CSEE 3827: Fundamentals of Computer Systems

Caches

Memory Technology: Old prices (2007 / 2008?)

Static RAM (SRAM) → 0.5ns – 2.5ns, $2000 – $5000 per GB

Dynamic RAM (DRAM) → 50ns – 70ns, $20 – $75 per GB

Magnetic disk → 5ms – 20ms, $0.20 – $2 per GB

Ideal memory = access time of SRAM + capacity and cost/GB of disk

Principle of Locality

Programs access a small proportion of their address space at any time

Temporal Locality:

Items accessed recently are likely to be accessed again soon

e.g., instructions in a loop, induction variables

Spatial locality:

Items near those accessed recently are likely to be accessed soon

E.g., sequential instruction access, array data

Taking Advantage of Locality

Organize memory hierarchically

Copy **more recently accessed (and nearby) items** from DRAM to smaller SRAM memory → cache attached to CPU

Copy **recently accessed (and nearby) items** to smaller DRAM memory → main memory

*Store **everything** on disk*

Canonical Memory Hierarchy

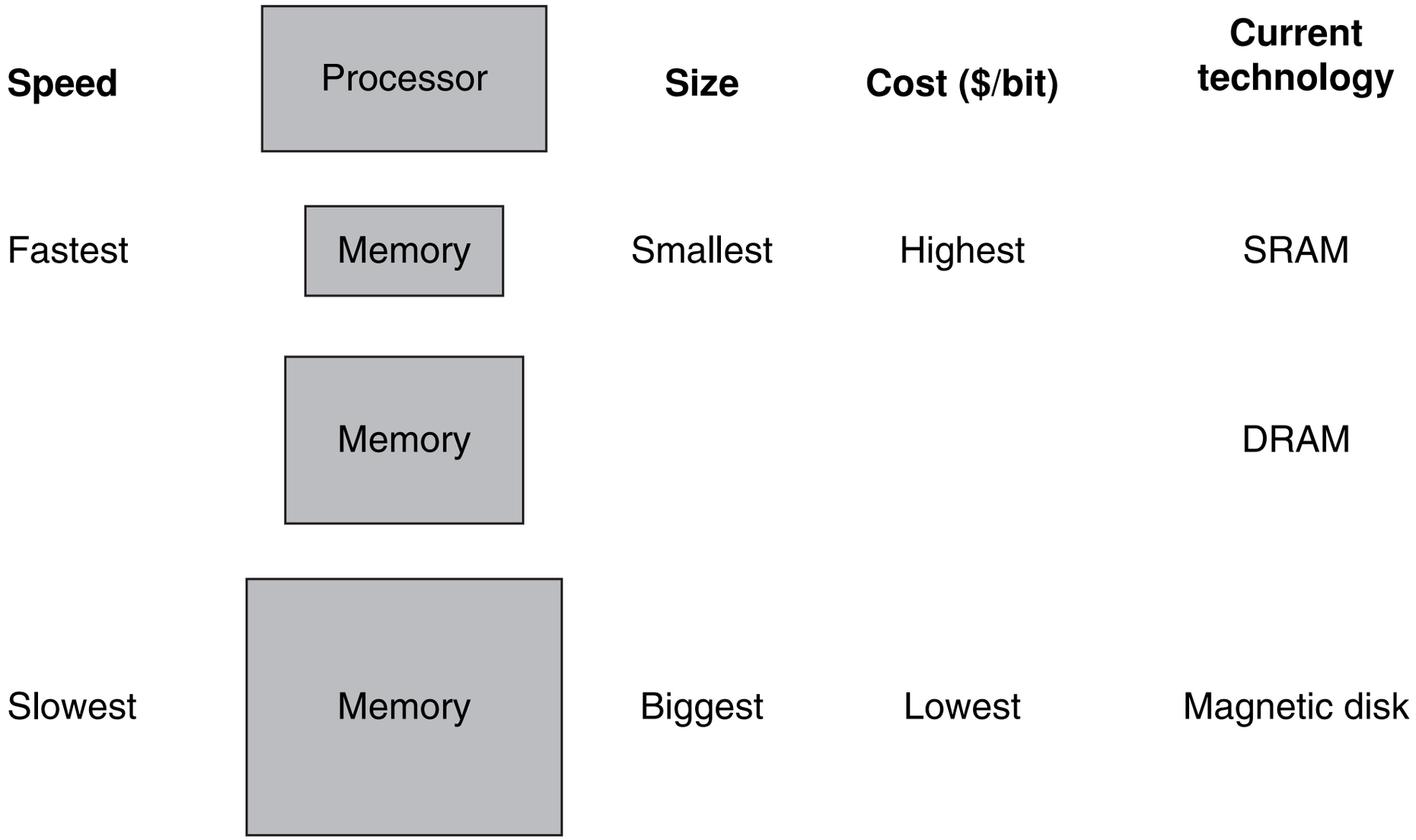
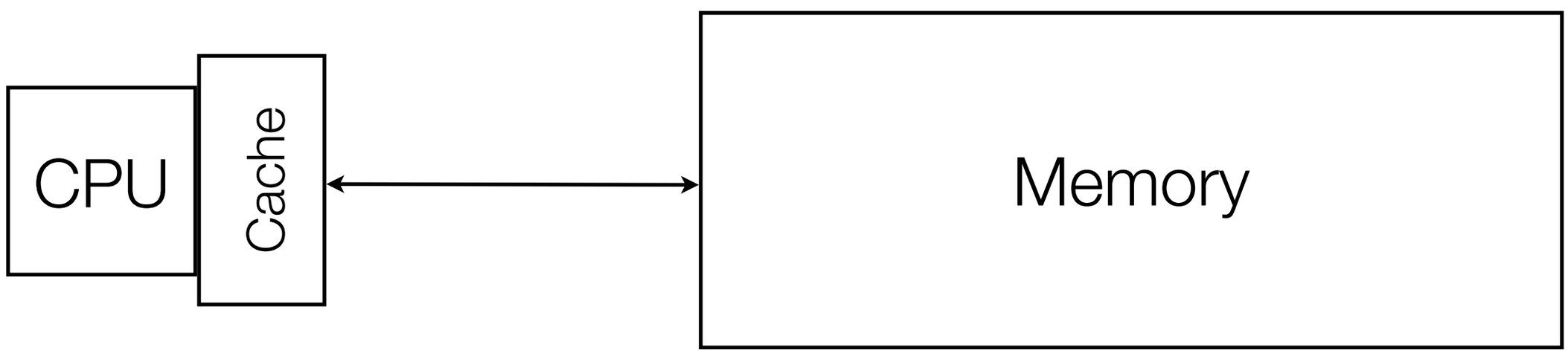


FIGURE 5.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many embedded devices, and may lead to a new level in the storage hierarchy for desktop and server computers; see Section 6.4.

What is a cache?

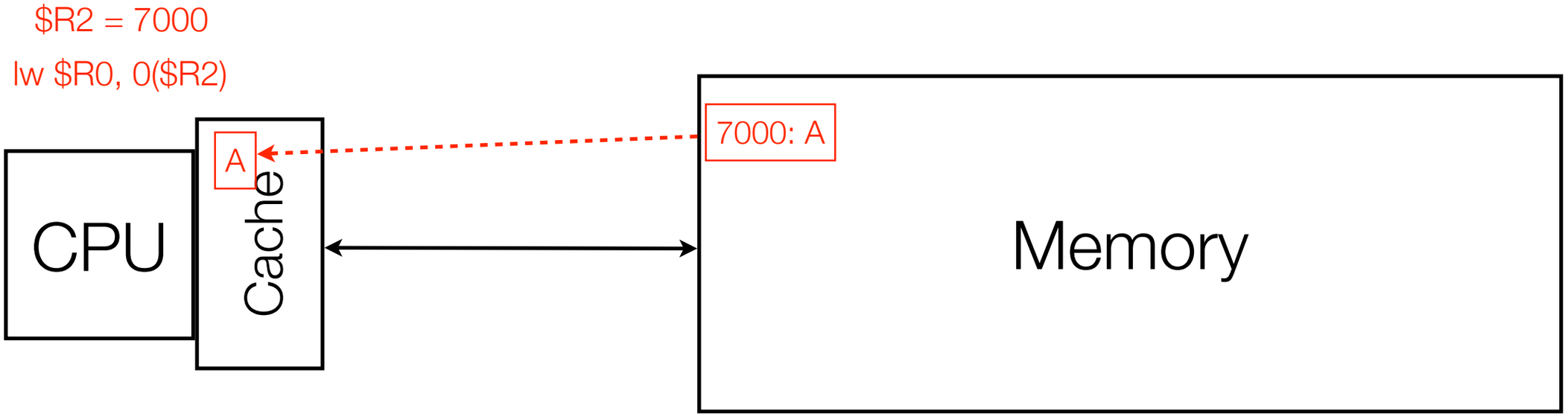
- Temporary copy of memory
  - CPU can read from and write to cache in much less time than memory



- Cache does not have separate addresses
  - CPU (and assembly code) still “thinks” it is accessing main memory
  - Cache acts as a temporary stand-inTemporary copy of memory
  - CPU can read from and write to cache in much less time than memory

What is a cache?

- Temporary copy of memory
  - CPU can read from and write to cache in much less time than memory



- Cache does not have separate addresses
  - CPU (and assembly code) still “thinks” it is accessing main memory
  - Cache acts as a temporary stand-inTemporary copy of memory
  - CPU can read from and write to cache in much less time than memory

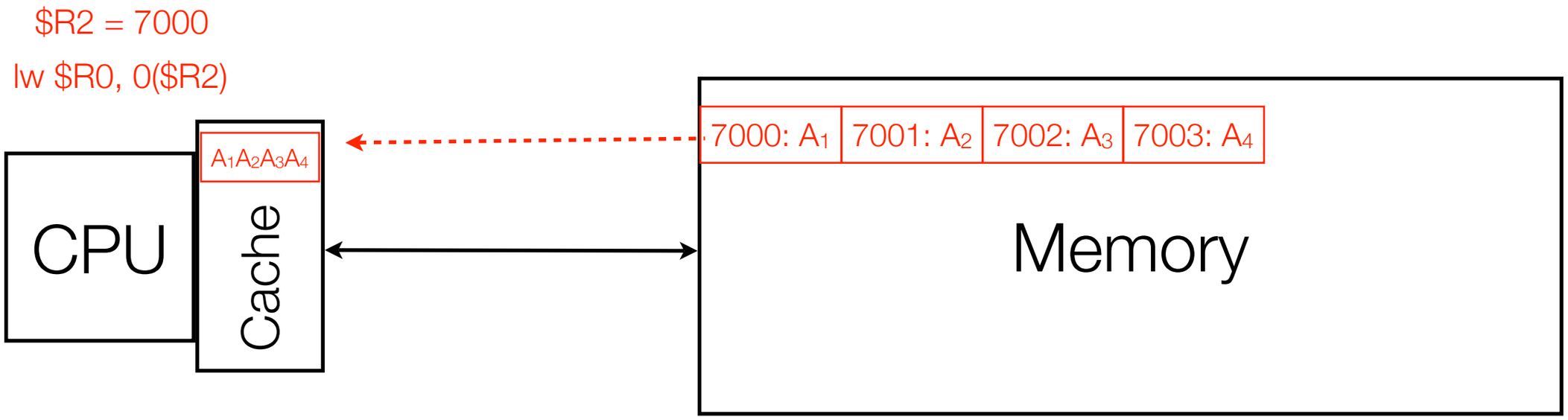
Memory Hierarchy Levels

Block (aka line): unit of copying, may be multiple words

→ If accessed address's data already present in cache
- **Cache Hit:** access satisfied
- **Hit ratio:** hits/accesses

→ If accessed address's data not yet in cache
- **Cache Miss:** block must be copied from memory
- **Miss penalty:** time required to copy the block from the cache
- **Miss ratio:** misses/accesses

e.g., 4 word block



For k-word blocks, a cache miss results in a transfer of k words at once

Handling Cache Misses

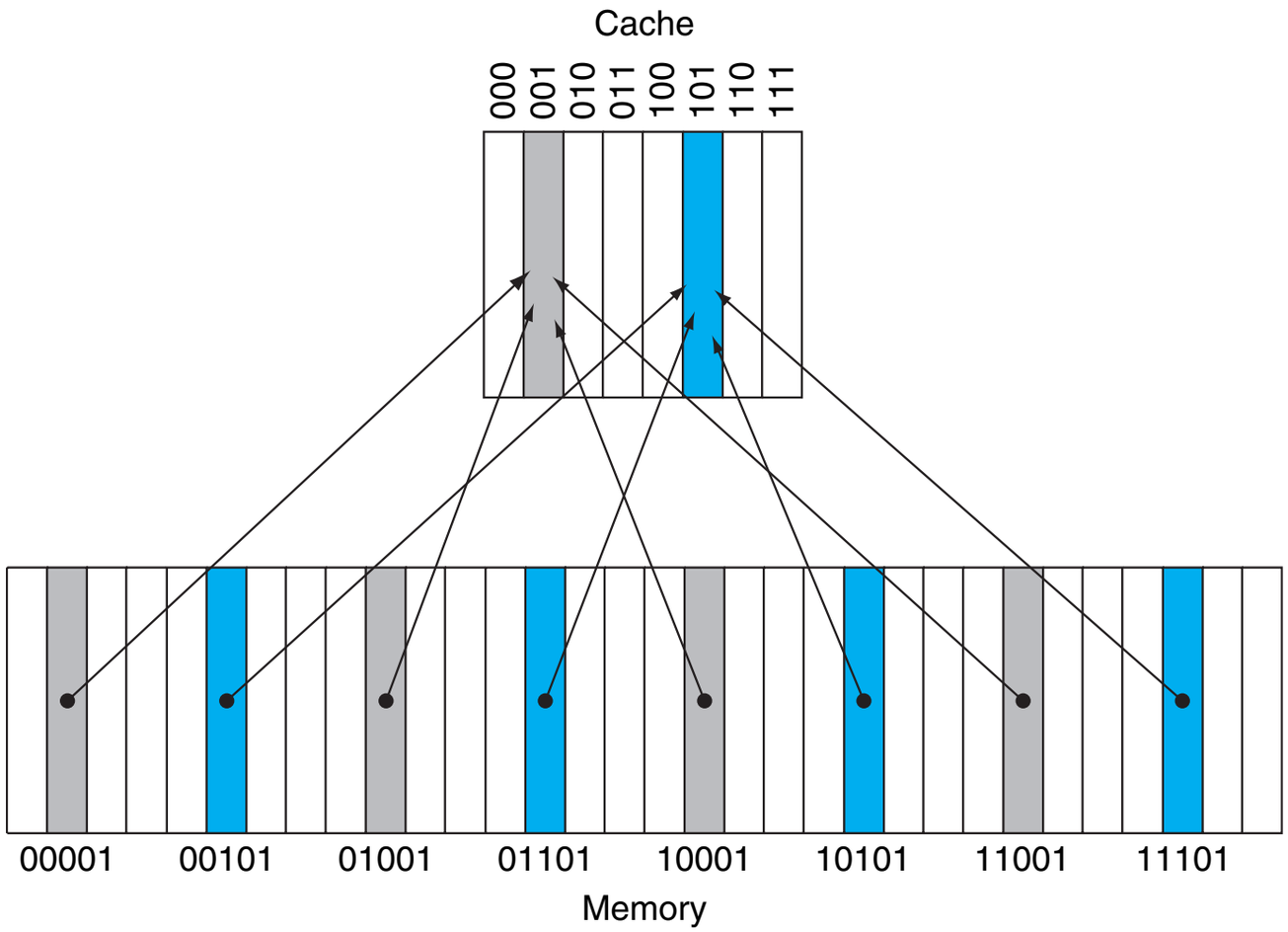
- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - After Instruction cache miss, restart instruction fetch
  - After data cache miss, complete data access

Cache Challenges

- **When** to move data from memory to cache?
    - Small blocksize?
        - Less “wasted” moves
        - Don’t take advantage of parallelism (many words movable at same time)
- **Where** in cache to move fetched memory?
    - Lots of options: least recently used, fixed mapping, will explore pros and cons
- Challenge: minimize cache misses (maximize cache hits)

Direct Mapped Cache

- Location determined by address (in memory): easy to locate data in cache
- Direct mapped: only one choice, e.g., (Block address) modulo (#Blocks in cache)
- Drawback: may overwrite some parts of cache while other parts are empty



If a power of two, use low order address bits

Several memory addresses would map to the same location in the cache

FIGURE 5.5 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the direct-mapped cache word X modulo 8. That is, the low-order $\log_2(8) = 3$ bits are used as the cache index. Thus, addresses 00001_{two}, 01001_{two}, 10001_{two}, and 11001_{two} all map to entry 001_{two} of the cache, while addresses 00101_{two}, 01101_{two}, 10101_{two}, and 11101_{two} all map to entry 101_{two} of the cache.

Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
    - 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?
    - Valid bit: 1 = present, 0 = not present
    - Initially 0

Address Subdivision

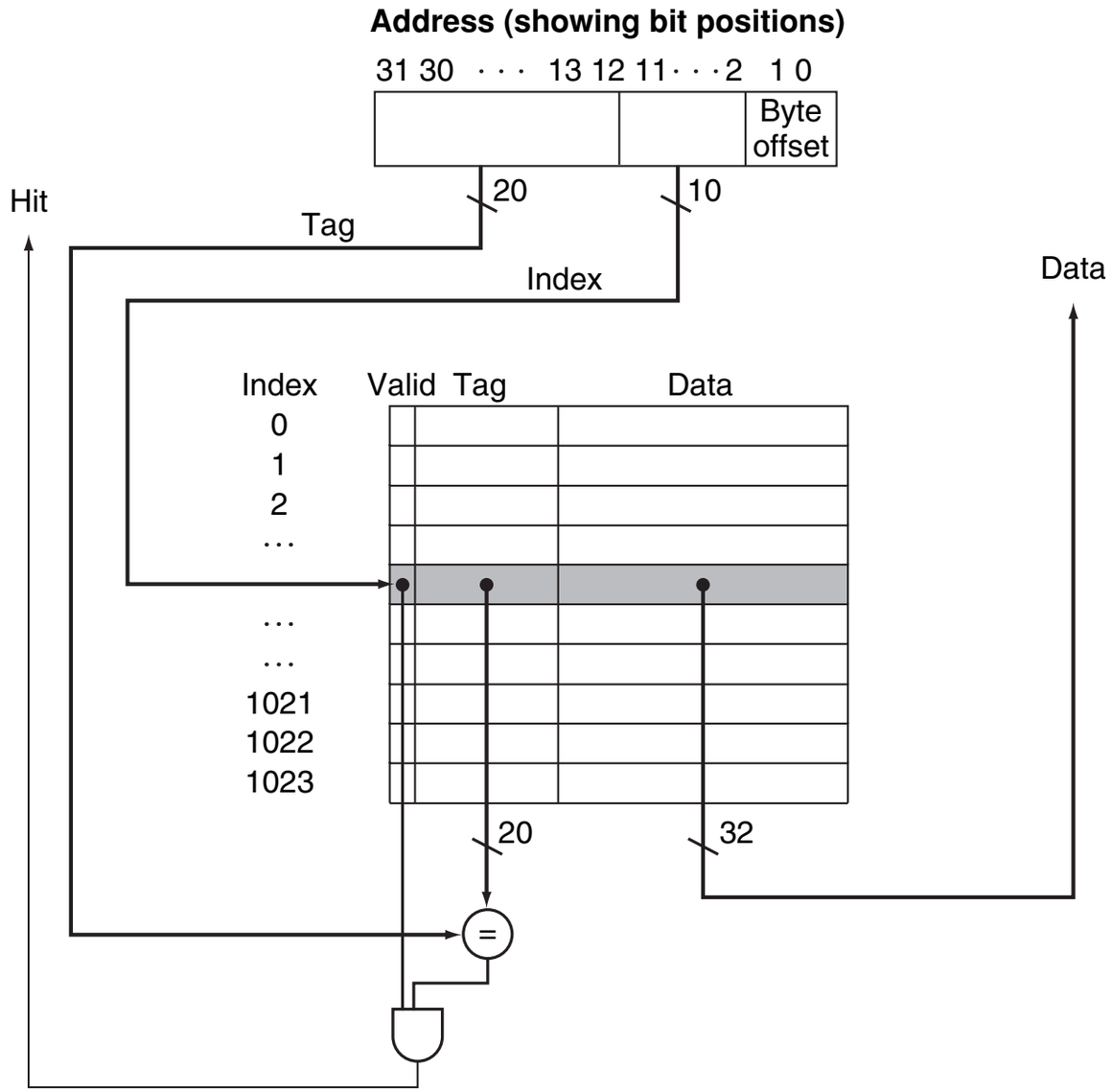


FIGURE 5.7 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. This cache holds 1024 words or 4 KB. We assume 32-bit addresses in this chapter. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 2^{10} (or 1024) words and a block size of one word, 10 bits are used to index the cache, leaving $32 - 10 - 2 = 20$ bits to be compared against the tag. If the tag and upper 20 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

Cache Example 1

8-block cache, 1 word/block, 32B memory

Index	V	Tag	Data
0 0 0	0		
0 0 1	0		
0 1 0	0		
0 1 1	0		
1 0 0	0		
1 0 1	0		
1 1 0	0		
1 1 1	0		

Initial state after power on

Index indicates where addresses ending with specific 3-bit sequence should be stored in cache

Cache Example 2

Index	V	Tag	Data
0 0 0	0		
0 0 1	0		
0 1 0	0		
0 1 1	0		
1 0 0	0		
1 0 1	0		
1 1 0	1	1 0	Mem[10110]
1 1 1	0		

After handling miss of address 10110

Cache Example 3

Index	V	Tag	Data
0 0 0	0		
0 0 1	0		
0 1 0	1	1 1	Mem[11010]
0 1 1	0		
1 0 0	0		
1 0 1	0		
1 1 0	1	1 0	Mem[10110]
1 1 1	0		

After handling miss of address 11010

Cache Example 4

Index	V	Tag	Data
0 0 0	1	1 0	Mem[10000]
0 0 1	0		
0 1 0	1	1 1	Mem[11010]
0 1 1	0		
1 0 0	0		
1 0 1	0		
1 1 0	1	1 0	Mem[10110]
1 1 1	0		

After handling miss of address 10000

Cache Example 5

Index
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

V	Tag	Data
1	1 0	Mem[10000]
0		
1	1 1	Mem[11010]
1	0 0	Mem[00011]
0		
0		
1	1 0	Mem[10110]
0		

After handling miss of address 00011

Cache Example 6

Index
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

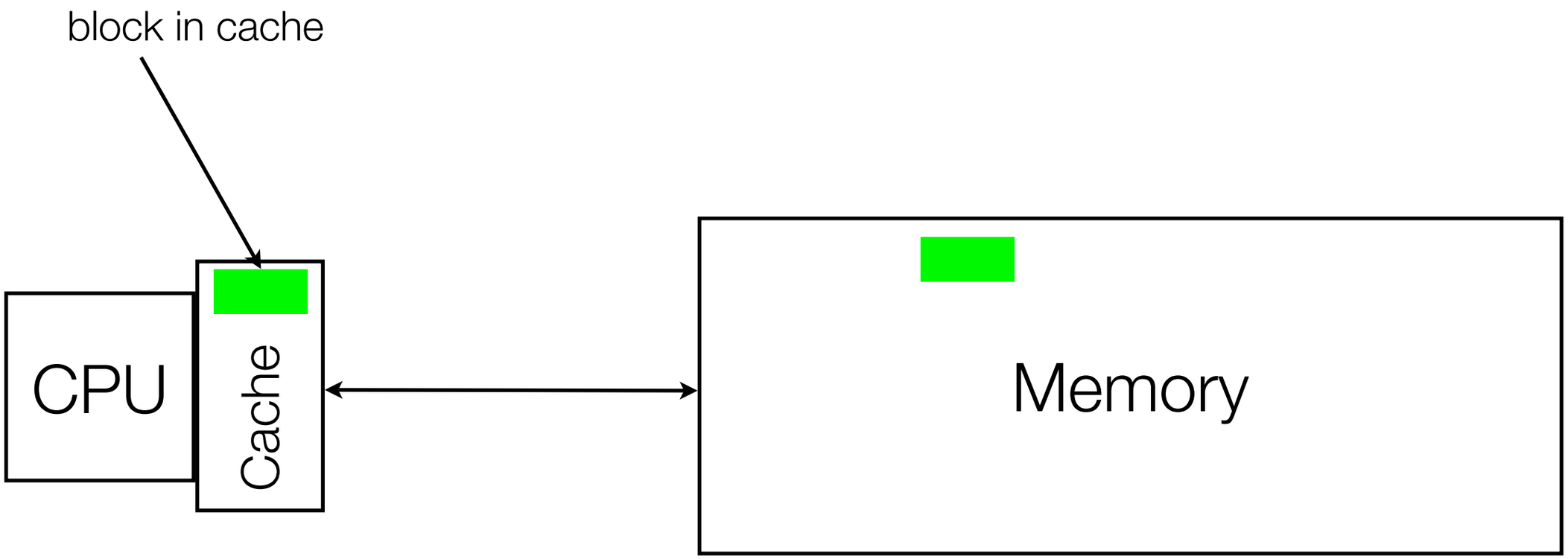
V	Tag	Data
1	1 0	Mem[10000]
0		
1	1 0	Mem[10010]
1	0 0	Mem[00011]
0		
0		
1	1 0	Mem[10110]
0		

After handling miss of address 10010
Cache Collision! Overwrite! (Note tag change)

Direct Mapping and Moving blocks

- Recall: All transfers between memory and cache are done in units of (fixed-size multiple-word) blocks

- Request of address not in cache: entire block must be moved into cache

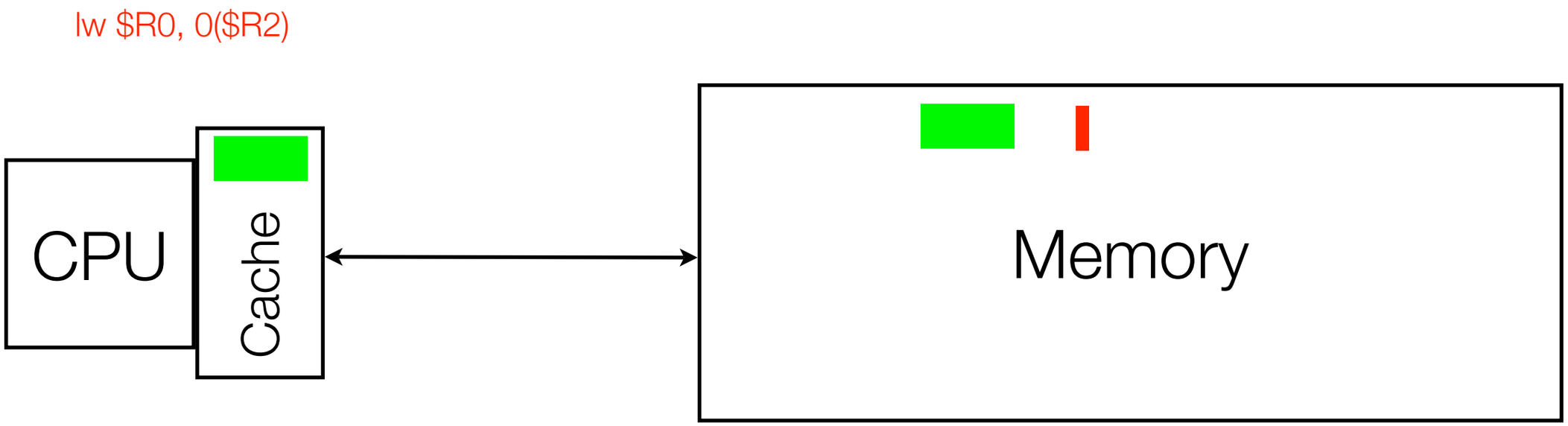


Direct Mapping and Moving blocks

- Recall: All transfers between memory and cache are done in units of (fixed-size multiple-word) blocks

- Request of address not in cache: entire block must be moved into cache

word of memory requested (not in cache)



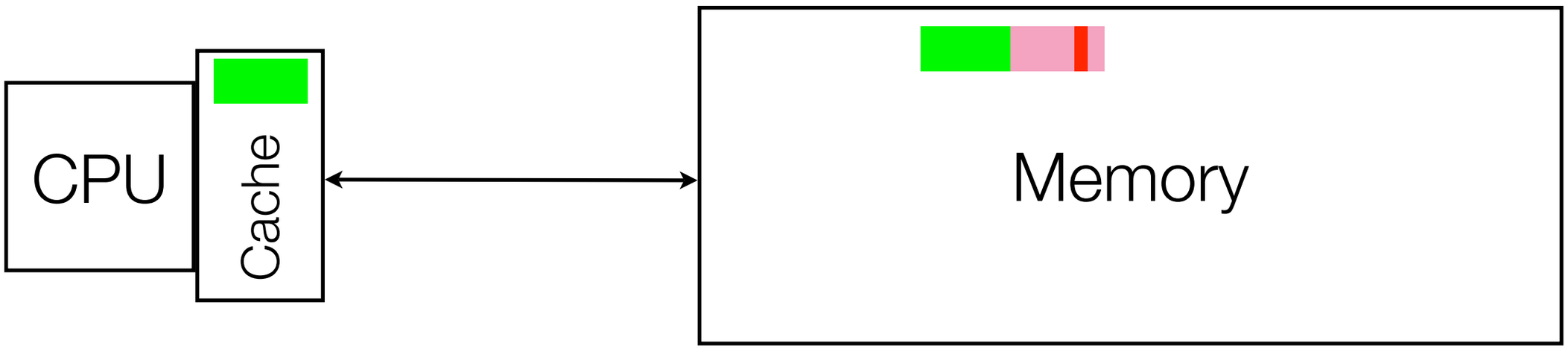
Direct Mapping and Moving blocks

- Recall: All transfers between memory and cache are done in units of (fixed-size multiple-word) blocks

- Request of address not in cache: entire block must be moved into cache

word of memory requested (not in cache)
Entire block containing word must be moved (previous entry overwritten)

lw $R0, 0($R2)

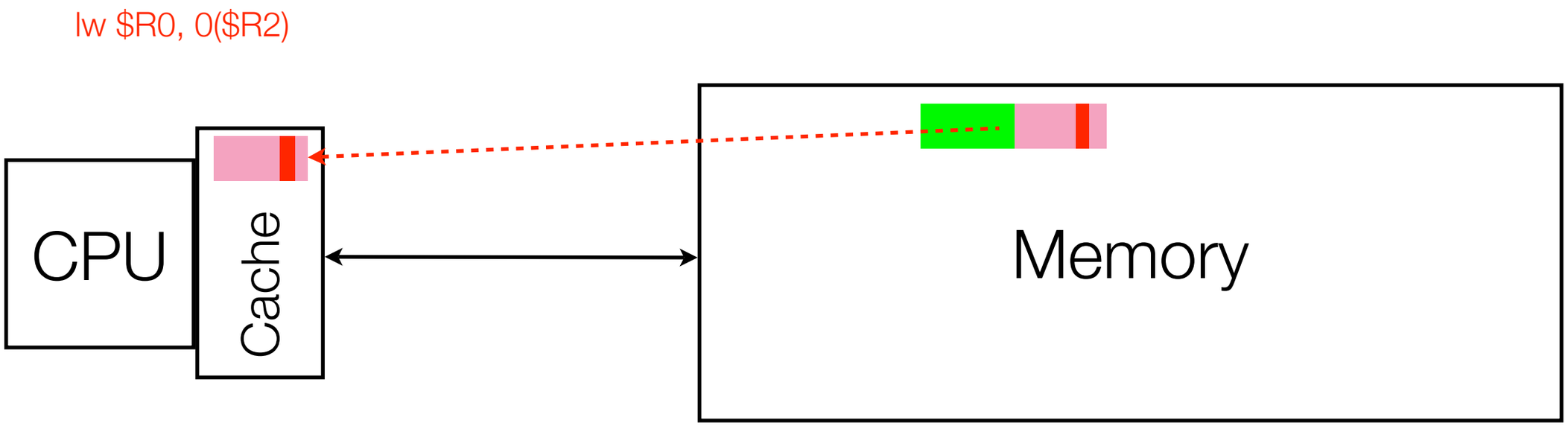


Direct Mapping and Moving blocks

- Recall: All transfers between memory and cache are done in units of (fixed-size multiple-word) blocks

- Request of address not in cache: entire block must be moved into cache

word of memory requested (not in cache)
Entire block containing word must be moved (previous entry overwritten)



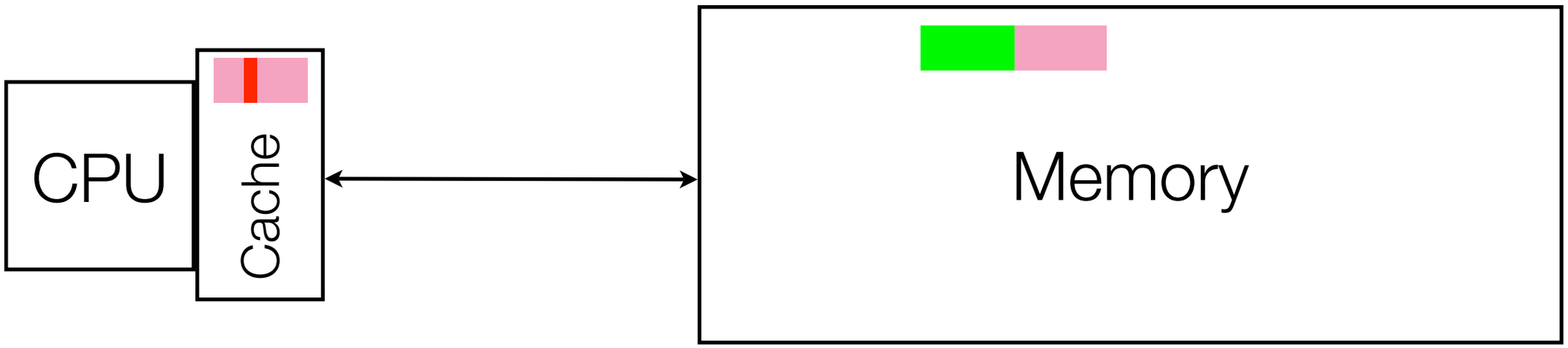
Direct Mapping and Moving blocks

- Recall: All transfers between memory and cache are done in units of (fixed-size multiple-word) blocks

- Request of address not in cache: entire block must be moved into cache

Other words in cached block can be accessed without (miss) penalty

lw $R0, 32($R2)

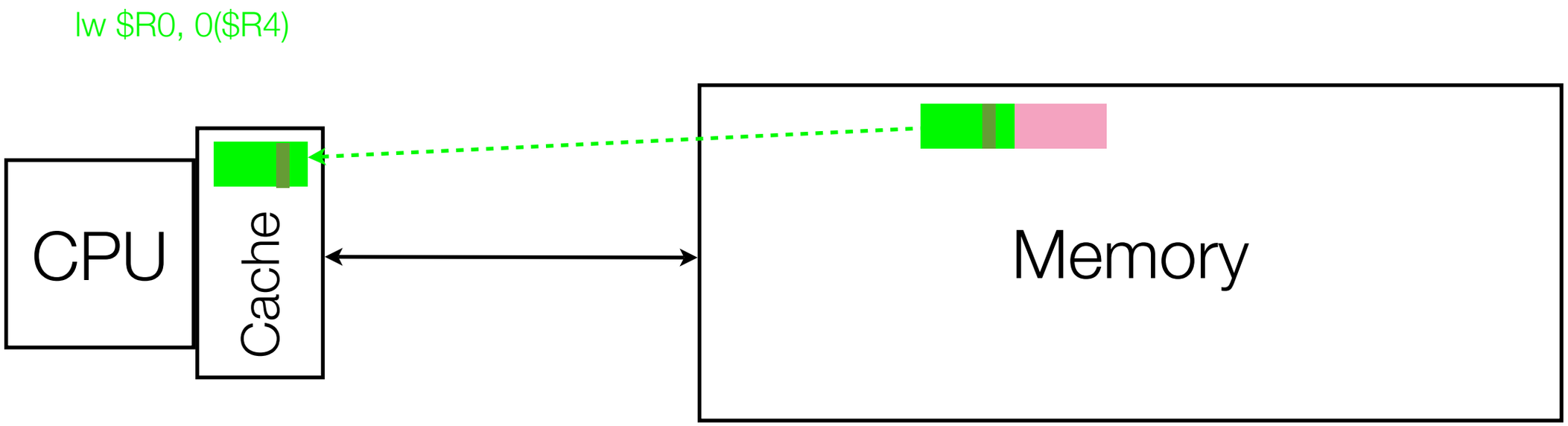


Direct Mapping and Moving blocks

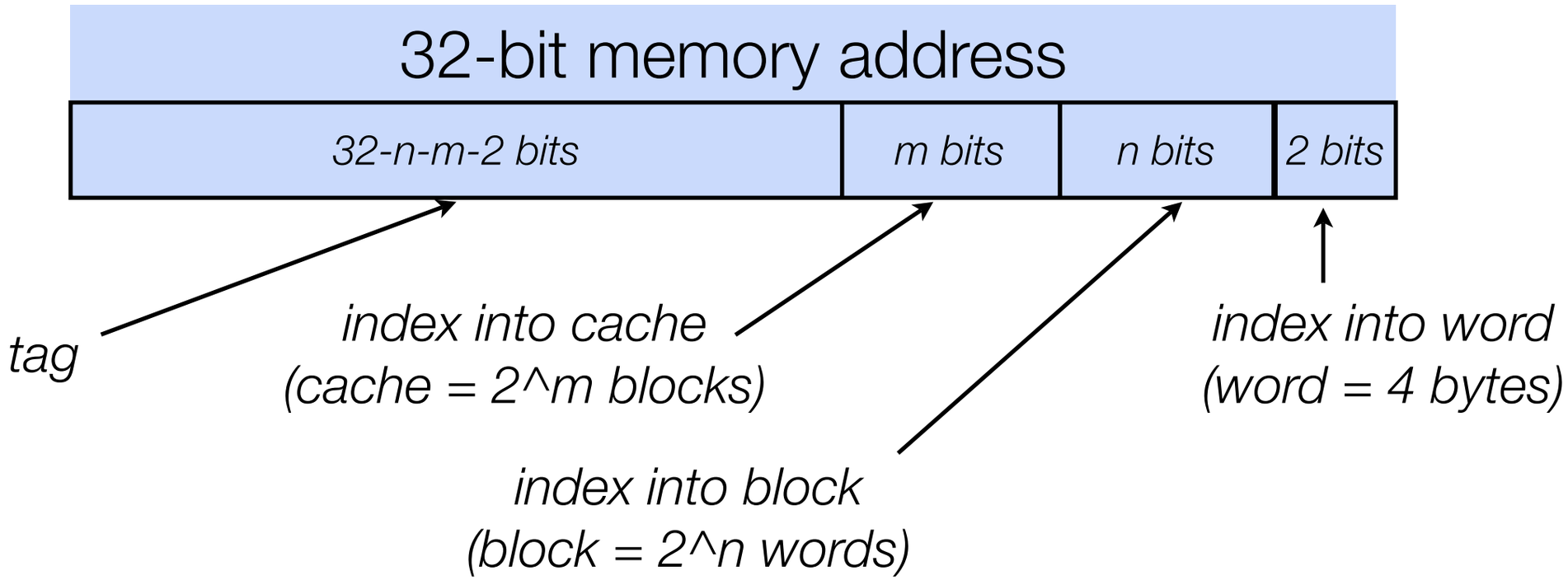
- Recall: All transfers between memory and cache are done in units of (fixed-size multiple-word) blocks

- Request of address not in cache: entire block must be moved into cache

Word in earlier (but evicted) block requested, must be re-cached (miss)

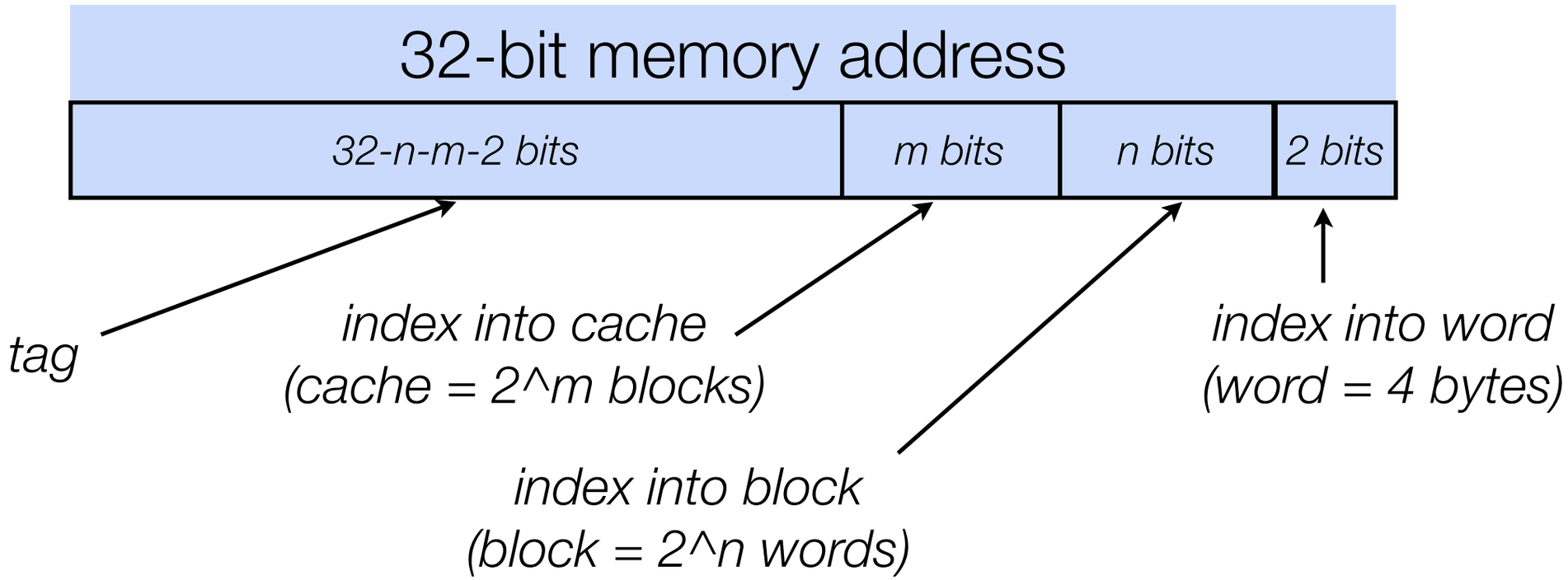


Multi-word Cache Blocks (Direct Mapping)



- e.g., m=5, n=4:
  - 11011000101010100110101 11001 1010 01:
    - byte #1 of 10th word in 25th block
  - 10010110110110110110110 11001 1110 00:
    - byte #0 of 14th word in 25th block
  - Note: both of these addresses cannot be stored in the direct-mapped cache simultaneously

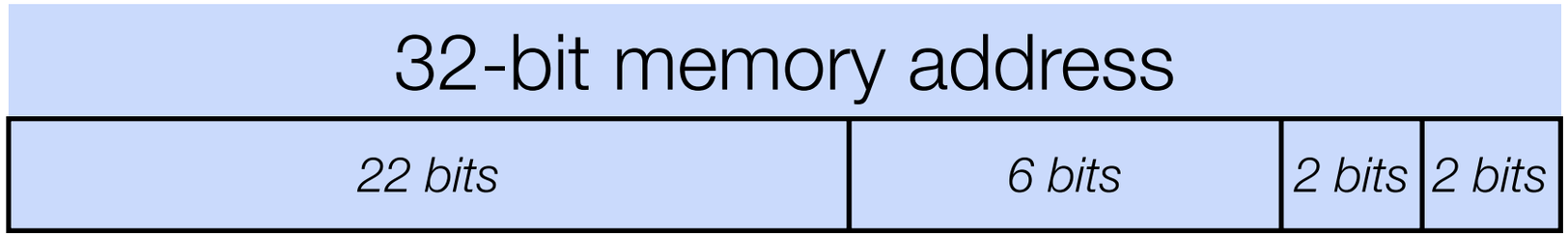
Multi-word Cache Blocks (Direct Mapping)



- e.g., m=5, n=4 (16 words per block, 32 blocks in cache: cache stores 32*16 words)
  - 11011000101010100110101 11001 1010 01:
    - byte #1 of 10th word in 25th block
  - All words whose address is prefixed with 11011000101010100110101 11001 moved into the 25th block of the cache simultaneously

Example: Larger Block Size

- 64 blocks, 16 bytes/block
    - To what block number does address 1200 map?
- Block address = floor(1200/16) = 75 (75th block in memory)
- Block number = 75 modulo 64 = 11 (Direct mapping, would map to 11th block in cache)

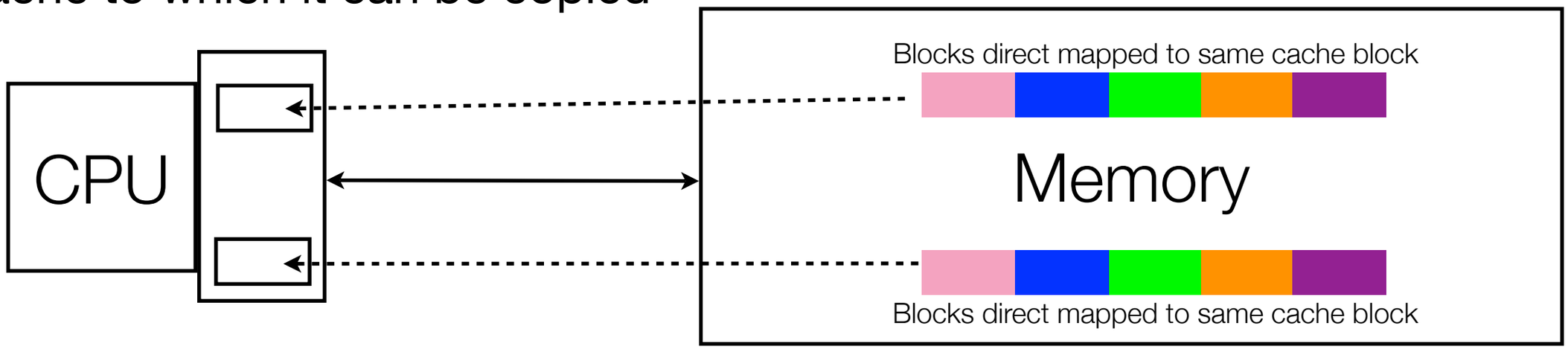


A Note on blocks

- Each memory location A maps to a block B
  - e.g., block size of 10, memory location 1099112 is in the 109911st block
- Each block of memory B maps to a block of cache B’
  - e.g., block 109911 of memory maps to block 11 of the cache
- To find a particular memory address A in the cache (or see if its there)
  - Step 1: Determine its block in memory (A → B)
  - Step 2: Look for that block in the cache (B → B’)
- When talking about caching policies, we can just look at the 2nd mapping (B → B’) since the first mapping is just a mapping within memory

Associative Caches

- Recall: **Direct-Mapped** - each block of memory has a single location in the cache to which it can be copied



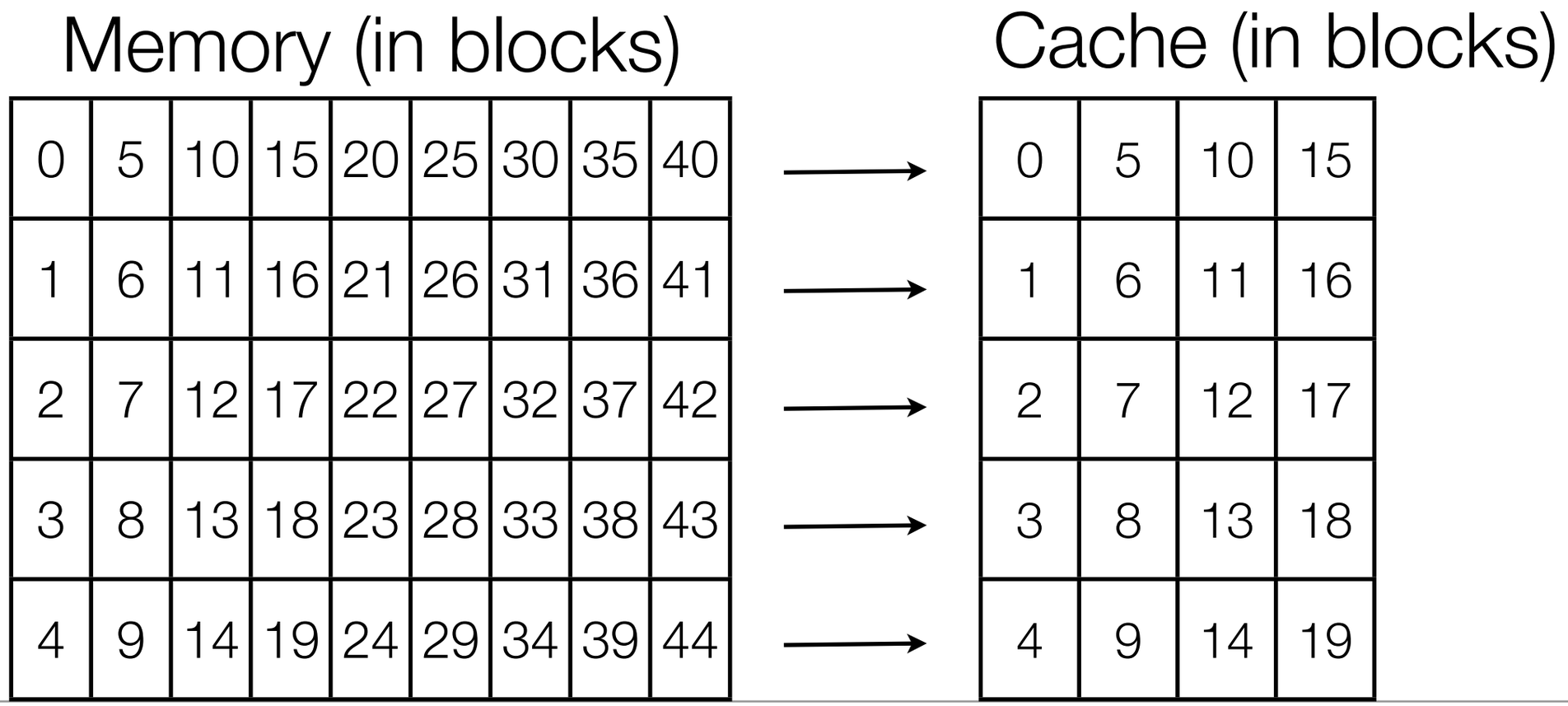
- **Fully Associative**: any block can go anywhere in cache
  - Memory still transferred in units of blocks
  - Where to put a newly retrieved block in cache?
    - Empty block (when one exists), then least recently accessed, or least frequently accessed
- Fully Associative Pros: less likely to overwrite recently accessed memory locations
- Fully Associative Cons: time needed to determine / locate a block of memory in the cache

Set Associative Caching

- Tradeoff of Fully Associative and Direct Mapped
  - n-way Associative: memory block can be stored in n locations in cache
- Given an M-block memory, an m-block cache that is n-way associative:
  - n=1: Direct mapped
  - n=m: Fully Associative
  - In general, M n / m blocks “share” cache locations
- Simple mapping scheme: memory block i can map to blocks i mod (m/n) + km/n for k=0,1,2,..., n-1
- e.g., m=20, n=4
  - each mem block 0, 5, 10, 15, 20, 25, ... maps to any of cache blocks 0, 5, 10, 15
  - each mem block 1, 6, 11, 16, 21, 26, ... maps to any of cache blocks 1, 6, 11, 16
  - ...
  - each mem block 4, 9, 14, 19, 24, 29, ... maps to any of cache blocks 4, 9, 14, 19

Set Associative Caching

- e.g., m=20, n=4: block i can map to blocks i mod 5 + 5k for k=0,1,2,..., 3
  - each mem block 0, 5, 10, 15, 20, 25, ... maps to any of cache blocks 0, 5, 10, 15
  - each mem block 1, 6, 11, 16, 21, 26, ... maps to any of cache blocks 1, 6, 11, 16
  - ...
  - each mem block 4, 9, 14, 19, 24, 29, ... maps to any of cache blocks 4, 9, 14, 19



Block Size Considerations

- Larger blocks should reduce miss rate, due to spatial locality
- But in a fixed-sized cache
  - Larger blocks → fewer of them → more competition → increased miss rate
  - Larger blocks → pollution
- Larger miss penalty, which could override benefit of reduced miss rate

Handling Writes: Write Through

- On data-write hit, could just update the block in cache, but then cache and memory would be inconsistent
    - Problem: Can’t overwrite block in cache until made consistent
- **Write through**: on write, update memory as well as cache
    - 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 which holds data waiting to be written to memory. CPU can now continue immediately, stalling only if write buffer is full.

Handling Writes: Write Back

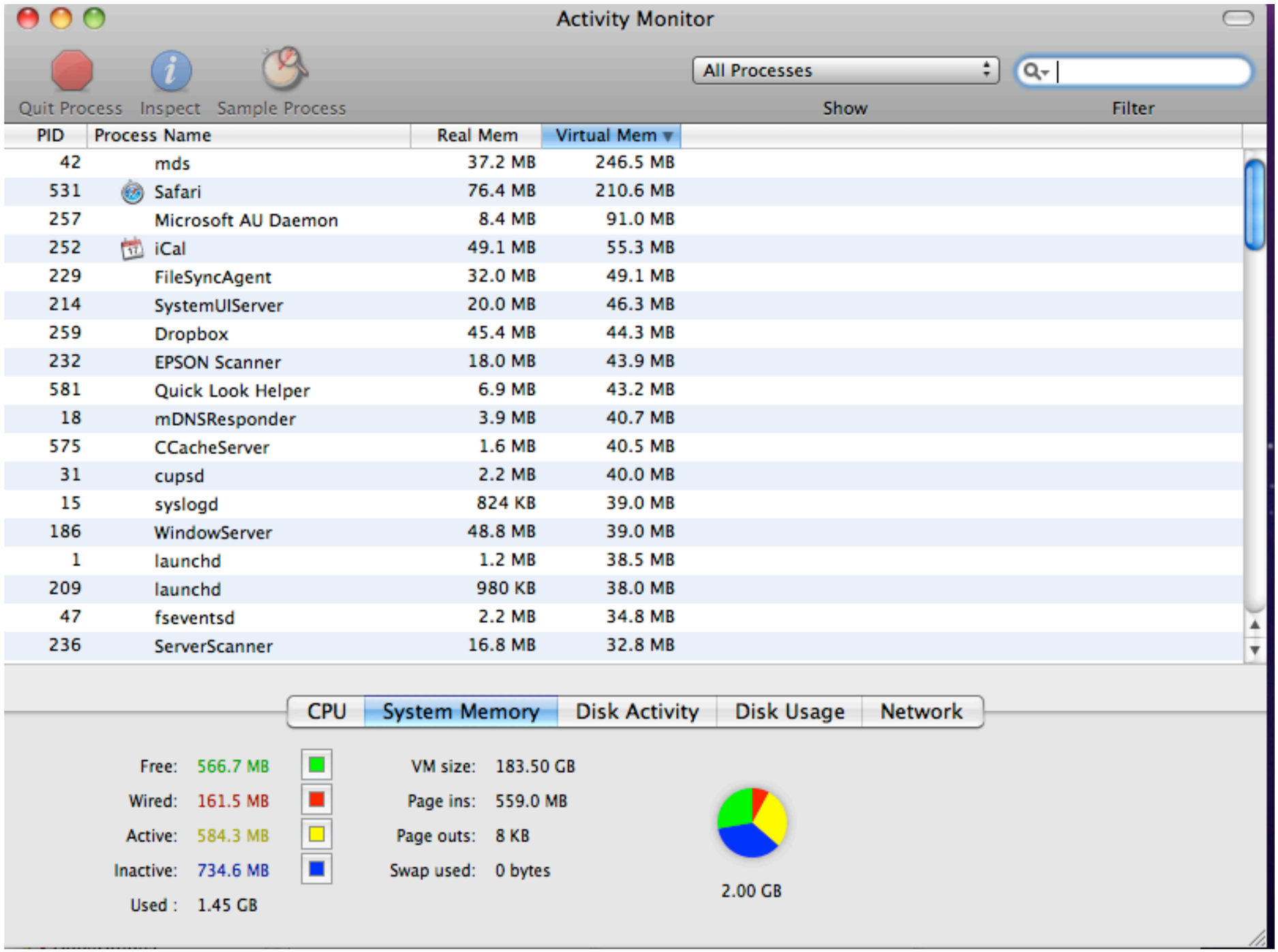
- An alternative to write through
- **Write Back**: on data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
  - When a dirty block is to be replaced, only then write it back to memory
  - Can use a write buffer to allow replacing block to be read first

Virtual Memory

- The “reverse” of cache
- All programs running don’t “fit” into memory
- Idea: use disk to store parts of memory used by program while they are not being run
- When part of program needs to access memory currently stored on disk, transfer from disk to mem.

Virtual Mem in practice

- Mac's Activity Monitor



Virtual Mem in practice

- Mac's Activity Monitor