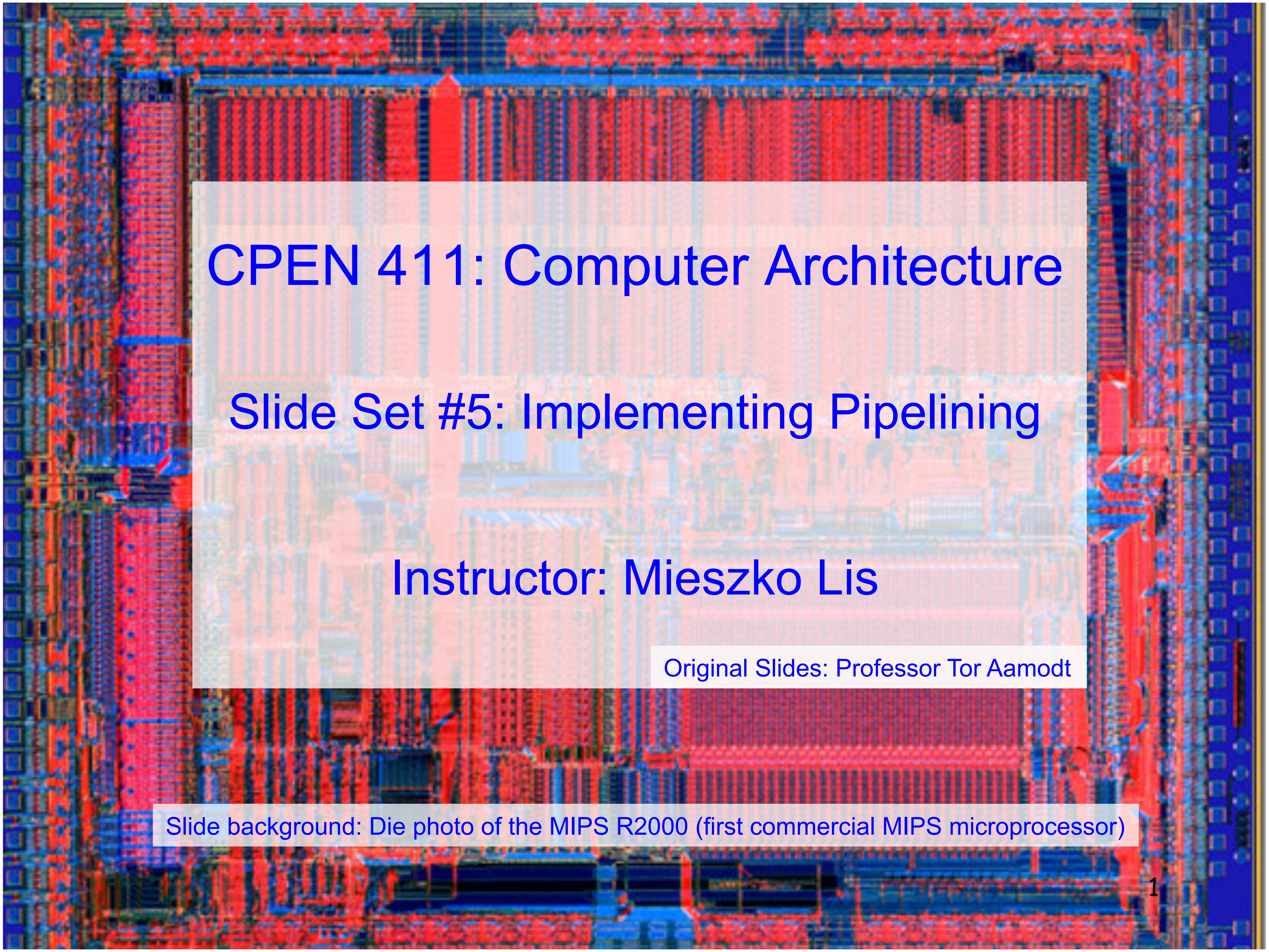


CPEN 411: Computer Architecture

Slide Set #5: Implementing Pipelining

Instructor: Mieszko Lis

Original Slides: Professor Tor Aamodt

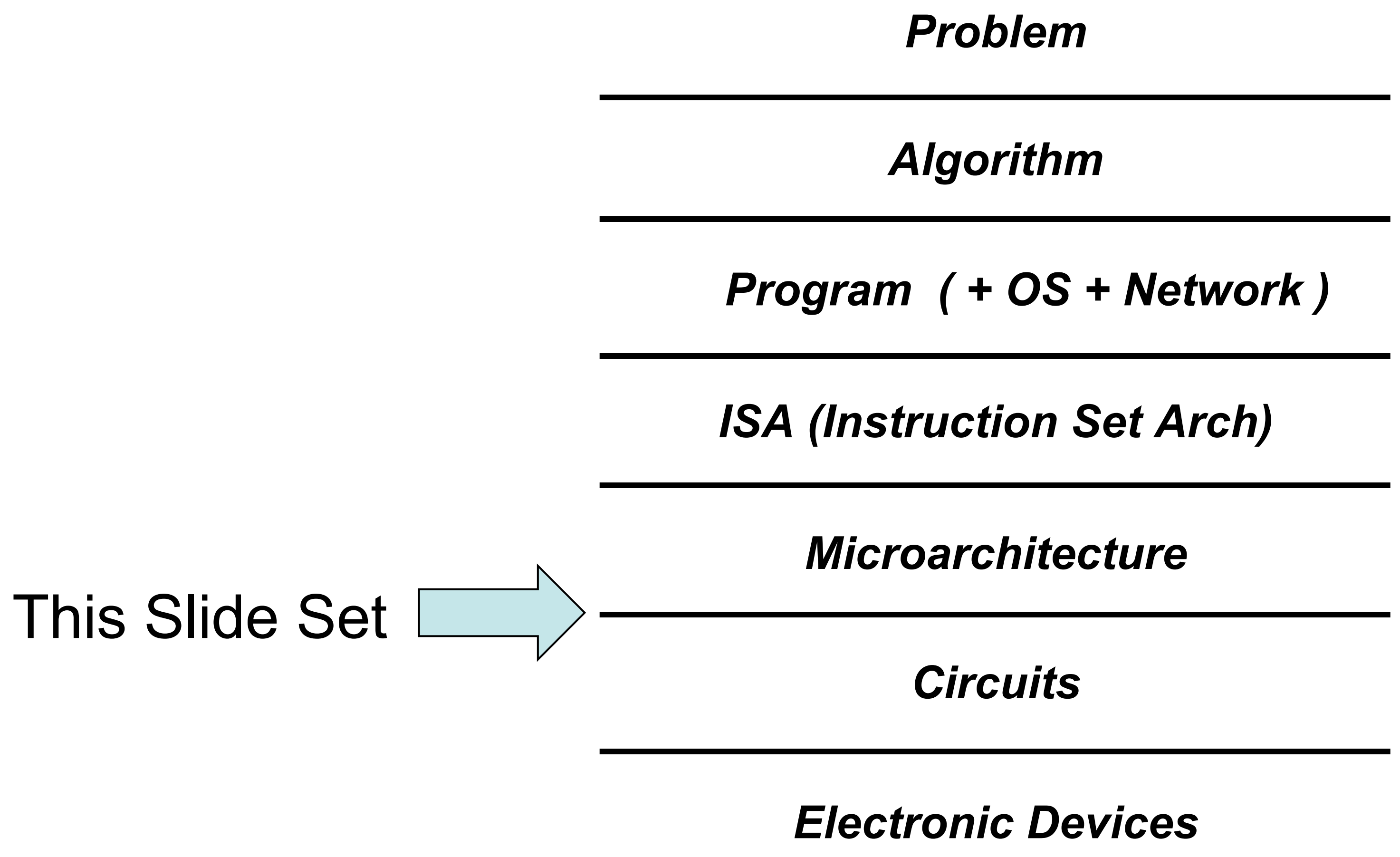
Slide background: Die photo of the MIPS R2000 (first commercial MIPS microprocessor)

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

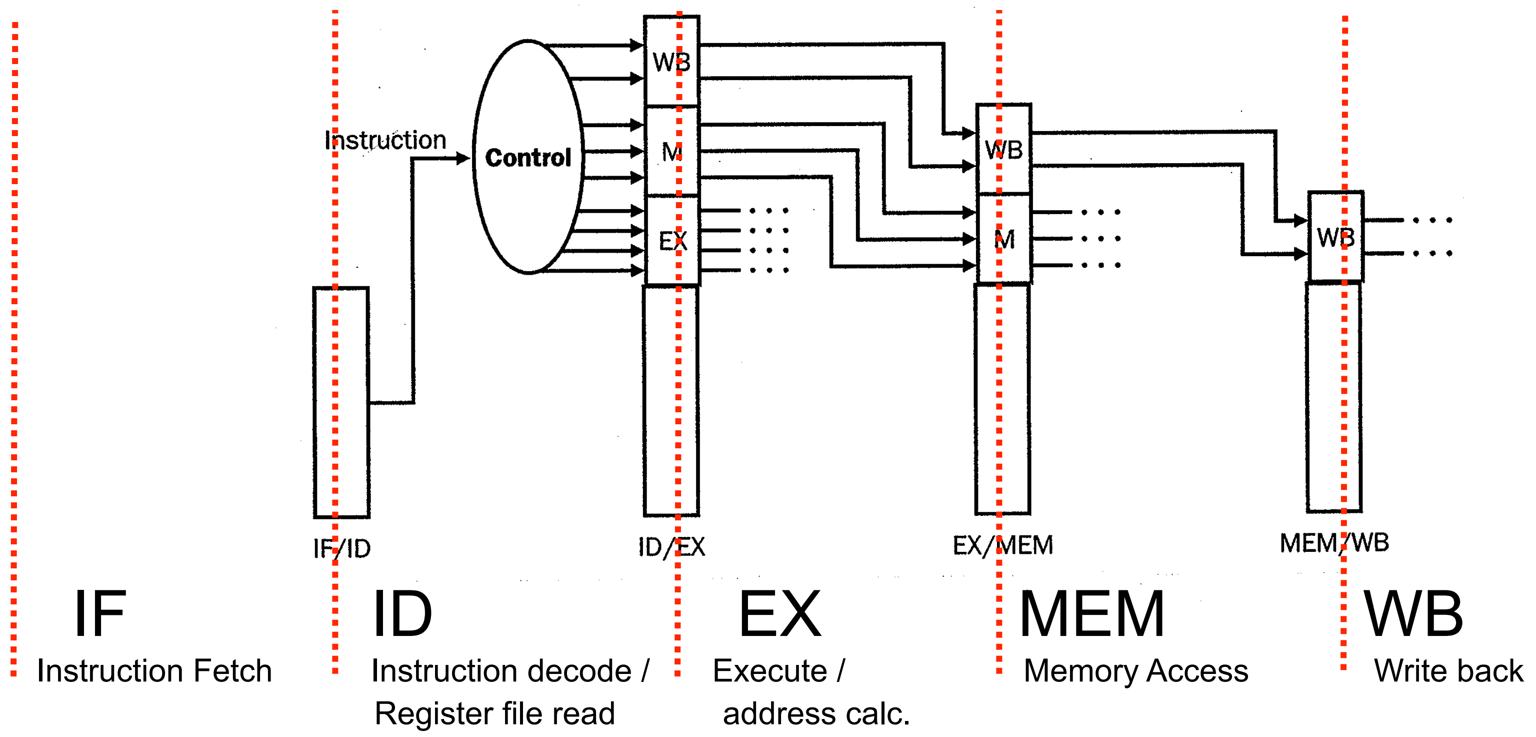
- After we finish this slide set you should be able to:
  - Explain the motivation for pipelined control, and how pipelined control is implemented.
  - Describe how forwarding is implemented using muxes and a forwarding control unit and explain how these operate.
  - Describe how stalls are implemented 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

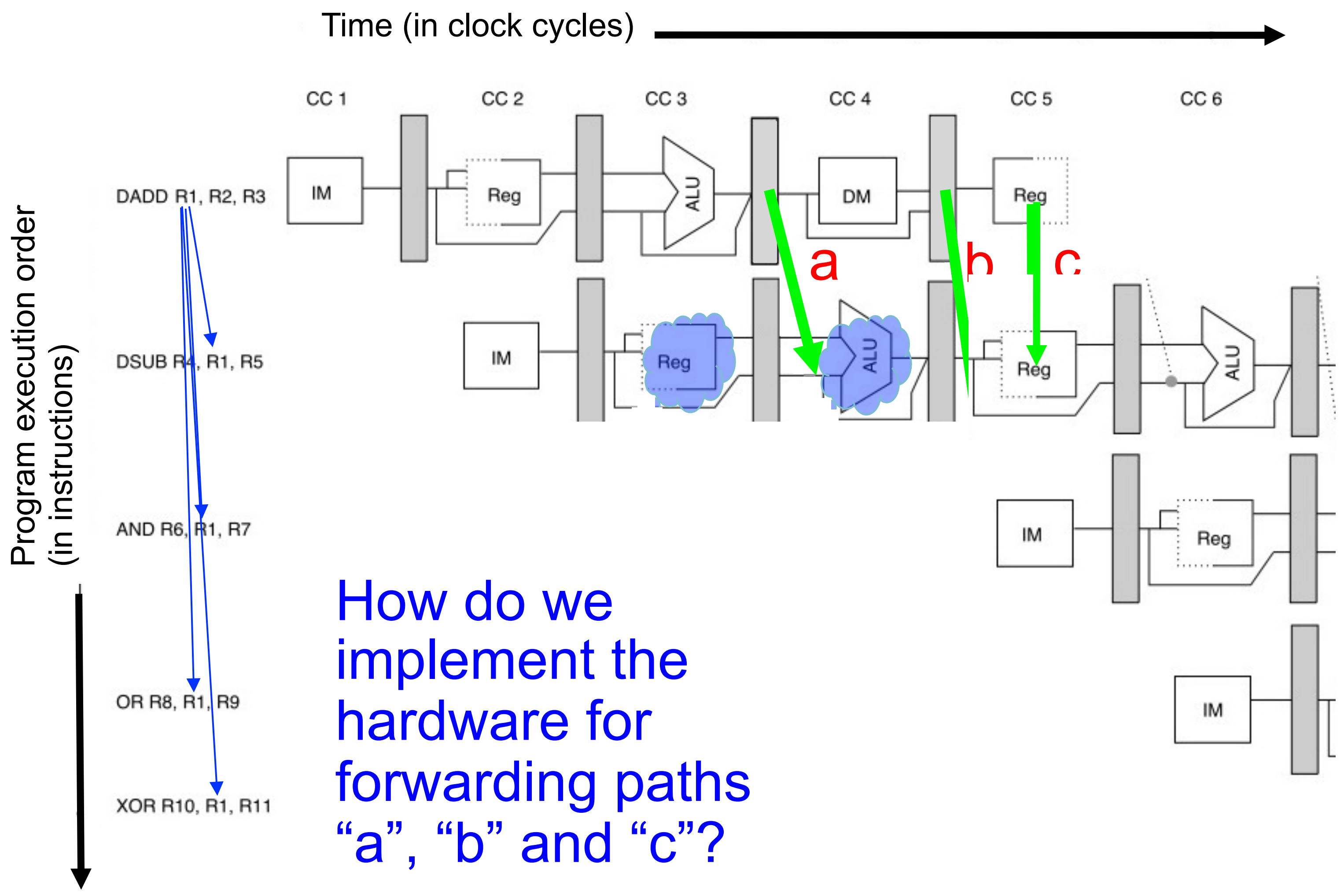
- Challenge: Control signals appear in each pipeline stage
- Review: Control strategies for non-pipelined processor:
  - Combinational Control (single-cycle CPU)
    - Cannot apply to pipelined processor since there are many instructions in pipeline
  - Sequential Control (FSM for multi-cycle CPU)
 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



Solution #2: Forwarding

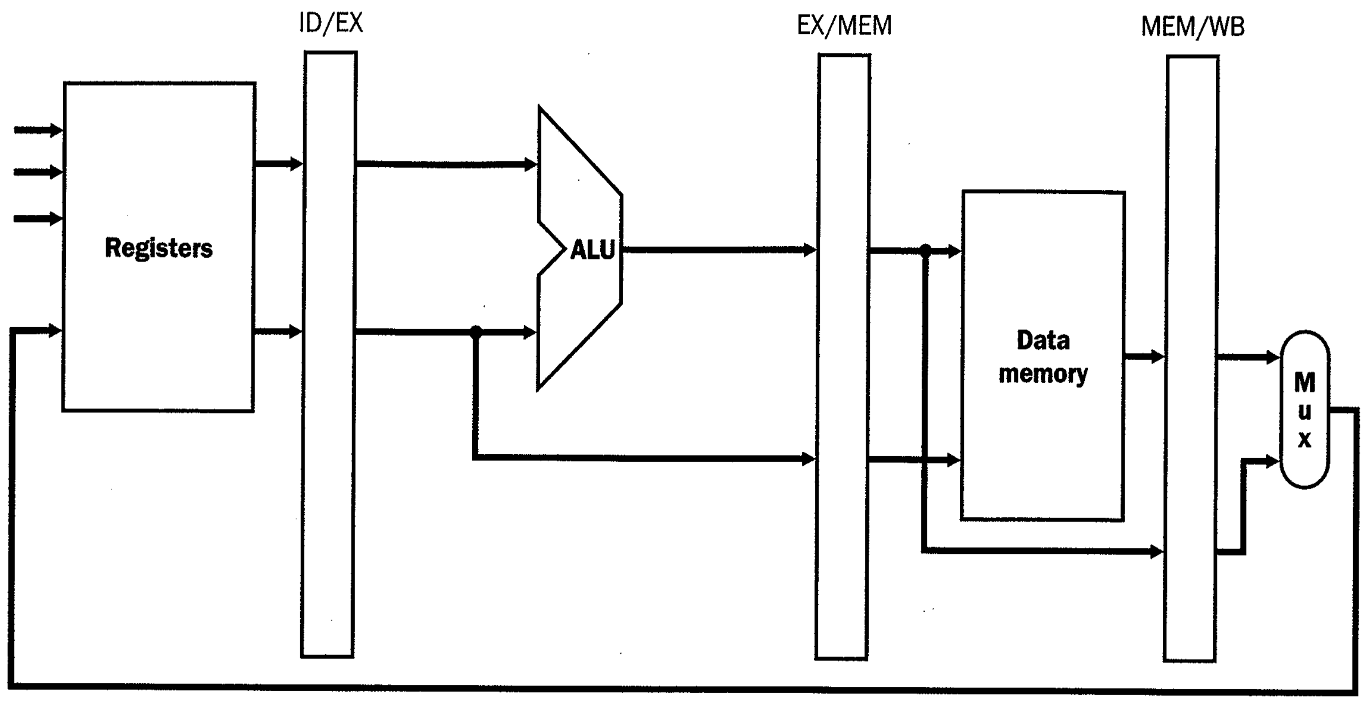


How do we implement the hardware for forwarding paths "a", "b" and "c"?

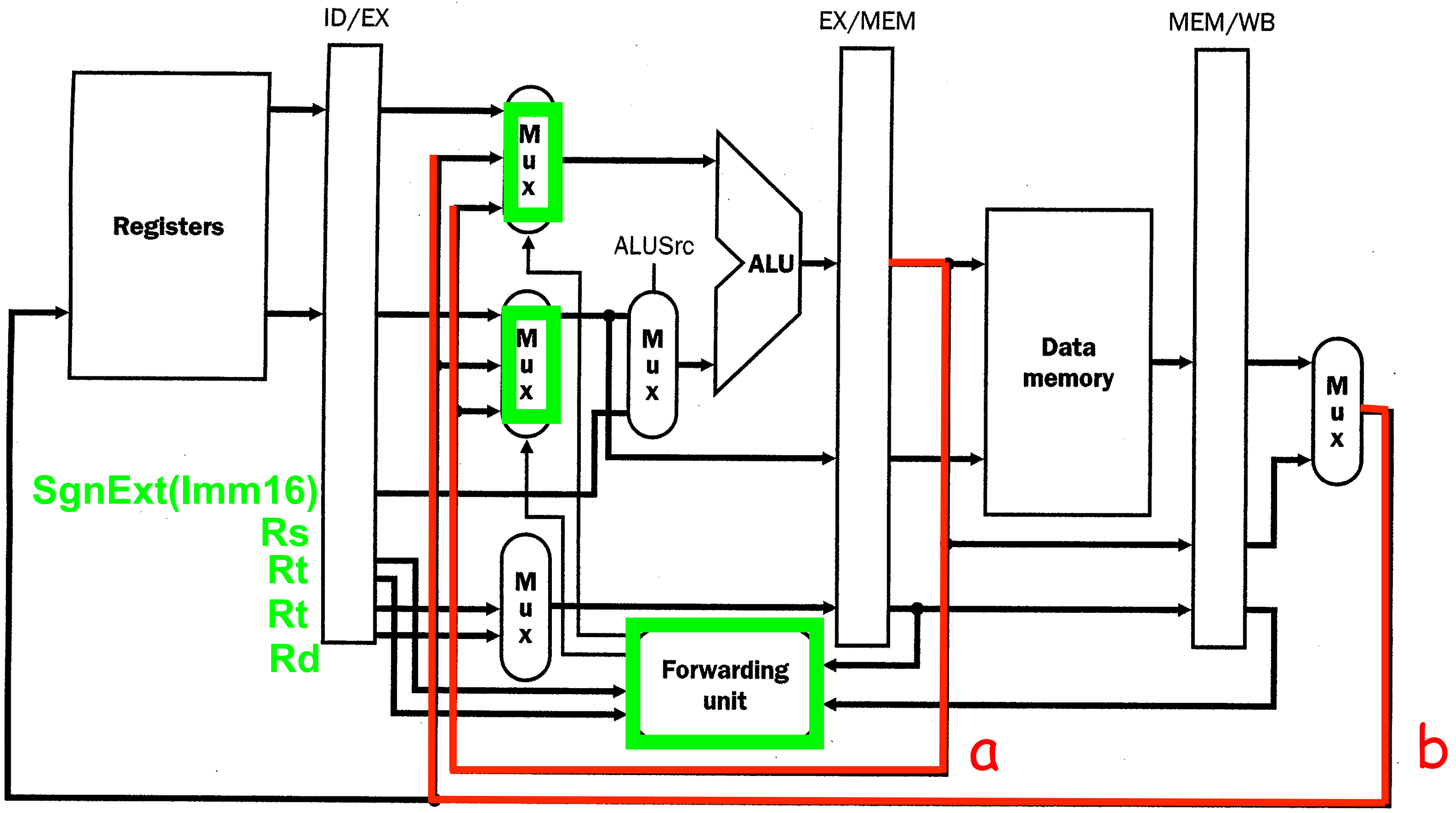
NOTE: Forwarding <u>begins</u> and <u>ends</u> within a single clock cycle

Implementing Forwarding

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



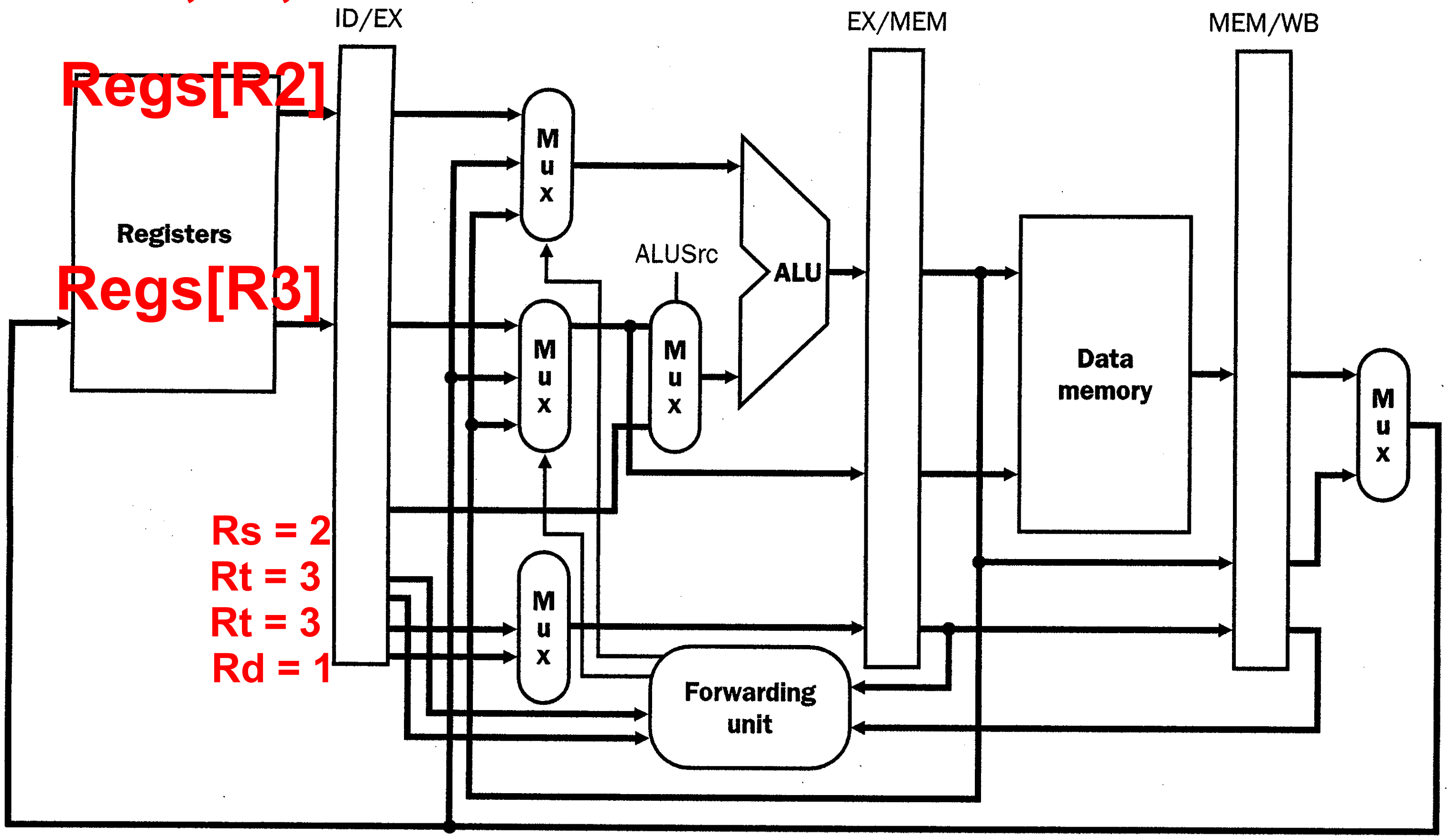
Implementing Forwarding



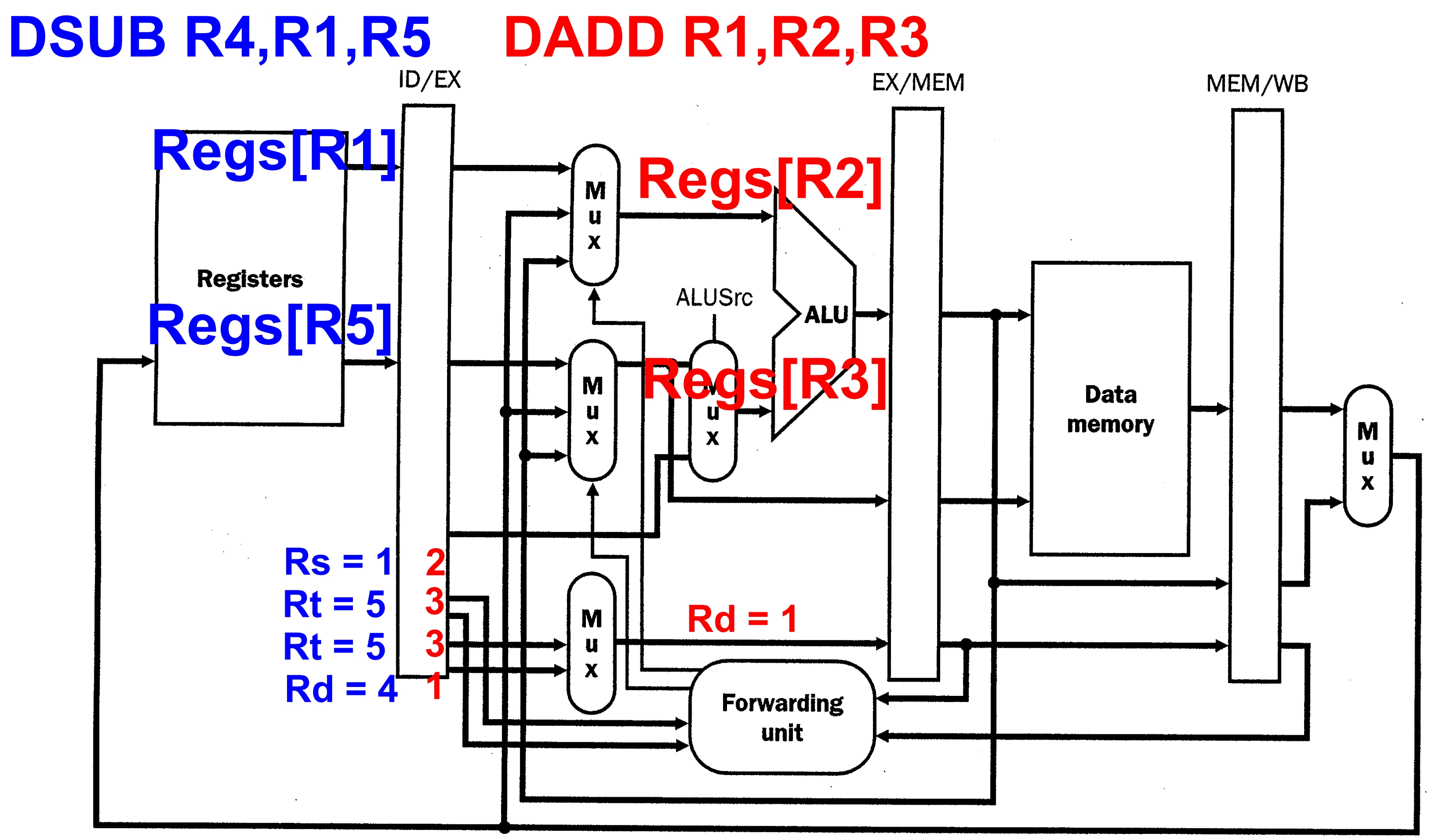
Forwarding paths often begin at output of pipeline register (values forwarded at beginning of clock cycle)

Forwarding Example

DADD R1,R2,R3

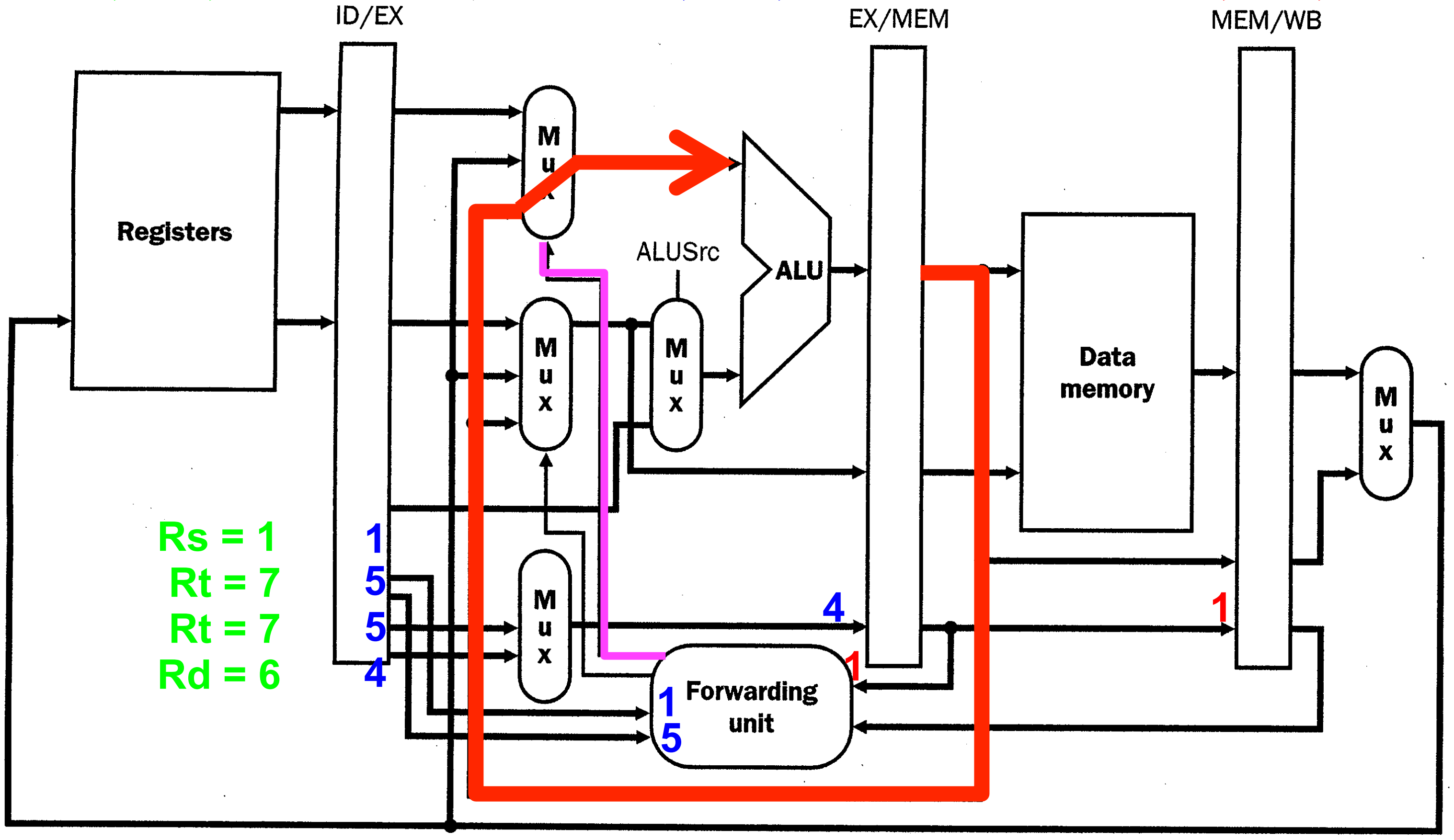


Forwarding Example



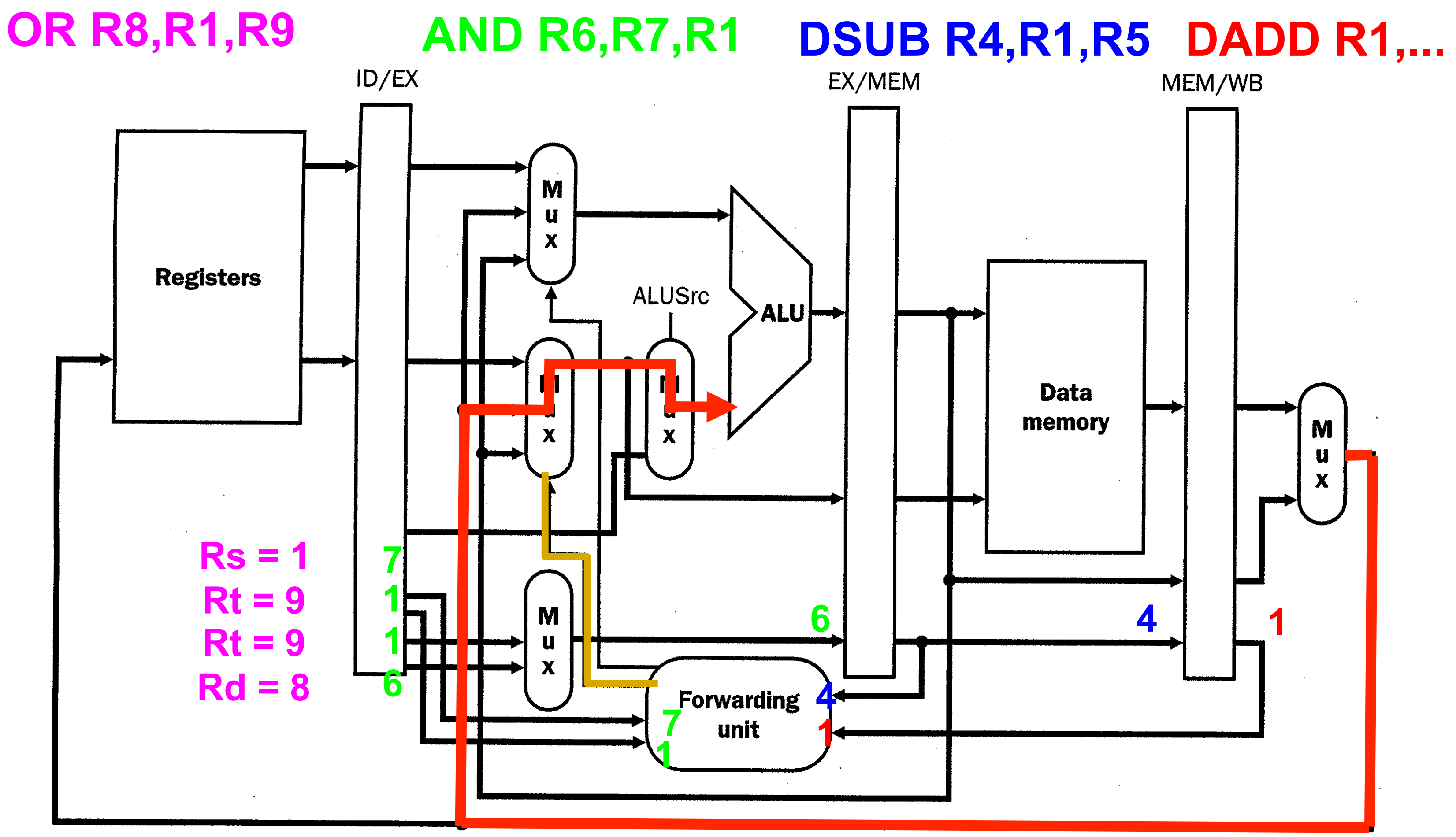
Forwarding Example

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

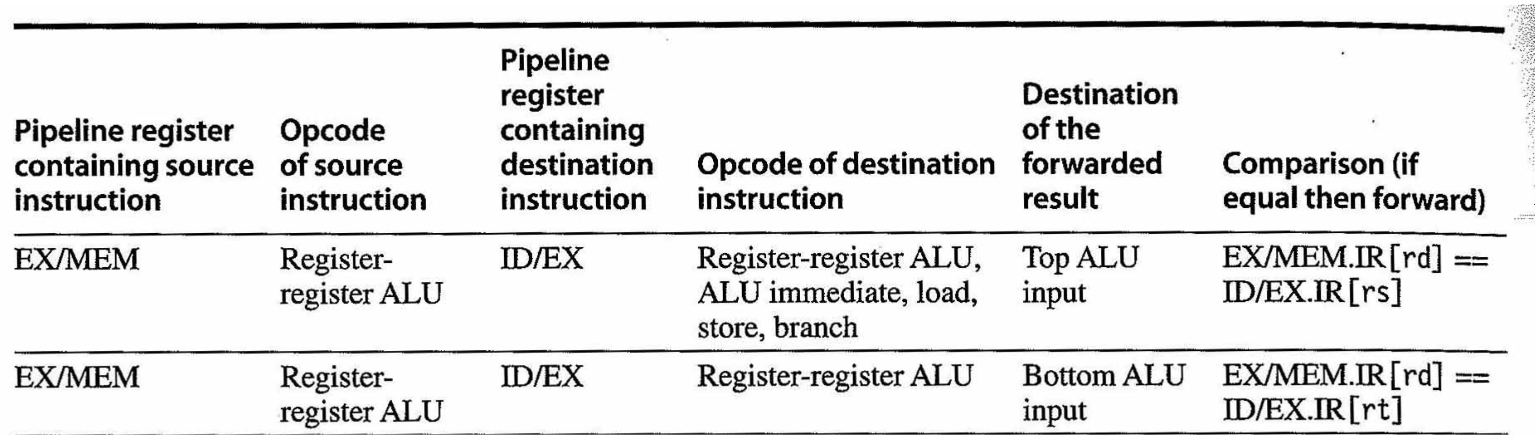


Forwarding Unit Logic

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

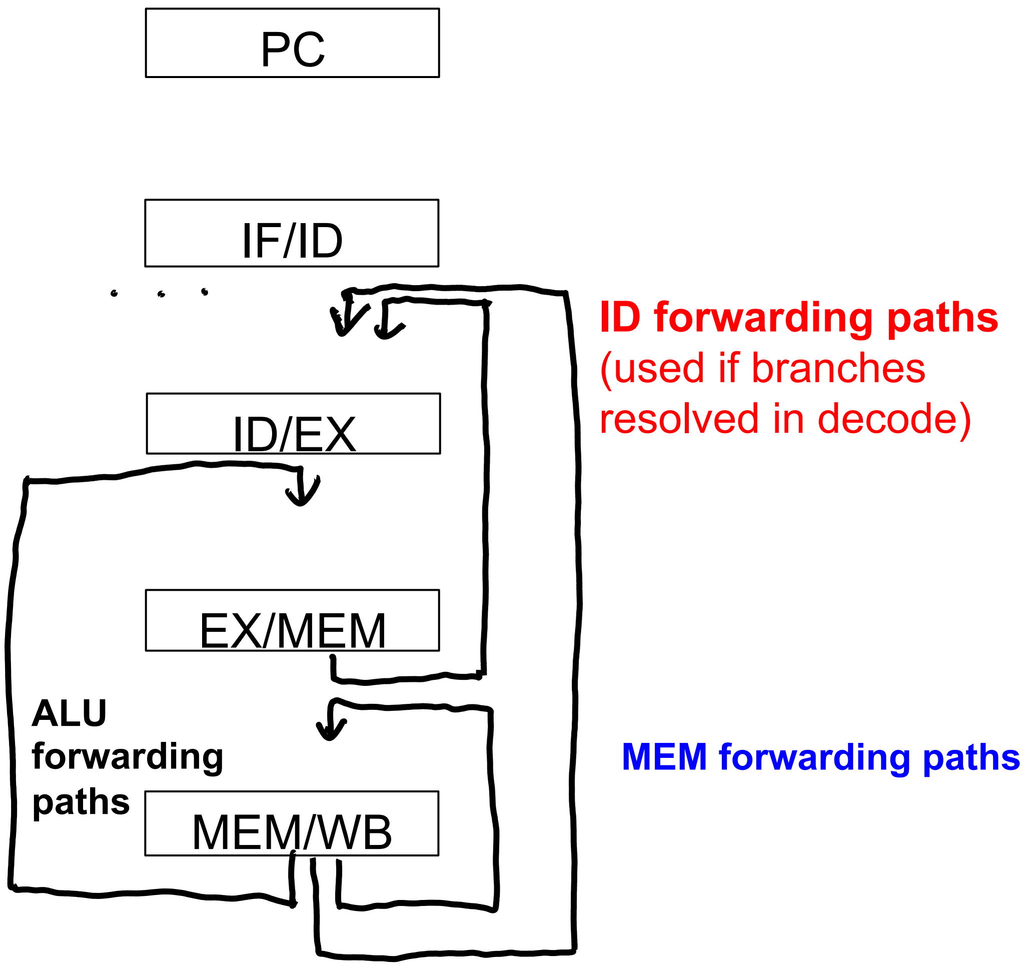
This unit needs to check for data hazards between instructions in different stages of the pipeline and enable the forwarding muxes.

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?

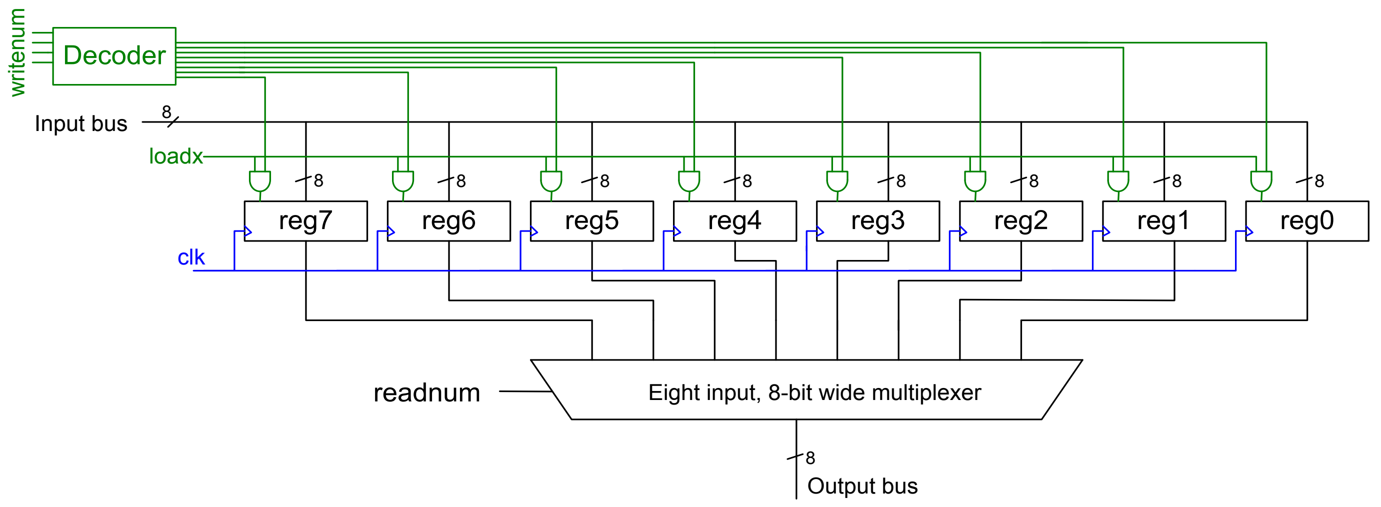


To fully reduce or eliminate need for stalling: Add forwarding paths starting from "producer" stages (and later stages) to earlier "consumer" stages.

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

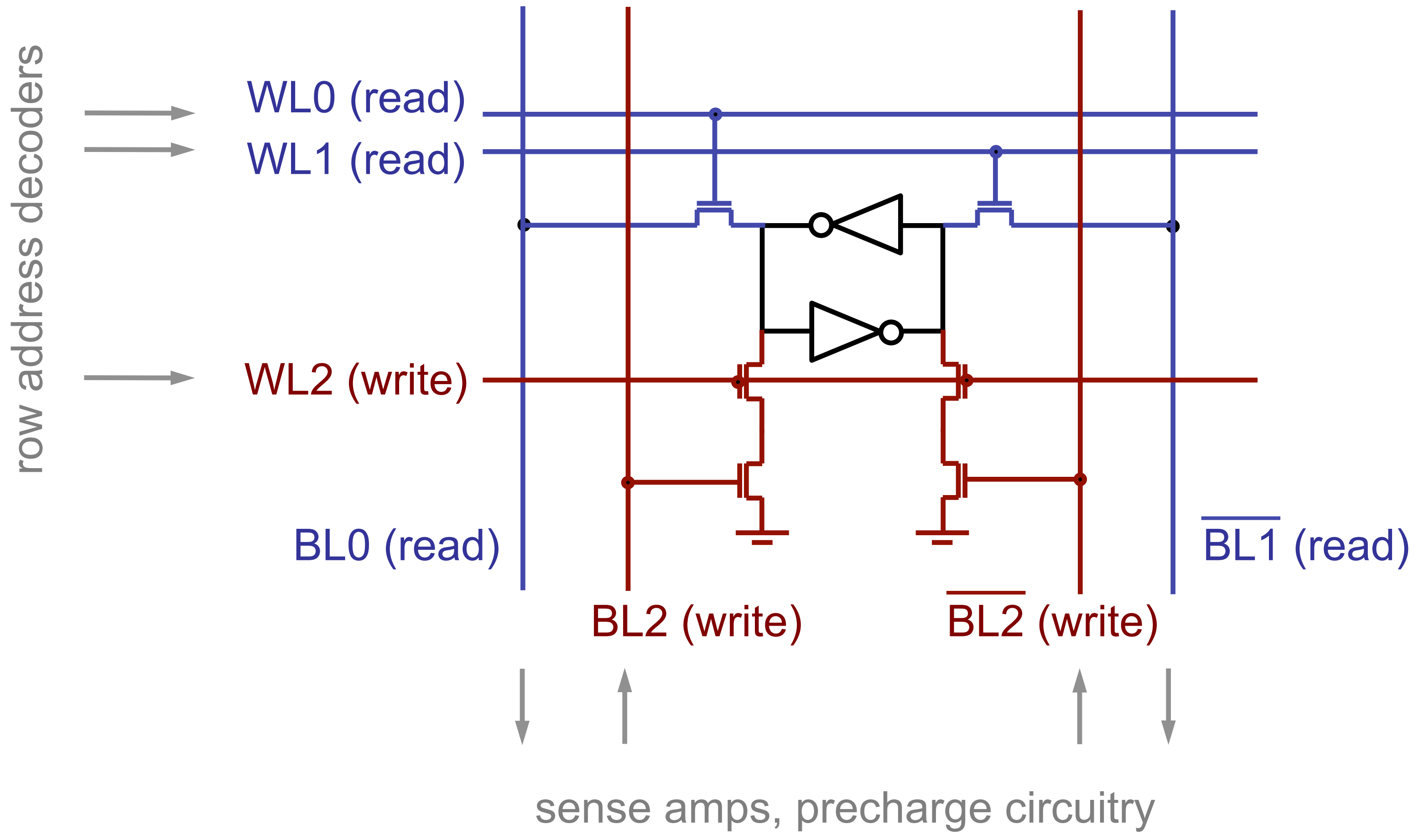
Forwarding through register file?

- Recall EECE 353 Lab 3:



- Add bypass mux for the forwarded value if reg conflict?
- Invert reg clocks to capture write value on falling edge?

A real register file cell


row address decoders
WL0 (read)
WL1 (read)
WL2 (write)
BL0 (read)
BL2 (write)
$\overline{\text{BL2}}$ (write)
$\overline{\text{BL1}}$ (read)
sense amps, precharge circuitry

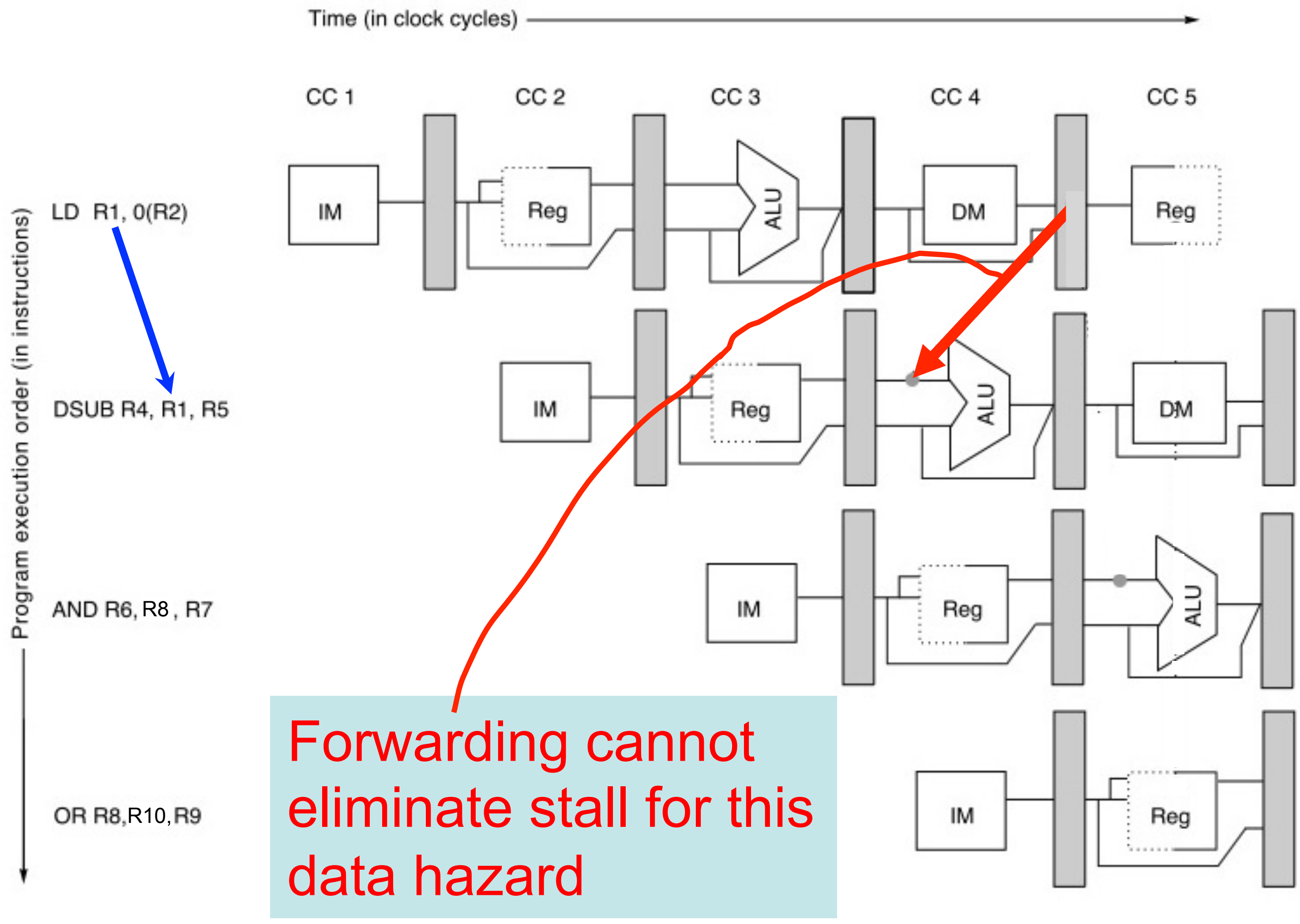
Pipeline Timing Diagram for Forwarding

	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

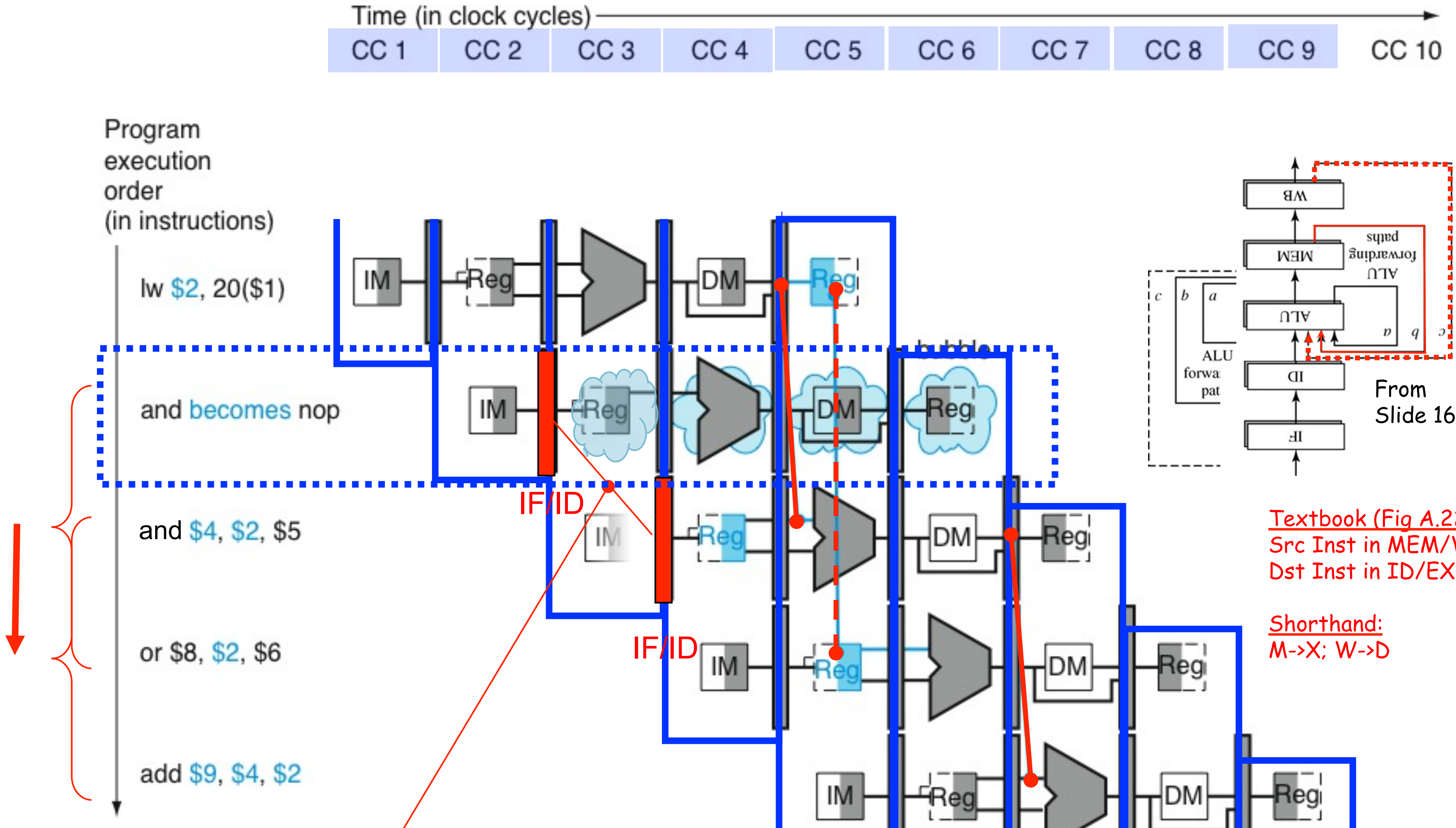


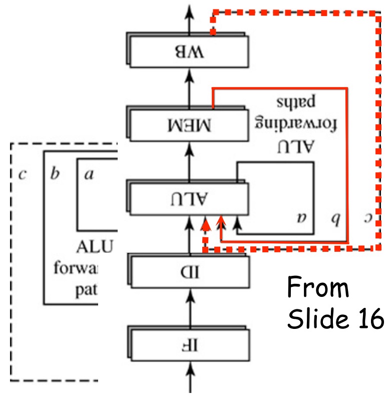
Forwarding cannot eliminate stall for this data hazard

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



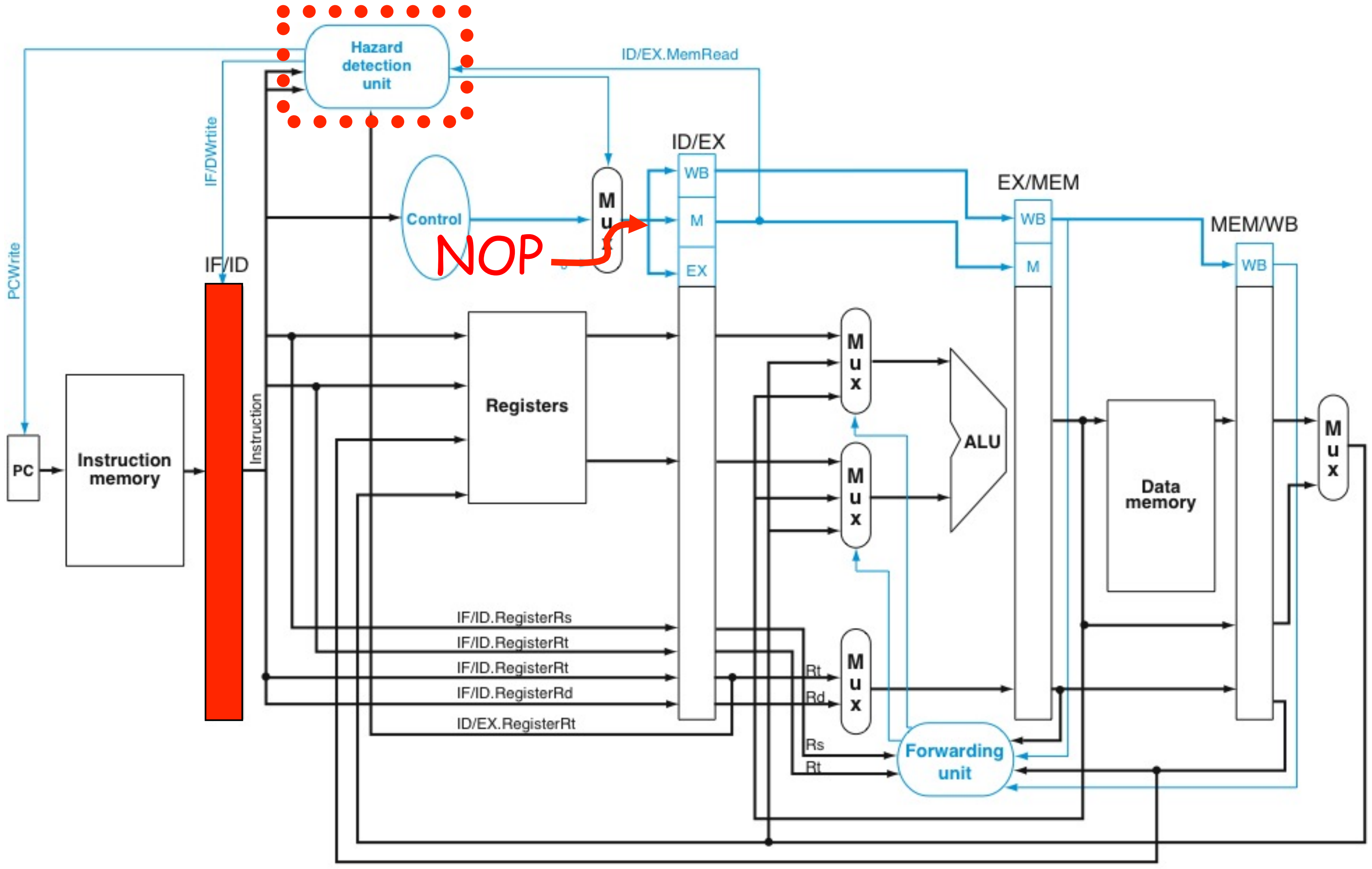


Textbook (Fig A.22):
Src Inst in MEM/WB
Dst Inst in ID/EX

Shorthand:
M->X; W->D

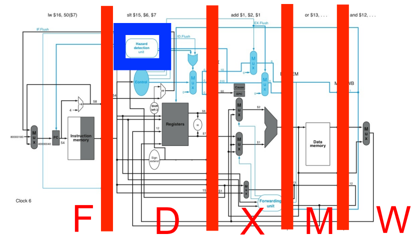
"and" stalls in IF/ID during CC3

Stall Hardware: Hazard Detection Unit

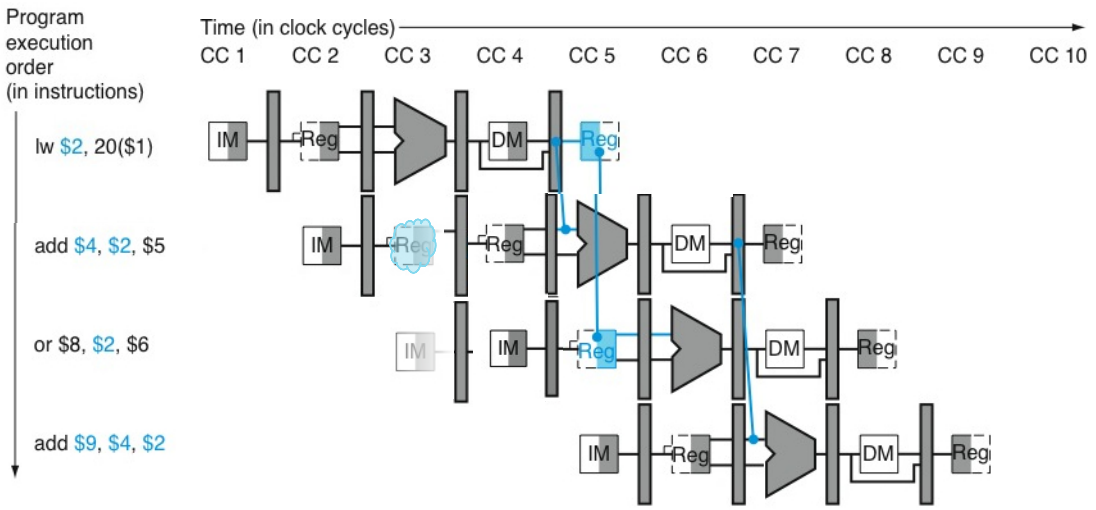


Hazard Detection Location Options

Hazard detection in **decode**:



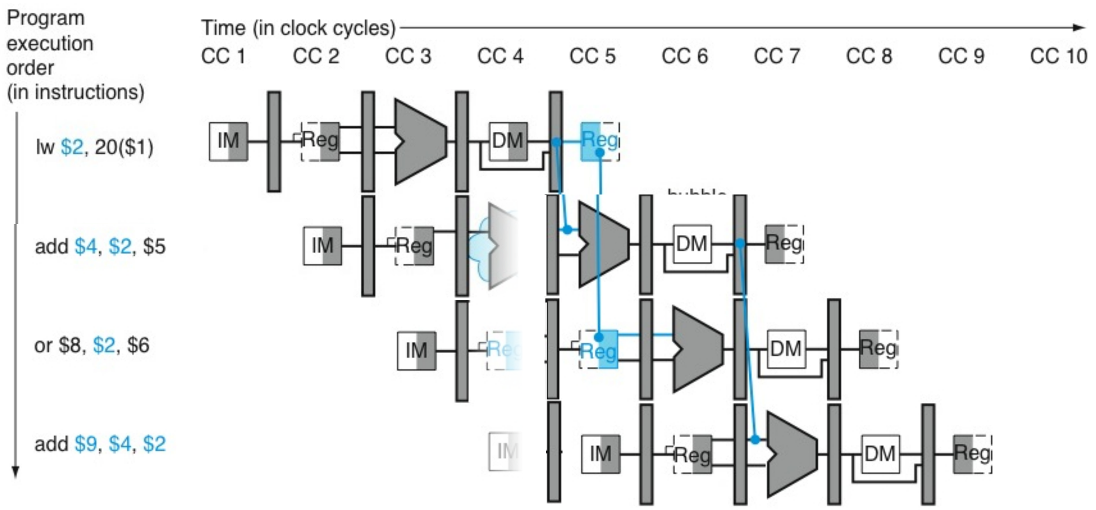
"and" instruction stalled in IF/ID during CC3



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

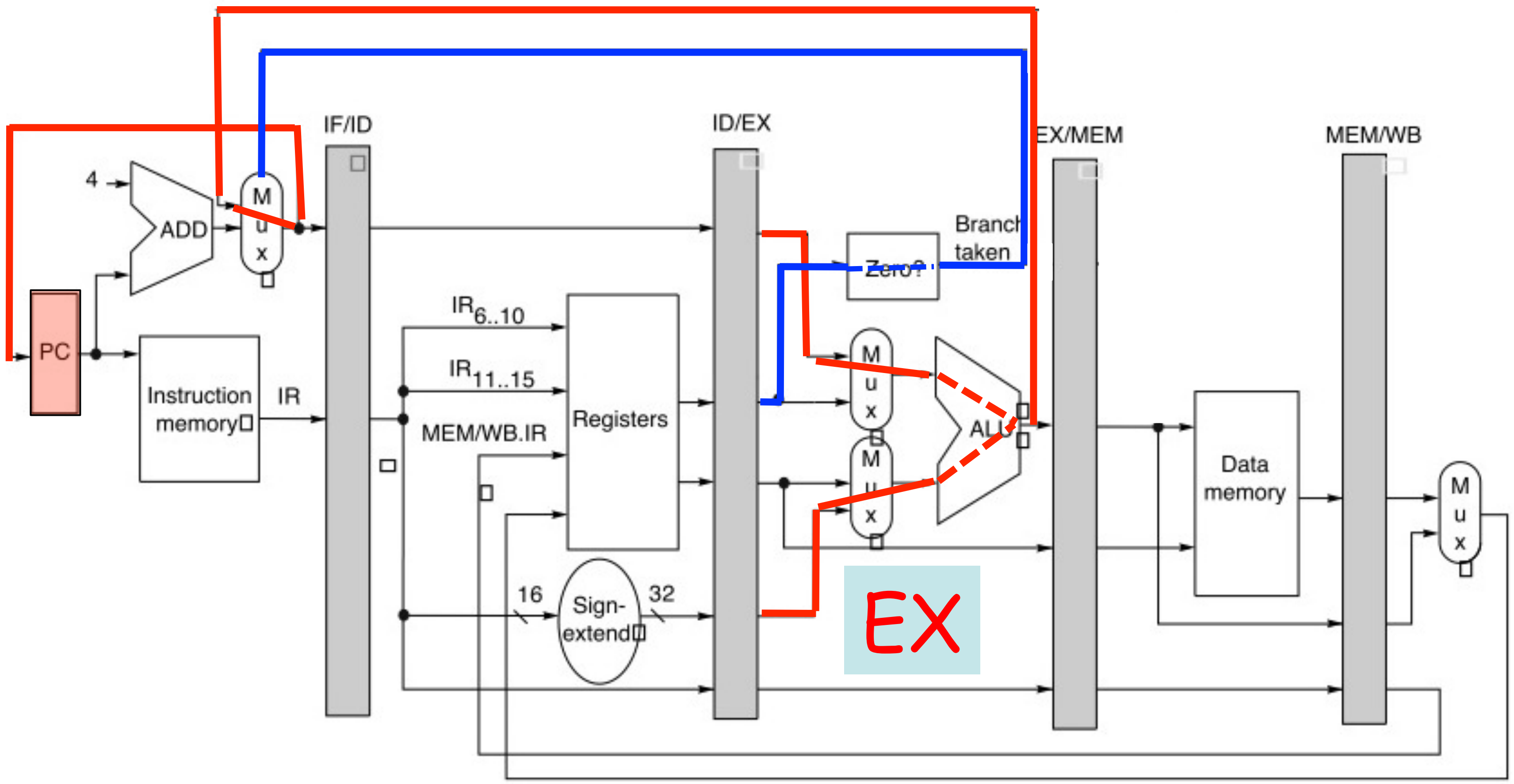
	Clock Number								
	1	2	3	4	5	6	7	8	9
LD R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB R4,R1,R5		IF	ID	EX	MEM	WB			
AND R6,R8,R7			IF	ID	EX	MEM	WB		
OR R8,R10,R9				IF	ID	EX	MEM	WB	

	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									

How to Draw Pipeline Diagrams

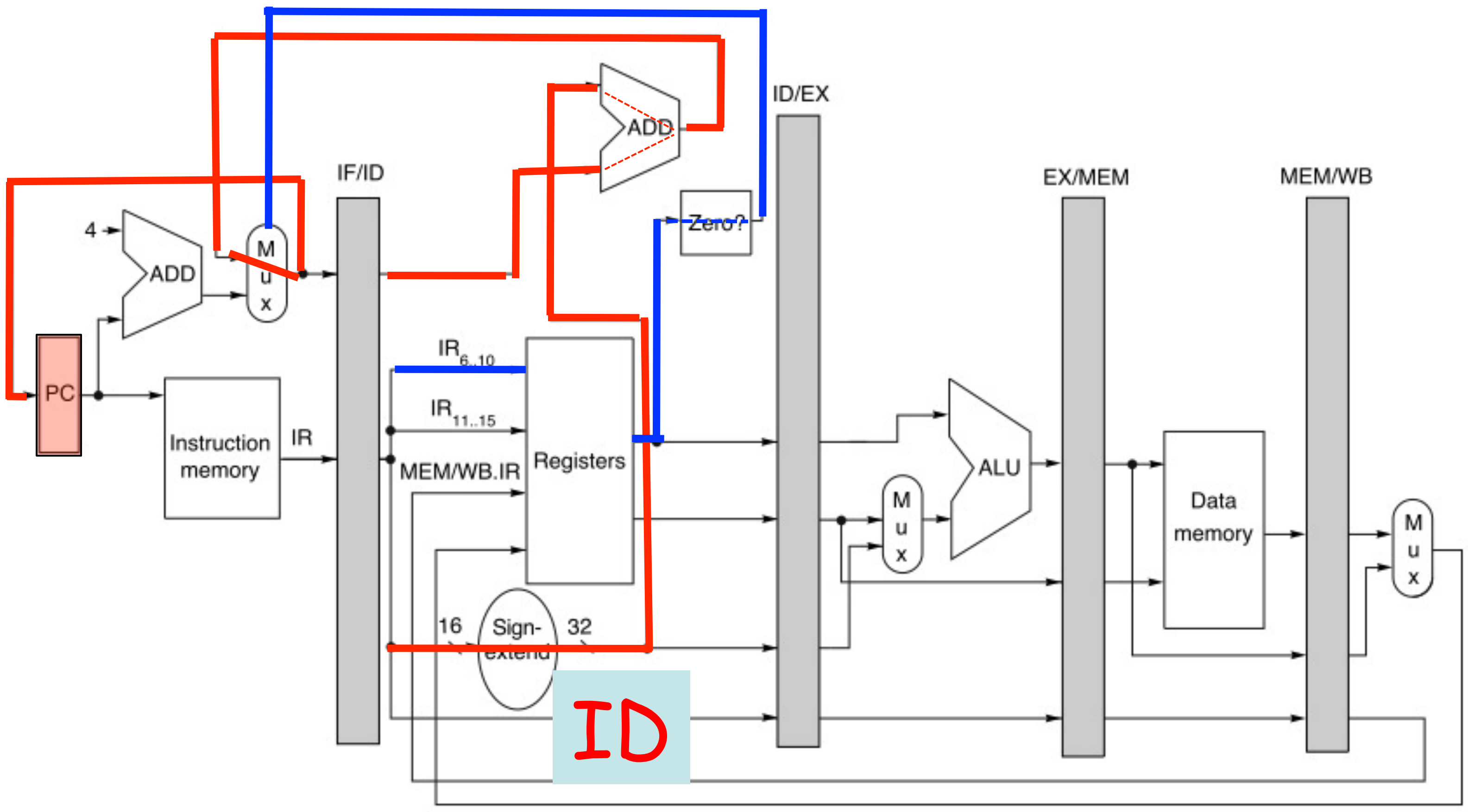
- Step 1: Identify data hazards, draw arrow from destination register of instruction producing value to source register of instruction consuming value. Note branch instructions that are "taken" (e.g., you might mark them with a star "*")
- Step 2: Consider each instruction "J" in the order it is executed by the program (here we use "J" to mean any instruction—it does not mean a jump instruction). Starting from fetch, determine on which clock cycle that instruction reaches each pipeline stage using the following "rules":
  - Does "J" follow a taken branch? If so, consider when that branch is "resolved" – fetch for "J" (e.g., "IF" stage) occurs cycle after branch is "resolved".
  - Consider any structural hazards involving "J" and an earlier instruction in the pipeline. Stall instruction "J" on cycles for which there is a structural hazard.
  - Consider any "data hazards" identified in Step 1 and identify any instruction "I" that produces a value consumed by instruction J". For each such instruction "I", consider which pipeline stage for I produces the value and which stage for J consumes the value.
  - Ensure the consuming stage for "J" occurs at least one cycle later than the producing stage for "I" inserting stalls for "J" 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 producer stage for "I" to consumer stage for "J".

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.

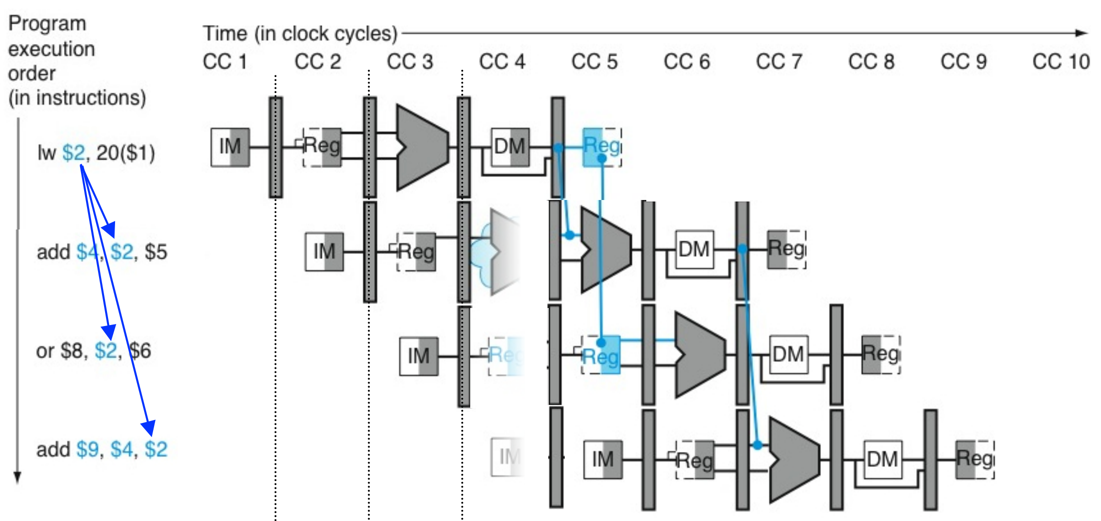
Resolving a Branch in Decode (ID)



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.

Pipeline Diagrams Formats

Abstract away detail



A

A=B=C

(all mean same thing)

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	X	X	M	W			
or $8,$2,$6			F	D	D	X	M	W		
add $9,$4,$2					F	D	X	M	W	

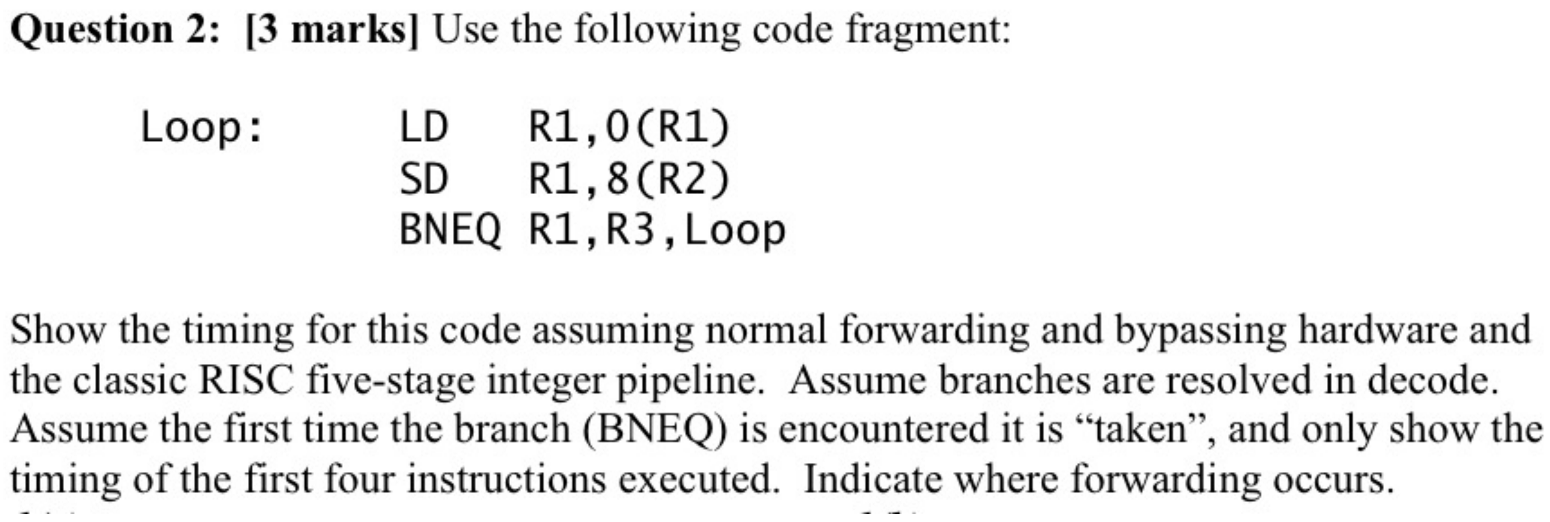
C

	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	

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.

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.

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 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:

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

Summary of Slide Set 5

In this slide set we learned the following:

- Pipelining Implementation Details
- Example pipeline timing diagram question
- Example CPI calculation
- Exceptions and pipelining