CCSYA

Large and Fast: Exploiting Memory Hierarchy

Departamento de Engenharia Informática

Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

Introduction

- **So far, we have assumed that…**
  - the memory system is a linear array of bytes
  - the CPU can access each memory location in a constant amount of time
- **… but as we will see that is a simplification**
- **We will focus on memory and input/output issues, which are frequently bottlenecks that limit the performance of a system**
  - How processors, memory and peripheral devices can be connected
  - How the memory systems is organized
  - How caches can dramatically improve the speed of memory accesses

Memory: Writing and Reading

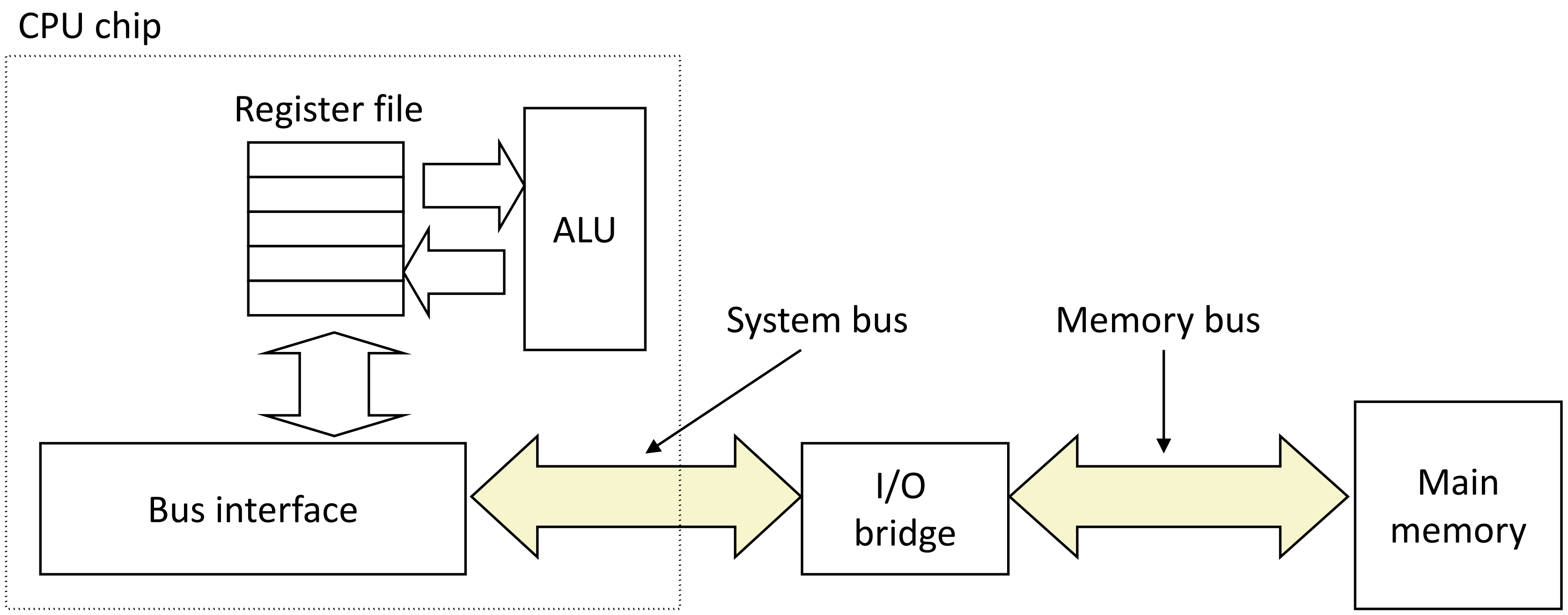
- **Write**
  - Transfer data from CPU to memory

    ```
    sw $t0,32($s3)
    ```
  - “Store” operation
- **Read**
  - Transfer data from memory to CPU

    ```
    lw $t0, 48($s3)
    ```
  - “Load” operation

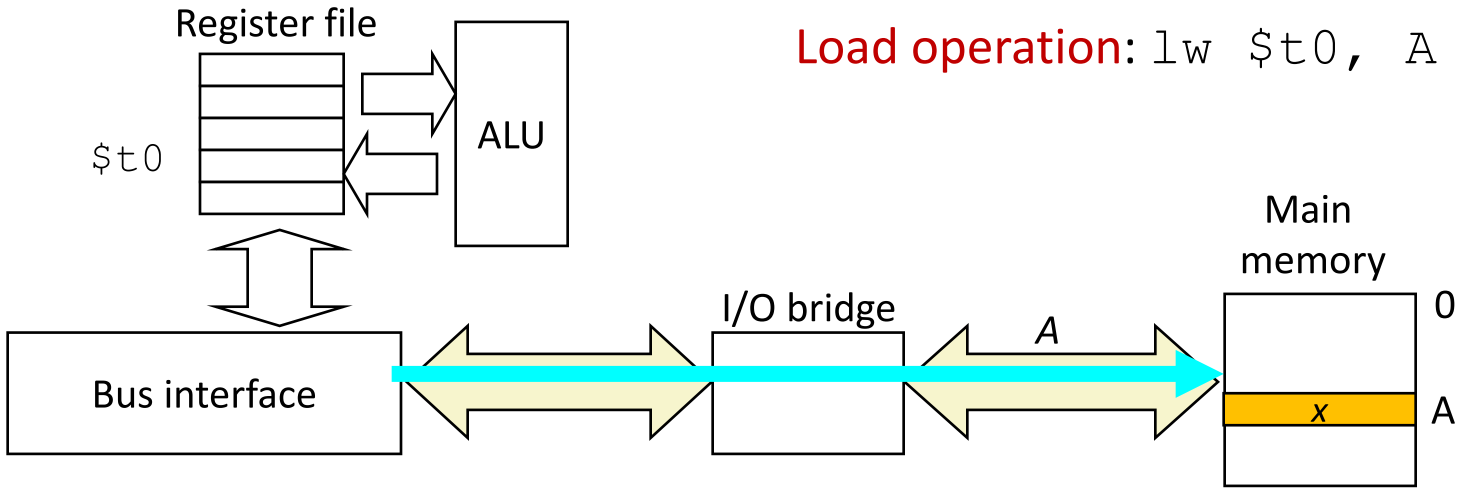
Traditional bus structure

- A **bus** is a collection of parallel wires that carry address, data, and control signals
- Buses are typically shared by multiple devices



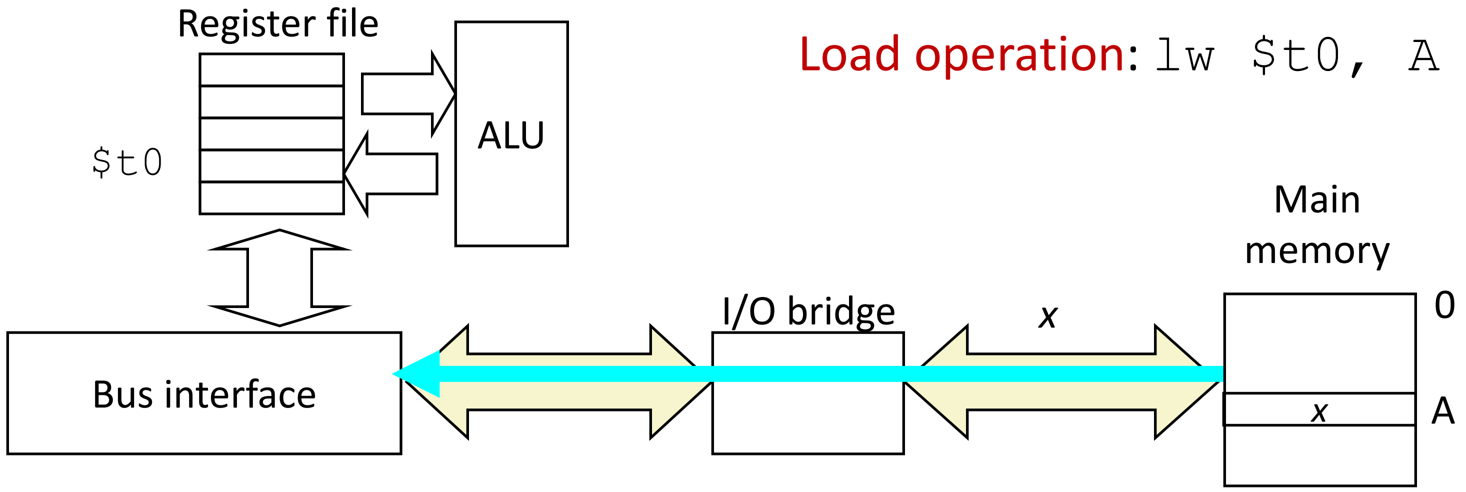
Memory read transaction (1/3)

- **CPU places address A on the memory bus**



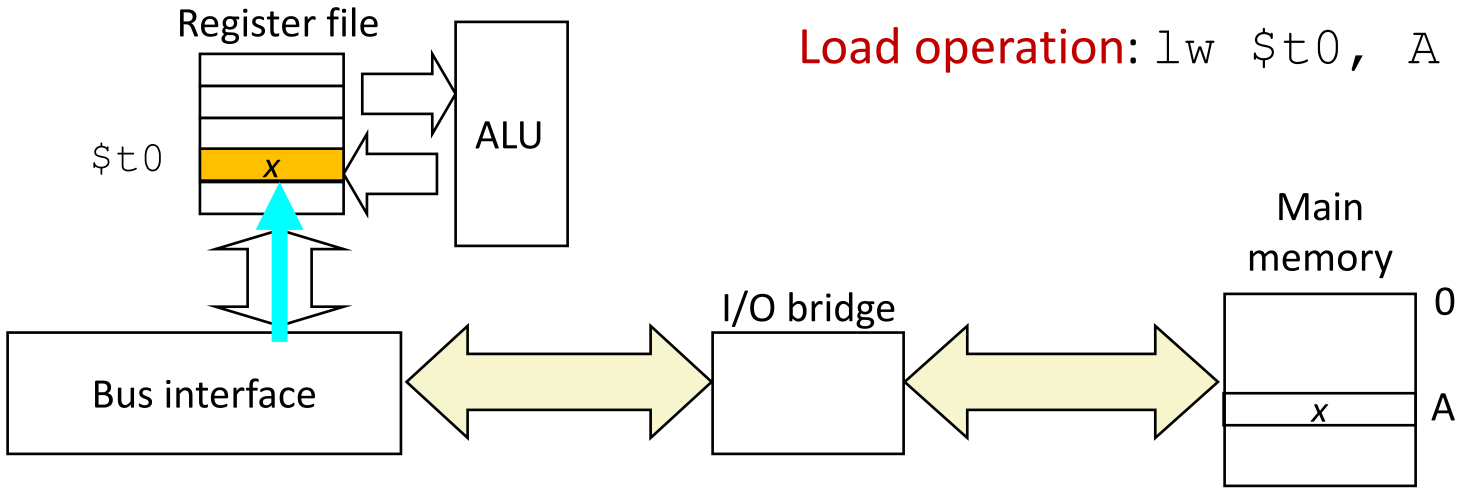
Memory read transaction (2/3)

- **Main memory reads A from the memory bus, retrieves word x, and places it on the bus**



Memory read transaction (3/3)

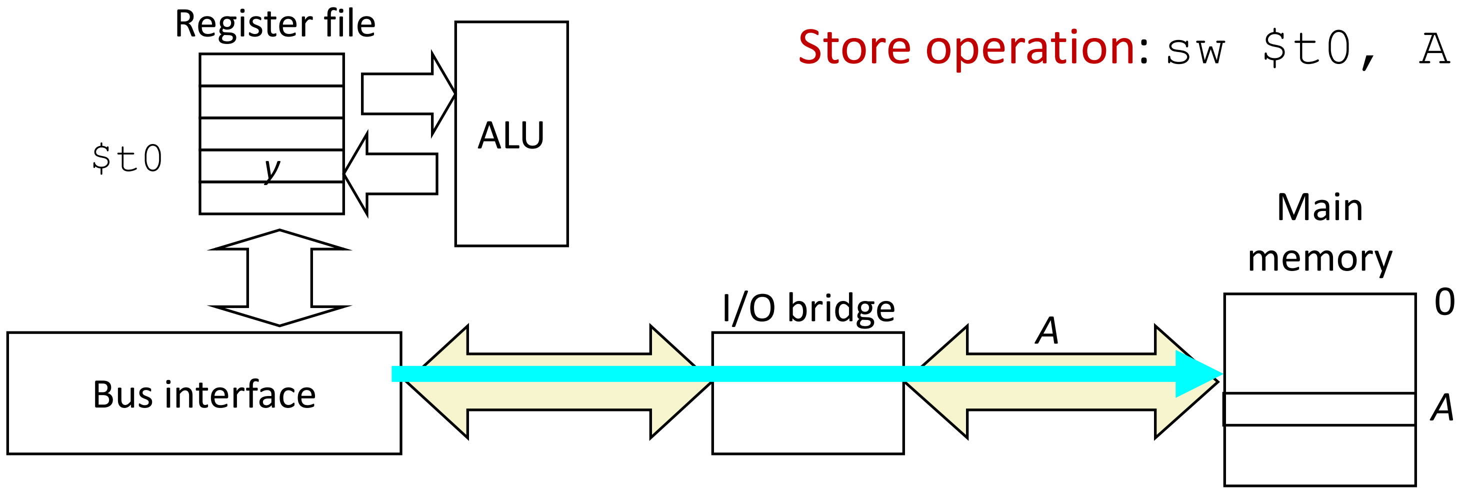
- **CPU reads word x from the bus and copies it into register `$t0`**



Memory write transaction (1/3)

- CPU places address A on bus
- Main memory reads it and waits for the corresponding data word to arrive

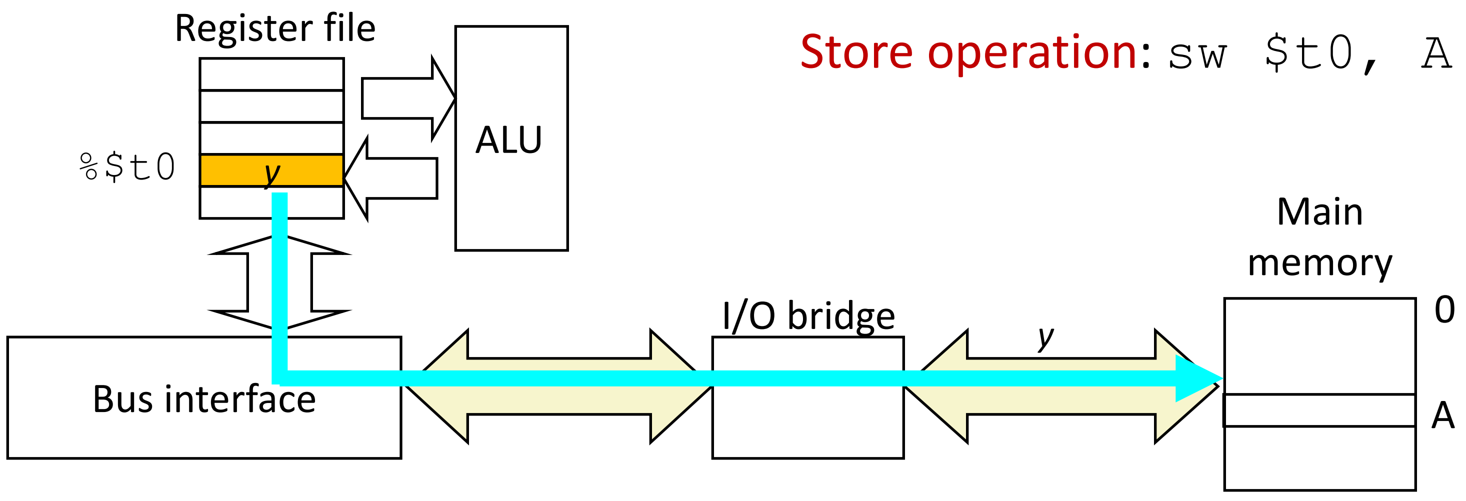
Store operation: `sw $t0, A`



Memory write transaction (2/3)

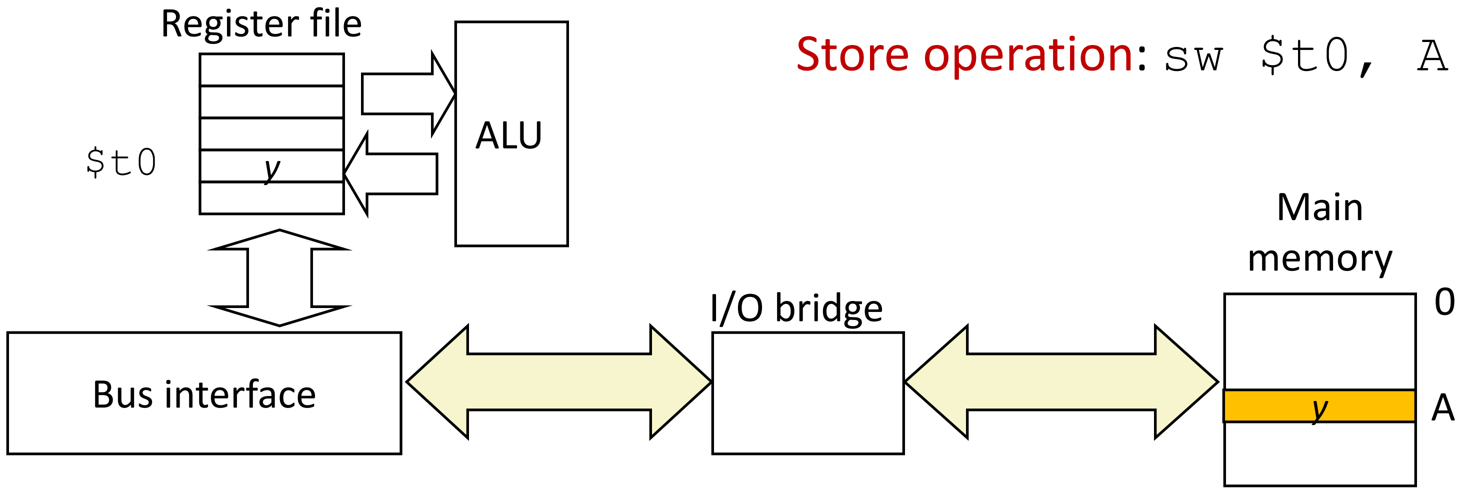
- CPU **places data** word y on the bus

Store operation: `sw $t0, A`



Memory write transaction (3/3)

- Main memory **reads data** word y from the bus and **stores it at address A**



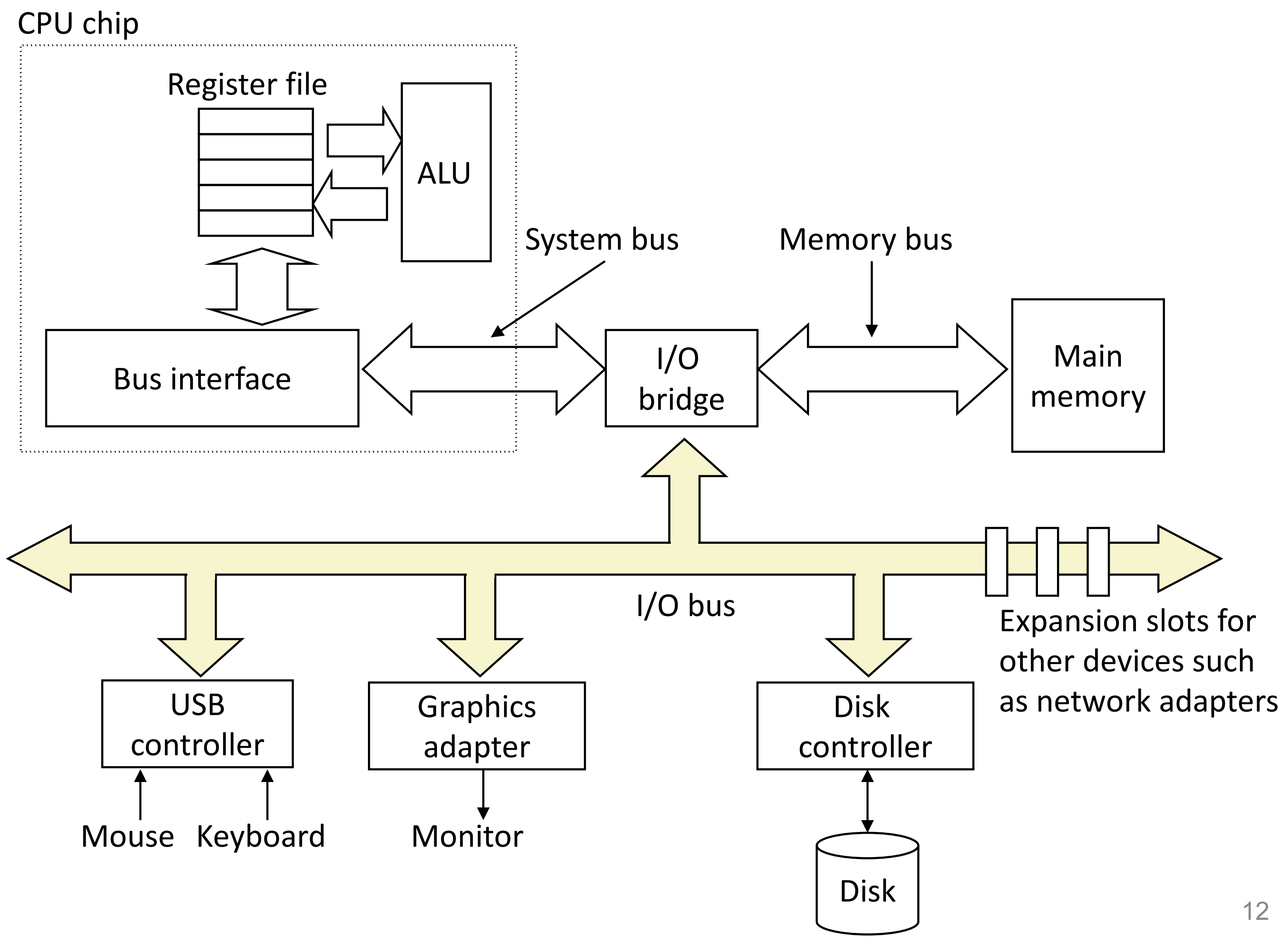
Connecting I/O devices

- **Input/output (I/O) devices such as graphics cards, monitors, mice, keyboards, and disks are connected to the CPU and main memory using an I/O bus**

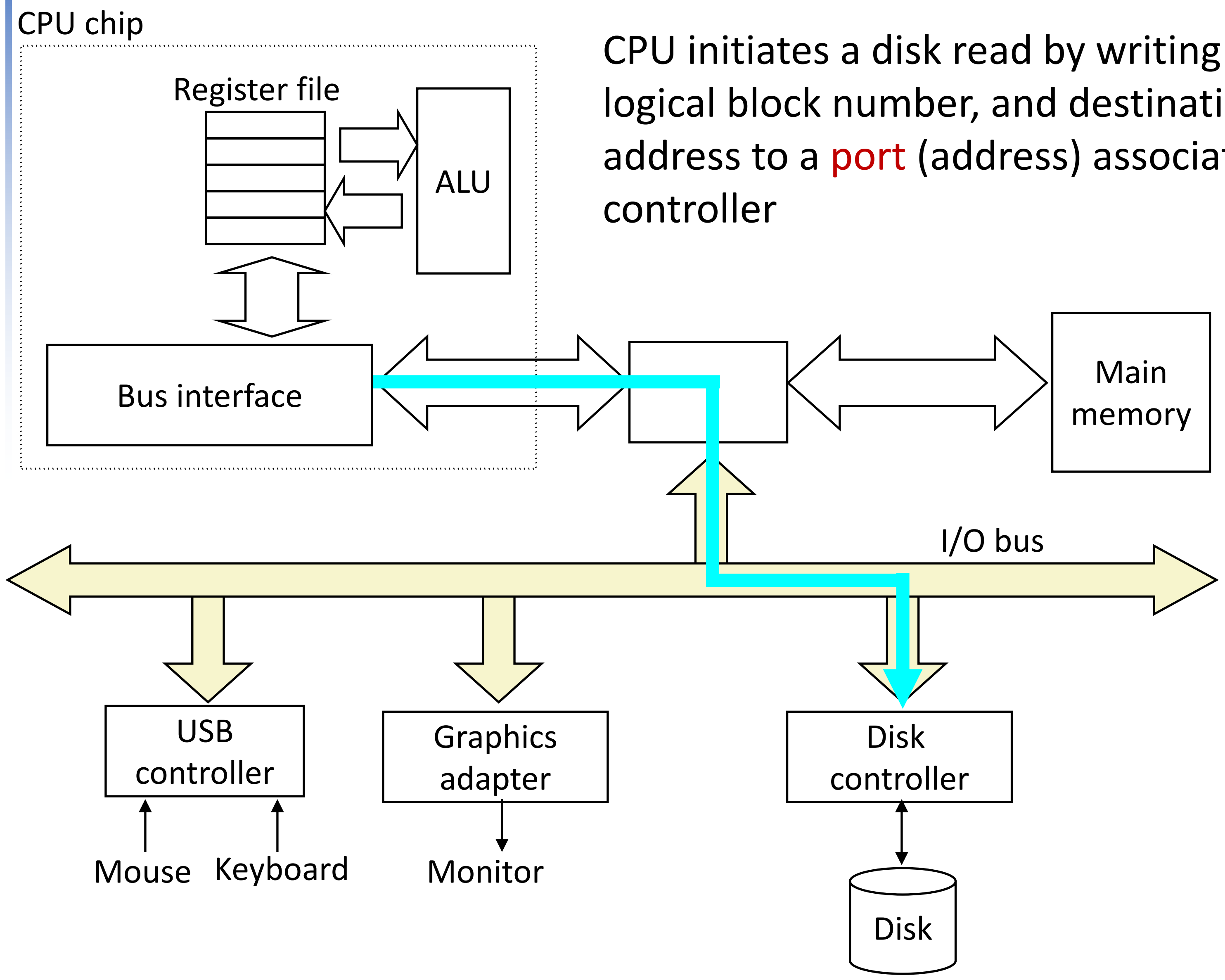
- **I/O buses are designed to be independent of the underlying CPU**
  - Slower than the system and memory buses, but it can accommodate a wide variety of third-party I/O devices

- **The CPU issues commands to I/O devices using a technique called memory-mapped I/O**
  - Each device is associated with (or mapped to) one or more **I/O ports** when it is attached to the bus

Connecting I/O devices

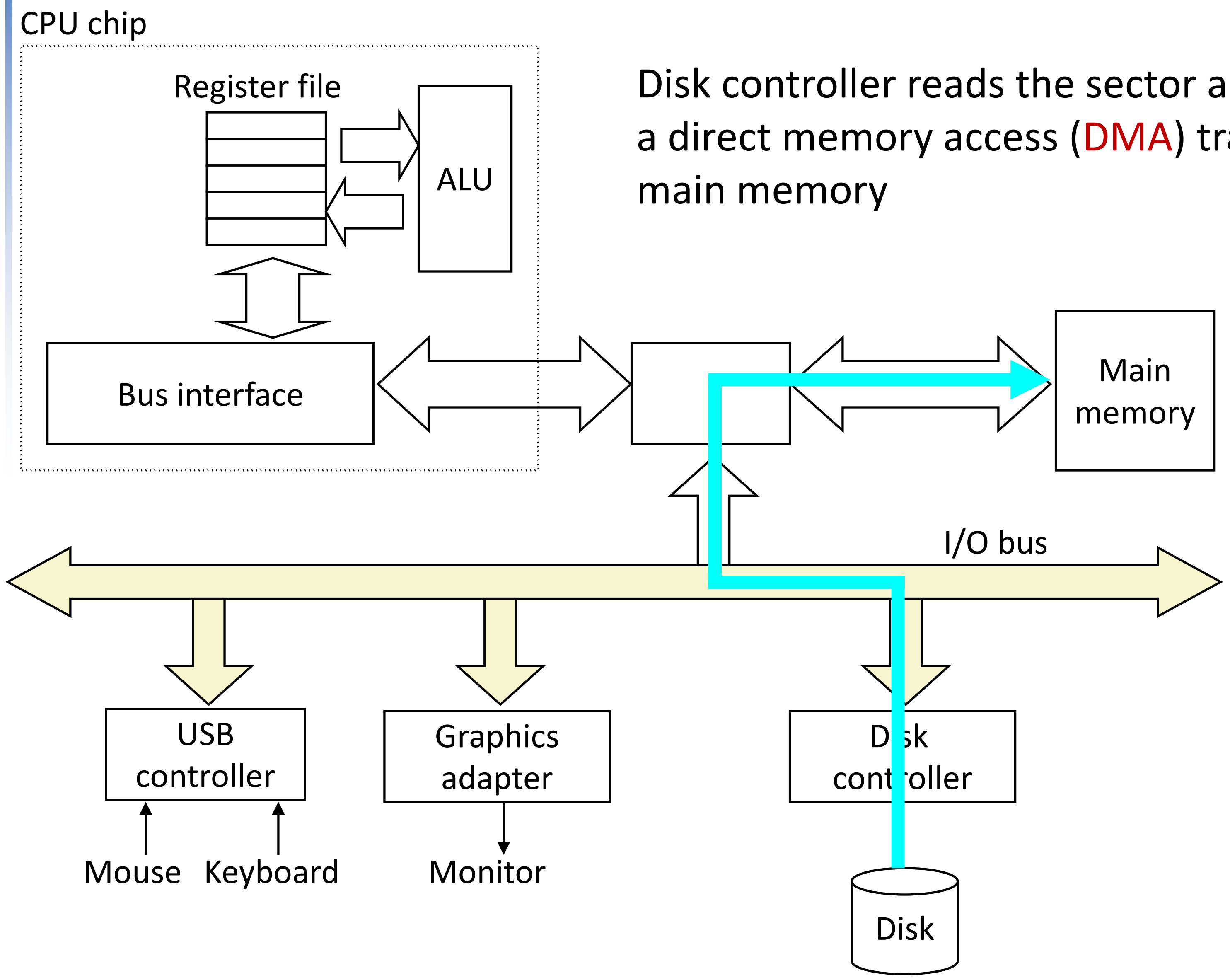


Reading a disk sector (1/3)



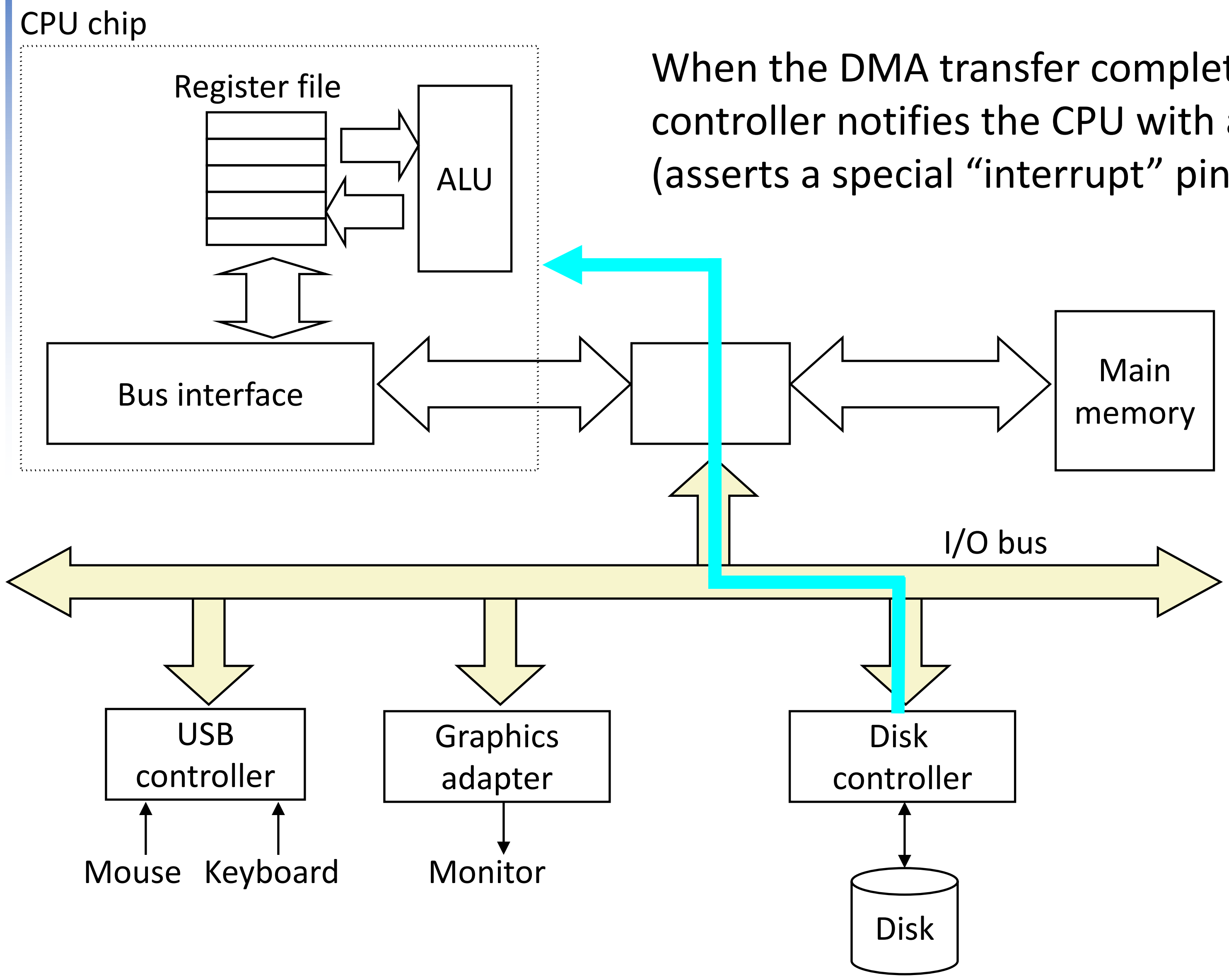
CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller

Reading a disk sector (2/3)



Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory

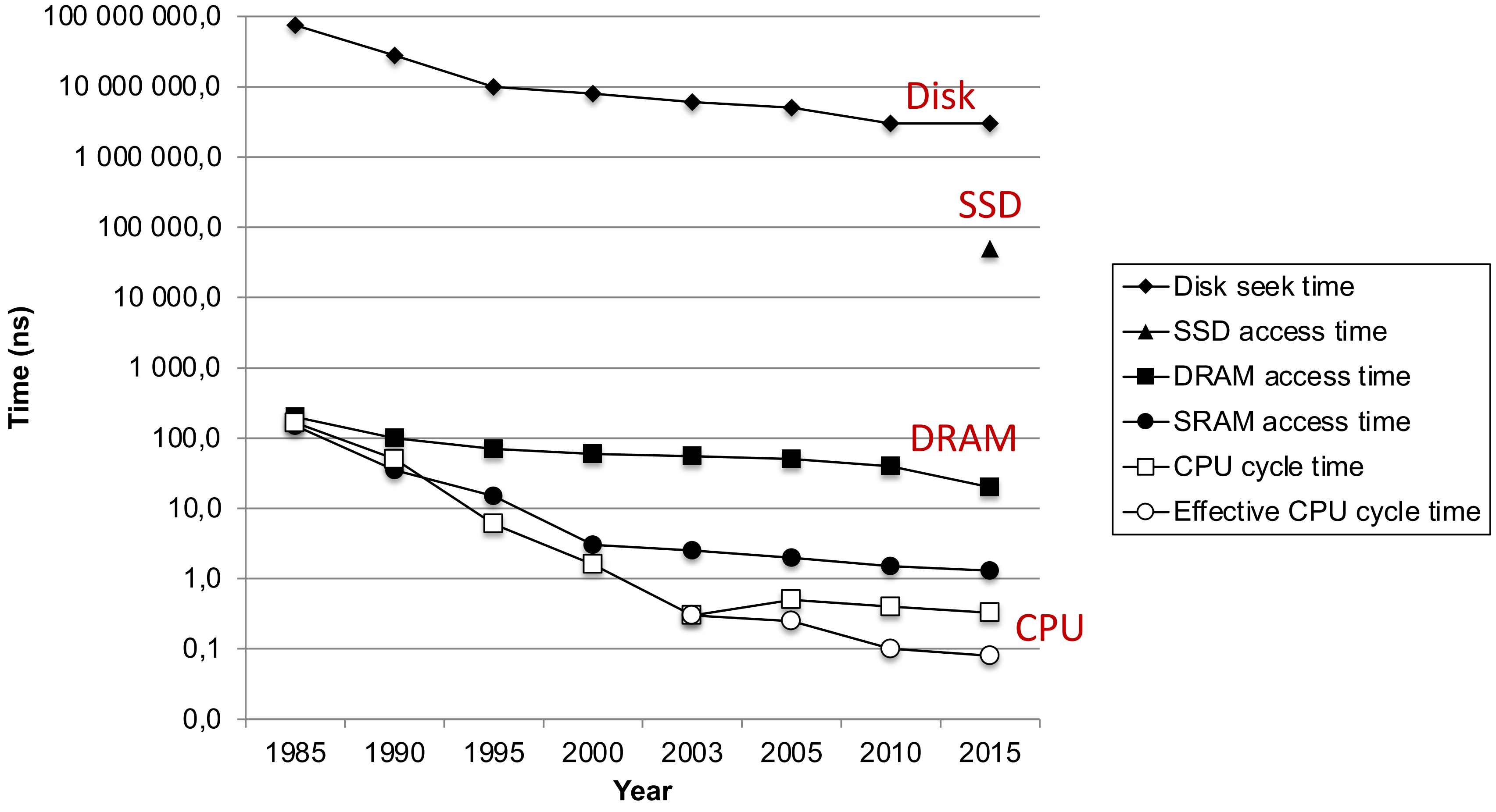
Reading a disk sector (3/3)



When the DMA transfer completes, the disk controller notifies the CPU with an interrupt (asserts a special "interrupt" pin on the CPU)

The CPU-Memory gap

- **The gap widens between DRAM, disk, and CPU speeds**



Locality to the rescue

- **The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality**

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

4.3ms

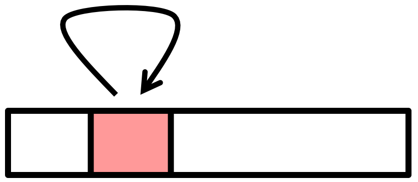
81.8ms

2.67 GHz Intel Core i7 Haswell

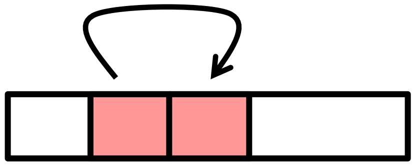
Locality

- **Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**

- **Temporal locality:**
  - Recently referenced items are likely to be referenced again soon



- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time



Locality

- **Loops are excellent examples of temporal and spatial locality in programs**
  - The loop body will be executed many times
  - The CPU will need to access those same few memory locations of repeatedly
  - Usually, data structures that are continuously allocated memory blocks are processed within loops

- **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

Locality – Example

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

Spatial or temporal locality?

- **Data references**
  - Reference array elements in succession (**stride-1 reference pattern**) — spatial
  - Reference variable **`sum`** each iteration — temporal
- **Instruction references**
  - Reference instructions in sequence — spatial
  - Cycle through loop repeatedly — temporal

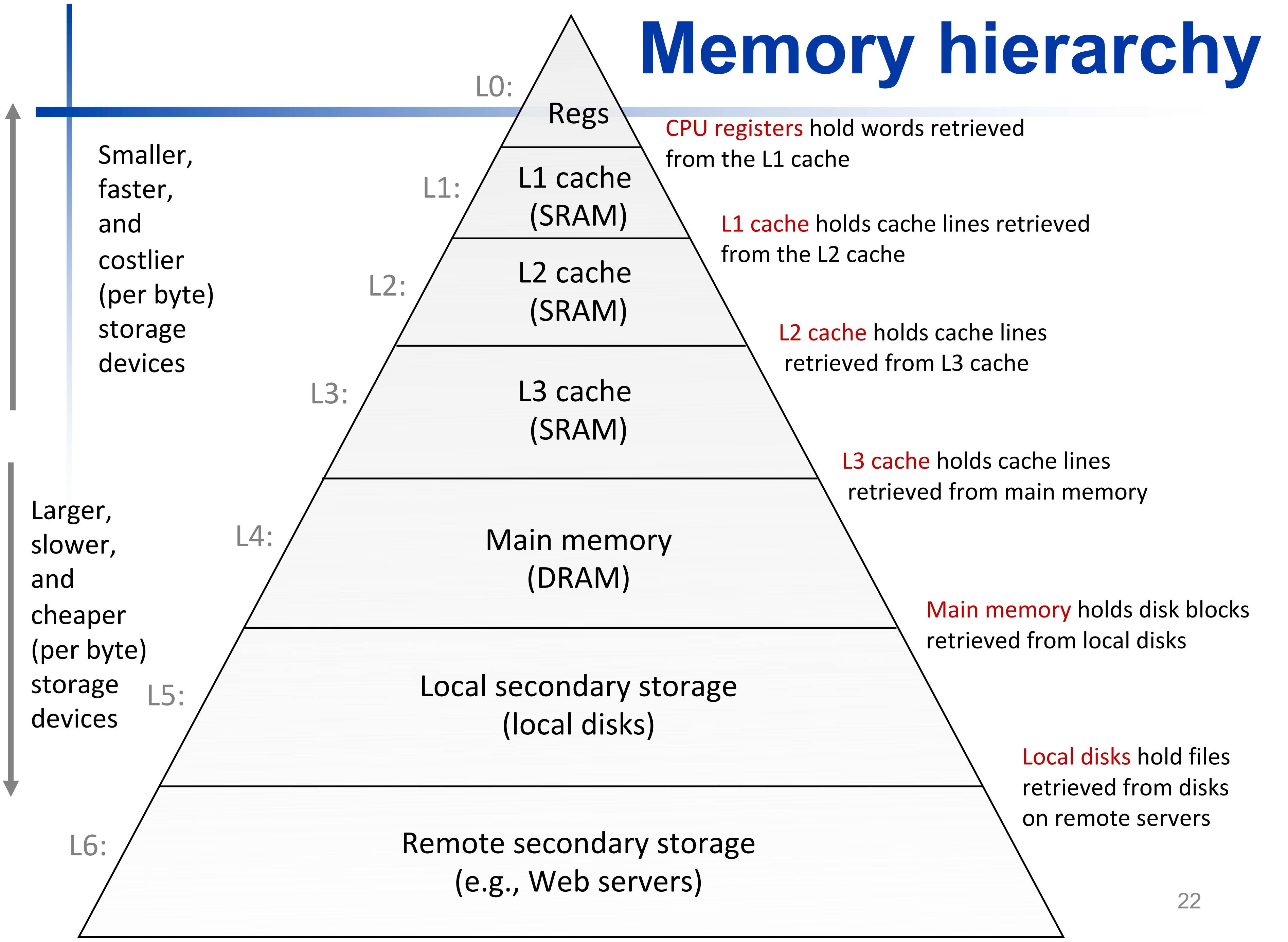
Memory hierarchy

- **Some fundamental and enduring properties of hardware and software:**
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!)
  - The gap between CPU and main memory speed is widening
  - Well-written programs tend to exhibit good locality

- **These fundamental properties complement each other beautifully**

- **They suggest an approach for organizing memory and storage systems known as a memory hierarchy**

Memory hierarchy



Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (e.g., Web servers)
CPU registers hold words retrieved from the L1 cache
L1 cache holds cache lines retrieved from the L2 cache
L2 cache holds cache lines retrieved from L3 cache
L3 cache holds cache lines retrieved from main memory
Main memory holds disk blocks retrieved from local disks
Local disks hold files retrieved from disks on remote servers

Memory hierarchy

- **Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer**
  - It has a big impact on the performance of your applications

- **Minimize how far down the memory hierarchy one must go to manipulate data**
  - One of the main ways to increase system performance
  - If the data your program needs are stored in a CPU register, then they can be accessed in 0 cycles during the execution of the instruction
  - If stored in a cache, 4 to 75 cycles. If stored in main memory, hundreds of cycles. And if stored in disk, tens of millions of cycles!

Caches

- **Cache*:* A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device**

- **Fundamental idea of a memory hierarchy:**
  - For each *k*, the faster, smaller device at level *k* serves as a cache for the larger, slower device at level *k+1*

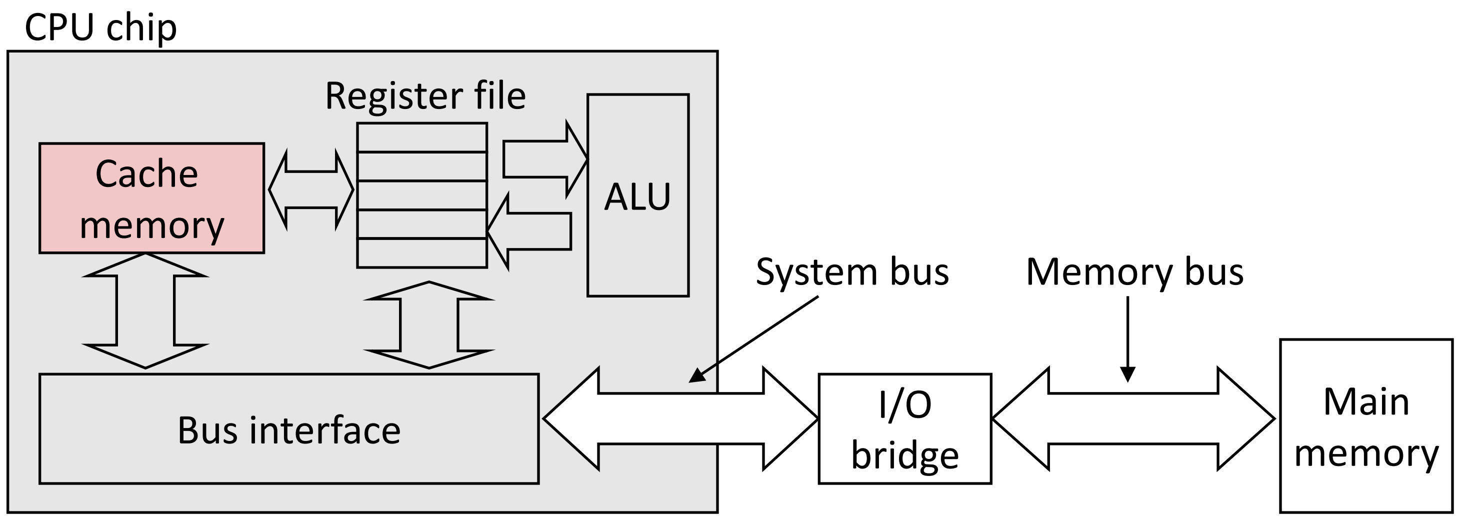
- **Why do memory hierarchies work?**
  - Because of locality, programs tend to access the data at level *k* more often than they access the data at level *k+1*
  - Thus, the storage at level *k+1* can be slower, and thus larger and cheaper per bit

Examples of caching

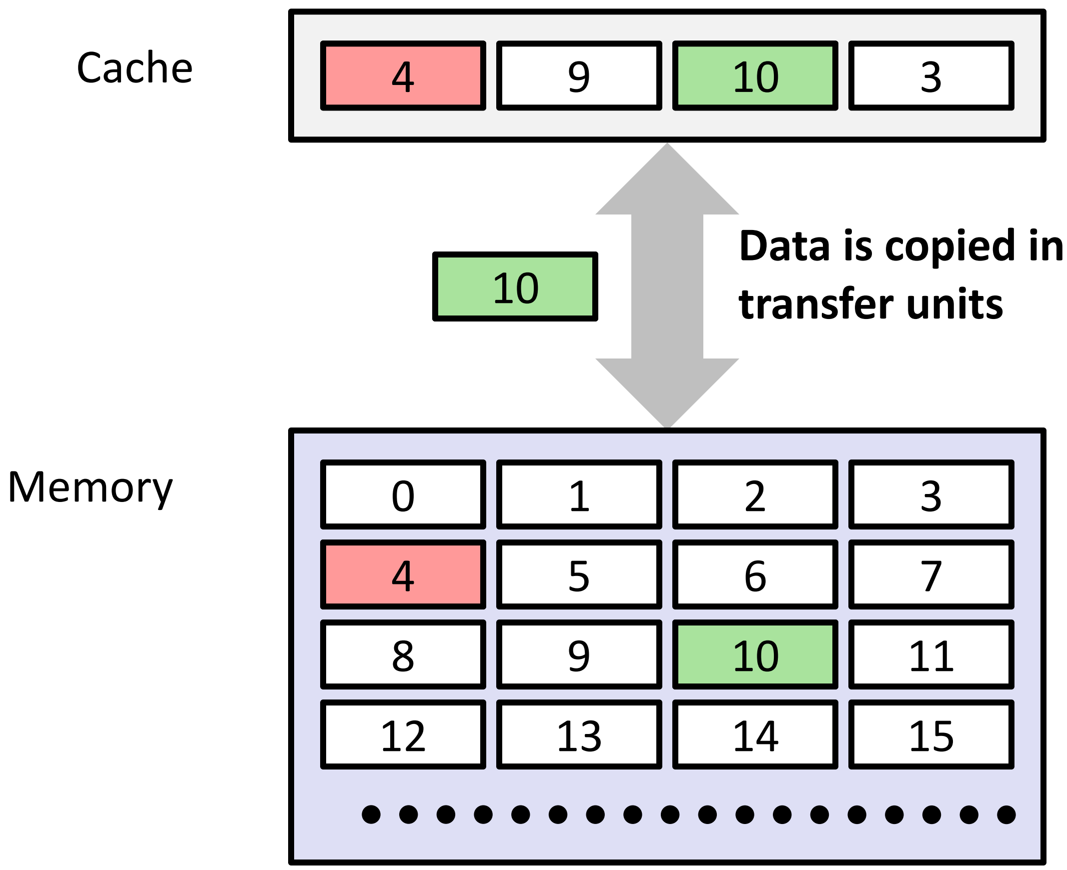
Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Cache memories

- **Cache memories are small, fast SRAM-based memories managed automatically in hardware**
  - Hold frequently accessed blocks of main memory
- **CPU looks first for data in cache**
- **Typical system structure:**



General cache concepts

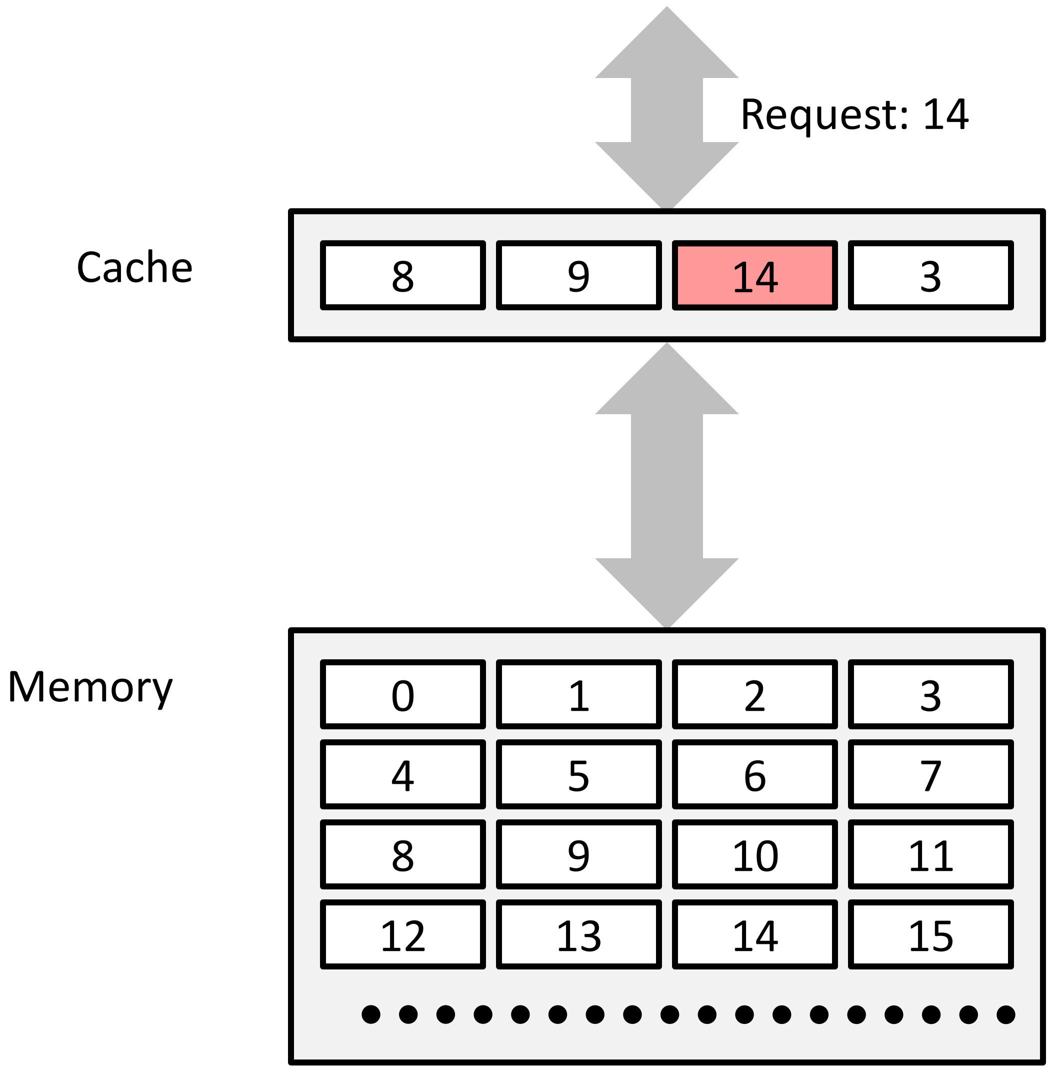


Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into "blocks"

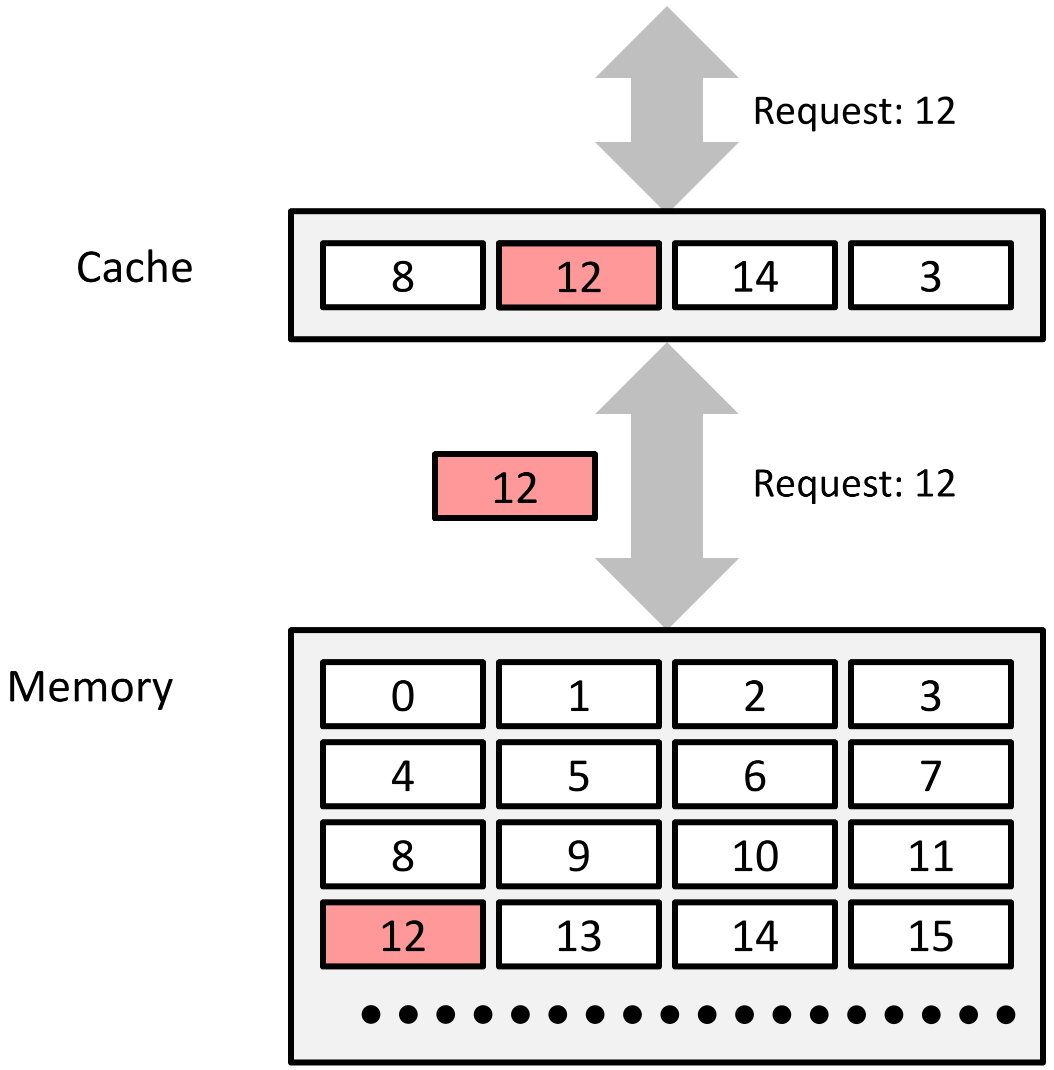
General cache concepts: Hit



Data in block b is needed

Block b is in cache: Hit!

General cache concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache
- **Placement policy**: determines where b goes
- **Replacement policy**: determines which block gets evicted (victim)

Types of Misses

- **Cold (compulsory) miss**
  - Cold misses occur because the cache starts empty and this is the first reference to the block
- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache
- **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time

General cache concepts: Rates

- **There are two basic measurements of cache performance**
  - The hit rate is the percentage of memory accesses that are handled
  - 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**
  - We'll talk more about cache performance

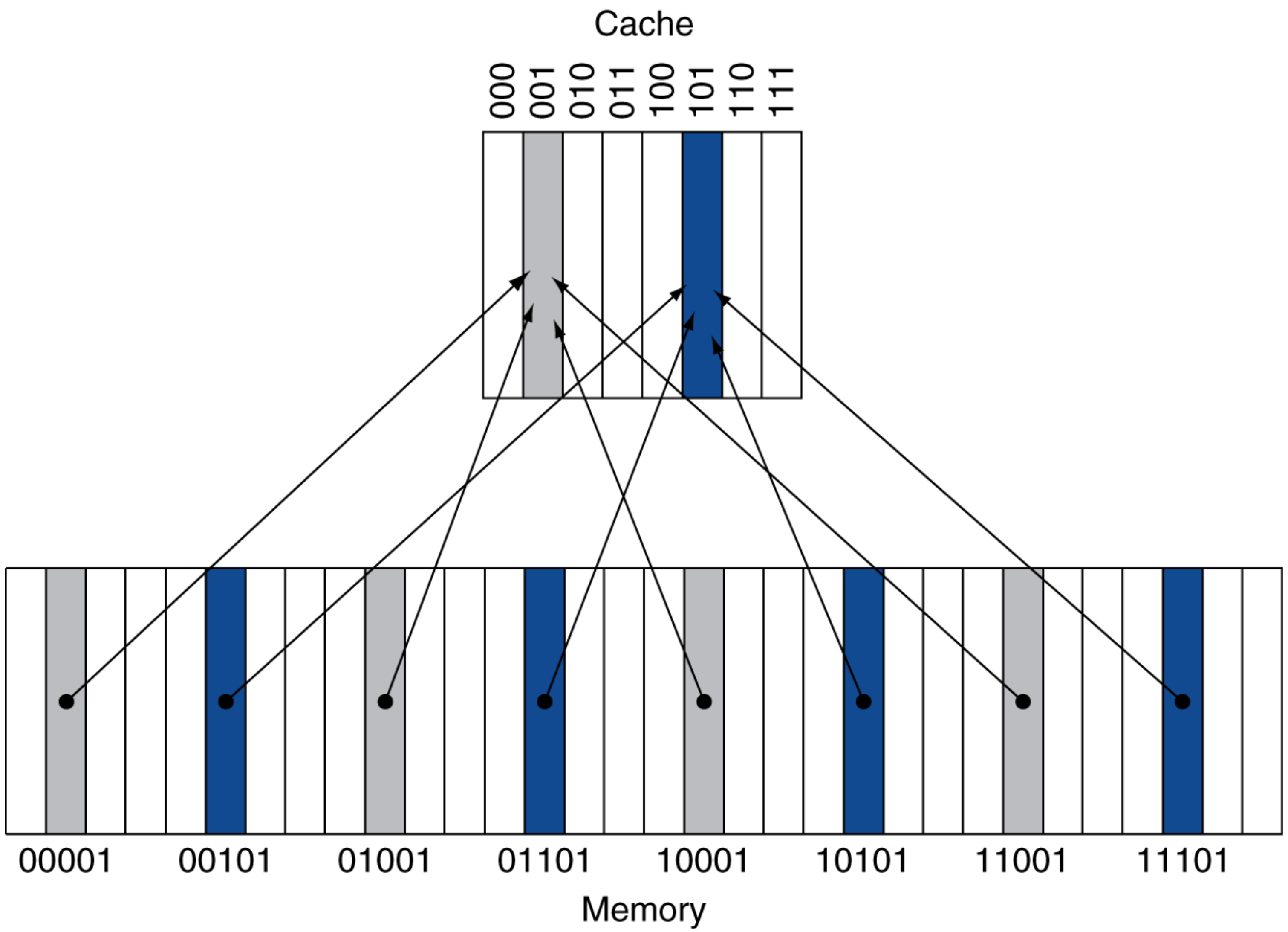
Cache

- **Memory has greater capacity than cache**
  - Therefore, several memory addresses must be mapped to the same cache location
- **Four essential questions**
  - Where can a block be placed?
  - How is a block found?
  - Which block should be replaced after a cache miss?
  - What happens on a write to memory?

Direct-Mapped Cache

- **Location in cache determined by address in memory**
  - Each memory location is mapped to exactly one location in the cache
- **The typical mapping between addresses and cache locations for a direct-mapped cache is usually simple**
  - (Block address) modulo (Number of blocks in cache)
  - If the number of blocks is a power of 2, we can use the $\log_2$ low-order address bits
- **Almost all direct-mapped caches use this mapping to find a block**

Direct-Mapped Cache



Direct-Mapped Cache

- **More than one memory address in the same cache entry**

- **How can we know if the value in cache corresponds to the required memory address?**

- **Conflict solved with a tag**
  - Uses the most significant bits of the address (not used in mapping)

Direct-Mapped Cache

- **More than one memory address in the same cache entry**

- **How can we know if the value in cache is valid?**
  - Entry can be empty or have invalid values

- **Solved with a valid bit**
  - Indicates if the entry is valid or not
  - Valid bit: 1 = present, 0 = not present
  - Initially set to 0

Direct-Mapped Cache Example

- **8-blocks, 1 word/block, direct-mapped**
- **Initial state:**

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

Direct-Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		

Direct-Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

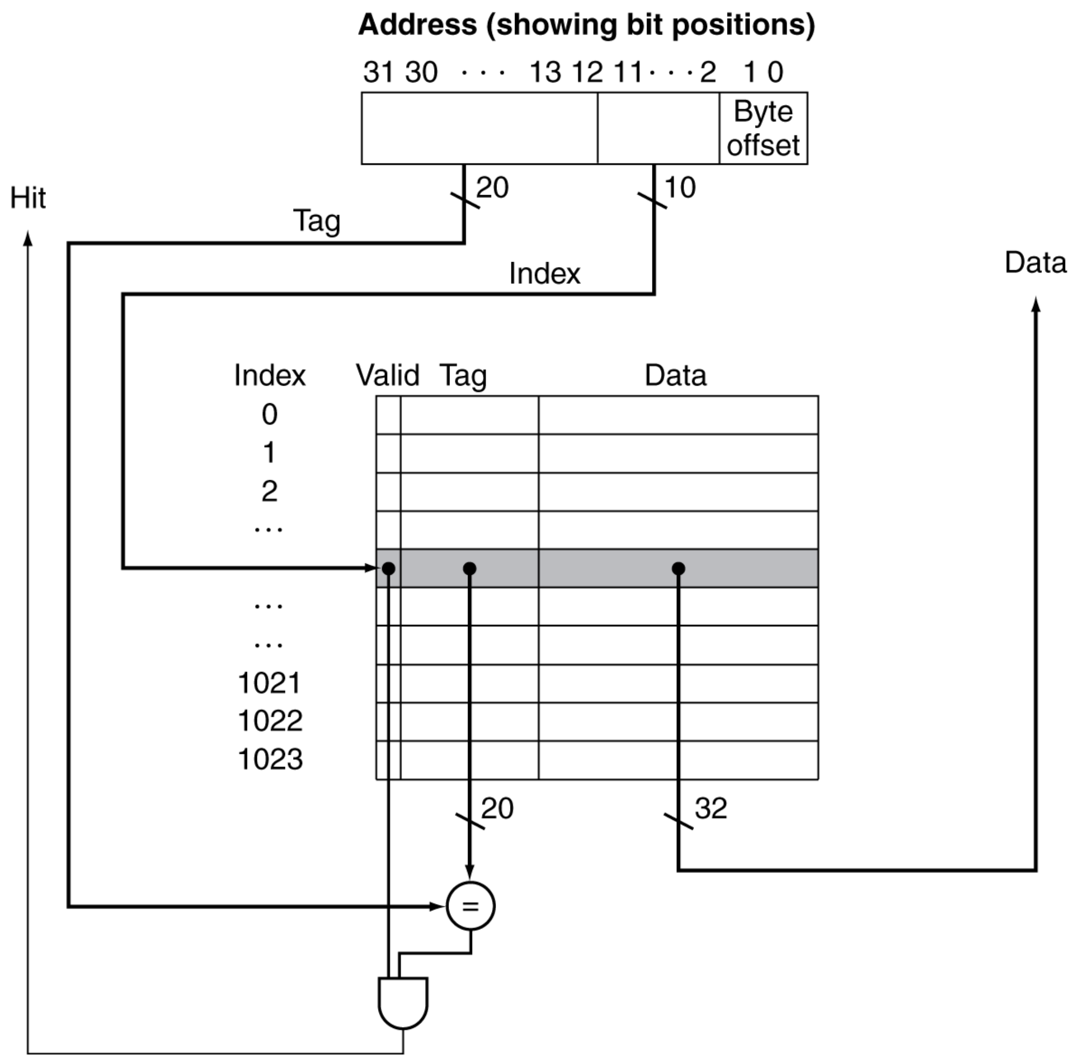
Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-Mapped Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

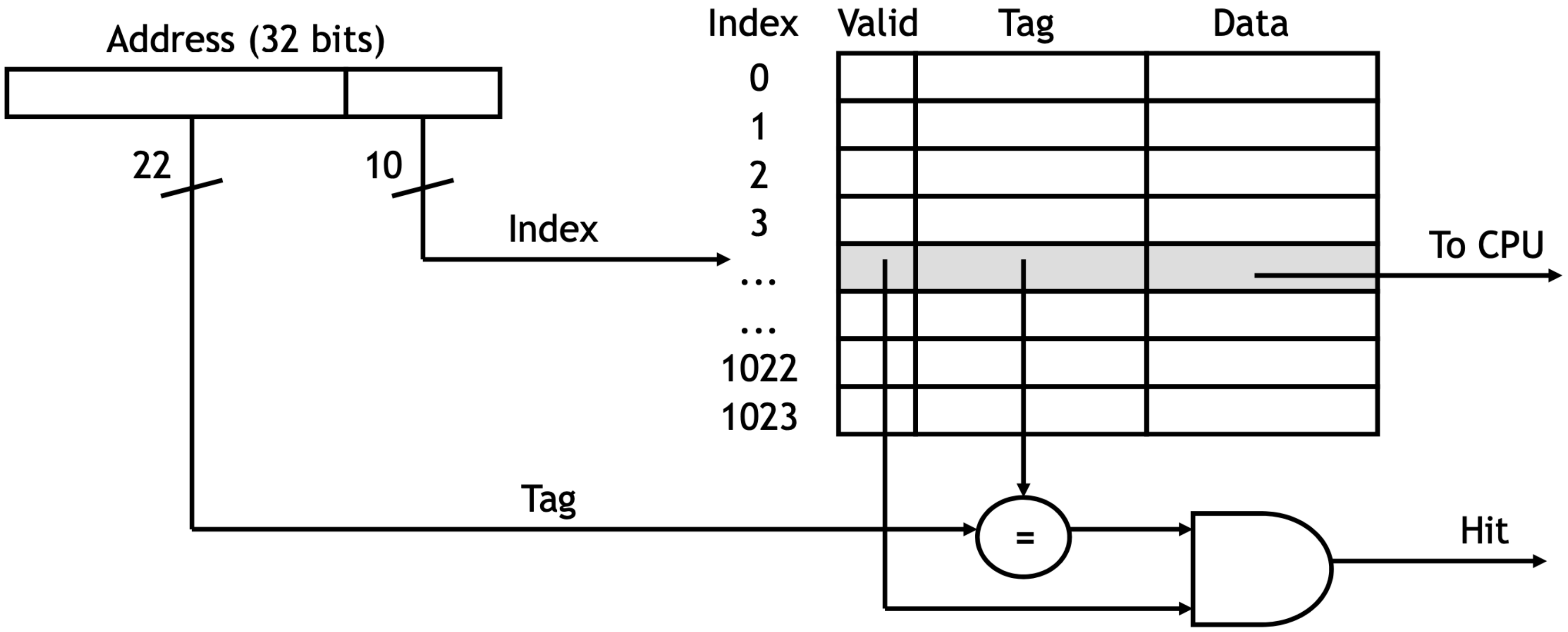
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Direct-Mapped Cache in MIPS



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 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

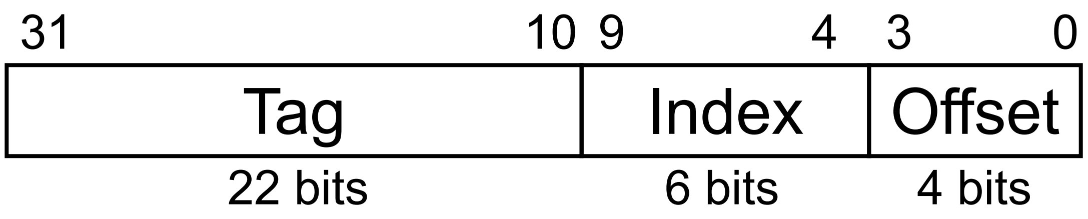


What Happens on a Cache Miss

- **The delays that we've been assuming for memories (e.g., 2ns) are really assuming cache hits**
  - If our CPU implementations accessed main memory directly, their cycle times would have to be much larger
  - Instead we assume that most memory accesses will be cache hits, which allows us to use a shorter cycle time
- **However, a much slower main memory access is needed on a cache miss**
  - The simplest thing to do is to **stall the pipeline** until the data from main memory can be fetched (and also copied into the cache)

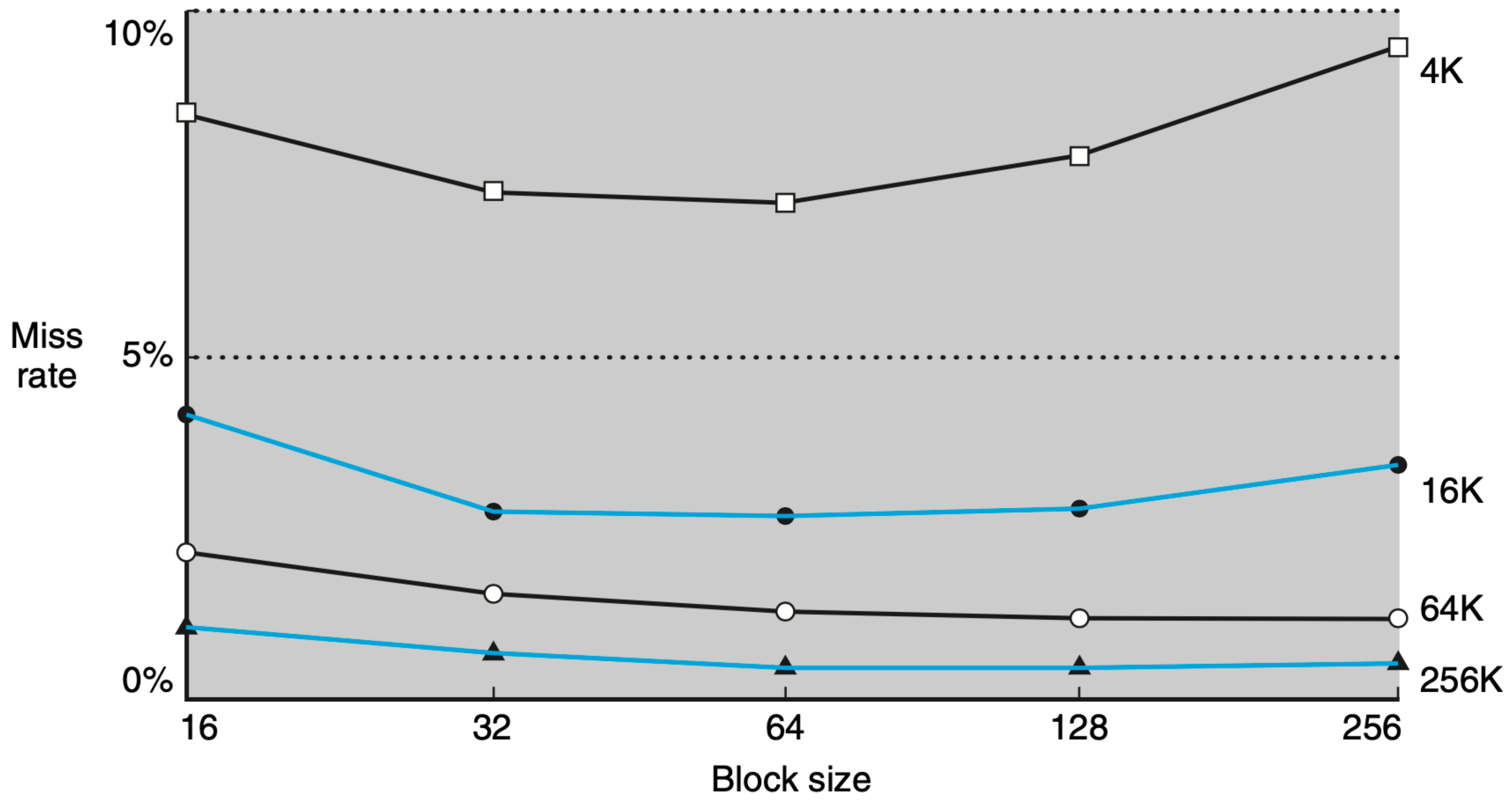
Increasing Block Size

- **Take advantage of spacial locality to lower miss rates**
  - If it takes a long time to process a miss, bring in as much information as you can when you do miss



- **But in a fixed-sized cache**
  - Larger blocks $\Rightarrow$ fewer of them
    - More competition $\Rightarrow$ increased miss rate

Increasing Block Size



Increasing Block Size

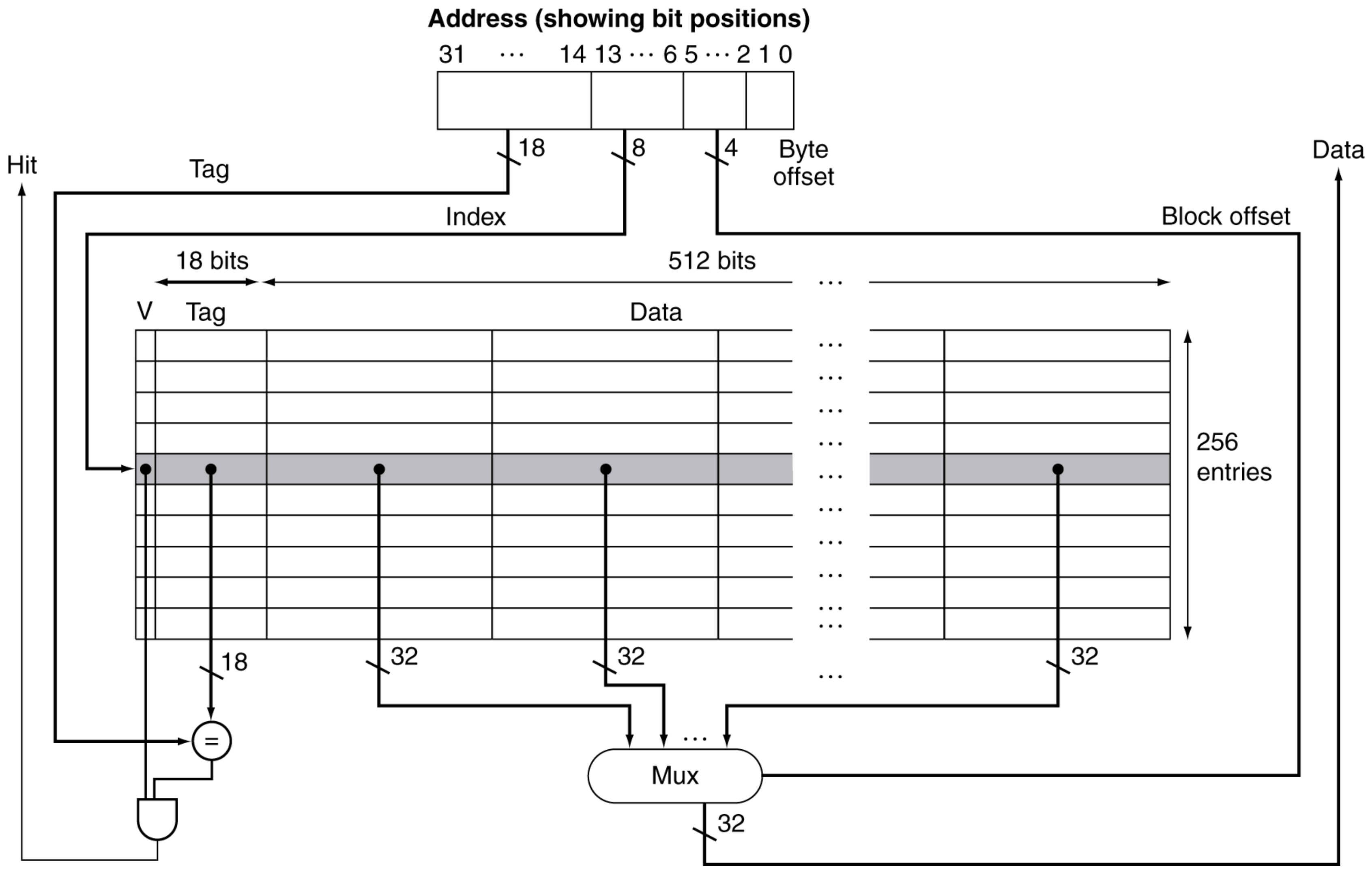
- **May also lead to a larger miss penalty**
  - Latency to the first word and the transfer time for the rest of the block
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

- **Design memory to transfer larger blocks more efficiently**
  - Common methods for increasing bandwidth to DRAM are making the memory wider and interleaving
  - We discuss this topic in the next lecture

Example: Intrinsity FastMATH

- **Embedded MIPS processor**
  - 12-stage pipeline
  - Instruction and data access on each cycle
- **Split cache: separate I-cache and D-cache**
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- **SPEC2000 miss rates**
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

Example: Intrinsity FastMATH



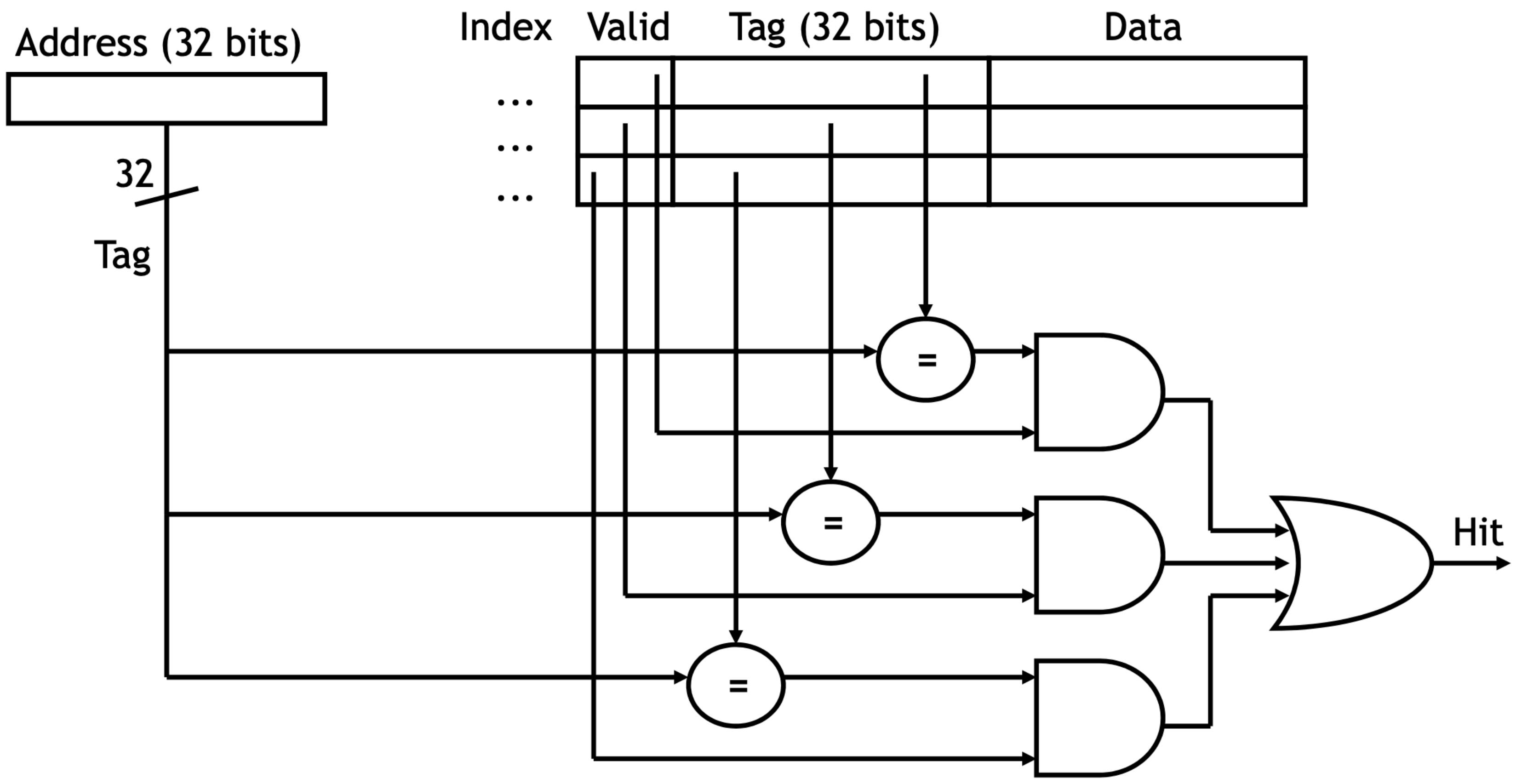
Fully Associative Caches

- **Reduce cache misses by a more flexible placement of blocks**

- **Allow a given block to go in any cache entry**
  - When data is fetched from memory, it can be placed in *any* unused block of the cache
  - No more conflicts

- **Requires all entries to be searched at once**
  - Comparator per entry (expensive)
  - Practical only for caches with a small number of blocks

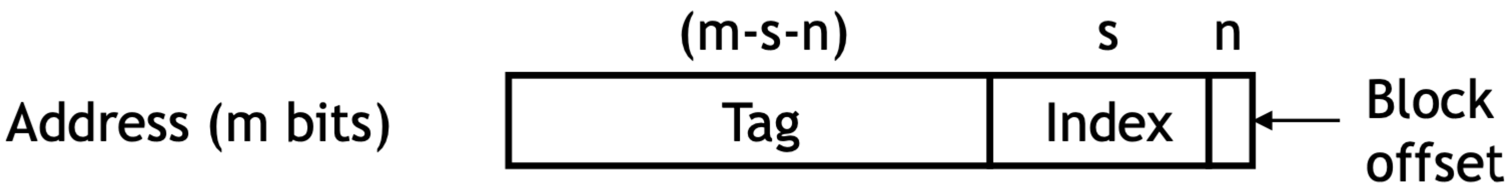
Fully Associative Caches



N-way Set Associative Caches

- **Cache is divided into *groups* of blocks, called sets**
  - Each set contains *n* entries
  - Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set

- **If a cache has 2^s sets and each block has 2^n bytes, the memory address can be partitioned as follows:**

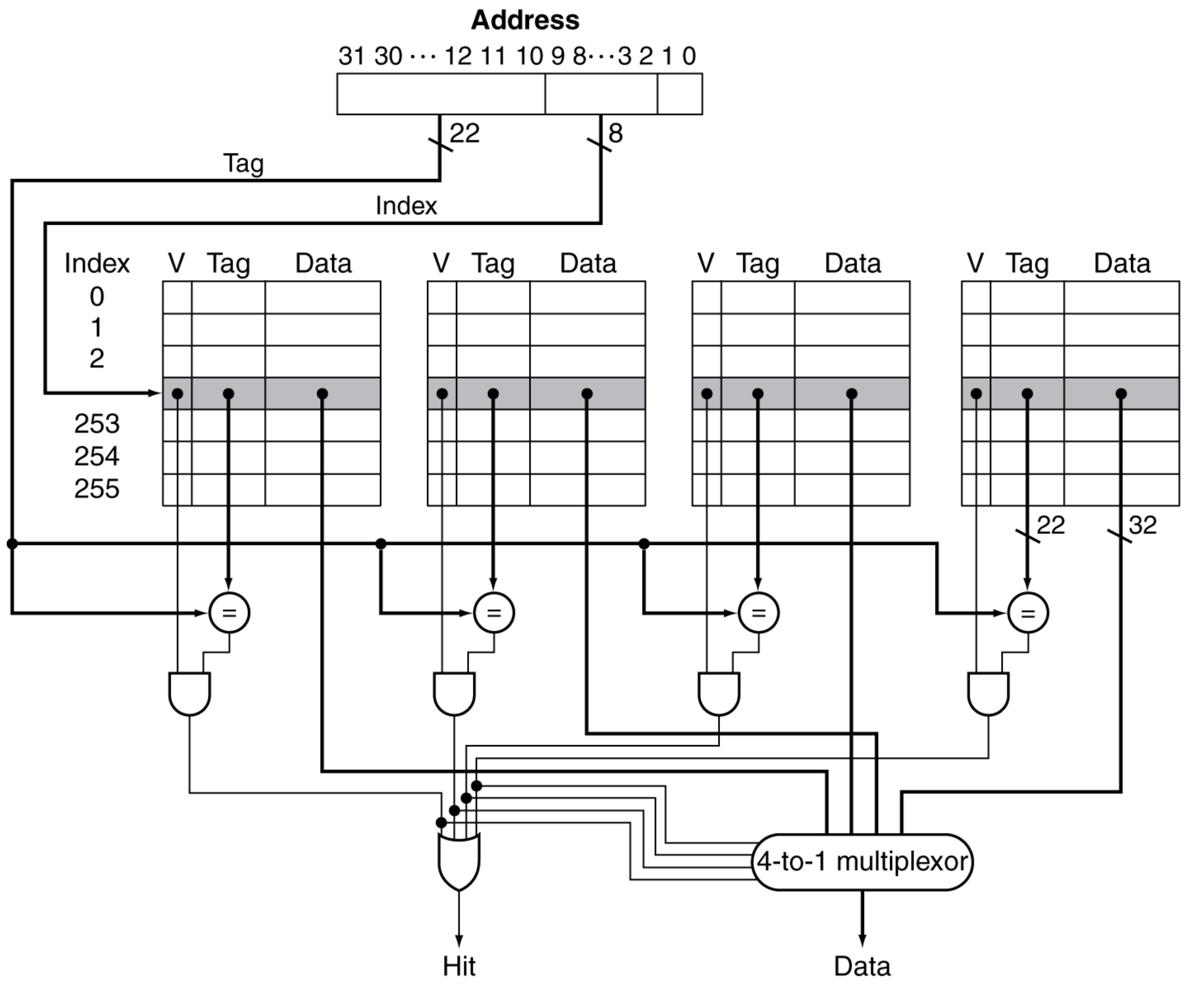


N-way Set Associative Caches

- **Compute a set index to select a set within the cache instead of an individual block**
  - Block offset = (Memory Address) modulo (2^n)
  - Block address = (Memory Address) / (2^n)
  - Set index = (Block Address) modulo (2^S)

- **Search all entries in a given set at once**
  - *n* comparators (less expensive)

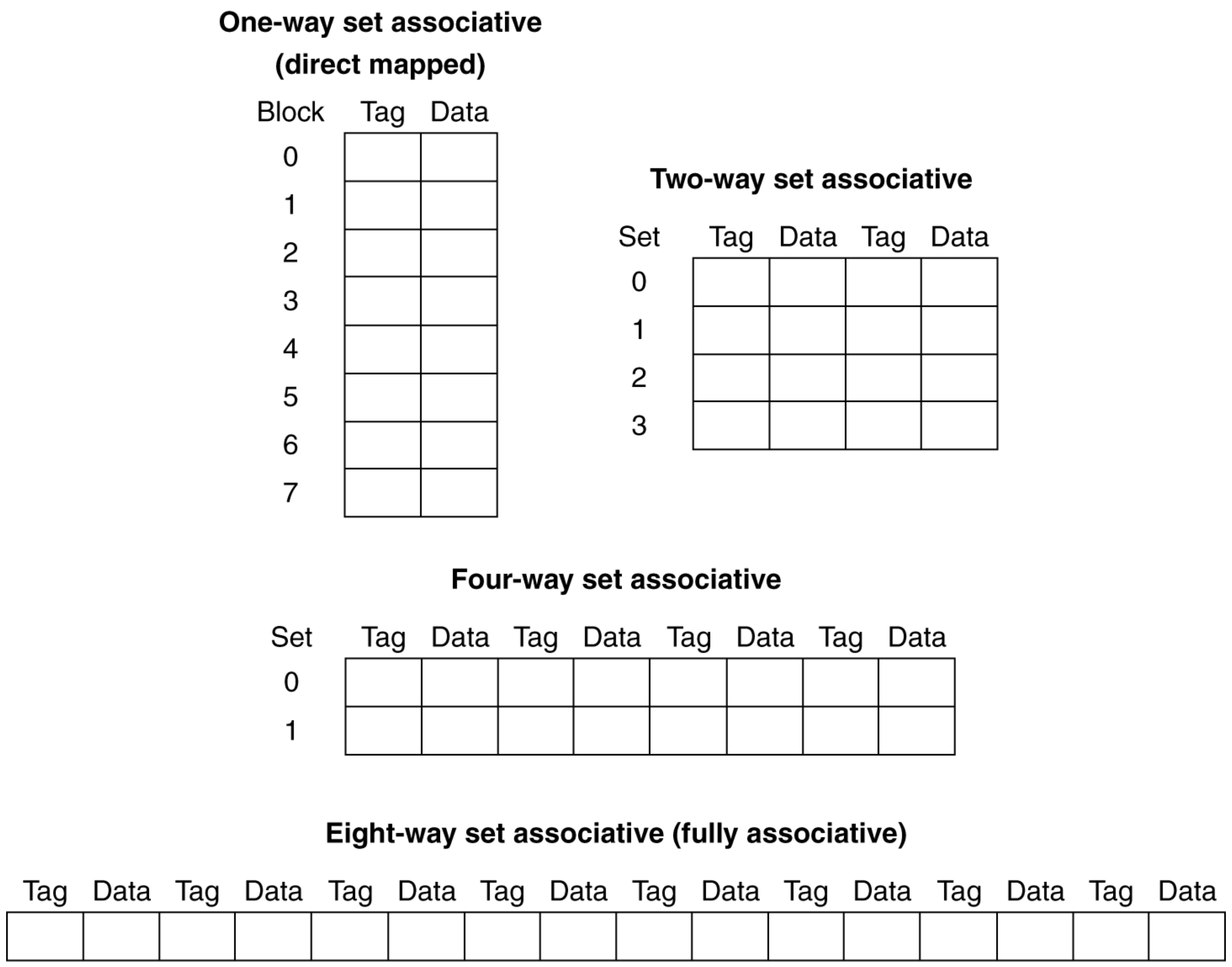
4-way Set Associative Cache



Spectrum of Associativity

- **A direct-mapped cache is simply a one-way set-associative cache**
- **In a cache with 8 blocks, an eight-way set-associative cache is the same as a fully associative cache**
- **The total size of the cache in blocks is equal to the number of sets times the associativity**
  - Thus, for a fixed cache size increasing the associativity decreases the number of sets while increasing the number of elements per set

Spectrum of Associativity



Associativity Example

- **Assume three small caches, each consisting of four one-word blocks**
  - Direct mapped, 2-way set associative, fully associative

- **Find the number of misses for each cache organization given the following sequence of block addresses:**
  - 0, 8, 0, 6, 8

Associativity Example

- **Direct-mapped**
  - (0 modulo 4) = 0
  - (6 modulo 4) = 2
  - (8 modulo 4) = 0

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	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]	

- **Generates five misses for the five accesses**

Associativity Example

- **2-way set associative**
  - (0 modulo 2) = 0
  - (6 modulo 2) = 2
  - (8 modulo 2) = 0
  - Assume LRU for block replacement (will discuss this shortly)

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]		

- **Generates four misses, one less than the direct-mapped cache**

Associativity Example

- **Fully associative**
  - Any memory block can be stored in any cache block

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]	

- **Has the best performance, with only three misses**
  - Because three unique block addresses are accessed
  - Notice that if we had eight blocks in the cache, there would be no replacements in the two-way set-associative cache
  - Similarly, if we had 16 blocks, all 3 caches would have the same number of misses

How Much Associativity

- **Increased associativity decreases miss rate**
  - But with diminishing returns

- **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%

What if the Cache Fills Up?

- **What to do if we run out of space in 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

- **Each replacement policy is a compromise between hit rate and latency**
  - More efficient replacement policies keep track of more usage information in order to improve the hit rate (for a given cache size)
  - Faster replacement strategies typically keep track of less usage information to reduce the amount of time required to update that information

Replacement Policy

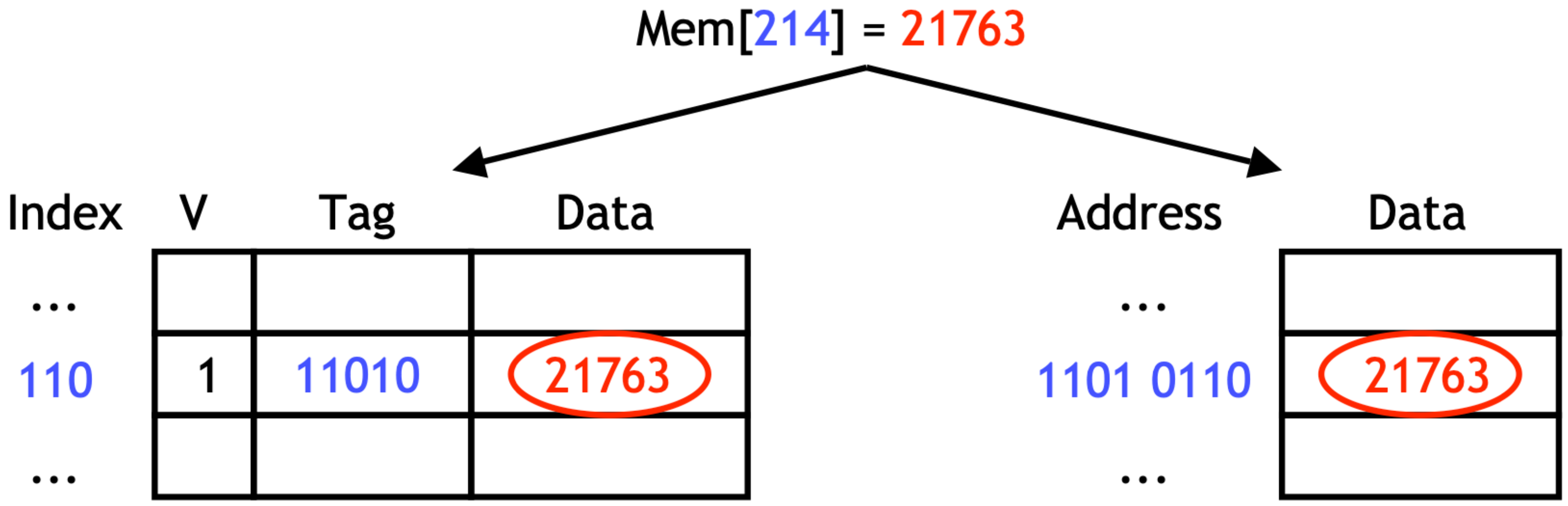
- **Direct mapped caches**
  - No choice, but keeps no information
- **N-way set associative caches**
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
  - Least-recently used (LRU)
    - Choose the one unused for the longest time
      - Simple for 2-way, manageable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

Handling Writes

- **Writes introduce several complications into caches that are not present for reads**
- **Write-hit**
  - If the destination address is present in the cache
  - Two policies
    - Write-through
    - Write-back
- **Write-miss**
  - If the write occurs to a location that is not present in the cache
  - Two policies
    - Write allocate
    - No-write allocate (write around)

Write-Through

- **Forces all writes to update both the cache *and* the main memory**
  - Ensuring that data is always consistent between the two

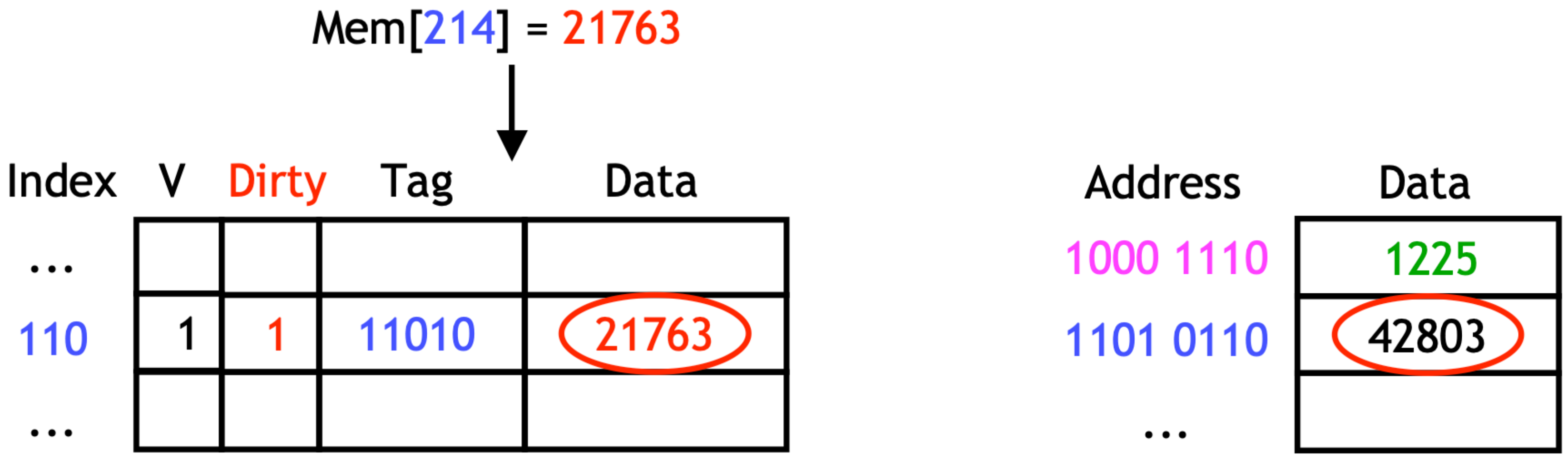


Write-Through

- **Simple to implement and keeps the cache and memory consistent**
- **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

- **Handles writes by updating values only to the block in the cache**
  - The memory is not updated until the cache block needs to be replaced (*e.g.*, when loading data into a full cache set)



Index	V	Dirty	Tag	Data
...				
110	1	1	11010	21763
...				

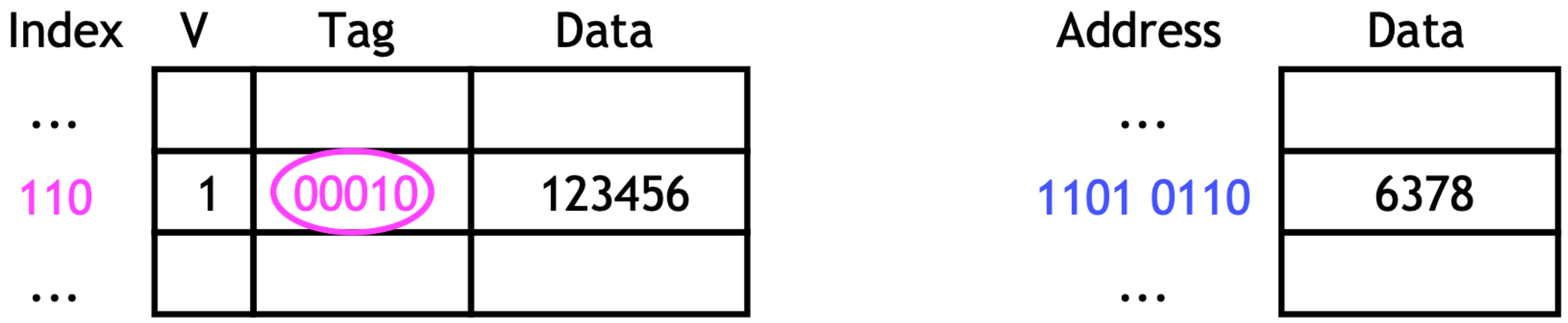
Address	Data
1000 1110	1225
1101 0110	42803
...	

Write-Back

- **To track whether a block has been written since it was read from memory, a dirty bit is added**
- **When a dirty block is replaced**
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first
- **Not all write operations need to access main memory:**
  - If a single address is frequently written to, then it doesn't pay to keep writing that data through to main memory
  - If several bytes within the same cache block are modified, they will only force one memory write operation at write-back time

Write Misses

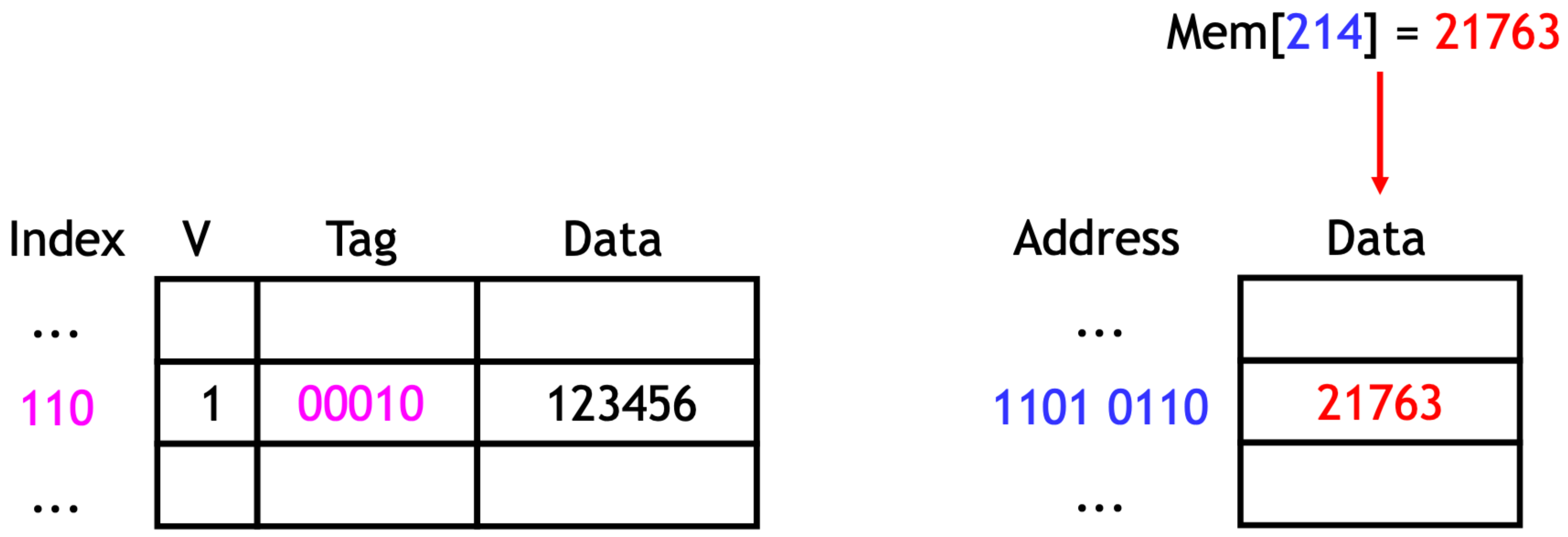
- Let's say we want to store 21763 into Mem[1101 0110] but we find that address is not currently in the cache



- When we update Mem[1101 0110], should we also load it into the cache?

No Write Allocate (Write Around)

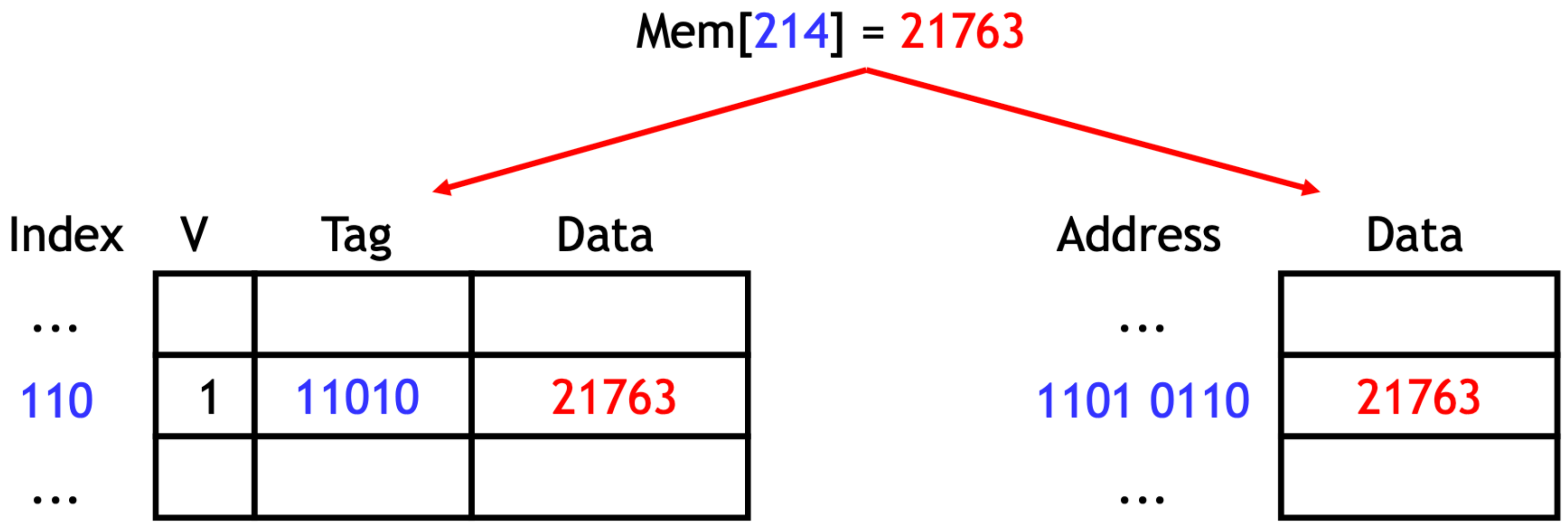
- The write operation goes directly to main memory *without* affecting the cache



- This is good when data is written but not immediately used again
  - In which case there's no point to load it into the cache yet

Write Allocate

- An allocate on write strategy would instead load the newly written data into the cache



- If that data is needed again soon, it will be available in the cache

Usual Combinations

- **Both write-through and write-back policies can use either of these write-miss policies, but usually they are paired this way:**

- **A write-back cache uses write allocate**
  - Hoping for subsequent writes (or even reads) to the same location, which is now cached

- **A write-through cache uses no-write allocate**
  - Here, subsequent writes have no advantage, since they still need to be written directly to memory
  - But programs often write a whole block before reading it (e.g., initialization)

Reducing Miss Penalty

- **Split Instruction/Data caches**
  - No structural hazard between IF & MEM stages
  - A single-ported unified cache stalls fetch during load or store

- **Cache hierarchies**
  - Trade-off between access time and hit rate
    - L1 cache can focus on fast access time (ok hit rate)
      - Small, but fast
    - L2 cache can focus on good hit rate (ok access time)
      - Larger, slower, but still faster than main memory
    - Some systems include a L3 cache
  - Such hierarchical design is another “big idea”

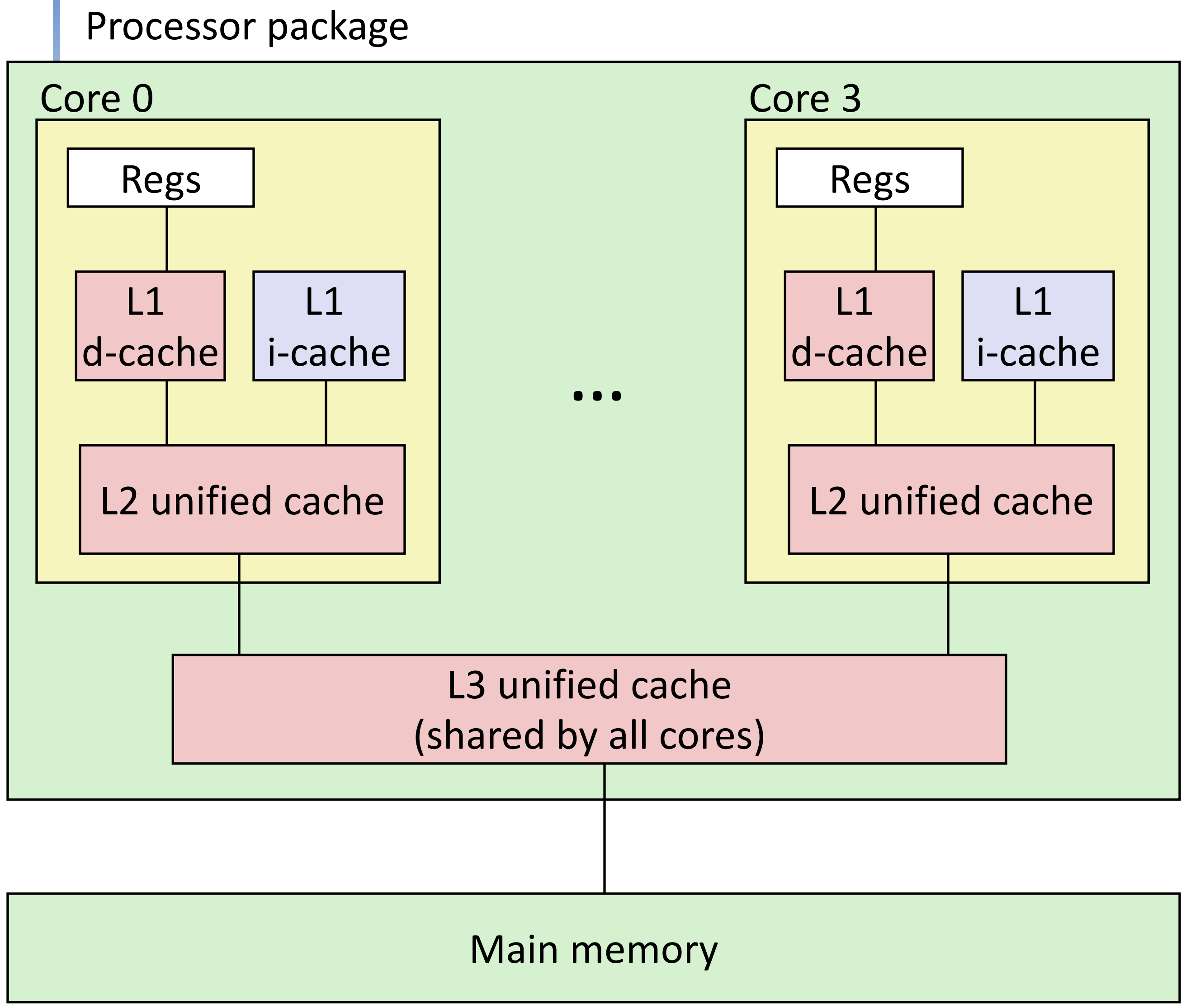
Multilevel Cache Example

- **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%

Example (cont.)

- **Primary miss with L-2 hit**
  - Penalty = 5ns/0.25ns = 20 cycles
- **Primary miss with L-2 miss**
  - Extra penalty = 500 cycles
- **CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4**
- **Performance ratio = 9/3.4 = 2.6**

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches

Concluding Remarks

- **Fast memories are small, large memories are slow**
  - We really want fast, large memories ☹
  - Caching gives this illusion ☺
- **Principle of locality**
  - Programs use a small part of their memory space frequently
- **Memory hierarchy**
  - L1 cache ↔ L2 cache ↔ … ↔ DRAM memory ↔ disk
- **Memory system design is critical for performance**