# Lecture 09
# Caches – part 1

Euhyun Moon, Ph.D.
Machine Learning Systems (MLSys) Lab
Computer Science and Engineering
Sogang University

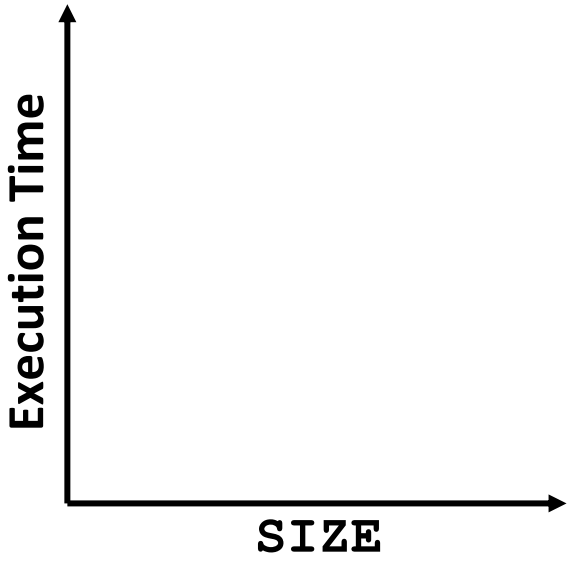

Slides adapted from Randy Bryant and Dave O'Hallaron: Introduction to Computer Systems, CMU

# How does execution time grow with SIZE?

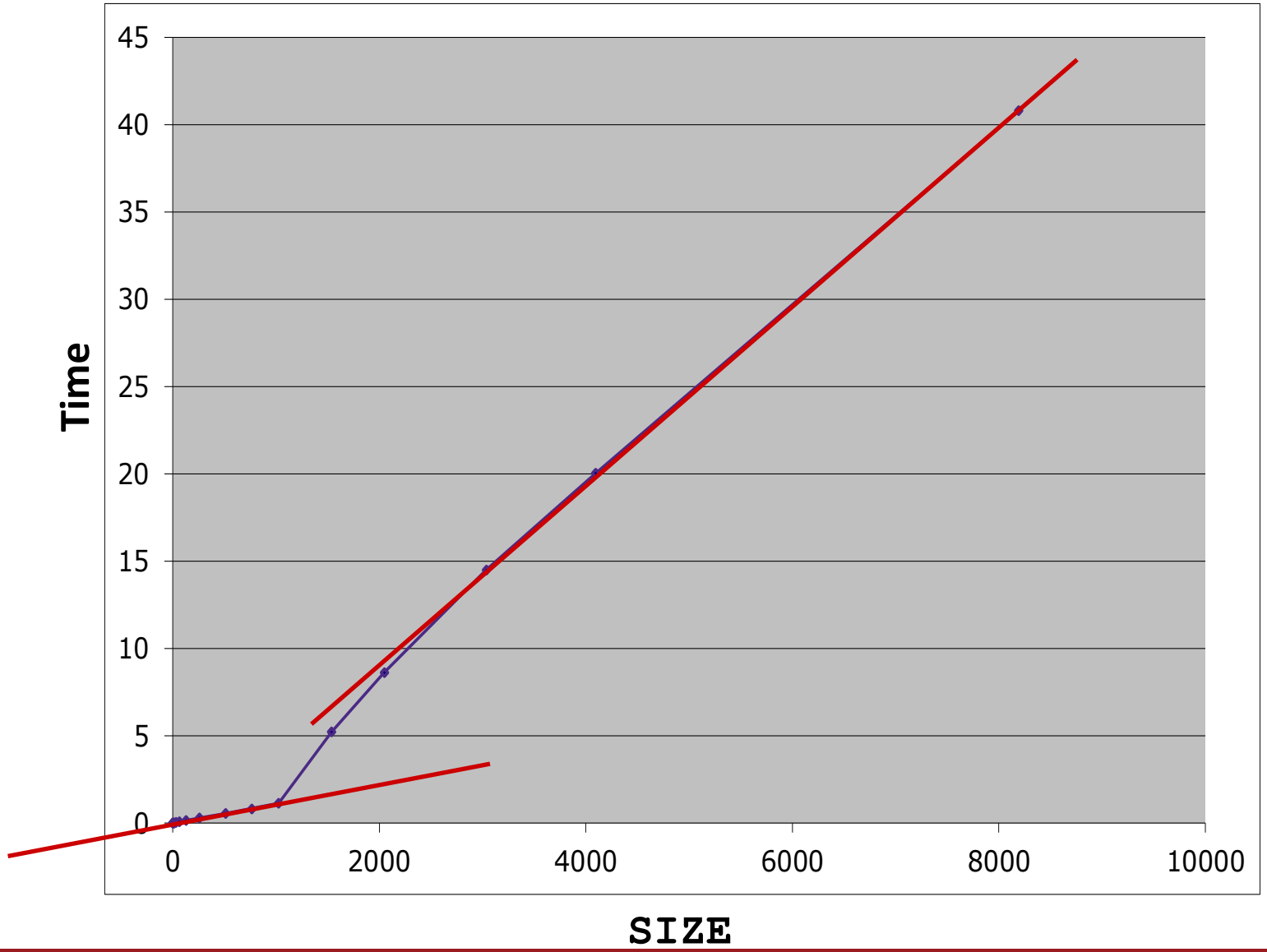
```
int array[SIZE];
int sum = 0;

for (int i = 0; i < 200000; i++) {
  for (int j = 0; j < SIZE; j++) {
    sum += array[j];
  }
}
```

**Plot:**



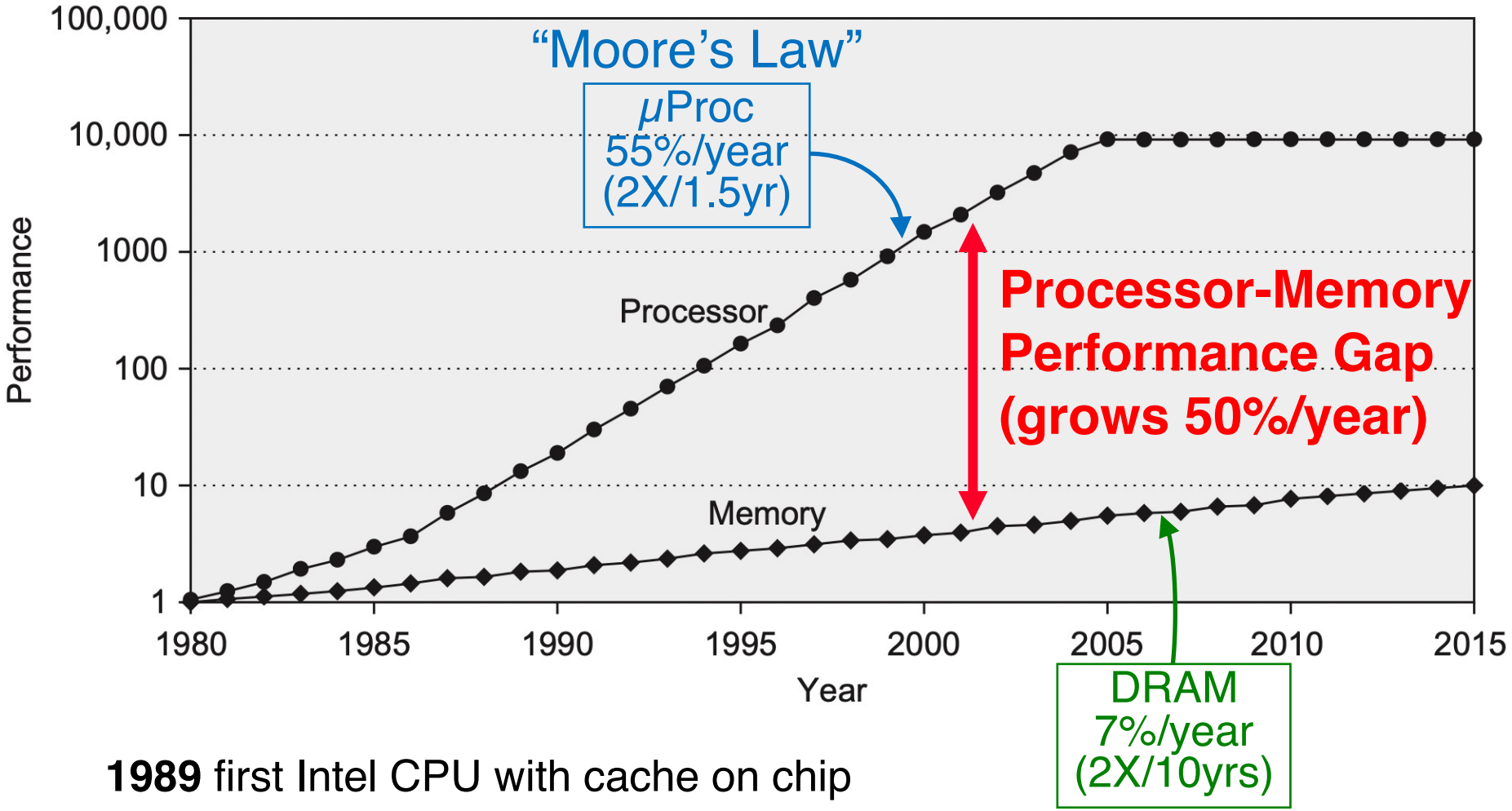
# Actual Data



# Making Memory Accesses Fast!

- **Cache basics**
- **Principle of locality**
- **Memory hierarchies**
- **Cache structure**
- Cache mappings
  - Direct-mapped cache
  - Set associative cache
  - Fully associative cache
- Cache performance metrics
- Cache-friendly code

# Processor-Memory Gap



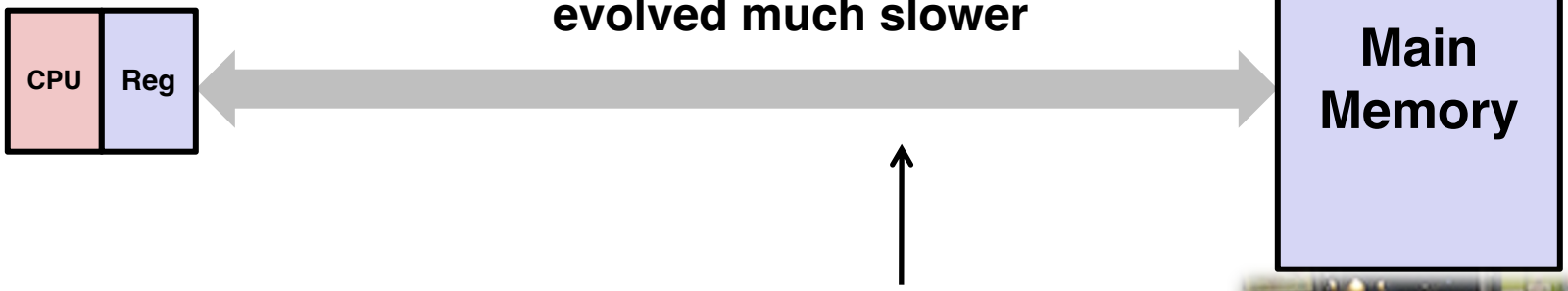

**1989** first Intel CPU with cache on chip

**1998** Pentium III has two cache levels on chip

# Problem: Processor-Memory Bottleneck

**Processor performance doubled about every 18 months**

**Bus latency / bandwidth evolved much slower**




***Core 2 Duo:***
**Can process at least**
256 Bytes/cycle

***Core 2 Duo:***
**Bandwidth**
2 Bytes/cycle
**Latency**
100-200 cycles (30-60ns)

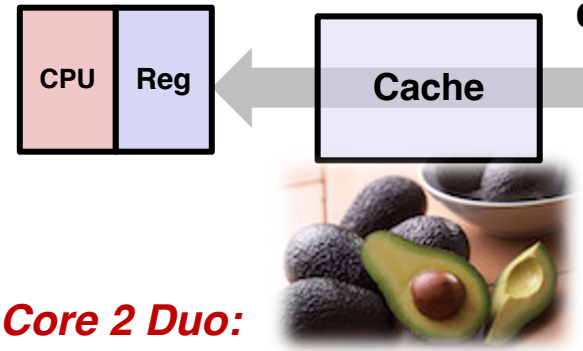

***Problem: lots of waiting on memory***

***cycle**: single machine step (fixed-time)*

# Problem: Processor-Memory Bottleneck

**Processor performance doubled about every 18 months**

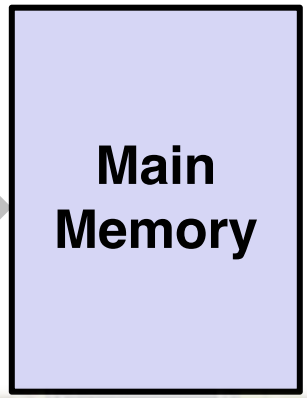
**Bus latency / bandwidth evolved much slower**






***Core 2 Duo:***
**Can process at least**
256 Bytes/cycle

***Core 2 Duo:***
**Bandwidth**
2 Bytes/cycle
**Latency**
100-200 cycles (30-60ns)





***Solution: caches***

***cycle**: single machine step (fixed-time)*

# Cache 💰

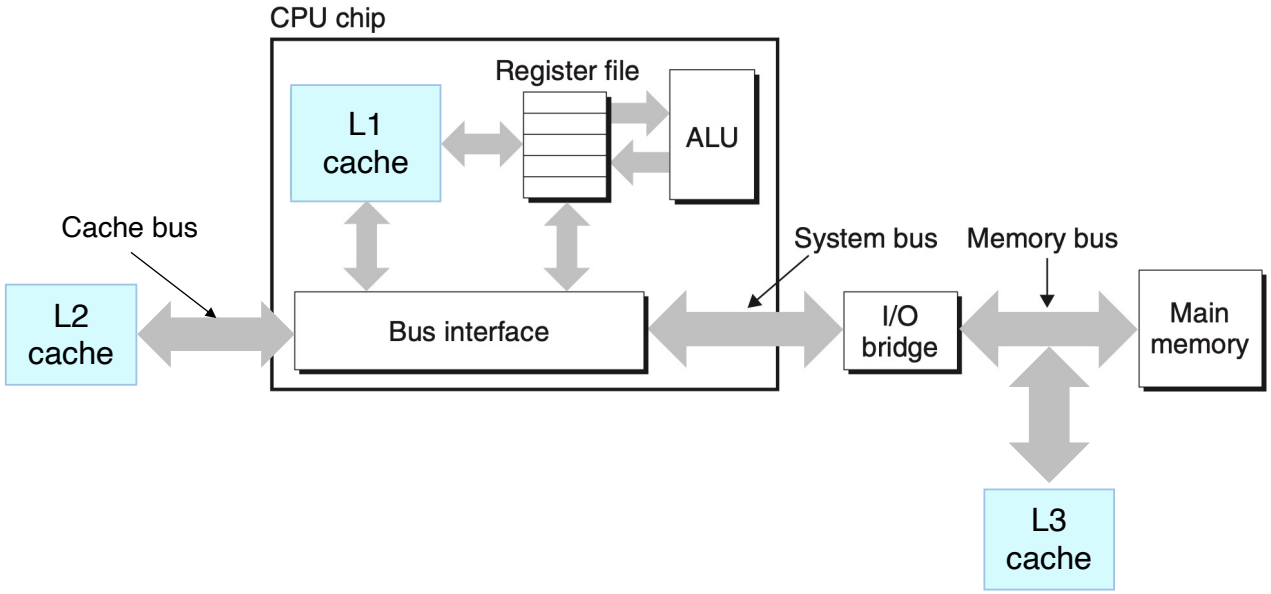
- Pronunciation: “cash”
  - We abbreviate this as “$”
- English: A hidden storage space for provisions, weapons, and/or treasures
- Computer: Memory with short access time used for the storage of frequently or recently used instructions (i-cache/I$) or data (d-cache/D$)
  - *More generally:* Used to optimize data transfers between any system elements with different characteristics (network interface cache, I/O cache, etc.)

# What is Cache Memory?

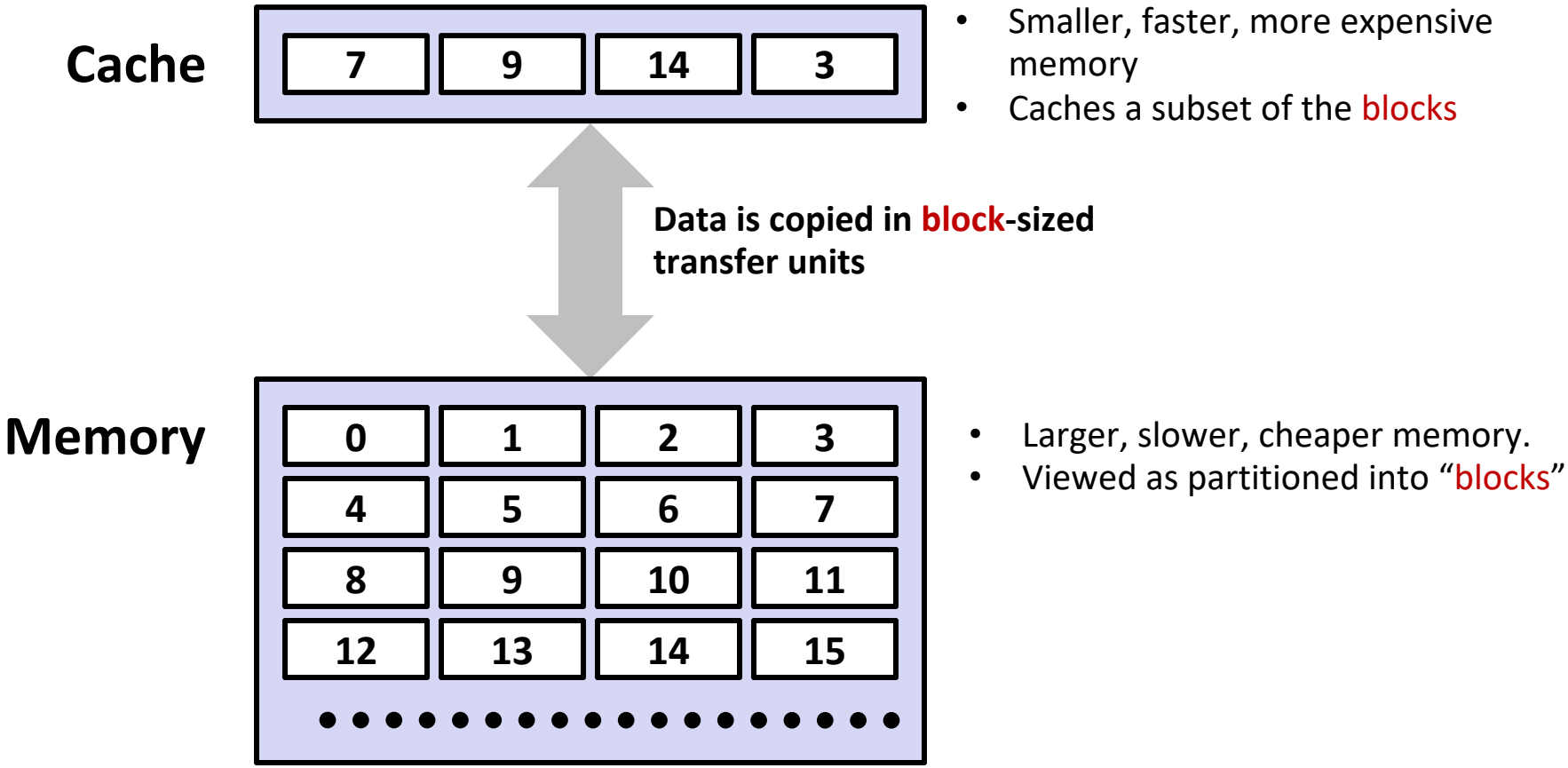
- something precious preserved or concealed in a convenient but private place
- a computer memory with very short access time used for storage of frequently or recently used instructions or data
- a small, fast subset of a larger collection, intended to provide faster average access to the larger whole

# Cache Memories

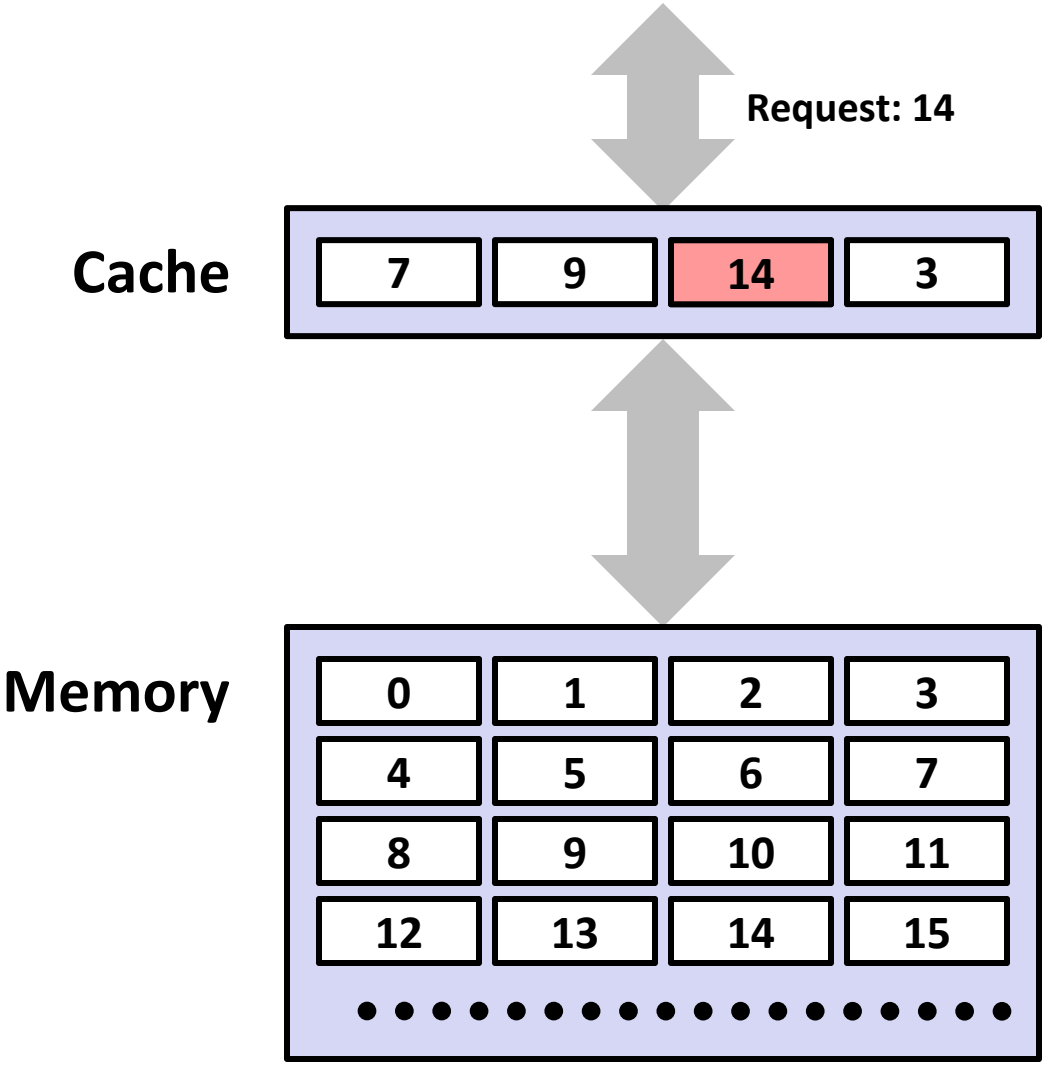
- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in L1, then in L2, then in main memory
- Typical bus structure:



# General Cache Mechanics



# General Cache Concepts: Hit

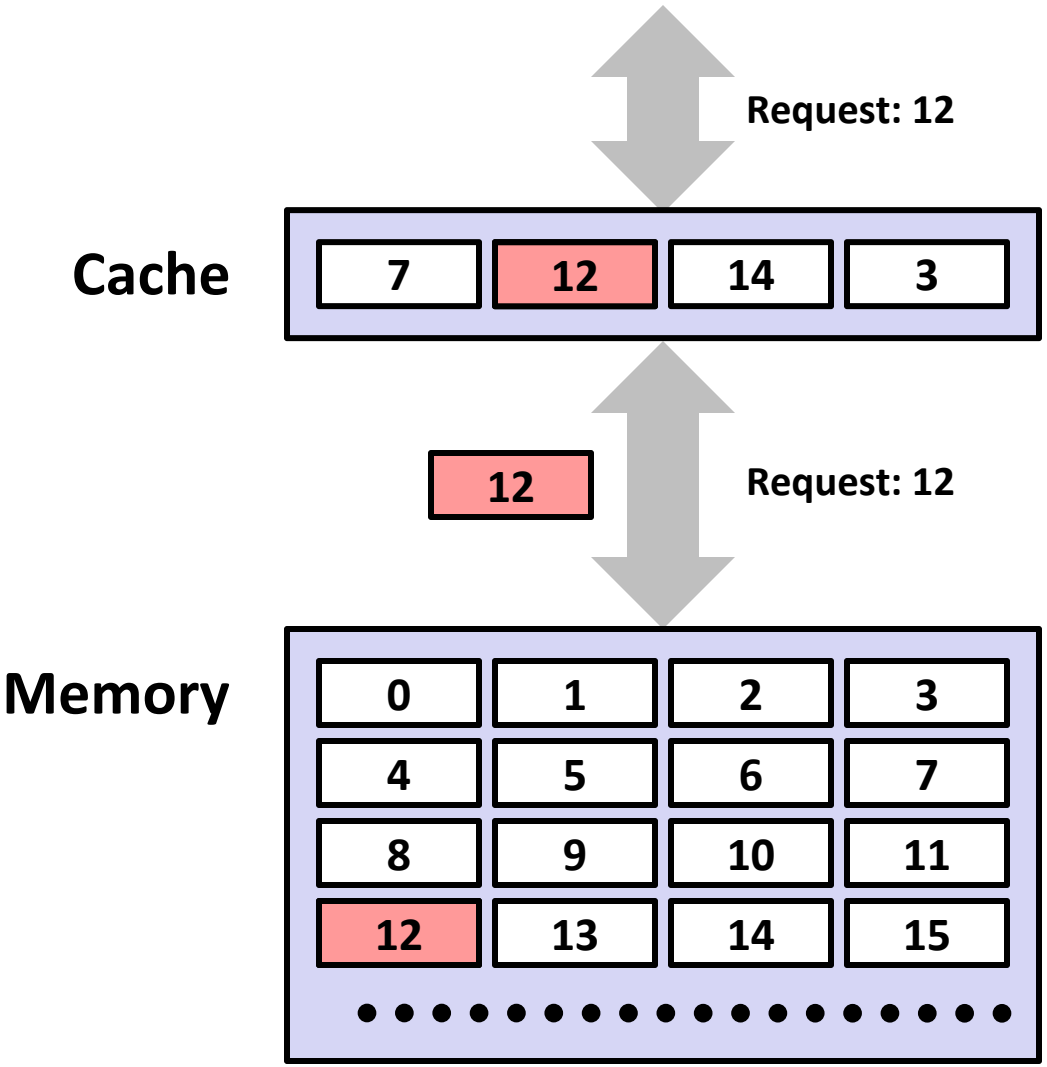



*Data in block b is needed*

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

*Data is returned to CPU*

# 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)

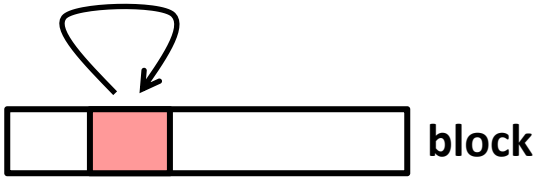
***Data is returned to CPU***

# Why Caches Work

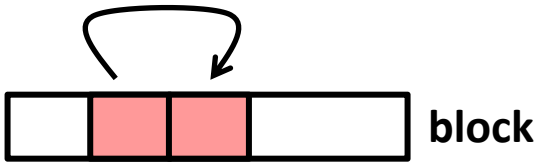
- Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

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

- *Temporal* locality:
  - Can we anticipate using a memory address based on when it was last used?
  - Recently referenced items are *likely* to be referenced again in the near future




- *Spatial* locality:
  - Can we anticipate using a memory address based on where it is?
  - Items with nearby addresses *tend* to be referenced close together in time




- How do caches take advantage of this?

# Example: Any Locality?

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

- **Data:**
  - Temporal: `sum` referenced in each iteration
  - Spatial: consecutive elements of array `a[]` accessed
- **Instructions:**
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

# Locality Example #1

```
int sum_array_rows(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 #1

```
int sum_array_rows(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;
}
```

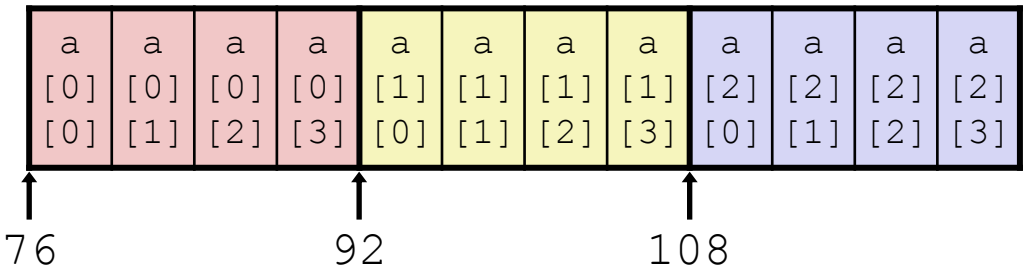
**M = 3, N=4**

| | | | |
|---|---|---|---|
| a[0][0] | a[0][1] | a[0][2] | a[0][3] |
| a[1][0] | a[1][1] | a[1][2] | a[1][3] |
| a[2][0] | a[2][1] | a[2][2] | a[2][3] |

**Access Pattern:**
stride = ?

1) a[0][0]
2) a[0][1]
3) a[0][2]
4) a[0][3]
5) a[1][0]
6) a[1][1]
7) a[1][2]
8) a[1][3]
9) a[2][0]
10) a[2][1]
11) a[2][2]
12) a[2][3]

**Layout in Memory**




**Note:** 76 is just one possible starting address of array a

# Locality Example #2

```
int sum_array_cols(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;
}
```

# Locality Example #2

```
int sum_array_cols(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;
}
```

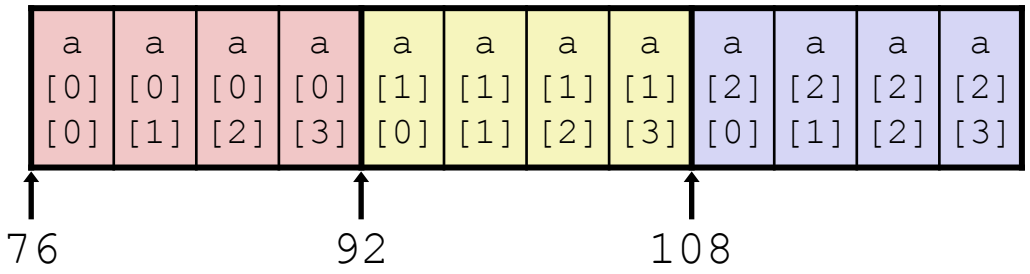
**M = 3, N=4**

| | | | |
|---|---|---|---|
| a[0][0] | a[0][1] | a[0][2] | a[0][3] |
| a[1][0] | a[1][1] | a[1][2] | a[1][3] |
| a[2][0] | a[2][1] | a[2][2] | a[2][3] |

**Access Pattern:**
stride = ?

1) a[0][0]
2) a[1][0]
3) a[2][0]
4) a[0][1]
5) a[1][1]
6) a[2][1]
7) a[0][2]
8) a[1][2]
9) a[2][2]
10) a[0][3]
11) a[1][3]
12) a[2][3]

**Layout in Memory**

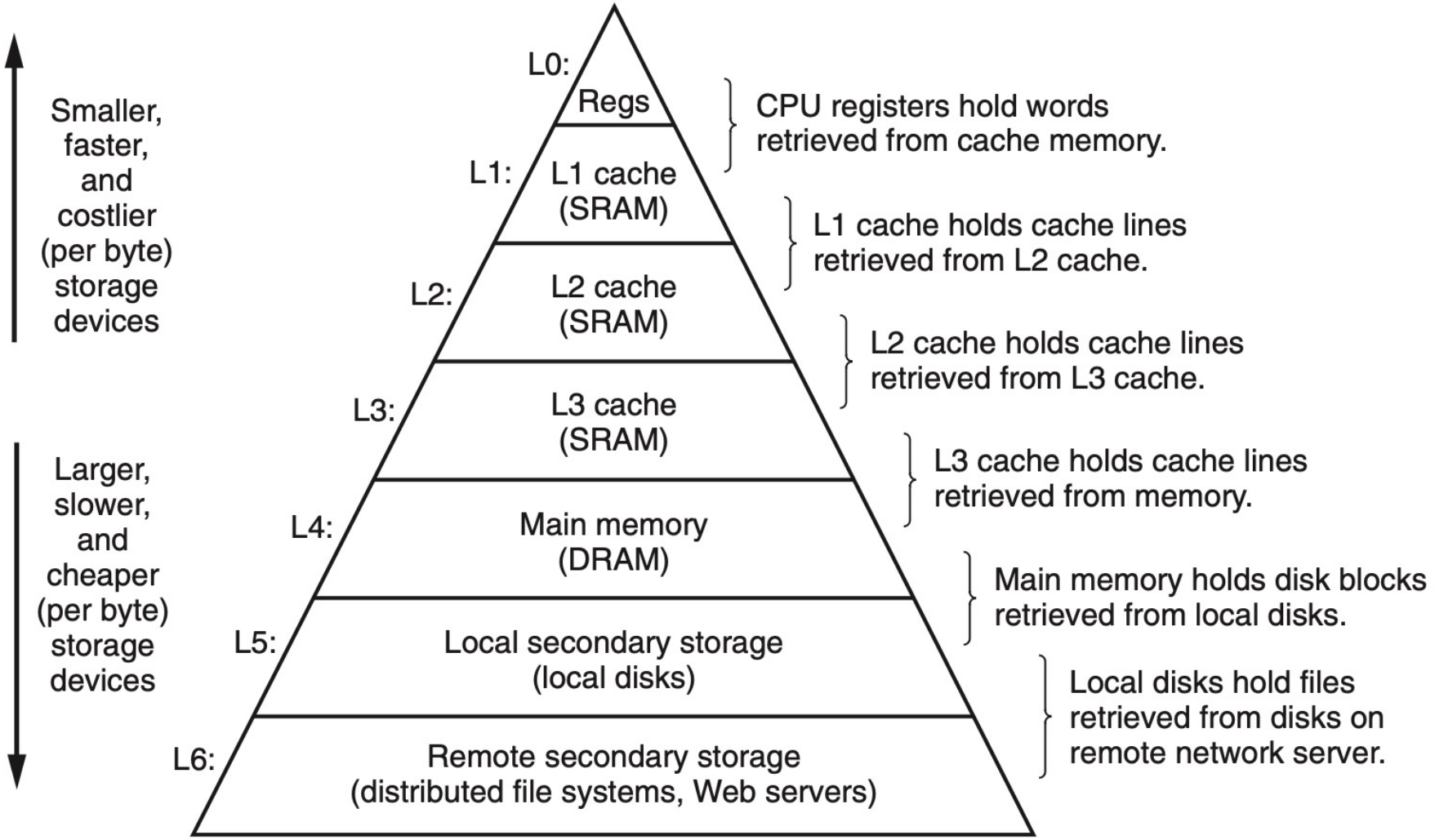


# Memory Hierarchies

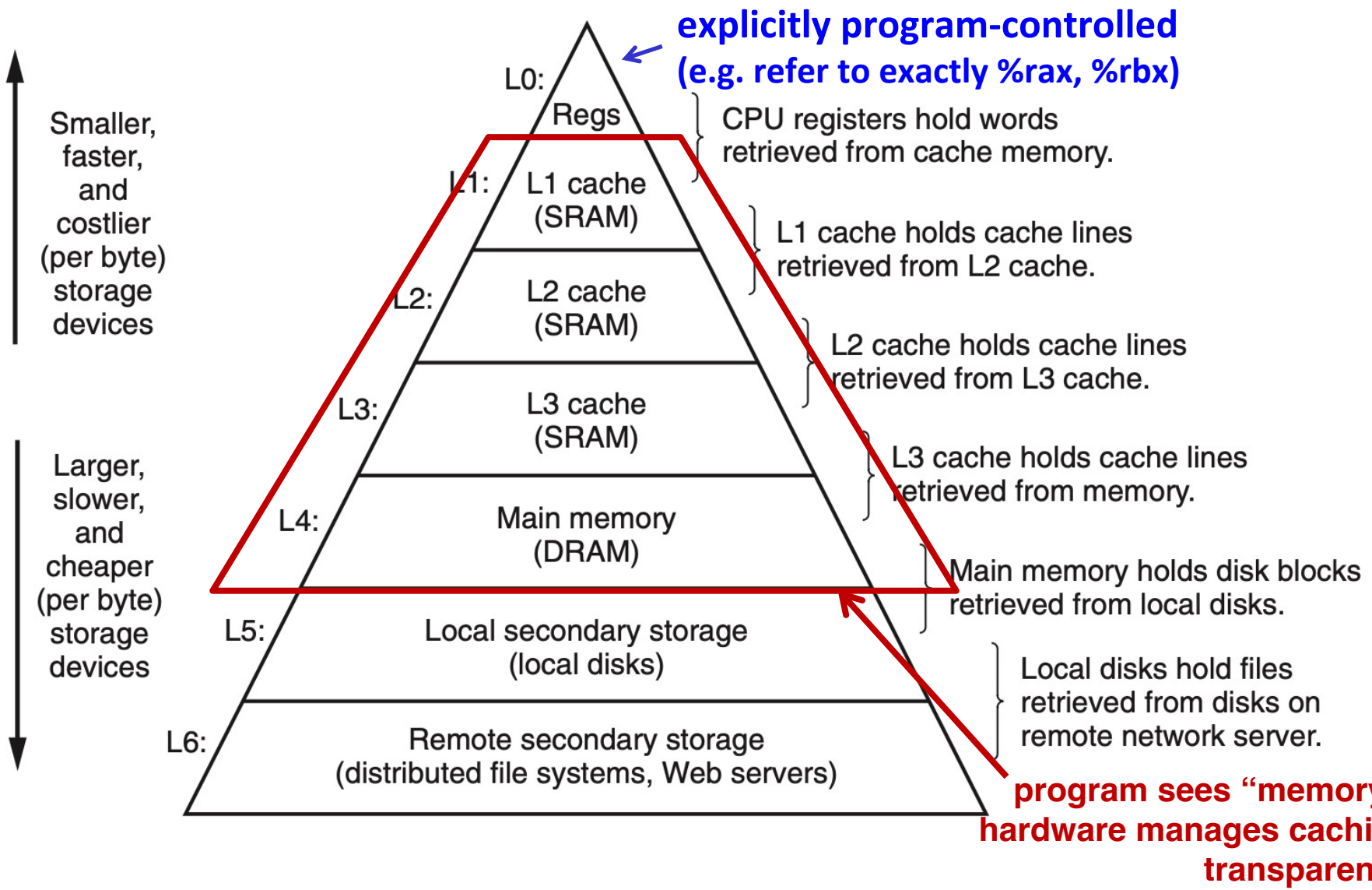
- Some fundamental and enduring properties of hardware and software systems:
  - Faster storage technologies almost always cost more per byte and have lower capacity
  - The gaps between memory technology speeds are widening
    - True for: registers ↔ cache, cache ↔ DRAM, DRAM ↔ disk, etc.
  - Well-written programs tend to exhibit good locality

- These properties complement each other beautifully
  - They suggest an approach for organizing memory and storage systems known as a memory hierarchy
    - For each level k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1

# Example Memory Hierarchy



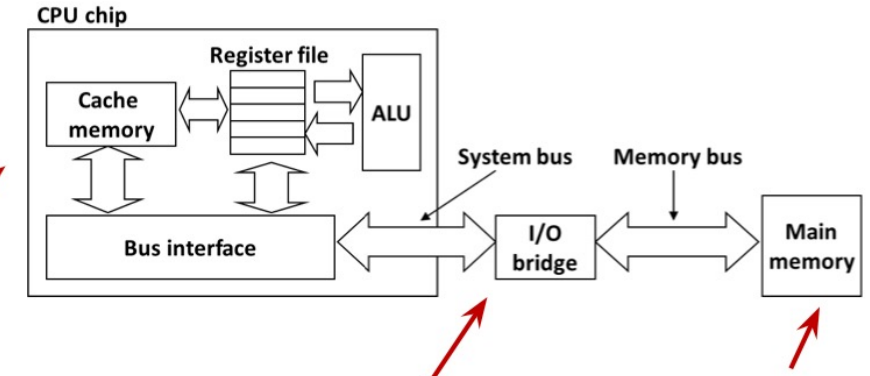
# Example Memory Hierarchy



# Inside a Computer

**Desktop PC**

*Source: Dell*




**CPU (Intel Core i7)**


*Source: PC Magazine*


*Source: techreport.com*

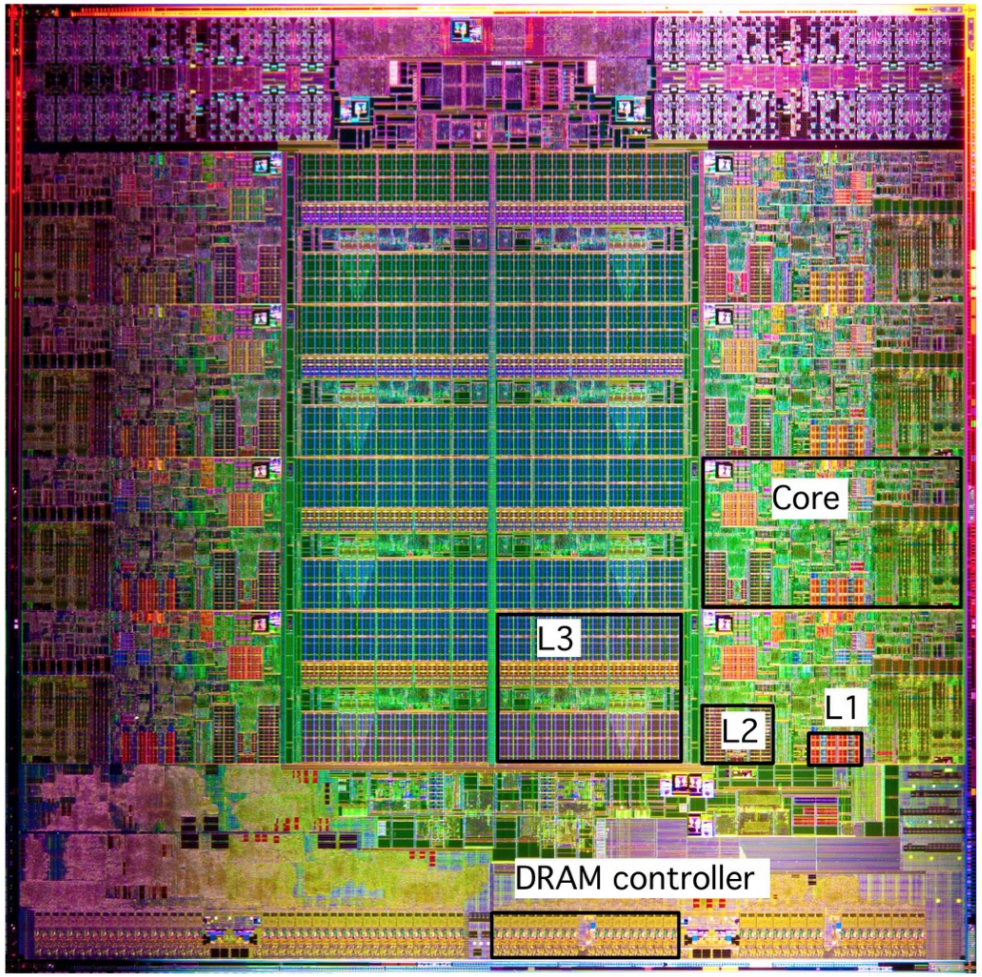
**Main memory (DRAM)**

*Source: Dell*

**Motherboard**

*Source: Dell*

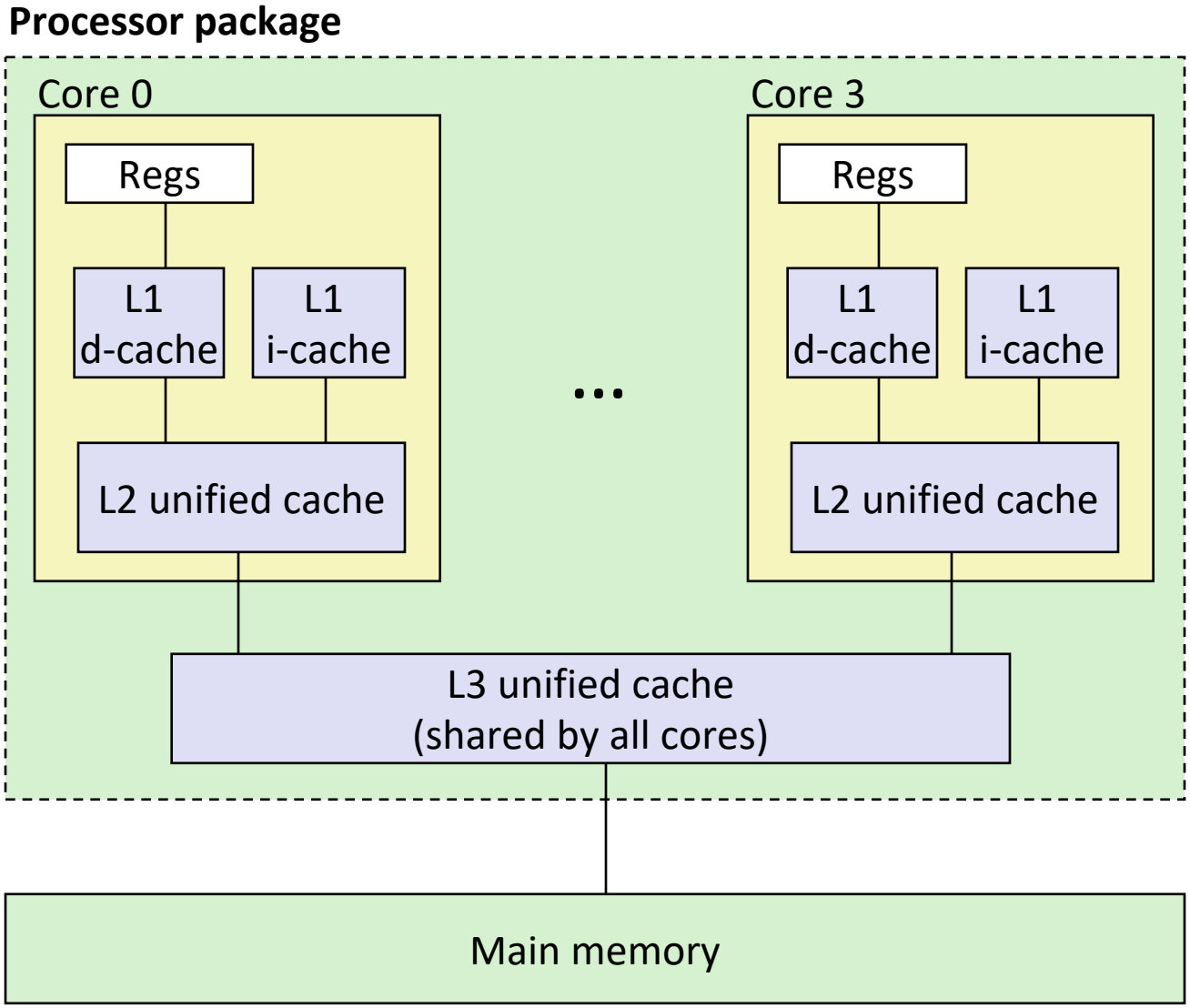
# Inside a Computer




**Intel Sandy Bridge Processor Die**

**L1: 32KB Instruction + 32KB Data**
**L2: 256KB**
**L3: 3–20MB**

# Intel Core i7 Cache Hierarchy



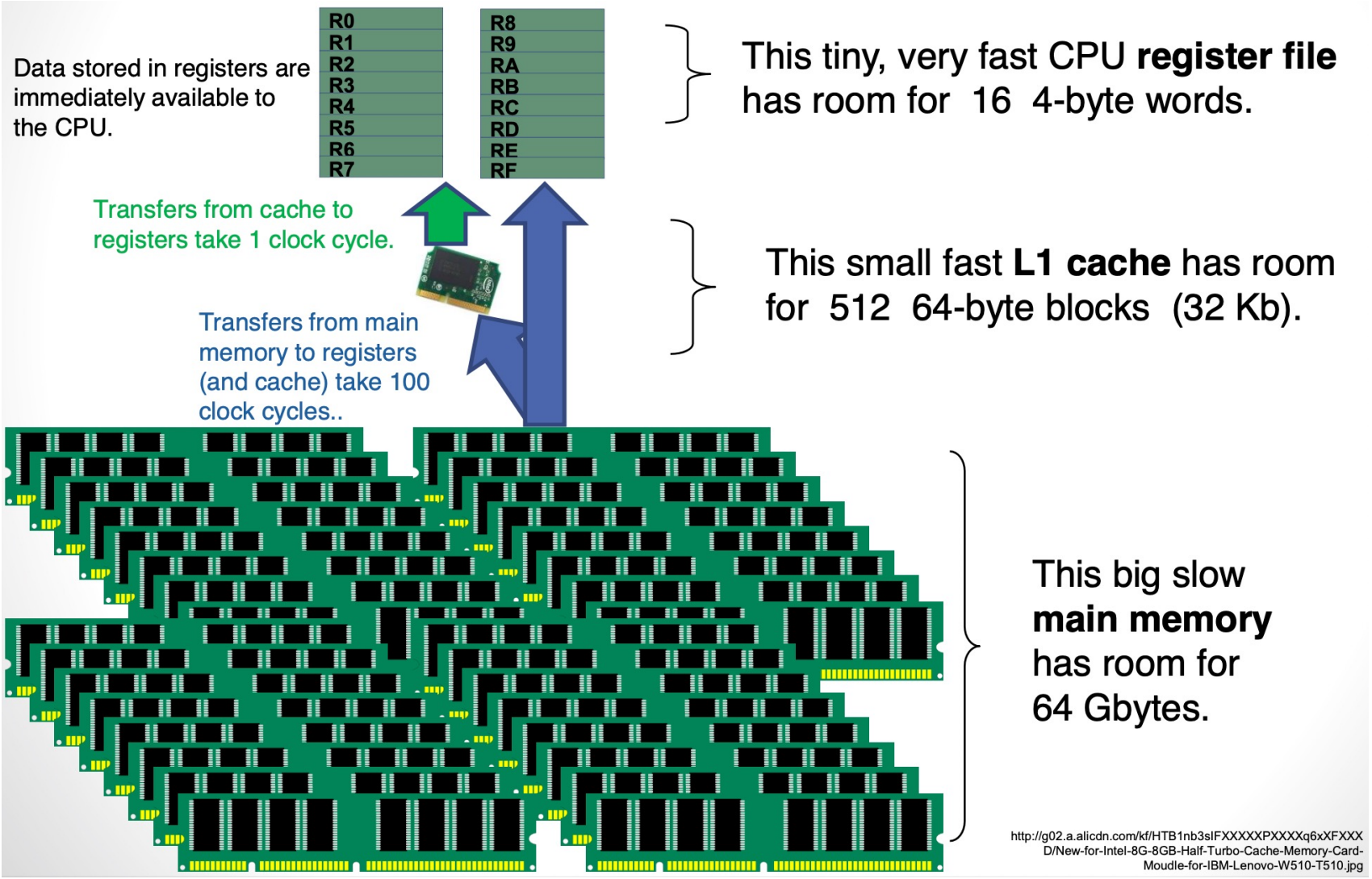

**Block size**:
64 bytes for all caches

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

**L2 unified cache:**
256 KB, 8-way,
Access: 11 cycles

**L3 unified cache:**
8 MB, 16-way,
Access: 30-40 cycles

# How Cache Works

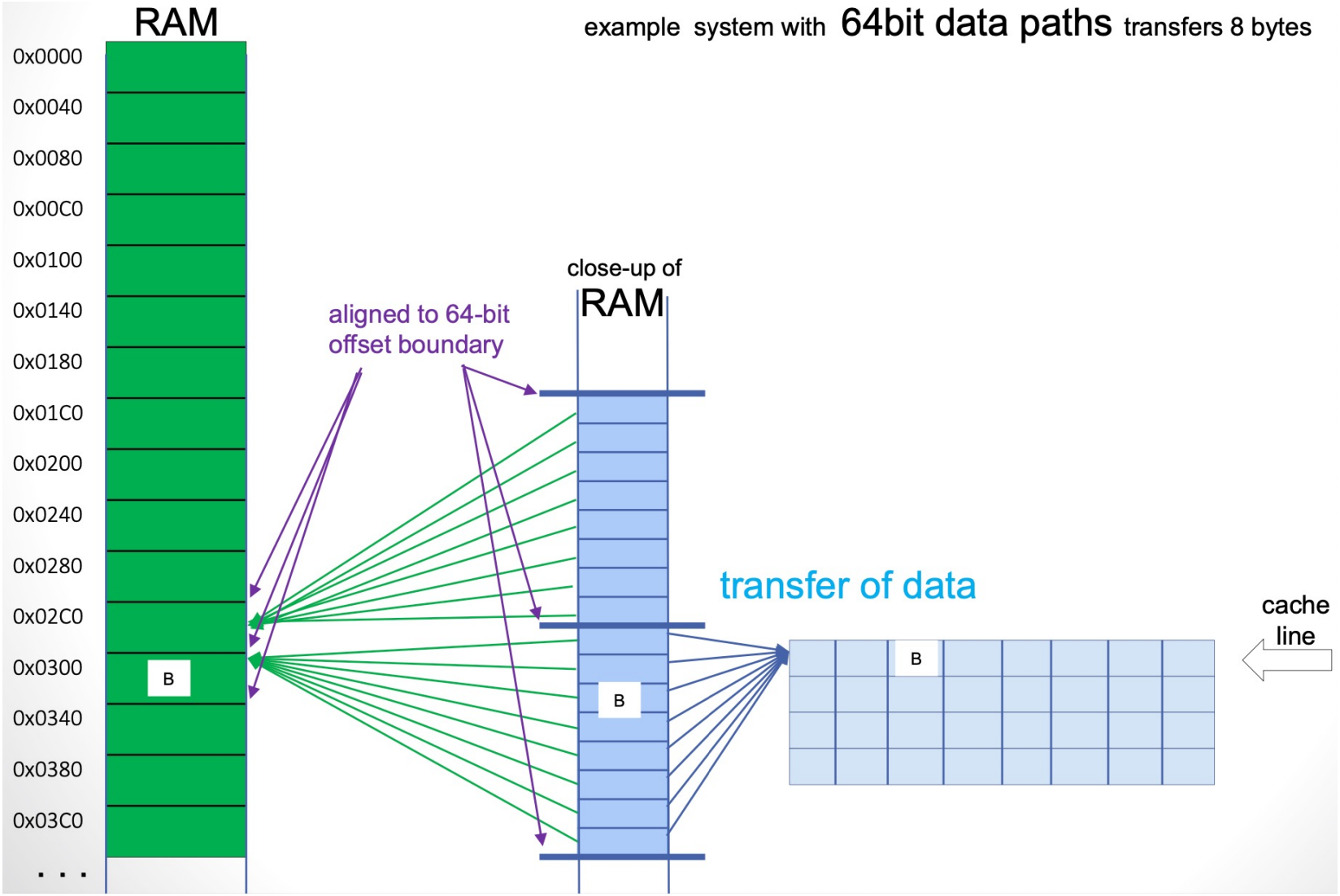


# Register Variables

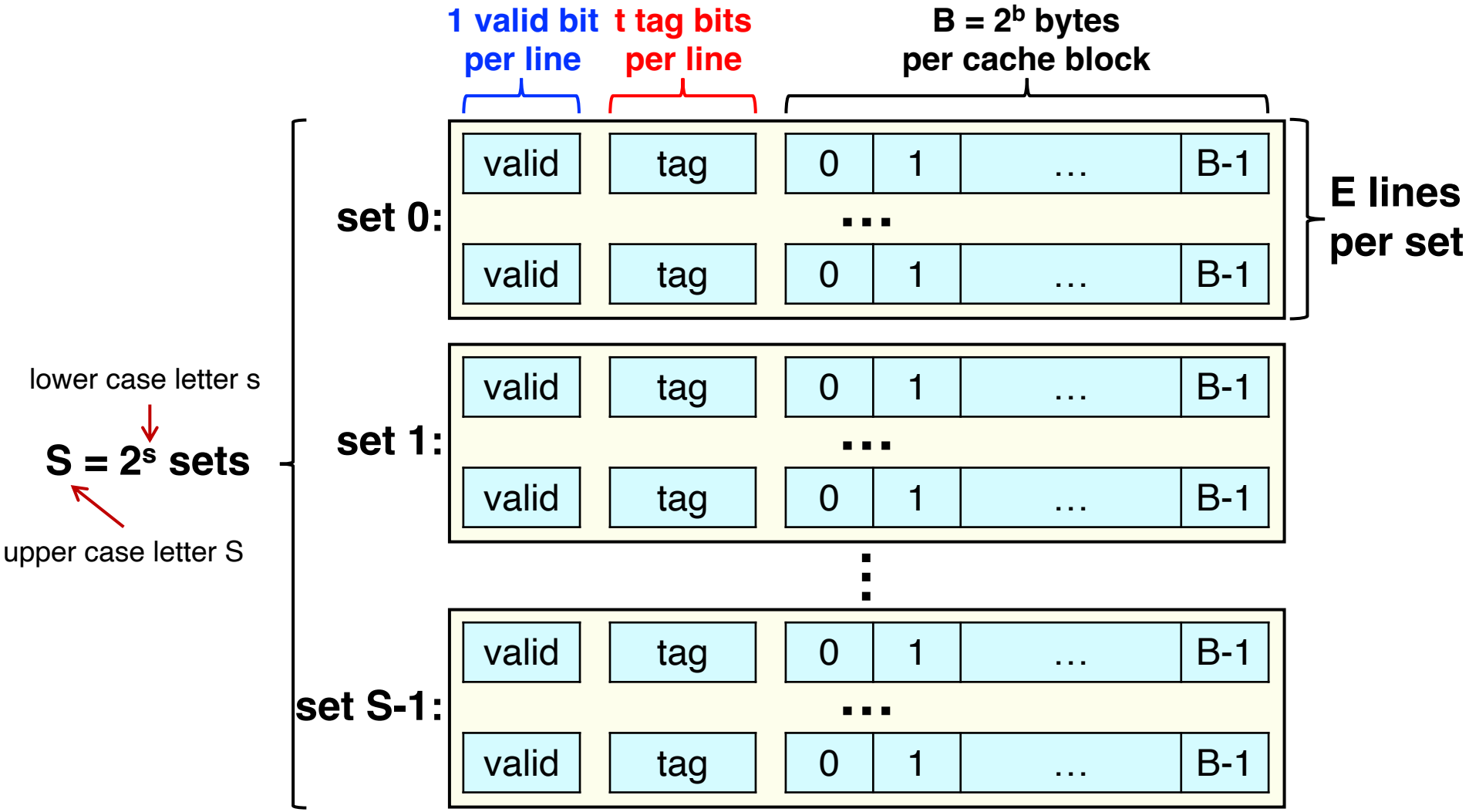
```
sum = 0;
for (i = 0; i < n; i++)
{
  sum += a[i];
}
return sum;
```

```
R1 = 0;
for (i = 0; i < n; i++)
{
  R1 += a[i];
}
sum = R1;
return sum;
```

# Cache Lines and Pre-fetching

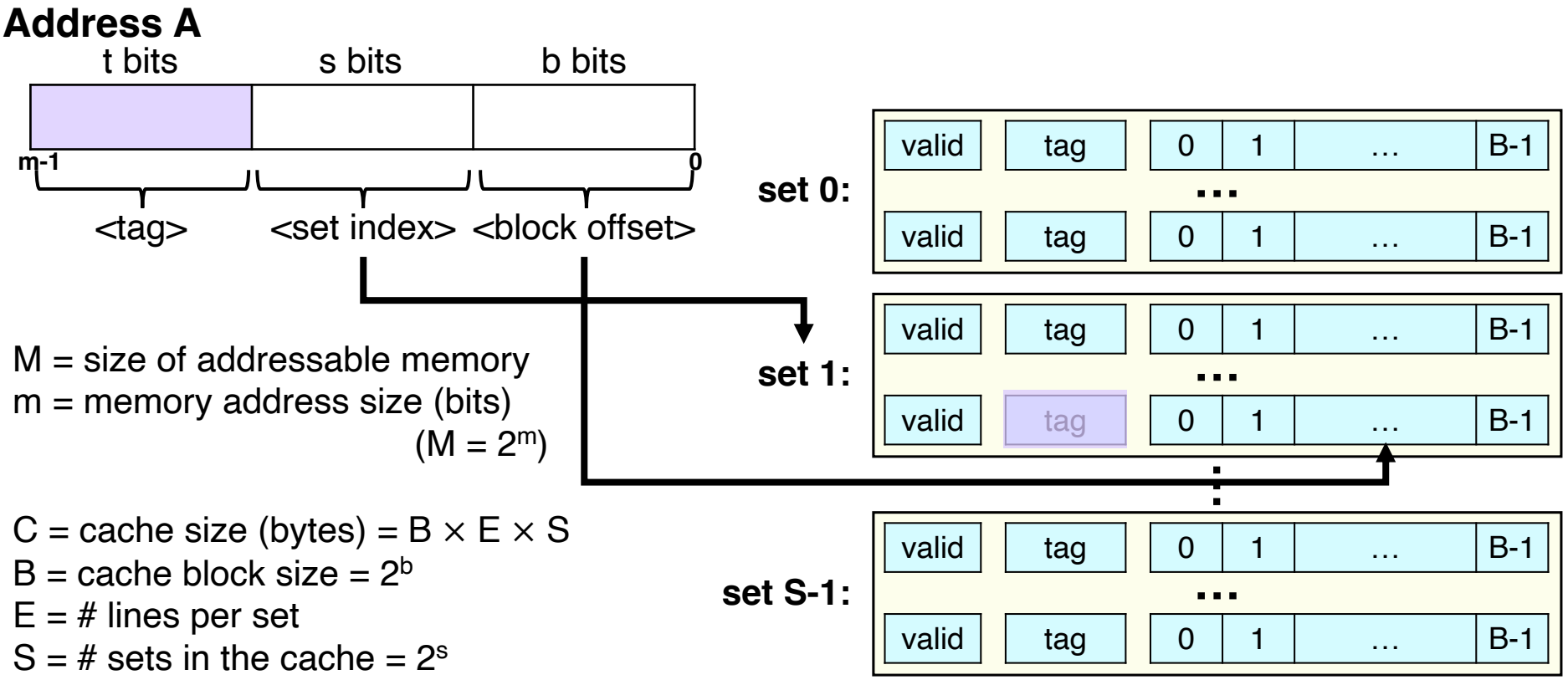


# Cache Structure Parameters




**Cache size: C = B × E × S data bytes**

# Cache Structure Parameters




M = size of addressable memory
m = memory address size (bits)
($M = 2^m$)

C = cache size (bytes) = B × E × S
B = cache block size = $2^b$
E = # lines per set
S = # sets in the cache = $2^s$

s = # bits for set offset
t = # tag bits
b = # bits for block offset

m = t + s + b

# Cache Size Example

| example | m | C | B | E | S | t | s | b |
|---|---|---|---|---|---|---|---|---|
| 1 | 32 | 1024 | 4 | 1 | | | | |
| 2 | 32 | 1024 | 8 | 4 | | | | |
| 3 | 32 | 1024 | 32 | 32 | | | | |
| 4 | 64 | 2048 | 64 | 16 | | | | |

# Cache Size Example

| example | m | C | B | E | S | t | s | b |
|---|---|---|---|---|---|---|---|---|
| 1 | 32 | 1024 | 4 | 1 | 256 | 22 | 8 | 2 |
| 2 | 32 | 1024 | 8 | 4 | | | | |
| 3 | 32 | 1024 | 32 | 32 | | | | |
| 4 | 64 | 2048 | 64 | 16 | | | | |

# Cache Size Example

| example | m | C | B | E | S | t | s | b |
|---|---|---|---|---|---|---|---|---|
| 1 | 32 | 1024 | 4 | 1 | 256 | 22 | 8 | 2 |
| 2 | 32 | 1024 | 8 | 4 | 32 | 24 | 5 | 3 |
| 3 | 32 | 1024 | 32 | 32 | | | | |
| 4 | 64 | 2048 | 64 | 16 | | | | |

# Cache Size Example

| example | m | C | B | E | S | t | s | b |
|---|---|---|---|---|---|---|---|---|
| 1 | 32 | 1024 | 4 | 1 | 256 | 22 | 8 | 2 |
| 2 | 32 | 1024 | 8 | 4 | 32 | 24 | 5 | 3 |
| 3 | 32 | 1024 | 32 | 32 | 1 | 27 | 0 | 5 |
| 4 | 64 | 2048 | 64 | 16 | | | | |

# Cache Size Example

| example | m | C | B | E | S | t | s | b |
|---|---|---|---|---|---|---|---|---|
| 1 | 32 | 1024 | 4 | 1 | 256 | 22 | 8 | 2 |
| 2 | 32 | 1024 | 8 | 4 | 32 | 24 | 5 | 3 |
| 3 | 32 | 1024 | 32 | 32 | 1 | 27 | 0 | 5 |
| 4 | 64 | 2048 | 64 | 16 | 2 | 57 | 1 | 6 |

# Class Activity – 9.1

| problem | m | C | B | E | S | t | s | b |
|---|---|---|---|---|---|---|---|---|
| 1 | 32 | 4096 | 8 | 1 | | | | |
| 2 | 64 | 1024 | 8 | | 8 | | | |
| 3 | 32 | | | 32 | | | 2 | 5 |
| 4 | 64 | 2048 | 64 | | 16 | | | |