

**CPS104**  
**Computer Organization and Programming**  
**Lecture 19: Pipelining**

**Robert Wagner**

# Lecture Overview

- **A Pipelined Processor :**

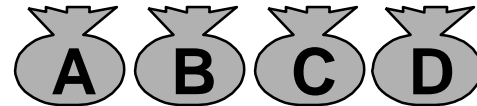
- ★ **Introduction to the concept of pipelined processor.**

**Reading: Chapters 5, 6**

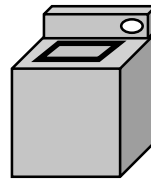
# Pipelining: Its Natural!

- Laundry Example

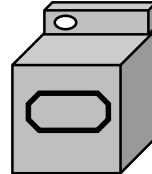
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold



- Washer takes 30 minutes



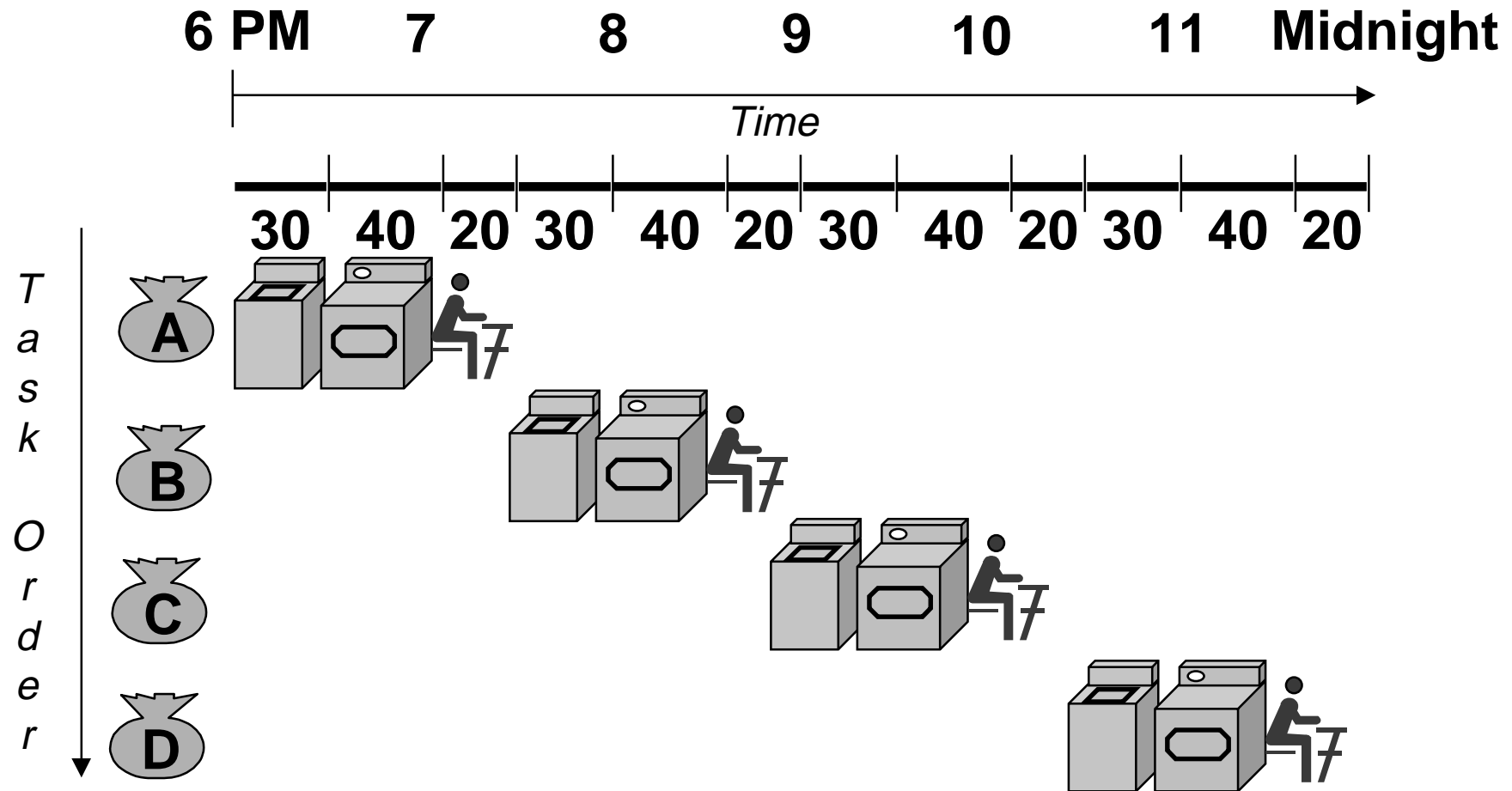
- Dryer takes 40 minutes



- “Folder” takes 20 minutes

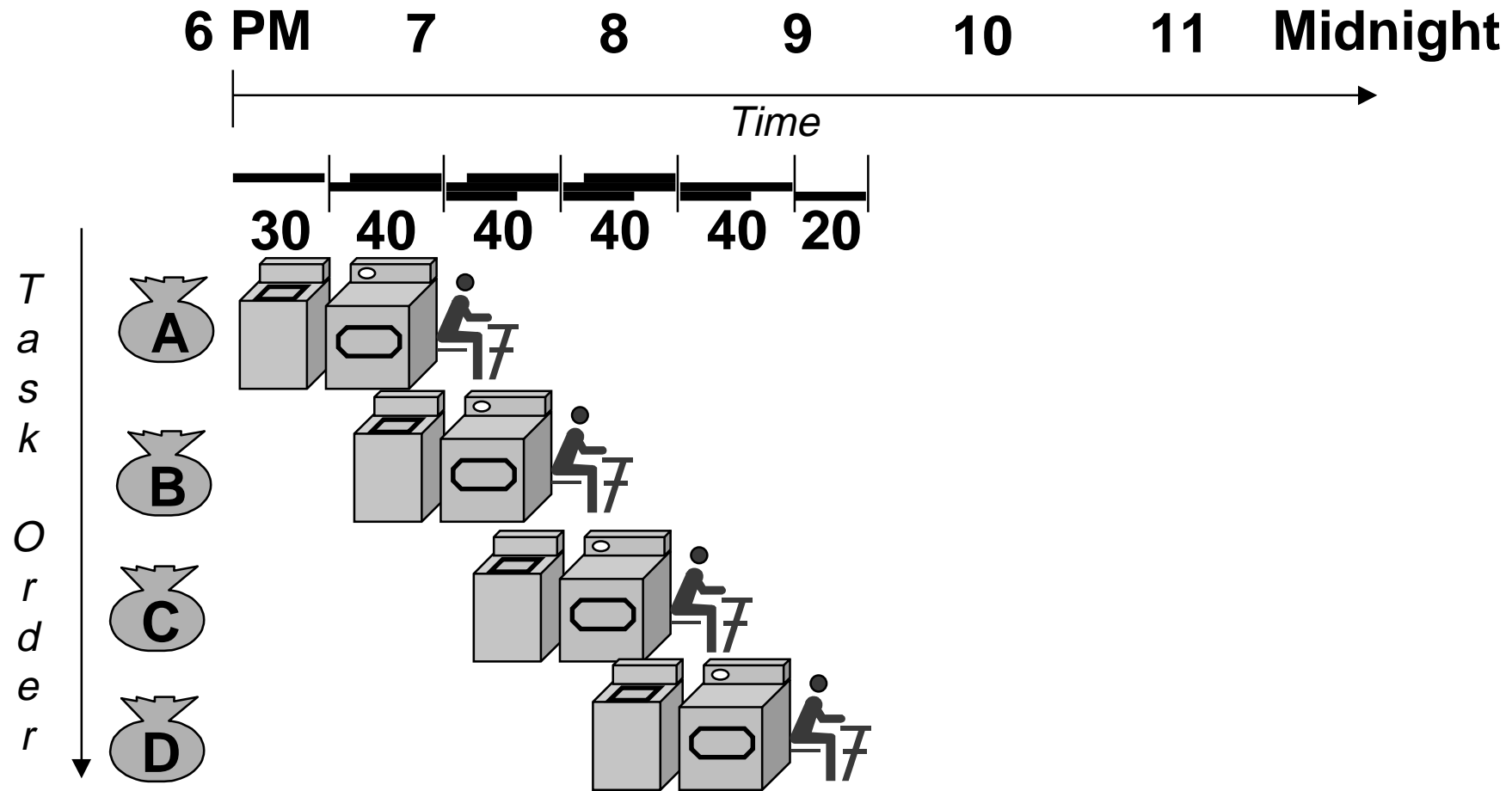


# Sequential Laundry



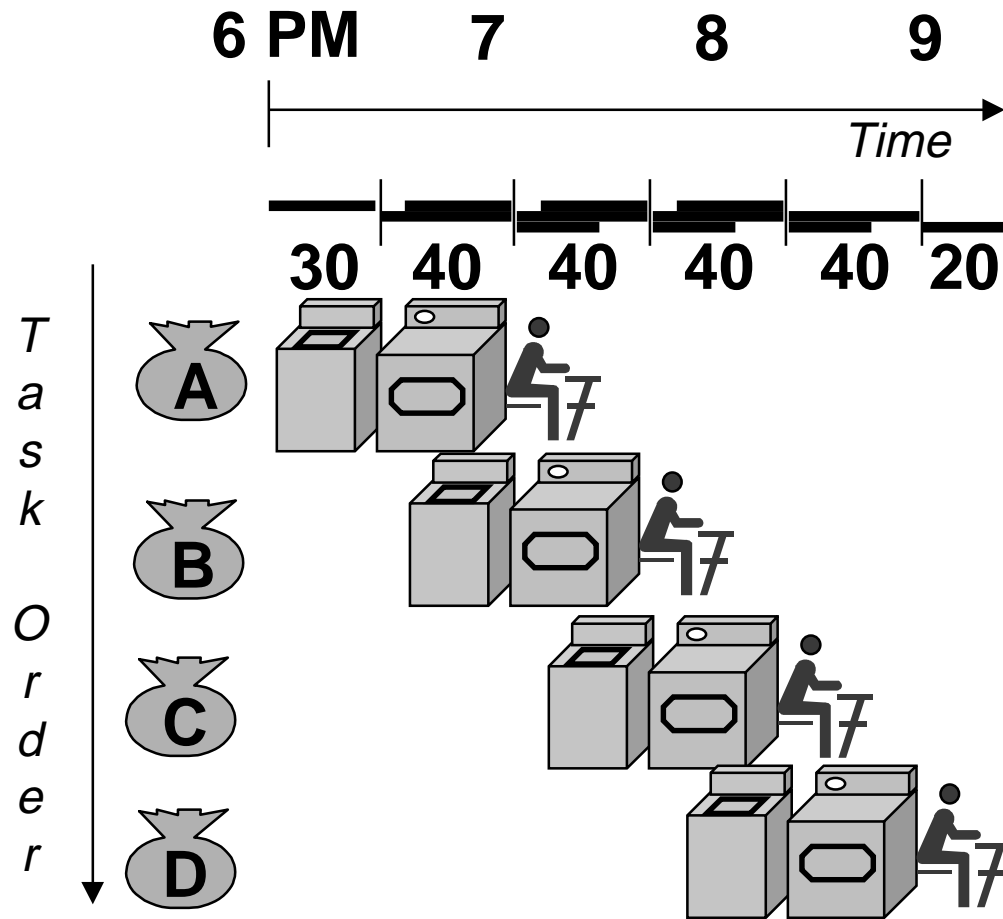
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

# Pipelined Laundry: Start work ASAP



● Pipelined laundry takes 3.5 hours for 4 loads

# Pipelining Lessons

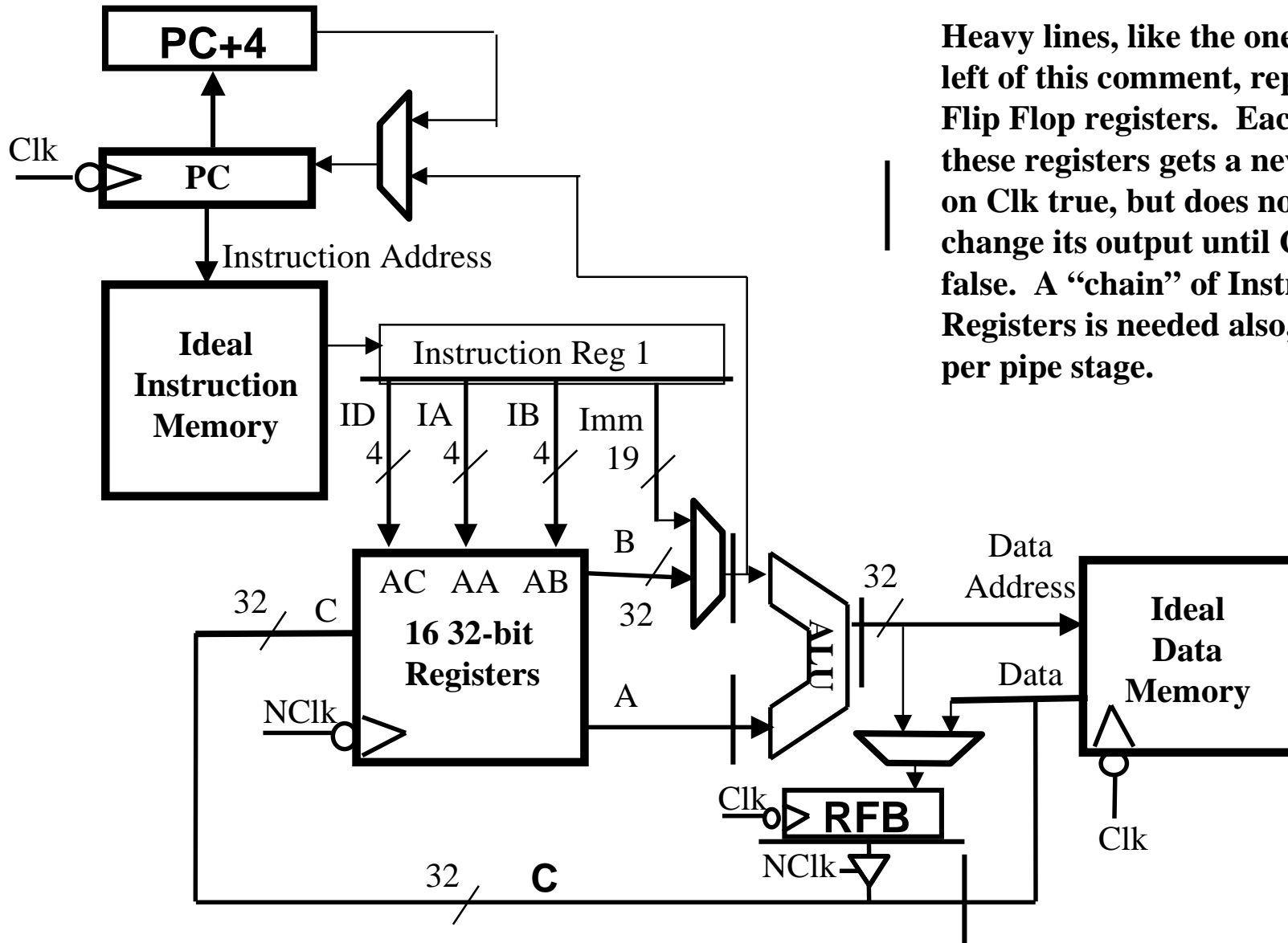


- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

# Overview of a Multiple Cycle Implementation

- **The root of the single cycle processor's problems:**
  - ★ **The cycle time has to be long enough for the slowest instruction**
- **Solution:**
  - ★ **Break the instruction into smaller steps**
  - ★ **Execute each step (instead of the entire instruction) in one cycle**
    - ➔ **Cycle time: time it takes to execute the longest step**
    - ➔ **Make all steps have similar length**
  - ★ **This is the essence of the multiple cycle processor**
- **The advantages of the multiple cycle processor:**
  - ★ **Cycle time is much shorter**
  - ★ **Different instructions take different number of cycles to complete**
    - ➔ **Load takes five cycles**
    - ➔ **Jump only takes three cycles**
  - ★ **Allows a functional unit to be used more than once per instruction**

# Data Path



Heavy lines, like the one to the left of this comment, represent Flip Flop registers. Each of these registers gets a new value on Clk true, but does not change its output until Clk goes false. A “chain” of Instruction Registers is needed also, one per pipe stage.

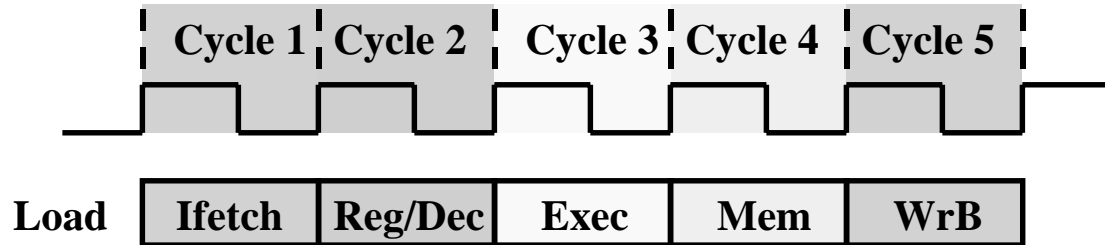
# Pipeline Problem: Using same hardware component again

- Suppose we want to pipeline at 1 Instruction per cycle, BUT we design our processor so an instruction uses one component (like memory) twice during each instruction:
  - ★ Then when the pipeline fills, several instructions will try to use that component at the same time:

		CYCLE						
		1	2	3	4	5	6	7
I N S T	I 1	A	B	A				
	I 2		A	B	A			
	I 3					A	B	A
	I 4						A	B

- ★ During Cycle 3, when instruction 3 wants to start, Component A is in use, so instruction 3 must be delayed until cycle 5
- ★ 4 instructions complete in 8 cycles, not in 4 as desired.

# The Five Stages of Load

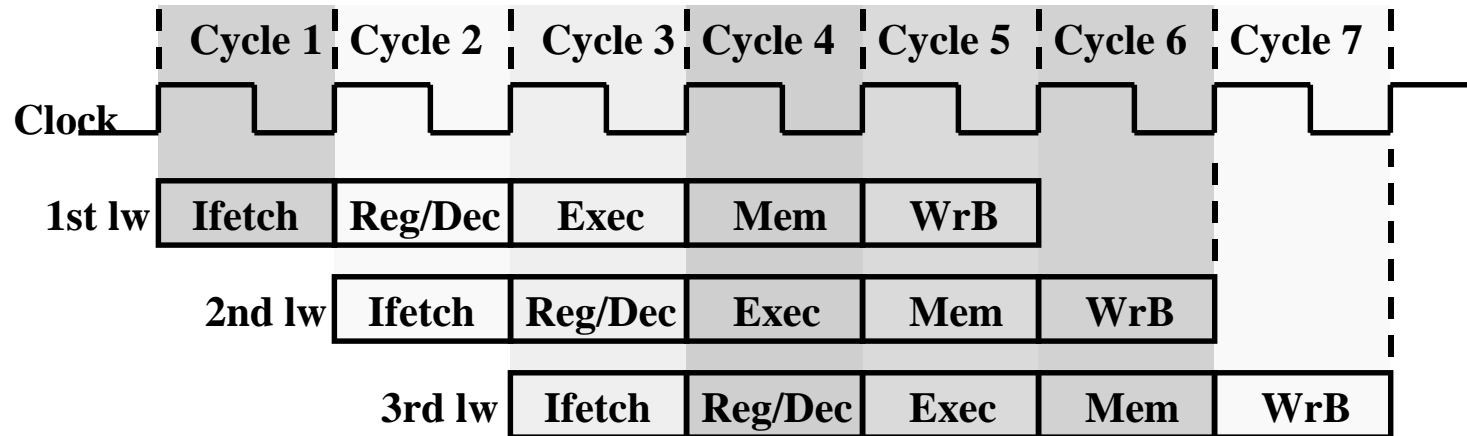


- **Ifetch: Instruction Fetch**
  - ★ Fetch the instruction from the Instruction Memory
- **Reg/Dec: Register Fetch and Instruction Decode**
- **Exec: Calculate the memory address**
- **Mem: Read the data from the Data Memory**
- **WrB: Write the data back to the register file**

# Key Ideas Behind Instruction Execution Pipelining

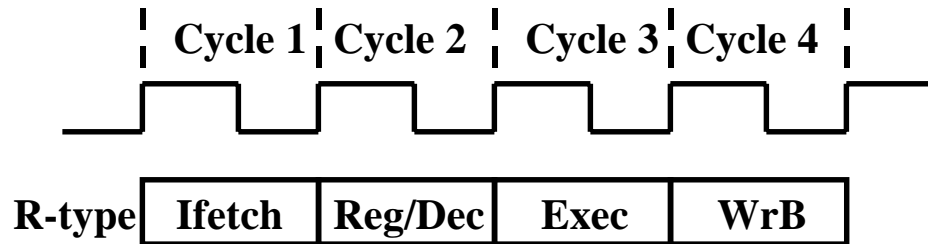
- **The load instruction has 5 stages: I-fetch, Reg- Fetch / I-Decode, Execute, Memory-Access, Register Write-Back.**
  - ★ **Five independent functional units to work on each stage**
    - **Each functional unit is used only once**
  - ★ **The 2nd load can start as soon as the 1st finishes its I-fetch stage**
  - ★ **Each load still takes five cycles to complete. latency is still 5 cycles**
  - ★ **The throughput is much higher; CPI is 1 with  $\sim 1/5$  cycle time.**
  - ★ **instructions start before the previous ones are completed.**

# Pipelining the Load Instruction



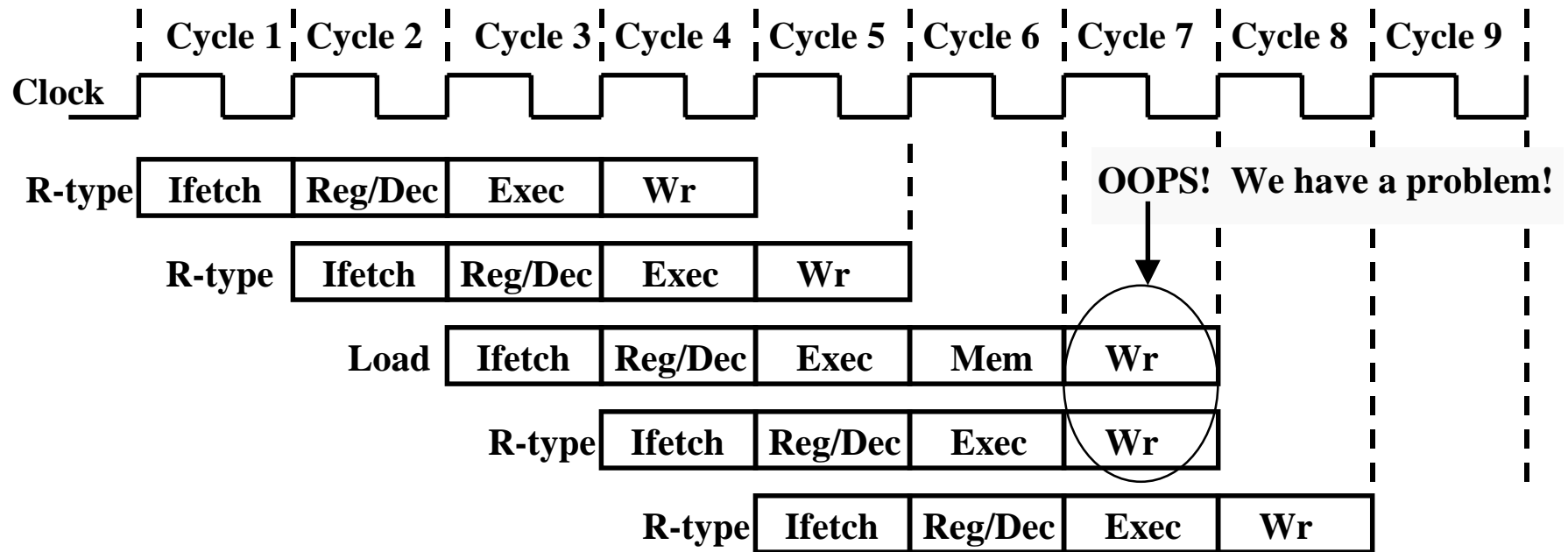
- The five independent functional units in the pipeline datapath are:
  - ★ Instruction Memory for the Ifetch stage
  - ★ Register File's Read ports (bus A and bus B) for the Reg/Dec stage
  - ★ ALU for the Exec stage
  - ★ Data Memory for the Mem stage
  - ★ Register File's Write port (bus W) for the WrB stage
- One instruction enters the pipeline every cycle
  - ★ One instruction comes out of the pipeline (completed) every cycle
  - ★ The "Effective" Cycles per Instruction (CPI) is 1;  $\sim 1/5$  cycle time

# The Four Stages of R-type



- **Ifetch: Instruction Fetch**
  - ★ Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec: ALU operates on the two register operands**
- **WrB: Write the ALU output back to the register file**

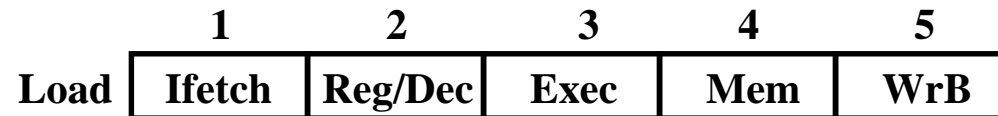
# Pipelining the R-type and Load Instruction



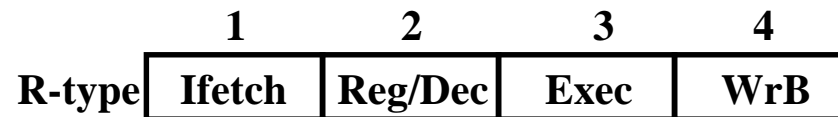
- We have a problem called a “pipeline conflict” or “hazard”:
  - ★ Two instructions try to write to the register file at the same time!

# Important Observation

- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage in all instructions:
  - ★ Load uses Register File's Write Port during its 5th stage



- ★ R-type uses Register File's Write Port during its 4th stage



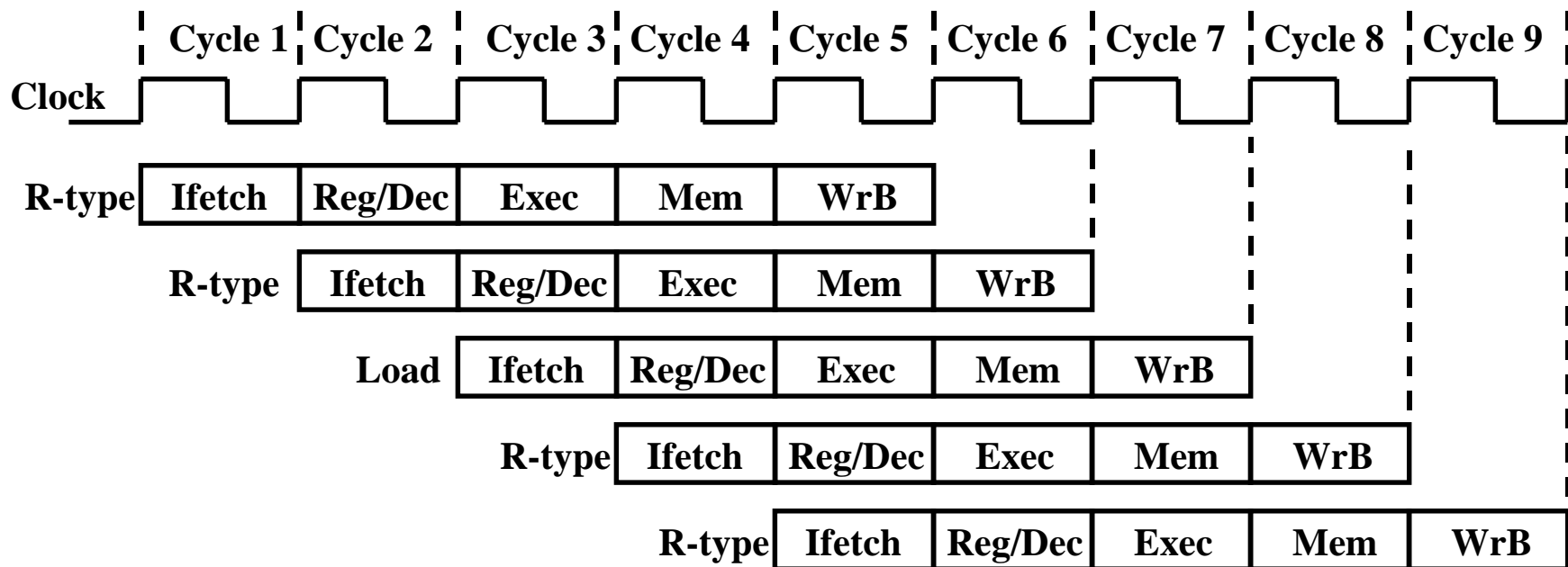
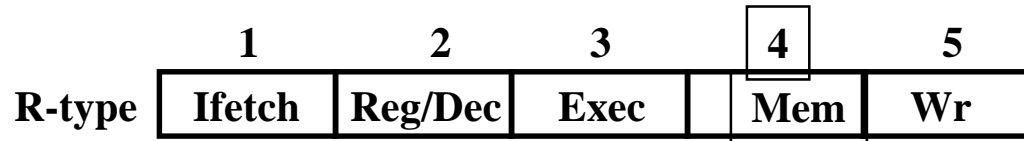
- How to solve this pipeline hazard?

# Solution: Delay R-type's Write by One Cycle

- Delay R-type's register write by one cycle:

- ★ Now R-type instructions also use Reg File's write port at Stage 5

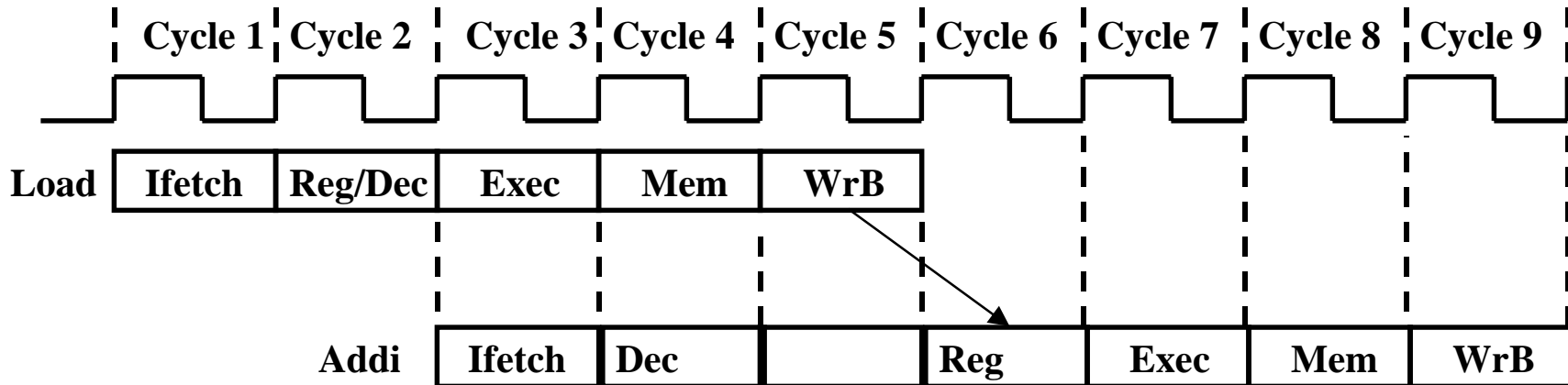
- ★ Mem stage is a NO-OP stage: nothing is being done. Effective CPI?



# Pipelining Problem: Data Availability Time

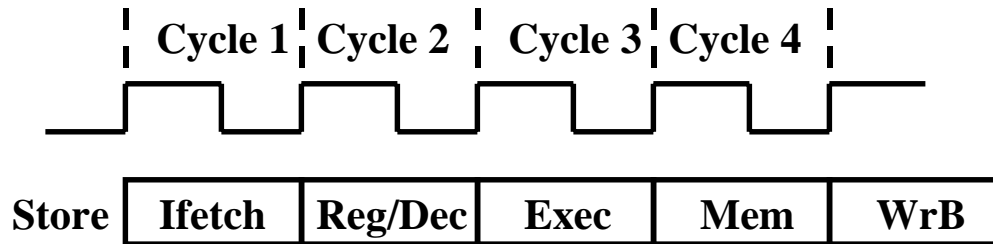
- What if a Load Instruction which sets register 5 is immediately followed by an Arithmetic (R-Type) instruction that USES register 5:

★ lw \$5, A  
 addi \$6,\$5,1



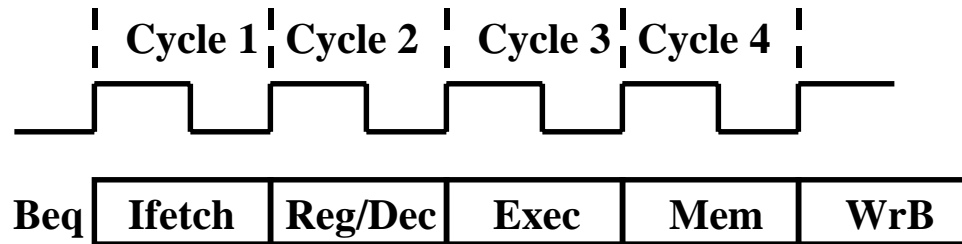
- Data needed by addi is not in register file until after WrB step of Load.
- “addi” Register Fetch step must be delayed until the data is ready.
- Optimization: The data could be “forwarded” from the Mem step of Load directly to the Exec step of “addi”, saving the delay slots

# The Four Stages of Store



- **Ifetch: Instruction Fetch**
  - ★ Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec: Calculate the memory address**
- **Mem: Write the data into the Data Memory**

# The Four Stages of Beq



- **Ifetch: Instruction Fetch**
  - ★ Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec: ALU compares the two register operands**
  - ★ Adder calculates the branch target address
- **Mem: If the registers we compared in the Exec stage are the same,**
  - ★ Write the branch target address into the PC