Computer Architecture (Fall 2022)

Pipelining

Dr. Duo Liu (刘铎)

Office: Main Building 0626

Email: liuduo@cqu.edu.cn

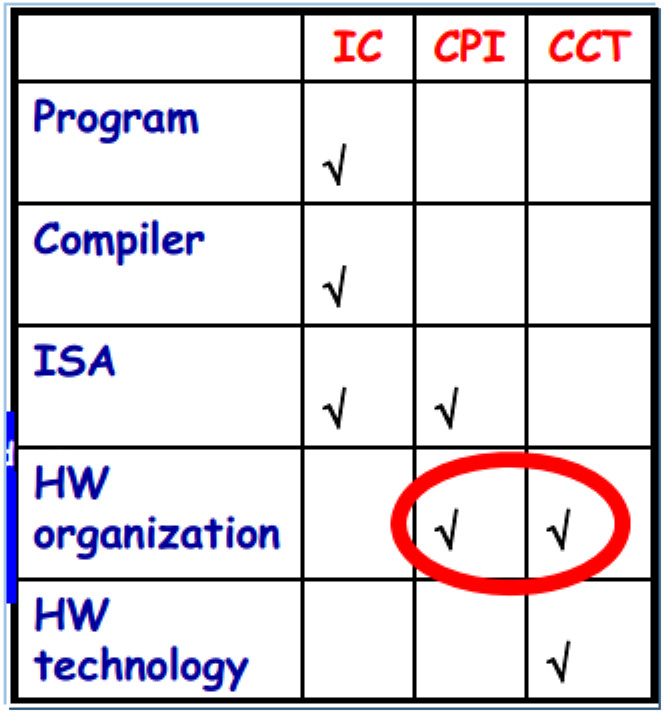
MIPS Pipelining: Basic Performance Issues

CPU Time = IC × CPI × CCT

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

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

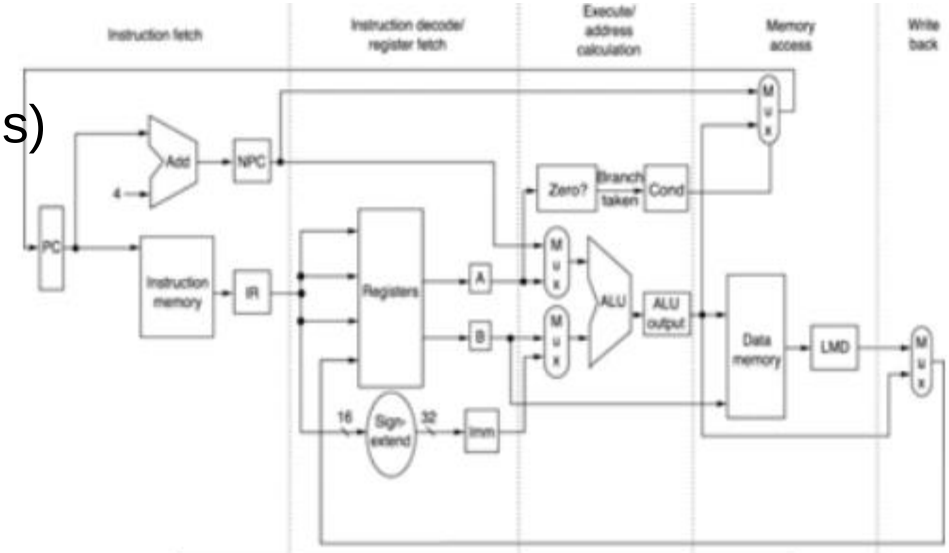
- Pipelining can be seen as either
  - decreasing the CPI of a multi-cycle un-pipelined implementation
  - decreasing the CCT of a single-cycle un-pipelined implementation

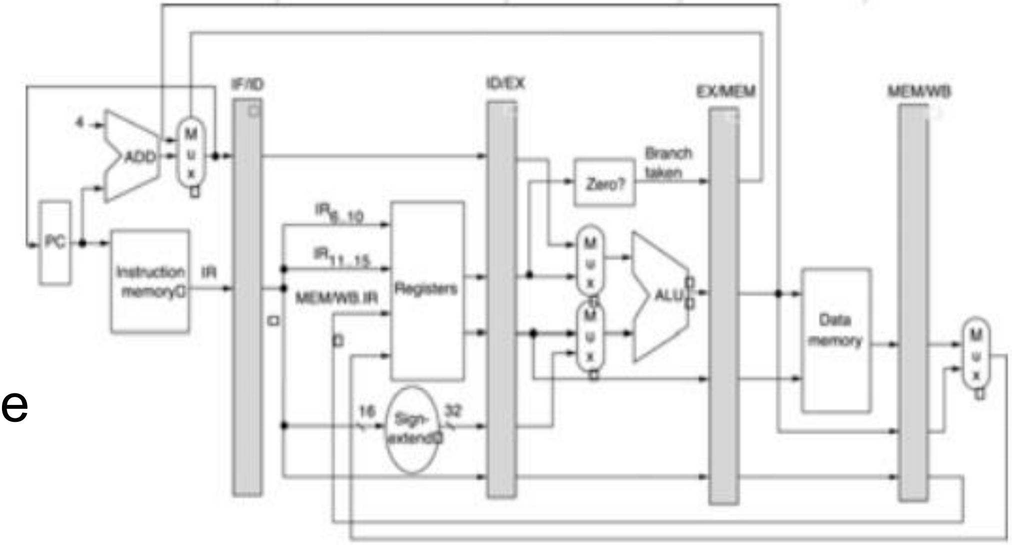


	IC	CPI	CCT
Program	√		
Compiler	√		
ISA	√	√	
HW organization		√	√
HW technology			√

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

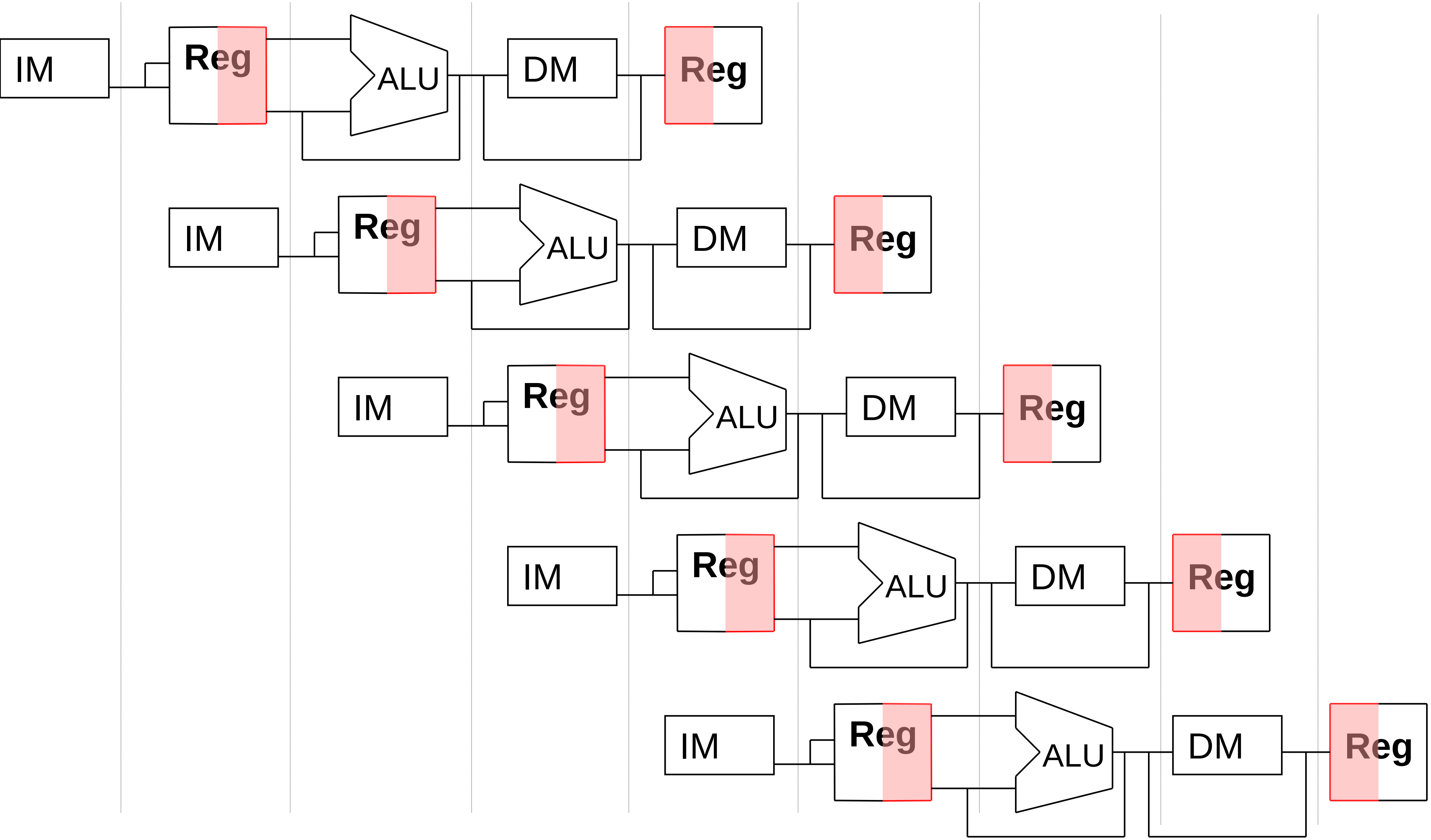




Performance limitations

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

Pipeline Resources



Pipeline Hazards and Their Classification

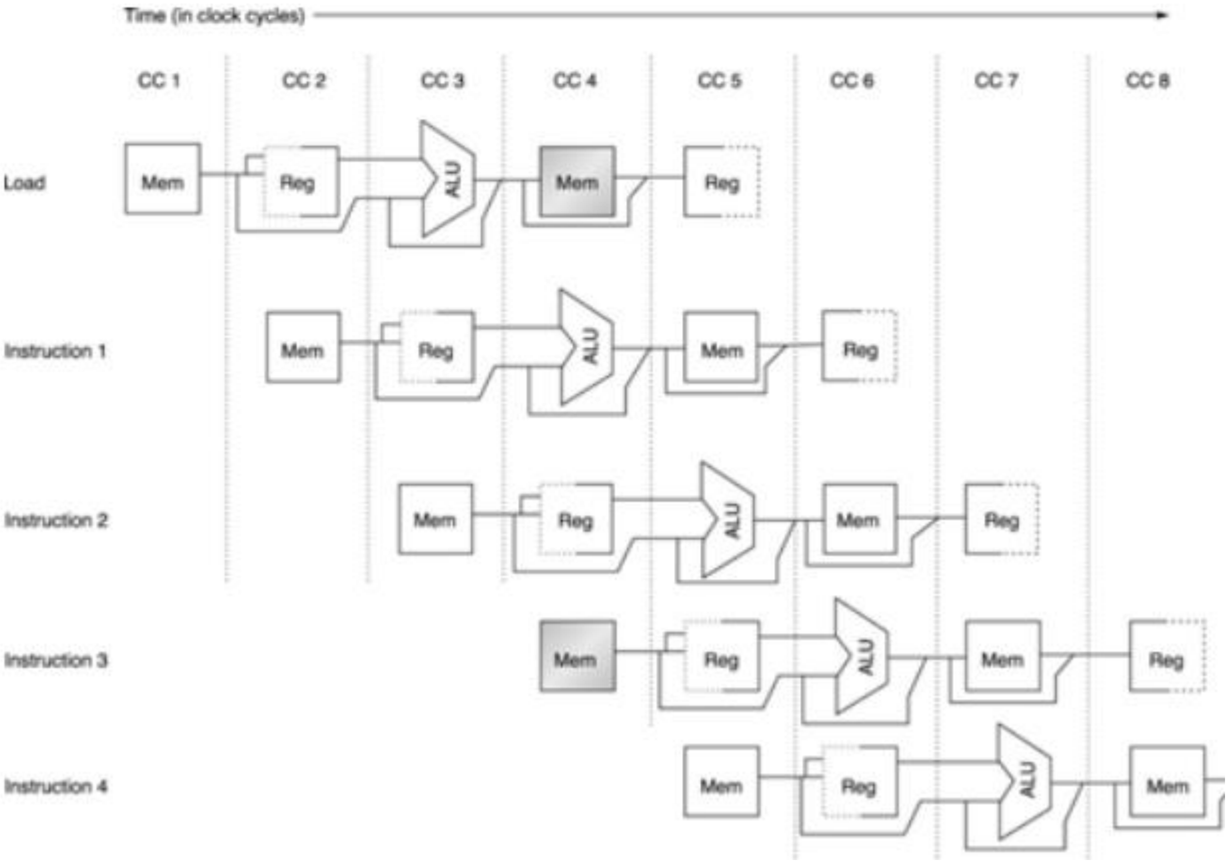
- Pipeline hazard situation
  - “the next instruction cannot execute in the following clock cycle”
- Pipeline hazards come in three different flavors
  - Structural Hazards
    - arise from resource conflict: the HW cannot support all possible instruction combinations simultaneously in overlapped execution
  - 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

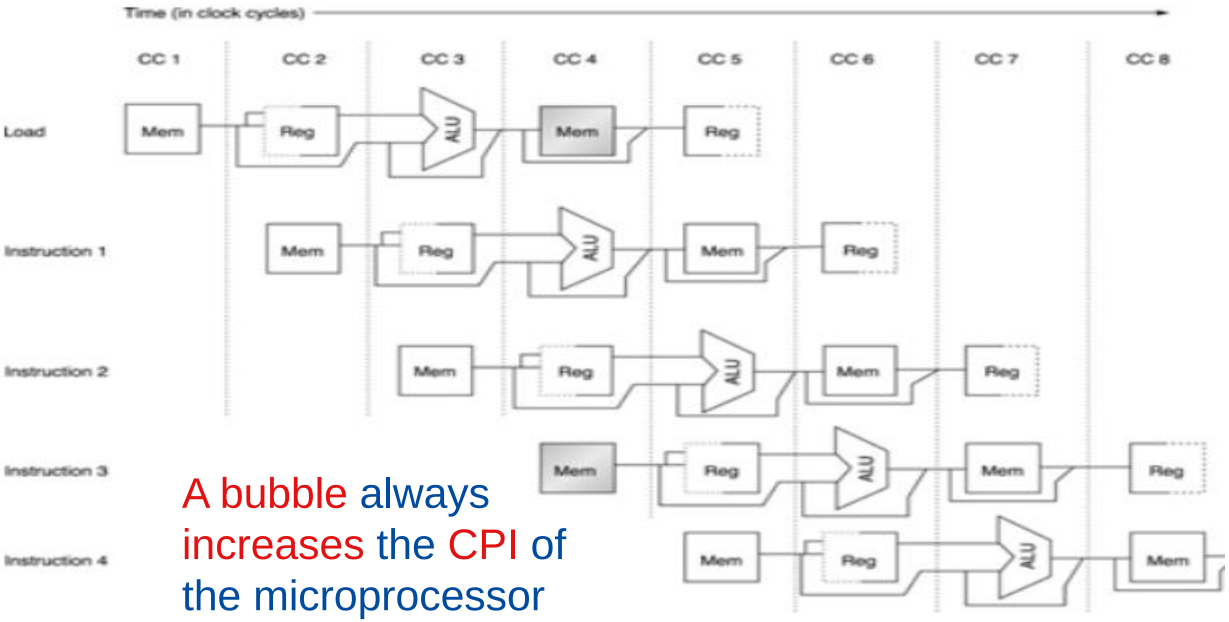
- Structural hazards are due to resource constraints:
  - some resources are not duplicated enough to allow all combination of instructions in the pipeline to execute
    - e.g., the architecture is not Harvard-like
  - a functional unit is not fully pipelined
    - e.g., an instruction takes more than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



A bubble always increases the CPI of the microprocessor implementation

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

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

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

- 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)

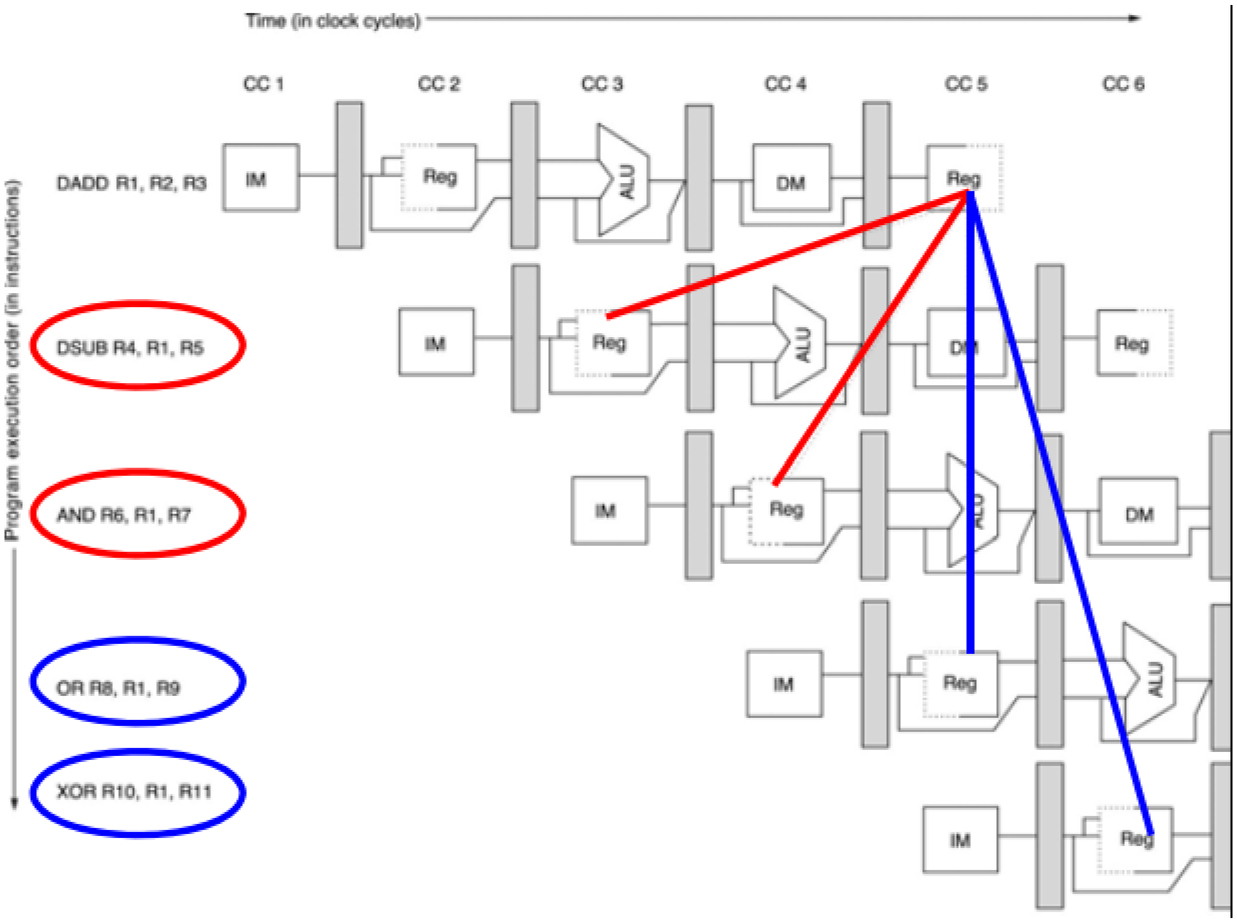
$$\mathrm{speedupFromPipelining} = \frac{\mathrm{pipelineDepth}}{1 + \mathrm{pipelineStallCyclesPerInstruction}}$$

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?
  - (averageInstructionTime)_noHaz = 1 * 1 = 1
  - (averageInstructionTime)_haz = (1 + 0.4 * 1) * 0.9 = 1.26
  - implementation without structural hazard is 1.26 times faster than the other

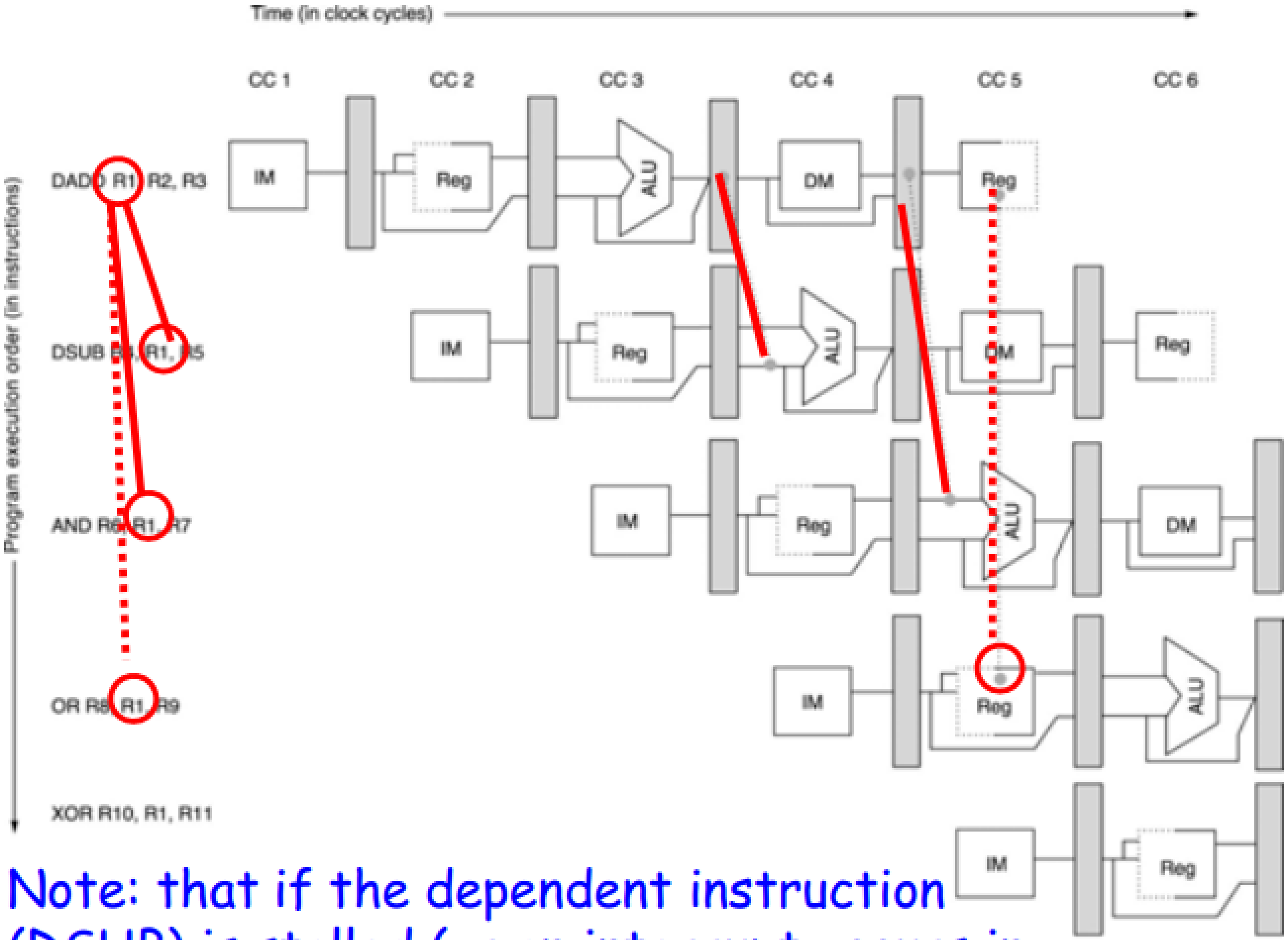
Data Hazards

- 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



Note: that if the dependent instruction (DSUB) is stalled (or an interrupt occurs in between the two instructions) then the forward path will not be activated

Forwarding to the ALU

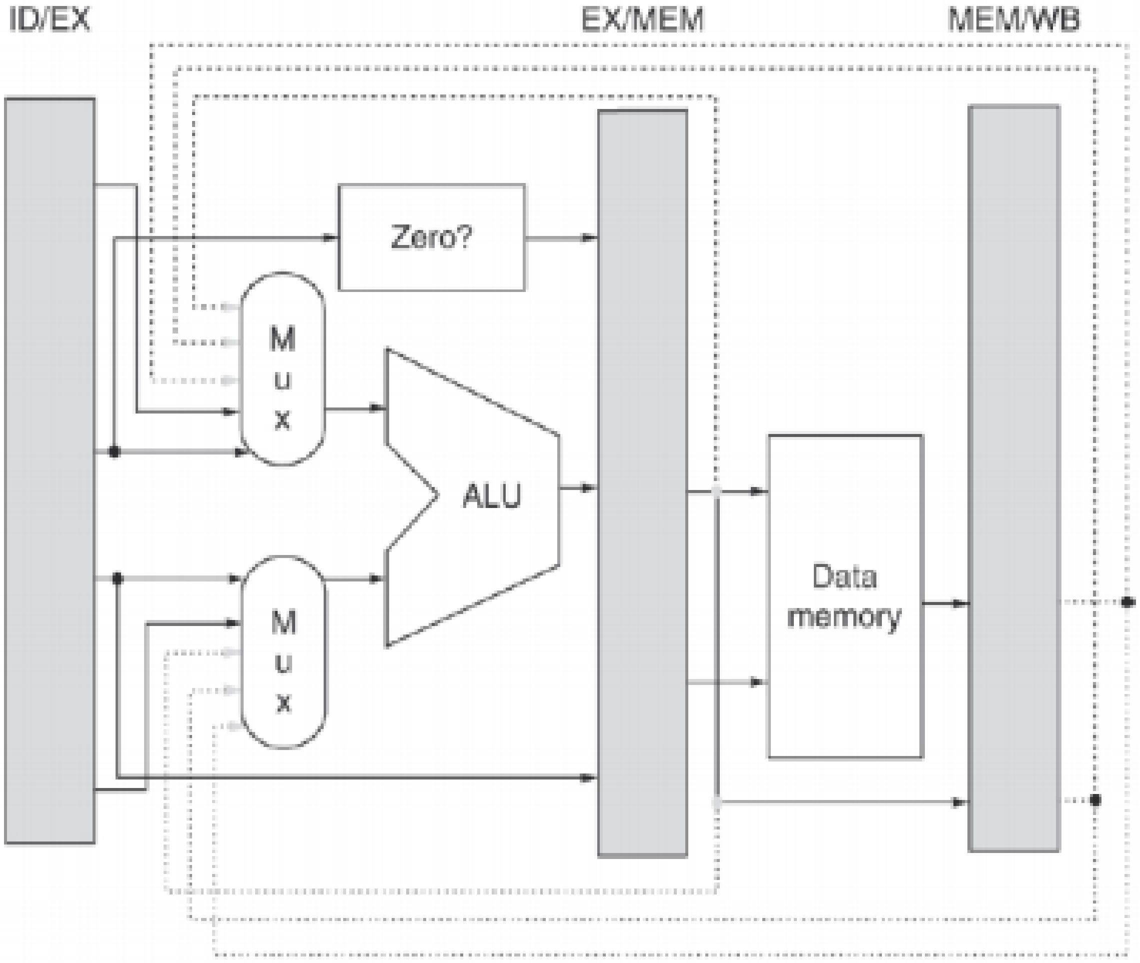
- 3 inputs added to the MUX

- 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)

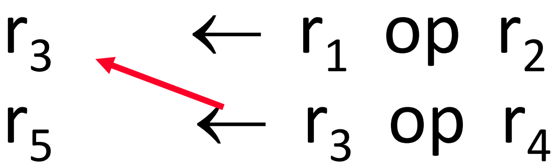
- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program are correct
  - Flow dependences always need to be obeyed because they constitute true dependence on a value
  - Anti and output dependences exist due to limited number of architectural registers
    - 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

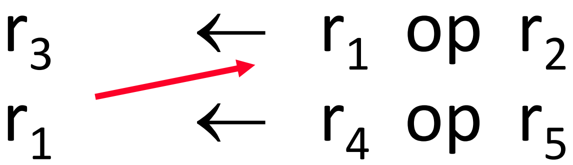


$r_3 \leftarrow r_1 \text{ op } r_2$

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

Read-after-Write (RAW)

Anti dependence

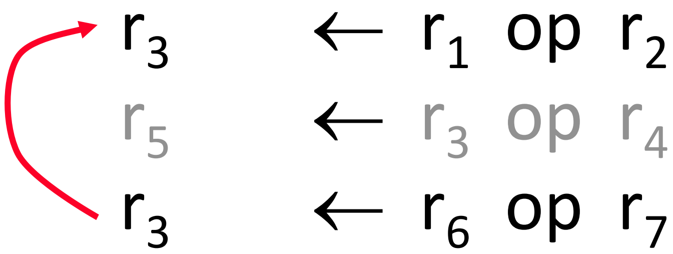


$r_3 \leftarrow r_1 \text{ op } r_2$

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

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)

Data hazards in the pipeline diagram

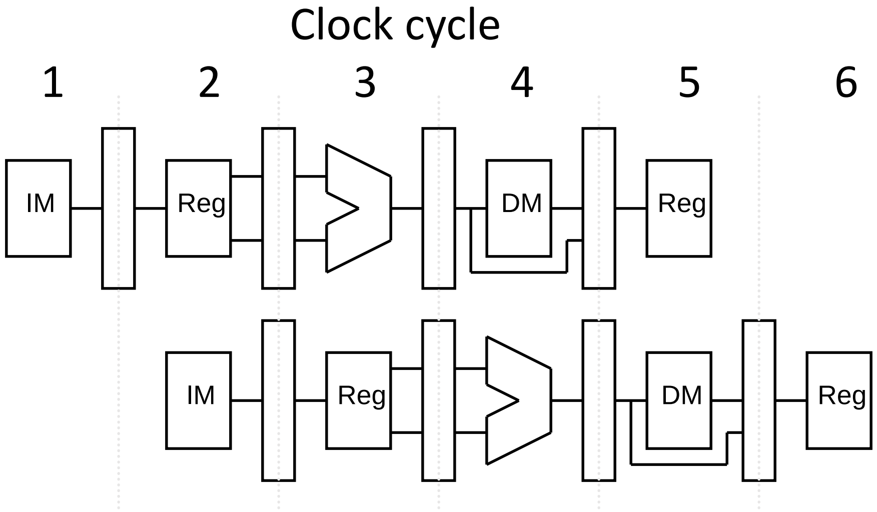
Clock cycle

	1	2	3	4	5	6	7	8	9
sub $2, $1, $3	IF	ID	EX	MEM	WB				
and $12, $2, $5		IF	ID	EX	MEM	WB			
or $13, $6, $2			IF	ID	EX	MEM	WB		
add $14, $2, $2				IF	ID	EX	MEM	WB	
sw $15, 100($2)					IF	ID	EX	MEM	WB

- The SUB instruction does not write to register $2 until clock cycle 5. This causes two data hazards in our current pipelined datapath.
  - The AND reads register $2 in cycle 3. Since SUB hasn't modified the register yet, this will be the *old* value of $2, not the new one.
  - Similarly, the OR instruction uses register $2 in cycle 4, again before it's actually updated by SUB.

What about loads?

- Imagine if the first instruction in the example was LW instead of SUB.
  - How does this change the data hazard?

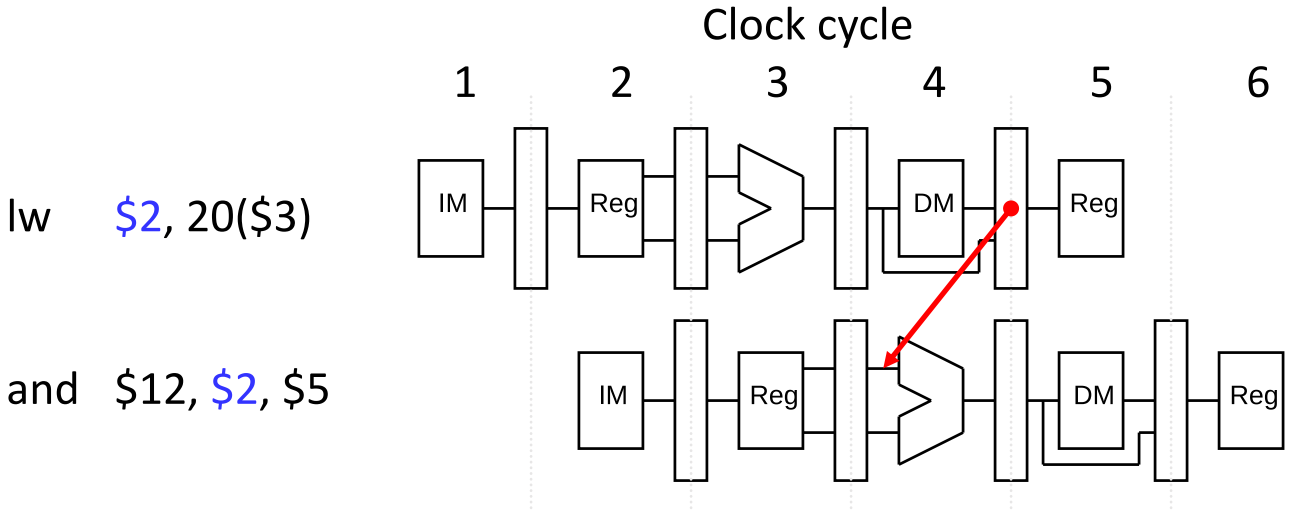


lw $2, 20($3)

and $12, $2, $5

What about loads?

- Imagine if the first instruction in the example was LW instead of SUB.
  - The load data doesn't come from memory until the *end* of cycle 4.
  - But the AND needs that value at the *beginning* of the same cycle!
- This is a "true" data hazard—the data is not available when we need it.

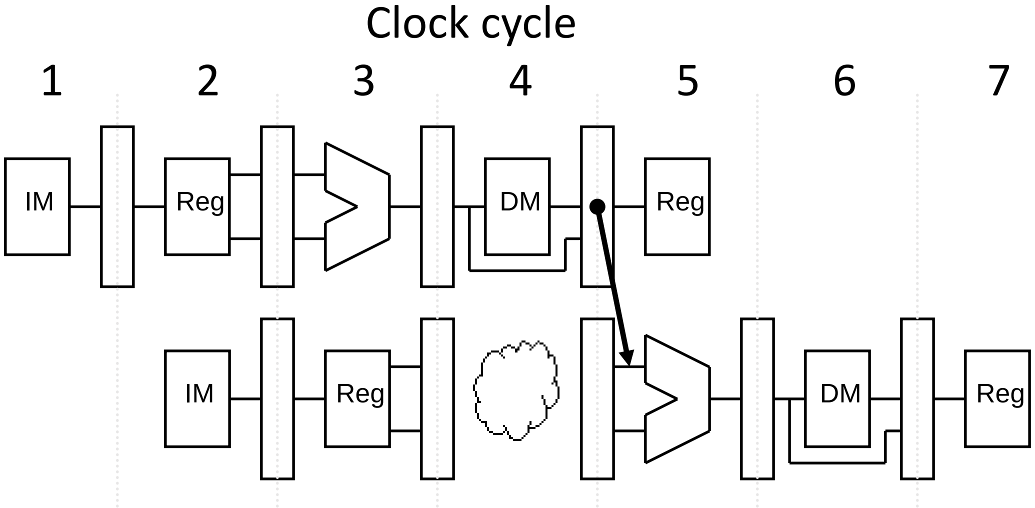


Stalling

- The easiest solution is to stall the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a bubble.

lw $2, 20($3)

and $12, $2, $5



- Notice that we’re still using forwarding in cycle 5, to get data from the MEM/WB pipeline register to the ALU.