CSE 140
Computer Architecture

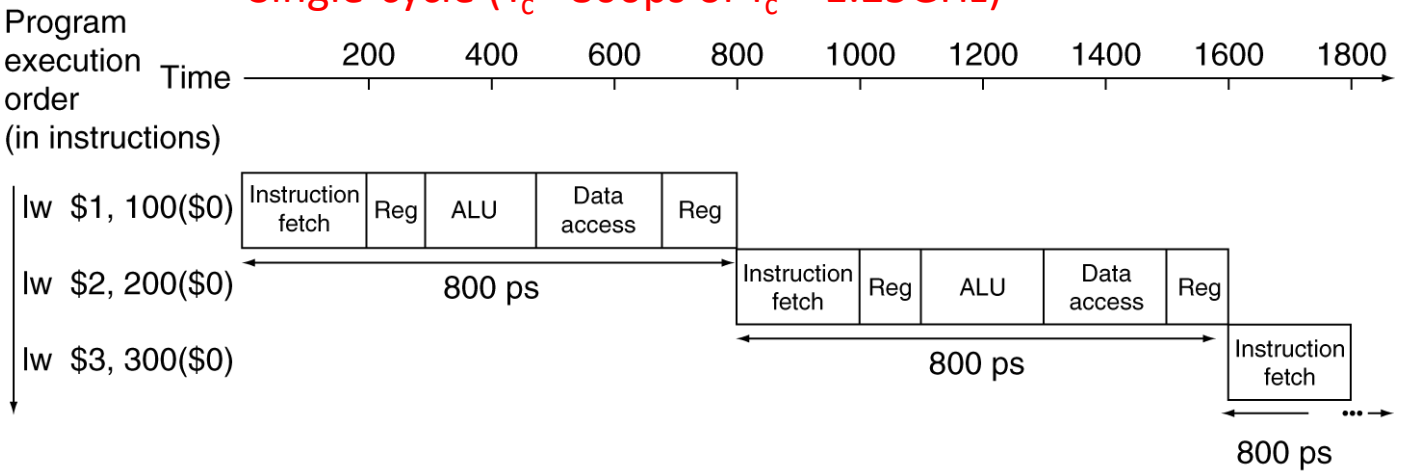
Lecture 5 – Pipelining (2)

Announcement

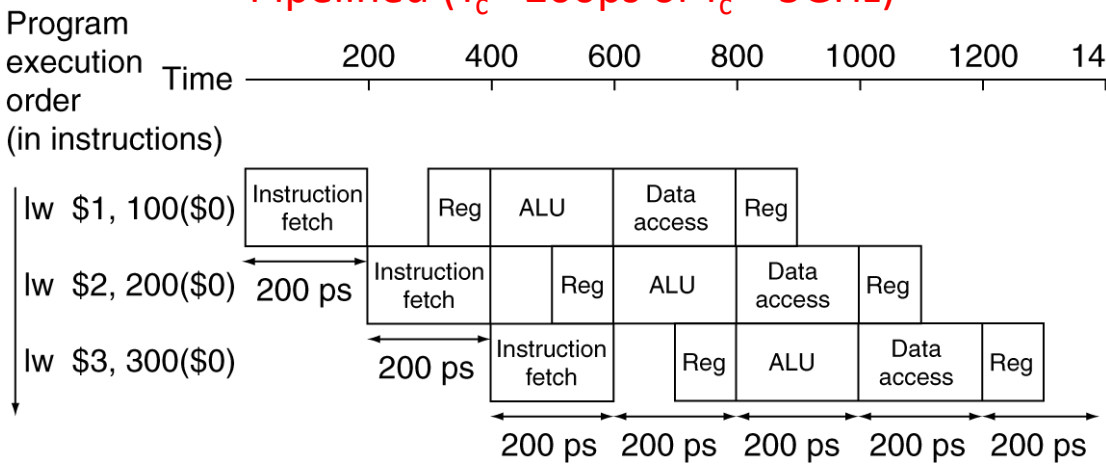
- Lab #1 this week
  - Due in one week
- Project #1 out this Friday
  - Due 10/11 (Monday) at 11:59pm
- Reading assignment #3
  - Chapter 5.1 – 5.4
    - Do all Participation Activities in each section
    - Access through CatCourses
    - Due Tuesday (9/19) at 11:59pm

Pipeline Performance

Single-cycle (T_c= 800ps or f_c = 1.25GHz)



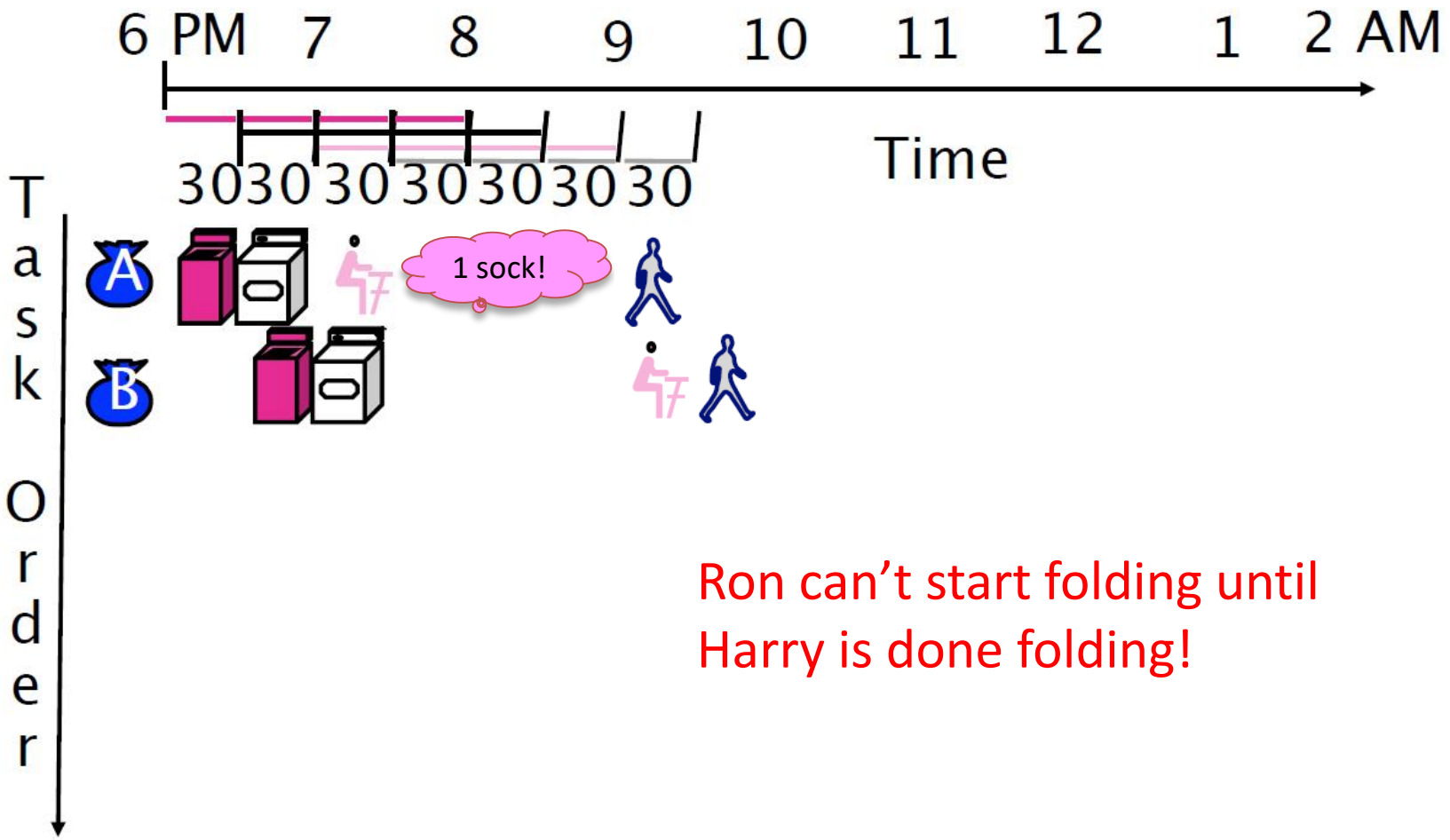
Pipelined (T_c= 200ps or f_c = 5GHz)



Pipeline Speedup

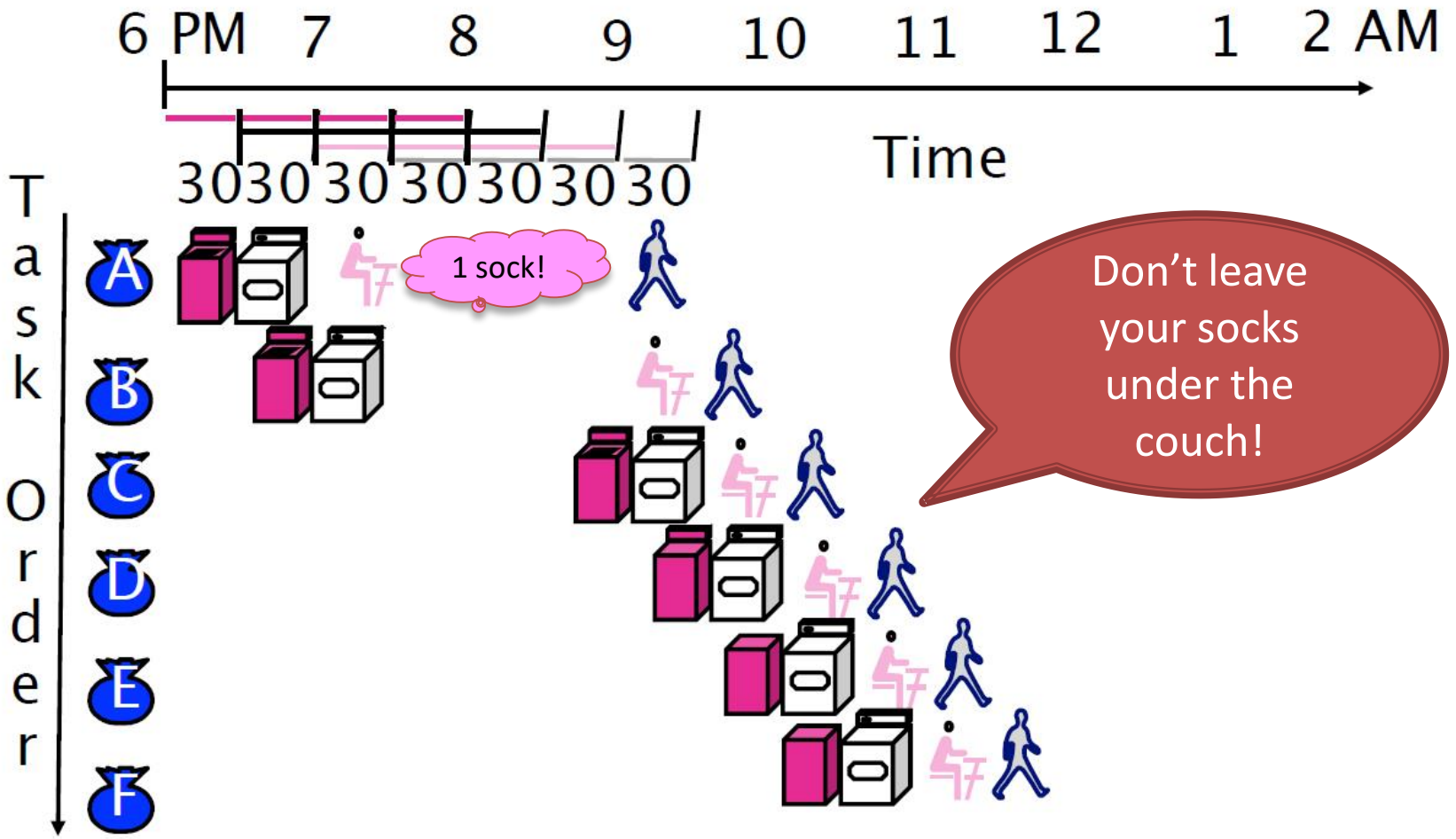
- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$
    $$= \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipeline Hazard:



Ron can't start folding until Harry is done folding!

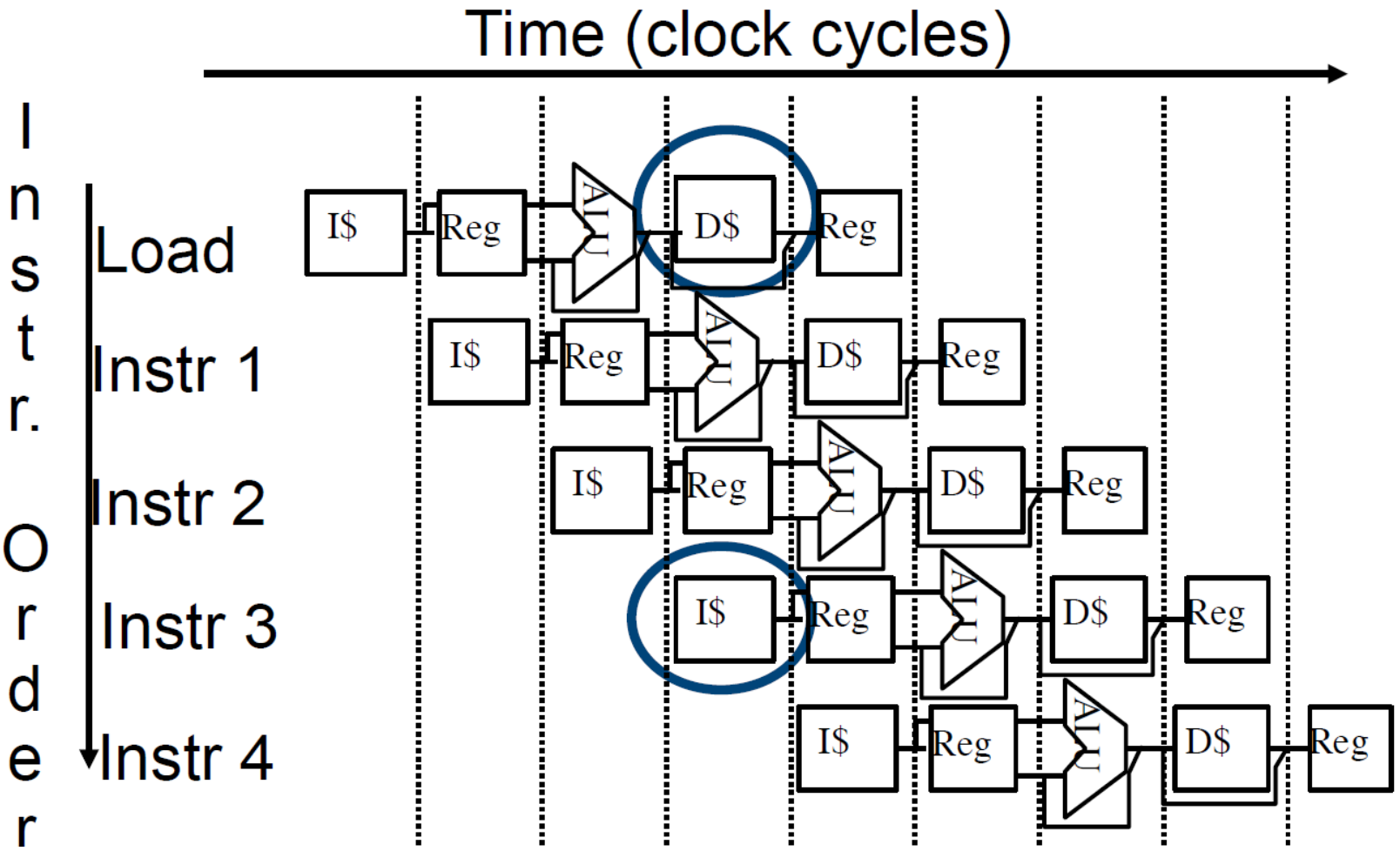
Pipeline Hazard:



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 (There is only one table for folding and stashing)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Pipelining of branches causes later instruction fetches to wait for the result of the branch
- These might result in pipeline stalls or "bubbles" in the pipeline.

Structural Hazard #1: Single Memory

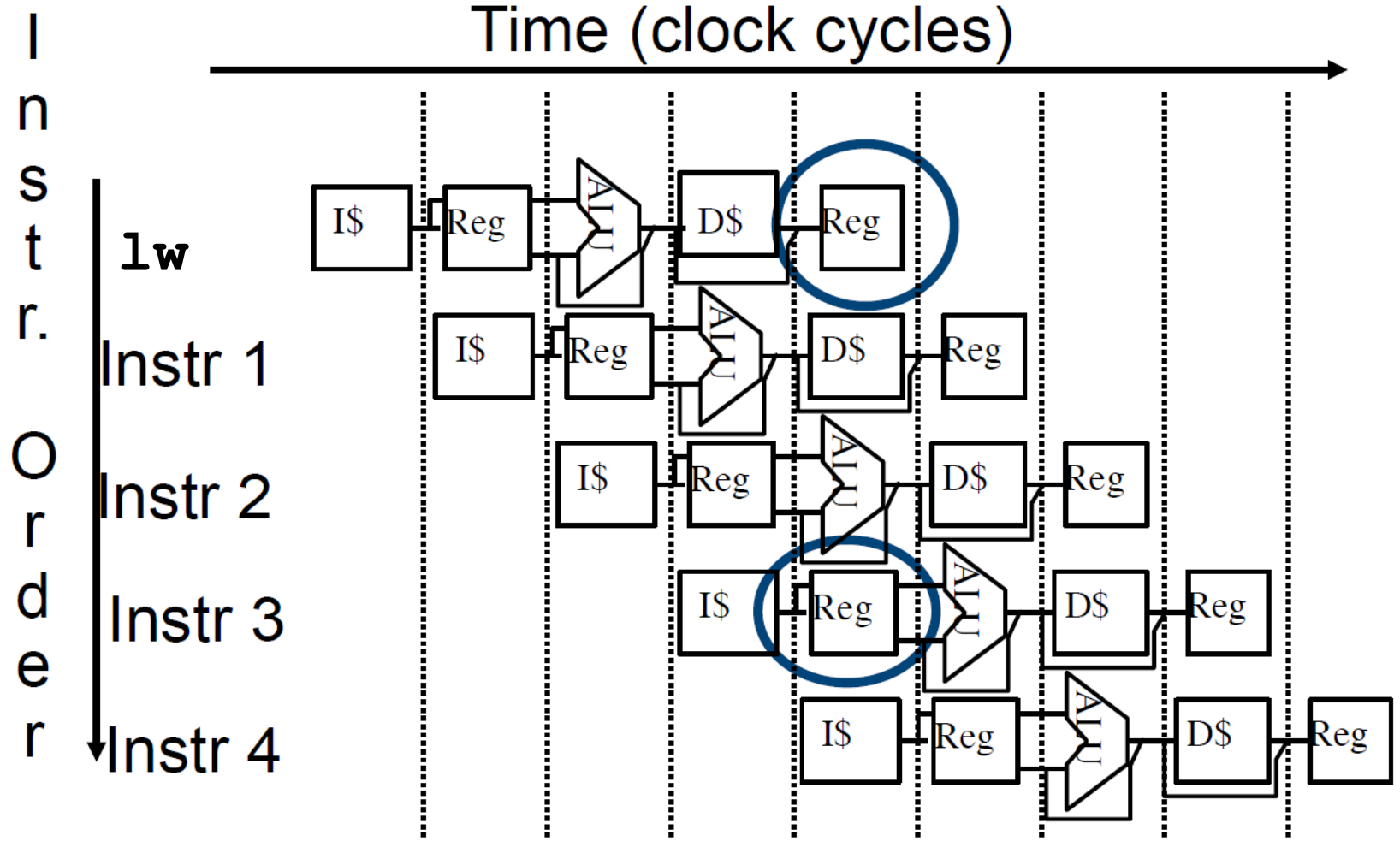


- Read same memory twice in the same cycle

Structural Hazard #1: Single Memory

- Solution:
  - Infeasible and inefficient to create second memory
    - (We'll learn more about this next week)
  - 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



- Can we read and write to registers simultaneously?

Structural Hazard #2: Registers

- Two different solutions have been used:

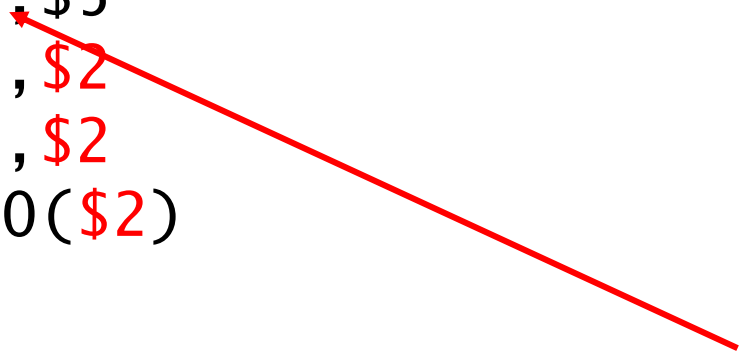
1. RegFile access is VERY fast: takes less than half the time of ALU stage
   - Write to Registers during first half of each clock cycle
   - Read from Registers during second half of each clock cycle
2. Build RegFile with independent read and write ports

- Result: can perform Read and Write during same clock cycle

Data Hazards

- Consider this sequence:

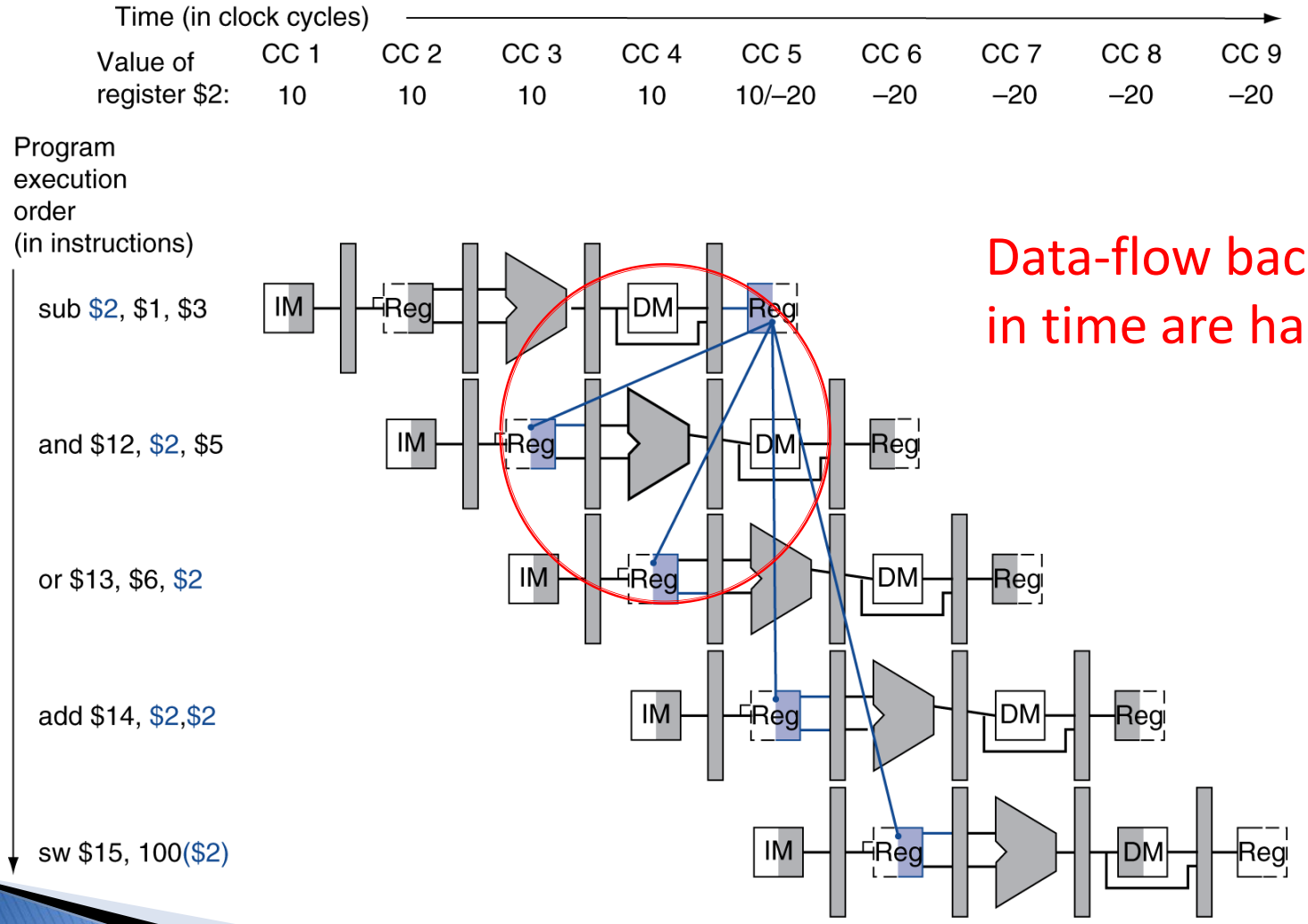
```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```



Can we read $2 during ID stage?

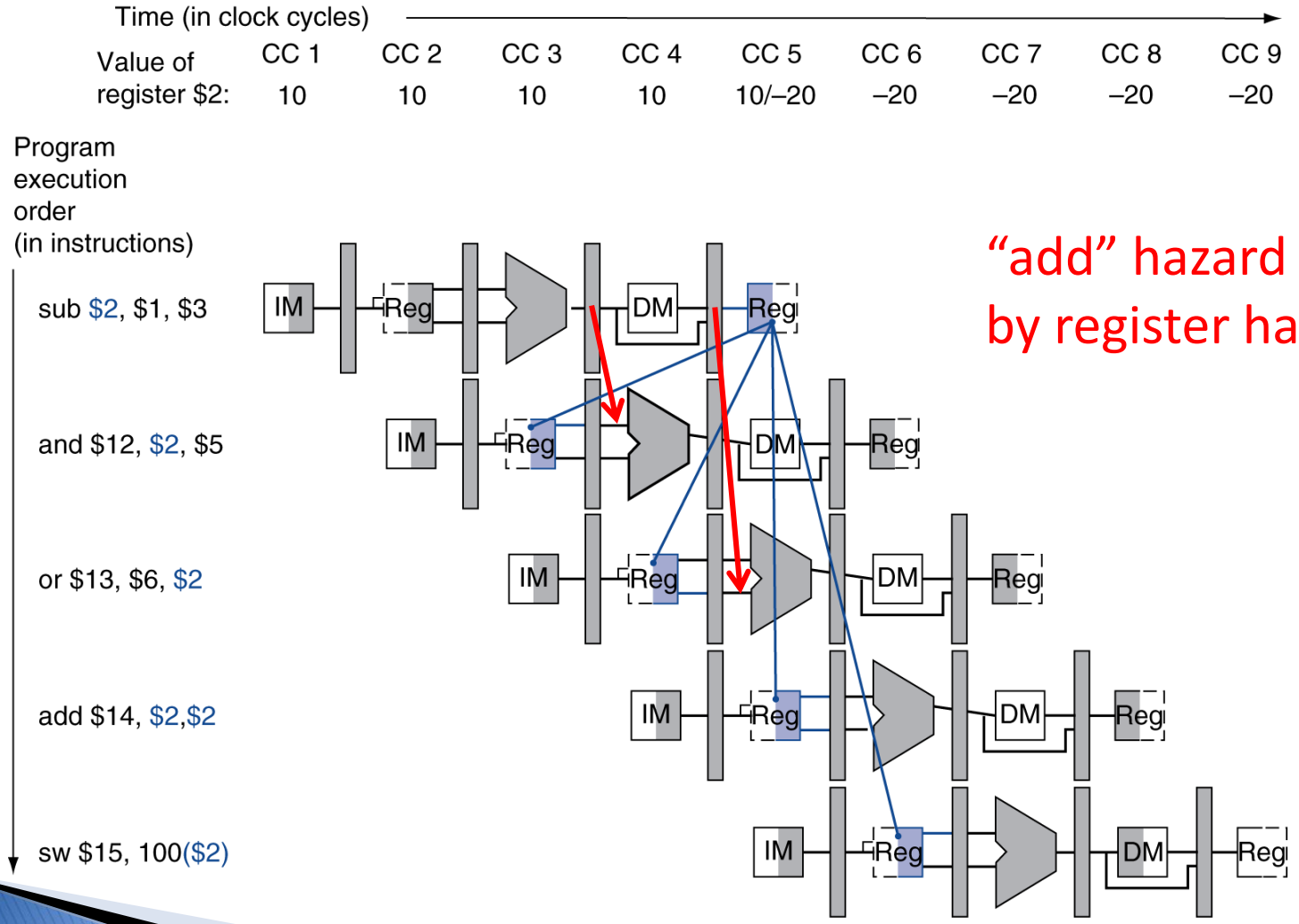
What stage is the last instr. in at this moment?

Data Hazards



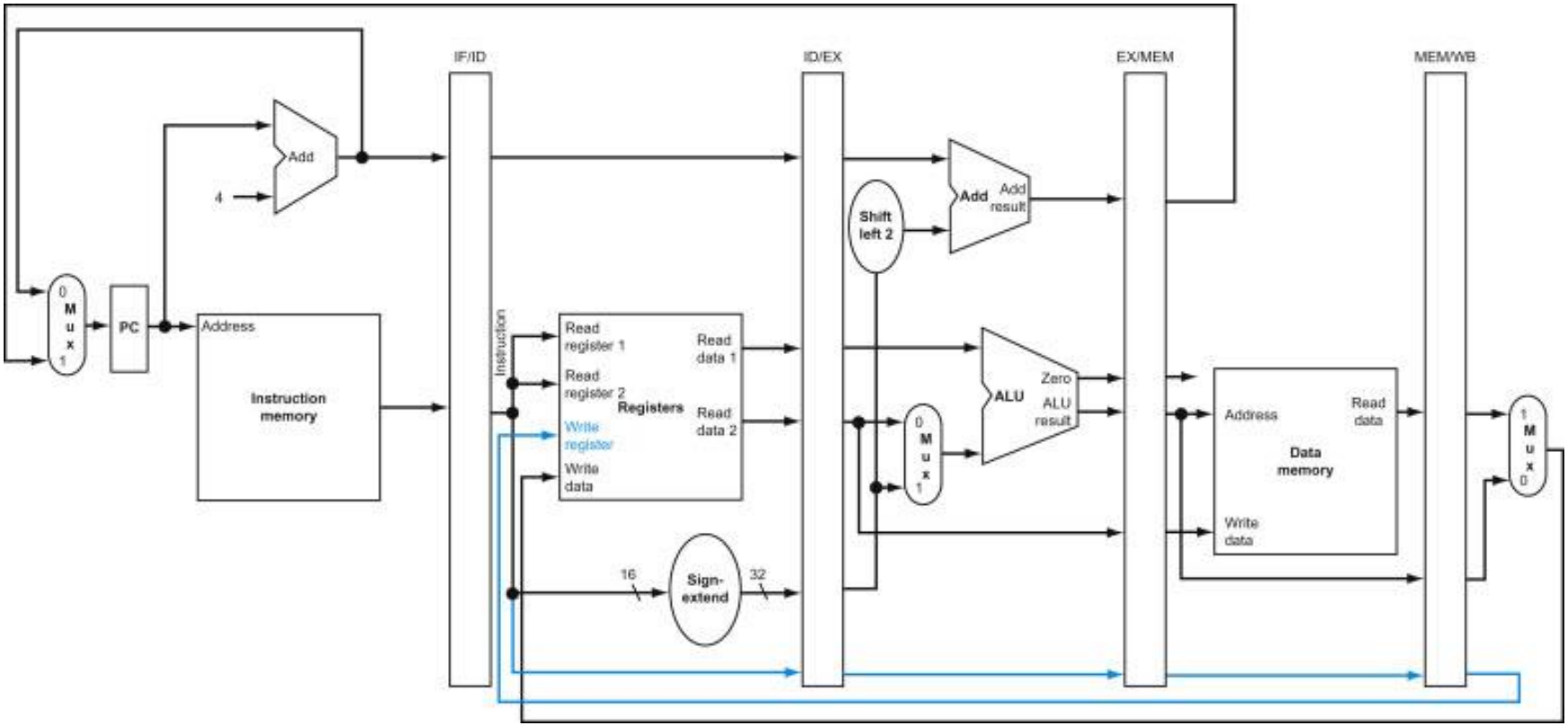
Data-flow backward in time are hazards!

Data Hazard Solution: Forwarding

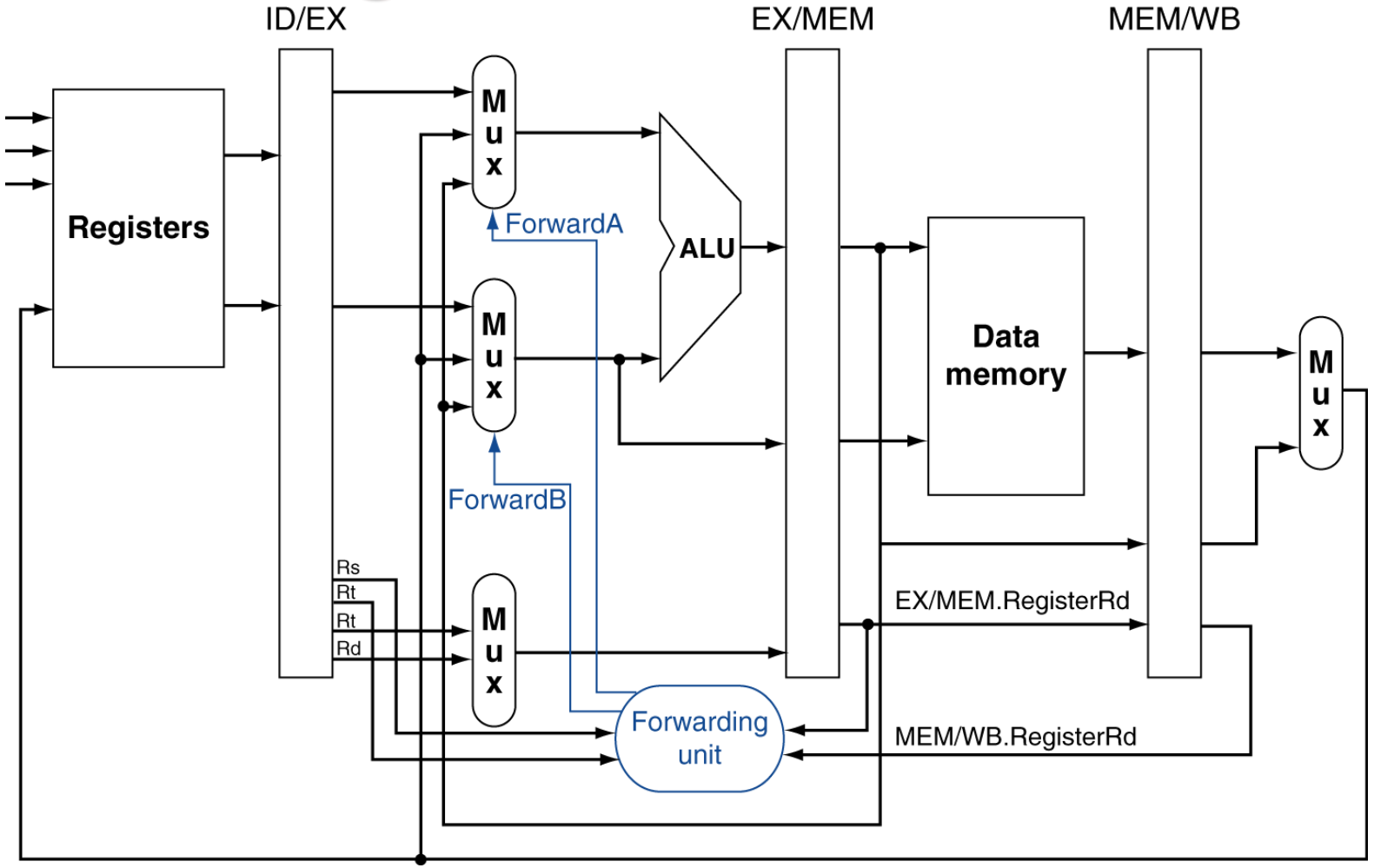


"add" hazard solved by register hardware

How do we Forward Data?

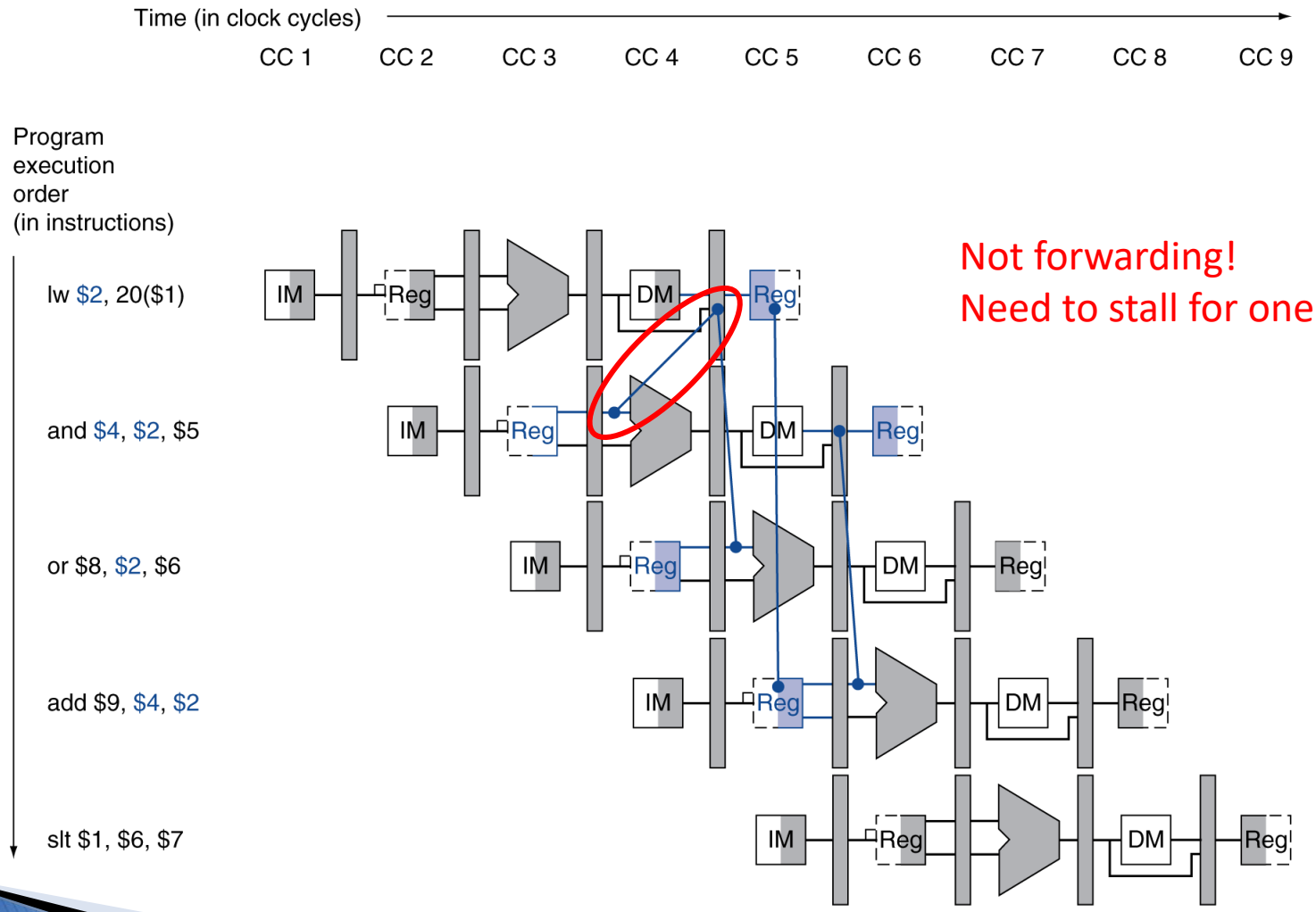


Forwarding Paths



b. With forwarding

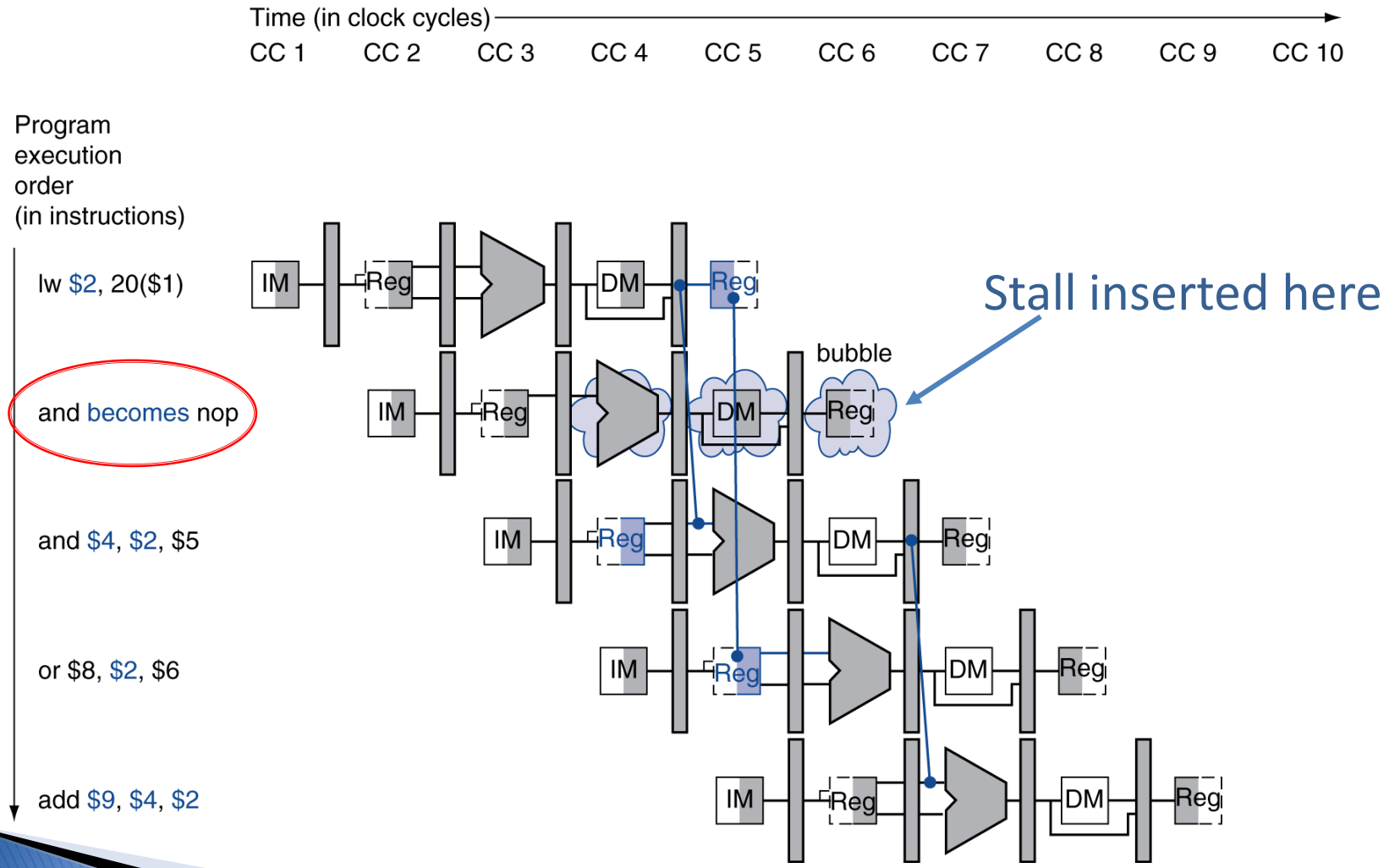
Data Hazard: Loads



Data Hazard: Loads

- Instruction slot after a load is called "load delay slot"
- If an 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)

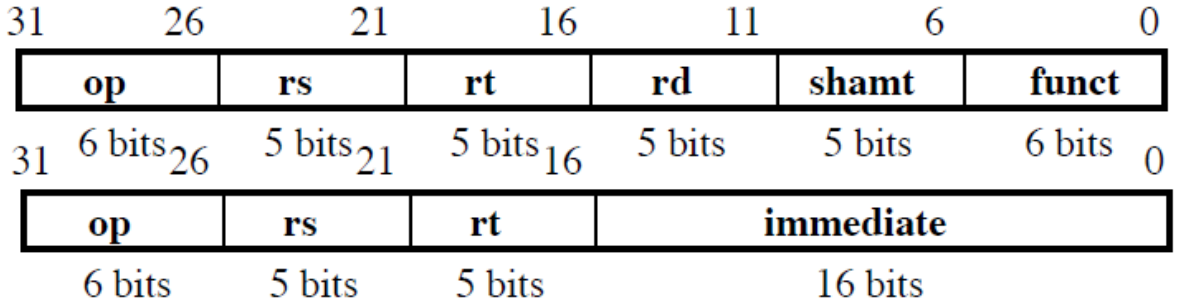
Data Hazard: Loads



Pipelining and ISA Design

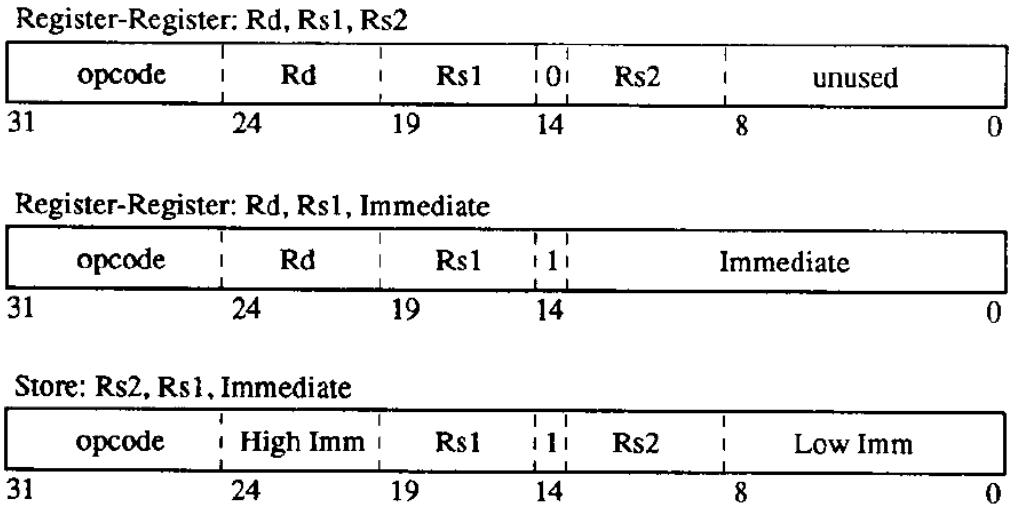
- MIPS ISA (Instruction Set Architecture) designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - x86: 1- to 17-byte instructions
      - HW translates instructions to internal RISC instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Why Isn't the Destination Register Always in the Same Field in MIPS ISA?

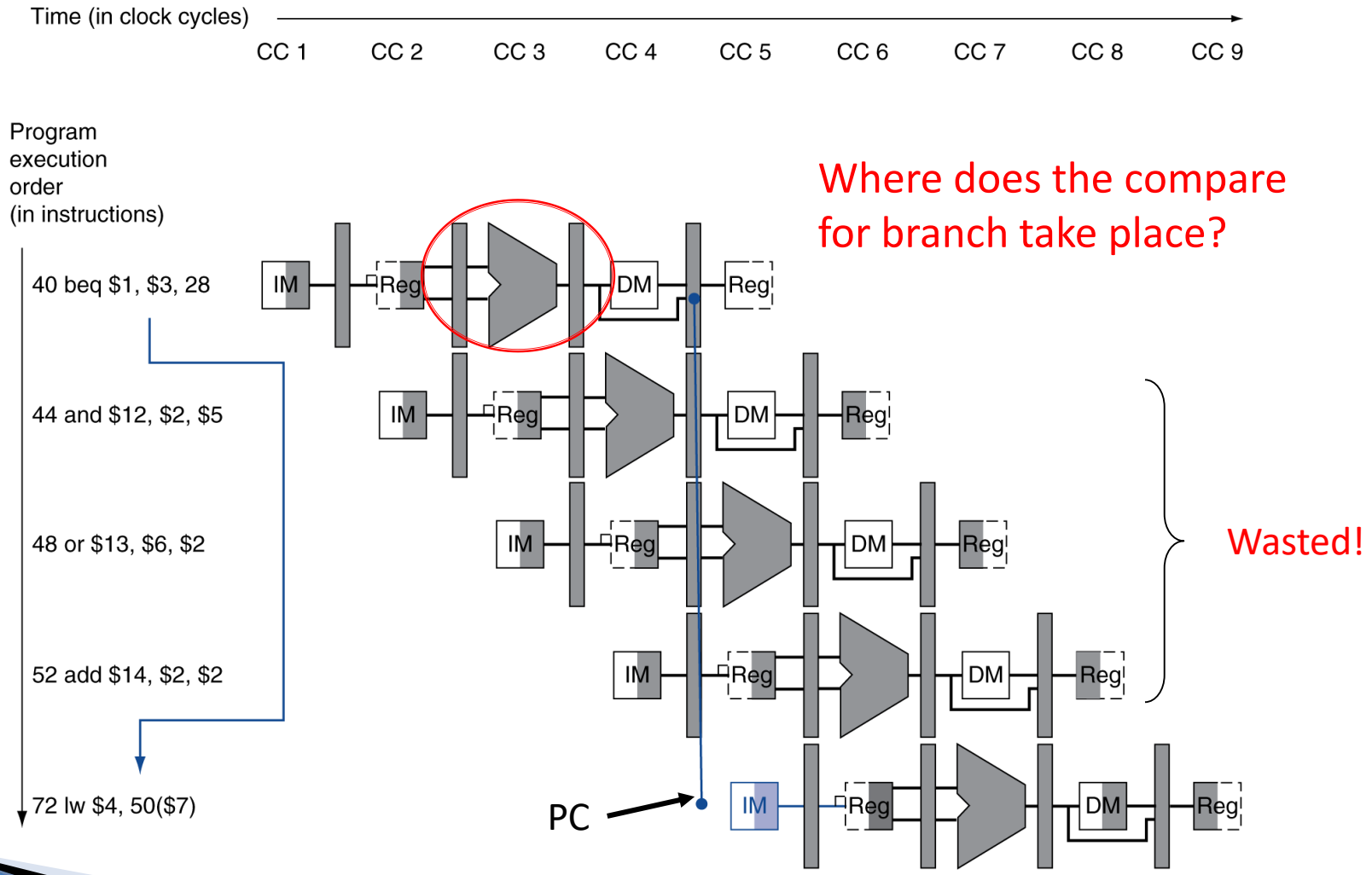


- Need to have 2 part immediate if 2 sources and 1 destination always in same place!

SPUR processor (Alternative ISA)



Control Hazard: Branching



Control Hazard: Branching

- We had 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 (ideal case)
  - 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

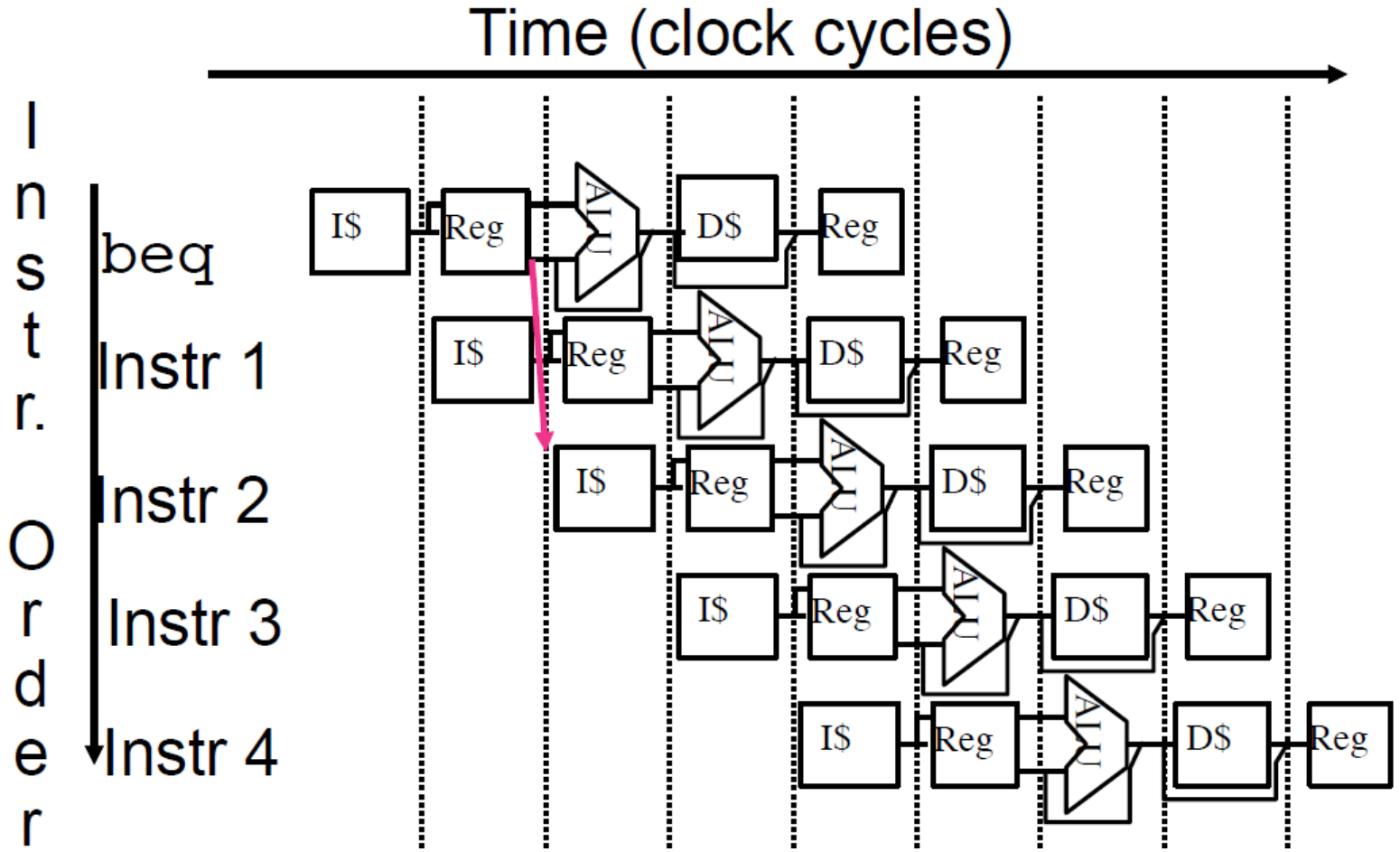
Control Hazard: Branching

- Initial Solution: Stall until decision is made
  - Insert "**nop**" instructions (those that accomplish nothing, just take time) or hold up the fetch of the next instruction (for 2 cycles).
  - Drawback: branches take 3 clock cycles each (assuming comparator is put in ALU stage)
    - How many if statements (or loops) in your programs???

Control Hazard: Branching

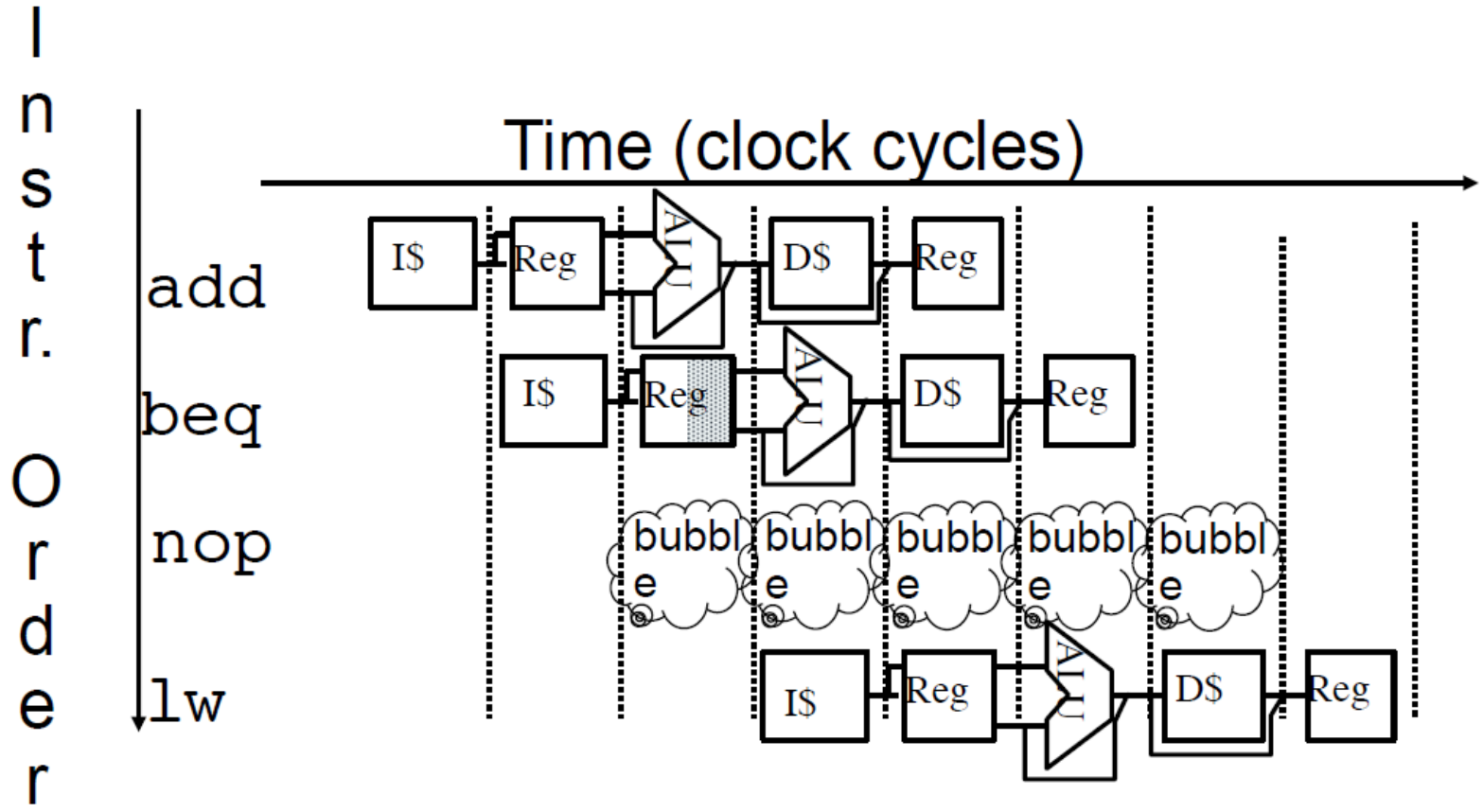
- Optimization #1:
  - Insert special branch comparator in Stage 2 (ID)
  - As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one nop is needed
  - Side Note: This means that branches are idle in Stages 3, 4 and 5.

Control Hazard: Branching



- Branch comparator moved to Decode stage

Control Hazard: Branching



- User inserting nop instruction
- Impact: 2 clock cycles per branch instruction → waste

Control Hazard: Branching

- 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)
- The term "Delayed Branch" means we always execute instruction after branch
- This optimization is used with MIPS!

Control Hazard: Branching

- Notes on Branch-Delay Slot
  - Worst-Case Scenario: can always put a nop 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
    - Video game after laundry → start video game while waiting for washer
    - Re-ordering instructions is a common method of speeding up programs (we can do this for Data Hazard from Load as well)
    - 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…

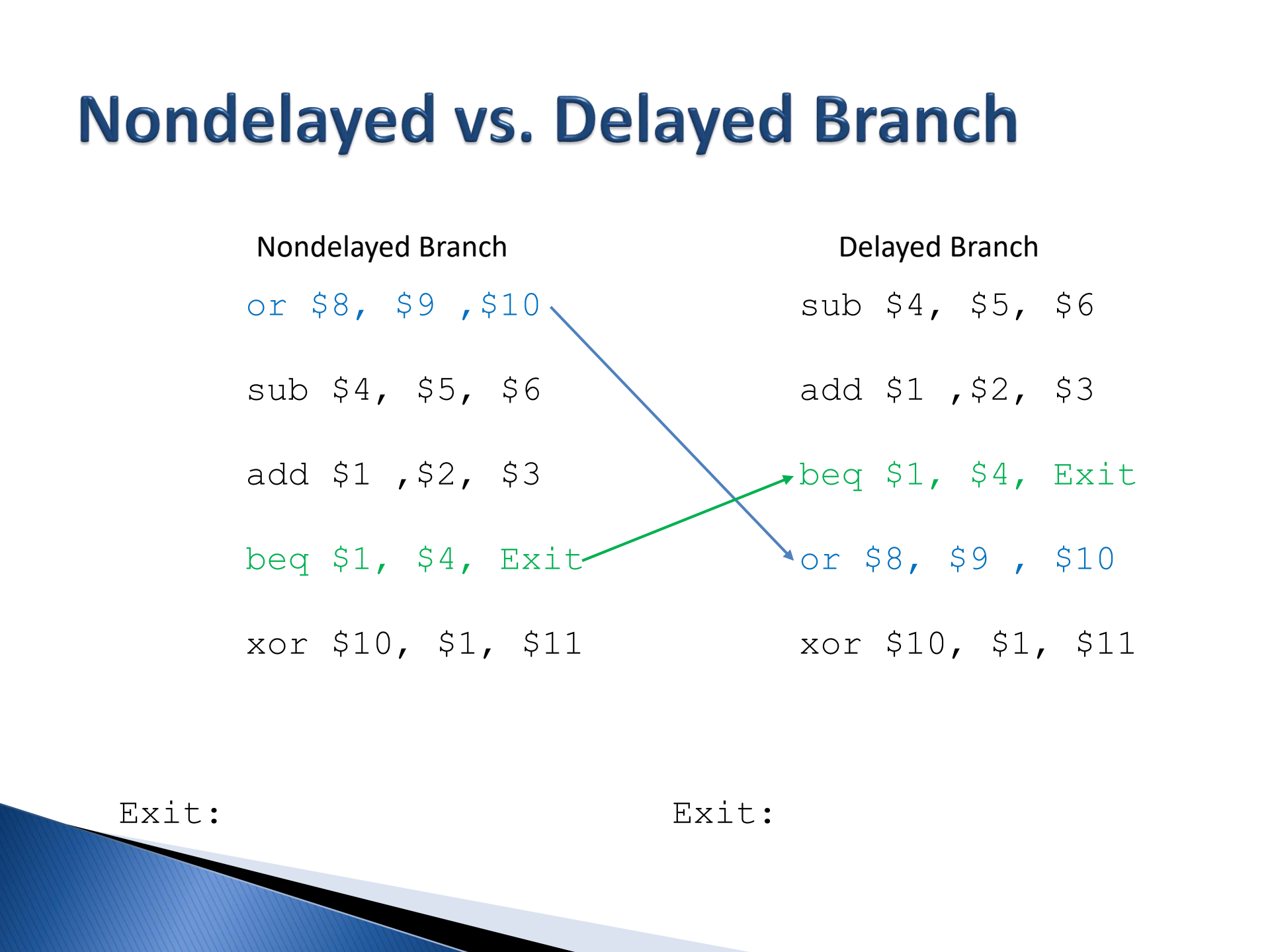
Nondelayed vs. Delayed Branch

Nondelayed Branch

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

Delayed Branch

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



Quiz

Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards (after 10^3 loops, so pipeline full)

How many pipeline stages (clock cycles) per loop iteration to execute this code?

```
Loop:   lw $t0, 0($s1)
        addu $t0, $t0, $s2
        sw $t0, 0($s1)
        addiu $s1, $s1, -4
        bne $s1, $zero, Loop
        addiu $s0, $t1, 4
```

1
2
3
4
5
6
7
8
9
10

Quiz

Assume 1 instr/clock, delayed branch, 5 stage pipeline, forwarding, interlock on unresolved load hazards (after 10^3 loops, so pipeline full)

How many pipeline stages (clock cycles) per loop iteration to execute this code?

2 (data hazard → stall)

```
Loop:   1 lw $t0, 0($s1)
        3 addu $t0, $t0, $s2
        4 sw $t0, 0($s1)
        5 addiu $s1, $s1, -4
        6 bne $s1, $zero, Loop
(delayed branch → nop) 7 addiu $s0, $t1, 4
```

1
2
3
4
5
6
7
8
9
10