CS 105 · LESSON 25

Pipelining, Instruction-Level Parallelism, and Loop Optimization

How modern processors squeeze parallelism out of straight-line code — and how to write code that lets them.

Adapted from Bryant & O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3e

COURSE CONTEXT

Where we are: making real code fast

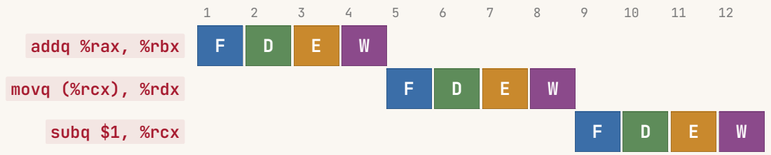
- You can already **read** x86-64 and reason about what it does, instruction by instruction.
- Today: the CPU does **not** run those instructions one at a time. Understanding why changes how you write hot loops.
- Goal — by the end of class you can look at a piece of code and say *roughly* how fast it will run, and why.

BY END OF CLASS

01 Explain pipelining and why it helps
02 Identify a data dependency that causes a stall
03 Reorder code (or unroll a loop) to avoid one
04 Judge whether a transformation is likely to help

One instruction at a time

A simple model of execution: do an instruction completely — fetch it, figure out what it is, execute it, write the result — *then* start the next one.



F Fetch D Decode E Execute W Writeback

3 instructions × 4 cycles each = **12 cycles.**
Throughput: **0.25 instr/cycle.**

Inside one instruction: four stages

Each instruction passes through different parts of the chip. While one stage works, the others sit idle.

F
Fetch
Read instruction bytes from memory at %rip.

D
Decode
Figure out what op it is; read source registers.

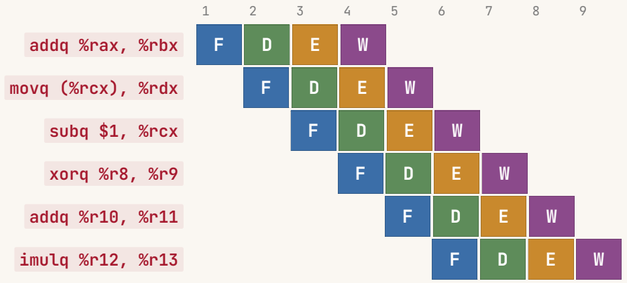
E
Execute
ALU work, address calc, branch comparison.

W
Writeback
Store result back to register or memory.

The trick: while instruction *i* is in *Execute*, instruction *i+1* could be in *Decode*, and instruction *i+2* in *Fetch*. Different hardware, different work — **no reason for them to wait their turn.**

Pipelining: overlap the stages

Start the next instruction every cycle. Same work, but multiple instructions in flight at once.



F Fetch D Decode E Execute W Writeback

6 instructions in 9 cycles. Throughput approaches **1 instr/cycle** as the pipeline fills.

Throughput vs. latency

LATENCY

Same as before

One instruction still takes 4 cycles, start to finish. Pipelining doesn't make any individual instruction faster.

THROUGHPUT

~4× better

We finish an instruction *every cycle* instead of every 4 cycles — because four are always in flight.

Programs care about throughput. A loop that runs a million times doesn't care how long any single iteration takes — it cares how many iterations finish per second. Pipelining is the first big lever.

The throughput formula

```
cycles = N + (k - 1)
```

for N instructions through a k-stage pipeline (no stalls)

N
instructions to run
the work itself

k
stages in the pipeline
depth

k-1
fill cost
startup, paid once

For large N, the *k−1* term washes out and throughput approaches **1 instruction per cycle.**

EXERCISE · 1

Predict the speedup

You run **1,000 independent** instructions on a CPU with a **4-stage** pipeline.

```
cycles = N + (k - 1)
```

COMPUTE

1. Sequential cycles?
2. Pipelined cycles?
3. Speedup (sequential ÷ pipelined)?
4. What if N = 4 instead?

Discuss with your neighbor (3 min). Don't worry about hazards yet — assume every instruction is independent.

Real pipelines are much deeper

- Our 4-stage pipeline is a teaching model. Real x86-64 cores have **14–20+ stages**.
- Deeper pipelines run at higher clock speeds — each stage does less work, so the cycle can be shorter.
- They also issue **multiple instructions per cycle** (superscalar) and **reorder them** (out-of-order) to keep stages busy.
- You don’t need to know the names of every stage. You need to understand **what makes the pipeline stall.**

APPROXIMATE PIPELINE DEPTH

Intel Pentium 4 (Prescott)	31
AMD Zen 4	~19
Intel Golden Cove	~17
Apple M-series (perf core)	~16
ARM Cortex-A78	13
Our toy CPU (this lecture)	4

Approximate; modern OOO cores are hard to count.

PART 2

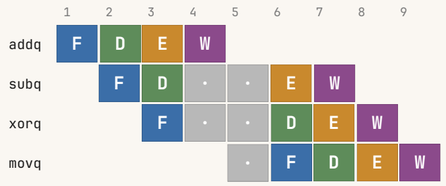
Stalls: when the pipeline can’t move.

The pipeline only hits 1 IPC if every instruction is ready when its stage opens. Let’s look at what stops that.

Data hazard: read-after-write

Instruction 2 needs the result of Instruction 1. It can’t Execute until Instruction 1 has Written Back.

```
addq  %rax, %rbx     # rbx = rax + rbx
subq  %rbx, %rcx     # needs rbx!
xorq  %r8,  %r9
movq  %r10, %r11
```



F Fetch D Decode E Execute W Writeback · Stall

Two · cells = a two-cycle **stall**. The whole pipeline waits.

Forwarding helps — sometimes

CPUs route a result *directly* from Execute back to the next instruction’s Execute — without waiting for Writeback.

- Most ALU-to-ALU dependencies cost zero stall cycles.
-
- Long-latency ops (mul, div, FP add) take many cycles in Execute.

APPROXIMATE SKYLAKE NUMBERS

`add / sub / and / or`	1 cycle
`imul (64-bit)`	3 cycles
`load (L1 hit)`	4 cycles
`load (L2 hit)`	12 cycles
`load (DRAM)`	~200 cycles
`div (64-bit)`	~25 cycles
`fp add / mul`	4 cycles

> A long latency only hurts if the *next* instruction depends on the result. That’s the whole game.

EXERCISE · 2

Find — and break — the stall

Assume `imulq` takes 3 cycles in Execute, others take 1.

```
movq   (%rdi), %rax     # load
imulq  $7,     %rax     # rax *= 7
addq   %rax,   %rbx     # use rax
movq   (%rsi), %rcx     # load
imulq  $3,     %rcx
addq   %rcx,   %rdx
```

YOUR JOB

1. Mark every read-after-write dependency.
2. Where does the pipeline stall?
3. **Reorder the six instructions** so back-to-back dependencies are spaced apart. Don't change what the program computes.

5 minutes, in pairs. Hint: the two halves of the program don't depend on each other.

Control hazard: branches

After a conditional jump, the CPU doesn't yet know *which* instruction to fetch next. But Fetch can't wait — it has to feed the pipeline every cycle.

```
    cmpq  %rax, %rbx
    jge   skip       # branch?
    addq  %rcx, %rdx
    movq  %rdx, (%rsi)
skip:
    xorq  %r8, %r9
```

THE CPU HAS TWO CHOICES

1. **Stall** Fetch until the branch resolves — guaranteed correct, costs ~k−1 cycles every branch.
2. **Predict** the direction, fetch speculatively — free if right, expensive if wrong.

Real CPUs **predict.** A branch happens roughly every 5 instructions — stalling on each one would erase most of pipelining's win.

Branch prediction (the one-slide version)

- **Idea:** guess the branch direction, fetch ahead. On misprediction, throw out the speculative work and restart.
- Modern predictors are **~95–99% accurate** on real code.
- A misprediction costs roughly the full pipeline depth — **15–20 cycles** on a modern CPU.
- Predictable branches (loops, error checks) are nearly free. Data-dependent branches (sorting, parsing) are where it hurts.

QUICK EXAMPLE

```
// loop runs 1,000,000 times
for (i = 0; i < n; i++) ...
```

The backward branch is taken 999,999 times and not taken once. The predictor learns this almost immediately. **Effectively zero cost.**

PART 3

Independent work fills the pipeline.

Stalls come from dependencies. The cure is giving the CPU more work it can do in parallel.

Unrolling, step 1: cut the overhead

ORIGINAL LOOP

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

Per iteration: 1 add of payload, plus a compare, branch, and increment. **~3 of 4 instructions are bookkeeping.**

UNROLLED ×4

```
for (i = 0; i < n; i += 4) {
    sum += a[i];
    sum += a[i+1];
    sum += a[i+2];
    sum += a[i+3];
}
// + cleanup loop for remainder
```

Same work, ¼ the loop overhead. Helps a little — but the *real* win is what comes next.

Note that all four `sum +=` lines still touch the same `sum`. Each one waits for the previous one. **Single dependency chain.**

Unrolling, step 2: multiple accumulators

ONE ACCUMULATOR

```
sum += a[i];
sum += a[i+1];   // waits on sum
sum += a[i+2];   // waits on sum
sum += a[i+3];   // waits on sum
```

Four adds, but they form one chain. Latency-bound: 4× the FP-add latency.

FOUR ACCUMULATORS

```
s0 += a[i];
s1 += a[i+1];    // independent!
s2 += a[i+2];    // independent!
s3 += a[i+3];    // independent!
// after loop: sum = s0+s1+s2+s3
```

Four **independent** chains. The CPU can run them in parallel on its multiple execution units.

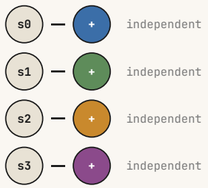
For floating-point: this changes the answer. `(a + b) + c` ≠ `a + (b + c)` in IEEE-754. The compiler won't do this for you under `-O2`; you have to opt in.

Why this works: breaking the chain

ONE CHAIN

Length 5. Time = 5 × (latency of +).

FOUR CHAINS



Length 2 critical path. **~4× faster** on a wide enough core.

The lever: Performance is bounded by the *longest dependency chain* the CPU has to walk. Shorten the chain — or split it into independent ones — and you go faster.

EXERCISE · 3

Will this transformation help?

A

CODE

```
for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
```

PROPOSED CHANGE

```
// unroll x4 with 4 accumulators
```

Helpful, useless, or harmful?

B

CODE

```
for (i=1; i<n; i++)
    a[i] = a[i-1] * 2;
```

PROPOSED CHANGE

```
// unroll x4 with 4 accumulators
```

Helpful, useless, or harmful?

C

CODE

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

PROPOSED CHANGE

```
// reorder to 4 accumulators
```

Helpful, useless, or harmful?

4 minutes, in pairs. Justify your answer in one sentence per case.

What to remember

01 **Pipelining trades latency for throughput.**
Each instruction takes the same time. The CPU just runs k of them at once.

02 **Stalls come from dependencies, not from instruction count.**
Two instructions that read the same register are cheap. One that needs another's result is the expensive case.

03 **Branches are predicted, not stalled.**
Mispredictions cost a full pipeline flush. Predictable branches are free.

04 **The longest dependency chain bounds your speed.**
To go faster: shorten the chain, or split it into independent ones (multiple accumulators).

05 **Loop unrolling matters for the chain, not the bookkeeping.**
Unrolling without breaking the chain gives you a few percent. Unrolling *plus* reassociation gives you the real win.

06 **When you reason about a transformation, ask:**
"Does this break a dependency chain that was actually limiting me?" If yes, ship it. If no, don't bother.

WRAP-UP

The CPU is a parallel machine pretending to run code in order.

Your job is to write code that doesn't spoil the illusion — code with short dependency chains and predictable control flow.

READ FOR NEXT TIME

CS:APP §5.7–5.9 (instruction-level parallelism, loop unrolling, multiple accumulators)

TOOLS TO TRY

`perf stat` for IPC; `llvm-mca` to see the pipeline schedule of your own code.