# Topics

- Single Cycle CPU Design (sections 4.1-4.4)

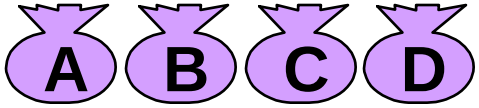
- **Pipelining**
  - ➔ Overview (section 4.6)
  - Pipelined Datapath and control (section 4.7)
  - Data hazard (section 4.8)
  - Control hazard (section 4.9)

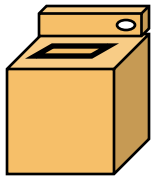
***Thus times do shift,***
***each thing his turn does hold;***
***New things succeed,***
***as former things grow old.***

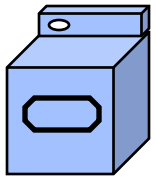
***Robert Herrick***

# Pipelining is Natural!

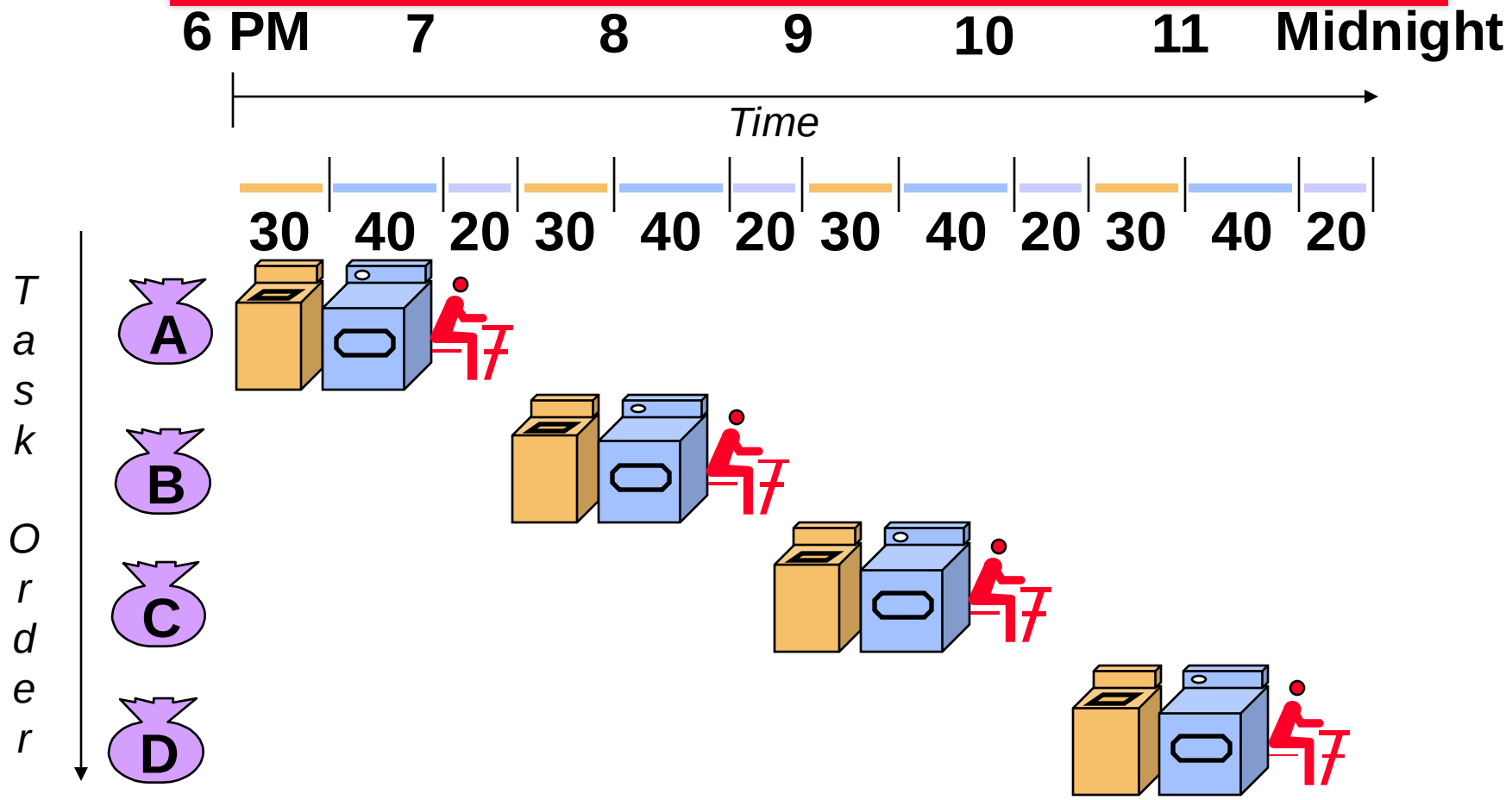
- **Pipelining is an implementation technique in which multiple instructions are overlapped in execution**
- **Laundry Example**
  - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes



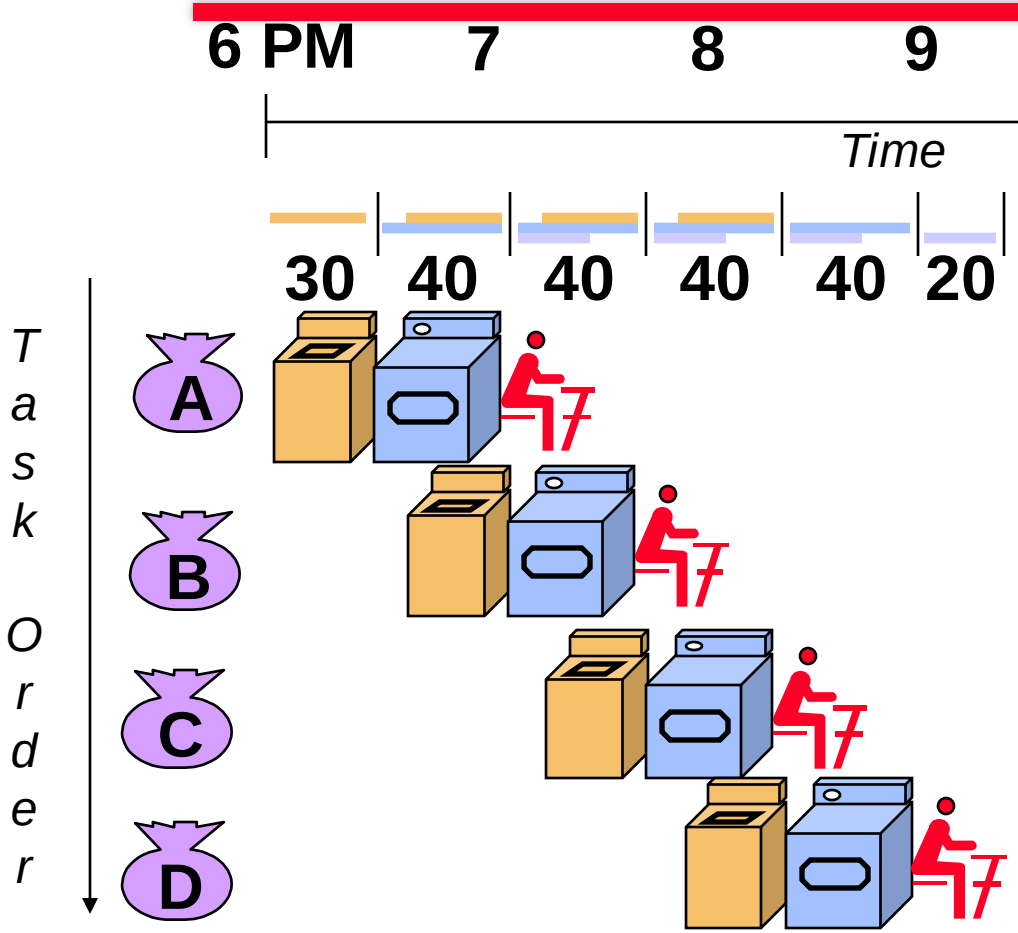


# Sequential Laundry



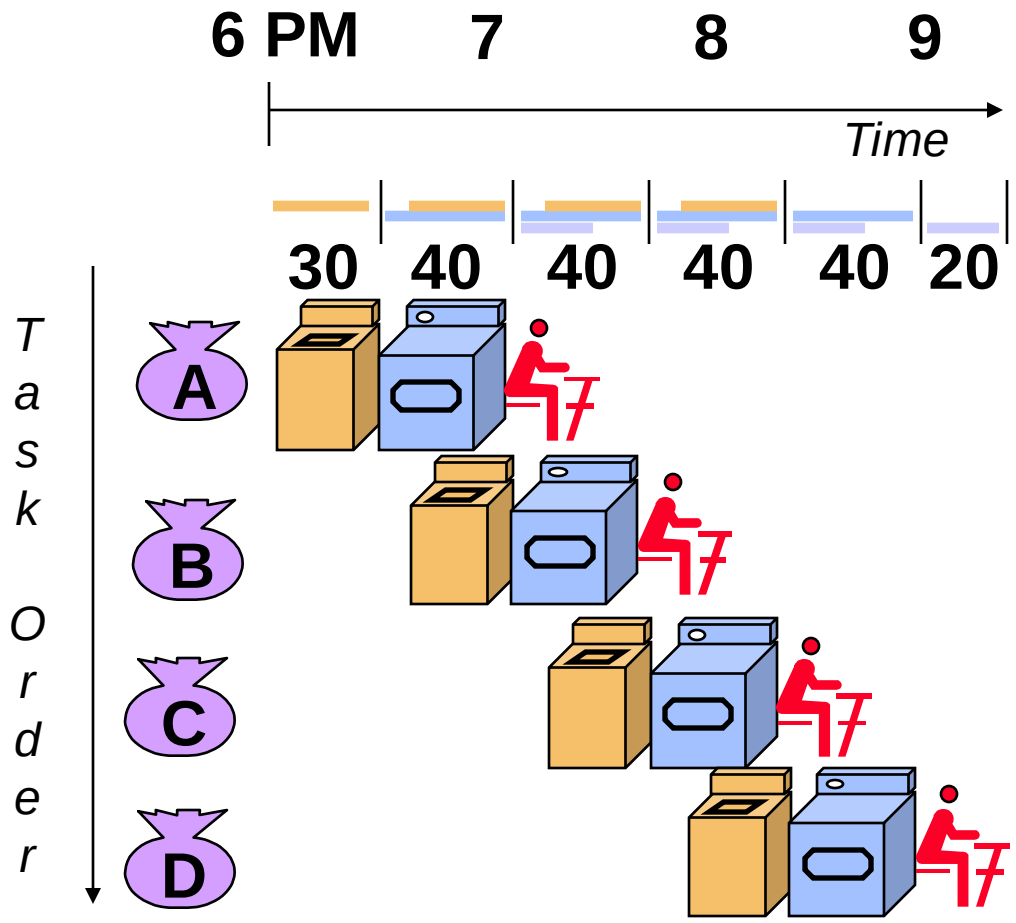

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

# Pipelined Laundry: Start work ASAP




- Pipelined laundry takes 3.5 hours for 4 loads
- 30 (1st load for washer) + 40*3 (next 3 loads for washer) + (40+20) (finishing up the last load) = 210 minutes

# Pipelining Lessons




- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate is limited by slowest pipeline stage
- Unbalanced lengths of pipeline stages reduces speedup
- Multiple tasks operating simultaneously using different resources
- Potential speedup = the number of pipeline stages
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependencies

# Ideal Speedup

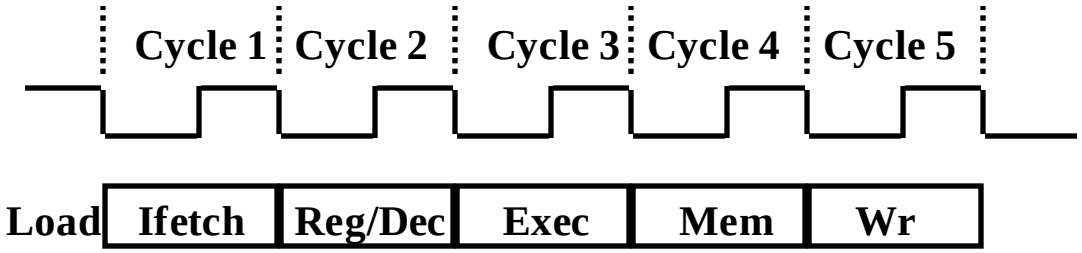
- **Ideal**
  - Each stage takes the same amount of time (t)
  - Each instruction needs the same number of stages (s)
  - The number of instructions (n) is infinite ∞
- **$ET_{sequential} = nst$**
- **$ET_{pipelined} = st + (n-1)t = (n-1+s)t$**
- **$Speedup = ET_{sequential}/ET_{pipelined} = ns/(n-1+s)$**
- **Ideal speedup**

**$= \lim_{n \to \infty} speedup$**

**$= \lim_{n \to \infty} (ns/(n-1+s))$**

**$= s$**

# The Five Stages of LW

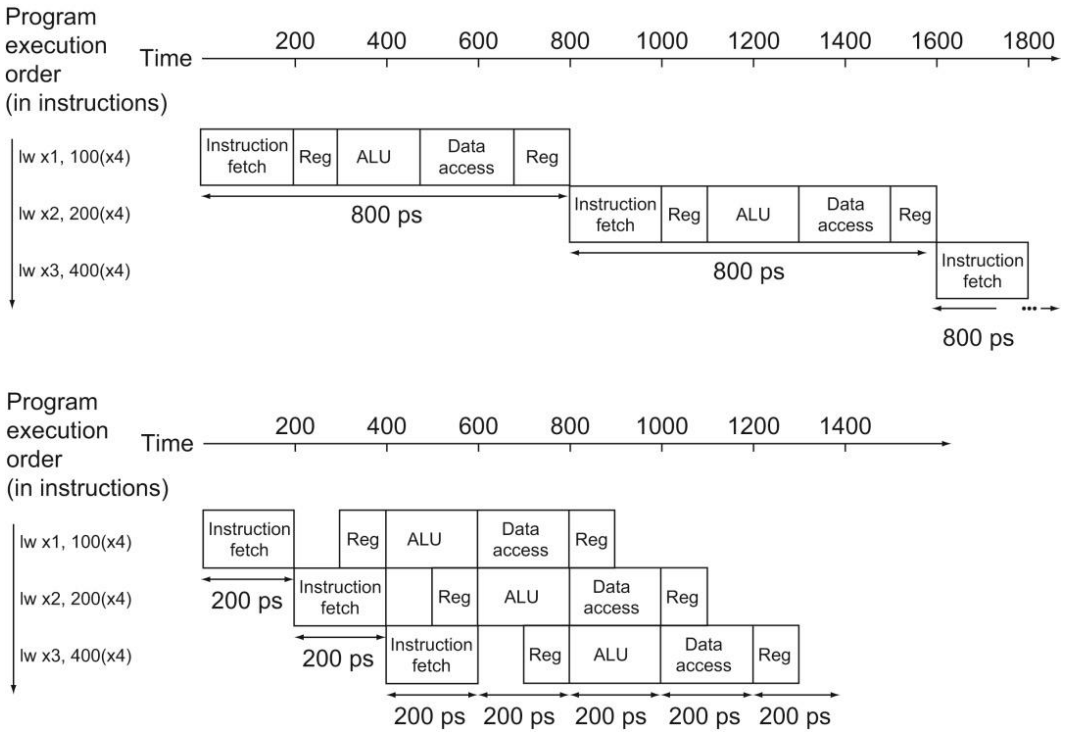



- **Each step takes one clock cycle**
- **One step = one stage**

- **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**

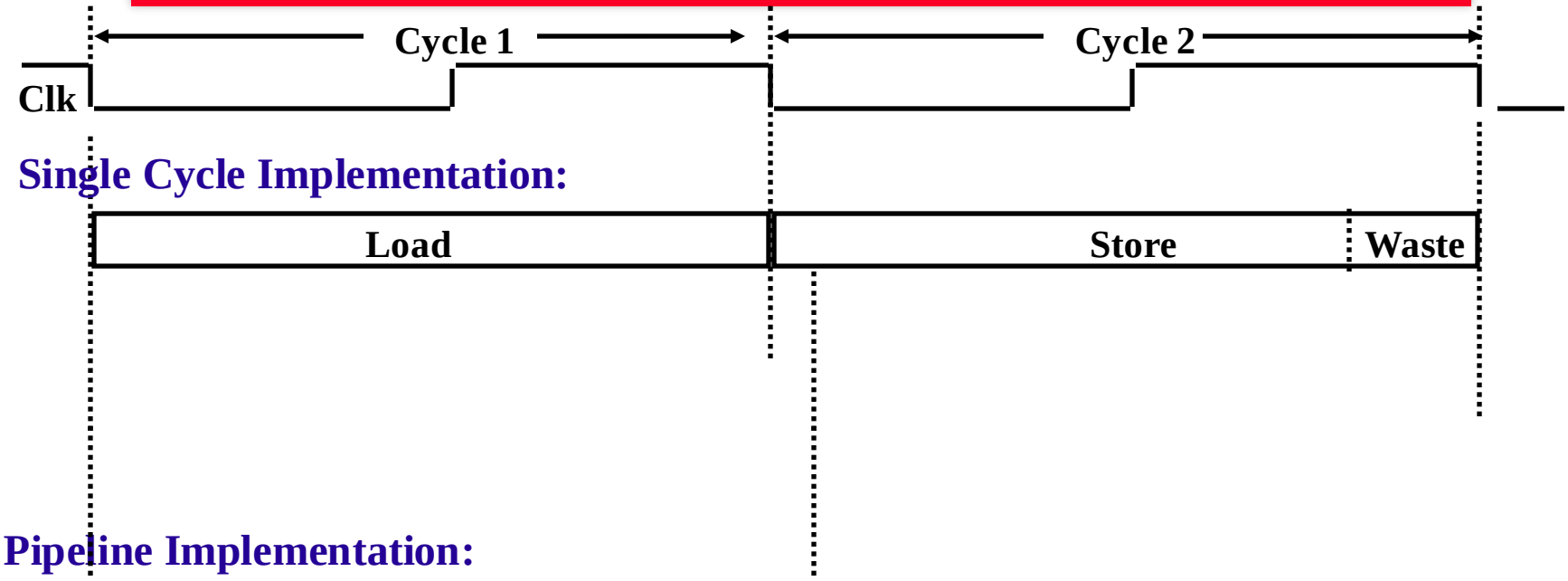
# Pipelining

- **Improve performance by increasing throughput**




- ***Do we achieve the ideal speedup (number of stages in the pipeline)?*** **NO!**
  - pipeline stage time: limited by the slowest resource (ALU operation, or memory access)
  - Fill and drain time

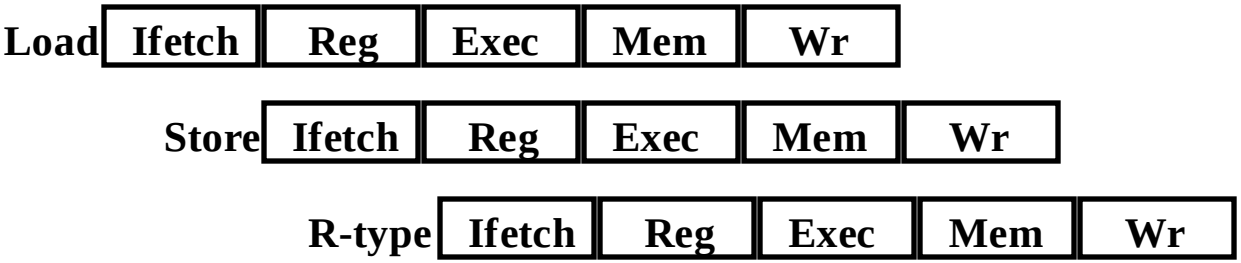
# Single Cycle vs. Pipeline




**Single Cycle Implementation:**



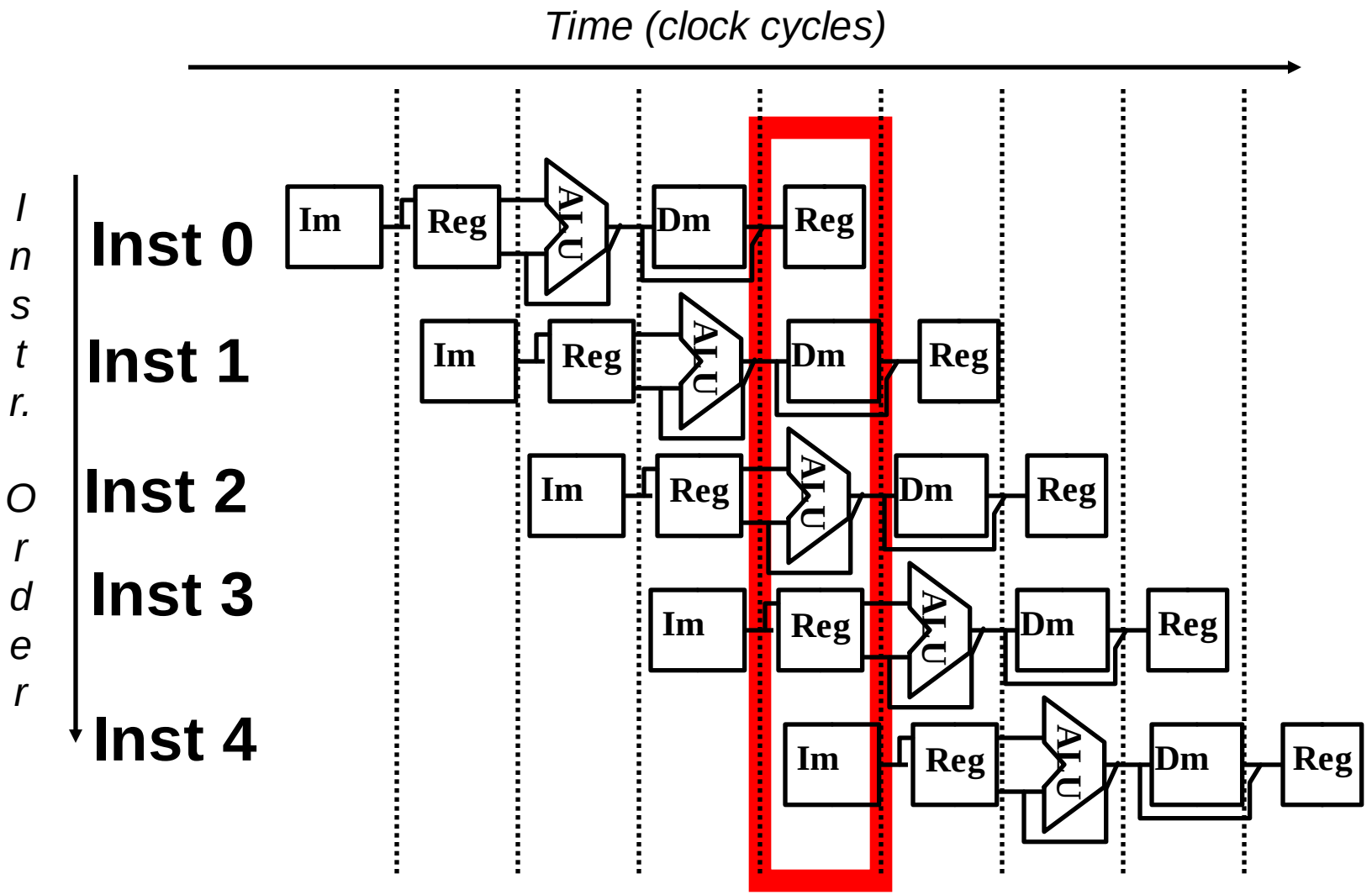

**Pipeline Implementation:**



# *Why Pipeline? To improve performance!*

- **Suppose**
  - we execute 100 instructions
  - 45 ns for lw (one cycle)
  - 10 ns for each stage
- **Single Cycle Machine**
  - 45 ns/cycle  x 1 CPI x 100 inst = 4500 ns
- **Ideal pipelined machine**
  - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns

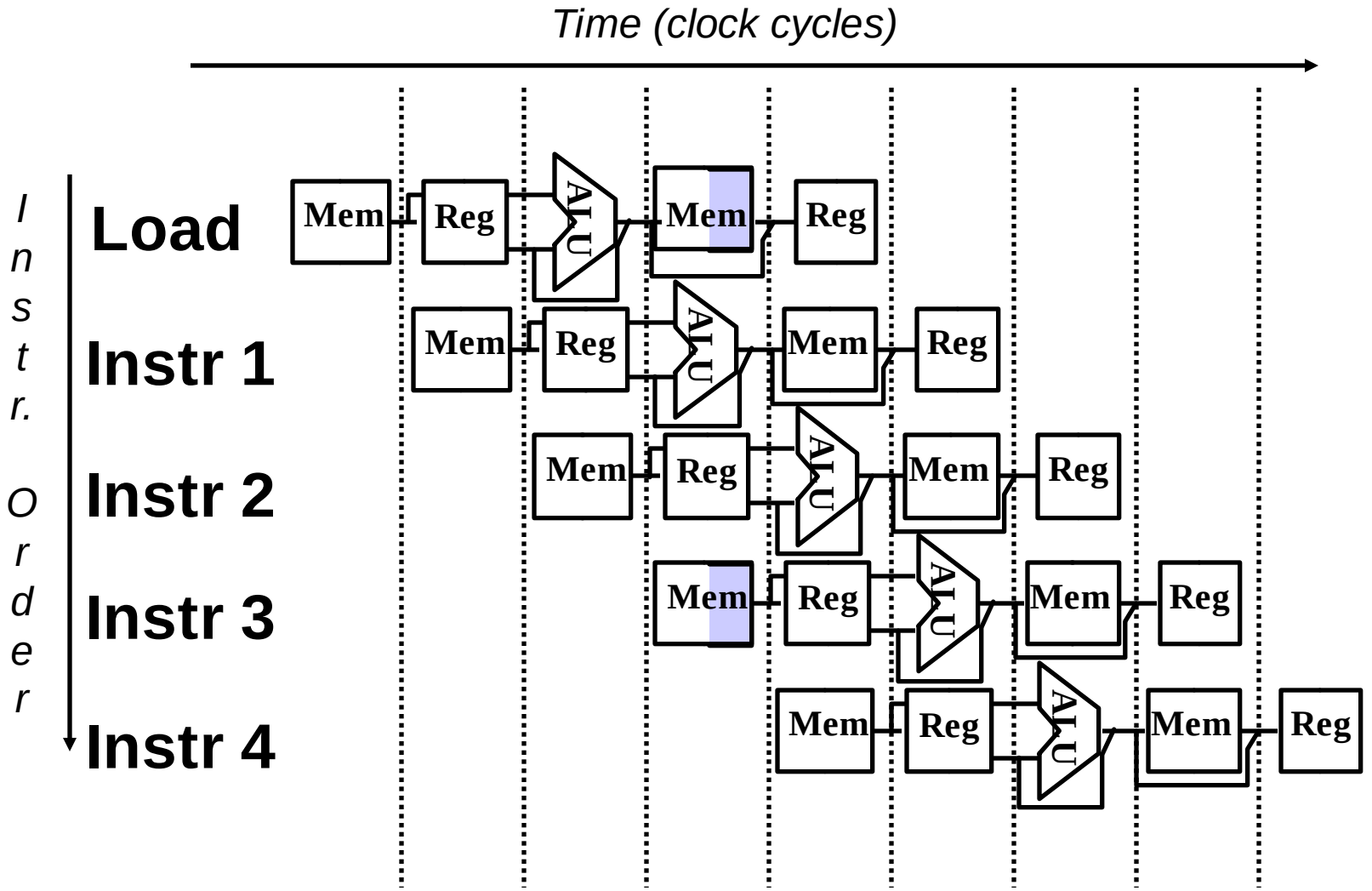
# Why Pipeline? Because resources are there!



# *Can pipelining get us into trouble?*

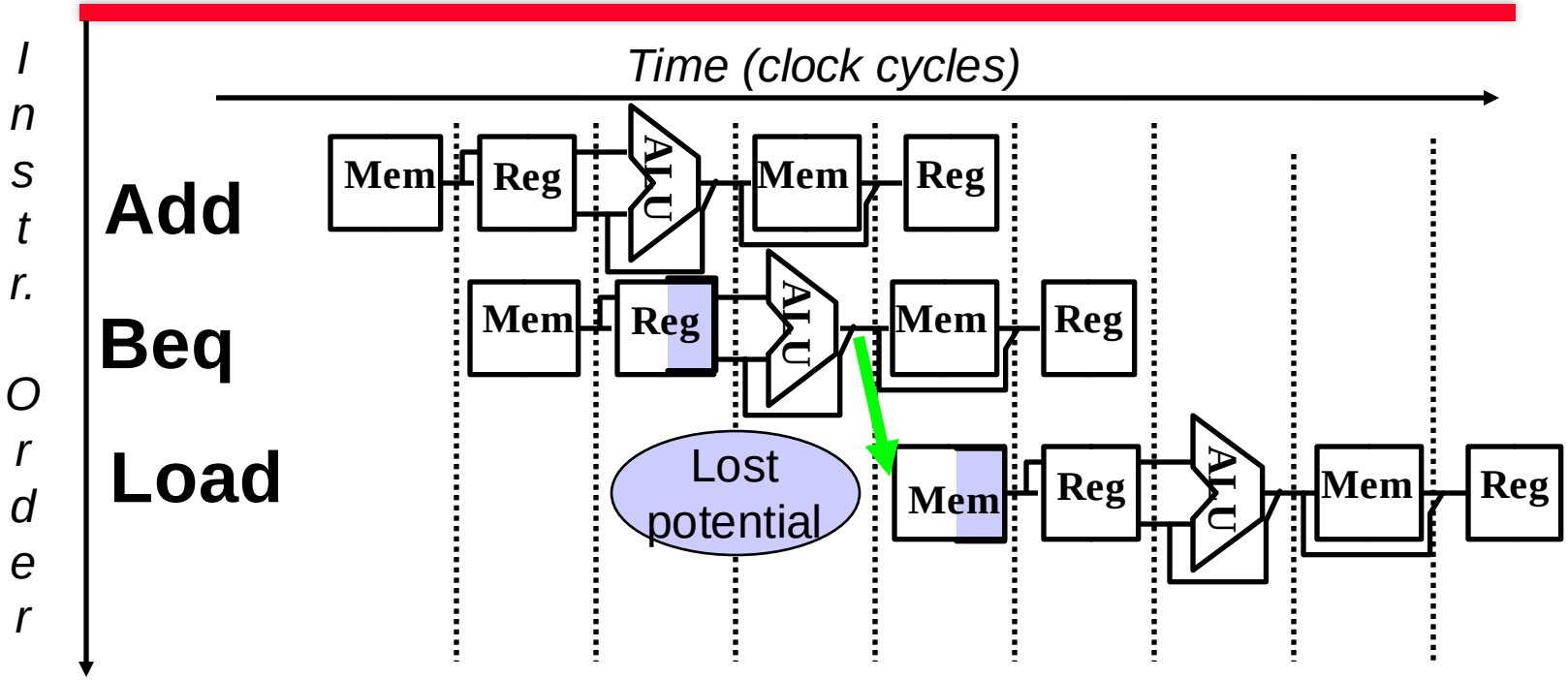
- **Yes: Pipeline Hazards**
  - Structural hazards:
    - attempt to use the same resource in two different ways at the same time
  - Control hazards:
    - attempt to make a decision before condition is evaluated
  - Data hazards:
    - attempt to use item before it is ready
- **Can always resolve hazards by waiting**
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards

# Single Memory is a Structural Hazard



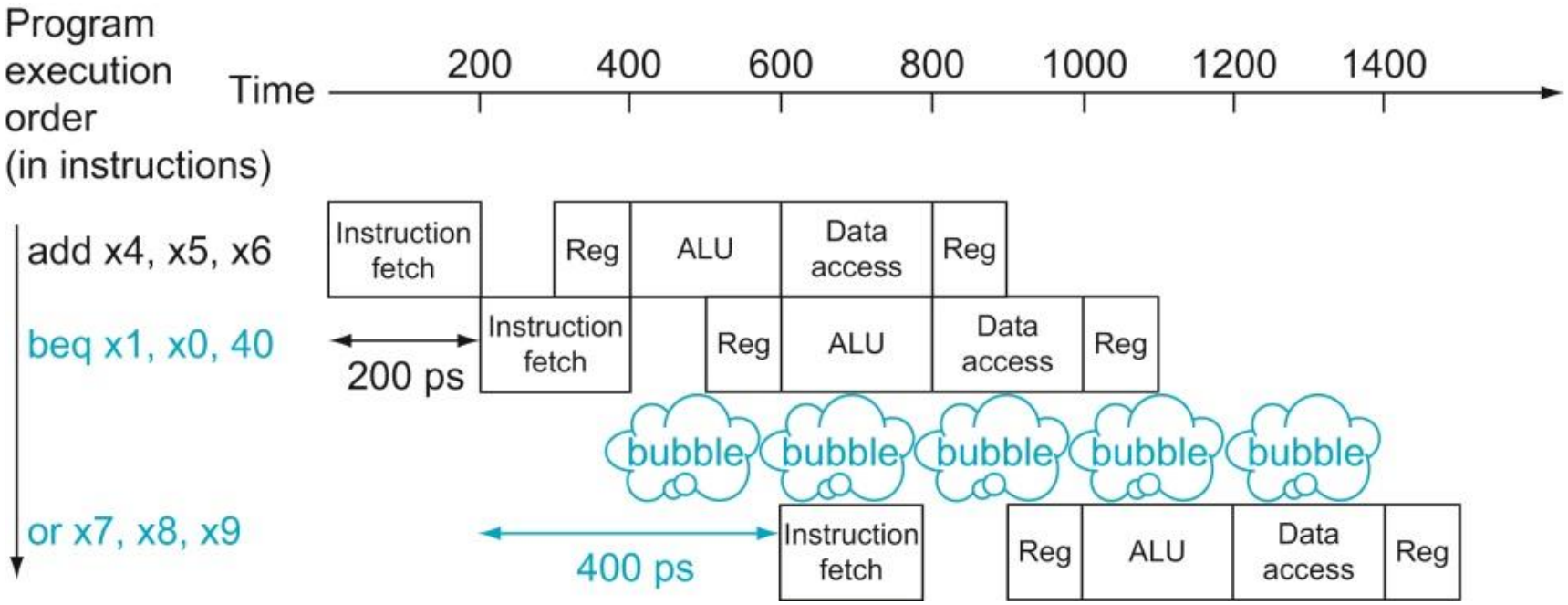

Memory notation: right half highlight means read, left half highlight means write

# Control Hazard Solution #1: Stall



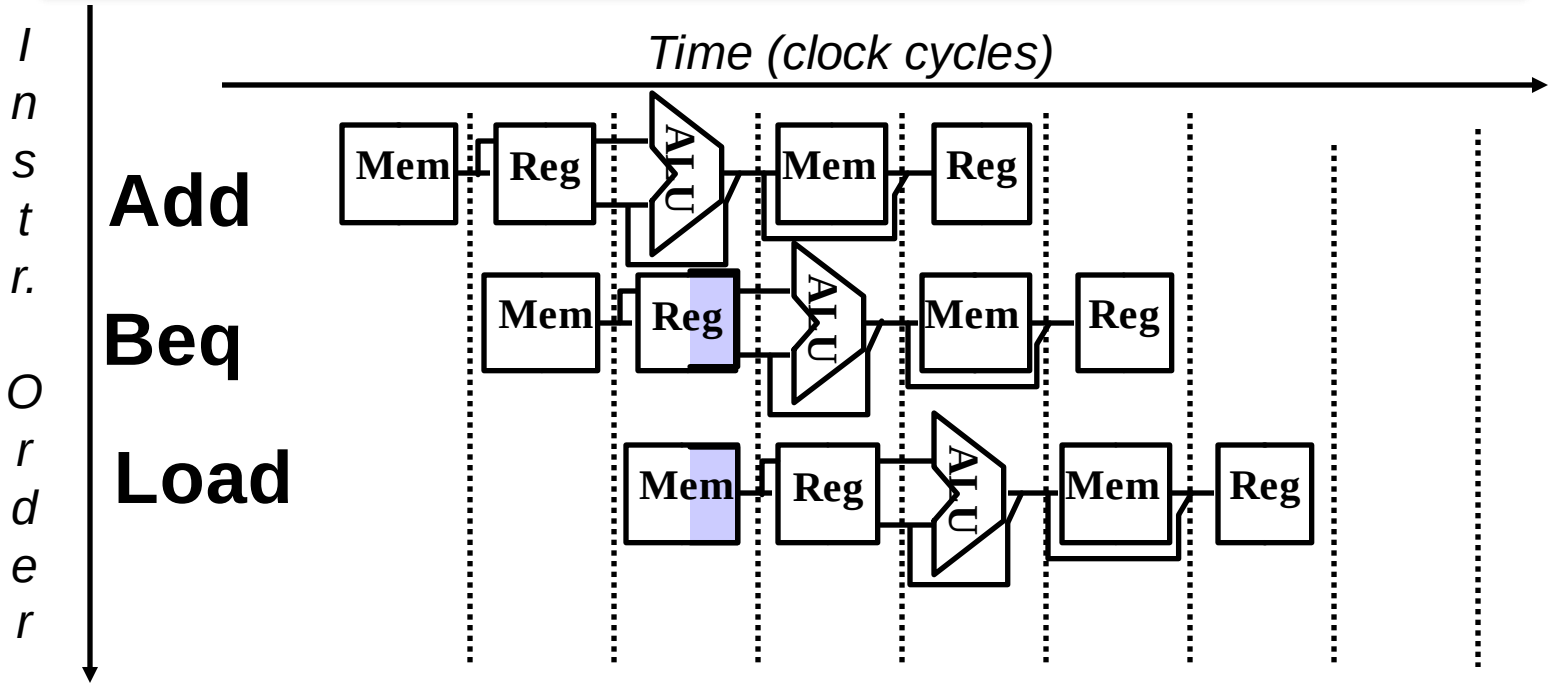

- **Stall: wait until decision is clear**
- **Impact**
  - 2 lost cycles (i.e., 3 clock cycles per branch instruction) =>slow
  - Example: If 20% instructions are BEQ, all others have CPI 1, what is the average CPI?

# Control Hazard Solution #1: Stall




- **Move decision to end of decode by improving hardware**
  - save 1 cycle per branch

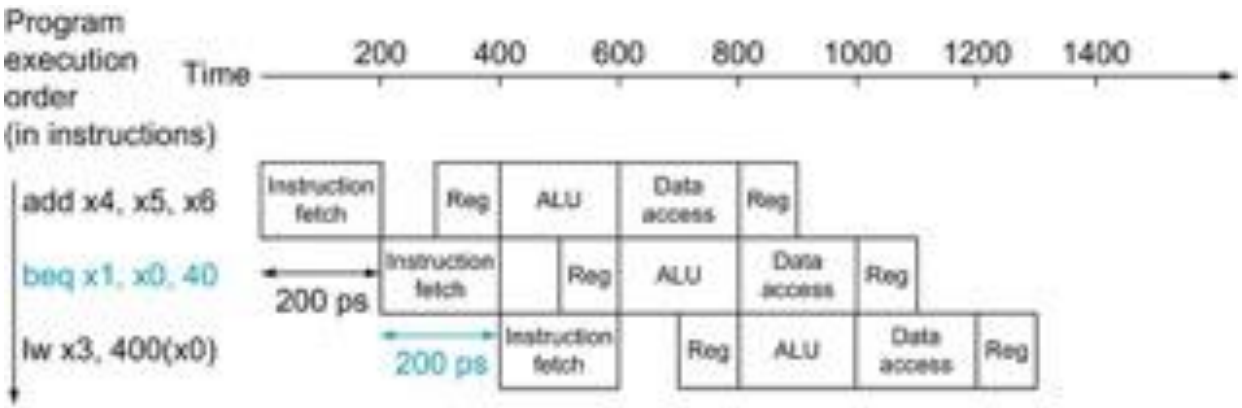
# Control Hazard Solution #2: Predict



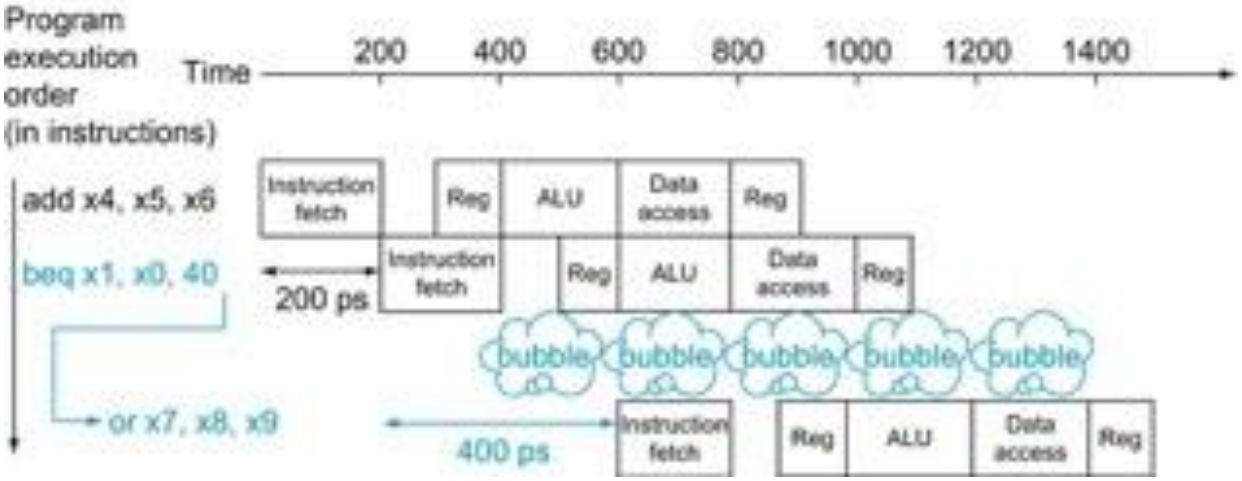

- **Predict: guess one direction then back up if wrong**
- **Impact: 0 lost cycles per branch instruction if right, 1 if wrong (prediction accuracy: 50%)**
  - Need to "Squash" and restart following instruction if wrong
  - Produce CPI on branch of (1 *.5 + 2 * .5) = 1.5
  - Total CPI might then be: 1.5 * .2 + 1 * .8 = 1.1 (20% branch)
- **More dynamic scheme: history of each branch (prediction accuracy: 90%)**

# Control Hazard Solution #2: Predict

**Predicting that branches are not taken**

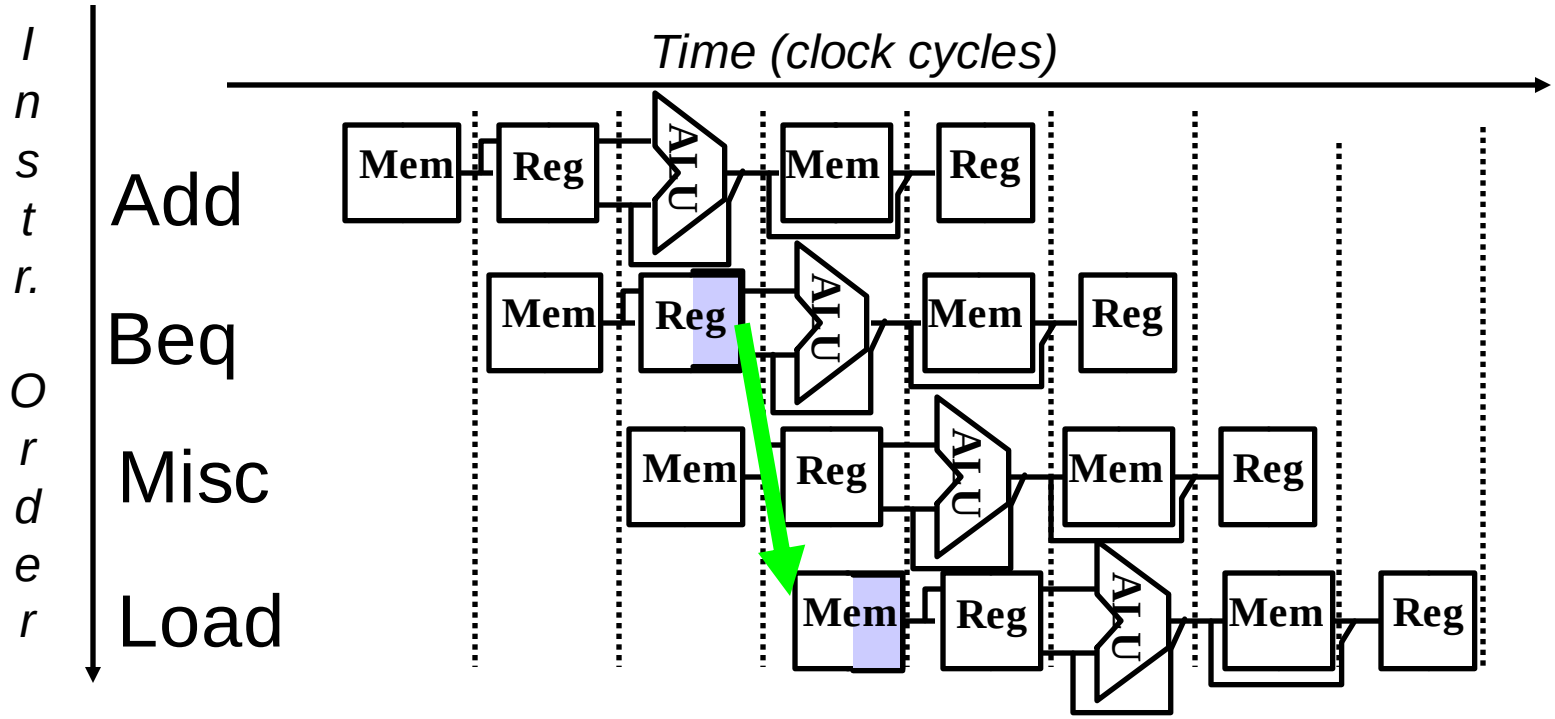



branches not taken



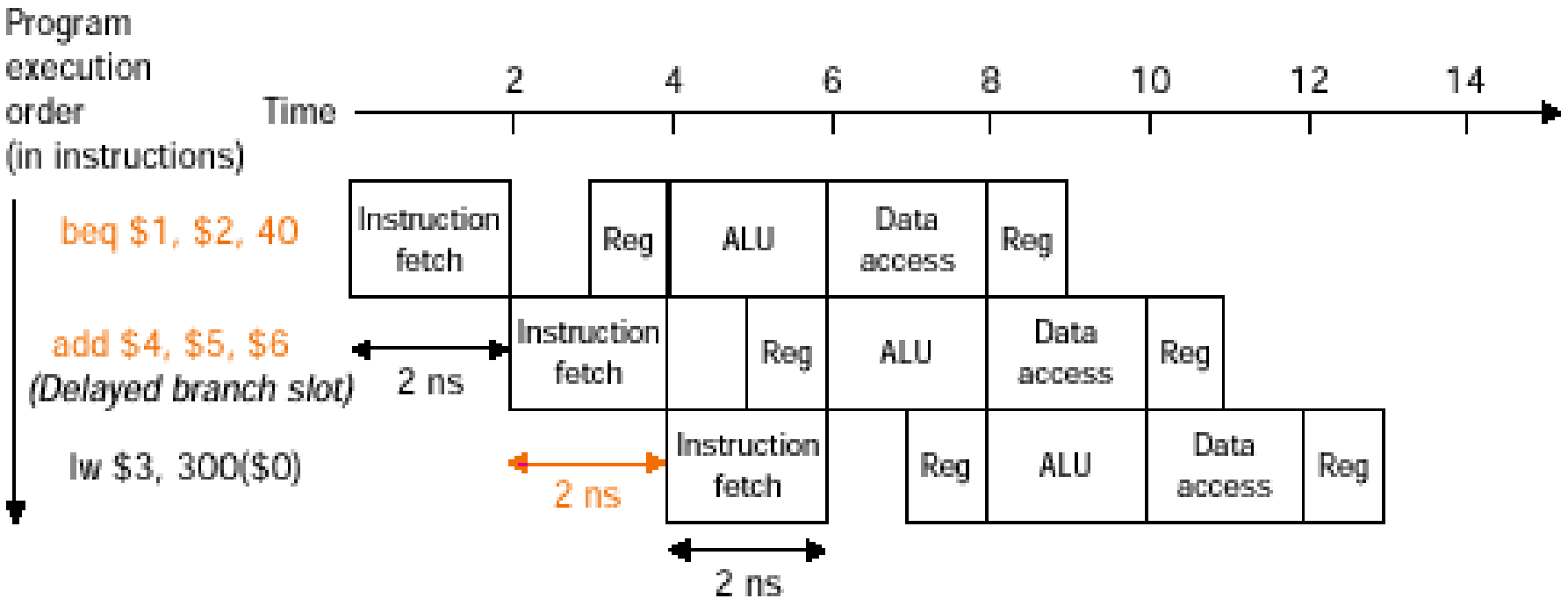

branches taken

# Control Hazard Solution #3: Delayed Branch




- **Delayed Branch:** Redefine branch behavior (takes place after next instruction)
- Impact: 0 extra clock cycles per branch instruction if can find instruction to put in "slot" ( 50% of time)
- The longer the pipeline, the harder to fill
- Used by RISC-V architecture

# Control Hazard Solution #3: Delayed Branch



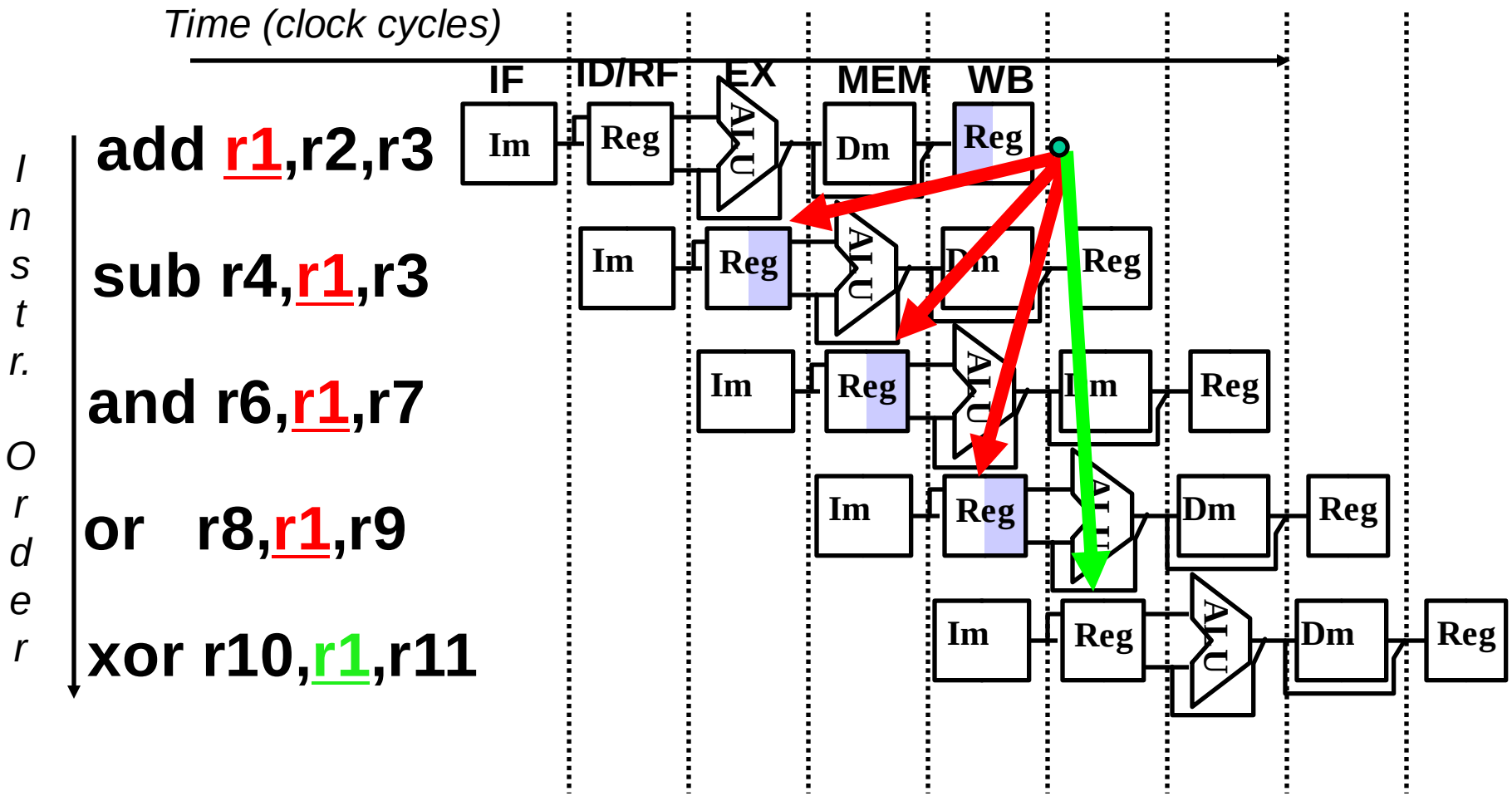
# Data Hazard

- An instruction depends on the result of a previous instruction still in the pipeline
- Example:

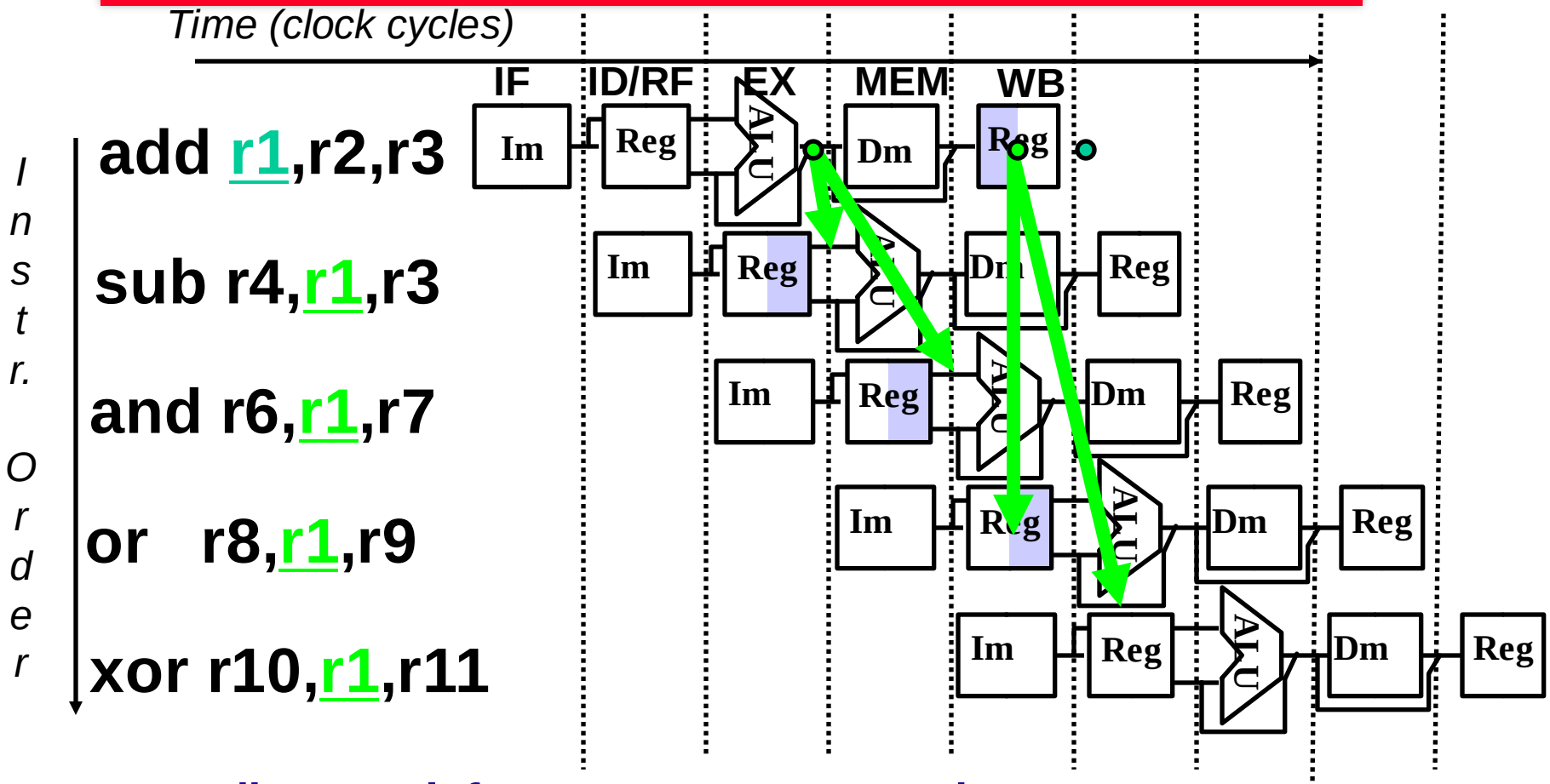
```
add r1 ,r2,r3

sub r4, r1 ,r3

and r6, r1 ,r7

or  r8, r1 ,r9

xor r10, r1 ,r11
```

# Data Hazard on r1

- **Hazards: Dependencies backwards in time**

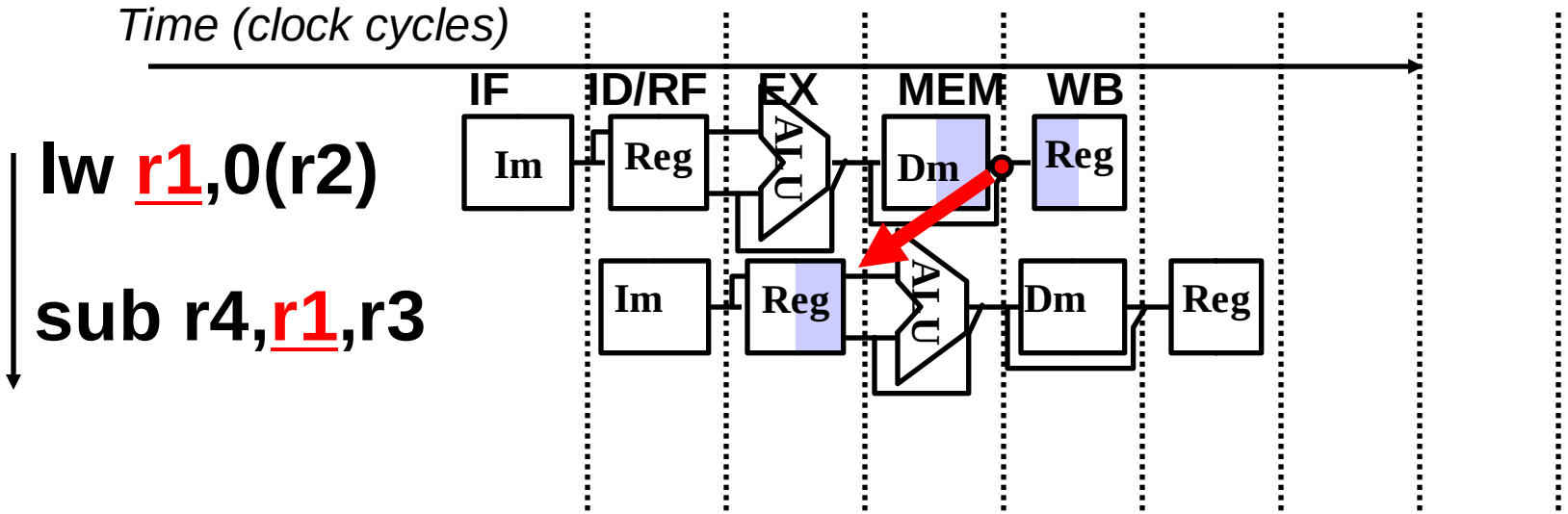


# Data Hazard Solution

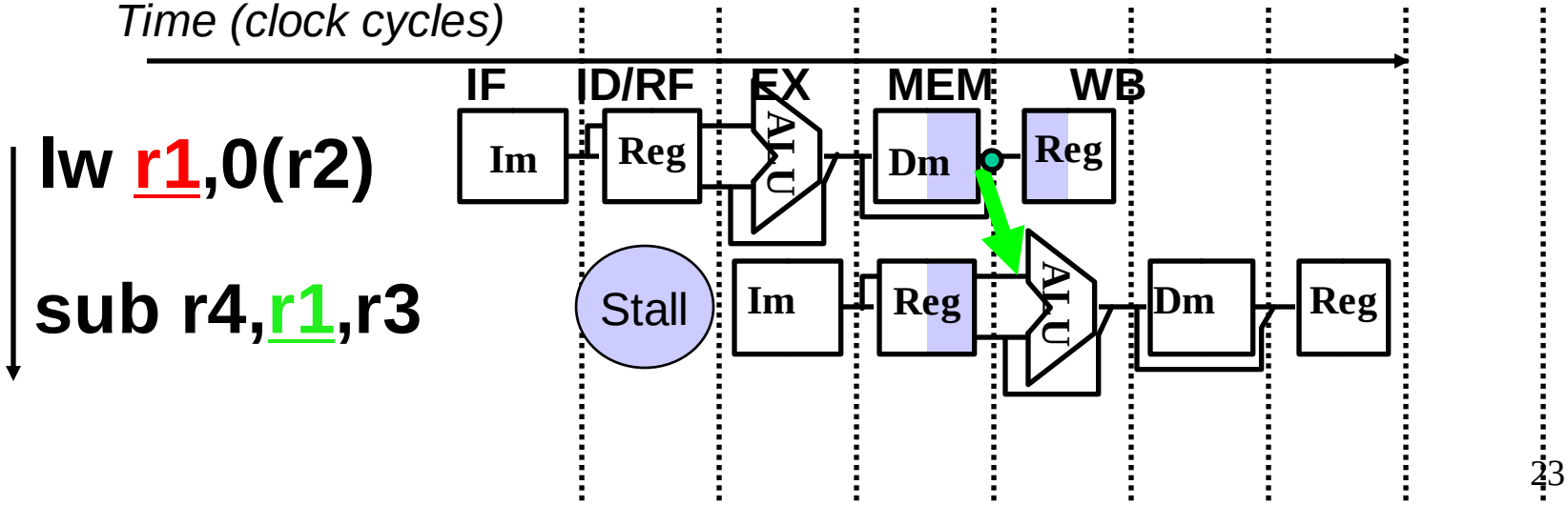



- **Forwarding result from one stage to another**
  - Adding extra hardware to retrieve the missing item early from the internal resources
- **"or" OK if read/write defined properly**
- **Forwarding can't prevent all data hazard**

# What about Loads?

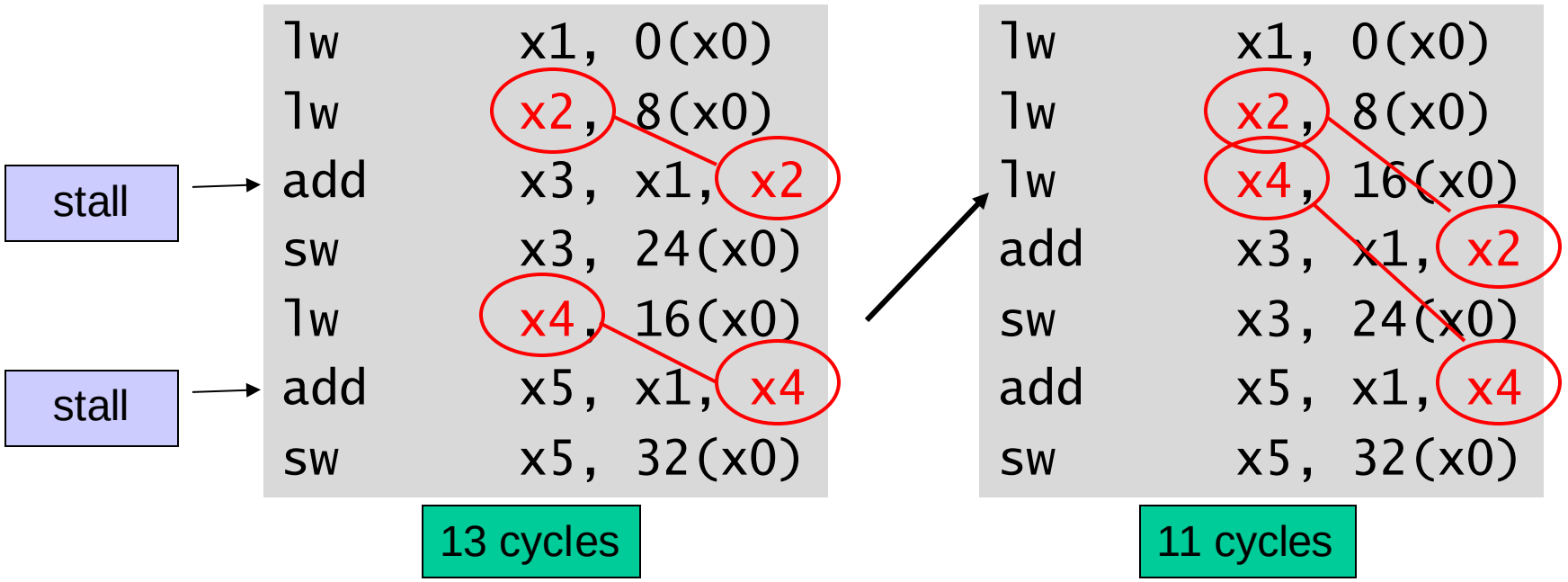



- Can't solve with forwarding
- Must delay/stall instruction dependent on loads



# Code Reordering to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `a = b + e; c = b + f;`



# Pipelining Overview Summary

- **Pipelining is a fundamental concept**
  - multiple steps using distinct resources
- **Utilize capabilities of the Datapath by pipelined instruction processing**
  - start next instruction while working on the current one
  - limited by length of longest stage (plus fill/flush)
  - detect and resolve hazards
- **What makes it easy**
  - all instructions are of the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores; Memory addresses are aligned
- **What makes it hard?**
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous one

# *Pipelining Overview Summary*

- **In modern processors, what really makes it hard:**
  - exception handling
  - trying to improve performance with out-of-order execution, etc.
- **We'll build a simple pipeline**
  - Only consider the core subset of RISC-V instructions: lw, sw, and, or, add, sub, beq