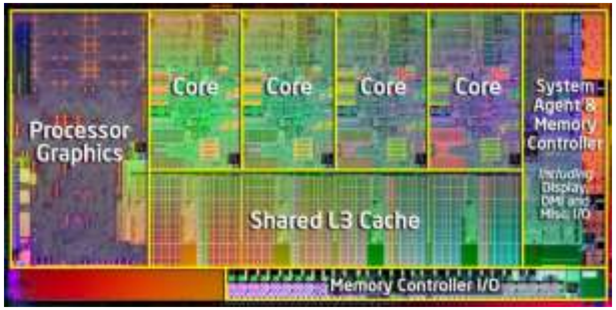
고려대학교
KOREA UNIVERSITY



COSE222 Computer Architecture

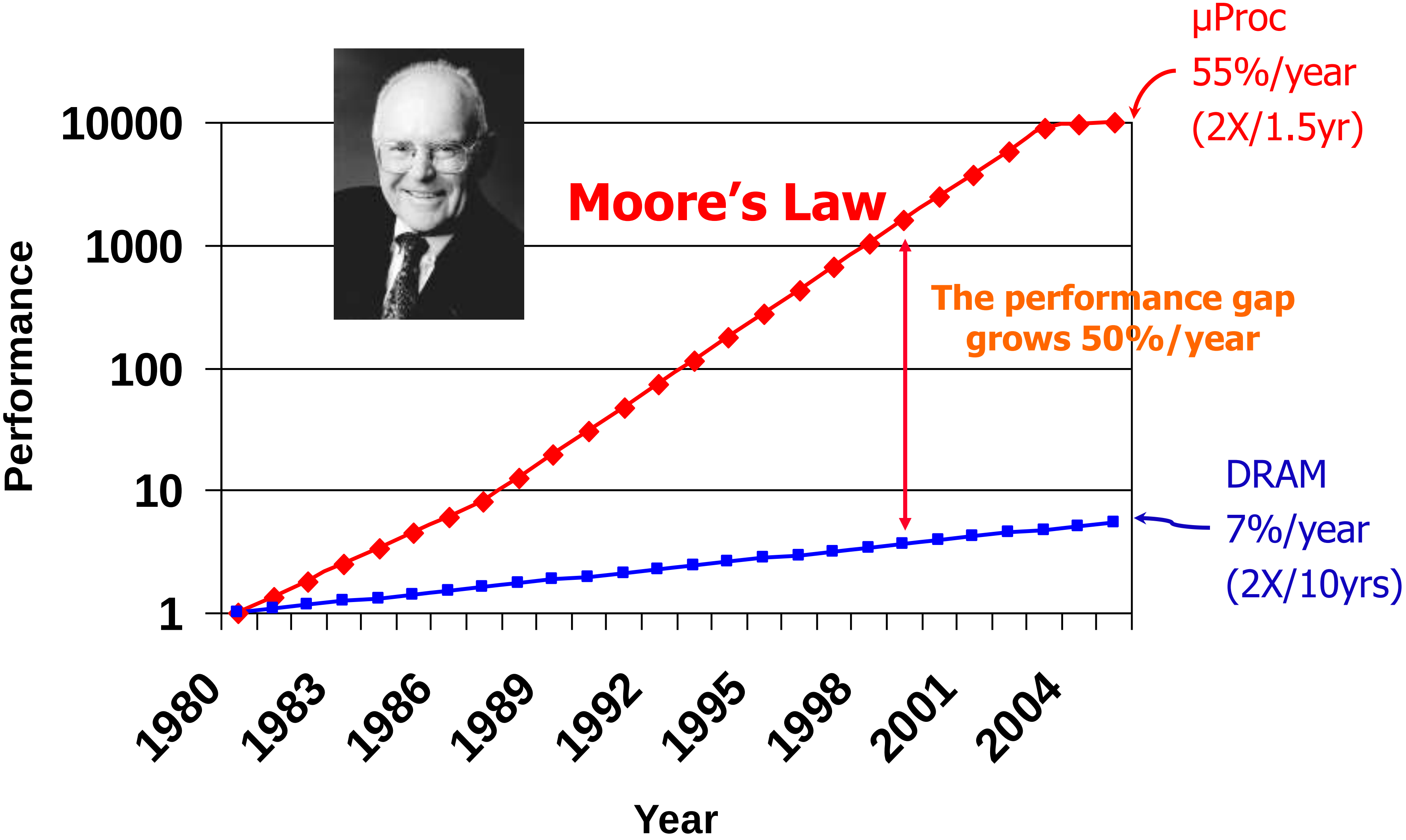
Lecture 6. Cache #1

Prof. Taeweon Suh

Computer Science & Engineering

Korea University

CPU vs Memory Performance



Moore's Law

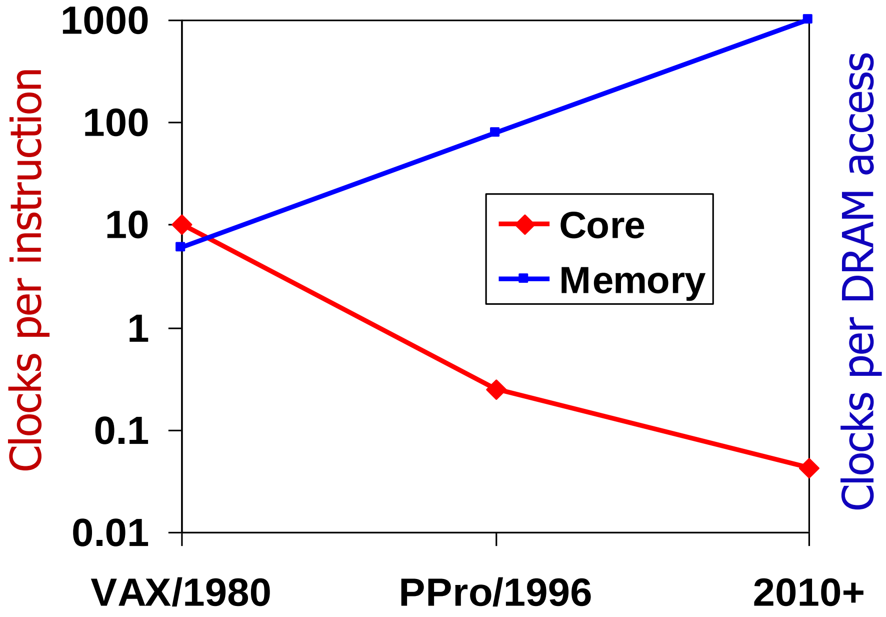
µProc
55%/year
(2X/1.5yr)

The performance gap grows 50%/year

DRAM
7%/year
(2X/10yrs)

Memory Wall

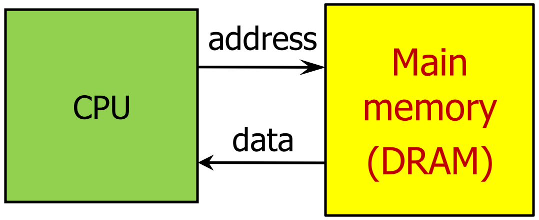
- CPU vs DRAM speed disparity continues to grow
  - The processor performance is limited by memory (memory wall)
  - Good memory hierarchy design is important to the overall performance of a computer system

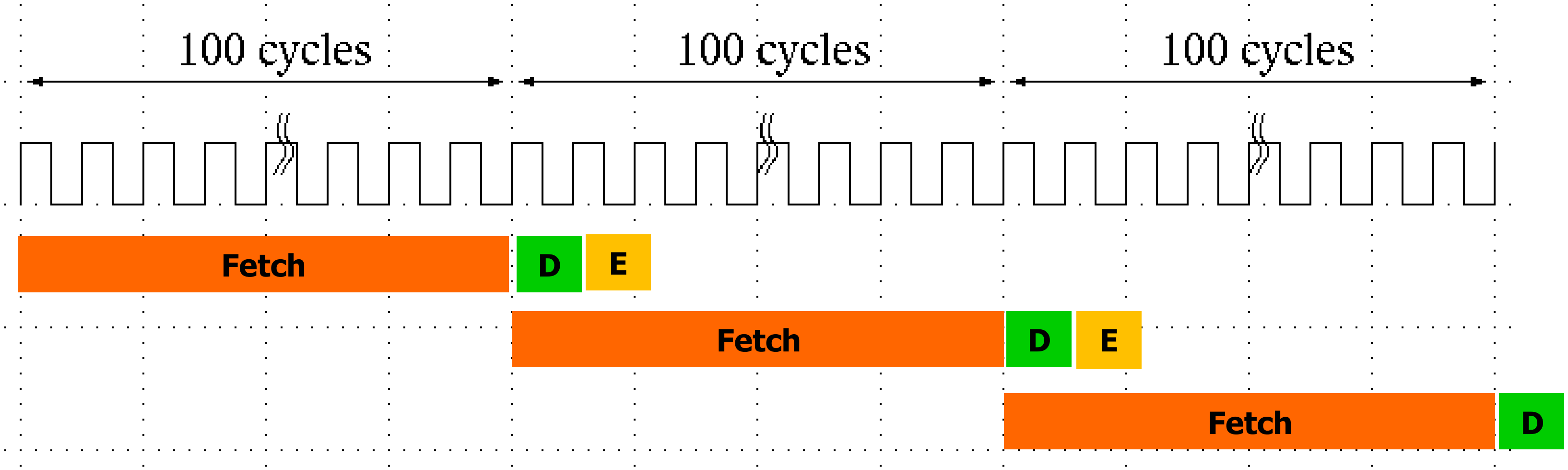


Memory Technology	Typical access time	$ per GB in 2008
SRAM	0.5 ~ 2.5 ns	$2000 ~ $5000
DRAM	50 ~ 70 ns	$20 ~ $70
Magnetic disk	5,000,000 ~ 20,000,000 ns	$0.20 ~ $2

Performance

- Consider the basic 5 stage pipeline design
  - CPU runs at 1GHz (1ns cycle time)
  - Main memory takes 100ns to access

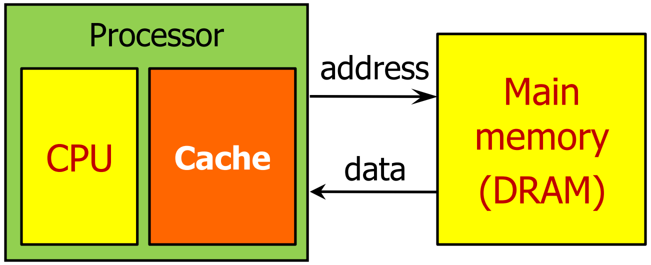


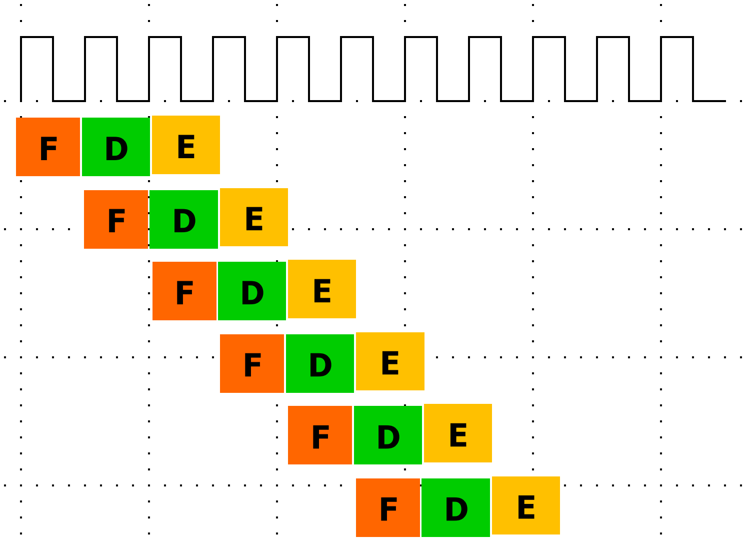


- The CPU effectively executes 1 instruction per 100 clock cycles
- The problem is getting worse as CPU speed grows much faster than DRAM

Typical Solution

- Cache (memory) is used to reduce the large speed gap between CPU and DRAM
  - Cache is ultra-fast and small memory inside processor
  - Cache is an SRAM-based memory
  - Frequently used instructions and data in main memory are placed in cache by hardware
  - Cache (L1) is typically accessed at 1 CPU cycle

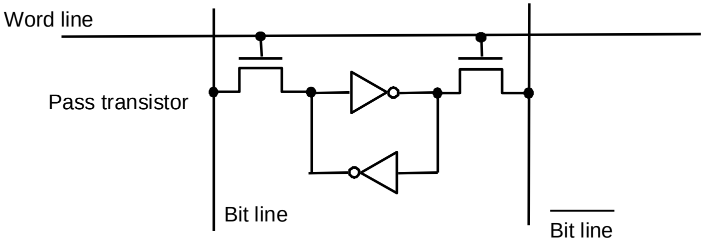




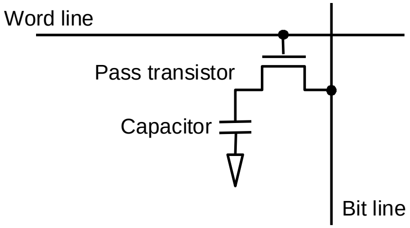
- Theoretically, the CPU is able to execute 1 instruction per 1 clock cycle

SRAM vs DRAM

- **SRAM**
  - A bit is stored on a pair of inverting gates
  - Very fast, but takes up more space (4 to 6 transistors) than DRAM
  - **Used to design cache**



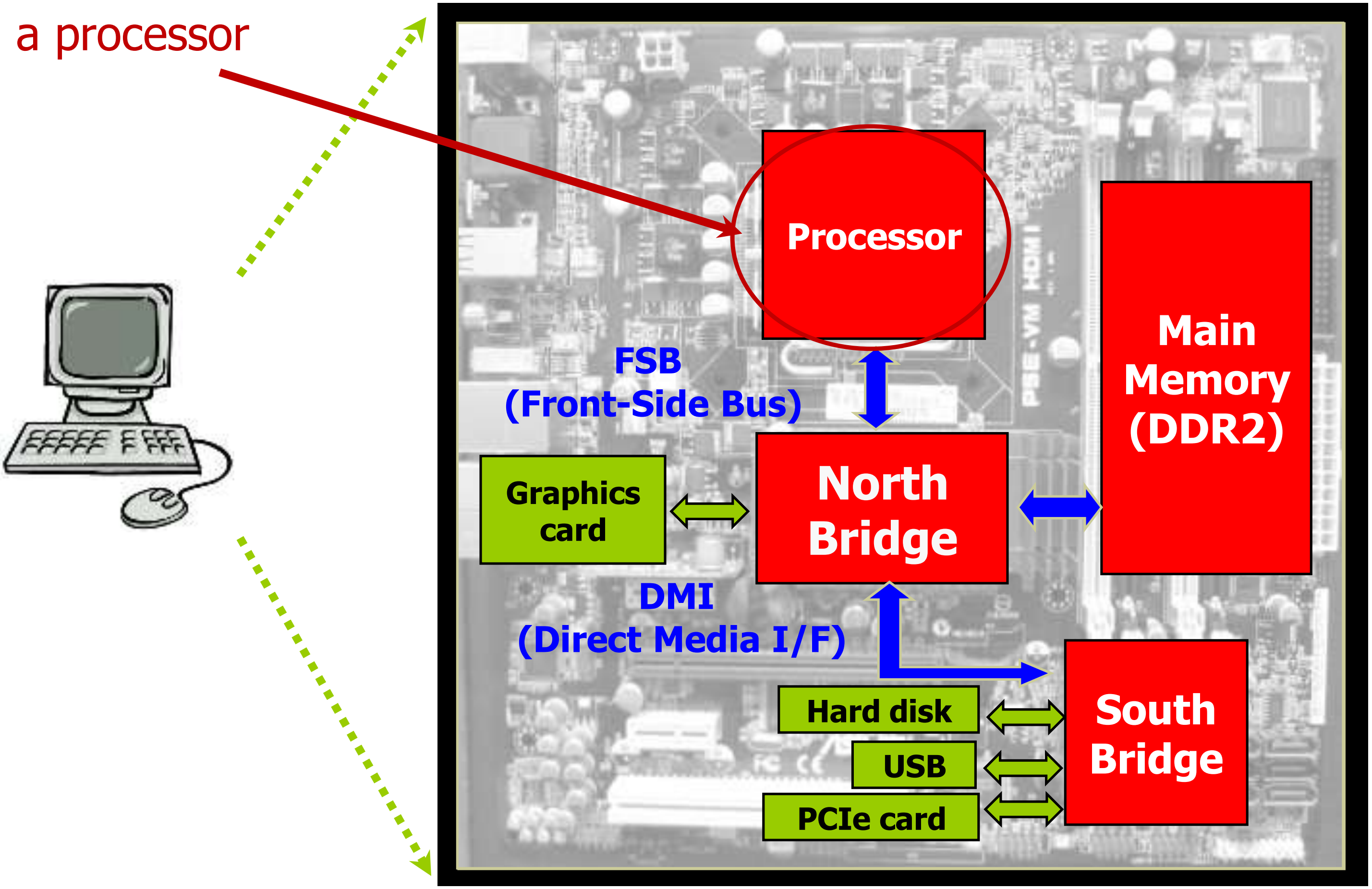
- **DRAM**
  - A bit is stored as a charge on capacitor (must be refreshed)
  - Very small, but slower than SRAM (factor of 5 to 10)
  - **Used for main memory** such as DDR SDRAM



Memory Technology	Typical access time	$ per GB in 2008
SRAM	0.5 ~ 2.5 ns	$2000 ~ $5000
DRAM	50 ~ 70 ns	$20 ~ $70
Magnetic disk	5,000,000 ~ 20,000,000 ns	$0.20 ~ $2

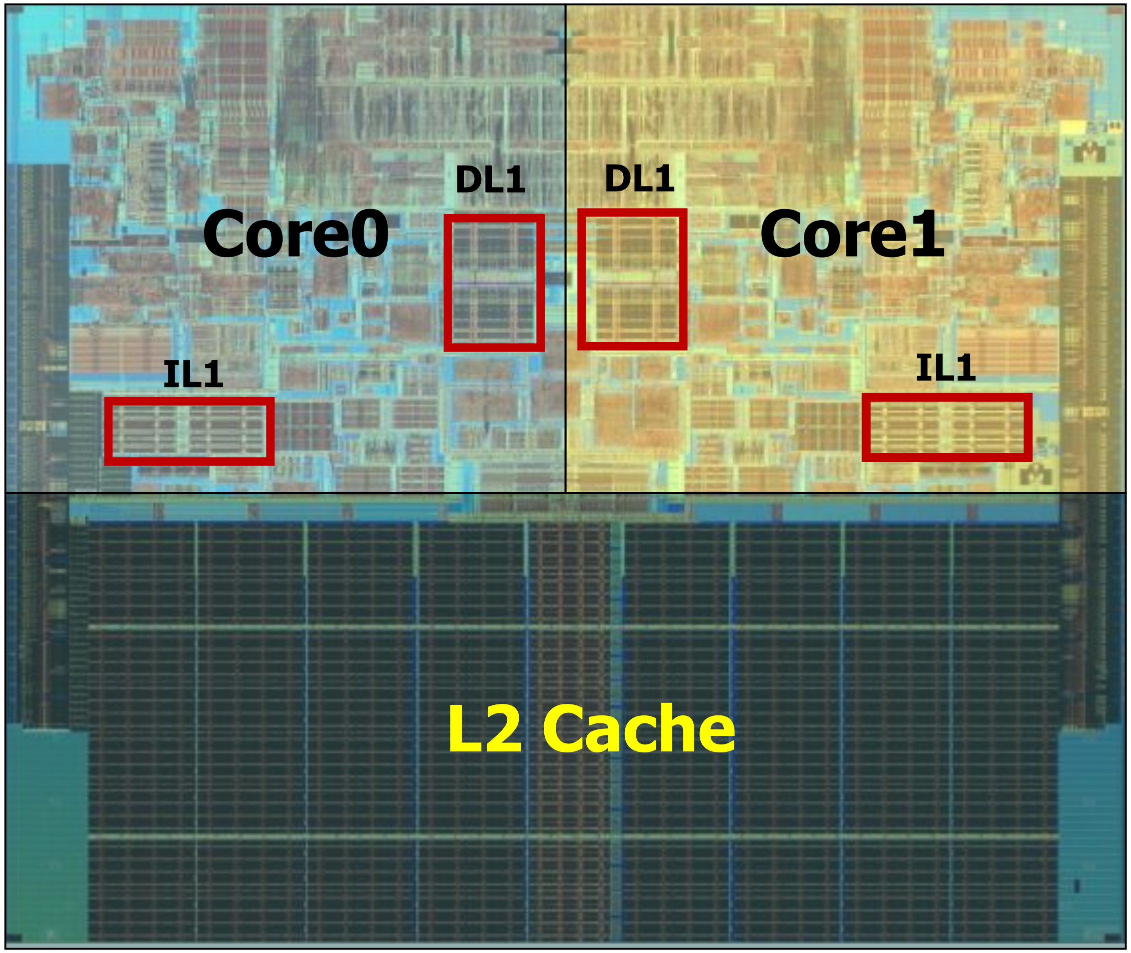
A Computer System

Caches are located
inside a processor



Core 2 Duo (Intel)

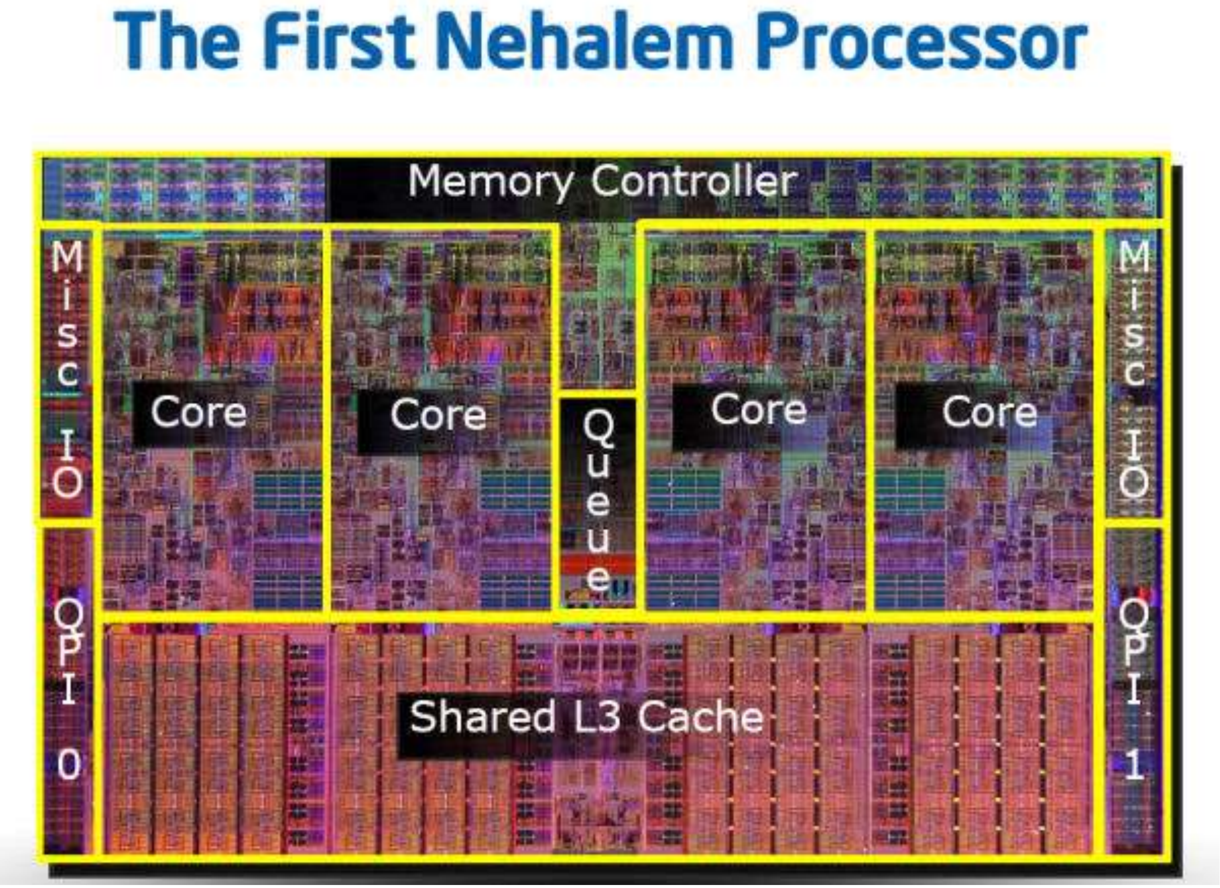
L1	32 KB, 8-Way, 64 Byte/Line, LRU, WB 3 Cycle Latency
L2	4.0 MB, 16-Way, 64 Byte/Line, LRU, WB 14 Cycle Latency



Source: http://www.sandpile.org

Core i7 (Intel)

- 4 cores on one chip
- Three levels of caches (L1, L2, L3) on chip
  - **L1**: **32KB**, 8-way
  - **L2**: **256KB**, 8-way
  - **L3**: **8MB**, 16-way
- 731 million transistors in 263 mm^2 with 45nm technology

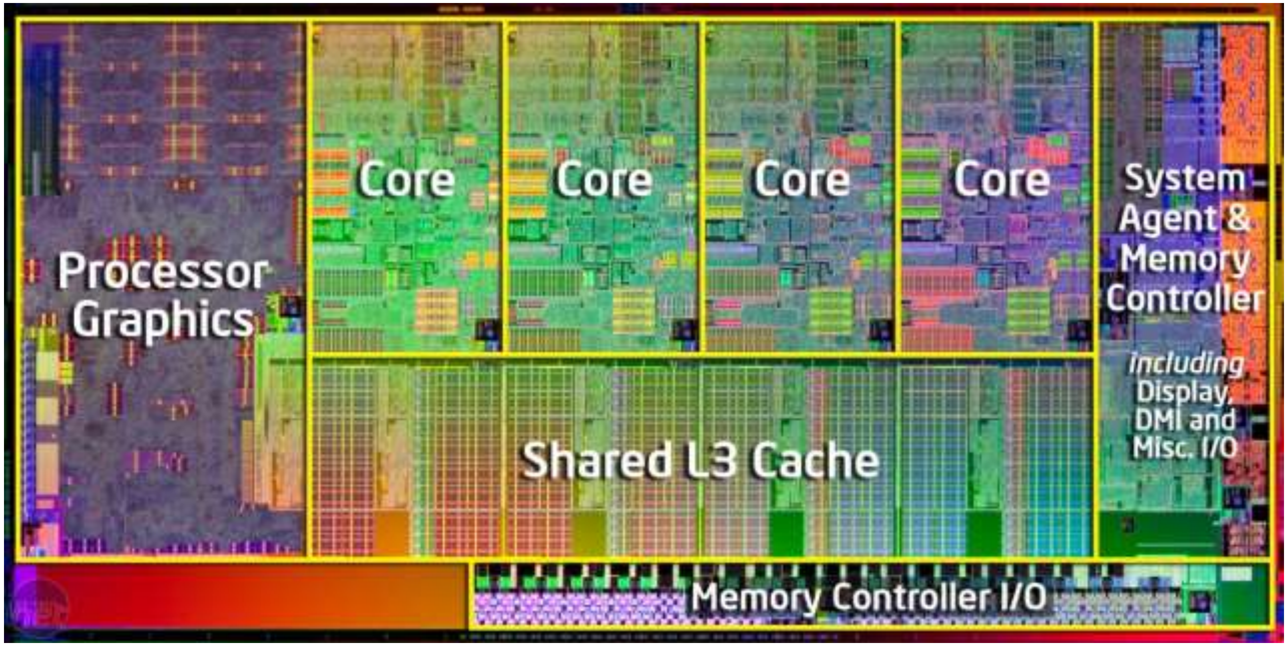


Core i7 (2nd Gen.)

2nd Generation Core i7

Sandy Bridge

L1	32 KB
L2	256 KB
L3	8MB



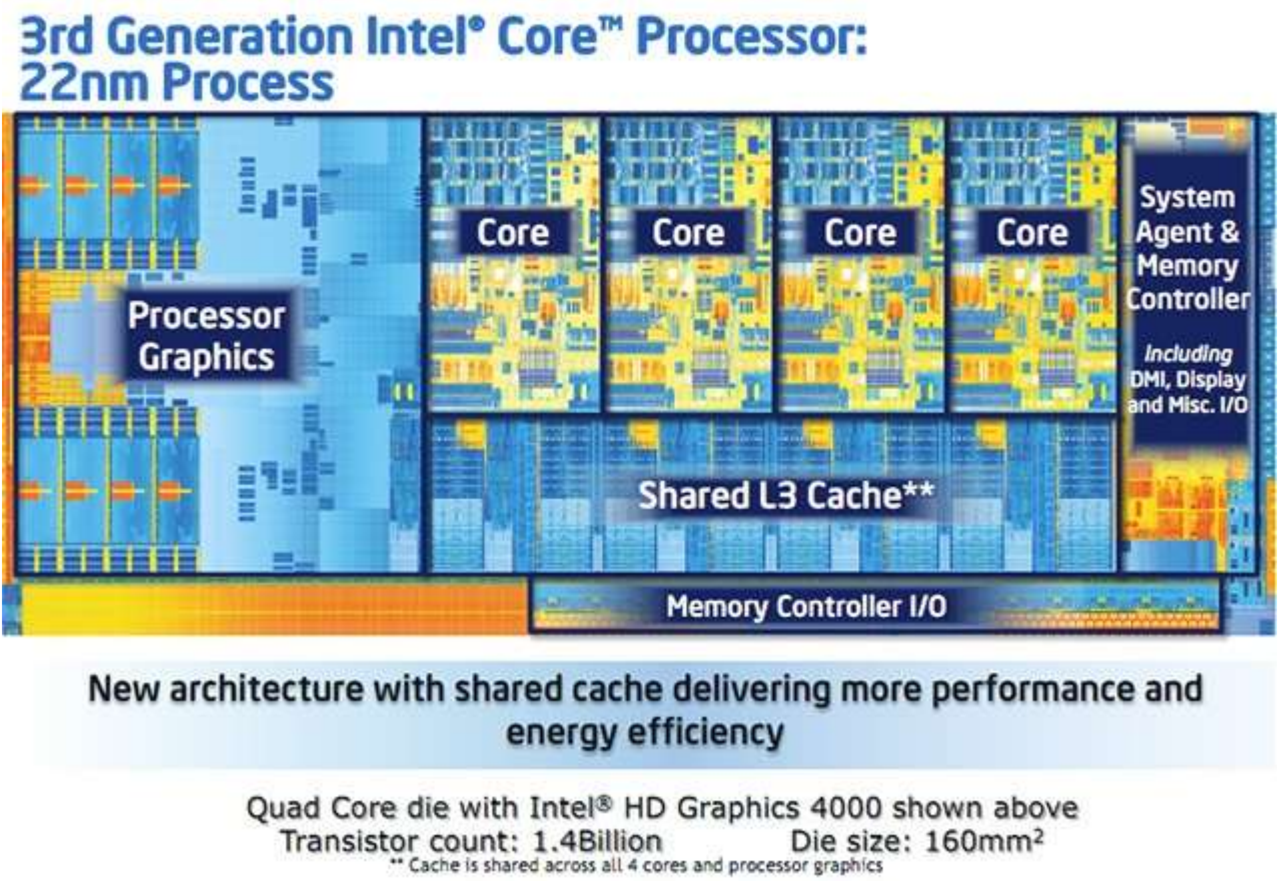
995 million transistors in 216 mm^2 with 32nm technology

Intel’s Core i7 (3rd Gen.)

3rd Generation Core i7

L1	64 KB
L2	256 KB
L3	8MB

1.4 billion transistors in 160 mm^2 with 22nm technology



3rd Generation Intel® Core™ Processor: 22nm Process

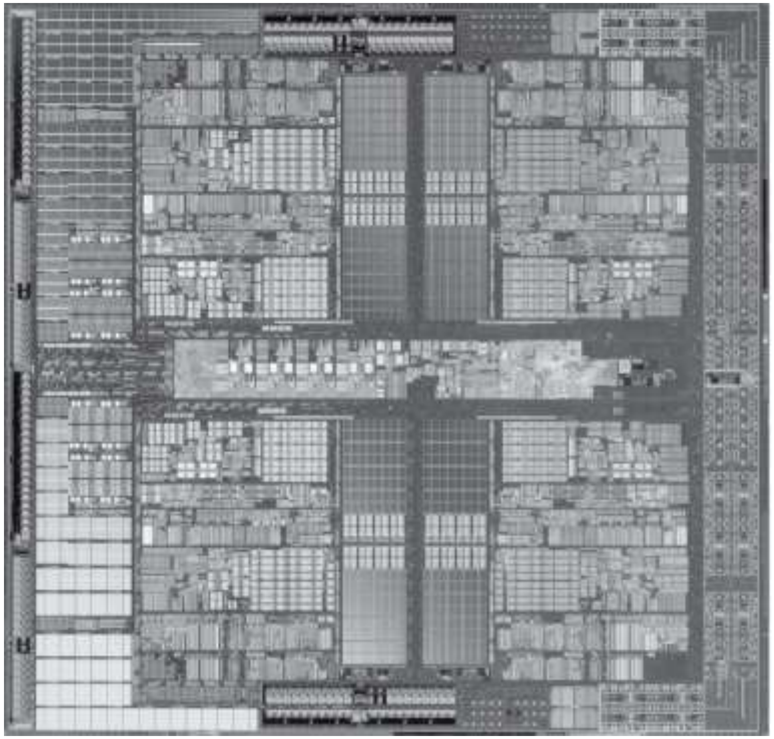
New architecture with shared cache delivering more performance and energy efficiency

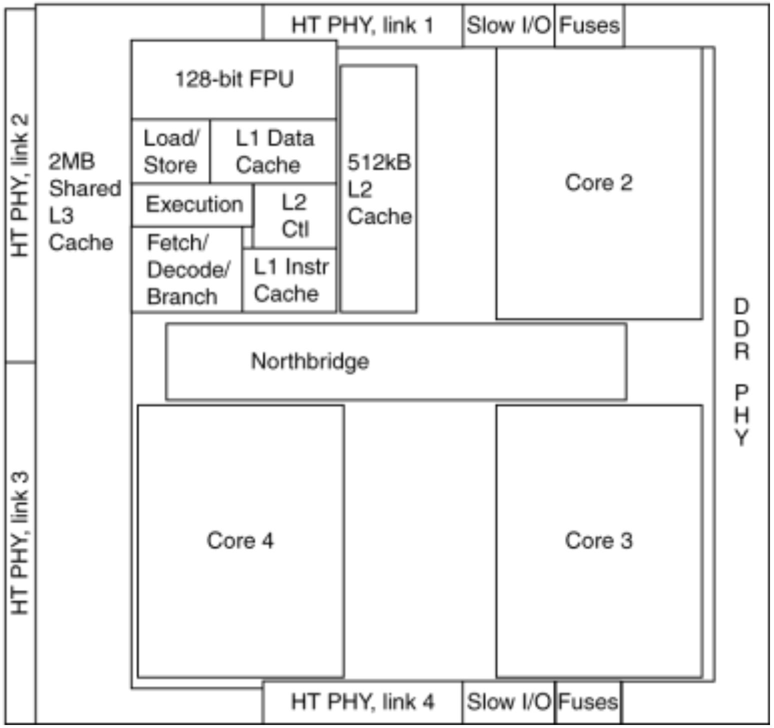
Quad Core die with Intel® HD Graphics 4000 shown above
Transistor count: 1.4Billion Die size: 160mm^2
** Cache is shared across all 4 cores and processor graphics

http://blog.mytechhelp.com/laptop-repair/the-ivy-bridge/

Opteron (AMD) – Barcelona (2007)

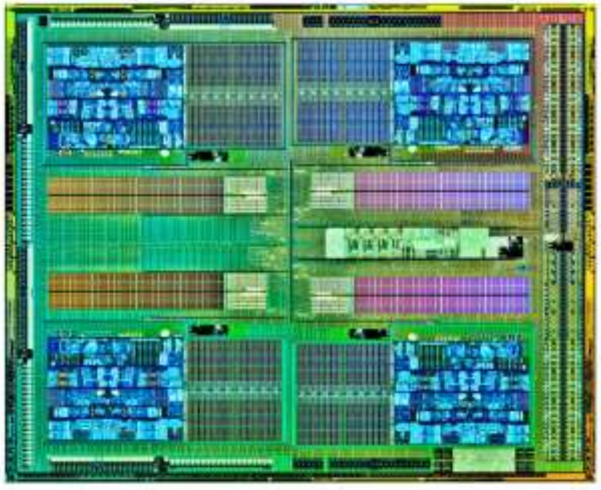
- 4 cores on one chip
- Three levels of caches (L1, L2, L3) on chip
  - **L1: 64KB, L2: 512KB, L3: 2MB**
- Integrated North Bridge





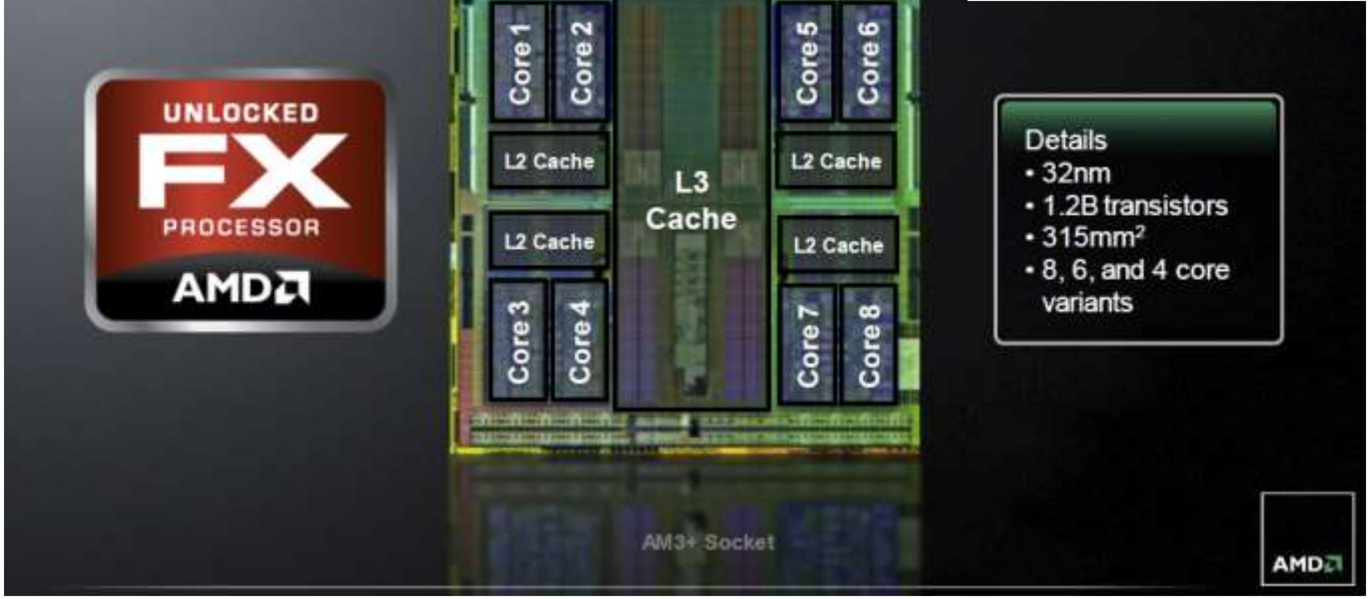
FX-8350 (AMD) – Piledriver (2012)

- 4GHz 8 cores on one chip
- Three levels of caches (L1, L2, L3) on chip
  - **L1: 4 x 64KB shared I$, 8 x 16KB D$**
  - **L2: 4 x 2MB shared $**
  - **L3: 8MB shared $**



2012 AMD FX SERIES OVERVIEW

"Piledriver" Cores

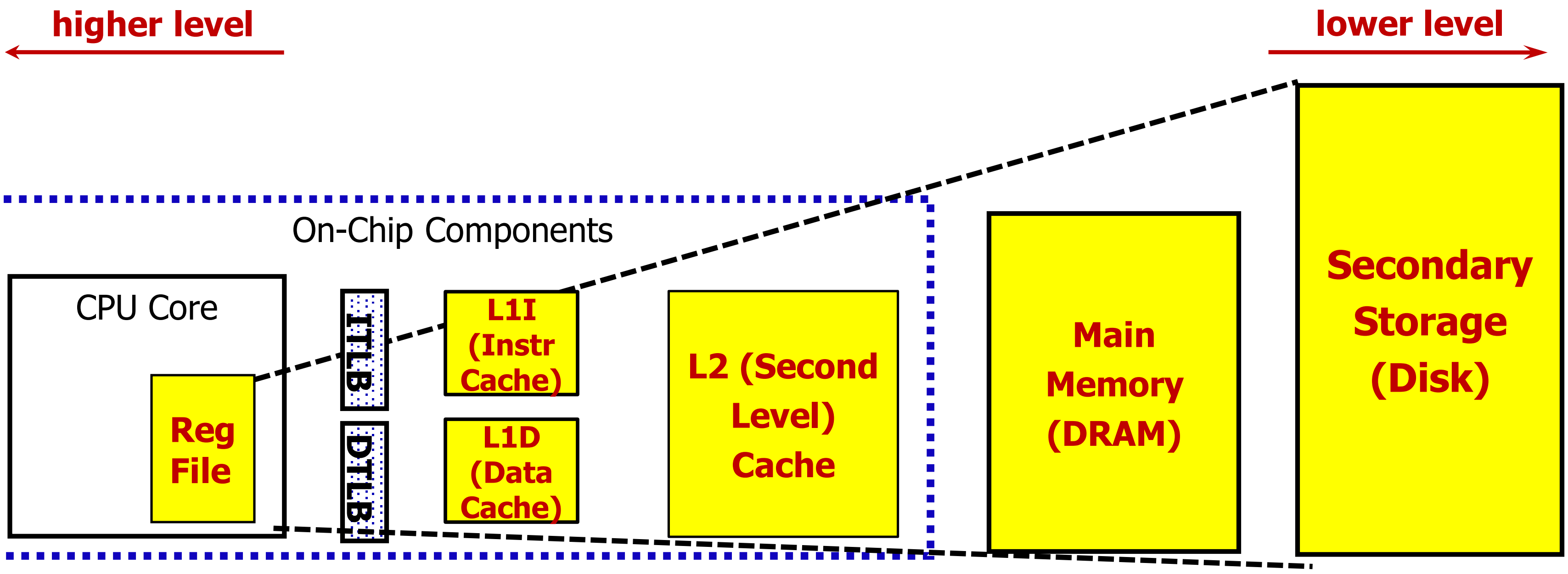


Details
- 32nm
- 1.2B transistors
- 315mm^2
- 8, 6, and 4 core variants

http://hothardware.com/Reviews/AMD-FX-8350-Vishera-8Core-CPU-Review/?page=1

A Typical Memory Hierarchy

- Take advantage of the **principle of locality** to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology

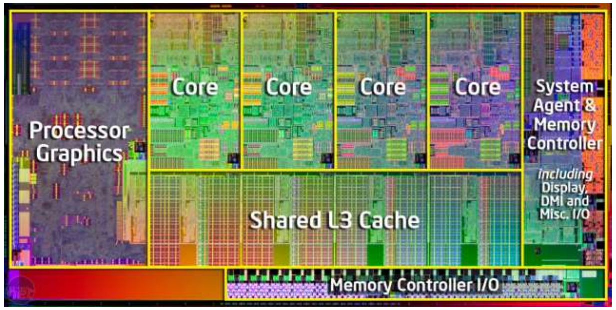


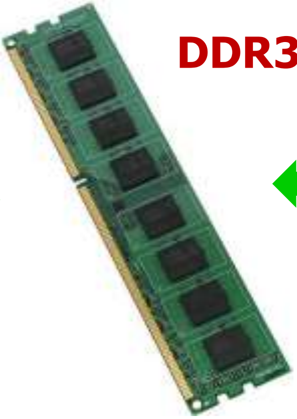
Speed (cycles):	½'s	1's	10's	100's	10,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost:	highest				lowest

How is the Hierarchy Managed?

- Who manages data transfer between
  - Main memory ↔ Disks
    - by the operating system (virtual memory)
    - by the programmer (files)
  - Registers ↔ Main memory
    - by compiler (and programmer)
  - Cache ↔ Main memory
    - by hardware (cache controller)

Sandy Bridge:
2nd Gen. Core i7 (2011)



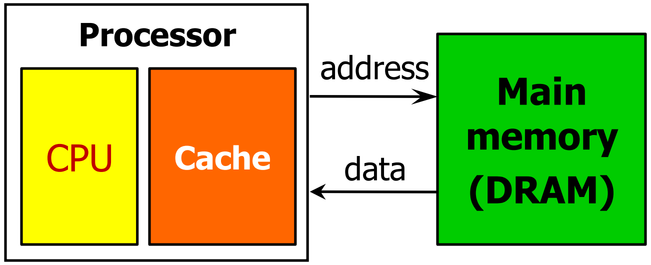


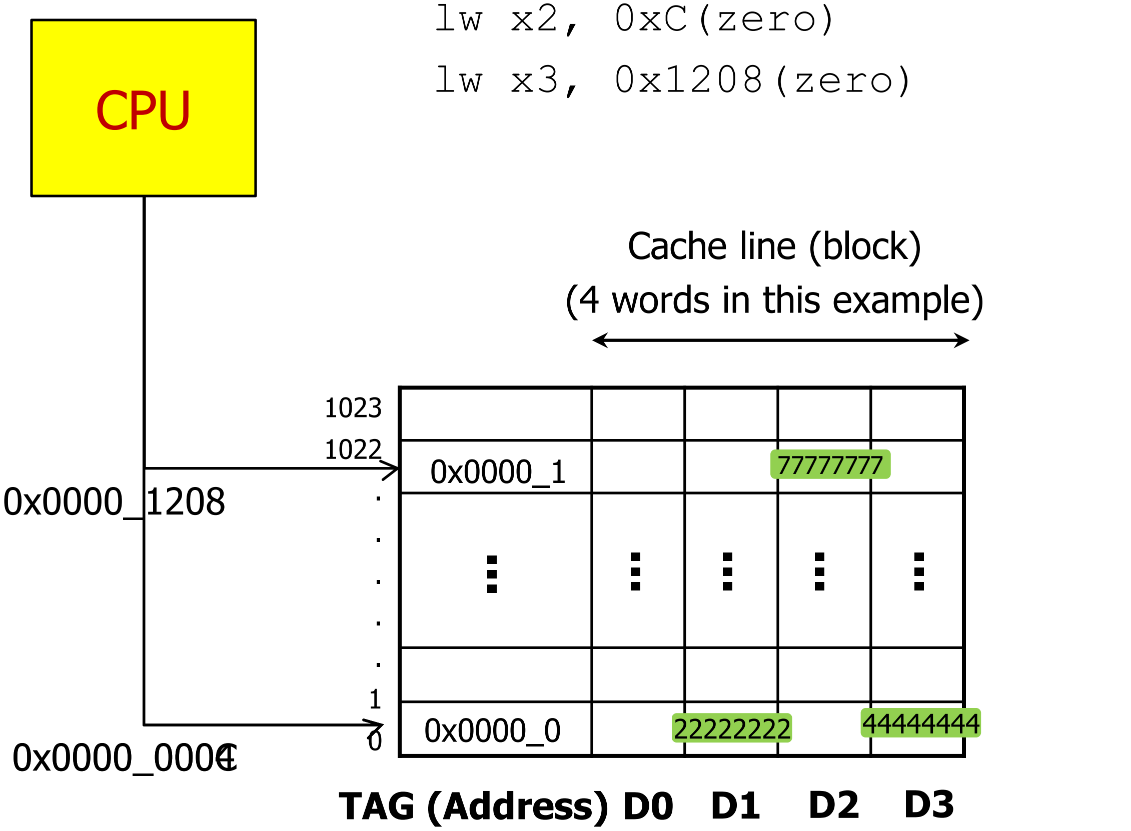
DDR3

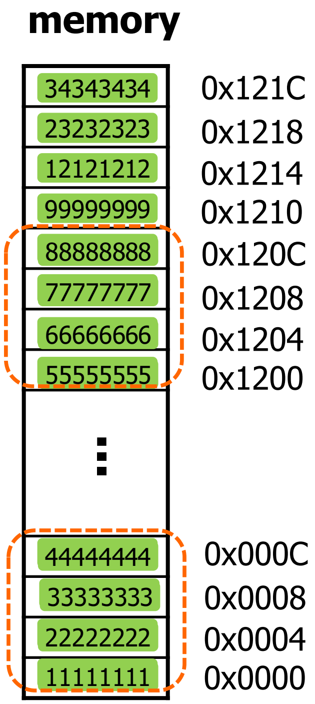
HDD

Basic Cache Operation

```
lw x1, 0x4(zero)
lw x2, 0xC(zero)
lw x3, 0x1208(zero)
```







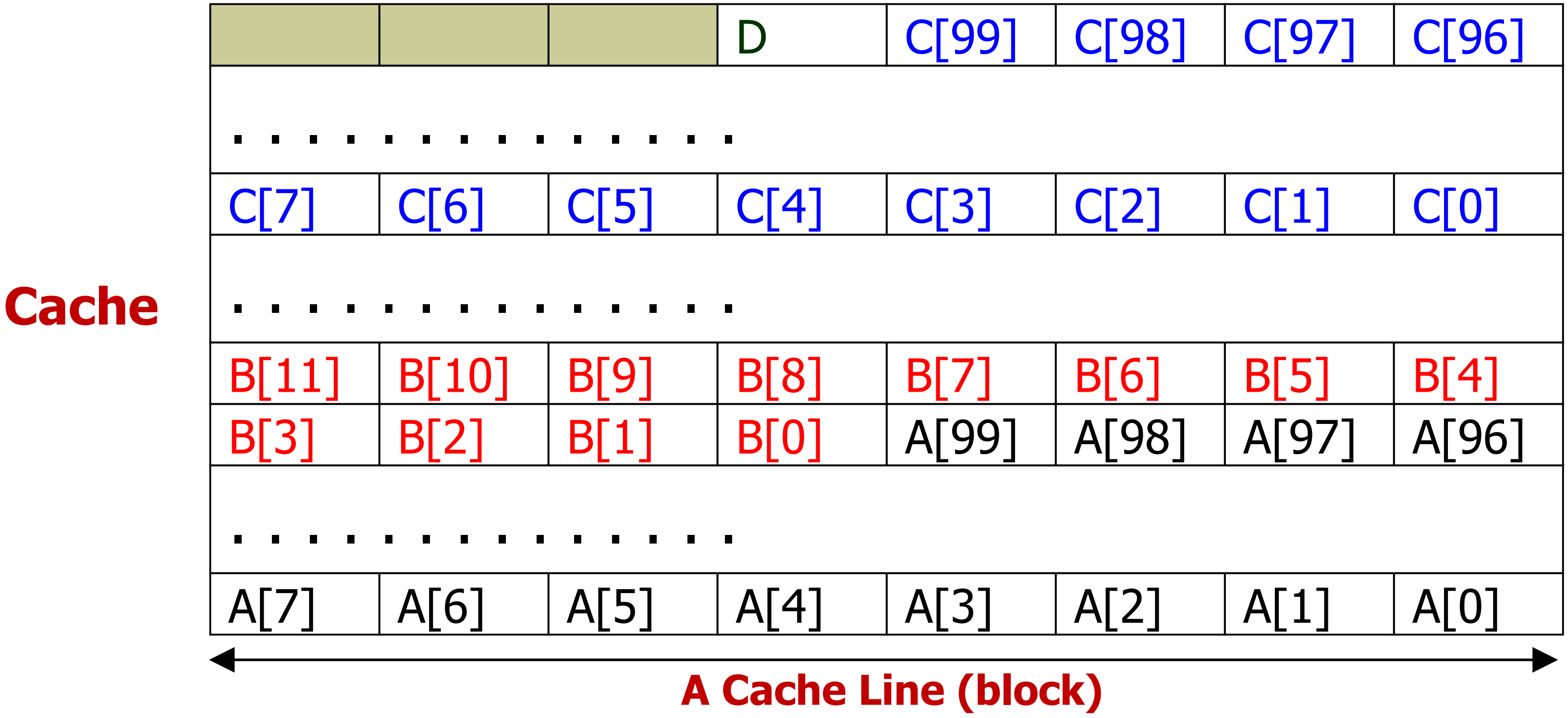
Why Caches Work?

- The size of cache is tiny compared to main memory
  - How to make sure that the data CPU is going to access is in caches?

- Caches take advantage of the principle of locality in your program
  - Temporal Locality (locality in time)
    - If a memory location is referenced, then it will tend to be referenced again soon. So, keep most recently accessed data items closer to the processor
  - Spatial Locality (locality in space)
    - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon. So, move blocks consisting of contiguous words closer to the processor

Example of Locality

```
int A[100], B[100], C[100], D;
for (i=0; i<100; i++) {
            C[i] = A[i] * B[i] + D;
}
```



Cache

			D	C[99]	C[98]	C[97]	C[96]
. . . .							
C[7]	C[6]	C[5]	C[4]	C[3]	C[2]	C[1]	C[0]
. . . .							
B[11]	B[10]	B[9]	B[8]	B[7]	B[6]	B[5]	B[4]
B[3]	B[2]	B[1]	B[0]	A[99]	A[98]	A[97]	A[96]
. . . .							
A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]

A Cache Line (block)

Cache Terminology

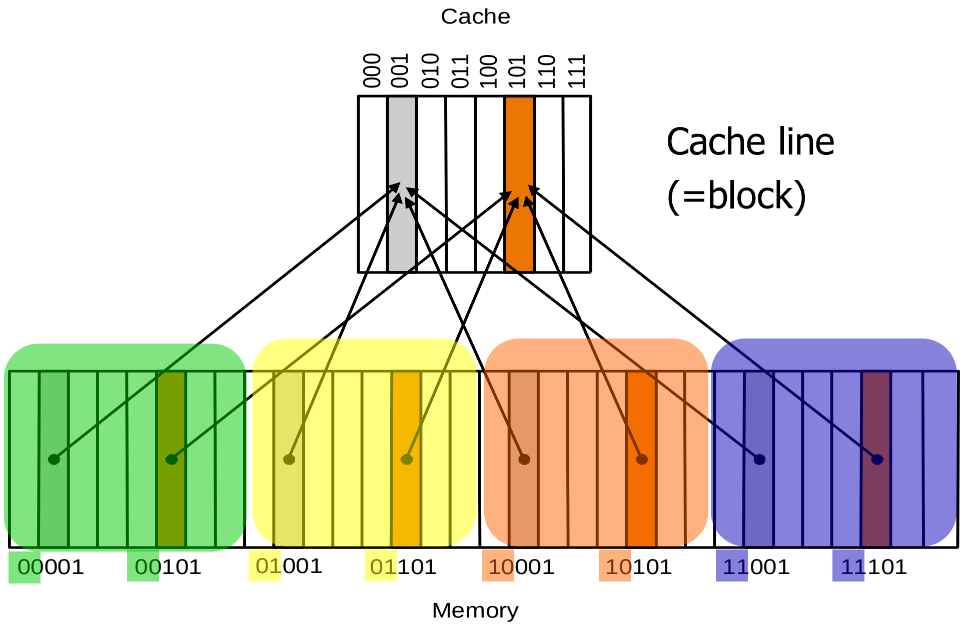
- **Block** (or **(cache) line**): the minimum unit of data present in a cache
  - For example, 64B / block in Core 2 Duo

- **Hit**: if the requested data is in cache, it is called a hit
  - **Hit rate**: the fraction of memory accesses found in caches
    - For example, CPU requested data from memory, and cache was able to supply data 90% of the requests. Then, the hit rate of the cache is 90%
  - **Hit time**: the time required to access the data found in cache

- **Miss**: if the requested data is not in cache, it is called a miss
  - **Miss rate**: the fraction of memory accesses not found in cache (= 1 - Hit Rate)
  - **Miss penalty**: the time required to fetch a block into a level of the memory hierarchy from the lower level

Direct-mapped Cache

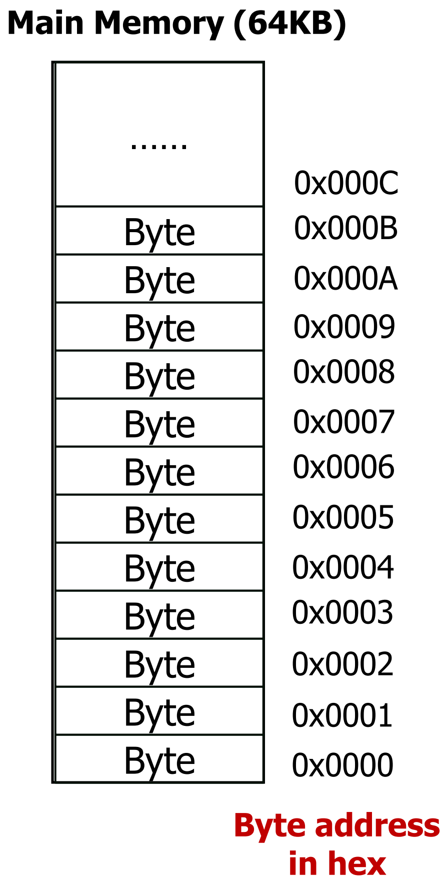
- The simplest cache structure: **Direct mapped cache**
  - Each memory block is mapped to exactly one block in cache
    - Lots of memory blocks must **share** a block in cache
  - Address mapping to cache
    - (block address) modulo (# of blocks in cache)
  - Have a **tag** associated with each cache block that contains the address information
    - The tag is the upper portion of the address required to identify the block



Valid	Tag	Data
	11	🟦
	11	🟦
	00	🟩
	10	🟧
	01	🟨
	00	🟩
	11	🟦
	10	🟧

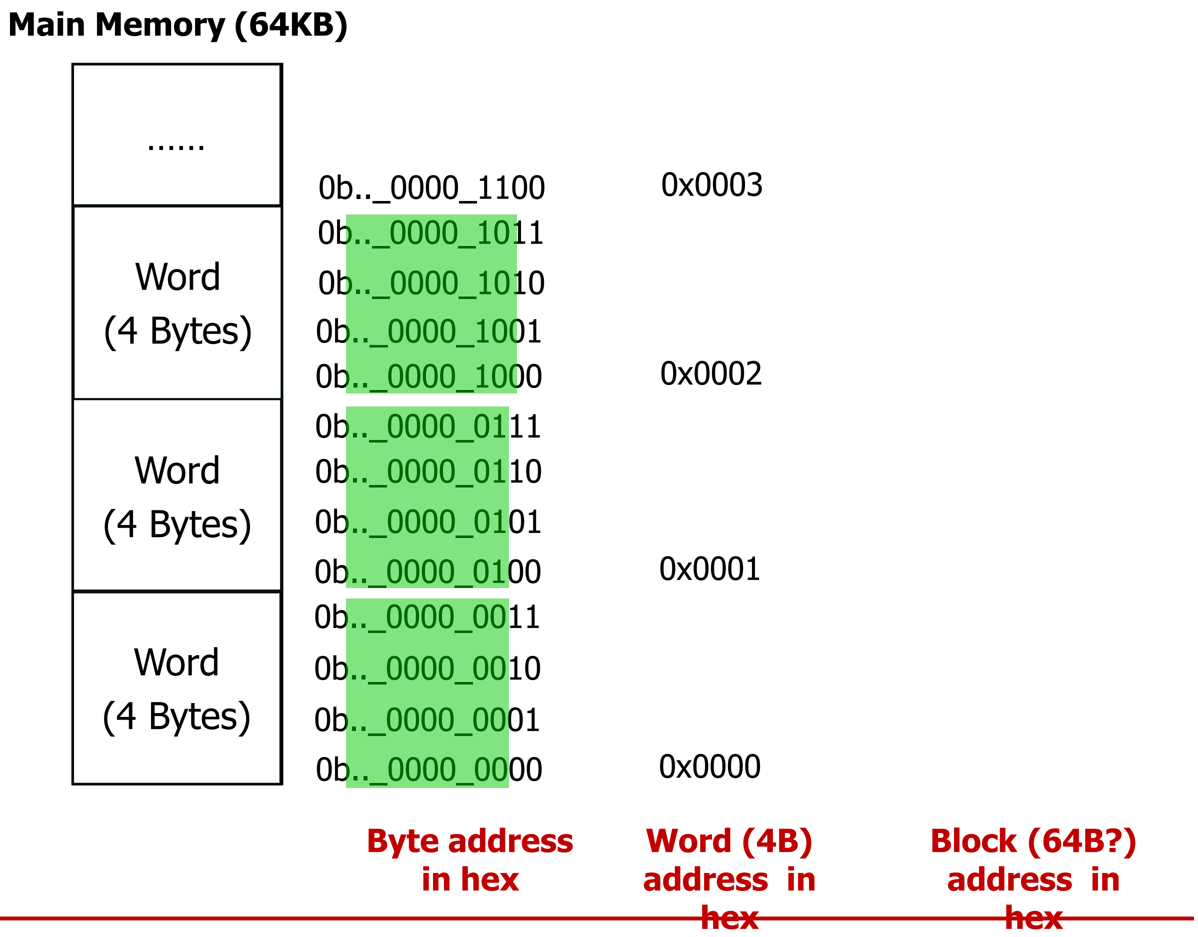
Memory Address

- Byte-address
- Word-address



Main Memory (64KB)

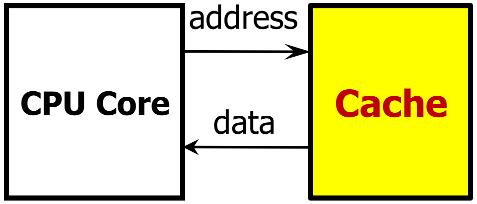
Main Memory (64KB)	Byte address in hex
......	0x000C
Byte	0x000B
Byte	0x000A
Byte	0x0009
Byte	0x0008
Byte	0x0007
Byte	0x0006
Byte	0x0005
Byte	0x0004
Byte	0x0003
Byte	0x0002
Byte	0x0001
Byte	0x0000



Main Memory (64KB)

Main Memory (64KB)	Byte address in hex	Word (4B) address in hex	Block (64B?) address in hex
......	0b.._0000_1100	0x0003	
Word (4 Bytes)	0b.._0000_1011		
	0b.._0000_1010		
	0b.._0000_1001		
	0b.._0000_1000	0x0002	
Word (4 Bytes)	0b.._0000_0111		
	0b.._0000_0110		
	0b.._0000_0101		
	0b.._0000_0100	0x0001	
Word (4 Bytes)	0b.._0000_0011		
	0b.._0000_0010		
	0b.._0000_0001		
	0b.._0000_0000	0x0000	

Memory Address

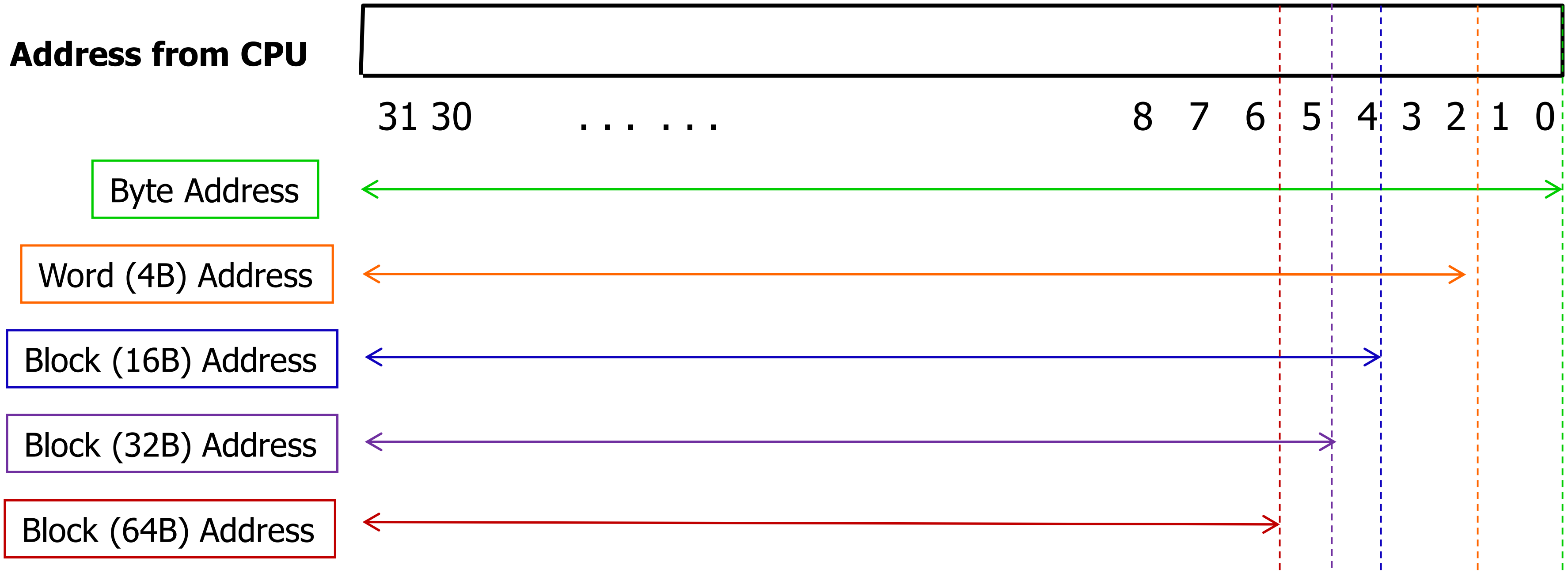


CPU Core

address

data

Cache



Address from CPU

31 30 . . . . . . 8 7 6 5 4 3 2 1 0

Byte Address

Word (4B) Address

Block (16B) Address

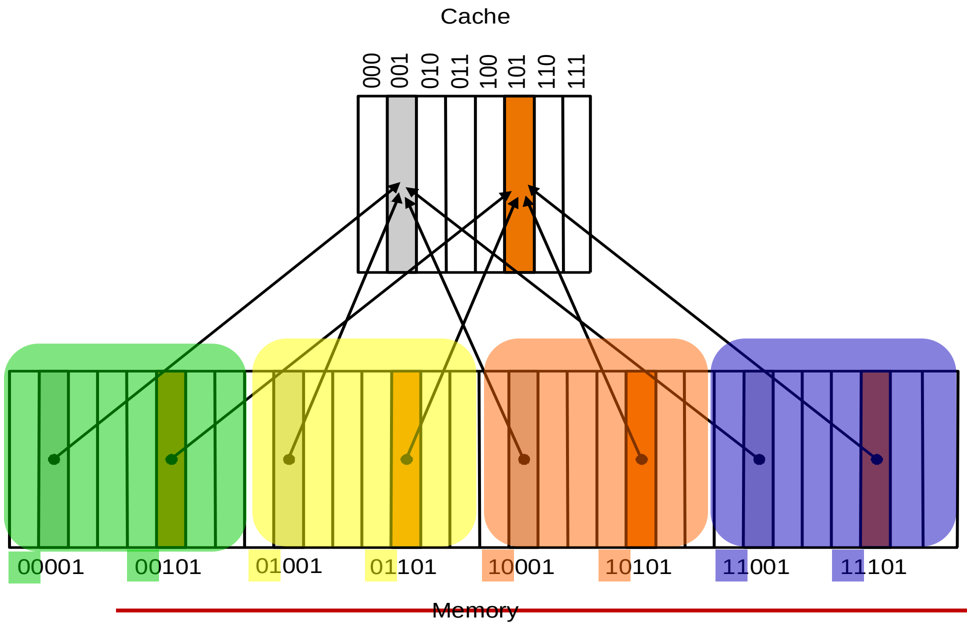
Block (32B) Address

Block (64B) Address

Direct-mapped Cache

- Mapping to cache
  - (block address) modulo (# of blocks in the cache)

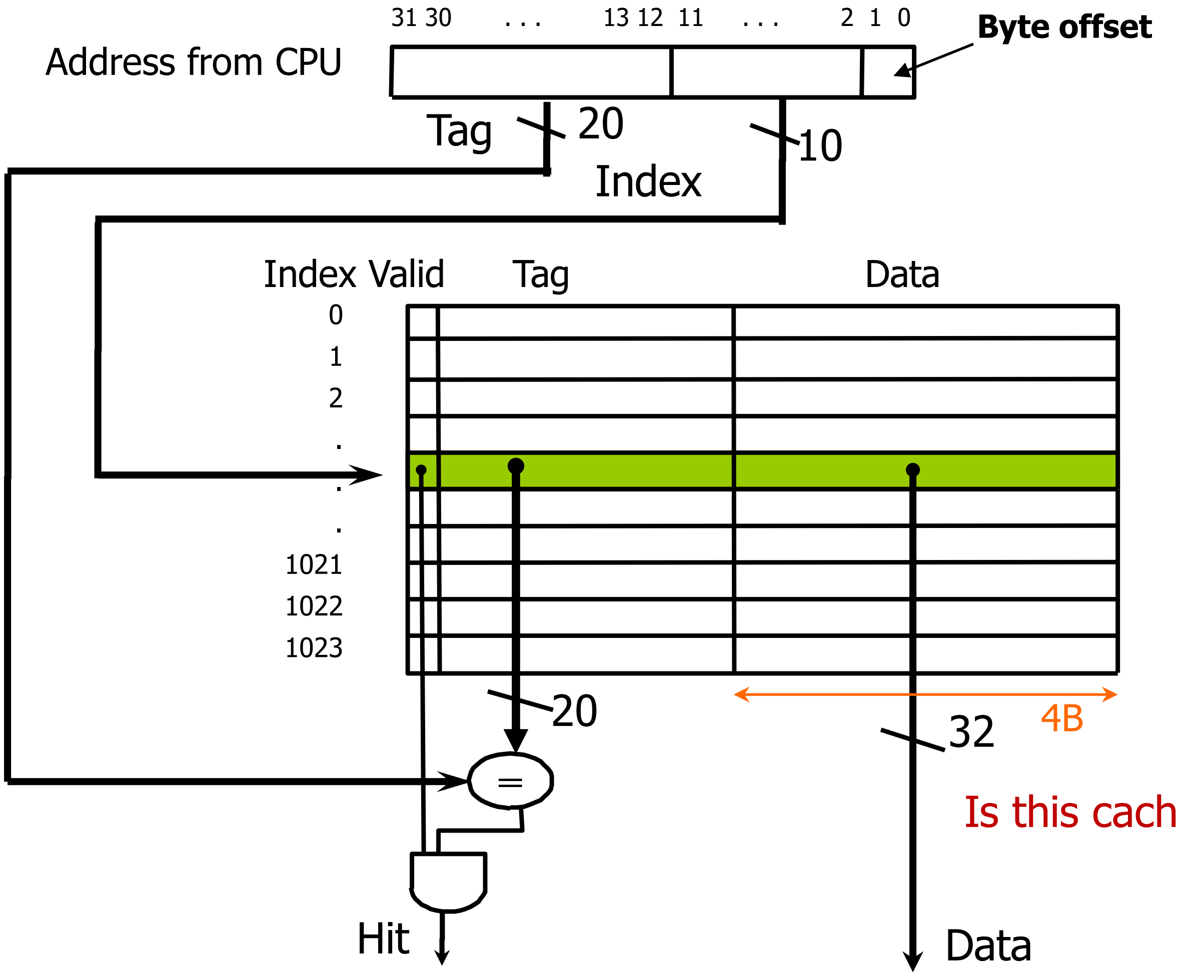
- Cache structure
  - **Data**: actual data
  - **Tag**: which block is mapped to the cache?
  - **Valid**: Is the block in the cache valid?

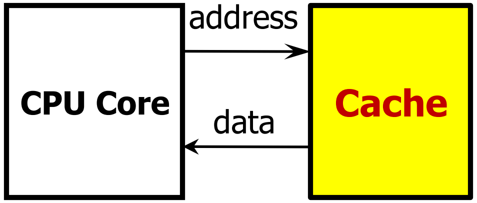


Valid	Tag	Data

Example

- 4KB direct-mapped cache with 1 word (32-bit) blocks
- How many blocks (cache lines) are there in the cache?





Is this cache structure taking advantage of what kind of locality?

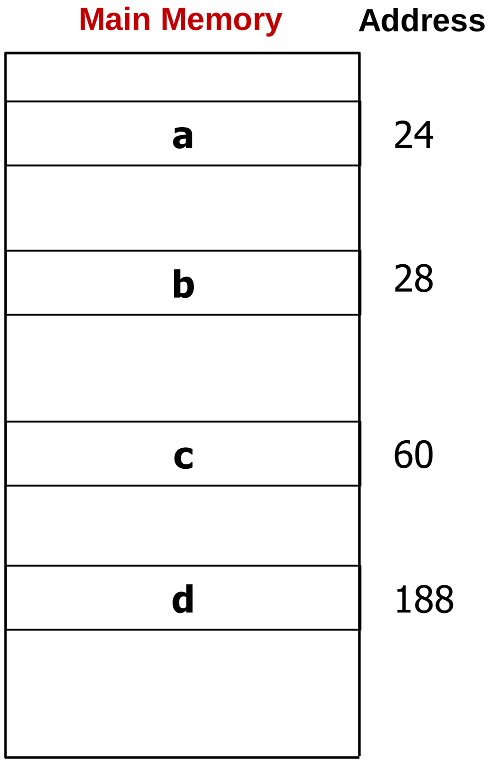
Example: DM$, 8-Entry, 4B blocks

- Assume that address bus from CPU is 8-bit wide

```
lw x1, 24(zero)
lw x2, 28(zero)
sw x3, 60(zero)
sw x4, 188(zero)
```

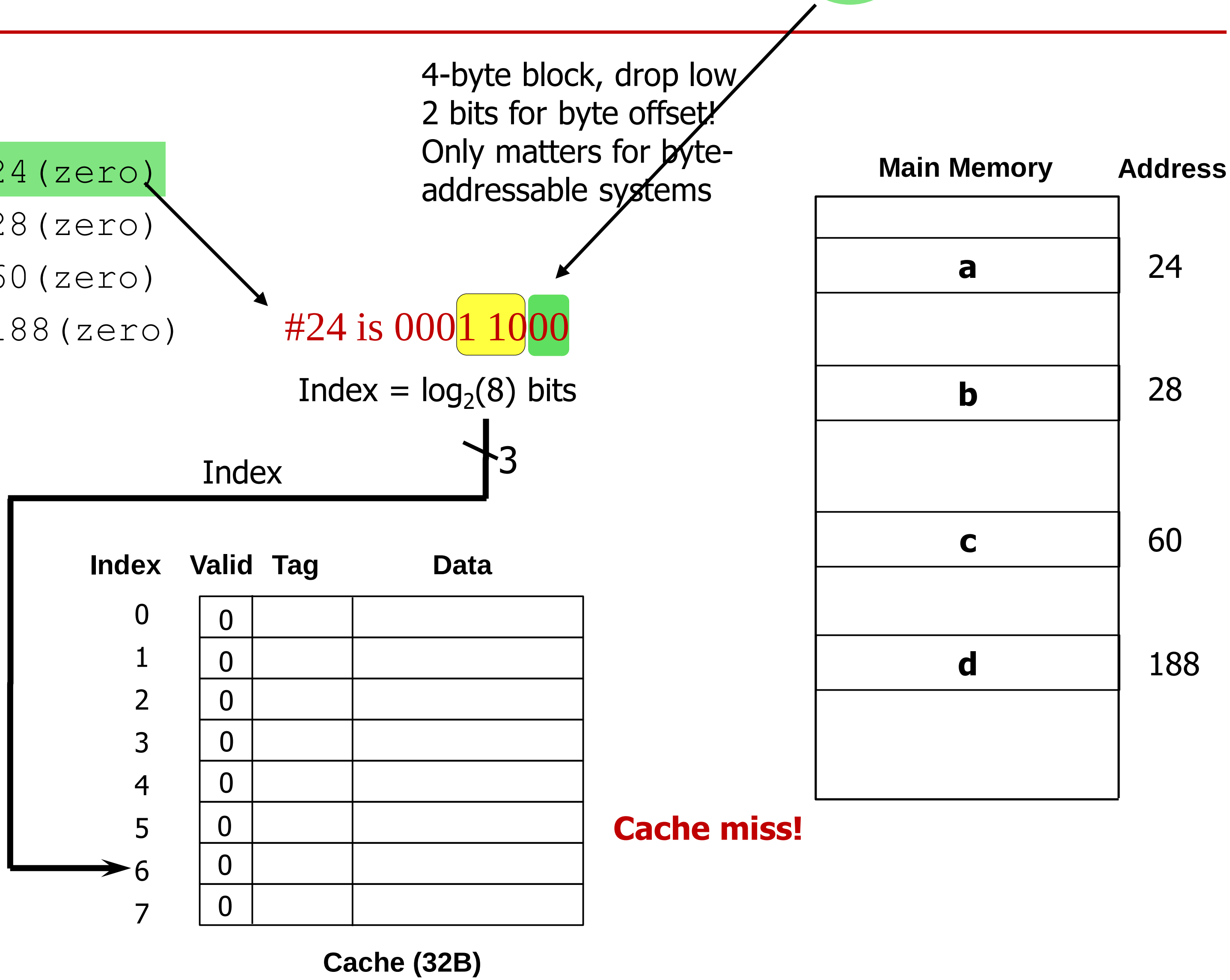
Index	Valid	Tag	Data
0			
1			
2			
3			
4			
5			
6			
7			

Cache (32B)



Example: DM$, 8-Entry, 4B blocks

```
lw x1, 24(zero)
lw x2, 28(zero)
sw x3, 60(zero)
sw x4, 188(zero)
```



4-byte block, drop low 2 bits for byte offset! Only matters for byte-addressable systems

#24 is 0001 1000

Index = $\log_2(8)$ bits

Index	Valid	Tag	Data
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		

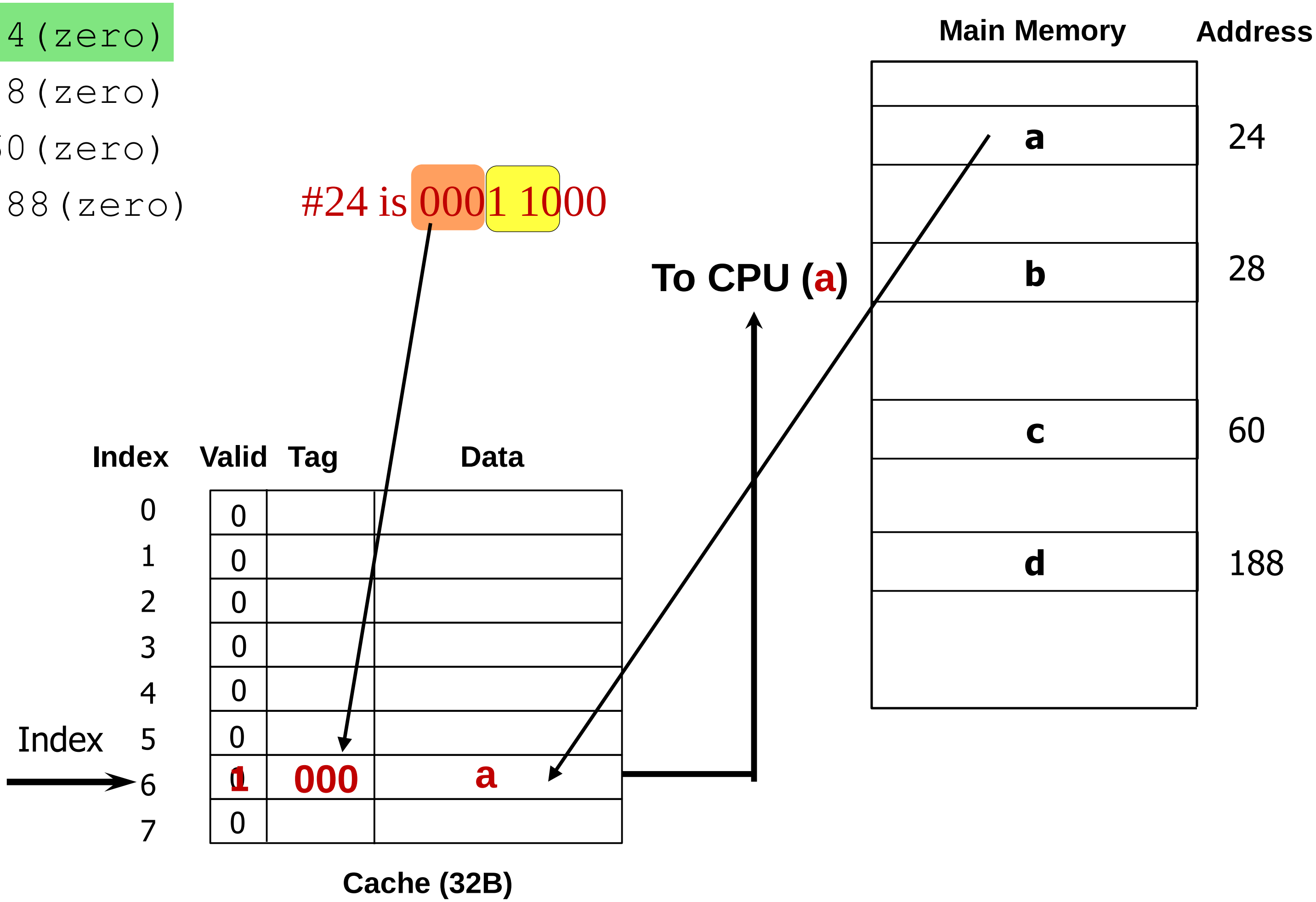
Cache (32B)

Cache miss!

Main Memory	Address
a	24
b	28
c	60
d	188

Example: DM$, 8-Entry, 4B blocks

```
lw x1, 24(zero)
lw x2, 28(zero)
sw x3, 60(zero)
sw x4, 188(zero)
```



#24 is 0001 1000

To CPU (a)

Index

Index	Valid	Tag	Data
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	1	000	a
7	0		

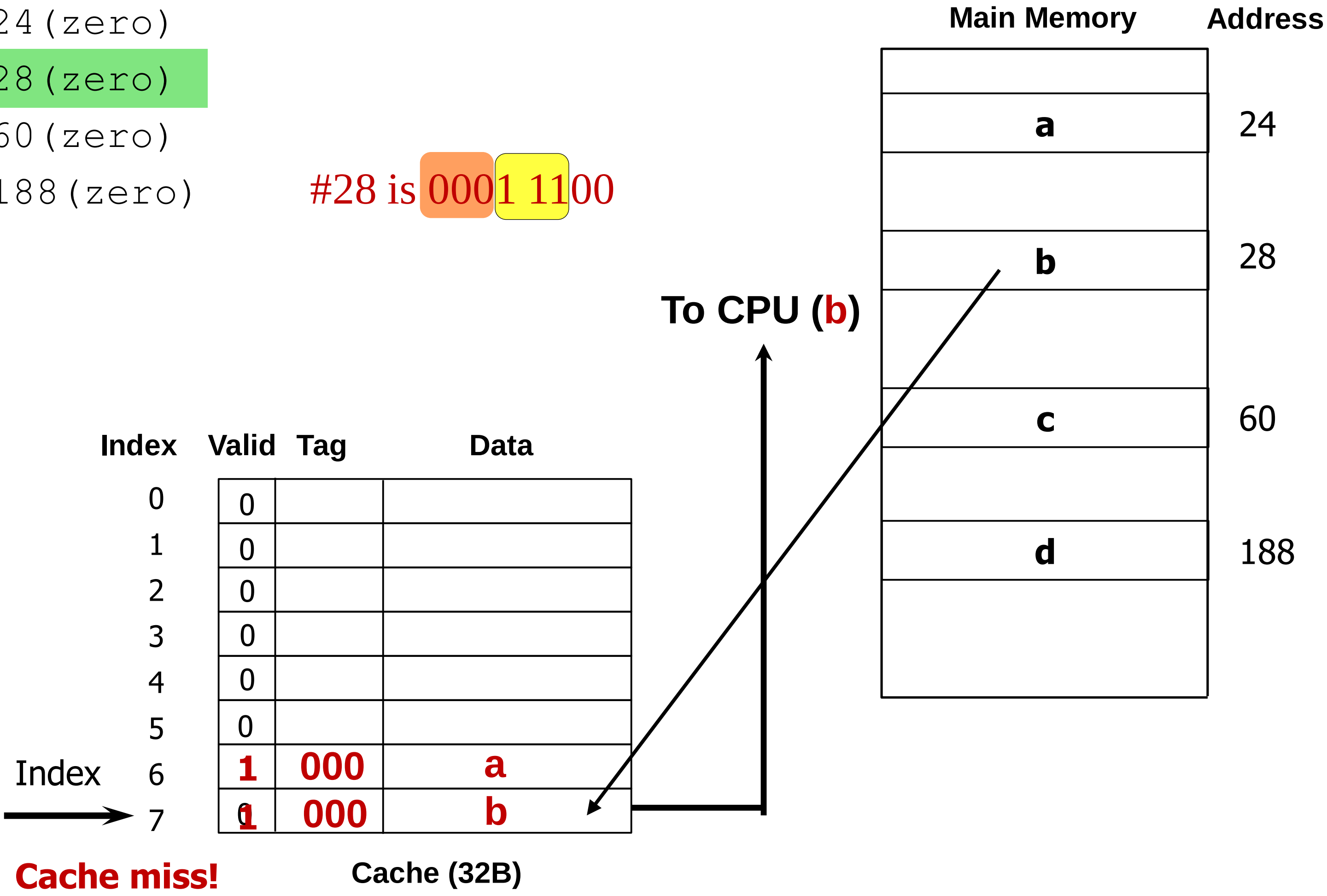
Cache (32B)

Main Memory	Address
a	24
b	28
c	60
d	188

Example: DM$, 8-Entry, 4B blocks

```
lw x1, 24(zero)
lw x2, 28(zero)
sw x3, 60(zero)
sw x4, 188(zero)
```

#28 is 0001 1100



Index	Valid	Tag	Data
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	1	000	a
7	1	000	b

Cache miss!

Cache (32B)

Example: DM$, 8-Entry, 4B blocks

```
lw x1, 24(zero)
lw x2, 28(zero)
sw x3, 60(zero)
sw x4, 188(zero)
```

#60 is 0011 1100

Main Memory	Address
a	24
b	28
c	60
d	188

Index	Valid	Tag	Data
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	**1**	**000**	a
7	**1**	**000**	b

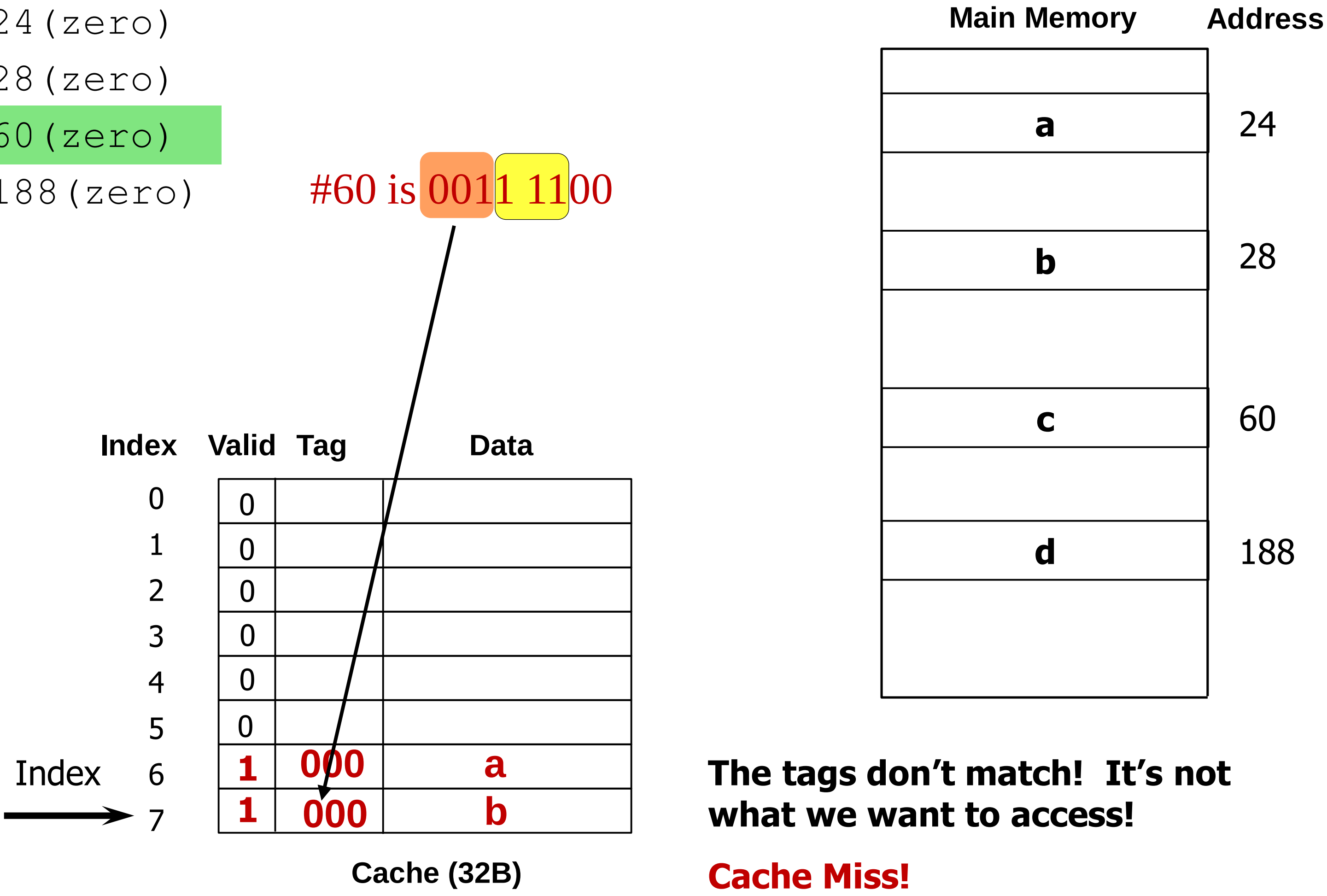
Index → 7

Cache (32B)

It's valid! Is it a hit or a miss?

Example: DM$, 8-Entry, 4B blocks

```
lw x1, 24(zero)
lw x2, 28(zero)
sw x3, 60(zero)
sw x4, 188(zero)
```



Index	Valid	Tag	Data
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	1	000	a
7	1	000	b

Cache (32B)

Main Memory	Address
a	24
b	28
c	60
d	188

The tags don’t match! It’s not what we want to access!

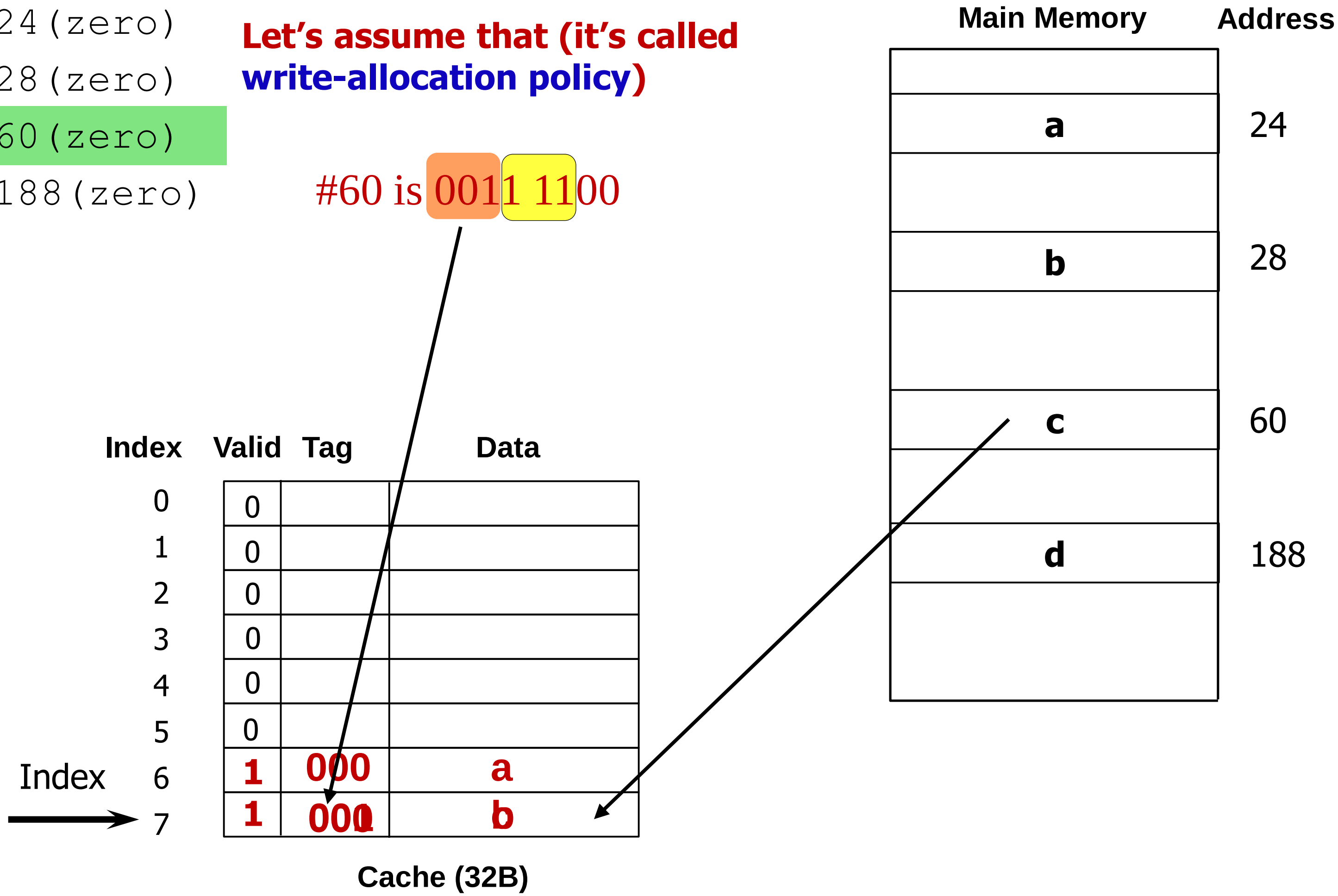
Cache Miss!

Example: DM$, 8-Entry, 4B blocks

```
lw x1, 24(zero)
lw x2, 28(zero)
sw x3, 60(zero)
sw x4, 188(zero)
```

Do we have to bring the block to the cache and write to cache?

Let's assume that (it's called **write-allocation policy**)



#60 is 0011 1100

Index	Valid	Tag	Data
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	1	000	a
7	1	000	b

Cache (32B)

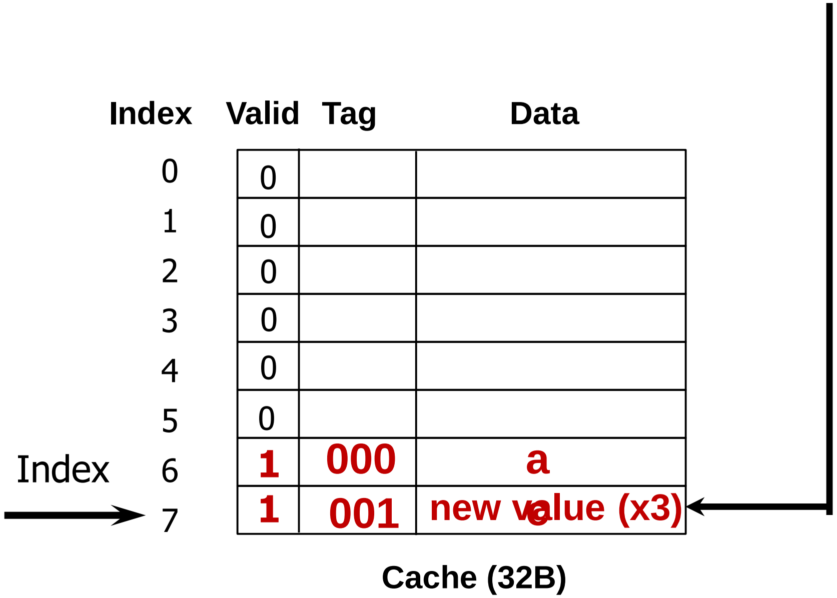
Main Memory	Address
a	24
b	28
c	60
d	188

Example: DM$, 8-Entry, 4B blocks

```
lw x1, 24(zero)
lw x2, 28(zero)
sw x3, 60(zero)
sw x4, 188(zero)
```

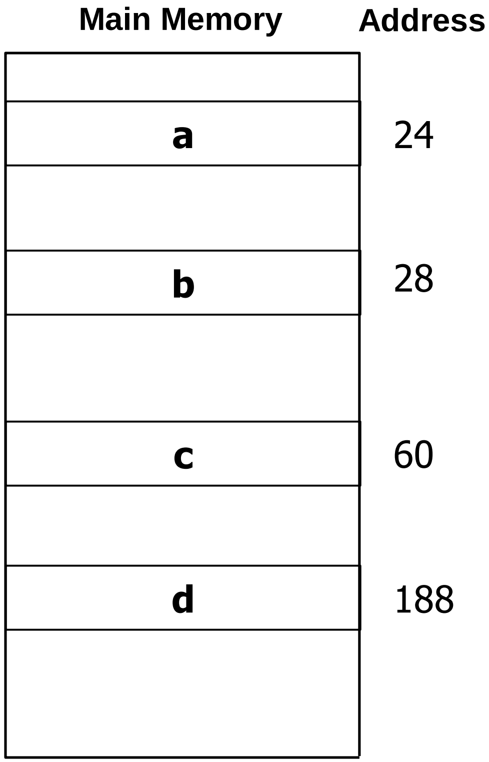
Now, we can write a new value to the location

#60 is 0011 1100



Index	Valid	Tag	Data
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	**1**	000	a
7	**1**	001	new value (x3) c

Cache (32B)



Main Memory	Address
a	24
b	28
c	60
d	188

Do we update memory now? Or later?

Assume later (it is called write-back cache)

Example: DM$, 8-Entry, 4B blocks

```
lw x1, 24(zero)
lw x2, 28(zero)
sw x3, 60(zero)
sw x4, 188(zero)
```

How do we know which blocks in cache need to be written back to main memory?

Need extra state!
The “dirty” bit!

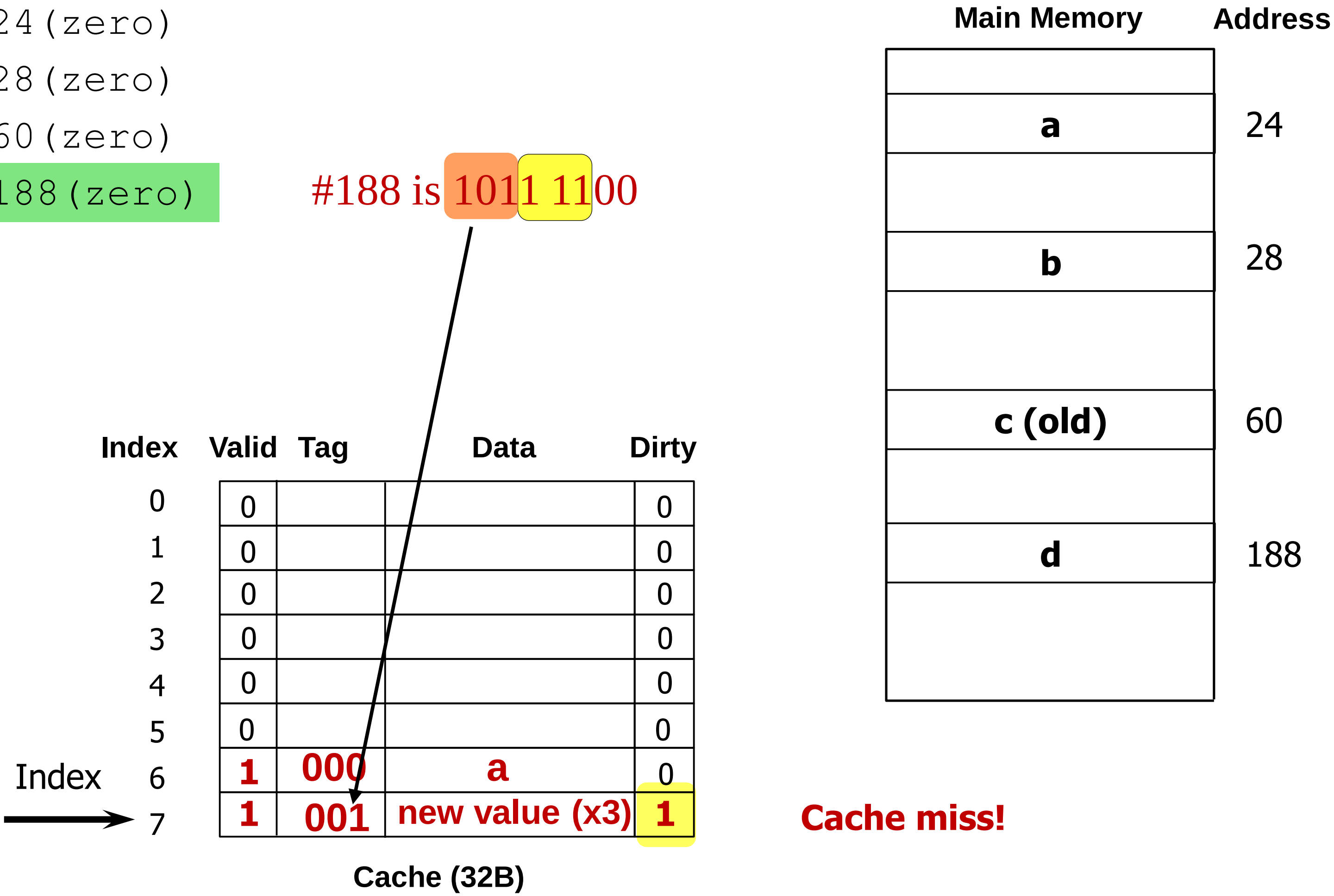
Main Memory	Address
a	24
b	28
c (old)	60
d	188

Index	Valid	Tag	Data	Dirty
0	0			0
1	0			0
2	0			0
3	0			0
4	0			0
5	0			0
6	1	000	a	0
7	1	001	new value (x3)	1

Cache (32B)

Example: DM$, 8-Entry, 4B blocks

```
lw x1, 24(zero)
lw x2, 28(zero)
sw x3, 60(zero)
sw x4, 188(zero)
```



#188 is 1011 1100

Index	Valid	Tag	Data	Dirty
0	0			0
1	0			0
2	0			0
3	0			0
4	0			0
5	0			0
6	1	000	a	0
7	1	001	new value (x3)	1

Index

Cache (32B)

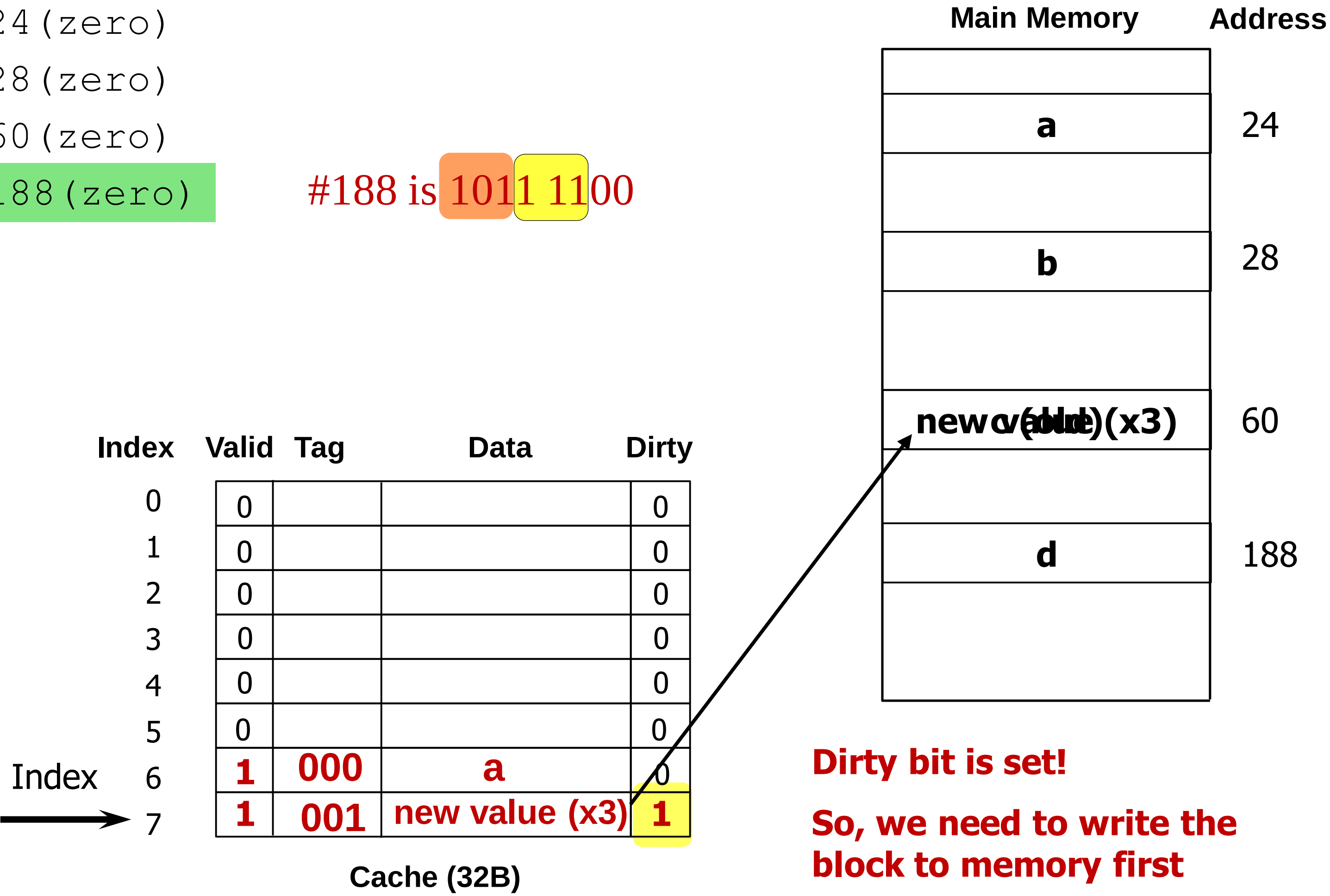
Main Memory	Address
a	24
b	28
c (old)	60
d	188

Cache miss!

Example: DM$, 8-Entry, 4B blocks

```
lw  x1, 24(zero)
lw  x2, 28(zero)
sw  x3, 60(zero)
sw  x4, 188(zero)
```

#188 is 1011 1100



Index	Valid	Tag	Data	Dirty
0	0			0
1	0			0
2	0			0
3	0			0
4	0			0
5	0			0
6	1	000	a	0
7	1	001	new value (x3)	1

Cache (32B)

Main Memory	Address
a	24
b	28
new value (x3)	60
d	188

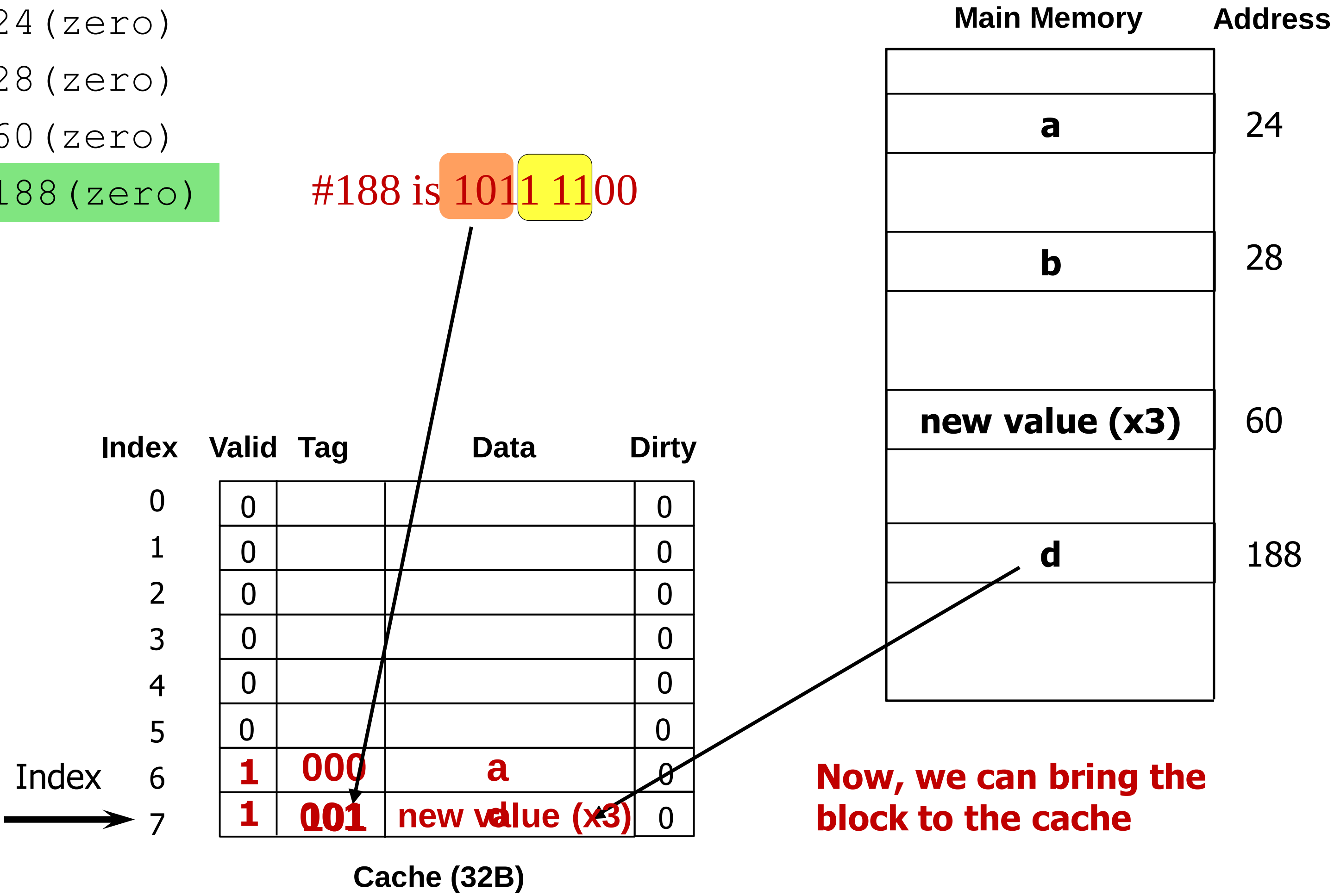
Dirty bit is set!

So, we need to write the block to memory first

Example: DM$, 8-Entry, 4B blocks

```
lw x1, 24(zero)
lw x2, 28(zero)
sw x3, 60(zero)
sw x4, 188(zero)
```

#188 is 1011 1100



Index	Valid	Tag	Data	Dirty
0	0			0
1	0			0
2	0			0
3	0			0
4	0			0
5	0			0
6	1	000	a	0
7	1	101	new value (x3)	0

Cache (32B)

Main Memory	Address
a	24
b	28
new value (x3)	60
d	188

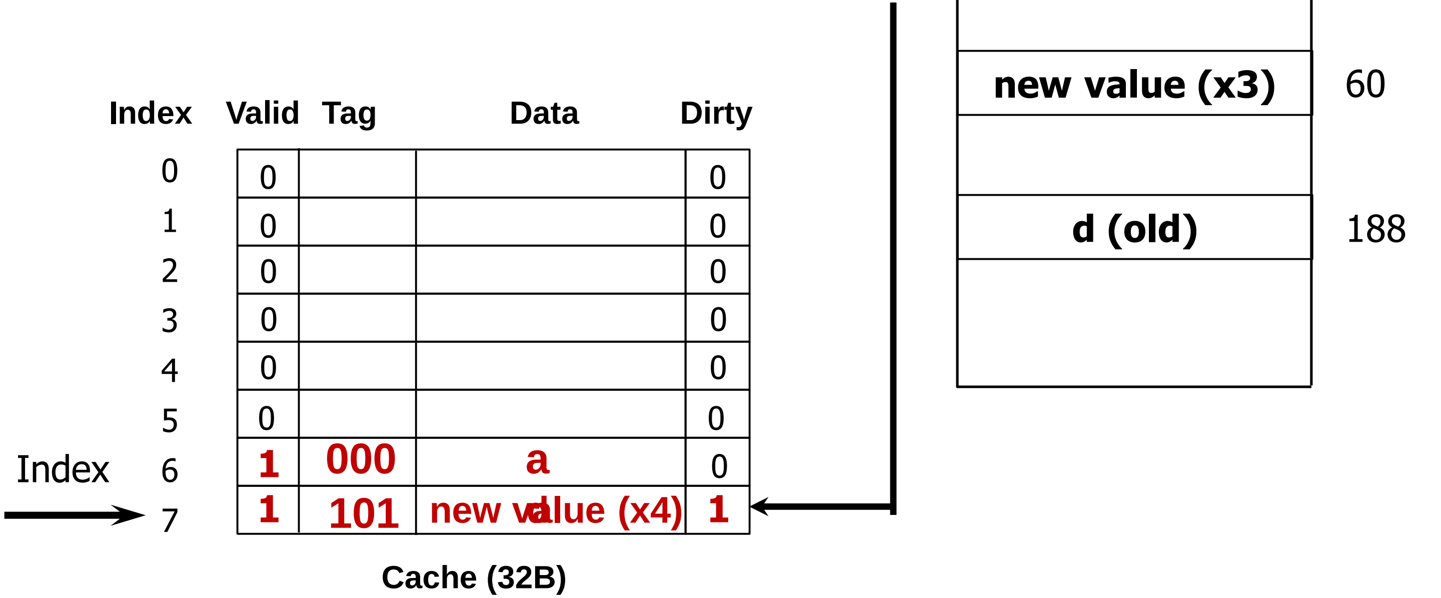
Now, we can bring the block to the cache

Example: DM$, 8-Entry, 4B blocks

```
lw  x1,  24(zero)
lw  x2,  28(zero)
sw  x3,  60(zero)
sw  x4,  188(zero)
```

Now, we can write a new value to the location

#188 is 1011 1100



Index	Valid	Tag	Data	Dirty
0	0			0
1	0			0
2	0			0
3	0			0
4	0			0
5	0			0
6	1	000	a	0
7	1	101	new value (x4)	1

Cache (32B)

Main Memory	Address
a	24
b	28
new value (x3)	60
d (old)	188

Handling Writes

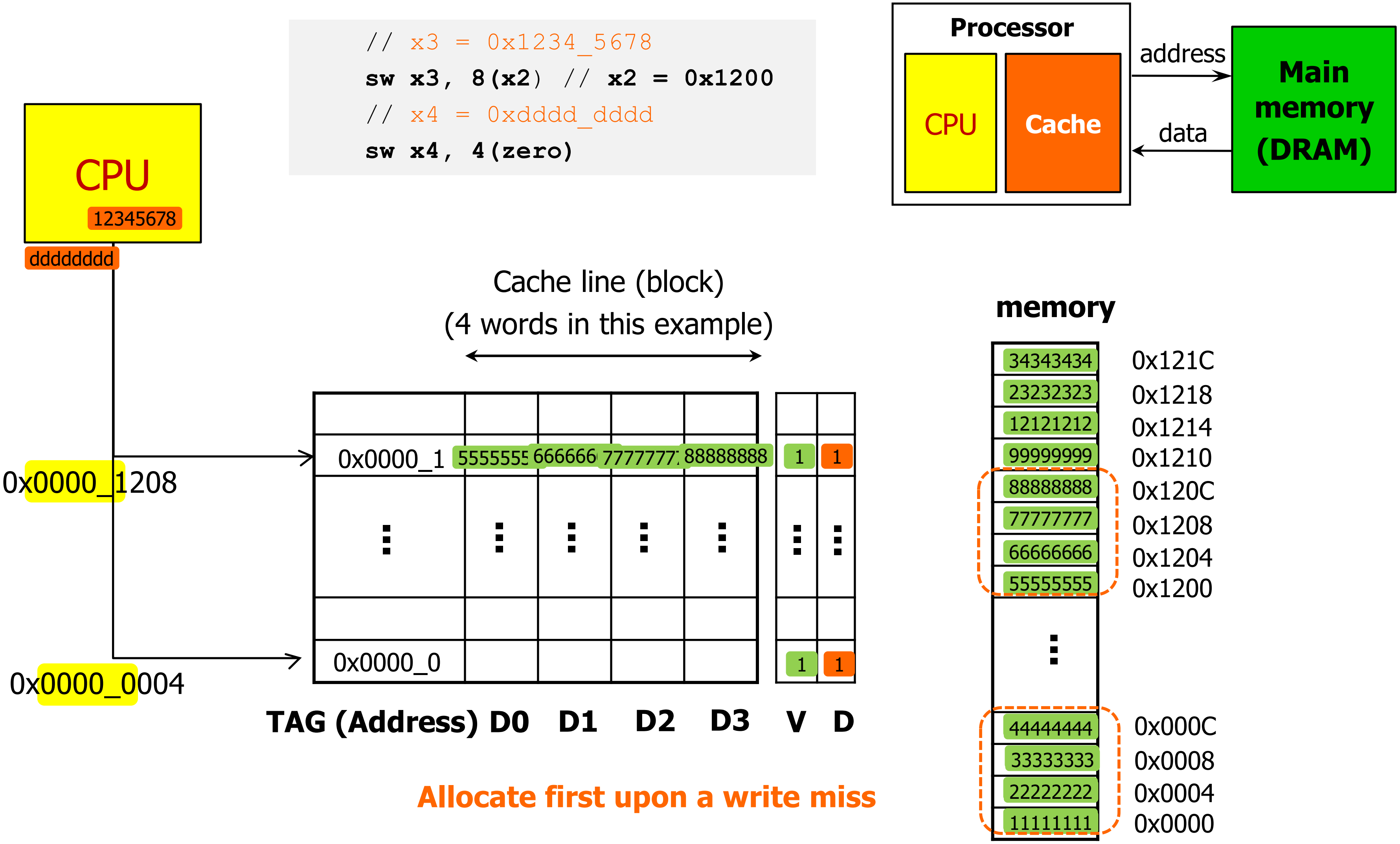
- When it comes to a read miss, we bring the block to cache and supply data to CPU

- When it comes to a write miss, there are options you can choose from
  - Upon a write-miss,
    - **Write-allocate**: bring the block into cache and write
    - **Write no-allocate**: write directly to main memory w/o bringing the block to the cache
  - When writing,
    - **Write-back**: update values only to the blocks in the cache and write the modified blocks to memory (or the lower level of the hierarchy) when the block is replaced
    - **Write-through**: update both the cache and the lower level of the memory hierarchy

- Write-allocate is usually associated with write-back policy

- Write no-allocate is usually associated with write-through policy

Write-Allocate & Write-back



```
// x3 = 0x1234_5678
sw x3, 8(x2) // x2 = 0x1200
// x4 = 0xdddd_dddd
sw x4, 4(zero)
```

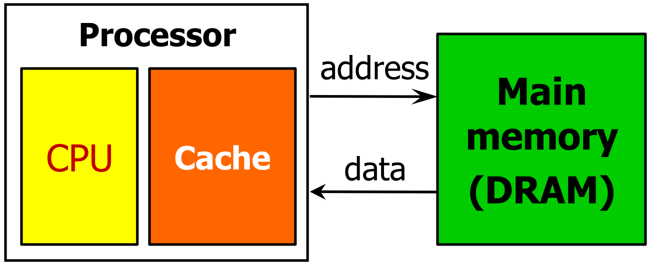
TAG (Address)	D0	D1	D2	D3	V	D
0x0000_1	5555555	666666	7777777	88888888	1	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0x0000_0					1	1

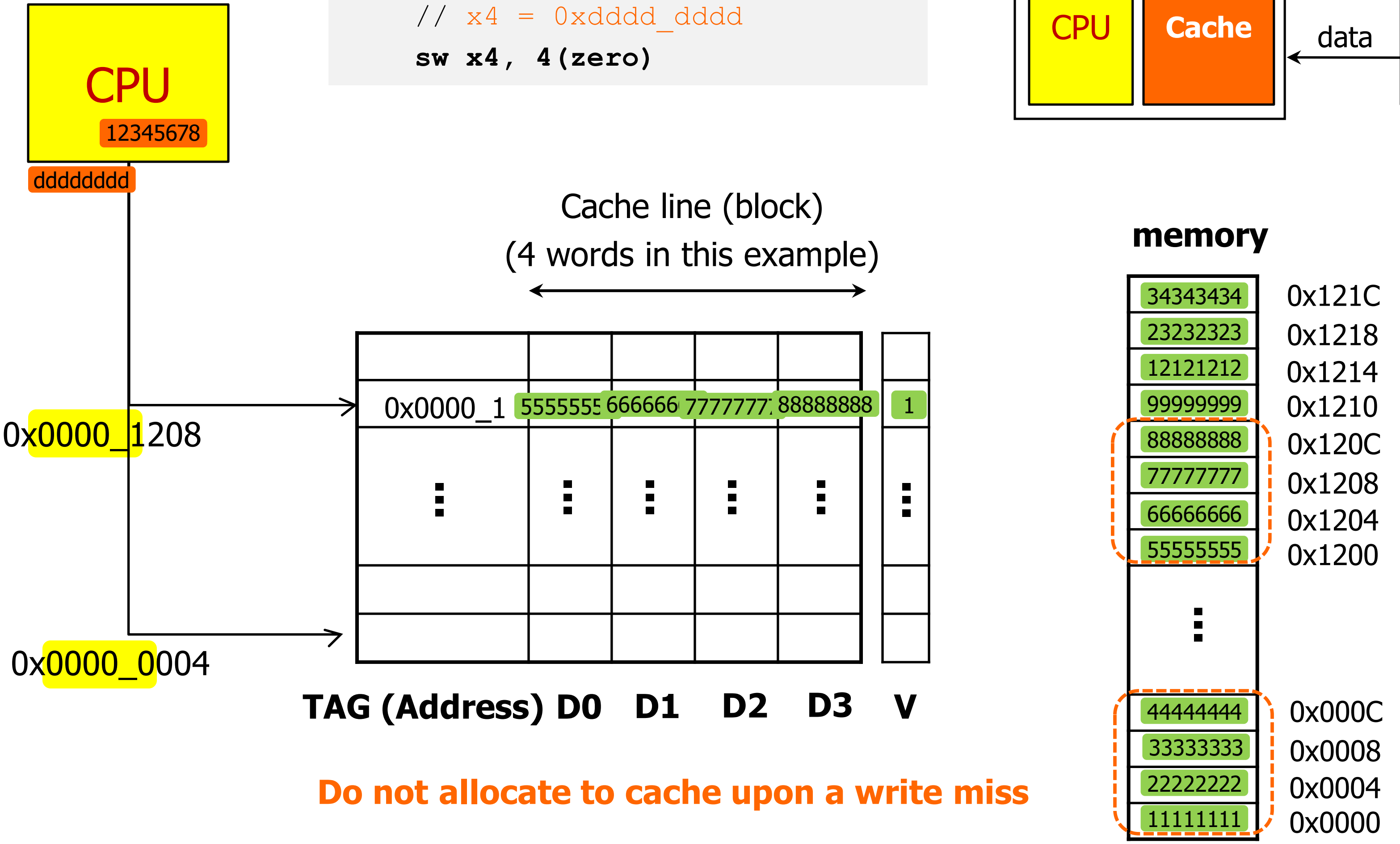
memory	
34343434	0x121C
23232323	0x1218
12121212	0x1214
99999999	0x1210
88888888	0x120C
77777777	0x1208
66666666	0x1204
55555555	0x1200
⋮	
44444444	0x000C
33333333	0x0008
22222222	0x0004
11111111	0x0000

Allocate first upon a write miss

Write No-allocate & Write-through

```
// x3 = 0x1234_5678
sw x3, 8(x2) // x2 = 0x1200
// x4 = 0xdddd_dddd
sw x4, 4(zero)
```



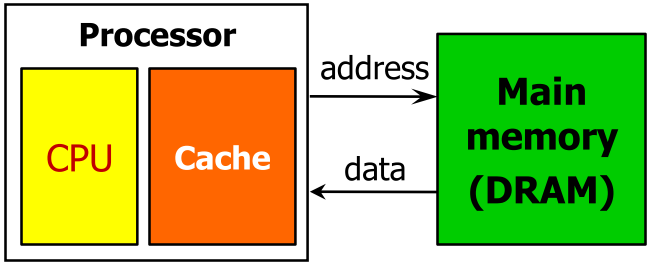


TAG (Address)	D0	D1	D2	D3	V
0x0000_1	5555555	666666	7777777	88888888	1
⋮	⋮	⋮	⋮	⋮	⋮

memory	
34343434	0x121C
23232323	0x1218
12121212	0x1214
99999999	0x1210
88888888	0x120C
77777777	0x1208
66666666	0x1204
55555555	0x1200
⋮	
44444444	0x000C
33333333	0x0008
22222222	0x0004
11111111	0x0000

Do not allocate to cache upon a write miss

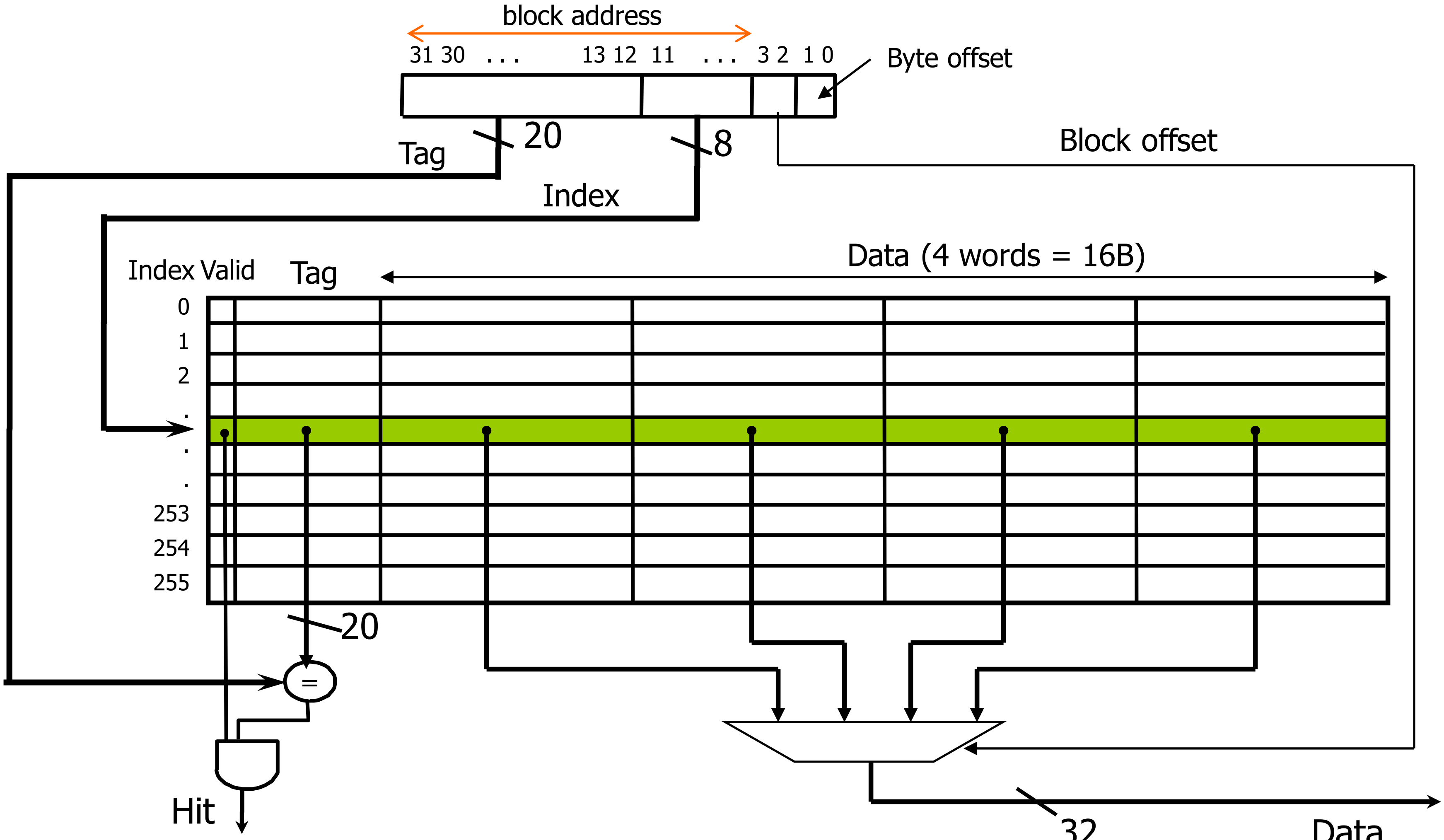
Hits vs. Misses



- Read hits
  - This is what we want!
- Read misses
  - Stall the CPU, fetch the corresponding block from memory, deliver to cache and CPU, and continue to run CPU
- Write hits
  - Write-through cache: write data to both cache and memory
  - Write-back cache: write the data **only** into the cache and set the dirty bit (and write the block to memory later when replaced)
- Write misses
  - Write-allocate with write-back: read the block into the cache, then write the word only to the cache
  - Write no-allocate with write-through: write the word only to the main memory (w/o bringing the block to cache)

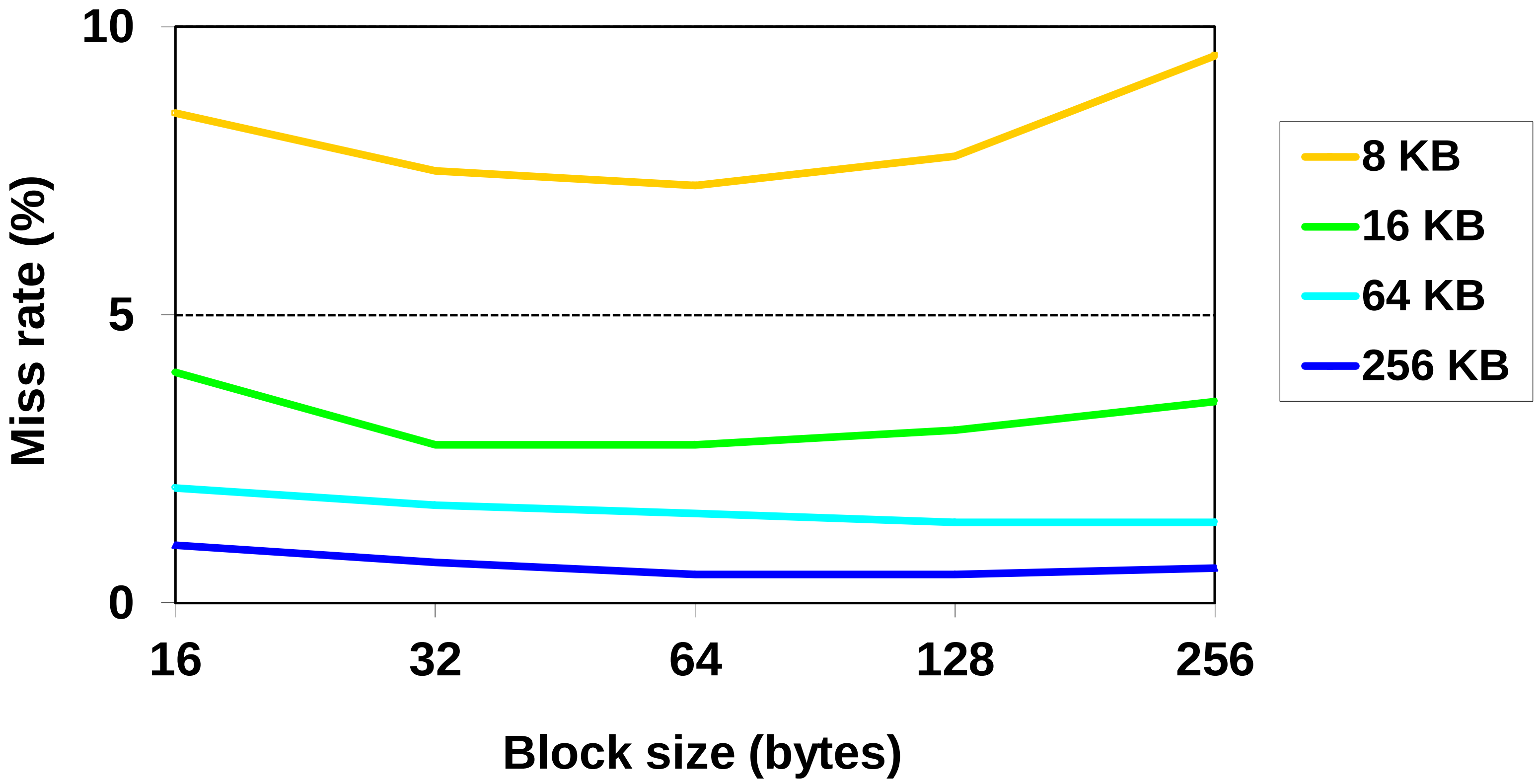
Direct Mapped Cache with 4-word Block

- Cache size = 4KB, 4 words/block



Is this cache structure taking advantage of what kind of locality?

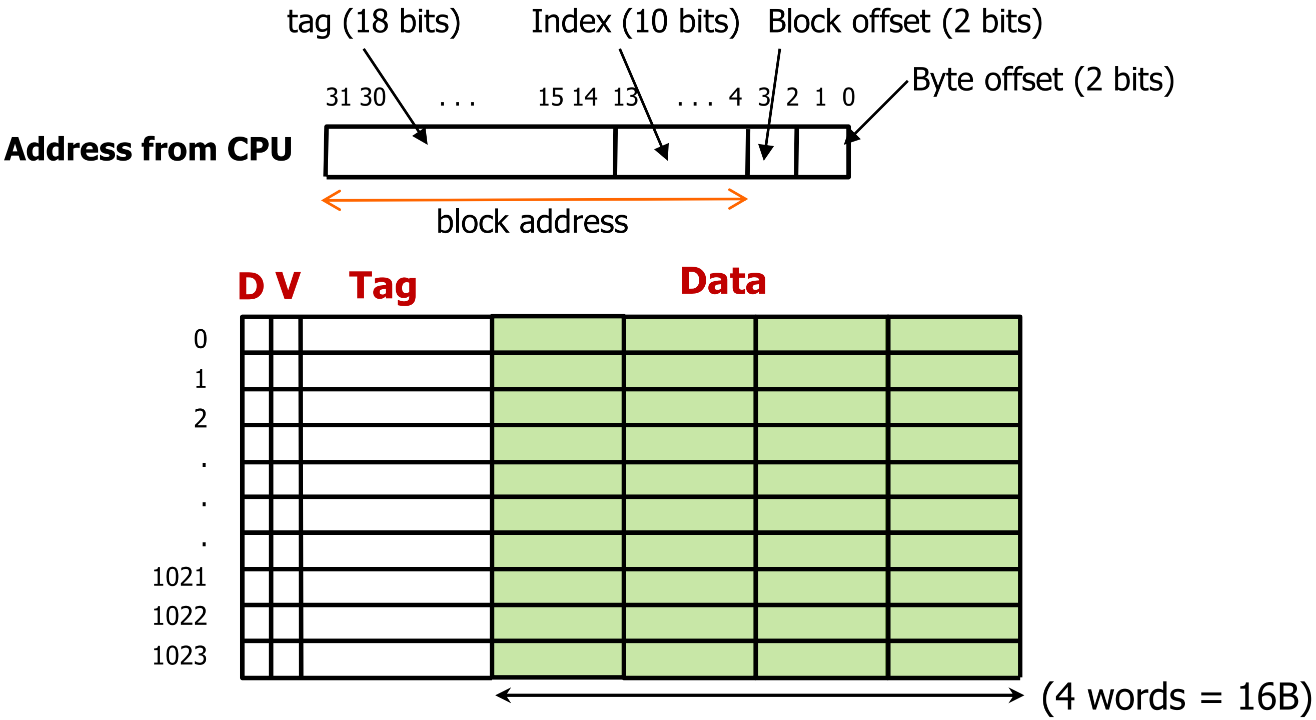
Miss Rate vs Block Size vs Cache Size



- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller
  - Stated alternatively, spatial locality among the words in a word decreases with a very large block; Consequently, the benefits in the miss rate become smaller

Cache Hardware Cost

- How many total bits are required for a direct mapped data cache with 16KB of data and 4-word blocks assuming a 32-bit address?



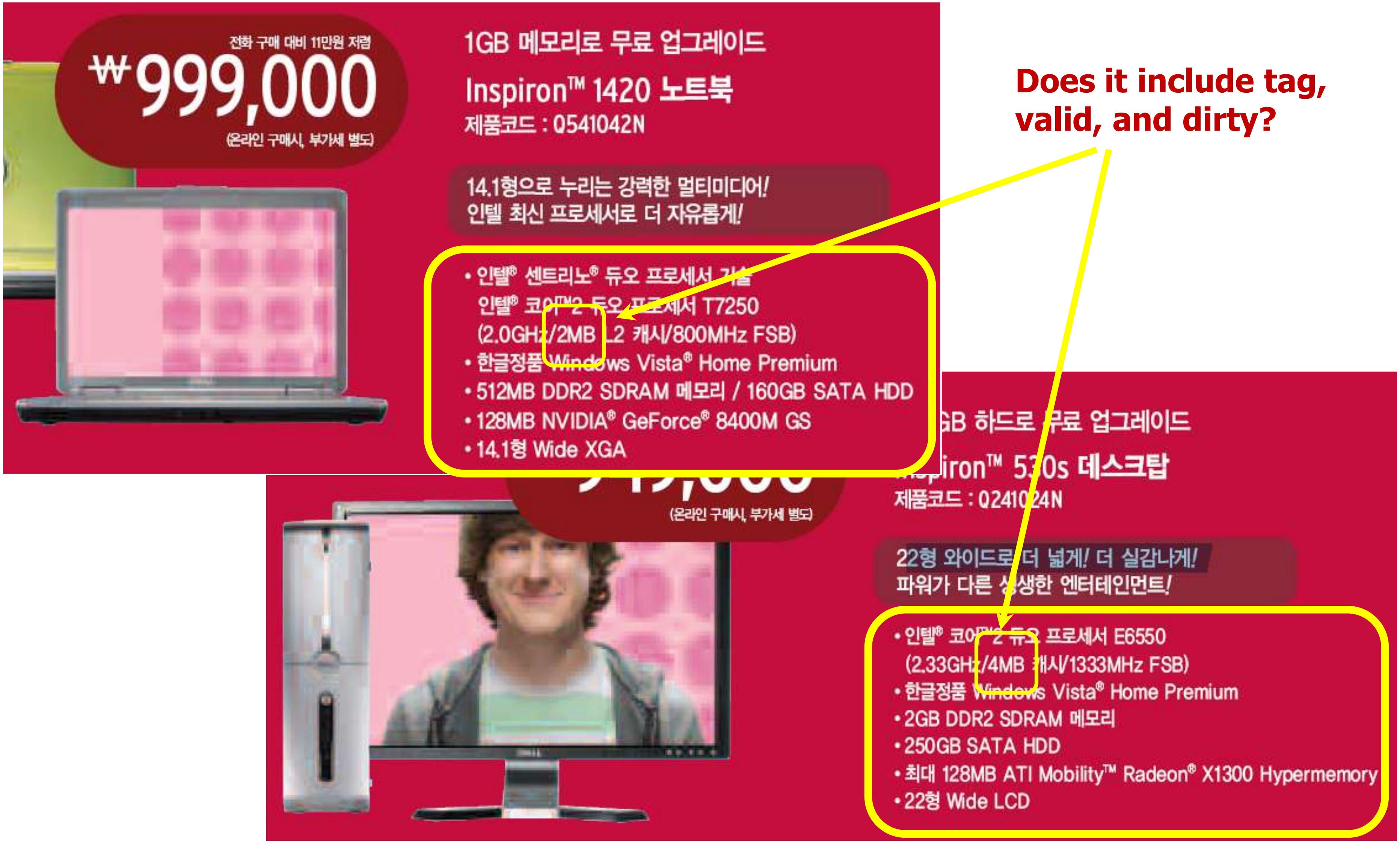
#bits = #blocks x (block size + tag size + valid size + dirty size)

= 1024 x (16B + 18 bits + 1 bit + 1 bit)

= 148 Kbits = 18.5 KB

15.6% larger than the storage for data

Understand Computer Ads?



전화 구매 대비 11만원 저렴

₩999,000

(온라인 구매시, 부가세 별도)

1GB 메모리로 무료 업그레이드

Inspiron™ 1420 노트북

제품코드 : Q541042N

14.1형으로 누리는 강력한 멀티미디어!
인텔 최신 프로세서로 더 자유롭게!

- 인텔® 센트리노® 듀오 프로세서 기술
  인텔® 코어™2 듀오 프로세서 T7250
  (2.0GHz/2MB L2 캐시/800MHz FSB)
- 한글정품 Windows Vista® Home Premium
- 512MB DDR2 SDRAM 메모리 / 160GB SATA HDD
- 128MB NVIDIA® GeForce® 8400M GS
- 14.1형 Wide XGA

(온라인 구매시, 부가세 별도)

GB 하드로 무료 업그레이드

Inspiron™ 530s 데스크탑

제품코드 : Q241024N

22형 와이드로 더 넓게! 더 실감나게!
파워가 다른 생생한 엔터테인먼트!

- 인텔® 코어™2 듀오 프로세서 E6550
  (2.33GHz/4MB 캐시/1333MHz FSB)
- 한글정품 Windows Vista® Home Premium
- 2GB DDR2 SDRAM 메모리
- 250GB SATA HDD
- 최대 128MB ATI Mobility™ Radeon® X1300 Hypermemory
- 22형 Wide LCD

Does it include tag, valid, and dirty?

Cache Hardware Cost

- The number of bits in a cache includes both the storage for data and for the tags
  - For a direct mapped cache with 2^n blocks, *n* bits are used for the index
  - For a block size of 2^m words (2^{m+2} bytes), *m* bits are used to address the word within the block and 2 bits are used to address the byte within the word

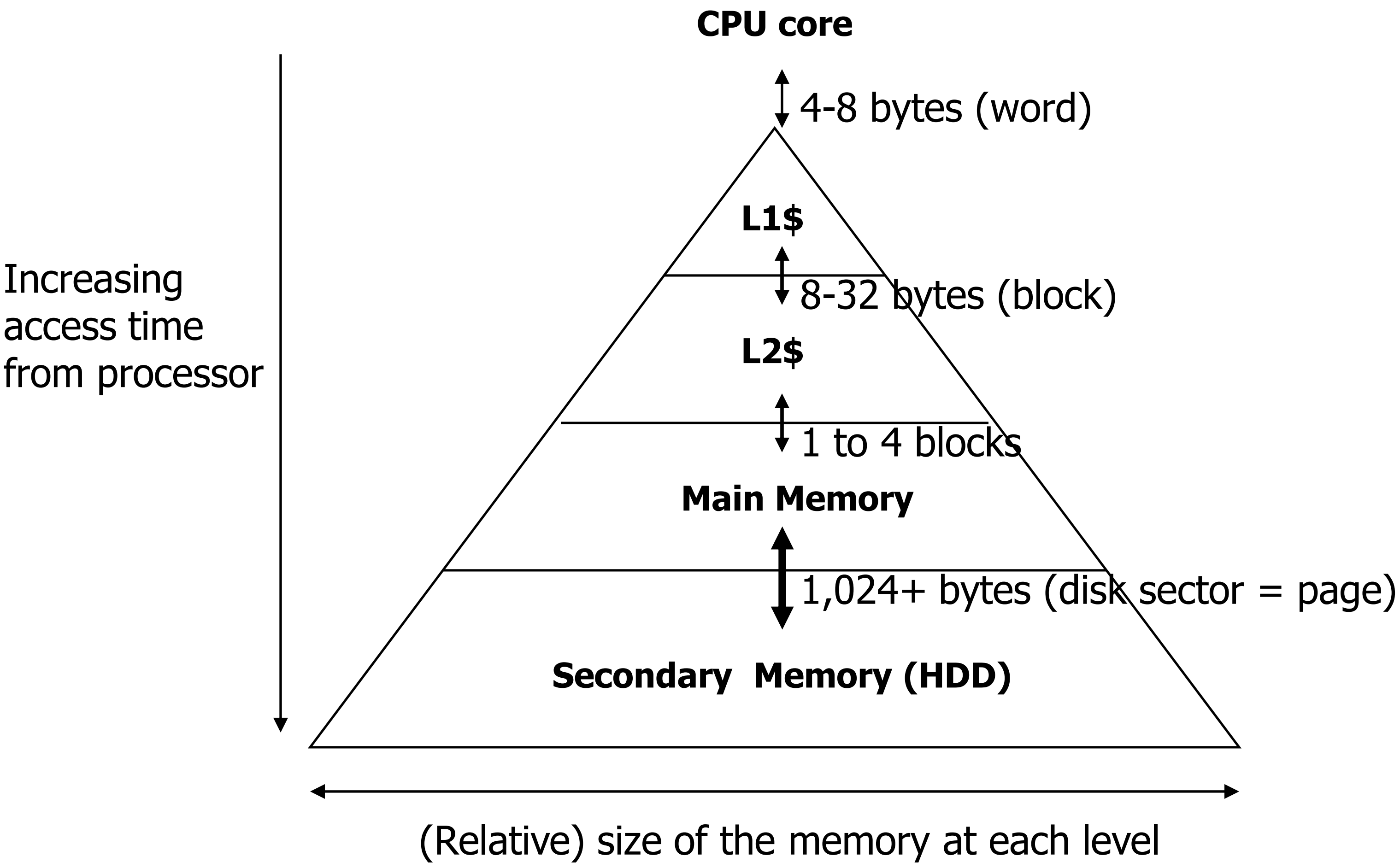
- The total number of bits in a direct-mapped cache is

  I$: = 2^n x (block size + tag size + valid size)

  D$: = 2^n x (block size + tag size + valid size + dirty size)

Backup

Characteristics of Memory Hierarchy


CPU core
4-8 bytes (word)
L1$
8-32 bytes (block)
L2$
1 to 4 blocks
Main Memory
1,024+ bytes (disk sector = page)
Secondary Memory (HDD)
Increasing access time from processor
(Relative) size of the memory at each level