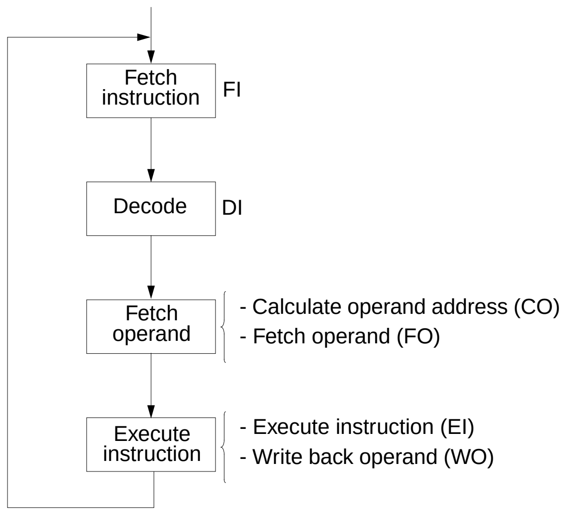
INSTRUCTION PIPELINING (I)

The Instruction Cycle

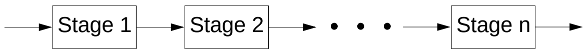

Fetch instruction
FI
Decode
DI
Fetch operand
- Calculate operand address (CO)
- Fetch operand (FO)
Execute instruction
- Execute instruction (EI)
- Write back operand (WO)

Instruction Pipelining

- Instruction execution is extremely complex and involves several operations which are executed *successively* (see slide 2). This implies a large amount of hardware, *but only one part of this hardware works at a given moment*.

- Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. This is solved without additional hardware but only by letting different parts of the hardware work for different instructions at the same time.

- The pipeline organization of a CPU is similar to an assembly line: the work to be done in an instruction is broken into smaller steps (pieces), each of which takes a fraction of the time needed to complete the entire instruction. Each of these steps is a *pipe stage* (or a *pipe segment*).
- Pipe stages are connected to form a pipe:



- The time required for moving an instruction from one stage to the next: a *machine cycle* (often this is one clock cycle). The execution of one instruction takes several machine cycles as it passes through the pipeline.

Acceleration by Pipelining

Two stage pipeline: FI: fetch instruction
EI: execute instruction

Clock cycle →	1	2	3	4	5	6	7	8
Instr. i	FI	EI						
Instr. i+1		FI	EI					
Instr. i+2			FI	EI				
Instr. i+3				FI	EI			
Instr. i+4					FI	EI		
Instr. i+5						FI	EI	
Instr. i+6							FI	EI

We consider that each instruction takes execution time T_{ex}.

Execution time for the 7 instructions, with pipelining:
$(T_{ex}/2)*8 = 4*T_{ex}$

Acceleration by Pipelining (cont'd)

Six stage pipeline (see also slide 2):

FI: fetch instruction	FO: fetch operand
DI: decode instruction	EI: execute instruction
CO: calculate operand address	WO:write operand

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
Instr. i	FI	DI	CO	FO	EI	WO						
Instr. i+1		FI	DI	CO	FO	EI	WO					
Instr. i+2			FI	DI	CO	FO	EI	WO				
Instr. i+3				FI	DI	CO	FO	EI	WO			
Instr. i+4					FI	DI	CO	FO	EI	WO		
Instr. i+5						FI	DI	CO	FO	EI	WO	
Instr. i+6							FI	DI	CO	FO	EI	WO

Execution time for the 7 instructions, with pipelining:

$(T_{ex}/6)*12= 2*T_{ex}$

- After a certain time (N-1 cycles) all the N stages of the pipeline are working: the pipeline is filled. Now, *theoretically*, the pipeline works providing maximal parallelism (N instructions are active simultaneously).

Acceleration by Pipelining (cont'd)

- Apparently a greater number of stages always provides better performance. However:
  - a greater number of stages increases the overhead in moving information between stages and synchronization between stages.
  - with the number of stages the complexity of the CPU grows.
  - it is difficult to keep a large pipeline at maximum rate because of *pipeline hazards*.

80486 and Pentium:	five-stage pipeline for integer instr.
	eight-stage pipeline for FP instr.
PowerPC:	four-stage pipeline for integer instr.
	six-stage pipeline for FP instr.

Pipeline Hazards

- Pipeline hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle. The instruction is said to be *stalled*. When an instruction is stalled, all instructions later in the pipeline than the stalled instruction are also stalled. Instructions earlier than the stalled one can continue. No new instructions are fetched during the stall.

- Types of hazards:
  1. Structural hazards
  2. Data hazards
  3. Control hazards

Structural Hazards

- Structural hazards occur when a certain resource (memory, functional unit) is requested by more than one instruction at the same time.

Instruction ADD R4,X fetches in the FO stage operand X from memory. The memory doesn't accept another access during that cycle.

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
ADD R4,X	FI	DI	CO	FO	EI	WO						
Instr. i+1		FI	DI	CO	FO	EI	WO					
Instr. i+2			FI	DI	CO	FO	EI	WO				
Instr. i+3				stall	FI	DI	CO	FO	EI	WO		
Instr. i+4						FI	DI	CO	FO	EI	WO	

Penalty: 1 cycle

- Certain resources are duplicated in order to avoid structural hazards. Functional units (ALU, FP unit) can be pipelined themselves in order to support several instructions at a time. A classical way to avoid hazards at memory access is by providing separate data and instruction caches.

Data Hazards

- We have two instructions, I1 and I2. In a pipeline the execution of I2 can start before I1 has terminated. If in a certain stage of the pipeline, I2 needs the result produced by I1, but this result has not yet been generated, we have a data hazard.

I1:	MUL	R2,R3	$R2 \leftarrow R2 * R3$
I2:	ADD	R1,R2	$R1 \leftarrow R1 + R2$

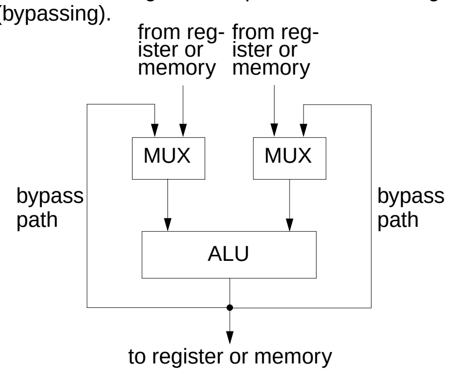
Clock cycle $\rightarrow$	1	2	3	4	5	6	7	8	9	10	11	12
MUL R2,R3	FI	DI	CO	FO	EI	WO						
ADD R1,R2		FI	DI	CO	stall	stall	FO	EI	WO			
Instr. i+2			FI	DI			CO	FO	EI	WO		

Before executing its FO stage, the ADD instruction is stalled until the MUL instruction has written the result into R2.

Penalty: 2 cycles

Data Hazards (cont'd)

- Some of the penalty produced by data hazards can be avoided using a technique called *forwarding* (bypassing).



- The ALU result is always fed back to the ALU input. If the hardware detects that the value needed for the current operation is the one produced by the previous operation (but which has not yet been written back) it selects the forwarded result as the ALU input, instead of the value read from register or memory.

Clock cycle $\rightarrow$	1	2	3	4	5	6	7	8	9	10	11	12
MUL R2,R3	FI	DI	CO	FO	EI	WO						
ADD R1,R2		FI	DI	CO	stall	FO	EI	WO				

After the EI stage of the MUL instruction the result is available by forwarding. The penalty is reduced to one cycle.

Control Hazards

- Control hazards are produced by branch instructions.

Unconditional branch

```
            - - - - - - - - - - - - - -
            BR      TARGET
            - - - - - - - - - - - - - -
TARGET      - - - - - - - - - - - - - -
```

After the FO stage of the branch instruction the address of the target is known and it can be fetched

Clock cycle $\rightarrow$	1	2	3	4	5	6	7	8	9	10	11	12
BR TARGET	FI	DI	CO	FO	EI	WO						
target		$\cancel{\text{FI}}$	stall	stall	FI	DI	CO	FO	EI	WO		
target+1						FI	DI	CO	FO	EI	WO	

The instruction following the branch is fetched; before the DI is finished it is not known that a branch is executed. Later the fetched instruction is discarded

Penalty: 3 cycles

Control Hazards (cont'd)

Conditional branch

```
            ADD      R1,R2        R1 ← R1 + R2
            BEZ      TARGET       branch if zero
            instruction i+1
            - - - - - - - - - - - -
TARGET      - - - - - - - - - - - -
```

Branch is taken

At this moment, both the condition (set by ADD) and the target address are known.

Clock cycle $\rightarrow$	1	2	3	4	5	6	7	8	9	10	11	12
ADD R1,R2	FI	DI	CO	FO	EI	WO						
BEZ TARGET		FI	DI	CO	FO	EI	WO					
target			$\cancel{\text{FI}}$	stall	stall	FI	DI	CO	FO	EI	WO	

Penalty: 3 cycles

Branch **not** taken

At this moment the condition is known and instr+1 can go on.

Clock cycle $\rightarrow$	1	2	3	4	5	6	7	8	9	10	11	12
ADD R1,R2	FI	DI	CO	FO	EI	WO						
BEZ TARGET		FI	DI	CO	FO	EI	WO					
instr i+1			FI	stall	stall	DI	CO	FO	EI	WO		

Penalty: 2 cycles

Control Hazards (cont'd)

- With conditional branch we have a penalty even if the branch has *not* been taken. This is because we have to wait until the branch condition is available.

- Branch instructions represent a major problem in assuring an optimal flow through the pipeline. Several approaches have been taken for reducing branch penalties (*see slides of the following lecture*).

Summary

- Instructions are executed by the CPU as a sequence of steps. Instruction execution can be substantially accelerated by *instruction pipelining*.

- A pipeline is organized as a succession of N stages. At a certain moment N instructions can be active inside the pipeline.

- Keeping a pipeline at its maximal rate is prevented by *pipeline hazards*. *Structural hazards* are due to resource conflicts. *Data hazards* are produced by data dependencies between instructions. *Control hazards* are produced as consequence of branch instructions