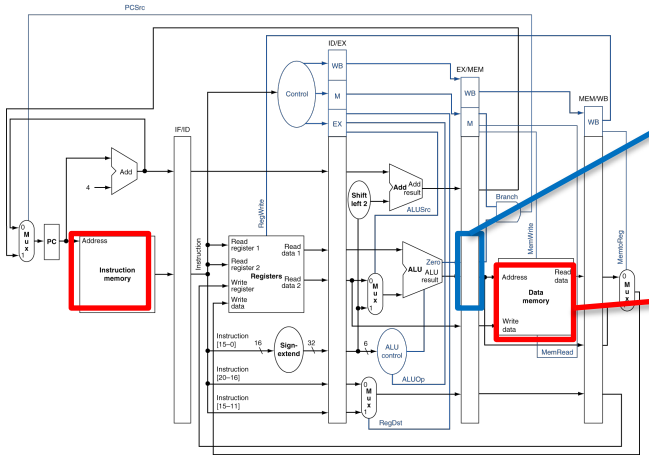
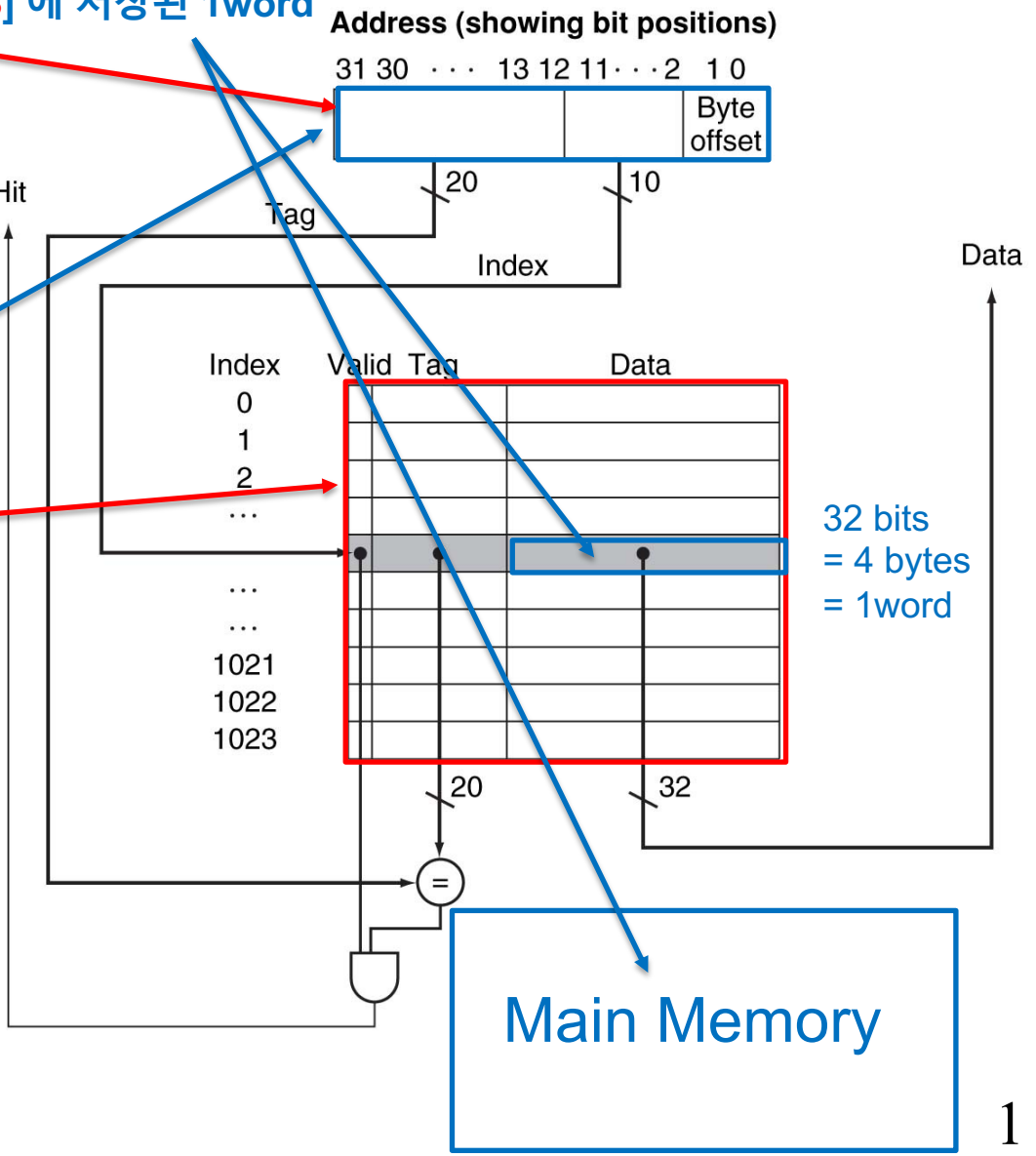
# Direct Mapped Cache (byte addressing, 1 word/block)

[메모리 주소 0x10000010~0x10000013] 에 저장된 1word



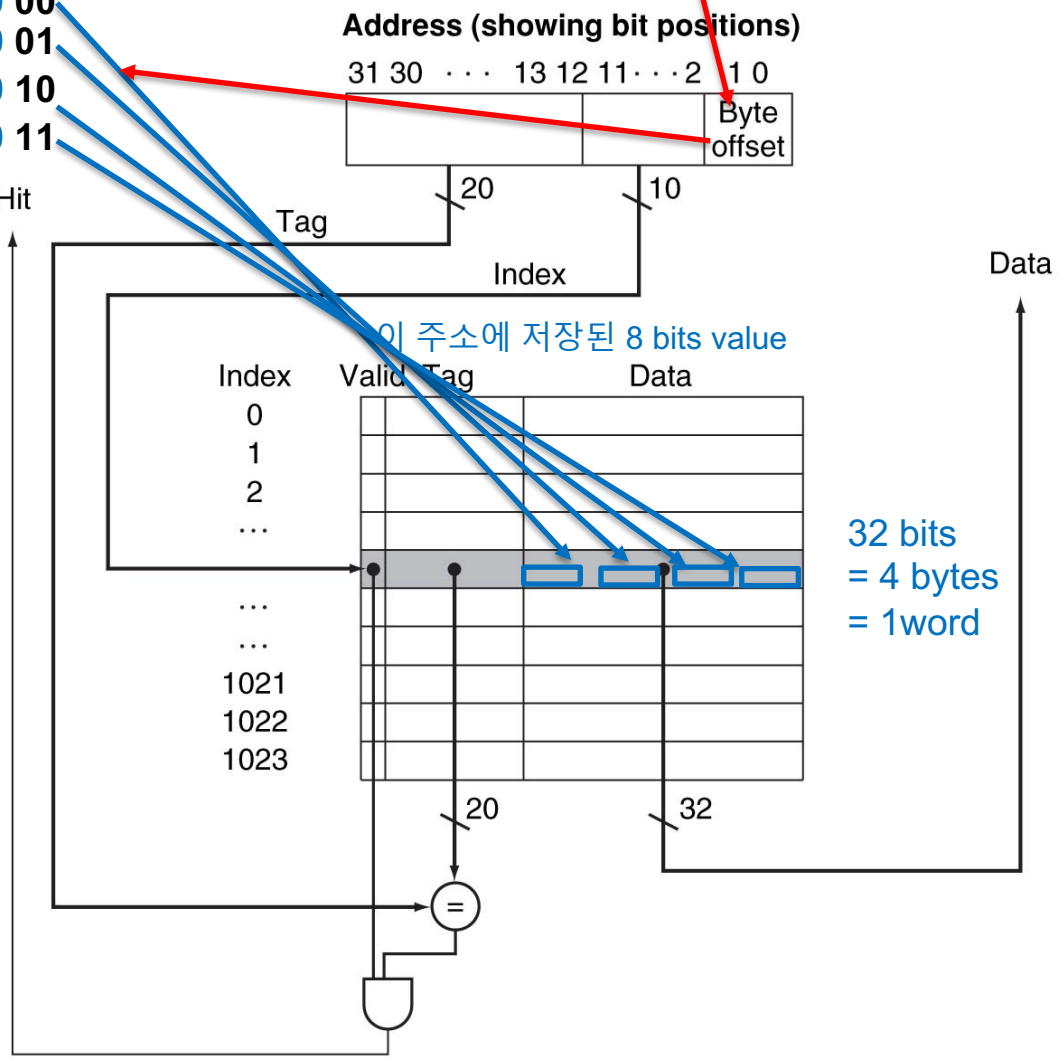
lw $4, 0($3)

$3 : 0x10000010



# Direct Mapped Cache (byte addressing, 1 word/block)

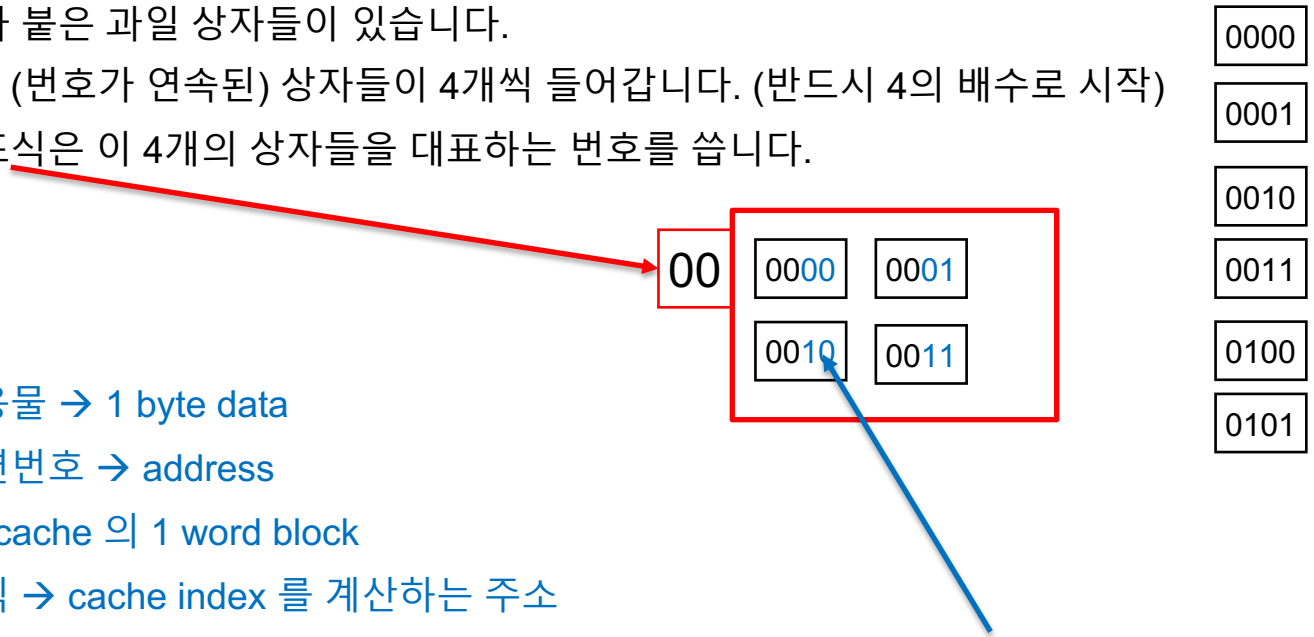
0001 0000 0000 0000 0000 0000 0001 00 00
0001 0000 0000 0000 0000 0000 0001 00 01
0001 0000 0000 0000 0000 0000 0001 00 10
0001 0000 0000 0000 0000 0000 0001 00 11
[메모리 주소 0x10000010~0x10000013]




*What kind of locality are we taking advantage of?*

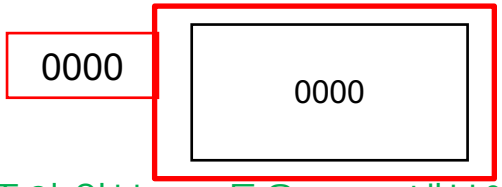
# byte offset

- 일련 번호가 붙은 과일 상자들이 있습니다.
- 바구니에는 (번호가 연속된) 상자들이 4개씩 들어갑니다. (반드시 4의 배수로 시작)
- 바구니의 표식은 이 4개의 상자들을 대표하는 번호를 씁니다.



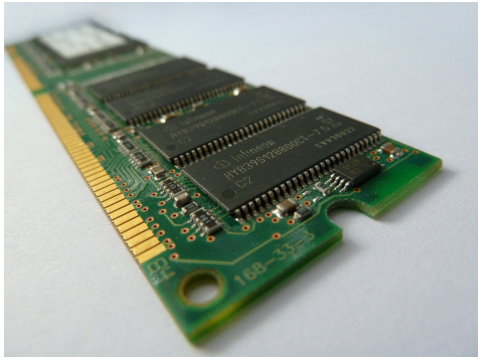

- 상자의 내용물 → 1 byte data
- 상자의 일련번호 → address
- 바구니 → cache 의 1 word block
- 바구니 표식 → cache index 를 계산하는 주소
- 바구니 안에서 4개의 상자들을 구분하기 위해서 사용하는 bits 가 byte offset

word addresing 을 하고 block 의 크기가 1 word 일 때는 1 block 에 저장된 data 의 주소가 1개이니까 block 내부에서 주소를 구분하는 offset 부분이 필요가 없었다.

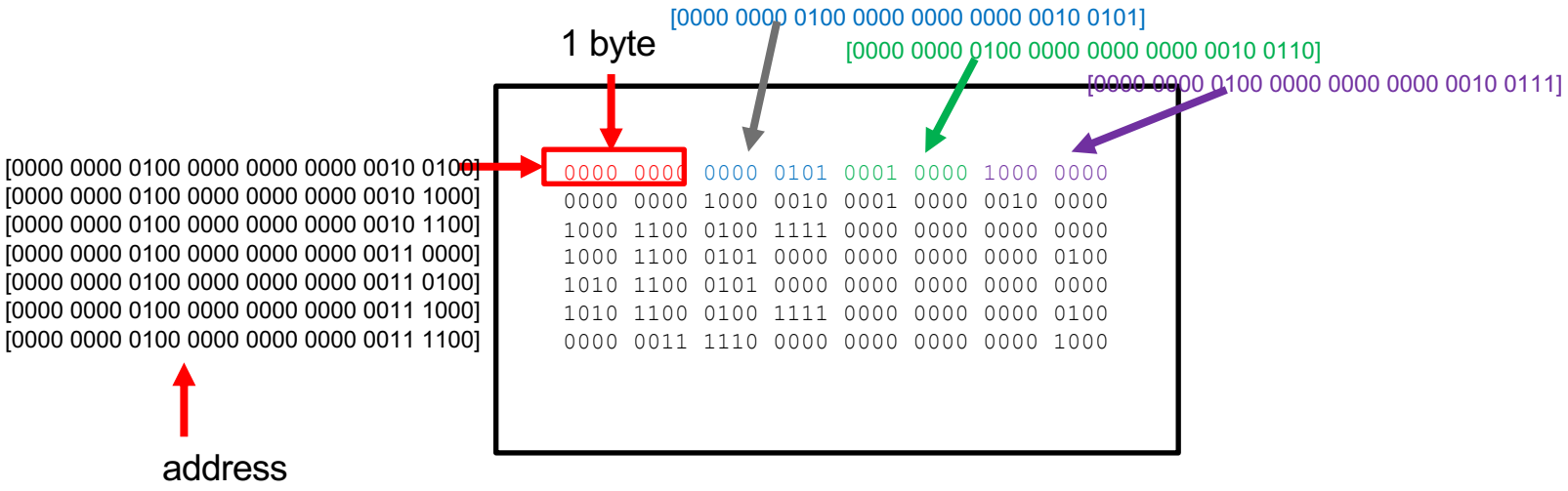



실제로는 block 내에 저장된 data 의 주소가 여러 개이므로 주소 중의 일부 bits 들은 block 내부에서 주소를 구분하는 offset 으로 사용된다.

# Memory

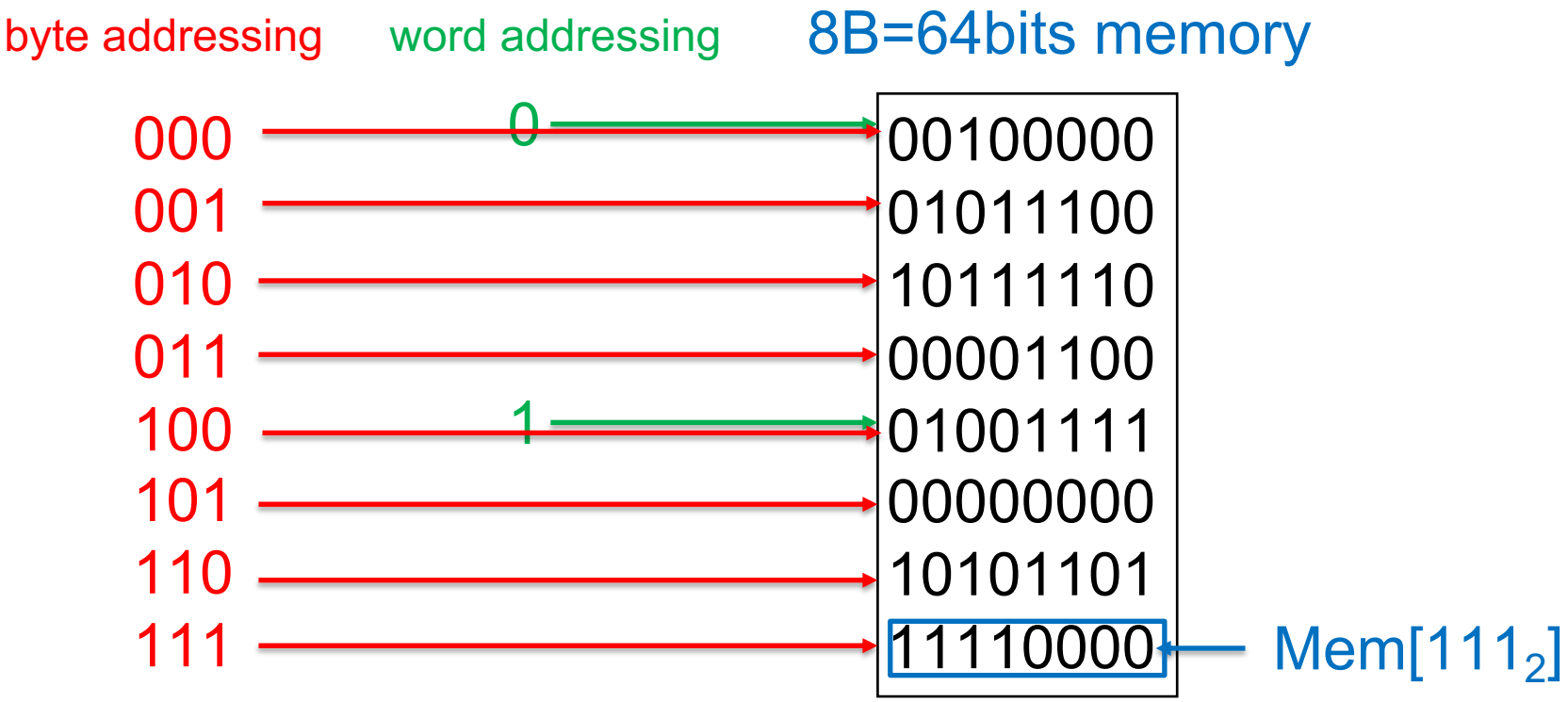


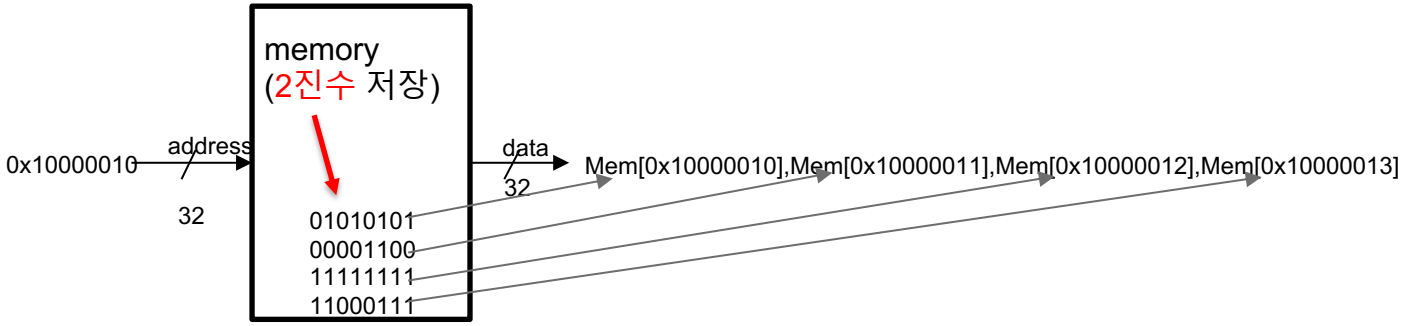
byte addressing



# What is memory address?

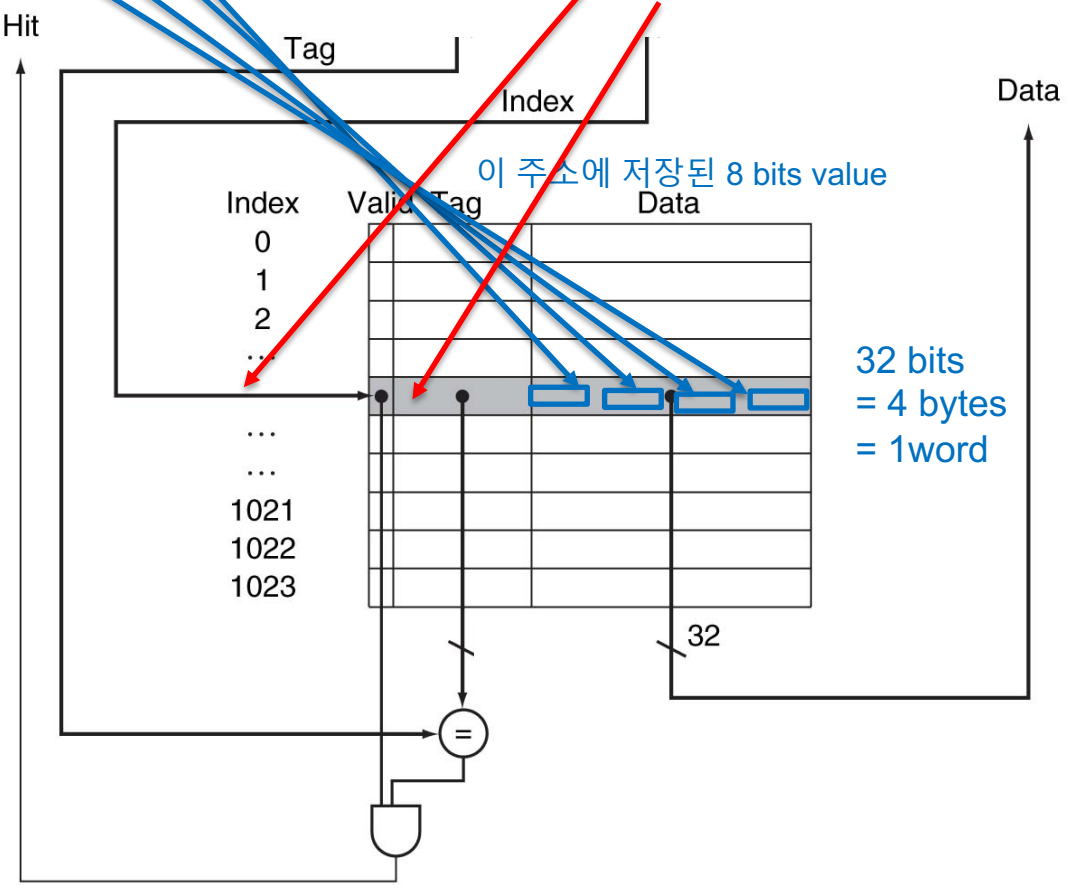
- 메모리에 저장된 것은 2진수 형태의 data 들 (bit 들의 모임) 이고,
- 이 수많은 bits 중 특정한 bits 를 지칭하기 위한 일련번호가 주소이며
- 다시 말하지만, 주소(index)는 메모리(cache)에 저장된 내용이 아닙니다!!!




memory
(2진수 저장)
0x10000010
address
32
01010101
00001100
11111111
11000111
data
32
Mem[0x10000010],Mem[0x10000011],Mem[0x10000012],Mem[0x10000013]

# Direct Mapped Cache (byte addressing, 1 word/block)

0001 0000 0000 0000 0000 0000 0001 0000
0001 0000 0000 0000 0000 0000 0001 0001
0001 0000 0000 0000 0000 0000 0001 0010
0001 0000 0000 0000 0000 0000 0001 0011
[메모리 주소 0x10000010~0x10000013]



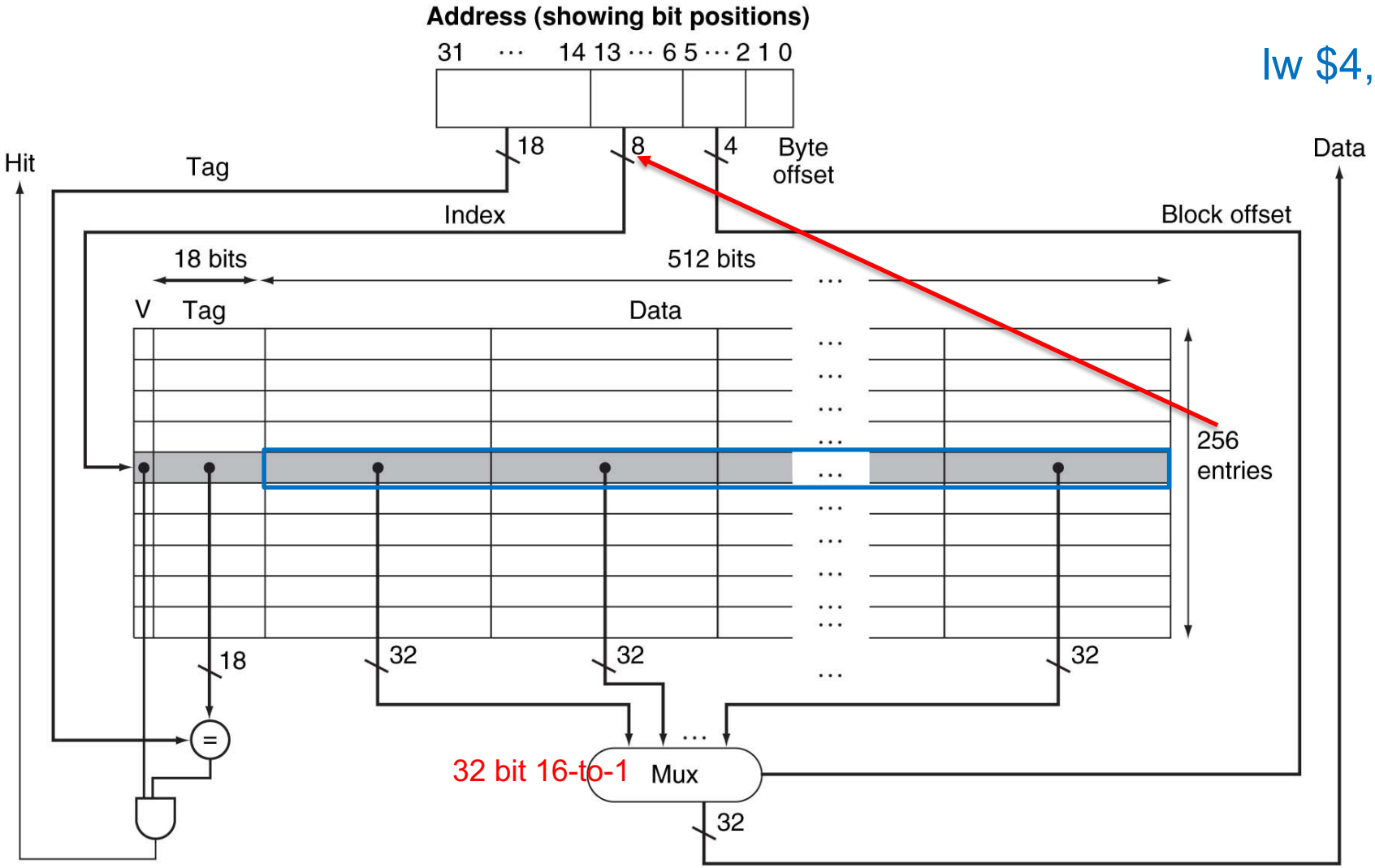
# Direct Mapped Cache (byte addressing, 16 words/block)

16 words
= 64 bytes
= $2^6$ bytes

- **Taking advantage of spatial locality:**

$3 : 0x1000 0110

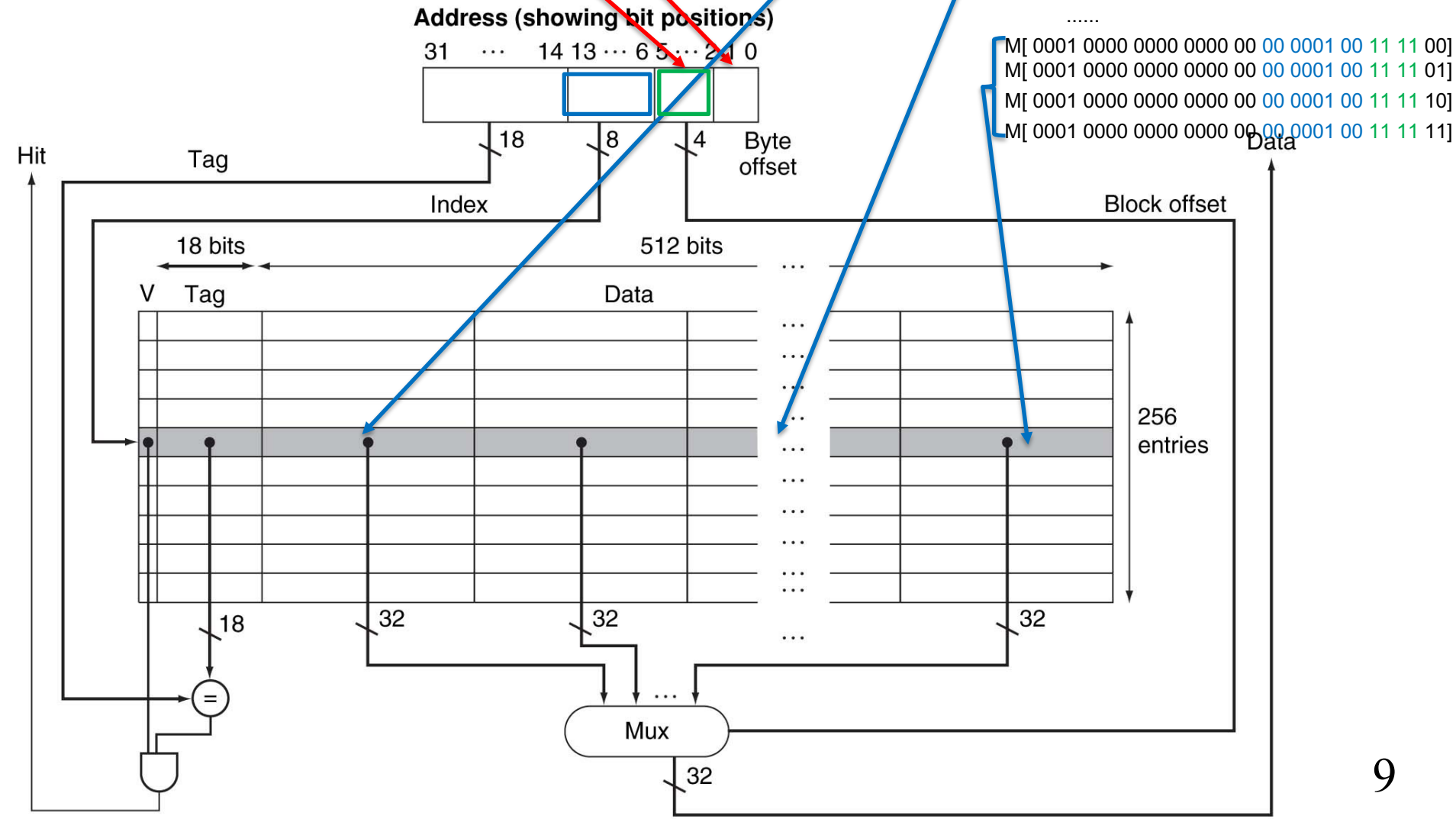
lw $4, 0($3)



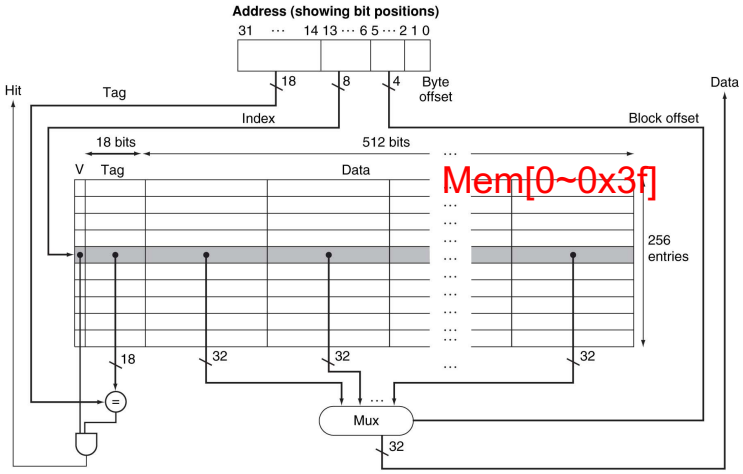
# Direct Mapped Cache (byte addressing, 16 words/block)

16 words
= 64 bytes
= $2^6$ bytes

M[0001 0000 0000 0000 00 00 0001 00 00 00 00]
M[0001 0000 0000 0000 00 00 0001 00 00 00 01]
M[ 0001 0000 0000 0000 00 00 0001 00 00 00 10]
M[ 0001 0000 0000 0000 00 00 0001 00 00 00 11]
.....
M[ 0001 0000 0000 0000 00 00 0001 00 01 00 00]
M[ 0001 0000 0000 0000 00 00 0001 00 01 00 01]
M[ 0001 0000 0000 0000 00 00 0001 00 01 00 10]
M[ 0001 0000 0000 0000 00 00 0001 00 01 00 11]
.....
M[ 0001 0000 0000 0000 00 00 0001 00 11 11 00]
M[ 0001 0000 0000 0000 00 00 0001 00 11 11 01]
M[ 0001 0000 0000 0000 00 00 0001 00 11 11 10]
M[ 0001 0000 0000 0000 00 00 0001 00 11 11 11]



# Direct Mapped Cache (byte addressing, 1 words/block) 과 비교

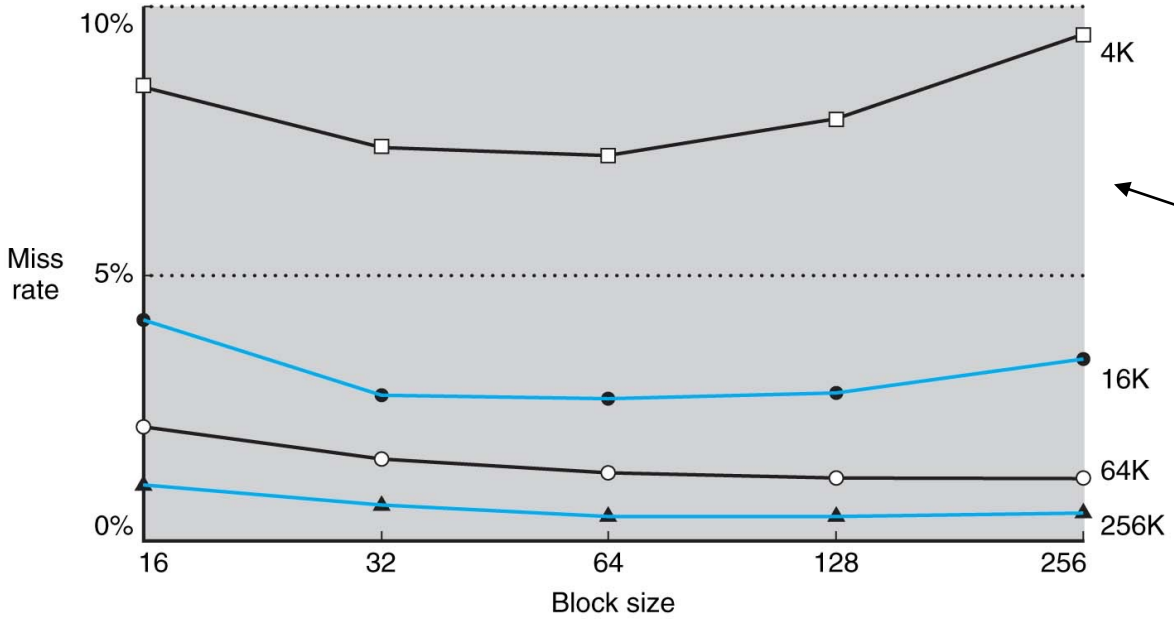



| address | address in binary | miss/hit 1word/block | miss/hit 16word/block |
|---|---|---|---|
| 0x1000 0100 | 0001 0000 0000 0000 0000 0001 0000 0000 | miss | miss |
| 0x1000 0110 | 0001 0000 0000 0000 0000 0001 0001 0000 | miss | hit |
| 0x1000 0120 | 0001 0000 0000 0000 0000 0001 0010 0000 | miss | hit |
| 0x1000 0130 | 0001 0000 0000 0000 0000 0001 0011 0000 | miss | hit |
| 0x1000 0140 | 0001 0000 0000 0000 0000 0001 0100 0000 | miss | miss |

block 크기를 증가하면 spatial locality 를 활용하여 compulsory miss 를 없앨 수 있다.

# miss rate vs. block size

- **Increasing the block size tends to decrease miss rate:**
-

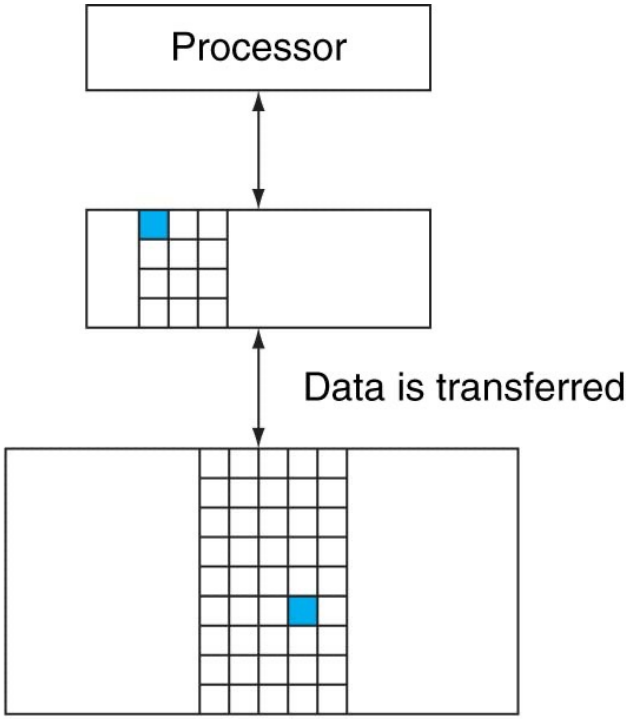



Increased cache conflict

- **Use split caches because there is more spatial locality in code:**

| Program | Block size in words | Instruction miss rate | Data miss rate | Effective combined miss rate |
|---|---|---|---|---|
| gcc | 1 | 6.1% | 2.1% | 5.4% |
| | 4 | 2.0% | 1.7% | 1.9% |
| spice | 1 | 1.2% | 1.3% | 1.2% |
| | 4 | 0.3% | 0.6% | 0.4% |

# Hits vs. Misses




- **Read hits**
  - **this is what we want!**
- **Read misses**
  - **stall the CPU, fetch block from memory, deliver to cache, restart**
- **Write hits: -> cache coherence problem**
  - **can replace data in cache and memory (write-through)**
  - **write the data only into the cache (write-back the cache later)**
- **Write misses:**
  - **read the entire block into the cache, then write the word**

# Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

# Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first

# Performance

$$\text{CPU Time} = \text{Instruction Count} \cdot \text{CPI} \cdot \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \cdot \text{CPI}}{\text{Clock Rate}}$$

- **Simplified model:**

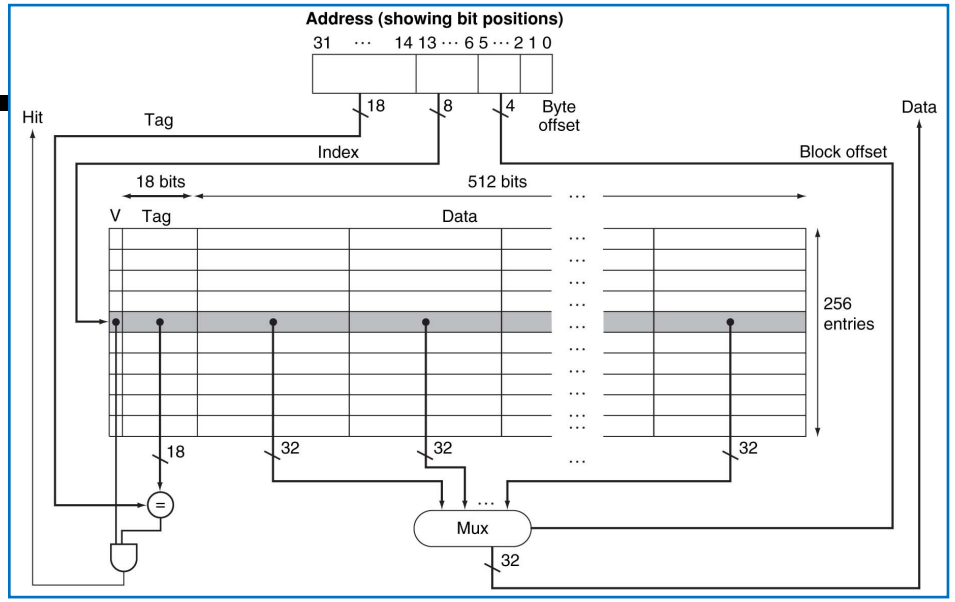
  **execution time = (execution cycles + stall cycles) × cycle time**

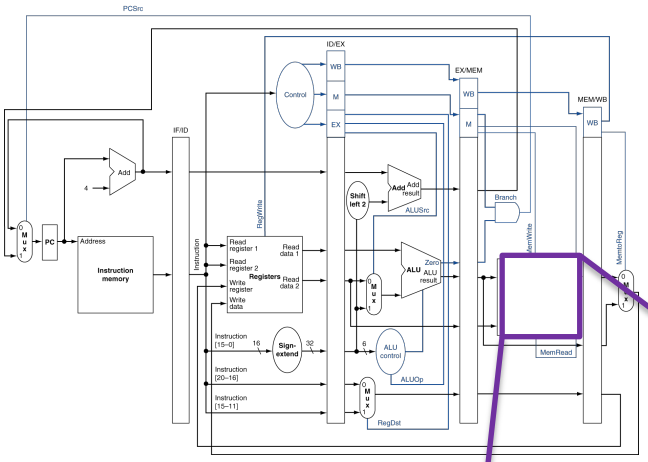
  **stall cycles = # of instructions × miss ratio × miss penalty**

- **Two ways of improving performance:**
  - **decreasing the miss ratio**
  - **decreasing the miss penalty**

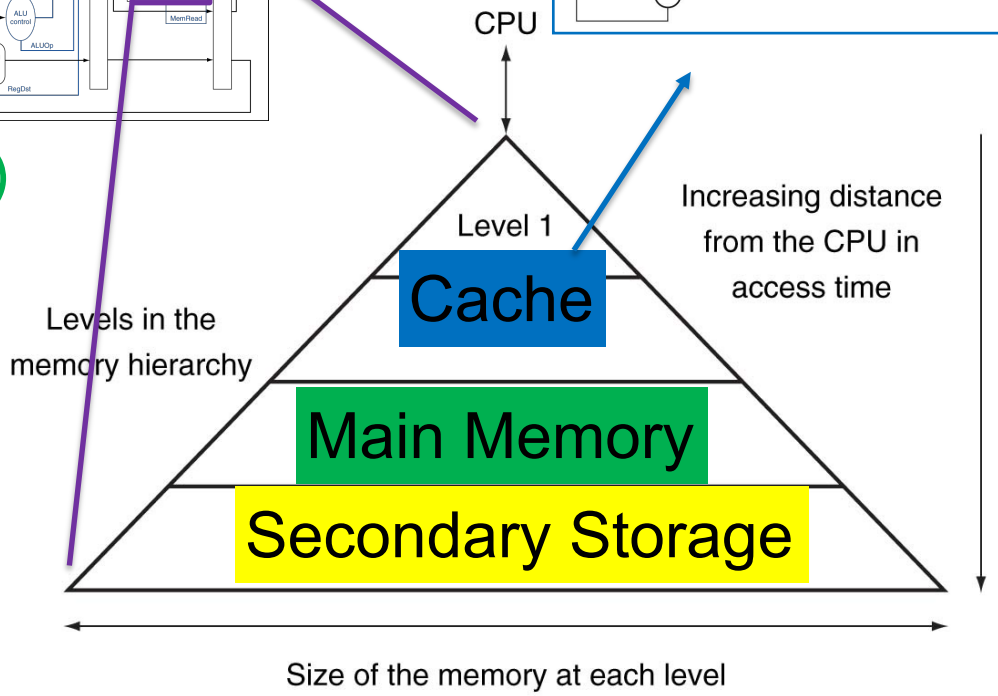
***What happens if we increase block size?***

# Memory Hierarchy



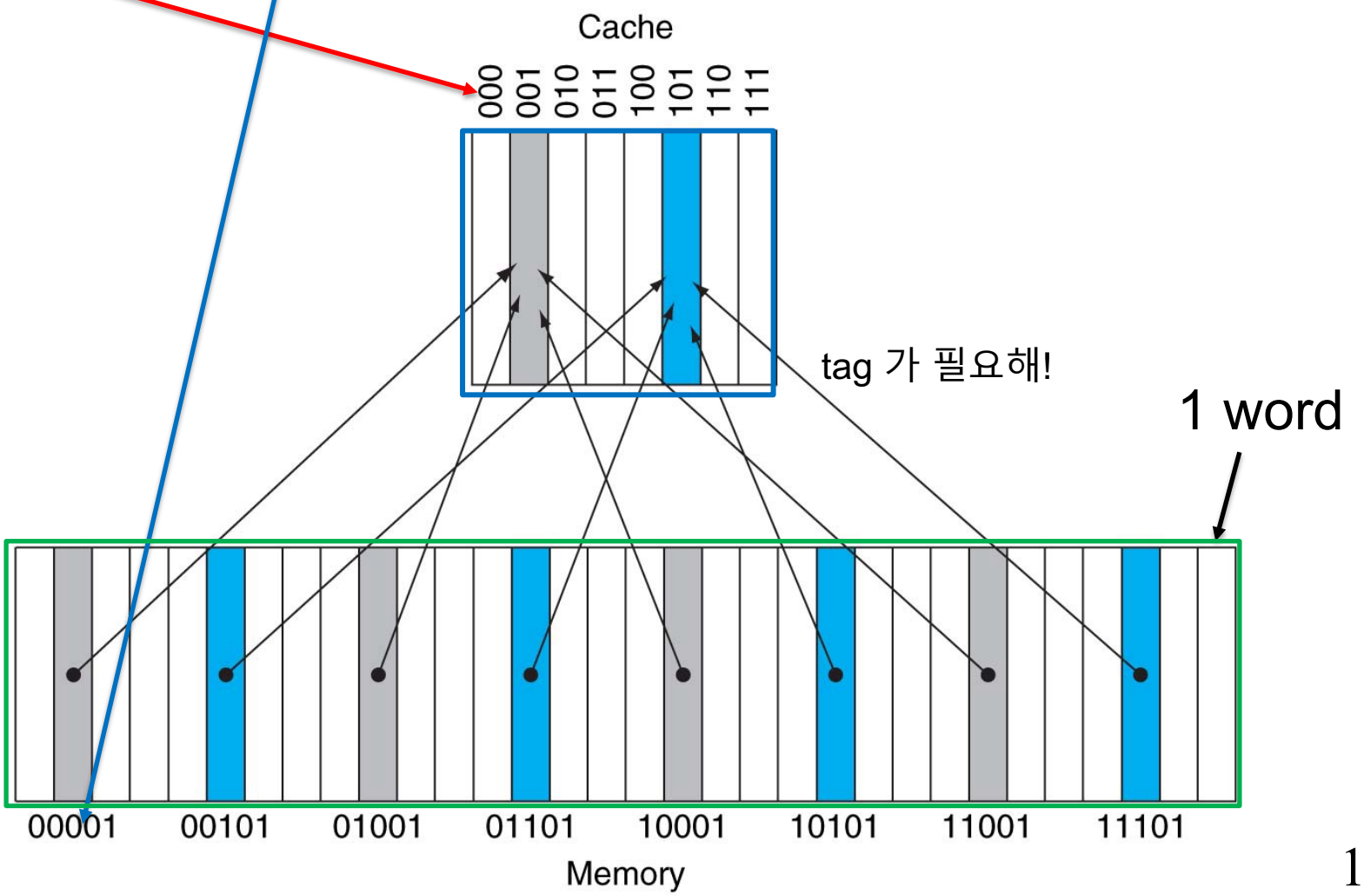



lw $4, 0($3)



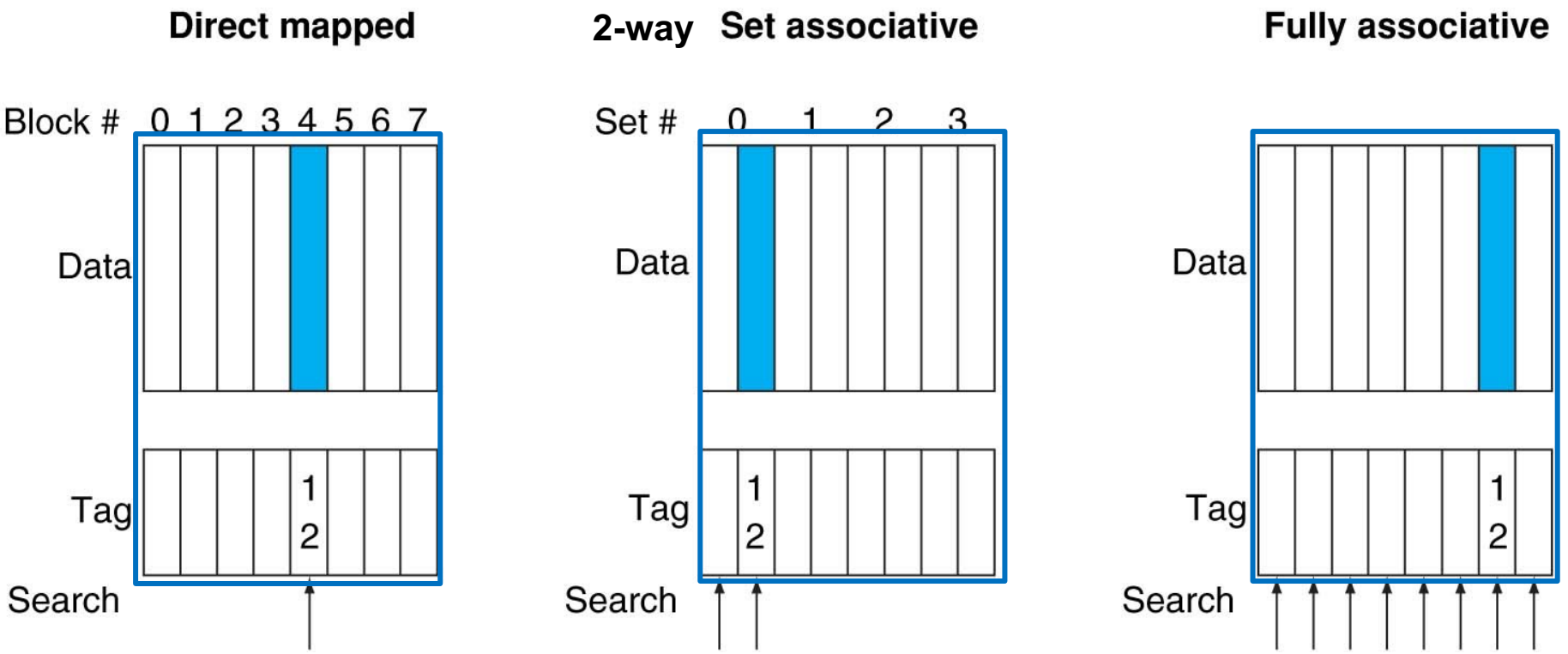
# Direct Mapped Cache (word addressing, 1 word/block)

**cache index** = memory address % **the number of blocks in the cache**



# Set Associative Cache

**1 word/ block, word addressing**




main memory

main memory

main memory

# Cache behavior example

- **Access sequence 0,8,0,6,8**

- **Direct mapped cache**

| Block address | Cache block |
|---|---|
| 0 | (0 modulo 4) = 0 |
| 6 | (6 modulo 4) = 2 |
| 8 | (8 modulo 4) = 0 |

| Address of memory block accessed | Hit or miss | Contents of cache blocks after reference | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| 0 | miss | Memory[0] | | | |
| 8 | miss | Memory[8] | | | |
| 0 | miss | Memory[0] | | | |
| 6 | miss | Memory[0] | | Memory[6] | |
| 8 | miss | Memory[8] | | Memory[6] | |

시간

cache (valid bit, tag 는 보이지 않음)

- **2-way set associative cache**

| Block address | Cache set |
|---|---|
| 0 | (0 modulo 2) = 0 |
| 6 | (6 modulo 2) = 0 |
| 8 | (8 modulo 2) = 0 |

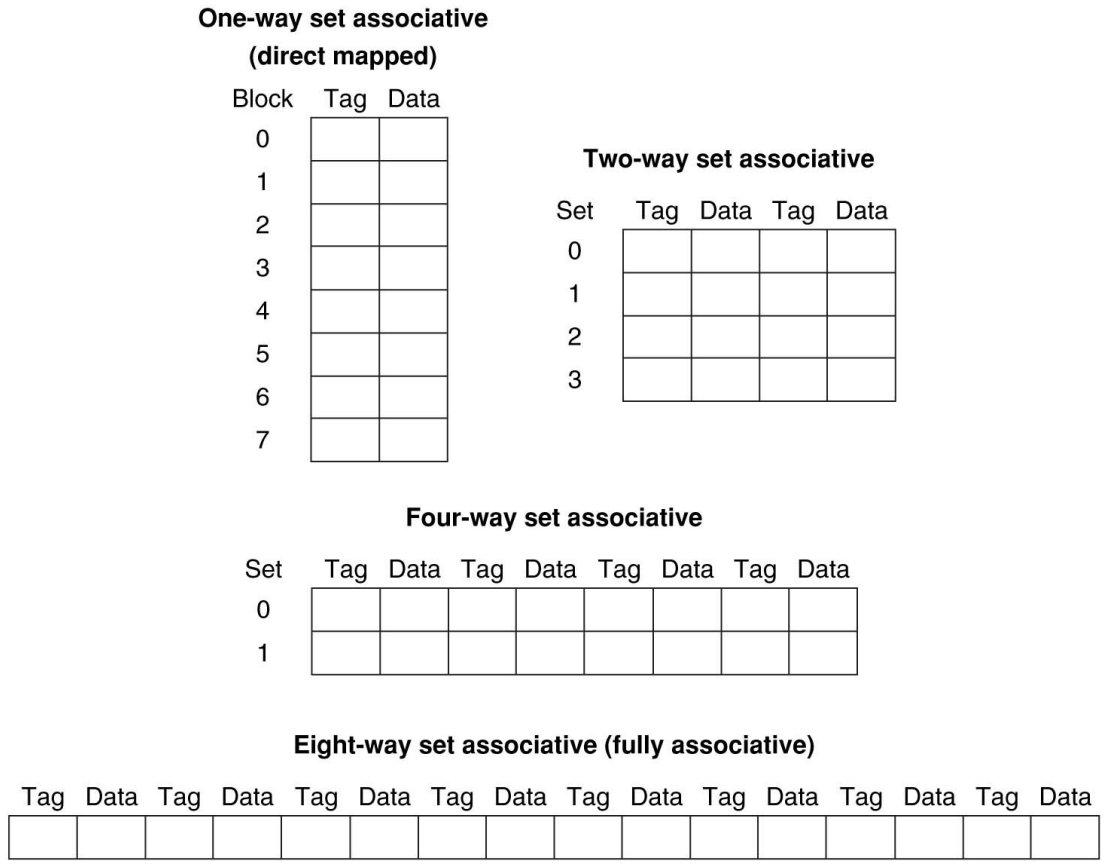
| Address of memory block accessed | Hit or miss | Contents of cache blocks after reference | | | |
|---|---|---|---|---|---|
| | | Set 0 | Set 0 | Set 1 | Set 1 |
| 0 | miss | Memory[0] | | | |
| 8 | miss | Memory[0] | Memory[8] | | |
| 0 | hit | Memory[0] | Memory[8] | | |
| 6 | miss | Memory[0] | Memory[6] | | |
| 8 | miss | Memory[8] | Memory[6] | | |

- **Fully associative cache**

| Address of memory block accessed | Hit or miss | Contents of cache blocks after reference | | | |
|---|---|---|---|---|---|
| | | Block 0 | Block 1 | Block 2 | Block 3 |
| 0 | miss | Memory[0] | | | |
| 8 | miss | Memory[0] | Memory[8] | | |
| 0 | hit | Memory[0] | Memory[8] | | |
| 6 | miss | Memory[0] | Memory[8] | Memory[6] | |
| 8 | hit | Memory[0] | Memory[8] | Memory[6] | |

| Index | V | Tag | Data |
|---|---|---|---|
| 000 | N | | |
| 001 | N | | |
| 010 | Y | $11_{two}$ | Memory ($11010_{two}$) |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | $10_{two}$ | Memory ($10110_{two}$) |
| 111 | N | | |

# Set associativity




*least recently used (LRU) replacement strategy*

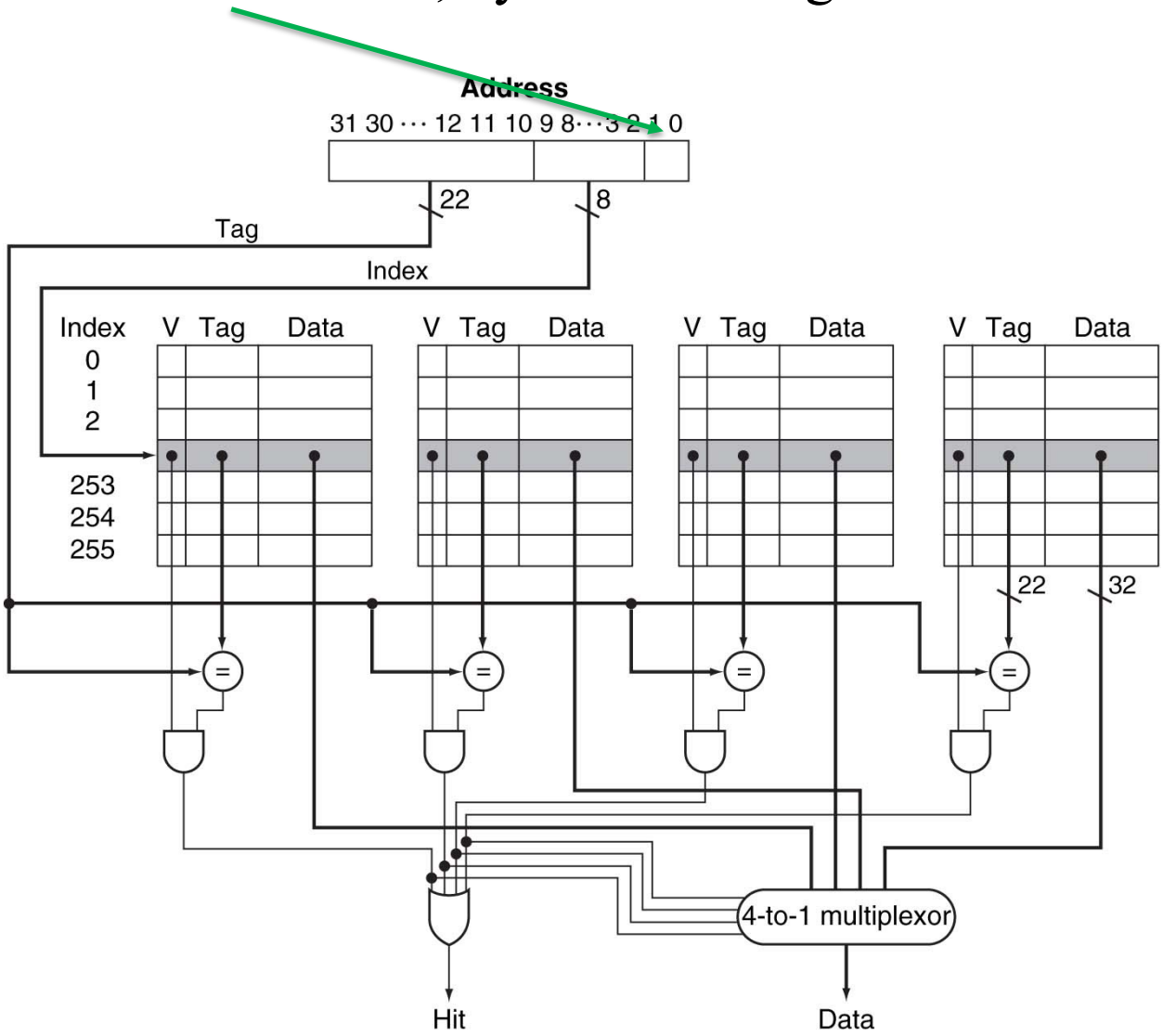
# Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity


**FIGURE 5.16 The three portions of an address in a set-associative or direct-mapped cache.** The index is used to select the set, then the tag is used to choose the block by comparison with the blocks in the selected set. The block offset is the address of the desired data within the block. 

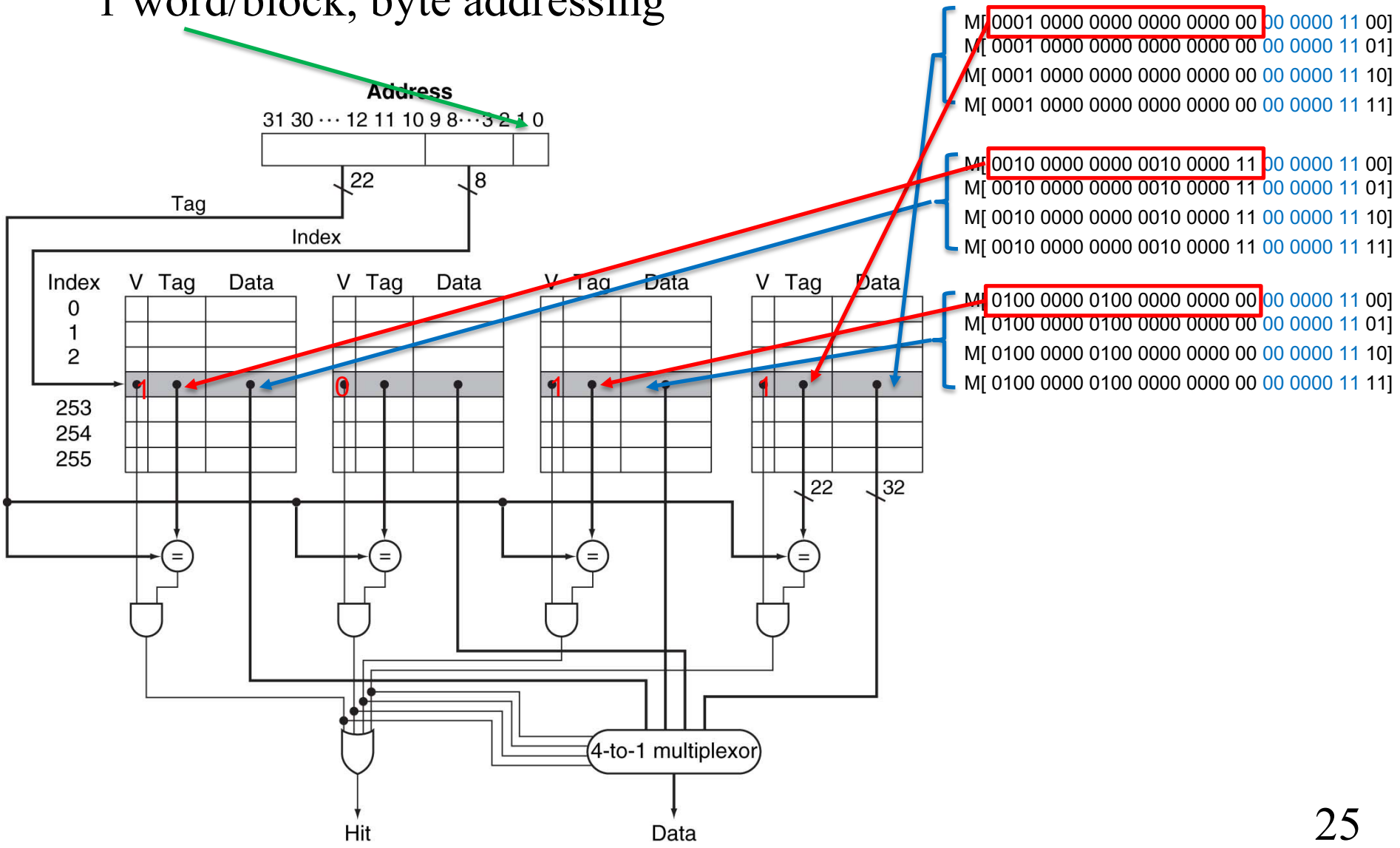
# An implementation of 4-way set associative cache

1 word/block, byte addressing



# An implementation of 4-way set associative cache

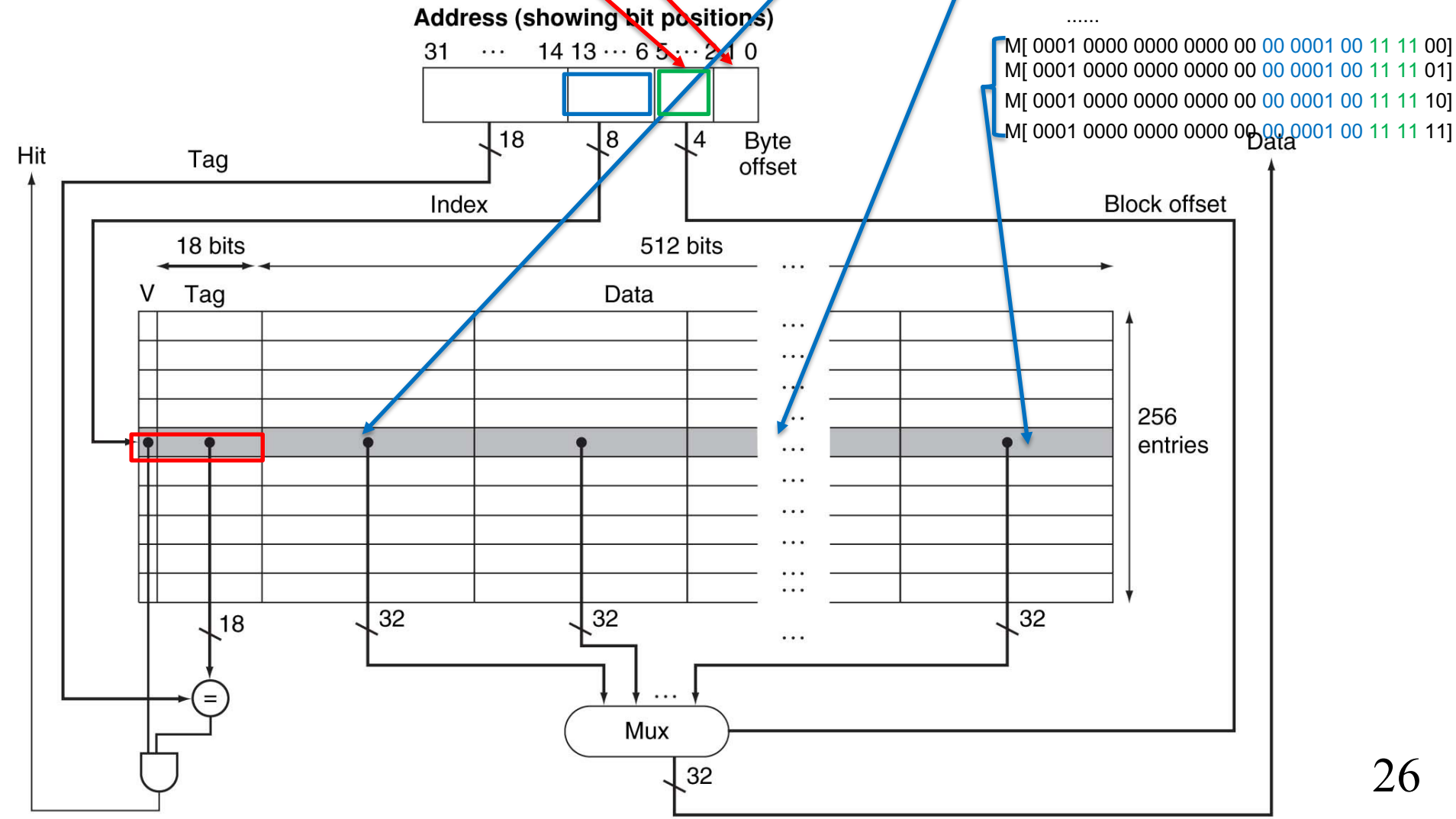
1 word/block, byte addressing



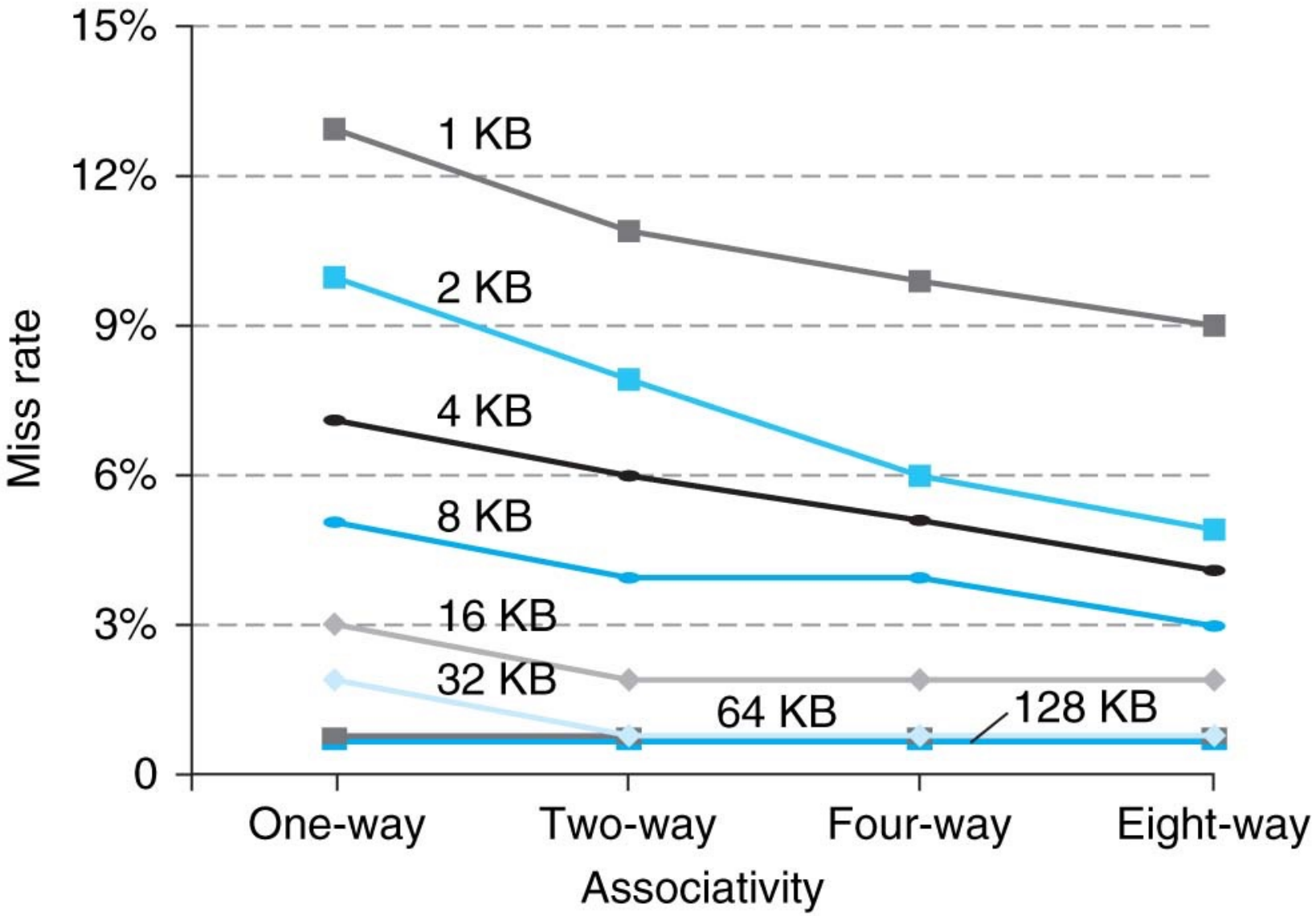
# Direct Mapped Cache
# (byte addressing, 16 words/block)

16 words
= 64 bytes
= $2^6$ bytes

M[0001 0000 0000 0000 00 00 0001 00 00 00 00]
M[0001 0000 0000 0000 00 00 0001 00 00 00 01]
M[ 0001 0000 0000 0000 00 00 0001 00 00 00 10]
M[ 0001 0000 0000 0000 00 00 0001 00 00 00 11]
.....
M[ 0001 0000 0000 0000 00 00 0001 00 01 00 00]
M[ 0001 0000 0000 0000 00 00 0001 00 01 00 01]
M[ 0001 0000 0000 0000 00 00 0001 00 01 00 10]
M[ 0001 0000 0000 0000 00 00 0001 00 01 00 11]
.....
M[ 0001 0000 0000 0000 00 00 0001 00 11 11 00]
M[ 0001 0000 0000 0000 00 00 0001 00 11 11 01]
M[ 0001 0000 0000 0000 00 00 0001 00 11 11 10]
M[ 0001 0000 0000 0000 00 00 0001 00 11 11 11]

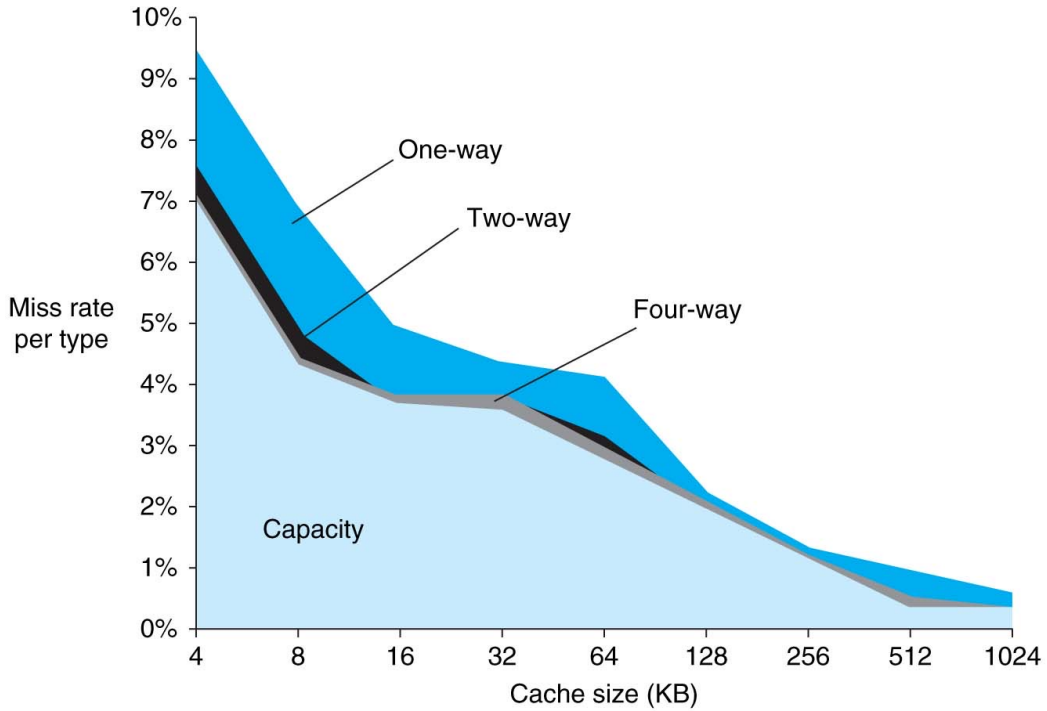


# Cache Performance as a function of associativity



# Cache misses

- Compulsory misses
- Capacity misses
- Conflict misses

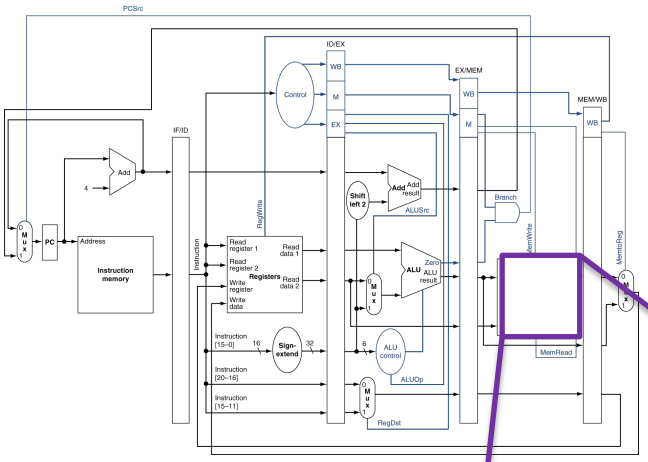



| Design Change | Effect on miss rate | Possible negative performance effect |
|---|---|---|
| Increase cache size | Decrease capacity misses | May increase access time |
| Increase set associativity | Decrease miss rate due to conflict misses | May increase access time |
| Increase block size | Decrease miss rate for a wide range of block sized due to spatial locality | Increase miss penalty. Very large block could increase miss rate |

# Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

# Memory Hierarchy



lw $4, 0($3)