CS 211: Intro to Computer Architecture

13.2: Memory Access Patterns and Caching

Minesh Patel

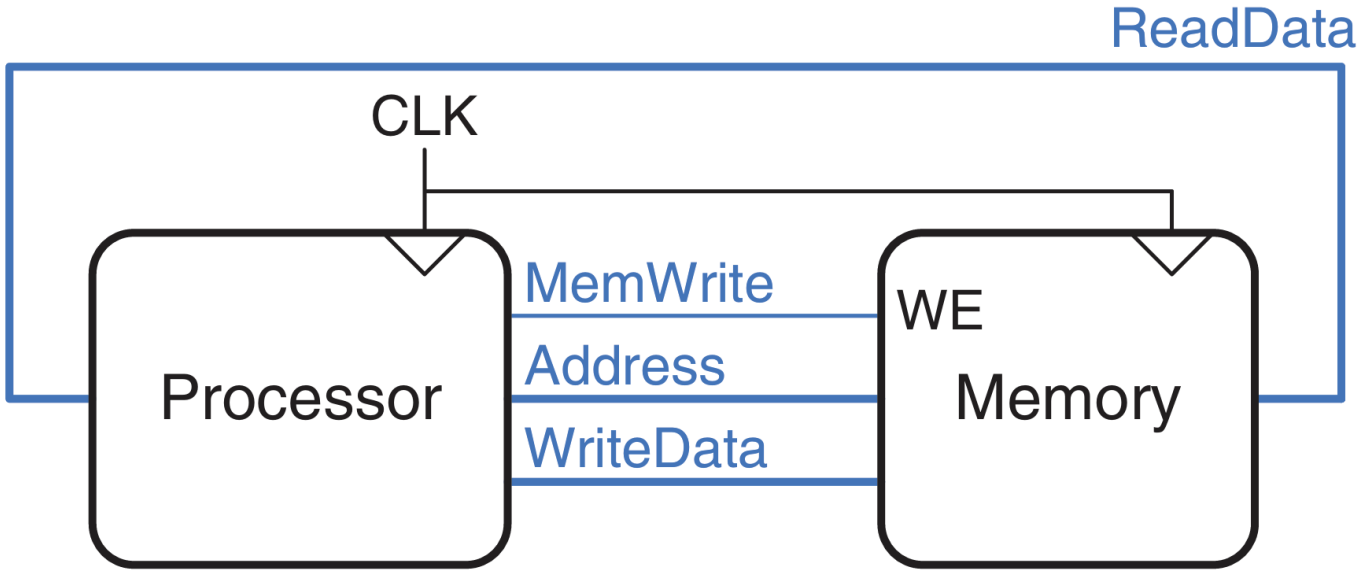
Spring 2025 – Thursday 24 April

Announcements

- Ongoing
  - **WA9:** due **Friday (April 18) @ 11:59 pm**
  - **PA5:** due **Monday (May 5) @ 11:59 pm**

- Upcoming
  - **WA10:** TBA: planned for Friday
  - **Final Exam:** May 14th

Our Mental Model (So Far)



- In reality, we have **different types of memory**
- **Today: registers** vs. **caches** vs. **main memory**

Agenda

- **Why Is Main Memory Slow?**

- Caching and the Memory Hierarchy

- Direct-Mapped Caches
  - Where to Place Data
  - How to Query the Cache

Semiconductor Memories

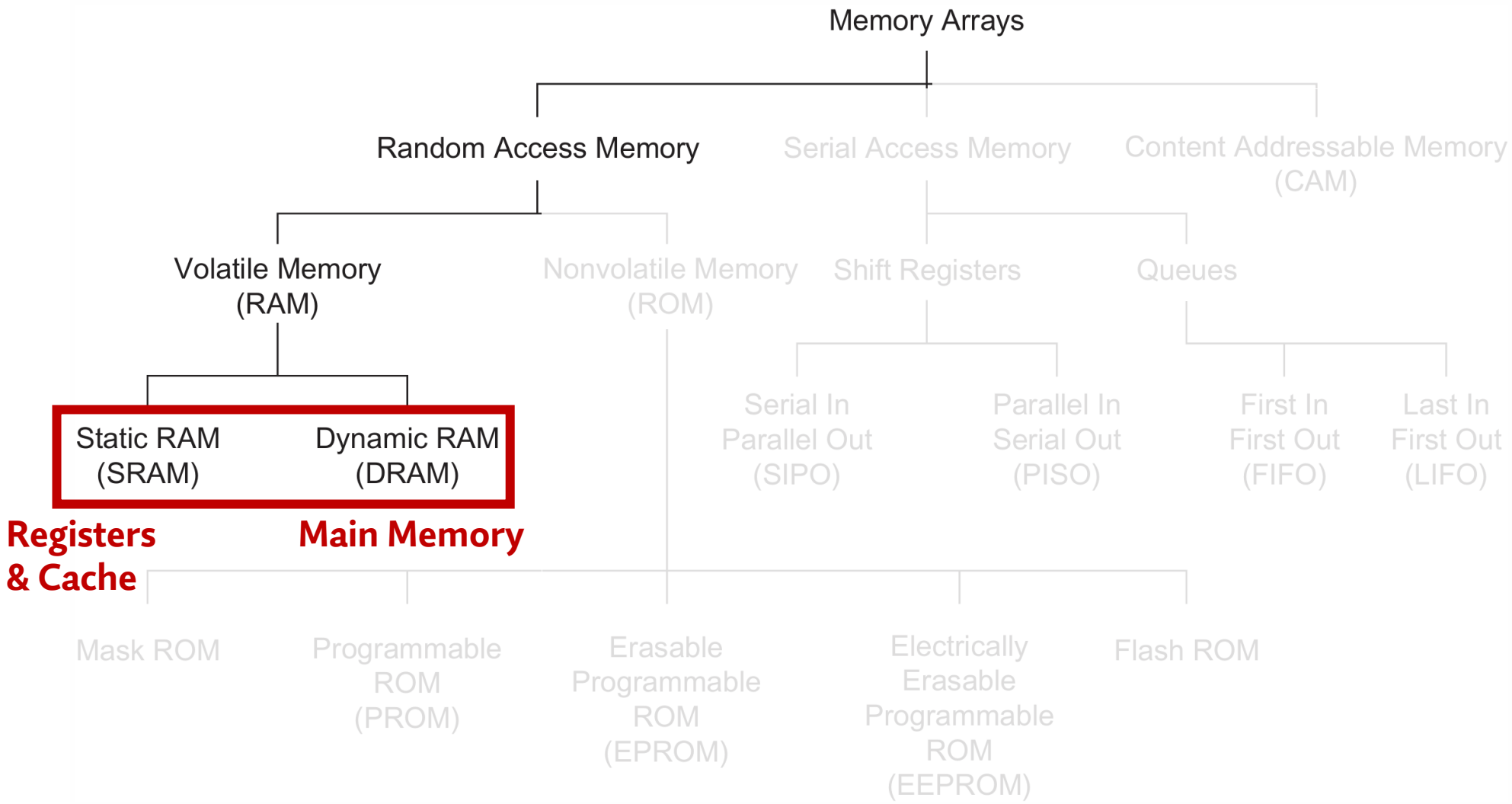
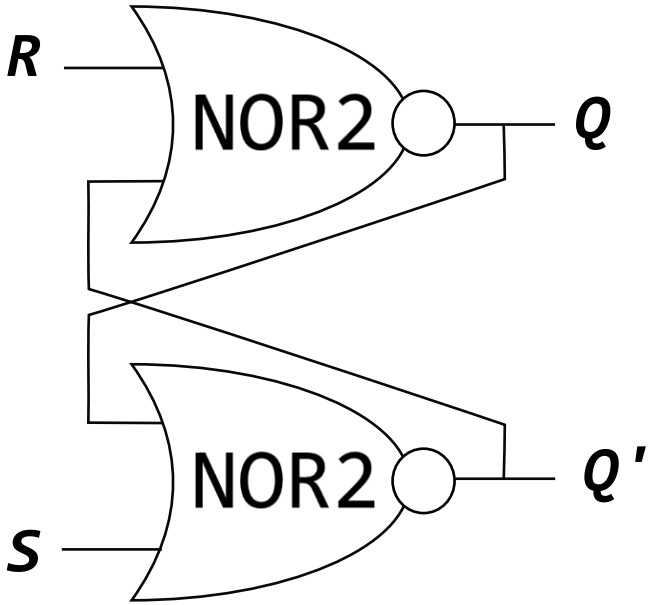


FIGURE 12.1 Categories of memory arrays

Main Memory vs. Registers

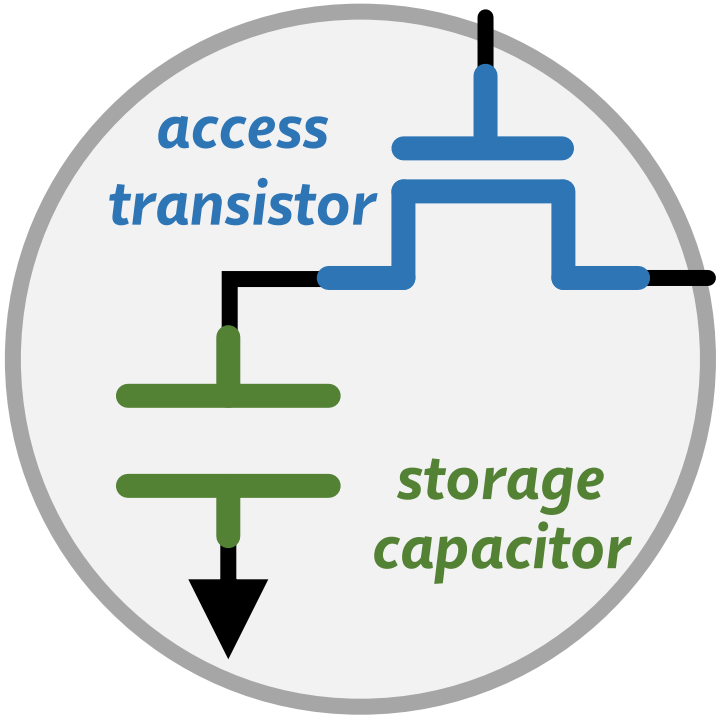
Registers (Flip-Flops)

Built from digital logic



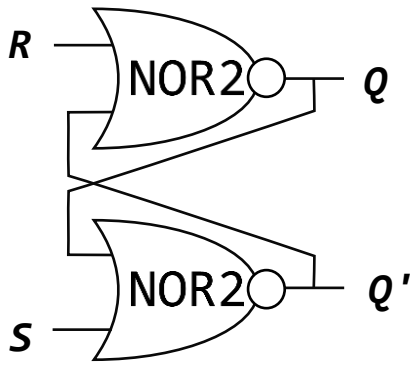
Main Memory (DRAM)

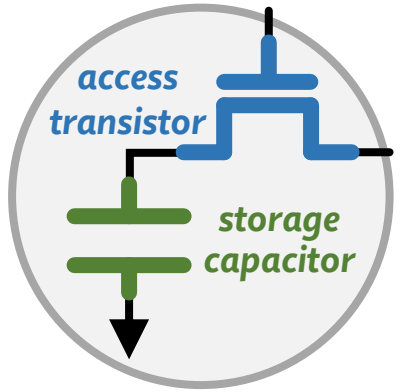
Built from analog circuits



each stores one bit of data

Main Memory vs. Registers

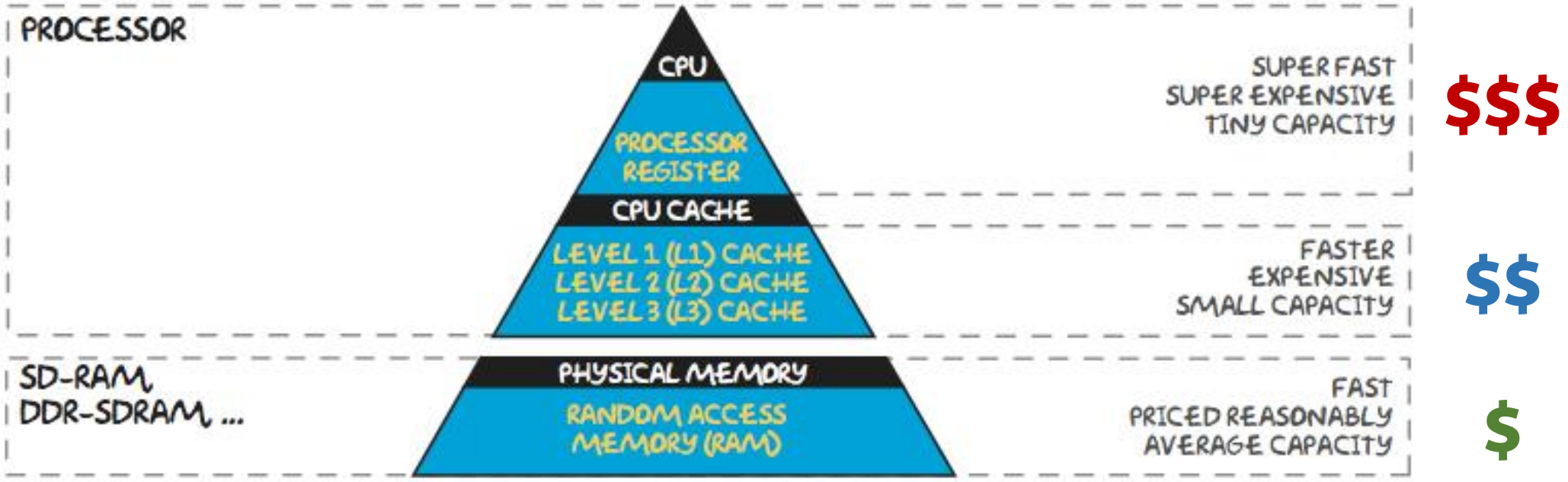




	Registers (Flip-Flops) *Built from digital logic*	Main Memory (DRAM) *Built from analog circuits*
Cost	Expensive	Cheap
Speed	Fast	Slow
Physical Size	Big	Small

Even If I Were a Billionaire….

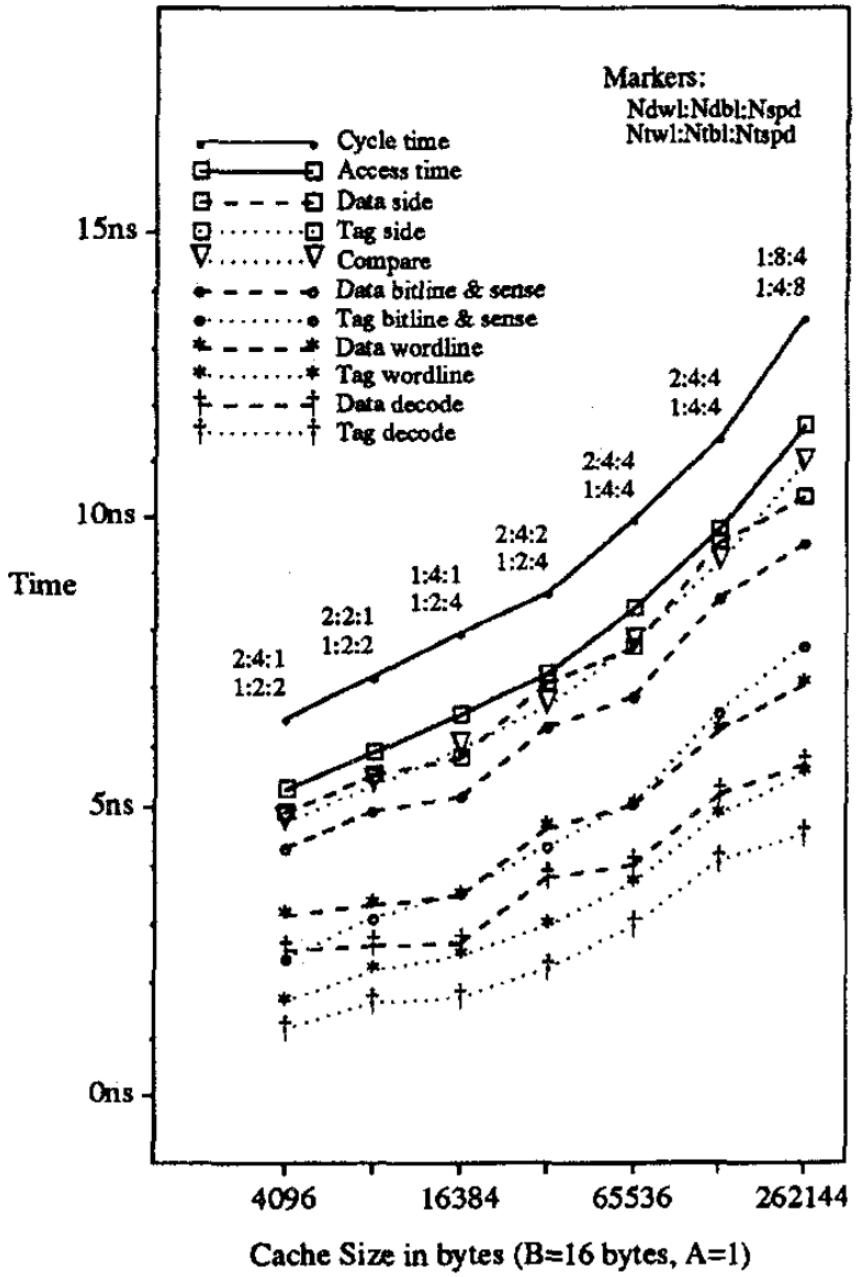
- I **still couldn't afford** to keep all data in CPU registers



https://zjustin.wordpress.com/2016/10/11/computer-memory-reflection/

Beyond Cost: Access Latency

- Fundamentally: **bigger = slower**
  - Electrical signals travel further
  - More wire resistance
  - …
- **Low-capacity** memories are fastest

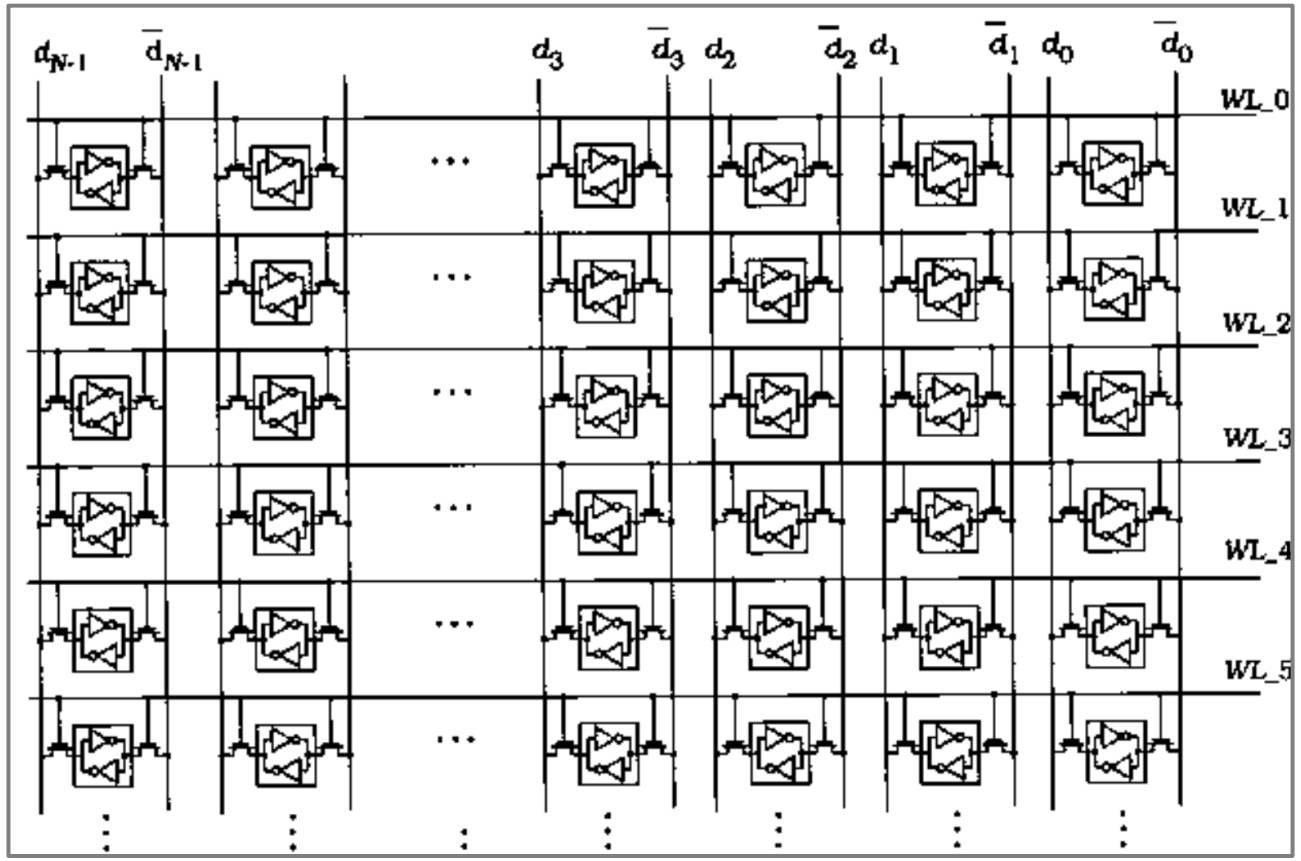


Wilton and Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," JSSC, 2002.

Beyond Cost: Density

Static RAM
(Registers, Caches)

Built from digital logic



Dynamic RAM
(Main Memory)

Built from analog circuits

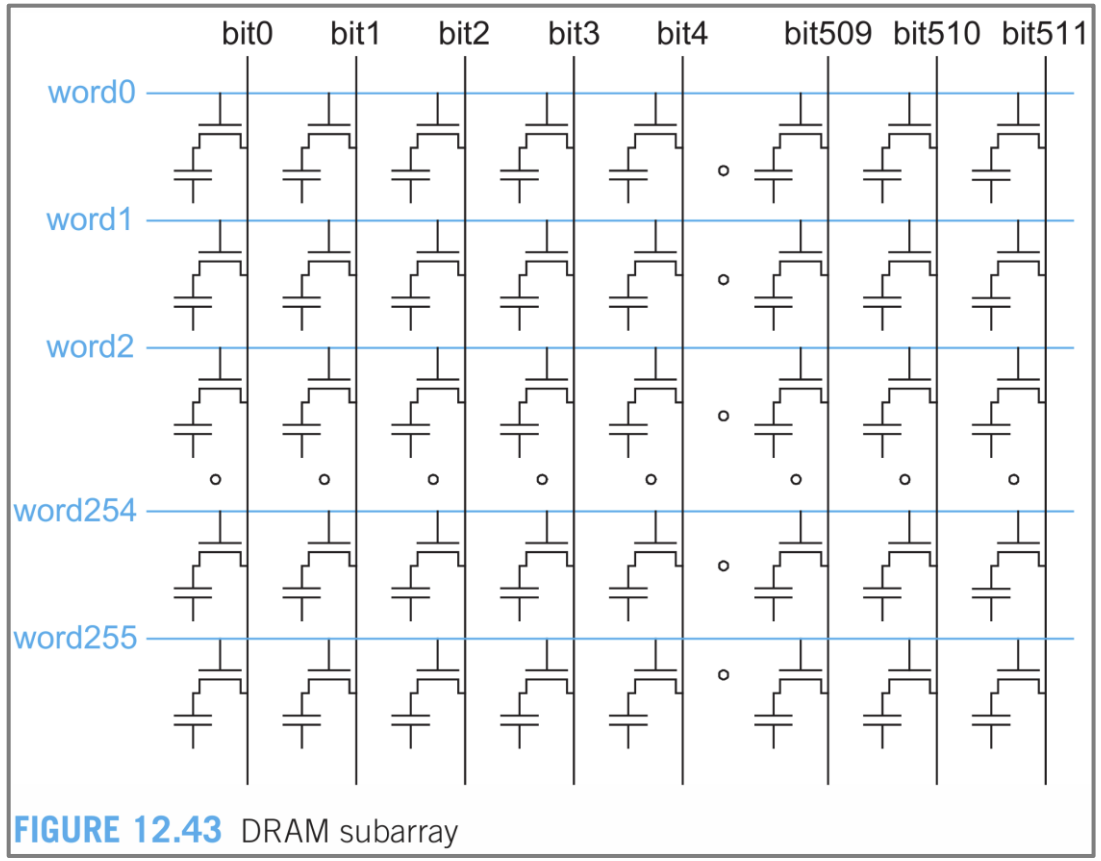


FIGURE 12.43 DRAM subarray

Beyond Cost: Energy

Energy Cost of Various CPU Operations

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3.1pJ

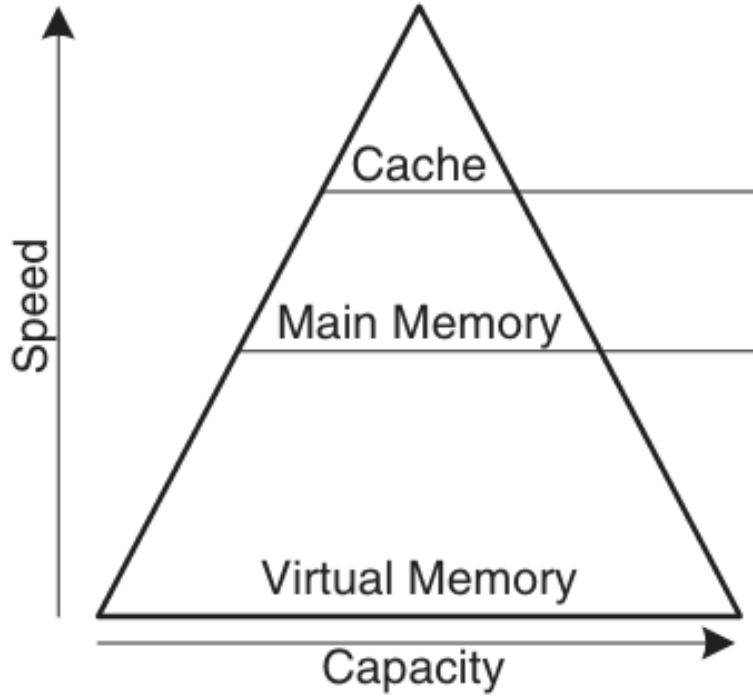
FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1.1pJ
32 bit	3.7pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ

10,000x

Horowitz, Mark. "Computing's Energy Problem (and What We Can Do About It)," ISSCC, 2014.

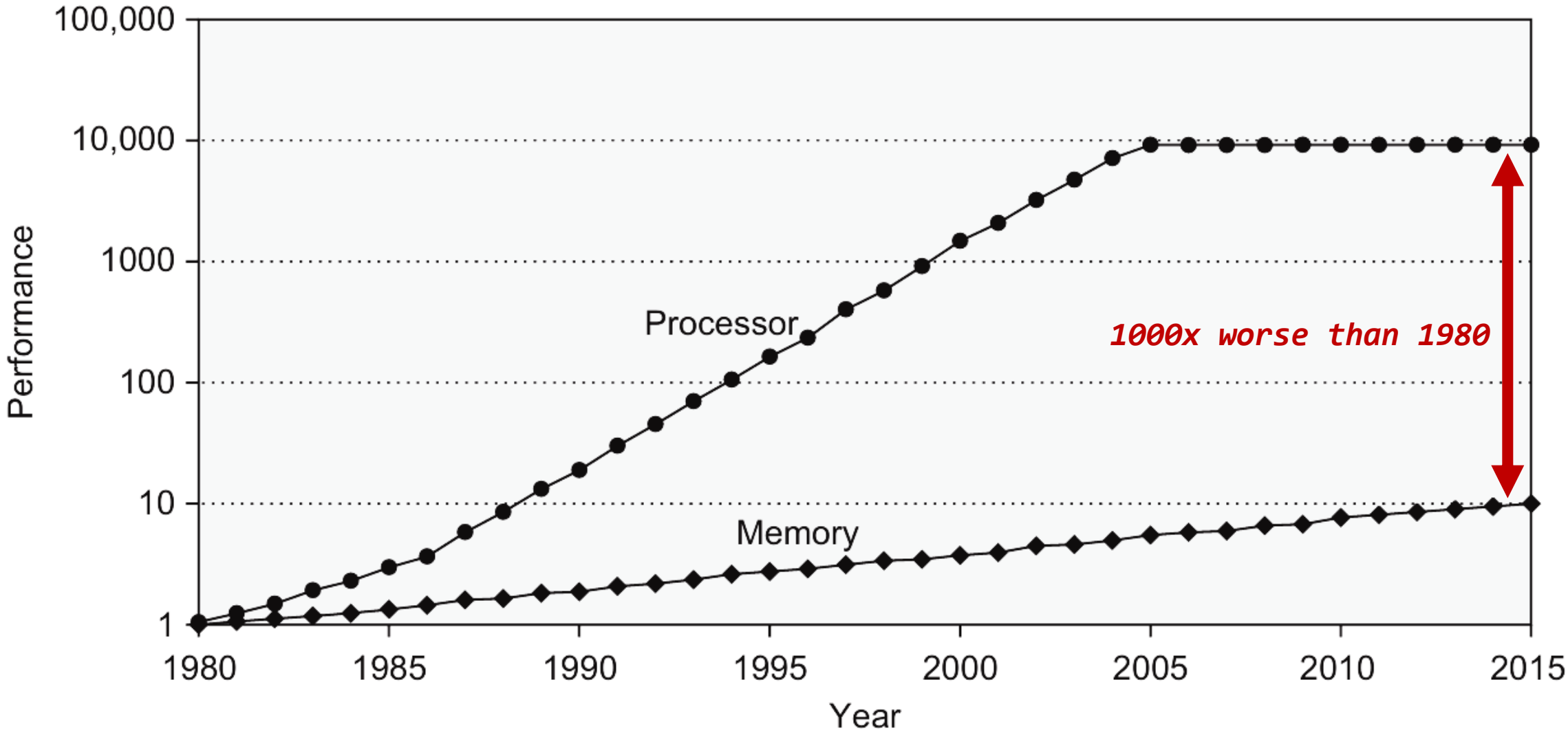
Cost-Performance Tradeoff



Technology	Price / GB	Access Time (ns)
SRAM	$100	0.2–3
DRAM	$3	10–50
SSD	$0.10	20,000
HDD	$0.03	5,000,000

- There are **other concerns** too (e.g., energy, reliability, etc...)

It's Just Hard to Improve Memory



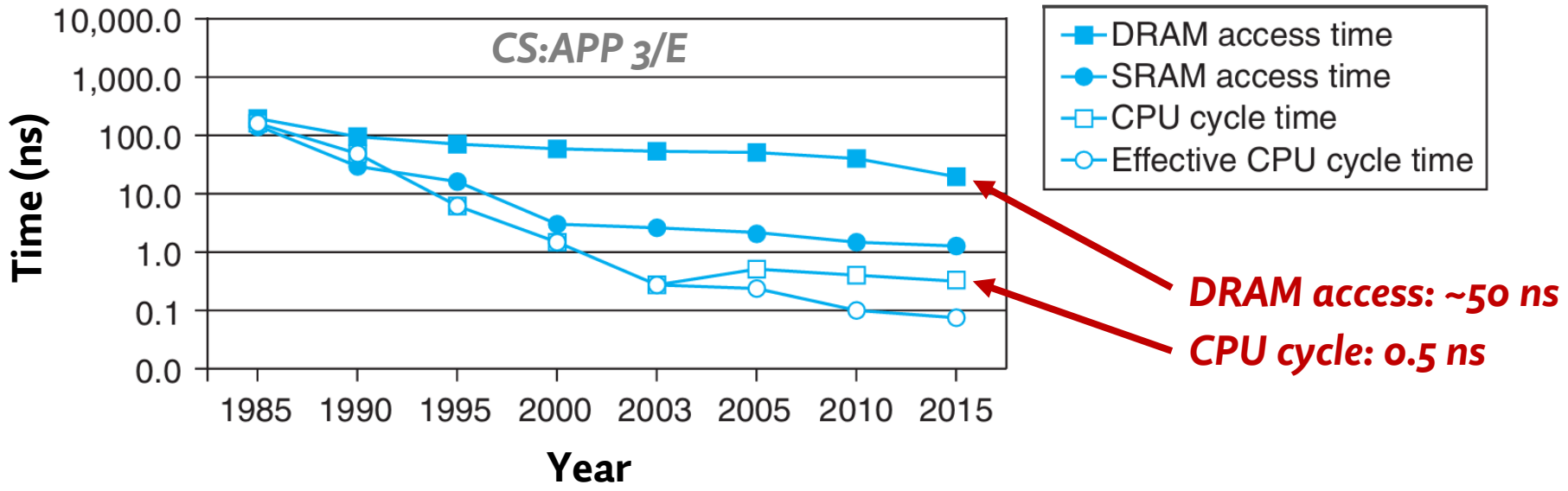
Hennessy & Patterson, "A Quantitative Approach to Computer Architecture," 6/E., Figure 2.2

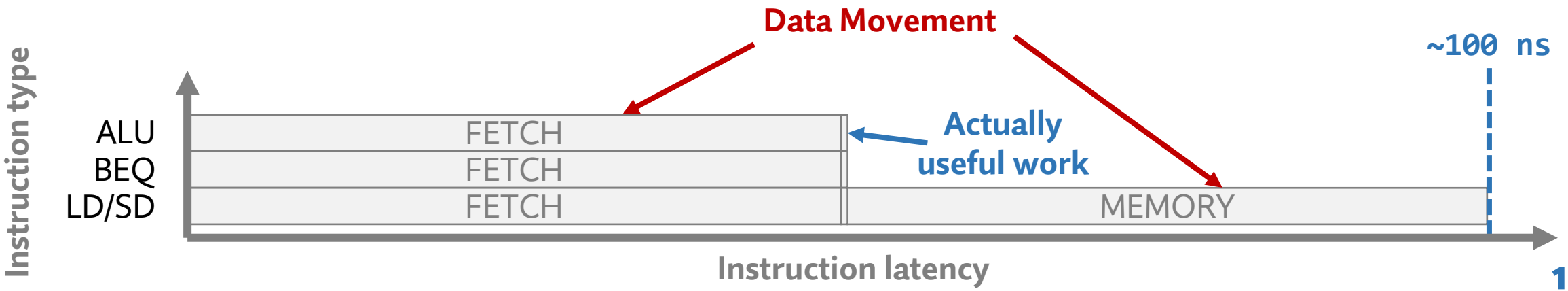
Agenda

- Why Is Main Memory Slow?
- **Caching and the Memory Hierarchy**
- Direct-Mapped Caches
  - Where to Place Data
  - How to Query the Cache

Main Memory Kills Performance

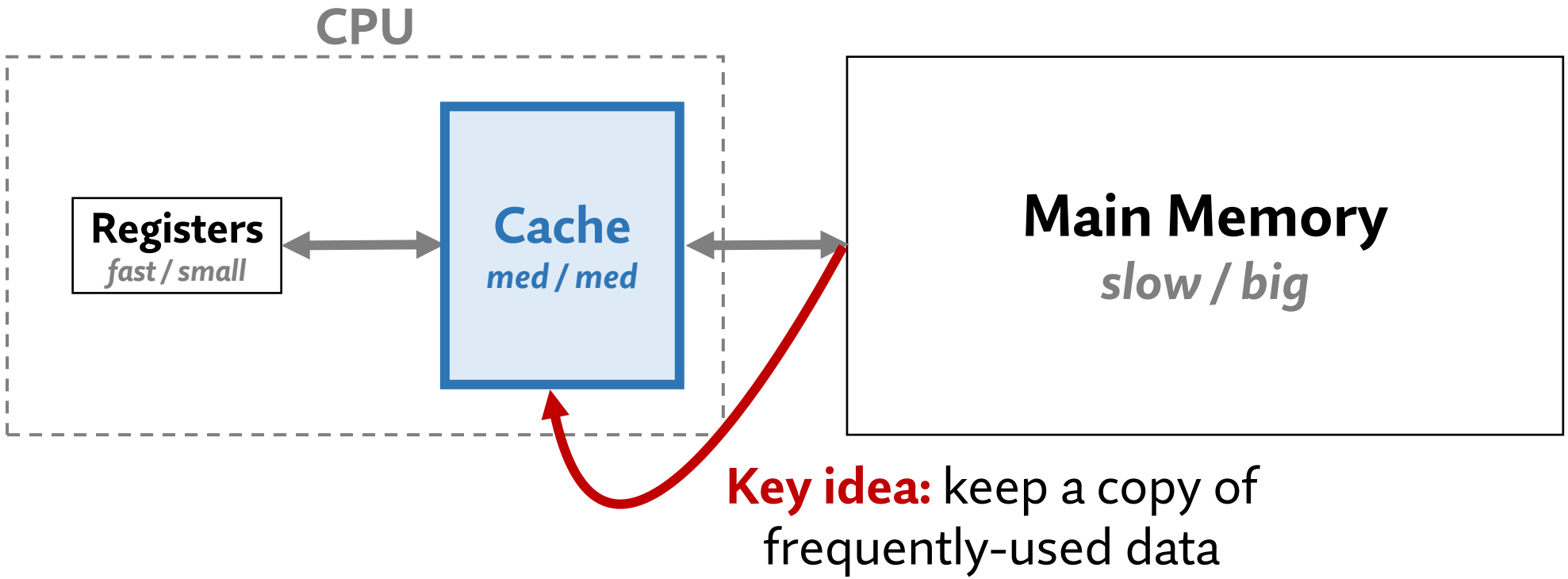
- The CPU wastes lots of time **just moving data**





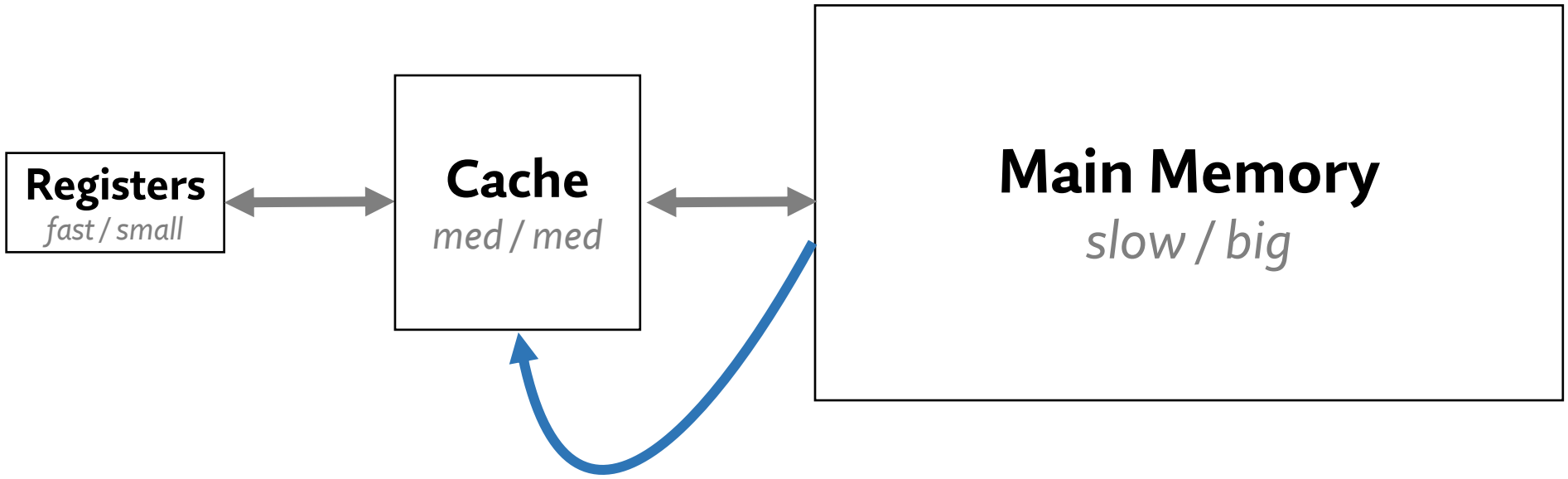
Getting the Best of Both Worlds

- **Goal:** Give the **illusion** that main memory is **large but fast**



- If we cache **the right data**, the CPU never goes to main memory
  - **Common case:** accesses are at the speed of cache
  - **Worst case:** data not in the cache

The Right Data to Cache



Million-dollar question:
What should we cache?

- **A:** Data that we know we will need soon (prediction problem!)
  - **Cache hit:** data is in the cache - fast
  - **Cache miss:** data is NOT in the cache - slow

Choosing What to Cache

- The answer comes from studying application behavior

```
void func(uint64_t *a, uint64_t N)
{
    for(int i = 0; i < N; i++)
        a[i]++;
}
```

```
func:
    slli a1, a1, 3
    add  a1, a0, a1
.L2:
    bne  a0, a1, .L3
    ret
.L3:
    ld   a5, 0(a0)
    addi a0, a0, 8
    addi a5, a5, 1
    sd   a5, -8(a0)
    j    .L2
```

Memory Access Pattern

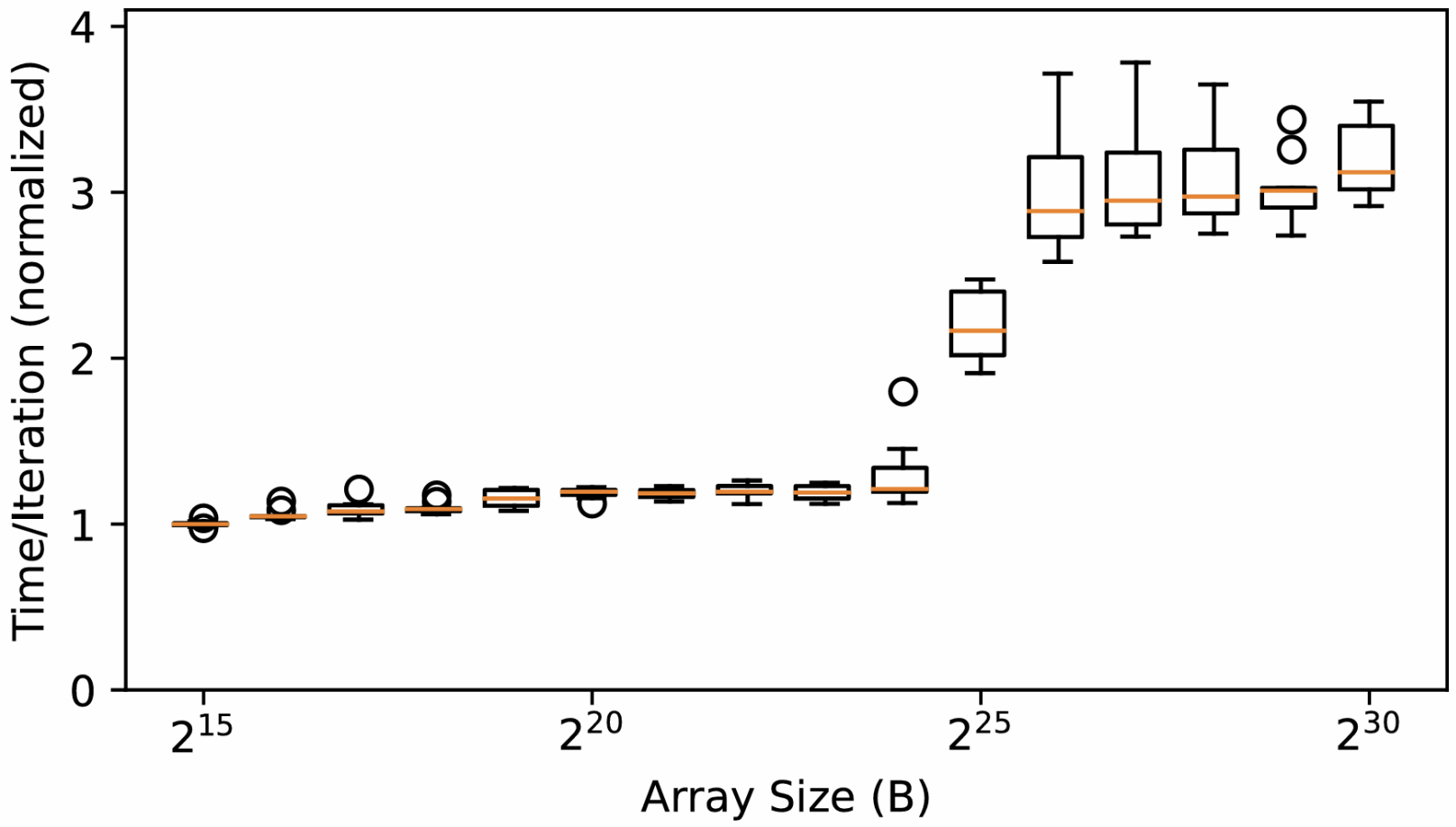
```
read(A[0])
write(A[0])
read(A[1])
write(A[1])
    ...
read(A[N-1])
write(A[N-1])
```

Observation 1: Re-use

Observation 2: Adjacency

Benchmarking ilab2

```
for(uint64_t i = 0; i < array_size; i += stride)
    temp = arr[i];
```




```
    li   a4, 0
.loop:
    bge  a4, a2, .done
    ld   t1, 0(t0)
    add  a4, a4, a3
    add  t0, t0, a3
    j    .loop
. done:
    ...
```

Example: Matrix Multiply (C = A*B)

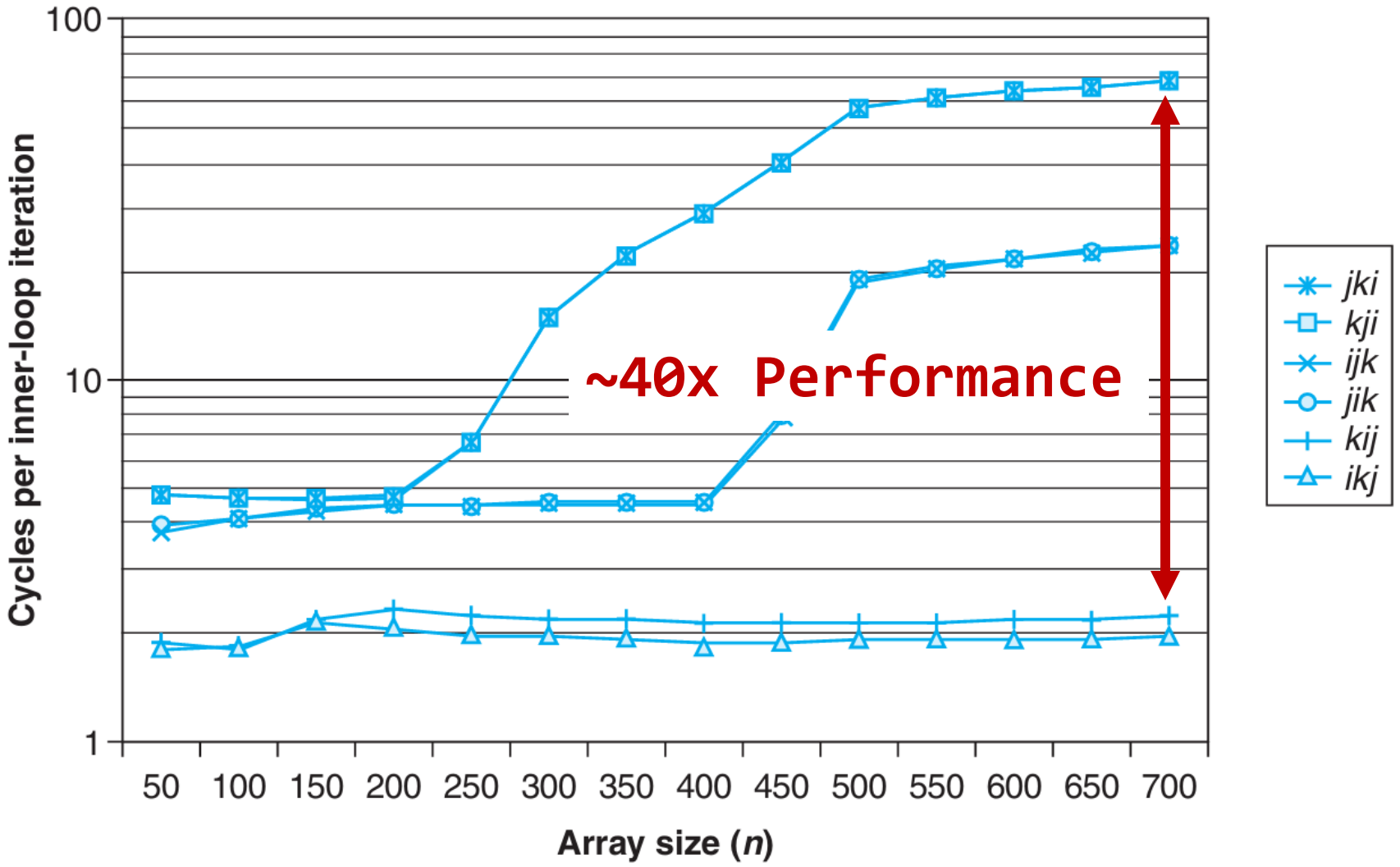


Figure 6.46 Core i7 matrix multiply performance.

(a) Version *ijk*

code/mem/matmult/mm.c

```
1   for (i = 0; i < n; i++)
2       for (j = 0; j < n; j++) {
3           sum = 0.0;
4           for (k = 0; k < n; k++)
5               sum += A[i][k]*B[k][j];
6           C[i][j] += sum;
7       }
```

code/mem/matmult/mm.c

(c) Version *jki*

code/mem/matmult/mm.c

```
1   for (j = 0; j < n; j++)
2       for (k = 0; k < n; k++) {
3           r = B[k][j];
4           for (i = 0; i < n; i++)
5               C[i][j] += A[i][k]*r;
6       }
```

code/mem/matmult/mm.c

(e) Version *kij*

code/mem/matmult/mm.c

```
1   for (k = 0; k < n; k++)
2       for (i = 0; i < n; i++) {
3           r = A[i][k];
4           for (j = 0; j < n; j++)
5               C[i][j] += r*B[k][j];
6       }
```

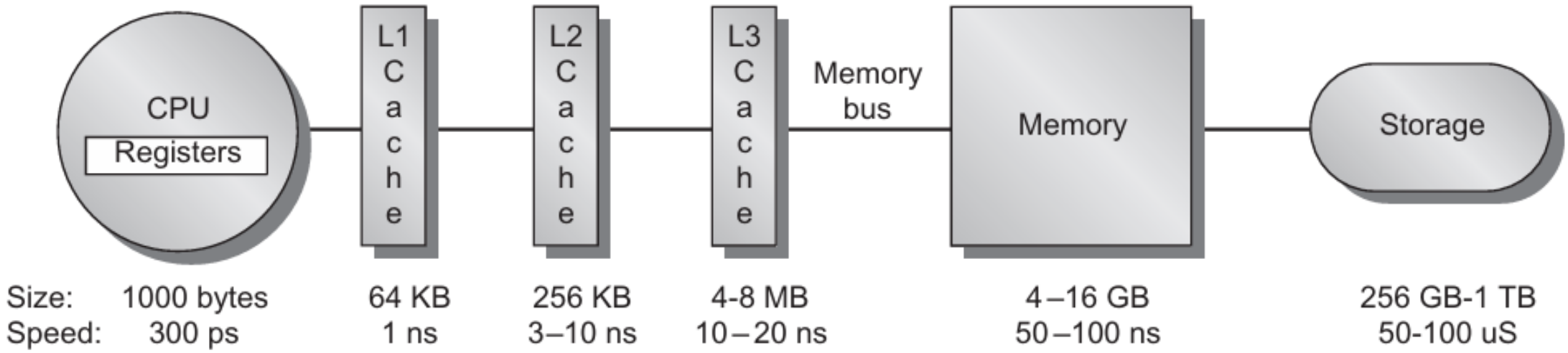
code/mem/matmult/mm.c

Figure 6.44 Six versions of matrix multiply. Each

CS:APP 3/E, "Section 6.6.2"

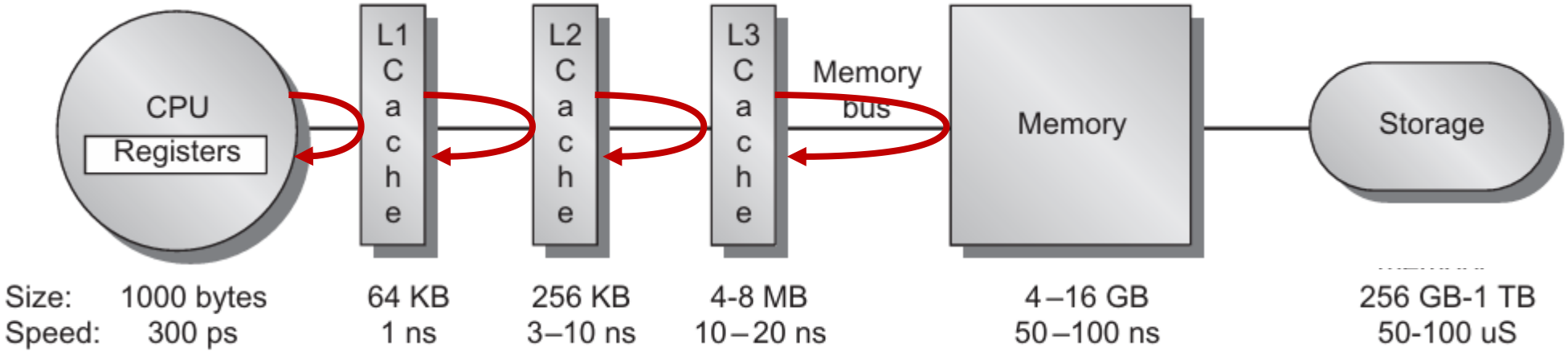
The Memory Hierarchy

- **Key idea:** Have successively larger caches
  - **"Hot data"** close to CPU (closer the better)
  - **"Cold data"** far from CPU



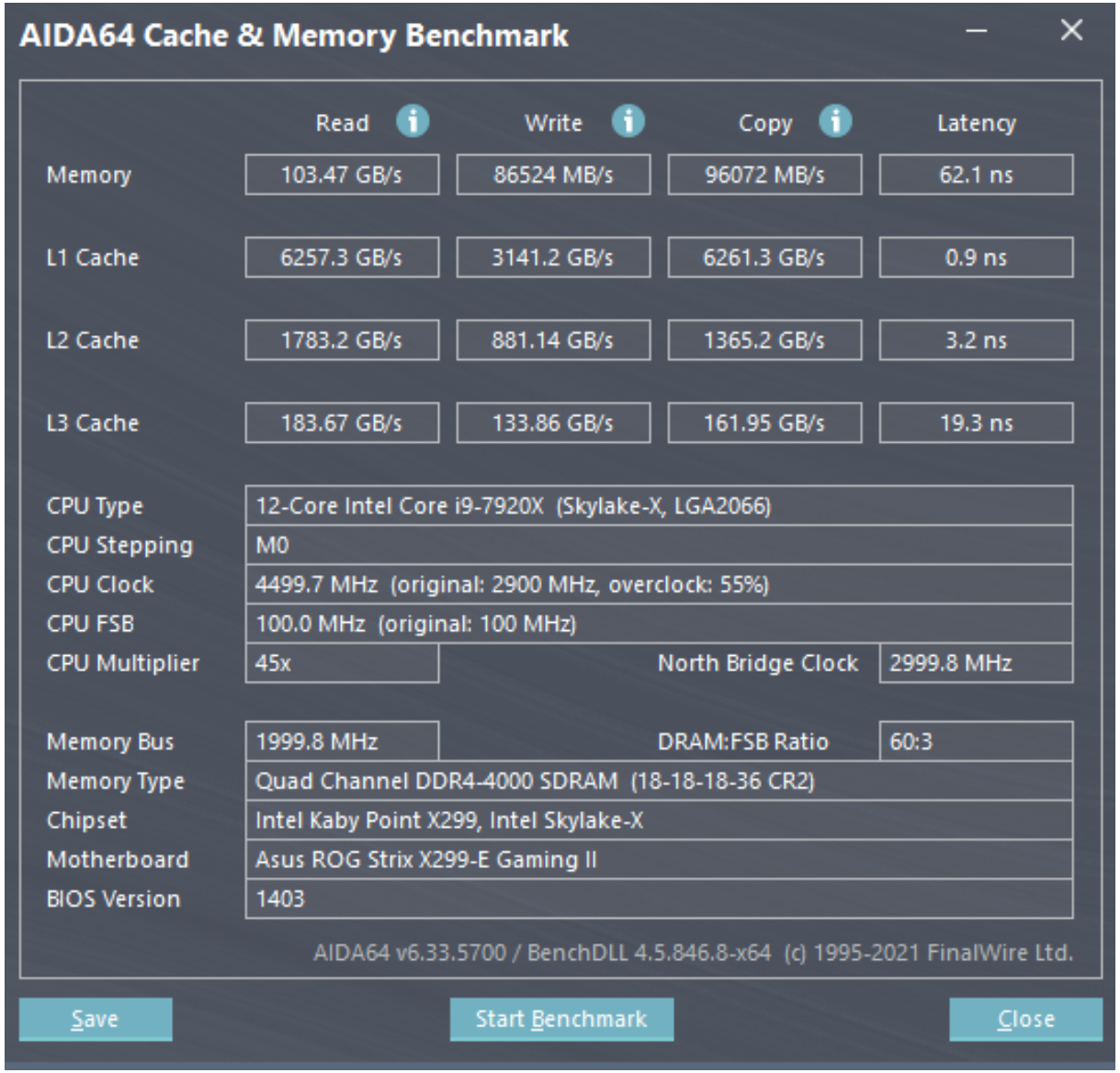
Hit/Miss Performance

- Performance depends on whether there is a **cache hit or miss**



- `L1 hit:`
- `L1 miss + L2 hit:`
- `L1 miss + L2 miss + L3 hit:`
- `L1 miss + L2 miss + L3 miss + Memory:`

Testing Memory/Cache Latency



https://www.techpowerup.com/forums/threads/share-your-aida-64-cache-and-memory-benchmark-here.186338/page-99

Cache Design Questions

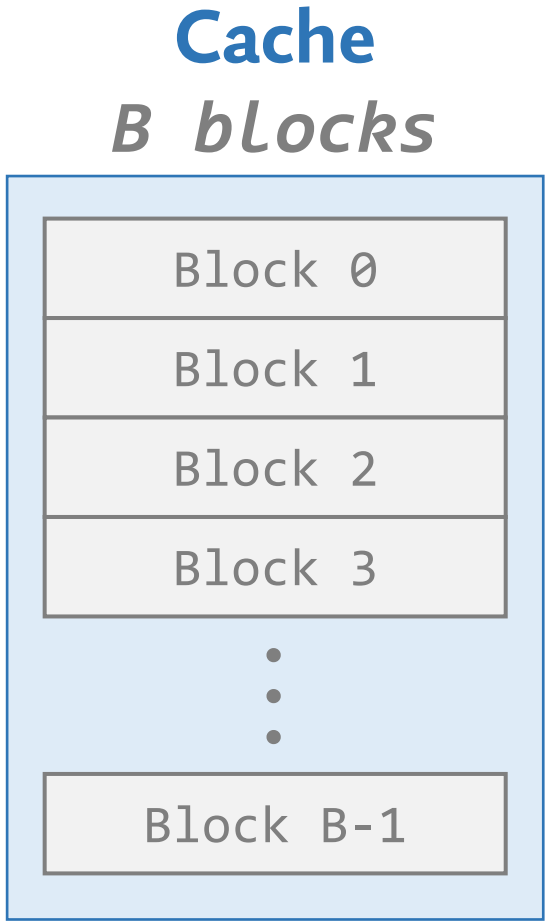
- **Which data** should we cache?
- How do we **access the cache**?
- **Where** in the cache should that data go?
- What happens when the cache **is full**?
- What if we **modify** data in the cache?

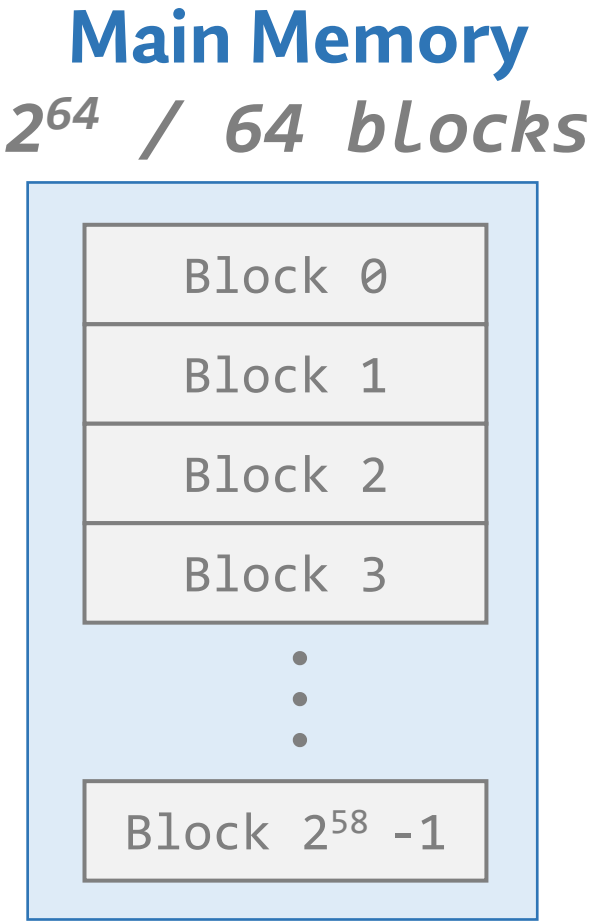
Agenda

- Why Is Main Memory Slow?
- Caching and the Memory Hierarchy
- **Direct-Mapped Caches**
  - Where to Place Data
  - How to Query the Cache

Cache Blocks

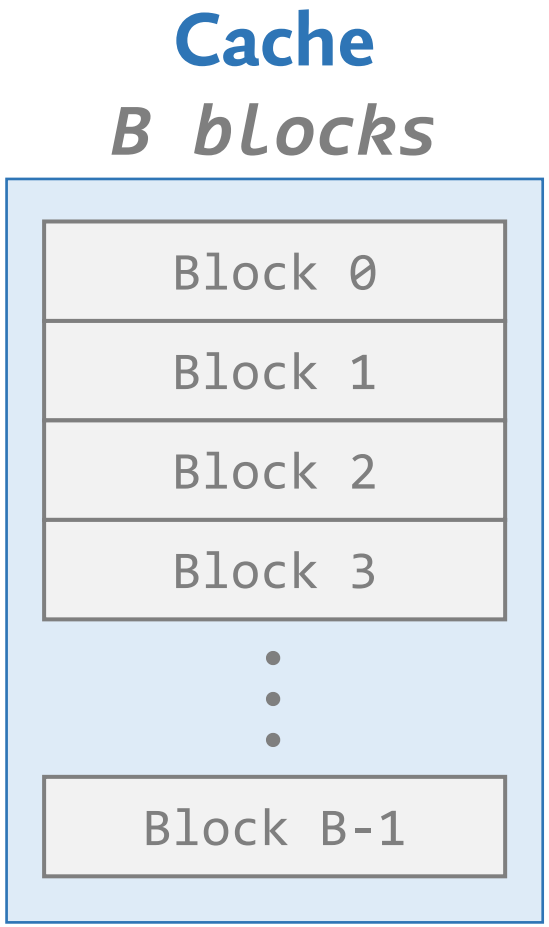
- Caches store **"blocks" of data** (usually 64 bytes)
- Think about it like a smaller version of main memory





Types of Caches

- **Q:** Suppose we issue an **`ld`** (8B): **which block** does it go in?



Fully Associative: Anywhere

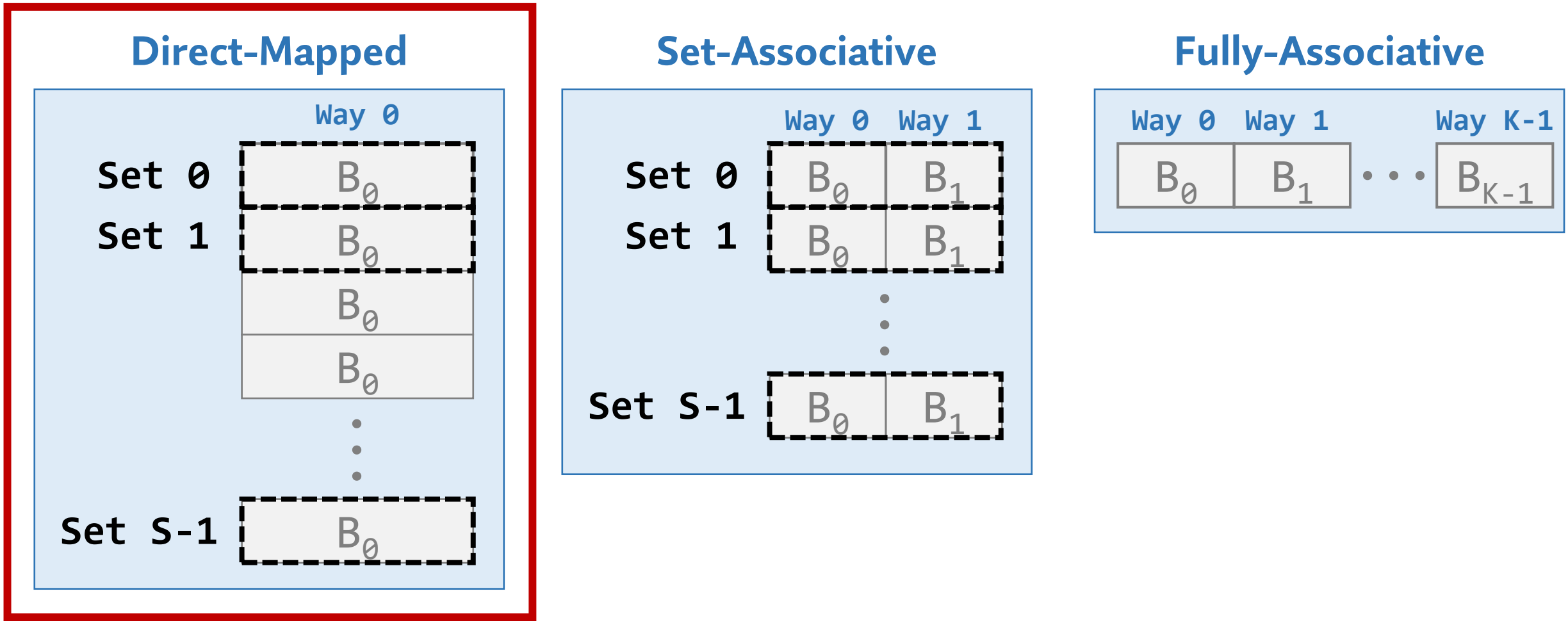
Set-Associative: Specific places

Direct Mapped: Exactly one place

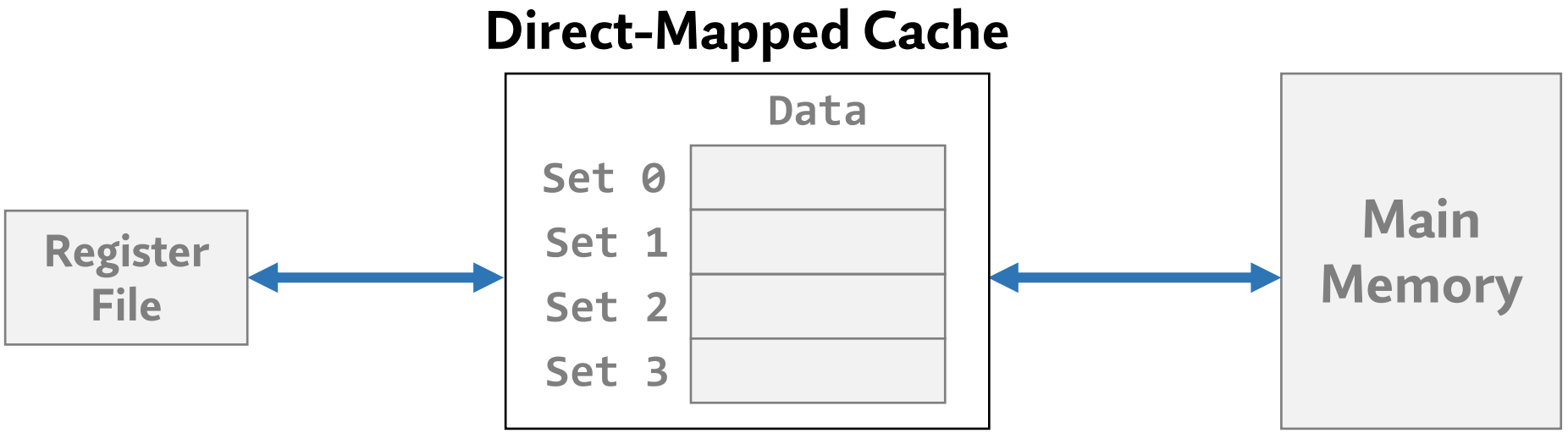
Today

Cache Organizations: Terminology

- Rows (**sets**) and columns (**ways**) of blocks



Toward a Direct-Mapped Cache



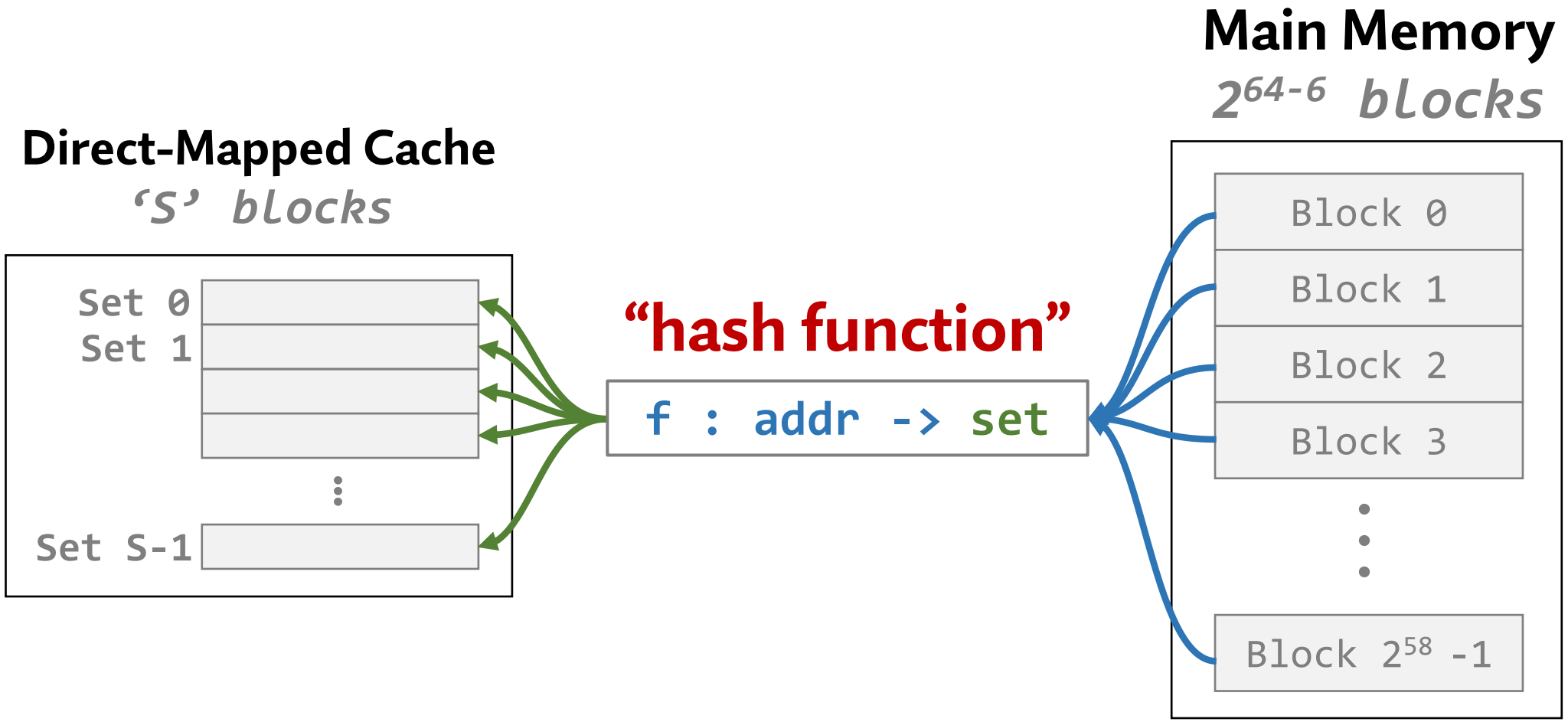
- **Design questions:**
  - Where to place data?
  - How to query the cache?

Agenda

- Why Is Main Memory Slow?
- Caching and the Memory Hierarchy
- Direct-Mapped Caches
  - **Where to Place Data**
  - How to Query the Cache

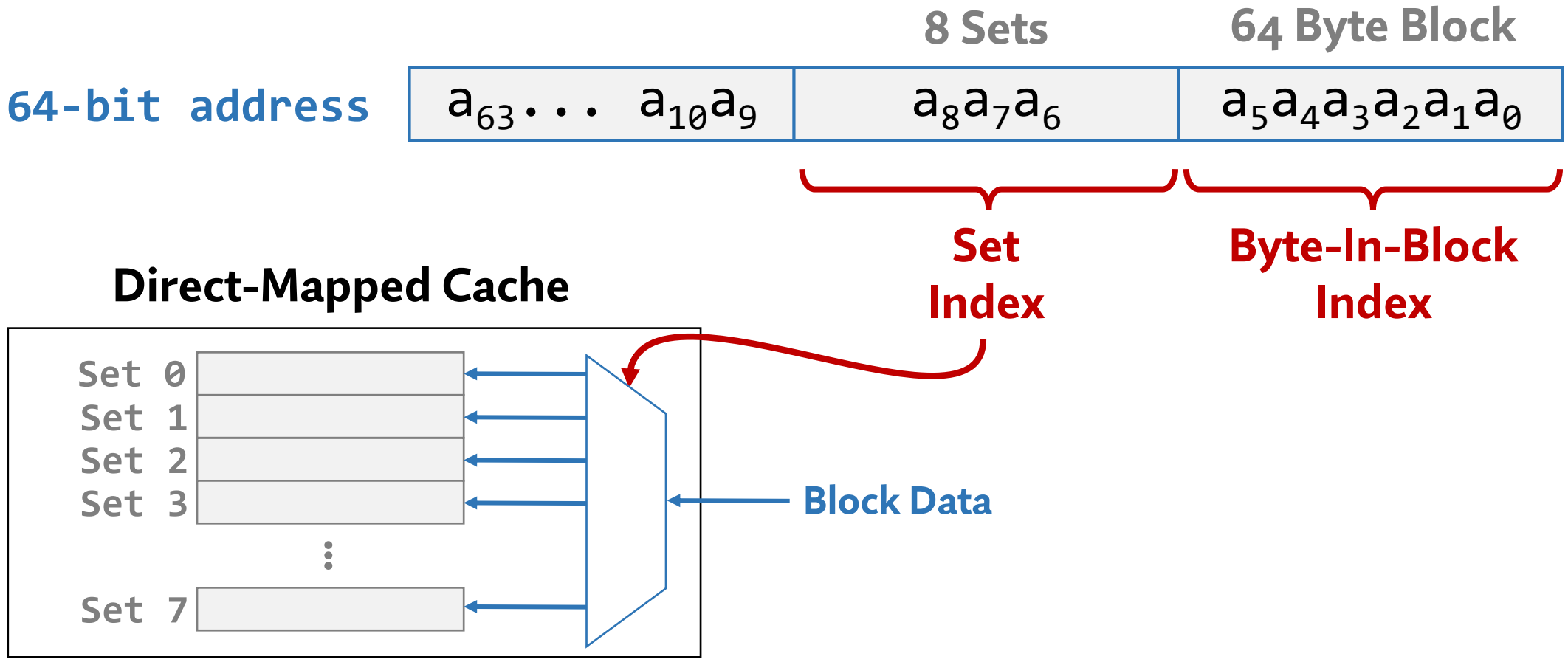
Mapping Data to a Direct-Mapped Cache

- Need to map **every memory location** to a specific set

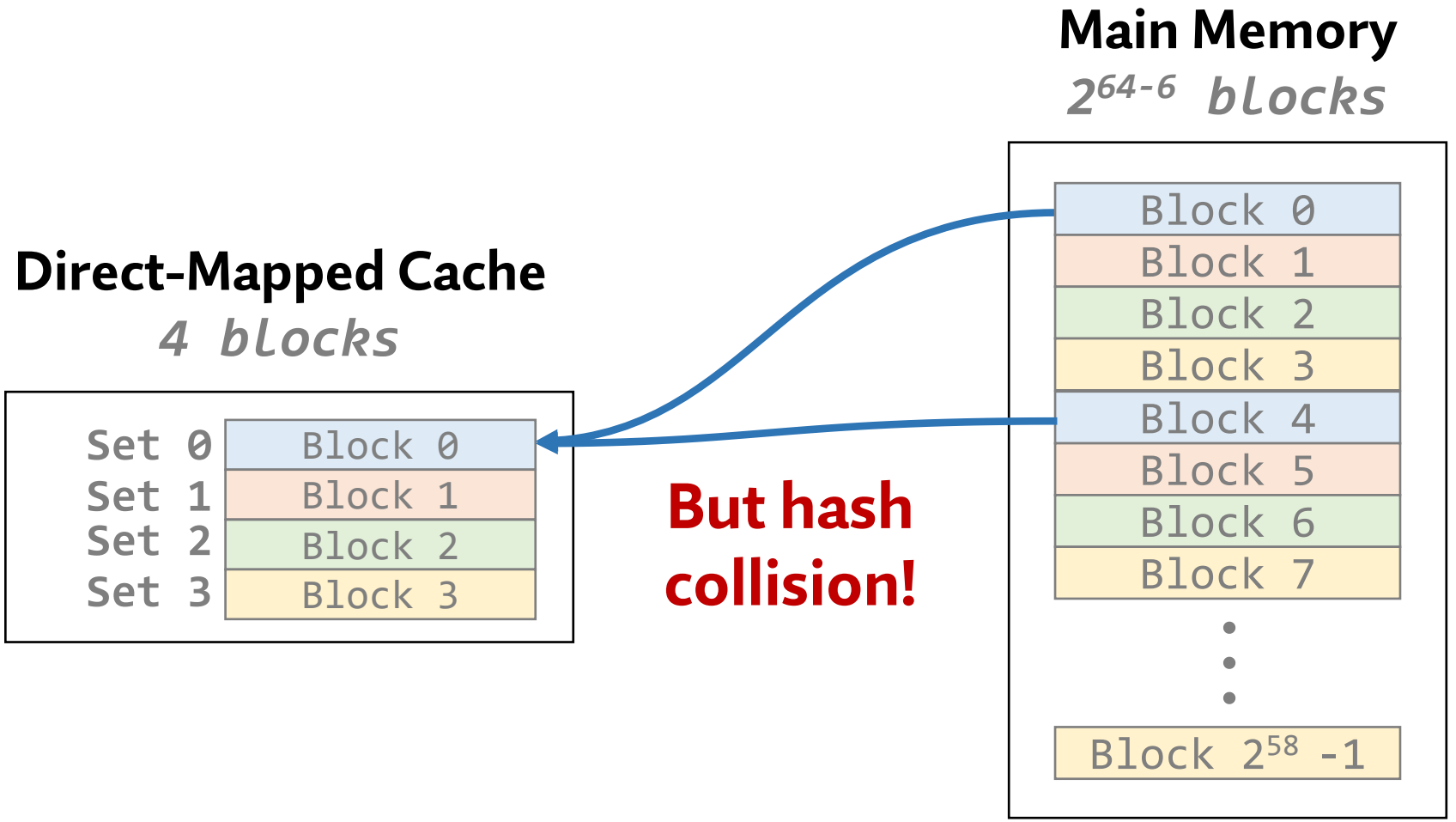


Set Index Hash Function

- We could use something fancy (lots of research on this!)
- **Simplest:** just use bits of the address

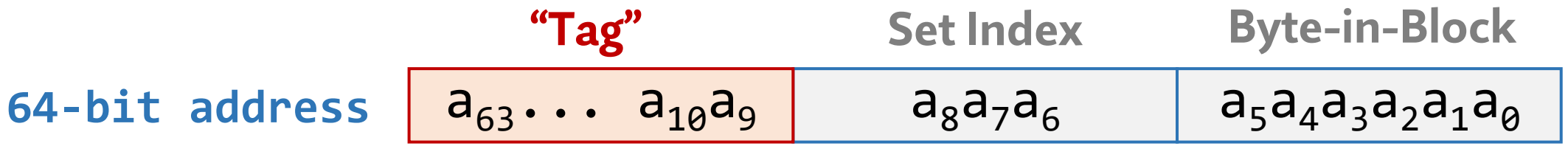


Visualizing the Cache Mapping

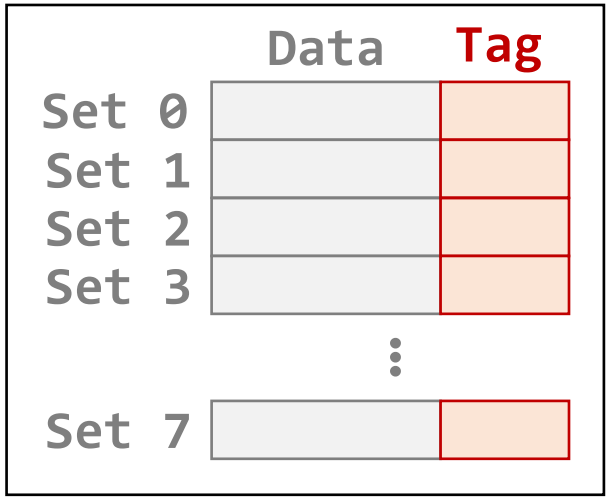


Resolving Hash Collisions

- **Solution:** cache **both data and address**



Direct-Mapped Cache



Must **compare tag bits** on every cache access to check for collision

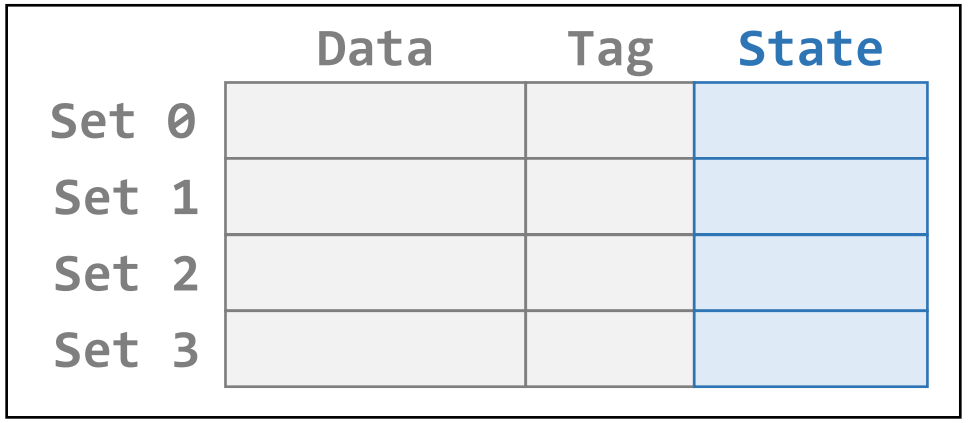
Agenda

- Why Is Main Memory Slow?
- Caching and the Memory Hierarchy
- Direct-Mapped Caches
  - Where to Place Data
  - **How to Query the Cache**

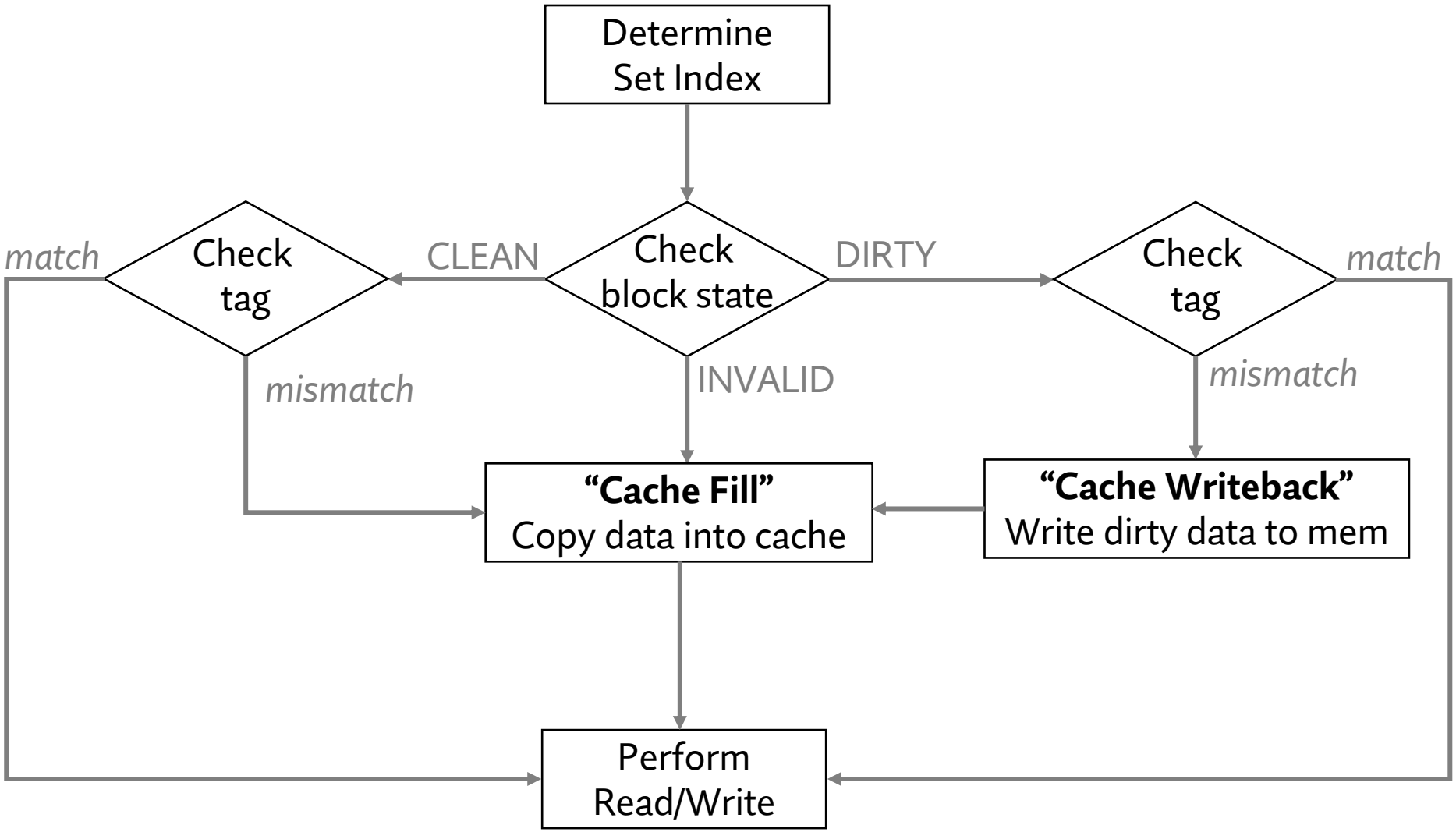
Querying the Cache

- The cache must **track the current state** of every block
  - **INVALID:** no data currently cached (tag is meaningless)
  - **CLEAN:** data is cached and unmodified
  - **DIRTY:** data is cached and modified by the CPU (doesn't match memory)

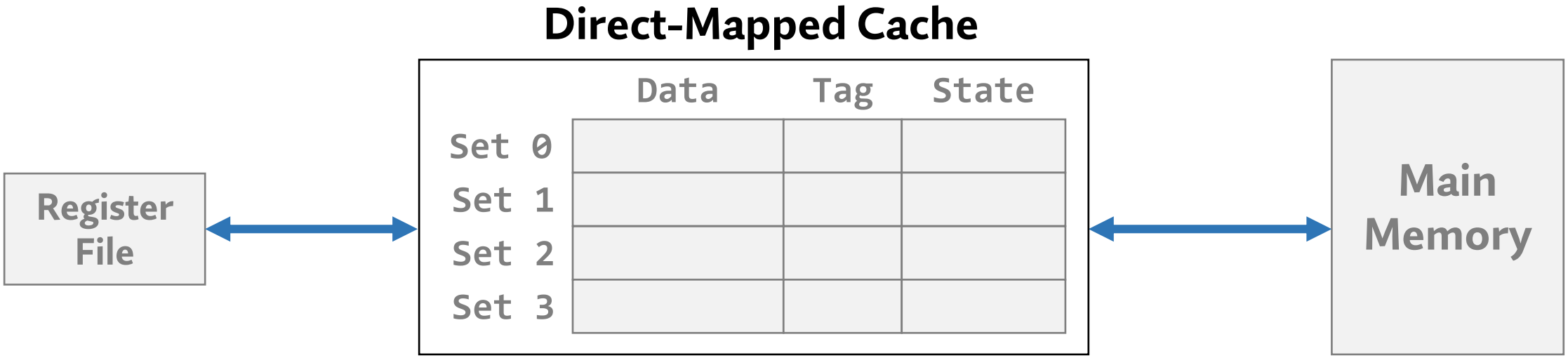
Direct-Mapped Cache



Querying the Cache: Cache Controller

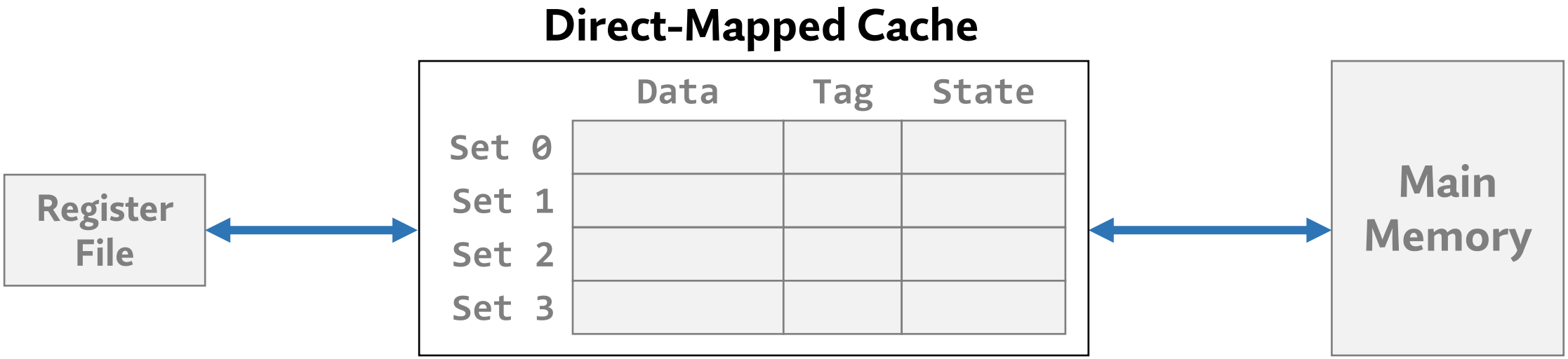


Querying the Cache: STORE Instruction



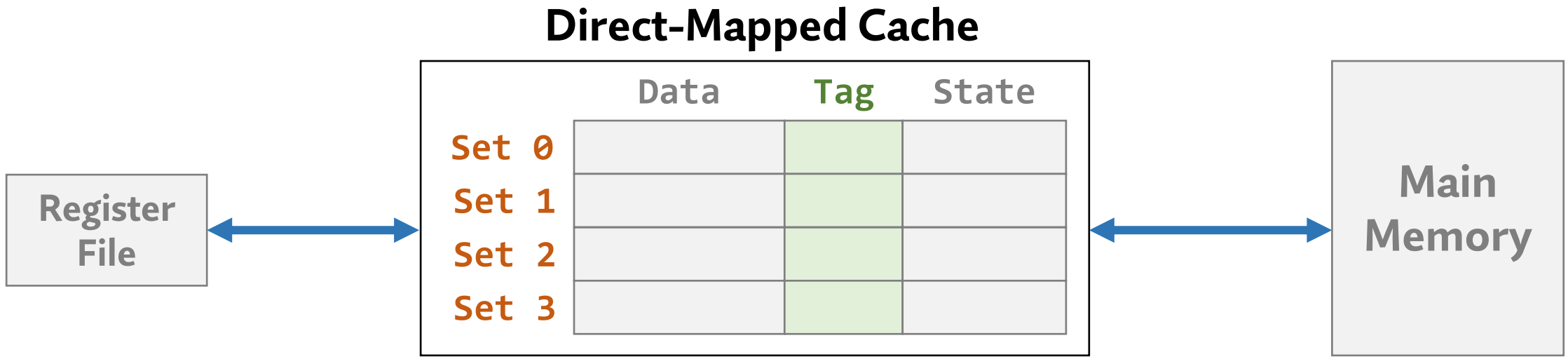

```
func:
    // a0 = 0x100
    // a1 = 0xff
    sd   a1, 0(a0)
```

Example: Accessing an Array




```
void sum(uint64_t a[10])
{
    // a = 0x100
    uint64_t s = 0;
    for(int i = 0; i < 10; i++)
        s += a[i];
    return s;
}
```

Example: Accessing an Array




```
void sum(uint64_t a[10])
{
    // a = 0x100
    uint64_t s = 0;
    for(int i = 0; i < 10; i++)
        s += a[i];
    return s;
}
```

```
a[0] = 1_0000_0000      read(a[0]) = MISS
a[1] = 1_0000_1000      read(a[1]) = HIT
a[2] = 1_0001_0000      read(a[2]) = HIT
a[3] = 1_0001_1000      read(a[3]) = HIT
a[4] = 1_0010_0000      read(a[4]) = HIT
a[5] = 1_0010_1000      read(a[5]) = HIT
a[6] = 1_0011_0000      read(a[6]) = HIT
a[7] = 1_0011_1000      read(a[7]) = HIT
a[8] = 1_0100_0000      read(a[8]) = MISS
a[9] = 1_0100_1000      read(a[9]) = HIT
```

CS 211: Intro to Computer Architecture

13.2: Memory Access Patterns and Caching

Minesh Patel

Spring 2025 – Thursday 24 April