

# The Processor (3)

Lecture 6

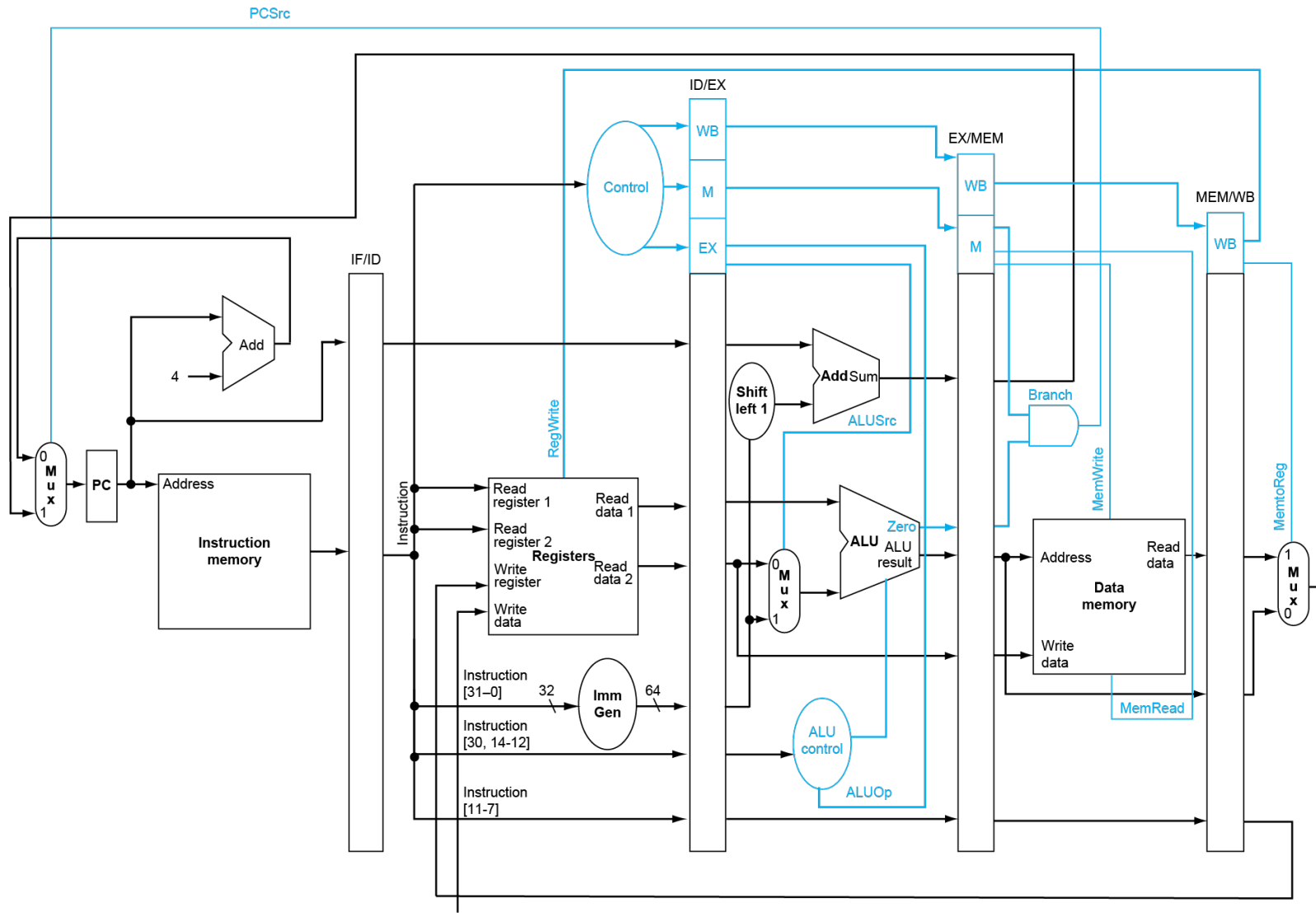
October 16<sup>th</sup>, 2023

Jae W. Lee ([jaewlee@snu.ac.kr](mailto:jaewlee@snu.ac.kr))

Computer Science and Engineering

Seoul National University

***Slide credits:*** [COD:RV2e] slides from Elsevier Inc.



# Hazards

- **Situations that prevent starting the next instruction in the next cycle**
- **Hazard 1: Structure hazards**
  - A required resource is busy
- **Hazard 2: Data hazard**
  - Need to wait for previous instruction to complete its data read/write
- **Hazard 3: Control hazard**
  - Deciding on control action depends on previous instruction

# Structure Hazards

- **Conflict for use of a resource**
- **In RISC-V pipeline with a single memory**
  - Load/store requires data access
  - Instruction fetch would have to stall for that cycle
    - Would cause a pipeline “bubble”
- **Hence, pipelined datapaths require separate instruction/data memories**
  - Or separate instruction/data caches

# Outline

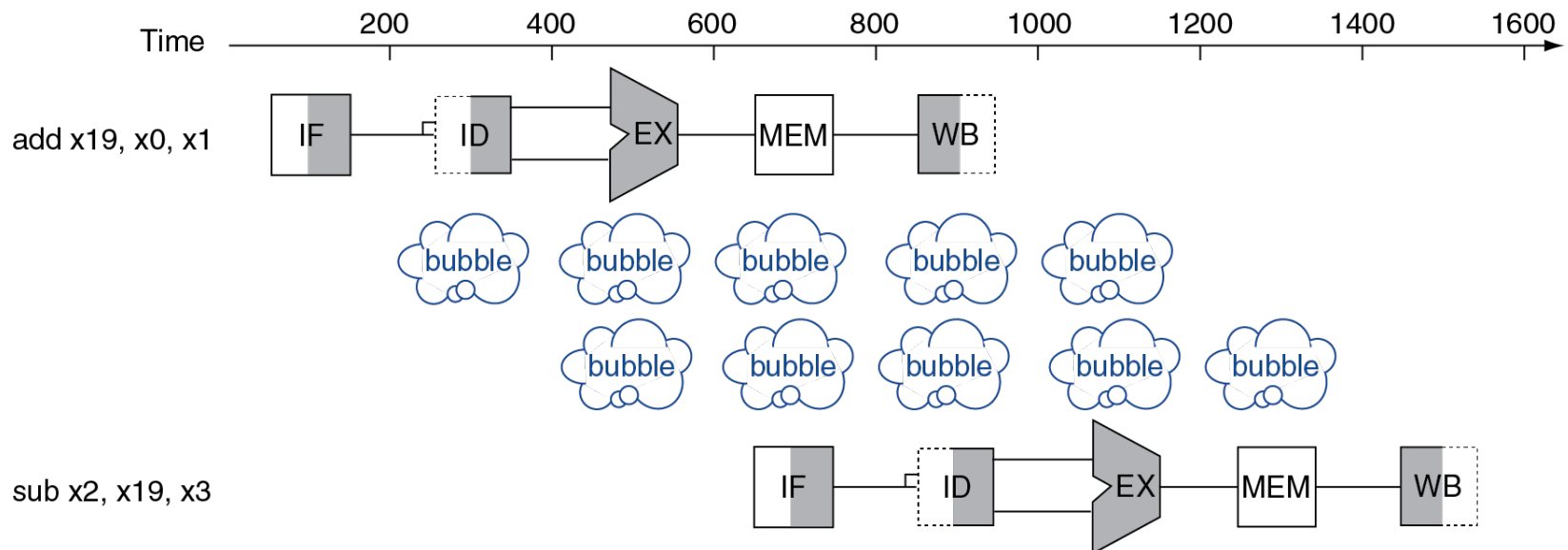
**Textbook: [COD] 4.7-4.8**

- **Data Hazards: Forwarding versus Stalling**
  - Stalling for Load-Use Hazards
- **Control Hazards**

# Data Hazards (Review)

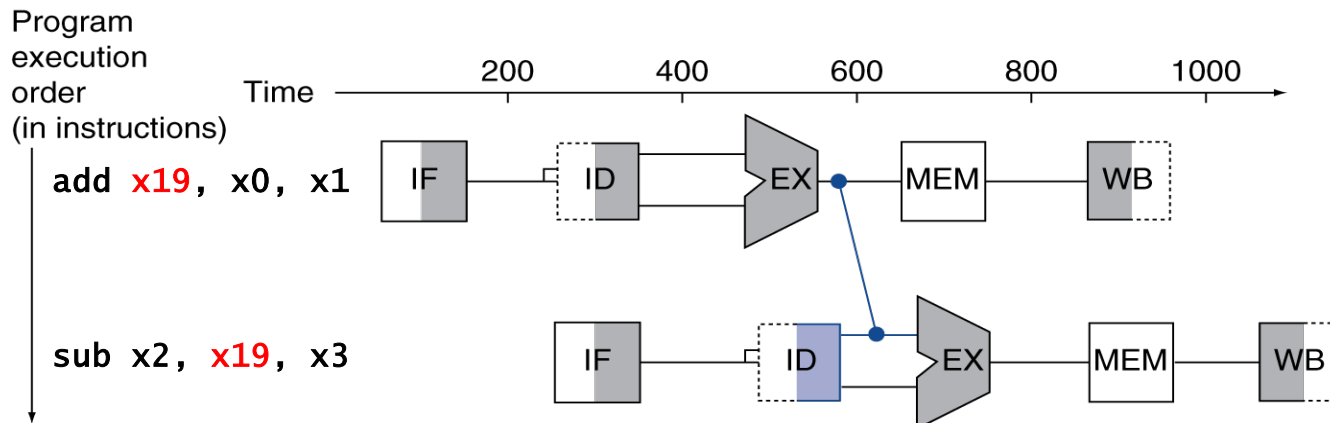
- An instruction depends on completion of data access by a previous instruction

```
add    x19, x0, x1
sub     x2, x19, x3
```



# 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



# Data Hazards in ALU Instructions

- Consider this sequence:

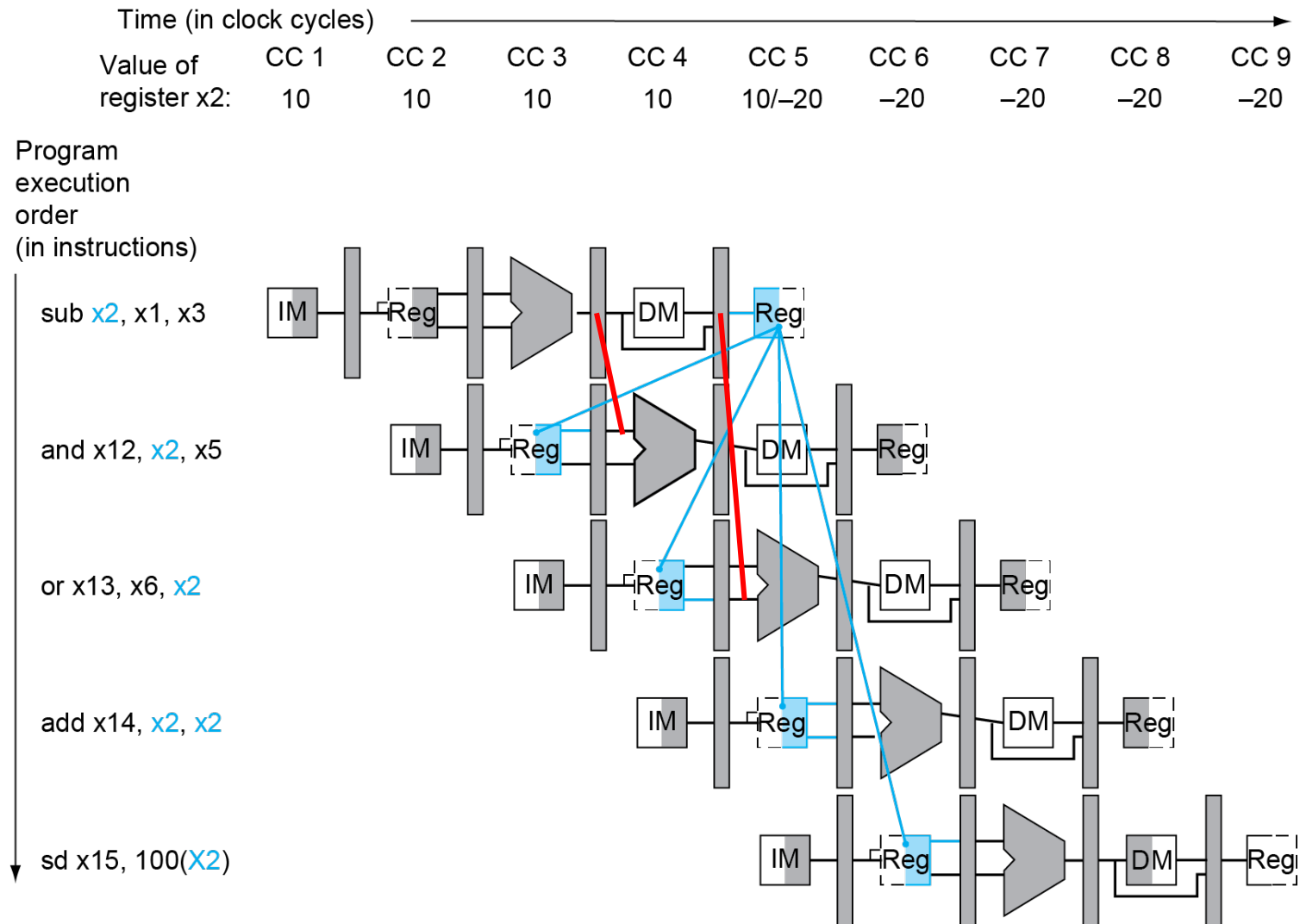
```
sub  x2, x1, x3
and  x12, x2, x5
or   x13, x6, x2
add  x14, x2, x2
sd   x15, 100(x2)
```

- We can resolve hazards with forwarding

- How do we detect when to forward?



# Dependencies & Forwarding



# Detecting the Need to Forward

- **Pass register numbers along pipeline**
  - e.g., ID/EX.RegisterRs1 = register number for Rs1 sitting in ID/EX pipeline register
- **ALU operand register numbers in EX stage are given by**
  - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- **Data hazards when**
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

}

Fwd from  
EX/MEM  
pipeline reg

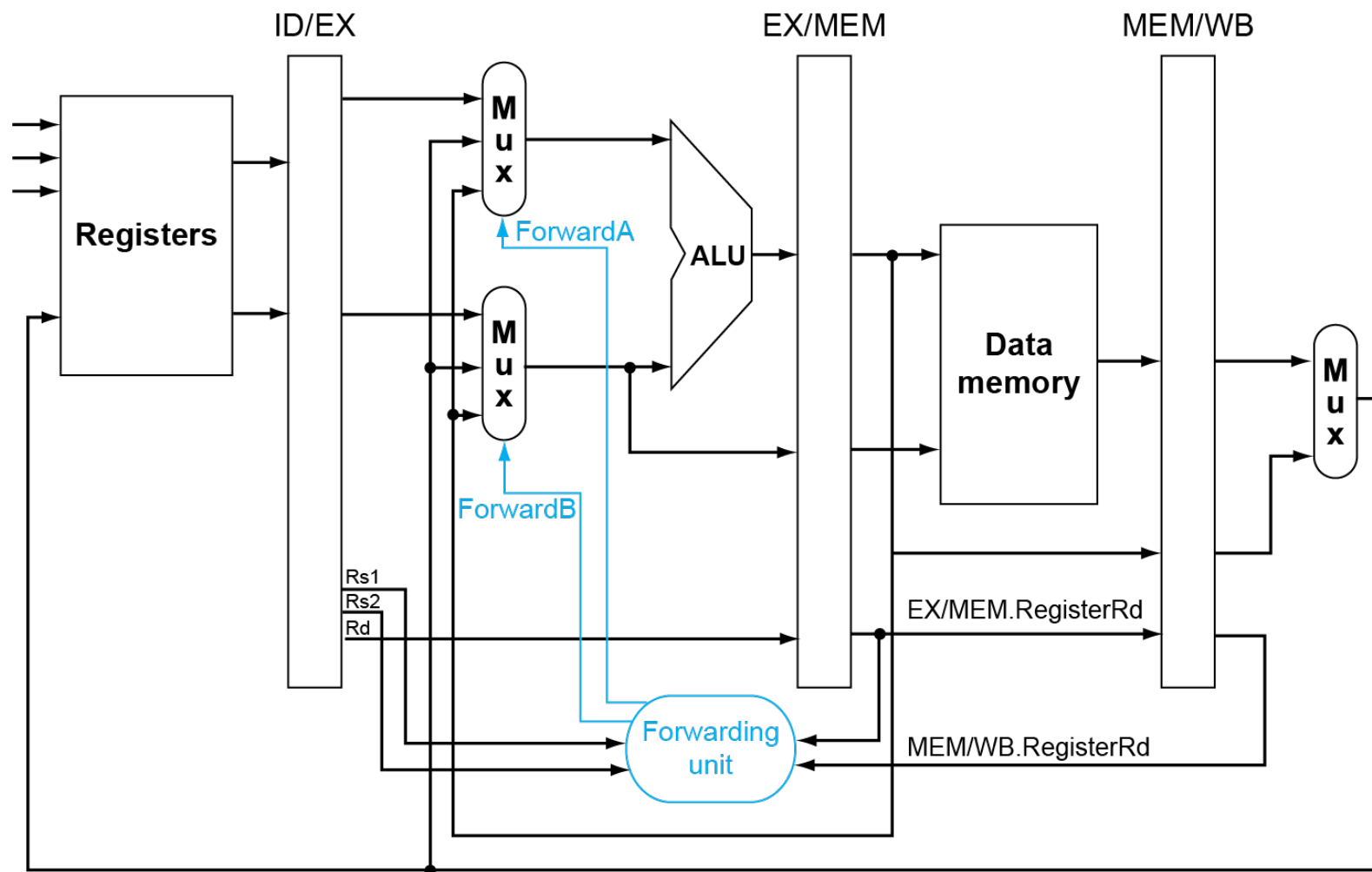
}

Fwd from  
MEM/WB  
pipeline reg

# Detecting the Need to Forward

- **But only if forwarding instruction will write to a register!**
  - EX/MEM.RegWrite, MEM/WB.RegWrite
  
- **And only if Rd for that instruction is not x0**
  - EX/MEM.RegisterRd  $\neq$  0,  
MEM/WB.RegisterRd  $\neq$  0

# Forwarding Paths



# Forwarding Conditions

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

# Double Data Hazard

- Consider the sequence:

  - add **x1**, x1, x2

  - add **x1**, **x1**, x3

  - add x1, **x1**, x4

- Both hazards occur

  - Want to use the most recent

- Revise MEM hazard condition

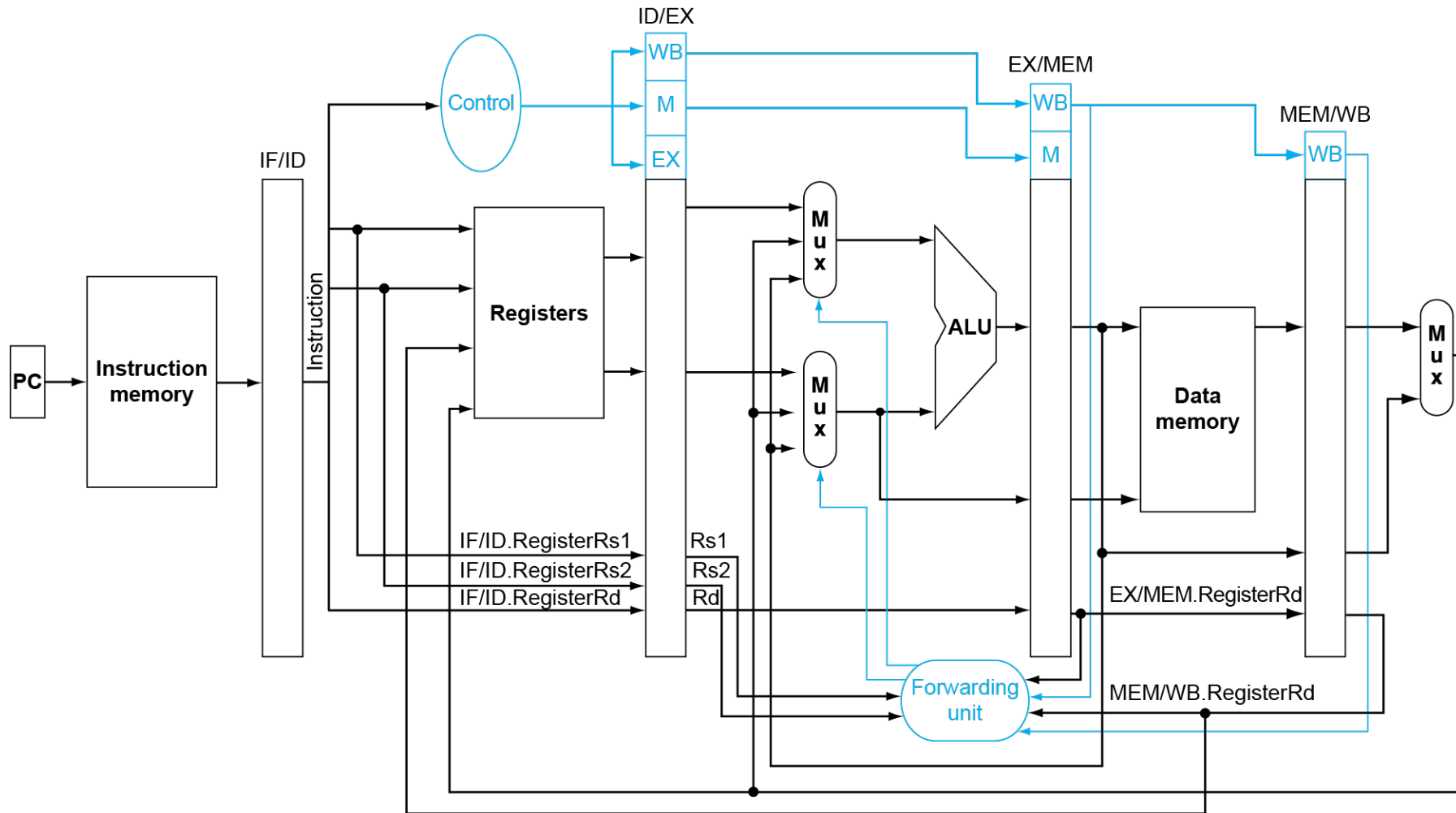
  - Only fwd if EX hazard condition isn't true

# Revised Forwarding Condition

## ■ MEM hazard

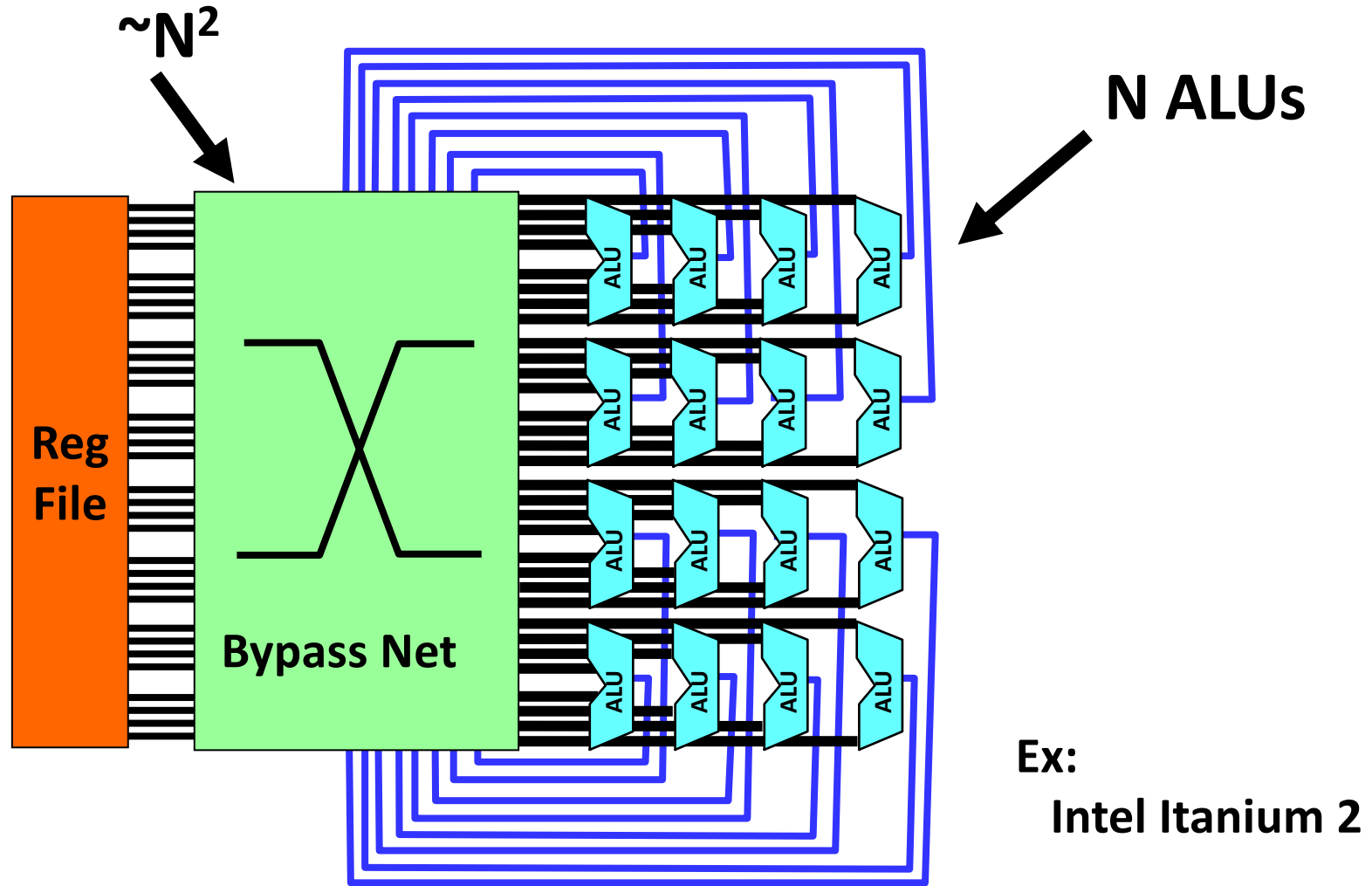
- if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd  $\neq$  0)  
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd  $\neq$  ID/EX.RegisterRs1))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
- if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd  $\neq$  0)  
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd  $\neq$  ID/EX.RegisterRs2))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

# Datapath with Forwarding





# Bypass (Forwarding) Network is a Real Concern



**Without modification, freq decreases linearly or worse.**

Source: Prof. Michael Taylor @ UCSD

SNU 4190.308-001: Computer Architecture (Fall 2023)

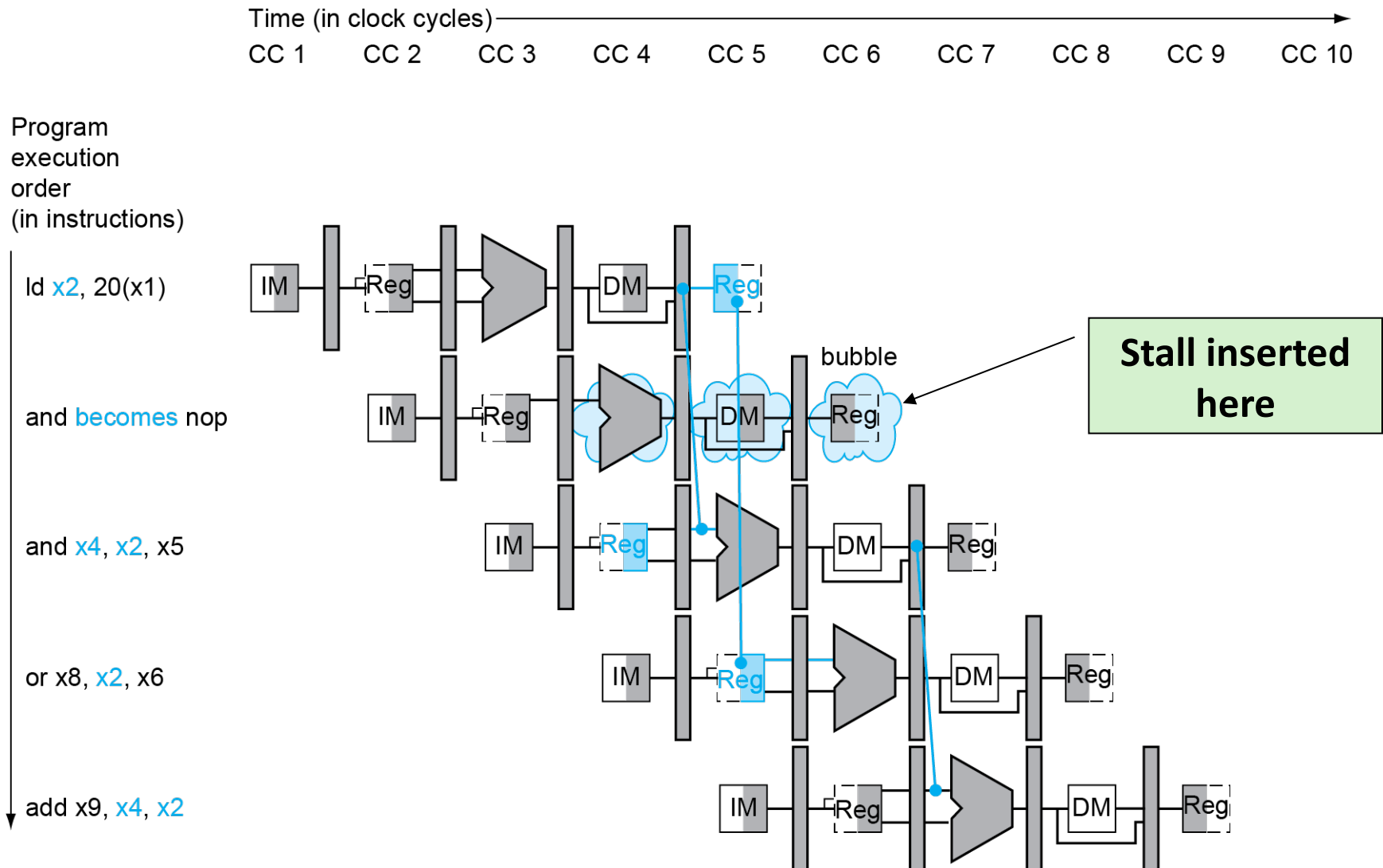
# Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- Load-use hazard when
  - ID/EX.MemRead and  
((ID/EX.RegisterRd = IF/ID.RegisterRs1) or  
(ID/EX.RegisterRd = IF/ID.RegisterRs2))
- If detected, stall and insert bubble

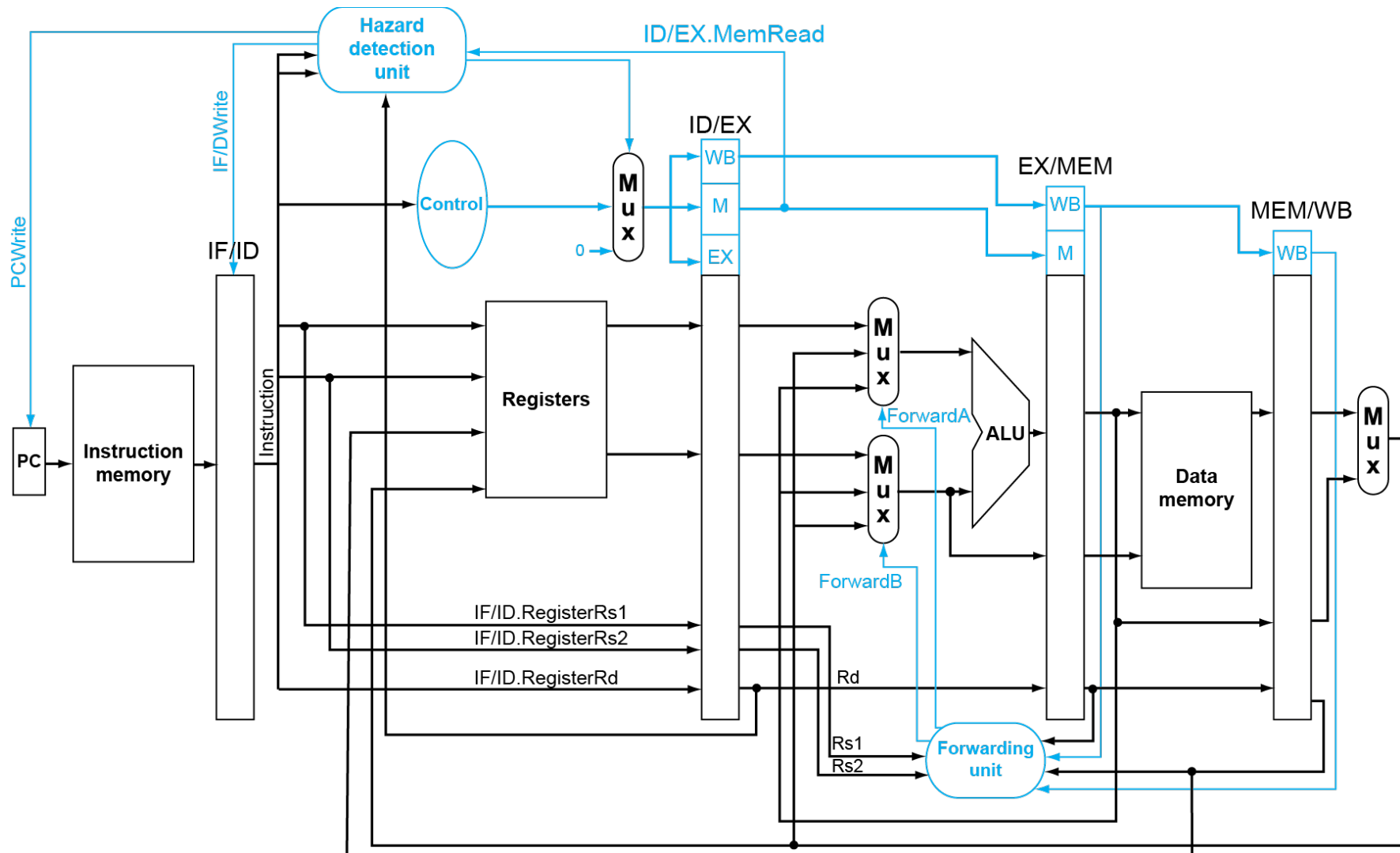
# How to Stall the Pipeline

- **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**
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for 1 d
    - Can subsequently forward to EX stage

# Load-Use Data Hazard



# Datapath with Hazard Detection



# Stalls and Performance

- **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

# Outline

**Textbook: [COD] 4.7-4.8**

- **Data Hazards: Forwarding versus Stalling**
  - Stalling for Load-Use Hazards
- **Control Hazards**

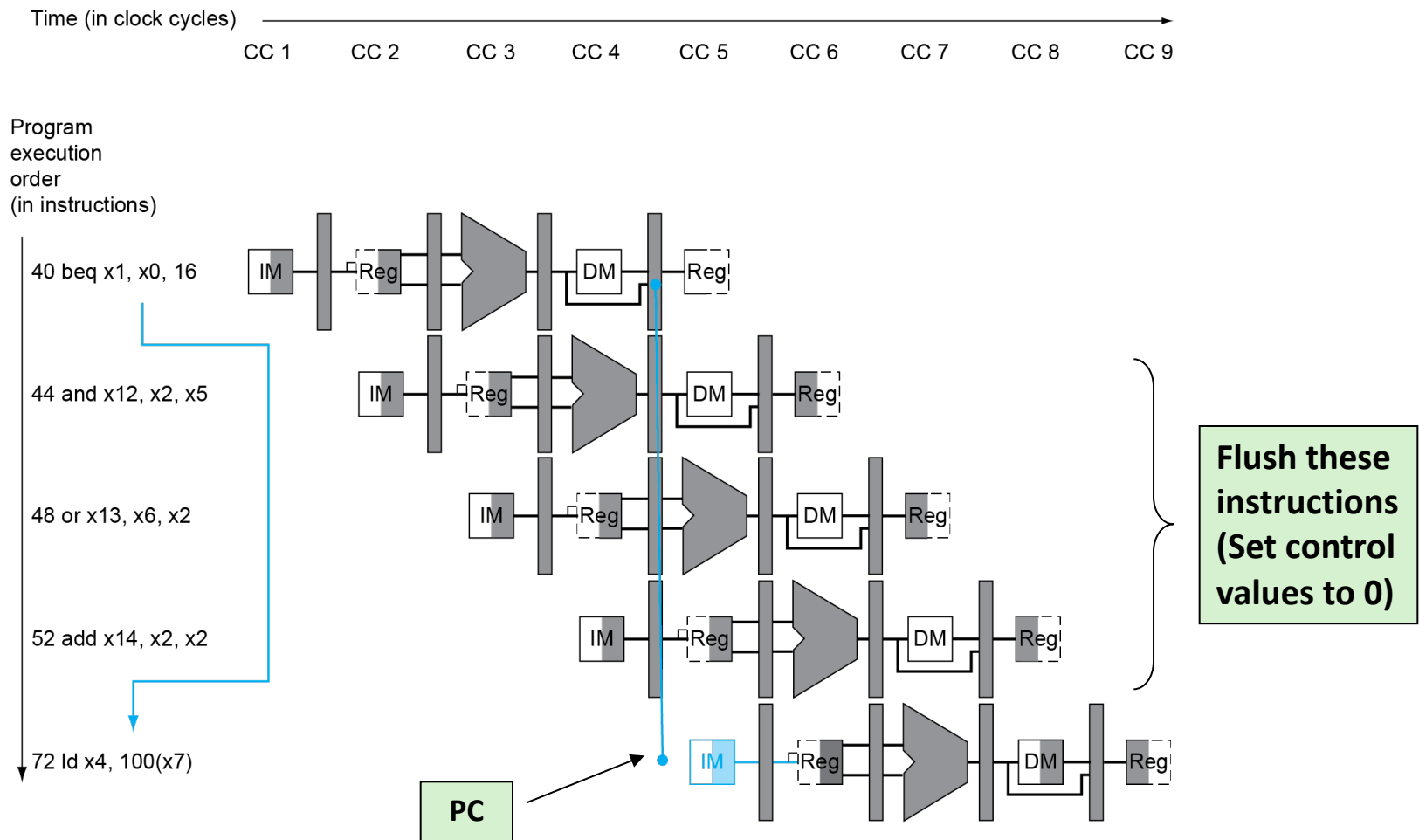
# Control Hazards

- **Branch 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



# Branch Hazards

## ■ If branch outcome determined in MEM



# Reducing Branch Delay

## ■ Move hardware to determine outcome to ID stage

- Target address adder
- Register comparator

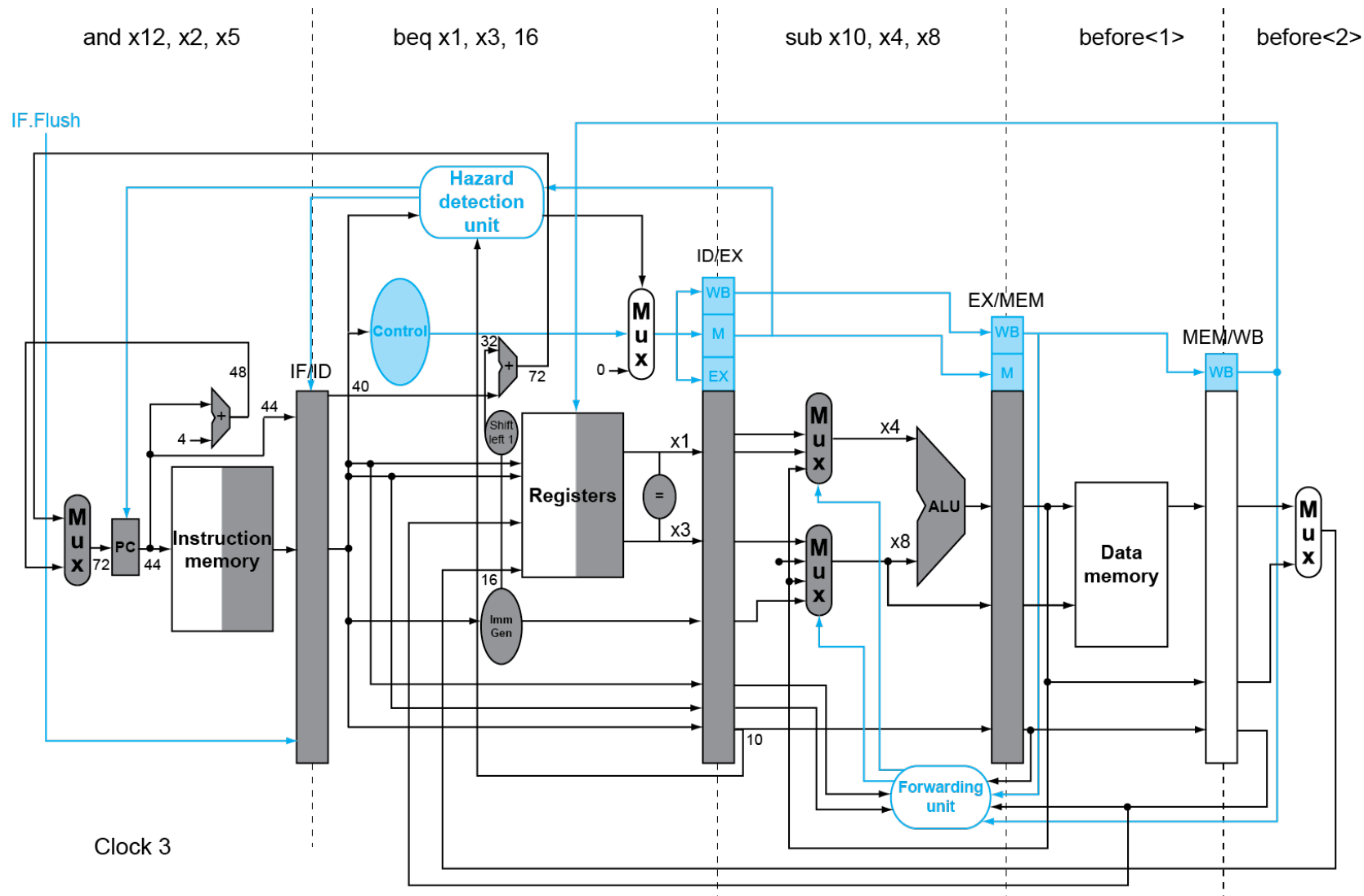
## ■ Example: branch taken

```
36:  sub    x10, x4, x8
40:  beq    x1,  x3, 16    // PC-relative branch
                             // to 40+16*2=72

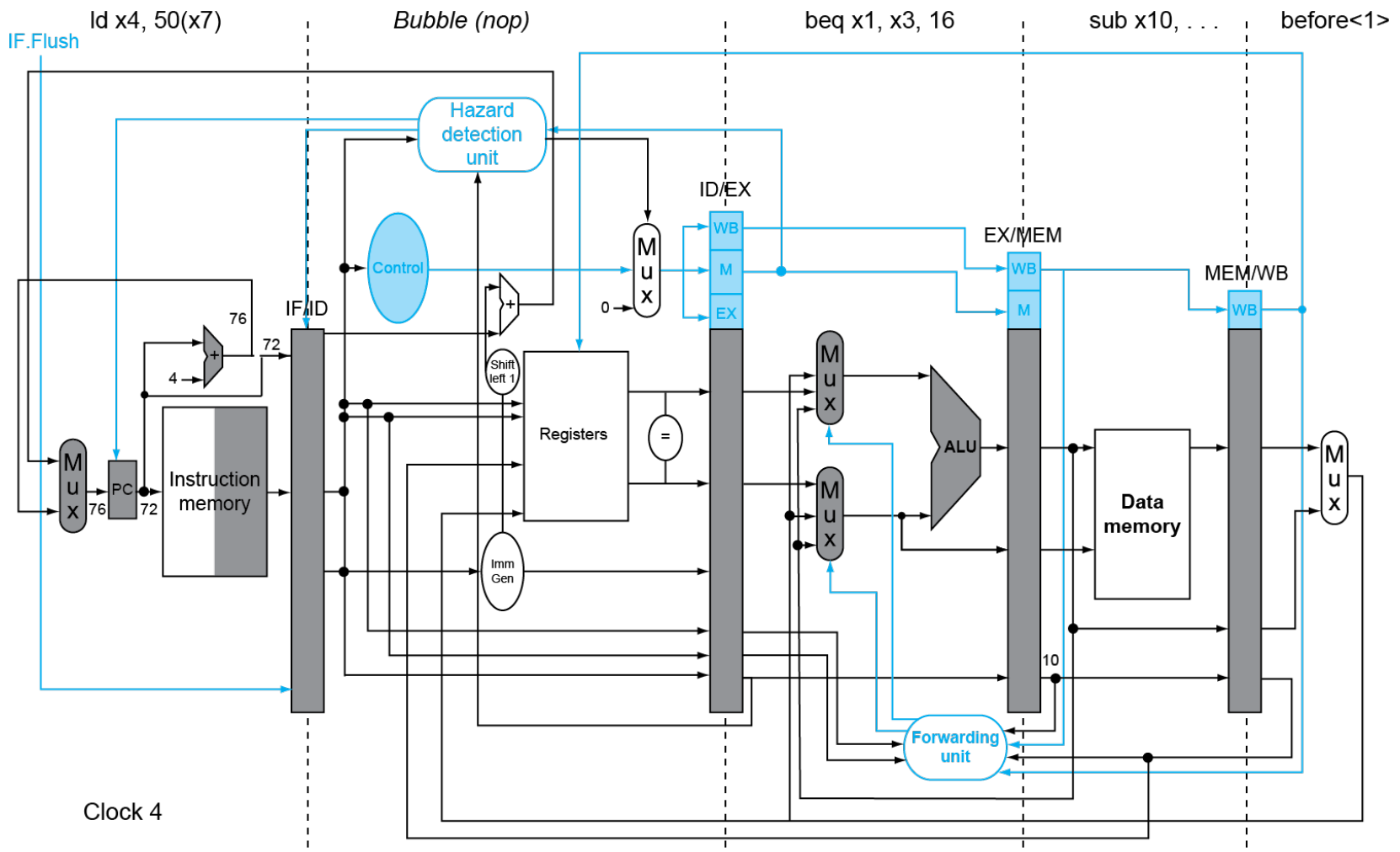
44:  and    x12, x2, x5
48:  orr    x13, x2, x6
52:  add    x14, x4, x2
56:  sub    x15, x6, x7

    ...
72:  ld     x4, 50(x7)
```

# Example: Branch Taken

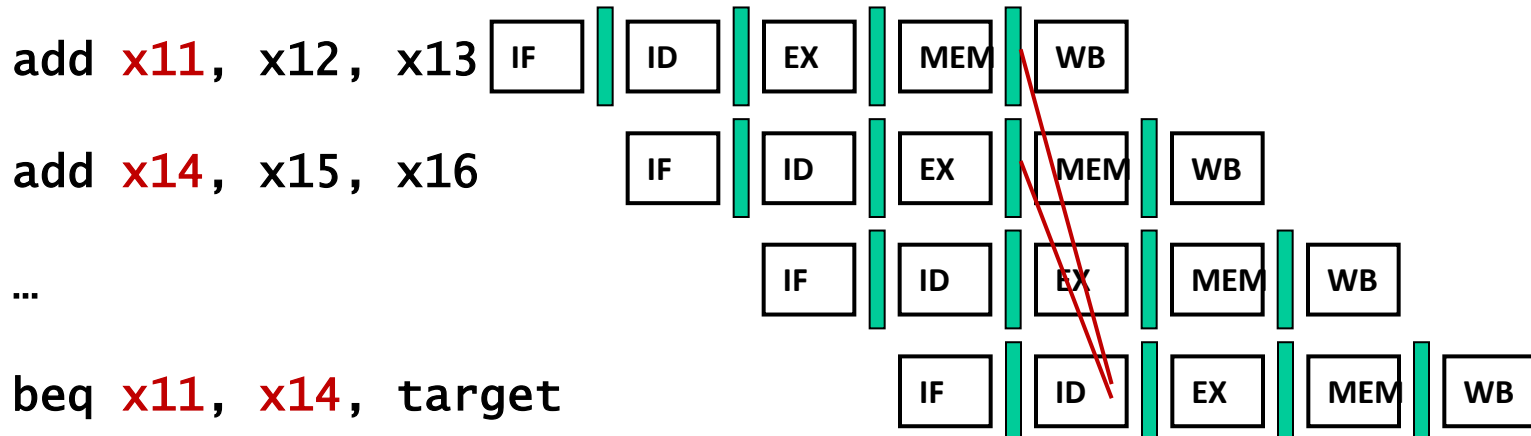


# Example: Branch Taken



# Data Hazards for Branches

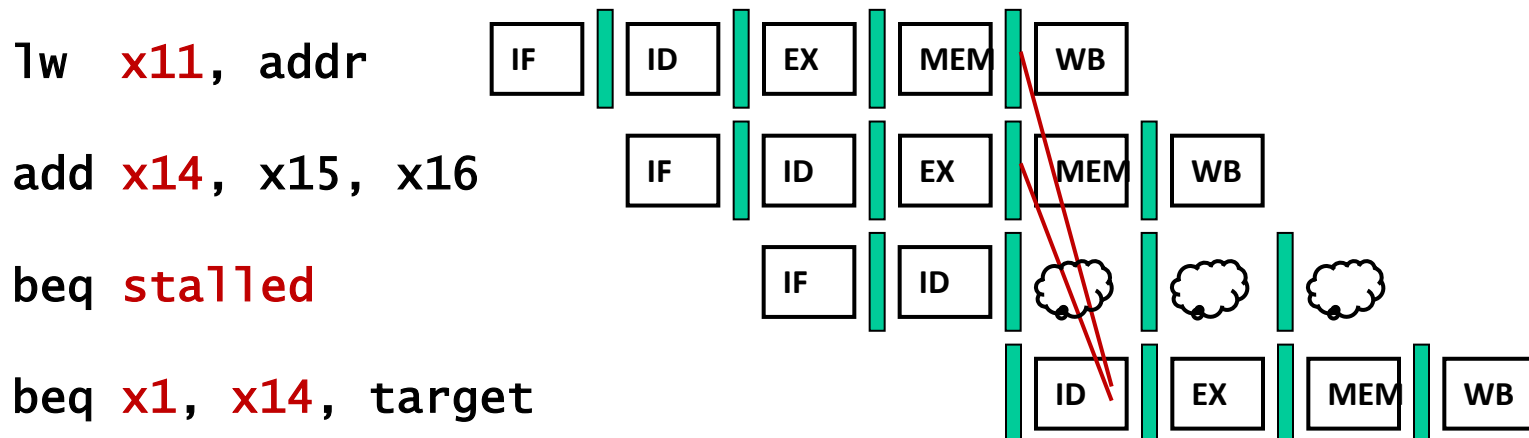
- If a comparison register is a destination of 2<sup>nd</sup> or 3<sup>rd</sup> preceding ALU instruction



- Can resolve using forwarding

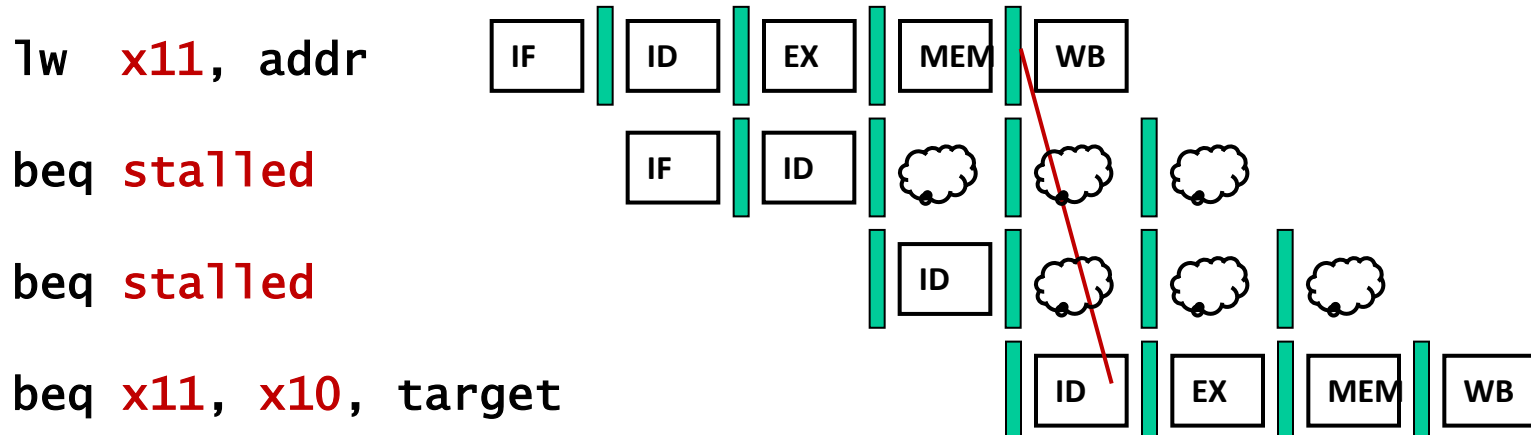
# Data Hazards for Branches

- 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



# Branch Prediction

- **Longer pipelines can't readily determine branch outcome early**
  - Stall penalty becomes unacceptable
- **Predict outcome of branch**
  - Only stall if prediction is wrong
- **In RISC-V pipeline**
  - Can predict branches not taken (**static** branch prediction)
  - Fetch instruction after branch, with no delay

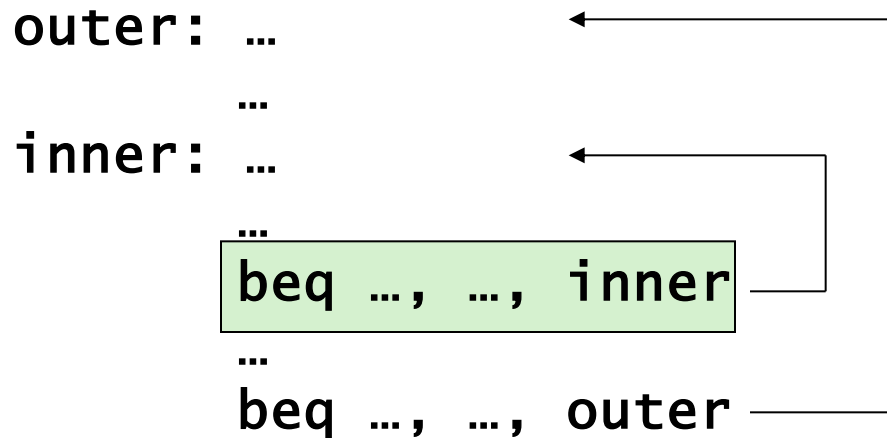


# Dynamic Branch Prediction

- 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: Shortcoming

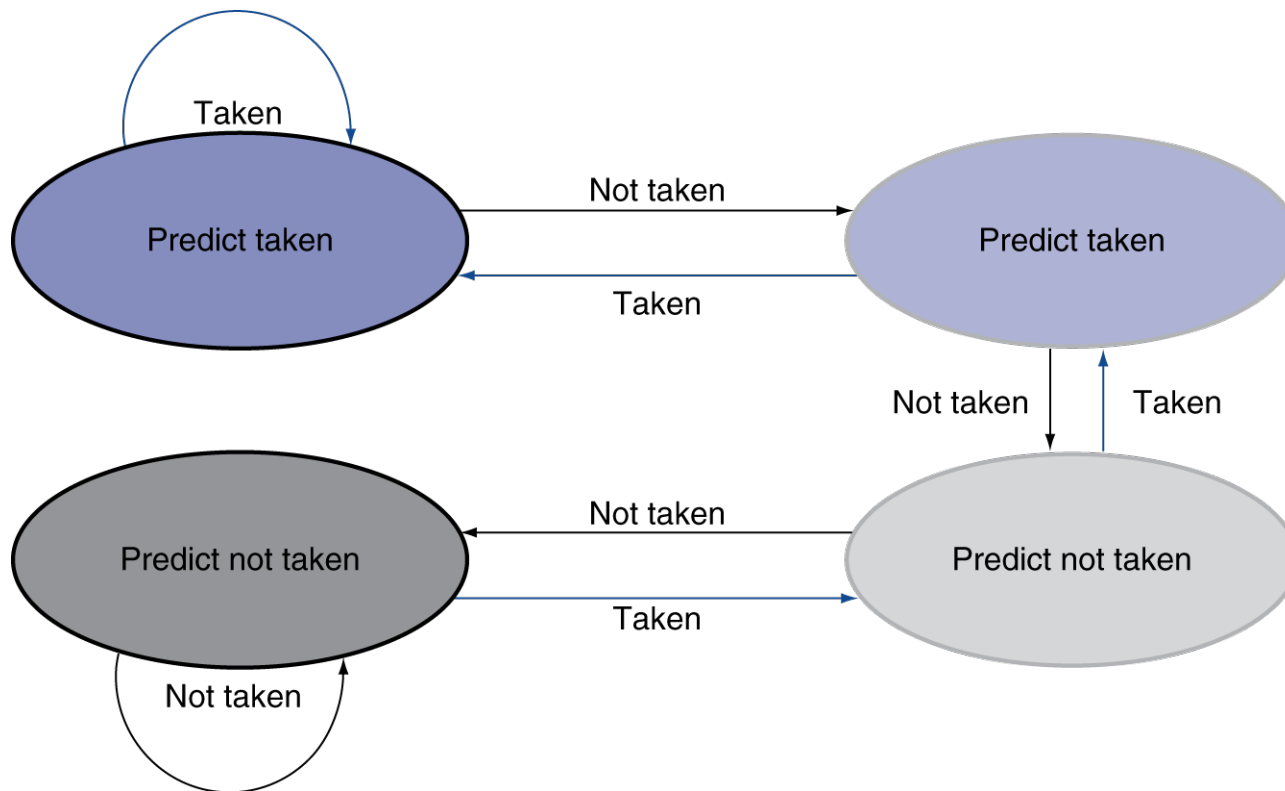
- Inner loop branches mispredicted twice!



- 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

- 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
  - You may think of it as key-value store (dictionary) with branch PC as key and target address as value
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

