CS3350B Computer Organization
Chapter 4: Instruction-Level Parallelism
Part 1: Pipelining

Iqra Batool

Department of Computer Science
University of Western Ontario, Canada

Wednesday February 28, 2024

Outline

Instruction-Level Parallelism

For a computer architecture, its **instruction-level parallelism** (ILP) is a measure of the number of instructions it can perform simultaneously.

ILP is usually achieved *dynamically*—after compile time—by the processor itself manipulating program execution.

Circuitry (and appropriate control signals) needs to be added to the processor to handle the execution of many instructions simultaneously and to handle the dynamic nature of ILP.

Achieving ILP

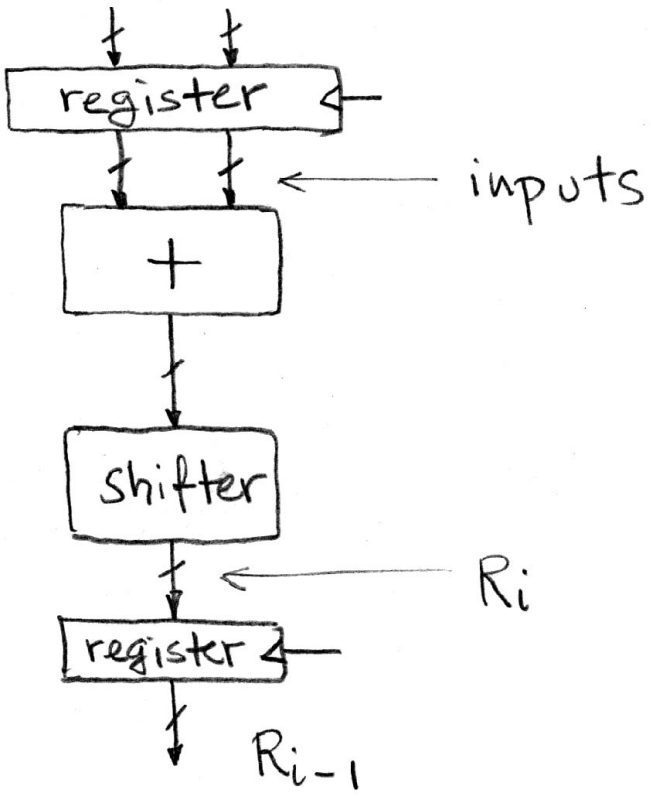
ILP can be achieved in many ways. Some topics we will look at:

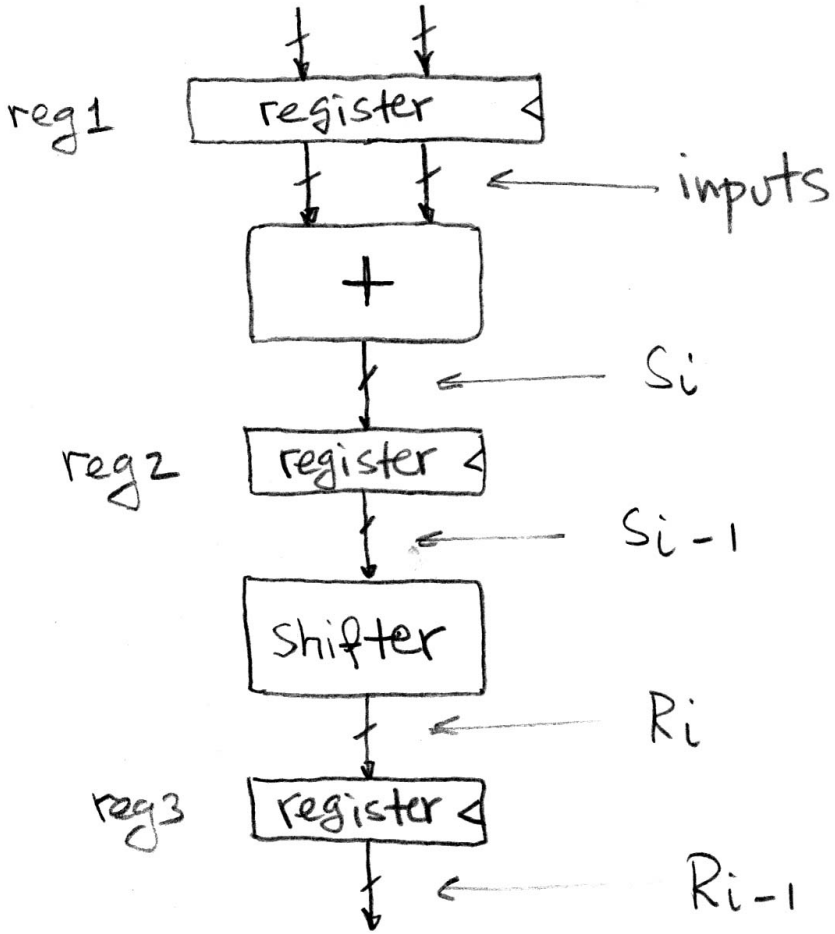
- **Pipelining**
- **Superscalar** execution
- **VLIW** – very long instruction word
- **Register renaming**
- **Branch prediction**

Lesson Goals

- Single-cycle vs Multi-cycle vs Pipelined data path.
- Understand pipelining as a concept to achieve parallelism.
- Understand how circuitry is used to:
  - ↳ Enable multi-cycle datapaths; in turn enabling pipelining.
  - ↳ Decrease minimum clock cycle time.
- Pipelining metrics: latency, throughput, ideal speedup, actual speedup.

"Pipelining" in Combinational Circuits





Break up a combinational circuit, reduce propagation delay, insert a register to store intermediate results, increase clock frequency.

Pipe, Pipeline, Pipelining

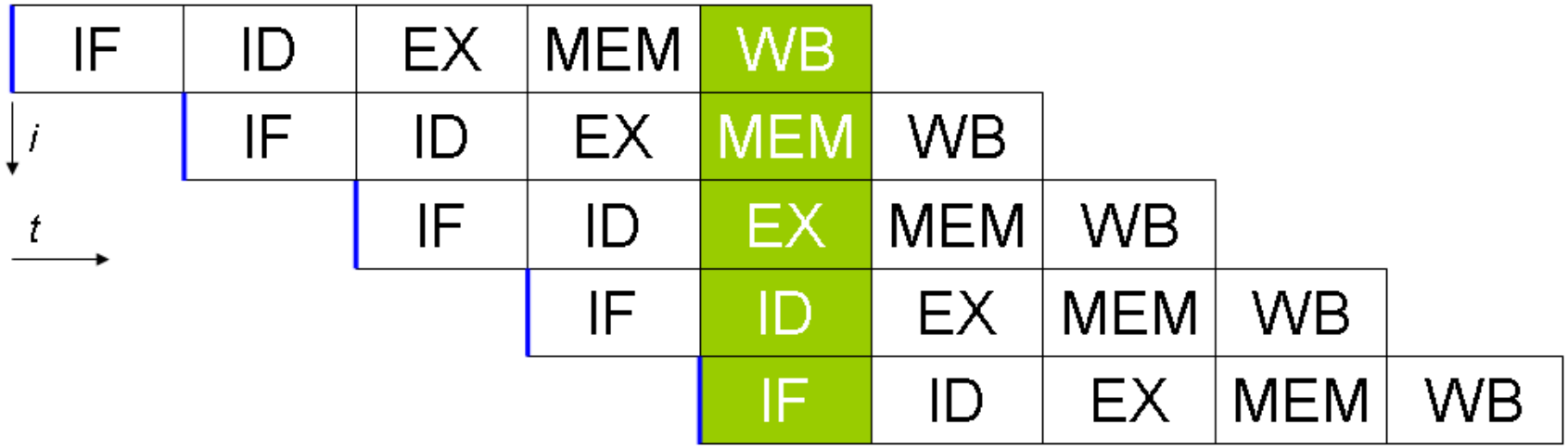
Unix pipe: pass data from one program to another.

```
ls -la | grep "foo.txt"
```

Data pipeline: a sequential series of processing elements (CPUs, circuits, programs, etc.) where the output of one is passed as the input to another. Buffer storage is needed between elements to store temporary data.

Pipelining: a technique for instruction-level parallelism where each stage of the datapath is always kept busy. Instructions are **overlapped**.

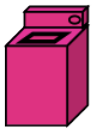
Pipelining the RISC Datapath

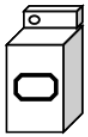


- Each stage is executing a *different* instruction.
- 5 stages $\Longrightarrow$ 5 instructions executed at once.

Outline

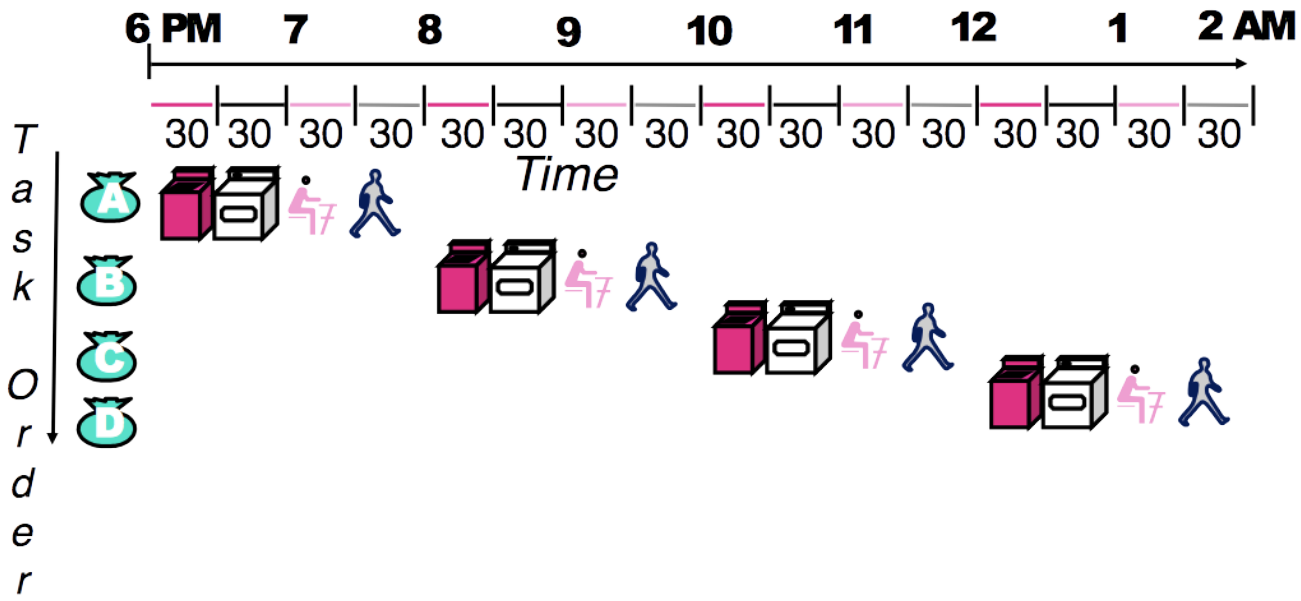
Doing Laundry





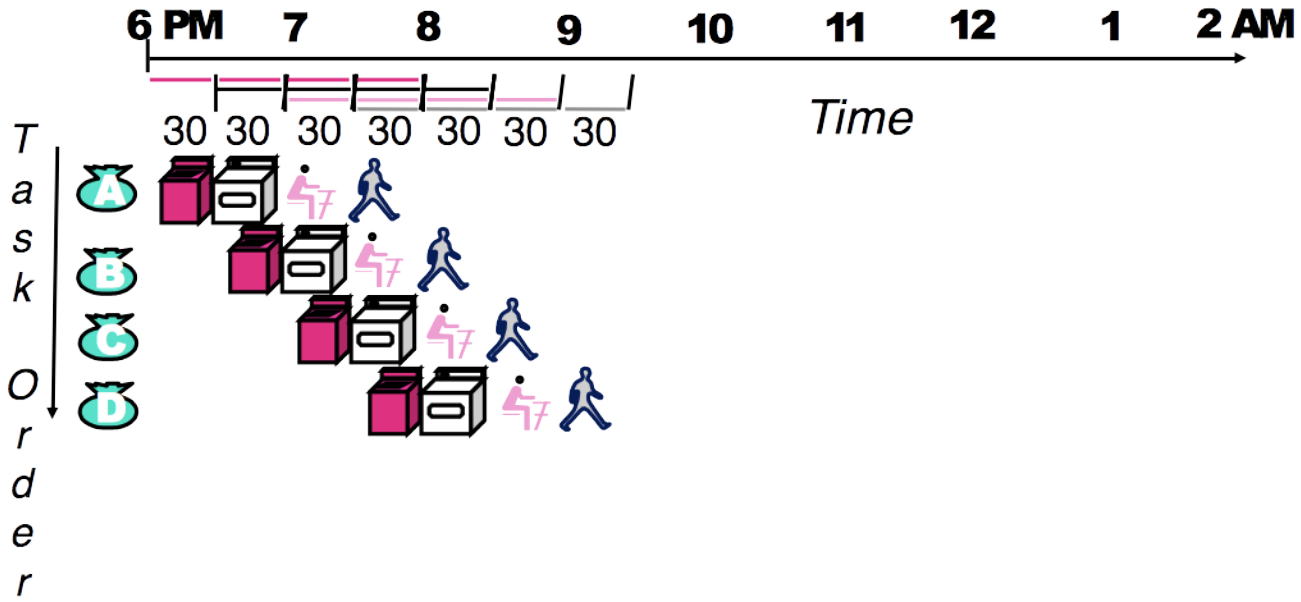
- We have 4 loads of laundry to do: A, B, C, D.
- To process each load we need to:
  - Wash
  - Dry
  - Fold
  - Put-away
- Each stage of doing laundry takes 30 minutes.
- Could process each load sequentially or use *pipelining*.

Doing Laundry: Sequentially



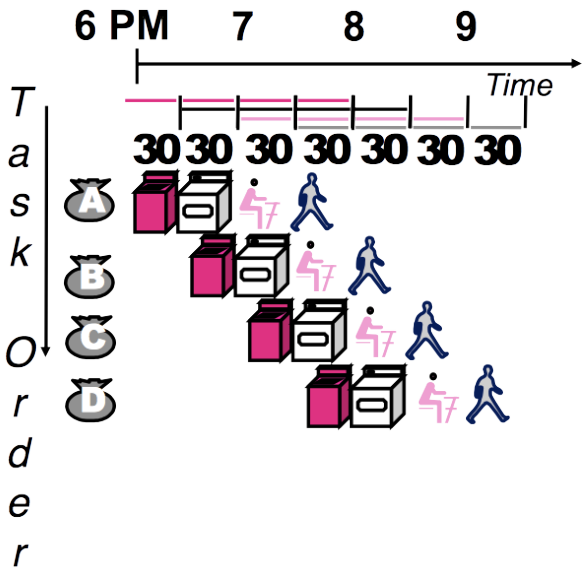
- Each load of laundry is done one at a time:
  - ↳ Wash A, Dry A, Fold A, Put-away A.
  - ↳ Wash B, Dry B, Fold B, Put-away B.
  - ⋮
- Takes 8 hours in total. There has to be a better way.

Doing Laundry: Pipeline



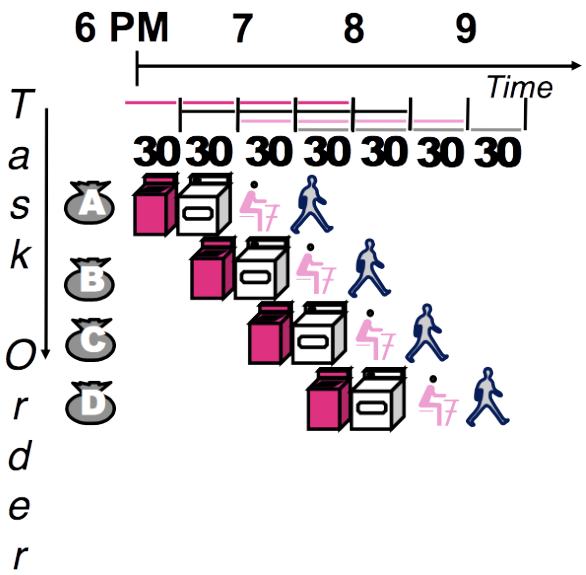
- Each *stage* of doing laundry must process each load sequentially.
- **But** each load of laundry can overlap.
- No dependency between drying load A and washing load B, etc.
- Put-away A while Folding B while drying C while washing D.
- Takes 3.5 hours in total.

Pipelining Terms via Analogy



- Pipelining: many tasks (loads of laundry) being executed simultaneously using different resources (washer, dryer, etc.).
- Time to complete a single task (**latency**) *does not* change.
  - ↳ Each load by itself still takes 2 hours.
- Number of tasks that can be completed in one unit of time (**throughput**) increases.
- Potential speed up via pipelining equals the number of stages in pipeline.
- Actual speed-up never exactly equals potential.

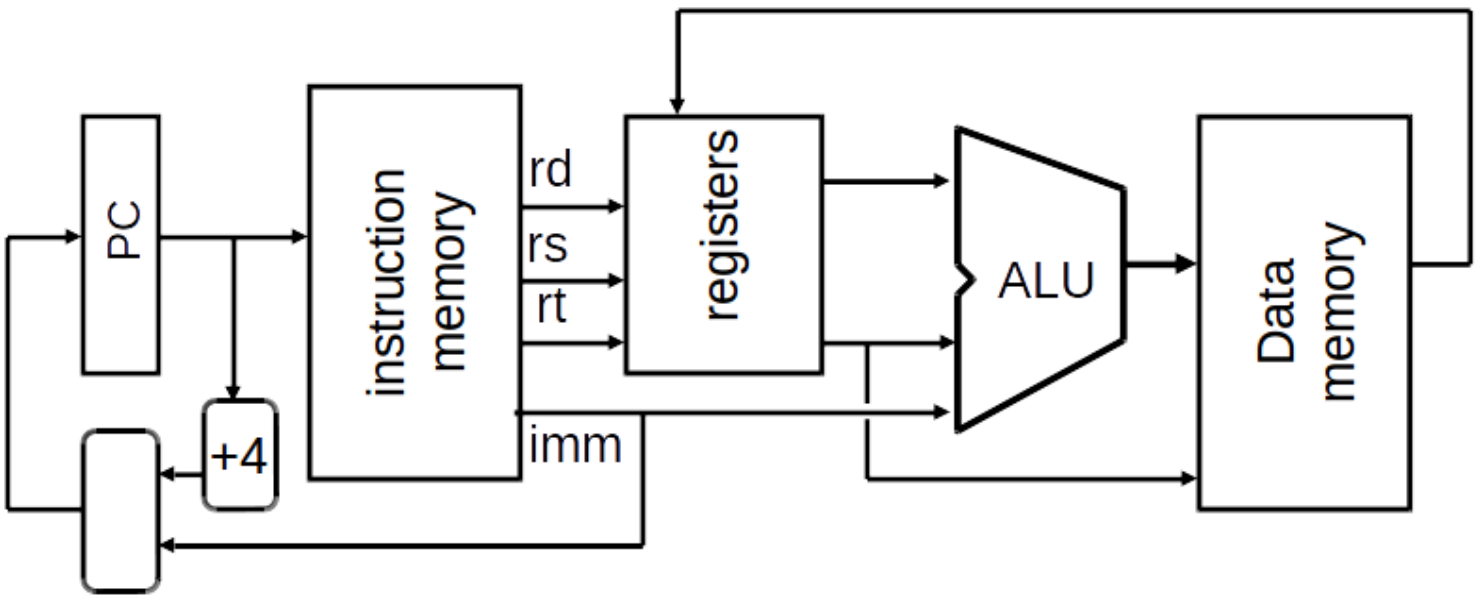
Pipelining Terms via Analogy



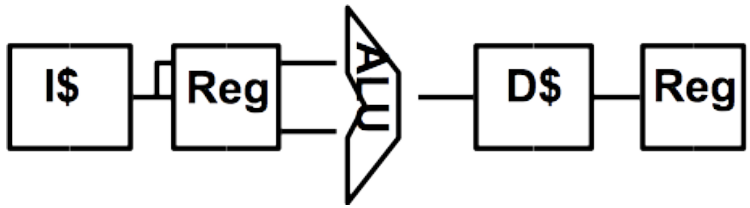
- Actual speed-up never exactly equals potential.
- **Fill time:** time taken to "fill" the pipeline. Initially, not every stage is used.
- **Drain time:** time taken to "empty" the pipeline. Not all stages are used once the last task begins.
- Imagine a new washing machine takes only 20 minutes. This *does not* increase pipeline speed.
  - ↳ Dryer still takes 30 minutes.
  - ↳ Washer must wait for dryer to finish before laundry can move from washer to dryer.

Outline

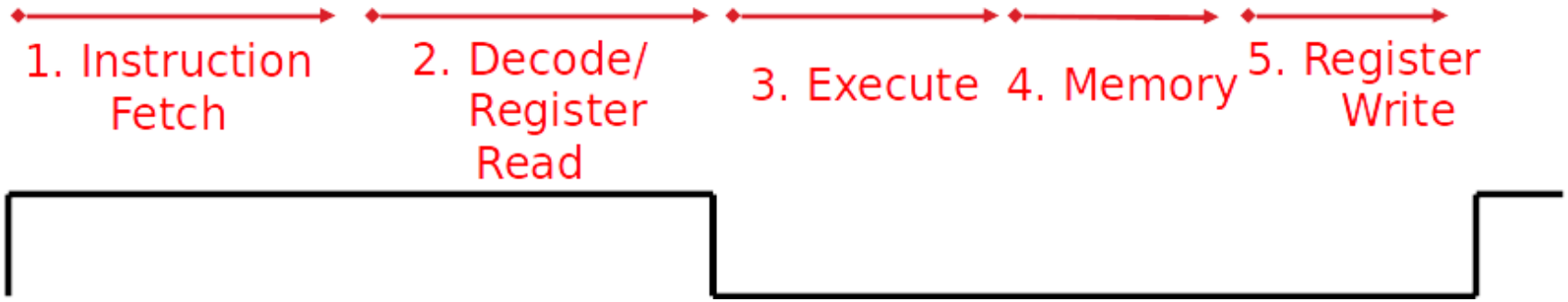
The RISC Datapath



IF | ID | EX | MEM | WB



Review: Single Cycle Datapath



- Clock cycle is long enough to handle *critical path* through datapath.
- Time for data to pass through entire datapath.

Performance of Single Cycle Datapath

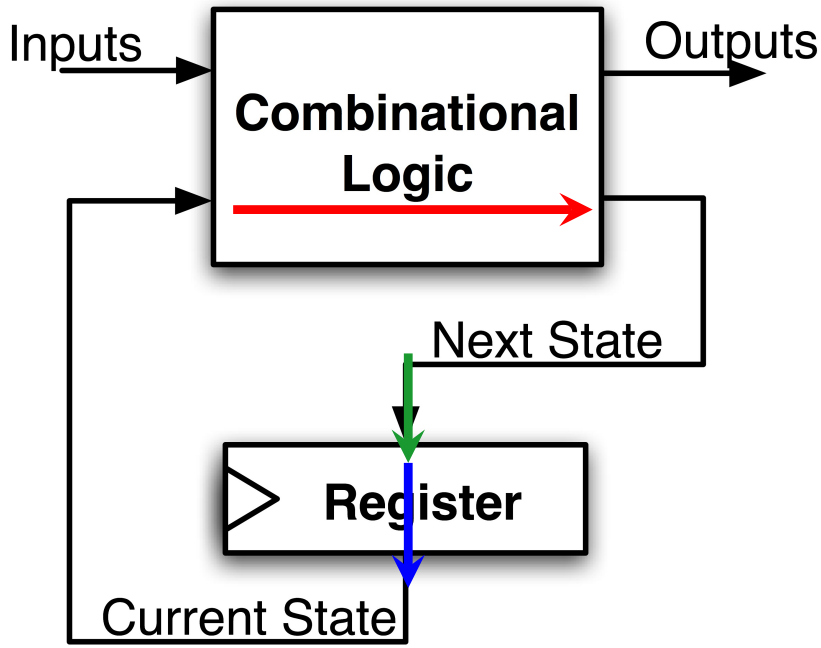
- Let's assume that accessing memory takes 200ps and ALU propagation delay is 200ps.
  - ↳ IF stage, EX stage, MEM stage.
- Let's assume accessing registers takes 100ps.
  - ↳ ID stage, WB stage.

- **What is the minimum clock cycle?**
  - ↳ Sum of all stages since some instructions use all stages.
  - ↳ $200 + 100 + 200 + 200 + 100 = 800$ps.

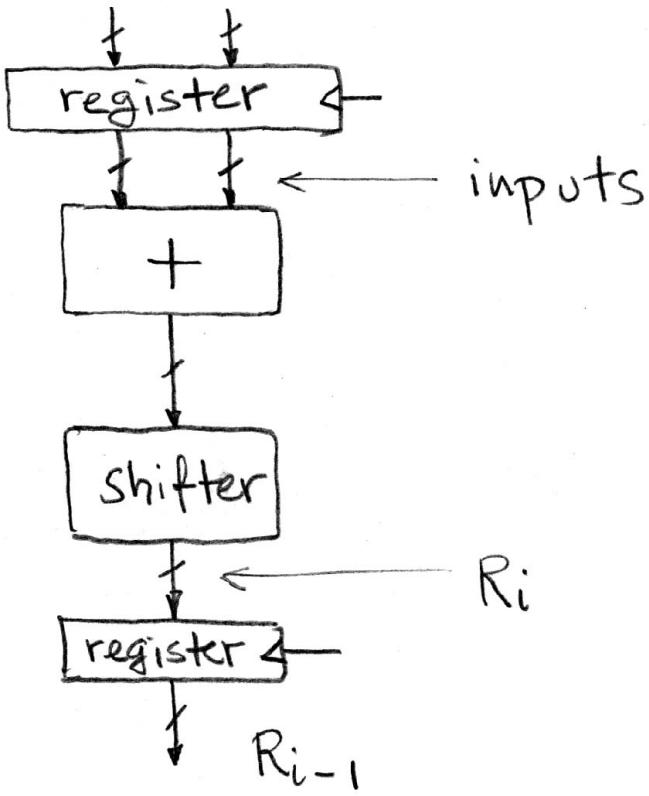
Instr.	IF	ID	EX	MEM	WB	Total
R-type	200ps	100ps	200ps	-	100ps	600ps
Branch	200ps	100ps	200ps	-	-	500ps
sw	200ps	100ps	200ps	200ps	-	700ps
lw	200ps	100ps	200ps	200ps	100ps	**800ps**

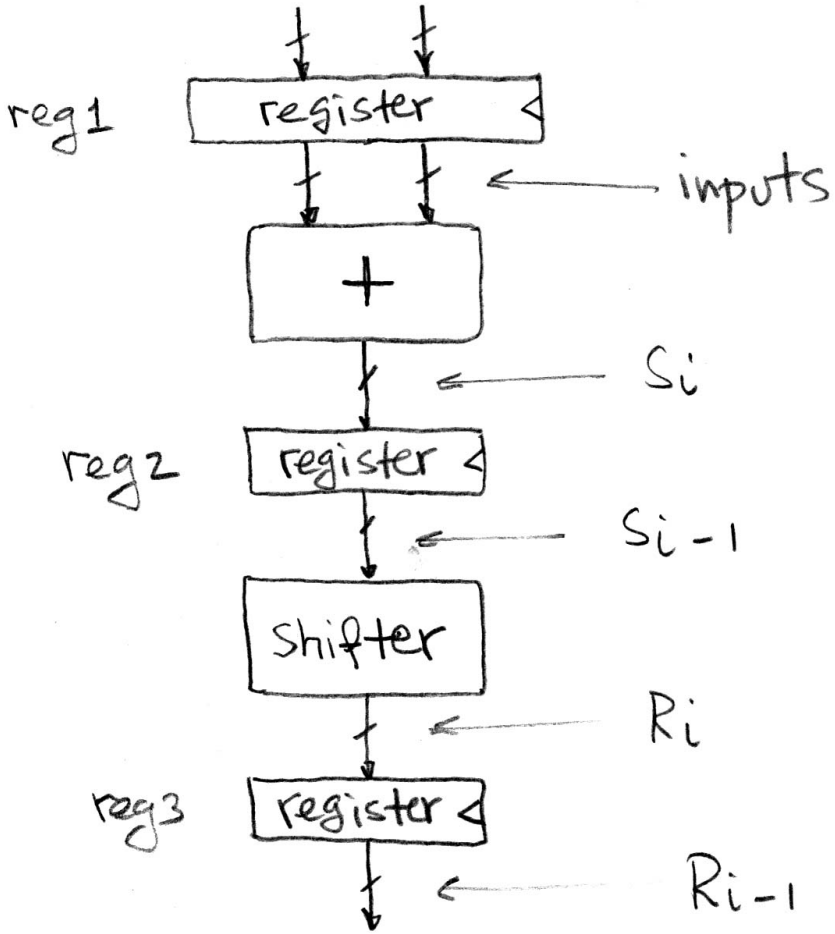
Improving Performance of Datapath

- **Clock frequency**
- Parallel execution of instructions via overlap: **pipelining**.
- Superscalar, VLIW (to come later).
- Branch prediction (to come later).



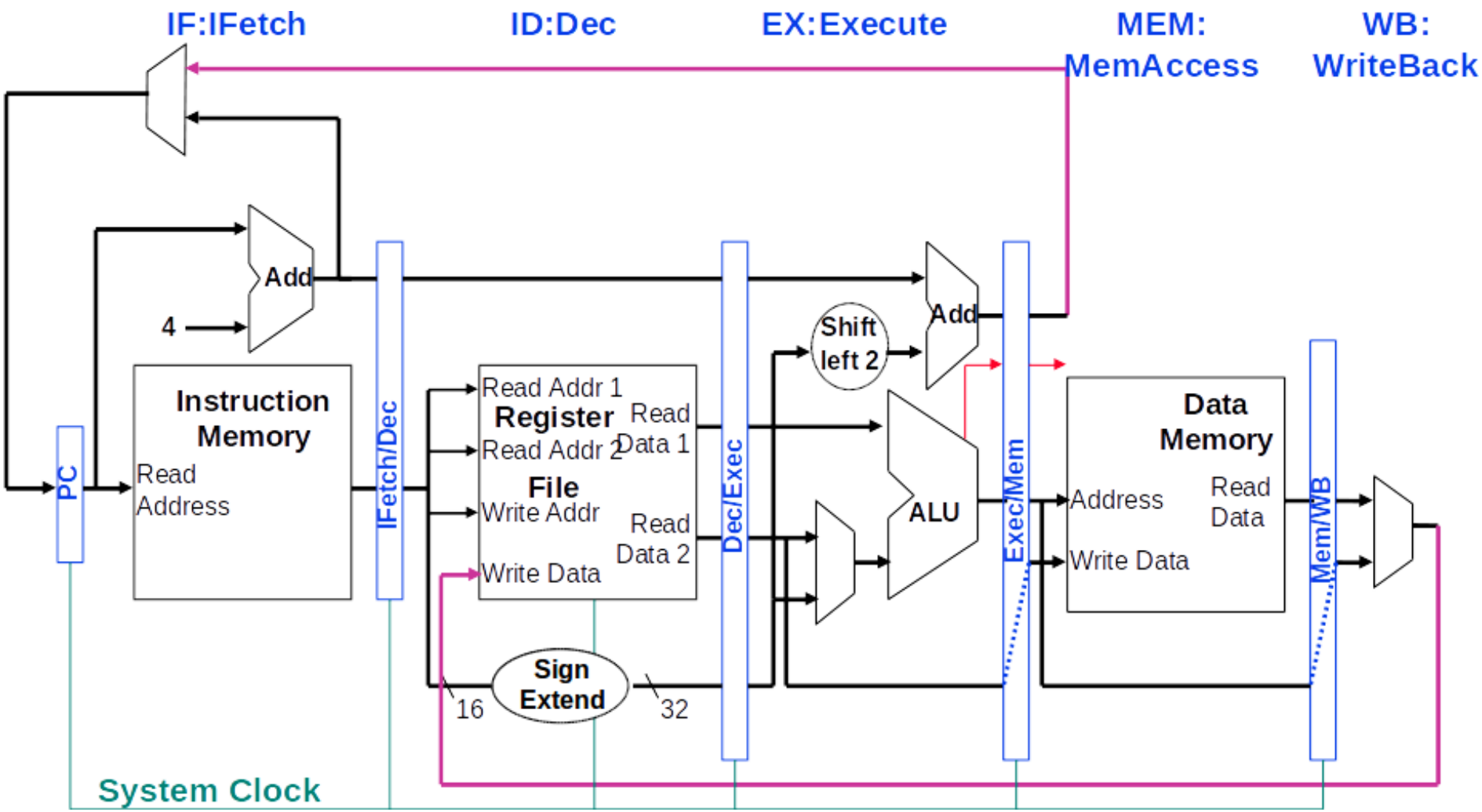
Review: Multi-Cycle for Combinational Circuits



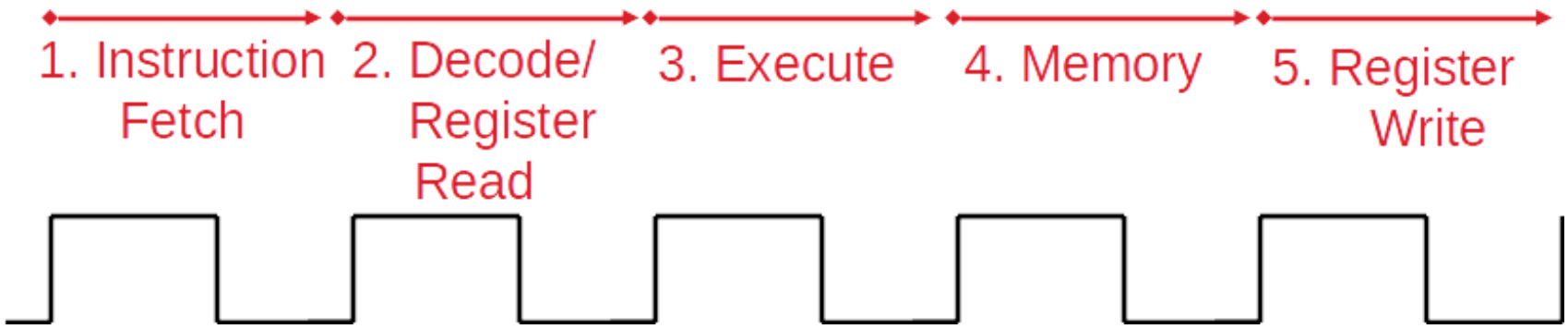


Break up a combinational circuit, reduce propagation delay, insert a register to store intermediate results, increase clock frequency.

Multi-Cycle Datapath for MIPS



Multi-Cycle Datapath



- Clock cycle is long enough to handle *slowest stage* of the pipeline.
- Time for data to pass through one (the slowest) stage of pipeline.

Example: Minimum clock cycle is 200ps.

Instr.	**IF**	ID	**EX**	**MEM**	WB	Total
R-type	**200ps**	100ps	**200ps**	-	100ps	600ps
Branch	**200ps**	100ps	**200ps**	-	-	500ps
sw	**200ps**	100ps	**200ps**	**200ps**	-	700ps
lw	**200ps**	100ps	**200ps**	**200ps**	100ps	800ps

Pipelining for Performance

- Further increase clock frequency?
- *Could* break up datapath into more and more stages but...
  - ↳ More registers.
  - ↳ More complexity in datapath and controller design ⇒ overhead.
  - ↳ Still limited by slowest stage (memory).
- Leverage the parallelism gained by pipelining.
- Parallelism in execution of instructions yields fewer *cycles per instruction* (CPI)

The Classic Performance Equation

$$\begin{aligned} \text{CPU time} &= \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle} \\ &\text{or} \\ \text{CPU time} &= \text{Instruction\_count} \times \text{CPI} \,/\, \text{clock\_rate} \end{aligned}$$

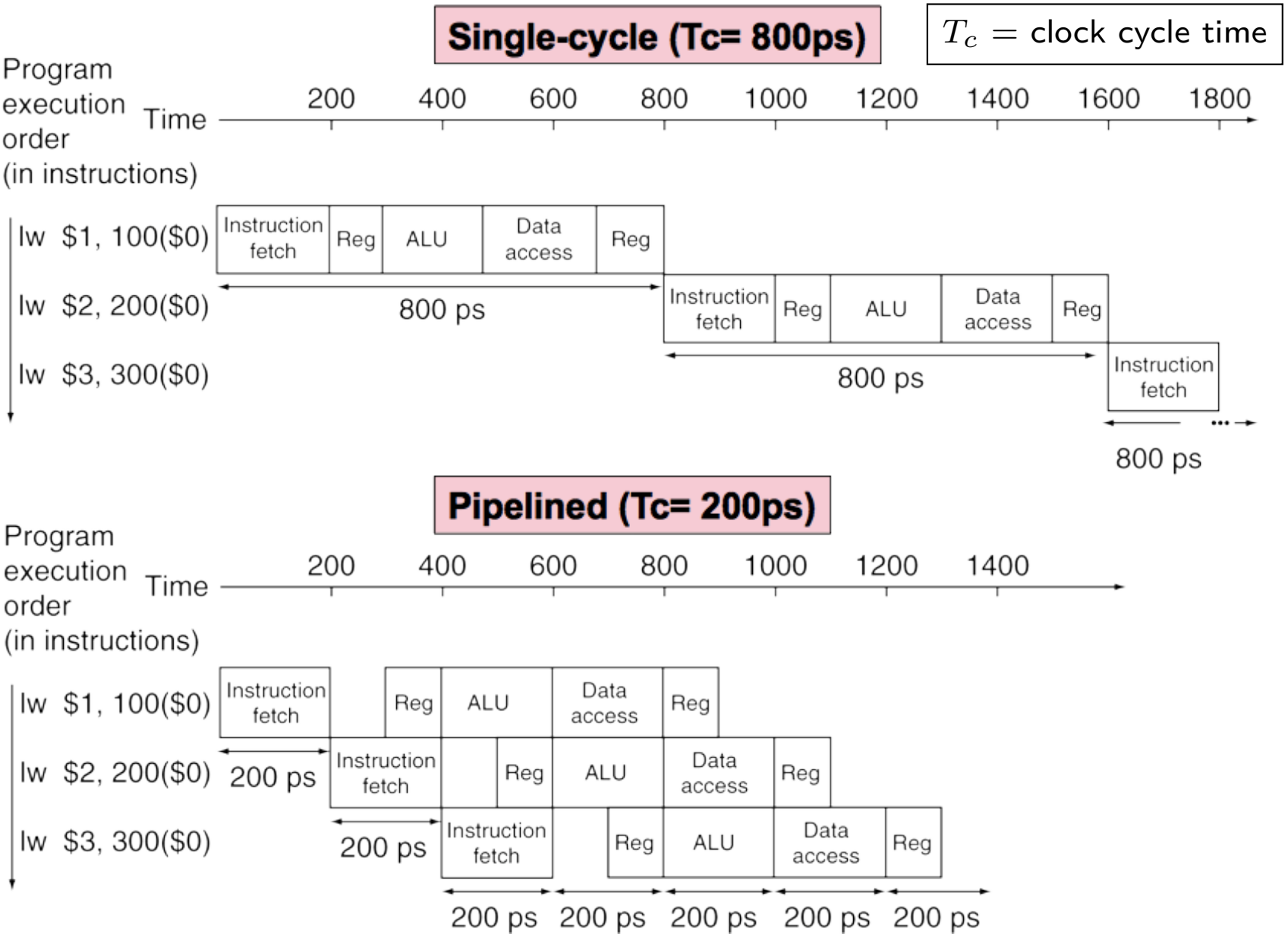
RISC Pipeline Performance

- Overlap instructions, start the next before the former completes.
- Some instructions will "waste" a cycle as they flow through unused stages.

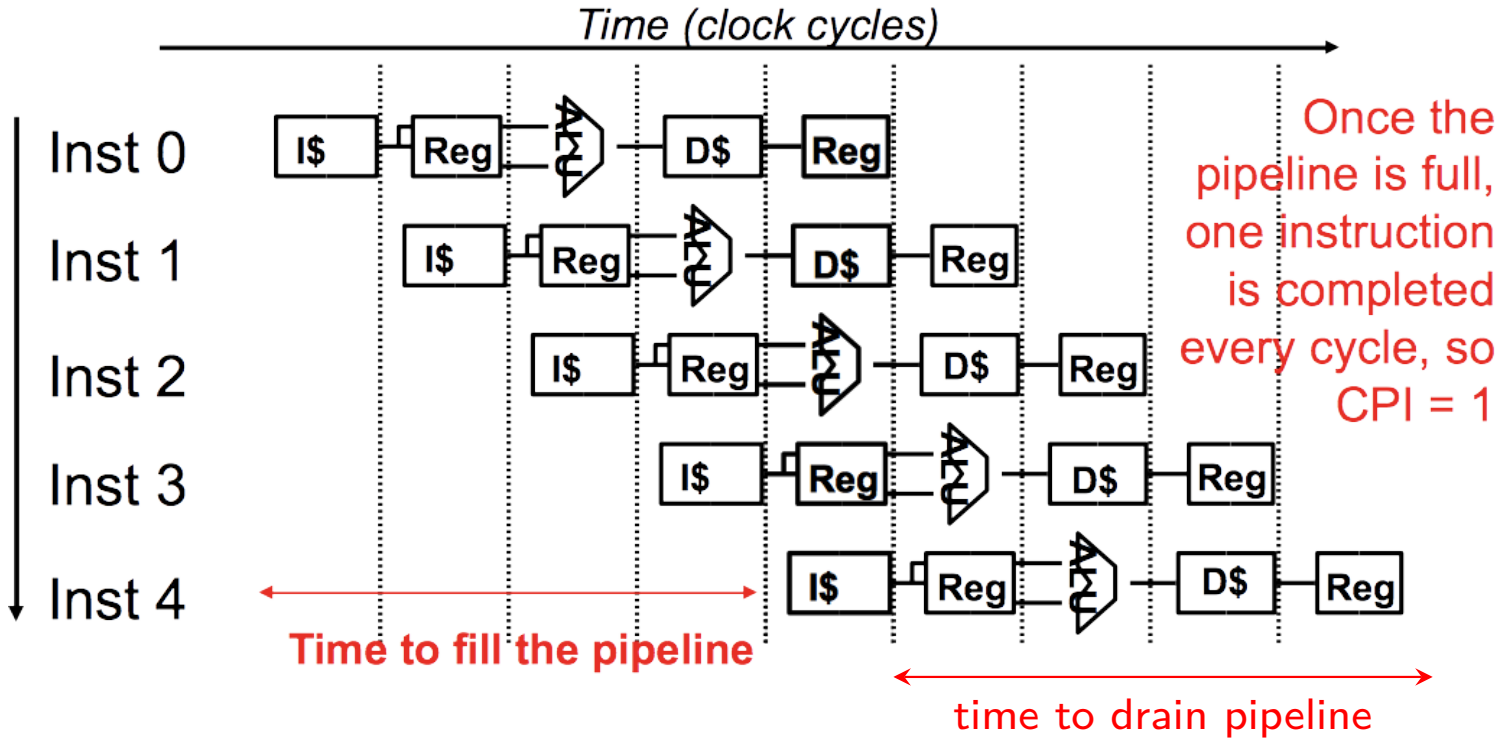
	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
`lw`	IFetch	Dec	Exec	Mem	WB			
`sw`		IFetch	Dec	Exec	Mem	WB		
`add`			IFetch	Dec	Exec	Mem	WB	

- **Latency:** time to complete one instruction. Does not decrease with pipelining. May actually increase slightly if each stage originally did not take the same amount of time.
- **Throughput**: number of instructions that can be completed in some amount of time. Increases with pipelining.
- Once pipeline is full CPI is 1.

Performance: With and Without Pipelining



Pipeline Parallelism



- Potential speed-up via parallelism is equal to the number of stages.
- 5 stages $\Longrightarrow$ 5x potential speed up.
- A pipeline is "full" when every stage is occupied by an instruction (every stage does not have to necessarily be doing work).
- Pipeline **fill time** and **drain time** reduce actual speed up.

Quantifying Pipelined Speedup

If the time for each stage is the same:

$$\textbf{Ideal Speedup} = \text{Number of Stages}$$

If the time for each stage is not the same:

$$\textbf{Ideal Speedup} = \frac{\text{Time between instructions}_{\text{single-cycle}}}{\text{Time between instructions}_{\text{pipelined}}}$$

$$\textbf{Actual Speedup} = \frac{\text{Time to complete}_{\text{single-cycle}}}{\text{Time to complete}_{\text{pipelined}}}$$

Note that a single-cycle datapath always has latency less than or equal to a multi-cycle/pipelined datapath.

Calculating Speedup

From previous example:

- Single-cycle datapath: 800ps clock cycle.
- Pipelined: 200ps clock cycle.
- **Uneven time for each stage**. ID and WB only 100ps.
- 3 `lw` instructions.

$$\text{Ideal Speedup} = \frac{800}{200} = 4 \qquad \text{Actual Speedup} = \frac{2400}{1400} = 1.714$$

- If we have 1000000 `lw` instructions?

$$\text{Actual Speedup} = \frac{1000000 \times 800}{1000000 \times 200 + 800} \approx 4$$

Calculating Pipelined Time

Classic Performance Equation:

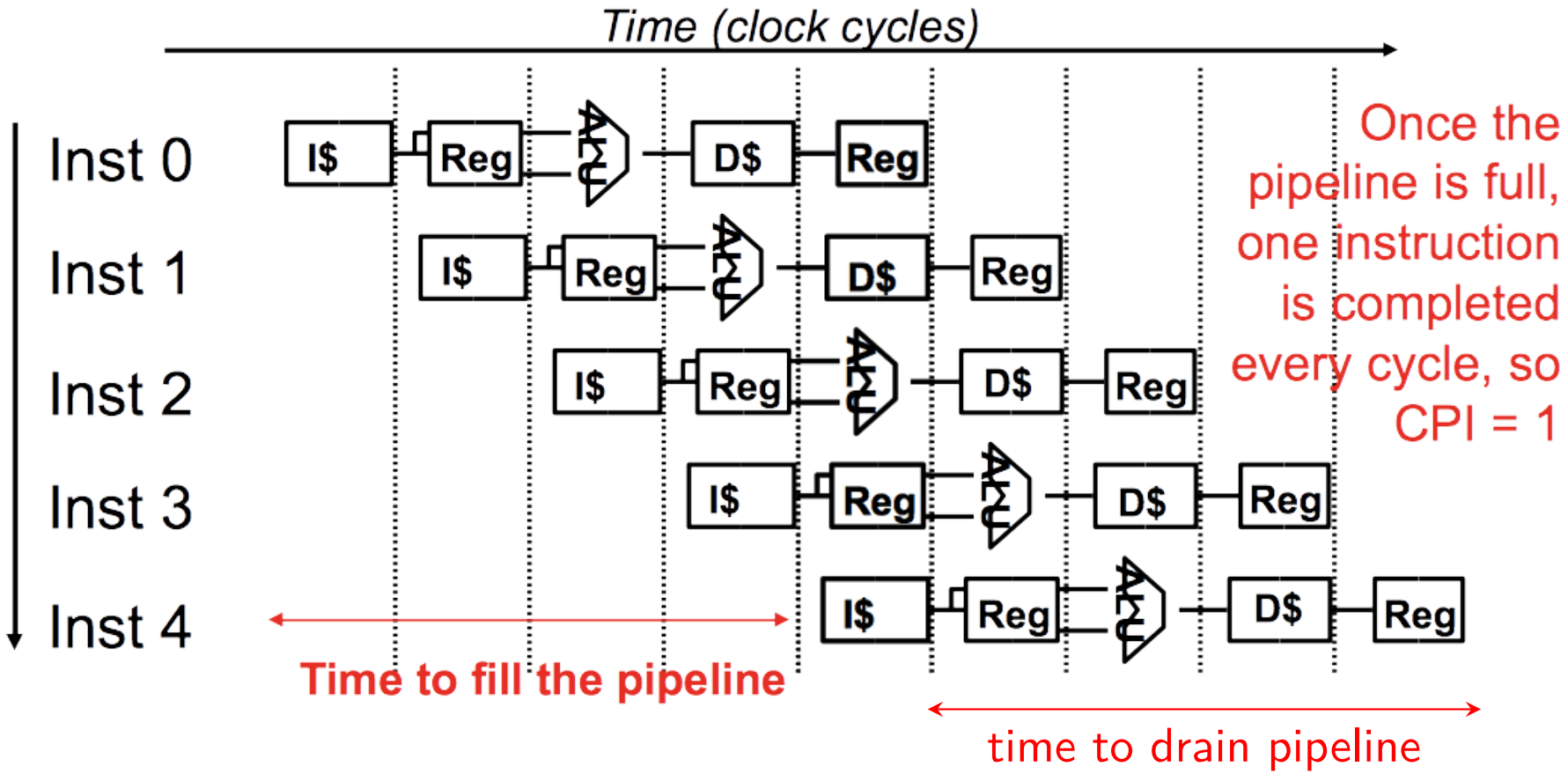
$$\text{CPU time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock cycle}$$

Time for pipelined execution:

$$\text{Time}_{\text{pipelined}} = \text{Fill time} + (\text{IC} \times \text{clock cycle})$$

- (Assuming no stalls or hazards.)
- Once pipeline is full, one instr. completes every cycle ⇒ CPI is 1.
  - ↳ Gives IC × 1 × clock cycle
- Pipeline is only *not* full during fill or drain time.
- Fill time = Drain time = (number of stages - 1) × clock cycle
  - ↳ Assuming number of instructions > number of stages.

Calculating Pipelined Time



Summary

- Pipelining is the simultaneous execution of multiple instructions each in a different stage of the datapath.
- Pipelining gives increased clock frequency by multi-cycle datapath.
- Limited by the slowest stage.
- Pipelining gives essentially a CPI of 1.
- Speed-up must account for fill time and drain time.
- All of the discussion so far assumed there is **no conflicts** between instructions, hardware, circuits, etc.
  - ↳ **Pipeline hazards** severely impact performance and potential speed-up.
  - ↳ Chapter 4: Part 2: Pipeline hazards.