EECS4030: Computer Architecture

Memory Hierarchy (II)

Prof. Chung-Ta King

Department of Computer Science

National Tsing Hua University, Taiwan

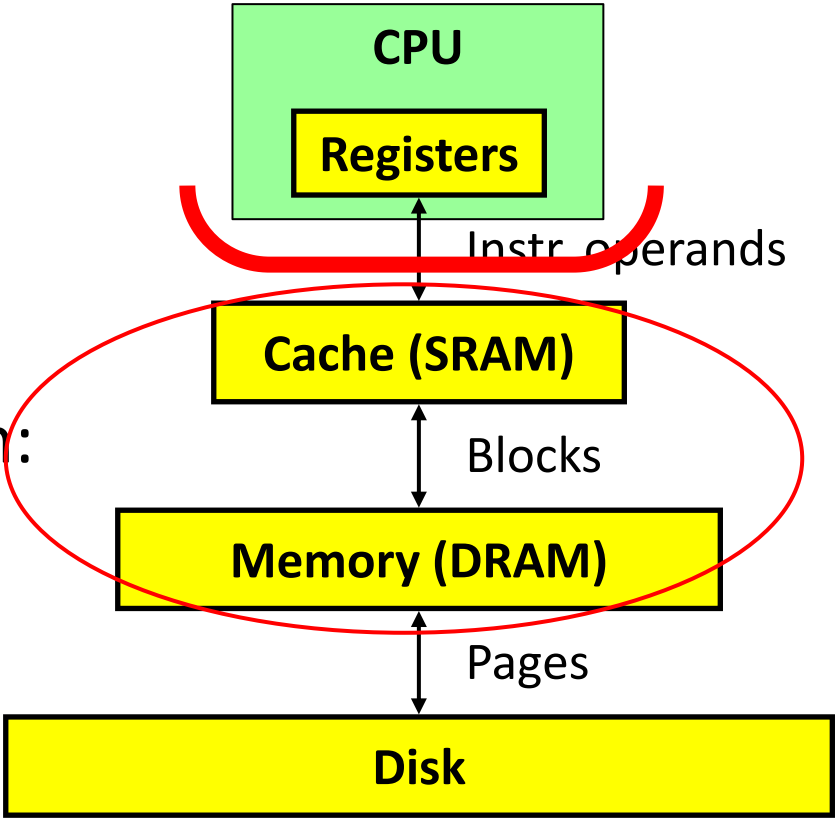
(Adapted from textbook slides https://www.elsevier.com/books-and-journals/book-companion/9780128122754/lecture-slides)

Outline

- Introduction to memory hierarchy (Sec. 5.1)
- Memory technologies (Sec. 5.2, 5.5)
- Caches (Sec. 5.3, 5.4, 5.9)
  - Basic organization and design alternatives (Sec. 5.3)
  - Performance and design tradeoffs (Sec. 5.4)
  - Cache controller (Sec. 5.9)
- Virtual memory (Sec. 5.7)
- Framework for memory hierarchy (Sec. 5.8)
- Virtual machines (Sec. 5.6)
- Parallelism and memory hierarchies (Sec. 5.10, 5.11)

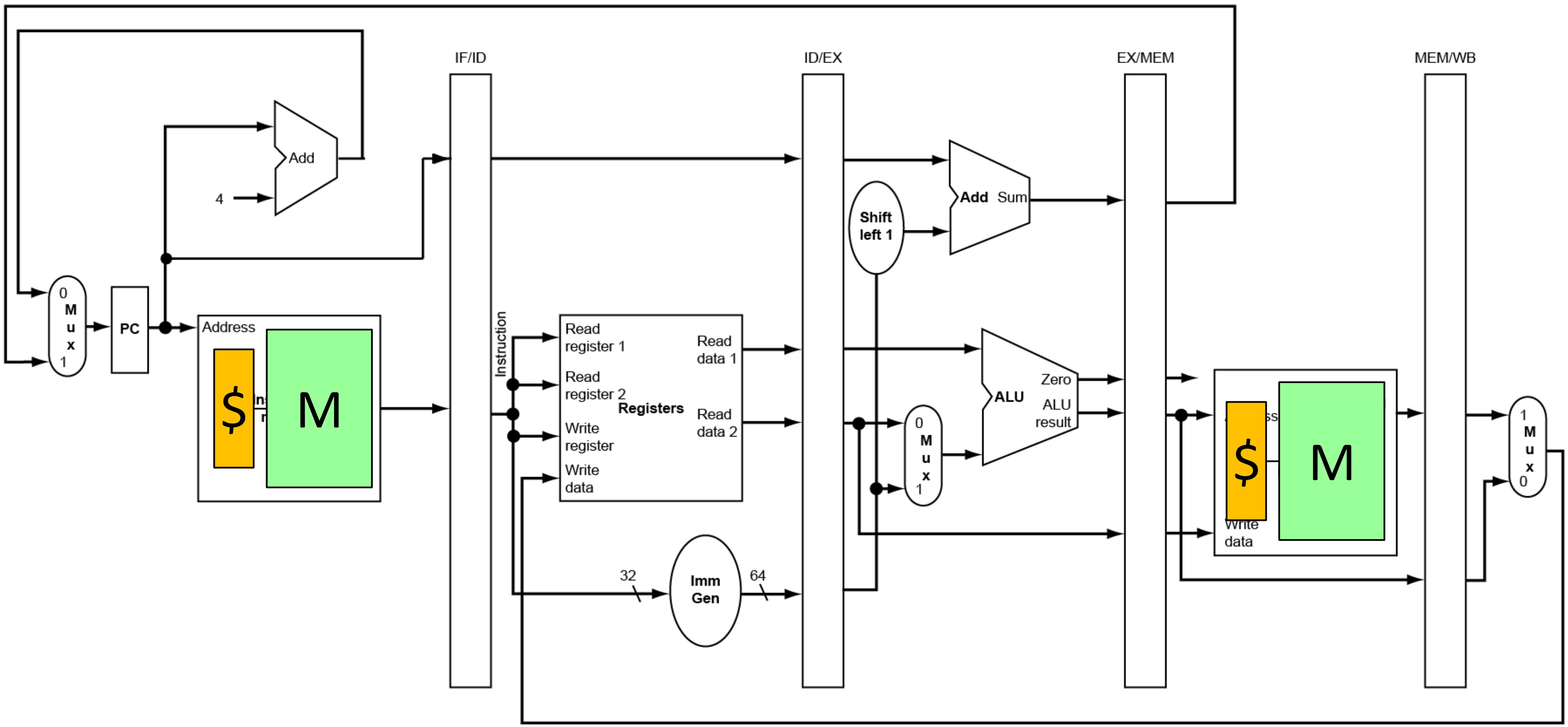
Cache Memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
  - Give CPU an illusion of a very fast main memory
  - Target of instruction fetch, ld and sd



- 4 questions for cache design:
  - Q1: block placement
  - Q2: block identification
  - Q3: block replacement
  - Q4: write policy

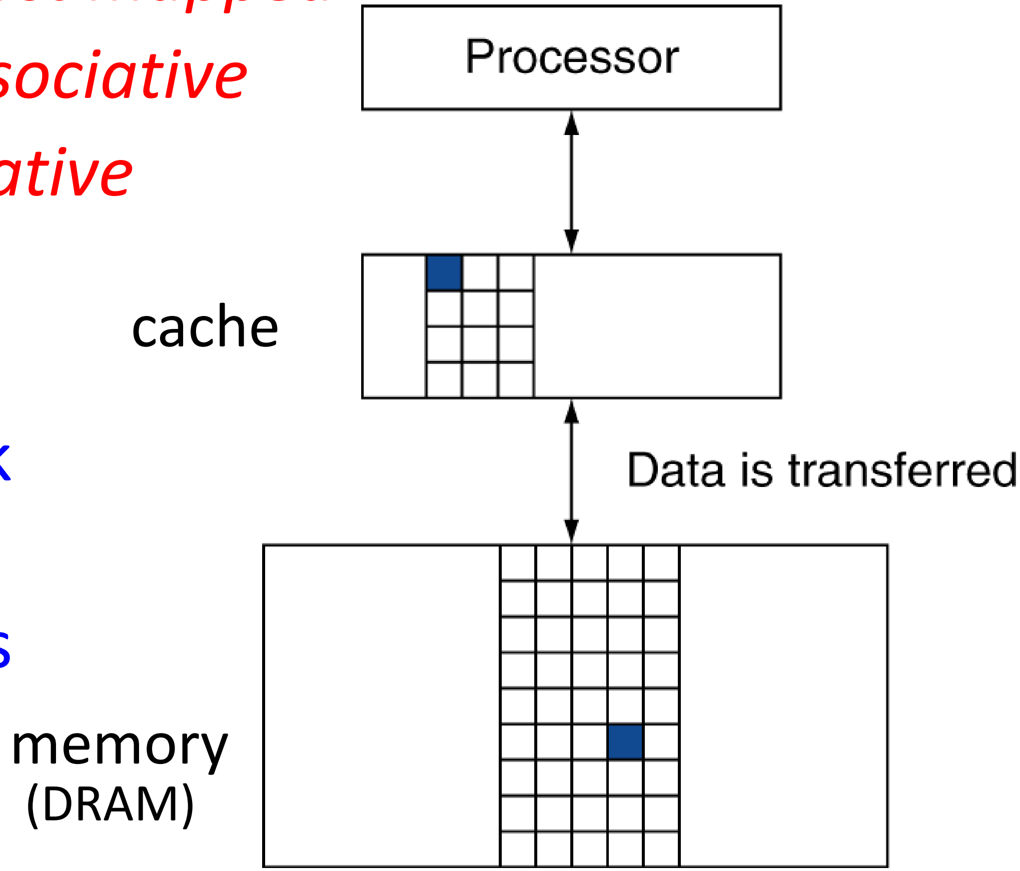
Cache and the CPU Pipeline



Q1: Block Placement

Where can a block be placed in upper level (cache)?

- One fixed location: *direct mapped*
- A few locations: *set associative*
- Anywhere: *fully associative*



- Location is determined by address of the block
- Block placement (Q1) determines how blocks are identified (Q2)

Direct Mapped Cache

- Direct mapped: only one choice
  - For each block in main memory, there is only one location in cache where it can go
  - (Block address) modulo (#Blocks in cache), where #Blocks is a power of 2
  - Use low-order address bits

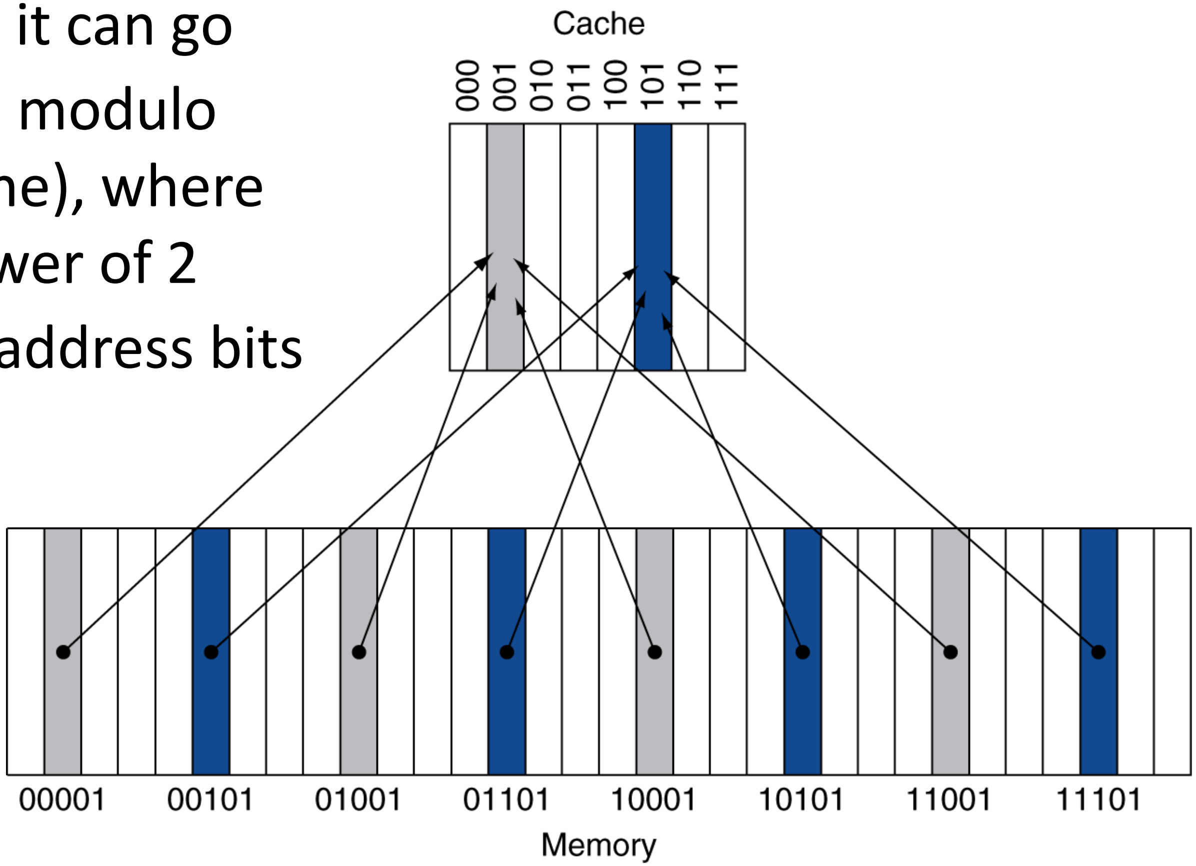


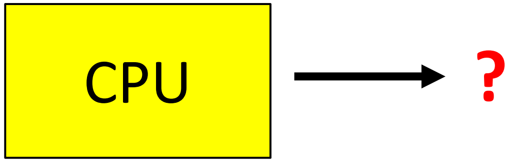
Fig. 5.8

Block Identification for Direct Mapped (Q2)

- Since a block can be stored in only one location in cache, we only need to check that location to find the block → simple and fast
- But, how do we know that location stores the right block, as multiple blocks may be stored there?
  - Store block address as well as the data
  - Actually only need the high-order bits, called the *tag*
- What if there is no data in a location?
  - Use *valid* bit for each location in cache: 1 = present, 0 = not present (initial value)

Organization of Direct Mapped Cache

- 1024 blocks, 1 word/block
  - 4 bytes/block
  - *Cache index*: lower 10 bits
  - *Cache tag*: upper 52 bits
  - *Valid bit* (0 on start up)
- Storage needed
  - (1+52+32)*1024



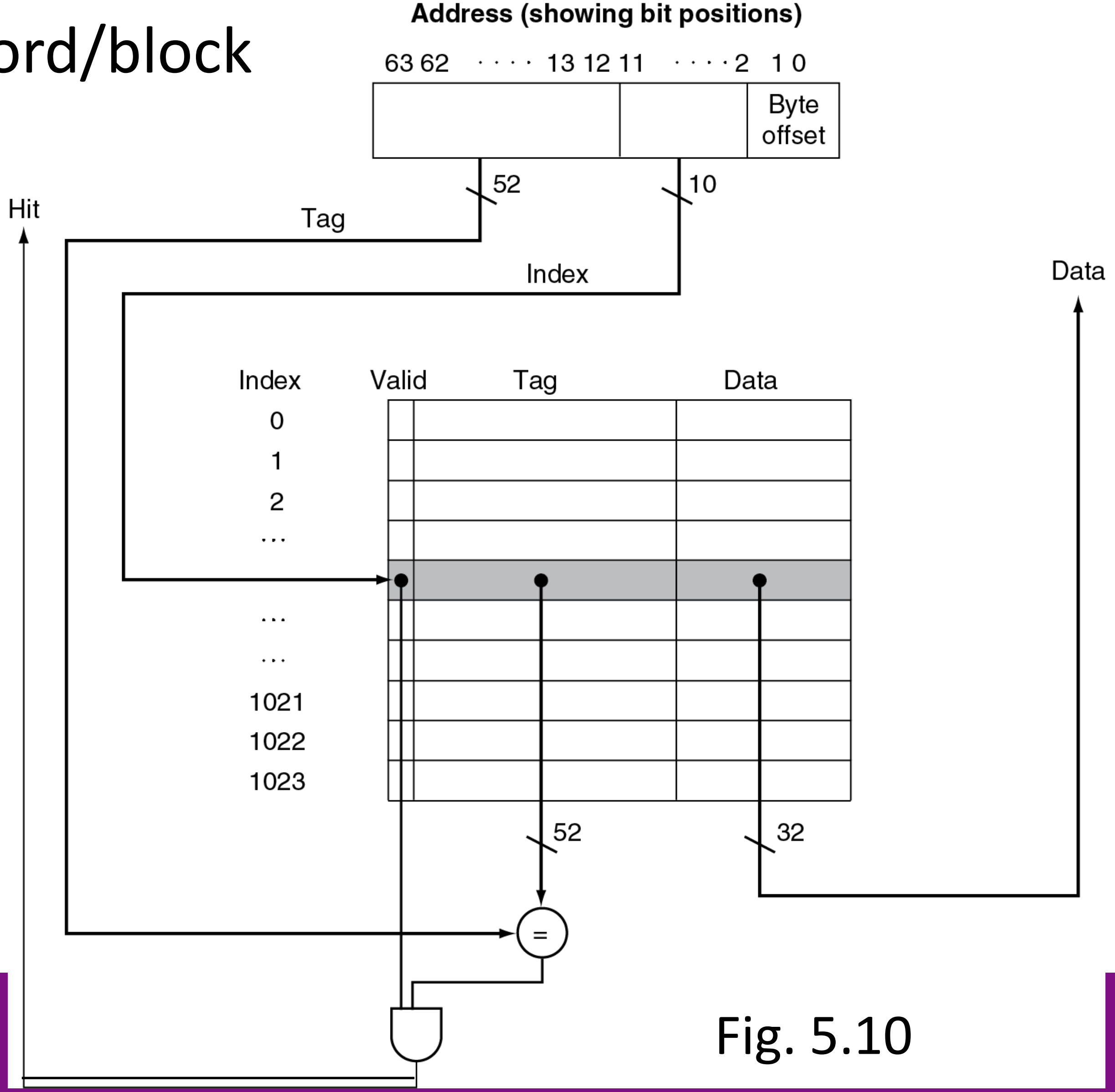
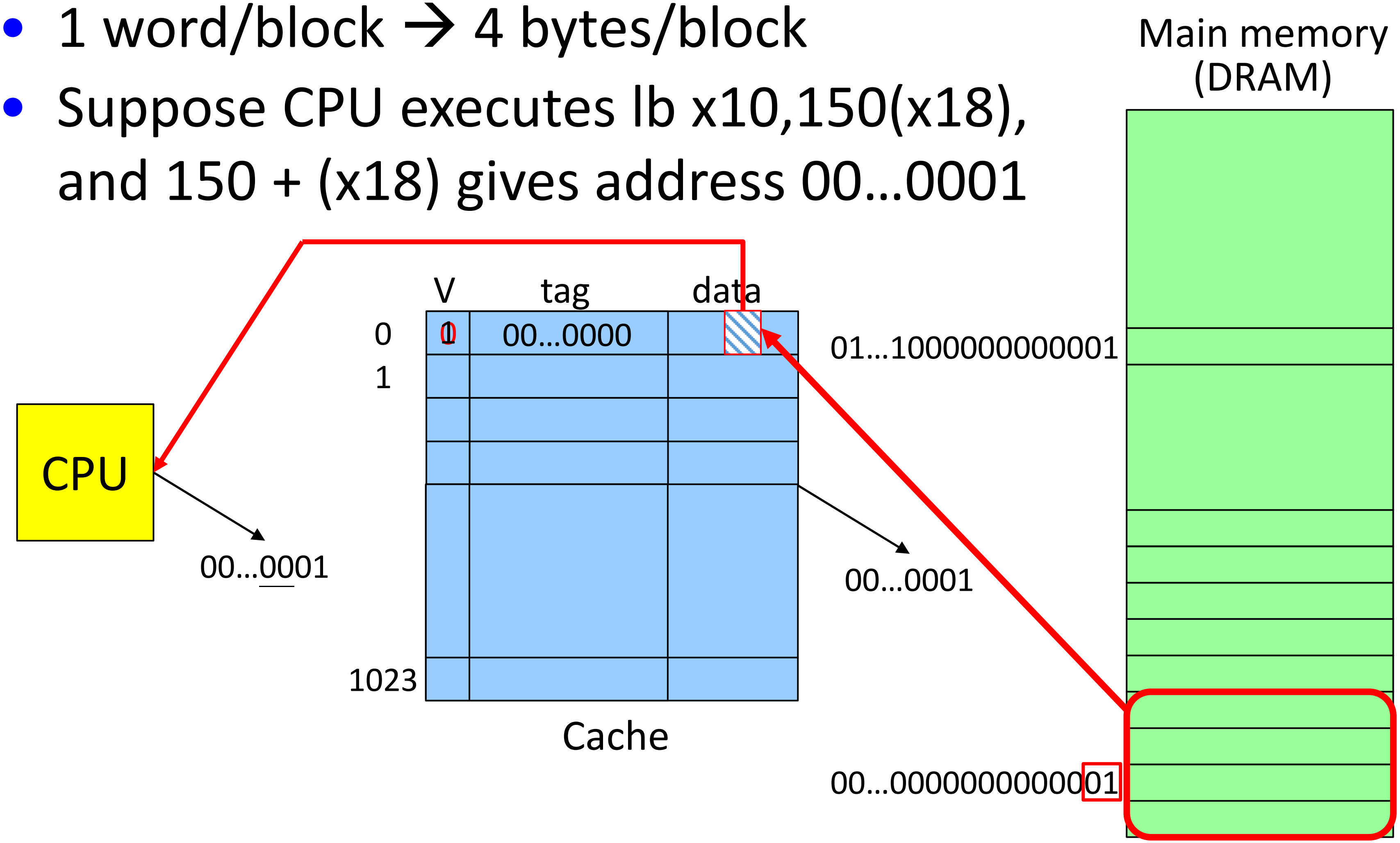


Fig. 5.10

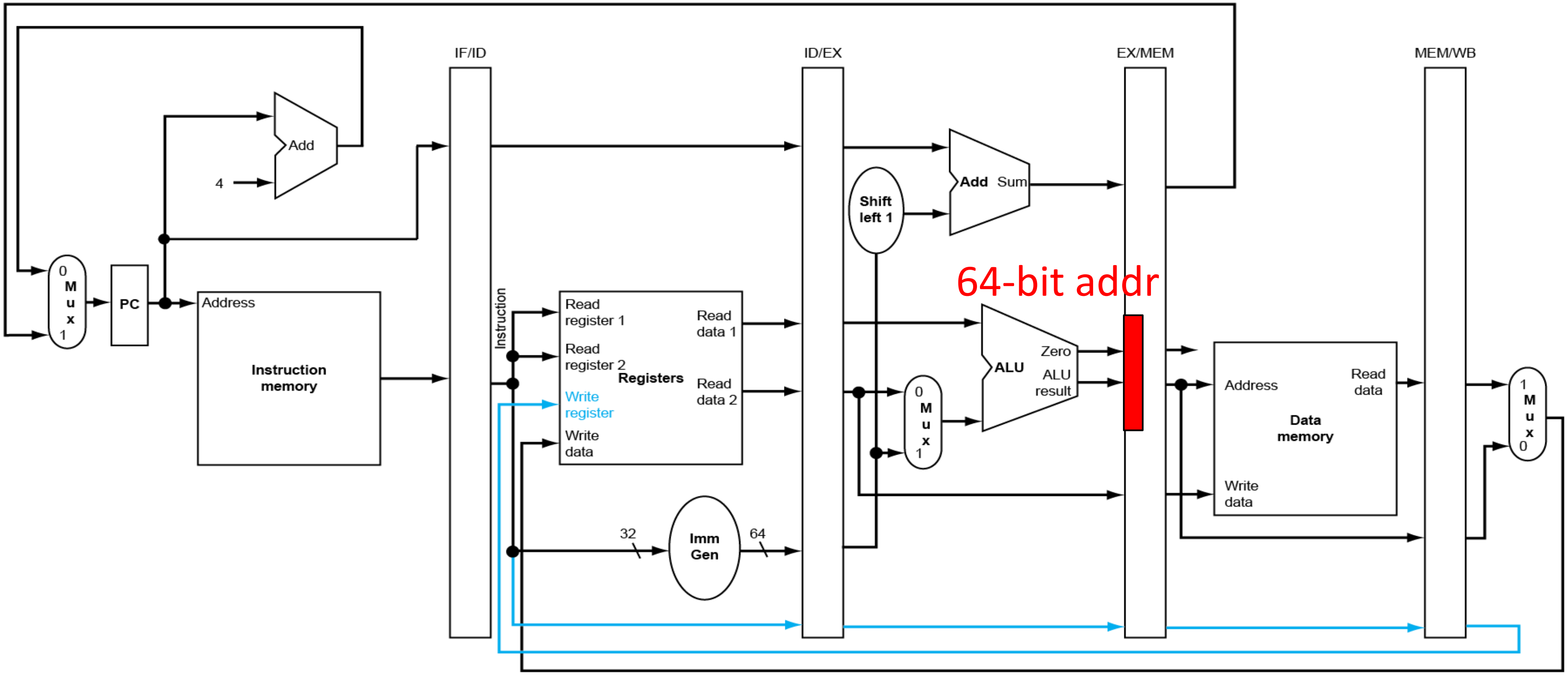
Block and Byte Offset

- 1 word/block → 4 bytes/block
- Suppose CPU executes lb x10,150(x18), and 150 + (x18) gives address 00…0001



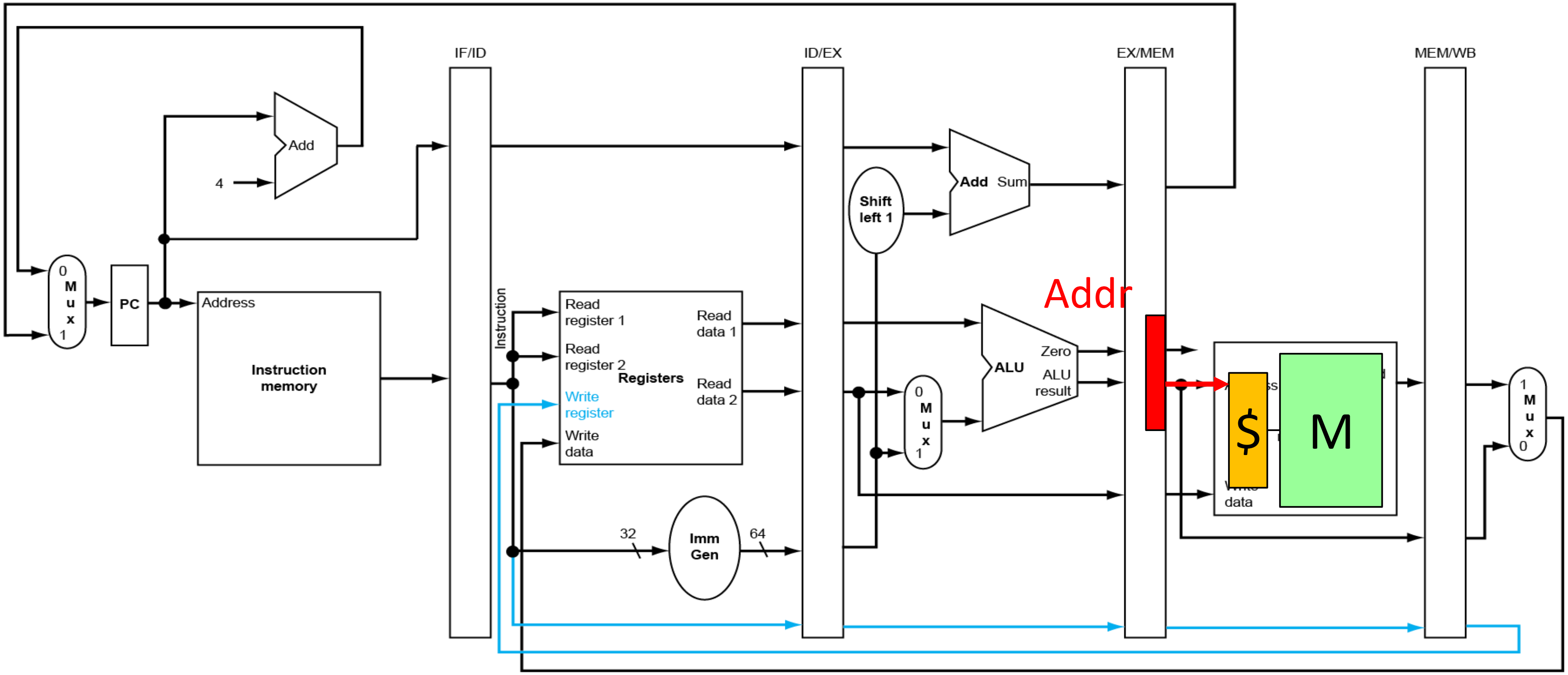
Consider lb x10,150(x18)

- Similar to ld
- At the EXE stage:

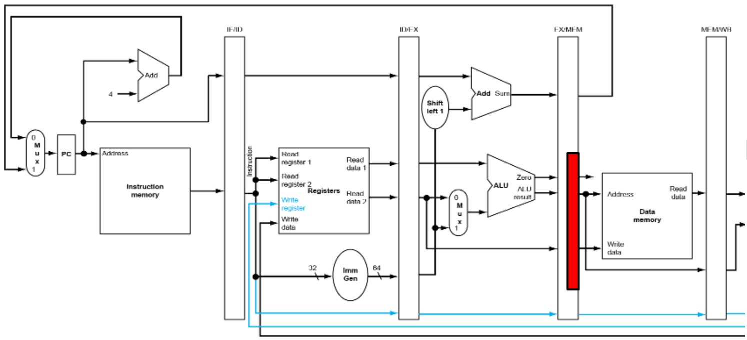


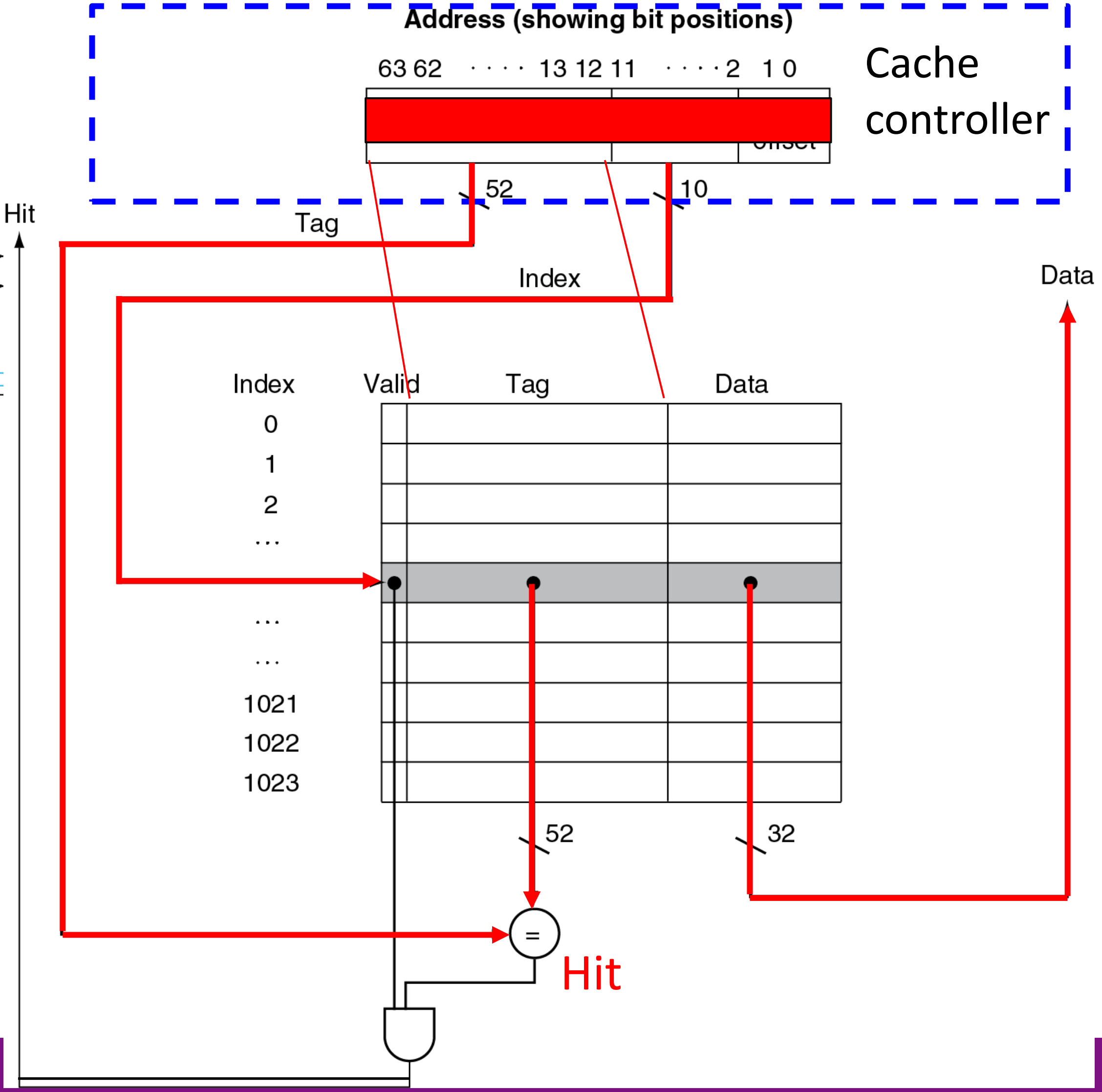
Consider lb x10,150(x18)

- At the MEM stage:



Direct-Mapped Cache





Index bits need not be stored and compared

Example of Direct Mapped Cache

- 8-blocks, 1 word/block, 10-bit address, direct mapped
- Initial state of the cache:

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Example of Direct Mapped Cache

From CPU

Binary addr	Cache block	Hit/miss
0001011000	110	**Miss**

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	**00010**	**Mem[0001011000]**
111	N		

Memory

Example of Direct Mapped Cache

From CPU

Binary addr	Cache block	Hit/miss
0001101000	010	**Miss**

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**00011**	**Mem[0001101000]**
011	N		
100	N		
101	N		
110	Y	00010	Mem[0001011000]
111	N		

Memory

Example of Direct Mapped Cache

From CPU

Binary addr	Cache block	Hit/miss
0001011000	110	**Hit**

Index	V	Tag	Data
000	N		
001	N		
010	Y	00011	Mem[0001101000]
011	N		
100	N		
101	N		
110	Y	00010	Mem[0001011000]
111	N		

CPU

Example of Direct Mapped Cache

From CPU

Binary addr	Cache block	Hit/miss
0001001000	010	**Miss**

Replace

Index	V	Tag	Data
000	Y	00010	Mem[0001000000]
001	N		
010	**Y**	**00010**	**Mem[0001001000]**
011	Y	00000	Mem[0000001100]
100	N		
101	N		
110	Y	00010	Mem[0001011000]
111	N		

Memory

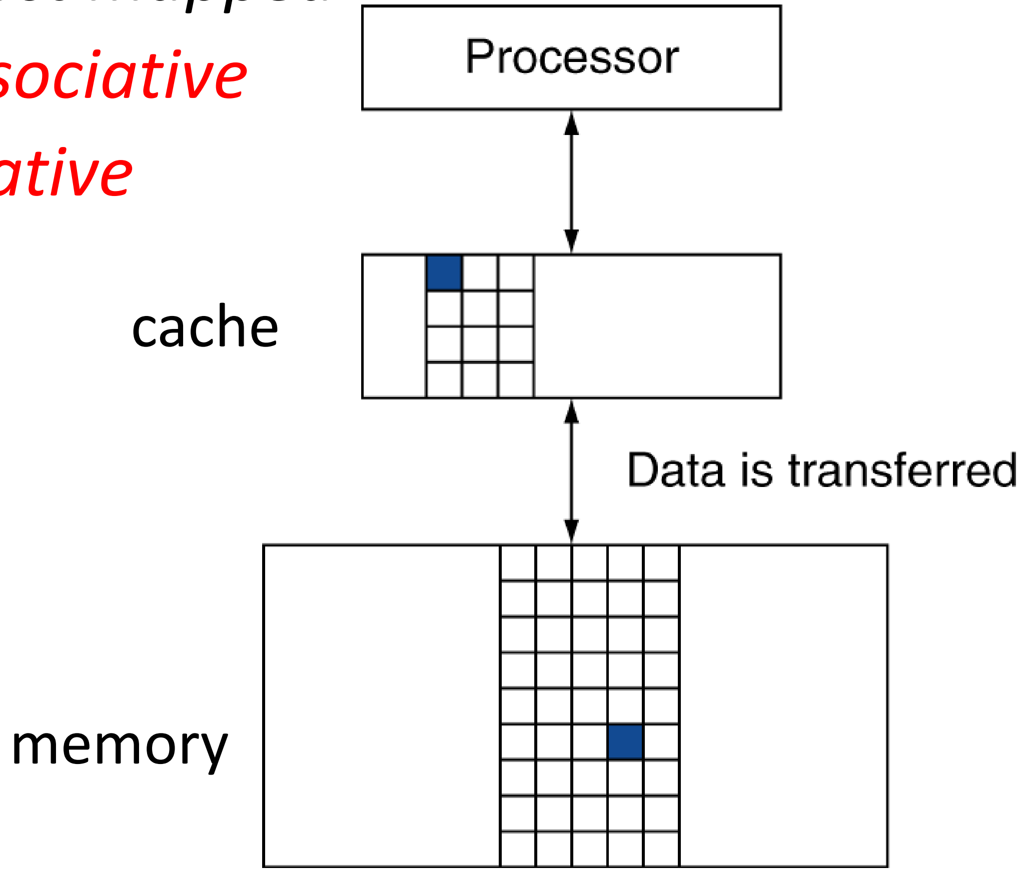
Cache Misses

- On cache hit (instruction fetch, ld, sd), CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from main memory (DRAM)
  - Instruction cache miss: restart instruction fetch
  - Data cache miss: complete data access
  - The time to service a cache miss is called *miss penalty*
- Problem with direct mapped: poor temporal locality
  - Many addresses may map to the same location in cache
  - The next time block A is accessed, its entry may be replaced by another block A’, even if other entries in cache are free

Other Block Placement Policies

Q1: Where can a block be placed in upper level (cache)?

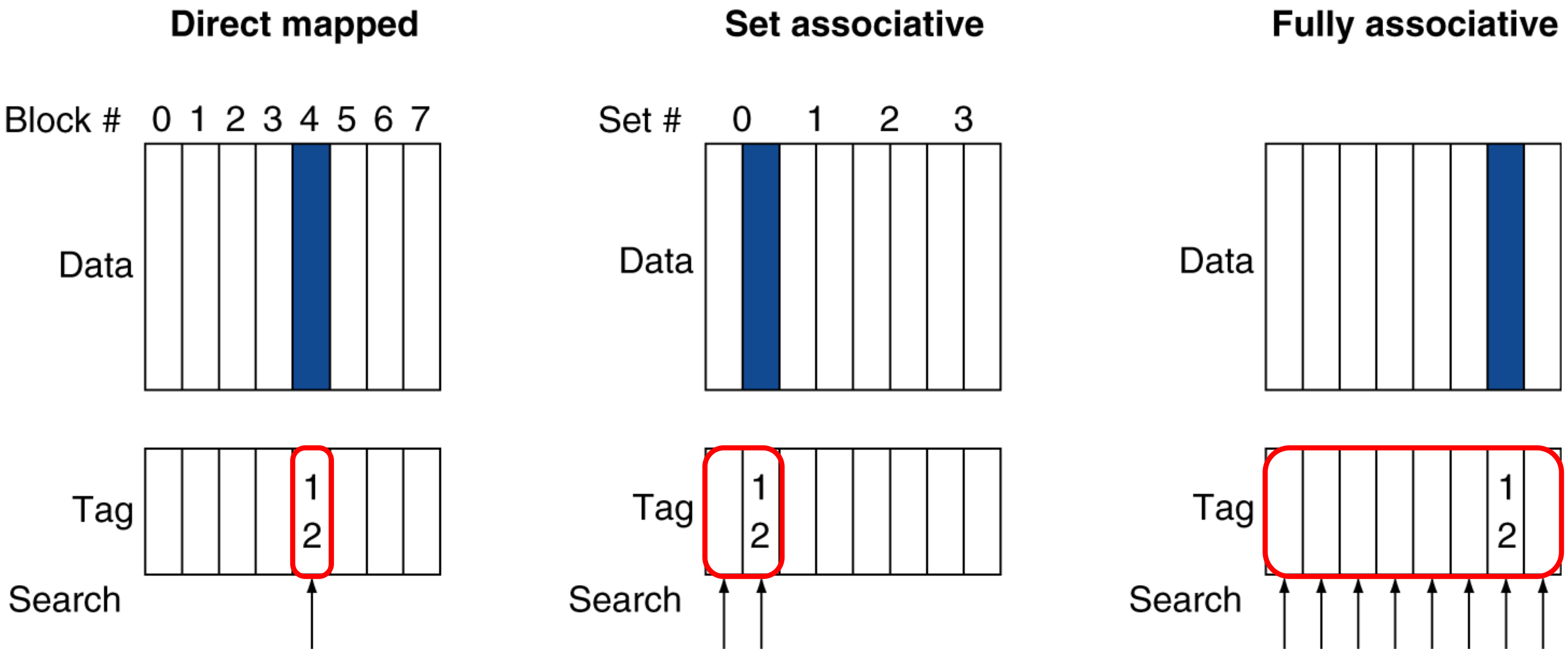
- One fixed location: *direct mapped*
- A few locations: *set associative*
- Anywhere: *fully associative*



Associative Caches

- Fully associative:
  - Let a block go in any cache entry → hard to be replaced
    - Replaced only when the cache is full
  - Cache identification: all entries are searched at once to find the block → a comparator per entry (expensive)
- n-way set associative: (a compromise)
  - Each set contains *n* entries and a given block can go to any entries in a set → fewer locations to place
  - Cache identification:
    - Block address determines which set:
      (Block number) modulo (#Sets in cache)
    - Search all entries in a given set at once → *n* comparators (less expensive)

Associative Cache Example



Data R/W and tag search at the same time

Fig. 5.14

Data R/W only after tag search & hit/miss decision

Spectrum of Associativity

- For a cache with 8 entries

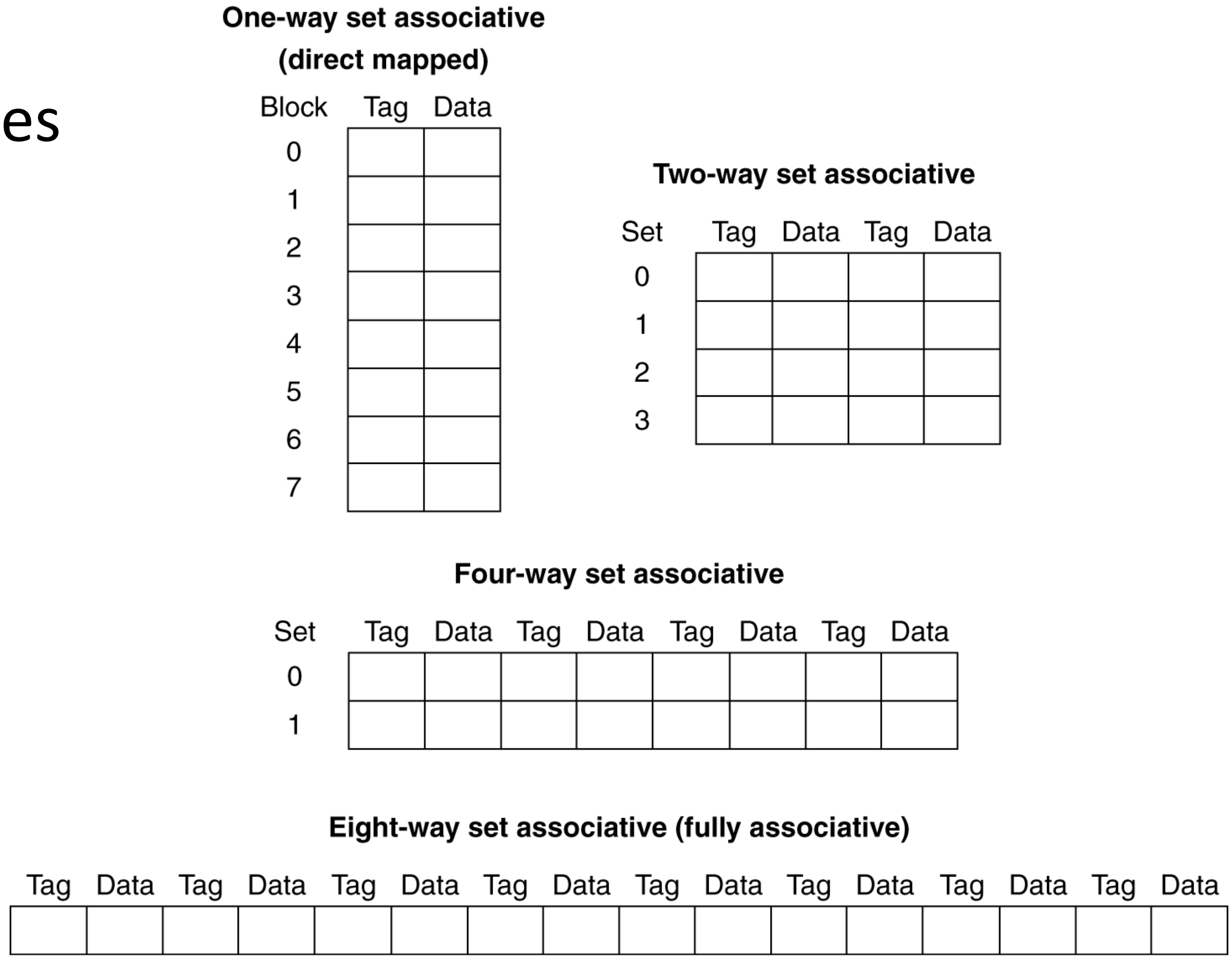


Fig. 5.15

Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped:

Block address	Cache index	Hit/miss	Cache content after access			
			Blk 0	Blk 1	Blk 2	Blk 3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Space not fully utilized

Associativity Example

- 2-way set associative:

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

Replacement matters

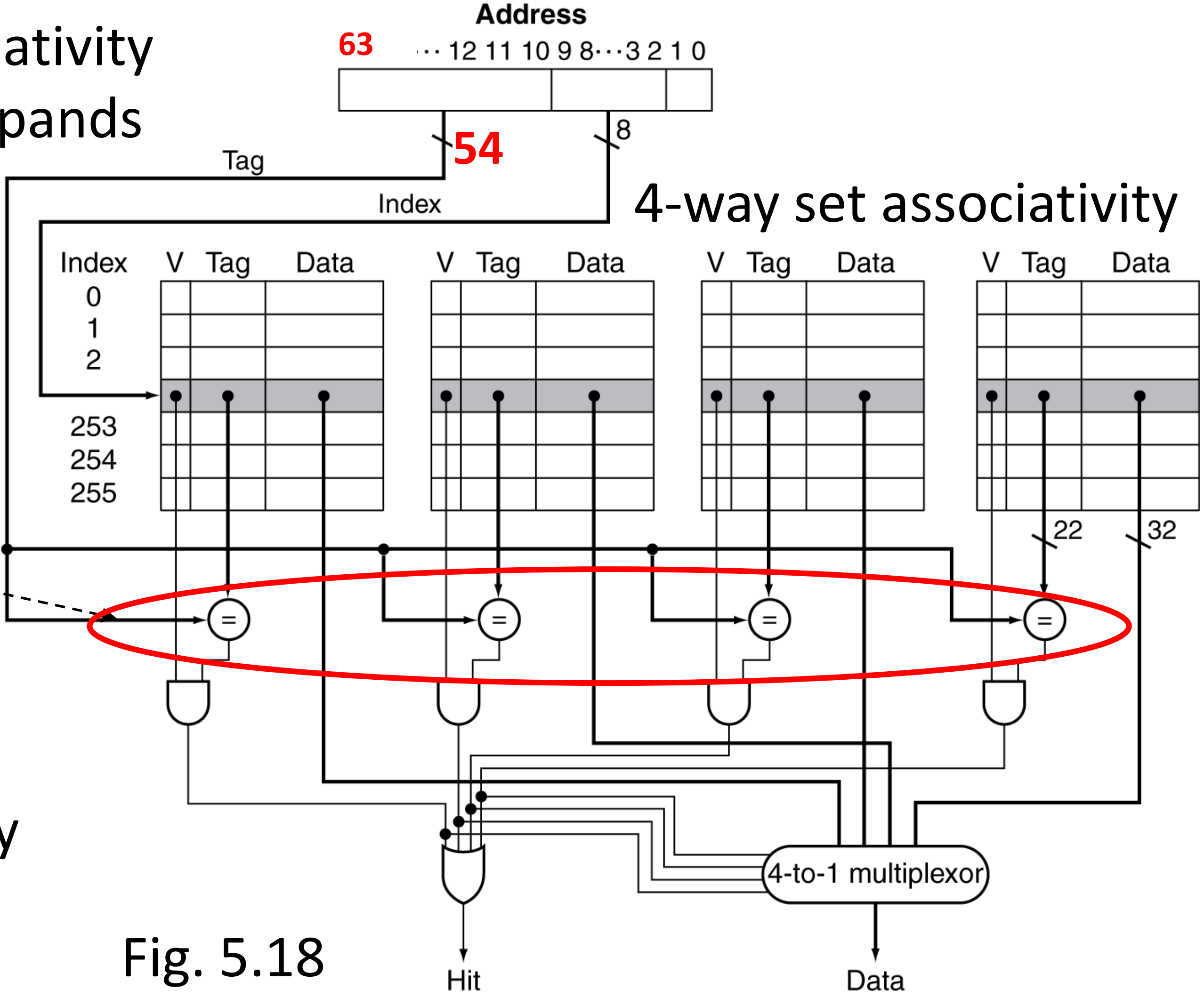
- Fully associative:

associativity ↑ ➔ miss rate ↓

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

Set Associative Cache Organization

- Increasing associativity shrinks index, expands tag



4-way set associativity

4 comparators

Fig. 5.18

- Fully associativity → no index field

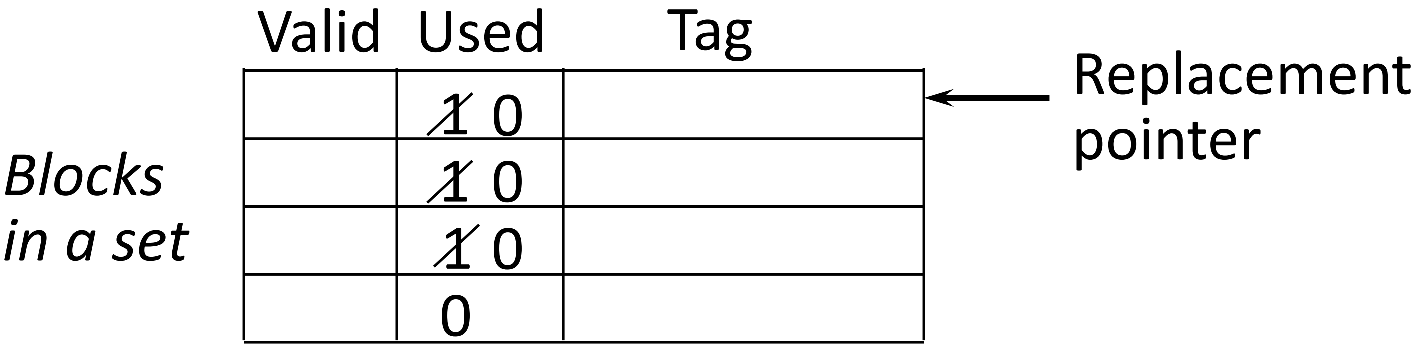
Q3: Block Replacement

Which block should be replaced on a miss?

- Direct mapped: no choice
- Set associative:
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Replacement policy: affect miss rate
  - Ideal: replace the block not to be used farthest in future
  - *Least-recently used* (LRU): use past history to predict future
    - Hardware chooses the one unused for the longest time
    - Simple for 2-way, too hard > 4-way → use approximation
  - *Random*:
    - Give similar performance as LRU for high associativity

An Approximate LRU

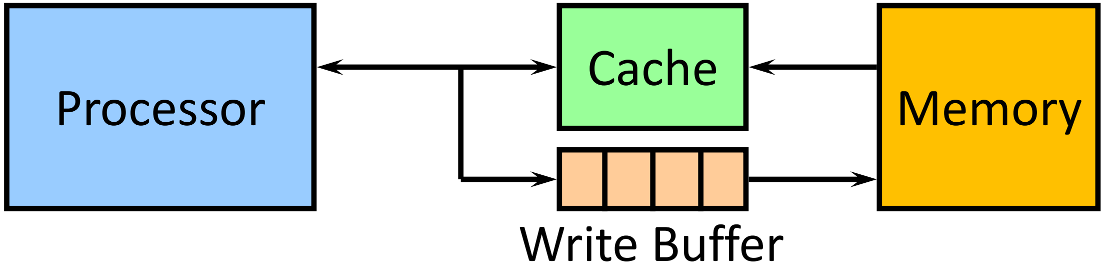
- A "*used bit*" (initial 0) is associated with every block
  - When a block is accessed (hit or miss), its used bit is set to 1
  - On a replacement, a *replacement pointer* scans through the blocks to find a block with used bit = 0 to replace
  - Along the way, the replacement pointer also reset the encountered used bits from 1 to 0
    - Alternatively, if on an access, all other used bits in a set are 1, they are reset to 0 except the bit of the block that is accessed



Valid	Used	Tag
	~~1~~ 0	
	~~1~~ 0	
	~~1~~ 0	
	0	

Q4: Write Policy

- On write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- *Write through*: also update memory
  - But makes write hits very long, 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 as if write is done, and stalls on write only if write buffer is full



Write Policy

- Alternative: *Write back*
  - On write hit, only update block in cache but not in memory
    - Faster write hits, less traffic to memory
    - But cache and memory are inconsistent
  - Need to track whether a block is dirty, by adding a *dirty bit* to each cache block
  - When a dirty block is to be replaced by a newly requested block, write it back to memory → longer read/write misses
    - Can use the write buffer to hold the write-back block while the newly requested block is read from memory. The write-back block is later written back to memory.

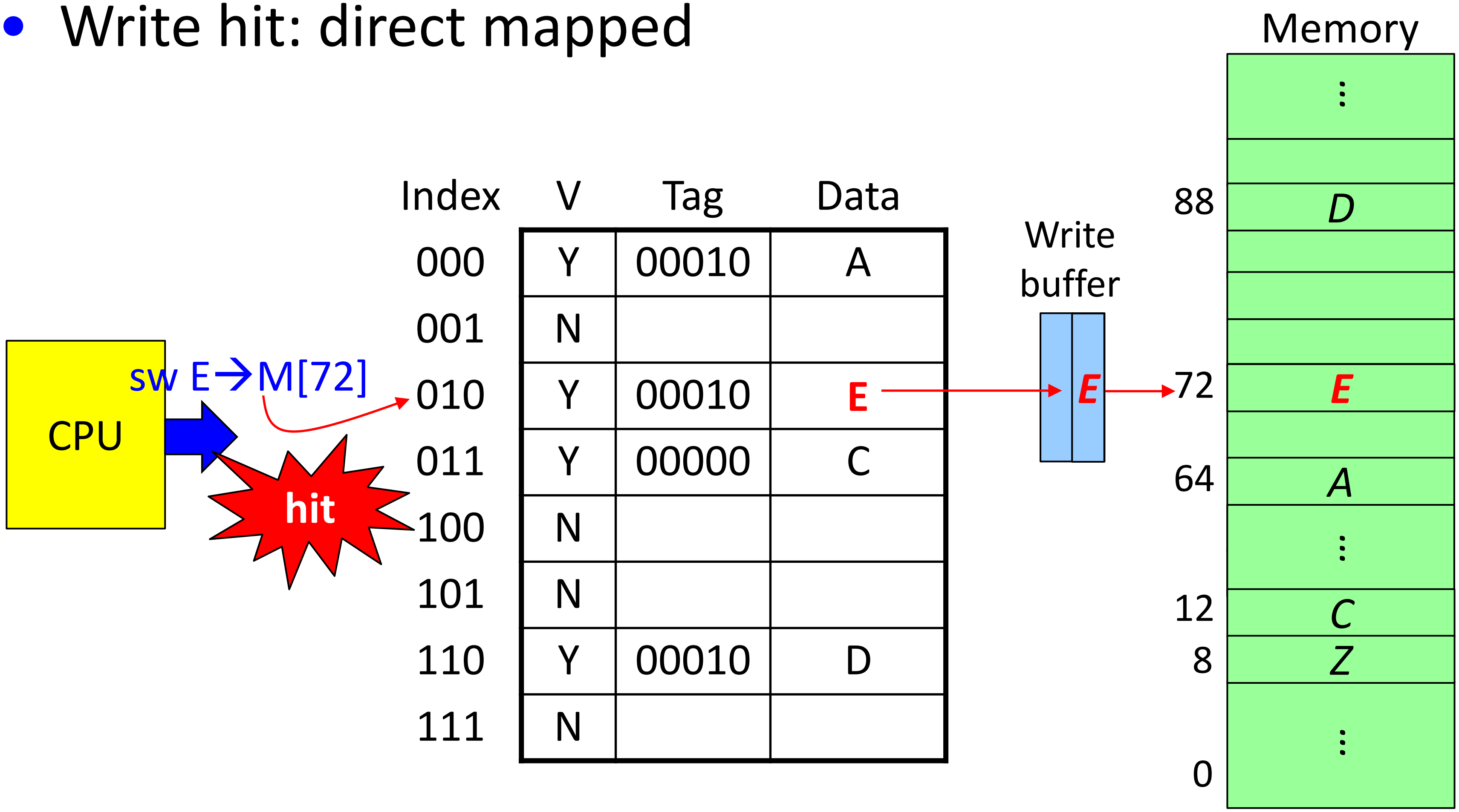
Write Policy

How to handle the write-missed block, which is in the memory?

- For write through
  - *Write allocate* on miss: fetch the block into cache
    - So that later reads from the block will hit
  - *No write allocate*: don’t fetch the block into cache
    - Since programs often write a whole array before reading it, e.g., initialization
    - Also, writes usually end a series of computations
- For write back
  - Usually fetch the block into cache (write allocate)

Example: Write Through with Allocate

- Write hit: direct mapped

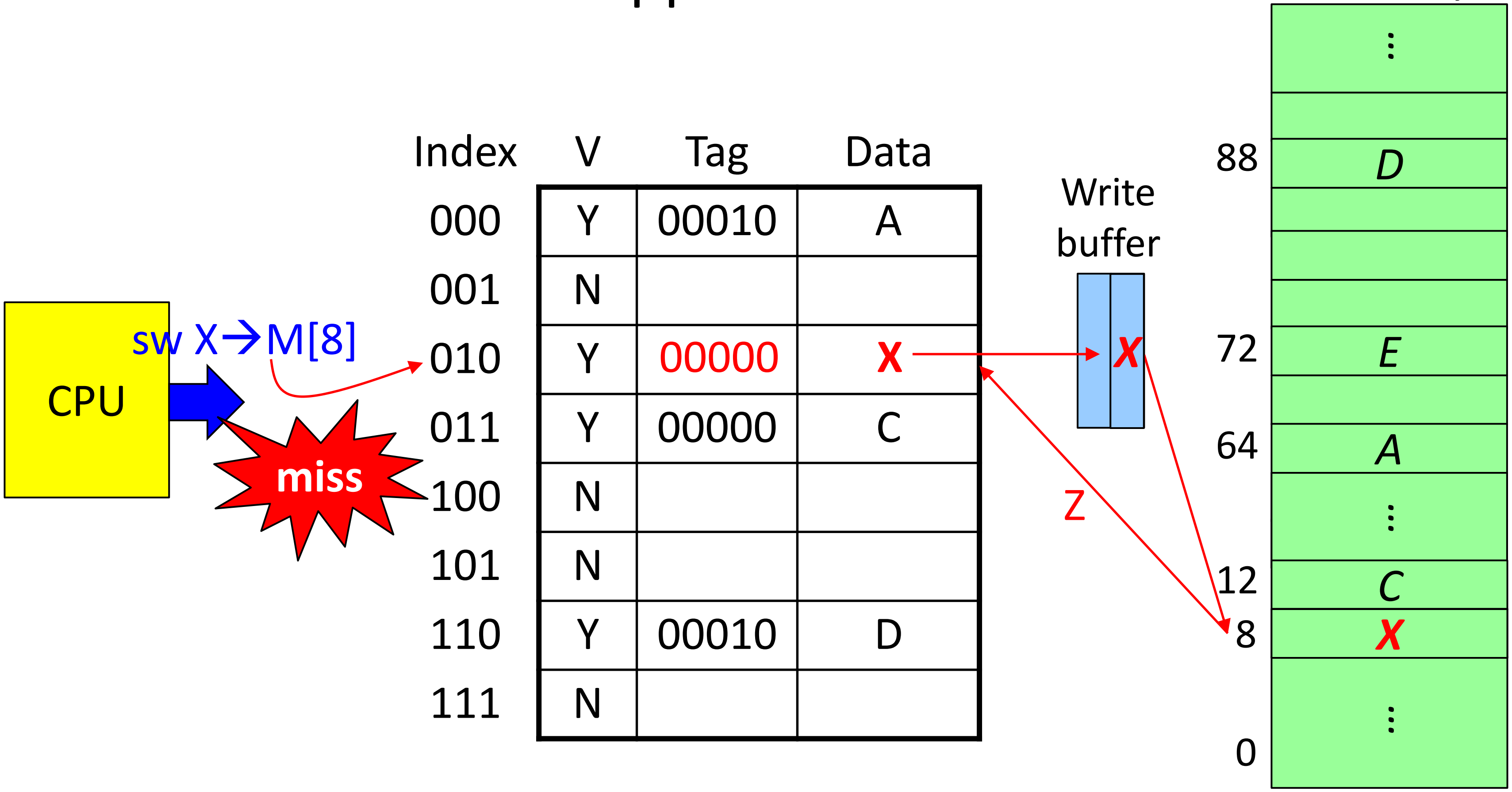


Index	V	Tag	Data
000	Y	00010	A
001	N		
010	Y	00010	E
011	Y	00000	C
100	N		
101	N		
110	Y	00010	D
111	N		

00010 010 00 = 72

Example: Write Through with Allocate

- Write miss: direct mapped

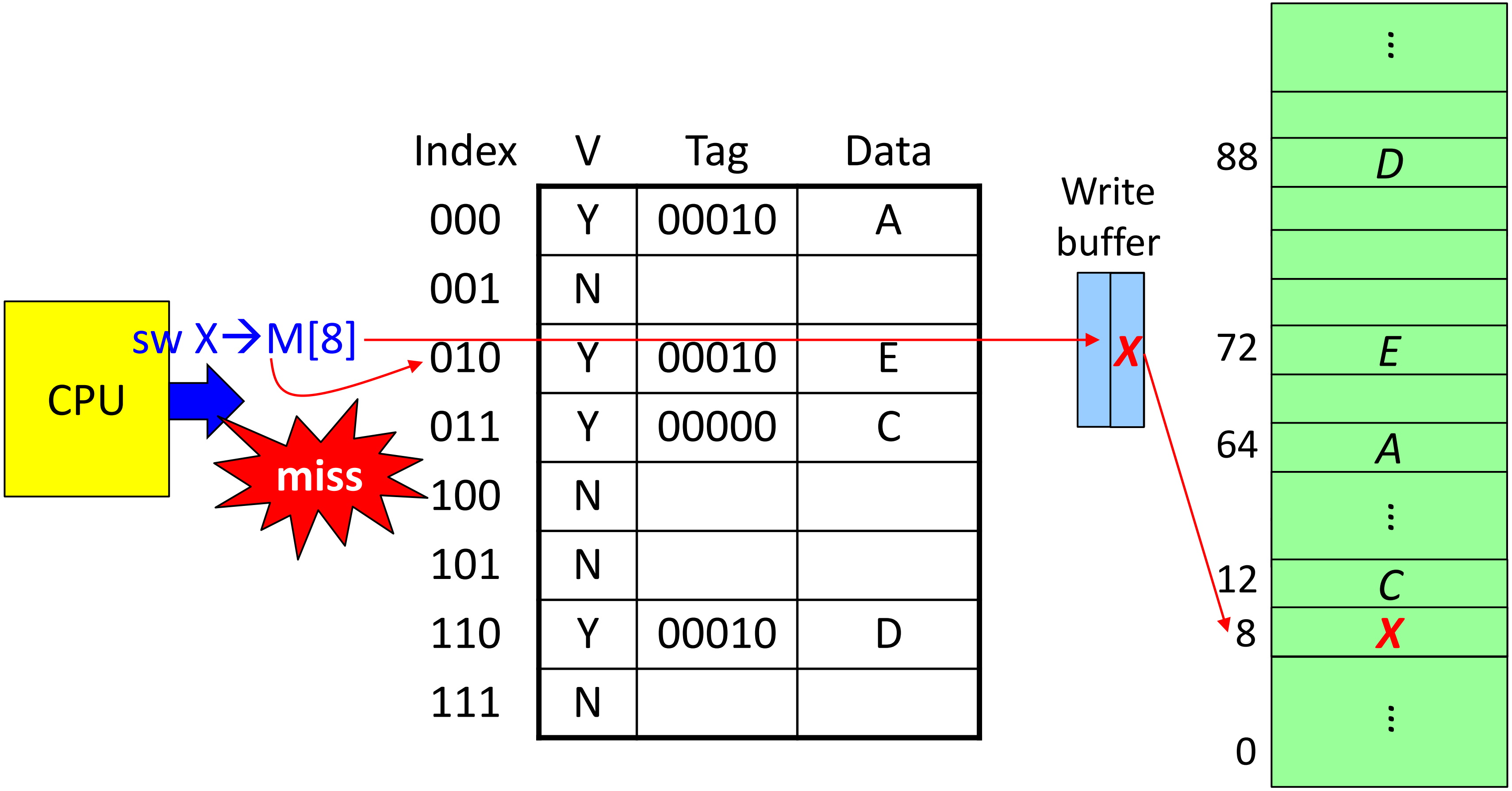


Index	V	Tag	Data
000	Y	00010	A
001	N		
010	Y	00000	X
011	Y	00000	C
100	N		
101	N		
110	Y	00010	D
111	N		

  - What happen on read miss, e.g., lw x2←M[72]?

Example: Write Through with No-Allocate

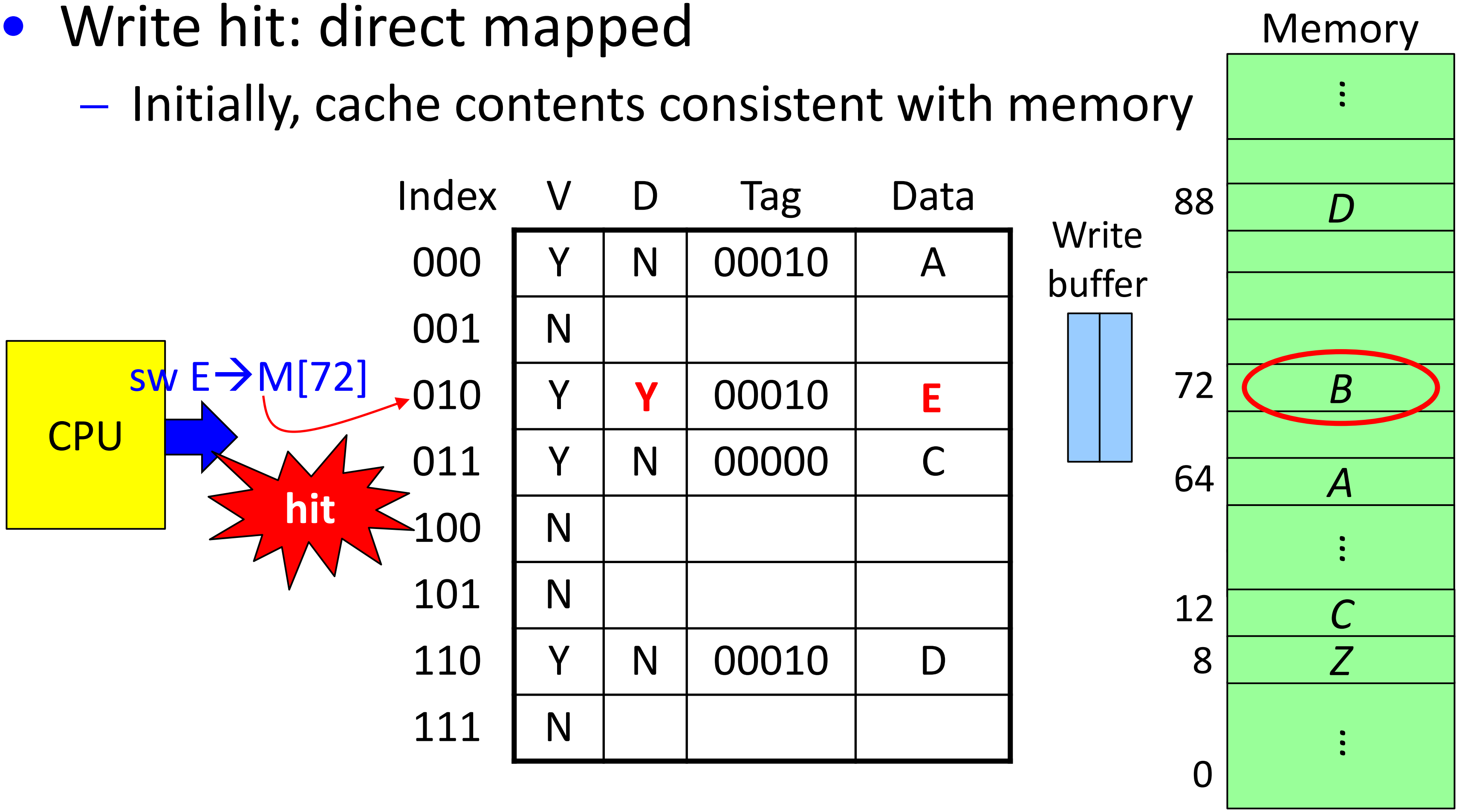
- Write miss under no write allocate



Index	V	Tag	Data
000	Y	00010	A
001	N		
010	Y	00010	E
011	Y	00000	C
100	N		
101	N		
110	Y	00010	D
111	N		

Example: Write Back

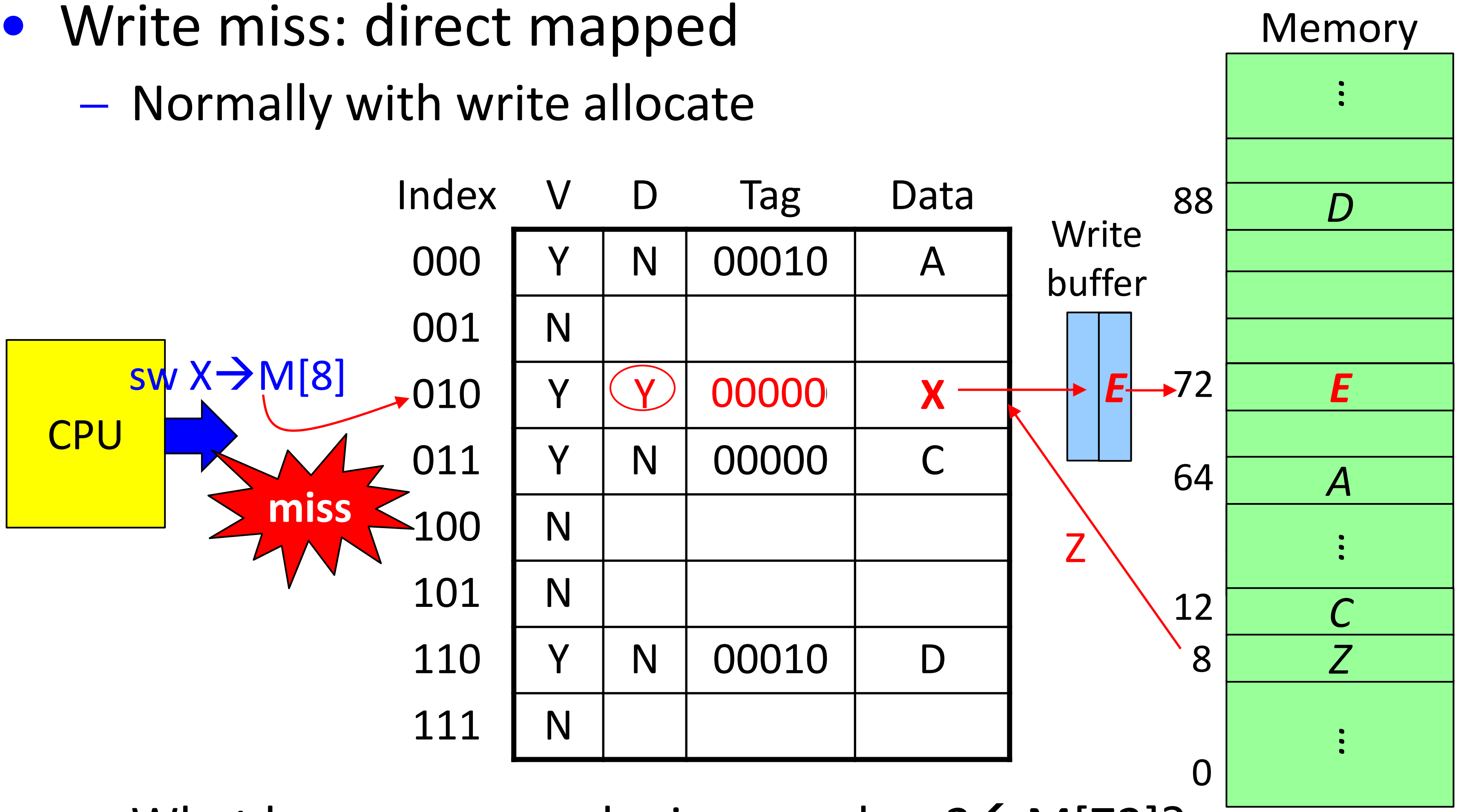
- Write hit: direct mapped
  - Initially, cache contents consistent with memory



Index	V	D	Tag	Data
000	Y	N	00010	A
001	N			
010	Y	Y	00010	E
011	Y	N	00000	C
100	N			
101	N			
110	Y	N	00010	D
111	N			

Example: Write Back

- Write miss: direct mapped
  - Normally with write allocate



Index	V	D	Tag	Data
000	Y	N	00010	A
001	N			
010	Y	Y	00000	X
011	Y	N	00000	C
100	N			
101	N			
110	Y	N	00010	D
111	N			

  - What happen on read miss, e.g., lw x2←M[72]?

Summary: Comparing Cache Organizations

- Direct mapped cache:
  - Simple and cheaper to implement:
    - Only one location to store a block in the cache → only one location to check (by simple indexing) and replace
  - Faster in detecting a cache hit and to replace:
    - Read data and tag at same time; compare only one tag
  - Poor temporal locality → higher miss rate
- N-way set-associative cache:
  - A block has a choice of N locations → higher hit rate
  - More expensive: N comparators, larger tag field
  - Slower:
    - Extra MUX delay to select a block from the set
    - Data comes AFTER hit/miss decision and set selection

Outline

- Introduction to memory hierarchy (Sec. 5.1)
- Memory technologies (Sec. 5.2, 5.5)
- Caches (Sec. 5.3, 5.4, 5.9)
  - Basic organization and design alternatives (Sec. 5.3)
  - Performance and design tradeoffs (Sec. 5.4)
  - Cache controller (Sec. 5.9)
- Virtual memory (Sec. 5.7)
- Framework for memory hierarchy (Sec. 5.8)
- Virtual machines (Sec. 5.6)
- Parallelism and memory hierarchies (Sec. 5.10, 5.11)

CPU Performance with Cache

- CPU time =

  (CPU execution cycles + Memory stall cycles)

  × Clock cycle time

  - CPU execution cycles: includes cache hit time
  - *Memory stall cycles*: mainly from cache misses

- Memory stall cycles

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

  - Assuming: R/W have same miss rates and penalty; write buffer stalls ignored

CPU Performance with Cache: Example

- Given
  - I-cache miss rate = 2%
    - Every instruction fetched
  - D-cache miss rate = 4%
    - Only for loads/stores; loads and stores are 36% of instructions
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
- Miss cycles per instruction = misses/instr. × penalty
  - I-cache: 0.02 × 100 = 2 cycles/instruction
  - D-cache: 0.36 × 0.04 × 100 = 1.44 cycles/instruction
- Actual CPI = 2 + 2 + 1.44 = 5.44 cycles/instruction
  - Ideal CPU is 5.44/2 =2.72 times faster

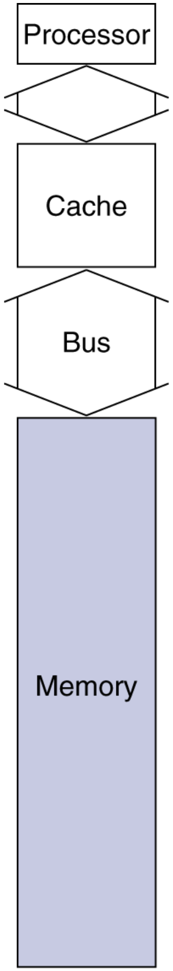
Average Memory Access Time

Another performance metric

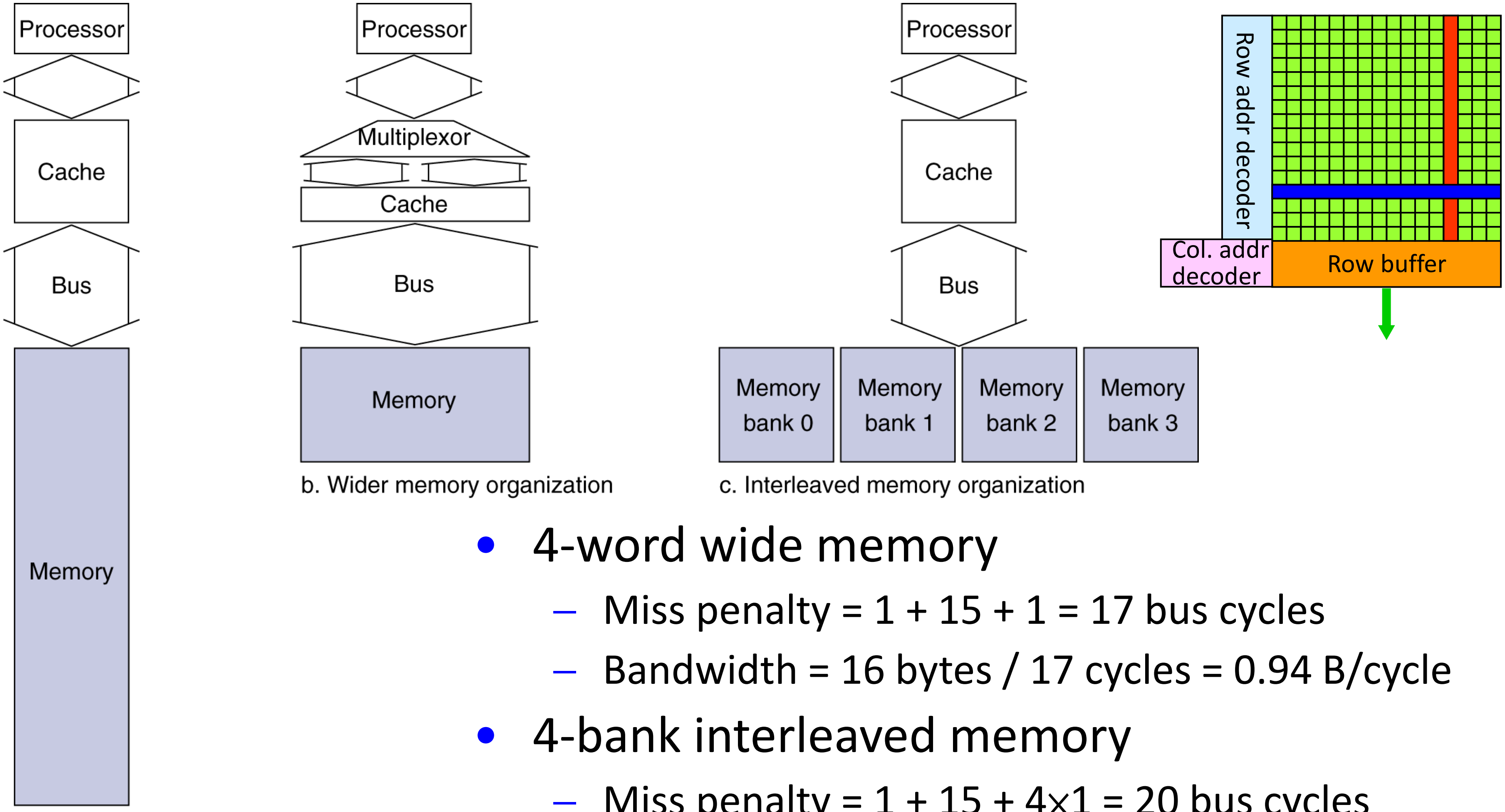
- Hit time, though counted in CPU execution cycle, cannot be ignored in considering CPU performance
  - Tend to increase with more flexible and complex cache designs → may dominate the improvement in hit rate
- Consider *Average memory access time* (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example: AMAT per instruction
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, miss rate = 0.05 misses per instruction (I+D caches)
  - R/W miss penalties are the same; ignore other write stalls
  - AMAT/instruction = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Main Memory Supporting Caches

- Use DRAMs for main memory
  - Fixed width (e.g., 1 word)
  - Connected by fixed-width clocked bus
    - Bus clock is typically slower than CPU clock
- Example cache block read
  - 1 bus cycle for address transfer
  - 15 bus cycles per DRAM access
  - 1 bus cycle per data transfer
- To read a 4-word block with 1-word-wide memory
  - Miss penalty = 1 + 4×15 + 4×1 = 65 bus cycles
  - Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle



Main Memory Supporting Caches



a. One-word-wide memory organization

b. Wider memory organization

c. Interleaved memory organization

- 4-word wide memory
  - Miss penalty = 1 + 15 + 1 = 17 bus cycles
  - Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle
- 4-bank interleaved memory
  - Miss penalty = 1 + 15 + 4×1 = 20 bus cycles
  - Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle

Implications of Cache Performance

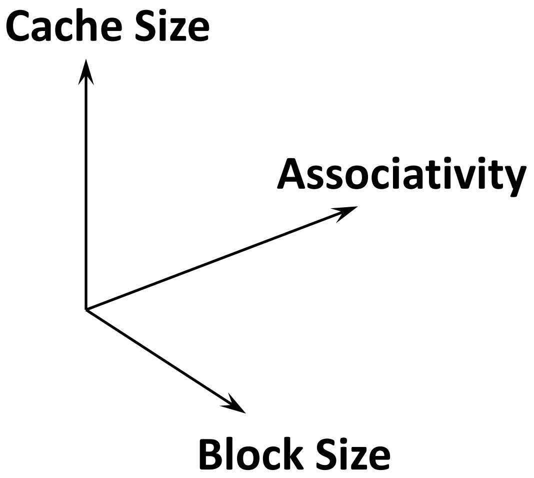
- When CPU performance is increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can’t neglect cache behavior when evaluating system performance

Sources of Cache Misses

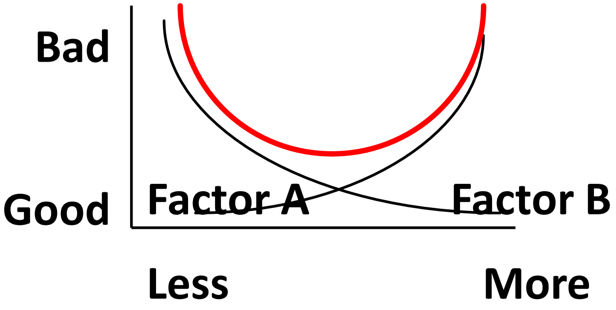
- *Compulsory* (cold start) misses:
  - First access to a block, not much we can do
  - Solution: larger block size
- *Conflict* (collision) misses:
  - >1 memory blocks mapped to same location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- *Capacity* misses:
  - Cache cannot hold all blocks needed by program execution; a replaced block is later accessed again
  - Solution: increase cache size

Cache Design Space

- Many interacting dimensions
  - Cache size
  - Block size
  - Associativity
  - Replacement policy
  - Write-through vs write-back
  - Write allocation

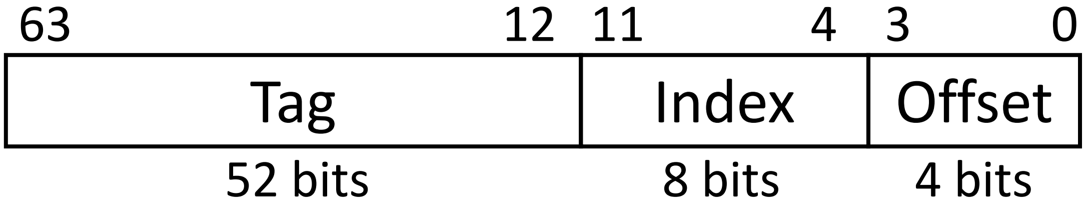


- The optimal choice is a compromise
  - Depends on access characteristics
    - Workload, use (I-cache, D-cache)
  - Depends on technology and cost
- Simplicity often wins



Design Tradeoff: Increase Block Size

- Instead of 1024 blocks and 4 bytes/block, can organize the cache as 256 blocks and 16 bytes/block



63 12 11 4 3 0

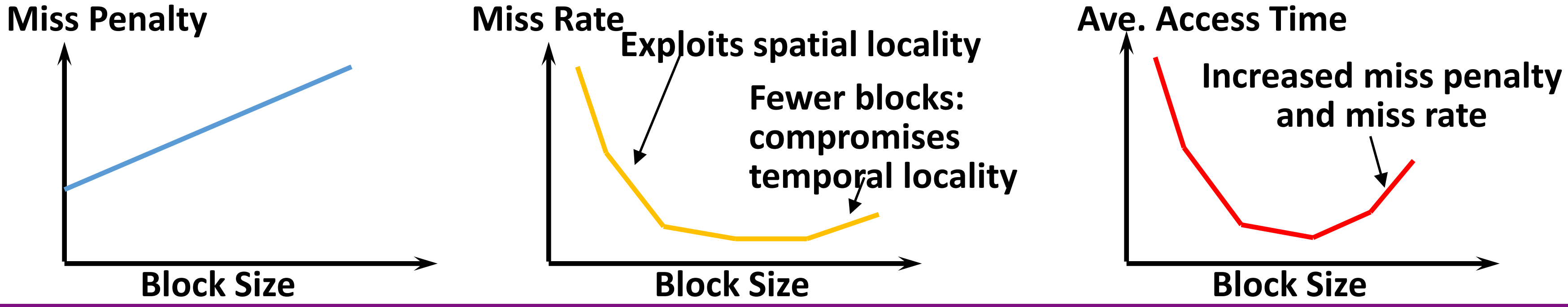
Tag | Index | Offset

52 bits | 8 bits | 4 bits

- To what block number does address 4500 map?
  - Block address = $\lfloor 4500/16 \rfloor$ = 281
    - 4500=00…00100011001$\underline{0100}_2$ /10000_2 → 00…001$00011001_2$
  - Block number = 281 modulo 256 = 25 = 00011001_2

Block Size Considerations

- Larger blocks should reduce miss rate (compulsory misses) due to spatial locality
- But in a fixed-sized cache
  - Larger blocks ⇒ fewer of them, more unused data
    More competition ⇒ increased miss rate (capacity misses)
- Larger blocks have larger miss penalty
  - Can offset benefit of reduced miss rate
  - Early restart and critical-word-first can help



Design Tradeoff: How Much Associativity

- Increasing associativity decreases miss rate (conflict misses)
  - But with diminishing returns
  - Also increases cost (e.g., more comparators) and hit time
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Optimization: Multilevel Caches

Why multilevel cache?

- Primary cache (Level-1) attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
  - Reduce miss penalty
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz → cycle time = 0.25ns
  - Miss rate/instruction = 2%, Memory access time = 100ns
- With just primary (L-1) cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = $1 + 0.02 \times 400$ = 9 cycles/instruction
- Now add L-2 cache
  - Access time = 5ns, miss rate to main memory = 0.5%
  - L-1 cache miss penalty with L-2 hit = 5ns/0.25ns = 20 cycles
  - L-1 cache miss penalty with L-2 miss = 400 cycles
  - CPI = $1 + 0.02 \times 20 + 0.005 \times 400$ = 3.4 cycles/instruction
- Performance ratio = 9/3.4 = 2.6

Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time to give a shorter clock cycle or fewer pipeline stages
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single-level cache, with a block size smaller than in L-2 to match smaller cache and reduce miss penalty
  - L2 cache much larger than 1-level cache (for access time less critical); larger block size, higher associativity than L-1

Cache Design Trade-offs

Design Change	Effect on Miss Rate	Negative Performance Effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty and for very large block size, may increase miss rate due to pollution

Fig. 5.37

Example: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - May have instruction and data accesses on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - Direct mapped
  - D-cache: write-through or write-back decided by OS
  - One-entry write buffer
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

Example: Intrinsity FastMATH

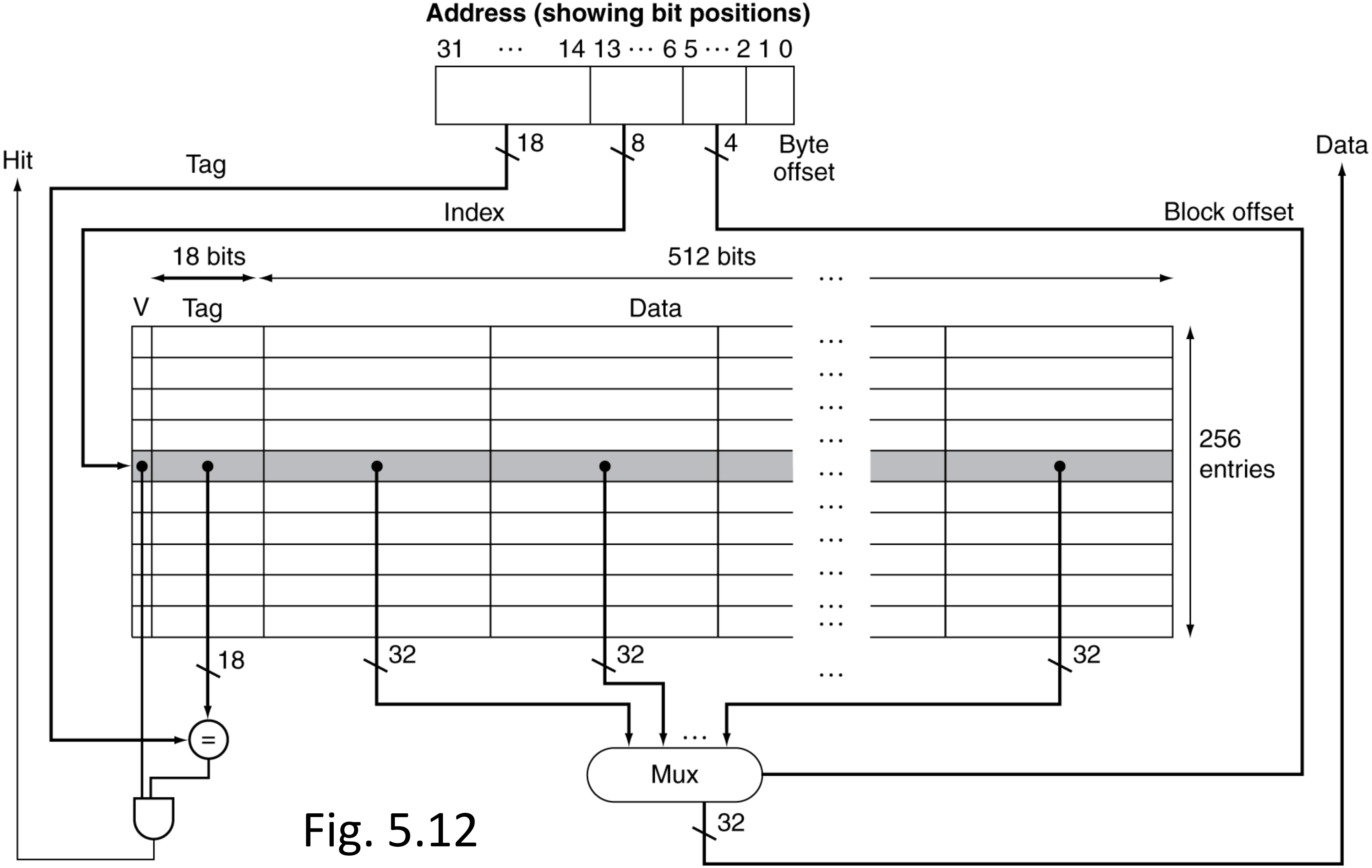


Fig. 5.12

Software Optimization via Blocking

- Goal: maximize accesses to data before it is replaced
- Consider inner loops of DGEMM:

```
for (int j = 0; j < n; ++j)  {
  double cij = C[i+j*n];
  for(int k = 0; k < n; k++)
    cij += A[i+k*n] * B[k+j*n];
  C[i+j*n] = cij;
}
```

Cache misses if k is large

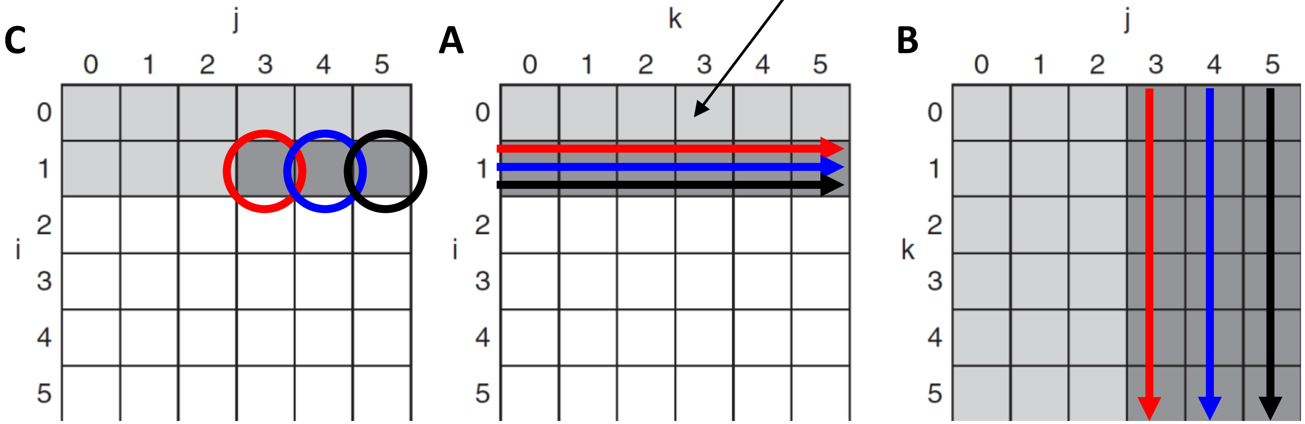


Fig. 5.20

Cache Blocked DGEMM

```
void do_block (int n, int si, int sj, int sk, double *A,
                 double *B, double *C) {
 for (int i = si; i < si+BLOCKSIZE; ++i)
  for (int j = sj; j < sj+BLOCKSIZE; ++j) {
   double cij = C[i+j*n];        /* cij = C[i][j] */
   for( int k = sk; k < sk+BLOCKSIZE; k++ )
     cij += A[i+k*n] * B[k+j*n]; /* cij+=A[i][k]*B[k][j] */
   C[i+j*n] = cij;               /* C[i][j] = cij */
  }
 }
void dgemm (int n, double* A, double* B, double* C) {
 for ( int sj = 0; sj < n; sj += BLOCKSIZE )
  for ( int si = 0; si < n; si += BLOCKSIZE )
   for ( int sk = 0; sk < n; sk += BLOCKSIZE )
    do_block(n, si, sj, sk, A, B, C);
}
```

Blocked DGEMM Access Pattern

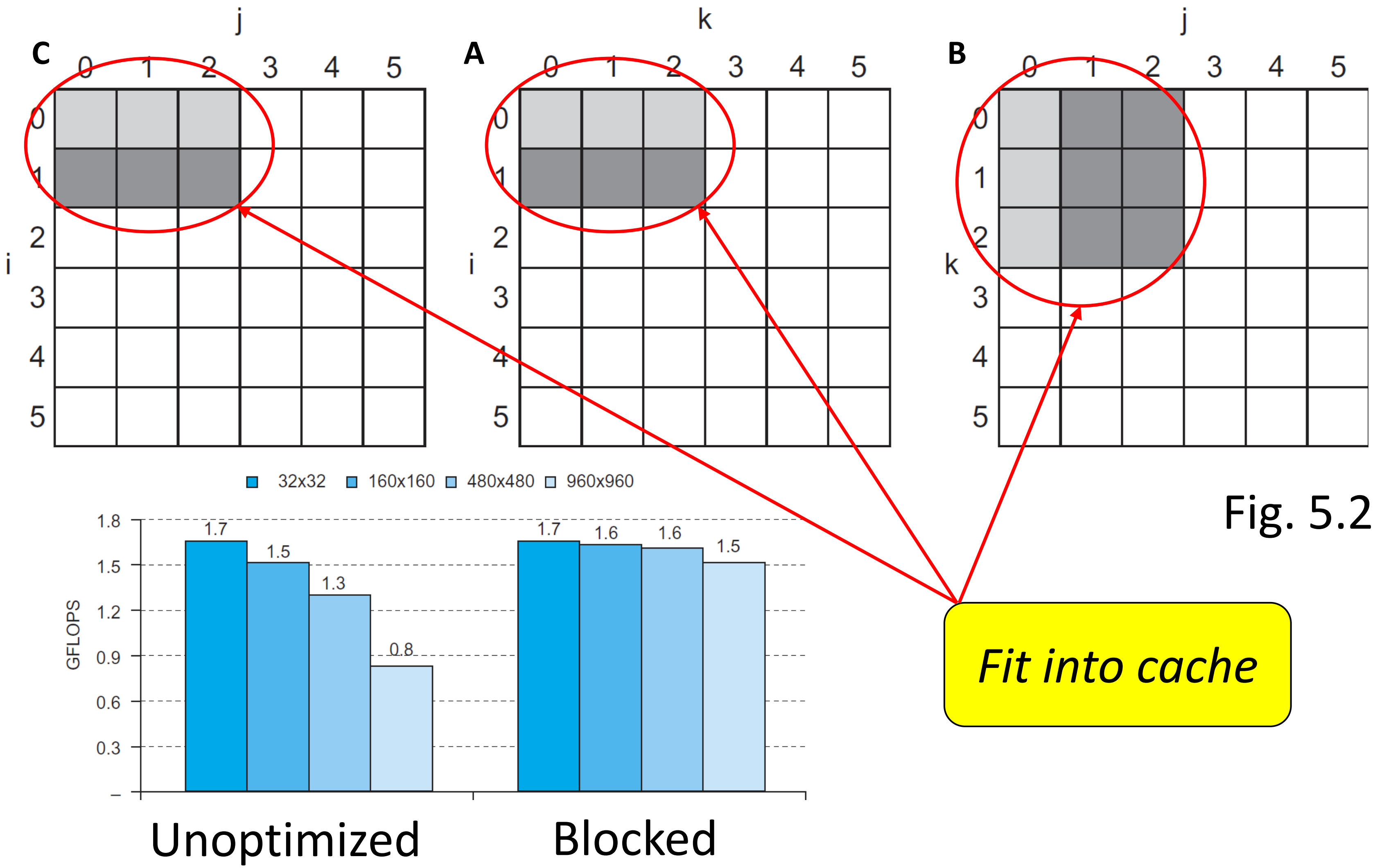


Fig. 5.22

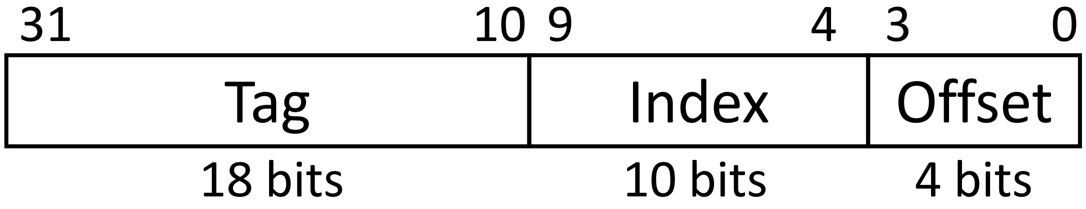
Fit into cache

Outline

- Introduction to memory hierarchy (Sec. 5.1)
- Memory technologies (Sec. 5.2, 5.5)
- Caches (Sec. 5.3, 5.4, 5.9)
  - Basic organization and design alternatives (Sec. 5.3)
  - Performance and design tradeoffs (Sec. 5.4)
  - Cache controller (Sec. 5.9)
- Virtual memory (Sec. 5.7)
- Framework for memory hierarchy (Sec. 5.8)
- Virtual machines (Sec. 5.6)
- Parallelism and memory hierarchies (Sec. 5.10, 5.11)

Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache: CPU waits until access is complete

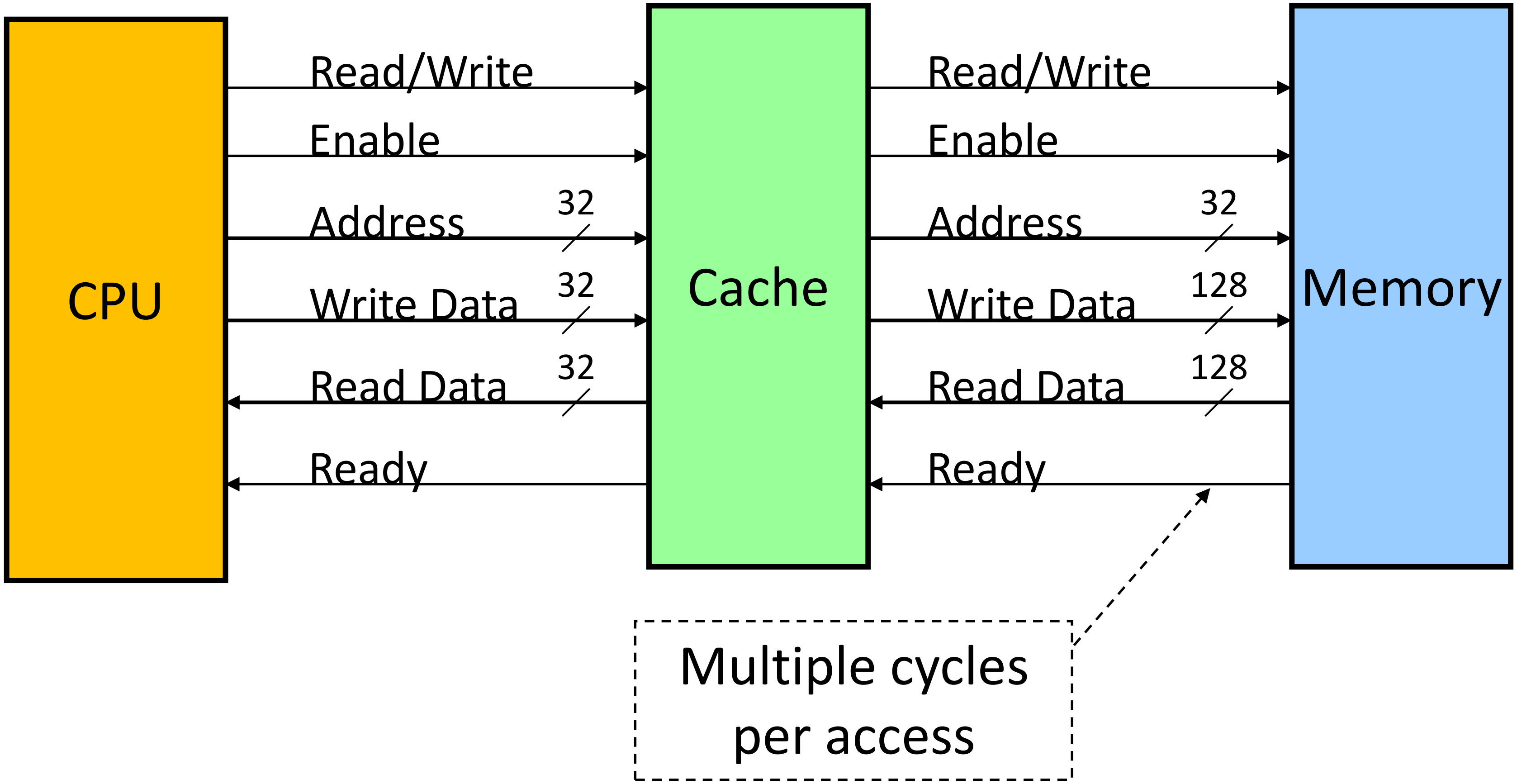


31 10 9 4 3 0

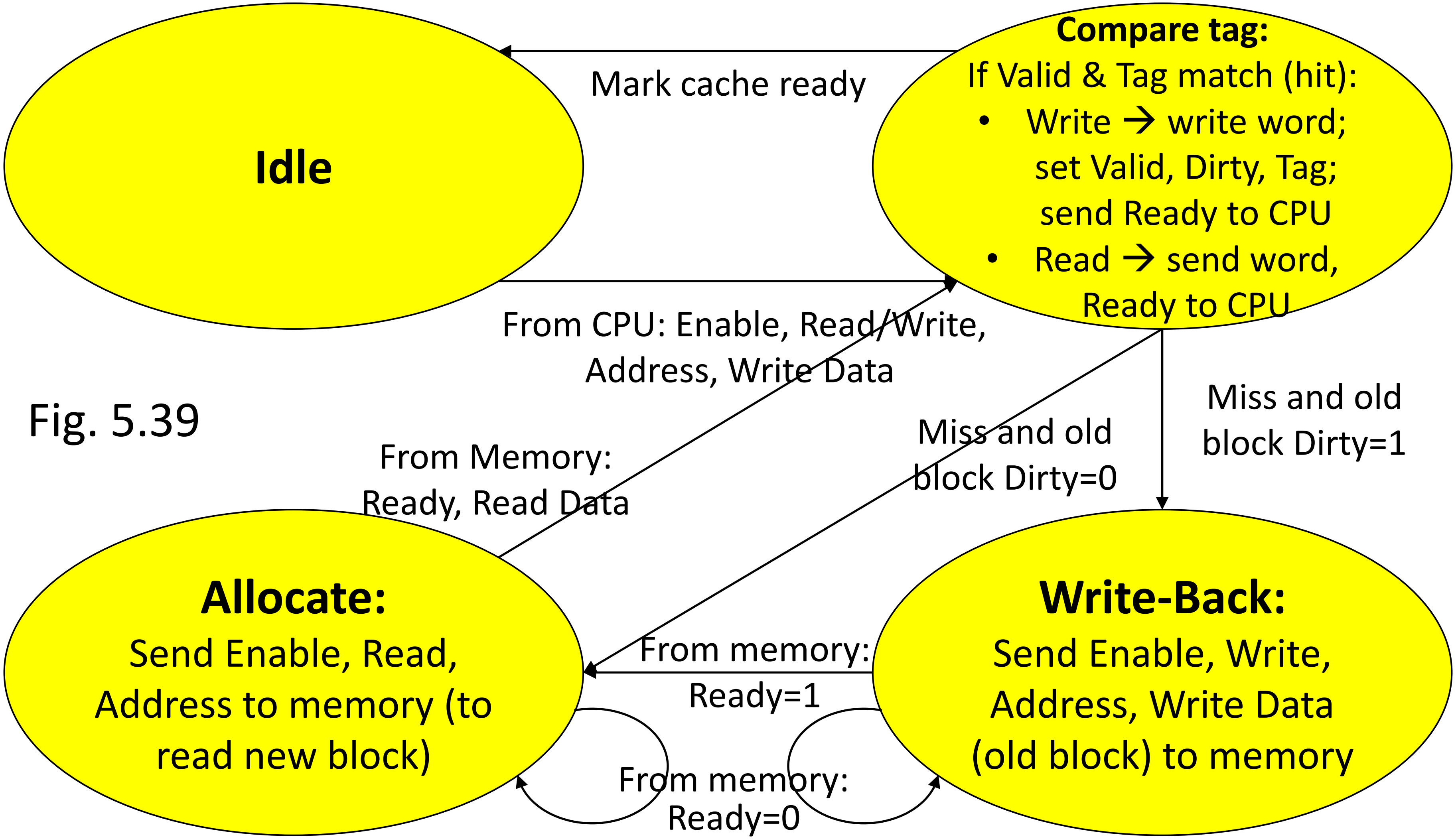
Tag | Index | Offset

18 bits | 10 bits | 4 bits

Interface Signals



FSM of Cache Controller



Idle

Mark cache ready

Compare tag:

If Valid & Tag match (hit):

- Write → write word; set Valid, Dirty, Tag; send Ready to CPU
- Read → send word, Ready to CPU

From CPU: Enable, Read/Write, Address, Write Data

Fig. 5.39

From Memory: Ready, Read Data

Miss and old block Dirty=0

Miss and old block Dirty=1

Allocate:

Send Enable, Read, Address to memory (to read new block)

From memory: Ready=1

Write-Back:

Send Enable, Write, Address, Write Data (old block) to memory

From memory: Ready=0

Finite State Machines

- Use an FSM to sequence control steps
- Set of states, transition on each clock edge
  - State values are binary encoded
  - Current state stored in the *state register*
  - Next state = f_n (current state, current inputs)
  - Control output signals = f_o (current state)

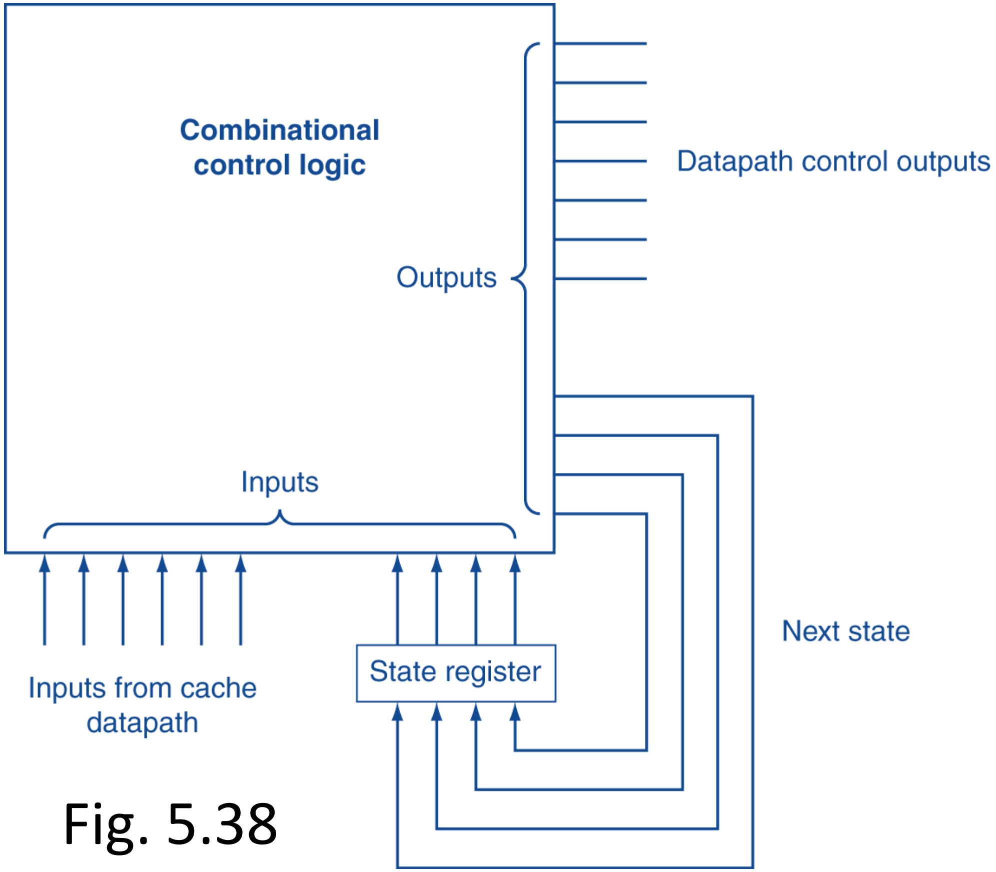


Fig. 5.38

Cache Summary

- Principle of locality:
  - Program likely to access a small portion of address space
    - Temporal locality: locality in time
    - Spatial locality: locality in space
- Three major categories of cache misses:
  - Compulsory: e.g., cold start misses.
  - Conflict: increase cache size or associativity
  - Capacity: increase cache size
- Cache design space
  - Total size, block size, associativity
  - Policies: replacement, write-hit (write-through, write-back), write-miss (write allocate)

Outline

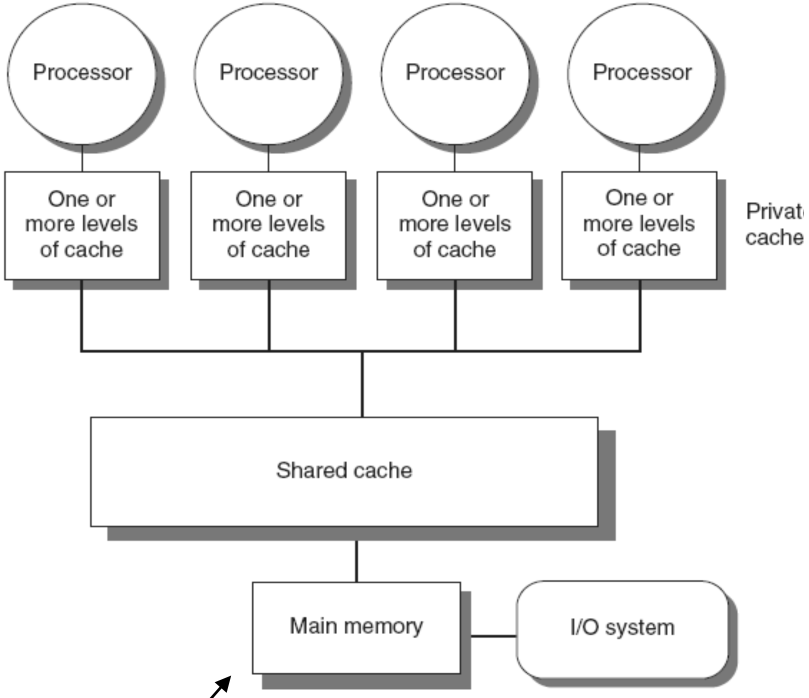
- Introduction to memory hierarchy (Sec. 5.1)
- Memory technologies (Sec. 5.2, 5.5)
- Caches (Sec. 5.3, 5.4, 5.9)
  - Basic organization and design alternatives (Sec. 5.3)
  - Performance and design tradeoffs (Sec. 5.4)
  - Cache controller (Sec. 5.9)
- Virtual memory (Sec. 5.7)
- Framework for memory hierarchy (Sec. 5.8)
- Virtual machines (Sec. 5.6)
- Parallelism and memory hierarchies (Sec. 5.10, 5.11)

Shared-Memory Multiprocessor

- Large multilevel caches can substantially reduce memory bandwidth demands of a processor/core → allow multiple processors to run a (parallel) program, e.g., multi-threaded, with shared memory
  - The multiple cores run different parts of the parallel program collaboratively to solve a single problem, assuming a large shared memory they can all read and write freely
  - Shared memory provides communication among processors through reads and writes of shared data ← Like a blackboard
  - Caches reduce bandwidth for shared memory by migration of data to local caches and reduce contention for memory accesses by replication of read-shared data

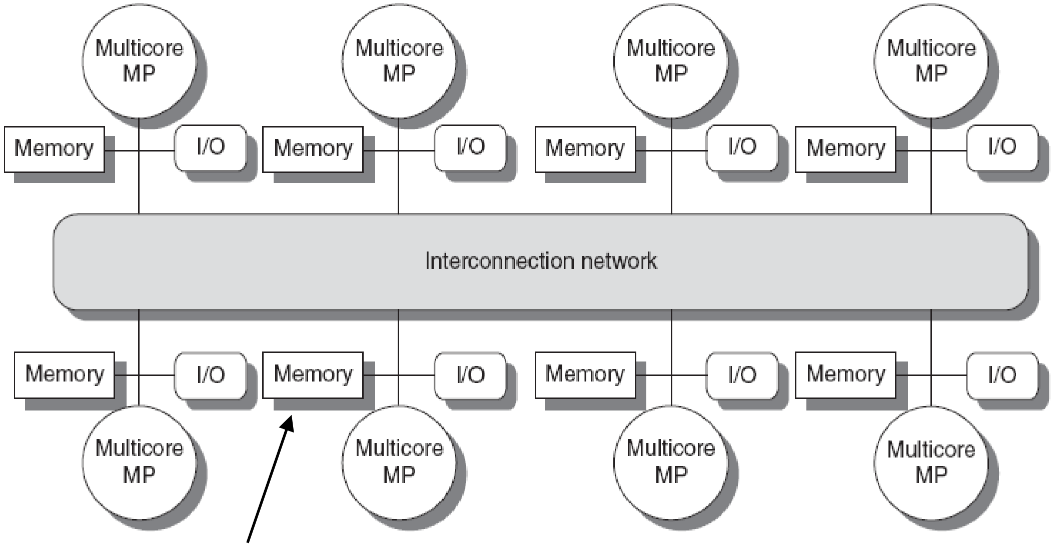
Two Types of Shared-Memory

Uniform memory access (UMA)/
Symmetric multiprocessor (SMP)


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Privat cache
Shared cache
Main memory
I/O system

Physically shared

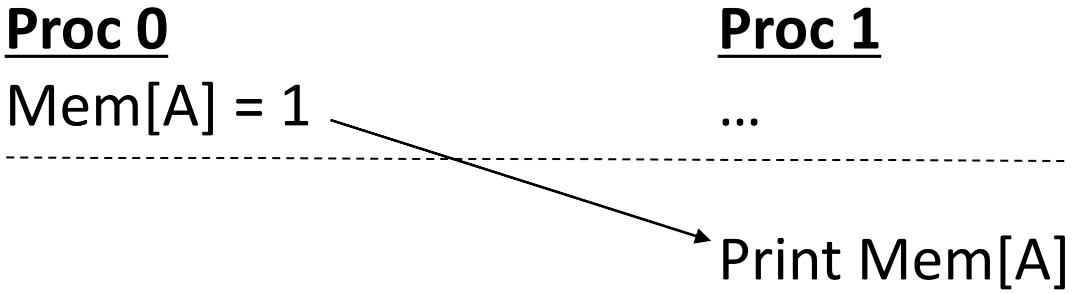
Non-uniform memory access (NUMA)/
Distributed shared memory (DSM)



Logically shared

Shared Memory Programming

- Different cores run different parts (threads) of a parallel program with a view of shared memory
  - P0 writes to an address, followed by P1 reading from it → communication between the two threads/cores



Proc 0

Mem[A] = 1

Proc 1

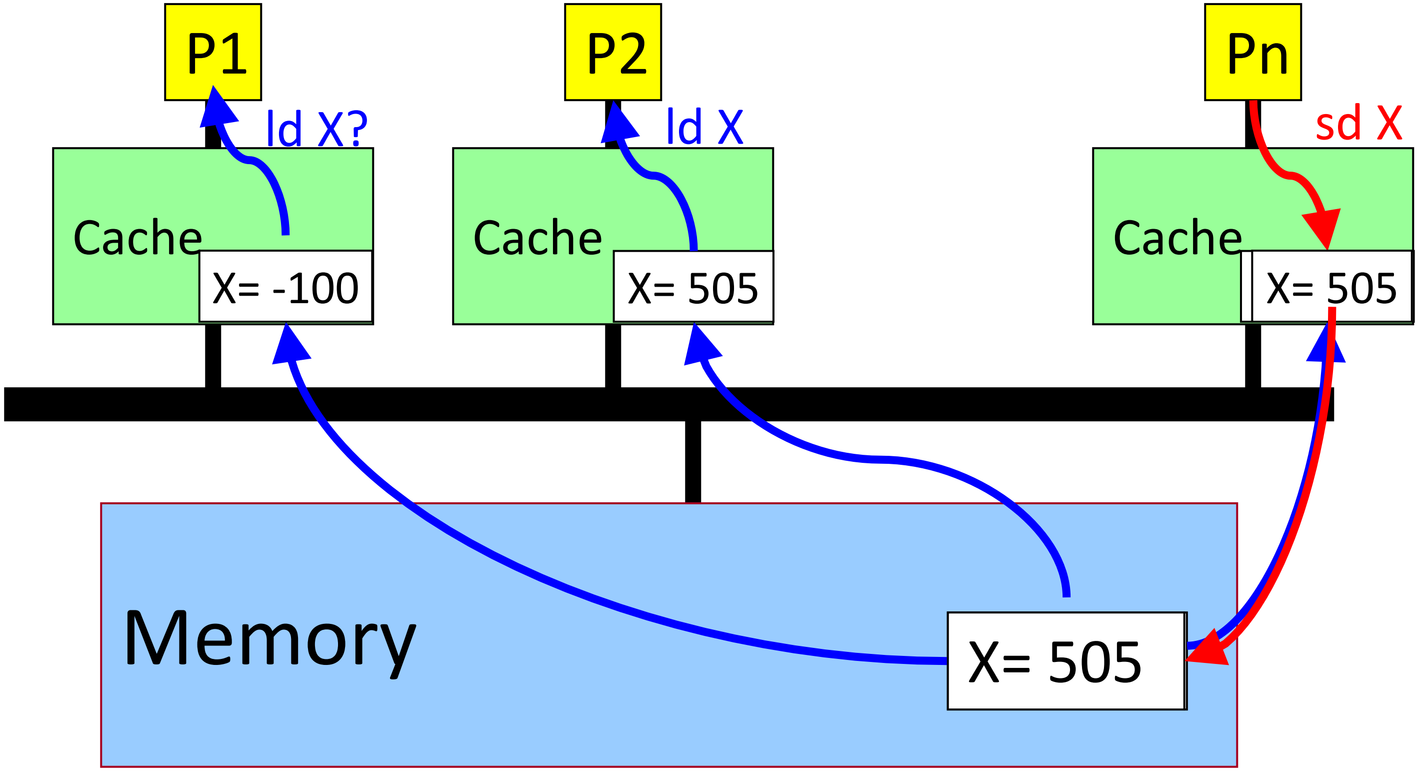
…

Print Mem[A]

- Intuitively, each read should receive the value last written, not only by itself but also by anyone
  - What if Mem[A] is cached (at either end)?
  - What does “last written” mean? → need synchronization

Problem due to Cache for Shared Memory

- Example: write-through cache



Local updates lead to incoherent state → How to ensure all cores see a *coherent state* (value) of memory contents, without explicit synchronization?

Coherence Defined

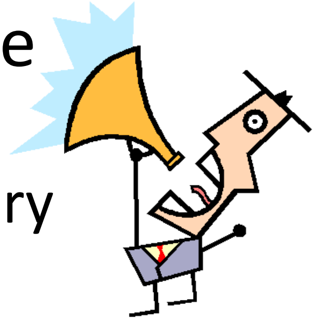
- Programs expect reads to return the most recently written value → too strict to implement; can do:
  - P writes X; P reads X (no intervening writes)
    ⇒ read returns written value
  - P_1 writes X; P_2 reads X (sufficiently later)
    ⇒ read returns written value
  - P_1 writes X; P_2 writes X
    ⇒ all processors see writes in the same order
    - End up with the same final value for X
  - Focusing on each memory location X

Ordering of updates to multiple memory locations requires explicit programming of synchronization

- Caches cause incoherence → what can caches do to satisfy programs' expectation?

Cache Coherence

- Cache ensures everyone always sees latest value
  - Basic strategy: let a write be known by all
  - A processor/cache "broadcasts" its write to a memory location to all other processors/caches
  - Another cache that has a local copy of the memory location either updates or invalidates its local copy
  - Serialization: only one can broadcast at a time

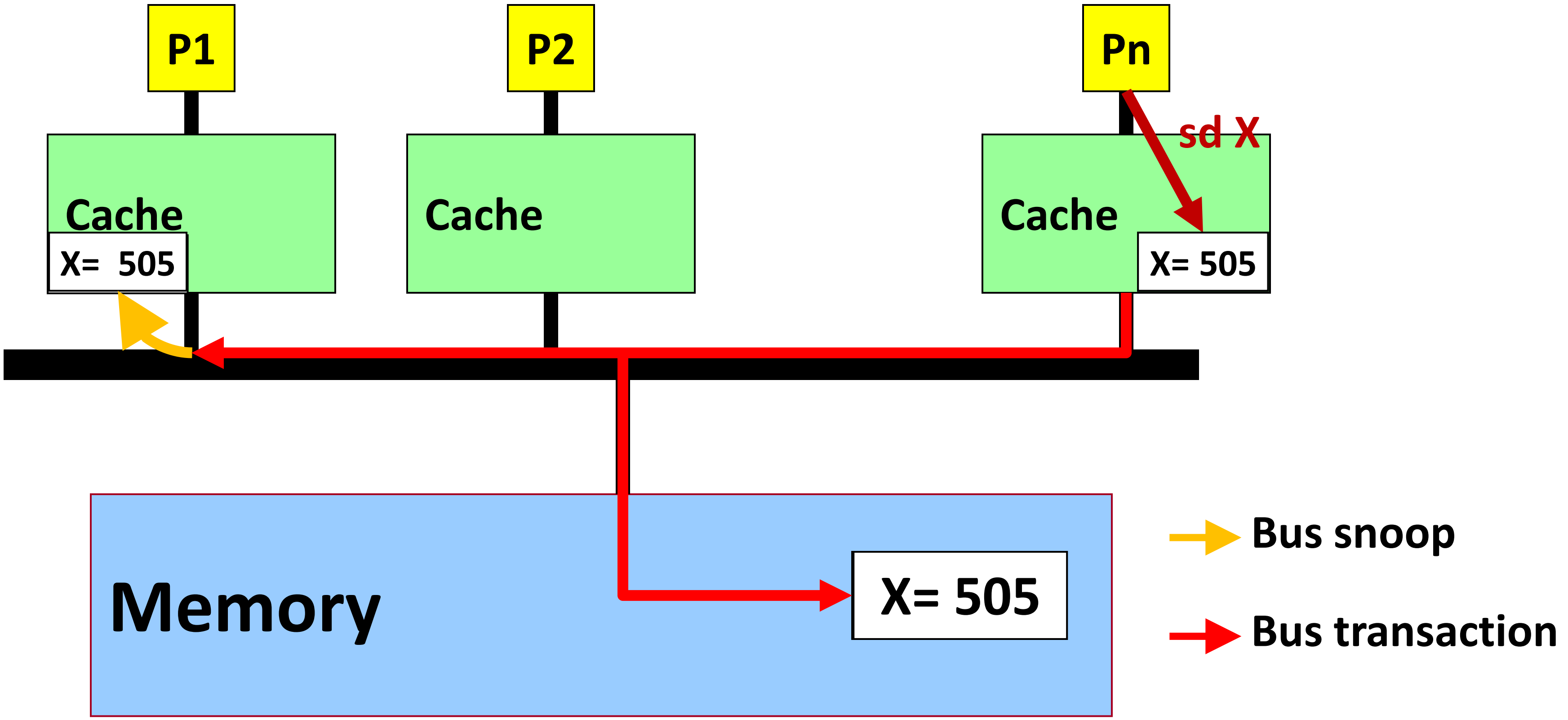


- How can we *safely update replicated cache data*?
  - Option 1 (update protocol): push the update to all copies
  - Option 2 (invalidate protocol): invalidate all others to ensure there is only one valid copy (local), and update it
    - On a read: If local copy isn't valid, request it either from the memory or from the node holding the valid copy

Two Cache Coherence Architectures

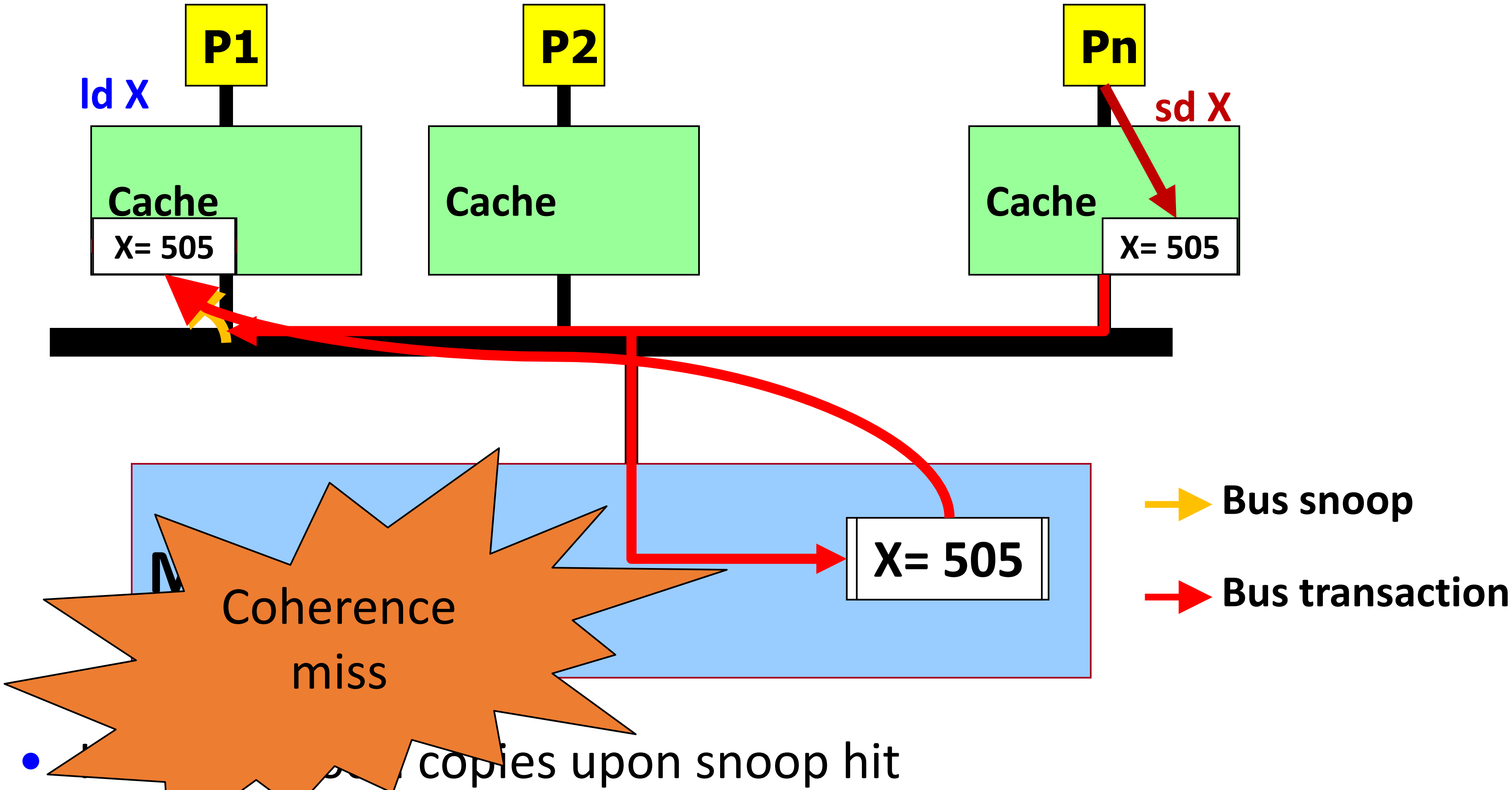
- How to "broadcast" the update?
- Snooping: (you shout at everyone)
  - Bus-based, single point of serialization for all requests
  - Processor broadcasts updates on bus → totally ordered
  - Processors observe ("snoop") other processors' actions
- Directory: (you tell someone to inform all others)
  - *Directory* tracks ownership (sharer set) for each block
  - Processors explicitly request for blocks through directory
  - Directory coordinates invalidation/update appropriately
  - Serves as ordering point for conflicting requests in unordered networks
  - No single broadcast media → more complex but scalable

Bus Snooping
(Update-based Protocol on Write-Through Cache)



- Update local copies upon snoop hit

Bus Snooping
(Invalidation-based Protocol on Write-Through Cache)



- copies upon snoop hit
(but the updated data has already been broadcasted on bus!)

Invalidating Snooping for Write-back

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

Effects of Bus Snooping

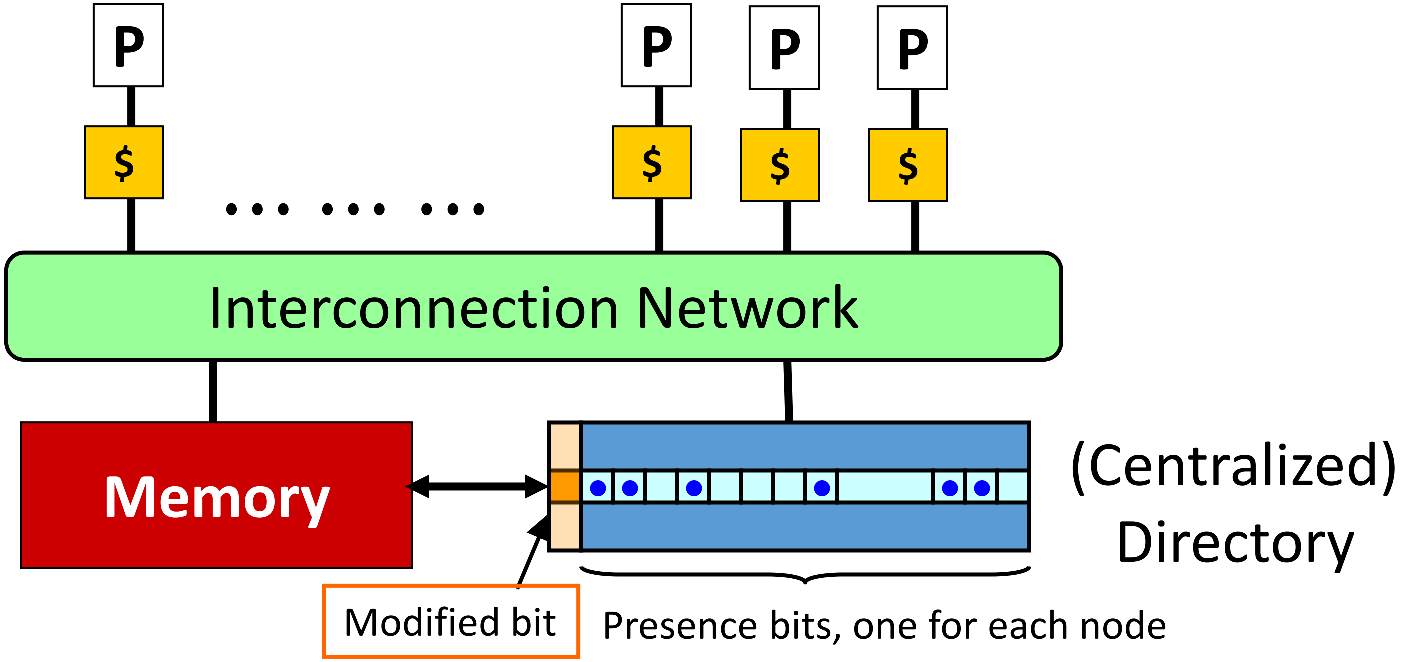
- When a processor writes to its local copy, this fact is broadcasted to all other caches and all caches will see the same update
  → *write propagation*
- When two processors write to their individual local copy of the same data at almost the same time, one processor will win the bus and broadcast its update information to all other caches. The other update must wait until it can grab the bus. This guarantees that all other caches see the same order
  → *write serialization*

Coherency Misses

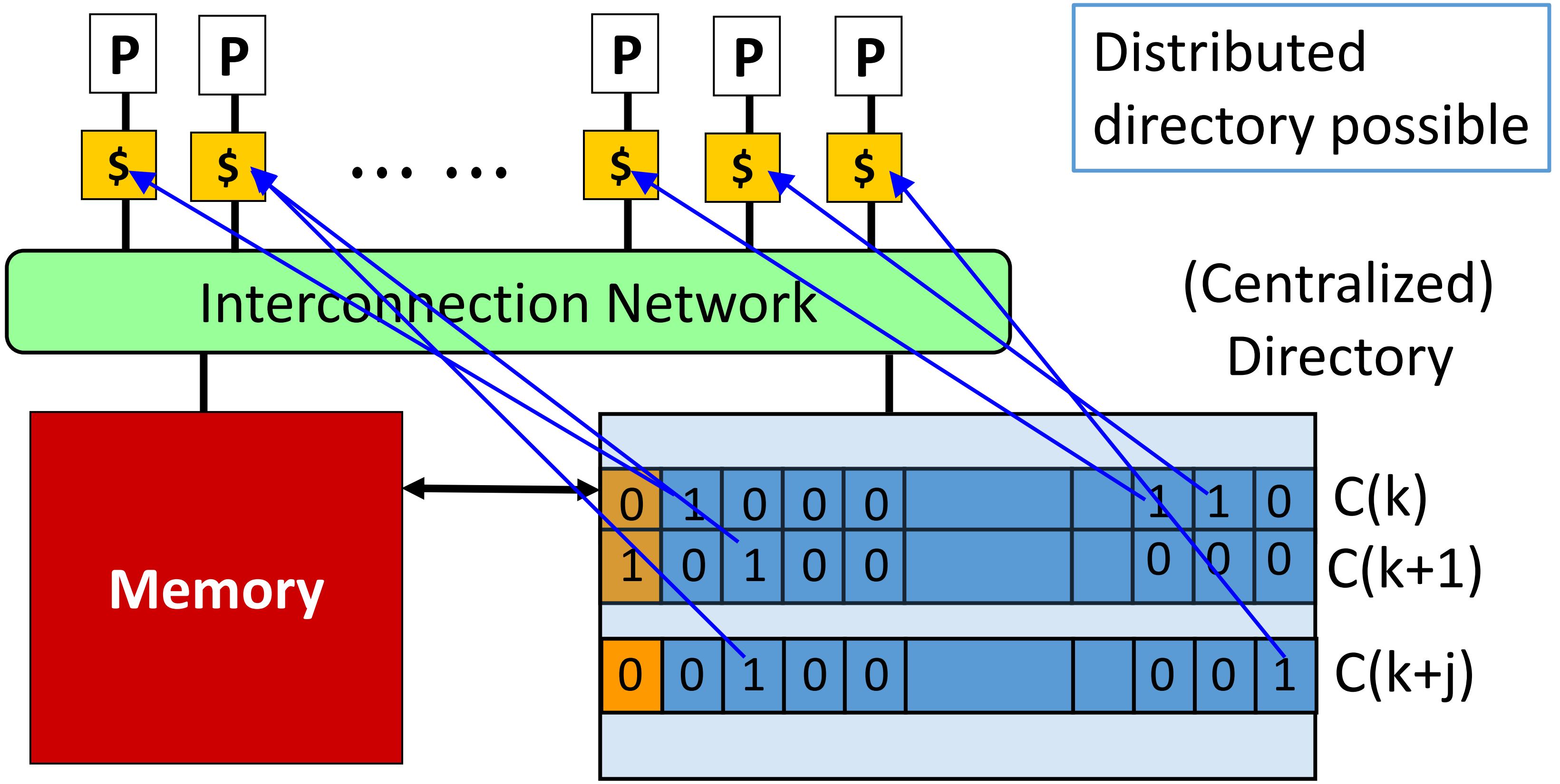
- *True sharing misses* arise from the communication of data through the cache coherence mechanism
  - P1 writes to a word, causing the block in P2 invalidated
  - P2 reads the word, has a cache miss (coherence miss), and receives up-to-date block from P1
  - Miss would still occur if block size were 1 word
- *False sharing misses* arise when accessing a block invalidated because some word in the block, other than the one being read, is written into
  - Invalidation does not cause a value to be communicated
    - But will cause an extra cache miss
  - Miss would not occur if block size were 1 word

Directory-based Cache Coherence

- Directory tracks who has what
- Every memory block has an entry in the directory
  - HW overhead for the directory (~ # blocks * # nodes)
- Can work with any interconnection network!



Directory-based Cache Coherence



Distributed directory possible

🟧 1 *modified bit* for each cache block in memory

🟦 1 *presence bit* for each processor and each cache block in memory