

Computer Architecture (Spring 2020)

Pipelining

Dr. Duo Liu (刘铎)

Office: Main Building 0626

Email: liuduo@cqu.edu.cn

MIPS Pipeline: Events per Stage

Stage	ALU Instruction	Load/Store Instruction	Branch Instruction
IF	$IF/ID.IR \leftarrow Mem[PC];$ $IF/ID.NPC \leftarrow \text{if } ((EX/MEM.opc == \text{branch}) \ \& \ EX/MEM.cond) \ EX/MEM.AluOutput \text{ else } PC+4;$		
ID	$ID/EX.A \leftarrow Regs[IF/ID.IR[rs]]; \quad ID/EX.B \leftarrow Regs[IF/ID.IR[rt]];$ $ID/EX.NPC \leftarrow IF/ID.NPC; \quad ID/EX.IR \leftarrow IF/ID.IR;$ $ID/EX.Imm \leftarrow \text{sign-extend}(IF/ID.IR[\text{immediate field}])$		
EX	$EX/MEM.IR \leftarrow ID/EX.IR;$ $EX/MEM.ALUOutput \leftarrow ID/EX.A \ op \ ID/EX.B \ \text{or}$ $EX/MEM.ALUOutput \leftarrow ID/EX.A \ op \ ID/EX.Imm$	$EX/MEM.IR \leftarrow ID/EX.IR;$ $EX/MEM.ALUOutput \leftarrow ID/EX.A \ op \ ID/EX.Imm$	$EX/MEM.ALUOutput \leftarrow ID/EX.NPC + (ID/EX.Imm \ll 2)$ $EX/MEM.Cond \leftarrow ID/EX.A == 0)$
MEM	$MEM/WB.IR \leftarrow EX/MEM.IR$ $MEM/WB.ALUOutput \leftarrow EX/MEM.AluOutput$	$MEM/WB.IR \leftarrow EX/MEM.IR$ $MEM/WB.LMD \leftarrow Mem[EX/MEM.AluOutput]$ <i>or</i> $Mem[EX/MEM.AluOutput] \leftarrow EX/MEM.B$	
WB	$Regs[MEM/WB.IR[rd]] \leftarrow MEM/WB.AluOutput \ \text{or}$ $Regs[MEM/WB.IR[rt]] \leftarrow MEM/WB.AluOutput$	<i>for load only:</i> $Regs[MEM/WB.IR[rt]] \leftarrow MEM/WB.LMD$	

MIPS Features that Simplify Pipelining

- ISA encoding: all instructions have same length
 - balanced separation of fetch and decode stages
- ISA encoding: few instructions formats with the source register fields in the same position
 - symmetry allows reading of register file and decoding of instruction simultaneously during the second stage (fixed-field decoding)
- Memory accessed only by load/store instructions
 - memory address is “pre-computed” during the execution stage for the following stage (no operation involves operands in memory)
- Each operation writes at most on result (and near the end of the pipeline)
 - simplifies forwarding (useful for handling data hazards)
- All instructions change an entire register, memory access does not require realignment...
 - simpler instructions can be decomposed in steps that can be performed each in a single pipeline stage

Basic Pipeline

Instr #	Clock number								
	1	2	3	4	5	6	7	8	9
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

The First Pipelined Computers

- IBM 7030 “Stretch”
 - considered one of the first general-purpose pipelined computer (late 1950s)
 - pipeline overlapping fetch, decode, execute stages
- CDC 6600
 - Control Data Corp. (Seymour Cray)
 - the first supercomputer (1964)
 - interaction between ISA and pipeline HW was well understood
 - ISA was kept simple on purpose
- IBM 360
 - more complex pipeline with Tomasulo’s Algorithm (1964)



MIPS Pipelining: Basic Performance Issues

- **Pipelining**

- improve **CPU Throughput** (reciprocal of CPU Time)
- slightly increases execution time
- result: program run faster!

$$\text{CPU Time} = \text{IC} \times \text{CPI} \times \text{CCT}$$

$$[\text{time per instruction}]_{\text{pipelined}} = \frac{[\text{time per instruction}]_{\text{nonpipelined}}}{\text{number of pipeline stages}}$$

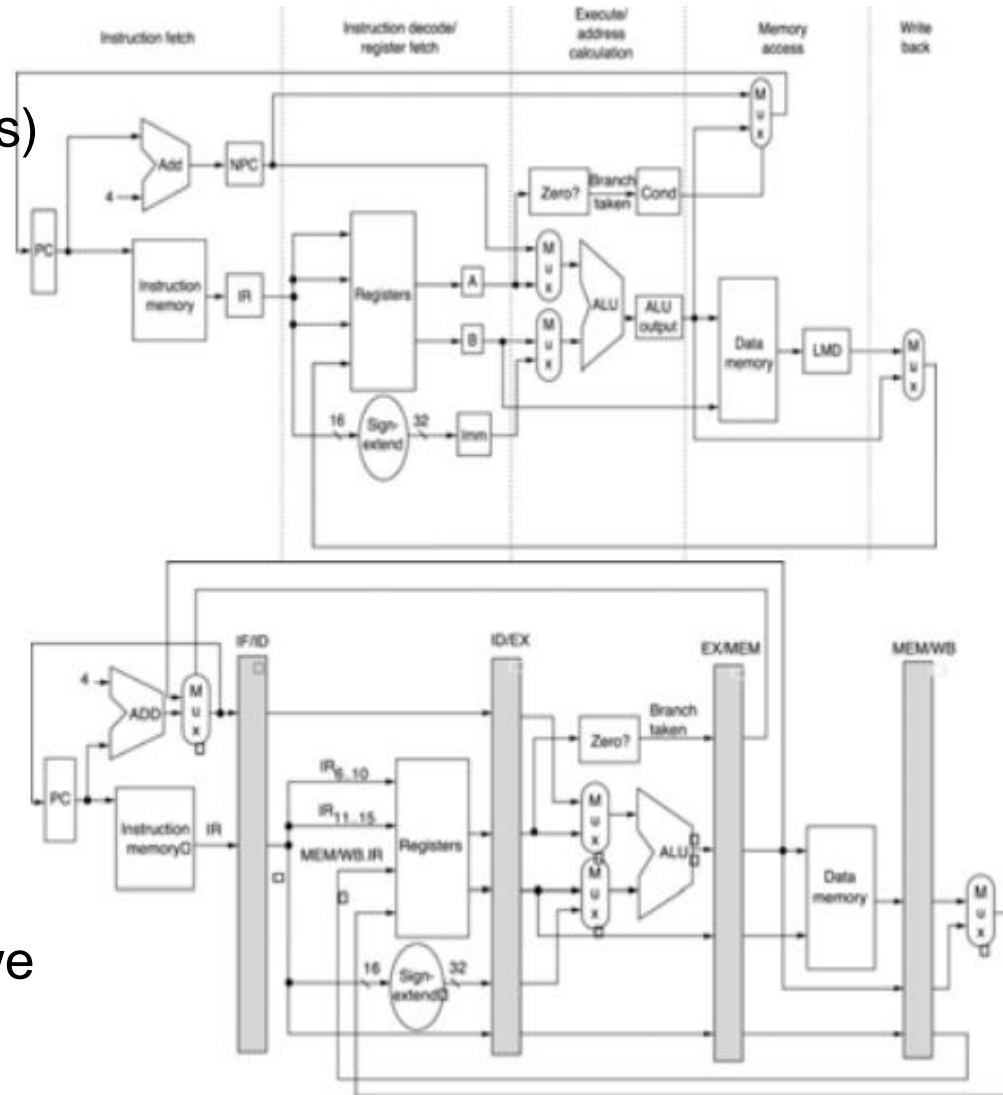
- **Pipelining can be seen as either**

- decreasing the **CPI** of a multi-cycle un-pipelined implementation
- decreasing the **CCT** of a single-cycle un-pipelined implementation

	IC	CPI	CCT
Program	✓		
Compiler	✓		
ISA	✓	✓	
HW organization		✓	✓
HW technology			✓

Example: Pipelining Speedup vs. Multi-cycle Non-Pipelined Implementation

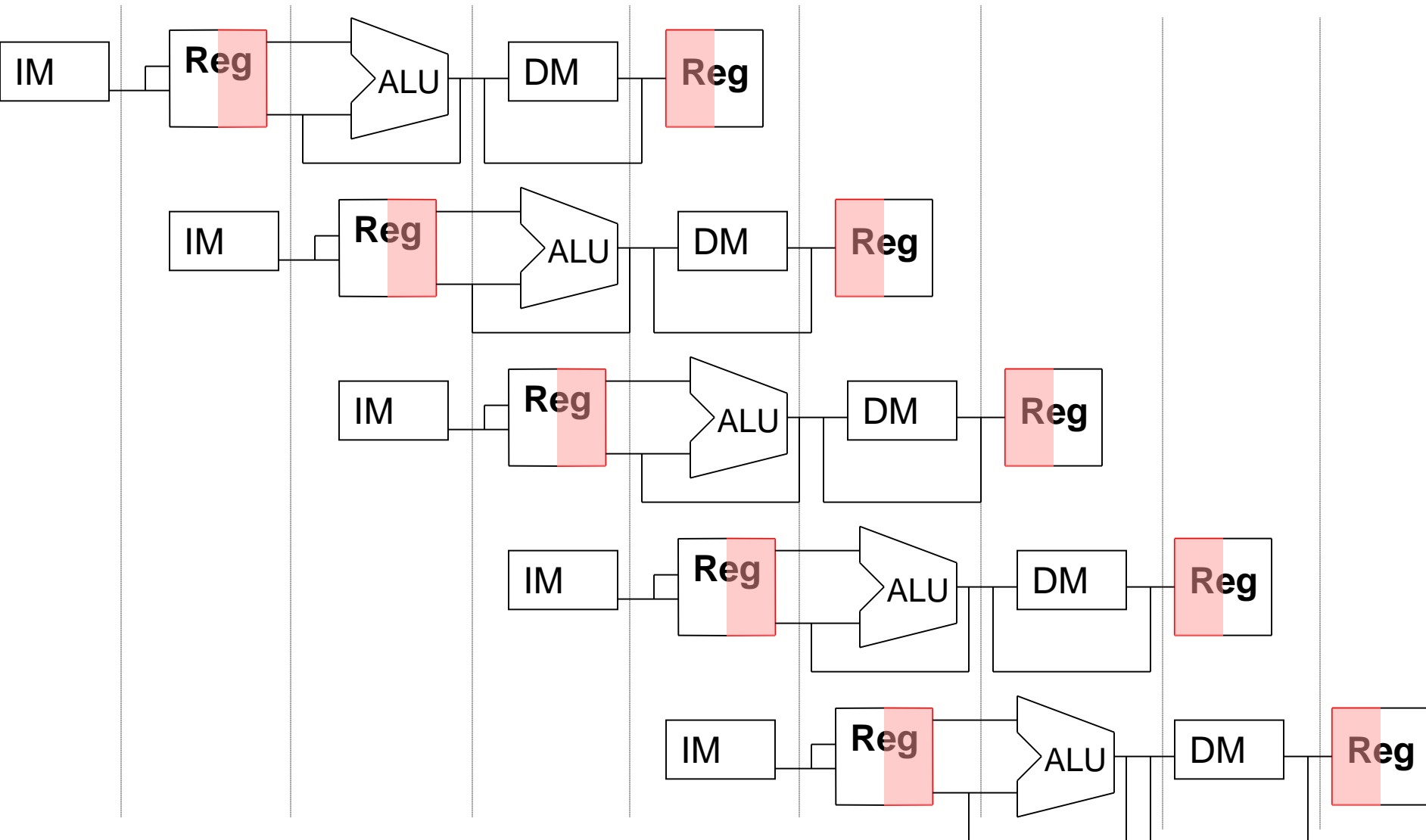
- **Instruction CPI**
 - 4 cycles (branches and stores)
 - 5 cycles (other instructions)
- **Average CPI = 4.75 assuming**
 - 15% branches
 - 10% stores
- **Average Instruction ExecTime is 4.75ns, assuming CCT = 1ns**
- **Ideal CPI = 1 (almost always)**
- **Average Instruction ExecTime is 1.2ns with CCT = 1ns and clock overhead = 0.2 ns**
- **Speedup is 3.96x for ideal pipeline**
 - as we'll see soon, in reality we need to sum the pipeline stalled clock cycles per



Performance limitations

- Imbalance among pipe stages
 - limits cycle time to slowest stage
- Pipelining overhead
 - Pipeline register delay
 - Clock skew
- $\text{Clock cycle} > \text{clock skew} + \text{latch overhead}$
- Hazards

Pipeline Resources



Pipeline Hazards and Their Classification

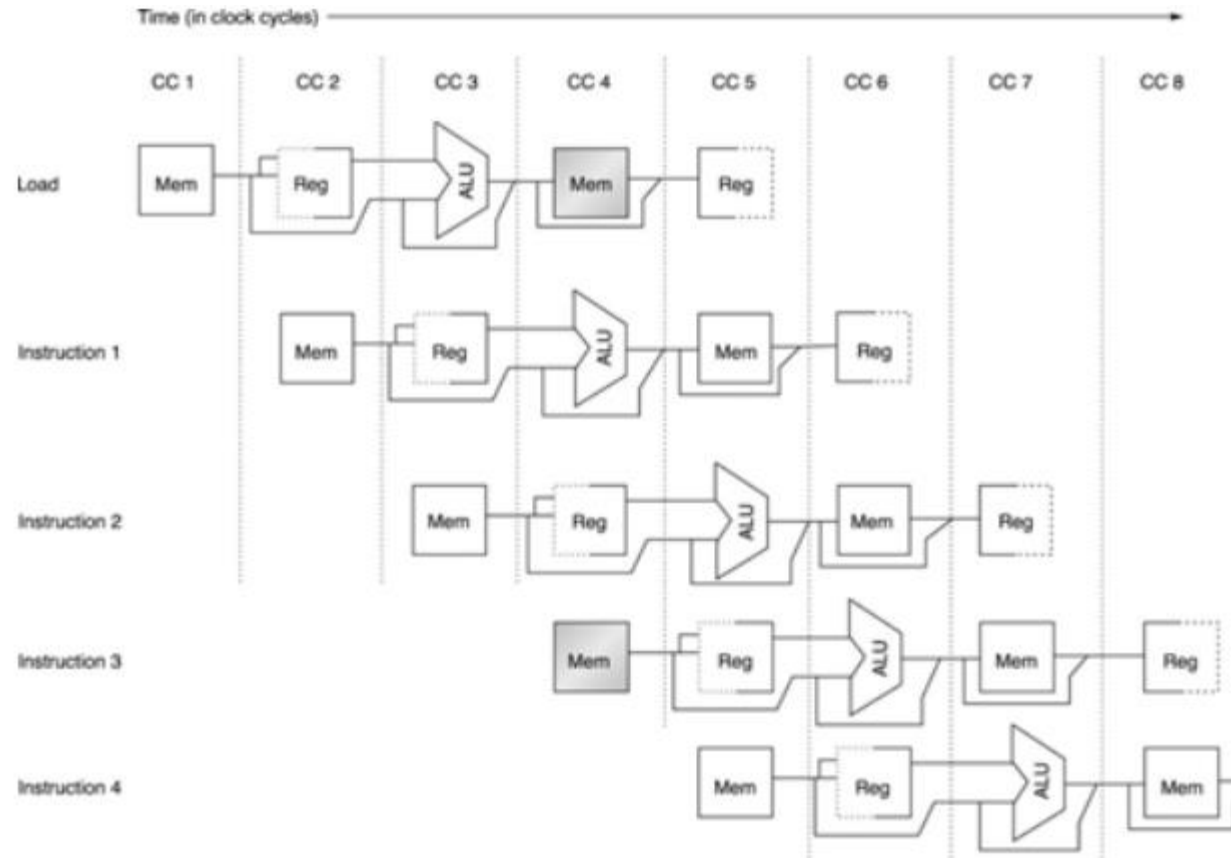
- Pipeline hazard situation
 - “the next instruction cannot execute in the following clock cycle”
- Pipeline hazards come in three different flavors
 - Structural Hazards
 - arise from resource conflict: the HW cannot support all possible instruction combinations simultaneously in overlapped execution
 - Data Hazards
 - arise when an instruction depends on the result of a previous instruction in a way exposed by the pipeline overlapped execution
 - Control Hazards
 - arise from the pipelining of branches, jumps...
- Hazards may force pipeline stalling
 - Instructions issued after the stalled one must stall also (and fetching is stalled) while all those issued earlier must proceed

Structural Hazards

- Overlapped execution of instructions:
 - Pipelining of functional units
 - Duplication of resources
- Structural Hazard
 - When the pipeline can not accommodate some combination of instructions
- Consequences
 - Stall
 - Increase of CPI from its ideal value (1)

Structural Hazards: Examples

- Structural hazards are due to resource constraints:
 - some resources are not duplicated enough to allow all combination of instructions in the pipeline to execute
 - e.g., the architecture is not Harvard-like
 - a functional unit is not fully pipelined
 - e.g., an instruction takes more than one clock cycle to go through it

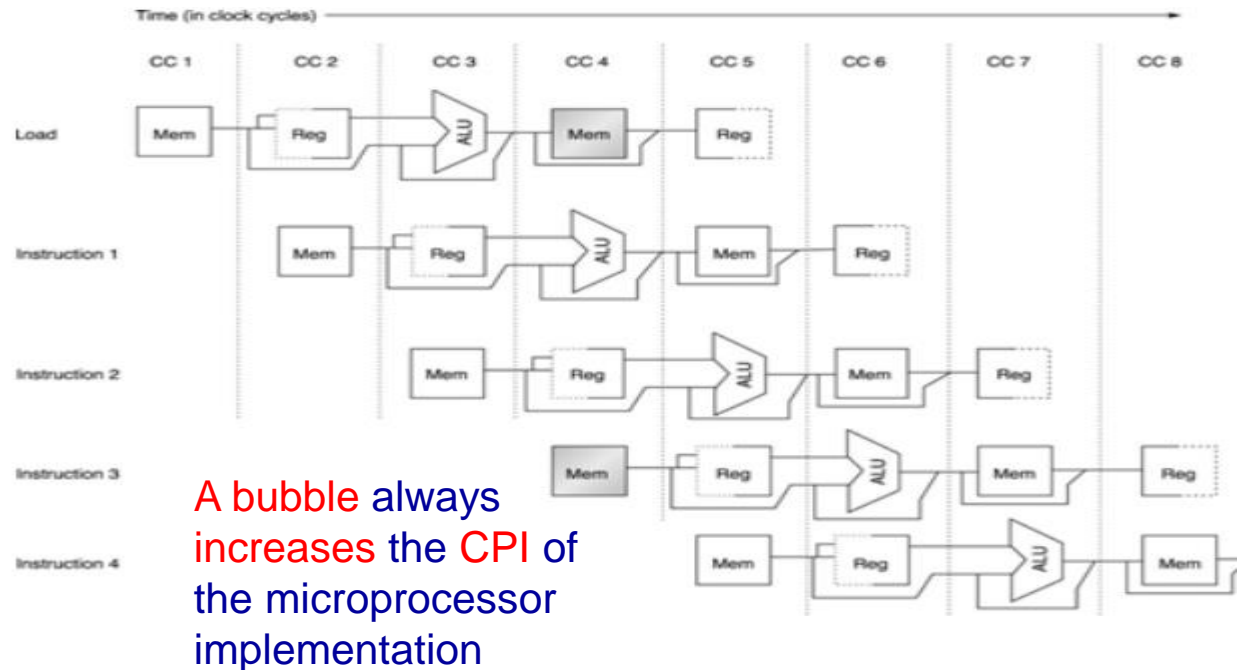


The ultimate reason for the presence of structural hazards is the designers' attempt to reduce implementation costs

Pipeline Stalls (or Bubbles): Example of Loading/ Fetching from a Unified Single-Port Memory

- Structural hazard causes a bubble

- it floats through the pipeline taking space but carrying no useful information



instruction	1	2	3	4	5	6	7	8	9
LW R10, 20(R1)	IF	ID	EX	MEM	WB				
SUB R11, R2, R3		IF	ID	EX	MEM	WB			
ADD R12, R3, R4			IF	ID	EX	MEM	WB		
STALL									
ADD R14, R5, R6					IF	ID	EX	MEM	WB

Impact of Stalls on the Performance of Pipelined Implementation

$$\text{speedupFromPipelining} = \frac{[CPI]_{\text{nonpipelined}} \times [CCT]_{\text{nonpipelined}}}{[CPI]_{\text{pipelined}} \times [CCT]_{\text{pipelined}}}$$

$$\begin{aligned}[CPI]_{\text{pipelined}} &= \text{idealCPI} + \text{pipelineStallCyclesPerInstruction} \\ &= 1 + \text{pipelineStallCyclesPerInstruction}\end{aligned}$$

- Special case: ignoring the pipeline clock-period overhead and assume that
 - the pipeline stages are perfectly balanced
 - all instructions take the same number of cycles in the non-pipelined implementation (equal to the pipeline depth)

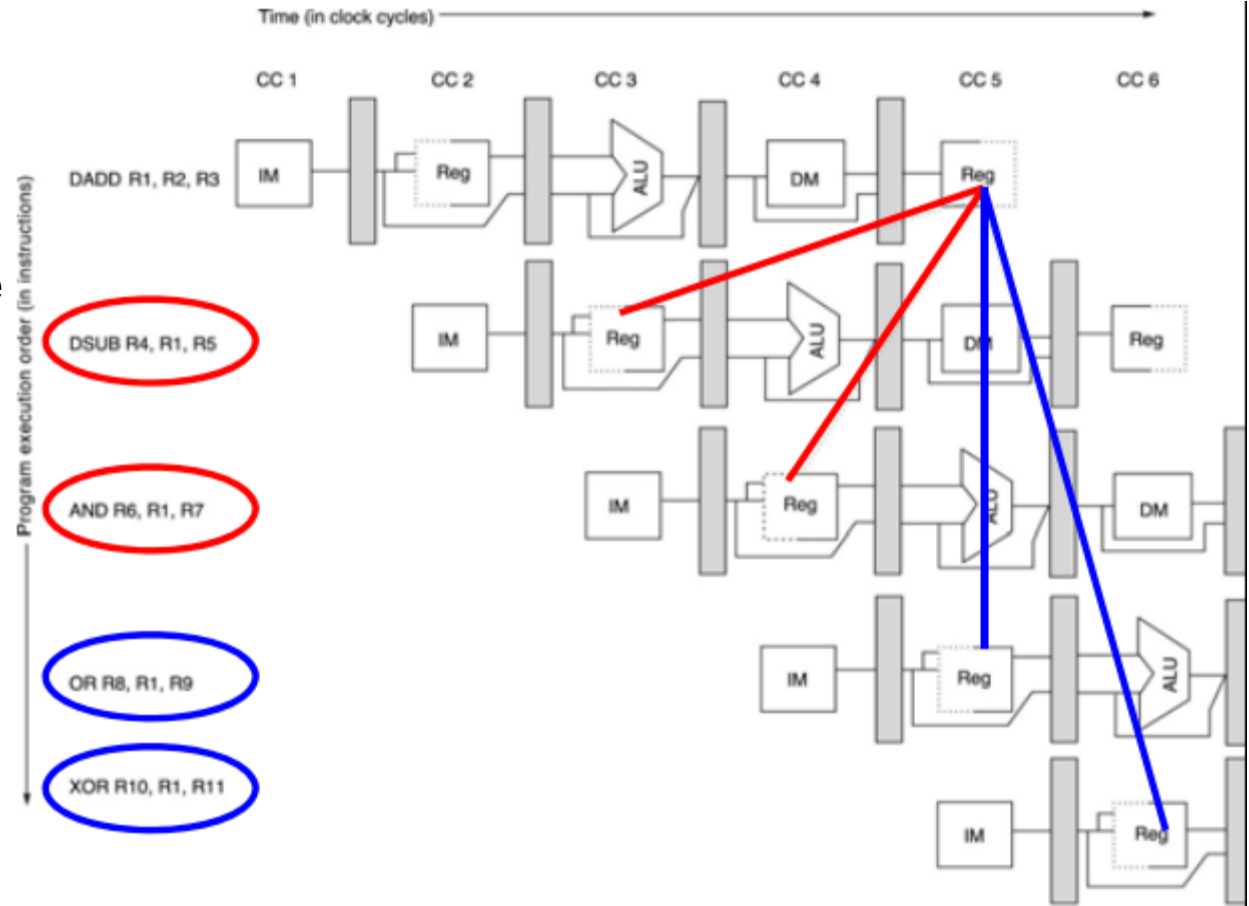
$$\text{speedupFromPipelining} = \frac{\text{pipelineDepth}}{1 + \text{pipelineStallCyclesPerInstruction}}$$

Impact of Stalls on the Performance of Pipelined Implementation - Example

- Implementation without structural hazards
 - ideal CPI = 1
 - CCT = 1ns
- Implementation with the “load/store” structural hazard
 - CCT = 900ps
- Suppose that
 - 40% of the executed instructions are loads or stores
- Which implementation is faster?
 - $(\text{averageInstructionTime})_{\text{noHaz}} = 1 * 1 = 1$
 - $(\text{averageInstructionTime})_{\text{haz}} = (1 + 0.4 * 1) * 0.9 = 1.26$
 - implementation without structural hazard is 1.26 times faster than the other

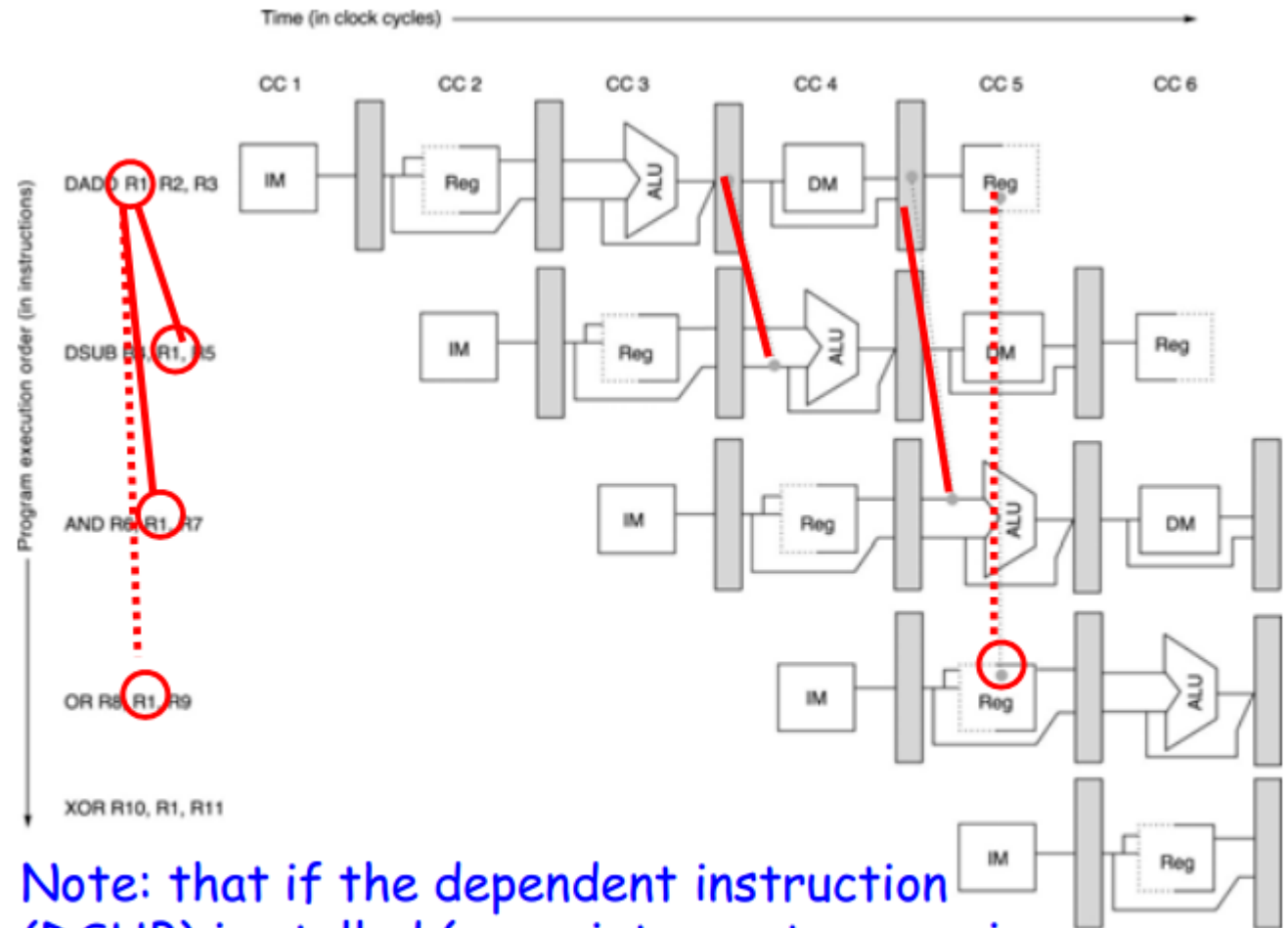
Data Hazards

- Which value do these instructions read from R1 ?
 - Unless we take care of the data hazard the value read is not even deterministic!
- This data hazard is called RAW (read after write)
 - the name refers to the expected execution flow



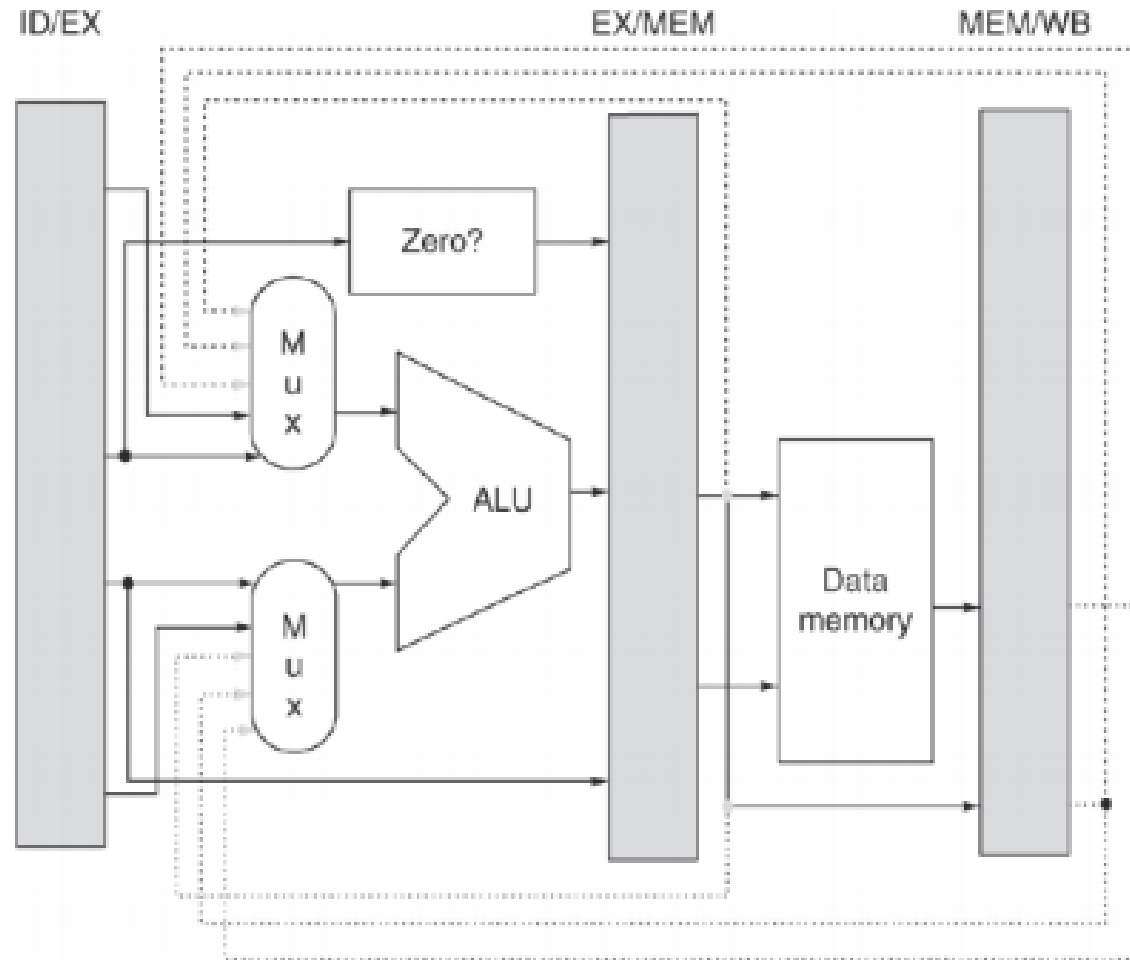
Minimizing Data Hazards: Forwarding (or By-Passing)

- An efficient technique based on a simple idea
 - subsequent instructions only need the value in R1 after this has been computed!
- ALU result is always fed back to the ALU input from both EX/MEM and MEM/WB



Forwarding to the ALU

- 3 inputs added to the MUX
- 3 bypass paths
 1. From ALU output at the end of EX
 2. From ALU output at the end of MEM stage
 3. The memory output at the end of the MEM stage



Data Dependences

- Types of data dependences
 - **Flow dependence** (true data dependence – read after write)
 - **Output dependence** (write after write)
 - **Anti dependence** (write after read)
- Which ones cause stalls in a pipelined machine?
 - For all of them, we need to ensure semantics of the program are correct
 - Flow dependences always need to be obeyed because they constitute true dependence on a value
 - Anti and output dependences exist due to limited number of architectural registers
 - They are dependence on a name, not a value
 - We will later see what we can do about them


How to Handle Data Dependences

- Anti and output dependences are easier to handle
 - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
 - Detect and wait until value is available in register file
 - Detect and forward/bypass data to dependent instruction
 - Detect and eliminate the dependence at the software level
 - No need for the hardware to detect dependence
 - Predict the needed value(s), execute “speculatively”, and verify
 - Do something else (fine-grained multithreading)
 - No need to detect

Data Dependence Types

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write
(RAW)

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read
(WAR)

Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$
 $r_3 \leftarrow r_6 \text{ op } r_7$



Write-after-Write
(WAW)