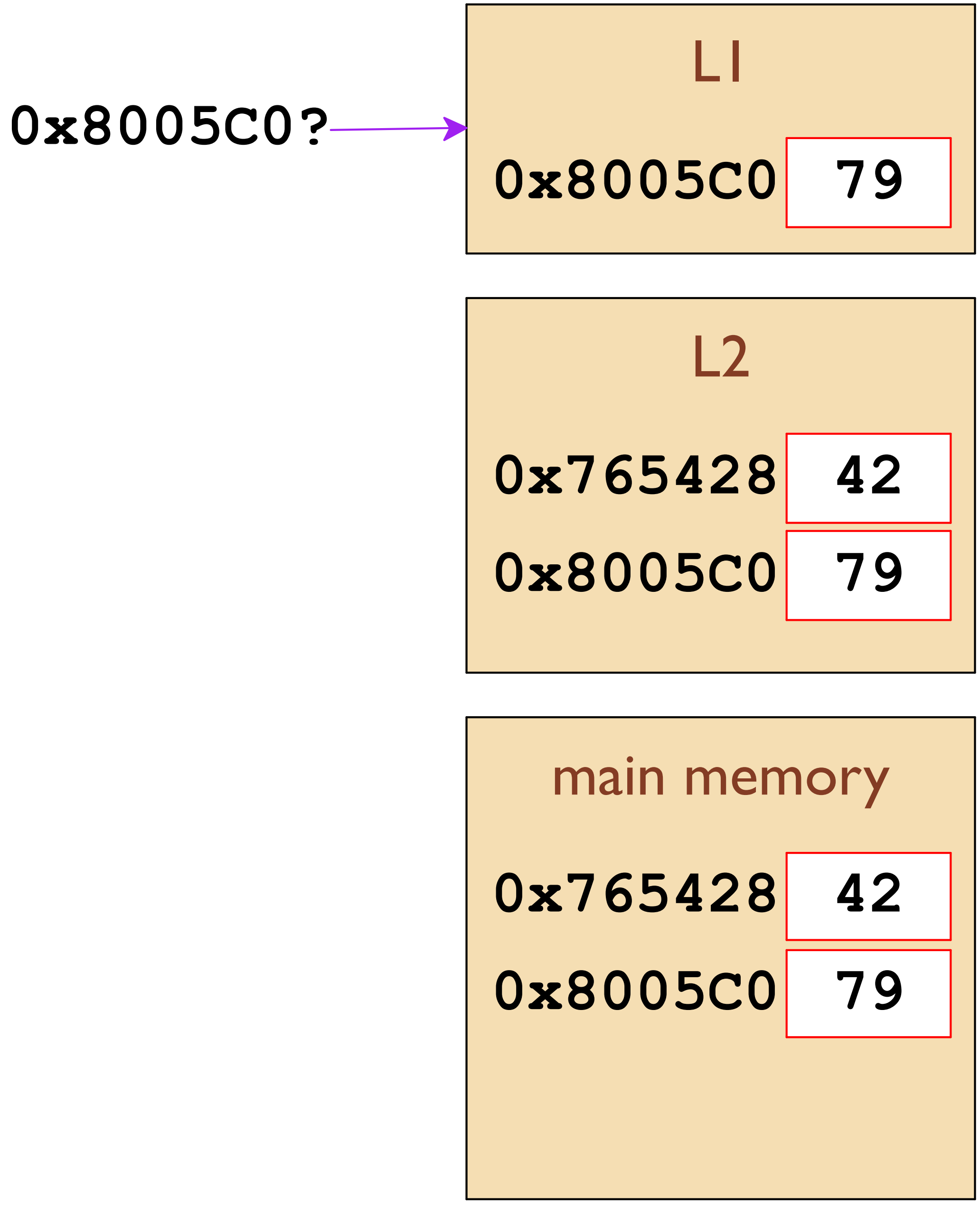
Cache Lookup



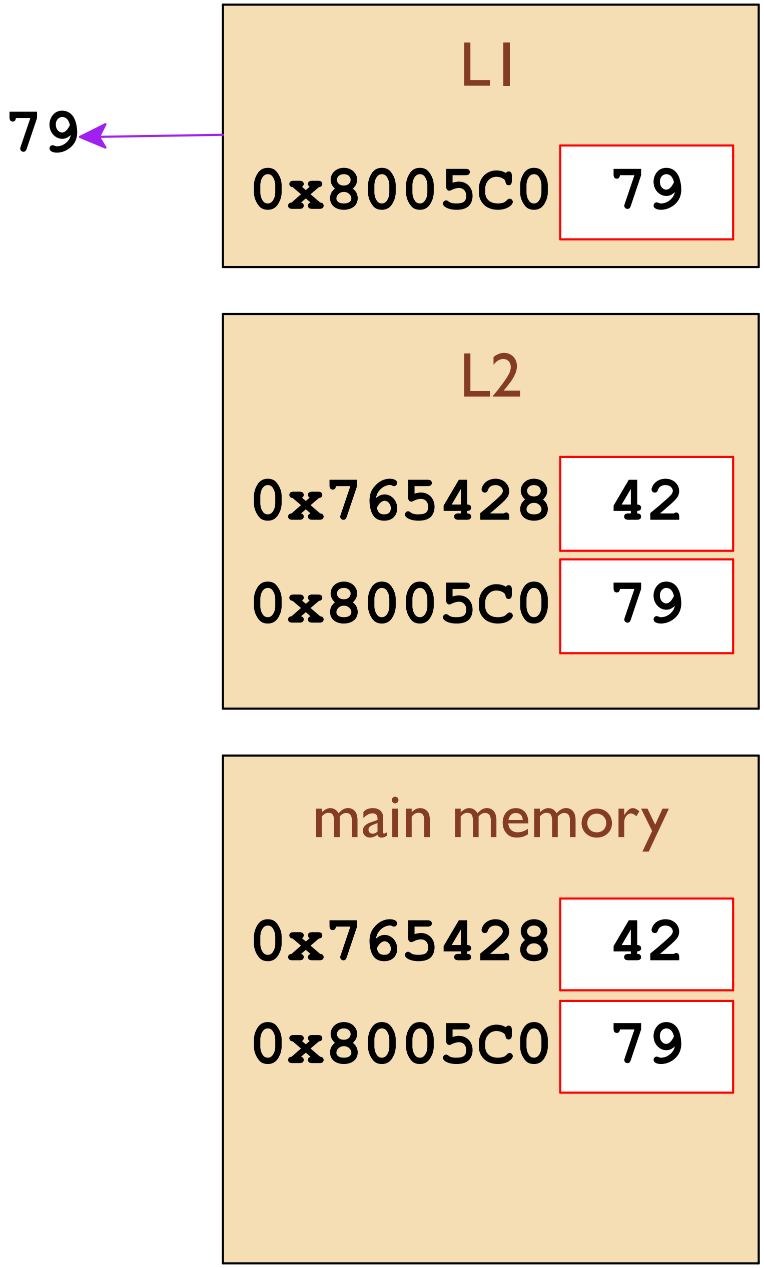
Cache Lookup



Cache Lookup



Cache Lookup



Cache Lookup

0x765428?

L1

0x8005C0	**79**

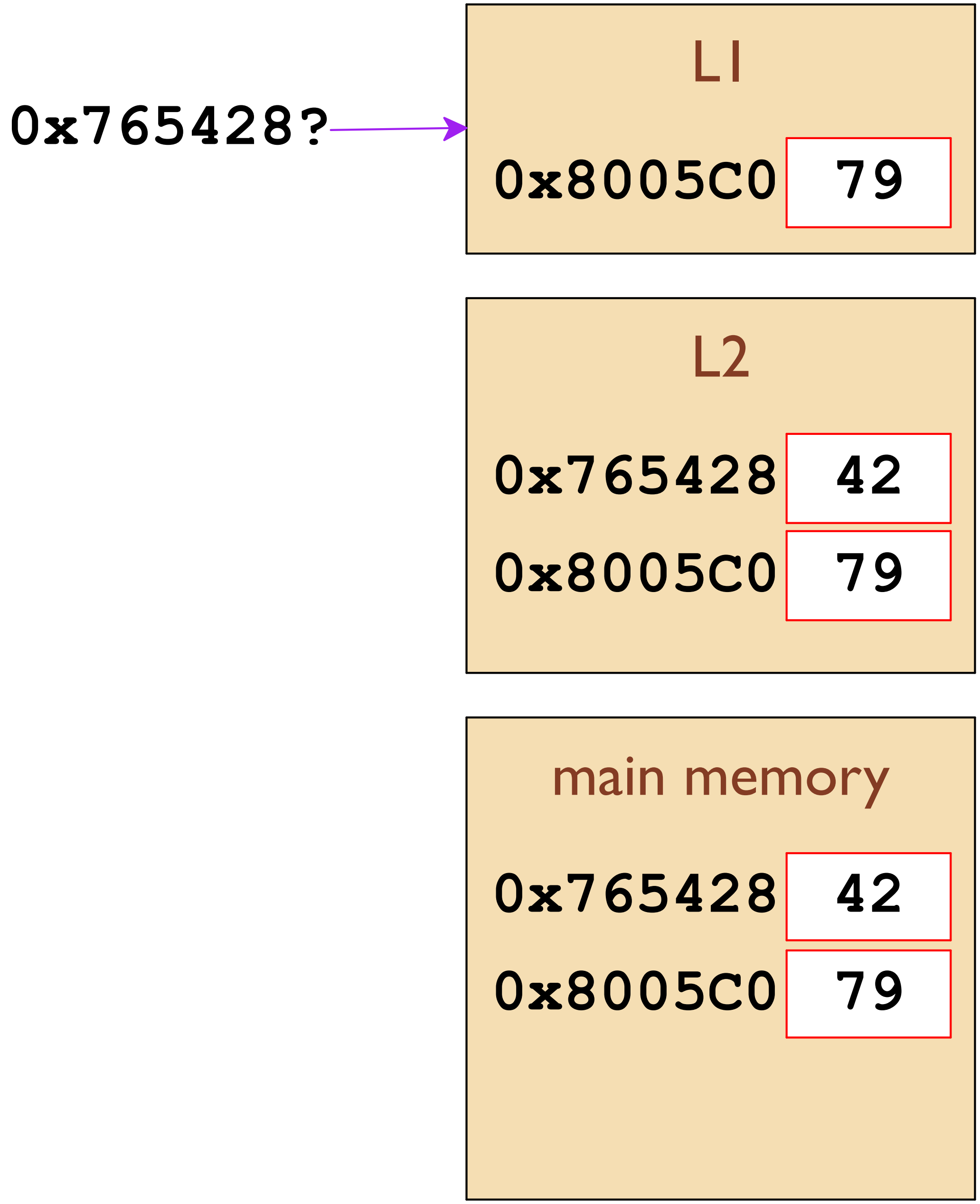
L2

0x765428	**42**
0x8005C0	**79**

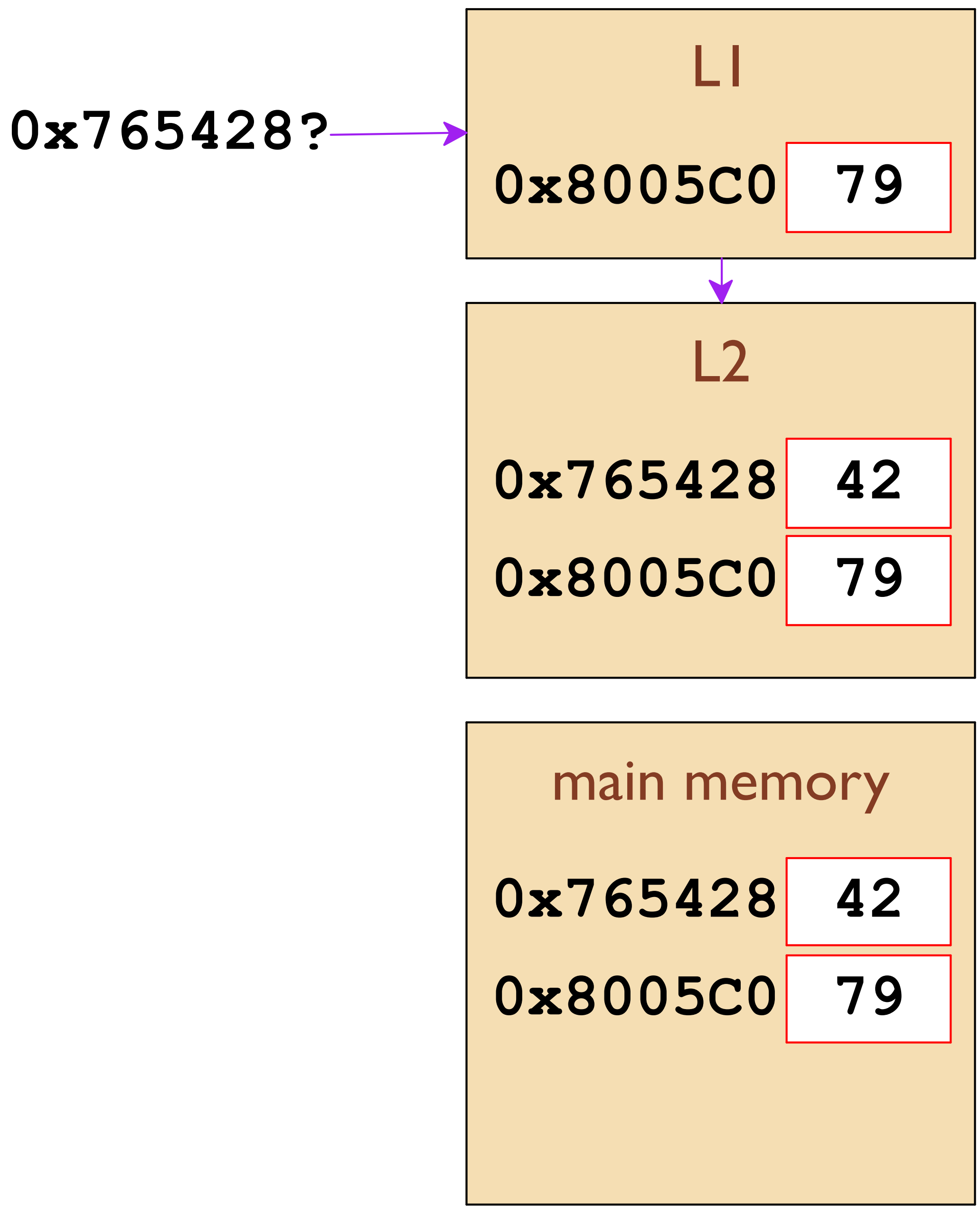
main memory

0x765428	**42**
0x8005C0	**79**

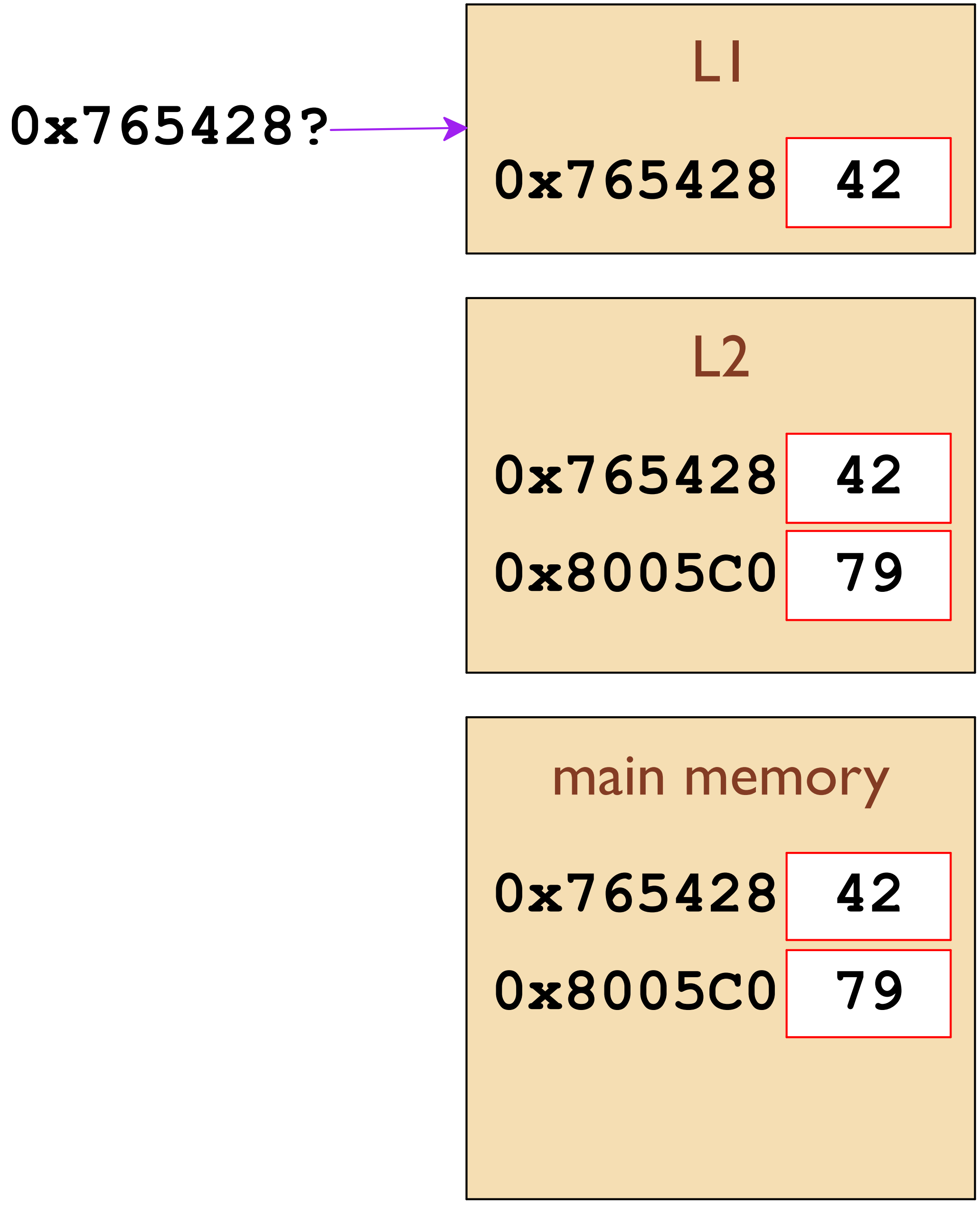
Cache Lookup



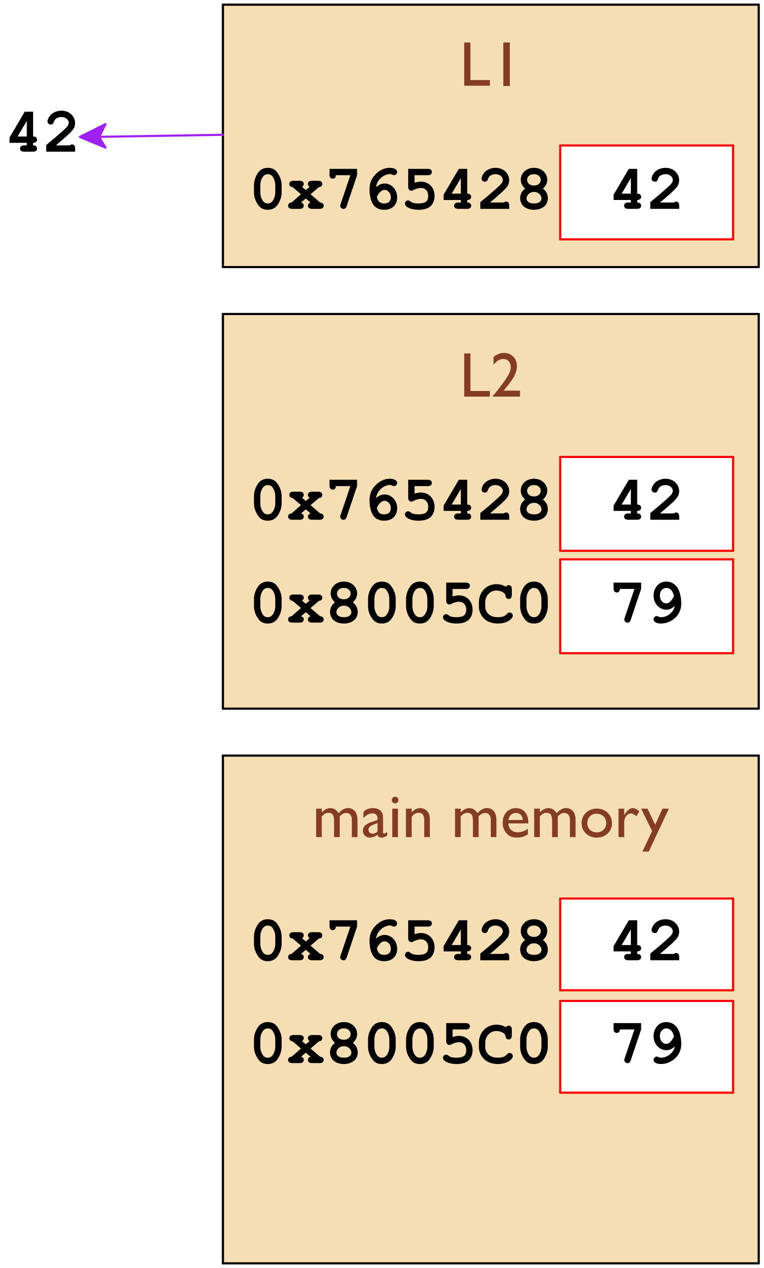
Cache Lookup



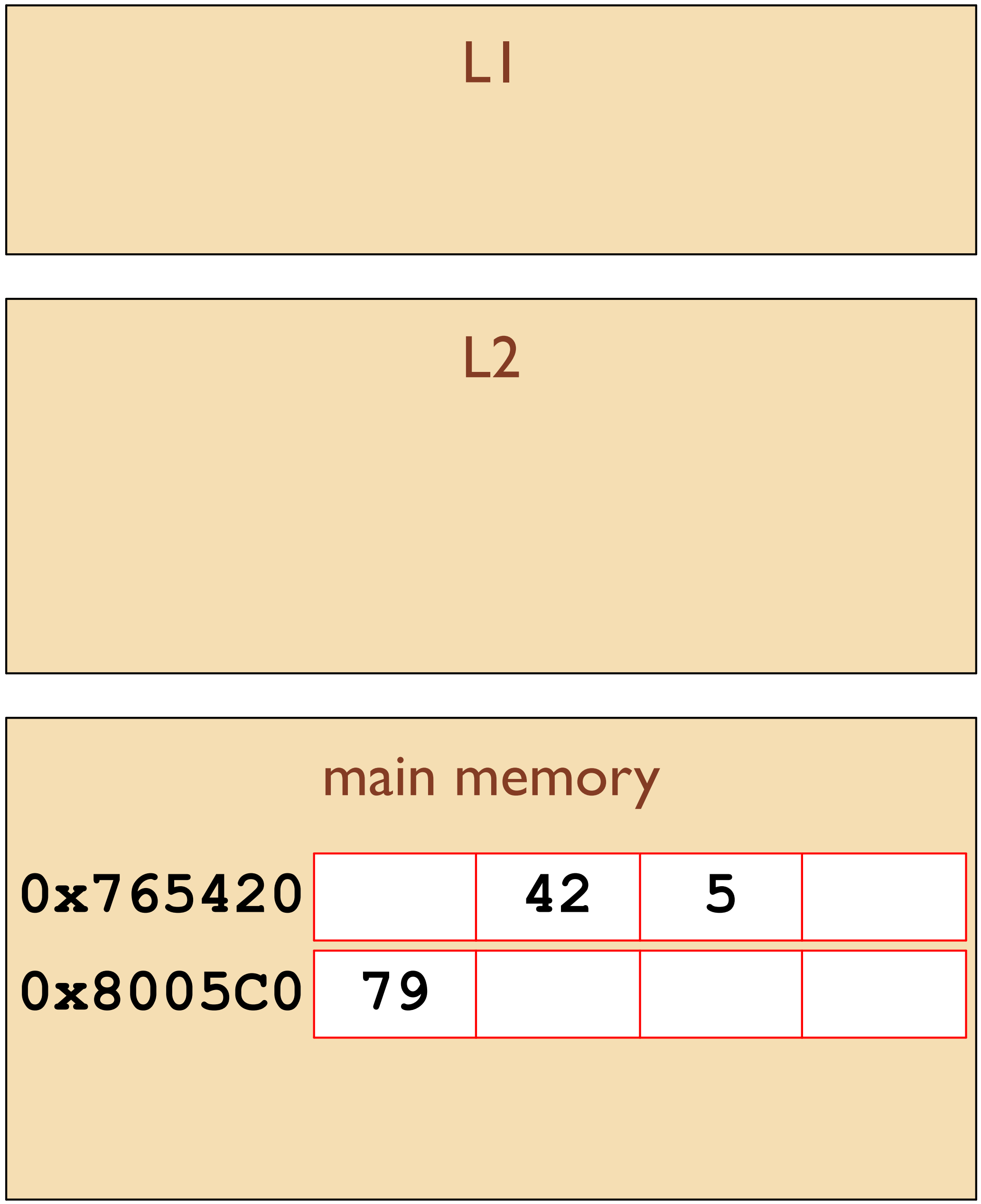
Cache Lookup



Cache Lookup



Cache Lookup



Cache Lookup

`0x765428?`

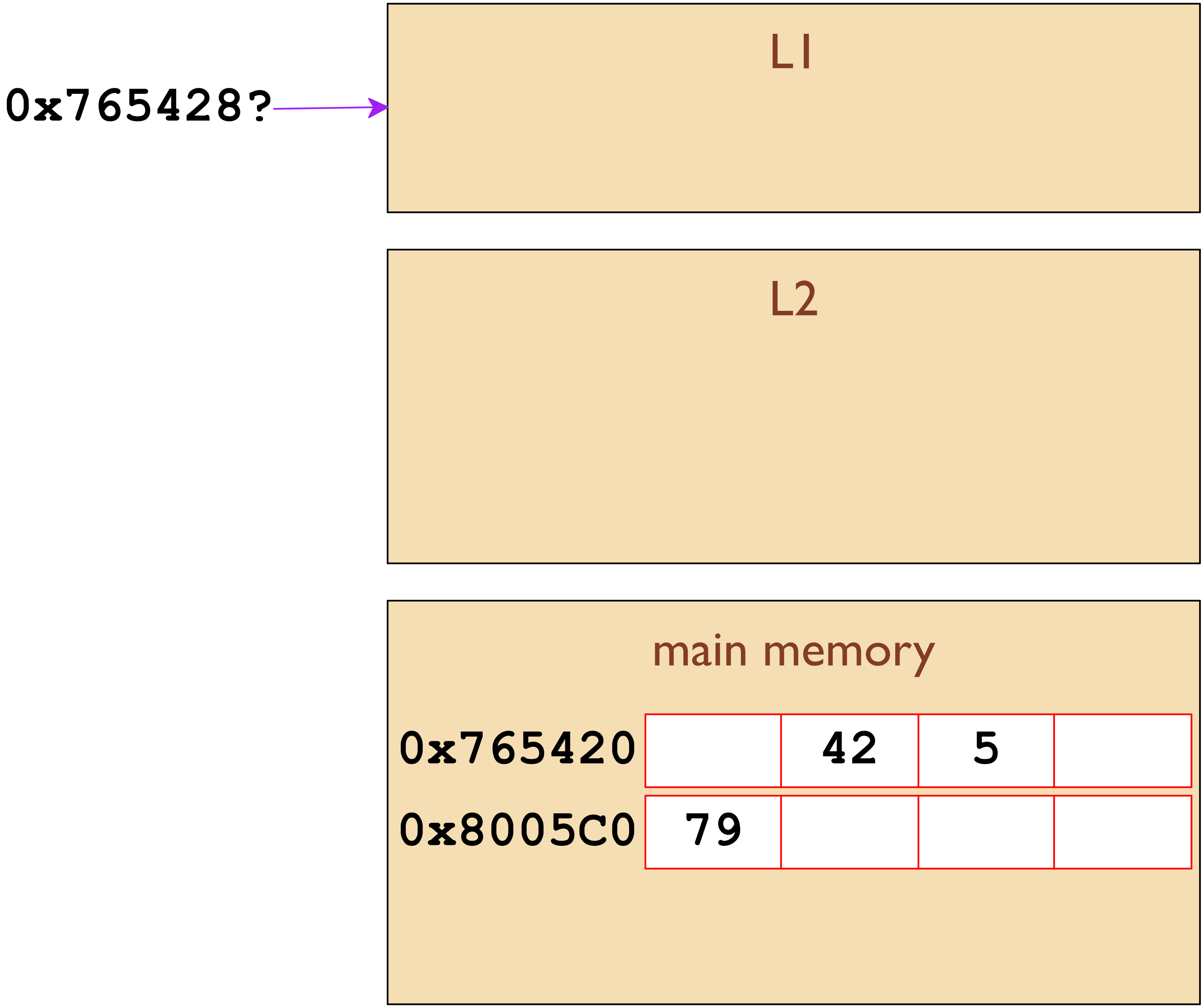
L1

L2

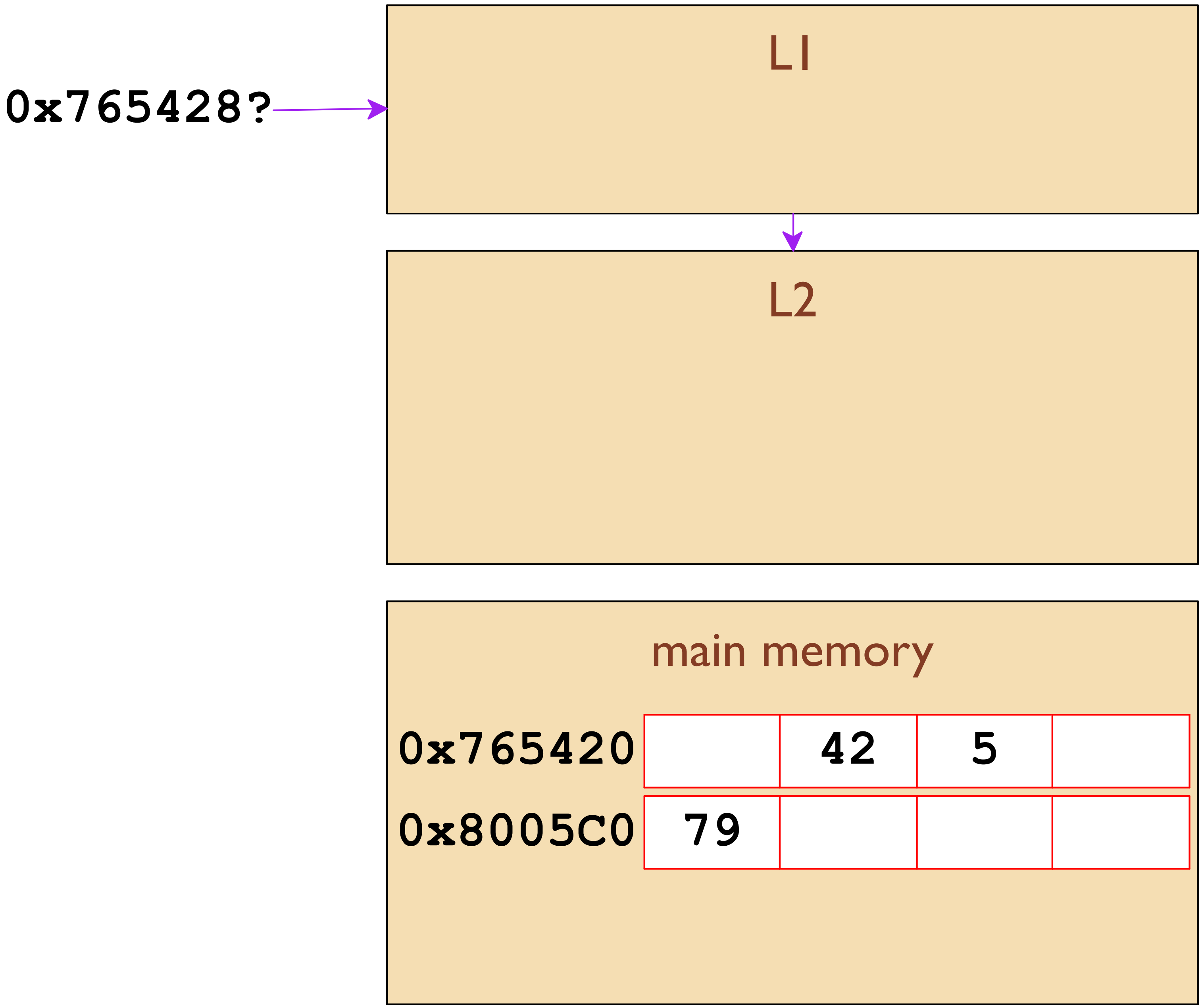
main memory

`0x765420`		42	5	
`0x8005C0`	79			

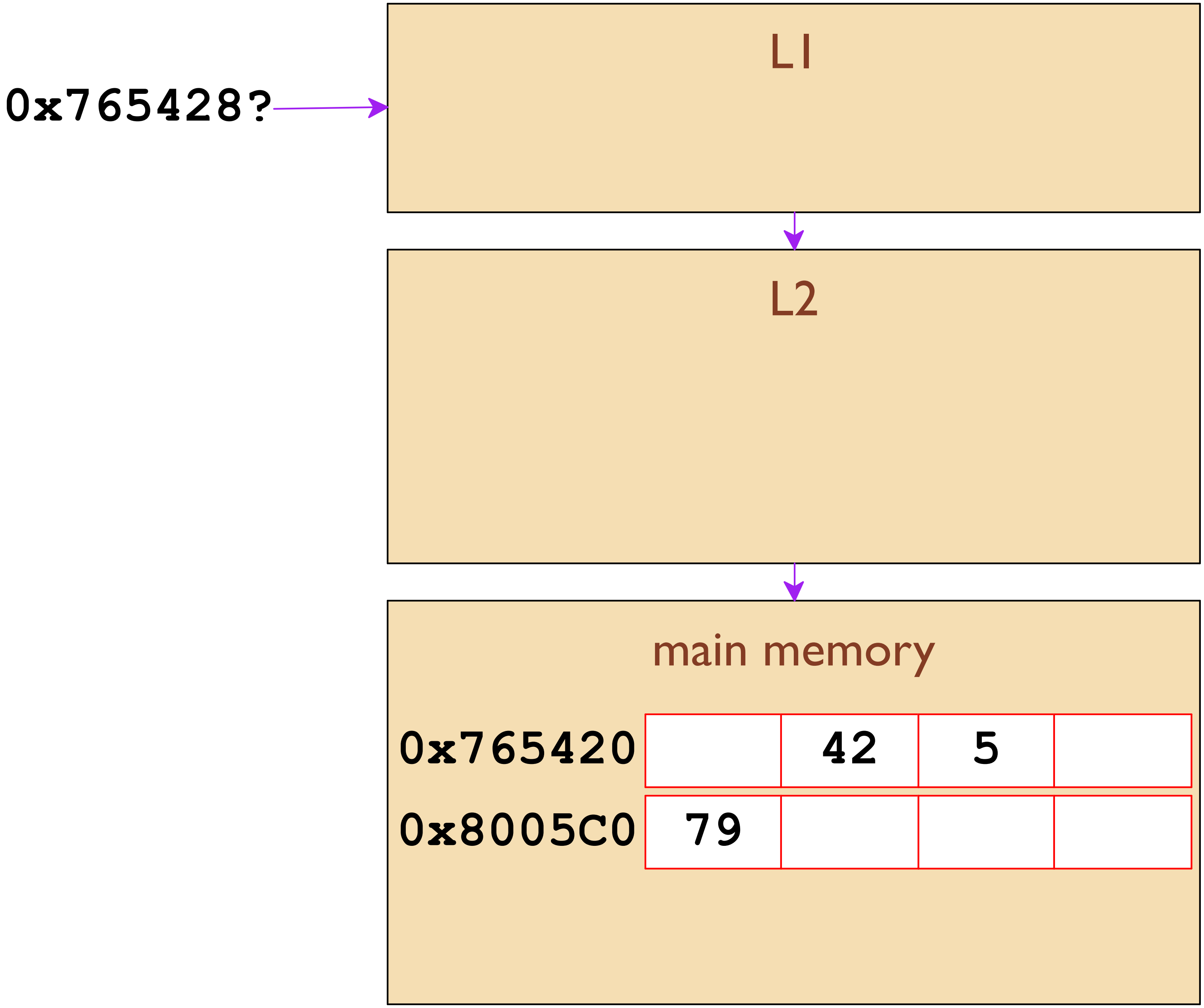
Cache Lookup



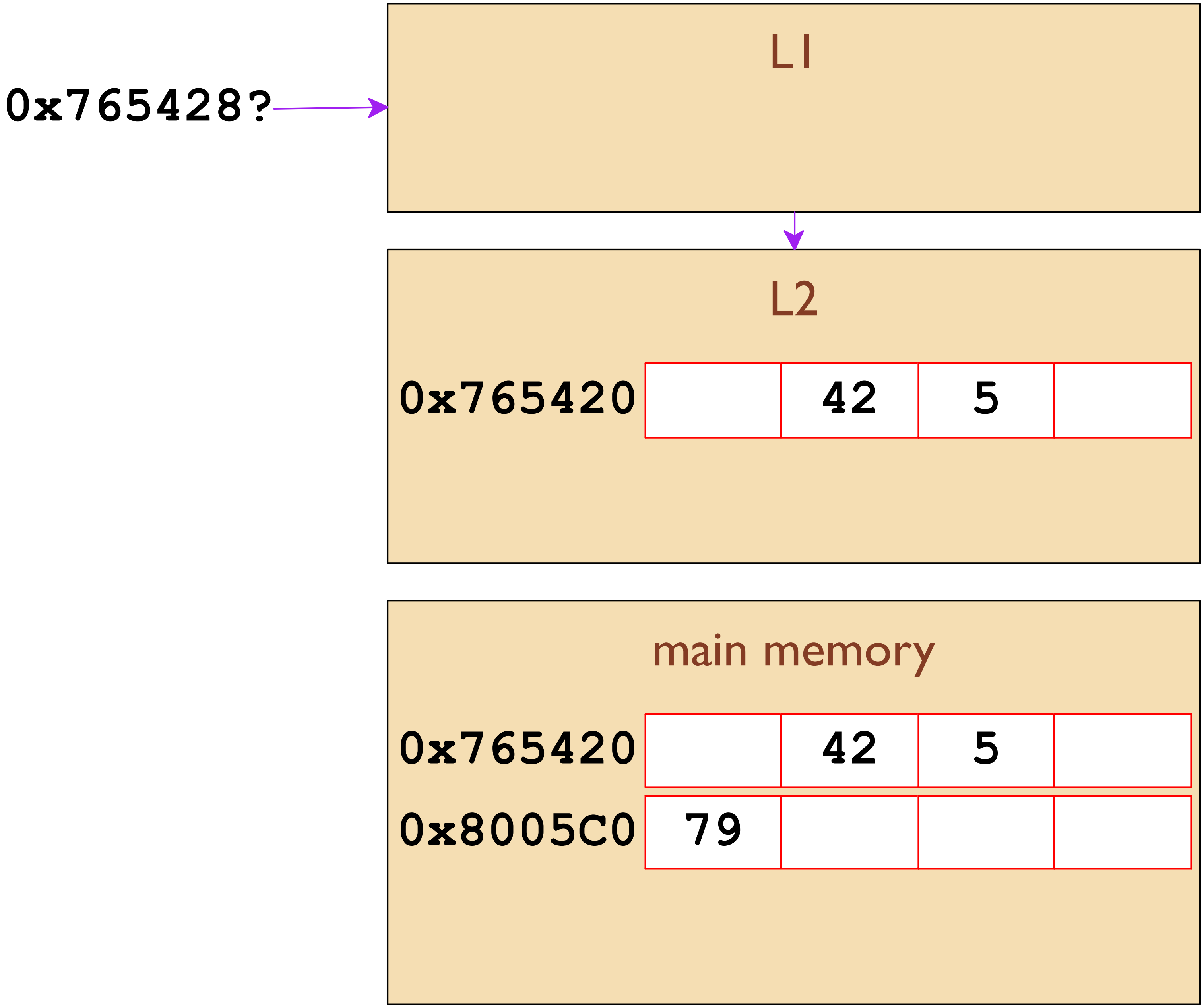
Cache Lookup



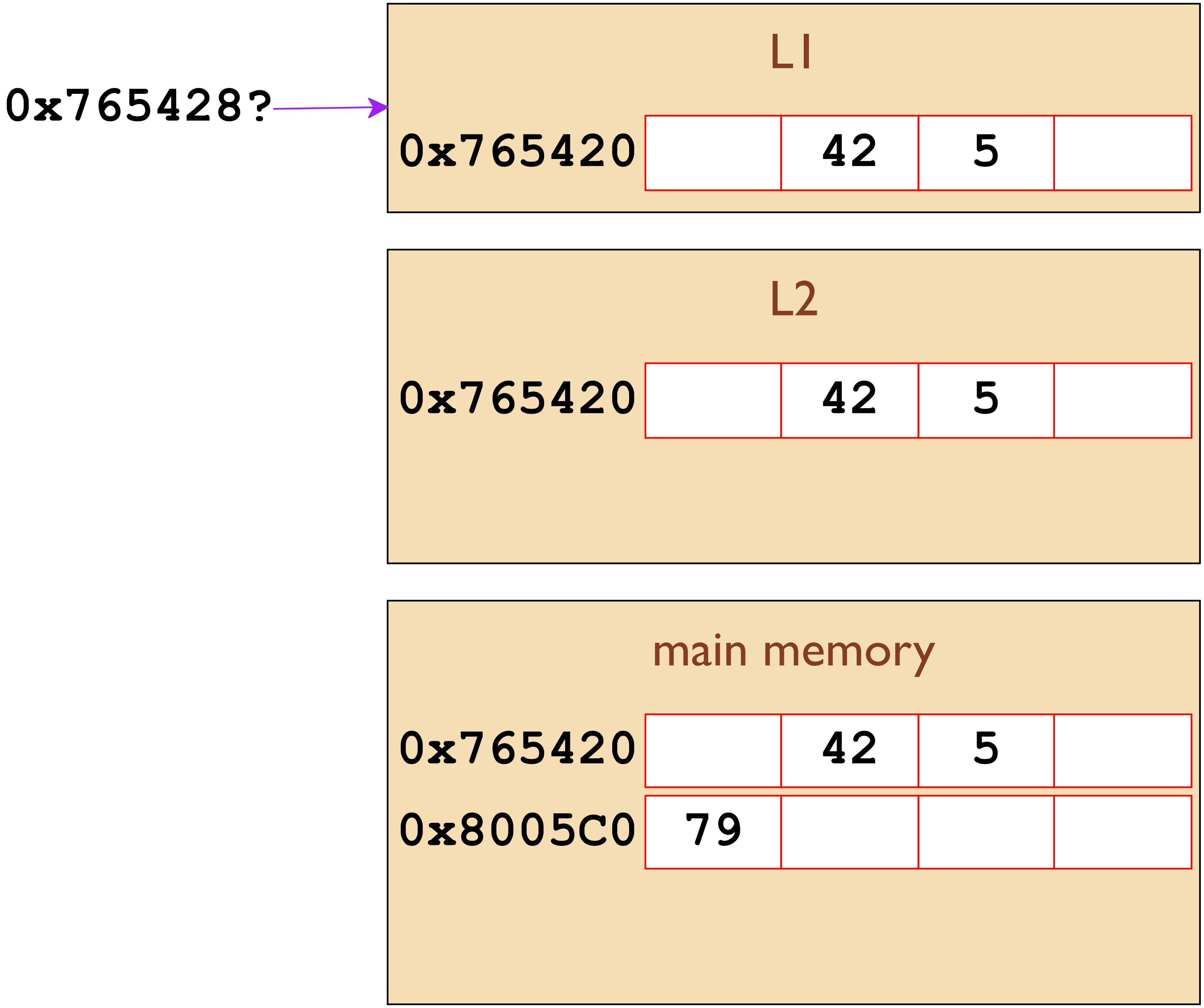
Cache Lookup



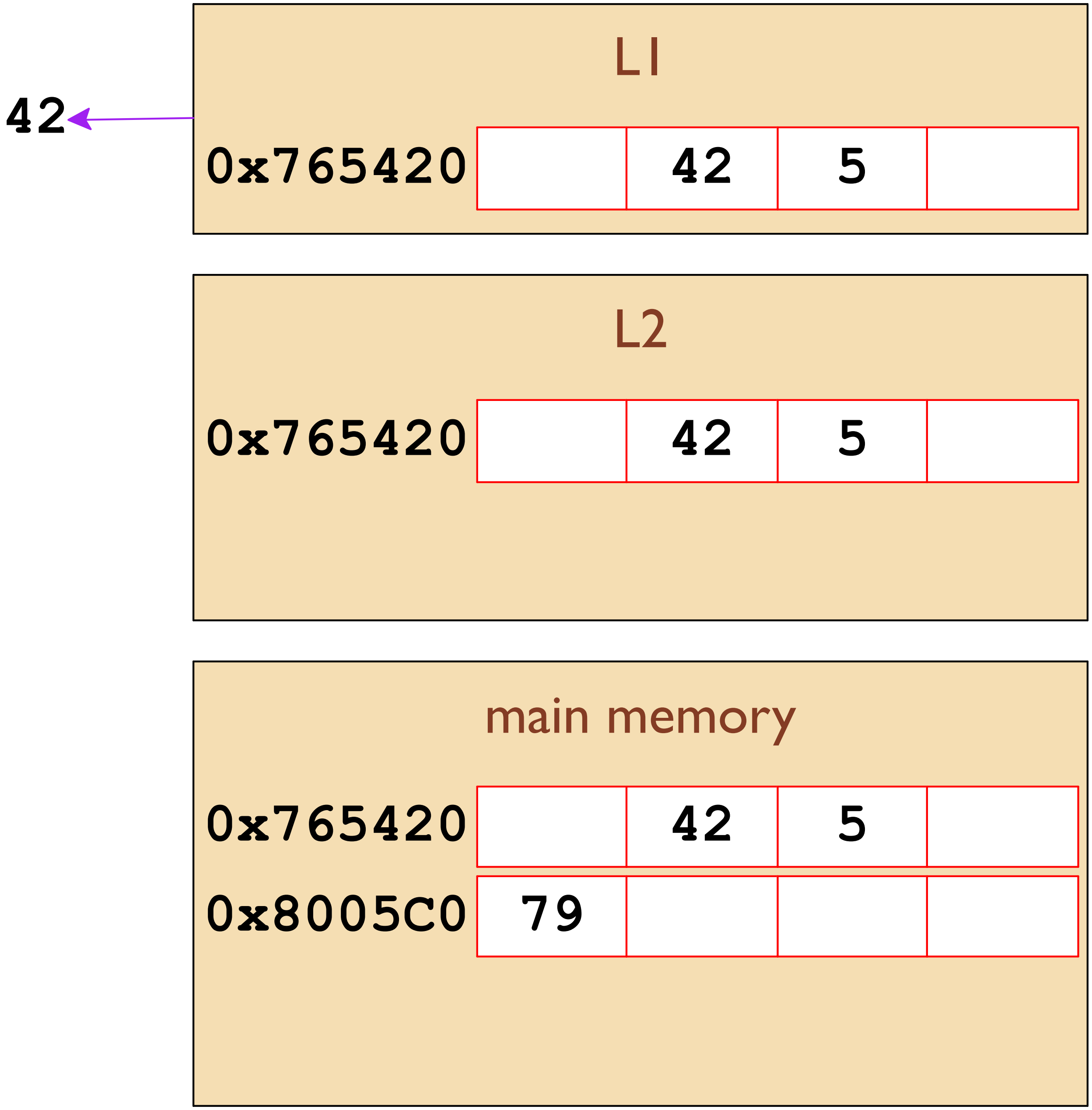
Cache Lookup



Cache Lookup



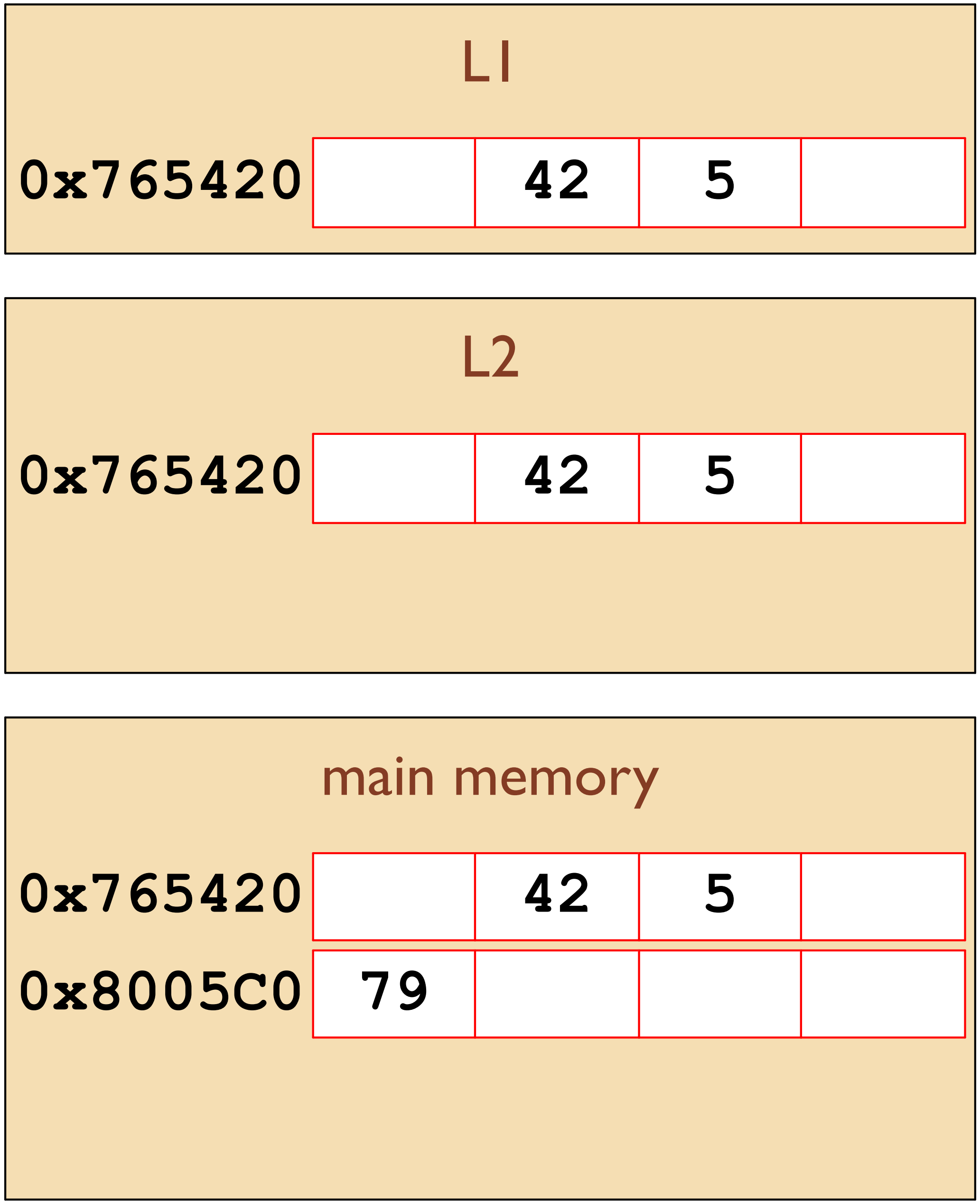
Cache Lookup



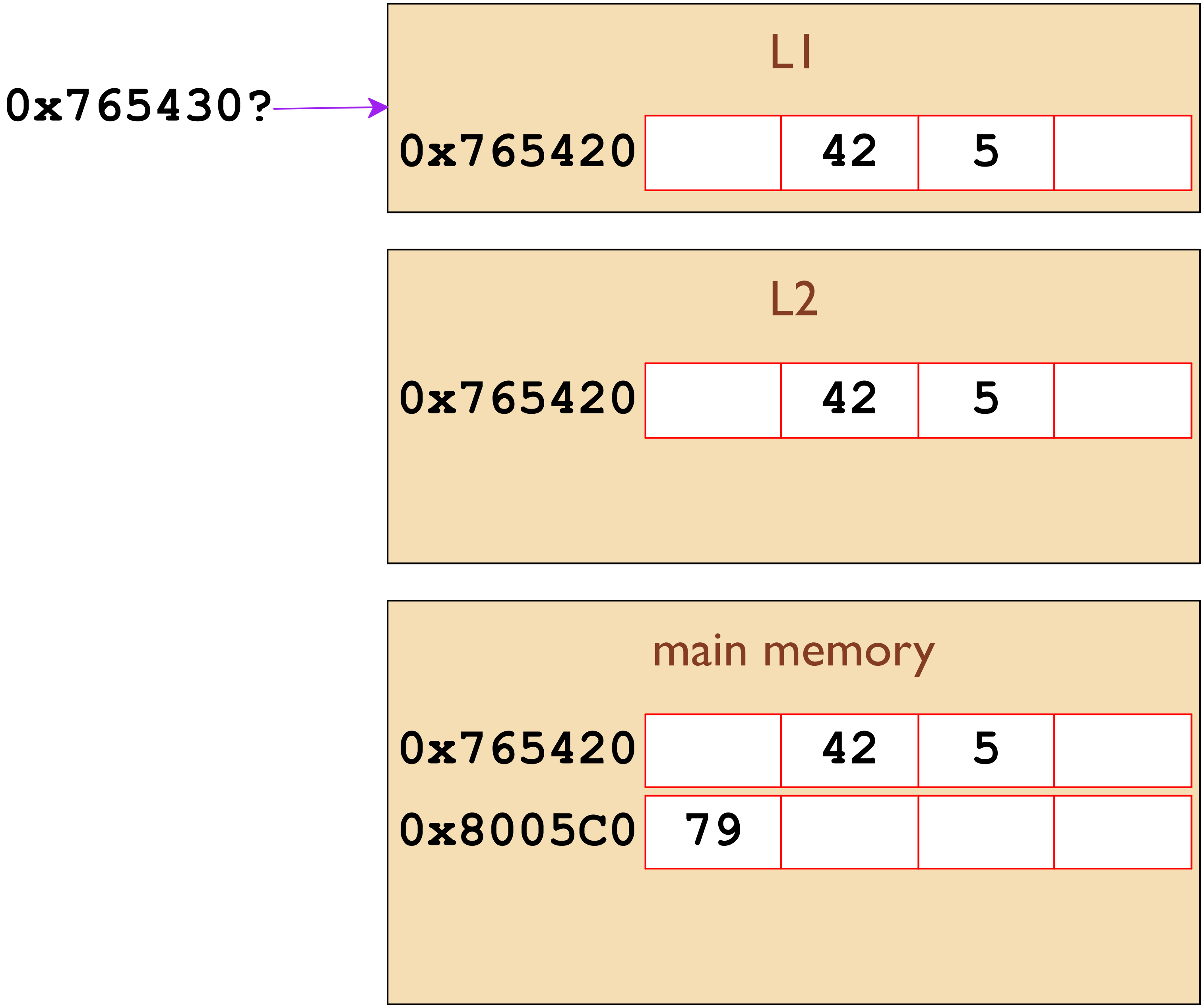
Cache Lookup

`0x765430?`

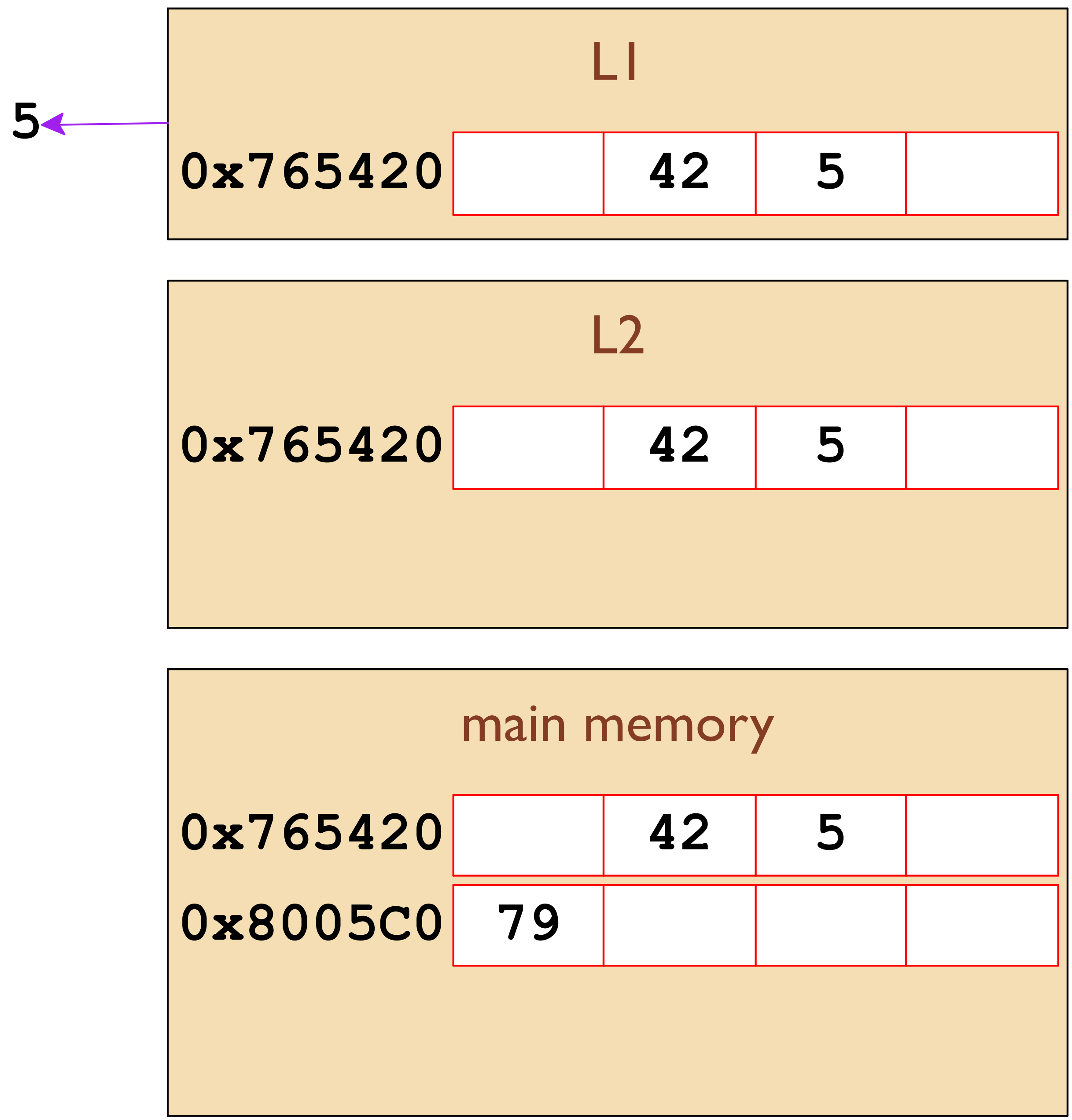
spatial locality



Cache Lookup



Cache Lookup



Memory Access Times

2^{30} sequential accesses of an `int` spread across...

64B	0.8s
128B	0.8s
...	
32kB	0.8s
64kB	0.83s
128kB	0.88s
256kB	0.91s
512kB	0.91s
1MB	0.95s
2MB	0.95s
4MB	0.99s
8MB	1.07s
16MB	1.08s

L1 cache: 32kB
- 4-5 cycles
- 64B blocks

L2 cache: 256kB
- 12 cycles
- 64B blocks

L3 cache: 3MB
- 36 cycles
- 64B blocks

Memory: 8GB
- ~100 cycles

64B = 16 **`ints`**

Locality

Temporal locality

A previously used address is likely to be used again soon

Cache hierarchy positively correlates performance to temporal locality

Spatial locality

Newly used addresses are likely to be near recently used addresses

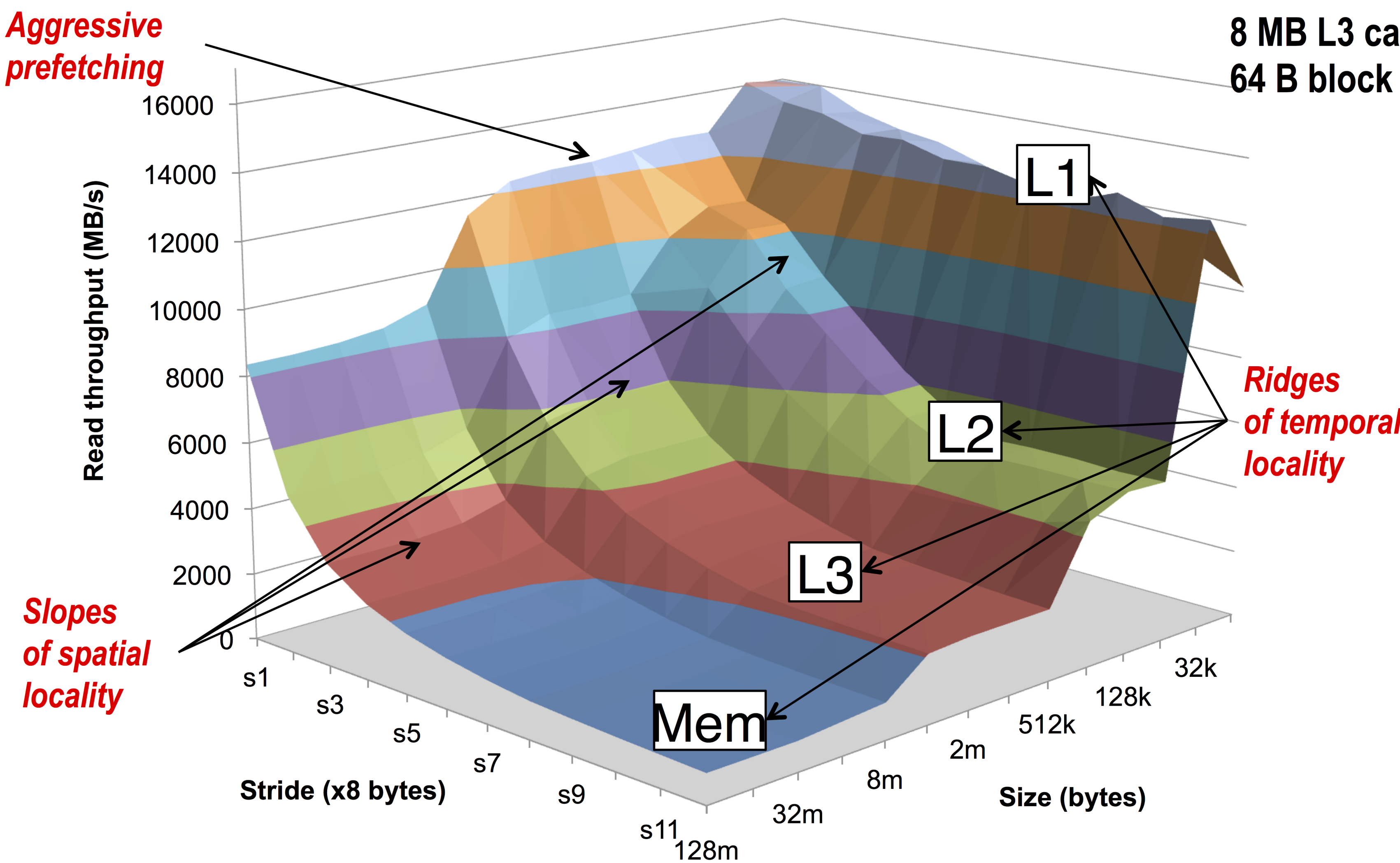
Blocking positively correlates performance to spatial locality

Size versus Stride

```
long data[MAXELEMS];

int test(int elems, int stride) {
  long i, sx2 = stride*2, sx3 = stride*3, sx4 = stride*4;
  long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
  long length = elems;
  long limit = length - sx4;

  for (i = 0; i < limit; i += sx4) {
    acc0 = acc0 + data[i];
    acc1 = acc1 + data[i+stride];
    acc2 = acc2 + data[i+sx2];
    acc3 = acc3 + data[i+sx3];
  }

  for (; i < length; i++) {
    acc0 = acc0 + data[i];
  }
  return ((acc0 + acc1) + (acc2 + acc3));
}
```

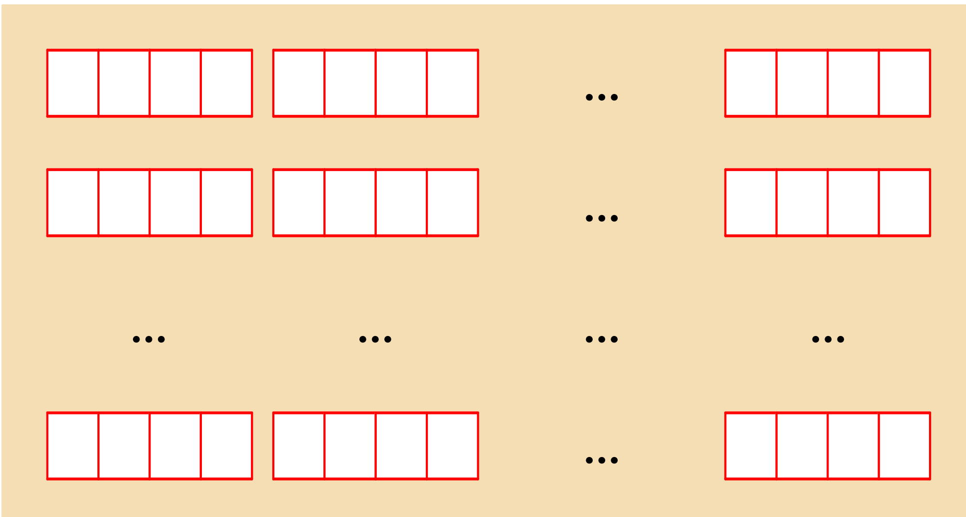
The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Finding a Cache Entry

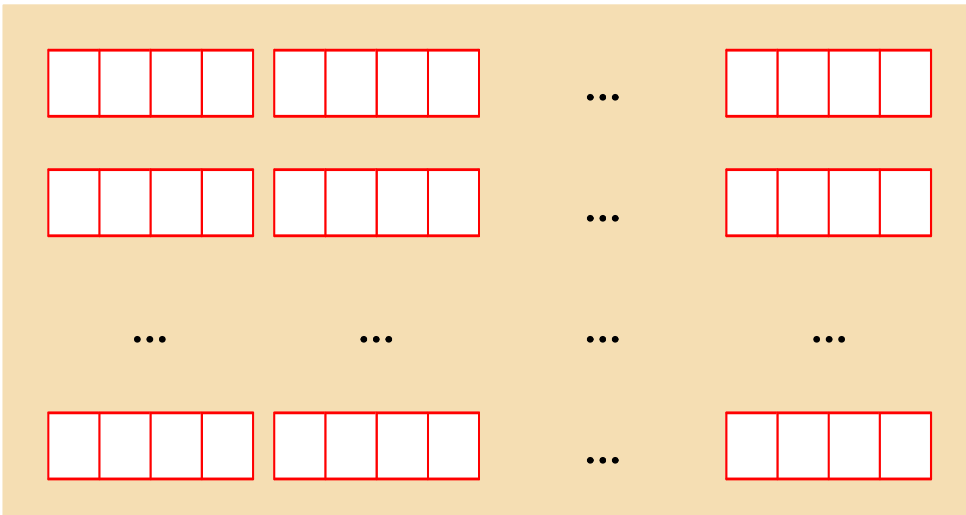


Address:

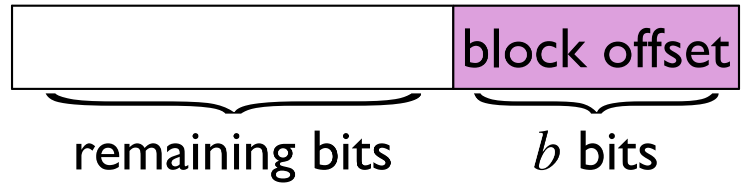
`0x765428`

Finding a Cache Entry

Block size $B = 2^b$

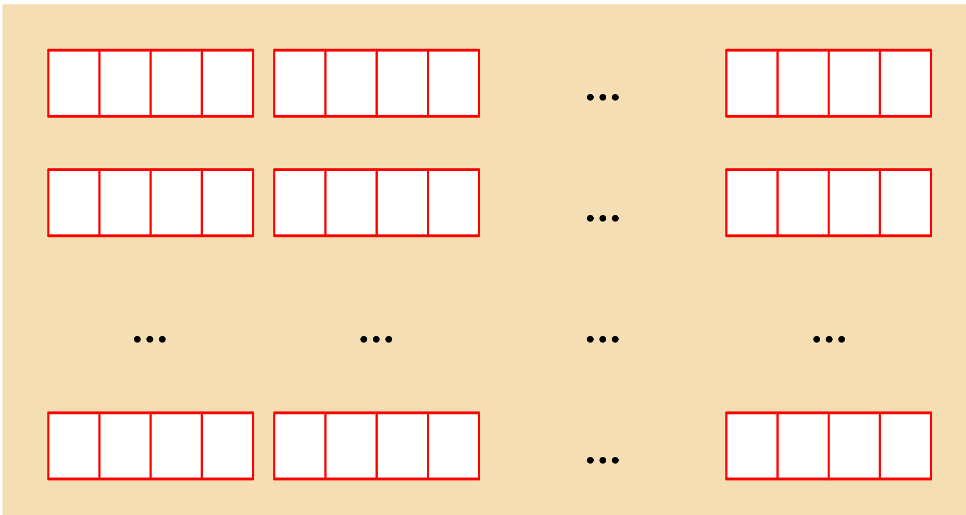


Address:



Finding a Cache Entry

Block size $B = 2^b$



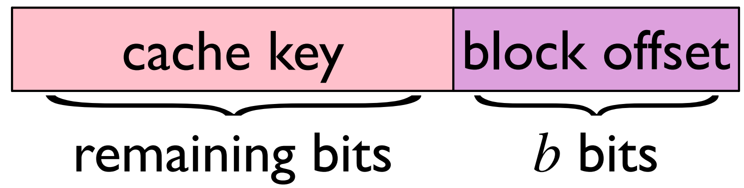
Address:

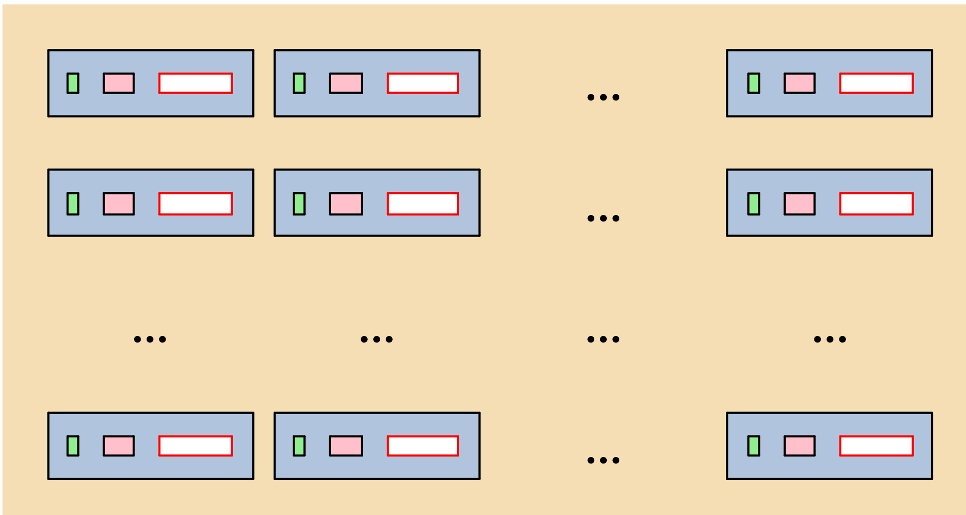
$\underbrace{\text{cache key}}_{\text{remaining bits}}$ $\underbrace{\text{block offset}}_{b \text{ bits}}$

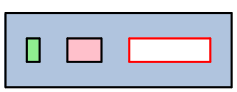
Finding a Cache Entry

Block size $B = 2^b$

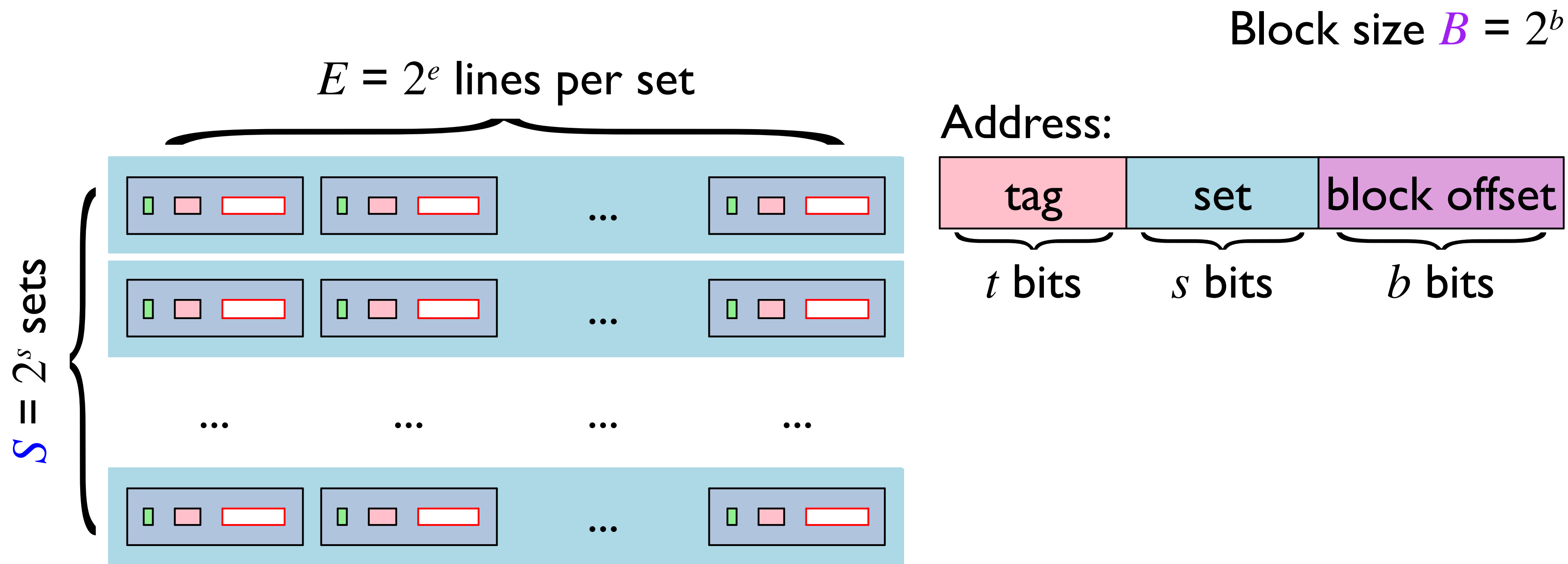
Address:



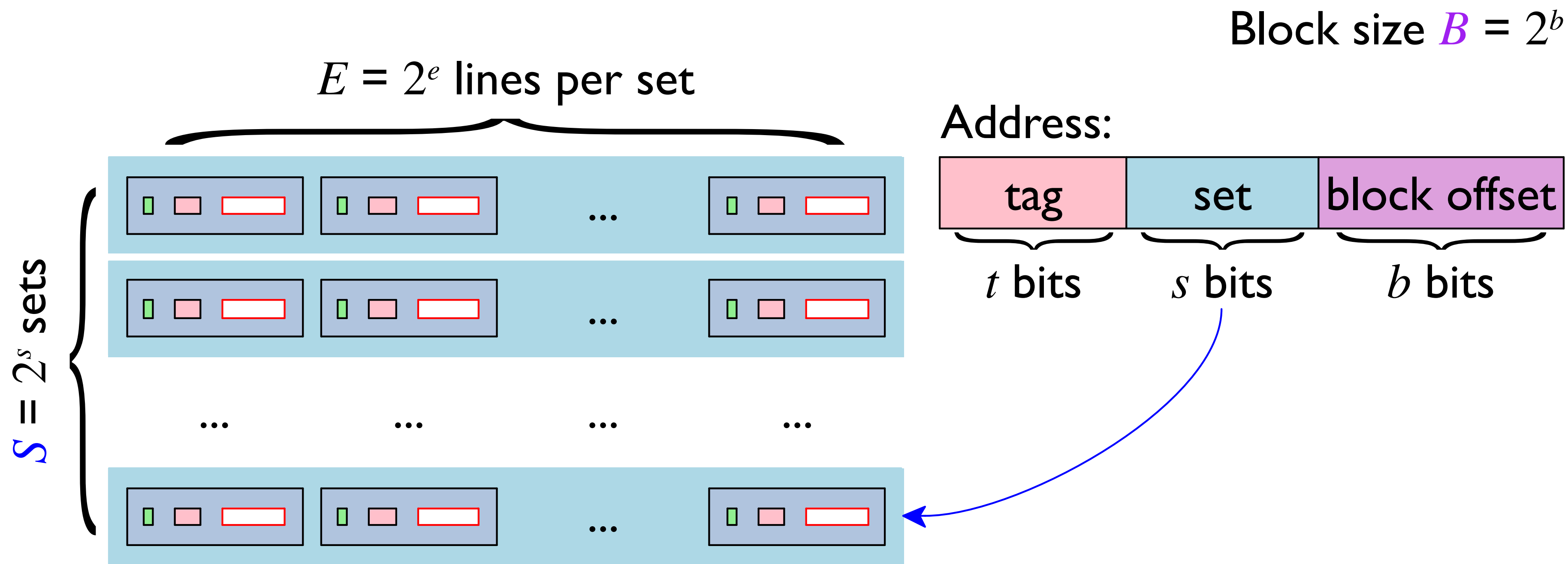



= ***cache line***

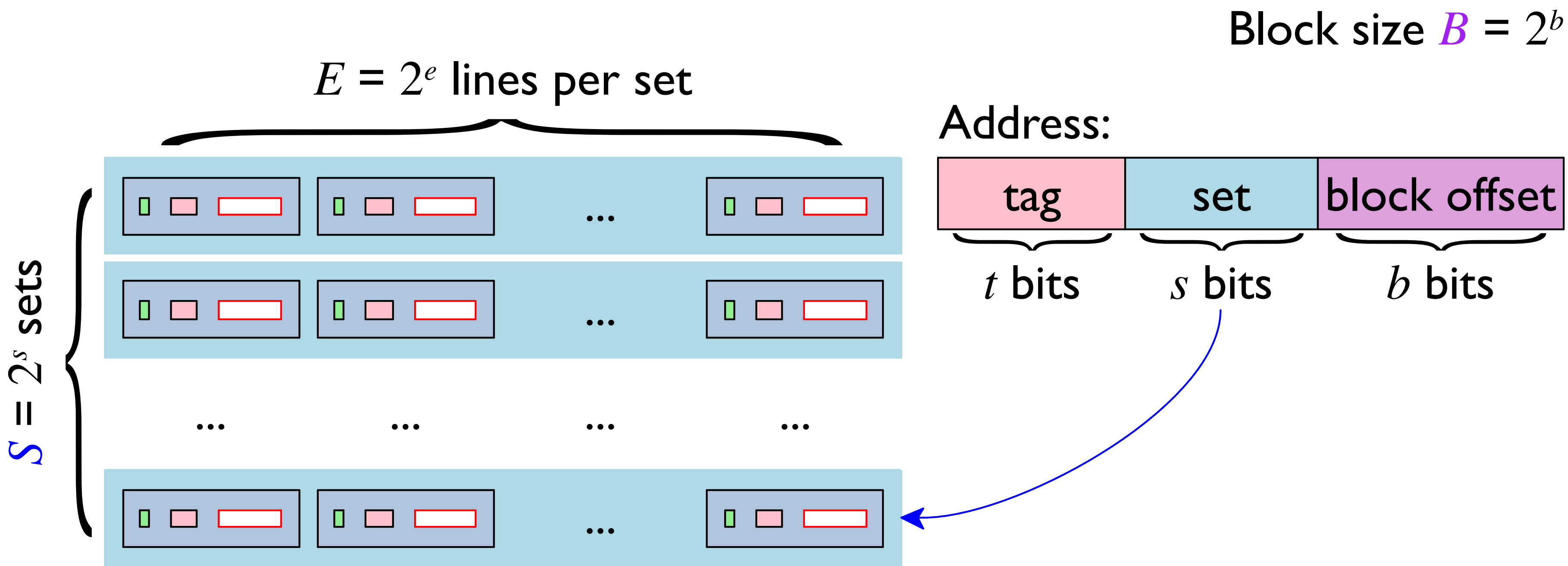
Finding a Cache Entry



Finding a Cache Entry



Finding a Cache Entry

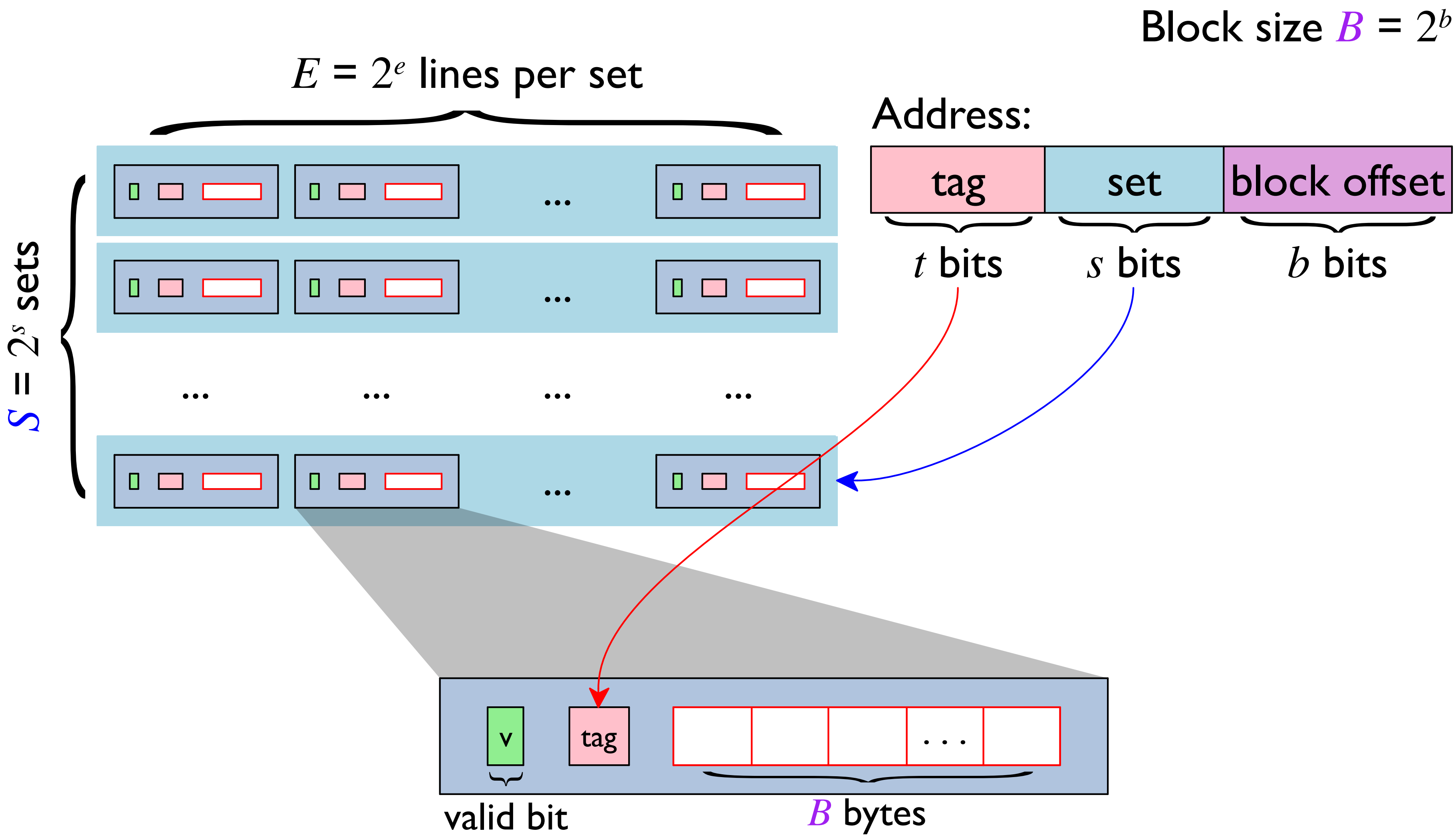


***E*-way associative cache**

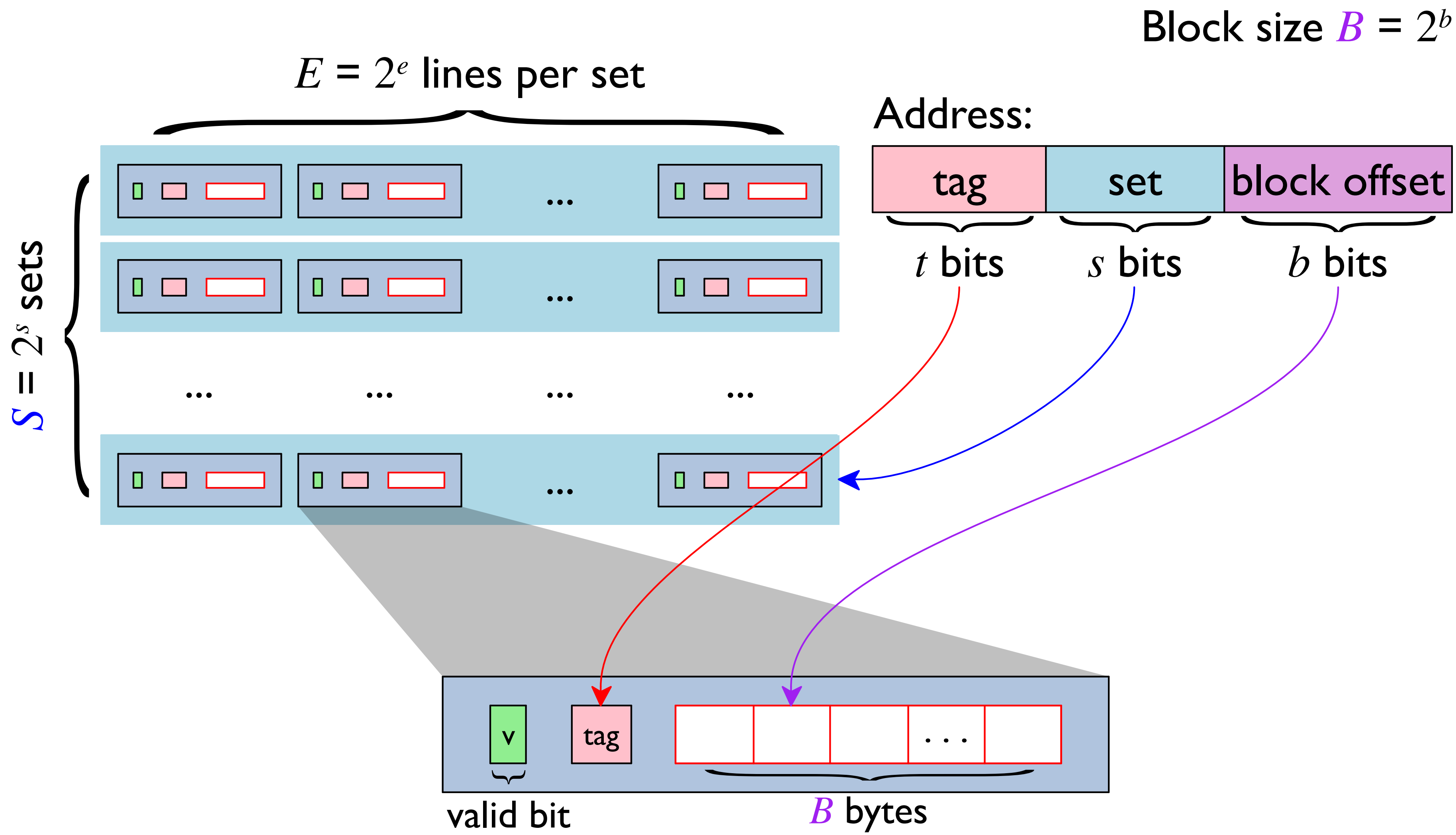
E = 1: ***direct-mapped cache***

Cache size $C = E \times S \times B$

Finding a Cache Entry



Finding a Cache Entry

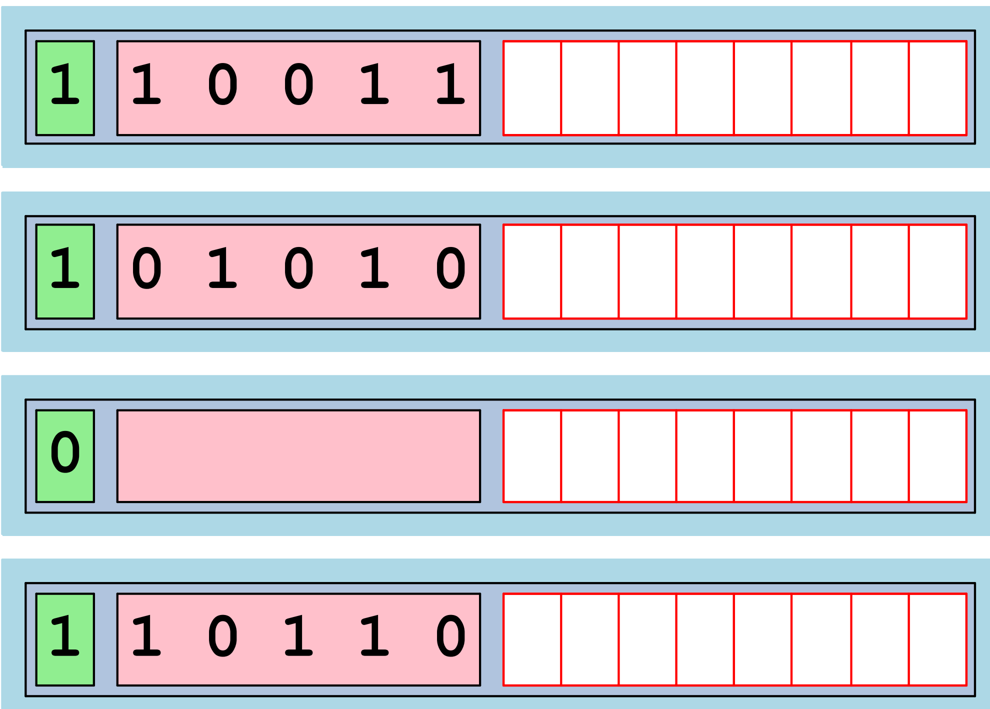


Cache Lookup Examples

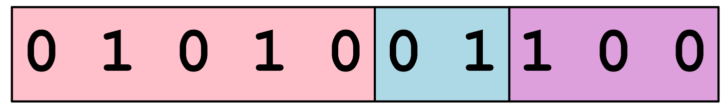
B = 8

S = 4

E = 1 (direct-mapped)



Address:

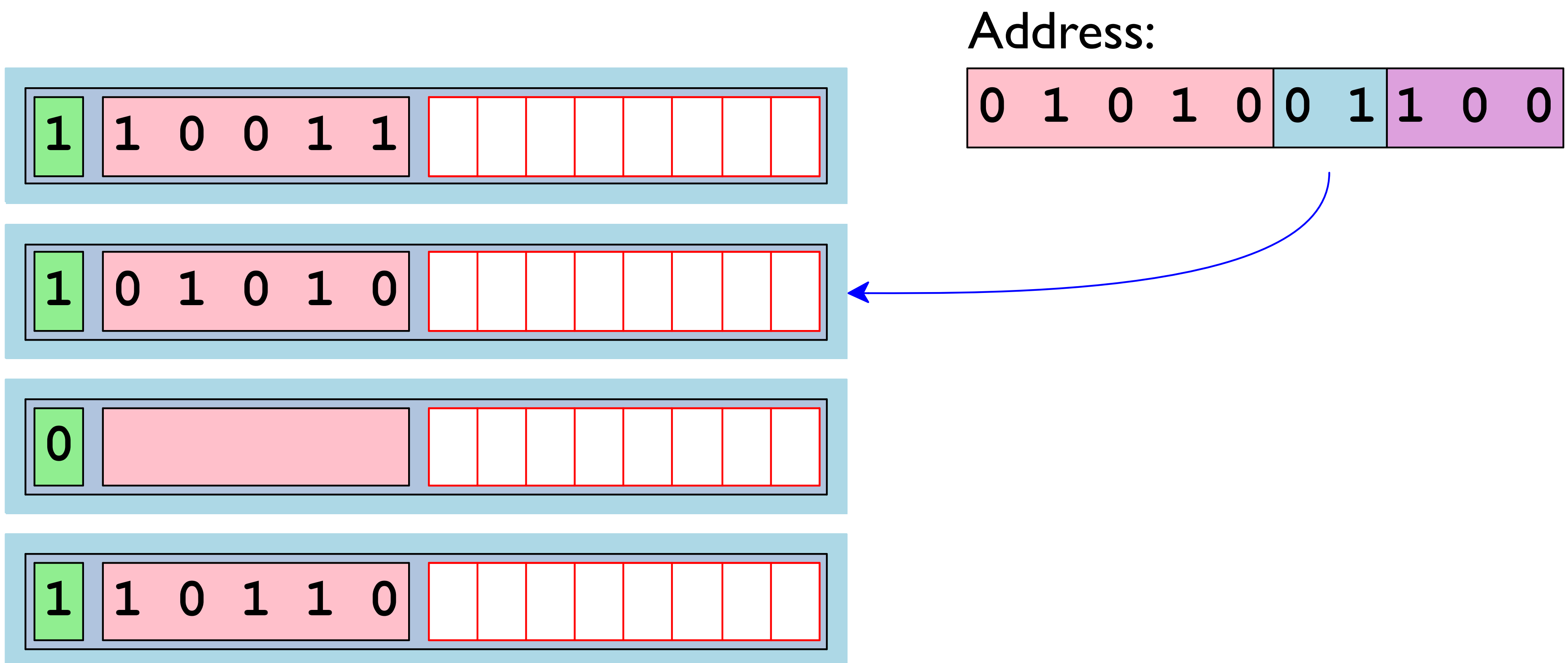


Cache Lookup Examples

$B = 8$

$S = 4$

$E = 1$ (direct-mapped)



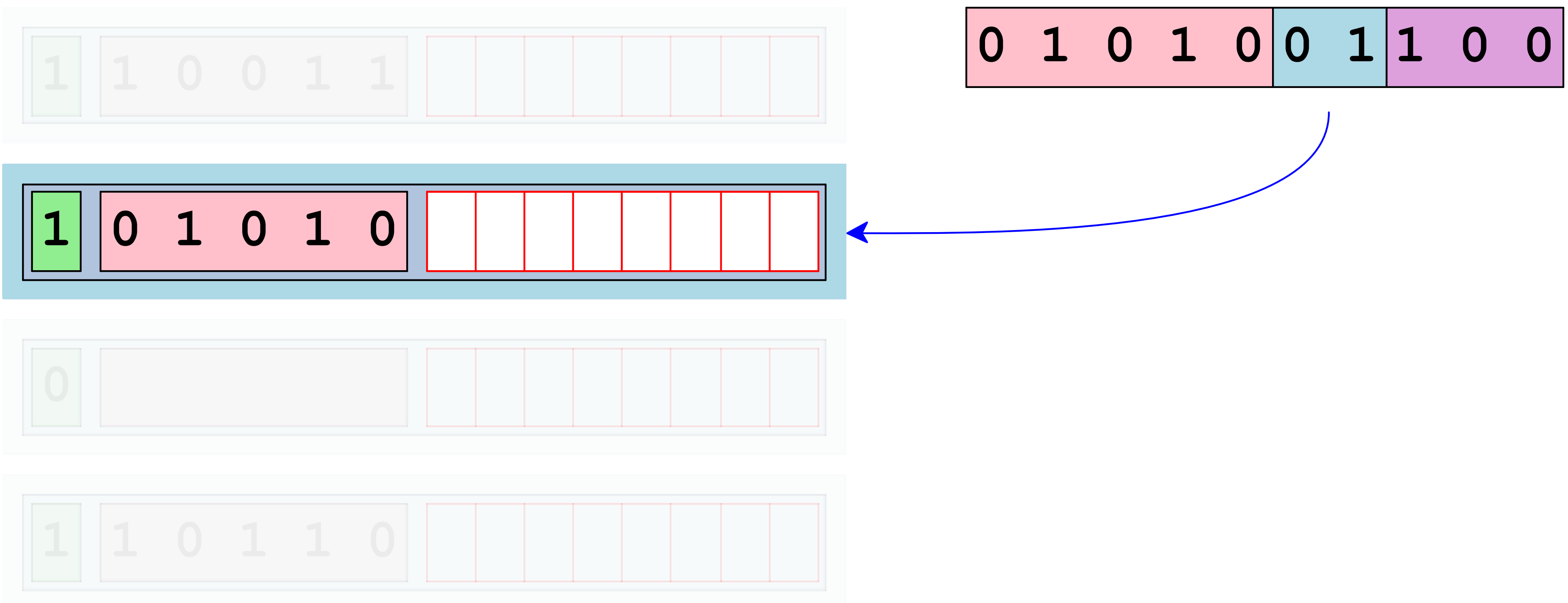
Cache Lookup Examples

$B = 8$

$S = 4$

$E = 1$ (direct-mapped)

Address:

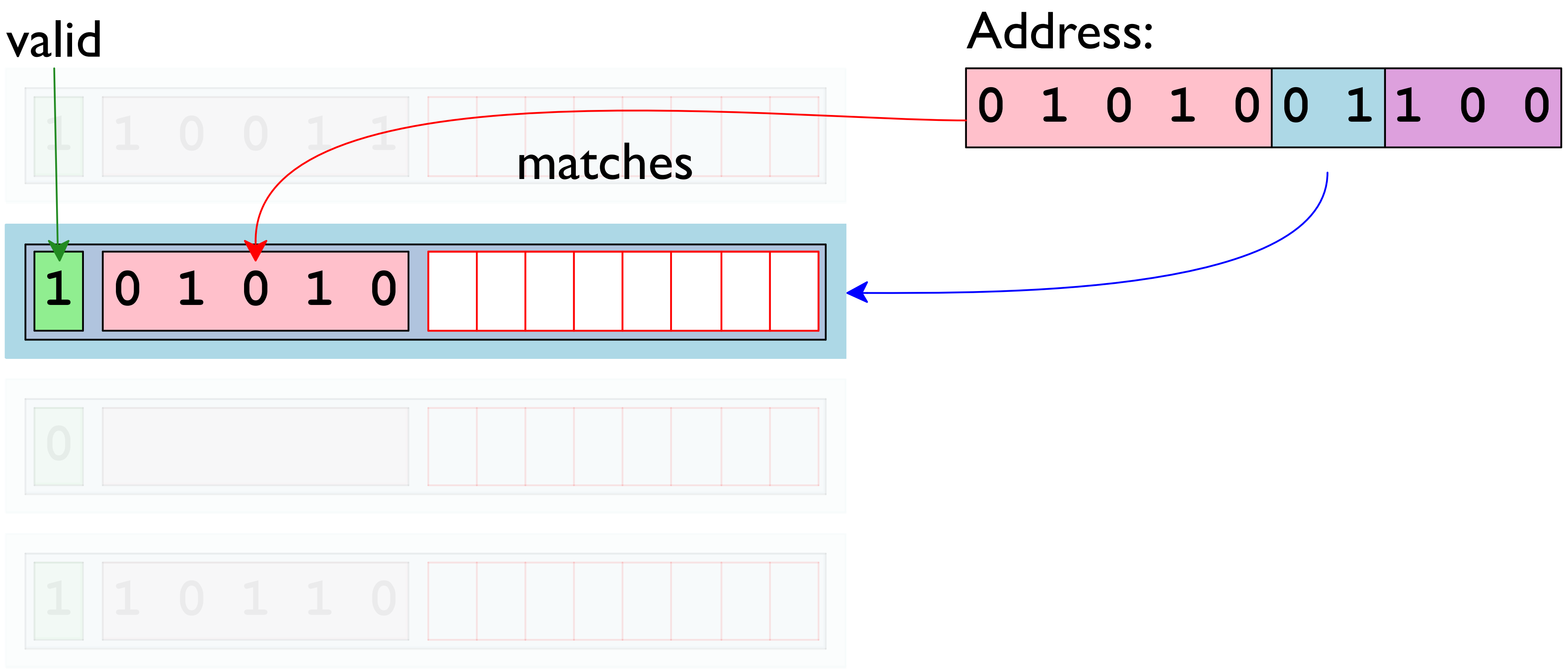


Cache Lookup Examples

$B = 8$

$S = 4$

$E = 1$ (direct-mapped)

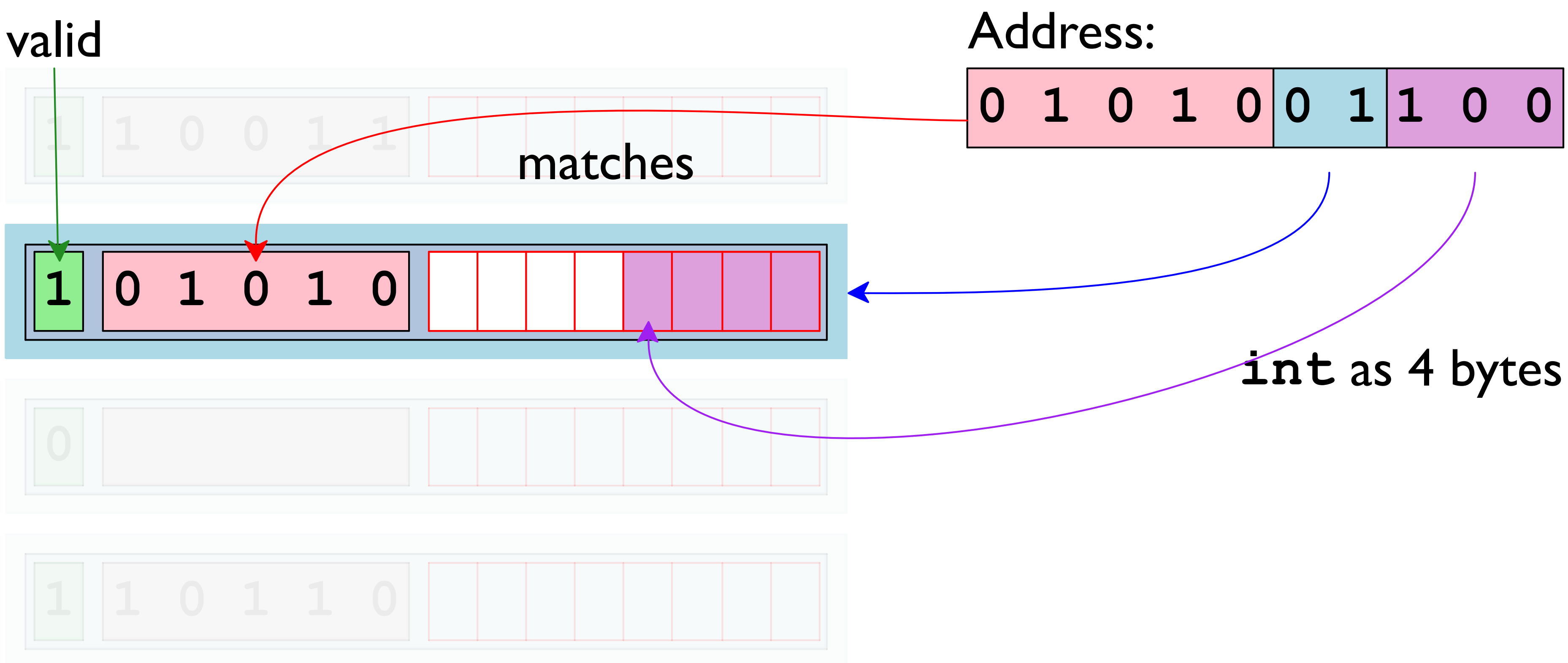


Cache Lookup Examples

$B = 8$

$S = 4$

$E = 1$ (direct-mapped)

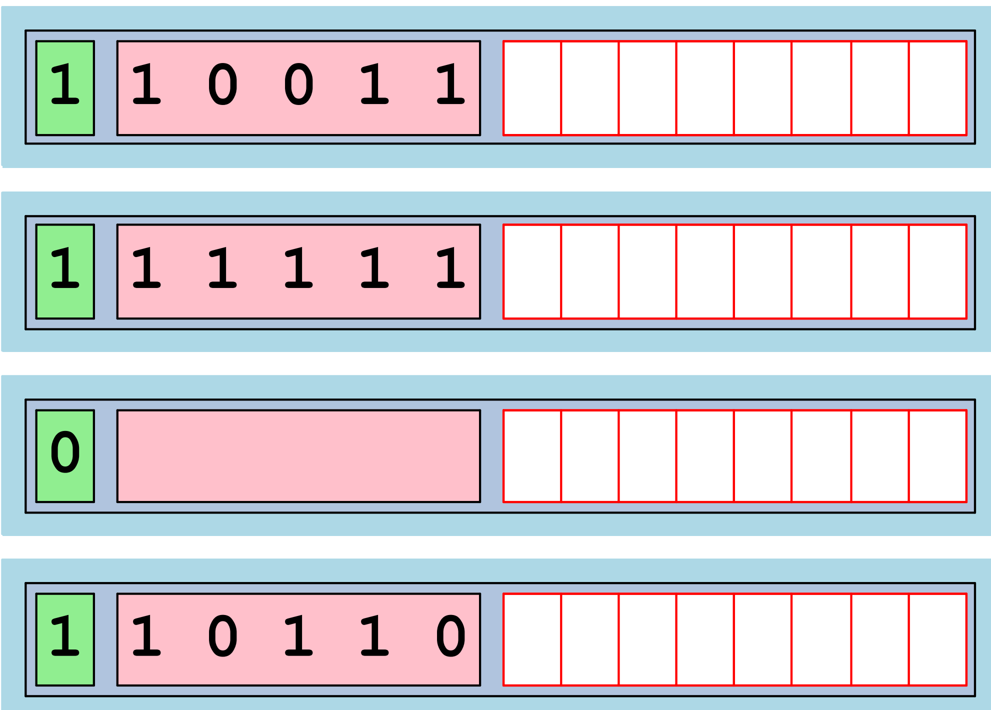


Cache Lookup Examples

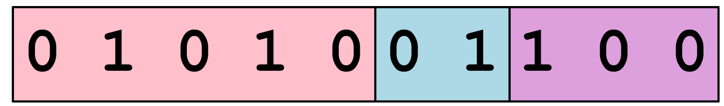
B = 8

S = 4

E = 1 (direct-mapped)



Address:

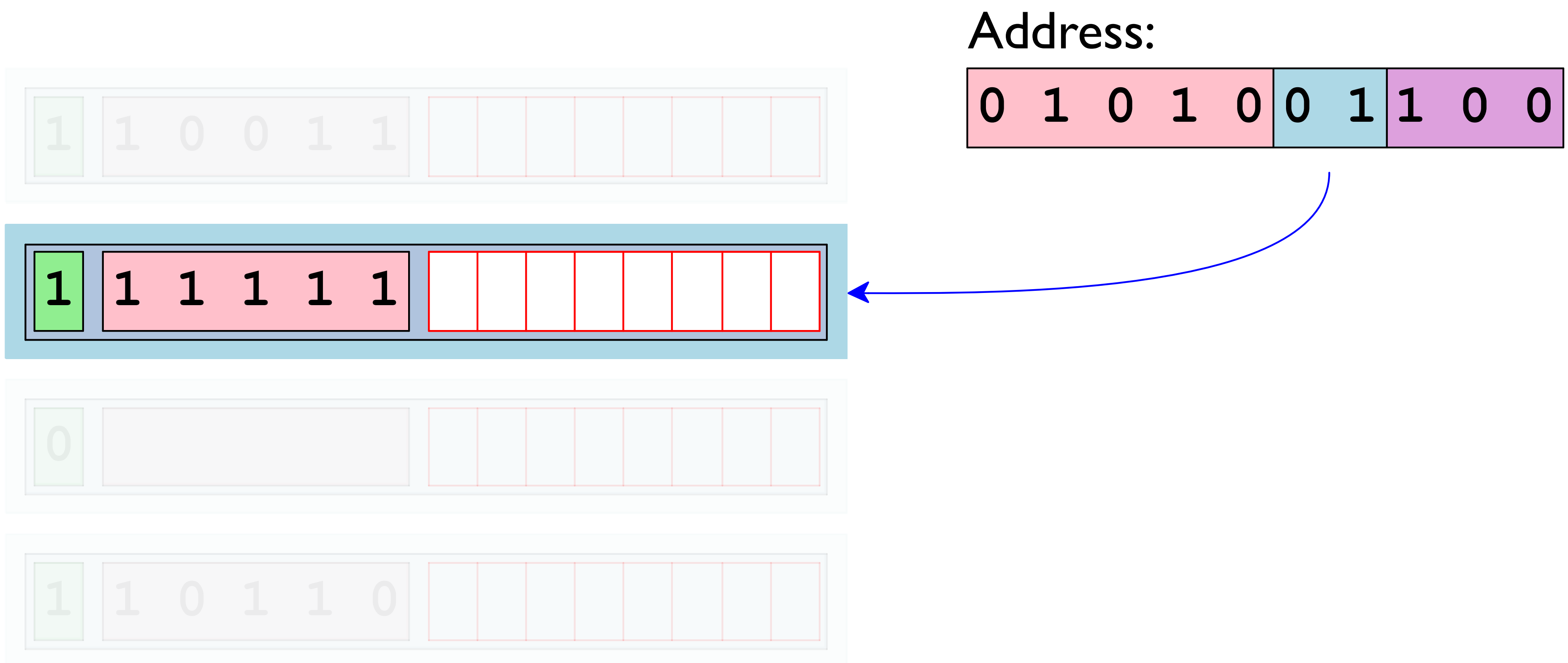


Cache Lookup Examples

B = 8

S = 4

E = 1 (direct-mapped)

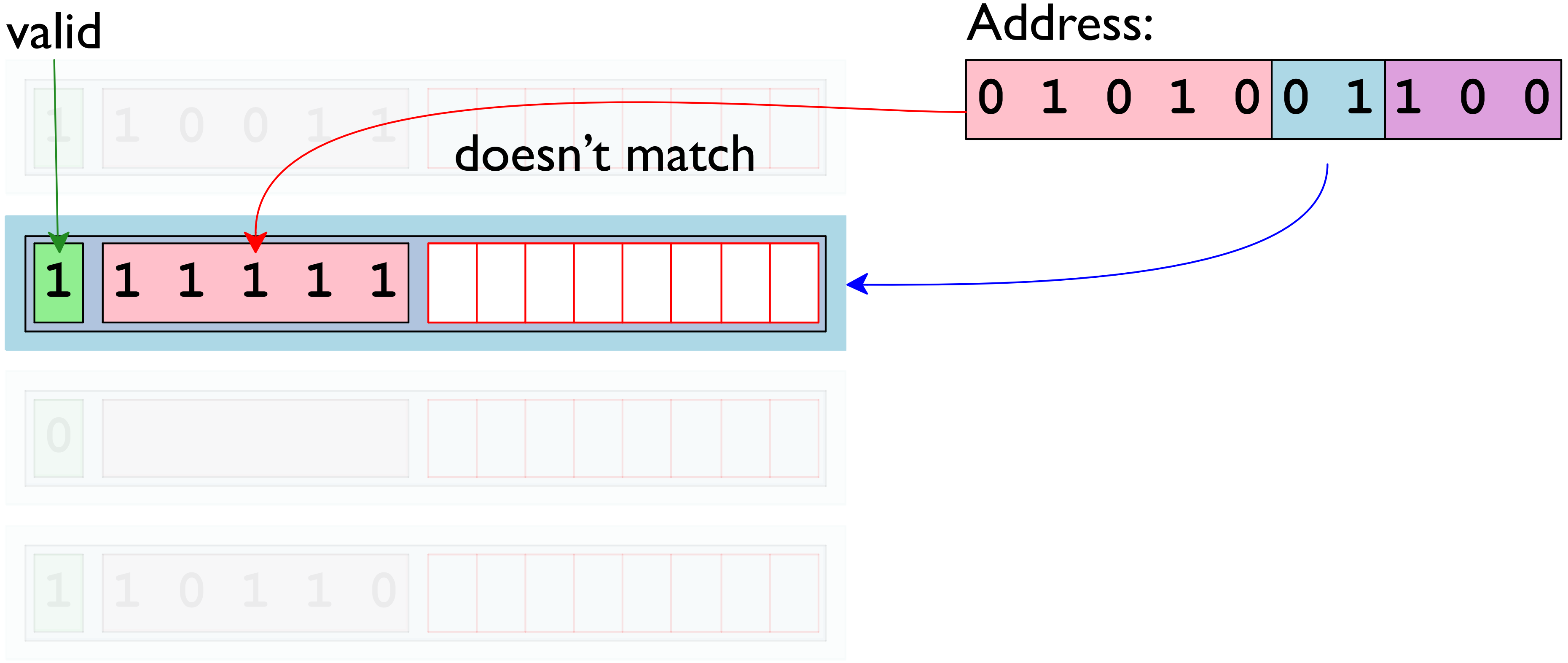


Cache Lookup Examples

B = 8

S = 4

E = 1 (direct-mapped)

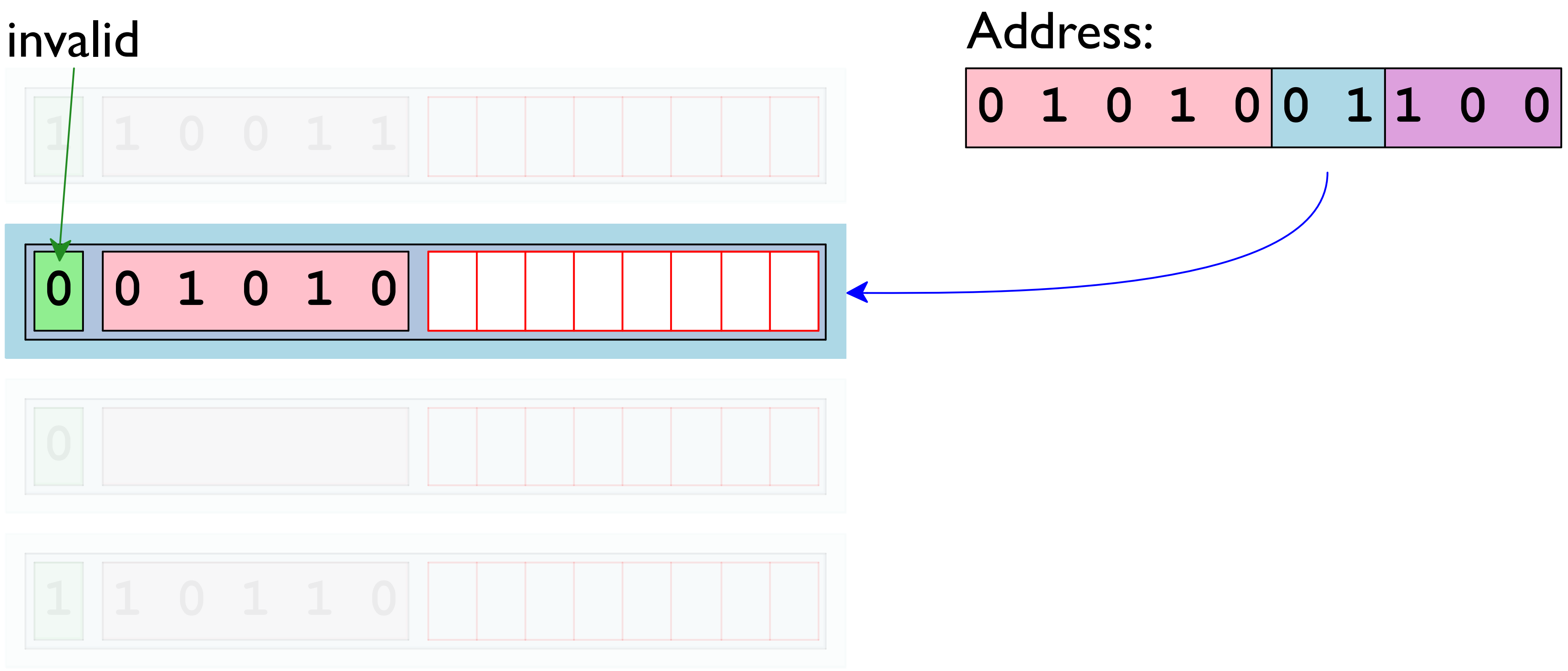


Cache Lookup Examples

B = 8

S = 4

E = 1 (direct-mapped)

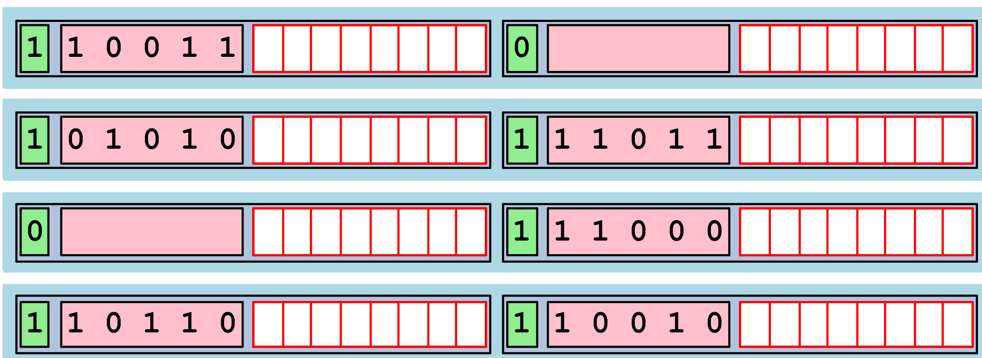


Cache Lookup Examples

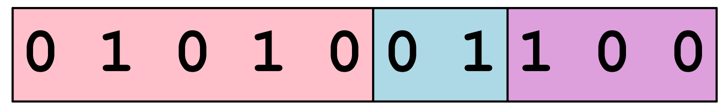
B = 8

S = 4

E = 2 (2-way)



Address:



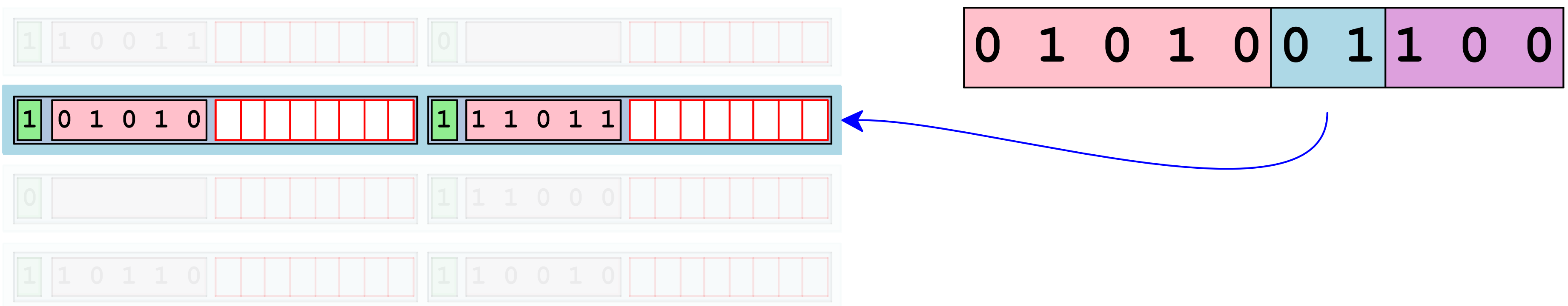
Cache Lookup Examples

$B = 8$

$S = 4$

$E = 2$ (2-way)

Address:

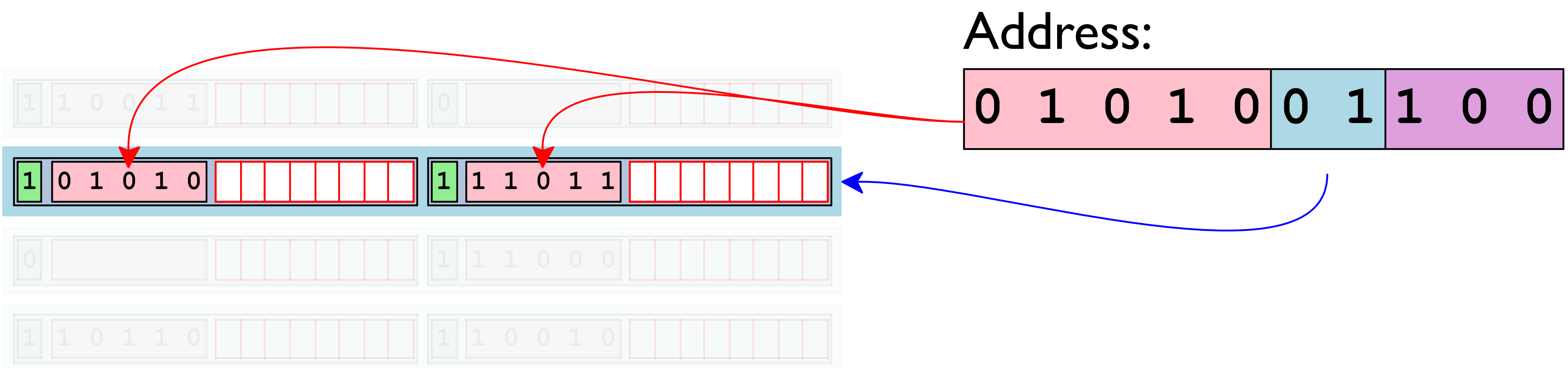


Cache Lookup Examples

B = 8

S = 4

E = 2 (2-way)

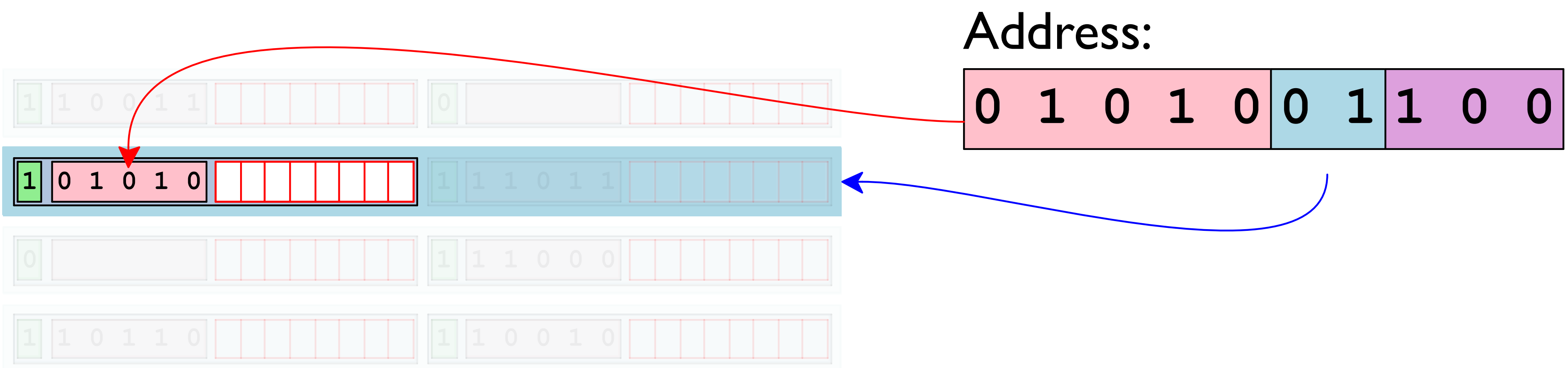


Cache Lookup Examples

$B = 8$

$S = 4$

$E = 2$ (2-way)



Cache Line Replacement

On a cache ***miss*** new data come in, old data gets lost

- random — pick a random cache line
- LRU — pick least-recently used line

Writes

Multiple copies of data exist

... in L1, L2, L3, Main Memory, Disk

What to do on a write-hit?

Write-through — write immediately to memory

Write-back — defer write to memory until replaced

What to do on a write-miss?

Write-allocate — load into cache, update in cache

No-write-allocate — write straight to memory

Typical implementations:

- Write-through + No-write-allocate
- Write-back + Write-allocate ← *good assumption*

Computing Cache Miss Rates

Cache configuration:

- Block size B = 32 bytes
- Sets S = 32
- Direct-mapped E = 1
- Size $C = E \times S \times B$ = 1024 bytes

```
struct {
  int x;
  int y;
} grid[16][16];
```

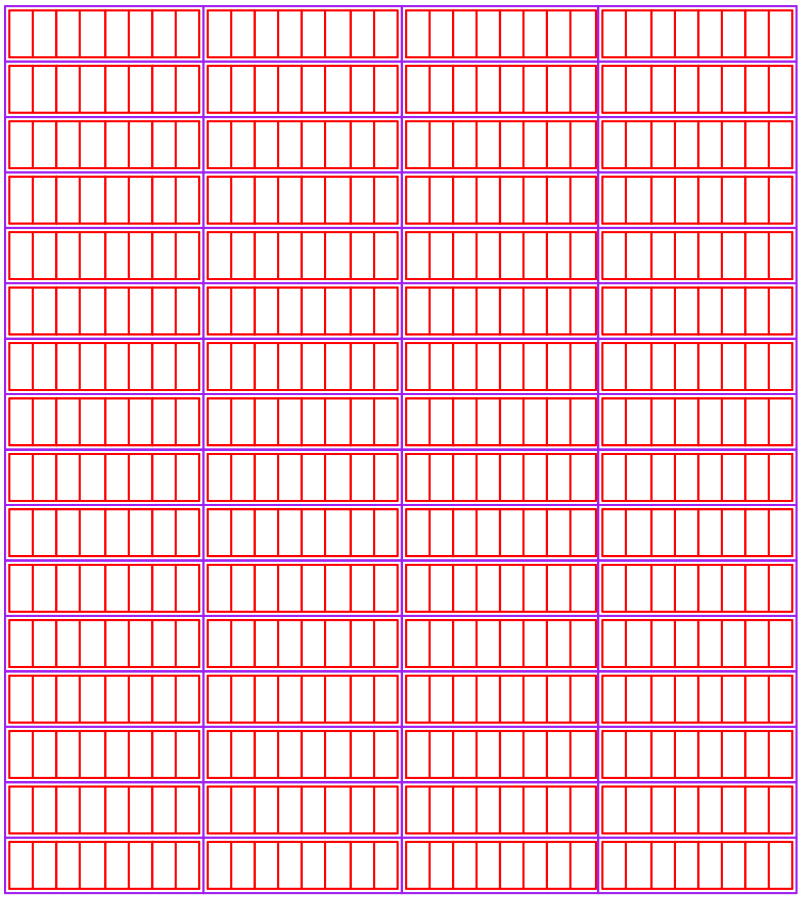
`sizeof(grid[0][0])` = **`2*sizeof(int)`** = 8

4 **`grid`** elements fit in a block/line

`sizeof(grid)` = **`16*16*2*sizeof(int)`** = 2048

Half of **`grid`** fits in the cache at a time

Computing Cache Miss Rates

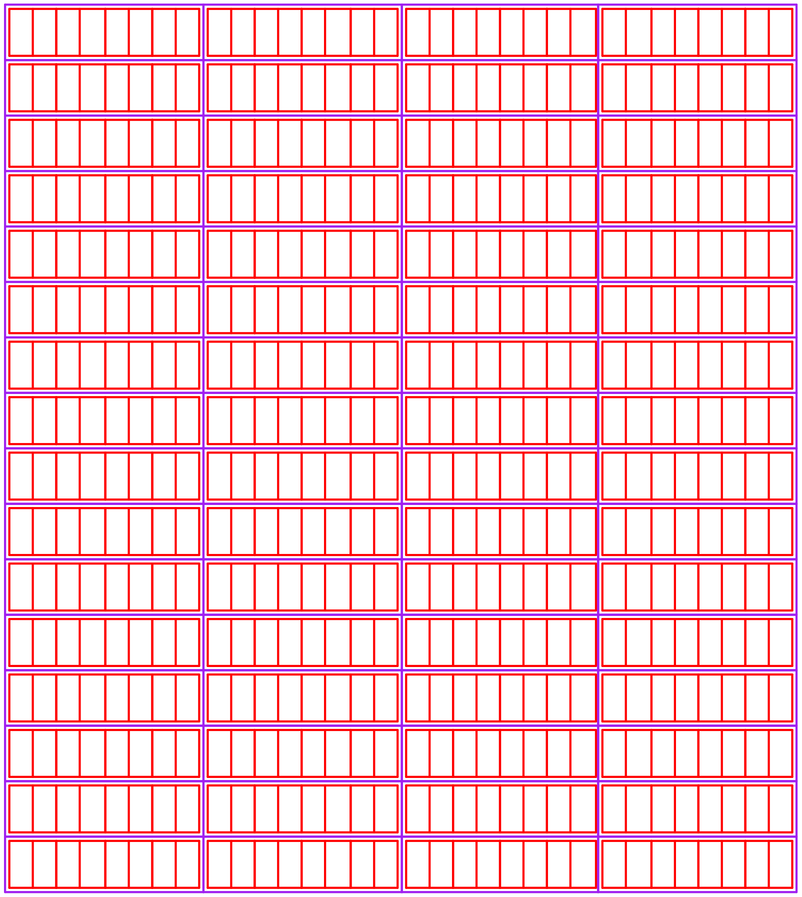


= **`grid`** element

= fits cache line

B = 32
S = 32
E = 1

Computing Cache Miss Rates



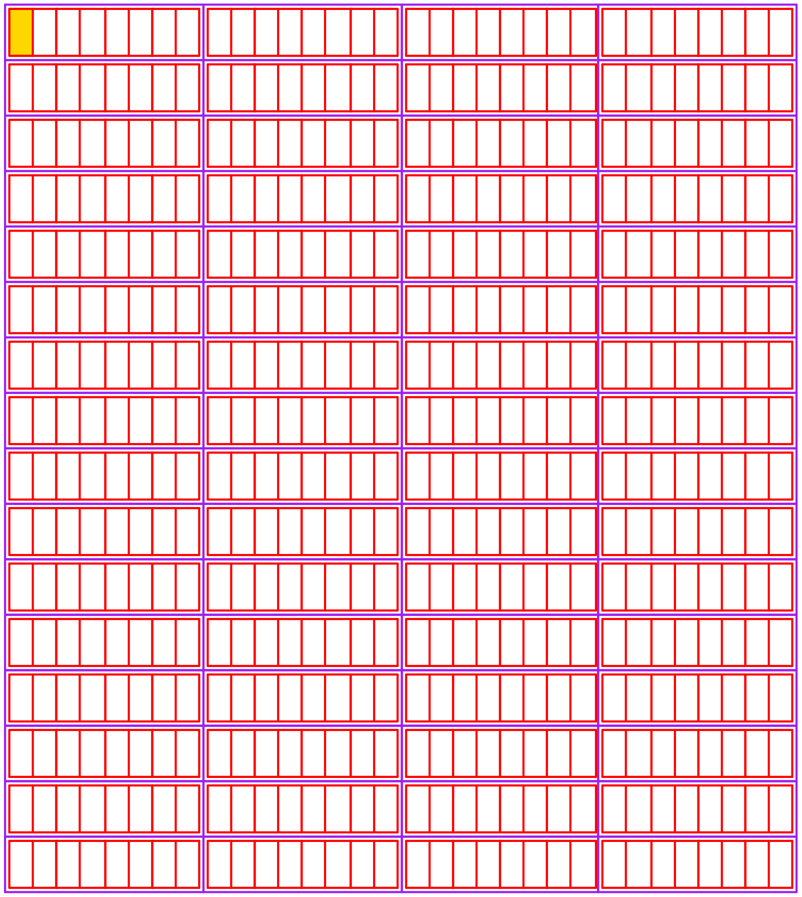
= **`grid`** element

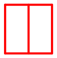
= fits cache line

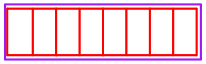
B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
```

Computing Cache Miss Rates



 = **`grid`** element

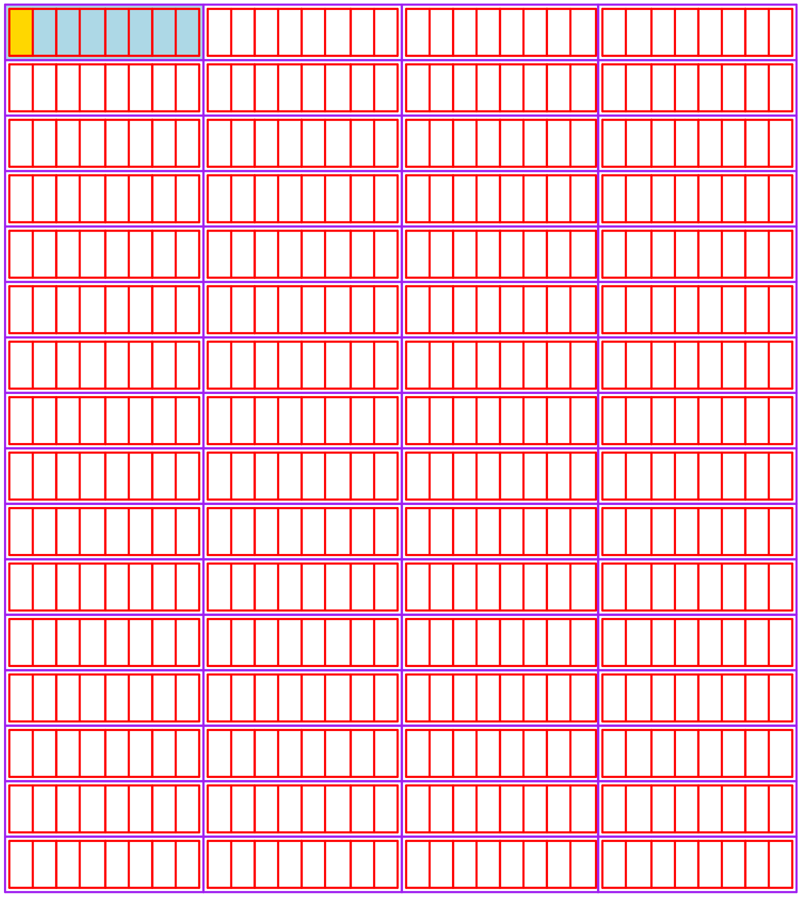
 = fits cache line

B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
```

`i` = 0 **`j`** = 0 miss

Computing Cache Miss Rates



= **`grid`** element

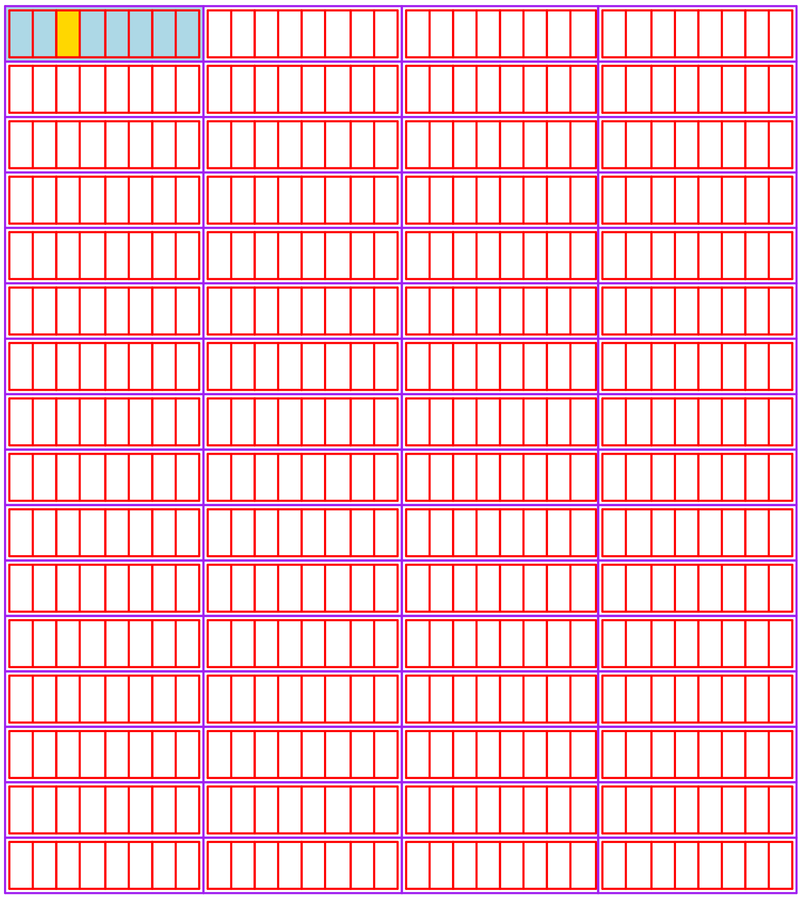
= fits cache line

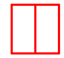
B = 32
S = 32
E = 1

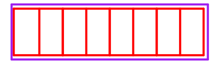
```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
```

`i` = 0 **`j`** = 0 miss

Computing Cache Miss Rates



 = **`grid`** element

 = fits cache line

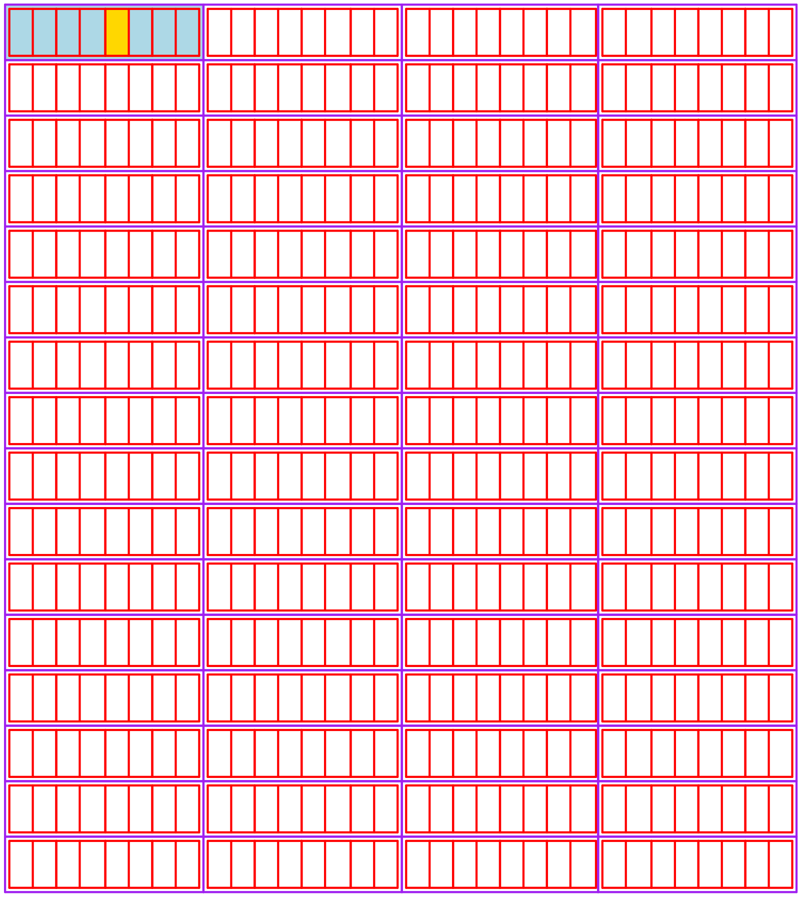
B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
```

`i` = 0 **`j`** = 0 miss

`i` = 0 **`j`** = 1 hit

Computing Cache Miss Rates



= **`grid`** element

= fits cache line

B = 32
S = 32
E = 1

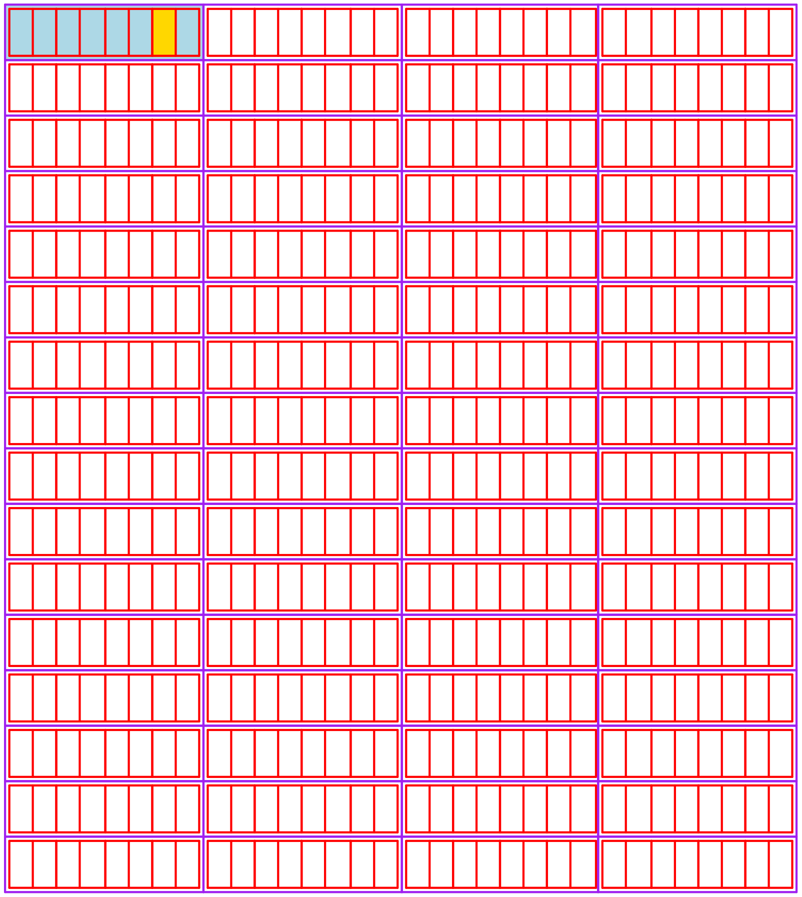
```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
```

`i` = 0 **`j`** = 0 miss

`i` = 0 **`j`** = 1 hit

`i` = 0 **`j`** = 2 hit

Computing Cache Miss Rates



= **`grid`** element

= fits cache line

B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
```

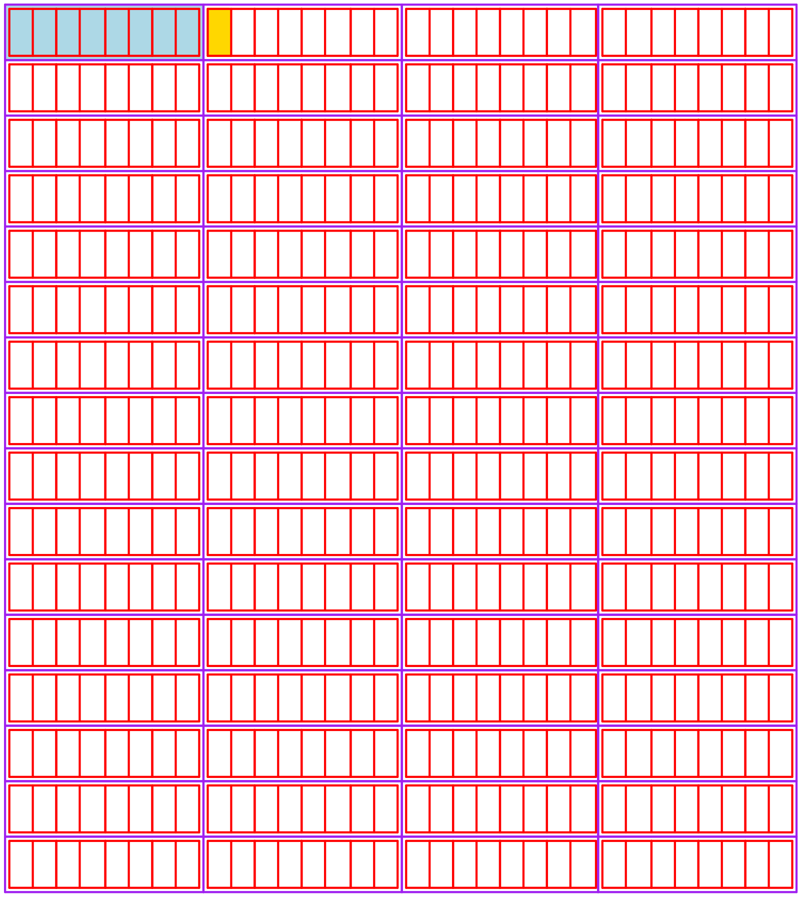
`i` = 0 `j` = 0 miss

`i` = 0 `j` = 1 hit

`i` = 0 `j` = 2 hit

`i` = 0 `j` = 3 hit

Computing Cache Miss Rates



= **grid** element

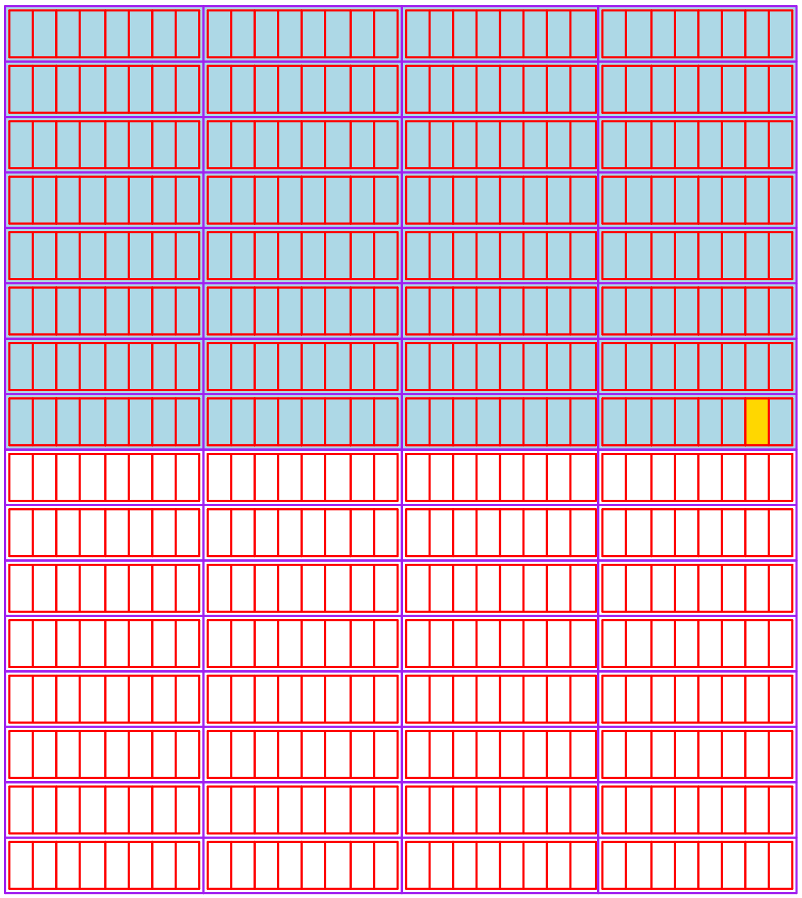
= fits cache line

B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
```

i = 0 **j** = 0 miss
i = 0 **j** = 1 hit
i = 0 **j** = 2 hit
i = 0 **j** = 3 hit
i = 0 **j** = 4 miss

Computing Cache Miss Rates



= **`grid`** element

= fits cache line

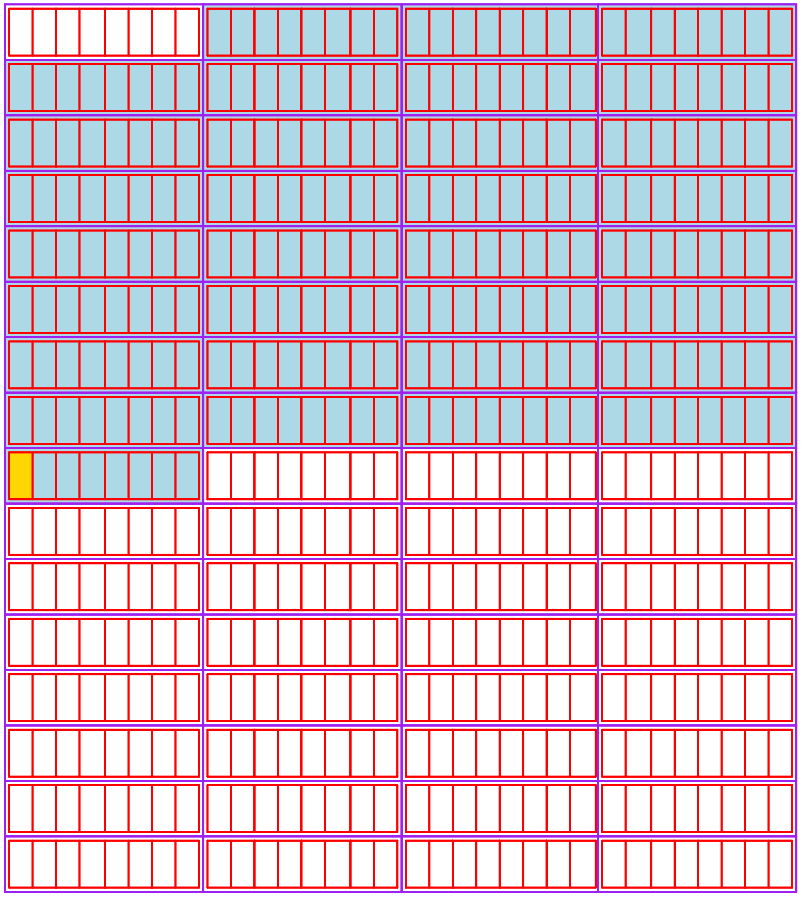
B = 32
S = 32
E = 1

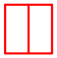
```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
```

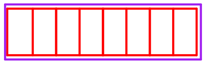
...

`i` = 7 `j` = 15 hit

Computing Cache Miss Rates



 = **`grid`** element

 = fits cache line

B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
```

...

`i` = 7 **`j`** = 15 hit

`i` = 8 **`j`** = 0 miss

Computing Cache Miss Rates

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31

= **`grid`** element

= fits cache line

B = 32
S = 32
E = 1

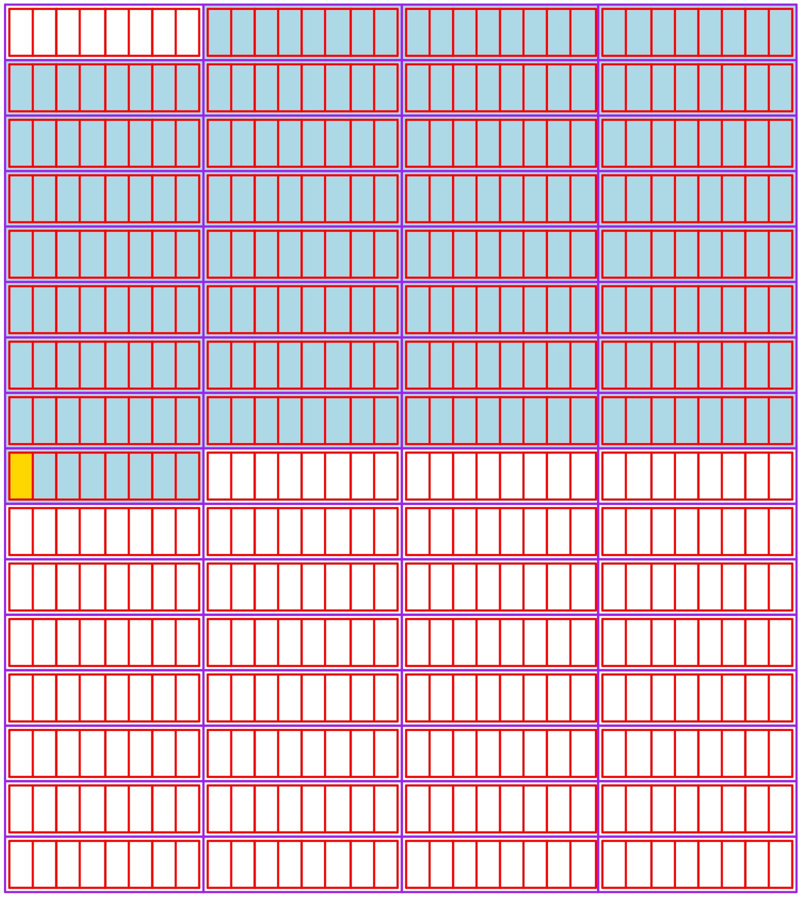
```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
```

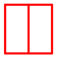
...

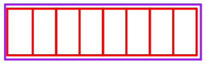
`i` = 7 `j` = 15 hit

`i` = 8 `j` = 0 miss

Computing Cache Miss Rates



 = **`grid`** element

 = fits cache line

B = 32
S = 32
E = 1

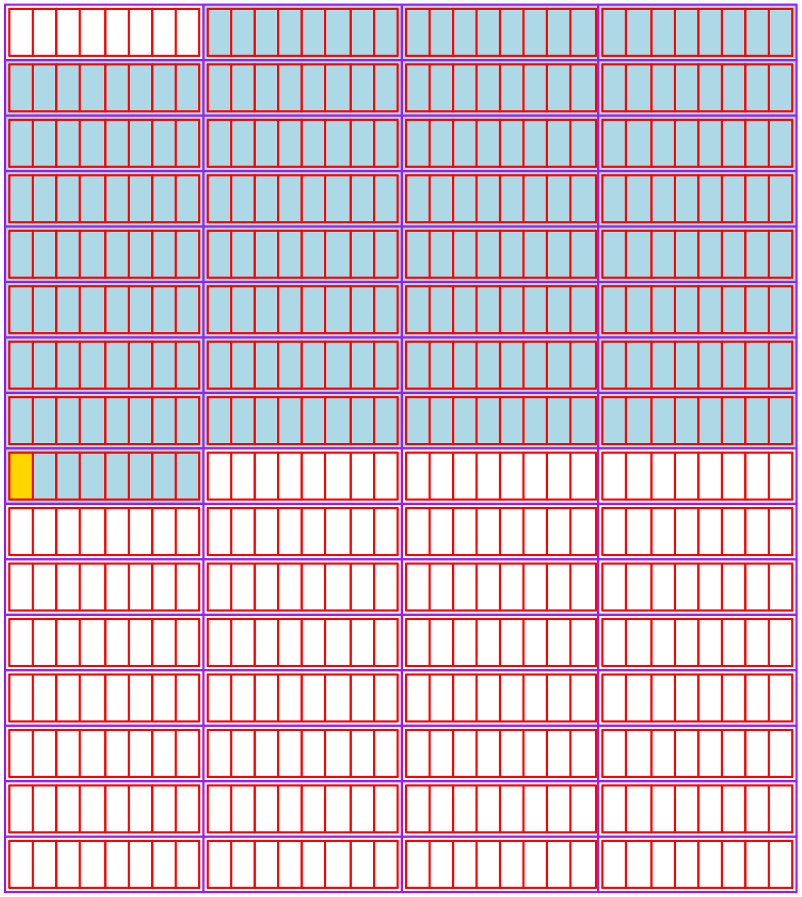
```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
```

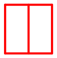
...

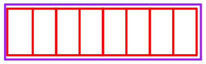
`i` = 7 `j` = 15 hit

`i` = 8 `j` = 0 miss

Computing Cache Miss Rates



 = **`grid`** element

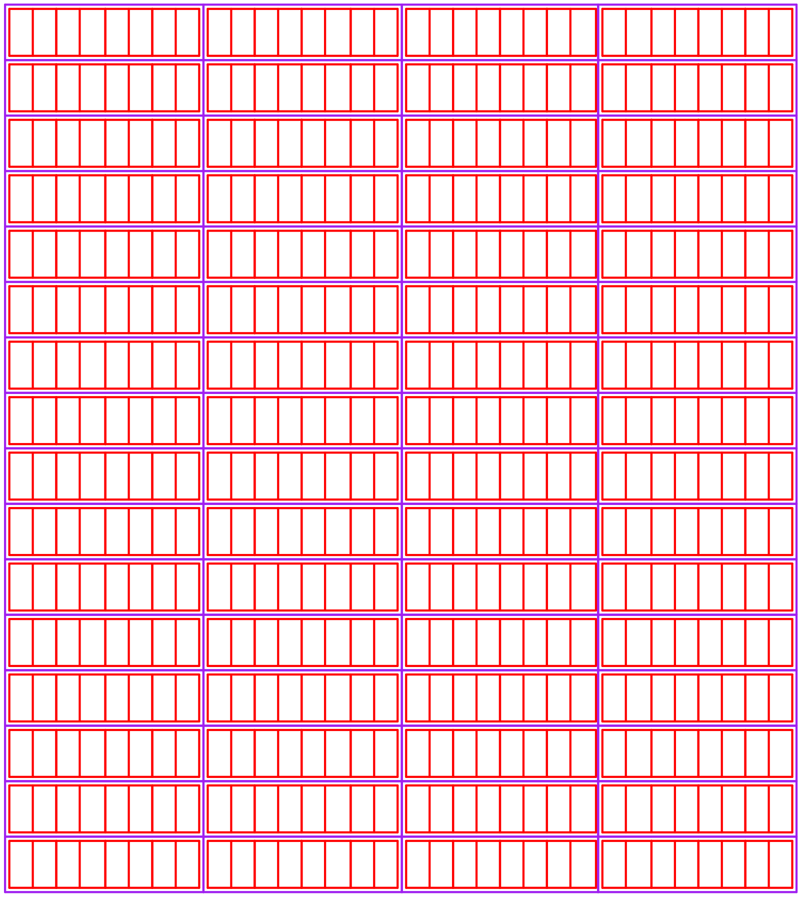
 = fits cache line

B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
```

Miss rate: 25%

Reverse Index Order



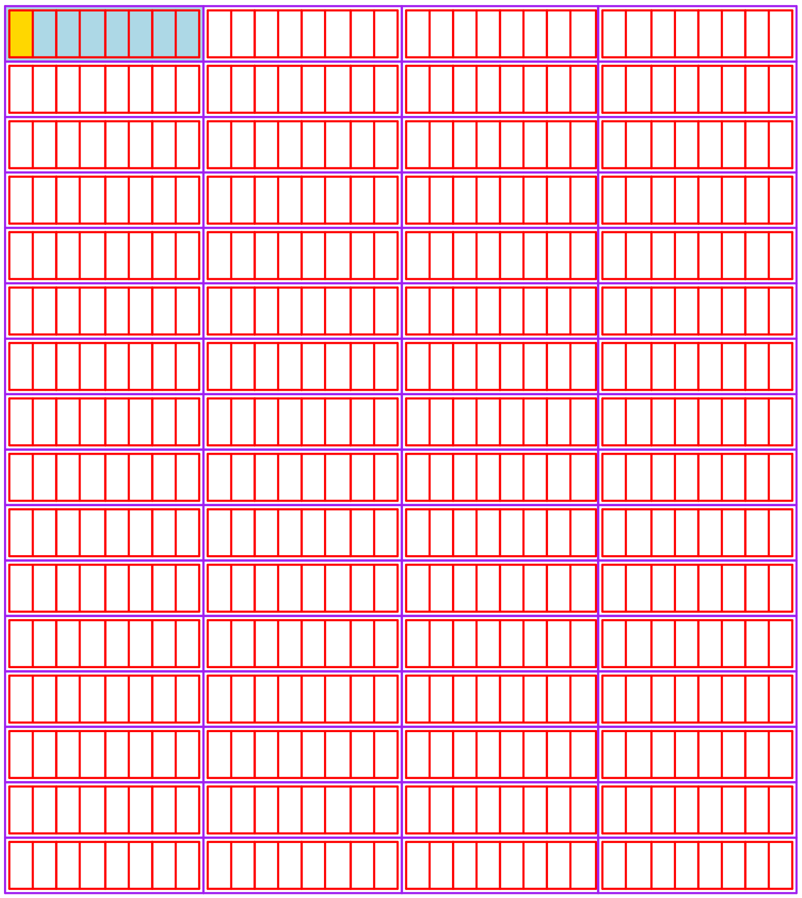
= **grid** element

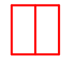
Move **j** to outer

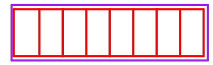
B = 32
S = 32
E = 1

```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

Reverse Index Order



 = `grid` element

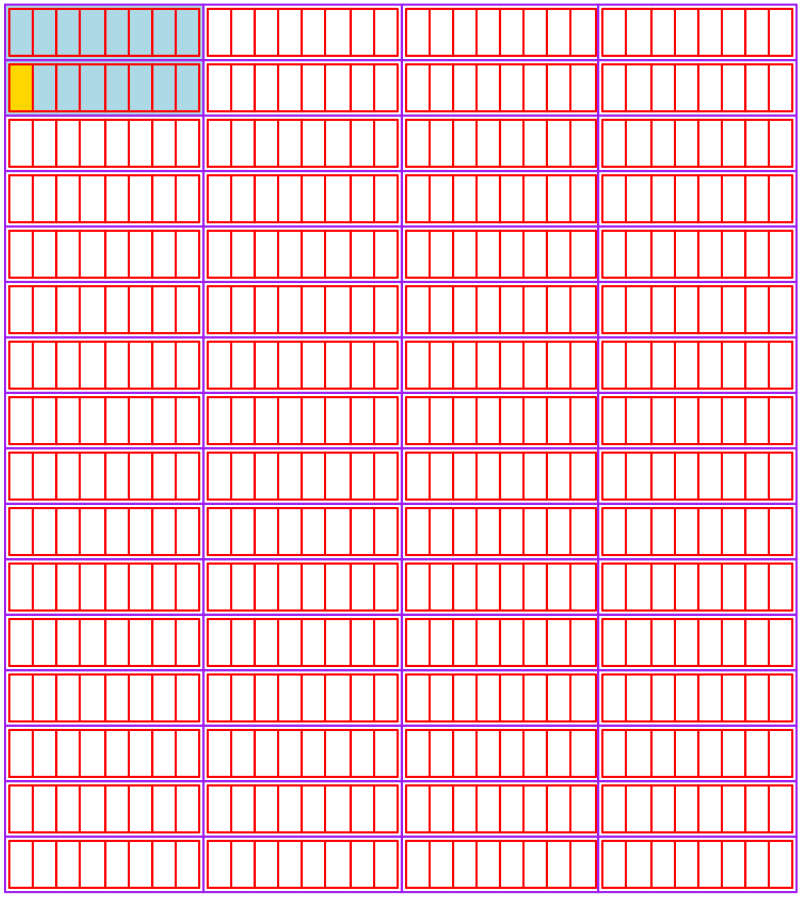
 = fits cache line

B = 32
S = 32
E = 1

```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

`i` = 0 `j` = 0 miss

Reverse Index Order



= `grid` element

= fits cache line

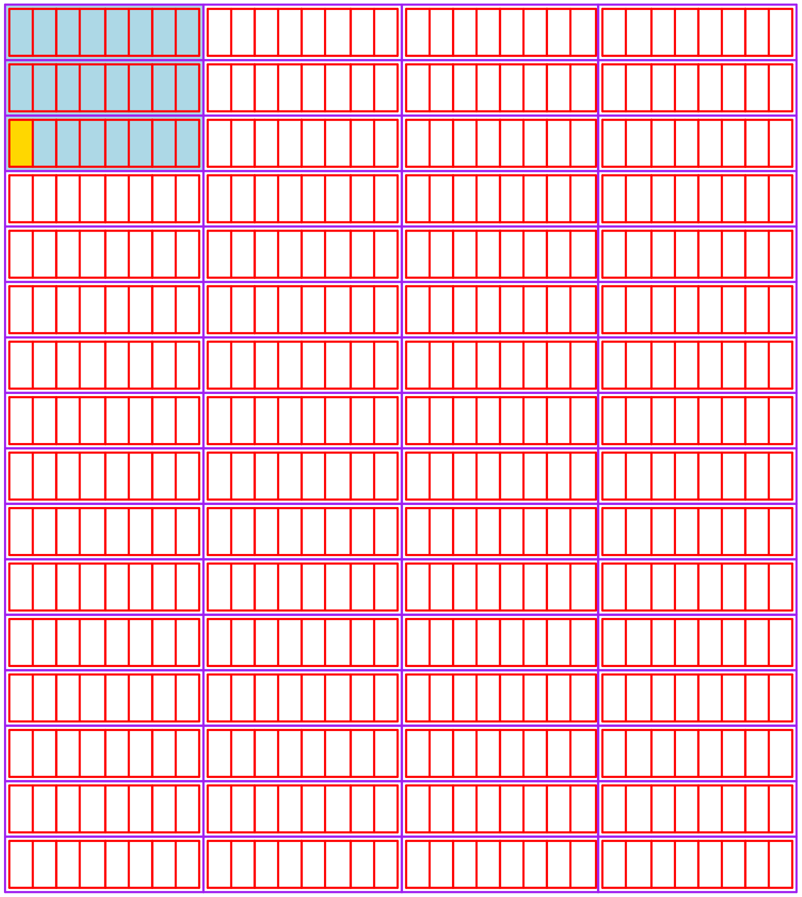
$B = 32$
$S = 32$
$E = 1$

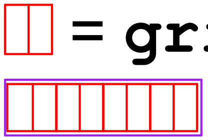
```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

`i` = 0 `j` = 0 miss

`i` = 1 `j` = 0 miss

Reverse Index Order





B = 32
S = 32
E = 1

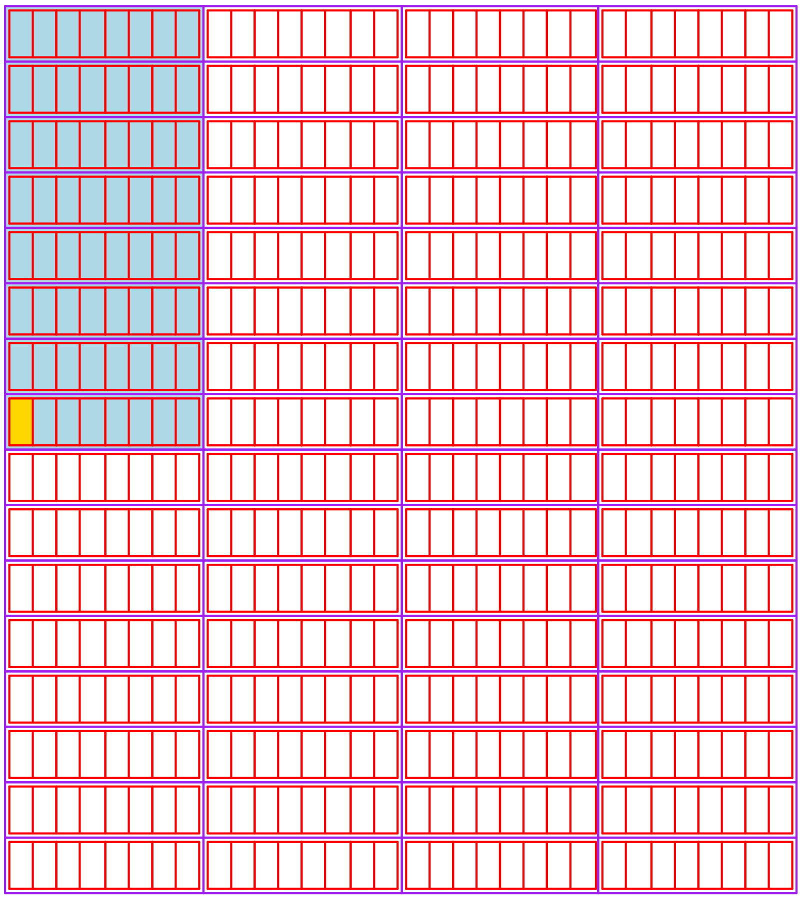
```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

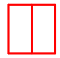
i = 0 **j** = 0 miss

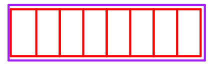
i = 1 **j** = 0 miss

i = 2 **j** = 0 miss

Reverse Index Order



 = `grid` element

 = fits cache line

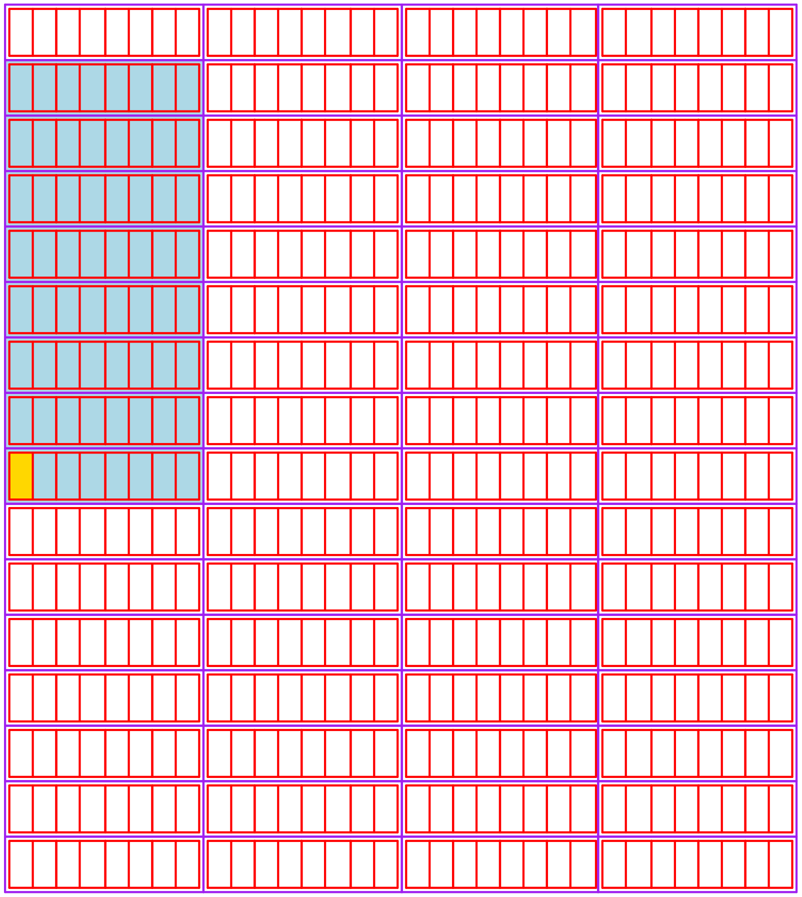
B = 32
S = 32
E = 1

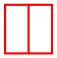
```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

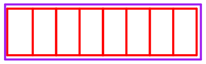
...

`i` = 7 `j` = 0 miss

Reverse Index Order



 = `grid` element

 = fits cache line

B = 32
S = 32
E = 1

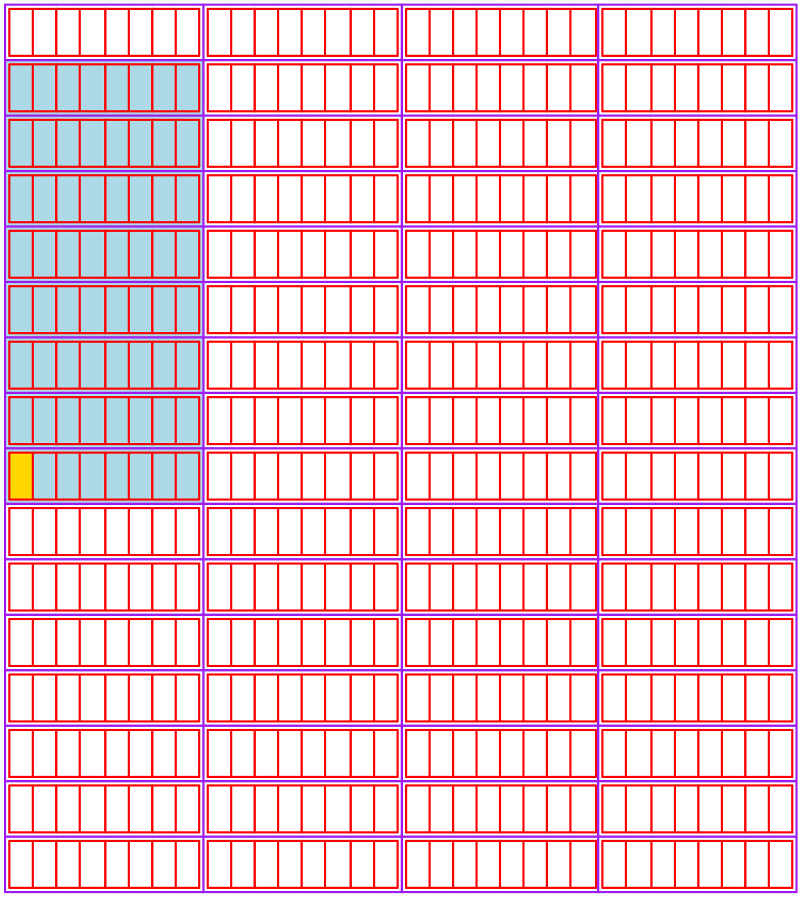
```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

...

`i` = 7 `j` = 0 miss

`i` = 8 `j` = 0 miss

Reverse Index Order



= **`grid`** element

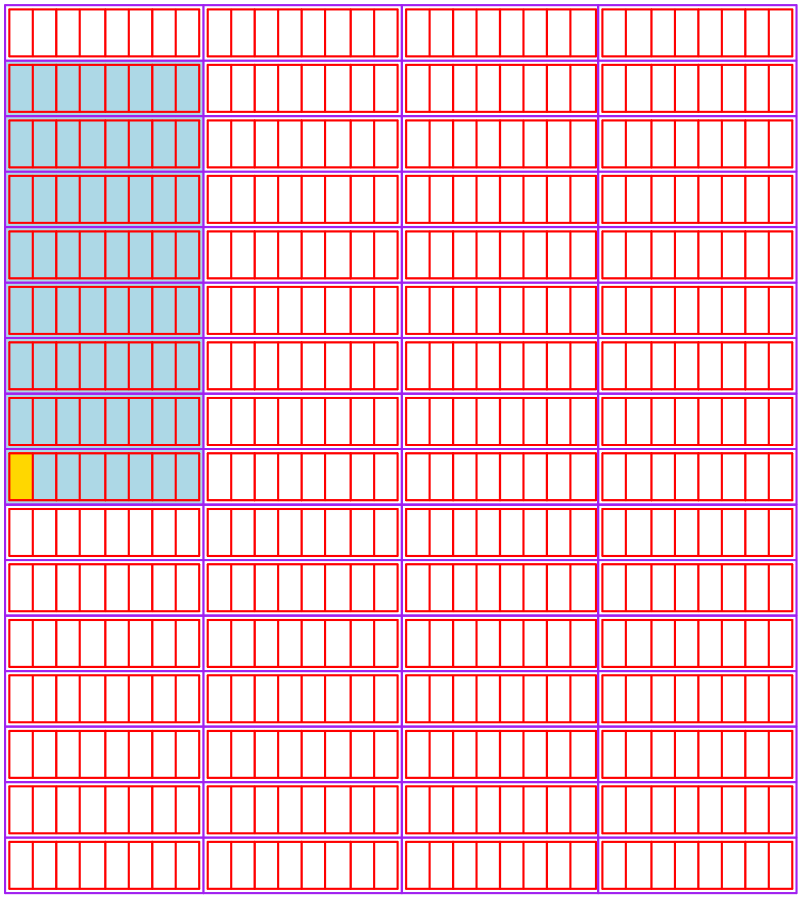
= fits cache line

B = 32
S = 32
E = 1

```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

Miss rate: 100%

Reverse Index Order



▯ = **`grid`** element

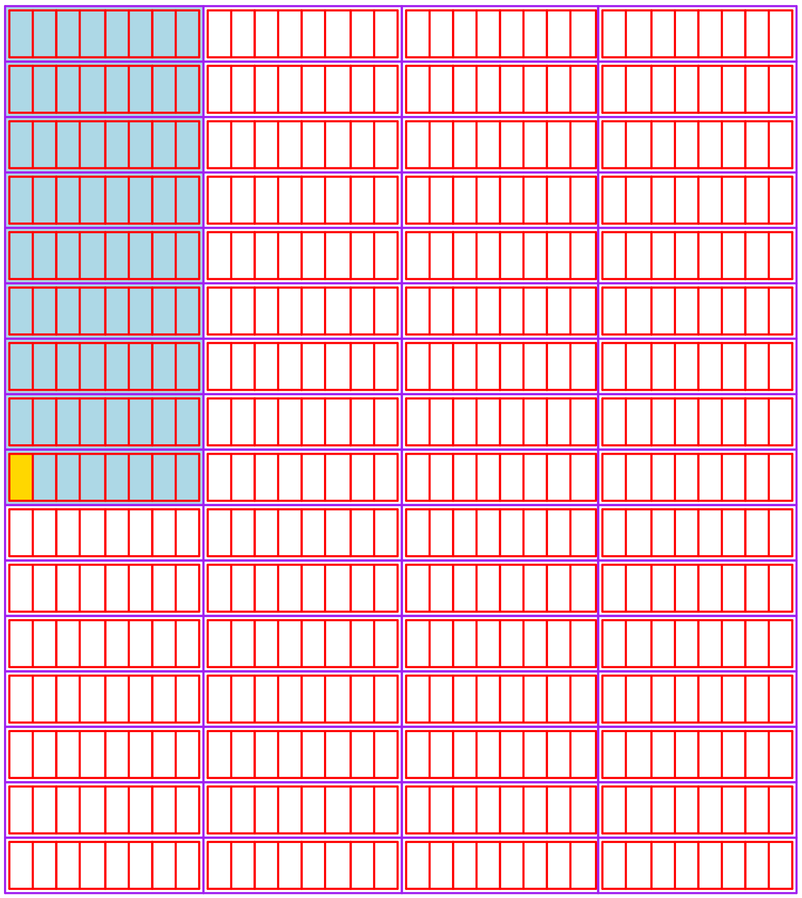
▯▯▯▯▯▯▯▯ = fits cache line

B = 32
S = 32
E = 1

```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

Traverse arrays ⇒
column as inner loop

When Data Fits in Cache



= `grid` element

= fits cache line

Larger cache

B = 32

S = **64**

E = 1

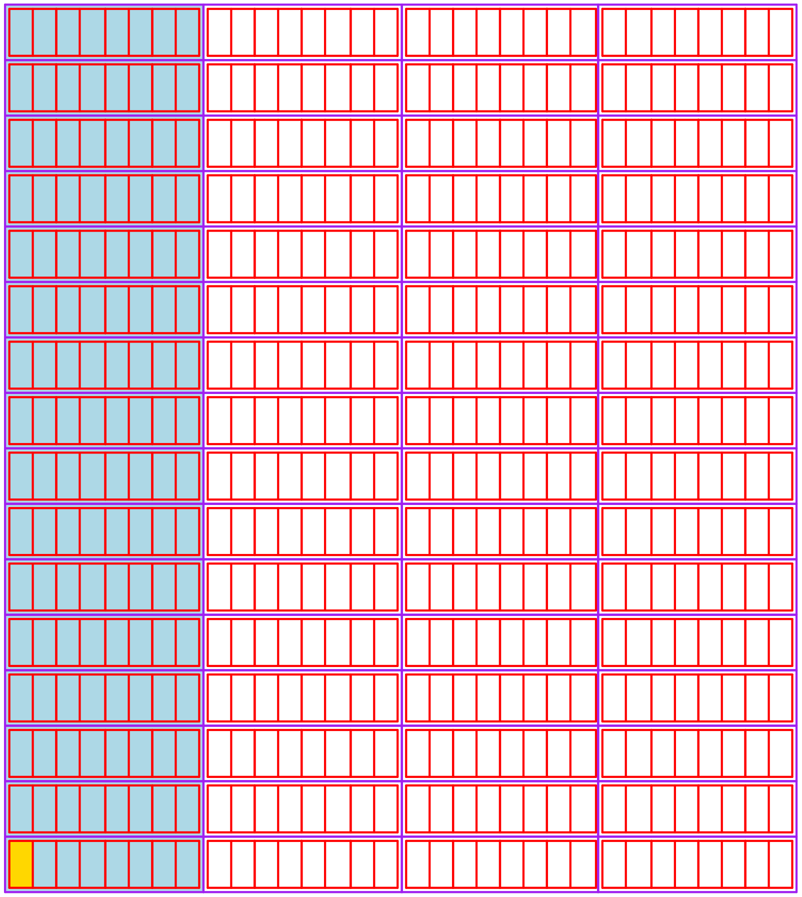
```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

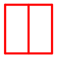
...

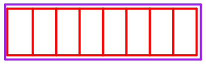
`i` = 7 `j` = 0 miss

`i` = 8 `j` = 0 miss

When Data Fits in Cache



 = **`grid`** element

 = fits cache line

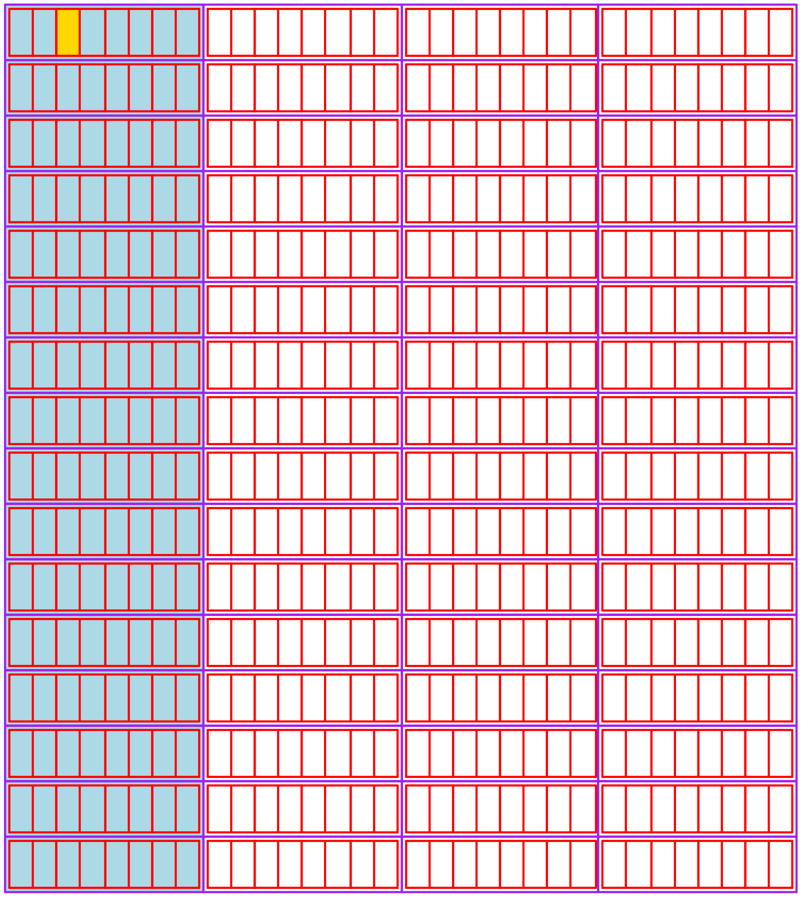
B = 32
S = **64**
E = 1

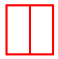
```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

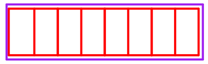
...

`i` = 15 **`j`** = 0 miss

When Data Fits in Cache



 = **`grid`** element

 = fits cache line

B = 32
S = **64**
E = 1

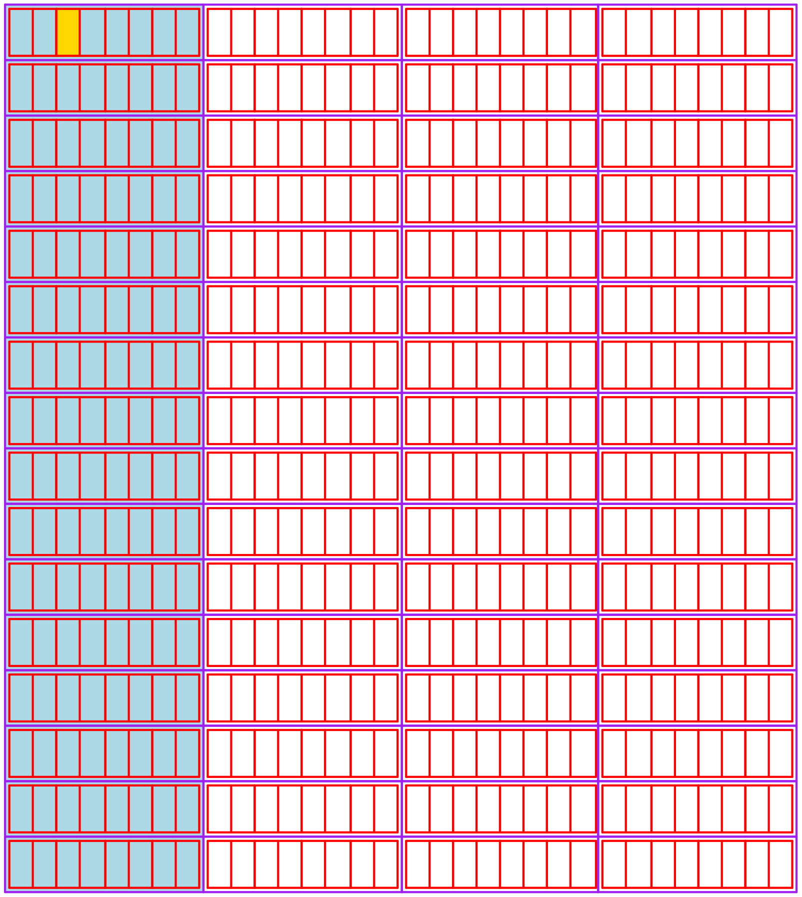
```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

...

`i` = 15 `j` = 0 miss

`i` = 0 `j` = 1 hit

When Data Fits in Cache



= **`grid`** element

= fits cache line

B = 32
S = **64**
E = 1

```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

Data fits in cache ⇒ order doesn't matter

When Data Fits in Cache

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31

= **`grid`** element

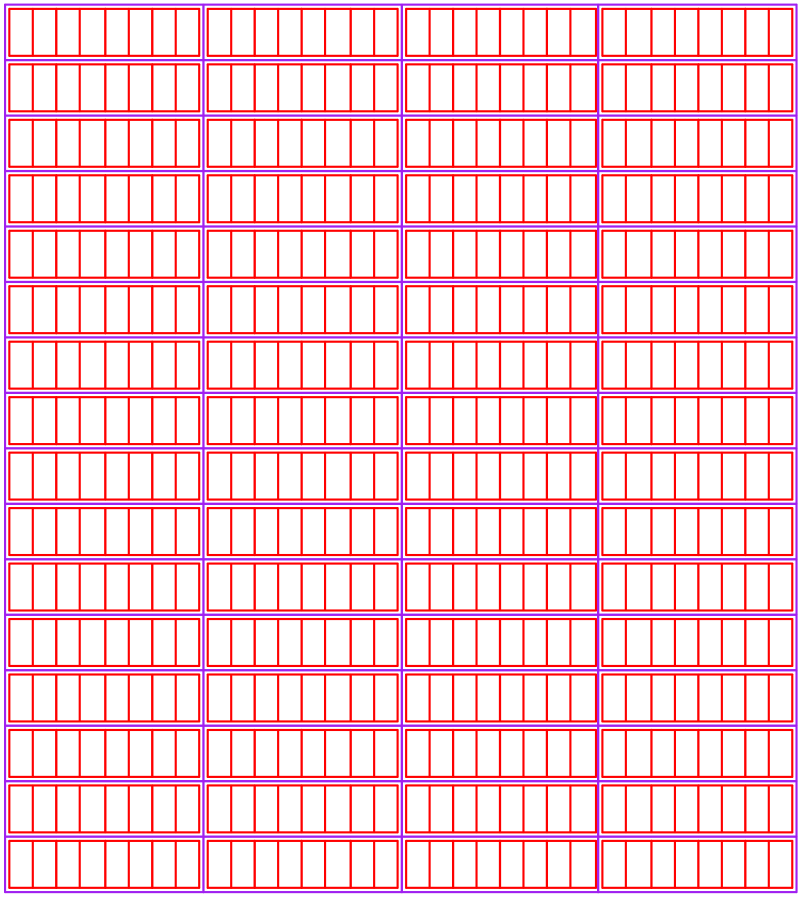
= fits cache line

B = 32
S = 32
E = **2**

```
for (j = 0; j < 16; j++)
  for (i = 0; i < 16; i++)
    total_x += grid[i][j].x;
```

Higher associativity increases the cache size, too

Multiple Loops



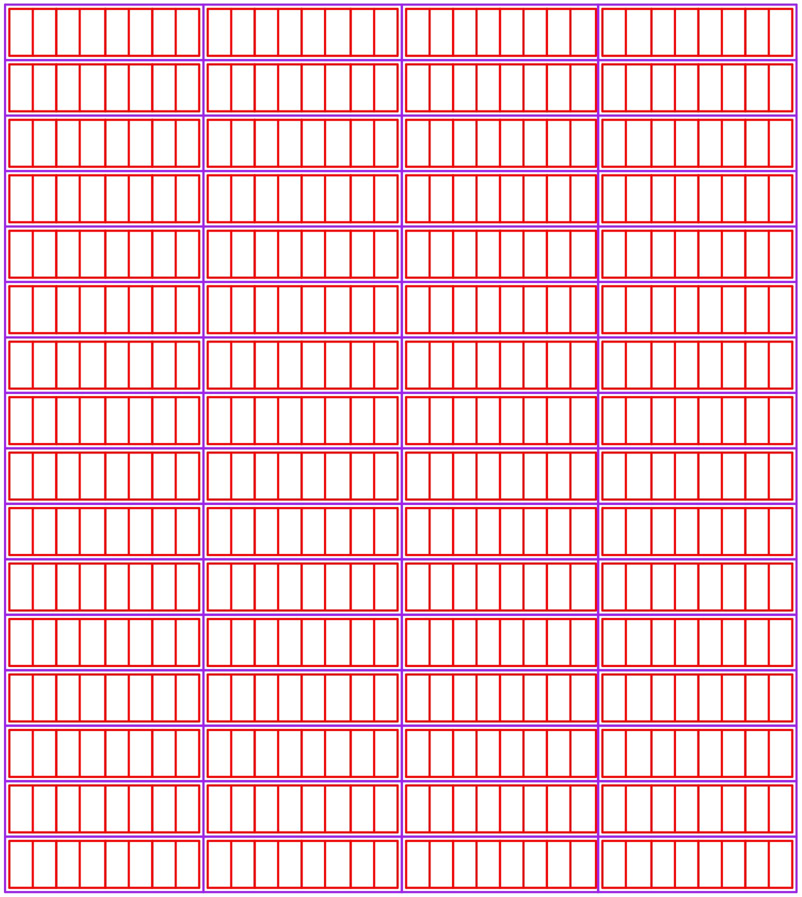
= **`grid`** element

= fits cache line

B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;

for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_y += grid[i][j].y;
```

Multiple Loops



= `grid` element

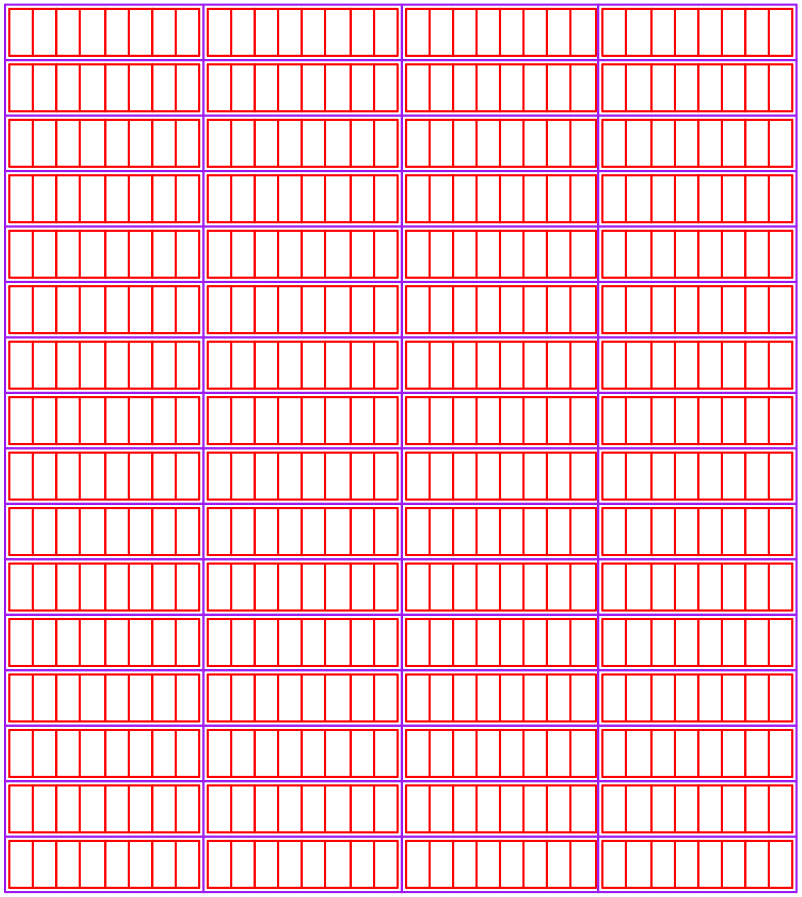
= fits cache line

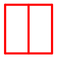
B = 32
S = 32
E = 1

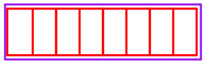
```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;

for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++)
    total_y += grid[i][j].y;
```

Miss rate: 25%

Multiple Loops



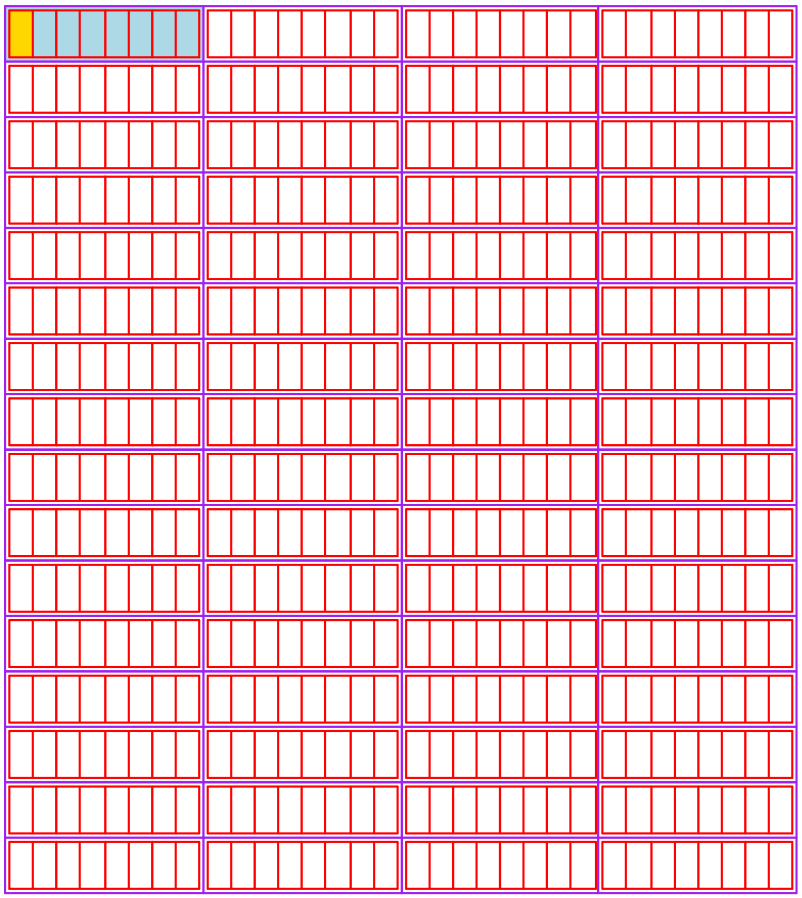
 = **`grid`** element

 = fits cache line

B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
```

Multiple Loops



= **`grid`** element

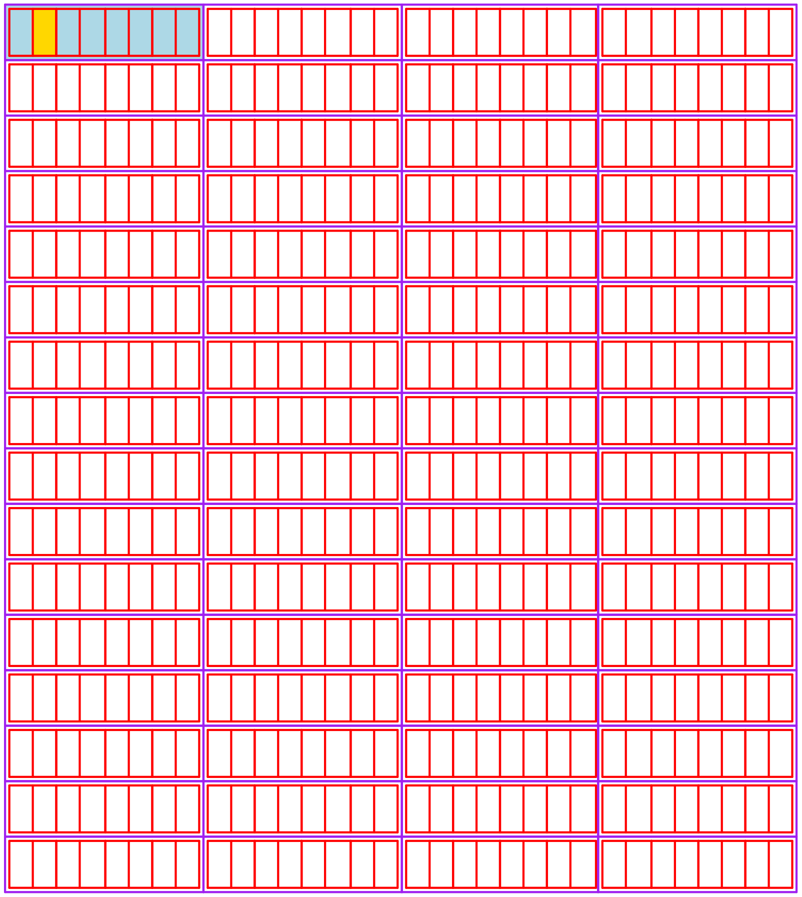
= fits cache line

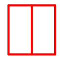
B = 32
S = 32
E = 1

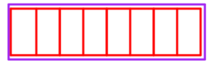
```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
```

`i` = 0 **`j`** = 0 miss

Multiple Loops




= `grid` element


= fits cache line

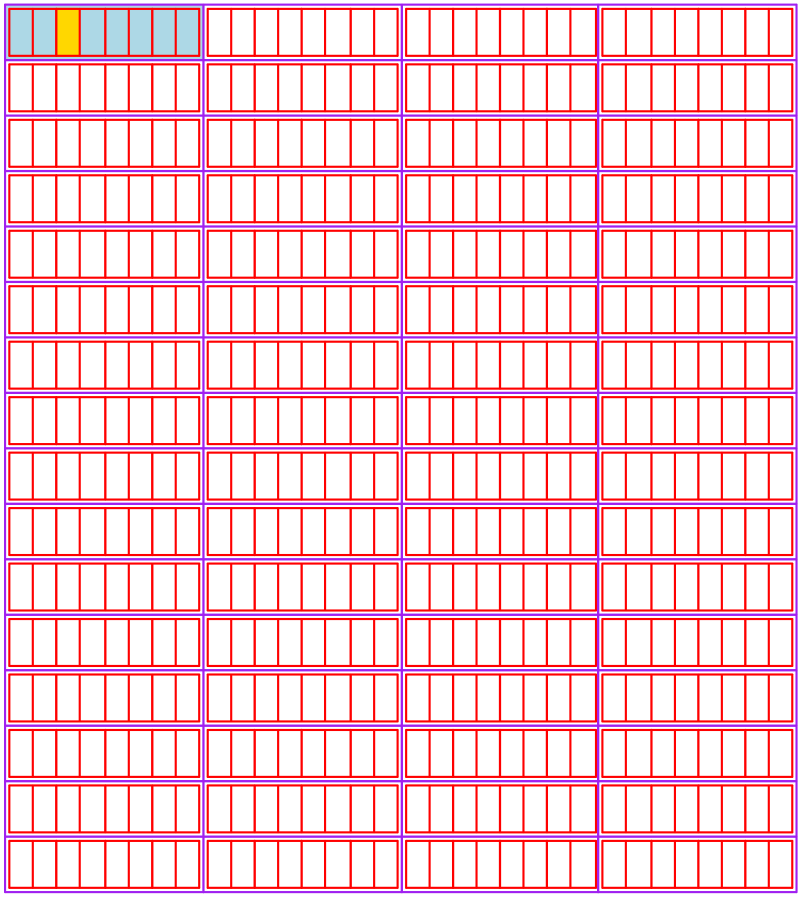
B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
```

`i` = 0 `j` = 0 miss

`i` = 0 `j` = 0 hit

Multiple Loops



= **grid** element

= fits cache line

B = 32
S = 32
E = 1

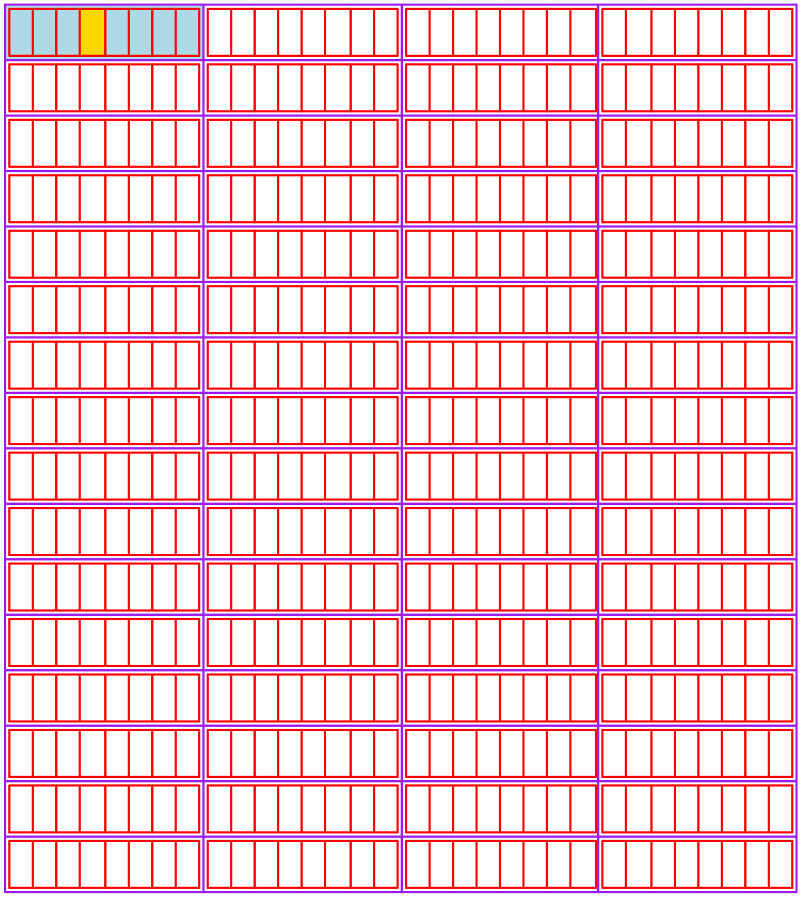
```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
```

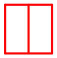
i = 0 **j** = 0 miss

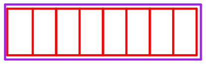
i = 0 **j** = 0 hit

i = 0 **j** = 1 hit

Multiple Loops



 = **`grid`** element

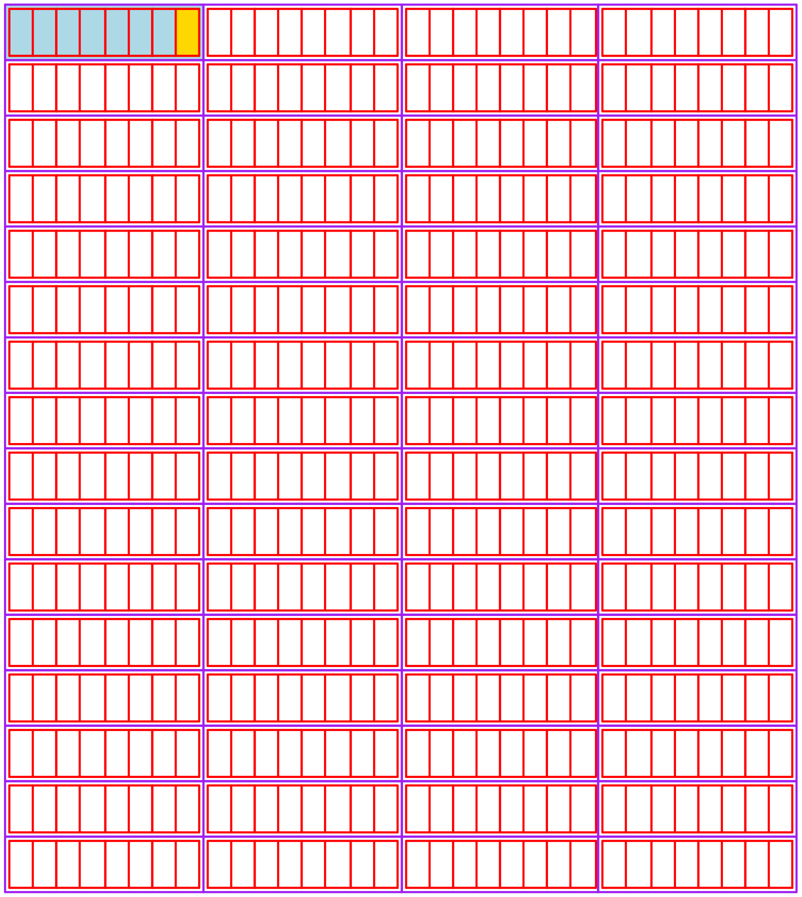
 = fits cache line

B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
```

`i` = 0 `j` = 0 miss
`i` = 0 `j` = 0 hit
`i` = 0 `j` = 1 hit
`i` = 0 `j` = 1 hit

Multiple Loops



[legend box] = **`grid`** element

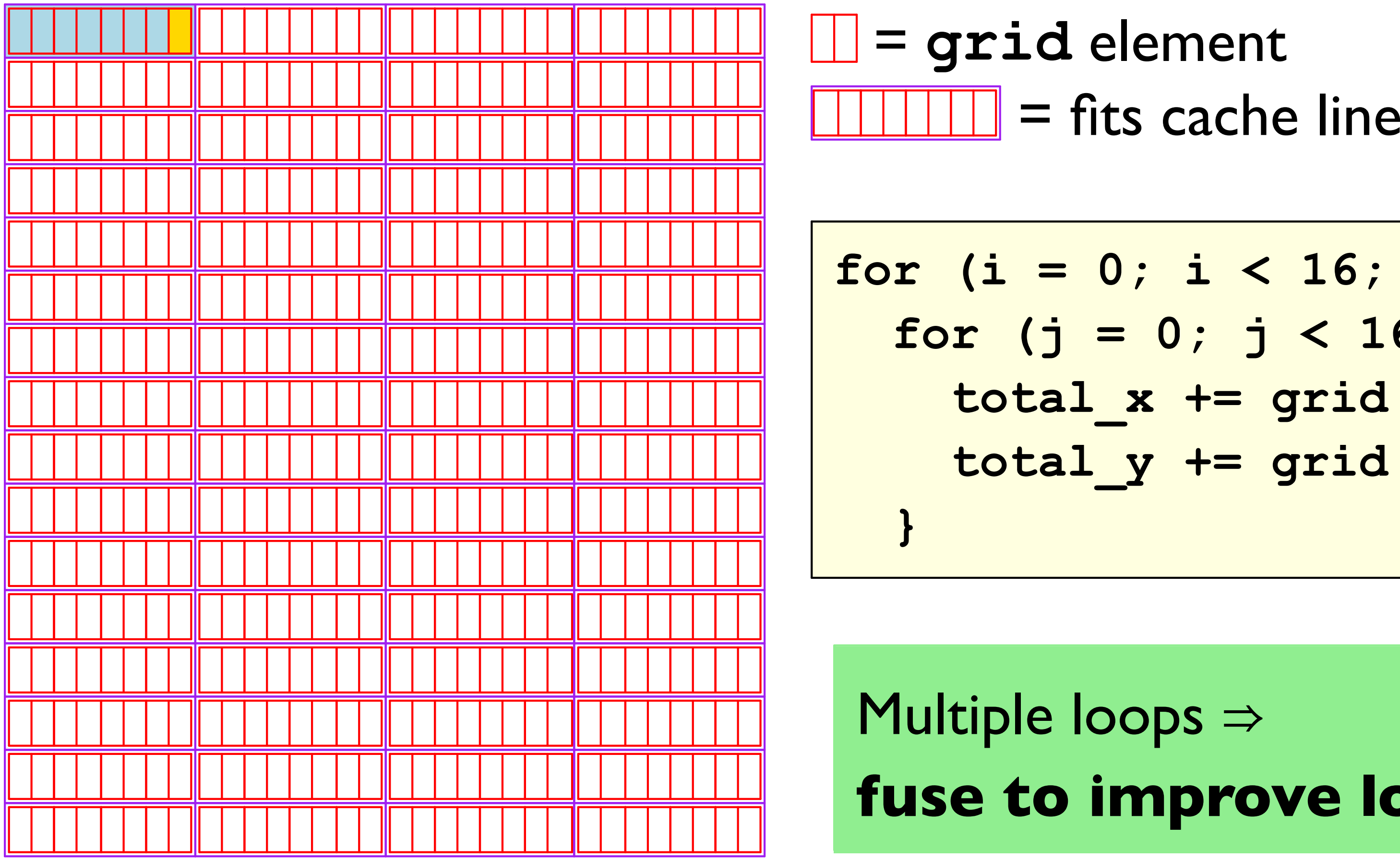
[legend box] = fits cache line

B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
```

Miss rate: 12.5%

Multiple Loops



B = 32
S = 32
E = 1

```
for (i = 0; i < 16; i++)
  for (j = 0; j < 16; j++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
```

Multiple loops ⇒
fuse to improve locality

Array and Cache Summary

Traverse arrays ⇒ **column as inner loop**

Data fits in cache ⇒ order doesn't matter

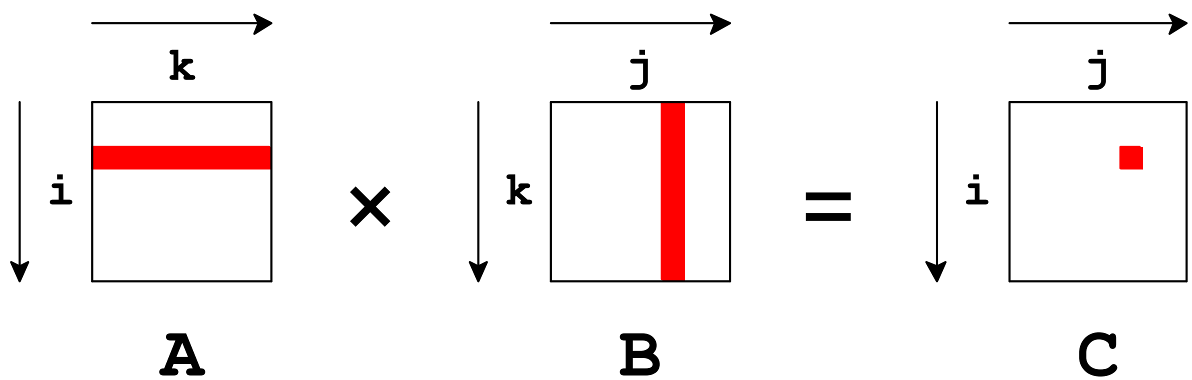
Multiple loops ⇒ **fuse to improve locality**

Single use of each array element:

$$\text{Optimal miss rate} = \frac{1}{\text{elements per block}}$$

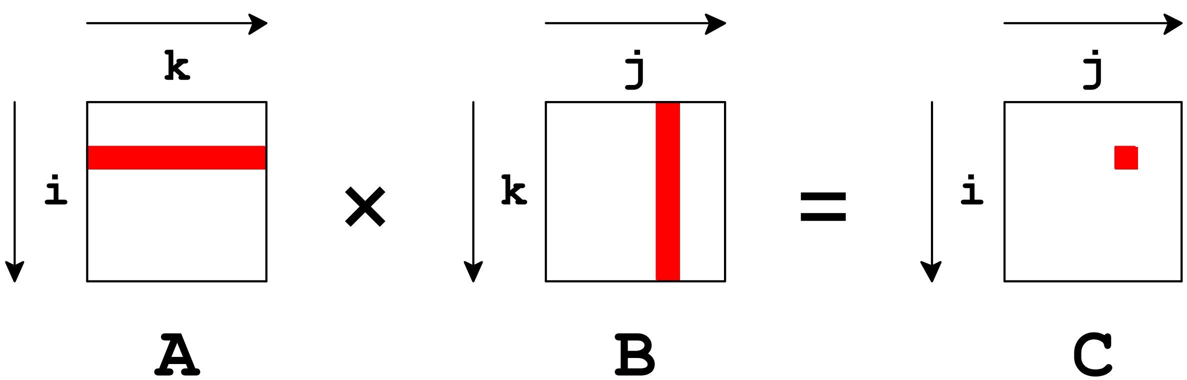
Matrix Multiplication

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i][j] += A[i][k] * B[k][j];
```



Matrix Multiplication

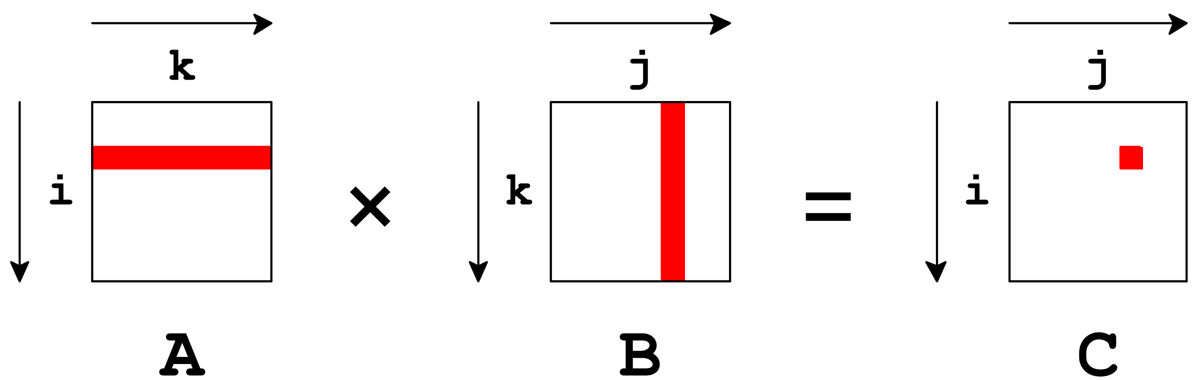
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i][j] += A[i][k] * B[k][j];
```



Time: $O(n^3)$

Matrix Multiplication

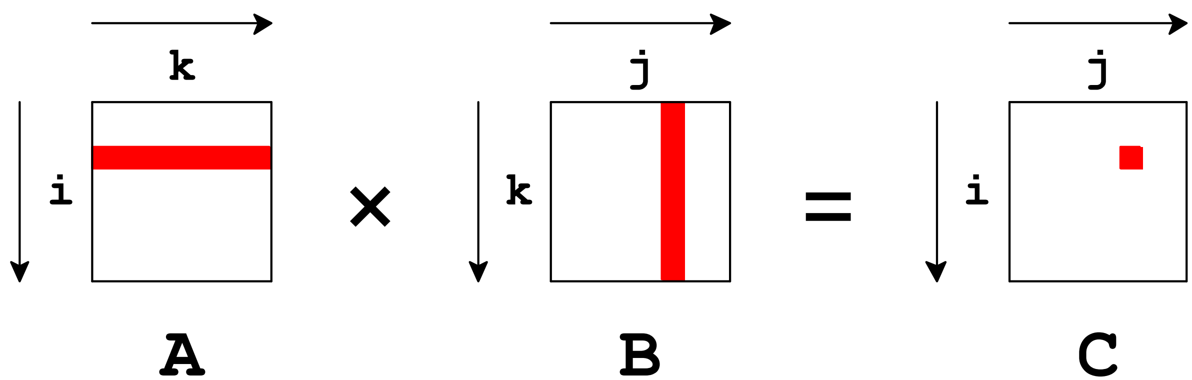
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i][j] += A[i][k] * B[k][j];
```



Space: O(n^2)

Matrix Multiplication

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i][j] += A[i][k] * B[k][j];
```

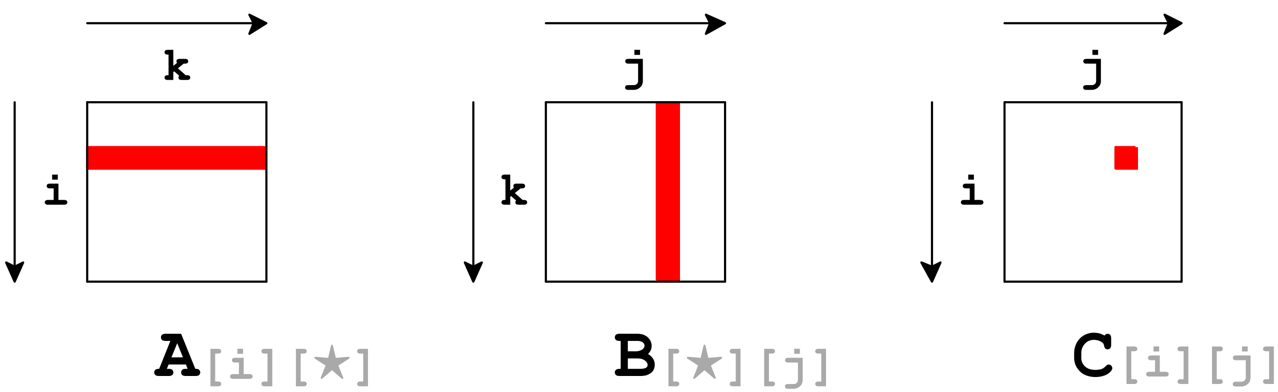


Cache assumptions

- n^2 >> cache size
- 4 elements per cache block

Matrix Multiplication

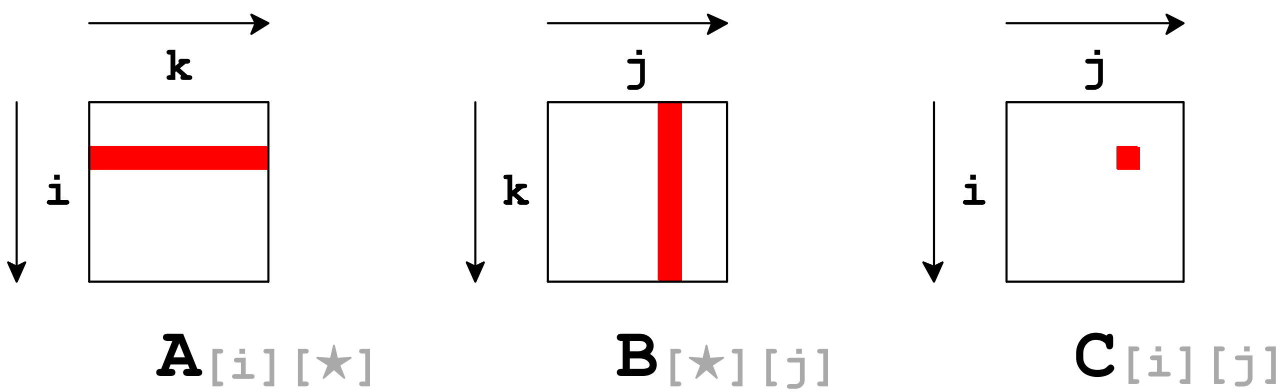
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i][j] += A[i][k] * B[k][j];
```



Cache performance: consider innermost loop

Matrix Multiplication

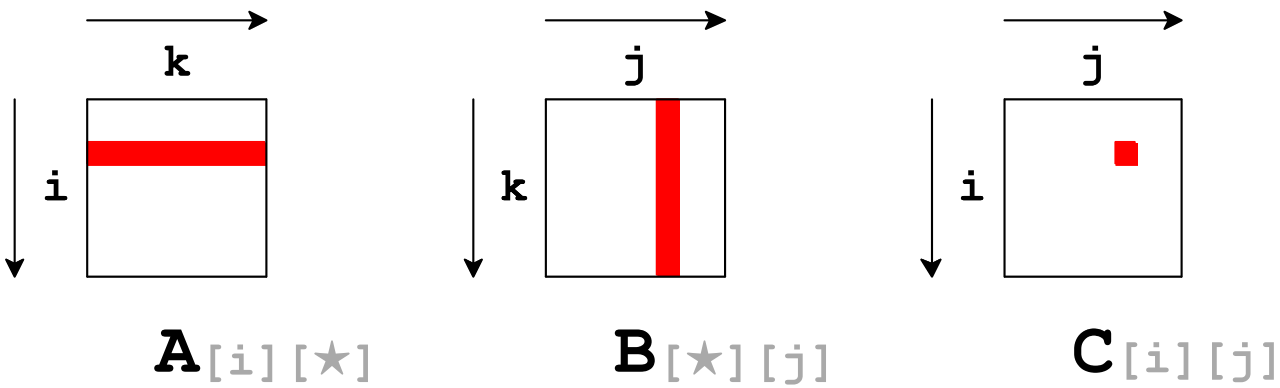
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i][j] += A[i][k] * B[k][j];
```



Miss rate: 0.25 + 1.0 + 0.0 = 1.25

Matrix Multiplication: `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;
  }
```

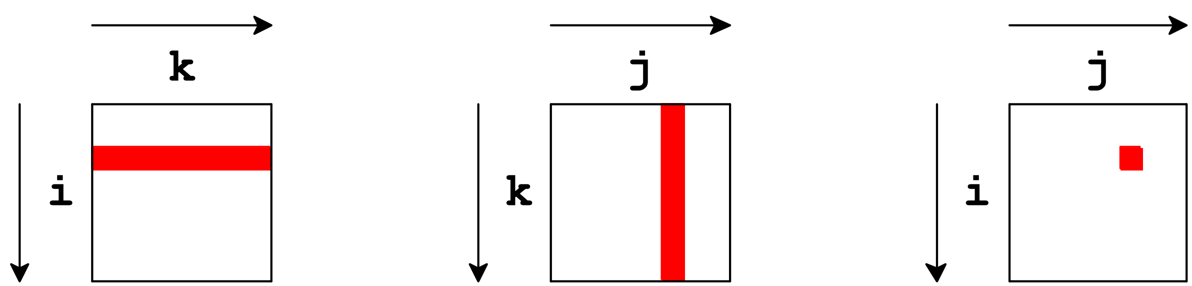


Miss rate: **0.25** + **1.0** + **0.0** = **1.25**

Matrix Multiplication: `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;
  }
```

Swapping has no effect

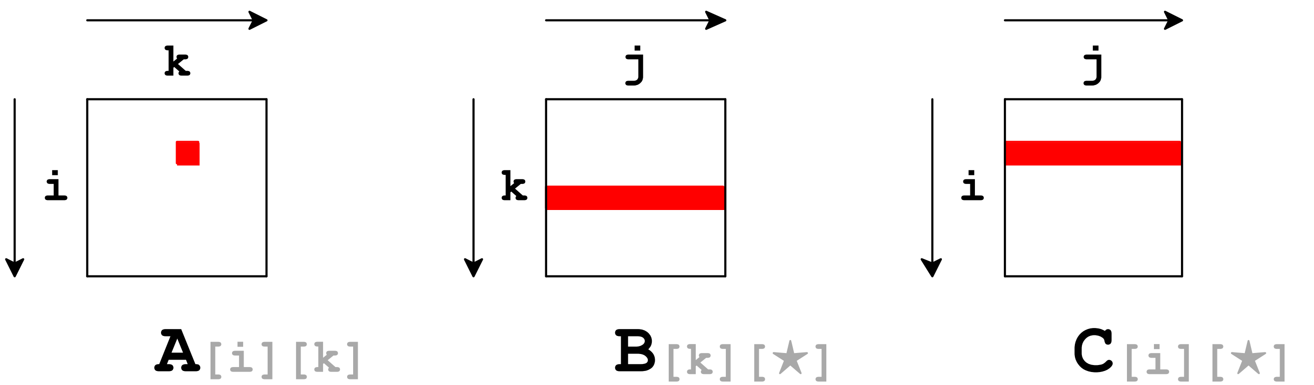


A[i][★] **B**[★][j] **C**[i][j]

Miss rate: 0.25 + 1.0 + 0.0 = 1.25

Matrix Multiplication: kij

```
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++) {
    a = A[i][k];
    for (j = 0; j < n; j++)
      C[i][j] += a * B[k][j];
  }
```

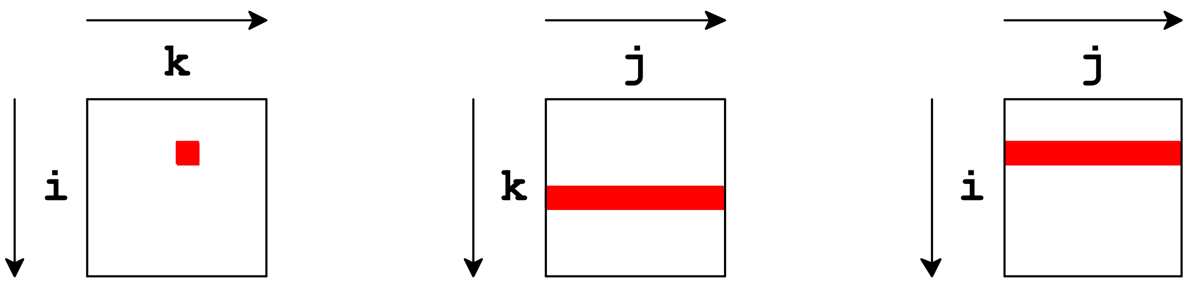


Miss rate: 0.0 + 0.25 + 0.25 = 0.5

Matrix Multiplication: `ikj`

```
for (i = 0; i < n; i++)
  for (k = 0; k < n; k++) {
    a = A[i][k];
    for (j = 0; j < n; j++)
      C[i][j] += a * B[k][j];
  }
```

Swapping has no effect

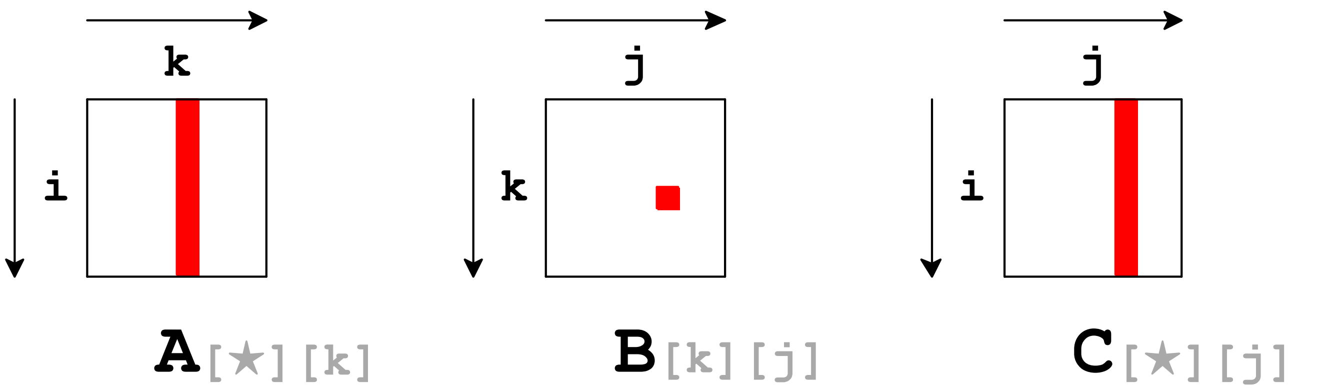


A[i][k] **B**[k][★] **C**[i][★]

Miss rate: 0.0 + 0.25 + 0.25 = 0.5

Matrix Multiplication: `jki`

```
for (j = 0; j < n; j++)
  for (k = 0; k < n; k++) {
    b = B[k][j];
    for (i = 0; i < n; i++)
      C[i][j] += A[i][k] * b;
  }
```

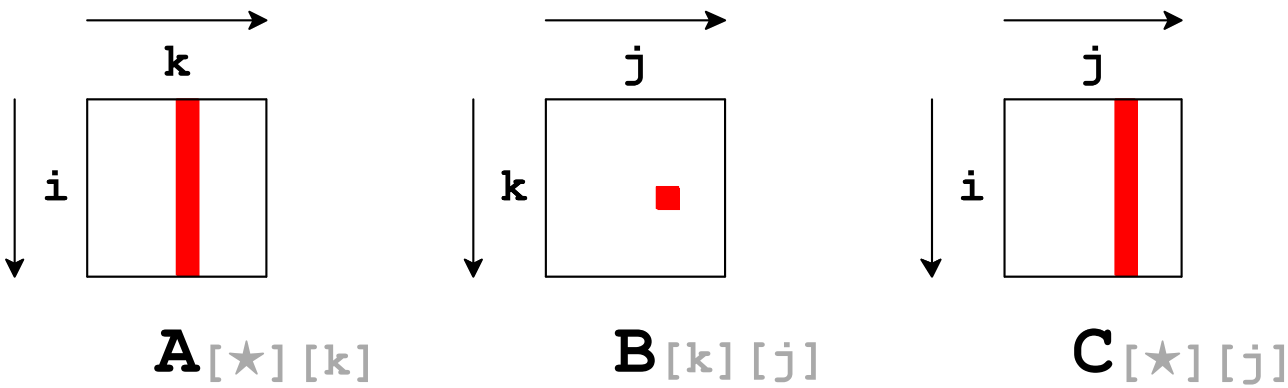


Miss rate: 1.0 + 0.0 + 1.0 = 2.0

Matrix Multiplication: **kji**

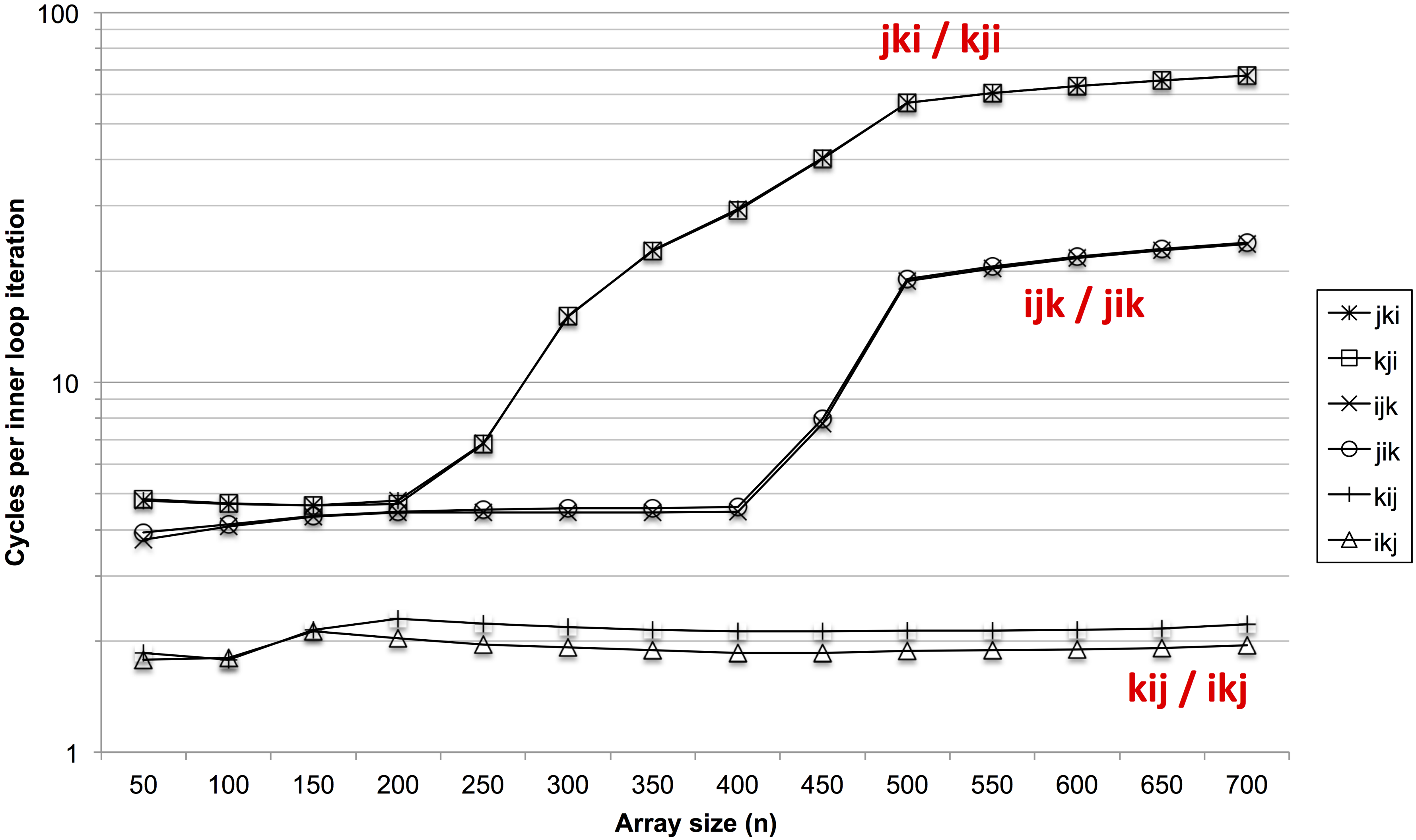
```
for (k = 0; k < n; k++)
  for (j = 0; j < n; j++) {
    b = B[k][j];
    for (i = 0; i < n; i++)
      C[i][j] += A[i][k] * b;
  }
```

Swapping has no effect



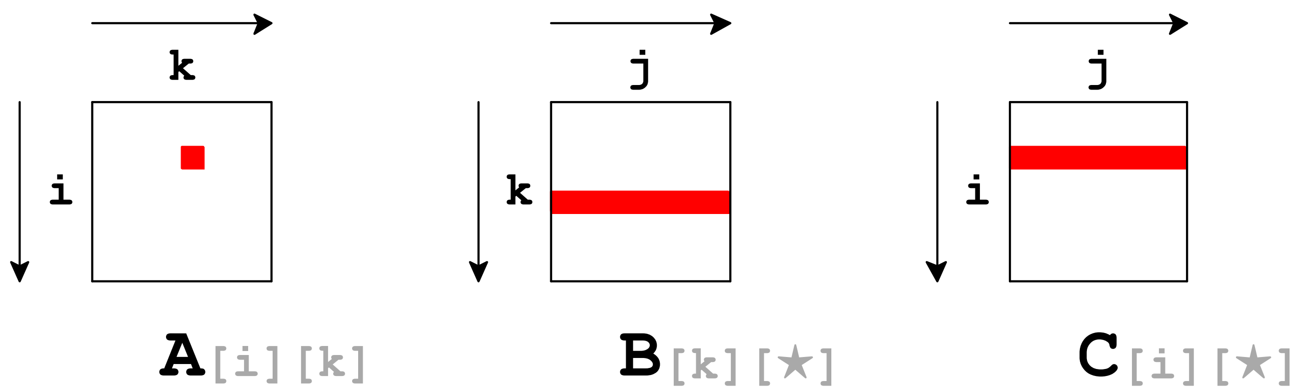
Miss rate: 1.0 + 0.0 + 1.0 = 2.0

Matrix Multiply Performance



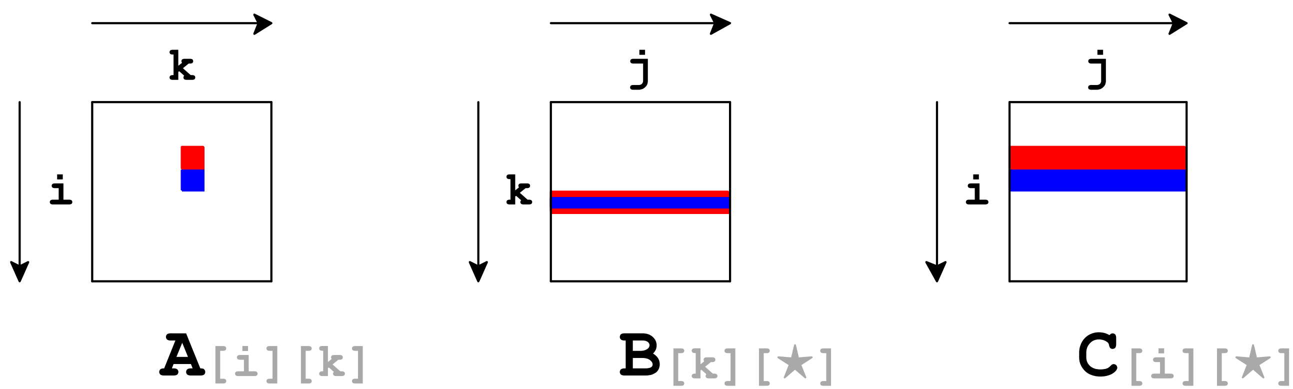
Room for Improvement?

```
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++) {
    a = A[i][k];
    for (j = 0; j < n; j++)
      C[i][j] += a * B[k][j];
  }
```



Room for Improvement?

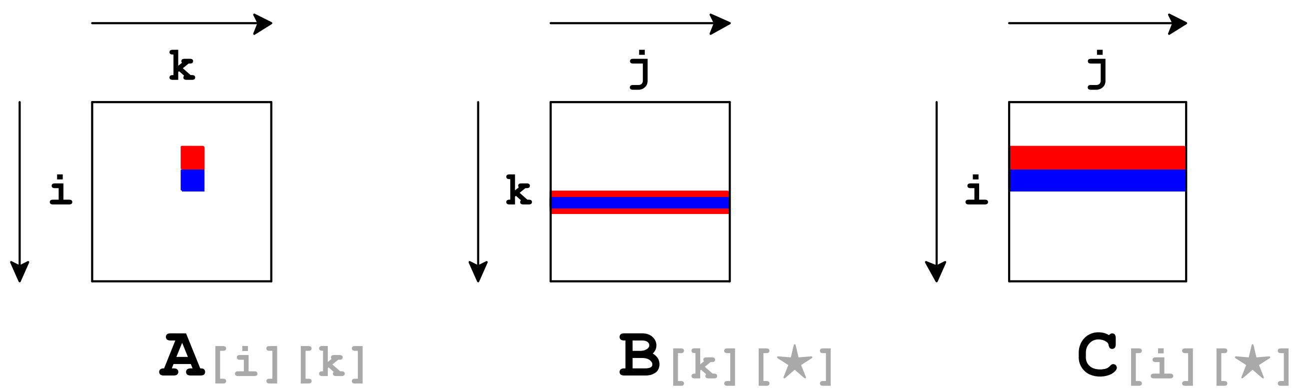
```
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++) {
    a = A[i][k];
    for (j = 0; j < n; j++)
      C[i][j] += a * B[k][j];
  }
```



If rows don't fit in cache, **B** starts over for ■

Room for Improvement?

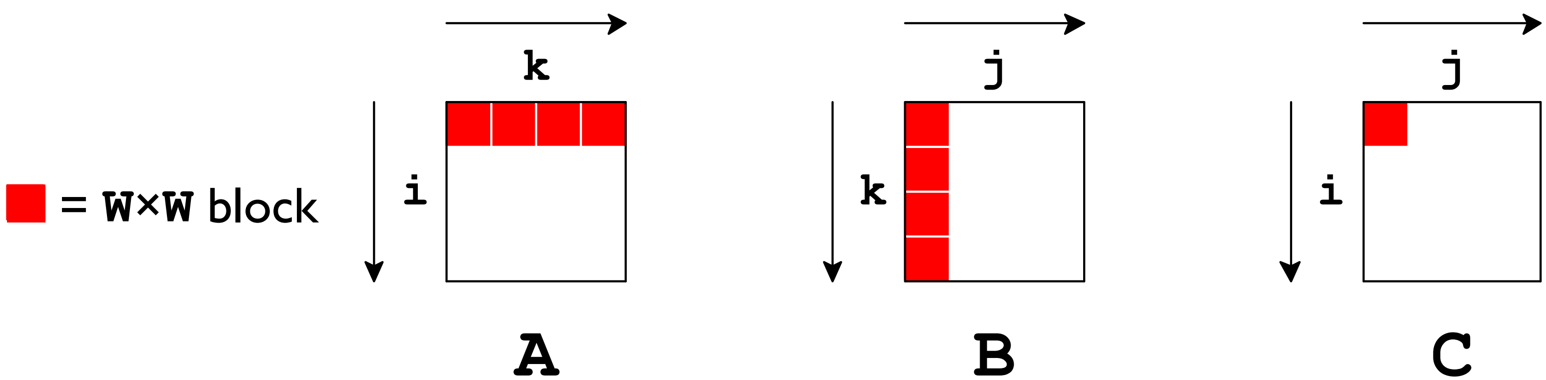
```
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++) {
    a = A[i][k];
    for (j = 0; j < n; j++)
      C[i][j] += a * B[k][j];
  }
```



Maybe try wide enough segments of ■ and ■ in **C** to stay in same region of **B**

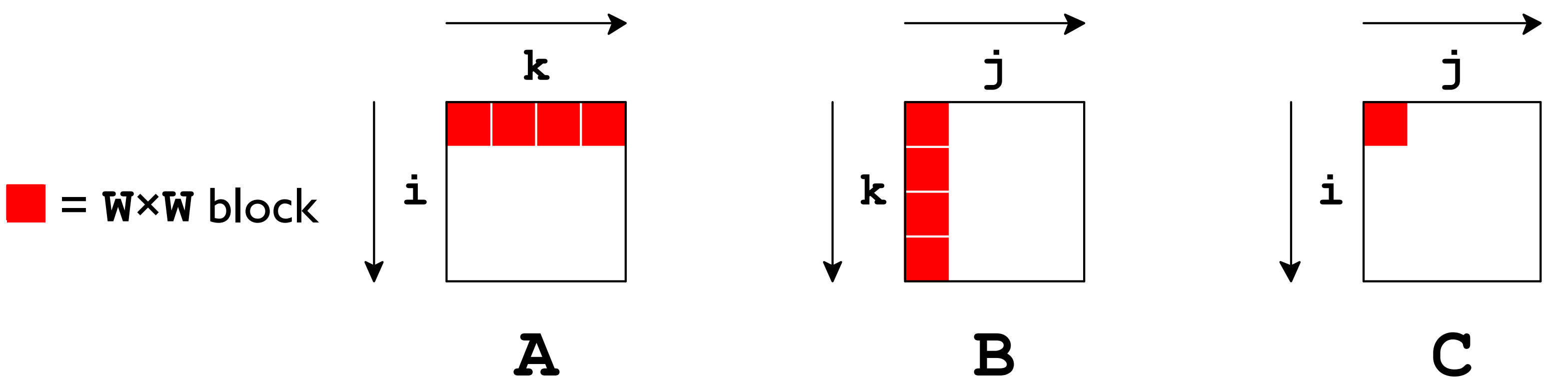
Blocking Matrix Multiplication

```
for (i = 0; i < n; i += W)
  for (j = 0; j < n; j += W)
    for (k = 0; k < n; k += W)
      for (ii = i; ii < i+W; ii++)
        for (jj = j; jj < j+W; jj++)
          for (kk = k; kk < k+W; kk++)
            C[ii][jj] += A[ii][kk] * B[kk][jj];
```



Blocking Matrix Multiplication

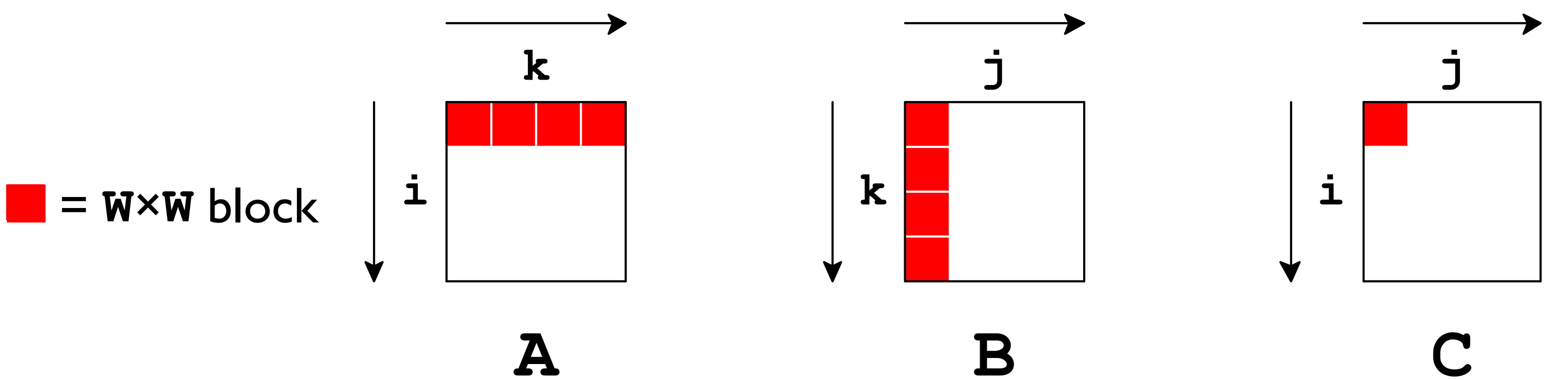
```
for (i = 0; i < n; i += W)
  for (j = 0; j < n; j += W)
    for (k = 0; k < n; k += W)
      for (ii = i; ii < i+W; ii++)
        for (jj = j; jj < j+W; jj++)
          for (kk = k; kk < k+W; kk++)
            C[ii][jj] += A[ii][kk] * B[kk][jj];
```



W^2 elements fit in cache ⇒ `0.25` miss rate

Blocking Matrix Multiplication

```
for (i = 0; i < n; i += W)
  for (j = 0; j < n; j += W)
    for (k = 0; k < n; k += W)
      for (ii = i; ii < i+W; ii++)
        for (jj = j; jj < j+W; jj++)
          for (kk = k; kk < k+W; kk++)
            C[ii][jj] += A[ii][kk] * B[kk][jj];
```

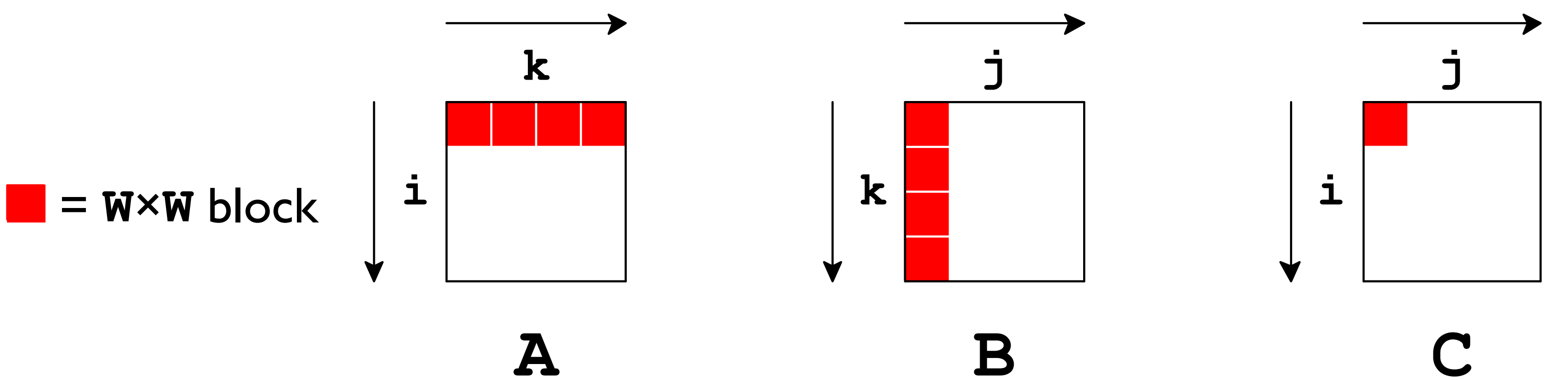


$2\frac{n}{W}W^2 + W^2$ total elements at `0.25` miss rate

repeated for $(\frac{n}{W})^2$ result blocks

Blocking Matrix Multiplication

```
for (i = 0; i < n; i += W)
  for (j = 0; j < n; j += W)
    for (k = 0; k < n; k += W)
      for (ii = i; ii < i+W; ii++)
        for (jj = j; jj < j+W; jj++)
          for (kk = k; kk < k+W; kk++)
            C[ii][jj] += A[ii][kk] * B[kk][jj];
```



$$0.25 \times 2\frac{n}{W}W^2 \times \left(\frac{n}{W}\right)^2 = \frac{n^3}{2W}$$

$$\Rightarrow \frac{1}{2W} \text{ miss rate}$$