

`inst.eecs.berkeley.edu/~cs61c`
CS61C : Machine Structures

Lecture #21 CPU Design: Pipelining to Improve Performance

2007-7-31



Scott Beamer, Instructor



Review: Single cycle datapath

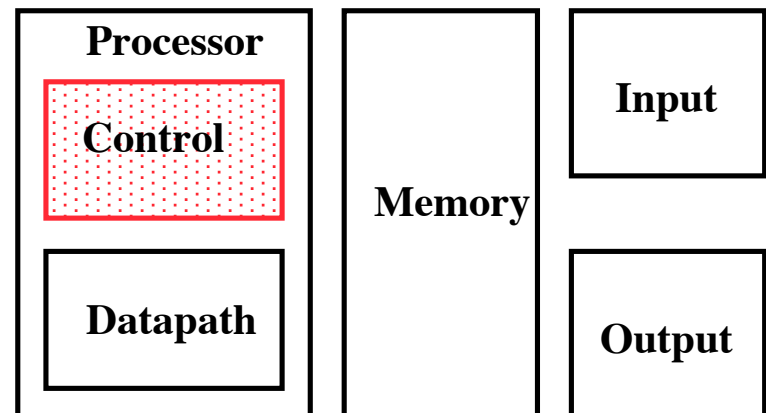
◦ 5 steps to design a processor

- 1. Analyze instruction set \Rightarrow datapath requirements
- 2. Select set of datapath components & establish clock methodology
- 3. Assemble datapath meeting the requirements
- 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
- 5. Assemble the control logic

◦ **Control** is the hard part

◦ **MIPS** makes that easier

- Instructions same size
- Source registers always in same place
- Immediates same size, location

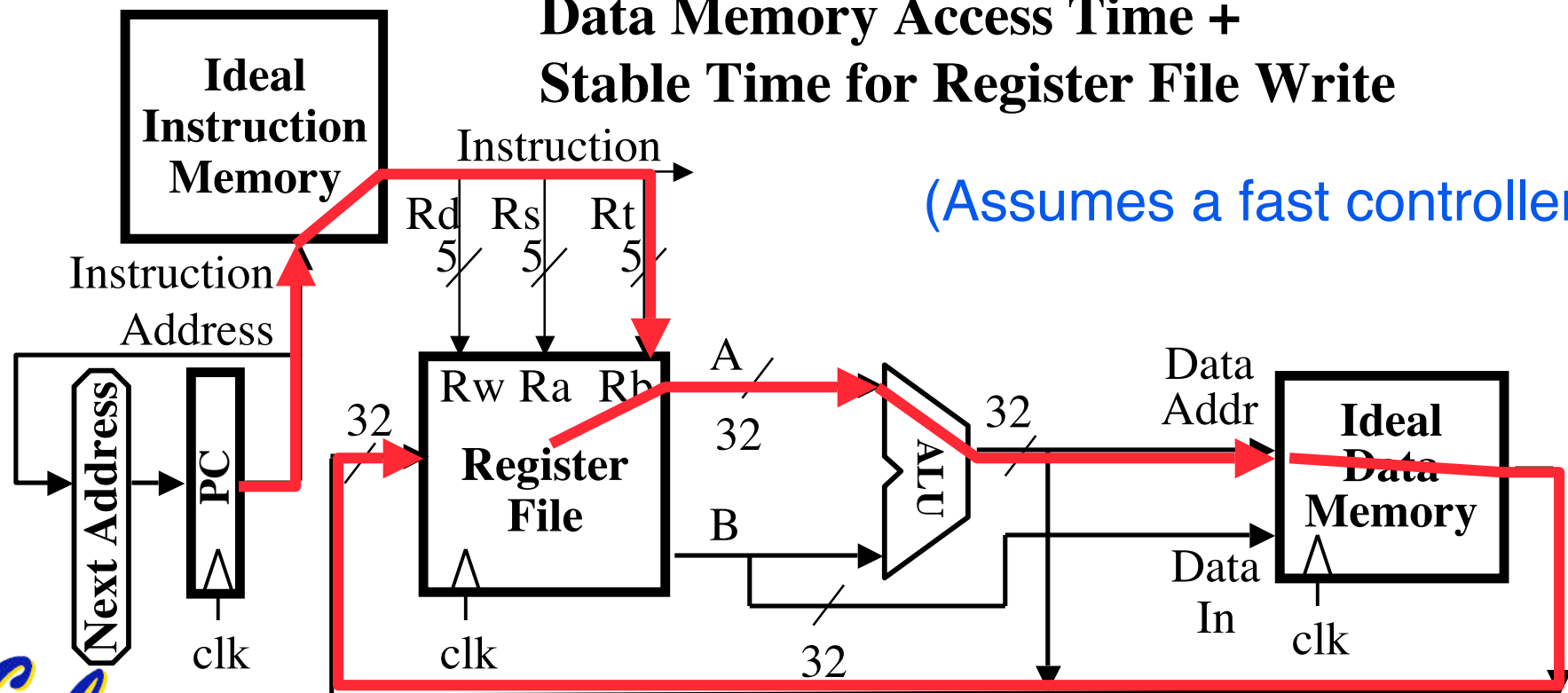


Operations always on registers/immediates

An Abstract View of the Critical Path

**Critical Path (Load Instruction) =
Delay clock through PC (FFs) +
Instruction Memory's Access Time +
Register File's Access Time, +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Stable Time for Register File Write**

(Assumes a fast controller)



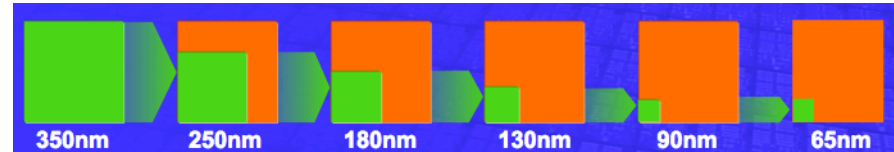
Processor Performance

- Can we estimate the clock rate (frequency) of our single-cycle processor? We know:
 - 1 cycle per instruction
 - `lw` is the most demanding instruction.
 - Assume approximate delays for major pieces of the datapath:
 - Instr. Mem, ALU, Data Mem : 2ns each, regfile 1ns
 - Instruction execution requires: $2 + 1 + 2 + 2 + 1 = 8\text{ns}$
 - $\Rightarrow 125\text{ MHz}$
- What can we do to improve clock rate?
- Will this improve performance as well?
 - We want increases in clock rate to result in programs executing quicker.



Ways to Improve Clock Frequency

- **Smaller Process Size**

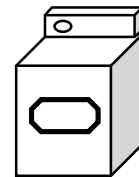
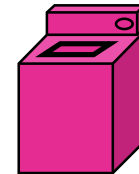
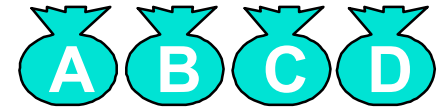


- Smallest feature possible in silicon fabrication
 - Smaller process is faster because of EE reasons, and is smaller so things are closer
- **Optimize Logic**
 - Re-arrange CL to be faster
 - Sometimes more logic can be used to reduce delay
 - **Parallel**
 - Do more at once - later...
 - **Cut Down Length of Critical Path**
 - Inserting registers (pipelining) to break up CL

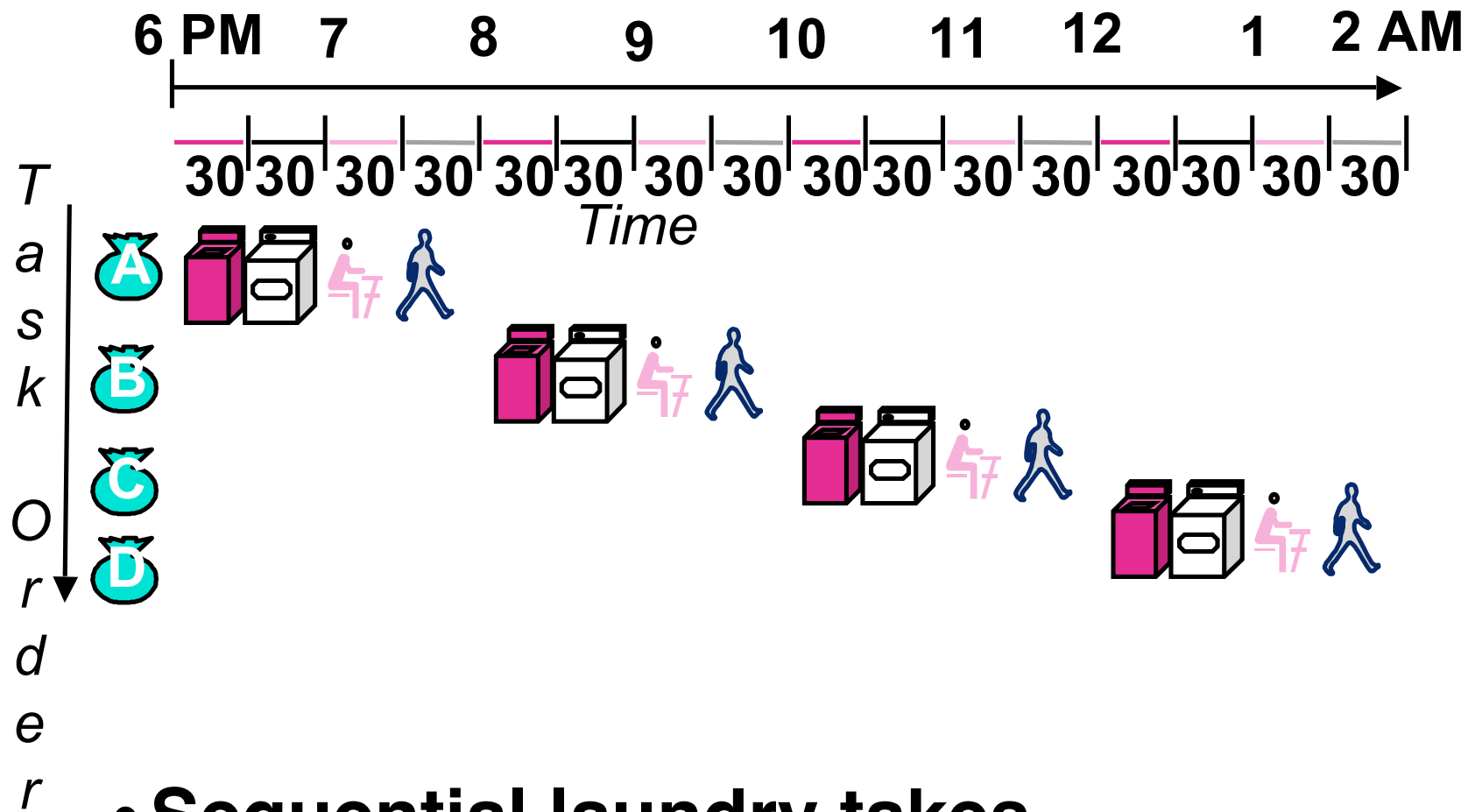


Gotta Do Laundry

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away
- Washer takes 30 minutes
- Dryer takes 30 minutes
- “Folder” takes 30 minutes
- “Stasher” takes 30 minutes to put clothes into drawers



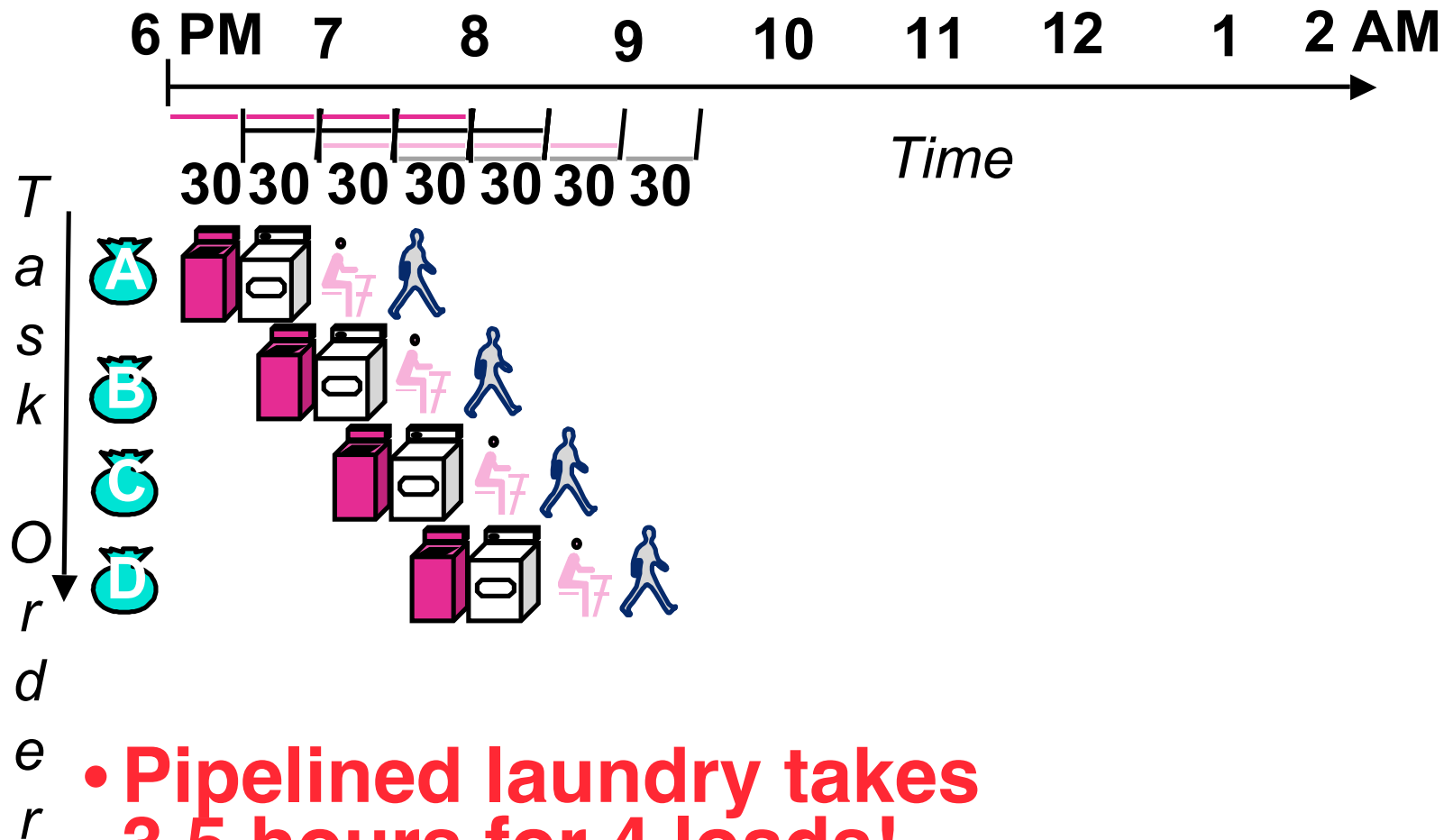
Sequential Laundry



- Sequential laundry takes 8 hours for 4 loads



Pipelined Laundry

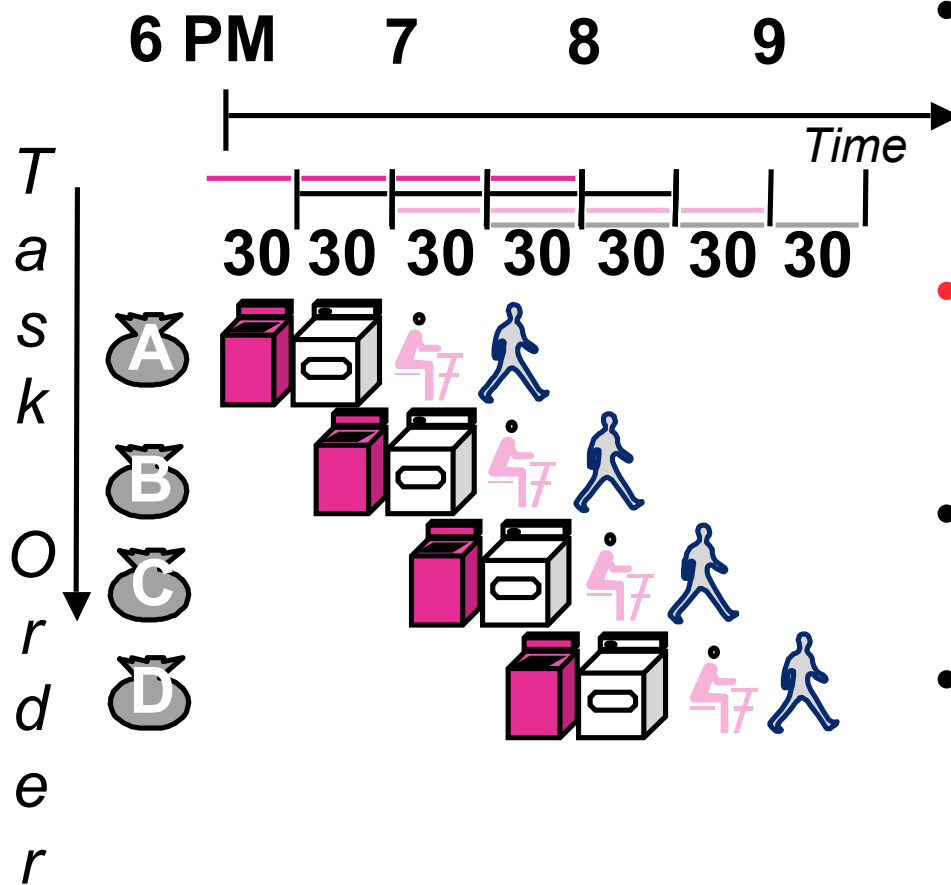


General Definitions

- **Latency**: time to completely execute a certain task (delay)
 - for example, time to read a sector from disk is disk access time or disk latency
- **Throughput**: amount of work that can be done over a period of time (rate)

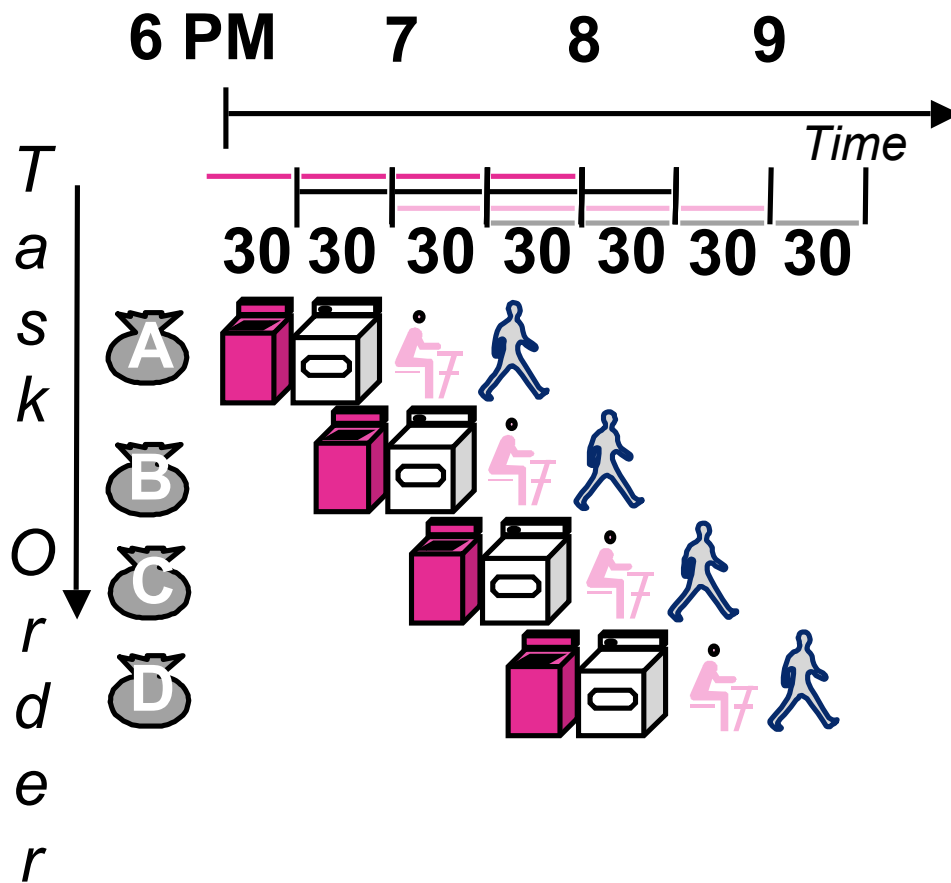


Pipelining Lessons (1/2)



- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- **Multiple** tasks operating simultaneously using different resources
- Potential speedup = **Number pipe stages**
- Time to “**fill**” pipeline and time to “**drain**” it reduces speedup: 2.3X v. 4X in this example

Pipelining Lessons (2/2)



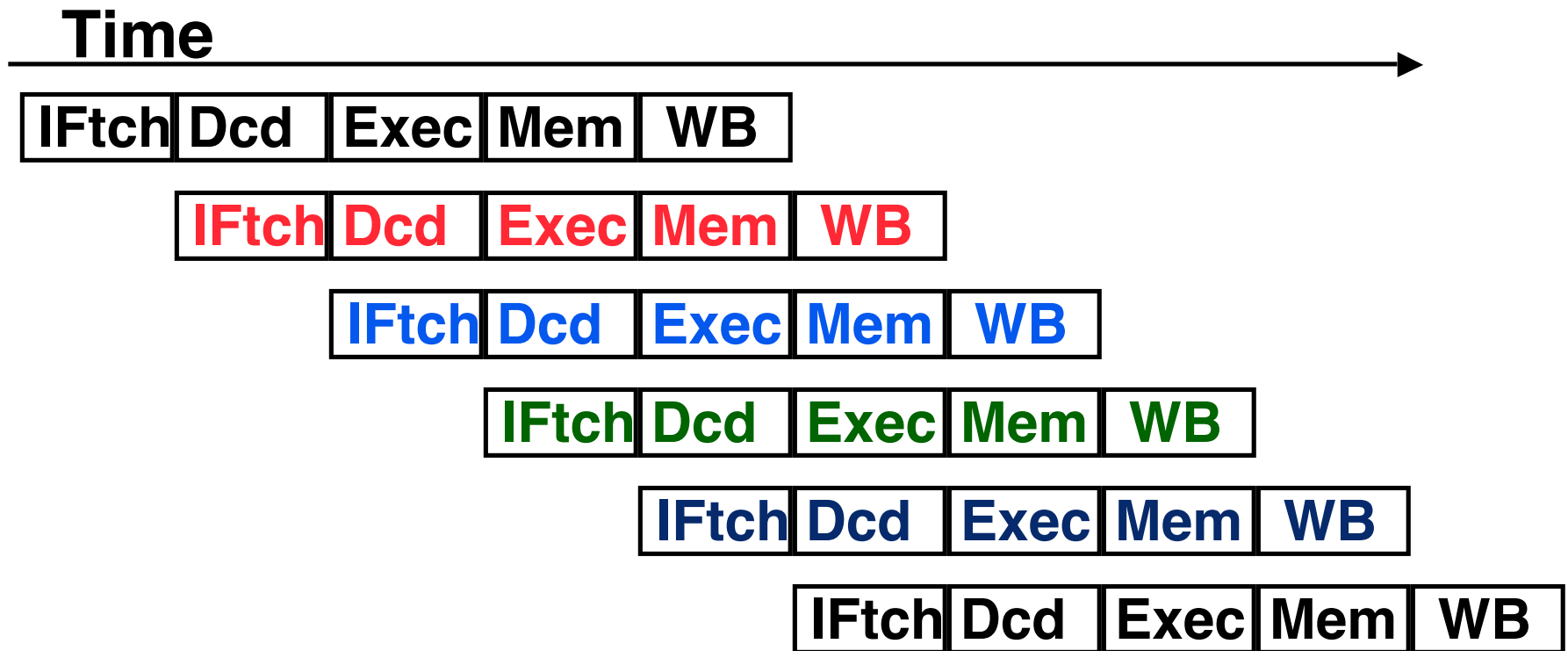
- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup

Steps in Executing MIPS

- 1) **IFtch**: Instruction Fetch, Increment PC
- 2) **Dcd**: Instruction Decode, Read Registers
- 3) **Exec**:
Mem-ref: Calculate Address
Arith-log: Perform Operation
- 4) **Mem**:
Load: Read Data from Memory
Store: Write Data to Memory
- 5) **WB**: Write Data Back to Register



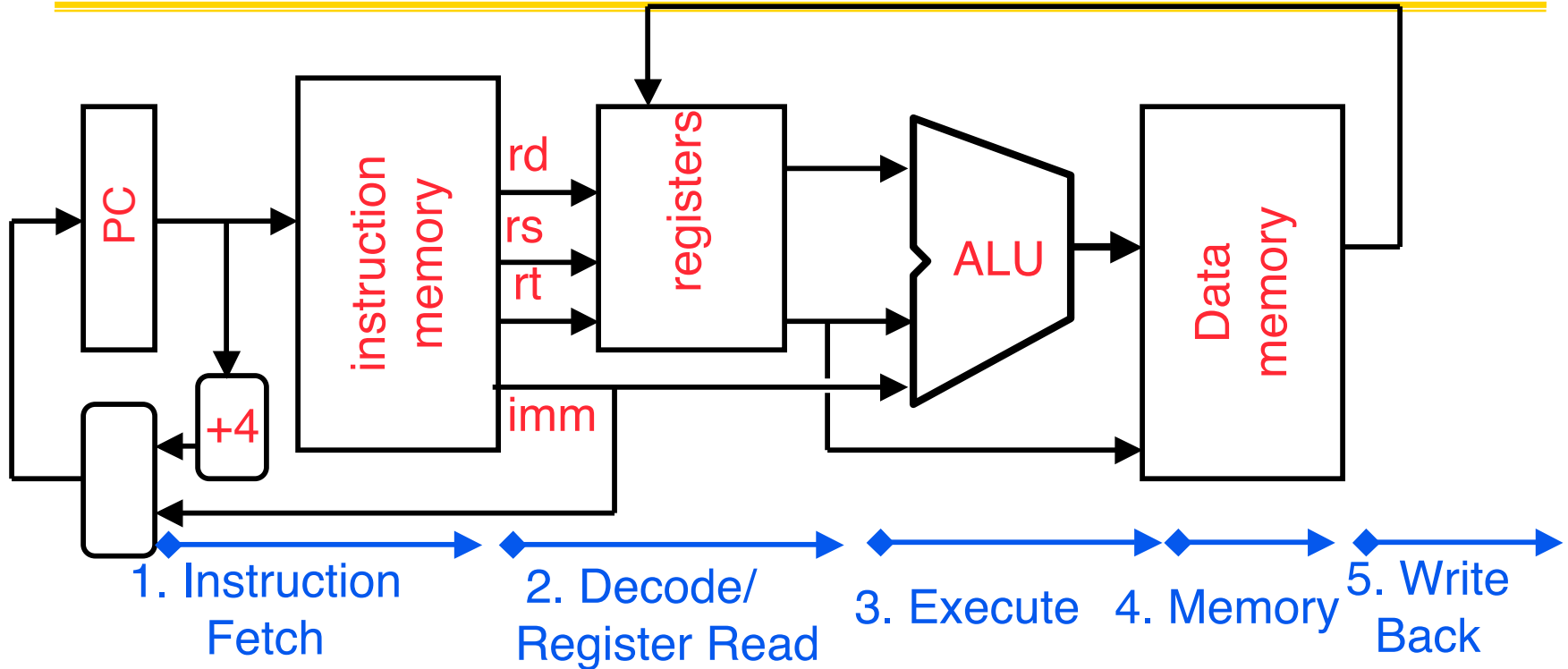
Pipelined Execution Representation



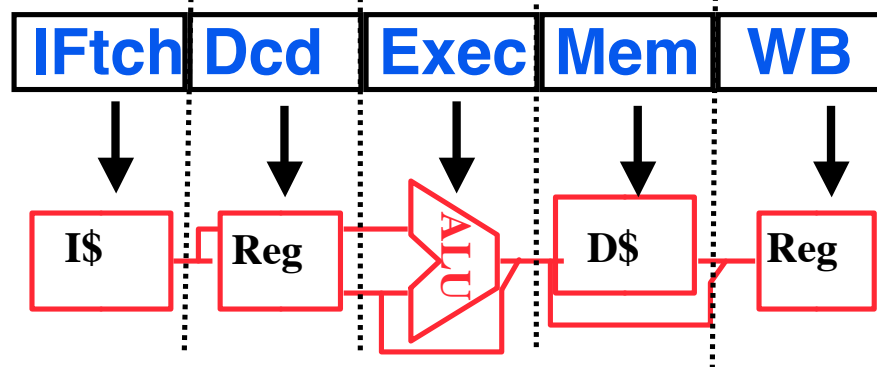
- Every instruction must take same number of steps, also called pipeline “**stages**”, so some will go idle sometimes



Review: Datapath for MIPS

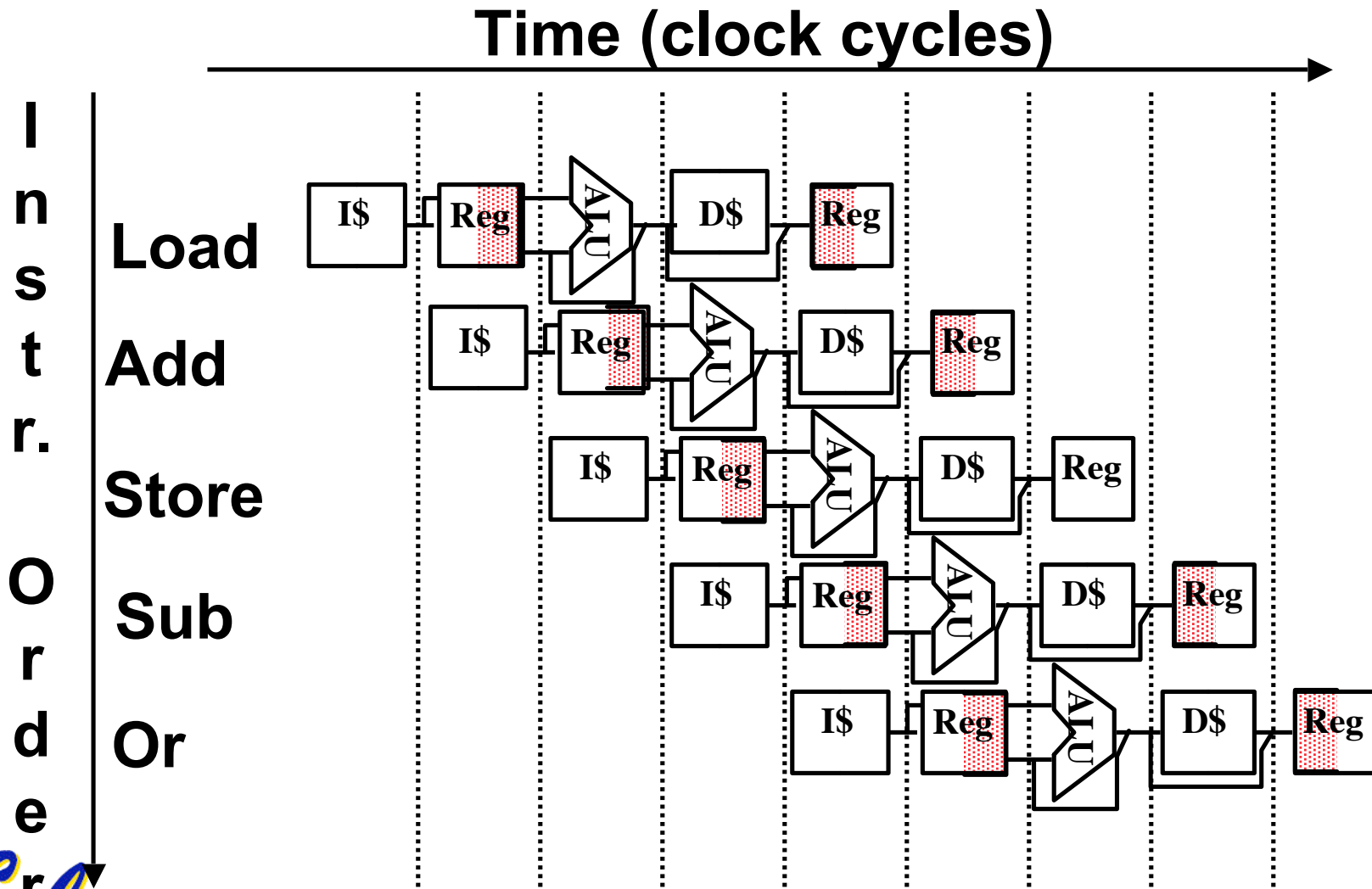


- Use datapath figure to represent pipeline



Graphical Pipeline Representation

(In Reg, right half highlight read, left half write)



Example

- Suppose 2 ns for memory access, 2 ns for ALU operation, and 1 ns for register file read or write; compute instr rate

- **Nonpipelined Execution:**

- **lw** : IF + Read Reg + ALU + Memory + Write Reg = 2 + 1 + 2 + 2 + 1 = 8 ns

- **add**: IF + Read Reg + ALU + Write Reg = 2 + 1 + 2 + 1 = 6 ns
(recall 8ns for single-cycle processor)

- **Pipelined Execution:**

- Max(IF, Read Reg, ALU, Memory, Write Reg) = 2 ns

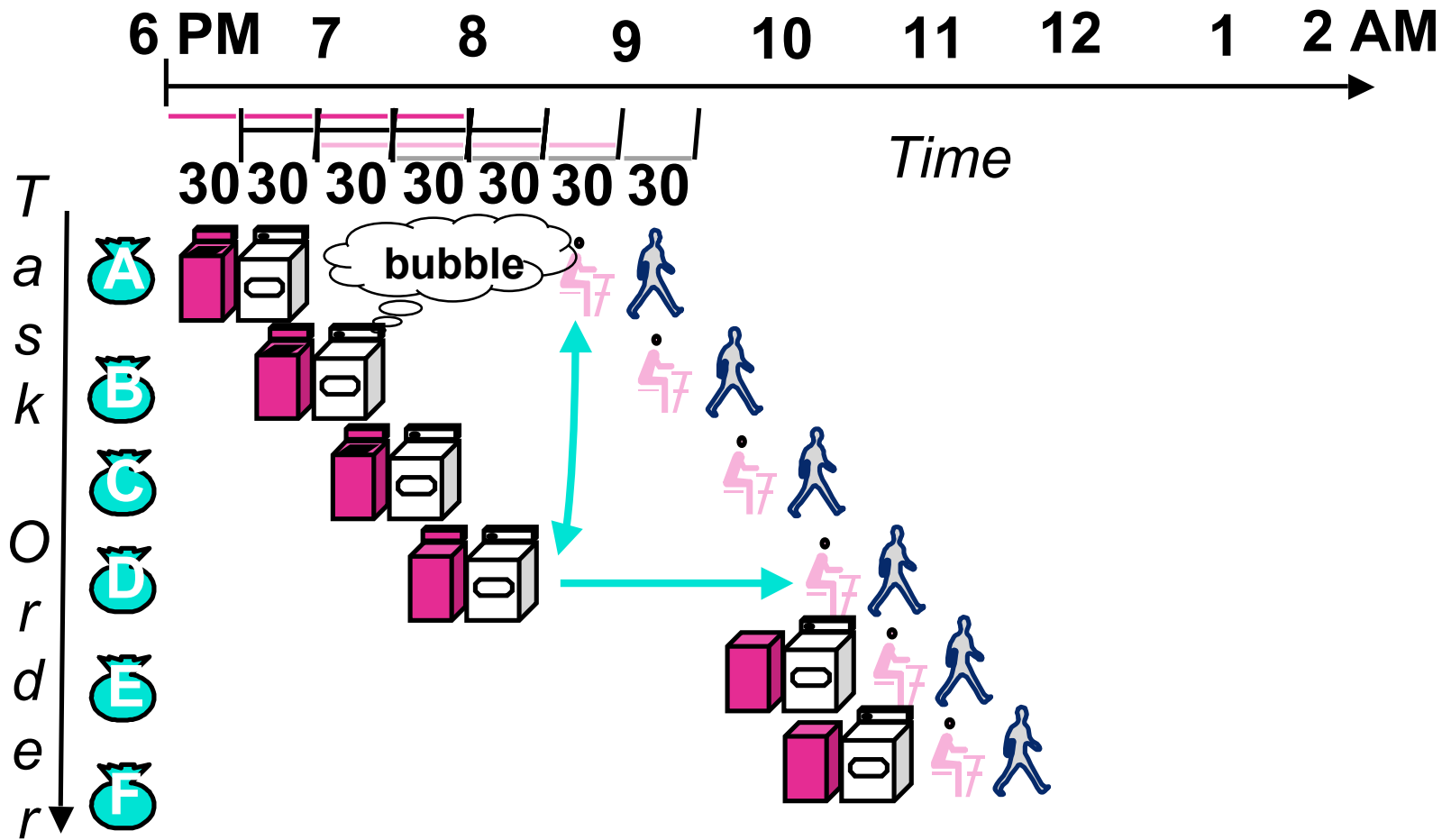


Administrivia

- **Assignments**
 - HW7 due 8/2
 - Proj3 due 8/5
- **Midterm Regrades due Wed 8/1**
- **Logisim in lab is now 2.1.6**
- **Valerie's OH on Thursday moved to 10-11 for this week**



Pipeline Hazard: Matching socks in later load



A depends on D; **stall** since folder tied up

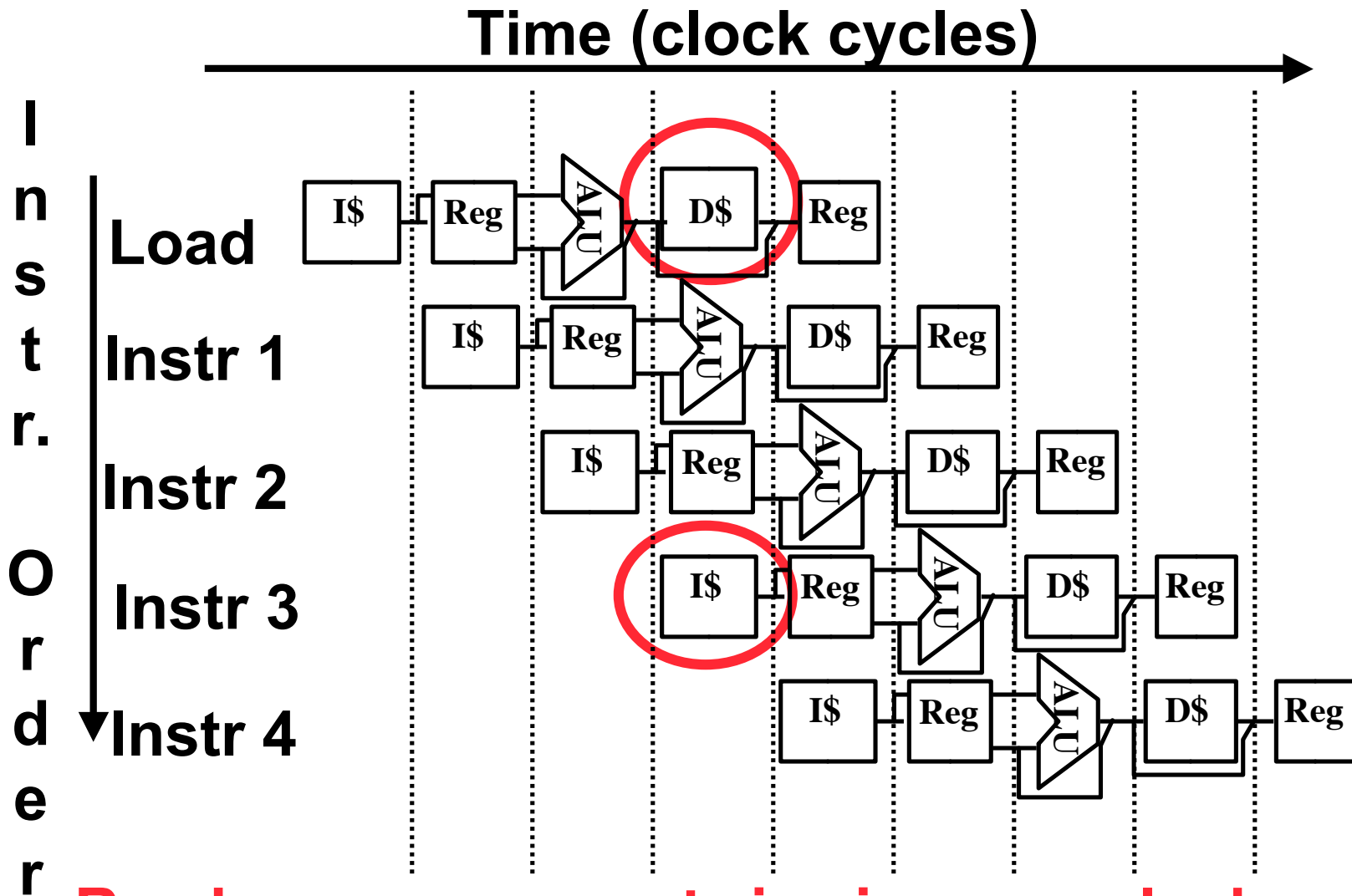


Problems for Pipelining CPUs

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
 - **Structural hazards**: HW cannot support some combination of instructions (single person to fold and put clothes away)
 - **Control hazards**: Pipelining of branches causes later instruction fetches to wait for the result of the branch
 - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
- These might result in pipeline **stalls** or “**bubbles**” in the pipeline.



Structural Hazard #1: Single Memory (1/2)



Read same memory twice in same clock cycle



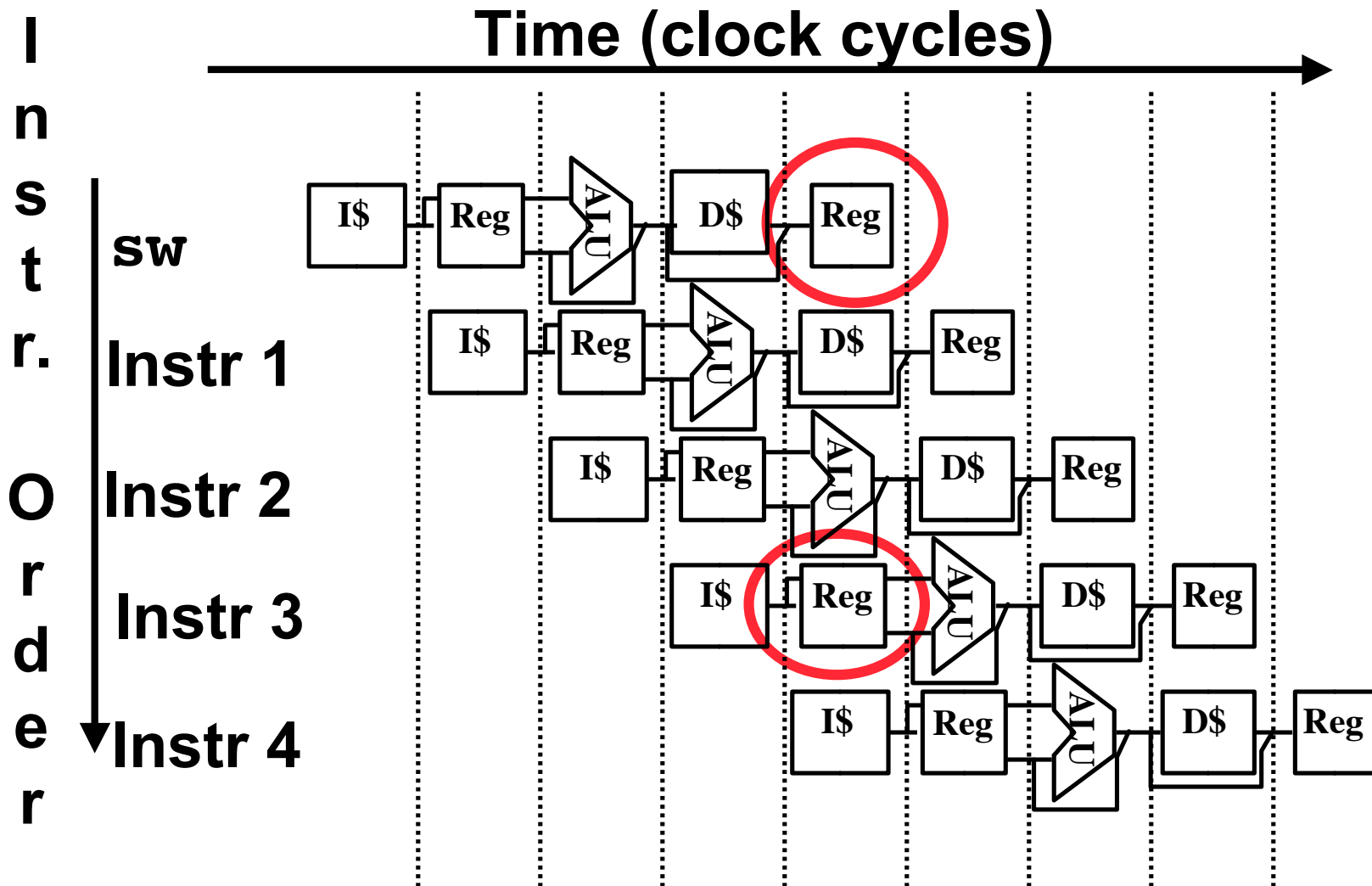
Structural Hazard #1: Single Memory (2/2)

- **Solution:**

- infeasible and inefficient to create second memory
- (We'll learn about this more next week)
- so simulate this by having two Level 1 Caches (a temporary smaller [of usually most recently used] copy of memory)
- have both an L1 Instruction Cache and an L1 Data Cache
- need more complex hardware to control when both caches miss



Structural Hazard #2: Registers (1/2)



Can we read and write to registers simultaneously?



Structural Hazard #2: Registers (2/2)

- **Two different solutions have been used:**
 - 1) **RegFile access is *VERY* fast: takes less than half the time of ALU stage**
 - **Write to Registers during first half of each clock cycle**
 - **Read from Registers during second half of each clock cycle**
 - 2) **Build RegFile with independent read and write ports**
- **Result: can perform Read and Write during same clock cycle**



Data Hazards (1/2)

- Consider the following sequence of instructions

add \$t0, \$t1, \$t2

sub \$t4, \$t0, \$t3

and \$t5, \$t0, \$t6

or \$t7, \$t0, \$t8

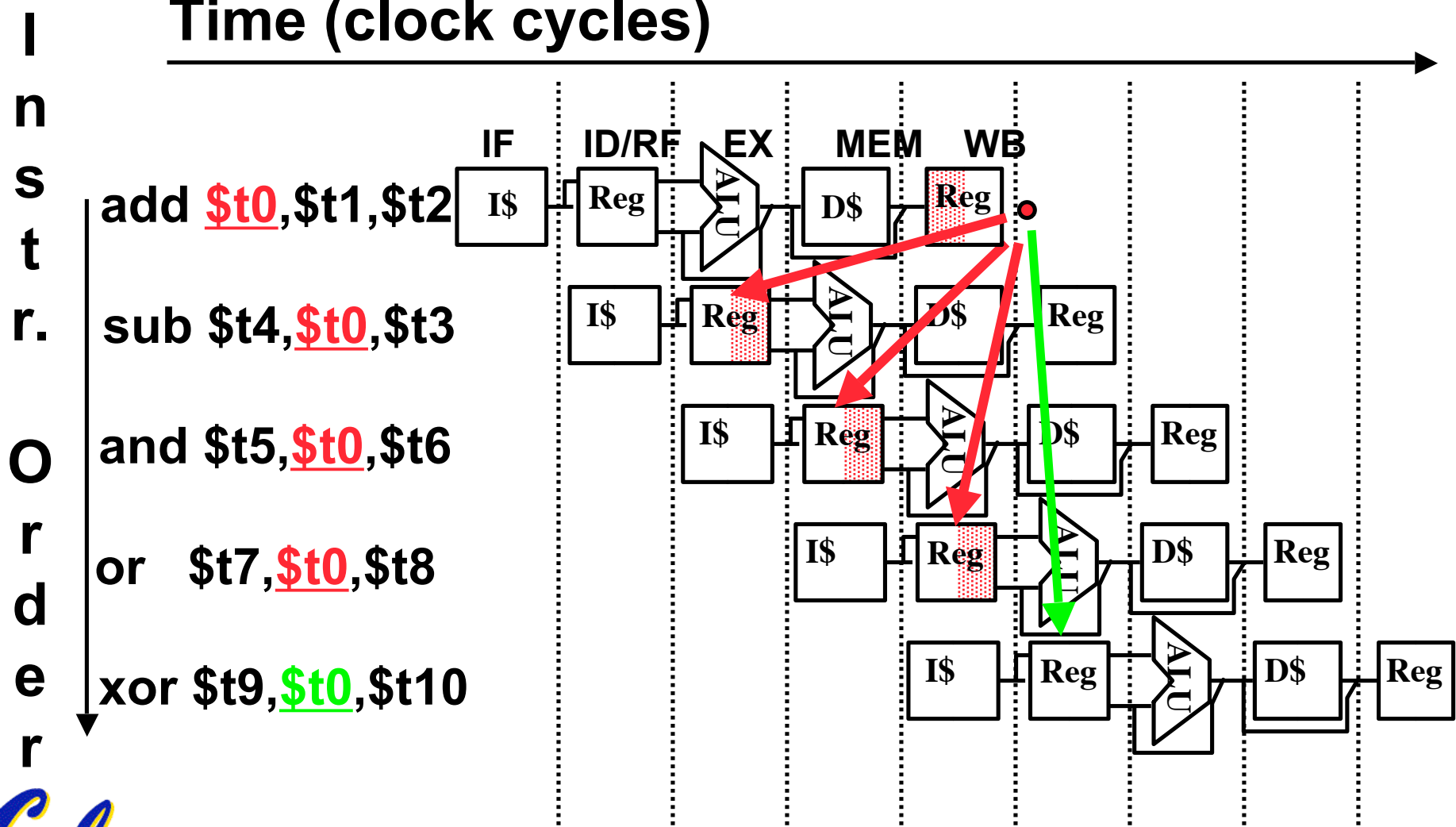
xor \$t9, \$t0, \$t10



Data Hazards (2/2)

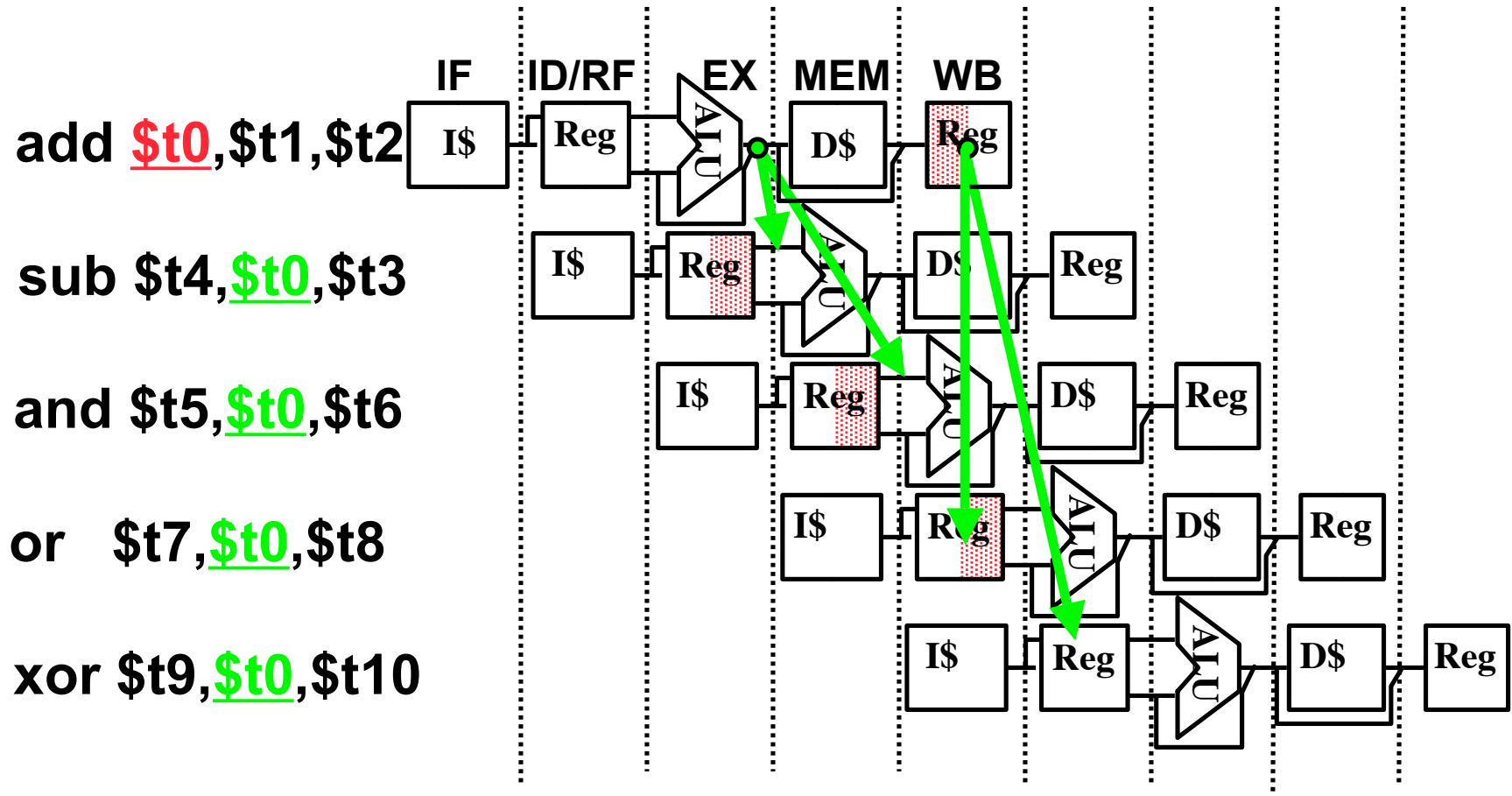
Data-flow backward in time are hazards

Time (clock cycles)



Data Hazard Solution: Forwarding

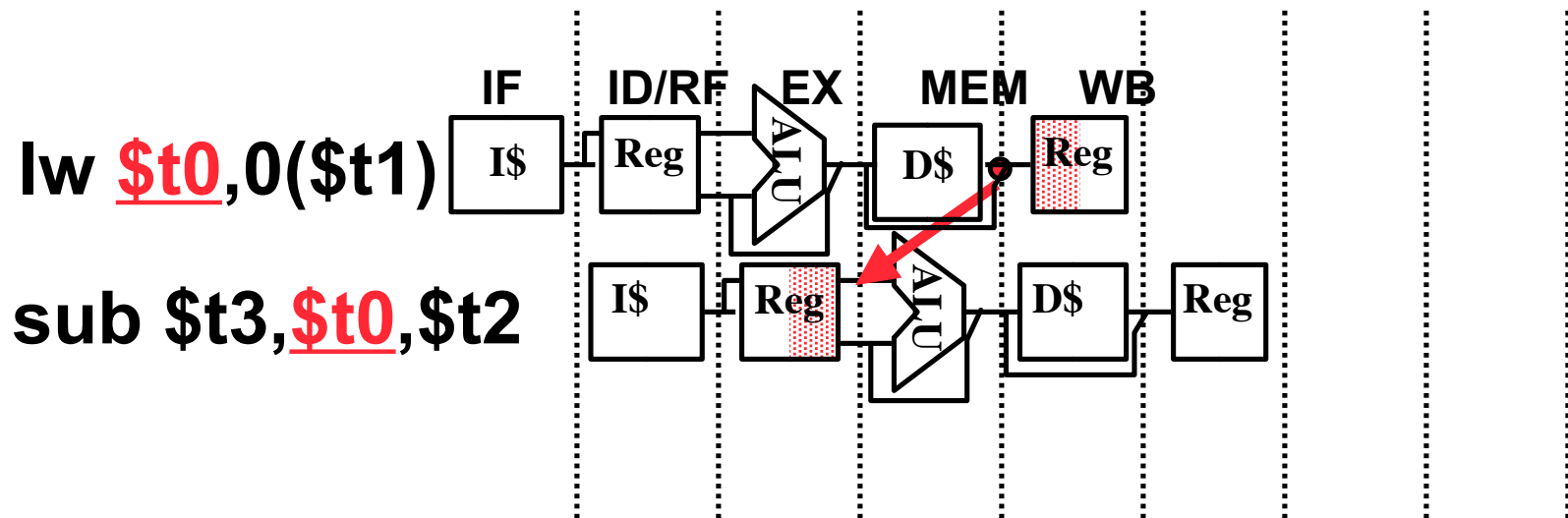
- **Forward** result from one stage to another



“or” hazard solved by register hardware

Data Hazard: Loads (1/4)

- Dataflow backwards in time are hazards



- Can't solve all cases with forwarding
- Must stall instruction dependent on load, then forward (more hardware)



Data Hazard: Loads (2/4)

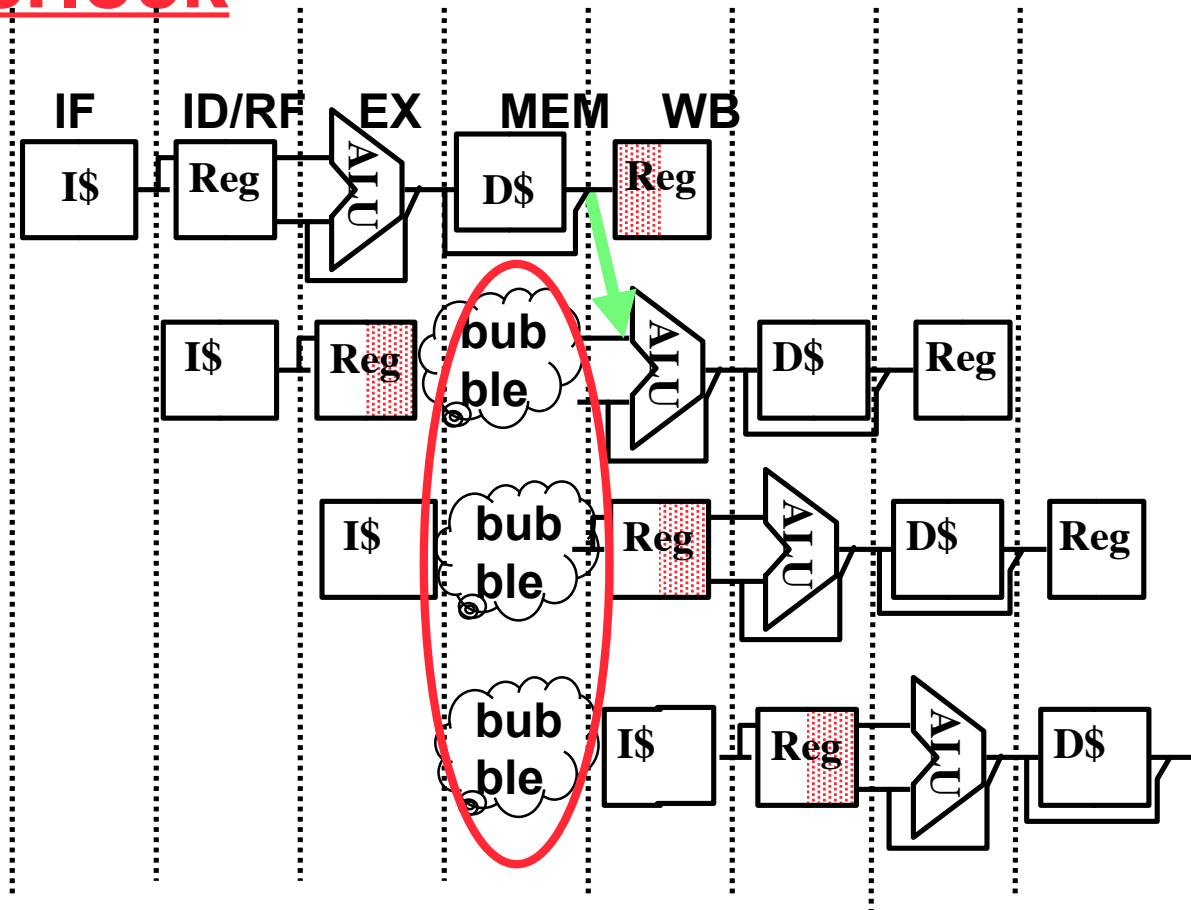
- **Hardware stalls pipeline**
- Called “**interlock**”

lw \$t0, 0(\$t1)

sub \$t3, \$t0, \$t2

and \$t5, \$t0, \$t4

or \$t7, \$t0, \$t6



Data Hazard: Loads (3/4)

- Instruction slot after a load is called **“load delay slot”**
- If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.
- If the compiler puts an unrelated instruction in that slot, then no stall
- Letting the hardware stall the instruction in the delay slot is equivalent to putting a nop in the slot (except the latter uses more code space)



Historical Trivia

- **First MIPS design did not interlock and stall on load-use data hazard**
- **Real reason for name behind MIPS:**
Microprocessor without
Interlocked
Pipeline
Stages
 - **Word Play on acronym for Millions of Instructions Per Second, *also* called MIPS**



Peer Instruction

- A. Thanks to pipelining, I have reduced the time it took me to wash my shirt.
- B. Longer pipelines are always a win (since less work per stage & a faster clock).
- C. We can rely on compilers to help us avoid data hazards by reordering instrs.

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TF
7:	TTT



Peer Instruction Answer

A. Throughput better, not execution time

B. "...longer pipelines do usually mean faster clock, but branches cause problems!"

C. "they happen too often & delay too long."
Forwarding! (e.g, Mem \Rightarrow ALU)

A. Thanks to **FAL**SE pipelining, I have reduced the time it took me to wash my shirt

B. Longer pipelines are always a win (since less work per stage & a faster clock). **FAL**SE

C. We can rely on compiler to help us avoid data hazards by reordering instrs. **FAL**SE

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TFE
7:	TTT



Things to Remember

- **Optimal Pipeline**

- Each stage is executing part of an instruction each clock cycle.
- One instruction finishes during each clock cycle.
- On average, execute far more quickly.

- **What makes this work?**

- Similarities between instructions allow us to use same stages for all instructions (generally).
- Each stage takes about the same amount of time as all others: little wasted time.

