15-213 Recitation
Caches and Blocking

Your TAs

Monday, February 28th, 2022

Attack Lab Conclusion

- Consider 15-330 Introduction to Computer Security if you enjoyed this lab
- Don’t use functions vulnerable to buffer overflow (like `gets`)
  - Use functions that allow you to specify buffer lengths:
    - `fgets` instead of `gets`
    - `strncpy` instead of `strcpy`
    - `strncat` instead of `strcat`
    - `snprintf` instead of `sprint`
  - Use `sscanf` and `fscanf` with input lengths (`%213s`)

- Stack protection makes buffer overflow very hard…
  - But very hard ≠ impossible!

Agenda

- Logistics
- Cache Lab
- Cache Concepts
- Activity 1: Traces
- Blocking
- Appendix: Examples, Style, Git, fscanf

Logistics

- Cache Lab is due **Thursday, March 3rd**
- NO Midterm!

Cache Lab: Cache Simulator Hints

- Goal: Count hits, misses, evictions and # of dirty bytes

- Procedure
  - Least Recently Used (LRU) replacement policy
  - Structs are good for storing cache line parts (valid bit, tag, LRU counter, etc.)
  - A cache is like a 2D array of cache lines

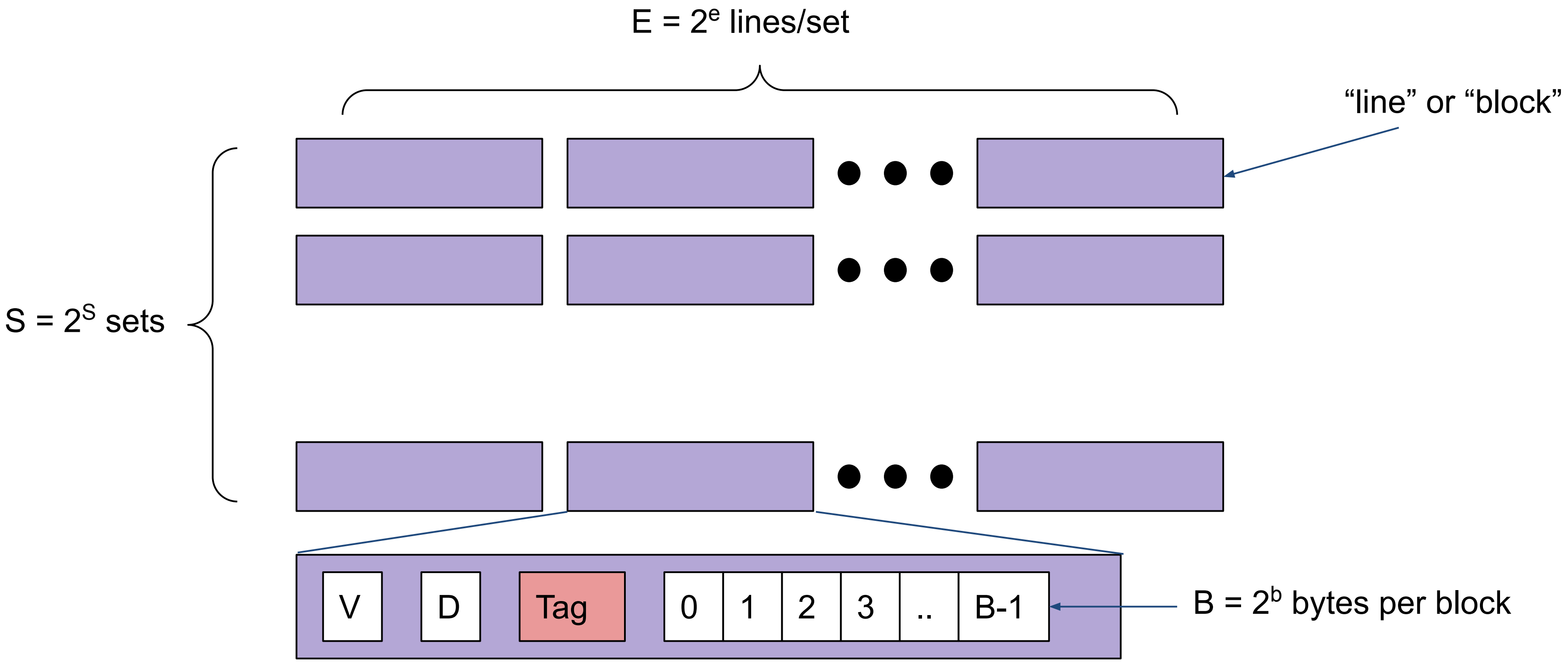
```
struct cache_line cache[S][E];
```

- Your simulator needs to handle different values of S, E, and b (block size) given at run time
  - Dynamically allocate memory!

- Dirty bytes: any payload byte whose corresponding cache block's dirty bit is set (i.e. the payload of that block has been modified, but not yet written back to main memory)

Cache Concepts

Cache Organization



$E = 2^e$ lines/set

"line" or "block"

$S = 2^S$ sets

V

D

Tag

0

1

2

3

..

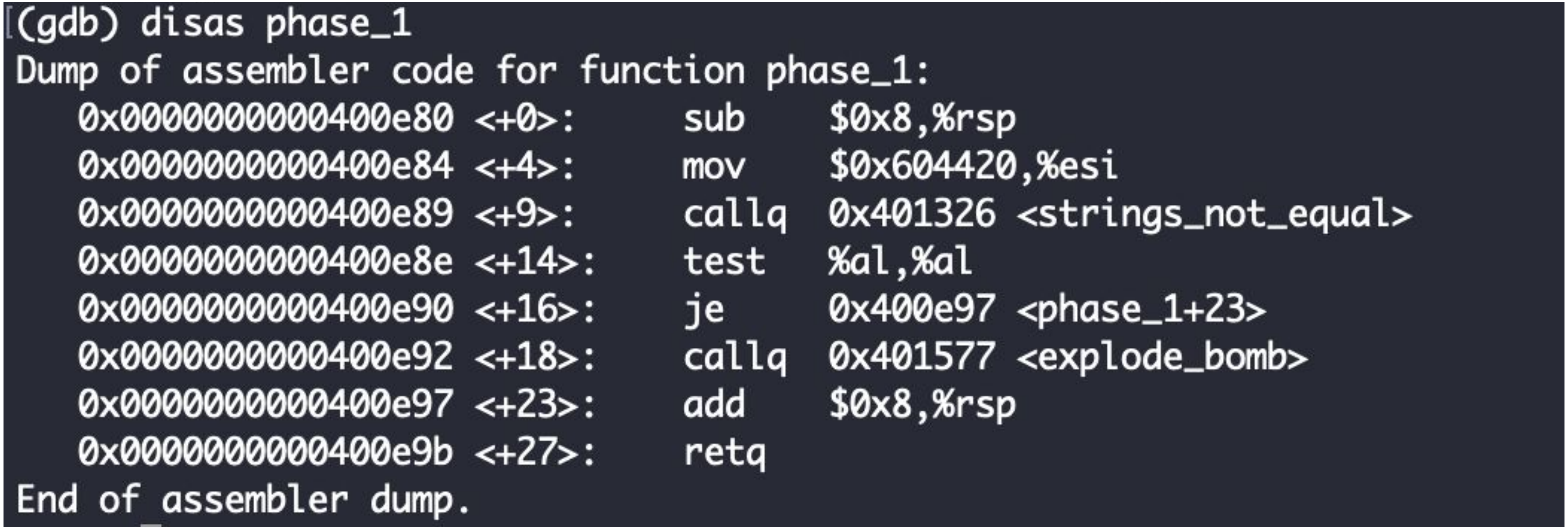
B-1

$B = 2^b$ bytes per block

Cache Read

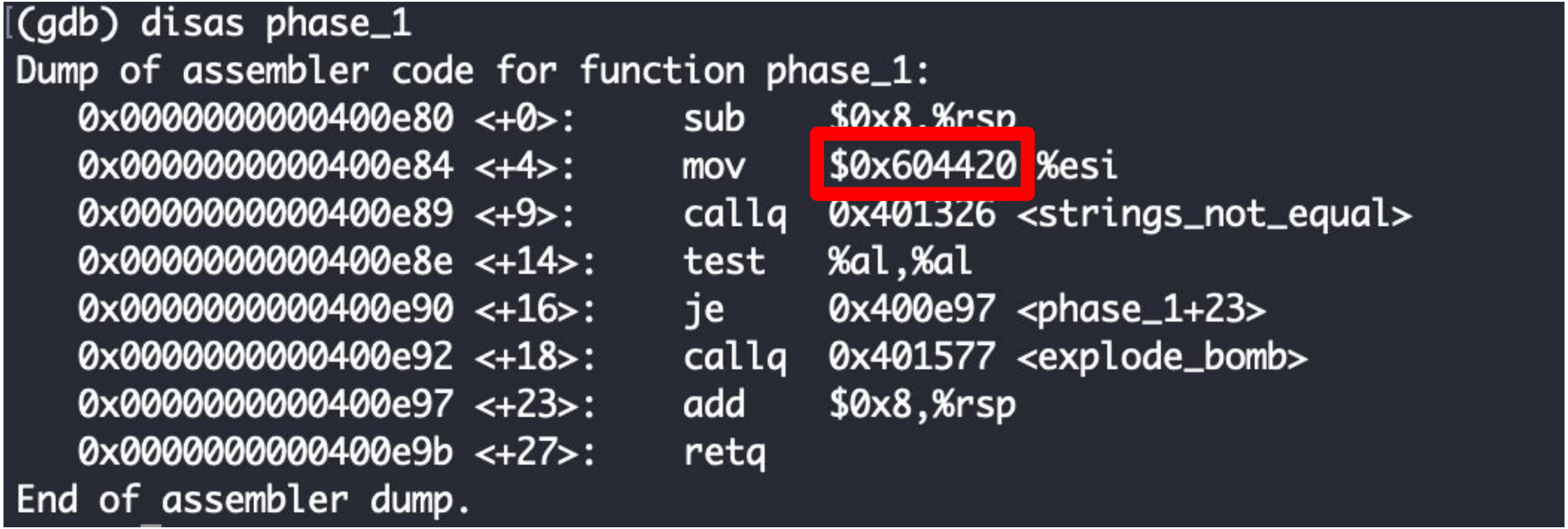
- Address of word: | t bits | s bits | b bits |
  - Tag: t bits
  - Set index: s bits
  - Block offset: b bits
- Steps:
  - Use set index to get appropriate set
  - Loop through lines in set to find matching tag
  - If found and valid bit is set: hit
  - Locate data starting at block offset

Tying it all together: Bomblab



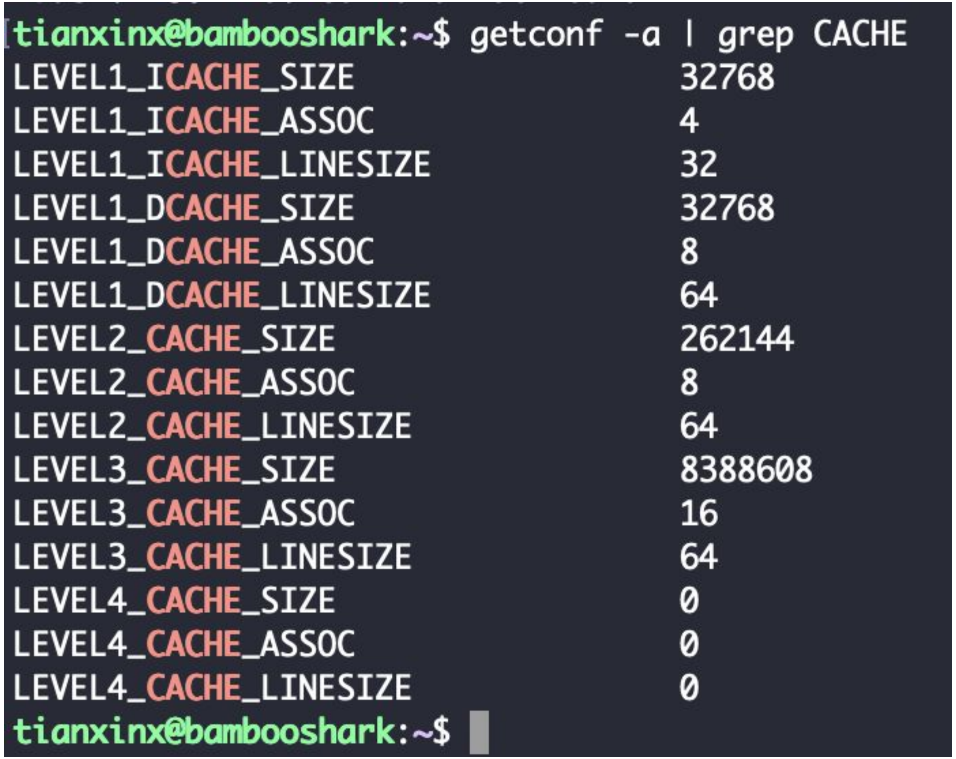
```
[(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x0000000000400e80 <+0>:     sub    $0x8,%rsp
   0x0000000000400e84 <+4>:     mov    $0x604420,%esi
   0x0000000000400e89 <+9>:     callq  0x401326 <strings_not_equal>
   0x0000000000400e8e <+14>:    test   %al,%al
   0x0000000000400e90 <+16>:    je     0x400e97 <phase_1+23>
   0x0000000000400e92 <+18>:    callq  0x401577 <explode_bomb>
   0x0000000000400e97 <+23>:    add    $0x8,%rsp
   0x0000000000400e9b <+27>:    retq
End of assembler dump.
```

Tying it all together: Bomblab



```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x0000000000400e80 <+0>:     sub    $0x8,%rsp
   0x0000000000400e84 <+4>:     mov    $0x604420,%esi
   0x0000000000400e89 <+9>:     callq  0x401326 <strings_not_equal>
   0x0000000000400e8e <+14>:    test   %al,%al
   0x0000000000400e90 <+16>:    je     0x400e97 <phase_1+23>
   0x0000000000400e92 <+18>:    callq  0x401577 <explode_bomb>
   0x0000000000400e97 <+23>:    add    $0x8,%rsp
   0x0000000000400e9b <+27>:    retq
End of assembler dump.
```

Tying it all together: Bomblab



```
[tianxinx@bambooshark:~$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                4
LEVEL1_ICACHE_LINESIZE             32
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 8
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  8388608
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
tianxinx@bambooshark:~$
```

For the L1 dCache (data)

C = 32768 (32 kb)
E = 8
B = 64
S = 64

How did we get S?

Tying it all together: Bomblab

- 64 bit address space: $m = 64$
- $b = 6$
- $s = 6$
- $t = 52$

Tying it all together: Bomblab

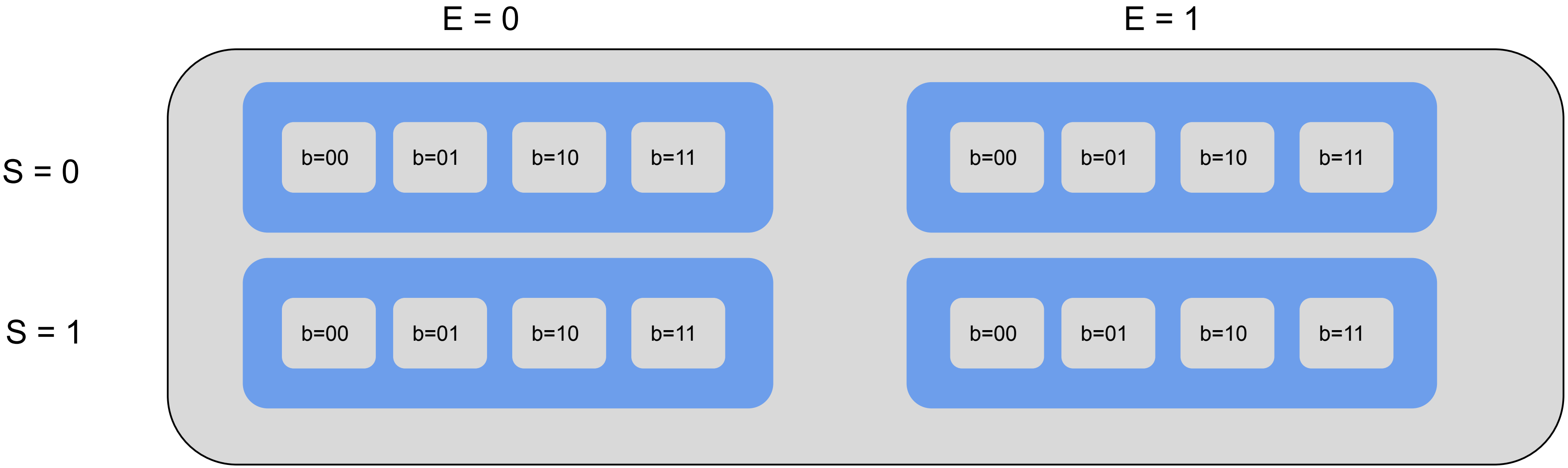
0x00604420 → 0b00000000011000000100010000100000

- tag bits: 00000000011000000100
- set index bits: 010000
- block offset bits: 100000

Activity 1: Traces

Tracing a Cache

Example Cache: -s 1 -E 2 -b 2 (S=2 B=4)



	E = 0				E = 1			
S = 0	b=00	b=01	b=10	b=11	b=00	b=01	b=10	b=11
S = 1	b=00	b=01	b=10	b=11	b=00	b=01	b=10	b=11

Example Trace

L - Load
S - Store

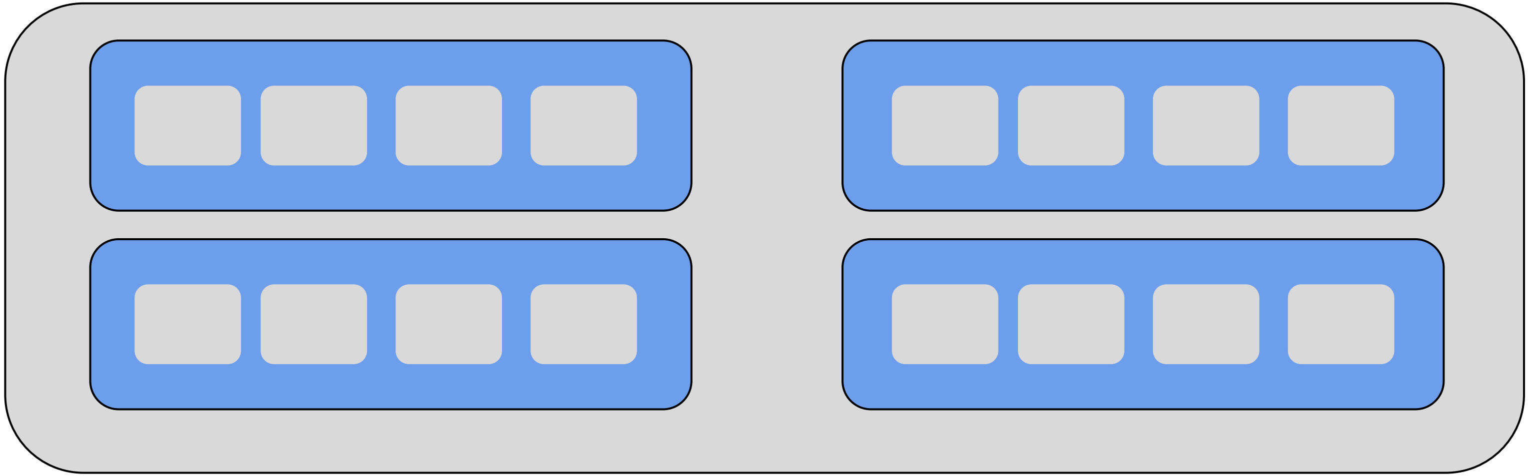
Memory Location

Size

Jack.trace

```
L 0,4
S 0,4
L 0,1
L 6,1
L 5,1
L 6,1
L 7,1
```

Example Trace



Jack.trace

L 0,4

S 0,4

L 0,1

L 6,1

L 5,1

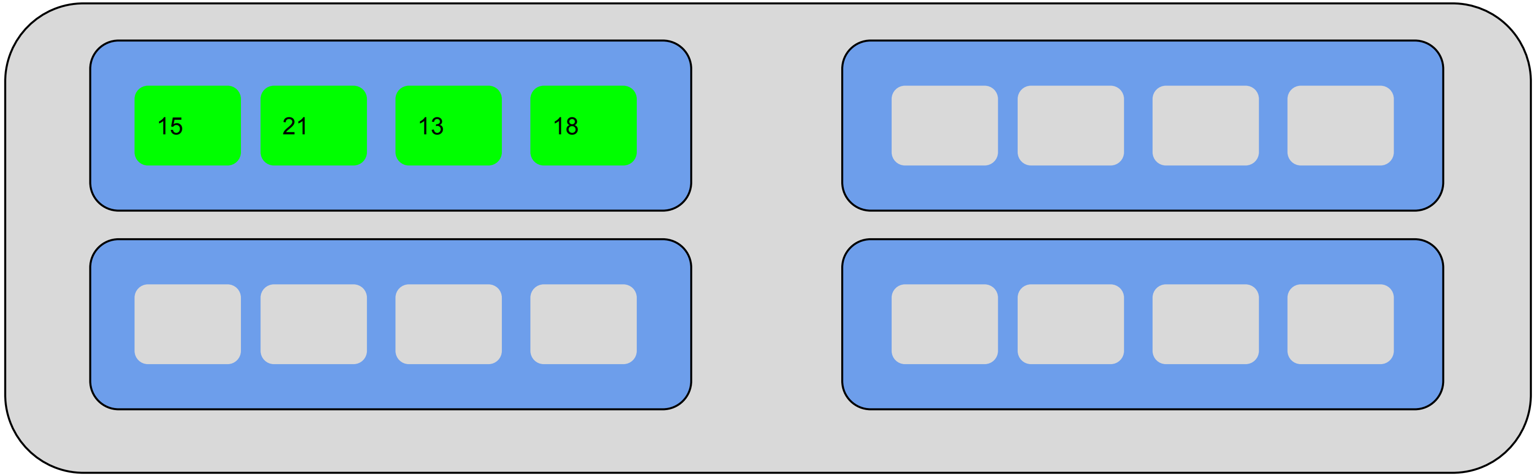
L 6,1

L 7,1

Memory

0x00 15	0x01 21	0x02 13	0x03 18
0x04 51	0x05 30	0x06 ac	0x07 b3

Example Trace



Jack.trace

L 0,4 M

S 0,4

L 0,1

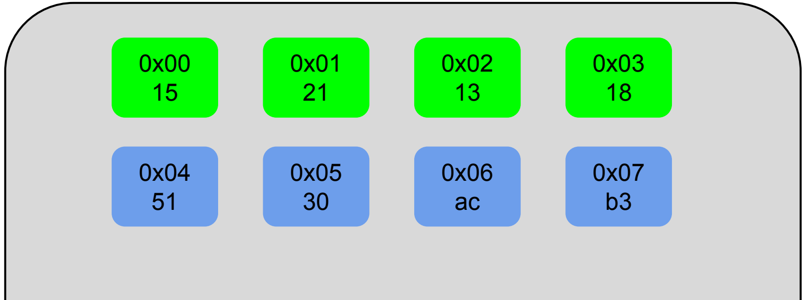
L 6,1

L 5,1

L 6,1

L 7,1

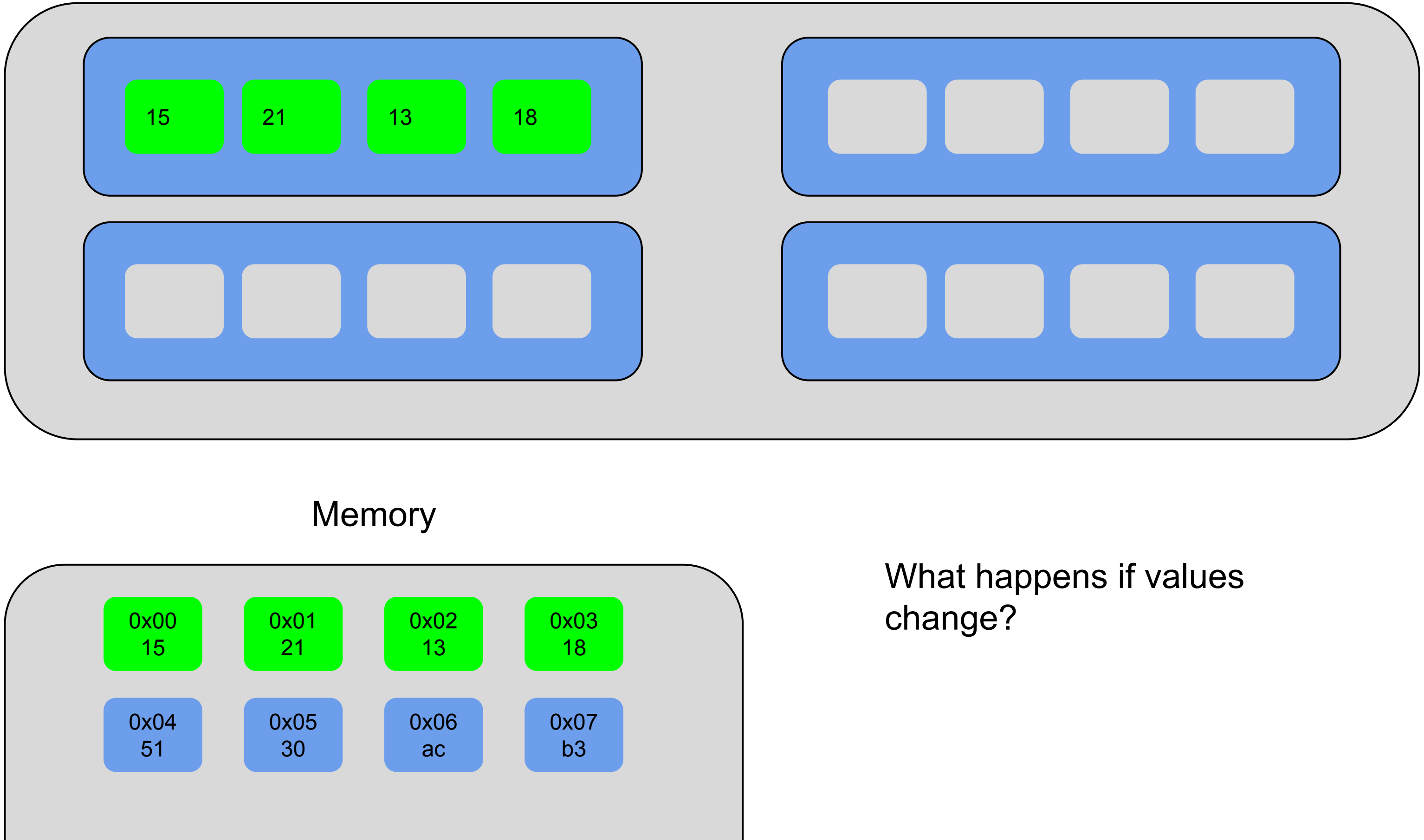
Memory



0x00 15	0x01 21	0x02 13	0x03 18
0x04 51	0x05 30	0x06 ac	0x07 b3

Why that line?
Where are those values from?

Example Trace

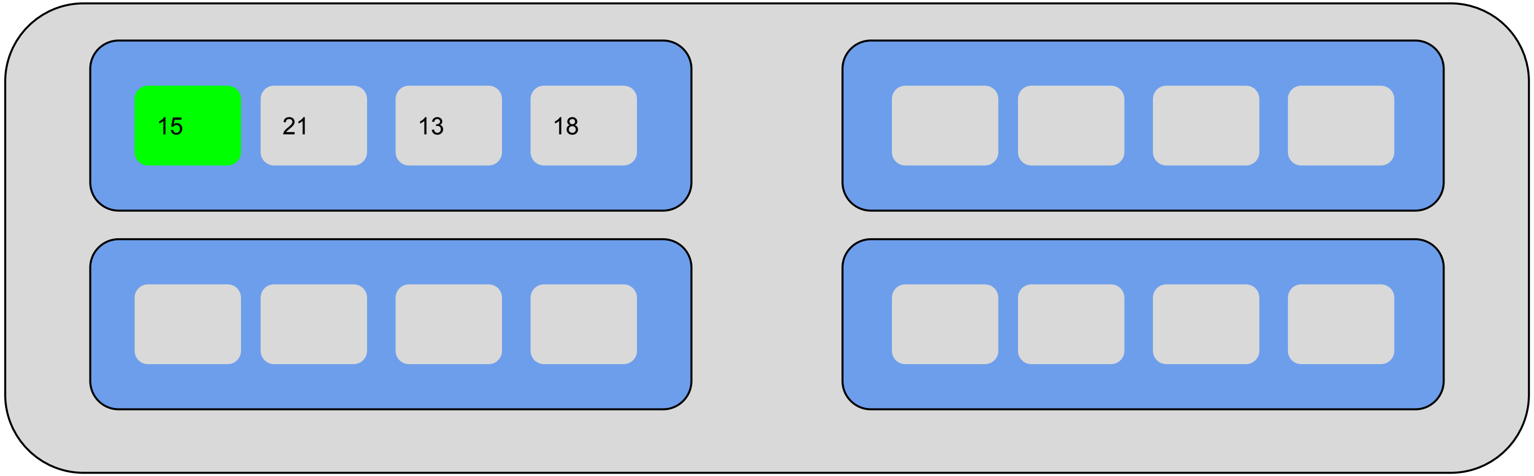


Jack.trace

L 0,4 M
S 0,4 H
L 0,1
L 6,1
L 5,1
L 6,1
L 7,1

What happens if values change?

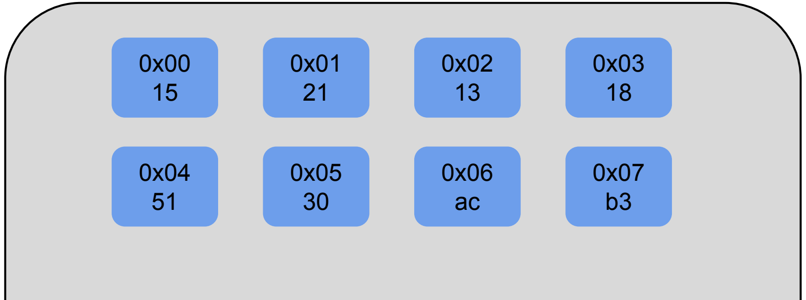
Example Trace



Jack.trace

L 0,4 M
S 0,4 H
L 0,1 H
L 6,1
L 5,1
L 6,1
L 7,1

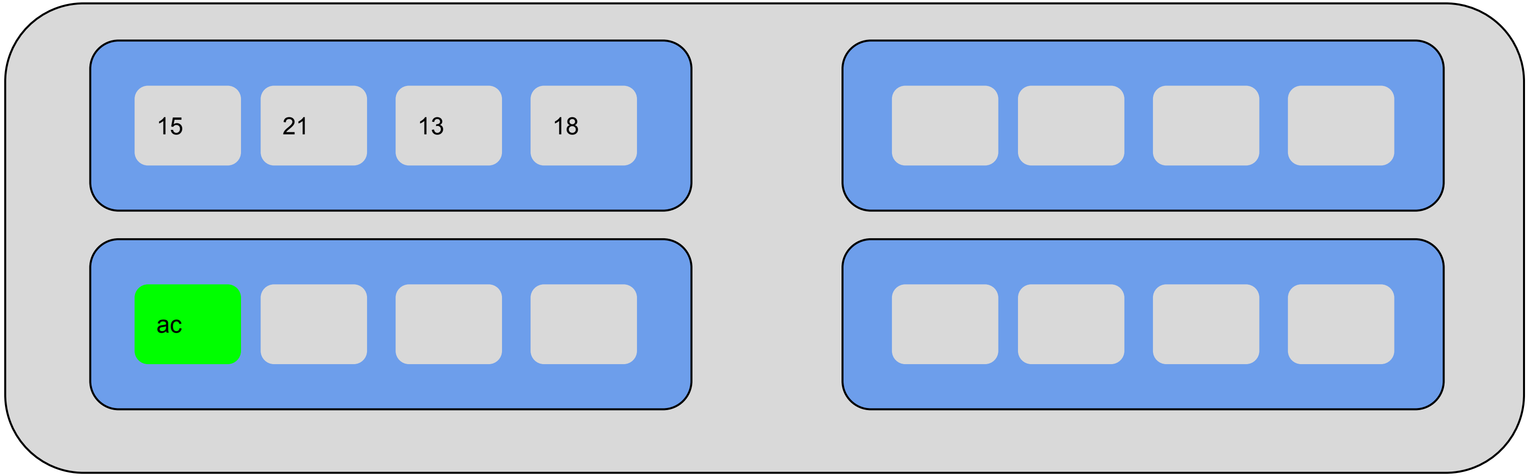
Memory



Why is this still a hit?

What would happen if we had not previously loaded all four bytes?

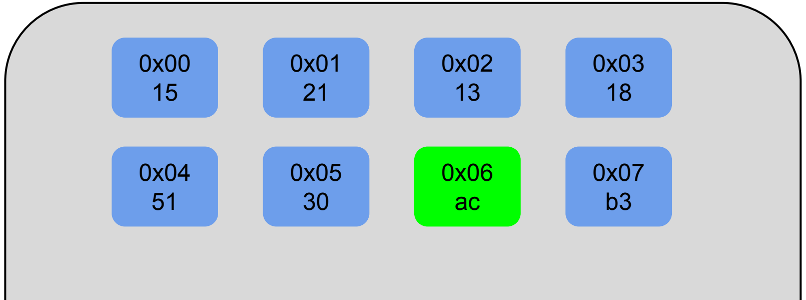
Example Trace



Jack.trace

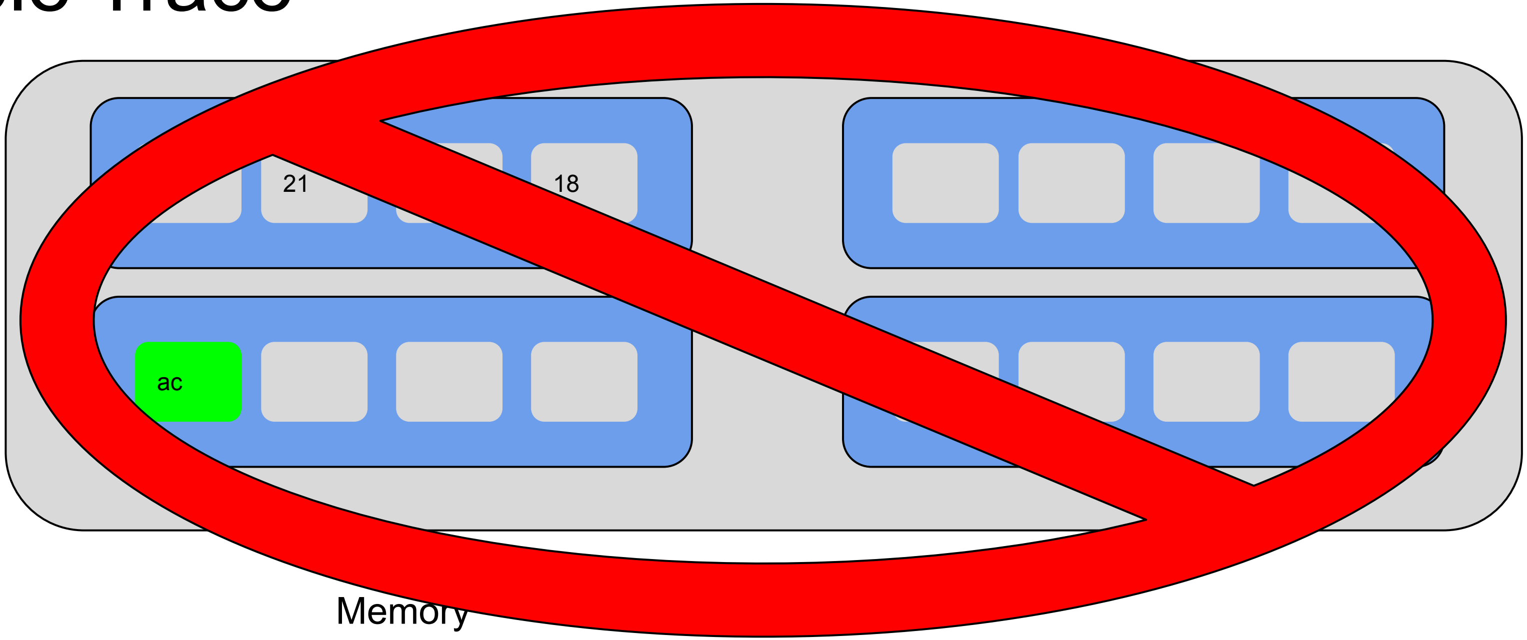
L 0,4 M
S 0,4 H
L 0,1 H
L 6,1 M
L 5,1
L 6,1
L 7,1

Memory



Just one Byte?

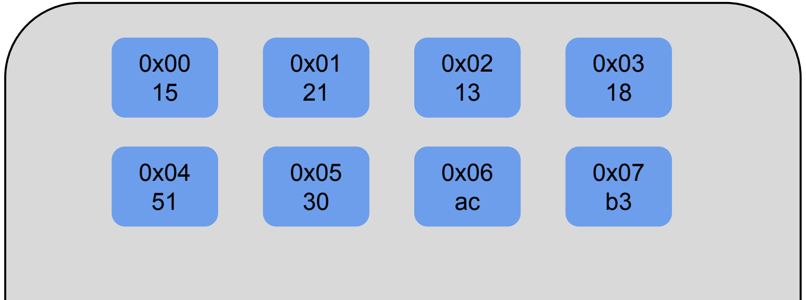
Example Trace



Jack.trace

L 0,4 M
S 0,4 H
L 0,1 H
L 6,1 M
L 5,1
L 6,1
L 7,1

Memory

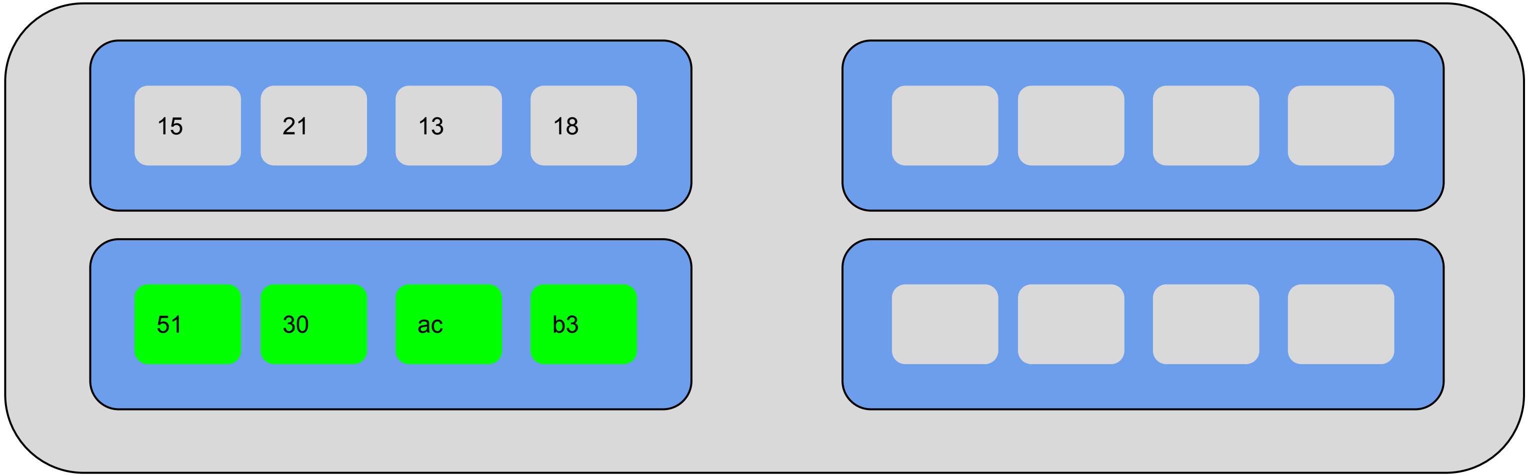


0x00 15	0x01 21	0x02 13	0x03 18
0x04 51	0x05 30	0x06 ac	0x07 b3

Just one Byte?

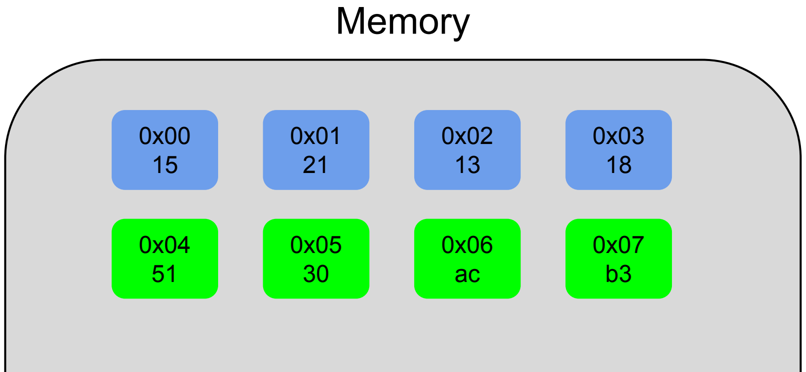
NO!

Example Trace



Jack.trace

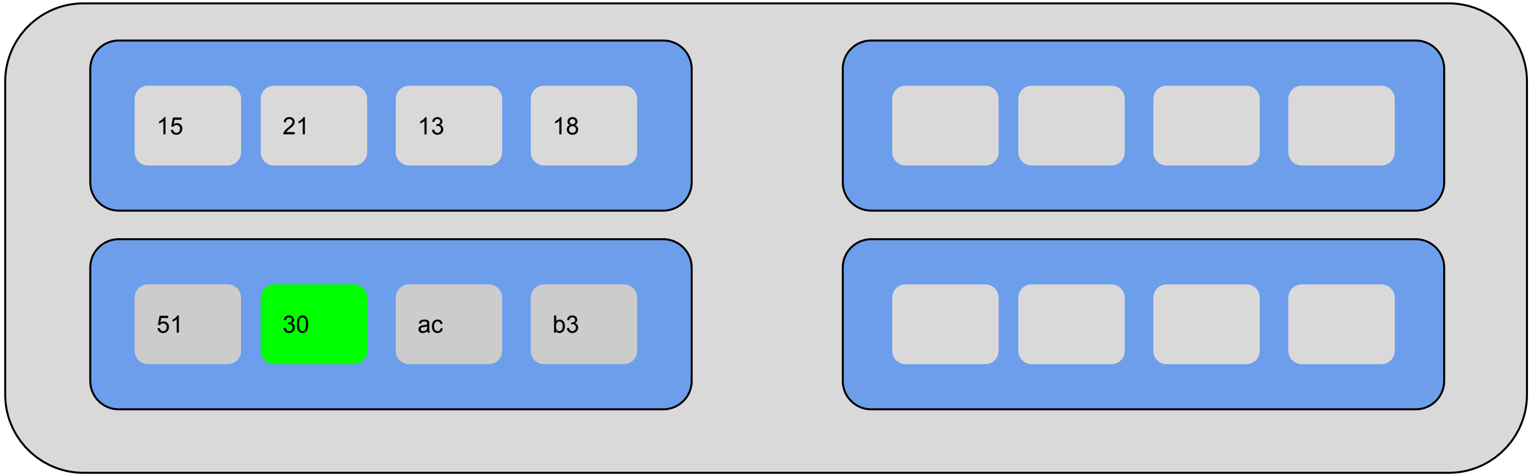
L 0,4 M
S 0,4 H
L 0,1 H
L 6,1 M
L 5,1
L 6,1
L 7,1



Why below and not above?

Why load all four bytes?

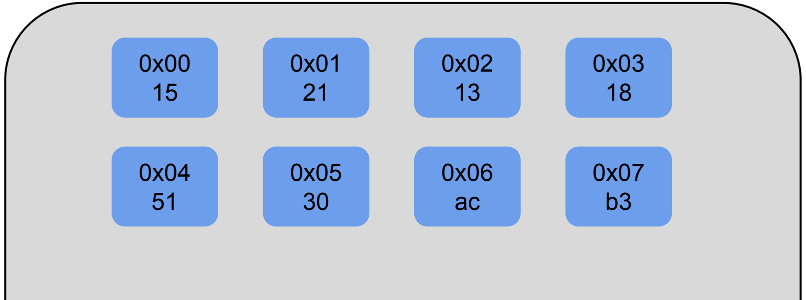
Example Trace



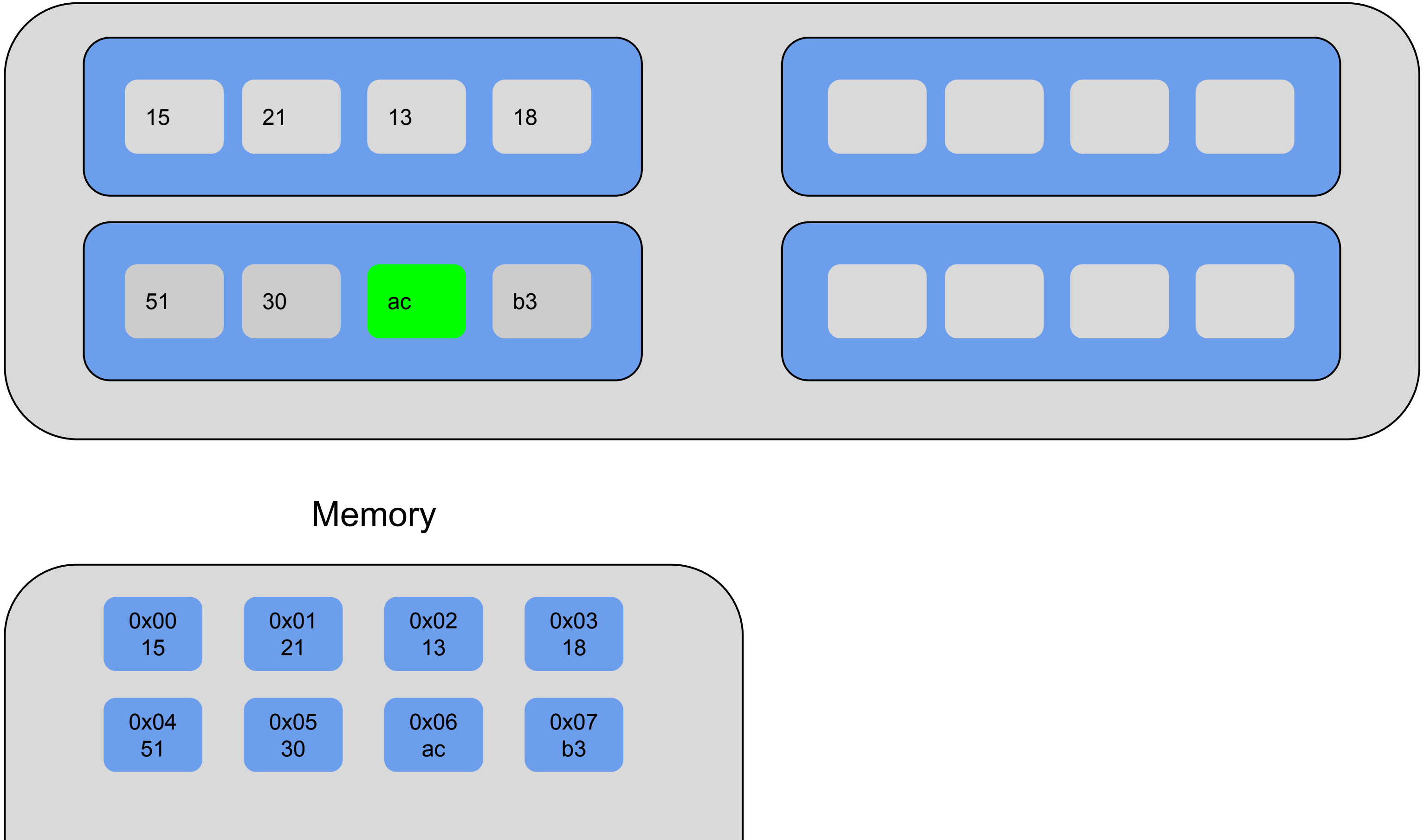
Jack.trace

L 0,4 M
S 0,4 H
L 0,1 H
L 6,1 M
L 5,1 H
L 6,1
L 7,1

Memory



Example Trace



15 21 13 18

51 30 ac b3

Memory

0x00 15	0x01 21	0x02 13	0x03 18
0x04 51	0x05 30	0x06 ac	0x07 b3

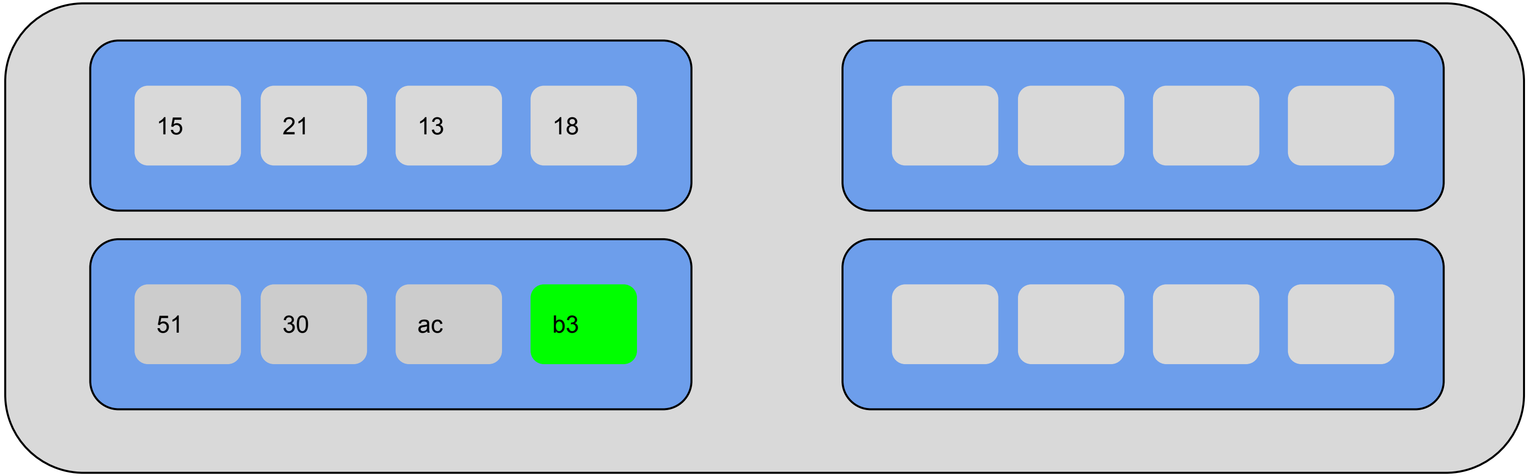
Jack.trace

L 0,4 M
S 0,4 H
L 0,1 H
L 6,1 M
L 5,1 H
L 6,1 H
L 7,1

Example Trace

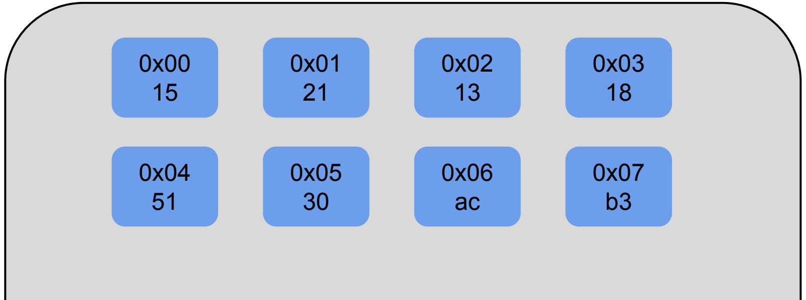
Jack.trace

L 0,4 M
S 0,4 H
L 0,1 H
L 6,1 M
L 5,1 H
L 6,1 H
L 7,1 H



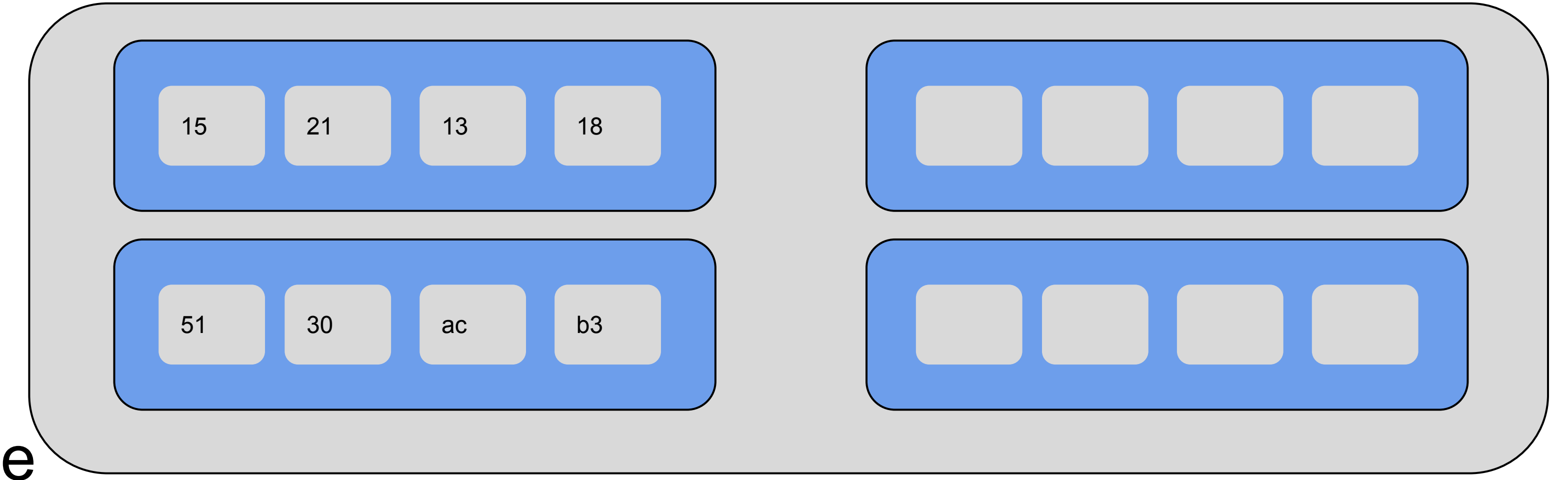
15	21	13	18				
51	30	ac	b3				

Memory



0x00 15	0x01 21	0x02 13	0x03 18
0x04 51	0x05 30	0x06 ac	0x07 b3

Example Trace

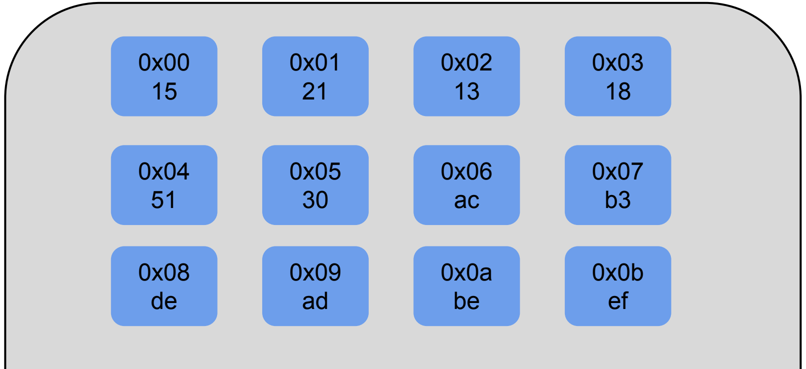


15	21	13	18				
51	30	ac	b3				

Jack2.trace

L 8,4 M

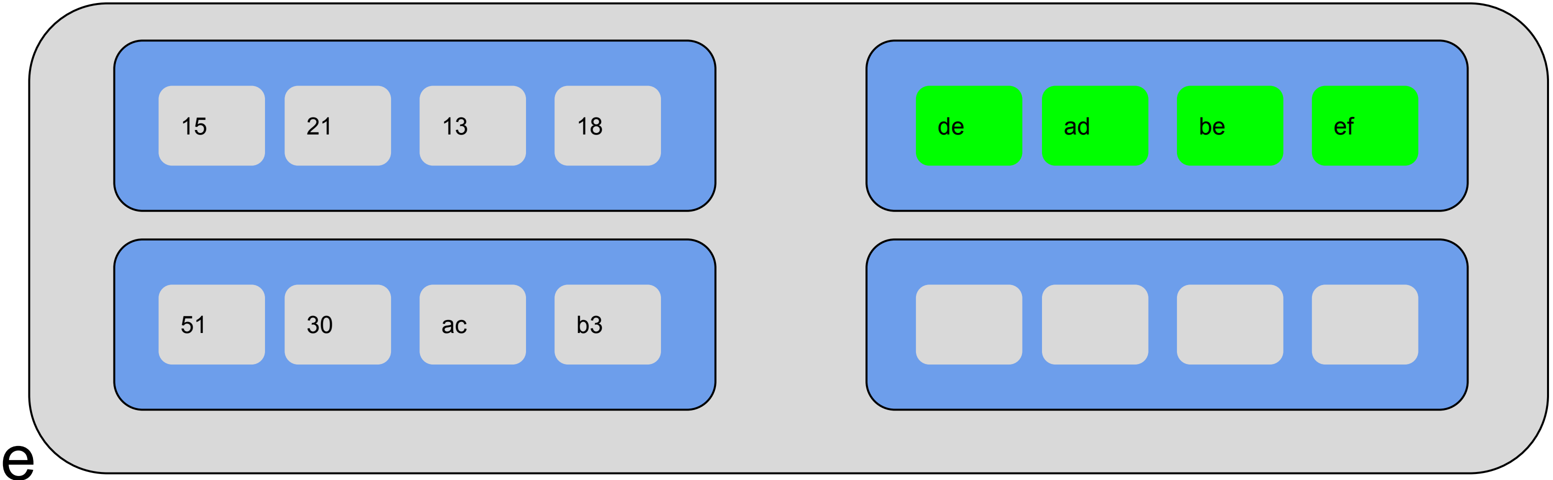
Memory



0x00 15	0x01 21	0x02 13	0x03 18
0x04 51	0x05 30	0x06 ac	0x07 b3
0x08 de	0x09 ad	0x0a be	0x0b ef

What would happen if we loaded from memory address 0x08?

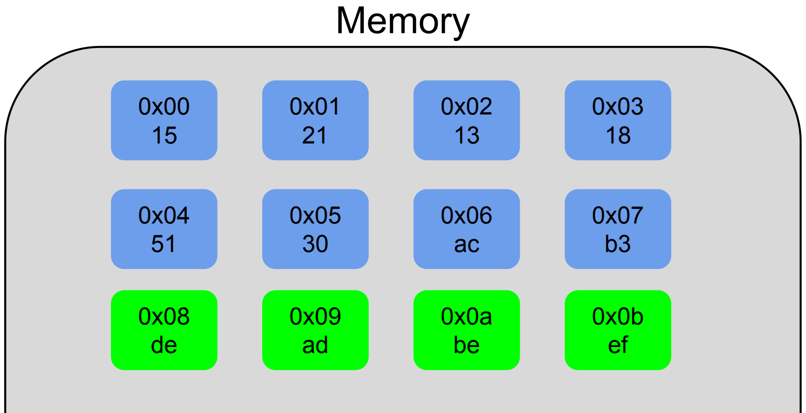
Example Trace



15	21	13	18	de	ad	be	ef
51	30	ac	b3				

Jack2.trace

L 8,4 M



Memory

0x00 15	0x01 21	0x02 13	0x03 18
0x04 51	0x05 30	0x06 ac	0x07 b3
0x08 de	0x09 ad	0x0a be	0x0b ef

What would happen if we loaded from memory address 0x08?

Blocking

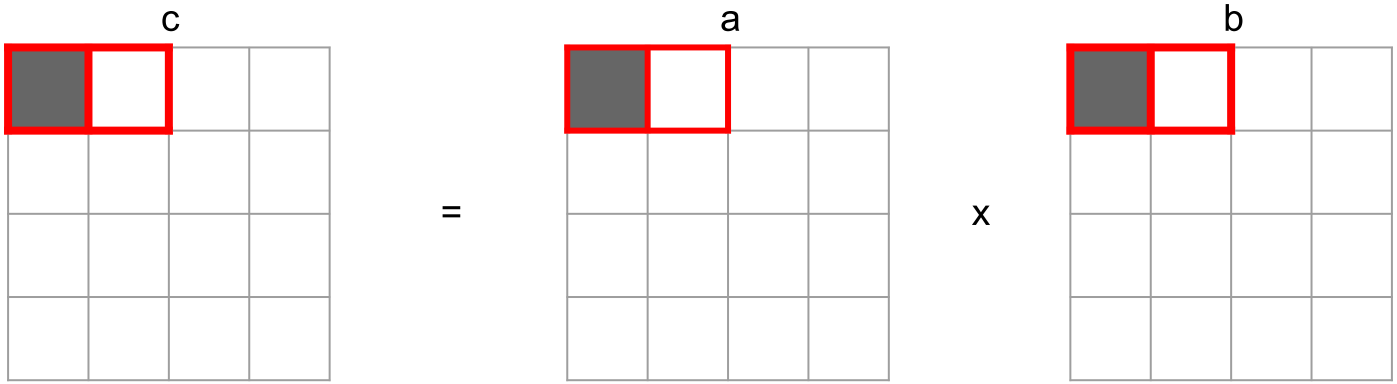
Example: Matrix Multiplication

```
/* 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];
```

Let’s step through this to see what’s actually happening

Example: Matrix Multiplication

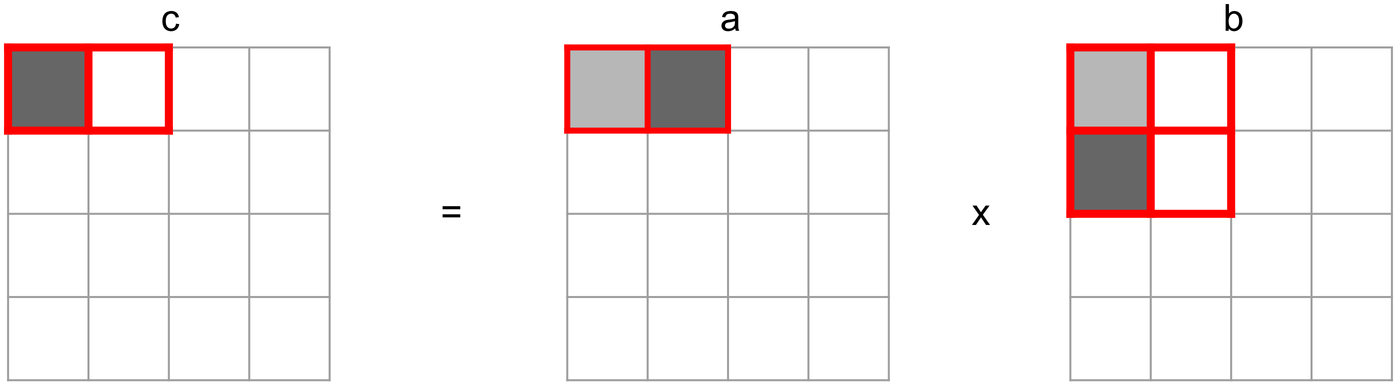
- Assume a tiny cache with 4 lines of 8 bytes (2 ints)
  - S = 1, E = 4, B = 8
- Let’s see what happens if we don’t use blocking



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]

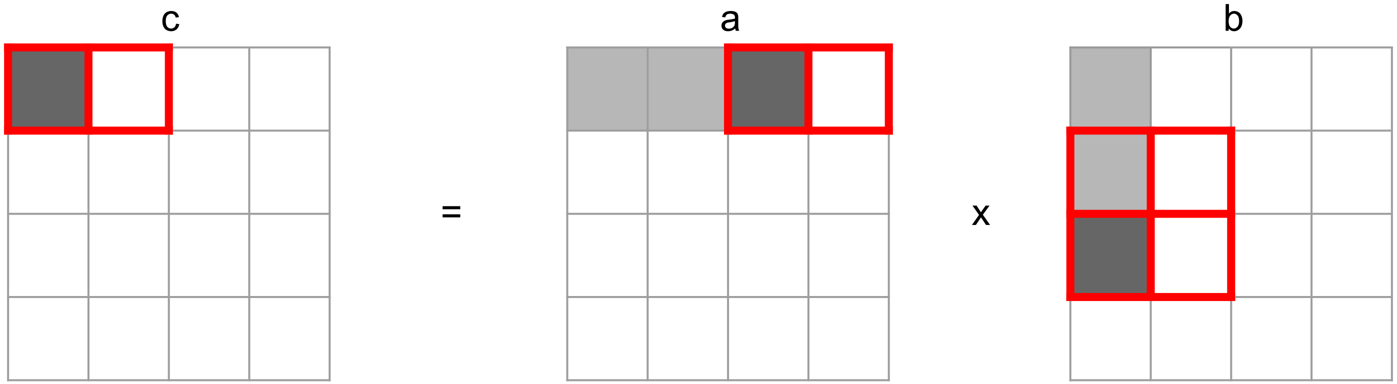
Key:

Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]

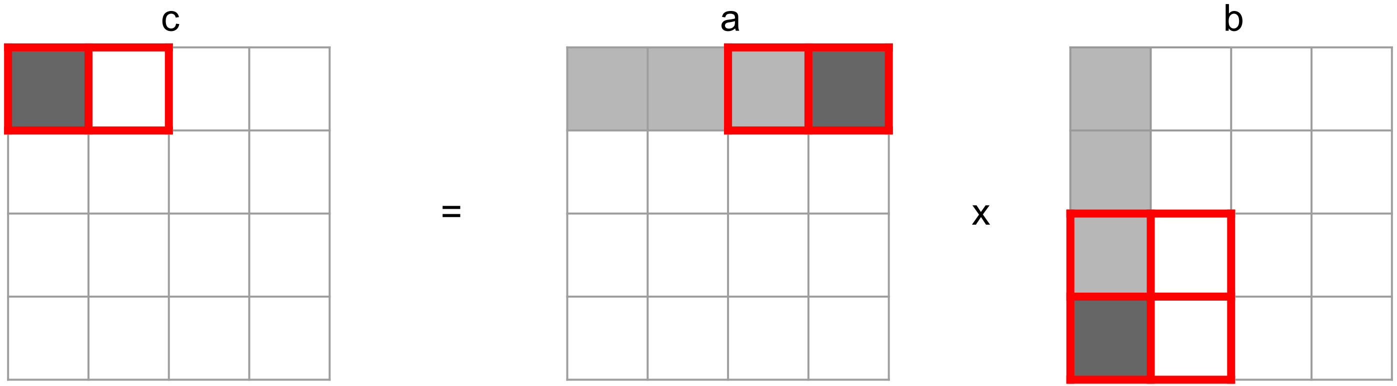
Key:
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	0	2	c[0][0] += a[0][2] * b[2][0]

Key:

Grey = accessed
Dark grey = currently accessing
Red border = in cache



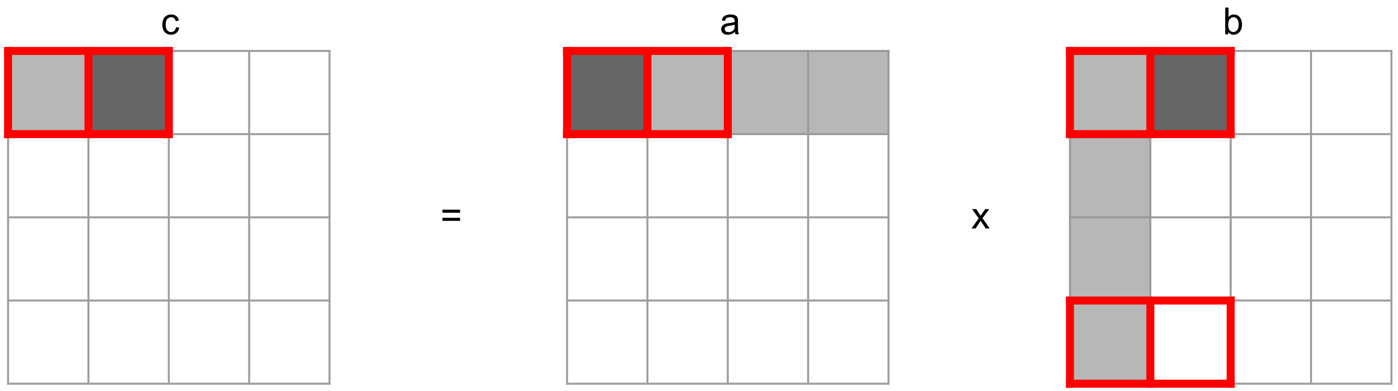
iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	0	2	c[0][0] += a[0][2] * b[2][0]
3	0	0	3	c[0][0] += a[0][3] * b[3][0]

Key:

Grey = accessed

Dark grey = currently accessing

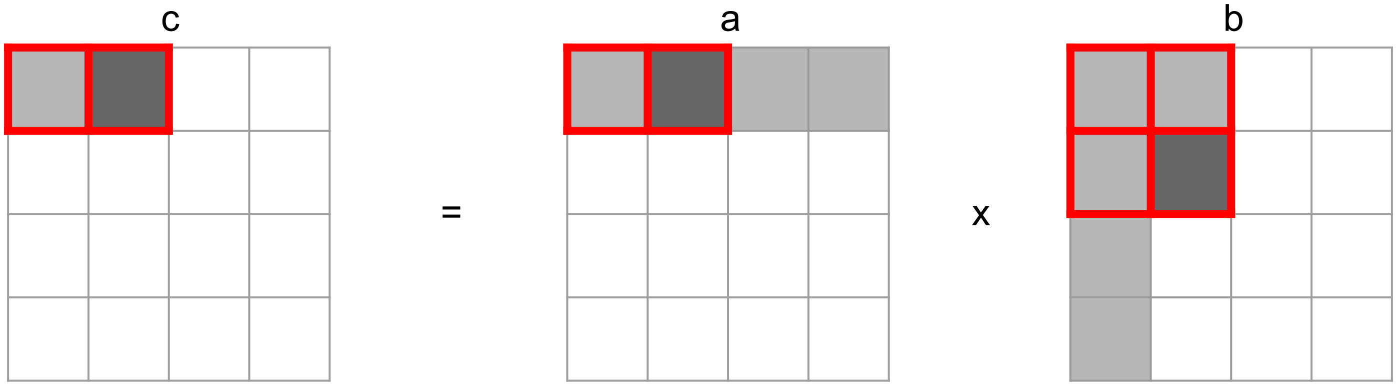
Red border = in cache



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	0	2	c[0][0] += a[0][2] * b[2][0]
3	0	0	3	c[0][0] += a[0][3] * b[3][0]
4	0	1	0	c[0][1] += a[0][0] * b[0][1]

Key:

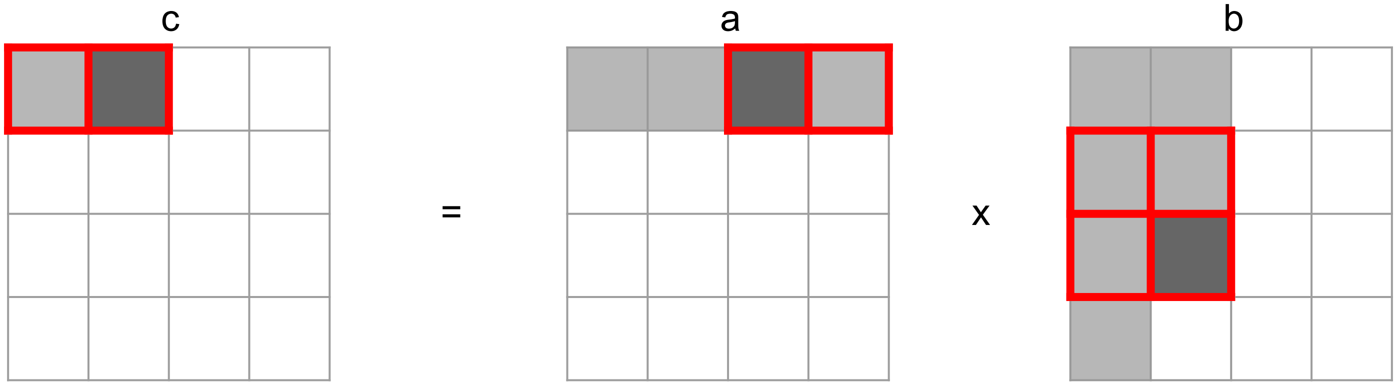
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	0	2	c[0][0] += a[0][2] * b[2][0]
3	0	0	3	c[0][0] += a[0][3] * b[3][0]
4	0	1	0	c[0][1] += a[0][0] * b[0][1]
5	0	1	1	c[0][1] += a[0][1] * b[1][1]

Key:

Grey = accessed
Dark grey = currently accessing
Red border = in cache



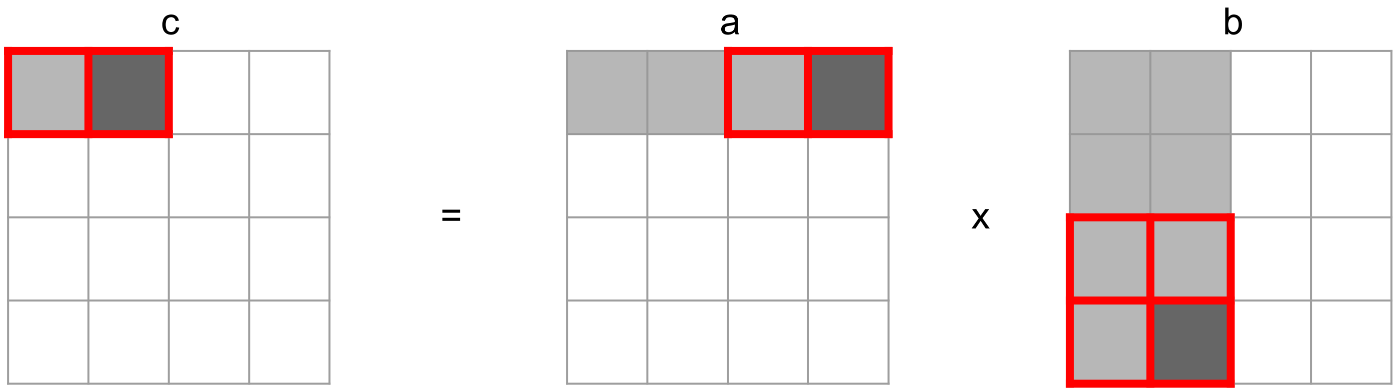
iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	0	2	c[0][0] += a[0][2] * b[2][0]
3	0	0	3	c[0][0] += a[0][3] * b[3][0]
4	0	1	0	c[0][1] += a[0][0] * b[0][1]
5	0	1	1	c[0][1] += a[0][1] * b[1][1]
6	0	1	2	c[0][1] += a[0][2] * b[2][1]

Key:

Grey = accessed

Dark grey = currently accessing

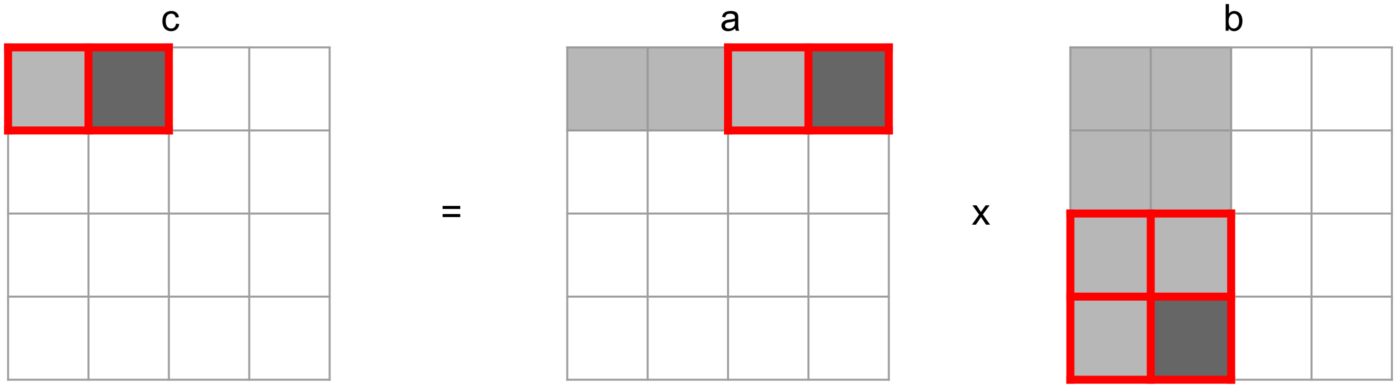
Red border = in cache



iter	**i**	**j**	**k**	**operation**
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	0	2	c[0][0] += a[0][2] * b[2][0]
3	0	0	3	c[0][0] += a[0][3] * b[3][0]
4	0	1	0	c[0][1] += a[0][0] * b[0][1]
5	0	1	1	c[0][1] += a[0][1] * b[1][1]
6	0	1	2	c[0][1] += a[0][2] * b[2][1]
7	0	1	3	c[0][1] += a[0][3] * b[3][1]

Key:

Grey = accessed
Dark grey = currently accessing
Red border = in cache

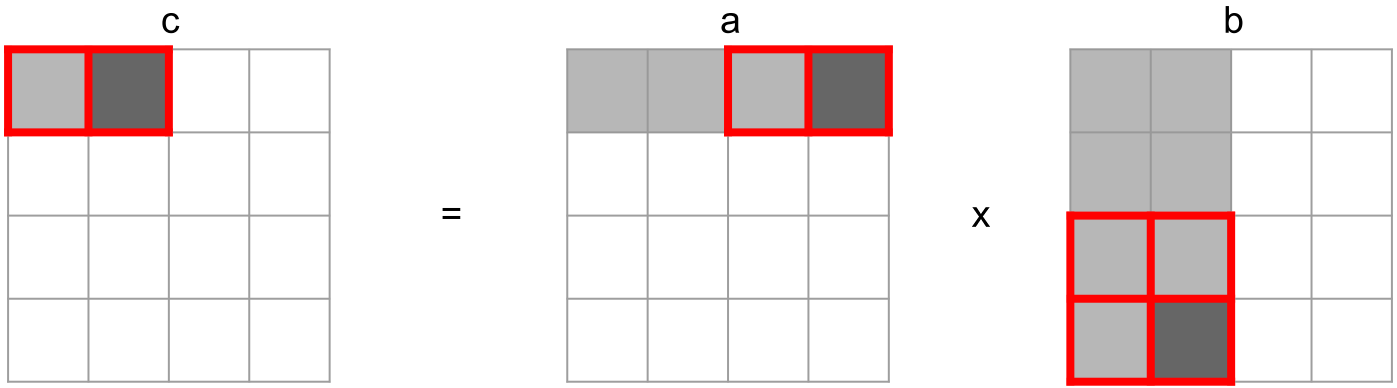


iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	0	2	c[0][0] += a[0][2] * b[2][0]
3	0	0	3	c[0][0] += a[0][3] * b[3][0]
4	0	1	0	c[0][1] += a[0][0] * b[0][1]
5	0	1	1	c[0][1] += a[0][1] * b[1][1]
6	0	1	2	c[0][1] += a[0][2] * b[2][1]
7	0	1	3	c[0][1] += a[0][3] * b[3][1]

Key:

Grey = accessed
Dark grey = currently accessing
Red border = in cache

What is the miss rate of a?



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	0	2	c[0][0] += a[0][2] * b[2][0]
3	0	0	3	c[0][0] += a[0][3] * b[3][0]
4	0	1	0	c[0][1] += a[0][0] * b[0][1]
5	0	1	1	c[0][1] += a[0][1] * b[1][1]
6	0	1	2	c[0][1] += a[0][2] * b[2][1]
7	0	1	3	c[0][1] += a[0][3] * b[3][1]

Key:

Grey = accessed
Dark grey = currently accessing
Red border = in cache

What is the miss rate of a?

What is the miss rate of b?

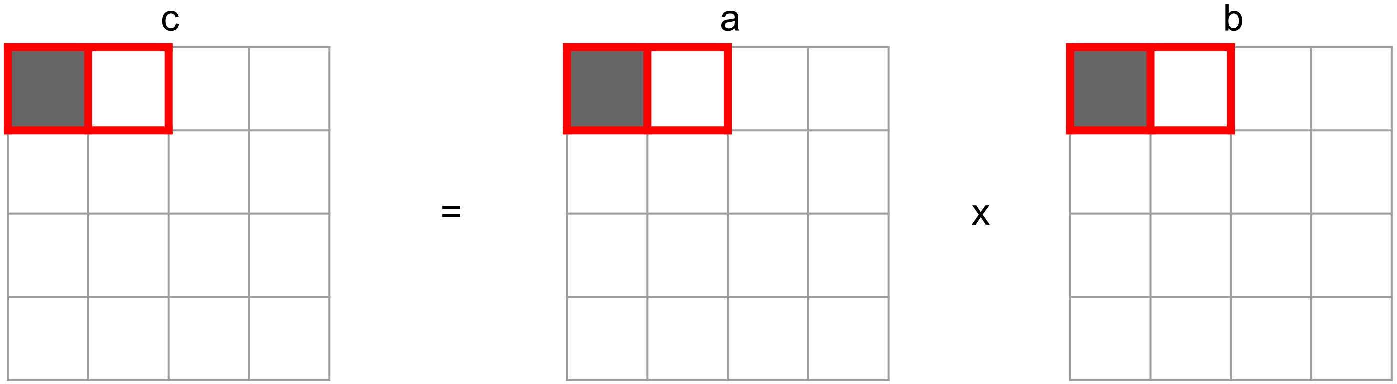
Example: Matrix Multiplication (blocking)

```
/* 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 step through this to see what’s actually happening

Example: Matrix Multiplication (blocking)

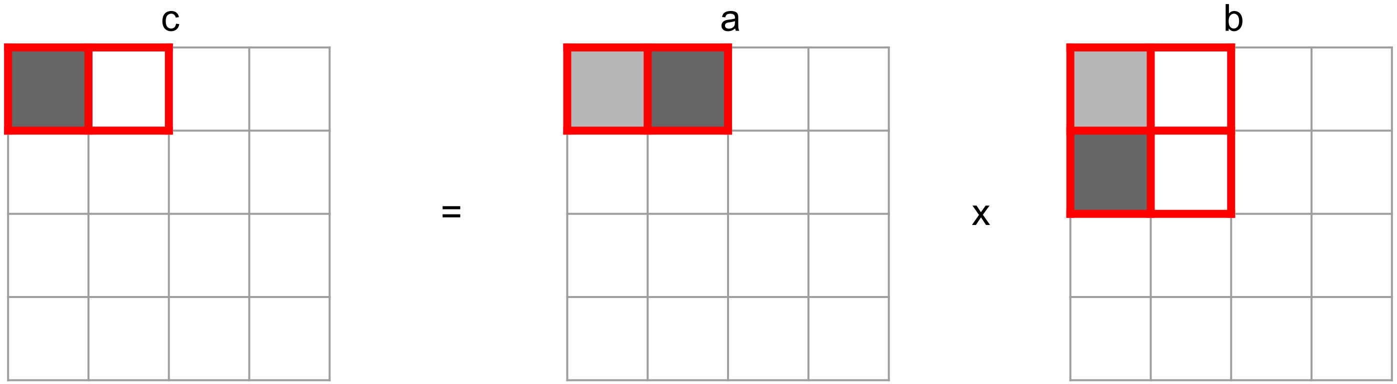
- Assume a tiny cache with 4 lines of 8 bytes (2 ints)
  - S = 1, E = 4, B = 8
- Let’s see what happens if we now use blocking



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]

Key:

Grey = accessed
Dark grey = currently accessing
Red border = in cache



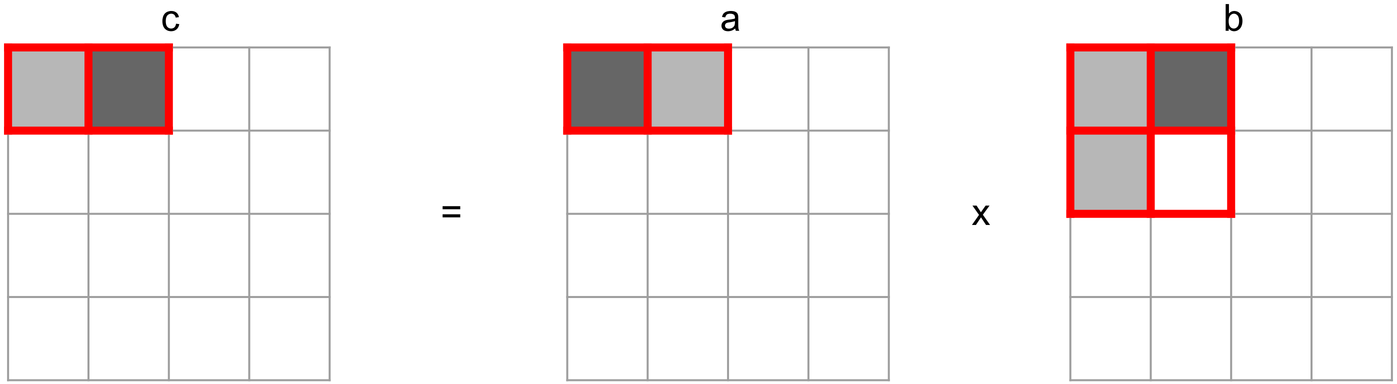
iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]

Key:

Grey = accessed

Dark grey = currently accessing

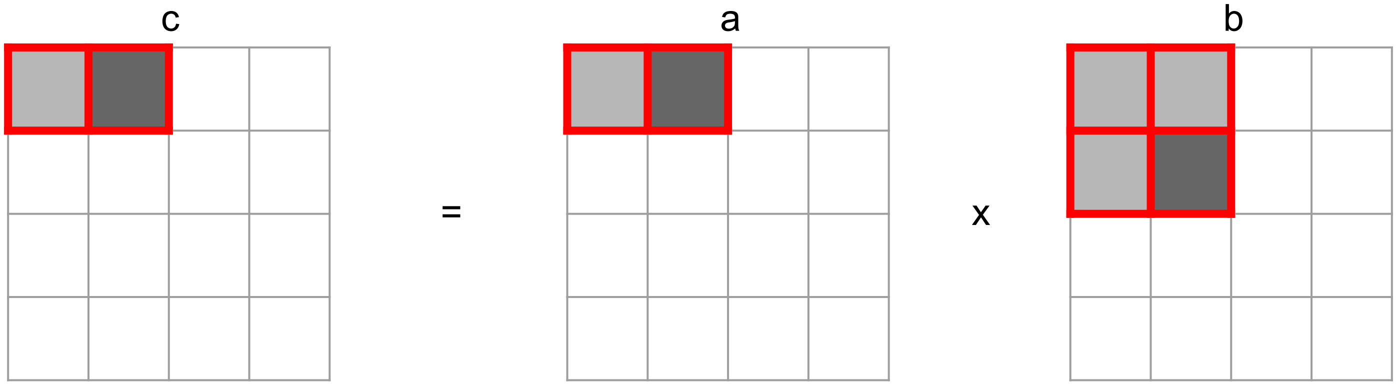
Red border = in cache



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]

Key:

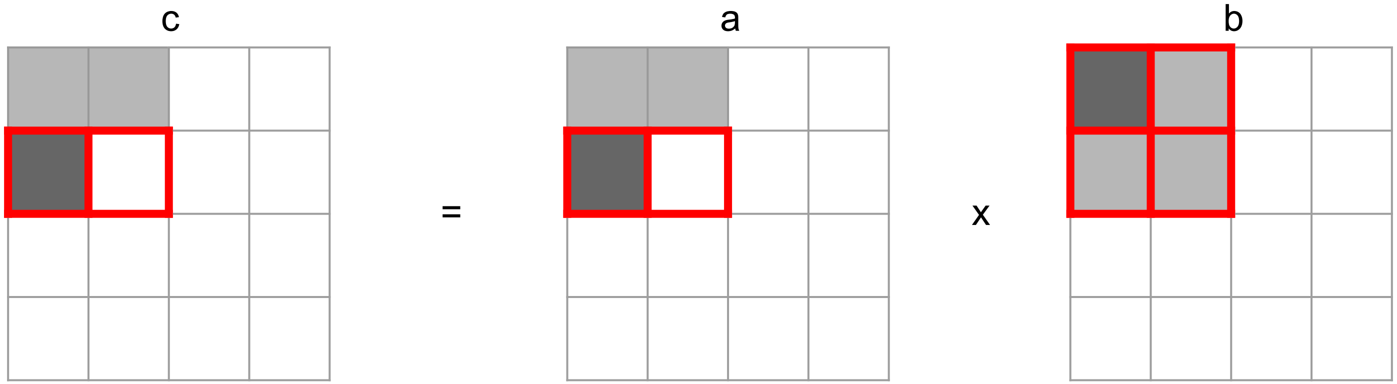
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]

Key:

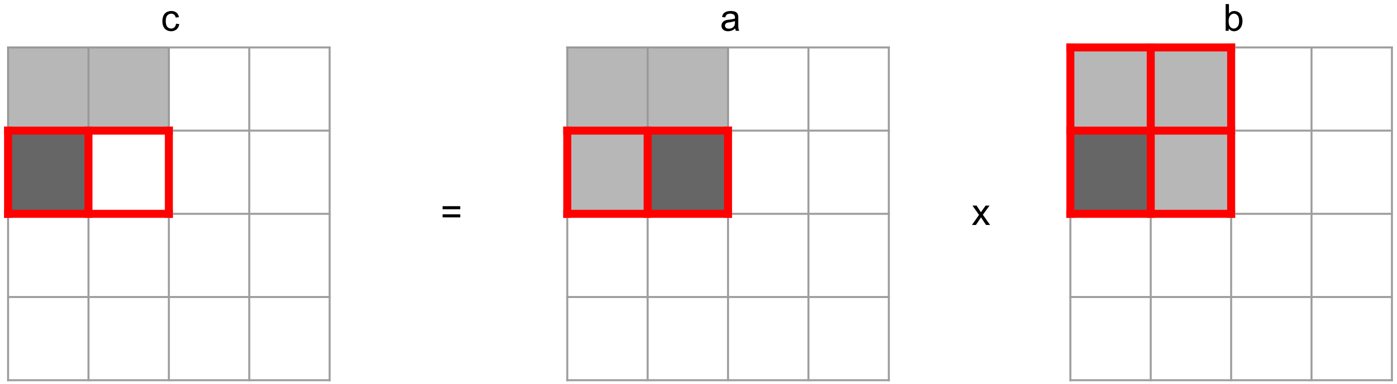
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]

Key:

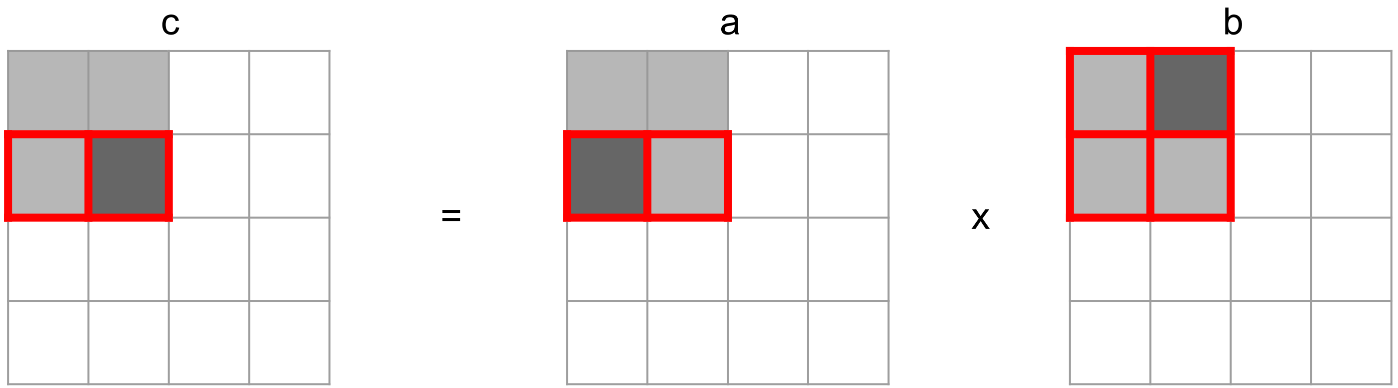
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]
5	1	0	1	c[1][0] += a[1][1] * b[1][0]

Key:

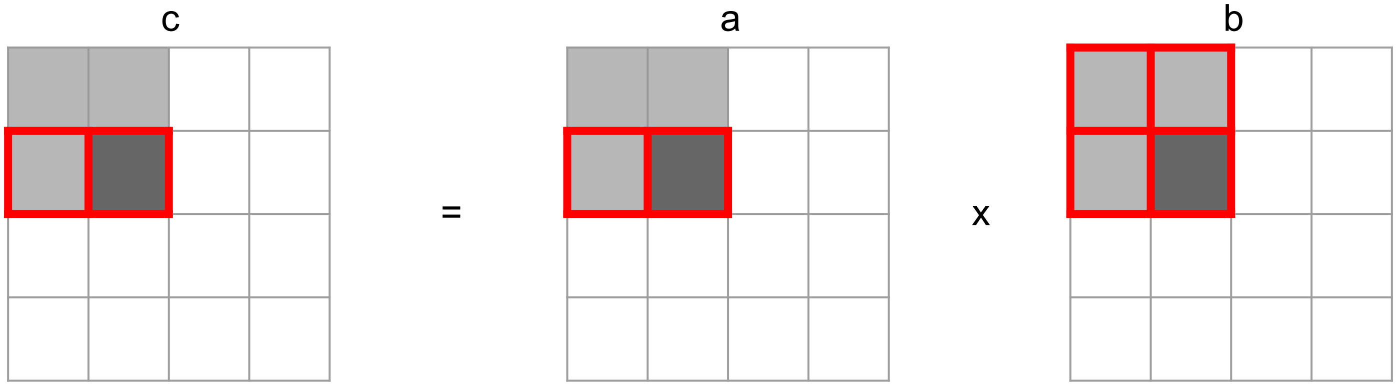
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]
5	1	0	1	c[1][0] += a[1][1] * b[1][0]
6	1	1	0	c[1][1] += a[1][0] * b[0][1]

Key:

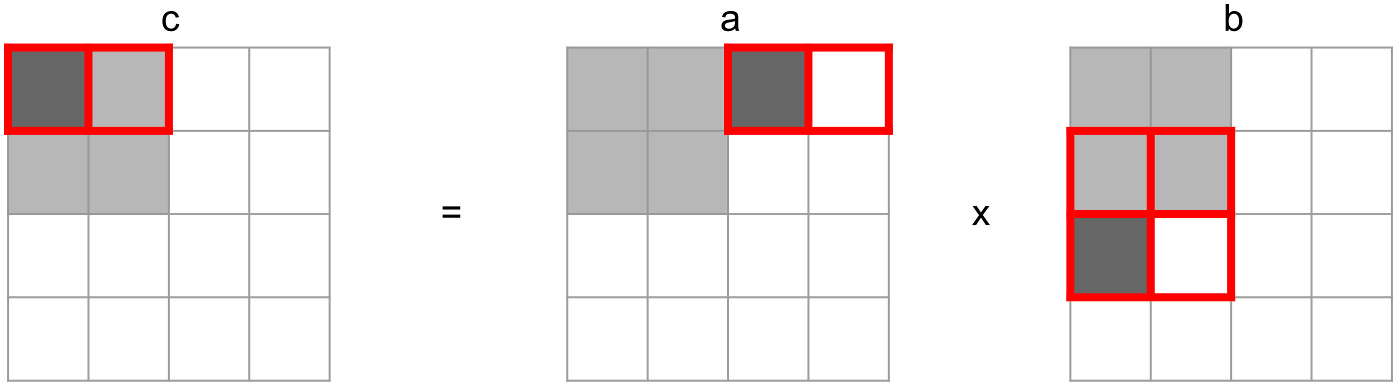
Grey = accessed
Dark grey = currently accessing
Red border = in cache



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]
5	1	0	1	c[1][0] += a[1][1] * b[1][0]
6	1	1	0	c[1][1] += a[1][0] * b[0][1]
7	1	1	1	c[1][1] += a[1][1] * b[1][1]

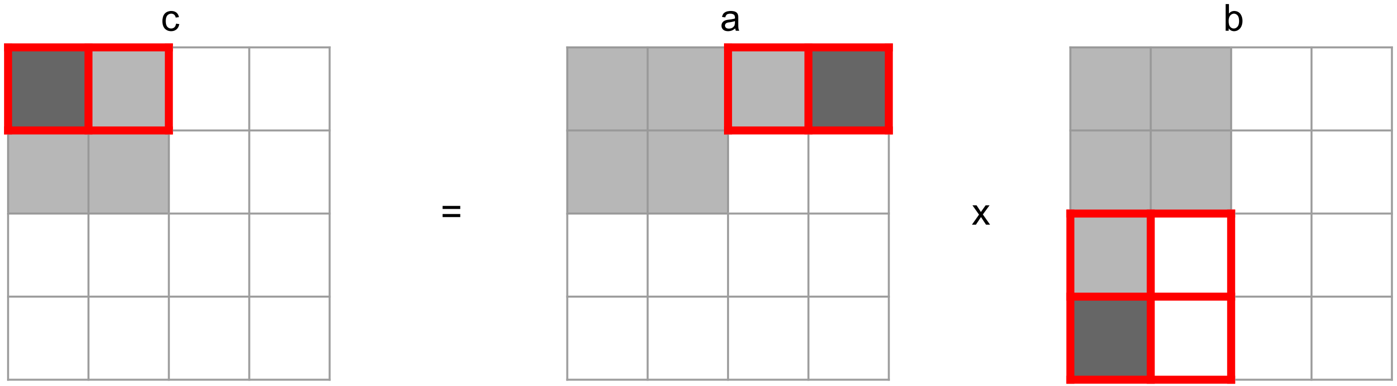
Key:

Grey = accessed
Dark grey = currently accessing
Red border = in cache



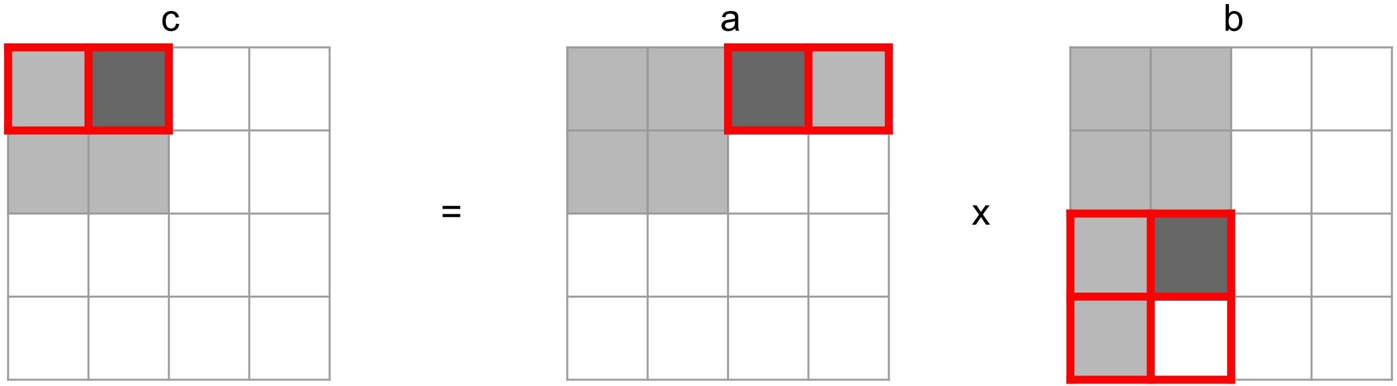
iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]
5	1	0	1	c[1][0] += a[1][1] * b[1][0]
6	1	1	0	c[1][1] += a[1][0] * b[0][1]
7	1	1	1	c[1][1] += a[1][1] * b[1][1]

iter	i	j	k	operation
8	0	0	2	c[0][0] += a[0][2] * b[2][0]



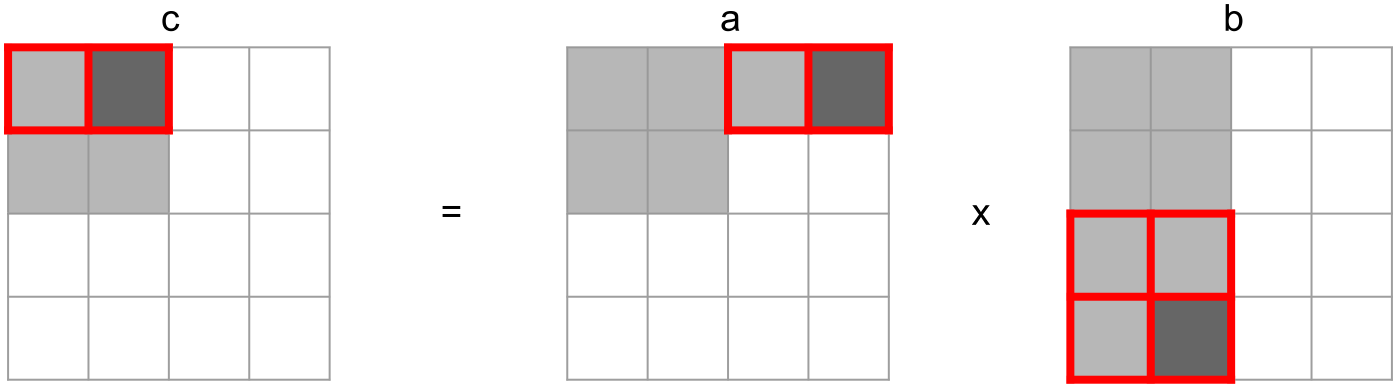
iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]
5	1	0	1	c[1][0] += a[1][1] * b[1][0]
6	1	1	0	c[1][1] += a[1][0] * b[0][1]
7	1	1	1	c[1][1] += a[1][1] * b[1][1]

iter	i	j	k	operation
8	0	0	2	c[0][0] += a[0][2] * b[2][0]
9	0	0	3	c[0][0] += a[0][3] * b[3][0]



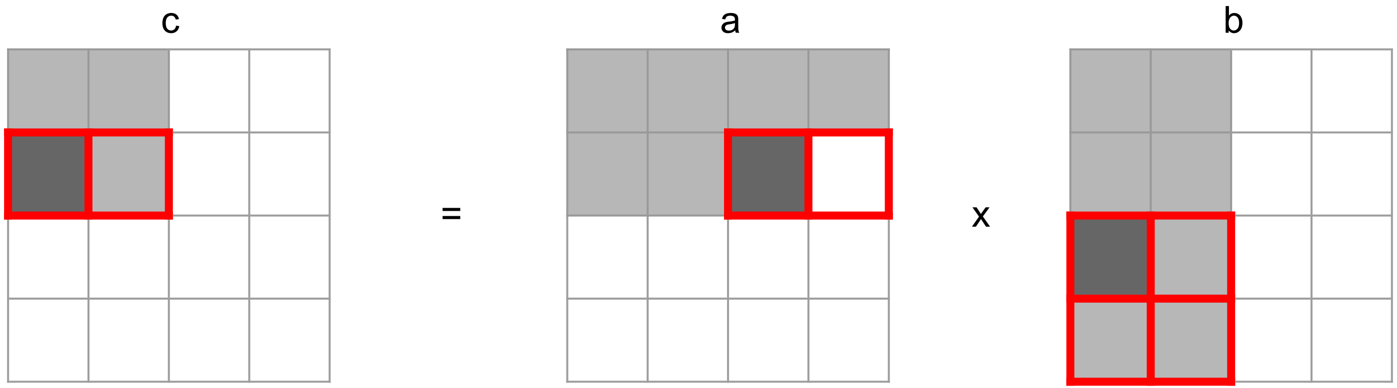
iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]
5	1	0	1	c[1][0] += a[1][1] * b[1][0]
6	1	1	0	c[1][1] += a[1][0] * b[0][1]
7	1	1	1	c[1][1] += a[1][1] * b[1][1]

iter	i	j	k	operation
8	0	0	2	c[0][0] += a[0][2] * b[2][0]
9	0	0	3	c[0][0] += a[0][3] * b[3][0]
10	0	1	2	c[0][1] += a[0][2] * b[2][1]



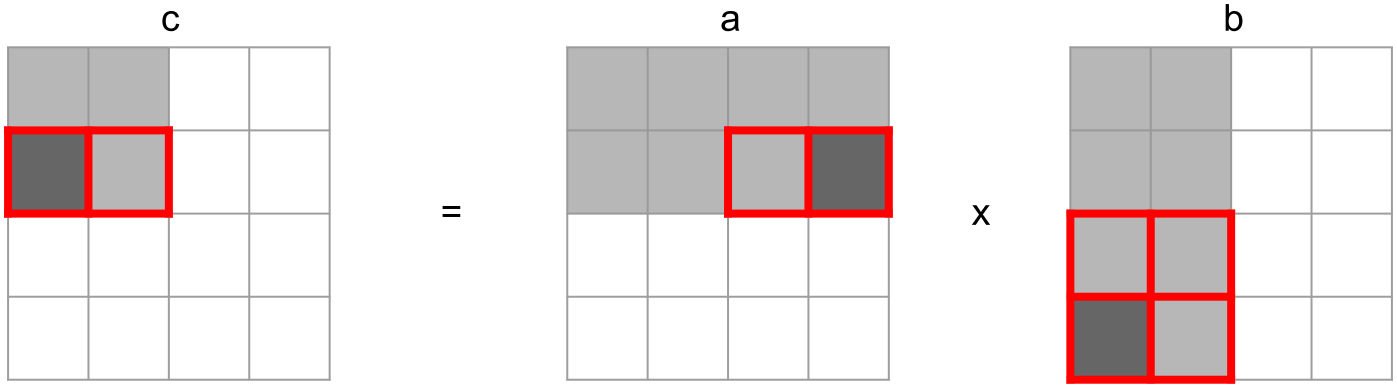
iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]
5	1	0	1	c[1][0] += a[1][1] * b[1][0]
6	1	1	0	c[1][1] += a[1][0] * b[0][1]
7	1	1	1	c[1][1] += a[1][1] * b[1][1]

iter	i	j	k	operation
8	0	0	2	c[0][0] += a[0][2] * b[2][0]
9	0	0	3	c[0][0] += a[0][3] * b[3][0]
10	0	1	2	c[0][1] += a[0][2] * b[2][1]
11	0	1	3	c[0][1] += a[0][3] * b[3][1]



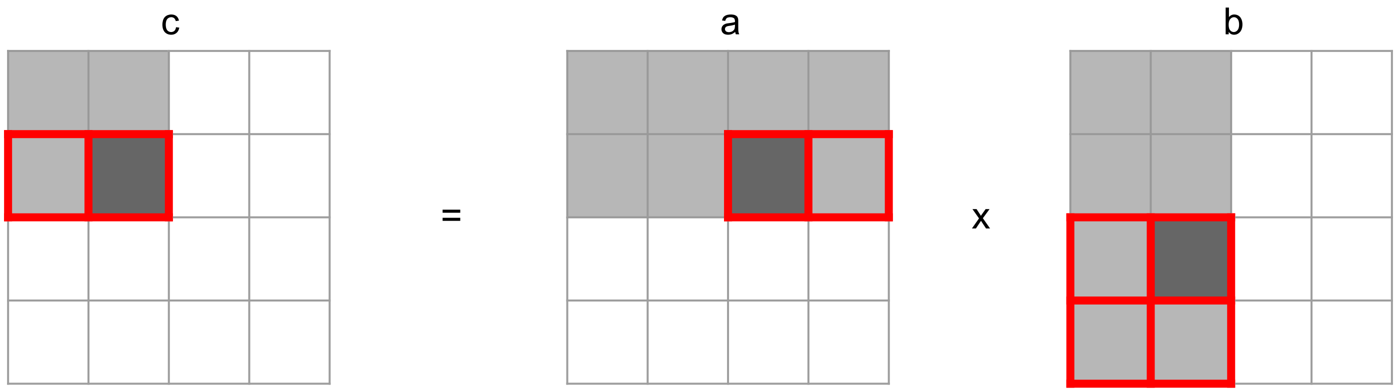
iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]
5	1	0	1	c[1][0] += a[1][1] * b[1][0]
6	1	1	0	c[1][1] += a[1][0] * b[0][1]
7	1	1	1	c[1][1] += a[1][1] * b[1][1]

iter	i	j	k	operation
8	0	0	2	c[0][0] += a[0][2] * b[2][0]
9	0	0	3	c[0][0] += a[0][3] * b[3][0]
10	0	1	2	c[0][1] += a[0][2] * b[2][1]
11	0	1	3	c[0][1] += a[0][3] * b[3][1]
12	1	0	2	c[1][0] += a[1][2] * b[2][0]



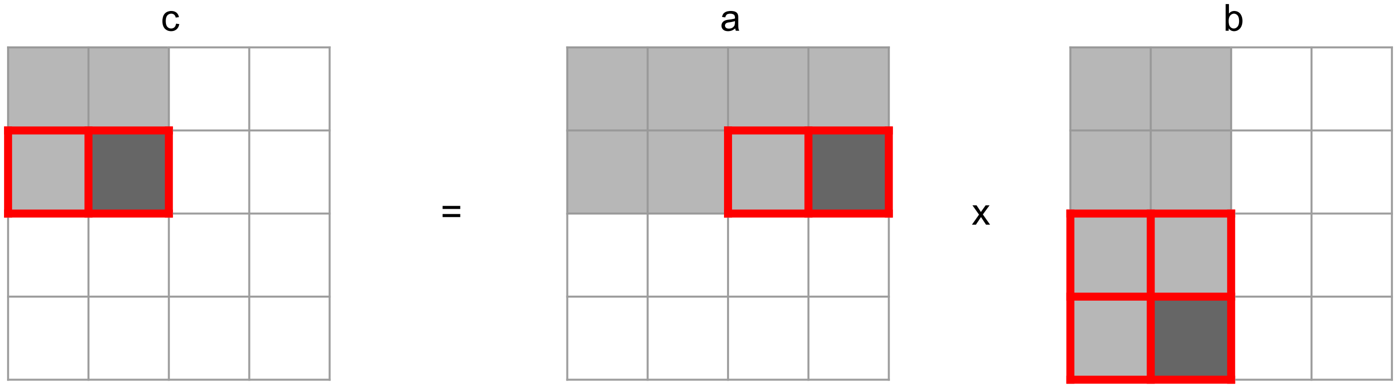
iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]
5	1	0	1	c[1][0] += a[1][1] * b[1][0]
6	1	1	0	c[1][1] += a[1][0] * b[0][1]
7	1	1	1	c[1][1] += a[1][1] * b[1][1]

iter	i	j	k	operation
8	0	0	2	c[0][0] += a[0][2] * b[2][0]
9	0	0	3	c[0][0] += a[0][3] * b[3][0]
10	0	1	2	c[0][1] += a[0][2] * b[2][1]
11	0	1	3	c[0][1] += a[0][3] * b[3][1]
12	1	0	2	c[1][0] += a[1][2] * b[2][0]
13	1	0	3	c[1][0] += a[1][3] * b[3][0]

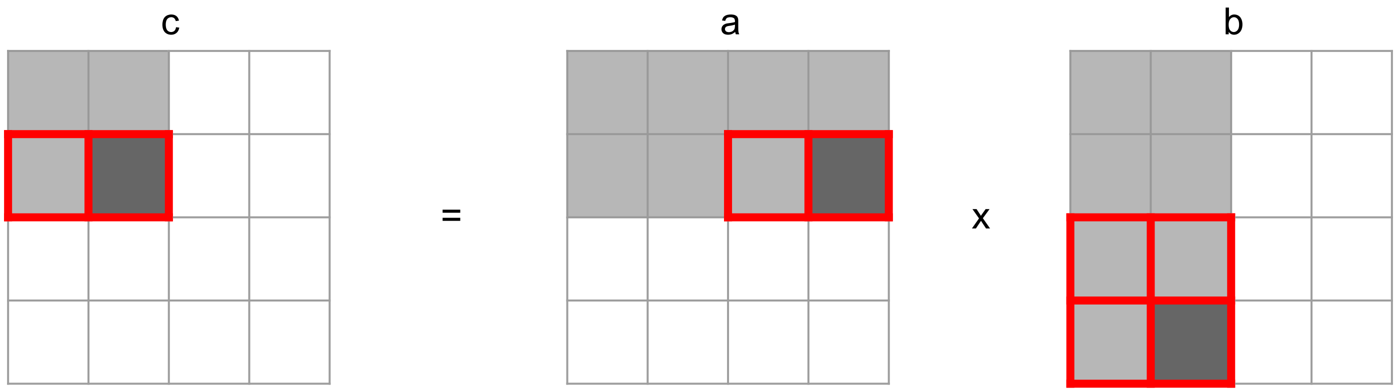


iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]
5	1	0	1	c[1][0] += a[1][1] * b[1][0]
6	1	1	0	c[1][1] += a[1][0] * b[0][1]
7	1	1	1	c[1][1] += a[1][1] * b[1][1]

iter	i	j	k	operation
8	0	0	2	c[0][0] += a[0][2] * b[2][0]
9	0	0	3	c[0][0] += a[0][3] * b[3][0]
10	0	1	2	c[0][1] += a[0][2] * b[2][1]
11	0	1	3	c[0][1] += a[0][3] * b[3][1]
12	1	0	2	c[1][0] += a[1][2] * b[2][0]
13	1	0	3	c[1][0] += a[1][3] * b[3][0]
14	1	1	2	c[1][1] += a[1][2] * b[2][1]



iter	i	j	k	operation	iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]	8	0	0	2	c[0][0] += a[0][2] * b[2][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]	9	0	0	3	c[0][0] += a[0][3] * b[3][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]	10	0	1	2	c[0][1] += a[0][2] * b[2][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]	11	0	1	3	c[0][1] += a[0][3] * b[3][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]	12	1	0	2	c[1][0] += a[1][2] * b[2][0]
5	1	0	1	c[1][0] += a[1][1] * b[1][0]	13	1	0	3	c[1][0] += a[1][3] * b[3][0]
6	1	1	0	c[1][1] += a[1][0] * b[0][1]	14	1	1	2	c[1][1] += a[1][2] * b[2][1]
7	1	1	1	c[1][1] += a[1][1] * b[1][1]	15	1	1	3	c[1][1] += a[1][3] * b[3][1]



iter	i	j	k	operation
0	0	0	0	c[0][0] += a[0][0] * b[0][0]
1	0	0	1	c[0][0] += a[0][1] * b[1][0]
2	0	1	0	c[0][1] += a[0][0] * b[0][1]
3	0	1	1	c[0][1] += a[0][1] * b[1][1]
4	1	0	0	c[1][0] += a[1][0] * b[0][0]
5	1	0	1	c[1][0] += a[1][1] * b[1][0]
6	1	1	0	c[1][1] += a[1][0] * b[0][1]
7	1	1	1	c[1][1] += a[1][1] * b[1][1]
8	0	0	2	c[0][0] += a[0][2] * b[2][0]
9	0	0	3	c[0][0] += a[0][3] * b[3][0]
10	0	1	2	c[0][1] += a[0][2] * b[2][1]
11	0	1	3	c[0][1] += a[0][3] * b[3][1]
12	1	0		0]
13	1	0		0]
14	1	1	2	c[1][1] += a[1][2] * b[2][1]
15	1	1		1]

What is the miss rate of a?

What is the miss rate of b?

If you get stuck…

- Reread the writeup
- Look at CS:APP Chapter 6
- Review lecture notes (http://cs.cmu.edu/~213)
- Come to Office Hours
- Post private question on Piazza
- `man malloc`, `man valgrind`, `man gdb`

Cache Lab Tips!

- Review cache and memory lectures
    - Ask if you don’t understand something
- Start early, this can be a challenging lab!
- Don’t get discouraged!
    - If you try something that doesn't work, take a well deserved break, and then try again
- Good luck!

Practice Problems

Class Question / Discussions

- We’ll work through a series of questions
- Write down your answer for each question
- You can discuss with your classmates

What Type of Locality?

- The following function exhibits which type of locality? Consider *only* array accesses.

```
void who(int *arr, int size) {
  for (int i = 0; i < size-1; ++i)
    arr[i] = arr[i+1];
}
```

A. Spatial

B. Temporal

C. Both A and B

D. Neither A nor B

What Type of Locality?

- The following function exhibits which type of locality? Consider *only* array accesses.

```
void who(int *arr, int size) {
  for (int i = 0; i < size-1; ++i)
    arr[i] = arr[i+1];
}
```

A. Spatial

B. Temporal

[✓ **C.**] Both A and B

D. Neither A nor B

What Type of Locality?

- The following function exhibits which type of locality? Consider *only* array accesses.

```
void coo(int *arr, int size) {
  for (int i = size-2; i >= 0; --i)
    arr[i] = arr[i+1];
}
```

A. Spatial

B. Temporal

C. Both A and B

D. Neither A nor B

What Type of Locality?

- The following function exhibits which type of locality? Consider *only* array accesses.

```
void coo(int *arr, int size) {
  for (int i = size-2; i >= 0; --i)
    arr[i] = arr[i+1];
}
```

A. Spatial

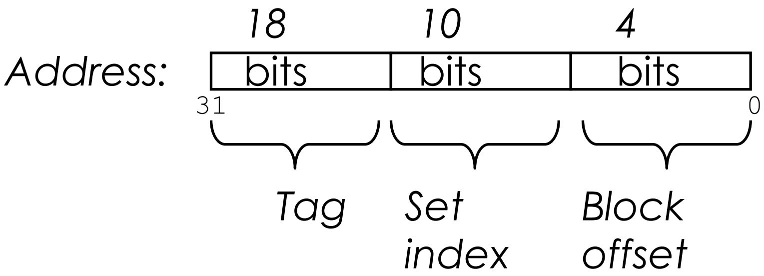
B. Temporal

[✓ C.] Both A and B

D. Neither A nor B

Calculating Cache Parameters

- Given the following address partition, how many `int` values will fit in a single data block?



Address:

18 bits | *10* bits | *4* bits

31 … 0

Tag | *Set index* | *Block offset*

# of int in block

A. 0

B. 1

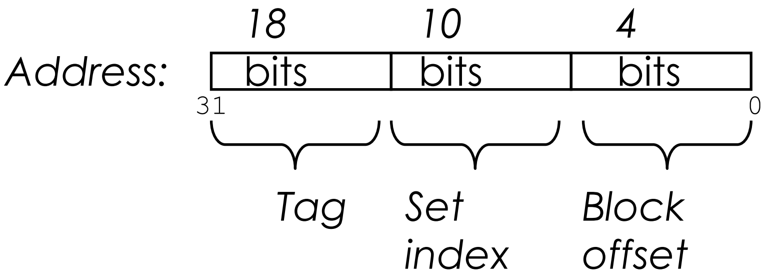
C. 2

D. 4

E. Unknown: We need more info

Calculating Cache Parameters

- Given the following address partition, how many `int` values will fit in a single data block?



Address:

18 bits	10 bits	4 bits
Tag	Set index	Block offset

31 … 0

\# of int in block

A. 0

B. 1

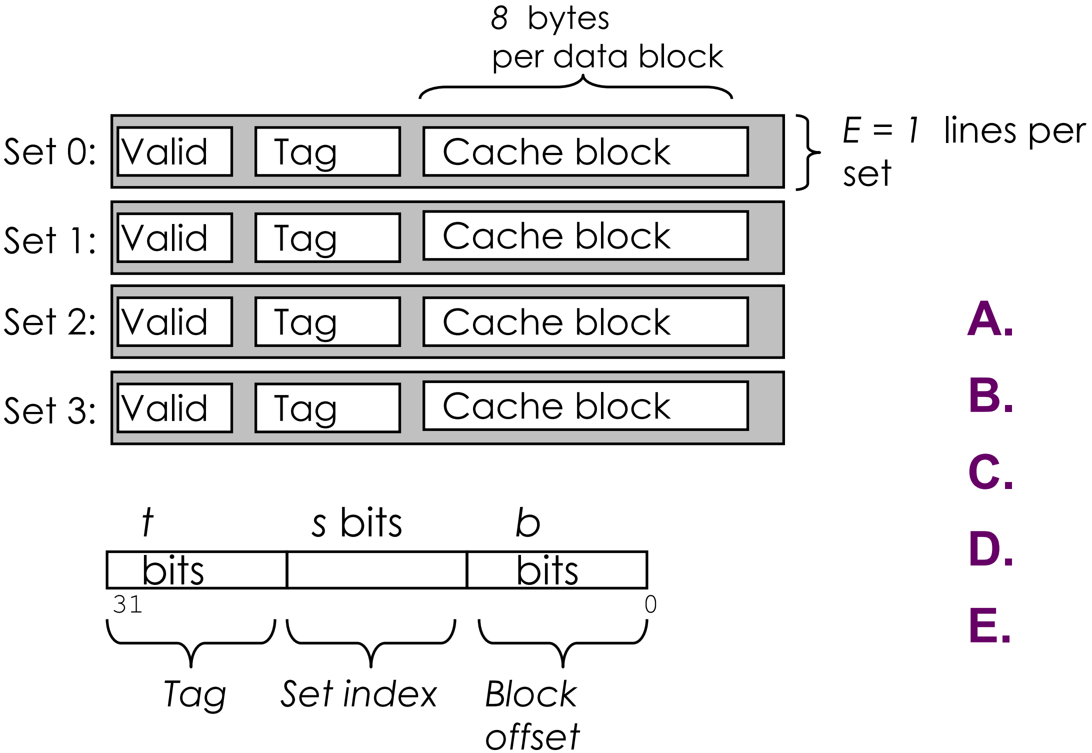
C. 2

[✓ **D.**] 4

E. Unknown: We need more info

Direct-Mapped Cache Example

- Assuming a 32-bit address (i.e. m=32), how many bits are used for tag (t), set index (s), and block offset (b).



8 bytes per data block

Set 0: Valid | Tag | Cache block

Set 1: Valid | Tag | Cache block

Set 2: Valid | Tag | Cache block

Set 3: Valid | Tag | Cache block

E = 1 lines per set

t bits | *s* bits | *b* bits

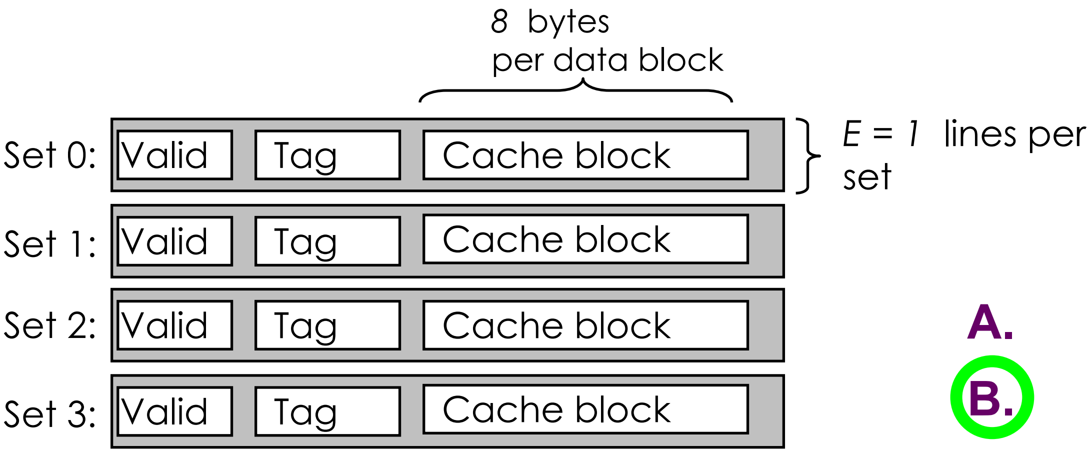
31 … 0

Tag | *Set index* | *Block offset*

	t	s	b
A.	1	2	3
B.	27	2	3
C.	25	4	3
D.	1	4	8
E.	20	4	8

Direct-Mapped Cache Example

- Assuming a 32-bit address (i.e. m=32), how many bits are used for tag (t), set index (s), and block offset (b).



8 bytes per data block

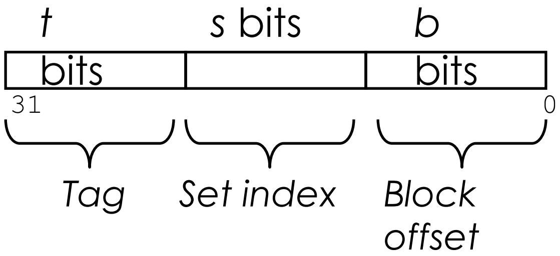
Set 0: Valid Tag Cache block

Set 1: Valid Tag Cache block

Set 2: Valid Tag Cache block

Set 3: Valid Tag Cache block

E = 1 lines per set



t bits

s bits

b bits

31

0

Tag

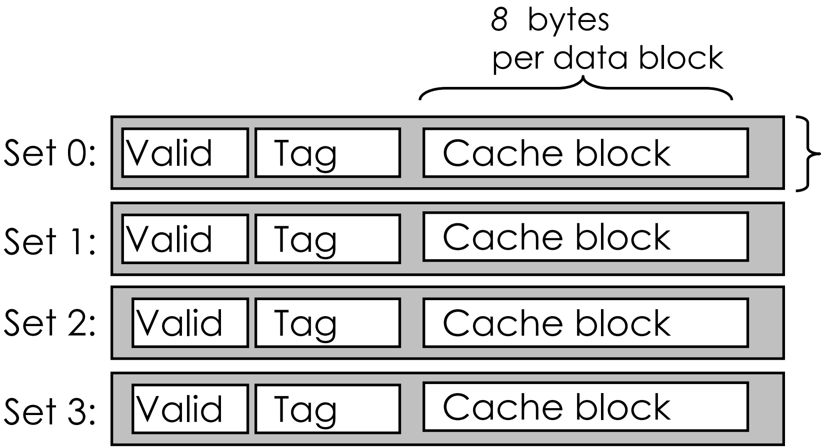
Set index

Block offset

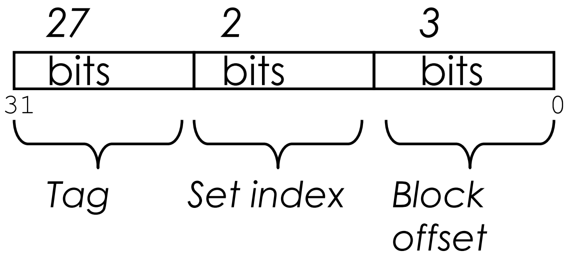
	t	s	b
A.	1	2	3
[✓ **B.**]	27	2	3
C.	25	4	3
D.	1	4	8
E.	20	4	8

Which Set Is it?

- Which set is the address **0xFA1C** located in?



E = 1 lines per set



Set # for 0xFA1C

A. 0

B. 1

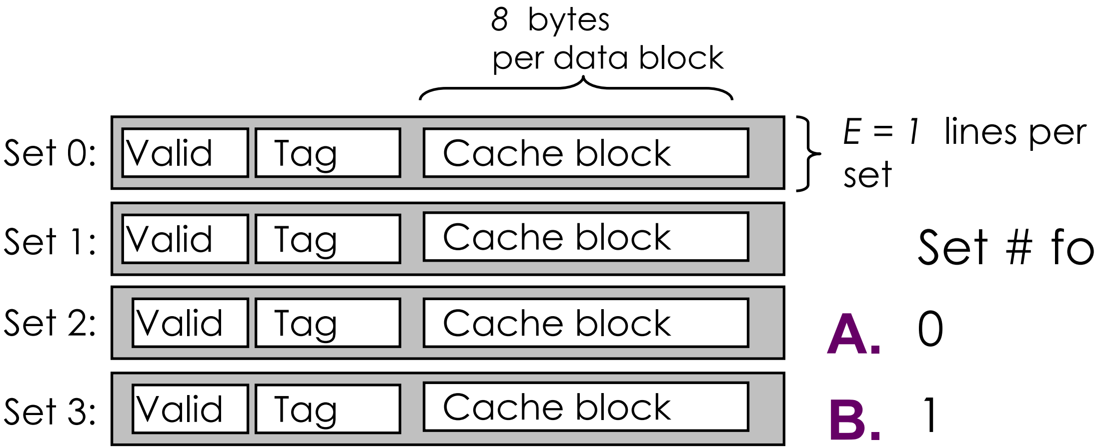
C. 2

D. 3

E. More than one of the above

Which Set Is it?

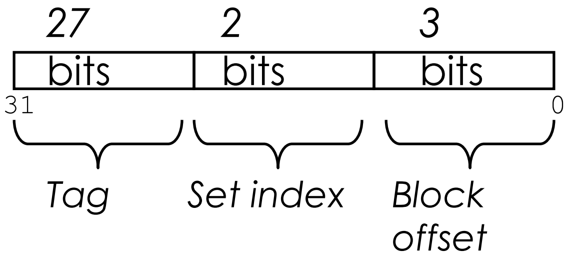
- Which set is the address **0xFA1C** located in?



8 bytes
per data block

Set 0: Valid | Tag | Cache block
Set 1: Valid | Tag | Cache block
Set 2: Valid | Tag | Cache block
Set 3: Valid | Tag | Cache block

E = 1 lines per set



27 bits | 2 bits | 3 bits

31 … 0

Tag | *Set index* | *Block offset*

Set # for 0xFA1C

A. 0

B. 1

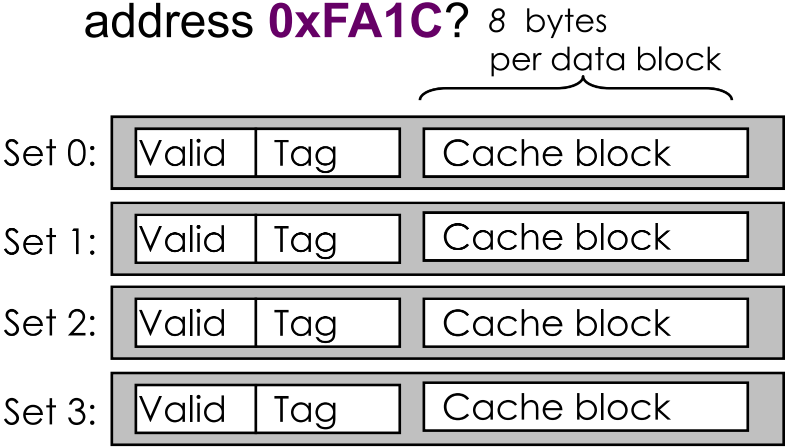
C. 2

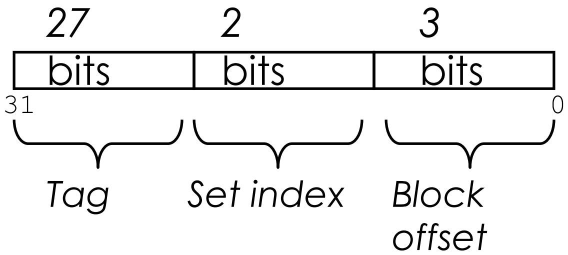
[✓ **D.**] 3

E. More than one of the above

Cache Block Range

- What range of addresses will be in the same block as address **0xFA1C**?





Addr. Range

A. 0xFA1C

B. 0xFA1C – 0xFA23

C. 0xFA1C – 0xFA1F

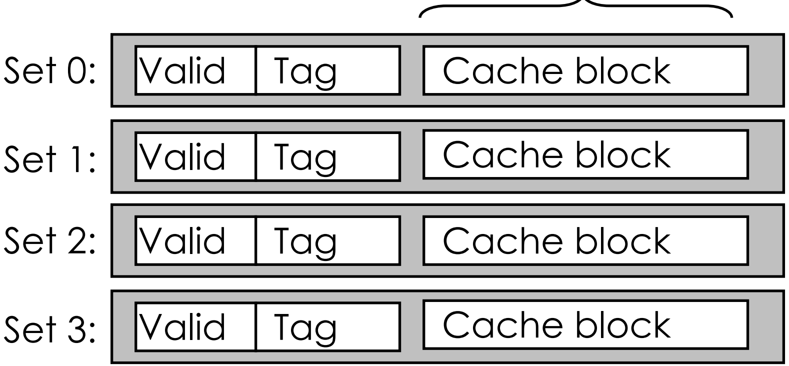
D. 0xFA18 – 0xFA1F

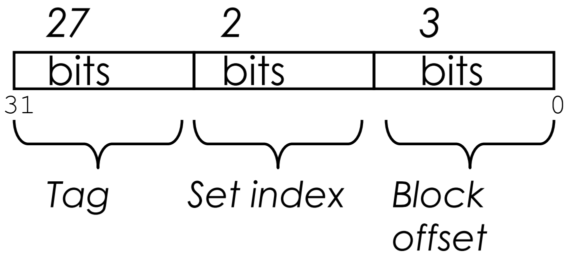
E. It depends on the access size (byte, word, etc)

Cache Block Range

- What range of addresses will be in the same block as address **0xFA1C**?

8 bytes per data block





Addr. Range

A. 0xFA1C

B. 0xFA1C – 0xFA23

C. 0xFA1C – 0xFA1F

[✓ **D.**] 0xFA18 – 0xFA1F

E. It depends on the access size (byte, word, etc)

Cache Misses

If N = 16, how many bytes does the loop access of a?

```
int foo(int* a, int N)
{
    int i;
    int sum = 0;
    for(i = 0; i < N; i++)
    {
        sum += a[i];
    }
    return sum;
}
```

	Accessed Bytes
A	4
B	16
C	64
D	256

Cache Misses

If N = 16, how many bytes does the loop access of a?

```
int foo(int* a, int N)
{
    int i;
    int sum = 0;
    for(i = 0; i < N; i++)
    {
        sum += a[i];
    }
    return sum;
}
```

	Accessed Bytes
A	4
B	16
[✓ **C**]	64
D	256

Cache Misses

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on **‘pass 1’**?

```
void muchAccessSoCacheWow(int *bigArr){
    // 48 KB array of ints
    int length = (48*1024)/sizeof(int);

    int access = 0;

    // traverse array with stride 8

    // pass 1
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }

    // pass 2
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }
}
```

	Miss Rate
A	0 %
B	25 %
C	33 %
D	50 %
E	66 %

Cache Misses

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on **‘pass 1’**?

```
void muchAccessSoCacheWow(int *bigArr){
    // 48 KB array of ints
    int length = (48*1024)/sizeof(int);

    int access = 0;

    // traverse array with stride 8

    // pass 1
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }

    // pass 2
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }
}
```

	Miss Rate
A	0 %
B	25 %
C	33 %
[✓ **D**]	50 %
E	66 %

Cache Misses

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on **‘pass 2’**?

```
void muchAccessSoCacheWow(int *bigArr){
    // 48 KB array of ints
    int length = (48*1024)/sizeof(int);

    int access = 0;

    // traverse array with stride 8

    // pass 1
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }

    // pass 2
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }
}
```

	Miss Rate
A	0 %
B	25 %
C	33 %
D	50 %
E	66 %

Cache Misses

Consider a 32 KB cache in a 32 bit address space. The cache is 8-way associative and has 64 bytes per block. A LRU (Least Recently Used) replacement policy is used. What is the miss rate on **‘pass 2’**?

```
void muchAccessSoCacheWow(int *bigArr){
    // 48 KB array of ints
    int length = (48*1024)/sizeof(int);

    int access = 0;

    // traverse array with stride 8

    // pass 1
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }

    // pass 2
    for(int i = 0; i < length; i+=8){
        access = bigArr[i];
    }
}
```

	Miss Rate
A	0 %
B	25 %
C	33 %
[✓ **D**]	50 %
E	66 %

Detailed explanation in Appendix!

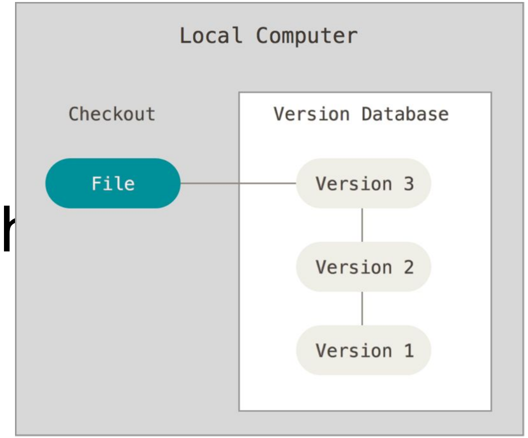
Appendix

Appendix: C Programming Style

- Properly document your code
  - Function + File header comments, overall operation of large blocks, any tricky bits
- Write robust code – check error and failure conditions
- Write modular code
  - Use interfaces for data structures, e.g. create/insert/remove/free functions for a linked list
  - No magic numbers – use `#define` or `static const`
- Formatting
  - 80 characters per line (use Autolab's highlight feature to double-check)
  - Consistent braces and whitespace
- No memory or file descriptor leaks

Appendix: Git: What is Git?

- Most widely used version control system out there
- Version control:
  - Help track changes to your source over time
  - Help teams manage changes on sh code



Appendix: Git Usage

- Commit early and often!
  - At minimum at every major milestone
  - Commits don’t cost anything!
- Popular stylistic conventions
  - Branches: short, descriptive names
  - Commits: A single, logical change. Split large changes into multiple commits.
  - Messages:
    - Summary: Descriptive, yet succinct
    - Body: More detailed description on **what** you changed, **why** you changed it, and what **side effects** it may have

Git Commands

- Clone: git clone <clone-repository-url>
- Add: git add . or git add <file-name>
- Push / Pull: git push / git pull
- Commit: git commit -m “your-commit-message”
  - Good commit messages are key!
  - Bad:“commit”, “change”, “fixed”
  - Good: “Fixed buffer overflow potential in AttackLab”

Appendix: Parsing Input with fscanf

- fscanf(FILE *stream, const char *format, …)
  - “scanf” but for files

- Arguments
  1. A stream pointer, e.g. from fopen()
  2. Format string for parsing, e.g “%c %d,%d”

  3+. **Pointers** to variables for parsed data
    - Can be pointers to stack variables

- Return Value
  - Success: # of parsed vars
  - Failure: EOF
- man fscanf

Appendix: fscanf() Example

```
FILE *pFile;
pFile = fopen(“trace.txt”, "r");  // Open file for reading

// TODO: Error check sys call

char access_type;
unsigned long address;
int size;

// Line format is " S 2f,1" or " L 7d0,3"
//      - 1 character, 1 hex value, 1 decimal value
while (fscanf(pFile, " %c %lx, %d", &access_type, &address, &size) > 0)
{
    // TODO: Do stuff
}

fclose(pFile); // Clean up Resources
```

Appendix: Discussion Questions

- What did the optimal transversal orders have in common?
- How does the pattern generalize to `int[8][8] A` and a cache that holds 4 lines each of 4 `int’`s?

Appendix: `Valgrind`

- Finding memory leaks

  -
    ```
    $ valgrind –leak-resolution=high –leak-check=full –show-reachable=yes –track-fds=yes ./myProgram arg1 arg
    ```
- Remember that Valgrind can be used for other things, like finding invalid reads and writes!

Appendix: `$ man 3 getopt`

- `int getopt(int argc, char * const argv[], const char *optstring);`
  - `int argc` → argument count passed to `main()`
    - Note: includes executable, so `./a.out 1 2` has `argc=3`
  - `char * const argv` is argument string array passed to `main`
  - `const char *optstring` → string with command line arguments
    - Characters followed by colon require arguments
      - Find argument text in `char *optarg`
    - `getopt` can't find argument or finds illegal argument sets `optarg` to "?"
    - Example: "`abc:d:`"
      - a and b are boolean arguments (not followed by text)
      - c and d are followed by text (found in `char *optarg`)
- Returns: `getopt` returns -1 when done parsing

Appendix: Clang / LLVM

- Clang is a (gcc equivalent) C compiler
  - Support for code analyses and transformation
  - Compiler will check you variable usage and declarations
  - Compiler will create code recording all memory accesses to a file
  - Useful for Cache Lab Part B (Matrix Transpose)