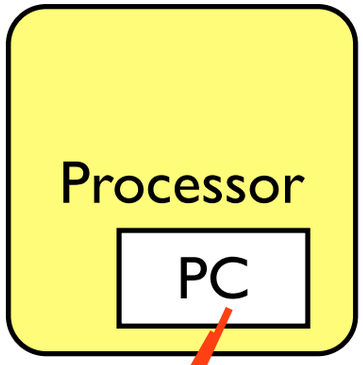
Memory hierarchy / Cache

Hung-Wei Tseng

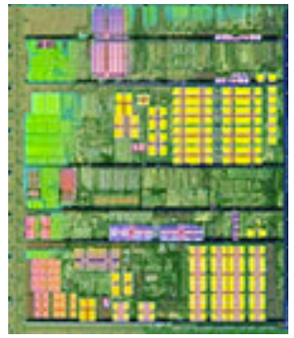
Memory gap

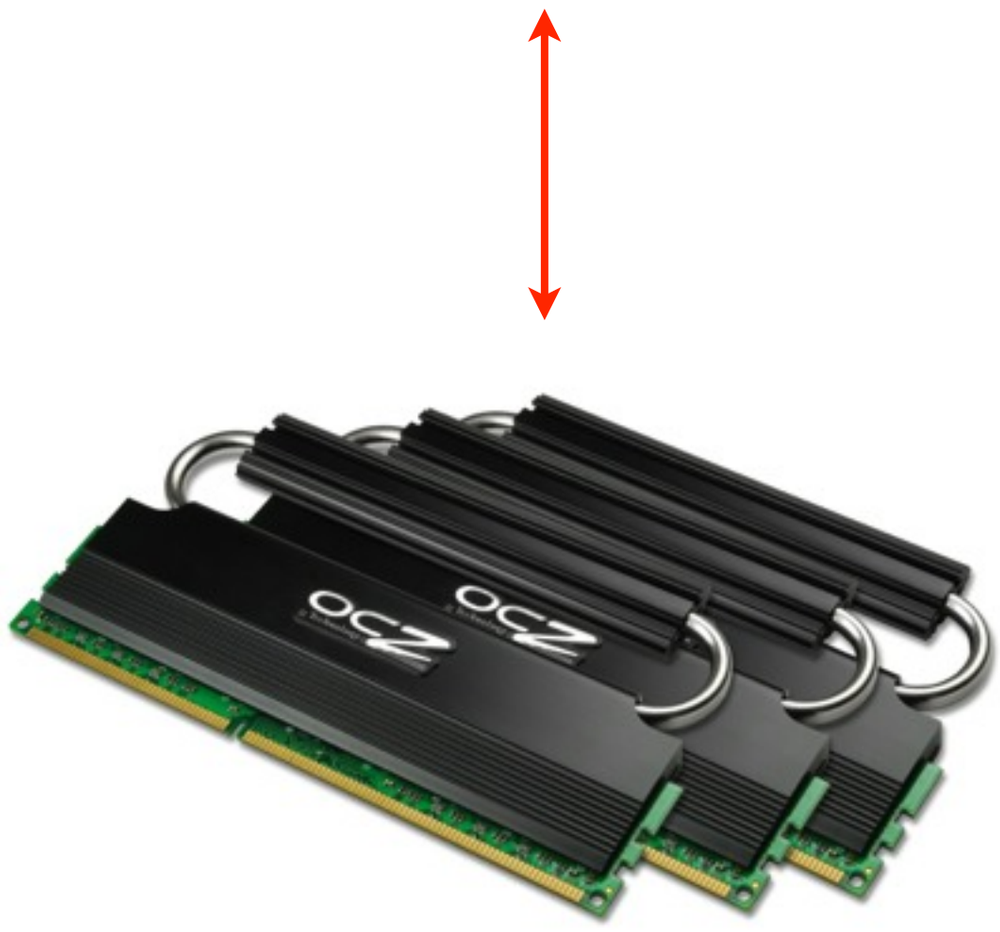
Memory in stored program computer



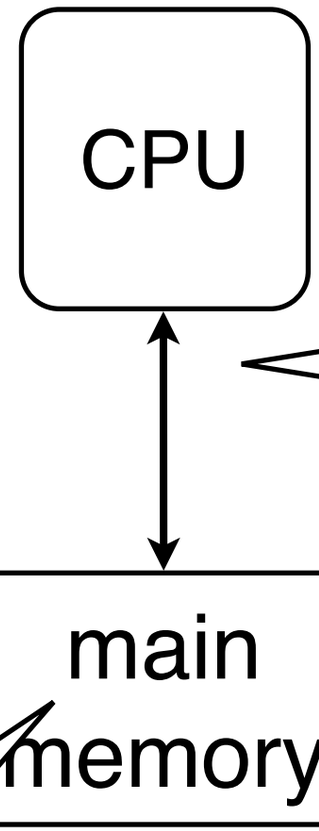
instruction memory

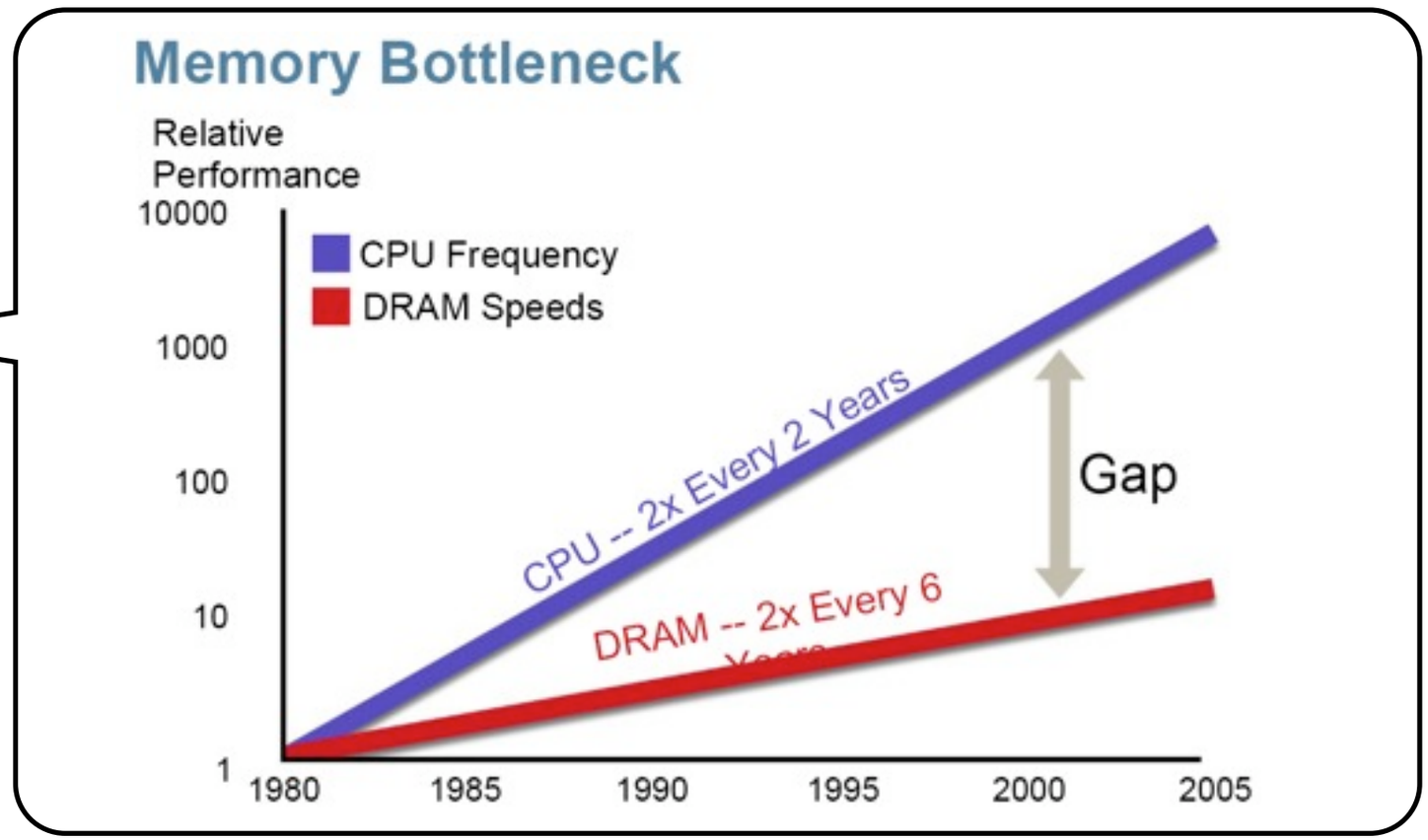
```
120007a30:  0f00bb27   ldah   gp,15(t12)
120007a34:  509cbd23   lda    gp,-25520(gp)
120007a38:   00005d24   ldah   t1,0(gp)
120007a3c:   0000bd24   ldah   t4,0(gp)
120007a40:   2ca422a0   ldl    t0,-23508(t1)
120007a44:   130020e4   beq    t0,120007a94
120007a48:   00003d24   ldah   t0,0(gp)
120007a4c:   2ca4e2b3   stl    zero,-23508(t1)
120007a50:   0004ff47   clr    v0
120007a54:   28a4e5b3   stl    zero,-23512(t4)
120007a58:   20a421a4   ldq    t0,-23520(t0)
120007a5c:   0e0020e4   beq    t0,120007a98
120007a60:   0204e147   mov    t0,t1
120007a64:   0304ff47   clr    t2
120007a68:   0500e0c3   br     120007a80
```





Why memory hierarchy?






```
lw   $t2, 0($a0)
add  $t3, $t2, $a1
addi $a0, $a0, 4
subi $a1, $a1, 1
bne  $a1, LOOP
lw   $t2, 0($a0)
add  $t3, $t2, $a1
```

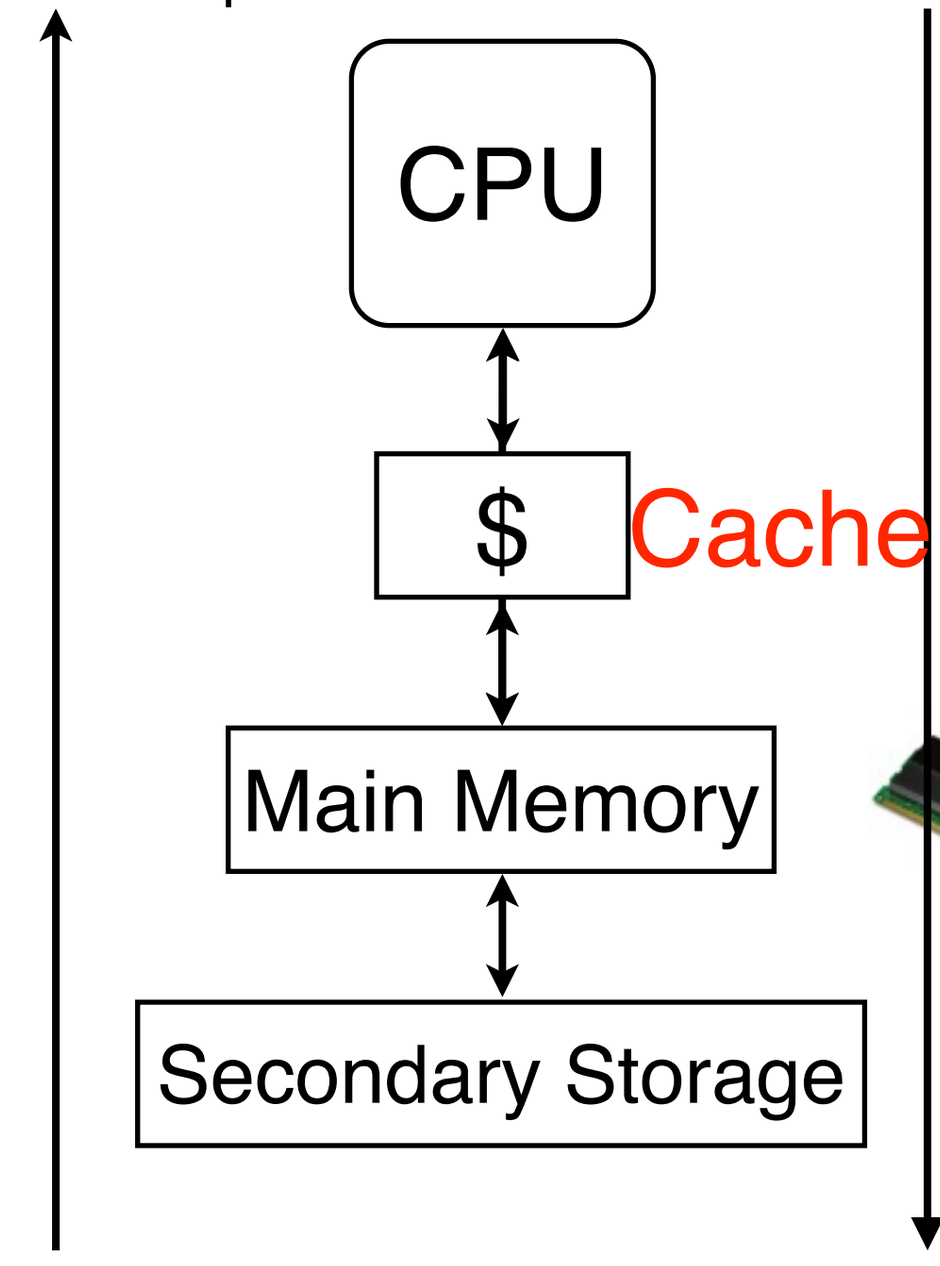
The access time of DDR3-1600 DRAM is around 50ns

100x to the cycle time of a 2GHz processor!

SRAM is as fast as the processor, but $$$

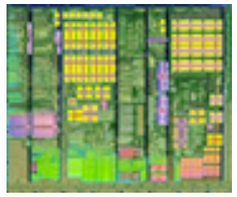
Memory hierarchy

Fastest,
Most Expensive

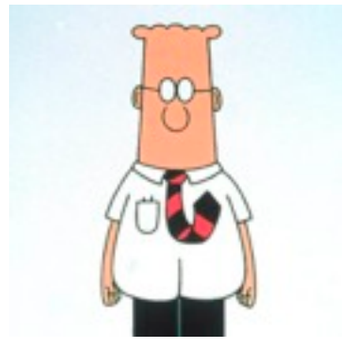


Biggest

Access time



< 1ns



< 1ns ~
20 ns

50-60ns

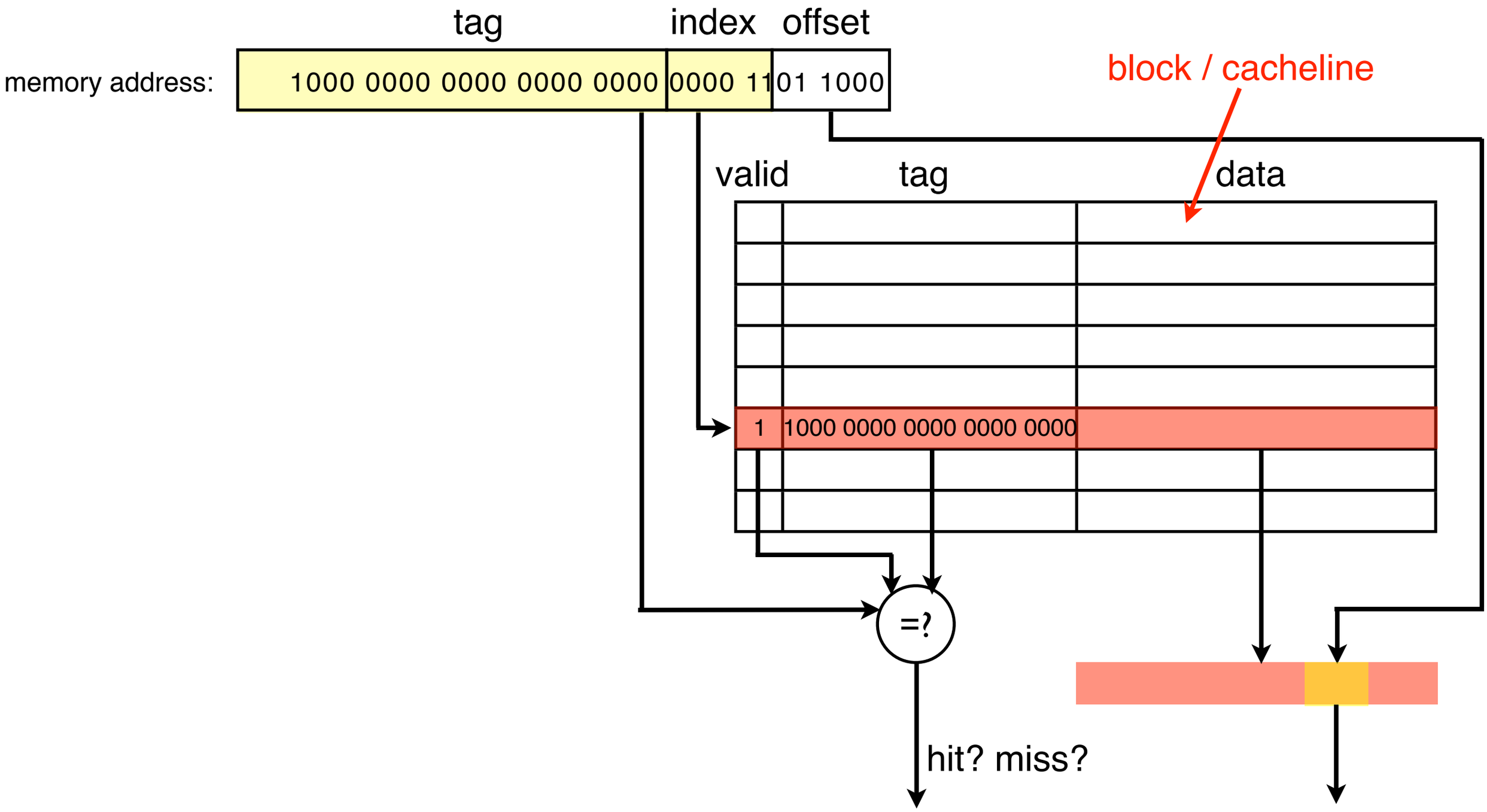
10,000,000ns

Cache organization

What is Cache?

- Cache is a hardware hash table!
  - each hash entry contains a block of data
    - caches operate on “blocks”
    - cache blocks are a power of 2 in size. Contains multiple words of memory
    - usually between 16B-128Bs
    - need lg(block_size) bits offset field to select the requested word/byte
  - hit: requested data is in the table
  - miss: requested data is not in the table
  - basic hash function:
    - block_address = byte_address/block_size
    - block_address % #_of_block

Accessing cache



Accessing cache

Tag:
the high order address bits stored along with the data to identify the actual address of the cache line.

Block (cacheline):
The basic unit of data in a cache.

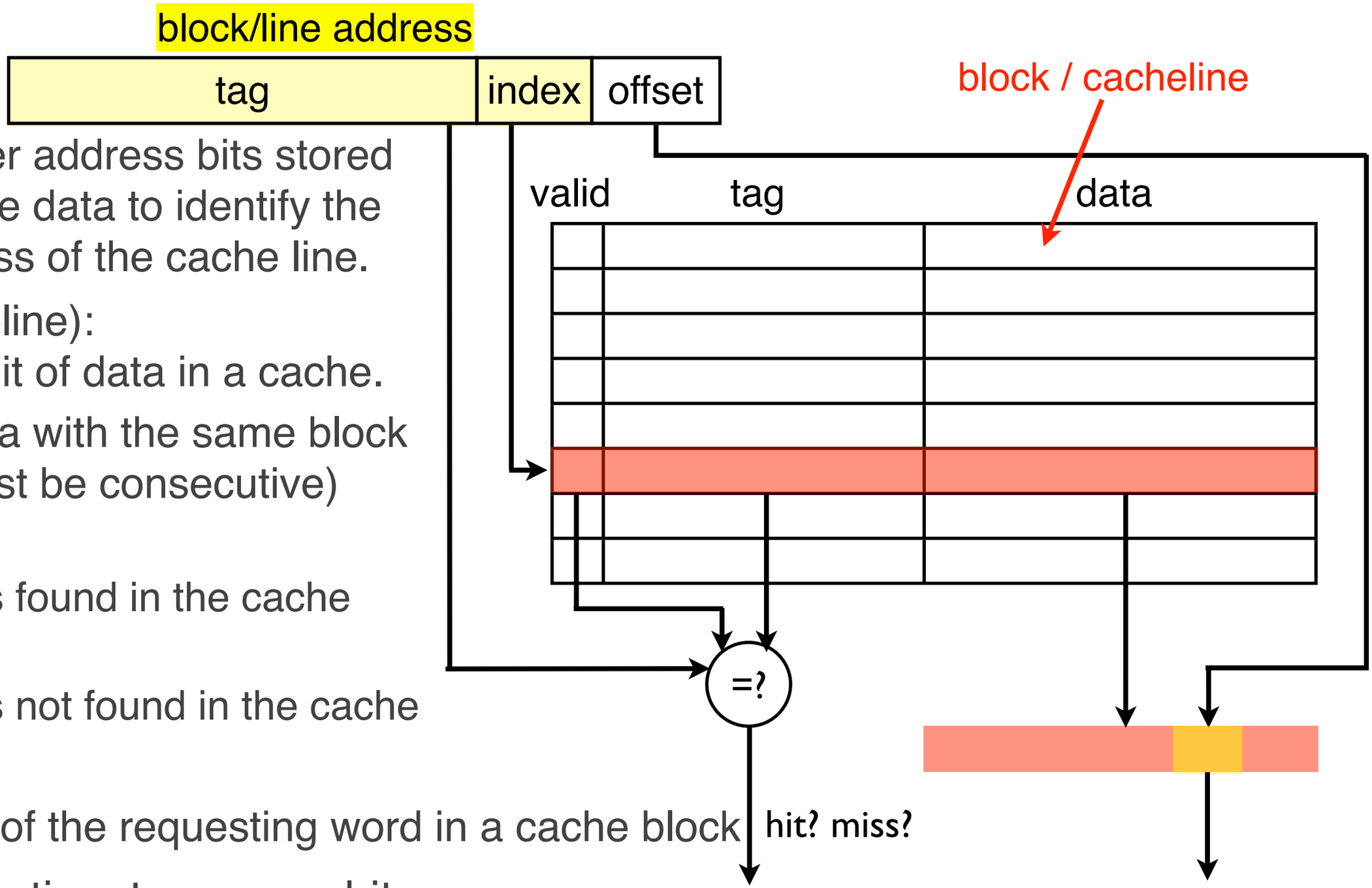
Contains data with the same block address (Must be consecutive)

Hit:
The data was found in the cache

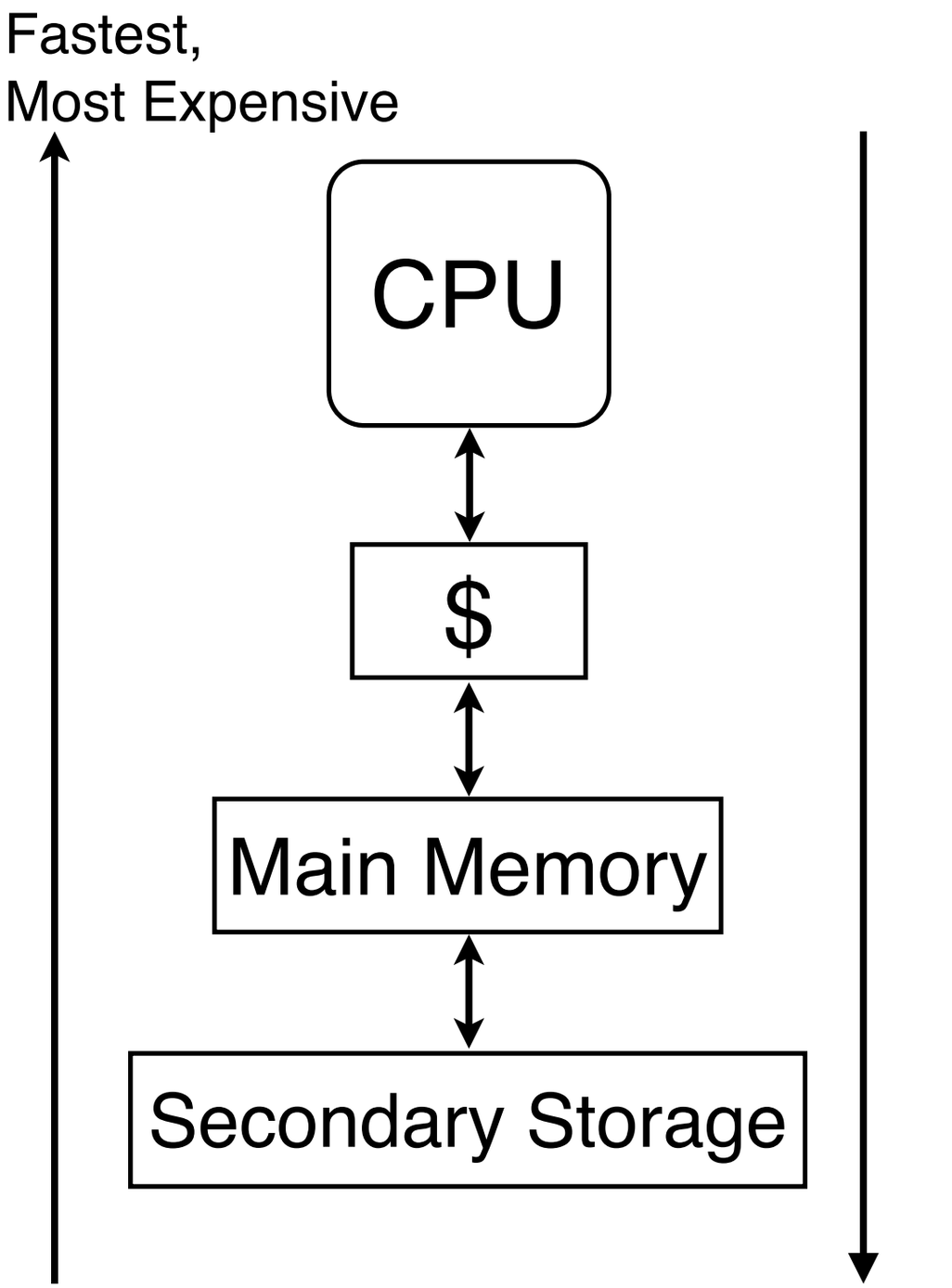
Miss:
The data was not found in the cache

Offset:
The position of the requesting word in a cache block

Hit time: The time to serve a hit



Locality

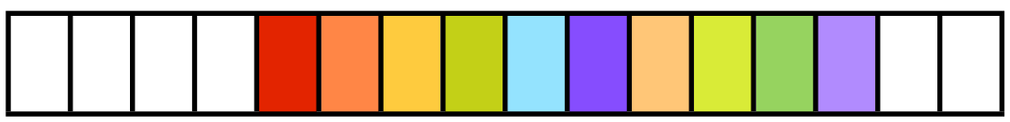


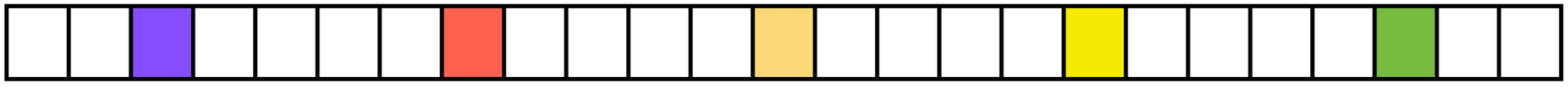
- Temporal Locality
  - Referenced item tends to be referenced again soon.
- Spatial Locality
  - Items close by referenced item tends to be referenced soon.
    - example: consecutive instructions, arrays

Demo revisited

for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
  c[i][j] = a[i][j] + b[i][j];
  }
}
	for(j = 0; j < ARRAY_SIZE; j++)
	{
	  for(i = 0; i < ARRAY_SIZE; i++)
	  {
	  c[i][j] = a[i][j] + b[i][j];
	  }
	}

Array_size = 1024, 0.048s (5.25X faster)	Array_size = 1024, 0.252s

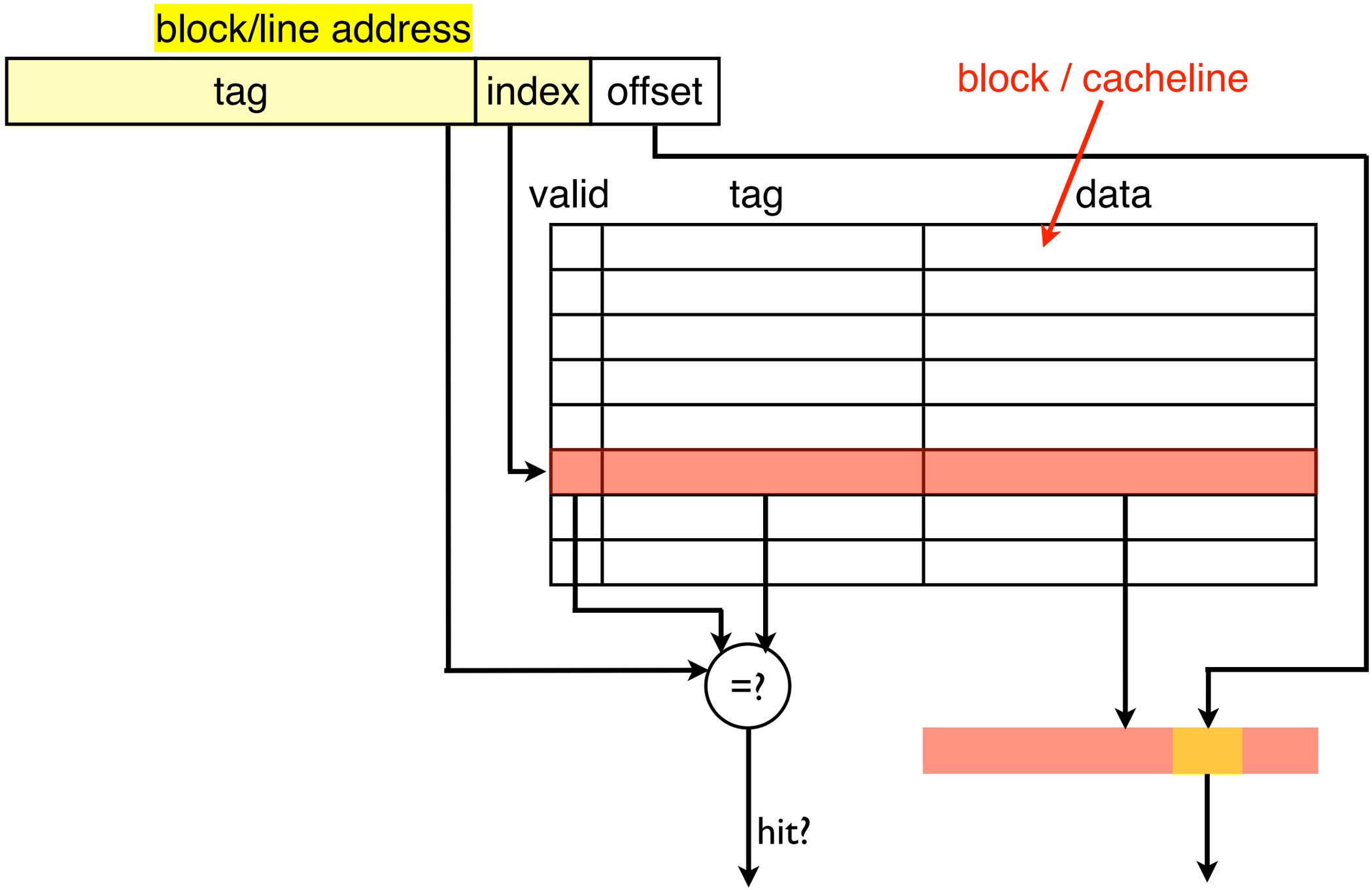




Data & Instruction caches

- Different area of memory
- Different access patterns
  - instruction accesses have lots of spatial locality
  - instruction accesses are predictable to the extent that branches are predictable
  - data accesses are less predictable
- Instruction accesses may interfere with data accesses
- Avoiding structural hazards in the pipeline
- Writes to I cache are rare

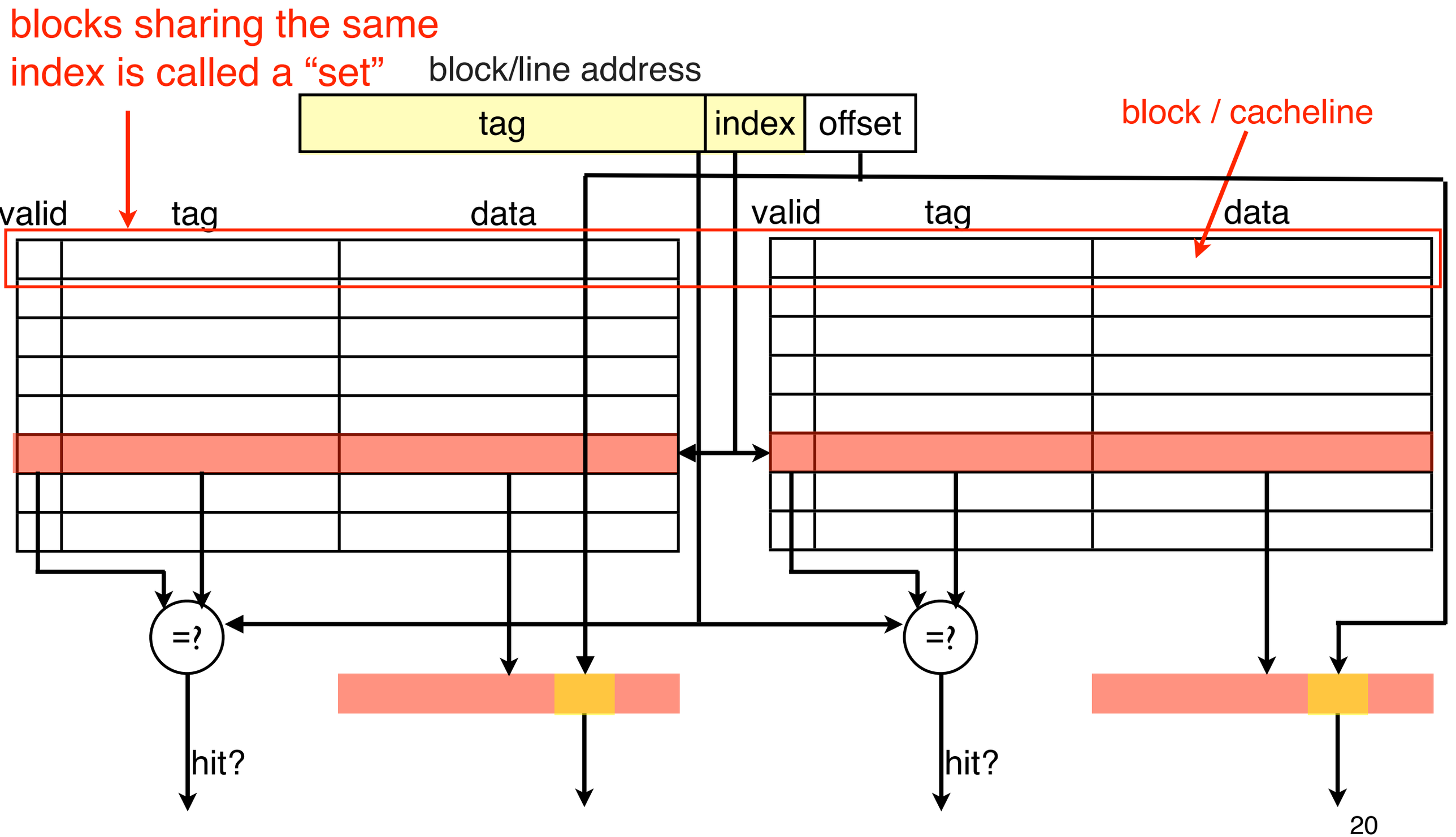
Basic organization of cache



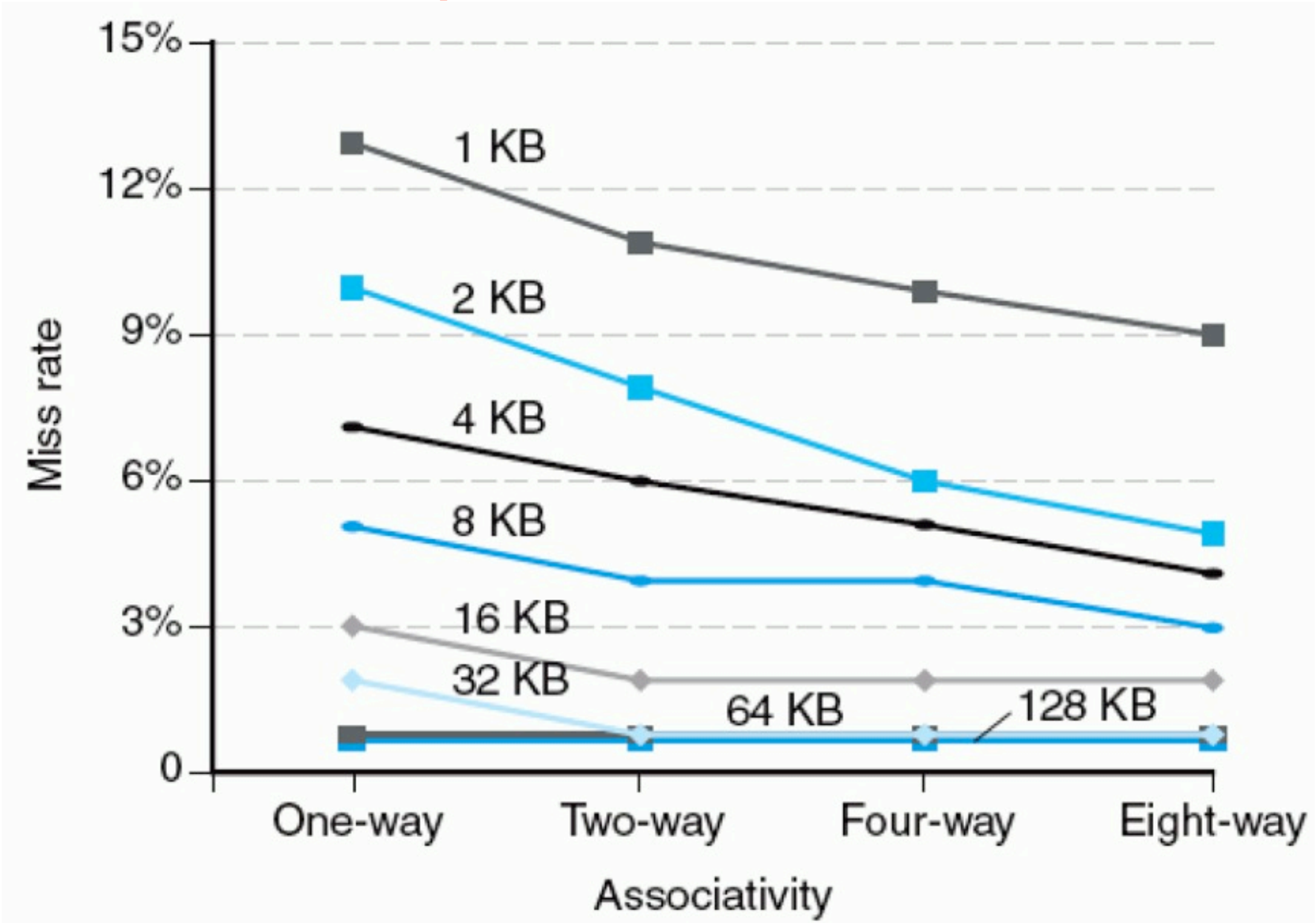
Way associativity

- Help alleviating the hash collision by having more blocks associating with each different index.
  - N-way associative: the block can be in N blocks of the cache
- Fully associative
  - The requested block can be anywhere in the cache
  - Or say N = the total number of cache blocks in the cache
- Increased associativity requires multiple tag checks
  - N-Way associativity requires N parallel comparators
  - This is expensive in hardware and potentially slow.
- This limits associativity L1 caches to 2-8.
- Larger, slower caches can be more associative

Way-associative cache



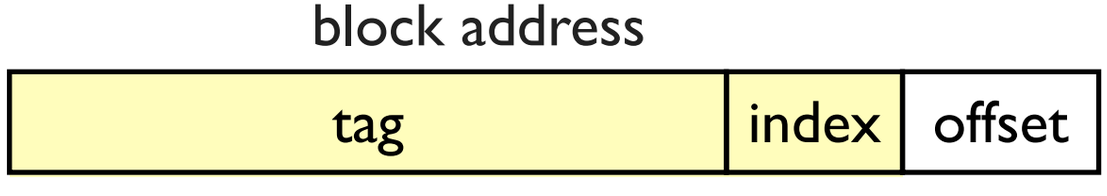
Way associativity and cache performance



C = ABS

- C = ABS
  - C: Capacity
  - A: Way-Associativity
    - How many blocks in a set
    - 1 for direct-mapped cache
  - B: Block Size (Cacheline)
    - How many bytes in a block
  - S: Number of Sets:
    - A set contains blocks sharing the same index
    - 1 for fully associate cache

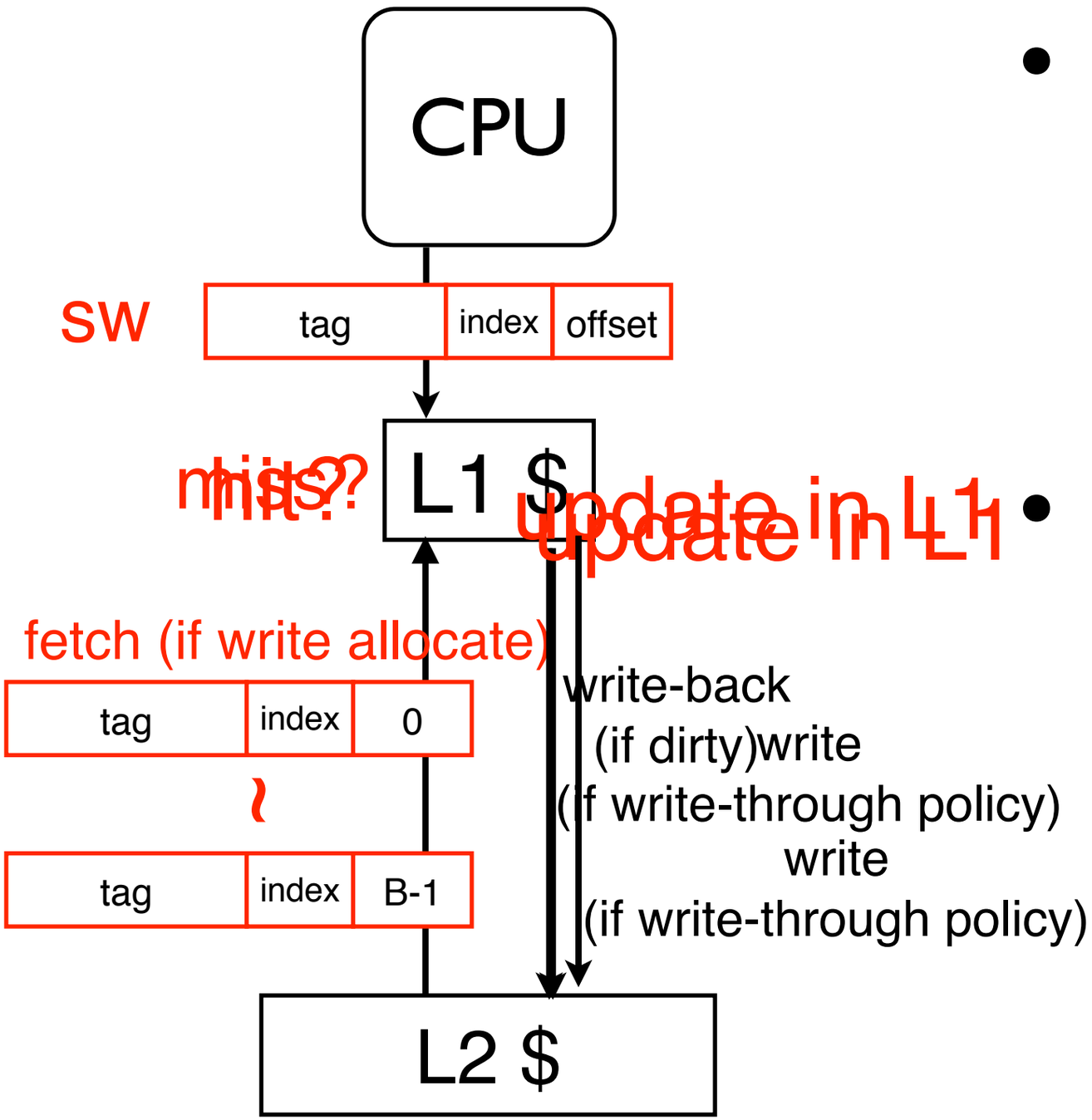
Corollary of C = ABS



- offset bits: lg(B)
- index bits: lg(S)
- tag bits: address_length - lg(S) - lg(B)
  - address_length is 32 bits for 32-bit machine
- (address / block_size) % S = set index

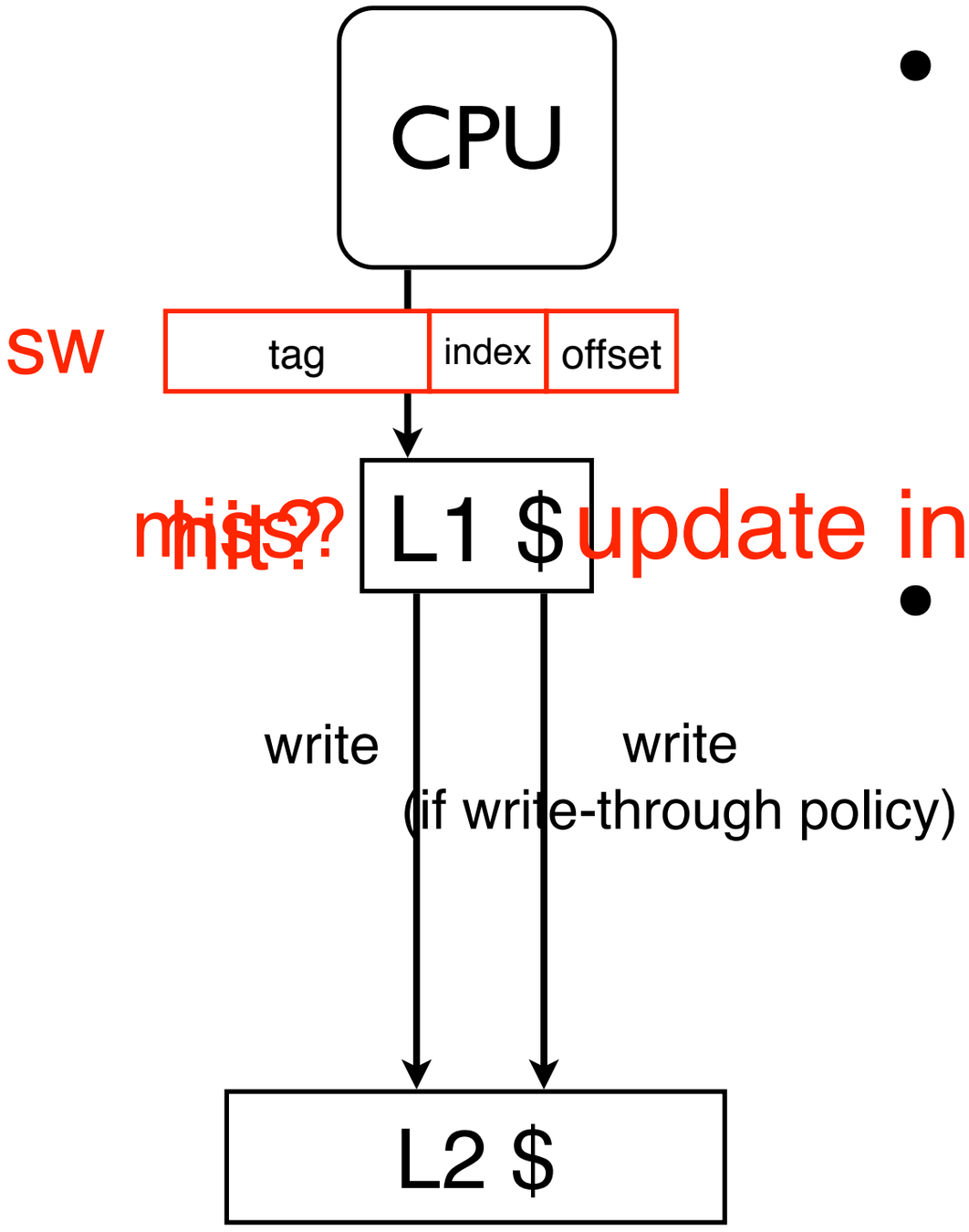
How cache works

What happens on a write? (Write Allocate)



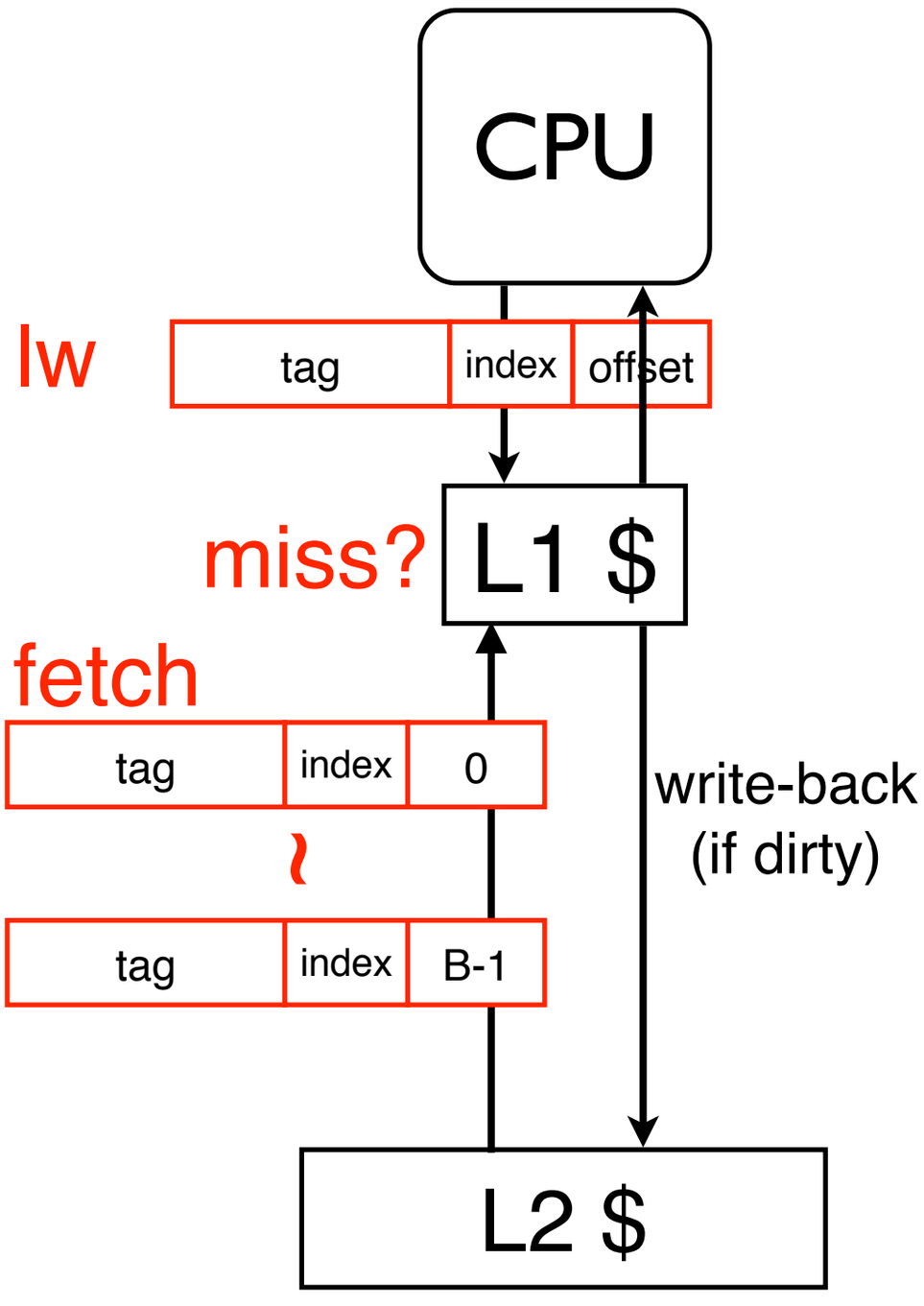
- Write hit?
  - Update in-place
  - Write to lower memory (Write-Through Policy)
  - Set dirty bit (Write-Back Policy)
- Write miss?
  - Select victim block
    - LRU, random, FIFO, ...
    - Write back if dirty
  - Fetch Data from Lower Memory Hierarchy
    - As a unit of a cache block
    - Miss penalty

What happens on a write? (No-Write Allocate)



- Write hit?
  - Update in-place
  - Write to lower memory (Write-Through only)
    - write penalty (can be eliminated if there is a buffer)
- Write miss?
  - Write to the first lower memory hierarchy has the data
    - Penalty

What happens on a read?



- Read hit
  - hit time
- Read miss?
  - Select victim block
    - LRU, random, FIFO, ...
    - Write back if dirty
- Fetch Data from Lower Memory Hierarchy
  - As a unit of a cache block
    - Data with the same “block address” will be fetch
  - Miss penalty

Evaluating cache performance

How to evaluate cache performance

- If the load/store instruction hits in L1 cache where the hit time is usually the same as a CPU cycle
  - The CPI of this instruction is the base CPI
- If the load/store instruction misses in L1, we need to access L2
  - The CPI of this instruction needs to include the cycles of accessing L2
- If the load/store instruction misses in both L1 and L2, we need to go to lower memory hierarchy (L3 or DRAM)
  - The CPI of this instruction needs to include the cycles of accessing L2, L3, DRAM

How to evaluate cache performance

- CPIAverage : the average CPI of a memory instruction

$$CPI_{Average} = CPI_{base} + miss\_rate_{L1} * miss\_penalty_{L1}$$

$$miss\_penalty_{L1} = CPI_{accessing\_L2} + miss\_rate_{L2} * miss\_penalty_{L2}$$

$$miss\_penalty_{L2} = CPI_{accessing\_L3} + miss\_rate_{L3} * miss\_penalty_{L3}$$

$$miss\_penalty_{L3} = CPI_{accessing\_DRAM} + miss\_rate_{DRAM} * miss\_penalty_{DRAM}$$

- If the problem (like those in your textbook) is asking for average memory access time, transform the CPI values into/from time by multiplying with CPU cycle time!

Average memory access time

- Average Memory Access Time (AMAT)
  = Hit Time+ Miss rate* Miss penalty
  - Miss penalty = AMAT of the lower memory hierarchy
  - AMAT = hit_time_{L1}+$miss\_rate_{L1}$*AMATL2
    - $AMAT_{L2}$ = hit_time_{L2}+$miss\_rate_{L2}$*$AMAT_{DRAM}$

Cause of cache misses

Cause of misses

- 3Cs of Cache miss
  - Compulsory miss
    - First access to a block
  - Capacity miss
    - The working set size of an application is bigger than cache size!
  - Conflict miss
    - Required data replaced by block(s) mapping to the same set

Cache simulation

- Consider a direct mapped cache with 16 blocks, a block size of 16 bytes, and the application repeat the following memory access sequence:
  - 0x80000000, 0x80000008, 0x80000010, 0x80000018, 0x30000010
- 16 = 2^4 : 4 bits are used for the index
- 16 = 2^4 : 4 bits are used for the byte offset
- The tag is 32 - (4 + 4) = 24 bits
- For example: 0x80000010

Cache simulation

	valid	tag	data
0	1	800000	
1	1	300000	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 miss: compulsory

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 hit!

0x80000008 hit!

0x80000010 miss: conflict

0x80000018 hit!

Cache simulation

- Consider a 2-way cache with 16 blocks (8 sets), a block size of 16 bytes, and the application repeat the following memory access sequence:
  - 0x80000000, 0x80000008, 0x80000010, 0x80000018, 0x30000010
  - 8 = 2^3 : 3 bits are used for the index
  - 16 = 2^4 : 4 bits are used for the byte offset
  - The tag is 32 - (3 + 4) = 25 bits
  - For example: 0b1000 0000 0000 0000 0000 0000 0001 0000

Cache simulation

	v	tag	data	v	tag	data
0	1	0x1000000				
1	1	0x1000000		1	0x600000	
2						
3						
4						
5						
6						
7						

0x80000000 miss: compulsory

0x80000008 hit!

0x80000010 miss: compulsory

0x80000018 hit!

0x30000010 miss: compulsory

0x80000000 hit!

0x80000008 hit!

0x80000010 hit!

0x80000018 hit!

3Cs and A, B, C

- Regarding 3Cs: compulsory, conflict and capacity misses and
  A, B, C: associativity, block size, capacity
  How many of the following are correct?
  - Increasing associativity can help reducing conflict misses
  - Increasing associativity can reducing hit time
  - Increasing block size can increase the miss penalty
  - Increasing block size can help reducing compulsory misses

A. 0

B. 1

C. 2

D. 3

E. 4

Improving 3Cs

Improvement of 3Cs

- 3Cs and A, B, C of caches
  - Compulsory miss
    - Increase B: increase miss penalty (more data must be fetched from lower hierarchy)
  - Capacity miss
    - Increase C: increase cost, access time, power
  - Conflict miss
    - Increase A: increase access time and power
- Or modify the memory access pattern of your program!

Live demo

- Live Demo
  - Matrix Multiplication
  - valgrind --tool=cachegrind cmd
    - cachegrind is a tool profiling the cache performance

```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

CSE101 tells you it's $O(n^3)$

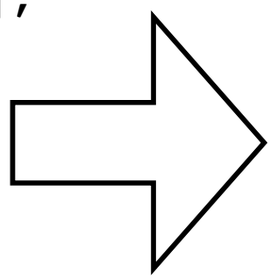
If n=512, it takes about 1 sec

How long is it take when n=1024?

Block algorithm for matrix multiplication

- Live Demo
  - Block Algorithm for Matrix Multiplication
  - valgrind --tool=cachegrind cmd
    - cachegrind is a tool profiling the cache performance

```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

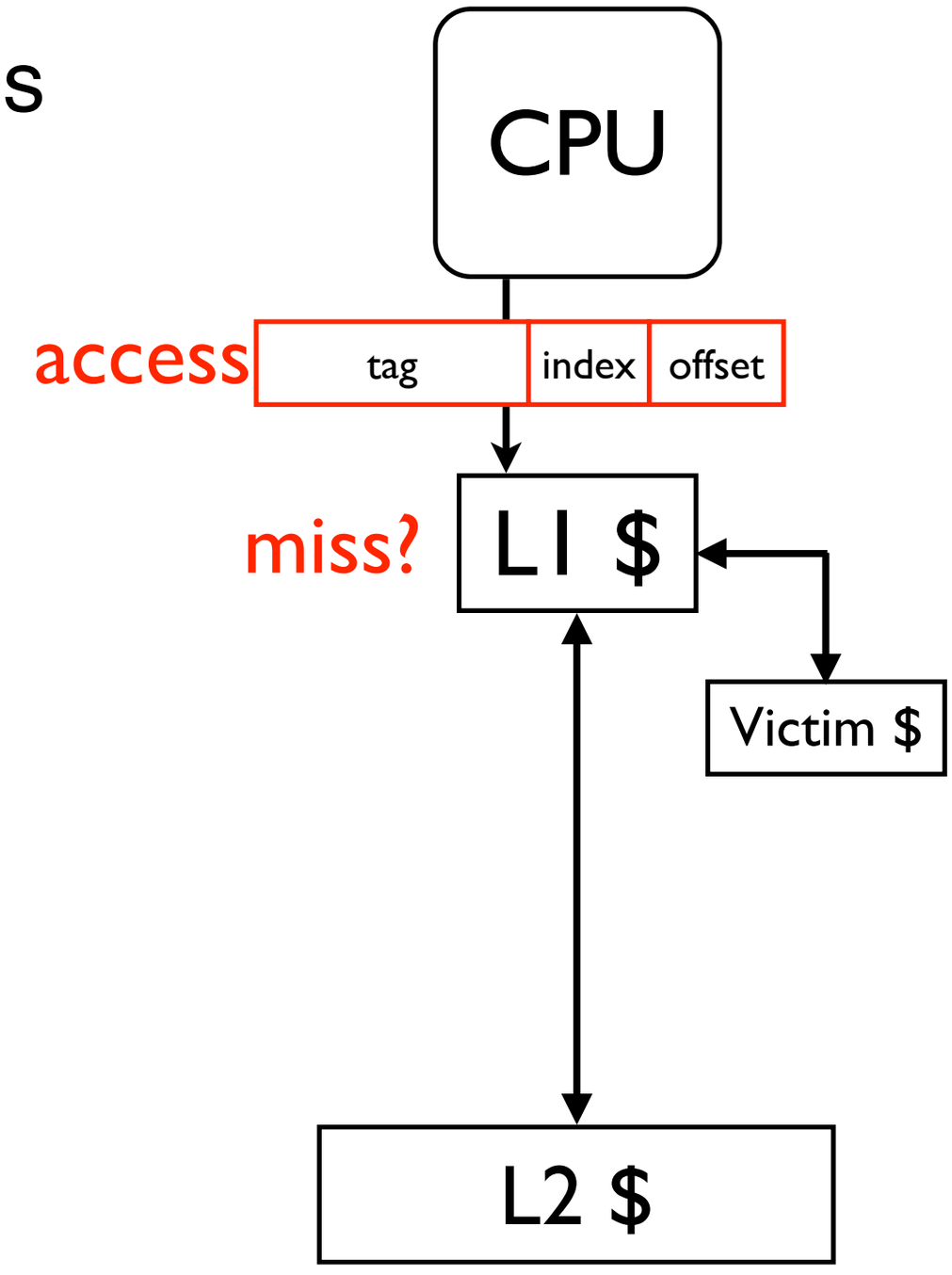


```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```

Other cache optimizations

Victim cache

- A small cache that captures the evicted blocks
  - Can be built as fully associative since it's small
  - Consult when there is a miss
  - Athlon has an 8-entry victim cache
- Reduce miss penalty of conflict misses



Prefetching

- Identify the access pattern and proactively fetch data before the application asks for.
  - Think about this code:

```
for(i = 0;i < 1000000; i++) {
    sum += data[i];
}
```

- Hardware prefetch:
  - The processor can keep track the distance between misses. If there is a pattern, fetch miss_data_address+distance for a miss.
- Software prefetching
  - Load data into $zero
  - Using prefetch instructions
- Reduce compulsory misses

Write buffer

- Every write to lower memory will first write to a small SRAM buffer.
  - The write buffer will continue writing data to lower-level memory
  - The processor/higher-level memory can response as soon as the data is written to write buffer.
- Help reduce miss penalty
- Help improve write through performance
- Write merge
  - Since application has locality, it's highly possible the evicted data have neighboring addresses. Write buffer delays the writes and allows these neighboring data to be grouped together.

Q & A