

CS61C : Machine Structures

Lecture 5.2.2 Pipelining I

2004-07-22

Kurt Meinz

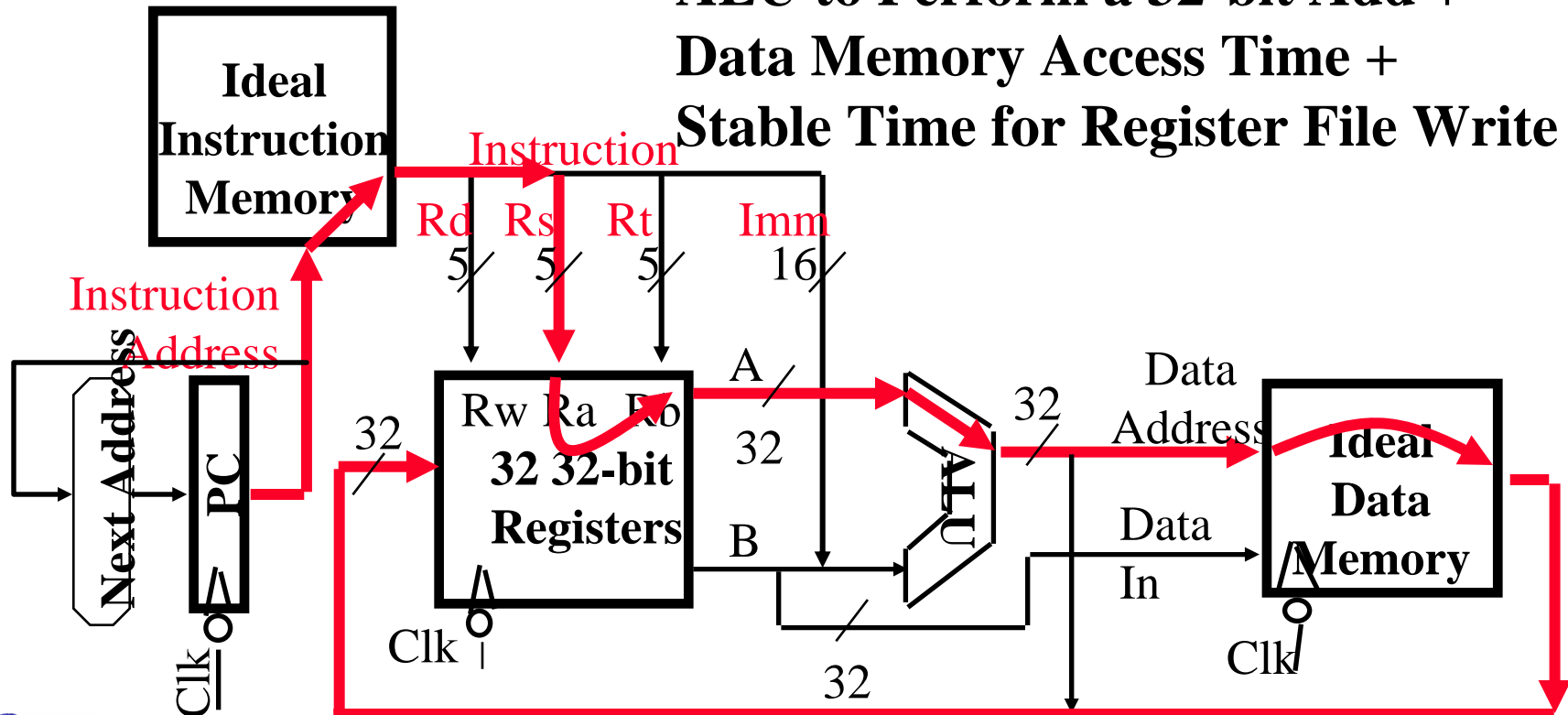
`inst.eecs.berkeley.edu/~cs61c`



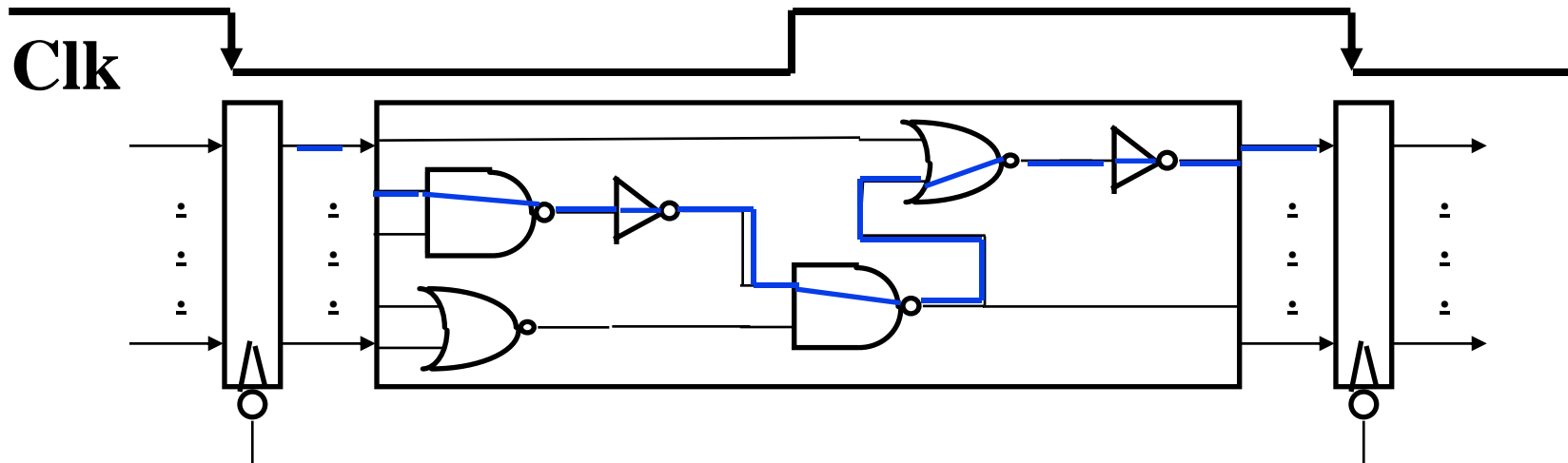
An Abstract View of the Critical Path

- This affects how much you can overclock your PC!

Critical Path (Load Operation) =
 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



Improve Critical Path → Improve Clock



- “**Critical path**” (longest path through logic) determines length of clock period
- To reduce clock period → decrease path through CL by inserting State

Review: Single cycle datapath

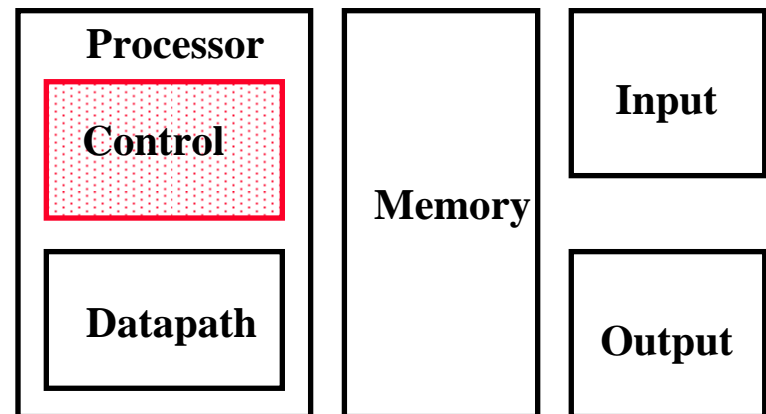
◦ 5 steps to design a processor

- 1. Analyze instruction set => 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

Review Datapath (1/3)

- **Datapath is the hardware that performs operations necessary to execute programs.**
- **Control instructs datapath on what to do next.**
- **Datapath needs:**
 - **access to storage (general purpose registers and memory)**
 - **computational ability (ALU)**
 - **helper hardware (local registers and PC)**

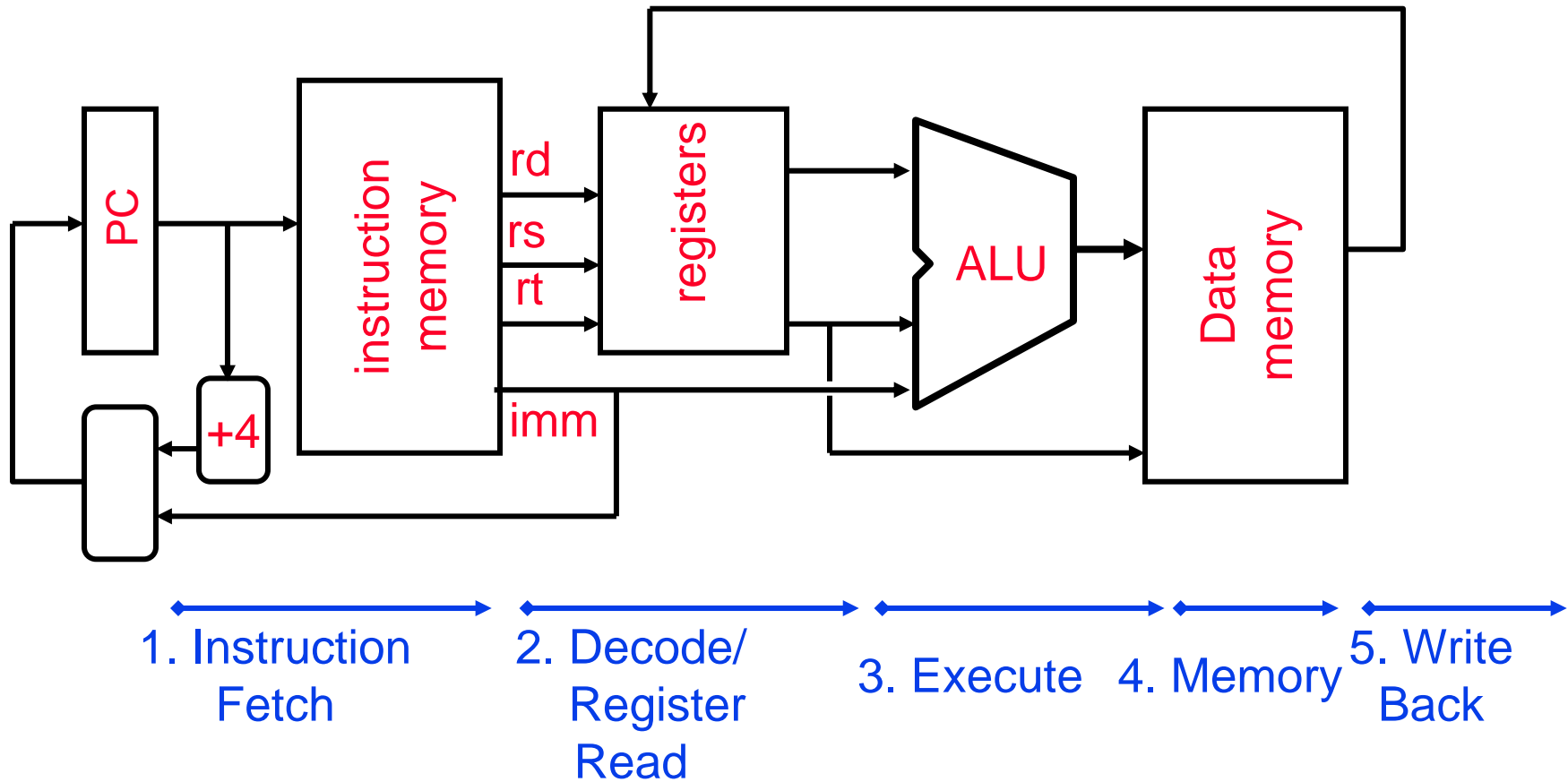


Review Datapath (2/3)

- **Five stages of datapath (executing an instruction):**
 1. **Instruction Fetch (Increment PC)**
 2. **Instruction Decode (Read Registers)**
 3. **ALU (Computation)**
 4. **Memory Access**
 5. **Write to Registers**
- **ALL instructions must go through ALL five stages.**

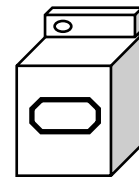
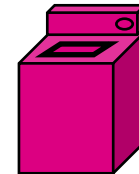
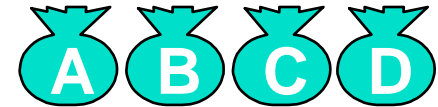


Review Datapath (3/3)

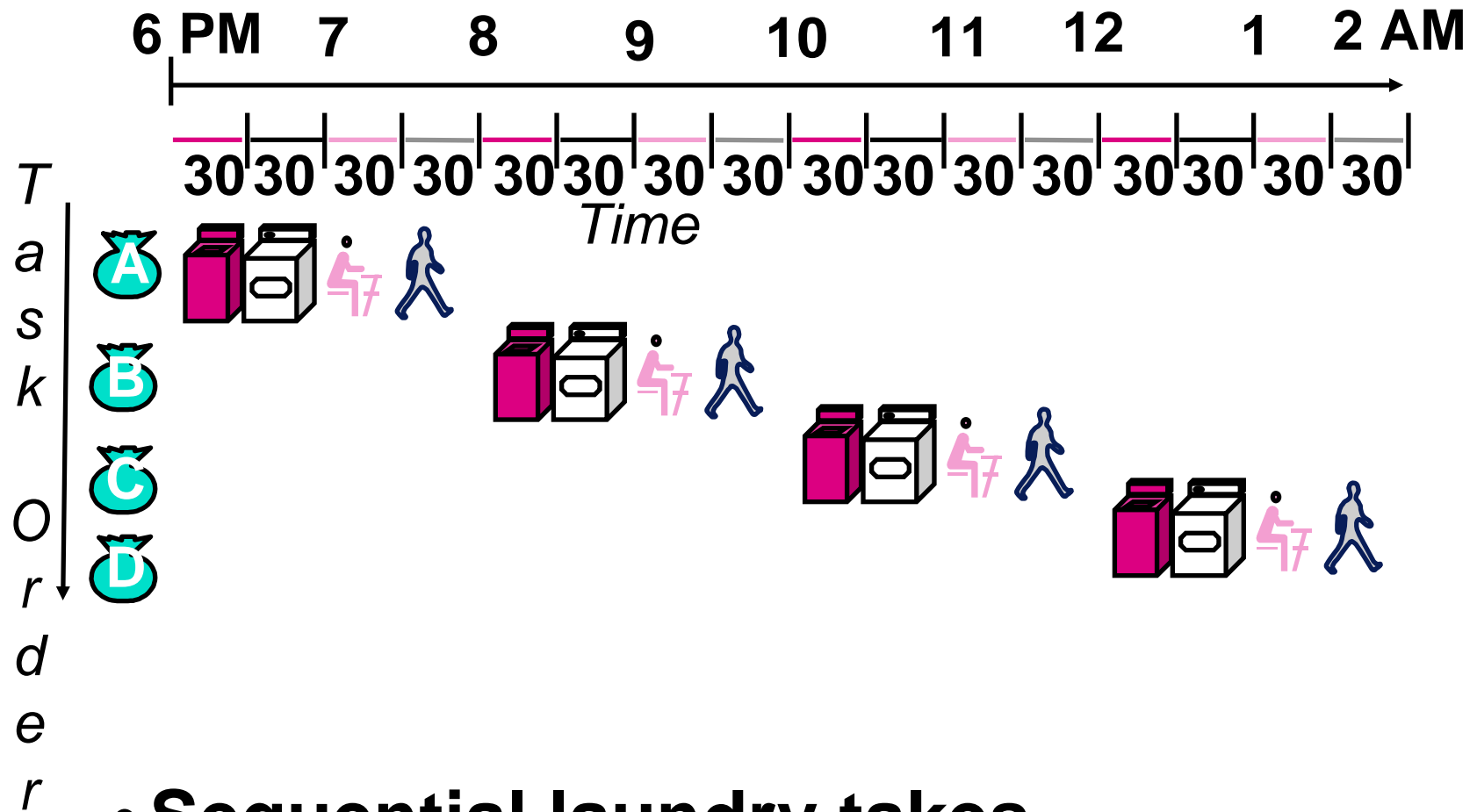


Gotta Do Laundry

- 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



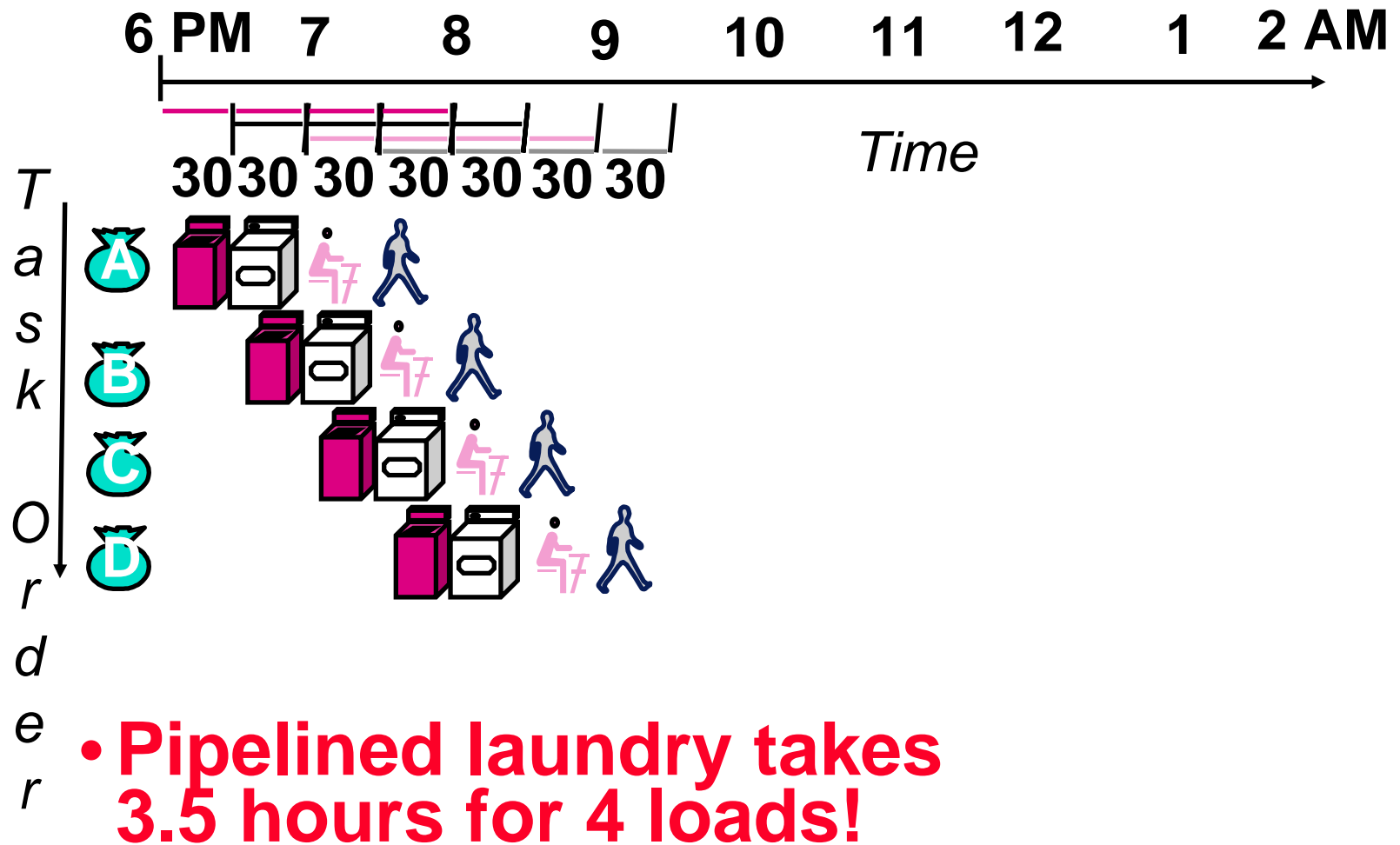
Sequential Laundry



- Sequential laundry takes 8 hours for 4 loads



Pipelined Laundry

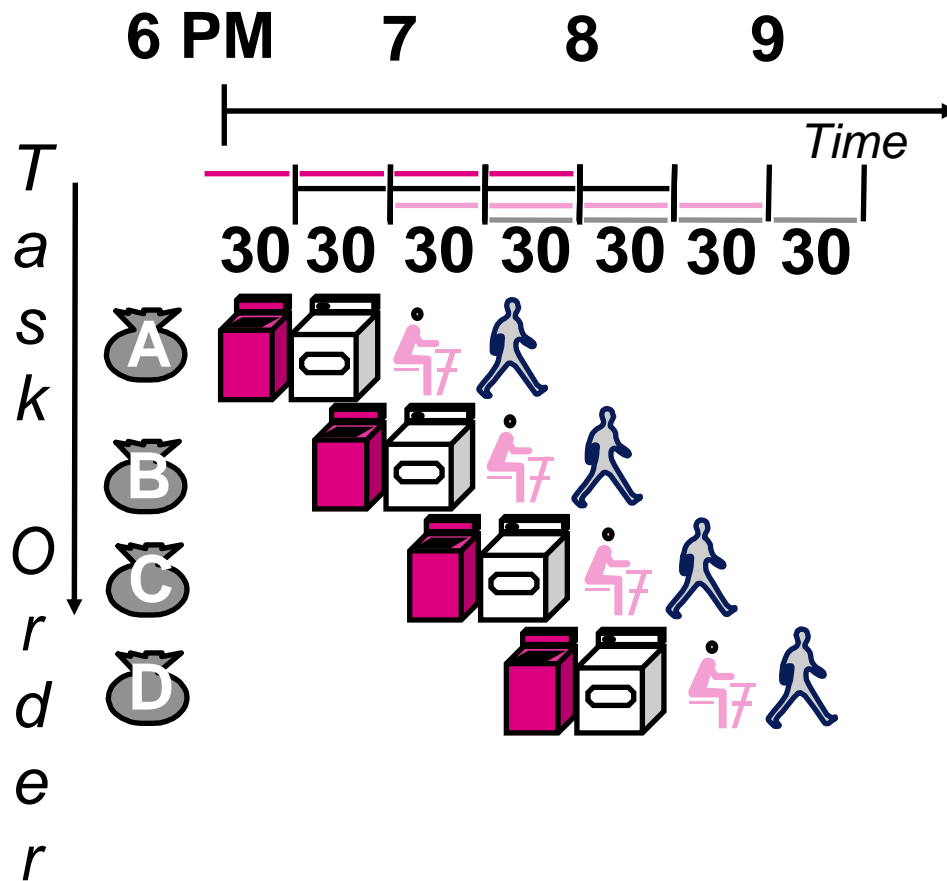


General Definitions

- **Latency**: time to completely execute a certain task
 - for example, time to read a sector from disk is disk access time or disk latency
 - Instruction latency is time from when instruction starts to time when it finishes.
- **Throughput**: amount of work that can be done over a period of time



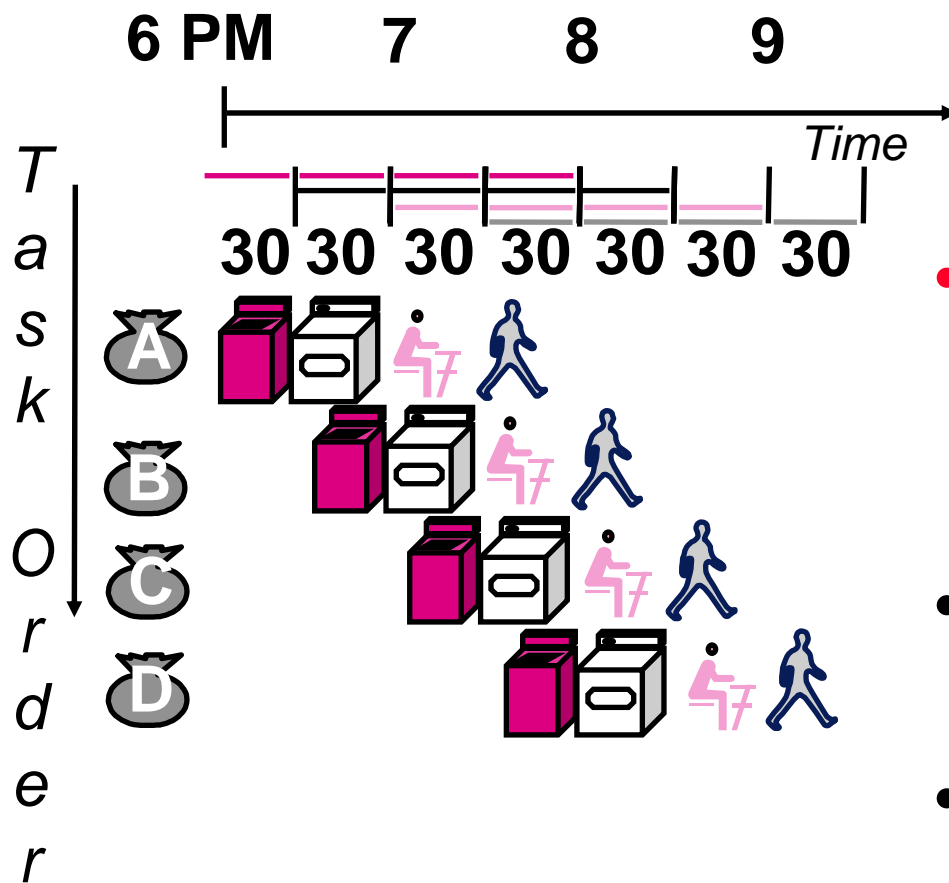
Pipelining Lessons (0/2)



• Terminology:

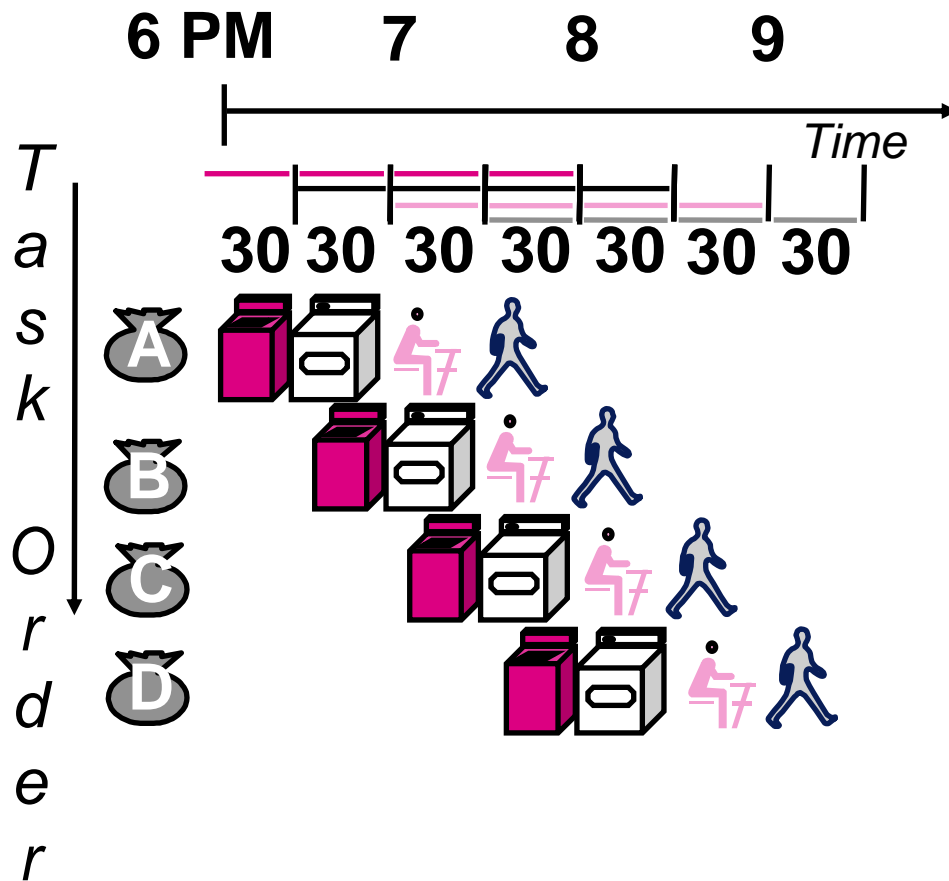
- **Issue:** When instruction goes into first stage of pipe.
- **Commit:** when instruction finishes last stage

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)



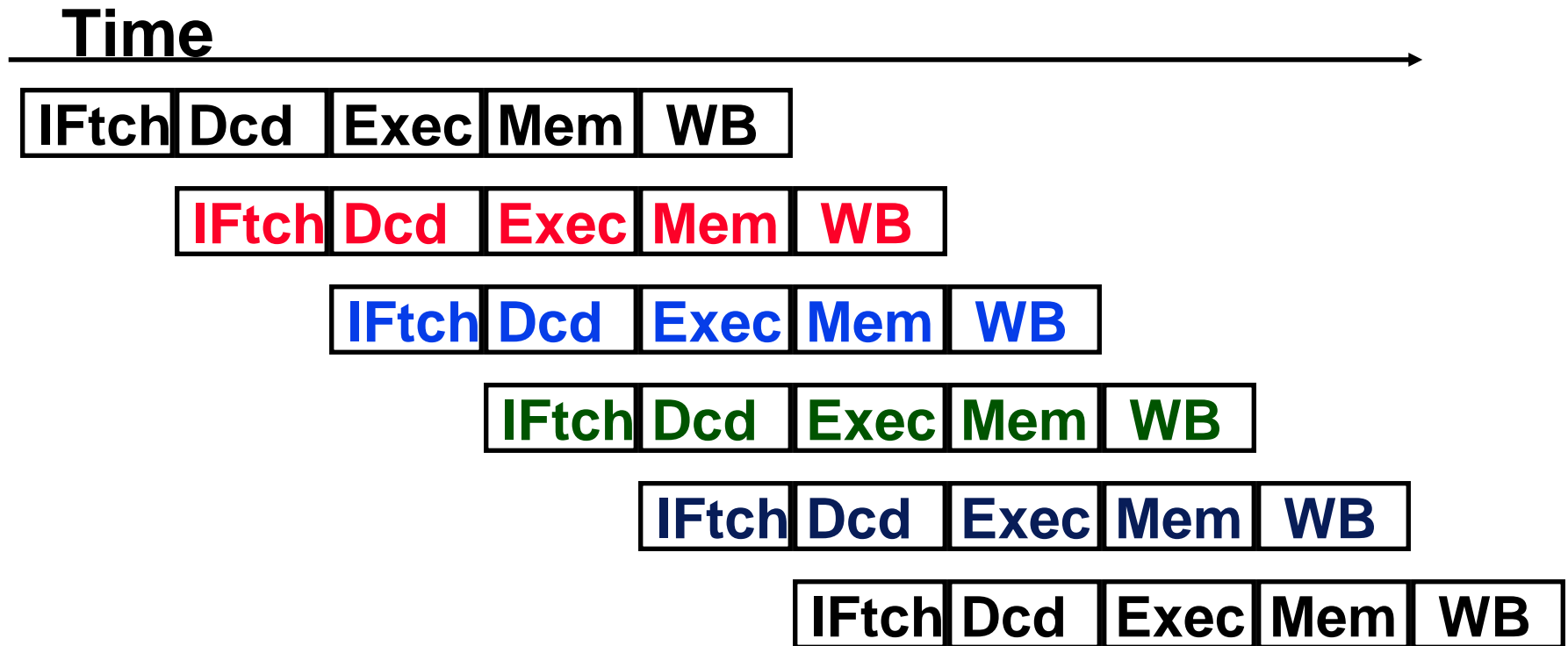
- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
- Pipeline rate limited by **slowest** pipeline stage
- Unbalanced lengths of pipe stages also reduces speedup

Steps in Executing MIPS

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



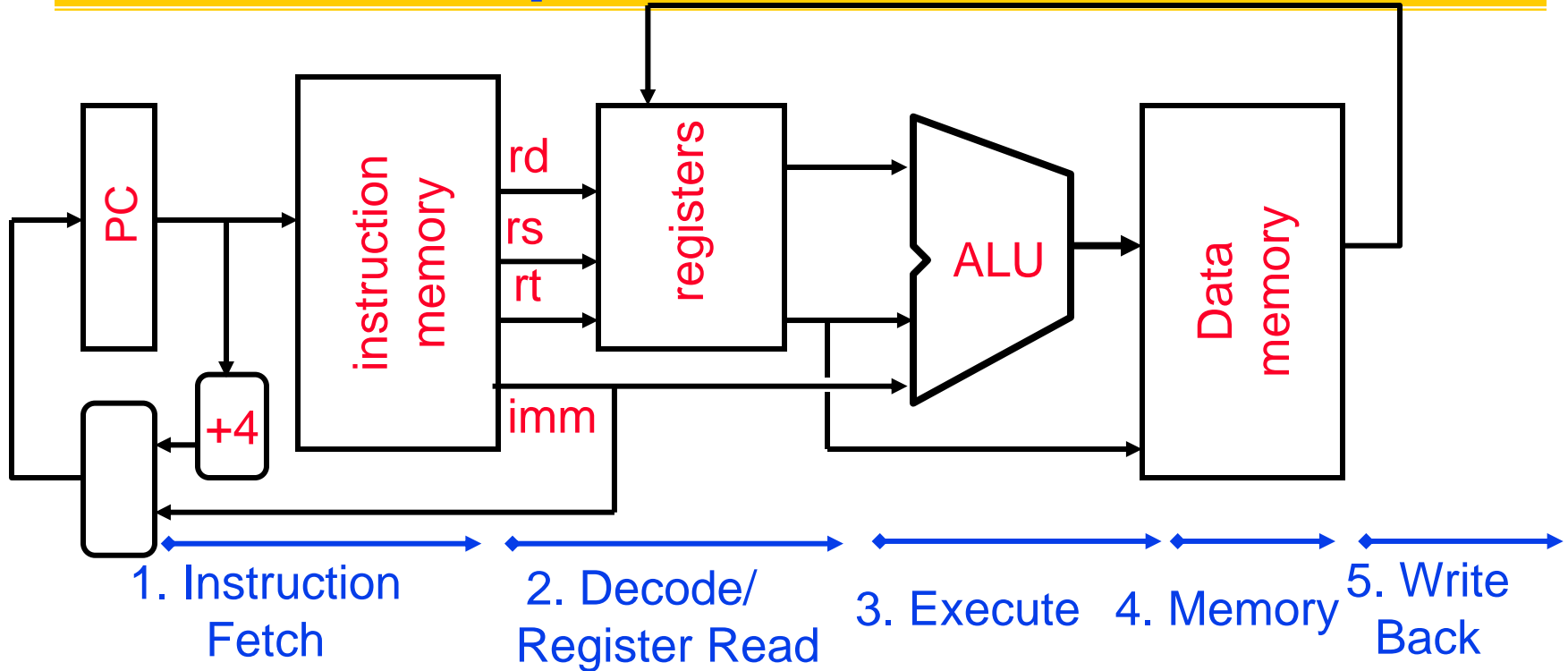
Pipelined Execution Representation



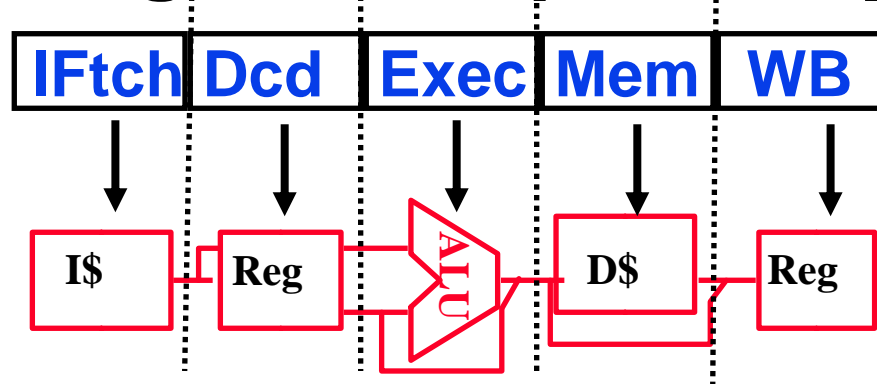
- Every instruction must take same number of steps, also called pipeline “stages”, so some will go idle sometimes



Review: Datapath for MIPS



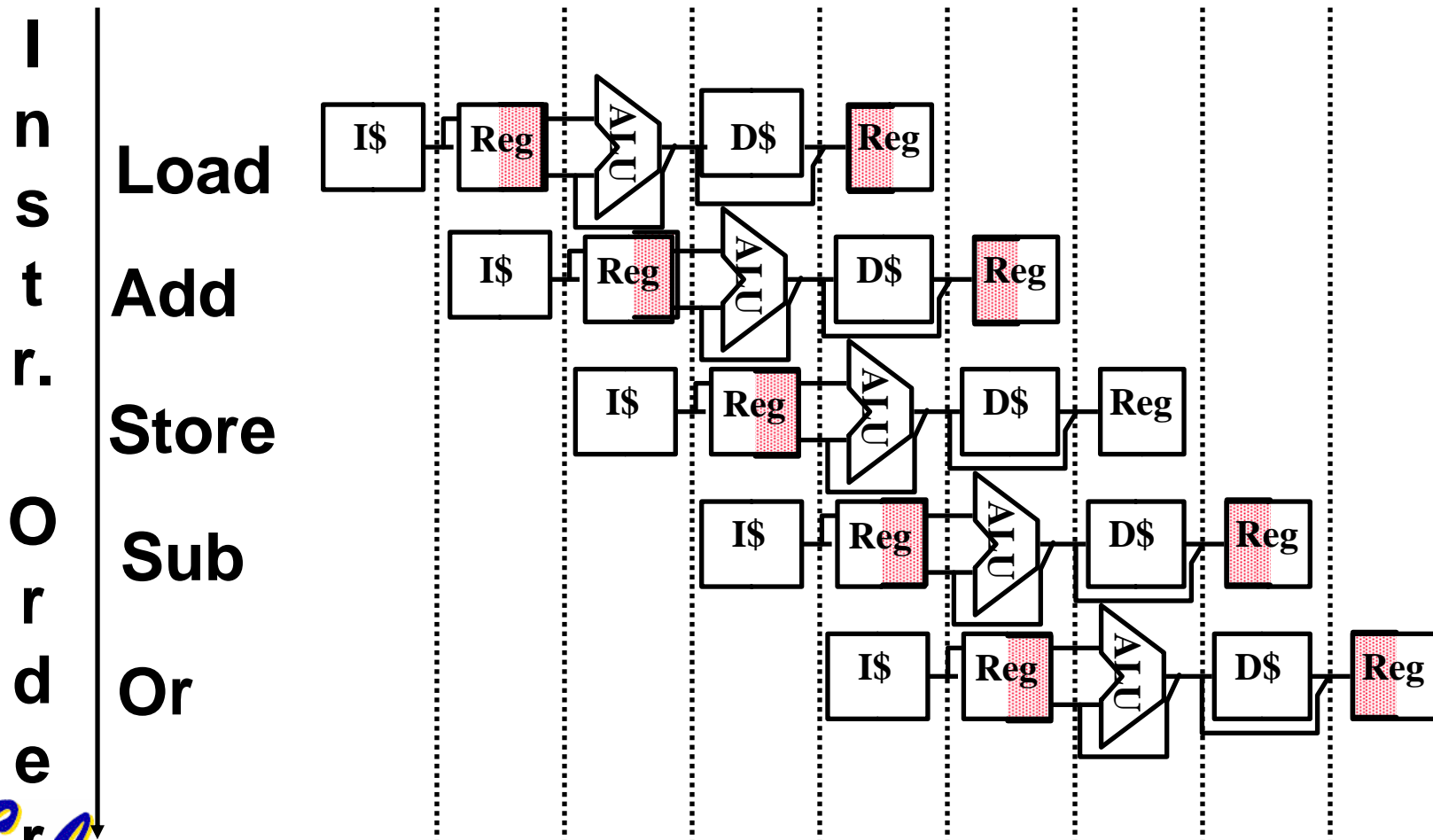
- Use datapath figure to represent pipeline



Graphical Pipeline Representation

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

Time (clock cycles)



Example

- Suppose 2 ns for memory access, 2 ns for ALU operation, and 1 ns for register file read or write; compute instruction **throughput**
- **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
- **Pipelined Execution:**
 - **Max**(IF,Read Reg,ALU,Memory,Write Reg) = 2 ns



Example

- Suppose 2 ns for memory access, 2 ns for ALU operation, and 1 ns for register file read or write; compute instruction latency
- 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
- Pipelined Execution:
 - **SUM**(IF,Read Reg,ALU,Memory,Write Reg) = 10 ns



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*, executes 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.**



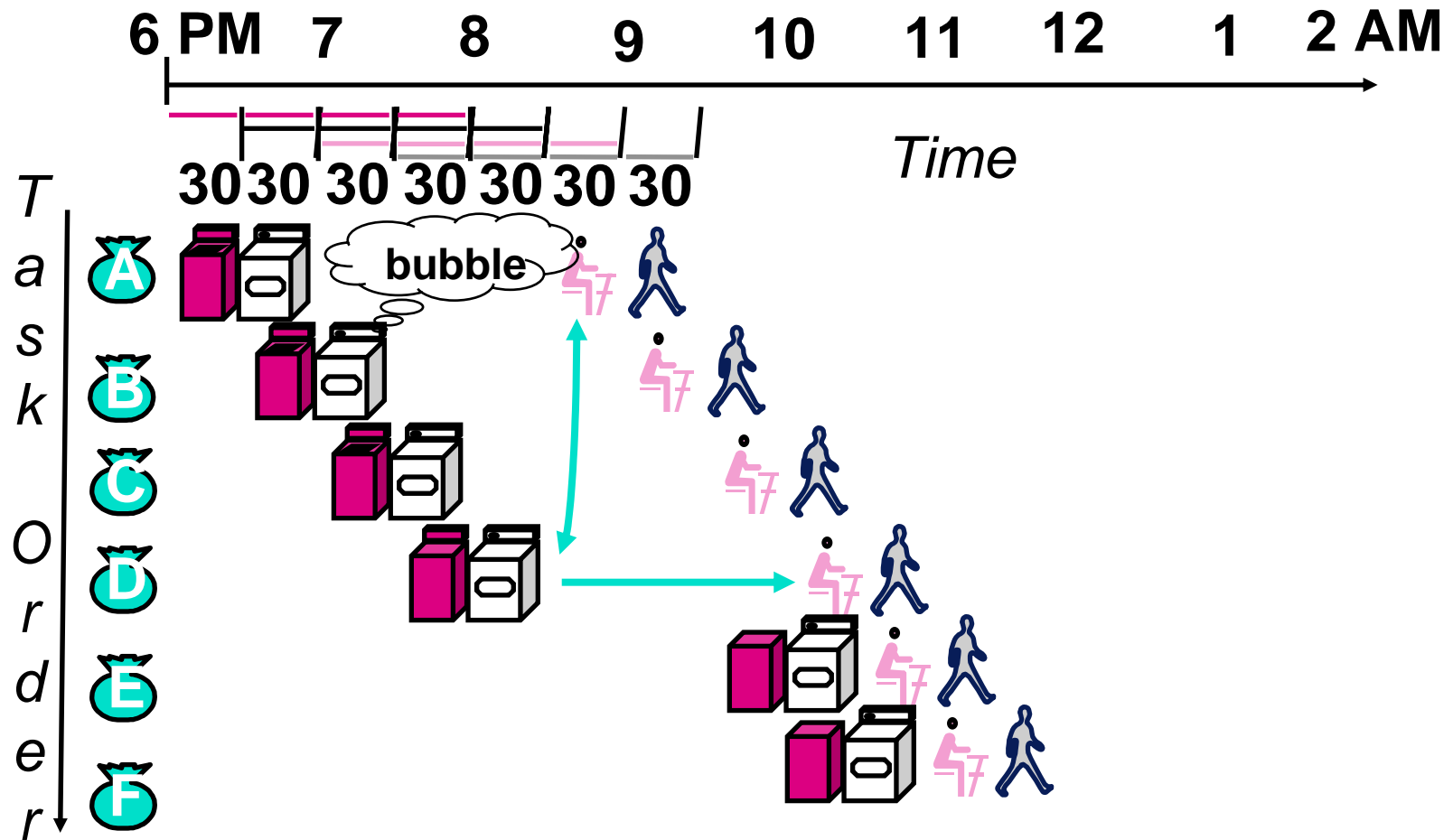
Pipeline Summary

- **Pipelining is a BIG IDEA**
 - **widely used concept**

- **What makes it less than perfect? ...**



Pipeline Hazard: Matching socks in later load



A depends on D; stall since folder tied up

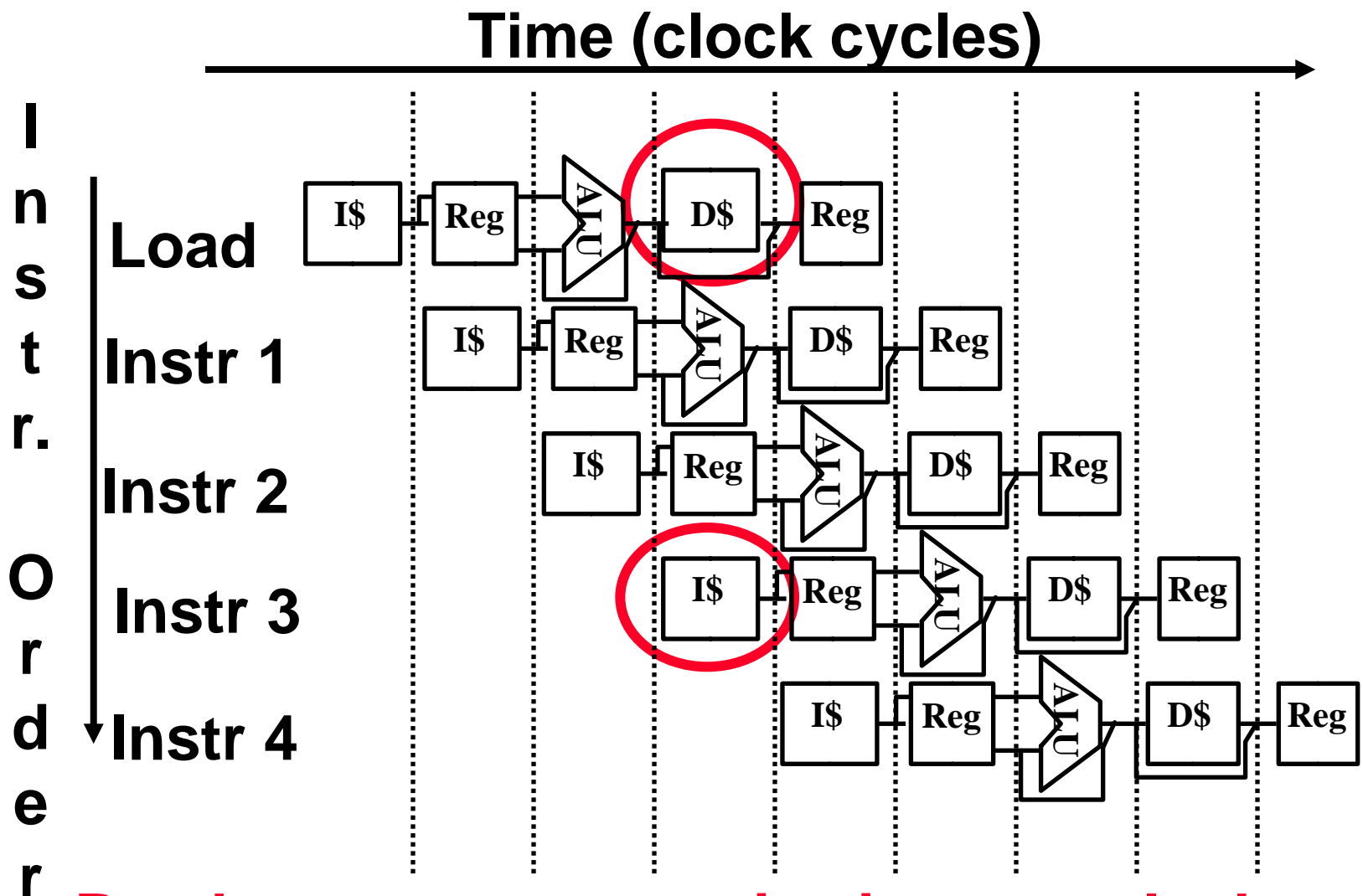


Problems for Computers

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
 - **Structural hazards**: HW cannot support this combination of instructions (single person to fold and put clothes away)
 - **Control hazards**: Pipelining of branches & other instructions **stall** the pipeline until the hazard; “**bubbles**” in the pipeline
 - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)



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



Read same memory twice in same clock cycle



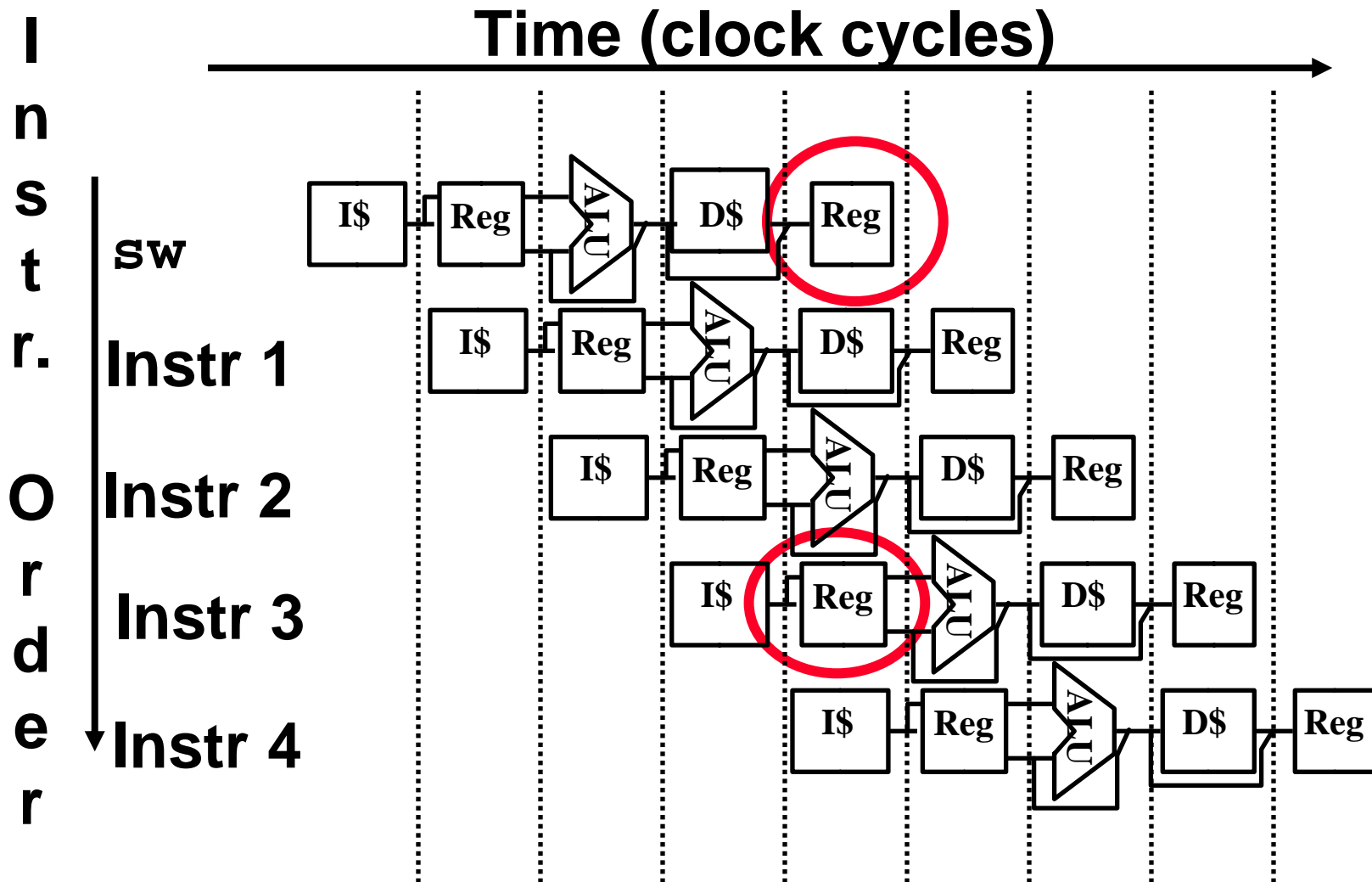
Structural Hazard #1: Single Memory (2/2)

- **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
- requires complex hardware to control when both caches miss!



Structural Hazard #2: Registers (1/2)



Can't read and write to registers simultaneously

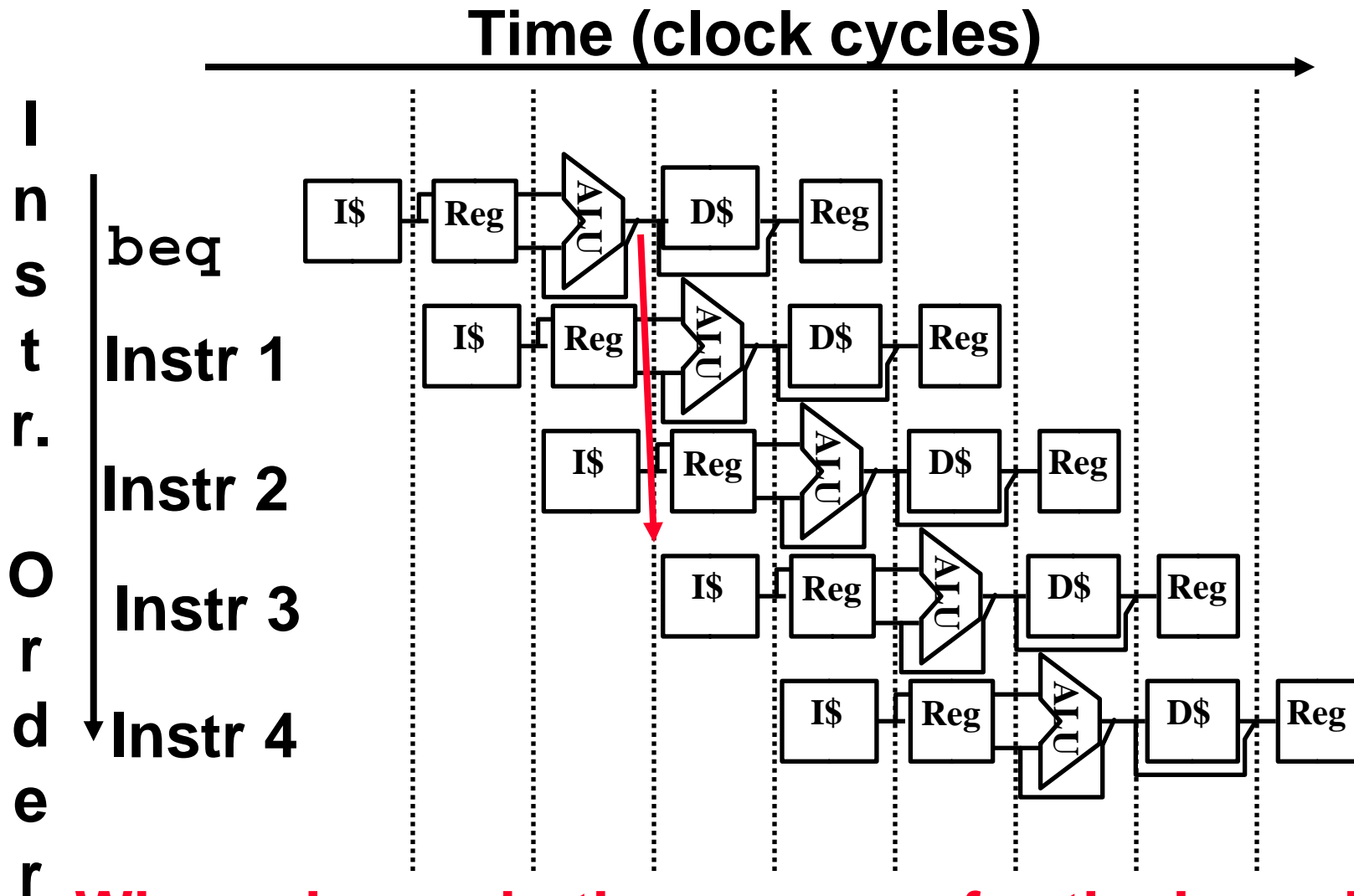


Structural Hazard #2: Registers (2/2)

- **Fact: Register access is *VERY* fast: takes less than half the time of ALU stage**
- **Solution: introduce convention**
 - **always Write to Registers during first half of each clock cycle**
 - **always Read from Registers during second half of each clock cycle (easy when async)**
 - **Result: can perform Read and Write during same clock cycle**



Control Hazard: Branching (1/7)



Where do we do the compare for the branch?



Control Hazard: Branching (2/7)

- **We put branch decision-making hardware in ALU stage**
 - therefore two more instructions after the branch will *always* be fetched, whether or not the branch is taken
- **Desired functionality of a branch**
 - if we do not take the branch, don't waste any time and continue executing normally
 - if we take the branch, don't execute any instructions after the branch, just go to the desired label



Control Hazard: Branching (3/7)

- **Initial Solution: Stall until decision is made**
 - insert “no-op” instructions: those that accomplish nothing, just take time
 - **Drawback: branches take 3 clock cycles each (assuming comparator is put in ALU stage)**
 - **Drawback: Will still fetch inst at branch+4. Must either decode branch in IF or squash fetched branch+4.**



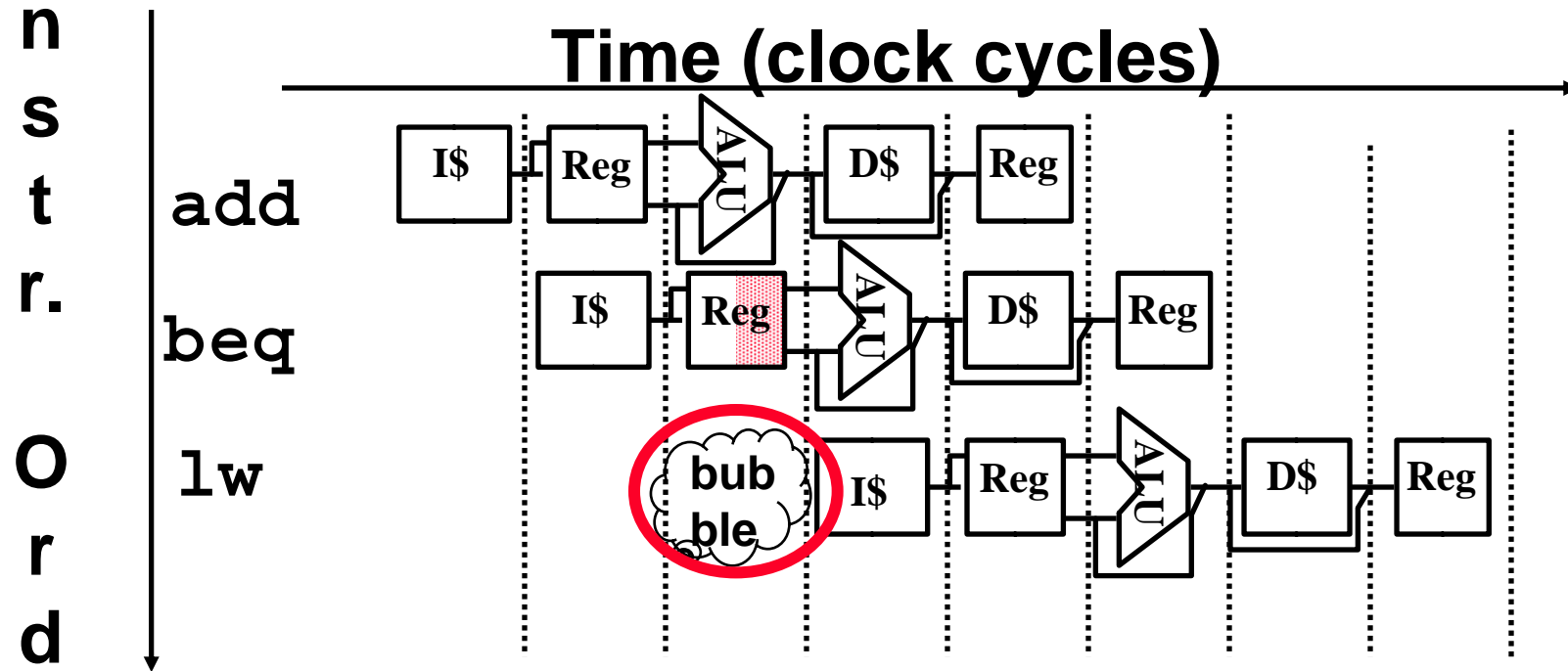
Control Hazard: Branching (4/7)

- **Optimization #1:**
 - move **asynchronous** comparator up to Stage 2
 - as soon as instruction is decoded (Opcode identifies is as a branch), immediately make a decision and set the value of the PC (if necessary)
 - **Benefit:** since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
 - **Side Note:** This means that branches are idle in Stages 3, 4 and 5.



Control Hazard: Branching (5/7)

- Insert a single no-op (bubble)



- Impact: 2 clock cycles per branch instruction \Rightarrow slow

Control Hazard: Branching (6/7)

- **Optimization #2: Redefine branches**
 - **Old definition: if we take the branch, none of the instructions after the branch get executed by accident**
 - **New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (called the **branch-delay slot**)**



Control Hazard: Branching (7/7)

- **Notes on Branch-Delay Slot**
 - **Worst-Case Scenario:** can always put a no-op in the branch-delay slot
 - **Better Case:** can find an instruction preceding the branch which can be placed in the branch-delay slot without affecting flow of the program
 - re-ordering instructions is a common method of speeding up programs
 - compiler must be very smart in order to find instructions to do this
 - usually can find such an instruction at least 50% of the time
 - Jumps also have a delay slot...



Example: Nondelayed vs. Delayed Branch

Nondelayed Branch

or \$8, \$9, \$10

add \$1, \$2, \$3

sub \$4, \$5, \$6

beq \$1, \$4, Exit

xor \$10, \$1, \$11

Exit:

Delayed Branch

add \$1, \$2, \$3

sub \$4, \$5, \$6

beq \$1, \$4, Exit

or \$8, \$9, \$10

xor \$10, \$1, \$11

Exit:



Data Hazards (1/2)

- Consider the following sequence of instructions

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

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

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

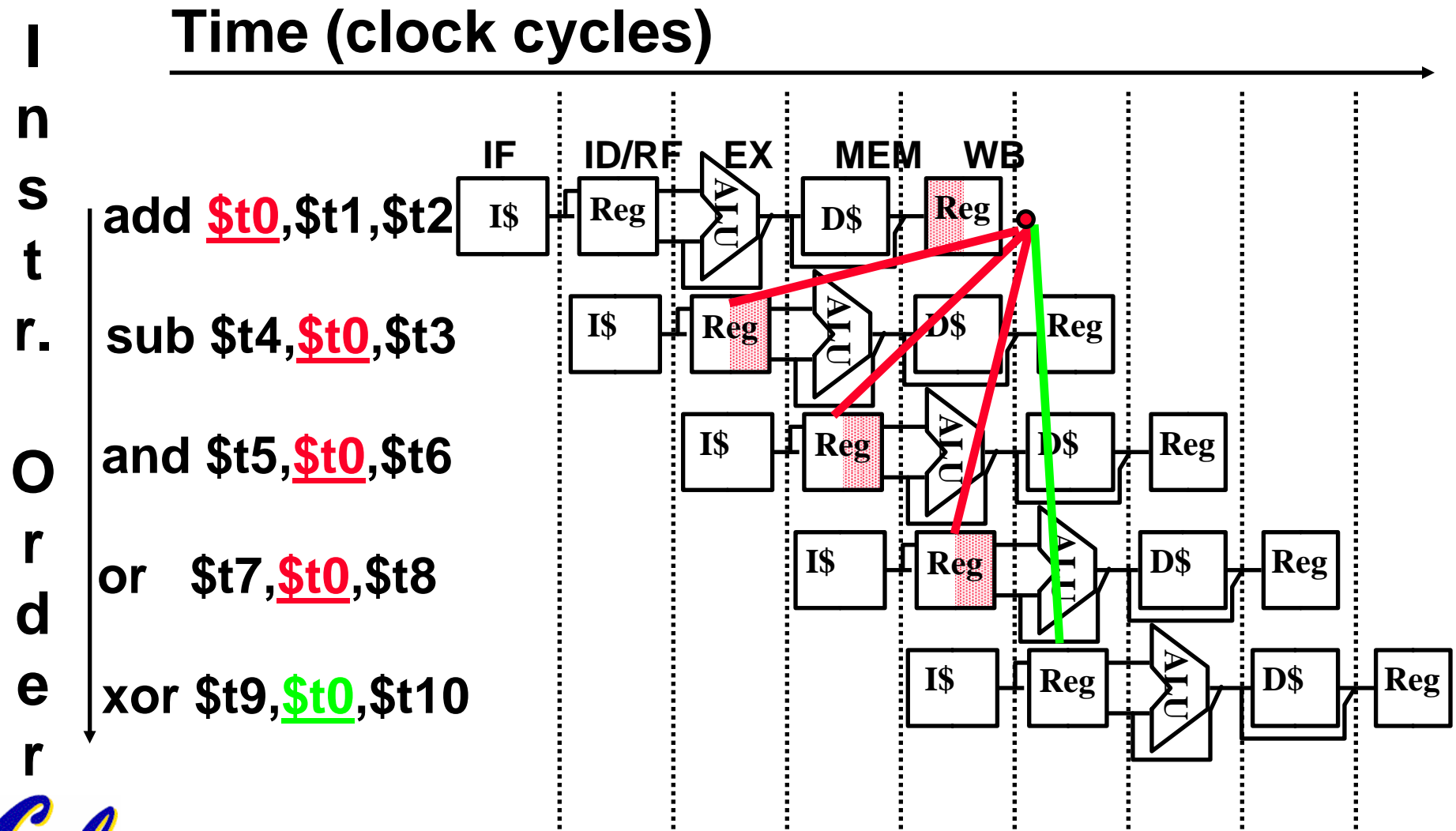
or \$t7, \$t0, \$t8

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



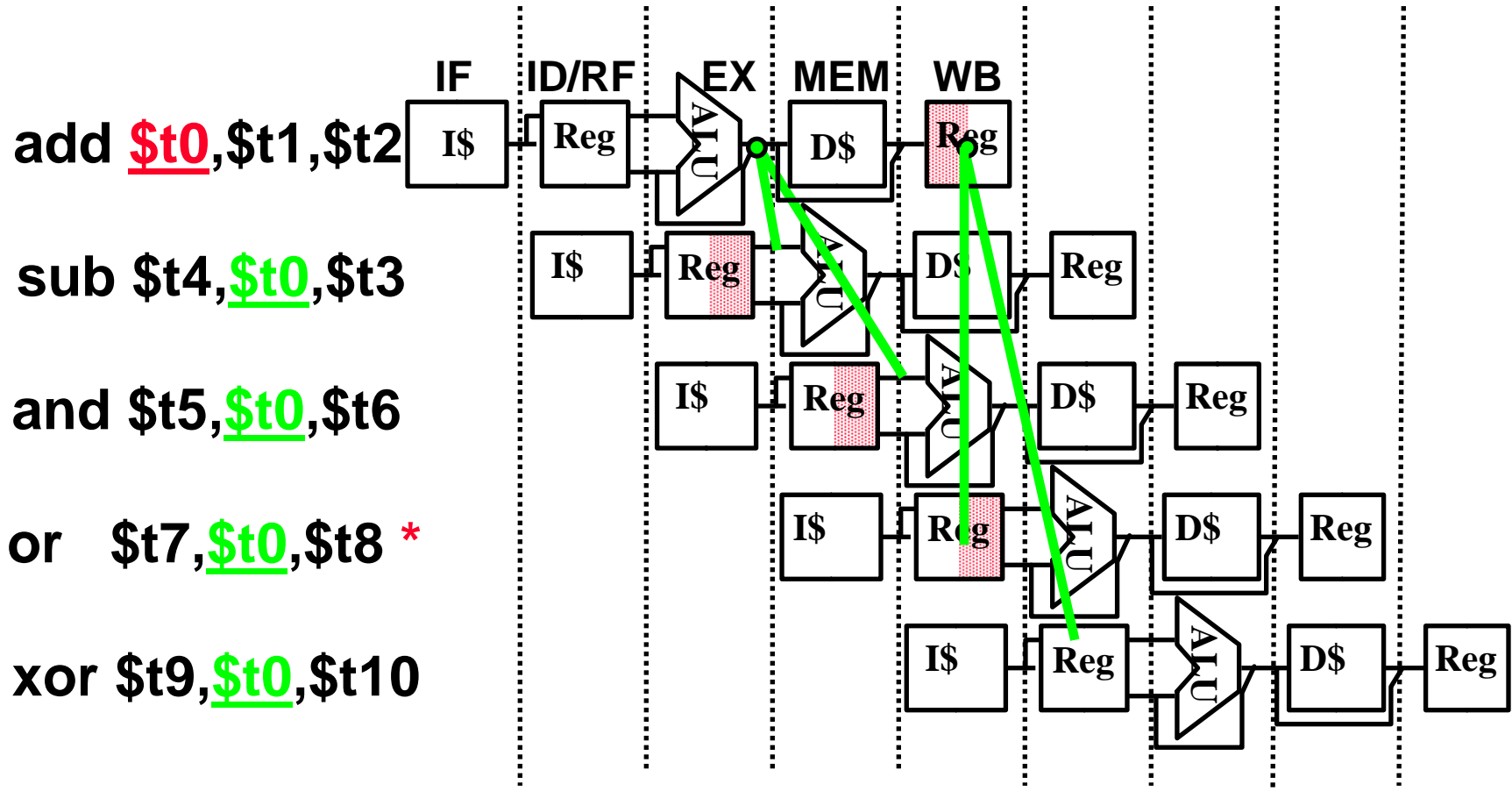
Data Hazards (2/2)

\$t0 not written back in time!



Data Hazard Solution: Forwarding

Fix by **Forwarding** result as soon as we have it to where we need it:

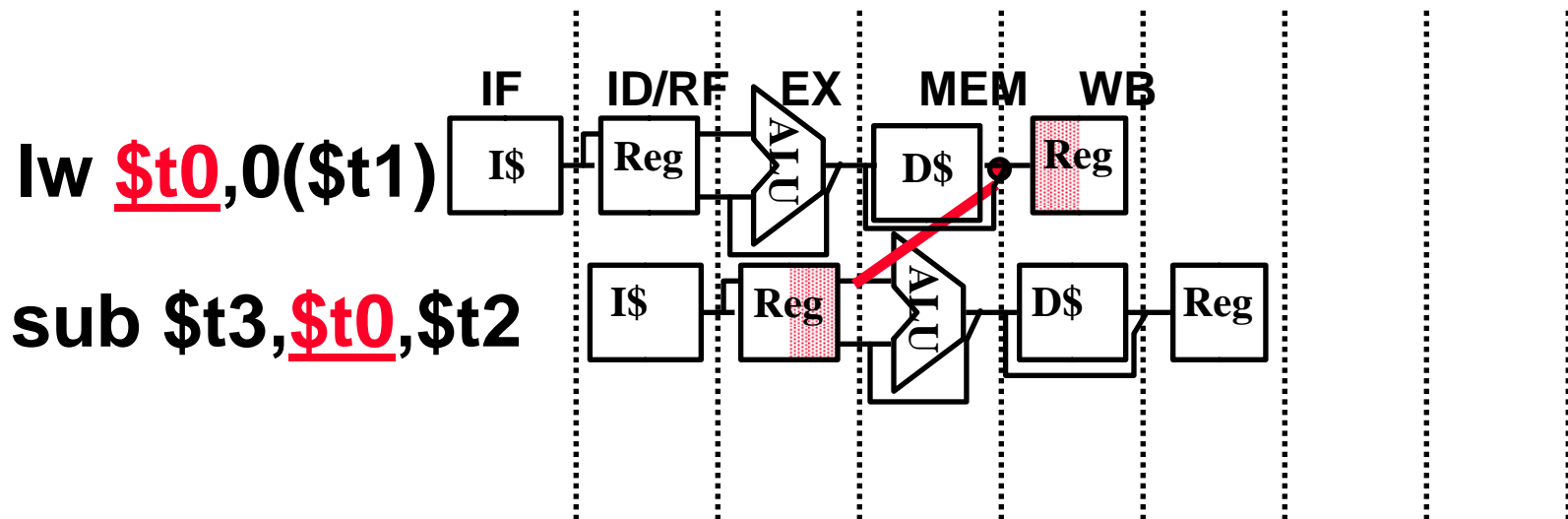


* “or” hazard solved by register hardware



Data Hazard: Loads (1/4)

- Forwarding works if value is available (but not written back) before it is needed. But consider ...



- Need result before it is calculated!
- Must stall use (sub) 1 cycle and *then* forward. ...



Data Hazard: Loads (2/4)

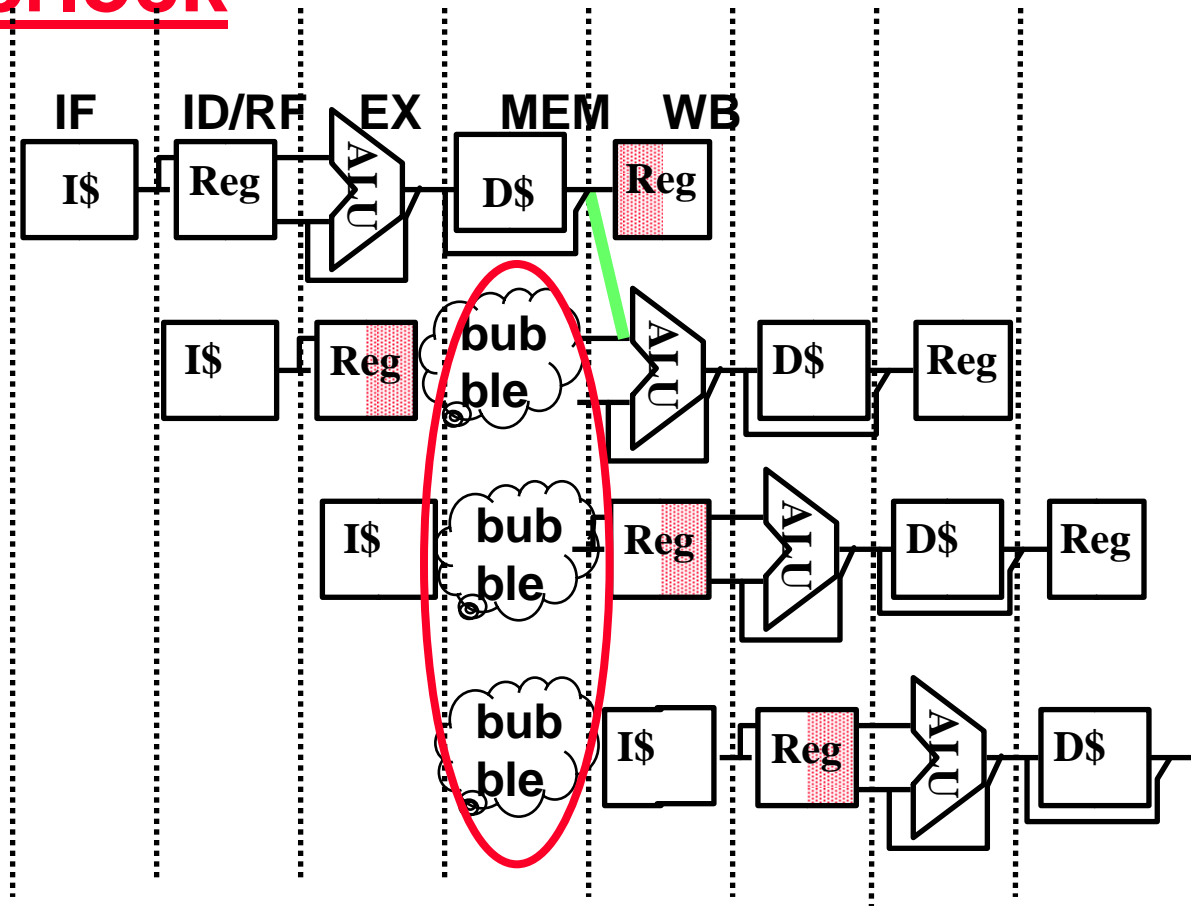
- Hardware must stall pipeline
- Called “interlock”

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)



C.f. Branch Delay vs. Load Delay

- **Load Delay occurs only if necessary (dependent instructions).**
- **Branch Delay always happens (part of the ISA).**

- **Why not have Branch Delay interlocked?**
 - **Answer: Interlocks only work if you can detect hazard ahead of time. By the time we detect a branch, we already need its value ... hence no interlock is possible!**



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
 - Load/Use → Wrong Answer!



“And in Conclusion..”

- Pipeline challenge is hazards
- Forwarding helps w/many data hazards
- Delayed branch helps with control hazard in 5 stage pipeline

- **Monday: Pipelined Datapath and Control in Detail!**
 - **Please finish phase 1 of proj 3 by Monday.**

