Memory Hierarchy

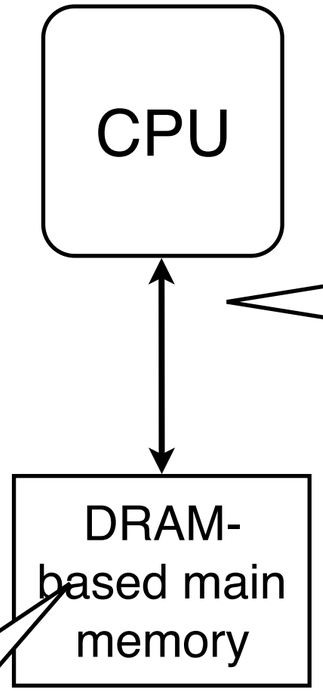
Hung-Wei Tseng

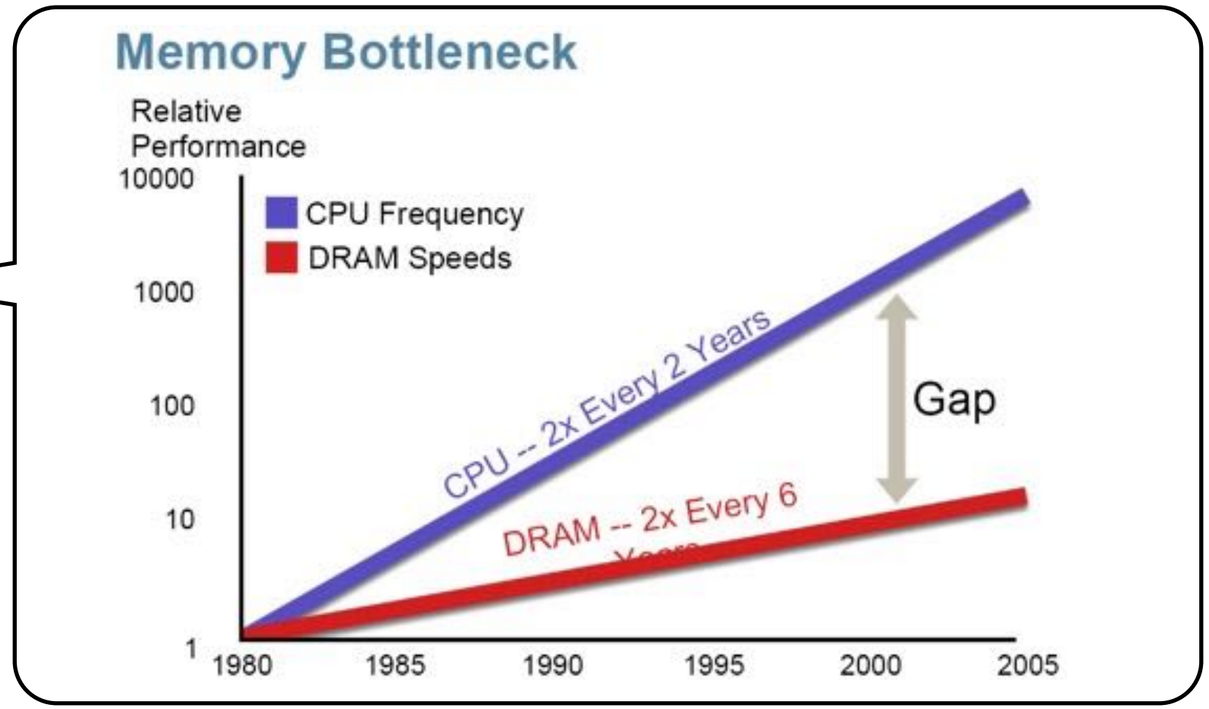
Outline

- Memory wall/gap problem
- Memory hierarchy
- Cache organization

Memory wall problem

The memory gap problem






```
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
```

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

The access time of DRAM is around 50ns

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

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

Why is C better than B

- How many of the following statements explains the reason why B outperforms C with compiler optimizations
  - ① C has lower dynamic instruction count than B
  - ② C has significantly lower branch mis-predictions than B
  - ③ C has significantly fewer branch instructions than B
  - ④ C can incur fewer data hazards

A. 0

B. 1

C. 2

D. 3

E. 4

— C only needs one load, one add, one shift, the same amount of iterations

— the same number being predicted.

— the same amount of branches

— Probably not. In fact, the load may have negative effect without architectural supports

Does this make sense if memory is so slow?

B

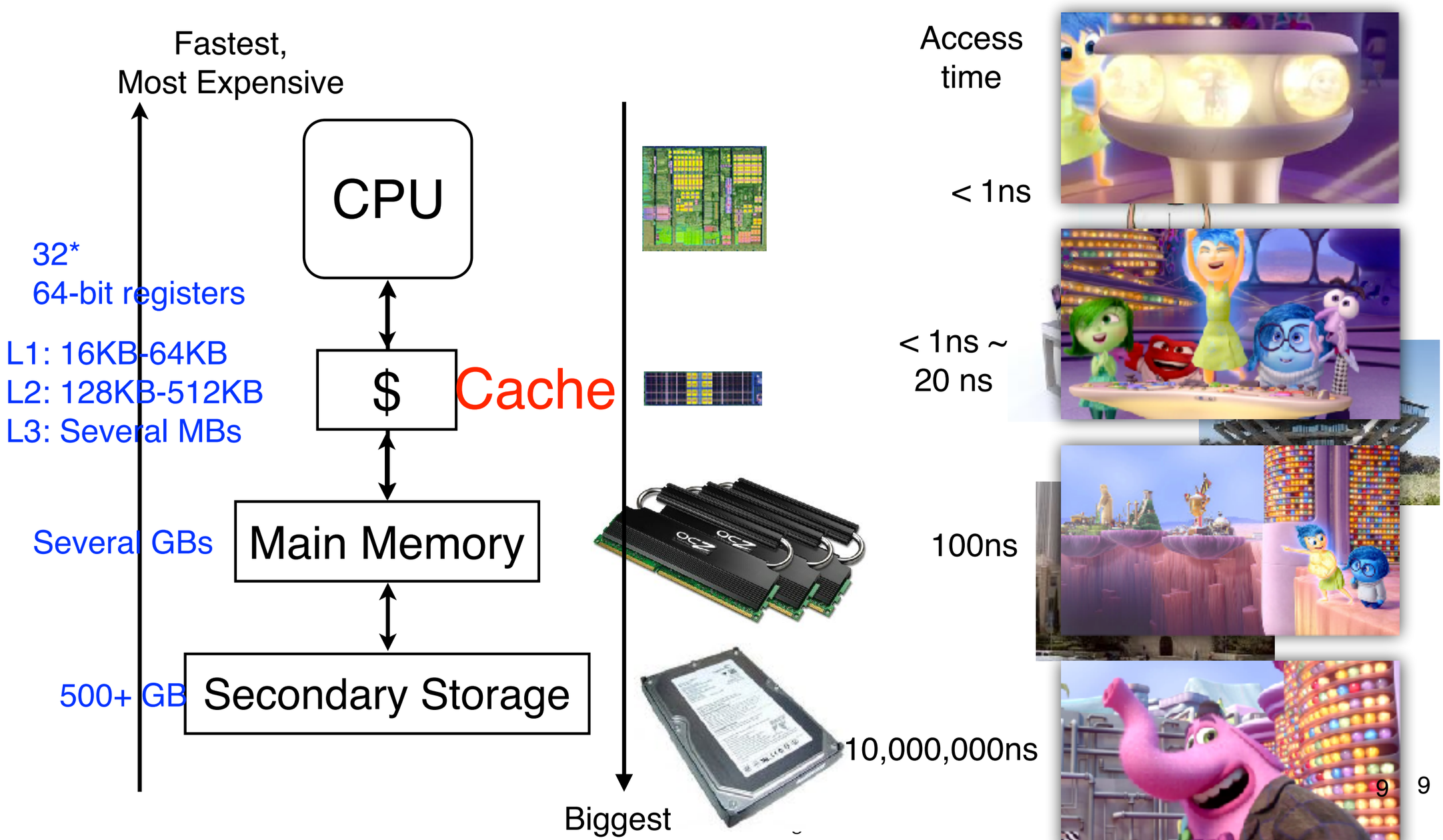
```
inline int popcount(uint64_t x){
   int c=0;
    while(x)     {
       c += x & 1;
       x = x >> 1;
       c += x & 1;
       x = x >> 1;
       c += x & 1;
       x = x >> 1;
       c += x & 1;
       x = x >> 1;
     }
    return c;
}
```

C

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2,
     while(x)     {
          c += table[(x & 0xF)];
          x = x >> 4;
     }
     return c;
}
```

Memory hierarchy

The memory hierarchy



Why can a small, fast SRAM help?

Localities in your code

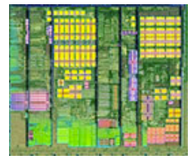
- Spatial locality: programs tend to access neighboring data/ instructions
  - Data structures (e.g. arrays) demonstrate strong spatial locality
  - Especially effective for code/instructions — you usually just move to the next instruction or loop back to the small piece of code
- Temporal locality: programs tend to have frequently accessed data
  - You may update/reference the same set of memory locations many times in your code

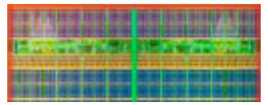
Cache organization

Architecting caches to capture localities

- To capture spatial locality
  - We need to put not only just a “word” or small piece of data/instructions, but a “block” of data/instructions
- To capture temporal locality
  - We need to keep frequently used data

Organizing memory locations into blocks





Architecting caches to capture localities

- To capture spatial locality
  - We need to put not only just a “word” or small piece of data/instructions, but a “block” of data/instructions
  - **How to distinguish each block?**
- To capture temporal locality
  - We need to keep frequently used data

How do you make a cheatsheet?

- Go through your homework
- Write down the topic and content
- If running out of space: kick out the least recently used content

1. Performance equation
2. Amdahl's law
3. MIPS
4. Power consumption
5. Performance equation
6. Amdahl's law
7. MFLOPS

Tag: the address prefix of data in the cacheline/block

Performance equation	ET=IC*CPI*CT
Amdahl's law	ET_after = ET_affected/Speedup + ET_unaffected
MFLOPS	MIPS = IC/(ET*10^6)
Power consumption	P = aCV²f

Cacheline/block: data with the same prefix in their addresses

A simple cache: now with tags associated with blocks

- Assume each block contains 16B data
- A total of 4 blocks

tag	data
0b0000	content of 0b00000000 - 0b00001111
0b0100	content of 0b01000000 - 0b01001111
0b1100	content of 0b11000000 - 0b11001111
0b1111	content of 0b11110000 - 0b11111111

Architecting caches to capture localities

- To capture spatial locality
  - We need to put not only just a “word” or small piece of data/instructions, but a “block” of data/instructions
  - A tag associated with each block
- To capture temporal locality
  - A cache replacement policy to keep most frequently used data (e.g. LRU)
- LRU — kick out the least recently used block when we need to kick out one

A simple cache: a block can go anywhere

- Assume each block contains 16B data
- A total of 4 blocks
- LRU — kick out the least recently used whenever we need to

1. 0x4 0b00000100
2. 0x48 0b01001000
3. 0xC4 0b11000100
4. 0xFC 0b11111100
5. 0x12 0b00001100
6. 0x44 0b01000100
7. 0x68 0b01100100

tag	data
0b0000	content of 0b00000000 - 0b00001111
0b0100	content of 0b01000000 - 0b01001111
0b0100	content of 0b**01**000000 - 0b**01**001111
0b1111	content of 0b11110000 - 0b11111111

- **Too slow if the number of entries/blocks/cachelines is huge**

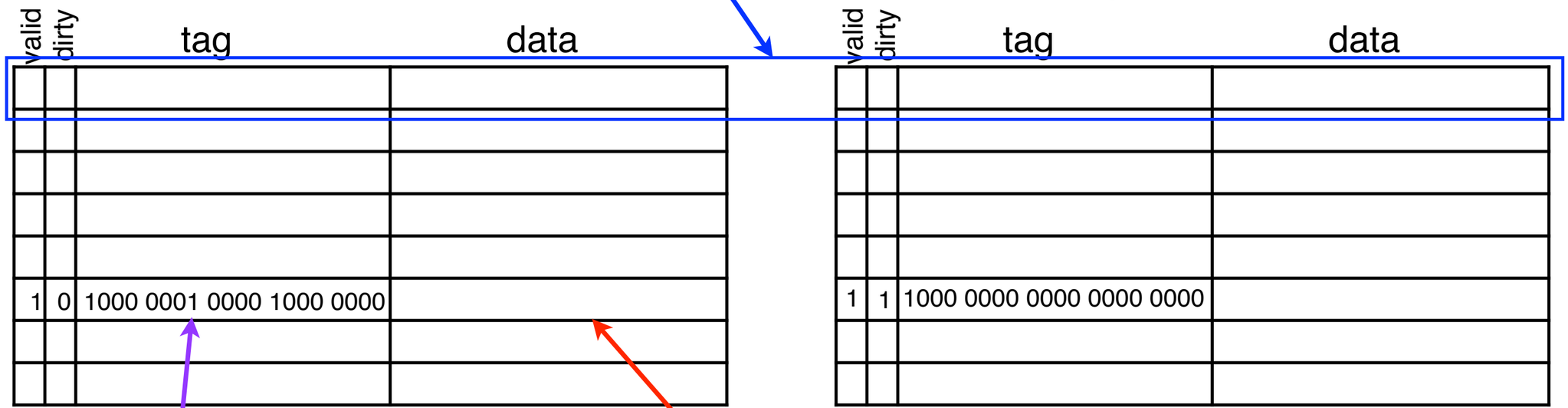
Architecting caches to capture localities

- To capture spatial locality
  - We need to put not only just a “word” or small piece of data/instructions, but a “block” of data/instructions
  - A tag associated with each block
- To capture temporal locality
  - A cache replacement policy to keep most frequently used data (e.g. LRU)
    - LRU — kick out the least recently used block when we need to kick out one
- Performance needs to be better than linear search
  - Make cache a hardware hash table!
  - The hash function takes memory addresses as inputs

The structure of a cache

Set: cache blocks/lines sharing the same index.
A cache is called N-way set associative cache if N blocks share the same set/index (this one is a 2-way set cache)

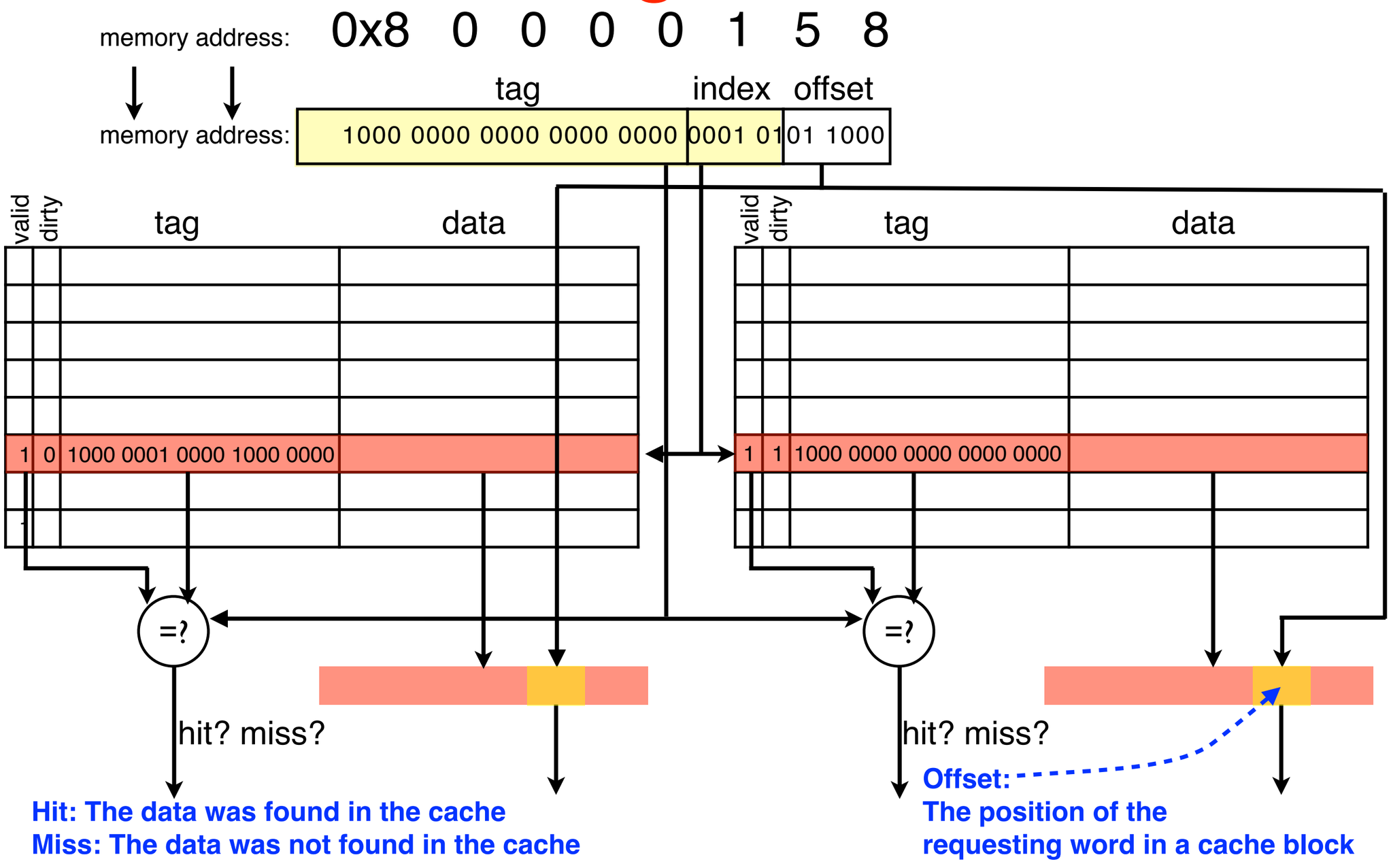
valid: if the data is meaningful
dirty: if the block is modified



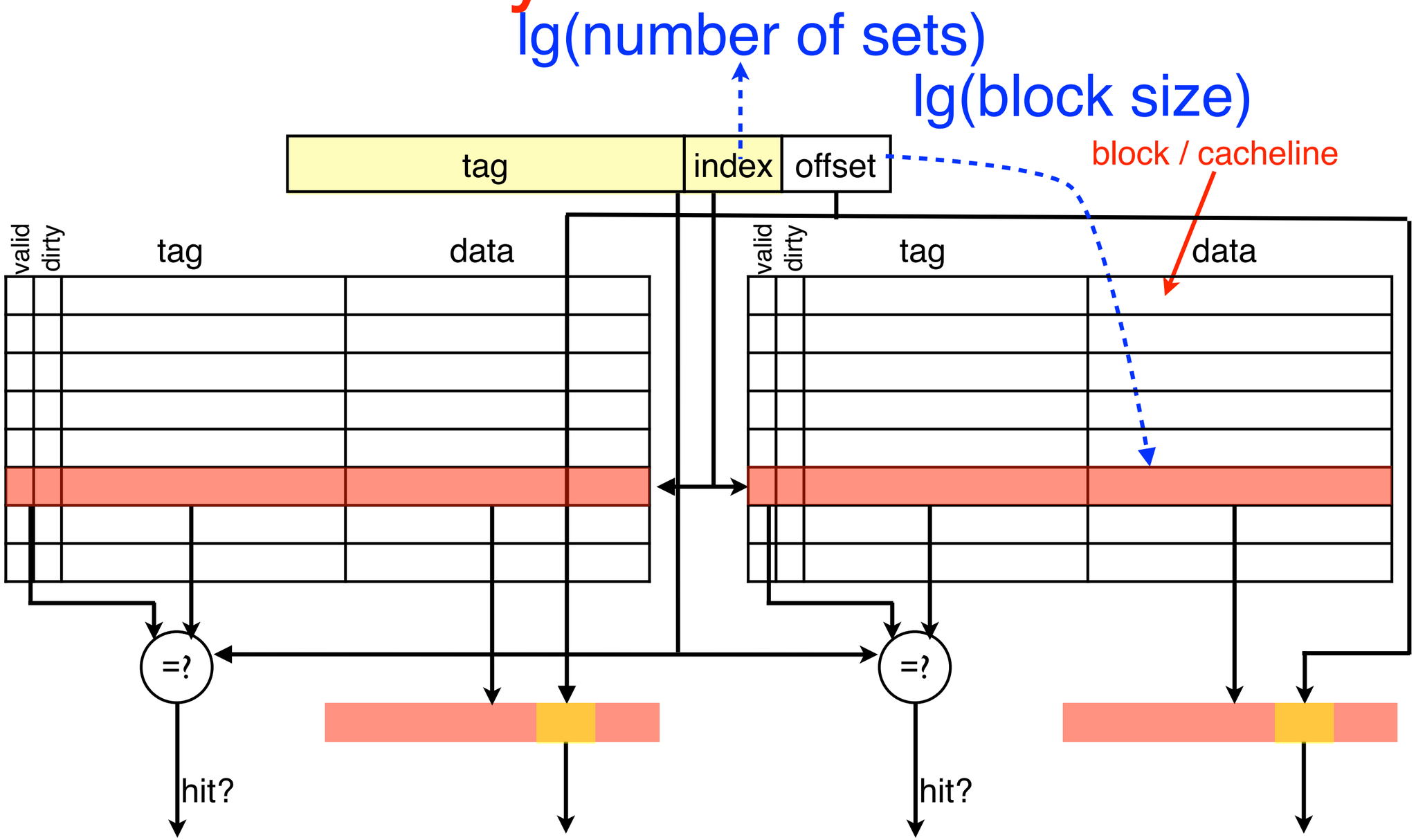
Block / Cacheline: The basic unit of data storage in cache. Contains all data with the same tag/prefix and index in their memory addresses

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

Accessing the cache



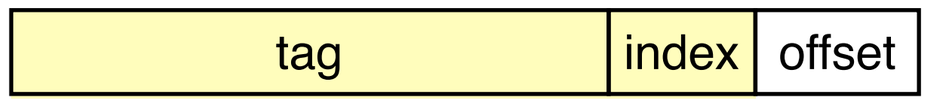
How many bits in each field?



C = ABS

- C: Capacity in data arrays
- A: Way-Associativity
  - N-way: N blocks in a set, A = N
  - 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

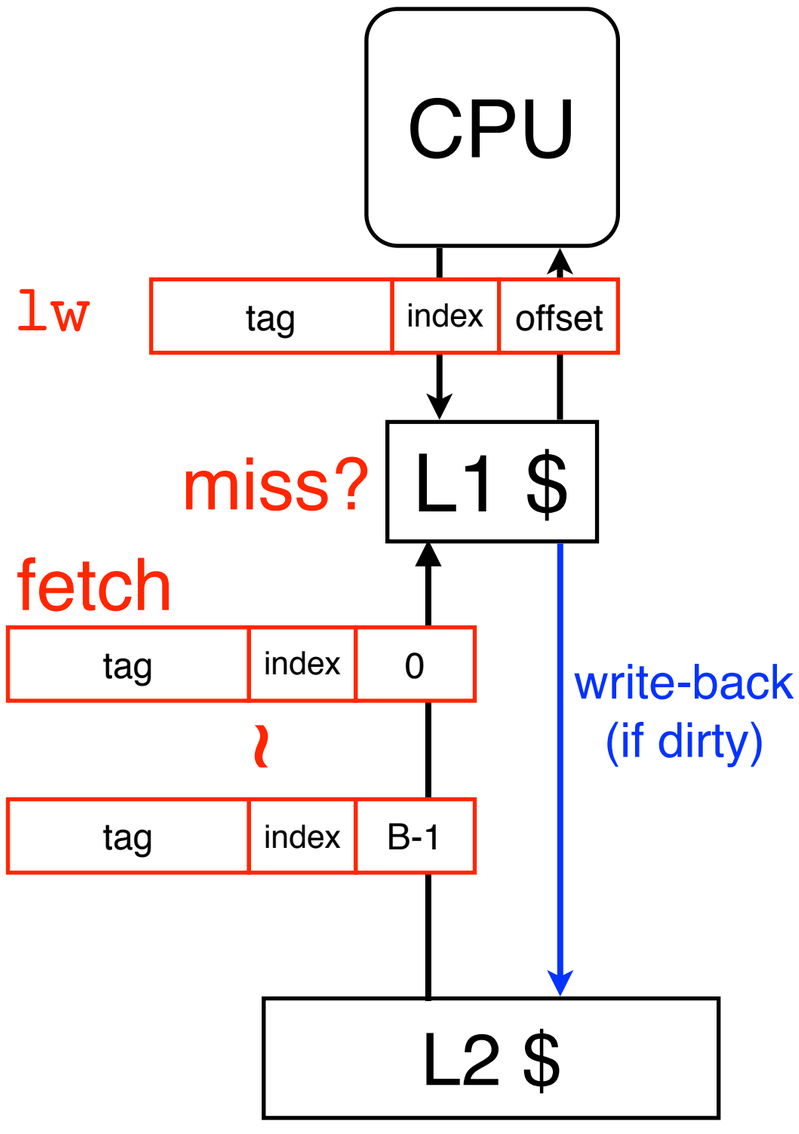


- offset bits: lg(B)
- index bits: lg(S)
- tag bits: address_length - lg(S) - lg(B)
  - address_length is 32 bits for 32-bit machine
- (address / block_size) % S = set index

Put everything all together:

How cache interacts with CPU

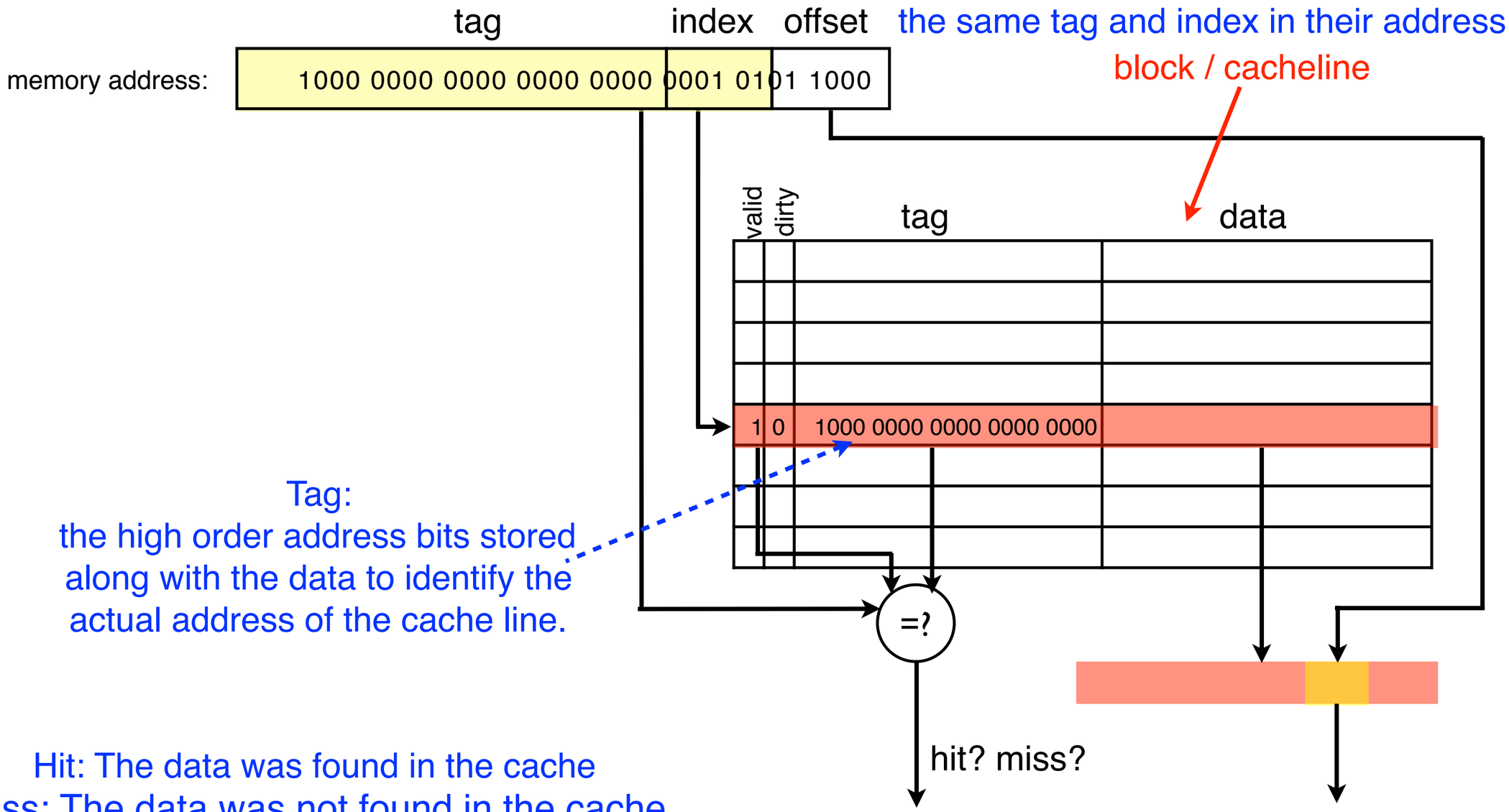
What happens on a read?



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

Special case: a direct-mapped cache

Block (cacheline): The basic unit of data storage in cache. Contains all data with the same tag and index in their address



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

Hit: The data was found in the cache
Miss: The data was not found in the cache

Simulate a direct-mapped cache

- Consider a direct mapped (1-way) cache with 16 blocks, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000
  - C = A B S
  - S=256/(16*1) = 16
  - lg(16) = 4 : 4 bits are used for the index
  - lg(16) = 4 : 4 bits are used for the byte offset
  - The tag is 48 - (4 + 4) = 40 bits
  - For example: 0b1000 0000 0000 0000 0000 0000 1000 0000

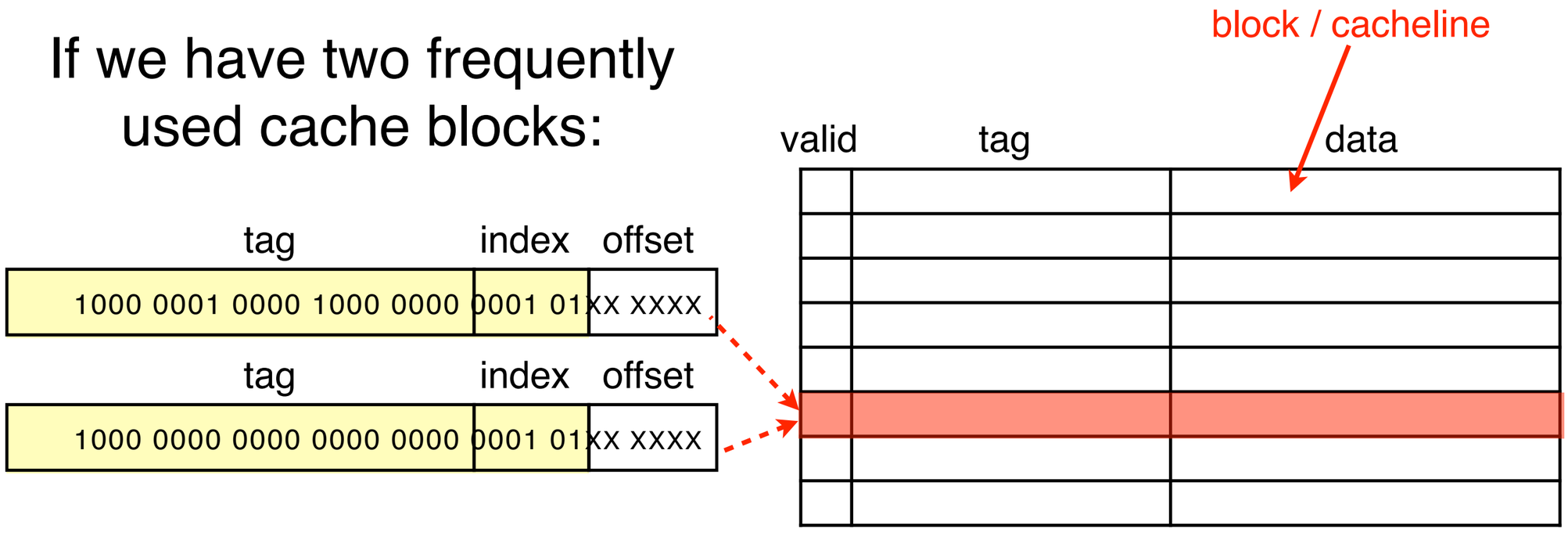
Simulate a direct-mapped cache

	valid	tag	data
0	1	0b10	
1	1	0b10	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

tag	index		
0b10	0000	0000	miss
0b10	0000	1000	hit!
0b10	0001	0000	miss
0b10	0001	0100	hit!
0b11	0001	0000	miss
0b10	0000	0000	hit!
0b10	0000	1000	hit!
0b10	0001	0000	miss
0b10	0001	0100	hit!

Conflict in direct-mapped cache

If we have two frequently used cache blocks:



If they are usually used back-to-back, one will kick out the other all the time

Simulate a 2-way cache

- Consider a 2-way cache with 16 blocks (8 sets), a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000

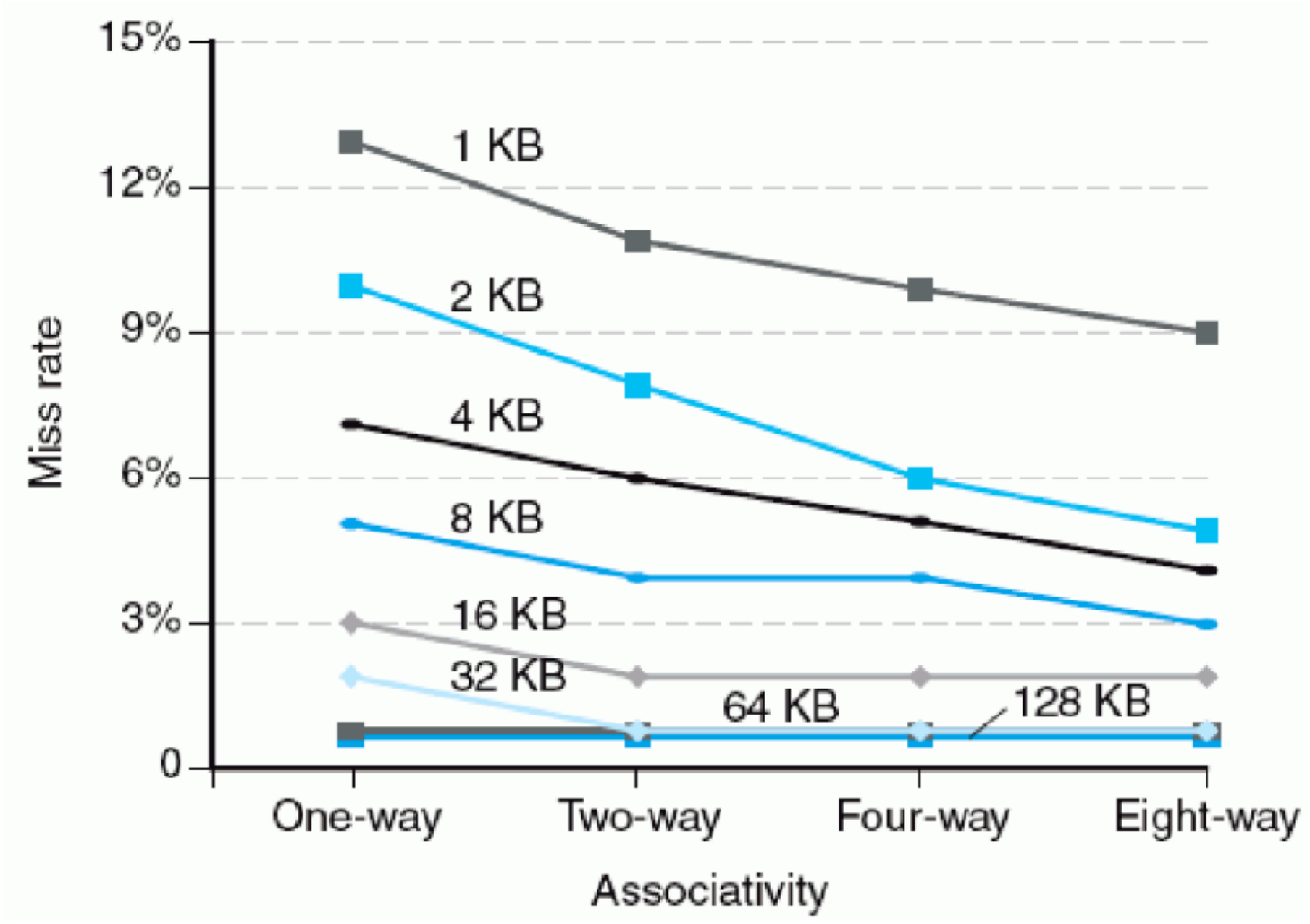
- 8 = 2^3 : 3 bits are used for the index
- 16 = 2^4 : 4 bits are used for the byte offset
- The tag is 32 - (3 + 4) = 25 bits
- For example: 0b1000 0000 0000 0000 0000 0000 0001 0000

Simulate a 2-way cache

	v	tag	data	v	tag	data
0	1	0b100				
1	1	0b100		1	0b110	
2						
3						
4						
5						
6						
7						

	tag	index		
0b	10	0000	0000	miss
0b	10	0000	1000	hit!
0b	10	0001	0000	miss
0b	10	0001	0100	hit!
0b	11	0001	0000	miss
0b	10	0000	0000	hit!
0b	10	0000	1000	hit!
0b	10	0001	0000	hit!
0b	10	0001	0100	hit!

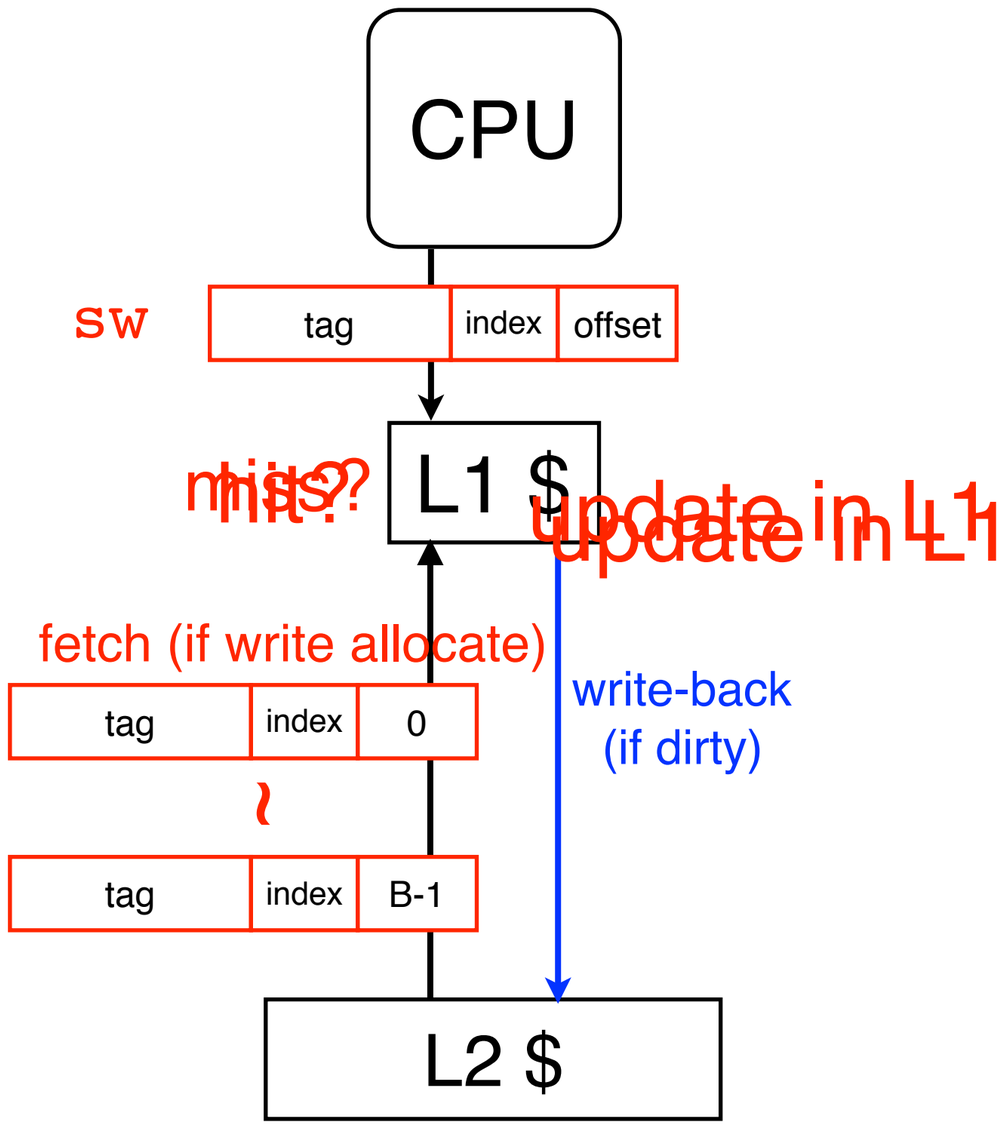
Way associativity and cache performance



Pros & cons of way-associate caches

- 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
- Slower
  - Increasing 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

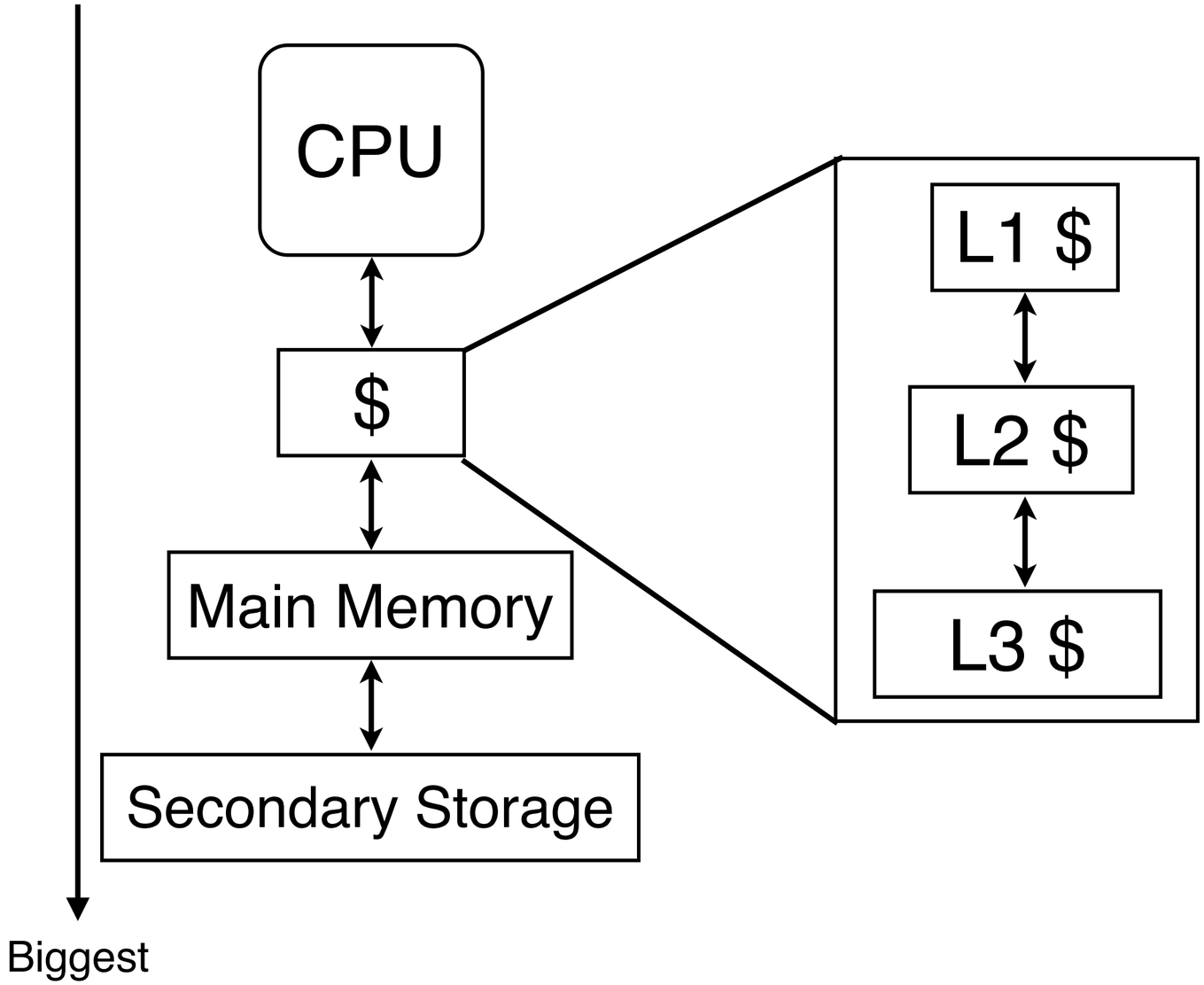
What happens on a write? (Write Allocate, write back)



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

Performance evaluation considering cache

Multi-layer caches



- Speed of L1 matches the processor
- Caches data/code as many as possible in L2/ L3 to avoid DRAM accesses

Performance evaluation considering cache

- 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

- $CPI_{Average}$: 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 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

3Cs of misses

- Compulsory miss
  - Cold start miss. First-time 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
  - Similar collision in hash

Simulate a 2-way cache

- Consider a 2-way cache with 16 blocks (8 sets), a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000

- 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

Simulate a 2-way cache

	v	tag	data	v	tag	data
0	1	0b100				
1	1	0b100		1	0b110	
2						
3						
4						
5						
6						
7						

	tag index	
0b10 0000 0000		**compulsory miss**
0b10 0000 1000		hit!
0b10 0001 0000		compulsory miss
0b10 0001 0100		hit!
0b11 0001 0000		compulsory miss
0b10 0000 0000		hit!
0b10 0000 1000		hit!
0b10 0001 0000		hit!
0b10 0001 0100		hit!

Simulate a direct-mapped cache

- Consider a direct mapped (1-way) cache with 16 blocks, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000

- 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: 0b1000 0000 0000 0000 0000 0000 1000 0000

Simulate a direct-mapped cache

	valid	tag	data
0	1	0b10	
1	1	0b10	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

tag	index		
0b10	0000	0000	**compulsory miss**
0b10	0000	1000	hit!
0b10	0001	0000	compulsory miss
0b10	0001	0100	hit!
0b11	0001	0000	compulsory miss
0b10	0000	0000	hit!
0b10	0000	1000	hit!
0b10	0001	0000	**conflict miss**
0b10	0001	0100	hit!

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!

Memory hierarchy and your code

Demo

```
#ifndef COL_MAJOR
     for(i = 0; i < ARRAY_SIZE; i++)
     {
       for(j = 0; j < ARRAY_SIZE; j++)
       {
         c[i][j] = a[i][j]+b[i][j];
       }
     }
#else
     for(j = 0; j < ARRAY_SIZE; j++)
     {
       for(i = 0; i < ARRAY_SIZE; i++)
       {
         c[i][j] = a[i][j]+b[i][j];
       }
     }
#endif
```

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

Array of structures or structure of arrays

	Array of objects	object of arrays

	`struct grades`
	`{`
	`  int id;`
	`  double *homework;`
	`  double average;`
	`};`
		`struct grades`
		`{`
		`  int *id;`
		`  double **homework;`
		`  double *average;`
		`};`

average of each homework	`for(i=0;i<homework_items; i++)`
	`{`
	`gradesheet[total_number_students].homework[i] = 0.0;`
	`  for(j=0;j<total_number_students;j++)`
	`gradesheet[total_number_students].homework[i]`
	`+=gradesheet[j].homework[i];`
	`  gradesheet[total_number_students].homework[i] /=`
	`(double)total_number_students;`
	`}`
		`for(i = 0;i < homework_items; i++)`
		`{`
		`  gradesheet.homework[i][total_number_students] = 0.0;`
		`  for(j = 0; j <total_number_students;j++)`
		`  {`
		`  gradesheet.homework[i][total_number_students] +=`
		`gradesheet.homework[i][j];`
		`  }`
		`  gradesheet.homework[i][total_number_students] /=`
		`total_number_students;`
		`}`

Column-store or row-store

- If you're designing an in-memory database system, will you be using

RowId	EmpId	Lastname	Firstname	Salary
1	10	Smith	Joe	40000
2	12	Jones	Mary	50000
3	11	Johnson	Cathy	44000
4	22	Jones	Bob	55000

  - column-store — stores data tables column by column

```
10:001,12:002,11:003,22:004;
Smith:001,Jones:002,Johnson:003,Jones:004;
Joe:001,Mary:002,Cathy:003,Bob:004;
40000:001,50000:002,44000:003,55000:004;
```

  - row-store — stores data tables row by row

```
001:10,Smith,Joe,40000;
002:12,Jones,Mary,50000;
003:11,Johnson,Cathy,44000;
004:22,Jones,Bob,55000;
```

```
select Lastname, Firstname from table
```

Case study: Matrix Multiplication

- Matrix Multiplication

```
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];
    }
  }
}
```

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

If n=512, it takes about 1 sec

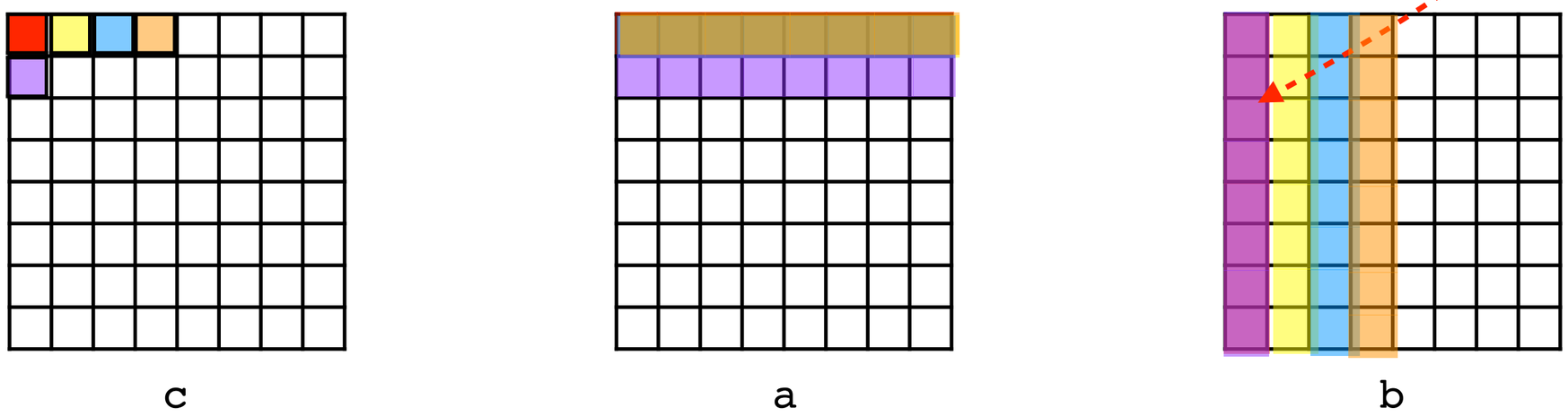
How long is it take when n=1024?

Matrix Multiplication

- Matrix Multiplication

```
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];
    }
  }
}
```

Very likely a miss if array is large



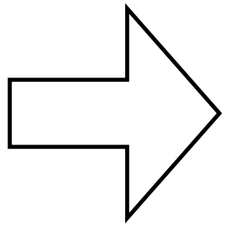
- If each dimension of your matrix is 1024
  - Each row takes 1024*8 bytes = 8KB
  - The L1 $ of intel Core i7 is 32KB, 8-way, 64-byte blocked
  - You can only hold at most 4 rows/columns of each matrix!
  - You need the same row when j increase!

Block algorithm for matrix multiplication

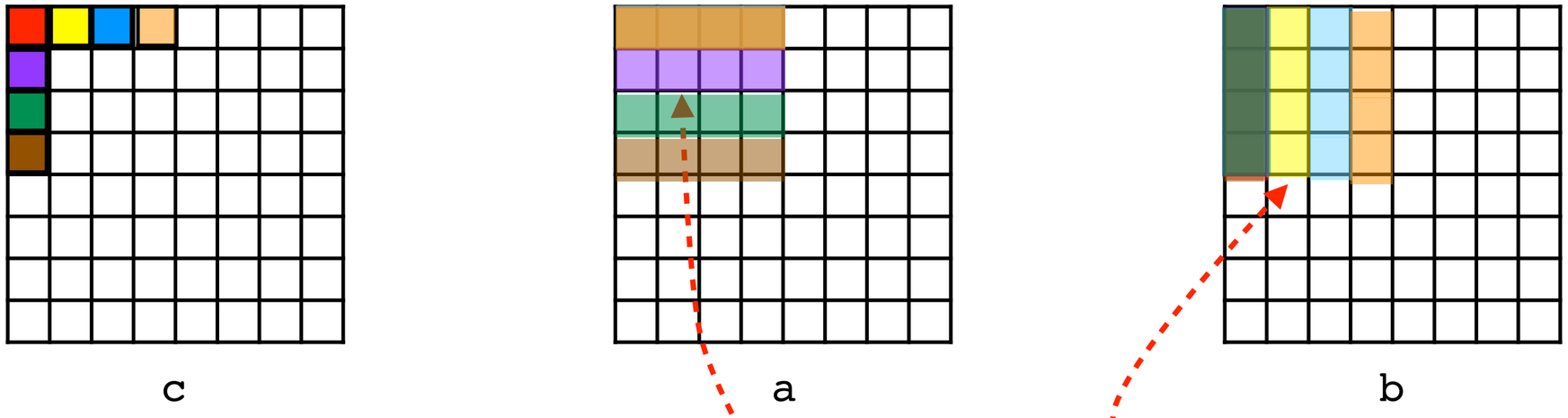
- Discover the cache miss rate
  - valgrind --tool=cachegrind cmd
    - cachegrind is a tool profiling the cache performance
  - Performance counter
    - Intel® Performance Counter Monitor http://www.intel.com/software/pcm/

Block algorithm for matrix multiplication

```
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];
    }
  }
}
```



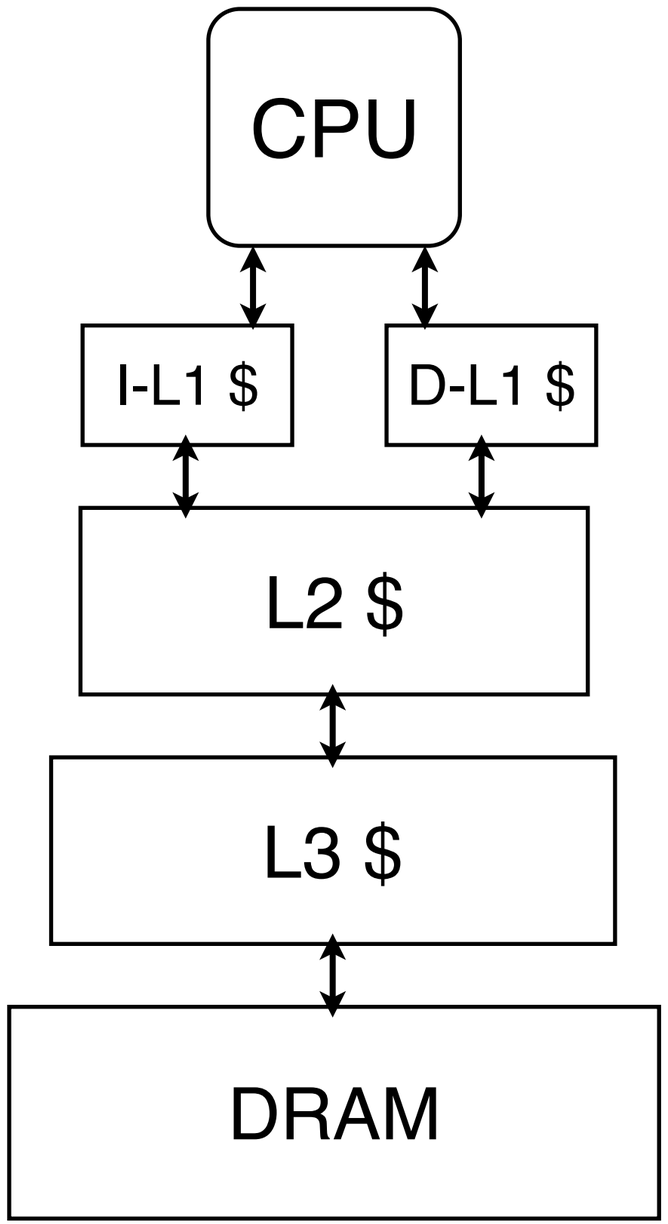
```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
        for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
          for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
            for(kk = k; kk < k+(ARRAY_SIZE/n);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



You only need to hold these sub-matrices in your cache

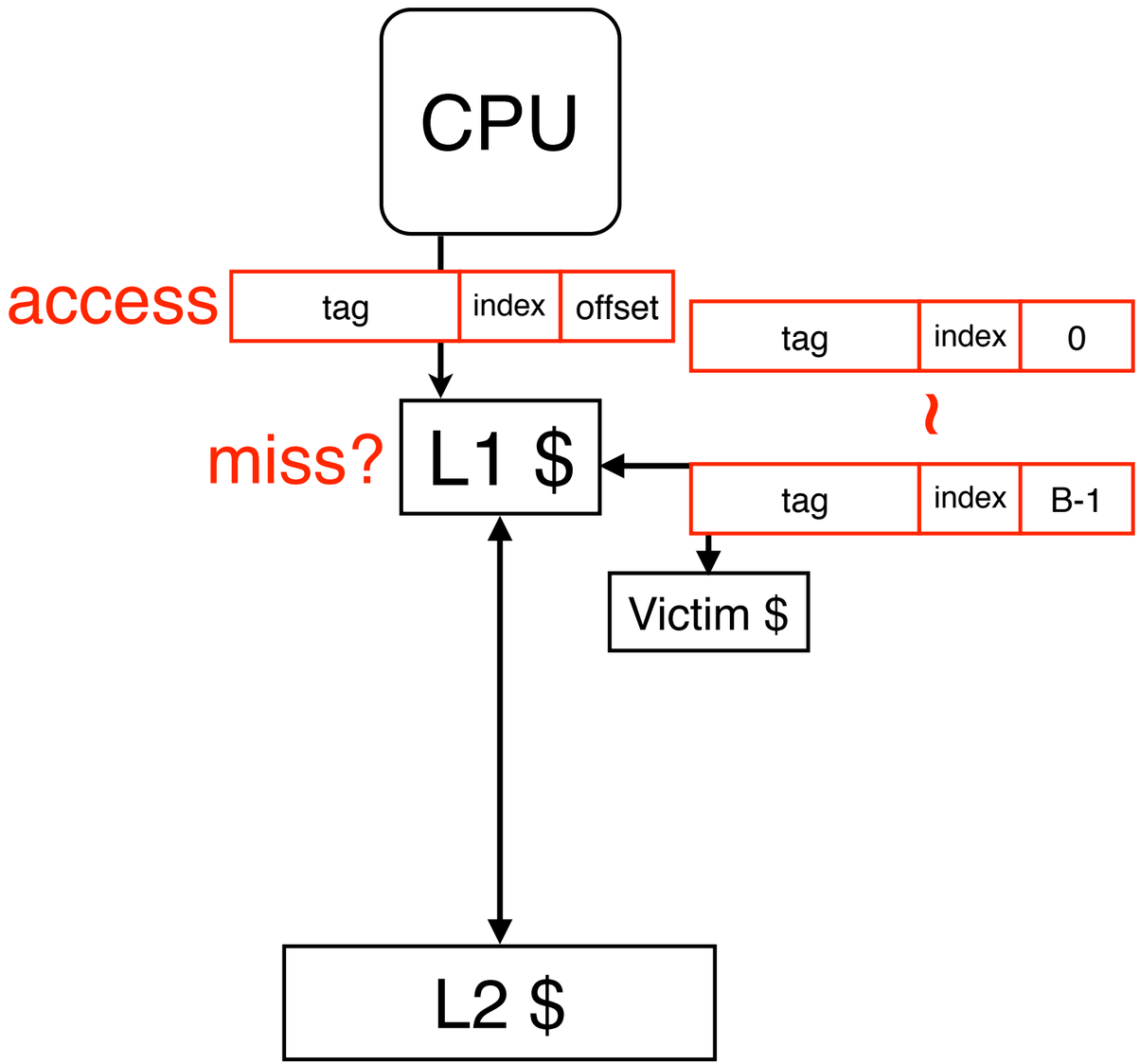
Other cache optimizations

Split 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

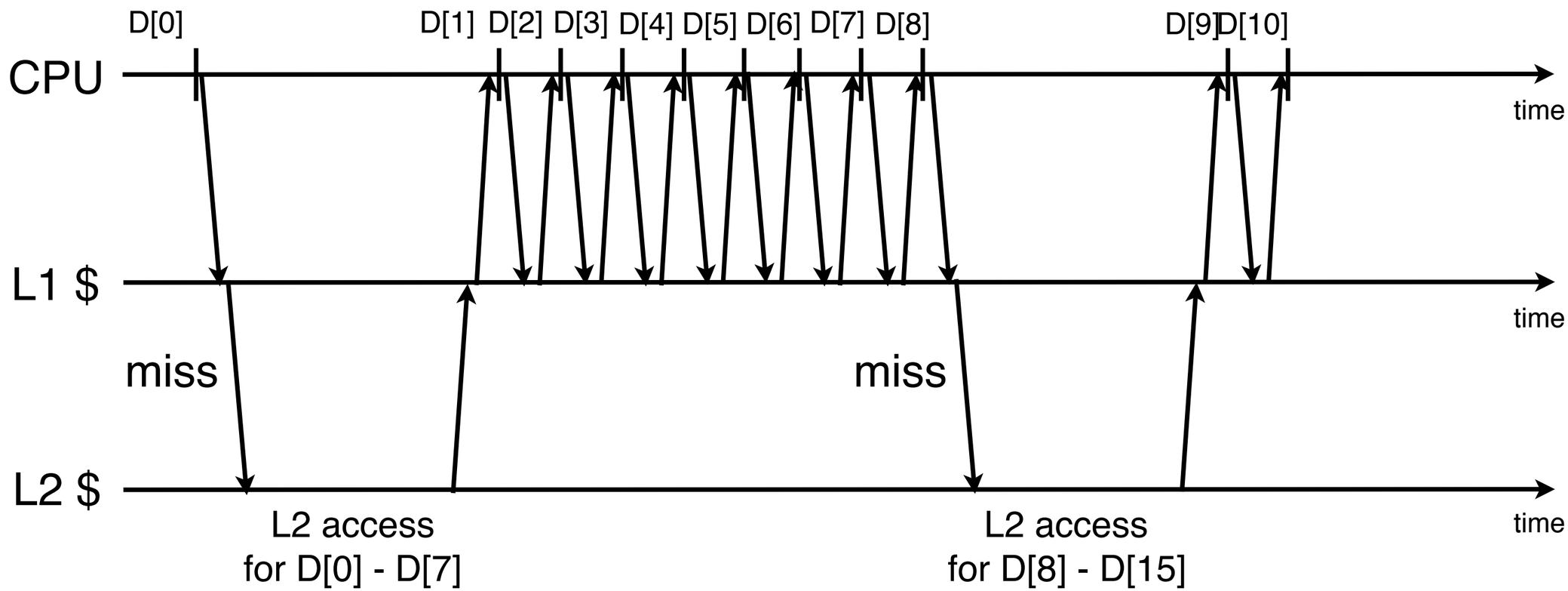
Victim cache



- 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 the **miss penalty** of conflict misses

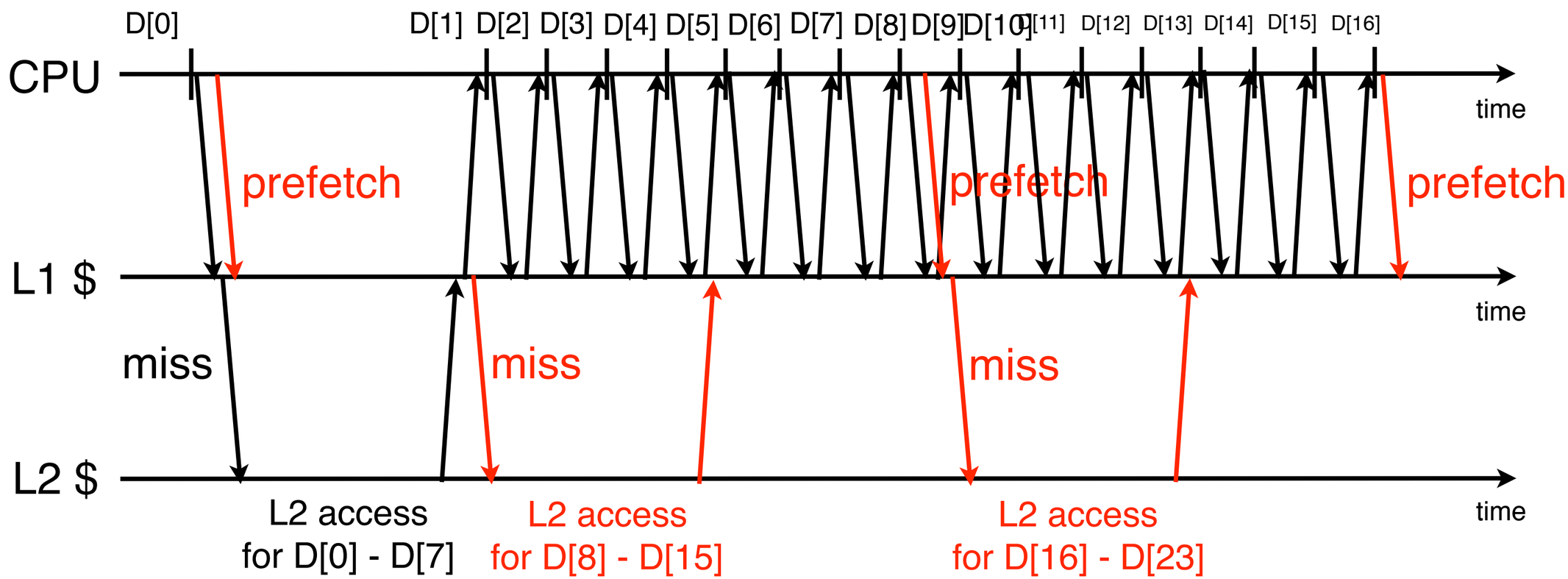
Characteristic of memory accesses

```
for(i = 0;i < 1000000; i++) {
     D[i] = rand();
}
```



Prefetching

```
for(i = 0;i < 1000000; i++) {
      D[i] = rand();
      // prefetch D[i+8] if i % 8 == 0
}
```



Prefetching

- Identify the access pattern and proactively fetch data/instruction before the application asks for the data/instruction
  - Trigger the cache miss earlier to eliminate the miss when the application needs the data/instruction
- 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

Write buffer

- Every write to lower memory will first write to a small SRAM buffer.
  - sw does not incur data hazards, but the pipeline has to stall if the write misses
  - 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
- 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.