Great Idea #4: Parallelism

Software

- Parallel Requests
  - Assigned to computer
  - e.g. search "Garcia"
- Parallel Threads
  - Assigned to core
  - e.g. lookup, ads
- Parallel Instructions
  - > 1 instruction @ one time
  - e.g. 5 pipelined instructions
- Parallel Data
  - > 1 data item @ one time
  - e.g. add of 4 pairs of words
- Hardware descriptions
  - All gates functioning in parallel at same time

Leverage Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Smart Phone

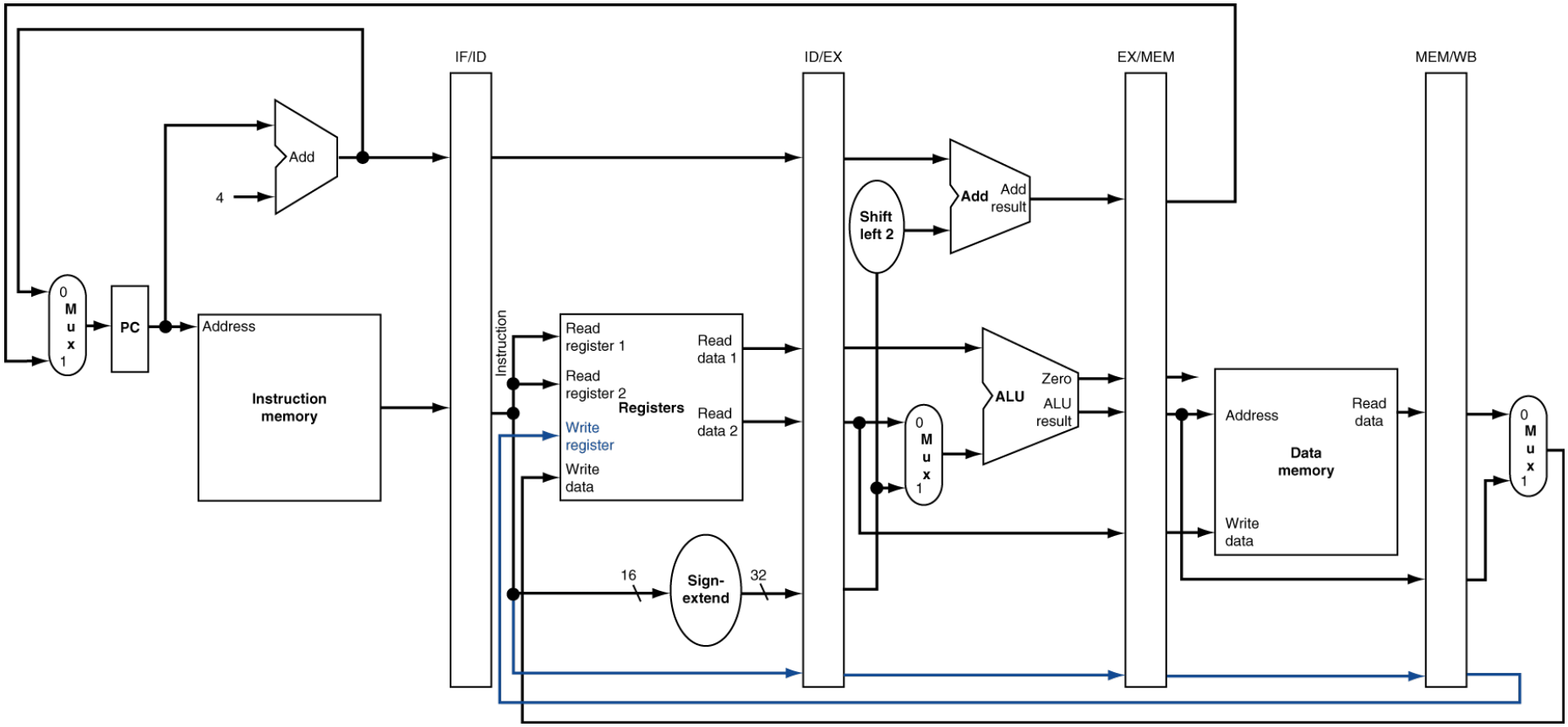
Review of Last Lecture

- Implementing controller for your datapath
  - Take decoded signals from instruction and generate control signals
  - Use “AND” and “OR” Logic scheme
- Pipelining improves performance by exploiting Instruction Level Parallelism
  - 5-stage pipeline for MIPS: IF, ID, EX, MEM, WB
  - Executes multiple instructions in parallel
  - What can go wrong???

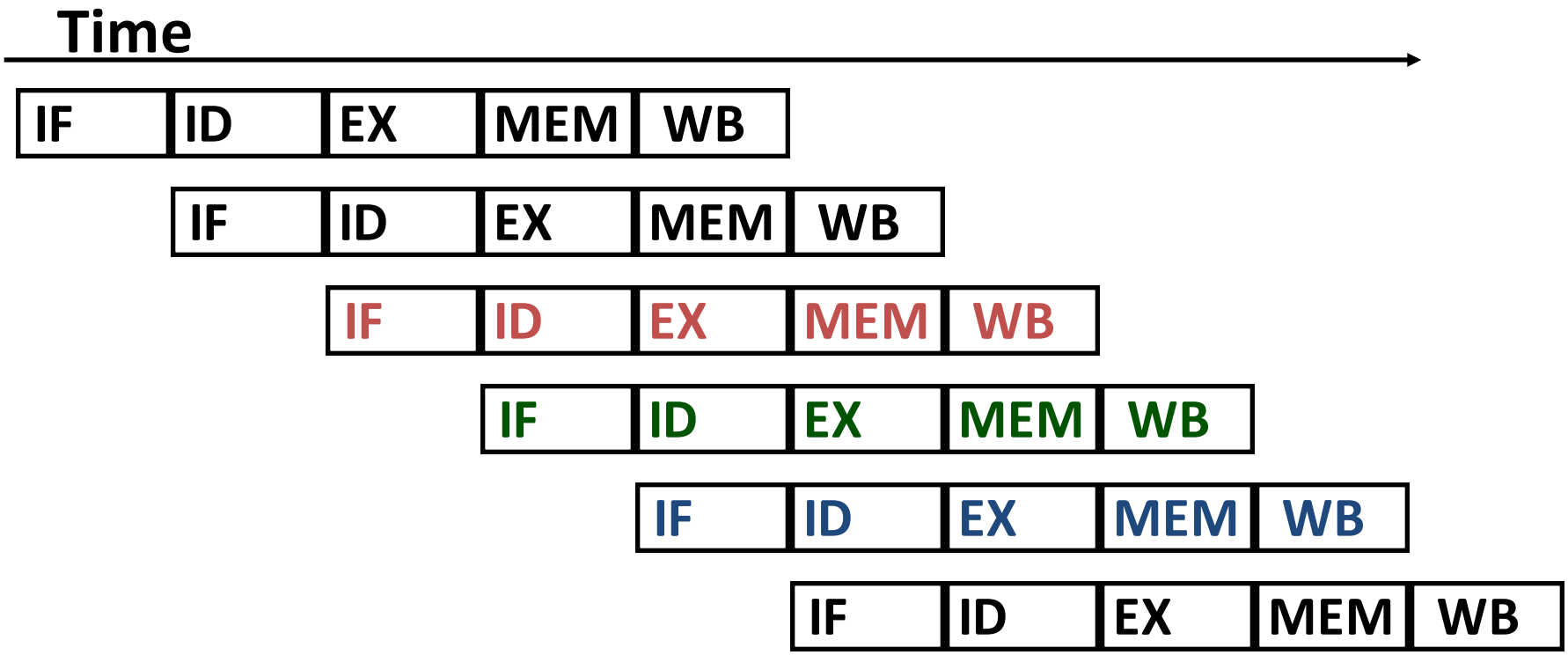
Agenda

- Pipelining Performance
- Structural Hazards
- Administrivia
- Data Hazards
  - Forwarding
  - Load Delay Slot
- Control Hazards

Review: Pipelined Datapath

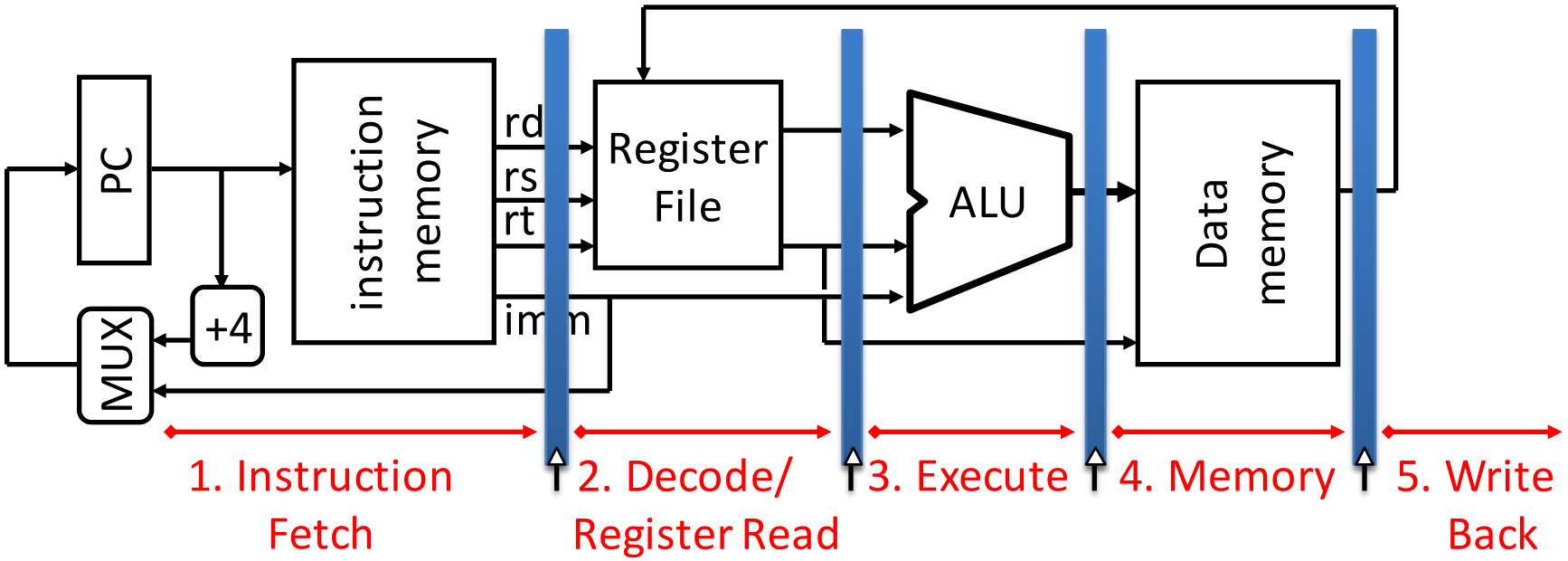


Pipelined Execution Representation

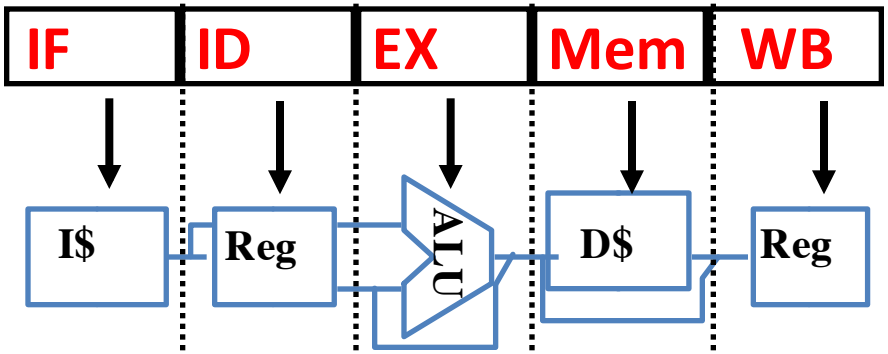


- Every instruction must take same number of steps, so some stages will idle
  - e.g. MEM stage for any arithmetic instruction

Graphical Pipeline Diagrams

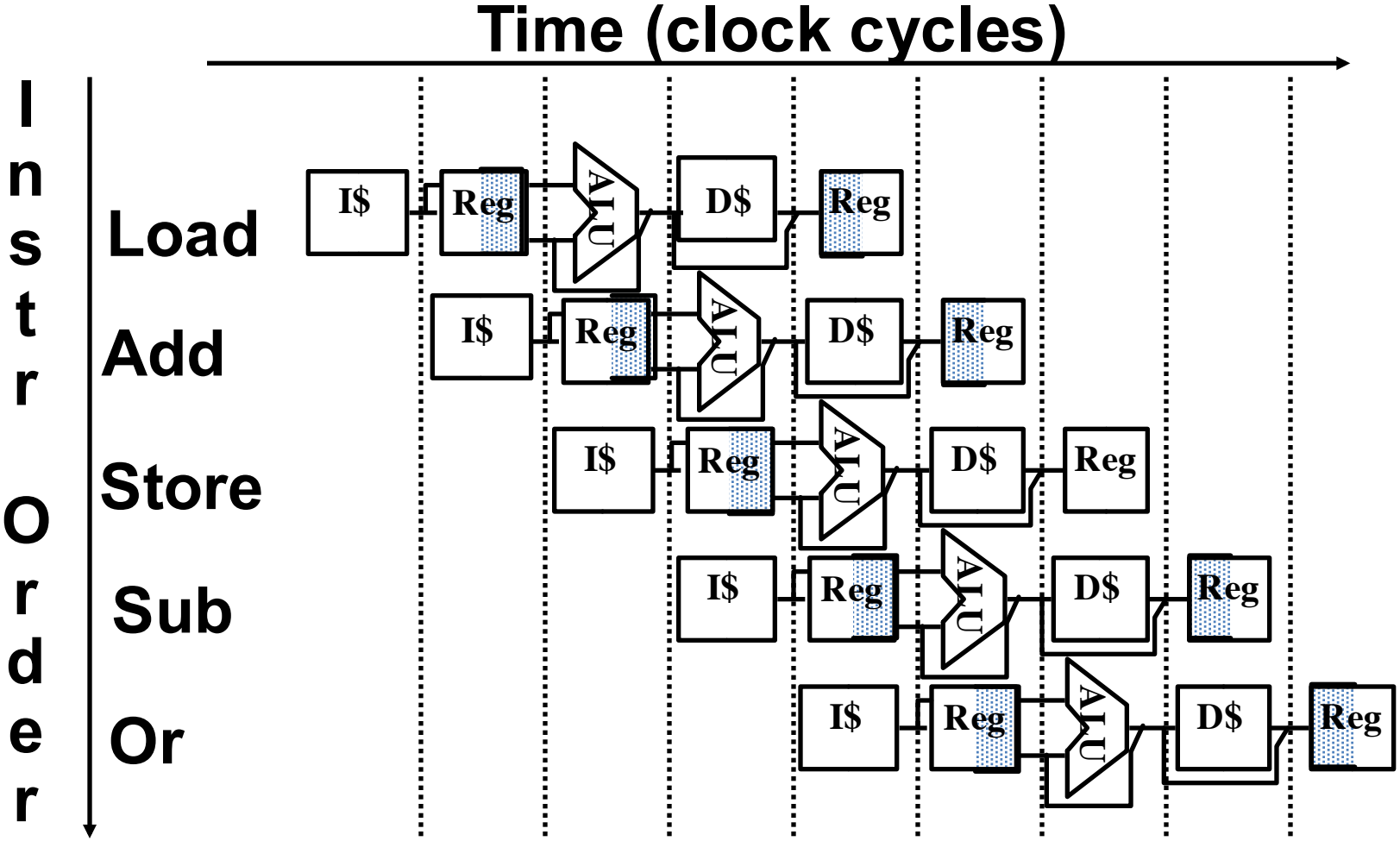


- Use datapath figure below to represent pipeline:



Graphical Pipeline Representation

- RegFile: left half is write, right half is read



Pipelining Performance (1/3)

- Use T_c ("time between completion of instructions") to measure speedup
  - $T_{c,pipelined} \geq \frac{T_{c,single-cycle}}{\text{Number of stages}}$
  - Equality only achieved if stages are *balanced* (i.e. take the same amount of time)
- If not balanced, speedup is reduced
- Speedup due to increased *throughput*
  - *Latency* for each instruction does not decrease

Pipelining Performance (2/3)

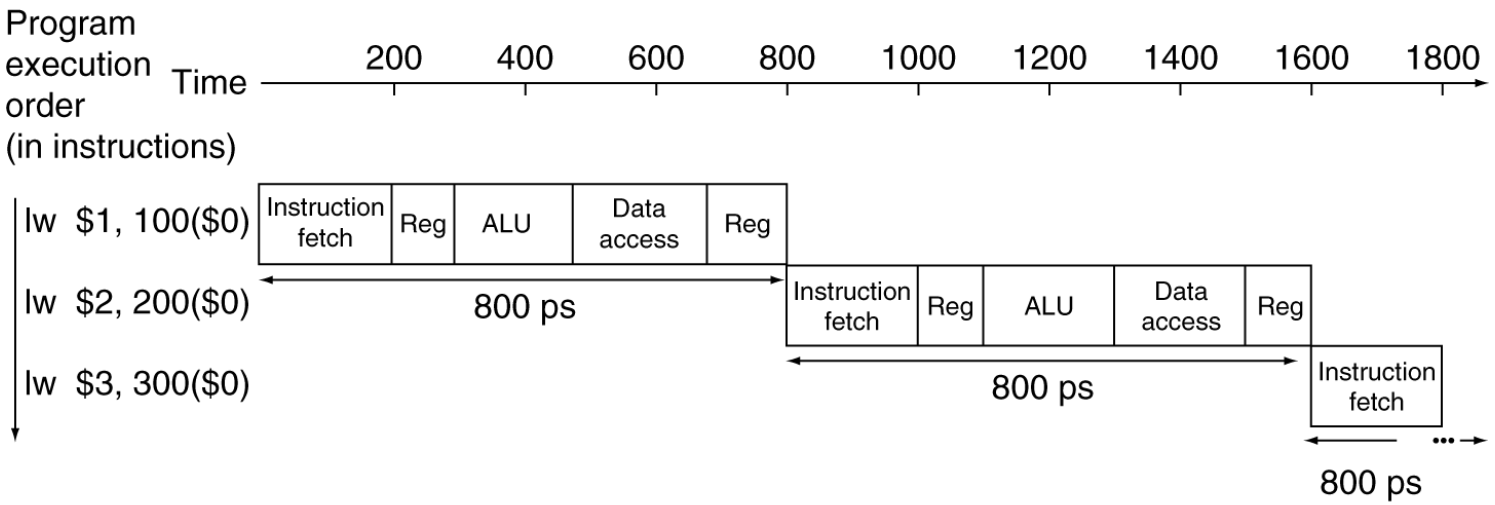
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

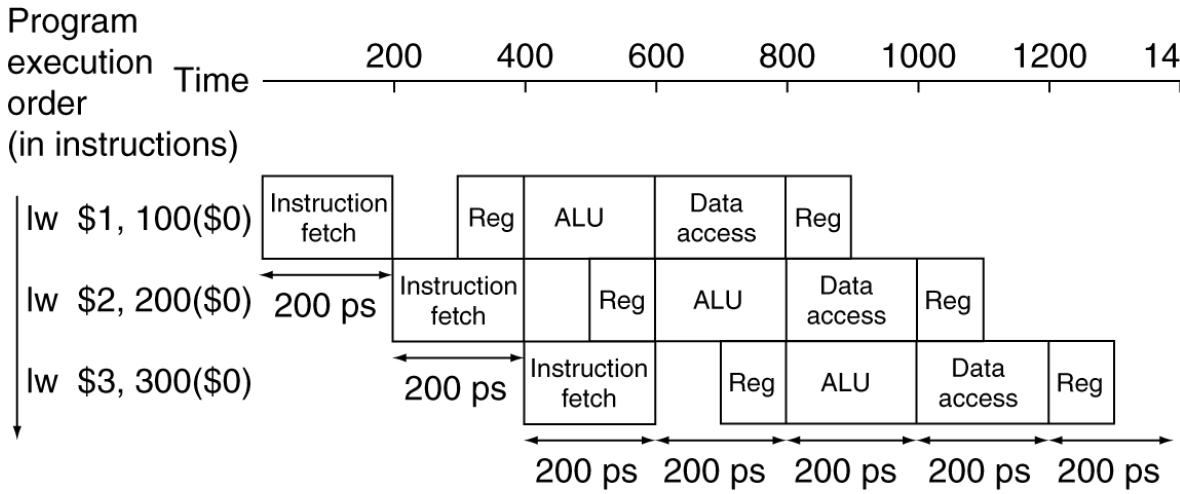
- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

Pipelining Performance (3/3)

Single-cycle
T_c = 800 ps



Pipelined
T_c = 200 ps



Pipelining Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) *Structural hazard*
   - A required resource is busy (e.g. needed in multiple stages)
2) *Data hazard*
   - Data dependency between instructions
   - Need to wait for previous instruction to complete its data read/write
3) *Control hazard*
   - Flow of execution depends on previous instruction

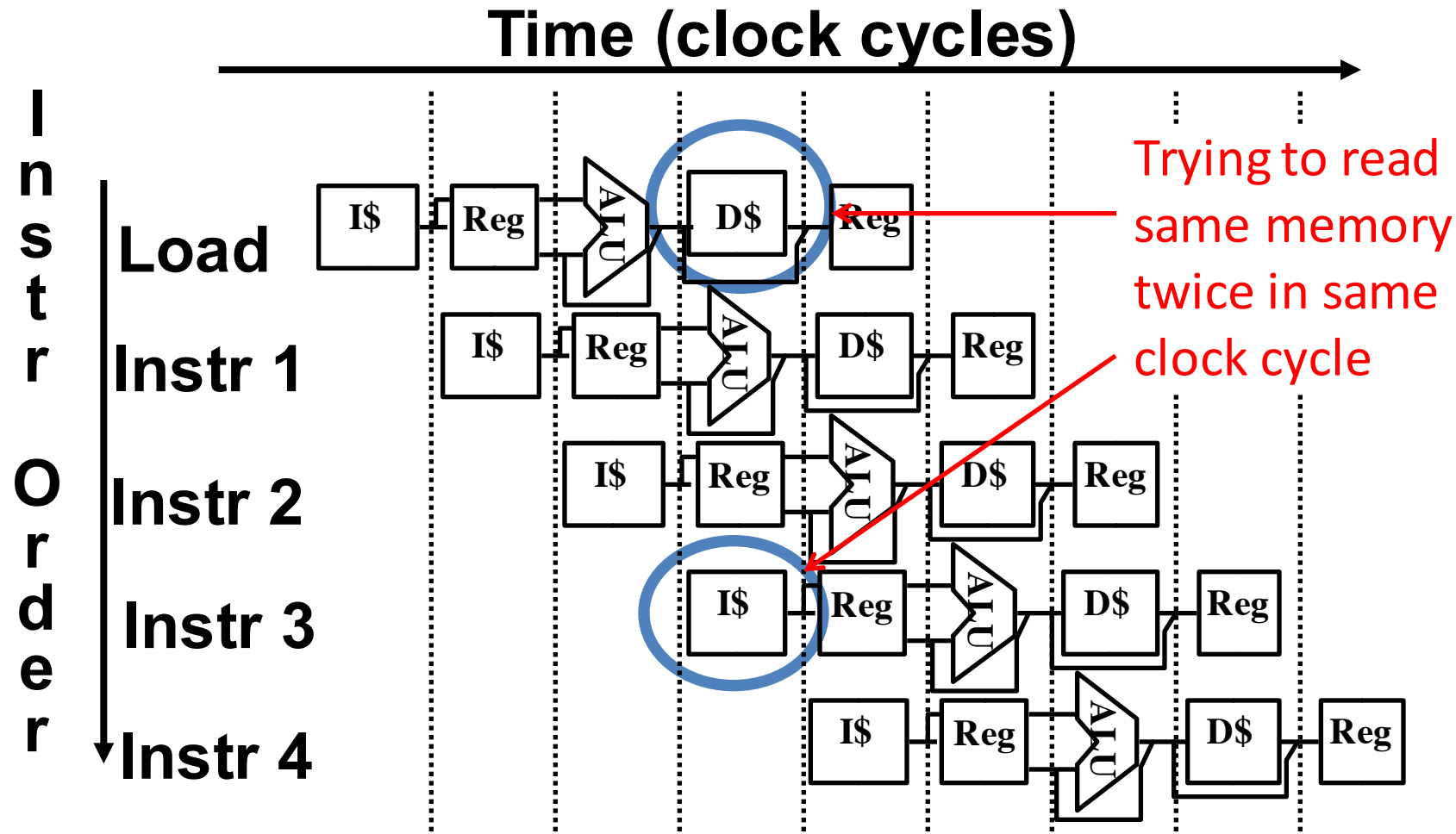
Agenda

- Pipelining Performance
- Structural Hazards
- Administrivia
- Data Hazards
  - Forwarding
  - Load Delay Slot
- Control Hazards

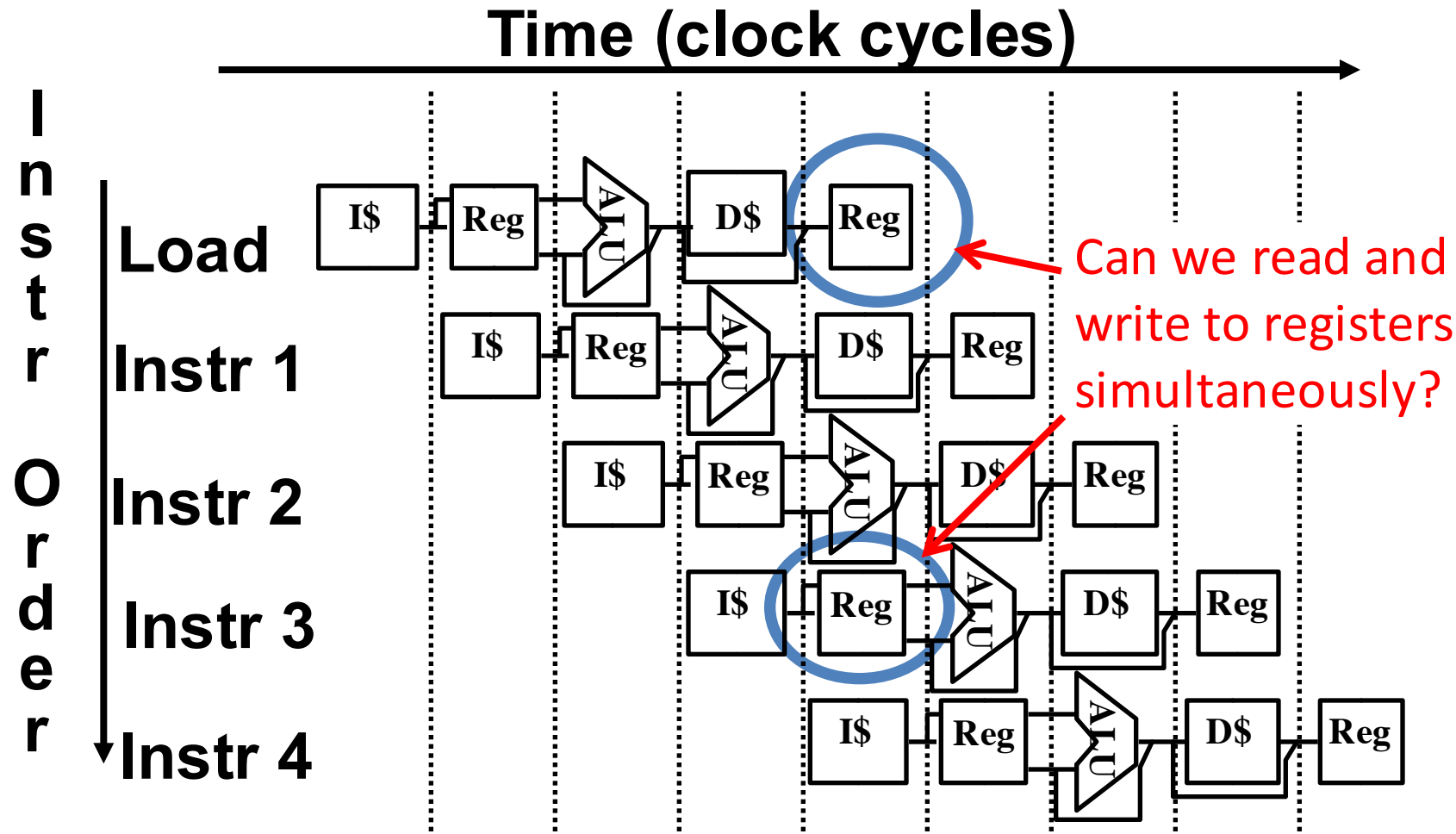
1. Structural Hazards

- Conflict for use of a resource
- MIPS pipeline with a single memory?
  - Load/Store requires memory access for data
  - Instruction fetch would have to *stall* for that cycle
    - Causes a pipeline "*bubble*"
- Hence, pipelined datapaths require separate instruction/data memories
  - Separate L1 I$ and L1 D$ take care of this

Structural Hazard #1: Single Memory



Structural Hazard #2: Registers (1/2)



Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) Split RegFile access in two: Write during 1st half and Read during 2nd half of each clock cycle
     - Possible because RegFile access is *VERY* fast (takes less than half the time of ALU stage)
  2) Build RegFile with independent read and write ports
- **Conclusion:** Read and Write to registers during same clock cycle is okay

Agenda

- Pipelining Performance
- Structural Hazards
- Administrivia
- Data Hazards
  - Forwarding
  - Load Delay Slot
- Control Hazards

Administrivia

- Check-in

Agenda

- Pipelining Performance
- Structural Hazards
- Administrivia
- Data Hazards
  - Forwarding
  - Load Delay Slot
- Control Hazards

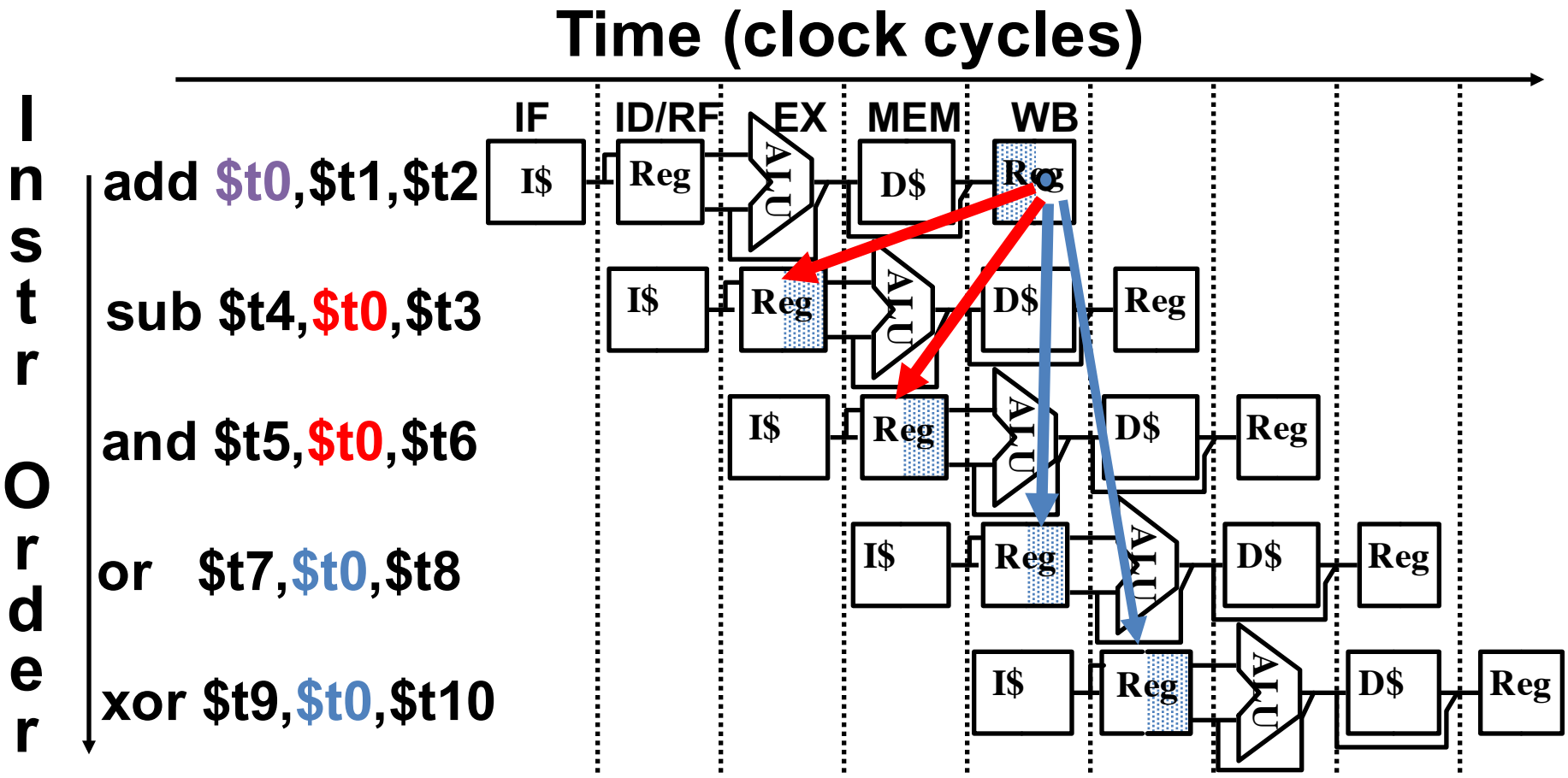
2. Data Hazards (1/2)

- 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
```

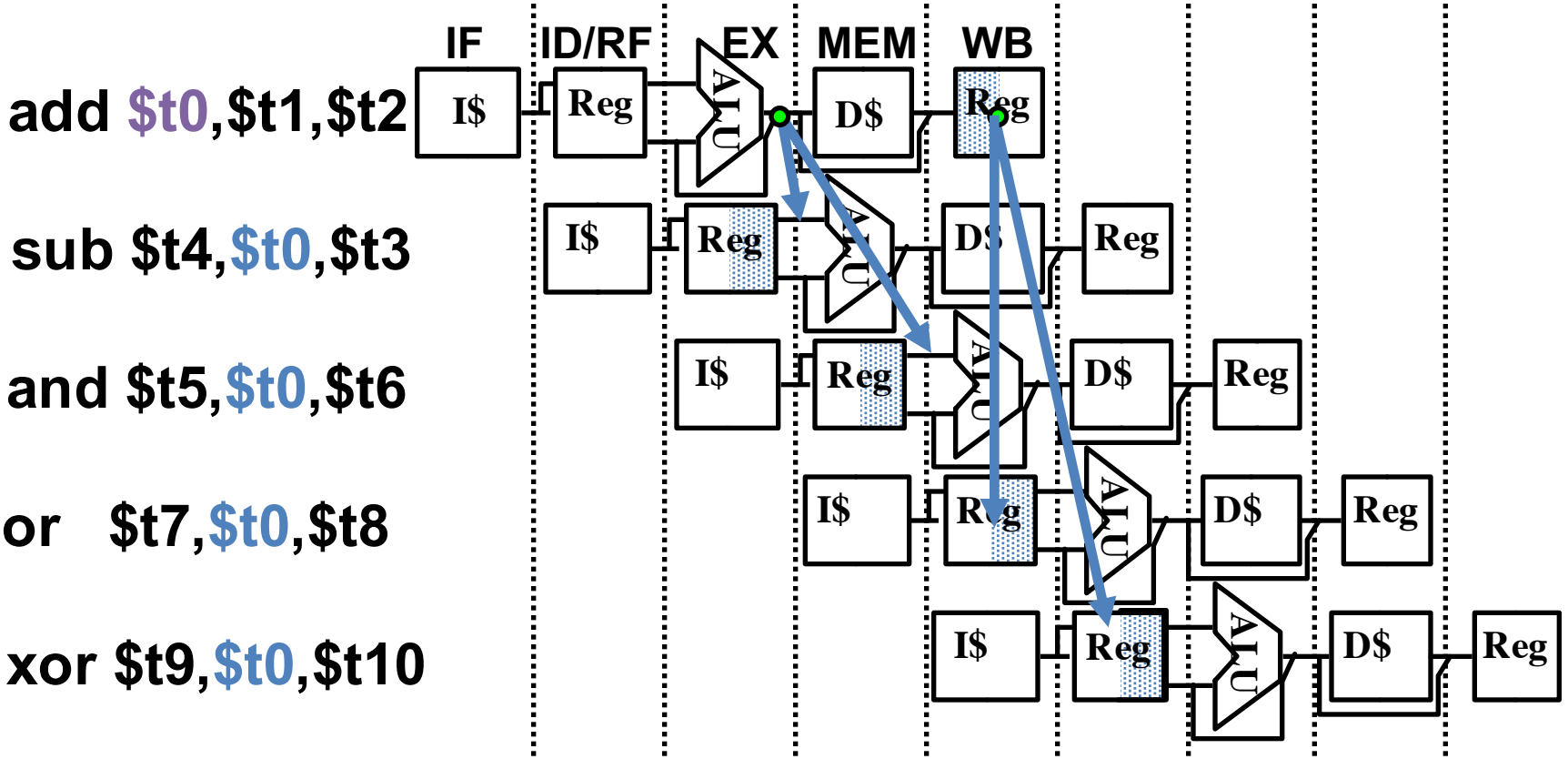
2. Data Hazards (2/2)

- Data-flow *backwards* in time are hazards



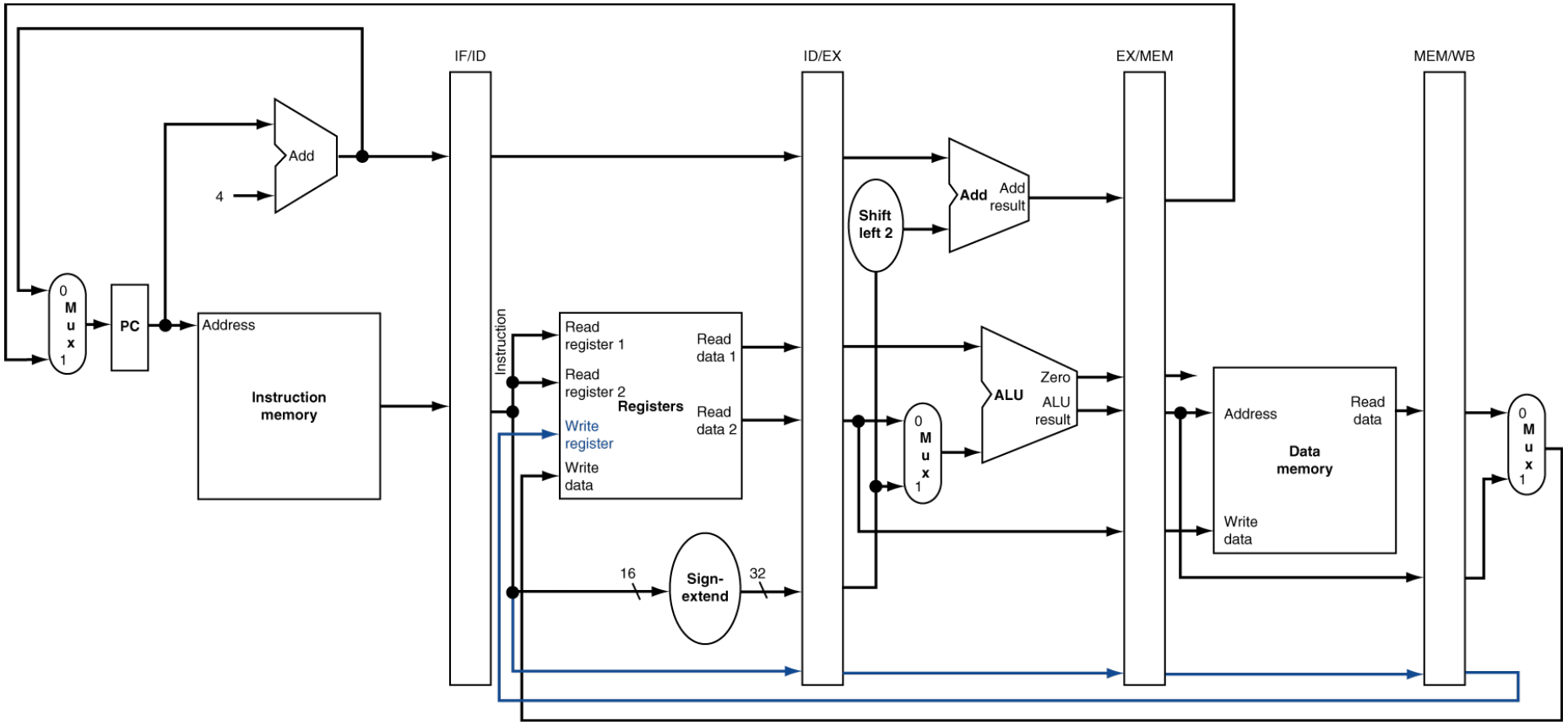
Data Hazard Solution: Forwarding

- Forward result as soon as it is available
  - OK that it's not stored in RegFile yet



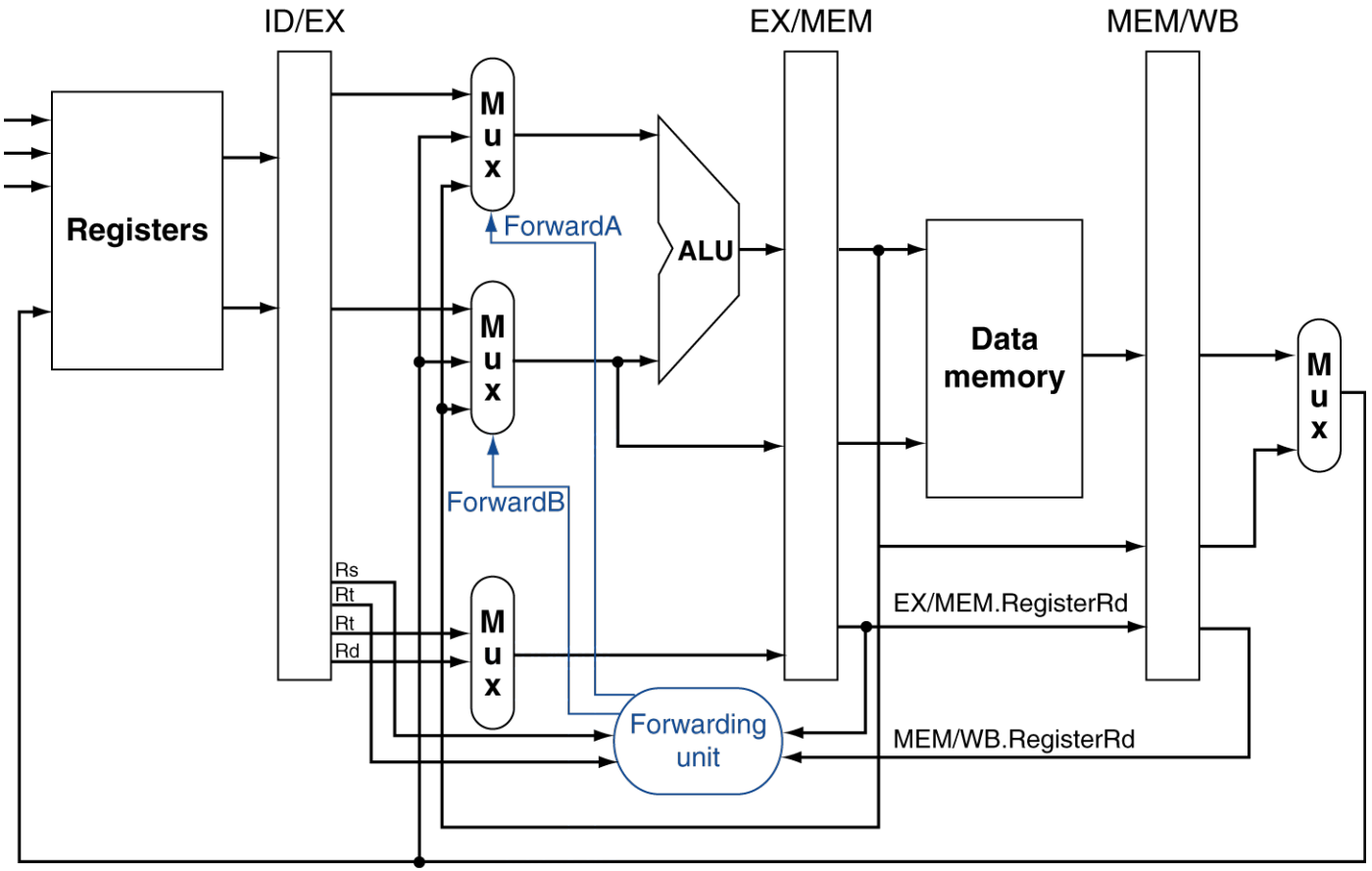
Datapath for Forwarding (1/2)

- What changes need to be made here?



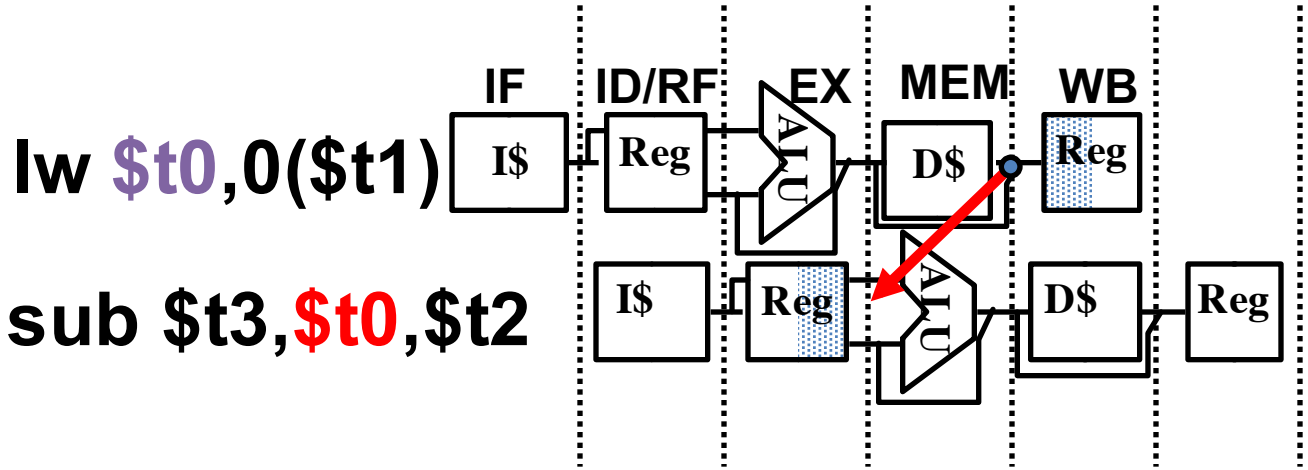
Datapath for Forwarding (2/2)

- Handled by *forwarding unit*



Data Hazard: Loads (1/4)

- **Recall:** Dataflow backwards in time are hazards

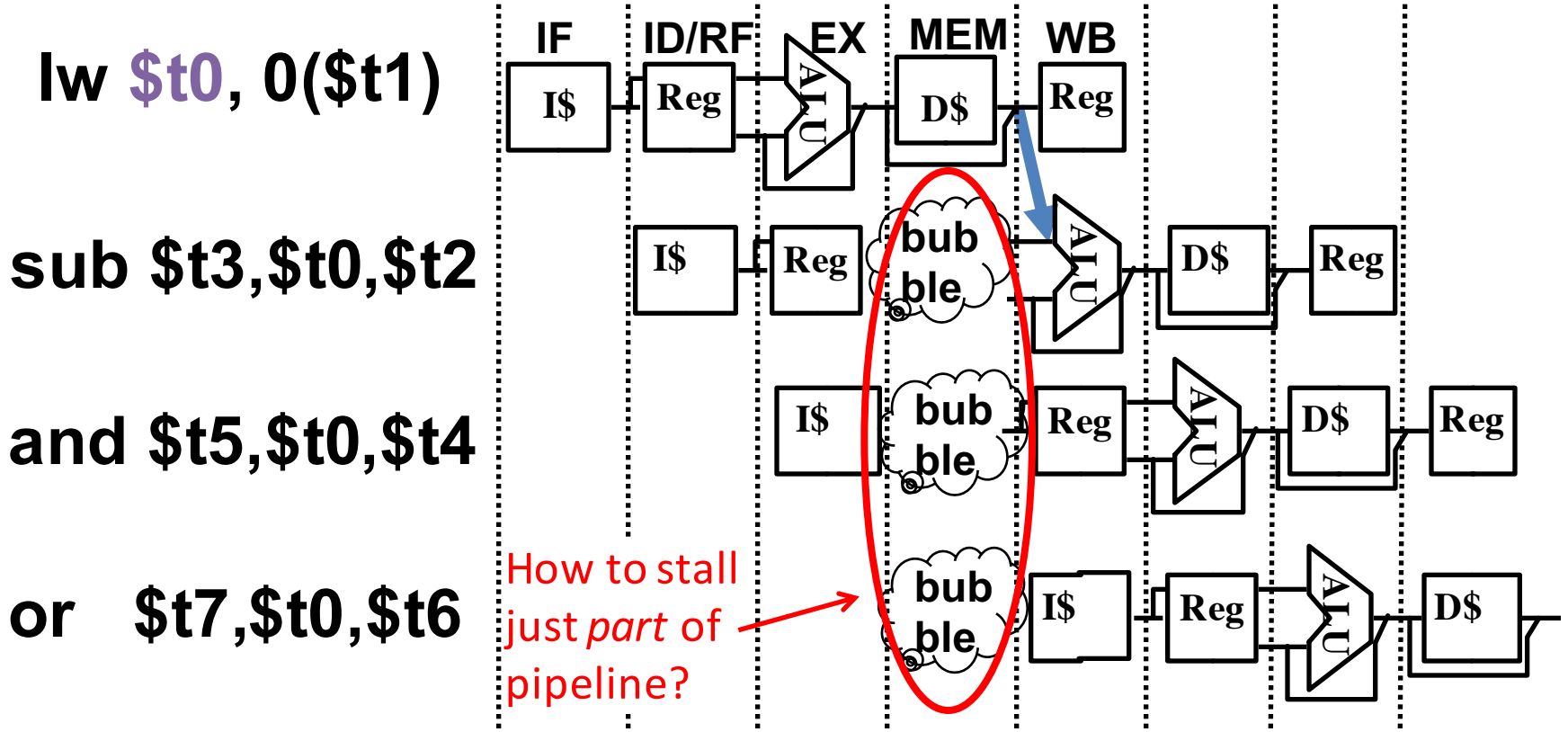


- Can't solve all cases with forwarding
  - Must *stall* instruction dependent on load, then forward (more hardware)

Data Hazard: Loads (2/4)

- *Hardware* stalls pipeline
  - Called "hardware interlock"

Schematically, this is what we want, but in reality stalls done "horizontally"



Data Hazard: Loads (3/4)

- Stall is equivalent to `nop`

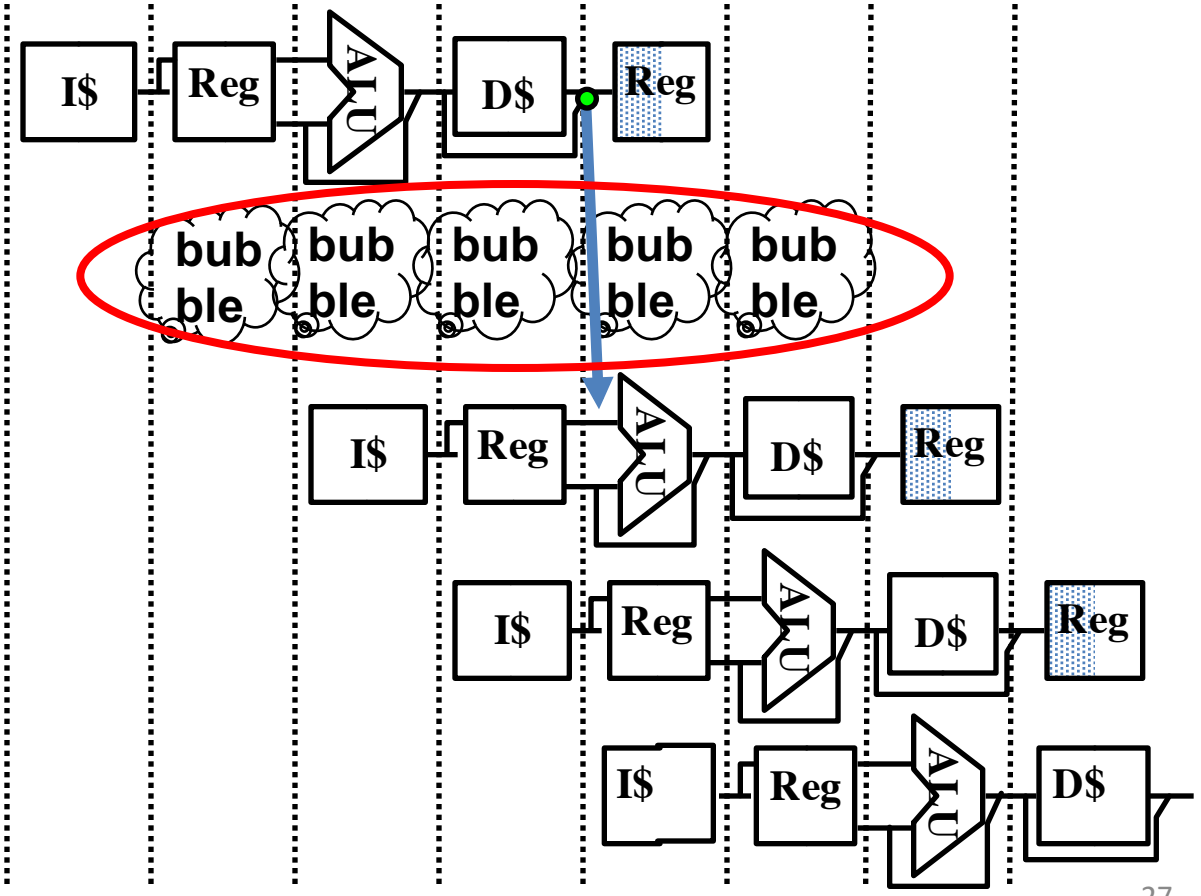
lw $t0, 0($t1)

nop

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6

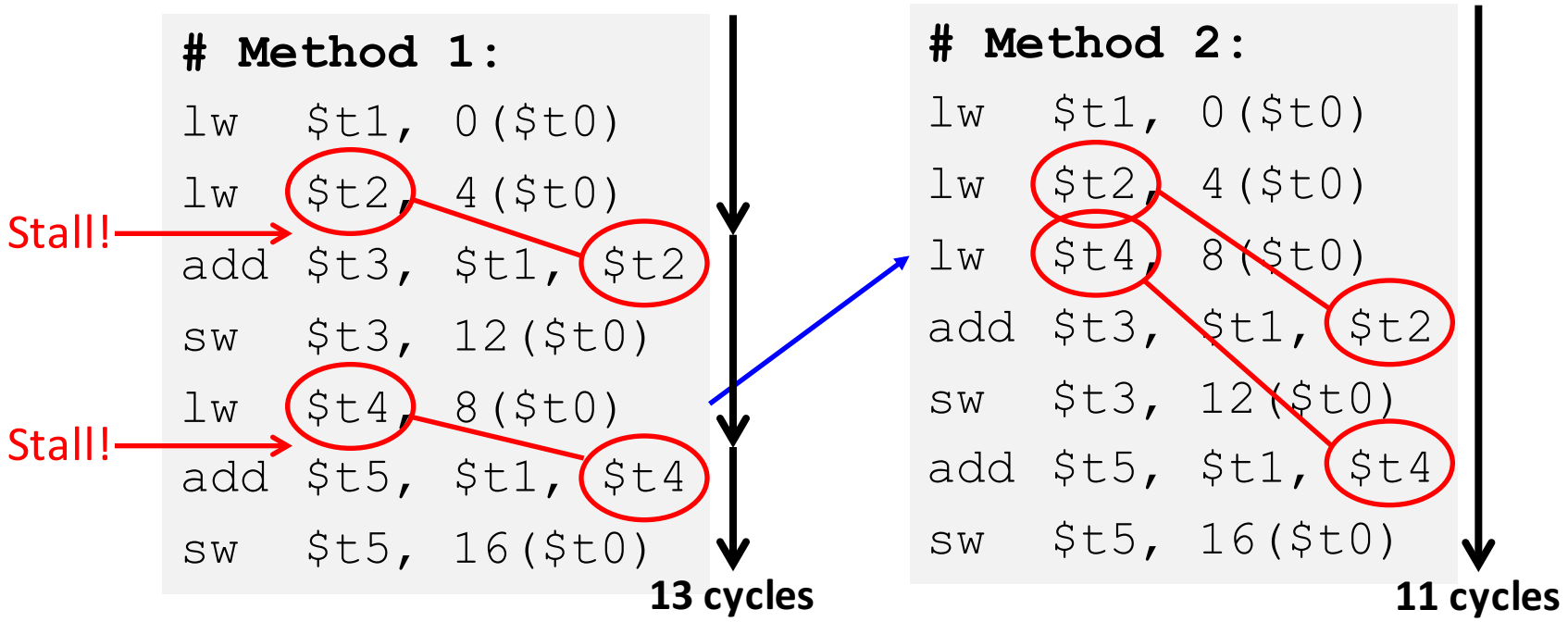


Data Hazard: Loads (4/4)

- Slot after a load is called a *load delay slot*
  - If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle
  - 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)
- **Idea:** Let the compiler put an unrelated instruction in that slot → no stall!

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction!
- MIPS code for `D=A+B; E=A+C;`



Agenda

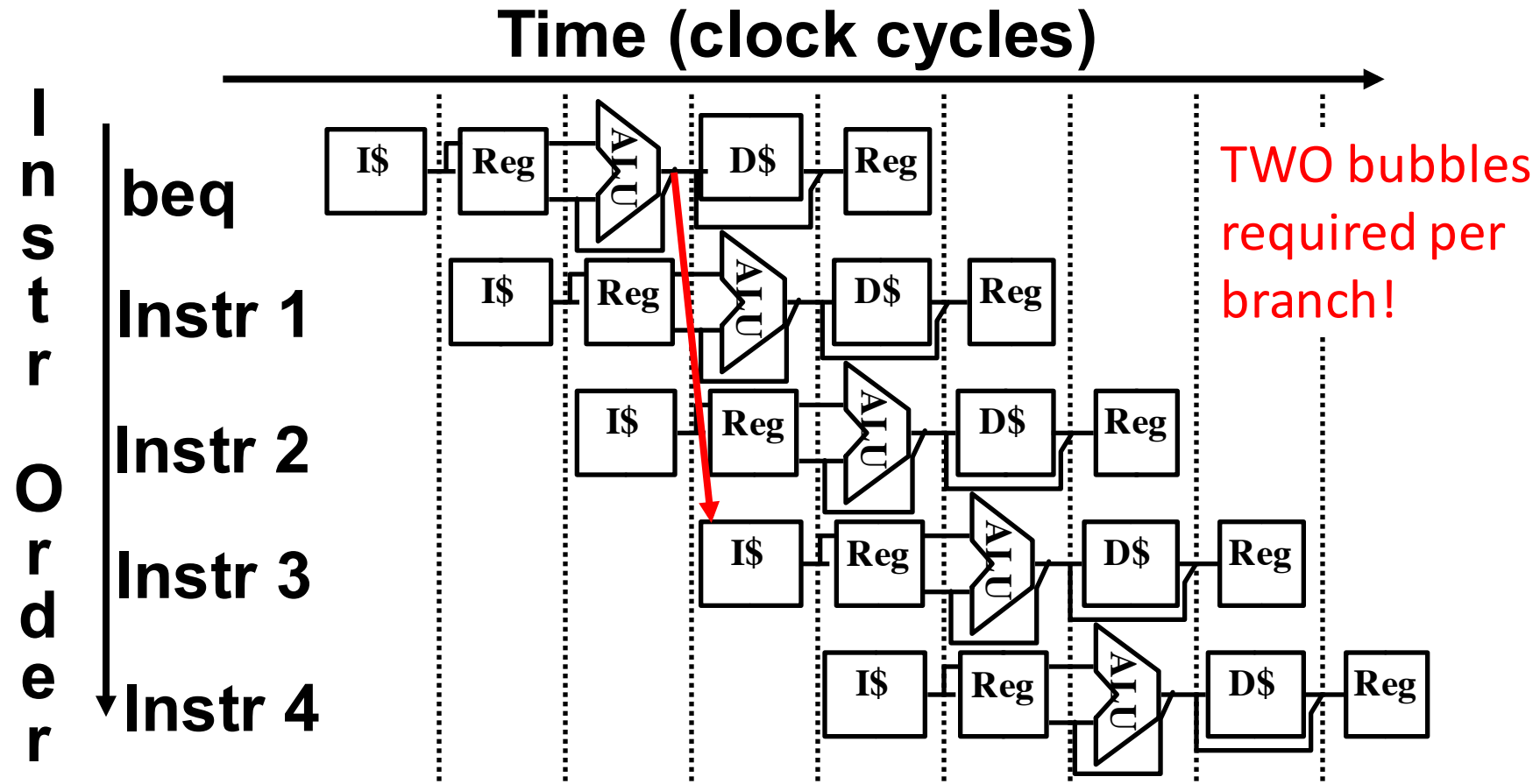
- More Pipelining
- Structural Hazards
- Administrivia
- Data Hazards
  - Forwarding
  - Load Delay Slot
- Control Hazards

3. Control Hazards

- Branch (`beq`, `bne`) determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- **Simple Solution:** Stall on *every* branch until we have the new PC value
  - How long must we stall?

Branch Stall

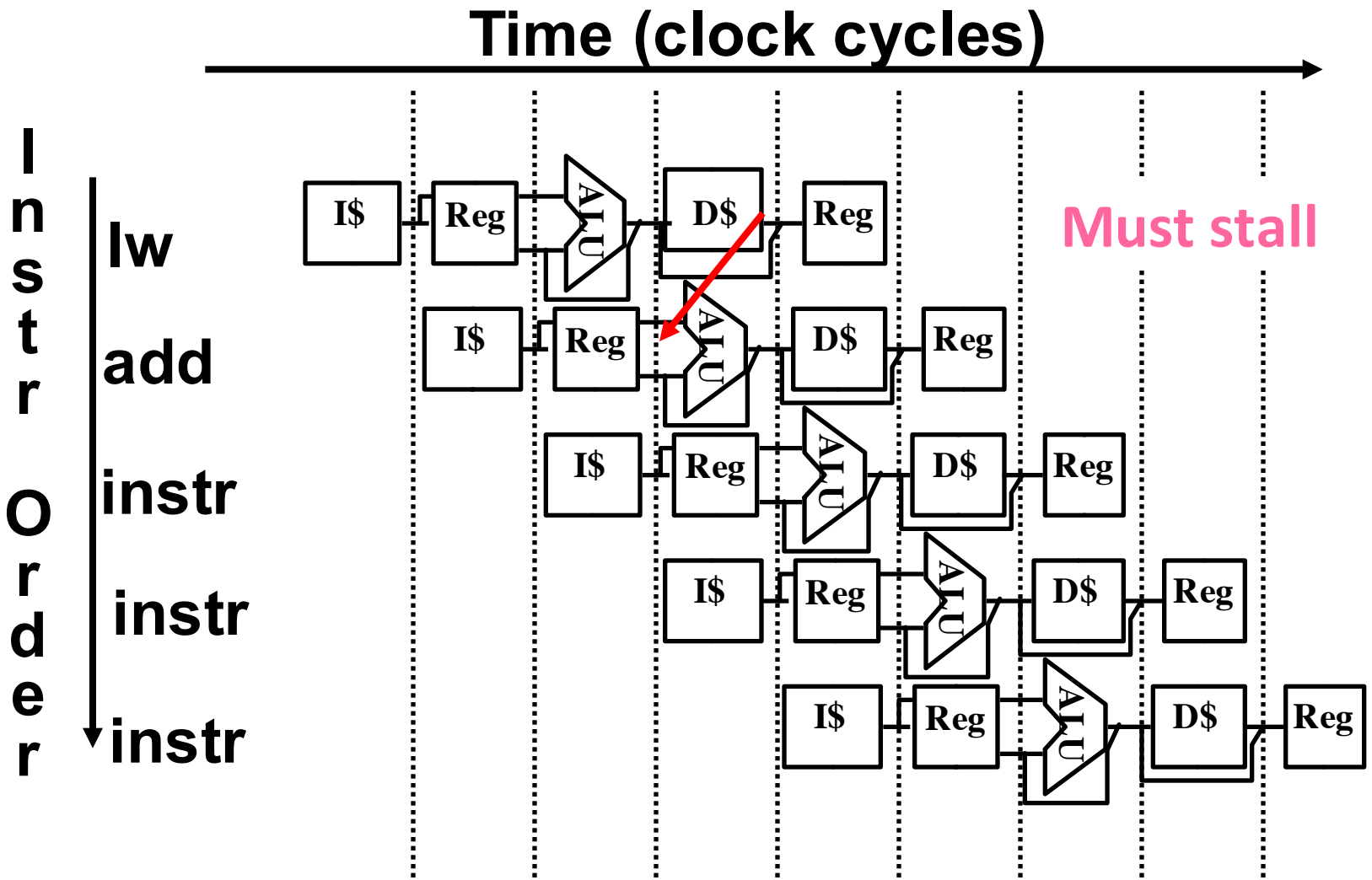
- When is comparison result available?



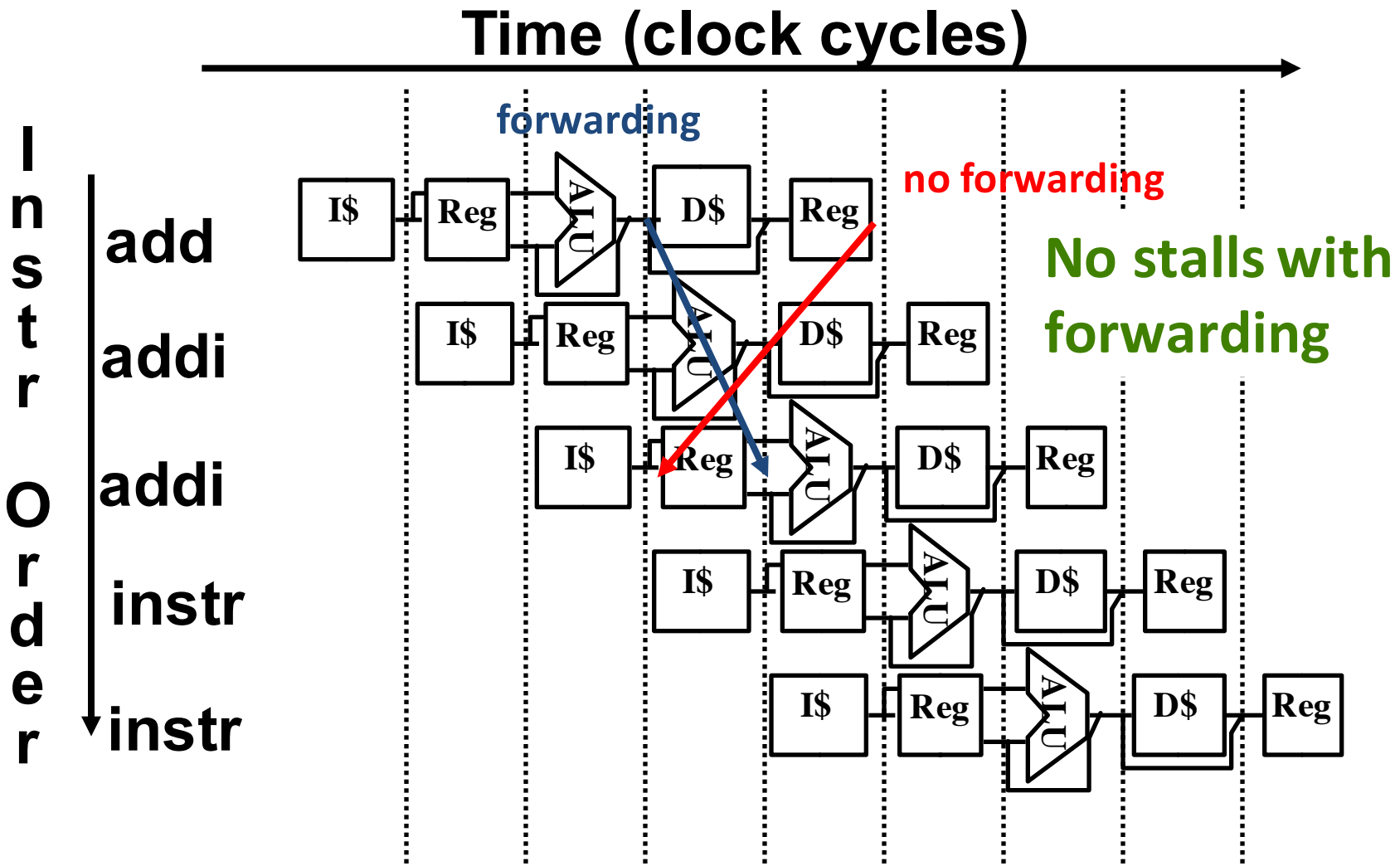
Summary

- Hazards reduce effectiveness of pipelining
  - Cause stalls/bubbles
- Structural Hazards
  - Conflict in use of datapath component
- Data Hazards
  - Need to wait for result of a previous instruction
- Control Hazards
  - Address of next instruction uncertain/unknown
  - *More to come next lecture!*

Code Sequence 1



Code Sequence 2



Code Sequence 3