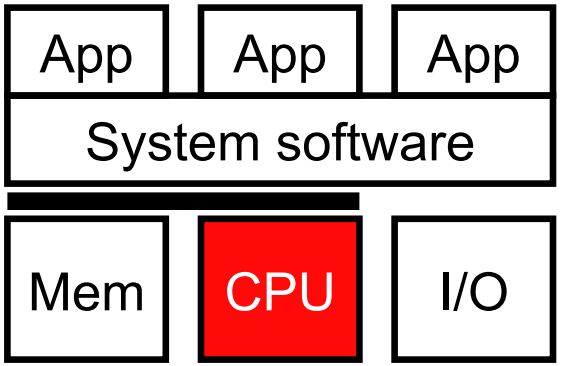
Pipelining

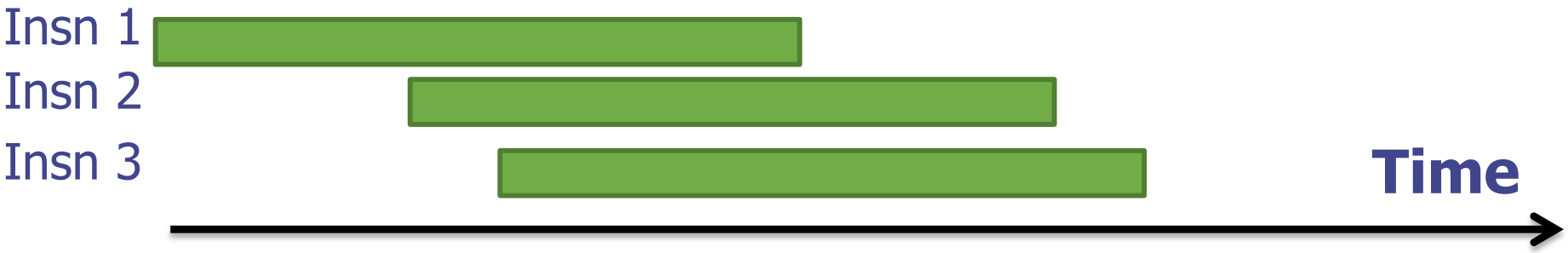
Pipelining



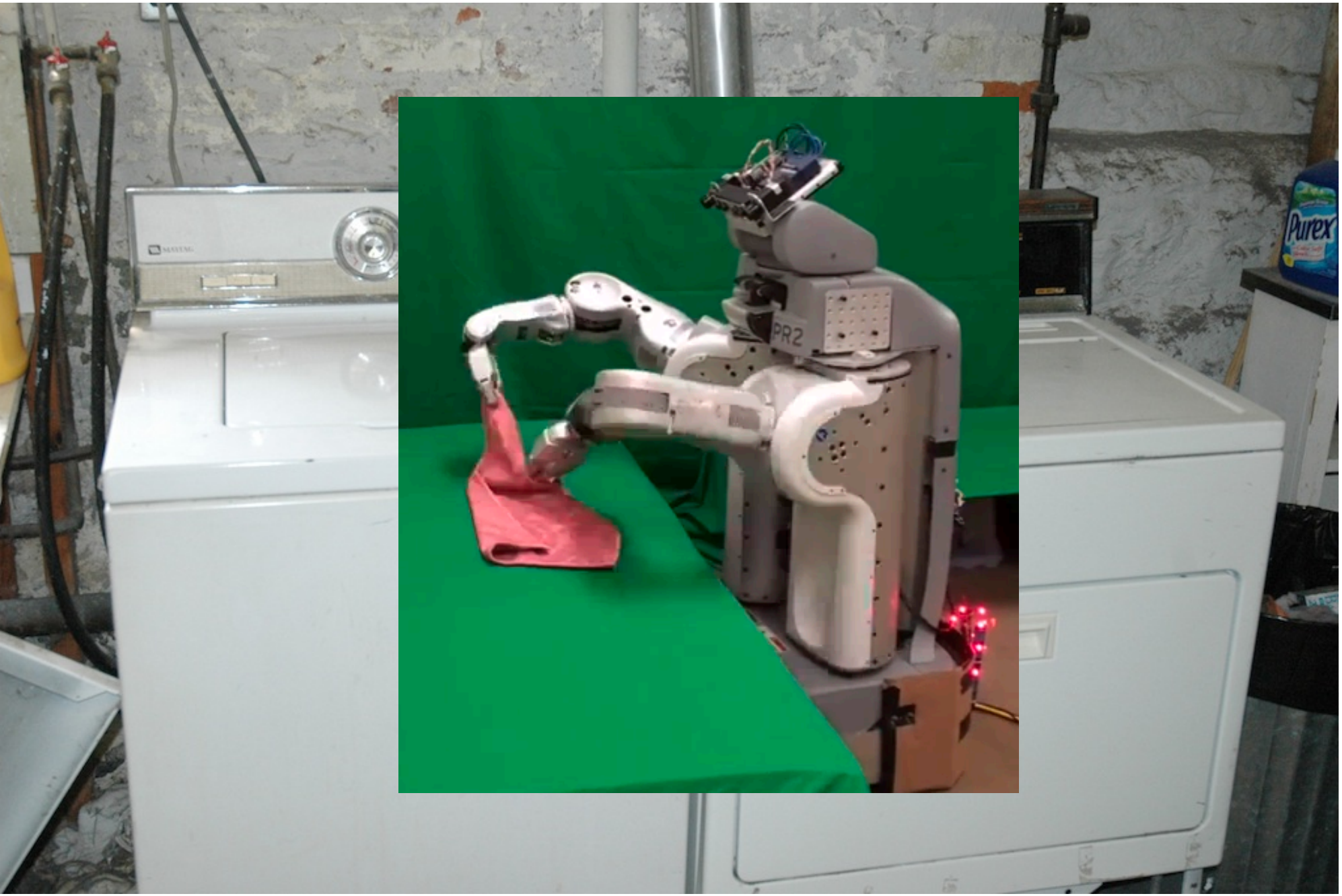
- Single-cycle datapaths
- Latency vs. throughput & performance
- Basic pipelining
- Data hazards
  - Bypassing
  - Load-use stalling
- Pipelined multi-cycle operations
- Control hazards
  - Branch prediction

Main Concept

- Instructions (C++) broken down into finite set of assembly language instructions
- Instructions (start) executing sequentially
- Pipelining method speeds up sequential execution of these instructions
  - The next instruction can get started, while the previous one is still running



The eternal pipelining metaphor

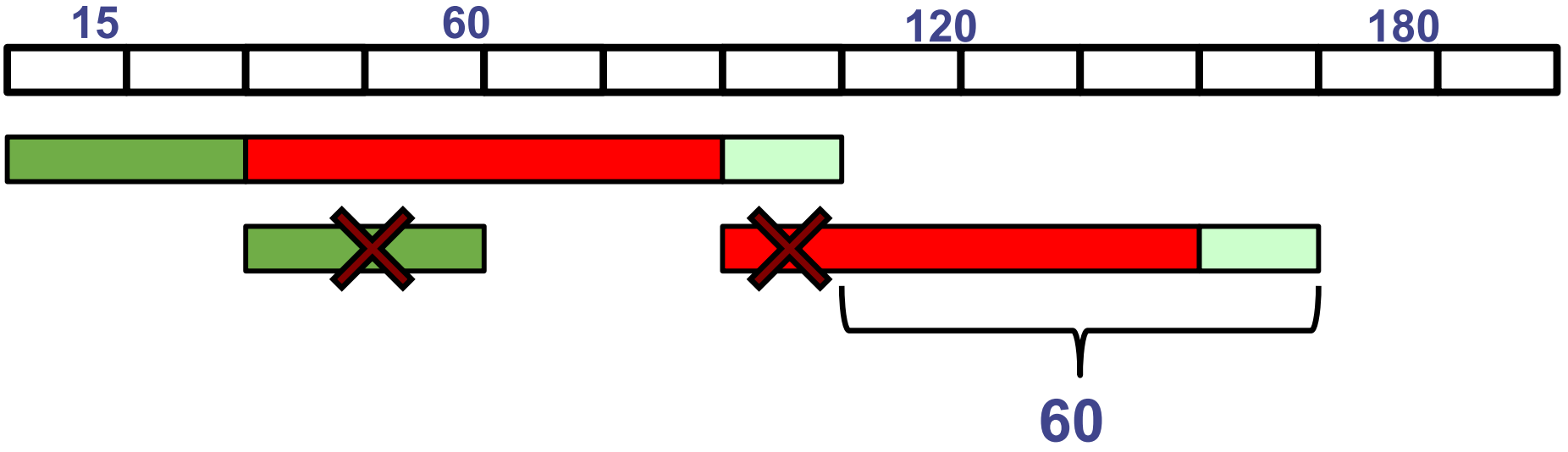


In-Class Exercise

- You have a washer, dryer, and "folding robot"
  - Each takes 1 unit of time per load
  - How long for one load in total?
  - How long for two loads of laundry?
  - How long for 100 loads of laundry?

- Now assume:
  - Washing takes 30 minutes, drying 60 minutes, and folding 15 min
  - How long for one load in total?
  - How long for two loads of laundry?
  - How long for 100 loads of laundry?

In-Class Exercise Answers



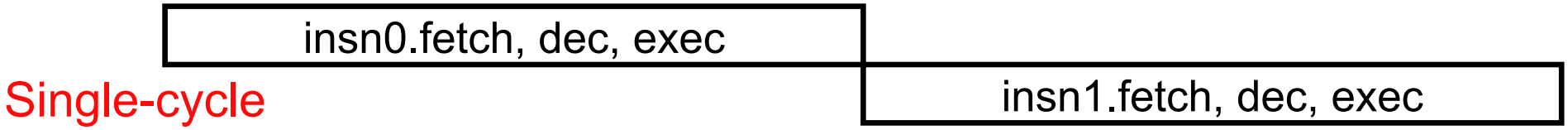
- Now assume:
  - Washing takes 30 minutes, drying 60 minutes, and folding 15 min
  - How long for one load in total? **105 minutes**
  - How long for two loads of laundry? 105 + 60 = **165 minutes**
  - How long for 100 loads of laundry? 105 + 60*99 = **6045 min**

Pipelined Datapath

Recall: Latency vs. Throughput

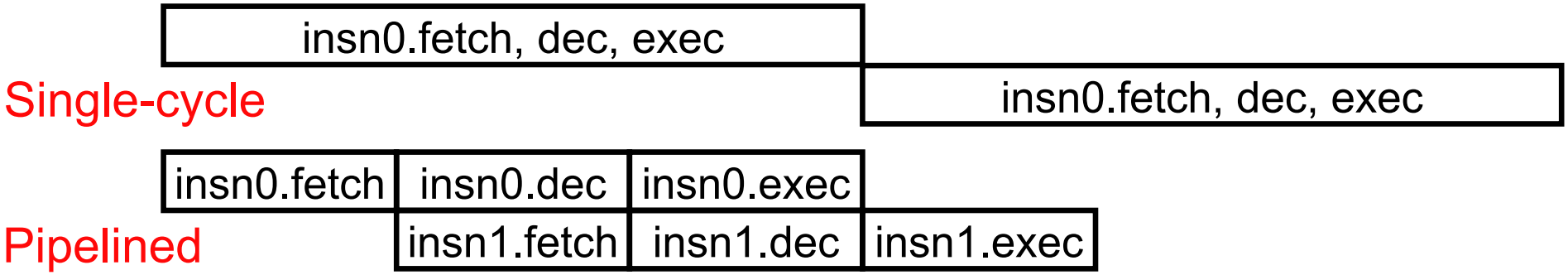
- **Latency (execution time)**: time to finish a fixed task
- **Throughput (bandwidth)**: number of tasks in fixed time
- Choose definition of performance that matches your goals
  - Scientific program? Latency, web server: throughput?

Latency vs. Throughput

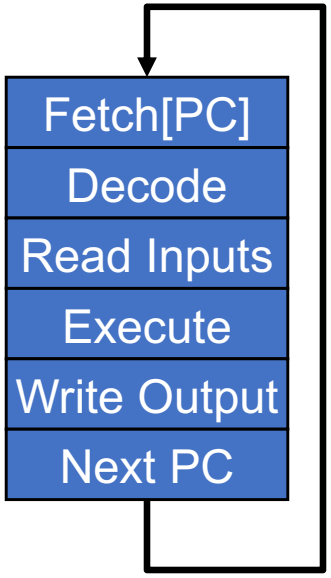


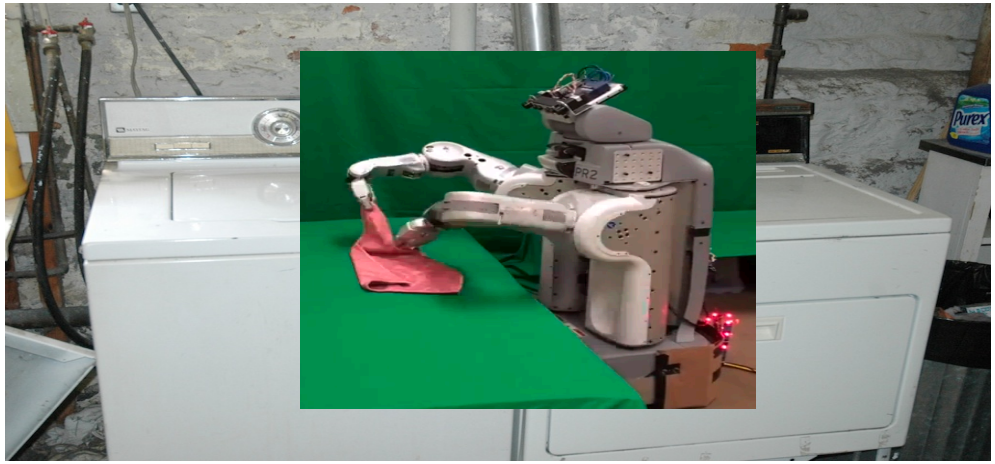
- Single instruction latency
  - Doesn't matter: programs comprised of billions of instructions
  - Difficult to reduce anyway
- Goal is to make programs, not individual insns, go faster
  - Instruction throughput → program latency
  - Key: **exploit Inter-insn Parallelism**

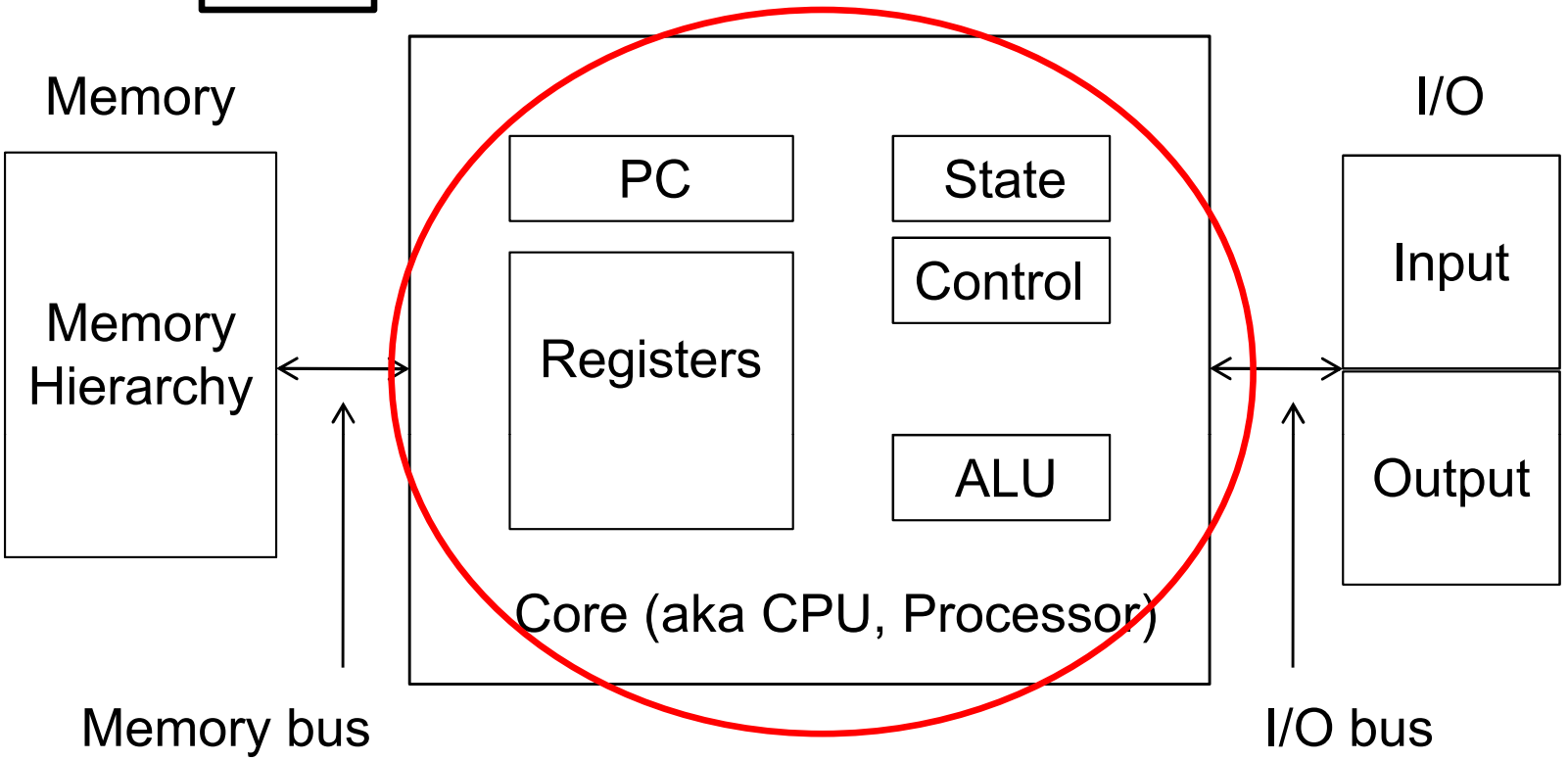
Pipelining



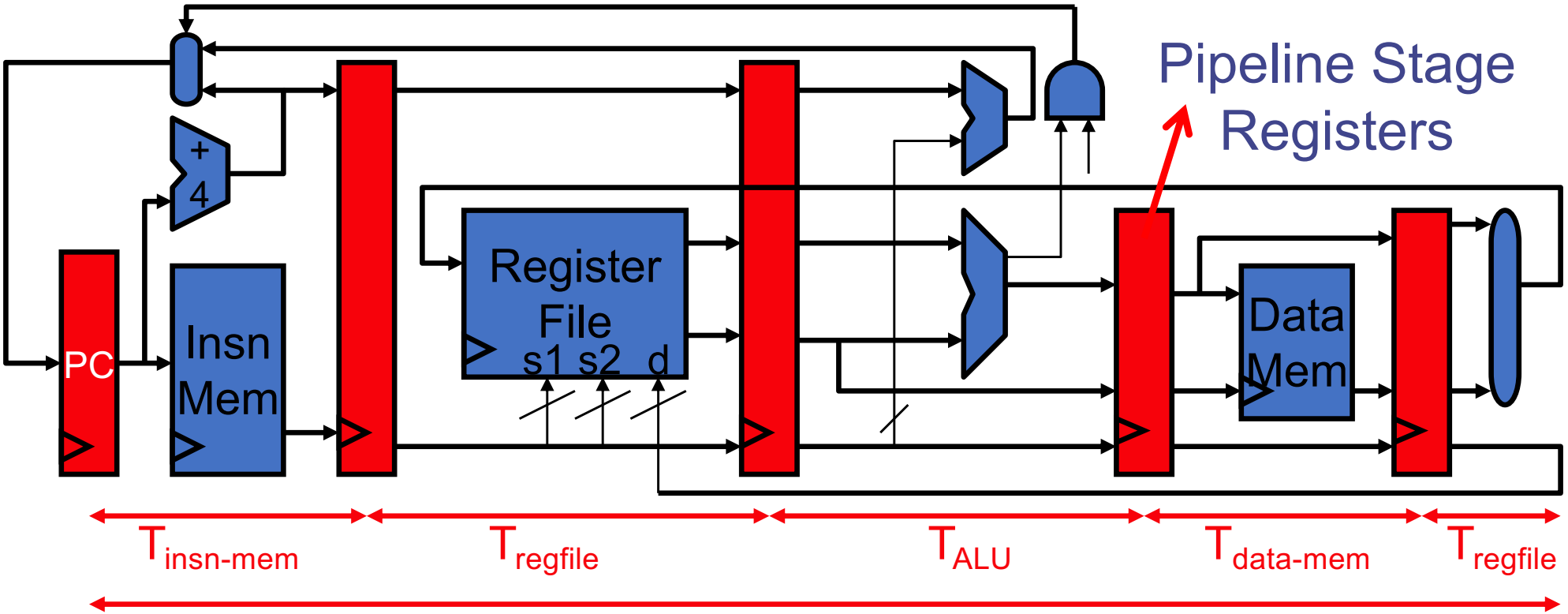
- Important performance technique
  - **Improves instruction throughput, not instruction latency**
- How it works
  - When insn advances from stage 1 to 2, next insn enters at stage 1
  - Maintains illusion of sequential fetch/execute loop
  - Individual instruction takes the same number of stages
  - \+ **But instructions enter and leave at a much faster rate**


Fetch[PC]
Decode
Read Inputs
Execute
Write Output
Next PC



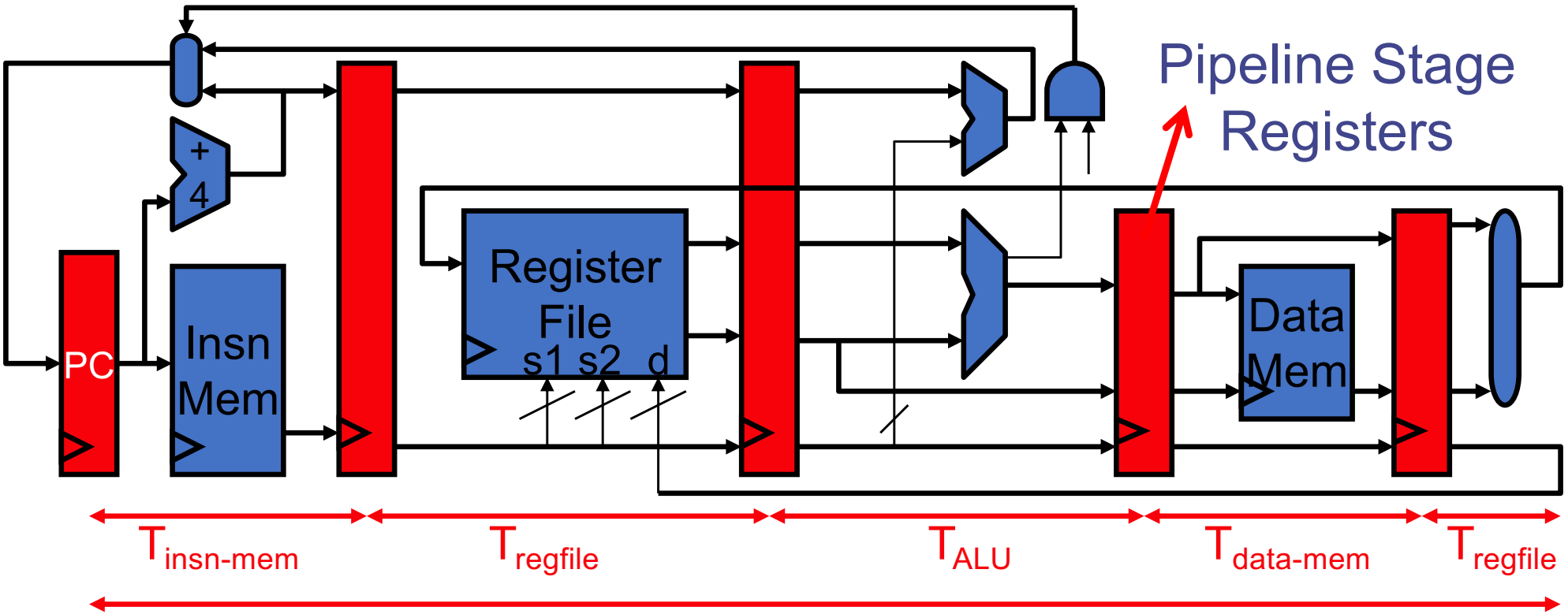

Memory
Memory Hierarchy
PC
State
Control
Registers
ALU
Core (aka CPU, Processor)
I/O
Input
Output
Memory bus
I/O bus

5 Stage Pipeline: Inter-Insn Parallelism

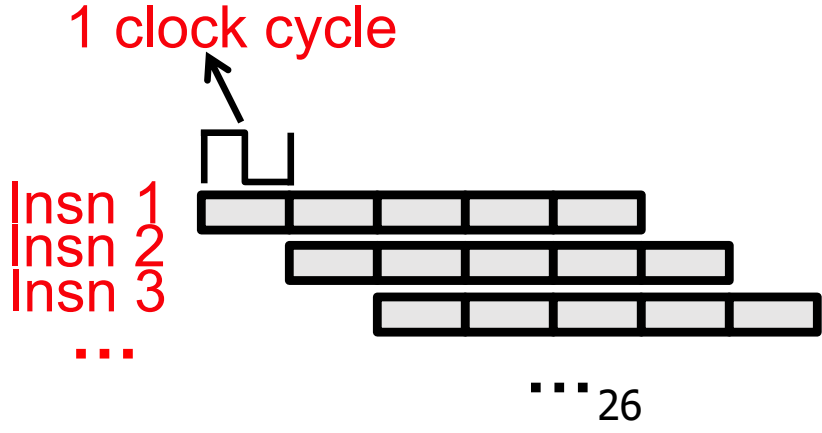


- If single-cycle:
  - Time to execute each instruction is $T_{singlecycle}$

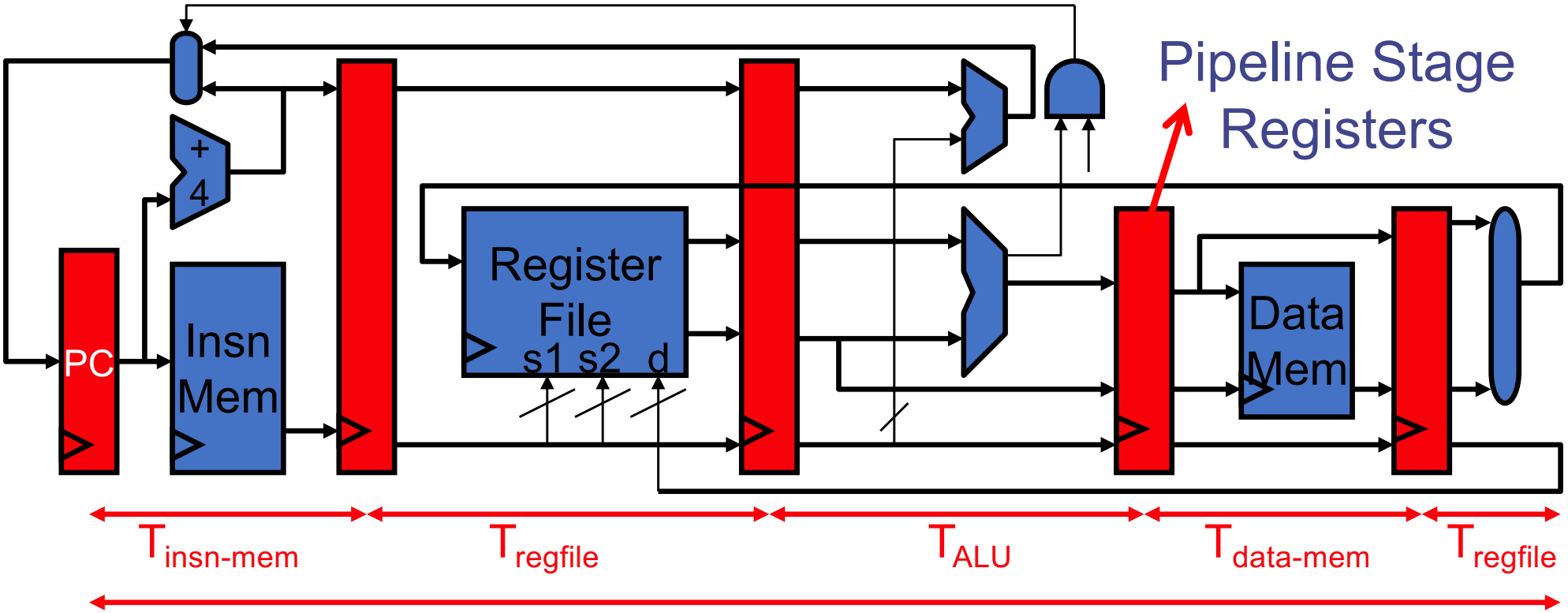
5 Stage Pipeline: Inter-Insn Parallelism



- **Pipelining**: cut datapath into N stages (here 5)
  - One insn in each stage in each cycle

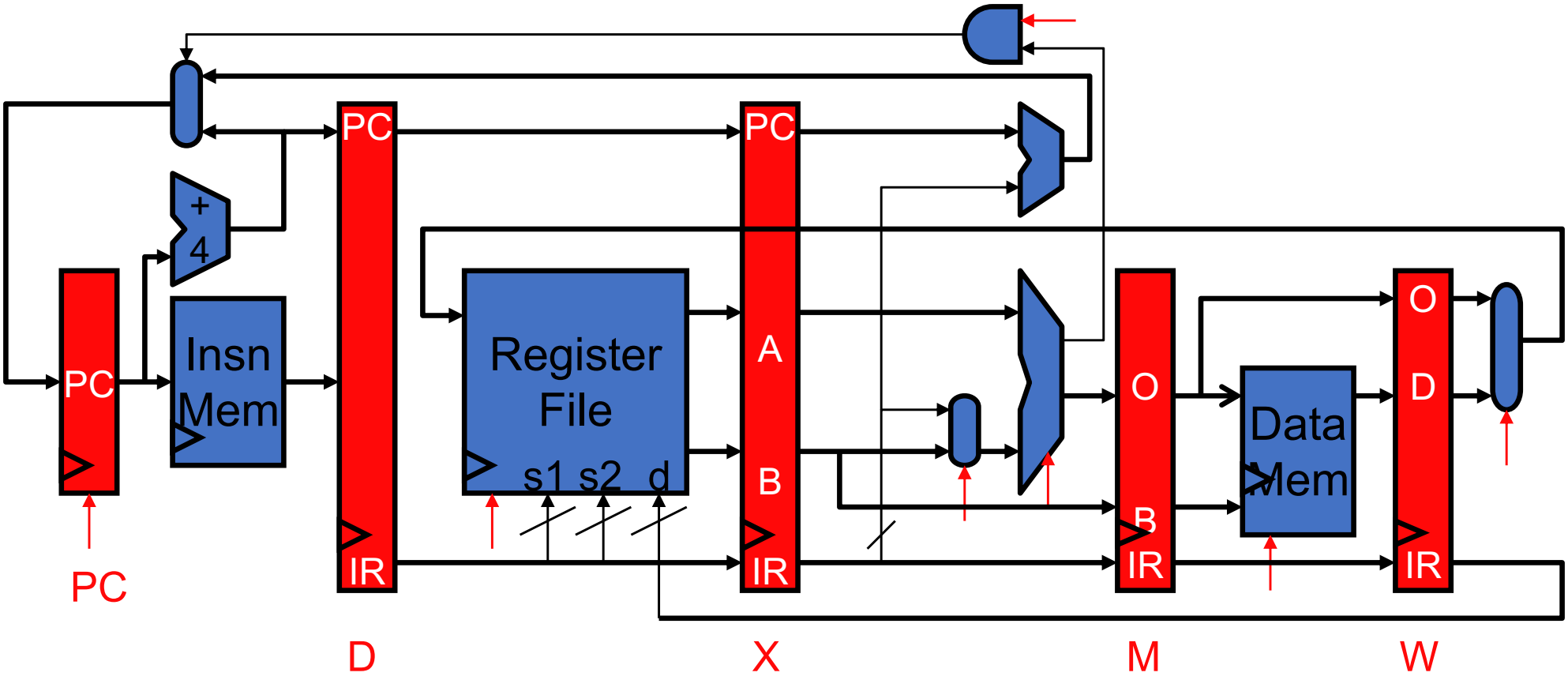


5 Stage Pipeline: Inter-Insn Parallelism



- **Pipelining**: cut datapath into N stages (here 5)
  - One insn in each stage in each cycle
  + Clock period = MAX($T_{insn-mem}$, $T_{regfile}$, T_{ALU}, $T_{data-mem}$)
  + Base CPI = 1: insn enters and leaves every cycle
    - Actual CPI > 1: pipeline must often "stall"
  - Individual insn latency increases (pipeline overhead)

5 Stage Pipelined Datapath



- Five stage: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
  - Nothing magical about 5 stages (Pentium 4 had 22 stages!)
- Latches (pipeline registers) named by stages they begin
  - **PC**, **D**, **X**, **M**, **W**