# Chapter 6:
# Improving Performance via Caching

## Chapter 6 Topics:

- Locality of reference
- Caching in the memory hierarchy
- Cache hit/miss rates
- Cache-friendly code

# Announcements

- Recitation Ex #4 available, due Mon Nov 10 in recitation
  - Will release solutions before midterm
- Midterm #2 on Tuesday Nov 11
- Performance Lab due Mon Nov 17 by 8 am
- Essential that you read the textbook in detail & do the practice problems
  - Read Chapter 6, all sections

# Recap…

## Limits on Optimization

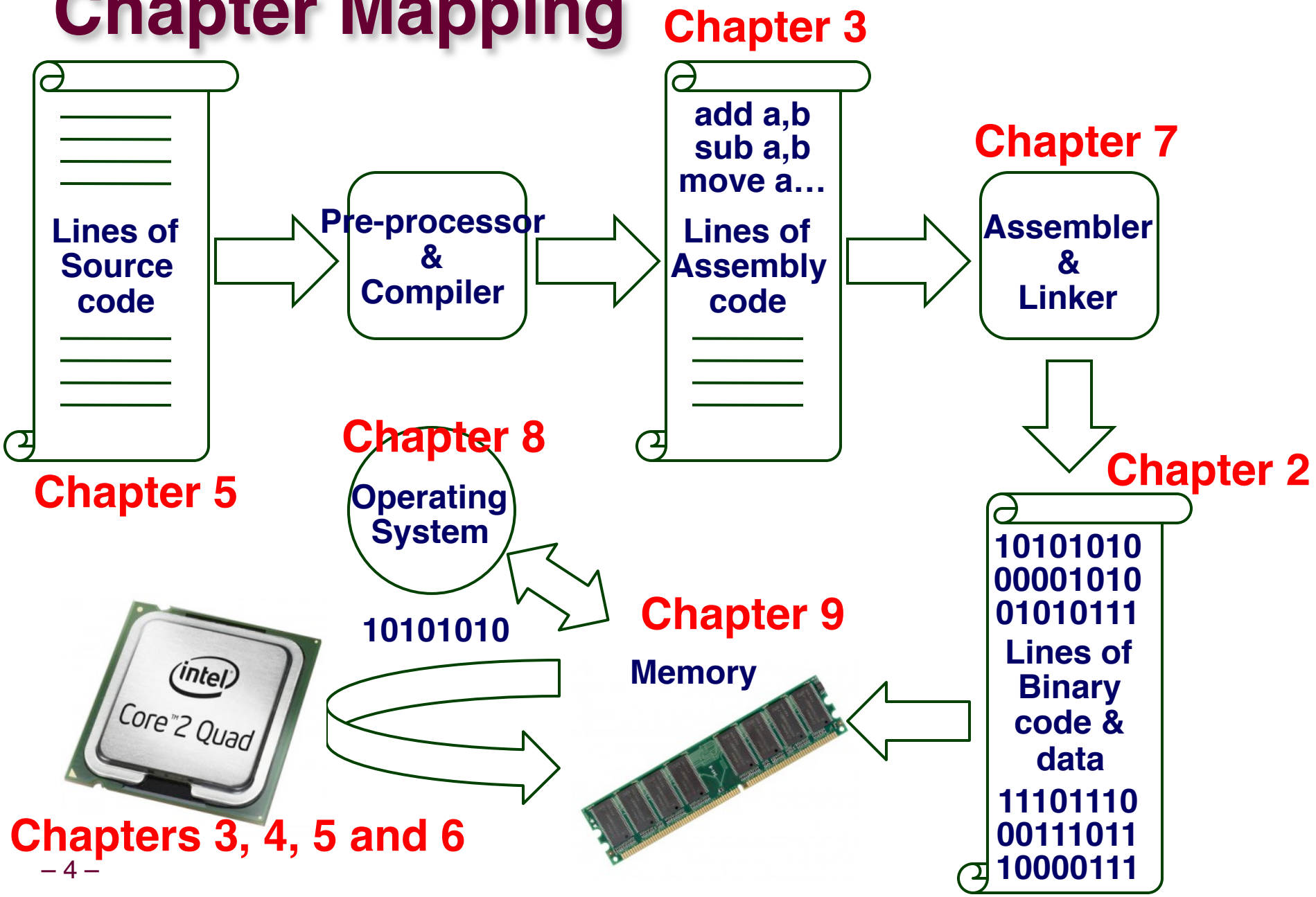
1. register spilling
2. branch misprediction
3. load/store dependencies
4. Amdahl's Law
5. limited # of functional units
6. limited degree of pipelining

## Profiling code – find the bottleneck and leaks using tools like gprof

Sort words in a text document from most frequent to least

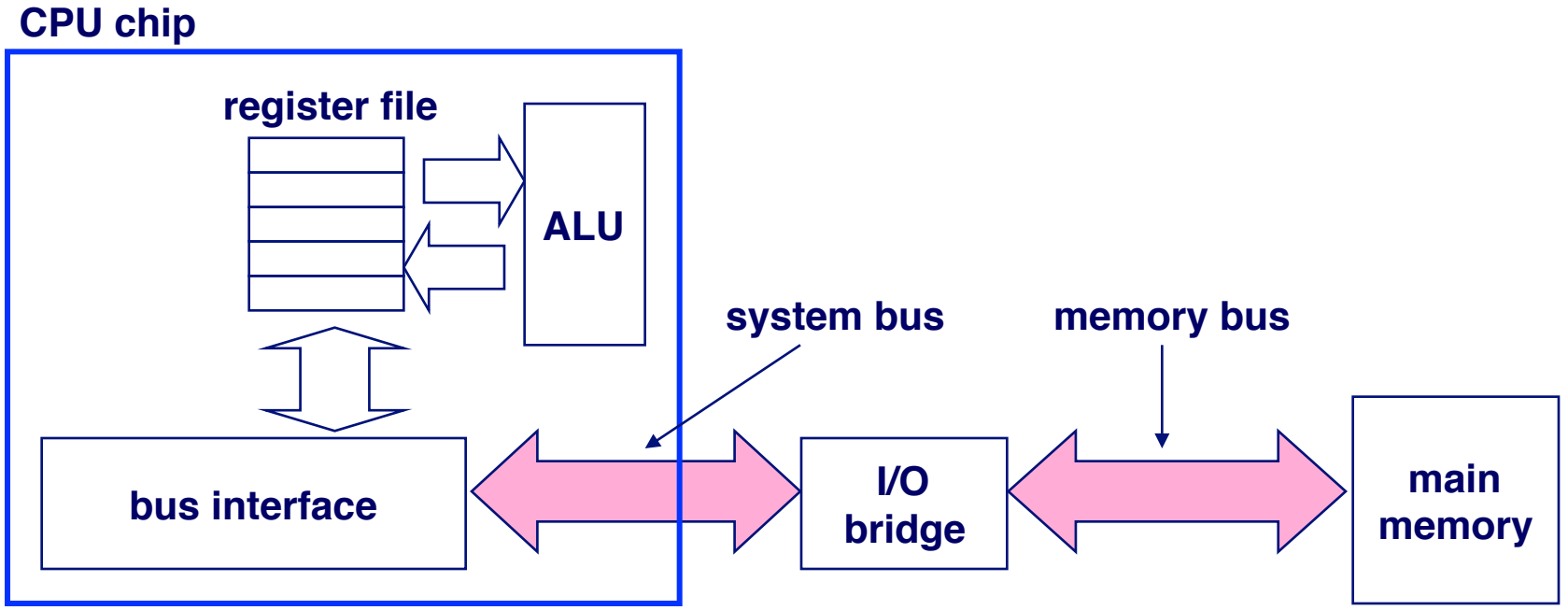
Four stage approach: Lower -> Hash -> Search Linked List -> Sort

# Chapter Mapping



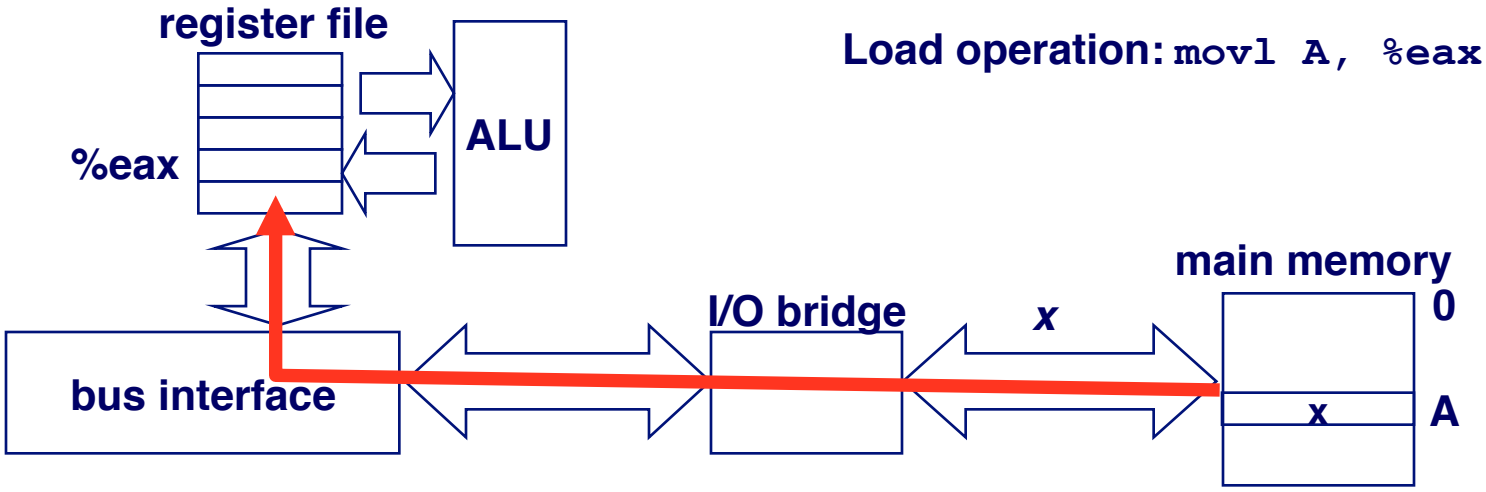
# Typical Computer Organization Connecting CPU and Memory

- CPU and main memory are connected by a communication bus.
- A bus is a collection of parallel wires that carry address, data, and control signals.



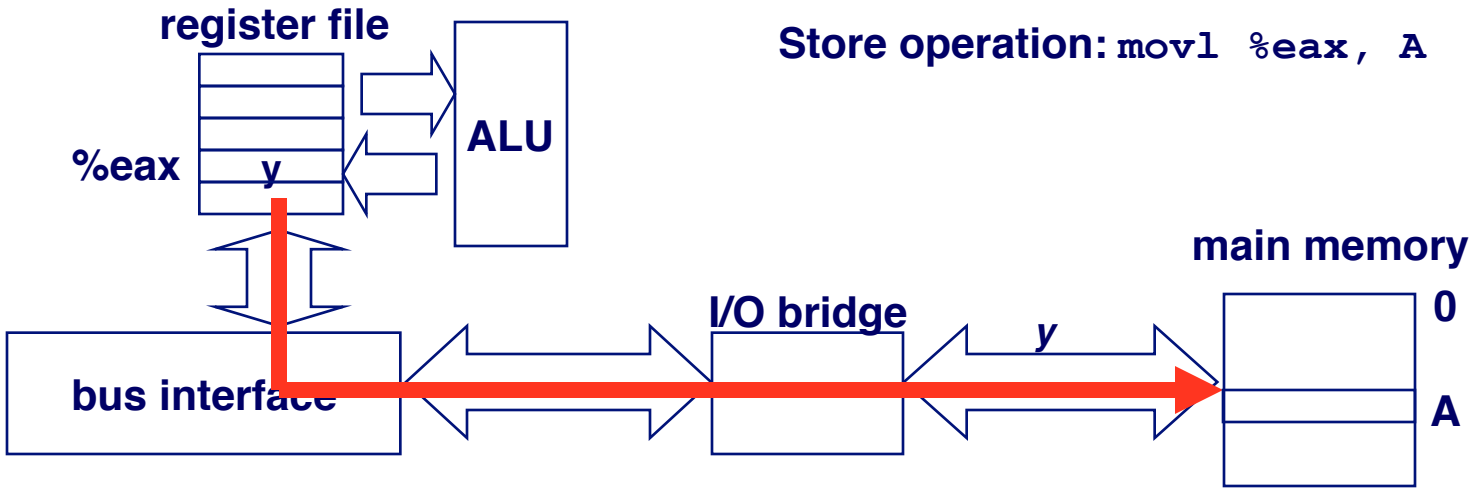
# Memory Read Transaction

- Main memory reads address for A from the memory bus, retrieves word x (A's value), and places it on the bus.



# Memory Write Transaction

- CPU places data word y on the bus and address of A to store A's value in memory location for A.



# Random-Access Memory (RAM)

## Key features

- RAM is packaged as a chip.
- Basic storage unit is a cell (one bit per cell).
- Multiple RAM chips form a memory.
- Volatile storage – loss of power causes loss of stored bits

## Static RAM (SRAM)

- Each cell stores bit with a six-transistor circuit.
- Retains value indefinitely, as long as it is kept powered.
- Relatively insensitive to disturbances such as electrical noise.
- Faster and more expensive than DRAM.

## Dynamic RAM (DRAM)

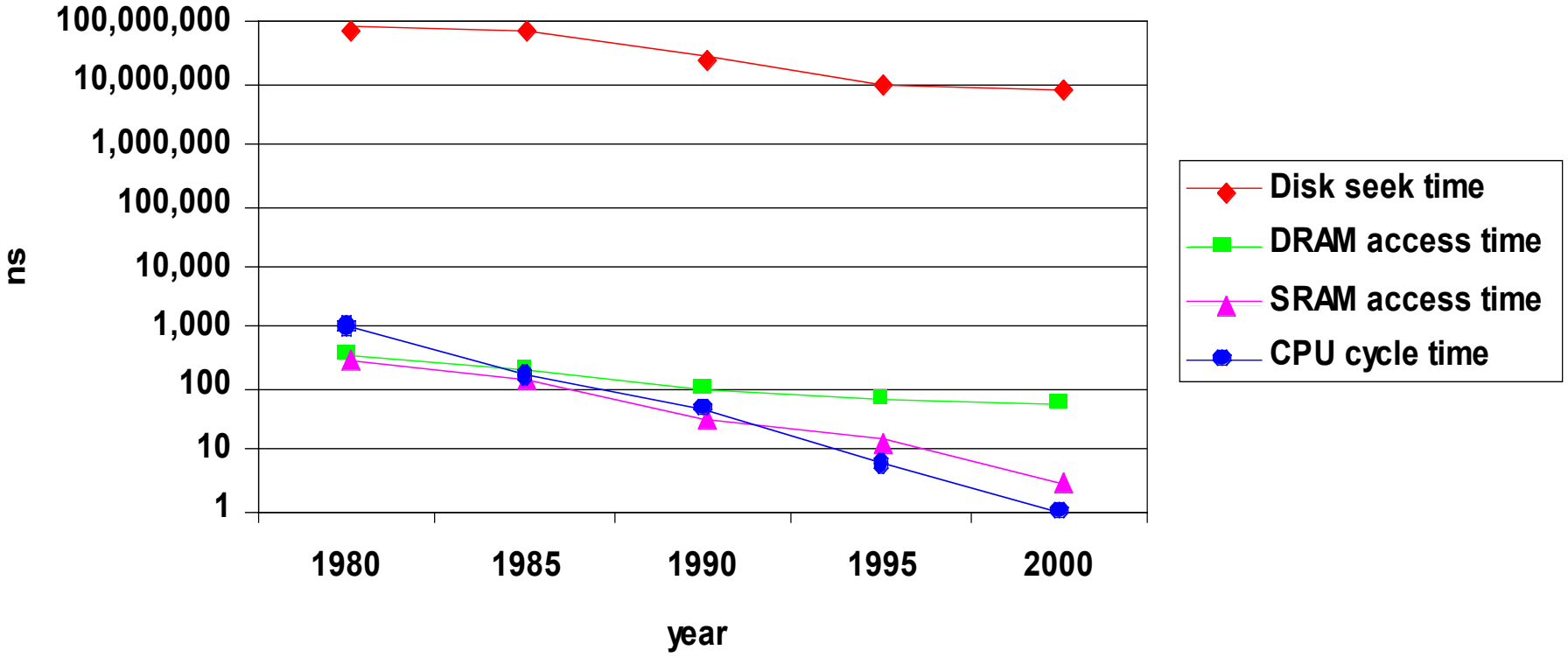
- Each cell stores bit with a capacitor and transistor.
- Value must be refreshed every 10-100 ms.
- Sensitive to disturbances.
- Slower and cheaper than SRAM.

# SRAM vs DRAM Summary

| | Tran. per bit | Access time | Persist? | Sensitive? | Cost | Applications |
|---|---|---|---|---|---|---|
| SRAM | 6 | 1X | Yes | No | 100x | cache memories |
| DRAM | 1 | 10X | No | Yes | 1X | Main memories, frame buffers |

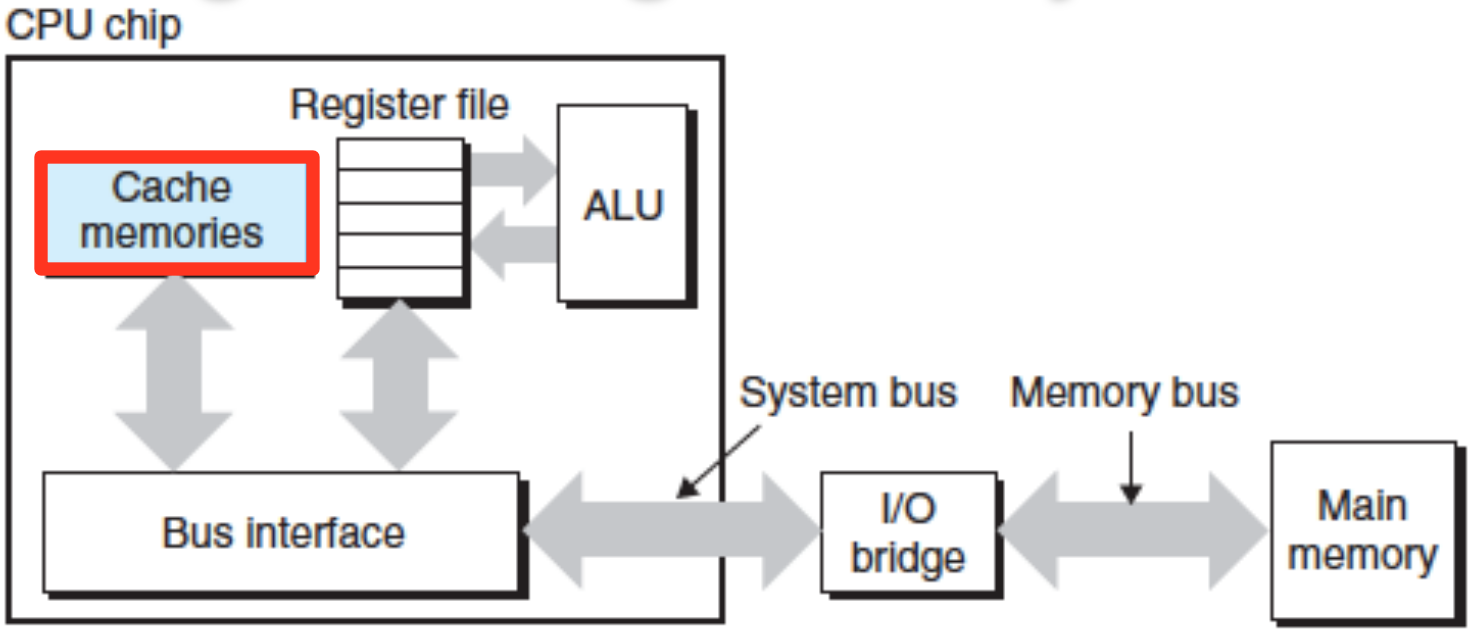
# The CPU-Memory Gap

**The increasing gap between DRAM, disk, and CPU speeds.**




**DRAM is inexpensive but slow, SRAM is 100X more expensive but 10X faster**
**So use DRAM for main memory, and SRAM for smaller caches in CPU**

# Adding Caching Memory to CPU




- Store commonly accessed data and instructions in CPU caches
- These caches have faster access times, higher throughput, but cost more, so can't be as large as main memory
- Benefit: Faster average access times, but only if the access patterns for data and instructions exhibit enough *locality*
- Want *a high cache hit rate, and a low cache miss rate*

# Locality

## Principle of Locality:

- Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.

- Temporal locality: Recently referenced items are likely to be referenced in the near future.
  - e.g. instructions in a loop, or data variables referenced in a loop

- Spatial locality: Items with nearby addresses tend to be referenced close together in time.
  - e.g. array elements a[i] and a[i+1] tend to be referenced in sequence, or instructions like “pop %epb” and “ret” tend to be referenced in sequence

# Locality Example

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

## Data:

- Temporal: `sum` referenced in each iteration
- Spatial: array `a[]` accessed in memory in succession, i.e. via a *stride-1 pattern (sequential in row-major order)*

## Instructions:

- Temporal: cycle through loop repeatedly
- Spatial: reference instructions in memory in sequence, looping repeatedly

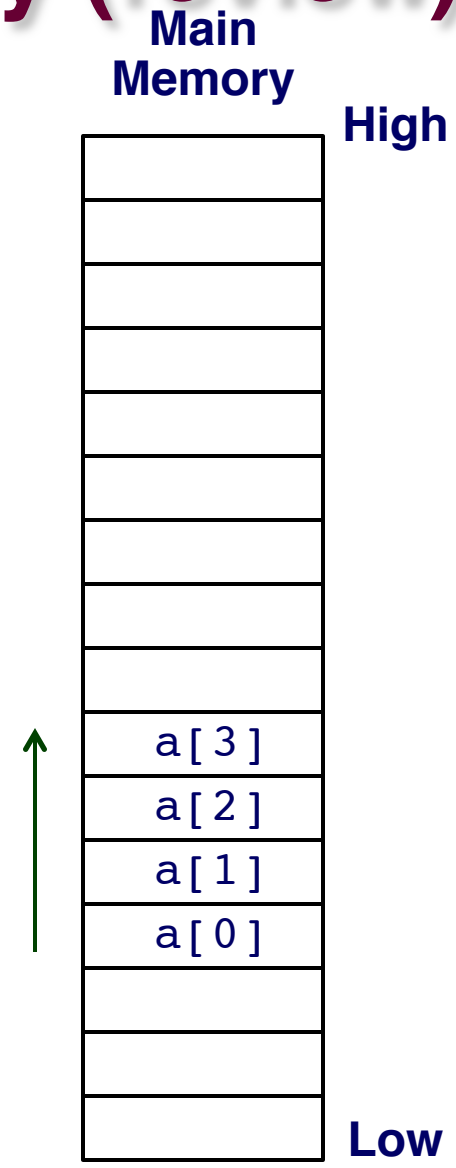
# Layout of Arrays in Memory (review)

## Arrays allocated sequentially

- each following element is in a higher contiguous memory location
- e.g. if `a[]` is an int array, then `a[i+1]` is located 4 bytes higher than `a[i]`, etc.

## Accessing the array sequentially in the same order it is allocated

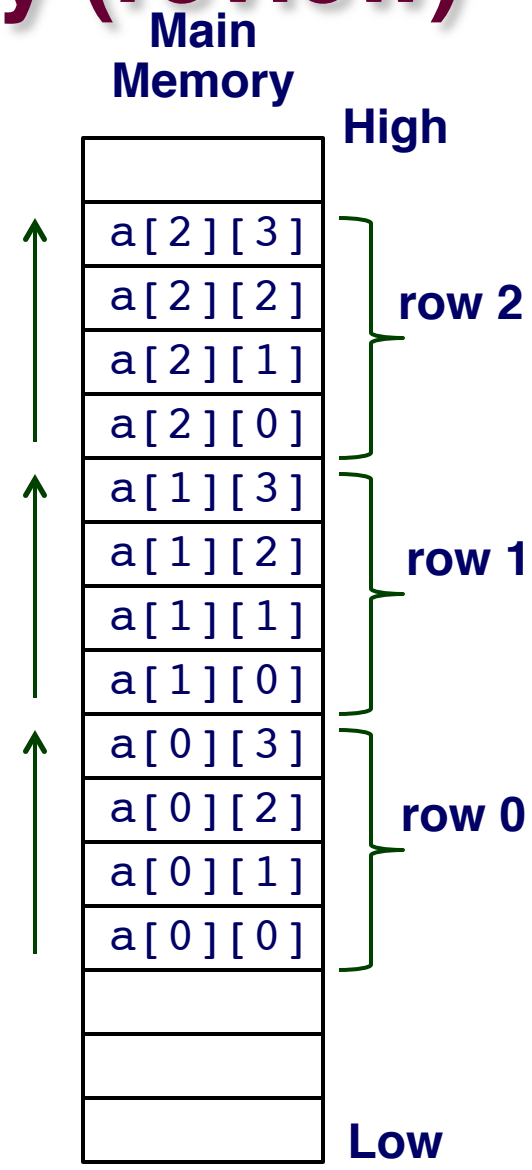
- `for (i = 0; i < N; i++)`
  `sum += a[i];`
- accesses successive elements in memory, column by column
- Stride-1 access pattern



# Layout of Arrays in Memory (review)

## Multi-dimensional arrays allocated sequentially in *row-major order*

- Lay out the first row first, then the second row next, followed by the third row, etc.
- Within each row, lay out each subsequent element (column) in a higher contiguous memory location
- e.g. if `a[][]` is a 2-D int array, then
  - `a[i][j+1]` is located 4 bytes higher than `a[i][j]`
  - `a[i+1][j]` is located 4*N bytes higher than `a[i][j]`, where N=# columns



# Locality Example (2)

**Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

This function reads a[] in row-major order, as it is laid out in memory, and has good spatial locality

- Inner loop reads elements of 1st row in order, then 2nd, etc.
- Stride-1 reference pattern

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum
}
```

# Locality Example (3)

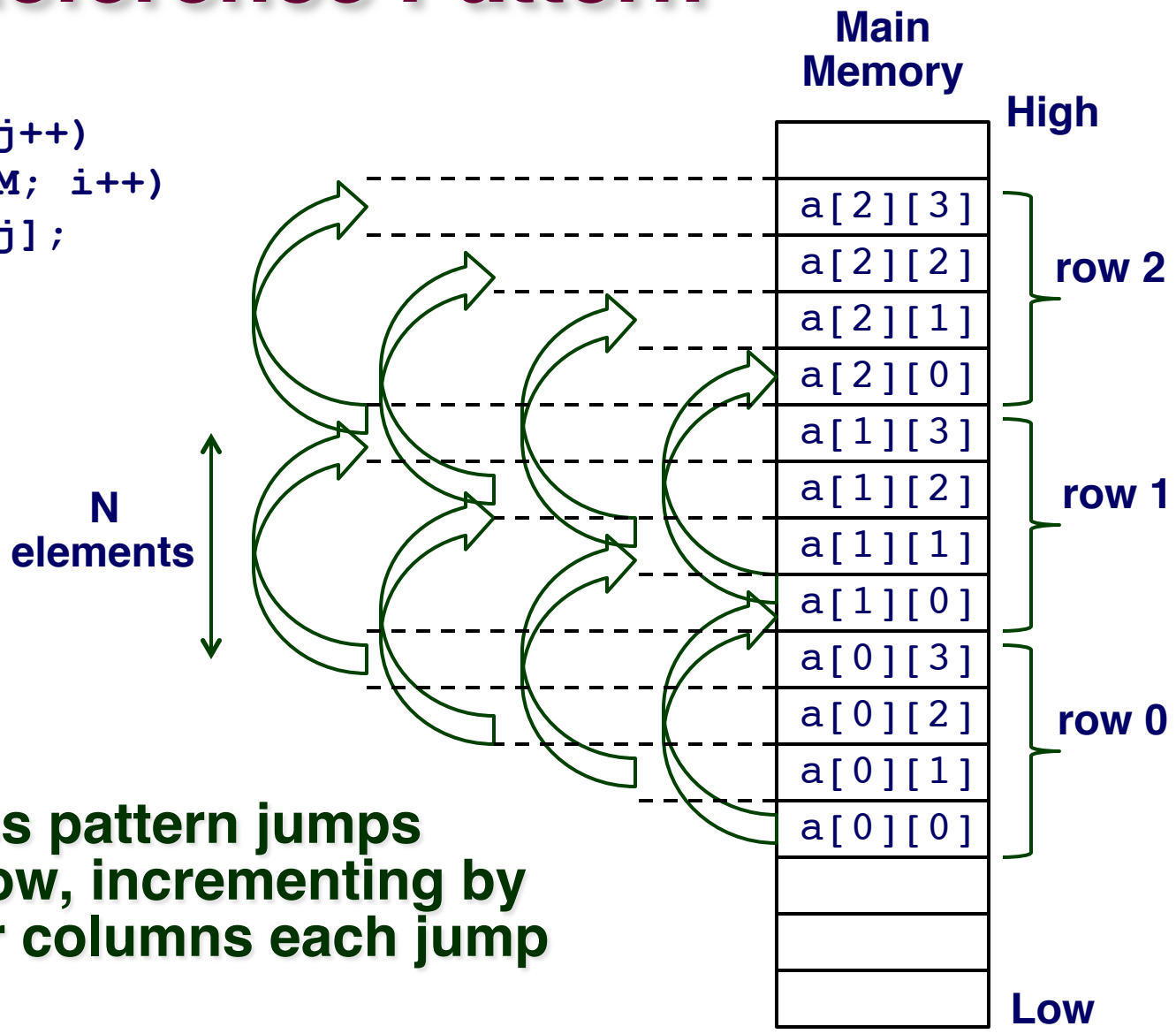
## Scans the array column-wise instead of row-wise

- Does this change the value of the sum?
  - No
- This exhibits a *Stride-N reference* pattern: a[0][j], then a[1][j], then a[2][j] – each successive memory location in the inner loop is N int's apart
  - This function exhibits poor spatial locality - can't take advantage of the fact that data is fetched as a *block* of contiguous nearby data words and then is cached

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum
}
```

# Stride-N Reference Pattern

```
for (j = 0; j < N; j++)
    for (i = 0; i < M; i++)
        sum += a[i][j];
```




**a Stride-N access pattern jumps from row to row, incrementing by N elements or columns each jump**

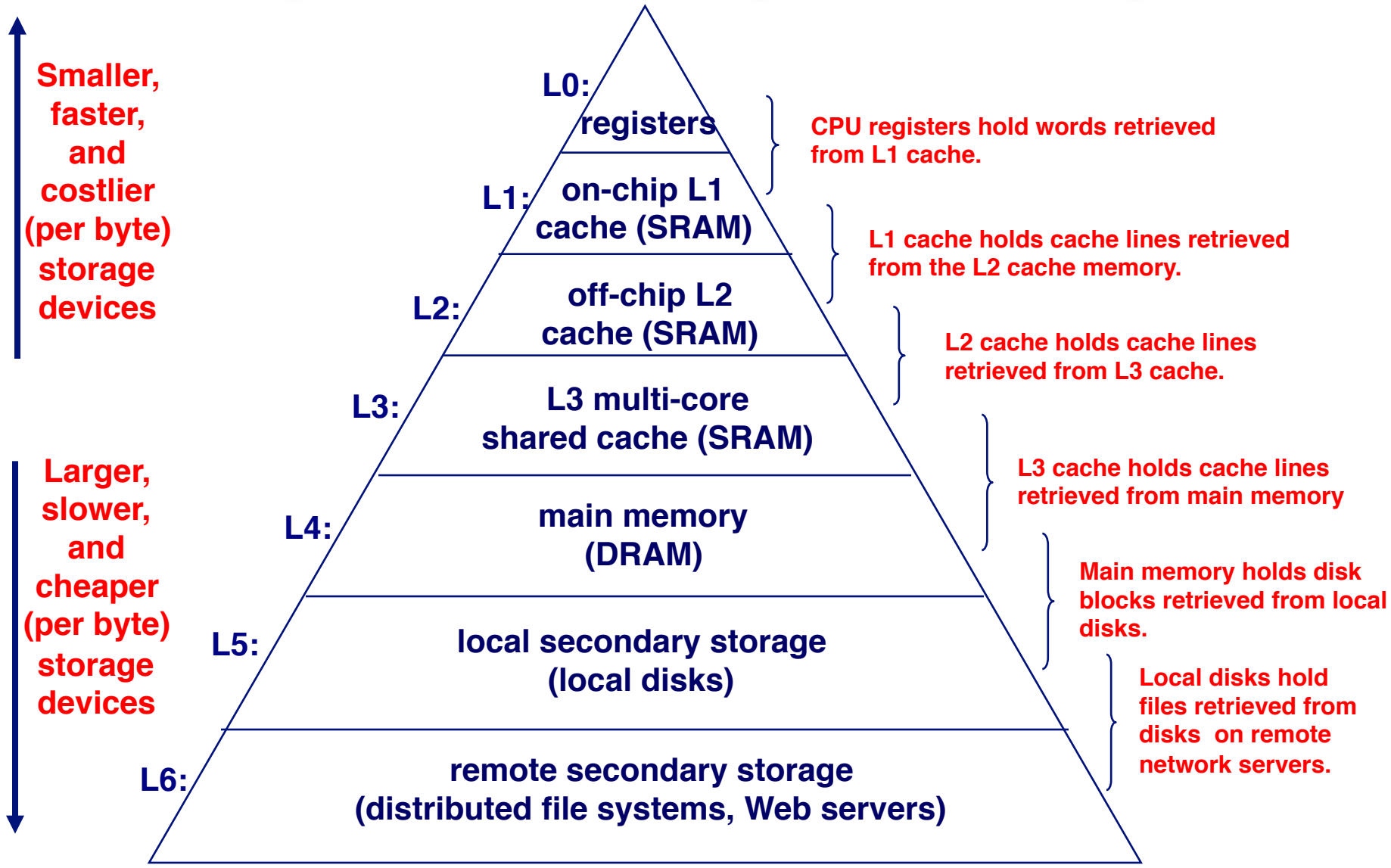
# Locality Example (4)

**Question:** Can you permute the loops so that the function scans the 3-d array `a[]` with a stride-1 reference pattern (and thus has good spatial locality)?

```
int sumarray3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum
}
```

**Answer: k on outside loop, i on middle loop, j on inner loop**

# A Computer Memory Hierarchy



# 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.

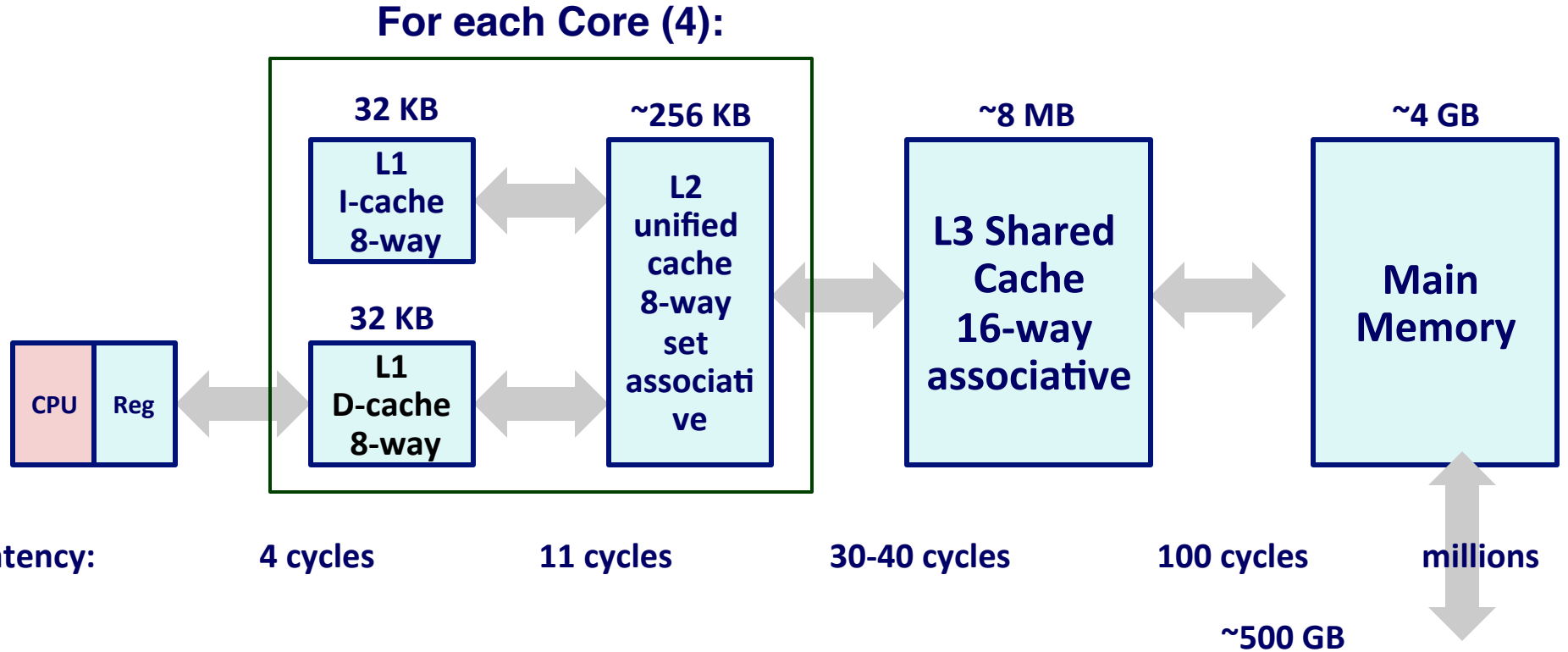
## Why do memory hierarchies work?

- 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.
- Net effect: A large pool of memory that costs only slightly more than the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

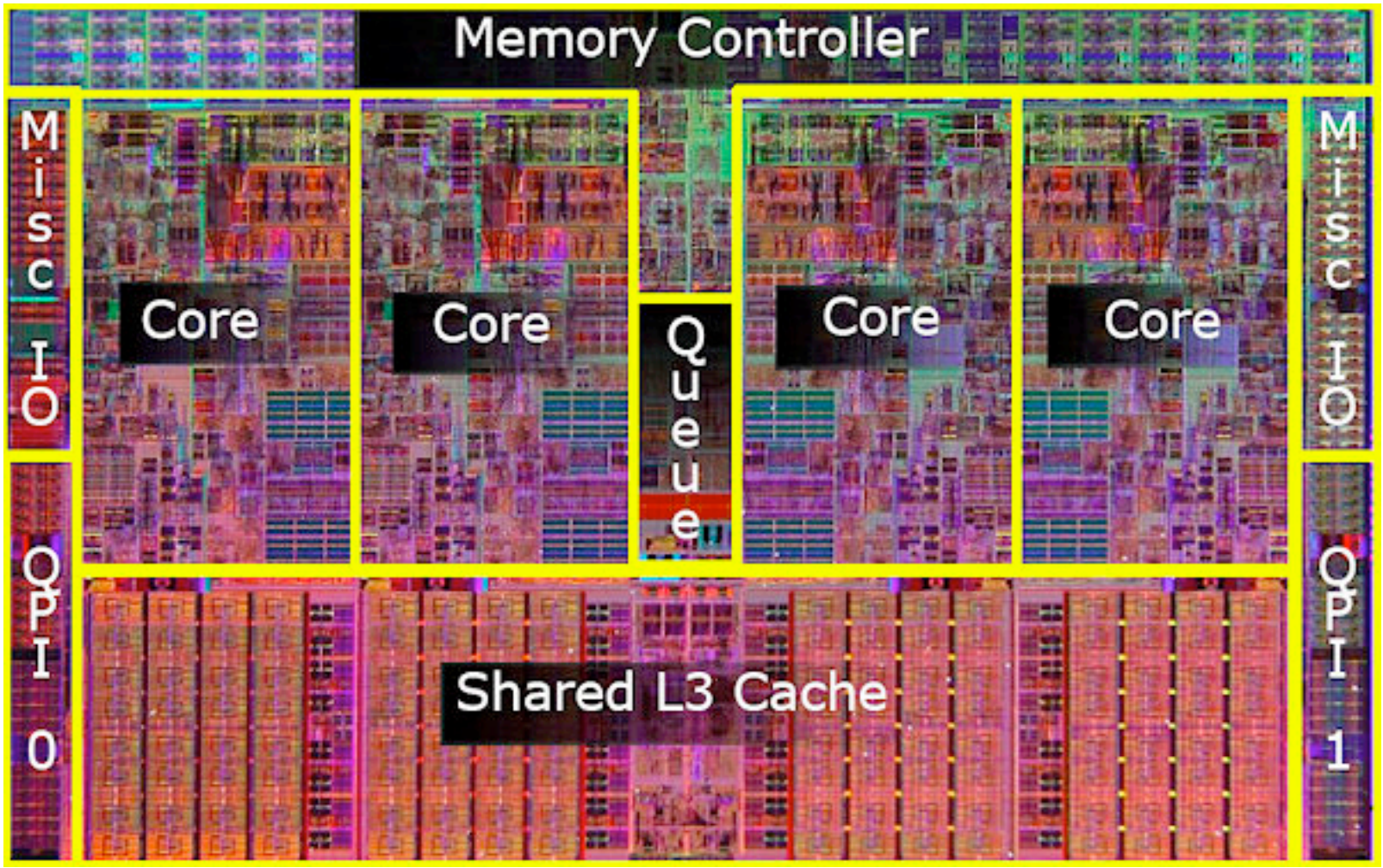
# Memory Hierarchy: Intel Core i7

*Not drawn to scale*

L1/L2/L3 caches: 64 B blocks



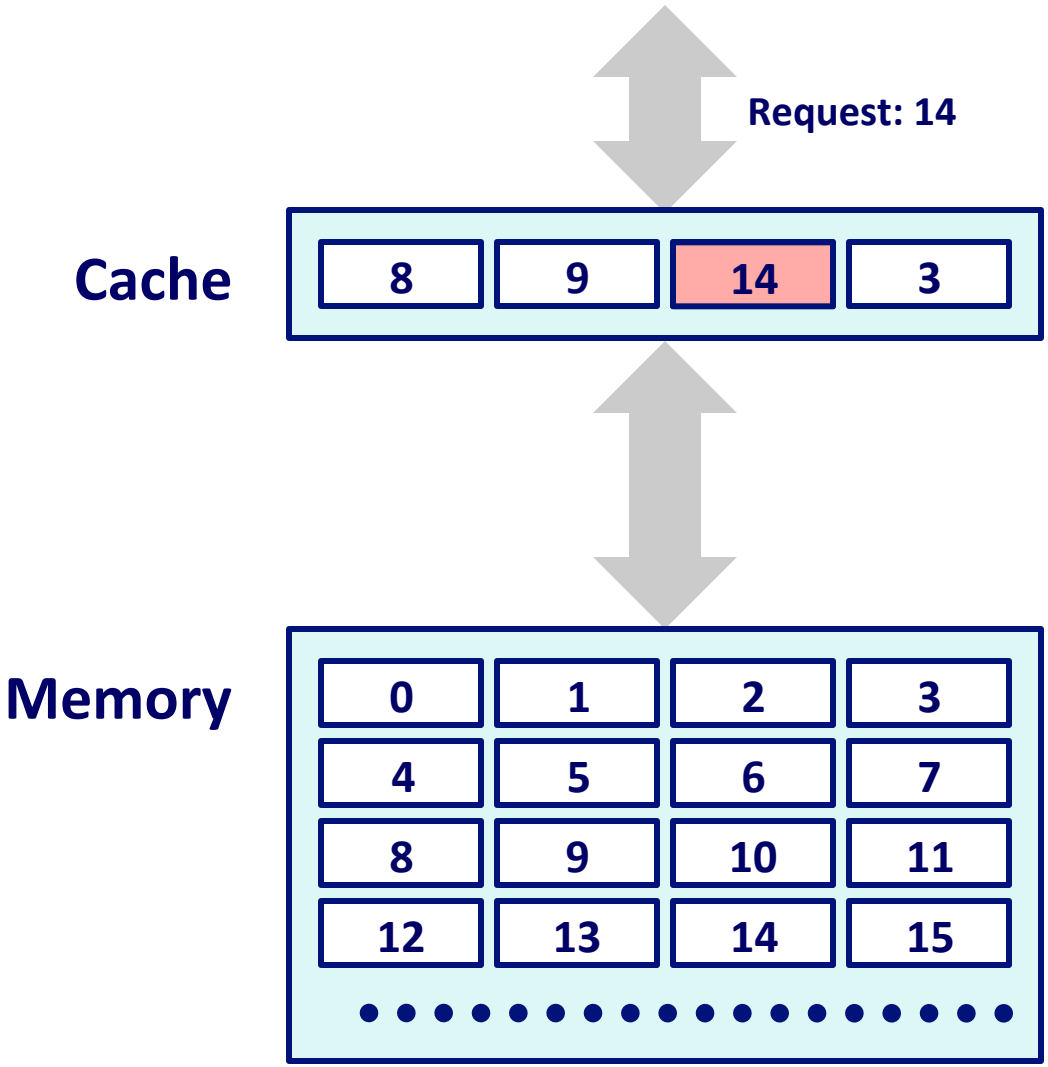
# Intel Core i7






circa 2008

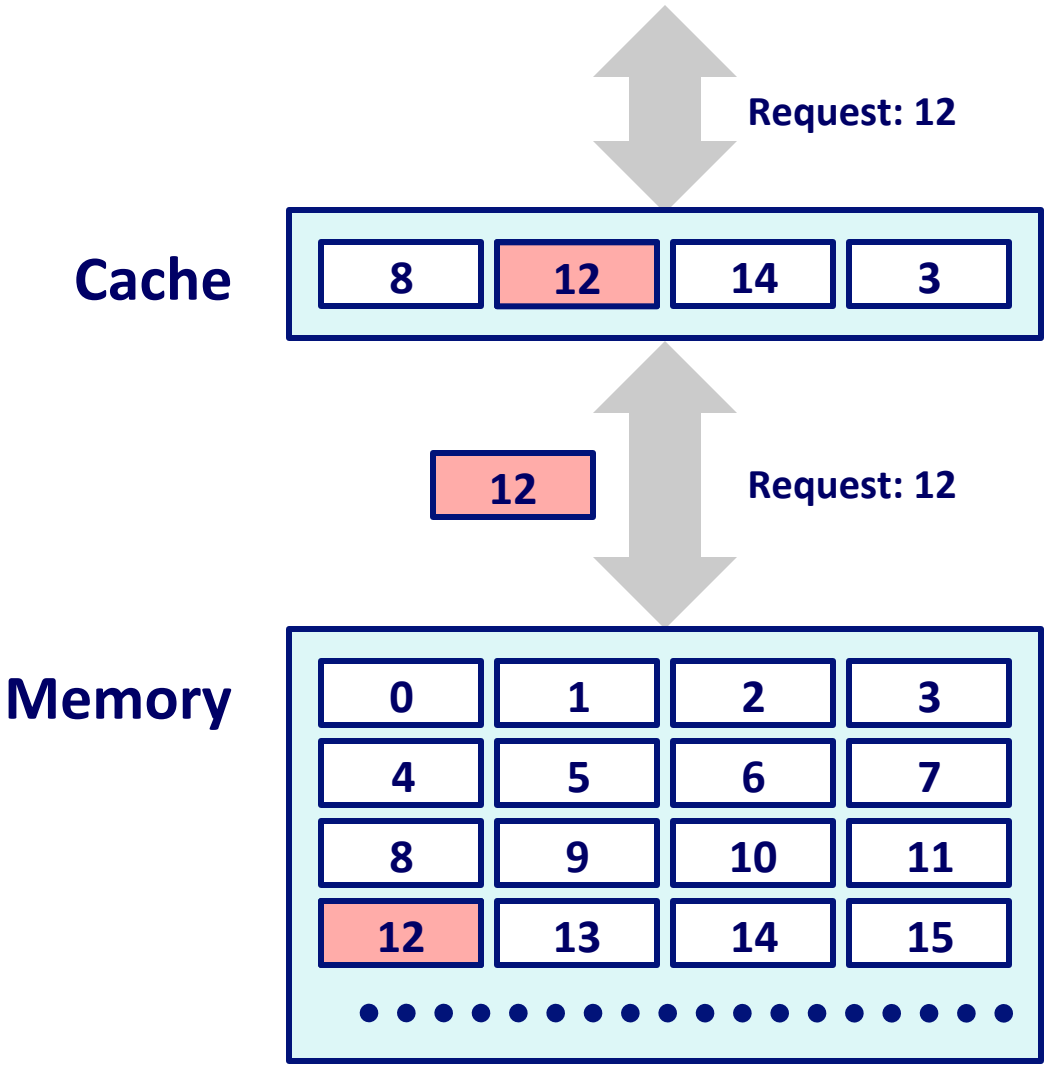
# General Cache Concepts: Hit




*Data in block b is needed*

*Block b is in cache:*
*Hit!*

# General Cache Concepts: Miss




***Data in block b is needed***

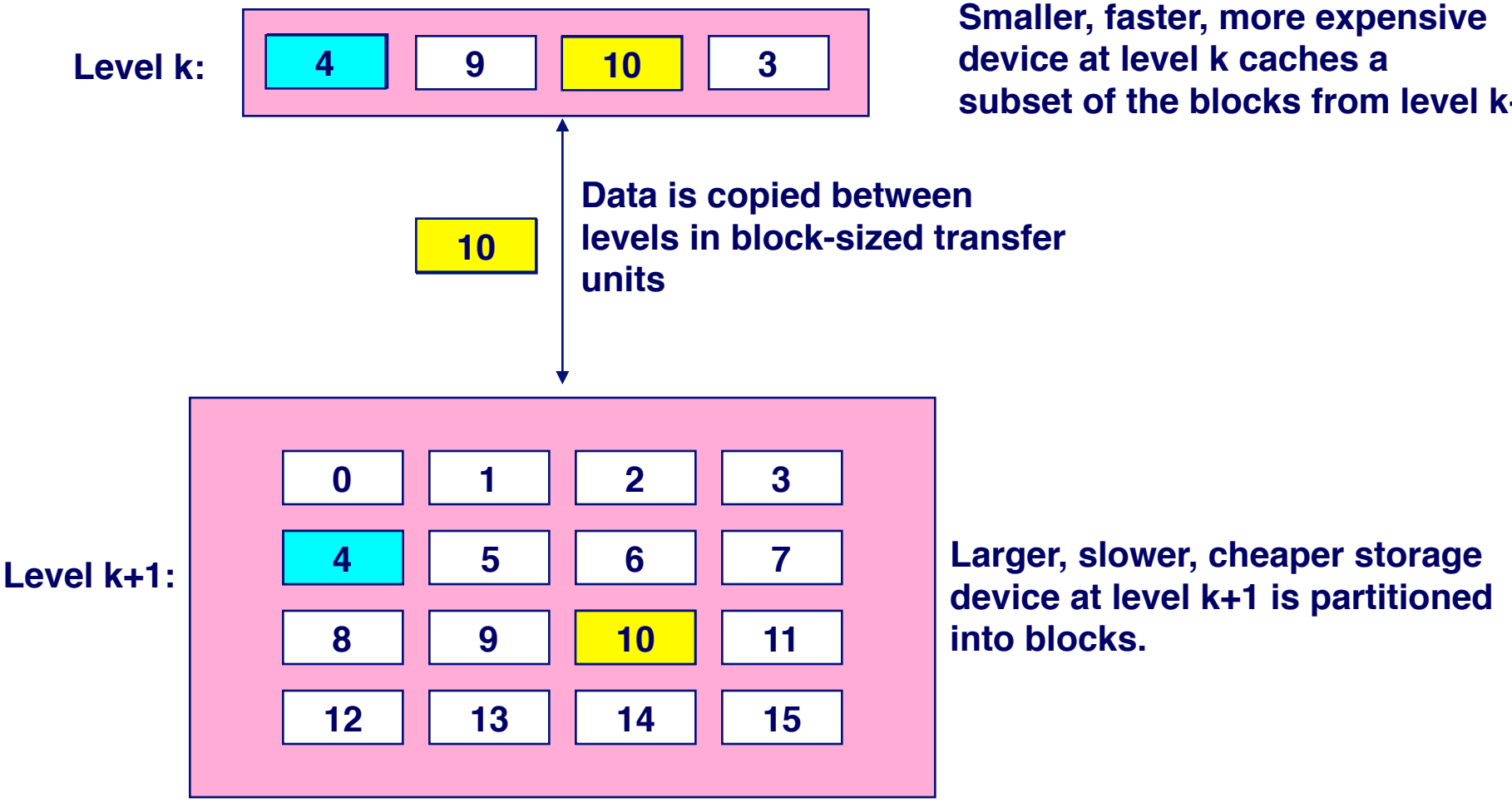
***Block b is not in cache: Miss!***

***Block b is fetched from memory***

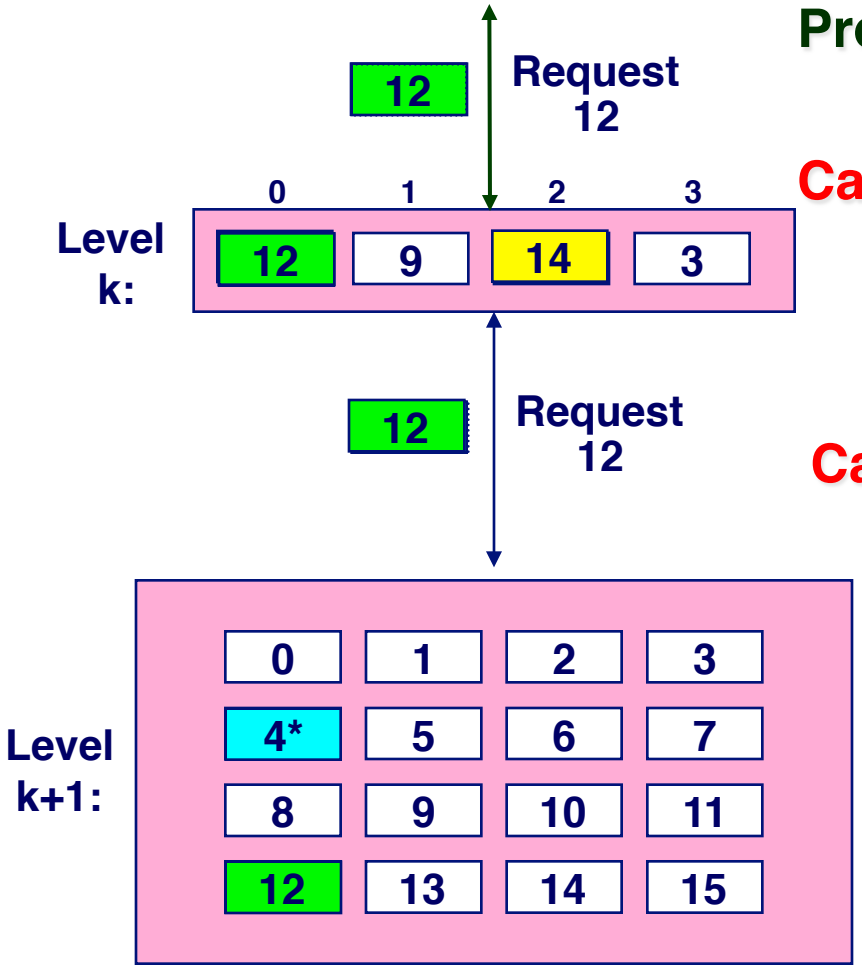
***Block b is stored in cache***

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

# Caching in a Memory Hierarchy



# General Caching Concepts




**Program needs object d, which is stored in some block b.**

**Cache hit**

- Program finds b in the cache at level k. E.g., block 14.

**Cache miss**

- b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
- If level k cache is full, then some current block must be replaced (evicted). Which one is the "victim"?
  - Placement policy: where can the new block go? E.g., b mod 4
  - Replacement policy: which block should be evicted? E.g., LRU

# Types of Cache Misses

## Cold (compulsory) miss

- Occurs on first access to a block
- Need to "warm up" cache in the beginning

## Conflict miss

- Most hardware caches limit blocks to a small subset (sometimes a singleton) of the available cache slots
  - e.g., block i must be placed in slot (i mod 4)
- Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
  - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time if placement policy is (i mod 4) and there is only one slot

## Capacity miss

- Occurs when the set of active cache blocks (working set) is larger than the cache

# Cache Performance Metrics

## Miss Rate

- Fraction of memory references not found in cache (misses / accesses)
  = 1 – hit rate
- Typical numbers (in percentages):
  - 3-10% for L1
  - can be quite small (e.g., < 1%) for L2, depending on size, etc.

## Hit Time

- Time to deliver a line in the cache to the processor
  - includes time to determine whether the line is in the cache
- Typical numbers:
  - 1-2 clock cycle for L1
  - 5-20 clock cycles for L2

## Miss Penalty

- Additional time required because of a miss
  - typically 50-200 cycles for main memory (Trend: increasing!)

# Average Cache Performance

## Huge difference between a hit and a miss

- Could be 100x, if just L1 and main memory

## Would you believe 99% hits is twice as good as 97%?

- Consider:
  cache hit time of 1 cycle
  miss penalty of 100 cycles

- Average access time:

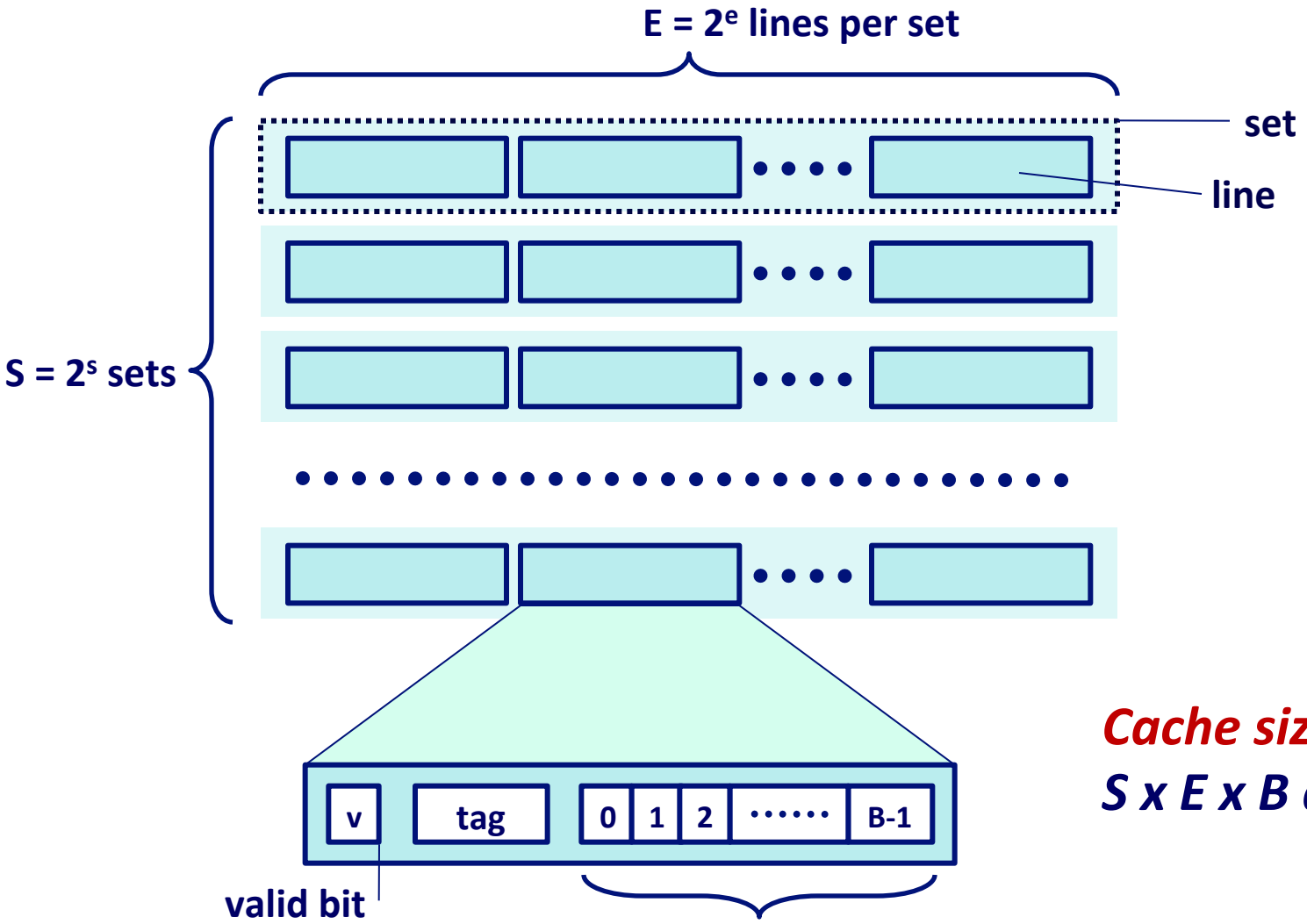
  97% hits: 0.97 * 1 cycle + 0.03 * 100 cycles ≈ 4 cycles

  99% hits: 0.99 * 1 cycle + 0.01 * 100 cycles ≈ 2 cycles

## This is why "miss rate" is used instead of "hit rate"

# General Cache Organization (S, E, B)

**For L1 hardware caches, access must be fast, so organize as follows:**




***Cache size:***

***S x E x B data bytes***

# Backup Slides

# Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte and have less capacity.
  - 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.

# Examples of Caching in the Hierarchy

| Cache Type | What is Cached? | Where is it Cached? | Latency (cycles) | Managed By |
|---|---|---|---|---|
| Registers | 4-byte words | CPU core | 0 | Compiler |
| TLB | Address translations | On-Chip TLB | 0 | Hardware |
| L1 cache | 64-bytes block | On-Chip L1 | 1 | Hardware |
| L2 cache | 64-bytes block | Off-Chip L2 | 10 | Hardware |
| Virtual Memory | 4-KB page | Main memory | 100 | Hardware+OS |
| Buffer cache | Parts of files | Main memory | 100 | OS |
| Network buffer cache | Parts of files | Local disk | 10,000,000 | AFS/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 |

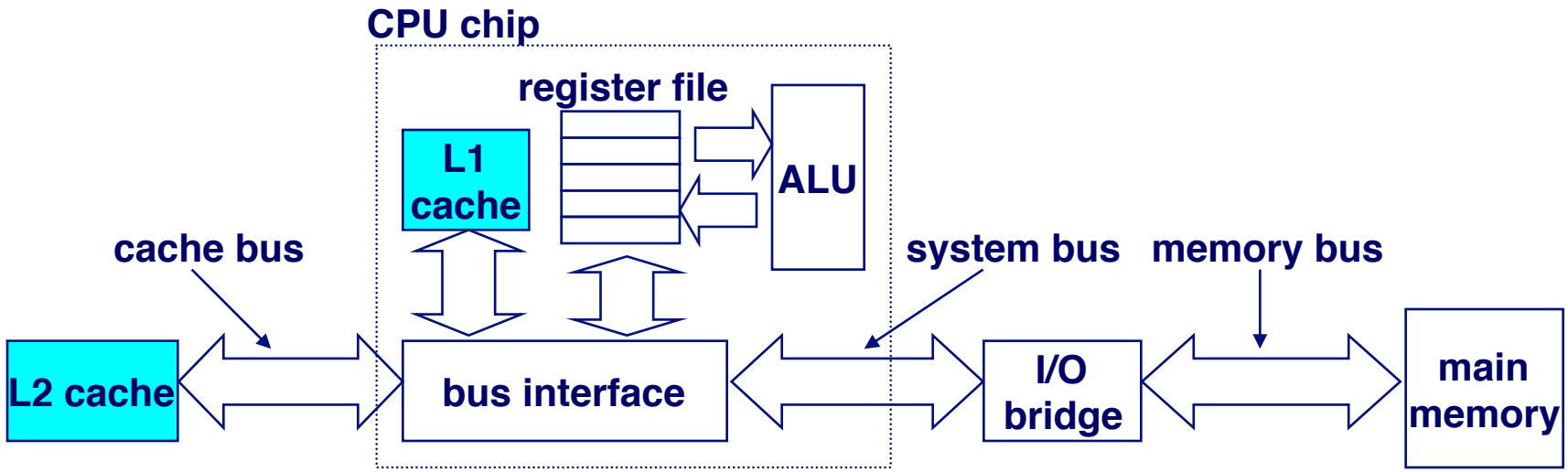
# Cache Memories

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 L1, then in L2, then in main memory.

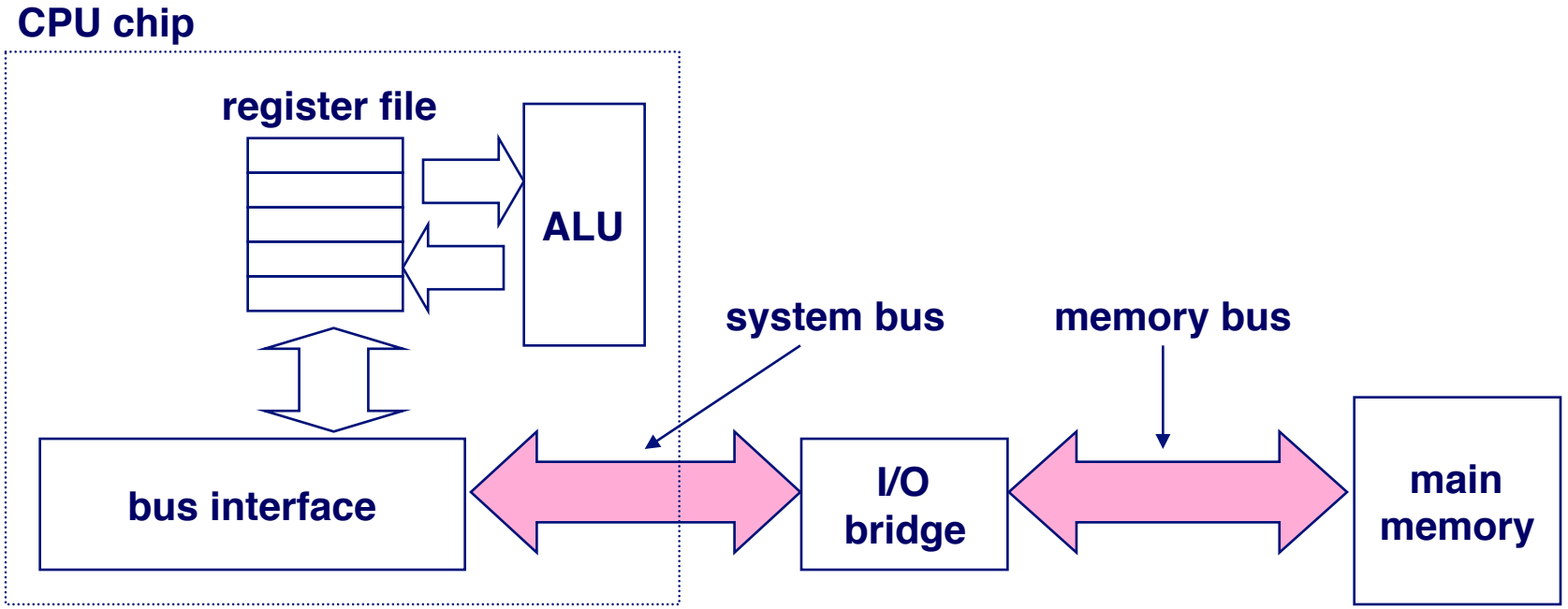
Typical bus structure:



# Typical Bus Structure Connecting CPU and Memory

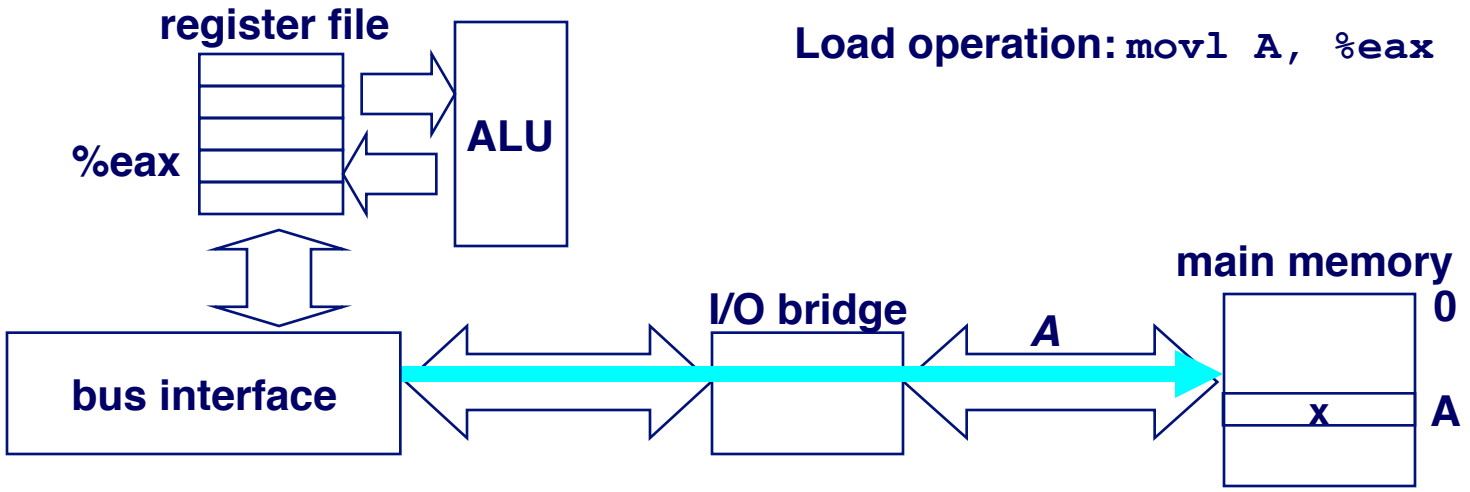
A bus is a collection of parallel wires that carry address, data, and control signals.

Buses are typically shared by multiple devices.



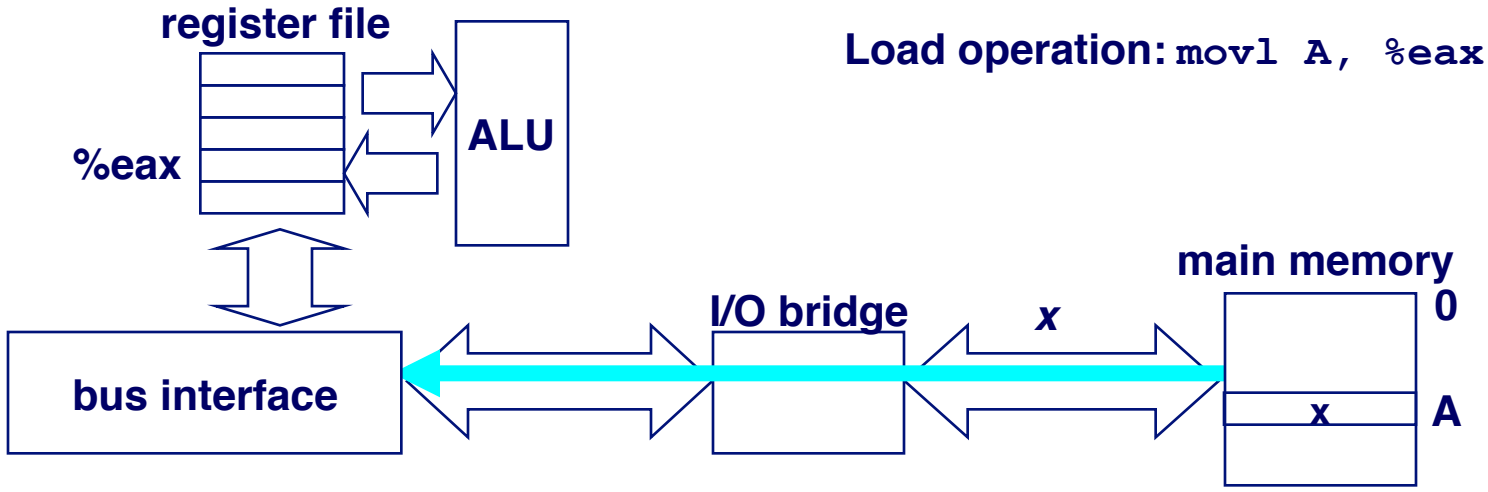
# Memory Read Transaction (1)

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



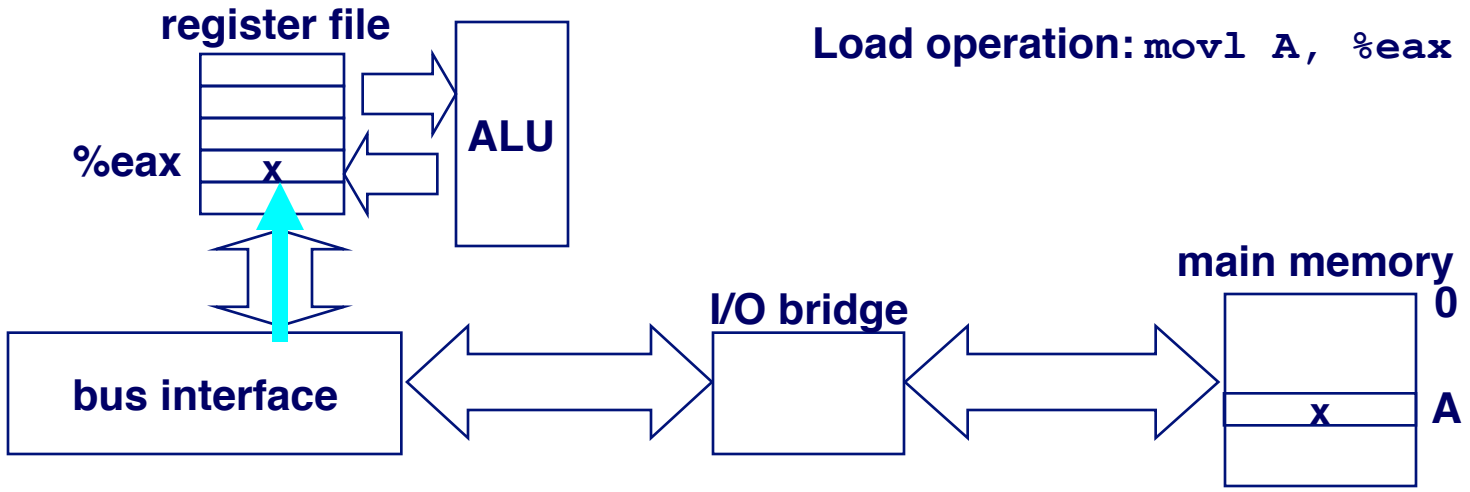
# Memory Read Transaction (2)

**Main memory reads A from the memory bus, retreives word x, and places it on the bus.**



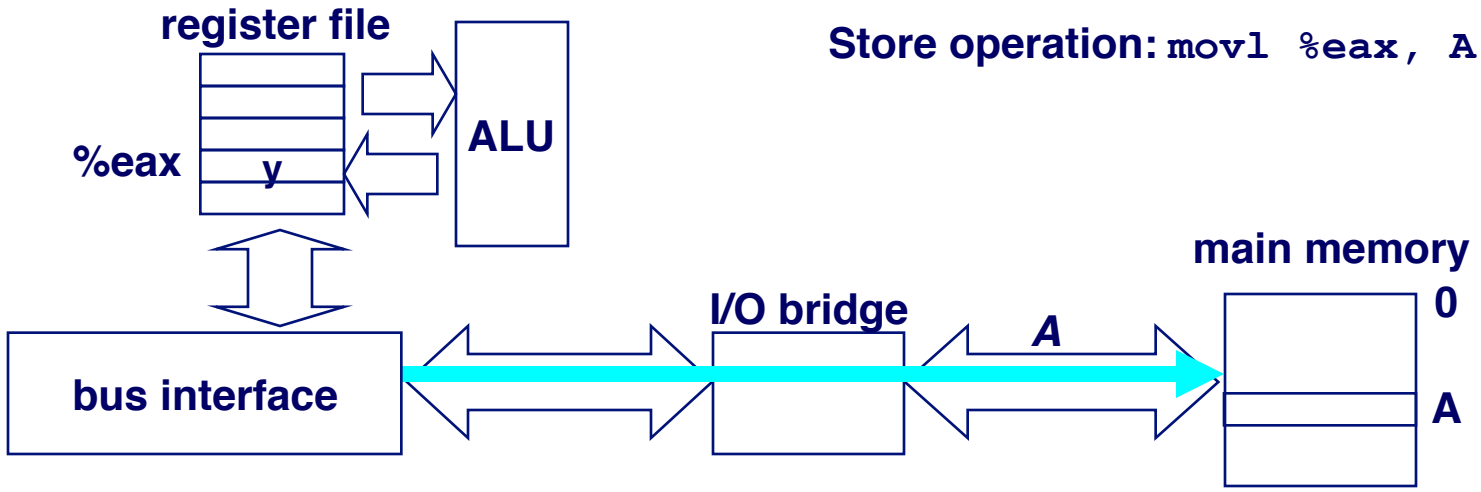
# Memory Read Transaction (3)

CPU read word x from the bus and copies it into register %eax.



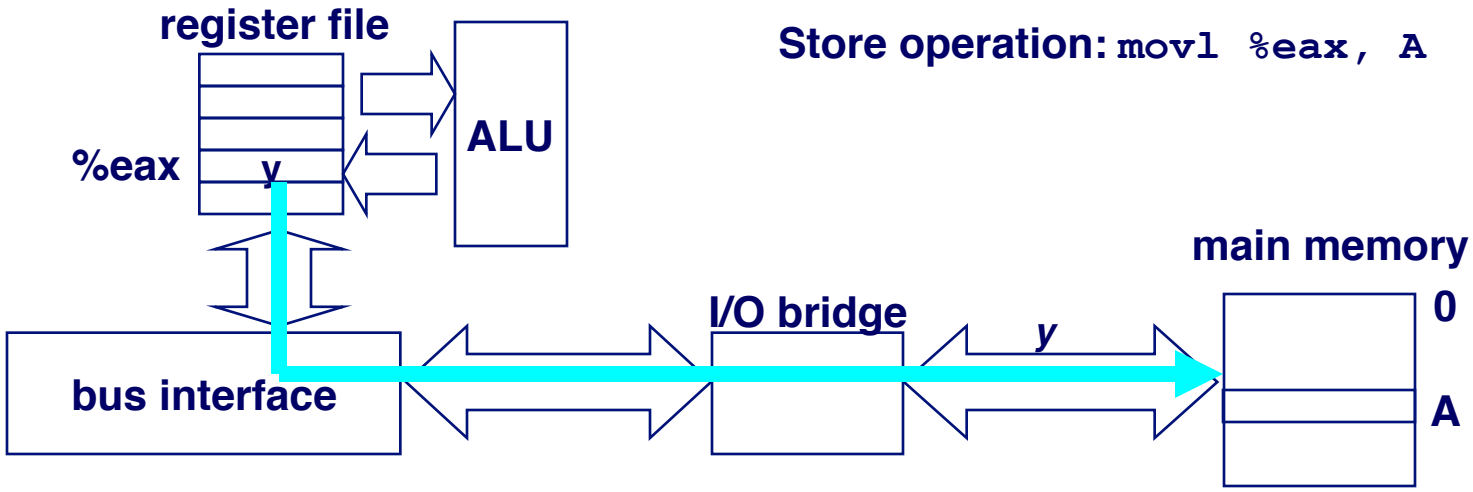
# Memory Write Transaction (1)

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



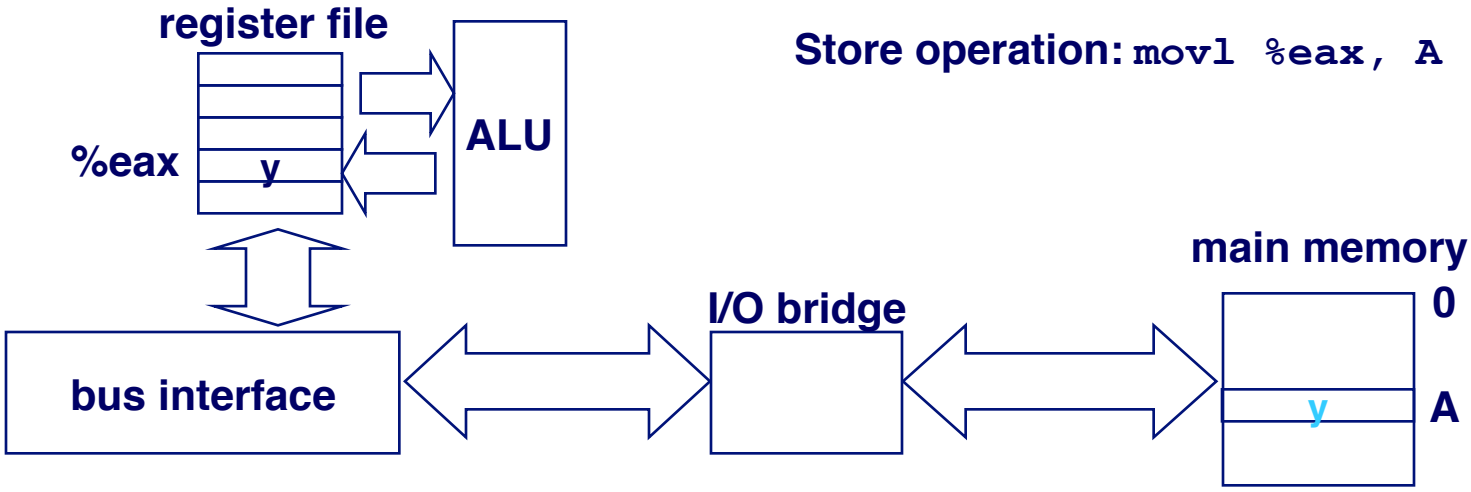
# Memory Write Transaction (2)

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



# Memory Write Transaction (3)

**Main memory read data word y from the bus and stores it at address A.**



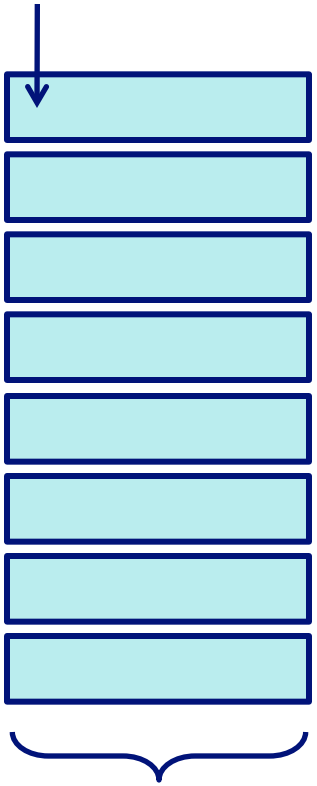
# Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

*Ignore the variables sum, i, j*

**assume: cold (empty) cache, a[0][0] goes here**



**32 B = 4 doubles**

**blackboard**

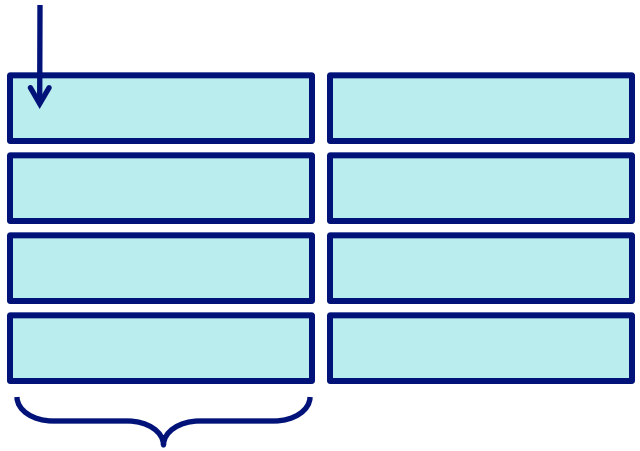
# Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

*Ignore the variables sum, i, j*

**assume: cold (empty) cache, a[0][0] goes here**

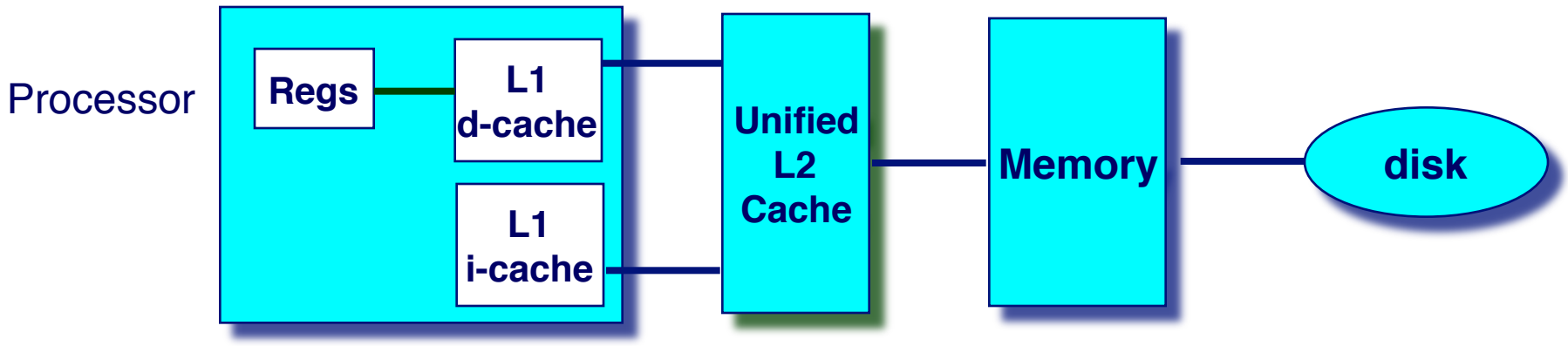


**32 B = 4 doubles**

**blackboard**

# Multi-Level Caches

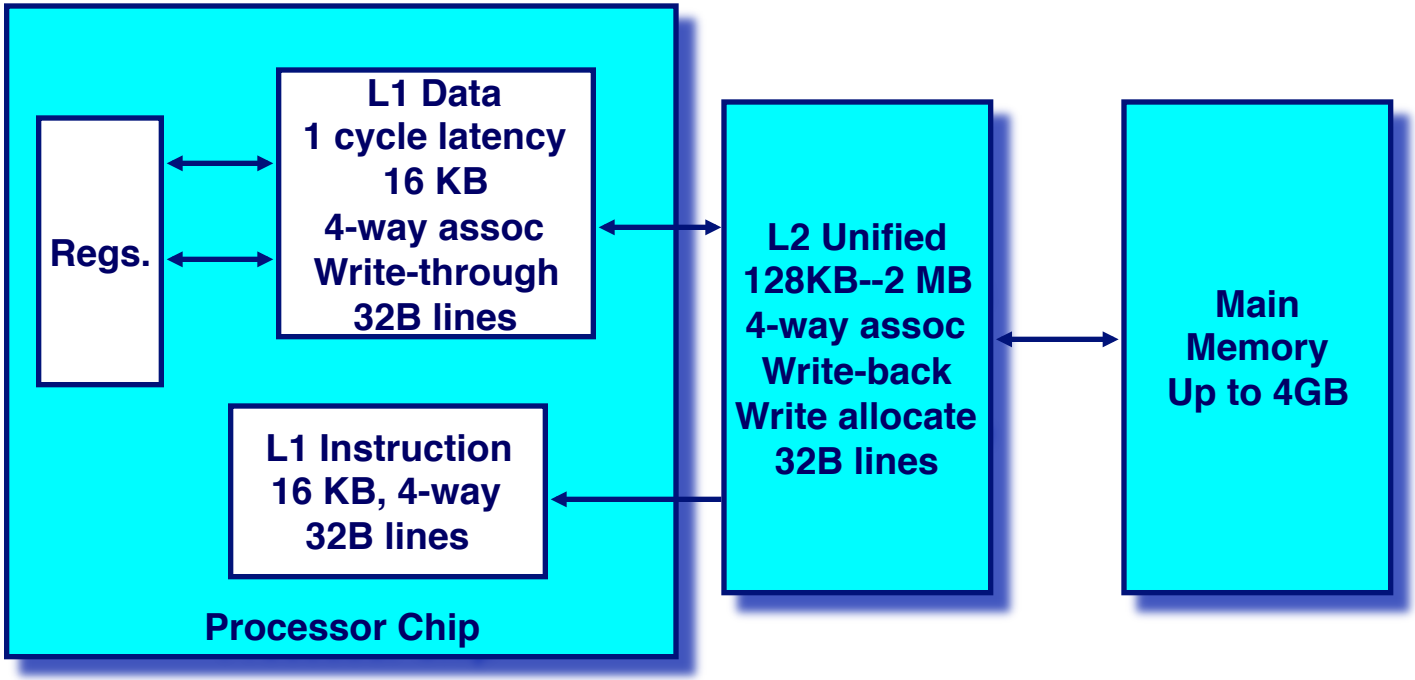
**Options: separate data and instruction caches, or a unified cache**




| | | | | | |
|---|---|---|---|---|---|
| size: | 200 B | 8-64 KB | 1-4MB SRAM | 128 MB DRAM | 30 GB |
| speed: | 3 ns | 3 ns | 6 ns | 60 ns | 8 ms |
| $/Mbyte: | | | $100/MB | $1.50/MB | $0.05/MB |
| line size: | 8 B | 32 B | 32 B | 8 KB | |

larger, slower, cheaper

# Intel Pentium Cache Hierarchy



# Memory Hierarchy: Core 2 Duo

*Not drawn to scale*

**L1/L2 cache: 64 B blocks**