

Pipelining

E. Sanchez

Politecnico di Torino
Dipartimento di Automatica e Informatica

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

I₁

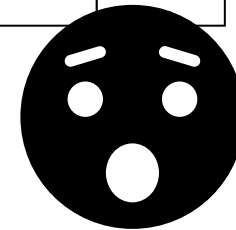
F	D	E	W
---	---	---	---

I₂

F	D	E	W
---	---	---	---

I₃

F	D	E	W
---	---	---	---



I₄

F	D	E	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 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

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

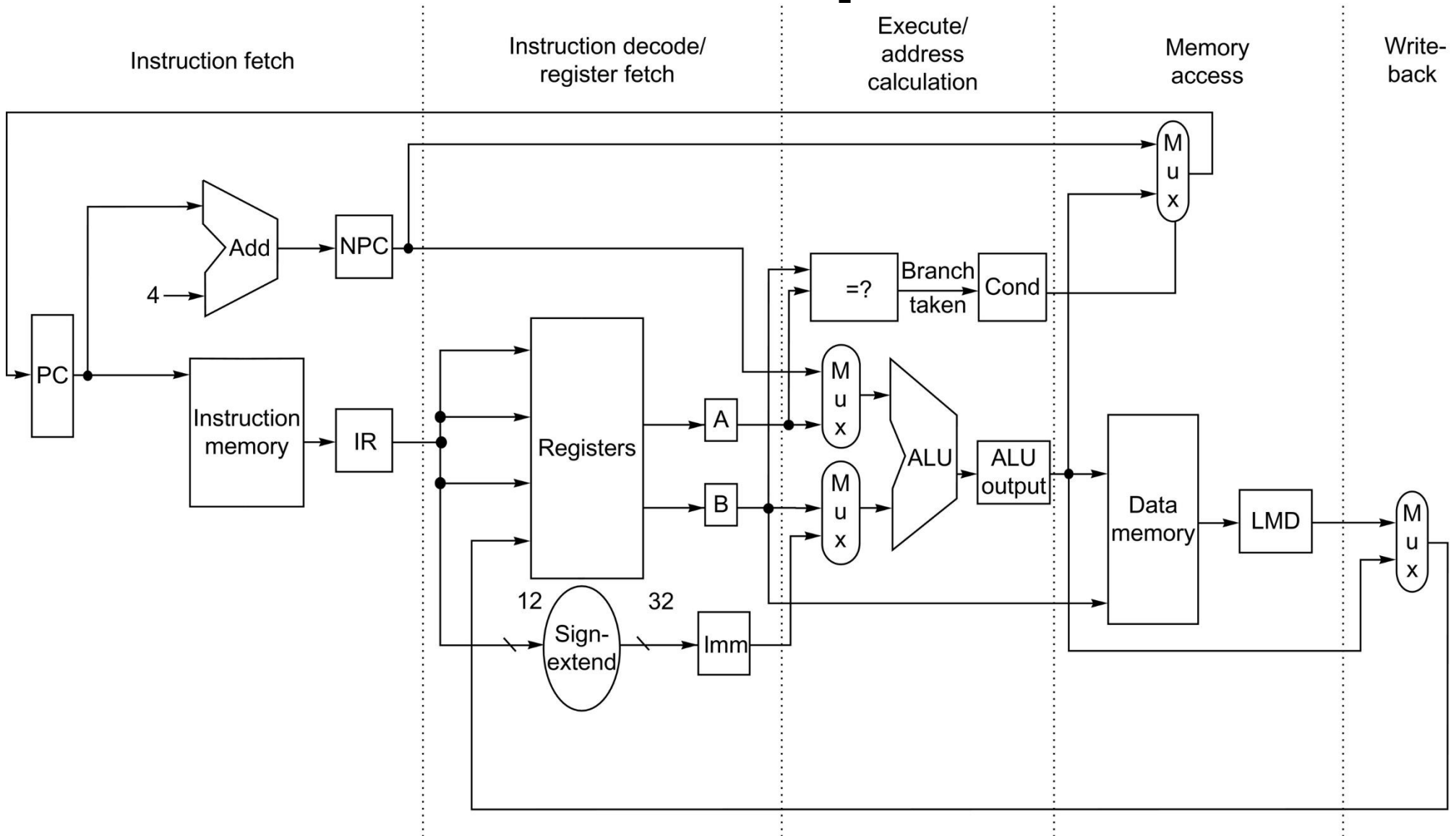
being n the number of pipeline stages.

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).**

The Datapath

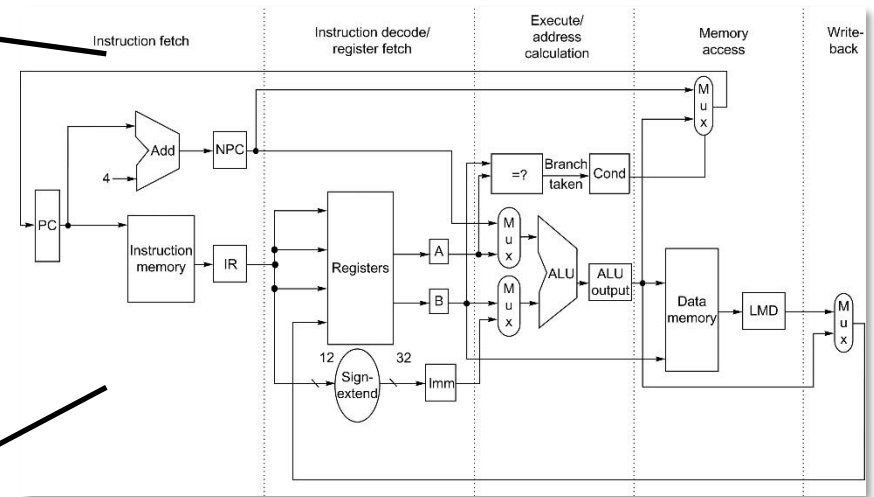
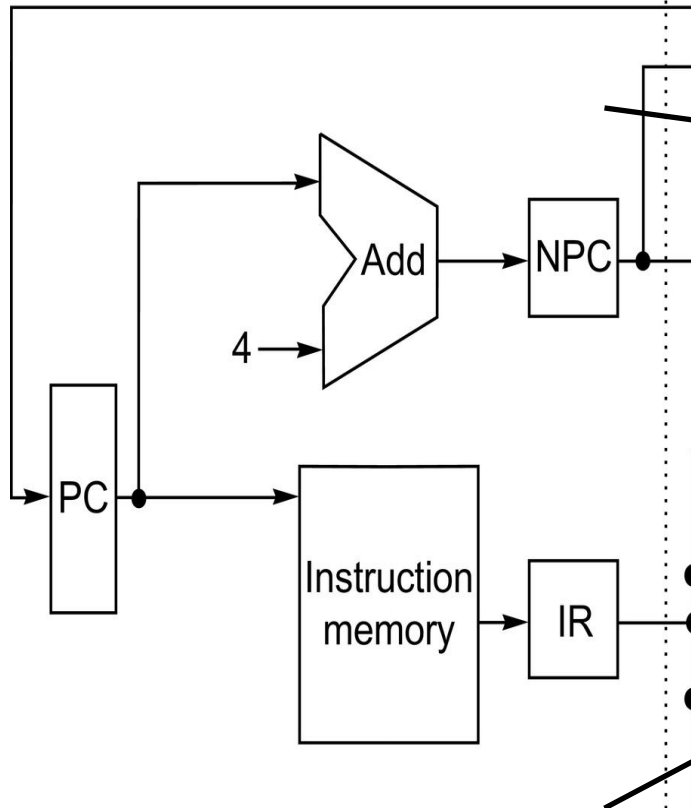


Instruction Fetch cycle

Instruction fetch

$IR \leftarrow Mem[PC]$

$NPC \leftarrow PC + 4$

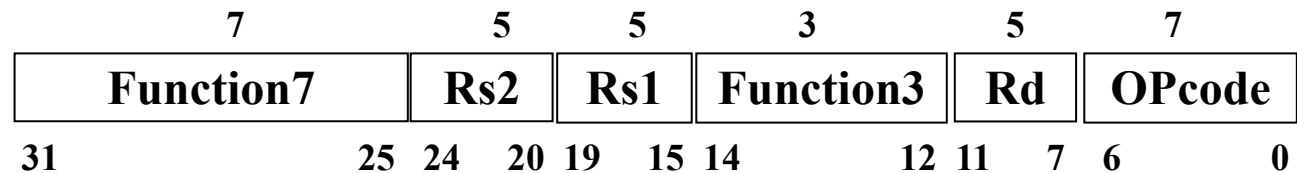


Instruction Decode/ Register Fetch

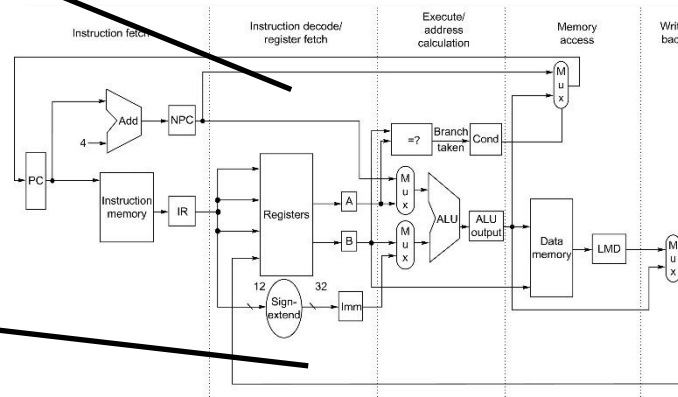
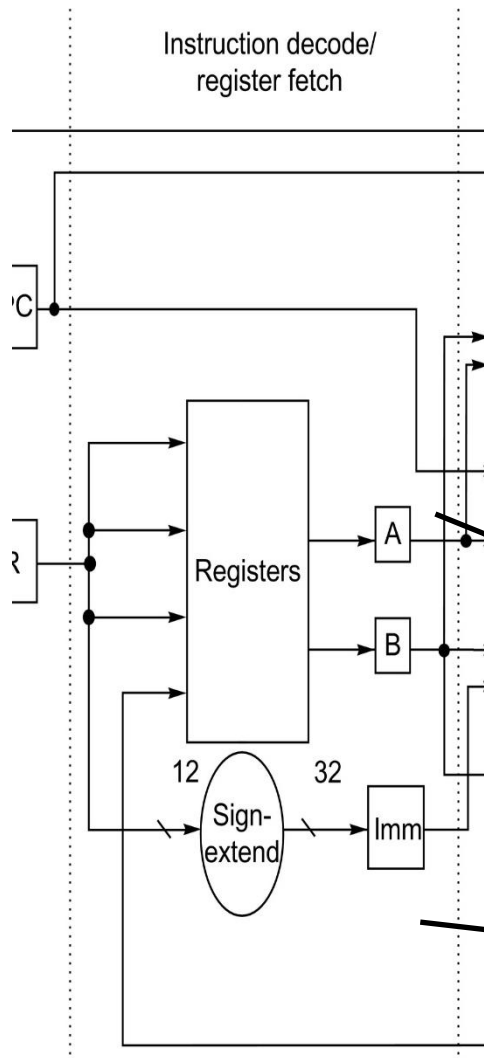
A ← Regs [IR_{19..15}]

B ← Regs [IR_{24..20}]

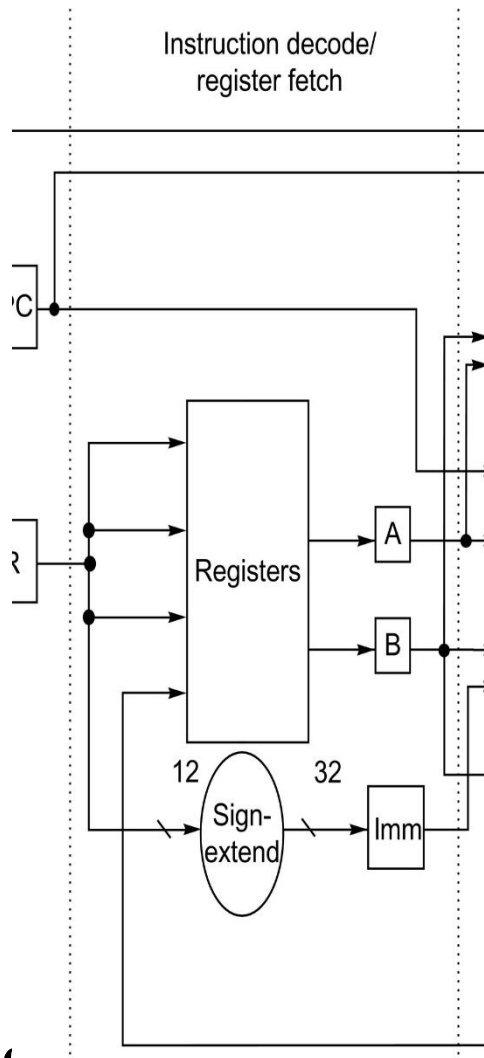
$$\text{Imm} \leftarrow (([\text{IR}]_{31})^{20} \#\#\text{IR}_{31..20})$$



R-type

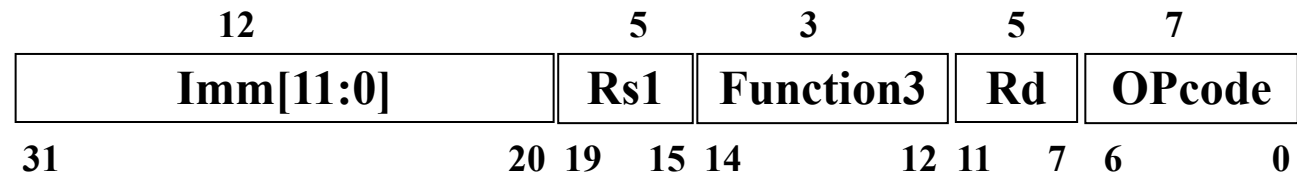


Instruction Decode/ Register Fetch



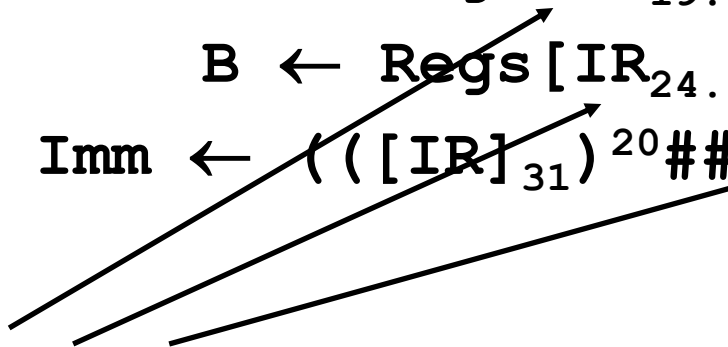
$$A \leftarrow \text{Regs}[\text{IR}_{19..15}]$$

$$B \leftarrow \text{Regs}[\text{IR}_{24..20}]$$

$$\text{Imm} \leftarrow (([\text{IR}]_{31})^{20} \# \text{IR}_{31..20})$$


I-type

Instruction Decode/ Register Fetch

$$\begin{aligned} A &\leftarrow \text{Regs}[\text{IR}_{19..15}] \\ B &\leftarrow \text{Regs}[\text{IR}_{24..20}] \\ \text{Imm} &\leftarrow (([\text{IR}]_{31})^{20} \#\#\text{IR}_{31..20}) \end{aligned}$$


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

Execution/Effective Address Cycle

- Memory reference

$$\text{ALUOutput} \leftarrow A + \text{Imm};$$

- Register-Register ALU instruction

$$\text{ALUOutput} \leftarrow A \text{ funct } B;$$

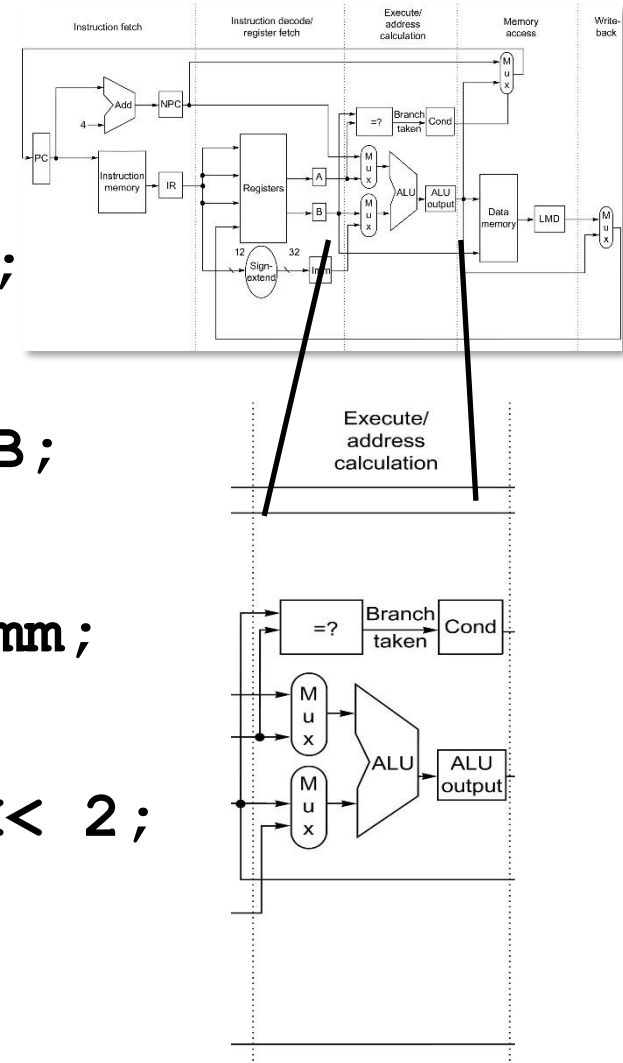
- Register-Immediate ALU instruction

$$\text{ALUOutput} \leftarrow A \text{ funct } \text{Imm};$$

- Branch

$$\text{ALUOutput} \leftarrow \text{NPC} + \text{Imm} \ll 2;$$

$$\text{Cond} \leftarrow (A == B);$$



Memory Access/Branch Completion Cycle

- Memory reference

$LMD \leftarrow Mem[ALUOutput] \text{ or}$
 $Mem[ALUOutput] \leftarrow B;$

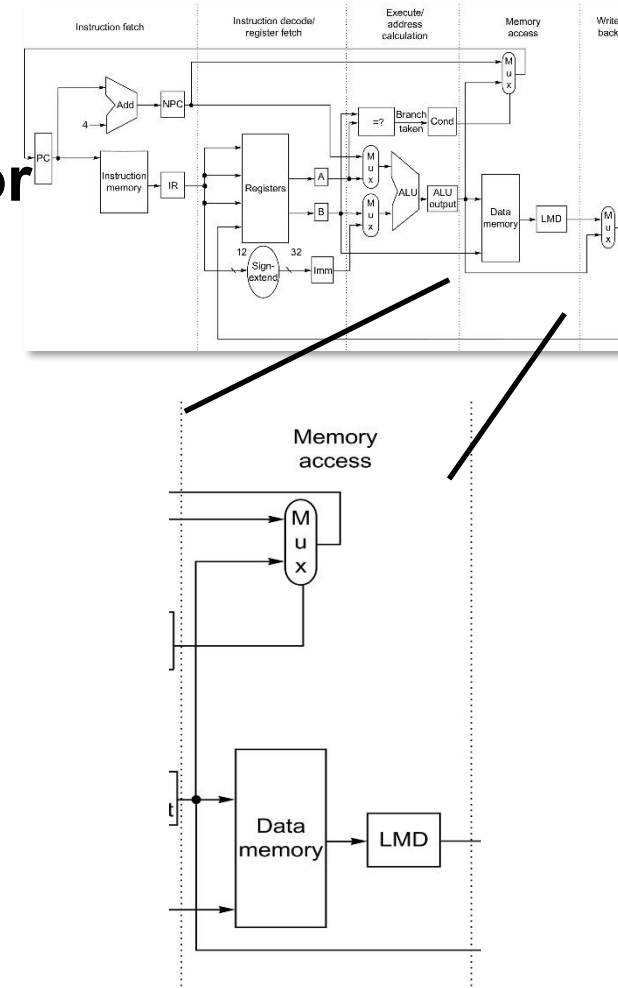
- Branch

if (cond)

$PC \leftarrow ALUOutput$

else

$PC \leftarrow NPC;$



Write-back Cycle

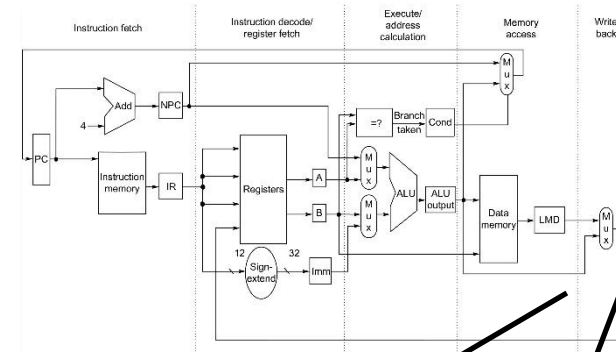
- Register-Register ALU instruction

$$\text{Regs}[\text{IR}_{11..7}] \leftarrow \text{ALUOutput};$$

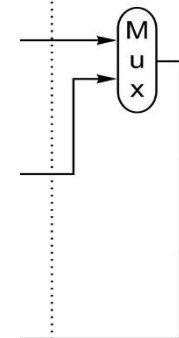
- Register-Immediate ALU instruction

$$\text{Regs}[\text{IR}_{11..7}] \leftarrow \text{ALUOutput};$$

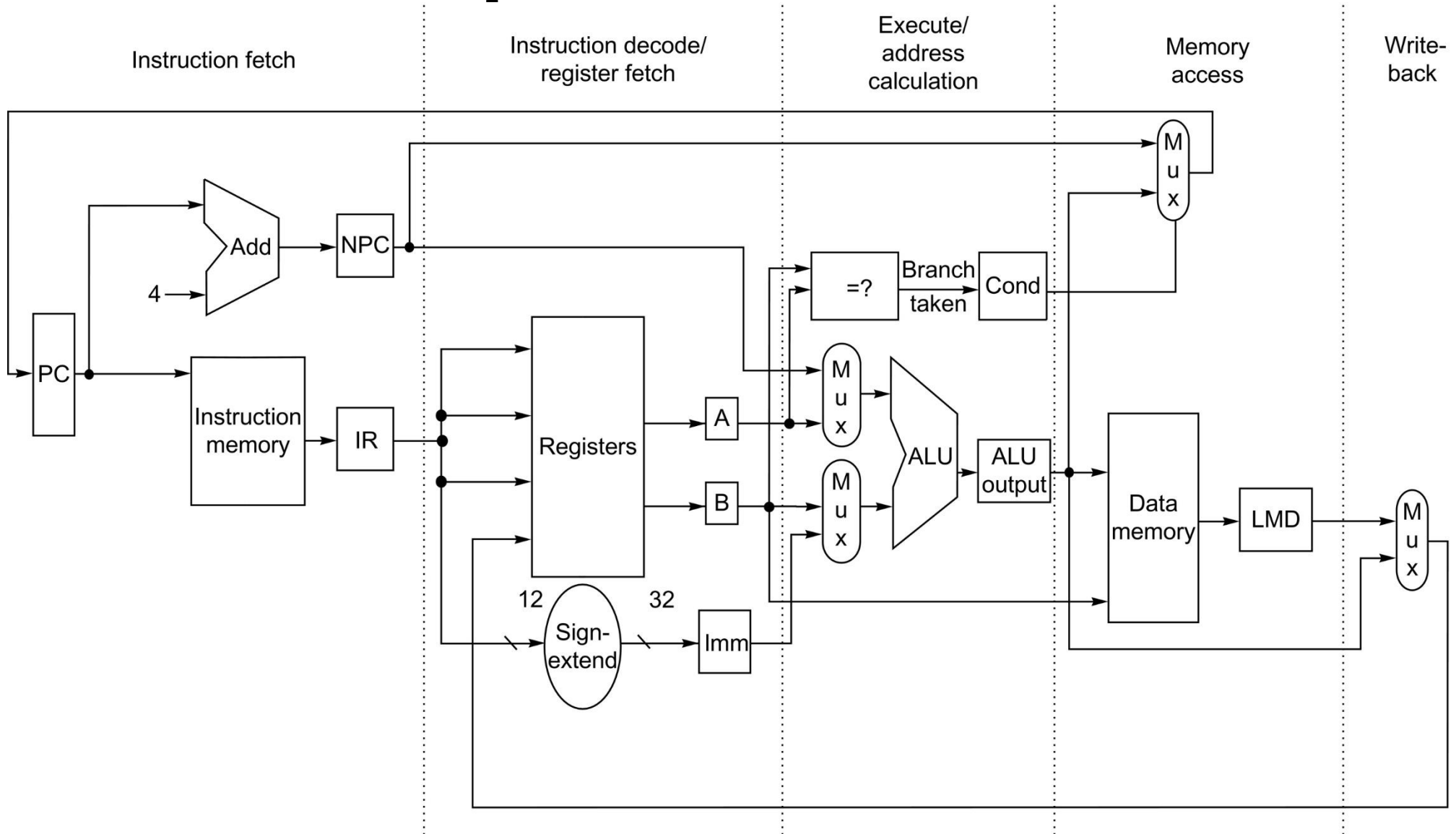
- Load instruction

$$\text{Regs}[\text{IR}_{11..7}] \leftarrow \text{LMD};$$


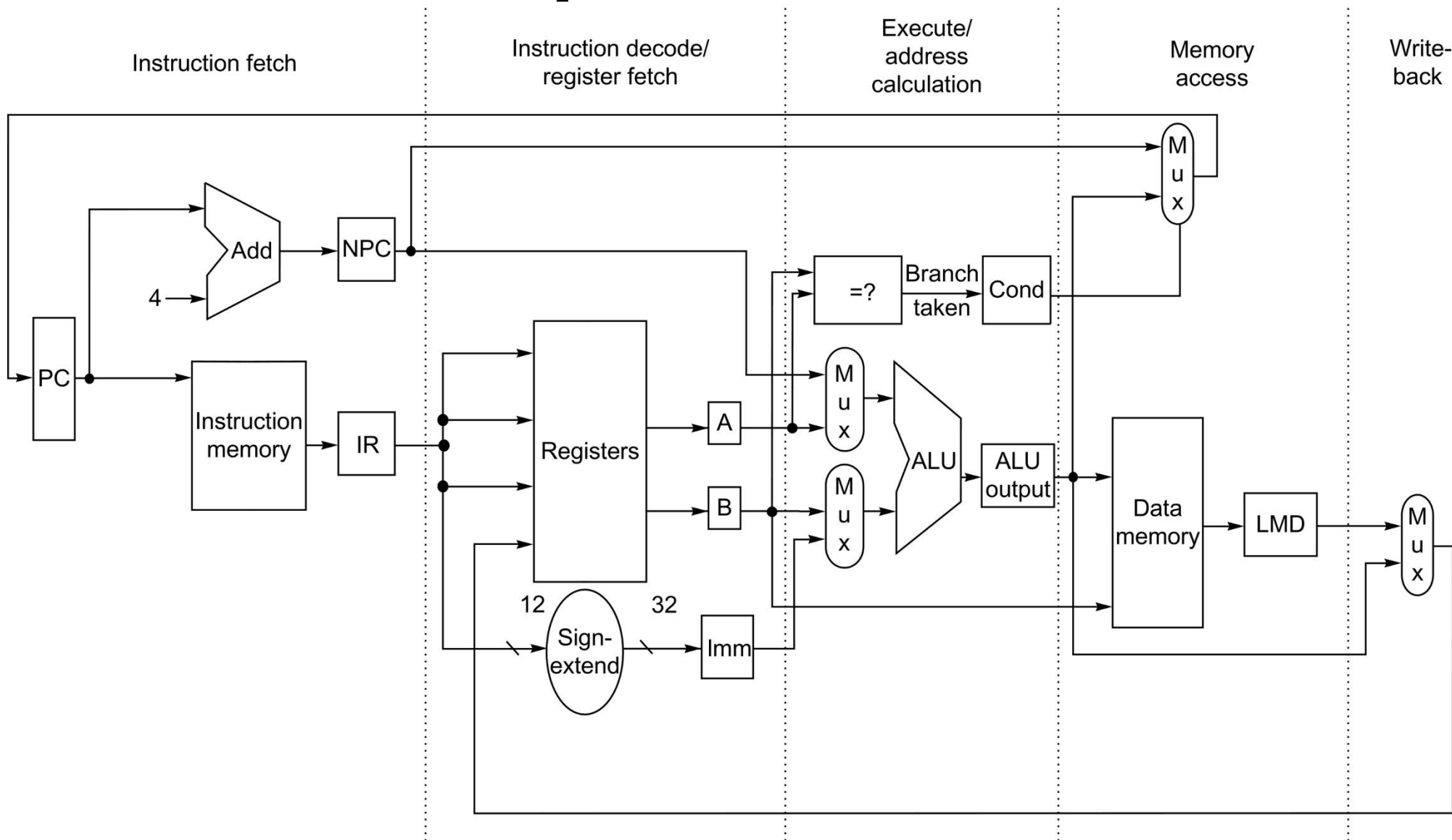
Write-back



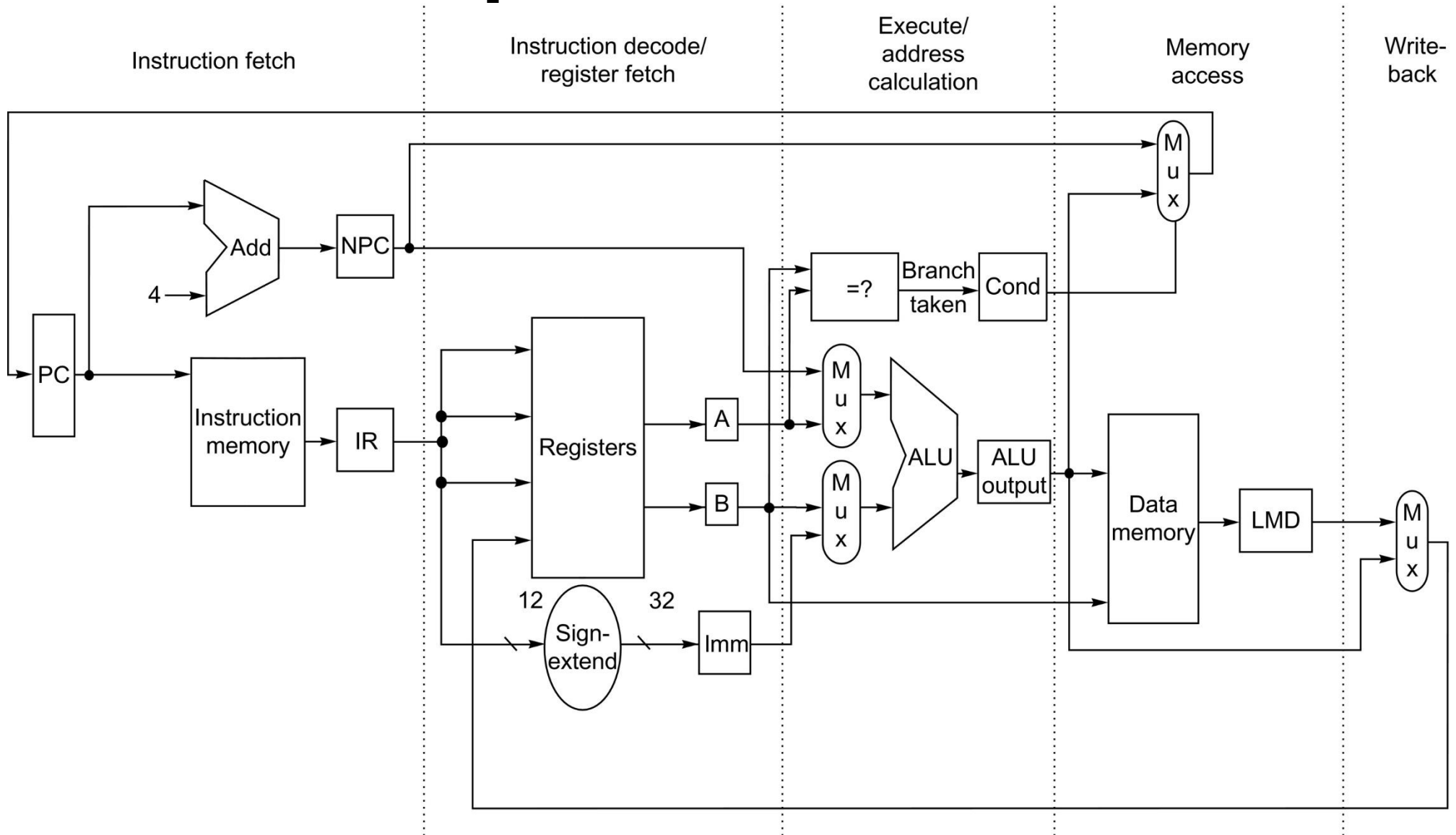
The Datapath – arithmetic instructions



The Datapath – branch instruction



The Datapath – load/store instruction



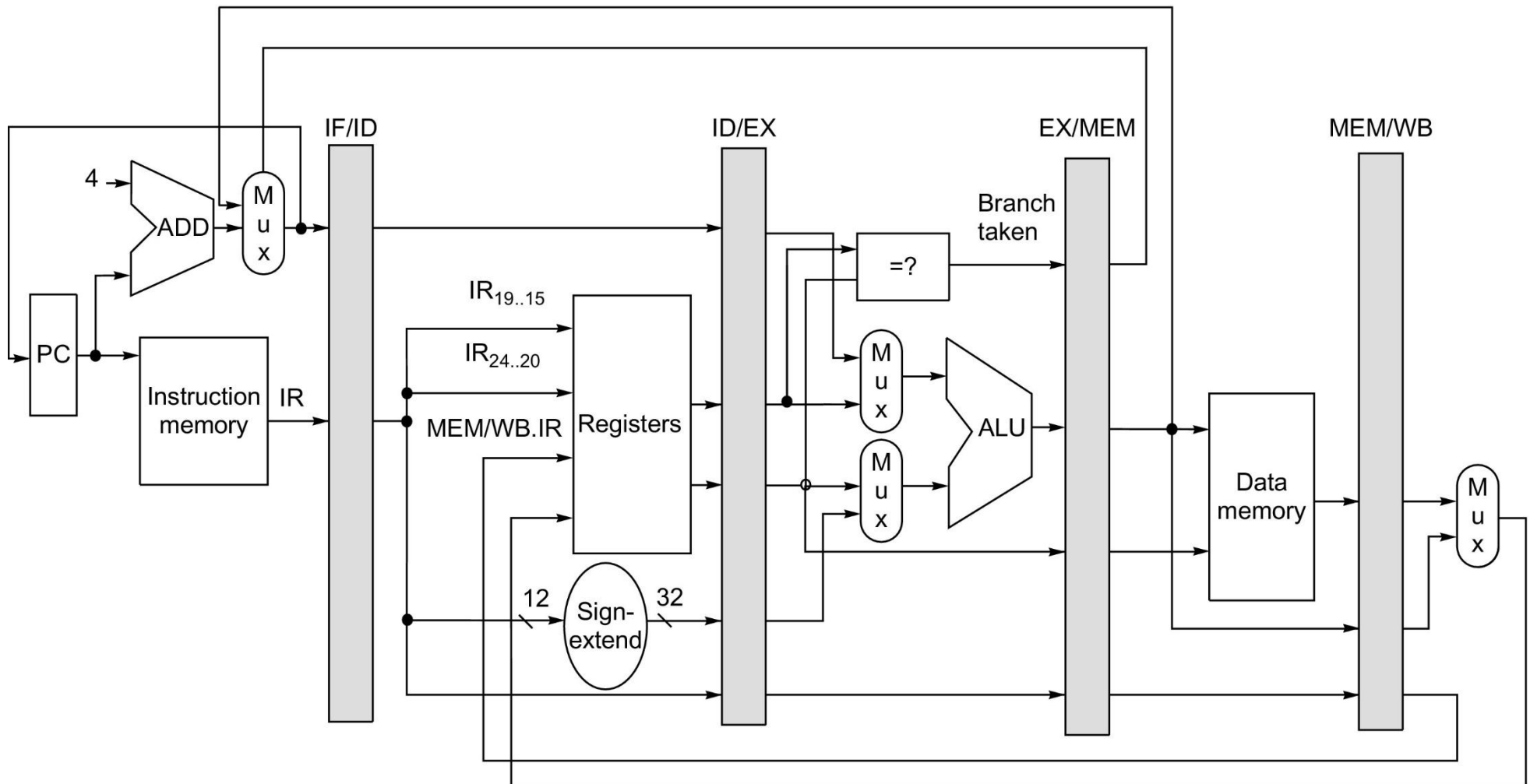
Behavior and optimizations

- 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.

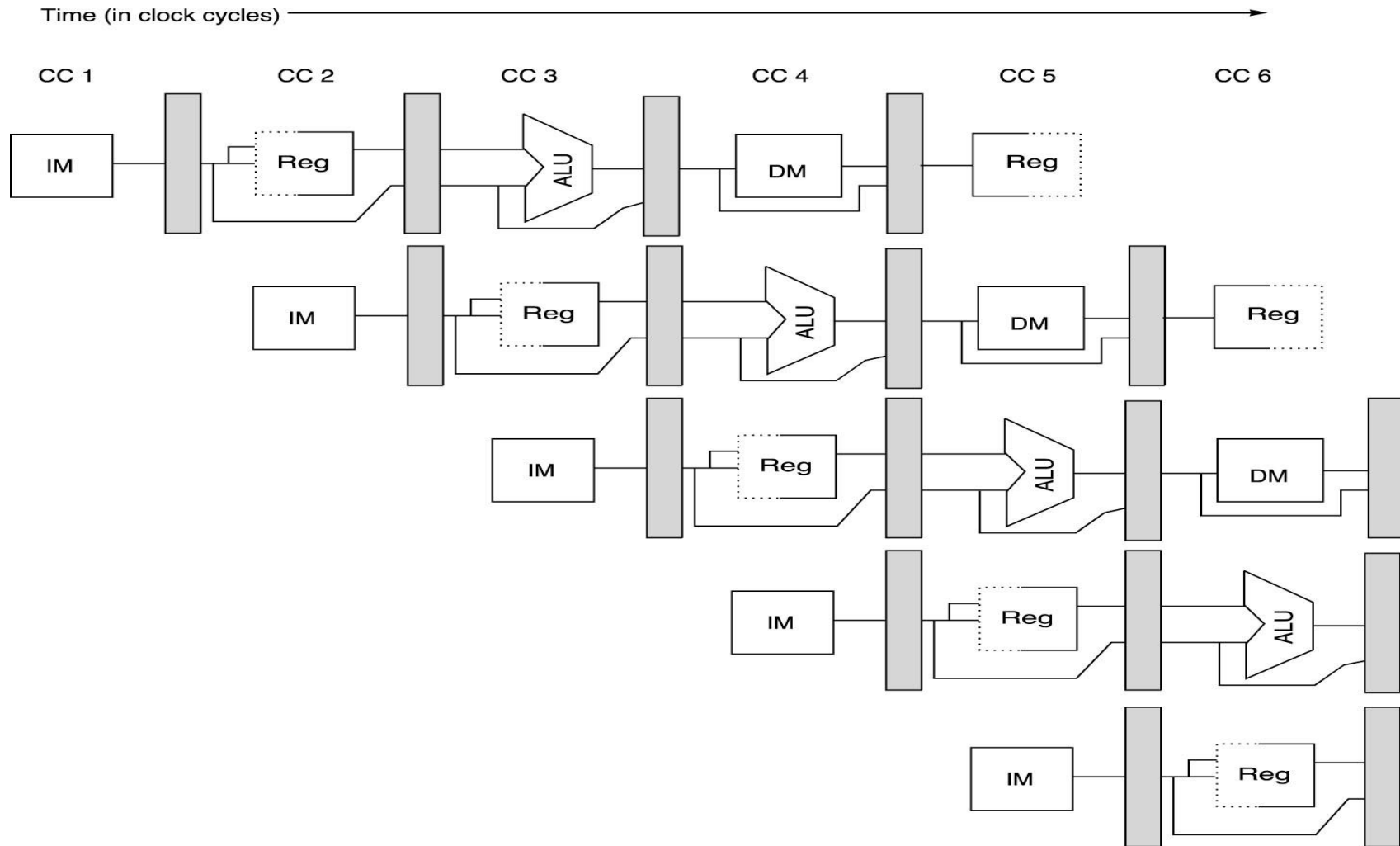
Example processor: basic pipelined version

- 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%, 20%, and 40%, respectively.

The average instruction execution time is

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

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 conflict
- ***data hazards***: an instruction depends on the result of a previous instruction
- ***control hazards***: depend on pipelining 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.

Examp

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 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

Examp

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

	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

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

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:

$$\text{Average Instruction Time} = \text{CPI} \times \text{clock cycle time}$$

For the machine with structural hazard:

$$\text{Clock}_{str} = \frac{\text{Clock}_{ideal}}{1.05} \rightarrow \text{Clock}_{ideal} = \text{Clock}_{str} * 1.05$$

$$\text{Avg Instr Time}_{struct} = \text{CPI}_{ideal} + \text{Instr_freq} \times \text{Stall}_{penalty}$$

$$\begin{aligned} \text{Avg Instr Time}_{struct} &= (1 + 0.4 \times 1) \times \text{Clock}_{ideal}[t] / 1.05 \\ &= 1.33 \times \text{Clock}_{ideal}[t] \end{aligned}$$

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:

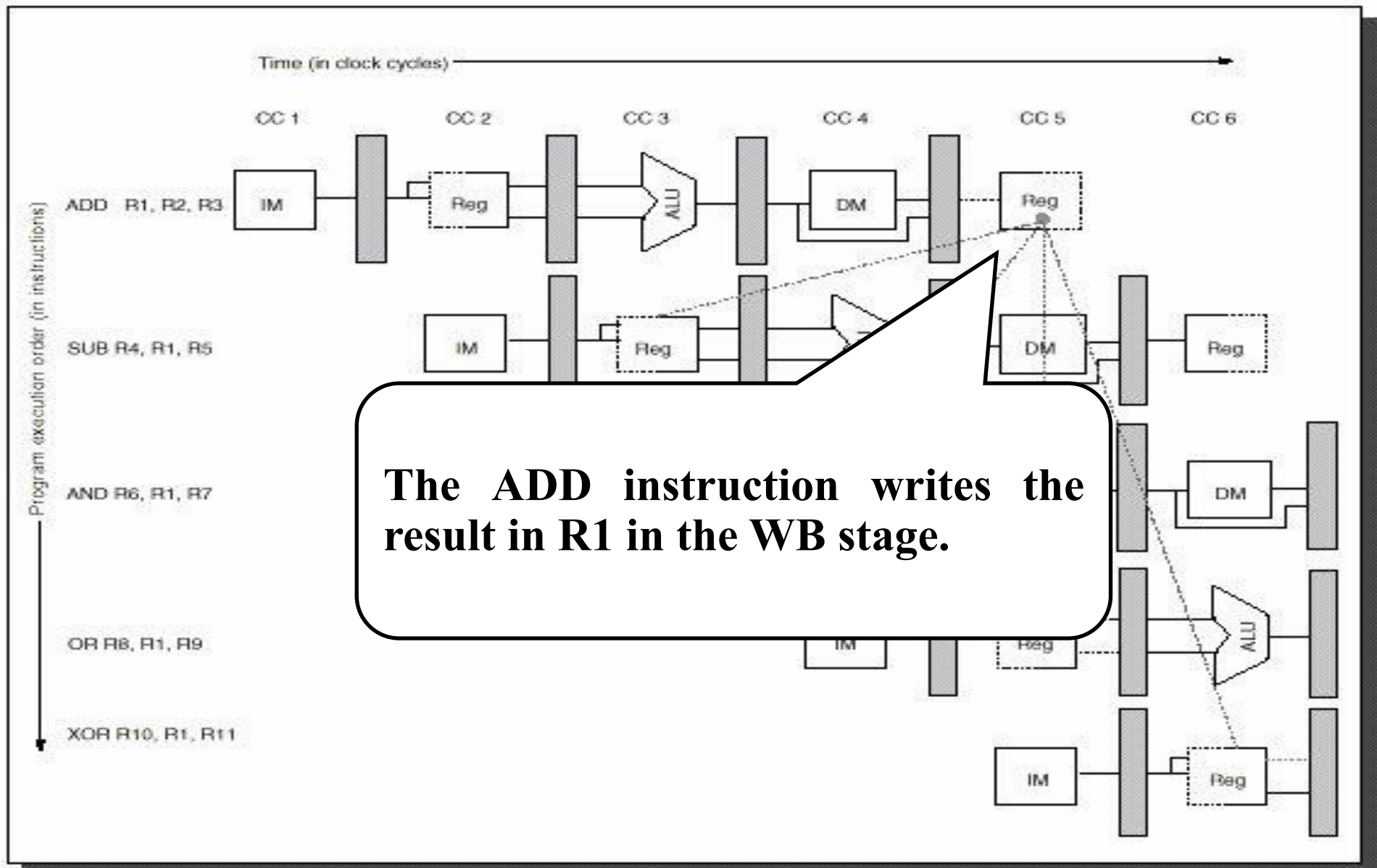
- wrong results**
- undeterministic behavior.**

Example

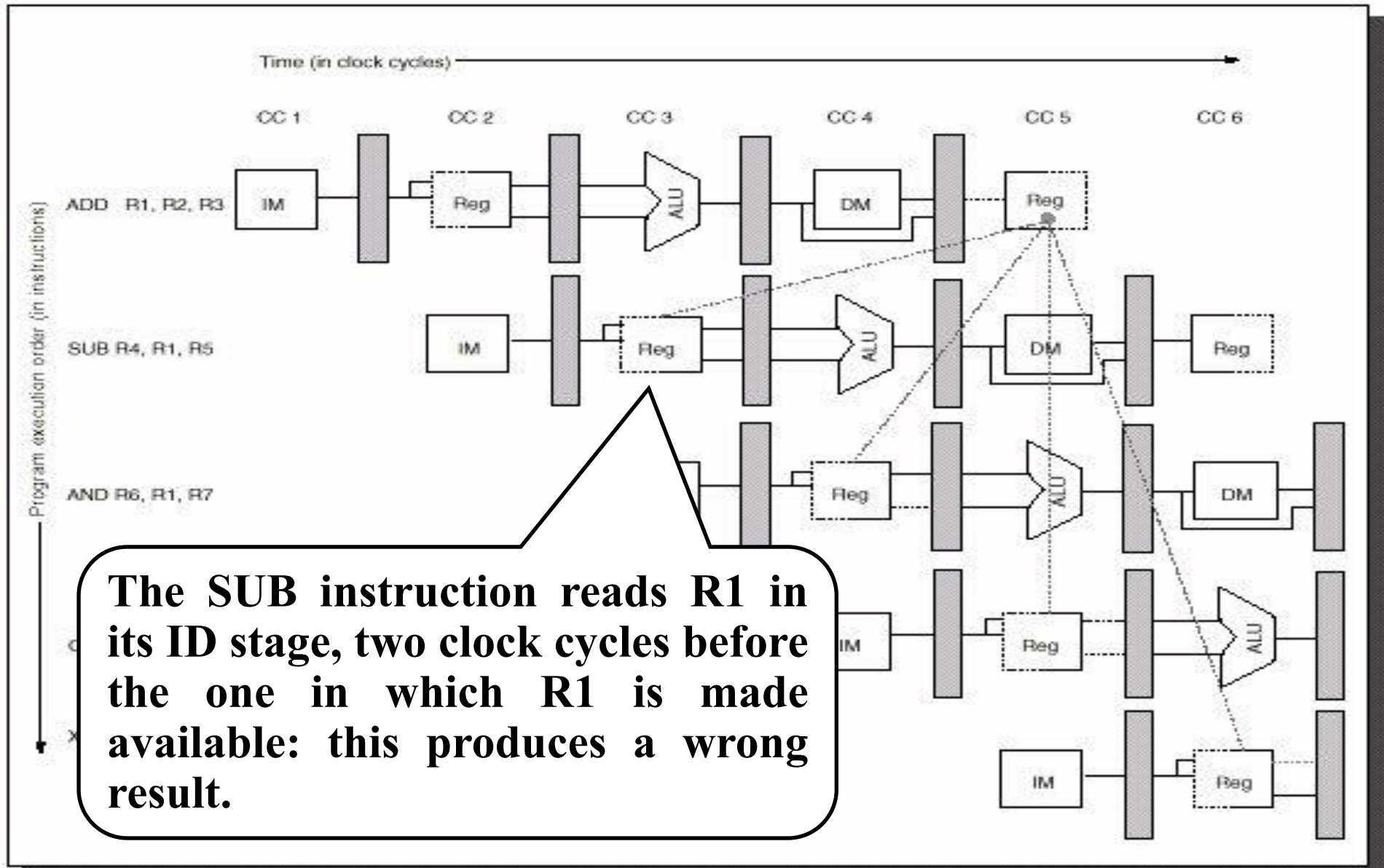
Let consider the following code fragment:

```
ADD    R1 , R2 , R3
SUB     R4 , R1 , R5
AND     R6 , R1 , R7
OR      R8 , R1 , R9
XOR     R10 , R1 , R11
```

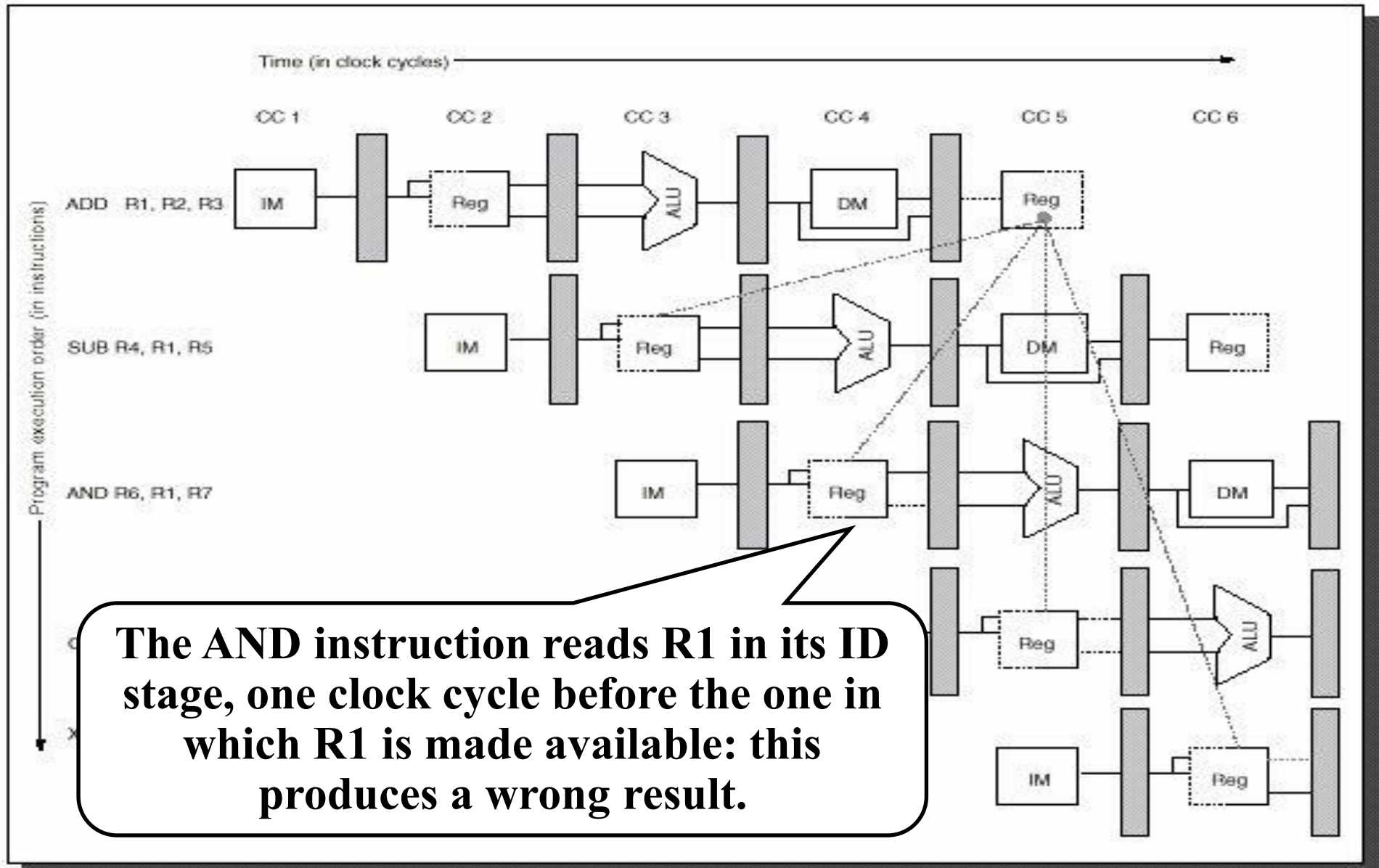
Example (cont'd)



Example (cont'd)

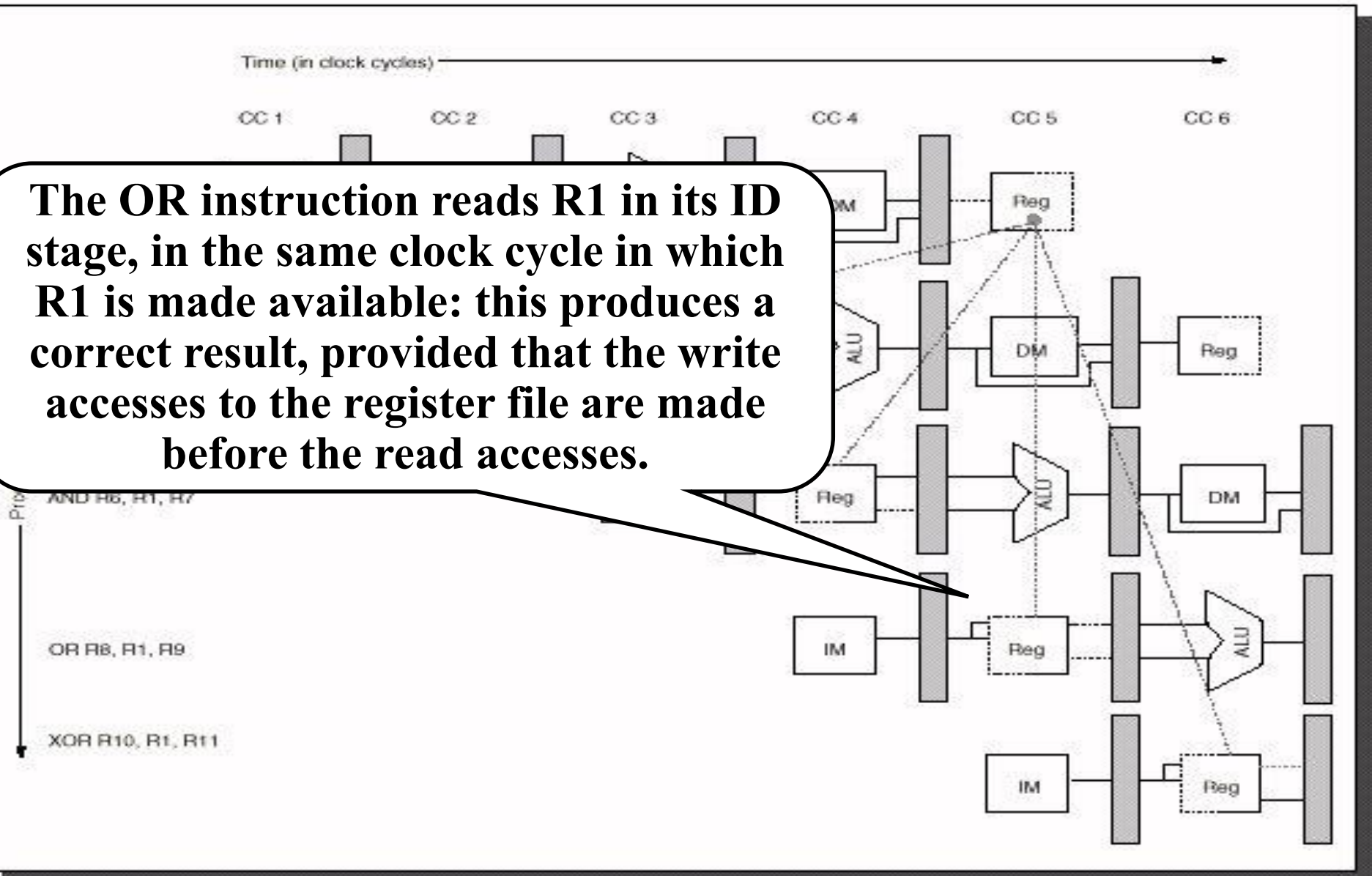


Example (cont'd)

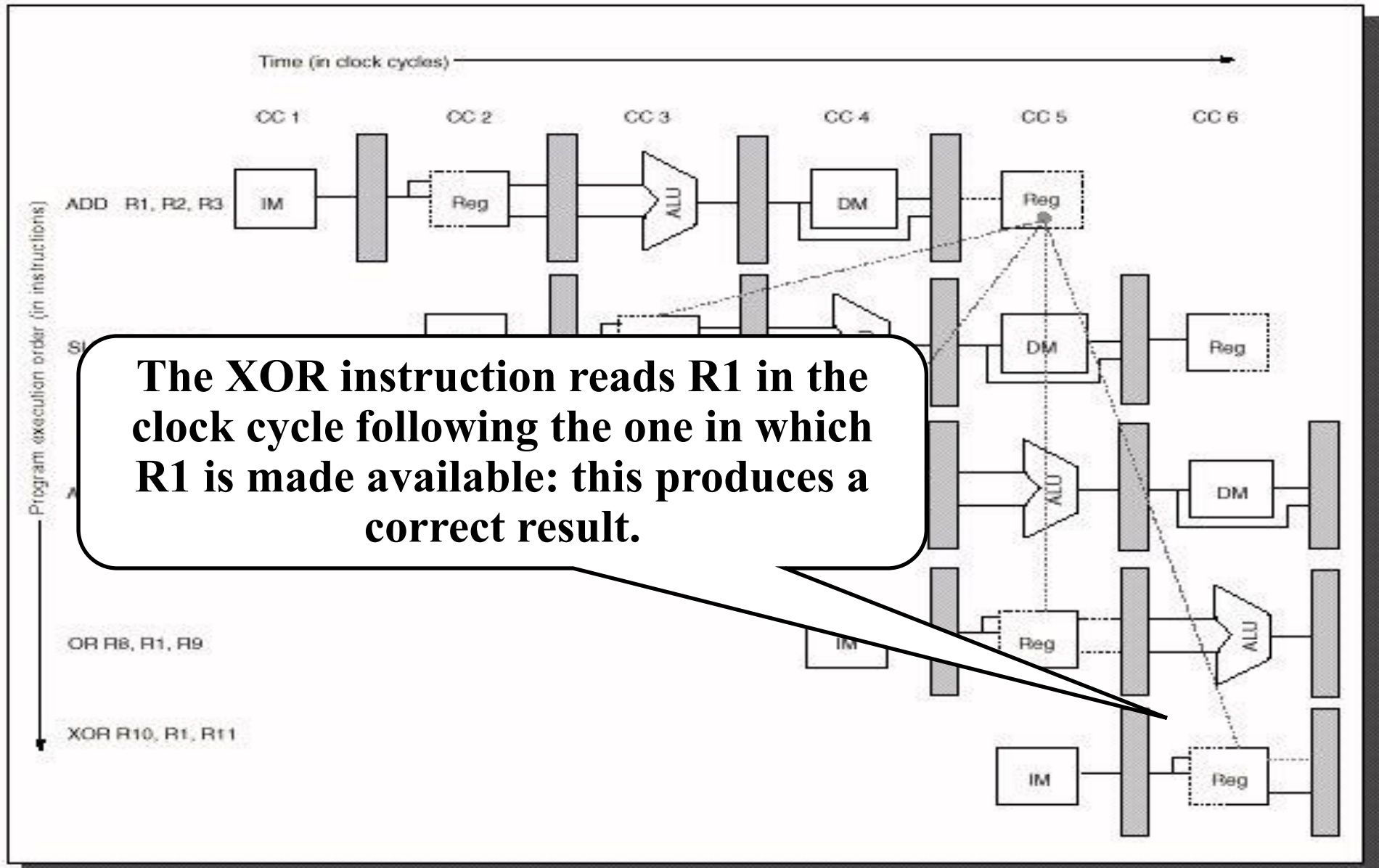


Example (cont'd)

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 restored.

This may cause an *undeterministic behavior*.

Overcoming data hazards effects

The wrong results produced by data hazards can be avoided:

- by stalling the instructions requiring the data until they are available
- by implementing a *forwarding* (or *bypassing*) technique.

Forwarding

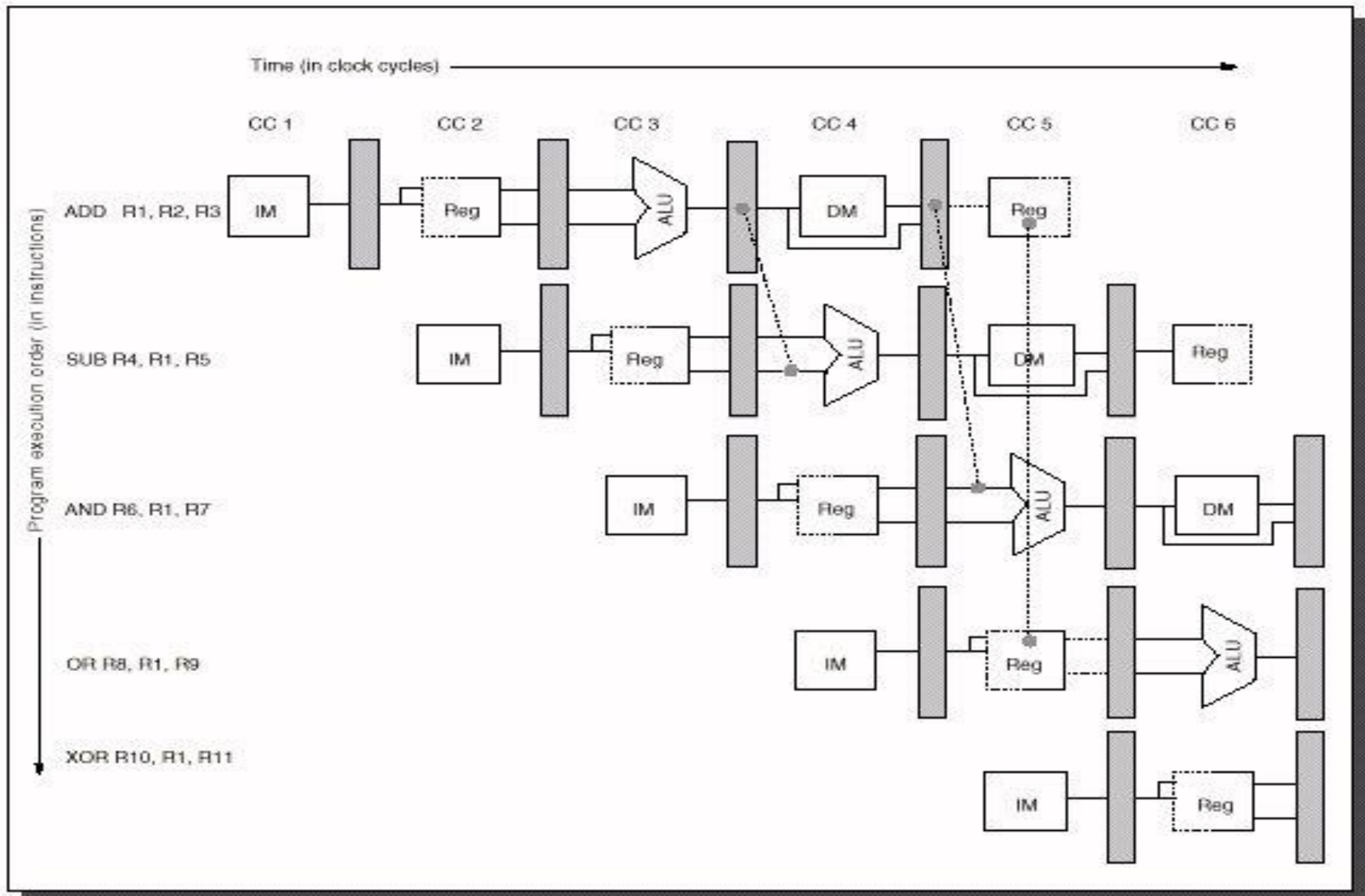
A special hardware in the datapath detects when a previous ALU operation should write the register corresponding to the source of the current ALU operation.

In this case, the hardware selects the ALU result as the ALU input rather than the value from the register file.

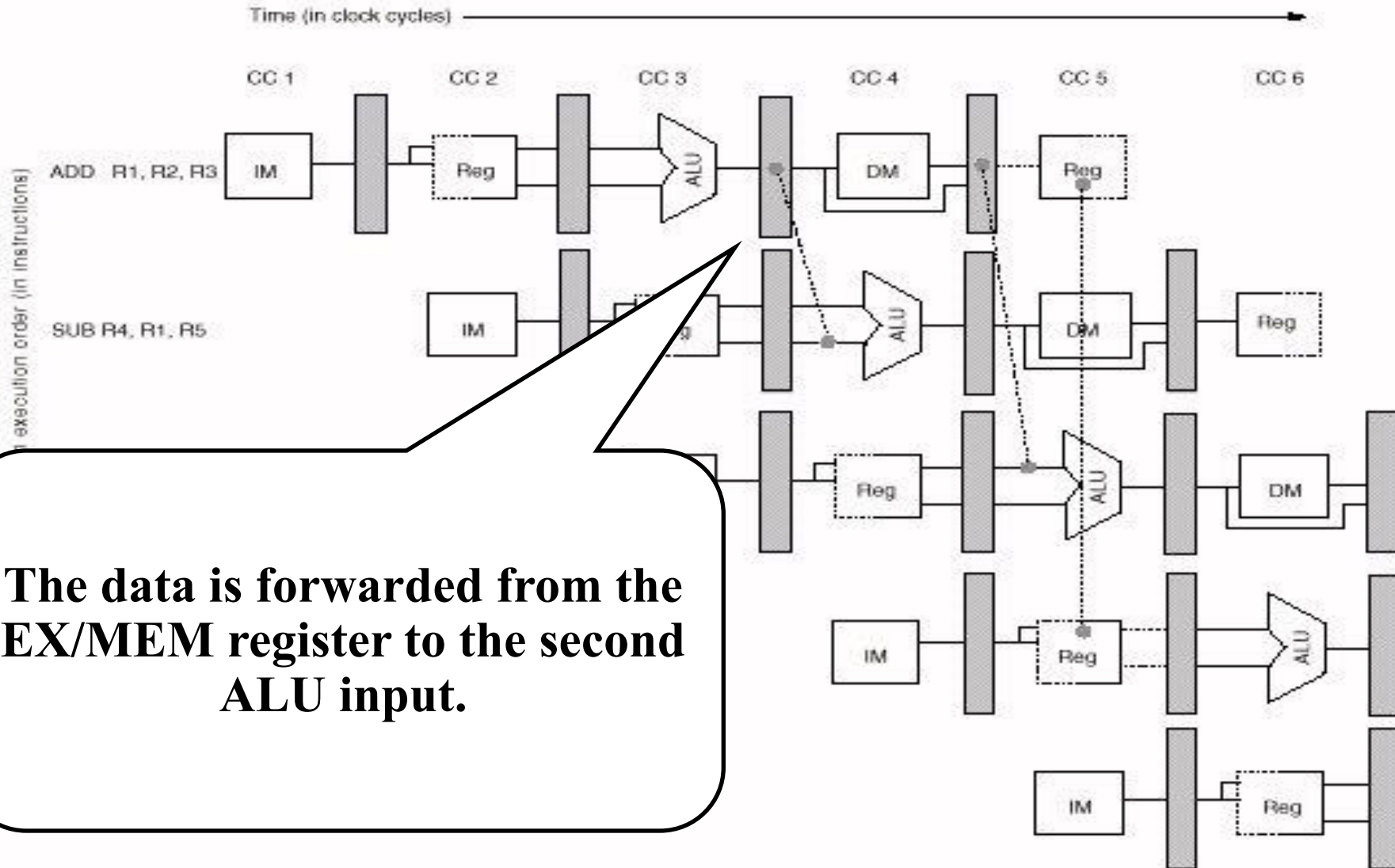
The hardware must be able

- 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.**

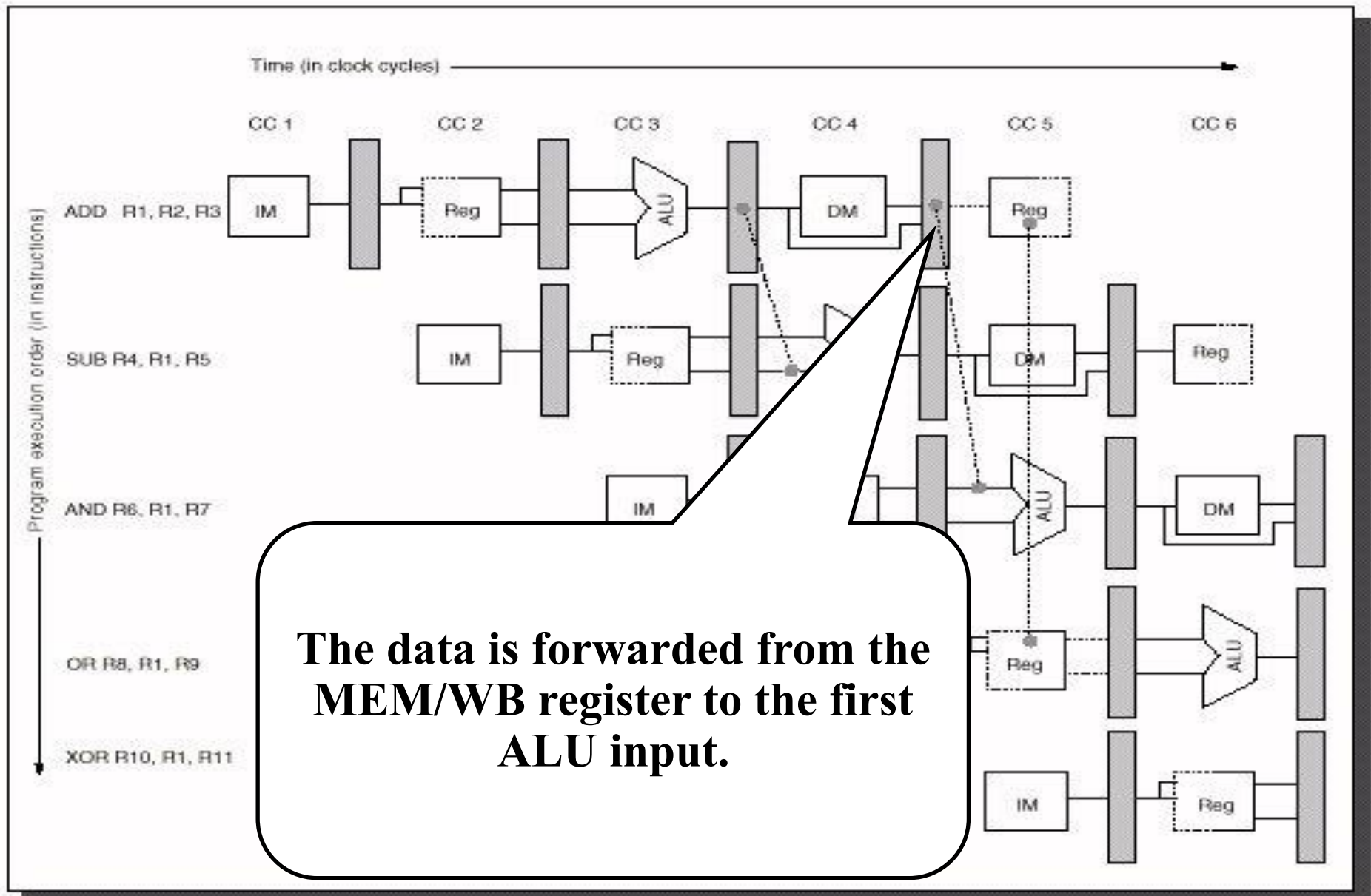
Example



Example



Example



Generalizing the forward technique

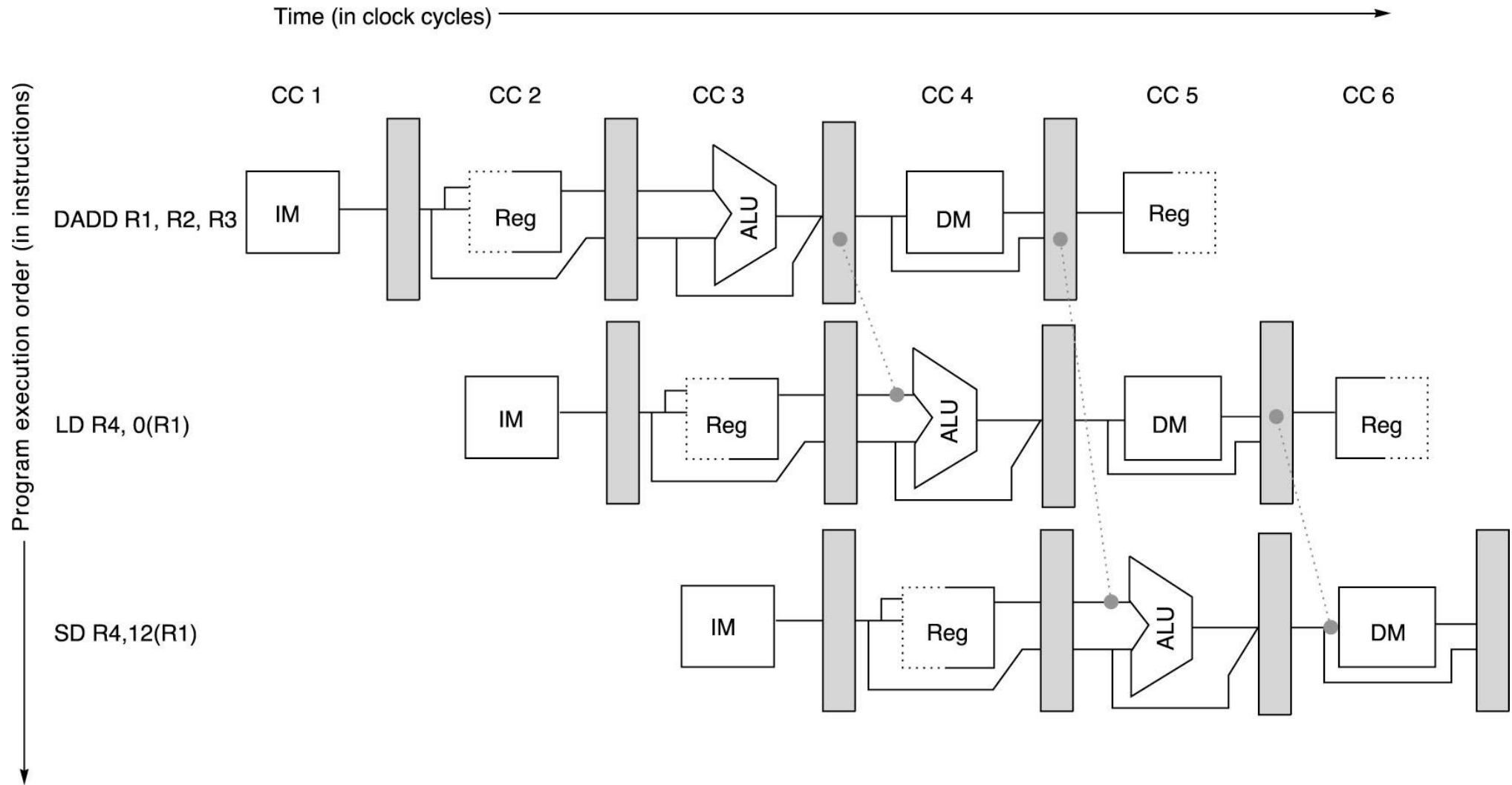
In order to always avoid stalling, forwarding should be made possible between any pipeline register to any input of any functional unit.

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.**

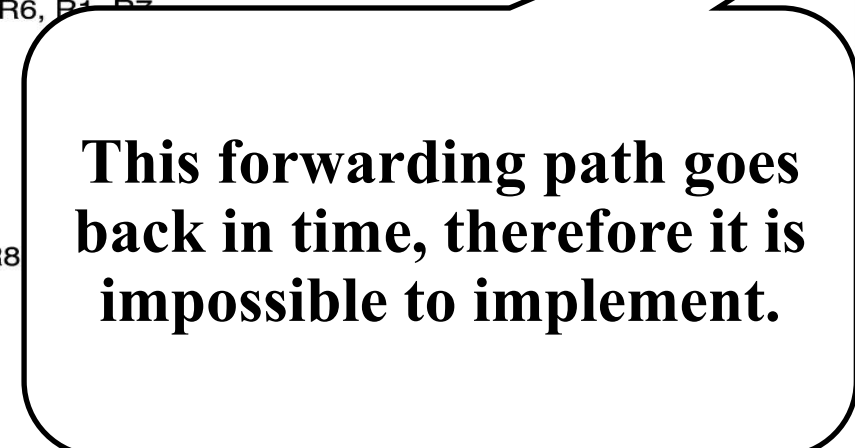
Data Hazards Requiring Stalls

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

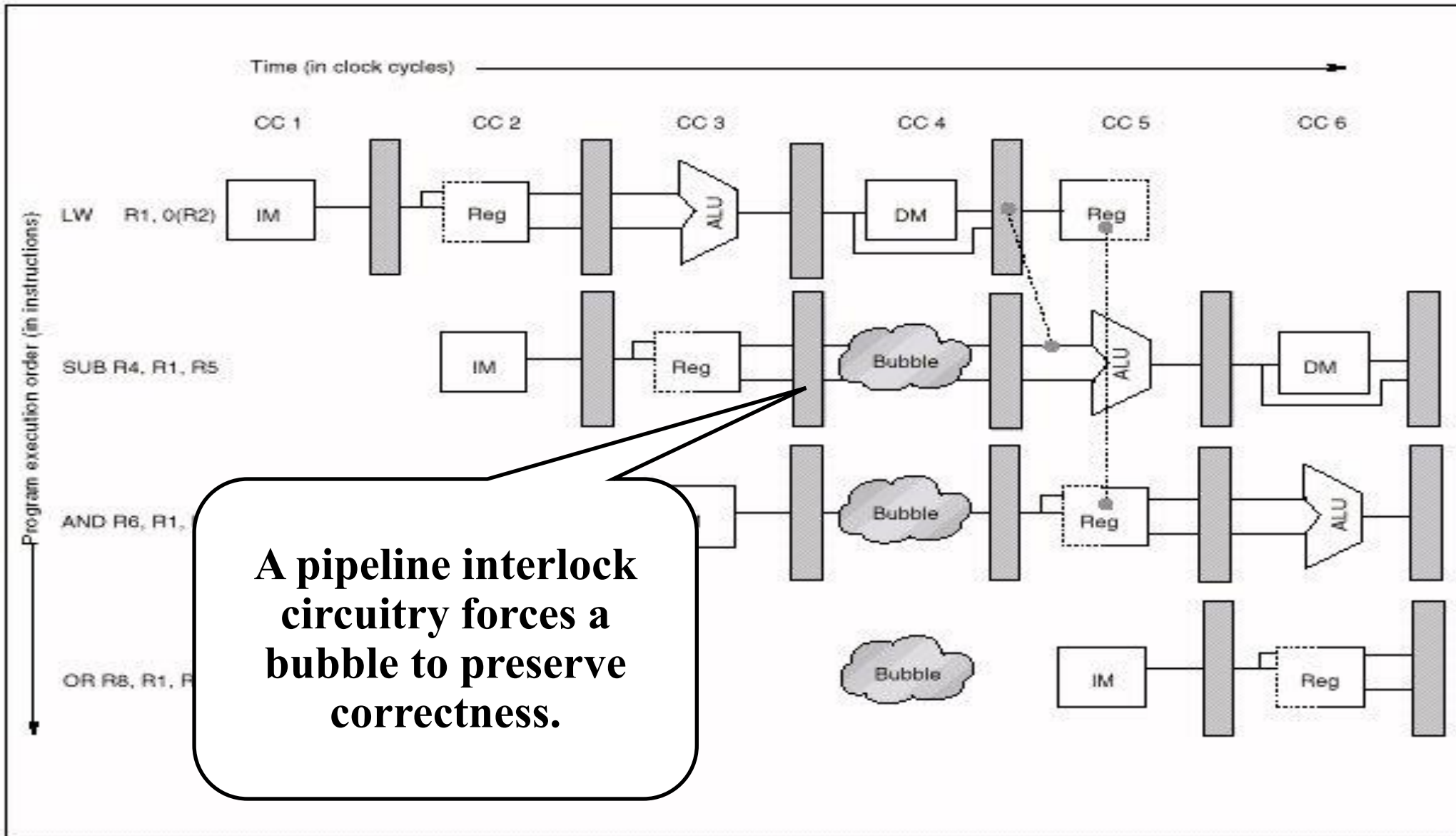
Example

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

This forwarding path goes back in time, therefore it is impossible to implement.



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 R1 ,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 R1 ,45(R2) DADD R5, R1 ,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 R1 ,45(R2) DADD R5,R6,R7 DSUB R8, R1 ,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 R1 ,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9, R1 ,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 and another instruction exploiting the loaded value is in the ID stage.

Opcode field of ID/EX (ID/EX.IR _{0..6})	Opcode field of IF/ID (IF/ID.IR _{0..6})	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 from advancing.

Introducing a stall

Given an instruction currently in the ID stage, introducing a stall in the EX stage can be done:

- **forcing a nop instruction in the ID/EX pipeline register (in RISC-V the nop corresponds to an `addi x0, x0, 0`)**
- **forcing the IF/ID pipeline register to maintain the current value**
- **Program counter must be frozen (maintain unaltered IF status).**

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).

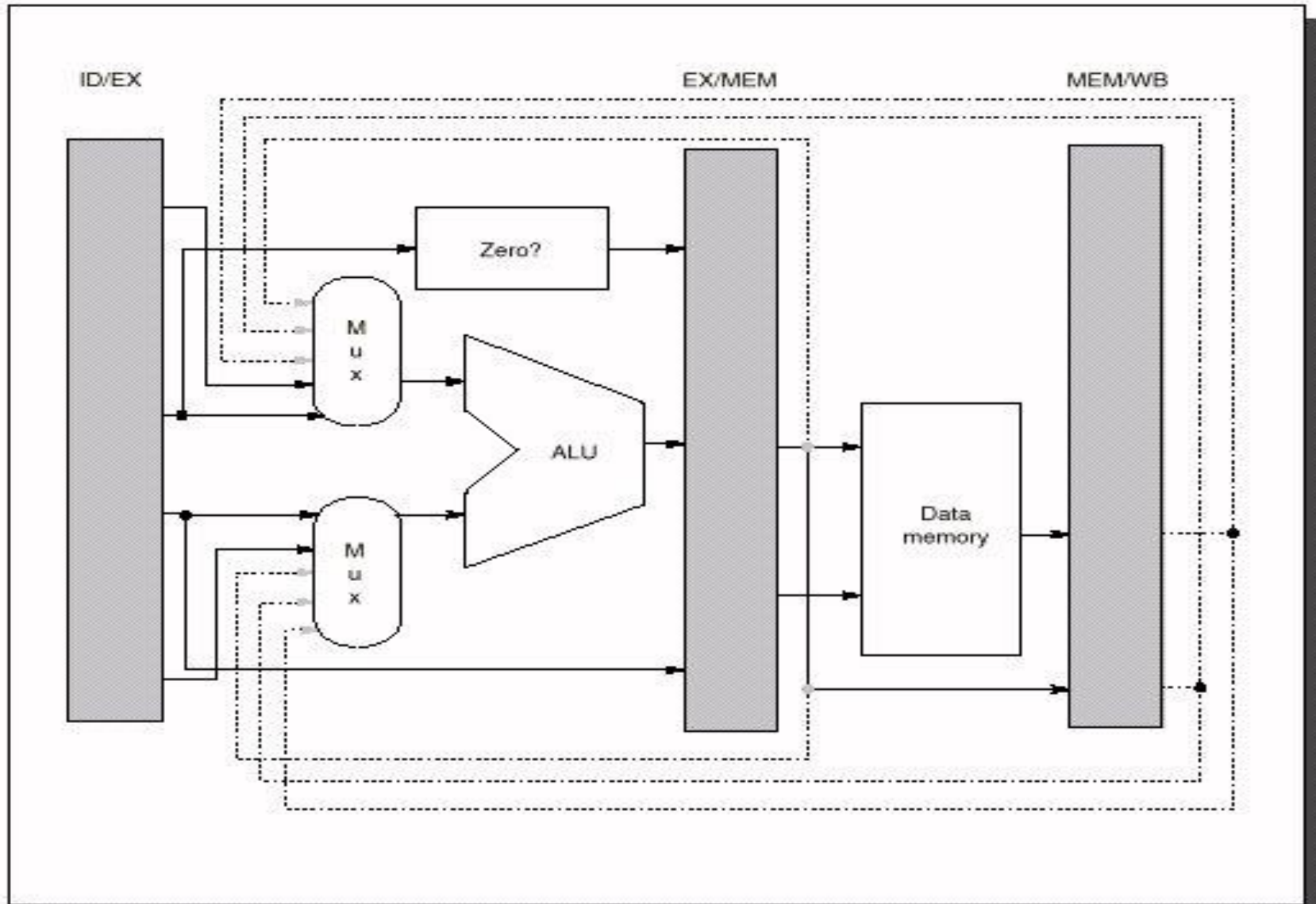
The forwarding logic must compare:

- 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 of source instruction	Opcode of source instruction	Pipeline register of destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU, ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR[rd] == ID/EX.IR[rs1]
EX/MEM	Register-register ALU, ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rd] == ID/EX.IR[rs2]
MEM/WB	Register-register ALU, ALU immediate, Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rd] == ID/EX.IR[rs1]
MEM/WB	Register-register ALU, ALU immediate, Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rd] == ID/EX.IR[rs2]

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 basic RISC-V implementation, the PC is written with the target address (if the jump is *taken*) at the end of the EXE stage, i.e., 2 clock cycles after the IF of the branch.

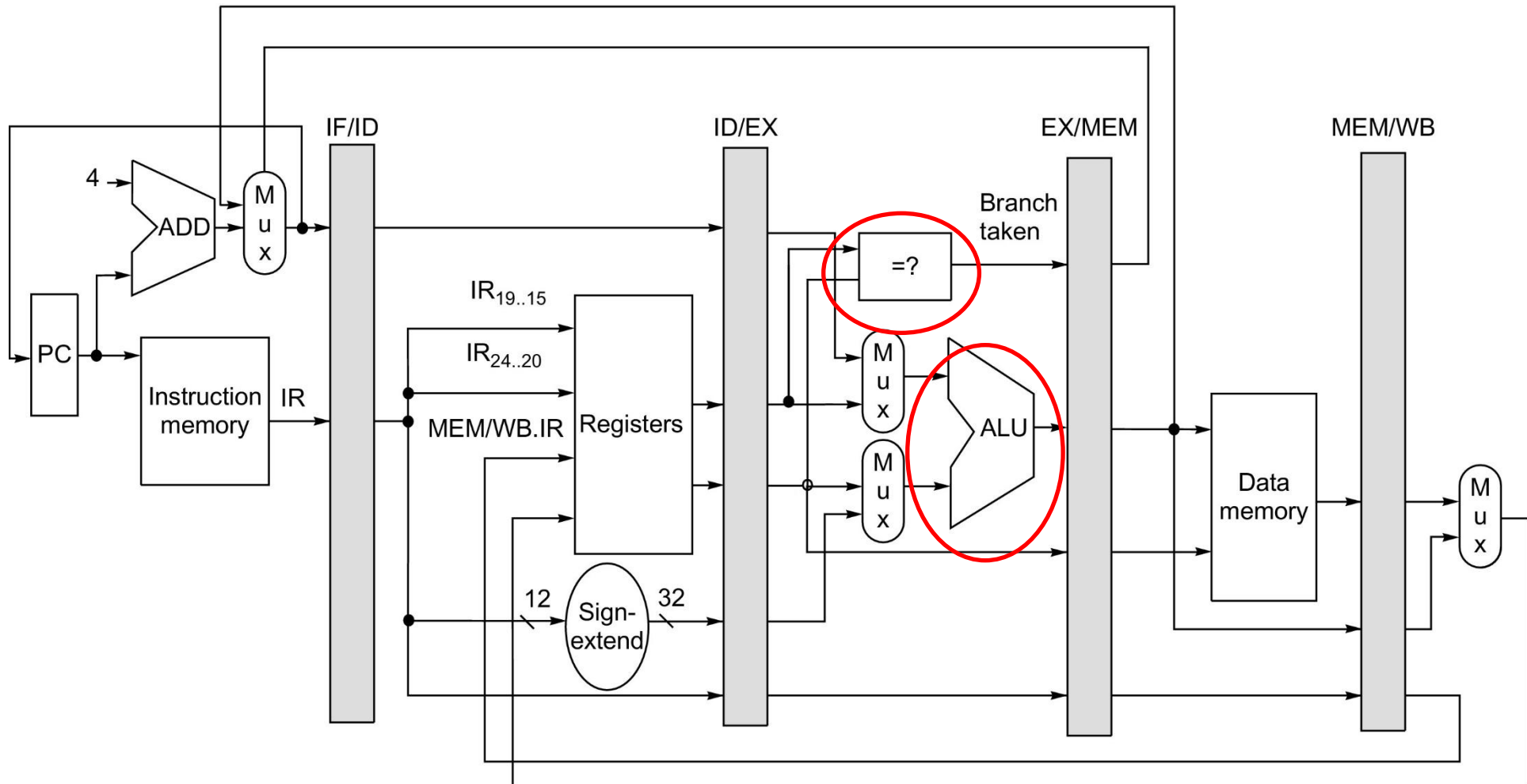
This behavior delays the target instruction by two clock cycles.

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



Example

Branch instruction	IF	ID	EX	MEM	WB		
Branch +1		IF	ID	--	--	--	
Branch +2			IF	--	--	--	--
Branch Target				IF	ID	EX	MEM
Branch Target + 1					IF	ID	EX

The branch instruction is taken.

Example

Branch instruction	IF	ID	EX	MEM	WB		
Branch +1		IF	ID	--	--	--	
Branch +2			IF	--	--	--	--
Branch Target				IF	ID	EX	MEM
Branch Target + 1					IF	ID	EX

This stage fetches the following instruction (as if the branch is not taken).

This stage fetches the branch + 2 instruction .

Example

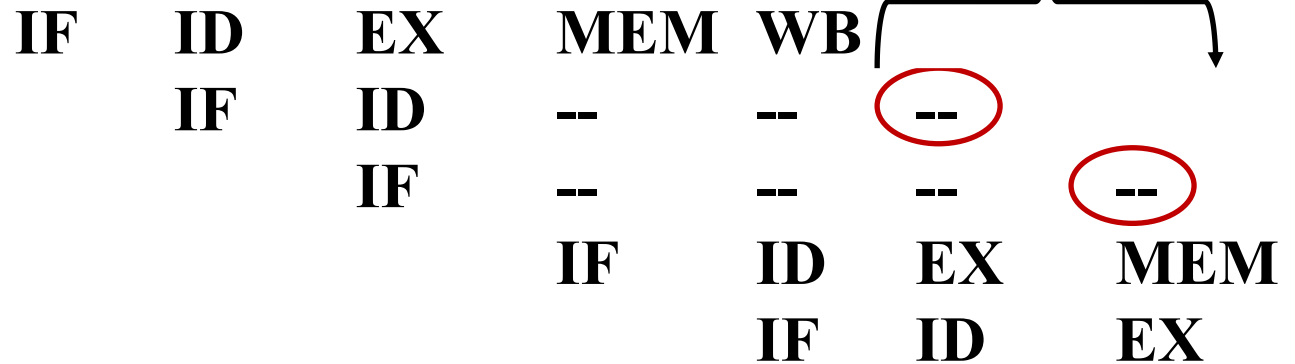
Branch instruction	IF	ID	EX	MEM	WB		
Branch +1		IF	ID	--	--	--	
Branch +2			IF	--	--	--	--
Branch Target				IF	ID	EX	MEM
Branch Target + 1					IF	ID	EX

The branch target instruction
is fetched here.

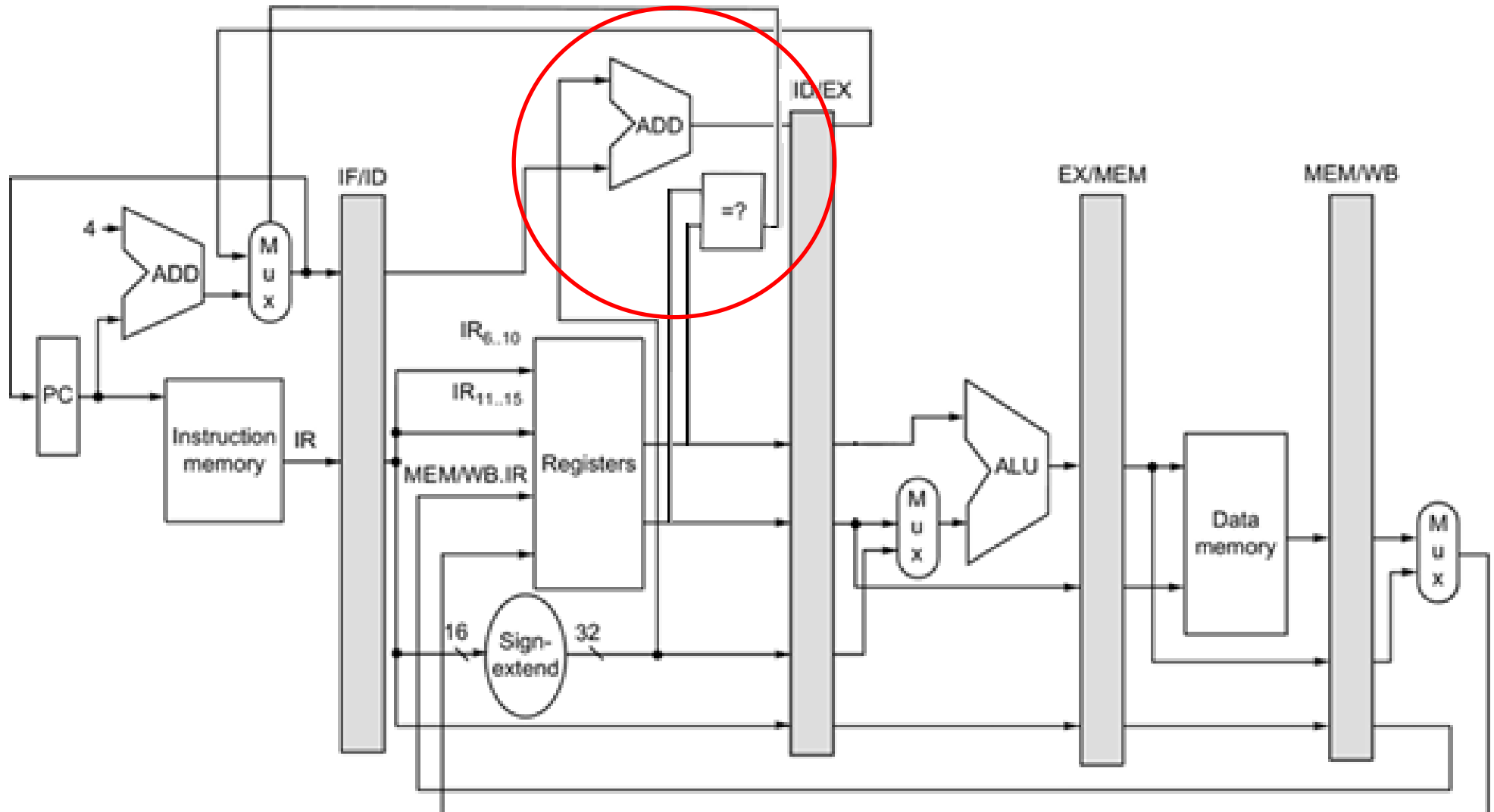
Example

If the branch is taken, two clock cycles are always lost!

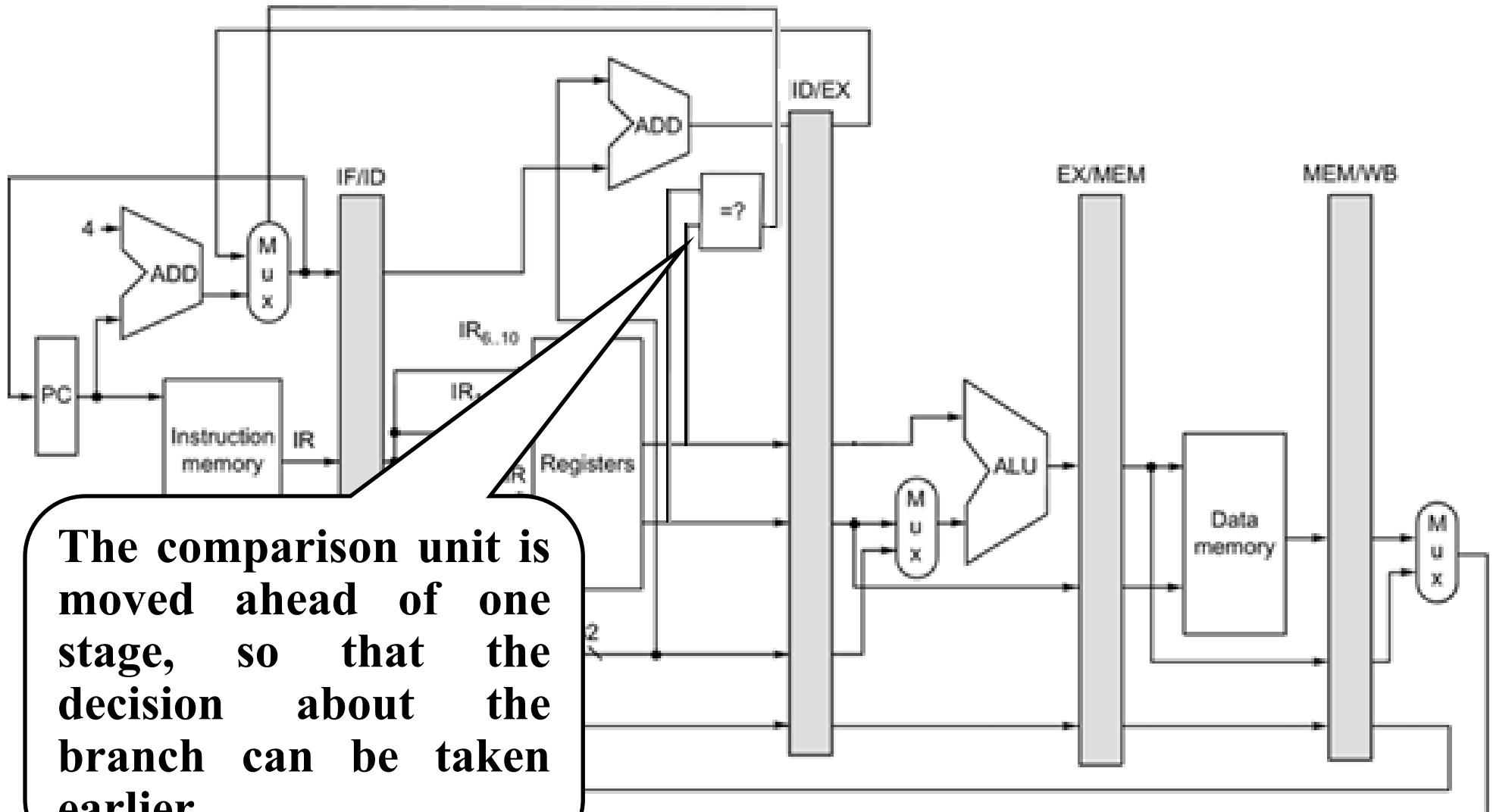
Branch instruction
 Branch +1
 Branch +2
 Branch Target
 Branch Target + 1



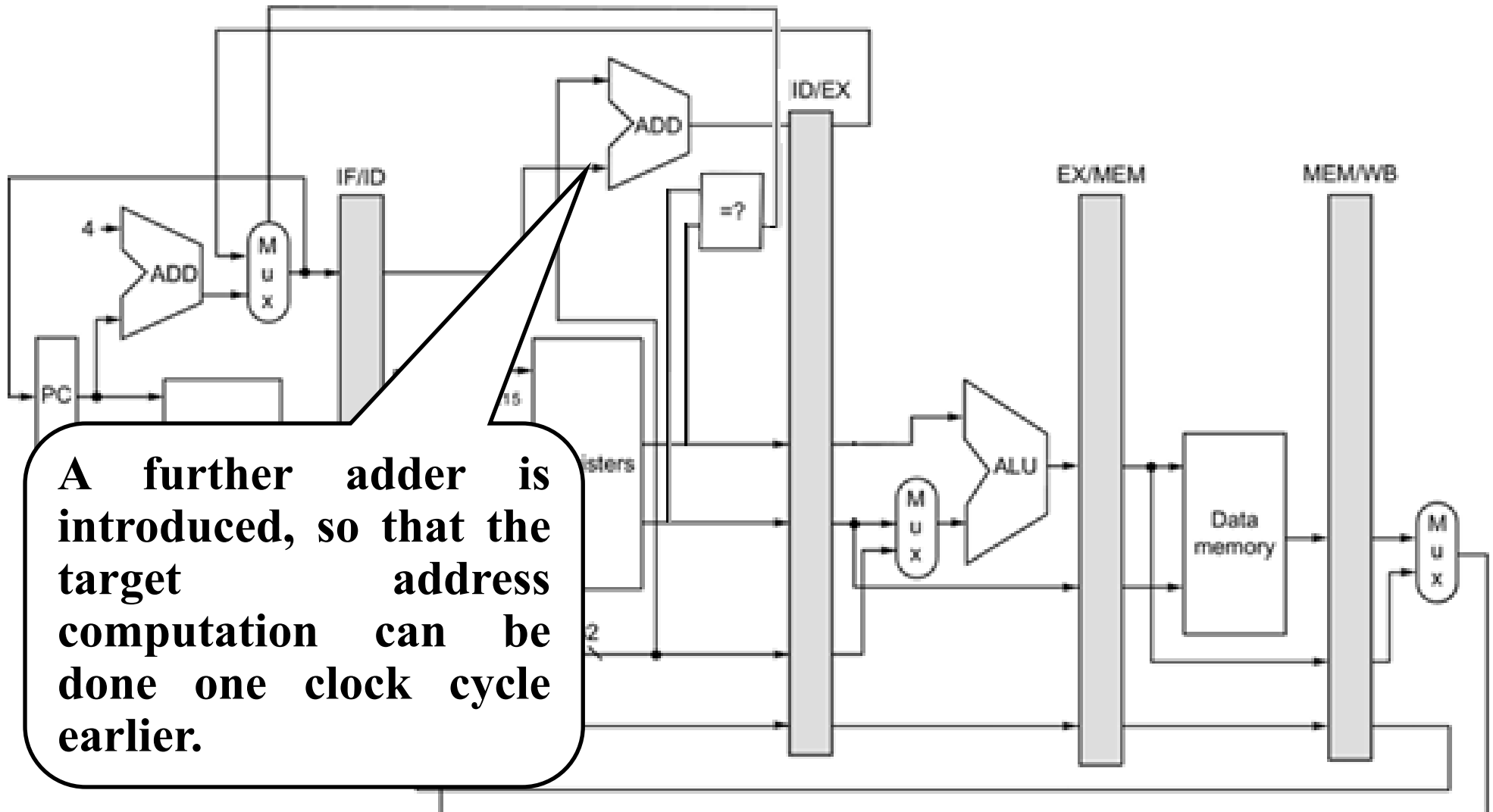
Modified pipeline



Modified pipeline



Modified pipeline



A further adder is introduced, so that the target address computation can be done one clock cycle earlier.

Example

Branch instruction	IF	ID	EX	MEM	WB		
Branch +1		IF	--	--	--	--	
Branch Target			IF	ID	EX	MEM	WB
Branch Target + 1				IF	ID	EX	MEM

The branch instruction is taken.

Example

Branch instruction	IF	ID	EX	MEM	WB		
Branch +1		IF	--	--	--	--	
Branch Target			IF	ID	EX	MEM	WB
Branch Target + 1				IF	ID	EX	MEM

This stage fetches the following instruction (as if the branch is not taken).

Example

Branch instruction	IF	ID	EX	MEM	WB				
Branch +1		IF	--	--	--	--			
Branch Target			IF	ID	EX	MEM	WB		
Branch Target + 1				IF	ID	EX	MEM		

This stage fetches the right instruction (which depends on the branch result).

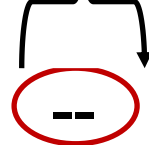
Example

Branch instruction	IF	ID	EX	MEM	WB			
Branch +1		IF	--	--	--	--		
Branch Target			IF	ID	EX	MEM	WB	
Branch Target + 1				IF	ID	EX	MEM	

**This operation is
ALWAYS useless.**

Example

If the branch is taken, only one clock cycle is always lost!

Branch instruction	IF	ID	EX	MEM	WB		
Branch +1		IF	--	--	--	--	
Branch Target			IF	ID	EX	MEM	WB
Branch Target + 1				IF	ID	EX	MEM

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						
Instruction $i + 1$							
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.

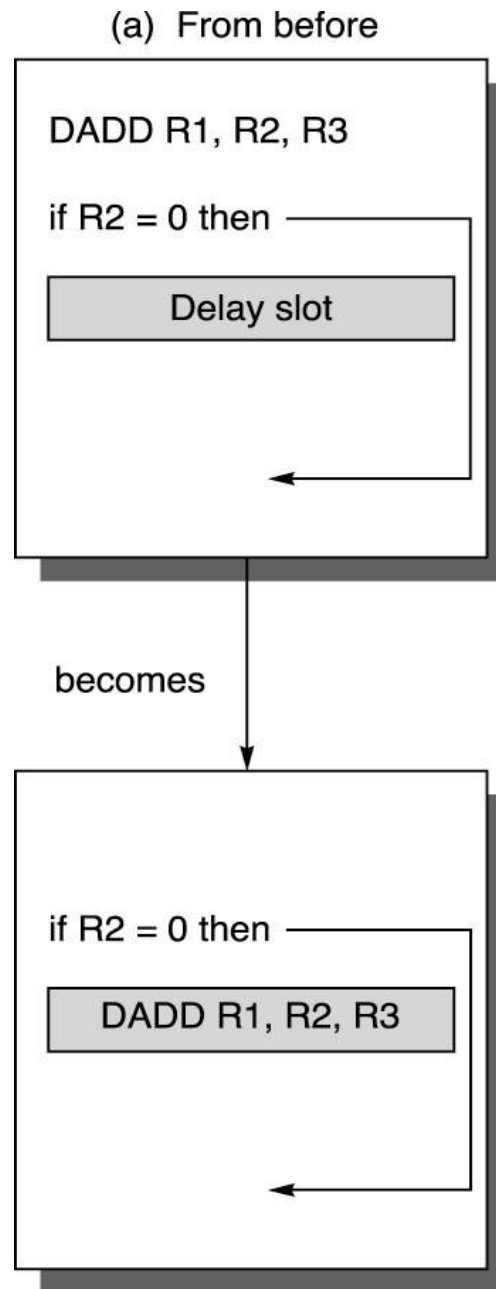
Delayed branch

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



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.