

# Pipelining



E. Sanchez, M. Sonza Reorda

Politecnico di Torino

Dipartimento di Automatica e Informatica (DAUIN)

Torino - Italy

This work is licensed under the Creative Commons (CC BY-SA) License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/>



# Introduction

- Pipelining is an implementation technique whereby multiple instructions are overlapped in execution
- In a pipeline, different units (called *pipe stages* or *segments*) are completing different parts of different instructions in parallel.

# Example

Clock cycle → 1 2 3 4 5 6 7 8

Instruction

I<sub>1</sub>

F	D	O	W
---	---	---	---

I<sub>2</sub>

F	D	O	W
---	---	---	---

I<sub>3</sub>

F	D	O	W
---	---	---	---

I<sub>4</sub>

F	D	O	W
---	---	---	---

# Definitions

- The *throughput* of a pipelined processor is the number of instructions which exit the pipeline in the time unit
- All the pipeline stages are synchronized (they proceed to executing a new task all together); the time for executing one step is called *machine cycle*, and normally corresponds to one clock cycle
- The length of the machine cycle is determined by the slowest stage
- CPI stands for *clock cycles per instruction*.

# Ideal pipeline

- In an ideal pipeline, all the stages are perfectly balanced (i.e., they require the same time)
- The throughput of an ideal pipeline (i.e., the number of instructions completed in a given time period) is

$$\text{throughput}_{\text{pipelined}} = \text{throughput}_{\text{unpipelined}} * n$$

being  $n$  the number of pipeline stages.

# Example processor:

## Unpipelined Implementation

- The execution of each instruction may be composed of at most five clock cycles:
  - Instruction fetch cycle (IF)
  - Instruction decode/register fetch cycle (ID)
  - Execution/effective address cycle (EX)
  - Memory access/branch completion cycle (MEM)
  - Write-back cycle (WB).

# Instruction Fetch cycle

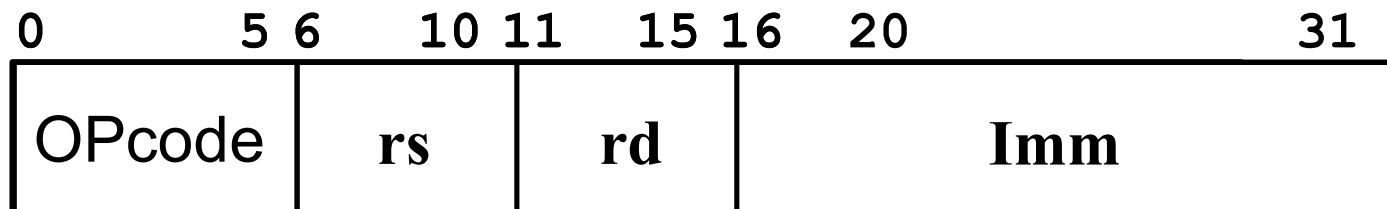
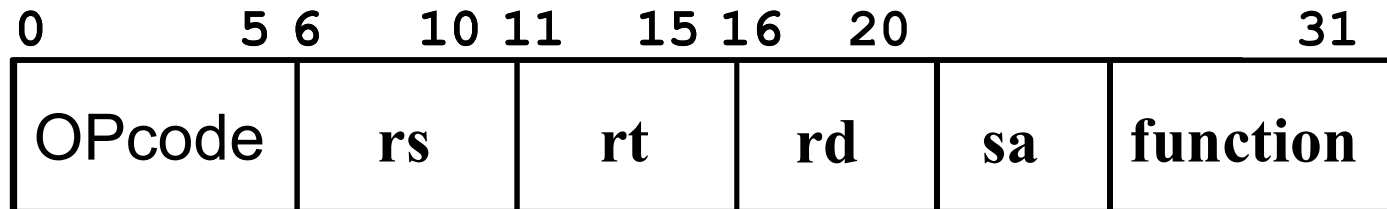
$$IR \leftarrow \text{Mem}[PC]$$
$$NPC \leftarrow PC + 4$$

# Instruction Decode/ Register Fetch

$A \leftarrow \text{Regs}[\text{IR}6..10]$

$B \leftarrow \text{Regs}[\text{IR}11..15]$

$\text{Imm} \leftarrow (([\text{IR}]_{16})^{16} \text{##IR}16..31)$



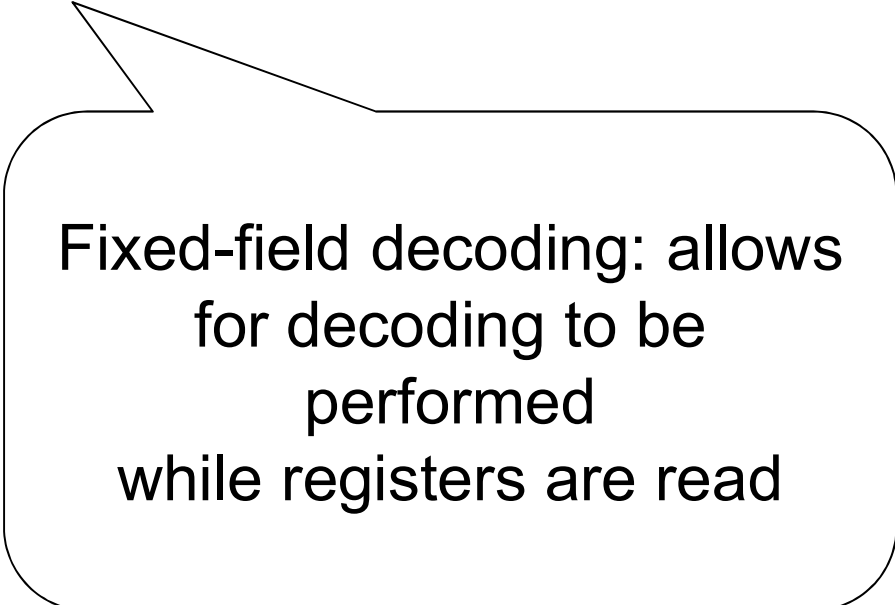


# Instruction Decode/ Register Fetch

$A \leftarrow \text{Regs}[\text{IR}6..10]$

$B \leftarrow \text{Regs}[\text{IR}11..15]$

$\text{Imm} \leftarrow (([\text{IR}]_{16})^{16} \text{##IR}16..31)$



Fixed-field decoding: allows  
for decoding to be  
performed  
while registers are read

# Instruction Decode/ Register Fetch

$A \leftarrow \text{Regs}[\text{IR}6..10]$

$B \leftarrow \text{Regs}[\text{IR}11..15]$

$\text{Imm} \leftarrow (([\text{IR}]_{16})^{16} \text{##IR}16..31)$



Sign extension

# Execution/Effective Address Cycle

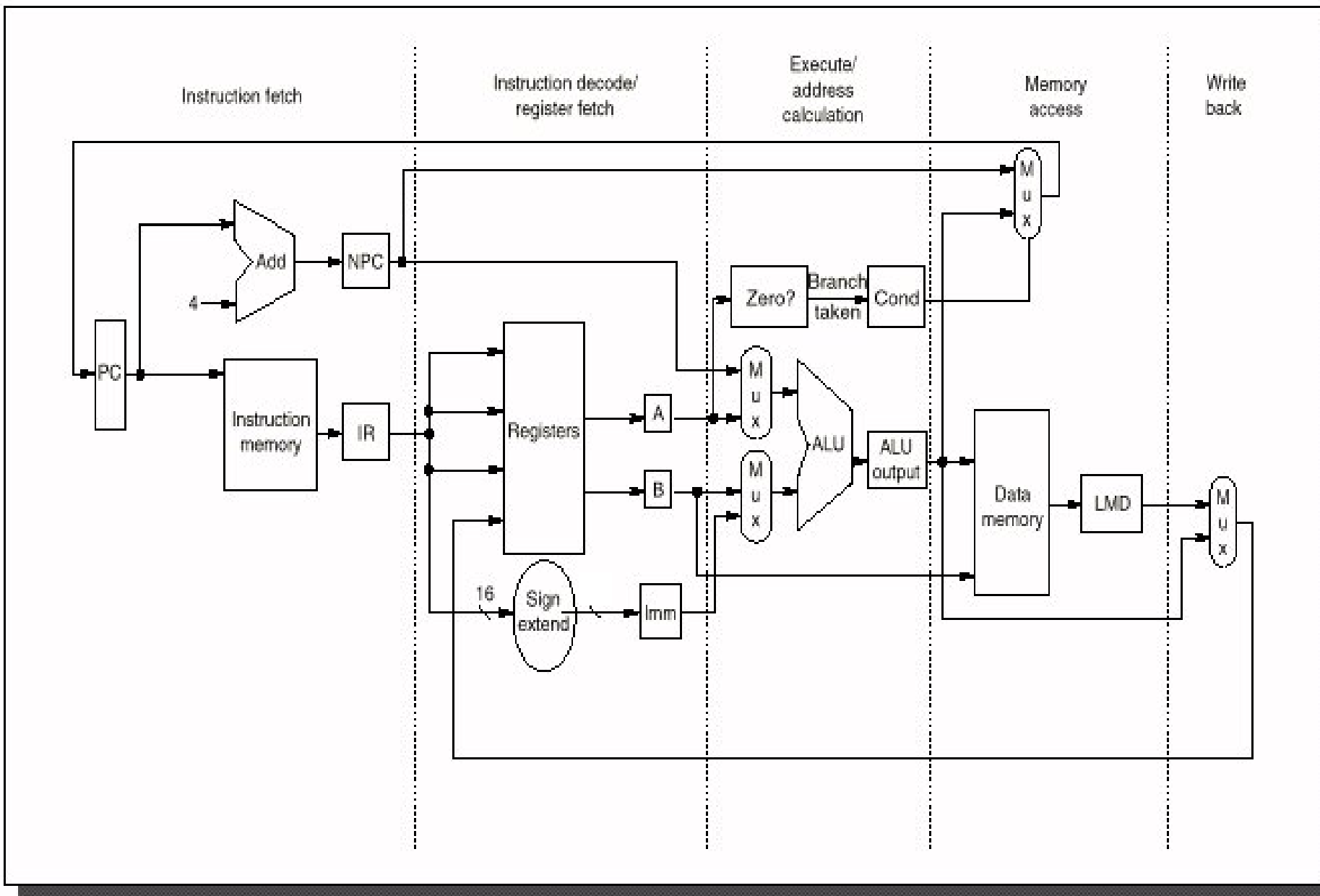
- Memory reference  
 $\text{ALUOutput} \leftarrow A + \text{Imm};$
- Register-Register ALU instruction  
 $\text{ALUOutput} \leftarrow A \text{ op } B;$
- Register-Immediate ALU instruction  
 $\text{ALUOutput} \leftarrow A \text{ op } \text{Imm};$
- Branch  
 $\text{ALUOutput} \leftarrow \text{NPC} + \text{Imm};$   
 $\text{Cond} \leftarrow (A \text{ op } 0);$

# Memory Access/Branch Completion Cycle

- Memory reference  
LMD  $\leftarrow$  Mem[ALUOutput] or  
Mem[ALUOutput]  $\leftarrow$  B;
- Branch  
if (cond) PC  $\leftarrow$  ALUOutput else PC  $\leftarrow$  NPC;

# Write-back Cycle

- Register-Register ALU instruction  
     $\text{Regs}[\text{IR16}..\text{20}] \leftarrow \text{ALUOutput};$
- Register-Immediate ALU instruction  
     $\text{Regs}[\text{IR11}..\text{15}] \leftarrow \text{ALUOutput};$
- Load instruction  
     $\text{Regs}[\text{IR11}..\text{15}] \leftarrow \text{LMD};$



# Behavior and optimizations

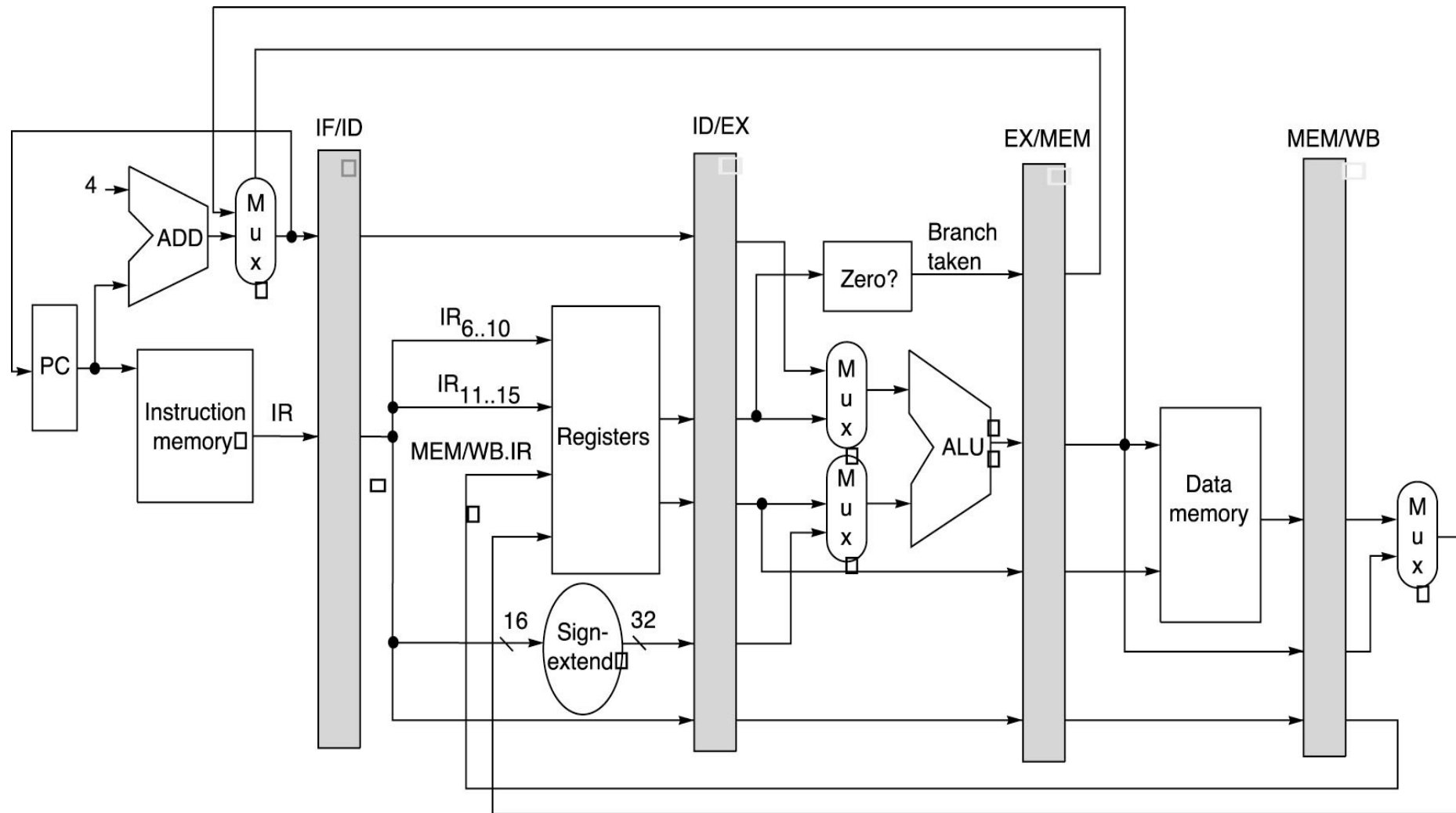
- All instructions require 5 clock cycles, unless branch instructions, which require 4 clock cycles
- Optimizations could be done for reducing the average CPI: as an example, the ALU instructions could be completed during the MEM cycle
- Hardware resources could be optimized by avoiding duplications (e.g., for ALUs, and memory)
- An alternative single-clock architecture (i.e., executing an instruction per clock cycle) can also be considered
- A simple control unit is required to produce the signals required by the datapath.

# Example processor: basic pipelined version

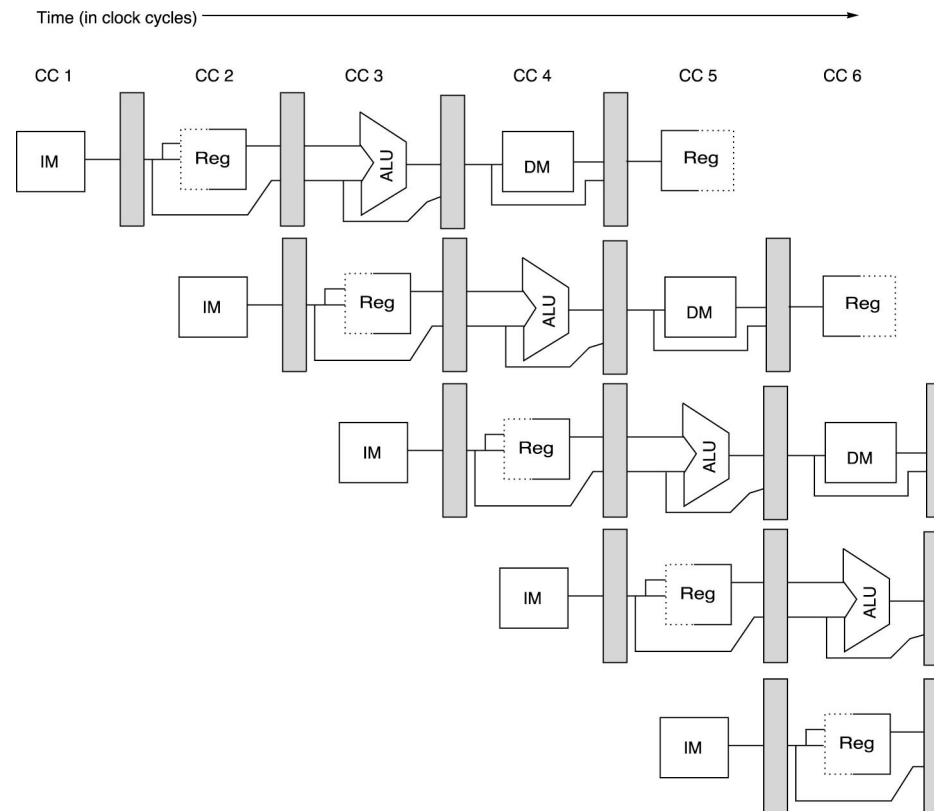
- A new instruction is started at each clock cycle
- Different resources work on different instructions at the same time
- At every clock cycle, each resource can be used for one purpose, only. This means that
  - Separate instruction and data memories (i.e., caches) must be used
  - The register file is used in two stages: for reading (second half of the cc) in ID and for writing (first half of the cc) in WB. It must be designed to satisfy these requests during the same clock cycle.
  - The PC must be changed in the IF stage. What about branches?
- Pipeline registers must be added between stages.



# Pipelined data path



# Evolution in time



# Pipeline performance

- Pipelining increases the processor throughput without making single instructions faster.
- Single instruction processing is made slower due to the pipeline control overheads.
- The depth of a pipeline is limited by
  - the need for balanced stages
  - pipelining overhead (pipeline register delay and clock skew).

# Example

- Consider the unpipelined processor, and suppose that
  - The clock cycle is 1 ns
  - ALU operations and branches require 4 cycles
  - Memory operations require 5 cycles
  - The relative frequency of these operations is 40% (ALU), 20% (branches), and 40% (mem).

- The average instruction execution time is

$$\begin{aligned} & \text{Clock cycle} \times \text{average CPI} \\ &= 1 \text{ ns} \times ((0.4 + 0.2) \times 4 + 0.4 \times 5) \\ &= 1 \text{ ns} \times 4.4 \\ &= 4.4 \text{ ns} \end{aligned}$$

## Example (II)

- Suppose that moving to the pipelined architecture slows down the clock of the slowest stage by 20%.
- The average instruction execution time is therefore 1.2 ns.
- The speedup introduced by pipelining is  
$$\text{speedup} = 4.4 \text{ ns} / 1.2 \text{ ns} = 3.7 \text{ times}$$

# Pipeline Hazards

- Hazards are situations that prevent an instruction from executing during its designated clock cycle.
- There are three classes of hazards:
  - *structural hazards*: coming from resource conflicts
  - *data hazards*: an instruction depends on the result of a previous instruction
  - *control hazards*: depend on branches and other instructions that change the PC.

# Stalls

- One way of dealing with hazards is to force the pipeline to stall, i.e., to block instructions for one or more clock cycles.
- When an instruction is stalled:
  - the instructions following the stalled instruction are also stalled
  - the instructions preceding the stalled instruction continue.
- A stall causes the introduction of a *bubble* in the pipeline.

# Example

Instruction	Clock cycle number									
	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



# Example

Suppose that only one access to memory can happen during each clock cycle: therefore, fetch of instruction  $i+3$  can not be performed here and must be delayed.

	Clock cycle									
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

# Example

As a consequence of the stall, no instruction is completed at clock cycle #8.

	Clock cycle number									
Instruction	1	2	3	4	5	6	7	8		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

# Structural hazards

- They may happen when some pipeline unit is not able to execute all the operations scheduled for a given cycle.
- Examples:
  - A given unit is not able to complete its task in one clock cycle
  - The pipeline owns only one register-file write port, but there are cycles in which two register writes are required
  - The pipeline refers to a single-port memory, and there are cycles in which different instructions would like to access to the memory together.

# Example

In this clock cycle the processor accesses twice to memory (one to data mem and the other to instr mem)

	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

# Example

If the processor includes two caches, the stall can be avoided

	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

# Example

In this clock cycle the register file is accessed twice

	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

# Example

If the register file is dual-port, the stall can be avoided

	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

# Removing Structural Hazards

- This requires adding new hardware or improving the existing one.
- The designer has to trade-off between performance and cost, basing on the frequency of occurrence of structural hazards.



# Example

- Load structural hazards happen when two instructions contemporarily try to make a memory access to a single-port memory.
- Assume that:
  - 40% of instructions make access to memory
  - the machine with structural hazard has a clock 1.05 times faster than the one without.
- How much faster is the machine without structural hazard?

# Solution

- For the machine without structural hazard:
  - Average Instruction Time =  $\text{CPI} \times \text{Clock Cycle Time}$
- For the machine with structural hazard:
  - Average Instruction Time =  $\text{CPI} \times \text{Clock Cycle Time}$ 
$$= (1 + 0.4 \times 1) \times \frac{\text{Clock Cycle Time}_{\text{ideal}}}{1.05}$$
$$= 1.3 \times \text{Clock Cycle Time}_{\text{ideal}}$$

# Data hazards

- Overlapping the execution of instructions, as it is done by pipelining, changes the order of read/write accesses to operands.
- This can result in:
  - wrong results
  - undeterministic behavior.

# Example

- Let consider the following code fragment:

ADD R1, R2, R3

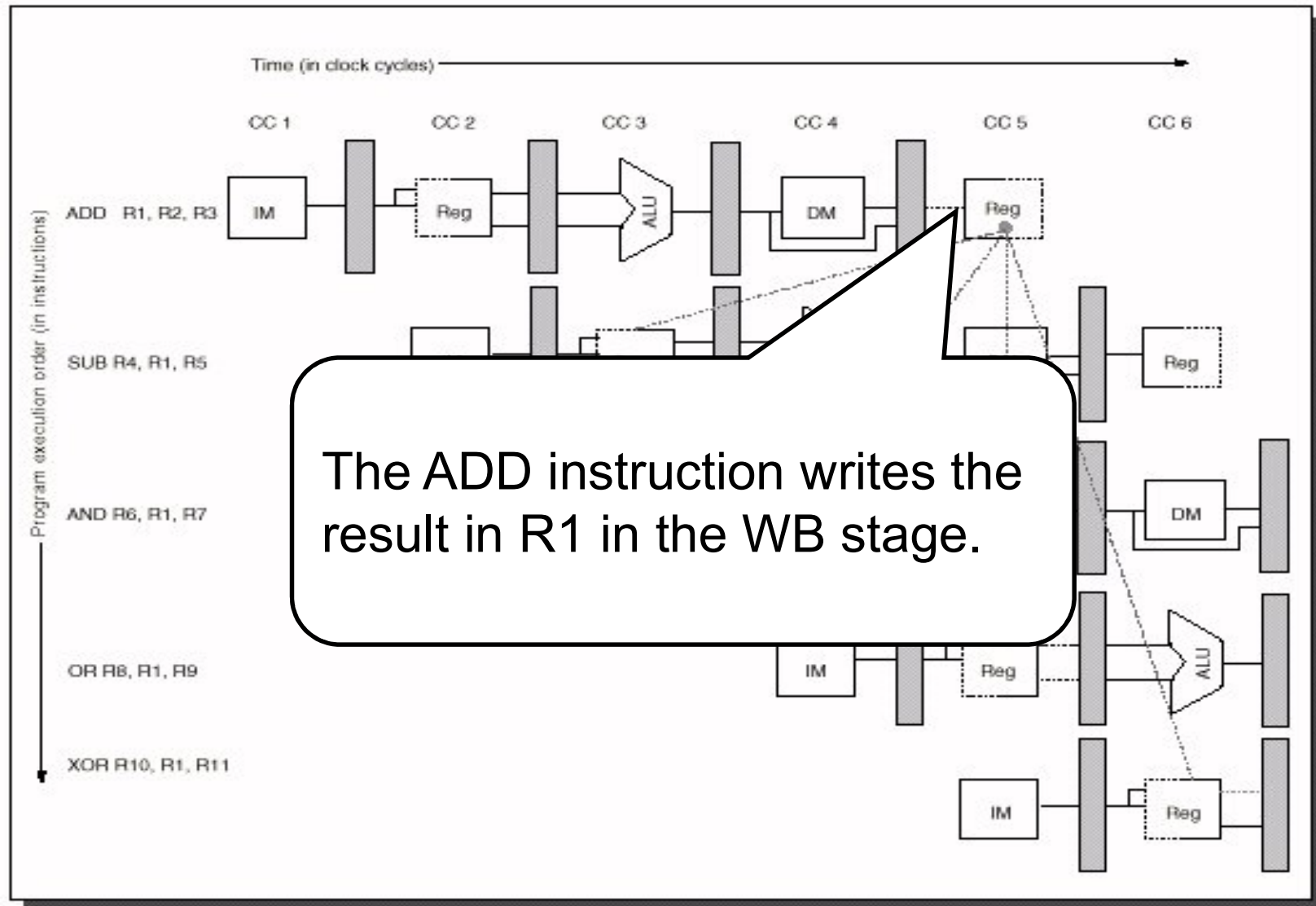
SUB R4, R1, R5

AND R6, R1, R7

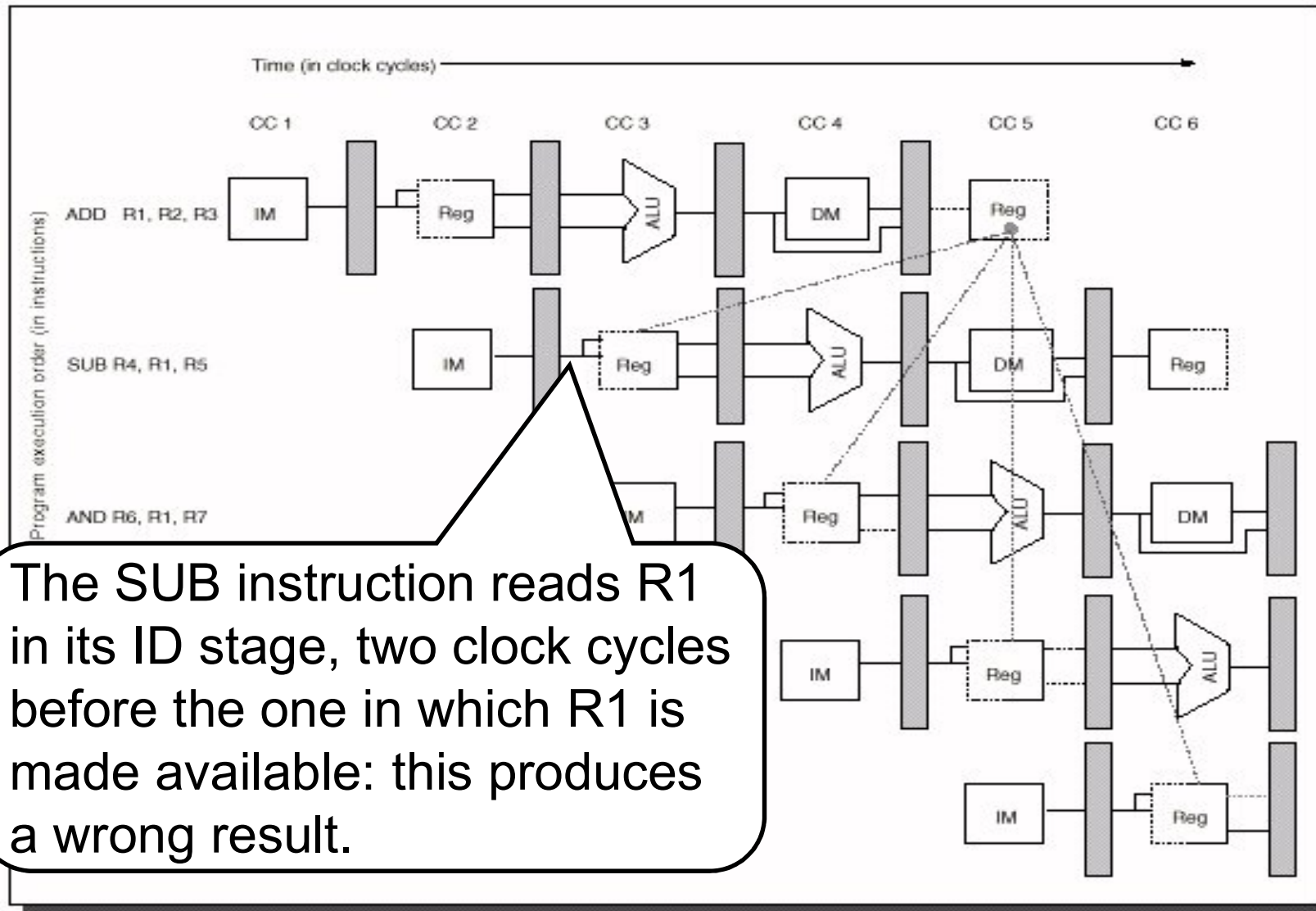
OR R8, R1, R9

XOR R10, R1, R11

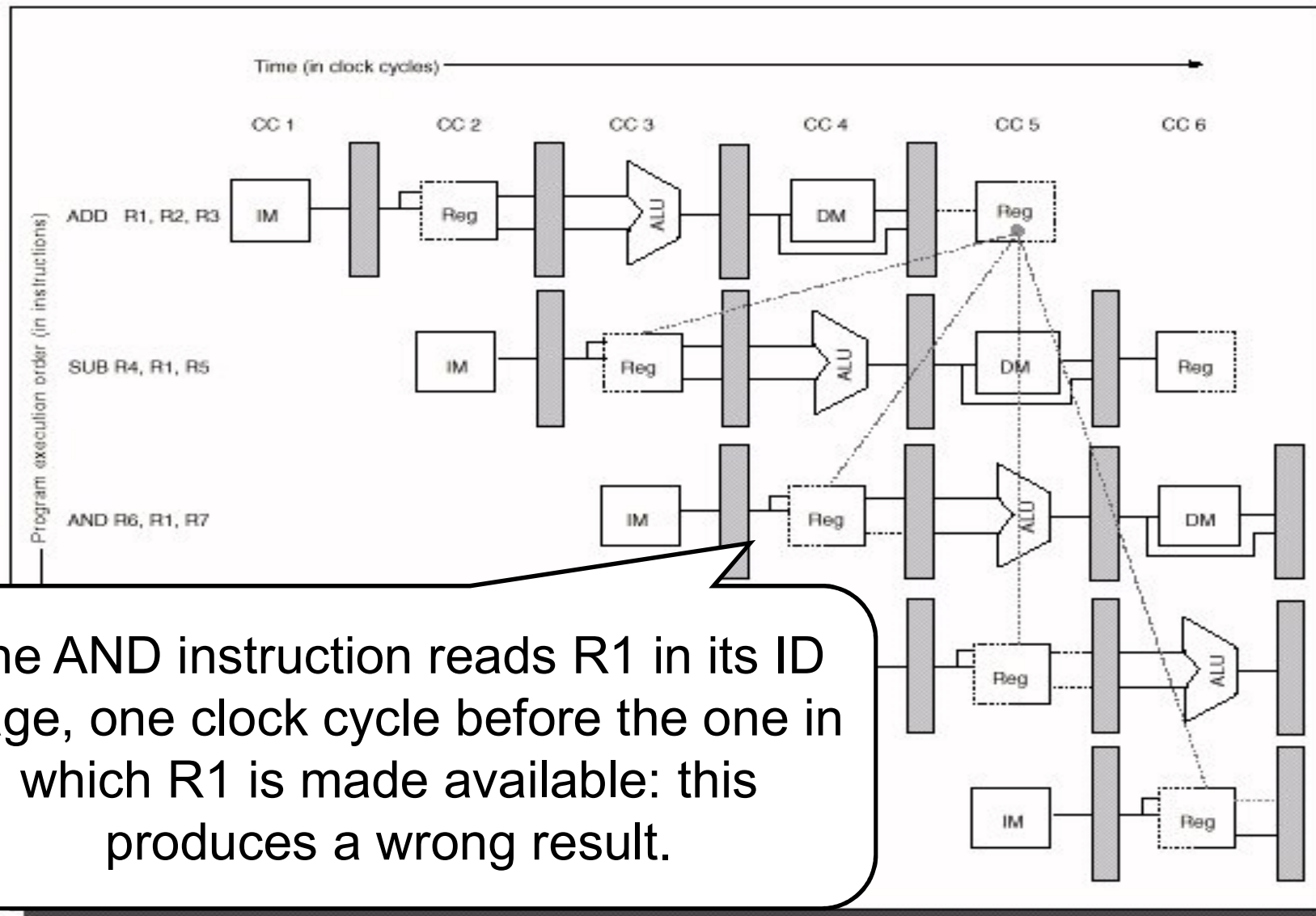
## Example (cont'd)



## Example (cont'd)

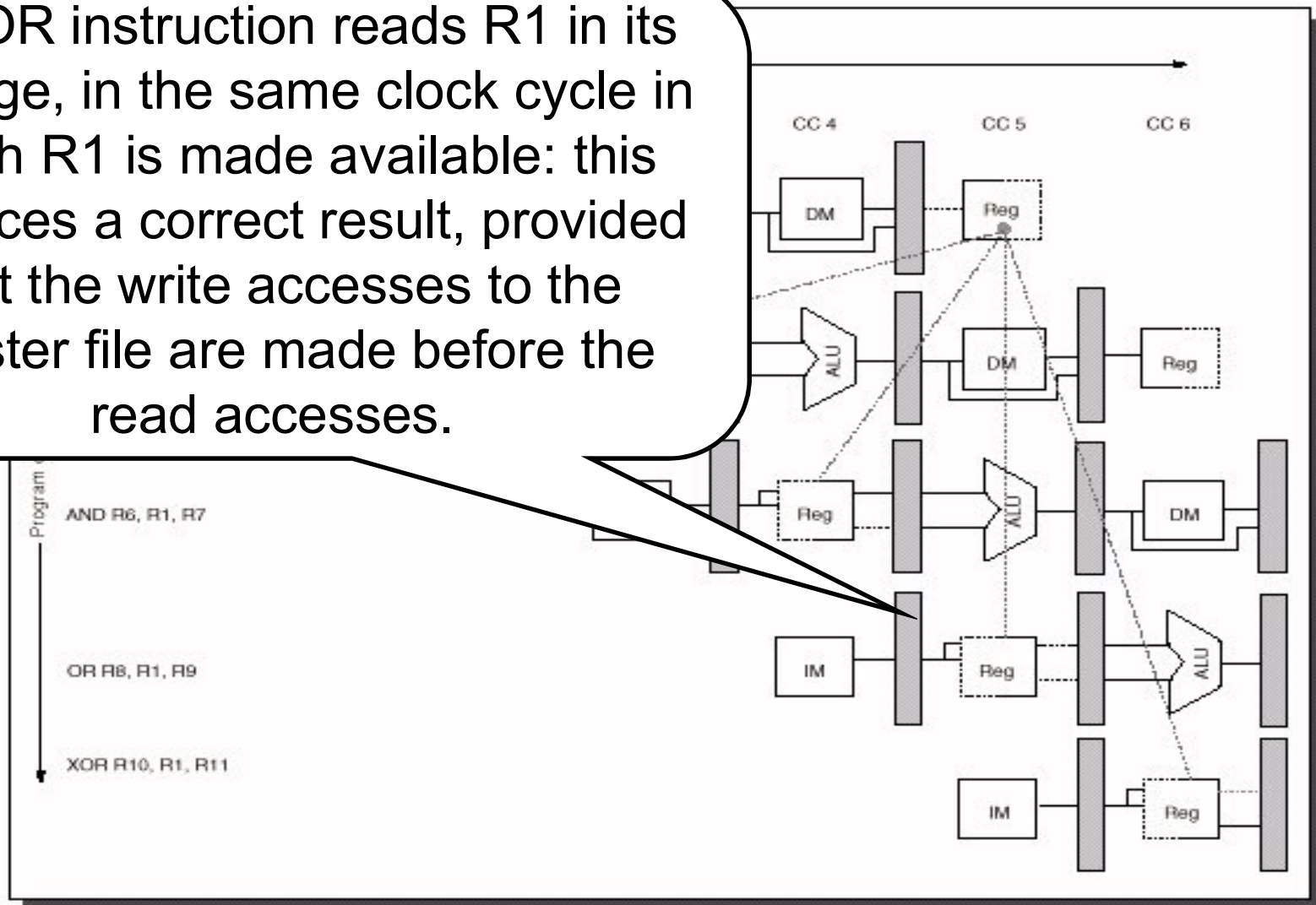


## Example (cont'd)



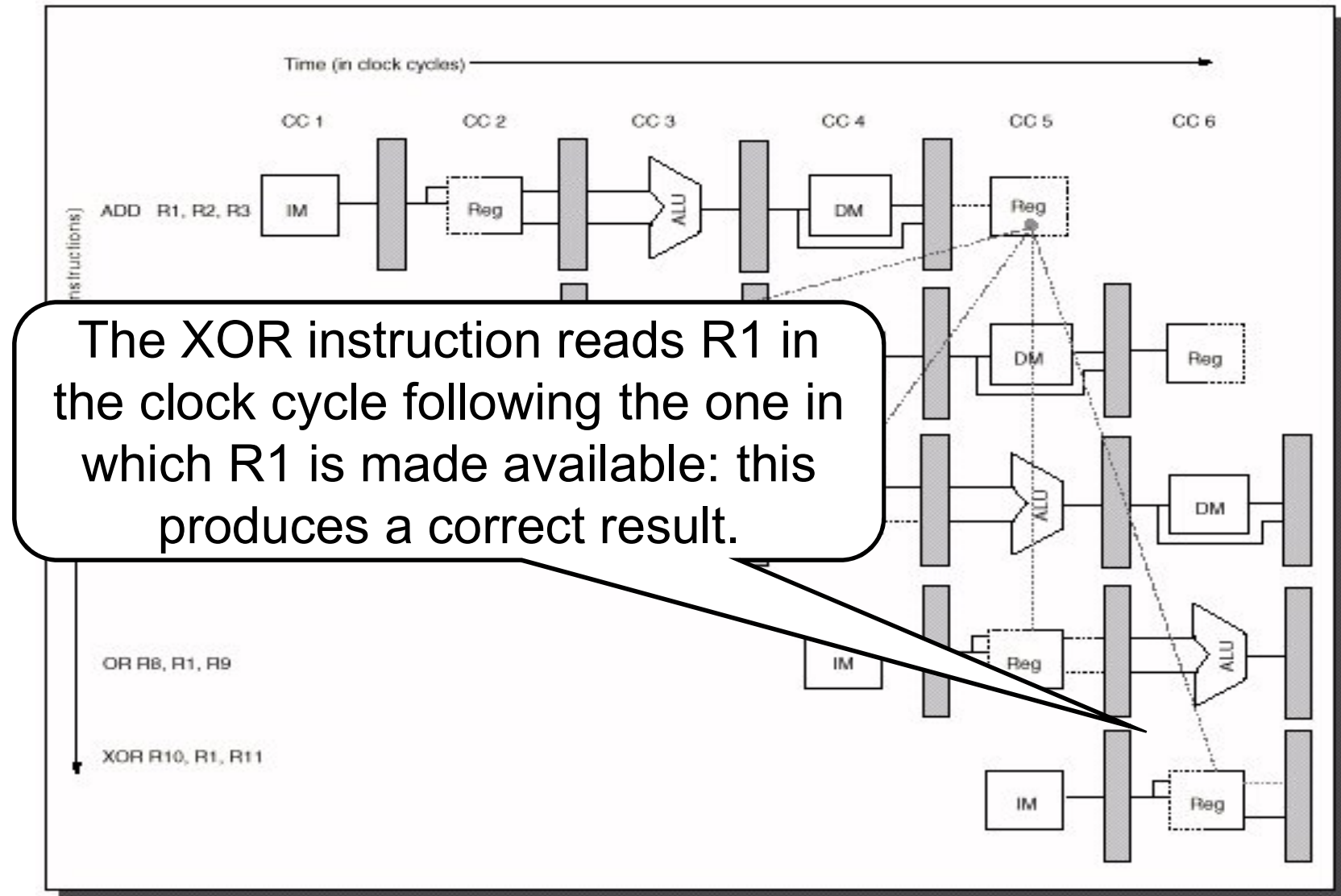
## Example (cont'd)

The OR instruction reads R1 in its ID stage, in the same clock cycle in which R1 is made available: this produces a correct result, provided that the write accesses to the register file are made before the read accesses.





## Example (cont'd)



# Interrupt effects

- If an interrupt occurs during the execution of a critical piece of code (from the point of view of data hazards) correctness may be lost or not, depending on the cases.
- This may cause an undeterministic behavior.

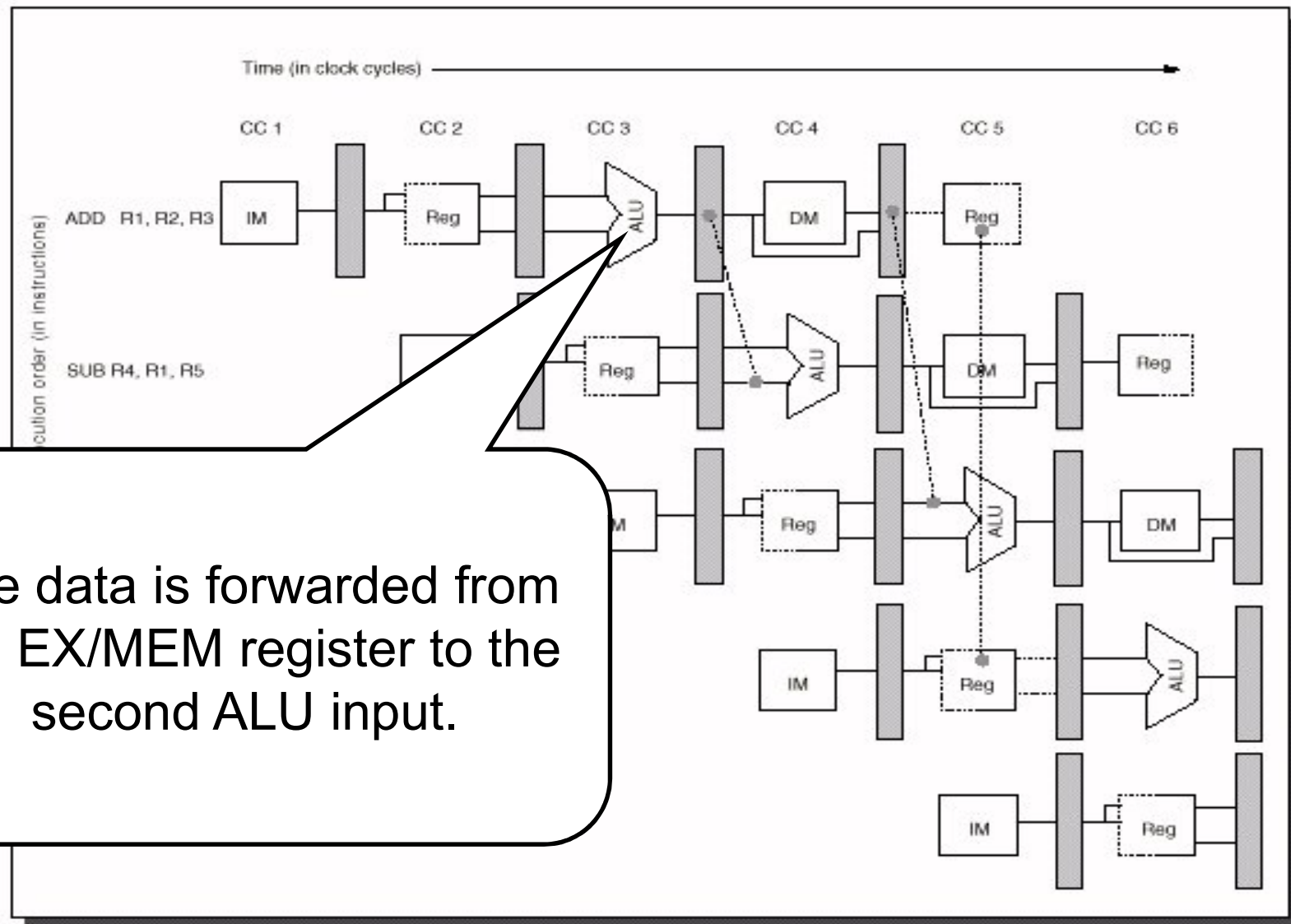
# Overcoming data hazards effects

- The wrong results produced by data hazards can be avoided:
  - by implementing a forwarding (or bypassing) technique
  - by stalling the instructions requiring the data until they are available.

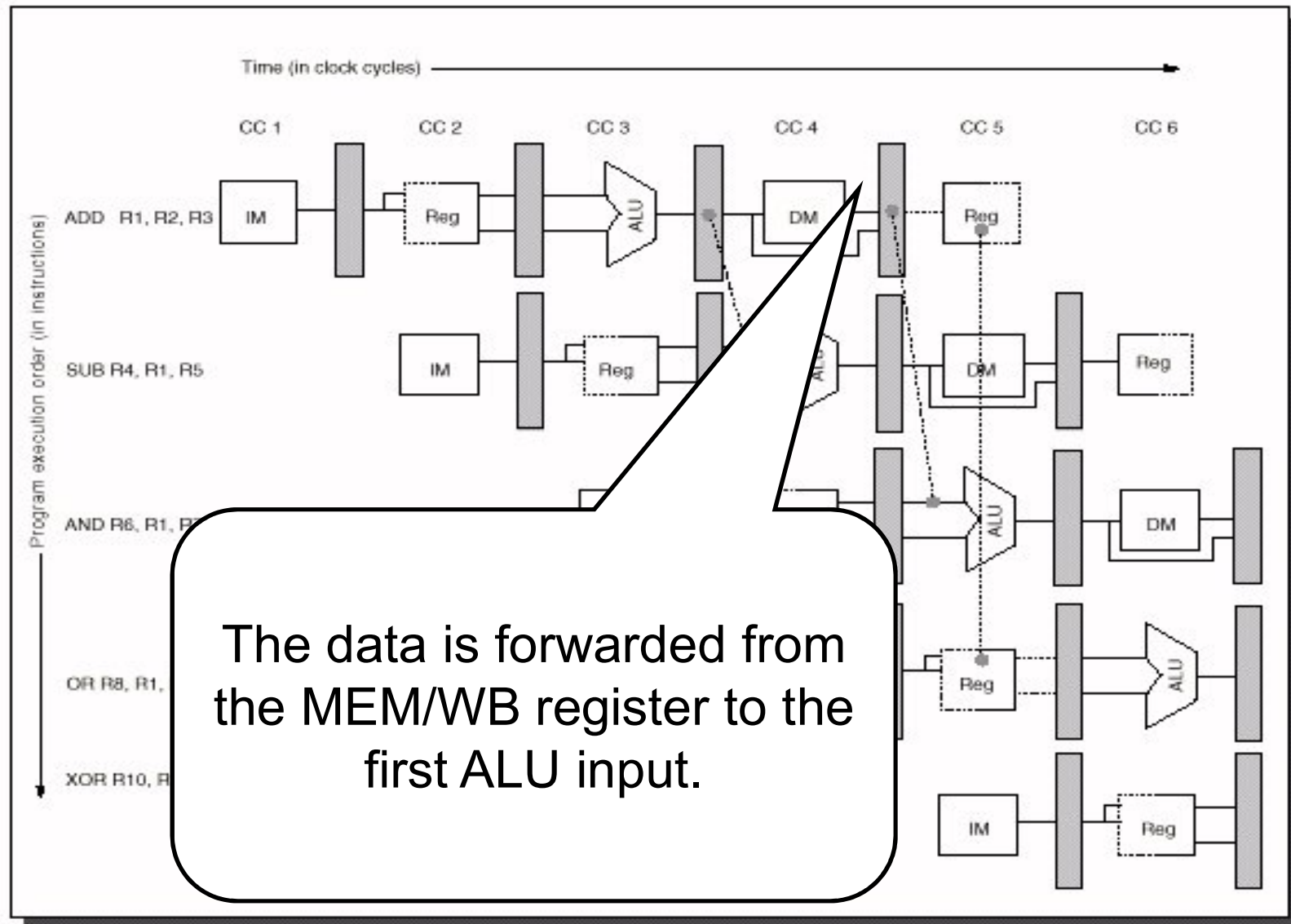
# Forwarding

- A special hardware in the datapath detects when a previous ALU operation should write the register corresponding to the source of the current ALU operation.
- In this case, the hardware selects the ALU result as the ALU input rather than the value from the register file.
- The hardware must be able
  - to forward a data from any of the previously started instructions (provided that they didn't already write the data in its final location)
  - not to forward anything, if the following instruction is stalled, or an interrupt occurred.

# Example



# Example



# Generalizing the forward technique

In order to always avoid stalling, forwarding should be made possible between any pipeline register to any input of any functional unit.

## Example

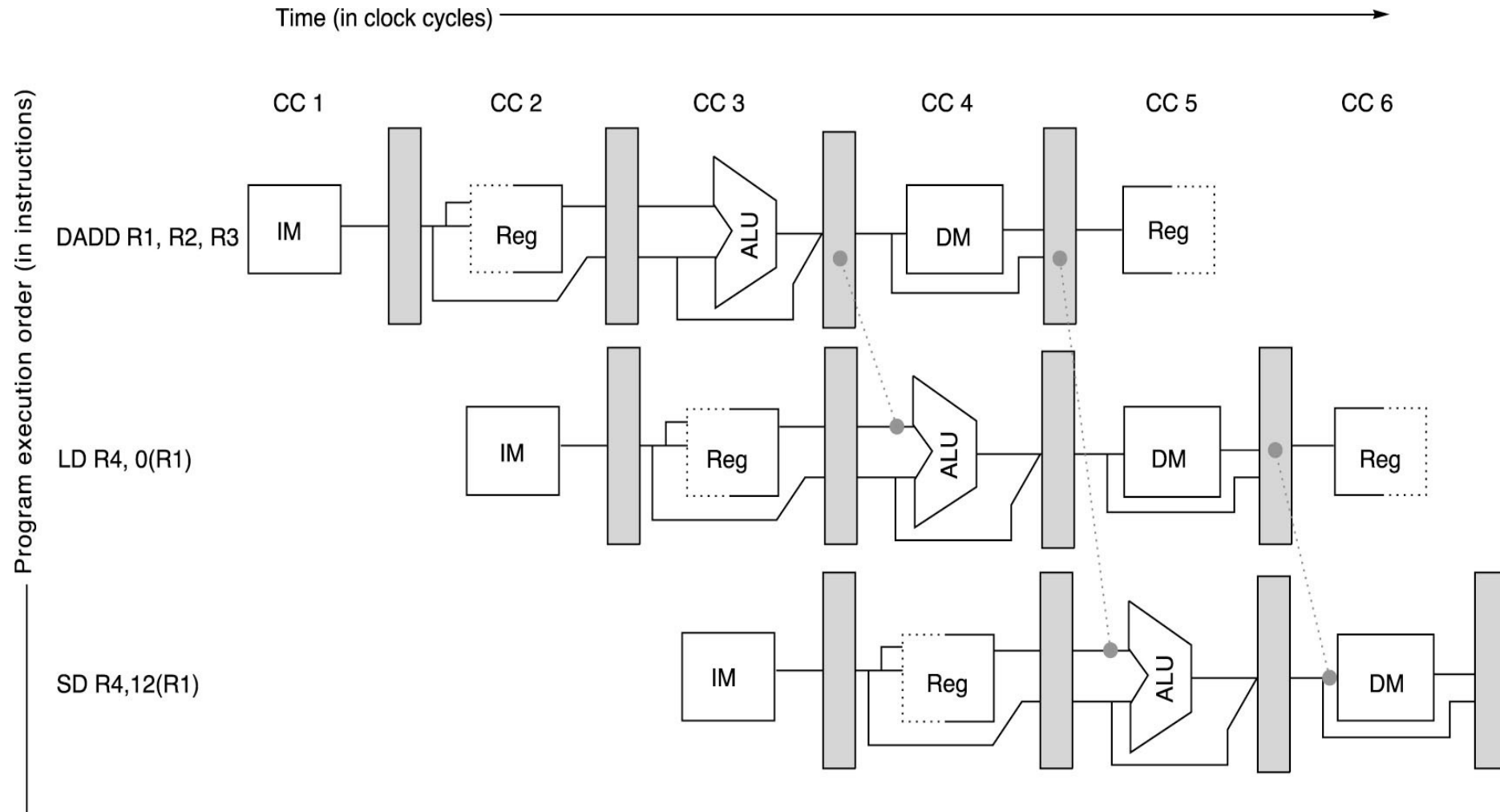
ADD R1, R2, R3

LD R4, 0(R1)

SD R4, 12(R1)

Forwarding must occur to ALU and data memory inputs.

# Example





# Causes of Data Hazards

- A hazard is created whenever there is dependence between instructions, and they are close enough that the overlap caused by pipelining would change the order of access to an operand.
- In general, this can happen for
  - register operands
  - memory operands: this is possible if
    - accesses to memory by load and store are not made in the same stage
    - execution can proceed while an instruction waits for a cache miss to be solved.

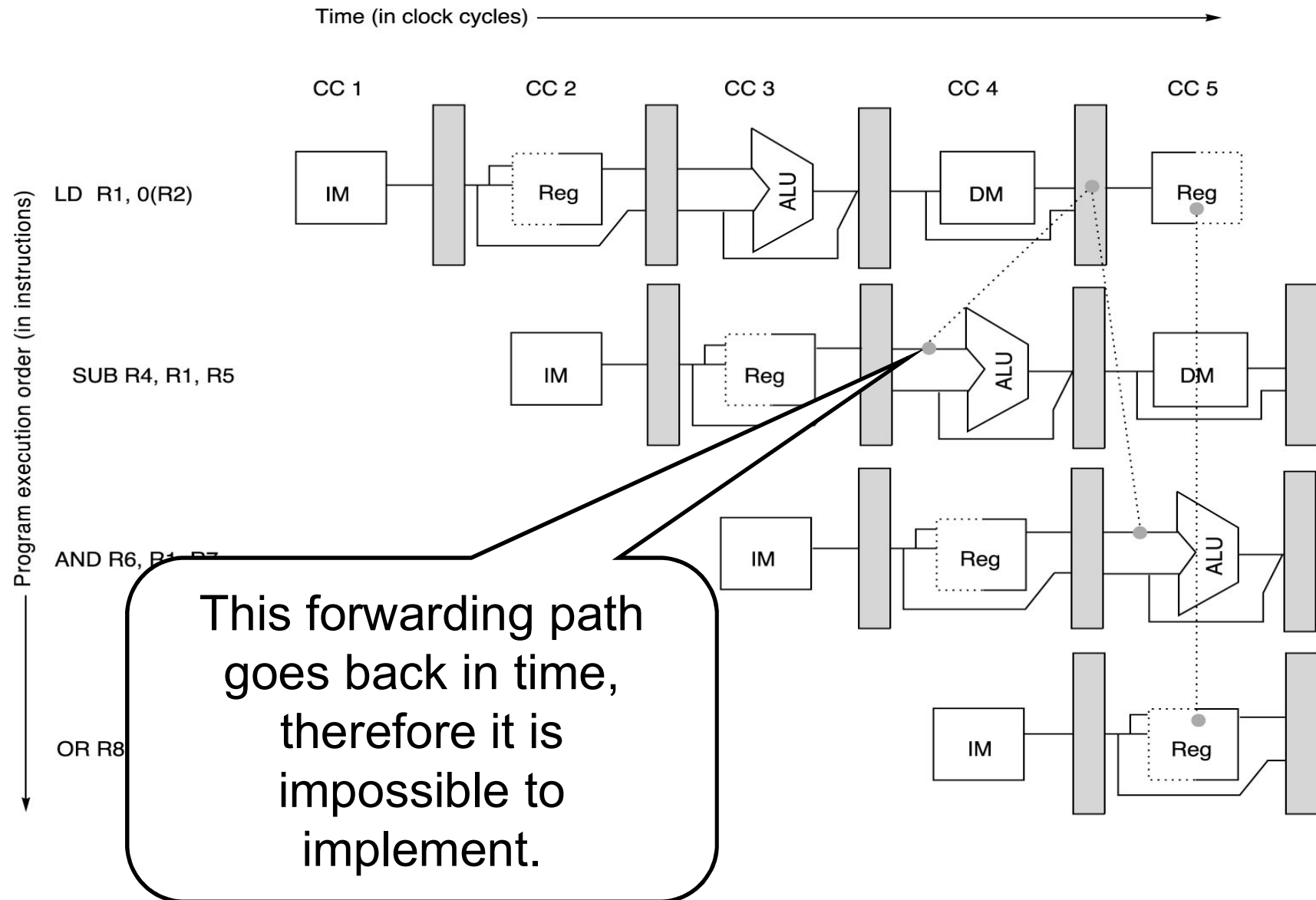
# Data Hazards Requiring Stalls

Not all potential data hazards can be solved through data forwarding.

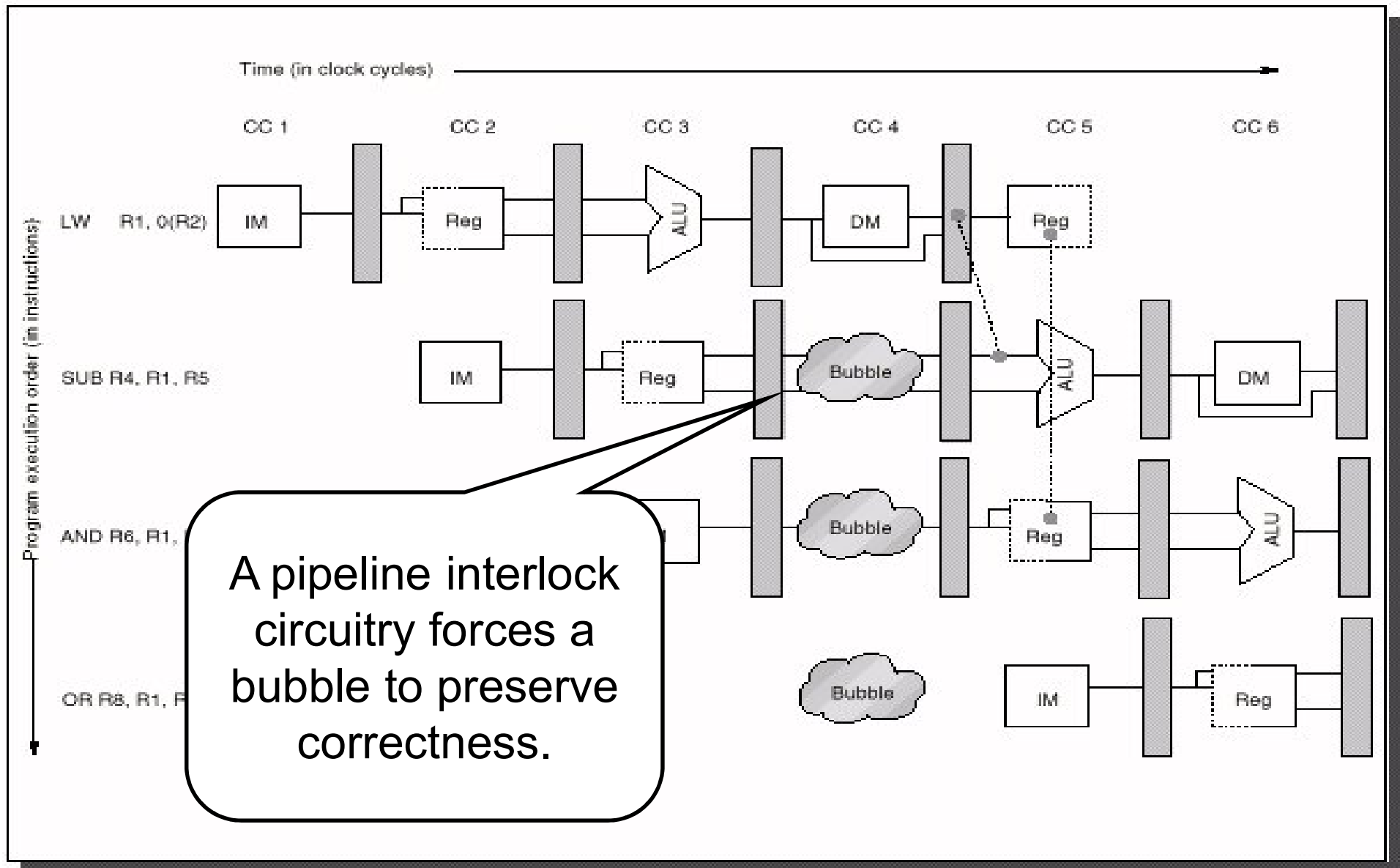
## Example

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

# Forwarding-based solution



# Solution with stall



# Implementing the control

- This requires that at each clock cycle:
  - All tests for detecting a possible data hazard concerning an instruction are performed when this is in the ID stage
  - If a data hazard is detected, two actions can be alternatively taken:
    - the appropriate forwarding is activated
    - the instruction is stalled before entering the stage where operands are not available (i.e., before being issued).

# Load Interlock Detection

Situation	Example code sequence	Action
No dependence	LD <b>R1</b> ,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R6,R7	No hazard possible because no dependence exists on R1 in the immediately following three instructions.
Dependence requiring stall	LD <b>R1</b> ,45(R2) DADD R5, <b>R1</b> ,R7 DSUB R8,R6,R7 OR R9,R6,R7	Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX.
Dependence overcome by forwarding	LD <b>R1</b> ,45(R2) DADD R5,R6,R7 DSUB R8, <b>R1</b> ,R7 OR R9,R6,R7	Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX.
Dependence with accesses in order	LD <b>R1</b> ,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9, <b>R1</b> ,R7	No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

# Load Interlock Detection (cont'd)

- The following checks have to be performed when a Load instruction is in the EX stage

Opcode field of ID/EX (ID/EX.IR <sub>0..5</sub> )	Opcode field of IF/ID (IF/ID.IR <sub>0..5</sub> )	Matching operand fields
Load	Register-register ALU	ID/EX.IR[rt] == IF/ID.IR[rs]
Load	Register-register ALU	ID/EX.IR[rt] == IF/ID.IR[rt]
Load	Load, store, ALU immediate, or branch	ID/EX.IR[rt] == IF/ID.IR[rs]

If operands match, a data hazard is detected and as a result, the control unit must insert a pipeline stall and prevent the instructions in IF and ID stages for advancing.

# Introducing a stall

- Introducing a stall in the EX stage can be done in one of the following ways:
  - forcing all 0s in the control portion of the ID/EX pipeline register (corresponding to a nop instruction)
  - forcing the IF/ID pipeline register to maintain the current value.



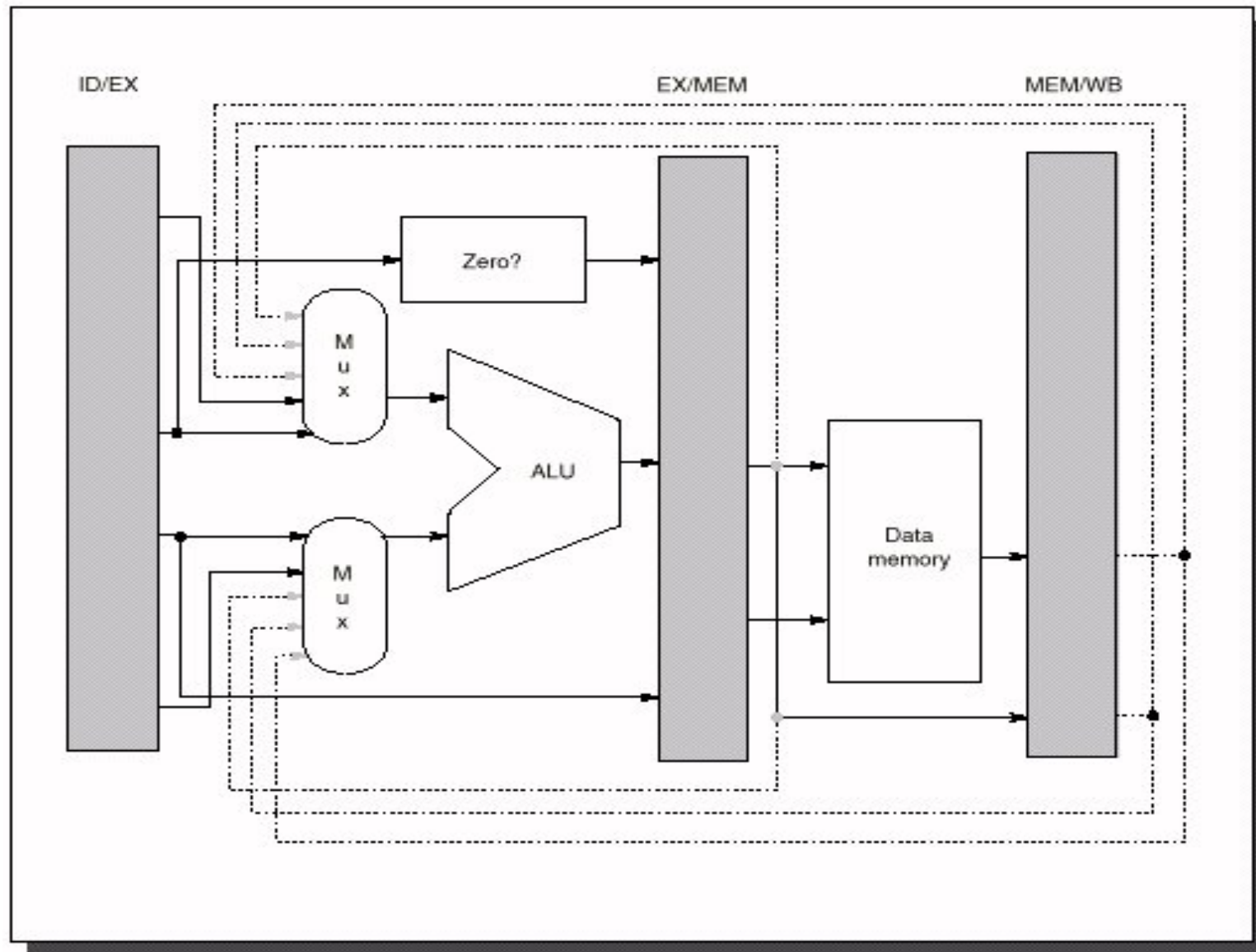
# Forwarding Logic

- Forwarding can be implemented
  - from the ALU or data memory output
  - to ALU inputs, data memory inputs, or the zero detection unit (branch instructions).
- The forwarding logic must compare:
  - the destination fields of the IR contained in the EX/MEM and MEM/WB registers with
  - the source fields of the IR contained in the IF/ID, ID/EX and EX/MEM registers.

# Forwarding to the ALU inputs

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 <sub>16..20</sub> = ID/EX.IR <sub>6..10</sub>
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR <sub>16..20</sub> = ID/EX.IR <sub>11..15</sub>
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR <sub>16..20</sub> = ID/EX.IR <sub>6..10</sub>
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR <sub>16..20</sub> = ID/EX.IR <sub>11..15</sub>
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR <sub>11..15</sub> = ID/EX.IR <sub>6..10</sub>
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR <sub>11..15</sub> = ID/EX.IR <sub>11..15</sub>
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR <sub>11..15</sub> = ID/EX.IR <sub>6..10</sub>
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR <sub>11..15</sub> = ID/EX.IR <sub>11..15</sub>
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR <sub>11..15</sub> = ID/EX.IR <sub>6..10</sub>
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR <sub>11..15</sub> = ID/EX.IR <sub>11..15</sub>

# Hardware changes to support forwarding to ALU inputs



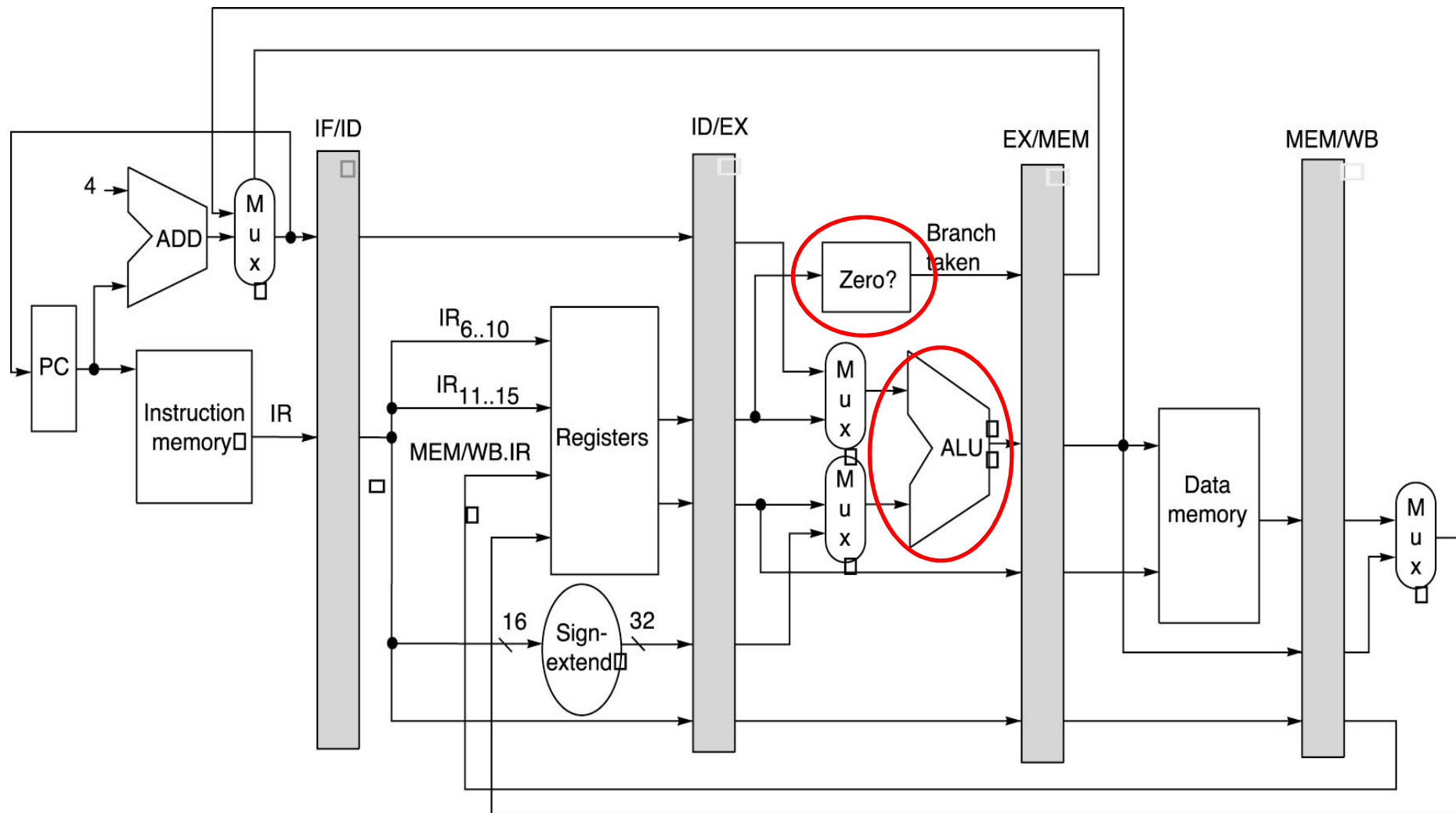
# Control hazards

- They are due to branches (conditional and unconditional), which may change the PC after the following instruction has been fetched already.
- In the case of conditional branches, the decision on whether the PC should be modified (branch taken) or not (branch untaken) can be taken even later.
- In the considered implementation, the PC is written with the target address (if the jump is taken) at the end of the ID stage.

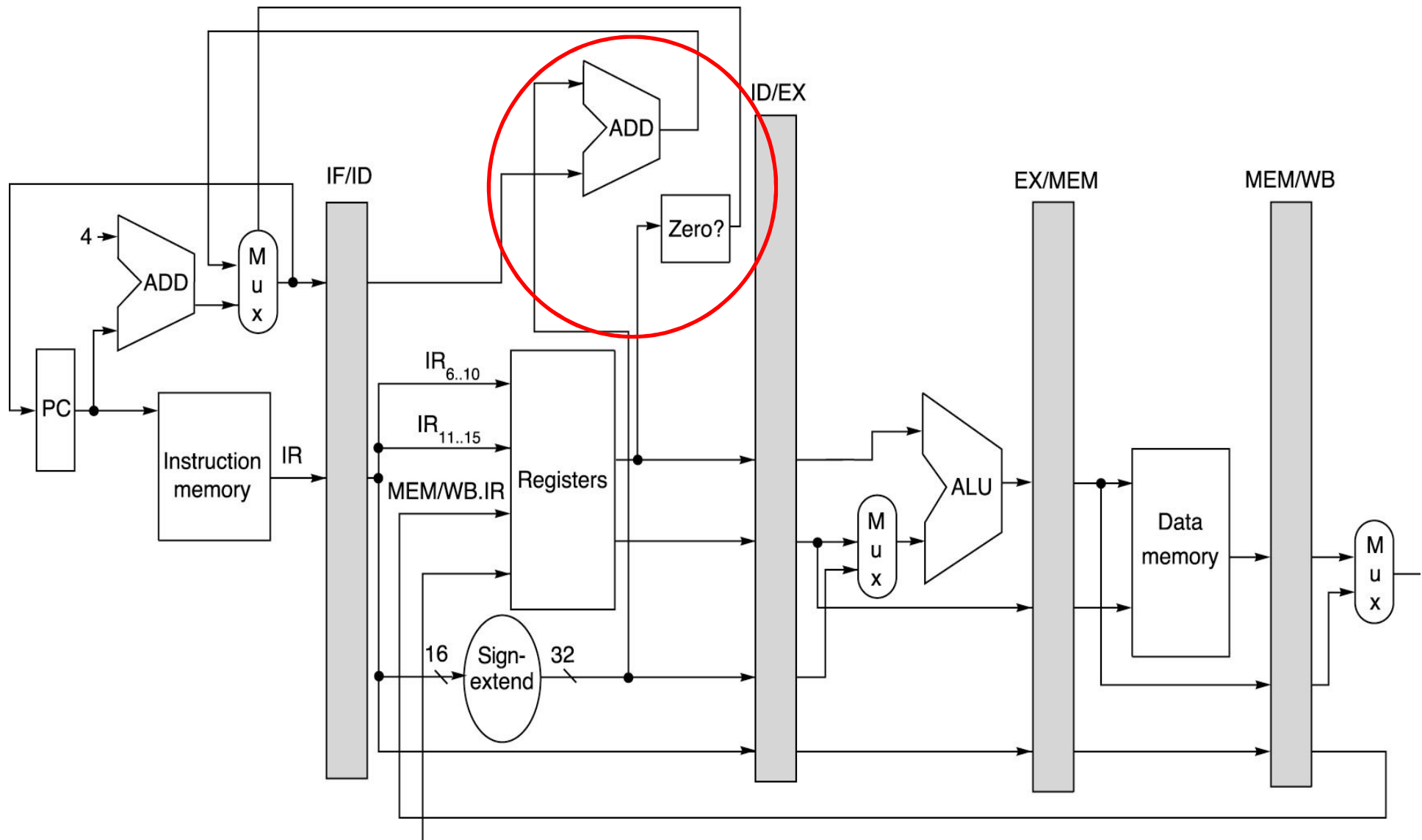
# Basic solution

- A possible solution is based on stalling the pipeline as soon as a branch instruction is detected (ID stage) by:
  - decide earlier whether the branch has to be taken or not
  - compute earlier the new PC value.

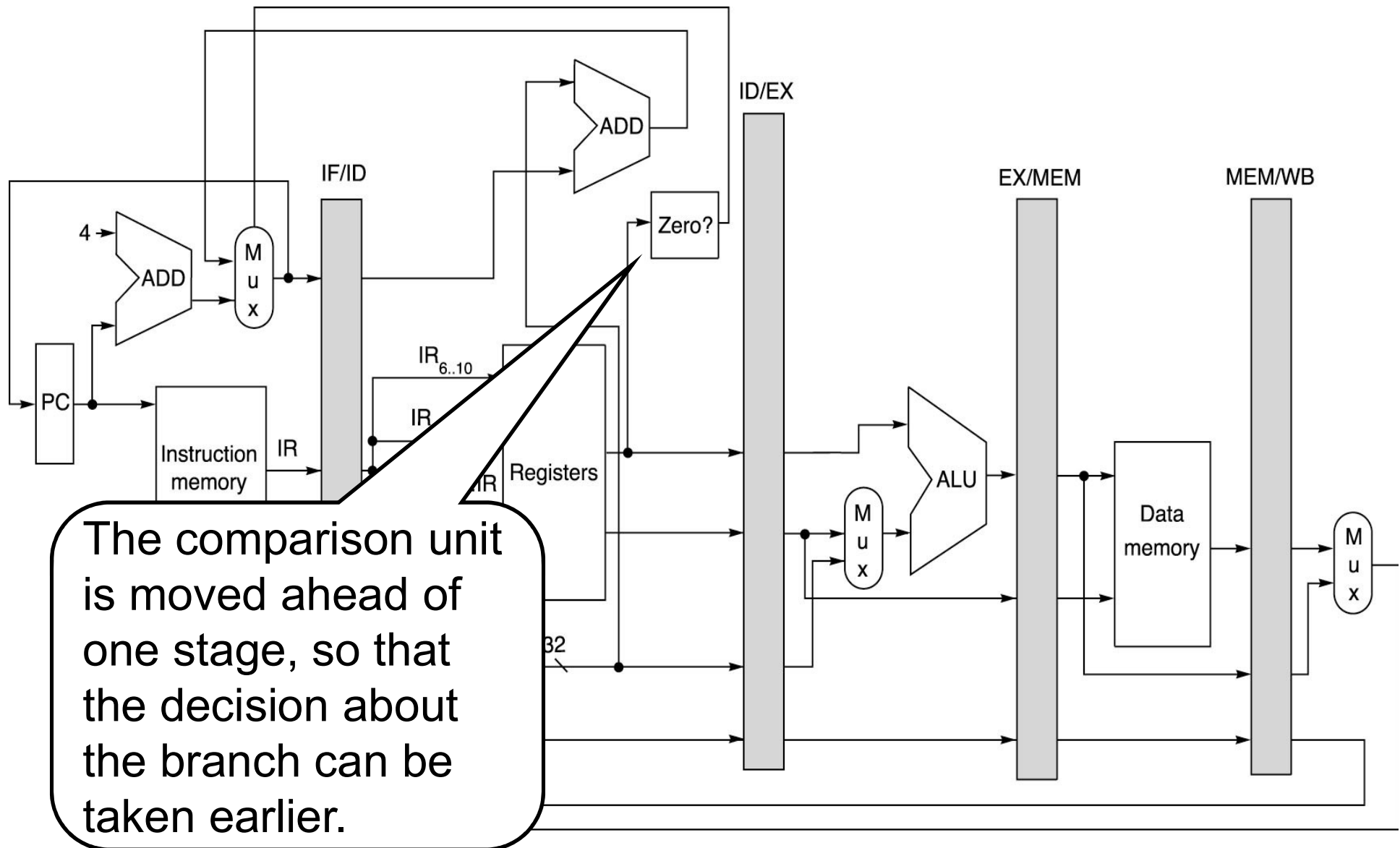
# Basic pipelined data path



# Modified pipeline

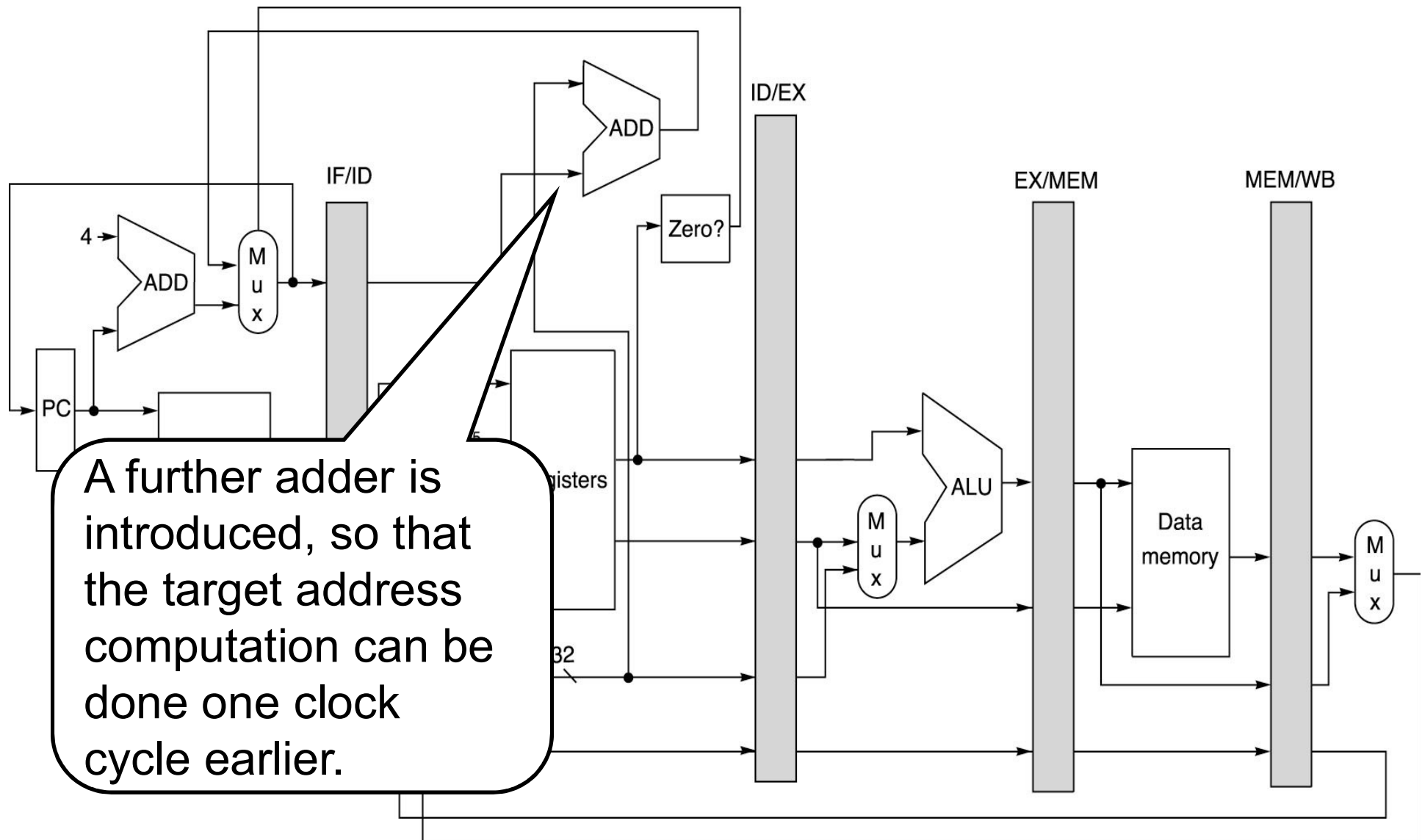


# Modified pipeline





# Modified pipeline



# Example

<b>Branch instruction</b>	<b>IF</b>	<b>ID</b>	<b>EX</b>	<b>MEM</b>	<b>WB</b>		
<b>Branch successor</b>		<b>IF</b>	<b>IF</b>	<b>ID</b>	<b>EX</b>	<b>MEM</b>	<b>WB</b>
<b>Branch successor+1</b>				<b>IF</b>	<b>ID</b>	<b>EX</b>	<b>MEM</b>
<b>Branch successor+2</b>					<b>IF</b>	<b>ID</b>	<b>EX</b>

# Example

<b>Branch instruction</b>	<b>IF</b>	<b>ID</b>	<b>EX</b>	<b>MEM</b>	<b>WB</b>		
<b>Branch successor</b>		<b>IF</b>	<b>IF</b>	<b>ID</b>	<b>EX</b>	<b>MEM</b>	<b>WB</b>
<b>Branch successor+1</b>				<b>IF</b>	<b>ID</b>	<b>EX</b>	<b>MEM</b>
<b>Branch successor+2</b>					<b>IF</b>	<b>ID</b>	<b>EX</b>

In this clock period the fetch stage fetches the following instruction (as if the branch is not taken).

# Example

**Branch instruction**

**Branch successor**

**Branch successor+1**

**Branch successor+2**

<b>IF</b>	<b>ID</b>	<b>EX</b>	<b>MEM</b>	<b>WB</b>			
	<b>IF</b>	<b>IF</b>	<b>ID</b>	<b>EX</b>	<b>MEM</b>	<b>WB</b>	
			<b>IF</b>	<b>ID</b>	<b>EX</b>	<b>MEM</b>	
				<b>IF</b>	<b>ID</b>	<b>EX</b>	

In this clock period the fetch stage fetches the right instruction (which depends on the branch result).

# Example

<b>Branch instruction</b>	<b>IF</b>	<b>ID</b>	<b>EX</b>	<b>MEM</b>	<b>WB</b>		
<b>Branch successor</b>		<b>IF</b>	<b>IF</b>	<b>ID</b>	<b>EX</b>	<b>MEM</b>	<b>WB</b>
<b>Branch successor+1</b>				<b>IF</b>	<b>ID</b>	<b>EX</b>	<b>MEM</b>
<b>Branch successor+2</b>					<b>IF</b>	<b>ID</b>	<b>EX</b>



This operation is  
ALWAYS useless.

# Improved solutions

- There are several techniques for reducing the performance degradation due to branches:
  - freezing the pipeline
  - predict untaken
  - predict taken
  - delayed branch.

# Freezing the pipeline

- It is the previously proposed solution: the pipeline is stalled (or flushed) as soon as a branch instruction is detected, and until the decision about the branch is known.
- It is the simplest solution to implement.

# Predict untaken

- This technique
  - assumes the branch is not taken
  - avoid any change in the pipeline status until the branch decision has been taken
  - undo all the performed operations if the branch turns out to be taken.



# Predict untaken

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB
Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

# Predict untaken

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	idle	idle	idle	idle			
Branch instruction			IF	ID	EX	MEM	WB		
Branch instruction				IF	ID	EX	MEM	WB	
Branch instruction					IF	ID	EX	MEM	WB

This result can also be obtained by turning the already fetched instruction into a nop.

# Predict untaken

Untaken branch instruction	IF	ID	EX	MEM	WB		
Instruction $i + 1$		IF	ID	EX	MEM	WB	
Instruction $i + 2$			IF	ID	EX	MEM	WB
Instruction $i + 3$							
Instruction $i + 4$							WB
Taken branch instruction	IF	ID	EX	MEM	WB		
Instruction $i + 1$		IF	ID	EX	MEM	WB	
Branch target							
Branch target + 1							
Branch target + 2							WB

The cost for the branch instruction is different depending whether the branch is taken or not.

The cost for the branch instruction is different depending whether the branch is taken or not.

# Predict taken

- If the target address is known before the branch outcome, it may be possible to assume the branch as taken.

# Compiler role

If the hardware supports the predict taken or predict untaken scheme, the compiler can improve performance by generating code which maximizes the chance for the processor to make the right prediction.

## Example

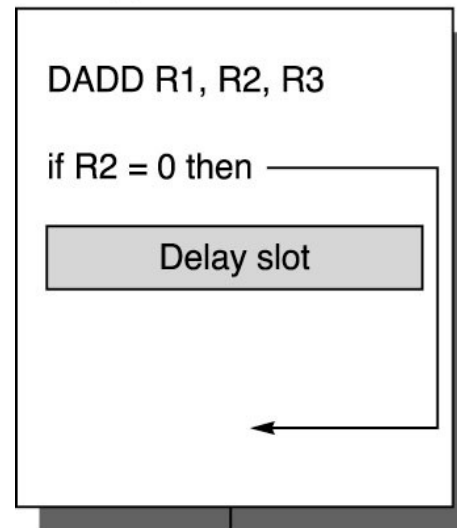
Considering the loop implementation, the for scheme is suitable for the predict untaken scheme, the do while for the predict taken.

# Delayed branch

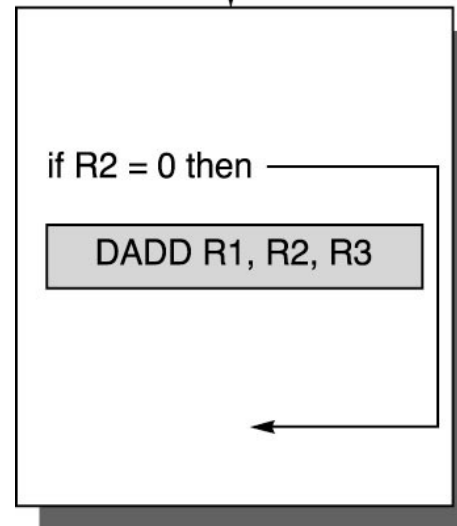
- This technique is based on filling the slot after the branch instruction (named *branch-delay slot*) with instructions which have to be executed no matter the branch outcome.
- It is up to the compiler to fill each branch-delay slot with the right instructions.
- The processor does nothing special when a branch instruction is decoded.

# Example

(a) From before



becomes



# Delayed-branch scheduling effectiveness

- It depends on the compiler ability in finding the right instructions to put in the delay slots.
- Using this technique, only about 30% of branches do produce a penalty.



# Trend

- With the advent of deeply pipelined processors, the delay slots are becoming longer, and the advantages of delayed-branches smaller.
- Therefore, several current RISC architectures do not support any more delayed branches.

# EXCEPTIONS

- Exceptions are events that modify the normal execution order of the program.
- Exceptional events (exceptions, interrupts, or faults) are more complex to handle in pipelined architectures because several instructions are being executed at a time.

# Exception sources

- Possible causes of exceptions are:
  - I/O device request
  - Operating system call by a user program
  - Tracing instruction execution
  - Breakpoint (programmer-requested interrupt)
  - Integer arithmetic overflow or underflow
  - FP arithmetic anomaly
  - Page fault
  - Misaligned memory accesses
  - Memory-protection violation
  - Undefined instruction
  - Hardware malfunction
  - Power failure.

# Exception Classification

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskable vs. user nonmaskable
- Within vs. between instructions
- Resume vs. terminate.

# Exception Classification

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskable vs. nonmaskable
- Within vs. between instructions
- Resume vs. nonresume

An exception is synchronous if it can be triggered always at the same position in the code. Asynchronous exceptions are normally generated by external devices.

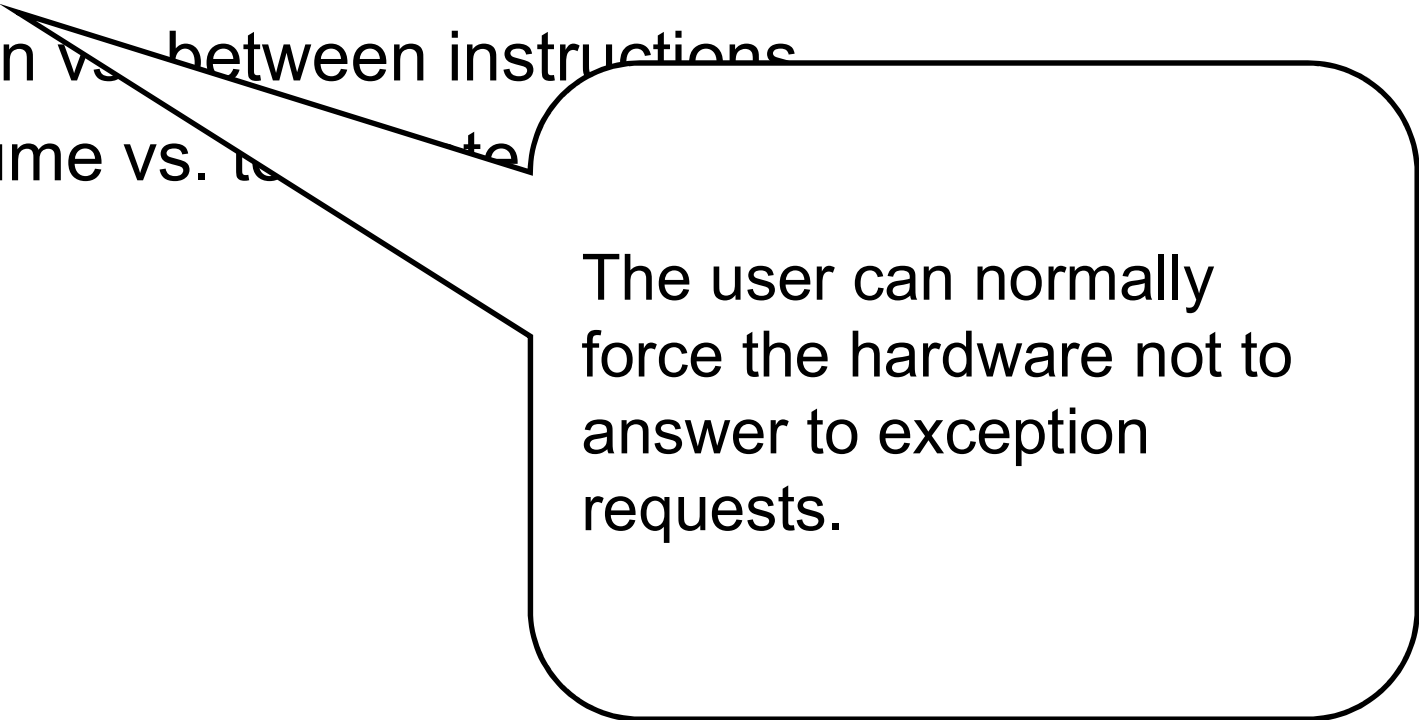
# Exception Classification

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskable vs. user nonmaskable
- Within vs. between instructions
- Resume

User requested exceptions are similar to procedures.  
Coerced exceptions are out of the control of the user program.

# Exception Classification

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskable vs. user nonmaskable
- Within vs. between instructions
- Resume vs. to



The user can normally force the hardware not to answer to exception requests.

# Exception Classification

- Synchronous vs. Asynchronous
- User requested vs. not requested
- User maskable vs. not maskable
- Within vs. between instructions
- Resume vs. terminate.

Exceptions can occur either within or between instructions. In the former case they are generated by the instruction itself.



# Exception Classifi

- Synchronous vs. asynch
- User requested vs. coe
- User maskable vs
- Within vs. between instruc
- Resume vs. terminate.

After activating an exception, the program can either terminate, or execute something and then resume.

# Restartable machines

- Restartable machines are able to handle an exception, save the state, and restart without affecting the execution of the program.
- Nearly all processors nowadays are restartable machines.

# Stopping the execution

- When an exception occurs, the pipeline must execute the following steps:
  - force a trap instruction into the pipeline on the next IF stage
  - until the trap is taken, turn off all writes for the instruction that raised the exception (i.e., the *faulty* instruction) and for all the following instructions in the pipeline
  - when the exception-handling procedure receives control, it immediately saves the PC of the faulty instruction.

# Restarting the execution

- After the exception has been handled, special instructions return the machine from the exception by reloading the PC and restarting the instruction stream.

# Precise exceptions

- A processor has *precise* exceptions if the pipeline can be stopped so that
  - the instructions just before the instruction that triggered the exception are completed, and
  - the instructions following the instruction that triggered the exception can be restarted from scratch.
- Restarting after an exception may be really hard if exceptions are not handled precisely.
- Precise exception handling is a must for most architectures, at least for integer instructions.

# Cost of precise exceptions

- Some processors (Alpha 21064, MIPS R800) implement precise exceptions in debug mode, only.
- This mode is about 10 times slower than usual normal mode.

# MIPS: Possible sources of Exceptions

Pipeline stage	Cause of exception
IF	Page fault on instruction fetch Misaligned memory access Memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch Misaligned memory access Memory-protection violation
WB	None

# Contemporary exceptions

## Example

LD	IF	ID	EX	MEM	WB
DADD		IF	ID	EX	MEM WB



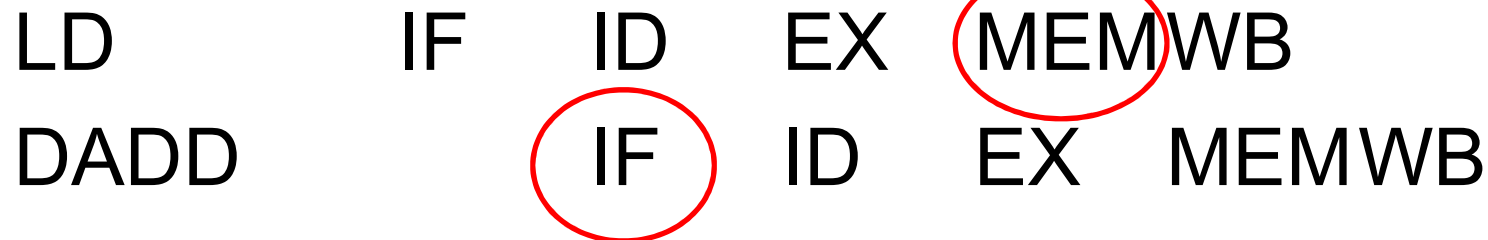
A data page fault exception can occur in the MEM stage of the LD instruction, and an arithmetic exception in the EX stage of the DADD instruction. The first exception is processed and, if its cause is removed, the second exception is handled.



# Exception order

There are cases in which two exceptions can occur in the opposite order of the instructions they relate to.

## Example

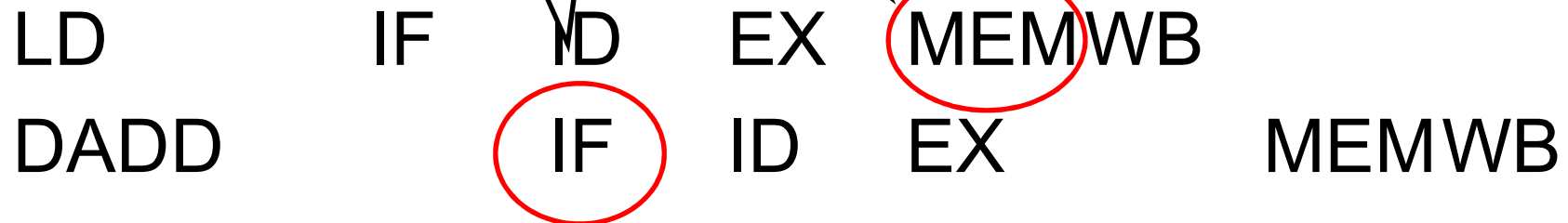


## Exception order

An exception caused by an instruction page fault in the second instruction occurs before an exception caused by a data page fault in the first instruction.

no exceptions can  
of the instructions

### Example



# Exception order (cont'd)

- A possible solution is the following:
  - a status flag is associated to each instruction in the pipeline
  - if an instruction causes an exception, the status flag is set
  - if the status flag is set, the instruction can not perform any write operation
  - when an instruction reaches the last stage, and its status flag is set, an exception is triggered.

# Instruction set complications

- When an instruction is guaranteed to complete it is called *committed*.
- Some machines have instructions that change the state before they are committed (e.g., those using autoincrement addressing modes).
- If one of these instructions is aborted because of an exception, it leaves the machine state altered.
  - Implementing precise exceptions is thus very difficult.
- A possible solution is based on allowing to roll-back all the state changes made by an instruction before it is committed.

# Instruction set complications (cont'd)

- Instructions implicitly updating condition codes create complications:
  - they can cause data hazards
  - they require to be saved and restored in the case of an exception
  - they make more difficult the task of the compiler for filling possible delay slot between the instruction writing the condition codes and the branch one.

# Instruction set complications (cont'd)

- Complex instructions are difficult to implement in a pipeline. Forcing them to have the same length can hardly be achieved.
- Sometimes these problems are solved by pipelining the microinstructions implementing each instruction.

# References

J.L. Hennessy, D.A. Patterson

*Computer Architecture: a Quantitative Approach*

Morgan Kaufmann Publishers, Inc., V Edition, 2012