# CS 61C:
# Great Ideas in Computer Architecture

## Lecture 13: *Pipelining*

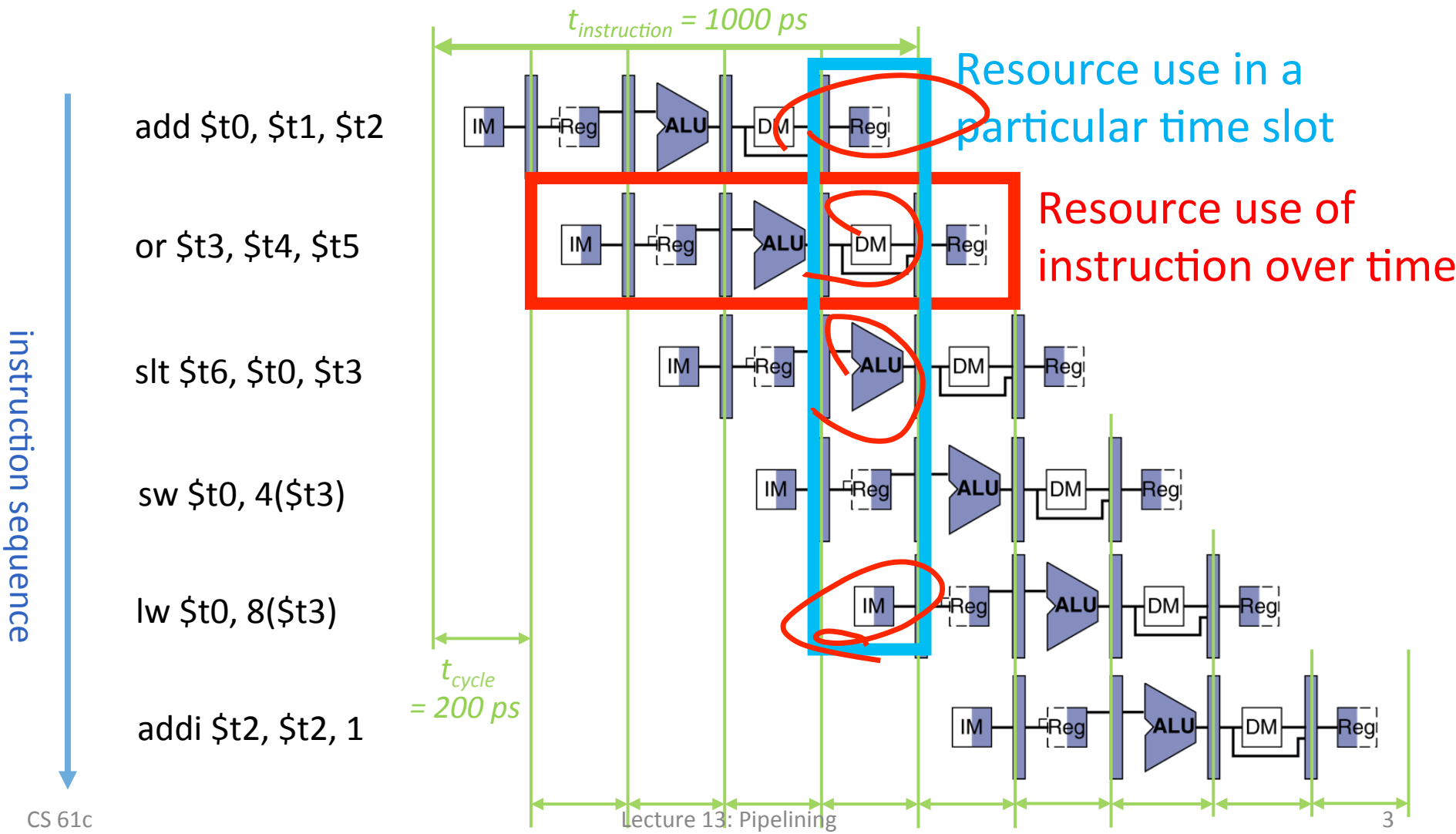
Bernhard Boser & Randy Katz

http://inst.eecs.berkeley.edu/~cs61c

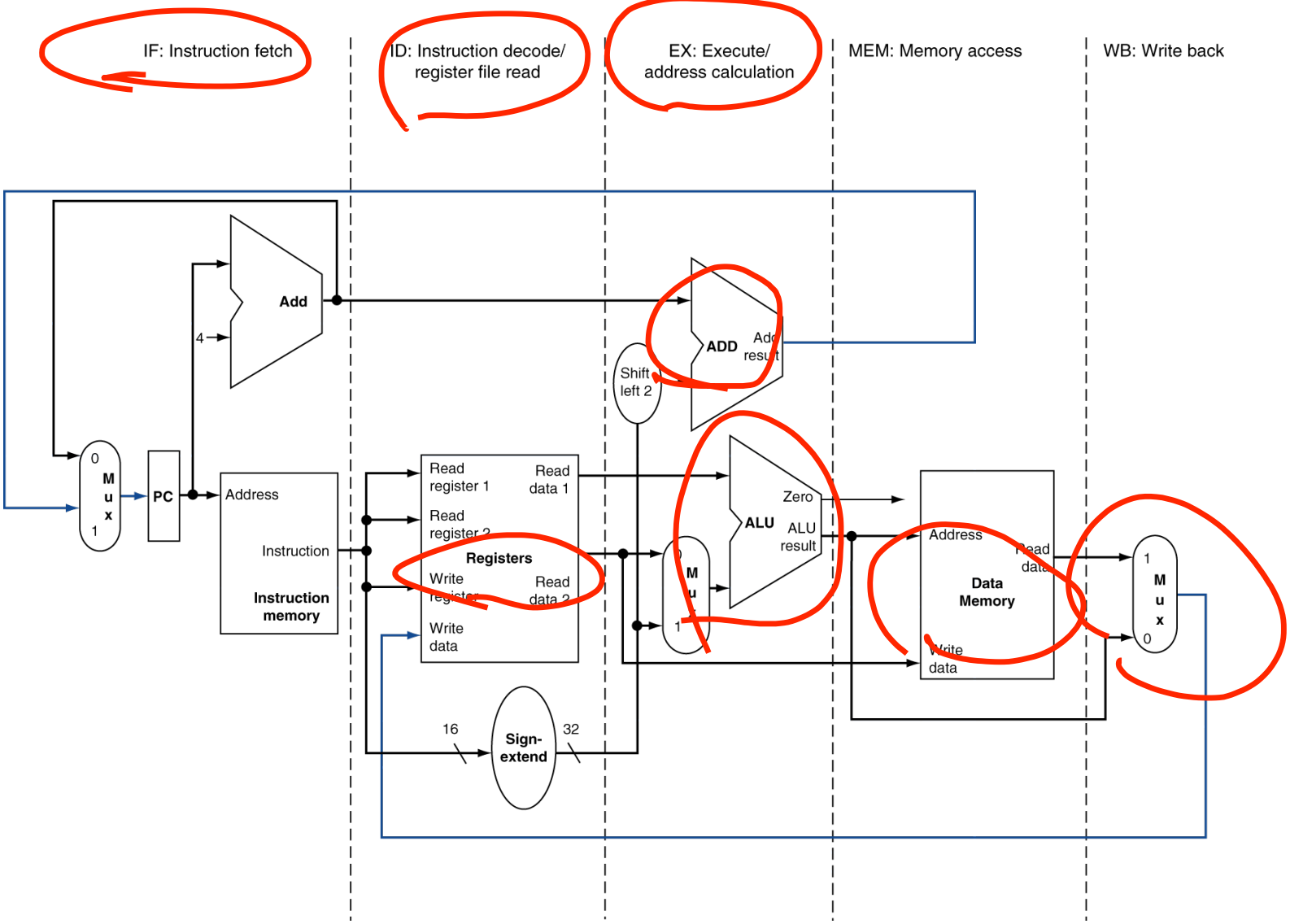
# Agenda

- **MIPS Pipeline**
- Pipeline Control
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

# MIPS Pipeline

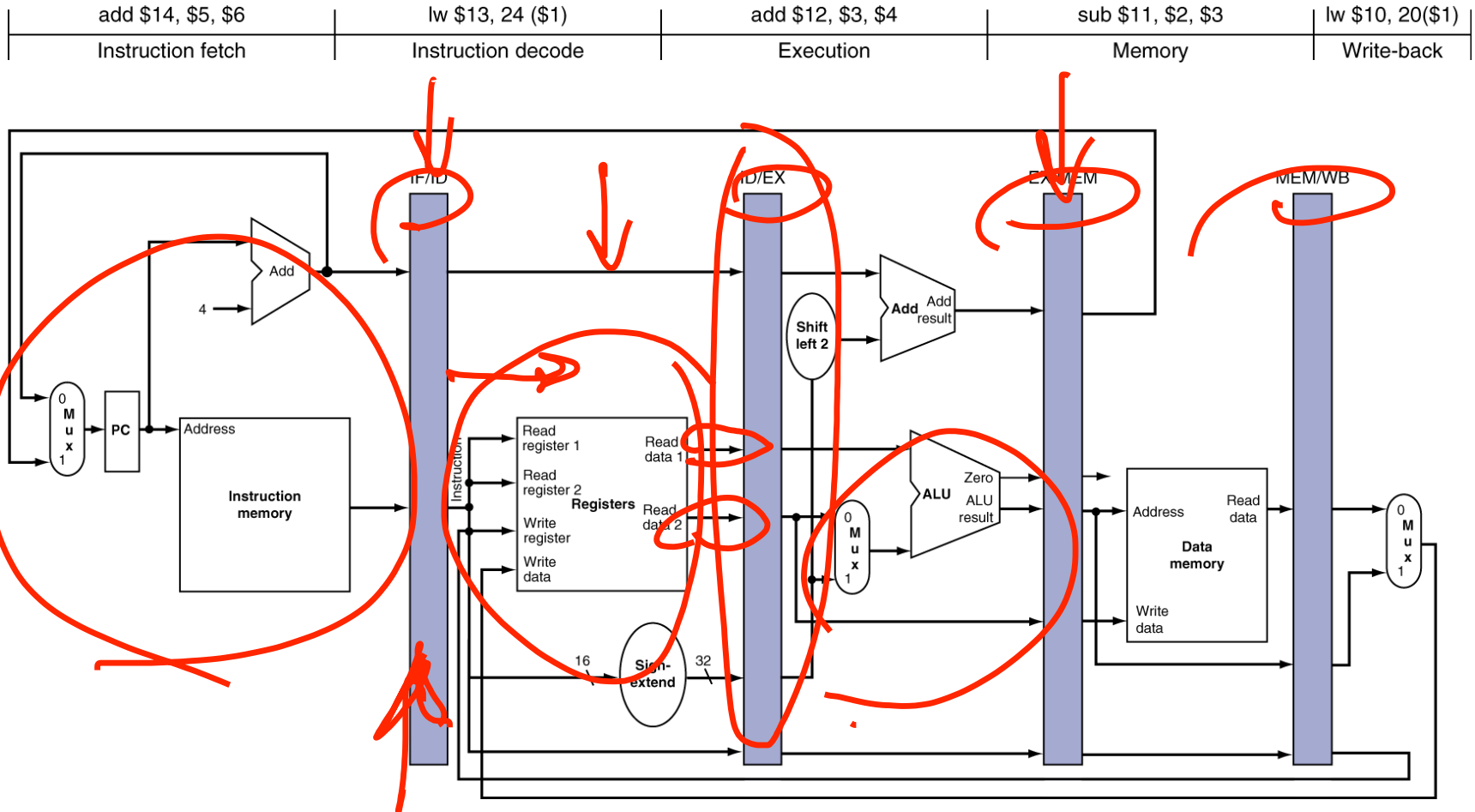


# Simplified MIPS Datapath

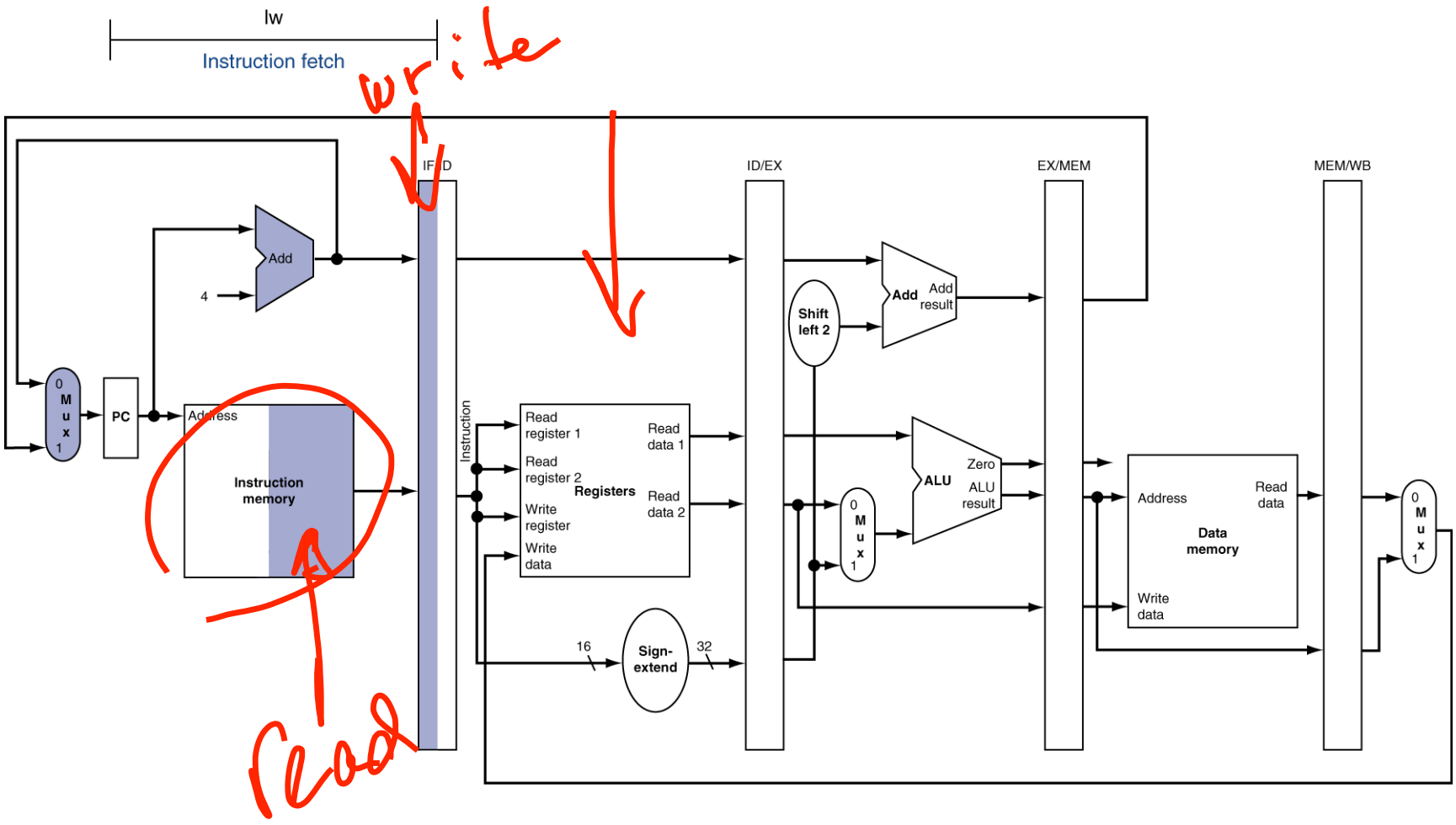


# Pipeline registers

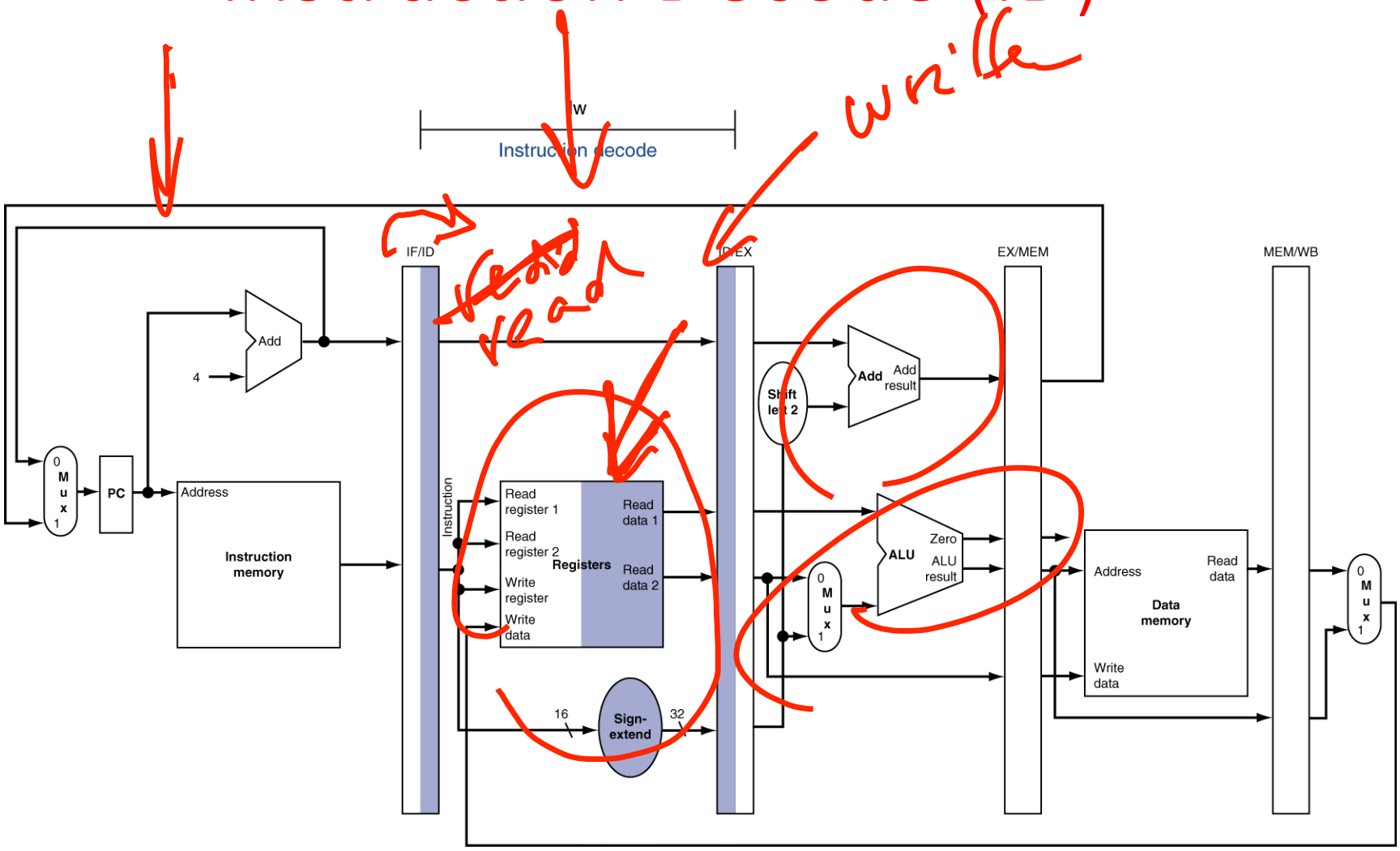
- Each stage operates on different data
  - E.g. ALU processes data fetched from registers during last cycle
  - → insert pipelining registers to hold data



# Instruction Fetch (IF)



# Instruction Decode (ID)

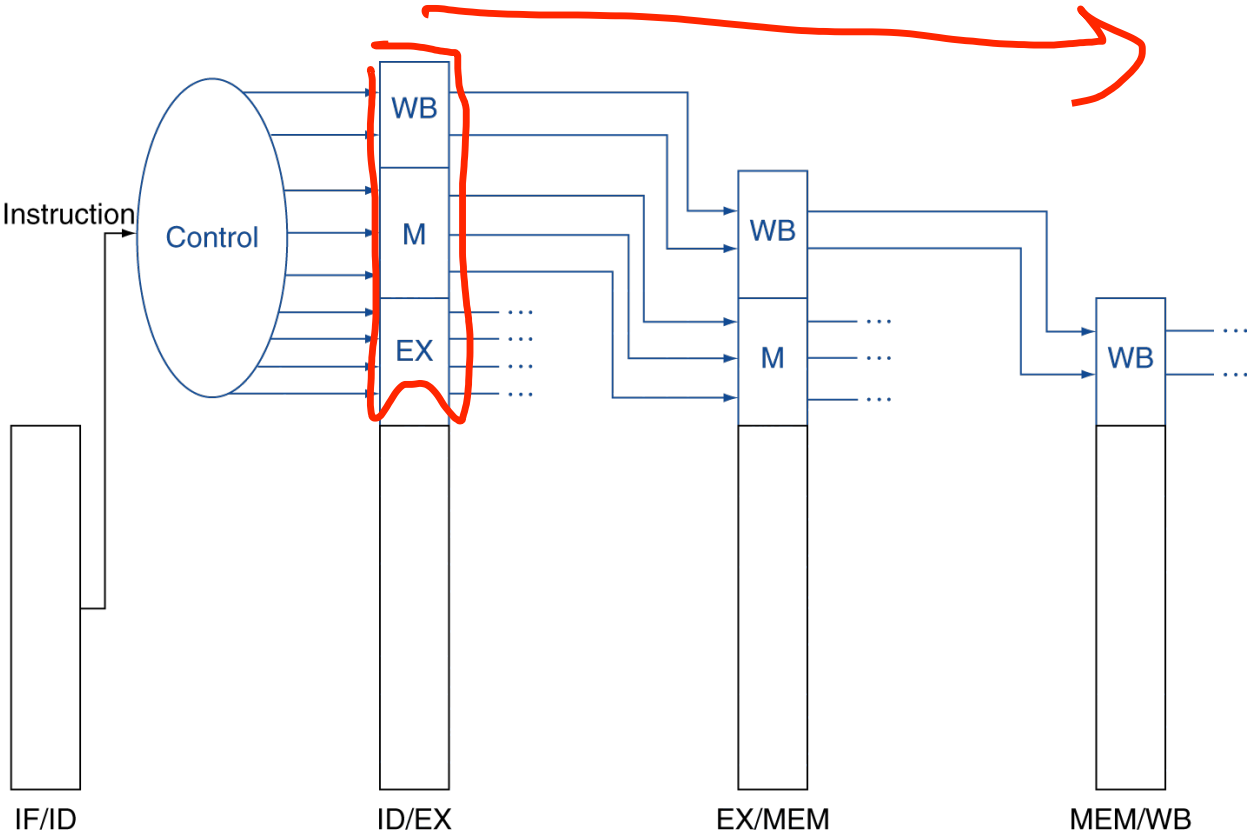


# Agenda

- MIPS Pipeline
- **Pipeline Control**
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

# Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
  - Information is stored in pipeline registers for use by later stages

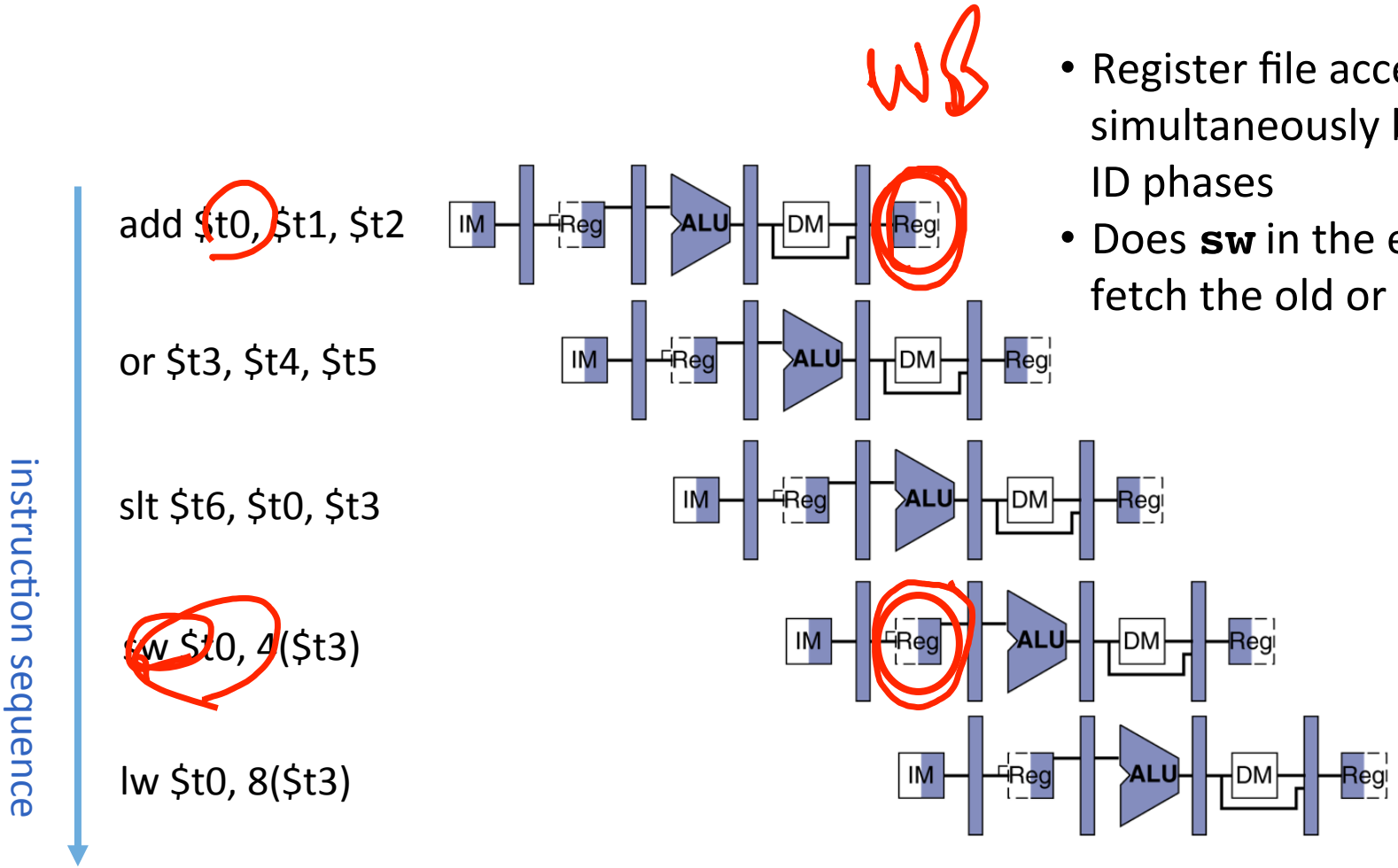


# Hazards Ahead

# Agenda

- MIPS Pipeline
- Pipeline Control
- Hazards
  - **Structural**
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

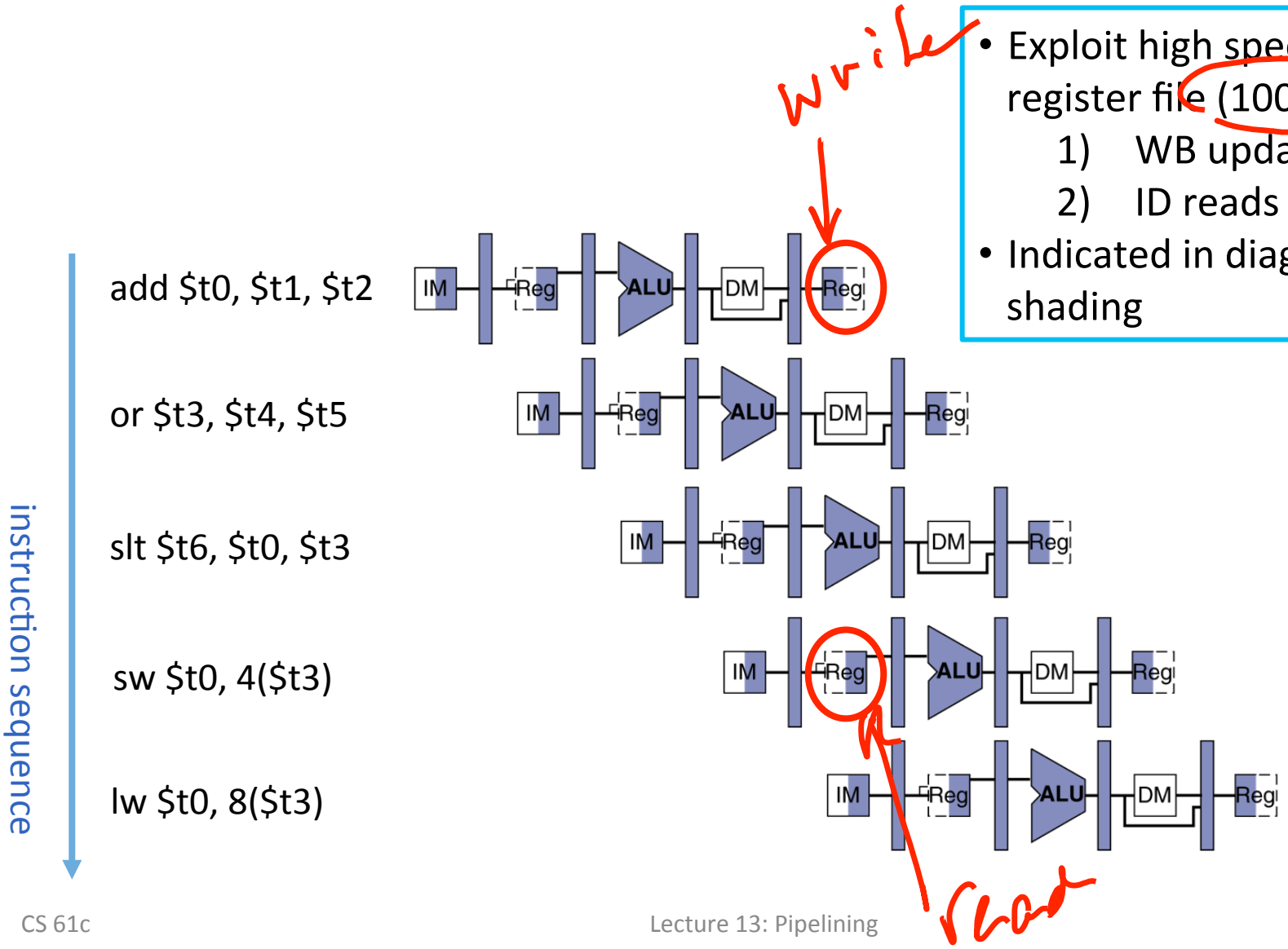
# Structural Hazard: Register Access




- Register file accessed simultaneously by WB and ID phases
- Does **`sw`** in the example fetch the old or new value?

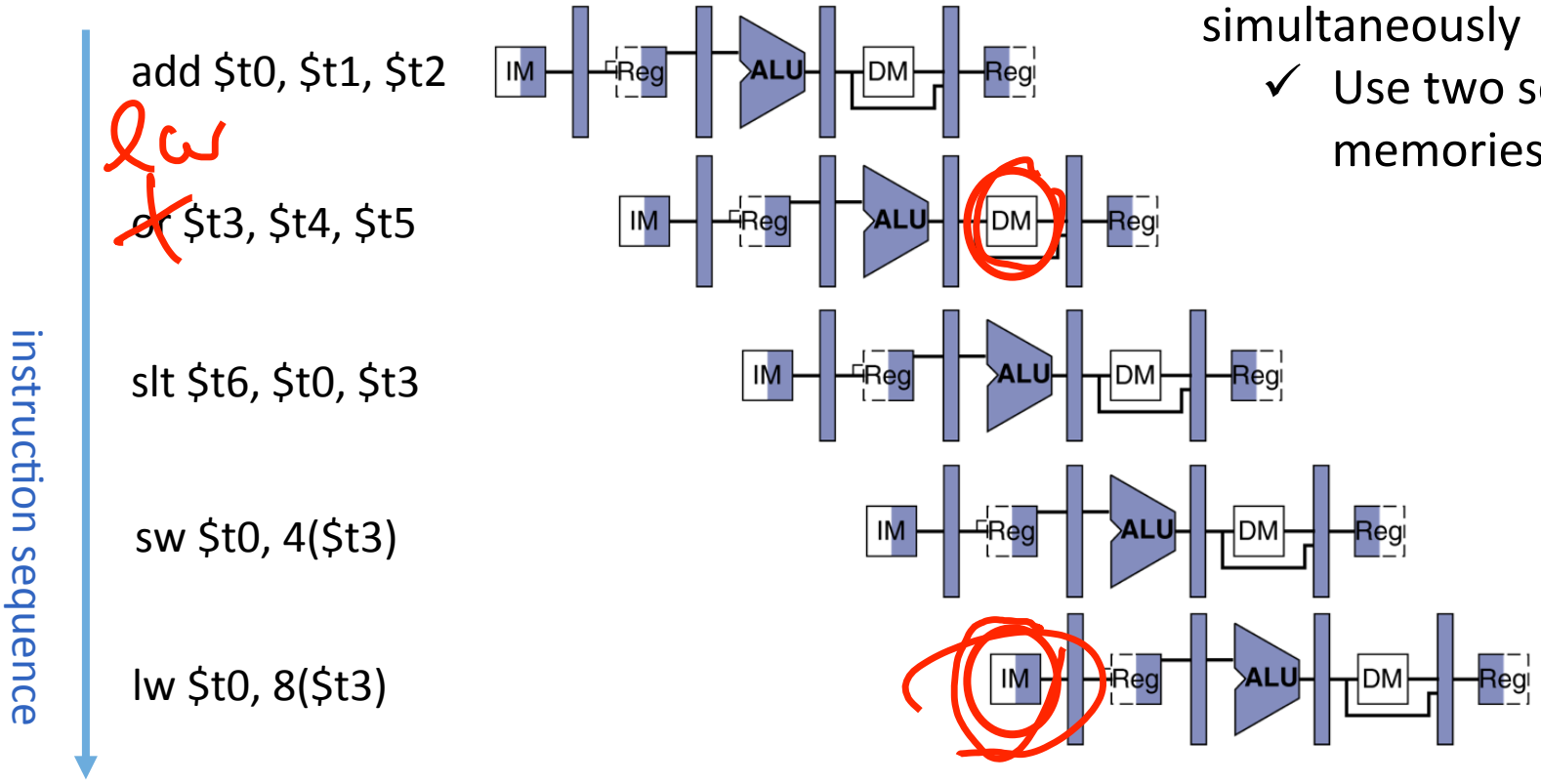
# Register Access Policy

- Exploit high speed of register file (100 ps)
  1) WB updates value
  2) ID reads new value
- Indicated in diagram by shading

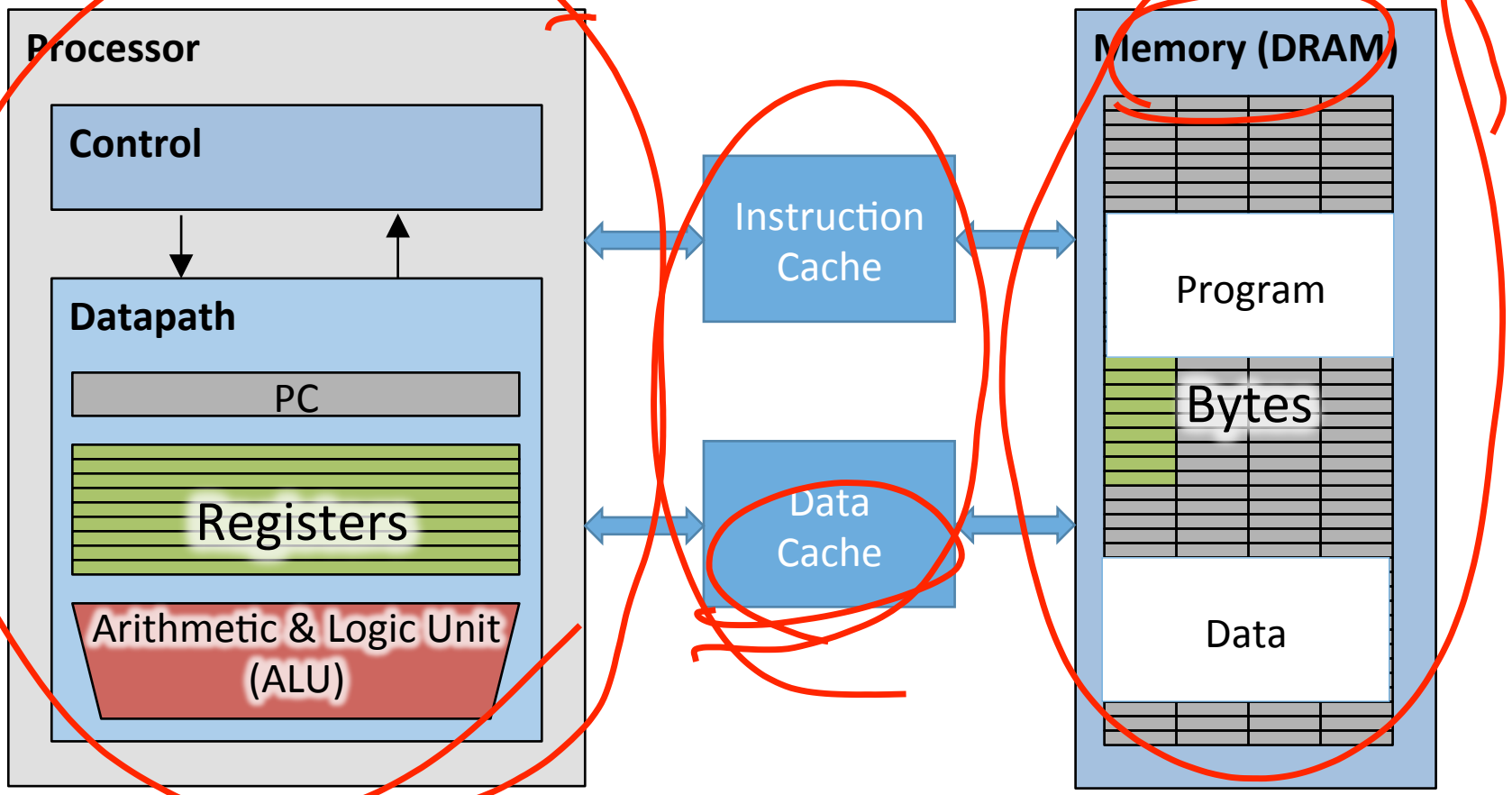


# Structural Hazard: Memory Access

- Instruction and data memory used simultaneously
  - ✓ Use two separate memories



# Instruction and Data Caches




Caches: small and fast "buffer" memories

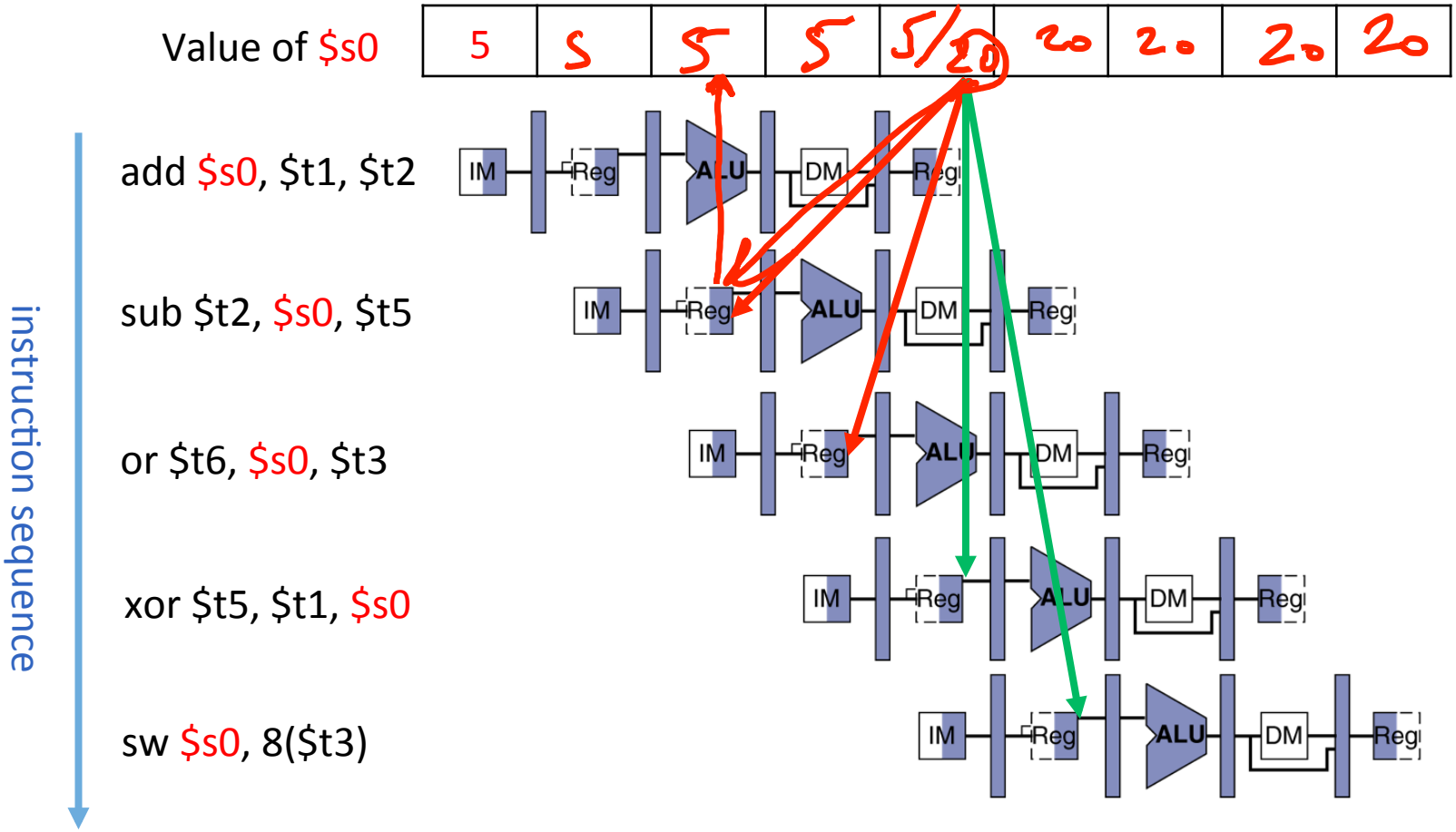
# Structure Hazards – Summary

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Without separate memories, instruction fetch would have to *stall* for that cycle
    - All other operations in pipeline would have to wait
- Pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

# Agenda

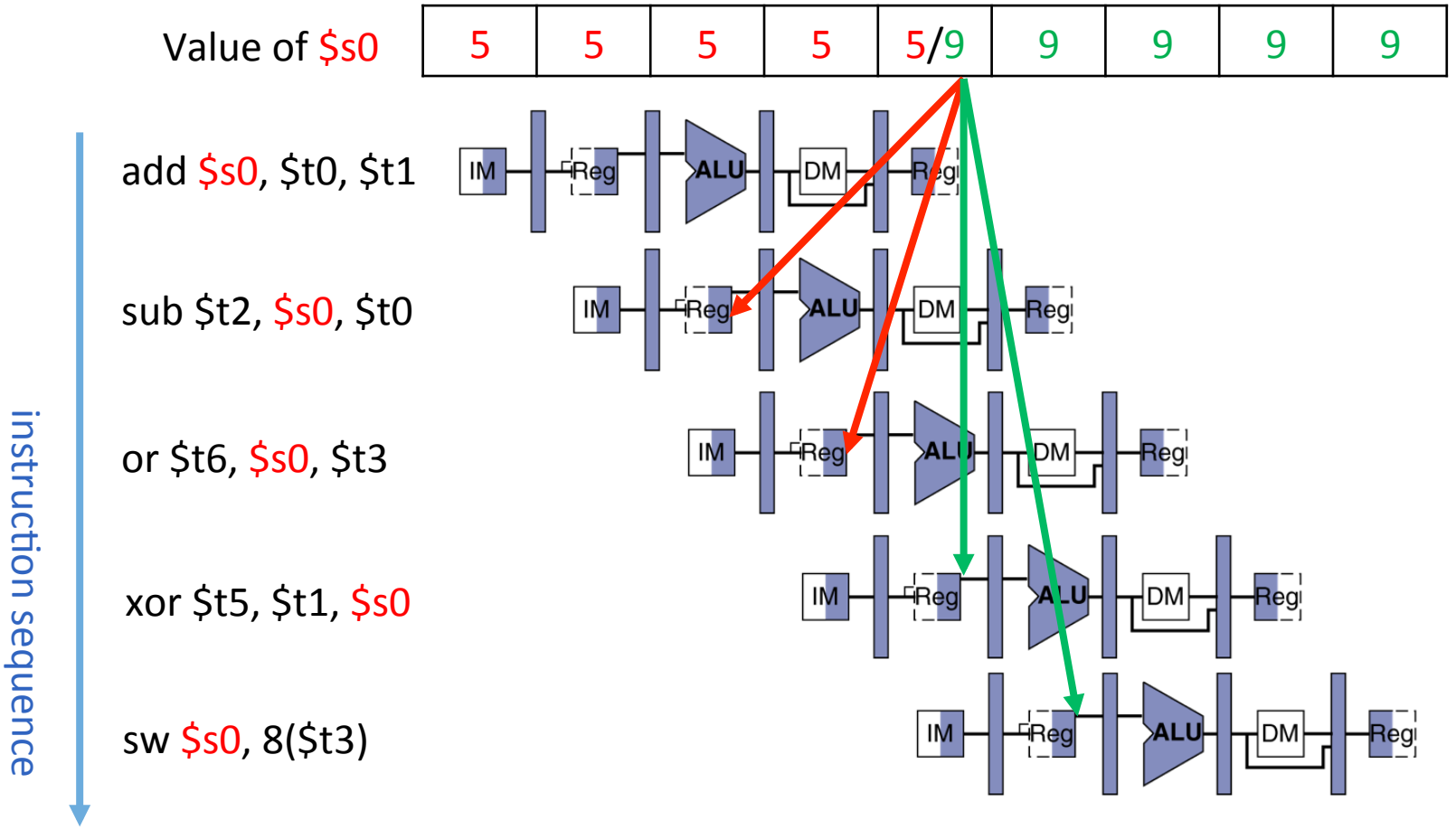
- MIPS Pipeline
- Pipeline Control
- Hazards
  - Structural
  - **Data**
    - **R-type instructions**
    - Load
  - Control
- Superscalar processors

# Data Hazard: ALU Result




Without fix, **`sub`** and **`or`** calculate wrong result!

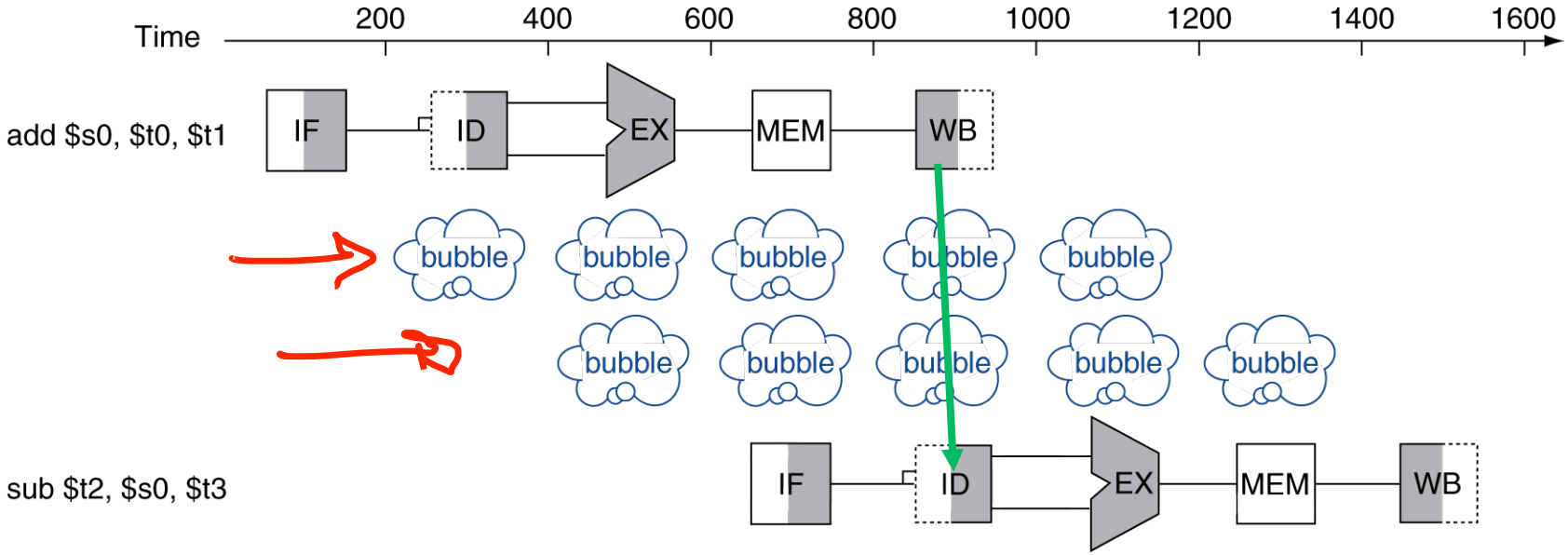
# Data Hazard: ALU Result




Without fix, **`sub`** and **`or`** calculate wrong result!

# Solution 1: Stalling

- Problem: Instruction depends on result from previous instruction
  - add $s0, $t0, $t1
    sub $t2, $s0, $t3

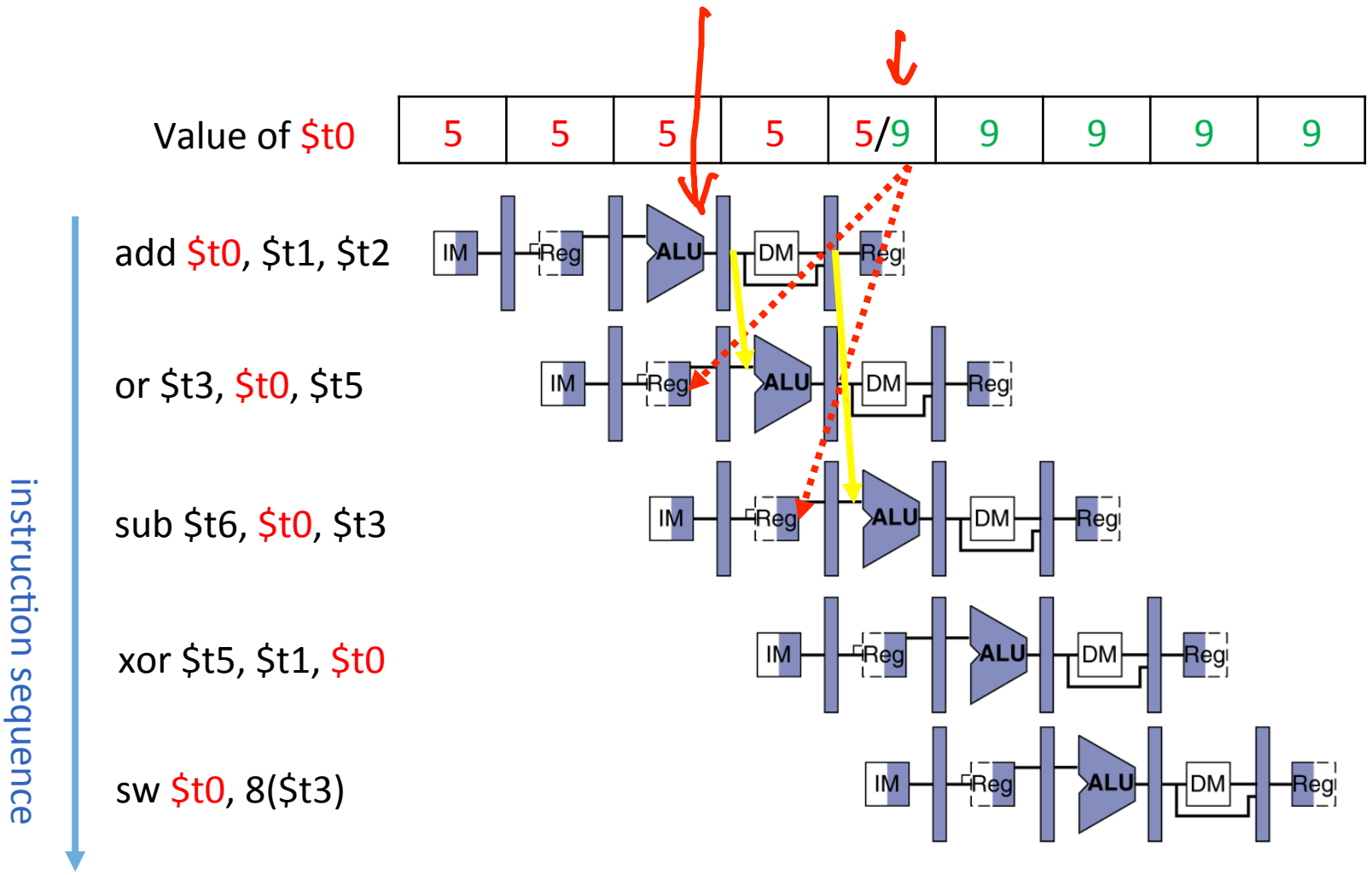



- Bubble:
  - effectively NOP: affected pipeline stages do "nothing"

# Stalls and Performance

- Stalls reduce performance
  - But stalls are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

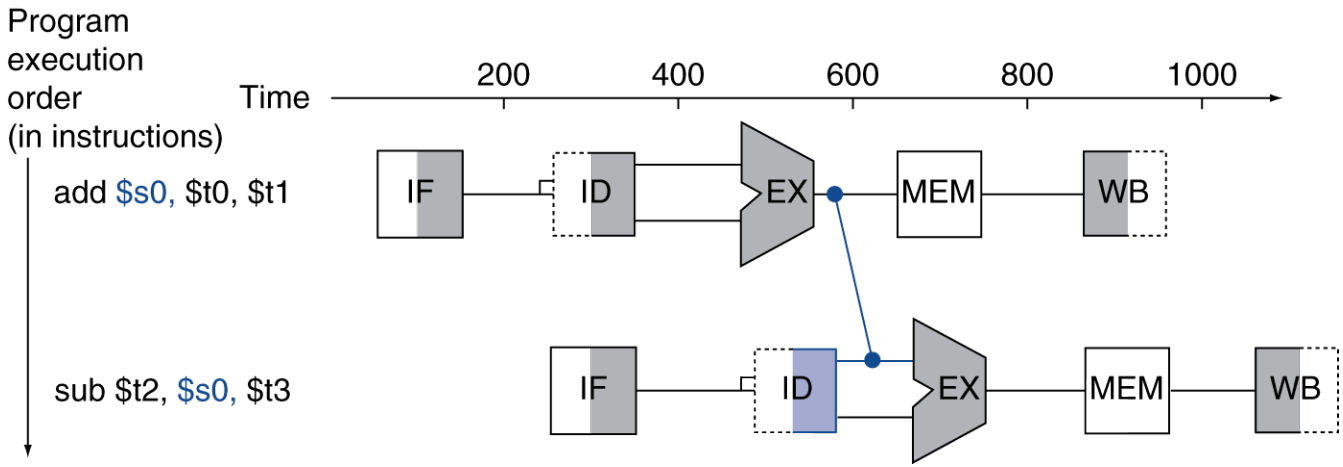
# Solution 2: Forwarding



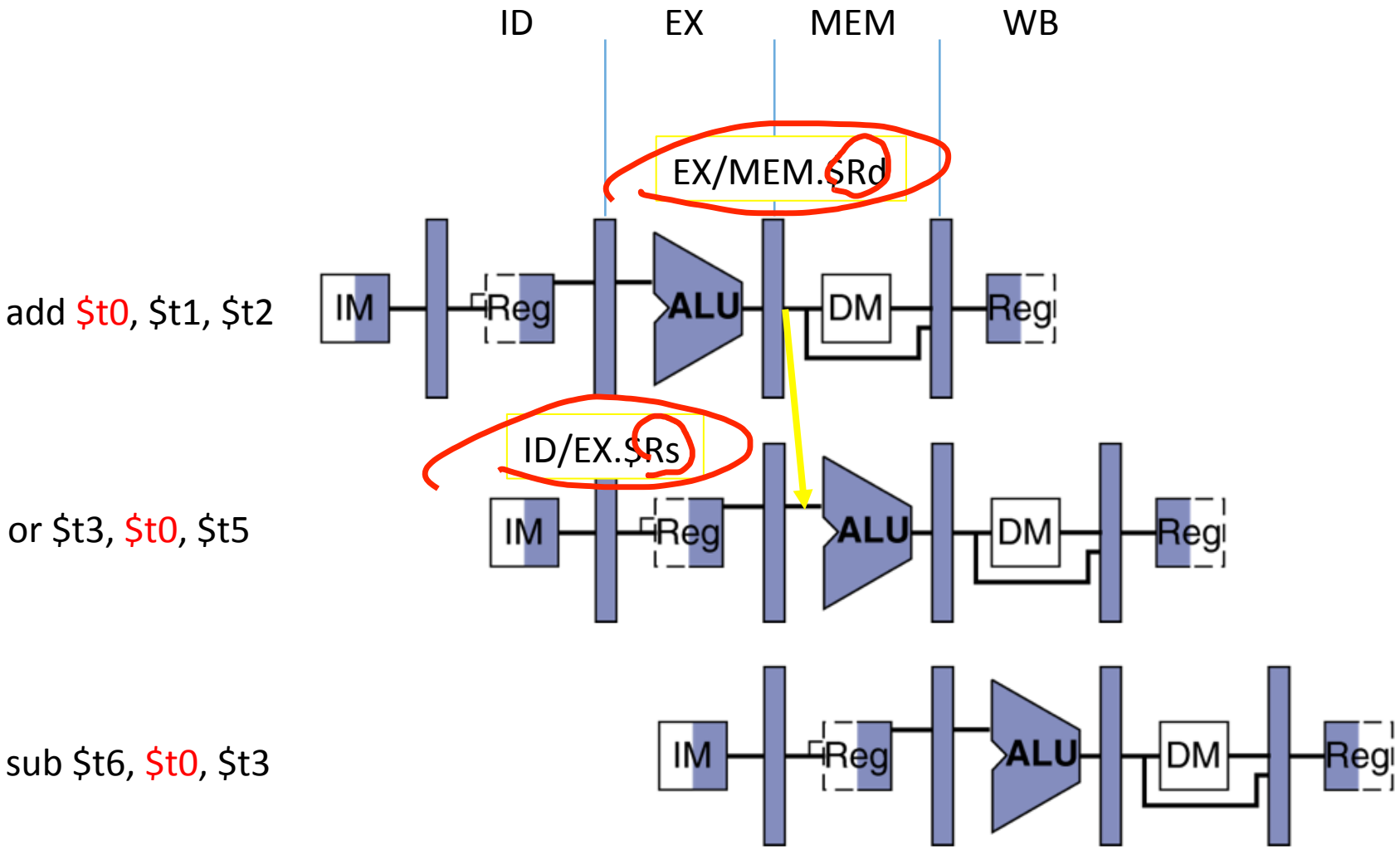

**Forwarding: grab operand from pipeline register, rather than register file**

# Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



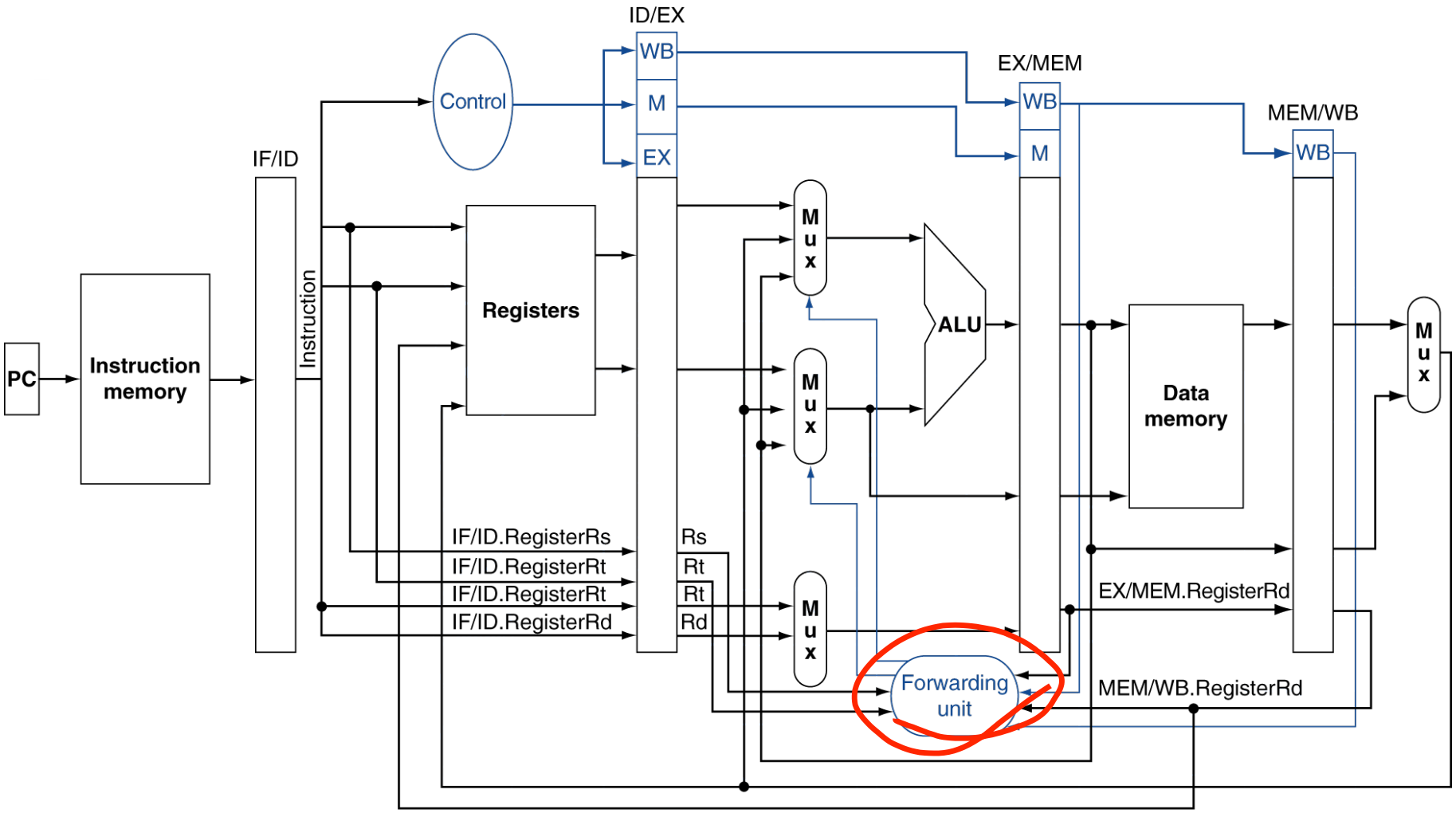
# 1) Detect Need for Forwarding (example)



# Complete Forwarding Conditions

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

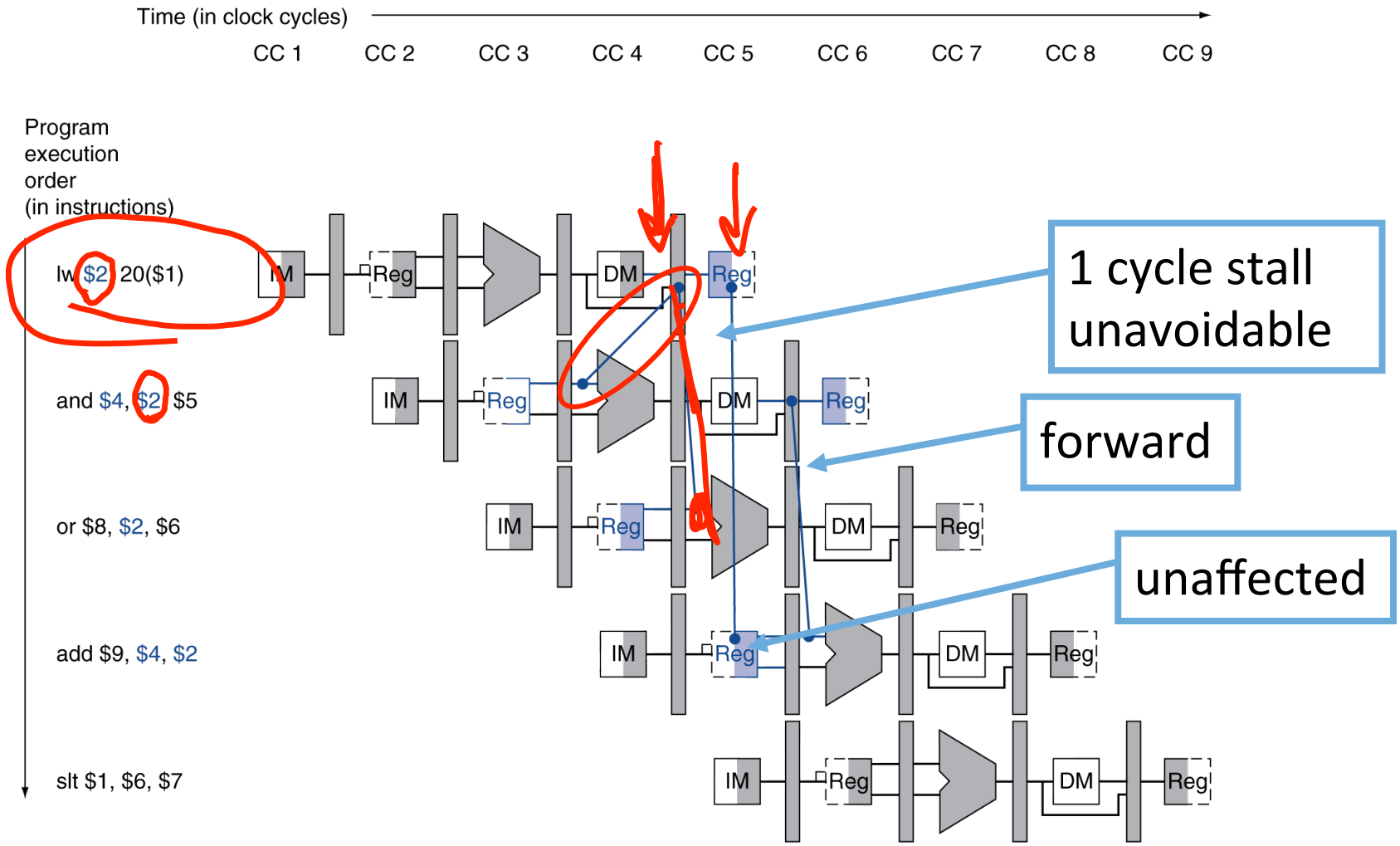
# 2) Datapath with Forwarding



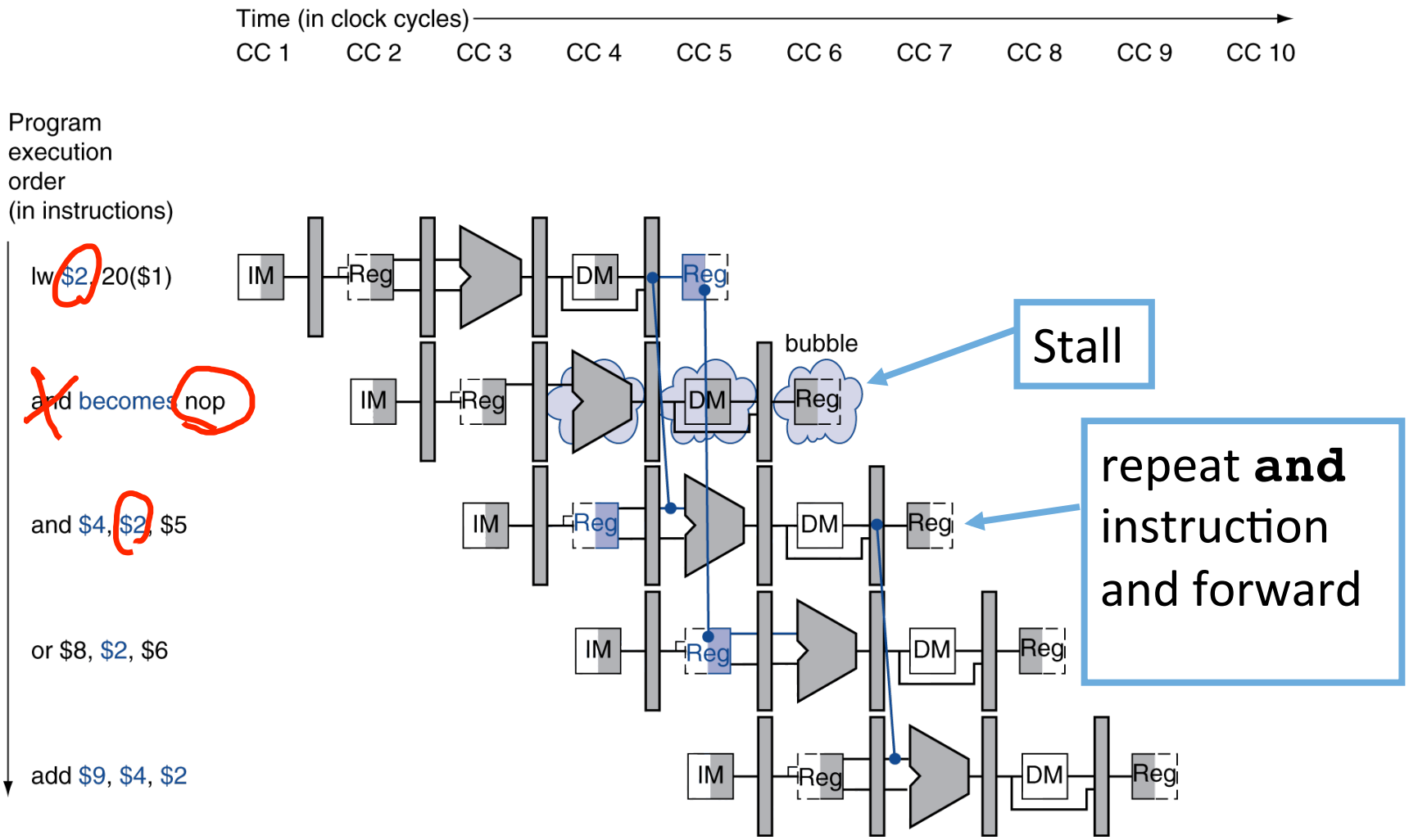
# Agenda

- MIPS Pipeline
- Pipeline Control
- Hazards
  - Structural
  - Data
    - R-type instructions
    - **Load**
  - Control
- Superscalar processors

# Load Data Hazard



# Stall Pipeline



# Your Turn

How many cycles (pipeline fill+process+drain) does it take to *complete* the following code?

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

| Answer | cycles |
|---|---|
| A | 7 |
| B | 9 |
| C | 11 |
| D | 13 |
| E | 14 |

# Your Turn

How many cycles (pipeline fill+process+drain) does it take to *complete* the following code?

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

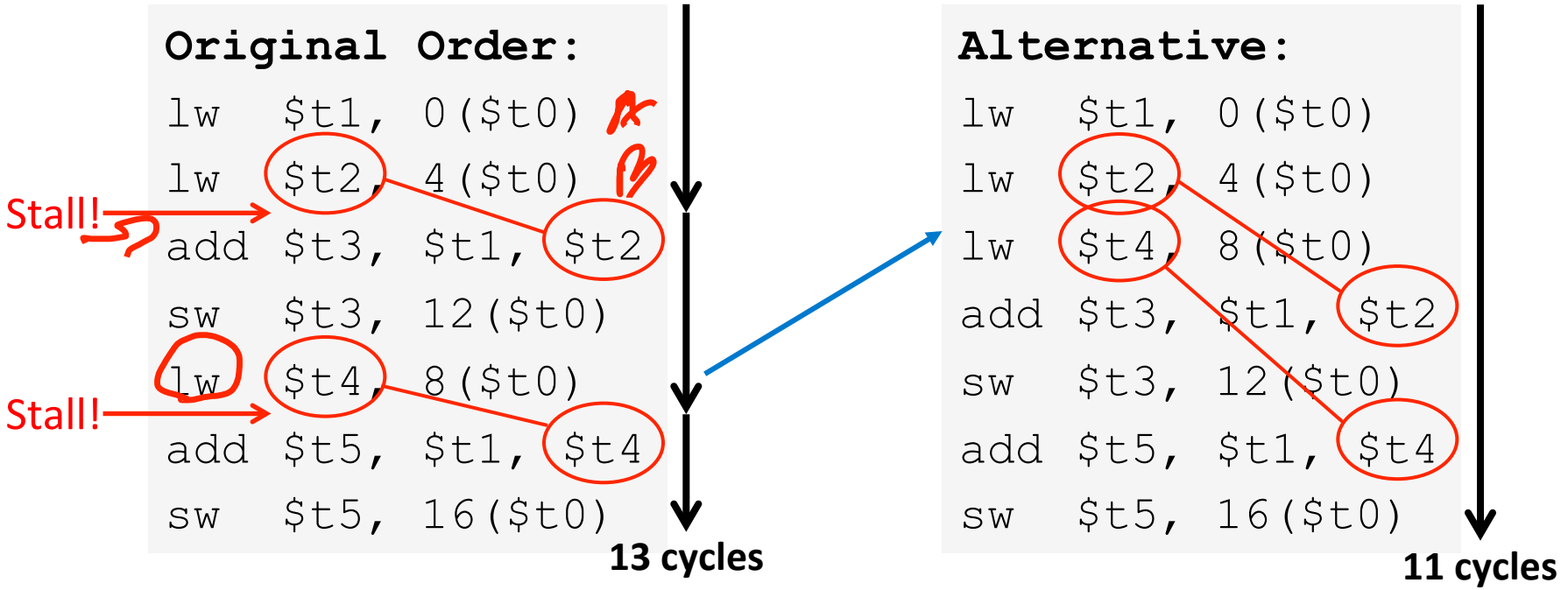
| Answer | cycles |
|---|---|
| A | 7 |
| B | 9 |
| C | 11 |
| D | 13 |
| E | 14 |

# `lw` Data Hazard

- Slot after a load is called a *load delay slot*
  - If that instruction uses the result of the load, then the hardware will stall for one cycle
  - Equivalent to inserting an explicit **`nop`** in the slot
    - except the latter uses more code space
  - Performance loss
- Idea:
  - Put unrelated instruction into load delay slot
  - No performance loss!

# Code Scheduling to Avoid Stalls

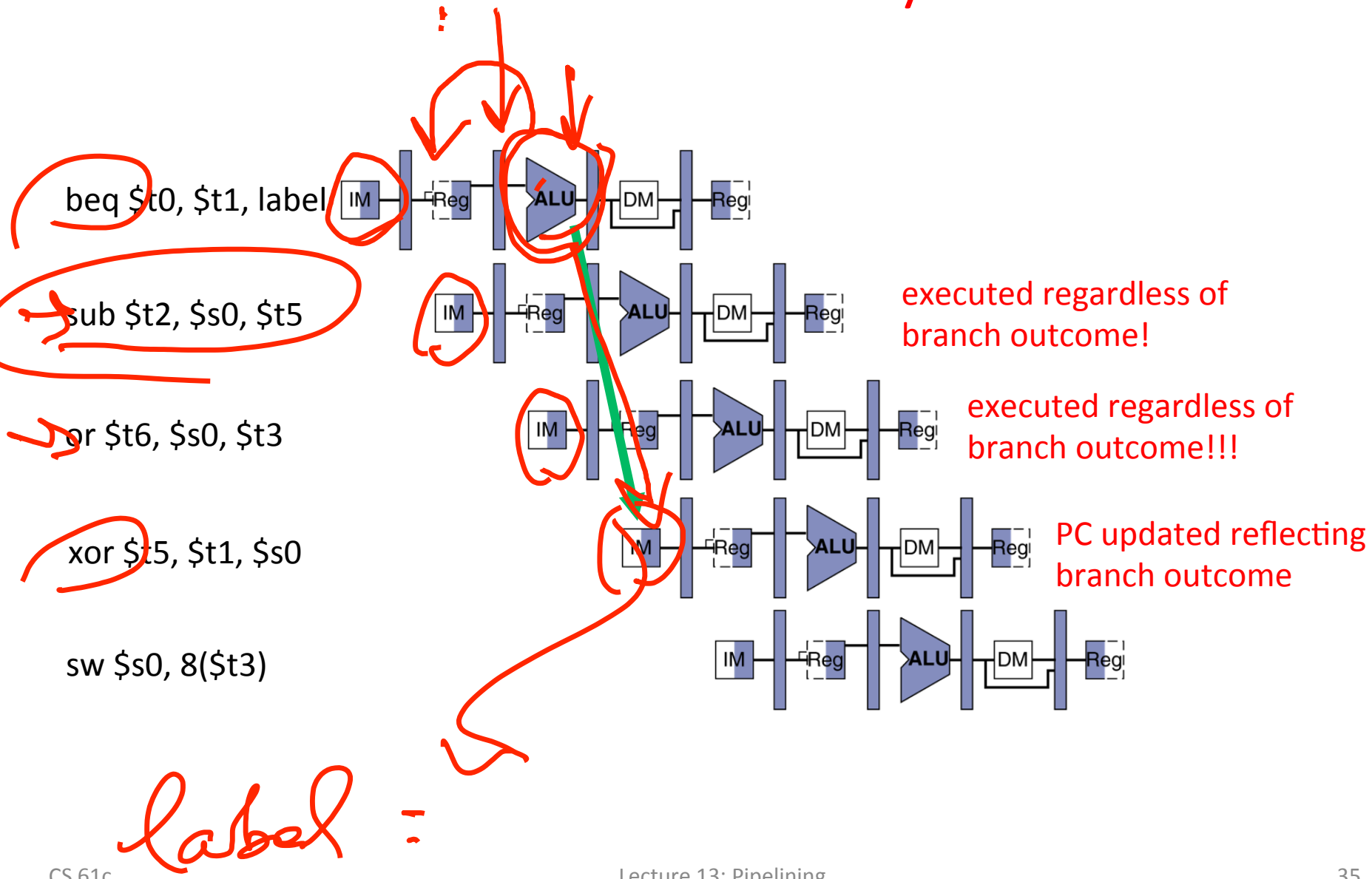
- Reorder code to avoid use of load result in the next instruction!
- MIPS code for `D=A+B; E=A+C;`



# Agenda

- MIPS Pipeline
- Pipeline Control
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - **Control**
- Superscalar processors

# Branch Delay



# Dealing with Branch Control Hazard

- Without hardware modification, 2 instructions after branch are executed
  - regardless of branch outcome
- Hardware modification:
  - move branch decision to ID stage
  - requires dedicated equality comparator
- Only 1 instruction after branch executed unconditionally
  - "branch delay slot"
  - MIPS assembler/compiler try to schedule unaffected instruction
  - **`nop`** if none

# Branch Delay Slot Scheduling Example

**Original sequence:**

```
or   $8, $9, $10
add  $1, $2, $3
sub  $4, $5, $6
beq  $1, $4, Exit
nop
xor  $10, $1, $11
```

**Improved sequence:**

```
add  $1, $2,$3
sub  $4, $5, $6
beq  $1, $4, Exit
or   $8, $9, $10
xor  $10, $1, $11
```

# Alternatives to Branch Delay Slot

- Predict ("guess") branch outcome
  - Transparent to programmer
  - Cancel instructions in pipeline if guess was wrong
    - "flush pipeline"
- Typical solutions:
  - Always guess that branch is not taken
  - Do same as last time branch instruction was executed
    - requires special "book-keeping hardware"
  - Solution used by most current processors
    - except MIPS

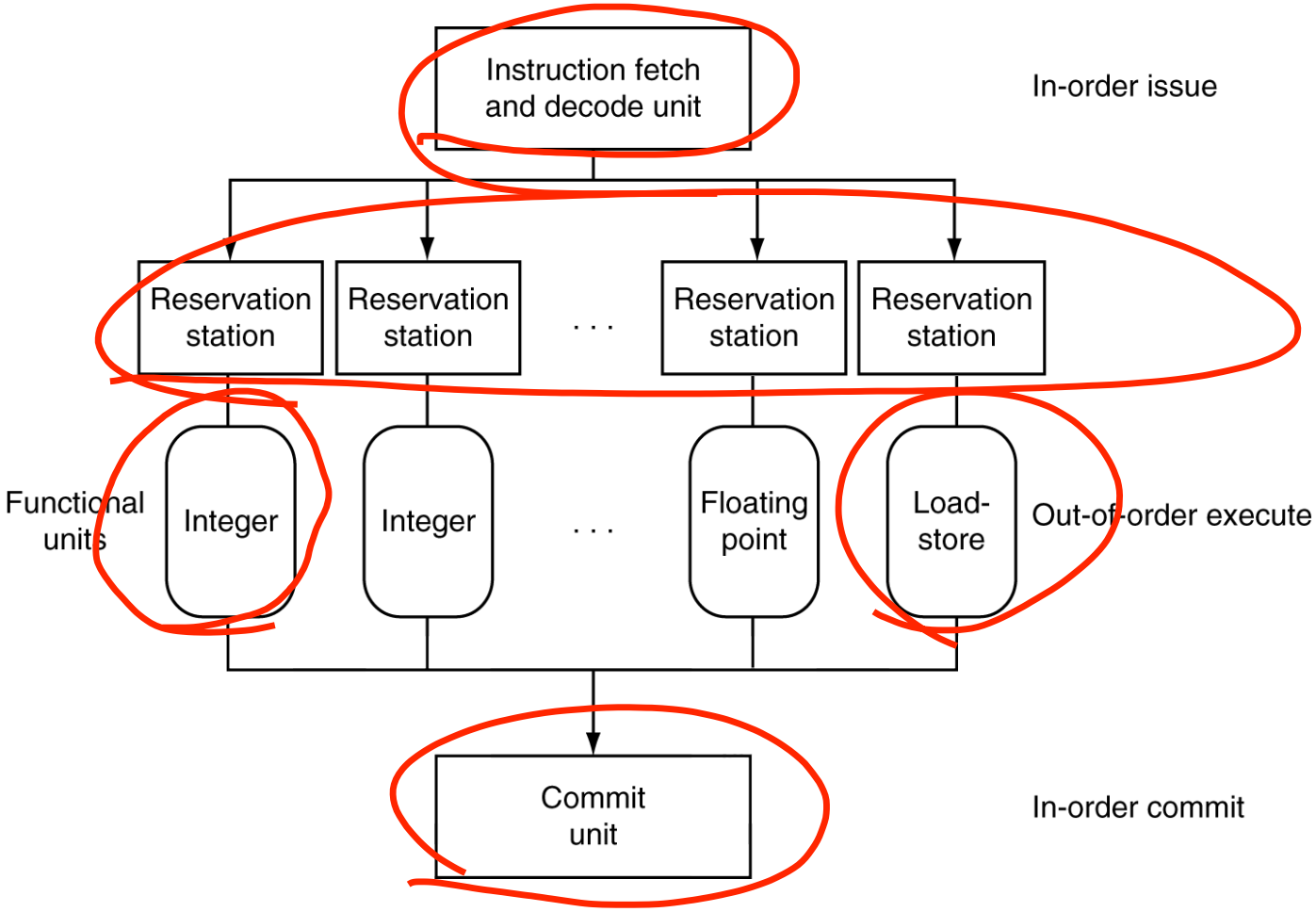
# Agenda

- MIPS Pipeline
- Pipeline Control
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- **Superscalar processors**

# Increasing Processor Performance

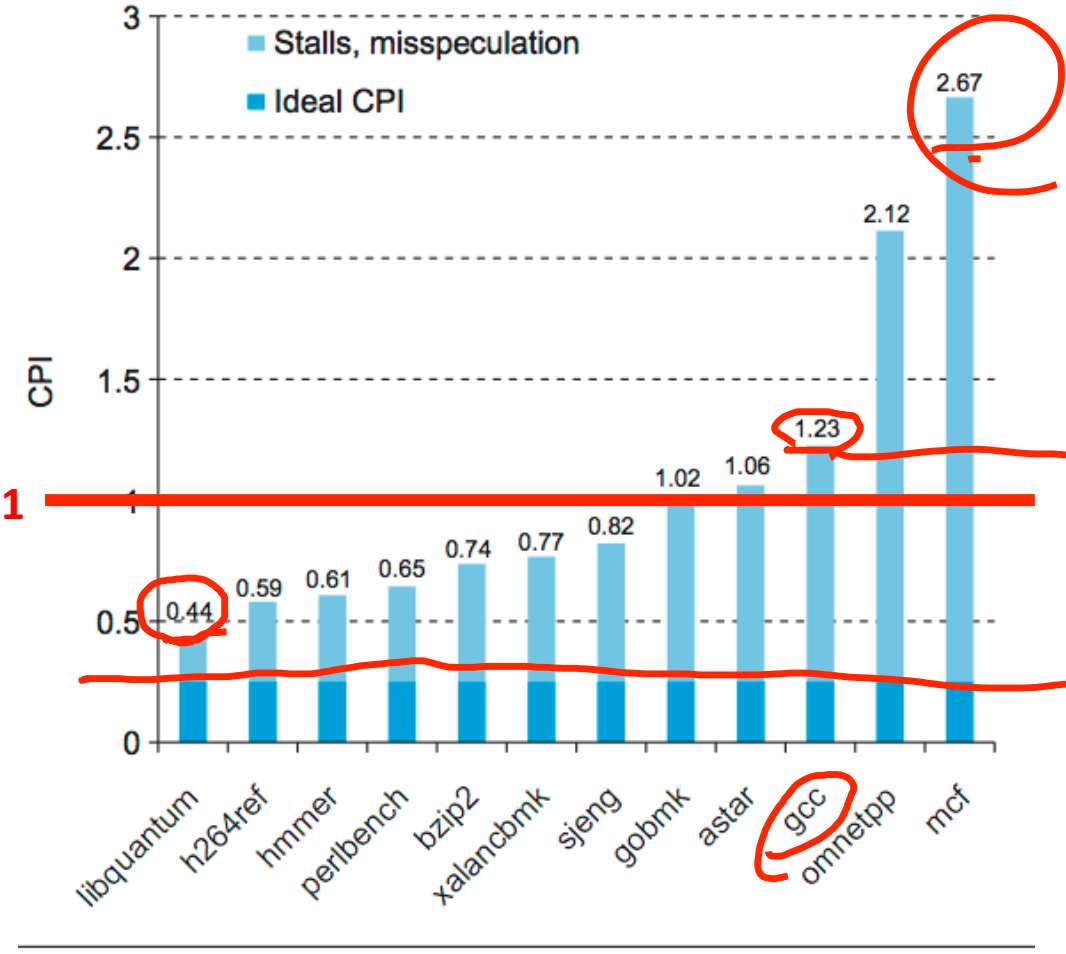
1. Clock rate
   - Limited by technology and power dissipation
2. Pipelining
   - "Overlap" instruction execution
   - Deeper pipeline: 5 => 10 => 15 stages
     - Less work per stage → shorter clock cycle
     - But more potential for hazards (CPI > 1)
3. Multi-issue "super-scalar" processor
   - Multiple execution units (ALUs)
     - Several instructions executed simultaneously
     - CPI < 1 (ideally)

# Superscalar Processor




P&H p. 340

# Benchmark: CPI of Intel Core i7




**CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.**

CPI = 1

P&H p. 350

# In Conclusion

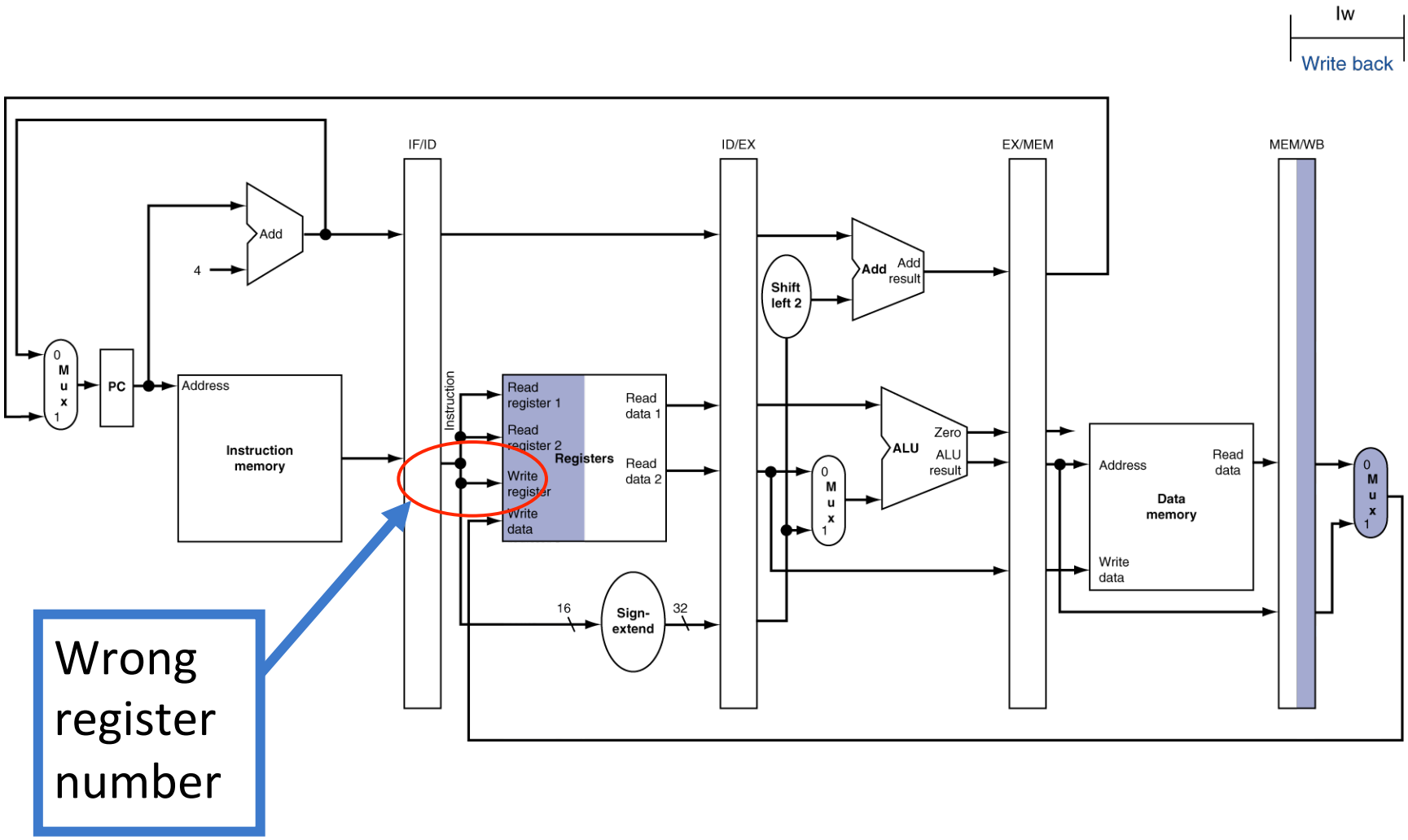
- Pipelining increases throughput by overlapping execution of multiple instructions
- All pipeline stages have same duration
  - Choose partition that accommodates this constraint
- Hazards potentially limit performance
  - Maximizing performance requires programmer/compiler assistance
  - E.g. Load and Branch delay slots
- Superscalar processors use multiple execution units for additional instruction level parallelism
  - Performance benefit highly code dependent

# Extra Slides

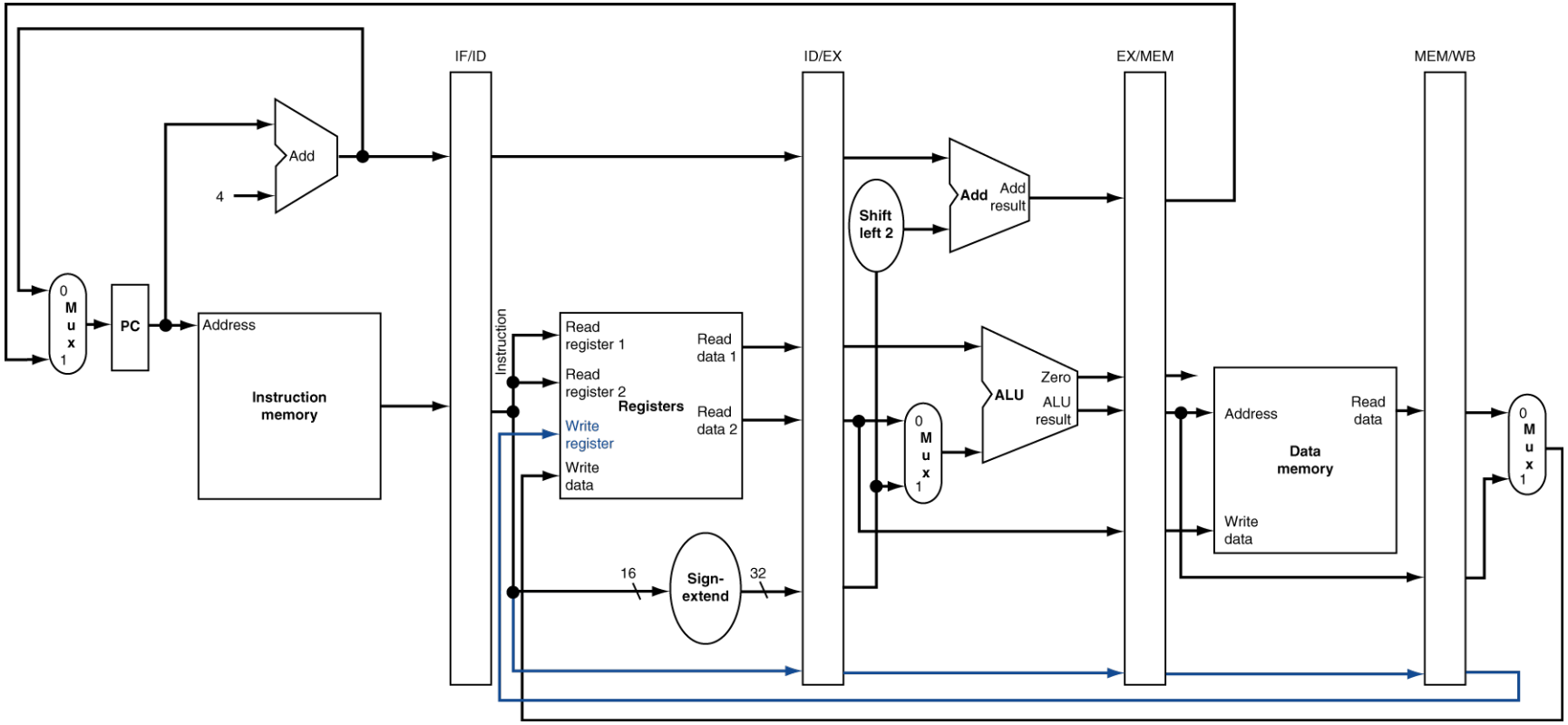
# Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easy to fetch and decode in one cycle
    - Versus x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Decode and read registers in one step
  - Load/store addressing
    - Calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

# Write-Back (WB, **`lw`**)



# Corrected Datapath for WB



# Superscalar Processor

- Multiple issue "superscalar"
  - Replicate pipeline stages ⇒ multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - Dependencies reduce this in practice
- "Out-of-Order" execution
  - Reorder instructions dynamically in hardware to reduce impact of hazards
- *CS152 discusses these techniques!*