

Unit 2. Pipelined Computers

Lecture 2.1 Pipelined instruction units.

J. Flich, P. López, V. Lorente,
A. Pérez, S. Petit, J.C. Ruiz, S. Sáez, J. Sahuquillo

Department of Computer Engineering
Universitat Politècnica de València



VIRTUAL TEACHING

Purpose:
Provision of the higher education public service
(art. 1 LOU)

Responsible:
Universitat Politècnica de València.

**Rights of access, rectification, deletion,
portability, limitation or opposition to the
treatment in accordance with privacy
policies:**
<http://www.upv.es/contenidos/DPD/>


Intellectual property:
Exclusive use in the virtual classroom
environment.

Dissemination, distribution or disclosure of
recorded classes, particularly on social networks
or services dedicated to sharing notes, is
prohibited.

Violation of this prohibition may generate
disciplinary, administrative or civil liability.



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA





Contents

- 1 Concept of Pipelining
- 2 The instruction cycle
- 3 Pipelining the instruction cycle
- 4 Hazards
- 5 Data hazards
- 6 Control hazards
- 7 Structural hazards
- 8 Exceptions

Bibliography

 John L. Hennessy and David A. Patterson.

Computer Architecture, Fifth Edition: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5
edition, 2012.

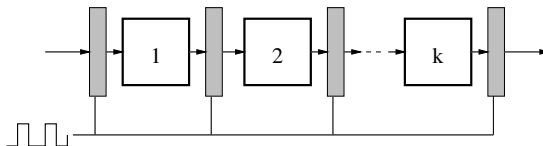
Contents

- 1 Concept of Pipelining
- 2 The instruction cycle
- 3 Pipelining the instruction cycle
- 4 Hazards
- 5 Data hazards
- 6 Control hazards
- 7 Structural hazards
- 8 Exceptions

1. Concept of Pipelining

Pipelining

- a process is decomposed into several subprocesses
- each subprocess is independently executed in an autonomous module
- each module works concurrently with the rest
- a system is pipelined in k stages:



→ Stage: module processing a subtask
+
latch register

1. Concept of Pipelining

Pipelining (cont.)

■ **Latches:**

They keep data stable during the time required by a module to perform its function.

- A clock synchronizes the advance of data through stages. The clock defines
 - when new data can enter the pipelined unit.
 - the time available for each stage to perform its function.

1. Concept of Pipelining

Pipelining (cont.)

■ Clock period

- Ideal case: same delay in all the modules

$$\tau = \frac{D}{k}$$

- D : original circuit delay
- k : number of stages
- Real case: modules with variable delays, latches and clock skew.

$$\tau = \max_{i=1}^k(\tau_i) + T_R + T_S \geq \frac{D}{k}$$

- τ_i : Module delay i .
- T_R : Inter-stage latch delay.
- T_S : Clock skew.

1. Concept of Pipelining

Benefits of pipelining

■ **Speed-up:**

$$S = \frac{T_{np}}{T_p} \approx \frac{nD}{n\tau} = \frac{D}{\tau}$$

- T_{np} : Time to process n instructions in the original (non-pipelined) unit.
- T_p : Idem for the pipelined unit.
- Ideal case: $\tau = \frac{D}{k}$.

$$S = \frac{D}{\tau} = k$$

- Real case: $\tau \geq \frac{D}{k}$.

$$S = \frac{D}{\tau} \leq k$$

1. Concept of Pipelining

Benefits of pipelining (cont.)

■ Throughput

- General expression:

$$\chi = \frac{n}{T}, \text{ where:}$$

- n : number of elements to process.
- T : time required to process n elements.
- Non-pipelined unit.

$$\chi_{np} = \frac{n}{T_{np}} = \frac{n}{nD} = \frac{1}{D} \text{ results/s}$$

- Pipelined unit:

$$\chi_p = \frac{n}{T_p} \approx \frac{n}{n\tau} = \frac{1}{\tau} \text{ results/s}$$

→ 1 result each τ seconds = 1 result per clock cycle.

Contents

- 1 Concept of Pipelining
- 2 The instruction cycle
- 3 Pipelining the instruction cycle
- 4 Hazards
- 5 Data hazards
- 6 Control hazards
- 7 Structural hazards
- 8 Exceptions

2. The instruction cycle

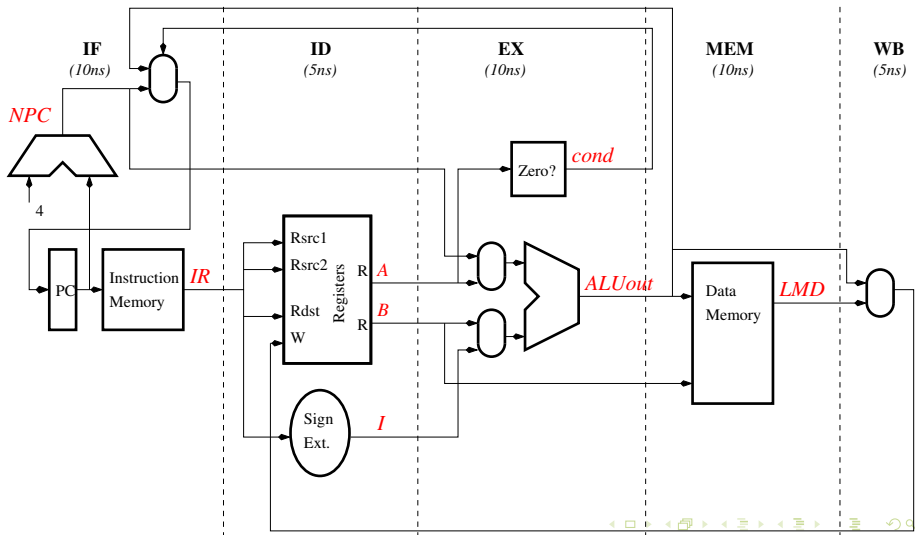
Example computer: Simplified MIPS

- Arithmetic instructions (reg-reg and reg-imm). Examples:
`dadd Rdst, Rsrc1, Rsrc2` & `daddi Rdst, Rsrc1, Imm`
- Loads and stores. Examples: `ld Rdst, Disp(Rsrc1)` & `sd Rsrc2, Disp(Rsrc1)`
- Conditional branches: `beqz Rsrc1, Disp` & `bnez Rsrc1, Disp`

	0 5	6 10	11 15	16 20	21 31
R	Codop	R _{src1}	R _{src2}	R _{dst}	Func (op)
	⇒ Instr. UAL reg-reg: R _{dst} ← R _{src1} op R _{src2}				
	0 5	6 10	11 15	16 31	
I	Codop	R _{src1}	R _{src2/dst}	Imm/Disp	
	⇒ ALU (reg-imm) instr.: R _{dst} ← R _{src1} op Imm				
	⇒ Load instr.: R _{dst} ← M[R _{src1} +Disp]				
	⇒ Store instr.: M[R _{src1} +Disp] ← R _{src2}				
	⇒ Conditional branches: if (R _{src1} =, ≠ 0) then PC ← PC+4+Disp				

2. The instruction cycle

MIPS datapath



2. The instruction cycle

MIPS Instruction cycle

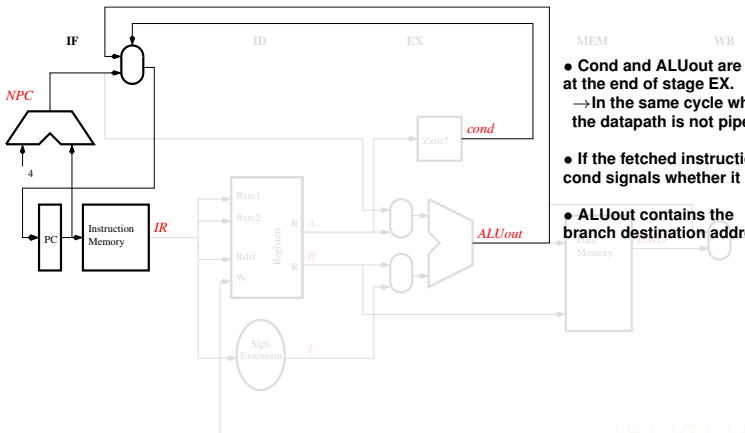
Stage	Example instructions			
	dadd Rdst,Rsrc1,Rsrc2 daddi Rdst,Rsrc1,Imm	ld Rdst,Disp(Rsrc1)	sd Rsrc2,Disp(Rsrc1)	beqz Rsrc1,Disp
IF	Read instruction from memory: $IR \leftarrow \text{Mem}[PC]$; Compute NPC: $NPC \leftarrow PC+4$; Following instruction: if cond then $PC \leftarrow \text{ALUout}$ else $PC \leftarrow NPC$;			
ID	Instruction decoding Read operands from register file: $A \leftarrow \text{Regs}[Rsrc1]$; $B \leftarrow \text{Regs}[Rsrc2]$; Extend the sign of the immediate/displacement field: $I \leftarrow \text{Sign ext. (Imm/Disp)}$;			
EX	Result computation: $\text{ALUout} \leftarrow A+(B \text{ o } I)$;	Address computation: $\text{ALUout} \leftarrow A+I$;		Compute dest. and cond.: $\text{ALUout} \leftarrow NPC+I$; $\text{cond} \leftarrow A=0?$;
MEM		Read data from memory: $\text{LMD} \leftarrow \text{Mem}[\text{ALUout}]$;	Write data to memory: $\text{Mem}[\text{ALUout}] \leftarrow B$;	
WB	Write back result to register file: $\text{Regs}[\text{Rdst}] \leftarrow \text{ALUout o LMD}$			

Contents

- 1 Concept of Pipelining
- 2 The instruction cycle
- 3 Pipelining the instruction cycle**
- 4 Hazards
- 5 Data hazards
- 6 Control hazards
- 7 Structural hazards
- 8 Exceptions

3. Pipelining the instruction cycle

Stage	Example instructions			
	dadd Rdst,Rsrc1,Rsrc2 daddi Rdst,Rsrc1,Imm	ld Rdst,Disp(Rsrc1)	sd Rsrc2,Disp(Rsrc1)	beqz Rsrc1,Disp
IF	Read instruction from memory: $IR \leftarrow Mem[PC]$; Compute NPC: $NPC \leftarrow PC+4$; Following instruction: if cond then $PC \leftarrow ALUout$ else $PC \leftarrow NPC$;			



- Cond and ALUout are available at the end of stage EX.

→ In the same cycle when the datapath is not pipelined.

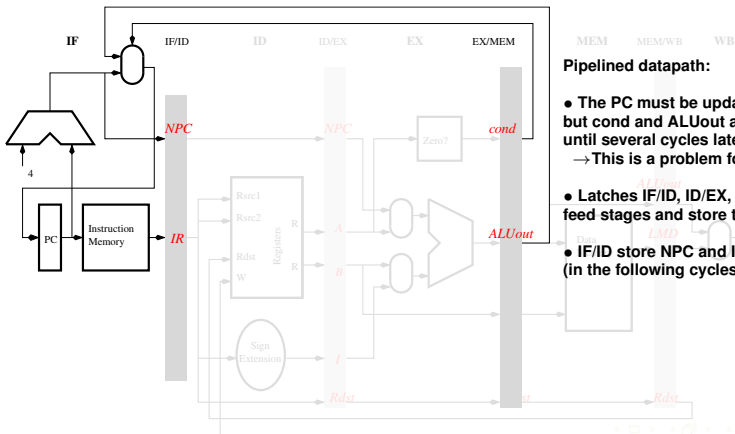
- If the fetched instruction is a branch, cond signals whether it must be taken or not.

- ALUout contains the branch destination address.

3. Pipelining the instruction cycle

Pipelining the instruction cycle

Stage	Example instructions			
	dadd Rdst,Rsrc1,Rsrc2	daddi Rdst,Rsrc1,Imm	ld Rdst,Disp(Rsrc1)	sd Rsrc2,Disp(Rsrc1) beqz Rsrc1,Disp
IF	Read instruction from memory: IF/ID . $IR \leftarrow \text{Mem}[PC]$; Compute NPC: IF/ID . $NPC \leftarrow PC+4$; Following instruction: if EX/MEM .cond then $PC \leftarrow \text{EX/MEM.ALUout}$ else $PC \leftarrow NPC$;			



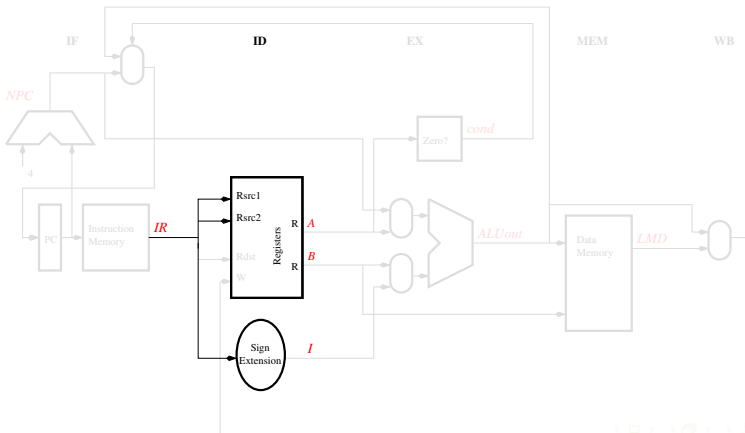
Pipelined datapath:

- The PC must be updated each cycle, but cond and ALUout are not available until several cycles later.
→ This is a problem for branches.
- Latches IF/ID, ID/EX, EX/MEM, MEM/WB feed stages and store their results.
- IF/ID store NPC and IR that will be later used (in the following cycles).

3. Pipelining the instruction cycle

Pipelining the instruction cycle

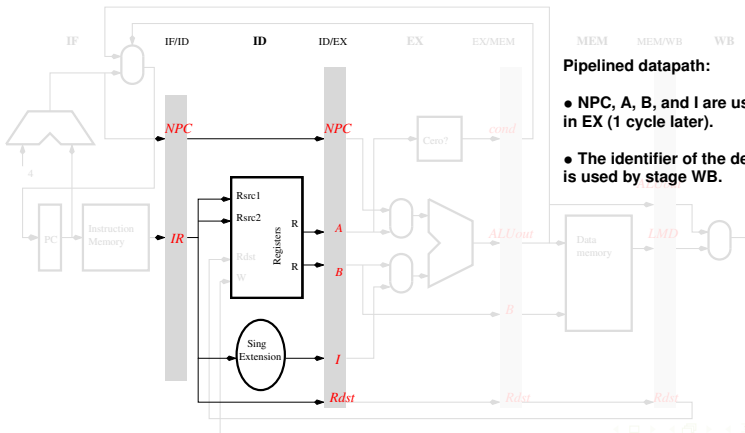
Stage	Example instructions			
	dadd Rdst,Rsrc1,Rsrc2	daddi Rdst,Rsrc1,Imm	ld Rdst,Disp(Rsrc1)	sd Rsrc2,Disp(Rsrc1) beqz Rsrc1,Disp
ID	Instruction decoding Read operands from register file: $A \leftarrow \text{Regs}[Rsrc1]$; $B \leftarrow \text{Regs}[Rsrc2]$; Extend the sign of the immediate/displacement field: $I \leftarrow \text{Sign ext. (Imm/Disp)}$;			



3. Pipelining the instruction cycle

Pipelining the instruction cycle

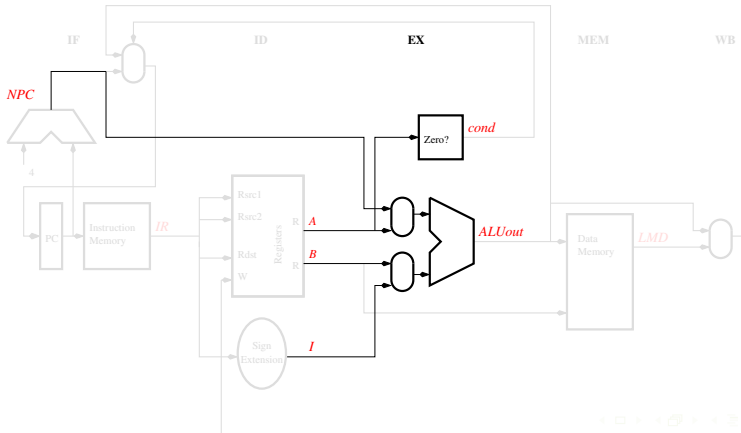
Stage	Example Instructions			
	dadd Rdst,Rsrc1,Rsrc2 daddi Rdst,Rsrc1,Imm	ld Rdst,Disp(Rsrc1)	sd Rsrc2,Disp(Rsrc1)	beqz Rsrc1,Disp
ID	Instruction decoding Read operands from register file: $ID/EX.A \leftarrow Regs[IF/ID.IR.Rsrc1]$; $ID/EX.B \leftarrow Regs[IF/ID.IR.Rsrc2]$; Extend sign of the immediate/displacement field: $ID/EX.I \leftarrow \text{Sign extension}(IF/ID.IR.Imm/Disp)$;			



3. Pipelining the instruction cycle

Pipelining the instruction cycle

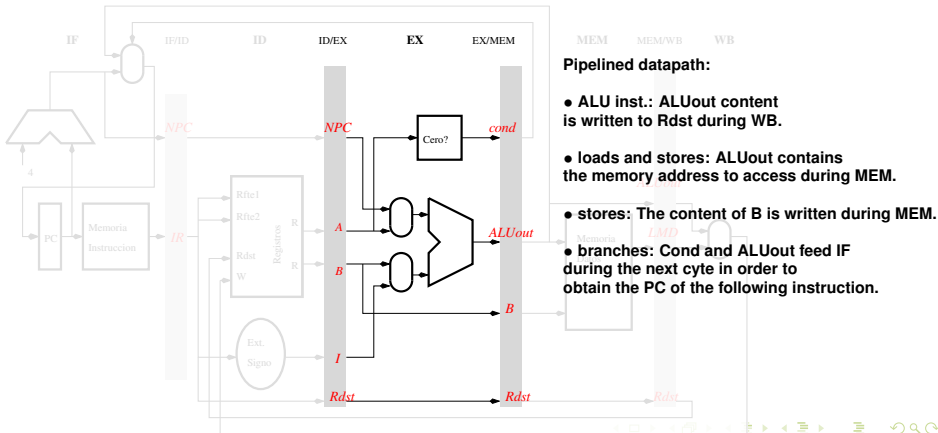
Stage	Example instructions			
	dadd Rdst,Rsrc1,Rsrc2 daddi Rdst,Rsrc1,Imm	ld Rdst,Disp(Rsrc1)	sd Rsrc2,Disp(Rsrc1)	beqz Rsrc1,Disp
EX	Result computation: $ALUout \leftarrow A+(B \circ I);$	Address computation: $ALUout \leftarrow A+I;$		Compute dest. and cond.: $ALUout \leftarrow NPC+I;$ $cond \leftarrow A=0?;$



3. Pipelining the instruction cycle

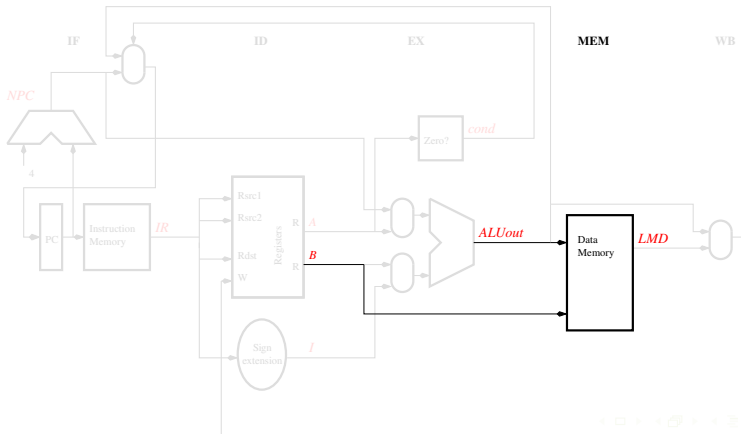
Pipelining the instruction cycle

Stage	Example Instructions			
	dadd Rdst,Rsrc1,Rsrc2 daddi Rdst,Rsrc1,Imm	ld Rdst,Disp(Rsrc1)	sd Rsrc2,Disp(Rsrc1)	beqz Rsrc1,Desp
EX	Result computation: EX/MEM.ALUout $\leftarrow \text{ID/EX.A} + \text{ID/EX.(B or I)};$	Address computation: EX/MEM.ALUout $\leftarrow \text{ID/EX.A} + \text{ID/EX.I};$		Compute dest. and cond.: EX/MEM.ALUout $\leftarrow \text{ID/EX.NPC} + \text{ID/EX.I};$ EX/MEM.cond $\leftarrow \text{IF/ID.A} = 0?;$



3. Pipelining the instruction cycle

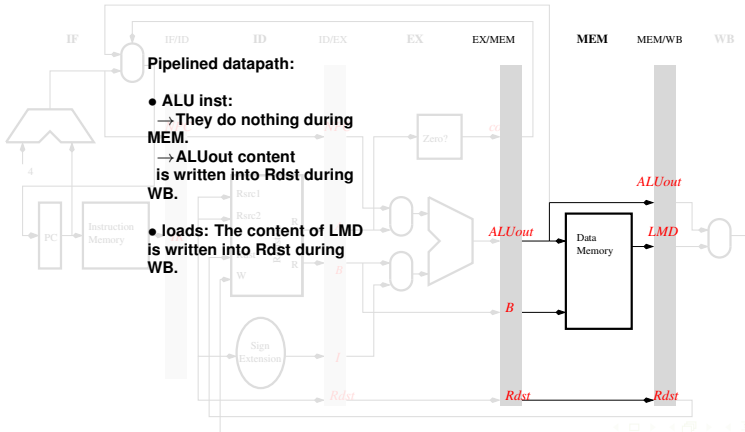
Stage	Example instructions			
	dadd Rdst,Rsrc1,Rsrc2 daddi Rdst,Rsrc1,Imm	ld Rdst,Disp(Rsrc1)	sd Rsrc2,Disp(Rsrc1)	beqz Rsrc1,Disp
MEM		Read data from memory: LMD \leftarrow Mem[ALUout];	Write data to memory: Mem [ALUout] \leftarrow B;	



3. Pipelining the instruction cycle

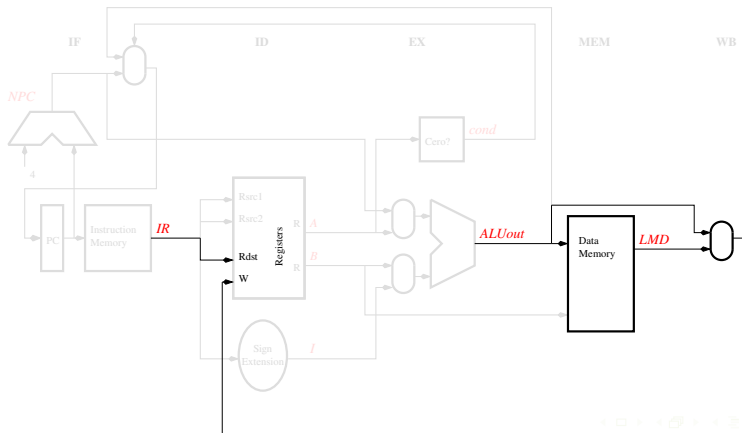
Pipelining the instruction cycle

Stage	Example instructions			
	dadd Rdst,Rsrc1,Rsrc2 daddi Rdst,Rsrc1,Imm	Rdst,Disp(Rsrc1)	Rsrc2,Disp(Rsrc1)	beqz Rsrc1,Disp
MEM		Read data from mem.: MEM/WB.LMD ← Mem[EX/MEM.ALUout];	Write data to Mem.: Mem[EX/MEM.ALUout] ← EX/MEM.B;	



3. Pipelining the instruction cycle

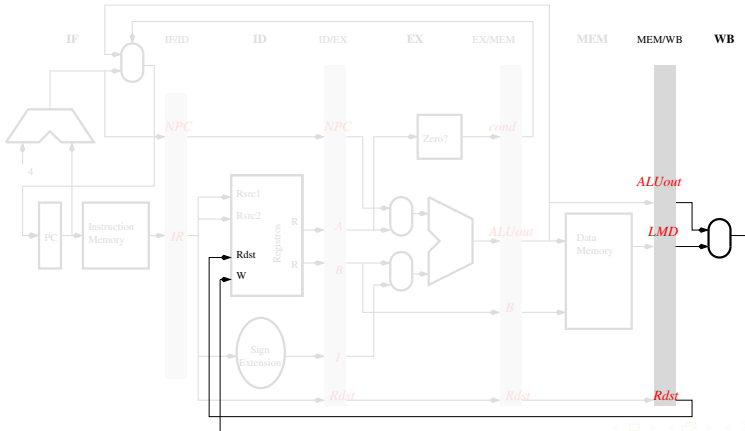
Stage	Example instructions			
	dadd Rdst,Rsrc1,Rsrc2	daddi Rdst,Rsrc1,Imm	ld Rdst,Disp(Rsrc1)	sd Rsrc2,Disp(Rsrc1)
WB	Write back result to register file: Regs[Rdst] ← ALUout or LMD			beqz Rsrc1,Disp



3. Pipelining the instruction cycle

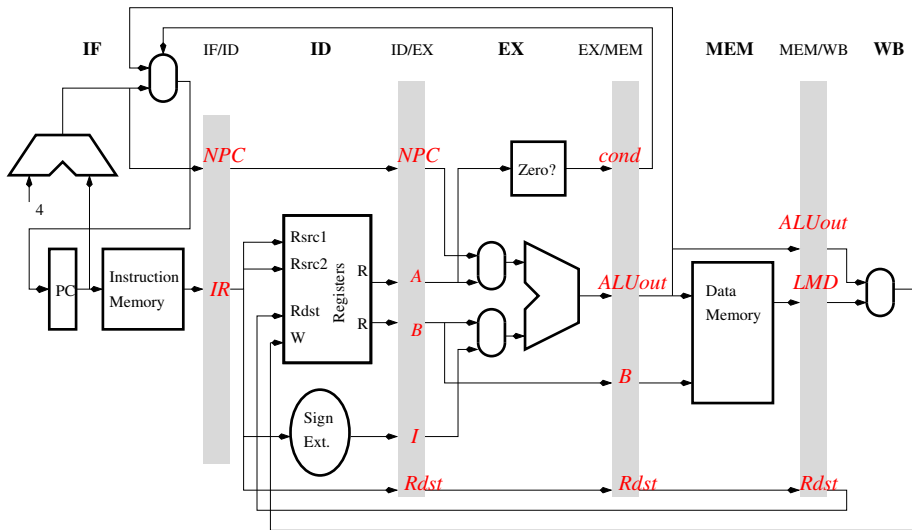
Pipelining the instruction cycle

Stage	Example instructions			
	dadd Rdst,Rsrc1,Rsrc2	daddi Rdst,Rsrc1,Imm	ld Rdst,Disp(Rsrc1)	sd Rsrc2,Disp(Rsrc1)
WB	Write result to the register file: $\text{Regs}[\text{MEM/WB.Rdst}] \leftarrow \text{MEM/WB.}(ALU\text{out or LMD})$			beqz Rsrc1,Disp



3. Pipelining the instruction cycle

Resulting Pipeline



3. Pipelining the instruction cycle

Pipelined instruction cycle

Stage	Example Instructions			
	dadd Rdst,Rsrc1,Rsrc2 daddi Rdst,Rsrc1,Imm	ld Rdst,Desp(Rsrc1)	sd Rsrc2,Desp(Rsrc1)	beqz Rsrc1,Desp
IF	Read instruction from memory: IF/ID.IR ← Mem[PC]; Compute NPC: IF/ID.NPC ← PC+4; Following instruction: if EX/MEM.cond then PC ← EX/MEM.ALUout else PC ← NPC;			
ID	Instruction decoding Read operands from register file: ID/EX.A ← Regs[IF/ID.IR.Rsrc1]; ID/EX.B ← Regs[IF/ID.IR.Rsrc2]; Extend the sign of the immediate/displacement field: ID/EX.I ← Sign extension(IF/ID.IR.Imm/Desp);			
EX	Result computation: EX/MEM.ALUout ← ID/EX.A + ID/EX.B (o l);	Address computation: EX/MEM.ALUout ← ID/EX.A + ID/EX.I ;	Compute dest. and cond. EX/MEM.ALUout ← ID/EX.NPC + ID/EX.I ; EX/MEM.cond ← IF/ID.A = 0?;	
MEM		Read data from Mem.: MEM/WB.LMD ← Mem[EX/MEM.ALUout];	Write data to Mem.: Mem[EX/MEM.ALUout] ← EX/MEM.B ;	
WB	Write result in the Register file: Regs[MEM/WB.Rdst] ← MEM/WB.ALUout (o LMD)			

3. Pipelining the instruction cycle

Execution of instructions

i	IF	ID	EX	MEM	WB				
$i + 1$		IF	ID	EX	MEM	WB			
$i + 2$			IF	ID	EX	MEM	WB		
$i + 3$				IF	ID	EX	MEM	WB	
$i + 4$					IF	ID	EX	MEM	WB

Speed-up:

	Non-pipelined	Pipelined
I	n	n
CPI	1	1
T	$\max(\text{branch } 25, \text{AL } 30, \text{store } 35, \text{load } 40) = 40 \text{ ns}$	$\max(5, 10) = 10 \text{ ns}$
$T_{ej} = I \cdot CPI \cdot T$	$n \cdot 1 \cdot 40 = 40n \text{ ns}$	$n \cdot 1 \cdot 10 = 10n \text{ ns}$

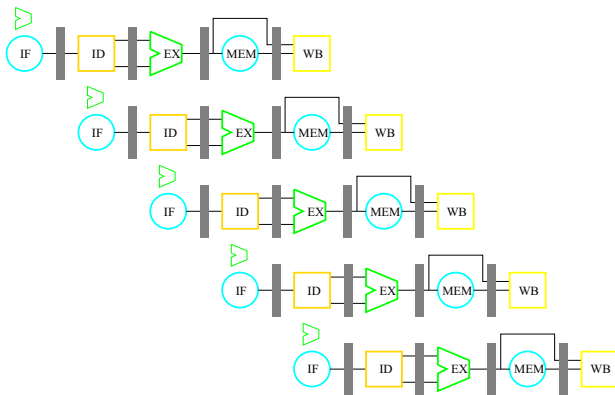
$$S = 4$$

The number of stages $k = 5$ is the speed-up upper-bound

3. Pipelining the instruction cycle

Hardware requirements:

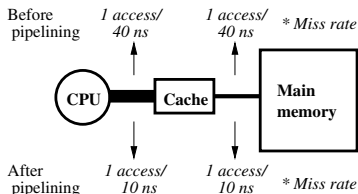
- In a single cycle there are 5 instructions simultaneously in execution → avoid hazards in functional units.



3. Pipelining the instruction cycle

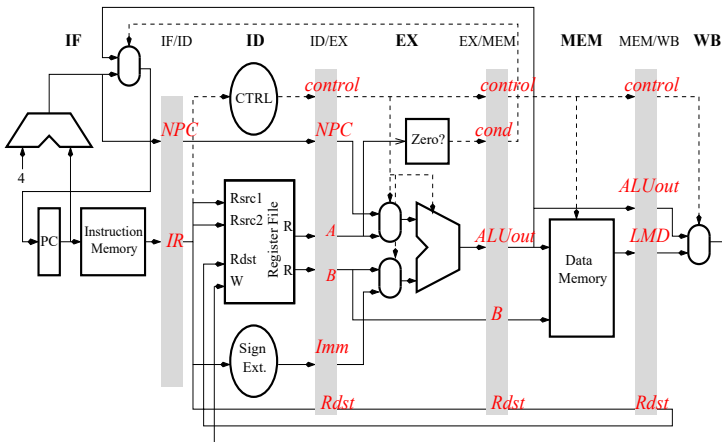
Hardware requirements: (cont.)

- Separated instruction (IF) and data (MEM) caches.
- Register file with two simultaneous Read (ID) and one write (WB) ports.
- *Cache* access time does not vary, but the required bandwidth is 4 times higher:



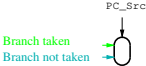
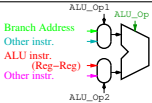

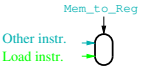
3. Pipelining the instruction cycle

Pipelined datapath



3. Pipelining the instruction cycle

Control signals

IF	ID	EX	MEM	WB
Mem_Read PC_Src	Reg_Read	ALU_Op ALU_Op1 ALU_Op2	Mem_Read Mem_Write	Reg_Write Mem_to_Reg
				

- Control signals required in stages EX, MEM and WB are all generated in stage ID.

Contents

- 1 Concept of Pipelining
- 2 The instruction cycle
- 3 Pipelining the instruction cycle
- 4 Hazards**
- 5 Data hazards
- 6 Control hazards
- 7 Structural hazards
- 8 Exceptions

4. Hazards

Concept and classification

Hazard : Situation leading to the execution of some instructions generating results that are not consistent with the ones produced by the non-pipelined datapath \Rightarrow Lost of binary compatibility. Hazards are originated by two or more instructions simultaneously present in the pipelined instruction unit.

Types of hazards:

Data

The result of one instruction is used as input data in the following one(s).

Control

Branch instructions modify the flow of instructions

Structural

Two or more instructions want to use the same resource

4. Hazards

Hazard solutions

Stall insertion Stop the instruction originating a hazard, and the following ones, but continue with the execution of those preceding such instructions (otherwise the hazard will never disappear)

- During these cycles no new instructions are fetched (*stalls*).
- Performance is degraded due to the insertion of stalls:

$$CPI_s = CPI_{ideal} + \frac{\text{stalls}}{\text{instruction}} = 1 + \frac{\text{stalls}}{\text{instruction}} > 1$$

4. Hazards

Hazard solutions (cont.)

Modification of the Datapath: Modify the *hardware* to detect and dynamically solve hazards. The modification can:

- Reduce the number of stalls required to solve the problem.
- Completely solve the hazard (which is sometimes impossible).

Modification of the ISA: Avoid the occurrence of the problem by avoiding the generation of certain sequences of instructions. In order to avoid them, the compiler can insert `NOP` instructions or reorder independent instructions.

- Drawback: Binary compatibility can be lost.

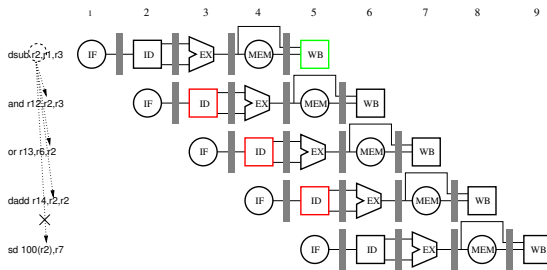
Contents

- 1 Concept of Pipelining
- 2 The instruction cycle
- 3 Pipelining the instruction cycle
- 4 Hazards
- 5 Data hazards**
- 6 Control hazards
- 7 Structural hazards
- 8 Exceptions

5. Data hazards

Causes

Some instructions rely on the results of previous ones → pipelining the instruction unit can invert the order of access to operands

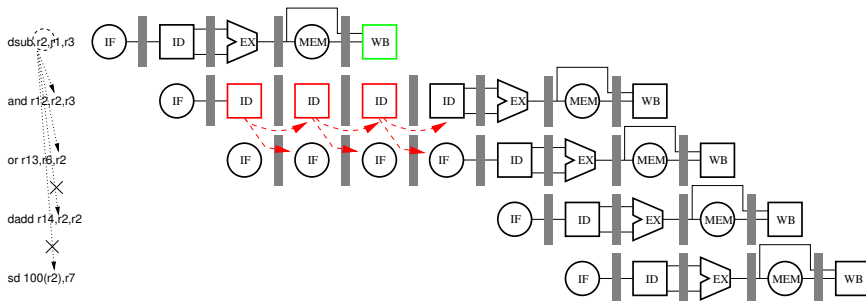


Hazards can involve up to 4 consecutive instructions

5. Data hazards

Stall insertion

Delay those operations originating the hazard



- 3 stalls for every two dependent and consecutive instructions.
- 2 stalls for every two dependent instr. separated by 1 instruction.
- 1 stall for every two dependent instr. separated by 2 instructions.
- significant loss of performance

5. Data hazards

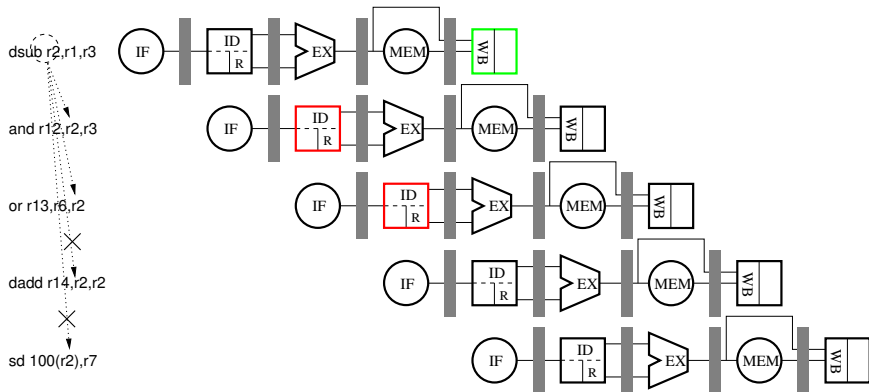
Reduction of the number of instructions involved in the hazard

⇒ Datapath modification

- Multiport register file supporting simultaneous read and write
- Modification of the access time to the register file:
 - Register read → Second part of the cycle
 - Register write → First part of the cycle.
- There is no problem if the register file access time is \leq half of the clock period.
- Simplifies the design of the register file: Simultaneous read and write operations are not required.

5. Data hazards

Reduction of the number of instructions involved in the hazard (cont.)



⇒ The number of involved instructions is 3: **dsusb**, **and** **y or**.

→ Penalty generated by dependent consecutive instructions: 2 *stalls*

5. Data hazards

Insertion of 2 stalls

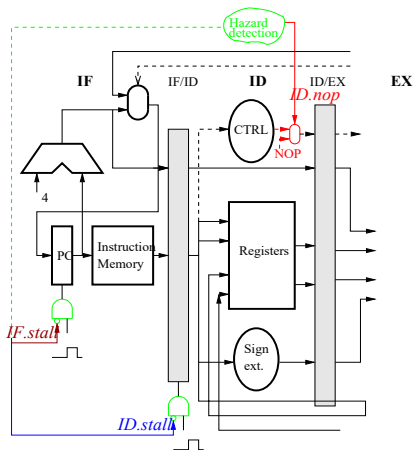
dsub r2 , r1, r3	IF	ID	EX	ME	WB						
and r12, r2 , r3		IF	id	id	ID	EX	ME	WB			
or r13, r6, r2			if	if	IF	ID	EX	ME	WB		
dadd r14, r2, r2						IF	ID	EX	ME	WB	
sd 100(r2), r7							IF	ID	EX	ME	WB

- Detect those situations (in bold) requiring stall insertion.
- In order to insert a stall:
 - Set control signals from ID to EX as they were the ones of a NOP instruction.
 - Preserve instruction in IF and ID.

5. Data hazards

Insertion of 2 stalls (cont.)

Control Logic



5. Data hazards

Insertion of 2 stalls (cont.)

■ First cycle:

```
if ((ID/EX.IR.CODOP = "ALU") and
    (IF/ID.IR.CODOP = "ALU") and
    ((ID/EX.IR.Rdst = IF/ID.IRsrc1) or
     (ID/EX.IR.Rdst = IF/ID.IRsrc2))
then
    IF.stall, ID.stall, ID.nop
```

■ Second cycle:

```
if ((EX/ME.IR.CODOP = "ALU") and
    (IF/ID.IR.CODOP = "ALU") and
    ((EX/ME.IR.Rdst = IF/ID.IRsrc1) or
     (EX/ME.IR.Rdst = IF/ID.IRsrc2))
then
    IF.stall, ID.stall, ID.nop
```

5. Data hazards

Forwarding

Problem: 2 *stalls* between consecutive instructions due to a data hazard

dsub r2 , r1, r3	IF	ID	EX	ME	WB				
and r12, r2 , r3		IF	id	id	ID	EX	ME	WB	
or r13, r6, r2			if	if	IF	ID	EX	ME	WB

Consider the hazard between the 2 first instructions if no stalls were inserted:

	1	2	3	4	5	
dsub r2 , r1, r3	IF	ID	EX	ME	WB	
and r12, r2 , r3		IF	ID	EX	ME	WB

- and requires the data at the beginning of cycle 4 (its EX stage)
- dsub provides a result at the beginning of cycle 4 (its MEM stage)

5. Data hazards

Forwarding (cont.)

⇒ datapath modification

Add a bus from the ALU output (MEM stage) to its inputs (EX stage) + control logic ⇒ “short-circuit” from MEM to EX.

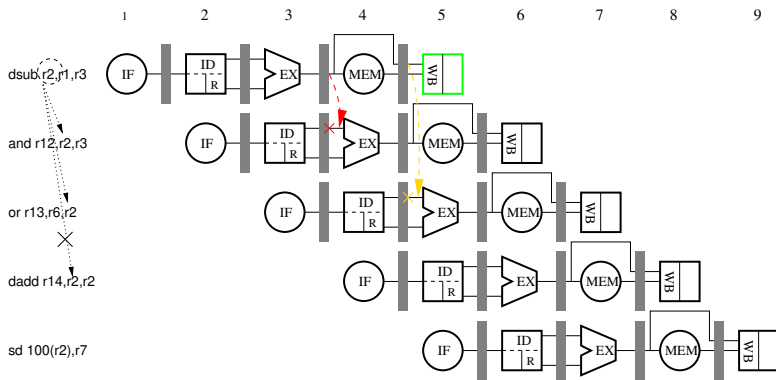
When the third instruction is considered, the hazard can be solved in a similar way, i.e. implementing a short-circuit between WB and EX.

dsub r2,r1,r3	IF	ID	EX	ME	WB	
and r12,r2,r3	IF	ID	EX	ME	WB	
or r13,r6,r2		IF	ID	EX	ME	WB

5. Data hazards

Forwarding (cont.)

Following instructions requiring the data will not need any shortcircuit.

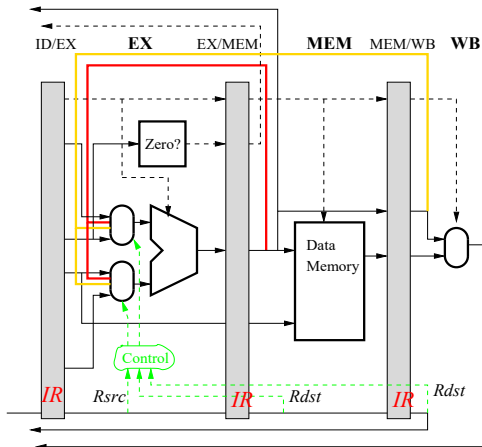


5. Data hazards

Forwarding (cont.)

Forwarding implementation

- Multiplexors in the ALU input



5. Data hazards

Forwarding (cont.)

Control logic

■ Shortcircuit from MEM to EX:

■ Rsrc1:

```
if ((EX/MEM.IR.CODOP = "ALU") and
    (ID/EX.IR.CODOP = "ALU") and
    (EX/MEM.IR.Rdst = ID/EX.IR.Rsrc1))
then
    Shortcircuit MEM-to-EX (ALU high input)
```

■ Rsrc2:

```
if ((EX/MEM.IR.CODOP = "ALU") and
    (ID/EX.IR.CODOP = "ALU") and
    (EX/MEM.IR.Rdst = ID/EX.IR.Rsrc2))
then
    Shortcircuit MEM-to-EX (ALU low input)
```


5. Data hazards

Forwarding (cont.)

■ Shorrcircuit from WB to EX:

■ Rscr1:

```
if ((MEM/WB.IR.CODOP = "ALU") and
    (ID/EX.IR.CODOP = "ALU") and
    (MEM/WB.IR.Rdst = ID/EX.IR.Rsrc1))
then
    Shorrcircuit WB-to-EX (ALU high input)
```

■ Rsrc2:

```
if ((MEM/WB.IR.CODOP = "ALU") and
    (ID/EX.IR.CODOP = "ALU") and
    (MEM/WB.IR.Rdst = ID/EX.IR.Rsrc2))
then
    Shorrcircuit WB-to-EX (ALU low input)
```

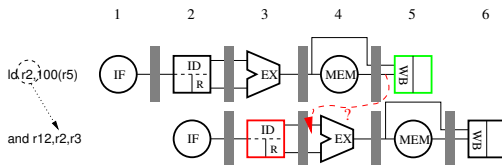
5. Data hazards

Consecutive and *Load*-dependent instruction

Consider the hazard between the following instructions:

ld r2,100(r5)
and r12,r2,r5

- When does **and** require the data? In cycle 4 (EX stage)
- When is the result of **ld** available? In cycle 5 (WB stage)

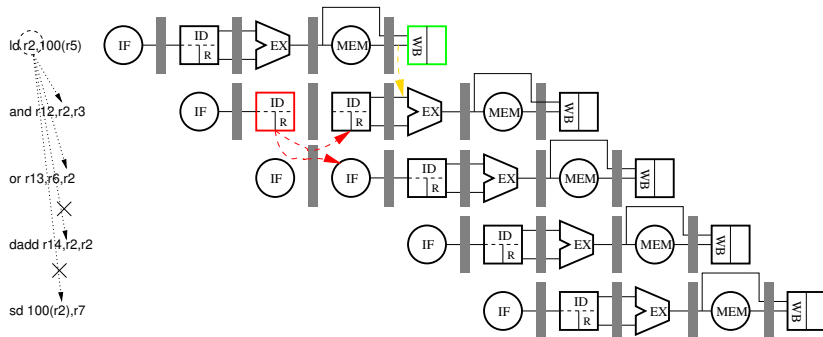


⇒ Cannot be solved using only data forwarding

5. Data hazards

Consecutive and *Load*-dependent instruction (cont.)

Even using a shortcircuit from WB to EX, it is necessary to insert 1 stall



5. Data hazards

Consecutive and *Load*-dependent instruction (cont.)

Required control logic to insert the stall

```
if (ID/EX.IR.CODOP = "LOAD") and  
    (IF/ID.IR.CODOP = "ALU") and  
    ((ID/EX.IR.Rdst = IF/ID.IR.src1) or  
     (ID/EX.IR.Rdst = IF/ID.IR.src2))  
then  
    ID.stall, IF.stall, ID.nop
```

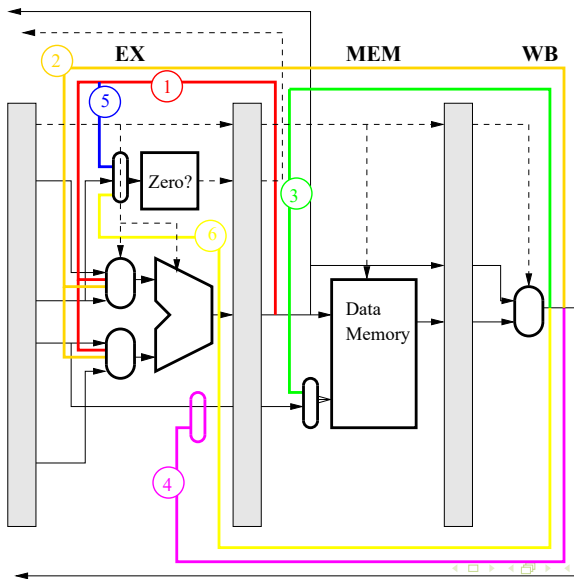
5. Data hazards

Summary of shortcircuits and stalls

Type of instructions	Example	Shortcircuit	Stalls	Fig.
ALU - ALU	DADD R1,R2,R3	MEM to EX	0	1
	DSUB R4,R1,R5 AND R7,R1,R6	WB to EX	0	2
Load - ALU	LD R1,20(R2) DADD R3,R1,R4	WB to EX	1	2
ALU - Load/Store	DADD R1,R2,R3	MEM to EX	0	1
	LD R2,20(R1) LD R3,40(R1)	WB to EX	0	2
ALU - Store	DADD R1,R2,R3	WB to MEM	0	3
	SD R1,20(R2) SD R1,40(R2)	WB to EX	0	4
Load - Store	LD R1,20(R3)	WB to MEM	0	3
	SD R1,20(R2) SD R1,40(R2)	WB to EX	0	4
Load - Load/Store	LD R1,30(R3) LD R2,20(R1)	WB to EX	1	2
ALU - Branch	DSLT R1,R2,R3 BEQZ R1,loop	MEM to EX	0	5
ALU - Branch	DSLT R1,R2,R3 ...			
	BEQZ R1,loop	WB to EX	0	6
Load - Branch	LD R1,20(R2) BEQZ R1,loop	WB to EX	1	6

5. Data hazards

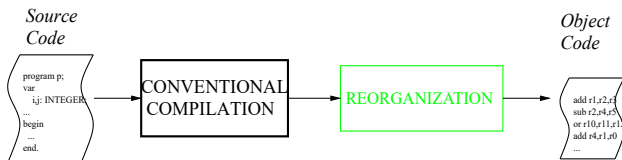
Summary of shortcircuits and stalls (cont.)



5. Data hazards

Reorganization of code

⇒ The compiler has an additional stage: code reorganization



⇒ The goal is to avoid stall by reorganizing the code. Example:

Código convencional	Código reorganizado
LD Rb,b	LD Rb,b
LD Rc,c	LD Rc,c
DADD Ra,Rb,Rc	LD Re,e
LD Re,e	DADD Ra,Rb,Rc
LD Rf,f	LD Rf,f
DSUB Rd,Re,Rf	SD Ra,a
SD Ra,a	DSUB Rd,Re,Rf
SD Rd,d	SD Rd,d
8 ins + 2 stalls = 10 ciclos	8 ins = 8 ciclos

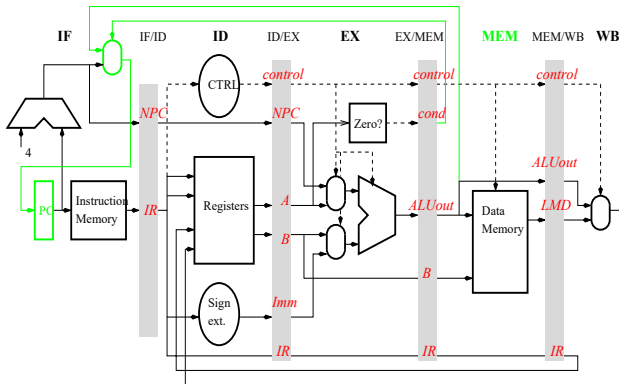
Contents

- 1 Concept of Pipelining
- 2 The instruction cycle
- 3 Pipelining the instruction cycle
- 4 Hazards
- 5 Data hazards
- 6 Control hazards**
- 7 Structural hazards
- 8 Exceptions

6. Control hazards

Causes

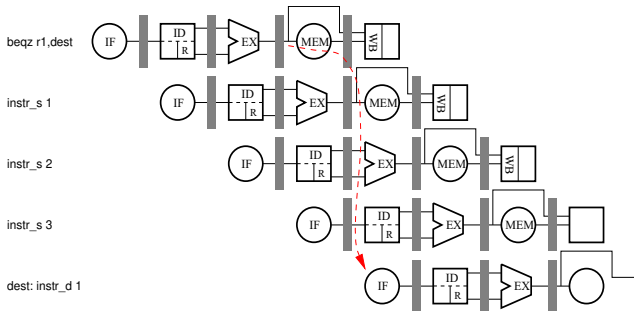
In the instruction flow, some instructions modify the value of the PC. Branch instructions modify the PC in the MEM stage.



6. Control hazards

Causes (cont.)

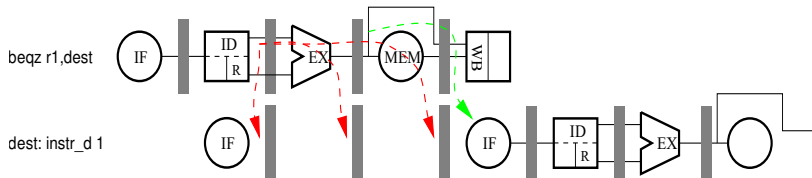
When this cycle is reached, 3 instructions have already been issued:



6. Control hazards

Stall insertion

Insert stalls *whenever* a branch instruction is decoded:



3 stalls → loss of performance

6. Control hazards

Stall insertion (cont.)

Control logic for the 3 stalls to insert

- Disable IF stage during 3 cycles, thus sending NOP to ID:

```
beqz r1,dest    IF ID EX ME WB  
<dest>         if if if IF ID EX ME WB
```

```
if ((IF/ID.IR.CODOP = "Branch") or  
    (ID/EX.IR.CODOP = "Branch") or  
    (EX/MEM.IR.CODOP = "Branch"))  
then  
    IF.stall, IF.nop
```

6. Control hazards

(Fixed) prediction

⇒ Datapath modification

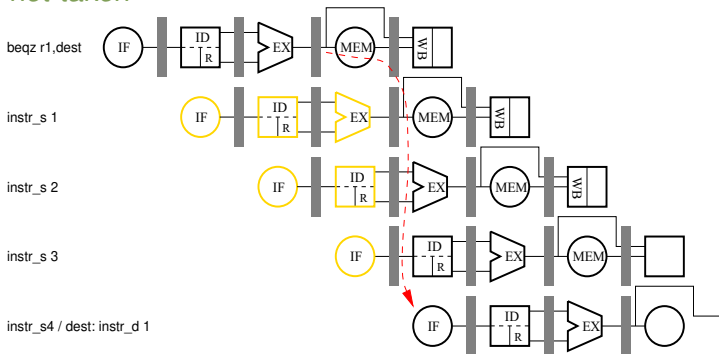
Predict-not taken Assume the branch is not taken → Instructions following the branch are correct.
At the end, if the branch is taken, these 3 instructions must be cancelled.
IMPORTANT: These instructions must not modify the computer state

Predict-taken Assume the branch is taken → once the destination address is known, new instructions are fetched
If the branch is finally not taken, such instructions are aborted.
It is only useful if the destination address is known *before* the branch condition → not useful in the case of the MIPS.

6. Control hazards

(Fixed) prediction (cont.)

Predict-not-taken



Control logic

```
if (EX/MEM.cond) then  
    IF.nop, ID.nop, EX.nop
```

6. Control hazards

Reducing the branch latency

⇒ Datapath modification

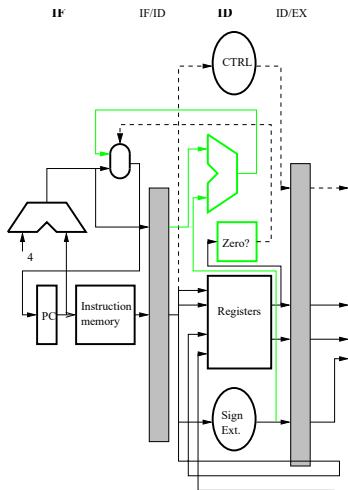
Reduce the number of cycles between fetching the branch instruction and computing the branch destination.

- Update the PC during EX → Num. of cycles = 2.
⇒ The PC for the next cycle is computed during the branch EX stage.
- Update the PC during ID → Num. of cycles = 1.
⇒ The PC for the next cycle is computed during the branch ID stage. This requires:
 - Move the computation of the branch destination address from EX to ID → additional adder required
 - Move the evaluation of the condition from EX to ID

6. Control hazards

Reducing the branch latency (cont.)

Updating the PC in the ID stage



6. Control hazards

Reducing the branch latency (cont.)

Control logic for updating the PC in the ID stage

- Assuming *predict-not-taken*, if the branch is not taken, fetch the next instruction.
- But if the branch is taken, fetched instruction must be cancelled during IF. → A NOP is provided to the ID stage.

```
beqz r1,dest    IF ID EX ME WB
<sgte>          ID X
<dest>          IF ID EX ME WB
if (IF/ID.IR.CODOP= "Branch")
then
    if (Regs[IF/ID.IR.Rsrc] op 0)
    then
        IF.nop
        PC <- IF/ID.NPC + IF/ID.Imm
    else
        PC <- PC + 4
```

6. Control hazards

Reducing the branch latency (cont.)

Impact on the clock period

Stage ID

register access time +
delay for evaluating the condition +
selection +
PC update

→ ID may become the slowest stage

⇒ branch condition must be simple (= and \neq).

This modification may be incompatible with the requirement of reading the registers in half a cycle.

6. Control hazards

Reducing the branch latency (cont.)

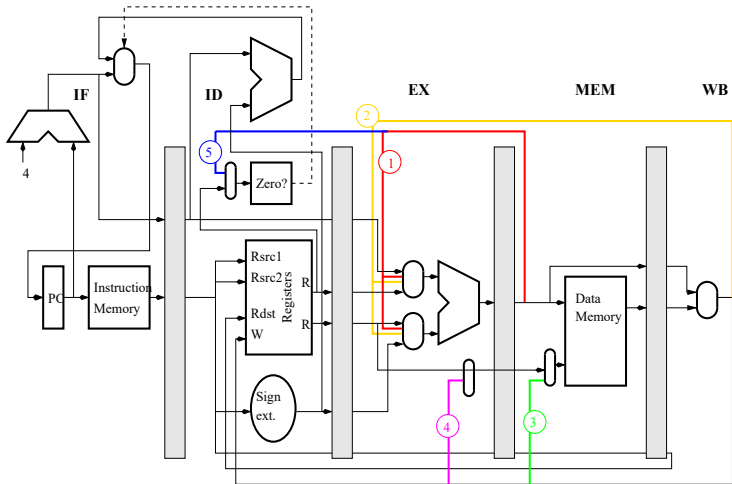
Impact on shortcircuits and stalls

Instructions	Example	Shortcircuit	<i>stalls</i>	Fig.
ALU - ALU	DADD R1,R2,R3	MEM to EX	0	1
	DSUB R4,R1,R5 AND R7,R1,R6	WB to EX	0	2
Load - ALU	LD R1,20(R2) DADD R3,R1,R4	WB to EX	1	2
ALU - Load/Store	DADD R1,R2,R3	MEM to EX	0	1
	LD R2,20(R1) LD R3,40(R1)	WB to EX	0	2
ALU - Store	DADD R1,R2,R3	WB to MEM	0	3
	SD R1,20(R2) SD R1,40(R2)	WB to EX	0	4
Load - Store	LD R1,20(R3)	WB to MEM	0	3
	SD R1,20(R2) SD R1,40(R2)	WB to EX	0	4
Load - Load/Store	LD R1,30(R3) LD R2,20(R1)	WB to EX	1	2
ALU - Branch	DSLT R1,R2,R3 BEQZ R1,loop	MEM to EX MEM to ID	0 1	5
ALU - Branch	DSLT R1,R2,R3 ... BEQZ R1,loop	WB to EX MEM to ID	0	5
Load - Branch	LD R1,20(R2) BEQZ R1,loop	WB to EX Due to BR	1 2	—
Load - Branch	LD R1,20(R2) ... BEQZ R1,loop	Due to BR	0 1	—

6. Control hazards

Reducing the branch latency (cont.)

Impact on shortcircuits and stalls



6. Control hazards

Delayed branch

⇒ Modification of the ISA (Instruction Set Architecture)

The compiler *must* place just after branches instructions that will be executed despite those branches will be taken or not.

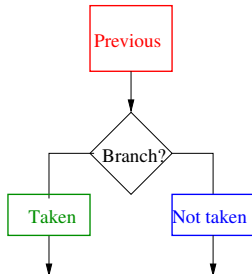
- Since these instructions will be always executed, it is unnecessary to cancel them or insert stalls.
- Normally, instructions are selected from those *s preceding* the branch that are independent from the branch condition.

Branch delay slot

- Number of instructions after the branch that must be always executed despite the branch is taken or not.
 - It is equal to the branch latency.
 - 1 cycle if the PC is updated during ID.

6. Control hazards

Delayed branch (cont.)



Conventional code:



Code with delayed branch:



Example:

Conventional	Delayed branch
ADD Rc, Ra, Rb	BEQZ Ra, dst
BEQZ Ra, dst	ADD Rc, Ra, Rb
INSTR1	INSTR1
dst: INSTR2	dst: INSTR2

6. Control hazards

Delayed branch (cont.)

The delayed branch in current instruction sets

The delayed branch is not used since early 90s. Some reasons:

- If the *branch delay slot* is big (more than 1 instruction), the compiler has many difficulties to find useful instructions to fill the slot. This happens when:
 - The branch latency is big (i.e. the datapath has many stages).
 - Multiple instructions are issued per cycle (more in Lecture 2.5).
- It conditions the datapath implementing the instruction set (lack of flexibility). For instance, a different datapath design may vary the branch latency, thus losing the compatibility.
- There are today better alternatives, such as dynamic branch prediction (more in Lecture 2.3).

Contents

- 1 Concept of Pipelining
- 2 The instruction cycle
- 3 Pipelining the instruction cycle
- 4 Hazards
- 5 Data hazards
- 6 Control hazards
- 7 Structural hazards**
- 8 Exceptions

7. Structural hazards

Causes

- The hardware does not allow all possible combinations between instructions in the unit.
 - A resource has not been replicated enough
- **Example:** Processor with single instruction–data cache.
 - The MEM stage of load/store instructions may collide with the IF stage of instructions fetched 3 cycles later

7. Structural hazards

Solutions

Datapath modifications

Replicate the resource in order to enable such combination of instructions.

- Example: *Harvard* architecture: it uses separated instruction and data caches.
- Cost increase
- Replicating the resource is not always possible or makes sense

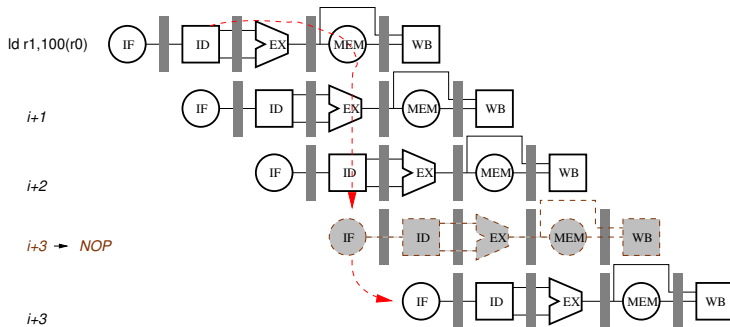
Stall insertion

Delay the instructions generating the hazard

- *stalls* → Loss of performance
- ⇒ The best approach depends on the % of each type of structural hazard

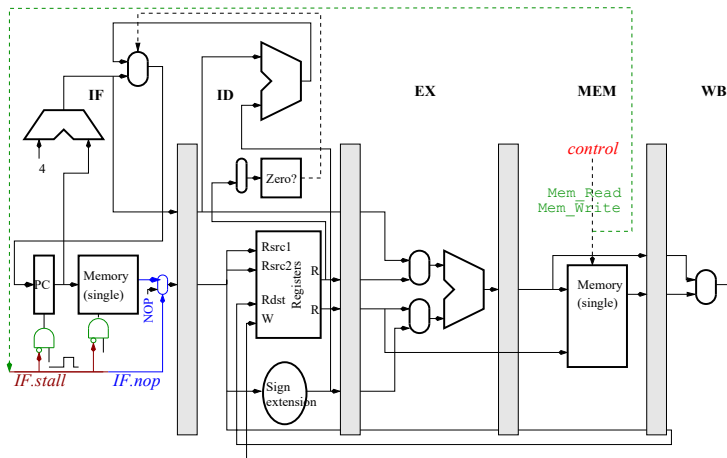
7. Structural hazards

Stall insertion



7. Structural hazards

Stall insertion (cont.)



7. Structural hazards

Stall insertion (cont.)

Control logic

When a load/store instruction is in MEM

- No instruction fetch is performed
- The instruction in IF is stalled
- A NOP is delivered to stage ID

```
if ((EX/MEM.Mem_Read) or (EX/MEM.Mem_Write))  
then  
    IF.stall,  IF.nop
```

Contents

- 1 Concept of Pipelining
- 2 The instruction cycle
- 3 Pipelining the instruction cycle
- 4 Hazards
- 5 Data hazards
- 6 Control hazards
- 7 Structural hazards
- 8 Exceptions**

8. Exceptions

Concept and classification

Names: Interrupt, **exception**.

Types:

- Synchronous vs. asynchronous. It is synchronous if the event is triggered at the same location on every program execution.
- User-driven vs. raised to the user.
- Maskable vs. unmaskable.
- During the execution of an instruction vs. between instructions.
- Continue the program execution vs. end the program.

An exception is an implicit conditional branch → Whenever the exception happens, the exception handler must be executed

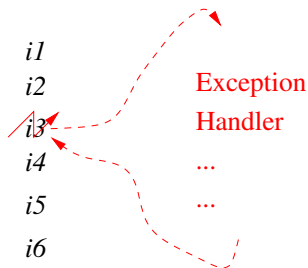
8. Exceptions

MIPS possible exceptions

<i>Stage</i>	<i>Exceptions</i>
IF	Instruction page fault, Misaligned access Violation of protection, E/S request
ID	Illegal instruction, E/S request
EX	Arithmetic exception, E/S request
MEM	Data page fault, Misaligned access Violation of protection, E/S request
WB	E/S request

8. Exceptions

Exceptions in conventional computers



Correct sequence: ... *i1*, *i2*, *i3* - handler - *i3*, *i4*, *i5*, *i6* ...

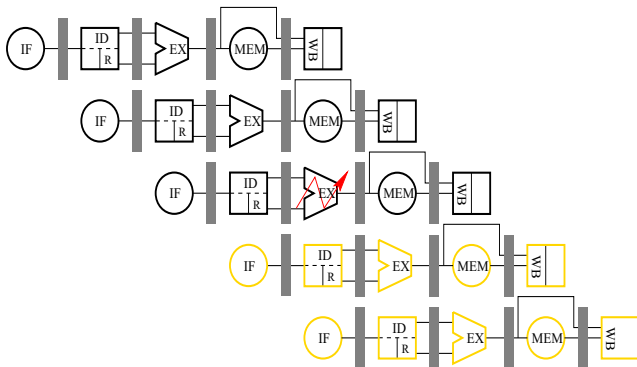
8. Exceptions

Exceptions in pipelined computers

Several instructions under execution when the exception occurs.

Problem:

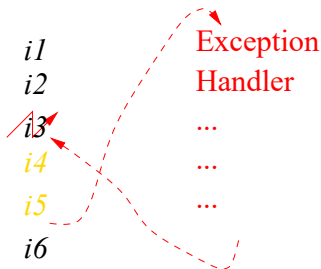
- There are instructions following the one raising the exception.



8. Exceptions

Exceptions in pipelined computers (cont.)

Behaviour is incorrect:



Sequence: ... *i1*, *i2*, *i3*, *i4*, *i5* - handler - *i3*, *i4*, *i5*, *i6* ...

- The PC of instructions is only used during IF.

First pipelined machines terminated the execution of programs by printing the PC of the current instruction in IF

⇒ they signaled *approximately* the instruction originating the exception
→ *imprecise* exceptions.

8. Exceptions

Precise exceptions in pipelined instruction units

A computer supports a *precise* behaviour in the presence of an exception if:

- Instructions preceding the one generating the exception correctly finish their execution.
- The instruction raising the exception and the following ones are aborted.
- After handler completion it is possible to restart the program from the instruction originating the exception.

⇒ It is possible to identify the instruction raising the exception.

⇒ The behavior is identical to the one exhibited by a non-pipelined computer.

8. Exceptions

Implementation of precise exceptions in the MIPS

Requirements:

- More than one exception (up to 5) can be raised
- ... in the same or in different clock cycles.
- It is necessary to take into account how the branch delay technique works.

Idea: Ensure a natural order when handling exceptions

⇒ Instructions must reach the last stage of the pipeline in order.

- 1 Each instruction entering the unit is related to a register with as many bits as stages in the pipeline able to raise an exception. Such register travels through the pipeline together with the instruction.

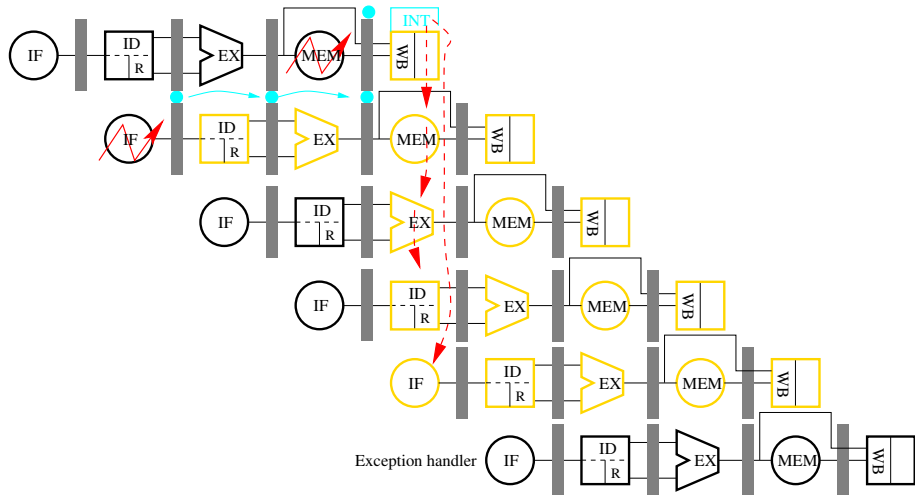
8. Exceptions

Implementation of precise exceptions in the MIPS (cont.)

- 2 If an exception is raised, the register bit of the corresponding stage becomes “1”. At the same time, the instruction generating the exception becomes a `NOP`
- 3 During the last stage, register's bits are checked.
If any bit is set, the following instructions become `NOP` and a `TRAP` is generated in the IF stage for the following cycle.
- 4 Save the PC of the instruction originating the exception.
- 5 The exception handler takes the control.
- 6 Once the handler ends its execution, the PC is restored, thus following the execution from that point.

8. Exceptions

Implementation of precise exceptions in the MIPS (cont.)



8. Exceptions

Implementation of precise exceptions in the MIPS (cont.)

