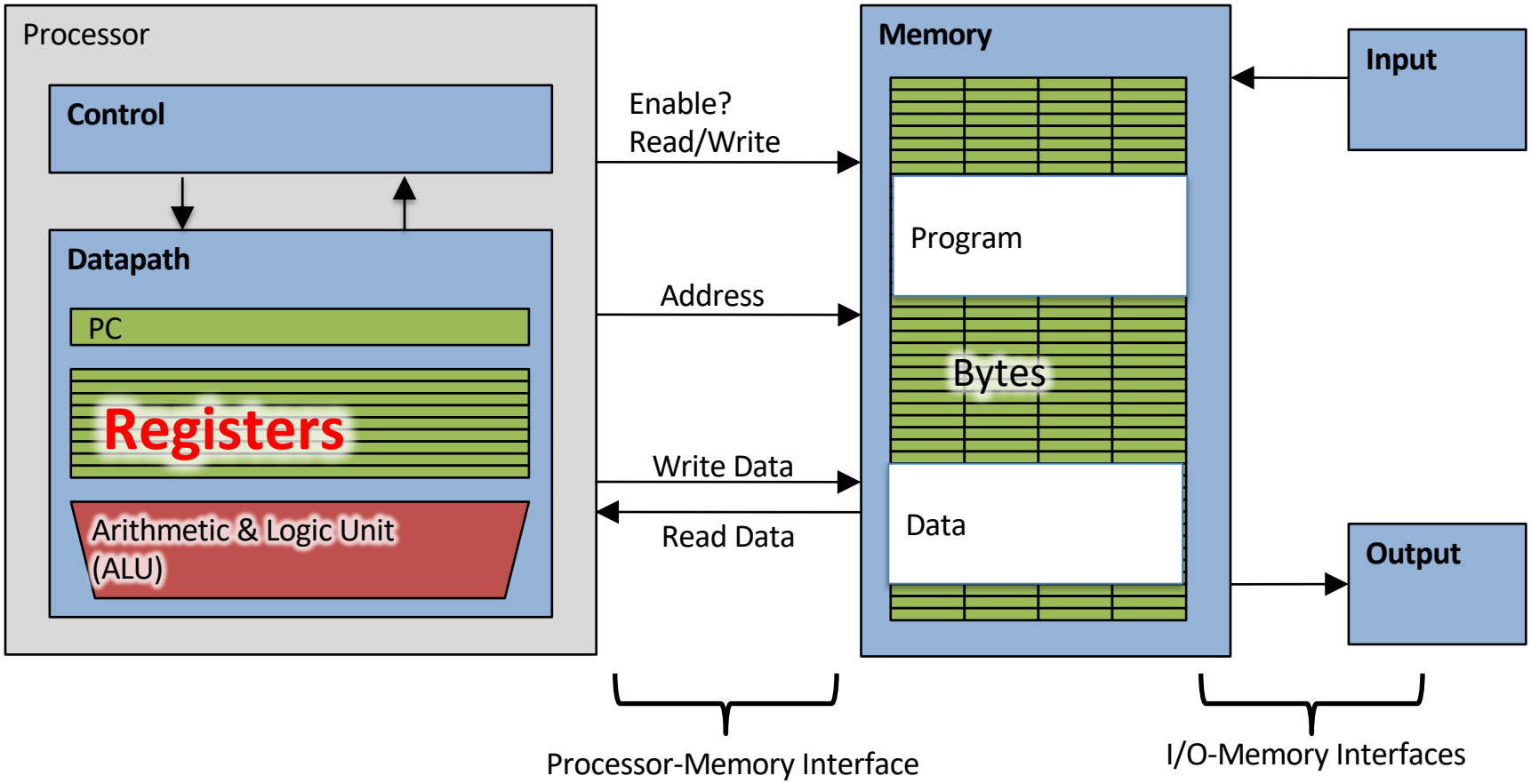
Caches and Memory Hierarchy

CMPT 295 Week 7

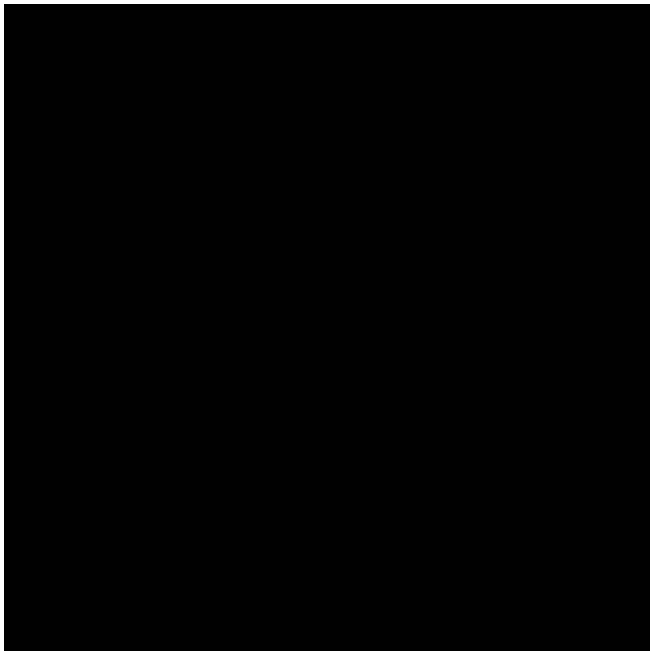
Aside: Registers are Inside the Processor

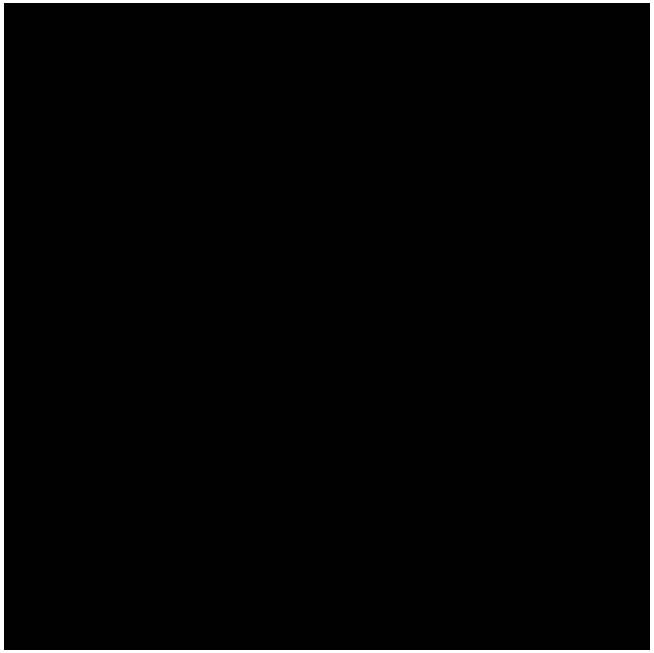


```
for(i=0;i < 4)      for(i=0;i < 4)
    sum += var          sum += A[i]
```

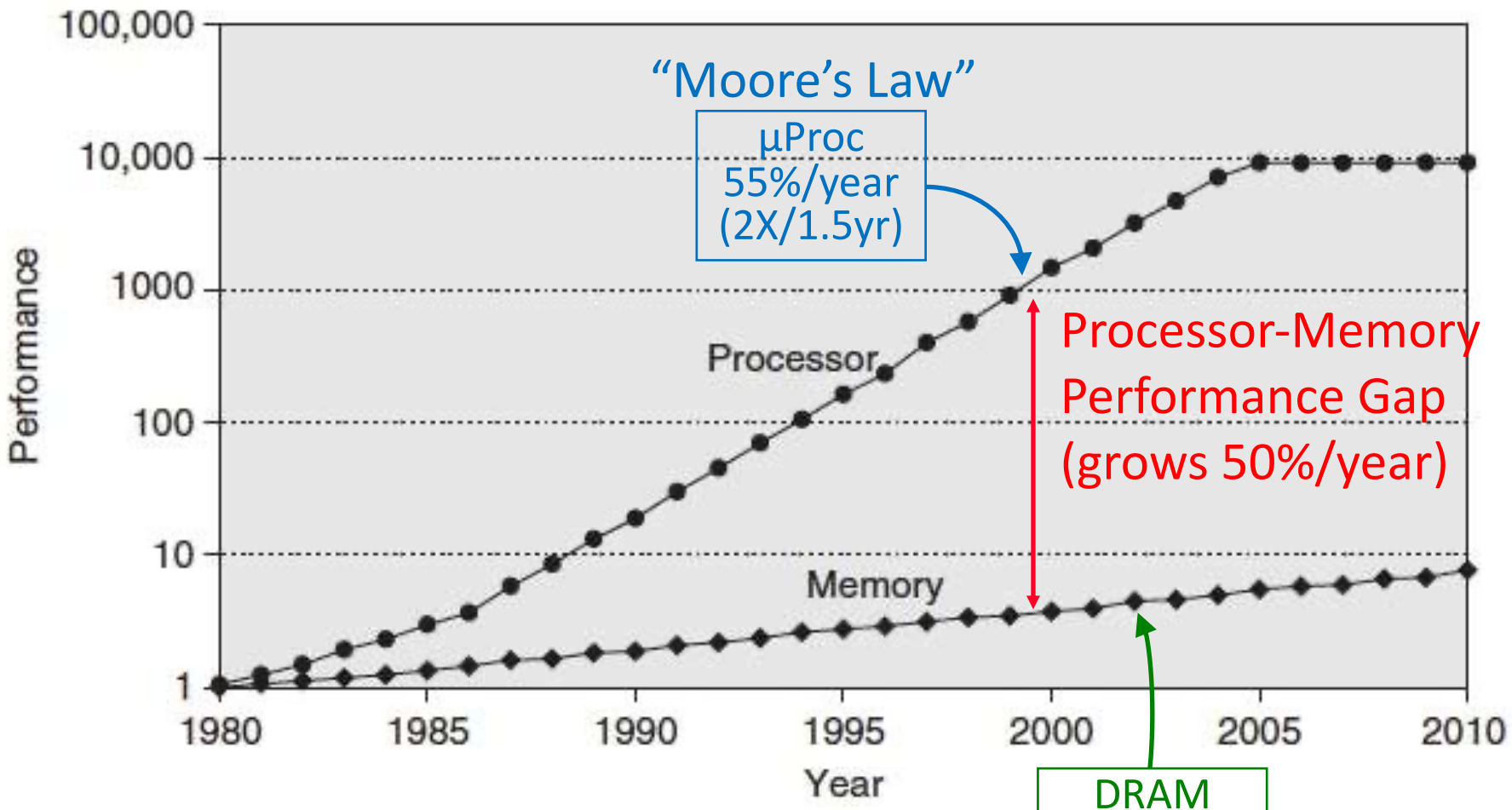
```
for (i = 0; i < 4){
    for(j = 0;  < 4) {
      A[i][j]
```

```
for (i = 0; i < 4){
    for(j = 0;  < 4) {
      A[j][i]
```



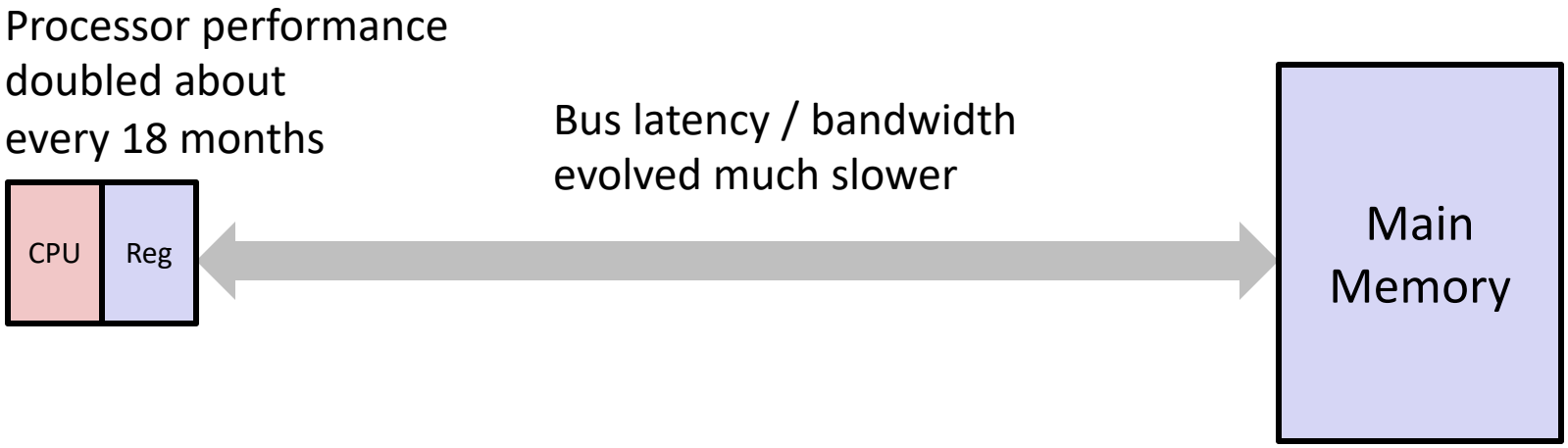


Processor-Memory Gap



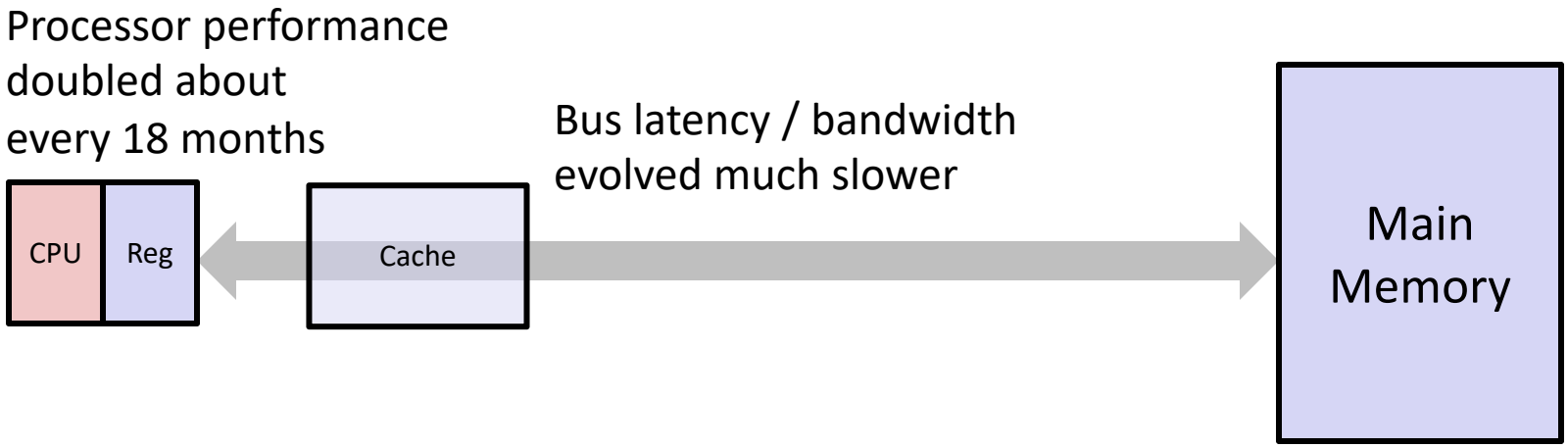
1989 first Intel CPU with cache on chip
1998 Pentium III has two cache levels on chip

Problem: Processor-Memory Bottleneck



Problem: lots of waiting on memory

Problem: Processor-Memory Bottleneck

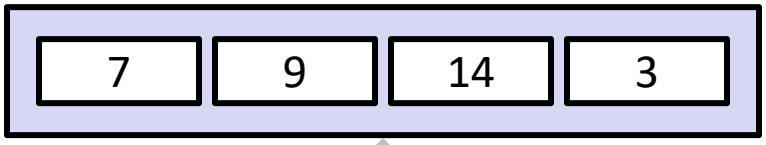


Solution: caches

Smaller memories, closer to CPU → faster

General Cache Mechanics

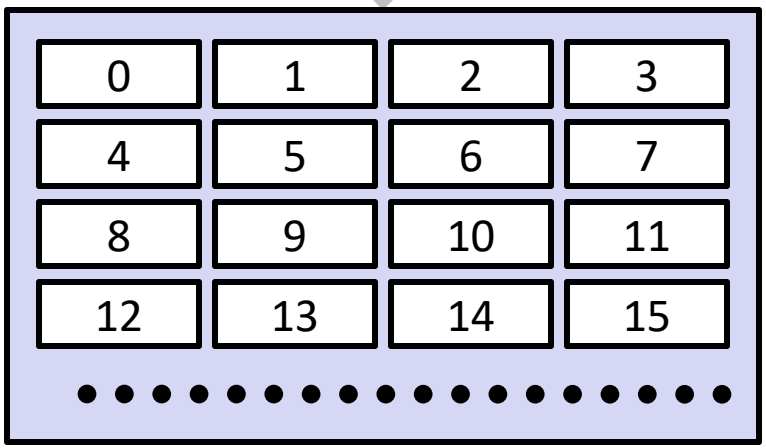
Cache



- Smaller, faster, more expensive memory
- Caches a subset of the blocks

Data is copied in block-sized transfer units

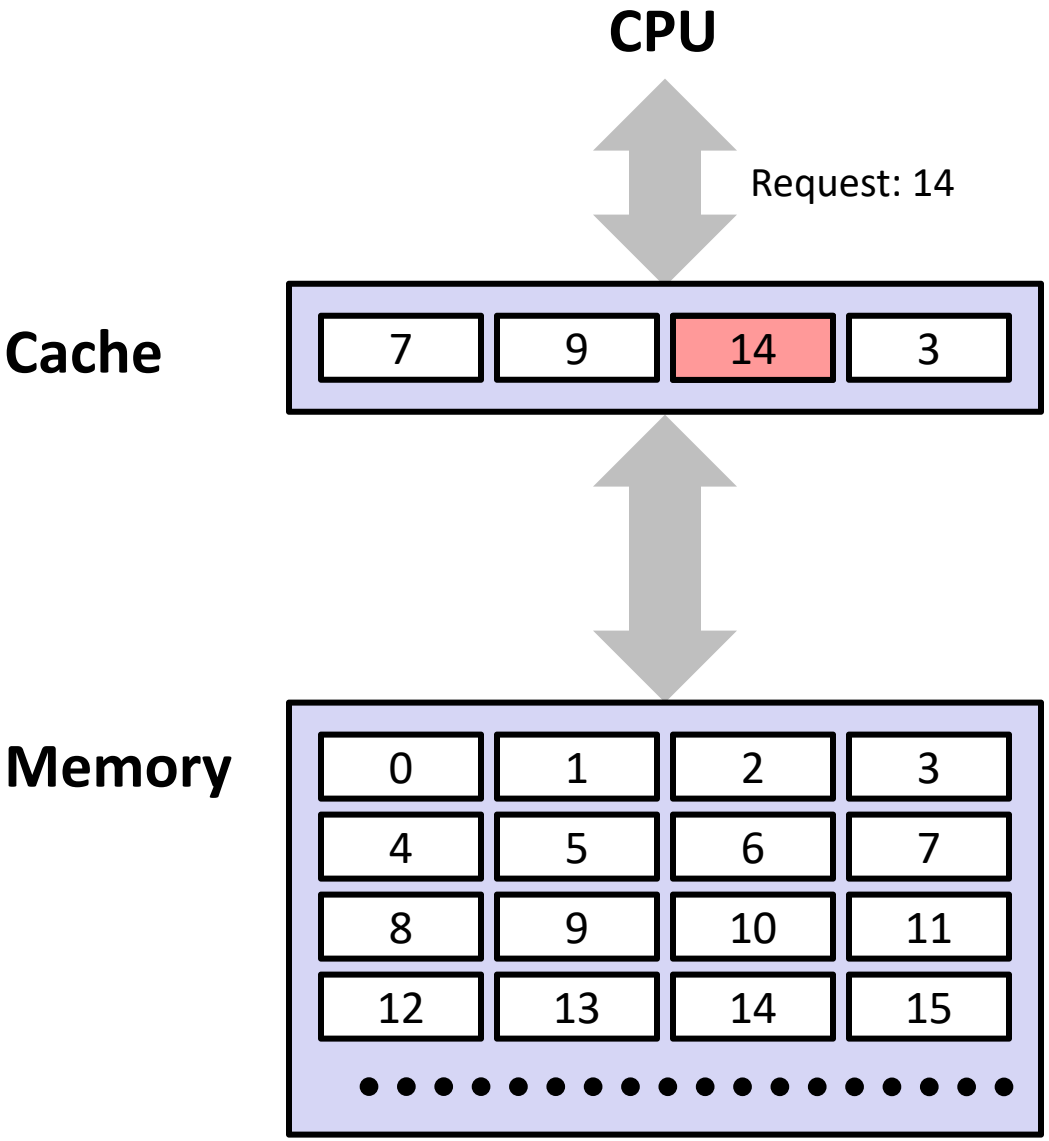
Memory



- Larger, slower, cheaper memory.
- Viewed as partitioned into "blocks"

Note: Cache "blocks" can also be referred to as cache "lines"

General Cache Concepts: Hit

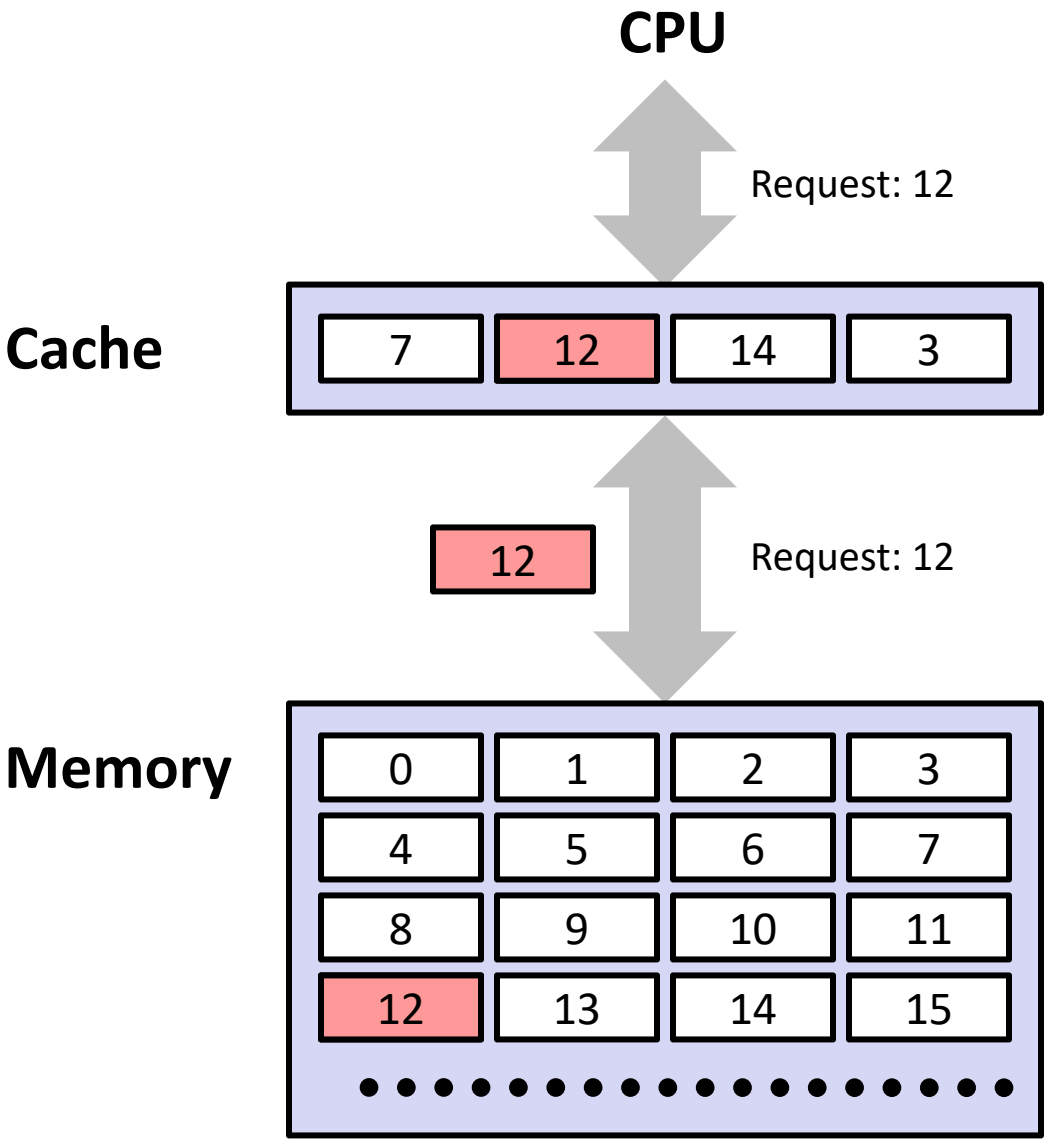


Data in block b is needed

Block b is in cache:
Hit!

Data is** returned **to CPU

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from *memory*

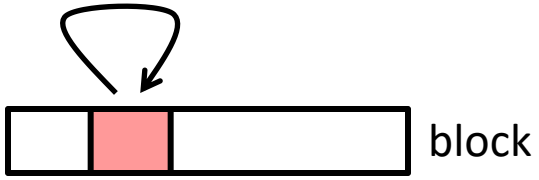
Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

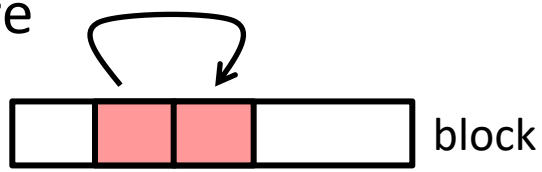
Data is *returned* ***to CPU***

Why Caches Work

- Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently



- *Temporal* locality:
  - *Recently referenced* items are likely to be referenced again in the near future



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

- How do caches take advantage of this?

Example: Any Locality?

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

- **Data:**
  - Temporal: `sum` referenced in each iteration
  - Spatial: array `a[]` accessed in stride-1 pattern
- **Instructions:**
  - Temporal: Cycle through loop repeatedly
  - Spatial: Reference instructions in sequential order

What Impacts Cache Hits/Misses?

- **Access Pattern**

  for (i = 0; i < 8; i++) vs. for (i = 0; i < 8; i=i+2)

- **Data layout**

  int a[8] vs. short a[8]

  int a[8] vs. int a[16]

- **Cache Geometry**

  Direct mapped vs. Set Associative (more later..)

DEMO

Int. Stride 1

```
// Block size 64 bytes
int a[8];
for (i = 0; i < 8; i++) {
    tmp = a[i];
```

Number of elements per block = 64/4 = 16

Hit:Access = 7:8 (Hit Rate)

Miss:Access = 1:8 (Miss Rate)

If we change loop to “i<16” and change array definition to int a[16], how would hit rate change?

Short Stride 1

```
// Block size 64 bytes
short a[20];
for (i = 0; i < 20; i++) {
    tmp = a[i];
```

Number of elements per block = 64/2 = 32

Hit:Access = 19:20

Miss:Access = 1:20

Int Stride 2

```
// Block size 64 bytes
int a[16];
for (i = 0; i < 16; i=i+2) {
    tmp = a[i];
```

Number of elements per block = 64/4 = 16

Accessed Elements per block = 16/2 = 8

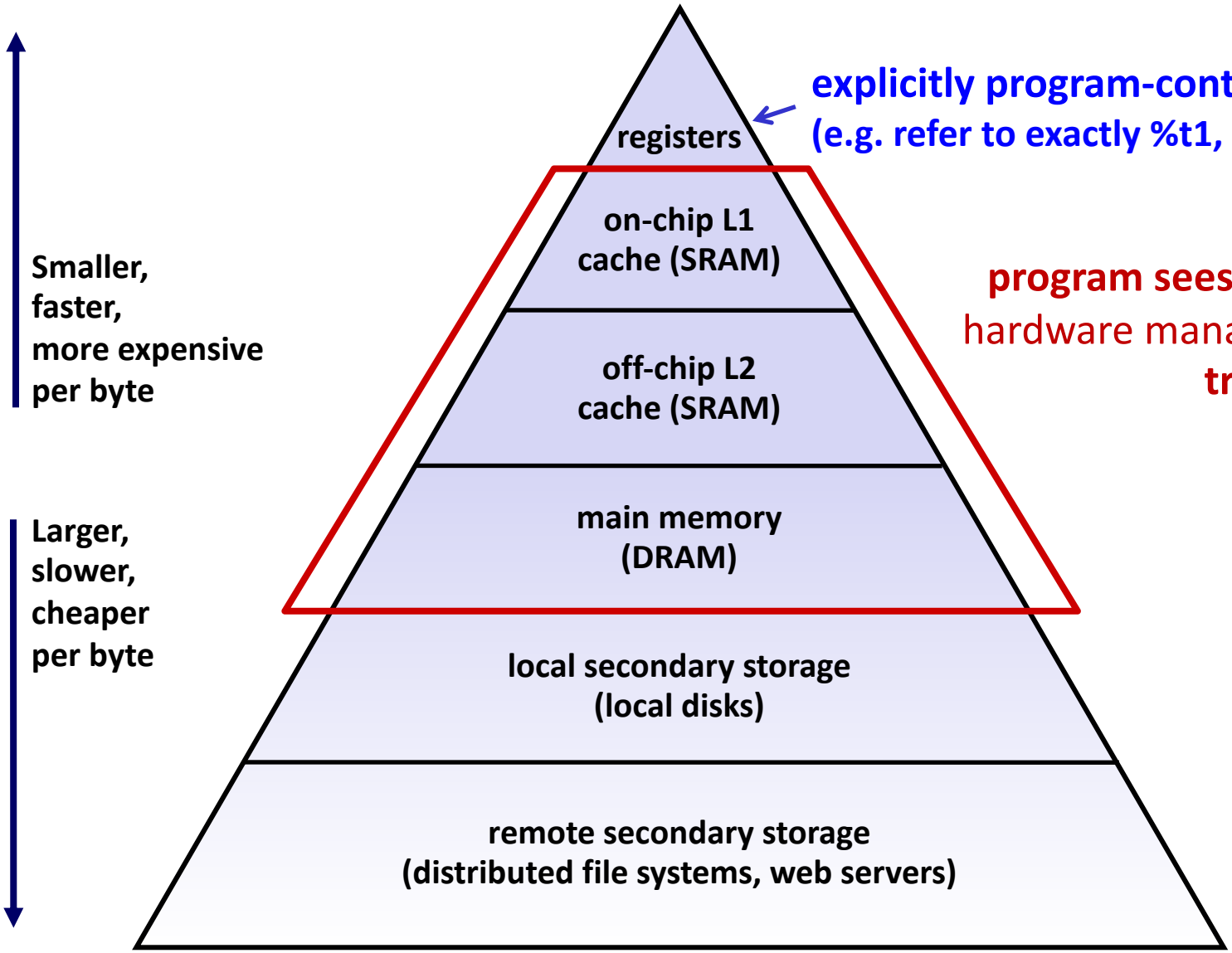
Hit:Access = 7:8

Miss:Access = 1:8

Caching Basics

- Caching in general
  - Successively higher levels contain “most used” data from lower levels
  - Exploits *temporal and spatial locality*
  - Caches are intermediate storage levels used to optimize data transfers between any system elements with different characteristics
- Cache Performance
  - Ideal case: found in cache (hit)
  - Bad case: not found in cache (miss), search in next level
  - Average Memory Access Time (AMAT) = HT + MR × MP
    - Hurt by Miss Rate and Miss Penalty

An Example Memory Hierarchy



Cache Performance

- Time for a memory operation depends on cache parameters:
  1. Hit time: Time to access cache on a cache hit
  2. Miss rate: Average misses per memory instruction
  3. Miss Penalty: Time to get data after a miss
- *Average Memory Access Time* (AMAT): average time to access memory considering both hits and misses

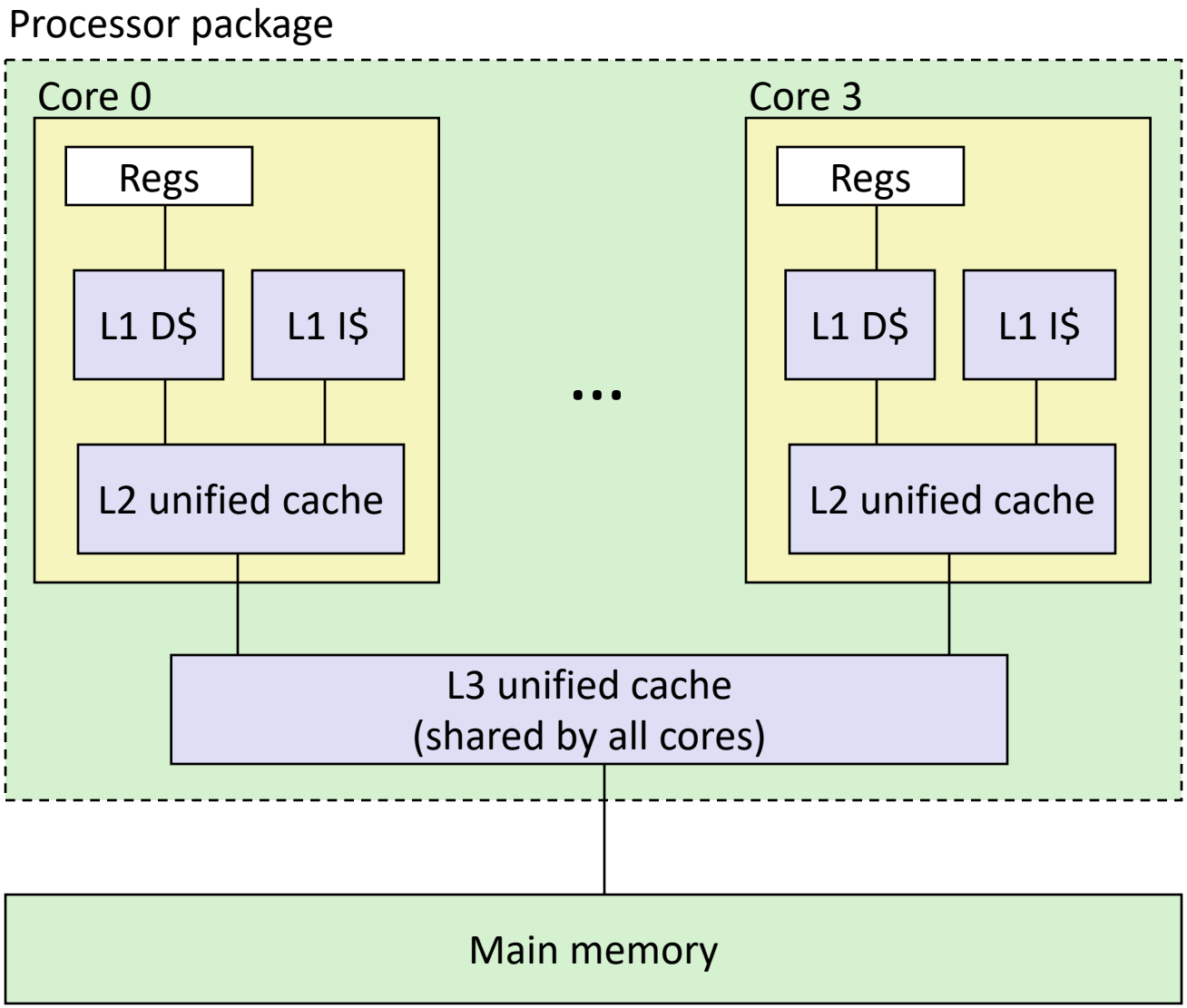
  AMAT = Hit time + Miss rate × Miss penalty

  (abbreviated AMAT = HT + MR × MP)
- 99% hit rate can be **twice** as good as 97% hit rate!
  - Assume HT of 1 ns and MP of 100 ns
  - 97%: AMAT = 1 + 0.03 x 100 = 4 ns
  - 99%: AMAT = 1 + 0.01 x 100 = 2 ns

Can we have more than one cache?

- Why?
  - Avoid going to memory, reduce miss penalty
- Typical performance numbers:
  - Miss Rate
    - L1 MR = 3-10%
    - L2 Global MR = Quite small (*e.g.* < 1%), depending on parameters, etc.
    - L2 (Local) MR typically larger than L1 MR (filtered by L1 hits)
  - Hit Time
    - L1 HT = 4 clock cycles
    - L2 HT = 10 clock cycles
  - Miss Penalty
    - MP = 50-200 cycles for missing in L2 & going to main memory
    - Trend: increasing!

Example Cache Hierarchy



Block size:
64 bytes for all caches

L1 I-cache and D-cache:
32 KiB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KiB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MiB, 16-way,
Access: 30-40 cycles

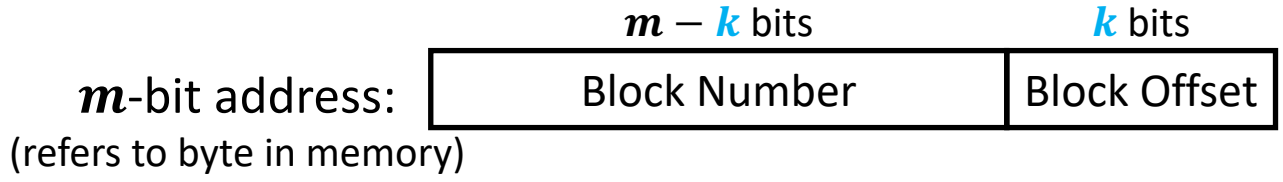
Internal Cache Organization

- We want to **store data efficiently** in the cache.
- We also need to **quickly find data** in the cache when we need it.

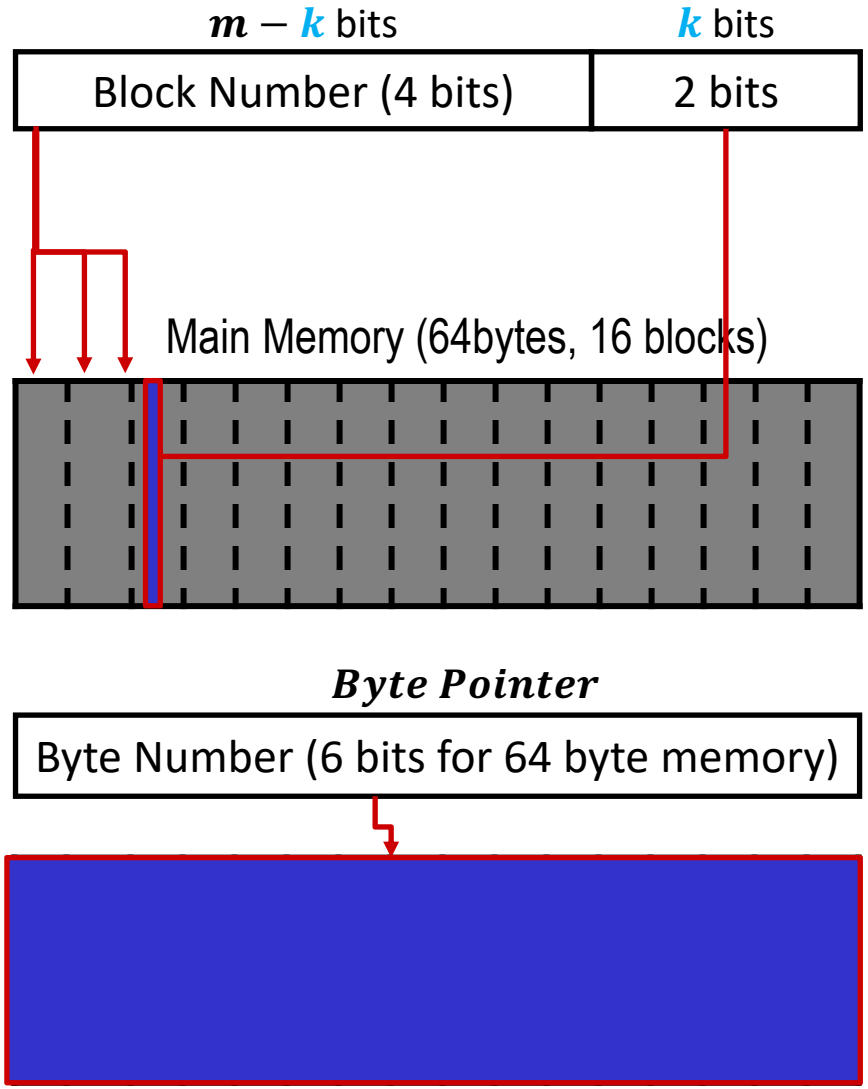
- Address Splitting:
  - **Determine where data goes in the cache**.
  - **Check if the data is already in the cache**.
  - **Access the exact piece of data we need**.

Cache Organization

- Block Size (K): Unit of transfer between $ and Mem
  - Given in bytes and always a power of 2 (*e.g.* 64 B)
  - Blocks consist of adjacent bytes (differ in address by 1)
  - **Exact byte within a block of data.**
- Offset field
  - Low-order $\log_2(K) = \boldsymbol{k}$ bits of address tell you which byte within a block
    - (address) mod 2^n = n lowest bits of address
- Block Number = (address) mod (Block Size)



Block based Addressing vs Byte Addressing



How to identify different blocks in cache?

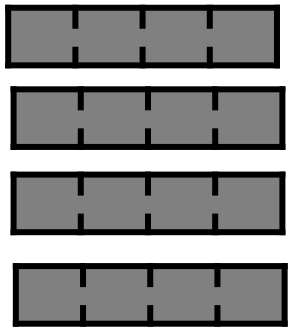
- for i = 0 to N

  Calculate(A[i])

```
lw      a0, 0(s1)
beqz    a0, LBB1_2
call    calculate
sw      a0, 0(s2)
addi    s2, s2, 4
addi    s0,s0,-1
addi    s1, s1, 4
j       LBB1_2
```

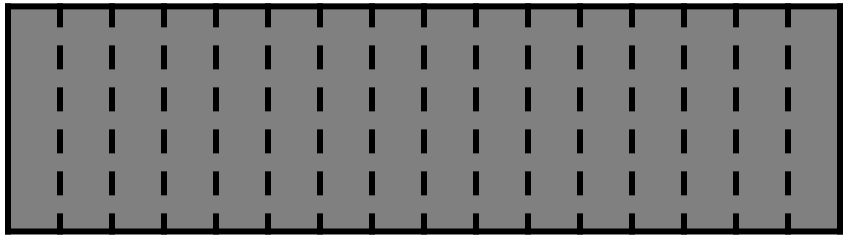
i = 0 – 0x000000 Block: 0000
i = 1 - 0x000004 Block: 0001
i = 2 - 0x000008 Block: 0002
.....

Cache (16 bytes, 4 blocks)



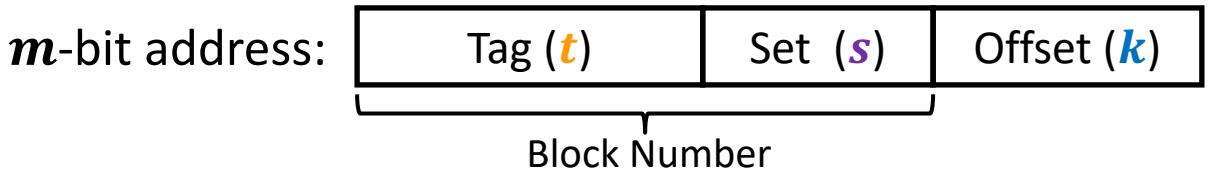
Block size: 4B
Cache size: 16B

Main Memory (64bytes, 16 blocks)



Mapping Memory Address to Cache

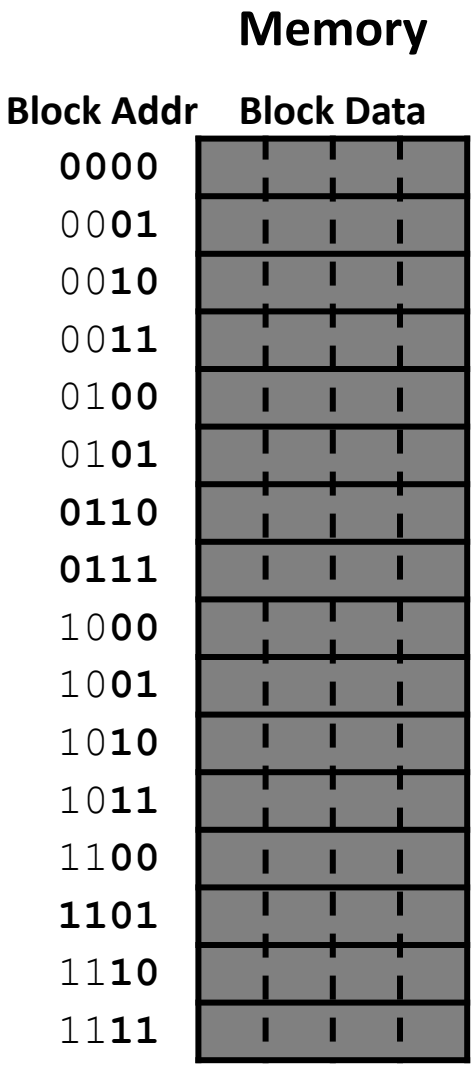
- CPU sends load/store address, address breakdown:

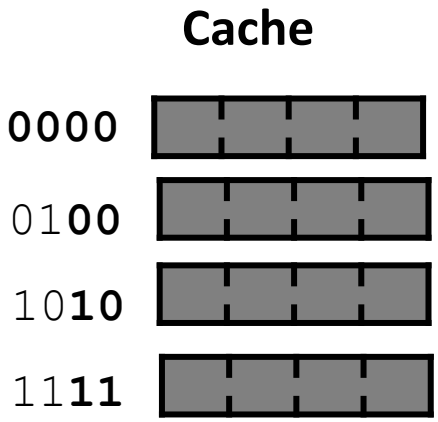


  - **Set Index** field tells you where to look in cache
  - **Tag** field lets you check that data is the block you want
  - **Offset** field selects specified start byte within block
  - $k = \log_2(K)$; $s = \log_2(C/(K*E))$; $t = m - s - k$
  - **K: Block Size (bytes), E: Associativity; C: Cache Size (bytes)**

- **Book Shelf Analogy**
  - **Block: page within a book**
  - **Set Index: Which shelf took look at**
  - **Tag: Unique QRcode or Barcode for every book**

Tags Differentiate Blocks in Cache





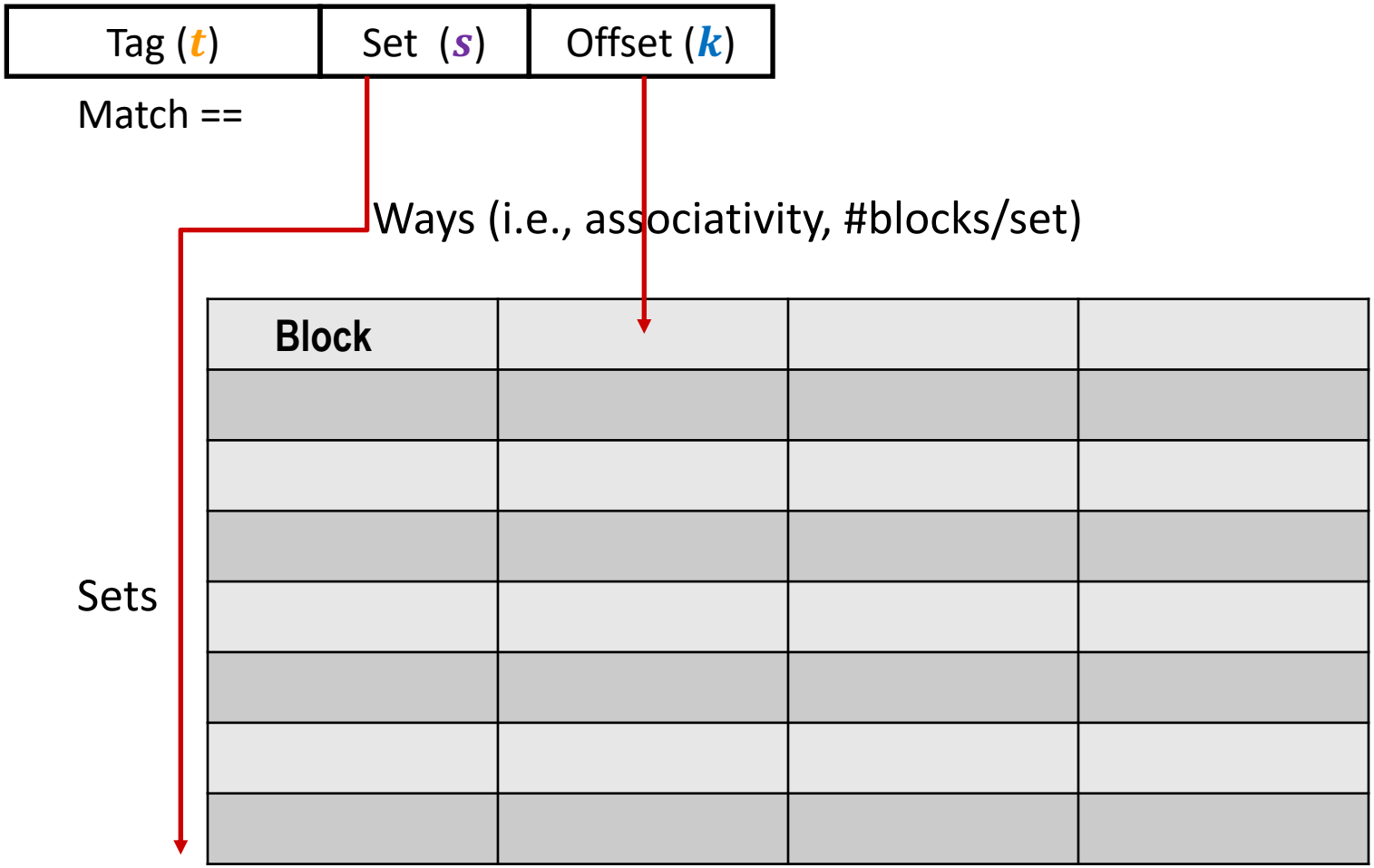
Block size (K) = 4B
Associativity (E) = 1
Cache size (C) = 16B
#Blocks = C/K = 4

- Offset bits k = $\log_2$(K)
- Set index bits s = $\log_2$(#Sets)
  = $\log_2$(C/(K*E))
- Tag = rest of address bits
  - t bits = $m - s - k$
  - Check this during a cache lookup

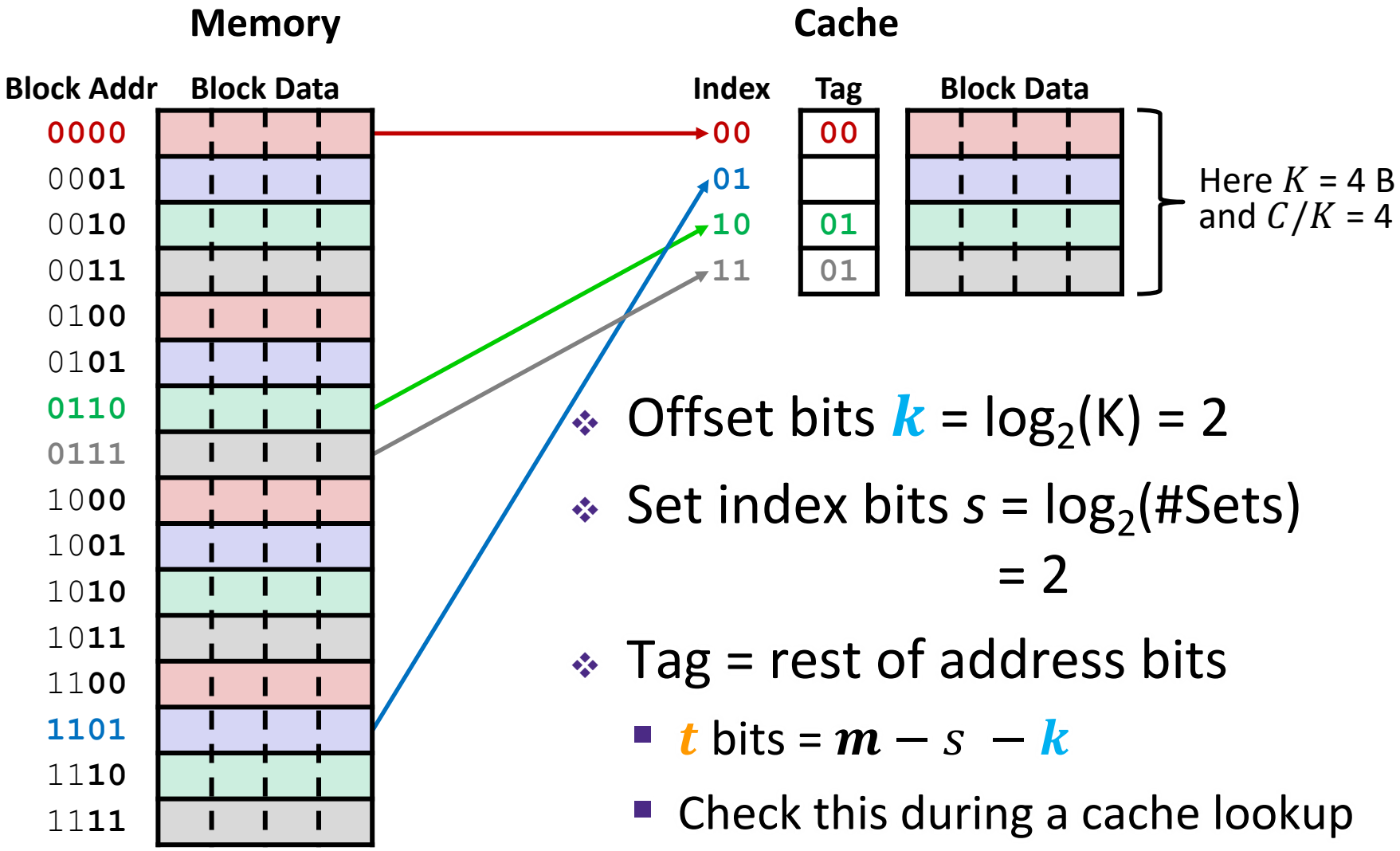
Cache Organization

- **Cache Size = #Blocks x Block Size**

  = #Sets x Associativity x Block Size



Tags Differentiate Blocks in Same Index



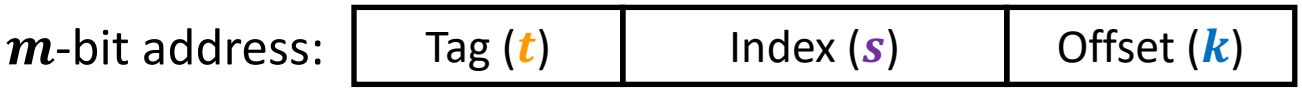
- Offset bits k = $\log_2$(K) = 2
- Set index bits s = $\log_2$(#Sets) = 2
- Tag = rest of address bits
  - t bits = $m - s - k$
  - Check this during a cache lookup

Example Placement

block size:	16 B
capacity:	8 blocks
address:	16 bits

- Where would data from address `0x1833` be placed?
  - Binary: `0b 0001 1000 0011 0011`

$t = m - s - k$ $\quad s = \log_2(C/(K * E))$ $\quad k = \log_2(K)$



s = ?

Direct-mapped

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

s = ?

2-way set associative

Set	Tag	Data
0		
1		
2		
3		

s = ?

4-way set associative

Set	Tag	Data
0		
1		

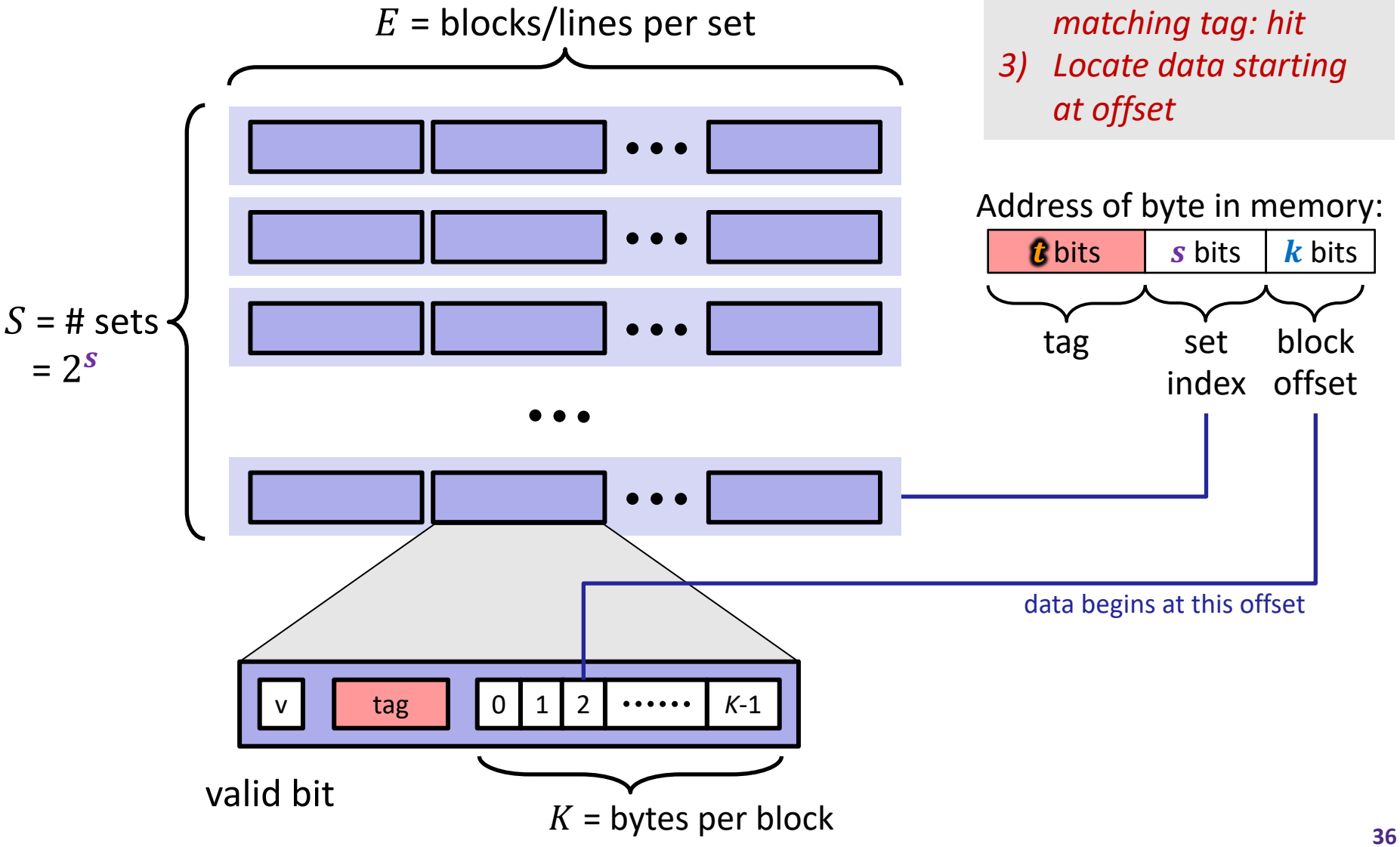
s = 0

Fully associative

Set	Tag	Data
0		

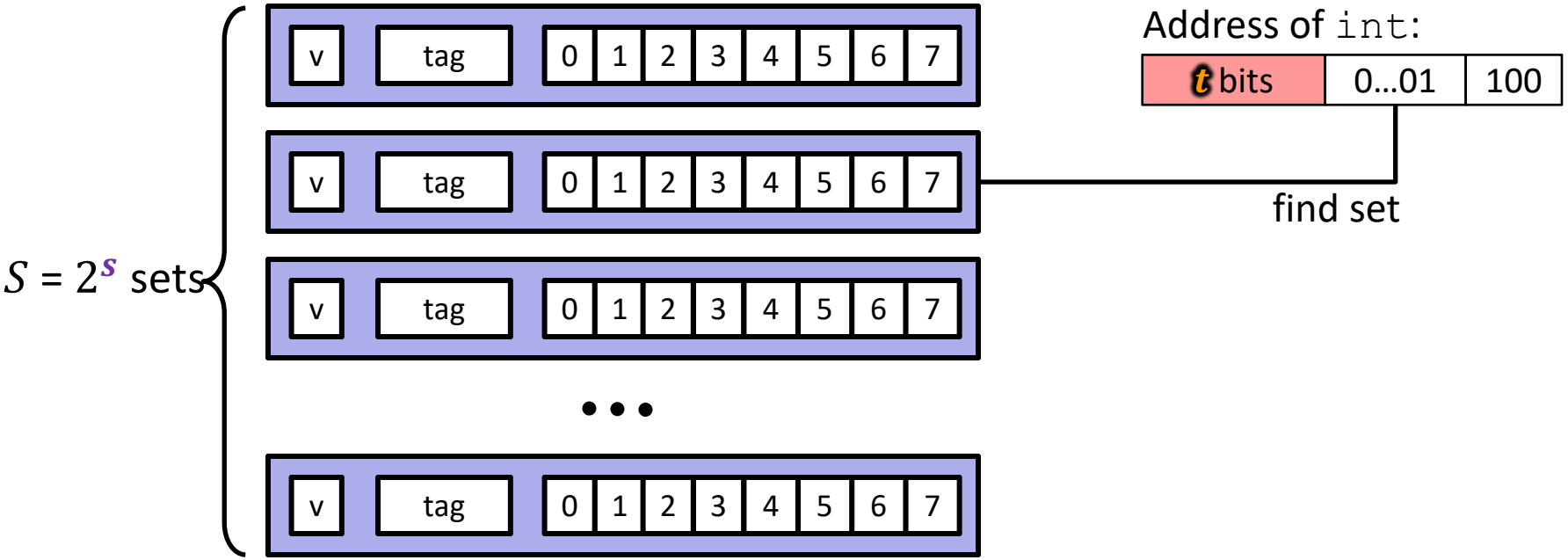
Cache Read

1) *Locate set*
2) *Check if any line in set is valid and has matching tag: hit*
3) *Locate data starting at offset*



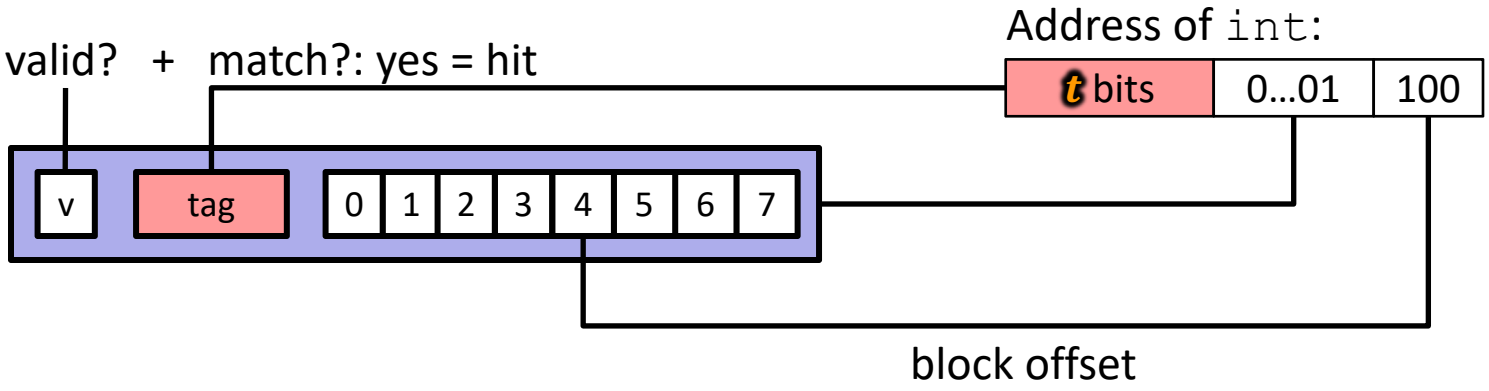
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Block Size K = 8 B



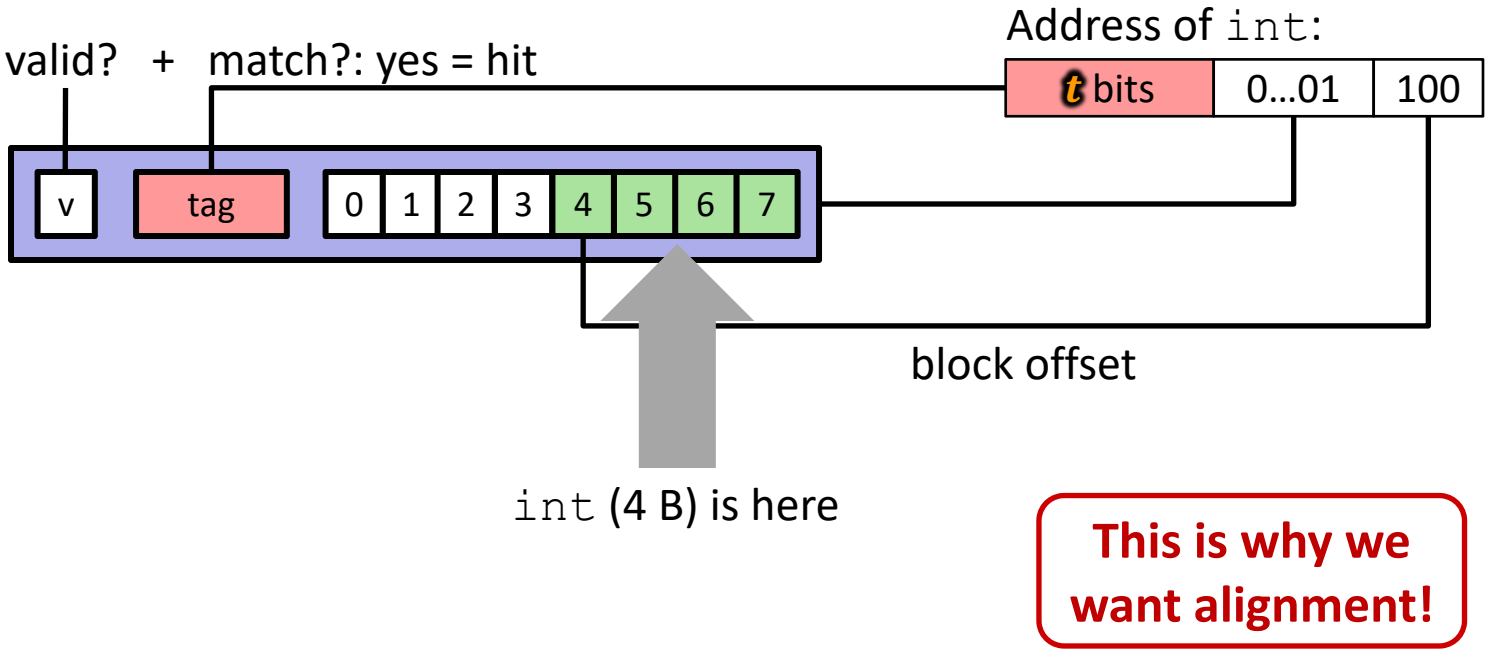
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Block Size K = 8 B



Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Block Size K = 8 B

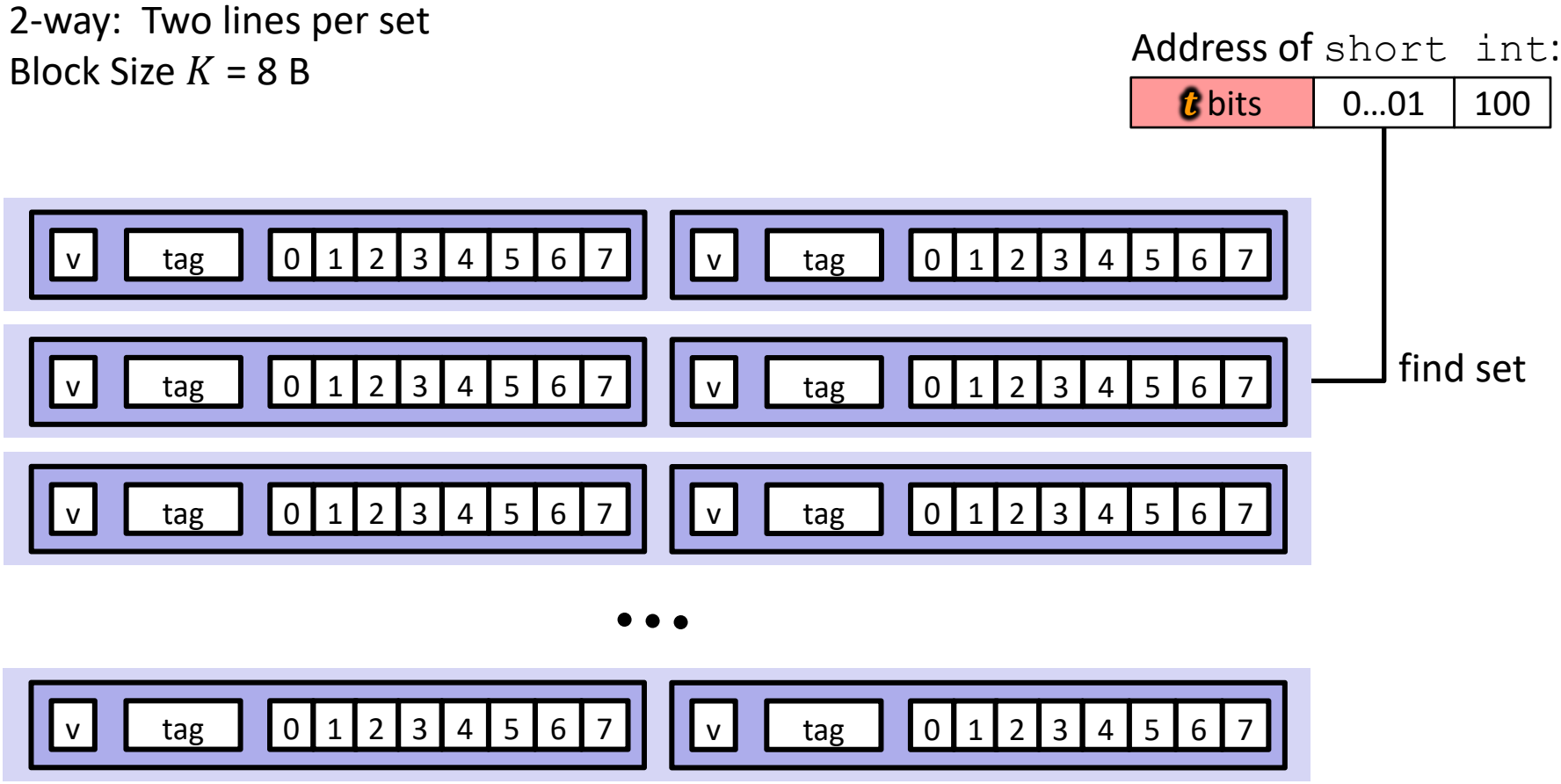


This is why we want alignment!

No match? Then old line gets evicted and replaced

Example: Set-Associative Cache (E = 2)

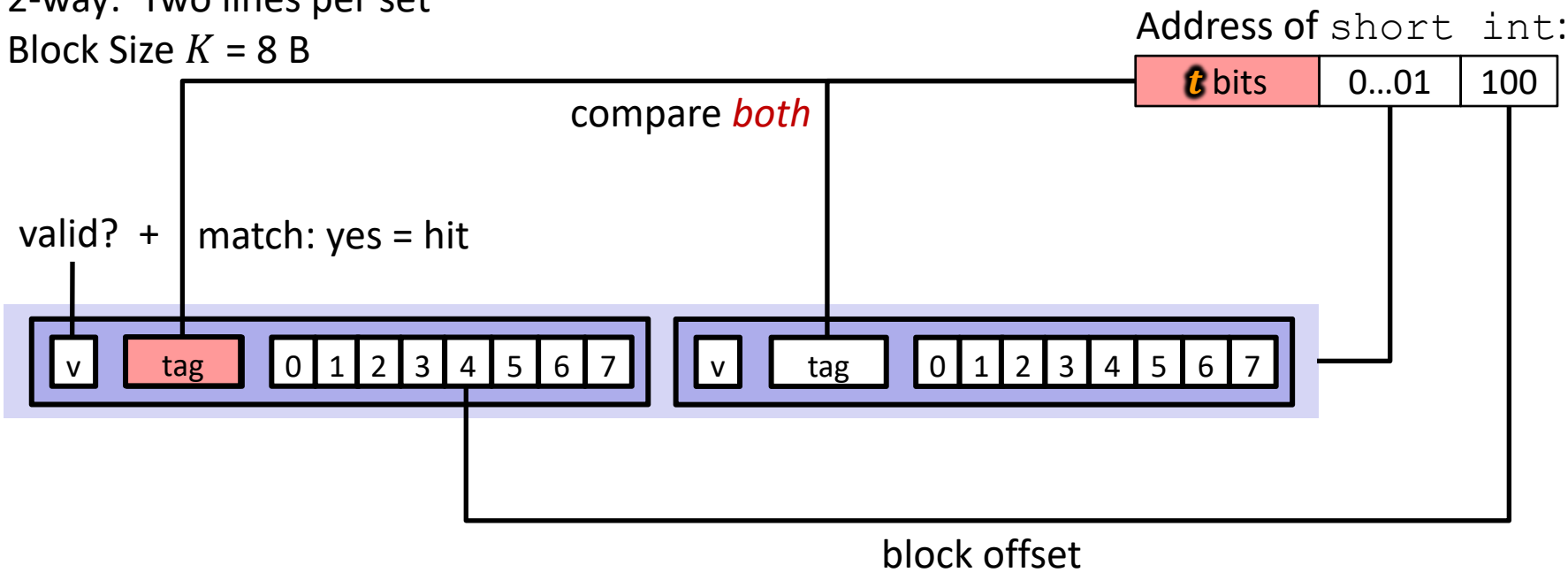
2-way: Two lines per set
Block Size K = 8 B



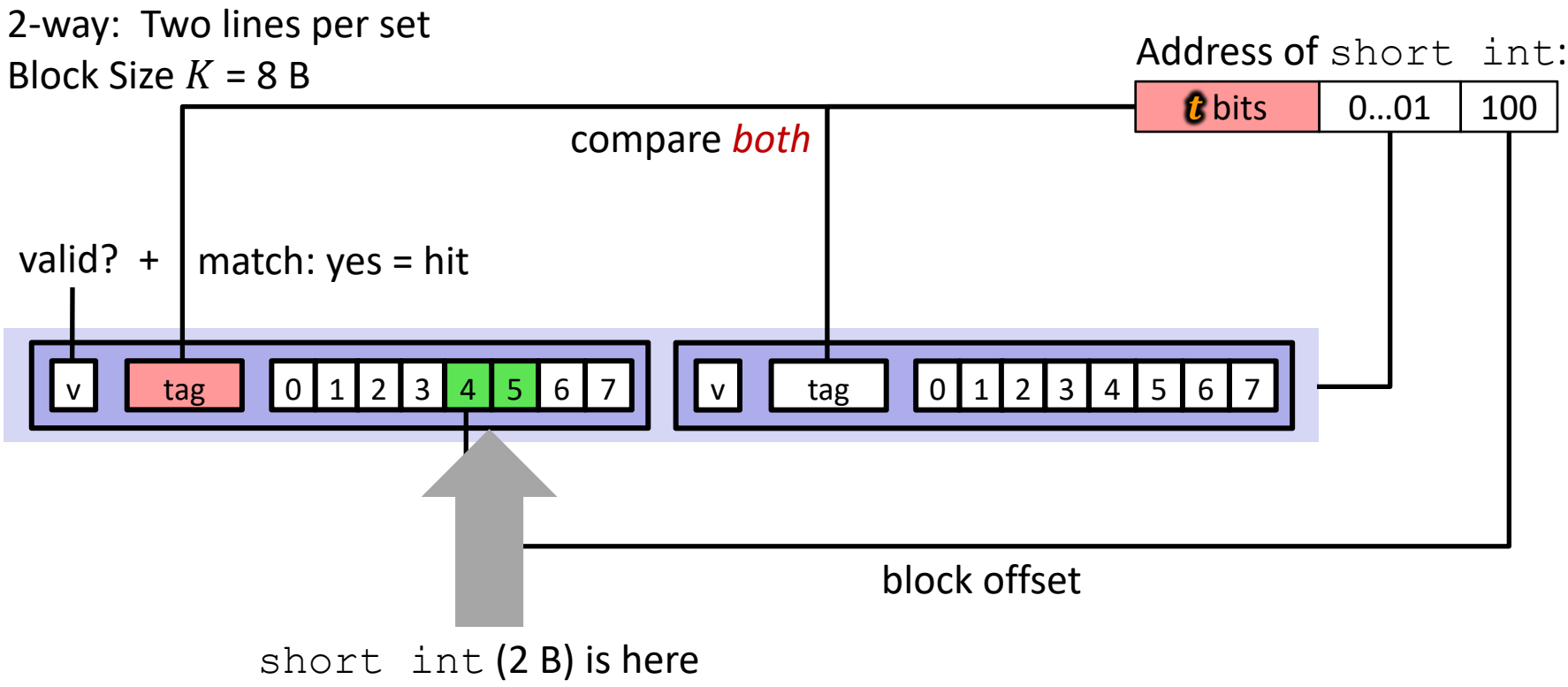
Example: Set-Associative Cache (E = 2)

2-way: Two lines per set

Block Size K = 8 B



Example: Set-Associative Cache (E = 2)



No match?

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

Example Code Analysis Problem

- Assuming the cache starts cold (all blocks invalid) and `sum` is stored in a register, calculate the **miss rate**:
  - m = 12 bits, C = 256 B, K = 32 B, E = 2

```
#define SIZE 8
long ar[SIZE][SIZE], sum = 0;  // &ar=0x800
for (int i = 0; i < SIZE; i++)
   for (int j = 0; j < SIZE; j++)
      sum += ar[i][j];
```

Row Major

```
for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
            A[i][j]
```

Hits: N-1 (N: number of elements per block)

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

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];

    return sum;
}
```

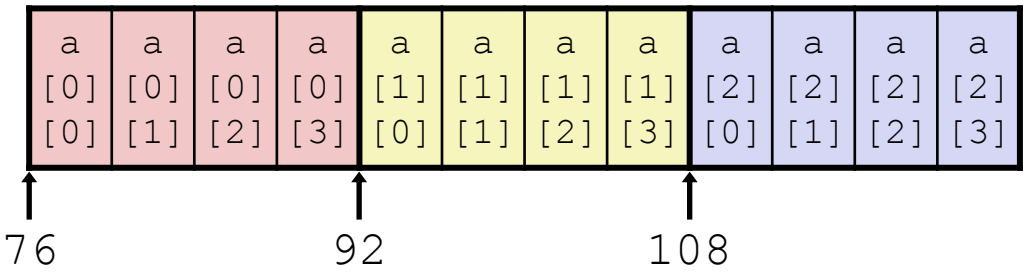
M = 3, N=4

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Access Pattern:
stride = ?

1) a[0][0]
2) a[0][1]
3) a[0][2]
4) a[0][3]
5) a[1][0]
6) a[1][1]
7) a[1][2]
8) a[1][3]
9) a[2][0]
10) a[2][1]
11) a[2][2]
12) a[2][3]

Layout in Memory



Column Major

```
for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
            A[j][i]
```

Hits: 0

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

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];

    return sum;
}
```

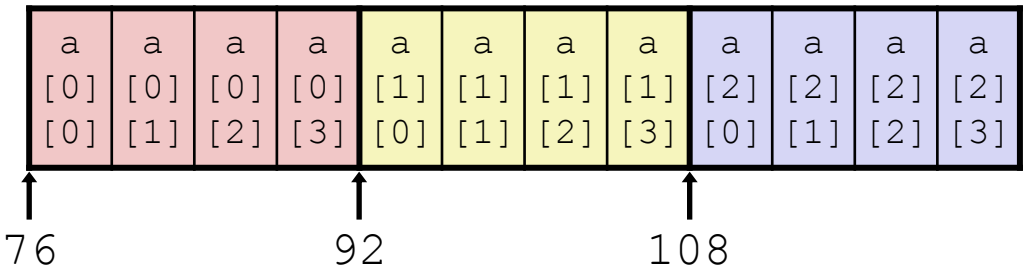
M = 3, N=4

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Access Pattern:
stride = ?

1) a[0][0]
2) a[1][0]
3) a[2][0]
4) a[0][1]
5) a[1][1]
6) a[2][1]
7) a[0][2]
8) a[1][2]
9) a[2][2]
10) a[0][3]
11) a[1][3]
12) a[2][3]

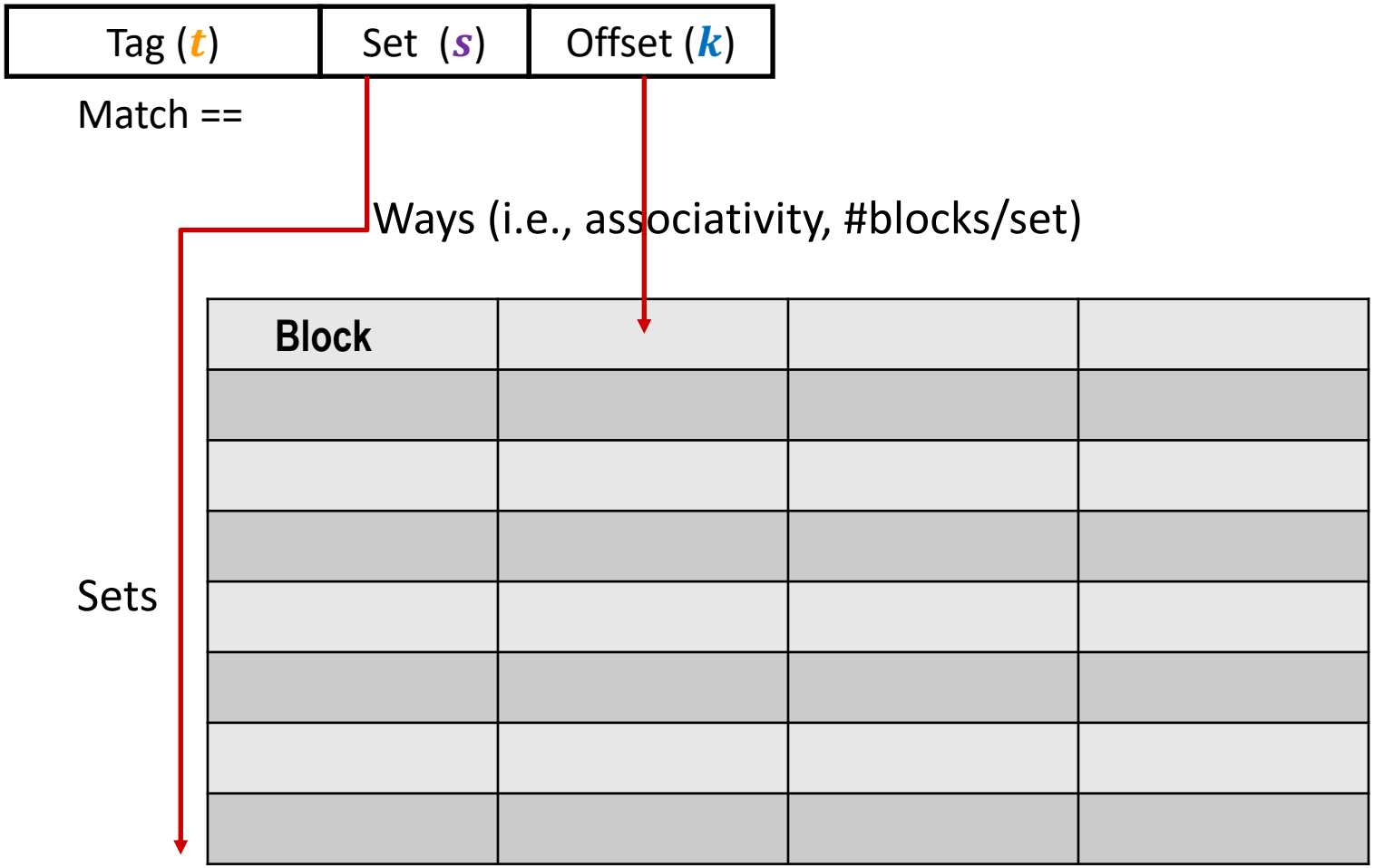
Layout in Memory



Cache Organization

- **Cache Size = #Blocks x Block Size**

  = #Sets x Associativity x Block Size



Peer Instruction Question

- We have a cache of size 2 KiB with block size of 128 B. If our cache has 2 sets, what is its associativity?
  - A. 2
  - B. 4
  - C. 8
  - D. 16
  - E. We're lost…

- If addresses are 16 bits wide, how wide is the Tag field?

Other Questions

- We have a cache with block size of 128 B. Cache is 4-way set-associative and has 8 sets. How big is the cache? (What is the cache capacity)?

Other Questions

- A 4KB Cache is 4-way set associative with 64 B blocks. Which bits are used for set index? (Also: How many sets does the cache have?)

Other Questions

- A 32KB Cache is 8-way set associative and has 16 sets. Which bits are used for byte offset? (Also: What is the block size?)

Other Questions

- A direct-mapped cache uses 4 bits for set index and 6 bits for byte offset. How big is the cache?

Cache Miss Classification: The 3Cs

- Compulsory: (Many names: cold start, process migration (switching processes), 1st reference)
  - First access to block impossible to avoid; Effect is small for long running programs
- Capacity:
  - Cache cannot contain all blocks accessed by the program, i.e., misses in a fully associative cache.
- Conflict: (collision)
  - Multiple memory locations mapped to the same cache set (not enough associativity)

Hit, Compulsory, Capacity or Conflict Miss

- 0x00000004
- 0x00000005
- 0x00000068
- 0x000000C8
- 0x00000068
- 0x000000DD
- 0x00000045
- 0x00000004
- 0x000000C8

Hit, Compulsory, Capacity or Conflict Miss

- • 0x00000004, Compulsory
- • 0x00000005
  • 0x00000068, Compulsory
- • 0x000000C8, Compulsory
- • 0x00000068
- • 0x000000DD, Compulsory
- • 0x00000045, Compulsory
- • 0x00000004
- • 0x000000C8

Hit, Compulsory, Capacity or Conflict Miss

- 0x00000004, Compulsory
- 0x00000005
- 0x00000068, Compulsory
- 0x000000C8, Compulsory
- 0x00000068
- 0x000000DD, Compulsory
- 0x00000045, Compulsory
- 0x00000004, Capacity
- 0x000000C8, Capacity

Hit, Compulsory, Capacity or Conflict Miss

- 0x00000004, Compulsory
- 0x00000005, Hit
- 0x00000068, Compulsory
- 0x000000C8, Compulsory
- 0x00000068
- 0x000000DD, Compulsory
- 0x00000045, Compulsory
- 0x00000004, Capacity
- 0x000000C8, Capacity

Hit, Compulsory, Capacity or Conflict Miss

- 0x00000004, Compulsory
- 0x00000005, N/A
- 0x00000068, Compulsory
- 0x000000C8, Compulsory
- 0x00000068, Conflict
- 0x000000DD, Compulsory
- 0x00000045, Compulsory
- 0x00000004, Capacity
- 0x000000C8, Capacity

Cache Code Analysis Problem

- Cache-A Direct-mapped, 4KB, 64 sets
- Cache-B Set-associative, 4KB, 2 ways, 32 sets

```
int size = 4096; // int is 4 bytes
int a[size];
long long int a_long[size]; // long long int is 8 bytes
/* loop 1 */
for (int i = 0; i < size; i++) {
    a[i] = i;
}
/* loop 2 */
for (long long int i = 0; i < size; i++) {
    a_long[i] = i;
}
/* loop 3 */
for (int i = 0; i < size/2; i += 1) {
    a[(size/2)+i] = a[i];
}
```

Question: What are hit rates for loop1, loop 2 (assume loop 1 has run), loop3 (assume both previous loops have run).