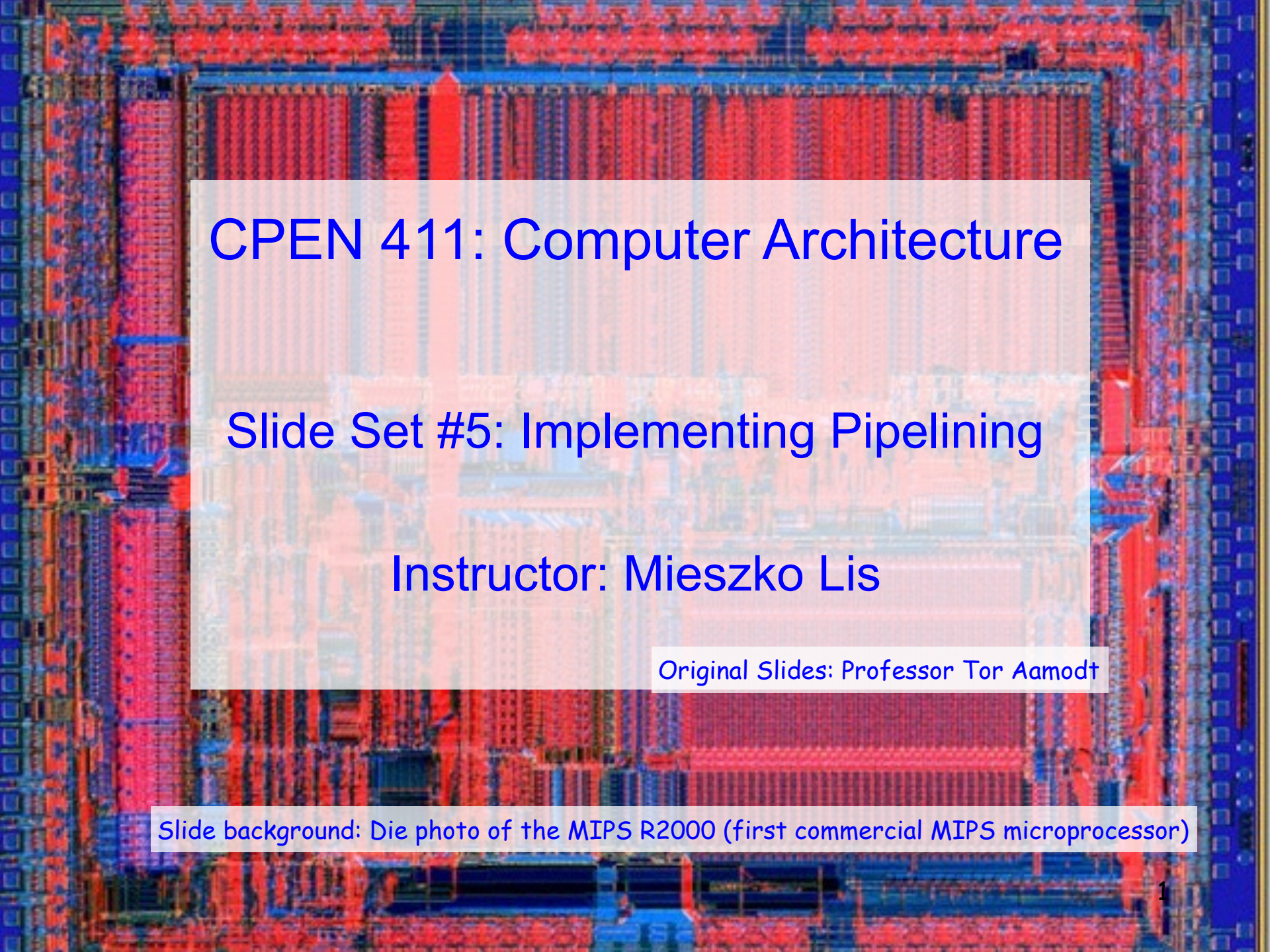


CPEN 411: Computer Architecture

Slide Set #5: Implementing Pipelining

Instructor: Mieszko Lis

Original Slides: Professor Tor Aamodt

Slide background: Die photo of the MIPS R2000 (first commercial MIPS microprocessor)

Introduction to Slide Set #5

In the last slide set we learned about pipelining at a high level.

In this slide set we will look at some important details of how to implement pipelining.

Today, an architect should think about these type of details, but would often not model them in full detail in their simulator. More commonly, a digital hardware designer would work out the exact details presented in this slide. The architect is interested in their impact on average cycles per instruction (CPI), but the hardware designer needs to be sure the design correctly executes programs which requires more detailed information.

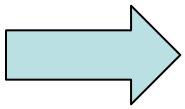
Problem

Algorithm

Program (+ OS + Network)

ISA (Instruction Set Arch)

Microarchitecture

This Slide Set 

Circuits

Electronic Devices

Learning Objectives

- After we finish this slide set you should be able to:
  - Explain the motivation for pipelined control, and how pipelined control is implemented.
  - Describe how forwarding is implemented using muxes and a forwarding control unit and explain how these operate.
  - Describe how stalls are implemented in a hardware pipeline.
  - Analyze pipeline timing using a pipeline timing diagram (which can be used to evaluate the average CPI for a specific sequence of instructions).

Pipelined Control

Pipelined Control

- Challenge: Control signals appear in each pipeline stage

Pipelined Control

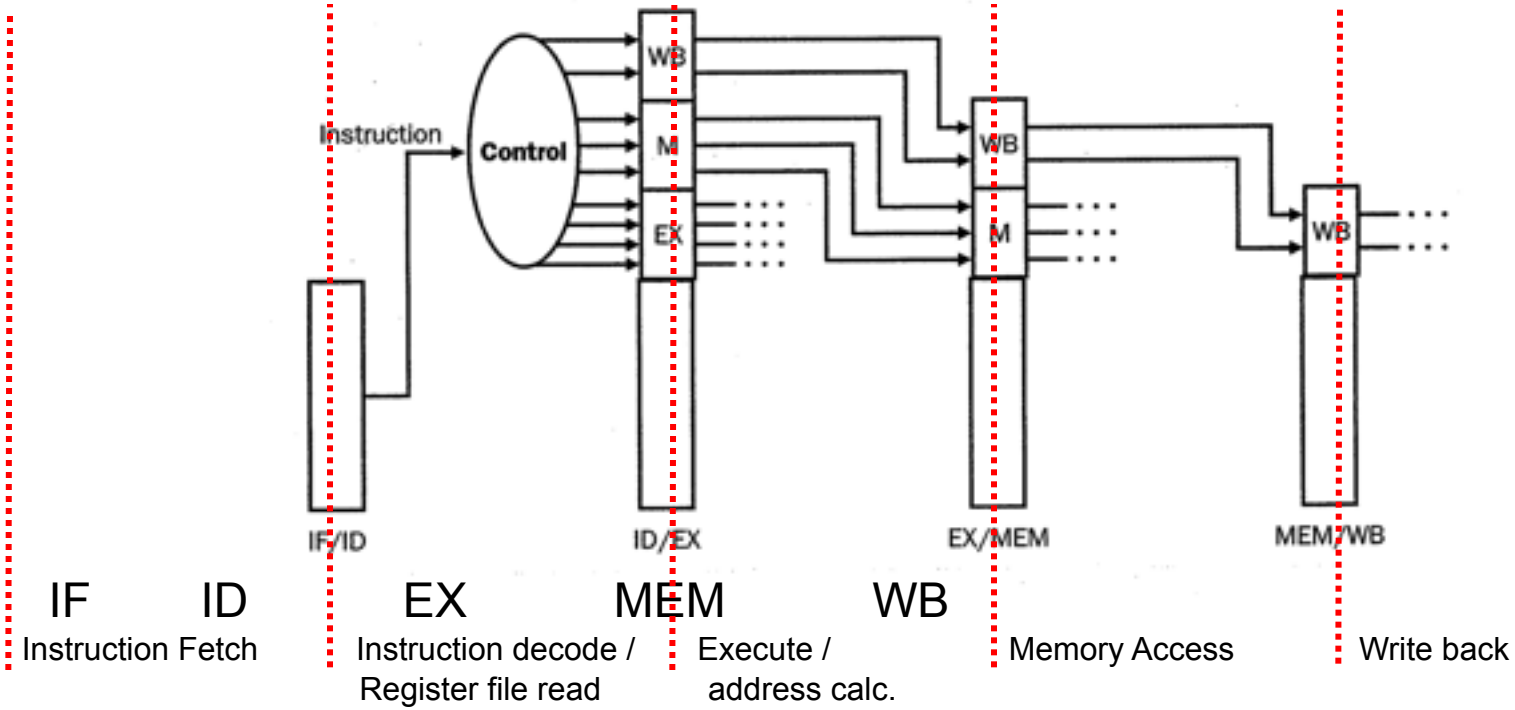
- Challenge: Control signals appear in each pipeline stage
- Review: Control strategies for non-pipelined processor:
  - Combinational Control (single-cycle CPU)
    - Cannot apply to pipelined processor since there are many instructions in pipeline
  - Sequential Control (FSM for multi-cycle CPU)
    - Cannot apply to pipelined processor since there are many instructions in pipeline

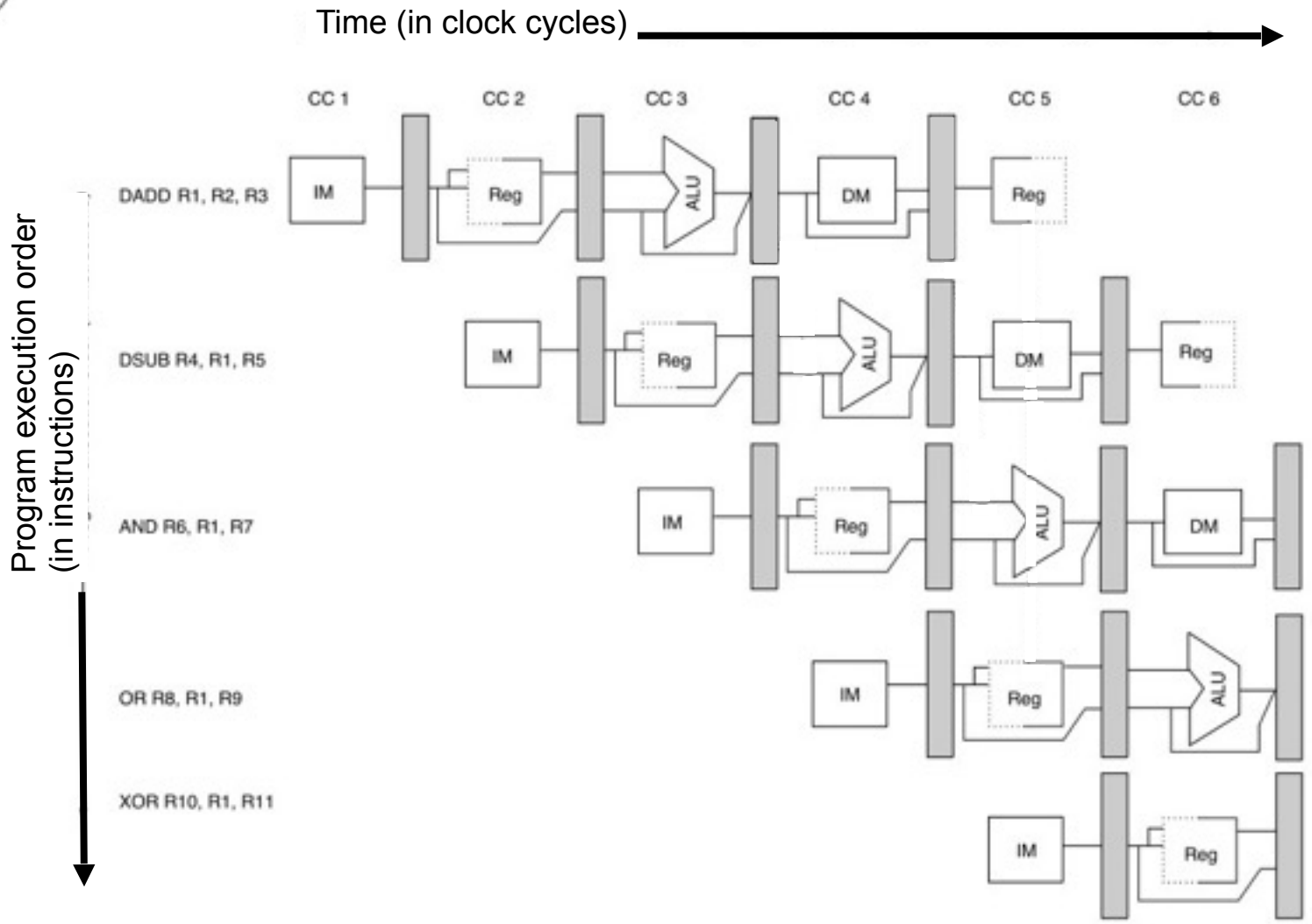
Pipelined Control

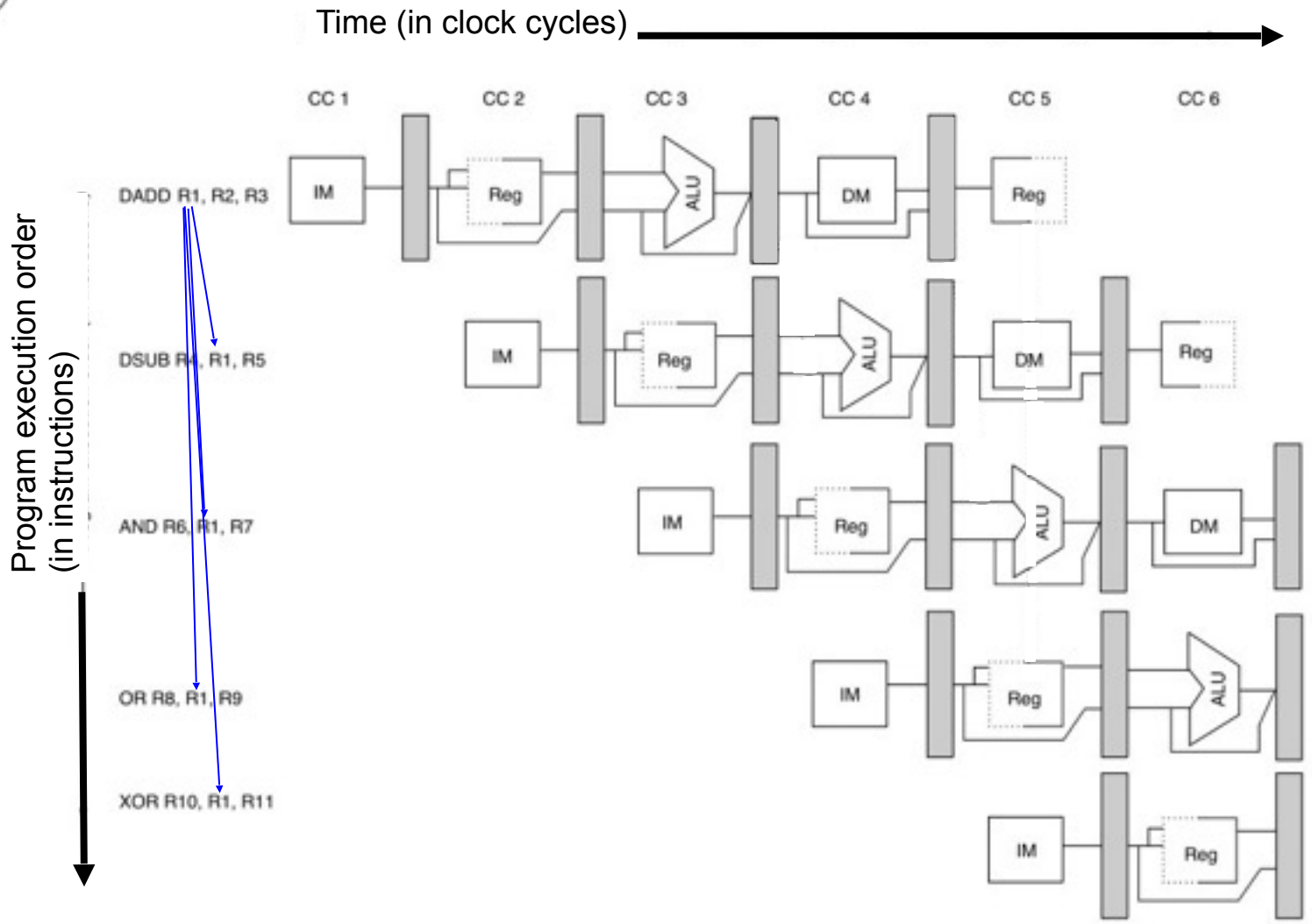
- Challenge: Control signals appear in each pipeline stage
- Review: Control strategies for non-pipelined processor:
  - Combinational Control (single-cycle CPU)
    - Cannot apply to pipelined processor since there are many instructions in pipeline
  - Sequential Control (FSM for multi-cycle CPU)
    - Cannot apply to pipelined processor since there are many instructions in pipeline
- New strategy
  - Start with combinational control
  - Pipeline it!
    - Make proper control signals flow alongside with each instruction

Pipelined Control

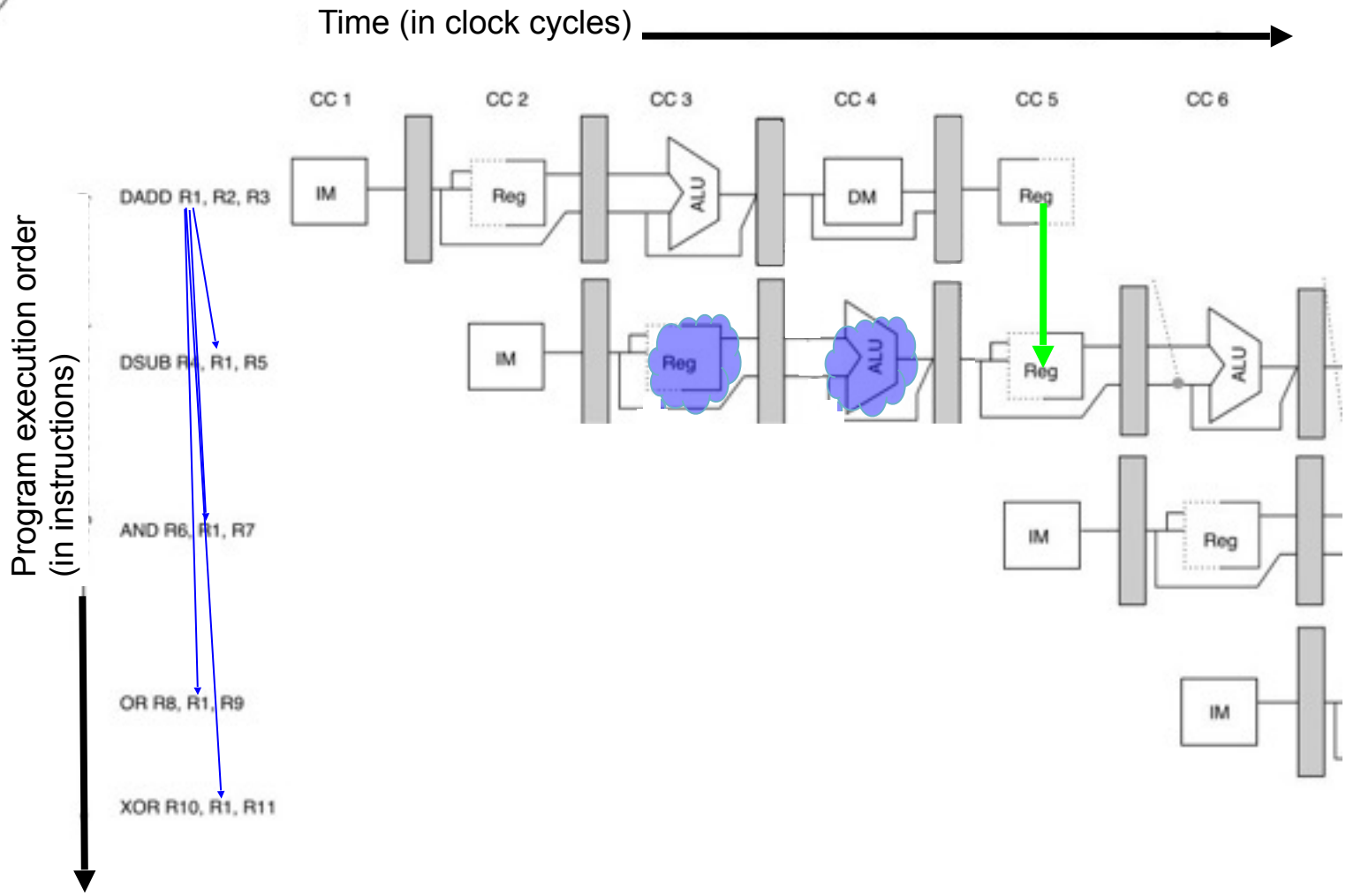
- Separate signals into groups (one per stage)
  - Each group contains all control signals for that stage
- Add pipeline registers



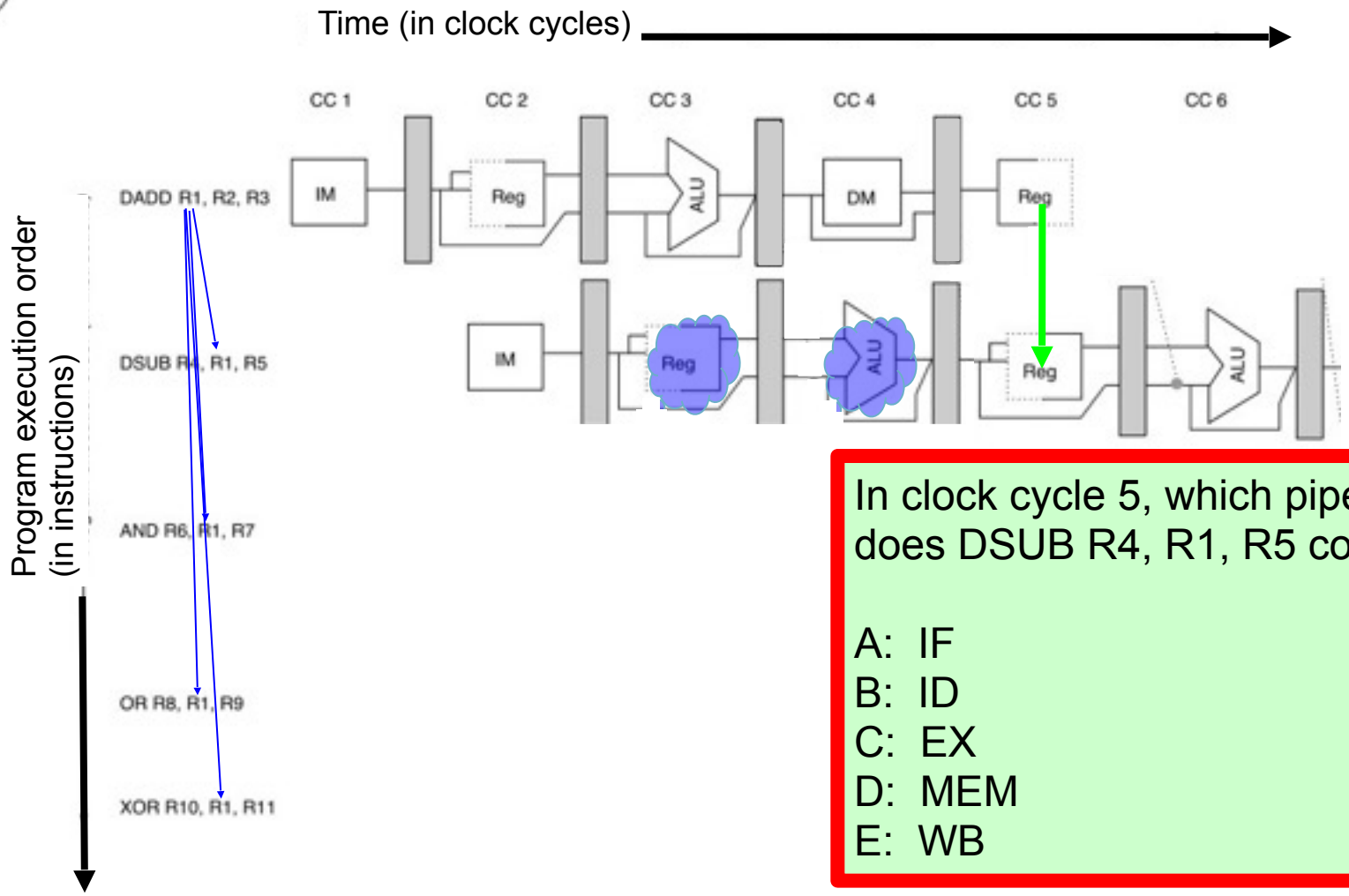

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
Program execution order (in instructions)
DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
IM
Reg
ALU
DM
Reg


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
Program execution order
(in instructions)
DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
IM
Reg
ALU
DM
Reg

Solution #1: Stall



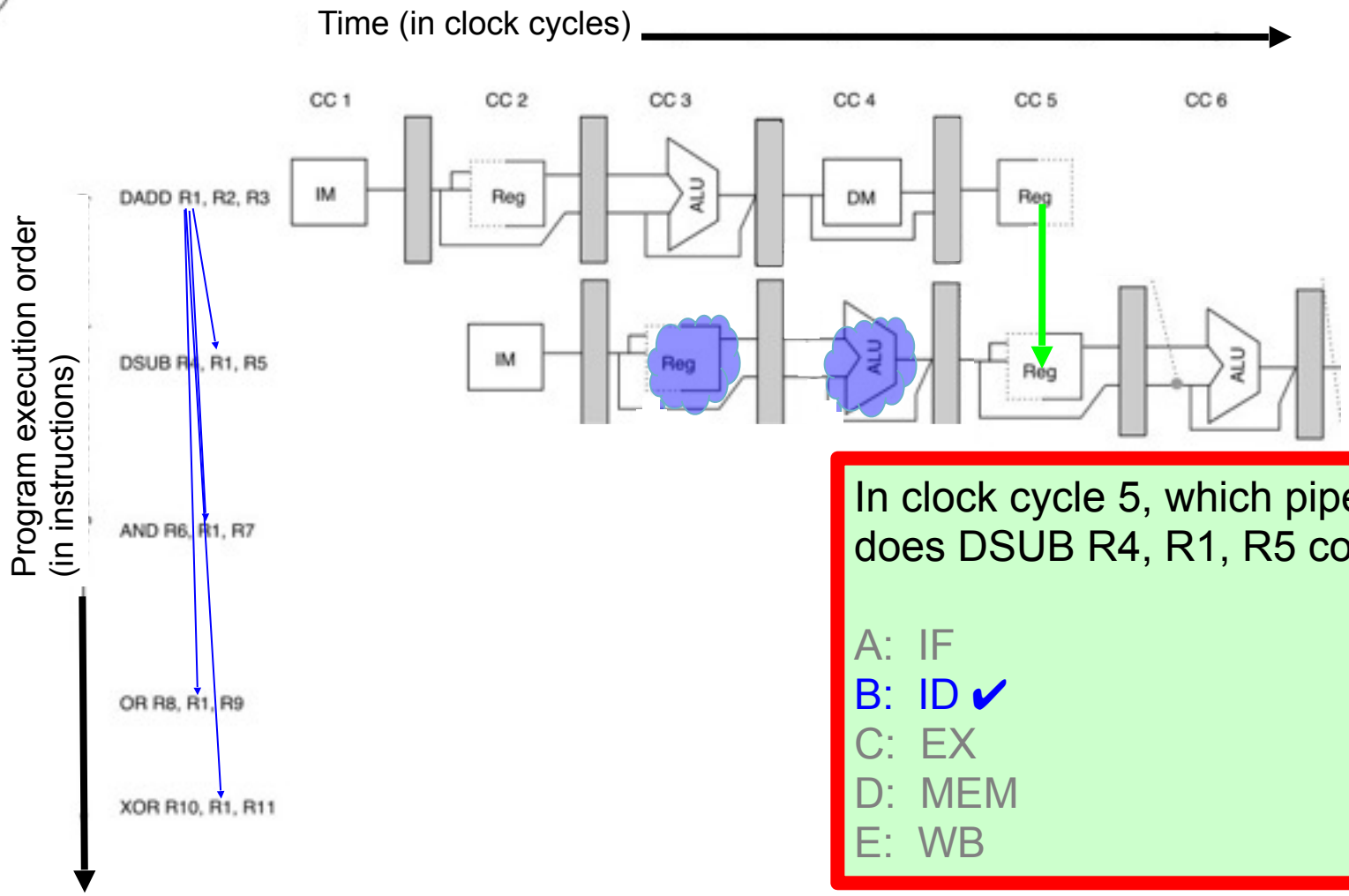
Solution #1: Stall



In clock cycle 5, which pipeline stage does DSUB R4, R1, R5 complete?

A: IF
B: ID
C: EX
D: MEM
E: WB

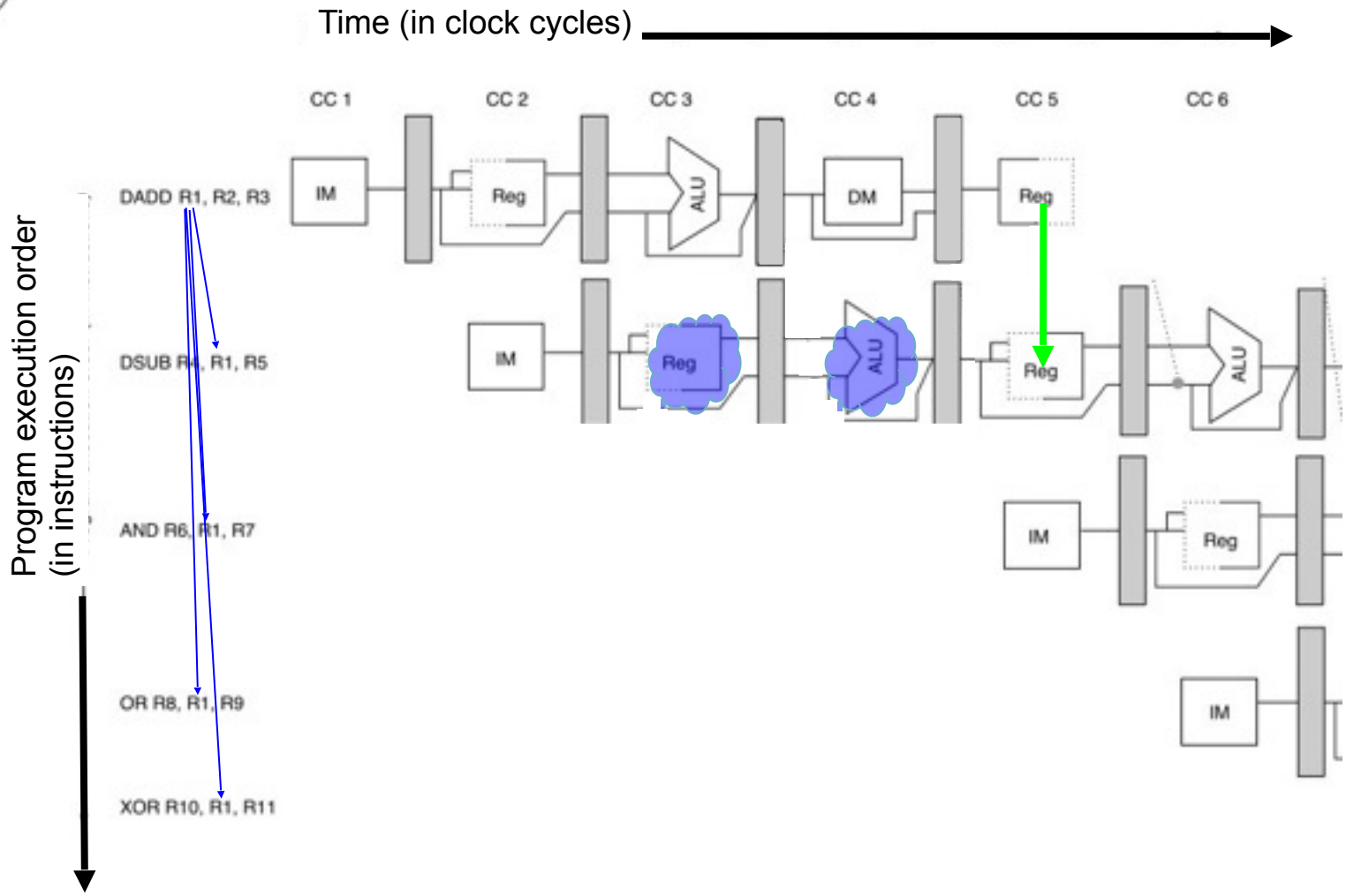
Solution #1: Stall

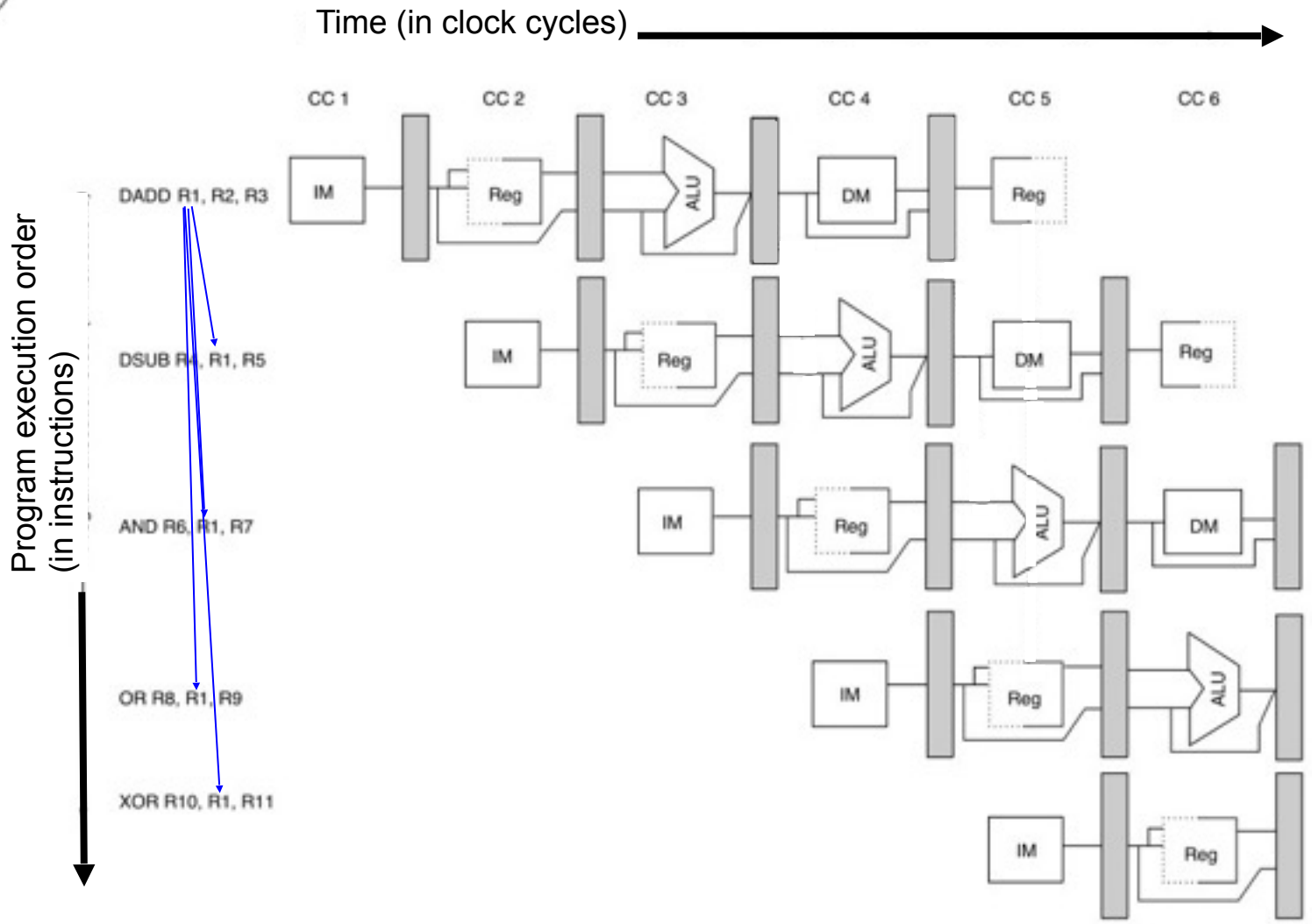


In clock cycle 5, which pipeline stage does DSUB R4, R1, R5 complete?

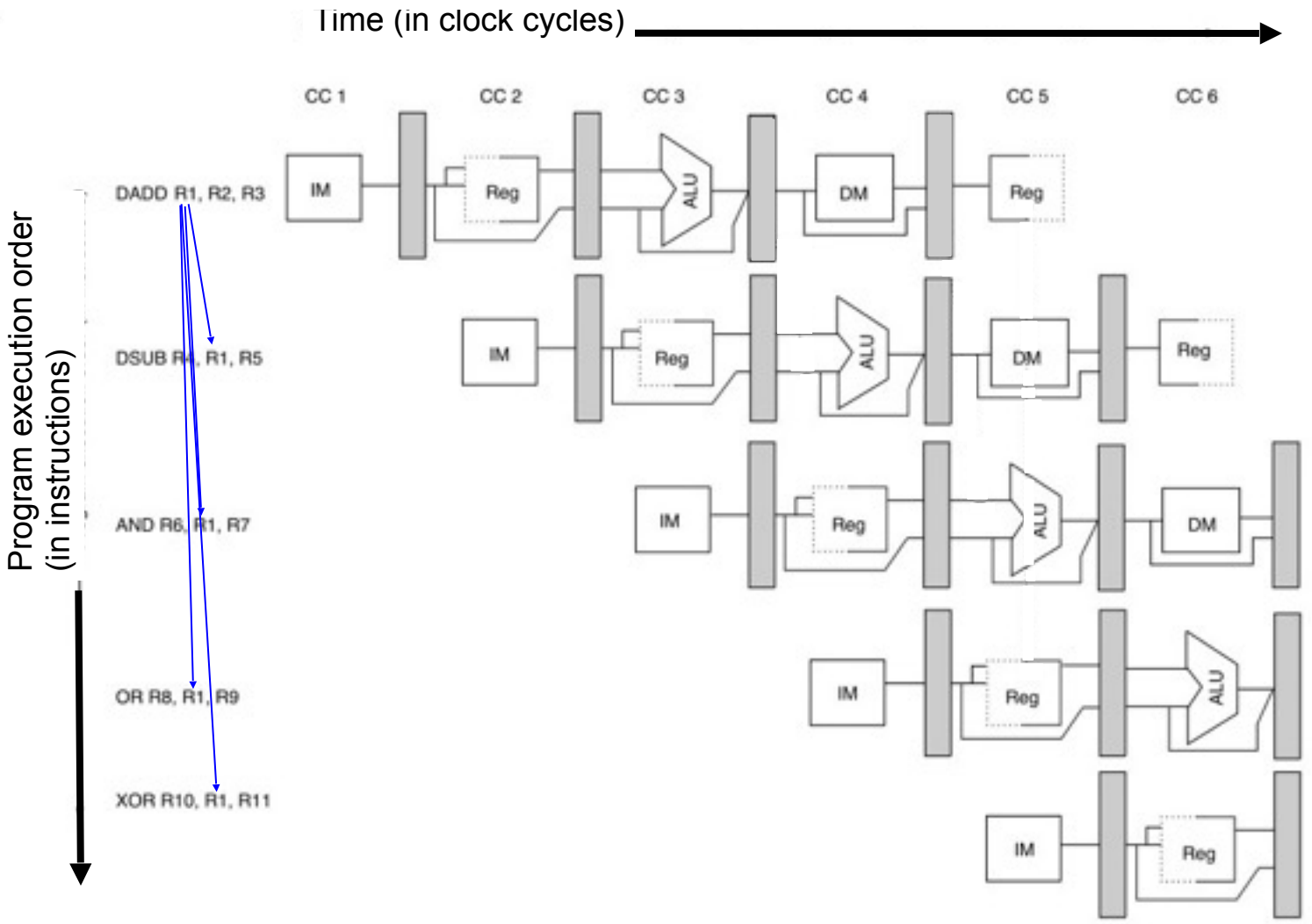
A: IF
B: ID ✔
C: EX
D: MEM
E: WB

Solution #1: Stall

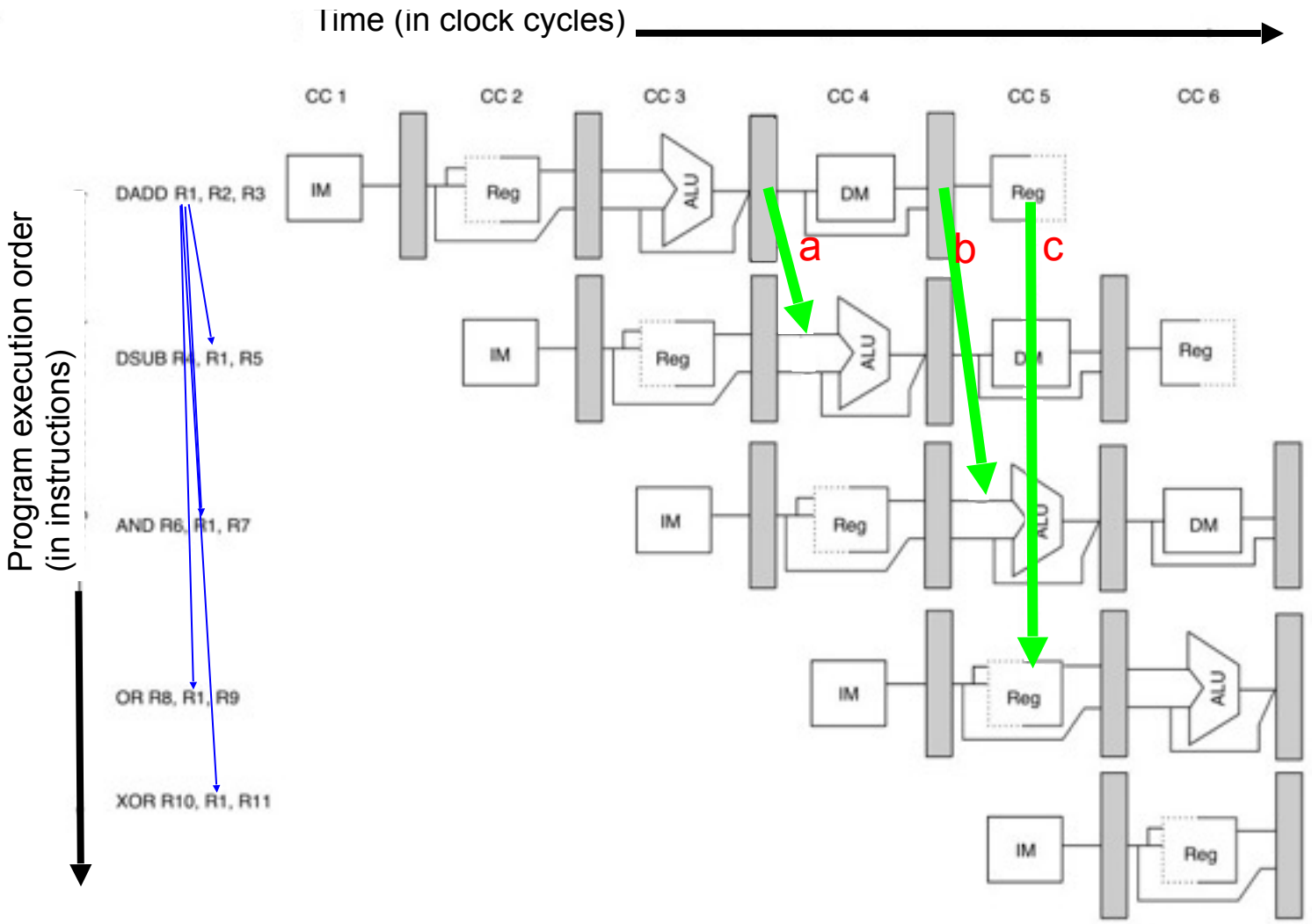



Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
Program execution order
(in instructions)
DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
IM
Reg
ALU
DM
Reg

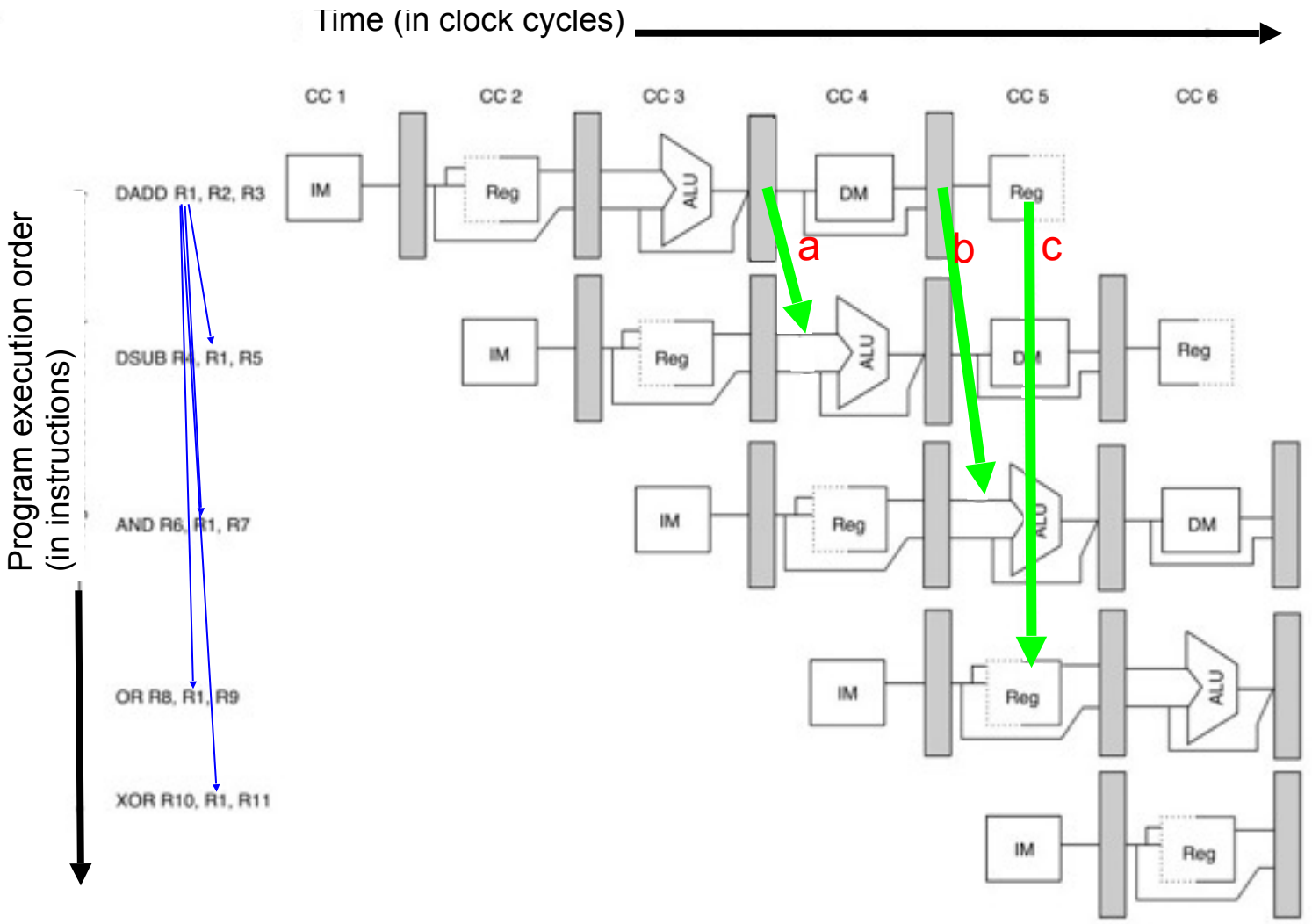
Solution #2: Forwarding



Solution #2: Forwarding

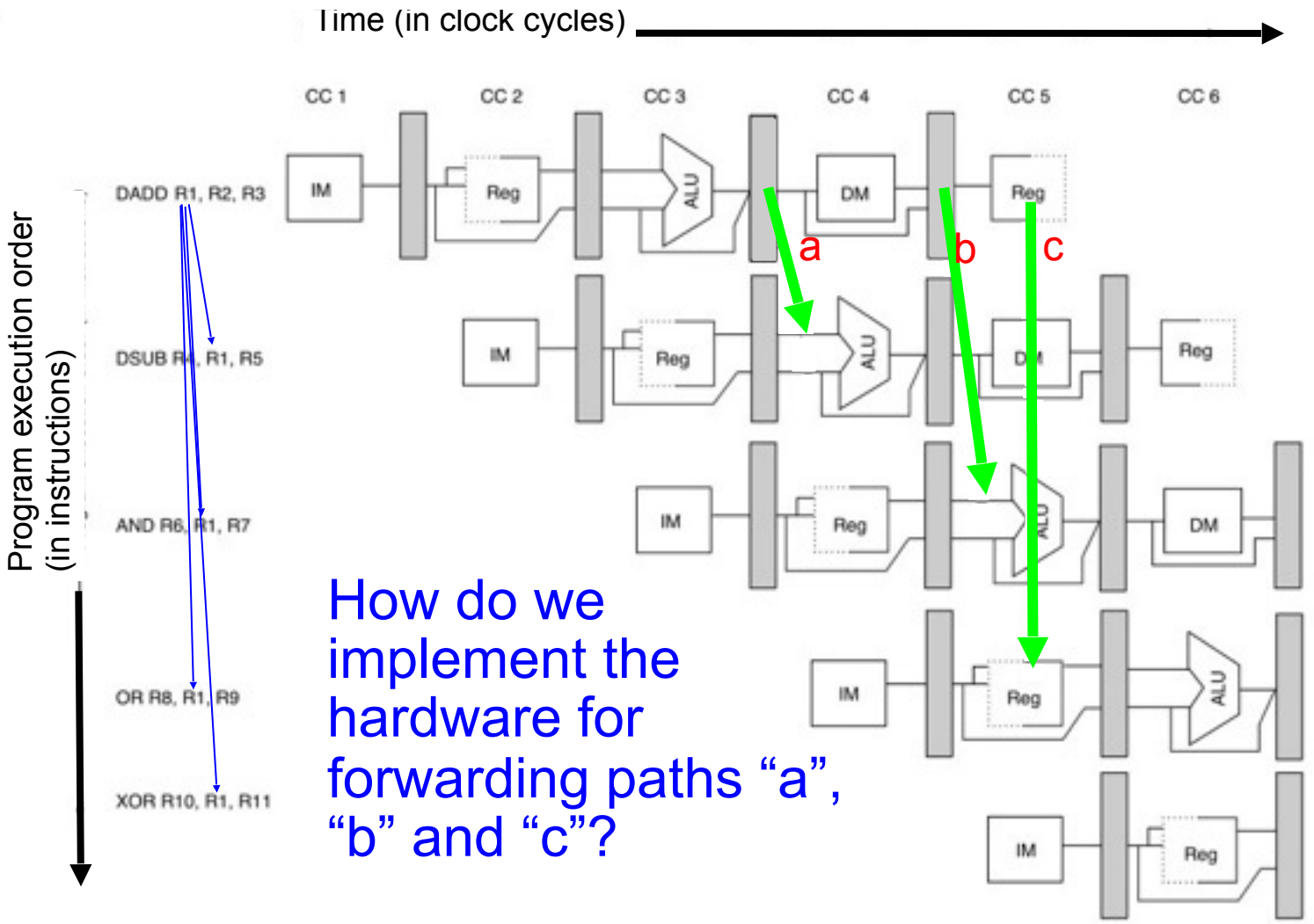


Solution #2: Forwarding



NOTE: Forwarding begins and ends within a single clock cycle

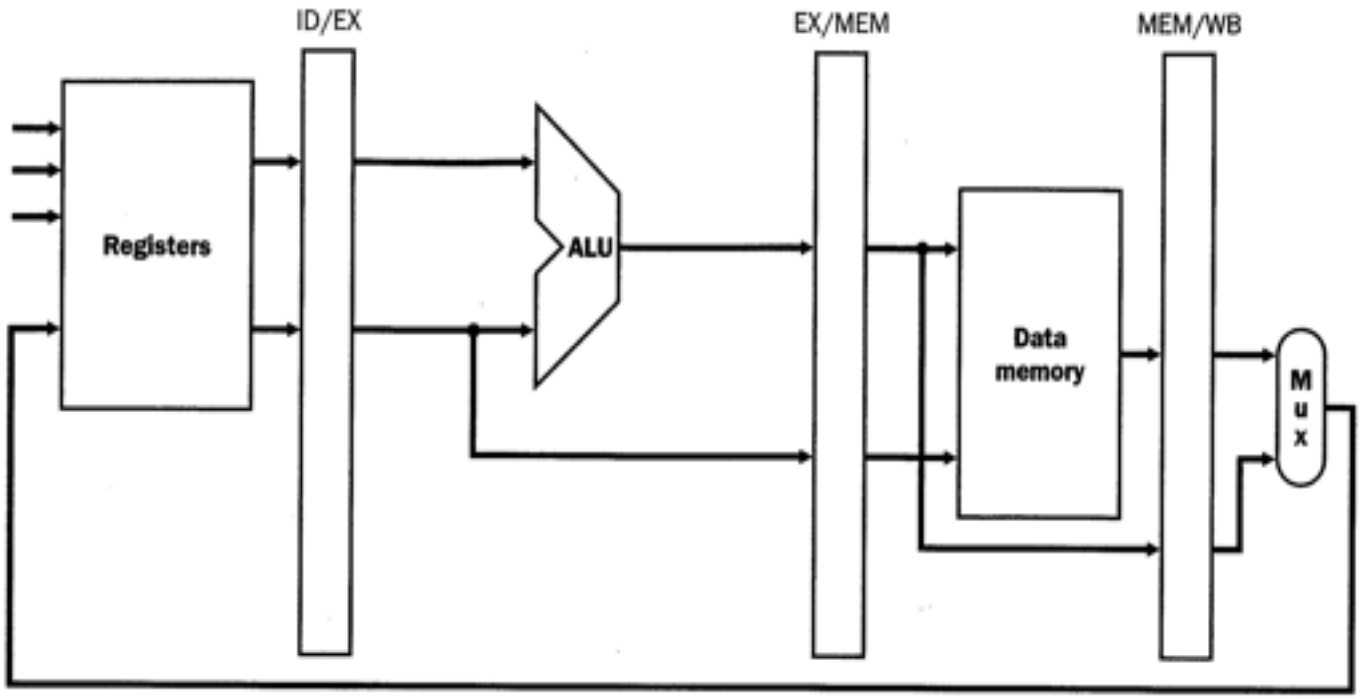
Solution #2: Forwarding



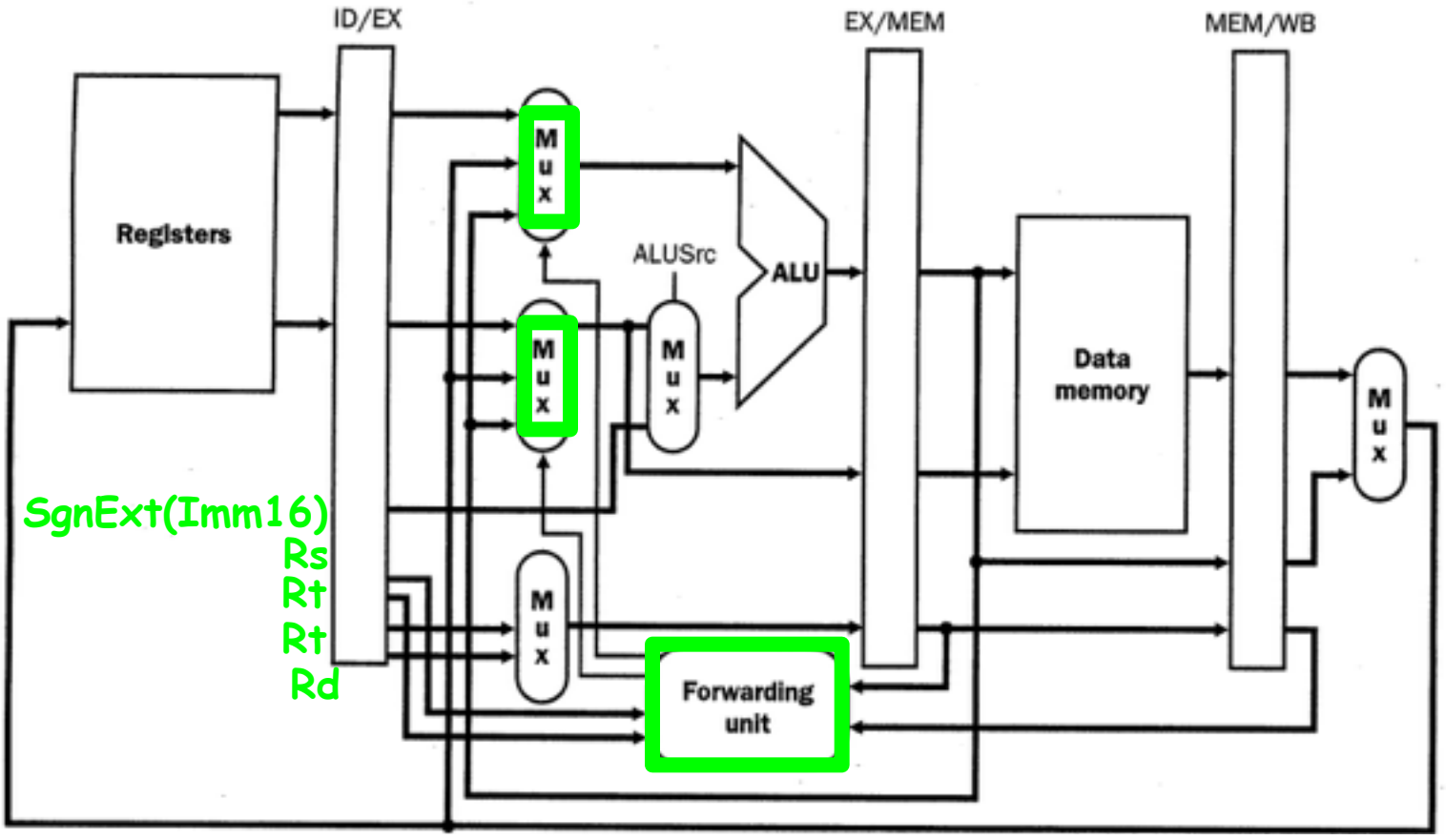
NOTE: Forwarding begins and ends within a single clock cycle

Implementing Forwarding

- The figure below shows a basic pipelined processor (focusing on data flow) but with **no forwarding**

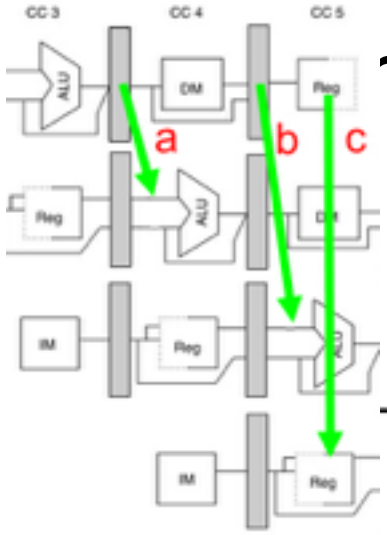


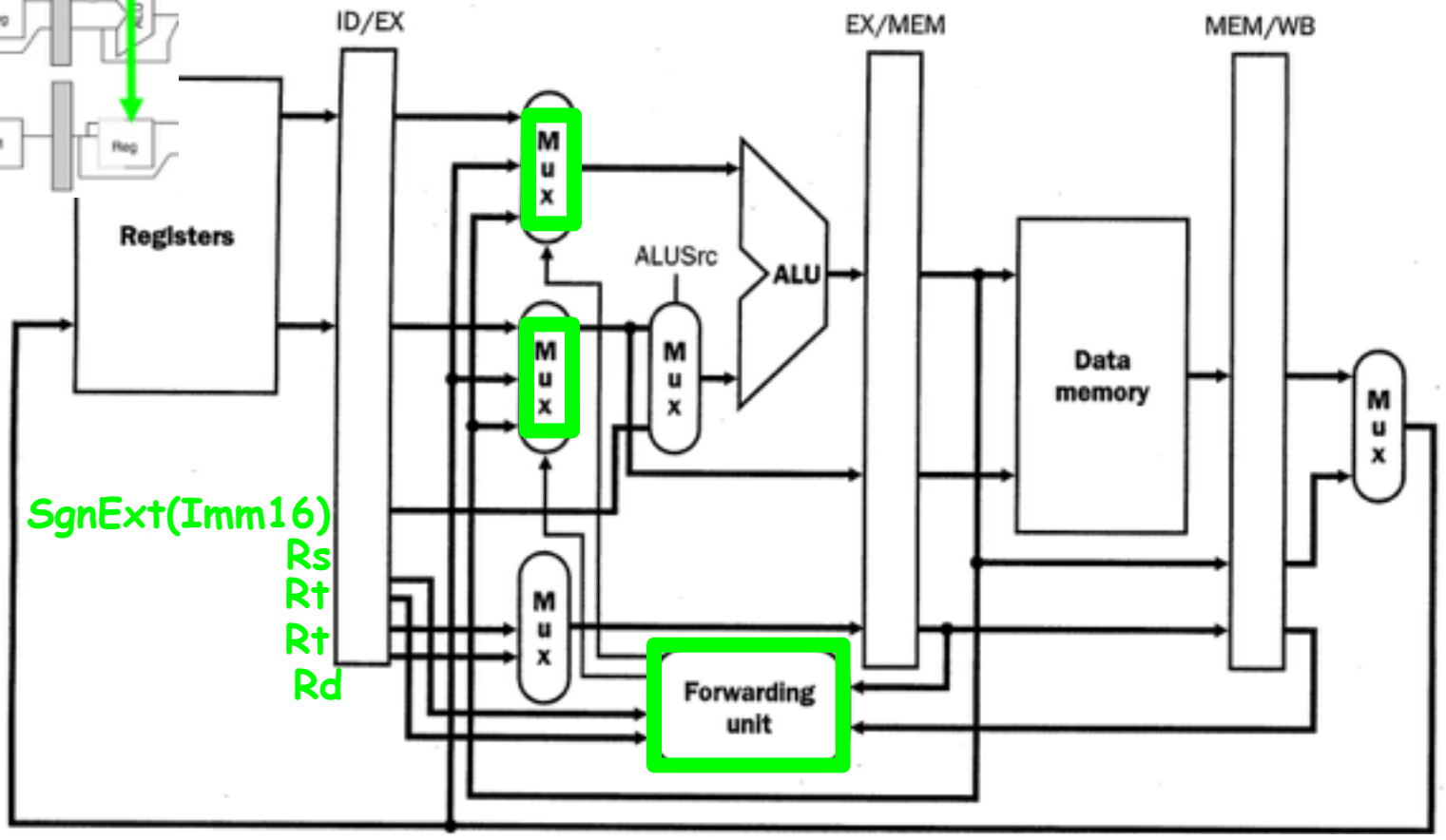
Implementing Forwarding



Forwarding paths often begin at output of pipeline register (values forwarded at beginning of clock cycle)

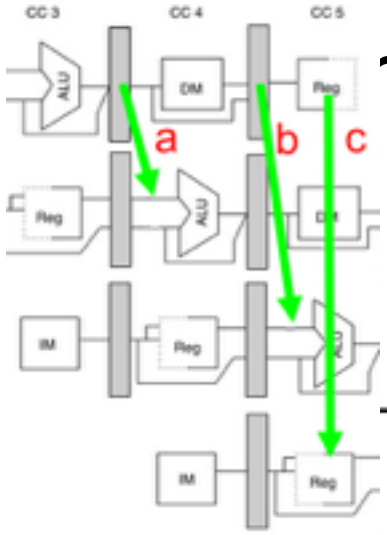
mplementing Forwarding

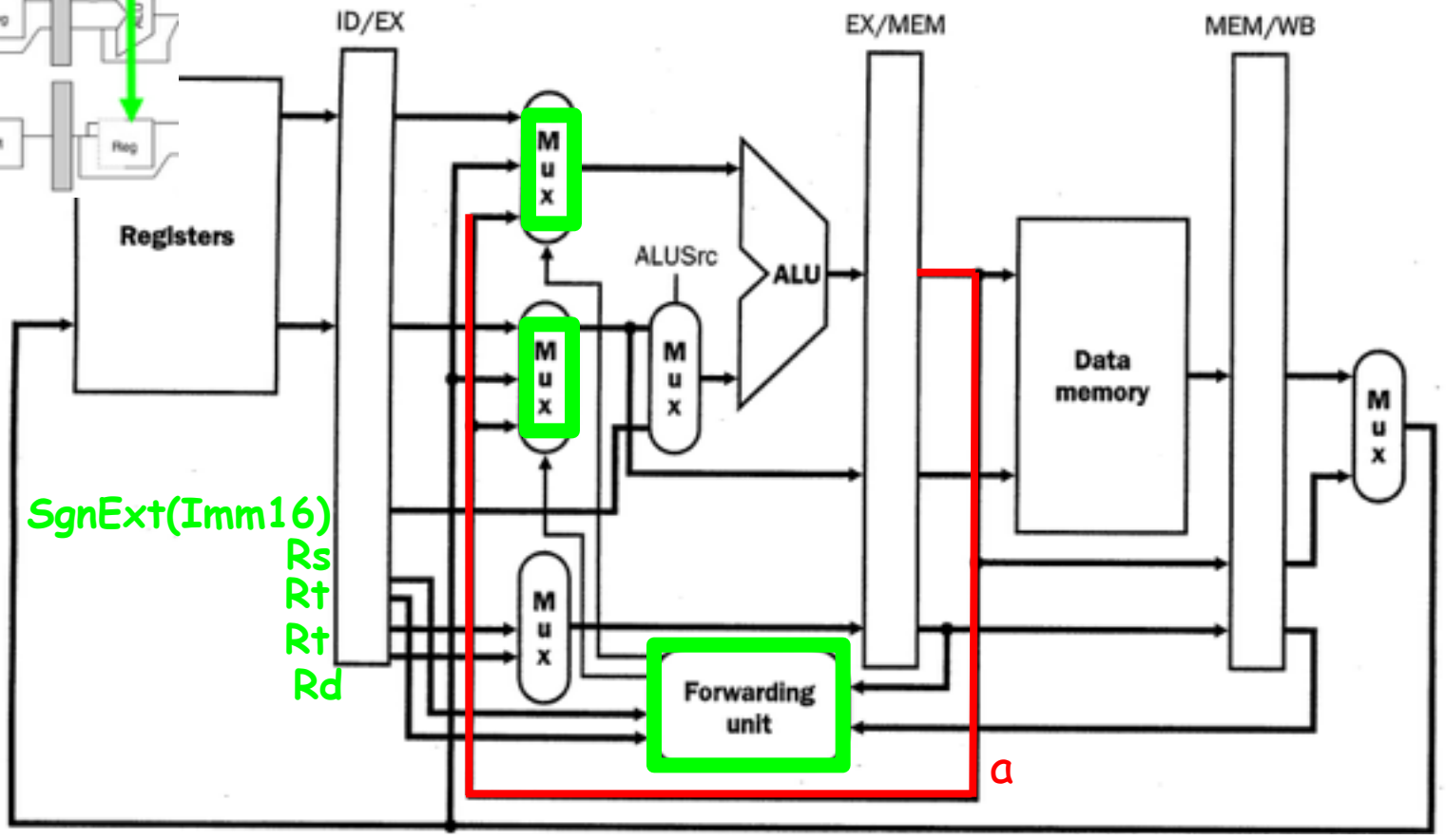




Forwarding paths often begin at output of pipeline register (values forwarded at beginning of clock cycle)

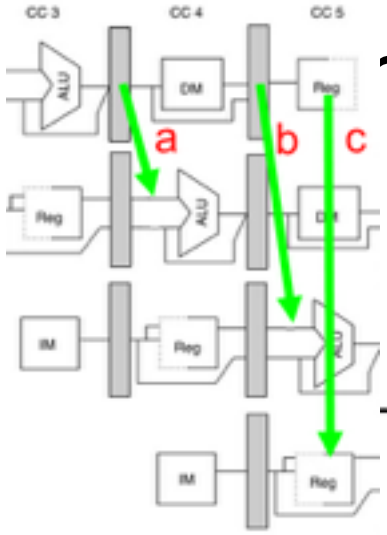
mplementing Forwarding

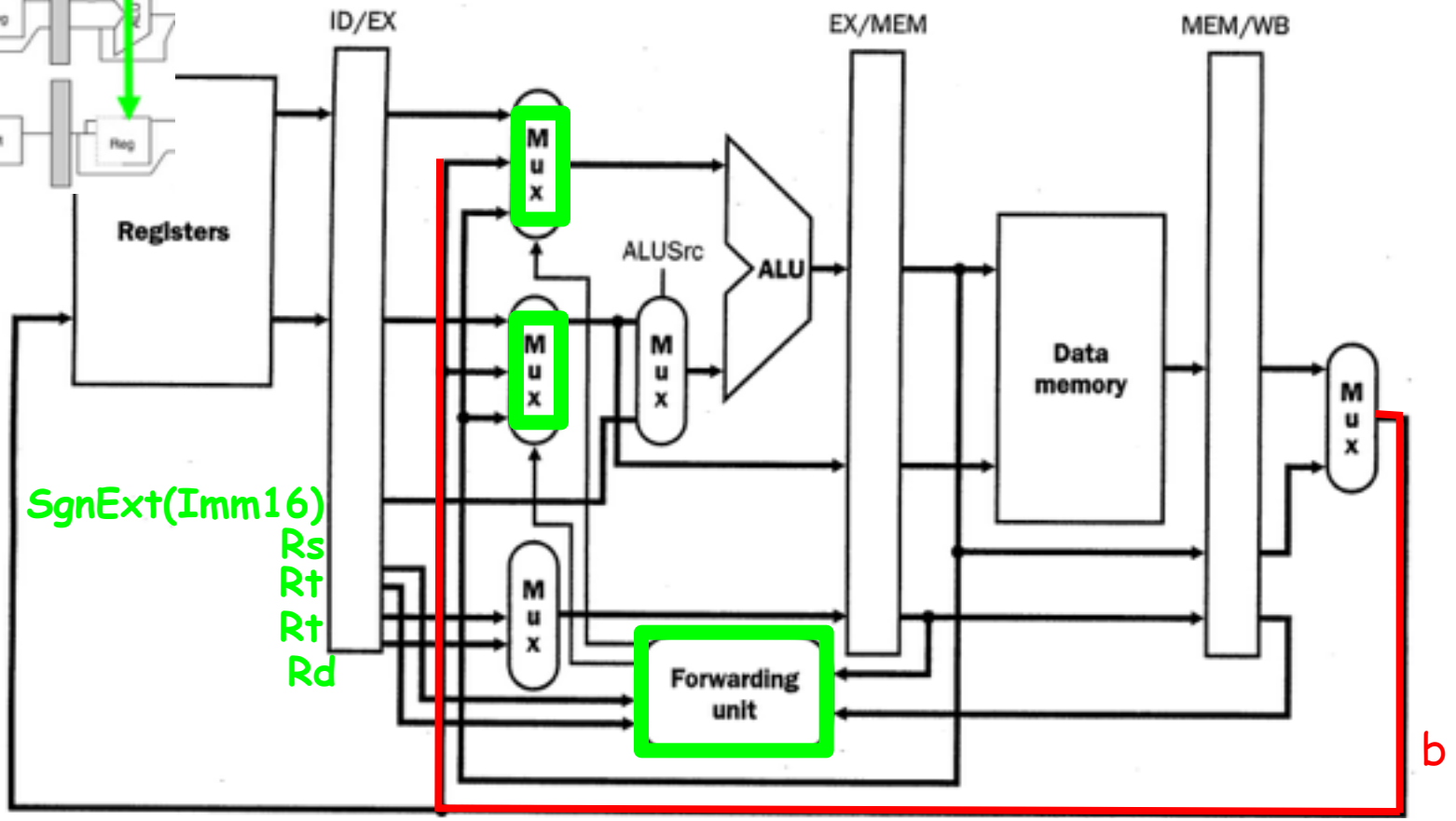




Forwarding paths often begin at output of pipeline register (values forwarded at beginning of clock cycle)

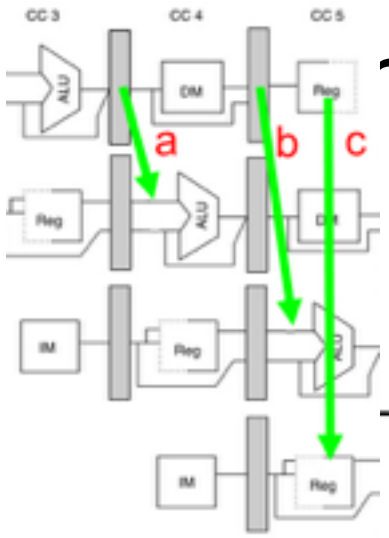
mplementing Forwarding





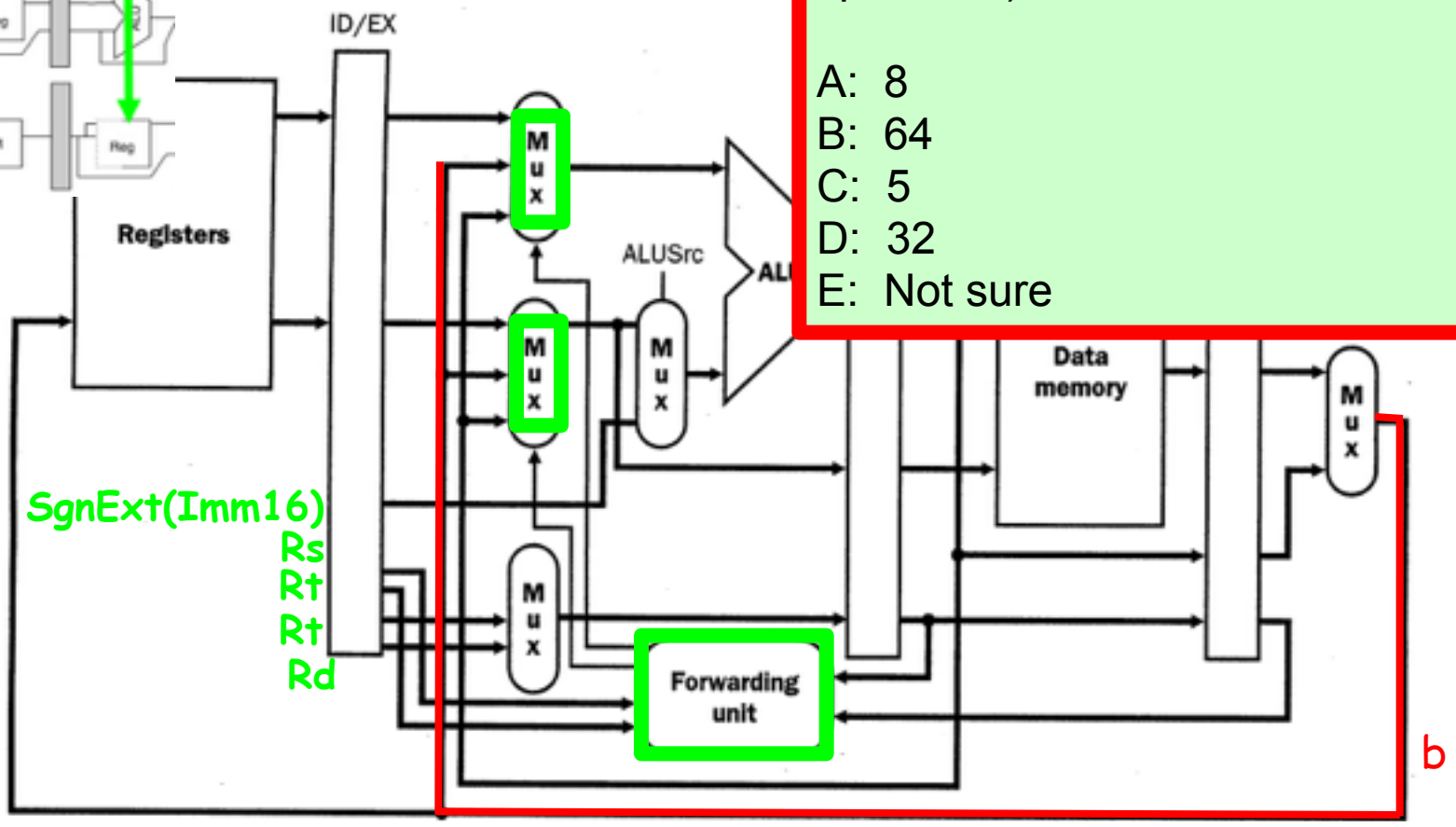
Forwarding paths often begin at output of pipeline register (values forwarded at beginning of clock cycle)

mplementing F



Assume we are building a pipelined MIPS64 processor. Considering only integer registers, how many bits are used to represent each of RS, RT, RD (the source and destination register specifiers)?

A: 8
B: 64
C: 5
D: 32
E: Not sure

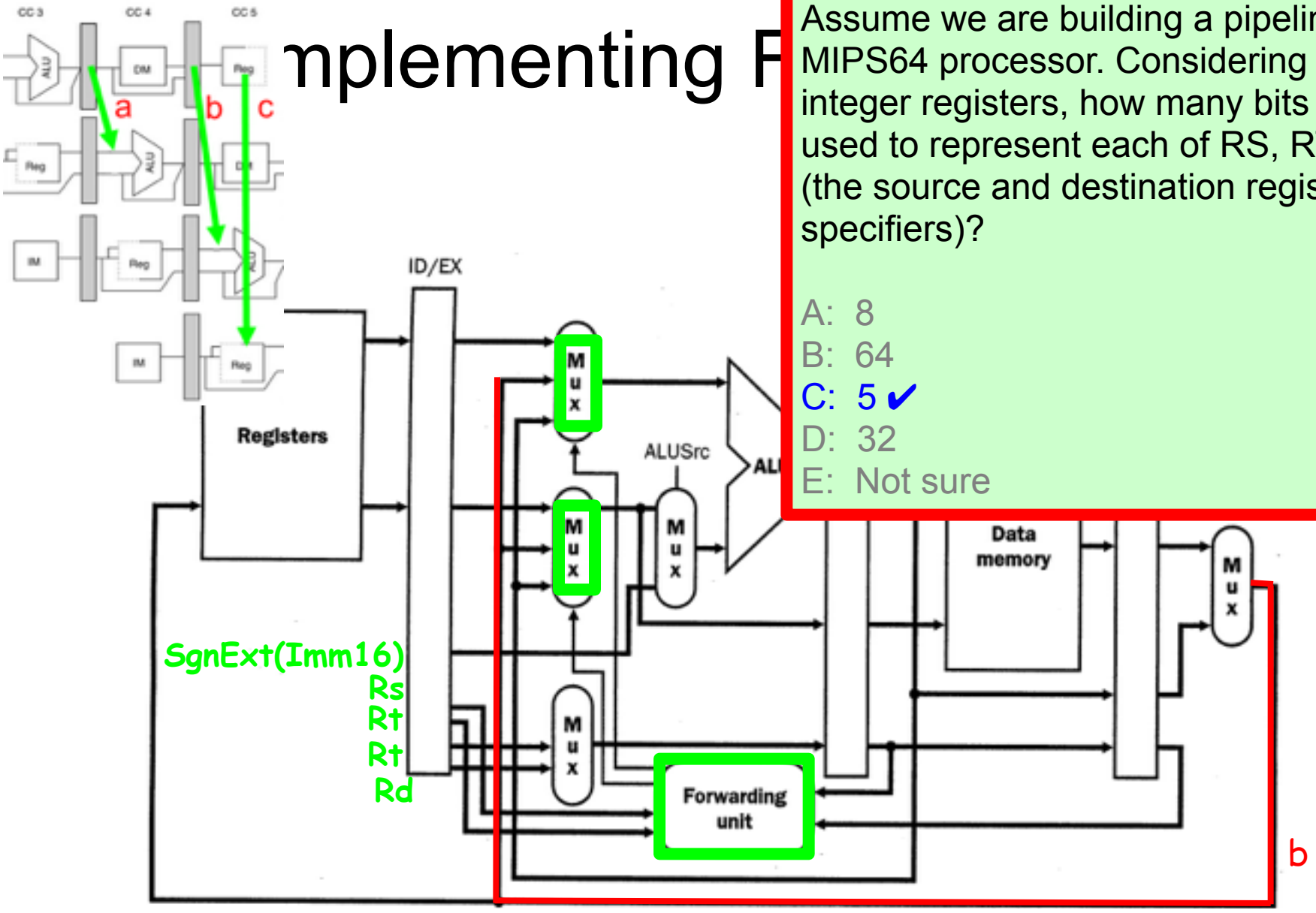


Forwarding paths often begin at output of pipeline register (values forwarded at beginning of clock cycle)

nplementing F

Assume we are building a pipelined MIPS64 processor. Considering only integer registers, how many bits are used to represent each of RS, RT, RD (the source and destination register specifiers)?

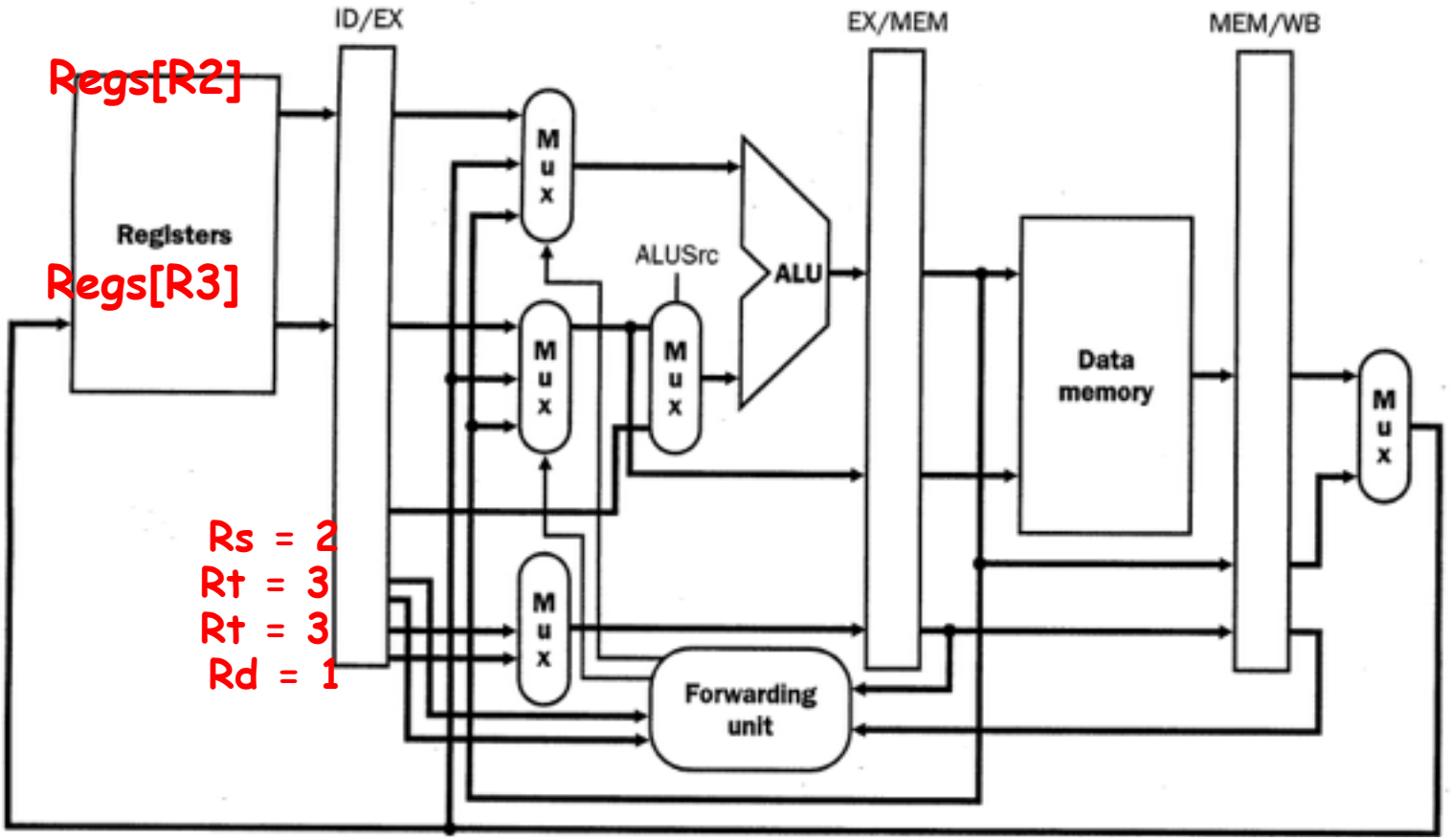
A: 8
B: 64
C: 5 ✔
D: 32
E: Not sure



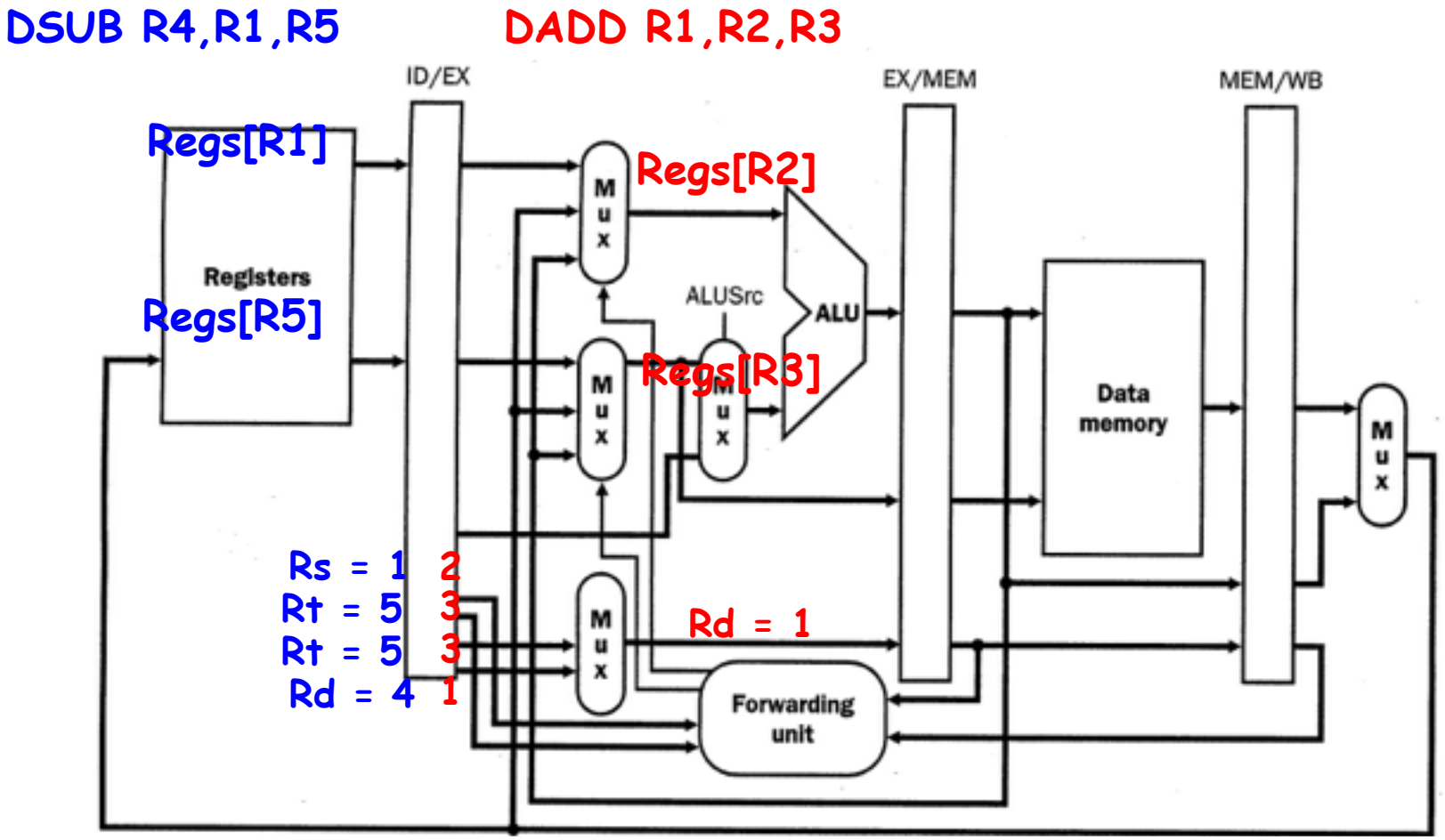
Forwarding paths often begin at output of pipeline register (values forwarded at beginning of clock cycle)

Forwarding Example

DADD R1,R2,R3

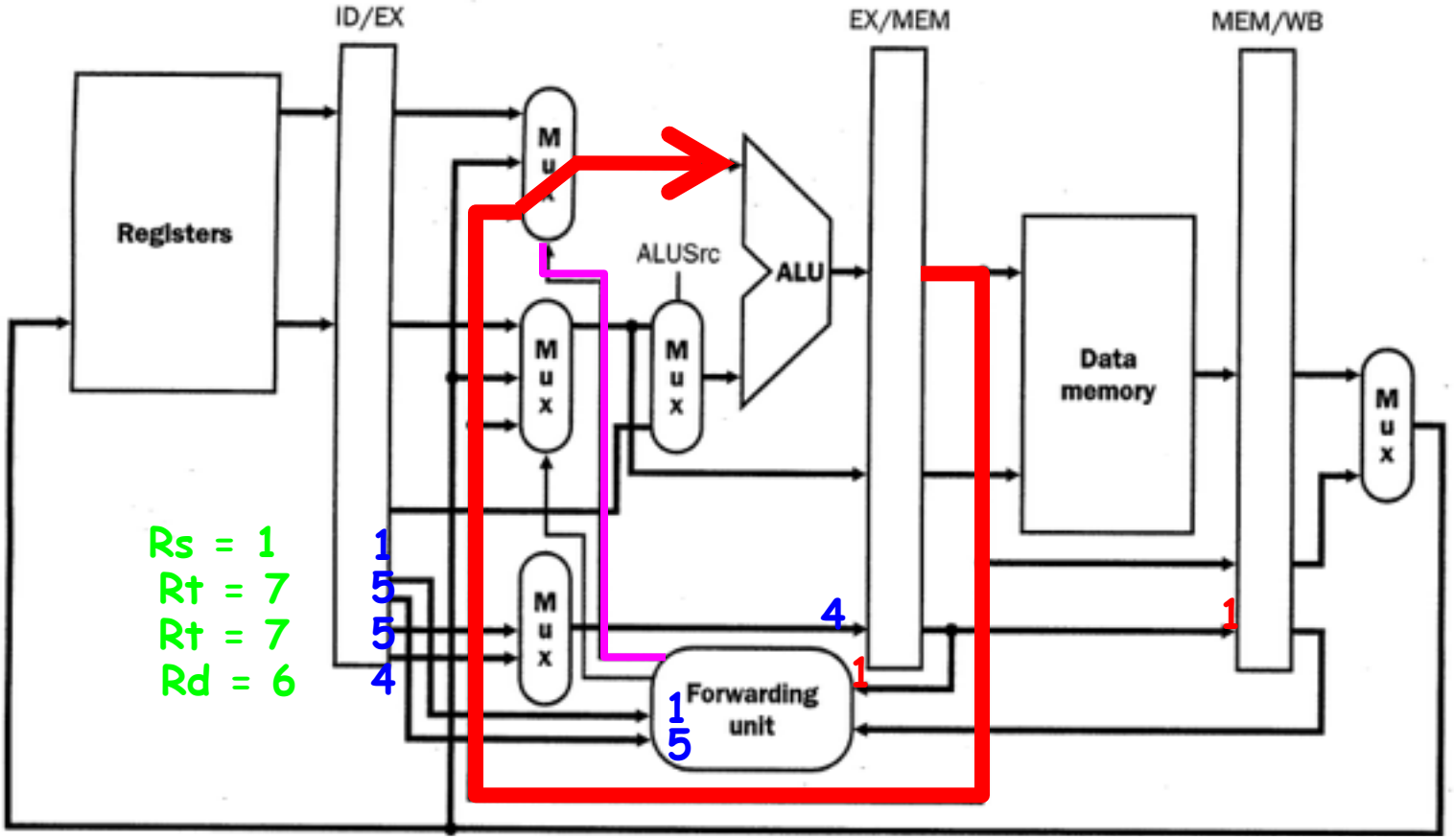


Forwarding Example



Forwarding Example

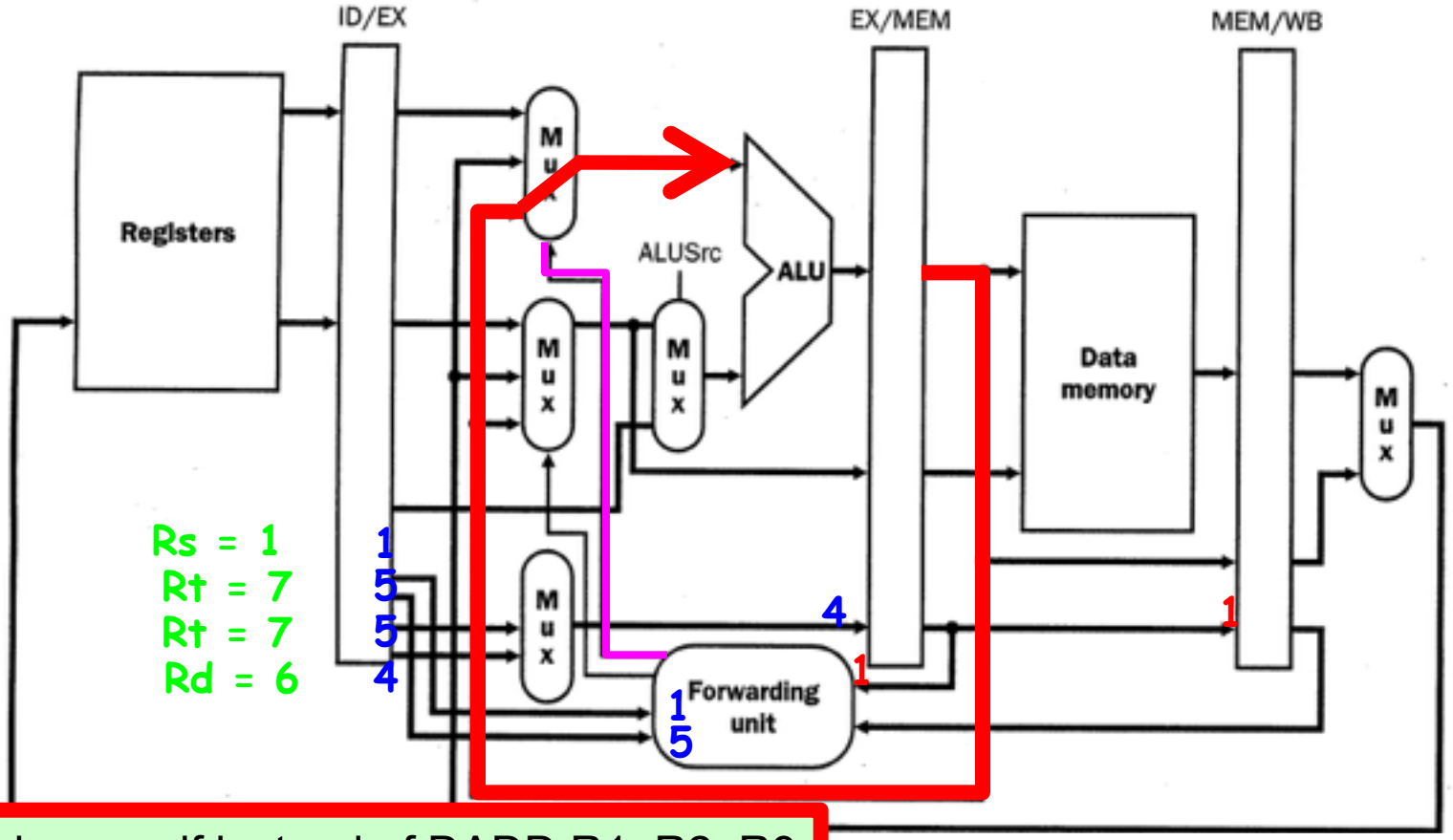
AND R6,R7,R1 DSUB R4,R1,R5 DADD R1,R2,R3



Above: Forwarding unit compares the destination register specifier of DADD instruction in EX/MEM to both source register specifiers of DSUB instruction in ID/EX. Finds a match (Rs == Rd == 1), and thus enables forwarding mux.

Forwarding Example

AND R6,R7,R1 DSUB R4,R1,R5 DADD R1,R2,R3



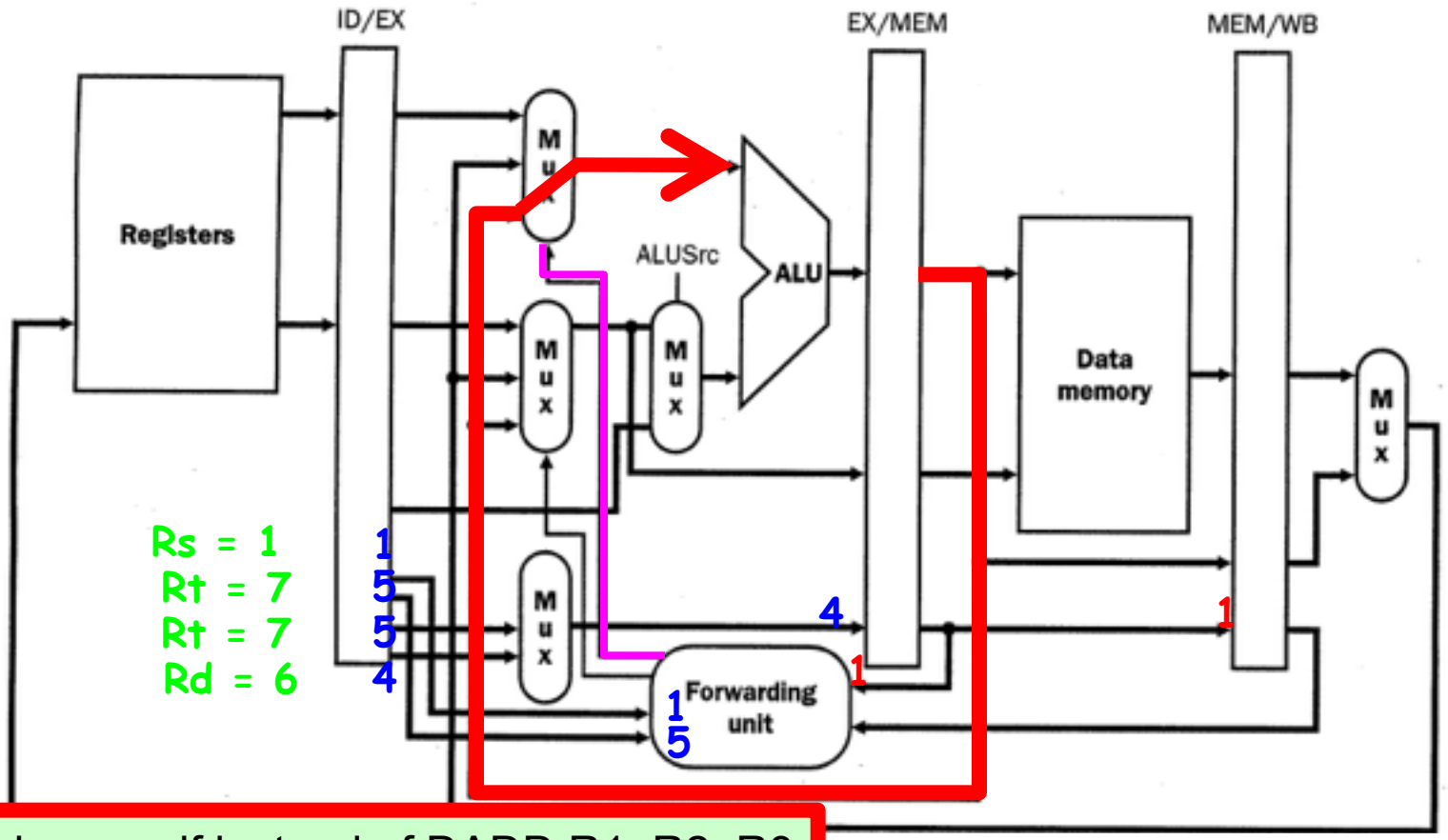
What would happen if instead of DADD R1, R2, R3 the first instruction was LD R1,0(R3)?

A: Use same forwarding path as above
B: Don't use same forwarding path as above
C: Not sure

…n register specifier of DADD
…pecifiers of DSUB instruction
…hus enables forwarding mux.

Forwarding Example

AND R6,R7,R1 DSUB R4,R1,R5 DADD R1,R2,R3



What would happen if instead of DADD R1, R2, R3 the first instruction was LD R1,0(R3)?

A: Use same forwarding path as above

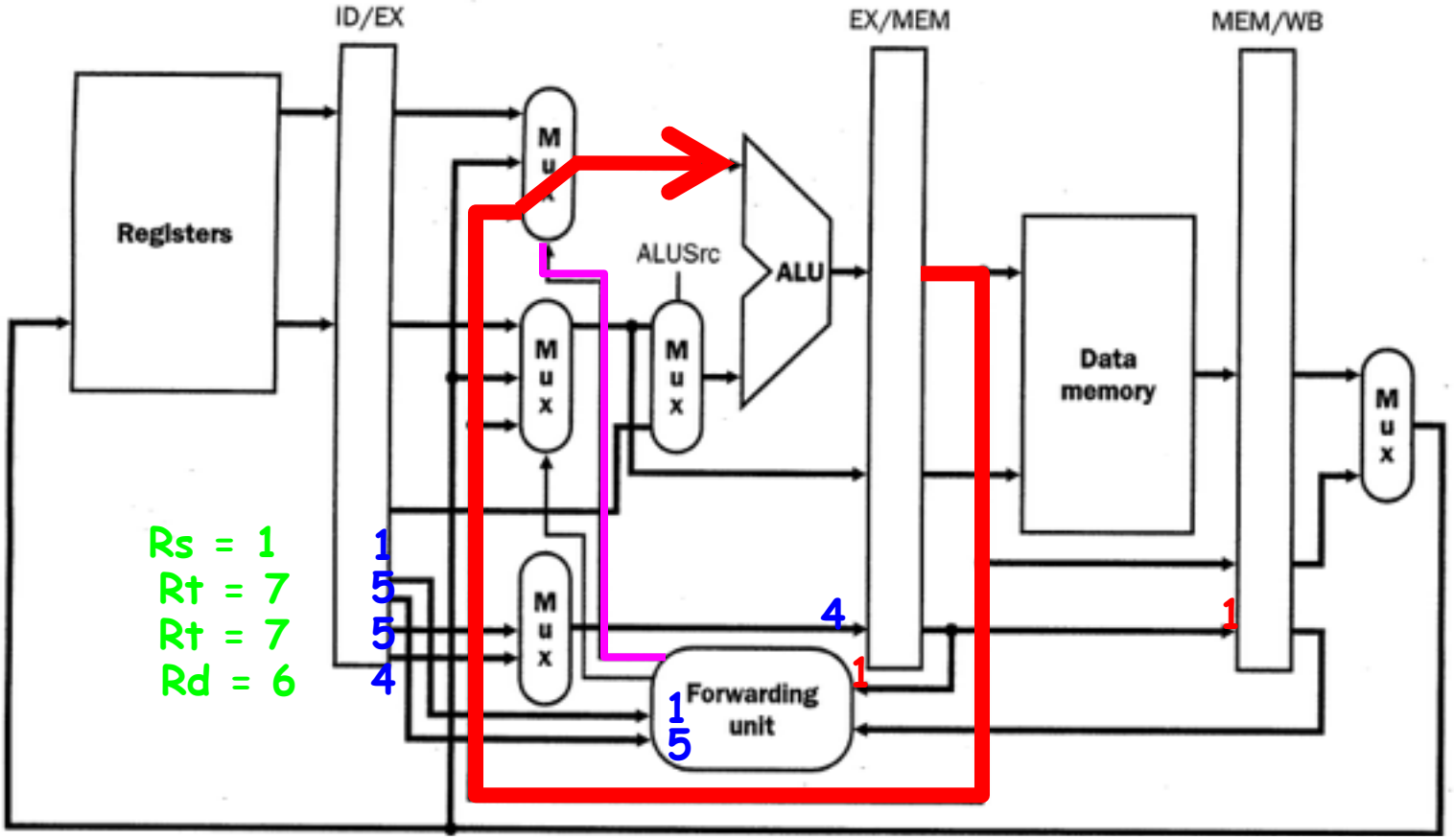
B: Don't use same forwarding path as above ✓

C: Not sure

…n register specifier of DADD …pecifiers of DSUB instruction …hus enables forwarding mux.

Forwarding Example

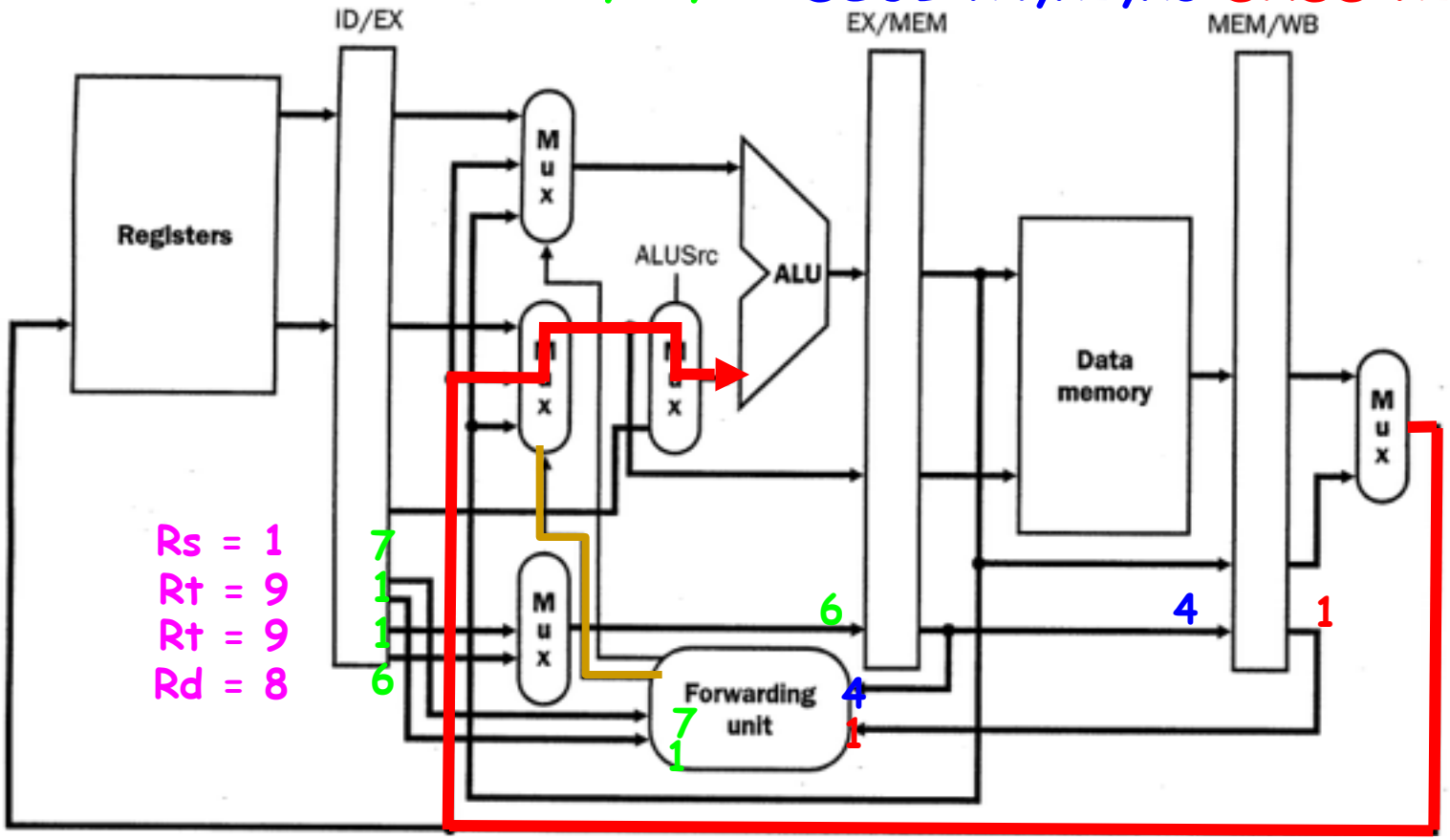
AND R6,R7,R1 DSUB R4,R1,R5 DADD R1,R2,R3



Above: Forwarding unit compares the destination register specifier of DADD instruction in EX/MEM to both source register specifiers of DSUB instruction in ID/EX. Finds a match (Rs == Rd == 1), and thus enables forwarding mux.

Forwarding Example

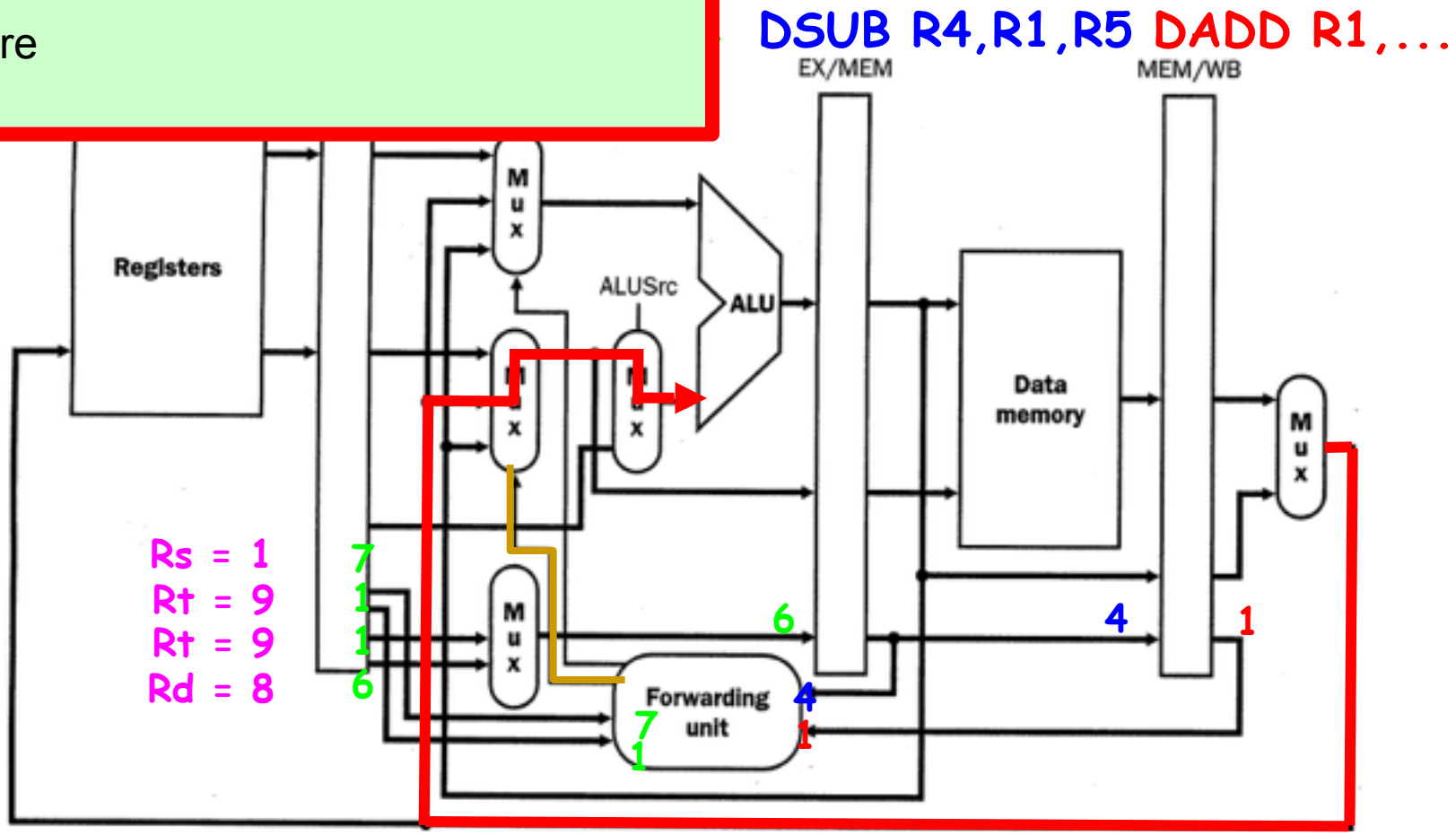
OR R8,R1,R9 AND R6,R7,R1 DSUB R4,R1,R5 DADD R1,...



Is it possible to forward one ALU source operand from EX/MEM and the other ALU source from MEM/WB at the same time?

A: Yes
B: Not sure
C: No

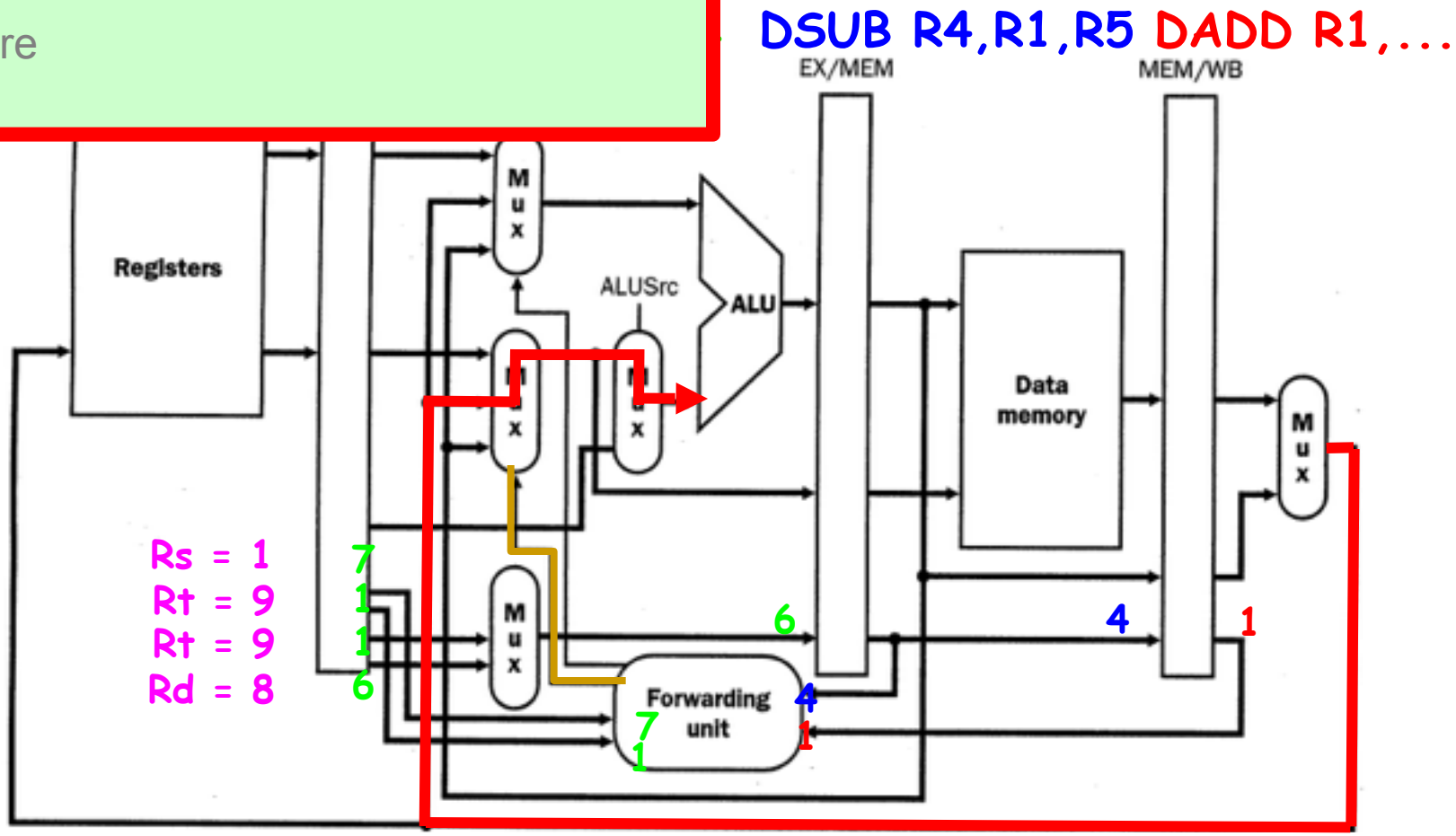
Example



Is it possible to forward one ALU source operand from EX/MEM and the other ALU source from MEM/WB at the same time?

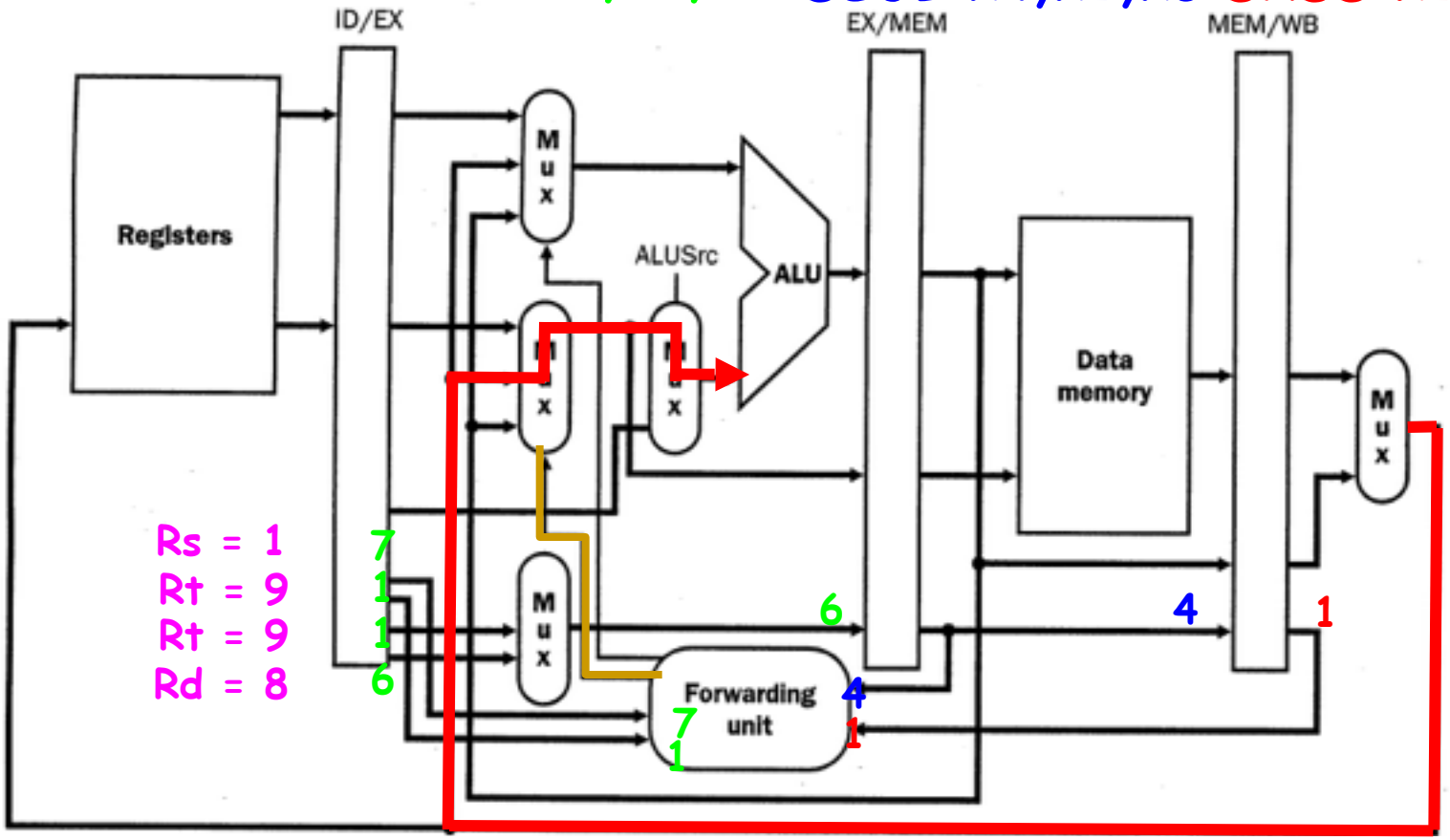
A: Yes✔
B: Not sure
C: No

Example



Forwarding Example

OR R8,R1,R9 AND R6,R7,R1 DSUB R4,R1,R5 DADD R1,...



Forwarding Unit Logic

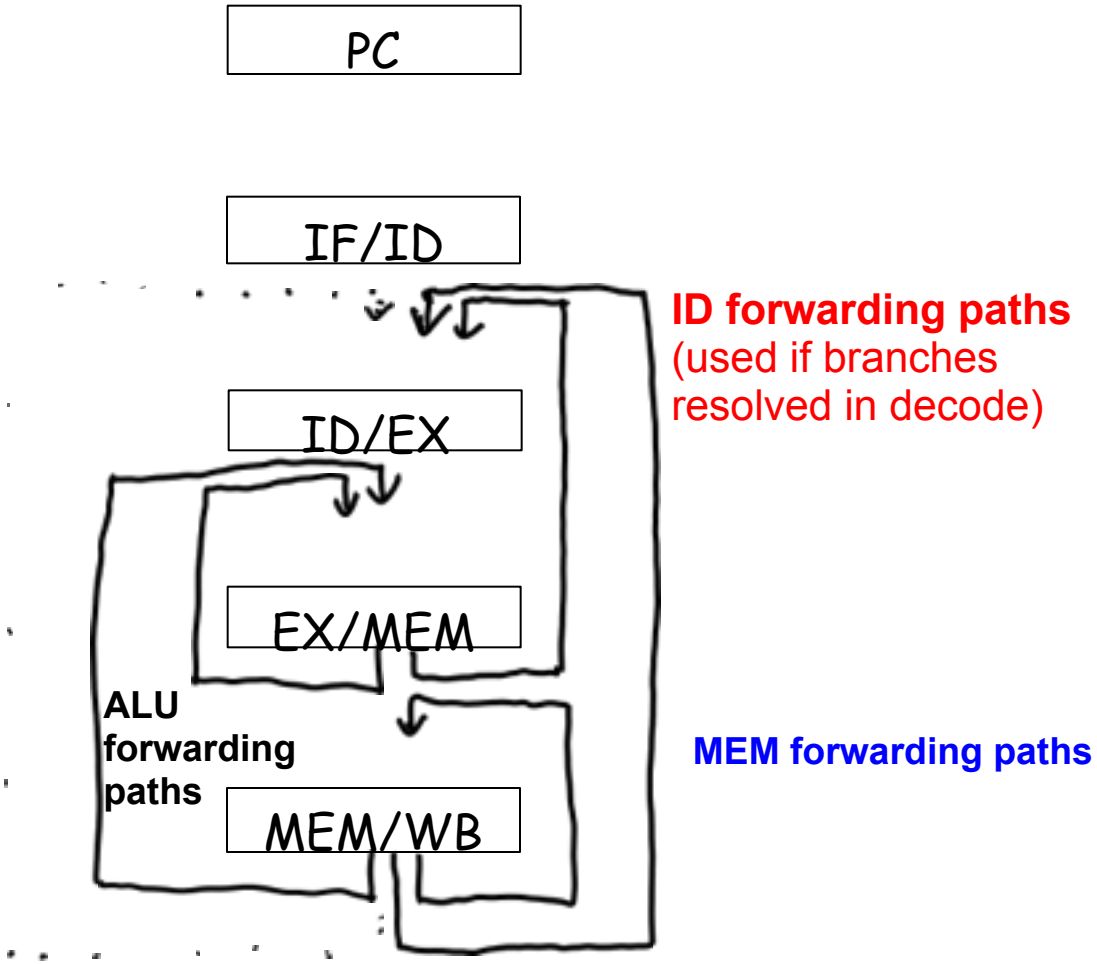
Let's briefly consider the forwarding unit in a bit more detail.

This unit needs to check for data hazards between instructions in different stages of the pipeline and enable the forwarding muxes.

Fig A-22 in the textbook (partially shown below) lists all the combinations a hardware designer would need to consider for controlling multiplexers for "a" and "b".

Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR[rd] == ID/EX.IR[rs]
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rd] == ID/EX.IR[rt]

Where do Forwarding Paths go?

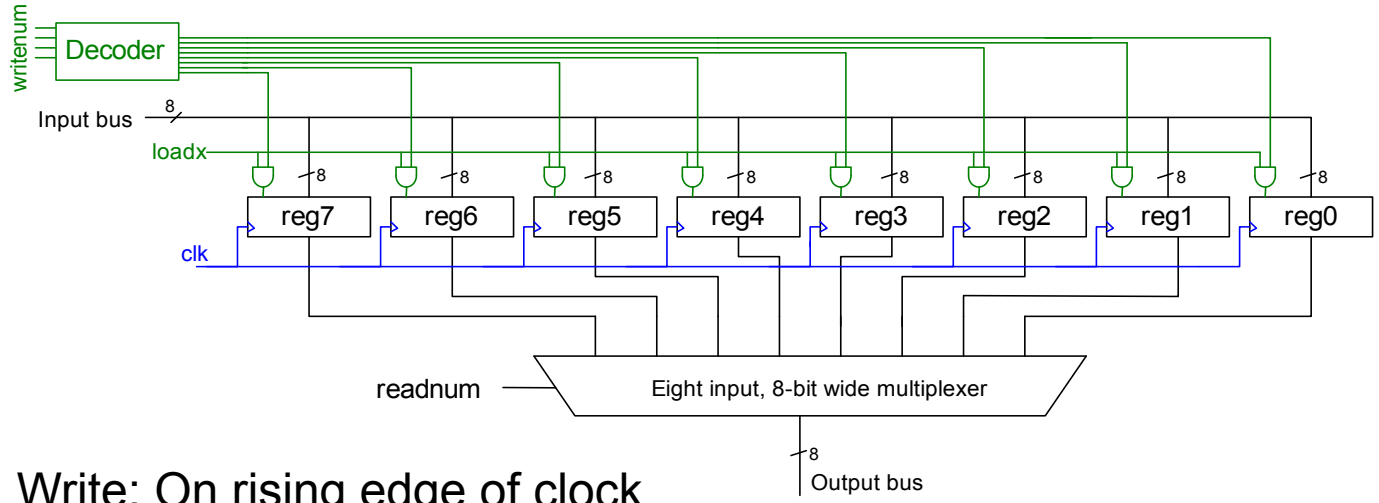


To fully reduce or eliminate need for stalling: Add forwarding paths starting from "producer" stages (and later stages) to earlier "consumer" stages.

Example: "All forwarding paths required to reduce or eliminate stalls" if branches are resolved in decode

Forwarding through register file?

- Recall EECE 353 Lab 3:



  - Write: On rising edge of clock
  - Read: “Output bus” changes whenever “readnum” changes or when contents of register changes.
  - To forward through register file, want to write in first half of clock cycle.
  - We can achieve this effect by inverting clock input to registers so that flip-flops capture write data on falling clock edge.

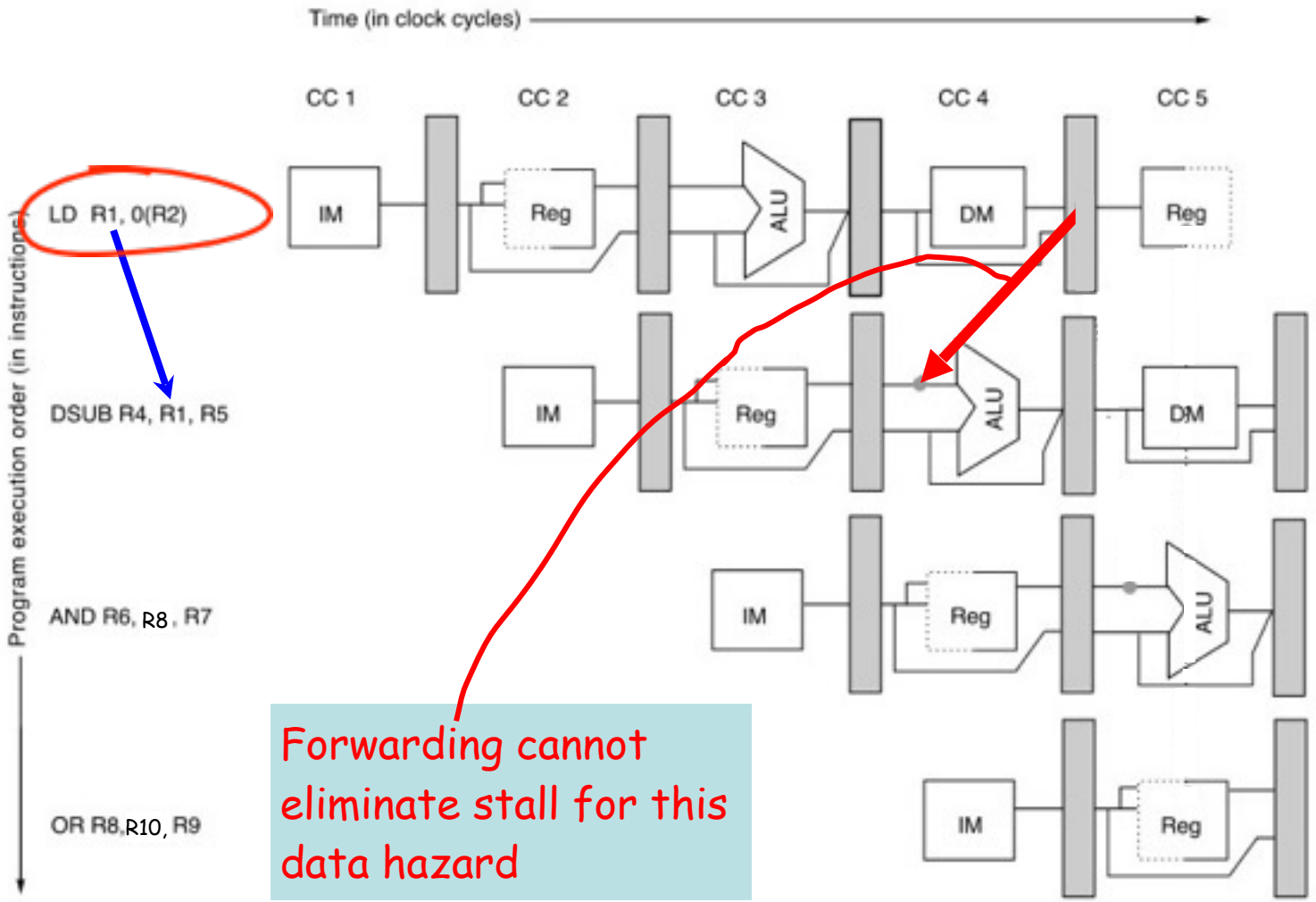
Pipeline Timing Diagram for Forwarding

	Clock Number								
	1	2	3	4	5	6	7	8	9
DADD R1,R2,R3	IF	ID	EX	MEM	WB				
DSUB 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	

Use arrows to show where forwarding occurs. Also mark instruction operands involved in forwarding on the left.

Note that forwarding arrows go either within same column, or from one column to next column. This is because forwarding of data through muxes occurs within a single clock cycle.

Unavoidable Stalls

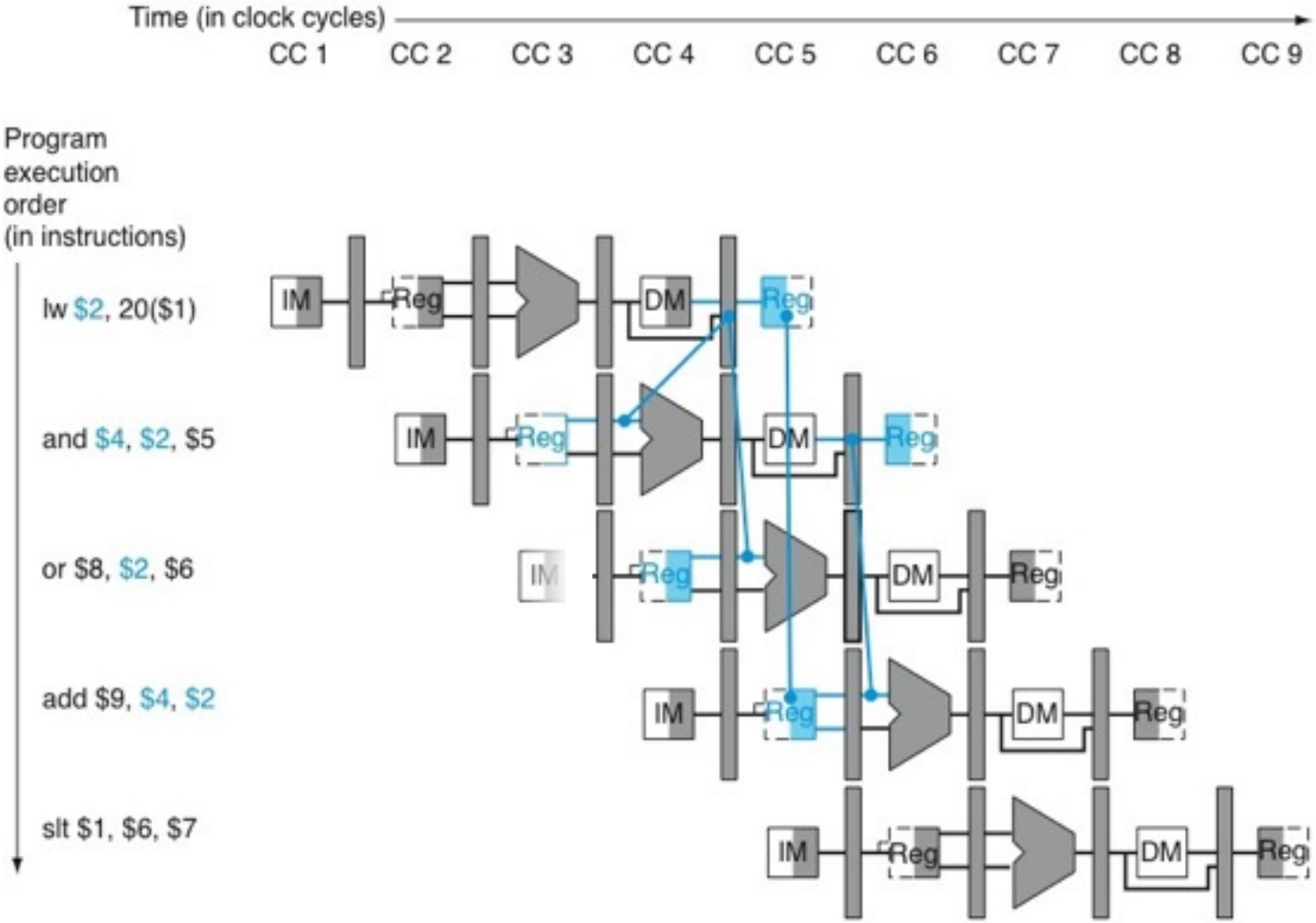


Pipeline Stalls

- Stalled instruction and subsequent instructions held in pipeline registers.
- We insert “no op” instruction(s) in place of stalled instruction.
- Unavoidable stalls occur when the stage **producing** the forwarded value is “later” in pipeline than stage **consuming** the value and there are “not enough” instructions separating these two instructions.

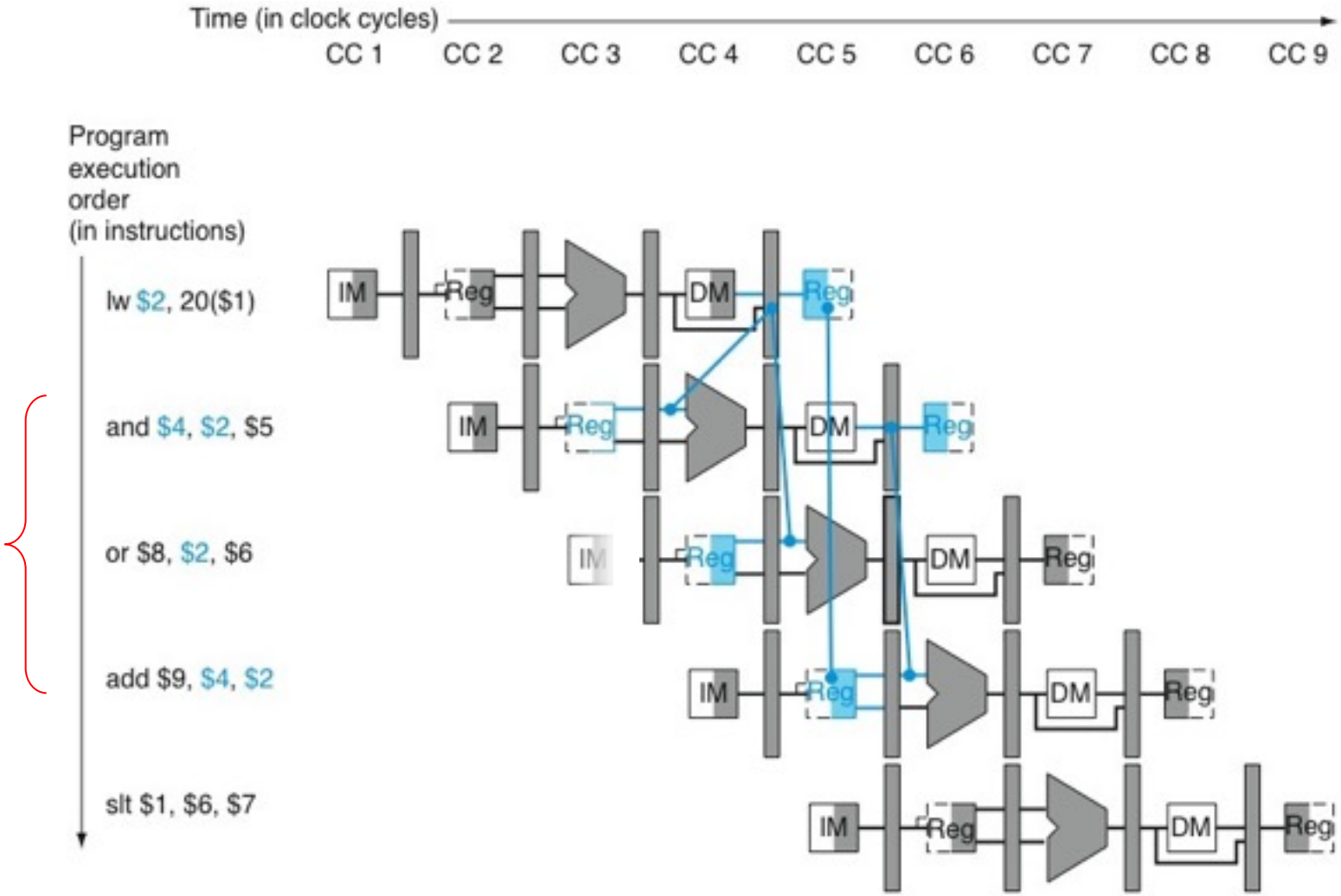
Unavoidable Stall Example:

Solution Load Interlock



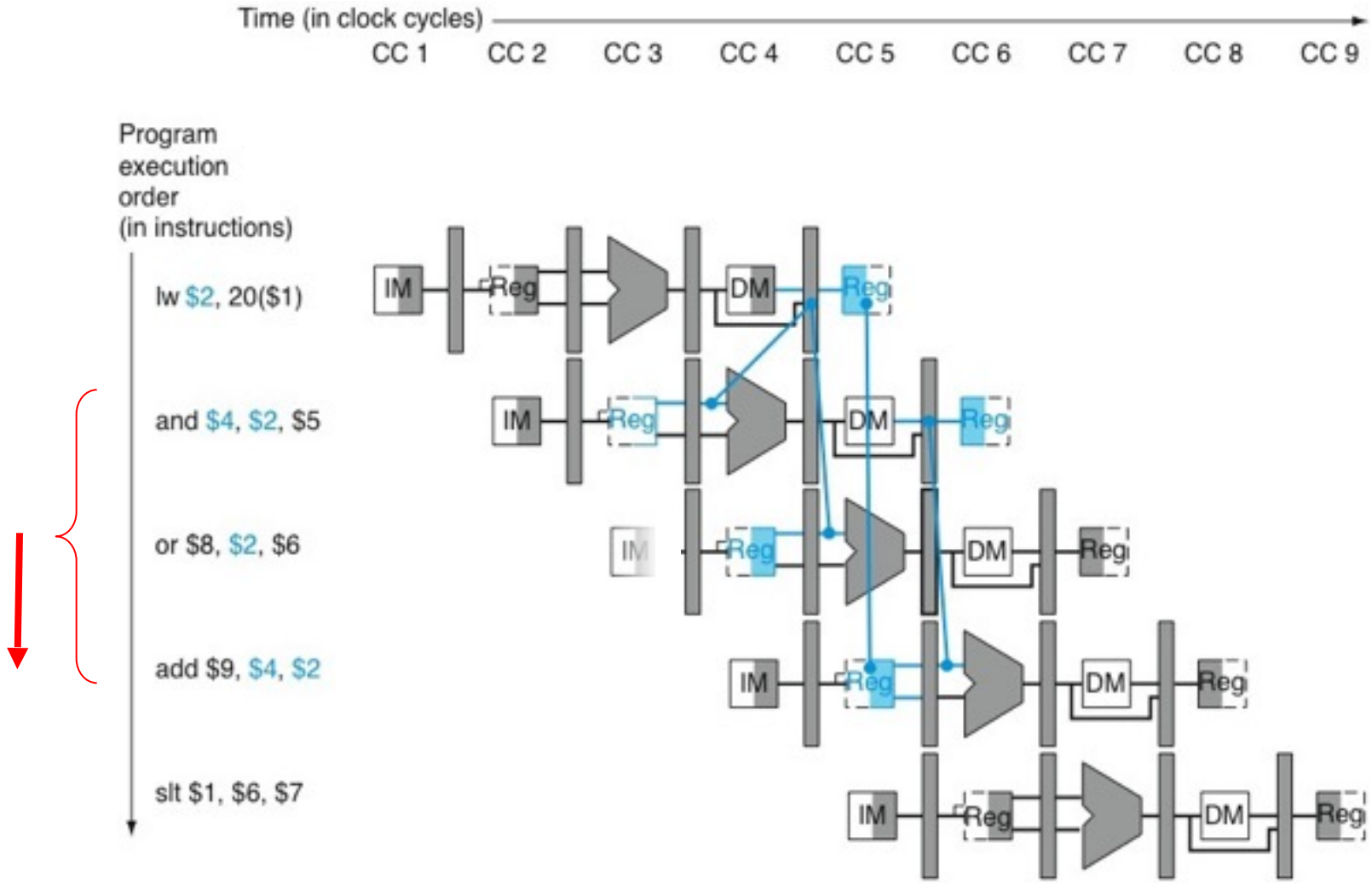
Unavoidable Stall Example:

Solution Load Interlock



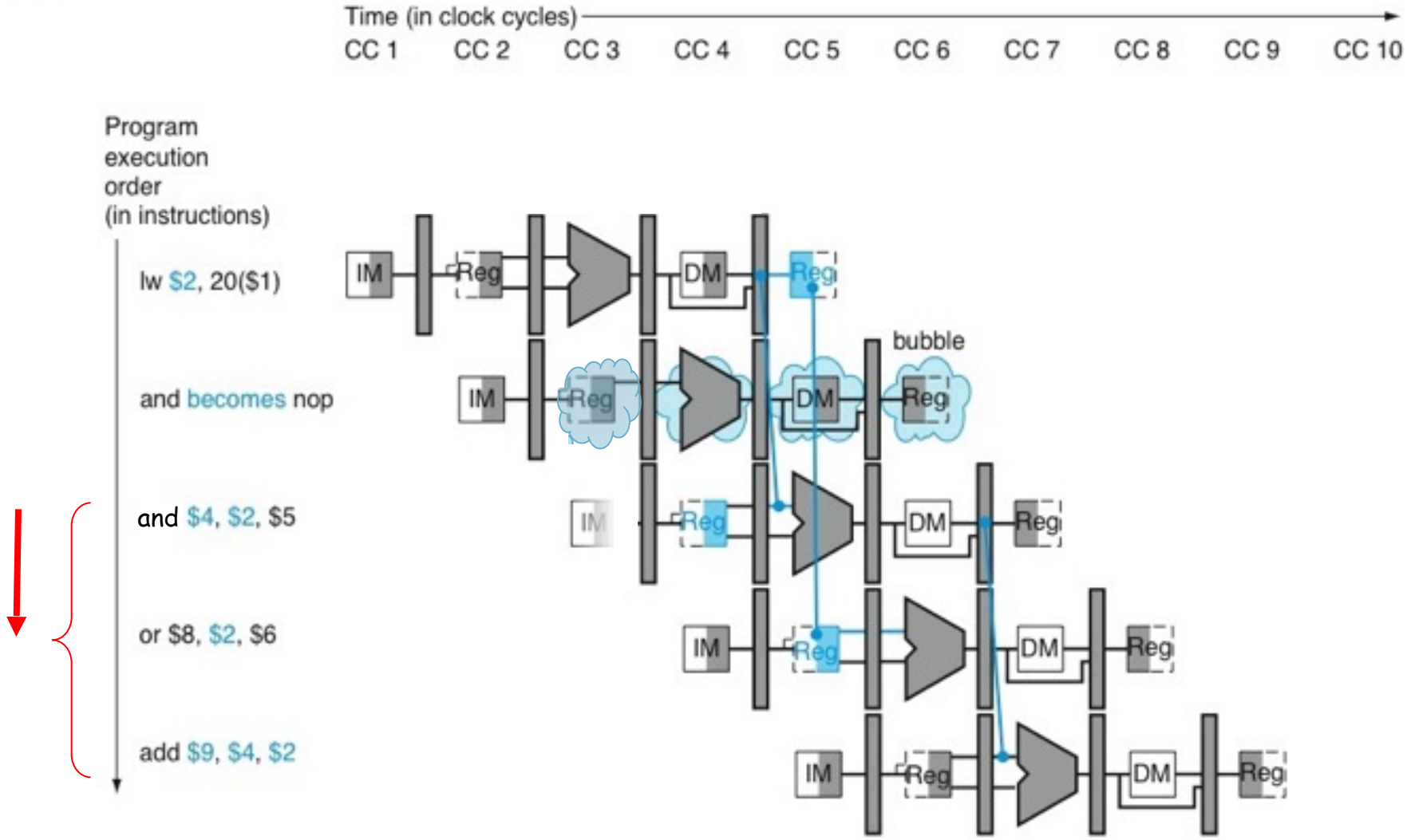
Unavoidable Stall Example:

Solution Load Interlock



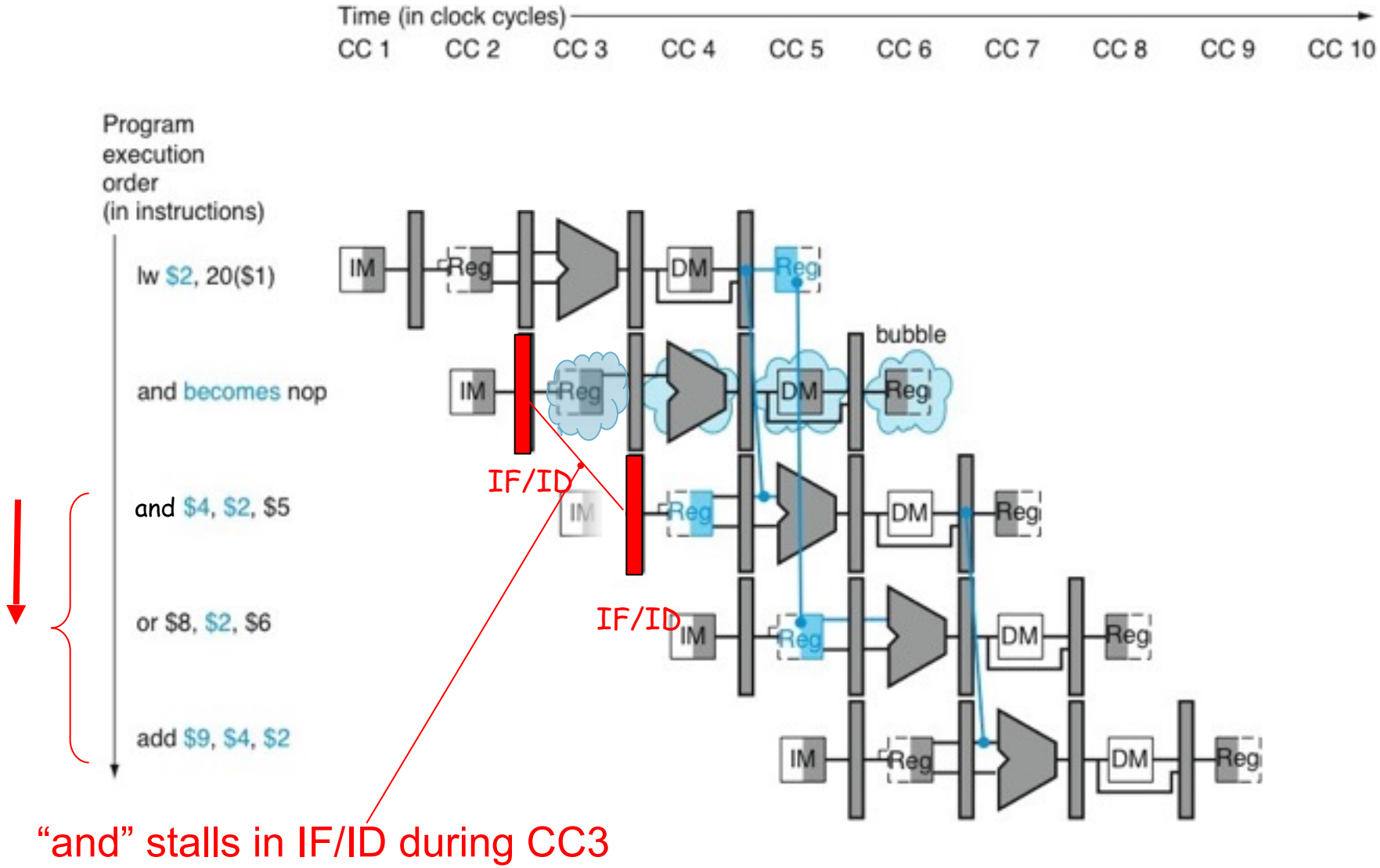
Unavoidable Stall Example:

Solution Load Interlock



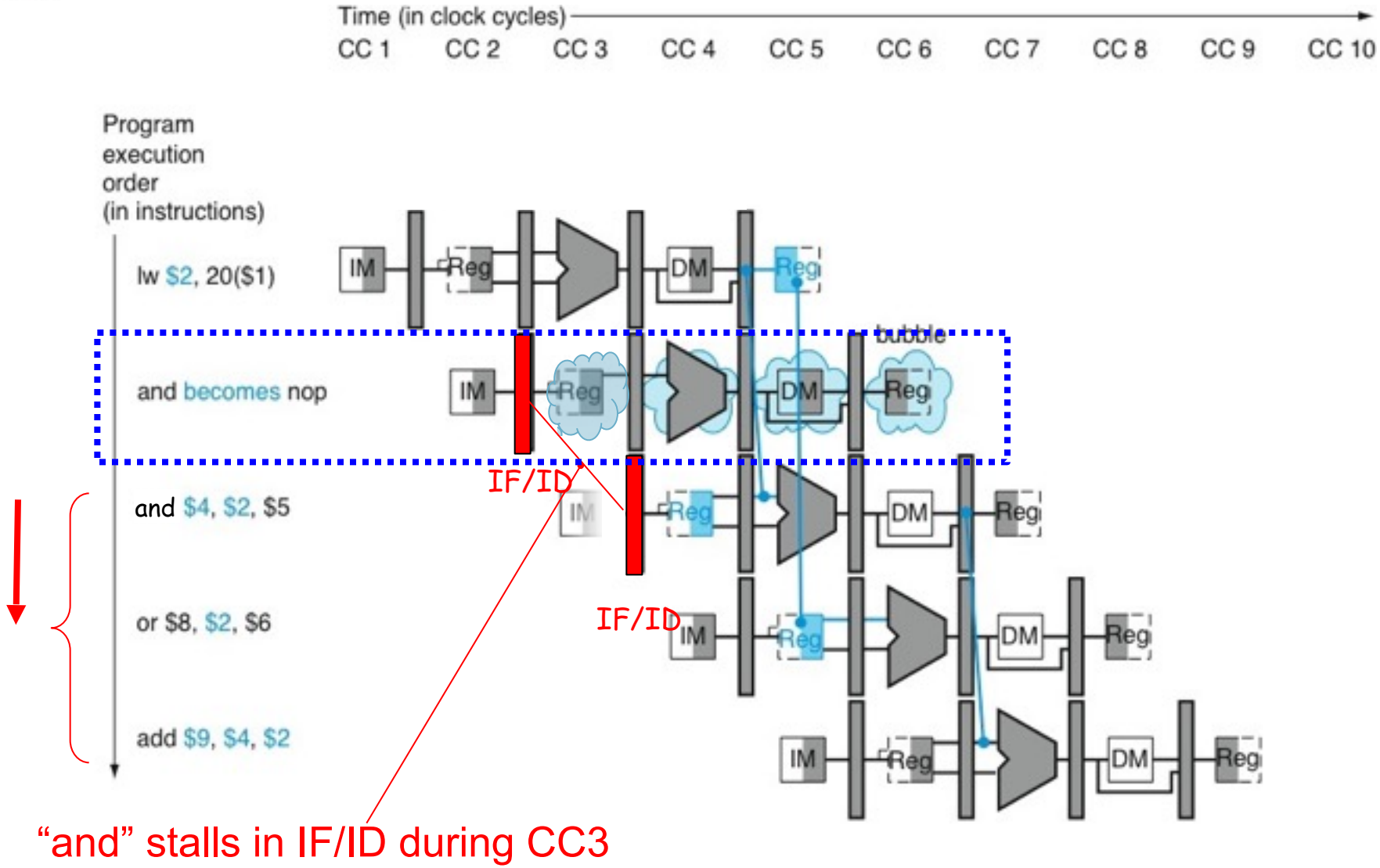
Unavoidable Stall Example:

Solution Load Interlock



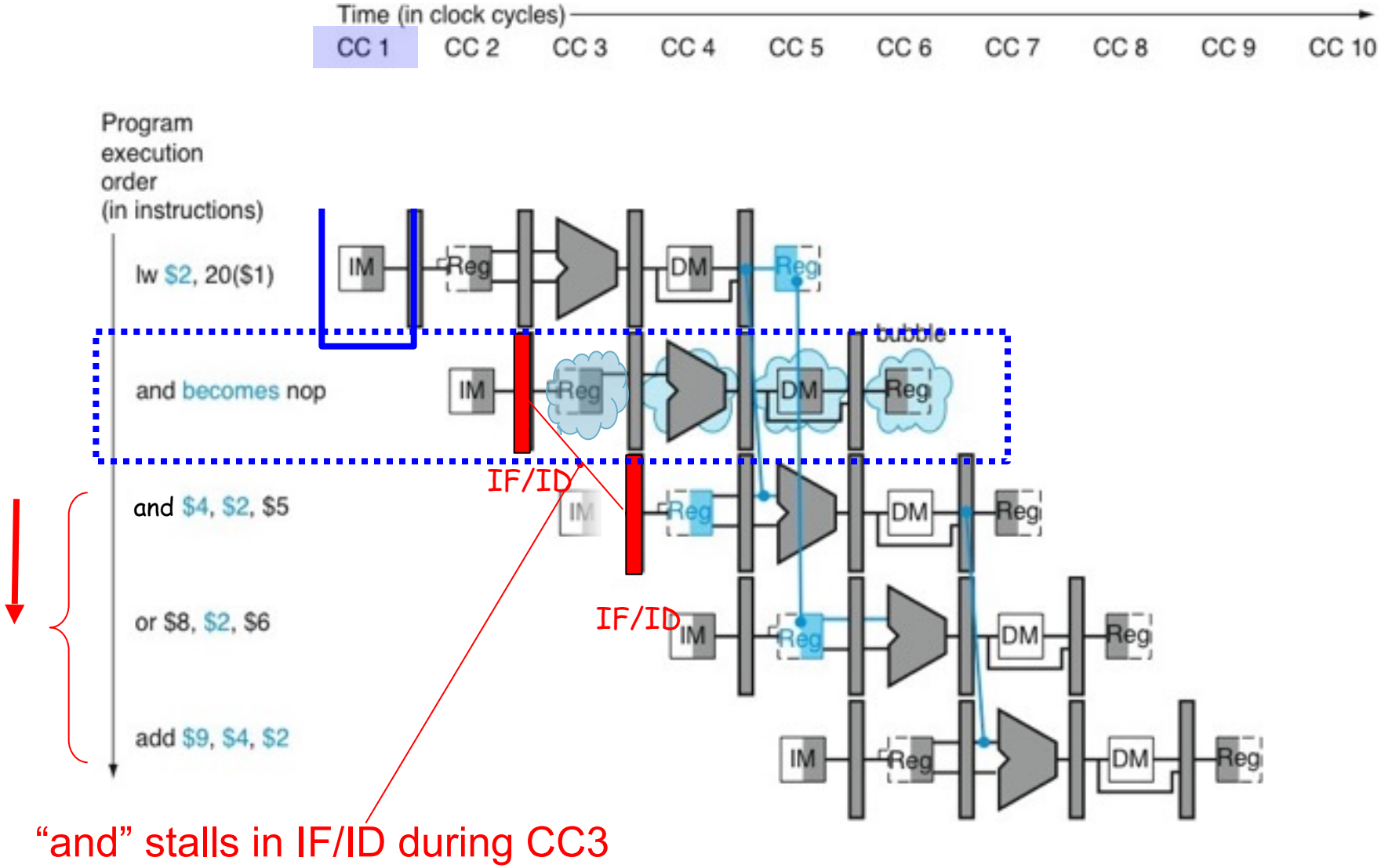
Unavoidable Stall Example:

Solution Load Interlock



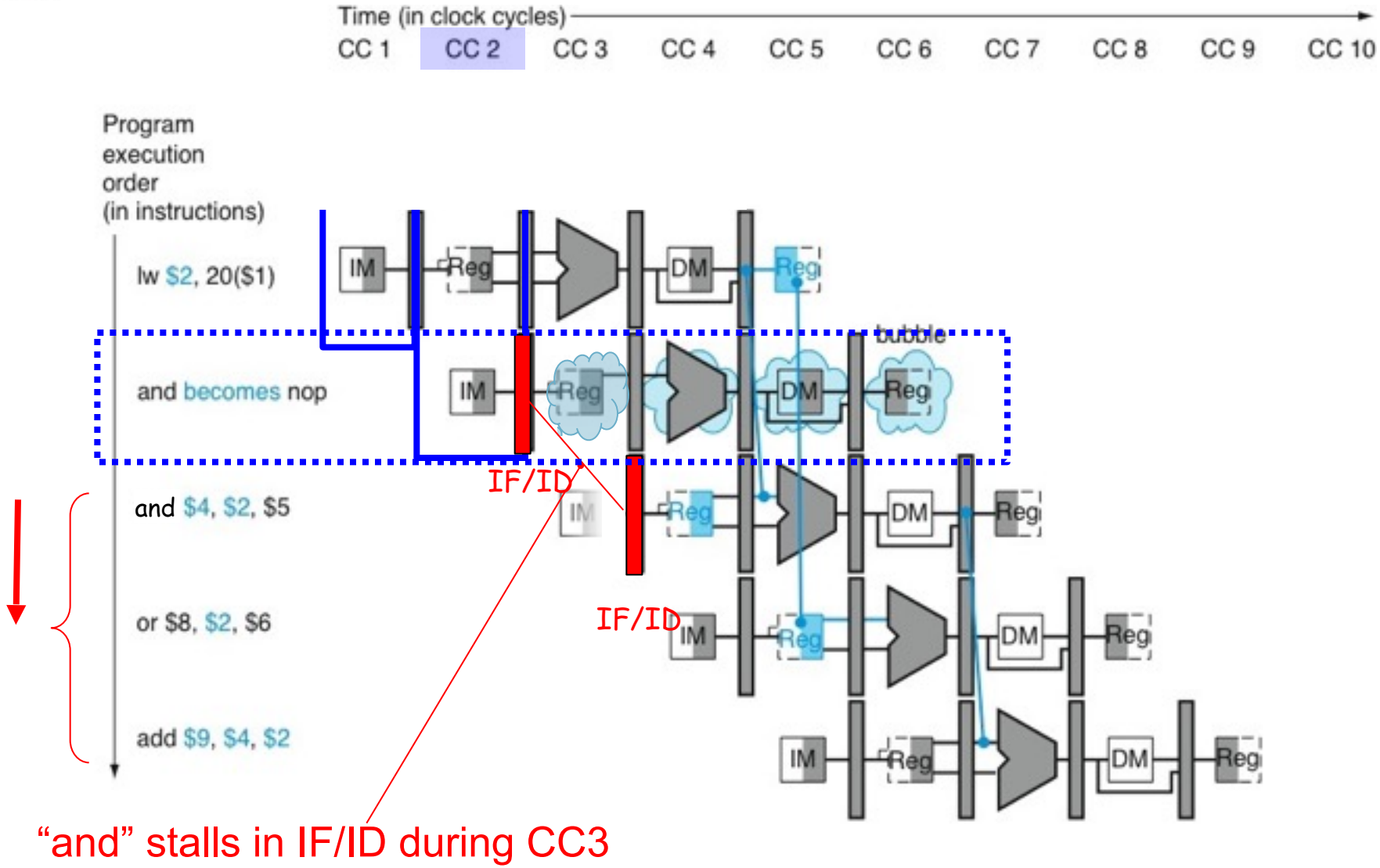
Unavoidable Stall Example:

Solution Load Interlock



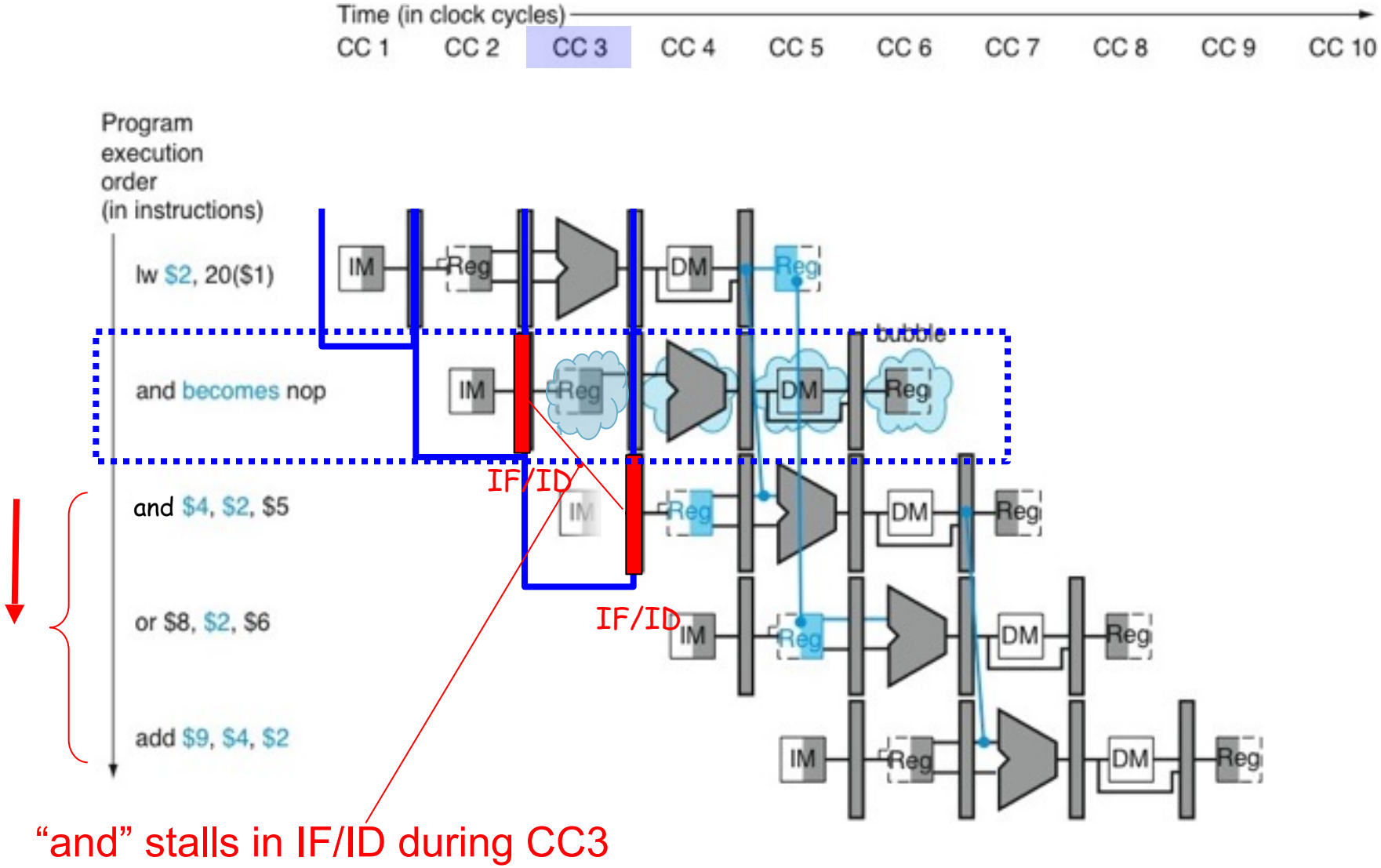
Unavoidable Stall Example:

Solution Load Interlock



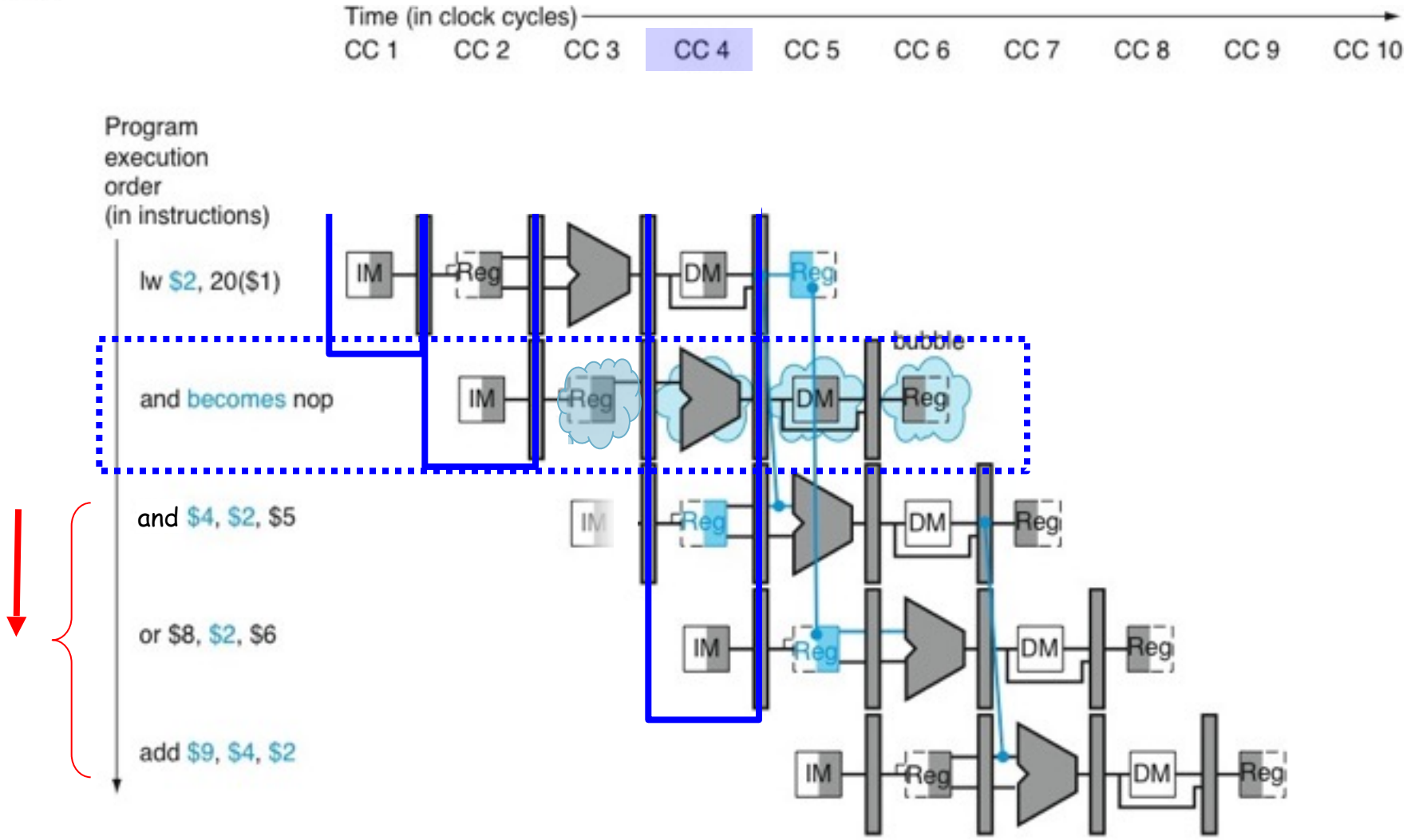
Unavoidable Stall Example:

Solution Load Interlock



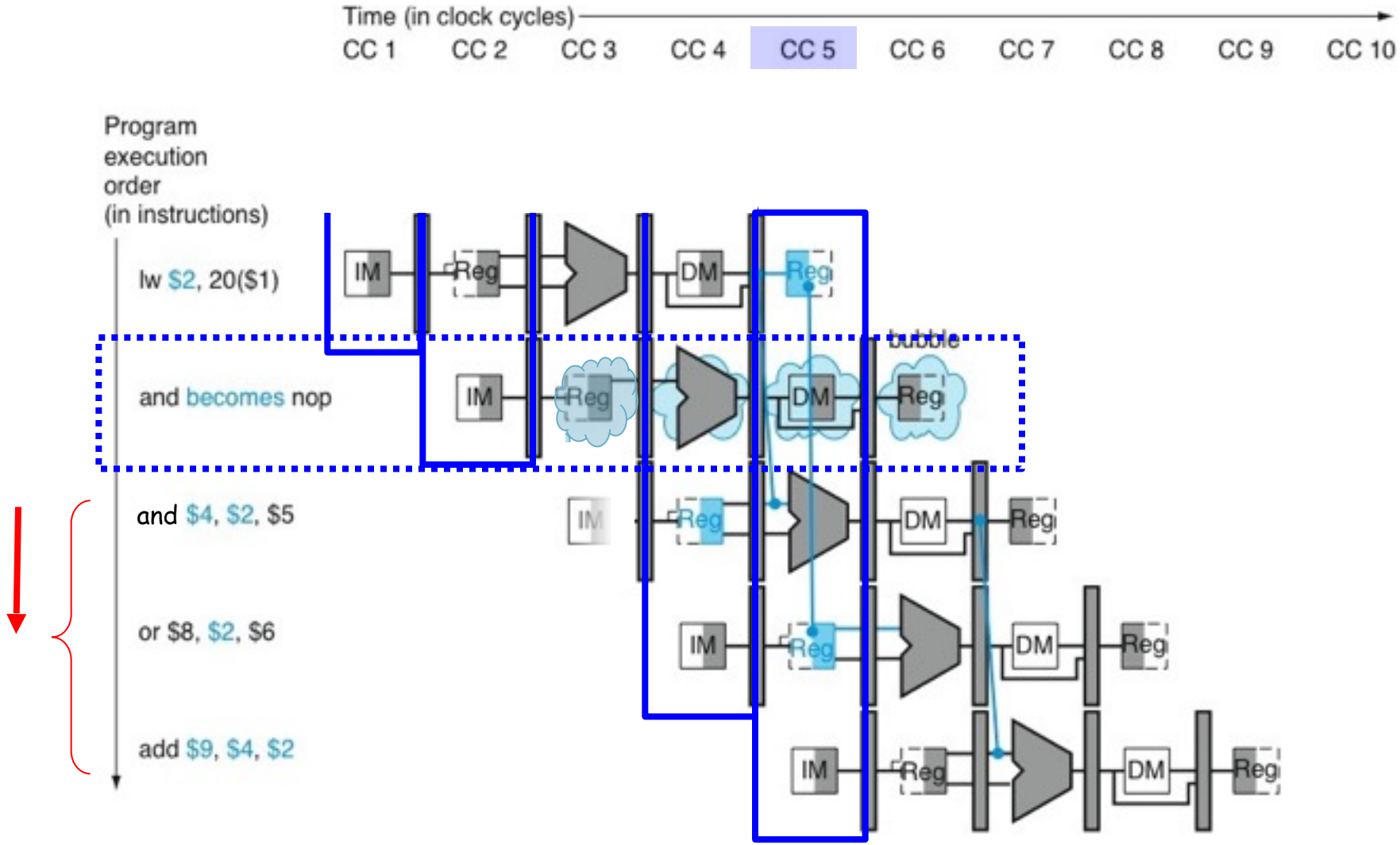
Unavoidable Stall Example:

Solution Load Interlock



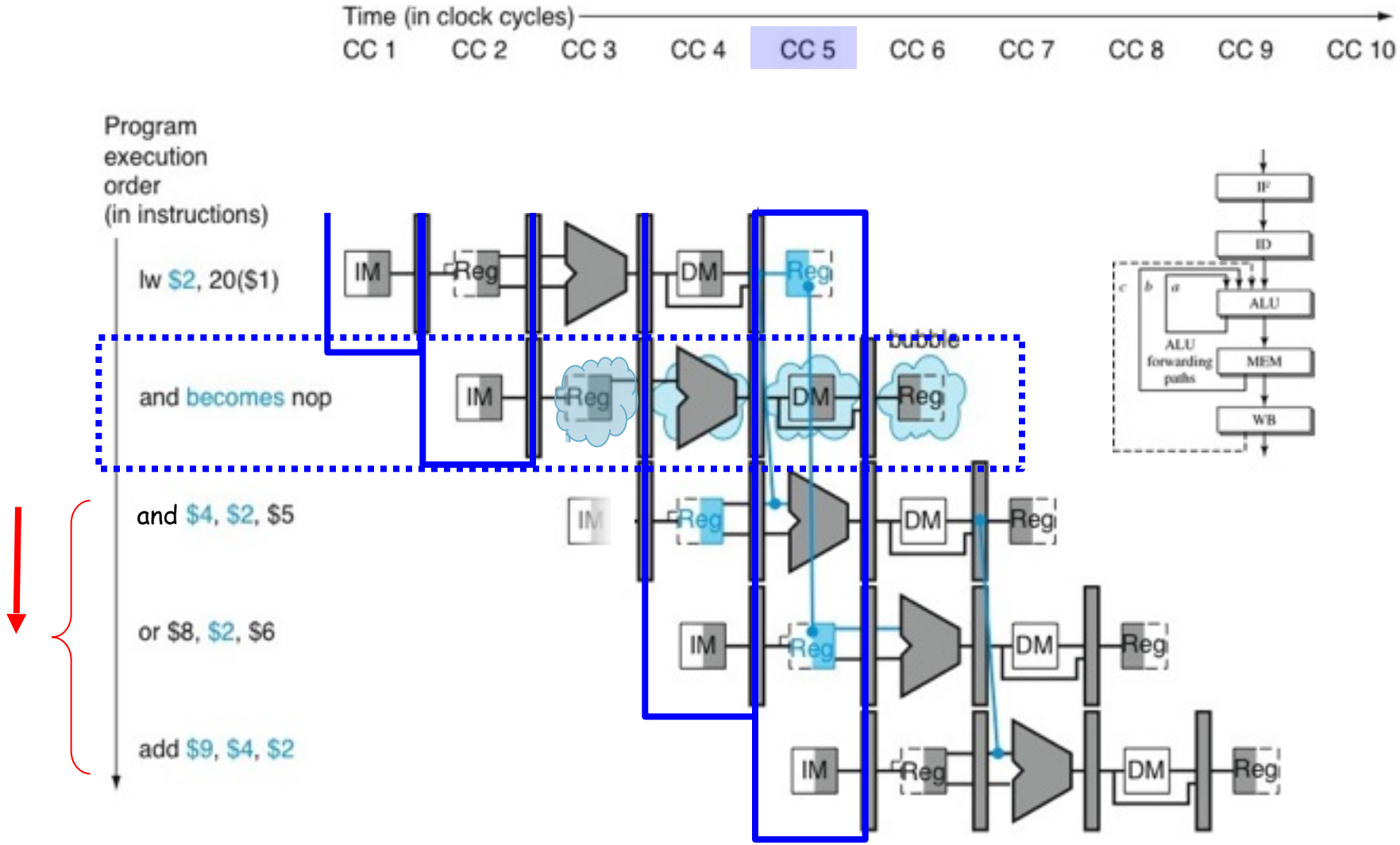
Unavoidable Stall Example:

Solution Load Interlock



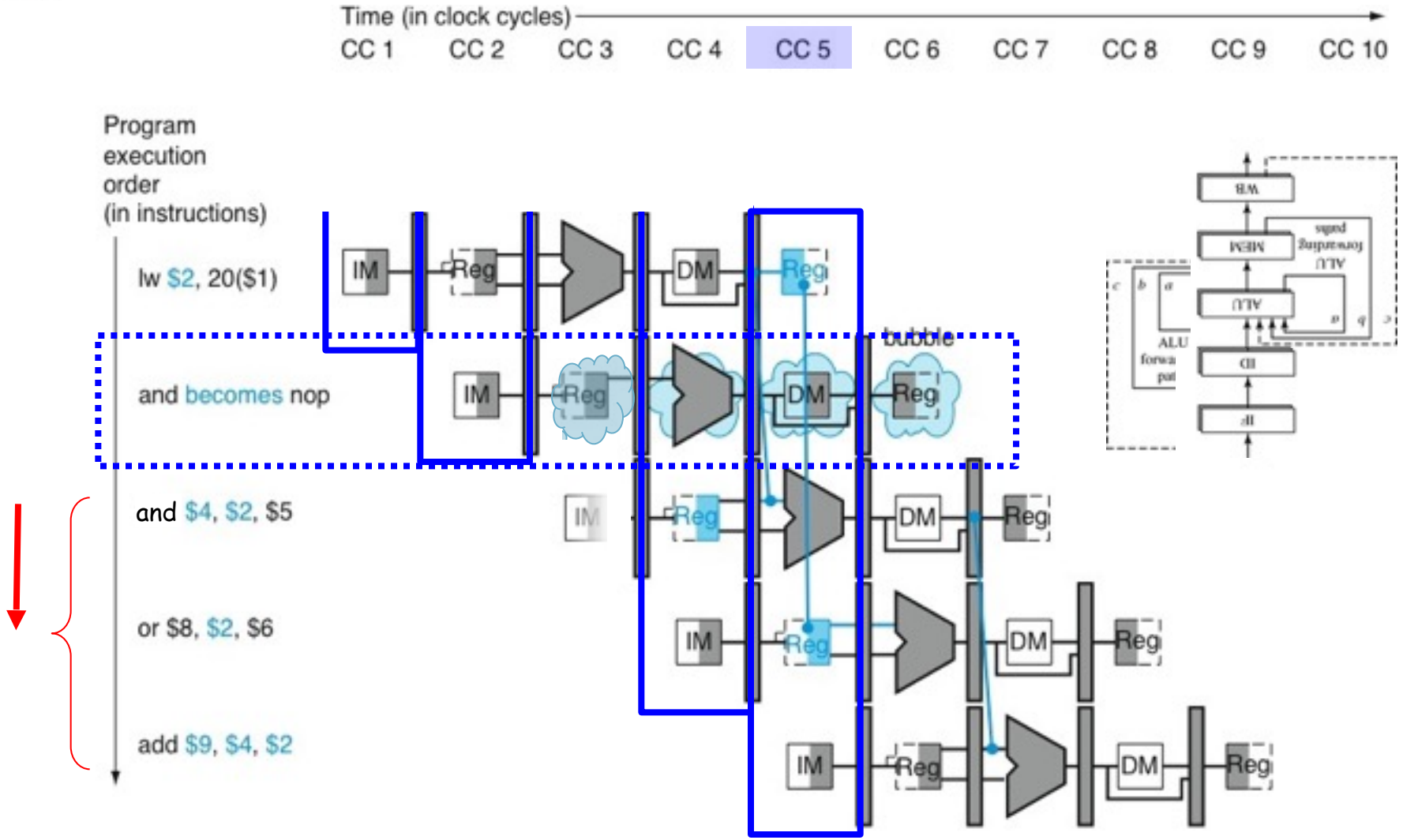
Unavoidable Stall Example:

Solution Load Interlock



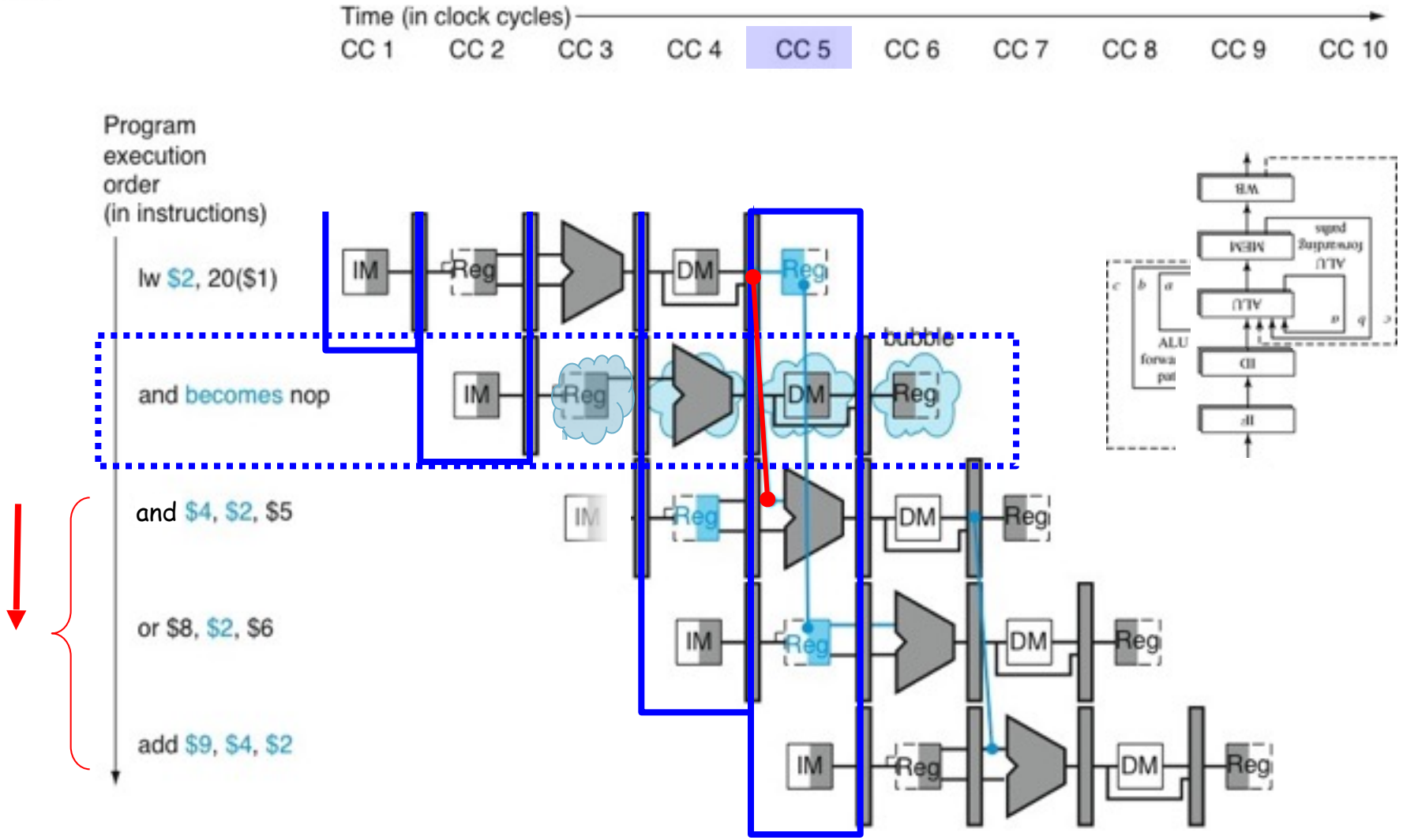
Unavoidable Stall Example:

Solution Load Interlock



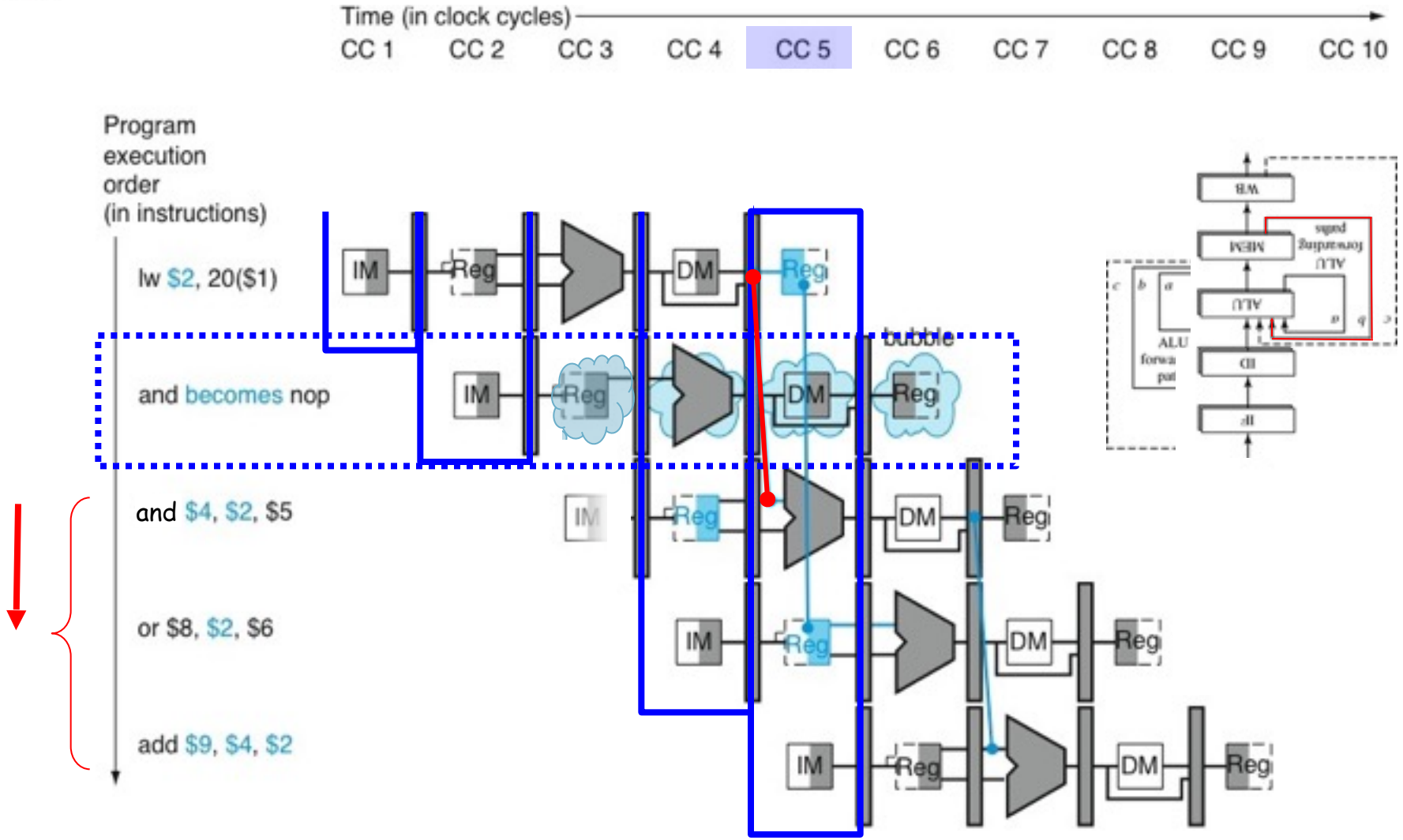
Unavoidable Stall Example:

Solution Load Interlock



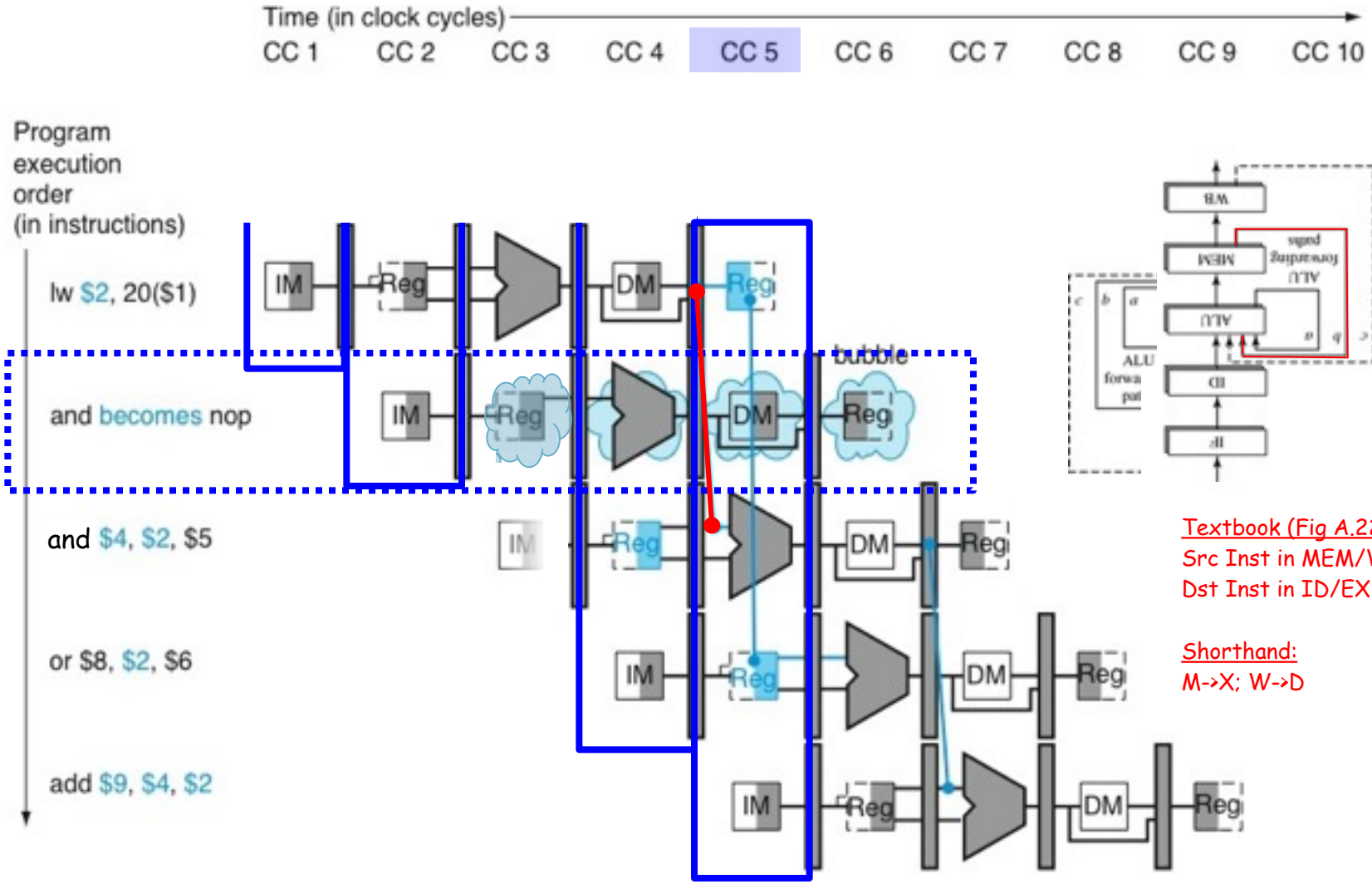
Unavoidable Stall Example:

Solution Load Interlock



Unavoidable Stall Example:

Solution Load Interlock

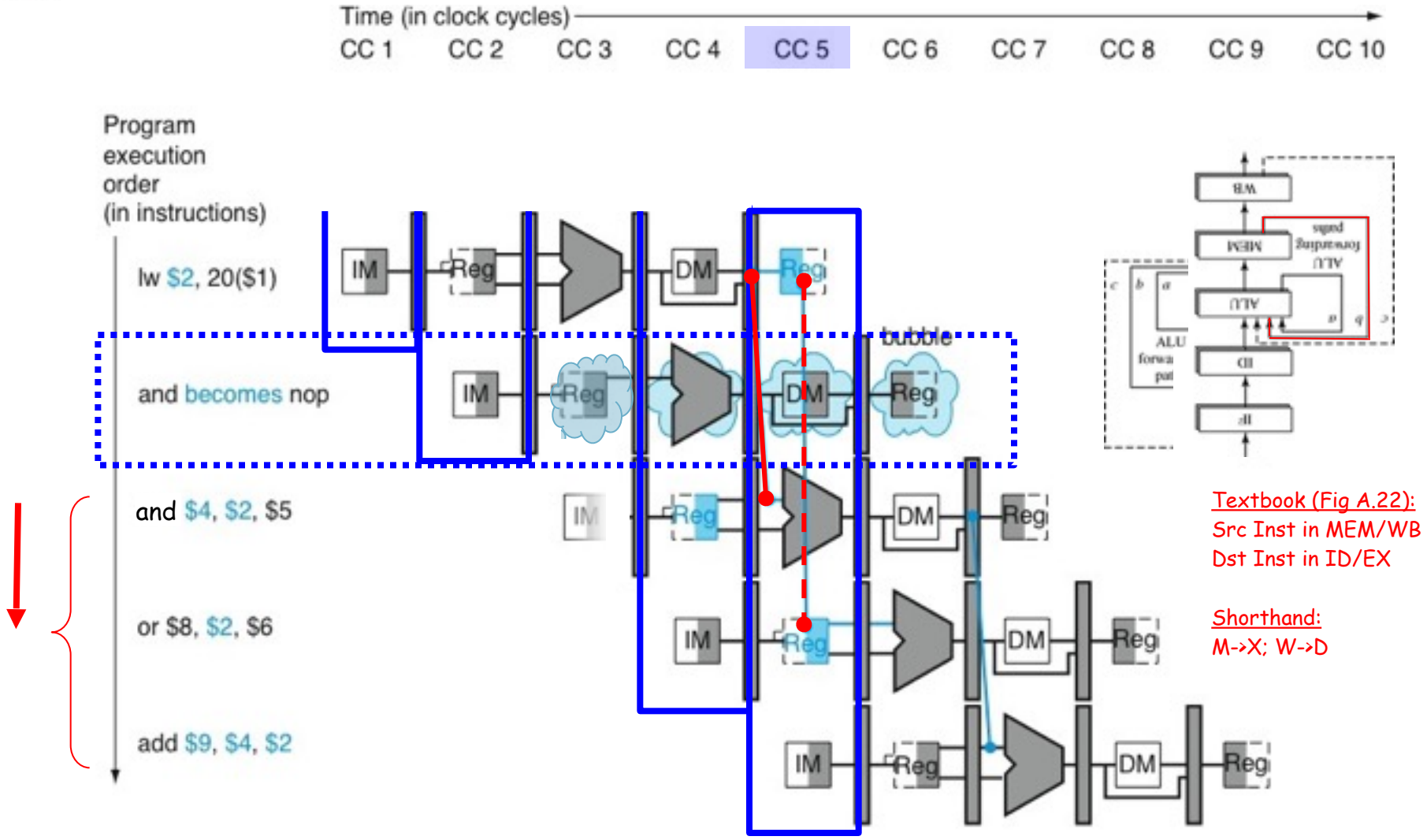


Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

Shorthand:
M->X; W->D

Unavoidable Stall Example:

Solution Load Interlock

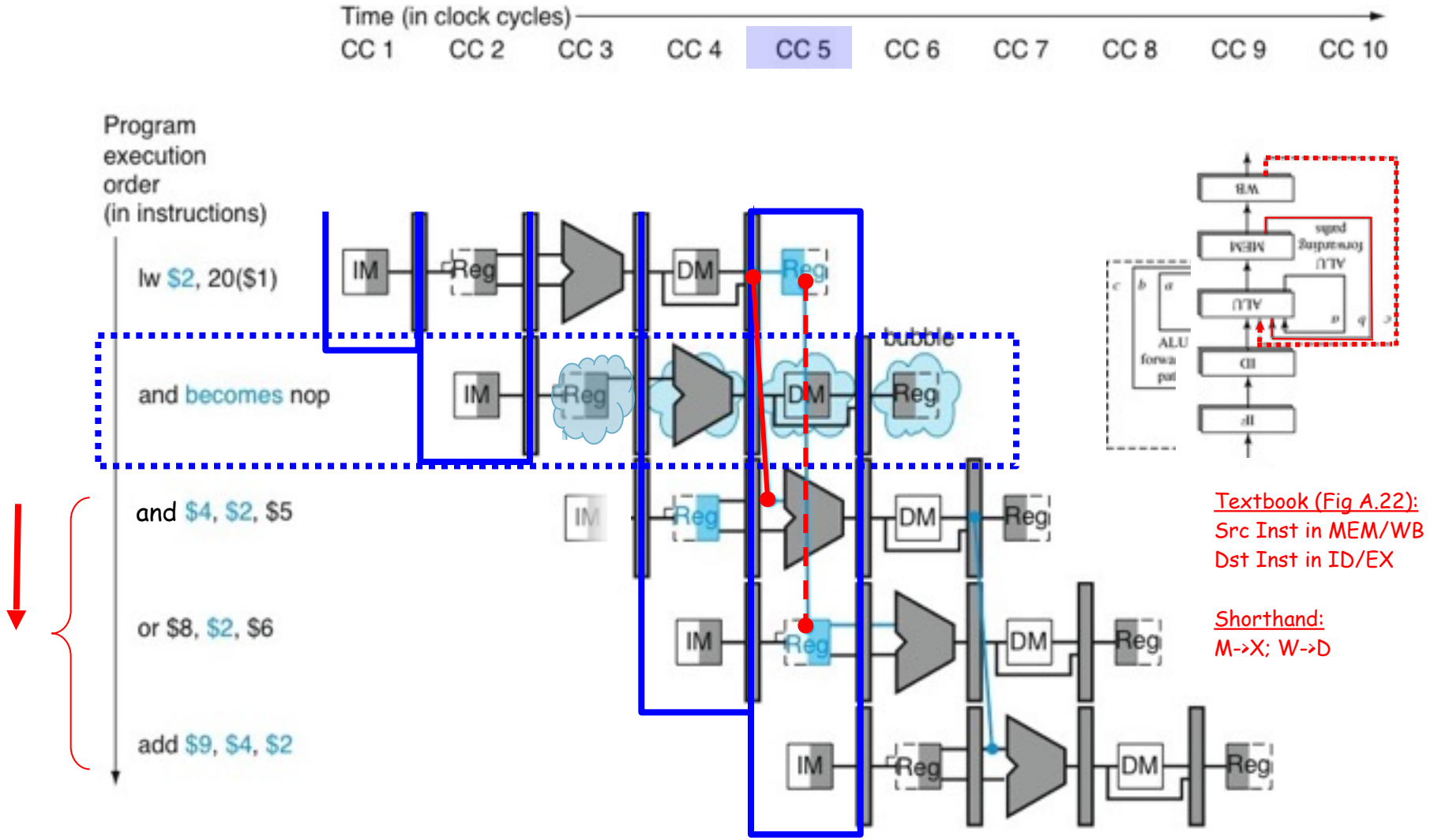


Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

Shorthand:
M->X; W->D

Unavoidable Stall Example:

Solution Load Interlock

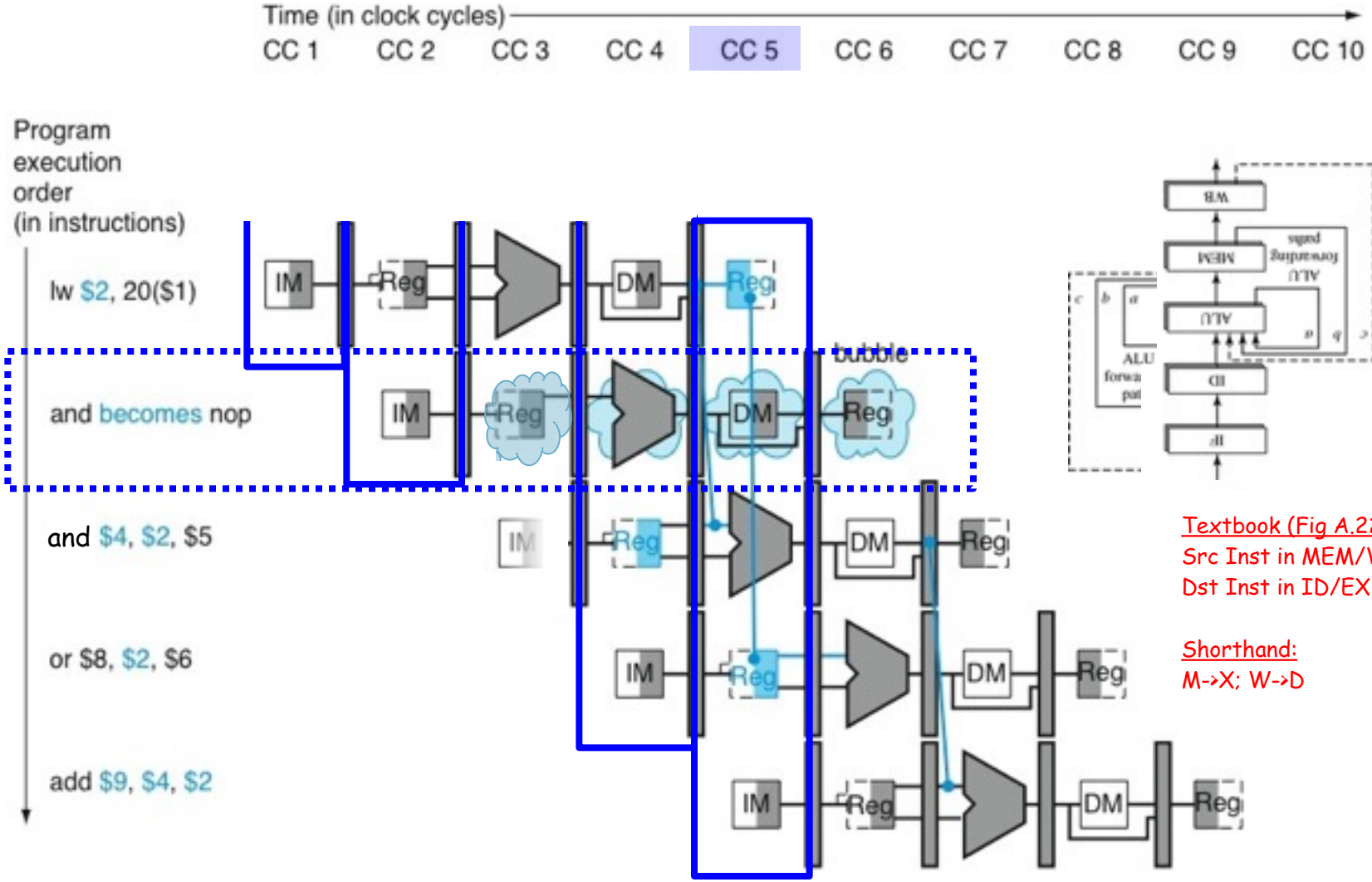


Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

Shorthand:
M->X; W->D

Unavoidable Stall Example:

Solution Load Interlock

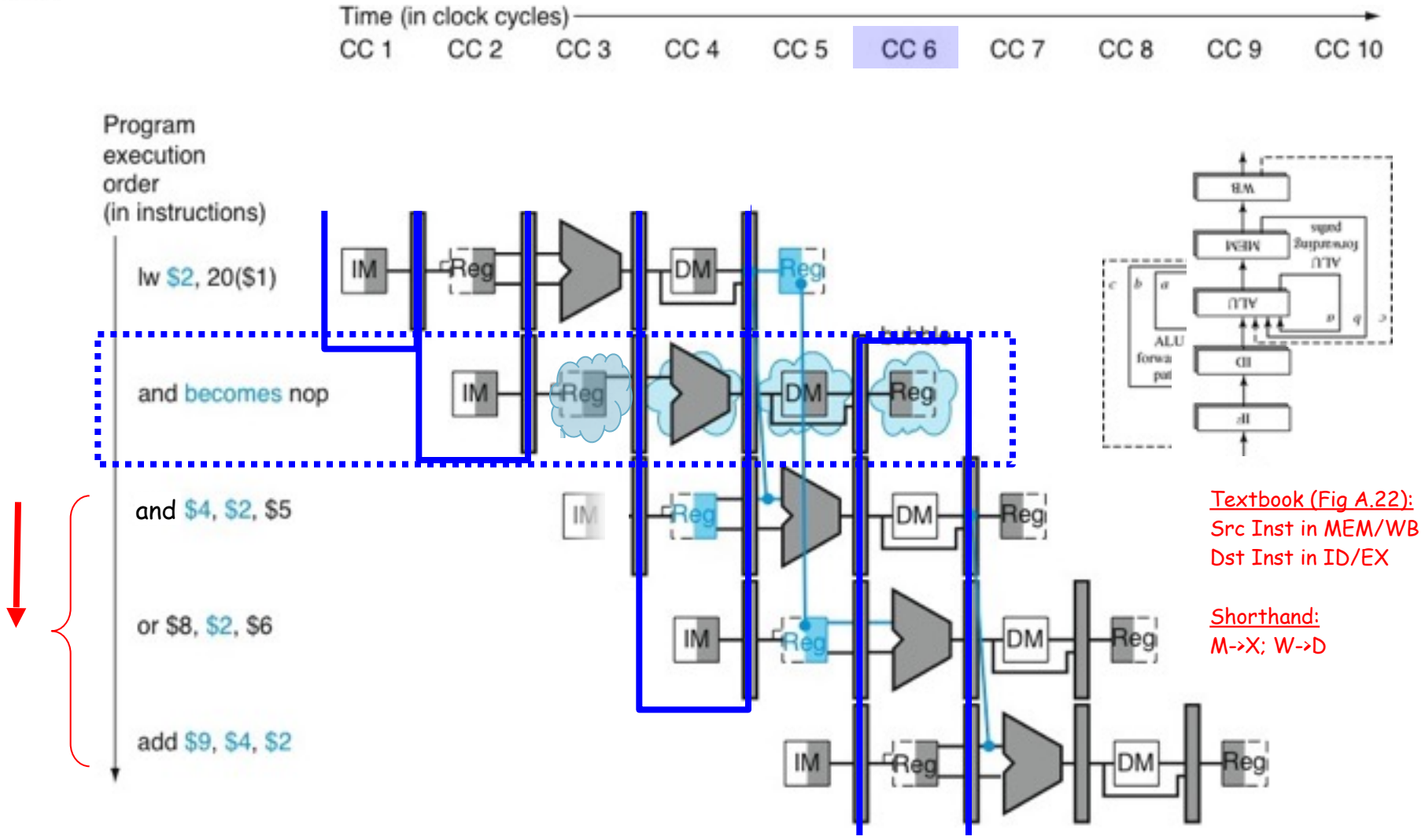


Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

Shorthand:
M->X; W->D

Unavoidable Stall Example:

Solution Load Interlock

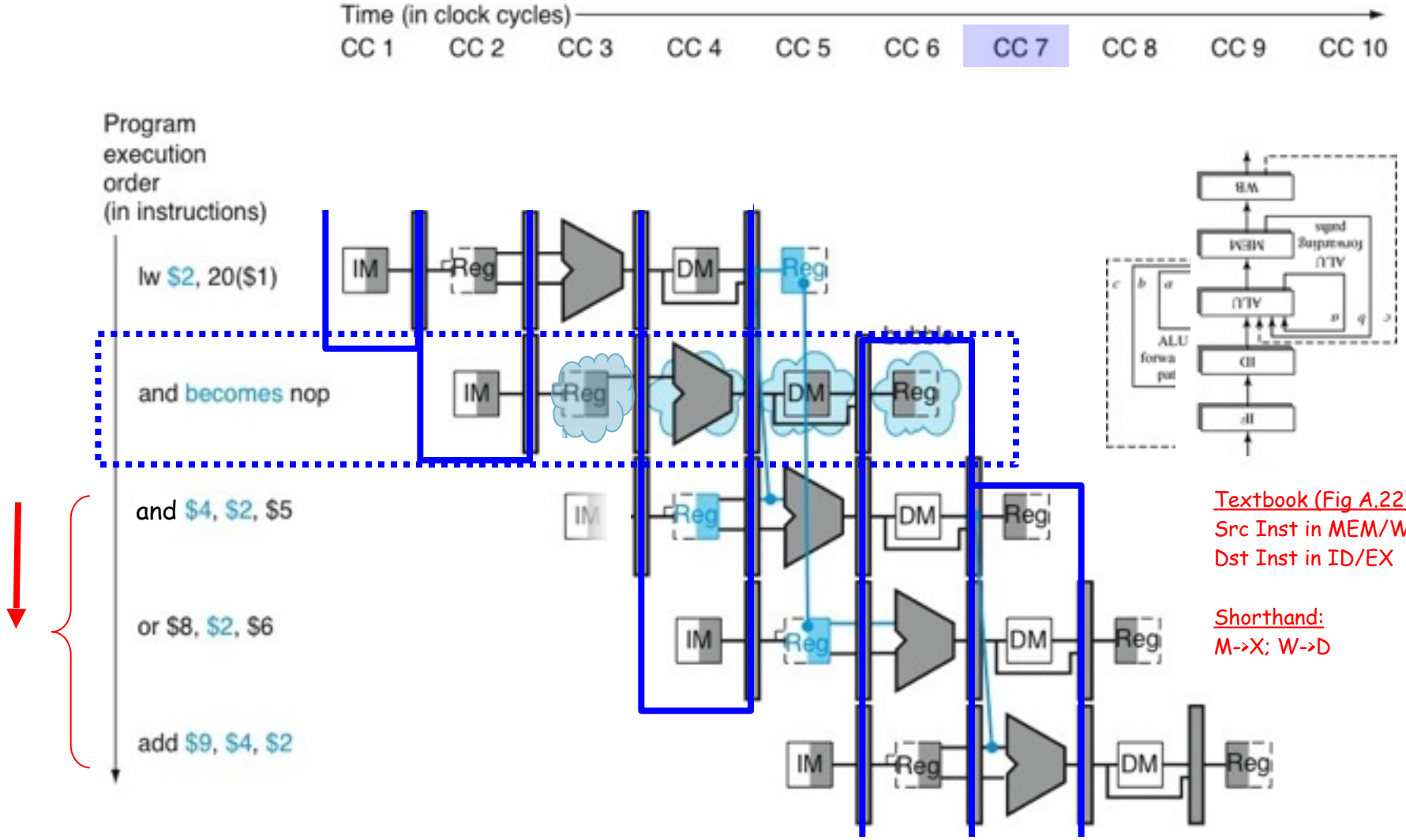


Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

Shorthand:
M->X; W->D

Unavoidable Stall Example:

Solution Load Interlock

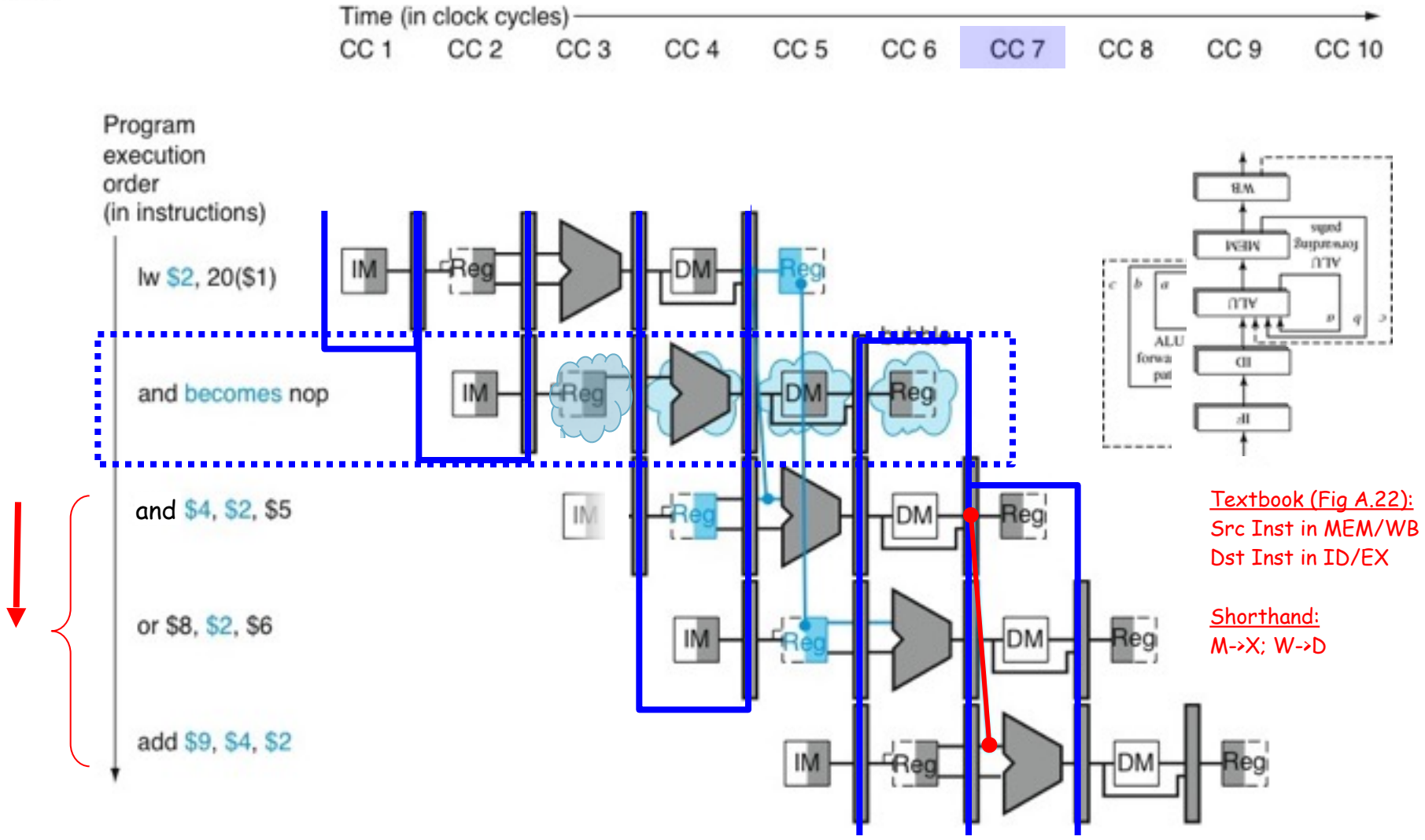


Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

Shorthand:
M->X; W->D

Unavoidable Stall Example:

Solution Load Interlock

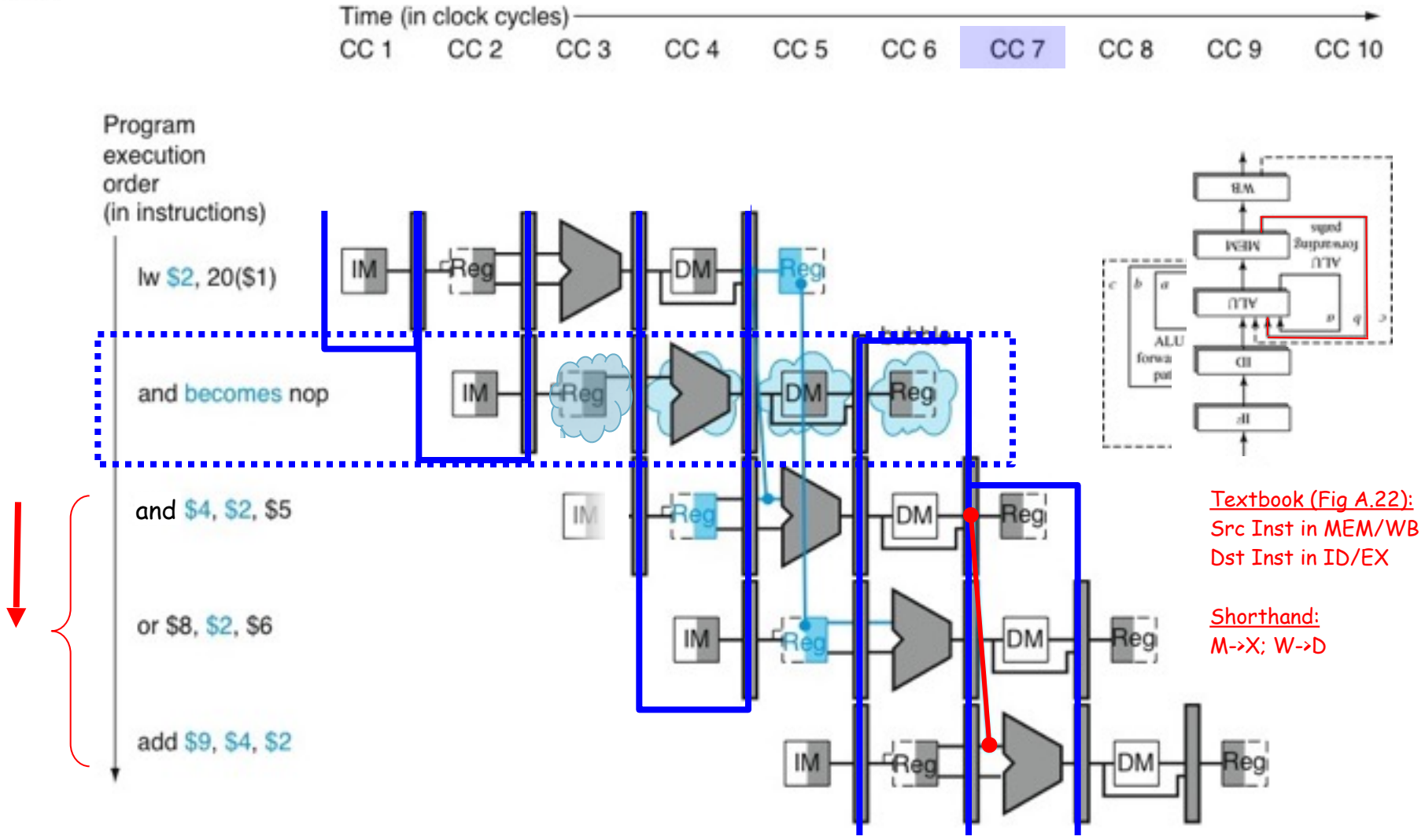


Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

Shorthand:
M->X; W->D

Unavoidable Stall Example:

Solution Load Interlock

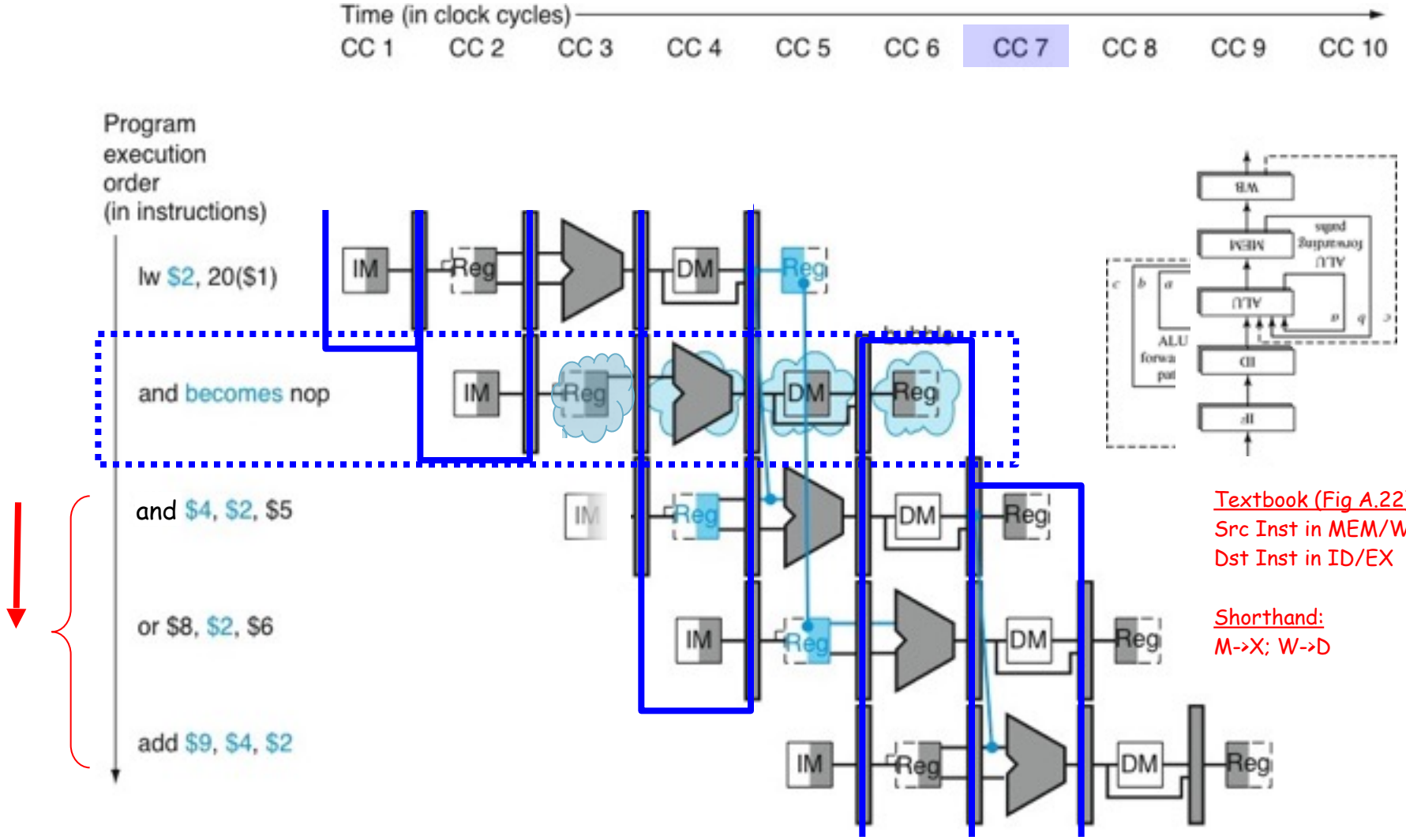


Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

Shorthand:
M->X; W->D

Unavoidable Stall Example:

Solution Load Interlock

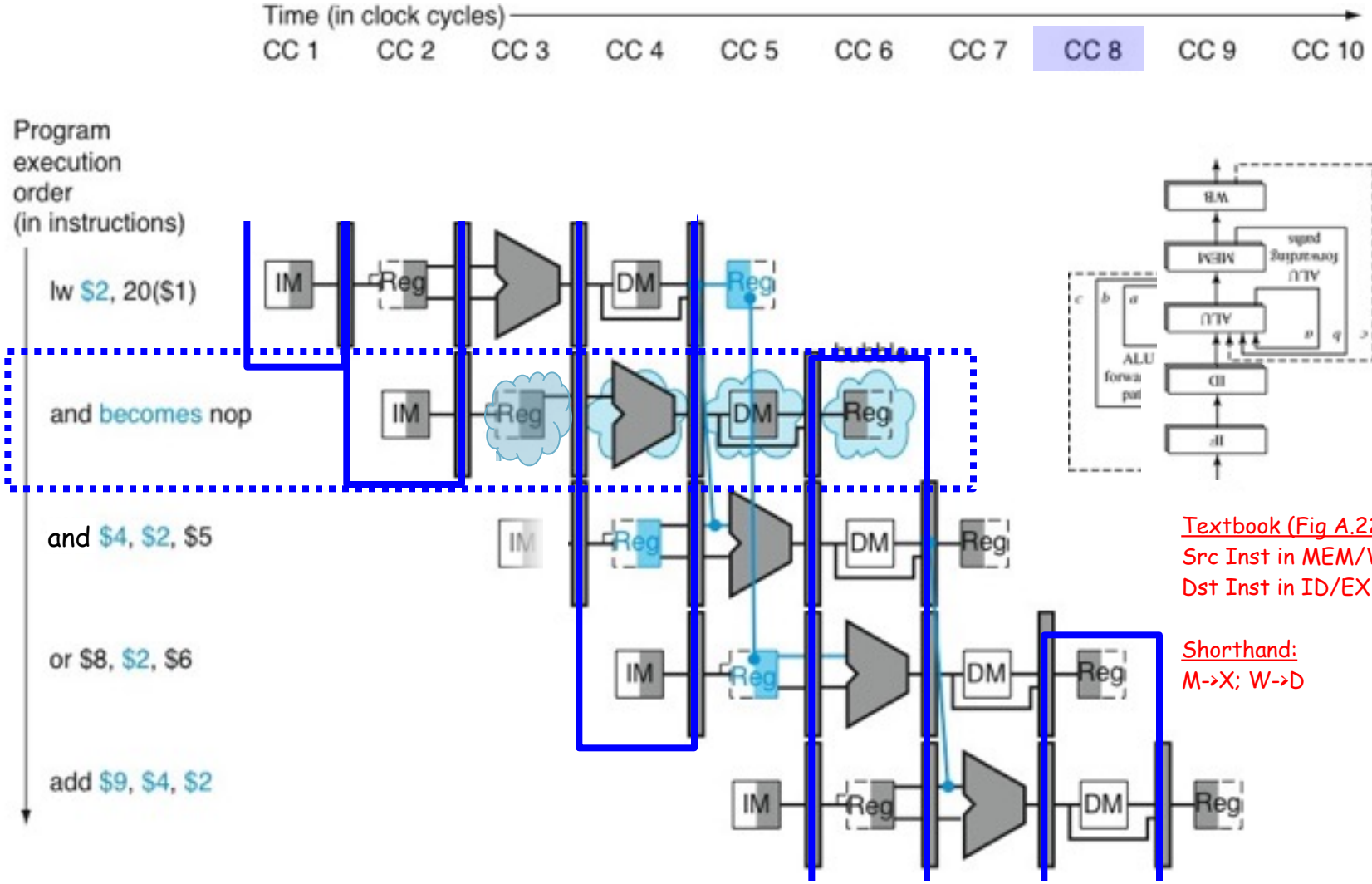


Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

Shorthand:
M->X; W->D

Unavoidable Stall Example:

Solution Load Interlock

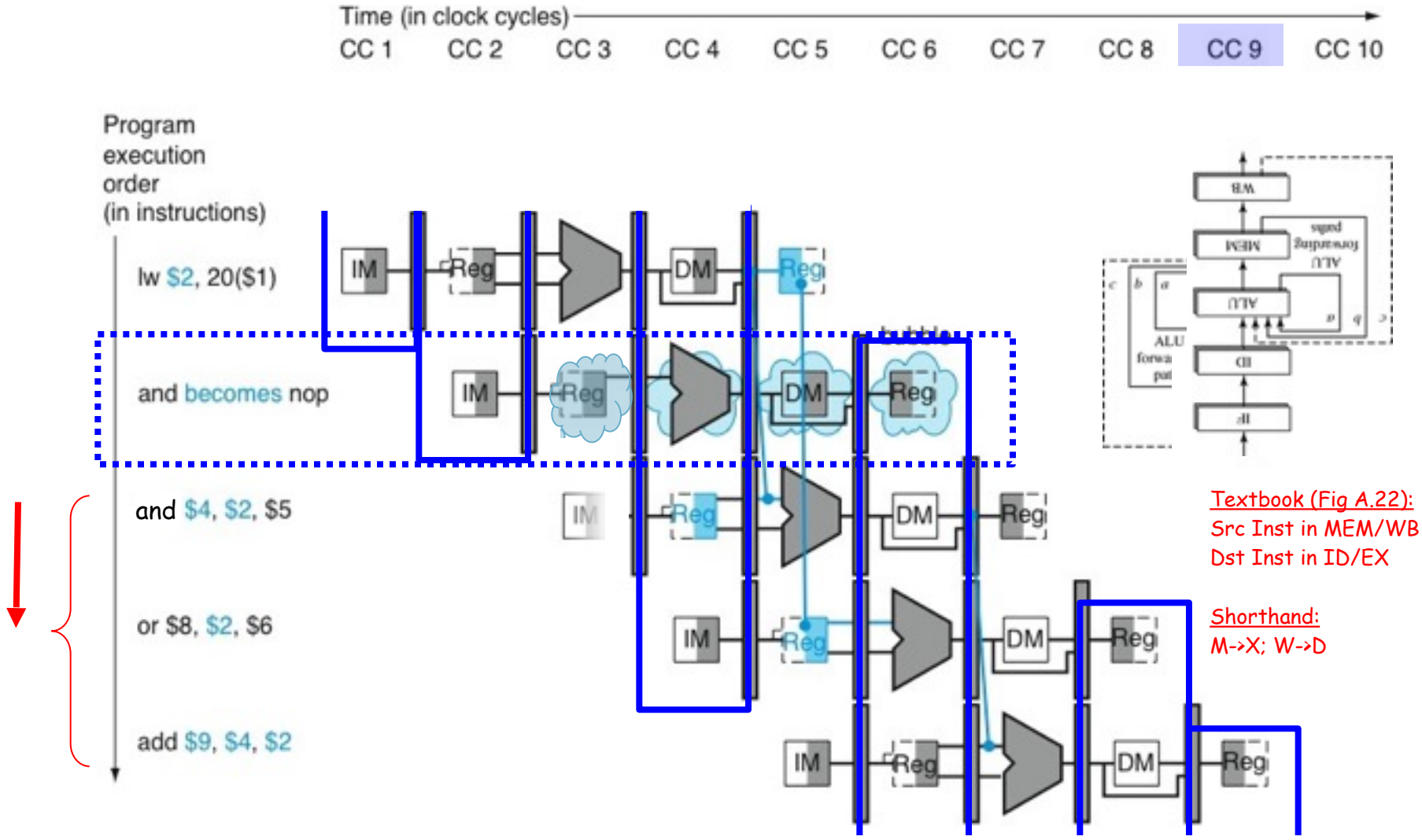


Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

Shorthand:
M->X; W->D

Unavoidable Stall Example:

Solution Load Interlock



Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

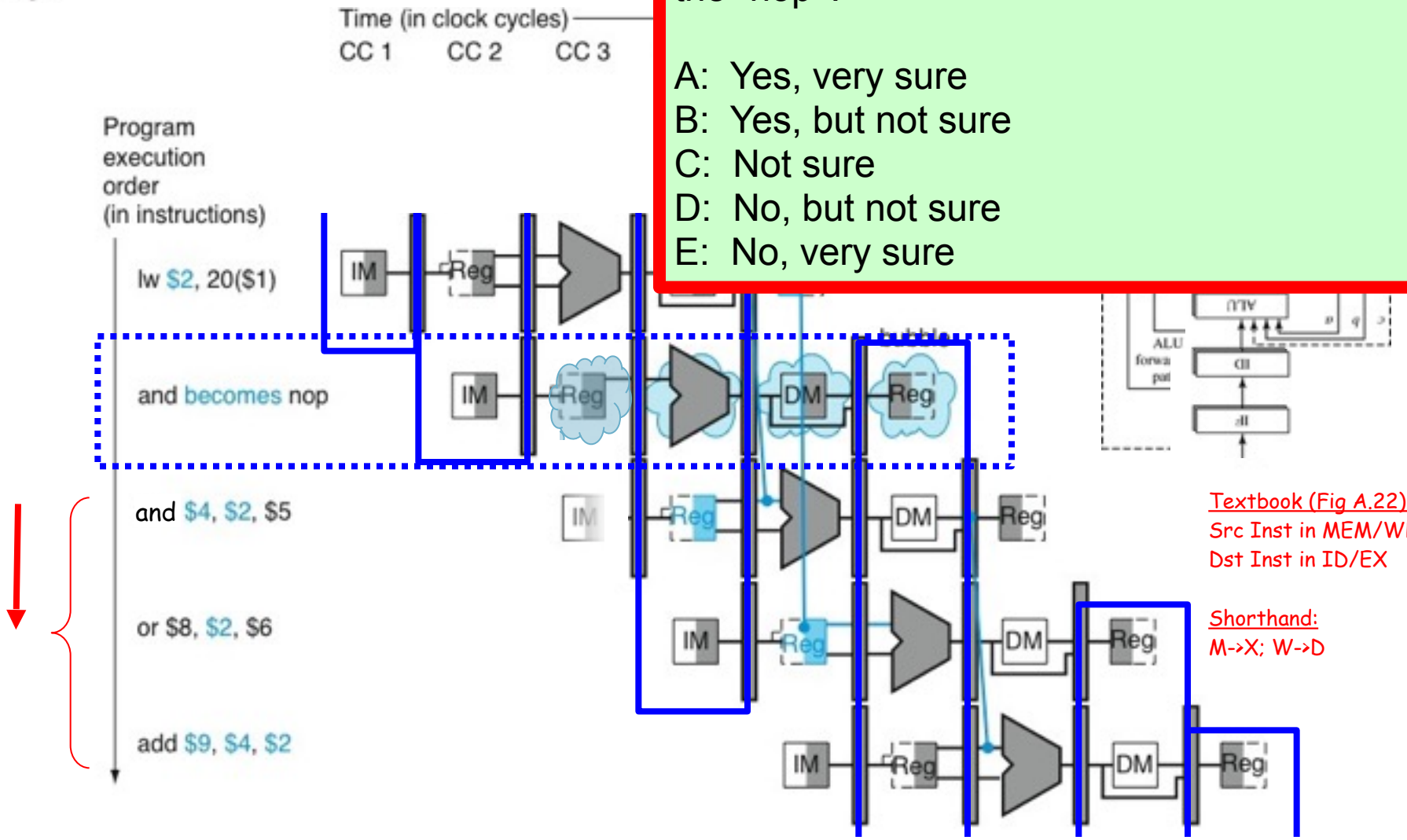
Shorthand:
M->X; W->D

Unavoid

Solution

In this example, was the assembly code modified (e.g., by the programmer) to include the “nop”?

A: Yes, very sure
B: Yes, but not sure
C: Not sure
D: No, but not sure
E: No, very sure



Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

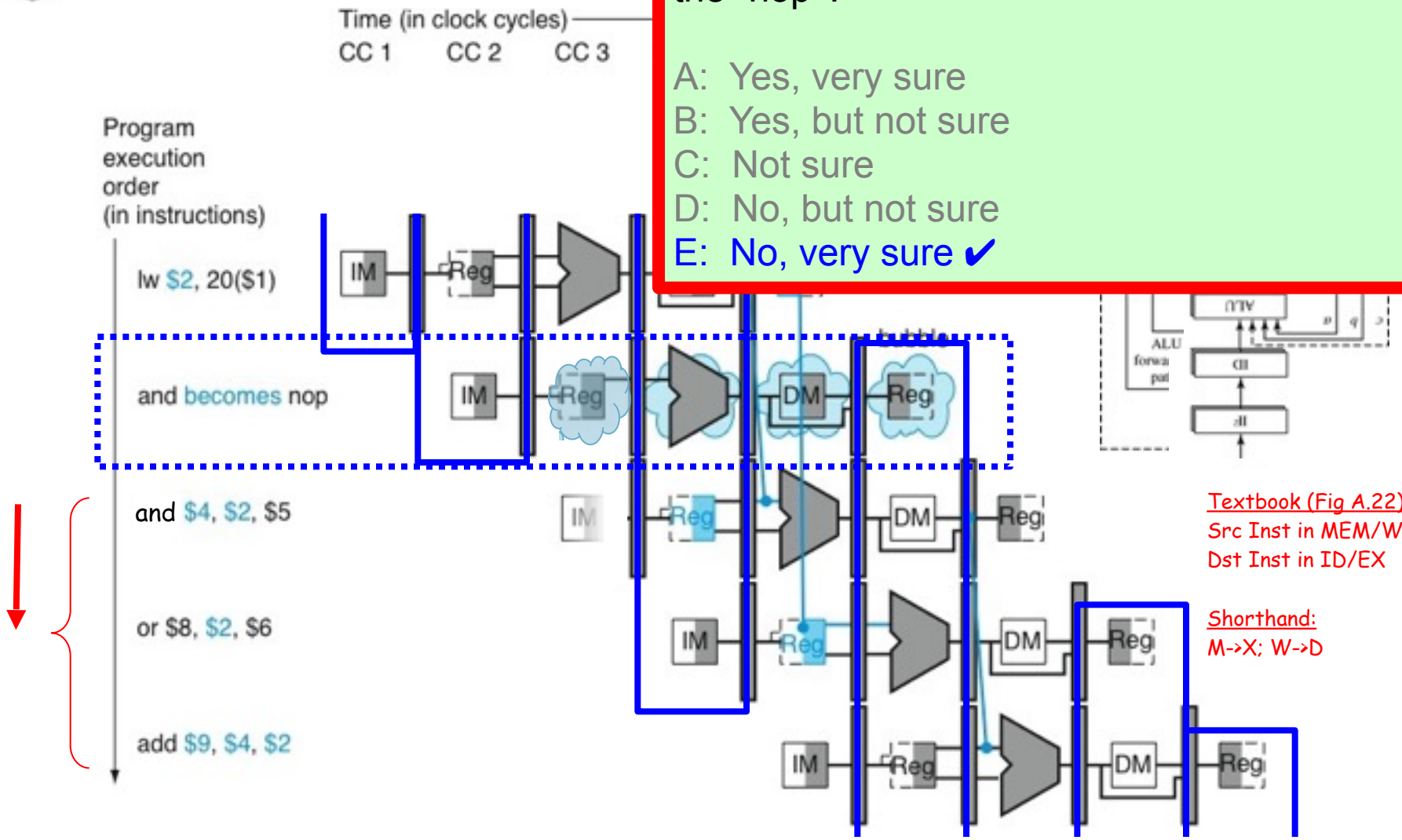
Shorthand:
M->X; W->D

Unavoid

Solution

In this example, was the assembly code modified (e.g., by the programmer) to include the “nop”?

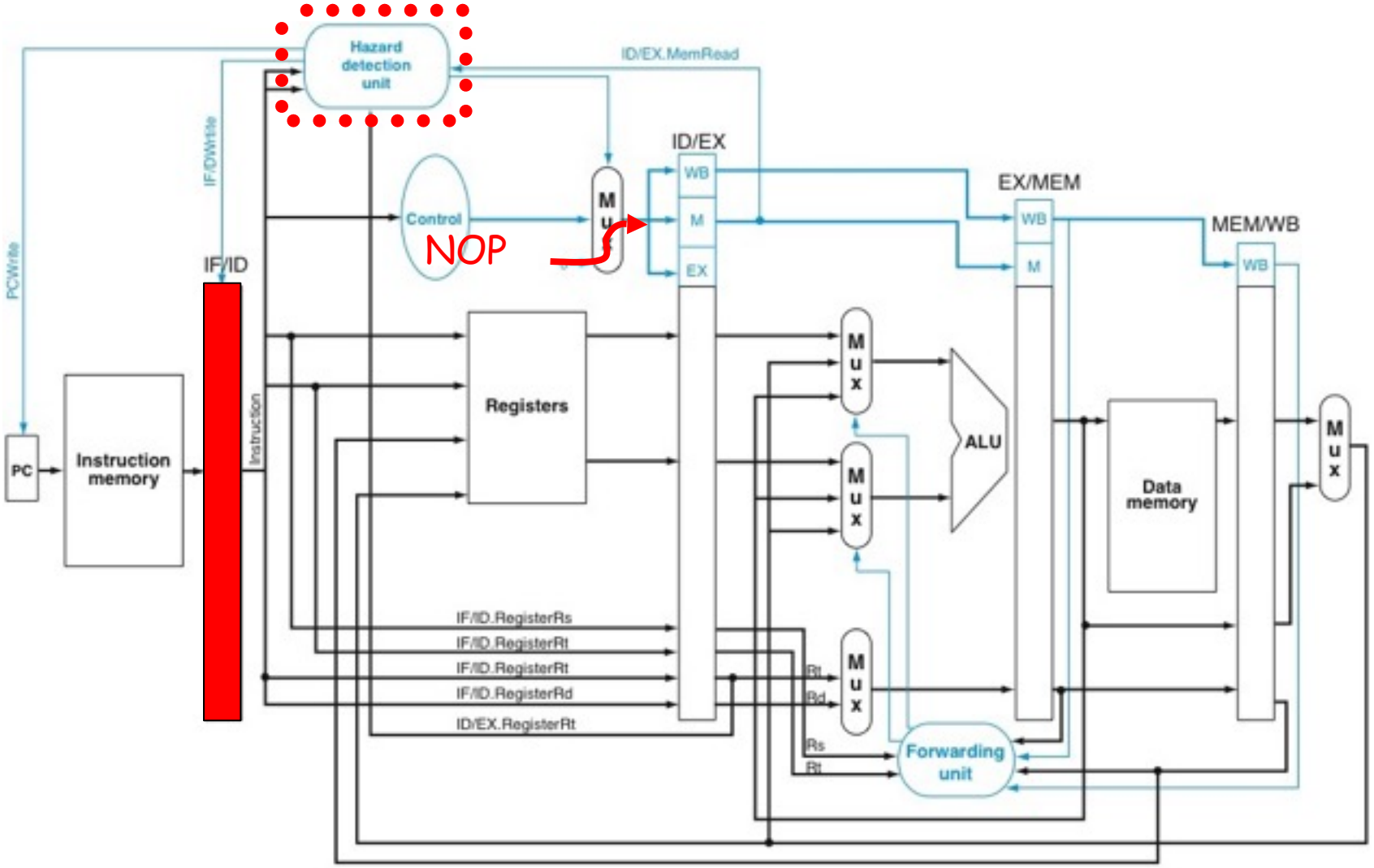
A: Yes, very sure
B: Yes, but not sure
C: Not sure
D: No, but not sure
E: No, very sure ✔



Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

Shorthand:
M->X; W->D

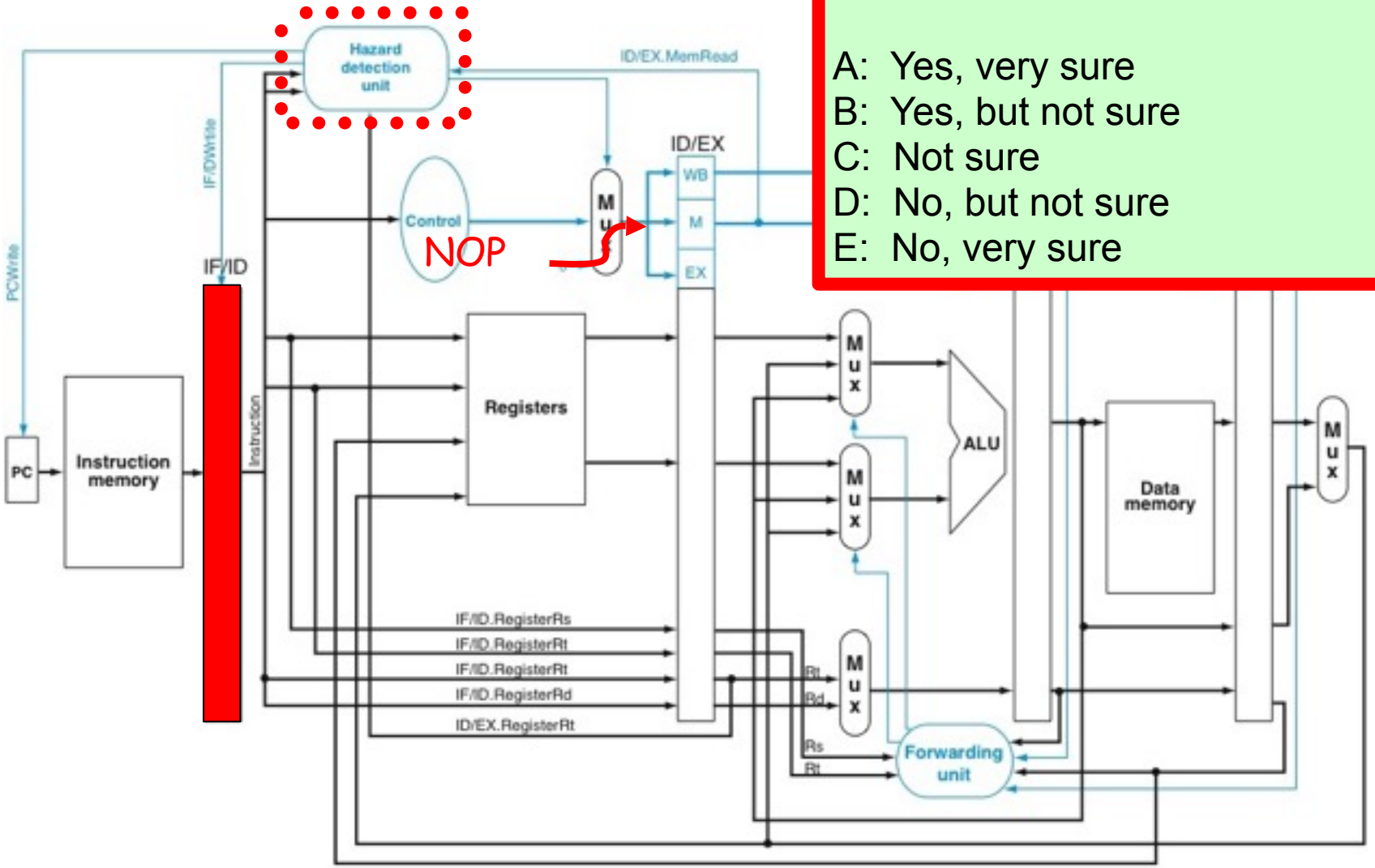
Stall Hardware: Hazard Detection Unit



Stall Hardware: Hazar

Does hazard detection unit control muxes used to forward values from one instruction to another?

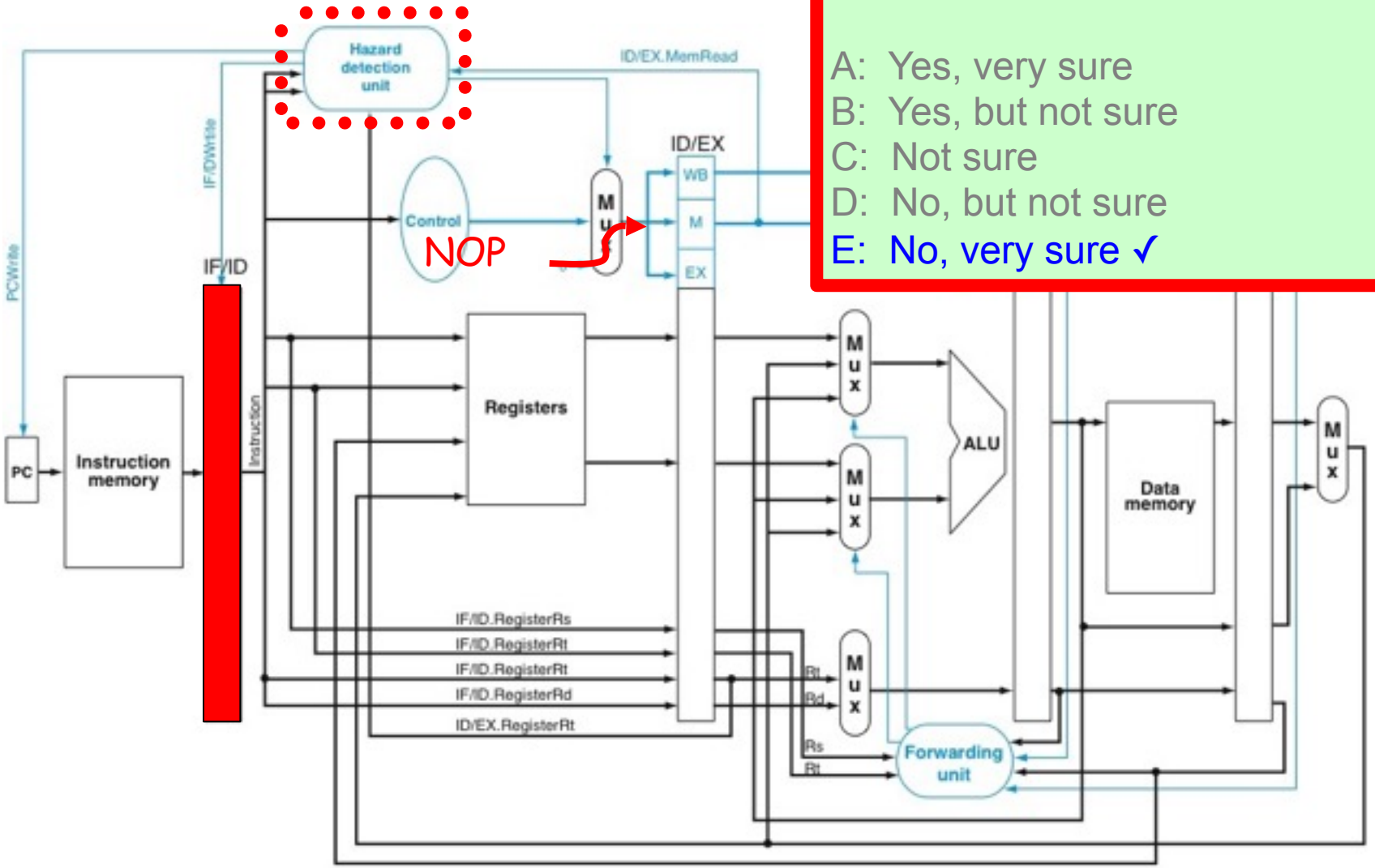
A: Yes, very sure
B: Yes, but not sure
C: Not sure
D: No, but not sure
E: No, very sure



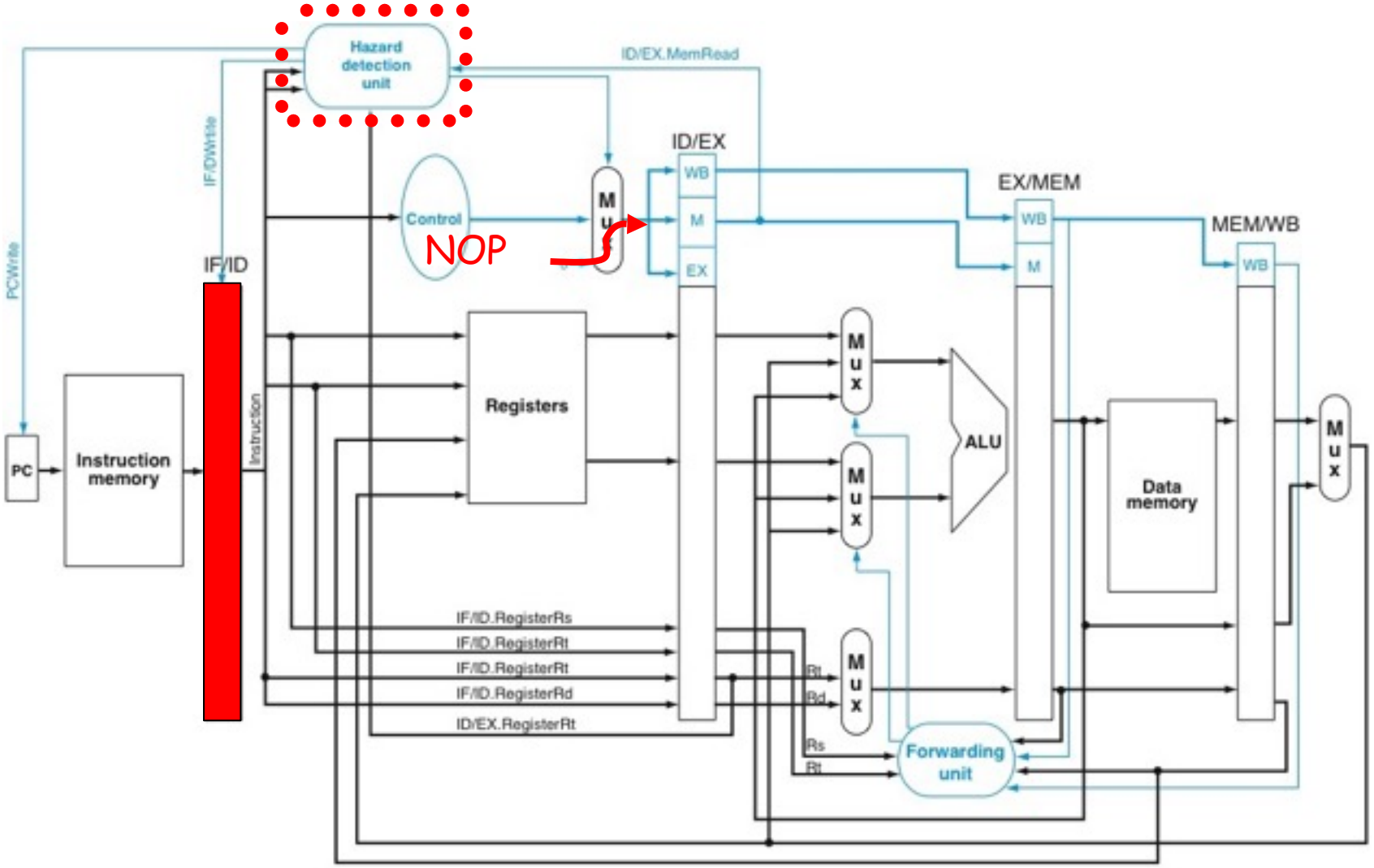
Stall Hardware: Hazar

Does hazard detection unit control muxes used to forward values from one instruction to another?

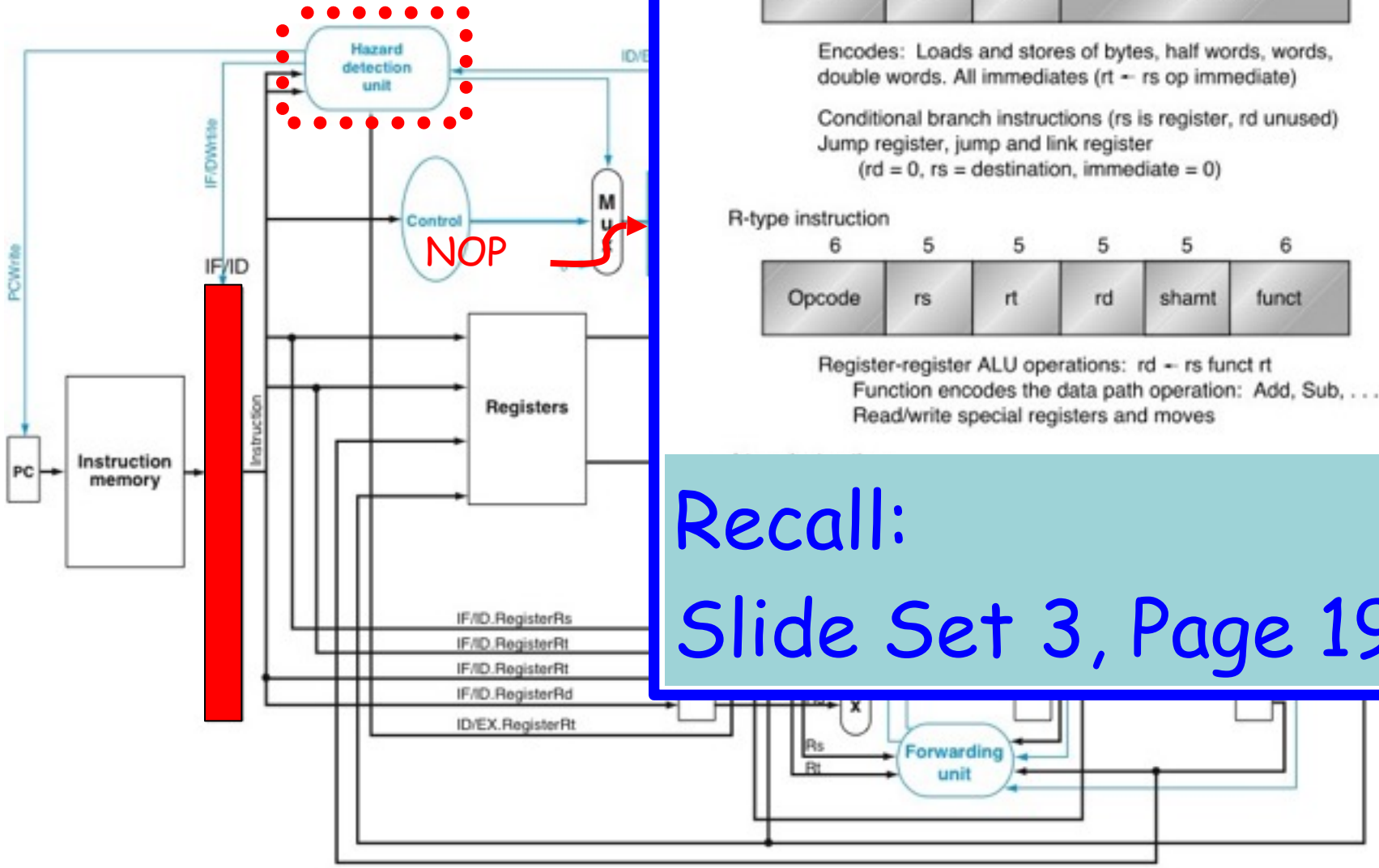
A: Yes, very sure
B: Yes, but not sure
C: Not sure
D: No, but not sure
E: No, very sure ✓



Stall Hardware: Hazard Detection Unit



Stall Hardware:



I-type instruction

6	5	5	16
Opcode	rs	rt	Immediate

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction

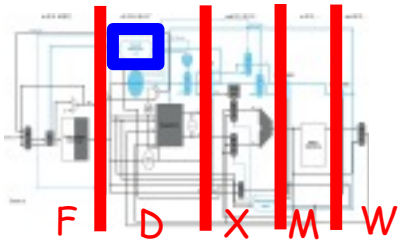
6	5	5	5	5	6
Opcode	rs	rt	rd	shamt	funct

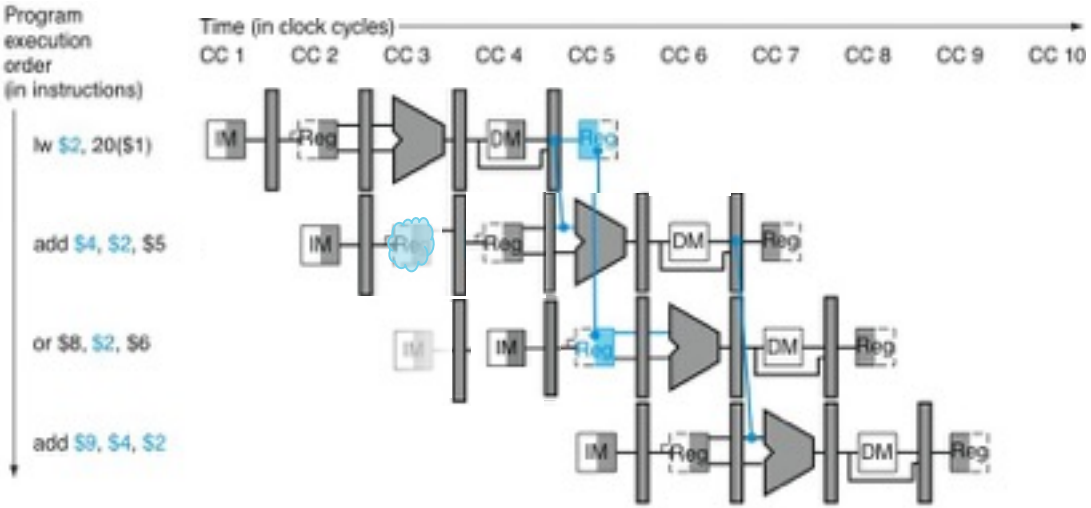
Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

Recall:
Slide Set 3, Page 19

Hazard Detection Location Options

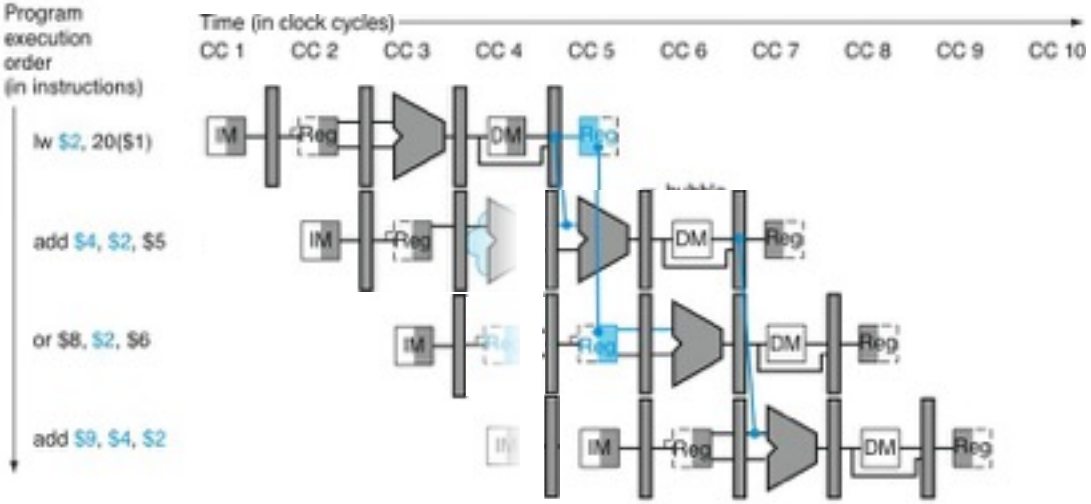
Hazard detection in **decode**:





"and" instruction stalled in IF/ID during CC3

Hazard detection in **execute**:



"and" instruction stalled in ID/EX during CC4

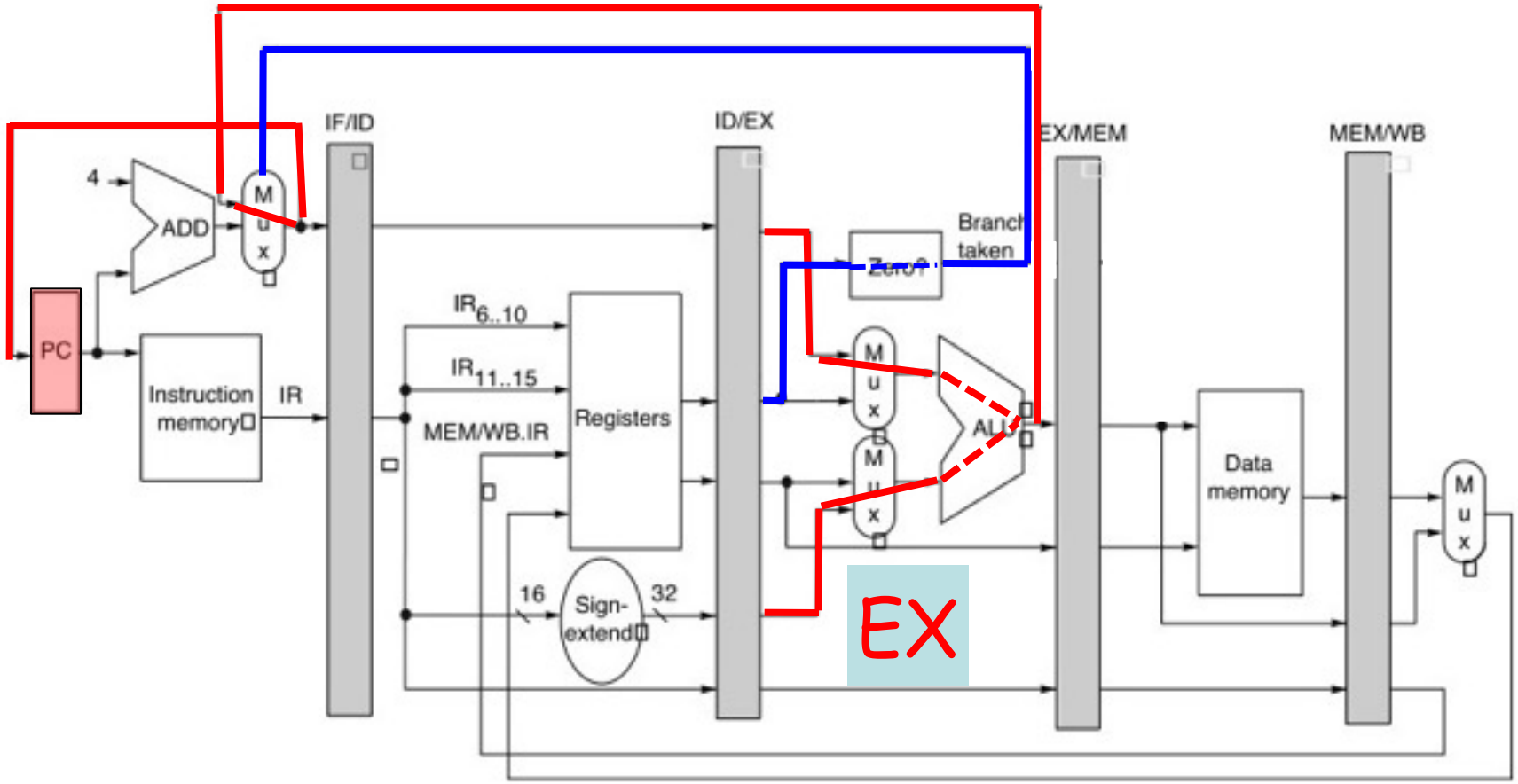
May lower clock frequency (need to send stall signal farther).

Unavoidable Data Hazard

	Clock Number								
	1	2	3	4	5	6	7	8	9
LD R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB R4,R1,R5		IF	ID	EX	MEM	WB			
AND R6,R8,R7			IF	ID	EX	MEM	WB		
OR R8,R10,R9				IF	ID	EX	MEM	WB	

	Clock Number								
	1	2	3	4	5	6	7	8	9
LD R1,0(R2)									
DSUB R4,R1,R5									
AND R6,R8,R7									
OR R8,R10,R9									

Resolving a Branch in Execute (EX)

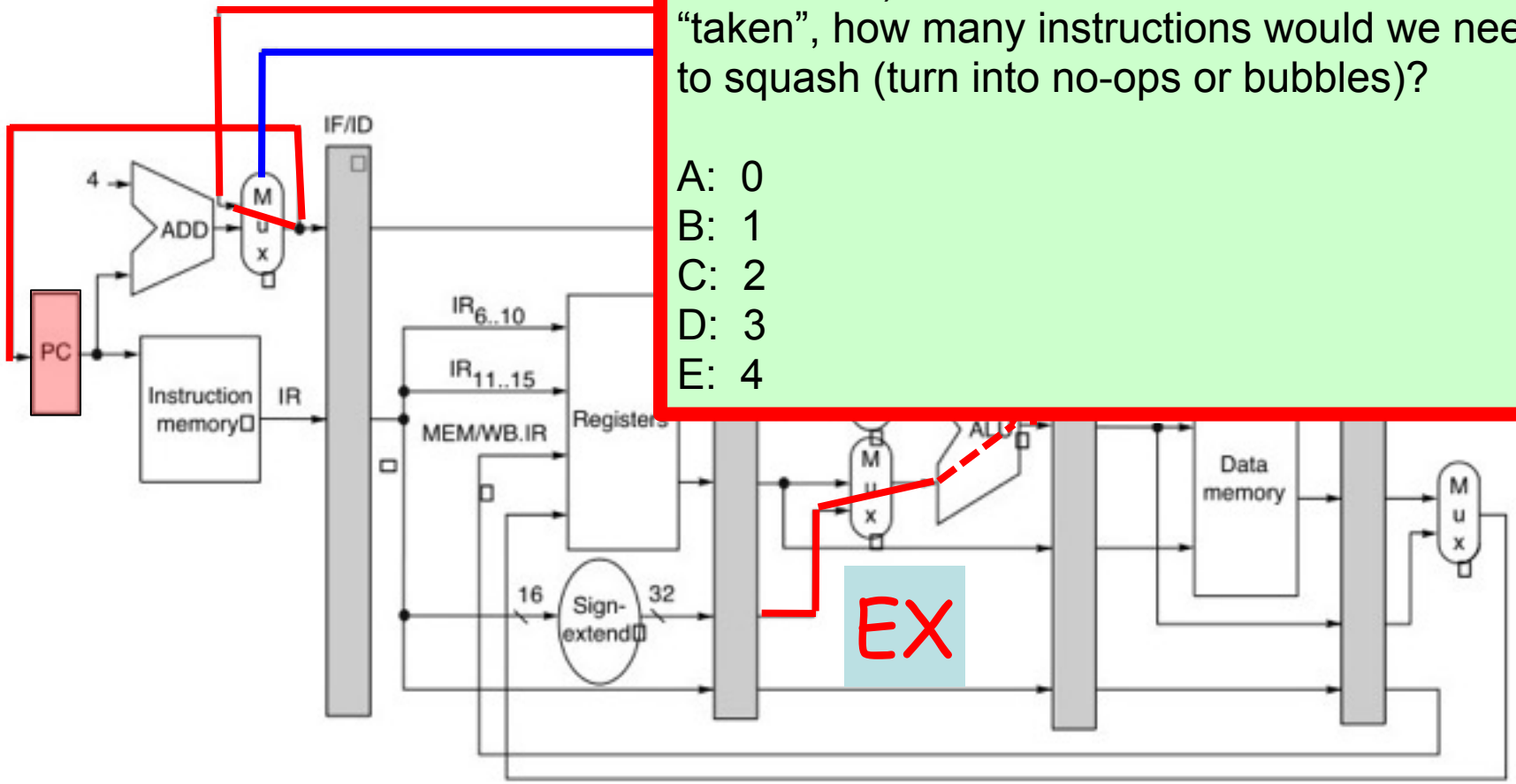


Check branch condition and compute target in execute stage.
Update PC with correct target by end of cycle that branch enters execute stage. Fetch correct target following cycle.

Resolving a Br

Assume branches are resolved in execute and we predict branches are "not taken" (as in Slide Set 4). If a branch turns out to be "taken", how many instructions would we need to squash (turn into no-ops or bubbles)?

A: 0
B: 1
C: 2
D: 3
E: 4

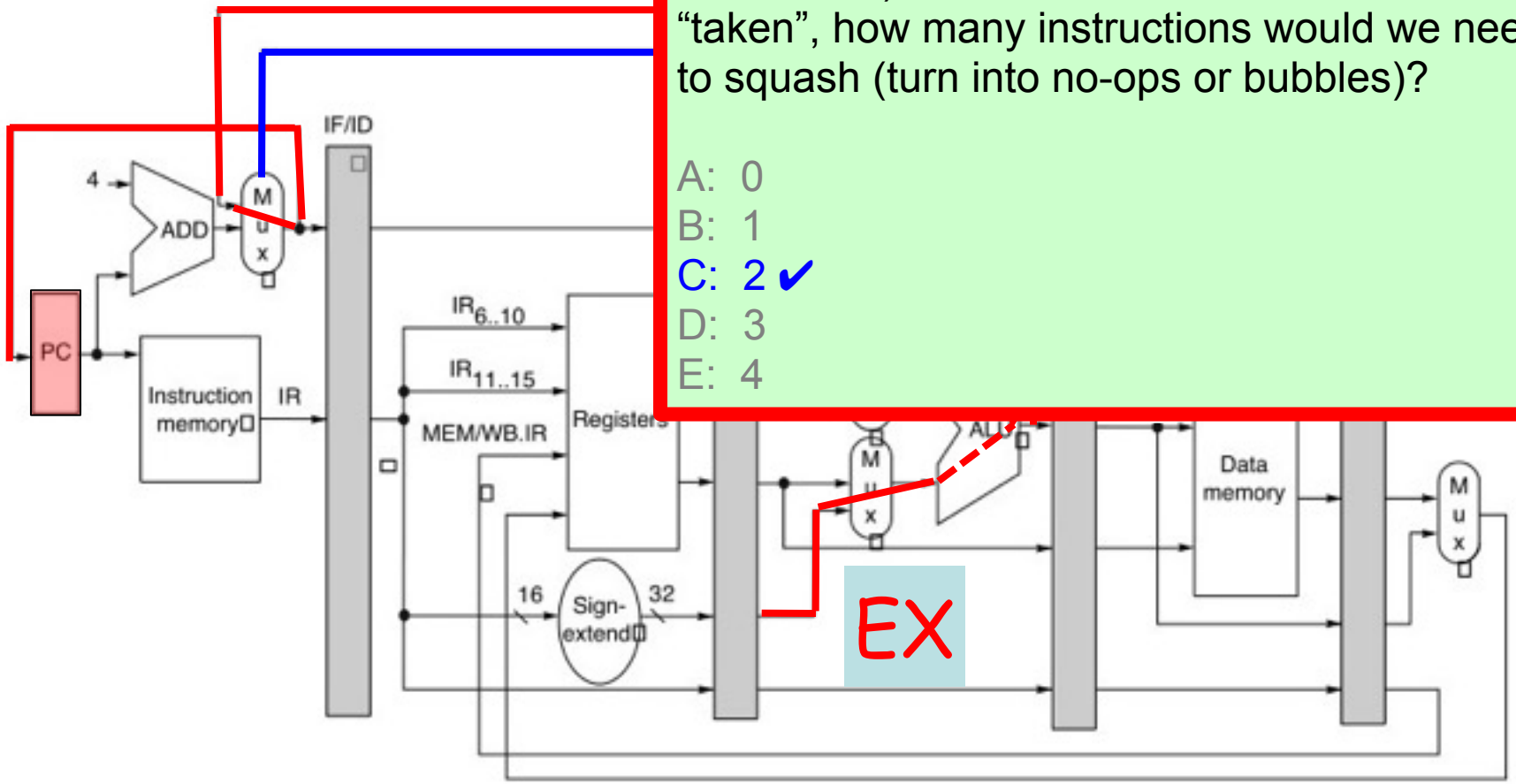


Check branch condition and compute target in execute stage.
Update PC with correct target by end of cycle that branch enters execute stage. Fetch correct target following cycle.

Resolving a Br

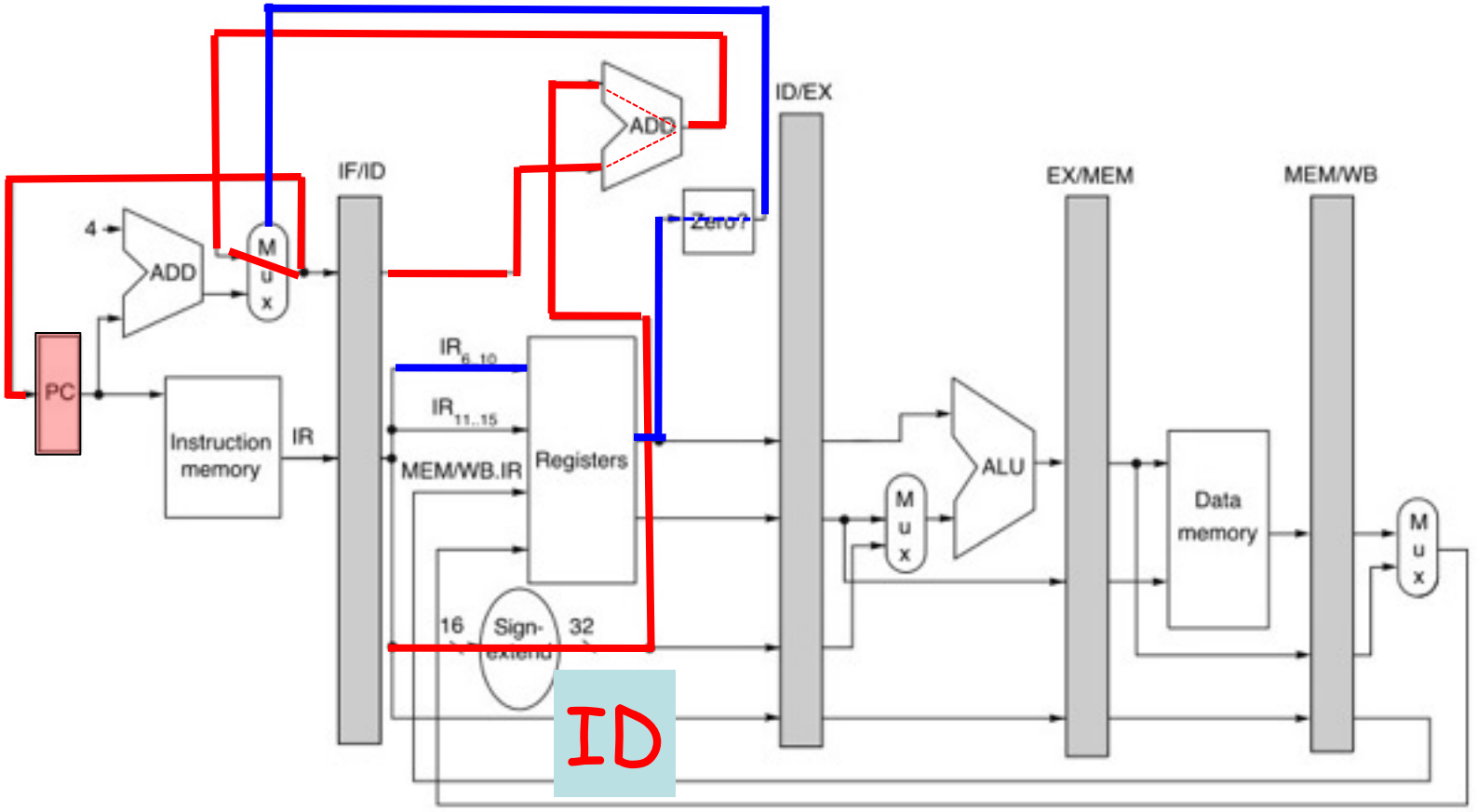
Assume branches are resolved in execute and we predict branches are “not taken” (as in Slide Set 4). If a branch turns out to be “taken”, how many instructions would we need to squash (turn into no-ops or bubbles)?

A: 0
B: 1
C: 2 ✔
D: 3
E: 4



Check branch condition and compute target in execute stage.
Update PC with correct target by end of cycle that branch enters execute stage. Fetch correct target following cycle.

Resolving a Branch in Decode (ID)

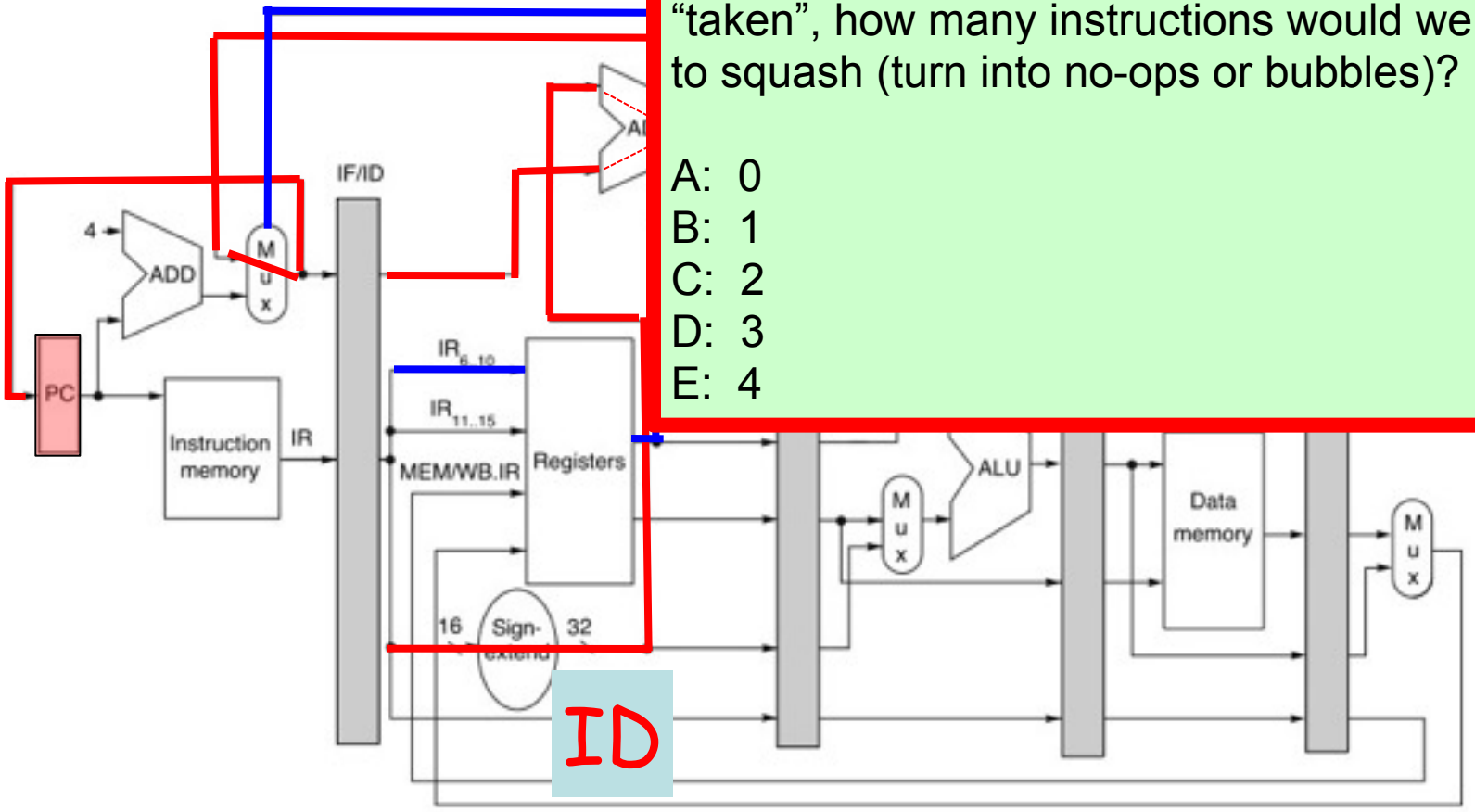


Check branch condition and compute target in decode stage. Update PC with correct target by end of cycle that branch enters decode stage. Fetch correct target following cycle.

Resolving a Br

Assume branches are resolved in decode and we predict branches are "not taken" (as in Slide Set 4). If a branch turns out to be "taken", how many instructions would we need to squash (turn into no-ops or bubbles)?

A: 0
B: 1
C: 2
D: 3
E: 4

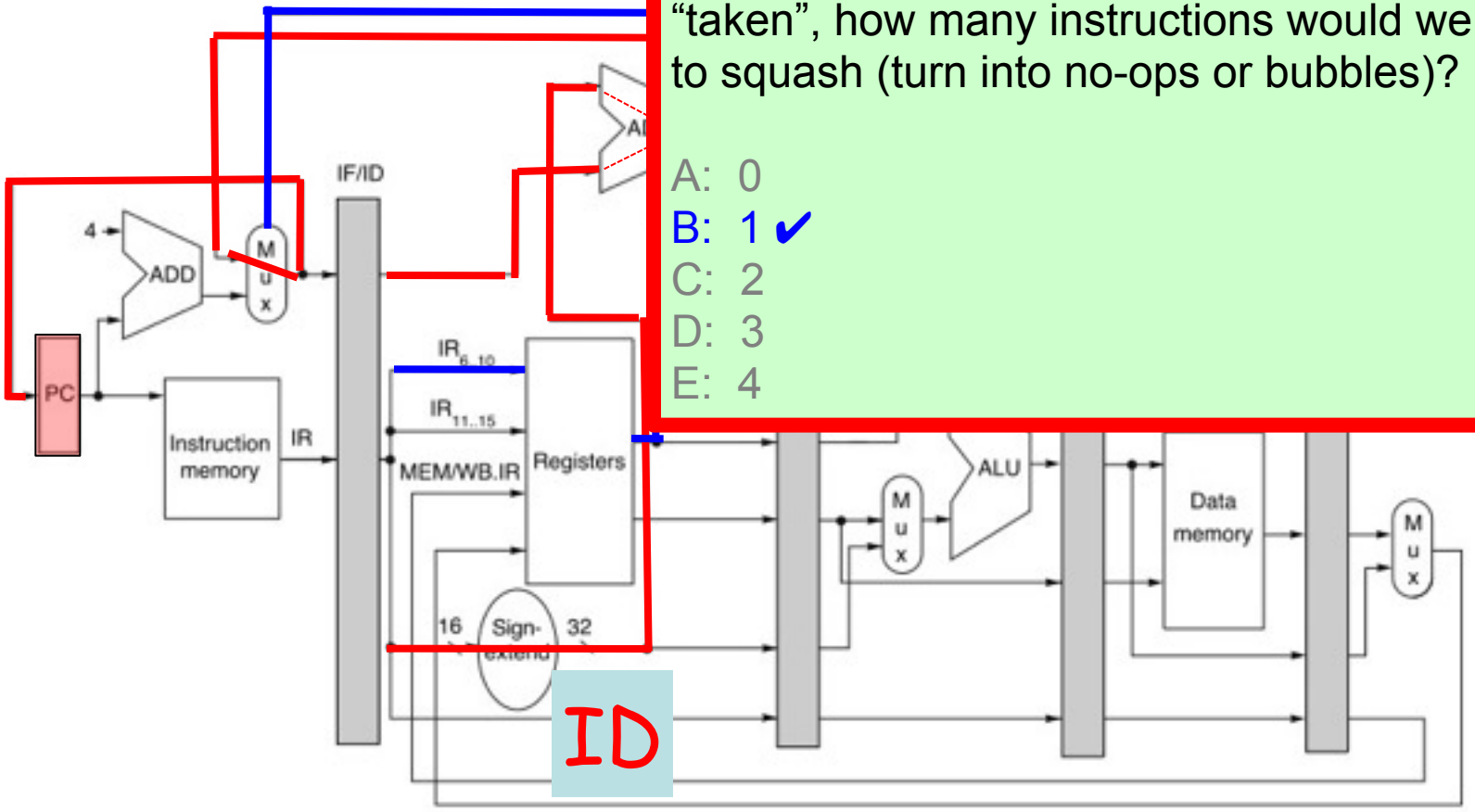


Check branch condition and compute target in decode stage. Update PC with correct target by end of cycle that branch enters decode stage. Fetch correct target following cycle.

Resolving a Br

Assume branches are resolved in decode and we predict branches are "not taken" (as in Slide Set 4). If a branch turns out to be "taken", how many instructions would we need to squash (turn into no-ops or bubbles)?

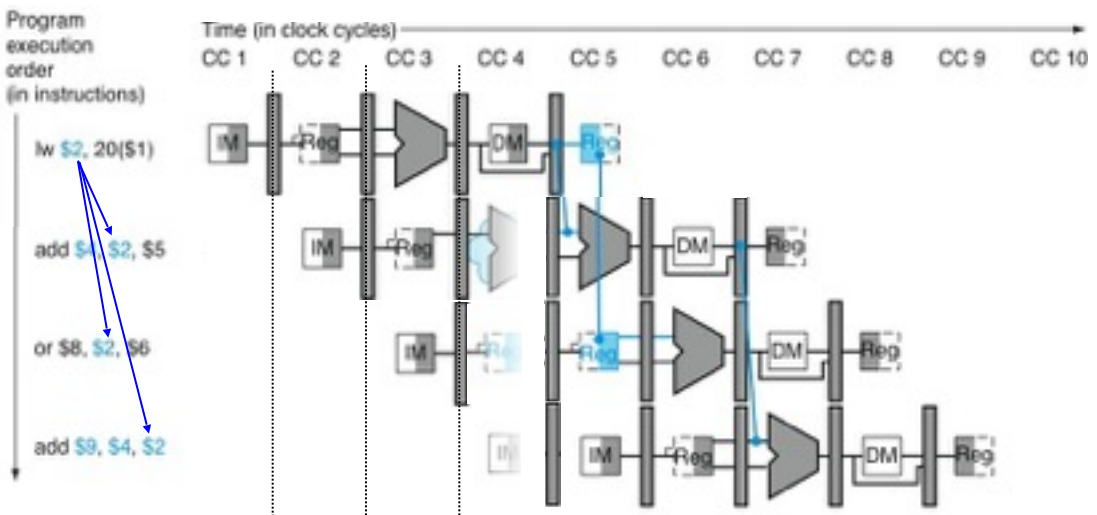
A: 0
B: 1 ✔
C: 2
D: 3
E: 4



Check branch condition and compute target in decode stage. Update PC with correct target by end of cycle that branch enters decode stage. Fetch correct target following cycle.

Pipeline Diagrams Formats

Abstract away detail



A

A=B=C

(all mean same thing)

	Clock Number									
	1	2	3	4	5	6	7	8	9	10
lw $2,20($1)	F	D	X	M	W					
add $4,$2,$5		F	D	X	X	M	W			
or $8,$2,$6			F	D	D	X	M	W		
add $9,$4,$2					F	D	X	M	W	

B

	Clock Number									
	1	2	3	4	5	6	7	8	9	10
lw $2,20($1)	F	D	X	M	W					
add $4,$2,$5		F	D	s	X	M	W			
or $8,$2,$6			F	s	D	X	M	W		
add $9,$4,$2					F	D	X	M	W	

C

Pipelining Tradeoffs

- The Simple Five Stage Pipeline (also called “Classic Five Stage Pipeline”, “Simple RISC Integer pipeline”, etc…) is only one possible way to pipeline a MIPS processor.
- One could divide up the pipeline over more stages to achieve higher clock frequencies.
- We can have no, some, or (more commonly) complete support for forwarding to reduce stalls… forwarding hardware increases area and can lower clock frequency. Whether the increase in performance is “enough” to merit the area cost depends upon how important performance is versus cost for a given application.
- Compilers can sometimes “schedule” instructions to eliminate or reduce data hazards.

Drawing Pipeline Timing Diagrams

- Timing diagrams may seem like magic or that rules seem “made up as we go along”, but this is not the case.
- A pipeline timing diagram is an abstract representation of real hardware.
- To analyze pipeline timing using a pipeline timing diagram, we need to keep in mind the hardware details even if we don’t draw pictures of those details in the timing diagram.

Drawing Pipeline Timing Diagrams

- Step 1: Identify data hazards, draw arrow from destination register of instruction producing value to source register of instruction consuming value. Note branch instructions that are "taken" (e.g., you might mark them with a star "*")
- Step 2: Consider each instruction "T" in the order it is executed by the program. Starting from fetch, determine on which clock cycle "T" reaches each pipeline stage using the following "rules":
  - Does "T" follow a taken branch? If so, consider when that branch is "resolved" – fetch for "T" (e.g., "IF" stage) occurs cycle after branch is "resolved".
  - Consider any structural hazards involving "T" and an earlier instruction in the pipeline. Stall instruction "T" on cycles for which there is a structural hazard.
  - Consider any "data hazards" identified in Step 1 and identify any instruction "S" that produces a value consumed by instruction "T". For each such instruction "S", consider which pipeline stage for "S" produces the value and which stage for "T" consumes the value.
  - Ensure the consuming stage for "T" occurs at least one cycle later than the producing stage for "S" inserting stalls for "T" if necessary – e.g., at the decode stage if hazard detection is in decode. Here you may need to consider whether forwarding is allowed (based upon the question you are trying to answer).
  - Draw arrow from stage forwarding to consumer stage for "T". This arrow should not cross more than one clock cycle.

Common Errors in Pipeline Diagrams

- Forgetting that store and branch instructions do NOT write a register.
- Forgetting that a taken branch has a control hazard (causing a pipeline bubble).
- Forgetting that loads and stores use the execute stage for effective address calculation.

Example 1

Question 2: [3 marks] Use the following code fragment:

```
Loop:   LD   R1,0(R1)
        SD   R1,8(R2)
        BNEQ R1,R3,Loop
```

Show the timing for this code assuming normal forwarding and bypassing hardware and the classic RISC five-stage integer pipeline. Assume branches are resolved in decode. Assume the first time the branch (BNEQ) is encountered it is "taken", and only show the timing of the first four instructions executed. Indicate where forwarding occurs.

Solution:

	1	2	3	4	5	6	7	8	9	10

Computing Cycles Per Instruction

To compute the average cycles per instruction of the simple pipelined processor

$$\text{CPI} = \text{CPI}_{\text{no-stalls}} + \sum_{j=1}^{n} \text{stall duration}_j \times F_j$$

where :

$\text{CPI}_{\text{no-stalls}} \equiv$ average CPI without any pipeline stalls

$\text{stall duration}_j \equiv$ duration of stall type " j"

$$F_j \equiv \frac{\text{number of executed instructions experiencing stall type " j"}}{\text{Instruction Count}}$$

Example 2

- Compute CPI for 5 stage pipelined processor that contains support for forwarding to minimize stalls due to data hazards:
  - 20% of executed instructions are loads of which 50% are followed immediately by a dependant register-register ALU instruction (R-type instruction).
  - 10% of executed instructions are conditional branches (no delay slot), of which 50% are taken (assume predict not-taken, branches resolved at execute stage)
  - 15% unconditional jumps (no delay slot, update PC at decode)
  - Ignore source register dependencies for branches/jumps.

Solution:

to

eliminate hazard.

Above: 1 stall cycle for load followed by ALU
2 stall cycles for taken conditional branches
1 stall cycle for jumps

Solution:

CPI is closest to:

A: 1.20
B: 1.30
C: 1.35
D: 1.55
E: Not sure

to

eliminate hazard.

Above: 1 stall cycle for load followed by ALU
2 stall cycles for taken conditional branches
1 stall cycle for jumps

Solution:

CPI is closest to:

A: 1.20
B: 1.30
C: 1.35 ✔
D: 1.55
E: Not sure

to

eliminate hazard.

Above: 1 stall cycle for load followed by ALU
2 stall cycles for taken conditional branches
1 stall cycle for jumps

Solution:

CPI is closest to:

A: 1.20
B: 1.30
C: 1.35 ✔
D: 1.55
E: Not sure

```
CPI = 1                    /* base CPI */
    + 1*(0.2*0.5)   /* 20% loads, 50% followed by ALU op */
    + 2*(0.1*0.5)   /* 10% cond branches, 50% taken */
    + 1*(0.15)      /* 15% jumps, 1 cycle penalty */
    = 1.35
```

How to solve: Consider frequency of hazard condition and amount of stall cycles required to eliminate hazard.

Above: 1 stall cycle for load followed by ALU
2 stall cycles for taken conditional branches
1 stall cycle for jumps

Solution:

CPI is closest to:

A: 1.20
B: 1.30
C: 1.35 ✔
D: 1.55
E: Not sure

```
CPI = 1                    /* base CPI */
    + 1*(0.2*0.5)   /* 20% loads, 50% followed by ALU op */
    + 2*(0.1*0.5)   /* 10% cond branches, 50% taken */
    + 1*(0.15)      /* 15% jumps, 1 cycle penalty */
    = 1.35
```

How to solve: Consider frequency of hazard condition and amount of stall cycles required to eliminate hazard.

Above: 1 stall cycle for load followed by ALU
2 stall cycles for taken conditional branches
1 stall cycle for jumps