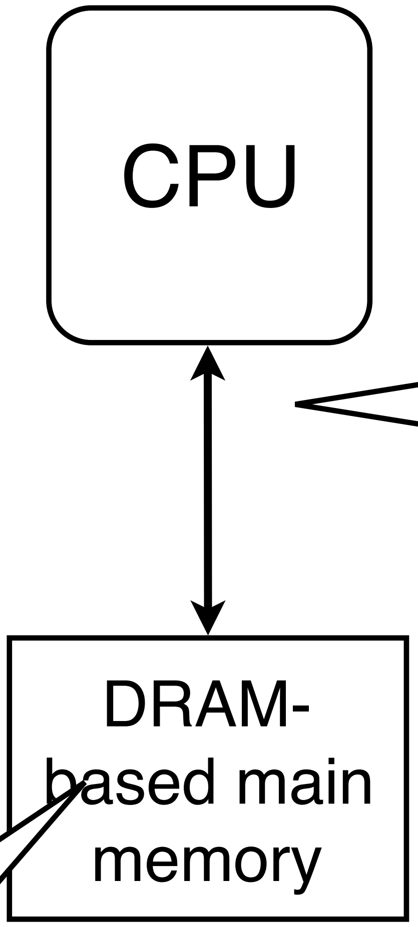
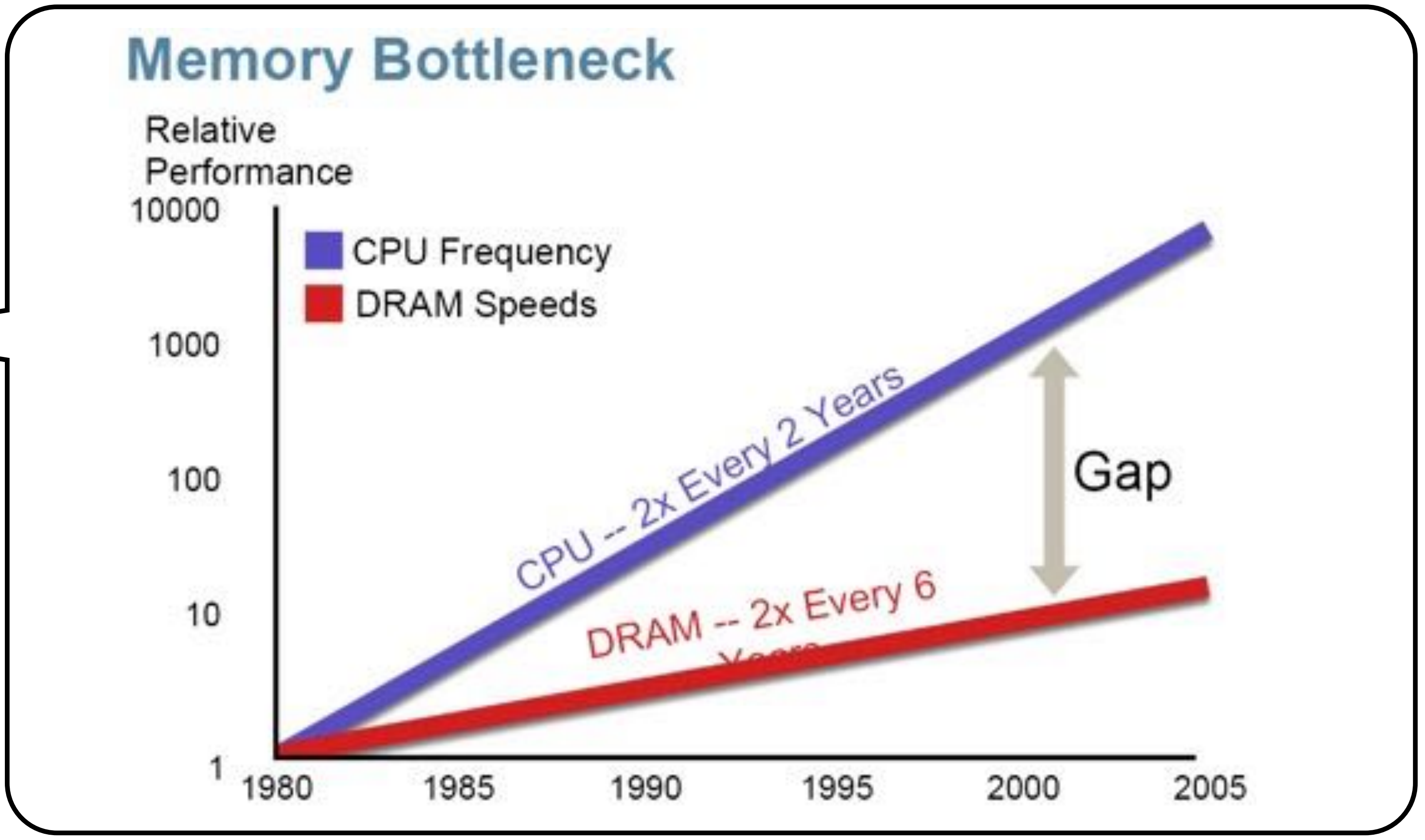
Memory Hierarchy (III)

Hung-Wei Tseng

Recap: 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
```

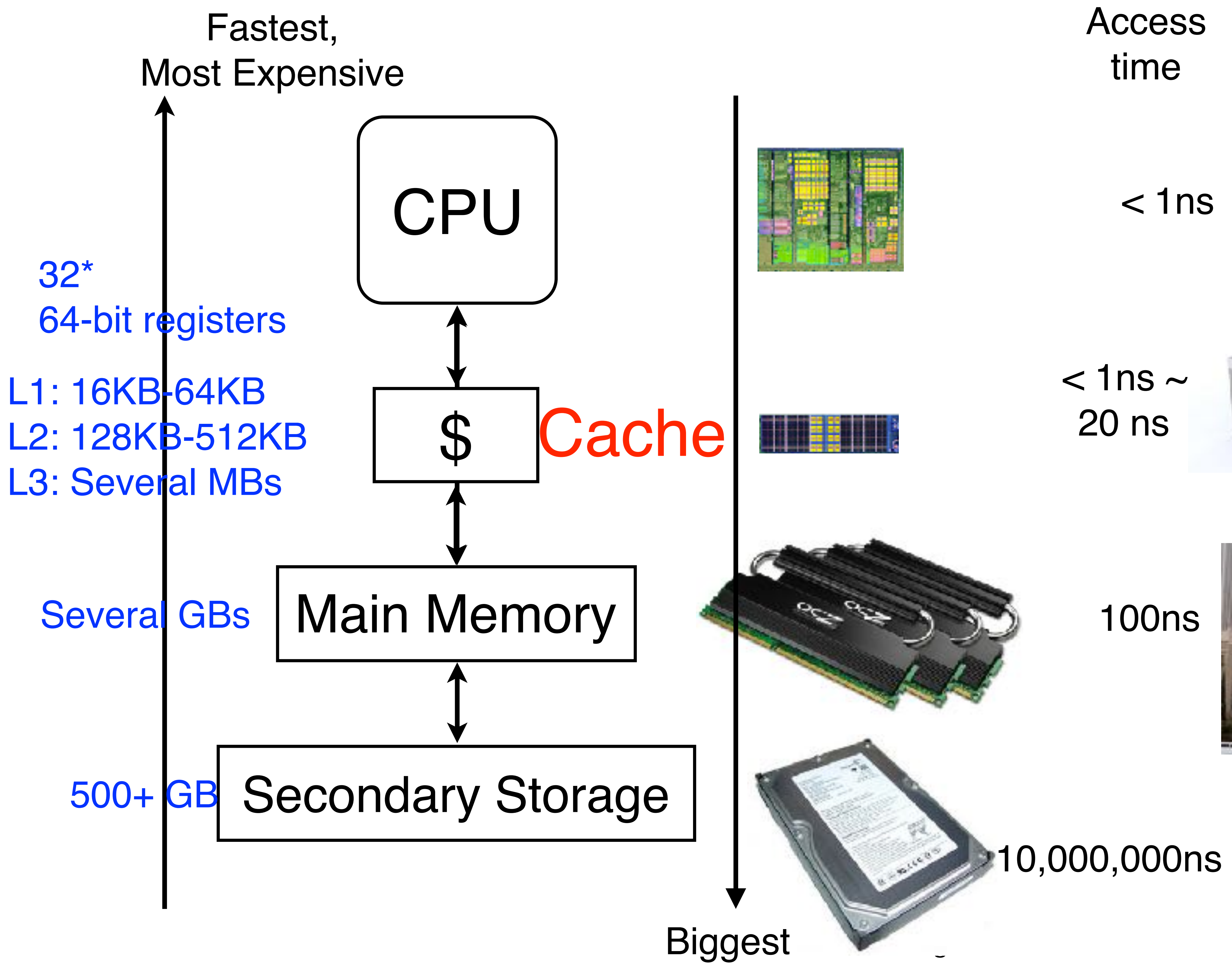
Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

The access time of DRAM is around 50ns

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

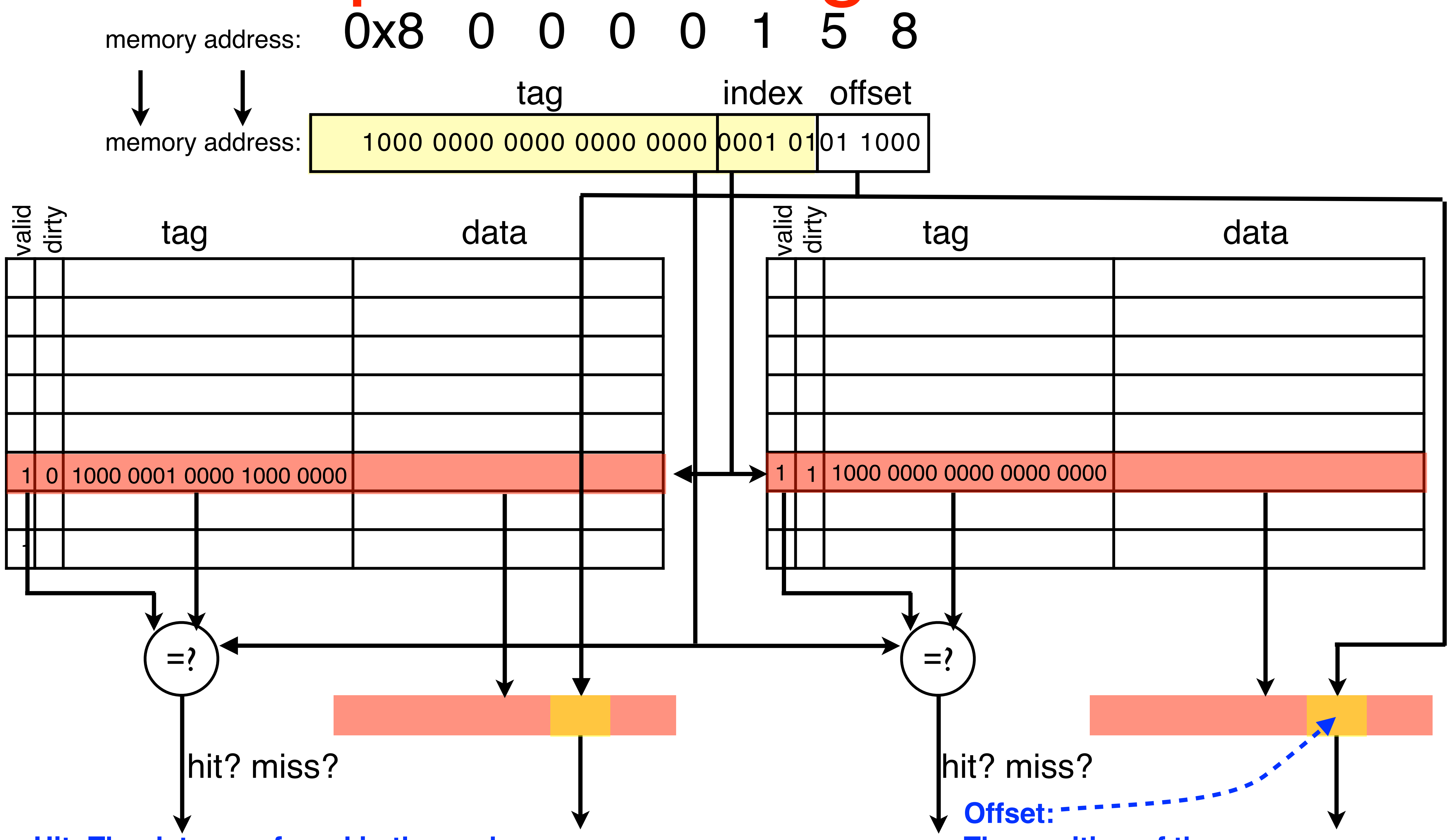
SRAM is as fast as the processor, but $$$

The memory hierarchy



3

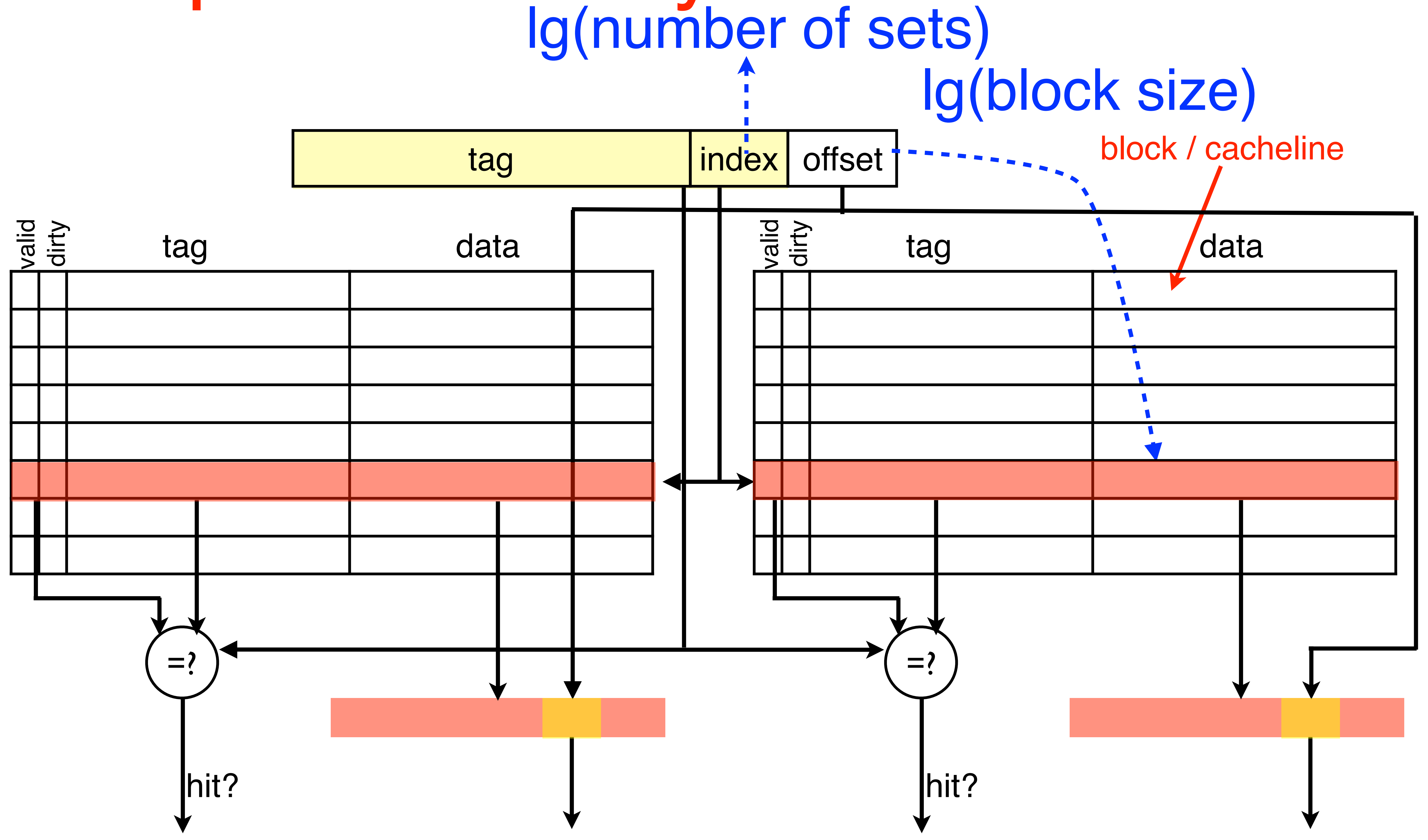
Recap: Accessing the cache



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

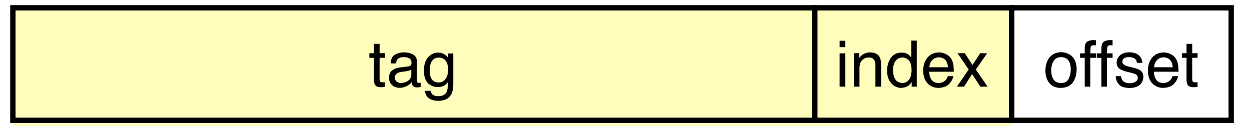
Recap: How many bits in each field?



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

AMD Phenom II

100% miss rate!

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 48-bit address.

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
```

C = ABS
64KB = 2 * 64 * S
S = 512
offset = lg(64) = 6 bits
index = lg(512) = 9 bits
tag = the rest bits

	address in hex	address in binary **tag** **index** **offset**	tag	index	hit? miss?
load a[0]	0x20000	10 0000 0000 0000 0000	0x4	0	miss
load b[0]	0x30000	11 0000 0000 0000 0000	0x6	0	miss
store c[0]	0x10000	1 0000 0000 0000 0000	0x2	0	miss, evict 0x4
load a[1]	0x20004	10 0000 0000 0000 0100	0x4	0	miss, evict 0x6
load b[1]	0x30004	11 0000 0000 0000 0100	0x6	0	miss, evict 0x2
store c[1]	0x10004	1 0000 0000 0000 0100	0x2	0	miss, evict 0x4
⋮	⋮	⋮	⋮	⋮	⋮
load a[15]	0x2003C	10 0000 0000 0011 1100	0x4	0	miss, evict 0x6
load b[15]	0x3003C	11 0000 0000 0011 1100	0x6	0	miss, evict 0x2
store c[15]	0x1003C	1 0000 0000 0011 1100	0x2	0	miss, evict 0x4
load a[16]	0x20040	10 0000 0000 0100 0000	0x4	1	miss
load b[16]	0x30040	11 0000 0000 0100 0000	0x6	1	miss
store c[16]	0x10040	1 0000 0000 0100 0000	0x2	1	miss, evict 0x4

intel Core i7

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
{
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
}
```

	address	tag	index	?
load a[0]	0x20000	0x20	0	miss
load b[0]	0x30000	0x30	0	miss
store c[0]	0x10000	0x10	0	miss
load a[1]	0x20004	0x20	0	hit
load b[1]	0x30004	0x30	0	hit
store c[1]	0x10004	0x10	0	hit
⋮	⋮	⋮	⋮	⋮
load a[15]	0x2003C	0x20	0	hit
load b[15]	0x3003C	0x30	0	hit
store c[15]	0x1003C	0x10	0	hit
load a[16]	0x20040	0x20	1	miss
load b[16]	0x30040	0x30	1	miss
store c[16]	0x1003C	0x10	1	miss

32*3/(512*3) = 1/16 = 6.25% (93.75% hit rate!)

Cache & Performance

- Application: 80% ALU, 20% Loads
- L1 I-cache miss rate: 5%, hit time: 1 cycle
- L1 D-cache miss rate: 10%, hit time: 1 cycle
- L2 U-Cache miss rate: 20%, hit time: 10 cycles
- Main memory hit time: 100 cycles
- What's the average CPI?

1 cycle (no overhead) if hit

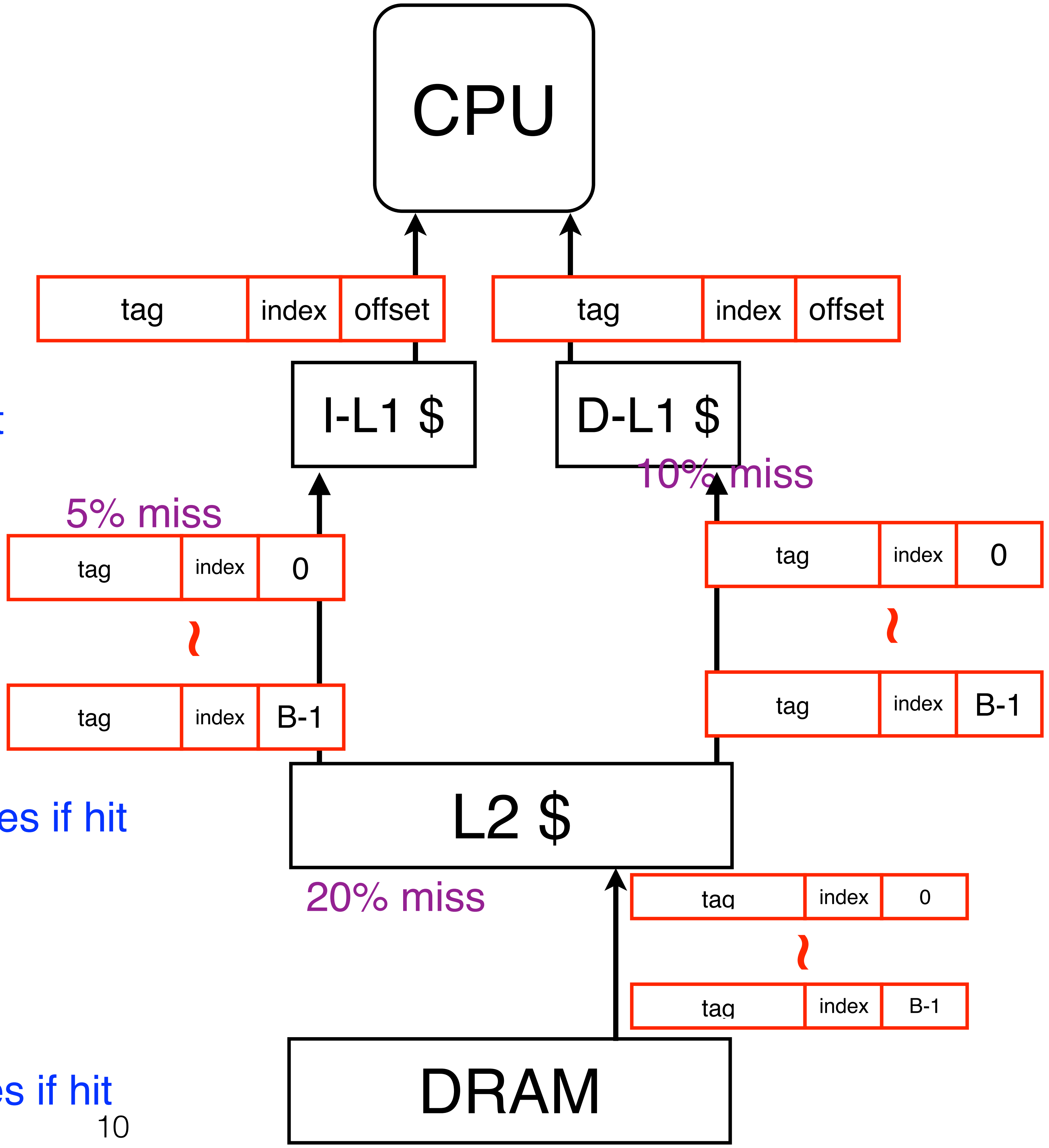
$$CPI_{Average} = CPI_{base} + miss\_rate*miss\_penalty$$

$$= 1+100\%*(5\%*(10+20\%*(1*100)))$$
$$+20\%*(10\%*(1)*(10+20\%*((1)*100)))$$

$$= 3.1$$

10 cycles if hit

100 cycles if hit



Outline

- Performance evaluation with cache
- Cause of misses
- Optimizations

Cache & Performance

- 5-stage MIPS processor.
  - Application: 80% ALU, 20% Loads and stores
  - L1 I-cache miss rate: 5%, hit time: 1 cycle
  - L1 D-cache miss rate: 10%, hit time: 1 cycle, 20% of the replaced blocks are dirty.
  - L2 U-Cache miss rate: 20%, hit time: 10 cycles, 10% of the replaced blocks are dirty.
  - Main memory hit time: 100 cycles
  - What's the average CPI?

A. 0.77

B. 2.6

[✓ C. 3.37]

D. 4.1

E. none of the above

Cache & Performance

- Application: 80% ALU, 20% Load/Store
- L1 I-cache miss rate: 5%, hit time: 1 cycle
- L1 D-cache miss rate: 10%, hit time: 1 cycle, 20% dirty
- L2 U-Cache miss rate: 20%, hit time: 10 cycles, 10% dirty
- Main memory hit time: 100 cycles
- What's the average CPI?

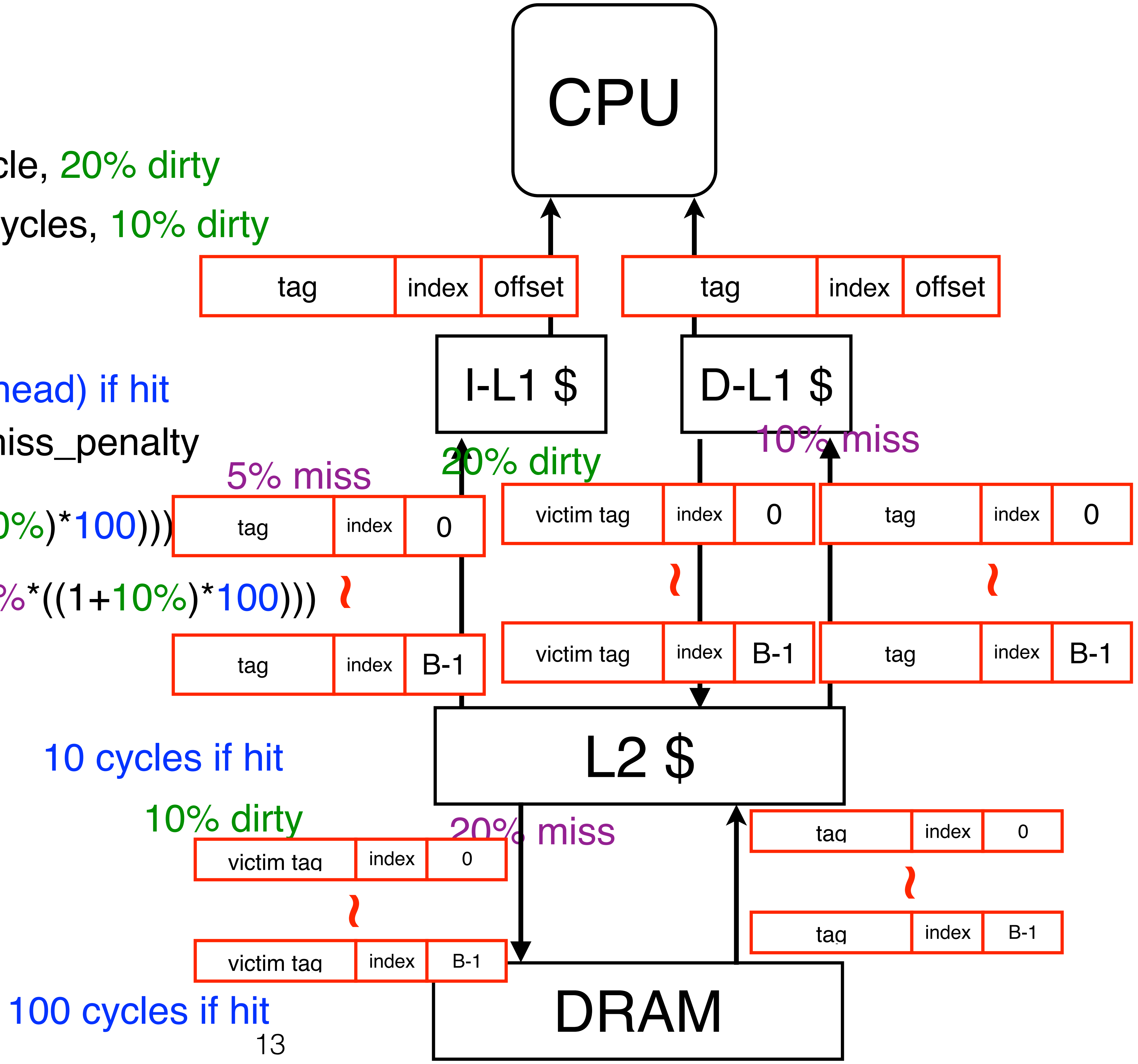
1 cycle (no overhead) if hit

$$CPI_{Average} = CPI_{base} + miss\_rate*miss\_penalty$$

$$= 1+100\%*(5\%*(10+20\%*((1+10\%)*100)))$$

$$+20\%*(10\%*(1+20\%)*(10+20\%*((1+10\%)*100)))$$

$$= 3.368$$



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: $0b\underbrace{1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0}_{\text{tag}}\underbrace{001}_{\text{index}}\ \underbrace{0000}_{\text{offset}}$

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: $0b\underbrace{1000\ 0000\ 0000\ 0000\ 0000\ 0000}_{\text{tag}}\ \underbrace{1000}_{\text{index}}\ \underbrace{0000}_{\text{offset}}$

Simulate a direct-mapped cache

	valid	tag	data
0	1	0b10	
1	1	0b10 0b11	
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

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

AMD Phenom II

- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.
  - Consider the following code

    -
      ```
      int a[16384], b[16384], c[16384];
      /* c = 0x10000, a = 0x20000, b = 0x30000 */
      for(i = 0; i < 512; i++) {
          c[i] = a[i] + b[i];
          //load a, b, and then store to c
      }
      ```
    - How many of the cache misses are "conflict misses"?

A. 6.25%

B. 66.67%

C. 68.75%

[✓ D. 93.75%]

E. 100%

AMD Phenom II

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
{
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
}
```

	address	tag	index	hit? miss?
load a[0]	0x20000	0x4	0	compulsory miss
load b[0]	0x30000	0x6	0	compulsory miss
store c[0]	0x10000	0x2	0	compulsory miss, evict 0x4
load a[1]	0x20004	0x4	0	conflict miss, evict 0x6
load b[1]	0x30004	0x6	0	conflict miss, evict 0x2
store c[1]	0x10004	0x2	0	conflict miss, evict 0x4
⋮	⋮	⋮	⋮	⋮
load a[15]	0x2003C	0x4	0	conflict miss, evict 0x6
load b[15]	0x3003C	0x6	0	conflict miss, evict 0x2
store c[15]	0x1003C	0x2	0	conflict miss, evict 0x4

intel Core i7

- D-L1 Cache configuration of Core i7
  - Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, 32-bit OS?
  - Consider the following code?

    -
      ```
      int a[16384], b[16384], c[16384];
      /* c = 0x10000, a = 0x20000, b = 0x30000 */
      for(i = 0; i < 512; i++) {
          c[i] = a[i] + b[i];
          //load a, b, and then store to c
      }
      ```
    - How many of the cache misses are “compulsory misses”?

A. 6.25%

B. 33.33%

C. 66.67%

D. 68.75%

[✓ E. 100%]

intel Core i7

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
{
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
}
```

	address	tag	index	?
load a[0]	0x20000	0x20	0	compulsory miss
load b[0]	0x30000	0x30	0	compulsory miss
store c[0]	0x10000	0x10	0	compulsory miss
load a[1]	0x20004	0x20	0	hit
load b[1]	0x30004	0x30	0	hit
store c[1]	0x10004	0x10	0	hit
⋮	⋮	⋮	⋮	⋮
load a[15]	0x2003C	0x20	0	hit
load b[15]	0x3003C	0x30	0	hit
store c[15]	0x1003C	0x10	0	hit

512/(64/4) = 32 compulsory misses each array

32*3/(512*3) = 1/16 = 6.25% (93.75% hit rate!)

Improving 3Cs

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?
  ① Increasing associativity can reduce conflict misses
  ② Increasing associativity can reduce hit time
  ③ Increasing block size can increase the miss penalty
  ④ Increasing block size can reduce compulsory misses

  A. 0
  B. 1
  C. 2
  [✓ D. 3]
  E. 4

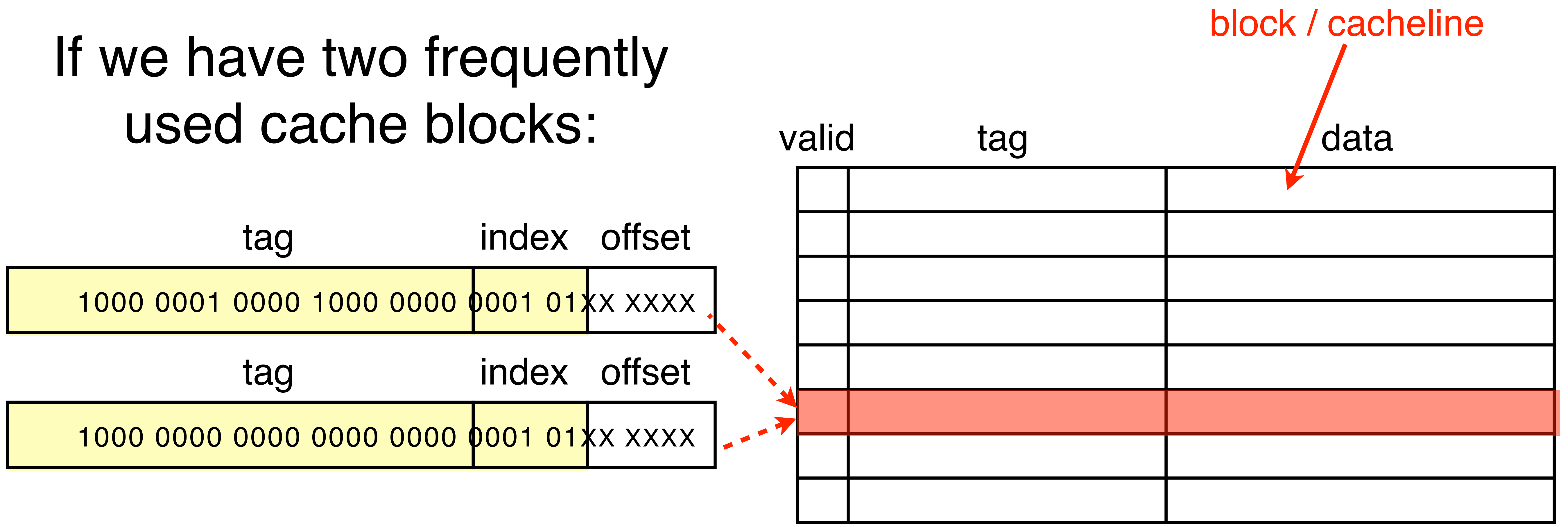
Increases hit time because your data array is larger (longer time to fully charge your bit-lines)

You need to fetch more data for each miss

You bring more into the cache when a miss occurs

Conflict in direct-mapped cache

If we have two frequently used cache blocks:


tag
index
offset
1000 0001 0000 1000 0000 0001 01xx xxxx
tag
index
offset
1000 0000 0000 0000 0000 0001 01xx xxxx
block / cacheline
valid
tag
data

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

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

```
#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
```

faster

Side-by-side comparison

i on row, j on col	i on col, j on row
`for(i = 0; i < ARRAY_SIZE; i++) {`	`for(j = 0; j < ARRAY_SIZE; j++)  {`
`  for(j = 0; j < ARRAY_SIZE; j++)  {`	`  for(i = 0; i < ARRAY_SIZE; i++)  {`
`    c[i][j] = a[i][j]+b[i][j];`	`    c[i][j] = a[i][j]+b[i][j];`
`  }`	`  }`
`}`	`}`

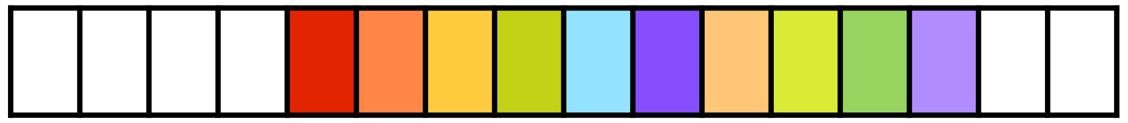
- What type(s) of cache locality does(do) the left-hand side code better exploit than the right-hand side code?
  [✓ A. Spatial locality]
  B. Temporal locality
  C. Both localities
  D. None of them

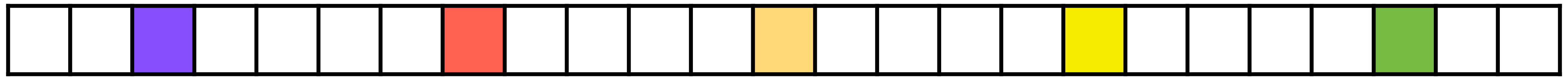
Demo revisited

- Why the left performs a lot better than t

`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`





What data structure is performing better

Array of objects	object of arrays

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

- Considering your workload would like to calculate the average score of each homework, which data structure would deliver better performance?
  A. Array of objects
  B. Object of arrays

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;
		}

What data structure is performing better

Array of objects	object of arrays
`struct grades`	`struct grades`
`{`	`{`
`  int id;`	`  int *id;`
`  double *homework;`	`  double **homework;`
`  double average;`	`  double *average;`
`};`	`};`

- Considering your workload would like to calculate the average score of each homework, which data structure would deliver better performance?
  A. Array of objects
  [✓ B. Object of arrays]

What if we want to calculate average scores for 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 $O(n^3)$

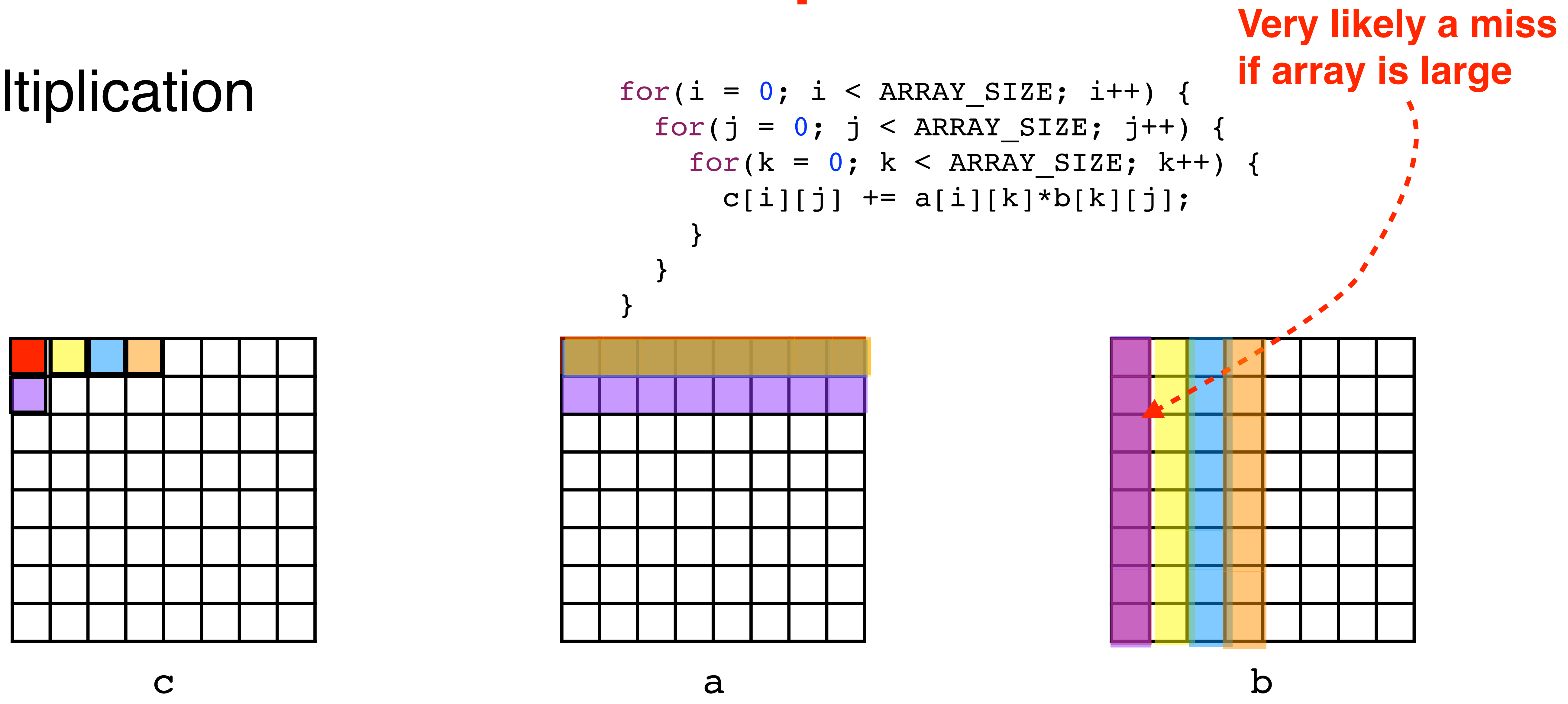
If n=512, it takes about 1 sec

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];
    }
  }
}
```



- 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!

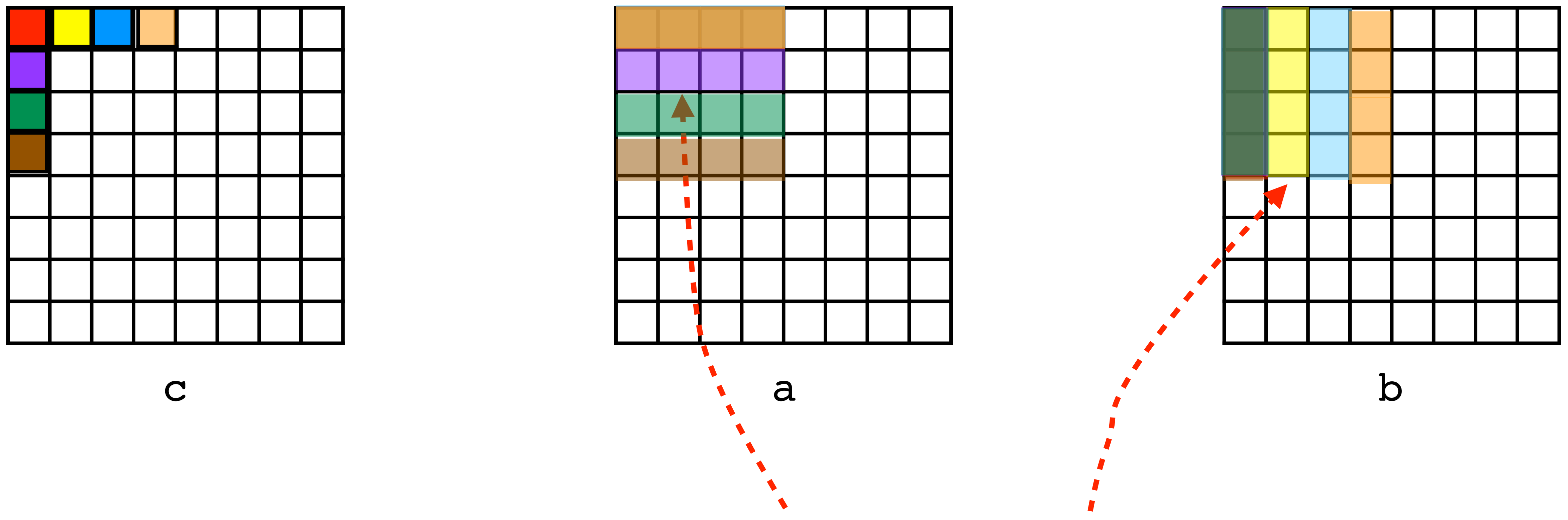
Block algorithm for matrix multiplication

- 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

Block algorithm for matrix multiplication

- Connecting architecture and software design now!
  - Block Algorithm for Matrix Multiplication
  - What value of n makes the block algorithm works the best?
  - If the demo machine has an L1 D-cache with 64KB, 2-way, 64B blocks, array_size is 1024, each word is "8 bytes"

A. 16

B. 32

C. 64

D. 128

E. 256

```
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];
    }
  }
}
```

Block algorithm for matrix multiplication

- Connecting architecture and software design now!
  - Block Algorithm for Matrix Multiplication
  - What value of n makes the block algorithm works the best?
  - If the demo machine has an L1 D-cache with 64KB, 2-way, 64B blocks, array_size is 1024, each word is “8 bytes”

A. 16

B. 32

C. 64

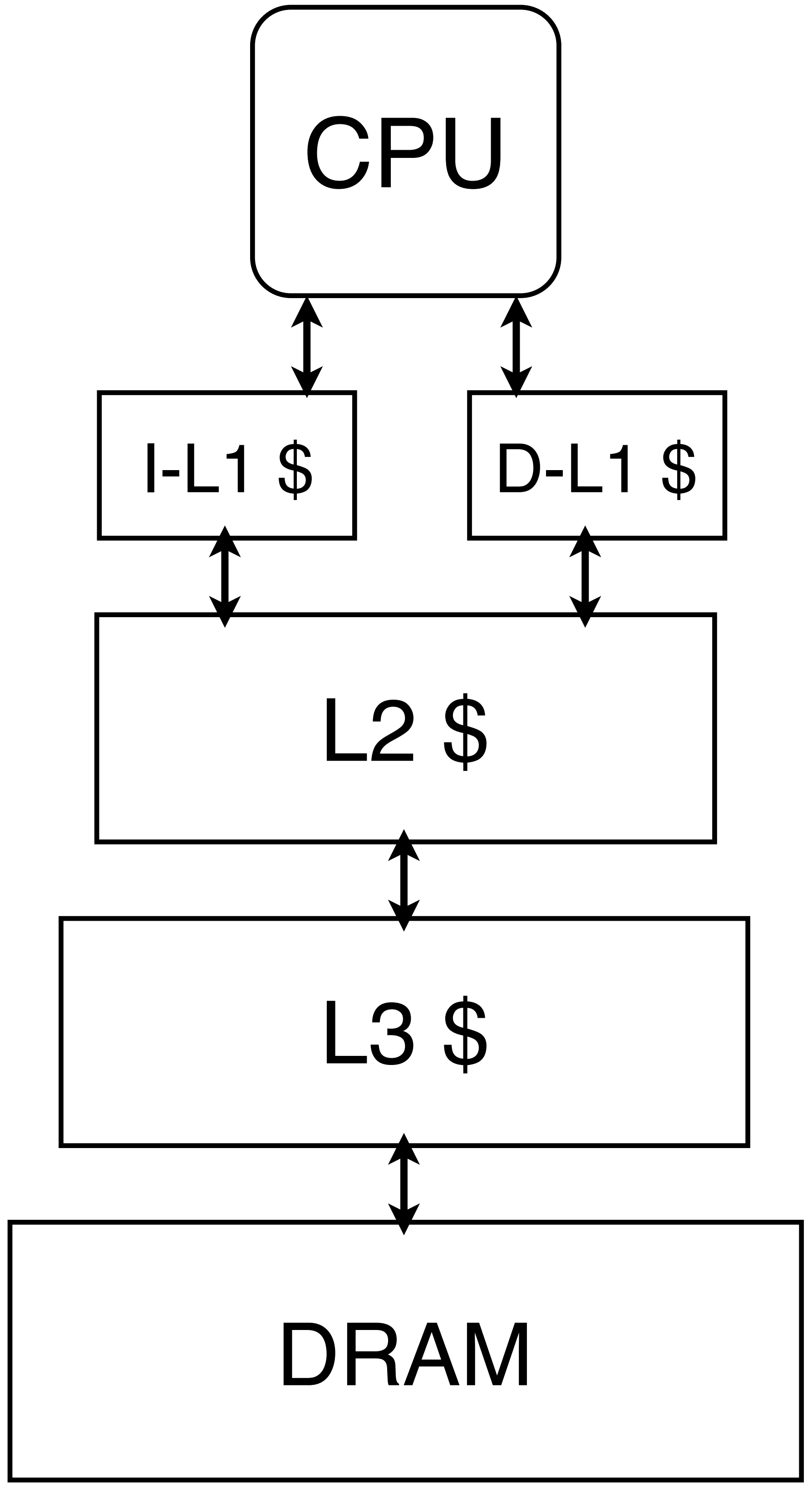
[✓ D. 128]

E. 256

```
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

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

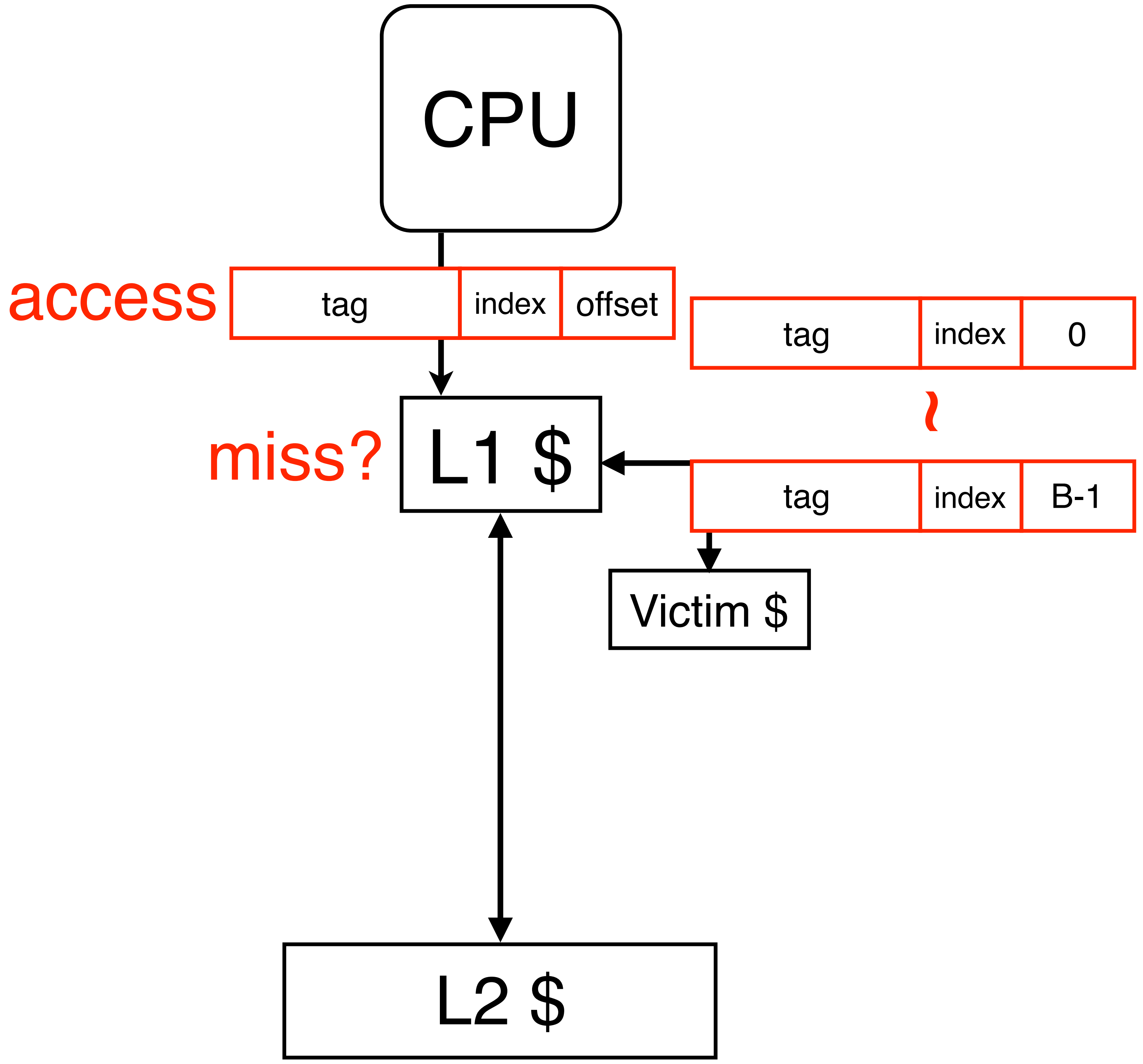
Revisit: Athlon 64

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
{
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
}
```

	address	tag	index	?
load a[0]	0x20000	0x4	0	compulsory miss
load b[0]	0x30000	0x6	0	compulsory miss
store c[0]	0x10000	0x2	0	compulsory miss, evict 0x4
load a[1]	0x20004	0x4	0	conflict miss, evict 0x6
load b[1]	0x30004	0x6	0	conflict miss, evict 0x2
store c[1]	0x10004	0x2	0	conflict miss, evict 0x4

100% miss rate due to a majority of conflict miss!

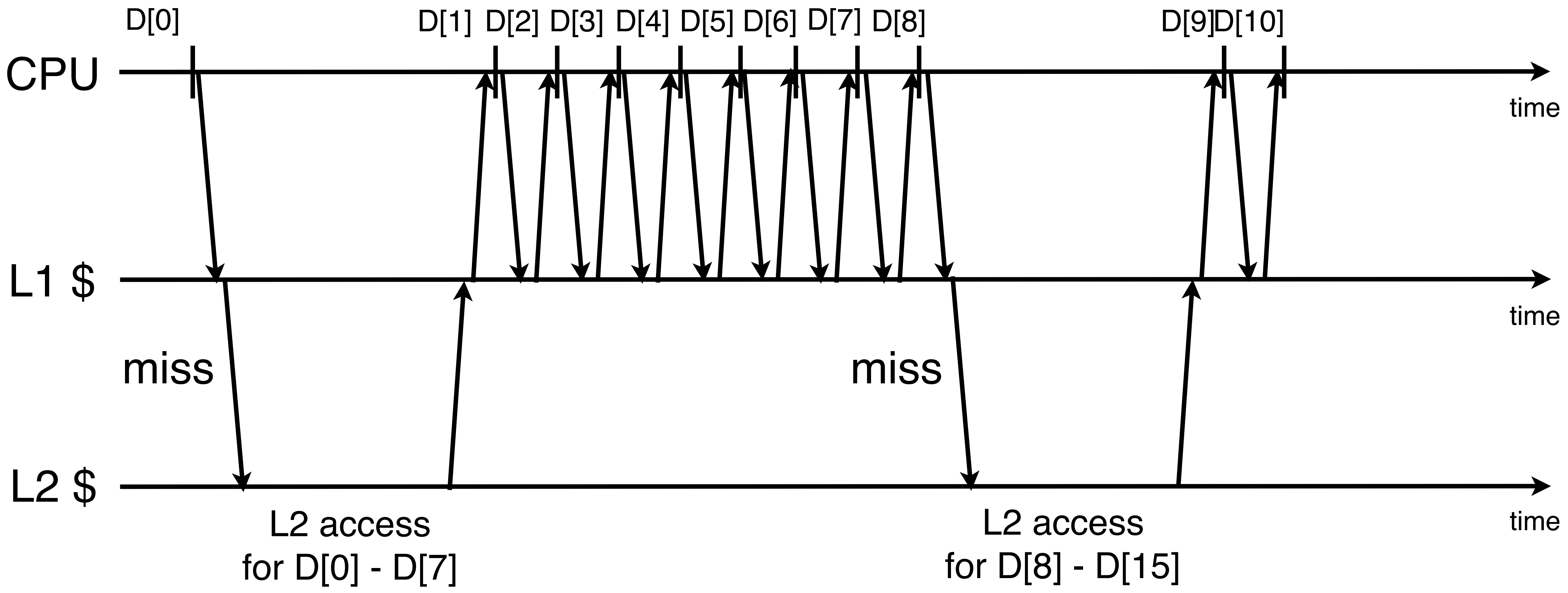
Victim cache



- 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

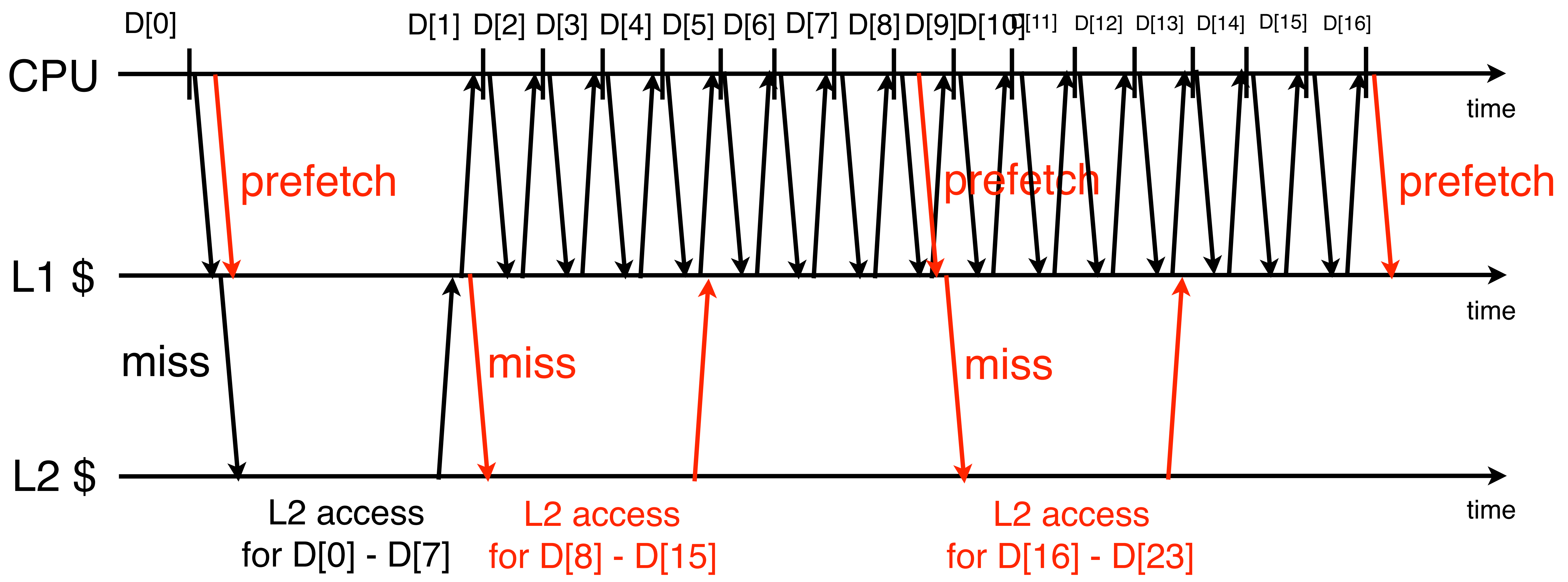
Characteristic of memory accesses

```
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