






# CPSC 313: Computer Hardware and Operating Systems

Unit 2: Pipelining  
Intro and Overview

# Admin

- No tutorials this week!
  - Instead tutorial times will be treated as office hours
  - These will be great times to ask questions about lab 3.
- Lab 3 due Sunday

   **WARNING:**     
**It is significantly more challenging than  
the first two labs!!!**

# Admin (Lab 3)

- Good news:
  - You can work with a partner.
- Less good news:
  - You are still responsible for understanding all the code that you and your partner submit.
- Lab 3 is released as two separate parts:
  - Lab3-Ind: These are a set of questions about the code and assignment designed to prepare you to undertake the coding portion. Worth  $\frac{1}{4}$  of the lab.
  - Lab3-Partner: This is the larger part and can be completed with a partner. Worth  $\frac{3}{4}$  of the lab.

# Admin (Lab 3)

- Do not leave this until this weekend!!!
- Spend enough time understanding what you have to do so that you can ask good questions in office hours and tutorial.

# Introduction to pi(e)pelined CPU architectures

- The key observation is that most of the time, most of the circuitry in our sequential CPU is idle.
- We want to avoid so much idle hardware
- Remember the little ponies!



# Please Do Steve's Laundry

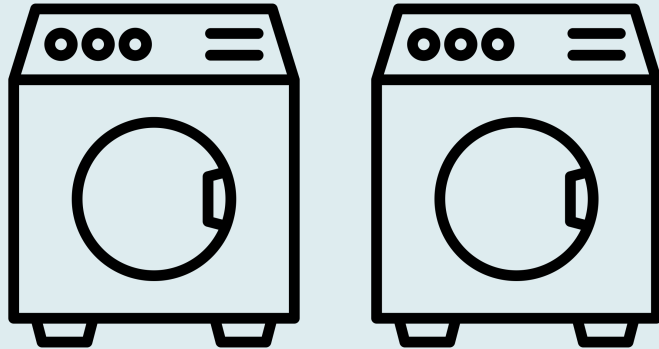
## Which is Better?



Regular Laundry World

Washer:  
40 mins

Dryer:  
60 mins

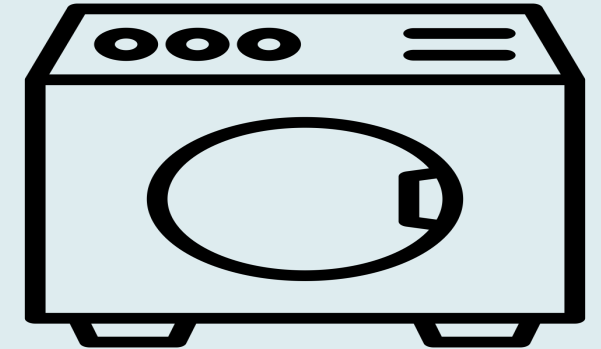


time to change: 20 minutes

CPSC 313 (Steve's reaaally sloooow)

Magic Laundry World

Magical  
Washer+Dryer:  
100 mins



(such a thing could not  
truly exist in mundane reality)

# Ha! Steve Has Kids. Now Do Steve's Laundry

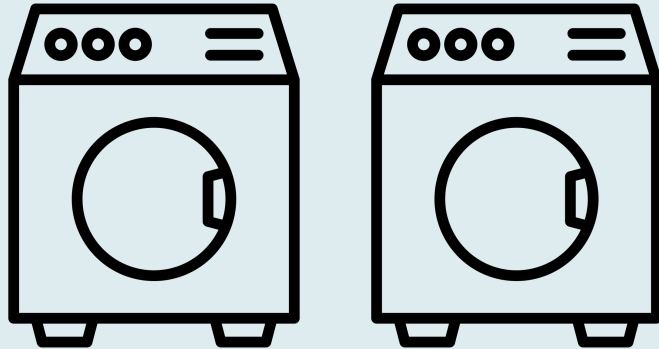
## Which is Better?



Regular Laundry World

Washer:  
40 mins

Dryer:  
60 mins

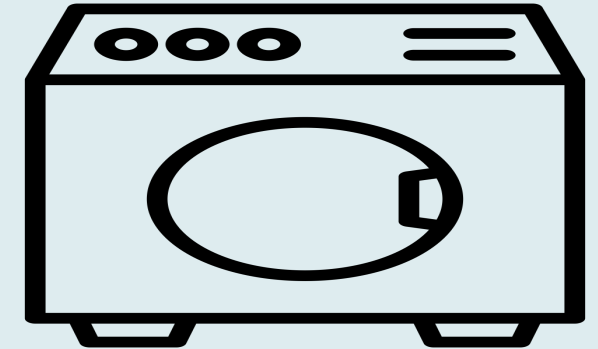


time to change: 20 minutes

CPSC 313 (Steve's reaaally sloooow)

Magic Laundry World

Washer+Dryer:  
100 mins



# Terminology we'll discuss now

- **Stage:**
  - What we used to call “phase”.\*
  - When we talk about pipelines, each piece of the pipeline is a stage.
  - A stage is the unit of computation that the processor performs in a single clock cycle.
- **Instruction statuses:**
  - **In-flight:** somewhere in the pipeline
  - **Retired:** instruction has completed execution
  - Go back to the laundry metaphor

\* We will screw up this terminology regularly. It's there just to remind us there's a difference, not to either have you test us or us test you on it. Be forgiving! ◀◀



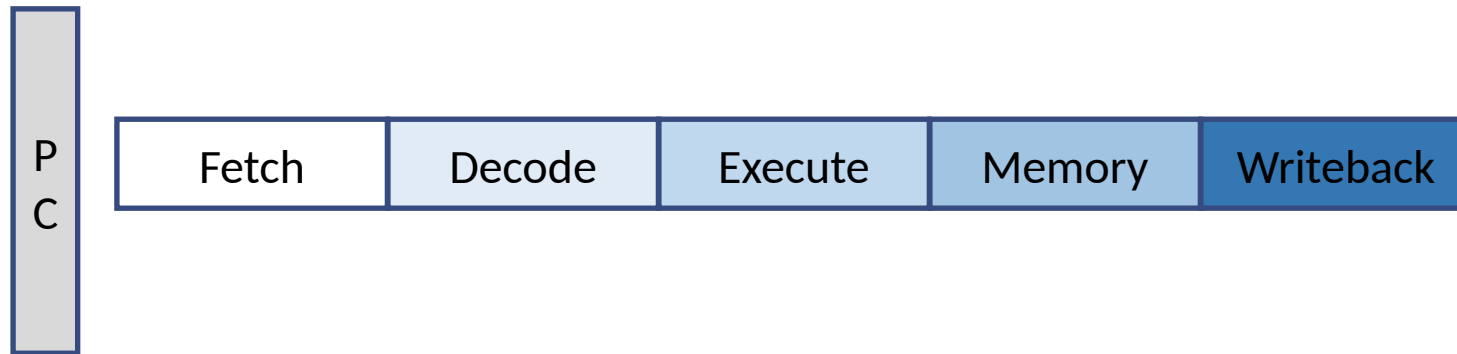
# Terminology we'll discuss later

- **Pipeline Register:**
  - Colloquially: the extra work we incur by pipelining.
  - In reality: a register in the processor that captures the set of signals (values) on which a stage will execute.
  - Each stage has its own pipeline register.
- **Hazard:**
  - The “problems” that we run into when we naively add pipelining

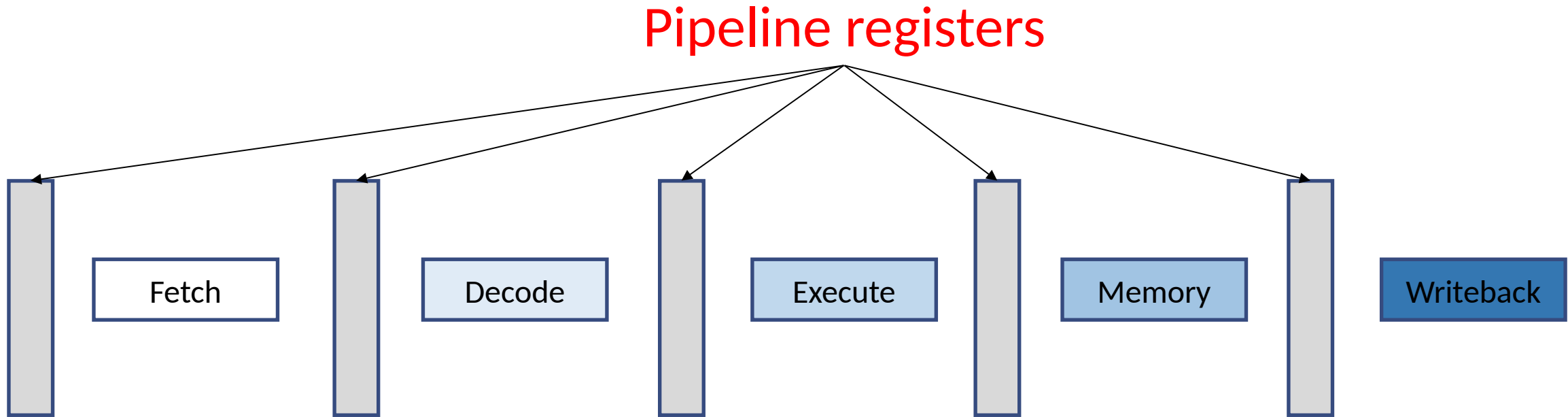
# Where we're going in this module

- **Today: Analyze pipeline timing**
- Next classes:
  - Find Hazards in the y86 pipelined implementation
  - Experiment with different ways to address **data hazards**
  - Address **control hazards** via **branch prediction**
  - Compare and contrast pipelining with other forms of parallelism that we might implement in hardware

# Before we had pipelining:



# With Pipelining

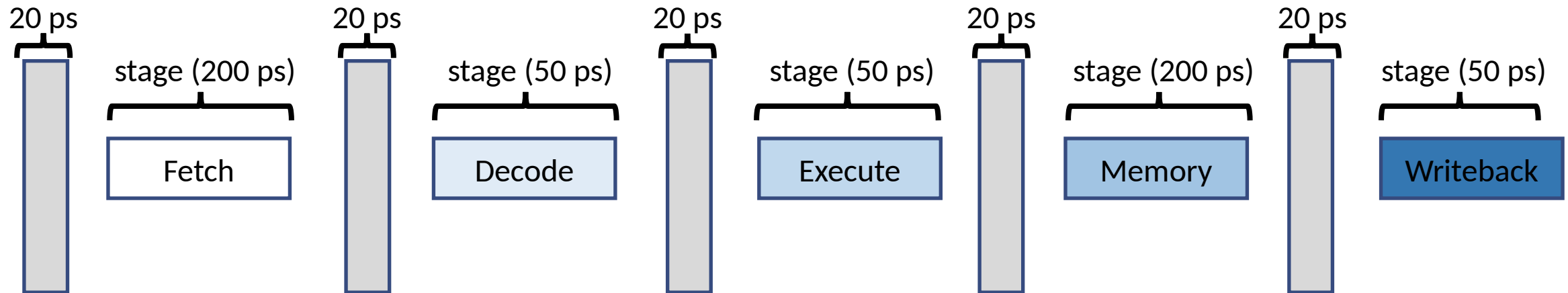


- Why do we need pipeline registers?
  - Each stage is working on a **different** instruction during the clock cycle.
  - We don't want one stage's output to affect the computations of the next stage within the same clock cycle.

# Performance: Latency

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds

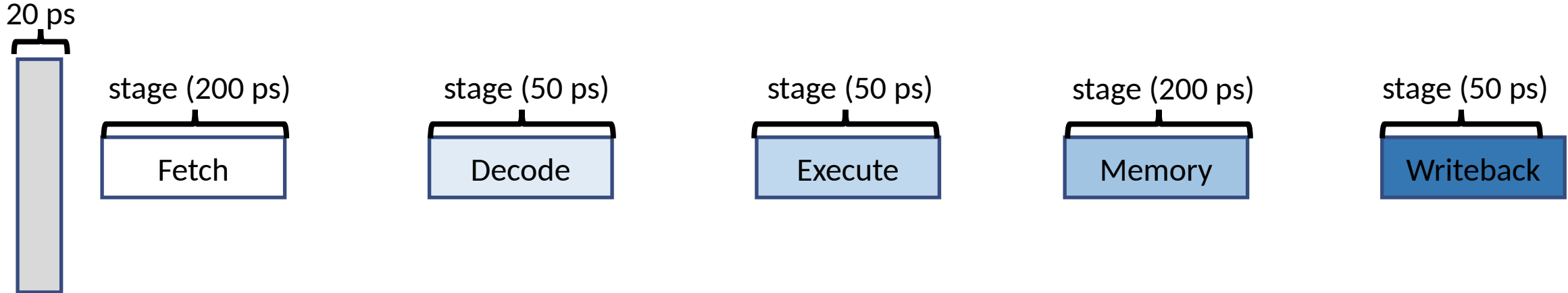


**Latency:** End to end time to complete execution of a single instruction.

# Performance: Latency (1)

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds

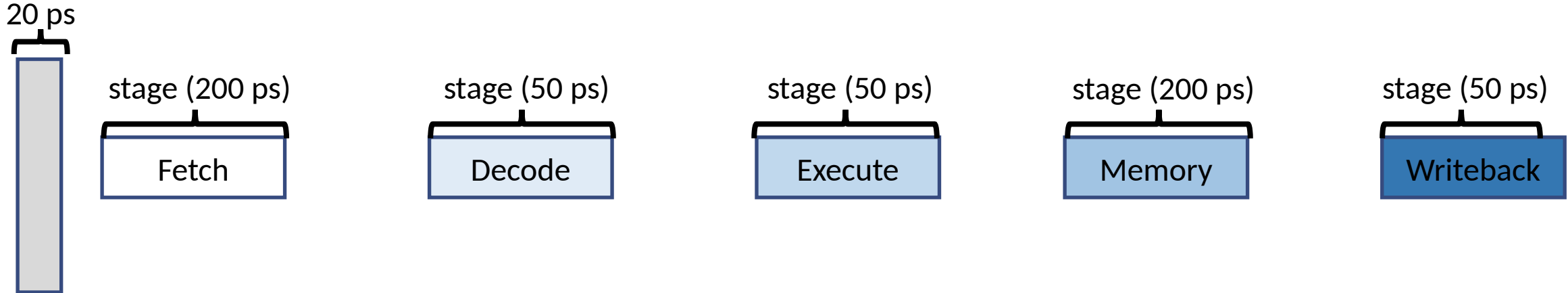


Q1: What is the **latency** of an instruction in an *UNPIPELINED* implementation?

# Performance: Latency (1)

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds



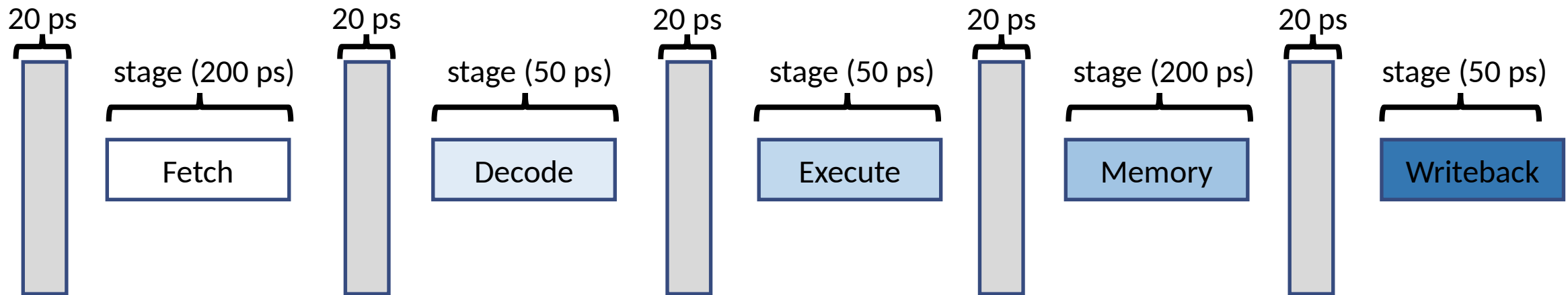
Q1: What is the **latency** of an instruction in an *UNPIPELINED* implementation?

$$20 + 200 + 50 + 50 + 200 + 50 = 570 \text{ ps}$$

# Performance: Latency (2)

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds



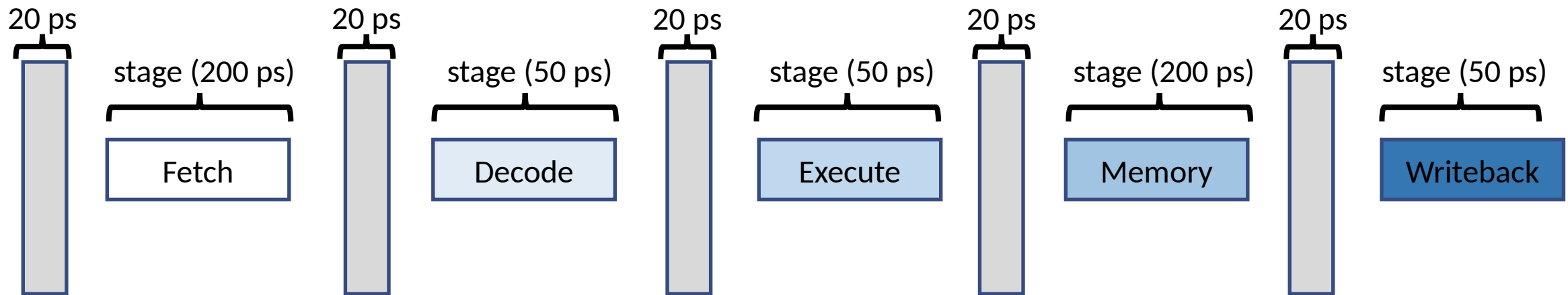
Q2: What is the **latency** of an instruction in a *PIPELINED* implementation?



# Performance: Latency (2)

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds



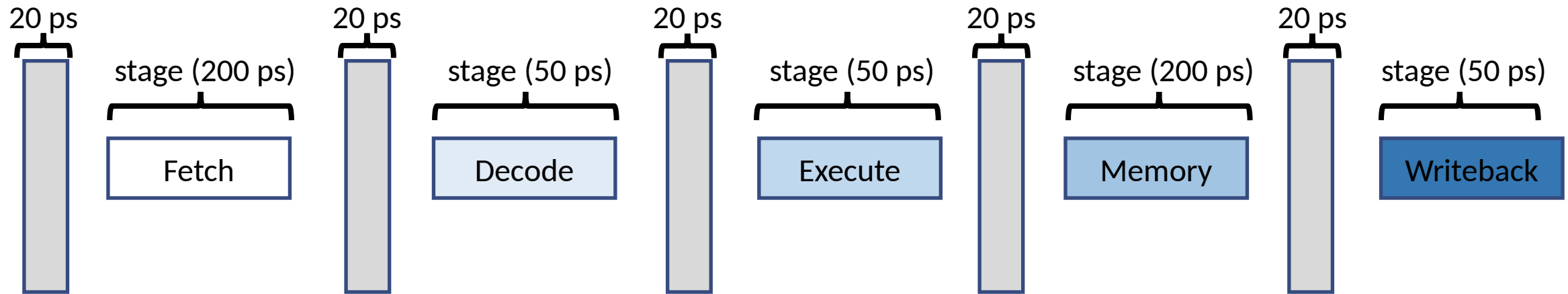
Q2: What is the **latency** of an instruction in a *PIPELINED* implementation?

$$200 * 5 + 20 * 5 = (200 + 20) * 5 = 1100 \text{ ps}$$

# Performance: Retirement Latency

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds

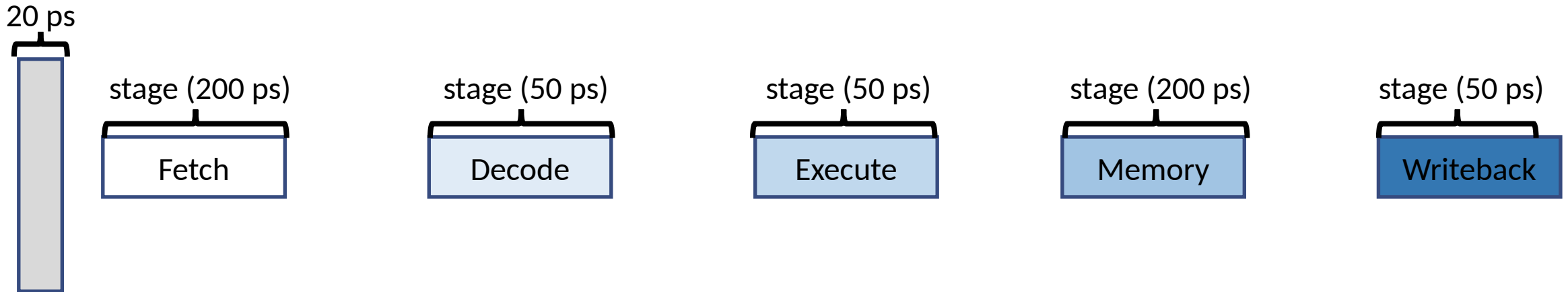


**Retirement Latency:** Time between the completion of one instruction and the completion of the next instruction.

# Performance: Retirement Latency (1)

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds

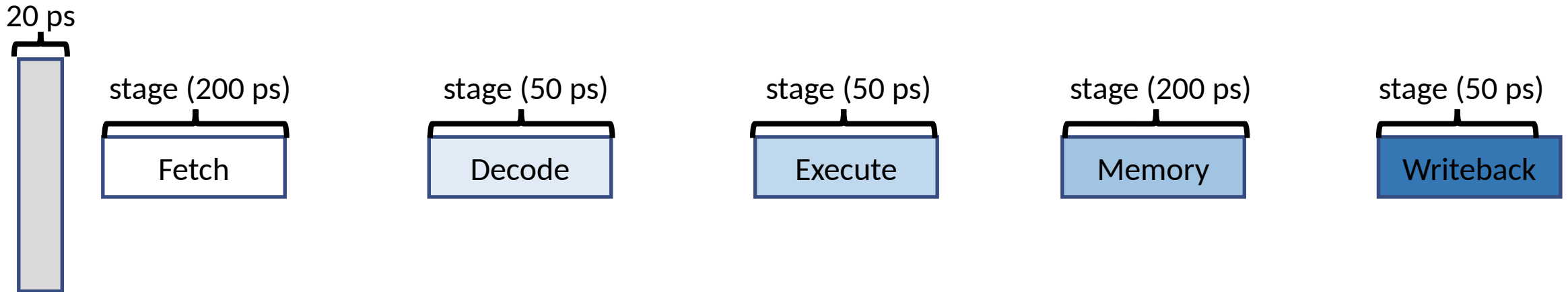


Q1: What is the **retirement latency** of an instruction in an *UNPIPELINED* implementation?

# Performance: Retirement Latency (1)

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds



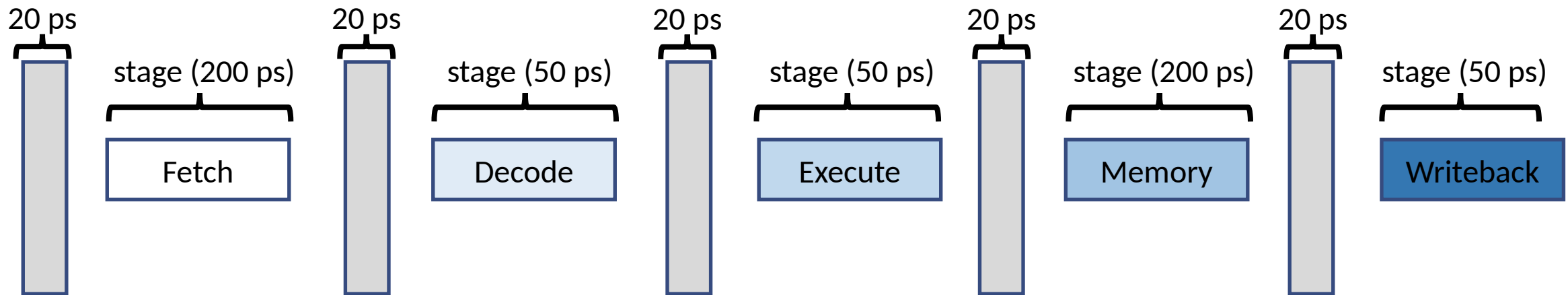
Q1: What is the **retirement latency** of an instruction in an *UNPIPELINED* implementation?

$$20 + 200 + 50 + 50 + 200 + 50 = 570 \text{ ps}$$

# Performance: Retirement Latency (2)

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds

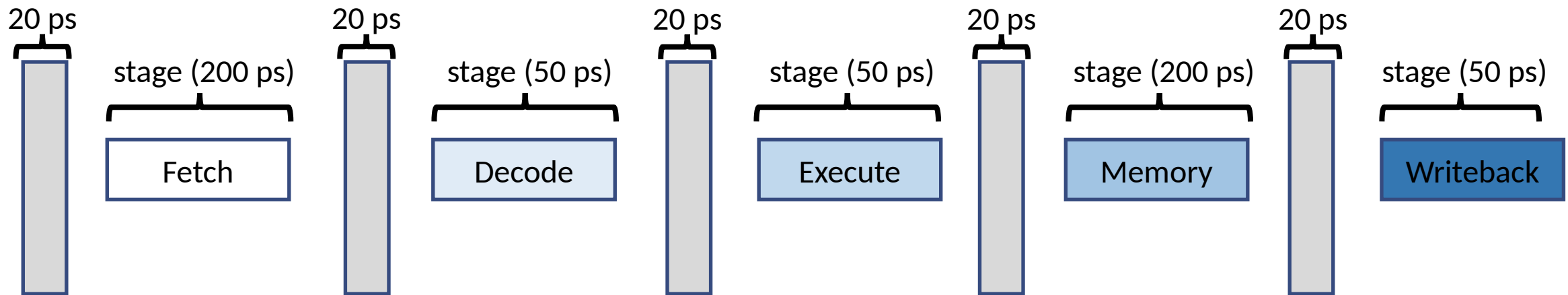


Q2: What is the **retirement latency** of an instruction in a *PIPELINED* implementation?

# Performance: Retirement Latency (2)

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds



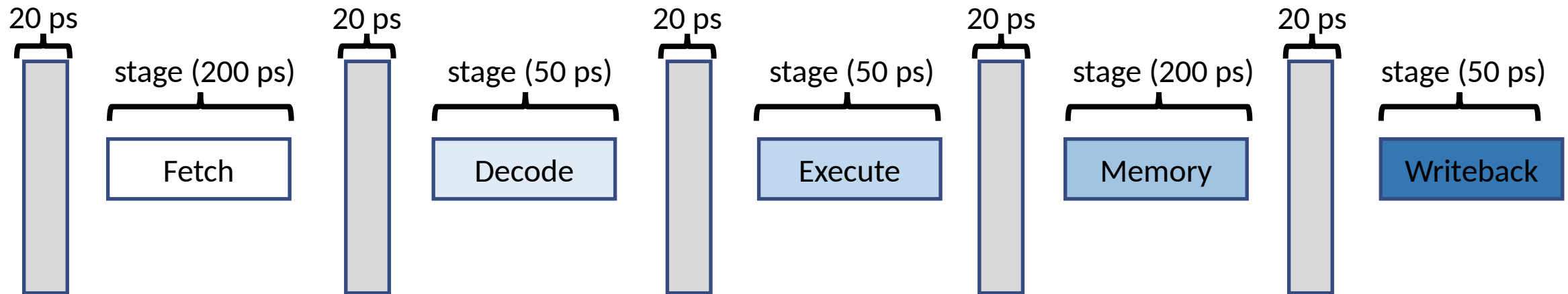
Q2: What is the **retirement latency** of an instruction in a *PIPELINED* implementation?

$$20 + 200 = 220 \text{ ps}$$

# Performance: Throughput

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds



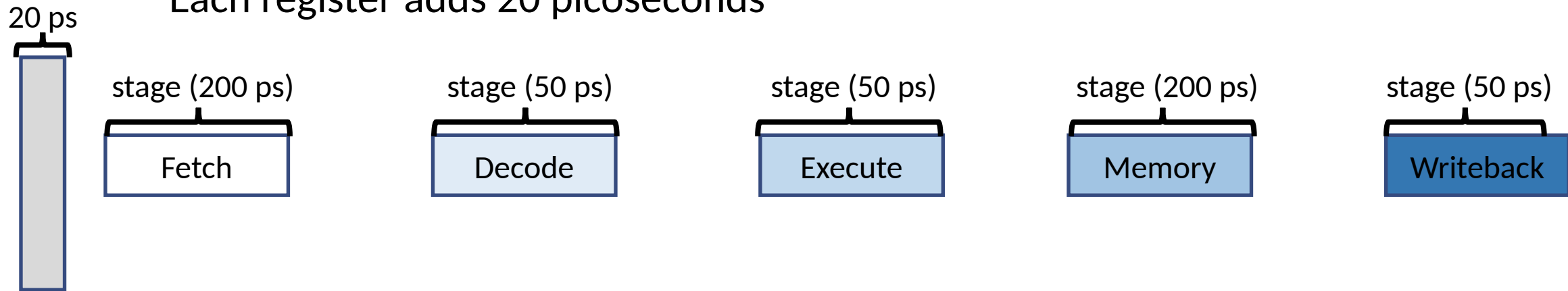
**Throughput:** The rate at which instructions complete. Typically expressed in GIPS: giga-instructions per second.

Note: We've computed picoseconds per instruction; throughput is instructions per second ...

# Performance: Throughput (1)

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds



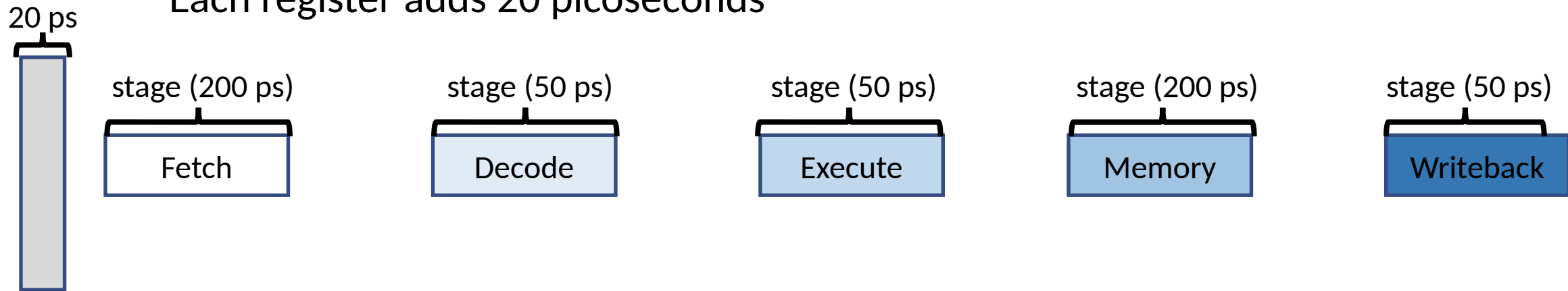
Q1: What is the **throughput** of an *UNPIPELINED* implementation?



# Performance: Throughput (1)

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds



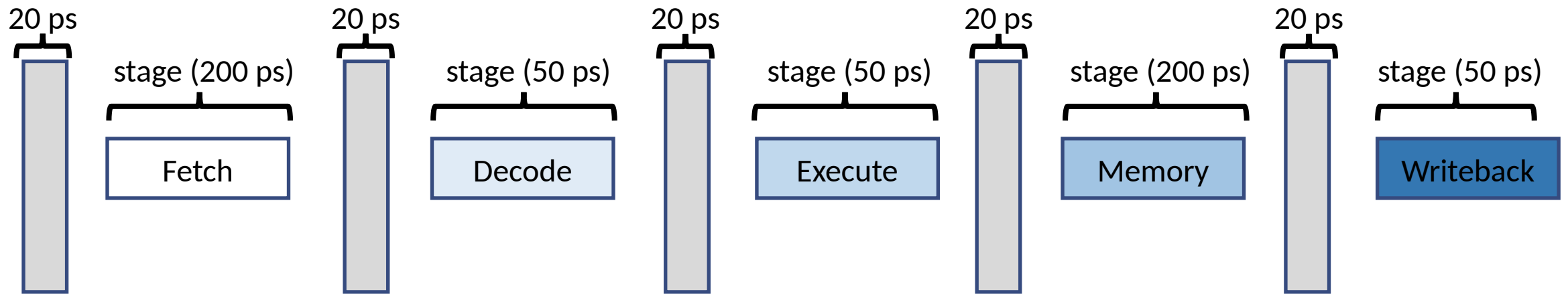
Q1: What is the **throughput** of an *UNPIPELINED* implementation?

$$\begin{aligned} &1 / 570 \text{ ps per instruction} = 1 / (570 \times 10^{-12}) \text{ s per instruction} \\ &(1 / 570) \times 10^{12} \text{ IPS} = (1000 / 570) \times 10^9 \text{ IPS} = \\ &(1000 / 570) \text{ GIPS} \approx 1.75 \text{ GIPS} \end{aligned}$$

# Performance: Throughput (2)

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds

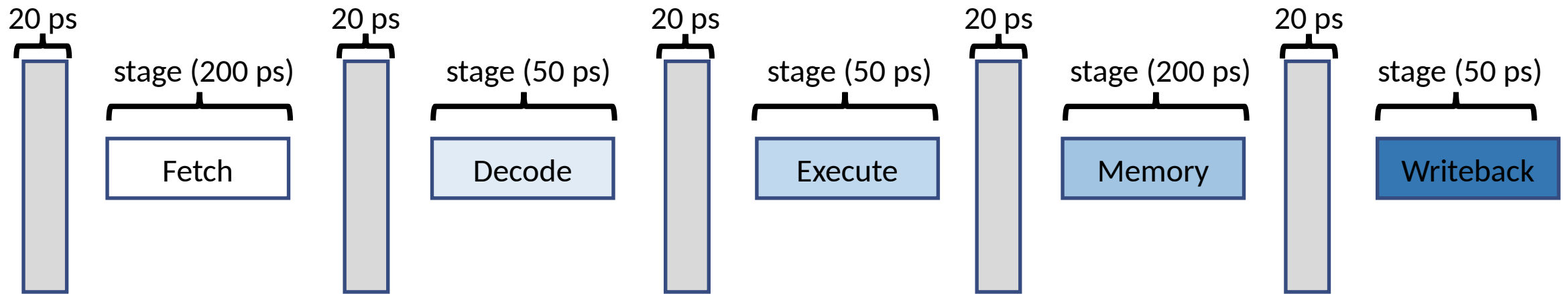


Q2: What is the **throughput** of a *PIPELINED* implementation?

# Performance: Throughput (2)

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds



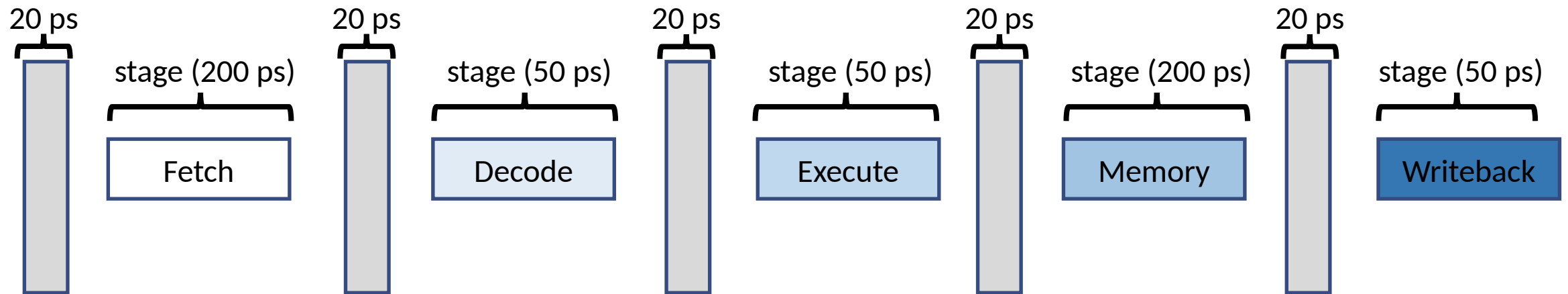
Q2: What is the **throughput** of a *PIPELINED* implementation?

$$\begin{aligned} &1 / 220 \text{ ps per instruction} = 1 / (220 \times 10^{-12}) \text{ s per instruction} \\ &(1 / 220) \times 10^{12} \text{ IPS} = (1000 / 220) \times 10^9 \text{ IPS} = \\ &(1000 / 220) \text{ GIPS} \approx 4.55 \text{ GIPS} \end{aligned}$$

# Performance: Throughput

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds



Q1: What is the **throughput** of an *UNPIPELINED* implementation?

**1.75 GIPS**

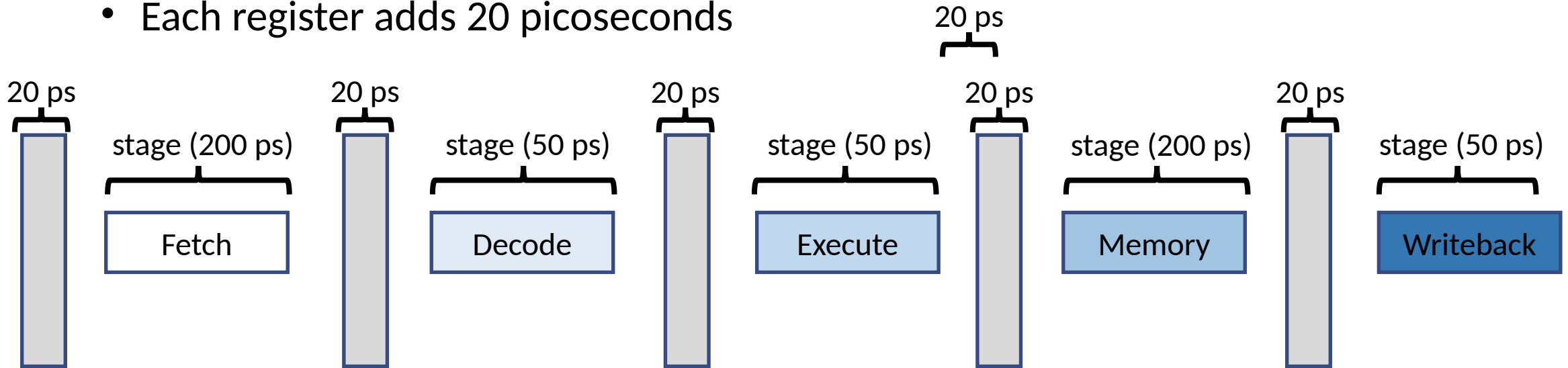
Q2: What is the **throughput** of a *PIPELINED* implementation?

**4.55 GIPS**

# How fast is our pipelined processor?

Assume that:

- Each stage needs either 50 or 200 picoseconds to execute
- Each register adds 20 picoseconds



We increased latency by 93% but multiplied throughput by a factor of 2.6

# Pipeline Issues

- For a pipelined processor, throughput is the inverse of the retirement latency.
- Why don't we just have thousands of really short stages to get maximum performance?

# Pipeline Issues

- For a pipelined processor, throughput is the inverse of the retirement latency.
- Why don't we just have thousands of really short stages to get maximum performance?
  - The hardware might not partition well into too many pieces.
  - Each stage introduces a pipeline register that will add latency (and take up space on the chip and consume power and...)
  - Dealing with hazards (next class) becomes more complicated as the number of stages increases.

# Pipeline length (depth) trivia (stackexchange.com)

Year	Microarchitecture	Pipeline stages
1993	P5 (Pentium)	5
1994	P6 (Pentium 3)	10
1995	P6 (Pentium Pro)	14
2001	NetBurst (Willamette)	20
2002	NetBurst (Northwood)	20
2004	NetBurst (Prescott)	31
2006	NetBurst (Cedar Mill)	31
2006	Core	14
	Bonnell	16
2011	Sandy Bridge	14 to 17
	Silvermont	14
2015	Haswell	14
2015	Skylake	14
2016	Kabylake	14



# In class activity

- PrairieLearn Inclass exercise 7 – Pipeline Performance

# Takeaways

- Pipelining
  - Introduces parallelism
  - Allows multiple instructions to be executing at the same time
  - **Increases** single instruction latency
  - **Decreases** retirement latency
  - **Increases** throughput