

Pipelining

M. Sonza Reorda

Politecnico di Torino
Dipartimento di Automatica e Informatica

1

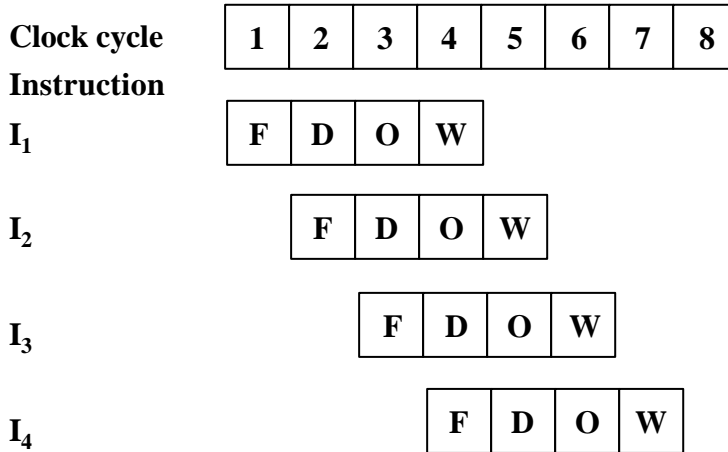
Introduction

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.

2

Example



3

Throughput

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*.

The length of the machine cycle is determined by the slowest stage.

4

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 stages.

5

Example processor: Unpipelined Implementation

The execution of each instruction is 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).

6

Instruction Fetch cycle

$IR \leftarrow Mem[PC]$

$NPC \leftarrow PC + 4$

7

Instruction Decode/ Register Fetch

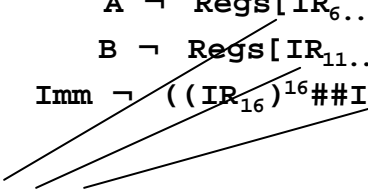
$A \leftarrow Regs[IR_{6..10}]$

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

$Imm \leftarrow ((IR_{16})^{16} \# \# IR_{16..31})$

8

Instruction Decode/ Register Fetch

$$\begin{aligned} A &\leftarrow \text{Regs}[\text{IR}_{6..10}] \\ B &\leftarrow \text{Regs}[\text{IR}_{11..15}] \\ \text{Imm} &\leftarrow ((\text{IR}_{16})^{16} \# \text{IR}_{16..31}) \end{aligned}$$


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

9

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

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

$$\text{Cond} \leftarrow (A \text{ op } 0);$$

10

Memory Access/Branch Completion Cycle

- Memory reference

$LMD \leftarrow Mem[ALUOutput]$ or
 $Mem[ALUOutput] \leftarrow B;$

- Branch

$if (cond) PC \leftarrow ALUOutput \text{ else } PC \leftarrow NPC;$

11

Write-back Cycle

- Register-Register ALU instruction

$Regs[IR_{16..20}] \leftarrow ALUOutput;$

- Register-Immediate ALU instruction

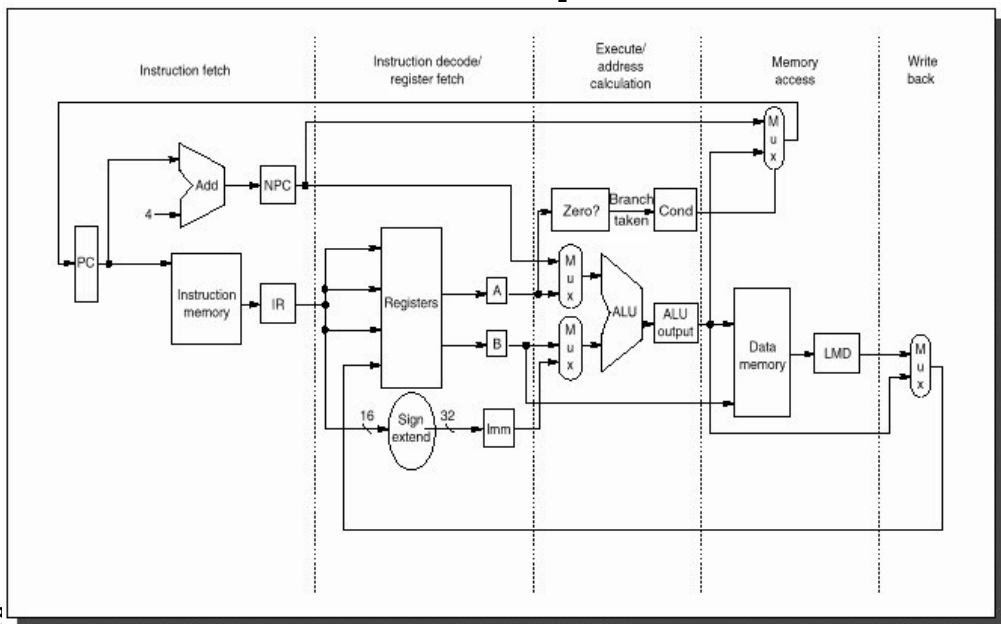
$Regs[IR_{11..15}] \leftarrow ALUOutput;$

- Load instruction

$Regs[IR_{11..15}] \leftarrow LMD;$

12

The Datapath



13

Behavior and optimizations

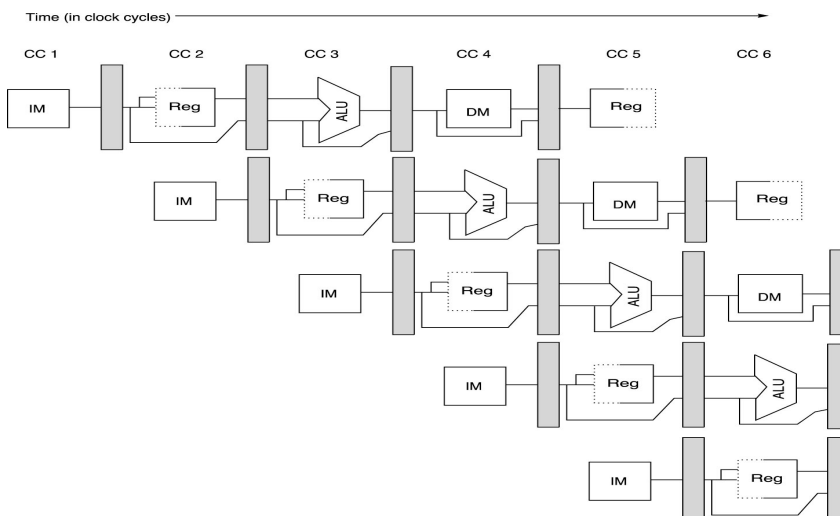
- All instructions require 5 clock cycles, unless branch instructions, which require 4 clock cycles
- Optimizations could be done for reducing the average CPI: as an example, the ALU instructions could be completed during the MEM cycle
- Hardware resources could be optimized by avoiding duplications (e.g., for ALUs, and memory)
- An alternative single-clock architecture 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.
- At every clock cycles, 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 in ID and for writing 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.

15

Evolution in time



16

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).

17

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

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

18

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

$$\text{speedup} = 4.4 \text{ ns} / 1.2 \text{ ns} = 3.7 \text{ times}$$

19

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.

20

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 started after the stalled instruction are also stalled
- the instructions started before the stalled instruction continue.

A stall causes the introduction of a *bubble* in the pipeline.

21

Example

Suppose that only one access to memory can happen during each clock cycle: therefore, fetch of this instruction can not be performed here.

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

22

Pipeline Performance with Stalls

$$\text{speedup from pipelining} = \frac{\text{average instruction time unpipeline d}}{\text{average instruction time pipelined}}$$

$$= \frac{\text{CPI}_{\text{unpipelined}} \times \text{clock cycle unpipeline d}}{\text{CPI}_{\text{pipelined}} \times \text{clock cycle pipelined}}$$

$$\text{CPI}_{\text{pipelined}} = \text{ideal_CPI} + \text{pipeline_stall_clock_cycles_per_instruction}$$

$$= 1 + \text{pipeline_stall_clock_cycles_per_instruction}$$

23

Simplified version

If

- the clock cycle in the two versions is comparable
- all the instructions take the same number of clock cycles as the depth of the pipeline:

$$\text{speedup} = \frac{\text{CPI}_{\text{unpipelined}}}{1 + \text{Pipeline_stall_cycles_per_instruction}}$$

$$= \frac{\text{Pipeline depth}}{1 + \text{Pipeline_stall_cycles_per_instruction}}$$

24

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.

25

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

26

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.

27

Example

Load structural hazards happens when two instructions both 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?

28

Solution

For the machine without structural hazard:

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

For the machine with structural hazard:

$$\text{Average Instruction Time} = \text{CPI} \times \text{Clock Cycle Time}$$

$$= (1 + 0.4 \times 1) \times \frac{\text{Clock Cycle Time}_{\text{ideal}}}{1.05}$$

$$= 1.3 \times \text{Clock Cycle Time}_{\text{ideal}}$$

29

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
- nondeterministic behavior.

30

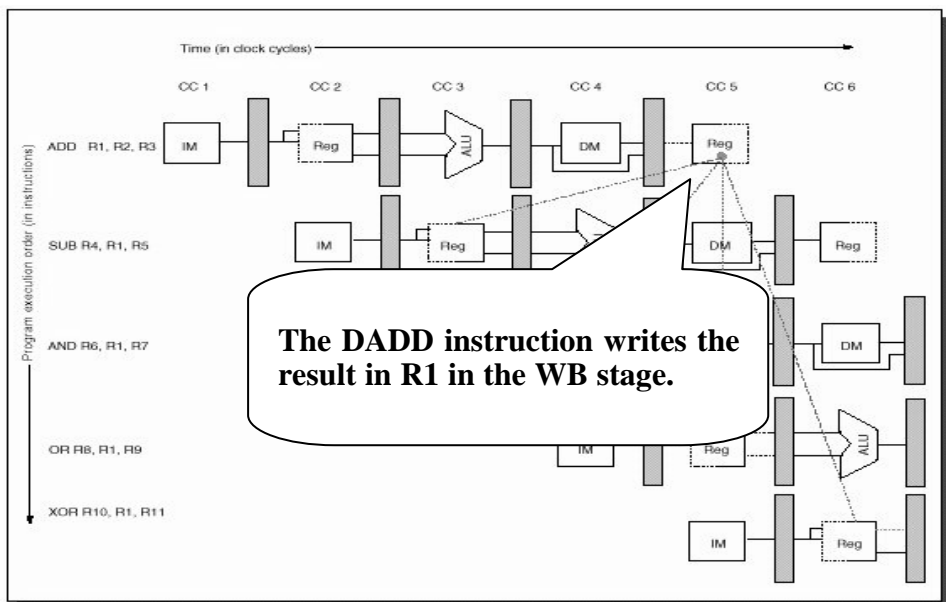
Example

Let consider the following code fragment:

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

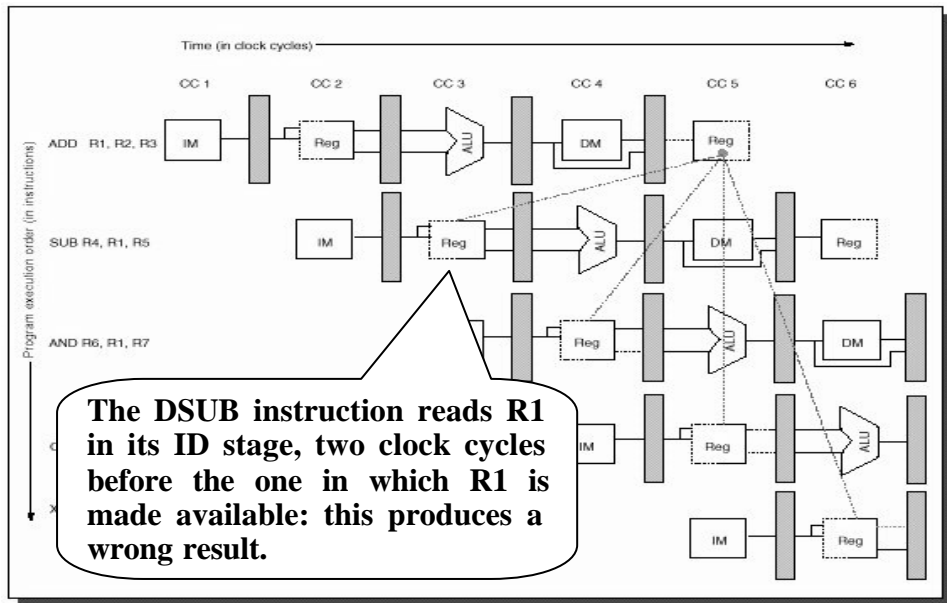
31

Example (cont'd)



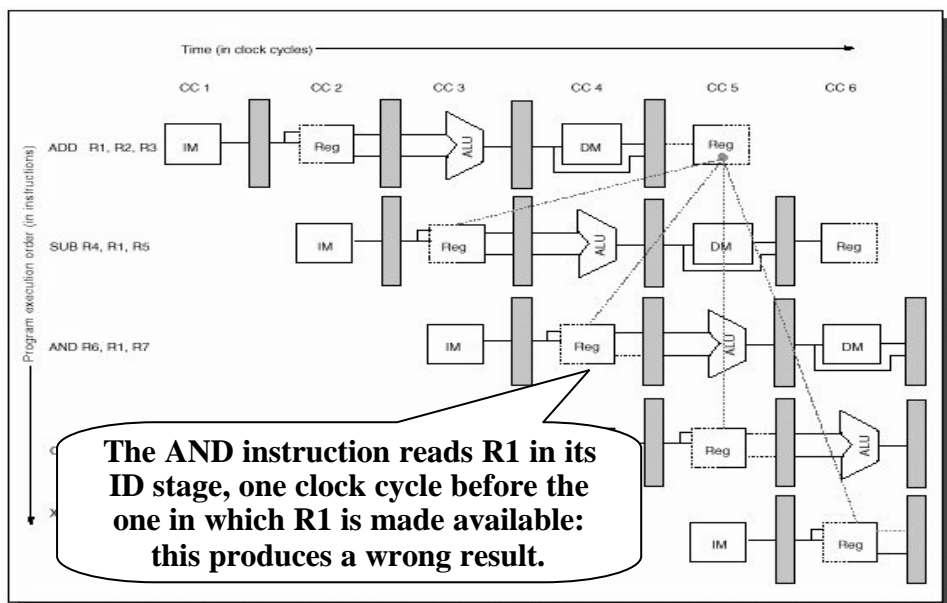
32

Example (cont'd)



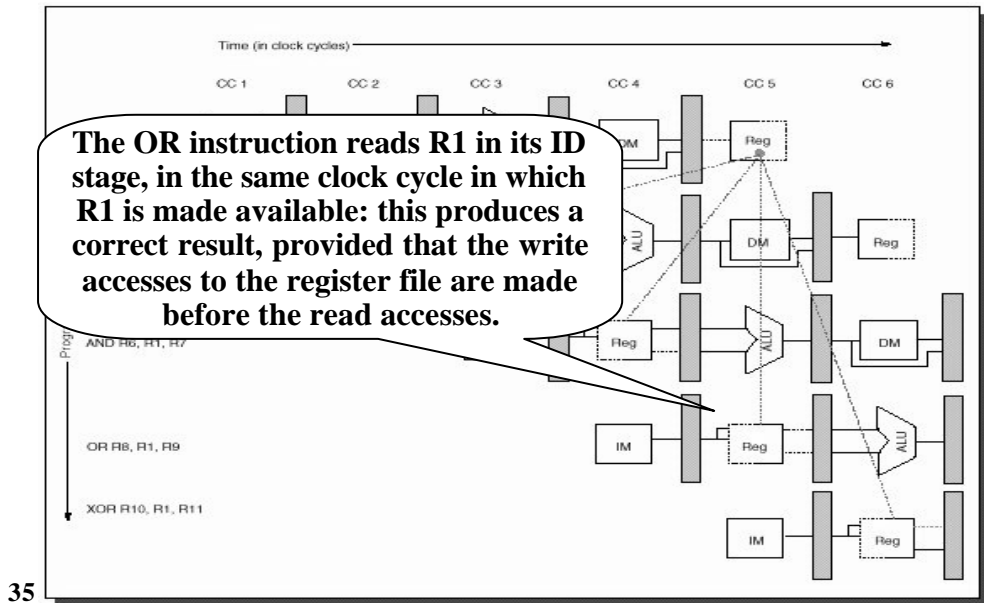
33

Example (cont'd)



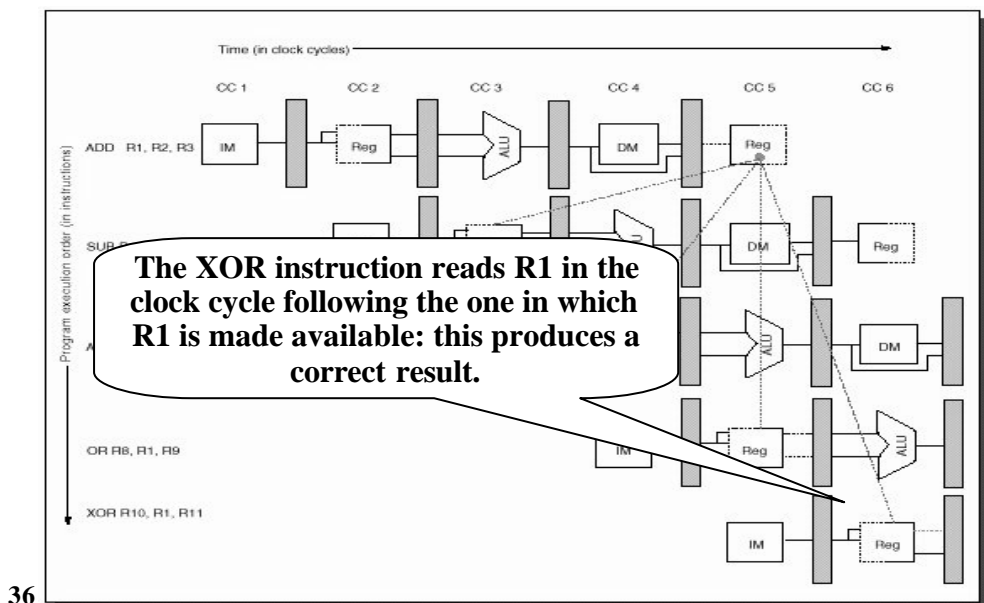
34

Example (cont'd)



35

Example (cont'd)



36

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*.

37

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.

38

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.

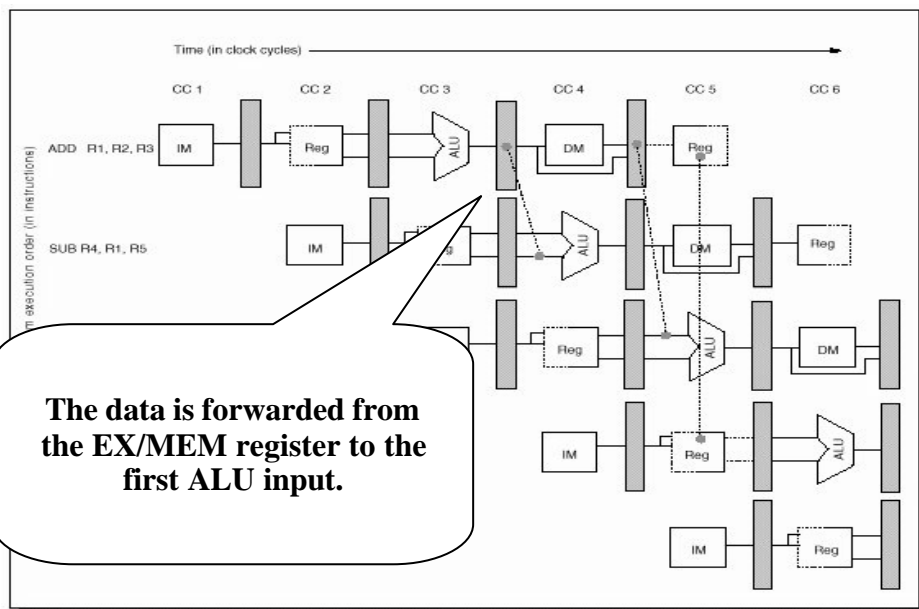
In this case, the hardware selects the ALU result as the ALU input rather than the value from the register file.

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.

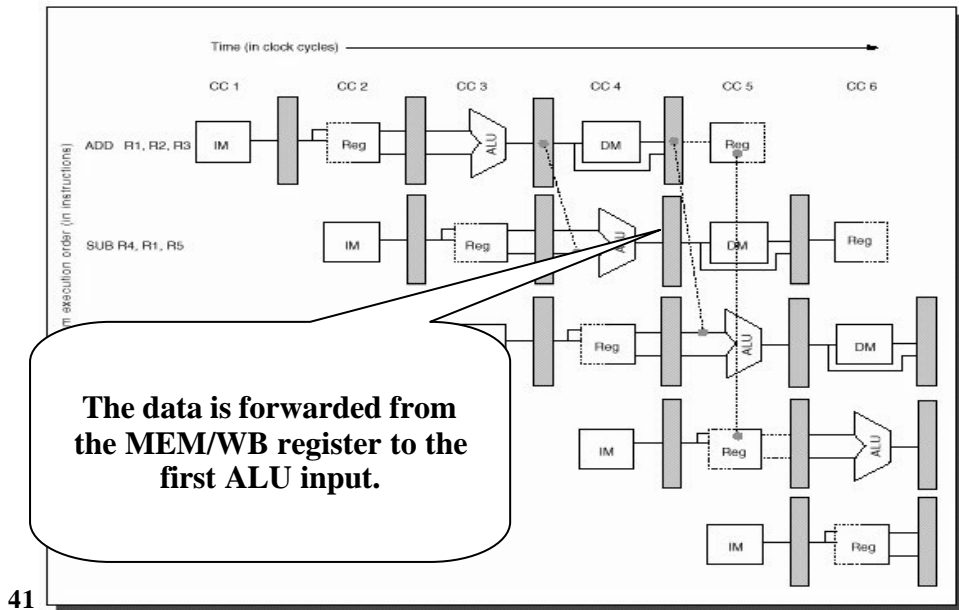
39

Example



40

Example



Generalizing the forward technique

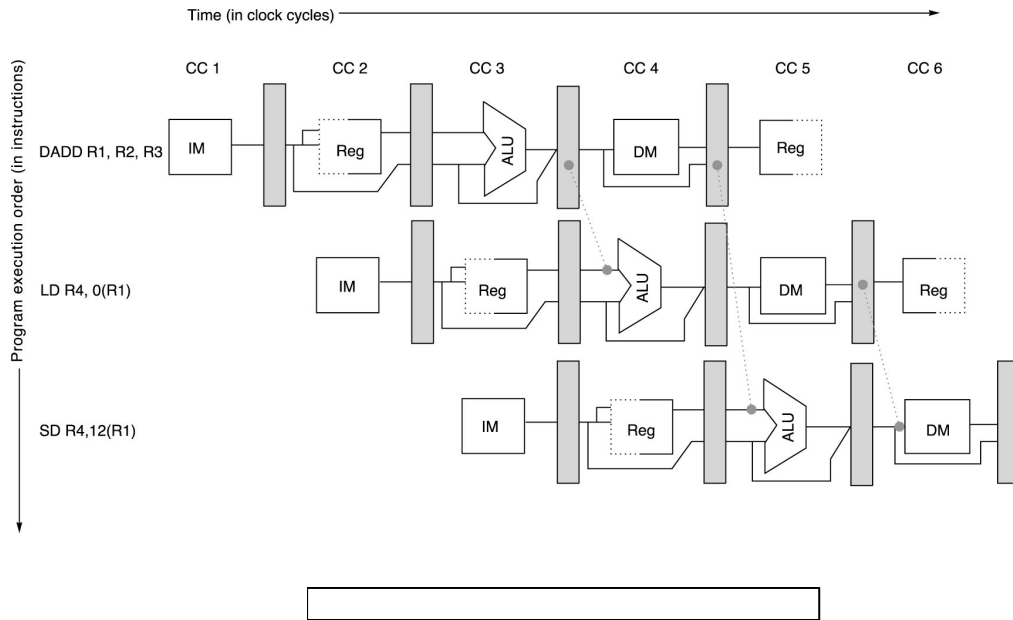
In order to always avoid stalling, forwarding should be made possible between any pipeline register to any input of any functional unit.

Example

```
DADD    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

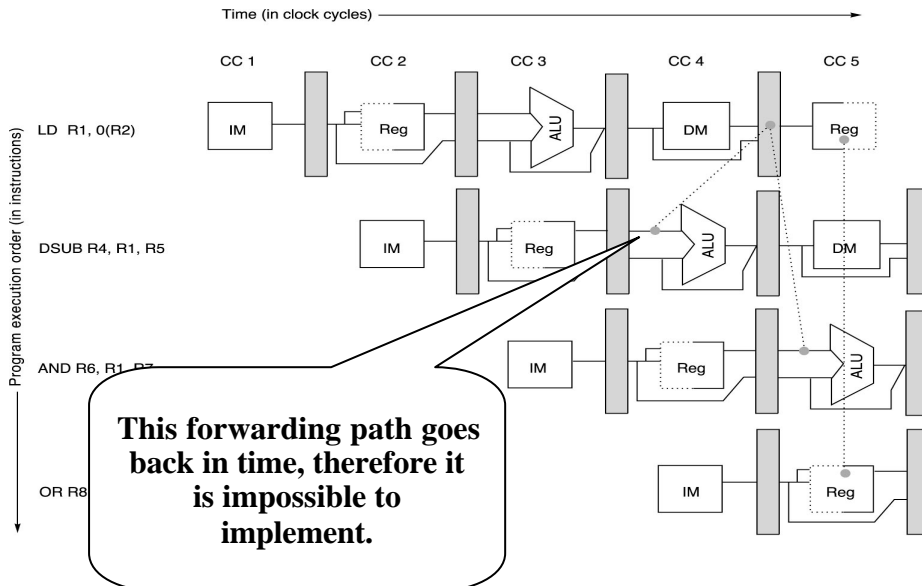
Not all potential data hazards can be solved through data forwarding.

Example

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

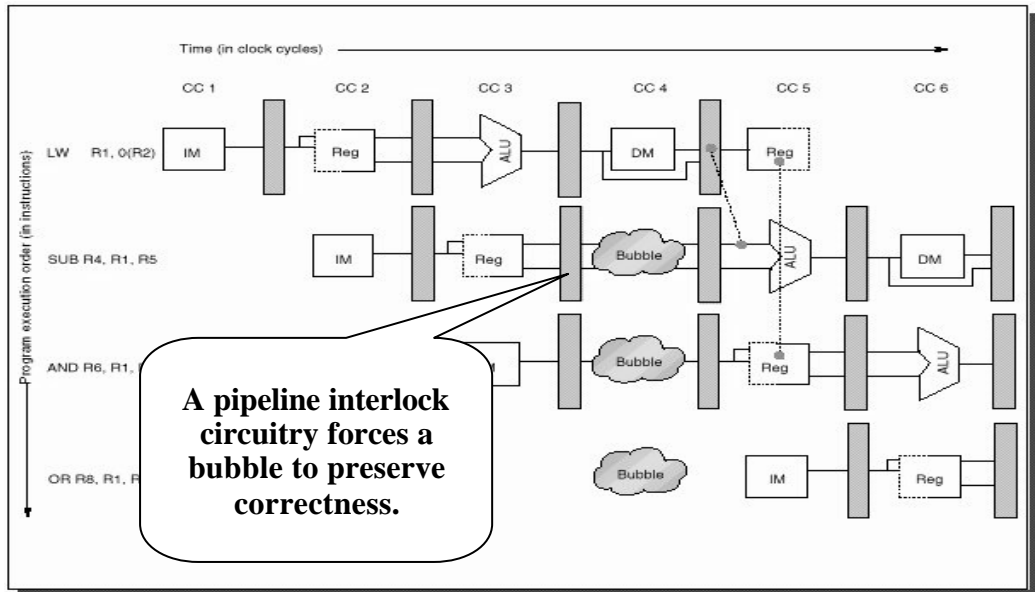
45

Forwarding-based solution



46

Solution with stall



CONTROL HAZARDS

They are mainly due to branches (conditional jumps), which may change the PC (*taken* branch) after the following instruction has been fetched already.

They can cause greater performance losses than data hazards.

Basic solution

A possible solution is based on stalling the pipeline as soon as a branch instruction is detected (ID stage).

Let us suppose that the PC is modified by the branch instruction at the end of the ID stage.

49

Example

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor+1				IF	ID	EX	MEM
Branch successor+2					IF	ID	EX

50

Example

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor+1				IF	ID	EX	MEM
Branch successor+2					IF	ID	EX

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

51

Example

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor+1				IF	ID	EX	MEM
Branch successor+2					IF	ID	EX

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

52

Example

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor+1				IF	ID	EX	MEM
Branch successor+2					IF	ID	EX



This operation is
ALWAYS useless.

53

Improved solutions

There are several techniques for reducing the performance degradation due to branches. The following are *compile-time* solutions:

- freezing the pipeline
- predict untaken
- predict taken
- delayed branch.

54

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.

55

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.

56

Branch 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

57

Branch taken

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

58

Branch taken

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
Branch target + 2					IF	ID	EX

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

59

Predict taken

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

60

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 taken scheme, the `do while` for the predict untaken.

61

Delayed branch

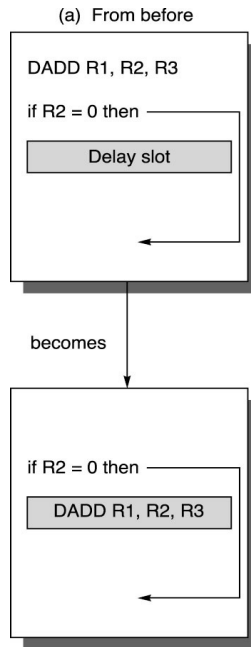
This technique is based on filling the slot after the branch instruction (named *branch-delay slot*) with instructions which have to be executed no matter the branch outcome.

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.

62

Example



63

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.

64

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.

65

Performance of Branch Schemes

$$\begin{aligned}\text{pipeline speedup} &= \frac{\text{pipeline depth}}{1 + \text{pipeline stall cycles from branches}} \\ &= \frac{\text{pipeline depth}}{1 + \text{branch frequency} \times \text{branch penalty}}\end{aligned}$$

66