



CS471- Parallel Processing

Dr. Ahmed Hesham Mostafa

Lecture 3 – Processor Performance equations and Pipeline Hazards

The Processor Performance Equation

- Essentially all computers are constructed using a clock running at a constant rate.
- CPU time for a program can then be expressed two ways:
- $\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$

The Processor Performance Equation

- the average number of clock cycles per instruction (CPI)
- $$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$
- instruction count (IC)
- By transposing the instruction count in the above formula, clock cycles can be defined as $\text{IC} \times \text{CPI}$. This allows us to use CPI in the execution time formula: $\text{CPU clock cycles for a program} = \text{IC} \times \text{CPI}$
- $$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$
- $$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Clock cycle time}$$
- $$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{Clock cycle time}$$
- $$\text{Instruction time} = \text{CPI} \times \text{Clock cycle time}$$

The Processor Performance Equation

- Sometimes it is useful in designing the processor to calculate the number of total processor clock cycles as
- **CPU clock cycles** = $\sum_i^n IC_i \times CPI_i$
- where IC_i represents the number of times instruction i is executed in a program and CPI_i represents the average number of clocks per instruction for instruction i .
- This form can be used to express CPU time as
- Average CPU time = $(\sum_i^n IC_i \times CPI_i) \times \text{Clock cycle time}$

The Processor Performance Equation

- and overall CPI as
- Average CPI = $\frac{\sum_i^n IC_i \times CPI_i}{instruction\ count} = \sum_{i=1}^n \frac{IC_i}{instruction\ count} \times CPI_i$
- The latter form of the CPI calculation uses each individual CPI_i and the fraction of occurrences of that instruction in a program (i.e., $IC_i \div \text{Instruction count}$).

Avg instr time = clock cycle x avg CPI

$$= \text{clock cycle} \times \sum_{i=1}^n \frac{IC_i}{instruction\ count} \times CPI_i$$

Example

Consider an **unpipelined** instruction.

1 ns clock cycle;

4 cycles for **ALU** and **branches**;

5 cycles for **memory** operations;

relative frequencies **40%**, **20%**, **40%**;

0.2 ns **pipeline** overhead (e.g., due to stage imbalance, pipeline register setup, clock skew)

Question: How much speedup by pipeline?

Example

- Speedup from pipelining = $\frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$
- *Average instruction time unpipelined* =
= **clock cycle** $\times \sum_{i=1}^n \frac{IC_i}{\text{instruction count}} \times CPI_i$
= 1 ns $\times [(0.4 + 0.2) \times 4 + 0.4 \times 5]$
= 4.4 ns
- The ideal CPI on a pipelined processor is almost always 1
- *Average instruction time pipelined* = **CPI \times Clock cycle time + pipelined overhead**
- **= (1ns \times 1) + 0.2 = 1.2**

Example

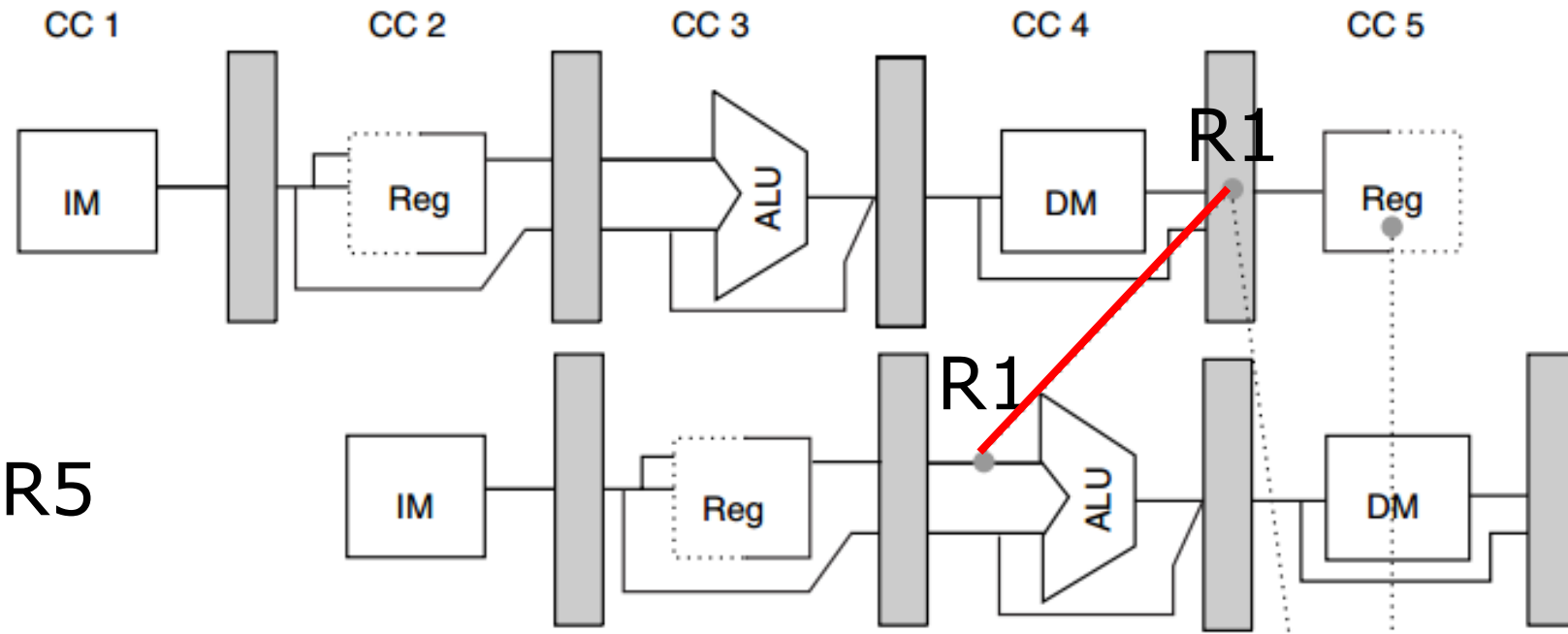
- **Answer**

speedup by pipelining $= 4.4 / 1.2 = 3.7$ times

When Pipeline Is Stuck

LD R1, 0(R2)

SUB R4, R1, R5



Actually, in many cases, instructions cannot be fully pipelined.

In this example, R1 is ready by load instruction at the end of clock cycle 4;

But subtract instruction requires R1 at the beginning of clock cycle 4; in this case, the processor can not fully pipeline the two instructions as we expect.

Pipeline Hazards

- **Hazards:** situations that prevent the next instruction from executing in the designated clock cycle.
- 3 classes of hazards:
 - structural hazard – resource conflicts
 - data hazard – data dependency
 - control hazard – pc changes (e.g., branches)

Pipeline Hazards

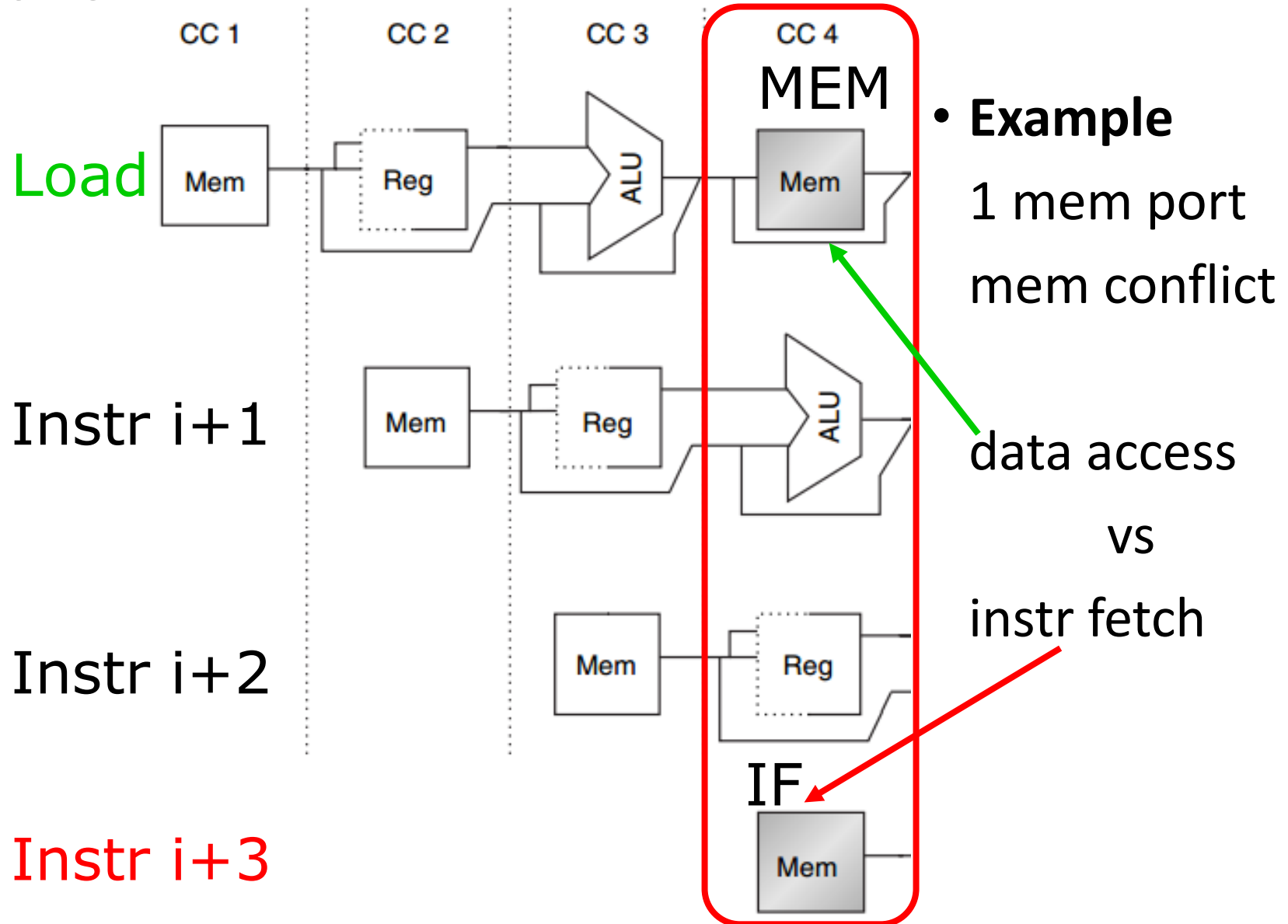
- Hazards in pipelines can make it necessary to stall the pipeline.
- Avoiding a hazard often requires that some instructions in the pipeline be allowed to proceed while others are delayed.
- when an instruction is stalled, all instructions issued later than the stalled instruction are also stalled.
- Instructions issued earlier than the stalled instruction must continue, since otherwise the hazard will never clear.
- As a result, no new instructions are fetched during the stall.
- We will see several examples of how pipeline stalls operate in this lecture—don't worry, they aren't as complex as they might sound!

Structural Hazard

- arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
- For example, a processor has only 1 register write port but intends two writes in the same clock cycle.
- **Solution** to structural hazard is to stall one of the instructions, let one instruction use the write port first, and then activate the other instruction when the write port is available.
- Structural hazards are easy to eliminate – increase the number of resources (for example, implement a separate instruction and data cache)

Structural Hazard

- Here's an example of structural hazard due to memory conflict.
- Assume the processor has only memory port.
- A structural hazard will arise in clock cycle 4 when the load instruction reads data from memory and instruction i plus 3 fetches instruction from memory.

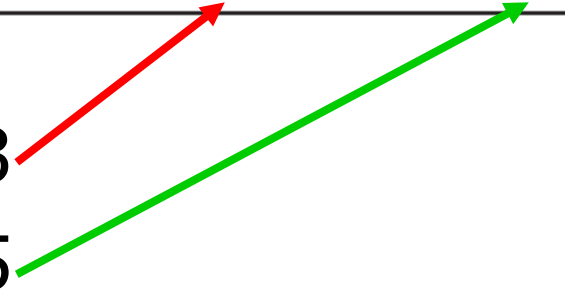


Structural Hazard

Instruction	1	2	3	4	5
Load instruction	IF	ID	EX	MEM	WB
Instruction $i + 1$		IF	ID	EX	MEM
Instruction $i + 2$			IF	ID	EX
<u>Instruction $i + 3$</u>				stall	<u>IF</u>

- The solution to this structural hazard is stall instruction $i+3$ for one clock cycle.
- A stall is commonly called a pipeline bubble or just bubble

Stall Instr $i+3$
till CC 5



Structural Hazard

	Clock cycle number									
Instruction	1	2	3	4	5	6	7	8	9	10
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				Stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

Because the instruction being fetched is stalled, all other instructions in the pipeline before the stalled instruction can proceed normally. The stall cycle will continue to pass through the pipeline, so that no instruction completes on clock cycle 8. Note that this figure assumes that instructions $i + 1$ and $i + 2$ are not memory references.

Performance of pipeline with stalls

- Speedup from pipelining =
$$\frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$
$$= \frac{CPI_{\text{unpipelined}} \times \text{Clock cycle unpipelined}}{CPI_{\text{pipelined}} \times \text{Clock cycle pipelined}}$$
- As $\text{Clock cycle unpipelined} = \text{Clock cycle pipelined}$

So Speedup =
$$\frac{CPI_{\text{unpipelined}}}{CPI_{\text{pipelined}}}$$

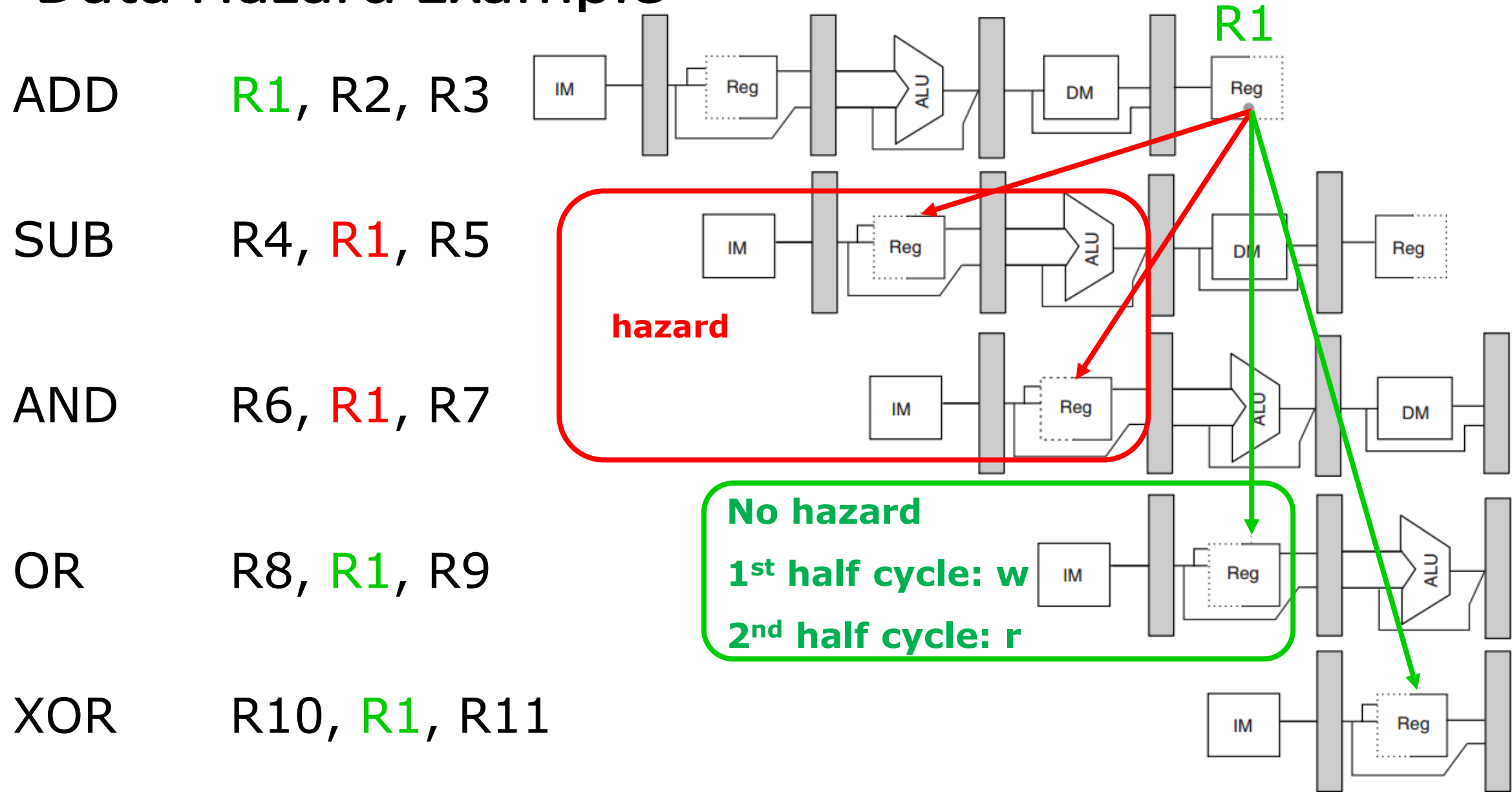
Performance of pipeline with stalls

- The ideal CPI on a pipelined processor is almost always 1. Hence, we can compute the pipelined CPI:
- $CPI_{pipelined} = 1 + \text{Number of stalls per Instruction}$
- $\text{Speedup} = \frac{CPI_{unpipelined}}{1 + \text{Number of stalls per Instruction}}$

Data Hazard

- Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor.

Data Hazard Example



Data Hazard Example

- In this example, the SUB and AND instructions need R1 before the ADD instruction prepares it.
- So R1 causes a data hazard that prevents normal pipelining of the subtract and AND instructions.
- Note that the OR instruction has no hazard because the add instruction prepares R1 in the first half of the clock cycle while the OR instruction needs R1 till the second half.

Solve Data Hazard With Stall Example

Clock Cycle	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
ADD R1,R2,R3	IF	ID	EX	Mem	WB						
SUB R4,R1,R5		IF	Stall	Stall	ID	EX	Mem	WB			
AND R6,R1,R7			Stall	Stall	IF	ID	EX	Mem	WB		
OR R8,R1,R9			Stall	Stall		IF	ID	EX	Mem	WB	
XOR R10,R1,R11			Stall	Stall			IF	ID	EX	Mem	WB

This solution is based on inserting Hazard Detector in Decoding Stage, if it found hazard insert a stall, and starting from Decoding stage

Data Hazard

- Hazards can be detected in software (Compiler) or the hardware, so Hardware may be able to detect the Data hazards and then it produces a Stall in the pipeline to delay execution of next instructions until hazard is solved.
- If the compiler detects that it can produce stalls to the pipeline, or choose other technique to solve the data hazard.
- Using stalls affects the pipeline performance as it affects the number of executing instructions at a time, so using other techniques are preferred:
 - Forward and Bypassing.
 - Out of Order execution(Tomasulo's algorithm).

Data Hazard - Forward and Bypassing

- Solution: forwarding directly feed back EX/MEM&MEM/WB pipeline regs' results to the ALU inputs;
- if forwarding hw detects that previous ALU has written the reg corresponding to a source for the current ALU, control logic selects the forwarded result as the ALU input.
- Using special decode signals and multiplexer , we can make the input to the execute stage uses inputs from the latches contains the output of the ALU operations instead of having the input from the register file , so we can forward the result from the ALU output directly to the input of the ALU , so we don't have to wait until the values are written to the register file and then read again.

Data Hazard: Forwarding

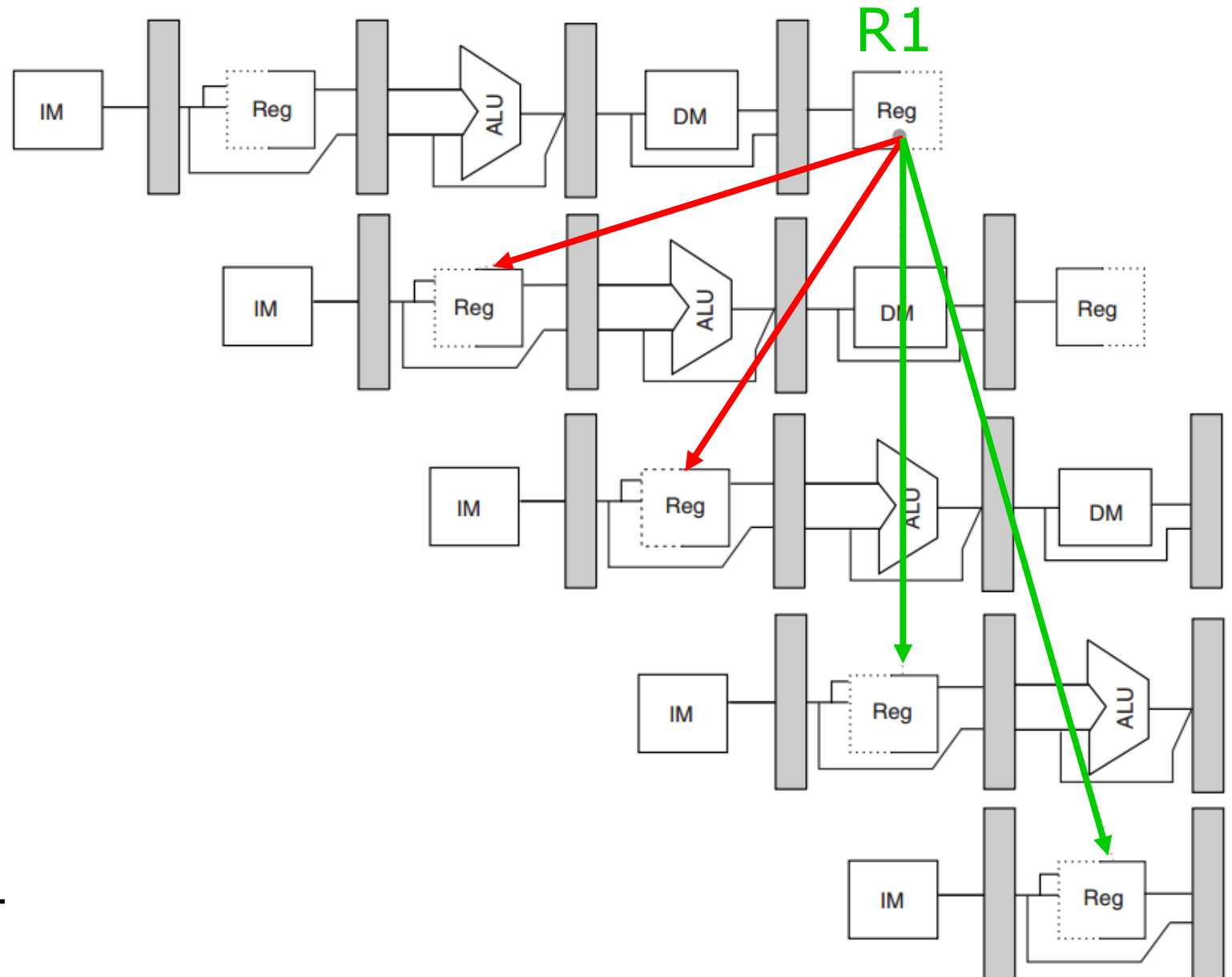
DADD R1, R2, R3

DSUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

XOR R10, R1, R11



Data Hazard: Forwarding

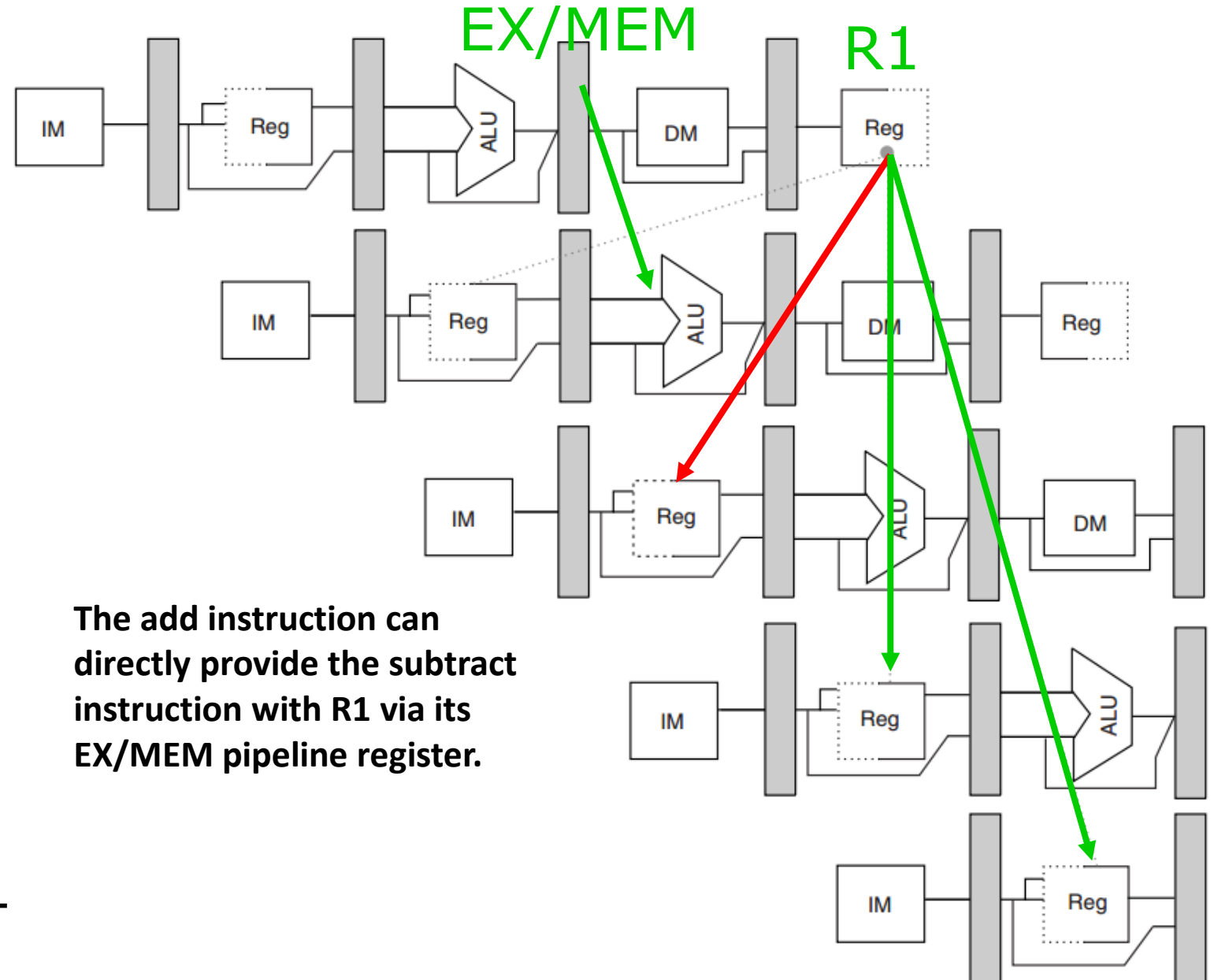
DADD **R1**, R2, R3

DSUB R4, **R1**, R5

AND R6, **R1**, R7

OR R8, **R1**, R9

XOR R10, **R1**, R11



Data Hazard: Forwarding

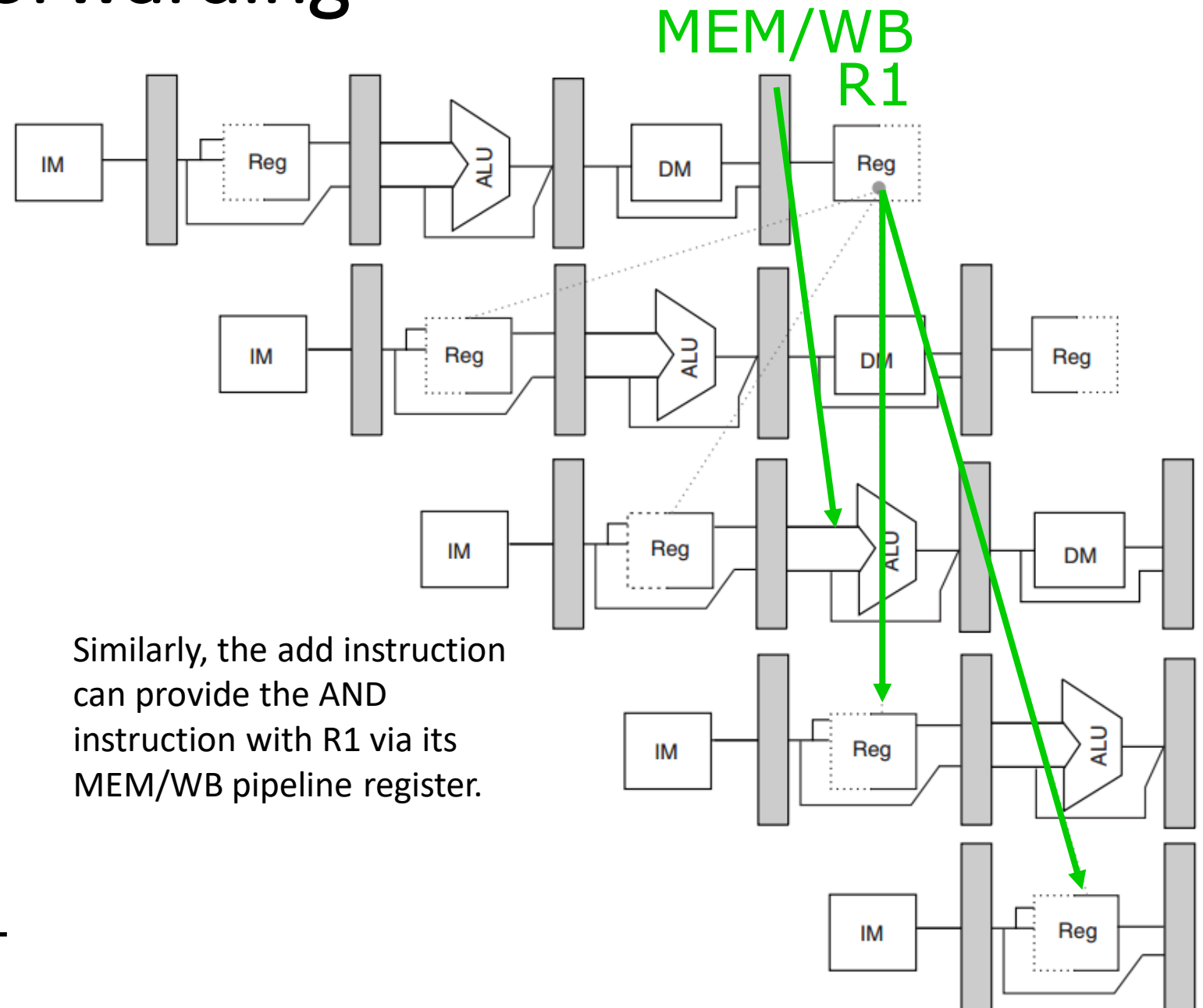
DADD R1, R2, R3

DSUB R4, R1, R5

AND R6, R1, R7

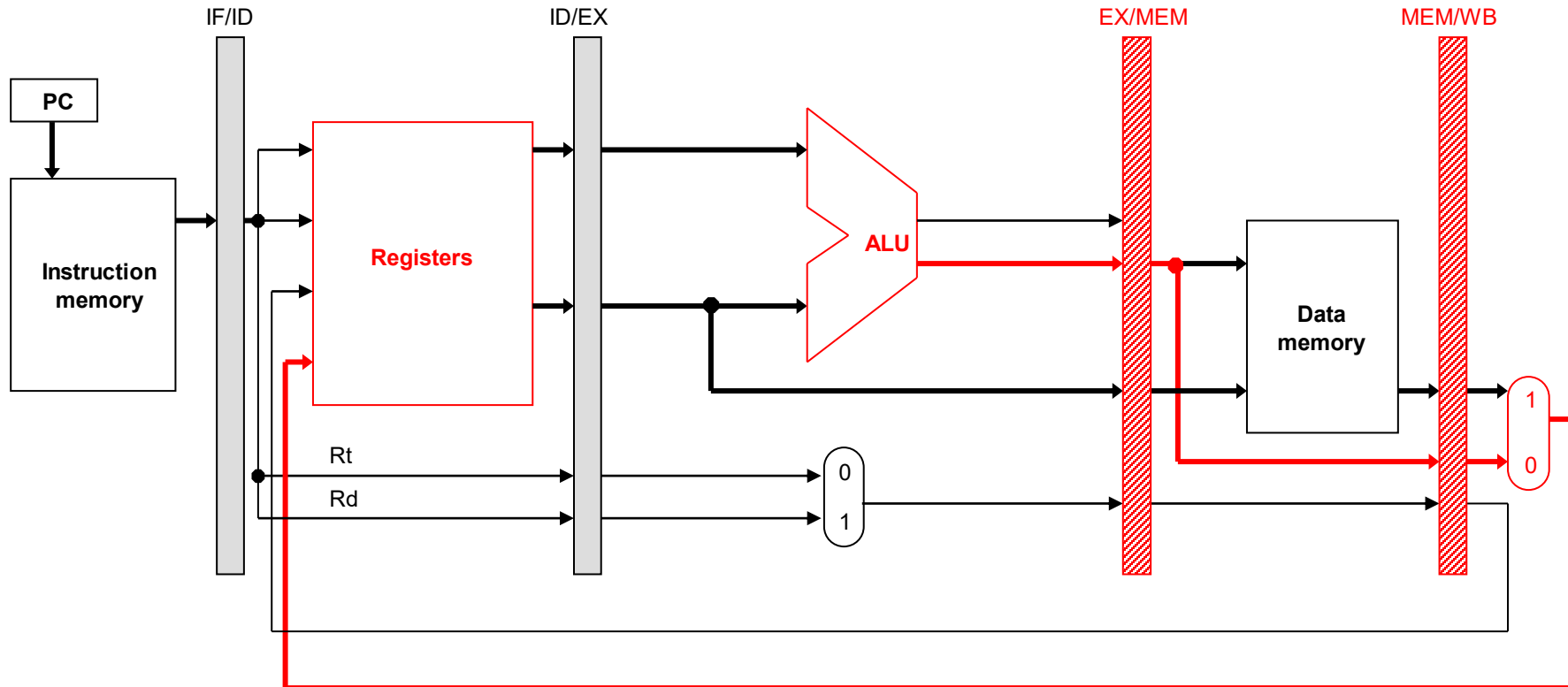
OR R8, R1, R9

XOR R10, R1, R11



Pipeline Registers to the rescue!

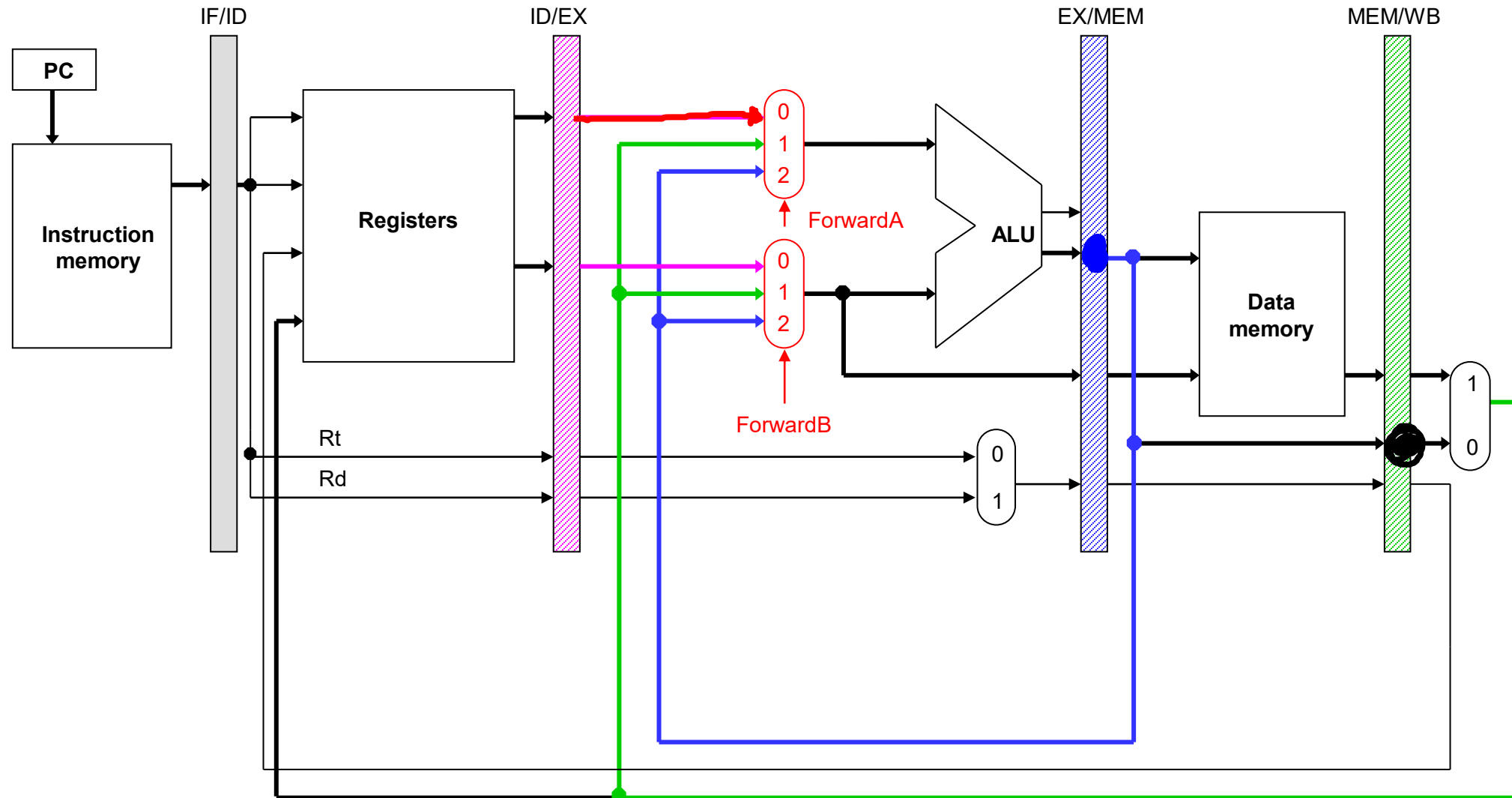
- Pipeline stages communicate through pipeline registers:
IF/ID ID/EX EX/MEM MEM/WB
- We “forward” data from pipeline registers to later instructions



Outline of forwarding hardware

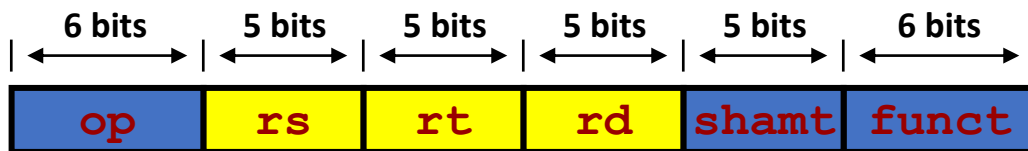
- A **forwarding unit** selects the correct ALU inputs for the EX stage:
 - No hazard: ALU's operands come from the **register file**, like normal
 - Data hazard: operands come from either the **EX/MEM** or **MEM/WB** pipeline registers instead
- The ALU sources will be selected by two new multiplexers, with control signals named **ForwardA** and **ForwardB**
- **ForwardA** (First operand register Rs)
- **ForwardB** (Second operand register Rt)

Simplified Datapath with forwarding MUXes



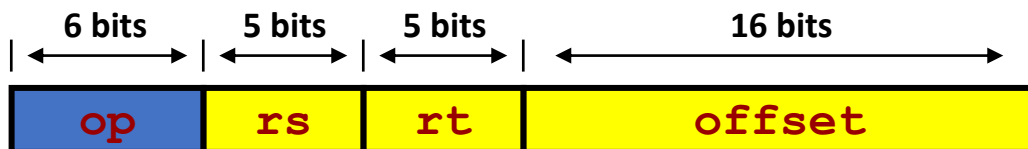
RISC Instruction Format

- Instructions are divided into three types: R (register), I (immediate), and J (jump).
- Every instruction starts with a 6-bit opcode.
- In addition to the opcode, R-type instructions specify three registers, a shift amount field, and a function field;
- I-type instructions specify two registers and a 16-bit immediate value; J-type instructions follow the opcode with a 26-bit jump target.

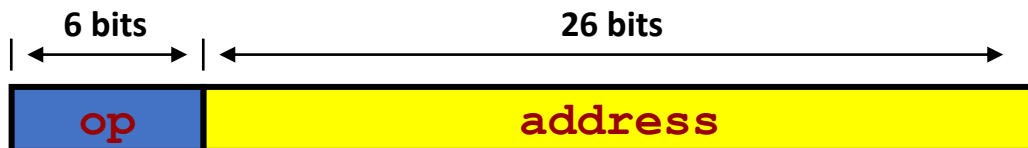


R-Format

- rs: 1st register operand (register source)
- rt: 2nd register operand
- rd: register destination



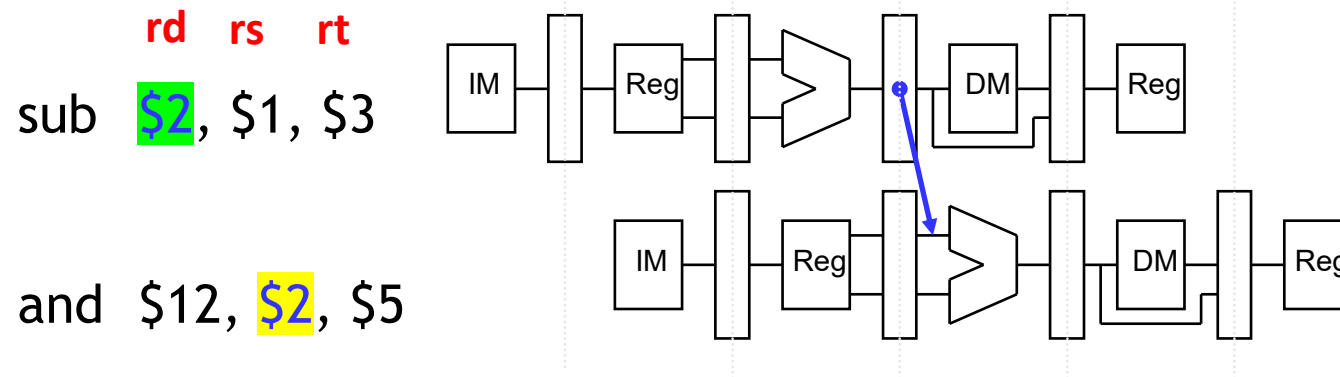
I-Format



J-Format

Detecting EX/MEM data hazards

- So how can the hardware determine if a hazard exists?
- An **EX/MEM hazard** occurs between the instruction currently in its EX stage and the previous instruction if:
 1. The previous instruction will write to the register file, *and*
 2. The destination is one of the ALU source registers in the EX stage.
- There is an EX/MEM hazard between the two instructions below.



- Data in a pipeline register can be referenced using a class-like syntax. For example, **ID/EX.RegisterRs** refers to the **rs** field stored in the ID/EX pipeline. while **EX/MEM.RegisterRd** refers to the **rd** field stored in EX/MEM pipeline

Detecting EX/MEM data hazards

- The first ALU source comes from the pipeline register when necessary.

if (EX/MEM.RegWrite = 1
 and EX/MEM.RegisterRd = ID/EX.RegisterRs)
then ForwardA = 2 (2 refer to EX/MEM register)

- The second ALU source is similar.

if (EX/MEM.RegWrite = 1
 and EX/MEM.RegisterRd = ID/EX.RegisterRt)
then ForwardB = 2 (2 refer to EX/MEM register)

Detecting MEM/WB data hazards

- A **MEM/WB hazard** may occur between an instruction in the EX stage and the instruction from two cycles ago.
- One new problem is if a register is updated twice in a row.

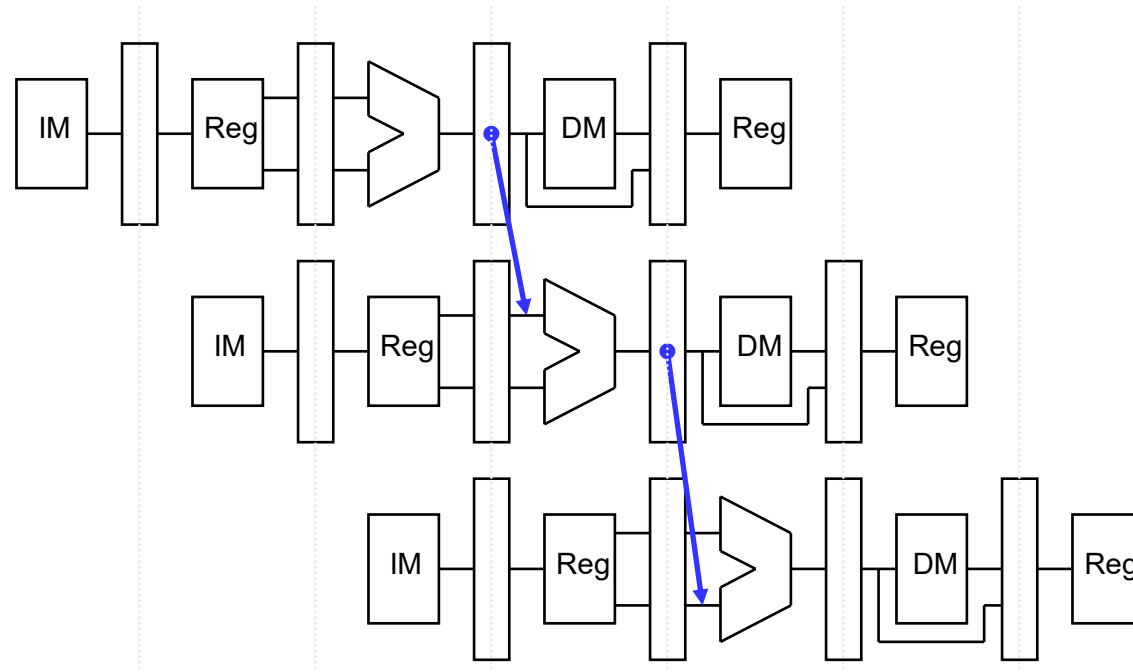
```
add  $1, $2, $3
add  $1, $1, $4
sub   $5, $5, $1
```

- Register \$1 is written by *both* of the previous instructions; from which instruction should it receive its value?

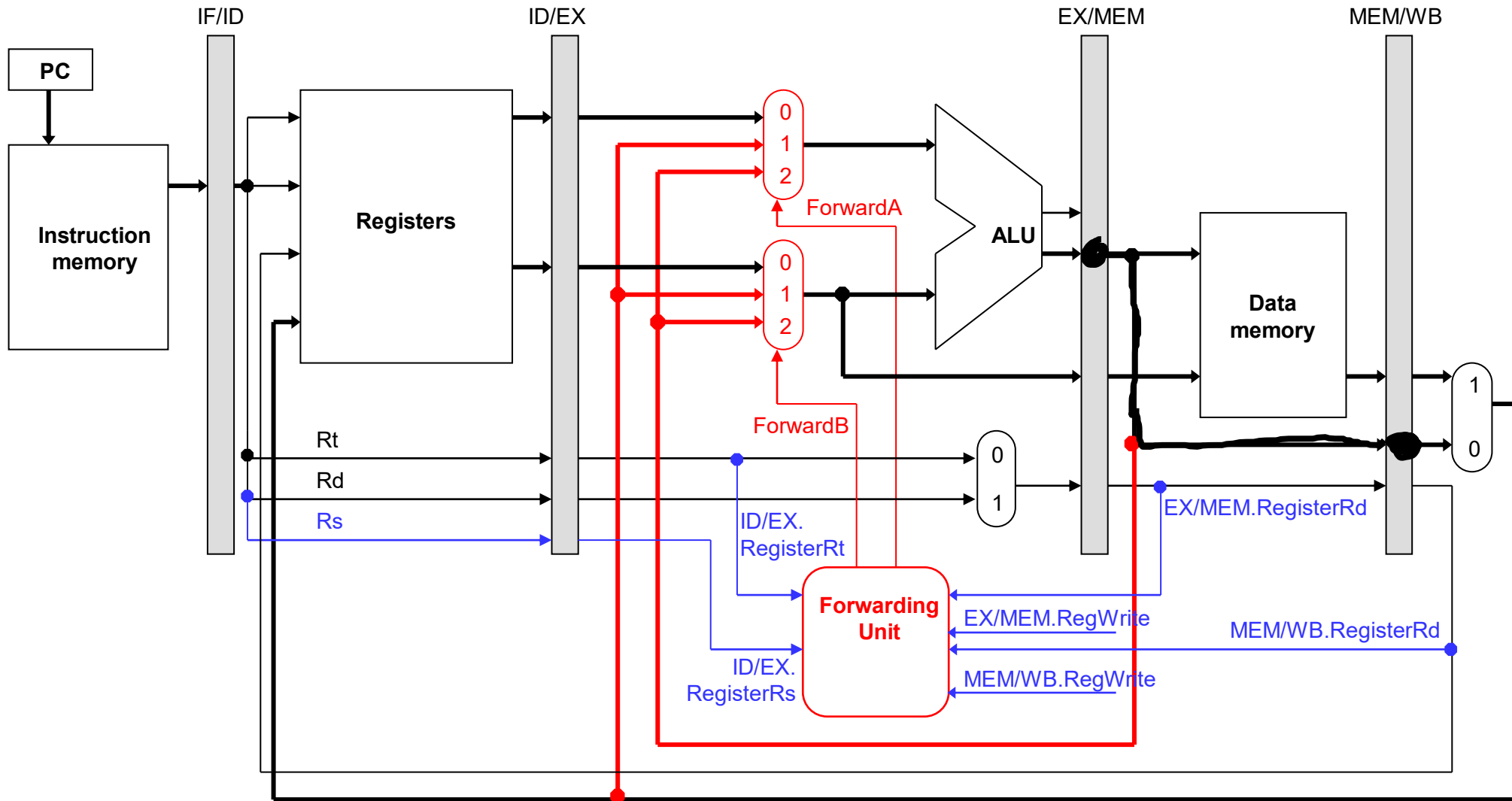
add \$1, \$2, \$3

add \$1, \$1, \$4

sub \$5, \$5, \$1



Simplified Datapath with forwarding



MEM/WB hazard equations

- Here is an equation for detecting and handling MEM/WB hazards for the first ALU source.

if ($\text{MEM/WB.RegWrite} = 1$
 and $\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs}$
 and ($\text{EX/MEM.RegisterRd} \neq \text{ID/EX.RegisterRs}$ or $\text{EX/MEM.RegWrite} = 0$)
then $\text{ForwardA} = 1$ (**1 refer to MEM/WB register**)

- The second ALU operand is handled similarly.

if ($\text{MEM/WB.RegWrite} = 1$
 and $\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt}$
 and ($\text{EX/MEM.RegisterRd} \neq \text{ID/EX.RegisterRt}$ or $\text{EX/MEM.RegWrite} = 0$)
then $\text{ForwardB} = 1$ (**1 refer to MEM/WB register**)

The forwarding unit

- The forwarding unit has several control signals as inputs.

ID/EX.RegisterRs EX/MEM.RegisterRd MEM/WB.RegisterRd
ID/EX.RegisterRt EX/MEM.RegWrite MEM/WB.RegWrite

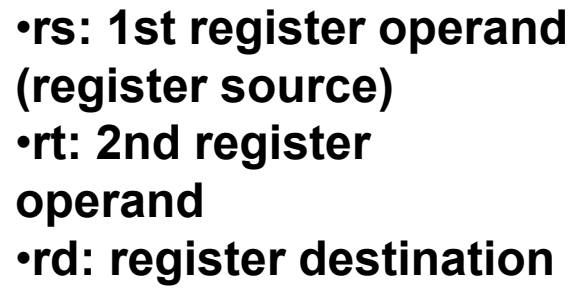
(The two RegWrite signals are not shown in the diagram, but they come from the control unit.)

- The forwarding unit outputs are selectors for the **ForwardA** and **ForwardB** multiplexers attached to the ALU. These outputs are generated from the inputs using the equations on the previous pages.
- Some new buses route data from pipeline registers to the new muxes.

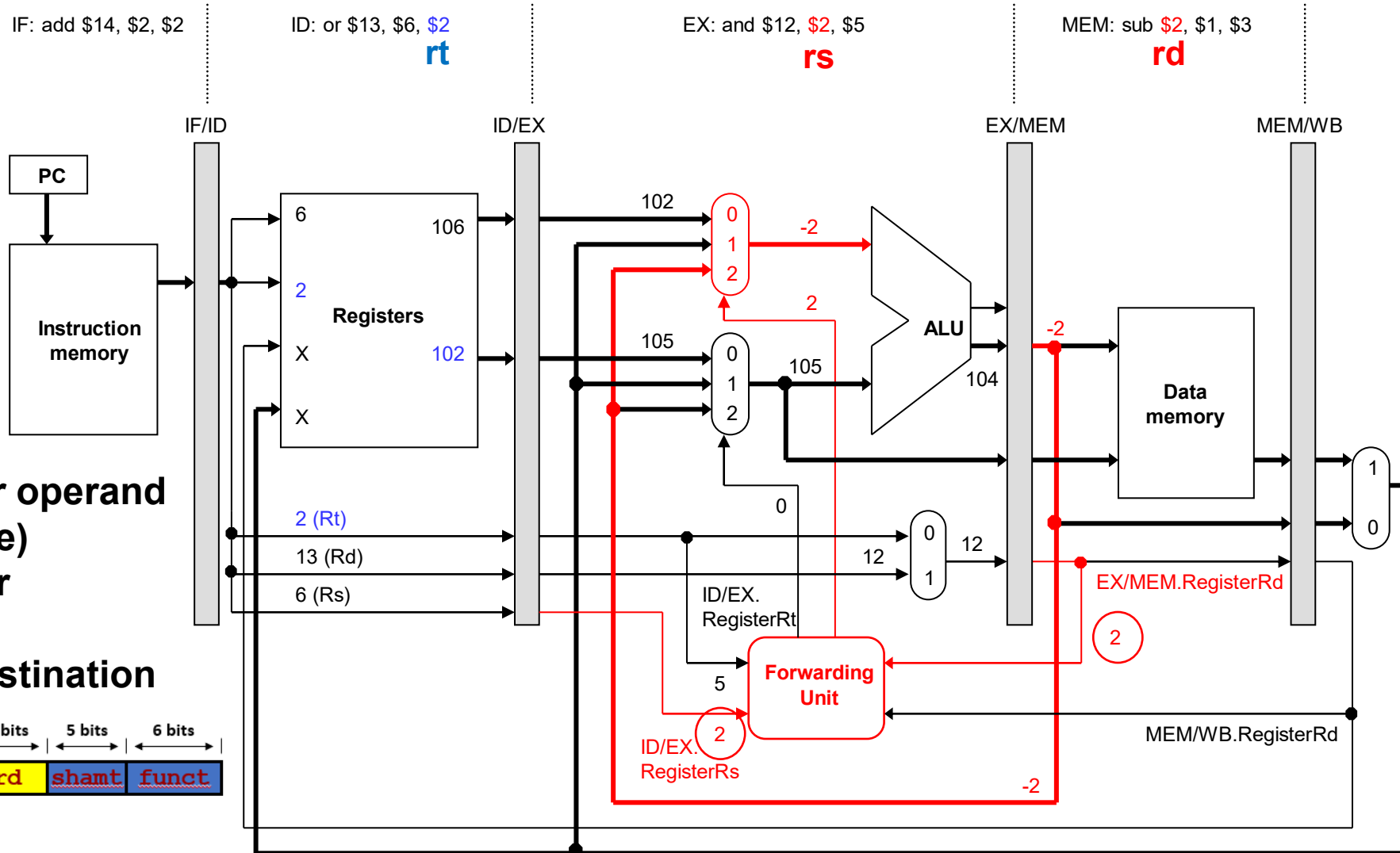
Example

```
sub    $2, $1, $3  
and    $12, $2, $5  
or     $13, $6, $2  
add    $14, $2, $2  
sw     $15, 100($2)
```

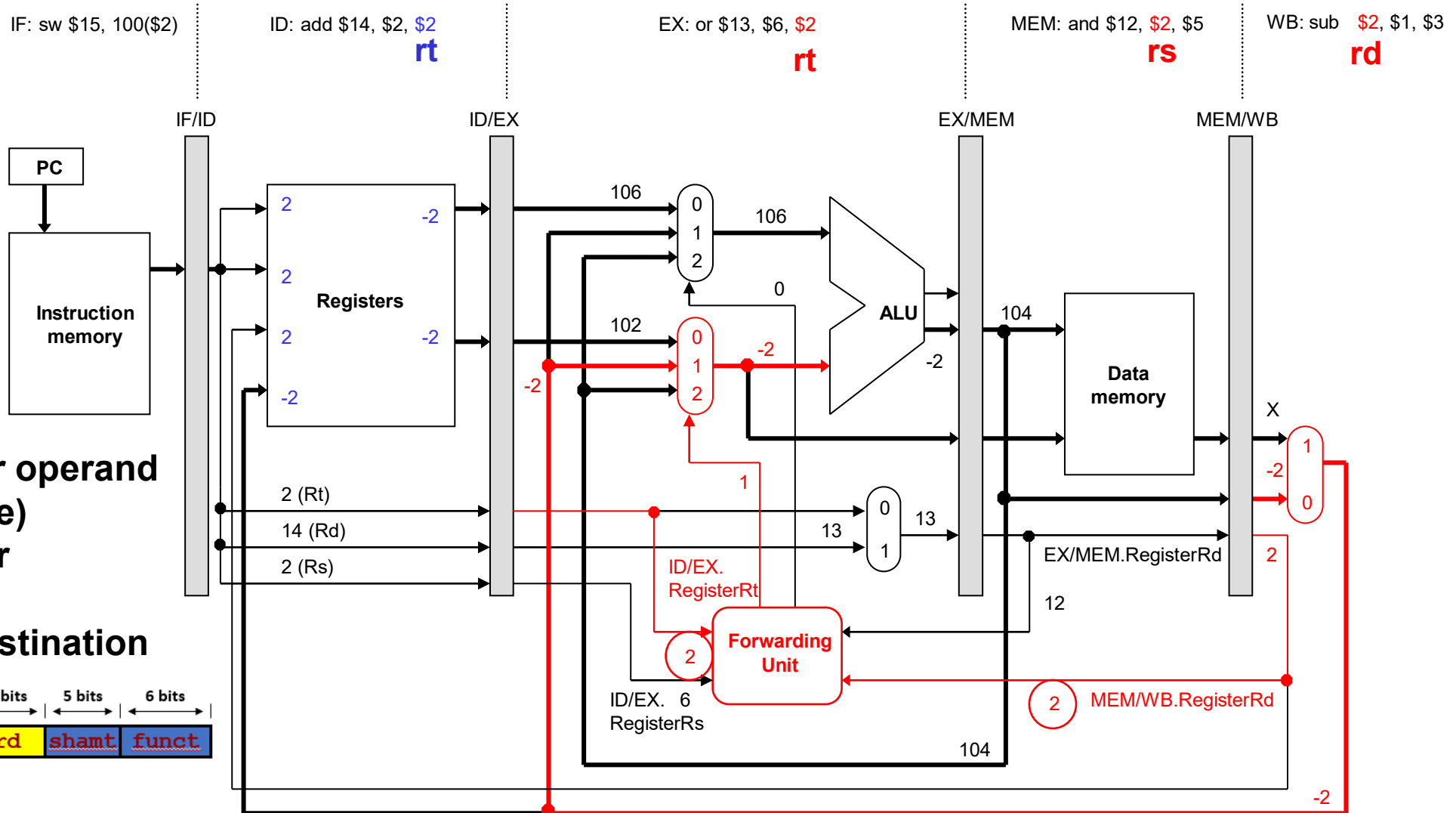
- Assume again each register initially contains its number plus 100.
 - After the first instruction, \$2 should contain -2 (101 - 103).
 - The other instructions should all use -2 as one of their operands.
- We'll try to keep the example short.
 - Assume no forwarding is needed except for register \$2.
 - We'll skip the first two cycles, since they're the same as before.



Clock cycle 4: forwarding \$2 from EX/MEM

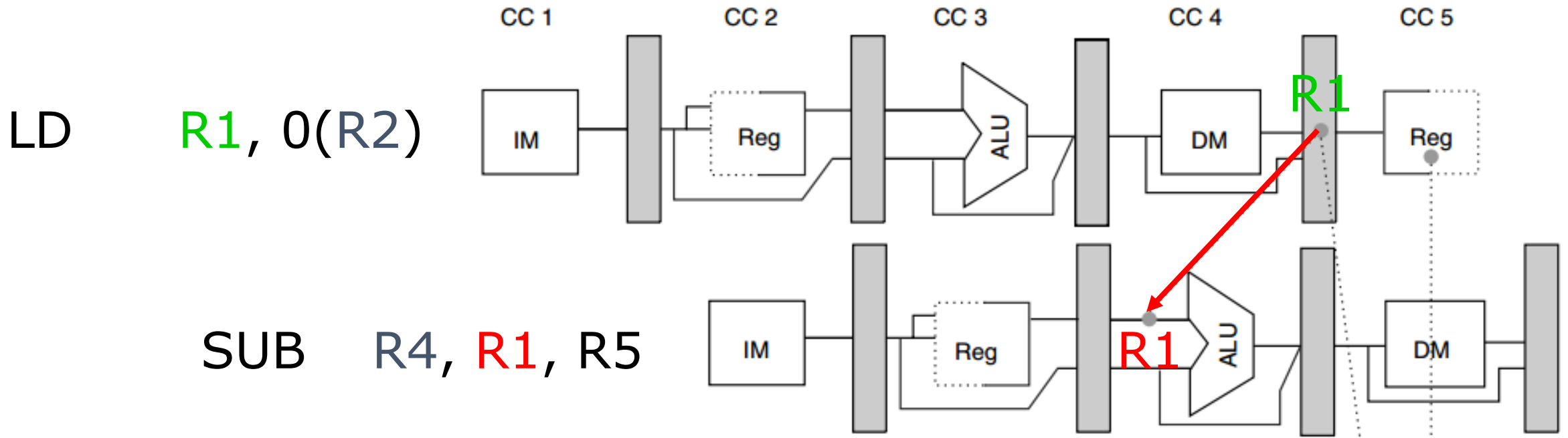


Clock cycle 5: forwarding \$2 from MEM/WB



Data Hazard

- Sometimes stall is necessary



In this example, the load instruction prepares R1 till it reaches the MEM/WB pipeline register at the end of clock cycle 4.

The subtract instruction, however, requires R1 at the beginning of clock cycle 4. So in this case, no forwarding can be backward and thus the subtract instruction has to stall for one clock cycle

Forwarding cannot be backward.

Has to stall.

Example for the following sequence of instruction create the table of 5 stage pipelines With Forward

LD R1, 0(R2)											
SUB R4,R1,R5											
AND R6,R1,R7											
OR R8,R1,R9											

Sometimes stall is necessary beside forwarding

Example for the following sequence of instruction create the table of 5 stage pipelines With Forward

LD R1, 0(R2)	IF	ID	EX	MEM	WB						
SUB R4,R1,R5		IF	ID								
AND R6,R1,R7			IF								
OR R8,R1,R9											

Sometimes stall is necessary beside forwarding


Example for the following sequence of instruction create the table of 5 stage pipelines With Forward

LD R1, 0(R2)	IF	ID	EX	MEM	WB						
SUB R4,R1,R5		IF	ID	Stall							
AND R6,R1,R7			IF	Stall							
OR R8,R1,R9				Stall							

Sometimes stall is necessary beside forwarding

Example for the following sequence of instruction create the table of 5 stage pipelines With Forward

LD R1, 0(R2)	IF	ID	EX	MEM	WB						
SUB R4,R1,R5		IF	ID	Stall	EX	MEM	WB				
AND R6,R1,R7			IF	Stall	ID	EX	MEM	WB			
OR R8,R1,R9				Stall	IF	ID	EX	MEM	WB		



References

- Computer System Architecture (3rd Edition) by M. Morris Mano
- Computer organization and architecture designing for performance (8th edition) by William Stallings
- Computer Architecture: A Quantitative Approach 5th Edition by John L. Hennessy
- Patterson & Hennessy, “Computer Organization & Design”
- Lecture Notes, Lecture 4: Pipelining Basics & Hazards, Kai Bu, kaibu@zju.edu.cn
- EmbeededNotes Pipeline ,
https://www.mediafire.com/file/yl157ng7eby7989/Video_6_-_PipeLine_-_1_Intro.pptx/file
- <https://www.youtube.com/watch?v=X84uO3H83bA>
- Lecture Notes, Introduction to Computer Architecture, at the University of California, Santa Barbara. © Behrooz Parhami

Thanks

