

# CSE261: Computer Architecture

## 14. Processor (6): Pipelining #3

Seongil Wi

# HW2

---

2



Due date: **11/26, 11:59PM**

# Quiz 2

---



- This is the last quiz of this semester
- **Date: 11/21 (TUE.), Class time**
- **Scope:**
  - Logic Design Basics
  - Processor (1) ~ Processor (6)
- Brint your own pen!
- T/F problems + 2~4 computation problems

# Where are We?



Situations that prevent starting the next instruction in the next cycle

## Hazard #1: Structural hazard

Conflict for use of a hardware resource

### Solution:

- Stall
- Resource duplication

## Hazard #2: Data hazard

An instruction cannot execute because data is not yet available

### Solution:

- Stall
- Forwarding
- Compiler optimization

## Hazard #3: Control hazard

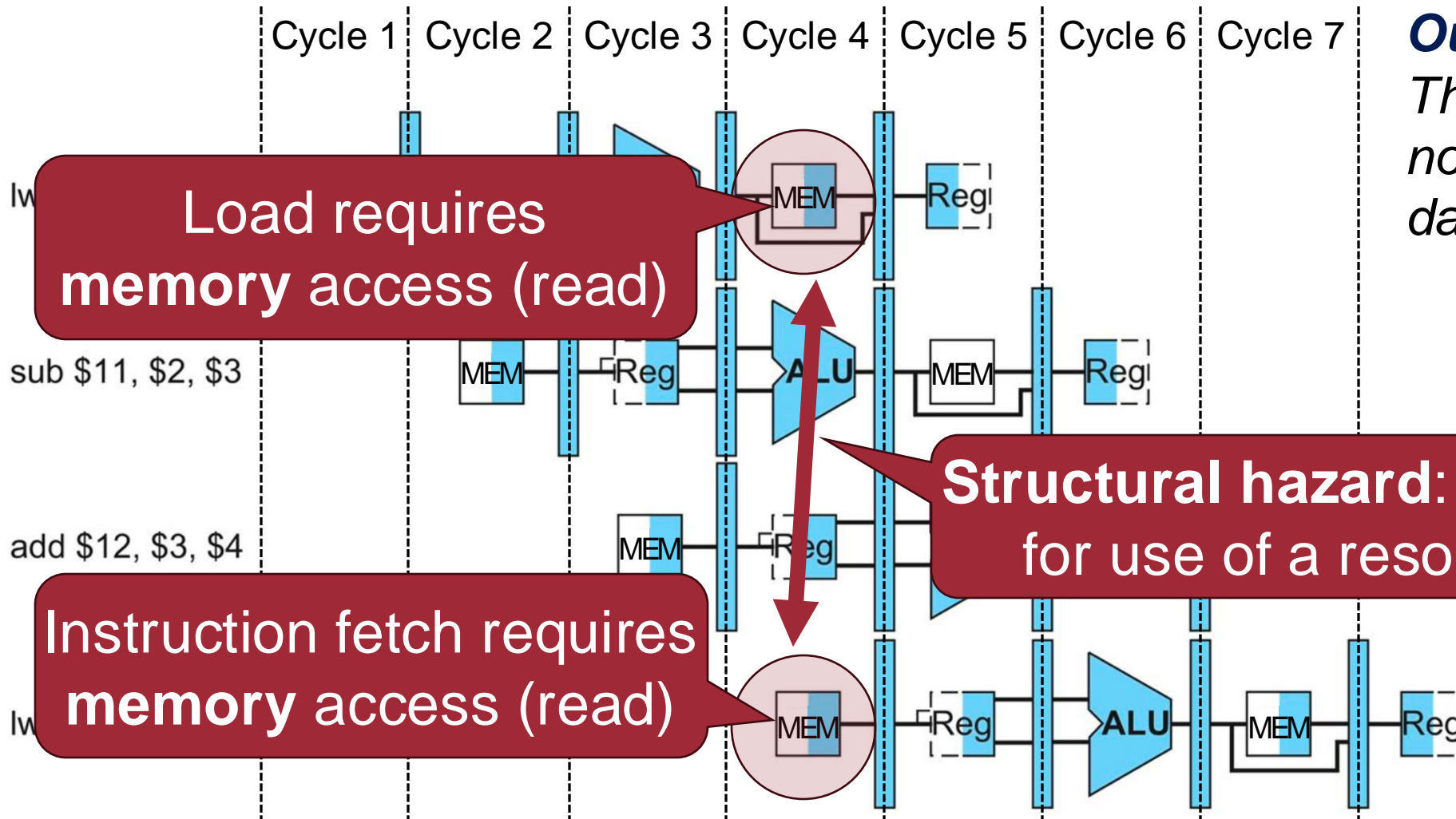
# Recap: Structural Hazard



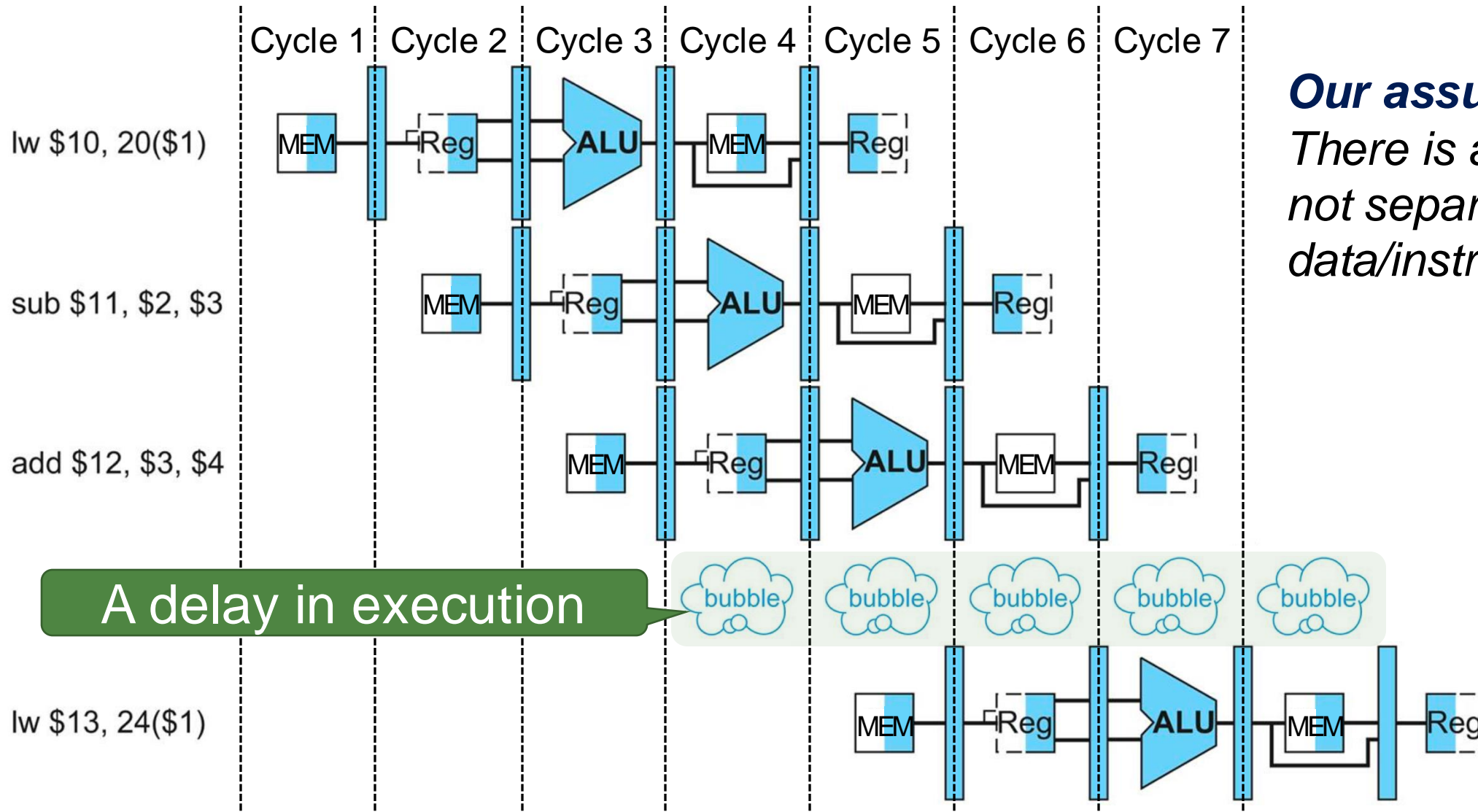
The **hardware** cannot support *the combination of instructions*

***Our assumption:***

*There is a single memory, not separate data/instruction memories*



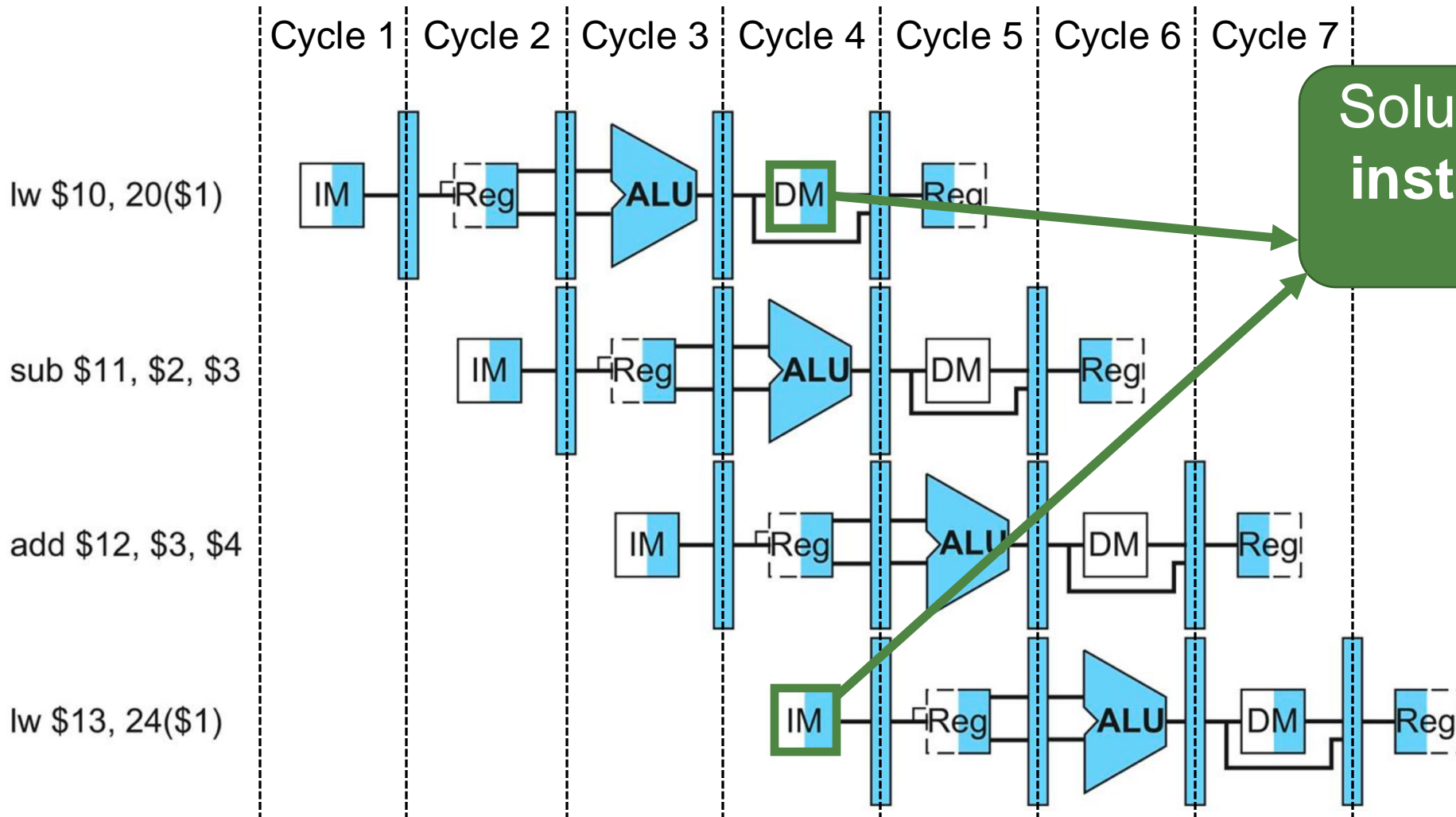
# Recap: Pipeline Stall



***Our assumption:***  
*There is a single memory,  
not separate  
data/instruction memories*

# Recap: Resource Duplication

7

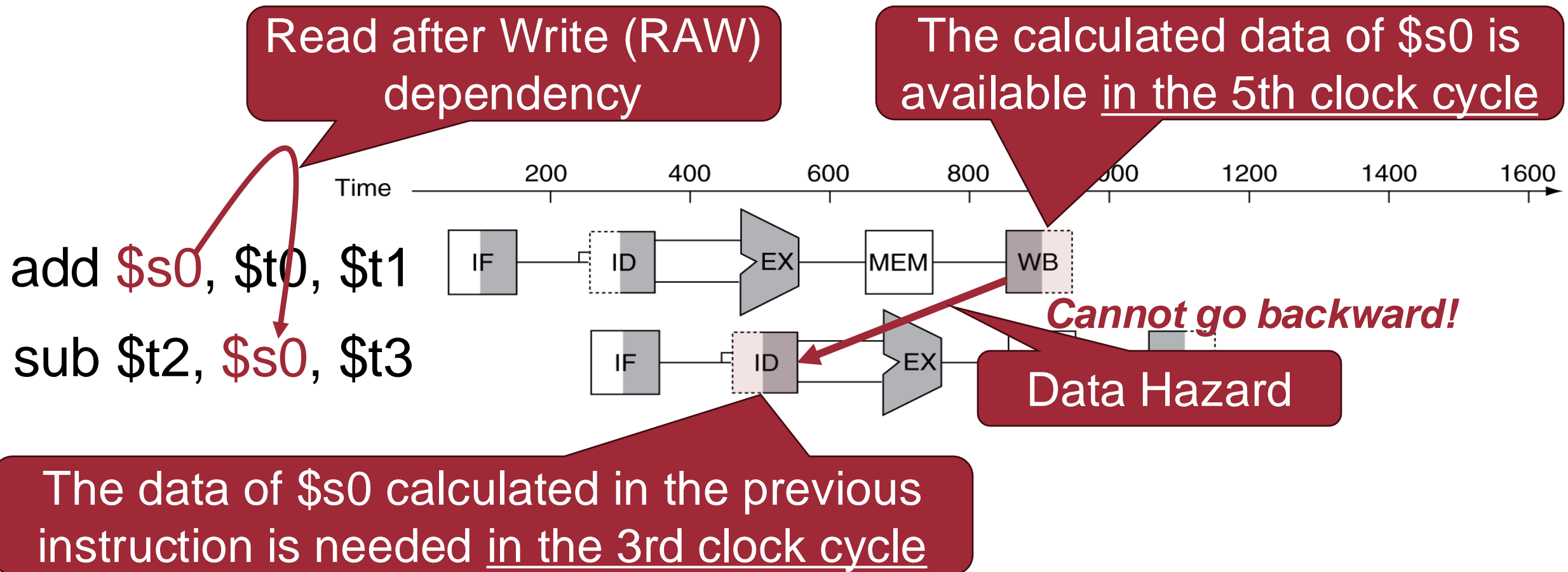


**Solution: use separate  
instruction and data  
memory**

# Recap: Data Hazard



A planned instruction **cannot execute** because data is not yet available





# Recap: Pipeline Stall

9

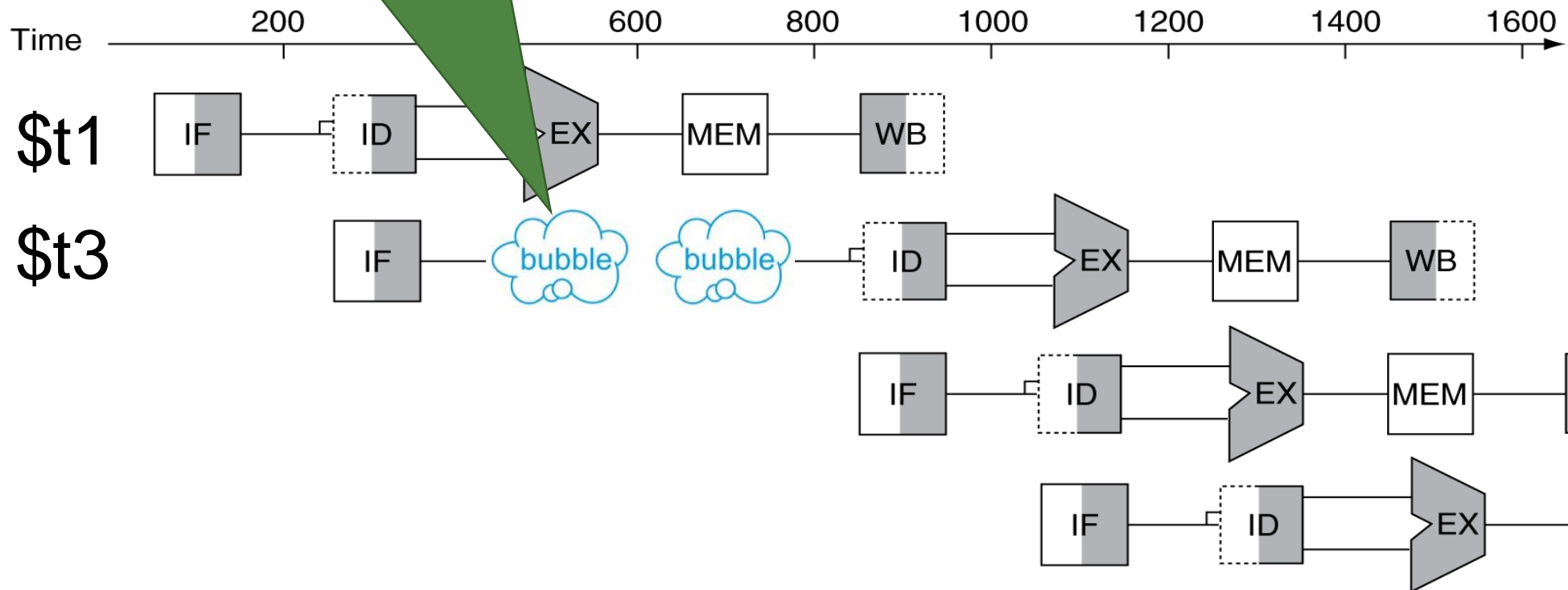
A delay in execution  
(2 clock cycle)

add \$s0, \$t0, \$t1

sub \$t2, \$s0, \$t3

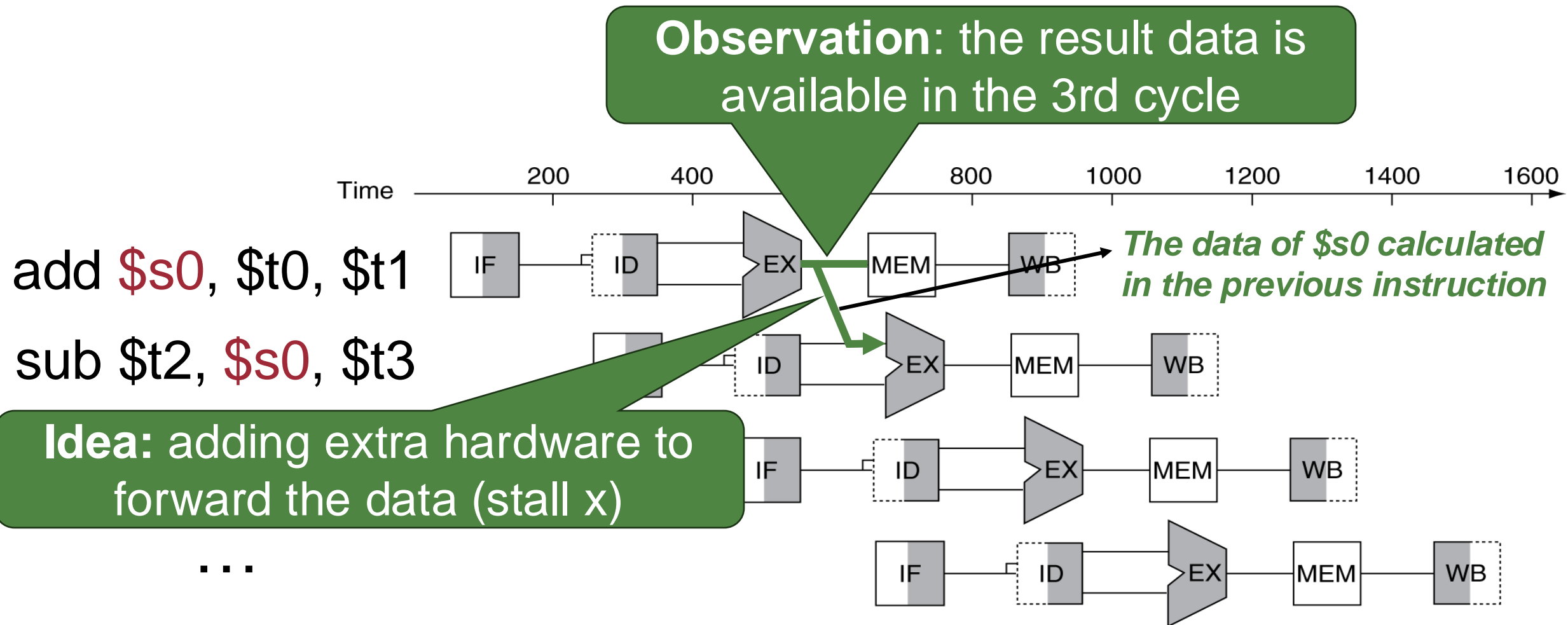
...

...



# Recap: Forwarding

- Use result when it is computed!



# Recap: Compiler Optimization

- Reorder code to avoid use of load result in the next instruction
- C code for  $v[3] = v[0] + v[1]; v[4] = v[0] + v[2];$

```
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)
```

1 stall

1 stall

Idea: code reordering to avoid stalls (by compiler)

*Compiler requires knowledge of the pipeline structure!*

13 cycles

# Pipelining Hazards Summary



Situations that prevent starting the next instruction in the next cycle

## Hazard #1: Structural hazard

Conflict for use of a hardware resource

### Solution:

- Stall
- Resource duplication

## Hazard #2: Data hazard

An instruction cannot execute because data is not yet available

### Solution:

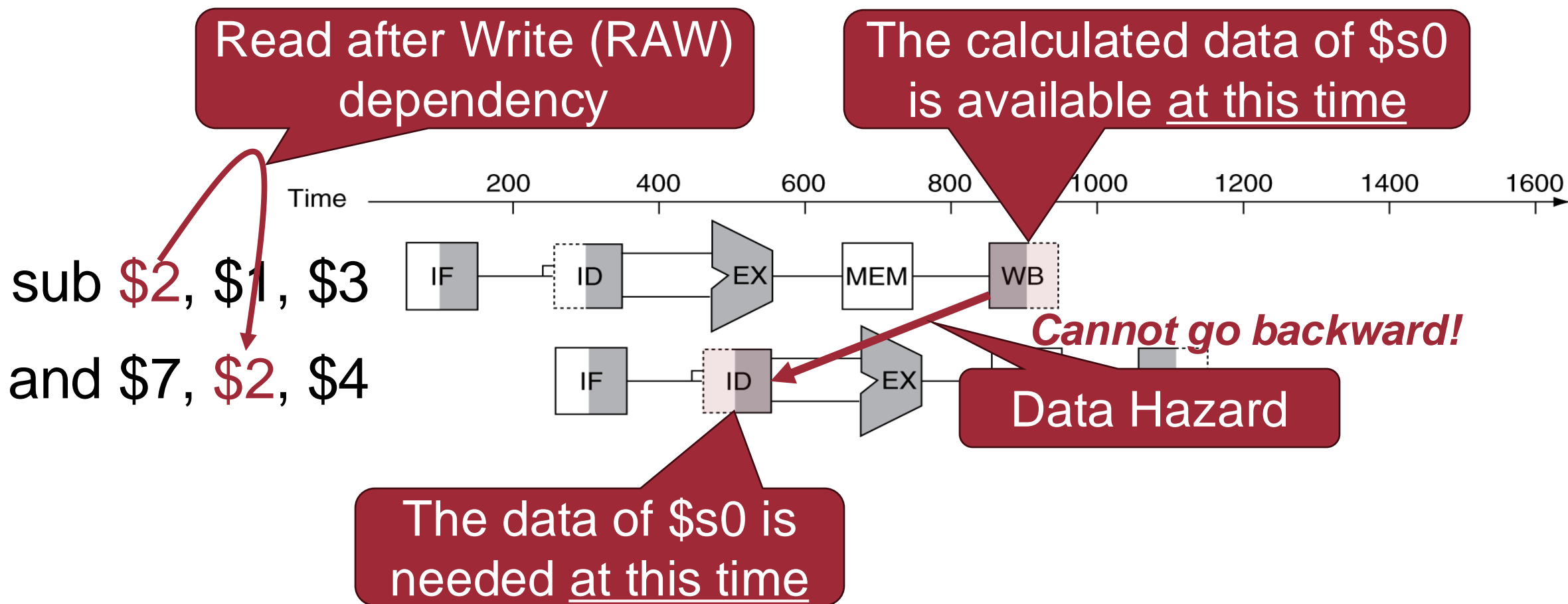
- Stall
- Forwarding
- Compiler optimization

Let's look at the hardware details

## Hazard #3: Control hazard

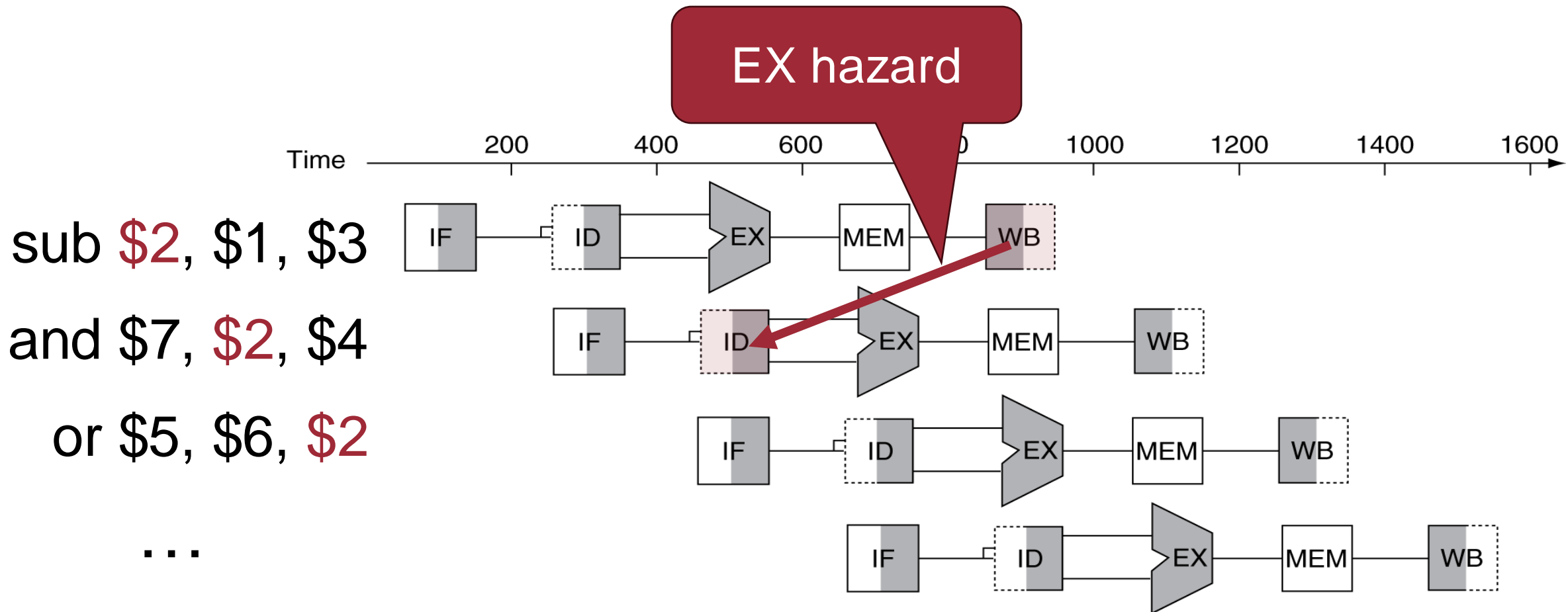
# Data Hazard

A planned instruction **cannot execute** because data is not yet available



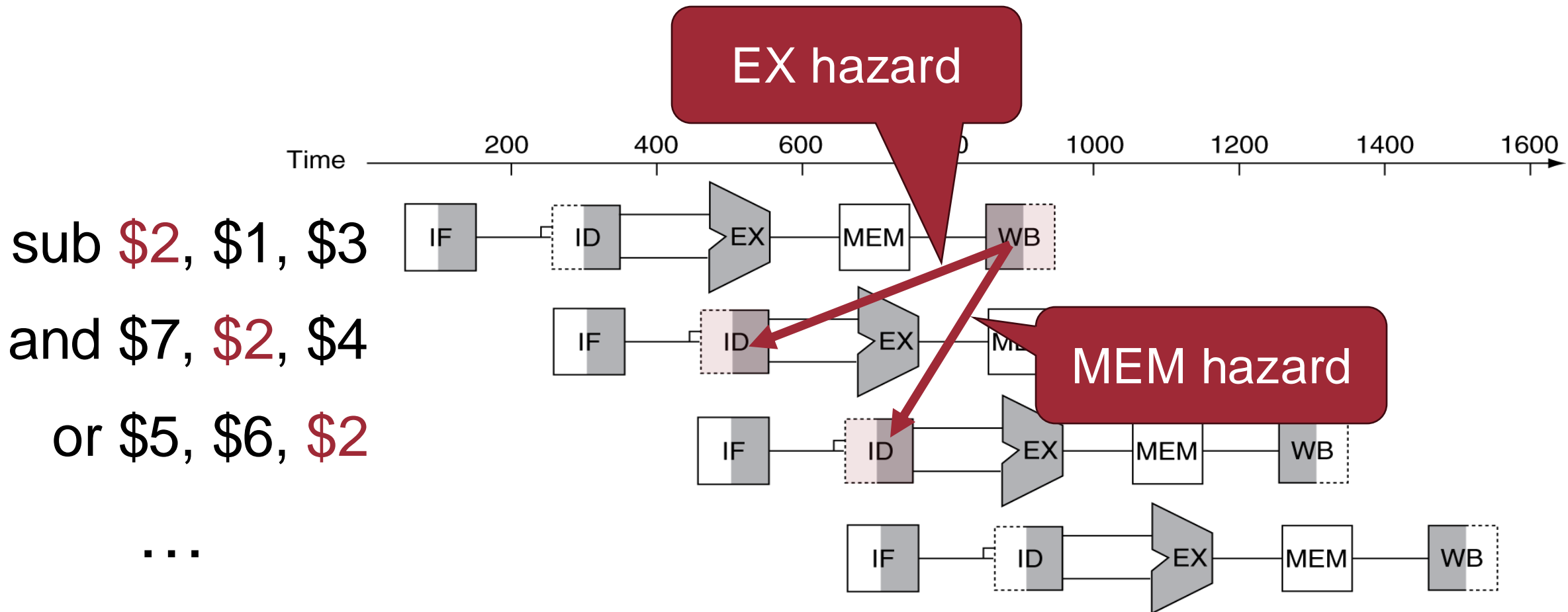
# Data Hazard #1: EX Hazard

A planned instruction **cannot execute** because data is not yet available



# Data Hazard #2: MEM Hazard

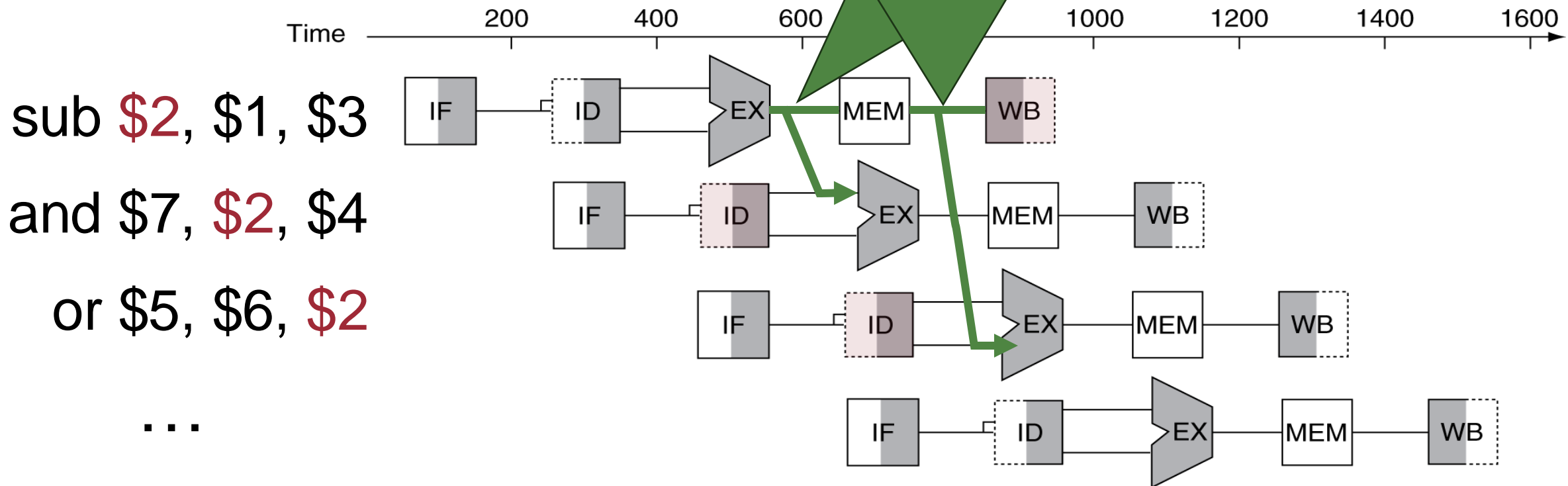
A planned instruction **cannot execute** because data is not yet available



# Solution: Forwarding

- Use result when it is computed!

Idea: adding extra hardware to forward the data





# Today's Topic

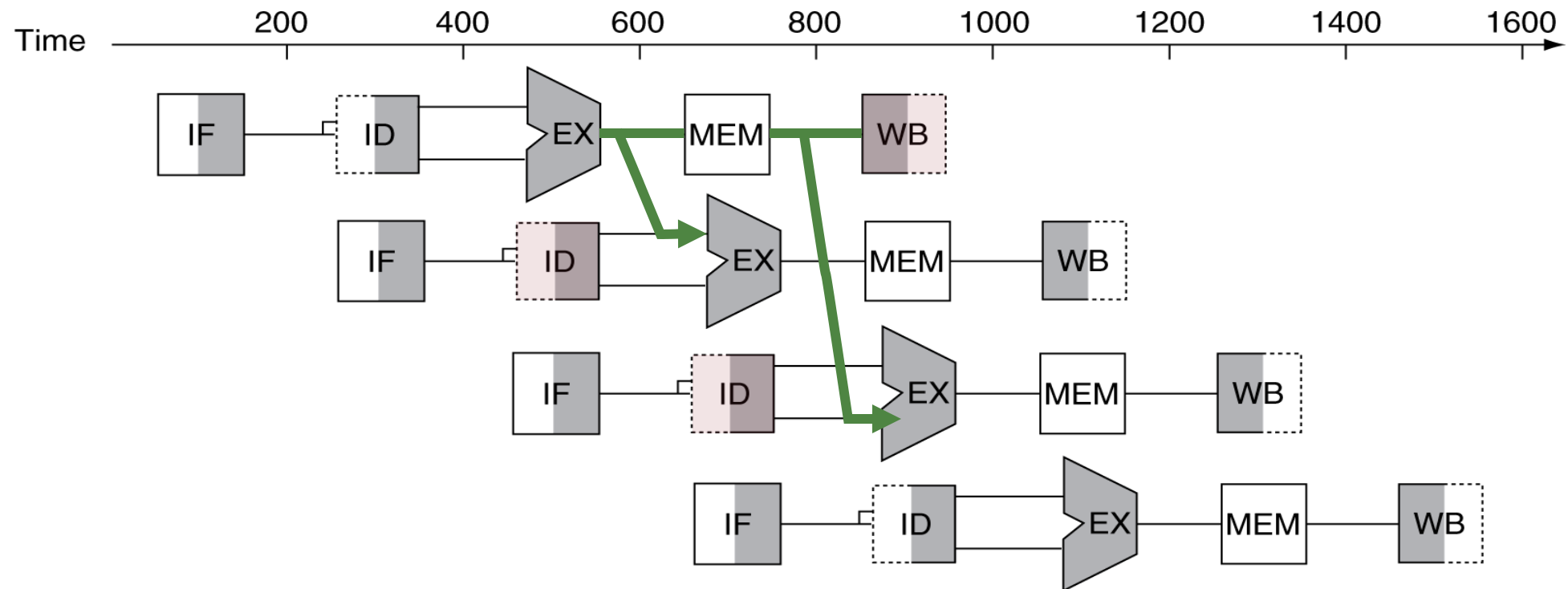
---



- How should the **hardware** be modified to support forwarding?
- How should the **hardware** be modified to detect data hazards?
- How should the **hardware** be modified to support stalls?

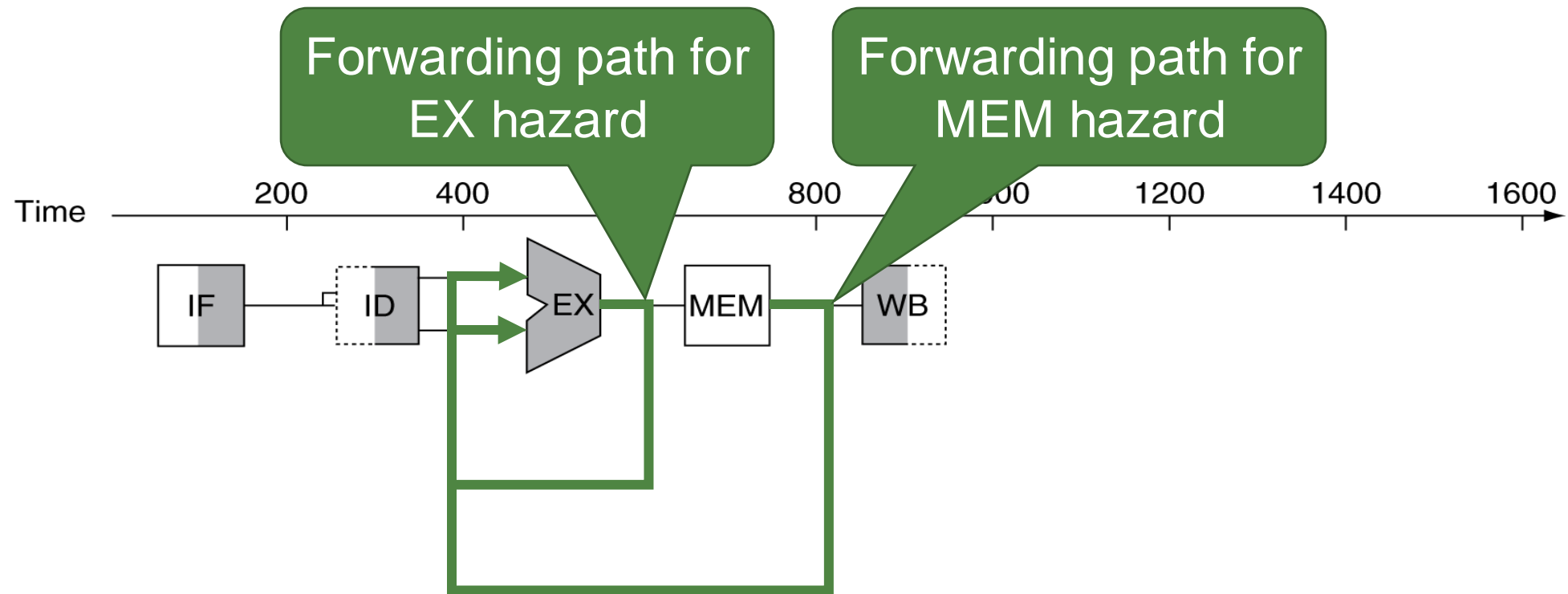
# Forwarding Paths Overview

18

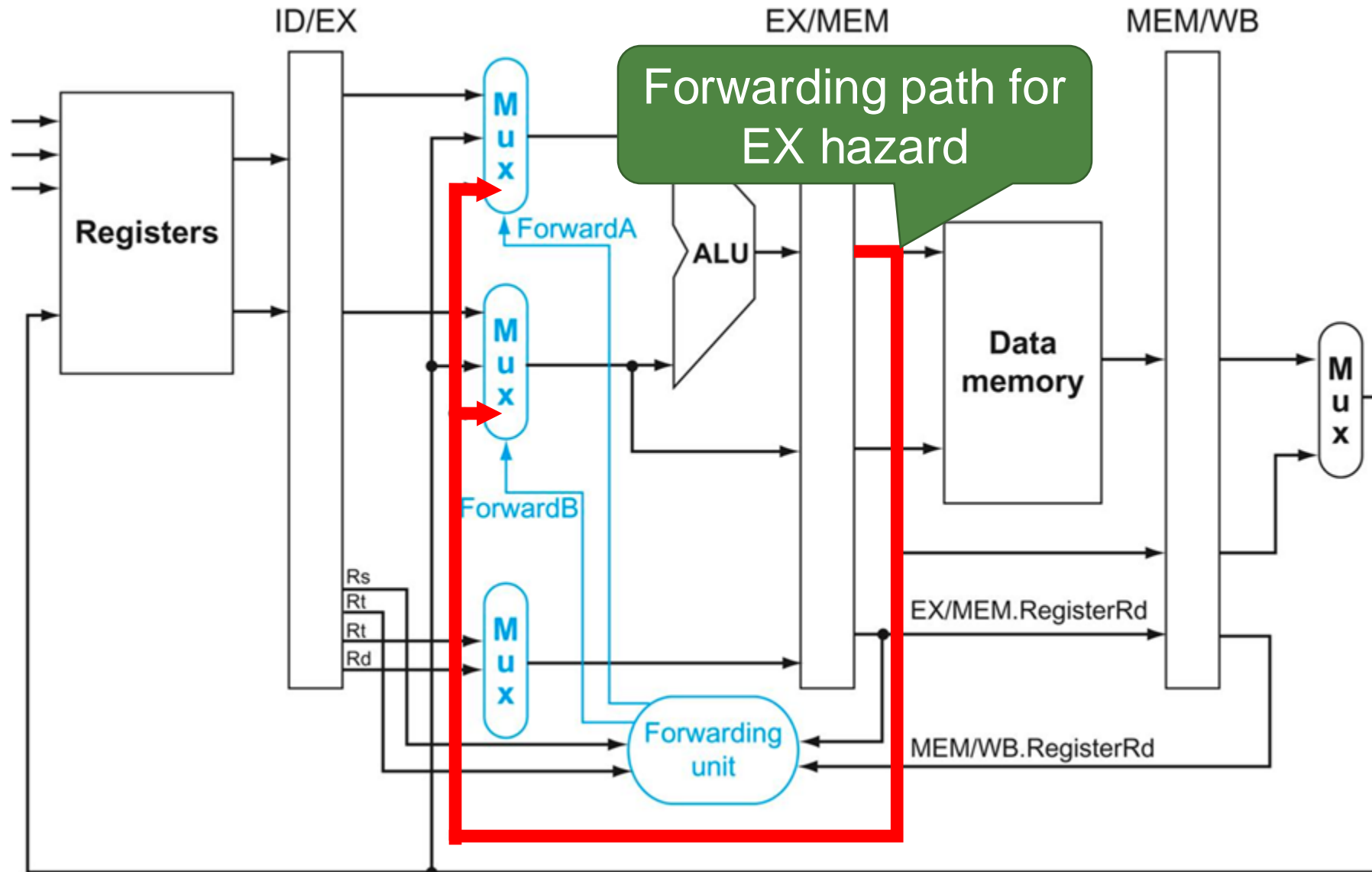


# Forwarding Paths Overview

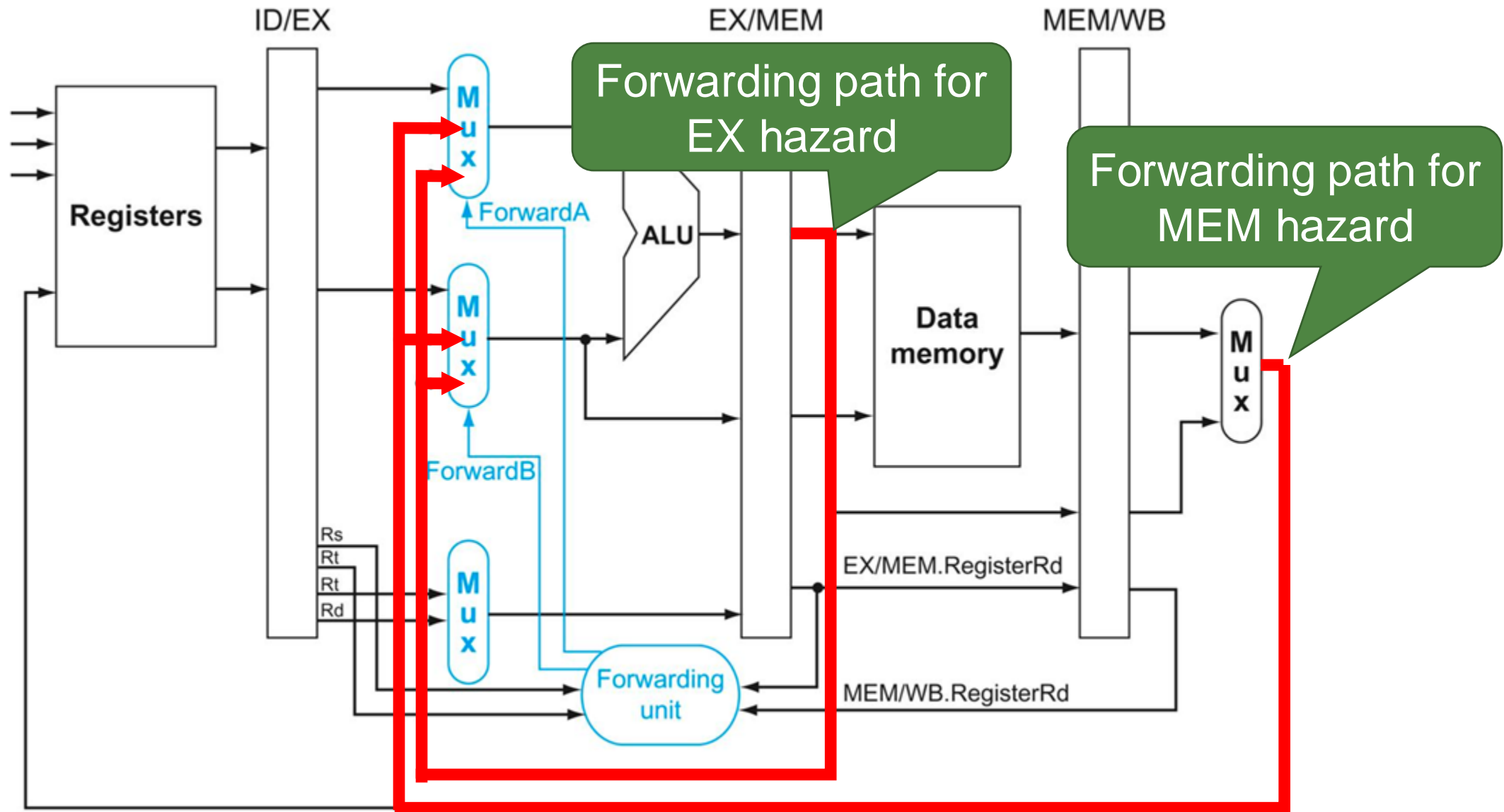
19



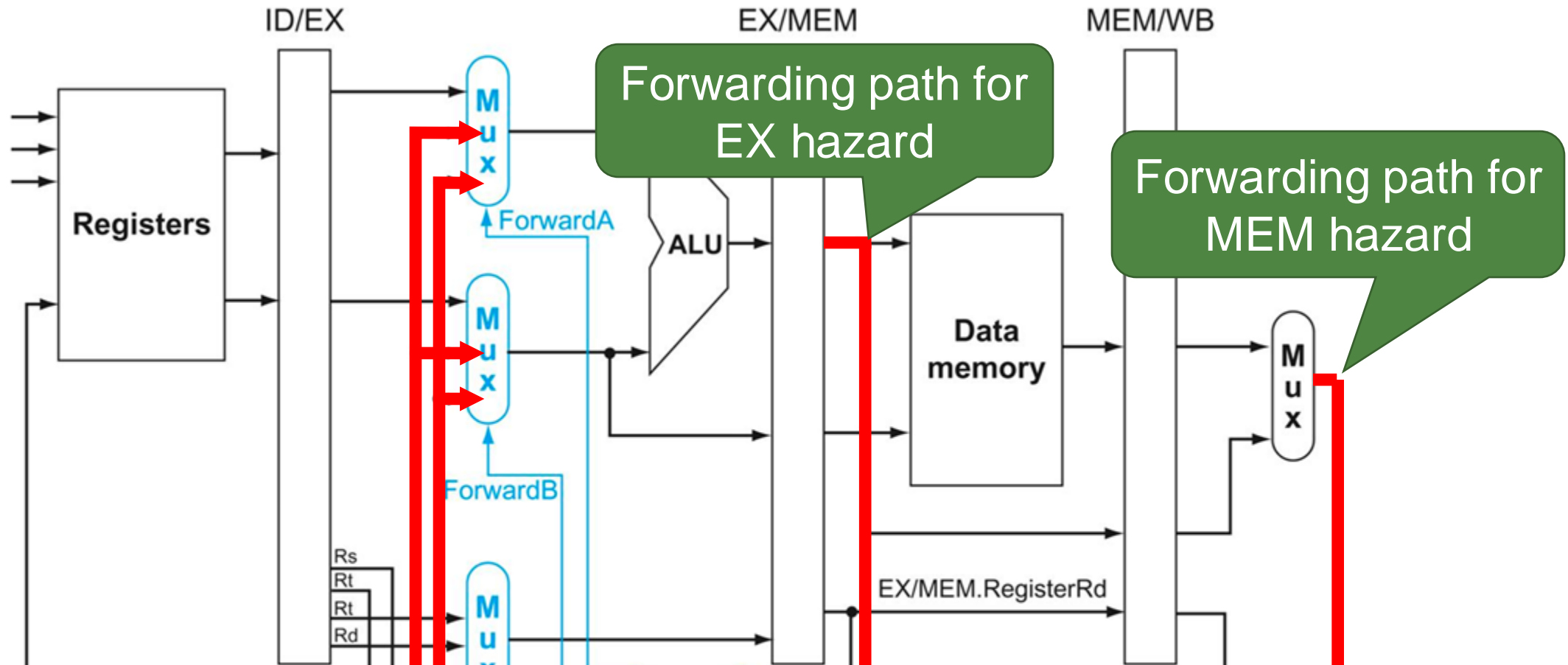
# Forwarding Paths: Details



# Forwarding Paths: Details

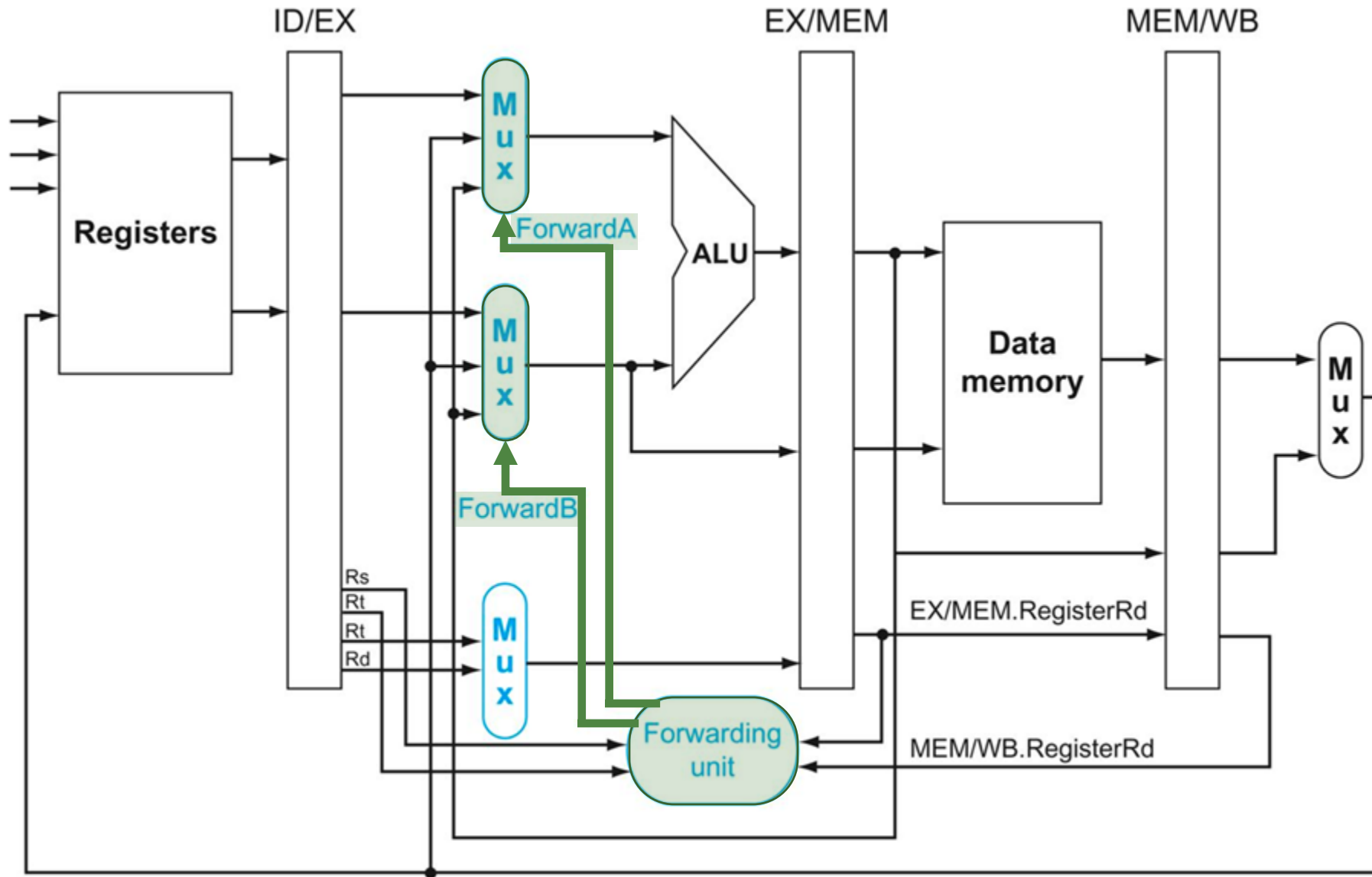


# Forwarding Paths: Details



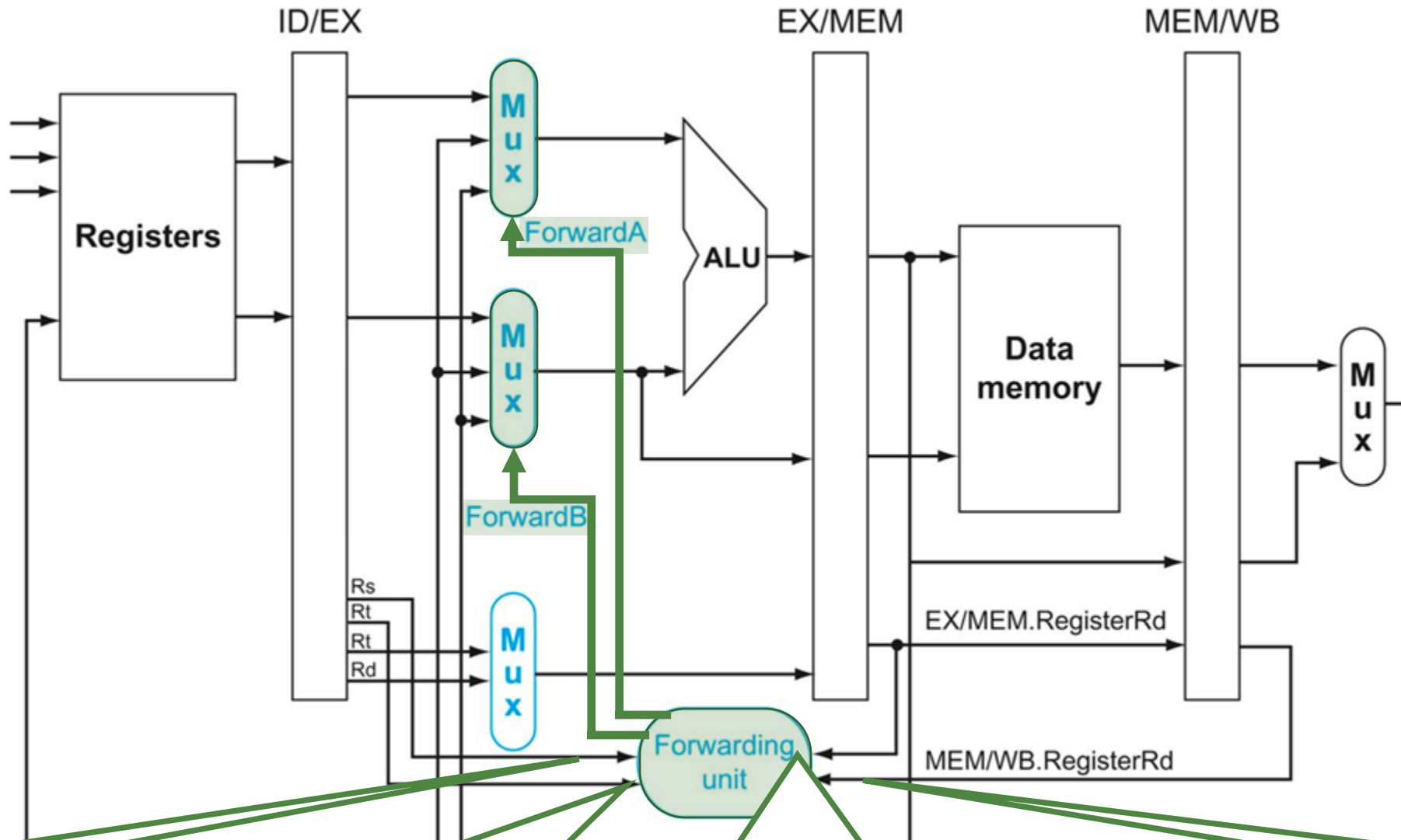
How to detecting the need to forward?

# Detecting the Need to Forward: Basic Idea



# Detecting the Need to Forward: Basic Idea

24



ID/EX.RegisterRs

ID/EX.RegisterRt

EX/MEM.RegisterRd

MEM/WB.RegisterRd

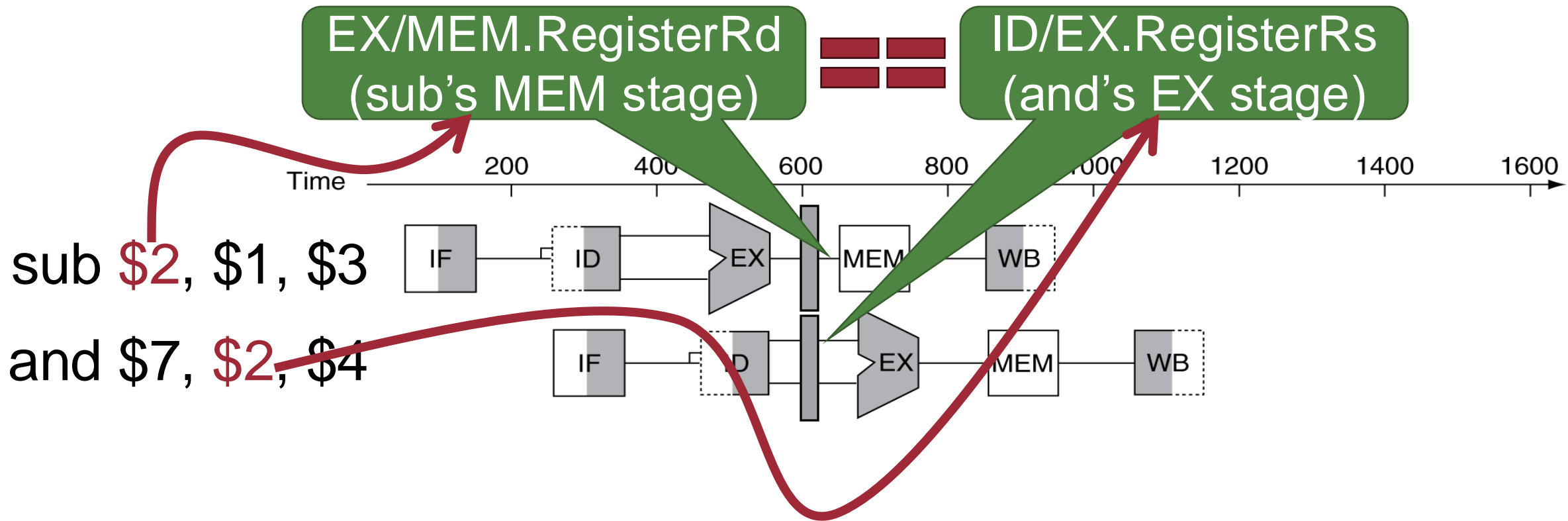


# Detecting the Need to Forward: Basic Idea

25

*EX hazard detection!*

*First operand  $\leftarrow$  EX/MEM.RegisterRd*

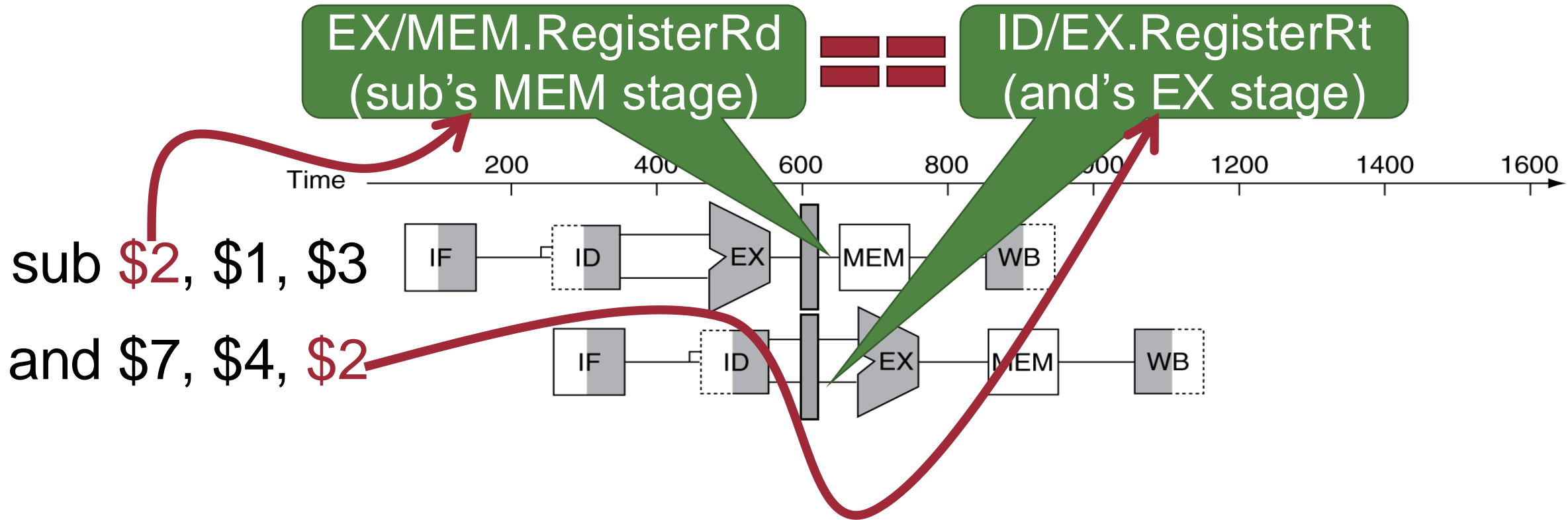


# Detecting the Need to Forward: Basic Idea

26

*EX hazard detection!*

*Second operand  $\leftarrow$  EX/MEM.RegisterRd*

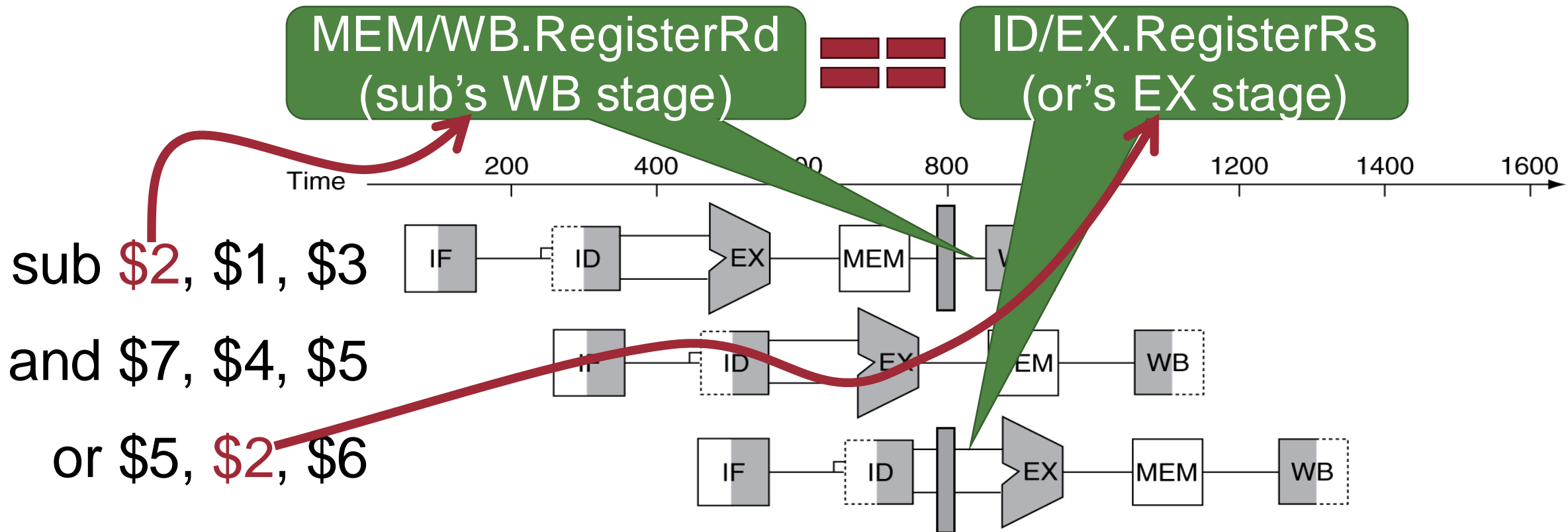


# Detecting the Need to Forward: Basic Idea

27

*MEM hazard detection!*

*First operand  $\leftarrow$  MEM/WB.RegisterRd*

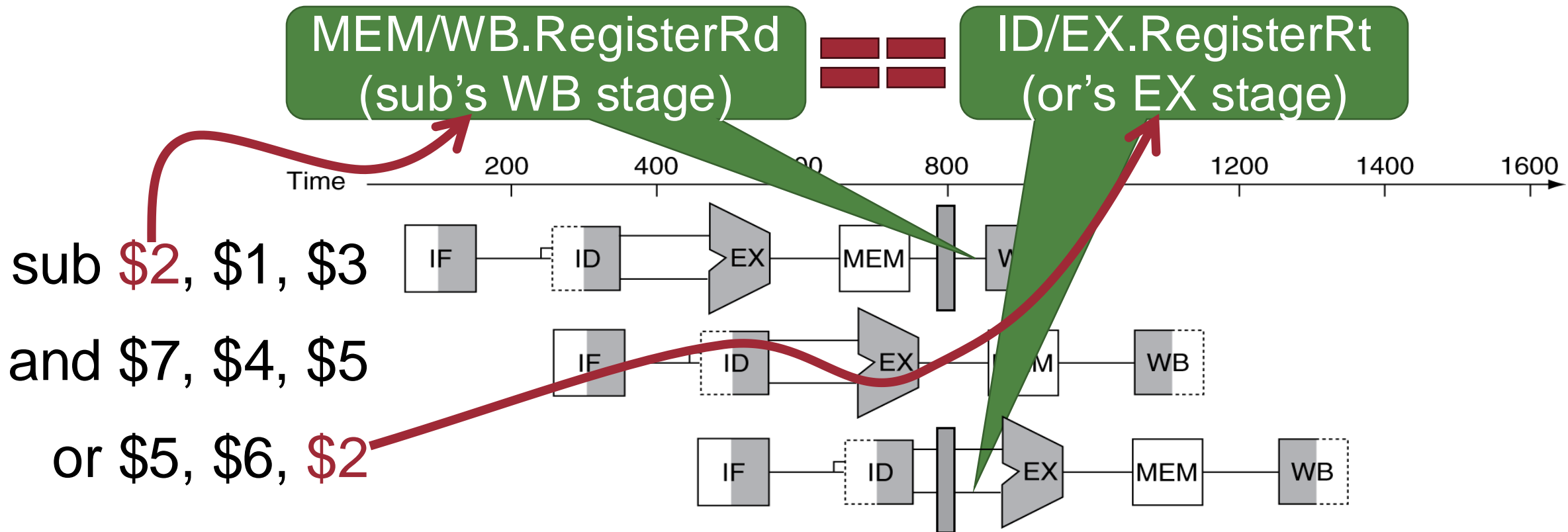


# Detecting the Need to Forward: Basic Idea

28

*MEM hazard detection!*

*Second operand  $\leftarrow$  MEM/WB.RegisterRd*



# Additional Conditions

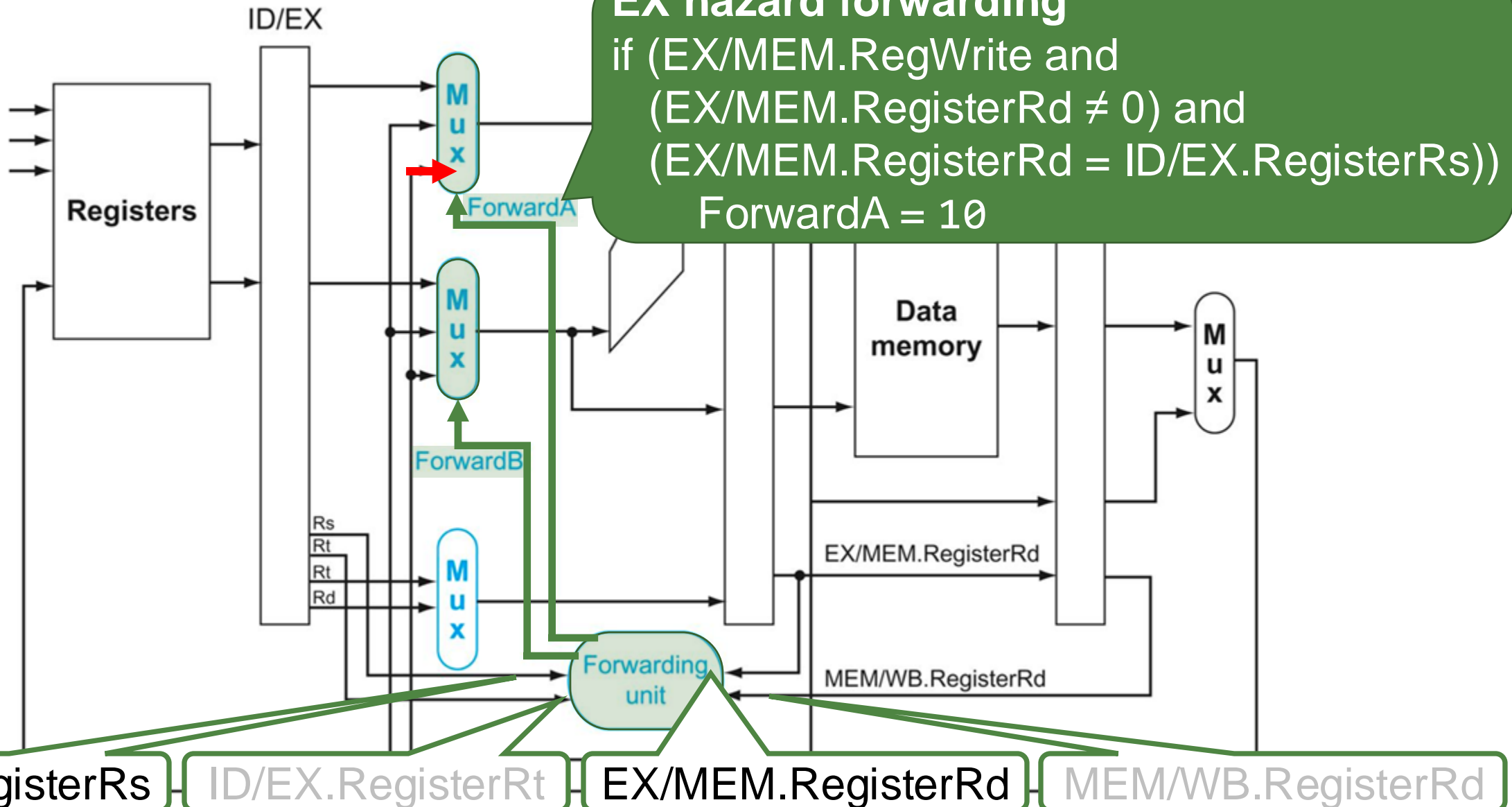
---



- But only if **forwarding instruction will write 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

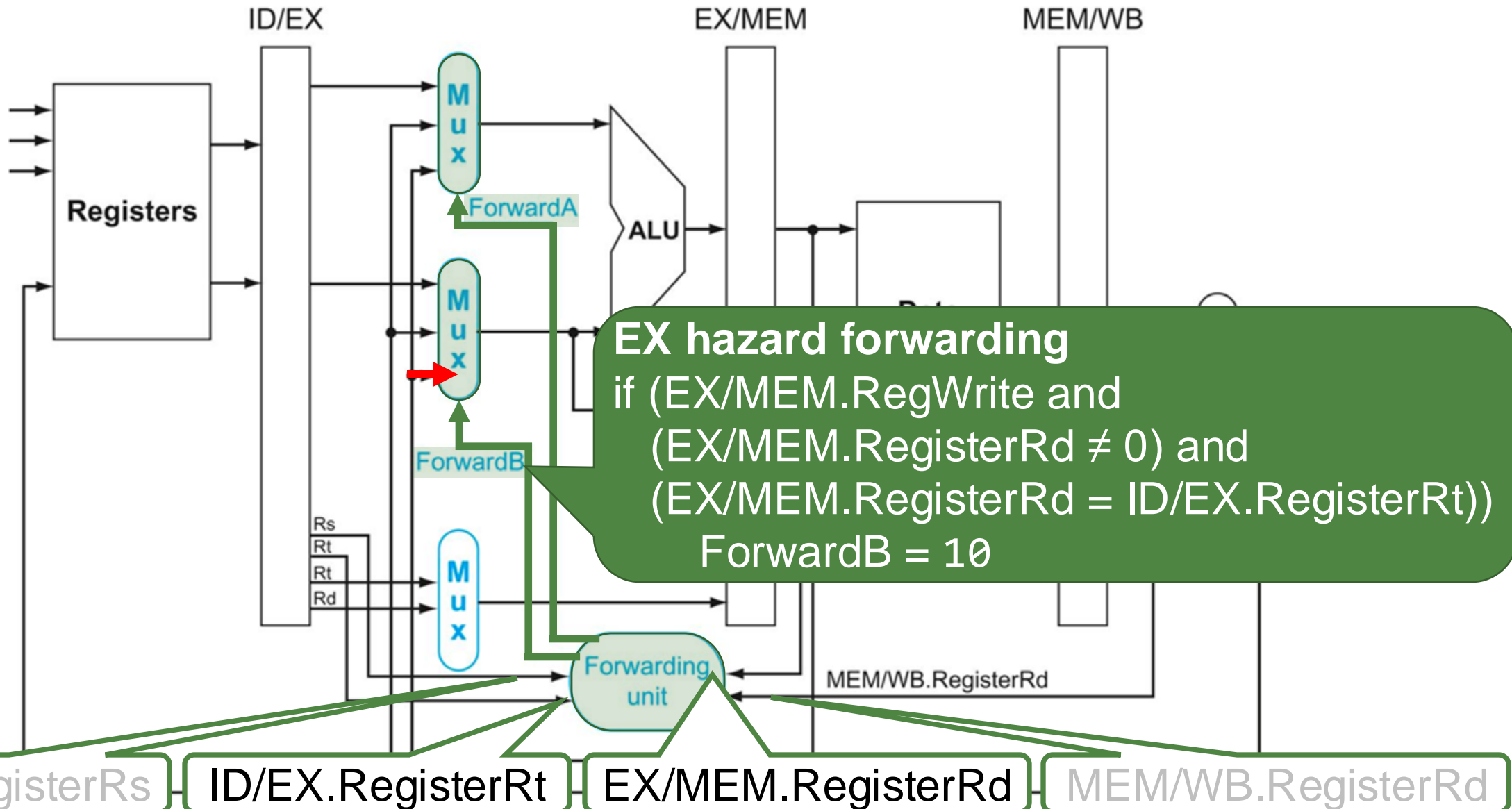
# Detecting the Need to Forward: Details

30



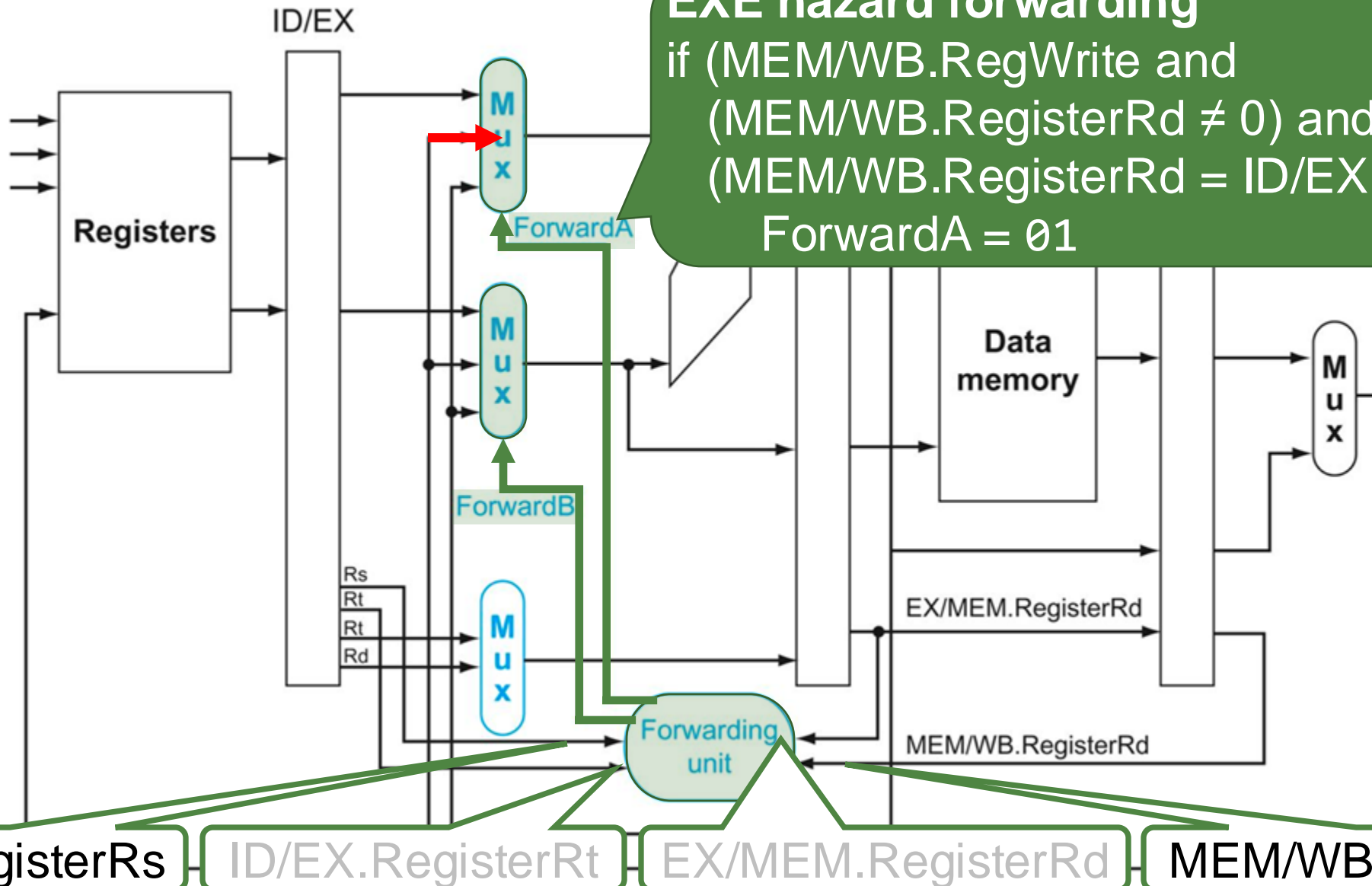
# Detecting the Need to Forward: Details

31



# Detecting the Need to Forward: Details

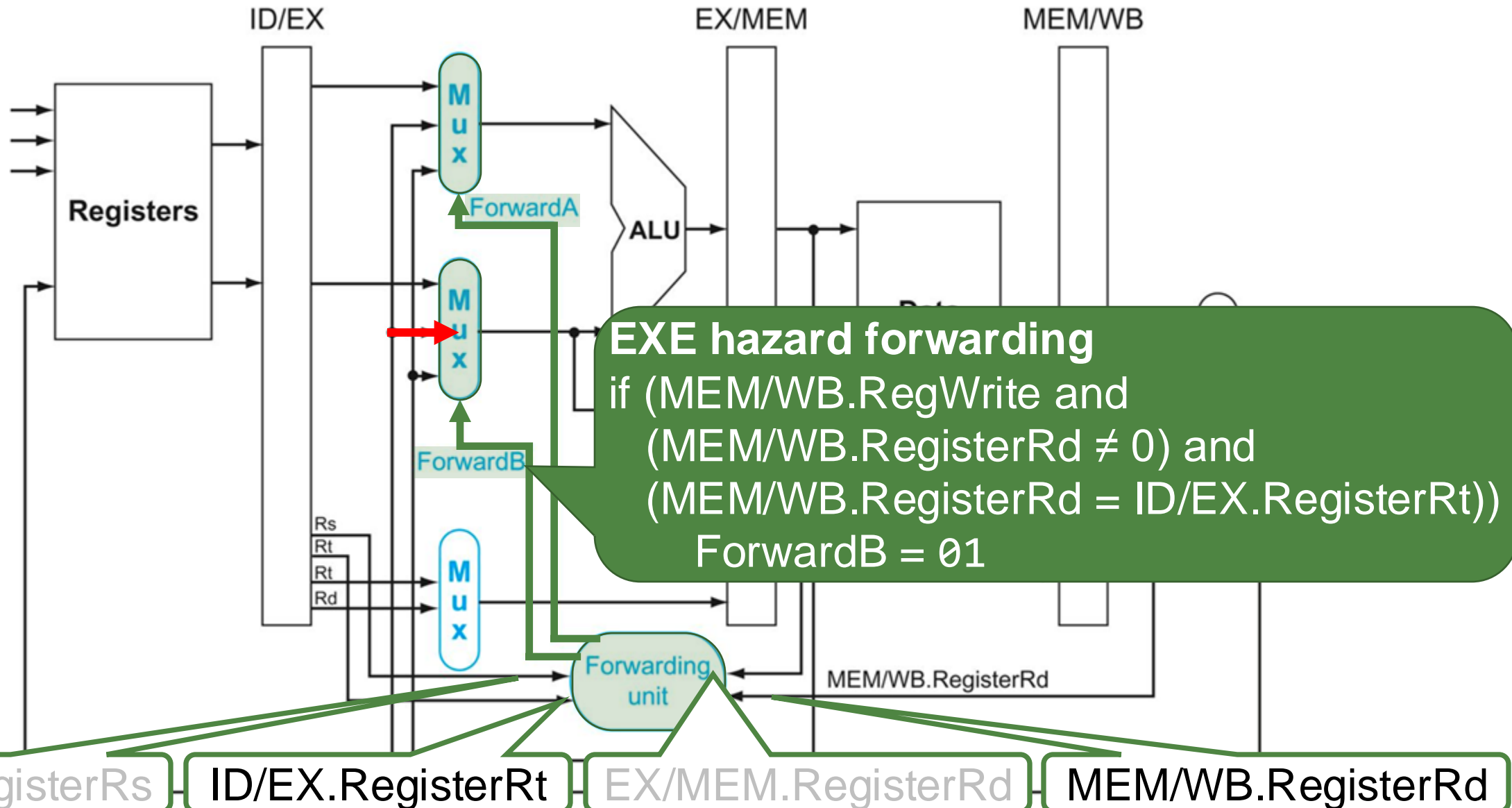
32





# Detecting the Need to Forward: Details

33



# Summary: Forwarding Conditions



- **EX hazard:** Forward EX/MEM value to EX stage
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))  
**ForwardA = 10**
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))  
**ForwardB = 10**
- **MEM hazard:** Forward MEM/WB value to EX stage
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd  $\neq$  0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))  
**ForwardA = 01**
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd  $\neq$  0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))  
**ForwardB = 01**

# Double Data Hazard

35



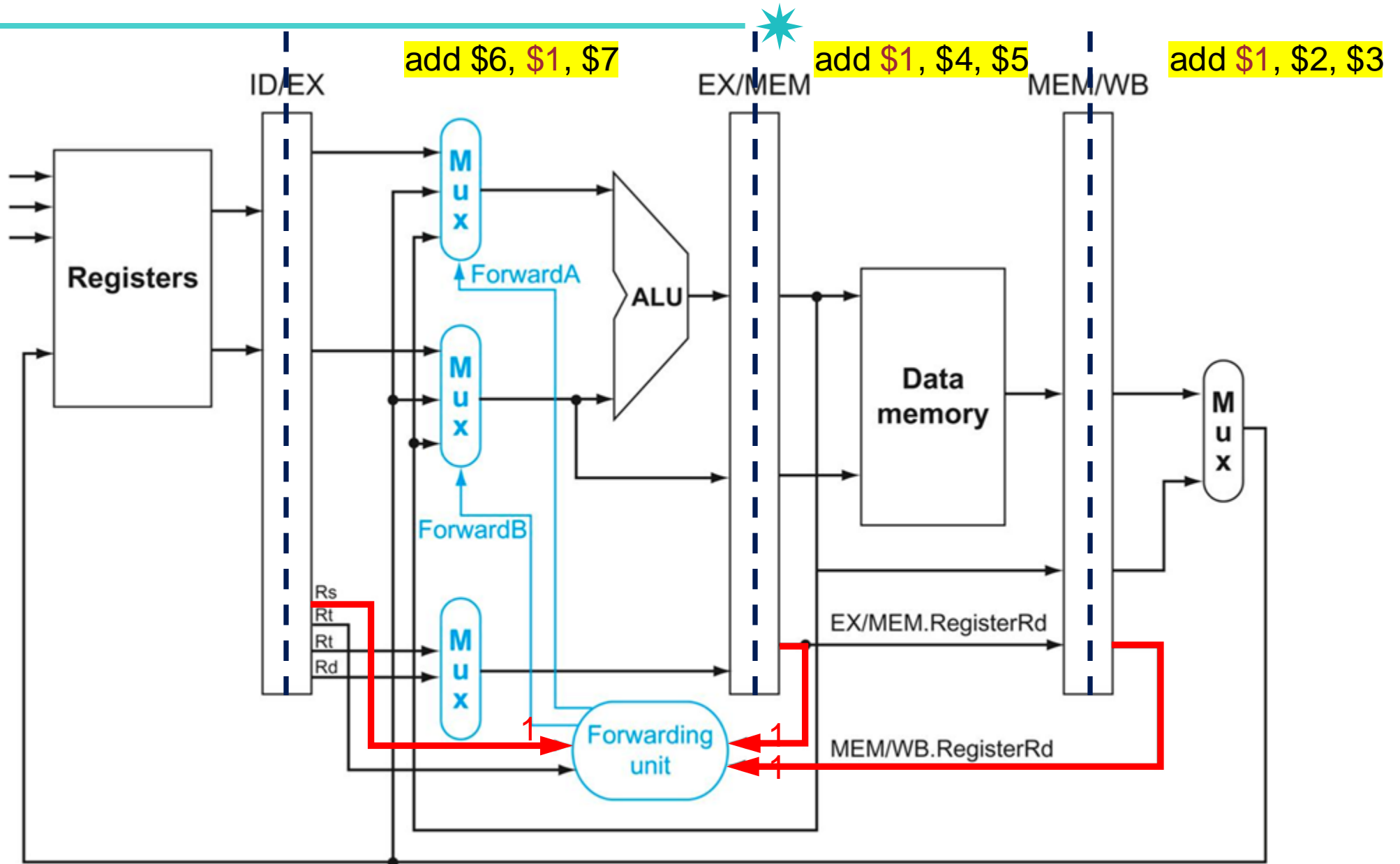
add \$1, \$2, \$3  
add \$1, \$4, \$5  
add \$6, \$1, \$7



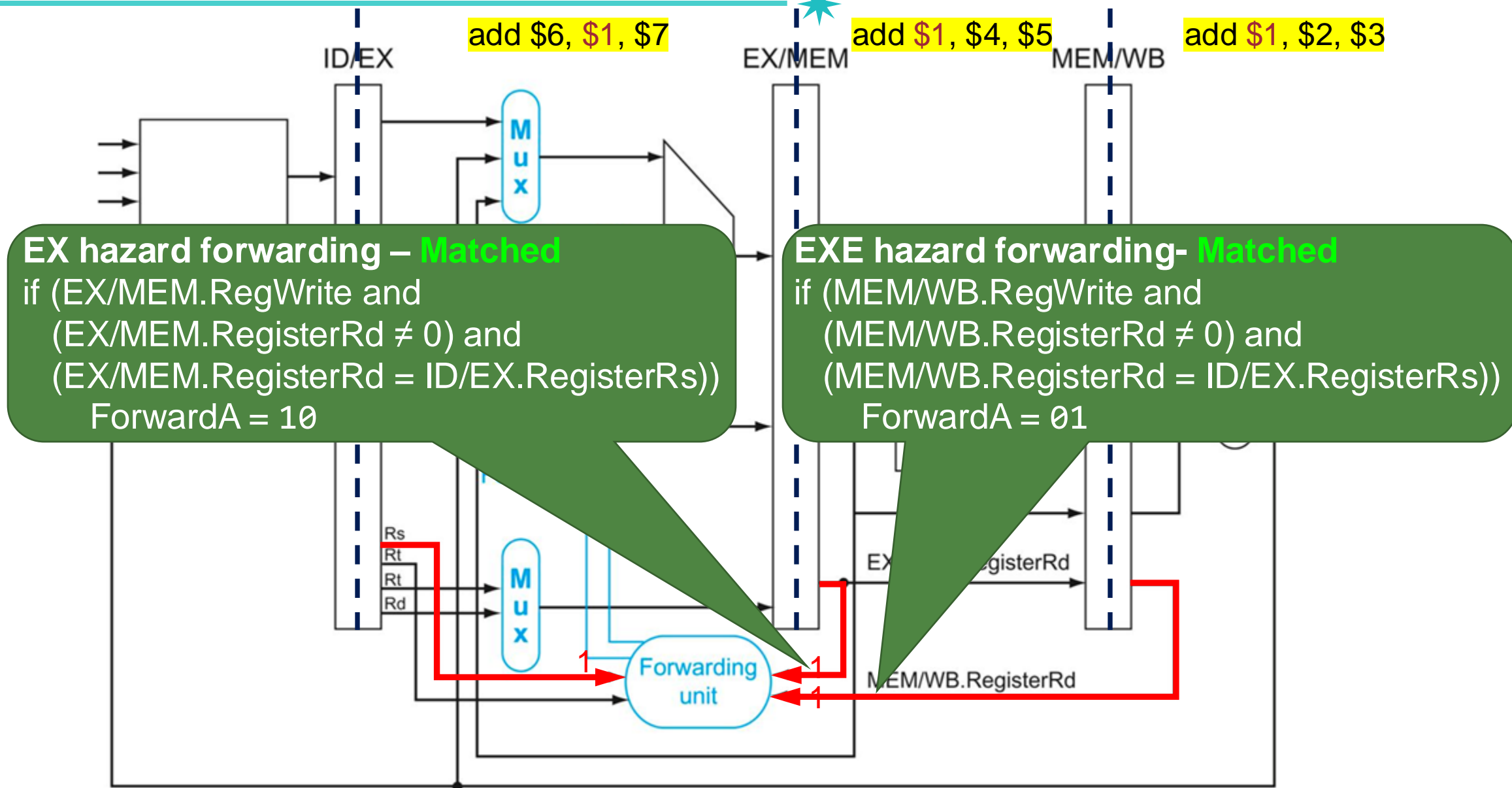
*Any problem?*

# Double Data Hazard

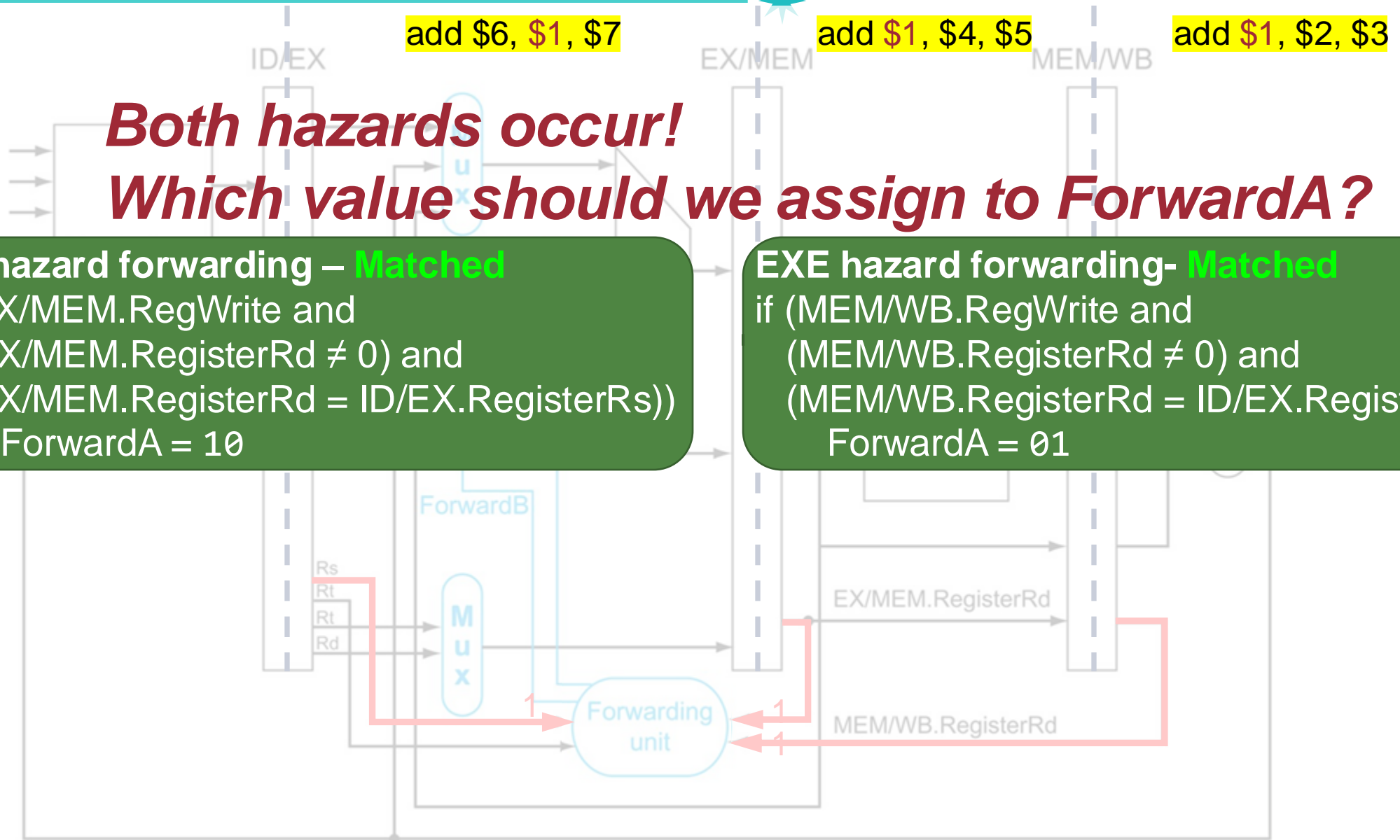
36



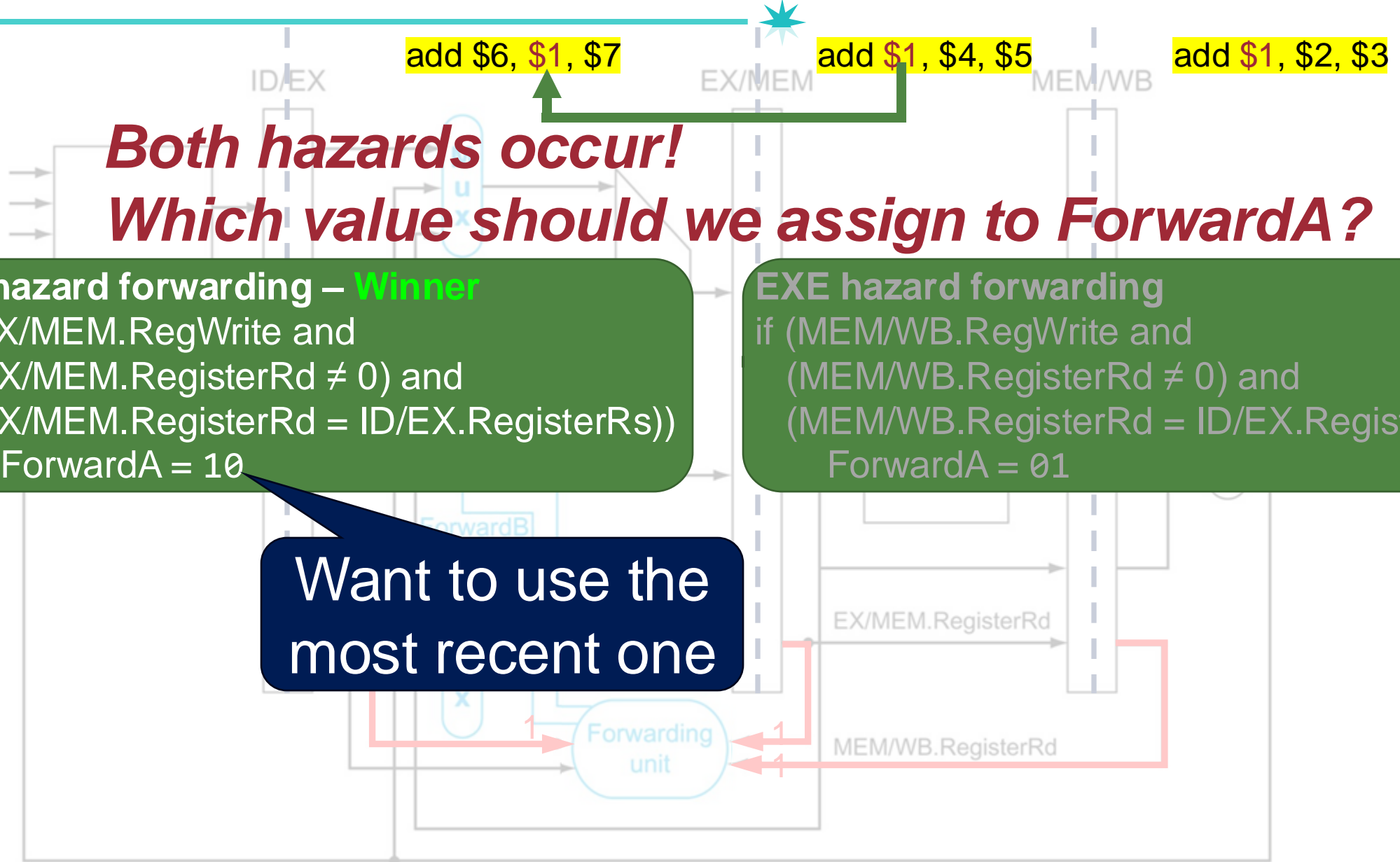
# Double Data Hazard: Problem



# Double Data Hazard: Problem



# Double Data Hazard: Problem



# Revise the MEM Hazard Condition



- Only forward if EX hazard condition isn't true

- 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 = ID/EX.RegisterRs))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

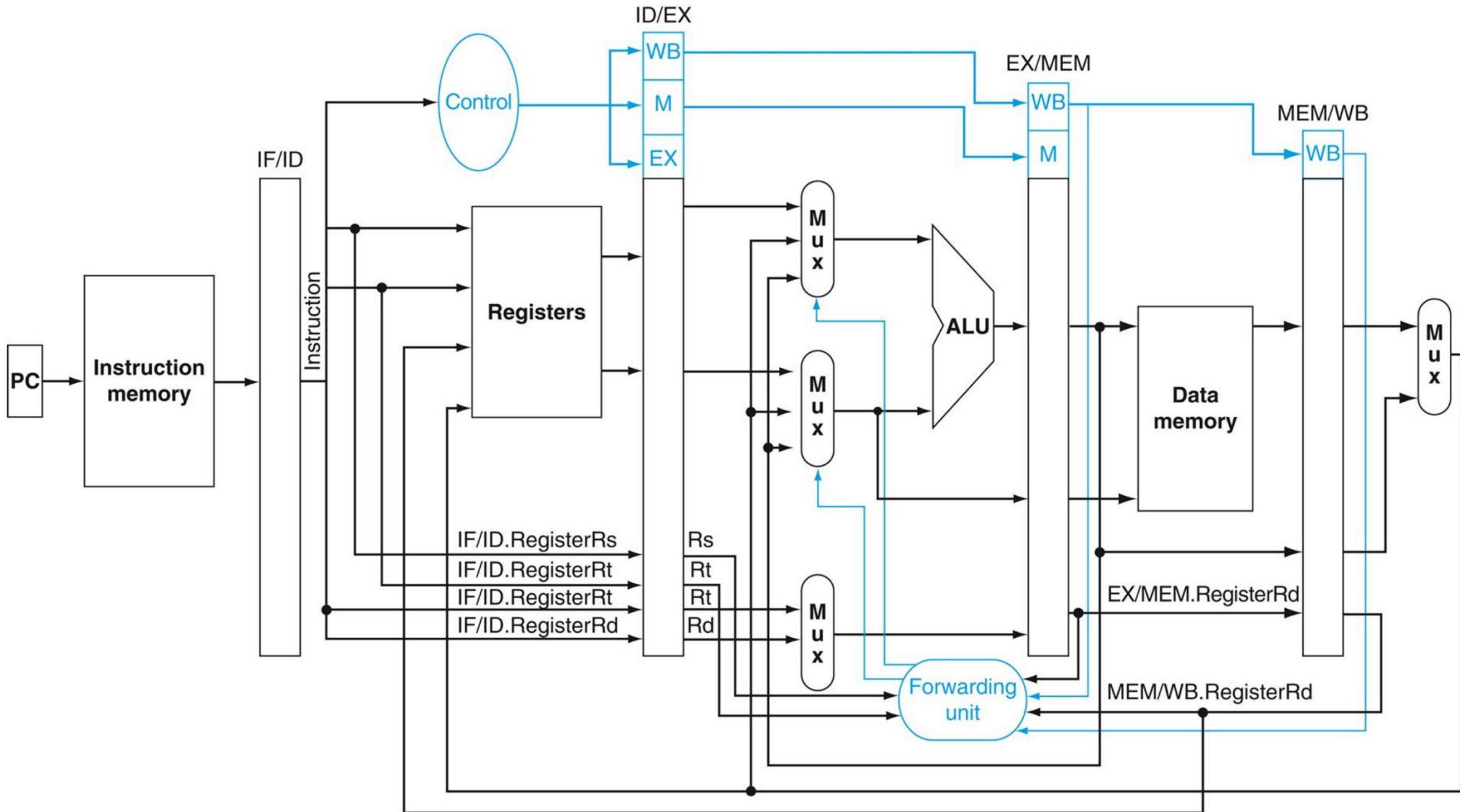
ForwardA = 01

- 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 = ID/EX.RegisterRt))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

ForwardB = 01



# (Current View) Datapath with Forwarding <sup>41</sup>



# Recall: Three Types of Data Hazard

---



1. EX Hazard
2. MEM Hazard
3. **Load-Use Hazard**

# Load-Use Data Hazard with Forwarding

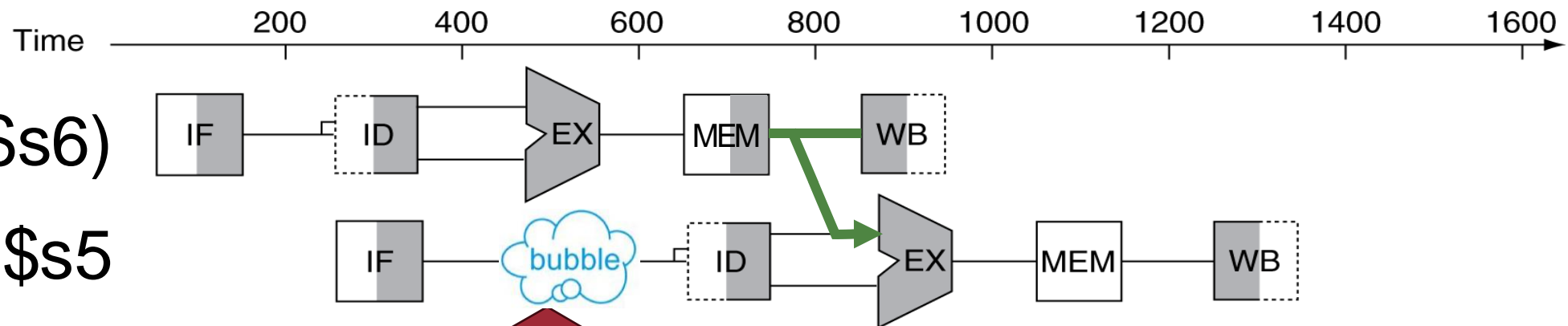
43

Load

lw **\$s1**, 32(\$s6)

sub \$s4, **\$s1**, \$s5

Use



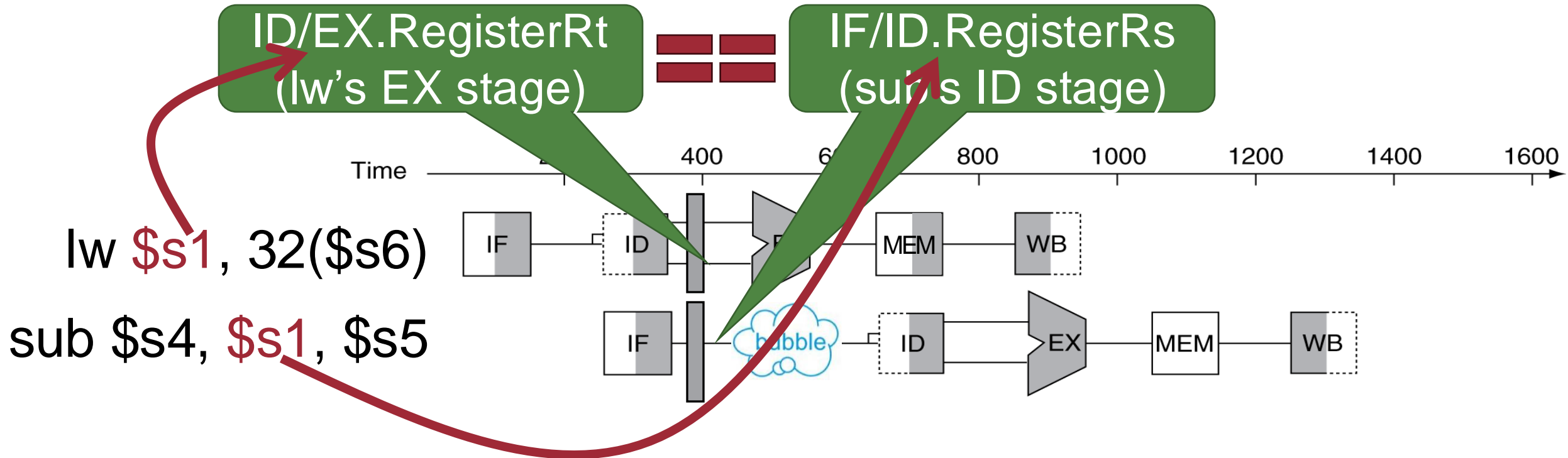
**lw still causes a hazard:** we need a stall even with forwarding when a load tries to use the data

# Load-Use Data Hazard Detection: Basic Idea

44

*Load-Use hazard detection!*

→ *Insert bubble for one clock cycle*

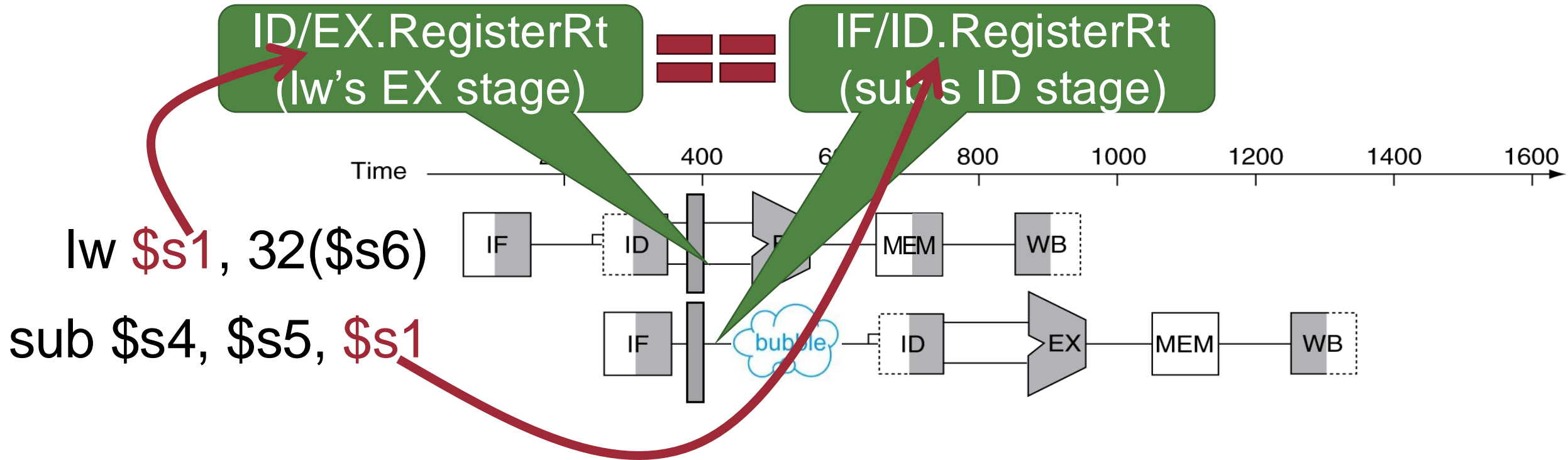


# Load-Use Data Hazard Detection: Basic Idea

45

*Load-Use hazard detection!*

→ *Insert bubble for one clock cycle*



# Additional Condition

---



- But only if **the instruction the ID/EX stage is a load instruction**
  - ID/EX.MemRead = 1

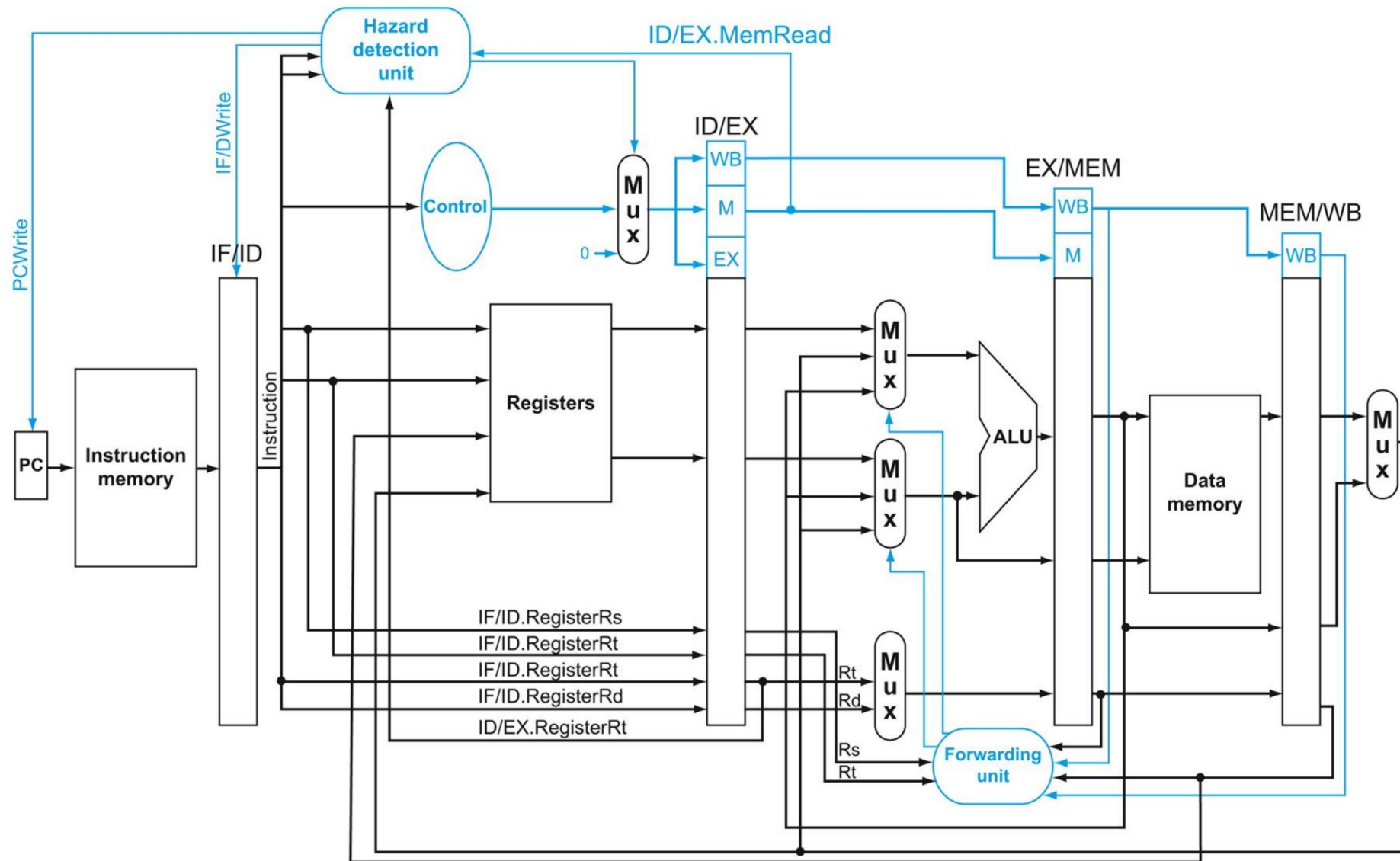
# Load-Use Data Hazard Detection: Basic Idea

47

if ID/EX.MemRead and  
((ID/EX.RegisterRt = IF/ID.RegisterRs) or  
(ID/EX.RegisterRt = IF/ID.RegisterRt))  
Insert bubble for one clock cycle (Stall)

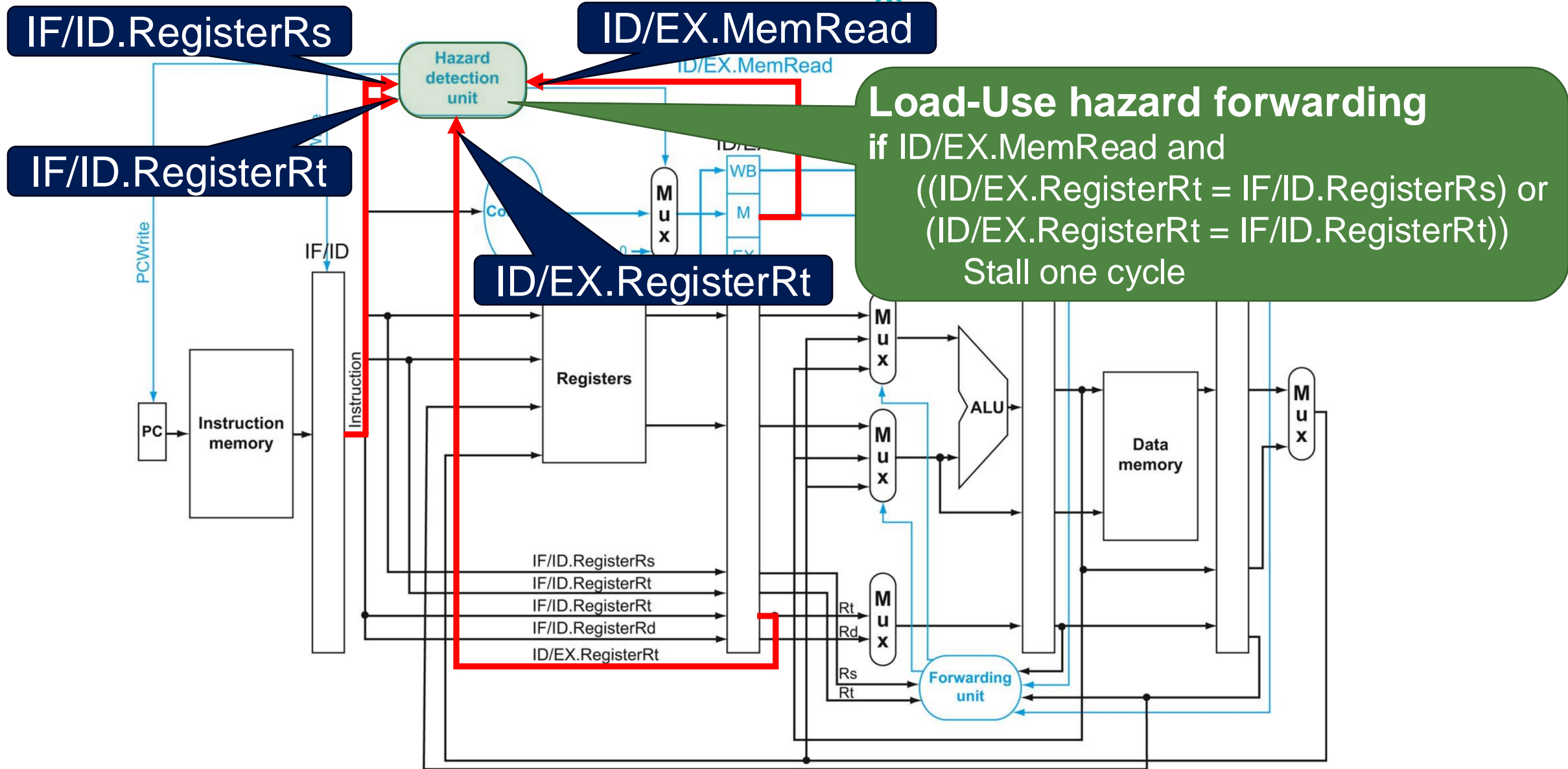
# Load-Use Data Hazard Detection: Details

48





# Load-Use Data Hazard Detection: Details 49



# Load-Use Data Hazard Detection: Basic Idea

50

if ID/EX.MemRead and  
((ID/EX.RegisterRt = IF/ID.RegisterRs) or  
(ID/EX.RegisterRt = IF/ID.RegisterRt))  
Insert bubble for one clock cycle (Stall)

How to stall the  
pipeline?

# How to Stall the Pipeline?

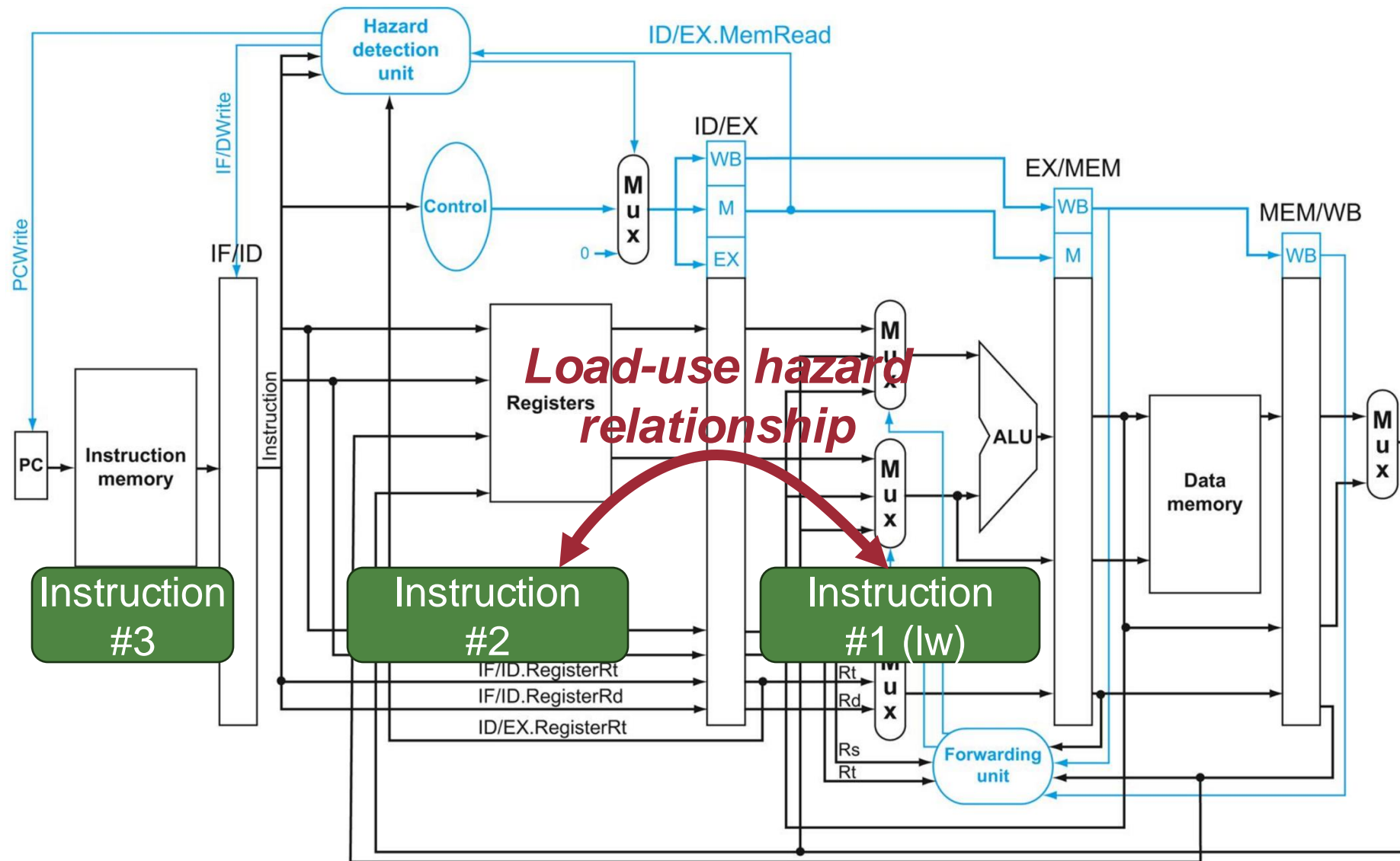
---



- 1) Prevent update of PC and IF/ID register
- 2) Force control values in ID/EX register to 0

# Stall Details: Current Cycle

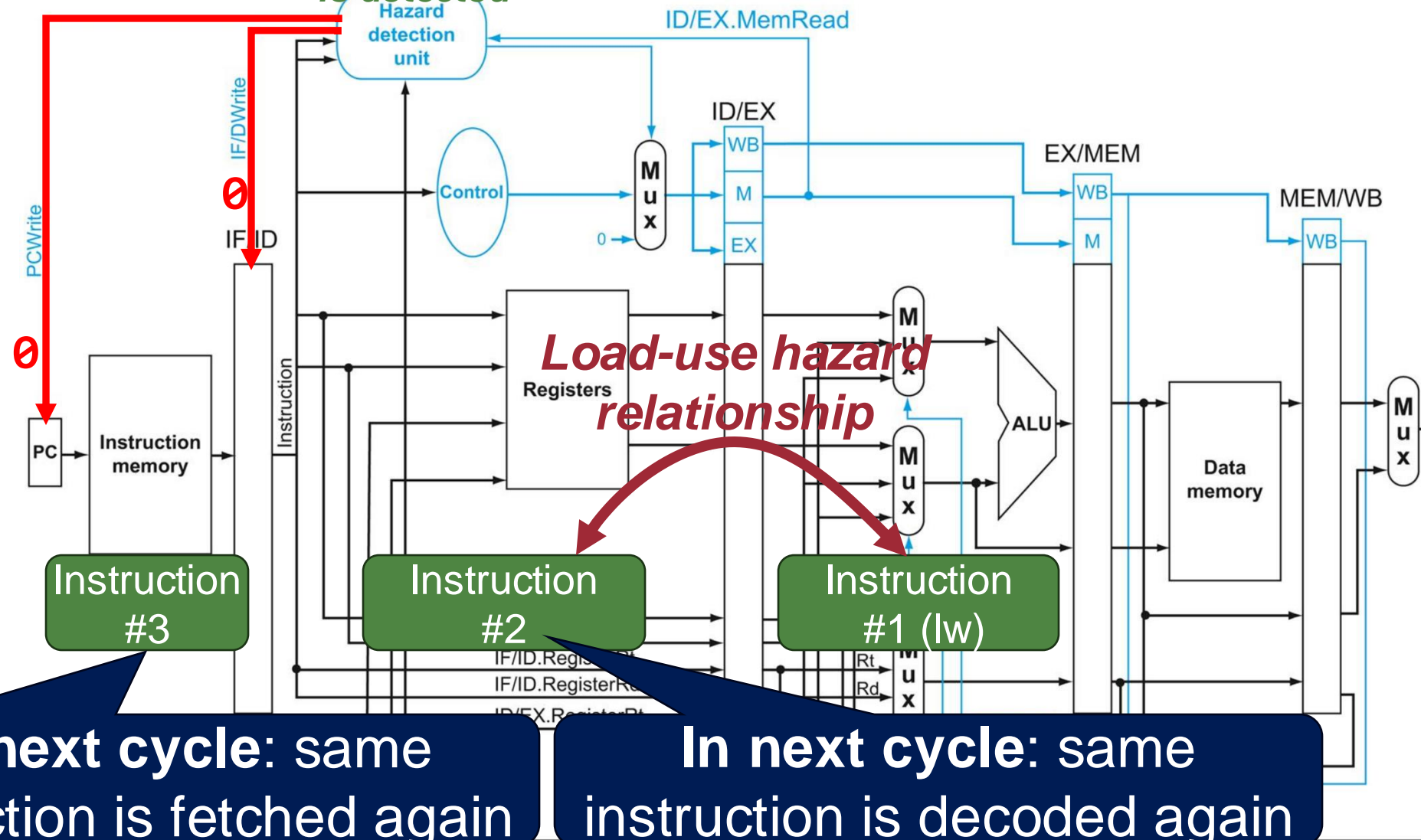
52



# 1) Prevent Update of PC and IF/ID Register

53

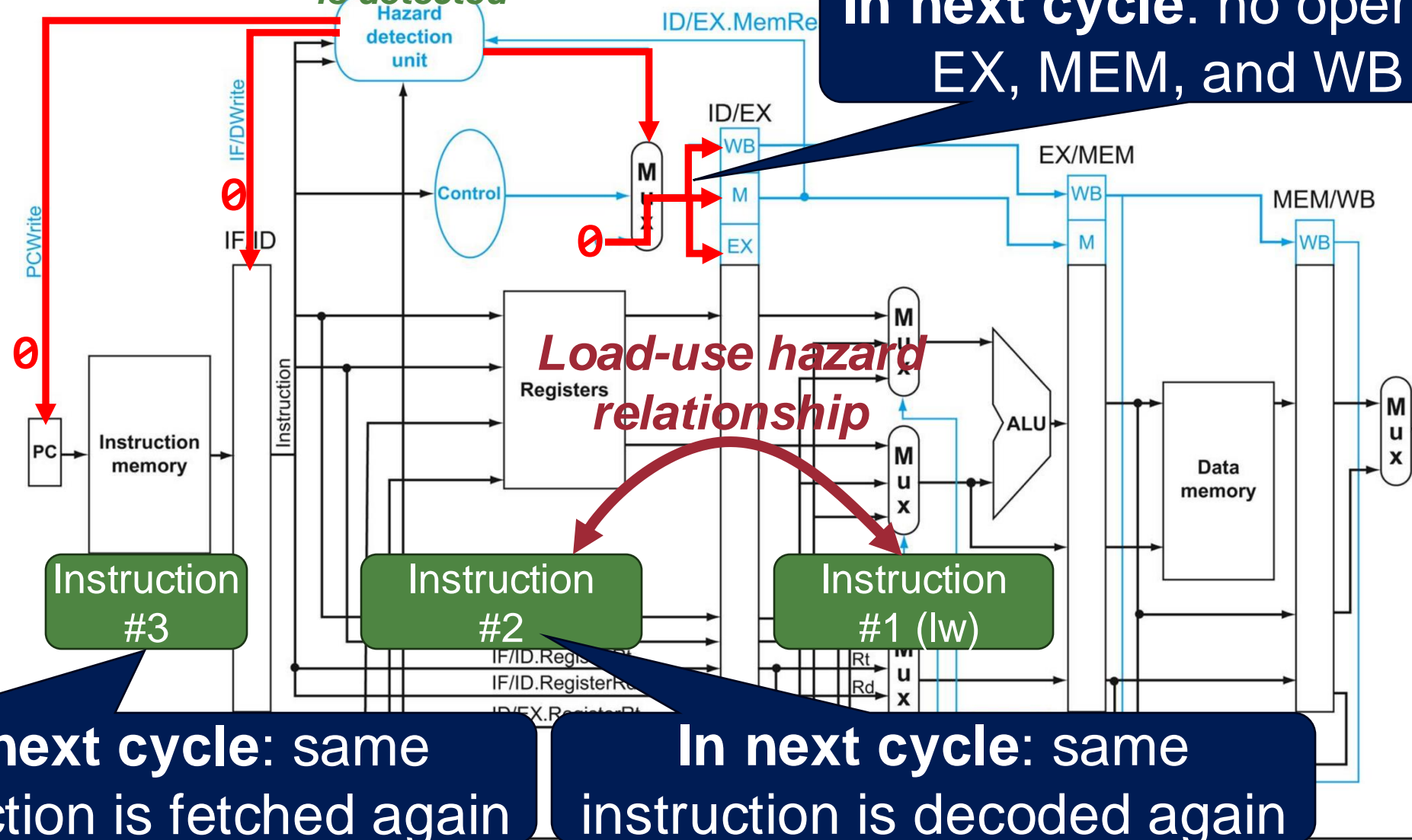
*If load-use hazard  
is detected*



## 2) Force Control Values in ID/EX Register to 0

**If load-use hazard is detected**

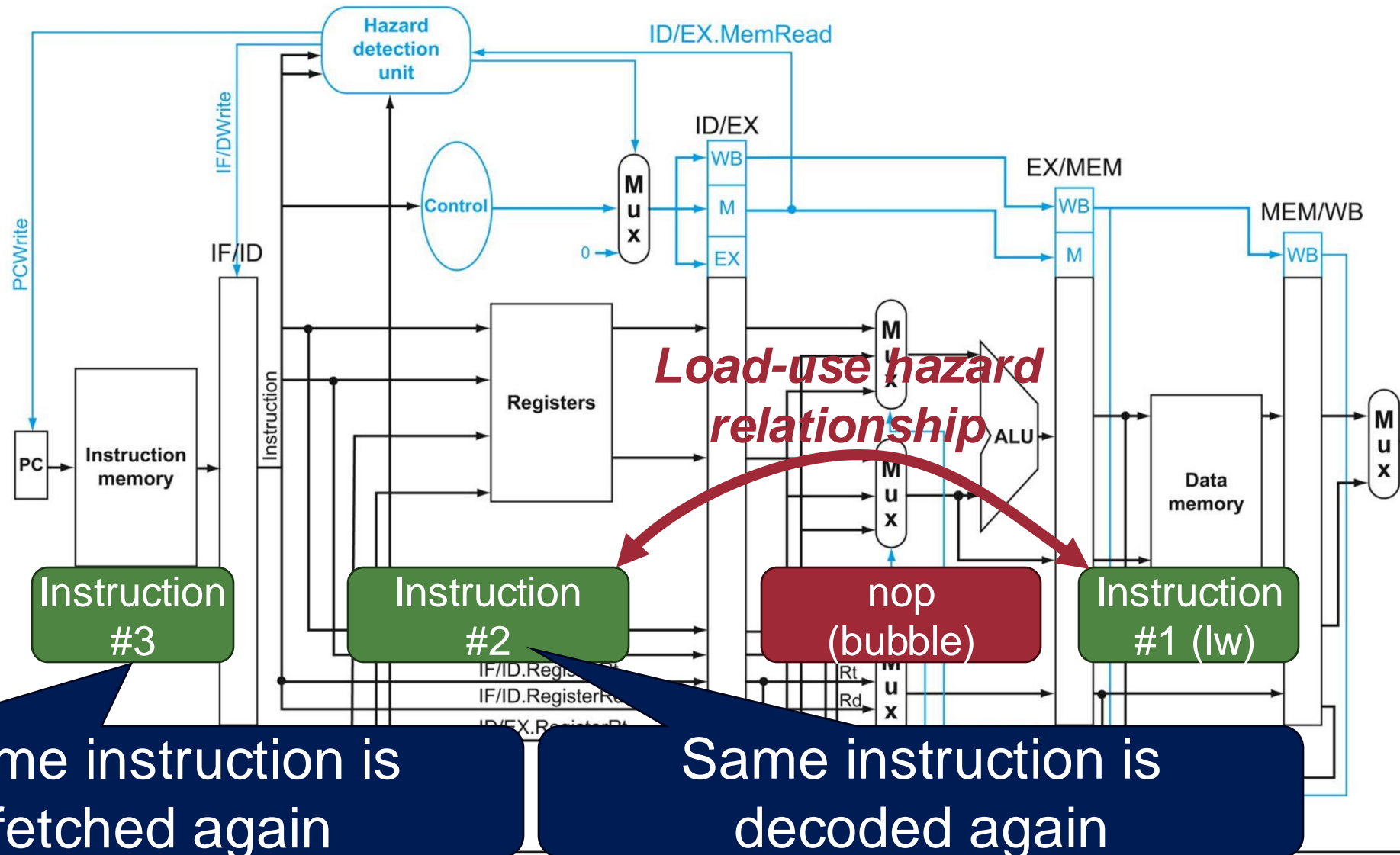
**In next cycle: no operation (nop)**  
EX, MEM, and WB do nop



**In next cycle: same instruction is fetched again**

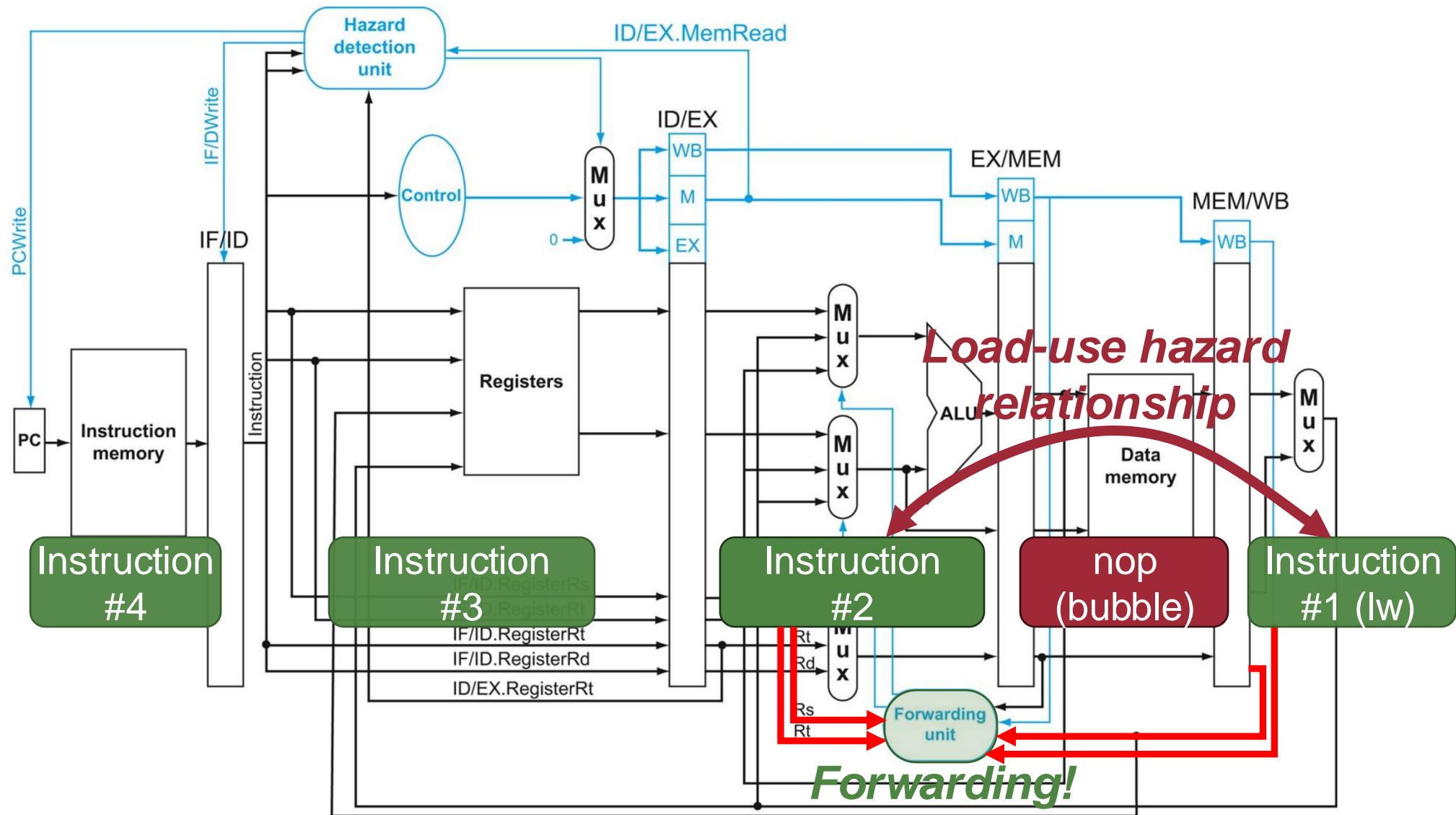
**In next cycle: same instruction is decoded again**

# Stall Details: Next Cycle (Stall Inserted)



# Stall Details: Next Next Cycle...

56





# Summary: Stall the Pipeline

---



## 1) Prevent update of PC and IF/ID register

- Same instruction is decoded again
- Same instruction (with same PC) is fetched again

## 2) Force control values in ID/EX register to 0

- EX, MEM and WB do nop (no-operation)

# Pipelining Hazards Summary



Situations that prevent starting the next instruction in the next cycle

## Hazard #1: Structural hazard

Conflict for use of a hardware resource

### Solution:

- Stall
- Resource duplication

## Hazard #2: Data hazard

An instruction cannot execute because data is not yet available

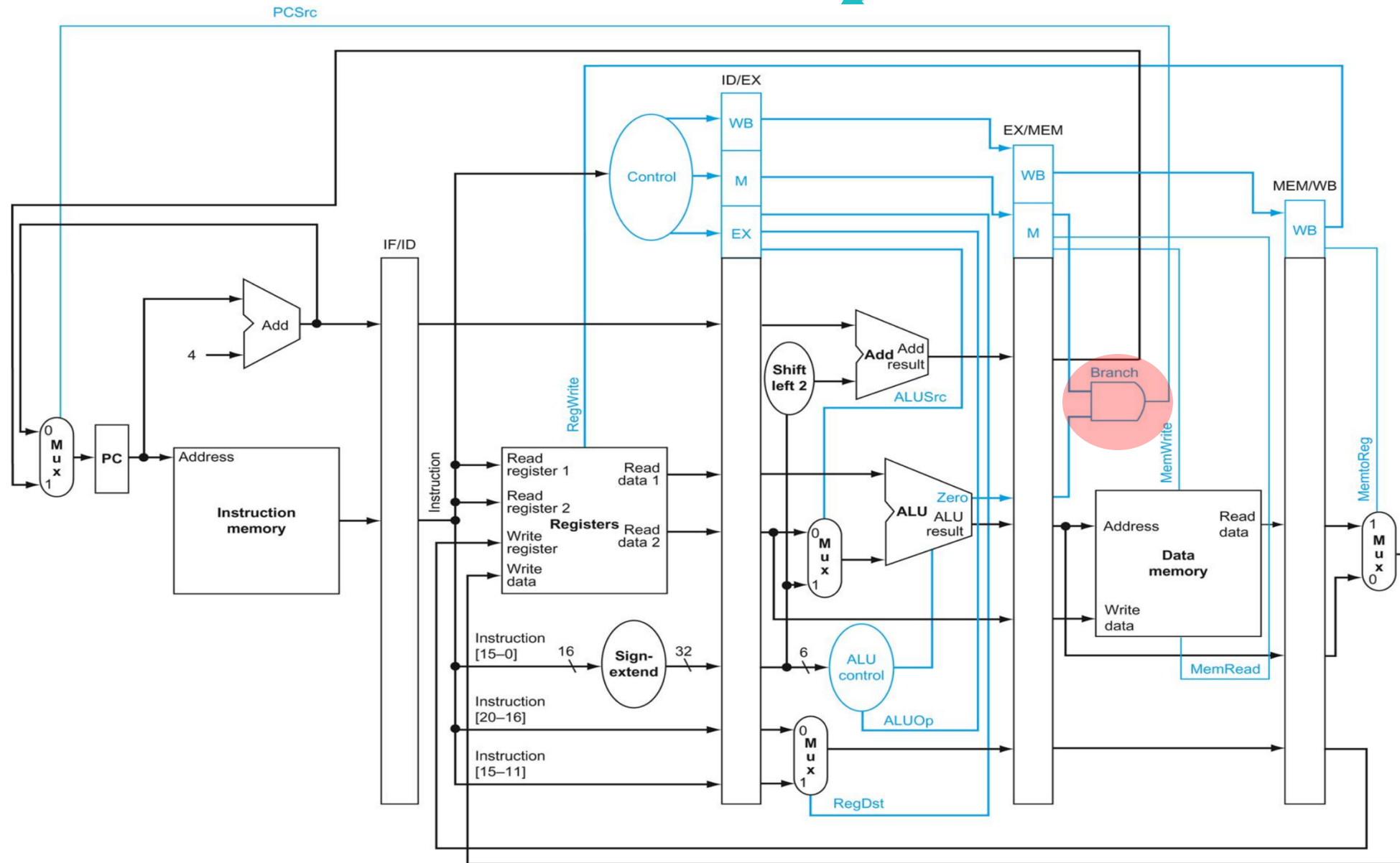
### Solution:

- Stall
- Forwarding
- Compiler optimization

## Hazard #3: Control hazard

# Control Hazard

# Recall: Branch Outcome Determined in MEM



# Control Hazard

---



- **Branch determines flow of control**
  - Fetching next instruction depends on branch outcome
  - **However**, the branch decision is made in the MEM stage

# Control Hazard: Example



beq \$1, \$3, 28

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

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

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

...

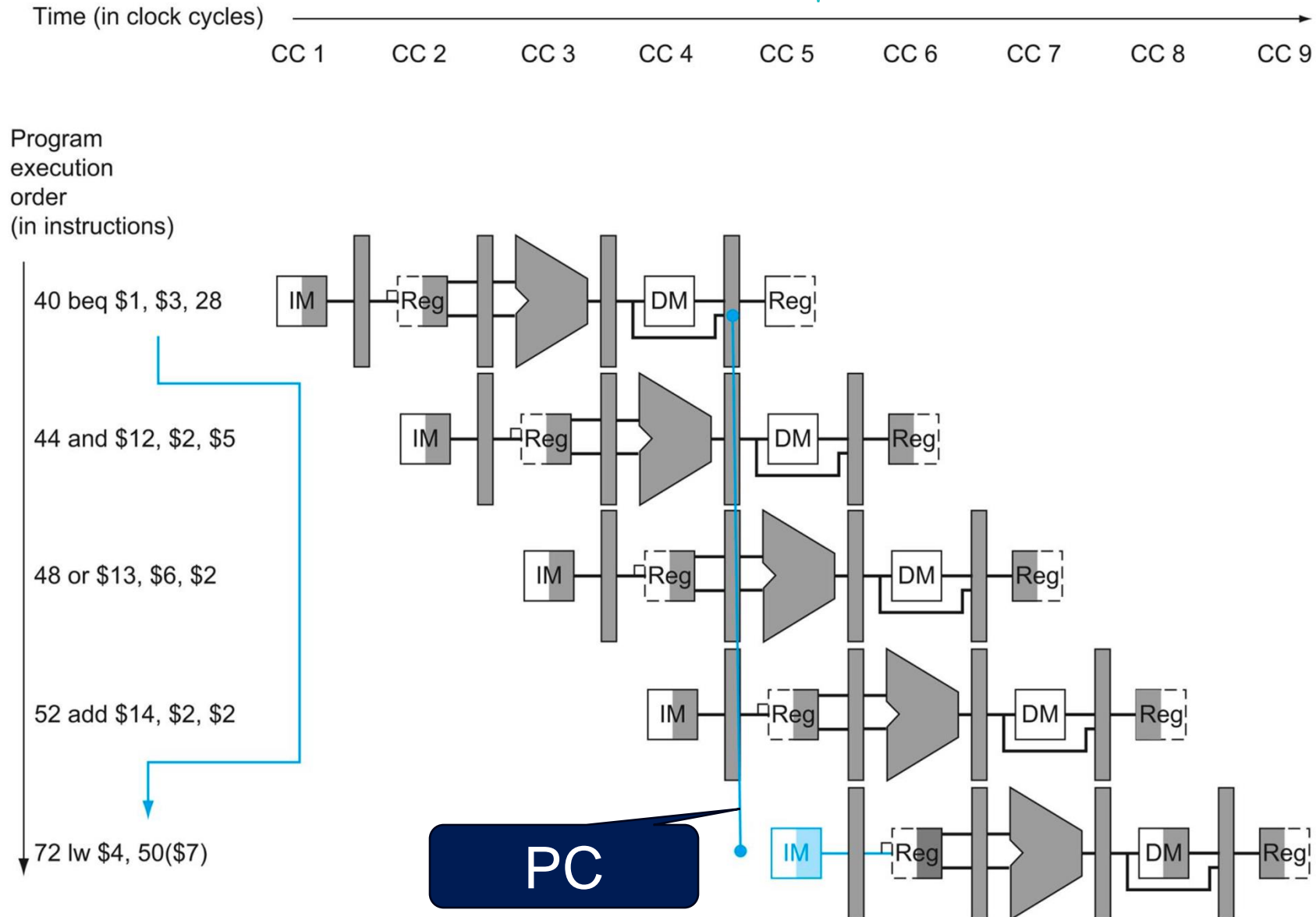
lw \$4, 50(\$7)

*Control flow*



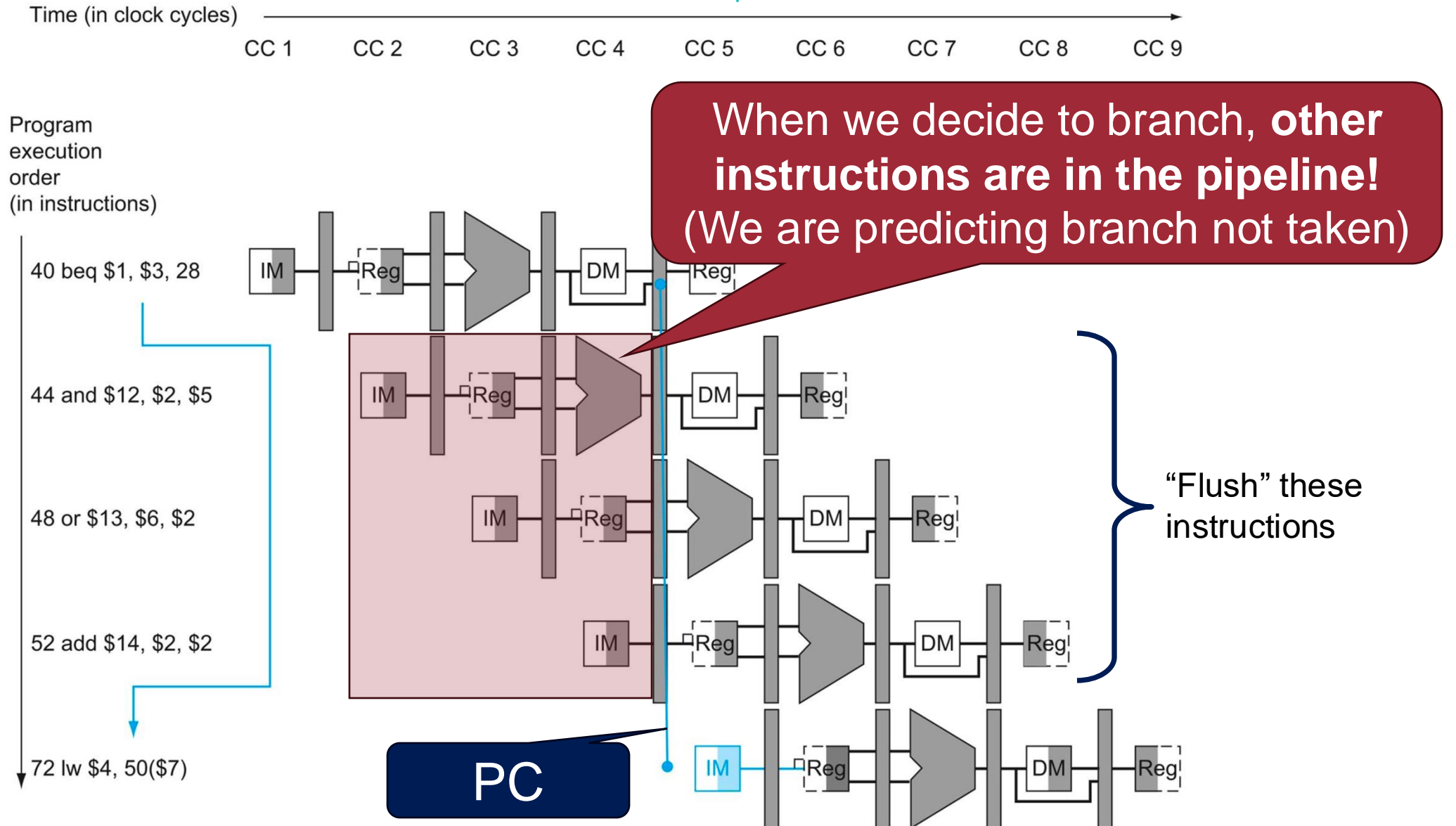
# Control Hazard: Example

63



# Control Hazard: Example

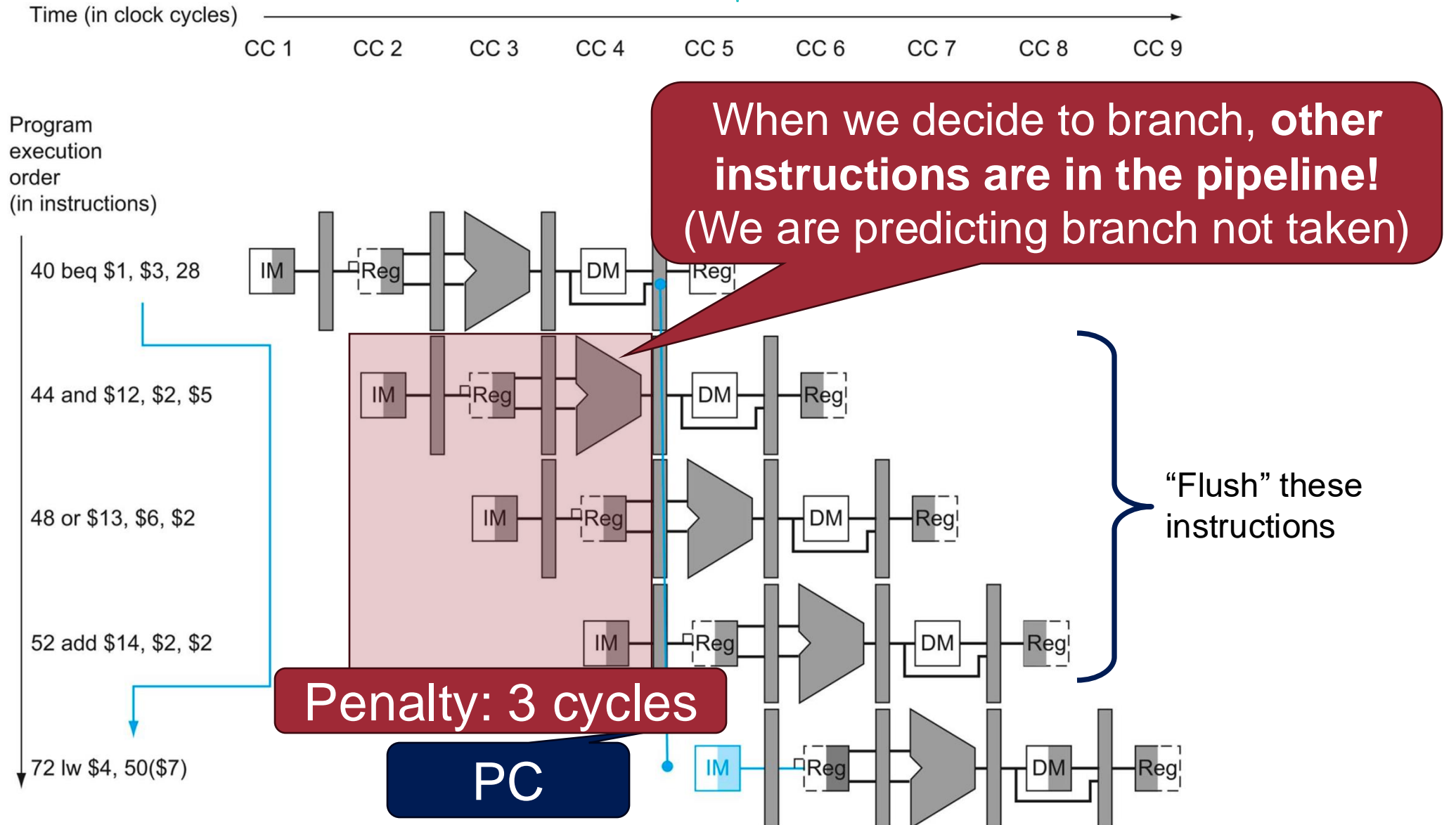
64





# Control Hazard: Example

65



# Pipelining Hazards Summary



Situations that prevent starting the next instruction in the next cycle

## Hazard #1: Structural hazard

Conflict for use of a hardware resource

### Solution:

- Stall
- Resource duplication

## Hazard #2: Data hazard

An instruction cannot execute because data is not yet available

### Solution:

- Stall
- Forwarding
- Compiler optimization

## Hazard #3: Control hazard

The next instruction is uncertain due to branching

### Solution:

- Stall
- Optimized branch processing
- Branch prediction
- Delayed branch

# Solution for Control Hazard: Stall

- **Most naïve approach: stall on branch**
  - Wait until branch outcome determined before fetching next instruction

[illegible]

# Solution for Control Hazard: Stall

- **Most naïve approach: stall on branch**
  - Wait until branch outcome determined before fetching next instruction

Branch instruction	IF	ID	EX	MEM	WB					
Branch successor		IF	stall	stall		IF	ID	EX	MEM	WB
Branch successor+1						IF	ID	EX	MEM	WB
Branch successor+2							IF	ID	EX	MEM
Branch successor+3								IF	ID	EX
Branch successor+4									IF	ID
Branch successor+5										IF

Wait for 3 clock cycles until  
the PC value is available

*Work correctly, but is slow  
(3 cycle penalty regardless of whether a branch occurs)*

# Solution for Control Hazard: Stall

- For 5-stage pipeline, 3 cycle penalty for stall
- 15% branch frequency

What is the CPI?

$$1 + (0.15 \times 3) = 1.45$$

CPI in the ideal  
situation of pipelining

3 cycle penalty

# Solution for Control Hazard: Stall

- **Most naïve approach: stall on branch**

- Wait until branch outcome determined before fetching next instruction

Branch instruction	IF	ID	EX	MEM	WB					
Branch successor		IF	stall	stall	IF	ID	EX	MEM	WB	
Branch successor+1						IF	ID	EX	MEM	WB
Branch successor+2							IF	ID	EX	MEM
Branch successor+3								IF	ID	EX
Branch successor+4									IF	ID
Branch successor+5										IF

Wait for 3 clock cycles until  
the PC value is available

*Work correctly, but is slow  
(3 cycle penalty regardless of whether a branch occurs)*

# Solution for Control Hazard: Stall

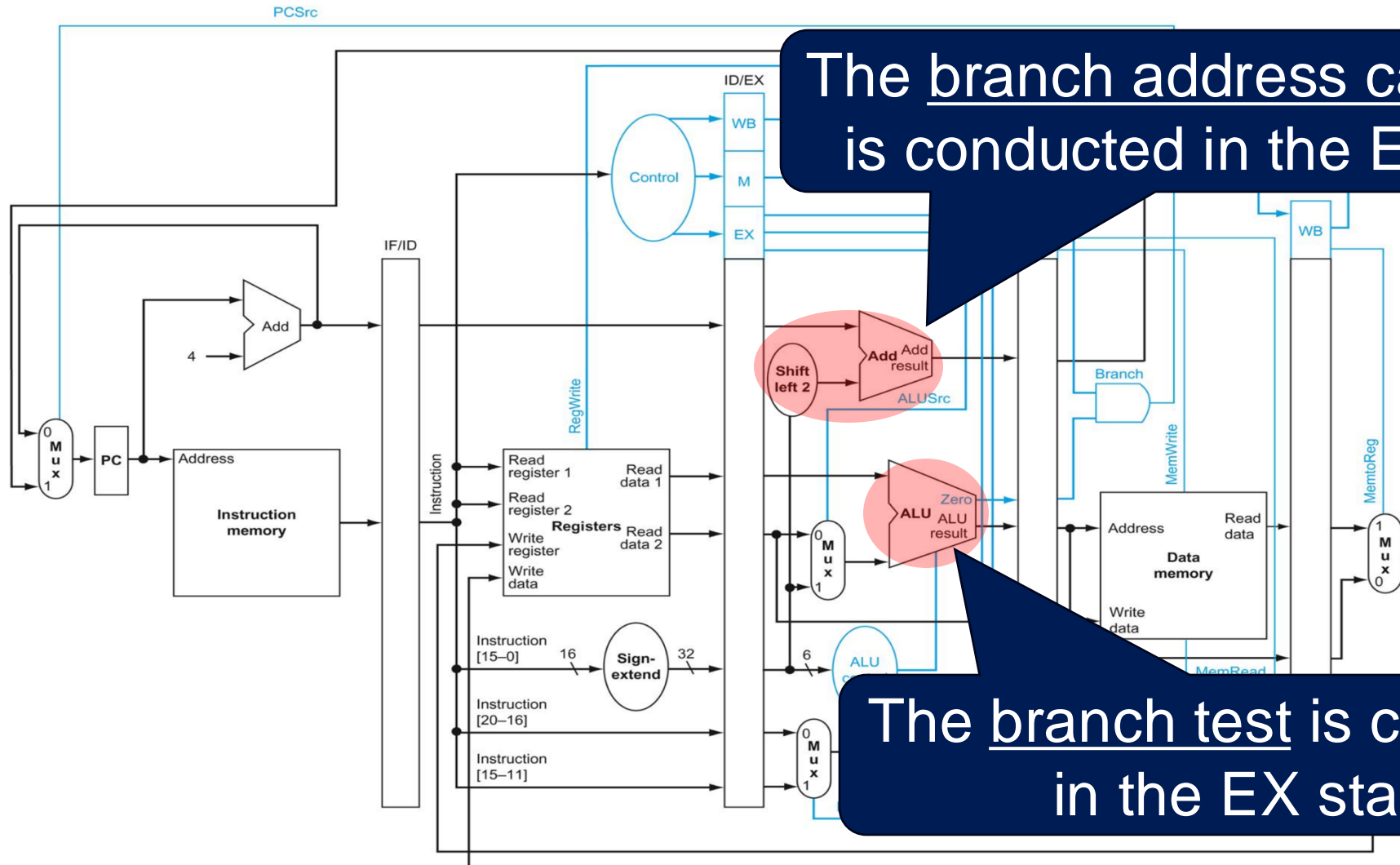
---



*How can the penalty be reduced to less than 3 clock cycles?*

Optimized branch processing!  
(H/W modification to achieve  
3 cycle penalty → 1 cycle penalty)

# Optimized Branch Processing: Observation



The branch address calculation is conducted in the EX stage

The branch test is conducted in the EX stage



The diagram illustrates the internal logic of a 5-stage MIPS processor. The stages are labeled at the top: IF (Instruction Fetch), ID/EX (Instruction Decode/Execute), WB (Write Back), M (Memory Access), and EX (Execute). The PCSrc (Program Counter Source) is shown at the top left. The diagram includes various components such as multiplexers (Mux), adders (Add), shifters (Shift left 2), and ALUs. It also shows the flow of data between registers (RegWrite) and memory (Data memory). The diagram is annotated with three callouts:

- Move up the branch address calculation**: A green callout pointing to the ALU in the EX stage, which calculates the branch address by adding the PCSrc to the branch offset (shifted left 2).
- Move up the branch test**: A green callout pointing to the ALU in the EX stage, which performs the branch test by comparing the ALU result to zero.
- The branch address calculation is conducted in the EX stage**: A blue callout pointing to the ALU in the EX stage, which calculates the branch address by adding the PCSrc to the branch offset (shifted left 2).
- The branch test is conducted in the EX stage**: A blue callout pointing to the ALU in the EX stage, which performs the branch test by comparing the ALU result to zero.

## Move up the branch address calculation

# Move up the branch test

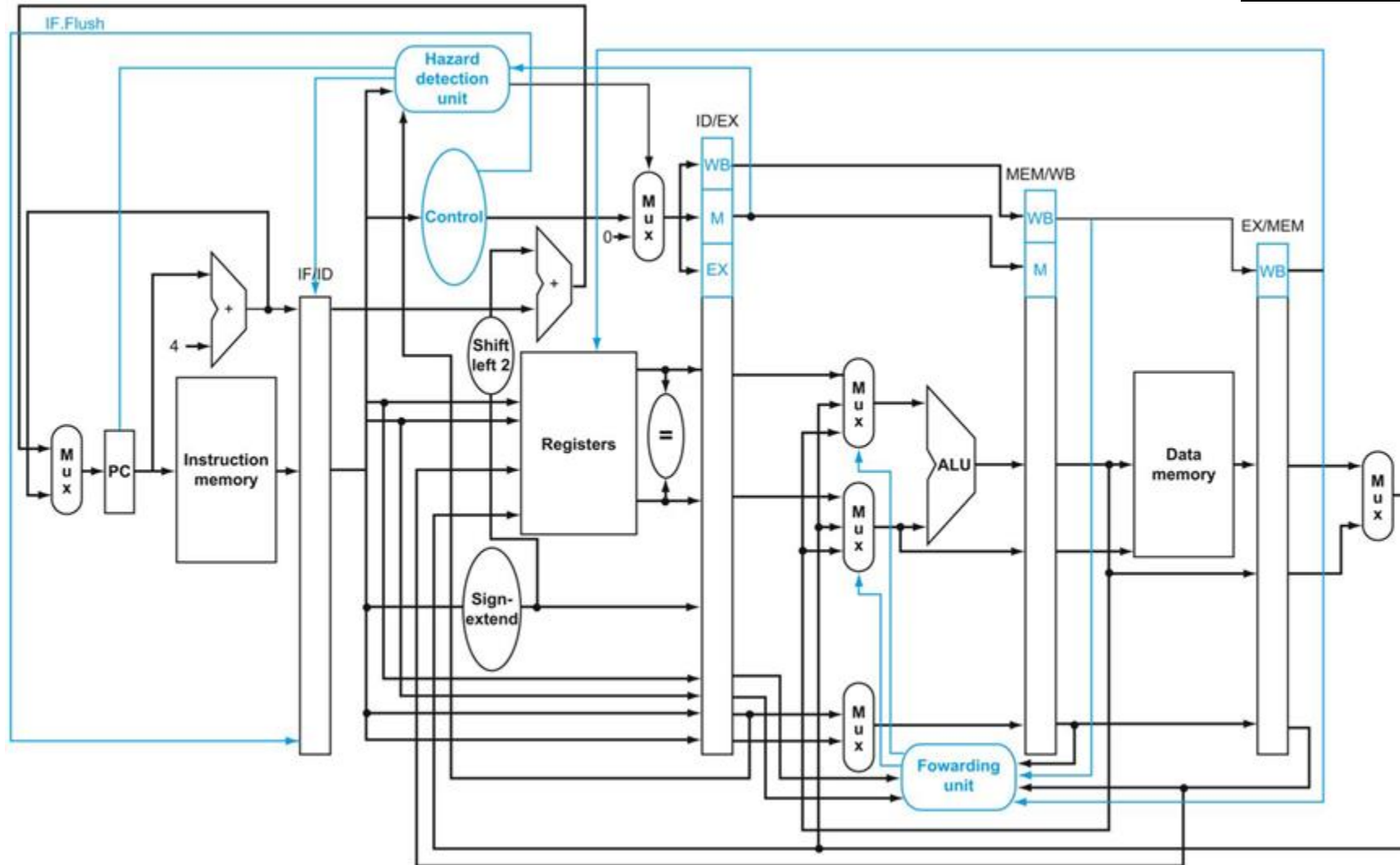
The branch address calculation  
is conducted in the EX stage

The branch test is conducted  
in the EX stage

# Solution for Control Hazard: Optimized Branch Processing

74

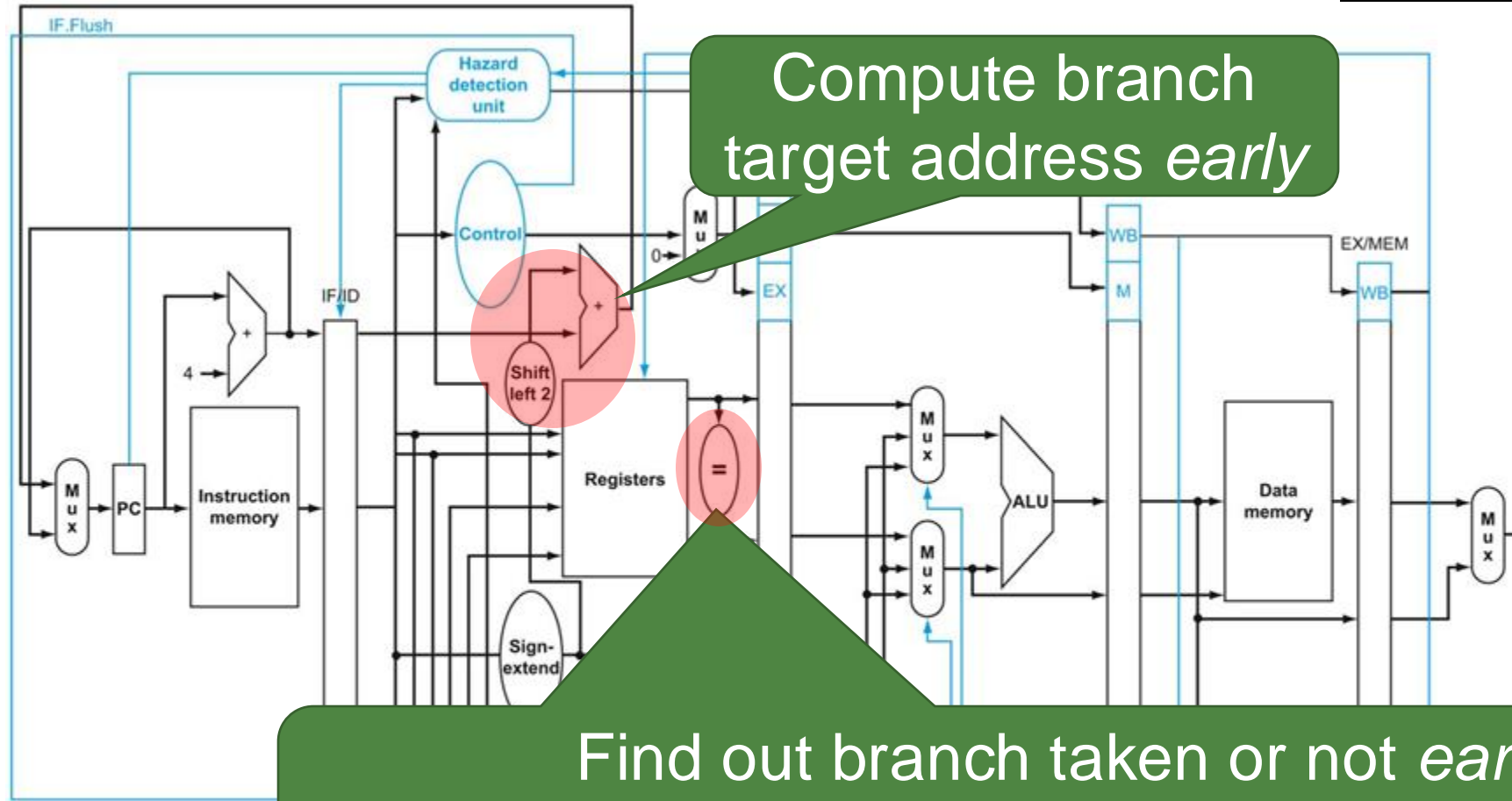
Move hardware to determine branch outcome to the ID stage



# Solution for Control Hazard: Optimized Branch Processing

75

Move hardware to determine branch outcome to the ID stage



- Bringing in the entire ALU is expensive (non-sense at all)
- Deciding whether to branch can be done with just an equality check

# Solution for Control Hazard: Optimized Branch Processing

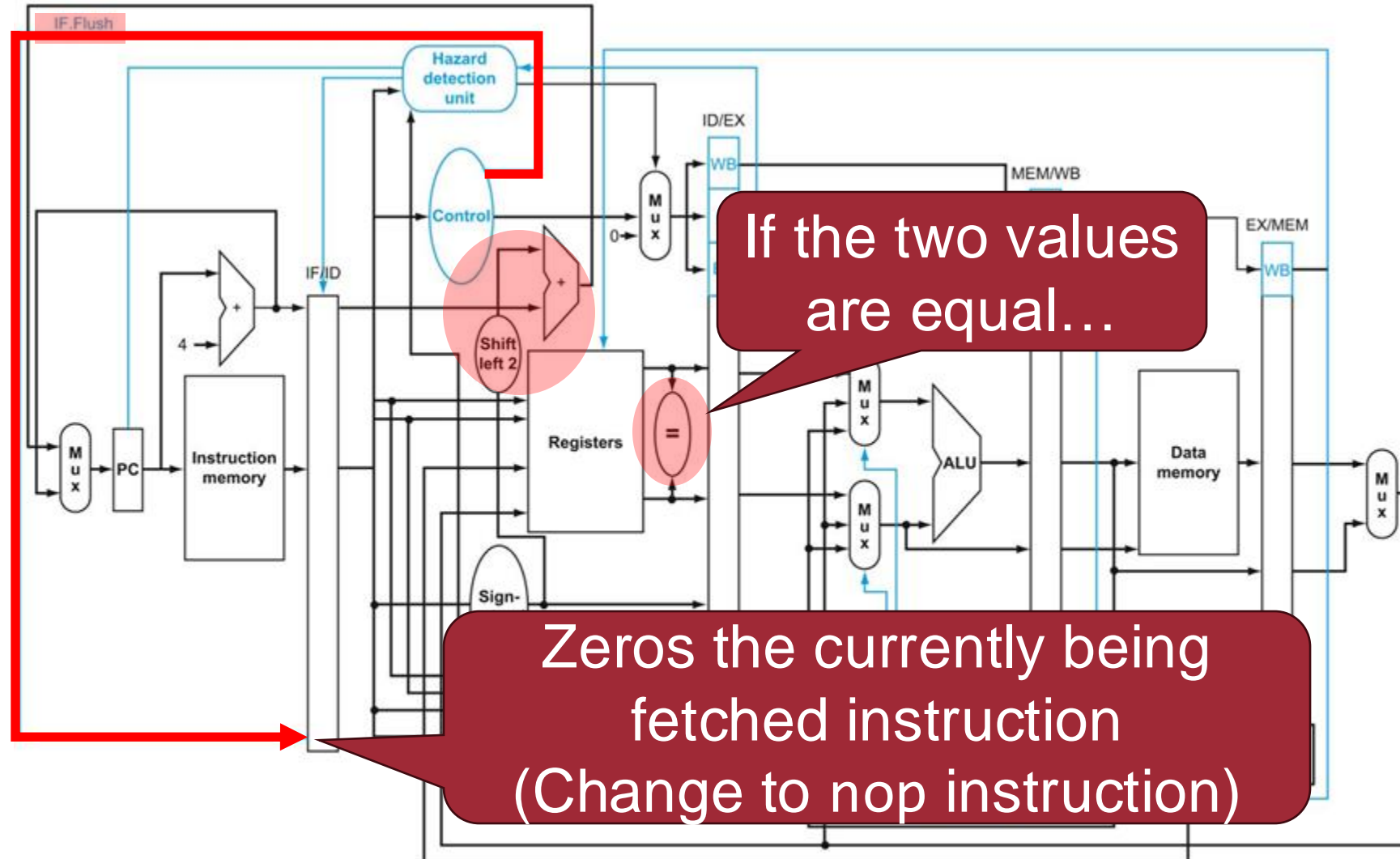
76

Move hardware to determine branch outcome to the ID stage

- Compute branch target address *early*
- Find out branch taken or not *early*
  - Extra lightweight hardware for equality check
  - Equality can be tested by first XORing inputs and then ORing all the results

Reduce branch delay to ***one clock cycle***  
(*Flush one instruction if the branch is taken*)

# Optimized Branch Processing: Flushing Logic



# Optimized Branch Processing: Example

78

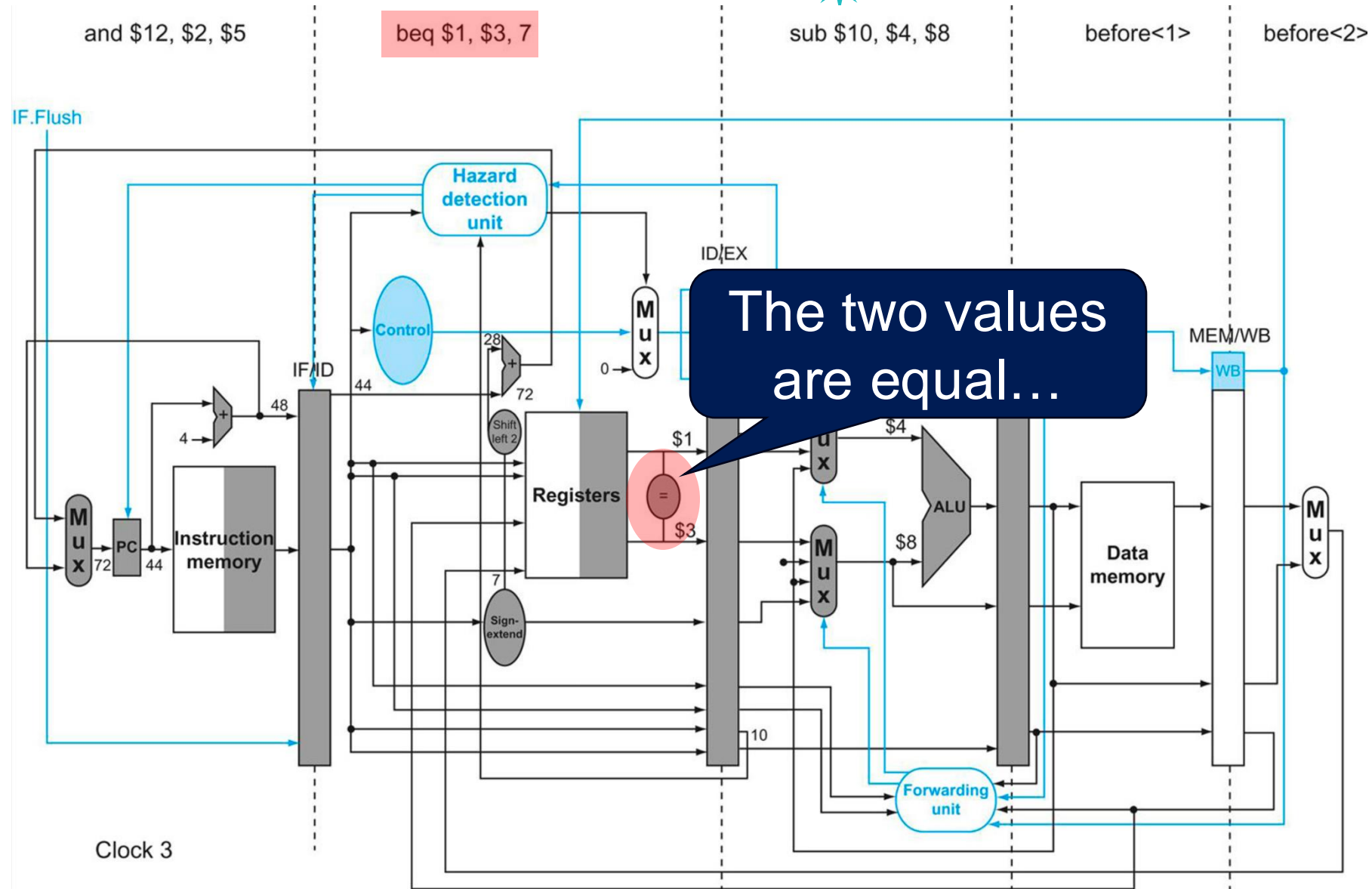
```
36:  sub    $10, $4, $8
40:  beq    $1,  $3, 7
44:  and    $12, $2, $5
48:  or     $13, $2, $6
52:  add    $14, $4, $2
56:  slt    $15, $6, $7
    ...
72:  lw     $4, 50($7)
```

*Control flow*

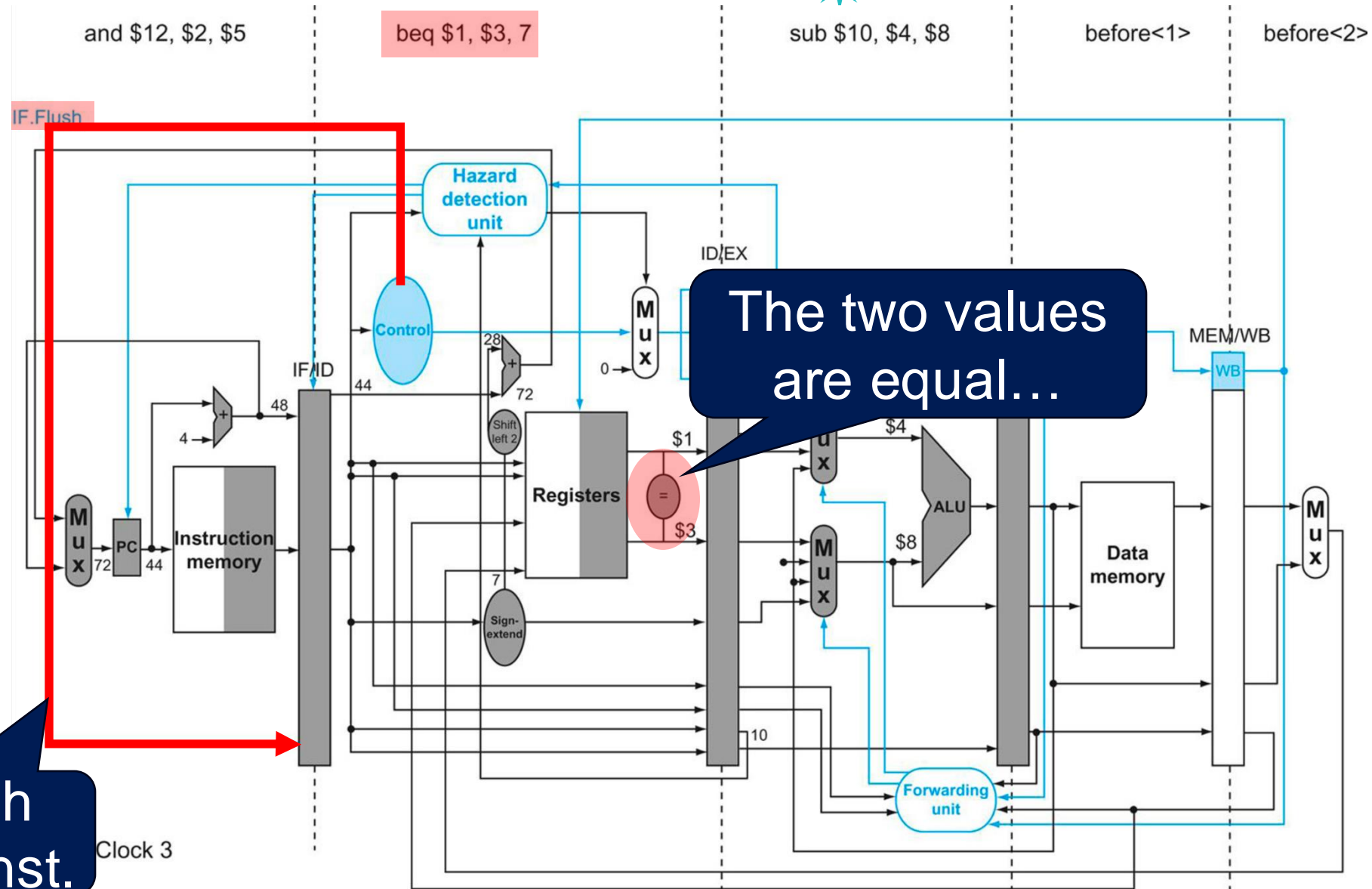


# Optimized Branch Processing: 3rd Cycle

79



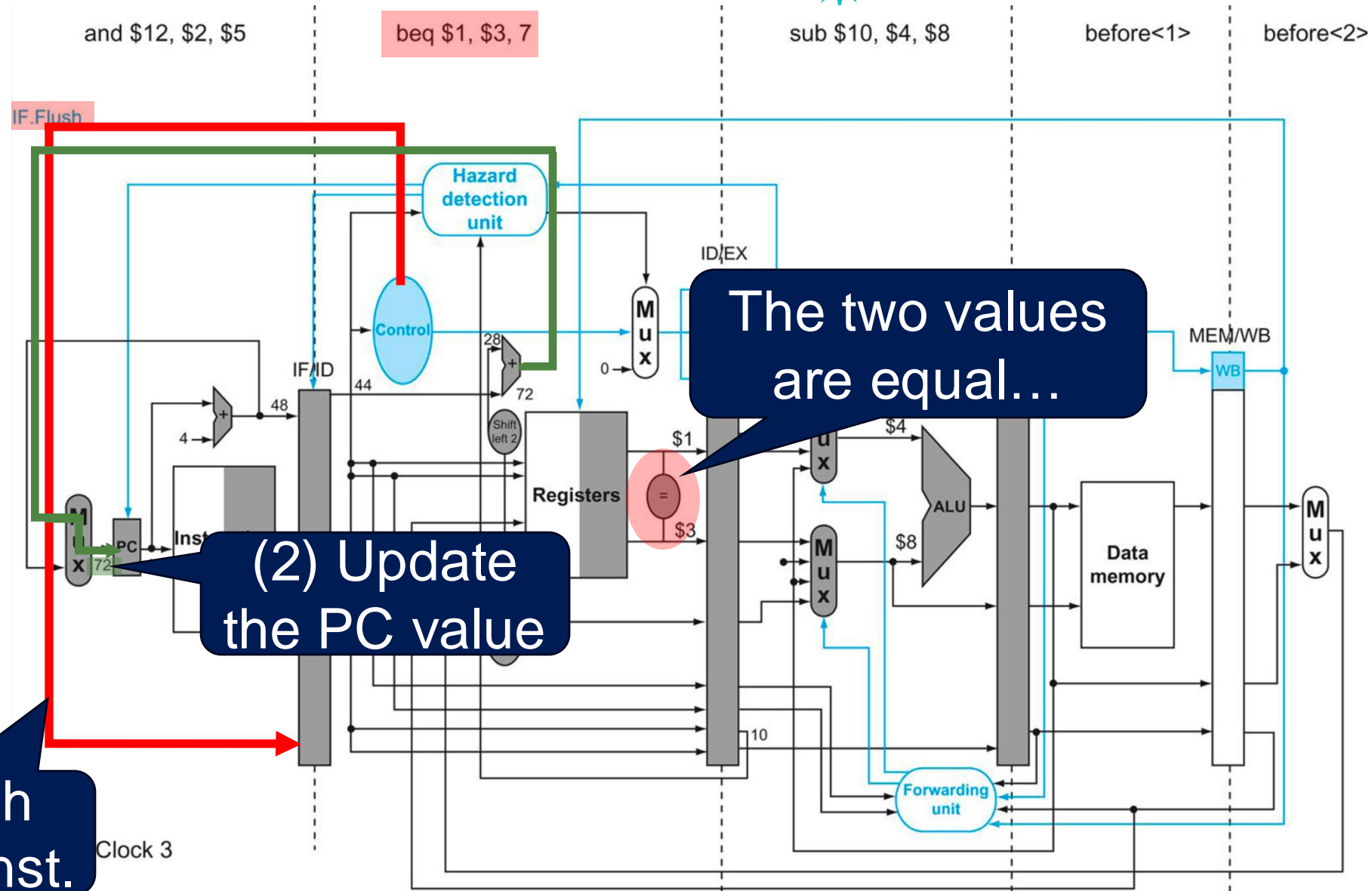
# Optimized Branch Processing: 3rd Cycle





# Optimized Branch Processing: 3rd Cycle

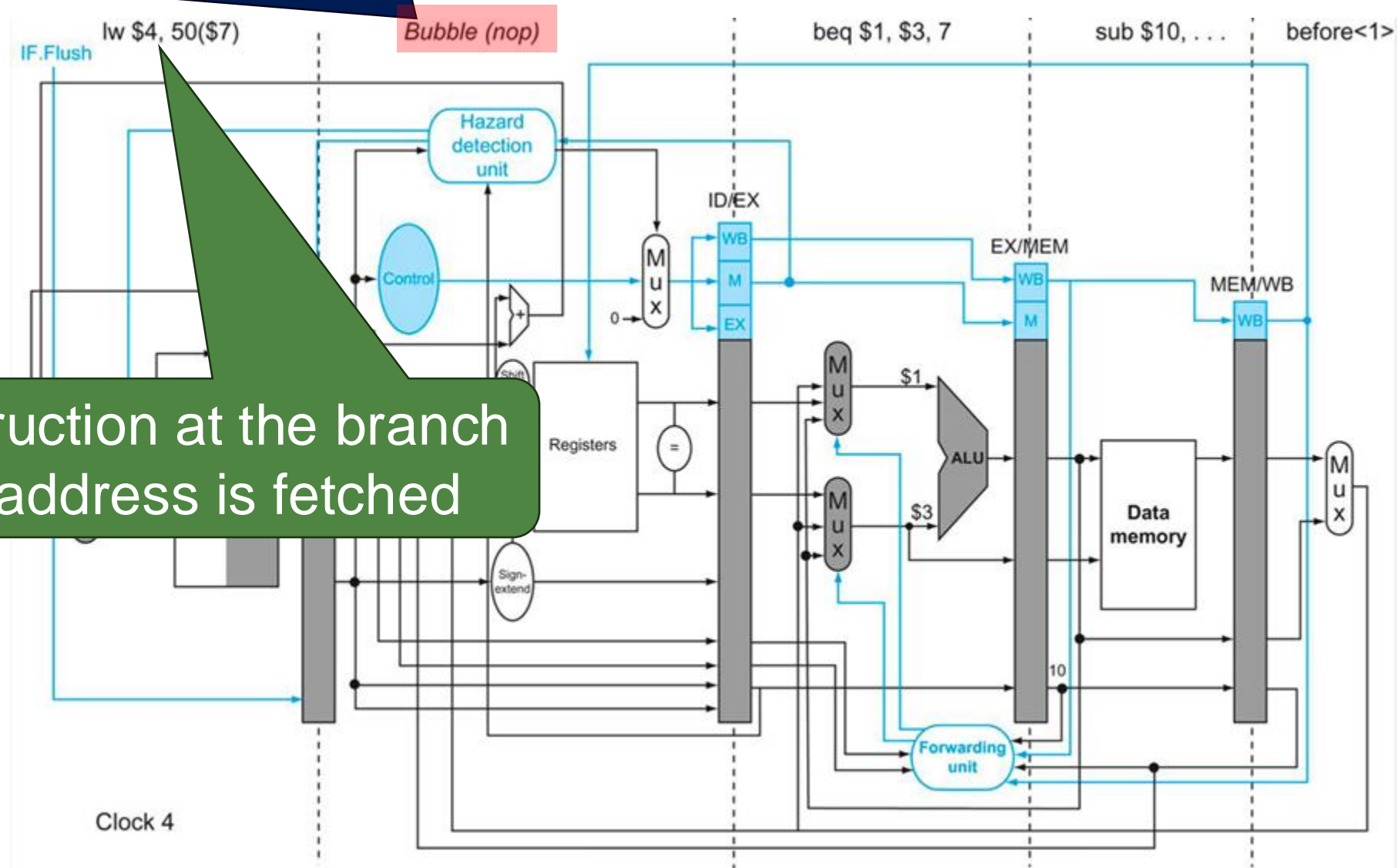
81



# Optimized Branch Processing: 4th Cycle

and inst. was discarded from pipeline (only one cycle penalty)

The instruction at the branch target address is fetched



# Solution for Control Hazard: Optimized Branch Processing

83



*If there is a branch instruction, there is still a 1-cycle penalty.*

*How can performance be further improved?*

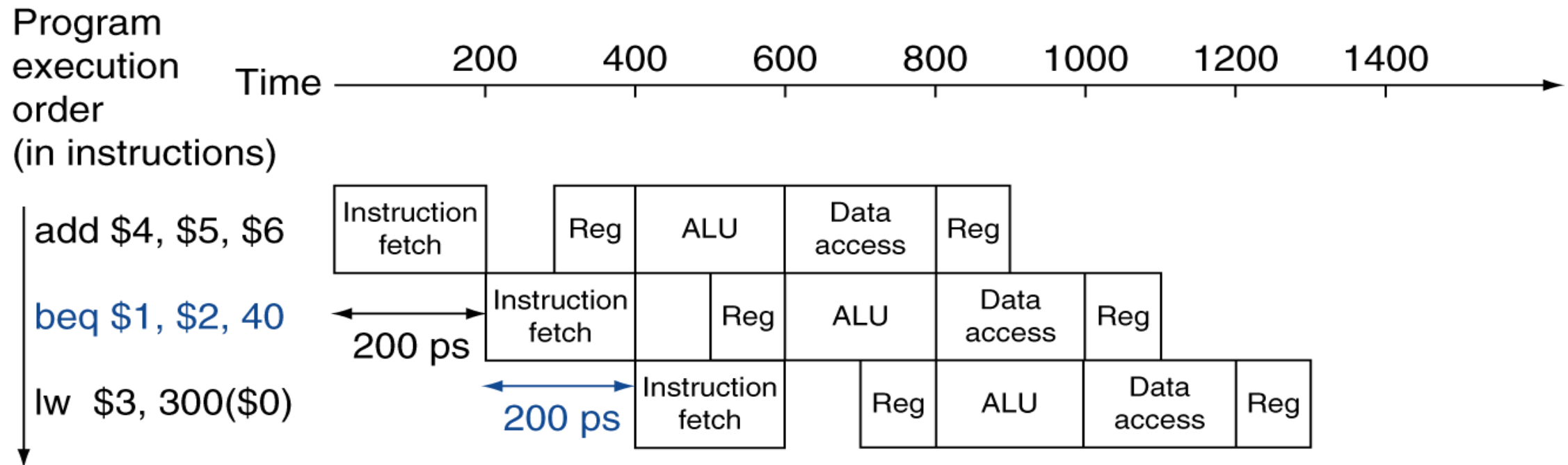
**Branch prediction!**

# Solution for Control Hazard: Branch Prediction

34

Guess one direction then backup if wrong

– **Our prediction so far:** *a branch is not taken*

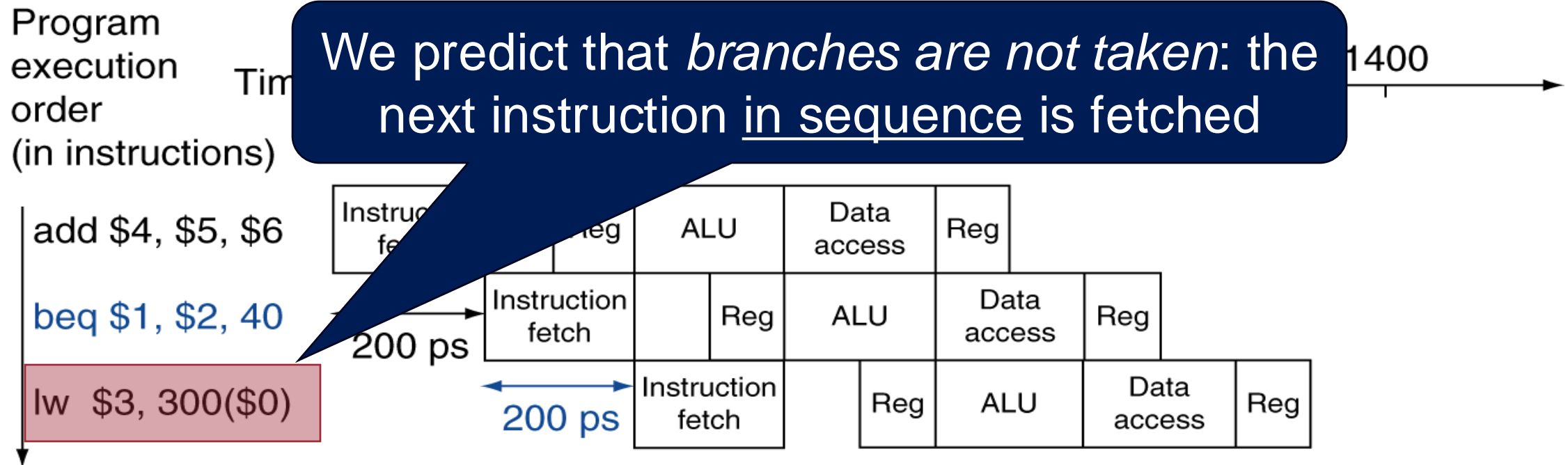


# Solution for Control Hazard: Branch Prediction

35

Guess one direction then backup if wrong

–Our prediction so far: *a branch is not taken*



Prediction  
correct

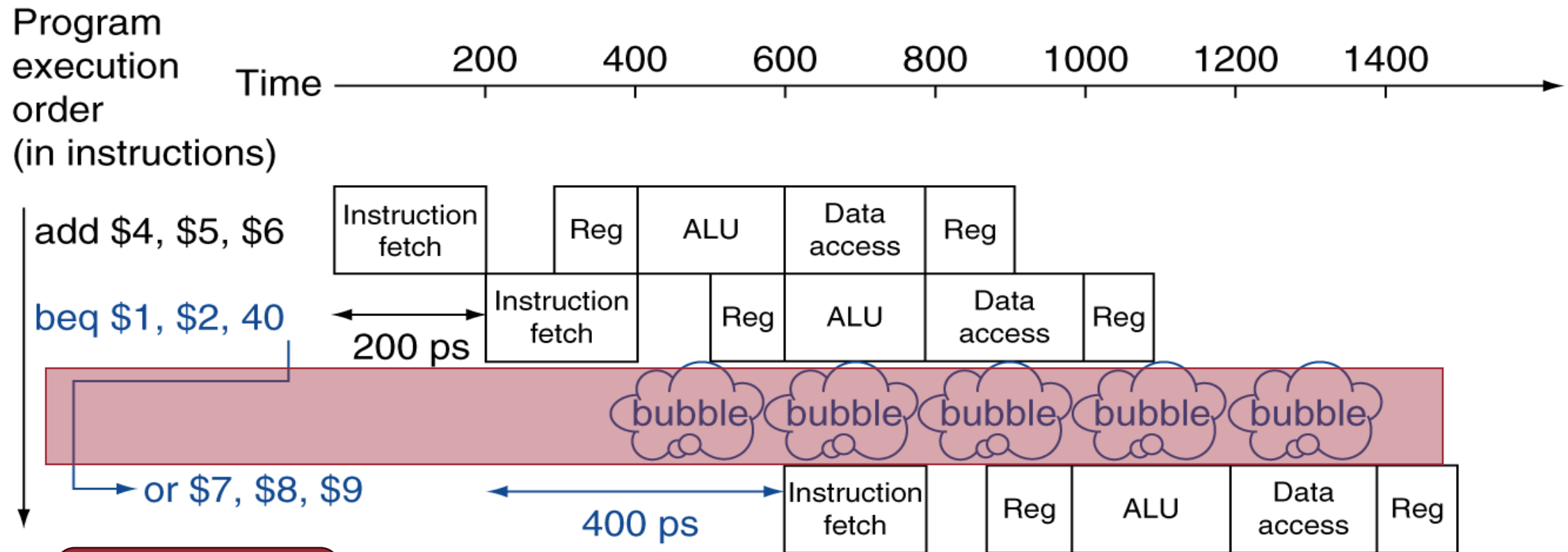
*No penalty*

# Solution for Control Hazard: Branch Prediction

36

Guess one direction then backup if wrong

– **Our prediction so far: a branch is not taken**



Prediction  
incorrect

*Flushing the pipeline (i.e., inserting a bubble)  
when we are wrong*

# Types of Branch Prediction

Guess one direction then backup if wrong

## 1. Static branch prediction

- A fixed decision based on certain predefined rules
- Rule #1: Predicting *Not Taken*
- Rule #2: Predicting *Taken*

Programs with few branches  
tend to have poor performance

Programs with many branches  
tend to have poor performance

**A simple static prediction scheme  
will probably waste too much performance**

# Types of Branch Prediction



Guess one direction then backup if wrong

## 1. Static branch prediction

- A fixed decision based on certain predefined rules
- Rule #1: Predicting *Not Taken*
- Rule #2: Predicting *Taken*

## 2. Dynamic branch prediction

- Prediction of branches at runtime using runtime information



# Dynamic Branch Prediction: Observation

89

```
for (i = 0, i < 100; i++)  
    loop body
```

*Runtime information*

**Branch history is important: if a branch was taken previously, it is likely that the branch will be taken again the next time**

# Dynamic Branch Prediction: Basic Idea

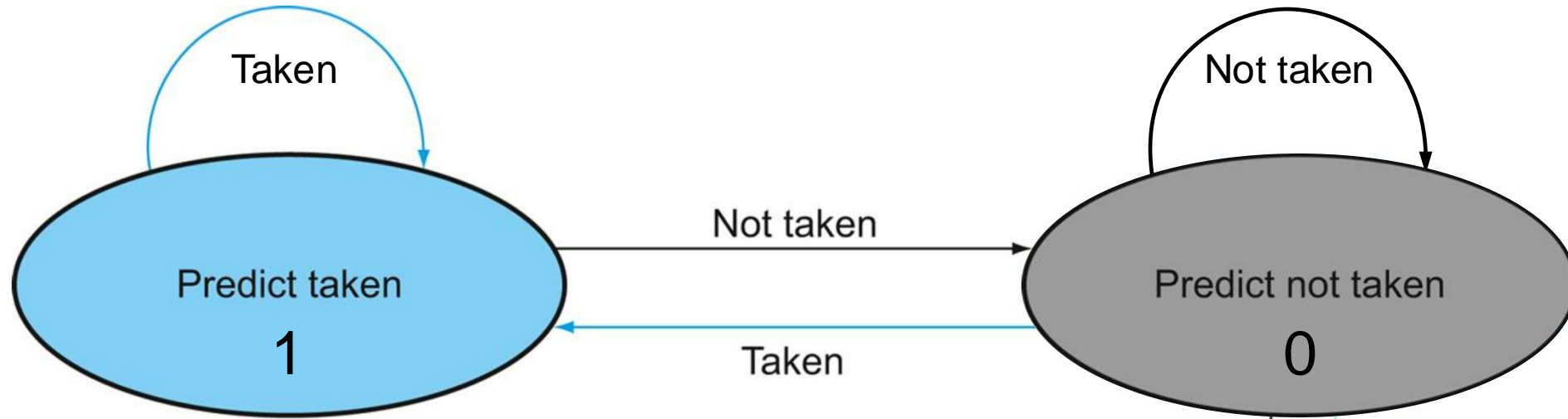
The memory contains one or more bits indicating whether the branch was recently taken or not

```
for (i = 0, i < 100; i++)  
    loop body
```



Branch prediction buffer  
(a.k.a., branch history table)

# Dynamic Branch Prediction: 1-bit Prediction



# Dynamic Branch Prediction: Initial State

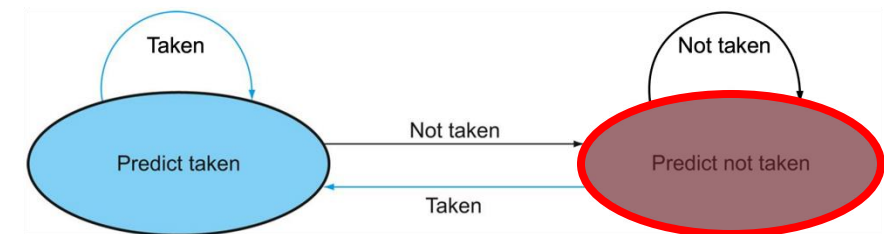
92

**Current policy:** the next instruction in sequence is fetched

```
for (i = 0, i < 100; i++)  
    loop body
```

0 (Not taken)

Branch prediction buffer  
(a.k.a., branch history table)



# Dynamic Branch Prediction: 1st Iteration

93

Branch outcome: taken

≠

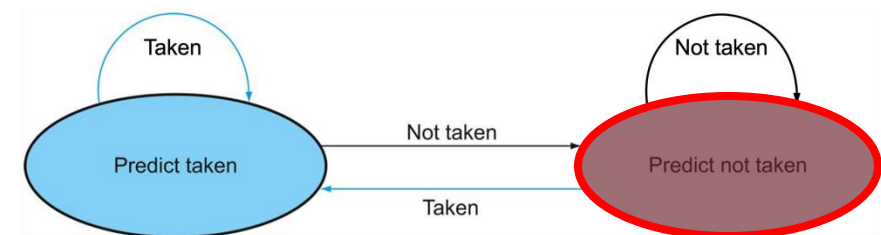
Current policy: the next instruction in sequence is fetched

PC → **for** ( $i = 0, i < 100; i++$ )  
    loop body

0 (Not taken)

Branch prediction buffer  
(a.k.a., branch history table)

Mispredict  
(1 cycle penalty)



# Dynamic Branch Prediction: Update Buffer

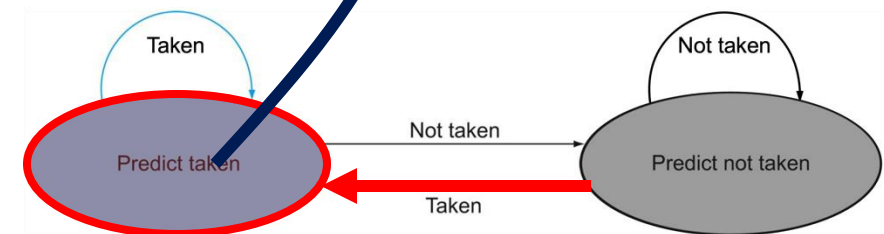
94

Branch outcome: taken

PC → **for** ( $i = 0, i < 100; i++$ )  
    loop body

1 (Taken)

Branch prediction buffer  
(a.k.a., branch history table)



# Dynamic Branch Prediction: 2nd Iteration <sup>95</sup>

Branch outcome: taken

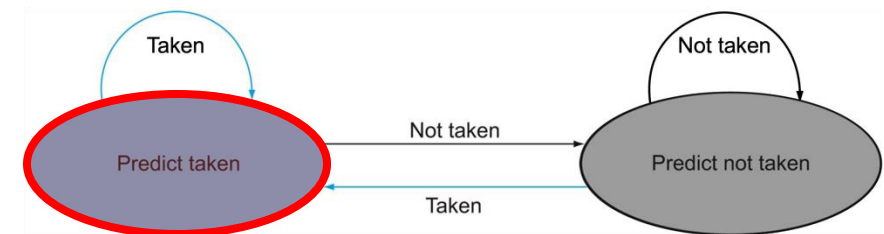
Current policy: the instruction  
at branch target is fetched

PC → **for** ( $i = 0, i < 100; i++$ )  
    loop body

1 (Taken)

Branch prediction buffer  
(a.k.a., branch history table)

No penalty



# Dynamic Branch Prediction: Update Buffer

96

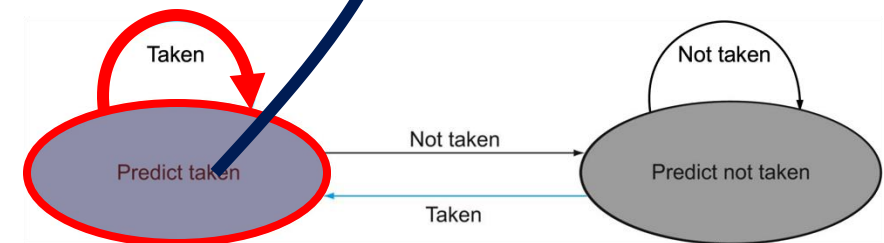
Branch outcome: taken

PC → **for** ( $i = 0, i < 100; i++$ )  
    loop body

1 (Taken)

Branch prediction buffer  
(a.k.a., branch history table)

*No change*





# Dynamic Branch Prediction: 3rd Iteration 97

Branch outcome: taken

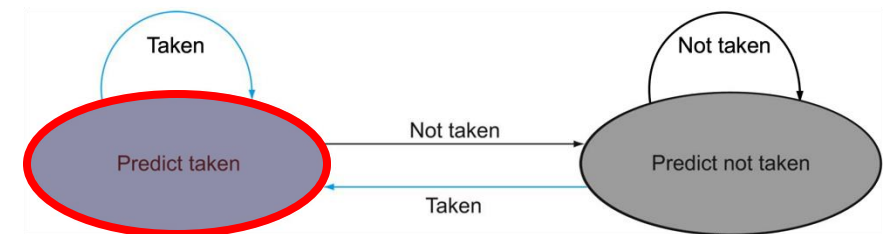
Current policy: the instruction  
at branch target is fetched

PC → **for** ( $i = 0, i < 100; i++$ )  
    loop body

1 (Taken)

Branch prediction buffer  
(a.k.a., branch history table)

No penalty



# Dynamic Branch Prediction



- Prediction of branches at runtime using runtime information
- Use a **branch prediction buffer** (a.k.a., branch history table)
  - A bit says whether the branch was recently taken or not
  - If an instruction is predicted to be taken, fetching begins from the branch target
- Ideally, each branch instruction would have its own branch prediction buffer
  - However, due to the performance issue, the buffer table size is limited
  - Therefore, multiple branch instructions can share the same buffer
  - Indexed using *the lower bits of the branch instruction's address*

# A Performance Shortcoming of 1-bit Prediction

```
for (i = 0, i < 100; i++)  
    for (j = 0, j < 3; j++)  
        loop body
```

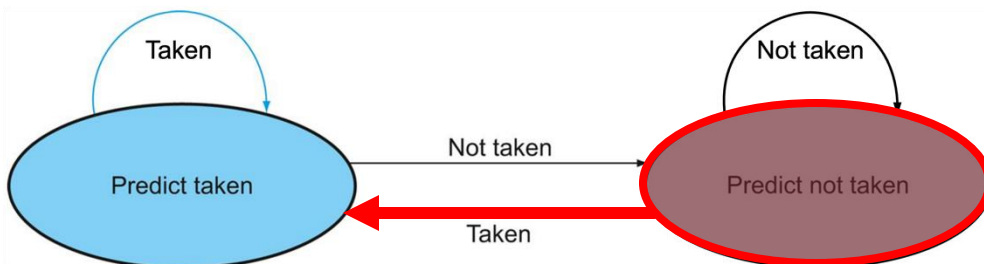
# A Performance Shortcoming of 1-bit Prediction

100

Initial state

Time	Buffer State	Outcome	Result?
1	Not taken	Taken	Wrong

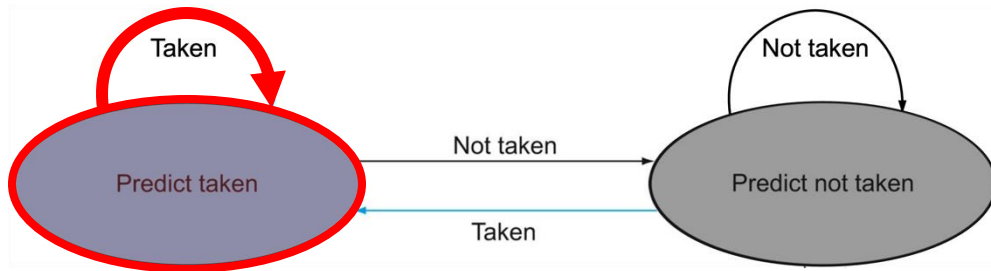
```
for (i = 0, i < 100; i++)  
  for (j = 0, j < 3; j++)  
    loop body
```



# A Performance Shortcoming of 1-bit Prediction

```
for (i = 0, i < 100; i++)  
  for (j = 0, j < 3; j++)  
    loop body
```

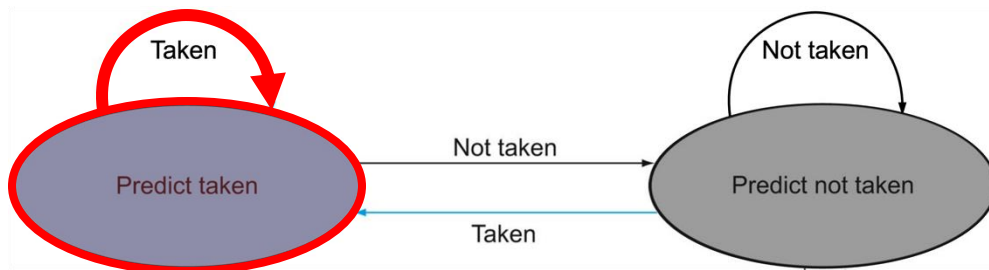
Time	Buffer State	Outcome	Result?
1	Not taken	Taken	Wrong
2	Taken	Taken	Correct



# A Performance Shortcoming of 1-bit Prediction

```
for (i = 0, i < 100; i++)  
  for (j = 0, j < 3; j++)  
    loop body
```

Time	Buffer State	Outcome	Result?
1	Not taken	Taken	Wrong
2	Taken	Taken	Correct
3	Taken	Taken	Correct

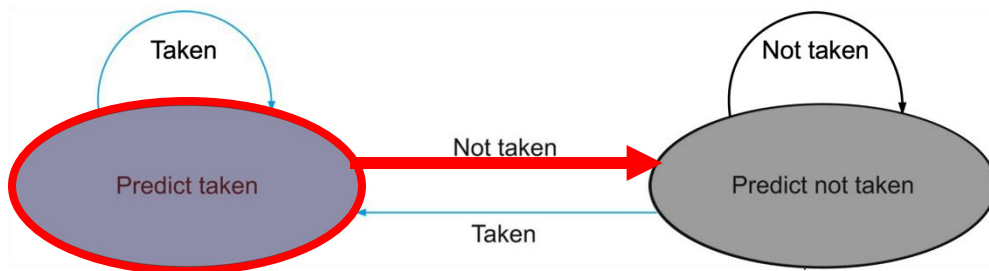


# A Performance Shortcoming of 1-bit Prediction

```
for (i = 0, i < 100; i++)  
  for (j = 0, j < 3; j++)  
    loop body
```

Time	Buffer State	Outcome	Result?
1	Not taken	Taken	Wrong
2	Taken	Taken	Correct
3	Taken	Taken	Correct
4	Taken	Not taken	Wrong

Mispredict as taken on last iteration of inner loop

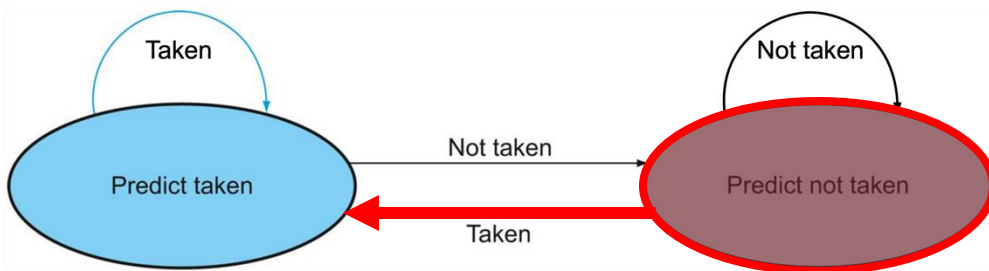


# A Performance Shortcoming of 1-bit Prediction

```
for (i = 0, i < 100; i++)  
  for (j = 0, j < 3; j++)  
    loop body
```

Time	Buffer State	Outcome	Result?
1	Not taken	Taken	Wrong
2	Taken	Taken	Correct
3	Taken	Taken	Correct
4	Taken	Not taken	Wrong
5	Not taken	Taken	Wrong

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

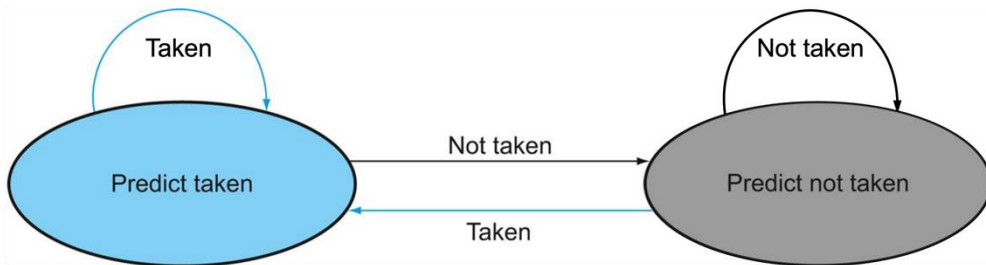




# A Performance Shortcoming of 1-bit Prediction

**Problem:** Inner loop branches mispredicted twice!

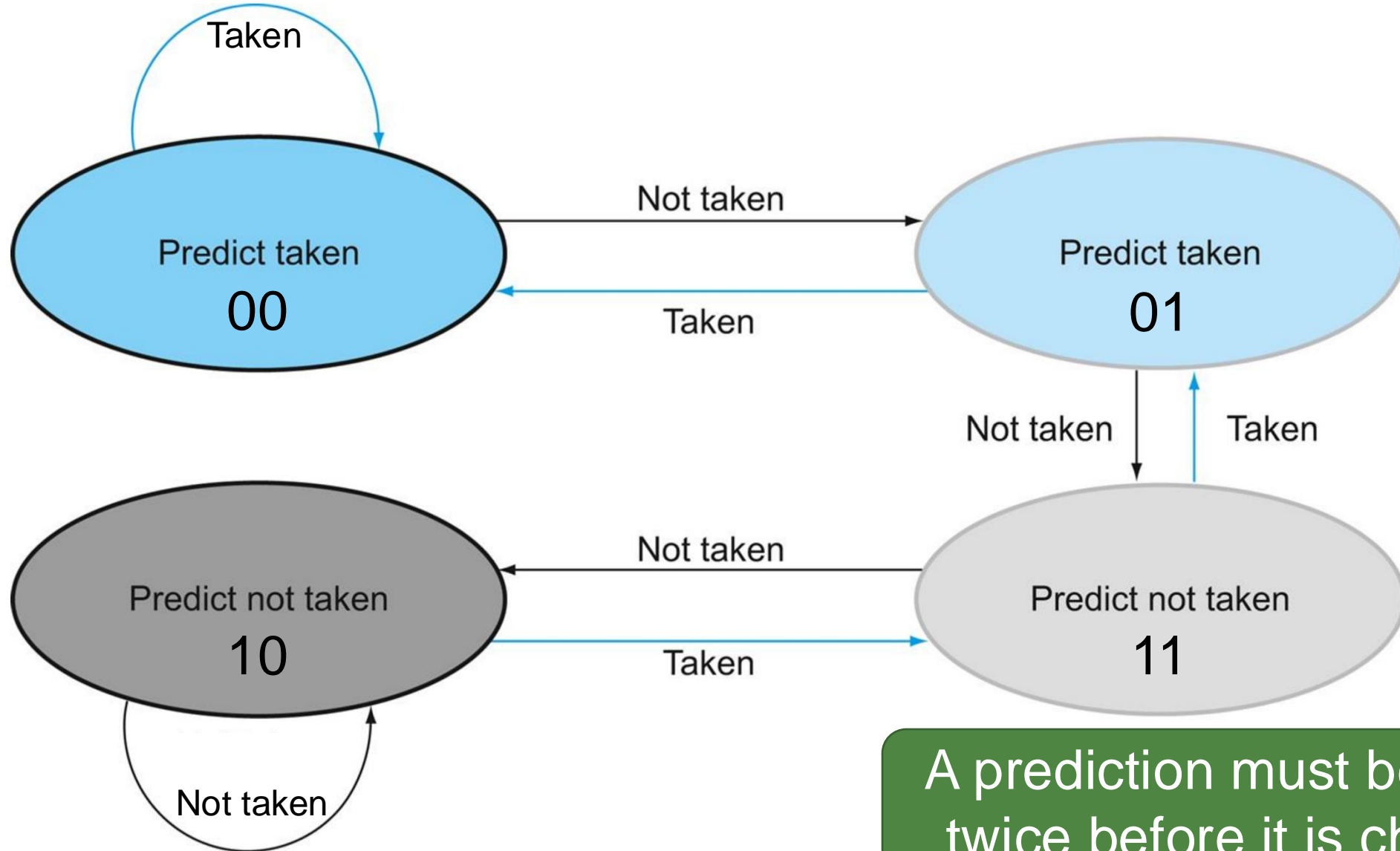
```
for (i = 0, i < 100; i++)  
  for (j = 0, j < 3; j++)  
    loop body
```



Time	Buffer State	Outcome	Result?
1	Not taken	Taken	Wrong
2	Taken	Taken	Correct
3	Taken	Taken	Correct
4	Taken	Not taken	Wrong
5	Not taken	Taken	Wrong
6	Taken	Taken	Correct
7	Taken	Taken	Correct
8	Taken	Not taken	Wrong
9	Not taken	Taken	Wrong
10	Taken	Taken	Correct
11	Taken	Taken	Correct
12	Taken	Not taken	Wrong
...	...	...	...

# Dynamic Branch Prediction: 2-bit Prediction

106



A prediction must be wrong twice before it is changed

# Dynamic Branch Prediction: 2-bit Prediction

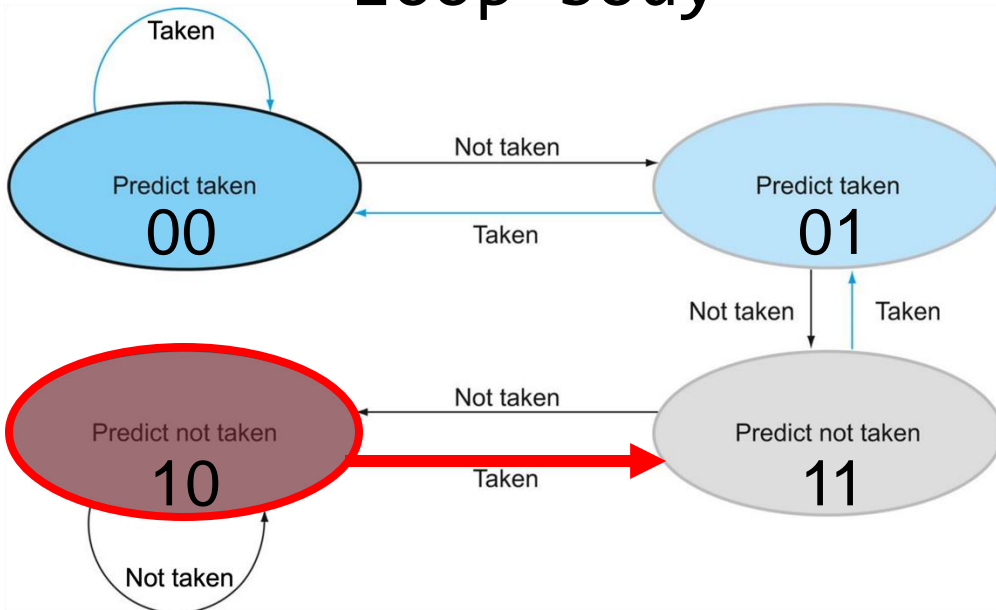
107

One misprediction each loop execution

Initial state

Time	Buffer State	Outcome	Result?
1	Not taken (10)	Taken	Wrong

```
for (i = 0, i < 100; i++)  
  for (j = 0, j < 3; j++)  
    loop body
```



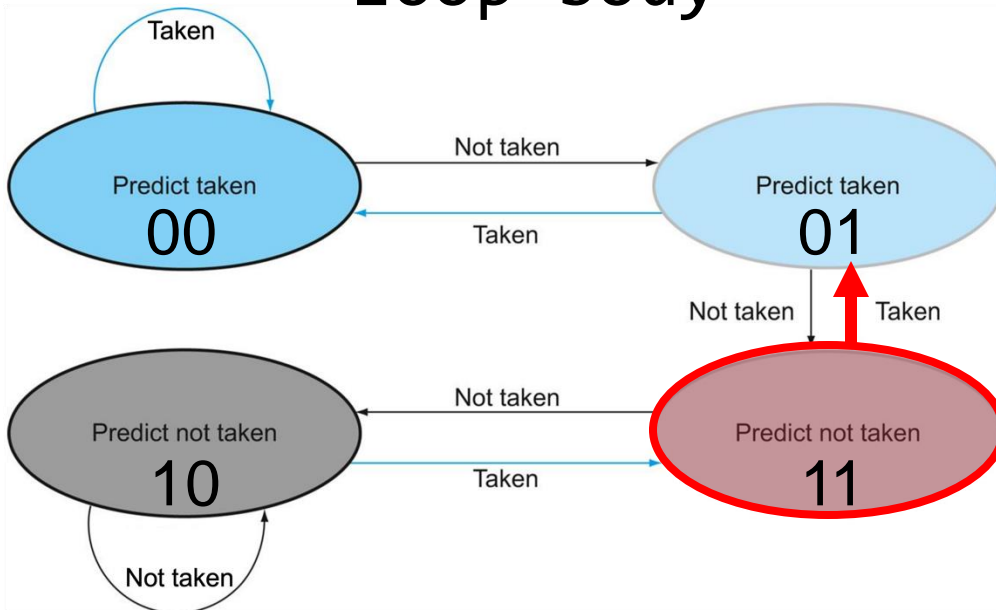
# Dynamic Branch Prediction: 2-bit Prediction

108

One misprediction each loop execution

```
for (i = 0, i < 100; i++)  
  for (j = 0, j < 3; j++)  
    loop body
```

Time	Buffer State	Outcome	Result?
1	Not taken (10)	Taken	Wrong
2	Not taken (11)	Taken	Wrong



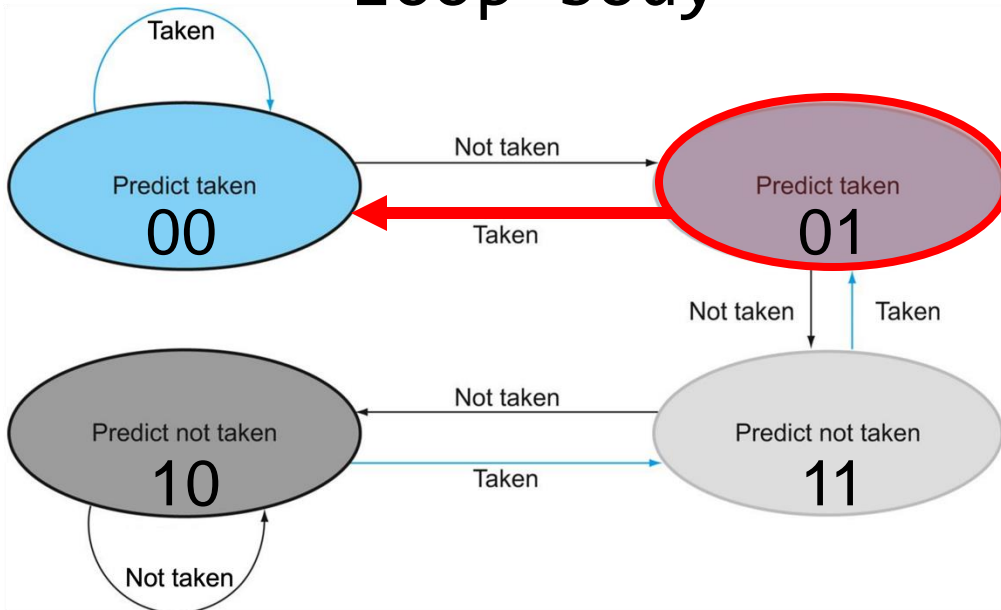
# Dynamic Branch Prediction: 2-bit Prediction

109

One misprediction each loop execution

```
for (i = 0, i < 100; i++)  
  for (j = 0, j < 3; j++)  
    loop body
```

Time	Buffer State	Outcome	Result?
1	Not taken (10)	Taken	Wrong
2	Not taken (11)	Taken	Wrong
3	Taken (01)	Taken	Correct



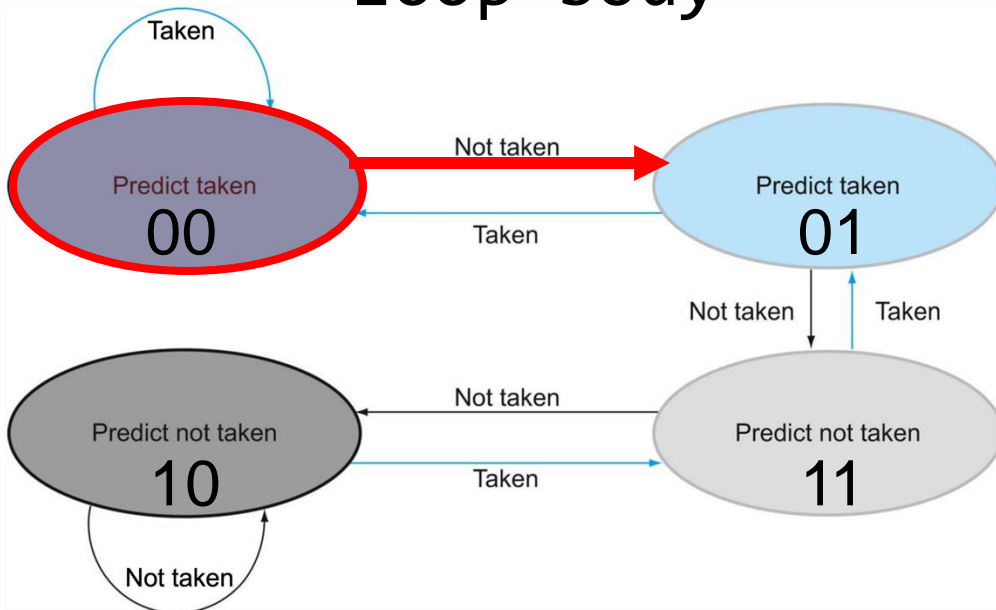
# Dynamic Branch Prediction: 2-bit Prediction

110

One misprediction each loop execution

```
for (i = 0, i < 100; i++)  
  for (j = 0, j < 3; j++)  
    loop body
```

Time	Buffer State	Outcome	Result?
1	Not taken (10)	Taken	Wrong
2	Not taken (11)	Taken	Wrong
3	Taken (01)	Taken	Correct
4	Taken (00)	Not taken	Wrong



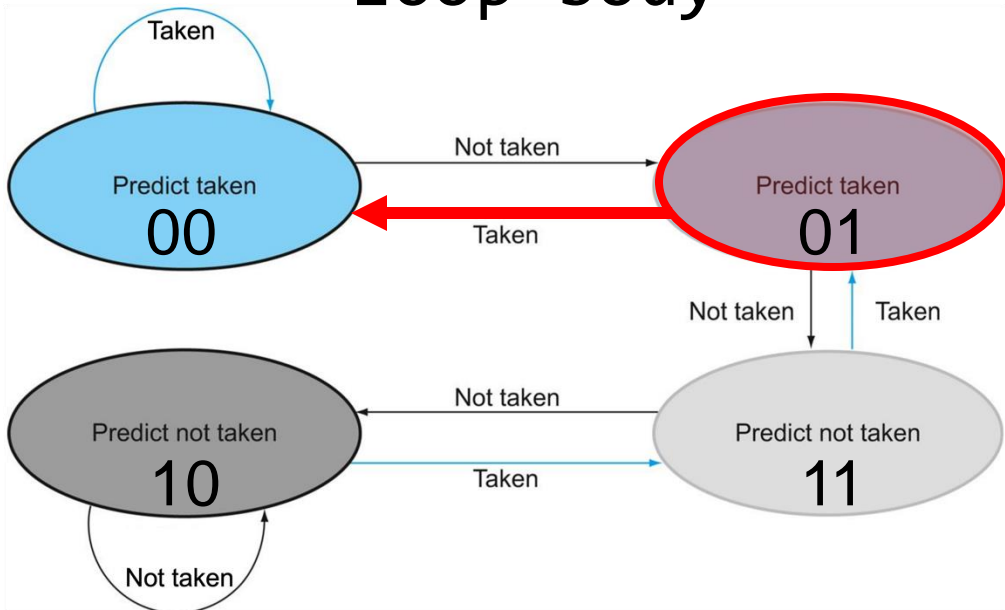
# Dynamic Branch Prediction: 2-bit Prediction

111

One misprediction each loop execution

```
for (i = 0, i < 100; i++)  
  for (j = 0, j < 3; j++)  
    loop body
```

Time	Buffer State	Outcome	Result?
1	Not taken (10)	Taken	Wrong
2	Not taken (11)	Taken	Wrong
3	Taken (01)	Taken	Correct
4	Taken (00)	Not taken	Wrong
5	Taken (01)	Taken	Correct



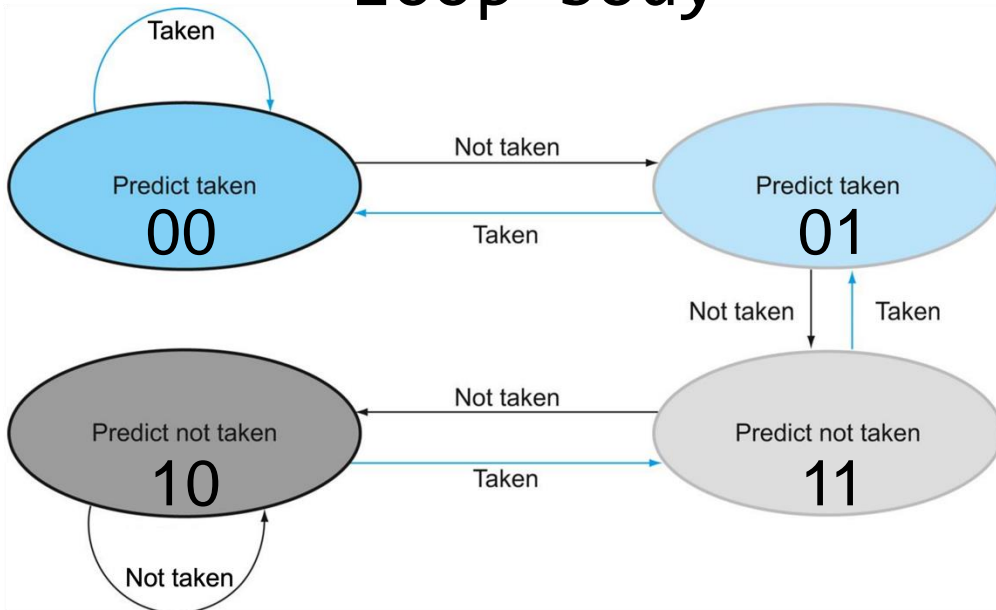
# Dynamic Branch Prediction: 2-bit Prediction

112

One misprediction each loop execution

One misprediction each loop execution

```
for (i = 0, i < 100; i++)  
  for (j = 0, j < 3; j++)  
    loop body
```



Time	Buffer State	Outcome	Result?
1	Not taken (10)	Taken	Wrong
2	Not taken (11)	Taken	Wrong
3	Taken (01)	Taken	Correct
4	Taken (00)	Not taken	Wrong
5	Taken (01)	Taken	Correct
6	Taken (00)	Taken	Correct
7	Taken (00)	Taken	Correct
8	Taken (00)	Not taken	Wrong
9	Taken (01)	Taken	Correct
10	Taken (00)	Taken	Correct
11	Taken (00)	Taken	Correct
12	Taken (00)	Not taken	Wrong
...	...	...	...



# Solution for Control Hazard: Branch Prediction

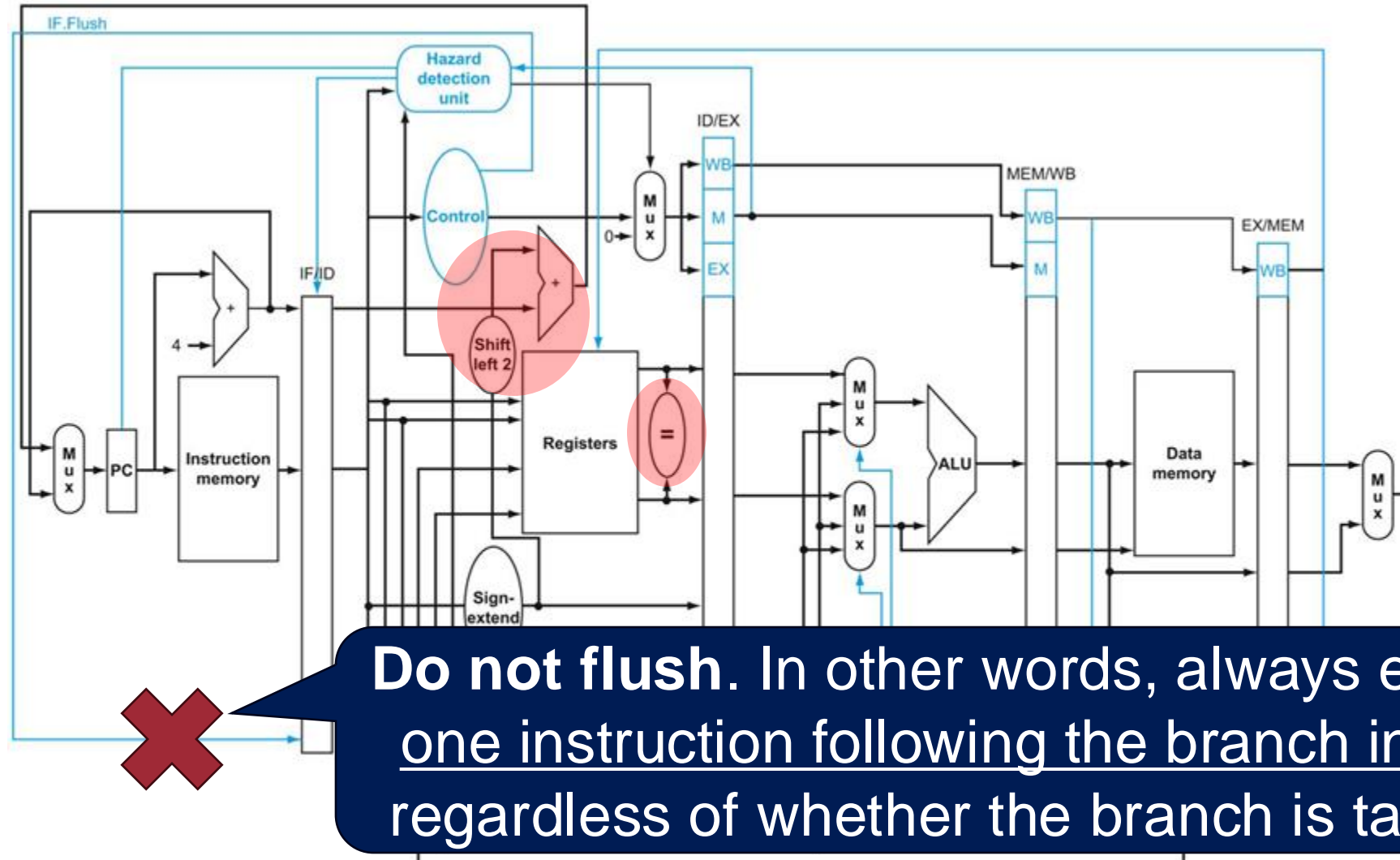


*Prediction is effective, but we don't have a hardware expert. In this situation, is there any way to resolve the control hazard in software manner?*

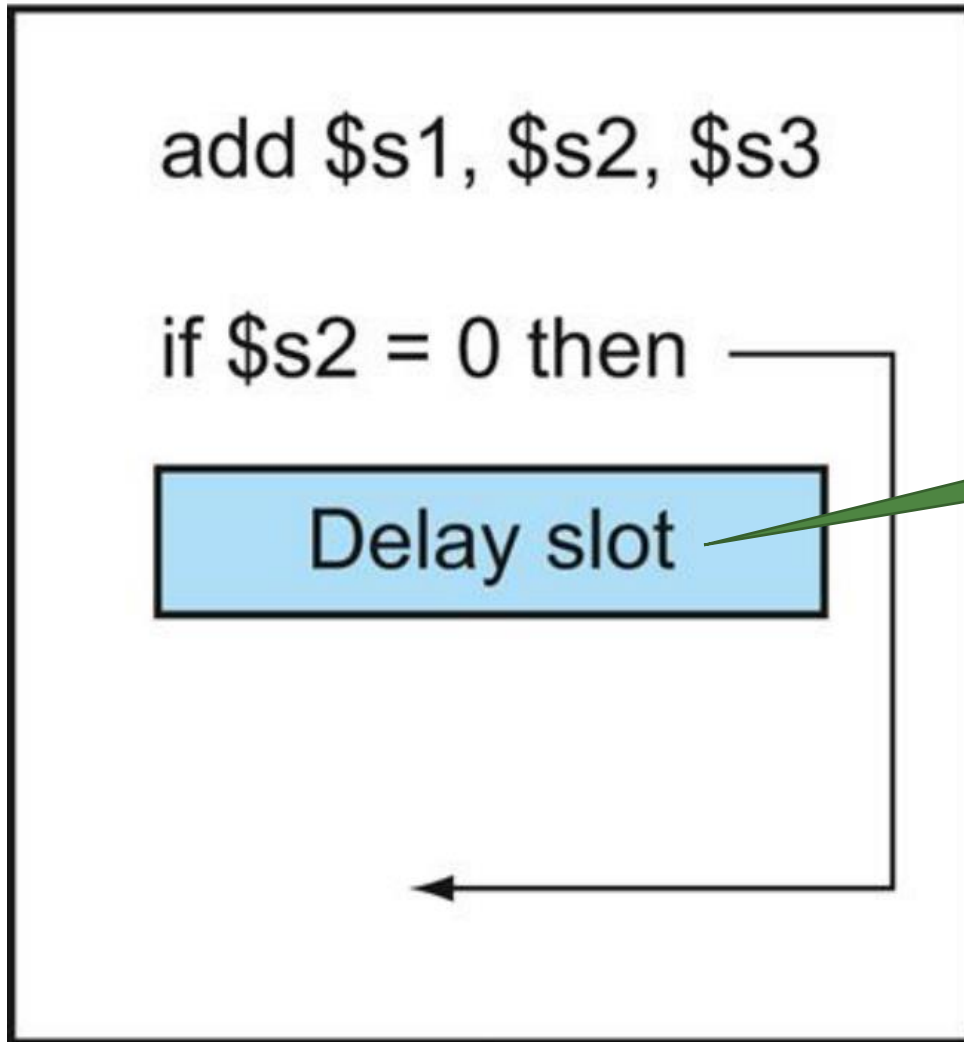
**Delayed Branch!**  
(Compiler-driven optimization)

# Our Assumption on HW: Do not Flush

114



# Solution for Control Hazard: Delayed Branch



Compiler insert the **branch delay slot** with instruction that is not affected by the branch

# Solution for Control Hazard: Delayed Branch

116



add \$s1, \$s2, \$s3

if \$s2 = 0 then

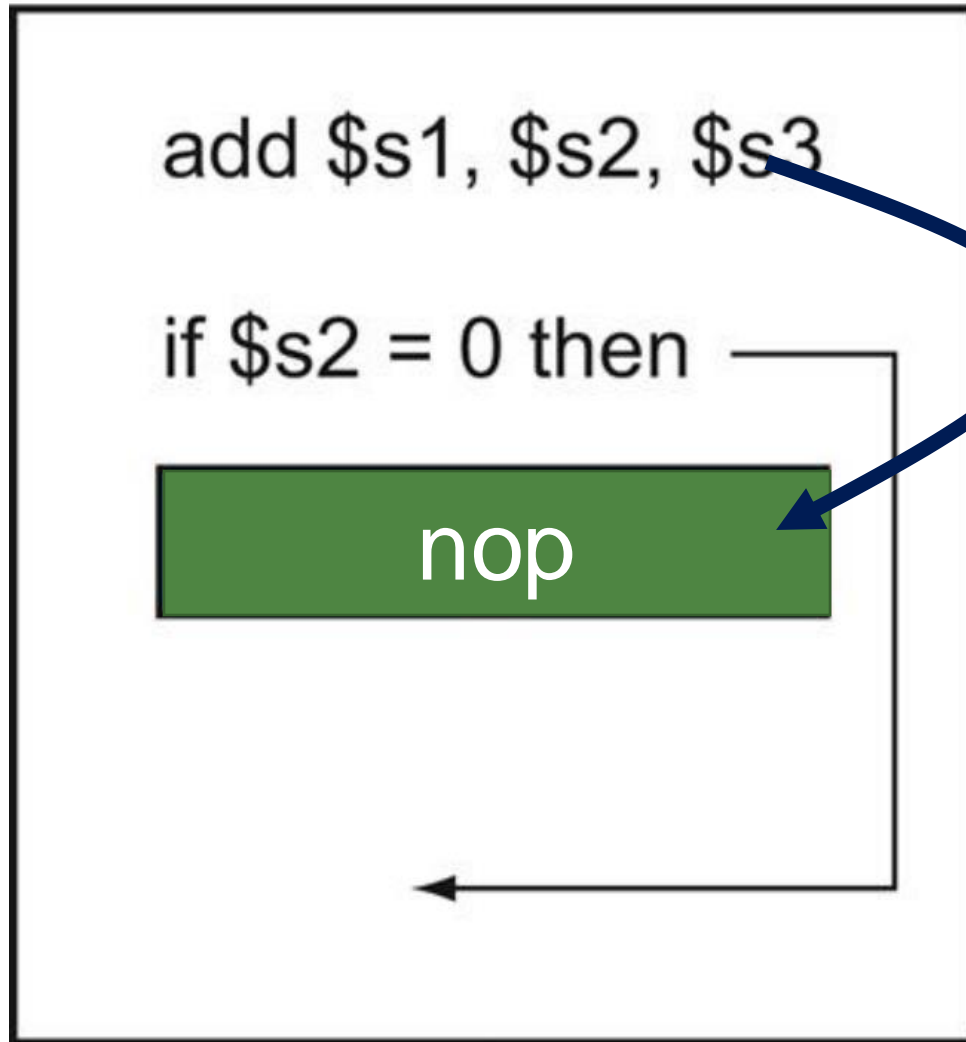
nop

Compiler insert the **branch delay slot** with instruction that is not affected by the branch

*The simplest method is to insert a nop instruction (1 cycle penalty)*

# Solution for Control Hazard: Delayed Branch

(with advanced compiler)

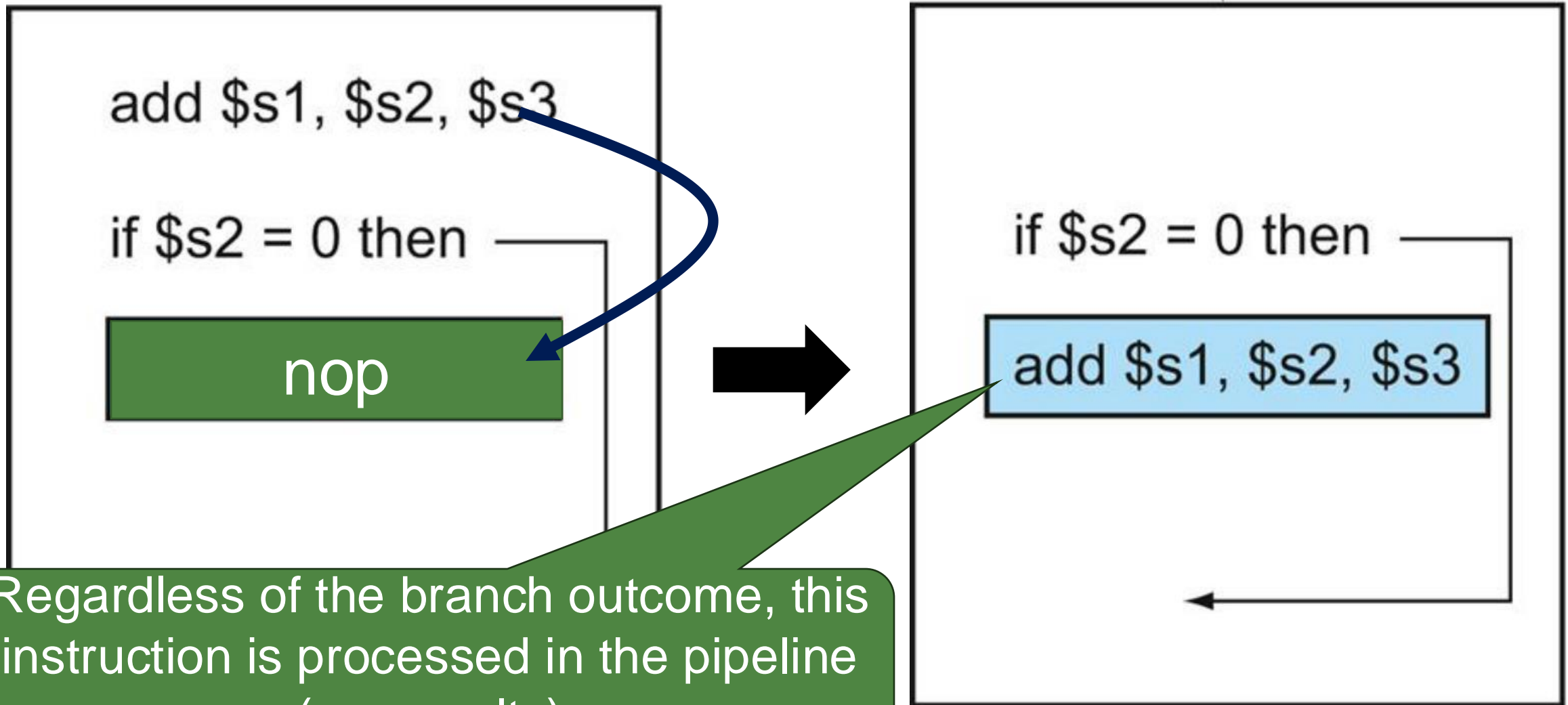


*Code reordering  
(move this instruction, which is  
not affected by the branch)*

# Solution for Control Hazard: Delayed Branch

(with advanced compiler)

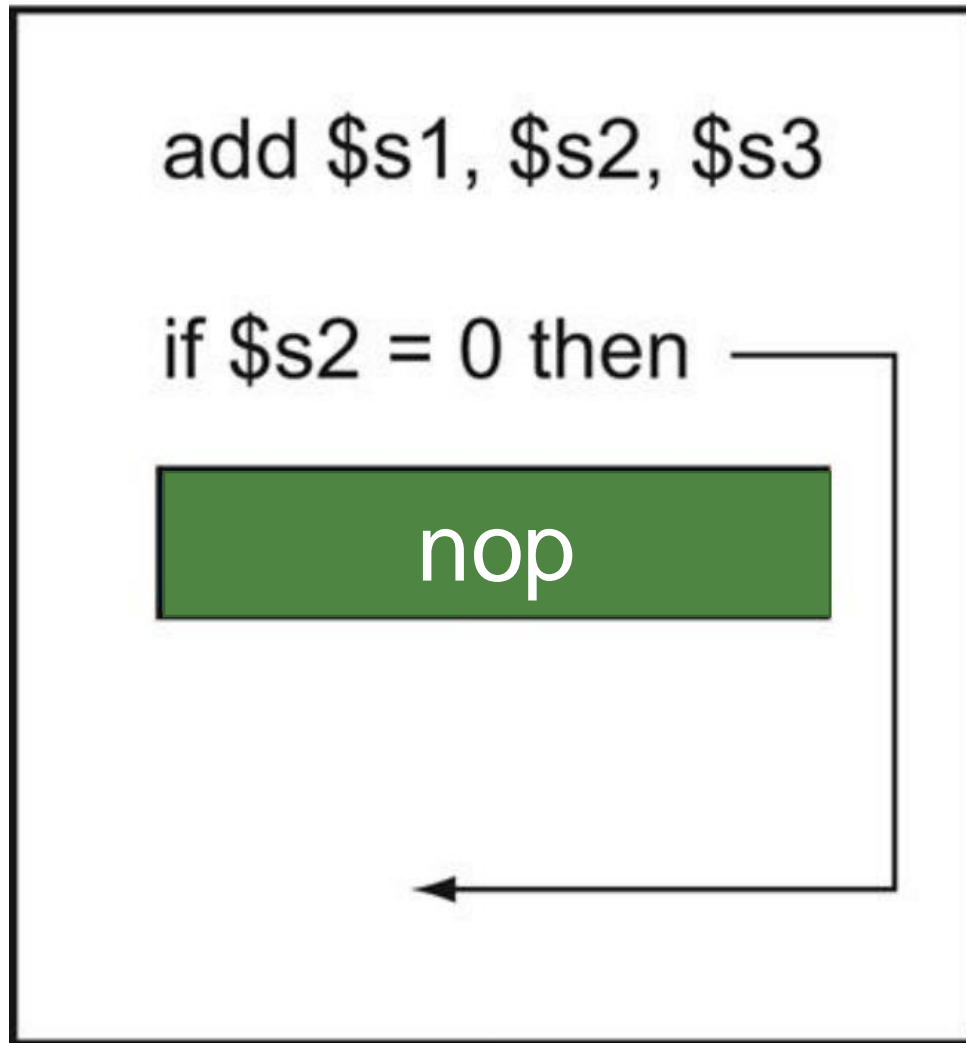
118



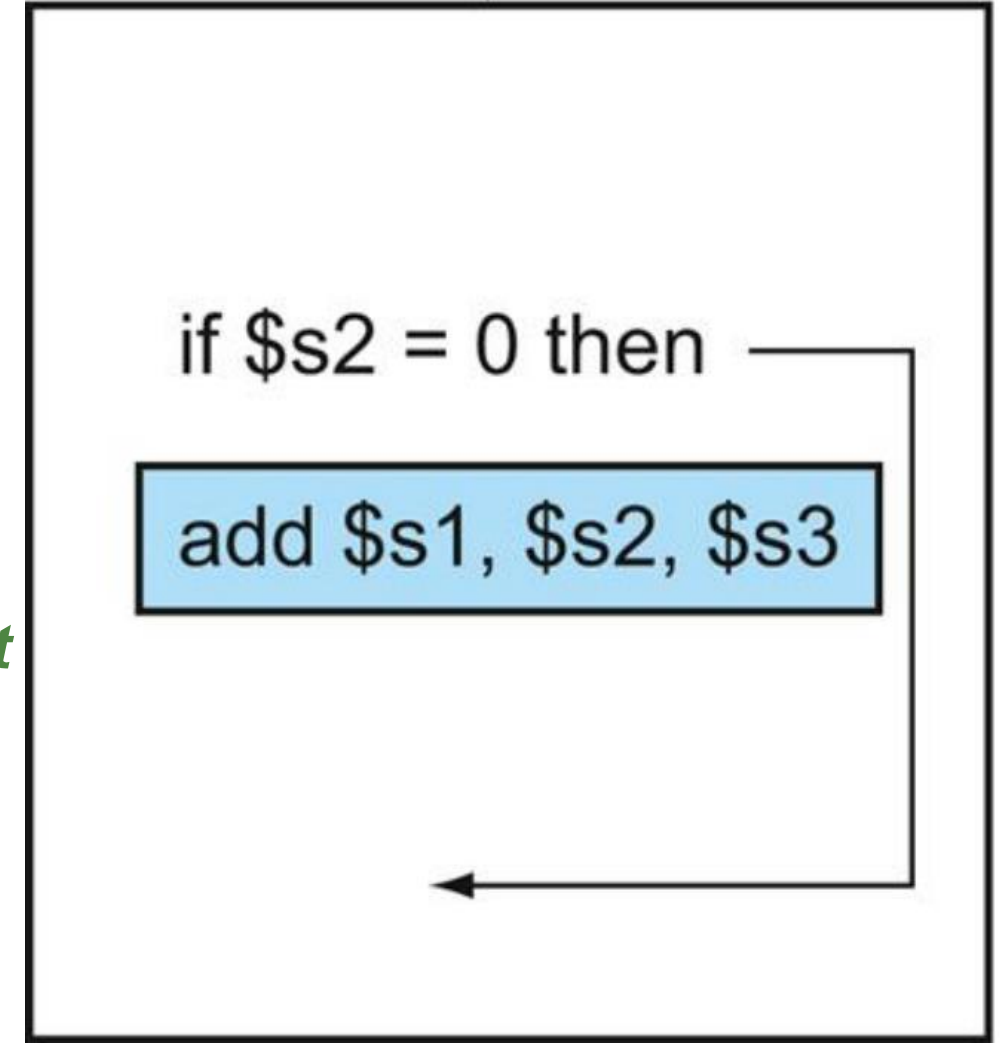
Regardless of the branch outcome, this instruction is processed in the pipeline (no penalty)

# Solution for Control Hazard: Delayed Branch

(with advanced compiler)



**=**  
*Equivalent meaning*



# Pipelining Hazards Summary



Situations that prevent starting the next instruction in the next cycle

## Hazard #1: Structural hazard

Conflict for use of a hardware resource

### Solution:

- Stall
- Resource duplication

## Hazard #2: Data hazard

An instruction cannot execute because data is not yet available

### Solution:

- Stall
- Forwarding
- Compiler optimization

## Hazard #3: Control hazard

The next instruction is uncertain due to branching

### Solution:

- Stall
- Optimized branch processing
- Branch prediction
- Delayed branch



**Question?**