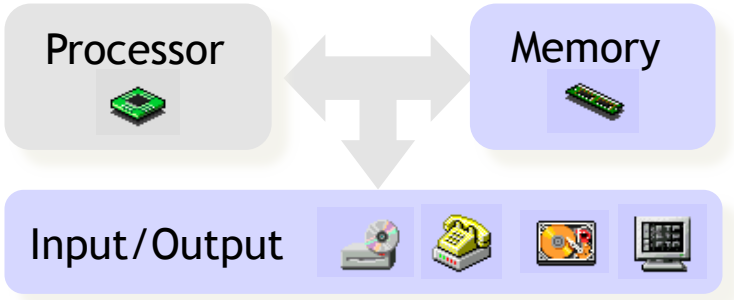
CS161 – Design and Architecture of Computer Systems

Cache $$$$$

UNIVERSITY OF CALIFORNIA, RIVERSIDE

Memory Systems

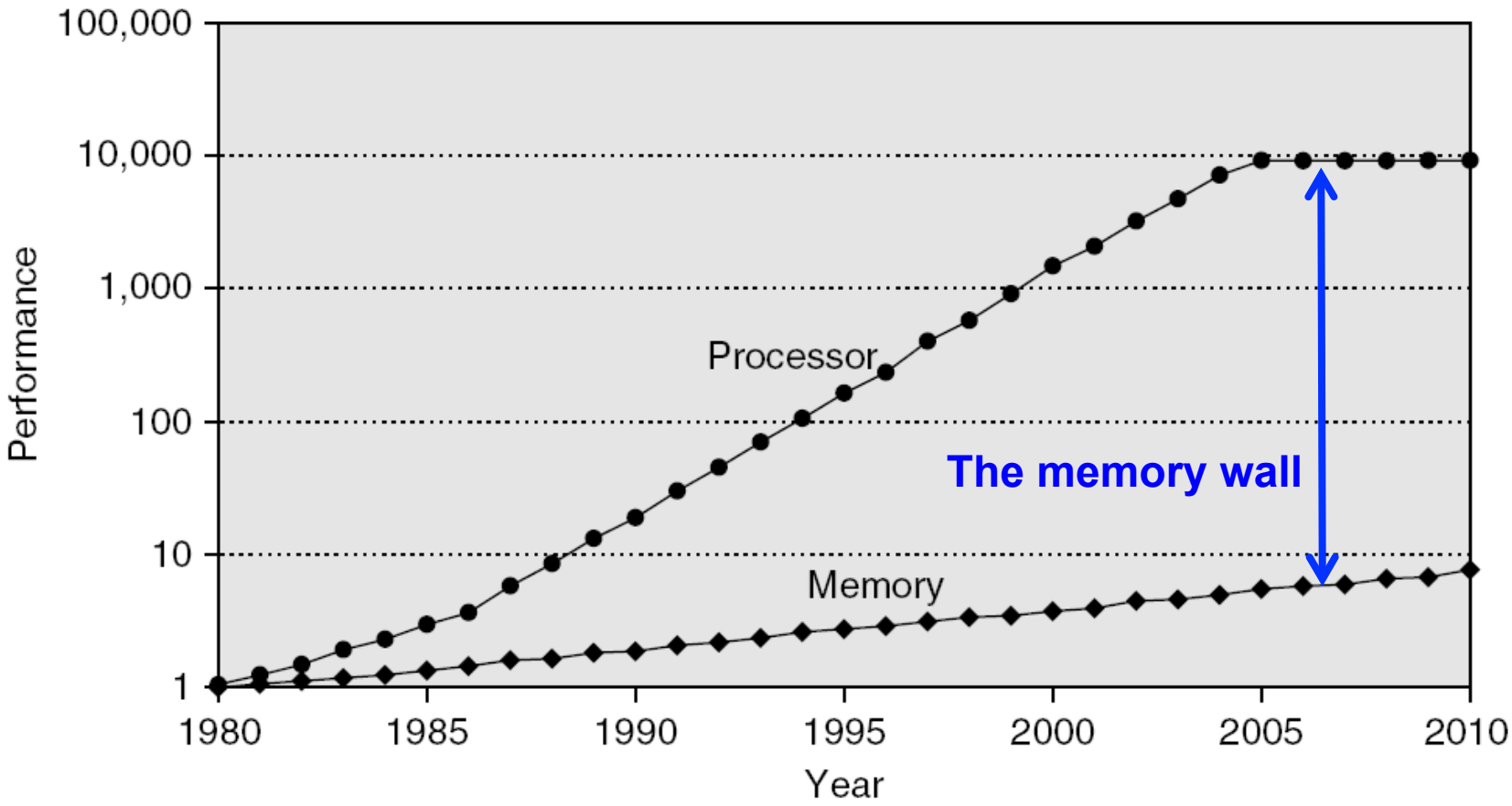
- How can we supply the CPU with enough data to keep it busy?
- We will focus on memory issues,
  - which are frequently bottlenecks that limit the performance of a system.



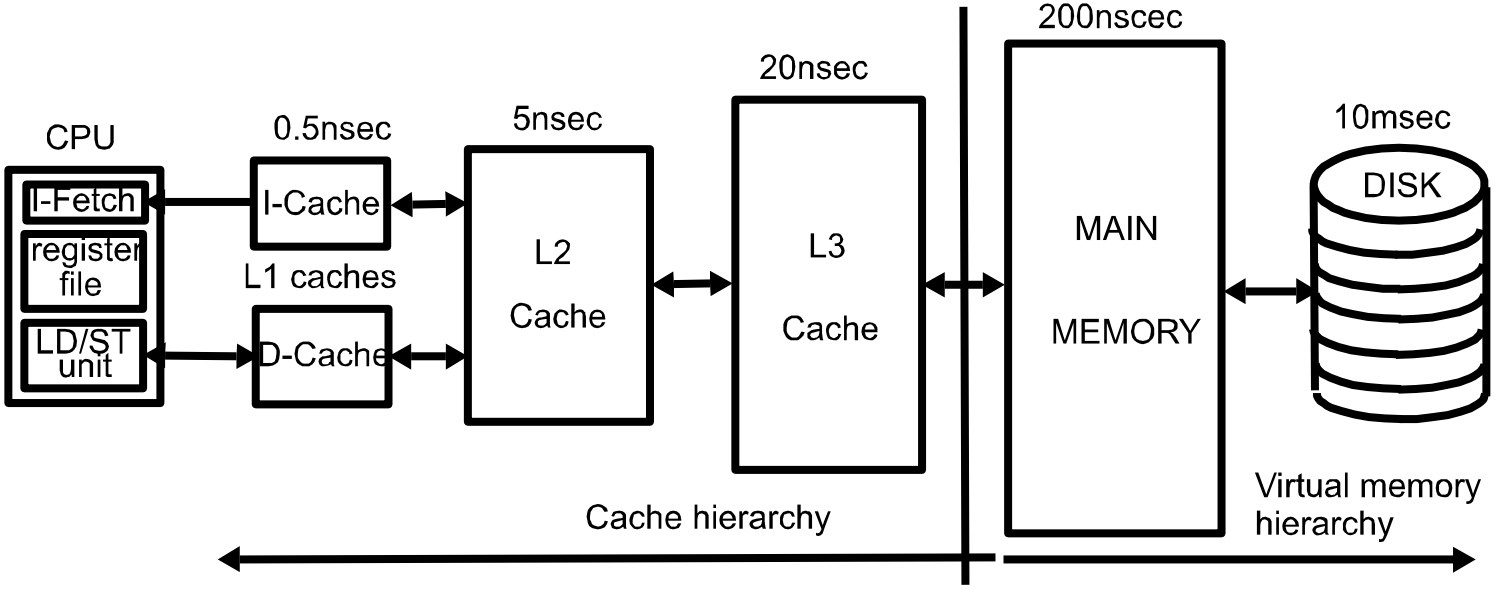
Storage	Speed	Cost	Capacity	Delay	Cost/GB
Static RAM	Fastest	Expensive	Smallest	0.5 – 2.5 ns	$1,000's
Dynamic RAM	Slow	Cheap	Large	50 – 70 ns	$10's
Hard disks	Slowest	Cheapest	Largest	5 – 20 ms	$0.1's

- Ideal memory: large, fast and cheap

Performance Gap



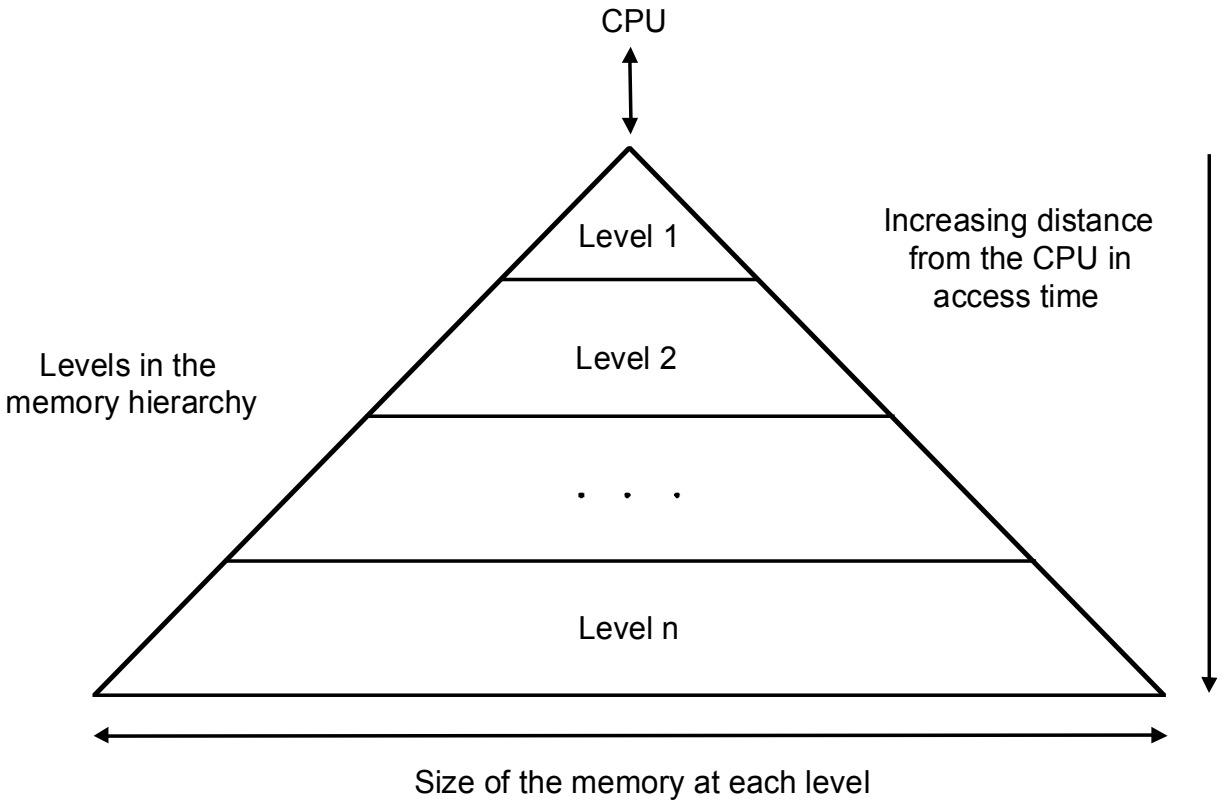
Typical Memory Hierarchy



- Principle of locality:
- A program accesses a relatively small portion of the address space at a time
- Two different types of locality:
  - Temporal locality: if an item is referenced, it will tend to be referenced again soon
  - Spatial locality: if an item is referenced, items whose addresses are close tend to be referenced soon

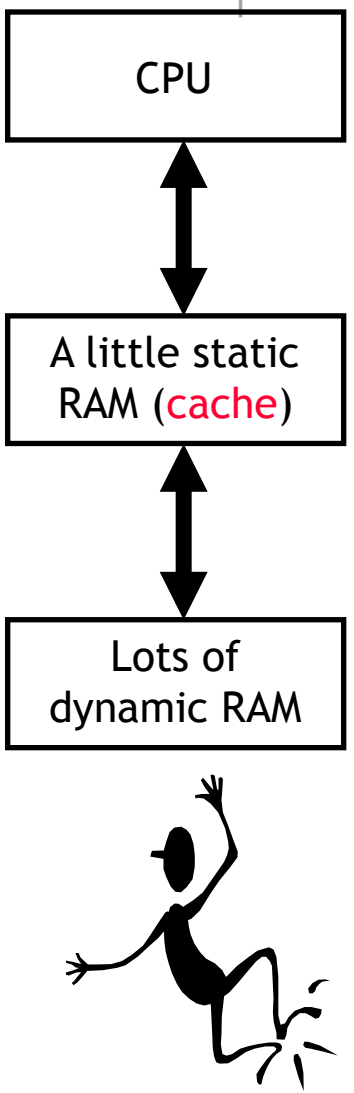
How to Create the Illusion of Big and Fast

- Memory hierarchy – put small and fast memories closer to CPU, large and slow memories further away



Introducing caches

- Introducing a cache – a small amount of fast, expensive memory.
  - The cache goes between the processor and the slower, dynamic main memory.
  - It keeps a copy of the most frequently used data from the main memory.
- Memory access speed increases overall, because we've made the common case faster.
  - Reads and writes to the most frequently used addresses will be serviced by the cache.
  - We only need to access the slower main memory for less frequently used data.



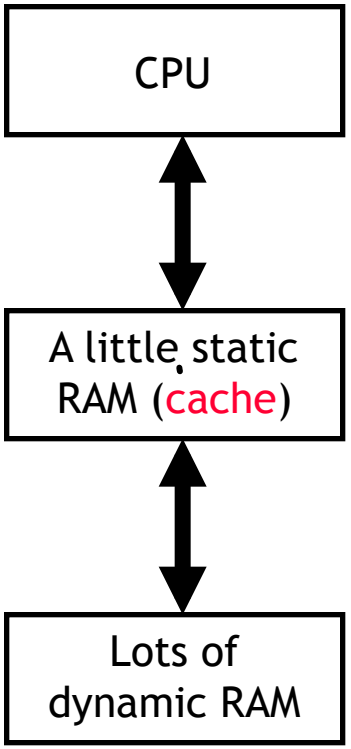
The principle of locality

- Why does the hierarchy work?

- Because most programs exhibit *locality*, which the cache can take advantage of.
  - The principle of temporal locality says that if a program accesses one memory address, there is a good chance that it will access the same address again.
  - The principle of spatial locality says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.

How caches take advantage of locality

- First time the processor reads from an address in main memory, a copy of that data is also stored in the cache.
  - The next time that same address is read, we can use the copy of the data in the cache *instead* of accessing the slower dynamic memory.
  - So the first read is a little slower than before since it goes through both main memory and the cache, but subsequent reads are much faster.
- This takes advantage of temporal locality—commonly accessed data is stored in the faster cache memory.
- By storing a block (multiple words) we also take advantage of spatial locality



Temporal locality in *instructions*

- Loops are excellent examples of temporal locality in programs.
  - The loop body will be executed many times.
  - The computer will need to access those same few locations of the instruction memory repeatedly.
- For example:

```
Loop:  lw    $t0, 0($s1)
       add   $t0, $t0, $s2
       sw    $t0, 0($s1)
       addi  $s1, $s1, -4
       bne   $s1, $0, Loop
```

  - Each instruction will be fetched over and over again, once on every loop iteration.

Temporal locality in *data*

- Programs often access the same variables over and over, especially within loops. Below, sum and i are repeatedly read and written.

```
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + f(i);
```

- Commonly-accessed variables can sometimes be kept in registers, but this is not always possible.
  - There are a limited number of registers.
  - There are situations where the data must be kept in memory, as is the case with shared or dynamically-allocated memory.

Spatial locality in *instructions*


```
sub  $sp, $sp, 16
sw   $ra, 0($sp)
sw   $s0, 4($sp)
sw   $a0, 8($sp)
sw   $a1, 12($sp)
```

- Nearly every program exhibits spatial locality, because instructions are usually executed in sequence — if we execute an instruction at memory location *i*, then we will probably also execute the next instruction, at memory location *i+1*.
- Code fragments such as loops exhibit *both* temporal and spatial locality.

Spatial locality in *data*

- Programs often access data that is stored contiguously.
  - Arrays, like a in the code on the top, are stored in memory contiguously.
  - The individual fields of a record or object like employee are also kept contiguously in memory.

```
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + a[i];
```

```
employee.name = "Homer Simpson";
employee.boss = "Mr. Burns";
employee.age = 45;
```

CACHE BASICS

Definitions: Hits and misses

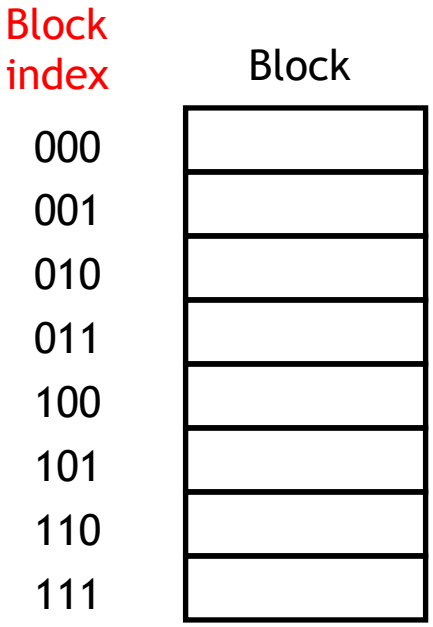
- A cache hit
  - occurs if the cache contains the data that we're looking for. Hits are good, because the cache can return the data much faster than main memory.

- A cache miss
  - occurs if the cache does not contain the requested data. This is bad, since the CPU must then wait for the slower main memory.

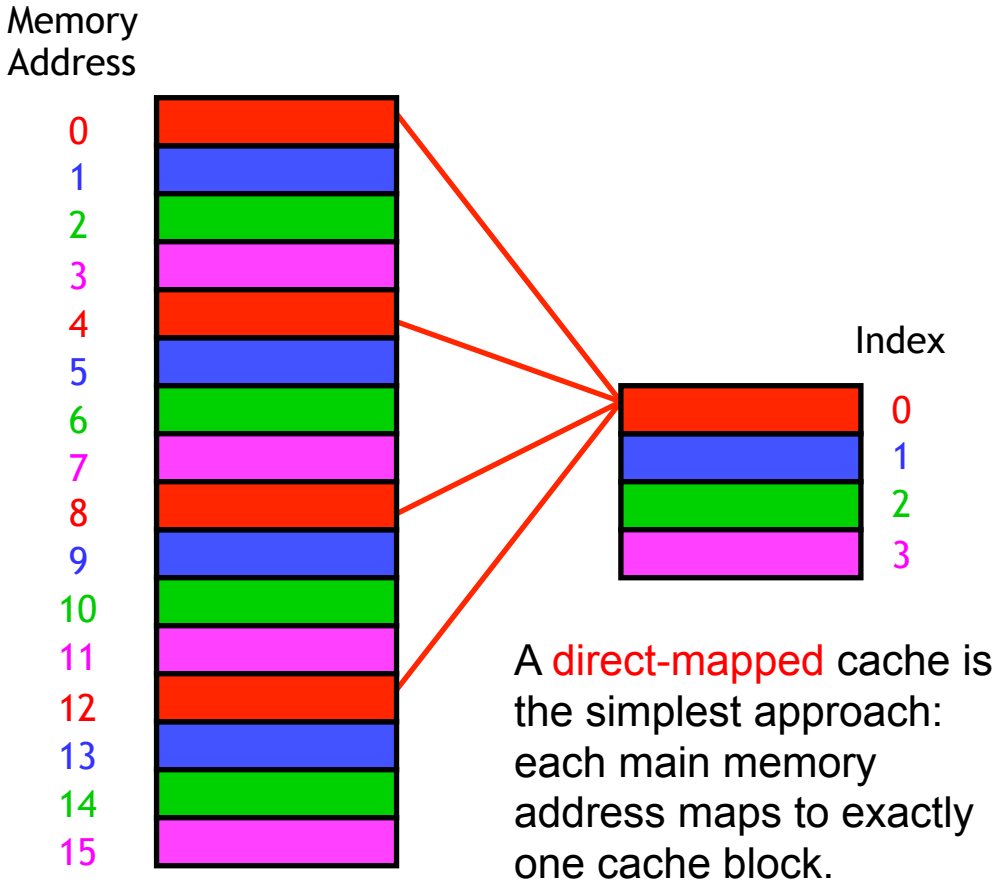
- There are two basic measurements of cache performance.
  - The hit rate is the percentage of memory accesses that are handled by the cache.
  - The miss rate (1 – hit rate) is the percentage of accesses that must be handled by the slower main RAM.
  - Typical caches have a hit rate of 95% or higher, so in fact most memory accesses will be handled by the cache and will be dramatically faster.

A simple cache design

- Caches are divided into blocks, which may be of various sizes.
  - The number of blocks in a cache is usually a power of 2.

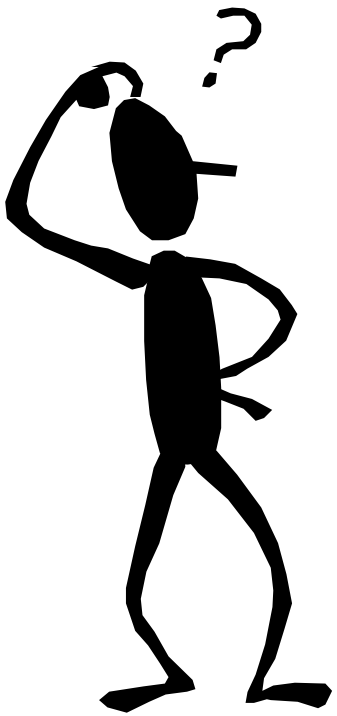


Here is an example cache with eight blocks, each holding one byte.



A direct-mapped cache is the simplest approach: each main memory address maps to exactly one cache block.

Four important questions

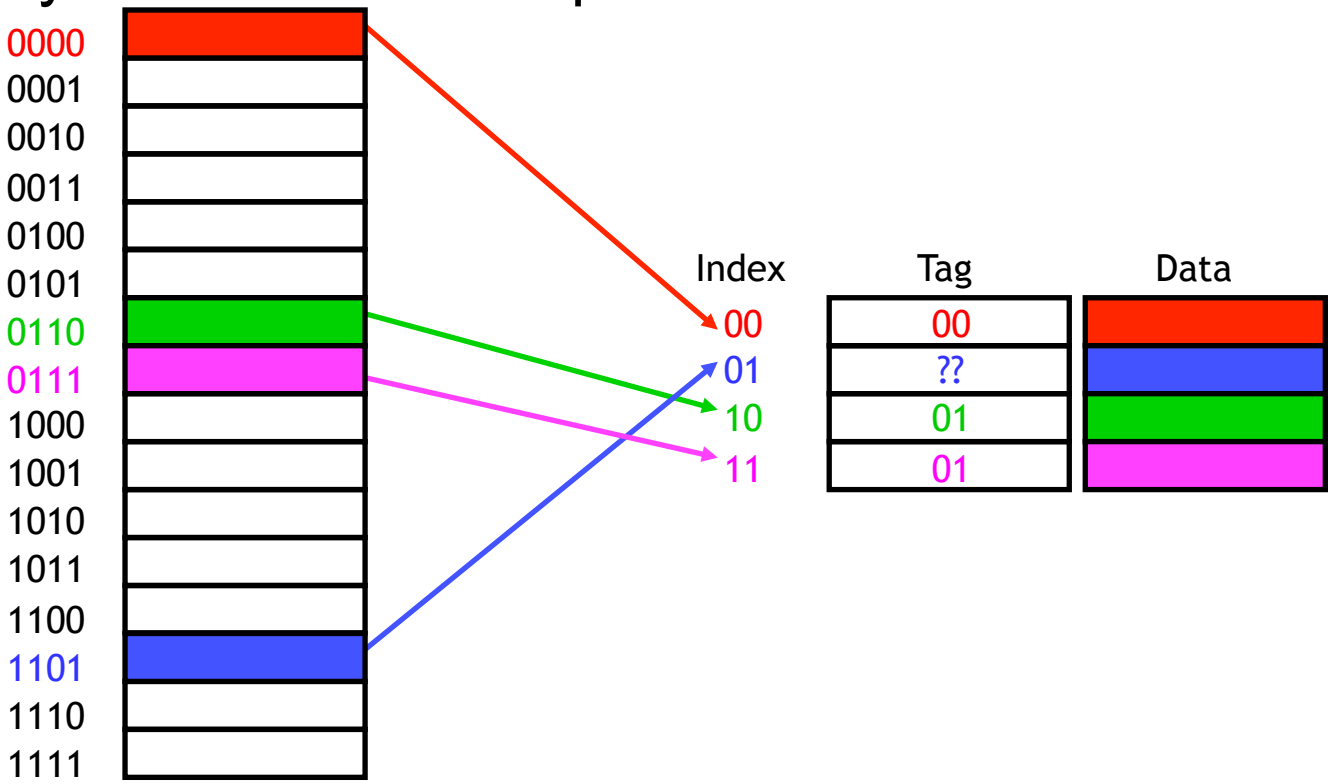


1. When we copy a block of data from main memory to the cache, where exactly should we put it?
2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?
3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we'd have to replace one of the existing blocks in the cache... which one?
4. How can *write* operations be handled by the memory system?

- Questions 1 and 2 are related—we have to know where the data is placed if we ever hope to find it again later!

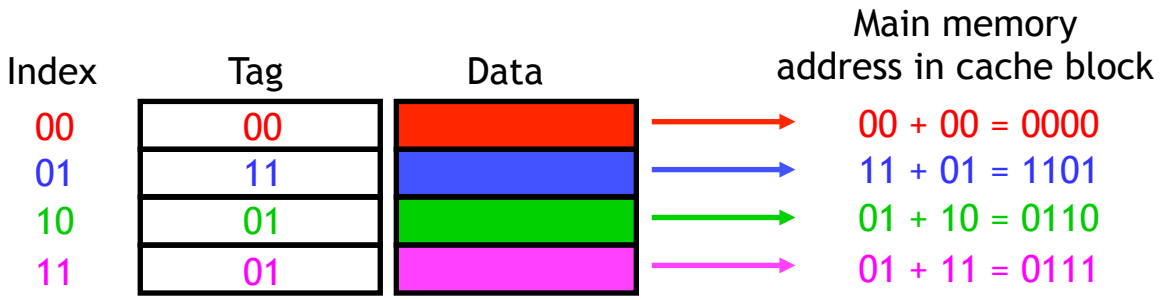
Adding tags

- We need to add tags to the cache, which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block.



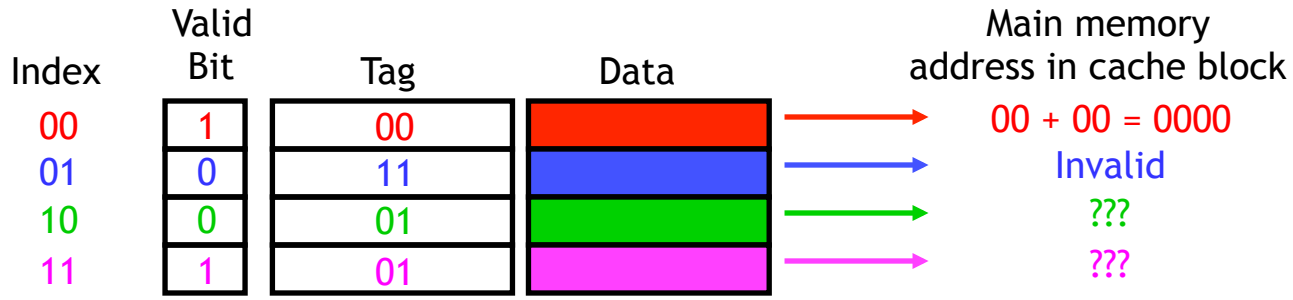
Figuring out what's in the cache

- Now we can tell exactly which addresses of main memory are stored in the cache, by concatenating the cache block tags with the block indices.



One more detail: the valid bit

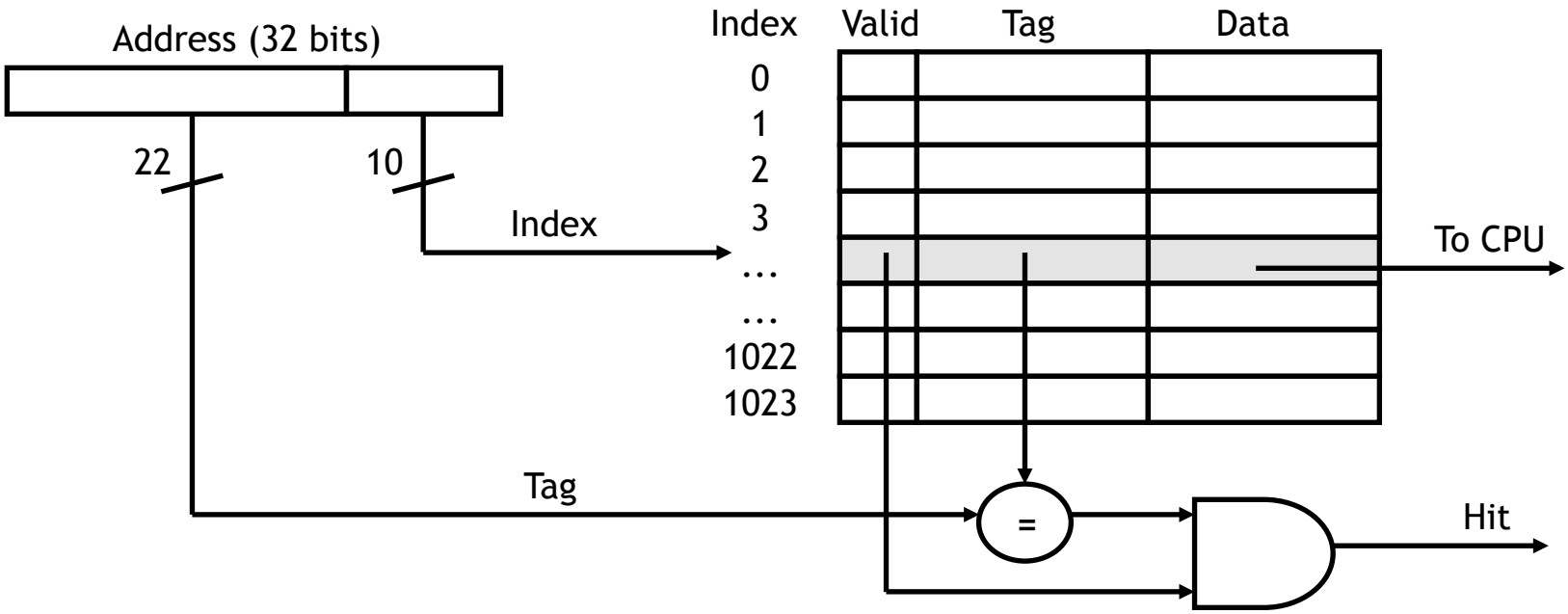
- When started, the cache is empty and does not contain valid data.
- We should account for this by adding a valid bit for each cache block.
  - When the system is initialized, all the valid bits are set to 0.
  - When data is loaded into a particular cache block, the corresponding valid bit is set to 1.



- So the cache contains more than just copies of the data in memory; it also has bits to help us find data within the cache and verify its validity.

What happens on a cache hit

- When the CPU tries to read from memory, the address will be sent to a cache controller.
  - The lowest *k* bits of the block address will index a block in the cache.
  - If the block is valid and the tag matches the upper (*m* − *k*) bits of the *m*-bit address, then that data will be sent to the CPU.
- Here is a diagram of a 32-bit memory address and a 2^{10}-byte cache.

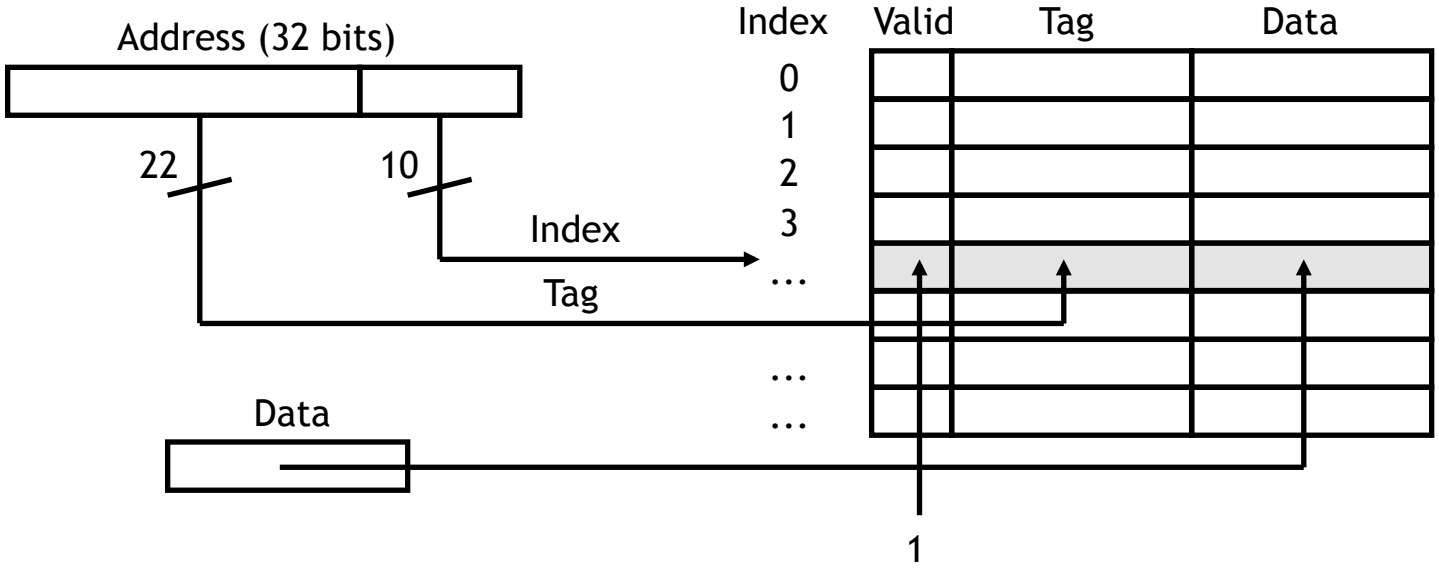


What happens on a cache miss

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access
- The delays that we have been assuming for memories (e.g., 2ns) are really assuming cache hits.

Loading a block into the cache

- After data is read from main memory, putting a copy of that data into the cache is straightforward.
  - The lowest *k* bits of the block address specify a cache block.
  - The upper (*m* − *k*) address bits are stored in the block's tag field.
  - The data from main memory is stored in the block's data field.
  - The valid bit is set to 1.



Memory Hierarchy Basics

- When a word is not found in the cache, a *miss* occurs:
  - Fetch word from lower level in hierarchy, requiring a higher latency reference
  - Lower level may be another cache or the main memory
  - Also fetch the other words contained within the *block*
    - Takes advantage of spatial locality
  - Place block into cache in any location within its *set*, determined by address

Cache Sets and Ways

Ways: Block can go anywhere

Sets: Block mapped by addr

block/line			

n-way set associative

(4-way set associative)

Example: Cache size = 16 blocks

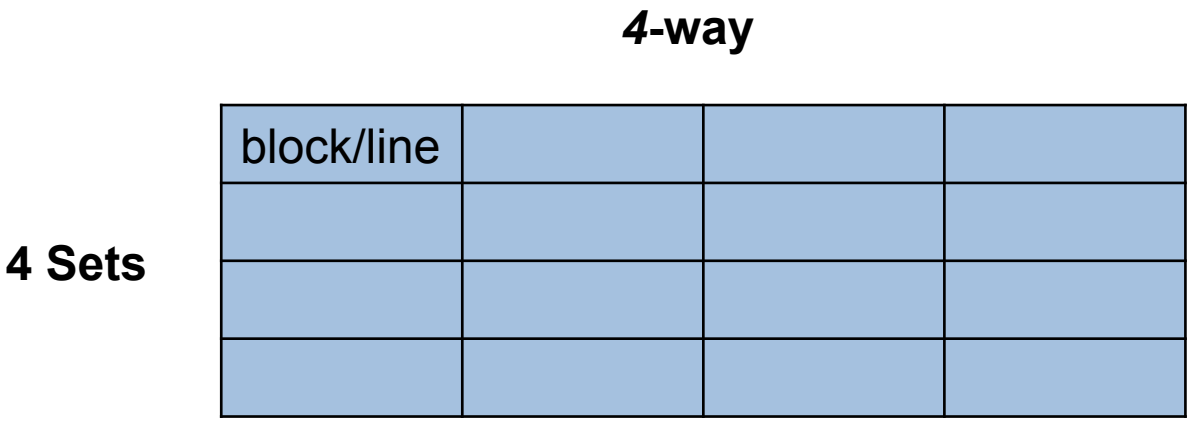
Direct-mapped Cache

Direct mapped cache
Each block maps to only one cache line

aka

1-way set associative

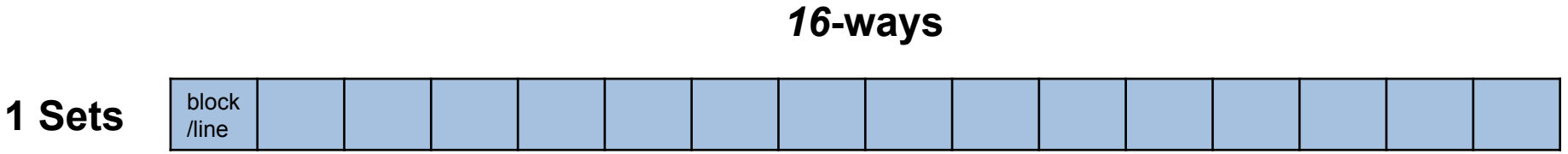
Set Associative Cache



n-way set associative
Each block can be mapped to a set of *n*-lines
Set number is based on block address

(4-way set associative)

Fully Associative Cache

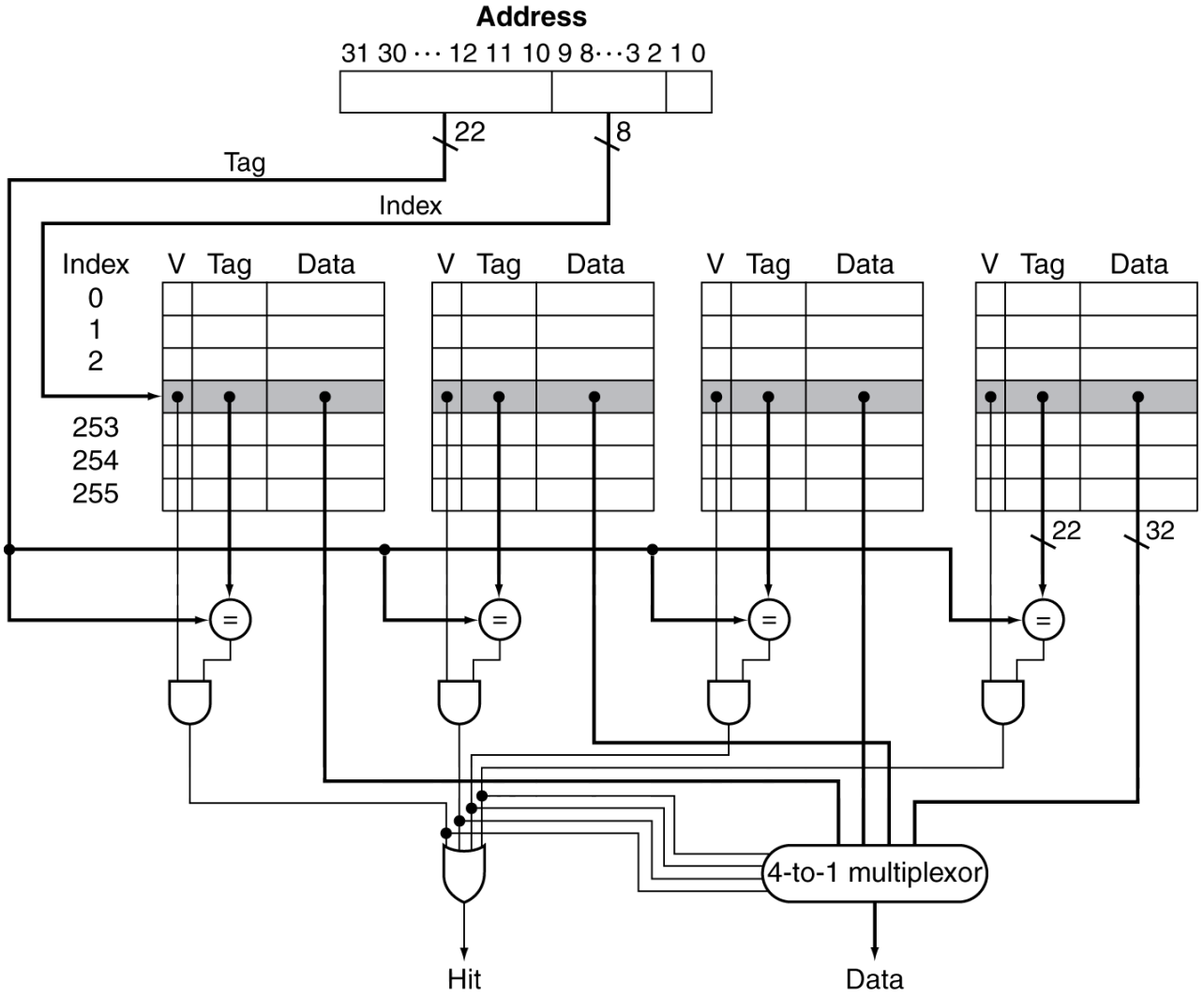


Fully associative
Each block can be mapped to any cache line

aka

m-way set associative
where *m* = size of cache in blocks

Set Associative Cache Organization



Cache Addressing

***n*-Ways**: Block can go anywhere

s-Sets: Block mapped by addr

block/line			

Address

Tag (remainder) bits = 32-*s*-*b*	Index (sets) bits = $\log_2 s$	Offset (block size) bits = $\log_2 b$

m = size of cache in blocks
n = number of ways
b = block size in bytes

Cache size = *s* * *n* * *b*
# of Sets (*s*) = *m* / *n*

Cache Addressing

Ex. 64KB cache, direct mapped, 16 byte block

Address	Tag (remainder) bits = 32-*s*-*b*	Index (sets) bits = $\log_2 s$	Offset (block size) bits = $\log_2 b$
	16	12	4

m = size of cache in blocks
n = number of ways
b = block size in bytes

Cache size = *s* * *n* * *b*
# of Sets (*s*) = *m* / *n*

Cache Addressing

Ex. 64KB cache, 2-way assoc., 16 byte block

Address	Tag (remainder) bits = 32-*s*-*b*	Index (sets) bits = $\log_2 s$	Offset (block size) bits = $\log_2 b$
	17	11	4

m = size of cache in blocks
n = number of ways
b = block size in bytes

Cache size = *s* * *n* * *b*
# of Sets (*s*) = *m* / *n*

Cache Addressing

Ex. 64KB cache, fully assoc., 16 byte block

Address	Tag (remainder) bits = 32-*s*-*b*	Index (sets) bits = $\log_2 s$	Offset (block size) bits = $\log_2 b$
	28	0	4

m = size of cache in blocks
n = number of ways
b = block size in bytes

Cache size = *s* * *n* * *b*
# of Sets (*s*) = *m* / *n*

What if the cache fills up?

- Our third question was what to do if we run out of space in our cache, or if we need to reuse a block for a different memory address.

- A miss causes a new block to be loaded into the cache, automatically overwriting any previously stored data.
  - This is a least recently used replacement policy, which assumes that older data is less likely to be requested than newer data.

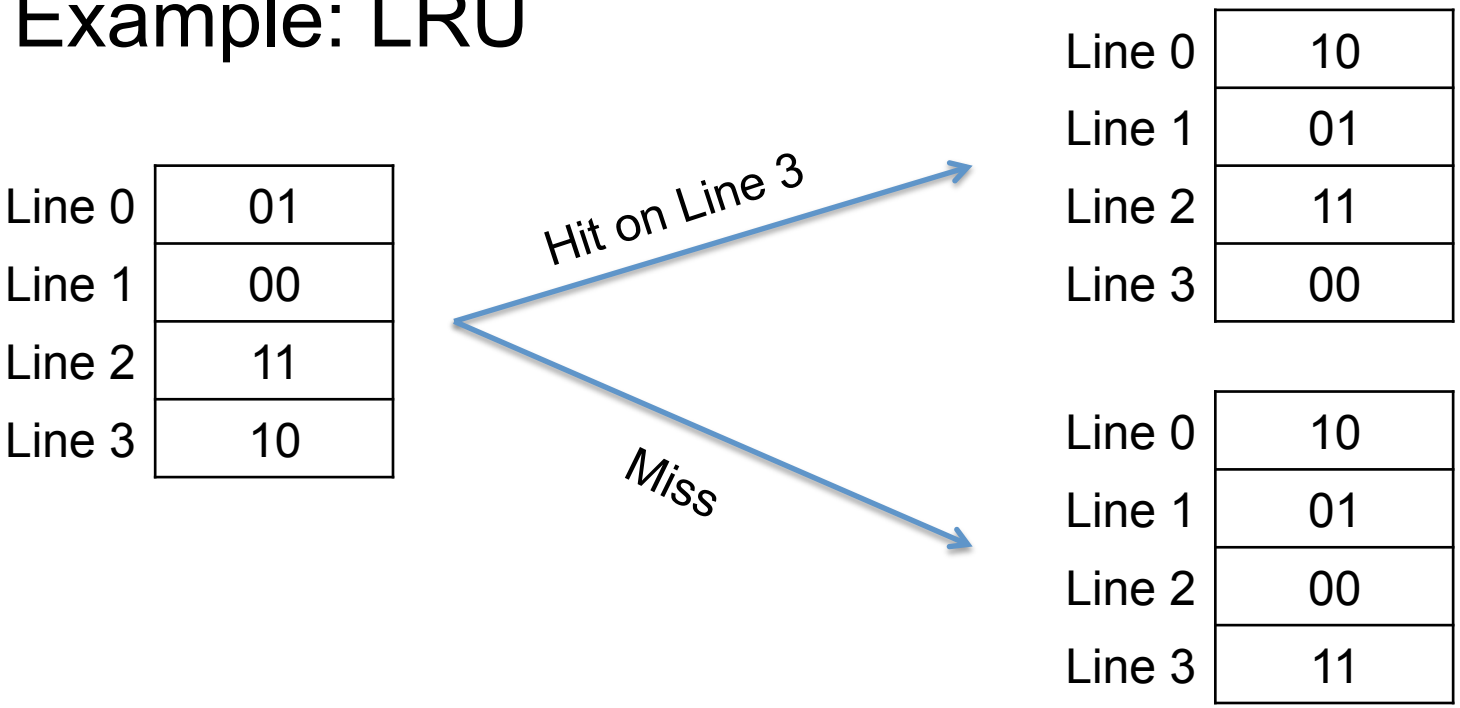
- There are other policies.

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

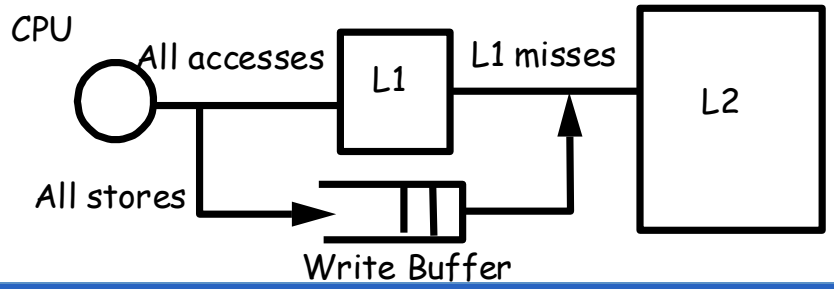
Cache Replacement Policies

- Picks which block to replace within the set
- Ex. - Random, First In First Out (FIFO), Least Recently Used (LRU), Psuedo-LRU
- Example: LRU



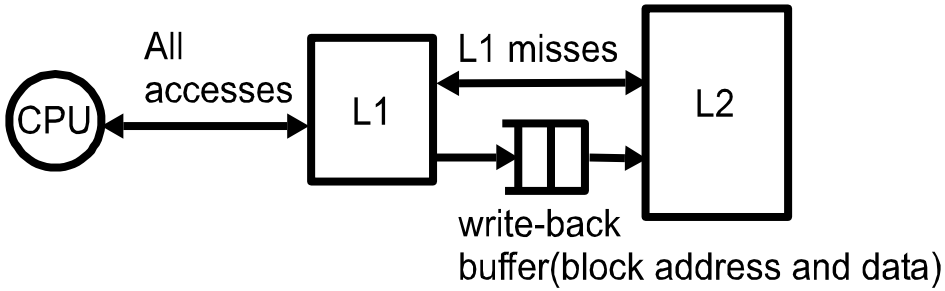
Write-Through

- 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



Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block

Measuring Cache Performance

- Components of CPU time
  - Program execution cycles: Includes cache hit time
  - Memory stall cycles: Mainly from cache misses
- With simplifying assumptions:

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

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

- Example:
  - Given:
    - I-cache miss rate = 2%, D-cache miss rate = 4%, Miss penalty = 100 cycles, Base CPI (ideal cache) = 2, Load & stores are 36% of instructions
  - Miss cycles per instruction
    - I-cache: 0.02 × 100 = 2
    - D-cache: 0.36 × 0.04 × 100 = 1.44
  - Actual CPI = 2 + 2 + 1.44 = 5.44
    - Ideal CPU is 5.44/2 =2.72 times faster

Average Access Time

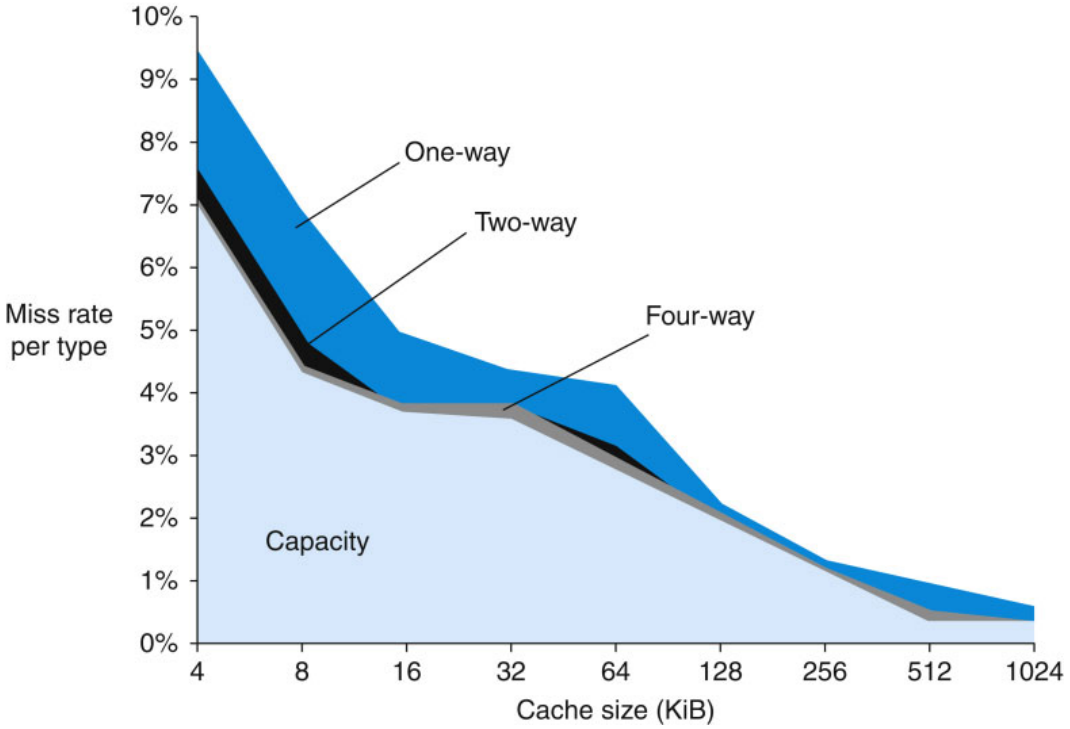
- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Performance Summary

- When CPU performance 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 Misses

- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size



Measuring/Classifying Misses

UCR

- How to find out?
  - Cold misses: Simulate a fully associative infinite cache size
  - Capacity misses: Simulate fully associative cache, then deduct cold misses
  - Conflict misses: Simulate target cache configuration then deduct cold and capacity misses
- Classification is useful to understand how to eliminate misses
- High conflict misses → need higher associativity
- High capacity misses → need larger cache

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

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9
- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
  - Primary miss with L2 hit
    - Penalty = 5ns/0.25ns = 20 cycles
  - Primary miss with L2 miss
    - Extra penalty = 500 cycles
  - CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
  - Performance ratio = 9/3.4 = 2.6

Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time
- L2 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 cache
  - L-1 block size smaller than L-2 block size