15-213/15-503 Recitation
Caches

Josh Kim

Friday, June 6th

Reminders

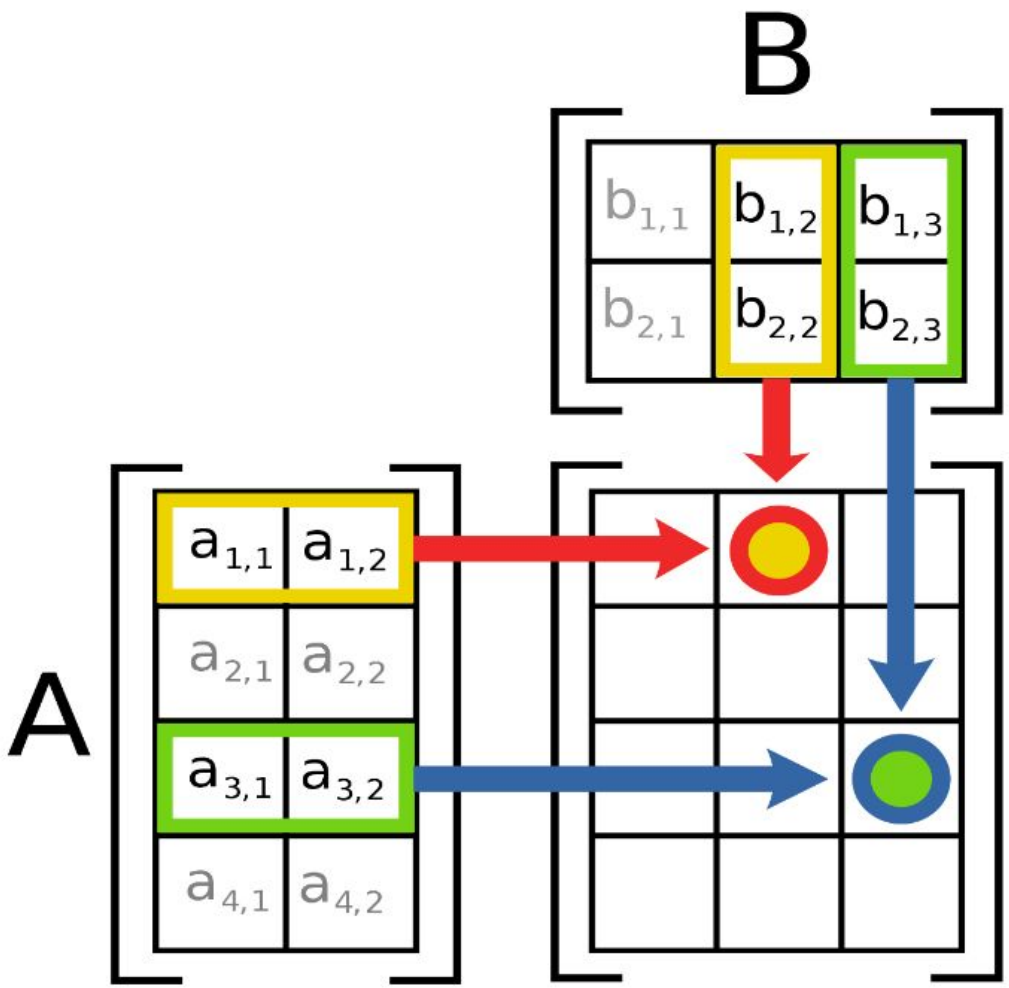
- **`attacklab`** is due on ***Tuesday (June 10th)***
- **`cachelab`** is releases ***Tuesday***

Agenda

- Wrapping up previous lecture
- Intro to `cachelab`
- Review: Cache Concepts
- Writing Cache-Friendly Code
- Blocking
- Activity: More Blocking!
- Connecting back to lecture

From Lecture: Rearranging loops to improve spatial locality

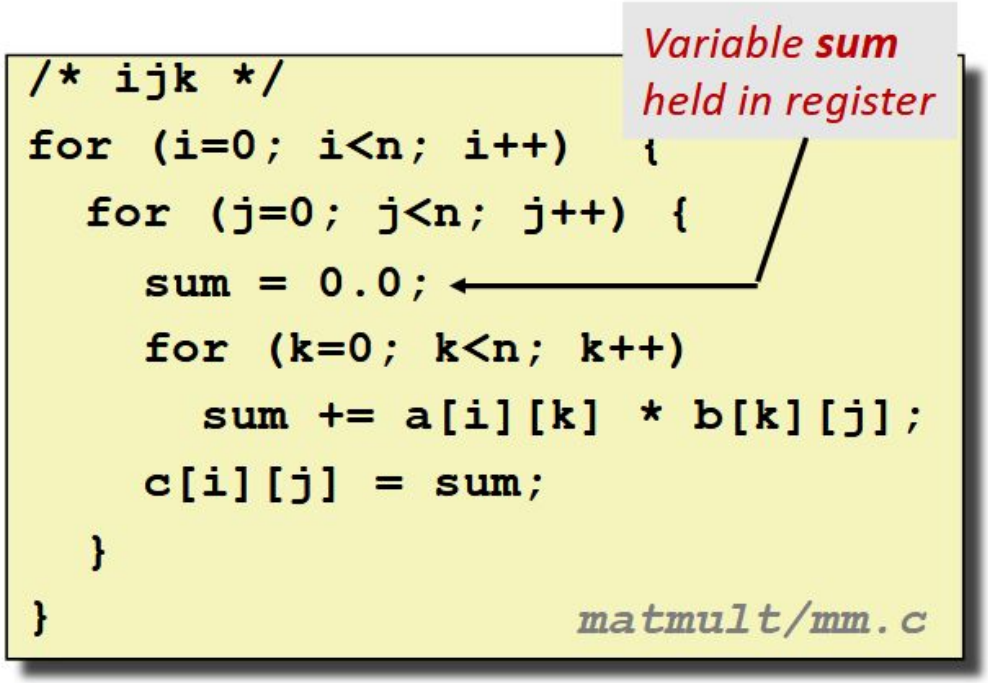
Remember matrix multiplication



Out[i, j] =
dot product(A[i, ..], B[..,j])
= sum (a[i, 0] * b[0, j],
a[i, 1] * b[1, j],
...
a[i, n] * b[n, j])

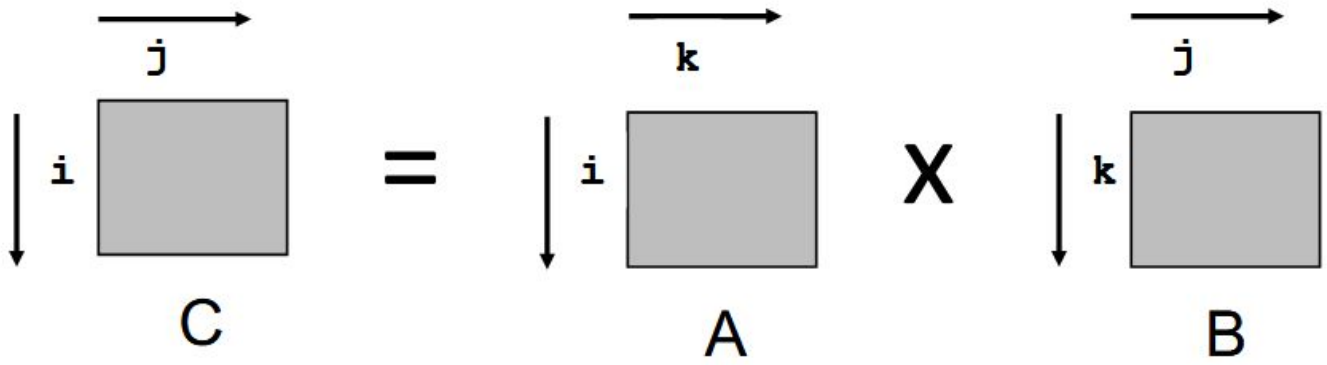
Matrix Multiplication Example

- **Description:**
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register



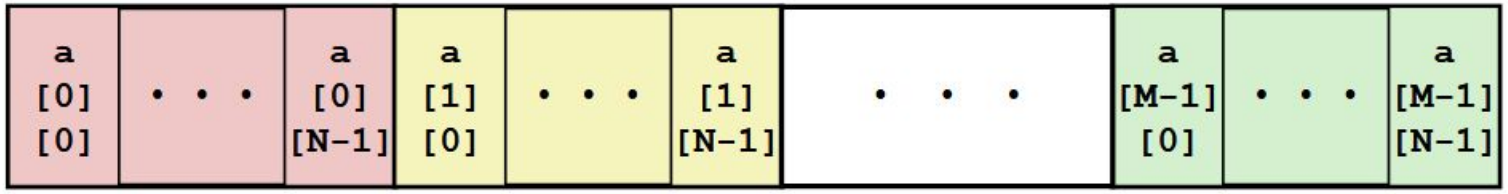
Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - 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

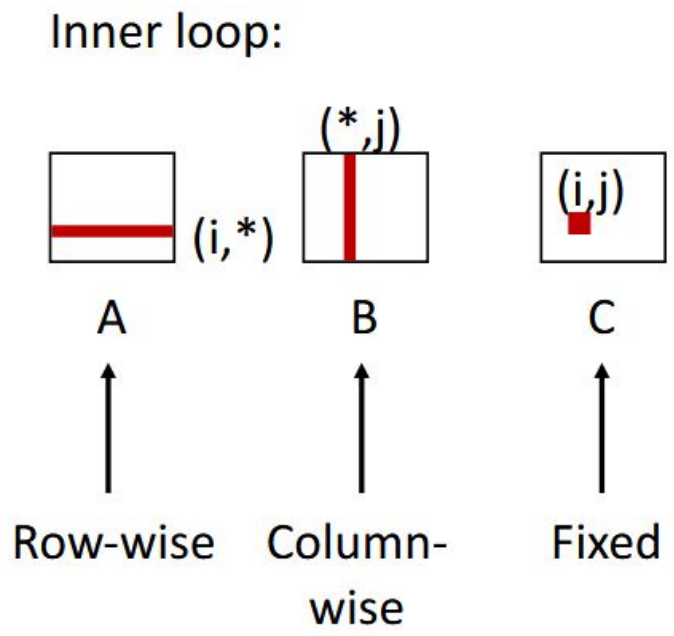


- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i]`
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- Stepping through rows in one column:
  - `for (i = 0; i < M; i++)`
    `sum += a[i][0];`
  - accesses distant elements: 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;
  }
}
```

matmult/mm.c



Miss rate for inner loop iterations:

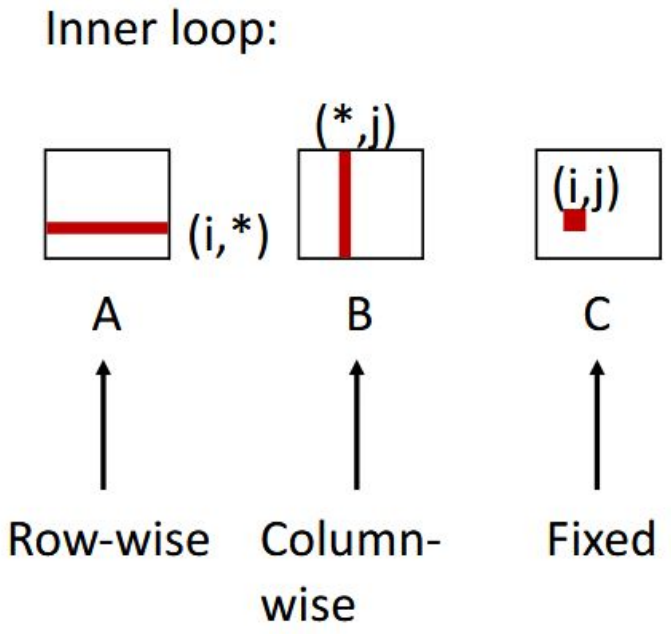
A	B	C

Block size = 32B (four doubles)

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;
  }
}
```

matmult/mm.c



Miss rate for inner loop iterations:

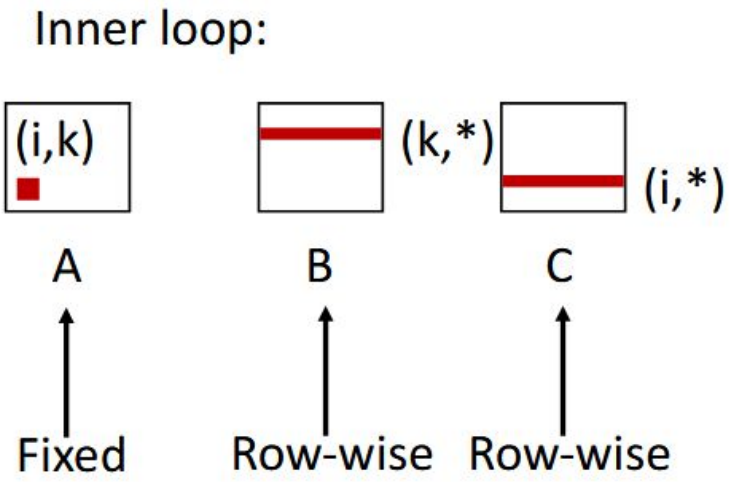
A	B	C
0.25	1.0	0.0

Block size = 32B (four doubles)

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];
  }
}
```

matmult/mm.c



Miss rate for inner loop iterations:

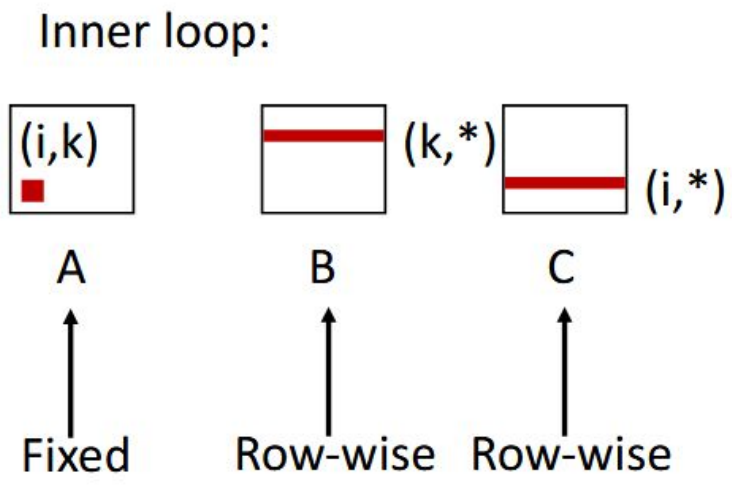
A	B	C

Block size = 32B (four doubles)

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];
  }
}
```

matmult/mm.c

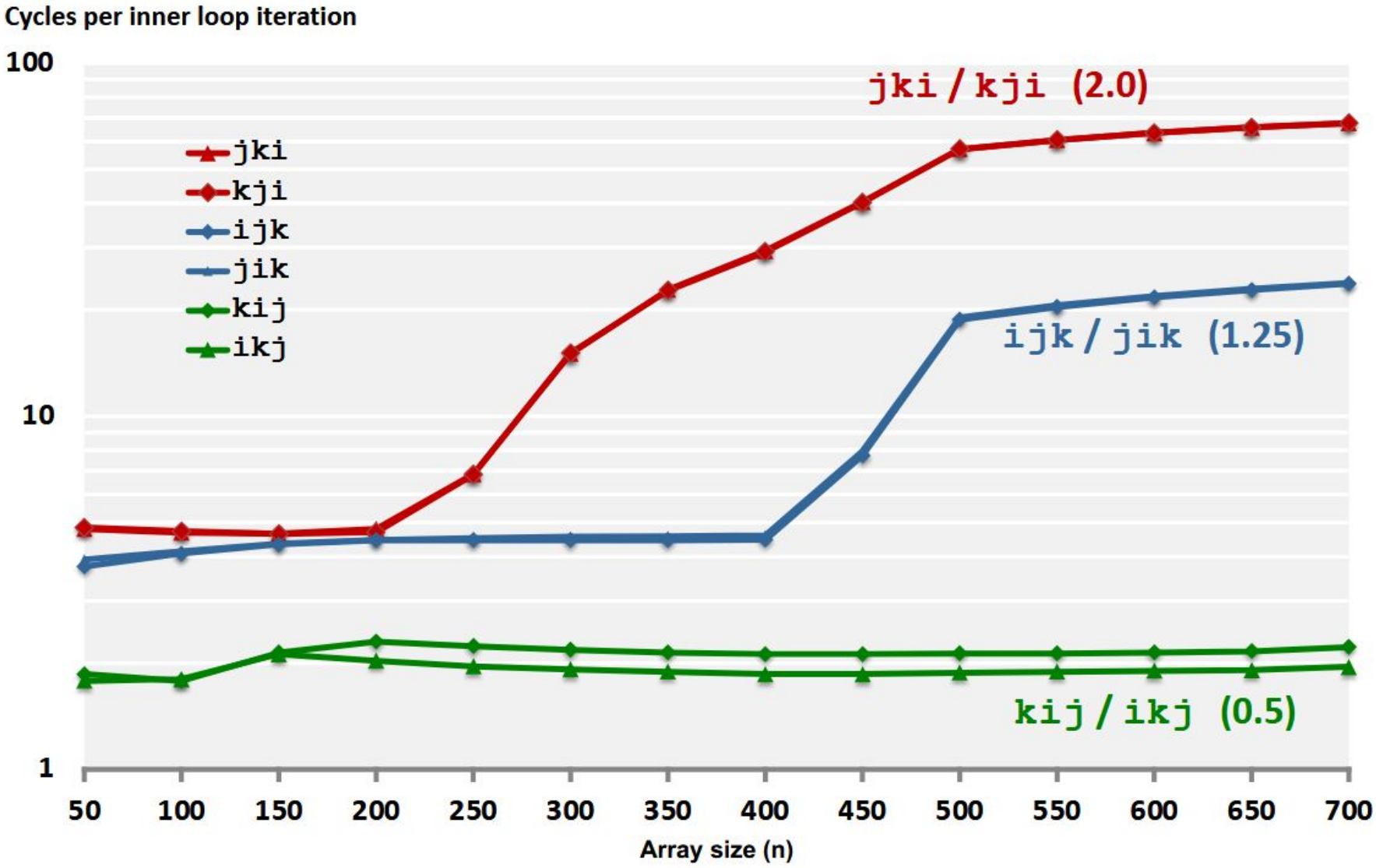


Miss rate for inner loop iterations:

A	B	C
0.0	0.25	0.25

Block size = 32B (four doubles)

Core i7 Matrix Multiply Performance



Introduction to `cachelab`

`cachelab`: Overview

- First project-based assignment:
  - You'll write a *cache simulator* in C from scratch!
- Take in parameters defining the cache structure (**`s`**, **`E`**, **`b`**).
- Read a "trace file" of memory accesses and simulate them.
- After simulating those accesses, return the number of hits, misses, evictions, etc.

Using Git

- This is the first lab where we will use Git!
  - Note you will download the assignment through Github
- Try to keep good version control practice!
  - Commit after every “good fix”!
  - Good version control can help you recover from “bad mistakes”!

Review: Cache Concepts

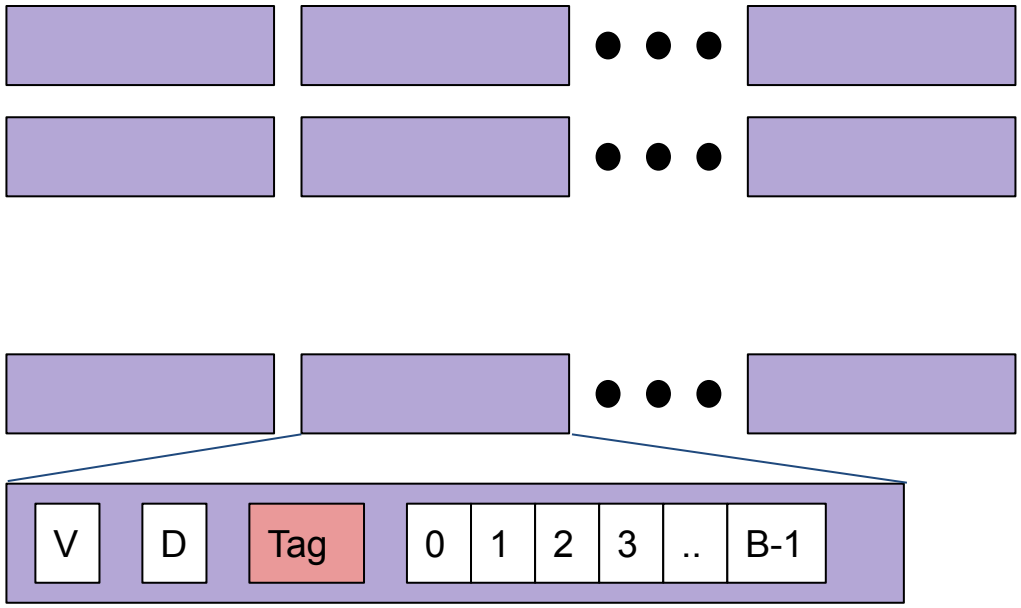
Cache Concepts: Configurations

- Your cache simulators need to support *parameters* (**`s`**, **`E`**, **`b`**) that allow the user to configure the layout of the cache.
- But what do these parameters mean?
- Let's review how a cache is organized!

Cache Organization

Note: don't need to store data for **`cachelab`**.

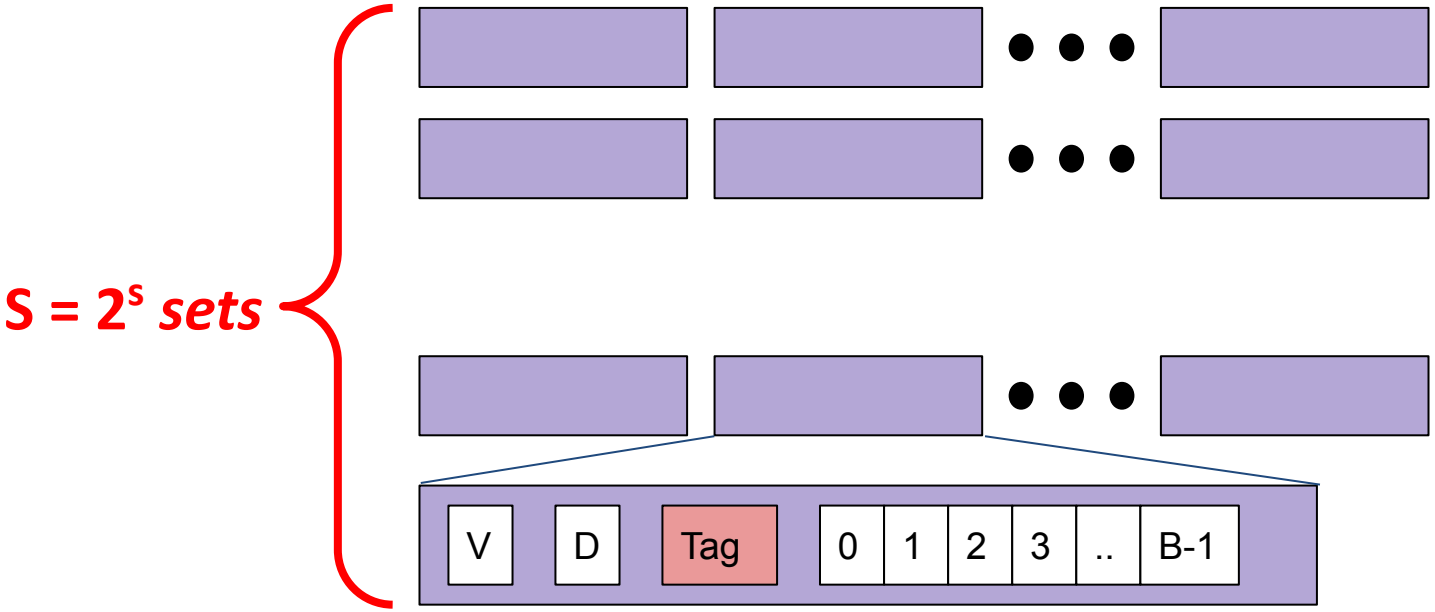
- A cache is composed of *sets*
- Each set is composed of some number of *lines*
- Each line stores the cached data itself, as well as information used by the cache.



Cache Organization

- A cache is composed of *sets*

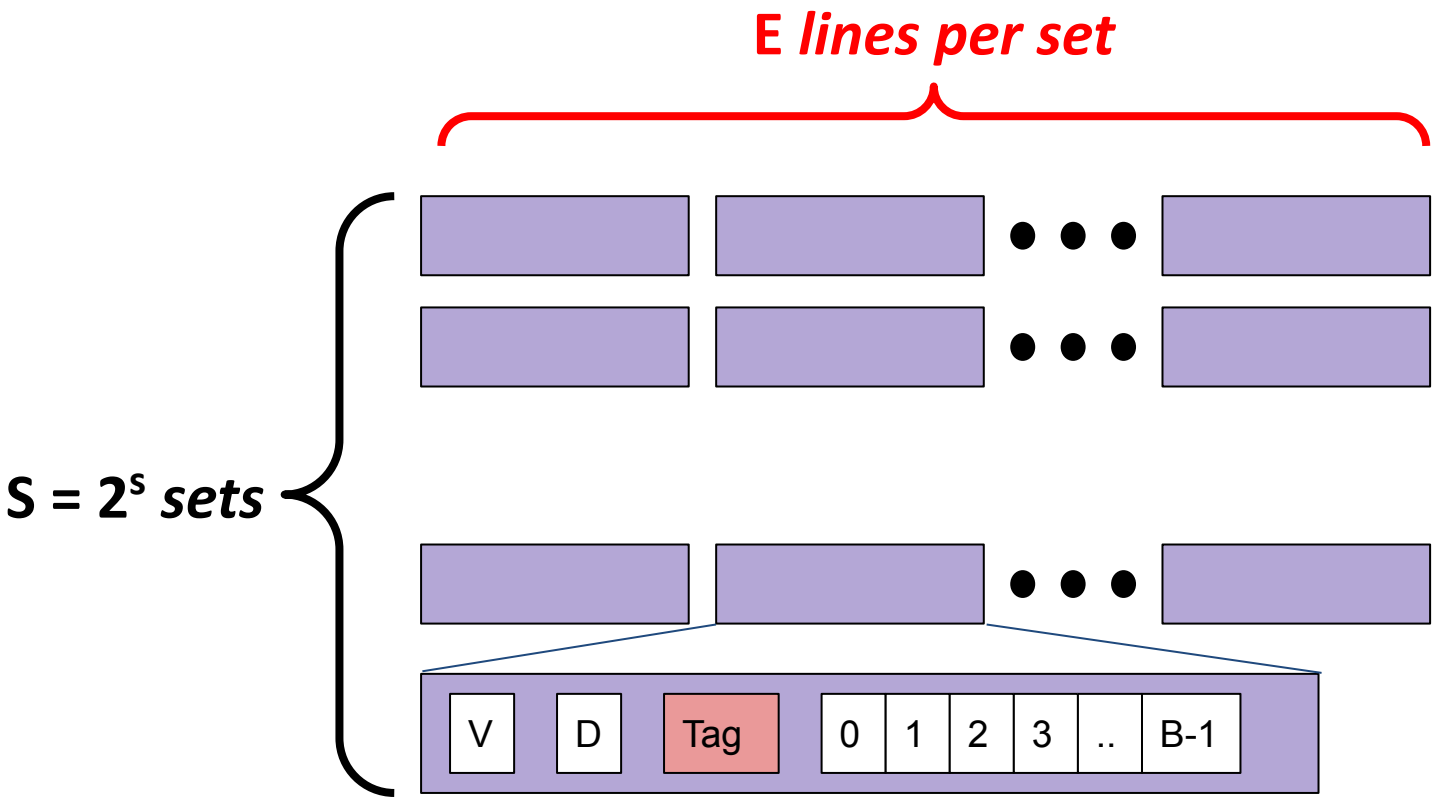
s – Number of set *bits*
$S = 2^s$ – Number of *sets*



Cache Organization

- Each set is composed of *lines*

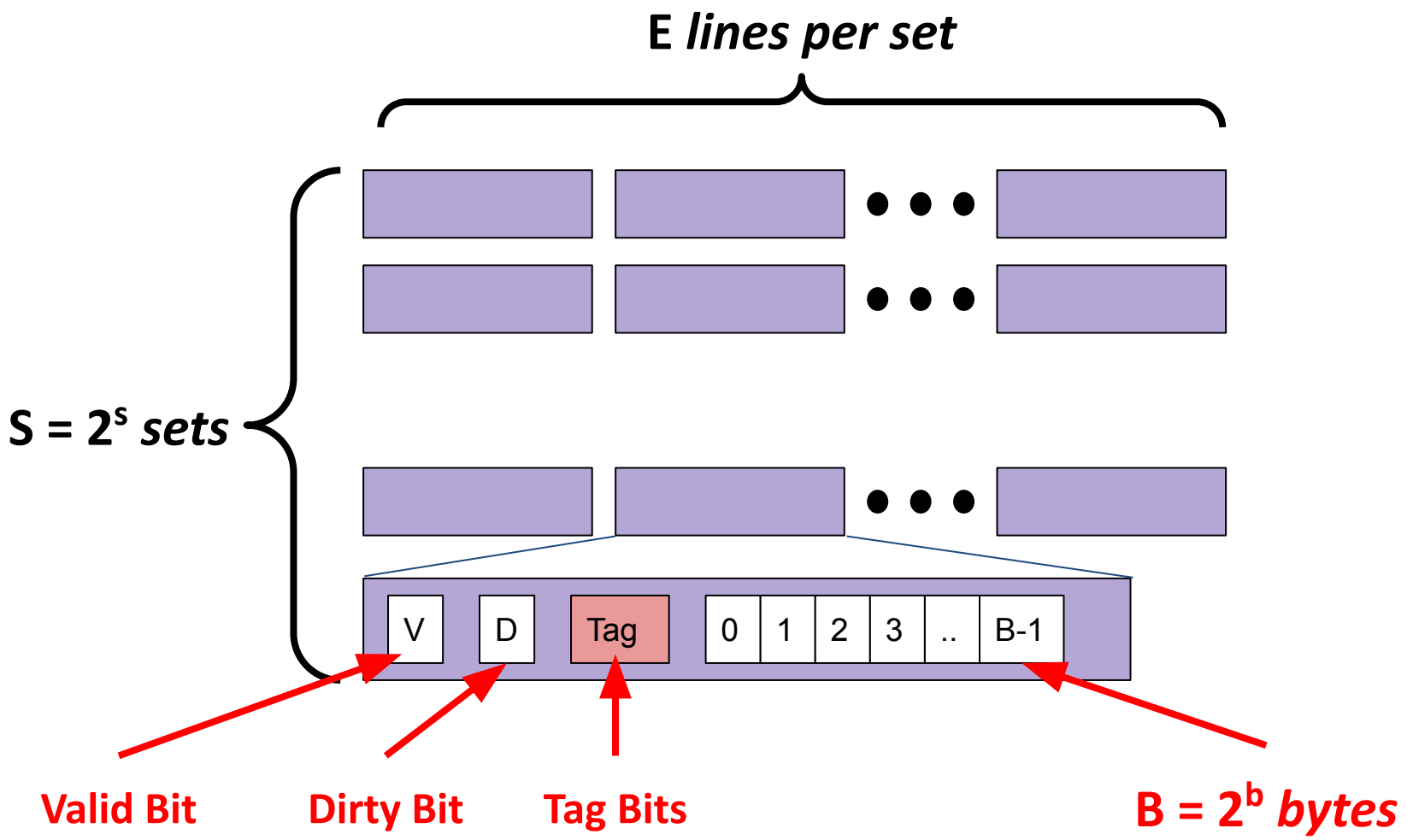
E – Number of *lines per set*



Cache Organization

- Each line stores data

b – Number of block offset *bits*
B = 2^b – Block Size



Cache Concepts: Cache Read

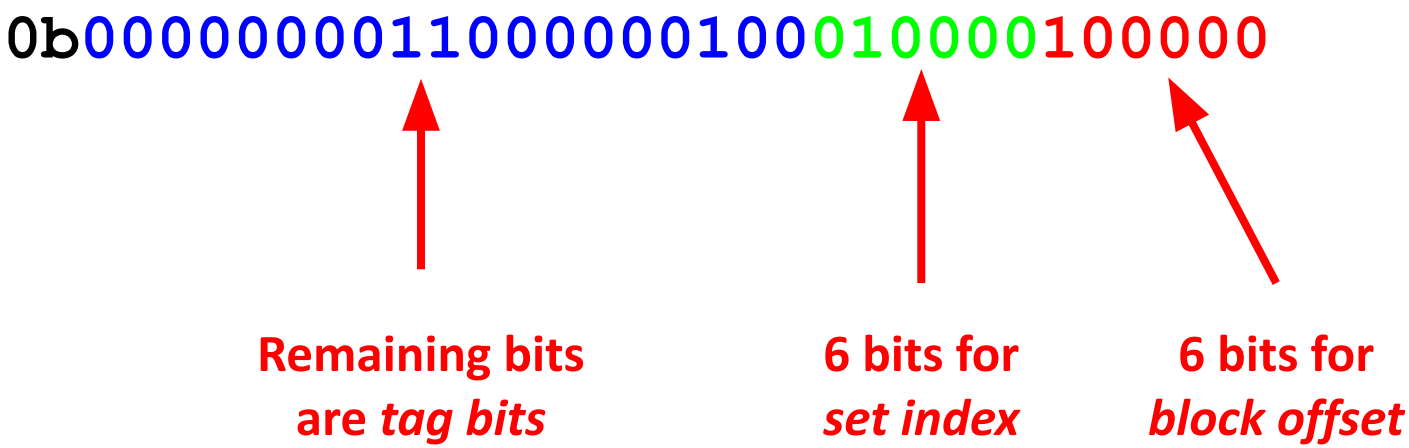
- We have an address that we want to look up in our cache.

`0x00604420`

- How do we search for it? Which set? Which line?
- Our *parameters* (`s` and `b`) determine how we partition the bits of our address.

Cache Concepts: Cache Read

- Our *parameters* (**s** and **b**) determine how we partition the bits of our address.
- Suppose **s** = 6 and **b** = 6



Cache Concepts: Cache Read

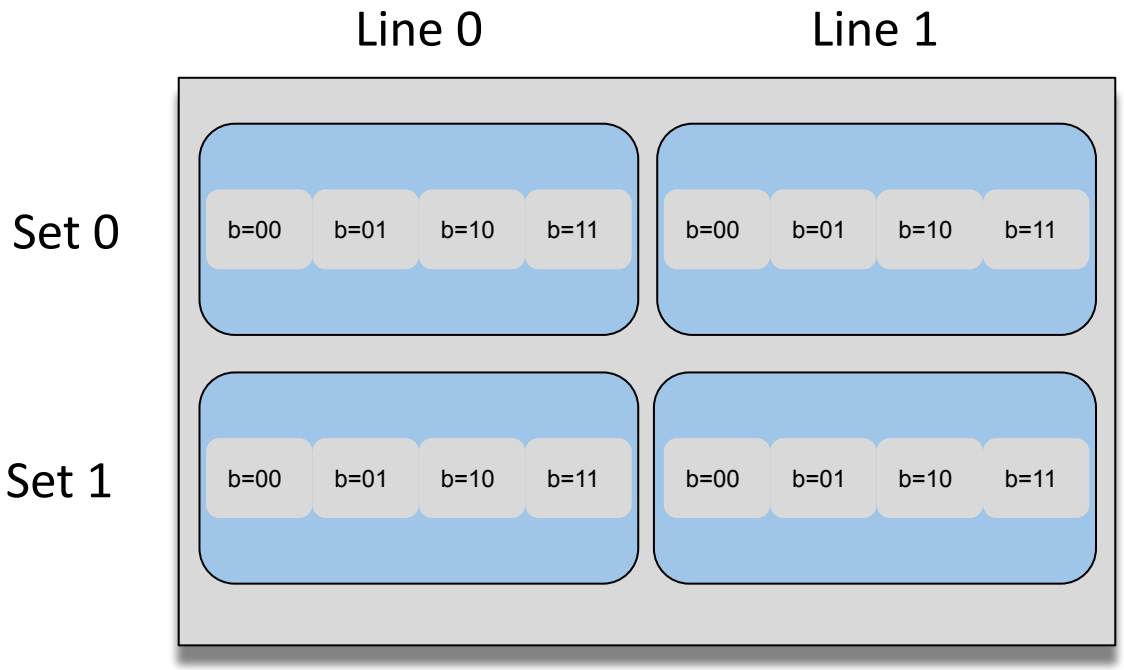
Tag: `0b000000011000000100`
Set: `0b010000`
Block Offset: `0b100000`

- These bits now tell us how to do the lookup in our cache!
- Use set index (`0b010000` = 16) to select the set
- Loop through lines in that set to find a matching tag (`0b000000011000000100`)
- If found and valid bit is set: ***Hit!***
  - Locate data starting at byte offset (`0b100000`)

Example Trace

Example Trace

- We will use the following configuration:
  - **`s`** = 1, **`E`** = 2, **`b`** = 2



Example Trace: Reading a Trace

bpr.trace

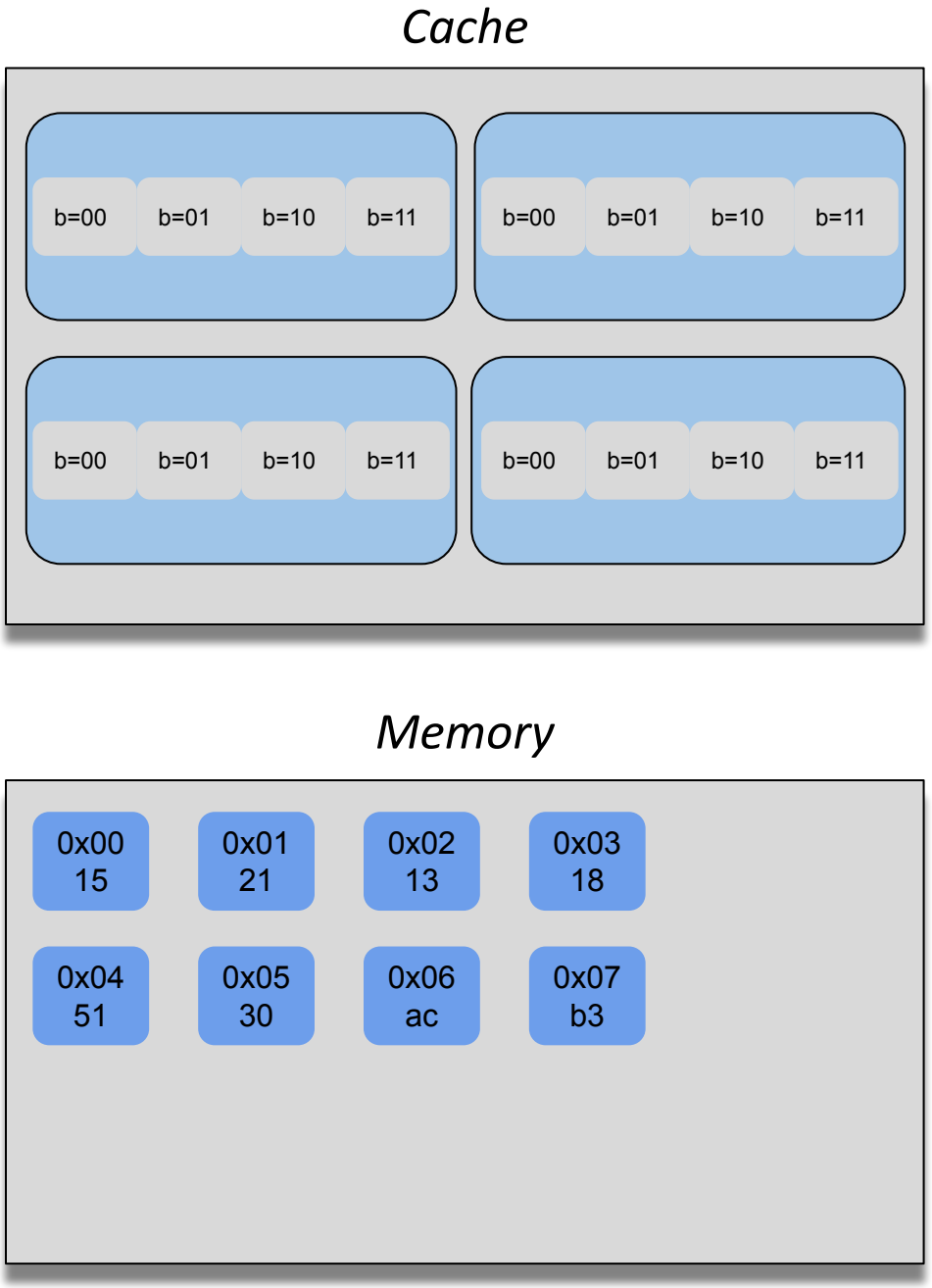
```
L 0x00,1
L 0x00,1
L 0x01,1
S 0x02,1
L 0x05,1
L 0x04,1
L 0x08,1
L 0x00,1
L 0x10,1
L 0x09,1
L 0x18,1
L 0x20,1
L 0x00,1
```

- `op <Addr>, <Size>`
- `op`:
  - `L` – **Load**
  - `S` – **Store**
- Note generally, we will **not** concern ourselves with the value being read/written
- For this trace however, we demonstrate write behavior through data values

Example Trace

```
...
L 0x00,1
...
```

Will this instruction result in a hit or a miss?



Example Trace

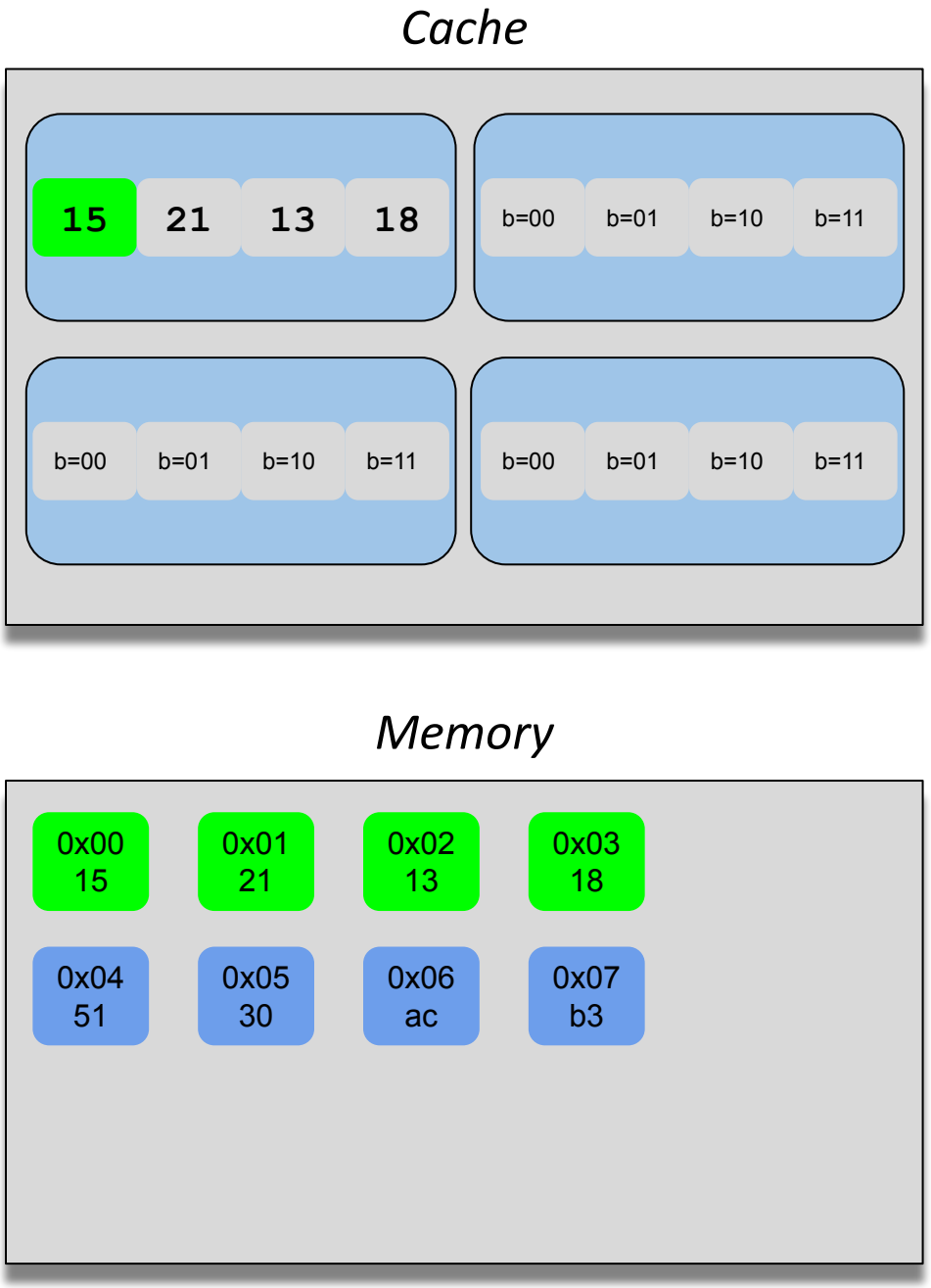
```
...
L 0x00,1
...
```

Miss

Why that line?

Where are those values from?

What kind of miss is this?



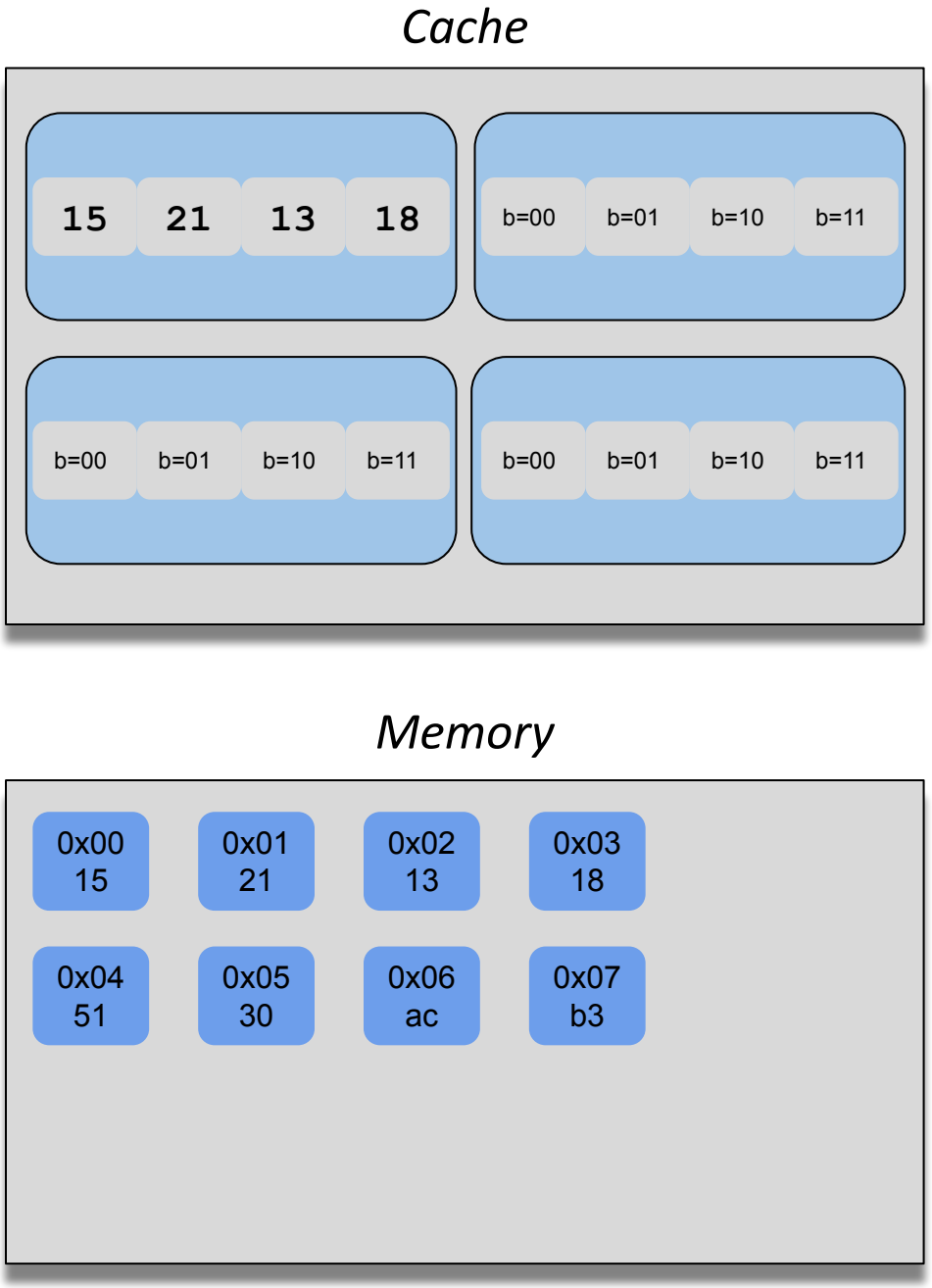
Example Trace

```
...
L 0x00,1
L 0x00,1
...
```

Miss

???

Will this instruction result in a hit or a miss?

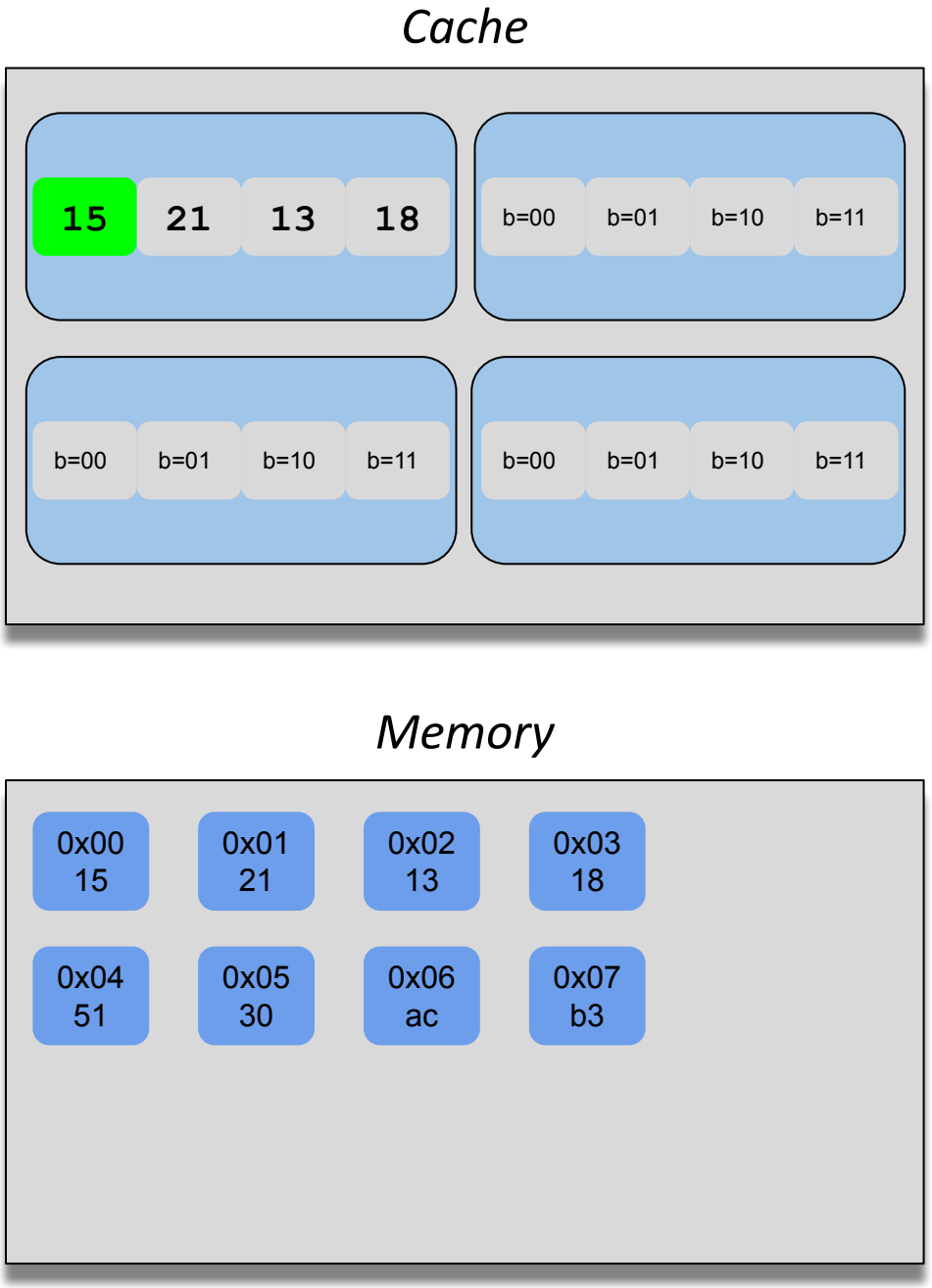


Example Trace

```
...
L 0x00,1
L 0x00,1
...
```

Miss

Hit!



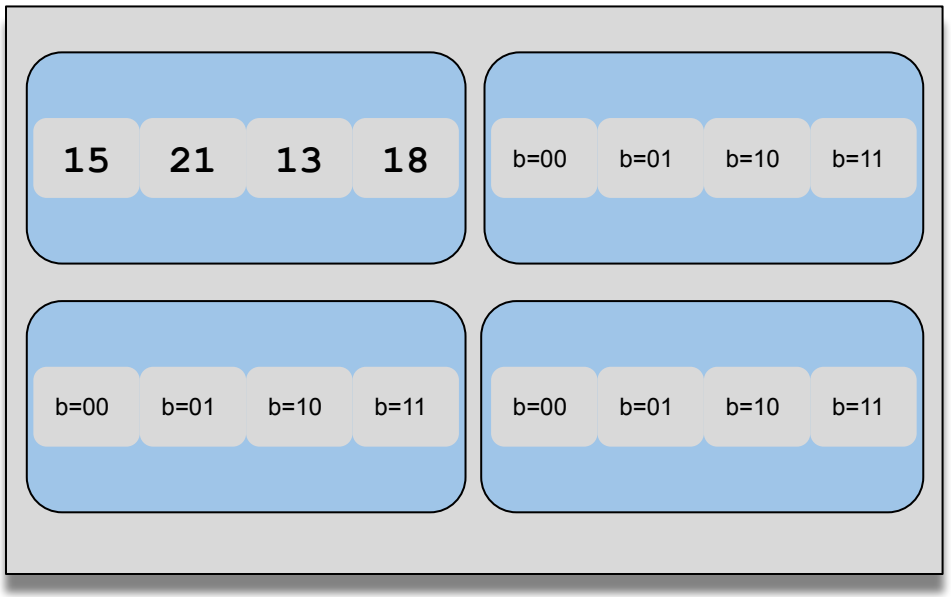
Example Trace

```
...
L 0x00,1
L 0x00,1
L 0x01,1
...
```

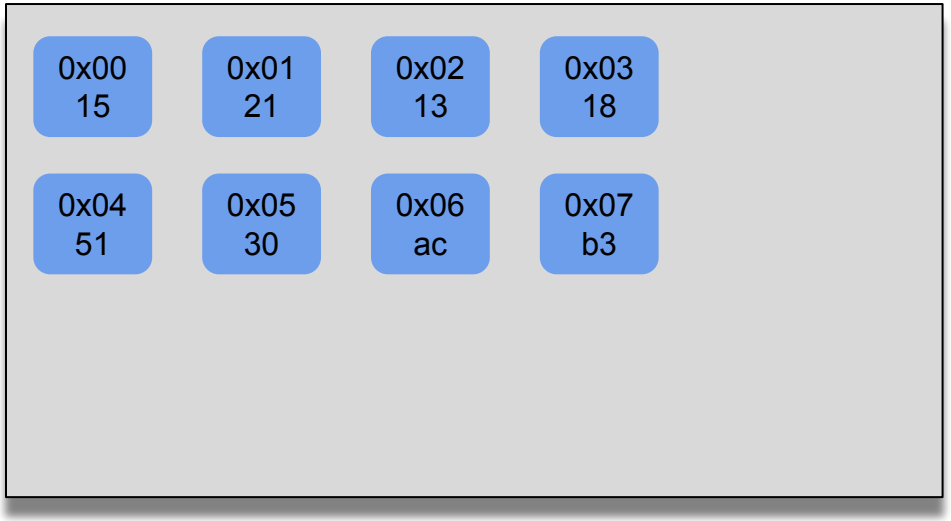
Miss
Hit!
???

Will this instruction result in a hit or a miss?

Cache



Memory



Example Trace

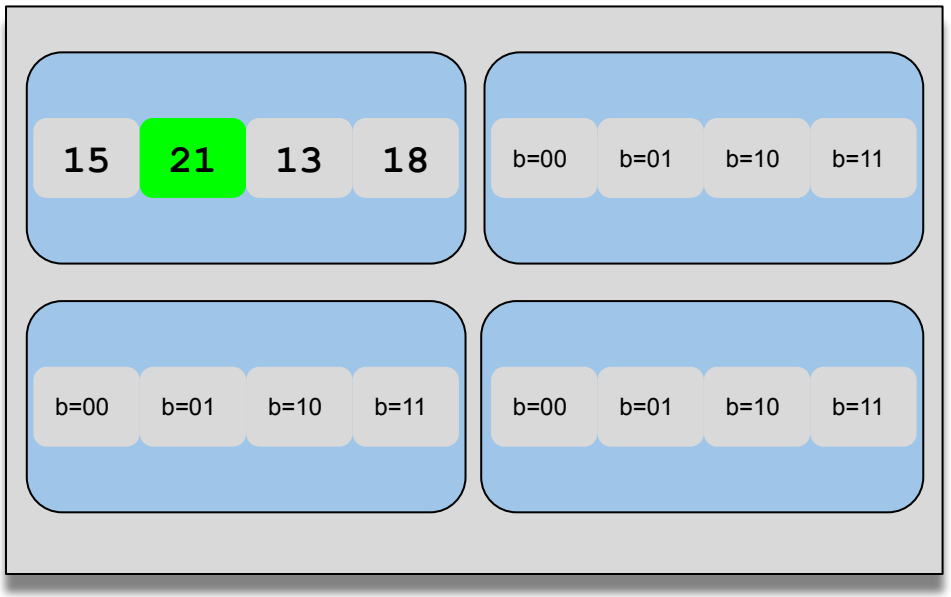
```
...
L 0x00,1
L 0x00,1
L 0x01,1
...
```

Miss
Hit!
Hit!

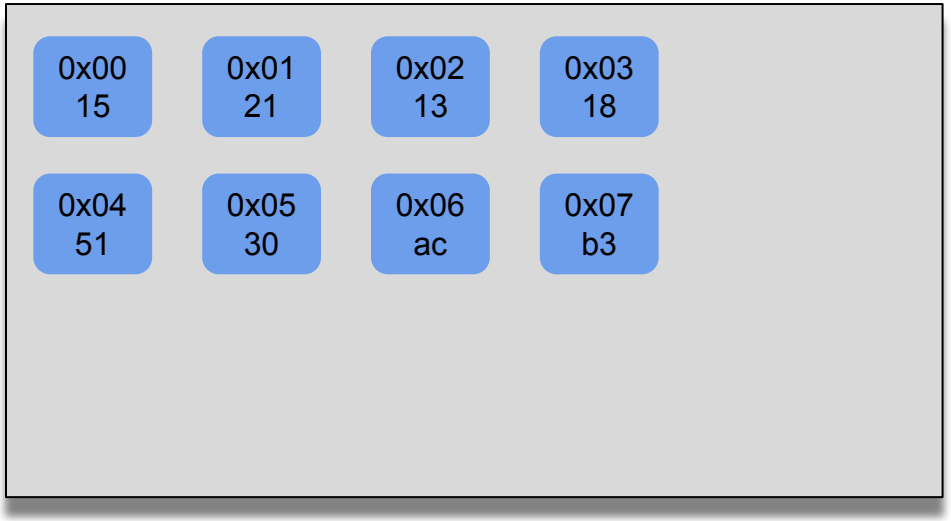
Not a miss!

We had already loaded all four bytes of the line into cache. Why?

Cache



Memory



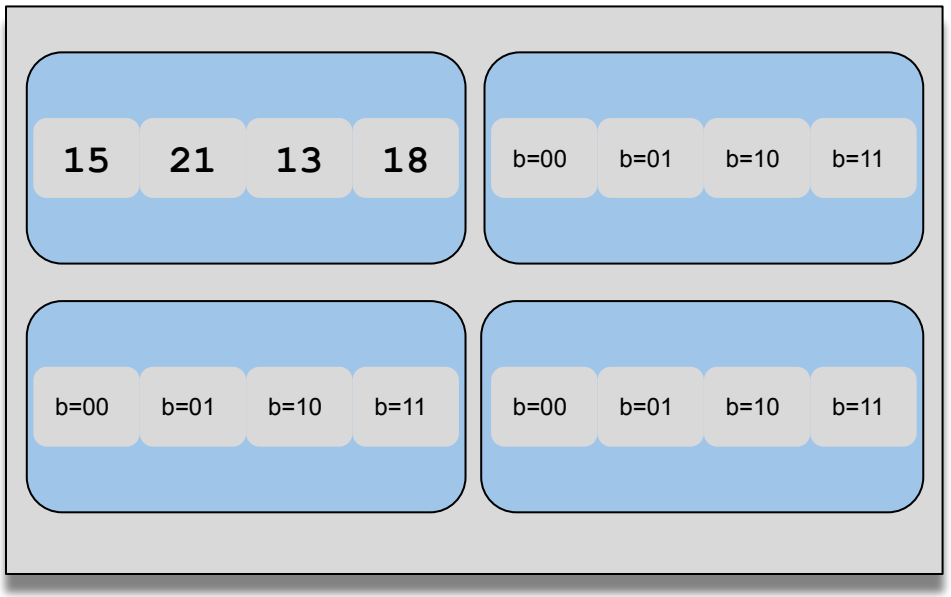
Example Trace

```
...
L 0x00,1
L 0x01,1
S 0x02,1
...
```

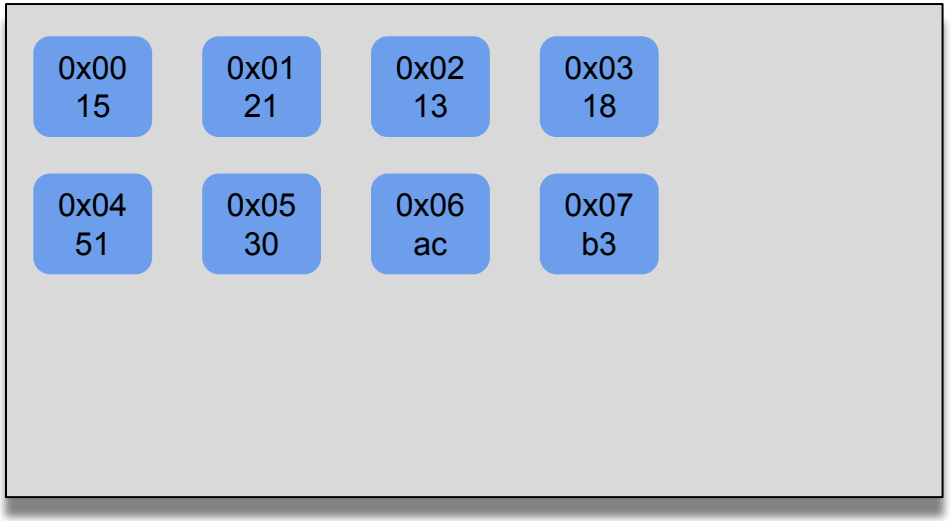
Hit!
Hit!
???

Will this instruction result in a hit or a miss?

Cache



Memory



Example Trace

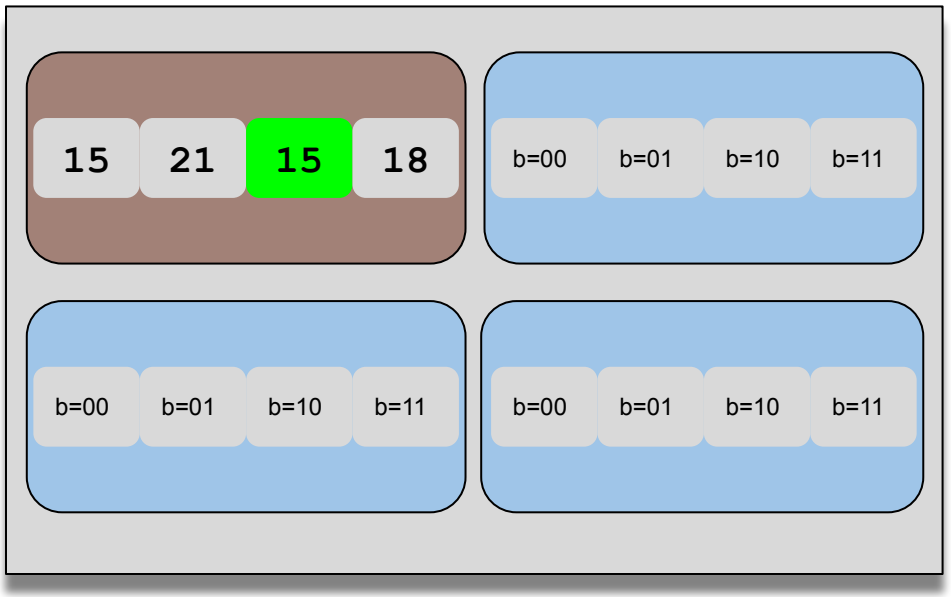
```
...
L 0x00,1
L 0x01,1
S 0x02,1
...
```

Hit!
Hit!
Hit!

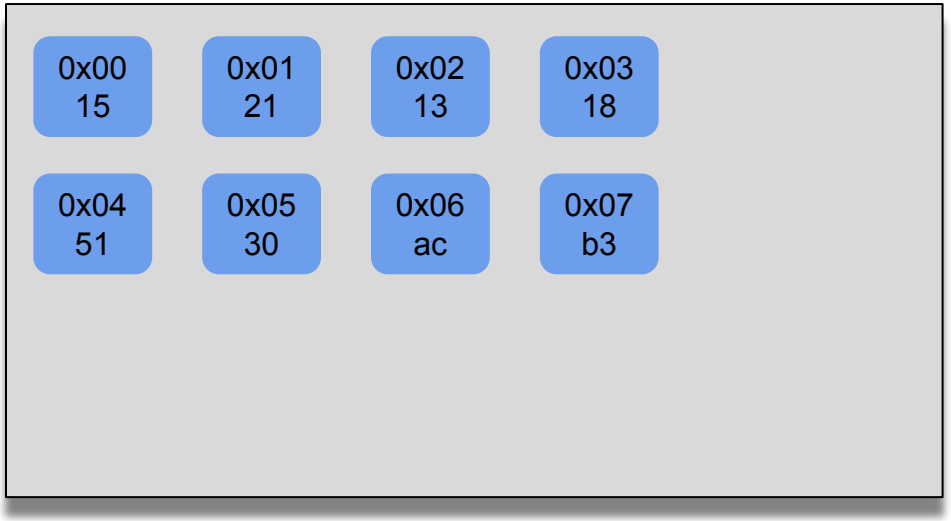
Write hit!

Set dirty bit.

Cache



Memory



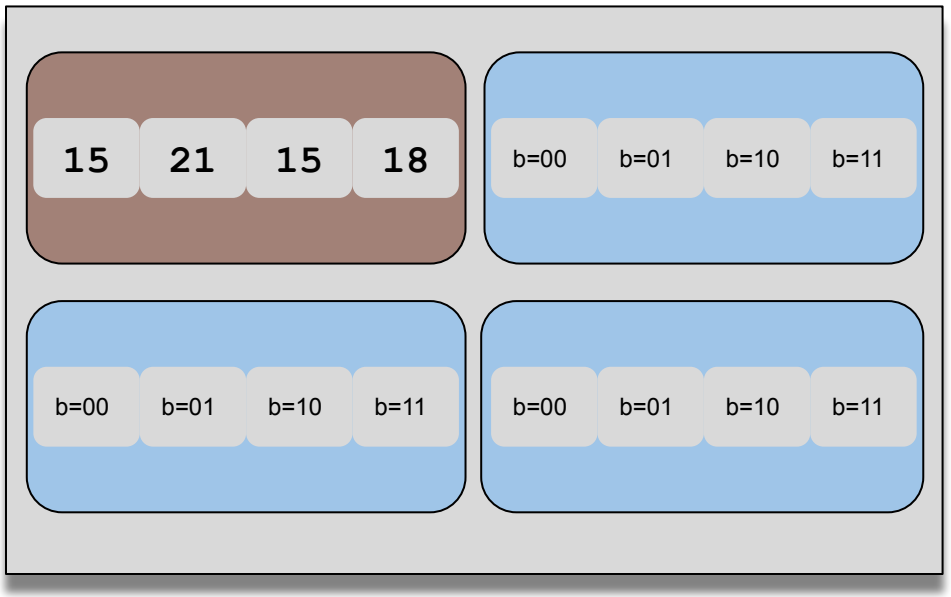
Example Trace

```
...
L 0x01,1
S 0x02,1
L 0x05,1
...
```

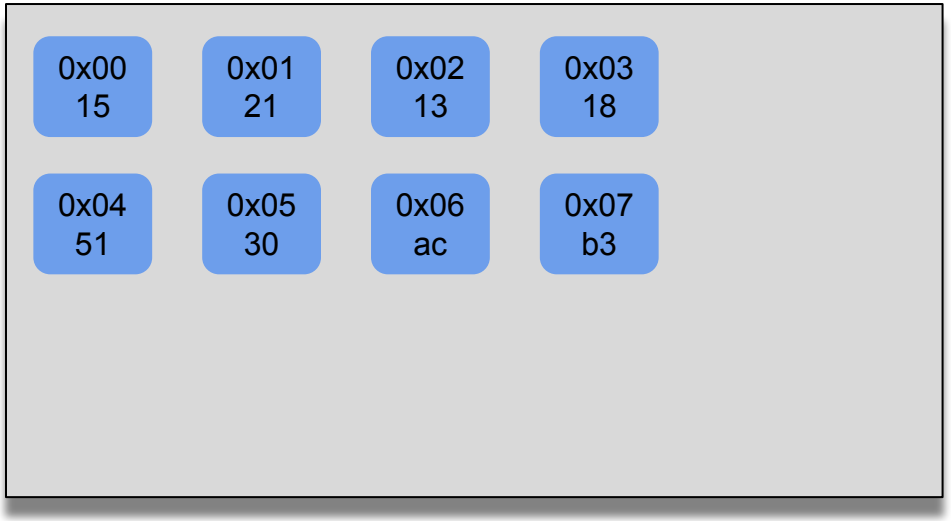
Hit!
Hit!
???

Will this instruction result in a hit or a miss?

Cache



Memory



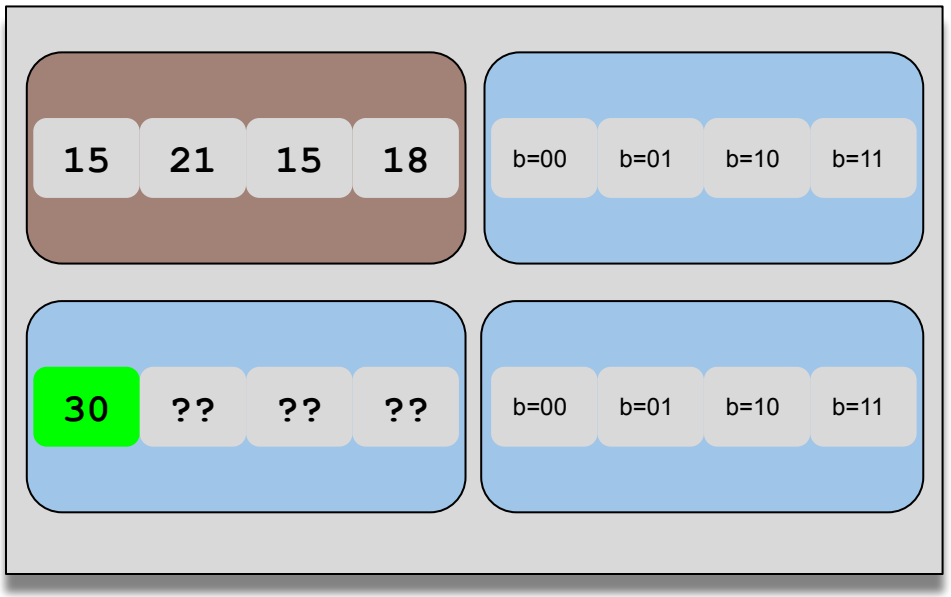
Example Trace

```
...
L 0x01,1
S 0x02,1
L 0x05,1
...
```

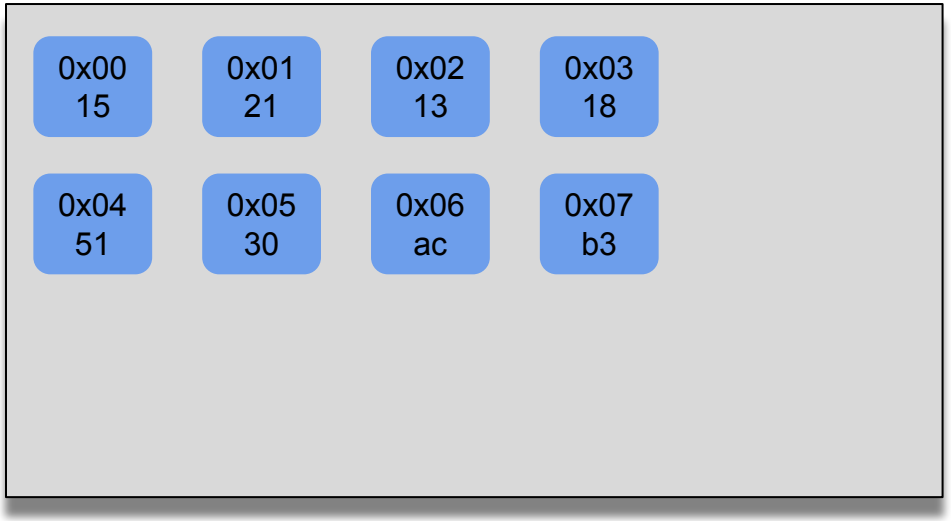
Hit!
Hit!
Miss

Do we load just one byte like this?

Cache



Memory



Example Trace

```
...
L 0x01,1
S 0x02,1
L 0x05,1
...
```

Hit!
Hit!
Miss

Do we load just one byte like this?

No!

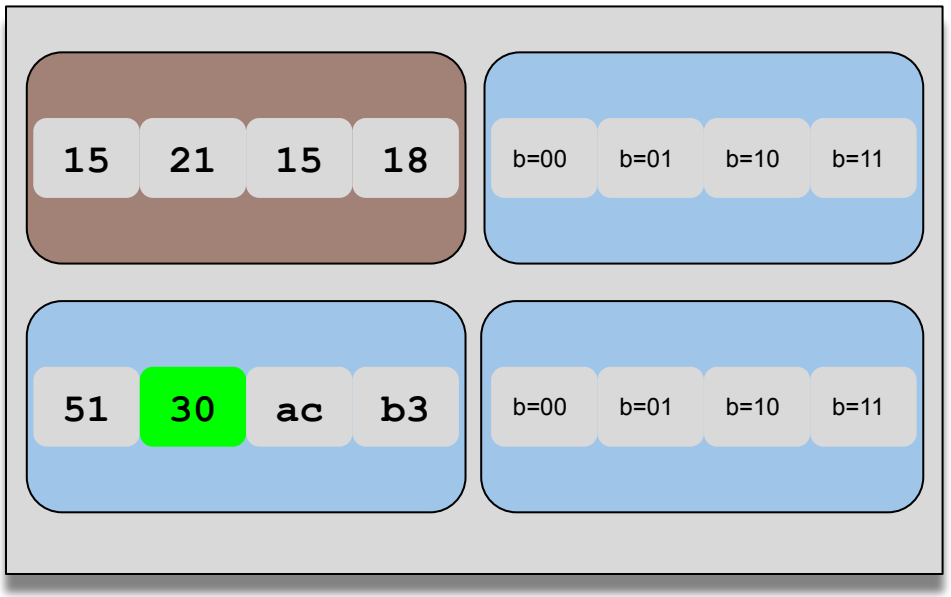
Example Trace

```
...
L 0x01,1
S 0x02,1
L 0x05,1
...
```

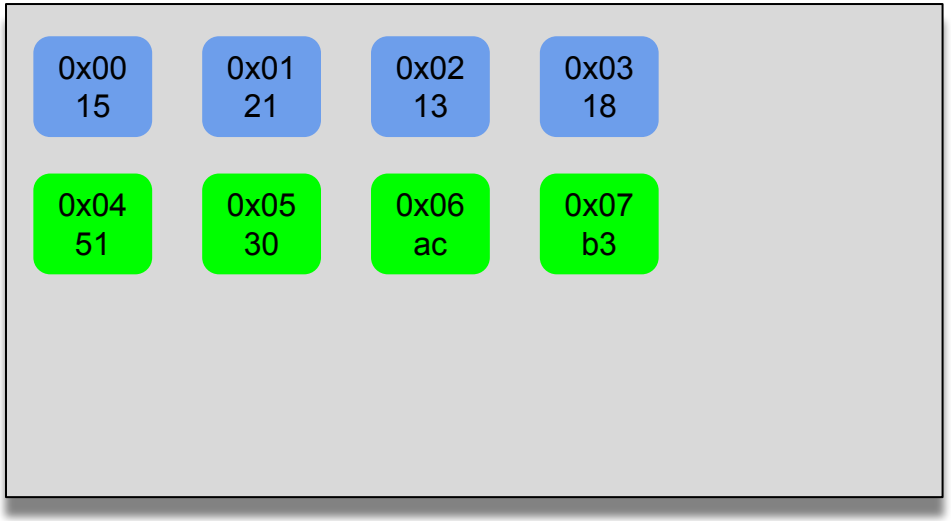
Hit!
Hit!
Miss

Why do we start with a byte from below address 5?

Cache



Memory



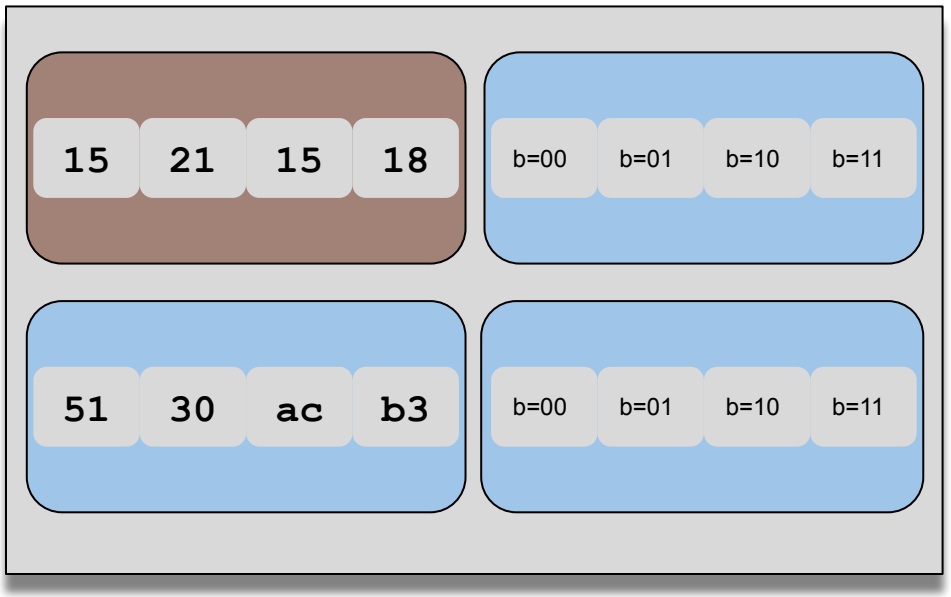
Example Trace

```
...
S 0x02,1
L 0x05,1
L 0x04,1
...
```

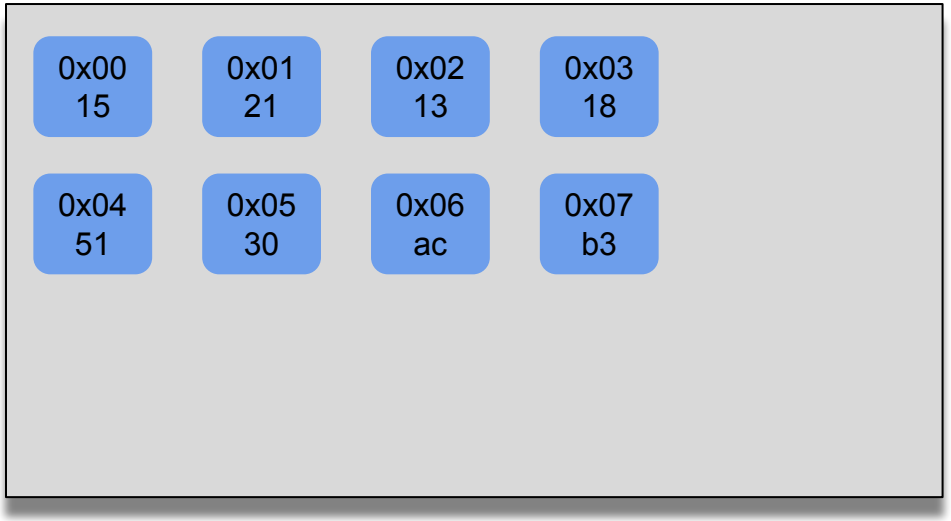
Hit!
Miss
???

Will this instruction result in a hit or a miss?

Cache



Memory

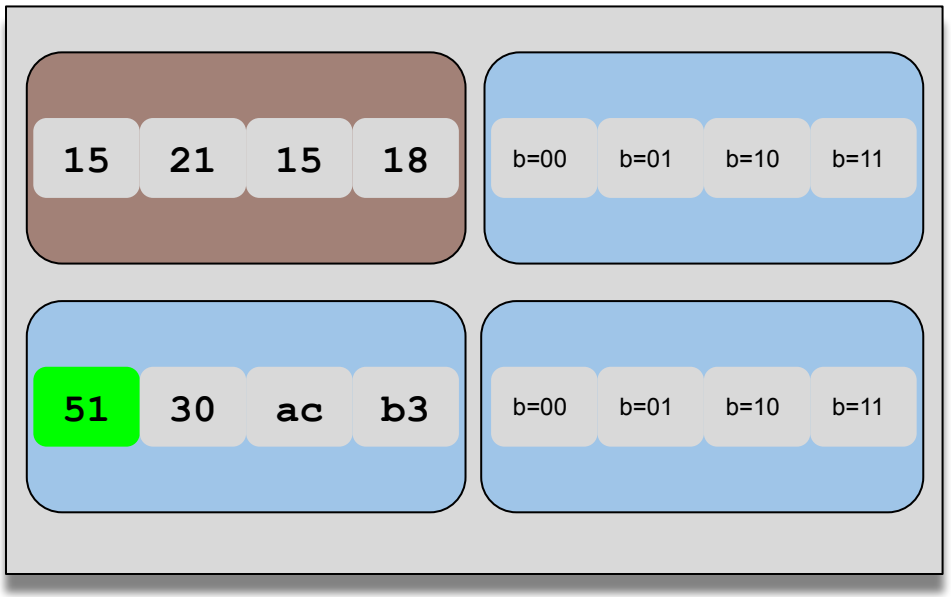


Example Trace

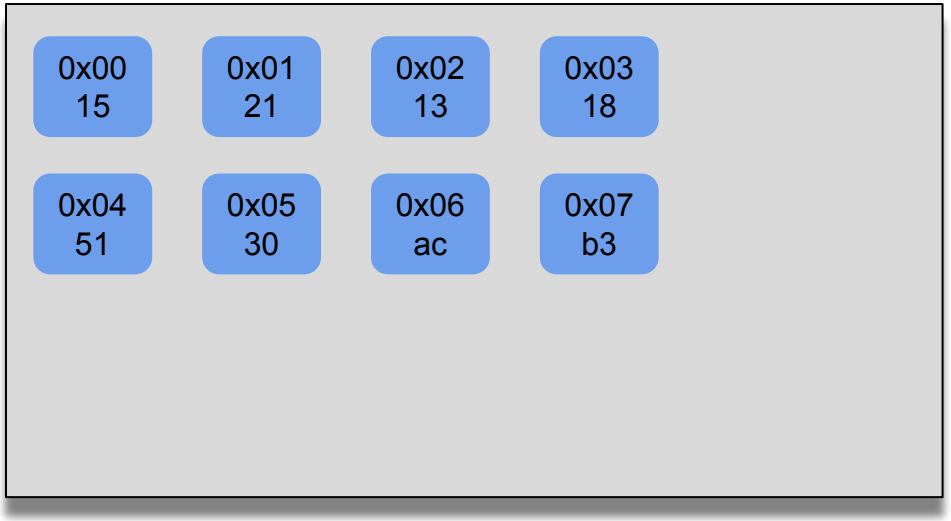
```
...
S 0x02,1
L 0x05,1
L 0x04,1
...
```

Hit!
Miss
Hit!

Cache



Memory



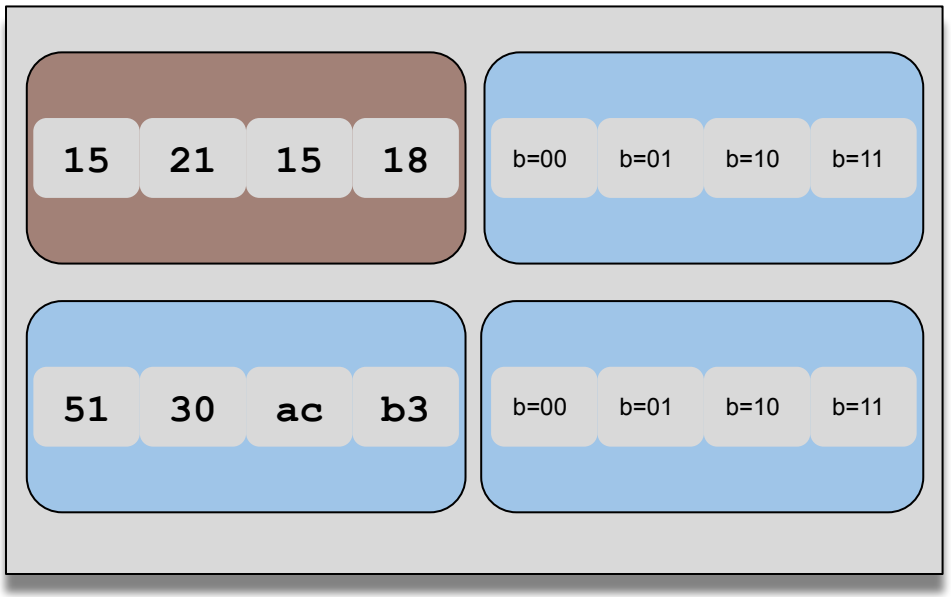
Example Trace

```
...
L 0x05,1
L 0x04,1
L 0x08,1
...
```

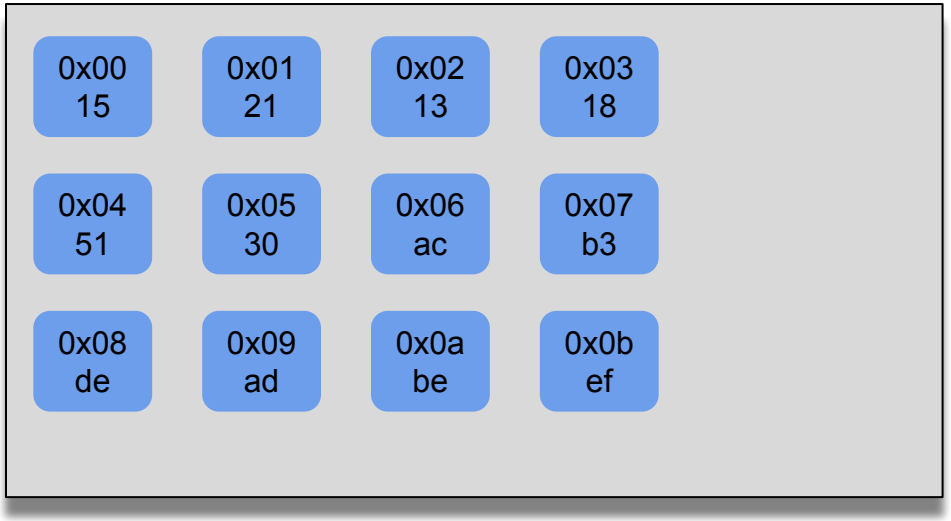
Miss
Hit!
???

Will this instruction result in a hit or a miss?

Cache



Memory



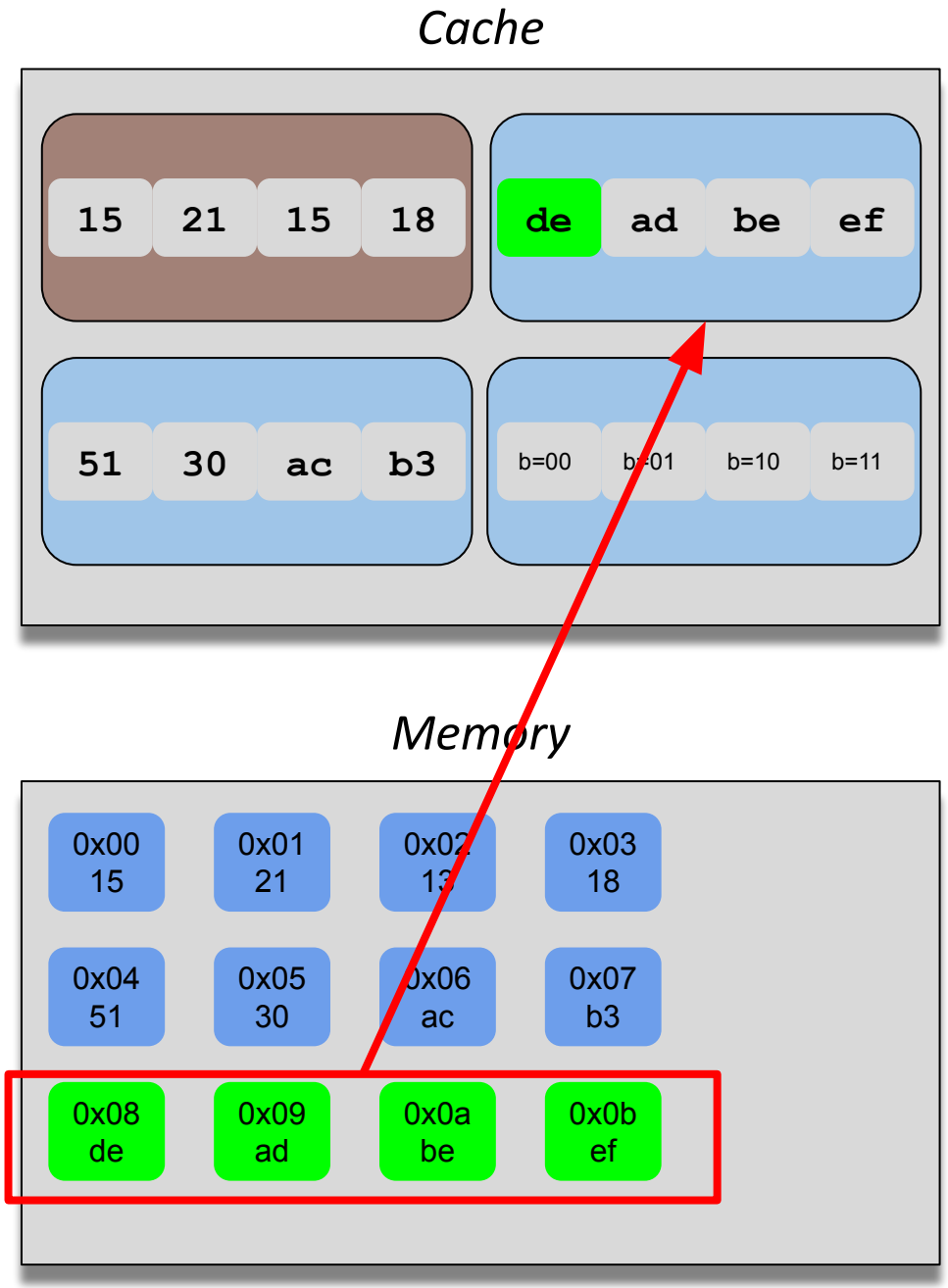
Example Trace

```
...
L 0x05,1
L 0x04,1
L 0x08,1
...
```

Miss
Hit!
Miss

Miss!

We had a free line, so just load the data into there.



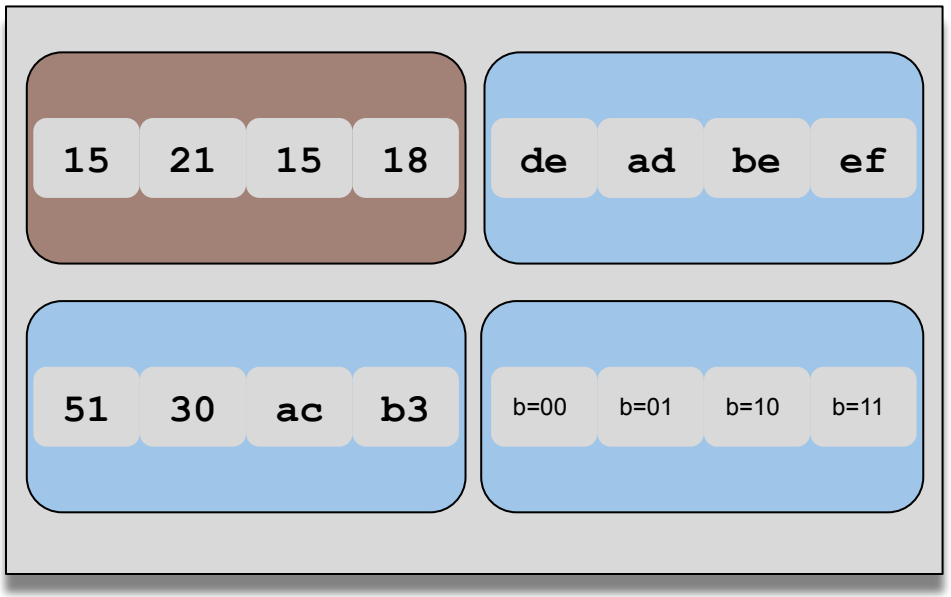
Example Trace

```
...
L 0x04,1
L 0x08,1
L 0x00,1
...
```

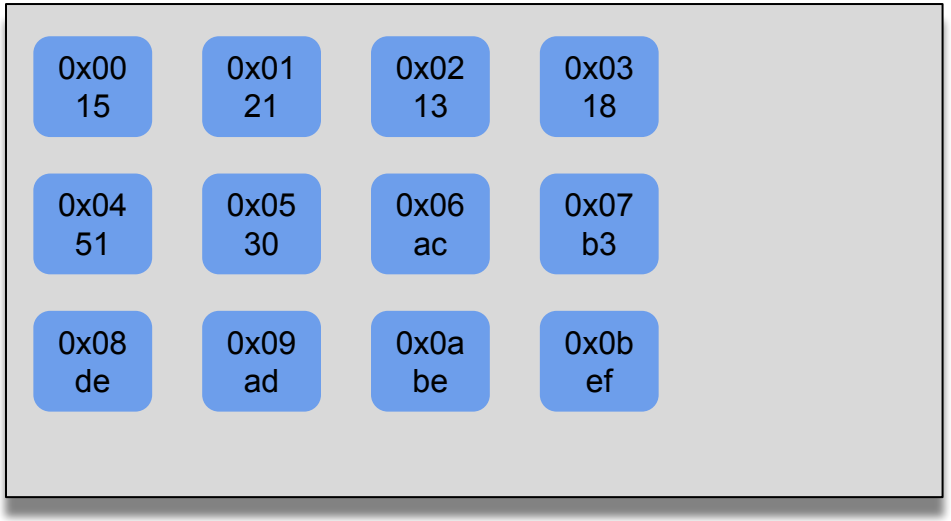
Hit!
Miss
???

Will this instruction result in a hit or a miss?

Cache



Memory

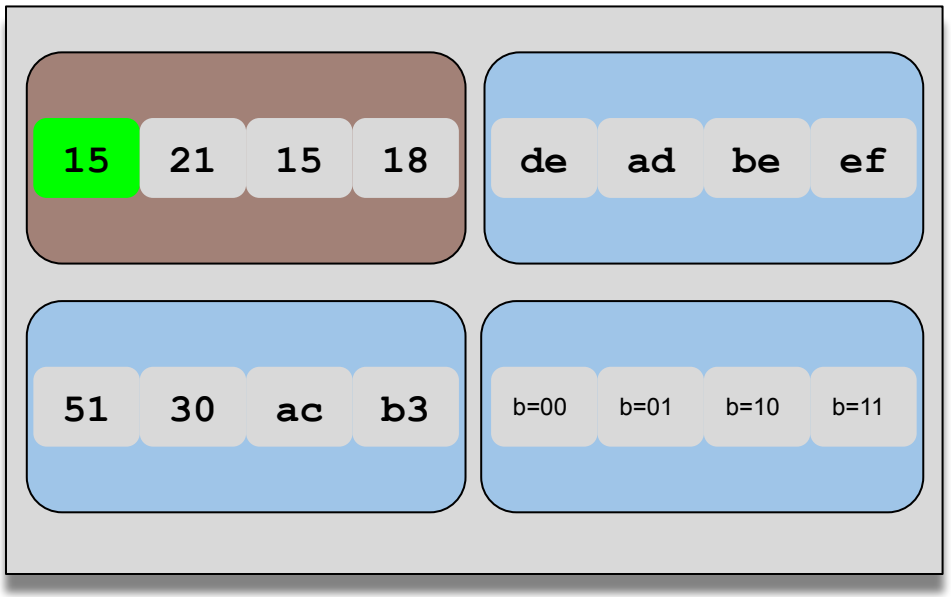


Example Trace

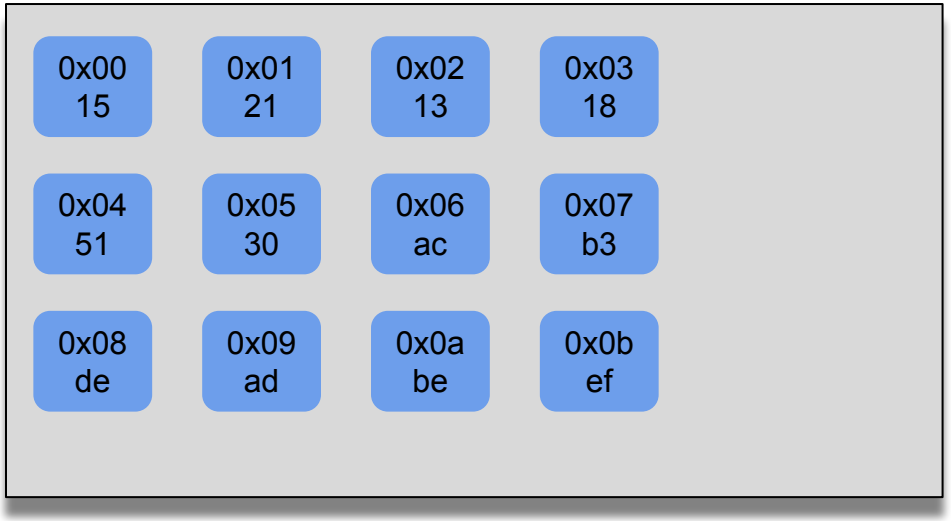
```
...
L 0x04,1
L 0x08,1
L 0x00,1
...
```

Hit!
Miss
Hit!

Cache



Memory



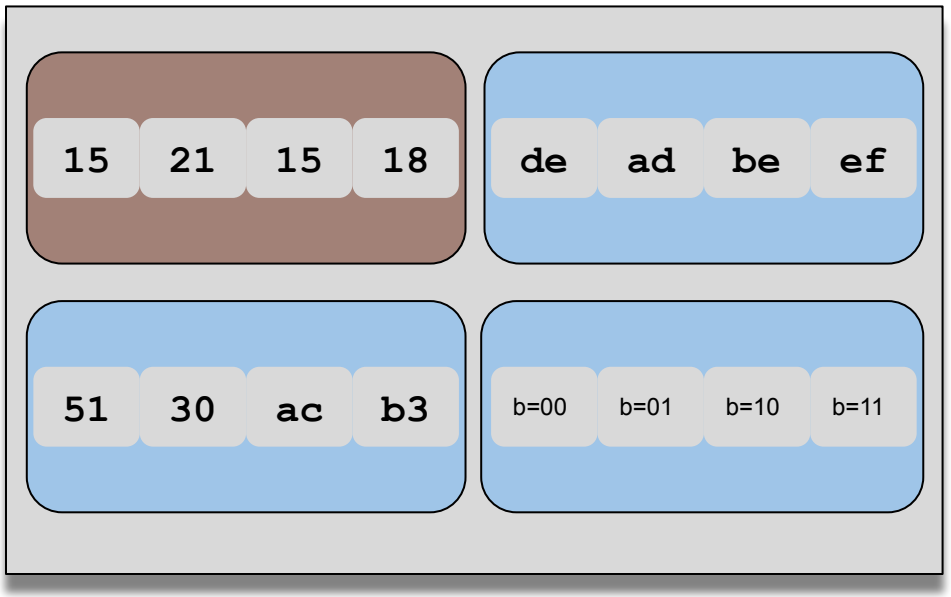
Example Trace

```
...
L 0x08,1
L 0x00,1
L 0x10,1
...
```

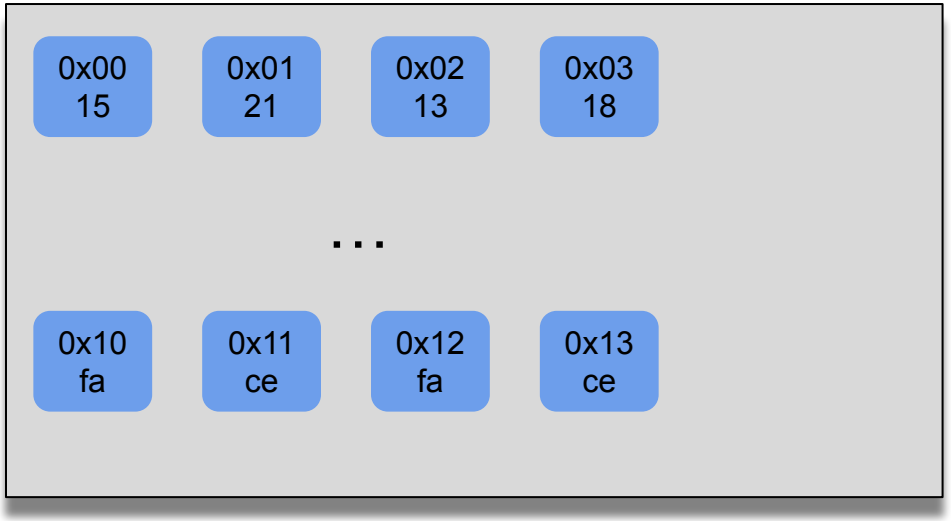
```
Miss
Hit!
???
```

Will this instruction result in a hit or a miss?

Cache



Memory



Example Trace

```
...
L 0x08,1
L 0x00,1
L 0x10,1
...
```

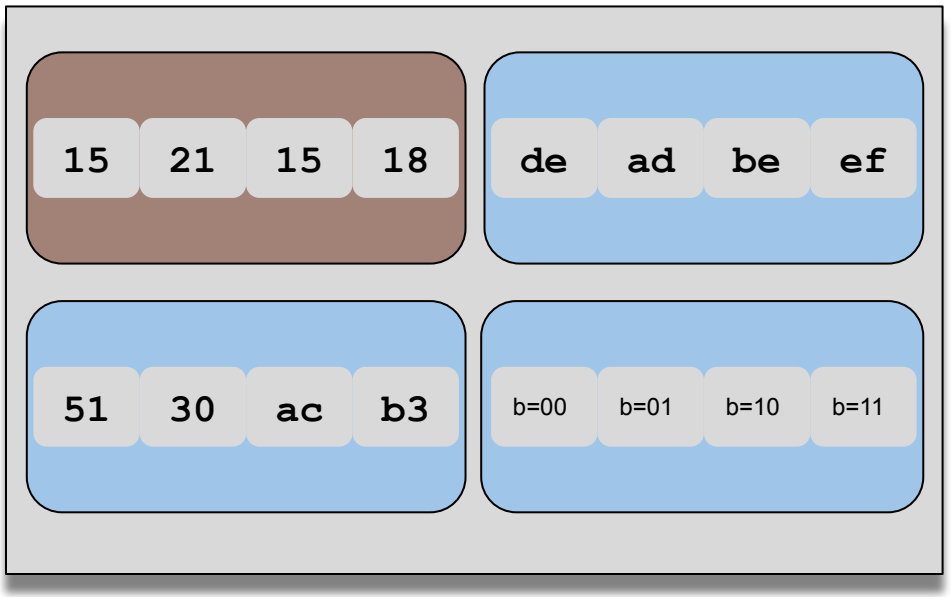
Miss
Hit!
Miss

What kind of miss is this?

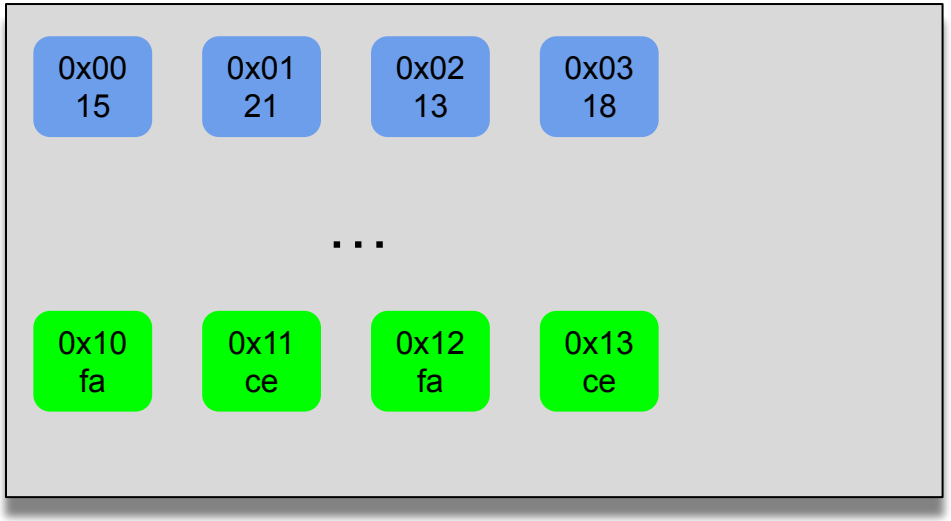
0x10 = 16 = 0b10000

=> Set Index 0
=> Have to evict!

Cache

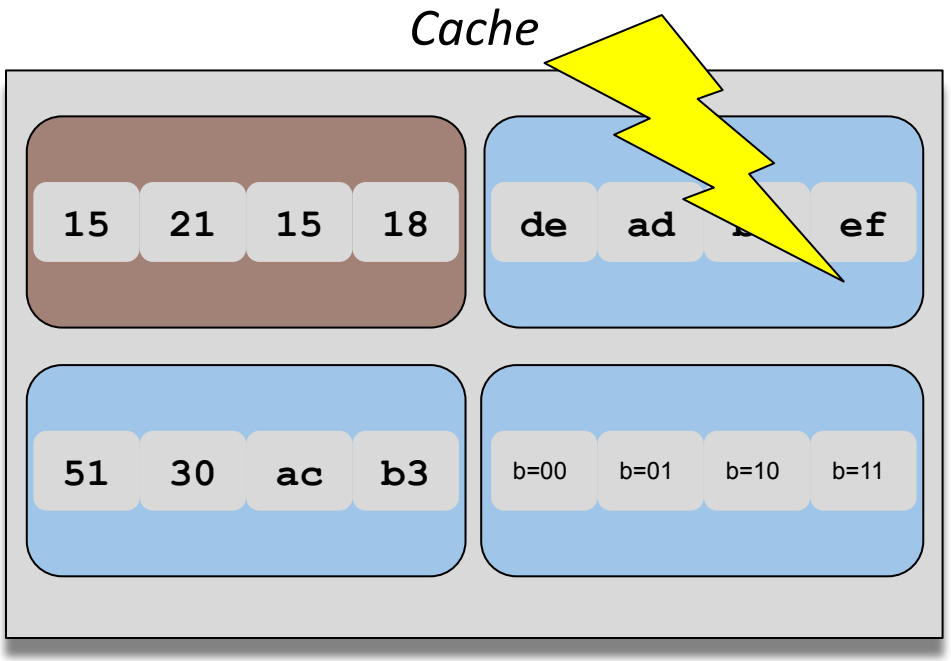


Memory

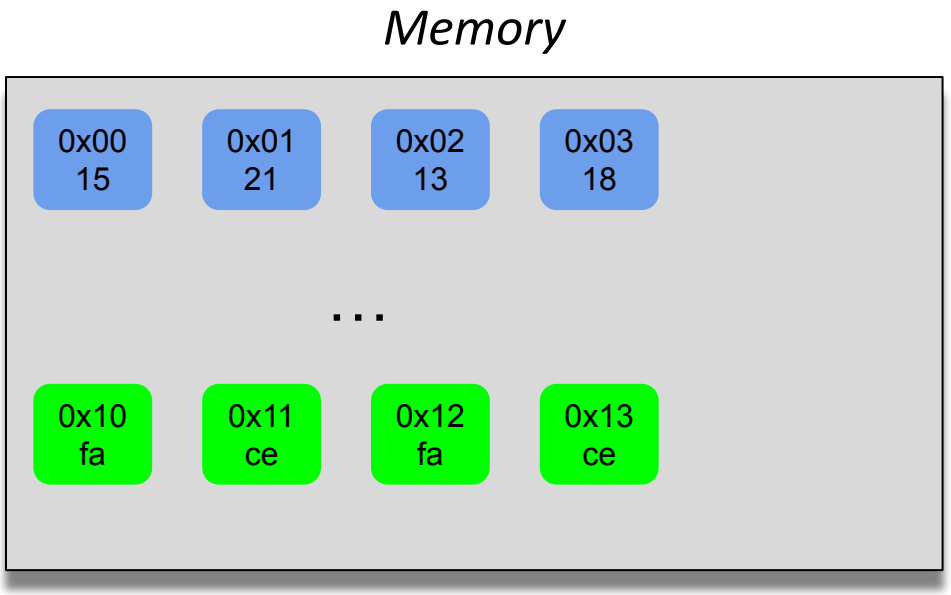


Example Trace

```
...
L 0x08,1    Miss
L 0x00,1    Hit!
L 0x10,1    Miss
...
```



1. ***Cold Miss*** *(first time seeing this block)*
1. ***Evict LRU*** *(Least Recently Used) line from set 0*

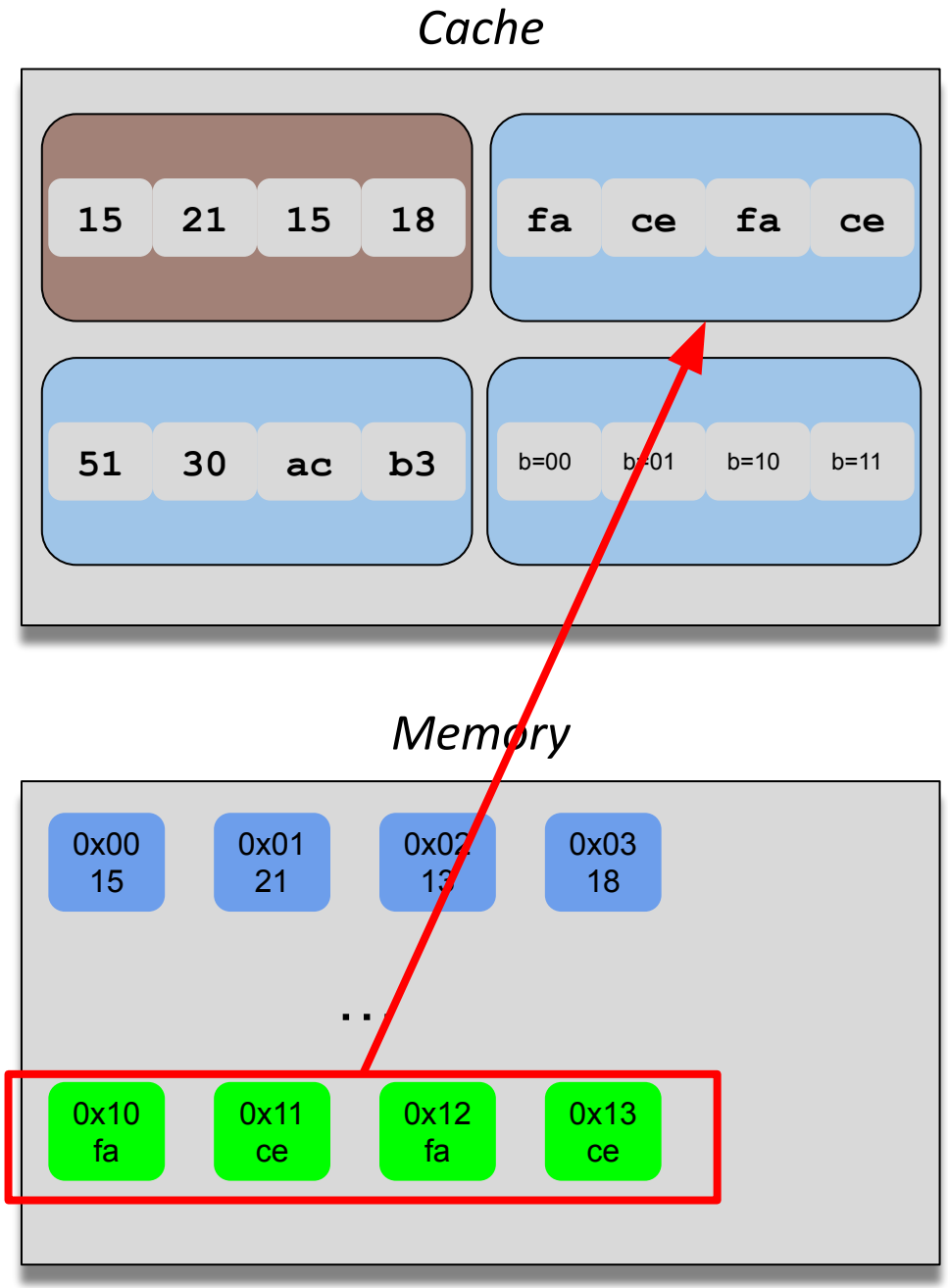


Example Trace

```
...
L 0x08,1
L 0x00,1
L 0x10,1
...
```

Miss
Hit!
Miss

Load new data into line



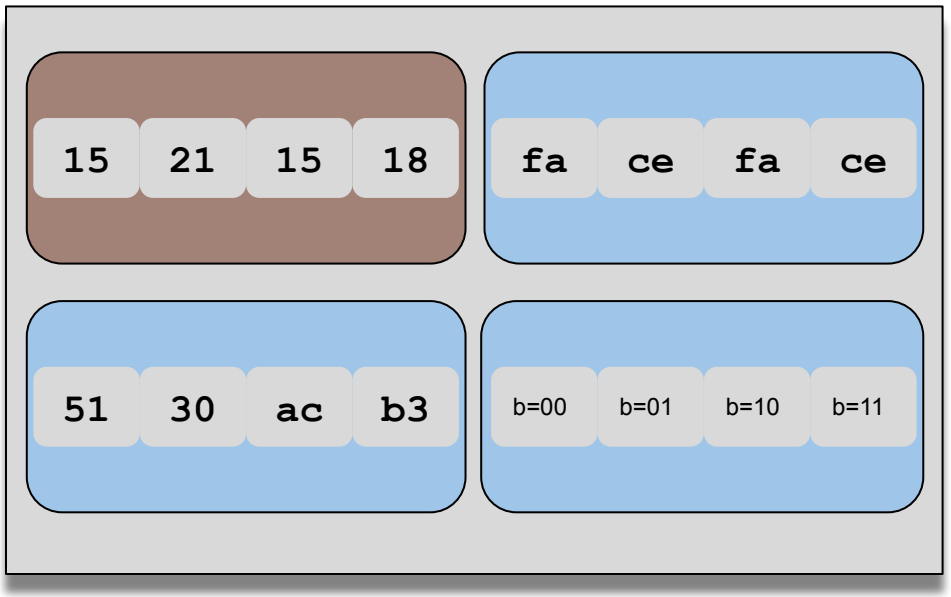
Example Trace

```
...
L 0x00,1
L 0x10,1
L 0x09,1
...
```

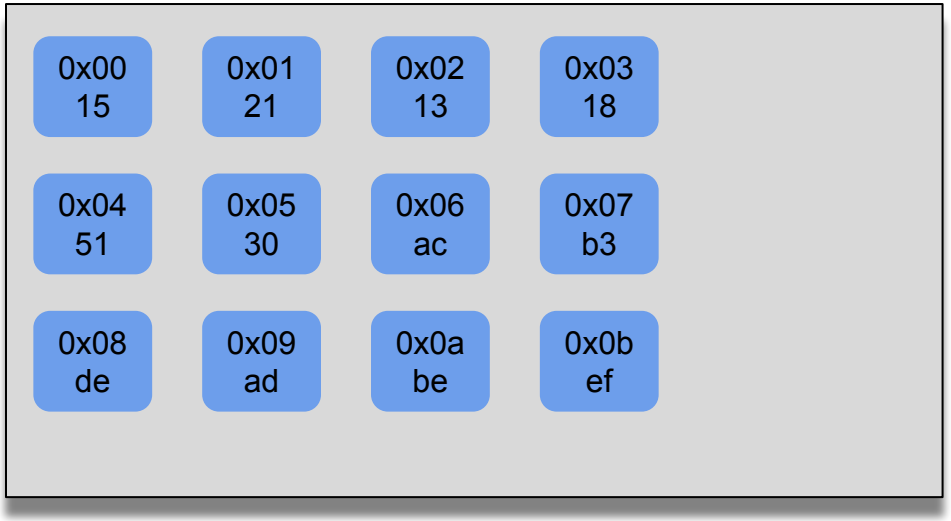
Hit!
Miss
???

Will this instruction result in a hit or a miss?

Cache



Memory



Example Trace

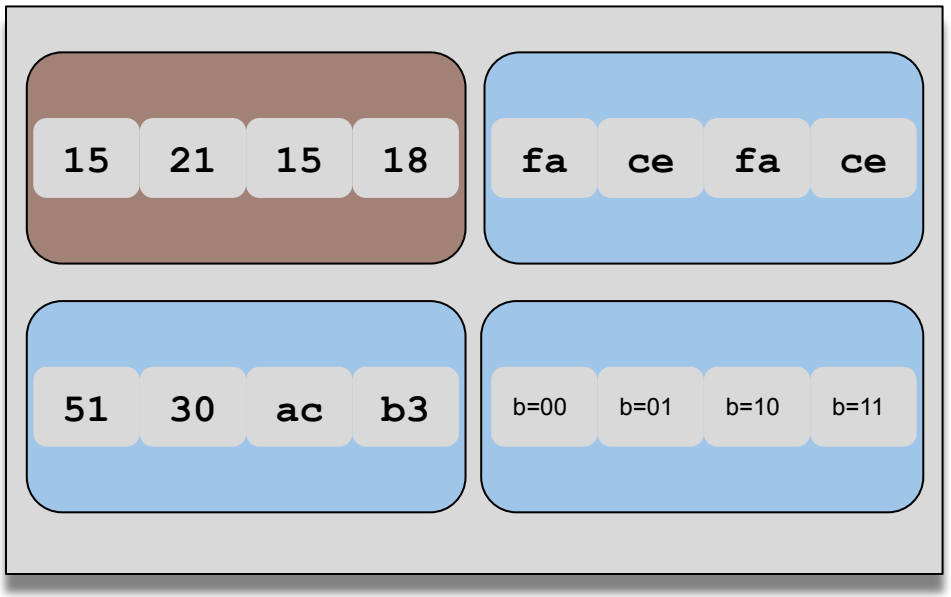
```
...
L 0x00,1
L 0x10,1
L 0x09,1
...
```

Hit!
Miss
Miss

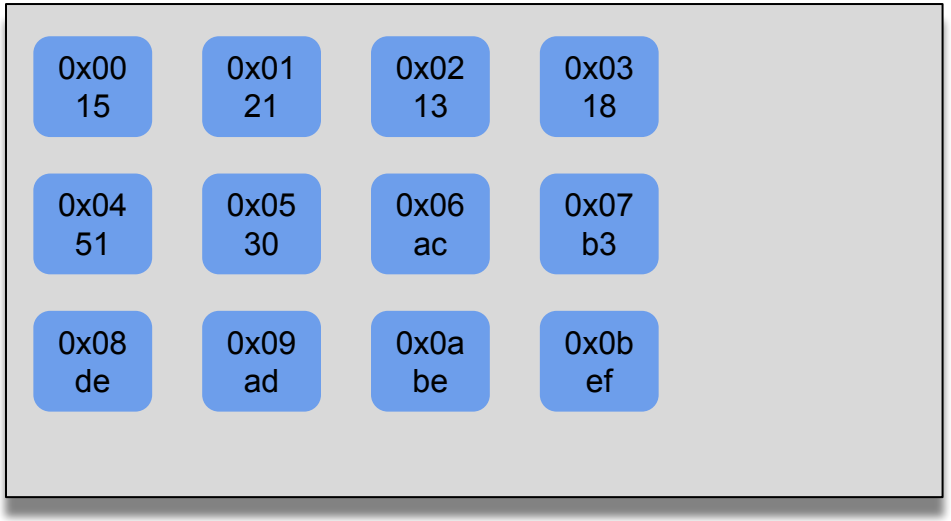
What kind of miss is this?

Has the block been in the cache before?

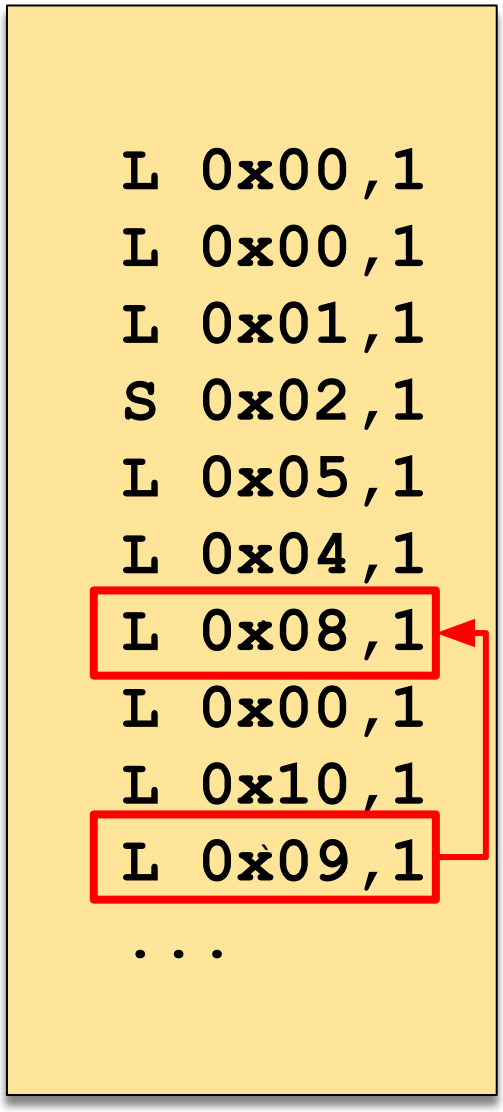
Cache



Memory

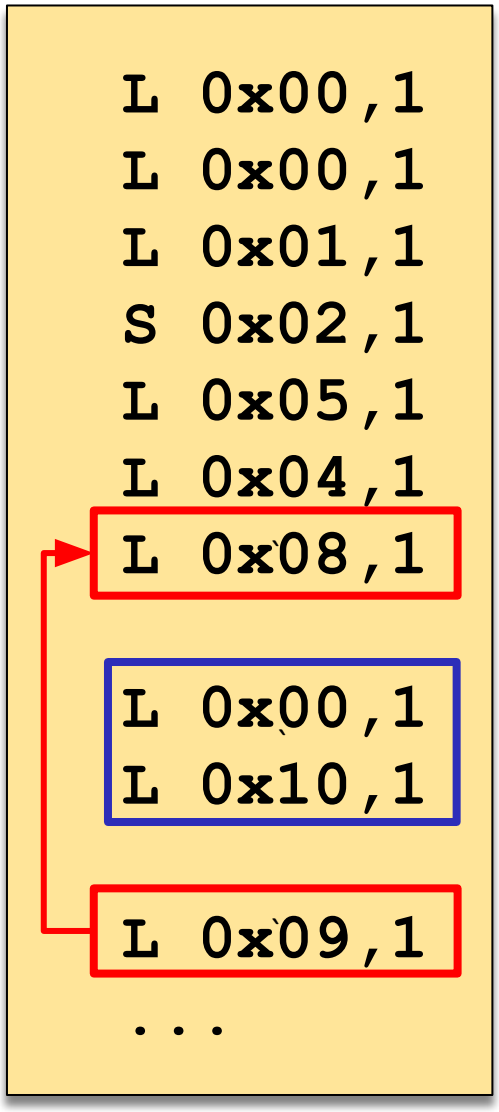


Cache Concepts: Conflict/Capacity Misses



- Has this block been in the cache before?
- Yes!
- If we've seen the block before:
  - Not a cold miss
  - Either a *conflict miss* or a *capacity miss*.

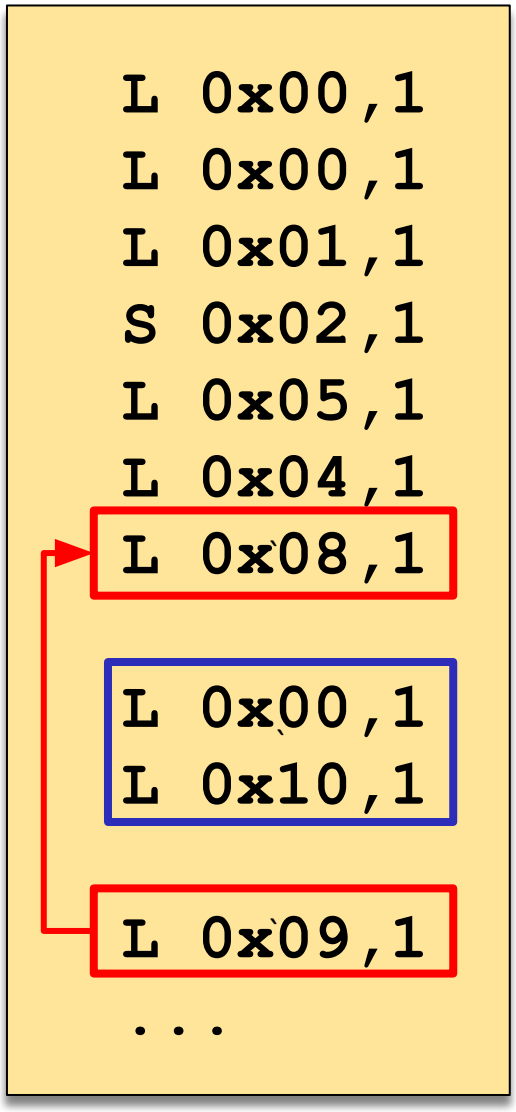
Cache Concepts: Conflict/Capacity Misses



How to distinguish between the two:

1. Find the last reference to that block in the trace.
2. Count the number of *unique* blocks referenced *in-between*:
    a. If the number is greater than or equal to the total number of lines in the cache: ***Capacity Miss***
    b. Otherwise: ***Conflict Miss***

Cache Concepts: Conflict/Capacity Misses



- In this case:
  - *Two* unique blocks in between current reference and last reference.
  - But we have *four* total lines in the cache
  - So we have a ***Conflict Miss***.

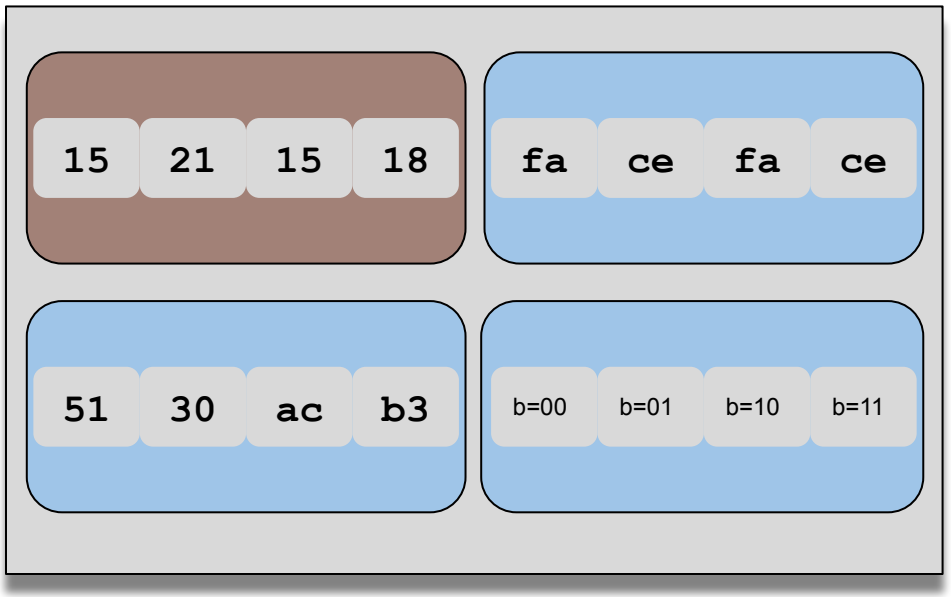
Example Trace

```
...
L 0x00,1    Hit!
L 0x10,1    Miss
L 0x09,1    Miss
...
```

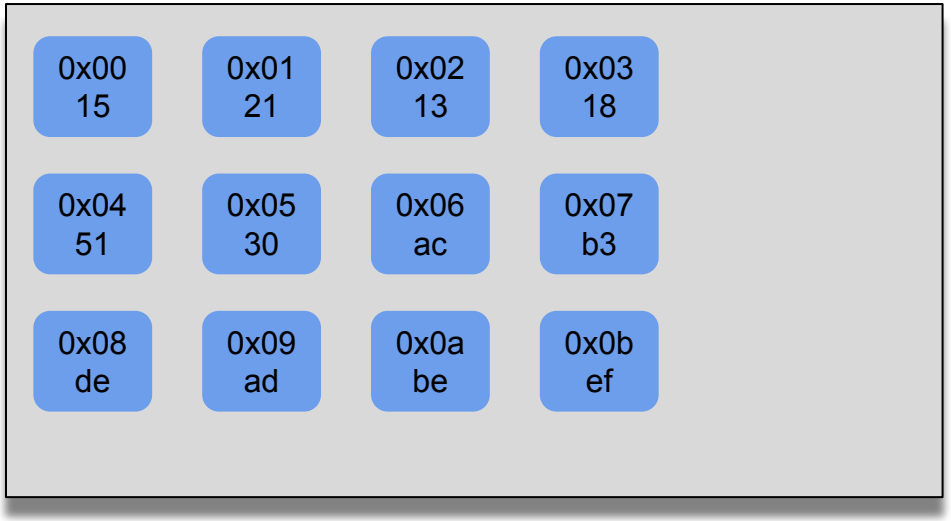
0x9 = 0b1001

=> Set Index 0
=> Have to evict!

Cache



Memory

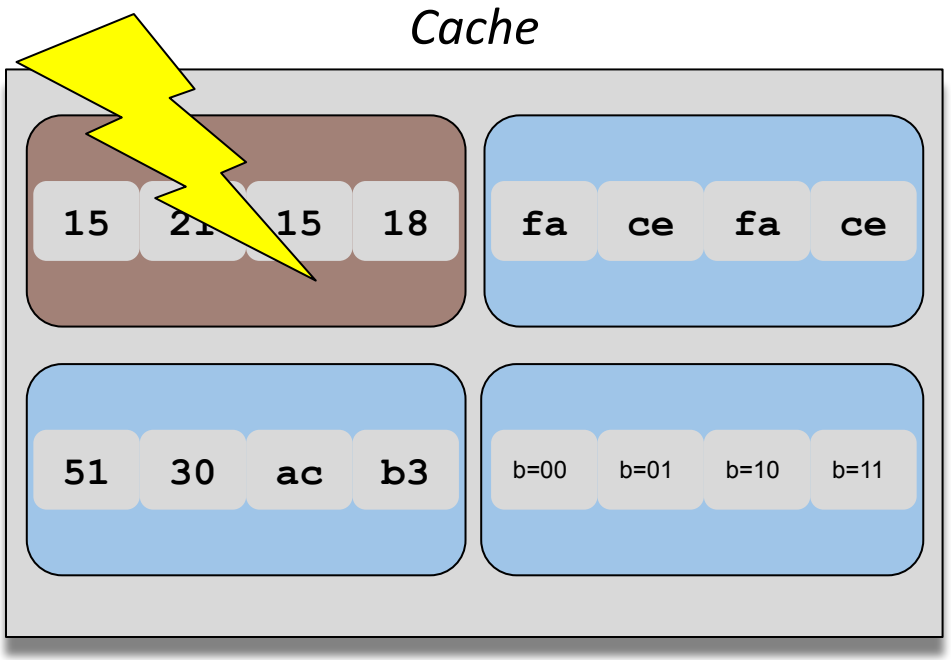


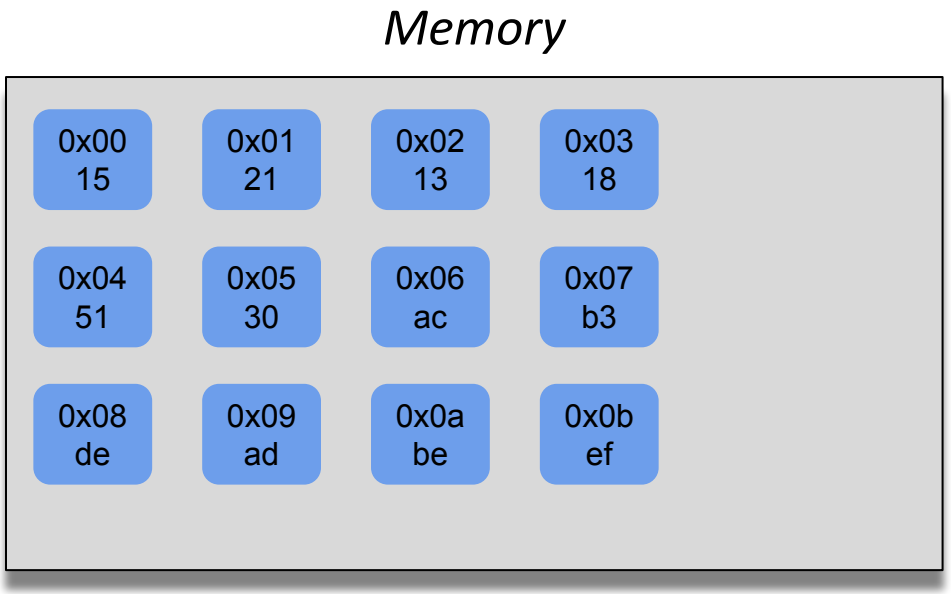
Example Trace

```
...
L 0x00,1    Hit!
L 0x10,1    Miss
L 0x09,1    Miss
...
```

Evict least recently used line

Dirty bit set => Dirty Eviction

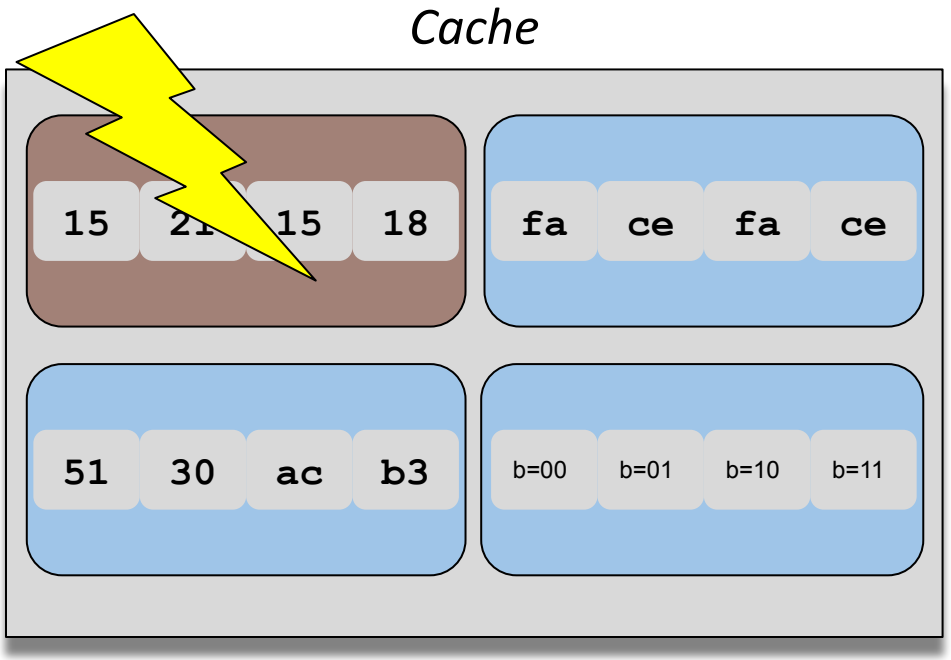




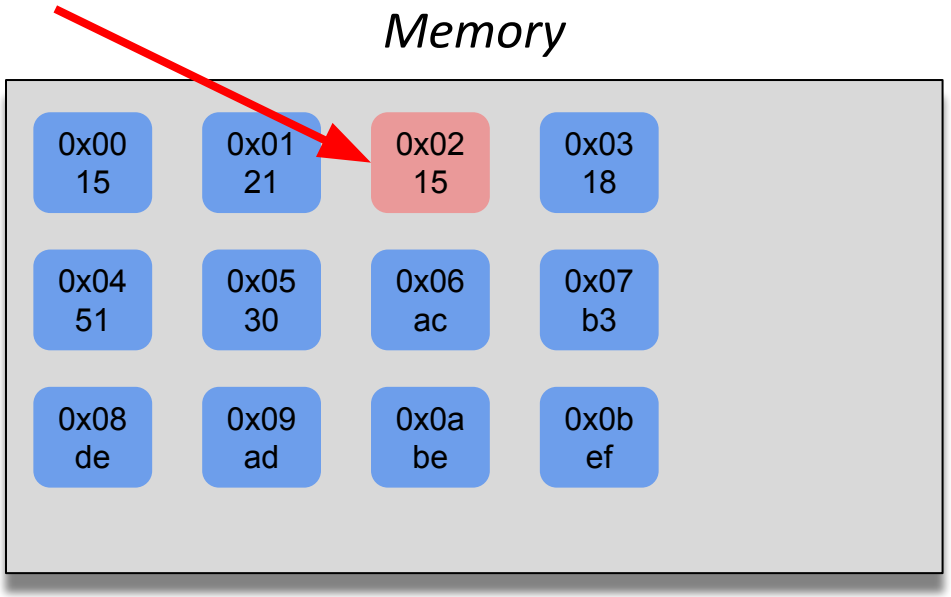
Example Trace

```
...
L 0x00,1
L 0x10,1
L 0x09,1
...
```

Hit!
Miss
Miss



Write-back policy!

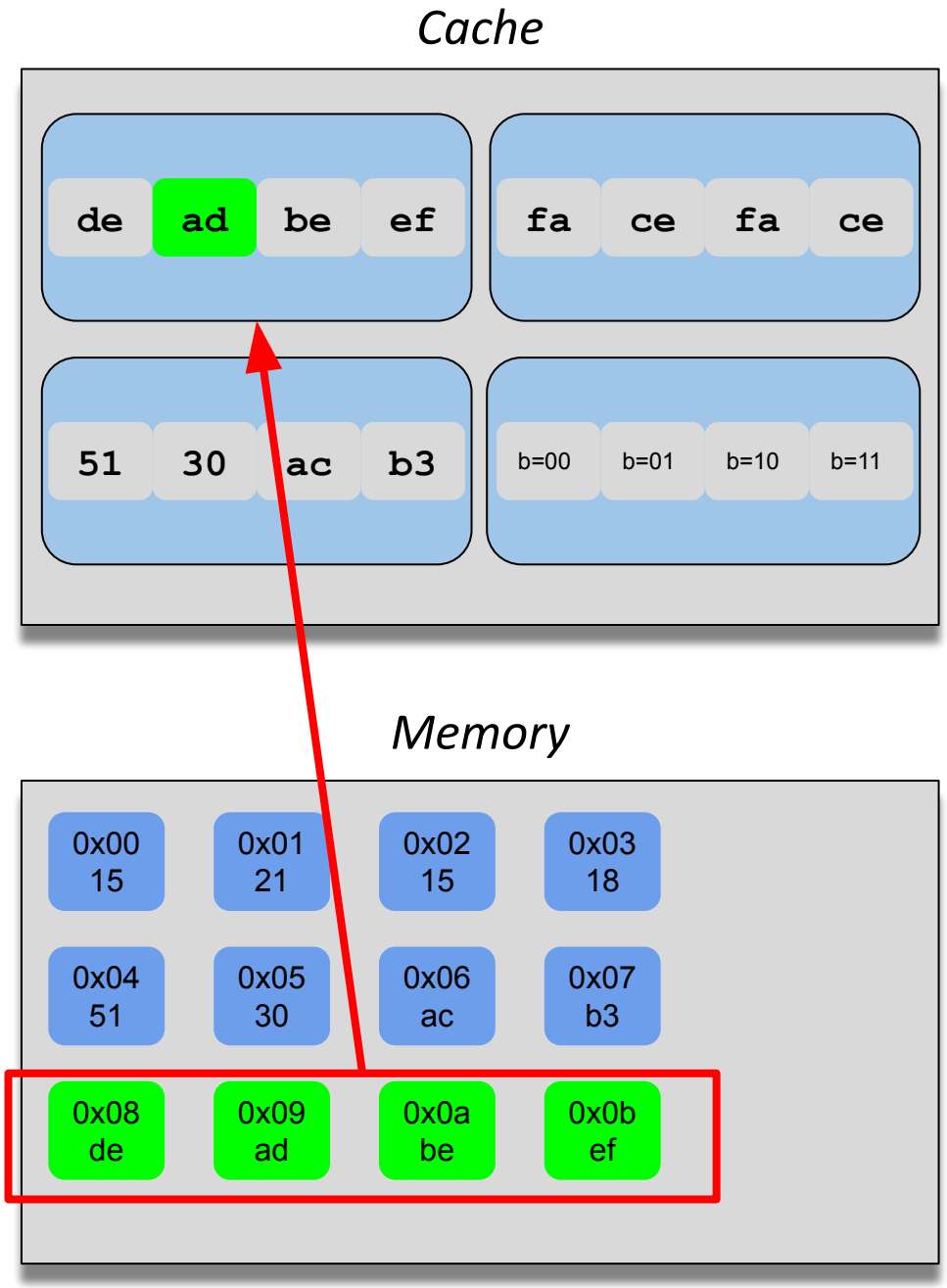


Example Trace

```
...
L 0x00,1
L 0x10,1
L 0x09,1
...
```

Hit!
Miss
Miss

Load new value into line

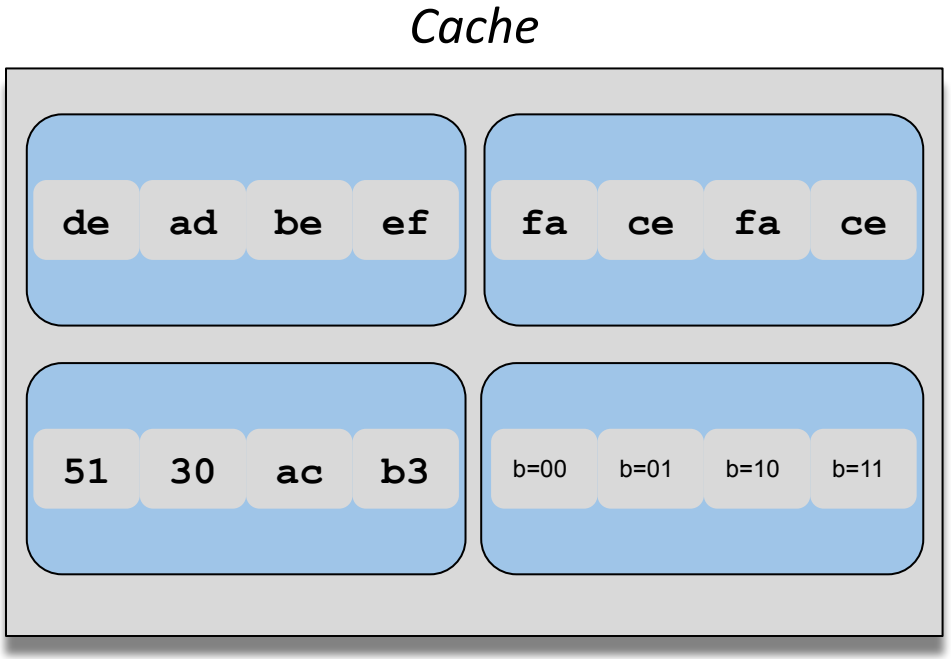


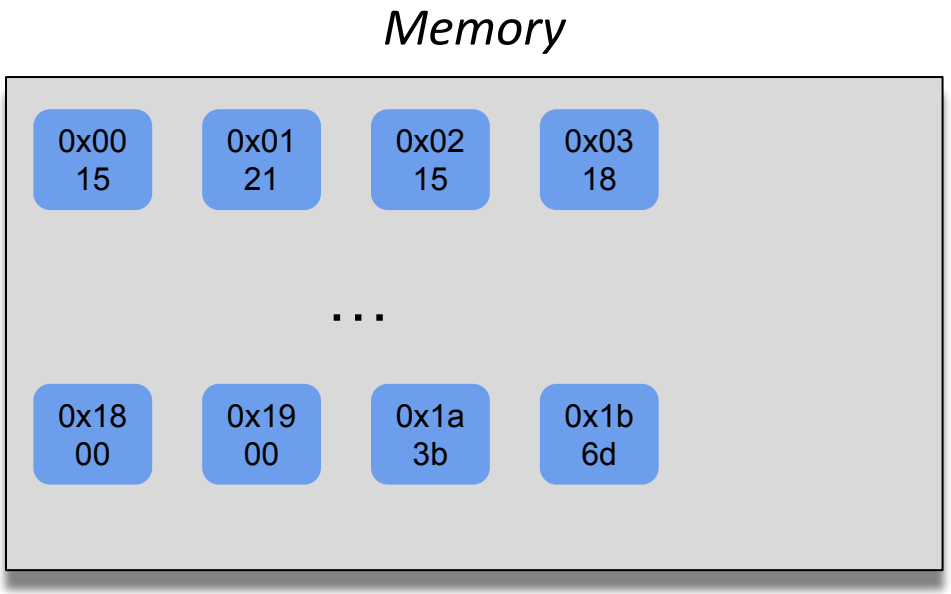
Example Trace

```
...
L 0x10,1
L 0x09,1
L 0x18,1
...
```

Miss
Miss
???

Will this instruction result in a hit or a miss?





Example Trace

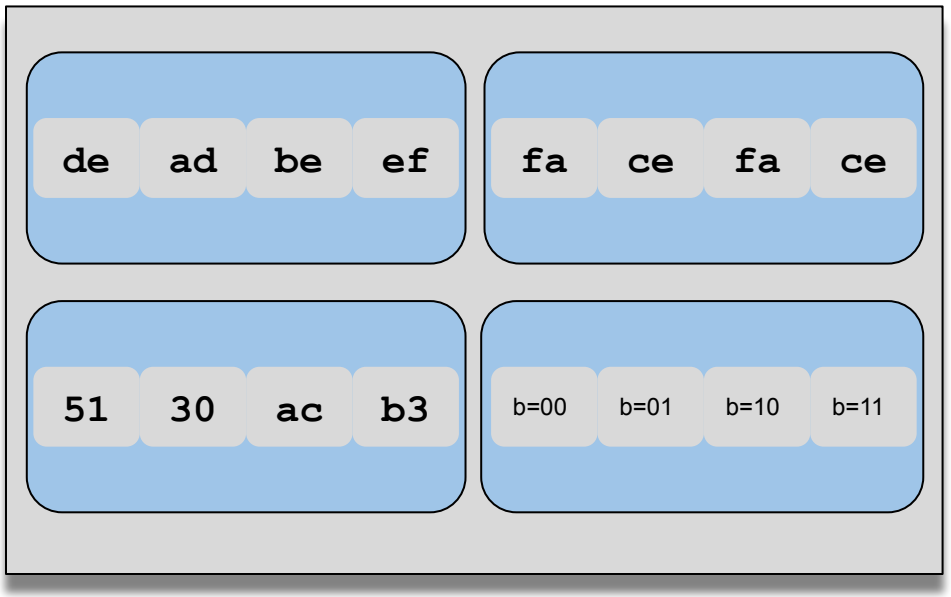
```
...
L 0x10,1
L 0x09,1
L 0x18,1
...
```

Miss
Miss
Miss

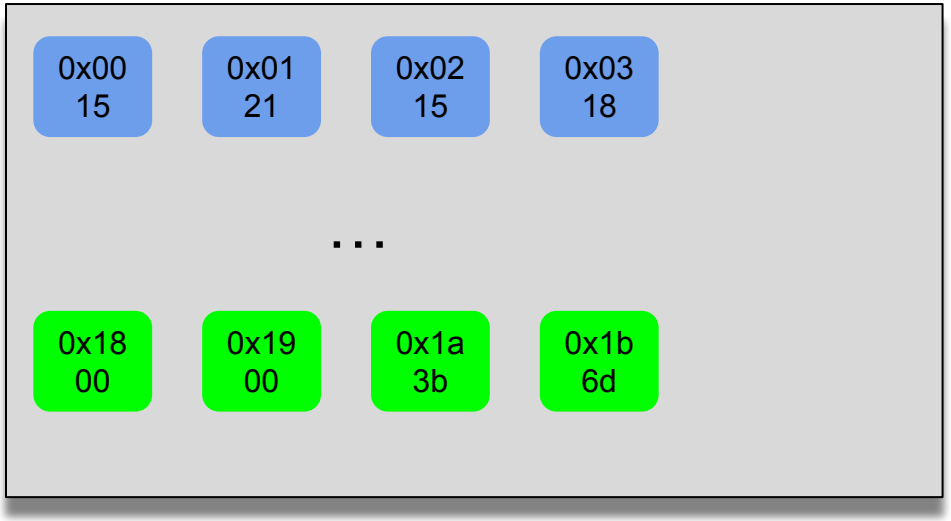
What type of miss is this?

Which line will get evicted?

Cache



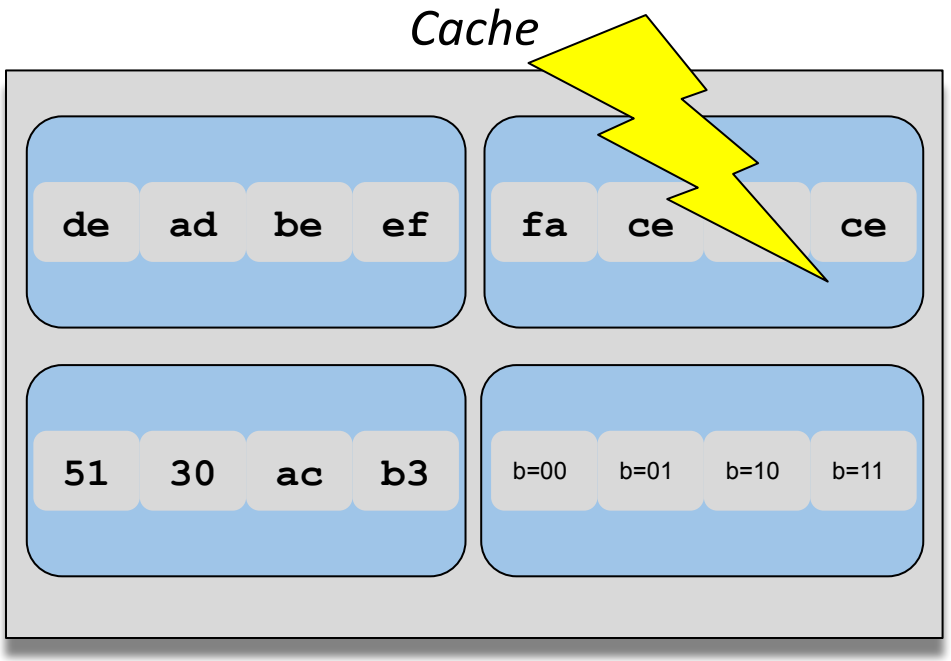
Memory



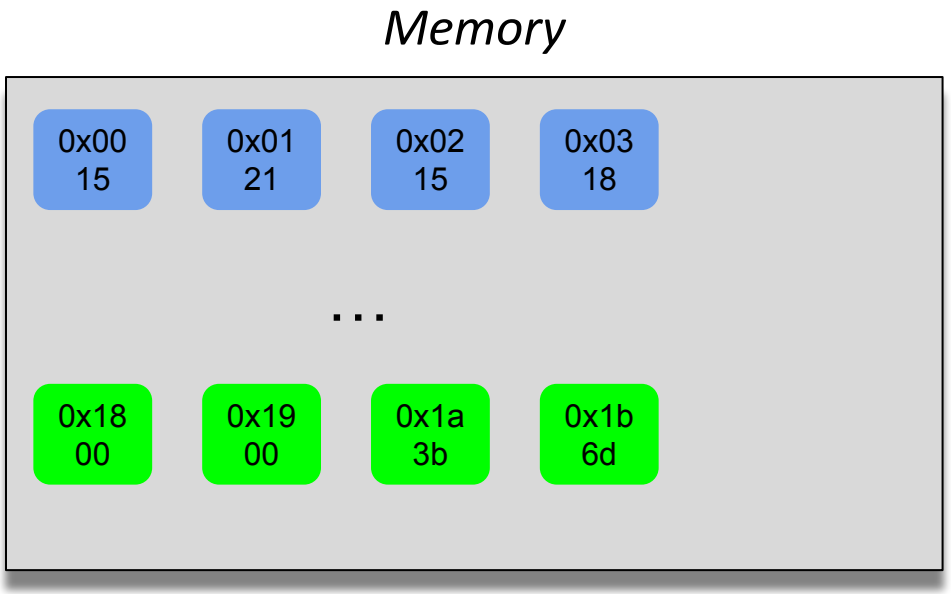
Example Trace

```
...
L 0x10,1
L 0x09,1
L 0x18,1
...
```

Miss
Miss
Miss



Evict least recently used line

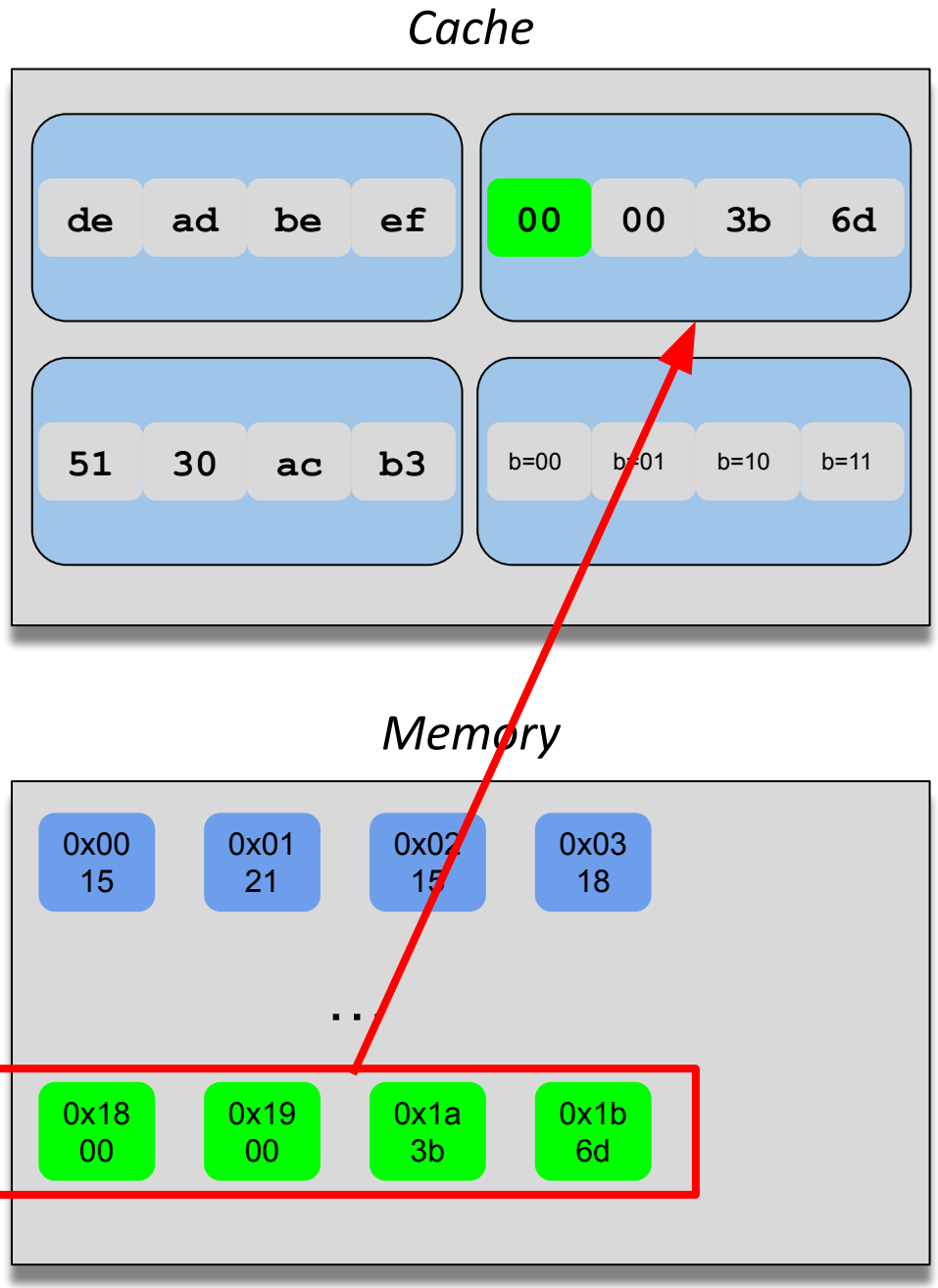


Example Trace

```
...
L 0x10,1
L 0x09,1
L 0x18,1
...
```

Miss
Miss
Miss

Load new value into line

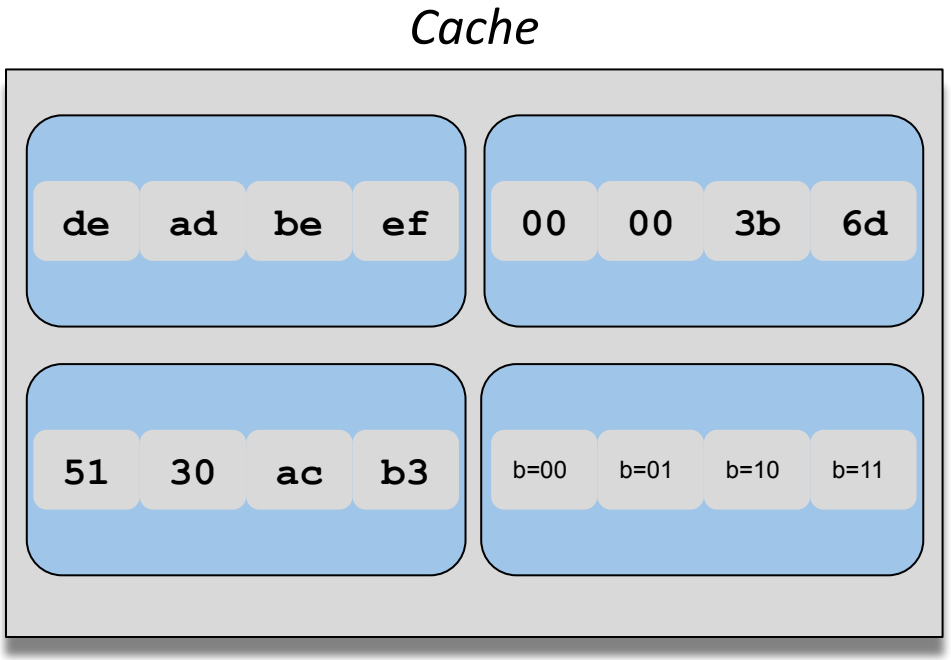


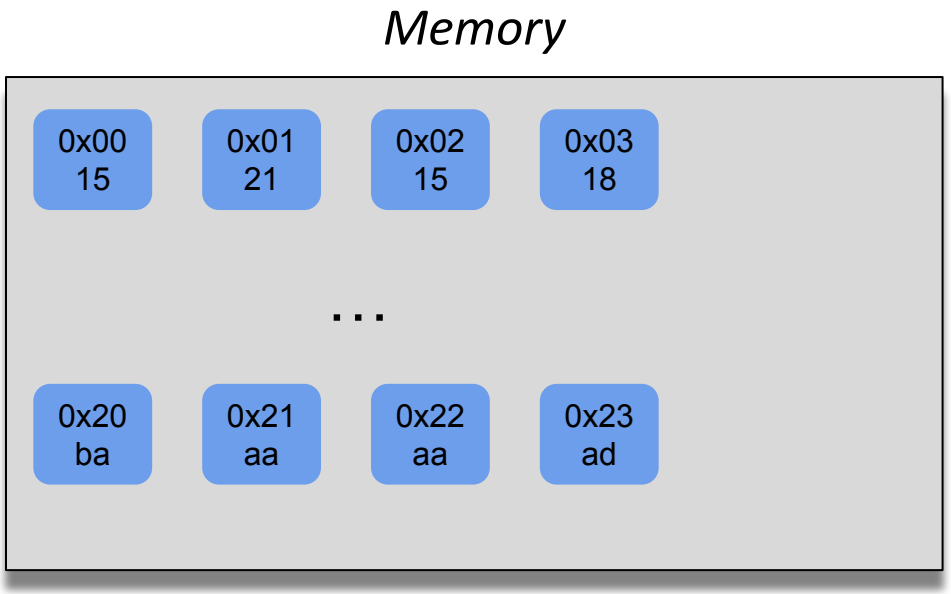
Example Trace

```
...
L 0x09,1
L 0x18,1
L 0x20,1
...
```

```
Miss
Miss
???
```

Will this instruction result in a hit or a miss?





Example Trace

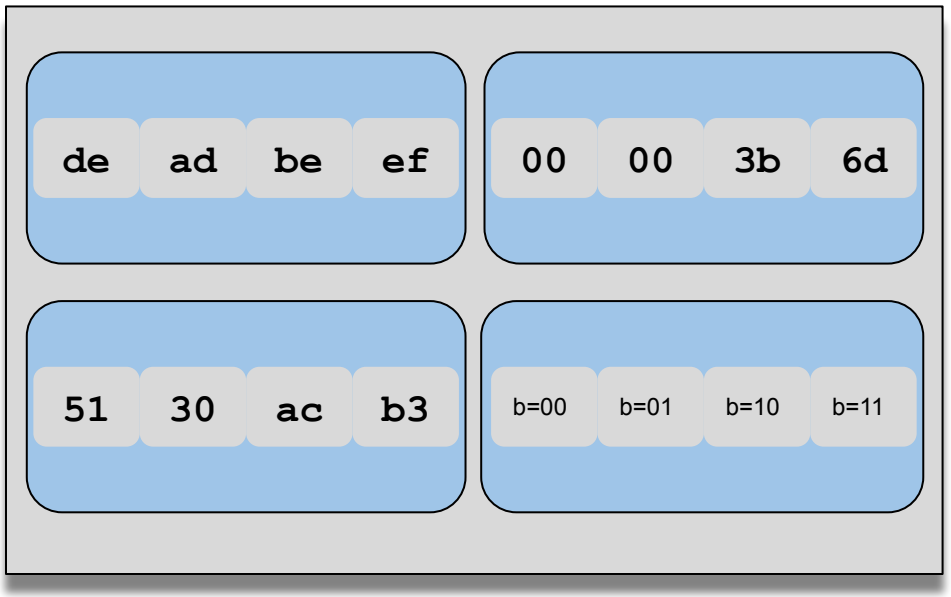
```
...
L 0x09,1
L 0x18,1
L 0x20,1
...
```

Miss
Miss
Miss

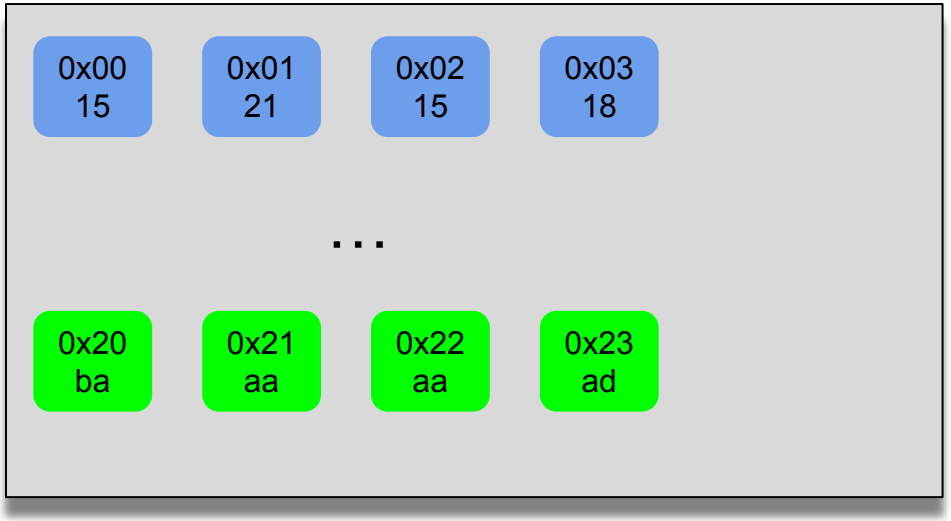
What type of miss is this?

Which line gets evicted?

Cache



Memory

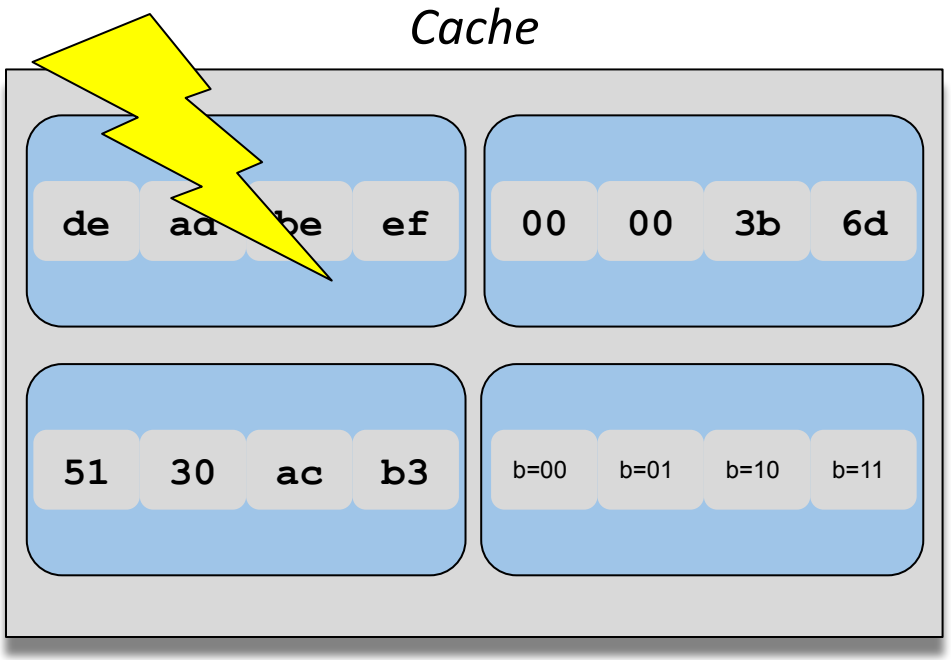


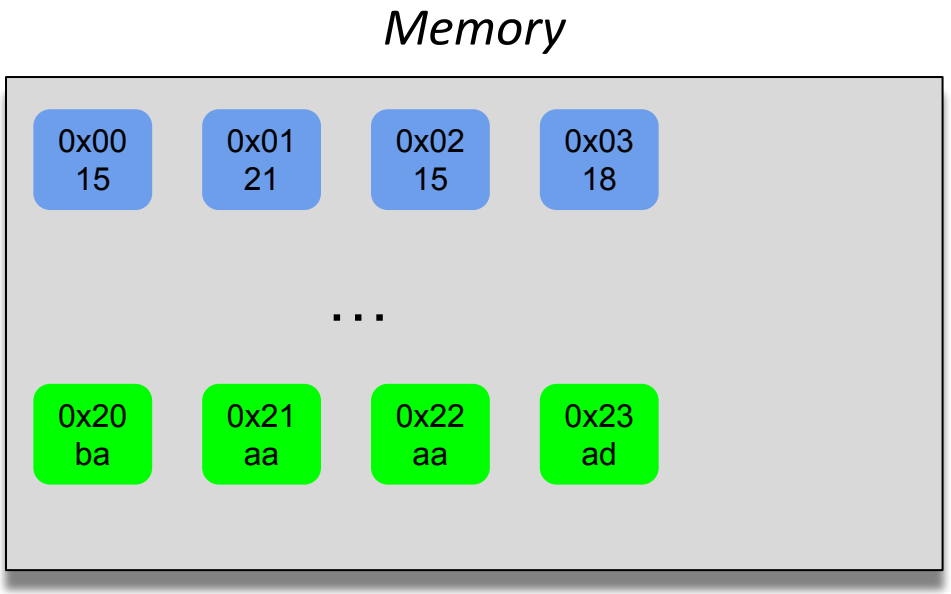
Example Trace

```
...
L 0x09,1
L 0x18,1
L 0x20,1
...
```

Miss
Miss
Miss

Evict least recently used line



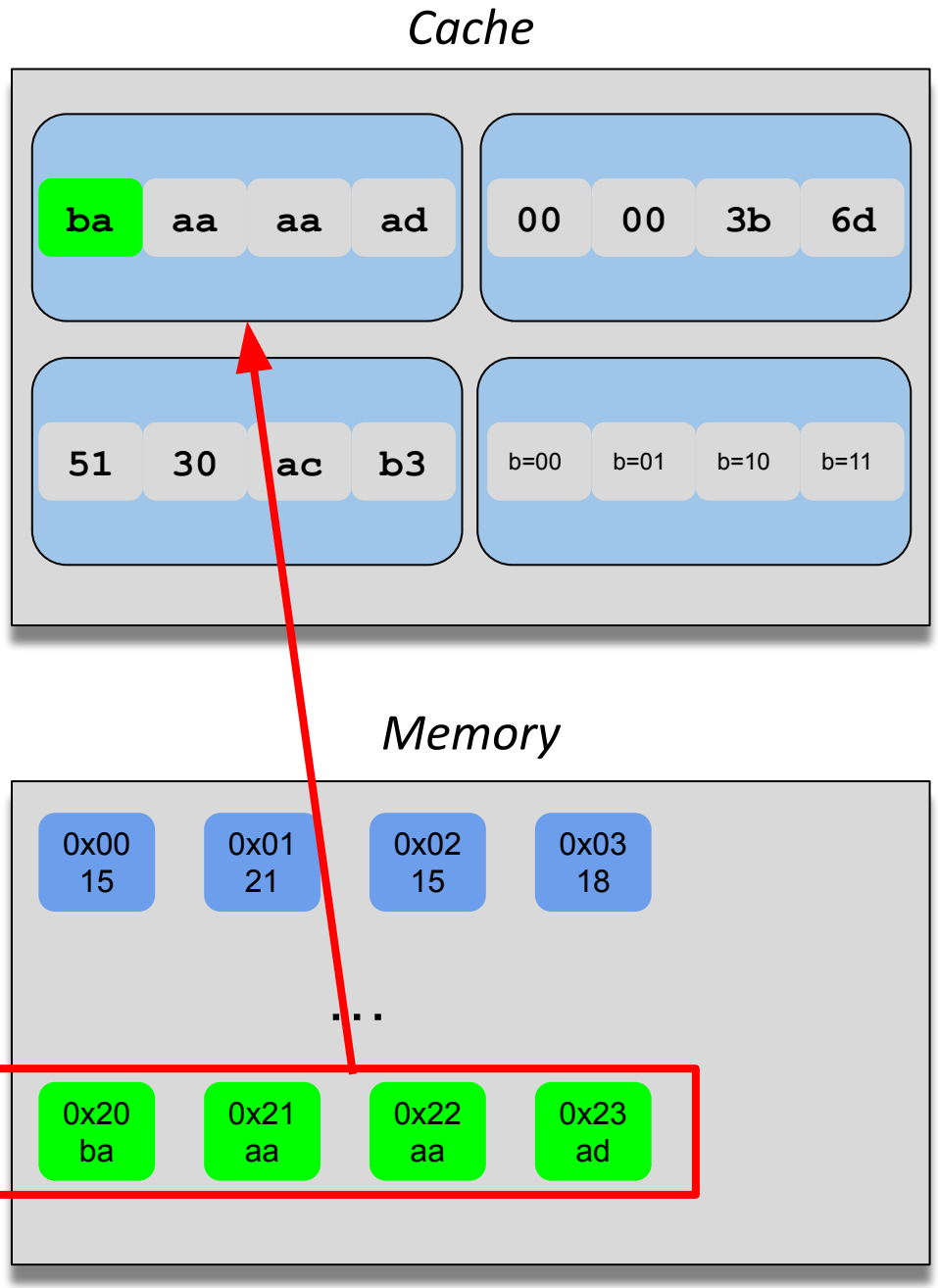


Example Trace

```
...
L 0x09,1
L 0x18,1
L 0x20,1
...
```

Miss
Miss
Miss

Load new value into line

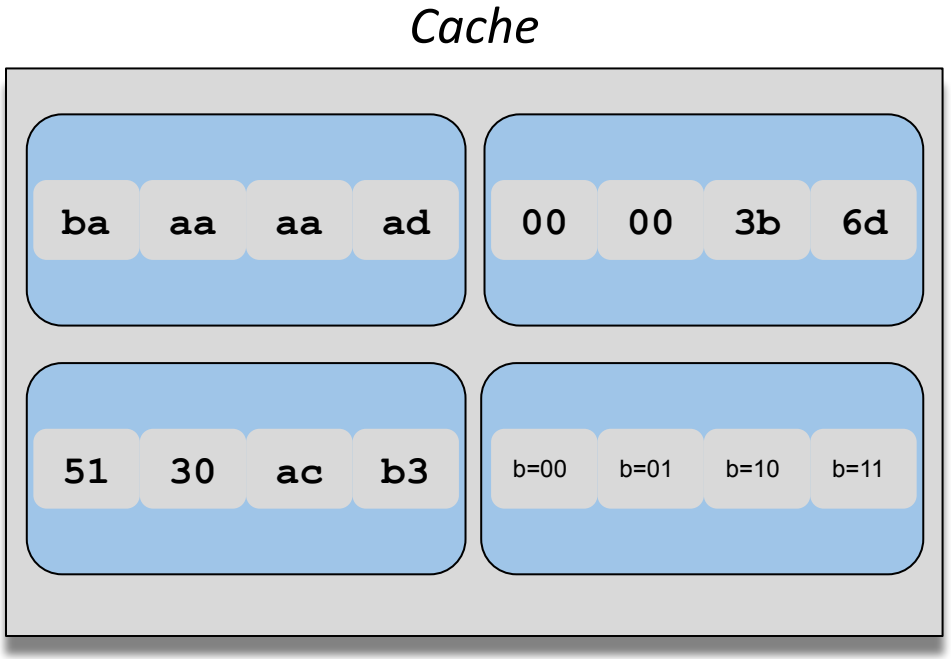


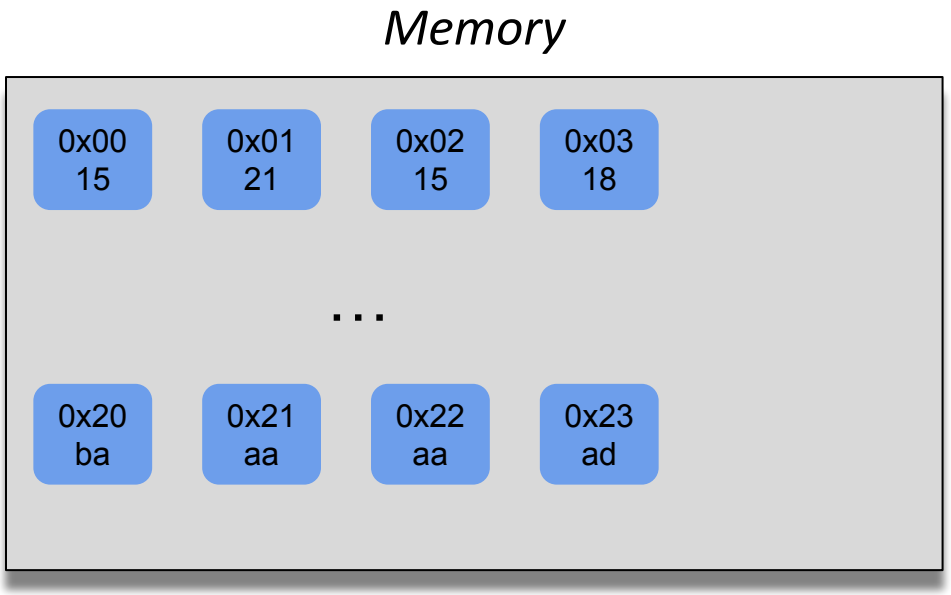
Example Trace

```
...
L 0x18,1
L 0x20,1
L 0x00,1
...
```

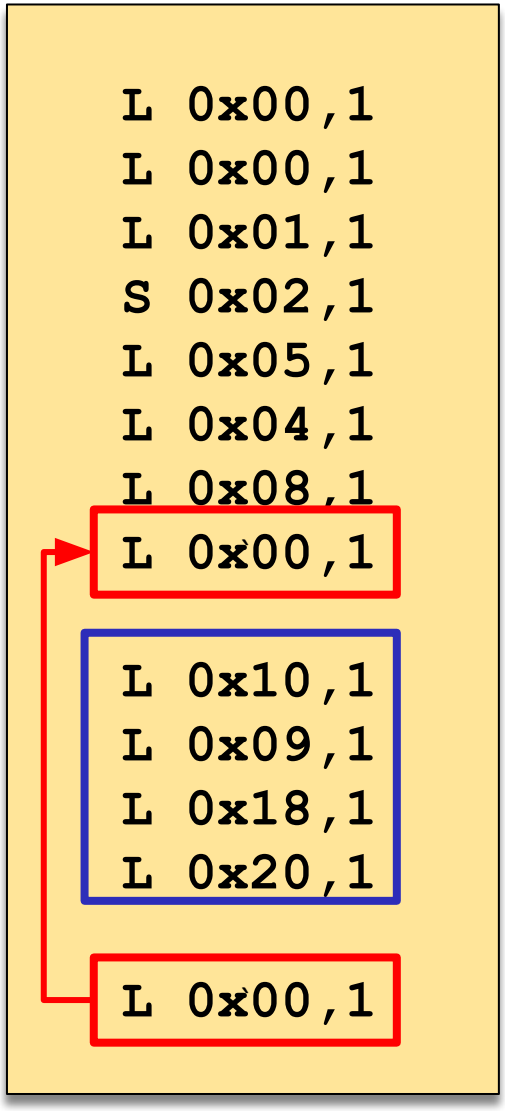
Miss
Miss
???

Will this instruction result in a hit or a miss?





Cache Concepts: Conflict/Capacity Misses

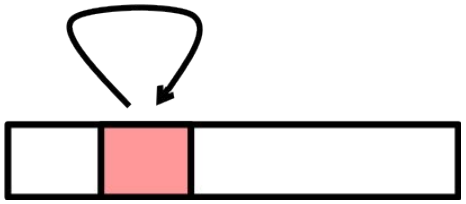


- In this case:
  - Number of unique blocks *in-between* current reference and most recent reference: 4
  - Our cache has 4 total lines
  - So: ***Capacity Miss***
- Note: the cache is **not** full!

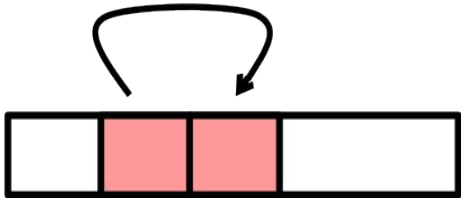
Writing Cache-Friendly Code

Recall: Temporal and Spatial Locality

- ***Temporal Locality:***
  - *Recently referenced* items are likely to be referenced again soon!

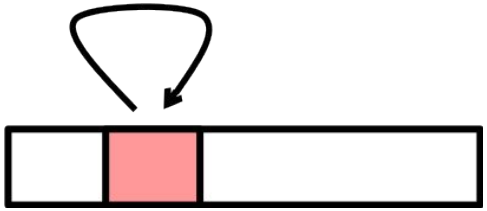


- ***Spatial Locality:***
  - Items with *nearby addresses* tend to be referenced close together in time.



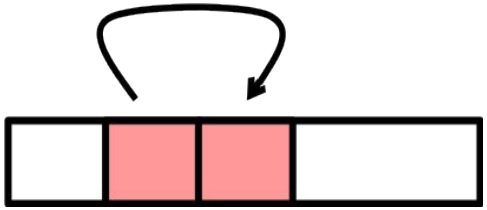
Optimizing for Locality

Temporal



- *Recently referenced* items are likely to be referenced again soon!
- To optimize: try to use data objects as often as possible once they're read from memory.
- Lecture Example: *Blocking*.

Spatial



- Items with *nearby addresses* tend to be referenced close together in time.
- To optimize: read objects sequentially, and with smaller stride.
- Lecture Example: *Rearranging loops*.

Review: Cache Configurations

Cache Configurations

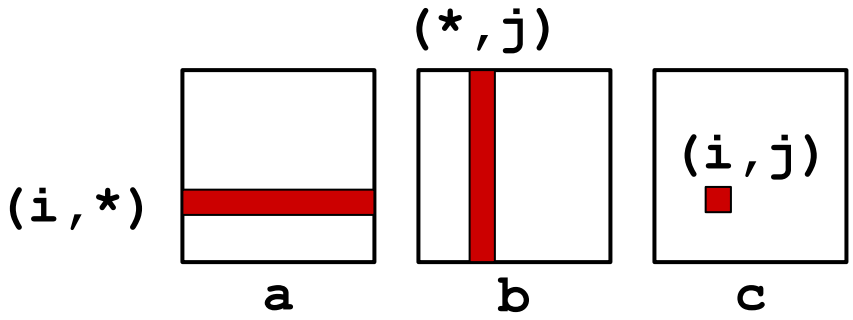
- **Direct Mapped**: Cache with **`E=1`**
  - one line per set
- **Fully Associative**: Cache with **`s=0`**
  - all lines in one set
- **k-way Associative**: Cache with **`E=k`**
  - k lines per set

Blocking

Example: Matrix Multiplication

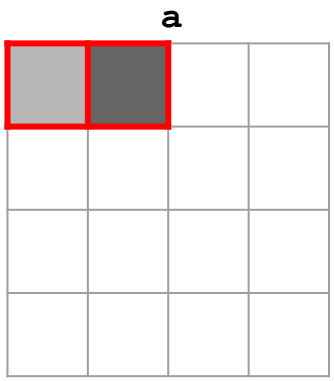
```
/* Multiply 4x4 matrices */
void mm(int a[4][4], int b[4][4], int c[4][4]) {
    int i, j, k;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                c[i][j] += a[i][k] * b[k][j];
```

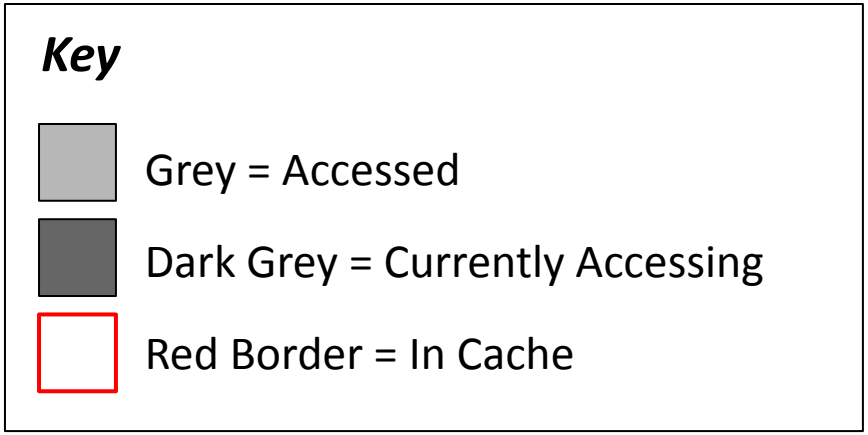
- "Standard" way of doing matrix multiplication (**ijk**):
  - **c[i][j]** is given by taking "dot product" of **i**-th row of **a** with **j**-th column of **b**.



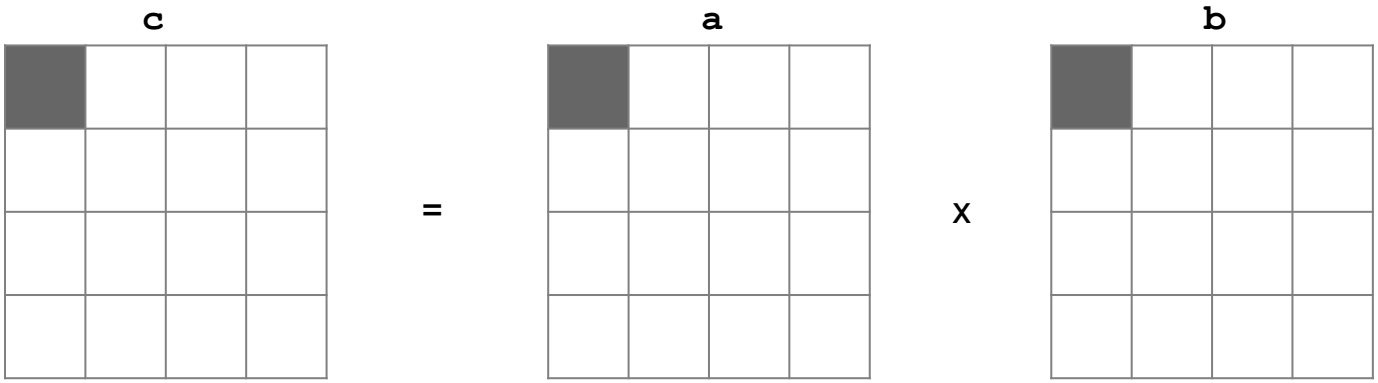
Example: Matrix Multiplication

- Assume a tiny cache with 4 lines of 8 bytes (2 **`int`**s each)
  - **`S`** = 1, **`E`** = 4, **`B`** = 8
- We'll use the following key:





- Let's see what happens if we don't use blocking…

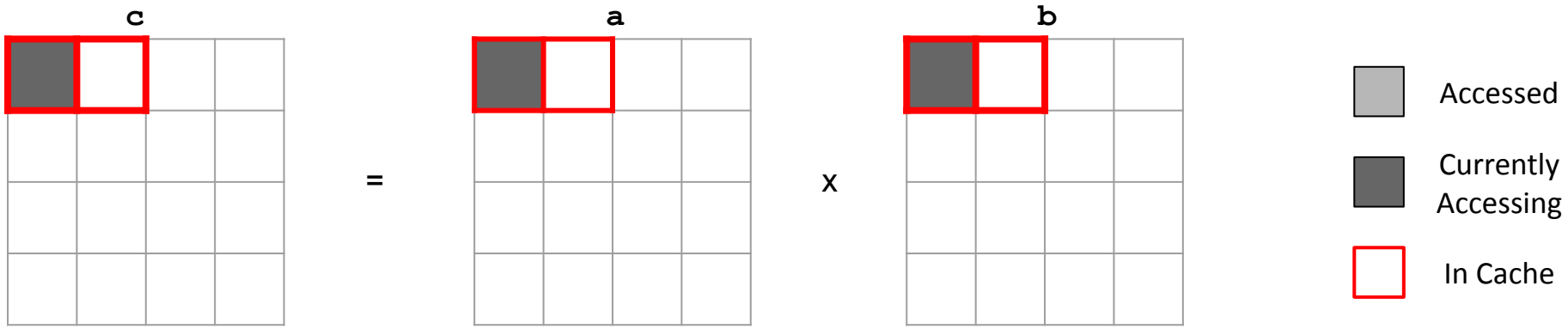


Accessed

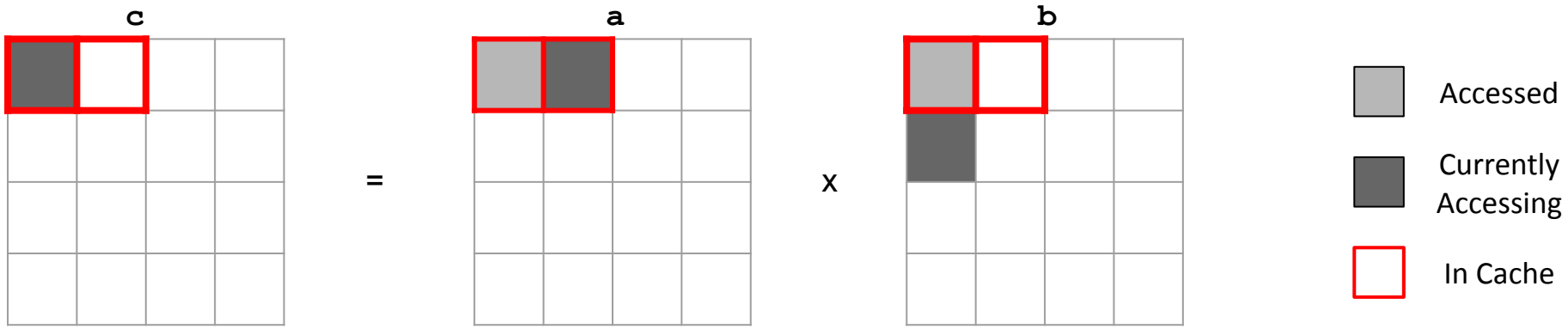
Currently Accessing

In Cache

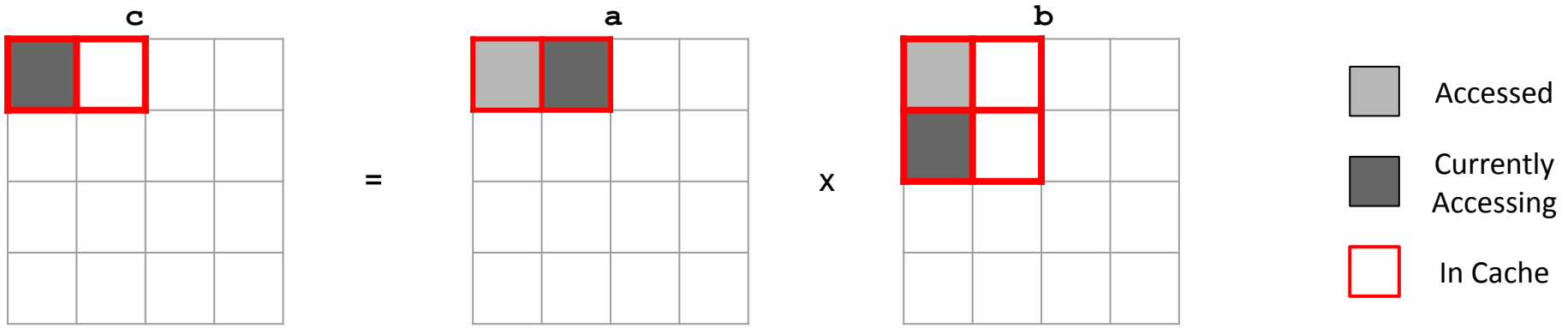
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	???



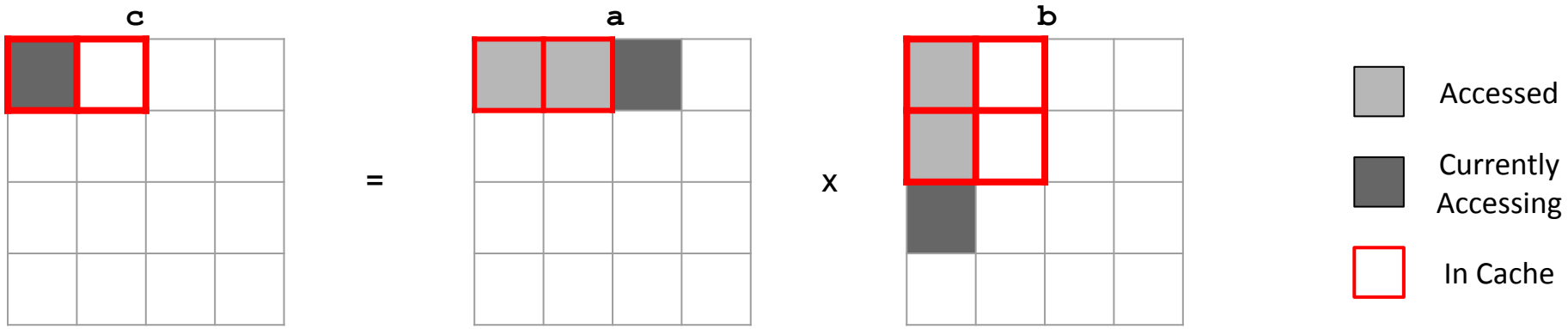
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)



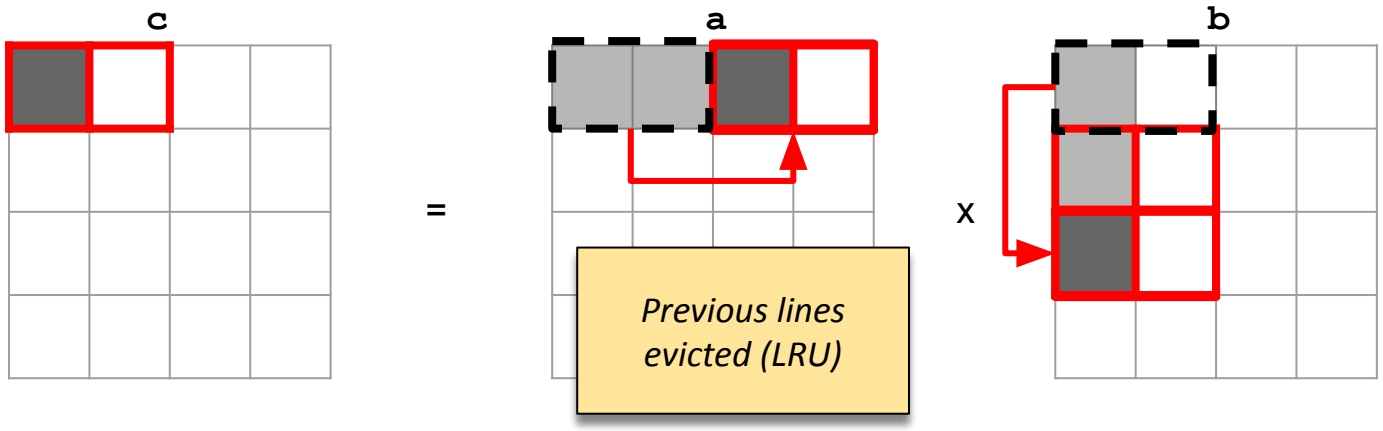
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	???



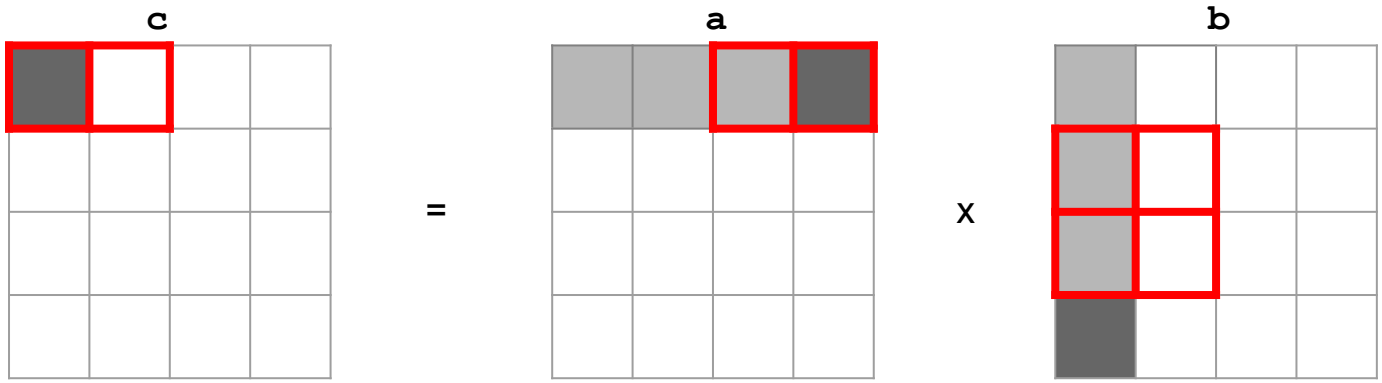
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)



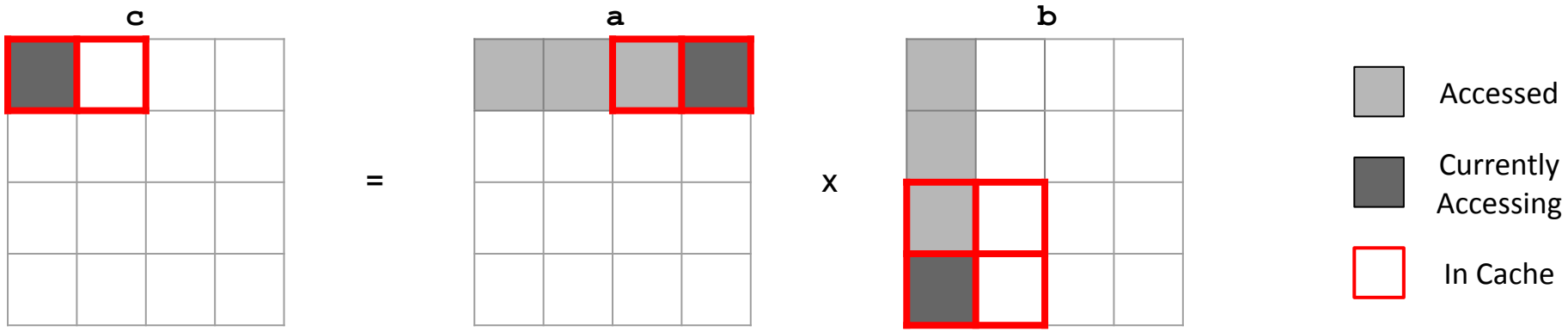
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	???



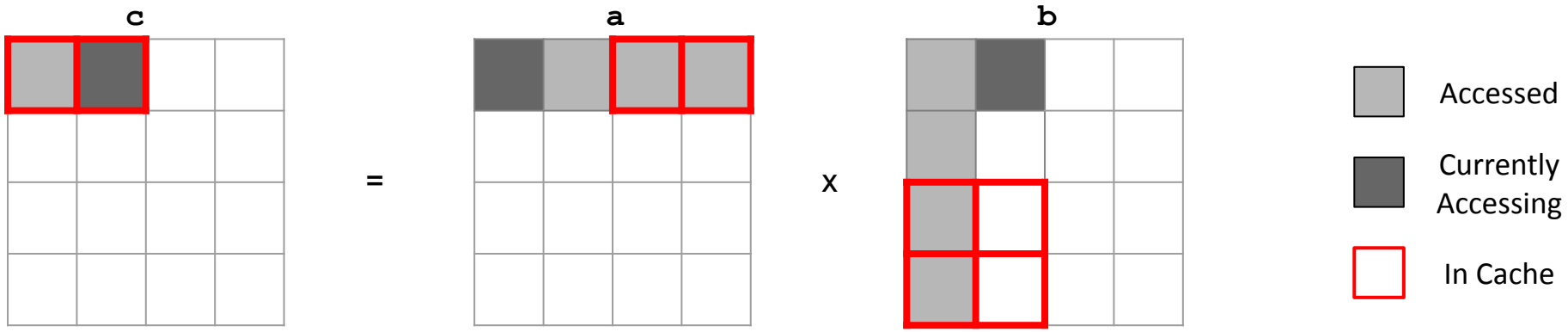
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)



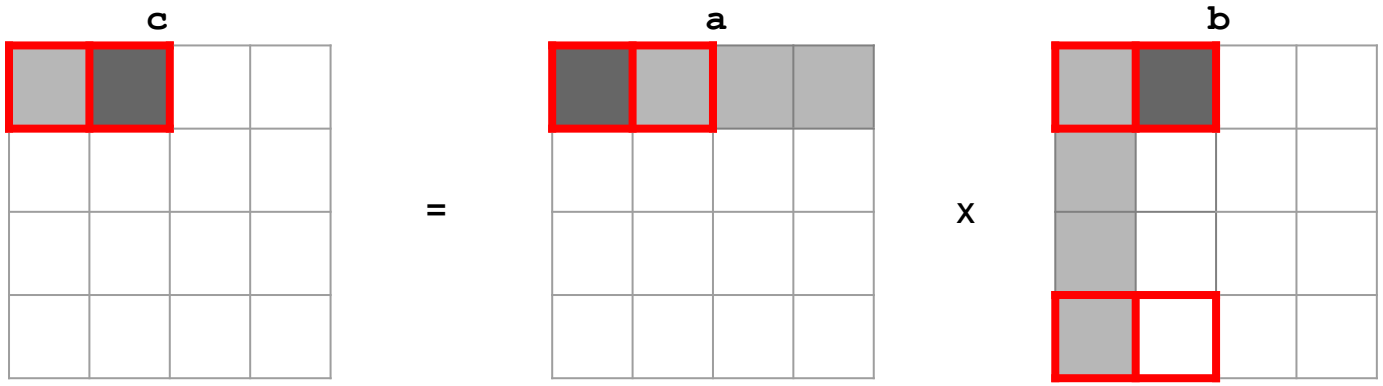
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
3	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	???



Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
3	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)

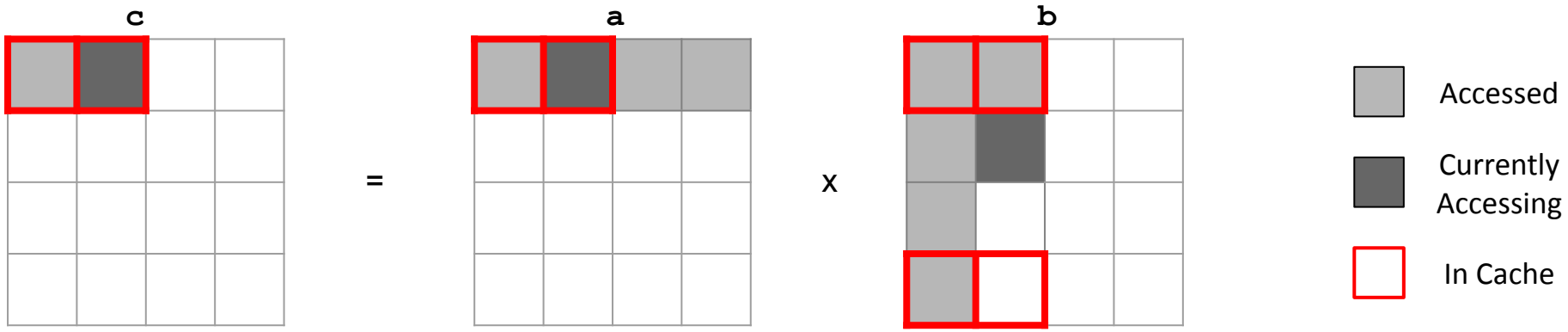


Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
3	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
4	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	???

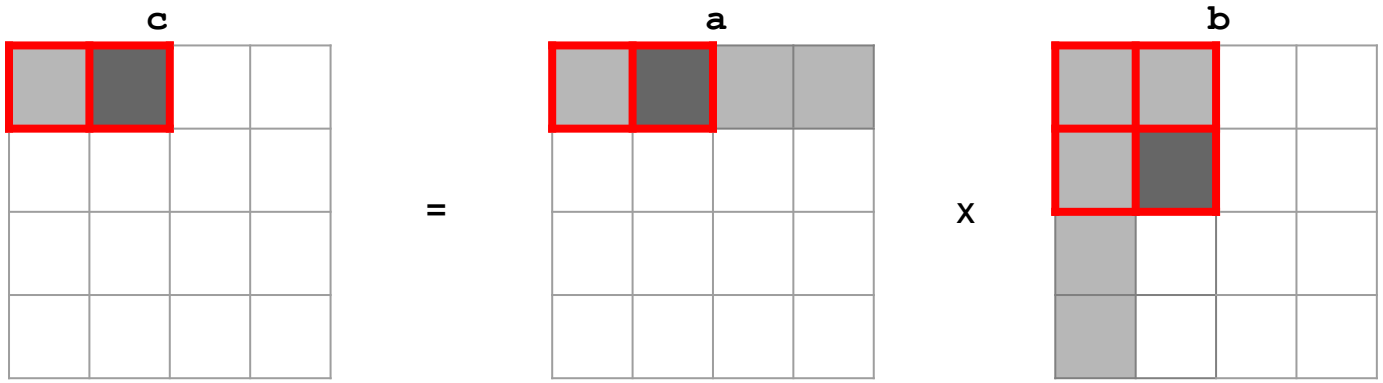


Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
3	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
4	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(m, m)

Have these blocks been in the cache before?

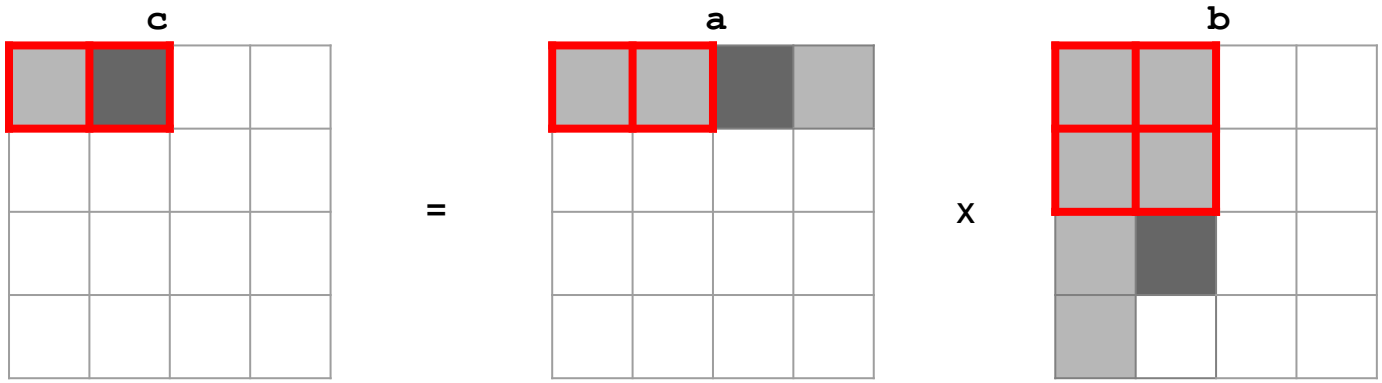


Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
3	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
4	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(m, m)
5	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	???

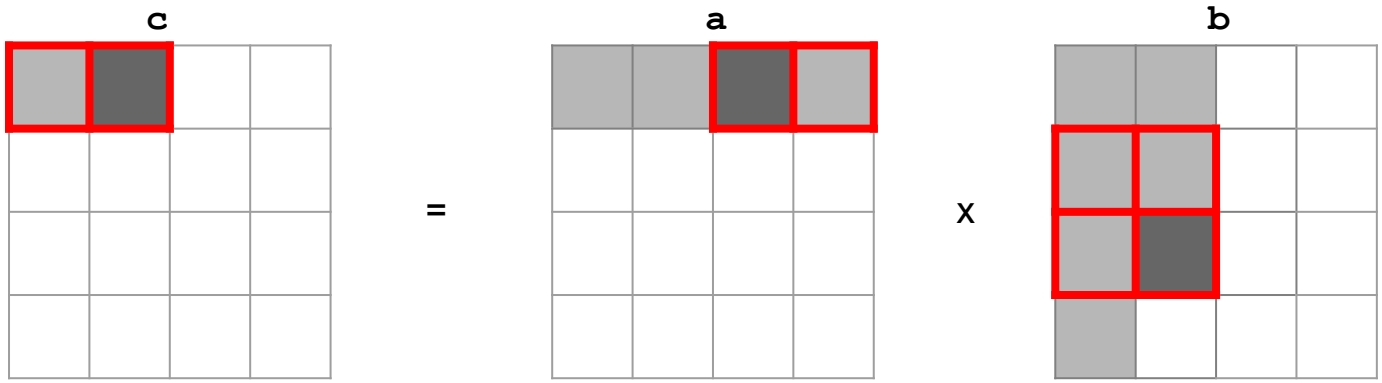


Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
3	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
4	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(m, m)
5	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, m)

Has this block been in cache before?

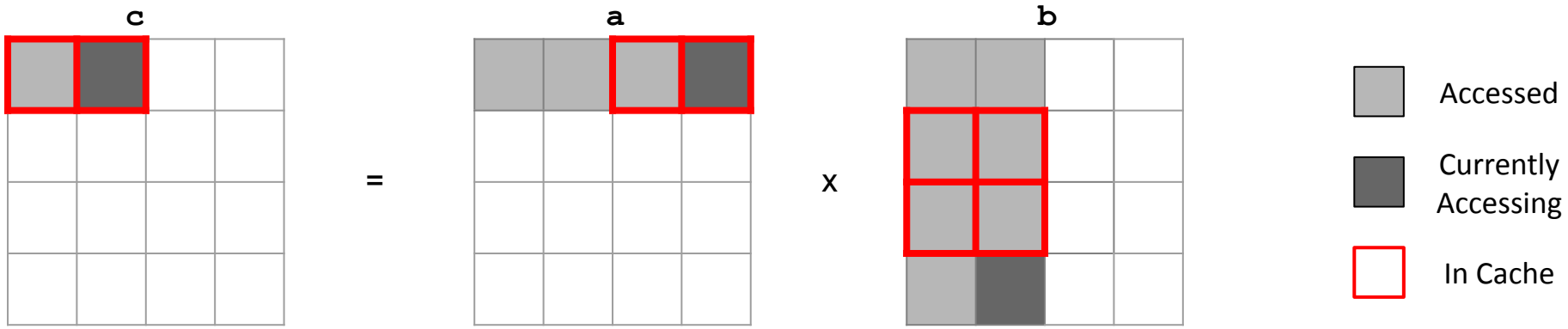


Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
3	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
4	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(m, m)
5	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, m)
6	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	???

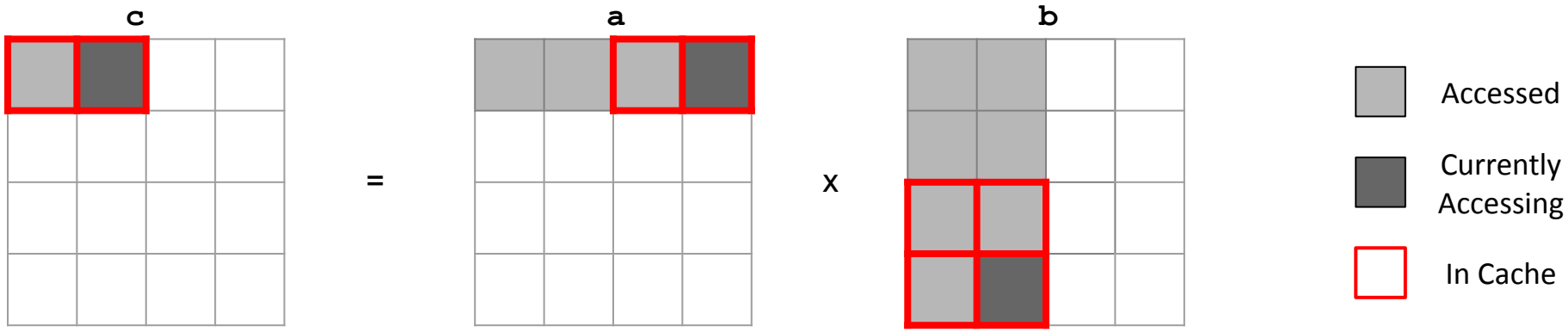


Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
3	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
4	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(m, m)
5	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, m)
6	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(m, m)

Have these blocks been in the cache before?



Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
3	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
4	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(m, m)
5	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, m)
6	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(m, m)
7	0	1	3	`c[0][1] += a[0][3] * b[3][1]`	???



Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
3	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
4	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(m, m)
5	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, m)
6	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(m, m)
7	0	1	3	`c[0][1] += a[0][3] * b[3][1]`	(h, m)

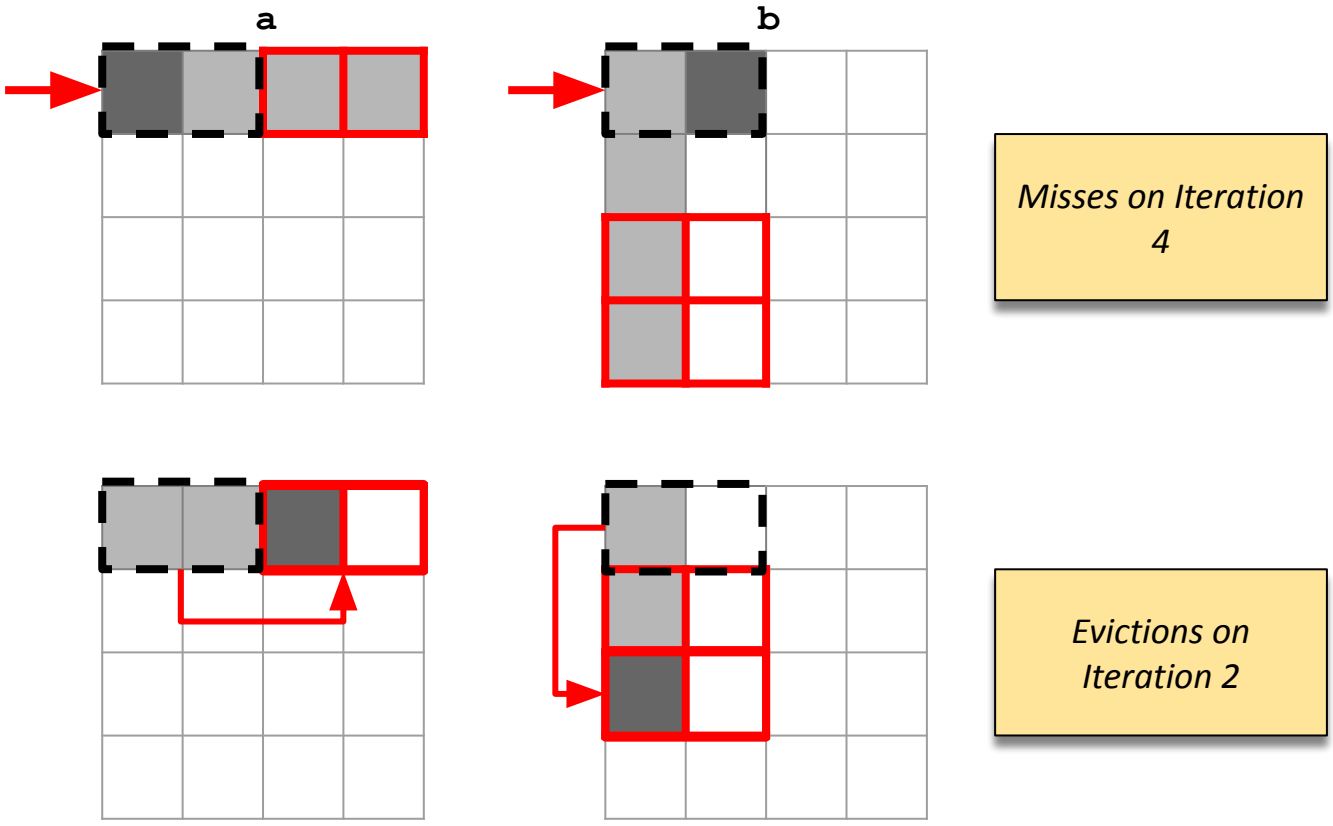
No Blocking: Analyzing Miss Rate

Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] + b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] + b[1][0]`	(h, m)
2	0	0	2	`c[0][0] += a[0][2] + b[2][0]`	(m, m)
3	0	0	3	`c[0][0] += a[0][3] + b[3][0]`	(h, m)
4	0	1	0	`c[0][1] += a[0][0] + b[0][1]`	(m, m)
5	0	1	1	`c[0][1] += a[0][1] + b[1][1]`	(h, m)
6	0	1	2	`c[0][1] += a[0][2] + b[2][1]`	(m, m)
7	0	1	3	`c[0][1] += a[0][3] + b[3][1]`	(h, m)

- What is the miss rate of **`a`**?
  - 4/8 = ***50%***
- What is the miss rate of **`b`**?
  - 8/8 = ***100%***

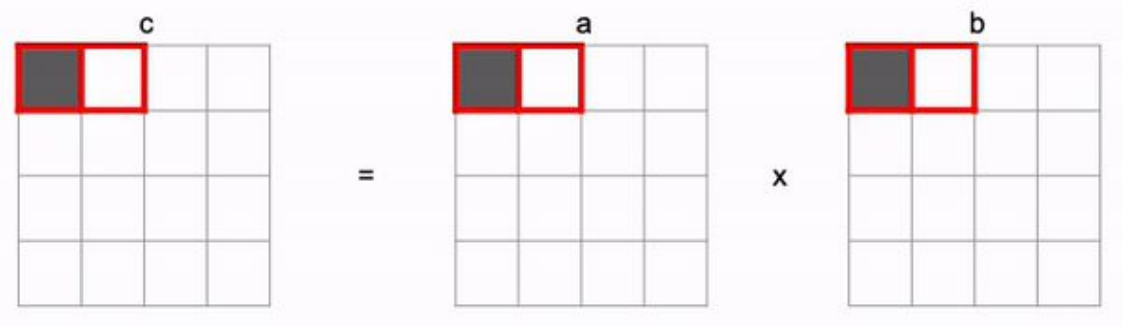
No Blocking: What went Wrong?

- Bad temporal locality!
- Blocks are used multiple times, but are never in cache when we need them.

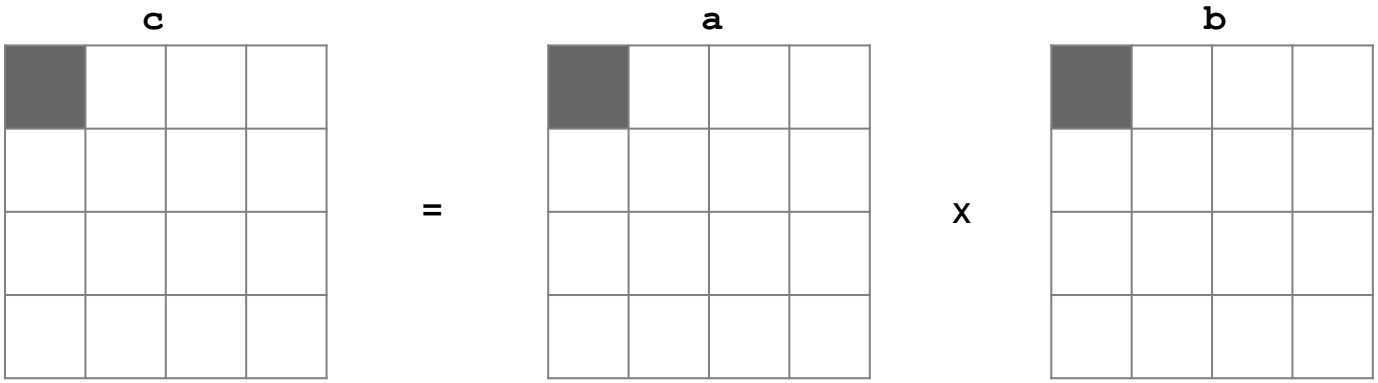


Example: Matrix Multiplication (with Blocking)

```
/* multiply 4x4 matrices using blocks of size 2 */
void mm_blocking(int a[4][4], int b[4][4], int c[4][4]) {
    int i, j, k;
    int i_c, j_c, k_c;
    int B = 2;
    // control loops
    for (i_c = 0; i_c < 4; i_c += B)
        for (j_c = 0; j_c < 4; j_c += B)
            for (k_c = 0; k_c < 4; k_c += B)
                // block multiplications
                for (i = i_c; i < i_c + B; i++)
                    for (j = j_c; j < j_c + B; j++)
                        for (k = k_c; k < k_c + B; k++)
                            c[i][j] += a[i][k] * b[k][j];
```



- Let's see what happens if we use blocking!

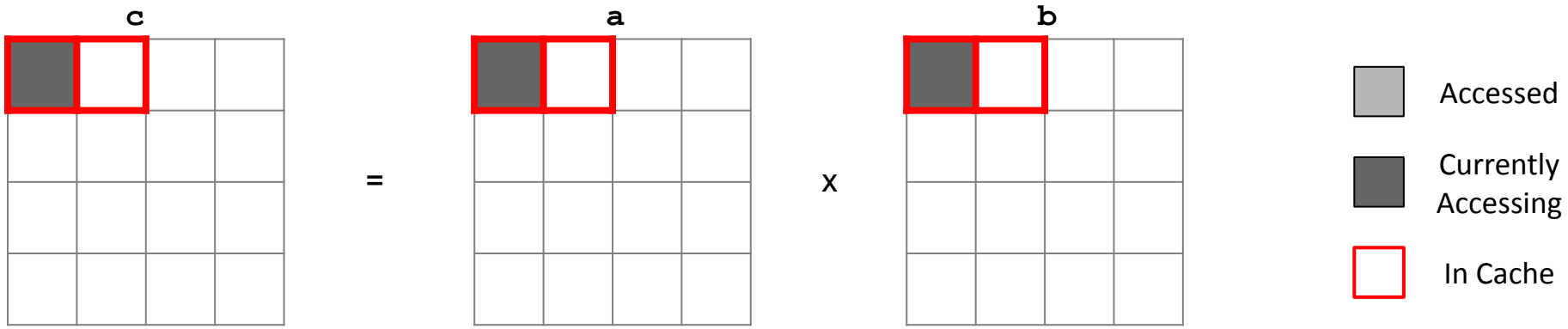


Accessed

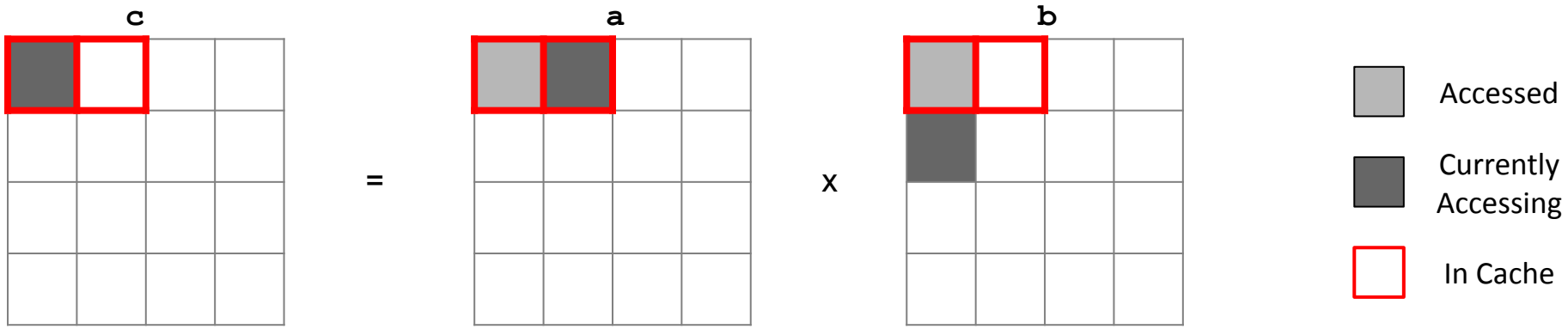
Currently Accessing

In Cache

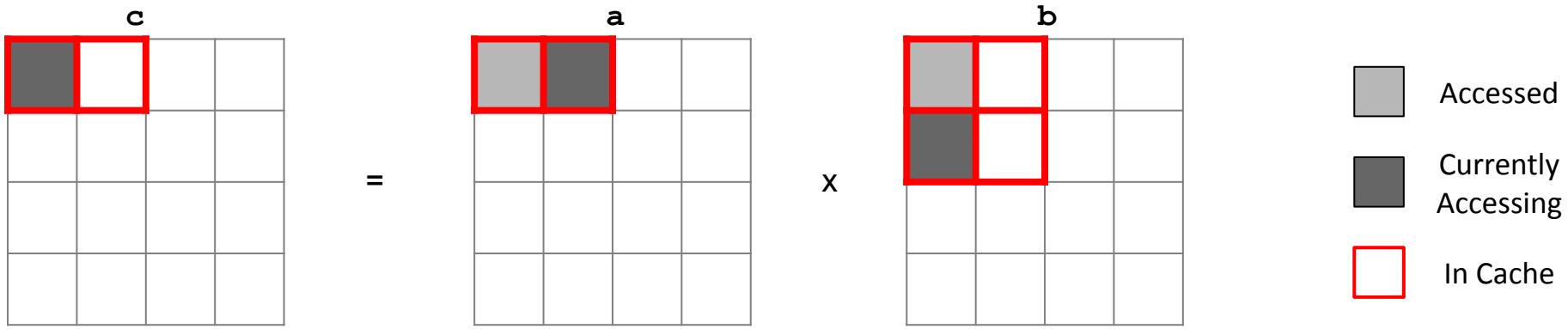
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	???



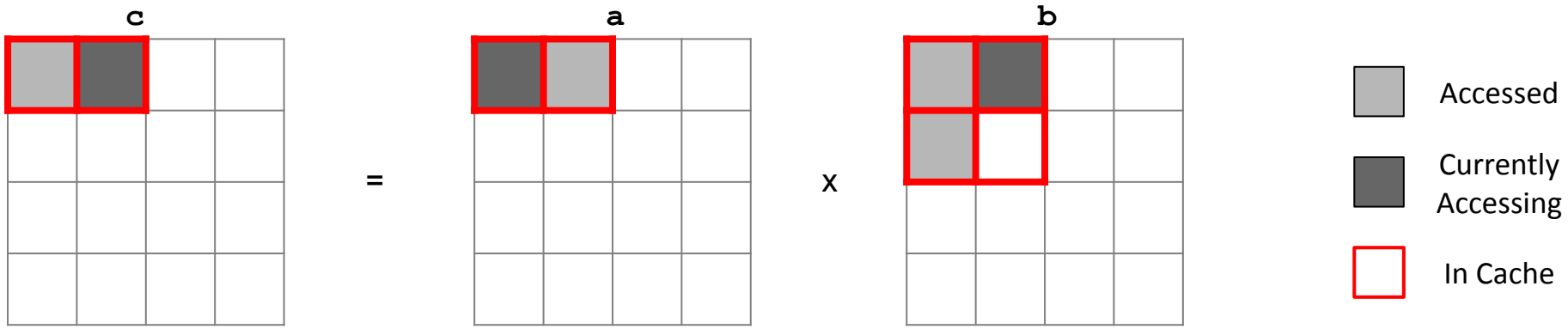
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)



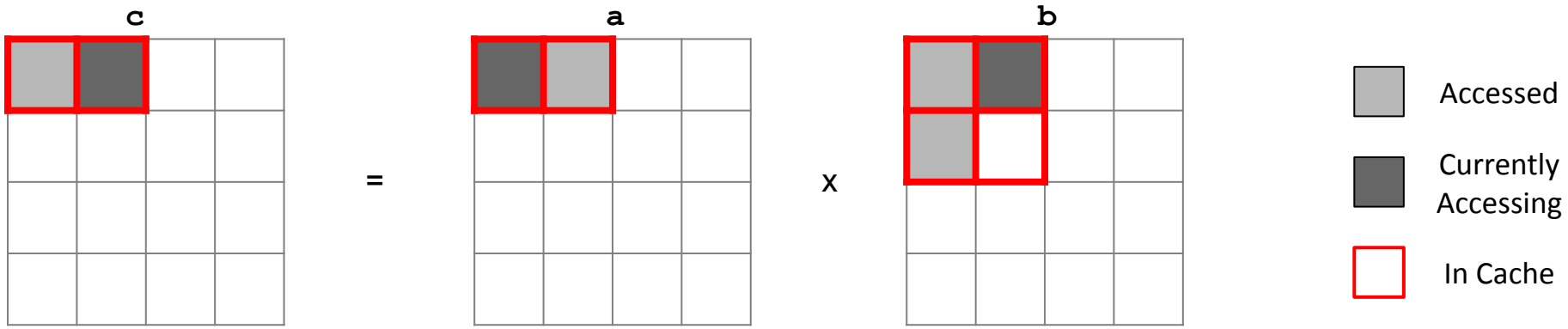
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	???



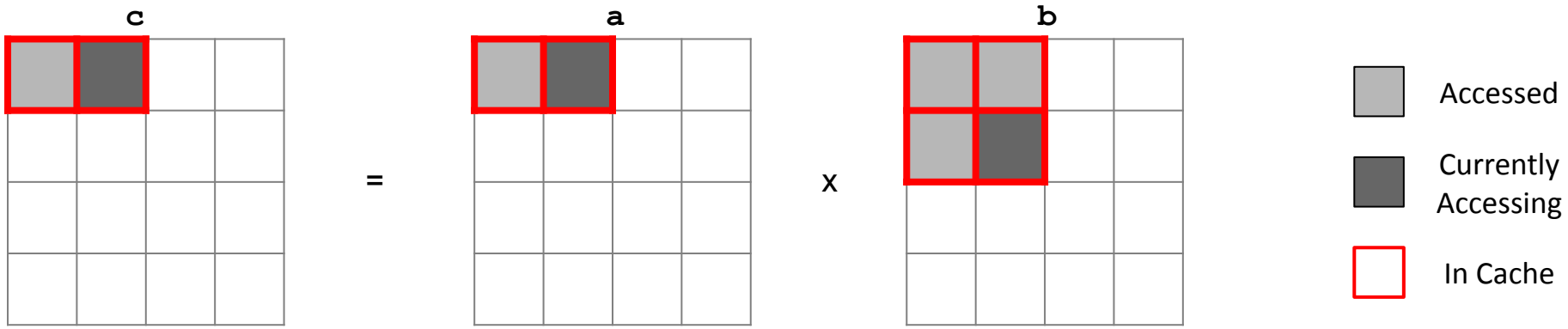
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)



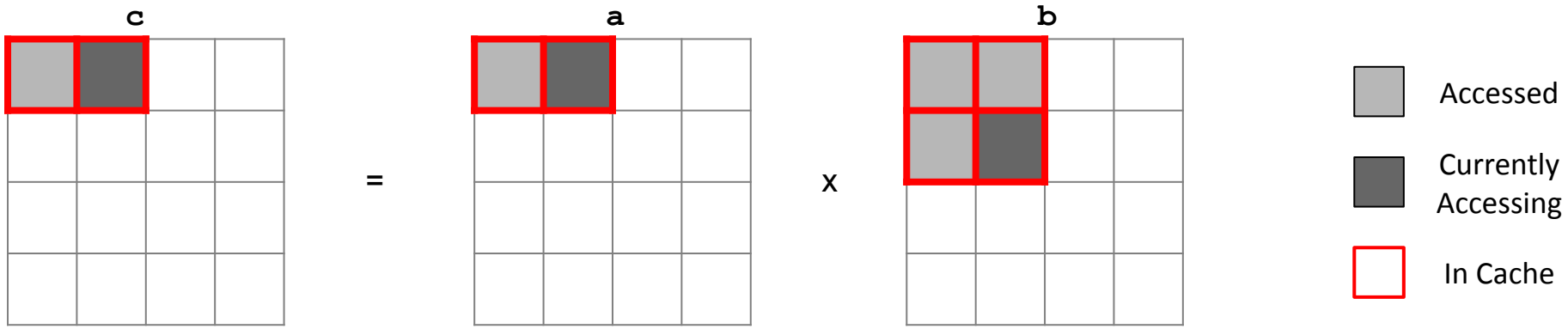
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	???



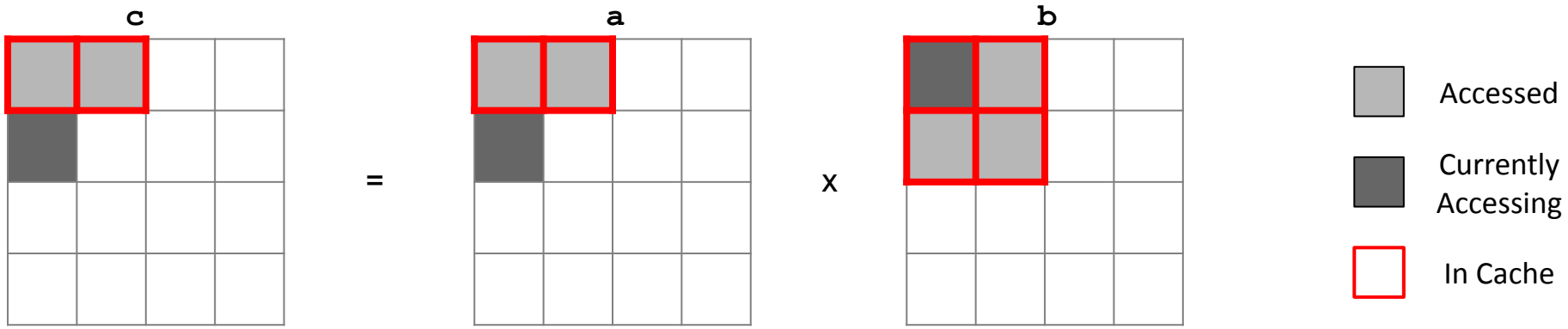
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(h, h)



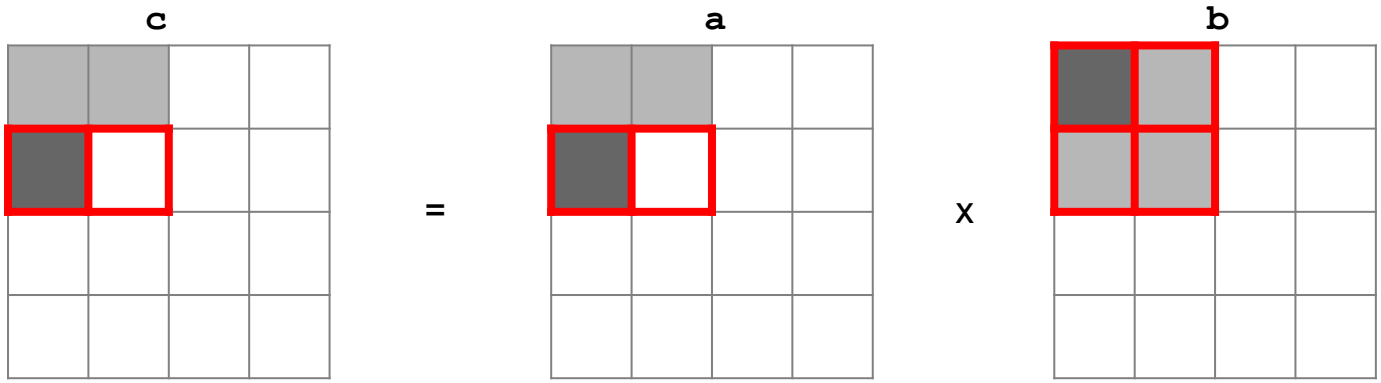
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(h, h)
3	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	???



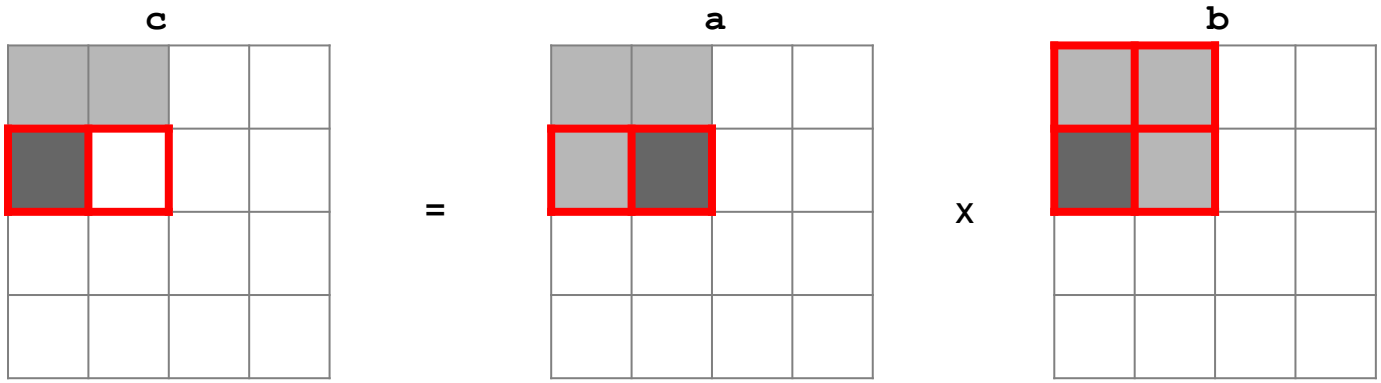
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(h, h)
3	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, h)



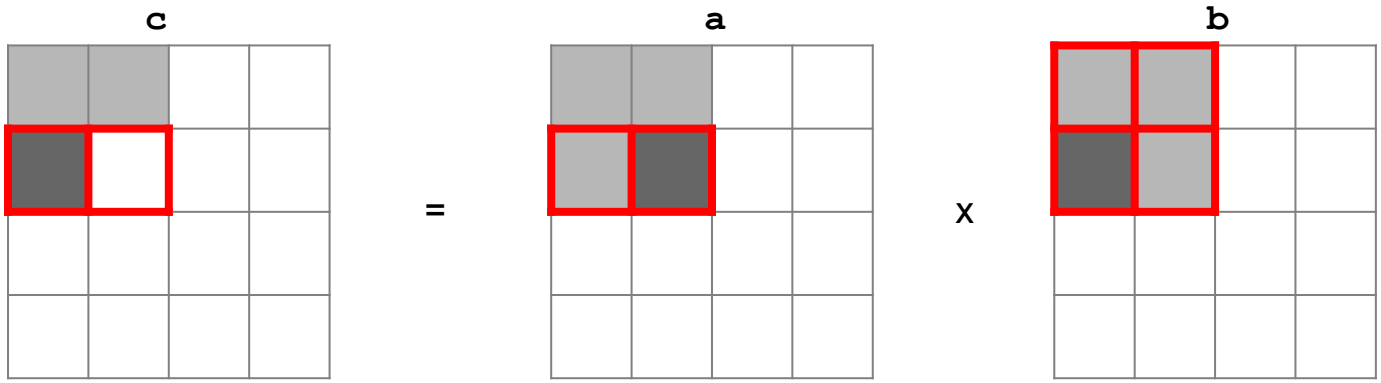
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(h, h)
3	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, h)
4	1	0	0	`c[1][0] += a[1][0] * b[0][0]`	???



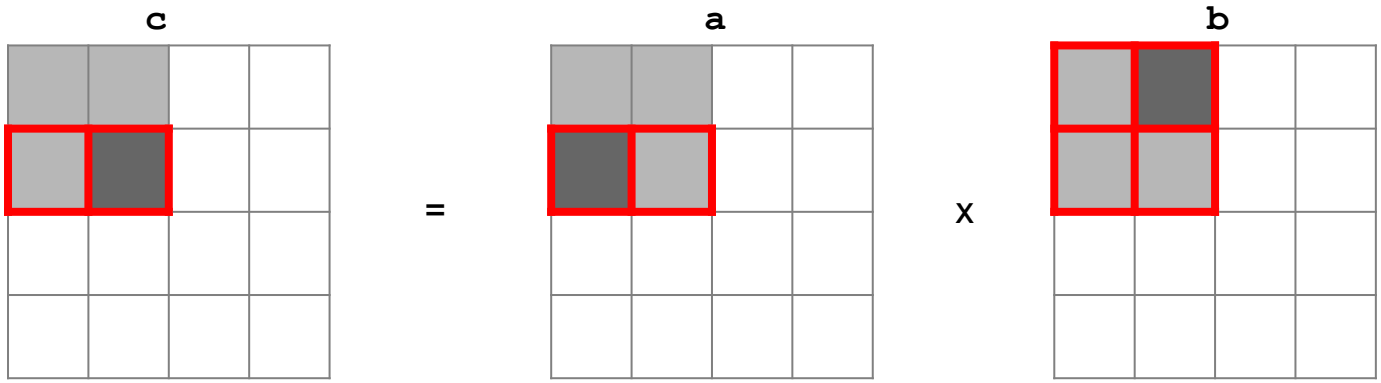
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(h, h)
3	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, h)
4	1	0	0	`c[1][0] += a[1][0] * b[0][0]`	(m, h)



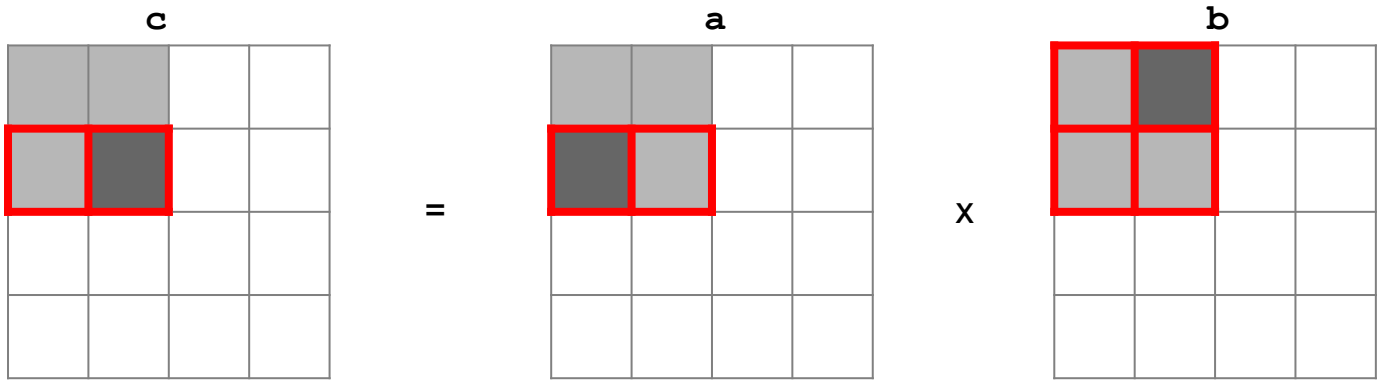
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(h, h)
3	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, h)
4	1	0	0	`c[1][0] += a[1][0] * b[0][0]`	(m, h)
5	1	0	1	`c[1][0] += a[1][1] * b[1][0]`	???



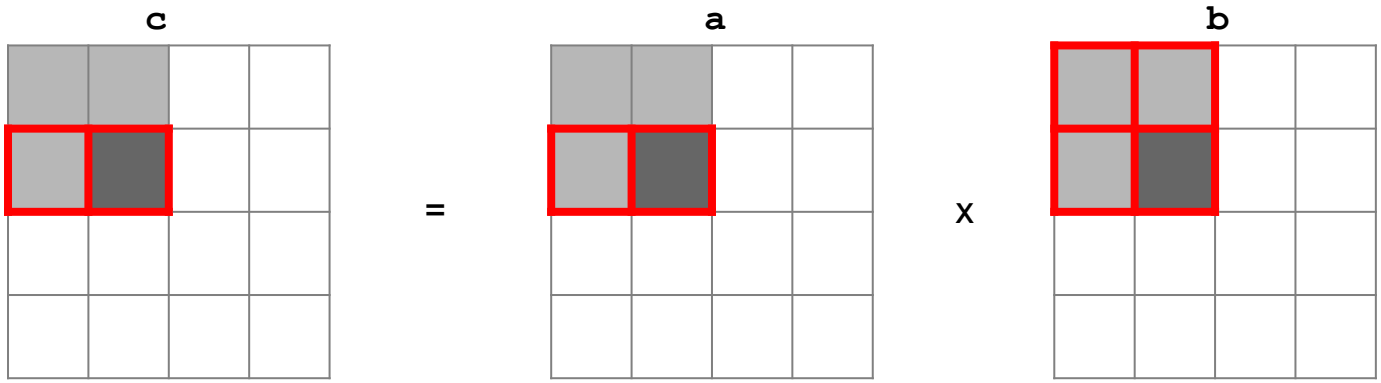
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(h, h)
3	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, h)
4	1	0	0	`c[1][0] += a[1][0] * b[0][0]`	(m, h)
5	1	0	1	`c[1][0] += a[1][1] * b[1][0]`	(h, h)



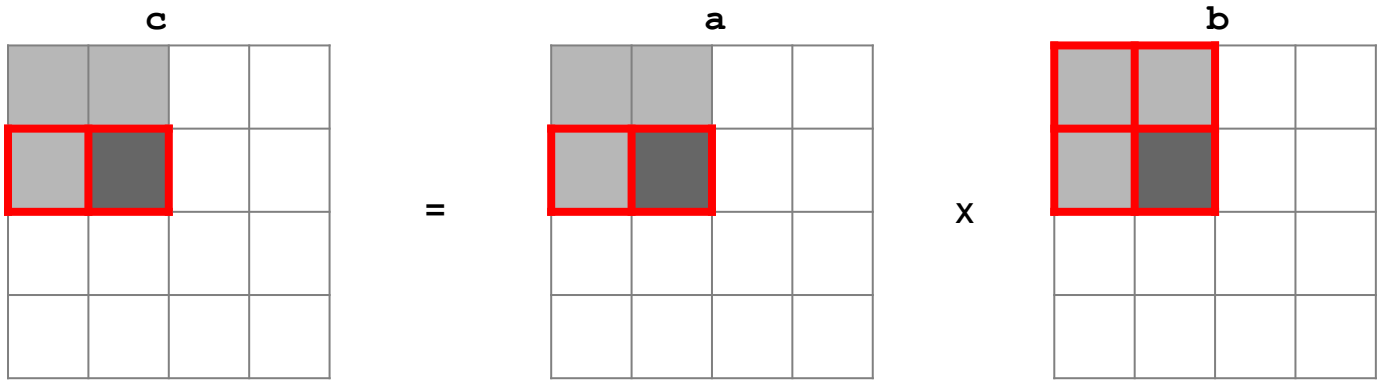
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(h, h)
3	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, h)
4	1	0	0	`c[1][0] += a[1][0] * b[0][0]`	(m, h)
5	1	0	1	`c[1][0] += a[1][1] * b[1][0]`	(h, h)
6	1	1	0	`c[1][1] += a[1][0] * b[0][1]`	???



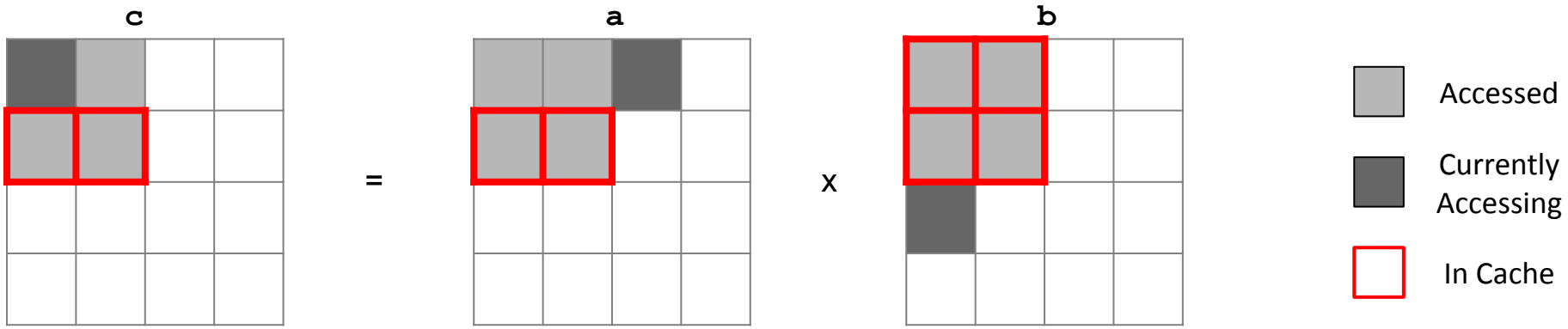
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(h, h)
3	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, h)
4	1	0	0	`c[1][0] += a[1][0] * b[0][0]`	(m, h)
5	1	0	1	`c[1][0] += a[1][1] * b[1][0]`	(h, h)
6	1	1	0	`c[1][1] += a[1][0] * b[0][1]`	(h, h)



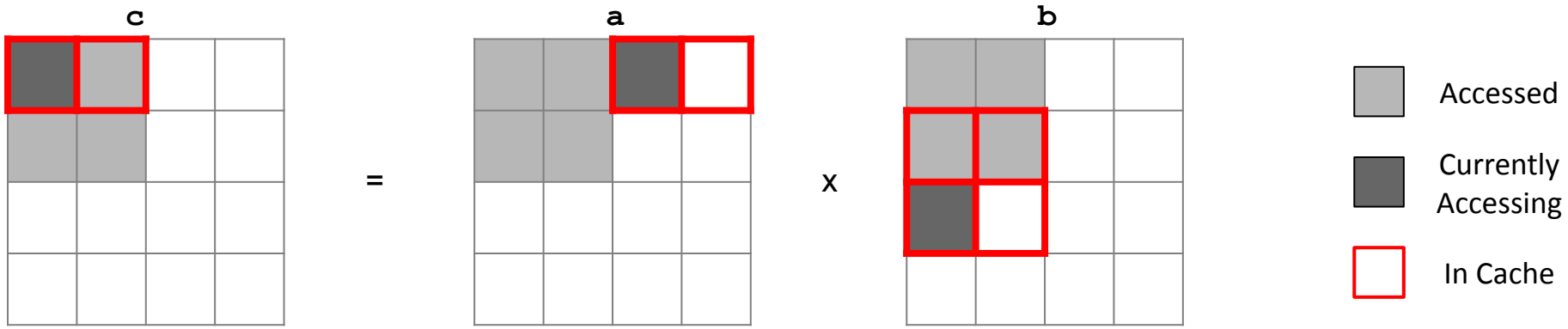
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(h, h)
3	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, h)
4	1	0	0	`c[1][0] += a[1][0] * b[0][0]`	(m, h)
5	1	0	1	`c[1][0] += a[1][1] * b[1][0]`	(h, h)
6	1	1	0	`c[1][1] += a[1][0] * b[0][1]`	(h, h)
7	1	1	1	`c[1][1] += a[1][1] * b[1][1]`	???



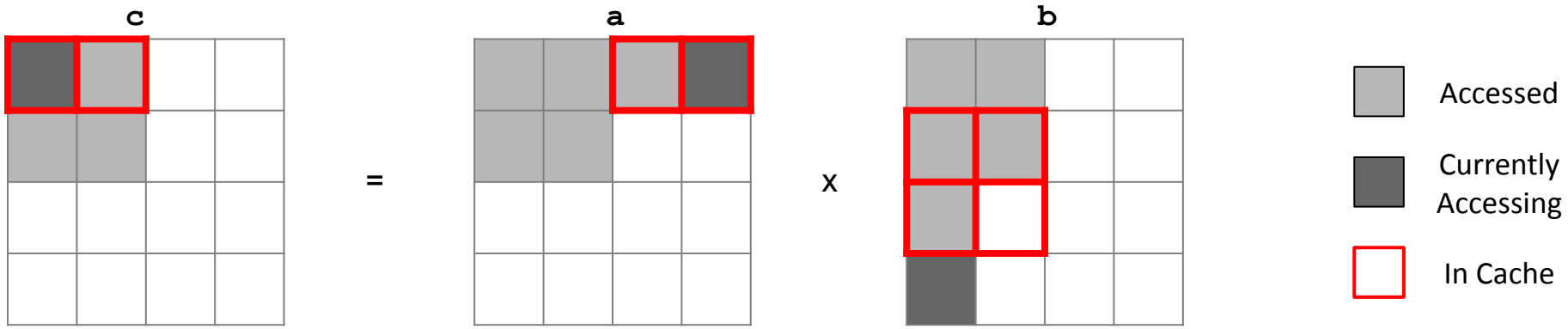
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
0	0	0	0	`c[0][0] += a[0][0] * b[0][0]`	(m, m)
1	0	0	1	`c[0][0] += a[0][1] * b[1][0]`	(h, m)
2	0	1	0	`c[0][1] += a[0][0] * b[0][1]`	(h, h)
3	0	1	1	`c[0][1] += a[0][1] * b[1][1]`	(h, h)
4	1	0	0	`c[1][0] += a[1][0] * b[0][0]`	(m, h)
5	1	0	1	`c[1][0] += a[1][1] * b[1][0]`	(h, h)
6	1	1	0	`c[1][1] += a[1][0] * b[0][1]`	(h, h)
7	1	1	1	`c[1][1] += a[1][1] * b[1][1]`	(h, h)



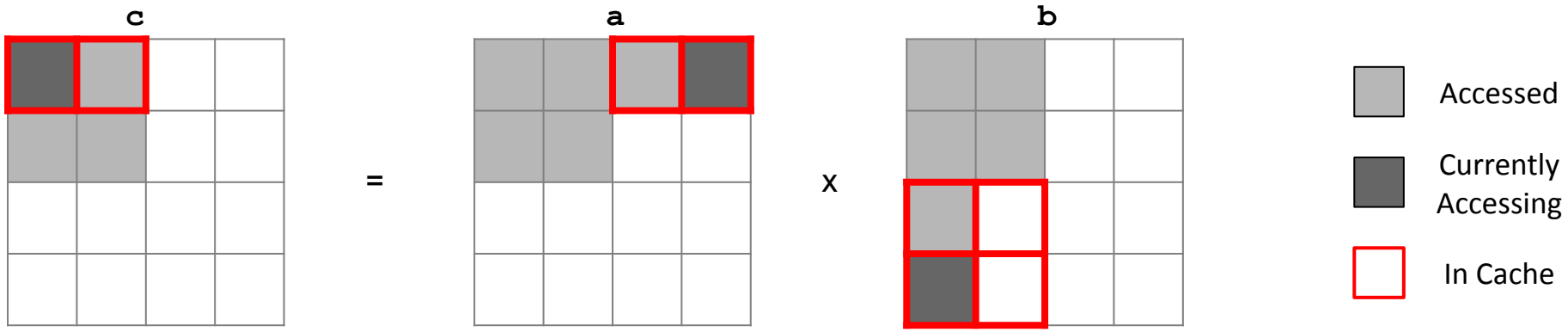
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	???



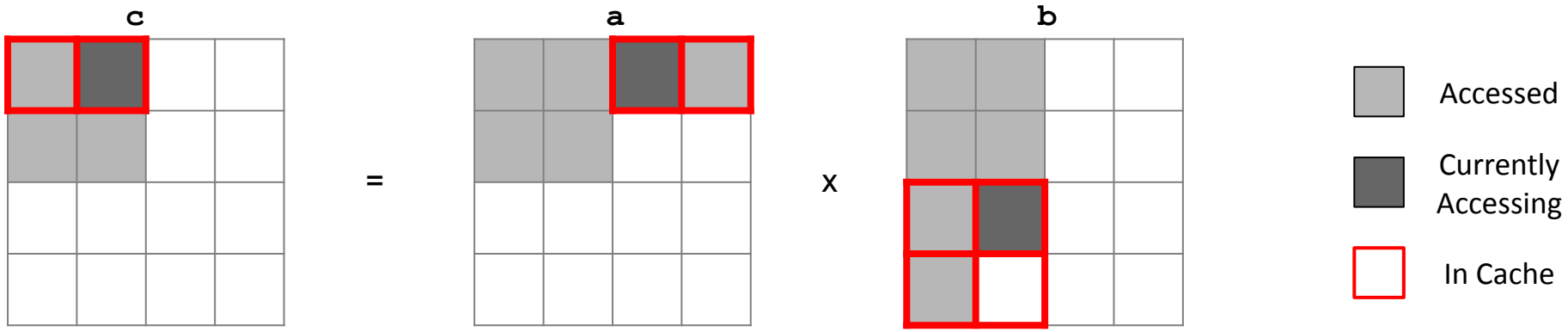
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)



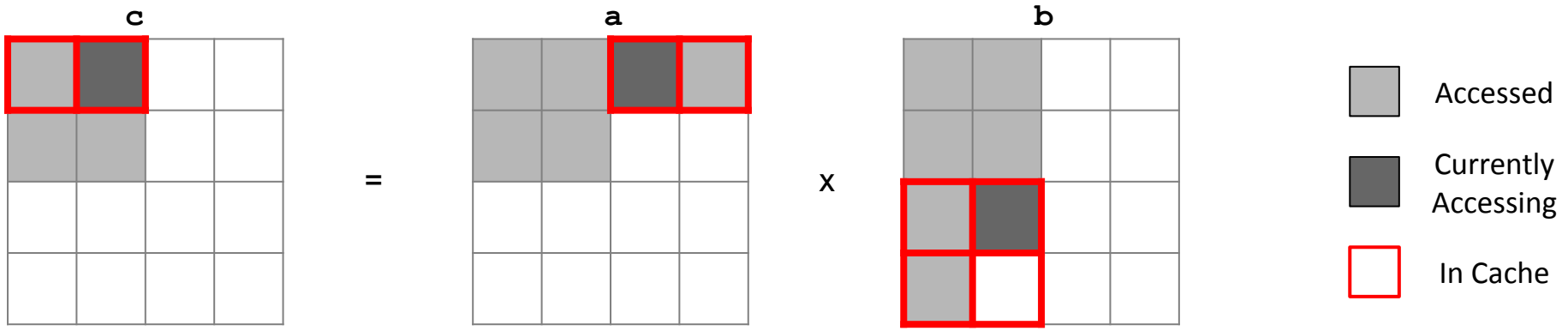
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	???



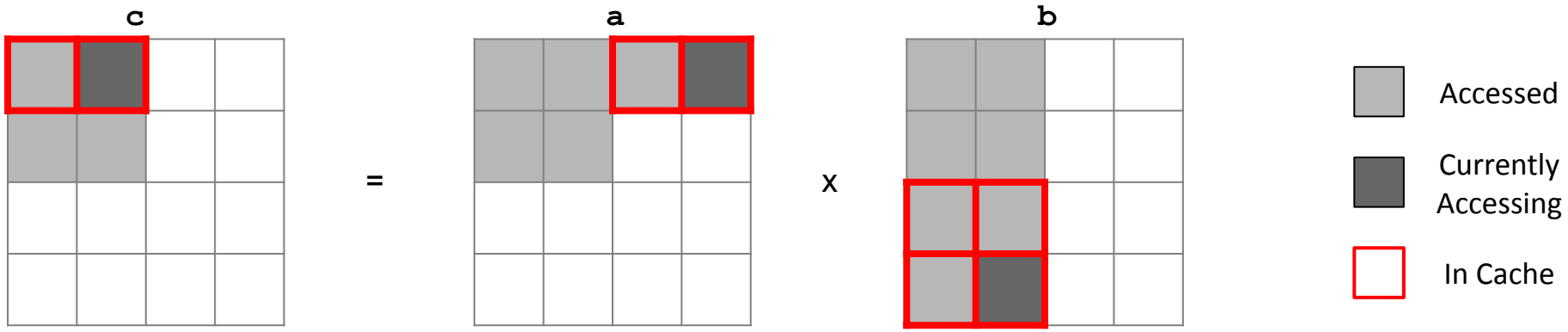
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)



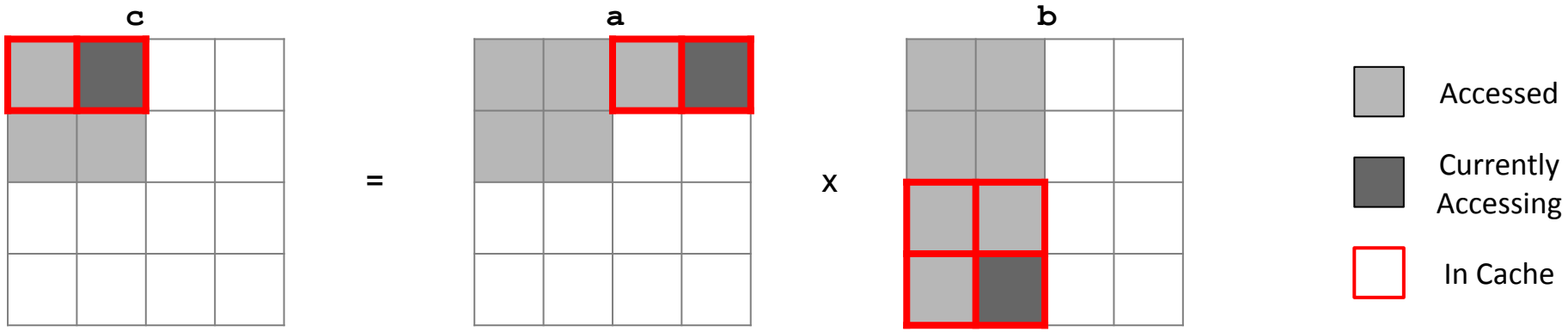
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
10	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	???



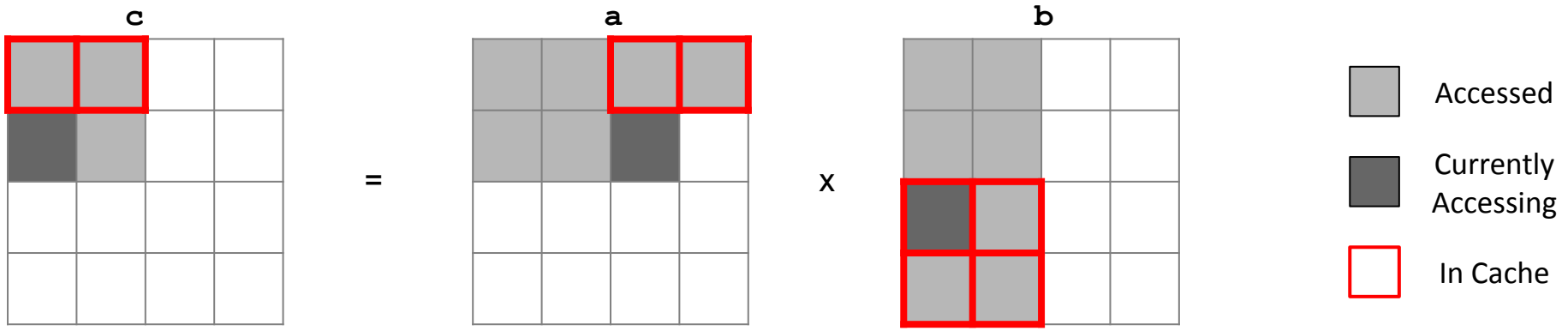
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
10	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(h, h)



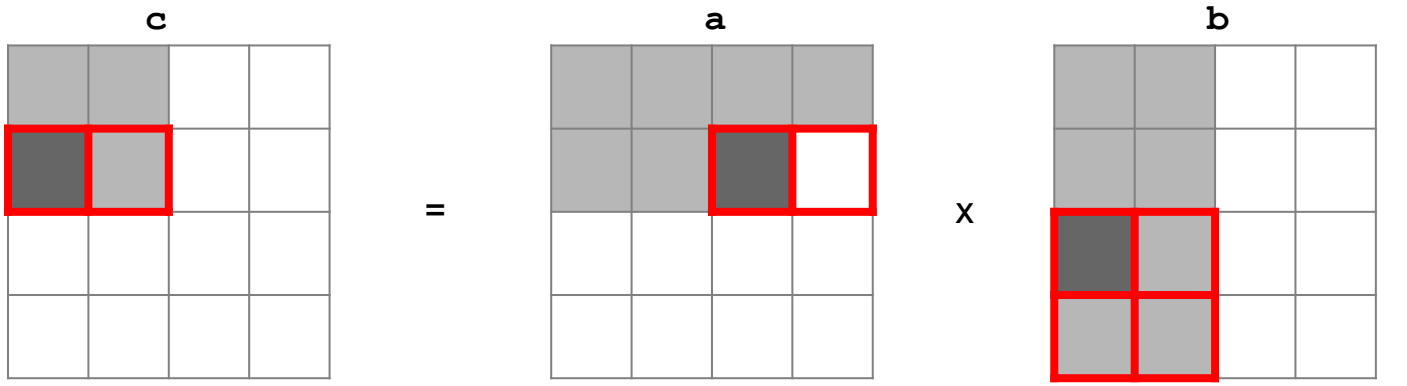
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
10	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(h, h)
11	0	1	3	`c[0][1] += a[0][3] * b[3][1]`	???



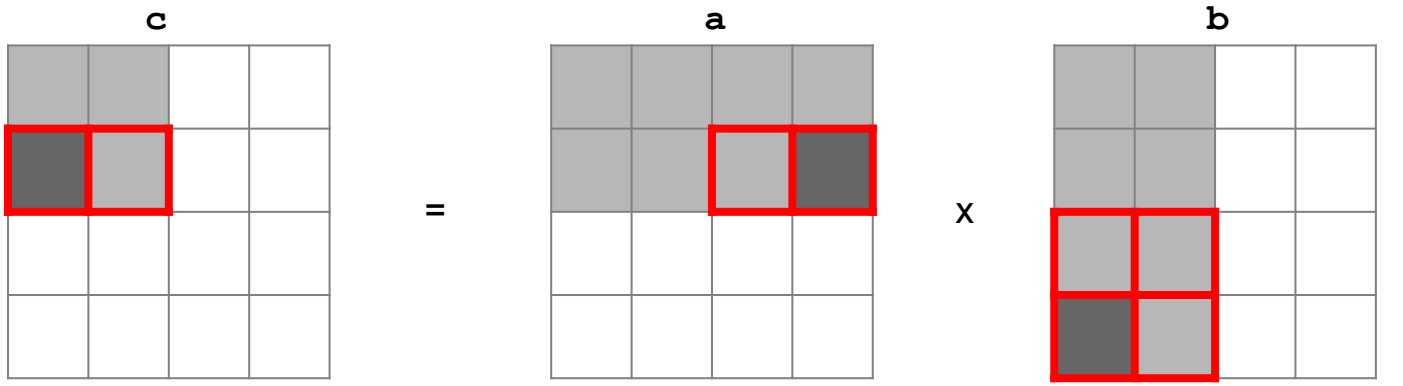
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
10	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(h, h)
11	0	1	3	`c[0][1] += a[0][3] * b[3][1]`	(h, h)



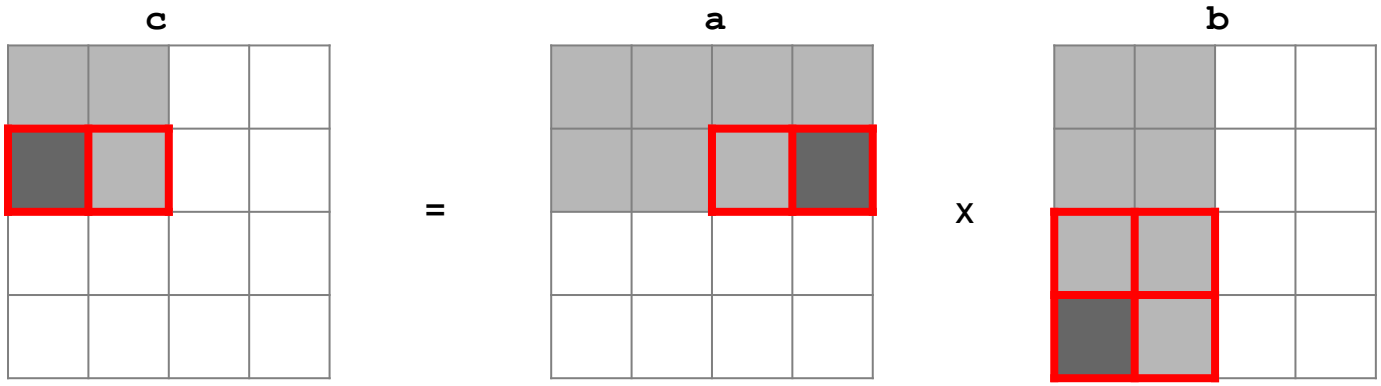
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
10	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(h, h)
11	0	1	3	`c[0][1] += a[0][3] * b[3][1]`	(h, h)
12	1	0	2	`c[1][0] += a[1][2] * b[2][0]`	???



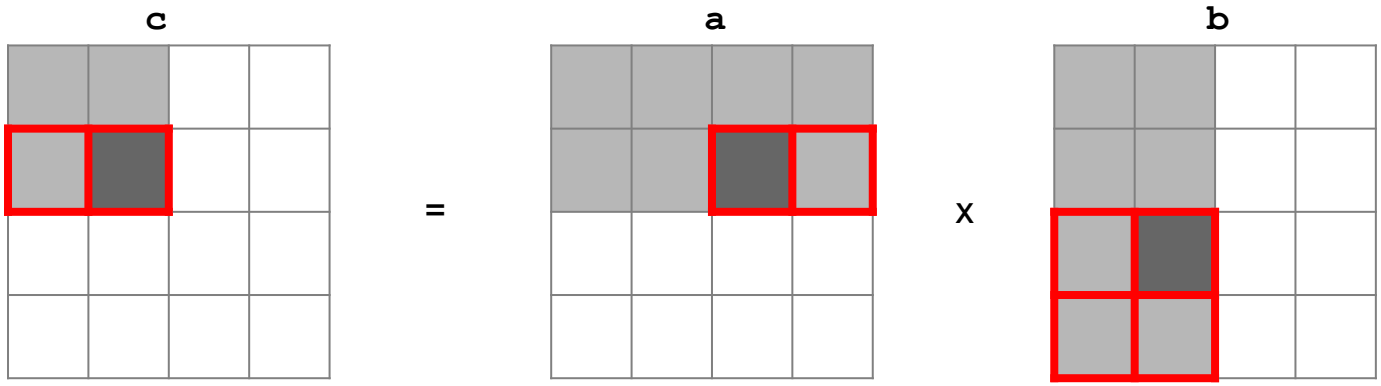
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
10	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(h, h)
11	0	1	3	`c[0][1] += a[0][3] * b[3][1]`	(h, h)
12	1	0	2	`c[1][0] += a[1][2] * b[2][0]`	(m, h)



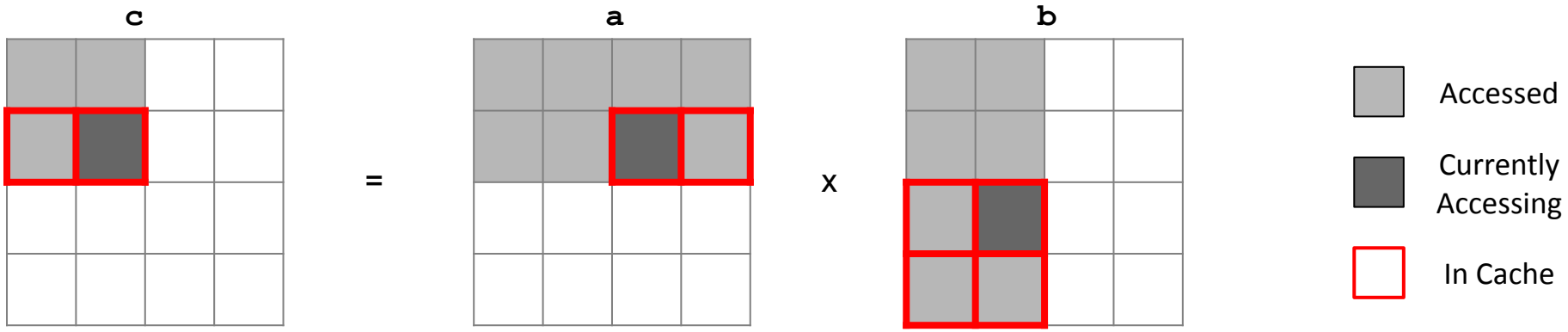
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
10	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(h, h)
11	0	1	3	`c[0][1] += a[0][3] * b[3][1]`	(h, h)
12	1	0	2	`c[1][0] += a[1][2] * b[2][0]`	(m, h)
13	1	0	3	`c[1][0] += a[1][3] * b[3][0]`	???



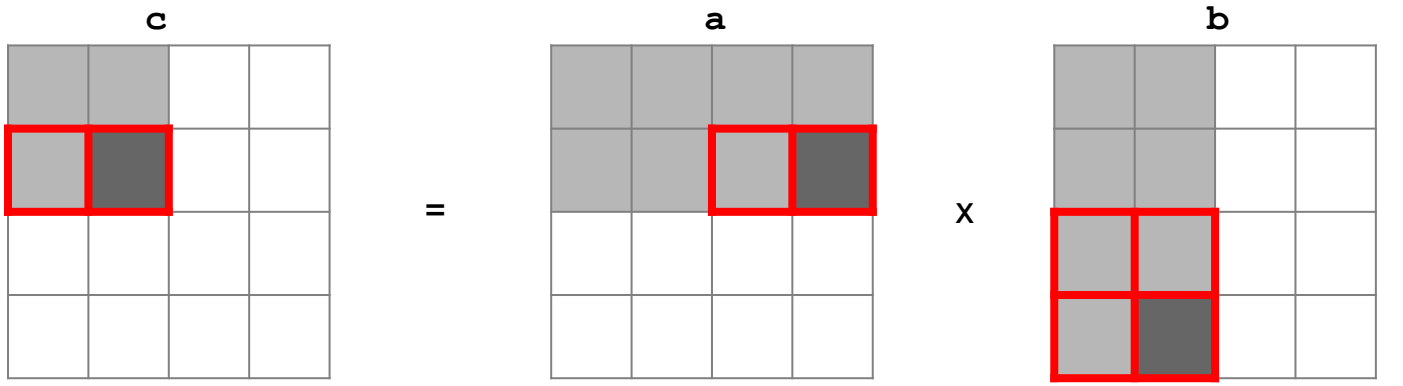
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
10	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(h, h)
11	0	1	3	`c[0][1] += a[0][3] * b[3][1]`	(h, h)
12	1	0	2	`c[1][0] += a[1][2] * b[2][0]`	(m, h)
13	1	0	3	`c[1][0] += a[1][3] * b[3][0]`	(h, h)



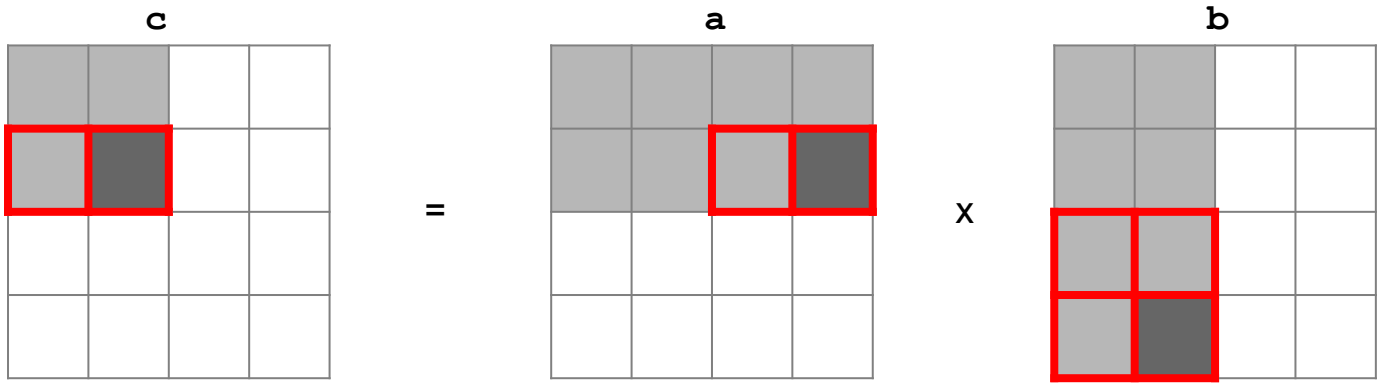
Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
10	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(h, h)
11	0	1	3	`c[0][1] += a[0][3] * b[3][1]`	(h, h)
12	1	0	2	`c[1][0] += a[1][2] * b[2][0]`	(m, h)
13	1	0	3	`c[1][0] += a[1][3] * b[3][0]`	(h, h)
14	1	1	2	`c[1][1] += a[1][2] * b[2][0]`	???



Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
10	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(h, h)
11	0	1	3	`c[0][1] += a[0][3] * b[3][1]`	(h, h)
12	1	0	2	`c[1][0] += a[1][2] * b[2][0]`	(m, h)
13	1	0	3	`c[1][0] += a[1][3] * b[3][0]`	(h, h)
14	1	1	2	`c[1][1] += a[1][2] * b[2][1]`	(h, h)



Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
10	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(h, h)
11	0	1	3	`c[0][1] += a[0][3] * b[3][1]`	(h, h)
12	1	0	2	`c[1][0] += a[1][2] * b[2][0]`	(m, h)
13	1	0	3	`c[1][0] += a[1][3] * b[3][0]`	(h, h)
14	1	1	2	`c[1][1] += a[1][2] * b[2][1]`	(h, h)
15	1	1	3	`c[1][1] += a[1][3] * b[3][1]`	???



Iteration	`i`	`j`	`k`	*Operation*	*Miss?*
8	0	0	2	`c[0][0] += a[0][2] * b[2][0]`	(m, m)
9	0	0	3	`c[0][0] += a[0][3] * b[3][0]`	(h, m)
10	0	1	2	`c[0][1] += a[0][2] * b[2][1]`	(h, h)
11	0	1	3	`c[0][1] += a[0][3] * b[3][1]`	(h, h)
12	1	0	2	`c[1][0] += a[1][2] * b[2][0]`	(m, h)
13	1	0	3	`c[1][0] += a[1][3] * b[3][0]`	(h, h)
14	1	1	2	`c[1][1] += a[1][2] * b[2][1]`	(h, h)
15	1	1	3	`c[1][1] += a[1][3] * b[3][1]`	(h, h)

Blocking: Analyzing Miss Rate

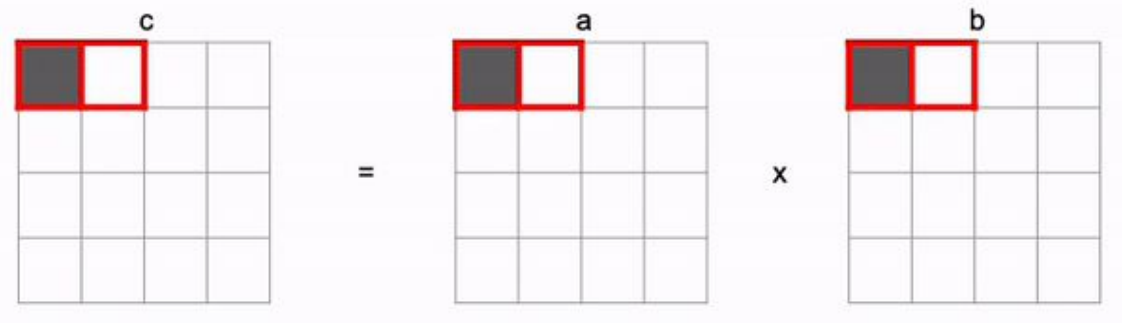
Iteration	*Miss?*
0	(m, m)
1	(h, m)
2	(h, h)
3	(h, h)
4	(m, h)
5	(h, h)
6	(h, h)
7	(h, h)

Iteration	*Miss?*
8	(m, m)
9	(h, m)
10	(h, h)
11	(h, h)
12	(m, h)
13	(h, h)
14	(h, h)
15	(h, h)

- What is the miss rate of **`a`**?
  - ***25%***
- What is the miss rate of **`b`**?
  - ***25%***

Blocking: What Happened?

- Good temporal locality!
- Blocks are re-used while they are still in the cache.



Blocking: What could go wrong?

Blocking with Different Cache

- Great - blocking allows us to better leverage locality! But does this behavior always appear…?
- Let's test it out by apply blocking to a cache that is NOT fully associative!

Blocking Continued

- Suppose our cache is now two-way associative with 2 sets
  - Notice we still have a total of 4 cache lines
- Assuming `C` does not interact with the cache (eg. in a register) and the start of `A,B` point to set 0, we will observe if there is any new behavior.
- Try to pay attention to iterations that leverages locality well versus iterations that are locality's enemy.

Group Activity: B-Accesses

- Let's revisit why we introduced the idea of blocking.
- Ignore all other accesses other than **`B`** and first observe the access pattern and hit/miss pattern for a non-blocking implementation.
  - Where do we see missed opportunities for locality?
- Now do the same analysis but for a blocking implementation. Do you notice anything "better"?

Group Activity: Adding A Back In

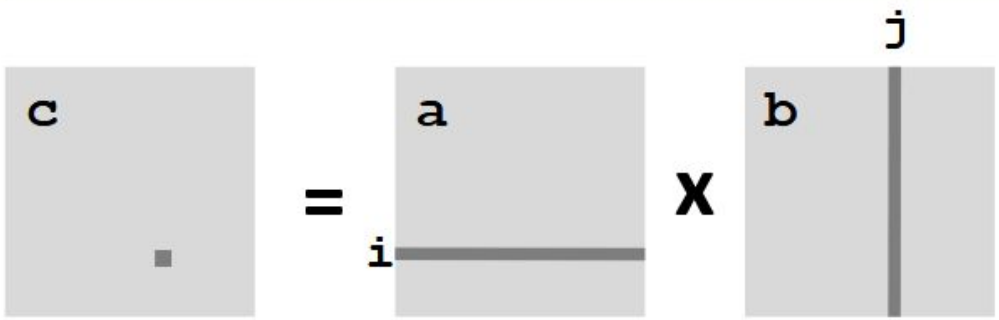
- Now we know how blocking works! Let's return to the original problem and introduce accesses to **A** on top of accesses to **B**.
- What do you observe that is different from a fully associative cache?
- Can you identify cases of good locality? What about bad locality?
- Make sure to draw out the cache/matrix states!

From Lecture: Using blocking to improve temporal locality

Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```

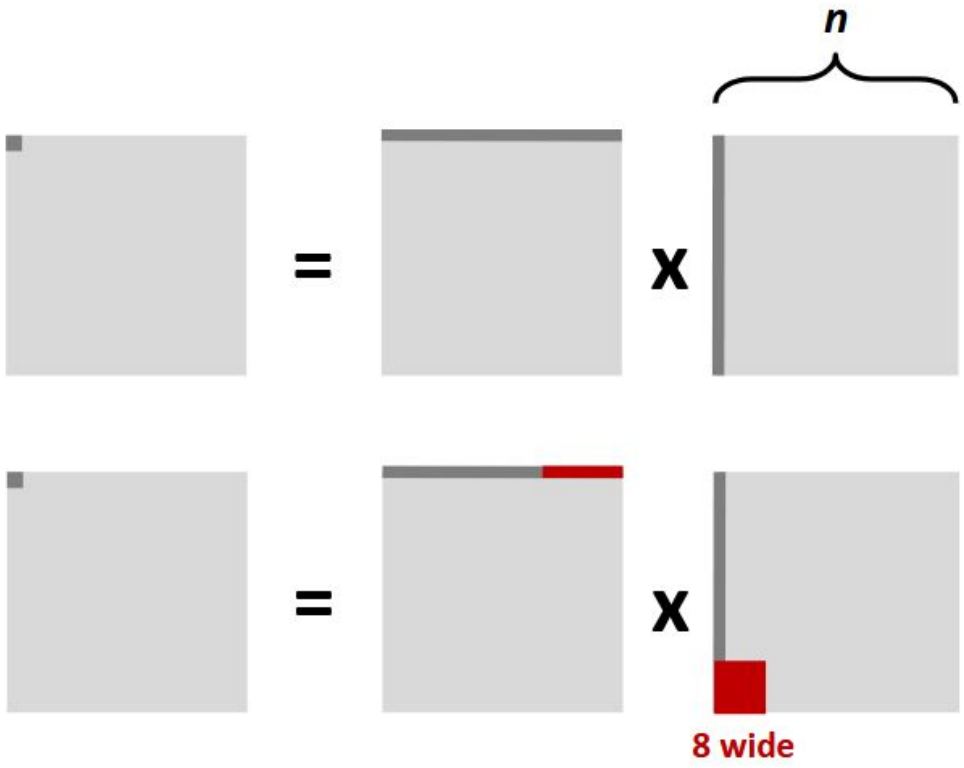


Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << *n* (much smaller than *n*)

- **First iteration:**
  - *n*/8 + n = 9*n*/8 misses

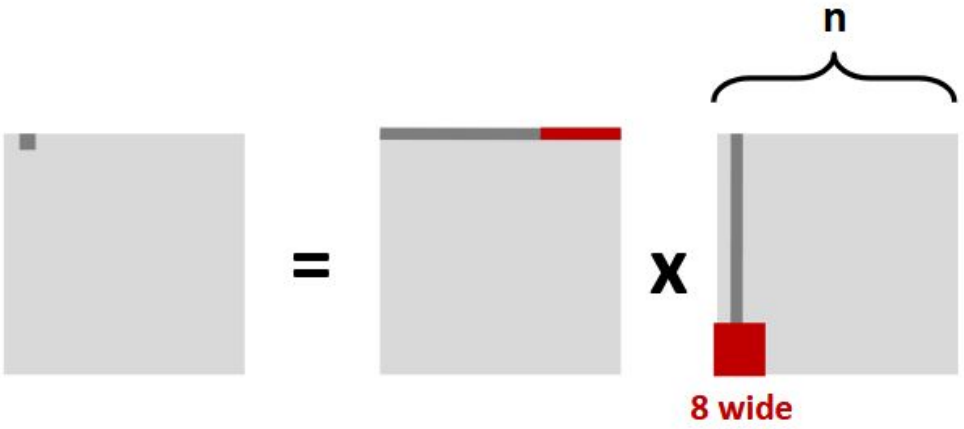
  - Afterwards in cache: (schematic)



Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << *n* (much smaller than *n*)

- **Second iteration:**
  - Again:
    n/8 + *n* = 9*n*/8 misses



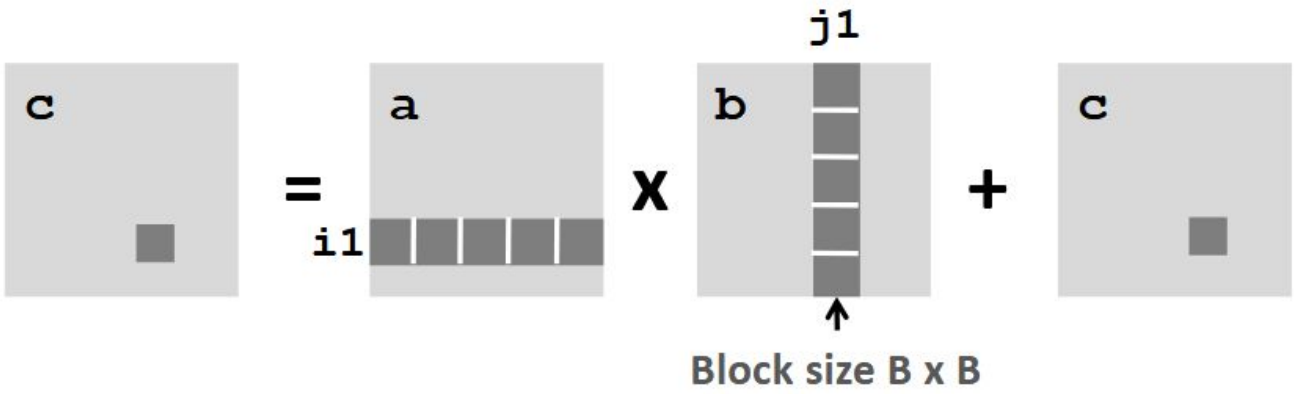
- **Total misses:**
  - (9*n*/8) n^2 = (9/8) n^3

Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

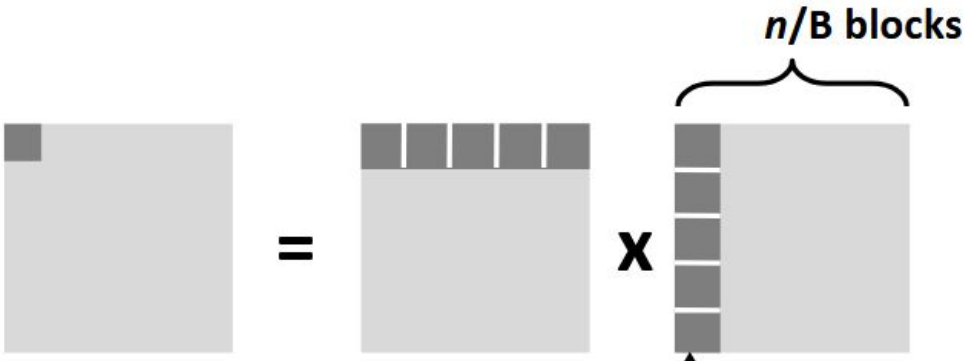
matmult/bmm.c



Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << *n* (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

- **First (block) iteration:**
  - B*B/8 misses for each block
  - $2n/B \times B^2/8 = nB/4$ (omitting matrix c)

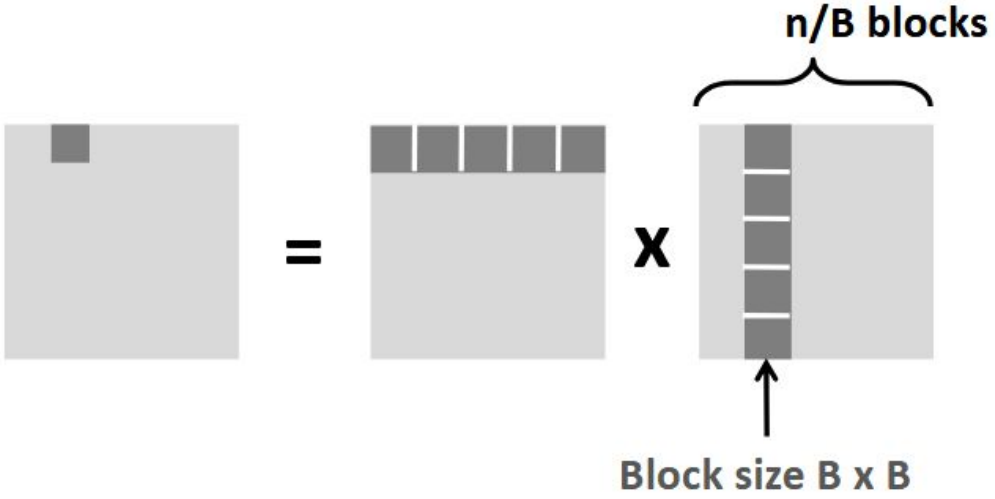


  - Afterwards in cache (schematic)

Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << *n* (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

- **Second (block) iteration:**
  - Same as first iteration
  - $2n/B \times B^2/8 = nB/4$



- **Total misses:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$

Blocking Summary

- **No blocking: (9/8) n^3 misses**
- **Blocking: (1/(4B)) n^3 misses**

- **Use largest block size B, such that B satisfies $3B^2 < C$**
  - Fit three blocks in cache! Two input, one output.

- **Reason for dramatic difference:**
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But program has to be written properly

The End!