

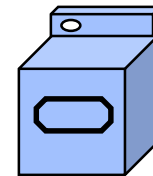
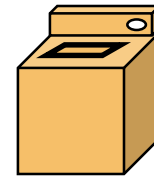
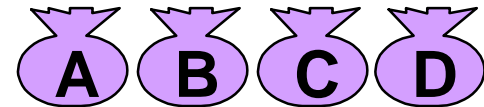
CSE331 – Computer Organization

Lecture 9: An Overview of Pipelining

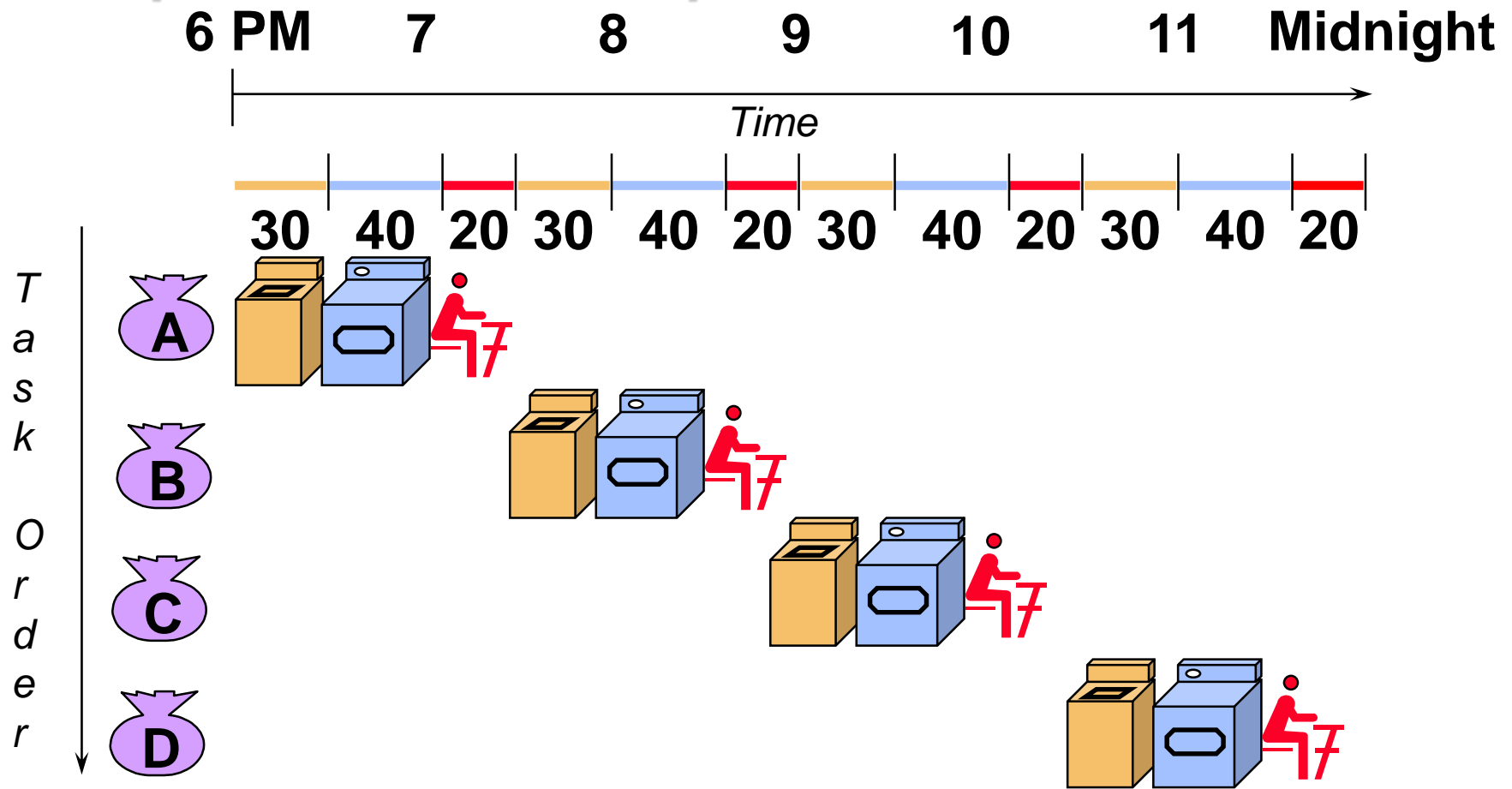
Pipelining

- ▶ Pipelining provides a method for executing multiple instructions at the same time.

- **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

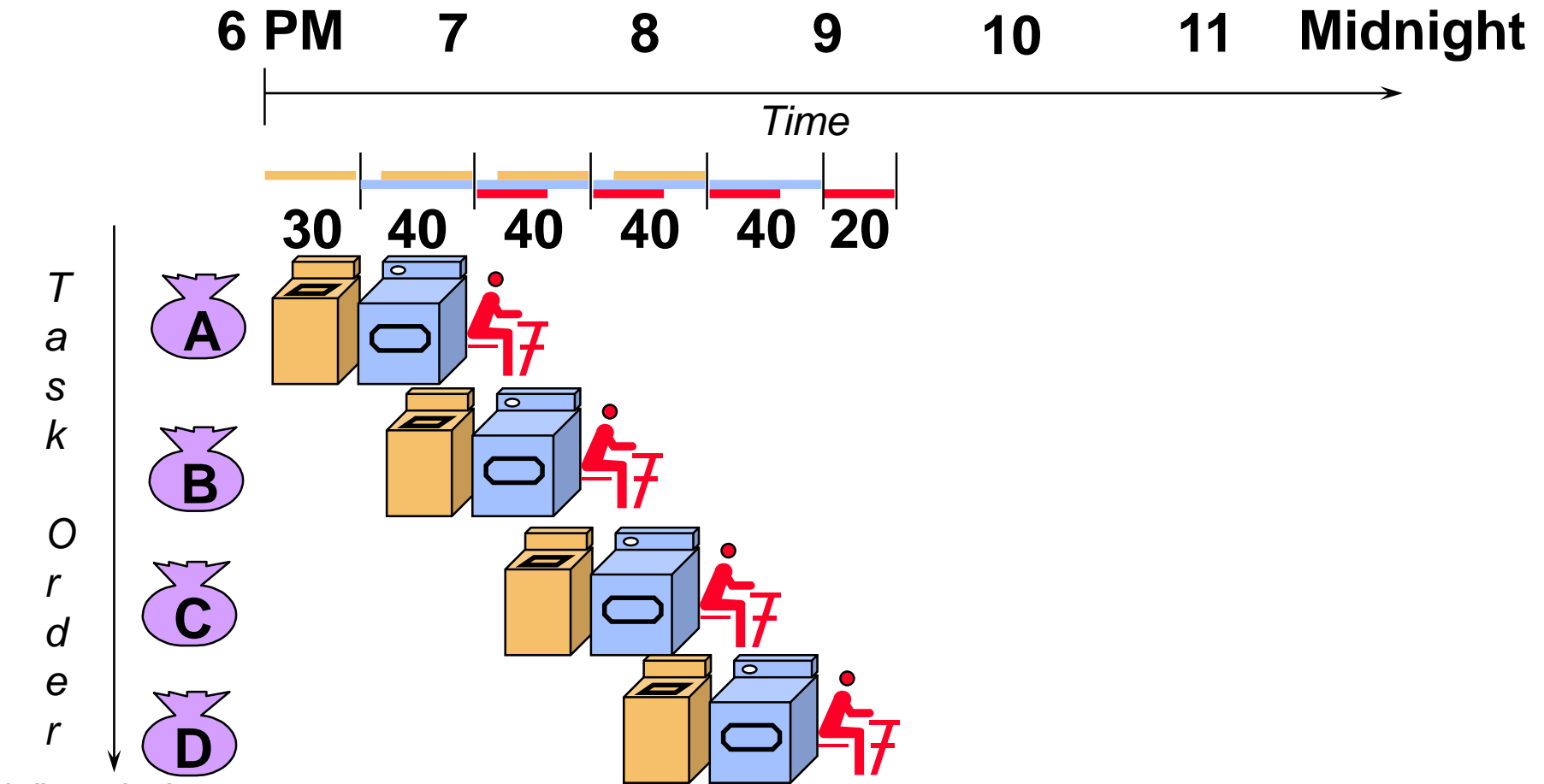


Sequential Laundry



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

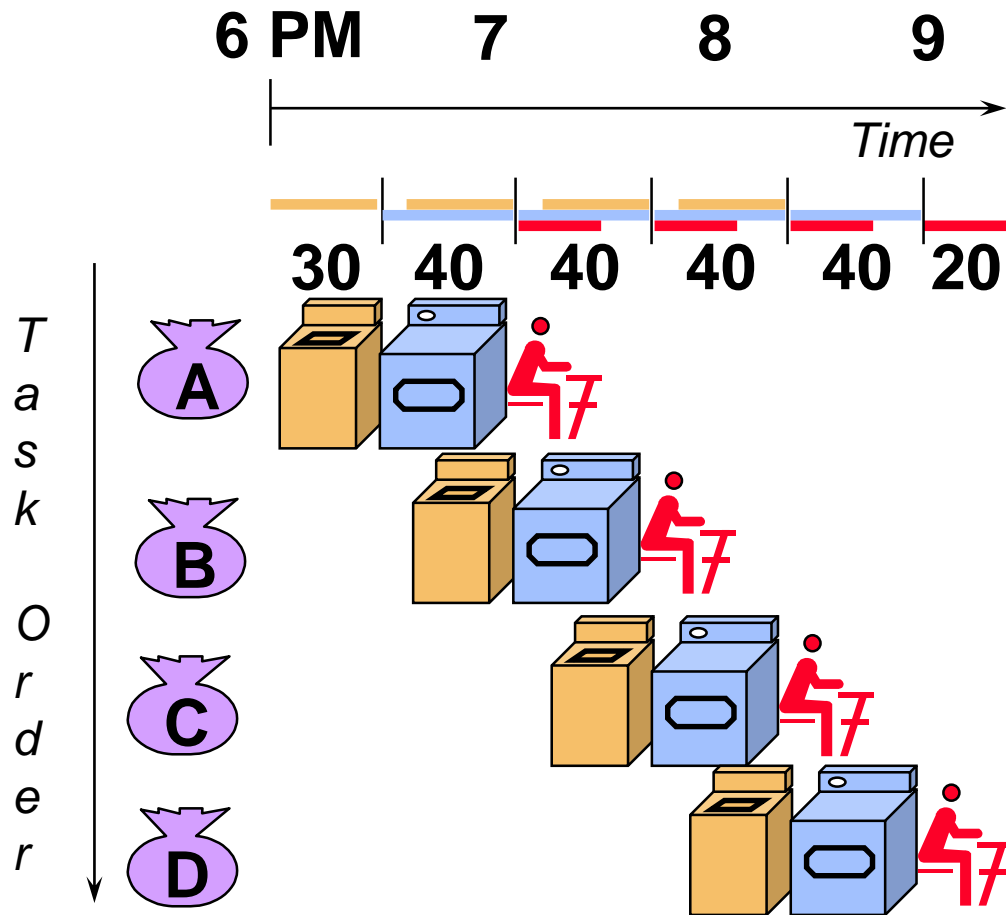
Pipelined Laundry: Start work ASAP



kullanıcı data'sını taşıyorsa aynı anda farklı datalar taşınabilir. (datapathı parçalara bölme)

► 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 using different resources
- ▶ Potential speedup = **Number pipe stages**
- ▶ Unbalanced lengths of pipe stages reduces speedup
- ▶ Time to “**fill**” pipeline and time to “**drain**” it reduces speedup
- ▶ Stall for Dependences

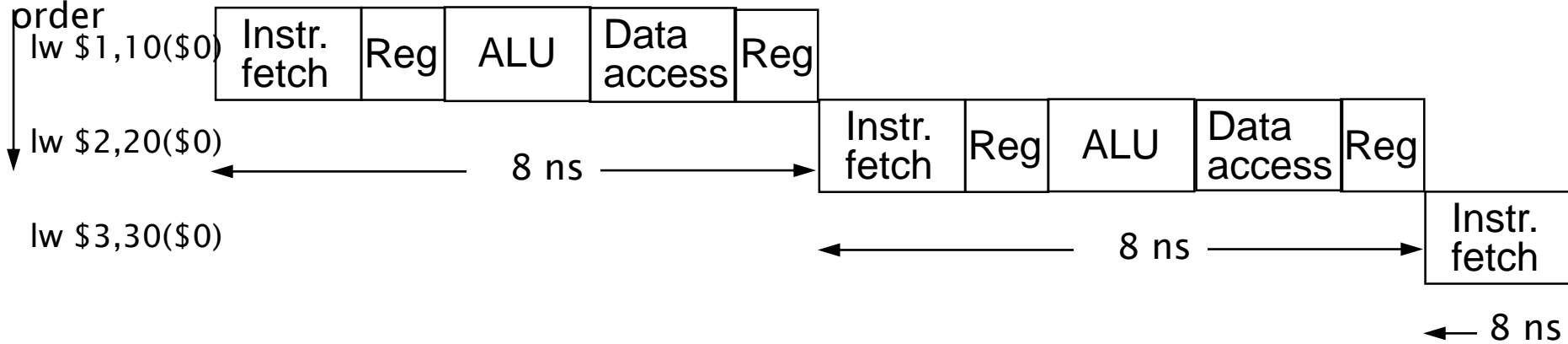
Total Time for Eight Instructions

Instr. class	Instr. fetch	Register read	ALU operation	Data access	Register write	Total time
Load word	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
Store word	2 ns	1 ns	2 ns	2 ns		7 ns
R-type	2 ns	1 ns	2 ns		1 ns	6 ns
Branch	2 ns	1 ns	2 ns			5 ns

- R-type instructions: add, sub, and, or, slt

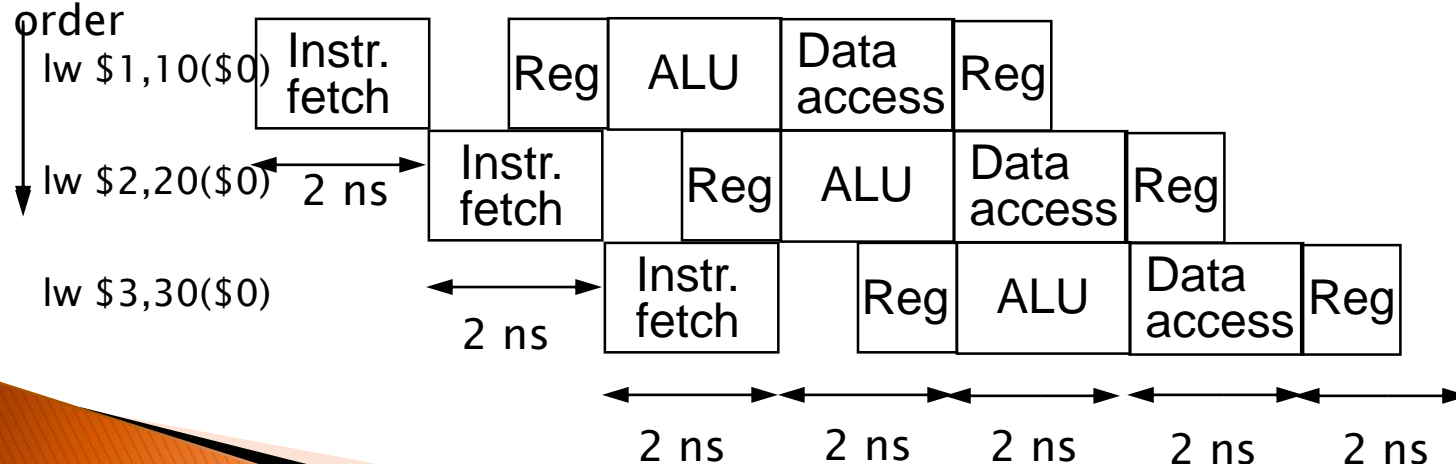
Single Cycle vs. Pipelined Execution

Program execution order



Total: 24 ns

Program execution order



Total: 14 ns

Pipelining Speedup

- ▶ If the stages are perfectly balanced:

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipeline stages}}$$

- ▶ Potential speedup = Number of pipeline stages
- ▶ In previous example, 3 instructions takes 14 ns.
If we would add 1000 instructions then each instruction will add 2 ns to the total execution time:

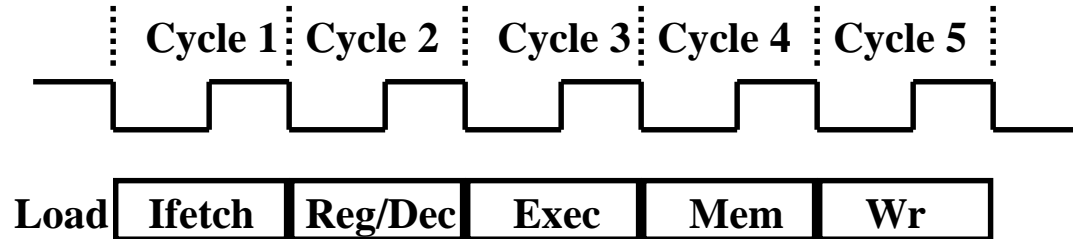
$$\text{Total execution time}_{\text{pipelined}} = 14 + 2000 = 2014 \text{ ns}$$

$$\text{Total execution time}_{\text{nonpipelined}} = 1003 * 8 = 8024 \text{ ns}$$

$$8024 / 2014 = 3.98 \sim 8 / 2$$

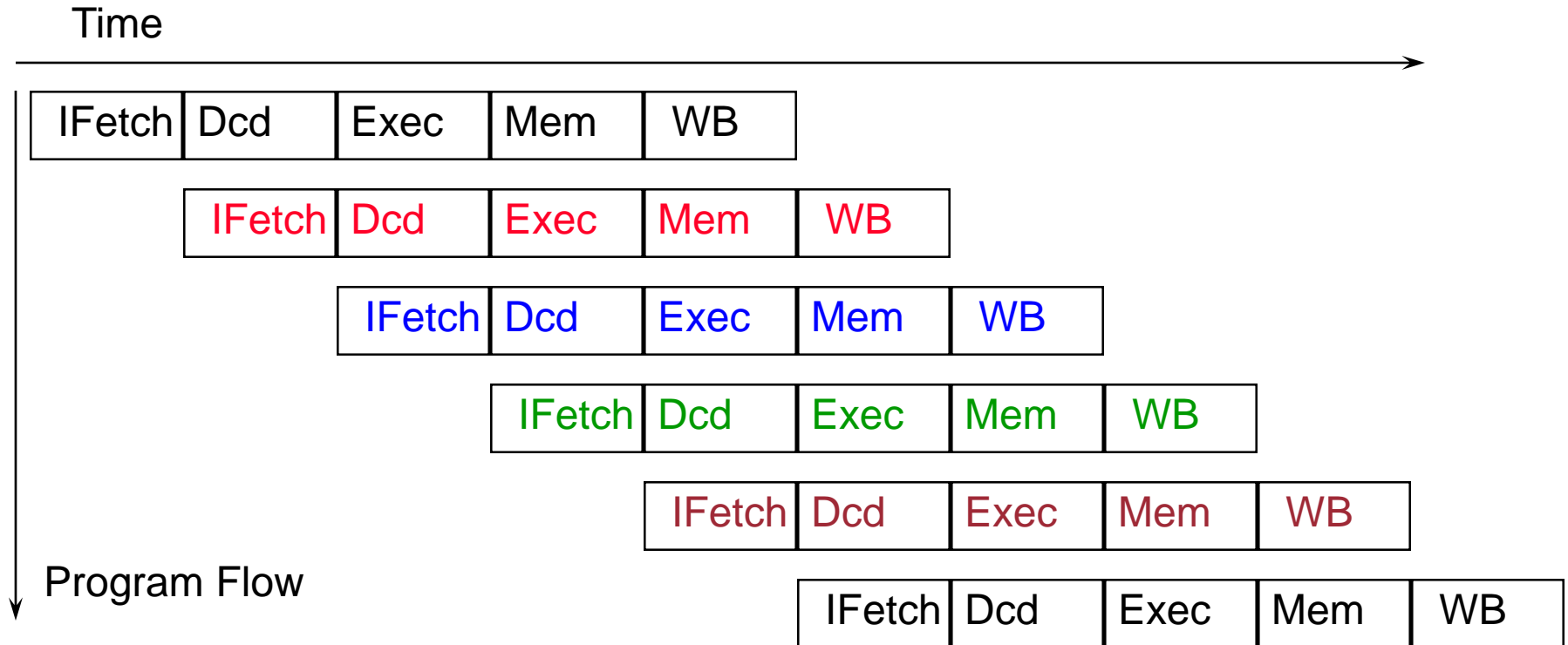
=

The Five Stages of the Load Instruction



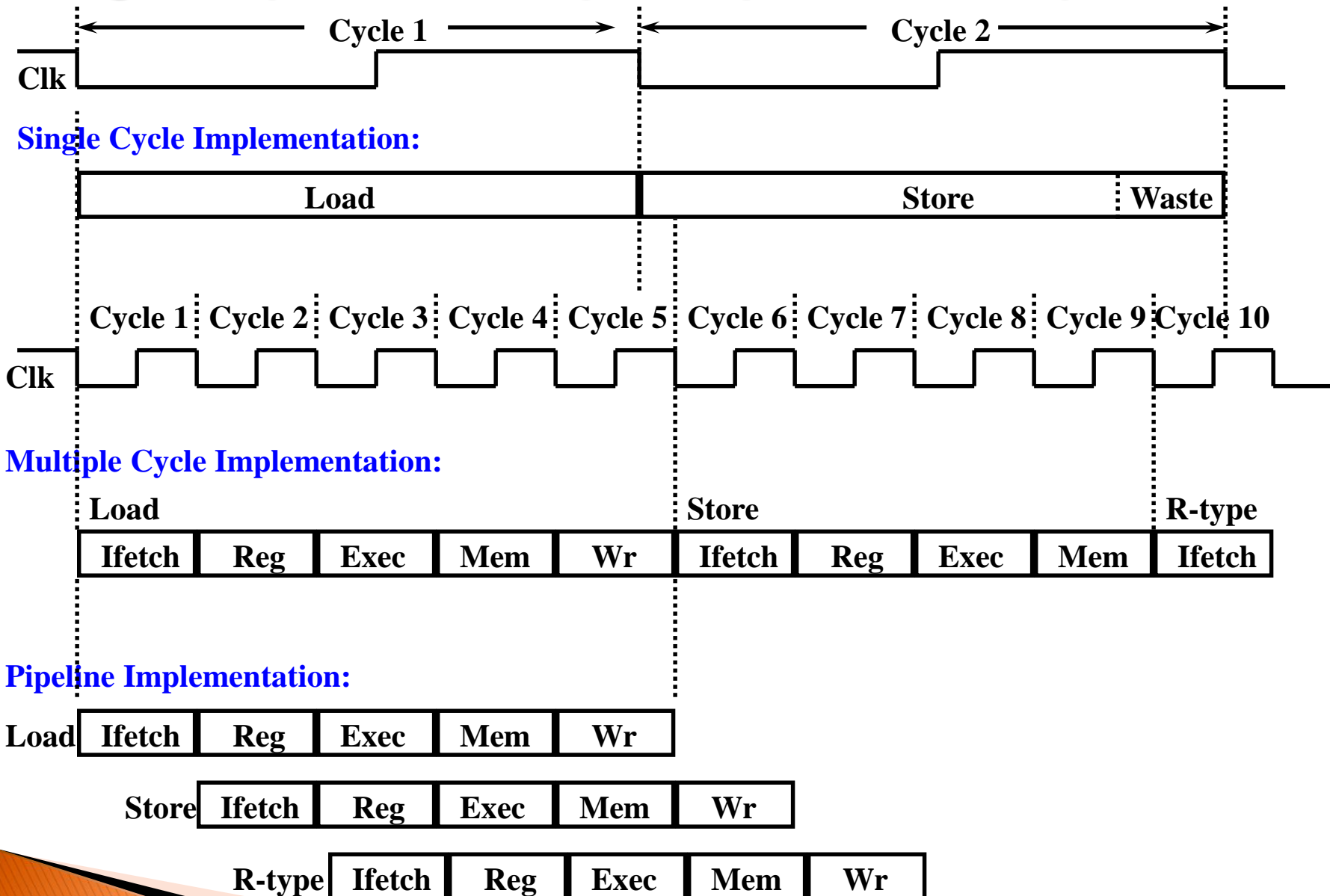
- ▶ **Ifetch:** Instruction Fetch
 - Fetch the instruction from the Instruction Memory
- ▶ **Reg/Dec:** Registers Fetch and Instruction Decode
- ▶ **Exec:** Calculate the memory address
- ▶ **Mem:** Read the data from the Data Memory
- ▶ **Wr:** Write the data back to the register file

Pipelined Execution

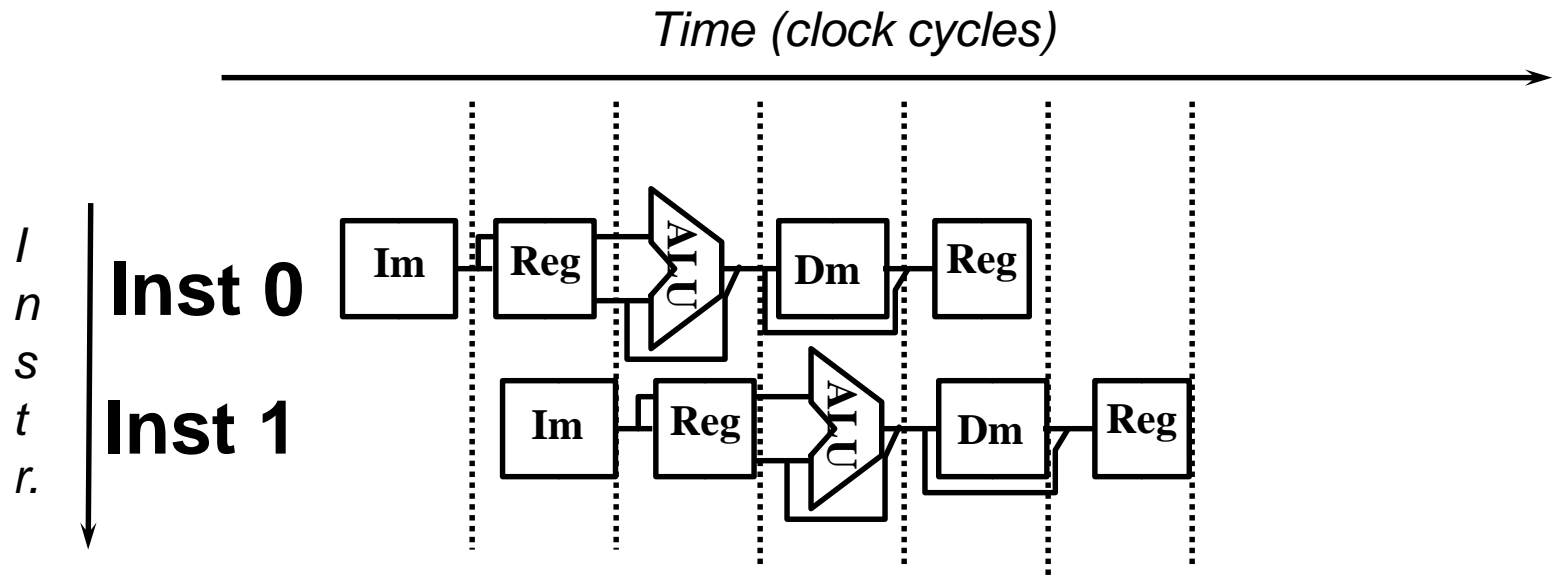


- ▶ On a processor multiple instructions are in various stages at the same time.
- ▶ Assume each instruction takes five cycles

Single Cycle, Multiple Cycle, vs. Pipeline

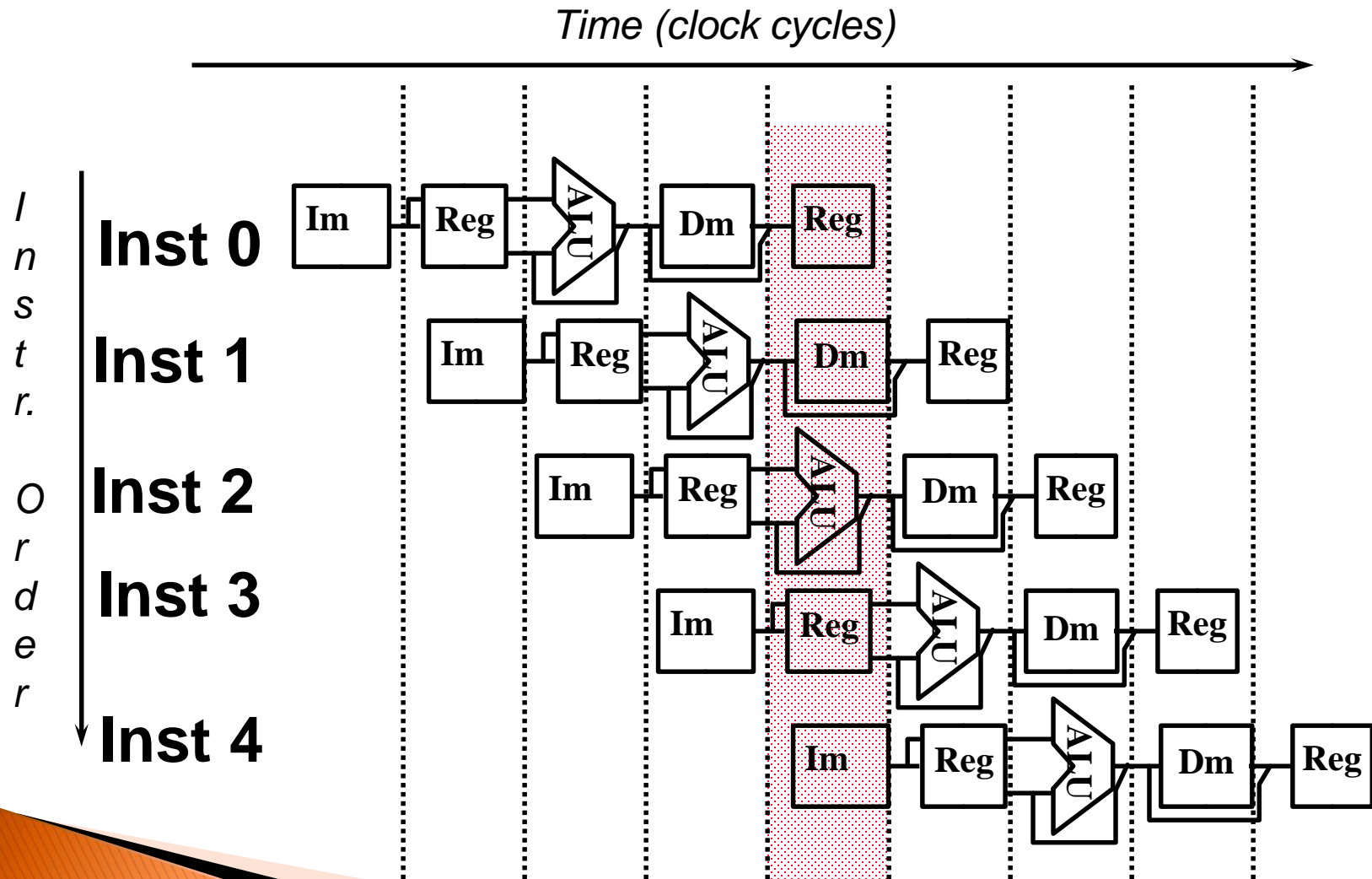


Graphically Representing Pipelines



- Can help with answering questions like:
 - How many cycles does it take to execute this code?
 - What is the ALU doing during cycle 4?
 - Are two instructions trying to use the same resource at the same time?

Why Pipeline? Because the resources are there!



Why Pipeline?

▶ Suppose

- 100 instructions are executed
- The single cycle machine has a cycle time of 45 ns
- The multicycle and pipeline machines have cycle times of 10 ns
- The multicycle machine has a CPI of 3.6

▶ Single Cycle Machine

- $45 \text{ ns/cycle} \times 1 \text{ CPI} \times 100 \text{ inst} = 4500 \text{ ns}$

▶ Multicycle Machine

- $10 \text{ ns/cycle} \times 3.6 \text{ CPI} \times 100 \text{ inst} = 3600 \text{ ns}$

▶ Ideal pipelined machine

- $10 \text{ ns/cycle} \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle drain}) = 1040 \text{ ns}$

▶ Ideal pipelined vs. single cycle speedup

- $4500 \text{ ns} / 1040 \text{ ns} = 4.33$

▶ What has not yet been considered?

Can pipelining get us into trouble?

► Yes: Pipeline Hazards

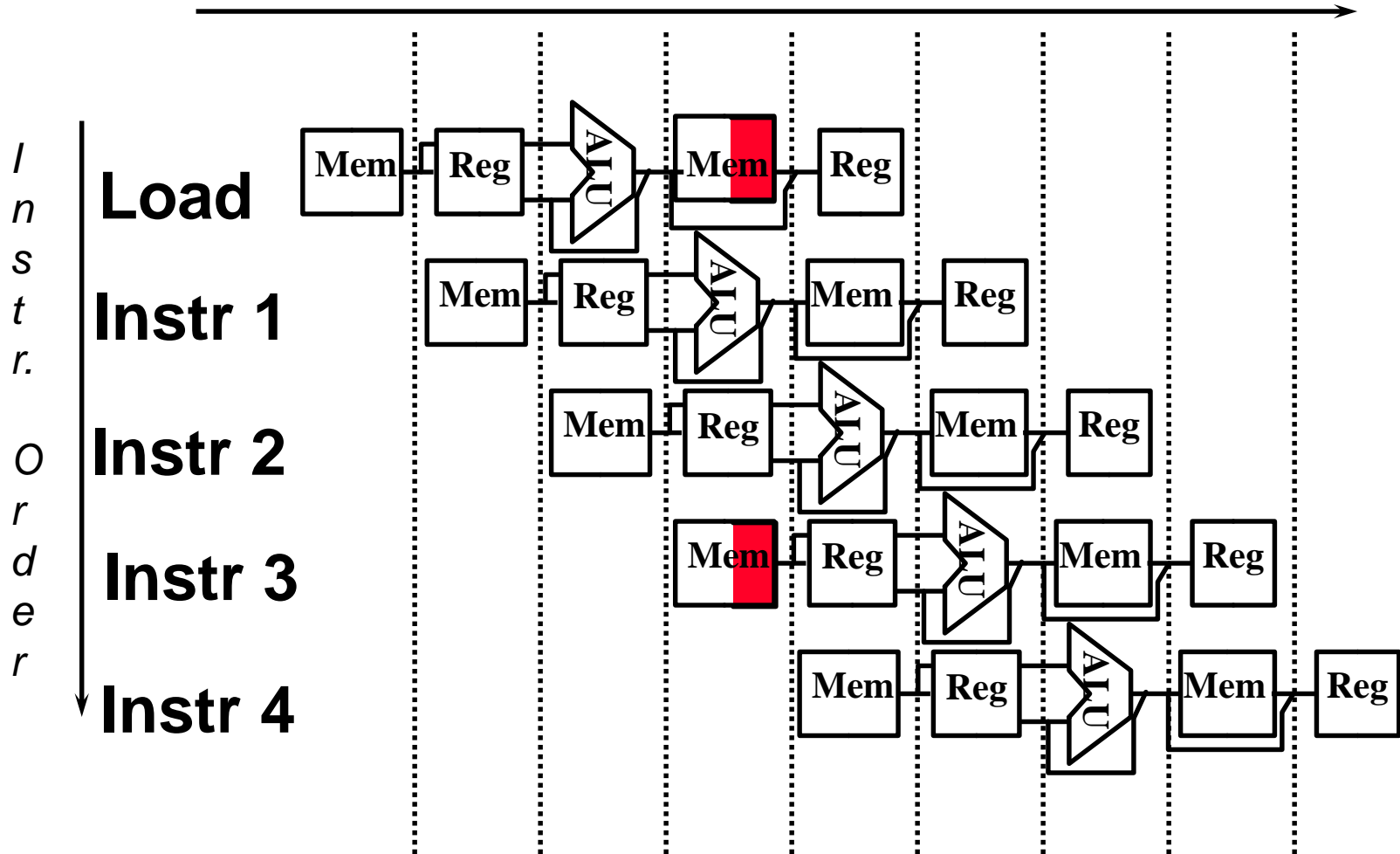
- **structural hazards**: attempt to use the same resource (hardware unit) two different ways at the same time
 - E.g., two instructions try to read the same memory at the same time
- **data hazards**: attempt to use item before it is ready
 - instruction depends on result of prior instruction still in the pipeline
 - add **r1**, r2, r3
 - sub r4, r2, **r1**
- **control hazards**: attempt to make a decision before condition is evaluated
 - branch instructions
 - beq r1, r2, loop
 - add r3, r4, r5

► Can always resolve hazards by **waiting**

- pipeline control must detect the hazard
- take action (or delay action) to resolve hazards

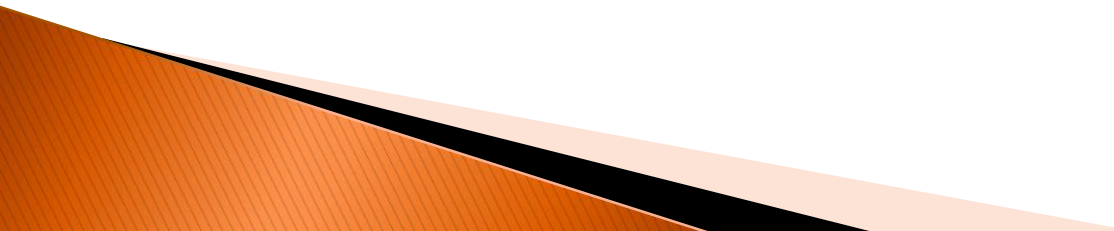
Single Memory is a Structural Hazard

Time (clock cycles)

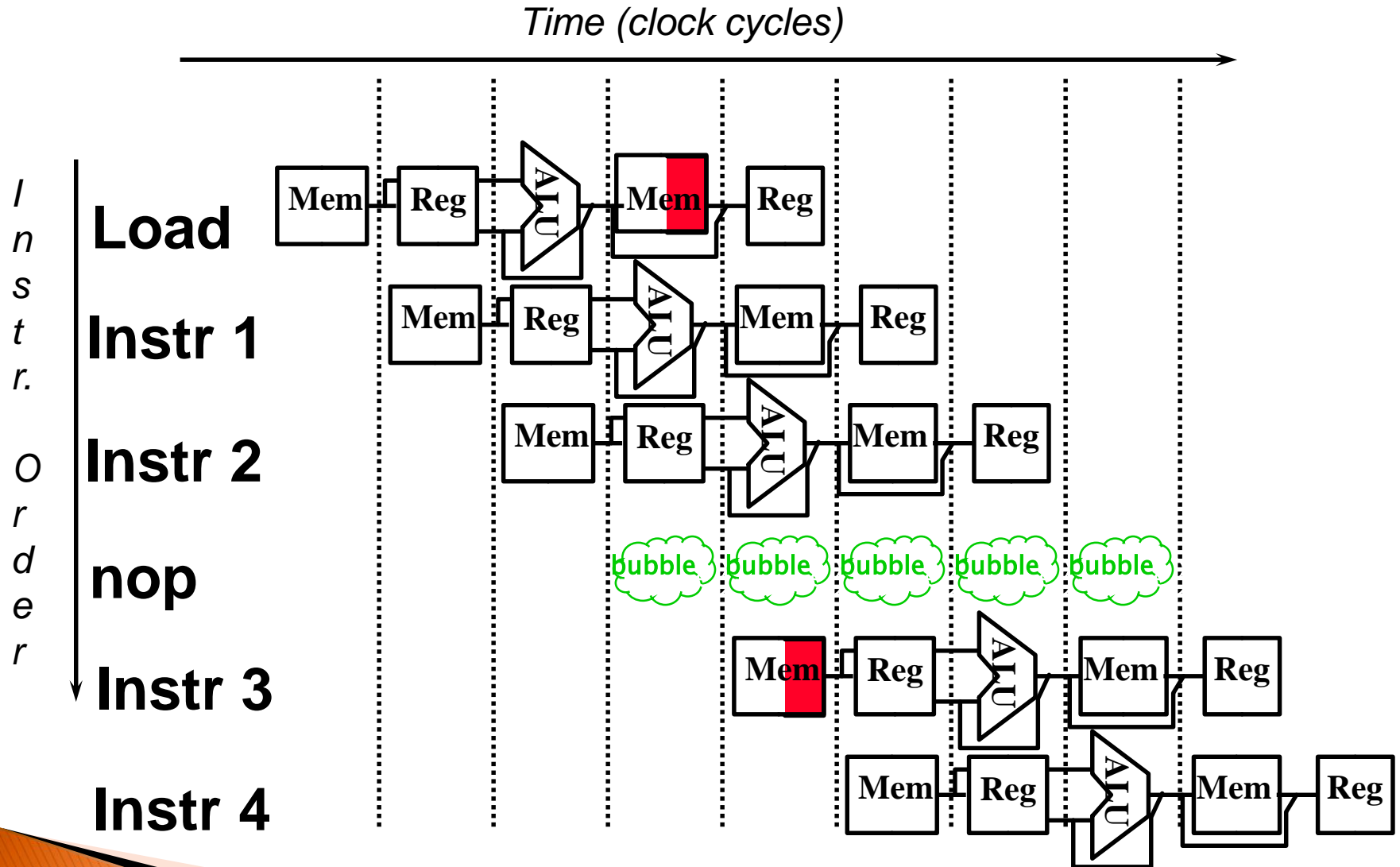


Detection is easy in this case! (right half highlight means read, left half write)

What's the Solution?

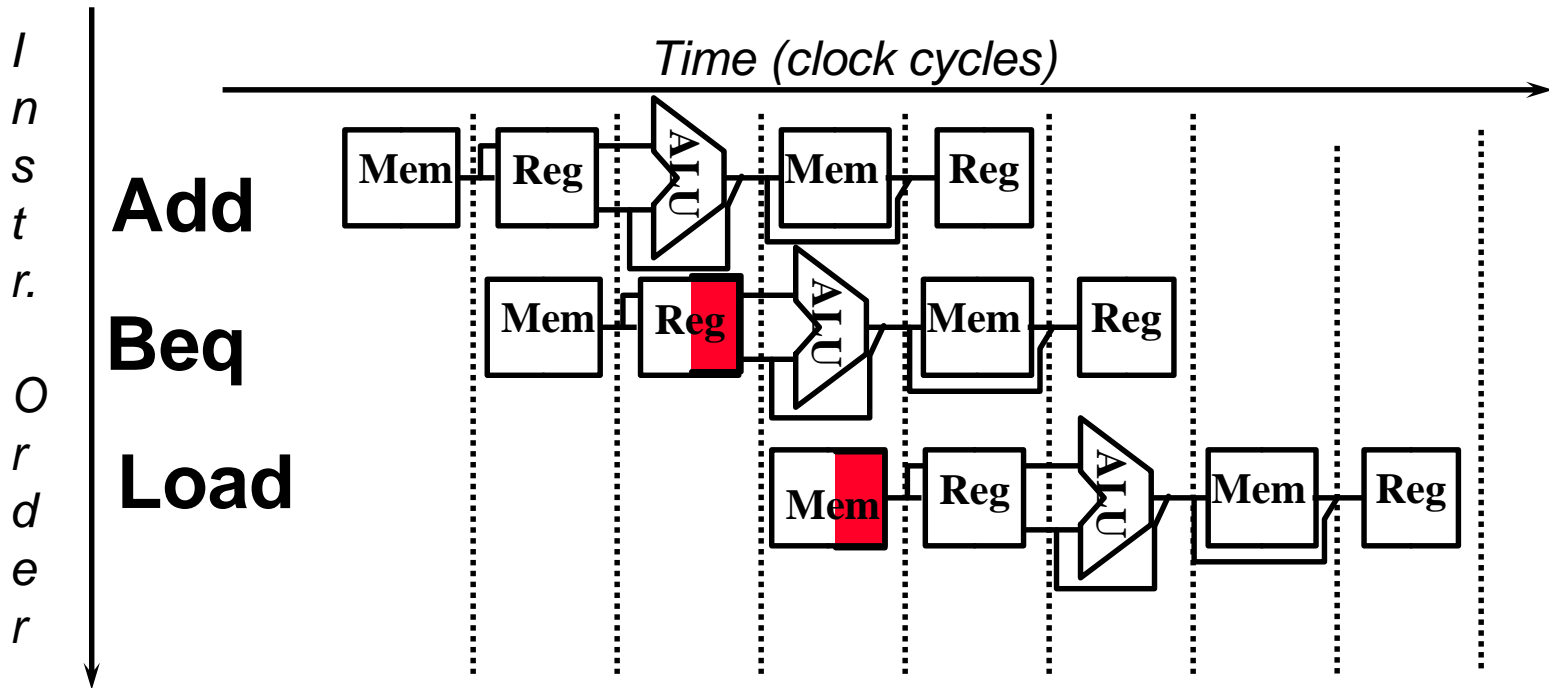
- ▶ **Solution 1:** Use separate instruction and data memories
 - ▶ **Solution 2:** Allow memory to read and write more than one word per cycle
 - ▶ **Solution 3:** Stall
- 

Solution 3: Stall



Control Hazard Solutions

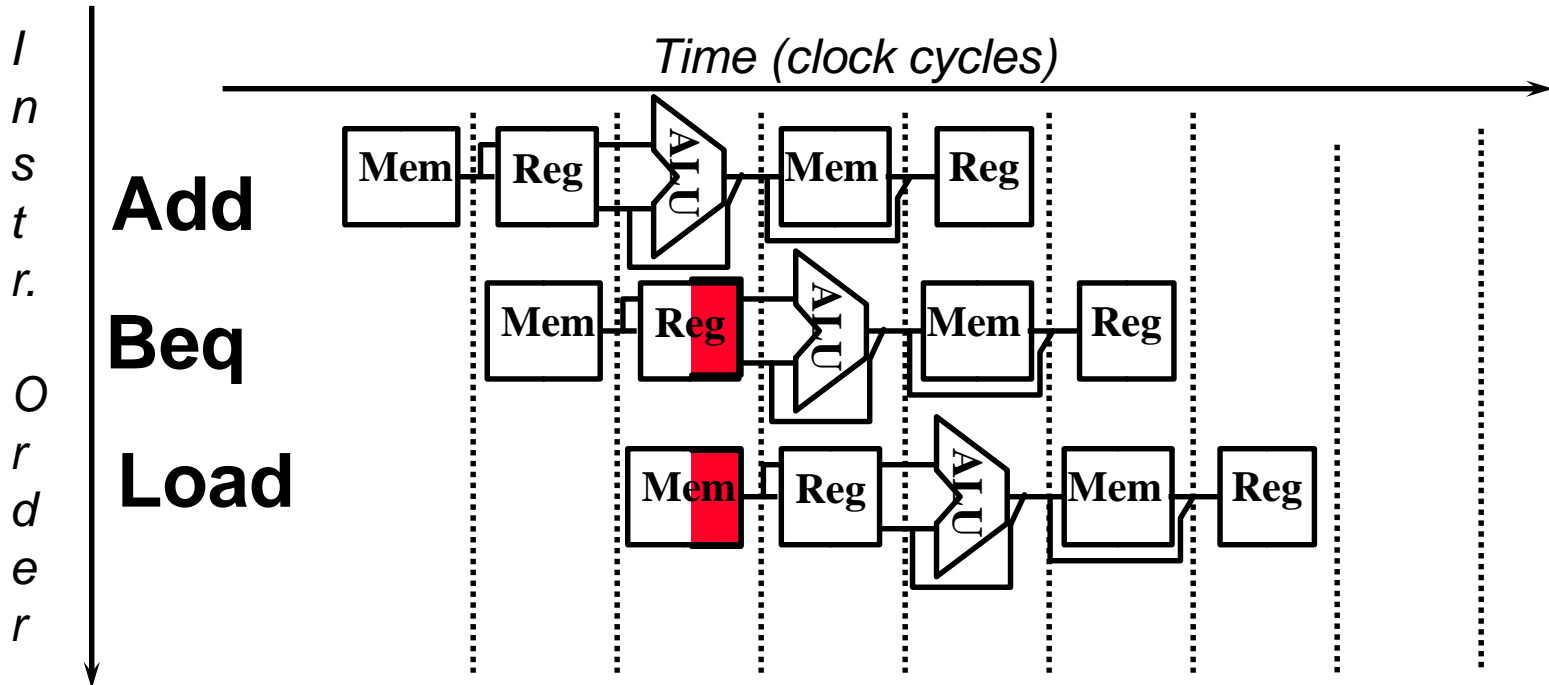
- ▶ Stall: wait until decision is clear
 - It is possible to move up decision to 2nd stage by adding extra hardware to check registers as being read



- ▶ Impact: 2 clock cycles per branch instruction => slow

Control Hazard Solutions

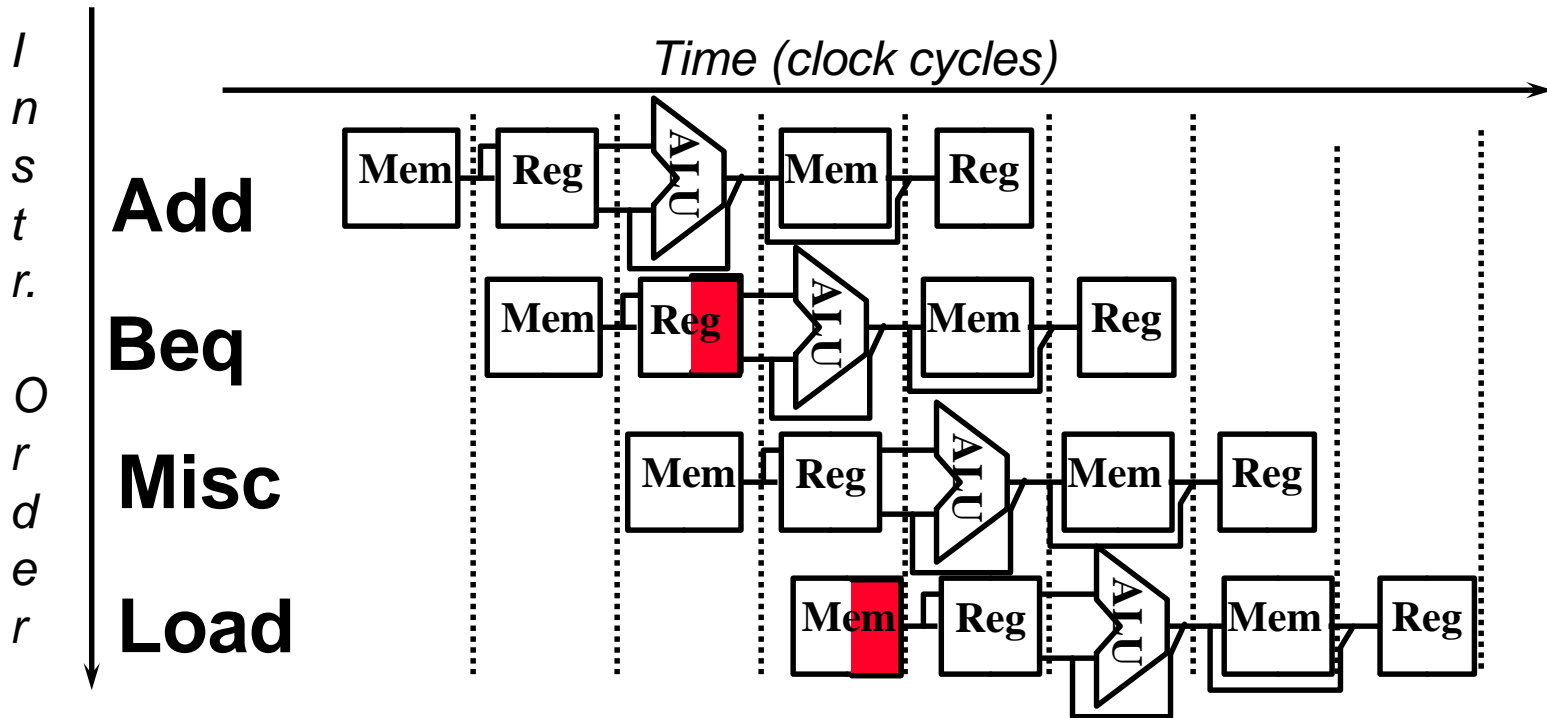
- ▶ Predict: guess one direction then back up if wrong
 - Predict not taken



- ▶ Impact: 1 clock cycle per branch instruction if right, 2 if wrong (right 50% of time)
- ▶ More dynamic scheme: history of 1 branch (90%)

Control Hazard Solutions

- ▶ Redefine branch behavior (takes place after next instruction) “**delayed branch**”



- ▶ Impact: 1 clock cycles per branch instruction if can find instruction to put in “slot” (50% of time)

Data Hazard on r1

- Problem: r1 cannot be read by other instructions before it is written by the add.

add r1, r2, r3

sub r4, r1, r3

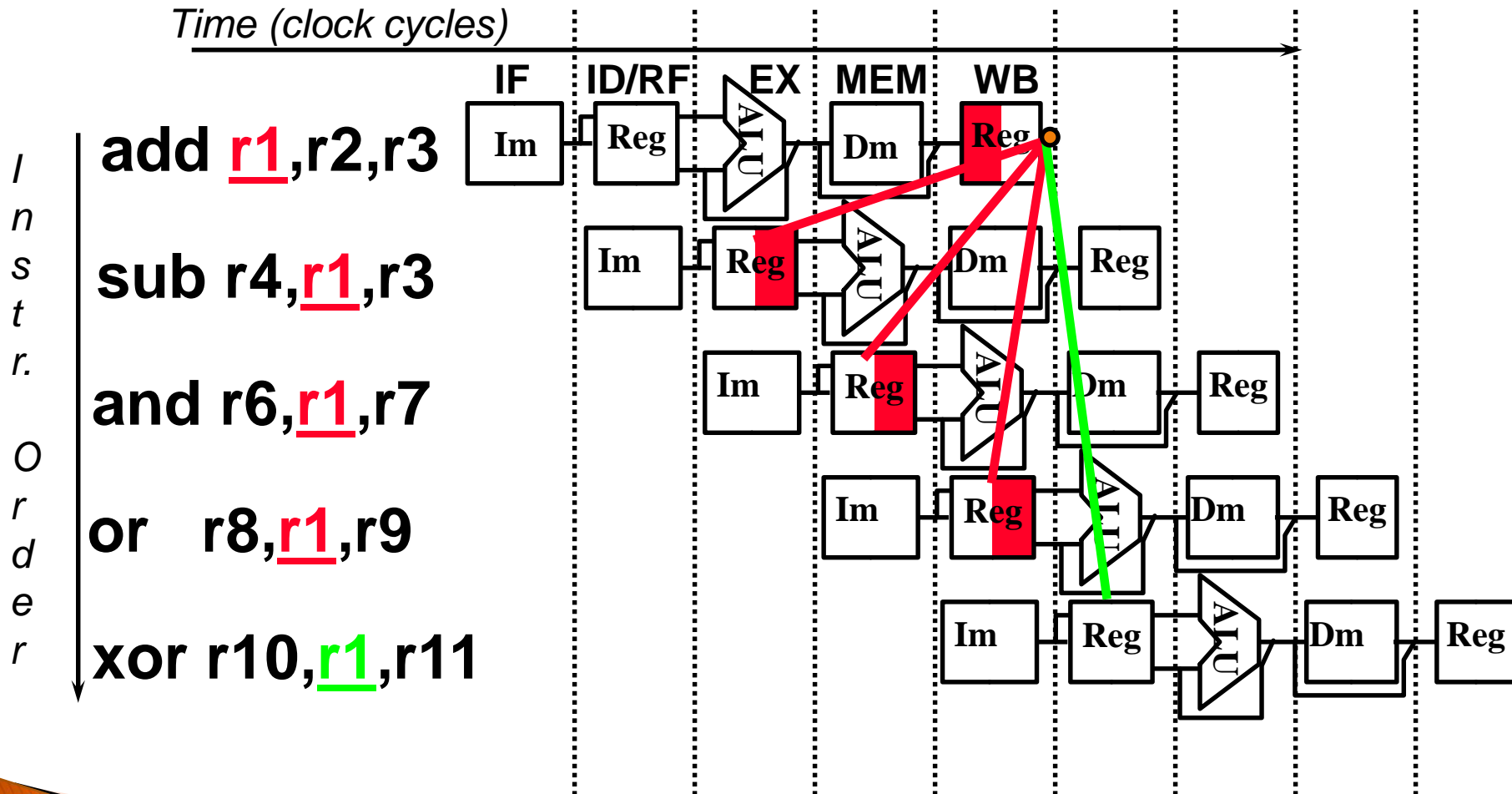
and r6, r1, r7

or r8, r1, r9

xor r10, r1, r11

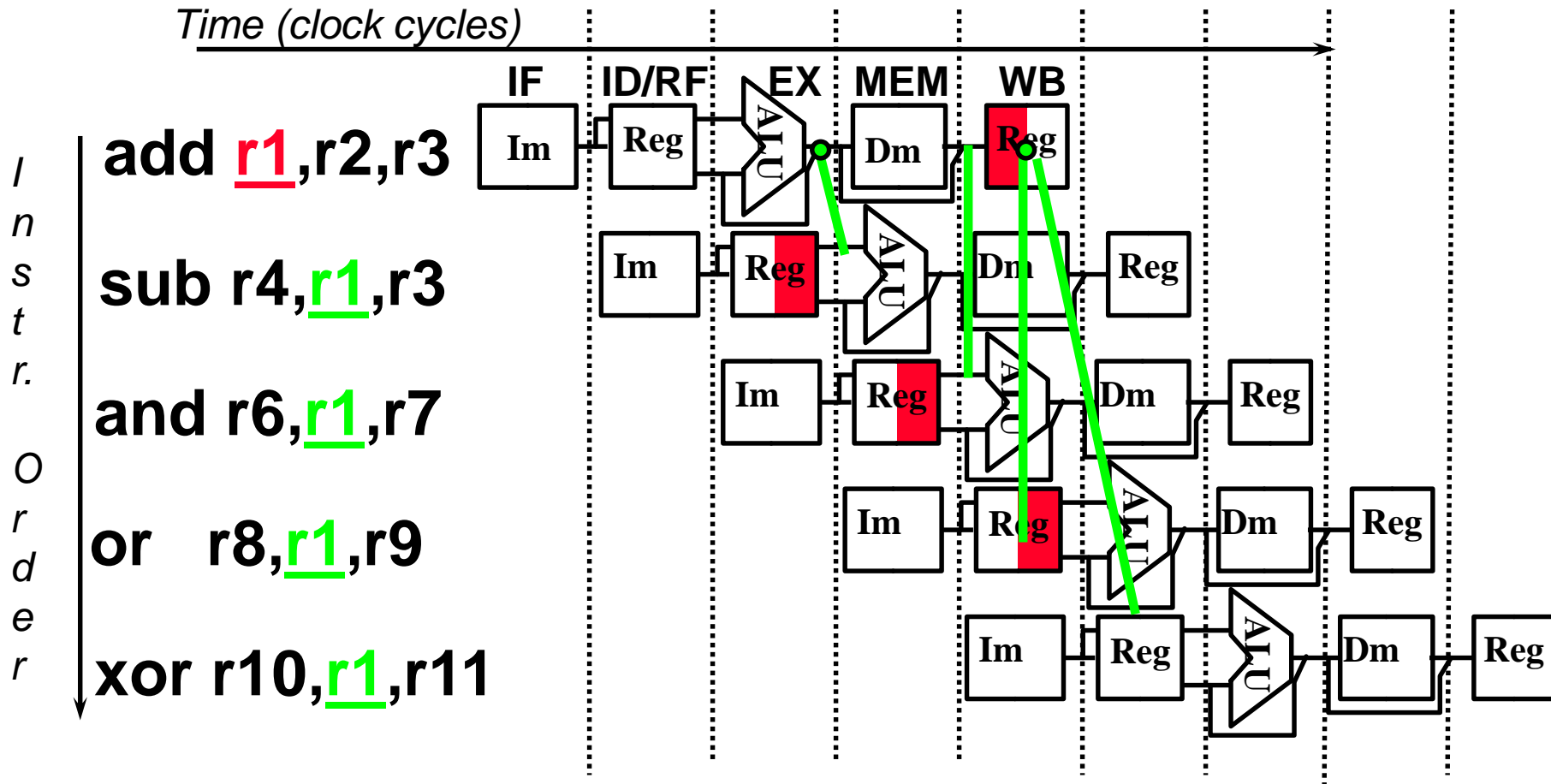
Data Hazard on r1:

- Dependencies backwards in time are hazards



Data Hazard Solution:

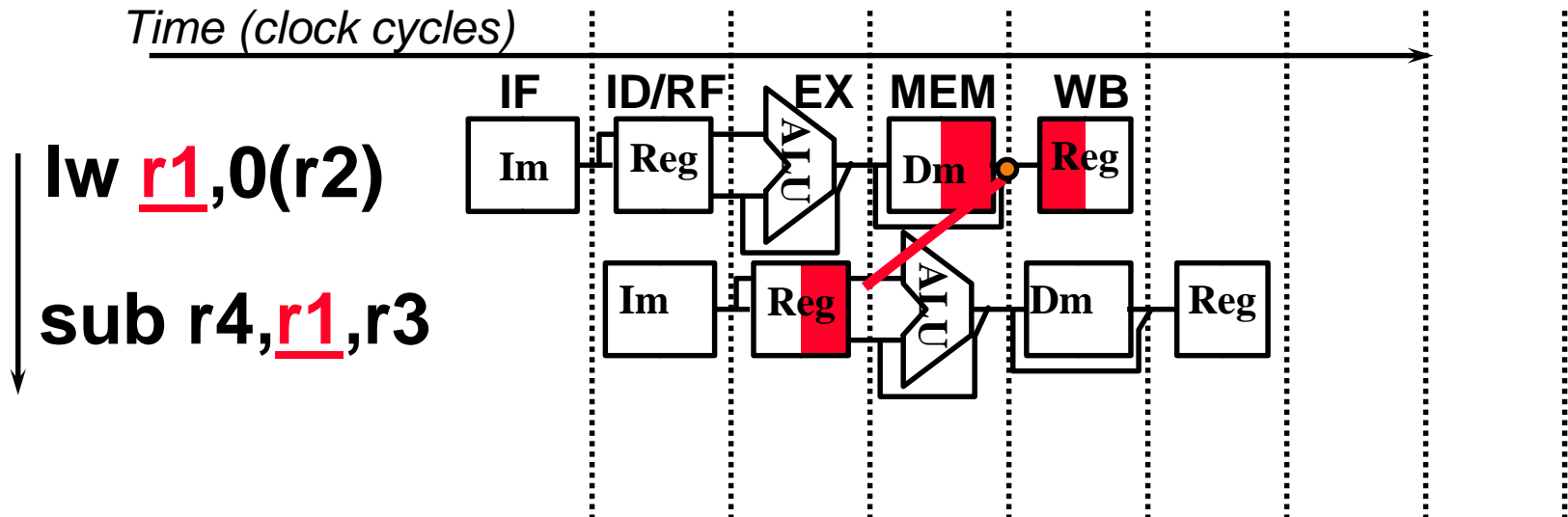
- **“Forward”** result from one stage to another



- “or” instruction is OK if define read/write properly

Forwarding (or Bypassing): What about Loads

- Dependencies backwards in time are hazards



- Can't solve with forwarding:
- Must delay/stall instruction dependent on loads