

Exercise walk-through

3.3

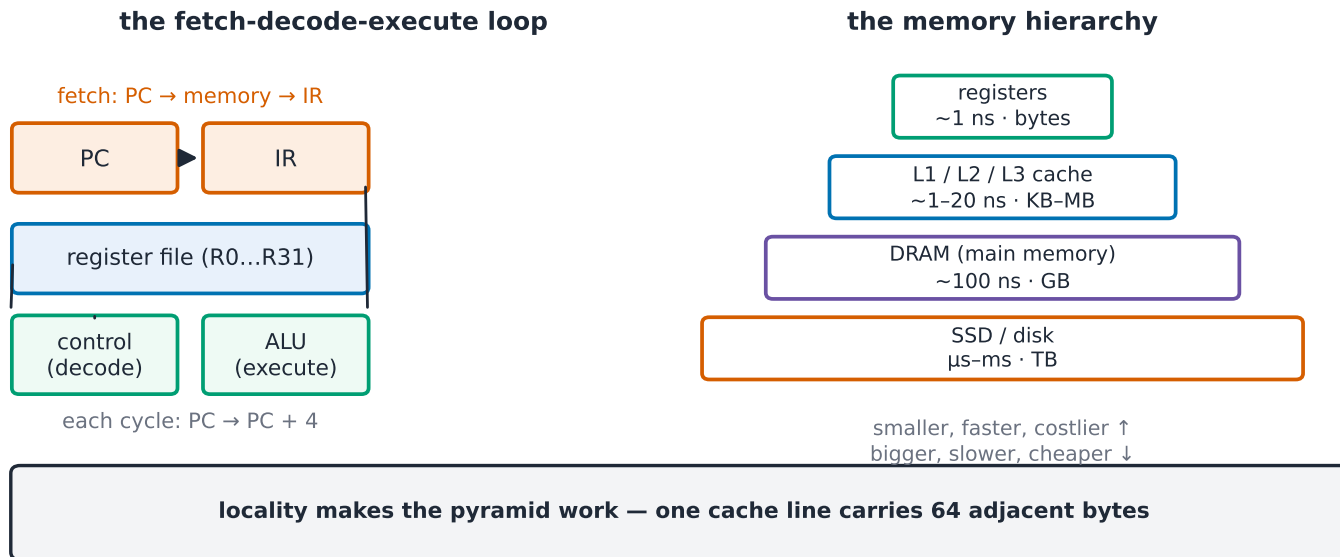
Processor and memory overview

University Computer Science

Processor and memory: the loop and the pyramid

- 1 **Loop** Fetch ($PC \rightarrow \text{memory} \rightarrow IR$), decode (control signals), execute (ALU/memory), $PC \rightarrow PC + 4$.
- 2 **Pyramid** Registers $\sim 1 \text{ ns}$ $\rightarrow$ L1/L2/L3 $\rightarrow$ DRAM $\sim 100 \text{ ns}$ $\rightarrow$ SSD: fast memory is expensive per byte, big memory is slow.
- 3 **Layout** One **cache** line carries 64 adjacent bytes — row-major access beats transposed loops at the same op count.

CPU and memory block diagram



The fetch-decode-execute loop around the register file, with the memory pyramid alongside.

Exercises 2.1-2.3

2.1 Which register receives the fetched instruction? [3]

2.2 By how much does the PC advance per instruction (RISC)? [3]

2.3 Name the two locality kinds. [3]

Exercises 2.1-2.3

2.1 Which register receives the fetched instruction? [3]

2.1

- The IR (instruction register) – PC supplies the address ✓.
-

2.2 By how much does the PC advance per instruction (RISC)? [3]

2.2

- 4 (one word) ✓.
-

2.3 Name the two locality kinds. [3]

2.3

- Temporal (recently used again soon) and spatial (neighbours used soon) ✓.

Exercise 3.1

3.1 Trace `LOAD R1, 0(R2)`, `ADD R3, R1, R1`, `STORE R3, 0(R2)` through fetch-decode-execute.

For each instruction, state the register (or memory) that changes and its new content, given `R2` holds 100 and `mem[100] = 7`. Give the PC values before and after each of the three instructions. [8]

Exercise 3.1

3.1 Trace `LOAD R1, 0(R2), ADD R3, R1, R1, STORE R3, 0(R2)` through fetch-decode-execute.

For each instruction, state the register (or memory) that changes and its new content, given `R2` holds 100 and `mem[100] = 7`. Give the PC values before and after each of the three instructions. [8]

3.1

- `LOAD`: $R1 \leftarrow \text{mem}[100] = 7$; $\text{PC } 0 \rightarrow 4$ ✓.
- `ADD`: $R3 \leftarrow 7 + 7 = 14$; $\text{PC } 4 \rightarrow 8$ ✓.
- `STORE`: $\text{mem}[100] \leftarrow 14$; $\text{PC } 8 \rightarrow 12$ ✓.
- Only these change: the loop names its own deltas ✓.

Exercise 3.2

3.2 A program sums a 4096×4096 integer array twice: once row-major, once column-major; the machine has a 64-byte cache line.

Explain why the row-major pass loads roughly one cache line per 16 elements while the column-major pass loads one line per element, and estimate the miss-ratio contrast. State which of temporal or spatial locality each pass exploits. Give one code change that repairs the column-major pass, and one hardware reason why a bigger cache alone cannot fully hide the difference.

[7]

Exercise 3.2

3.2 A program sums a 4096×4096 integer array twice: once row-major, once column-major; the machine has a 64-byte cache line.

Explain why the row-major pass loads roughly one cache line per 16 elements while the column-major pass loads one line per element, and estimate the miss-ratio contrast. State which of temporal or spatial locality each pass exploits. Give one code change that repairs the column-major pass, and one hardware reason why a bigger cache alone cannot fully hide the difference. [7]

3.2

- 64-byte line, 4-byte ints: 16 consecutive elements per line – row-major reuses each line $16\times$, column-major strides 16 KB and misses nearly every element ✓.
- Row-major exploits spatial locality; the reuse-of-line is spatial, not temporal ✓.
- Loop-interchange (reorder to keep the fast index inner) fixes the program; a bigger cache cannot hold the working set of a strided sweep – capacity misses persist ✓.