

Lecture 6. CPU Microarchitecture

Introductions to Data Systems and Data Design

Ce Zhang

CPU Microarchitecture Overview

What is a CPU?

At its core, a CPU does two things:

1. **Process instructions** (compute)
2. **Move data** (memory access)

What is a CPU?

At its core, a CPU does two things:

1. **Process instructions** (compute)
2. **Move data** (memory access)

Key insight: Modern CPUs are very good at (1), but fundamentally limited by (2).

Understanding *why* will help you design better data systems.

Skylake Server Microarchitecture

Front End

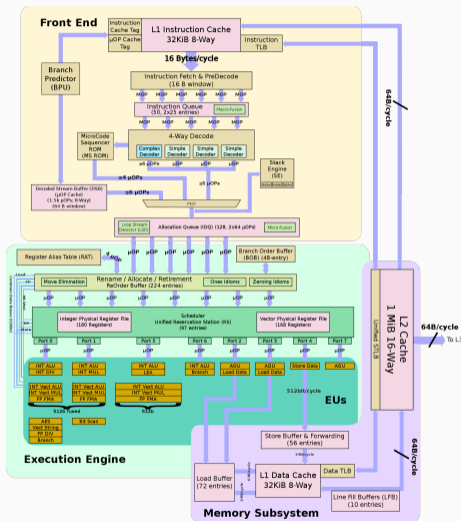
- Fetch instructions at 16 B/cycle
- Decode them into uOps

Execution Engine

- 8 ports, 4 ALUs

Memory Subsystem

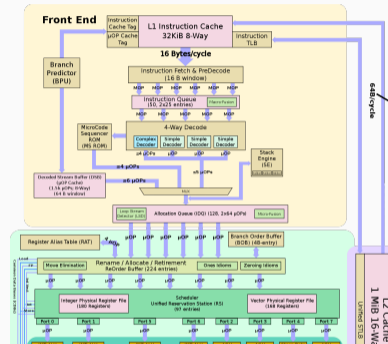
- L1D: 32 KB, 8-way, **4 cycles**
- L2: 1 MB, 16-way, **14 cycles**
- Load buffer: 72 entries
- Store buffer: 56 entries
- 64 B cache line, 64 B/cycle bandwidth



The Front End (Brief Overview)

What it does: Fetches and decodes x86 instructions into μ OPs

- **L1I cache** (32 KB, 8-way): Stores instruction bytes. 8-way = each address can map to 8 cache slots (reduces conflicts).
- **Fetch** (16 B/cycle): Reads from L1I using program counter (PC) \rightarrow outputs raw instruction bytes.
- **Decode** (4 instr/cycle): Translates x86 bytes \rightarrow μ OPs. (1 complex + 3 simple decoders)
- **μ OP cache** (6 μ OPs/cycle): Caches decoded μ OPs for hot loops — skips decode entirely.

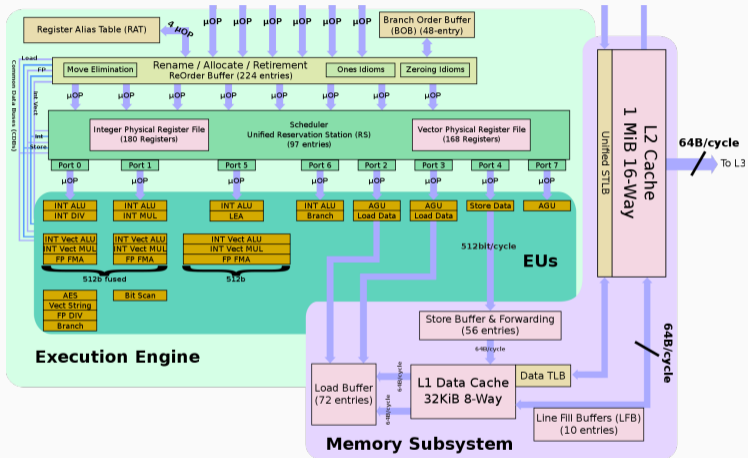


When does the front end stall?

- **I-cache miss:** Code doesn't fit in 32 KB L1I → wait ~14 cycles (L2) or ~100+ cycles (RAM)
 - *Example:* Large binaries, heavy inlining, many cold function calls
- **Branch mispredict:** CPU guesses branch direction *before* condition is evaluated (to keep pipeline full). Wrong guess → flush all speculative work, restart.
 - *Predictable:* Loop branches (for $i < n$), always-taken/never-taken → >99% accuracy
 - *Unpredictable:* Data-dependent branches like `if (array[i] > threshold)` on random data → ~50% accuracy
 - ~15–20 cycle penalty per mispredict

I-cache is real! We will not be focusing on the front-end in this course, but they will show up (often unexpectedly).

Our Focus: The Execution Engine & Memory



Execution Engine (yellow) + Memory Subsystem (green) + L2 Cache (right)

What it does: Holds decoded μ OPs waiting for operands, dispatches to execution ports when ready.

- **97 entries** — can hold ~ 97 in-flight μ OPs
- **Out-of-order execution:** Instructions don't execute in program order; they execute when *operands are ready*
- **Dependency tracking:** Tracks which μ OPs depend on which results

Key insight: The scheduler hides latency by doing *other work* while waiting for slow operations (e.g., memory loads).

Registers and Register Renaming

Architectural registers: What the programmer sees

- **16 general-purpose** (64-bit each): rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp, r8–r15
 - Can access as 32-bit (eax), 16-bit (ax), or 8-bit (al, ah)
 - Total: $16 \times 8 \text{ bytes} = 128 \text{ bytes}$
- **16/32 vector registers** (AVX-256: 256-bit each, AVX-512: 512-bit): ymm0–ymm15 or zmm0–zmm31
 - Used for SIMD: process 4 doubles or 8 floats in one instruction
 - Total: $16 \times 32 \text{ bytes} = 512 \text{ bytes}$ (AVX-256)

Physical registers: What the hardware actually has (Skylake: 180 integer, 168 vector). We will not go into details, do ask ChatGPT if you are curious!

Register access: ~ 0 cycles latency (same cycle), “unlimited bandwidth”

Execution Ports (Skylake)

How it works:

1. Scheduler checks which μ OPs have all operands ready
2. Each cycle, scheduler dispatches ready μ OPs to available ports
3. Each port has specific execution units — μ OP goes to a port that can handle it
4. Execution takes 1+ cycles depending on operation (add: 1 cycle, divide: 10+ cycles)
5. Result is written back, waking up dependent μ OPs

Key constraint: Each port can only execute **one μ OP per cycle**

Port	What it does	Latency
P0, P1	ALU, FMA (integer/FP math)	1 / 4 cycles
P0 only	Division	10-20 cycles
P5	ALU, vector shuffle	1 cycle
P6	ALU, branches	1 cycle
P2, P3	Load (2 loads/cycle)	4-5 cycles (L1)
P4, P7	Store data + address	~4 cycles

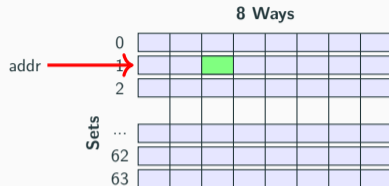
L1 Data Cache (32 KB, 8-Way Set Associative)

32 KB = 64 sets \times 8 ways \times 64 B

- **Cache line:** 64 bytes — minimum transfer (even for 1 byte read)
- **Latency:** 4-5 cycles

Mental model for this course:

- 32 KB of fast memory
- 64 B cache line (minimum unit)
- **~LRU eviction:** Least recently used data gets evicted when full
 - It is not real LRU for two reasons: (1) 8-way set associative and (2) not exact LRU implementation.
 - In this course we will ignore these details and only think about L1 as one big chunk of cache.



L2 Cache (1 MB, 16-Way Set Associative)

1 MB per core, private

- **Cache line:** 64 bytes
- **Latency:** 14 cycles ($3.5\times$ slower than L1)

Mental model for this course:

- 1 MB of medium-speed memory per core
- ~**LRU eviction**
- **Unified:** Holds both instructions and data
- Miss L1 \rightarrow check L2 \rightarrow 14 cycle penalty

L3 Cache (Last Level Cache)

Size: 1.375 MB per core (shared across all cores)

Latency: 50–70 cycles

Bandwidth: Lower than L2, shared among cores

Key properties:

- **Shared:** All cores access the same L3
- **Coherent:** Hardware keeps data consistent across cores

When you miss L3: DRAM access — 100+ cycles, ~100 ns.

Putting All Numbers Together

Operation	Latency	Relative
Register access	~0.3 ns	1x
L1 cache hit	~1 ns	3x
L2 cache hit	~4 ns	13x
L3 cache hit	~12 ns	40x
Main memory	~100 ns	333x
SSD random read	~16,000 ns	53,000x
HDD random read	~2,000,000 ns	6.6M x

Memory access is 100–300× slower than computation! This is *the* fundamental theme that has governed data systems design and optimization for decades.

Life Cycle of an Instruction

What Happens When You Run Code?

Every instruction goes through a journey:



What Happens When You Run Code?

Every instruction goes through a journey:



A Simple C++ Program

```
// add.cpp
int main() {
    long a = 5;
    long b = 3;
    long c = a + b;
    return c;
}
```

\$ cd examples && make add # Compile without optimization
\$./add && echo \$? # Execute, print return value

\$ make asm # Generate assembly files
\$ cat add.s # View assembly
\$ objdump -d add | less # Or disassemble binary

Try it! Use godbolt.org to see assembly instantly in browser.

The Assembly (Unoptimized)

main:

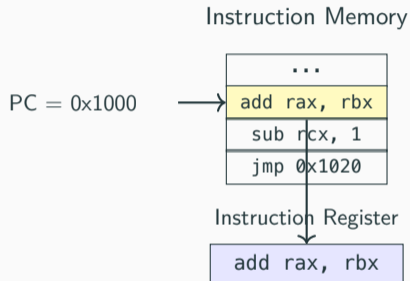
```
    push rbp                ; Save caller's base pointer
    mov rbp, rsp            ; Set up our stack frame

    mov QWORD PTR [rbp-8], 5 ; Store 5 on stack (variable a)
    mov QWORD PTR [rbp-16], 3 ; Store 3 on stack (variable b)

    mov rdx, QWORD PTR [rbp-8] ; Load a (5) into rdx
    mov rax, QWORD PTR [rbp-16] ; Load b (3) into rax
    add rax, rdx               ; rax = rax + rdx = 3 + 5 = 8

    mov QWORD PTR [rbp-24], rax ; Store result on stack (variable c)
    mov rax, QWORD PTR [rbp-24] ; Load c into rax (return value)
```

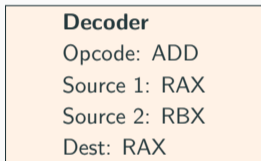
Stage 1: Fetch



- Read instruction bytes from memory (via I-cache)

Stage 2: Decode

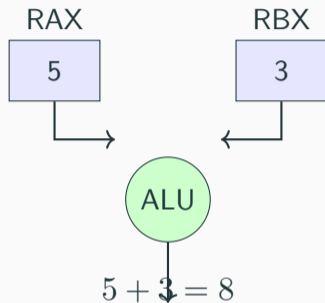
"add rax, rbx" (bytes: 48 01 D8)



μops to execution

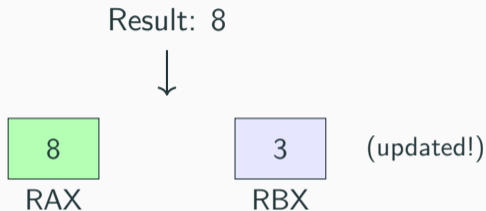
Modern CPUs decode complex instructions into simpler micro-operations.

Stage 3: Execute



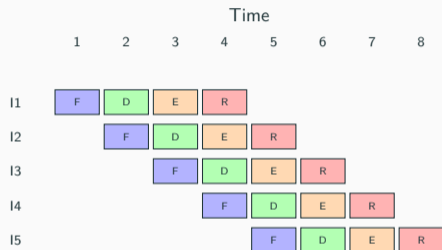
- Read operands from registers
- Perform the computation
- ALU operations are **fast**: 1 cycle for this case

Stage 4: Retire (Write Back)



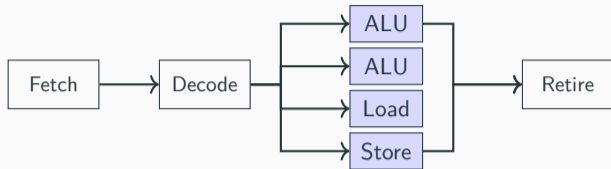
- Write result back to destination register
- Update architectural state
- Instruction is now “complete”

Pipelining: Overlapping Instructions



- After pipeline fills: 1 instruction completes per cycle
- Modern CPUs: 14-20 stages, 4-6 instructions/cycle

Superscalar: Instruction-Level Parallelism



Multiple instructions execute **in parallel** each cycle.

All modern CPUs are superscalar. Intel, AMD, Apple Silicon, ARM — they all execute multiple instructions per cycle.

Mental model for this course: Think of the CPU as being able to execute $\sim K$ independent instructions per cycle. If instruction B depends on the result of instruction A, they must run sequentially. If they are independent, they can run in parallel, as long as there are available *ports that can run the operation*.

Life Cycle of Data

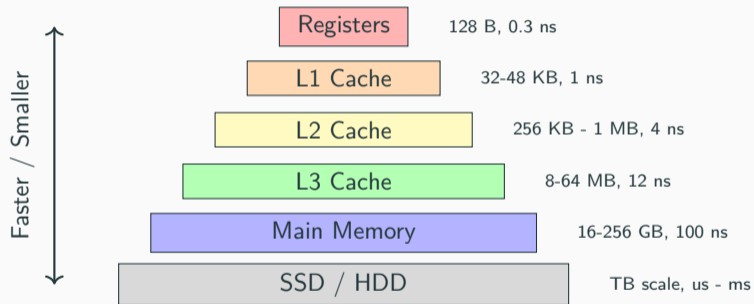
Where Does Data Live?

Data has its own journey through the memory hierarchy:



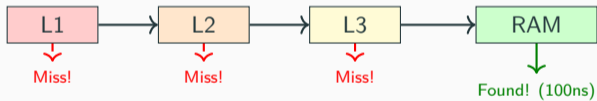
Each level is: **Smaller, Faster, More expensive**

The Memory Hierarchy



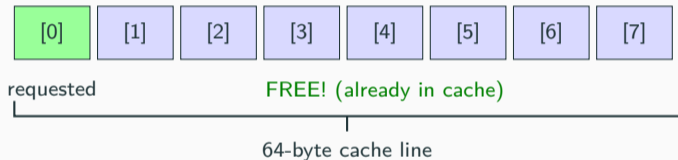
What Happens on a Memory Access?

Consider: `mov rdx, QWORD PTR [rbp-8]`



Cache Lines: The Unit of Transfer

You request array[0] (8 bytes), memory loads 64 bytes:

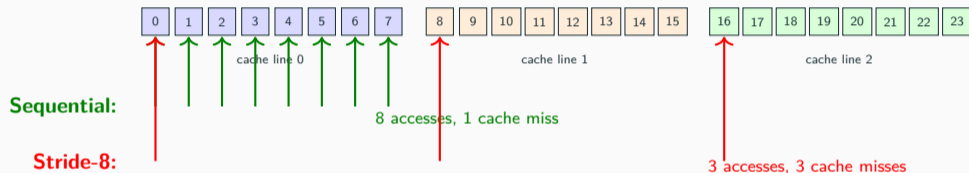


If you access array[1] next: **already in cache!**

Spatial locality: Sequential access is “free”.

Why Locality Matters

Memory: 24 elements across 3 cache lines



Sequential: All 8 accesses hit the same cache line → 1 miss total

Stride-8: Each access hits a different cache line → 1 miss per access (8× worse!)

Two Types of Locality

Temporal Locality:

- Data accessed recently will likely be accessed again
- Example: loop counter, frequently-used variables
- Keep hot data in cache

Spatial Locality:

- Data near recently accessed data will likely be accessed
- Example: array traversal, struct fields
- Access data sequentially when possible

Data systems design is all about locality. Databases, ML systems, and every high-performance system we study in this course optimizes for these two properties. We will see this pattern again and again.

What Can Go Wrong?

Let's see locality in action with a simple experiment.

```
int matrix[1000][1000];
```

```
long sum = 0;
```

```
for (int i = 0; i < 1000; i++)
```

```
    for (int j = 0; j < 1000; j++)
```

```
        sum += matrix[i][j];
```

```
$ cd examples && make matrix_row && ./matrix_row
```

```
Sum: ..., Time: 478 ms
```

Try this one

```
int matrix[1000][1000];  
long sum = 0;  
  
for (int j = 0; j < 1000; j++)  
    for (int i = 0; i < 1000; i++)  
        sum += matrix[i][j];  
  
$ cd examples && make matrix_col && ./matrix_col  
Sum: ..., Time: 623 ms
```

Why Slower?

Row-major traversal

```
for (i = 0; i < 1000; i++)  
    for (j = 0; j < 1000; j++)  
        sum += matrix[i][j];
```

Access: [0][0], [0][1], [0][2], ...

Linearized: **0, 1, 2, 3, 4, ...**

Stride = 4 bytes

Column-major traversal

```
for (j = 0; j < 1000; j++)  
    for (i = 0; i < 1000; i++)  
        sum += matrix[i][j];
```

Access: [0][0], [1][0], [2][0], ...

Linearized: **0, 1000, 2000, 3000, ...**

Stride = 4000 bytes

Why 7× Slower? (Cache View)

Row-major: stride = 4 bytes

Cache line = 64 bytes = 16 ints

Access 0, 1, 2, ..., 15 → **1 miss, 15 hits**

Then 16, 17, ..., 31 → **1 miss, 15 hits**

Miss rate (Theoretical): **6.25%**

Column-major: stride = 4000 bytes

Each access jumps 4000 bytes (62 cache lines!)

Access 0 → miss, load line

Access 1000 → miss, load line

Access 2000 → miss, load line

Miss rate (Theoretical): **100%**

Measuring Cache Performance

Performance Counters (PMU)

Real-world applications are far more complex than our matrix example. Reasoning alone isn't enough—we need to **benchmark, profile, and measure** to understand where time is spent.

Modern CPUs have **Performance Monitoring Units** that count hardware events:

- Cache hits/misses at each level (L1, L2, L3)
- Branch predictions/mispredictions
- Instructions retired, cycles, etc.

On Linux, use `perf` to read these counters:

```
$ cd examples && make perf      # Run all with perf profiling
```

Measuring Row-Major Access

```
$ cd examples
```

```
$ perf stat -e L1-dcache-loads,L1-dcache-load-misses ./matrix_row
```

```
Performance counter stats for './matrix_row':
```

```
6,028,299,510      L1-dcache-loads
```

```
62,665,470        L1-dcache-load-misses      #      1.04% of all L1-
```

```
dcache accesses
```

```
(1000 iterations × 62,500 = 62.5M misses)
```

Measuring Column-Major Access

```
$ cd examples
```

```
$ perf stat -e L1-dcache-loads,L1-dcache-load-misses ./matrix_col
```

```
6,021,517,439      L1-dcache-loads
```

```
1,012,648,771      L1-dcache-load-misses      #    16.82% of all L1-
```

```
dcache accesses
```

A lot more L-1 cache misses.

Cache Miss Summary

Traversal	L1 Loads	L1 Misses	Miss Rate
Row-major	6B	62M	1.04%
Column-major	6B	1B	16.82%

16× more L1 cache misses → 1.5× slower

Takeaway: perf lets you verify your mental model. When optimizing, measure first!

\$ cd examples && make perf *# Try it yourself!*

Hazards: When Things Go Wrong

The pipeline assumes instructions flow smoothly. Three things break this:

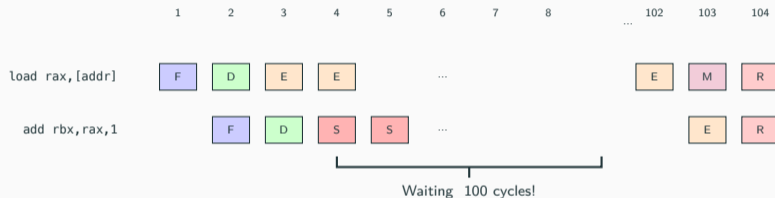
1. **Data hazards:** Instruction needs result from previous instruction
2. **Control hazards:** Branch—which instruction comes next?
3. **Structural hazards:** Two instructions need same hardware

The pipeline assumes instructions flow smoothly. Three things break this:

1. **Data hazards:** Instruction needs result from previous instruction
2. **Control hazards:** Branch—which instruction comes next?
3. **Structural hazards:** Two instructions need same hardware

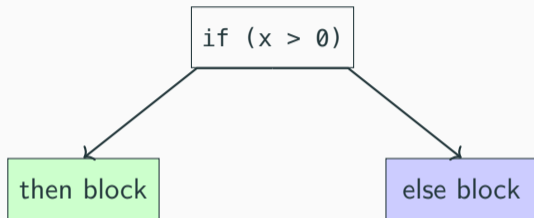
All of these are fundamentally about waiting for data.

Data Hazard Example



The add is **starved**—cannot execute until data arrives.

Control Hazard: Branches

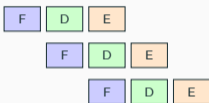


Which path to fetch???

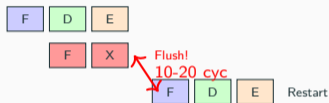
Pipeline must guess **before** condition is known!

Branch Misprediction Cost

Correct:



Wrong:



Modern predictors are $>95\%$ accurate, **but** data-dependent branches can be unpredictable.

Which is Fastest? (Random Data)

Version A

```
for (i = 0; i < n; i++)  
    if (data[i] > t)  
        count++;
```

Version B

```
for (i = 0; i < n; i++)  
    count += (data[i] > t);
```

Version C

```
sort(data, data + n);  
for (i = 0; i < n; i++)  
    if (data[i] > t)  
        count++;
```

```
$ cd examples && make && ./branch_random && ./branch_branchless && ./branch_sort
```

Version	Time
A (branching)	298 ms
B (branchless)	42 ms
C (sorted)	9235 ms
C (pre-sorted)	43ms

Branchless is **7× faster** than branching on random data!

Data-Dependent Branches: The Problem

C++ code

```
for (int i = 0; i < n; i++) {  
    if (data[i] > threshold)  
        count++;  
}
```

On random data: branch taken ~50% of the time, **unpredictably**.

Branch predictor accuracy: ~50%

~15-20 cycle penalty per mispredict!

Assembly (x86-64)

```
loop:  
    mov    eax, [rdi + rcx*4]  
    cmp    eax, esi  
    jle    skip      ; BRANCH!  
    inc    edx  
skip:  
    inc    rcx  
    cmp    rcx, r8  
    jl     loop
```

The `jle skip` is the problem—CPU must guess which way to go.

Solution 1: Branchless Code

C++ code

```
for (int i = 0; i < n; i++) {  
    count += (data[i] > threshold);  
}
```

The comparison (data[i] > threshold) evaluates to 0 or 1.

No branch = no misprediction penalty!

Assembly (x86-64)

```
loop:  
    mov    eax, [rdi + rcx*4]  
    cmp    eax, esi  
    setg   al                ; al = 1 if greater  
    movzx  eax, al  
    add    edx, eax          ; No branch!  
    inc    rcx  
    cmp    rcx, r8  
    jl     loop
```

setg + add replaces the conditional branch.

Solution 2: Sort First

C++ code

```
sort(data, data + n);  
  
for (int i = 0; i < n; i++) {  
    if (data[i] > threshold)  
        count++;  
}
```

After sorting: all values below threshold
come first, then all above.

Branch pattern: FFFFF...TTTTT

Predictor accuracy: >99%

Why it works

Sorted data:

[1, 3, 5, 8, 12, 15, 20, ...]

^ threshold = 10

First iterations: always skip (predict: skip)

Later iterations: always take (predict: take)

Only ~1 misprediction at the transition!

Branch Performance Comparison

```
$ cd examples && perf stat -e branches,branch-misses ./branch_random
$ cd examples && perf stat -e branches,branch-misses ./branch_branchless
$ cd examples && perf stat -e branches,branch-misses ./branch_sorted
$ cd examples && perf stat -e branches,branch-misses ./branch_presorted
```

Approach	Time (random data)	Why
Branching	~298 ms	A lot of misprediction
Branchless	~42 ms	No branches to mispredict
Branching over pre-sorted	~43 ms	Predictable pattern

Note: Sorting has $O(n \log n)$ cost, only worth it if you traverse multiple times or need sorted data anyway.

The Fundamental Problem

Every hazard is a data movement problem

- **Data hazard:** *Waiting for data from memory*
- **Control hazard:** *Waiting for branch condition (data!)*
- **Structural hazard:** *Waiting for hardware to move data*

The CPU can execute billions of operations per second...

...but only if the data is **in the right place at the right time.**

Strategies to Minimize Data Movement

1. Improve locality

- Sequential access patterns
- Keep working set small (fit in cache)
- Co-locate related data

2. Hide latency

- Prefetching (hardware and software)
- Out-of-order execution (do other work while waiting)
- Parallelism (multiple outstanding requests)

3. Reduce data volume

- Compression
- Filtering early (don't move data you'll discard)

Summary

Key Takeaways

1. **Instructions** flow through: Fetch → Decode → Execute → Retire
 - Pipelining and superscalar execution enable high throughput
 - But only if data is ready!
2. **Data** flows through: Disk → RAM → L3 → L2 → L1 → Registers
 - Each level faster but smaller
 - Cache lines (64 bytes) are the unit of transfer
3. **Hazards** occur when data isn't where it needs to be
 - Data hazards, control hazards—all about waiting for data
4. **Optimization is fundamentally about data movement**