

CS 31007

Autumn 2021

# COMPUTER ORGANIZATION AND ARCHITECTURE

Instructors

Rajat Subhra Chakraborty (*RSC*)

Bhargab B. Bhattacharya (*BBB*)

Lecture #42

**Processor Design: Pipelining (Contd..)**

08 November 2021

Indian Institute of Technology Kharagpur  
*Computer Science and Engineering*

# Pipeline Hazards

- Situations that prevent executing successor instructions in their intended cycles; slows down the pipeline!
- Structural hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Choice of next instruction depends on previous decision, which is awaited

# Example: Data Hazards in Pipeline

CC	1	2	3	4	5	6	7	8
ADD R1, R2, R3	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

data hazard

With forwarding mechanism such as EX  $\rightarrow$  EX and Mem  $\rightarrow$  EX all data hazards in this cases can be eliminated.

Total # clock cycles to run this code = 8

# Double Data Hazard

- Consider the sequence:  
    add \$1, \$1, \$2  
    add \$1, \$1, \$3  
    add \$1, \$1, \$4
- HMK: How to handle such hazards?

# Example: Load-Use Data Hazards in Pipeline

	CC	1	2	3	4	5	6	7	8
LW	R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB	R4, R1, R5		IF	ID	EX	MEM	WB		
AND	R6, R1, R7			IF	ID	EX	MEM	WB	
OR	R8, R1, R9				IF	ID	EX	MEM	WB

data hazard

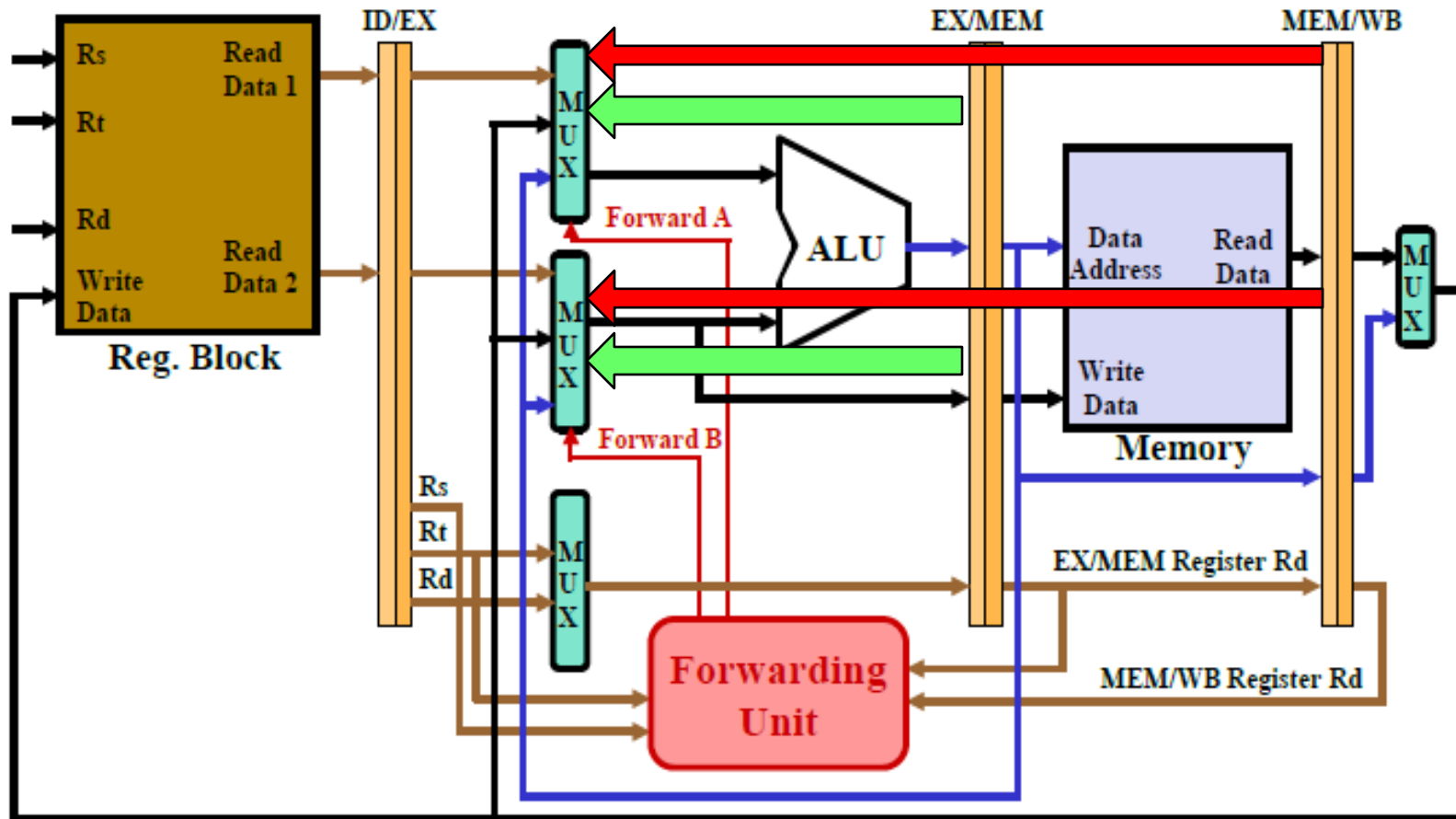
# stall-cycle = 1

	CC	1	2	3	4	5	6	7	8	9
LW	R1, 0(R2)	IF	ID	EX	MEM	WB				
SUB	R4, R1, R5		IF	ID	stall	EX	MEM	WB		
AND	R6, R1, R7			IF	stall	ID	EX	MEM	WB	
OR	R8, R1, R9				stall	IF	ID	EX	MEM	WB

Total # clock cycles to run this code = 9

Stall is inevitable when the *destination* of load is a *source* of the next instruction

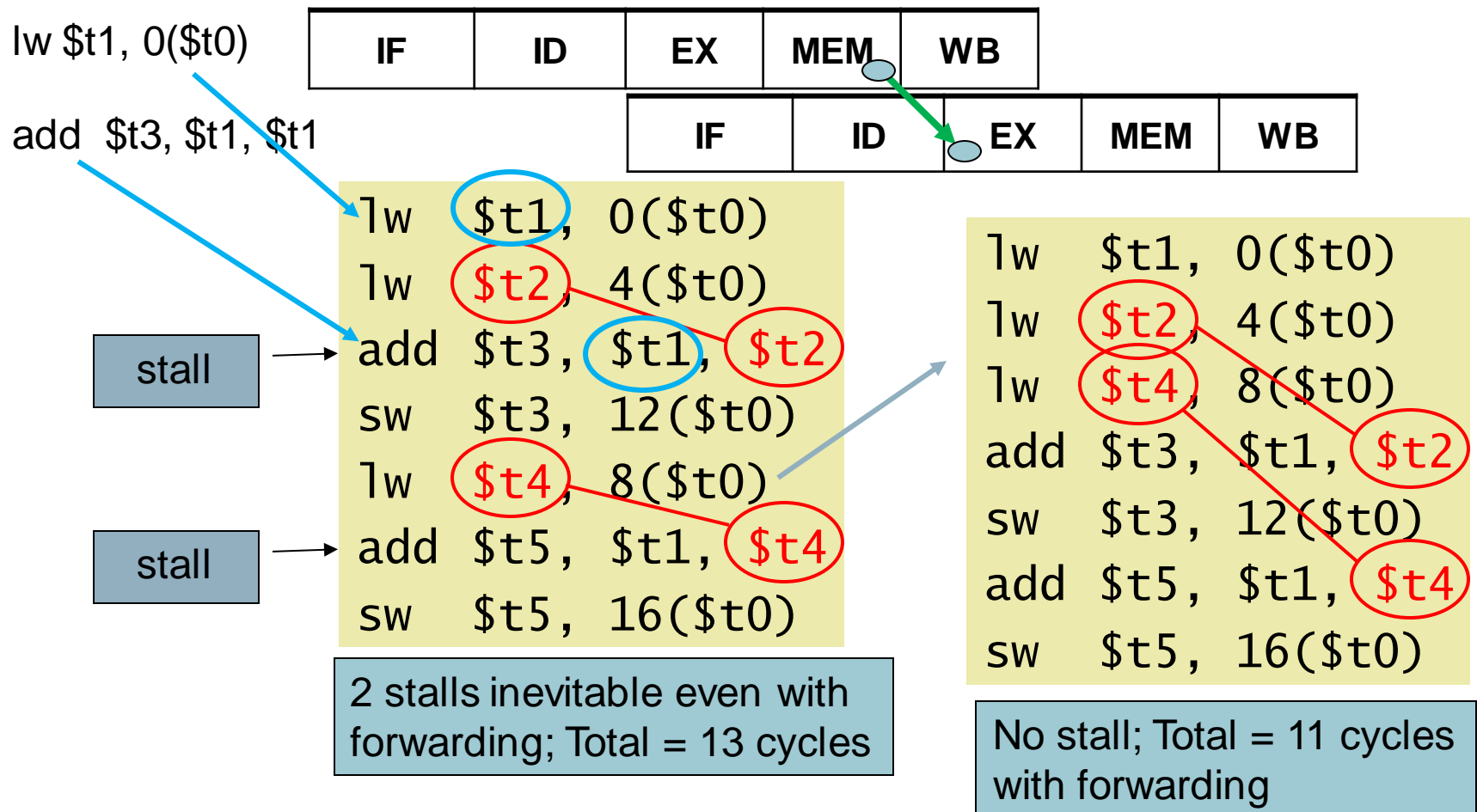
# Cons: Data Forwarding Needs Extra Hardware (Mem→Ex) in addition to Ex → EX



# Data hazards – Can we do something more?

- C code for  $A = B + E$ ;  $C = B + F$ ;

Reorder code to avoid use of load result in the next instruction



# RAW Data Hazards

## Read-After-Write (RAW)

$\text{Instr}_j$  tries to read operand before  $\text{Instr}_i$  writes it



I: add \$1, \$2, \$3  
J: sub \$4, \$1, \$3

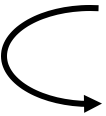
❑ Caused by “dependence”



# WAW Data Hazards

## Write-After-Write (WAW)

Because of hazard, Instr<sub>j</sub> tries to write operand *before* it is written by Instr<sub>i</sub>

 I: lw \$1, 0(\$2)      data cache miss  
J: sub \$1, \$2, \$3

I: IF   ID/RR   EX   MEM    $\phi$     $\phi$    WB

J:       IF       ID/RR   EX   MEM   WB

Write is performed in wrong order!

# WAR Data Hazards

## Write-After-Read (WAR)

 I: add \$4, \$1, \$3  
J: sub \$1, \$2, \$3

Can J write to a register earlier before I reads it?

In the MIPS pipeline scheme, reads are early (ID/RR) and writes are late (WB); hence, WAR cannot occur.

Read-After-Read (RAR): Not a hazard

# Summary: Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!
  - Need stalls so that forwarding is possible

CS 31007

Autumn 2021

# COMPUTER ORGANIZATION AND ARCHITECTURE

Instructors

Rajat Subhra Chakraborty (*RSC*)

Bhargab B. Bhattacharya (*BBB*)

Lecture #43, #44

**Processor Design: Pipelining (Contd..)**

09 November 2021

Indian Institute of Technology Kharagpur  
*Computer Science and Engineering*

# Control Hazards

A control hazard arises when the computation of the destination of a branch and update of PC are awaited, and hence, the instruction to be executed next, is uncertain at this moment

```
beq $t1, $t2, Target
```

```
lw  $t3, 0 ($t4)
```

```
...
```

```
...
```

```
Target: add $t1, $t2, $t3
```

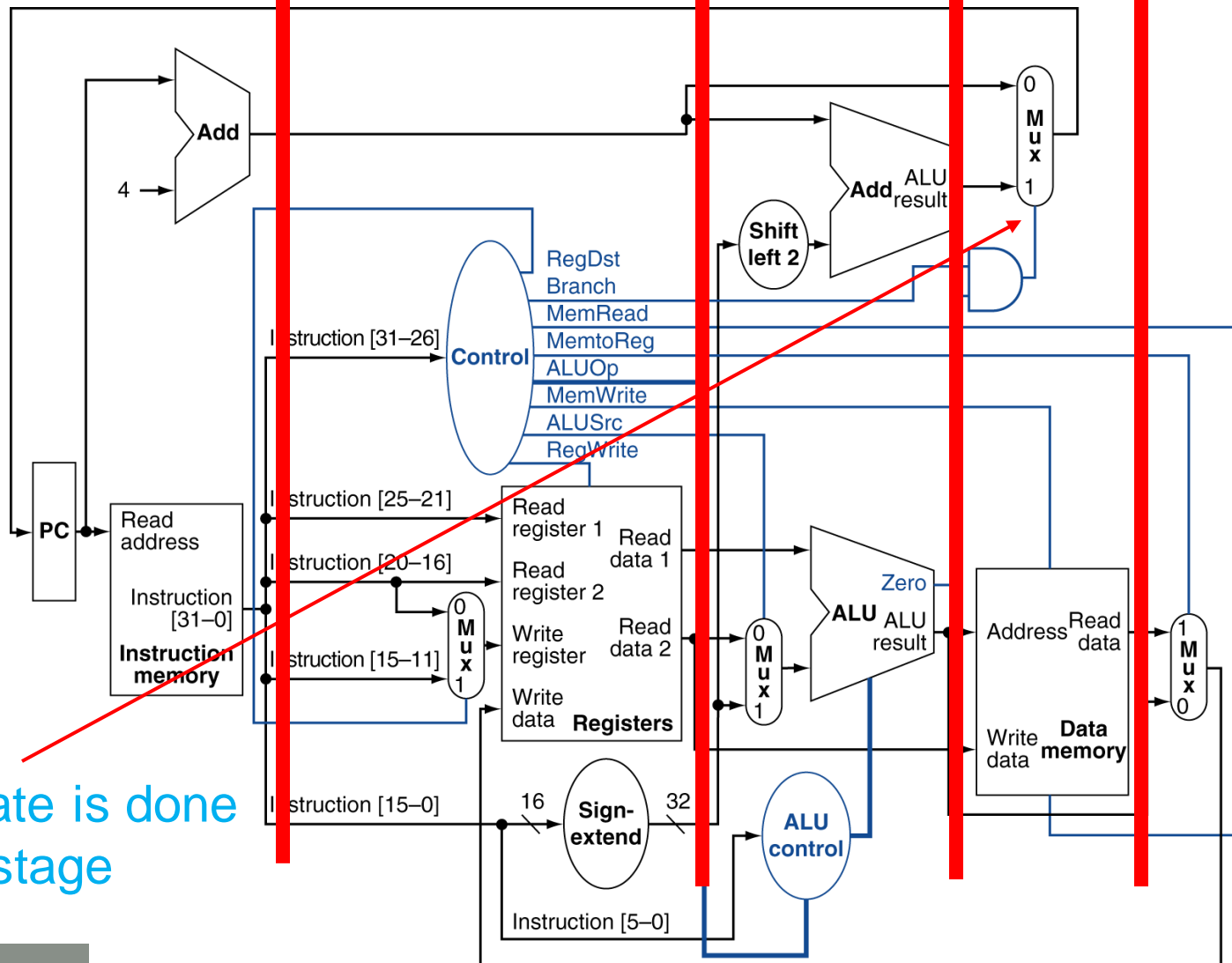
Instruction  
Fetch

Instr. Decode  
Reg. Fetch

Execute  
Addr. Calc

Memory  
Access

Write  
Back



PC-Update is done  
in Mem-stage

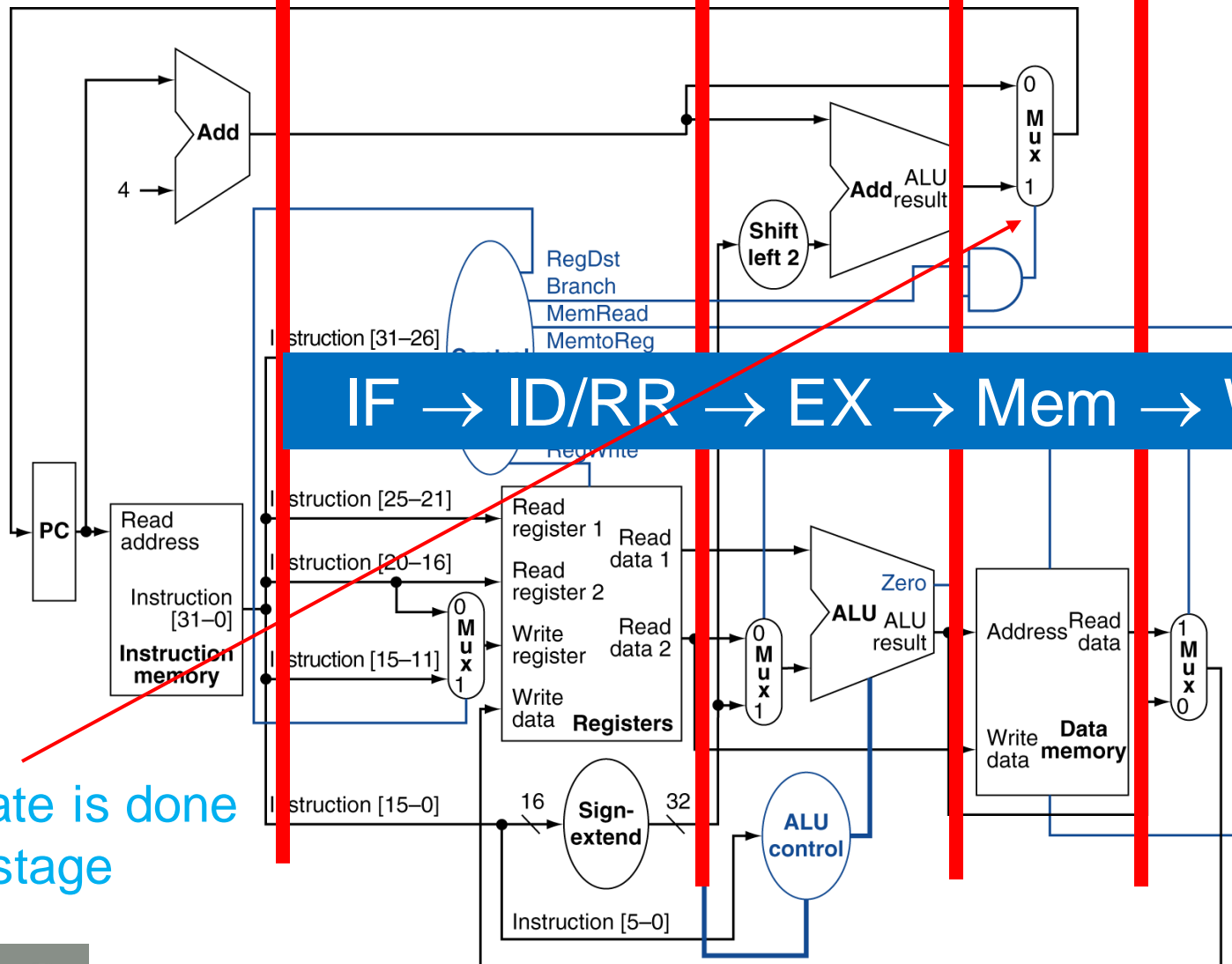
Instruction  
Fetch

Instr. Decode  
Reg. Fetch

Execute  
Addr. Calc

Memory  
Access

Write  
Back

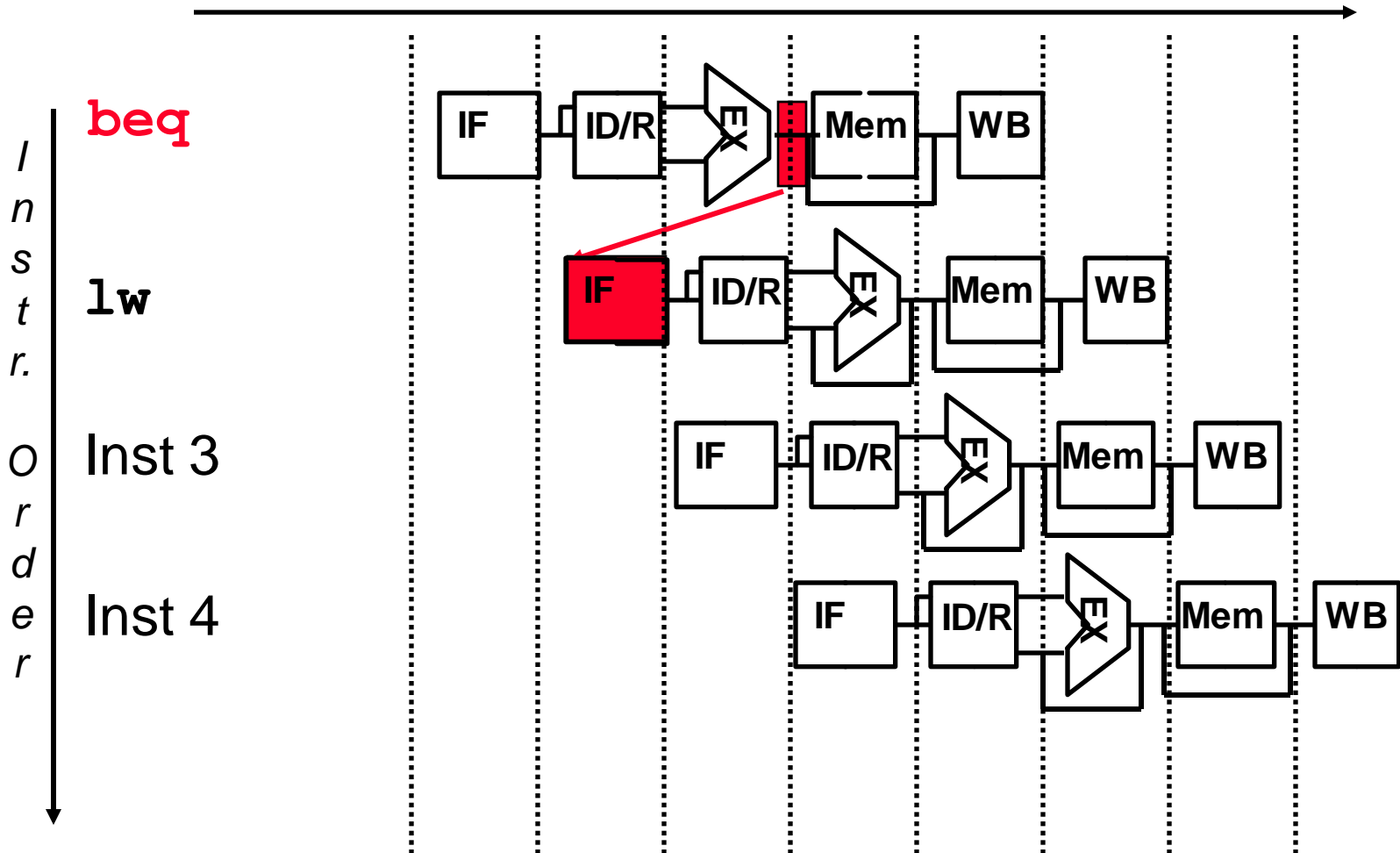


IF → ID/RR → EX → Mem → WB

PC-Update is done  
in Mem-stage

# Example: Control Hazards due to Branch

- Dependencies backward in time cause hazards





# Branch Penalty

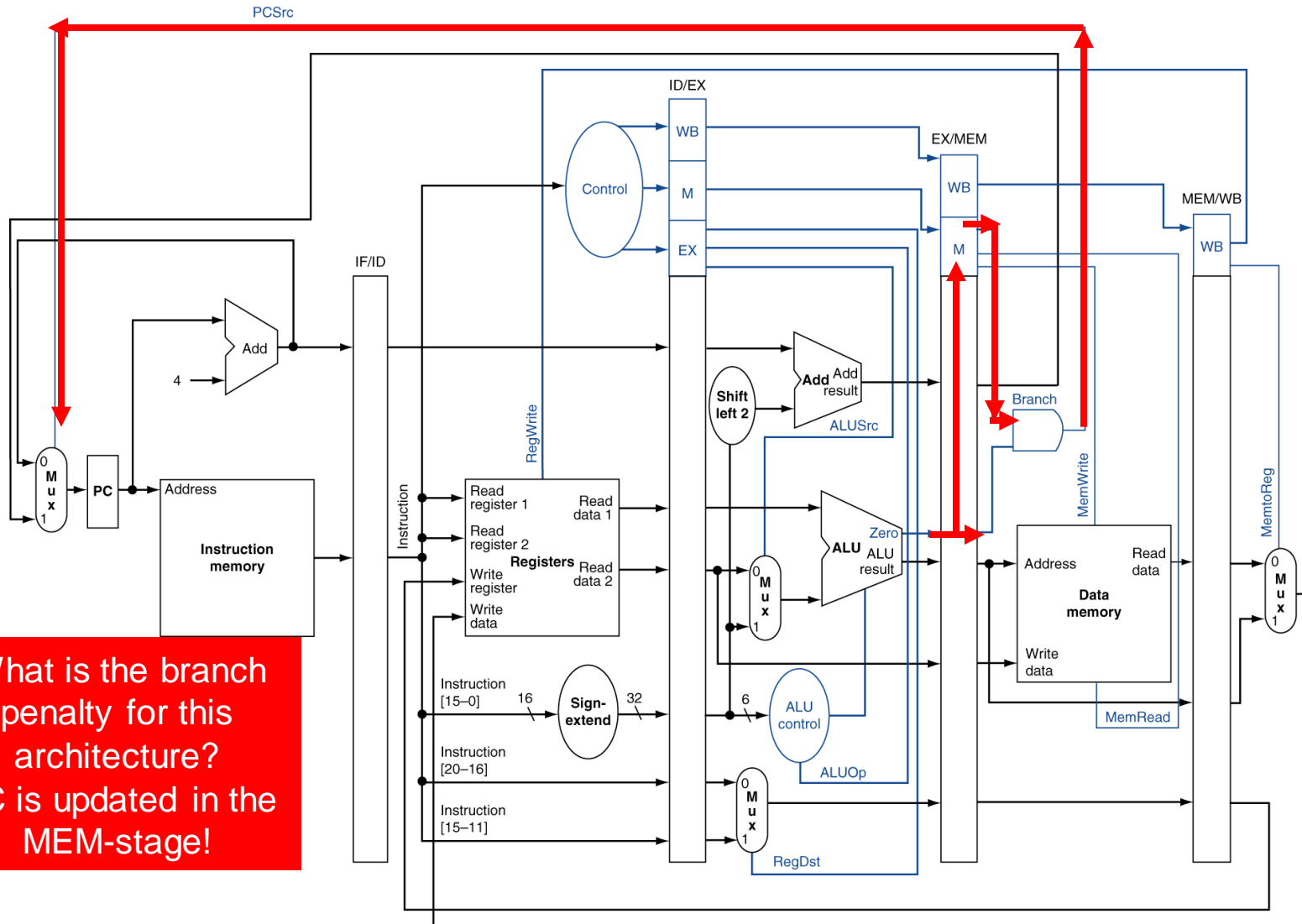
beq

IF	ID	EX	MEM	WB
----	----	----	-----	----

lw

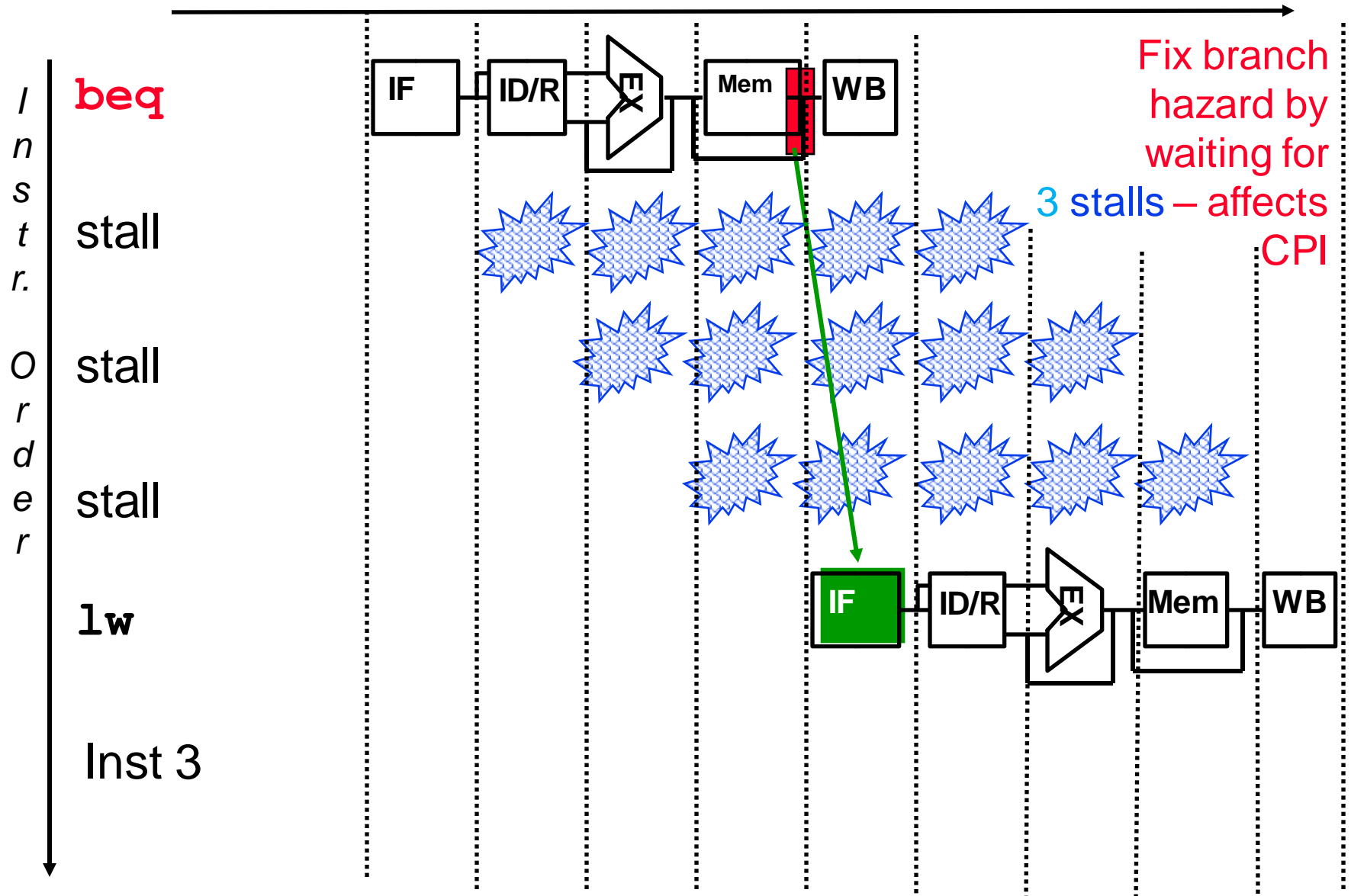
IF	ID	EX	MEM	WB
----	----	----	-----	----

branch penalty: 3 clock cycles



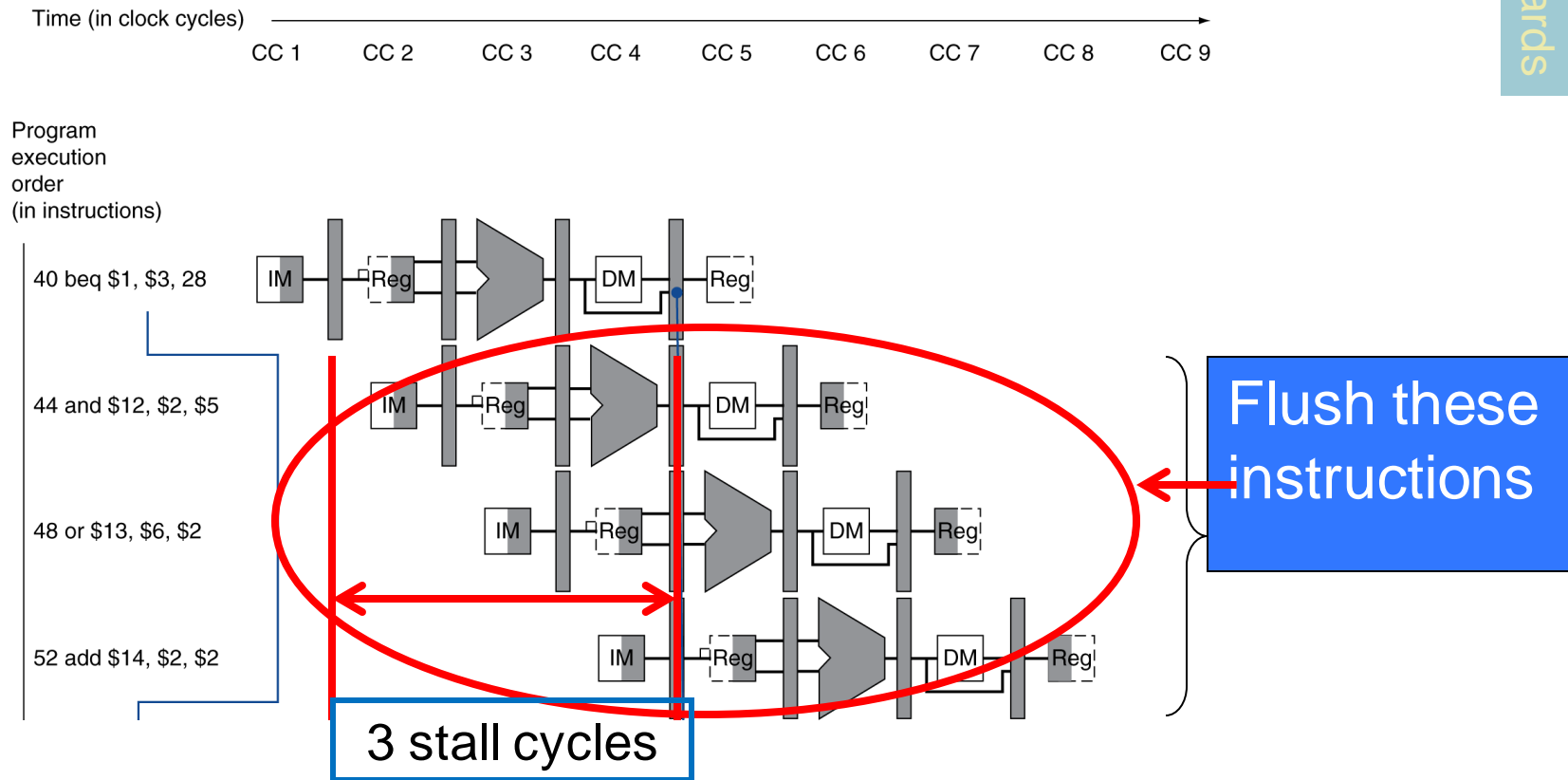
What is the branch penalty for this architecture?  
PC is updated in the MEM-stage!

# One Way to “Fix” a Control Hazard: Insert stalls



# Control Hazards: Continue, flush if branch is taken

- If branch outcome determined in MEM



# Control Hazards

- Hardware solution?

Can we compute branch target and PC-Update, earlier in the pipeline?

Would reduce stalls for successor instructions!

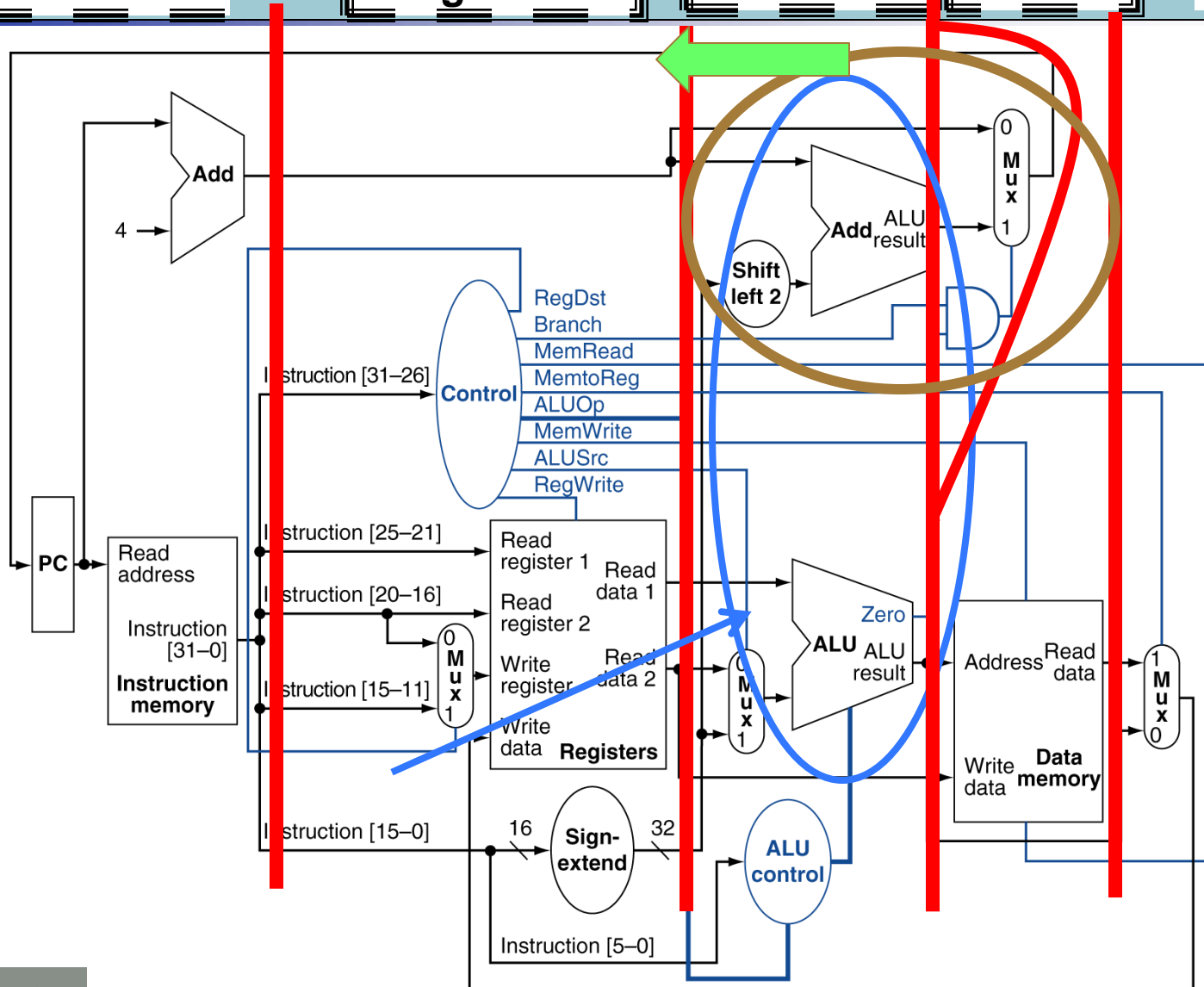
Instruction  
Fetch

Instr. Decode  
Reg. Fetch

Execute  
Addr. Calc

Memory  
Access

Write  
Back



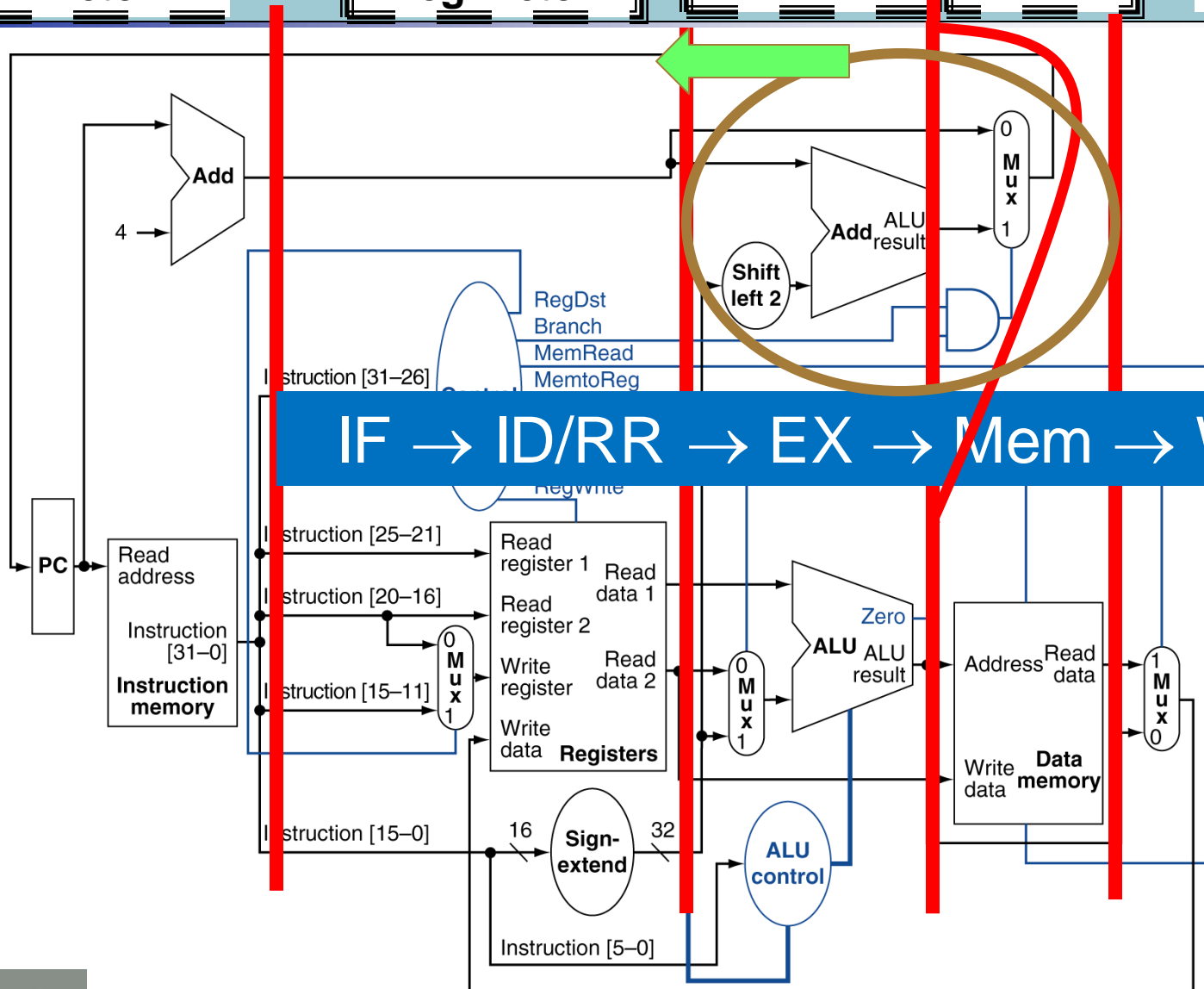
Instruction  
Fetch

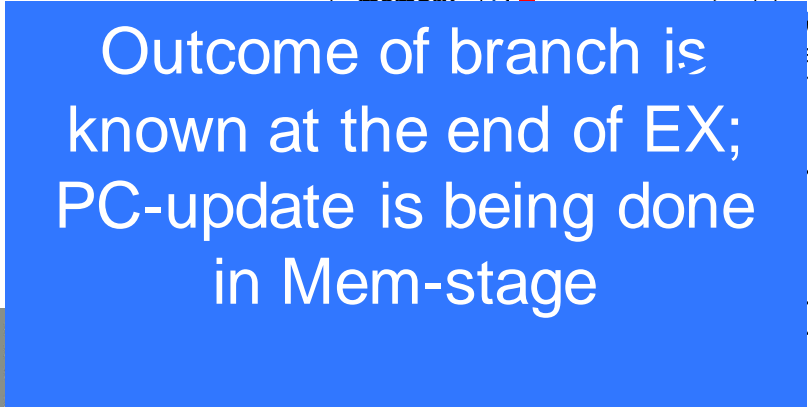
Instr. Decode  
Reg. Fetch

Execute  
Addr. Calc

Memory  
Access

Write  
Back





Instruction  
Fetch

Instr. Decode  
Reg. Fetch

Execute  
Addr. Calc

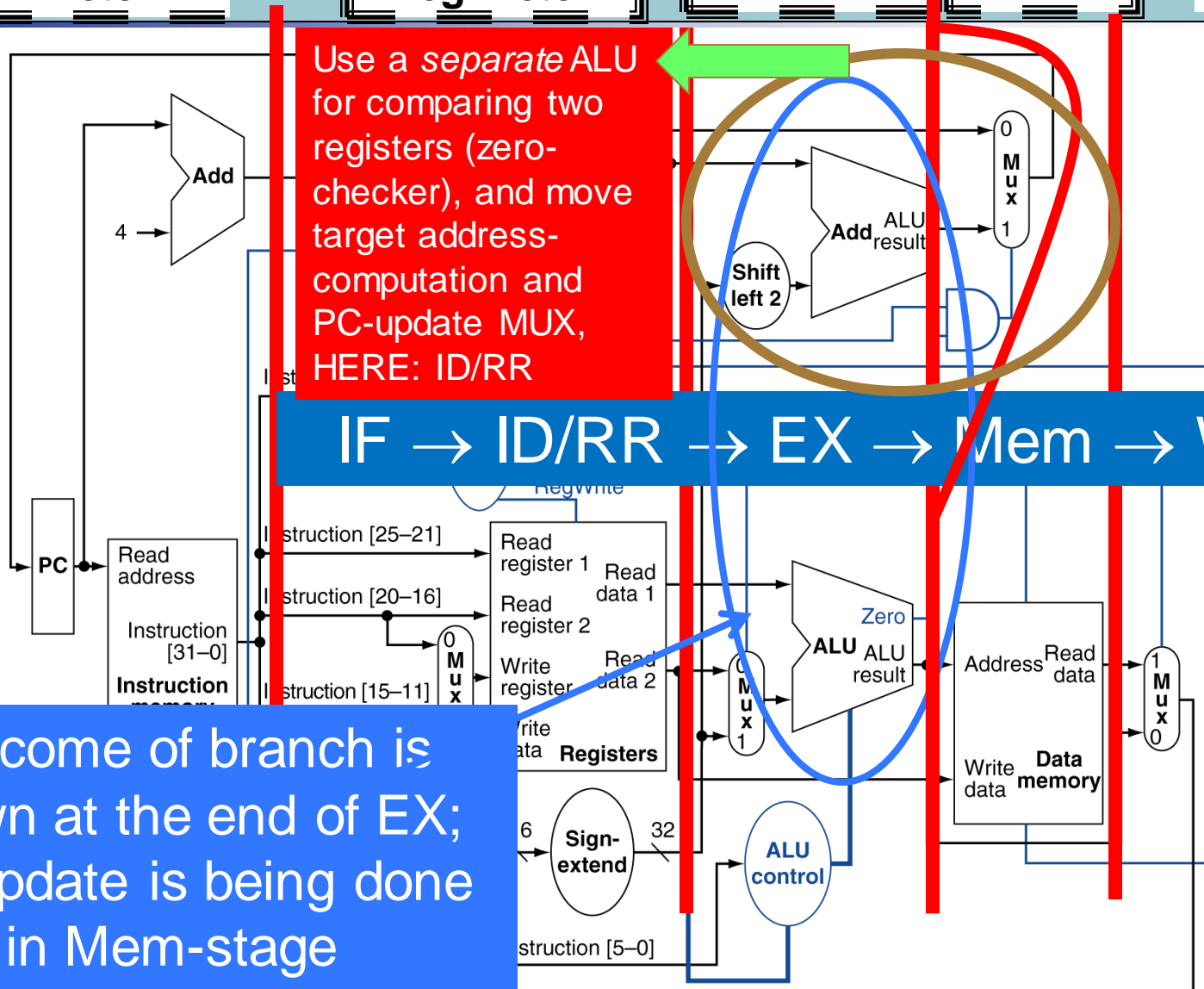
Memory  
Access

Write  
Back

Use a *separate* ALU for comparing two registers (zero-checker), and move target address-computation and PC-update MUX, HERE: ID/RR

IF → ID/RR → EX → Mem → WB

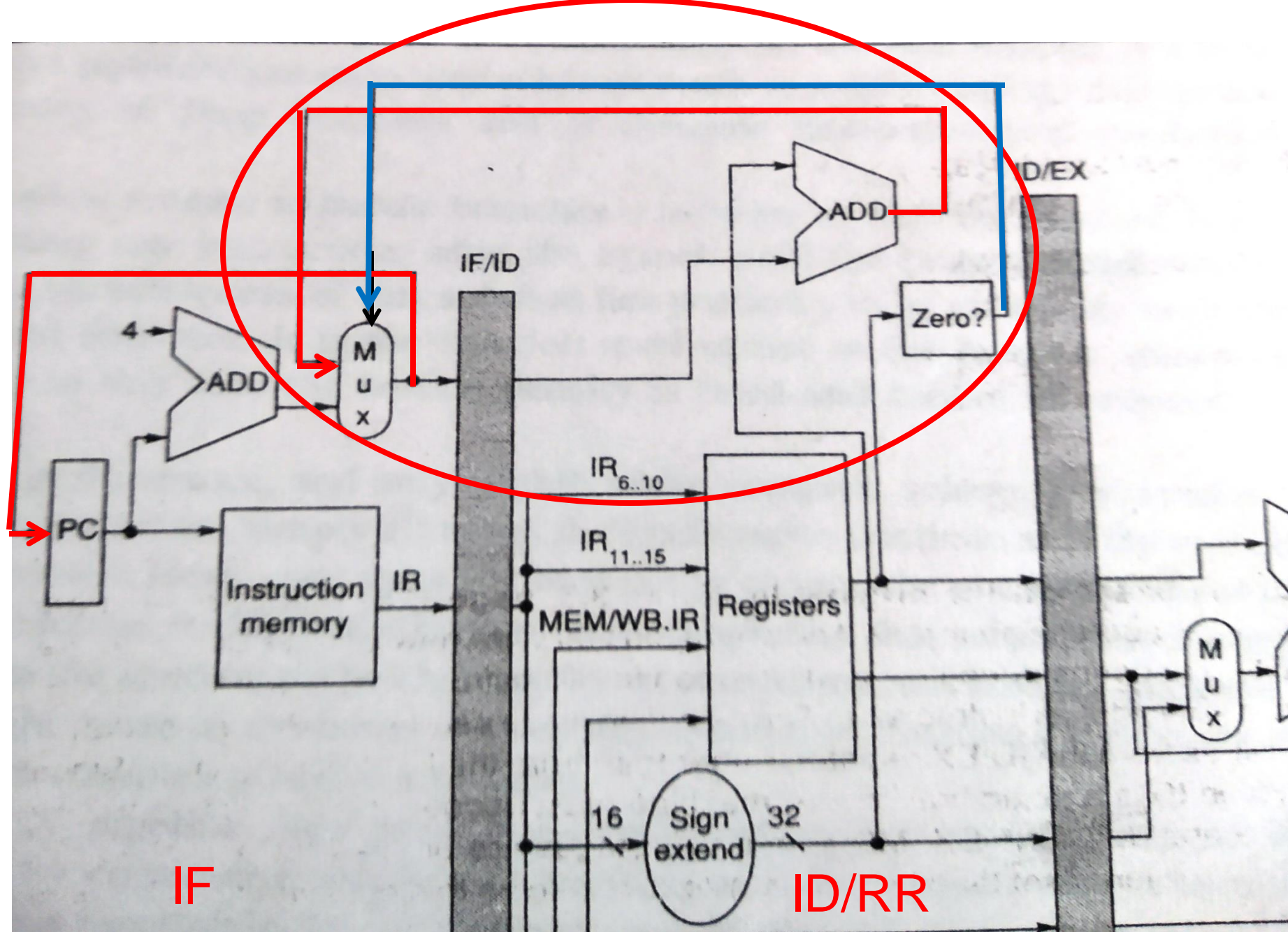
Outcome of branch is known at the end of EX; PC-update is being done in Mem-stage





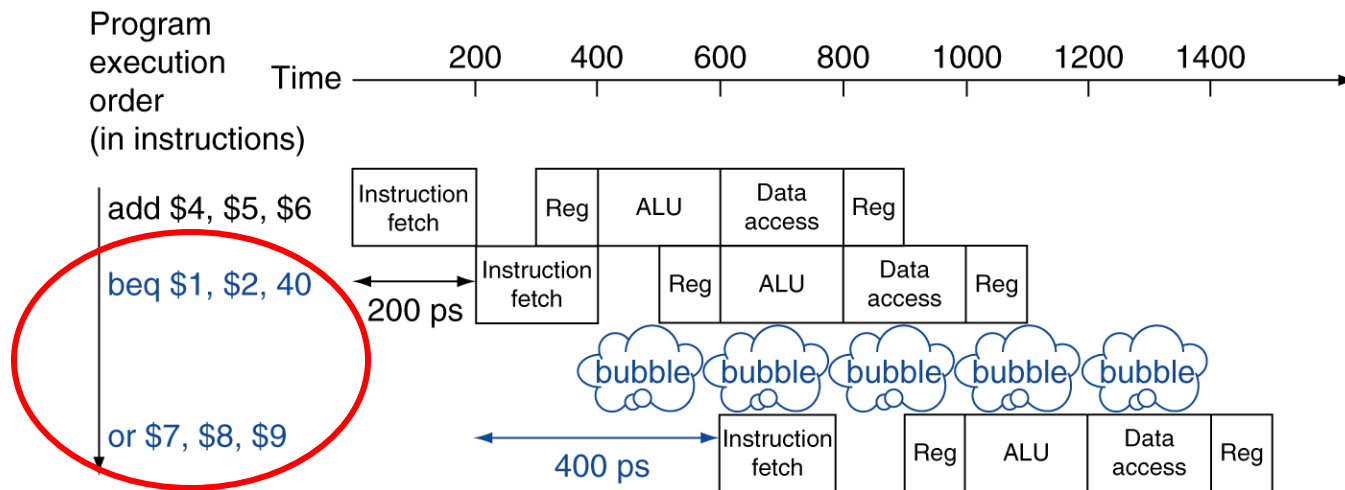
# Control Hazards (*beq*)

- In MIPS pipeline, we can reduce branch penalty
  - by comparing registers (zero-checking) and computing target (PC-update) early in the pipeline
  - Add hardware to do it **earlier** in **ID/RR-stage**



# Stall on Branch

- Wait until branch outcome determined (at ID/RR-stage) before fetching next instruction



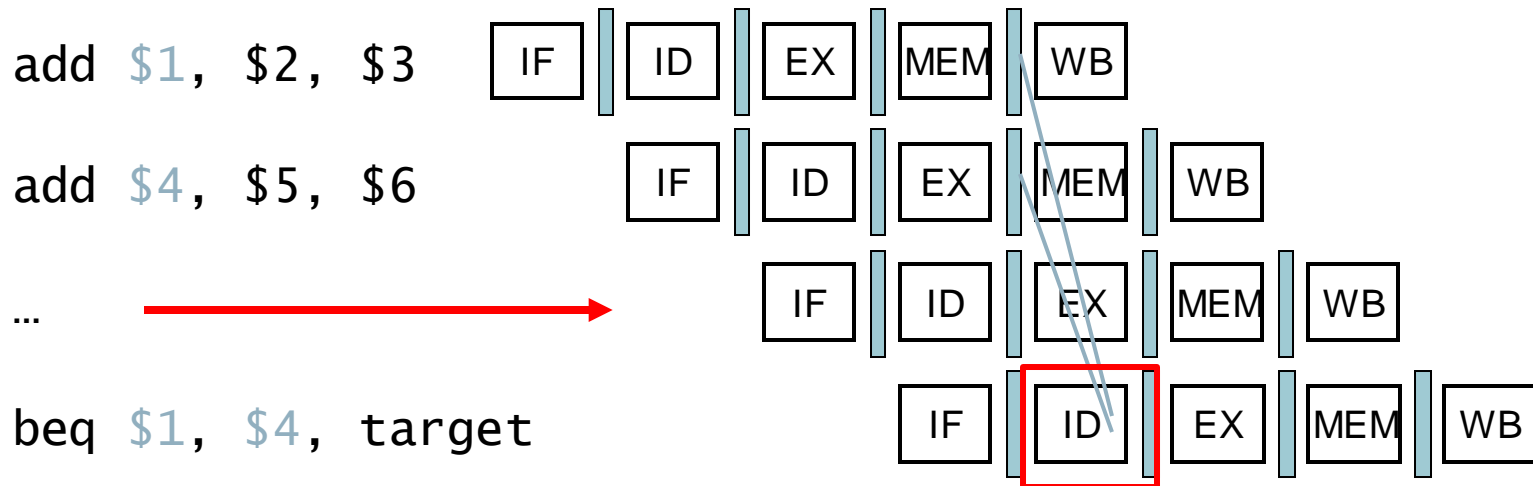
Instead of “three stalls”, we now have only “one stall”

# Stall on Branch

- Moving branch-target determination from Mem to ID/RR-stage helps resolve hazards in successor instructions
- However,
- it may pose problems to *preceding* instructions, when a branch source-register is a destination register of a preceding instruction!

# Data Hazards for Branches

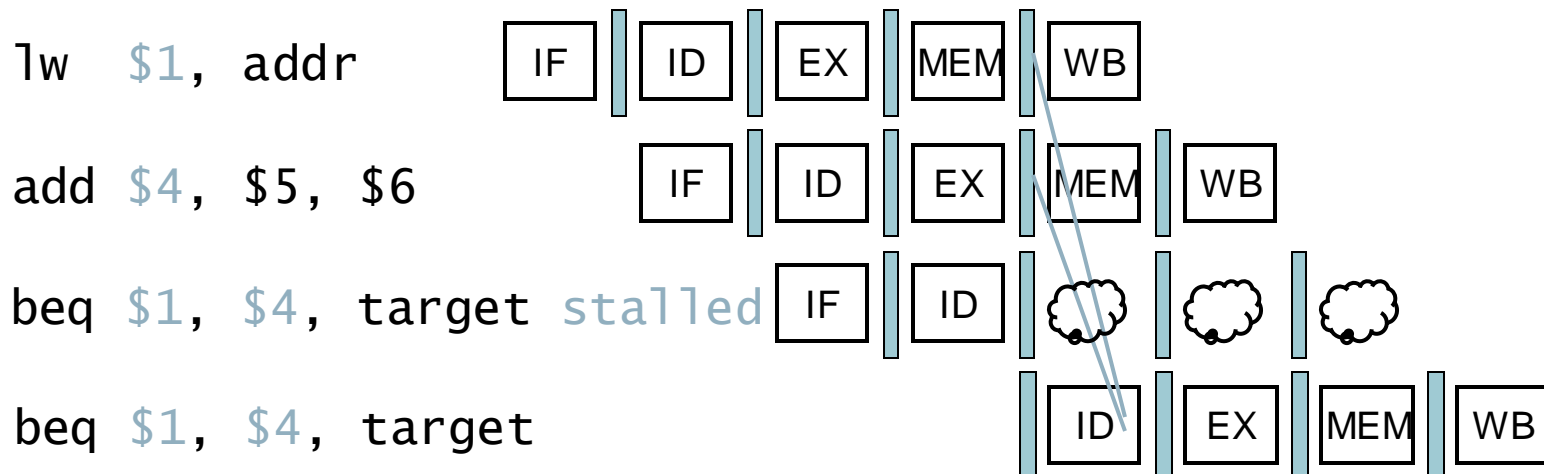
- If a comparison register is a destination of 2<sup>nd</sup> or 3<sup>rd</sup> preceding ALU instruction



- Can resolve using forwarding

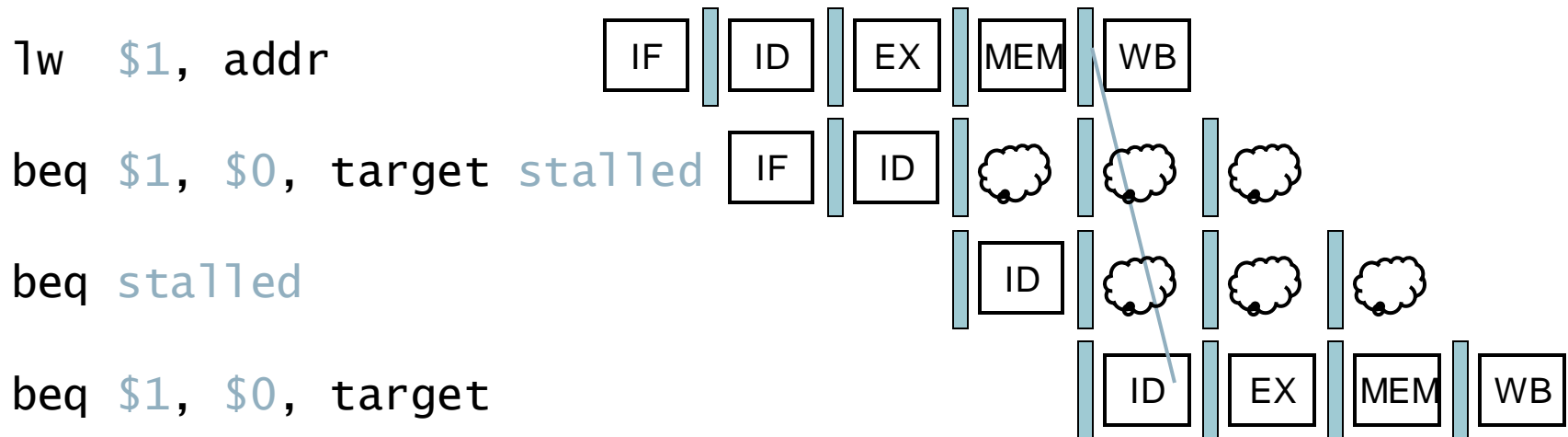
# Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2<sup>nd</sup> preceding load instruction
  - Need 1 stall cycle



# Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles



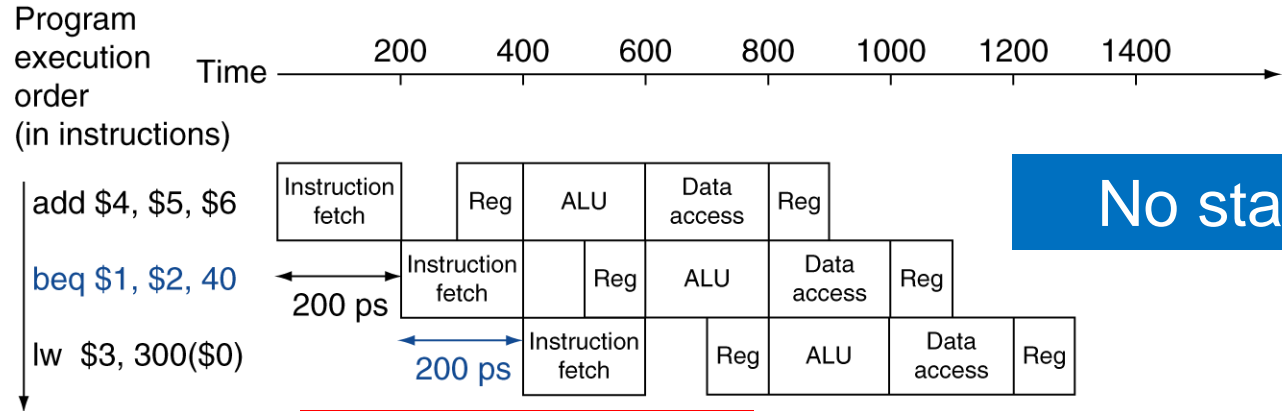
# Another Approach: Branch Prediction

- Longer pipelines cannot readily determine branch outcome early
  - Fixed stall-penalty becomes unacceptable
- Predict outcome of branch
  - Stall only if prediction is wrong
- In MIPS pipeline, use
  - Predict-not-taken (normal flow:  $PC + 4$ )
  - Fetch instruction after branch, with no delay



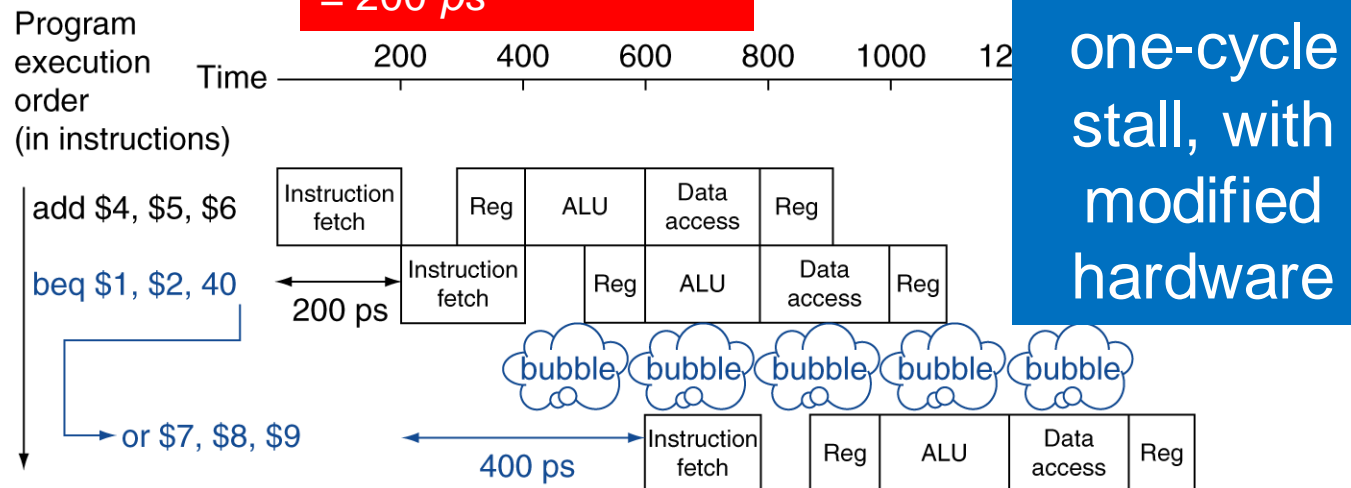
# MIPS with Predict-Not-Taken

Prediction  
correct (branch-  
not-taken)



1 clock cycle in  
pipelined machine  
= 200 ps

Prediction  
incorrect  
(branch taken)



# Control Hazards

## Branch Hazard Solutions

- #1: Stall until branch target is known (3 stalls per branch)
- #2: Move zero-checking (ZC) and target-address computation (TC) earlier (in ID/RR-Stage): Additional HW
- #3: Predict Branch-Not-Taken
  - Execute successor instructions in sequence (no stall if branch is not taken)
  - “Squash” instructions in pipeline if branch actually taken (one stall), with early ZCTC
  - 47% MIPS branches not taken on average
  - PC+4 already calculated, so use it to get next instruction

# More Realistic Branch Prediction

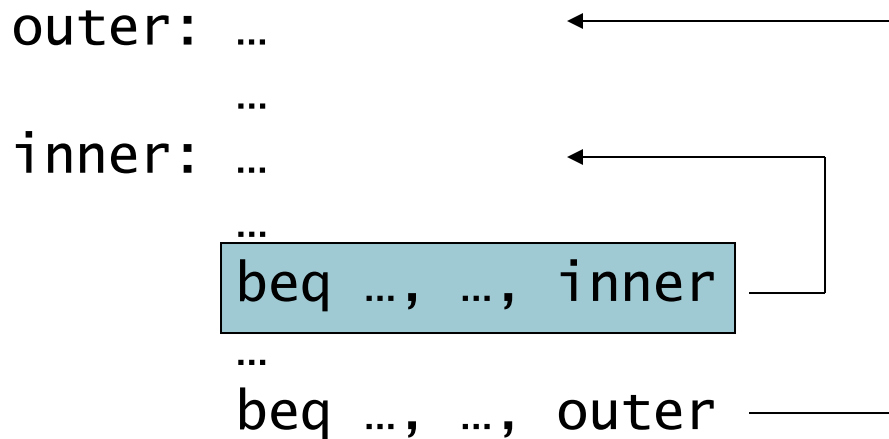
- *Static* branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not-taken
- *Dynamic* branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

# Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (*aka* branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

# 1-Bit Predictor

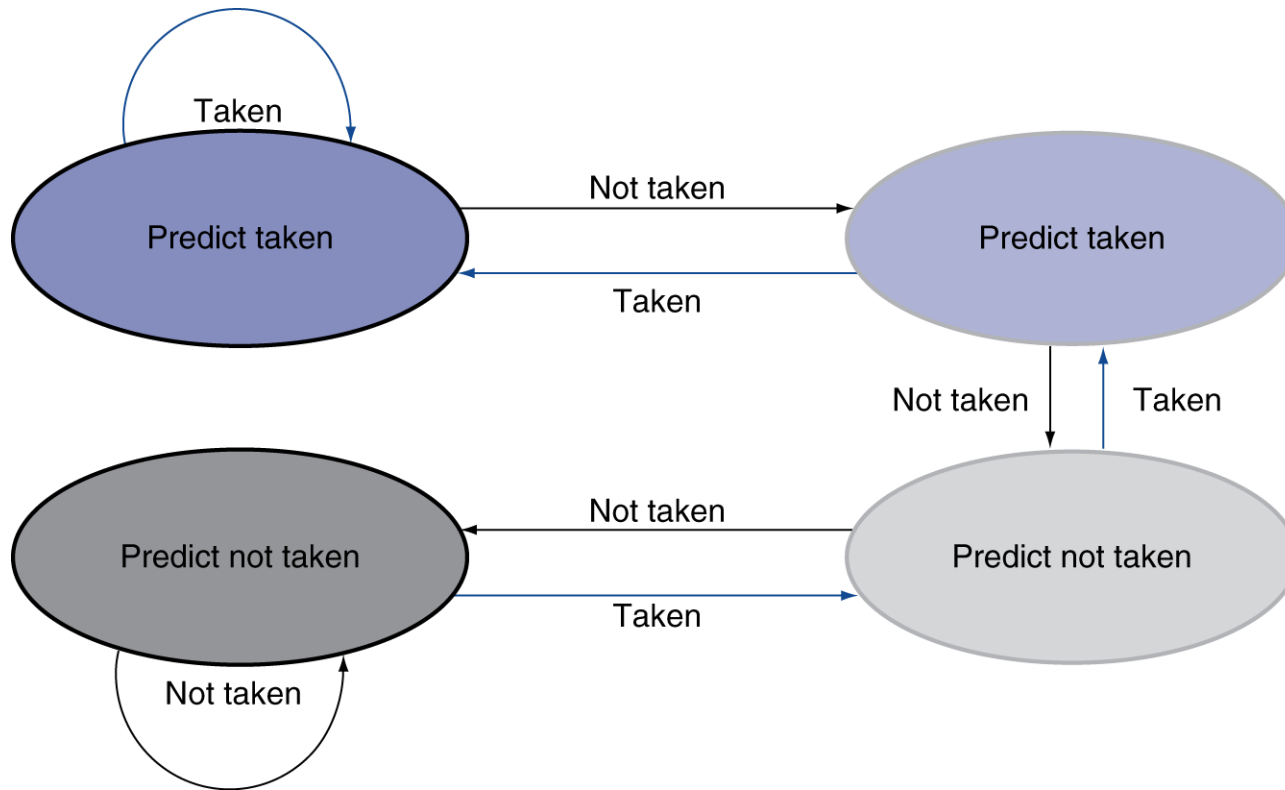
- Inner loop branches mispredicted twice!



- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

## 2-Bit Predictor: Increases Prediction Accuracy

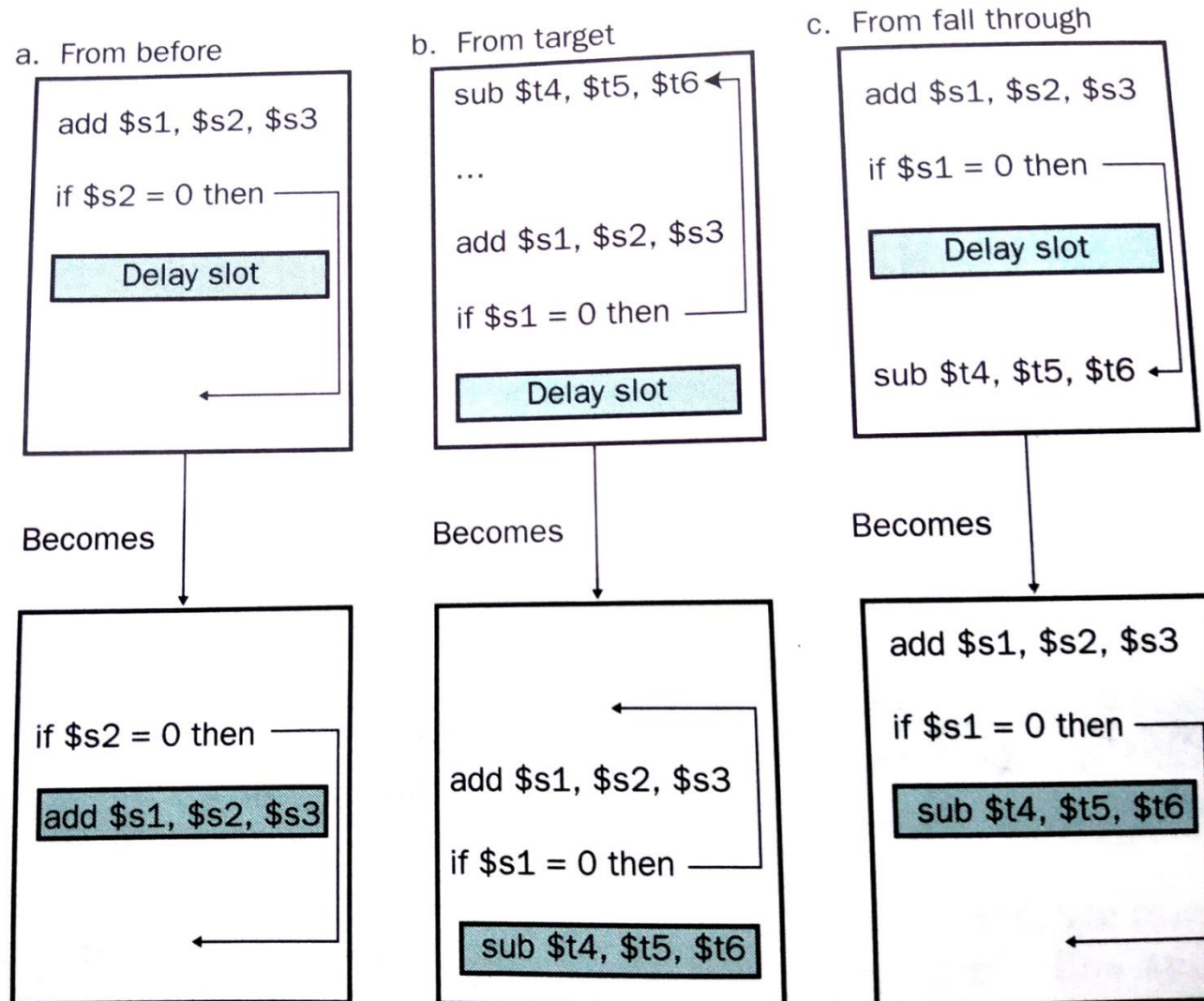
- Only change prediction on two successive mispredictions



# Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

# Another solution to improve branch hazard performance: Delay-Slot (Compiler assisted)





# Data Hazards in ALU Instructions

- Consider this sequence:

sub \$2, \$1, \$3

and \$12, \$2, \$5

or \$13, \$6, \$2

add \$14, \$2, \$2

sw \$15, 100(\$2)

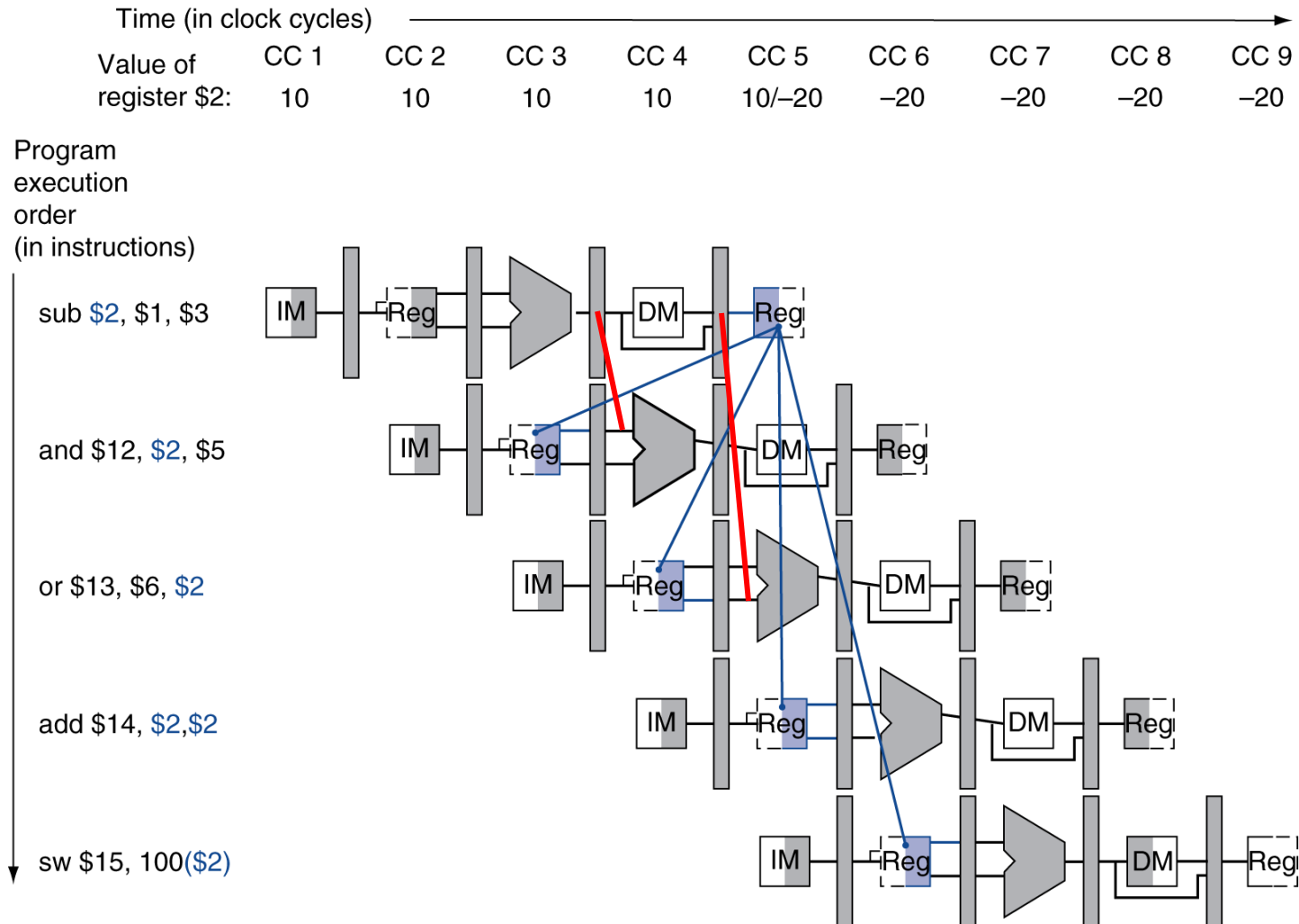
IF ID/RR EX Mem WB

IF ID/RR EX Mem WB



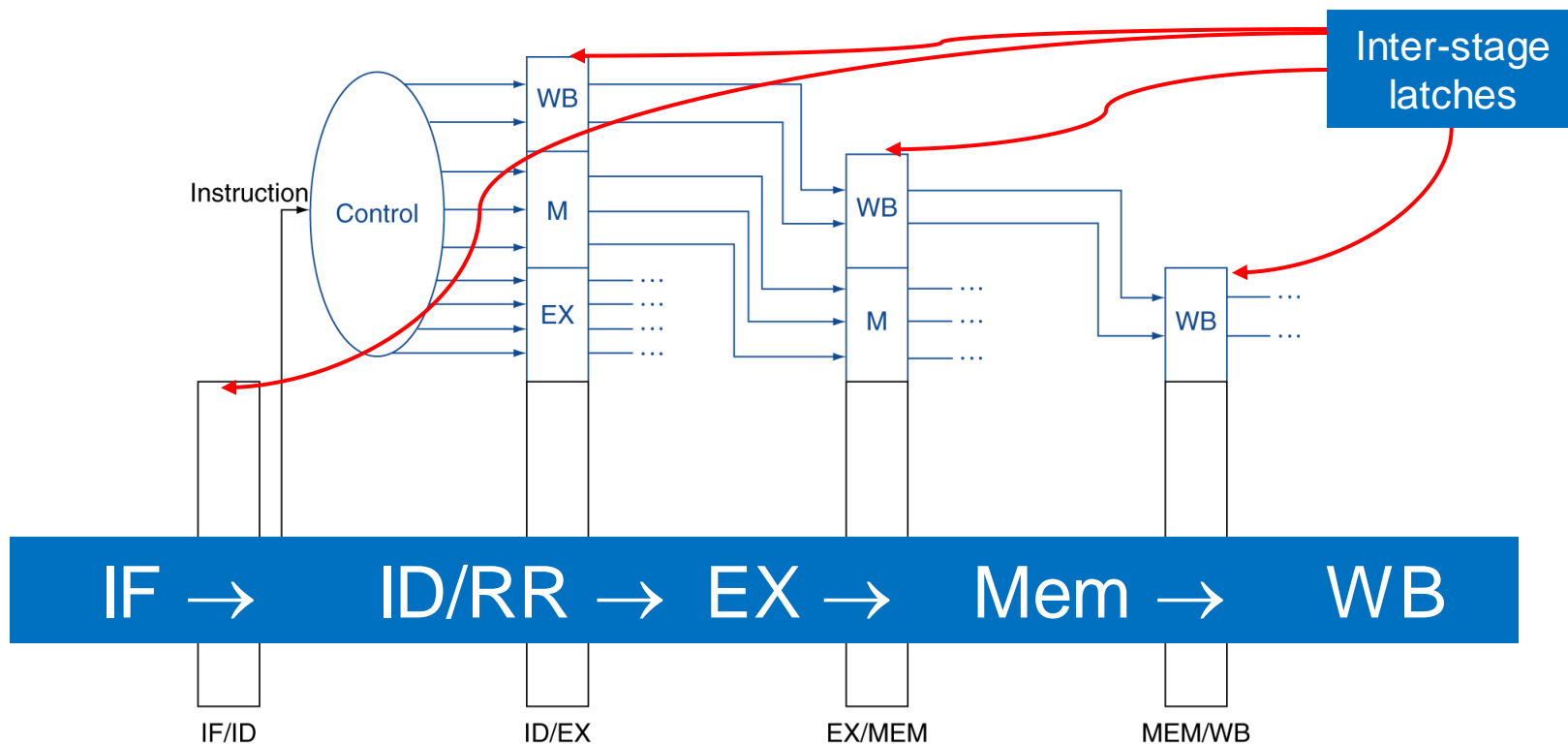
- We can resolve hazards with forwarding
  - How do we detect when to forward?

# Dependencies & Forwarding



# Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



# Detecting When to Forward

- Register numbers are passed along pipeline latches
  - e.g., ID/EX.RegisterRs  $\Rightarrow$  register number for Rs sitting in ID/EX pipeline interface latches

```
add $t1, $t2, $t3  IF  ID/RR  EX  Mem  WB
sub $t4, $t1, $t5           IF  ID/RR  EX  Mem  WB
```

- Data hazards when

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

Or 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

Or 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

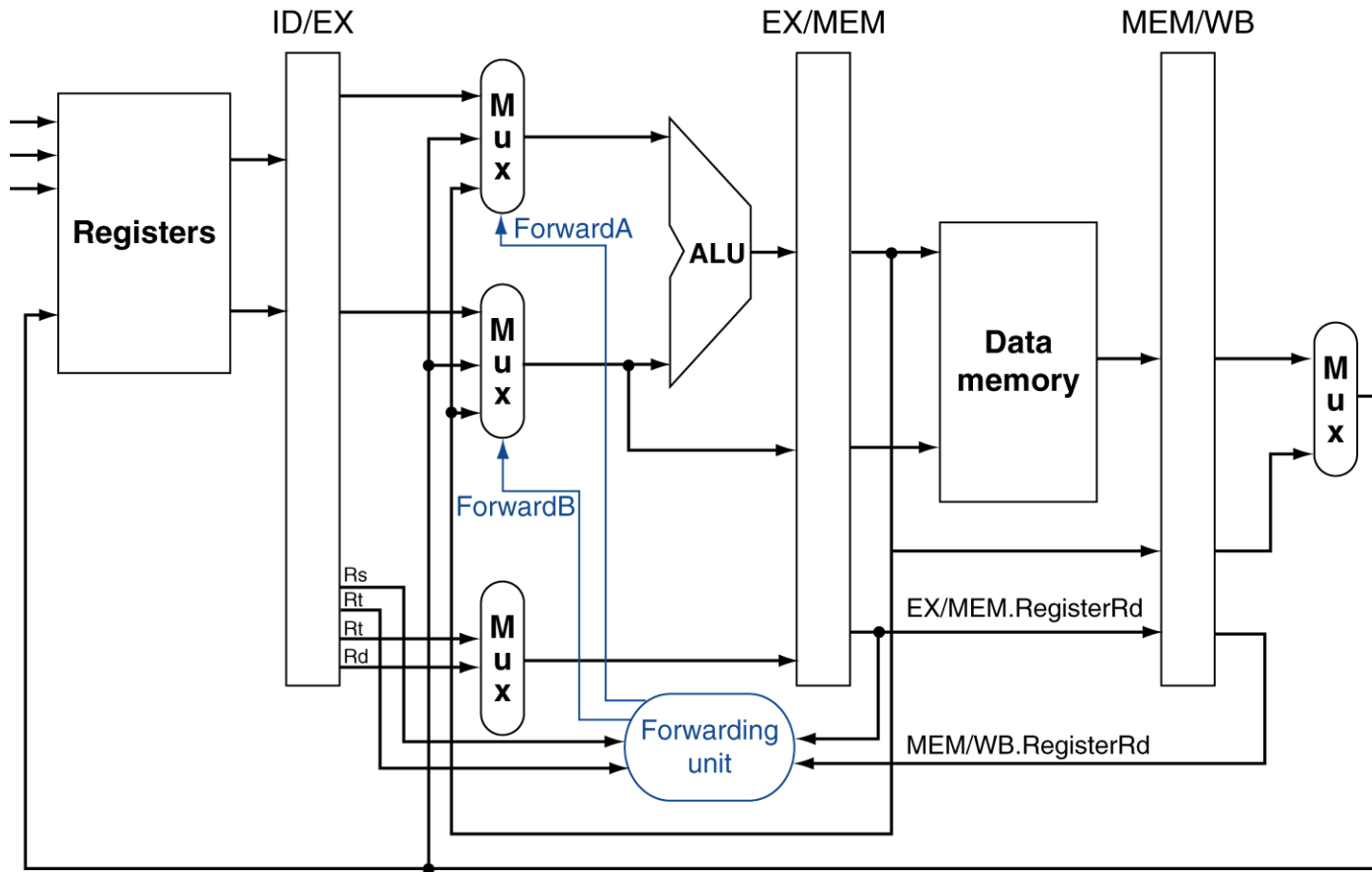
Fwd from  
EX/MEM  
pipeline reg

Fwd from  
MEM/WB  
pipeline reg

# Detecting the Need to Forward

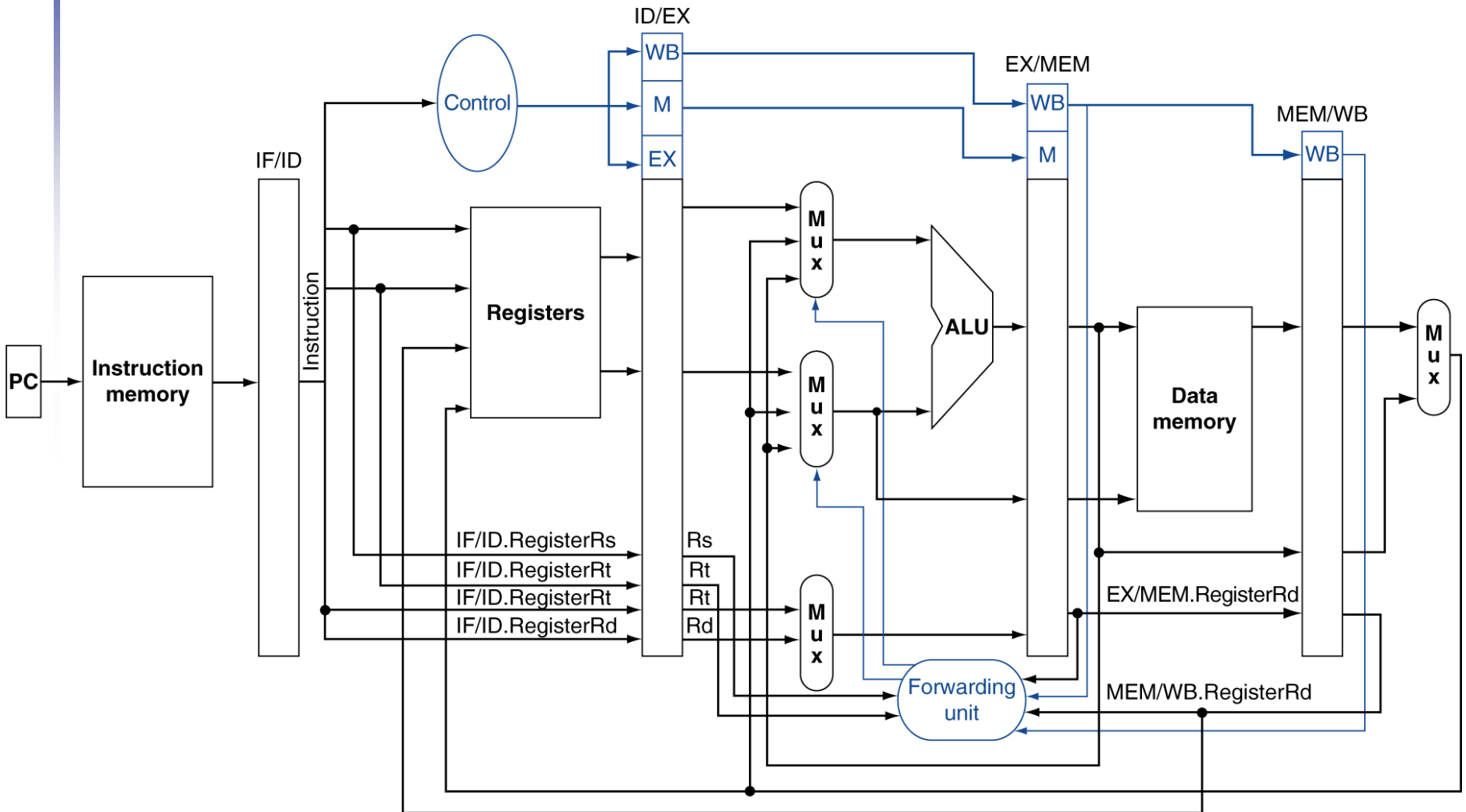
- But only when forwarded instruction writes to a register!
  - EX/MEM.RegWrite = 1, MEM/WB.RegWrite = 1
- And only if Rd for that instruction is not \$zero
  - EX/MEM.RegisterRd  $\neq$  0,  
MEM/WB.RegisterRd  $\neq$  0

# Forwarding Paths

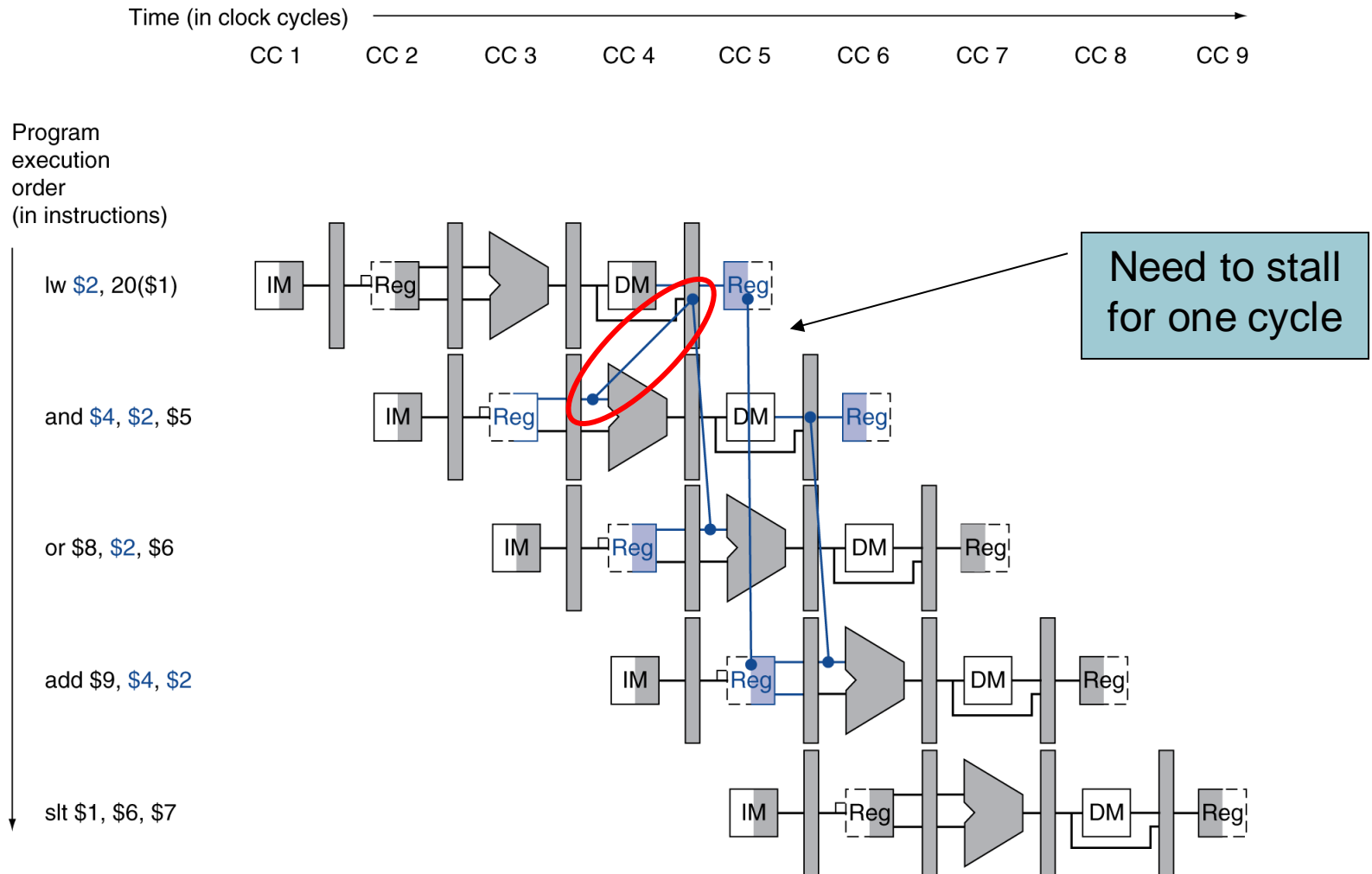


b. With forwarding

# Datapath with Forwarding



# Load-Use Data Hazard





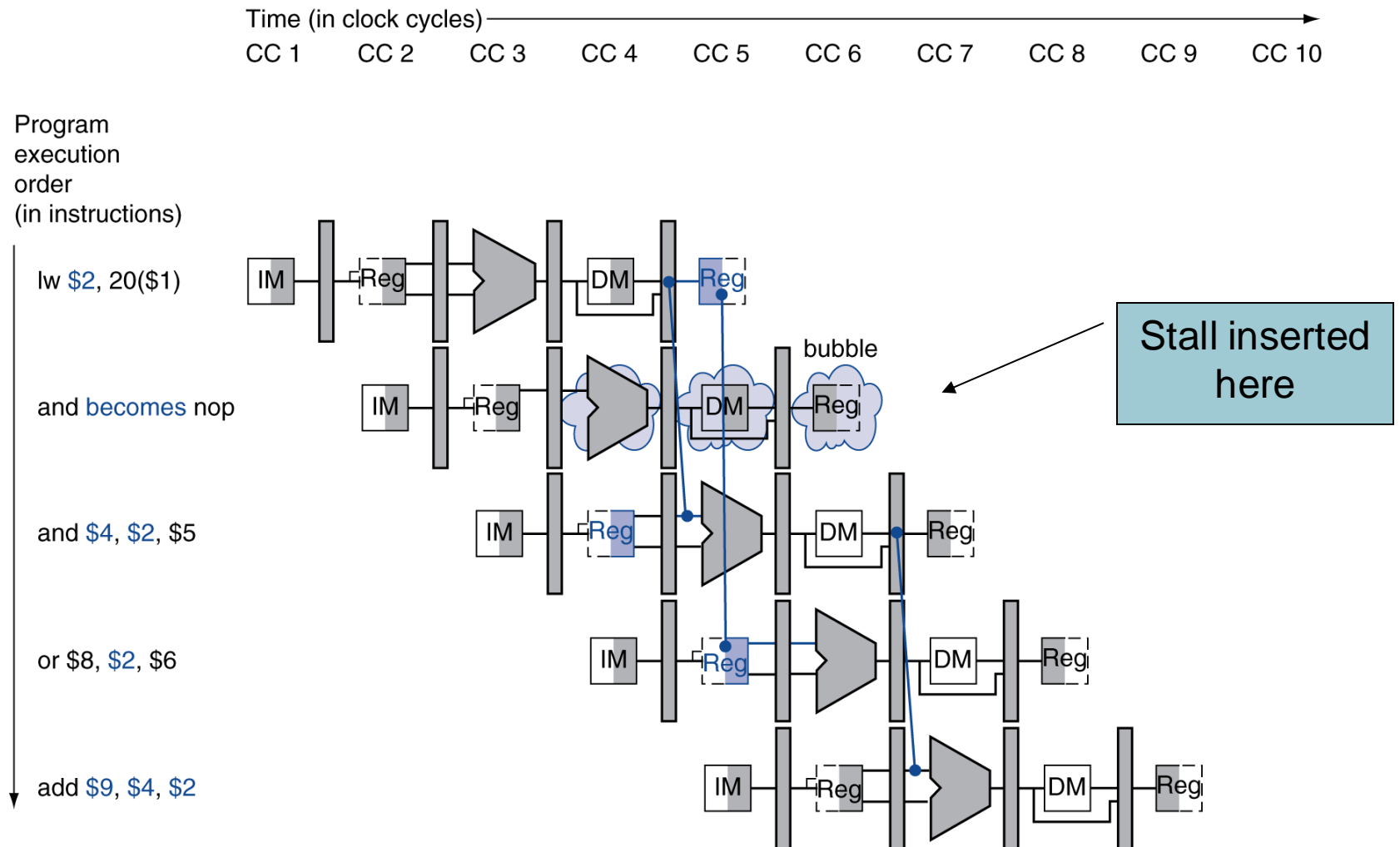
# Detection of Load-Data Hazards in Pipeline

CC	1	2	3	4	5	6	7	8
LW R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

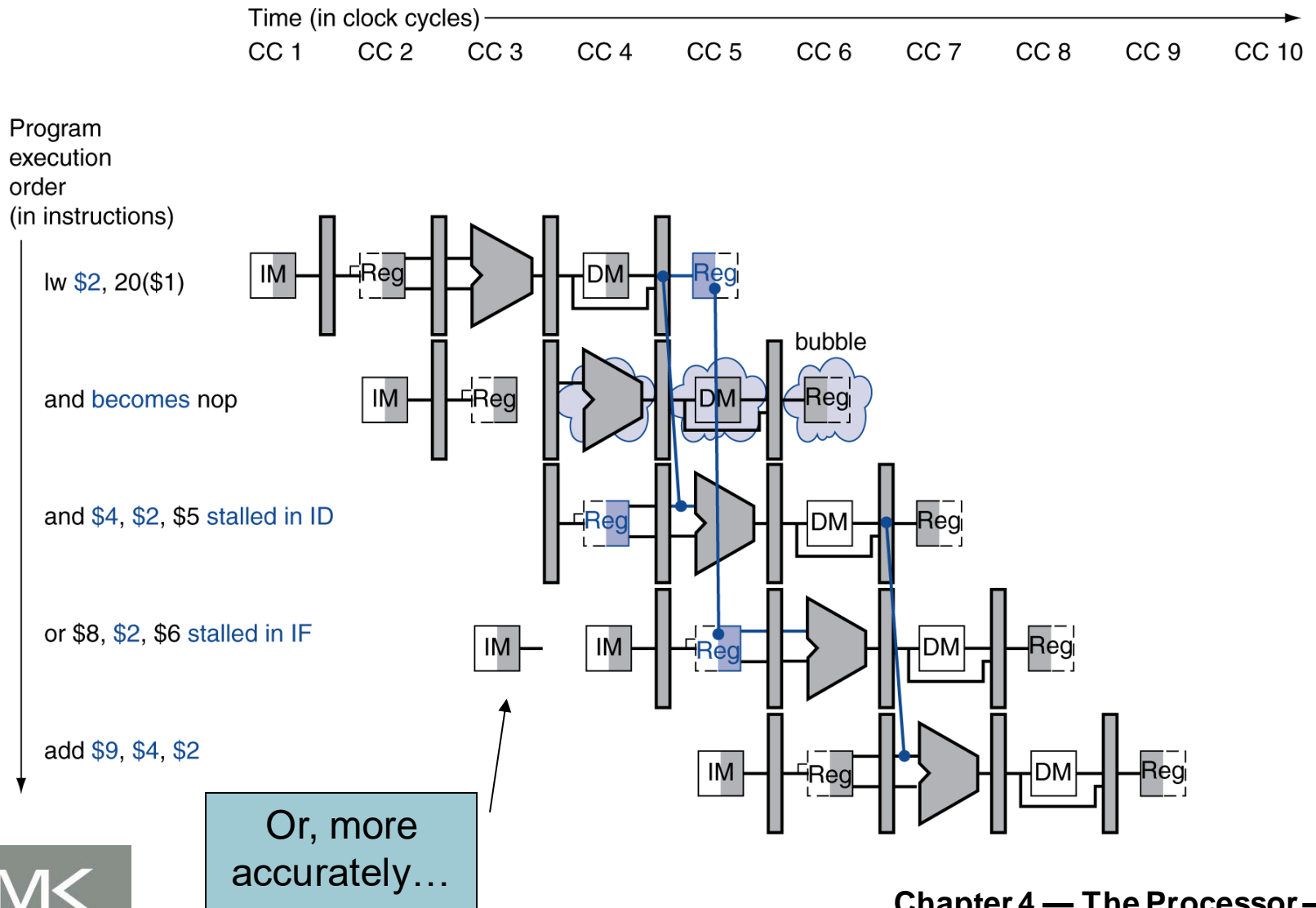
Stall is inevitable when the *destination* of load is a *source* of the next instruction

- Load-use hazard is detected when  
ID/EX.MemRead  
*and*  
((ID/EX.RegisterRt = IF/ID.RegisterRs) or  
(ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble

# Stall/Bubble in the Pipeline



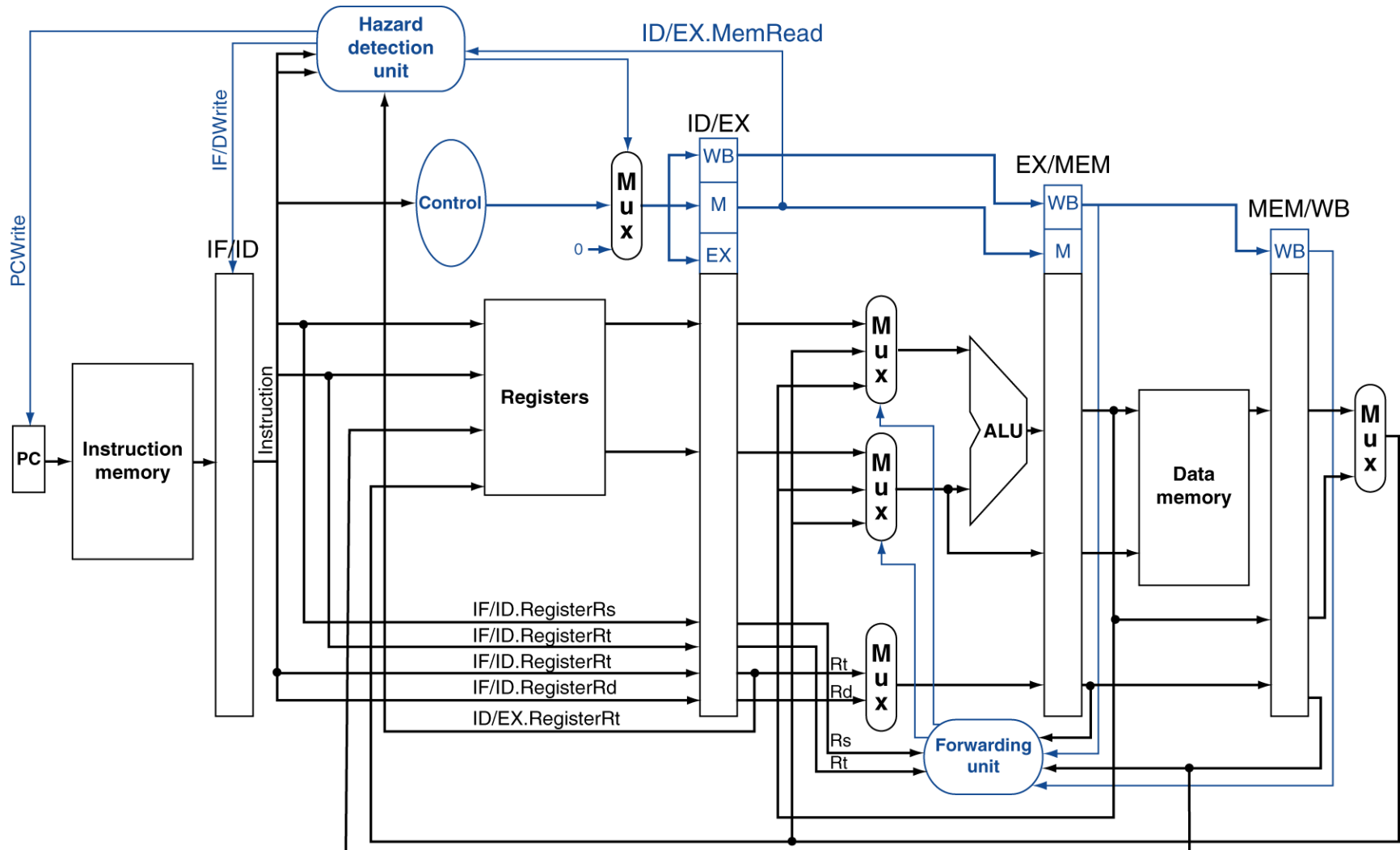
# Stall/Bubble in the Pipeline



# How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do *NOP* (no-operation)
- Prevent update of PC and IF/ID register
  - Used instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for 1w
    - Can subsequently forward to EX-stage

# Datapath with Forwarding and Hazard Detection



# Stalls and Performance

## The BIG Picture

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

# Performance Equation

Pipeline overhead may degrade ideal speed-up

- Pipeline Speed-Up in an *ideal* machine
- $= \text{CPI\_unpipelined} / \text{CPI\_Pipelined}$
- $\approx \text{Pipeline\_Depth} / \text{CPI\_Pipelined}$
- $\approx \text{Pipeline\_Depth}$

Ideal CPI\_Pipelined = 1

- Pipeline Speed-Up in the presence of *stalls*
- $= \text{CPI\_unpipelined} / \text{CPI\_Pipelined}$
- $\approx \text{Pipeline\_Depth} / (1 + \text{Avg. \#Pipeline\_Stall\_Cycles\_per\_Instruction})$
- Example (Impact of branch penalty):
- Pipeline Speed-Up
- $= \text{Pipeline\_Depth} / \{1 + (\text{Branch\_Freq.} \times \text{Branch\_Penalty})\}$

# Hazards: Summary

- Situations that prevent starting the next instruction in the next cycle; slows down the pipeline! Effective CPI increases.
- **Structural hazards**
  - A required resource is busy
- **Data hazard**
  - Need to wait for previous instruction to complete its data read/write
- **Control hazard**
  - Deciding on control action depends on previous instruction



# Hazards: Summary

To avoid data hazard:

1. Use bubbles (stalls)
2. Use half-cycle Register Write (first)/Register Read (second)
3. Use data forwarding by hardware (EX/Mem  $\rightarrow$  ALU\_in; Mem/WB  $\rightarrow$  ALU\_in)
4. Use code-reordering by compiler

To avoid structural hazard:

1. Use bubbles (stalls)
2. Use separate instruction/data memories or, separate instruction/data caches

# Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

# Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall
  - Handling exceptions need special techniques

# **Announcement of Class Test 03 (last one)**



- Monday, 15 November 2021
- Time: 12:00 noon – 02:00 pm
- Credit: 40%
- Coverage: Processor Design, Memory, Exceptions/Interrupts, and Pipelining