

# CS510 Computer Architecture

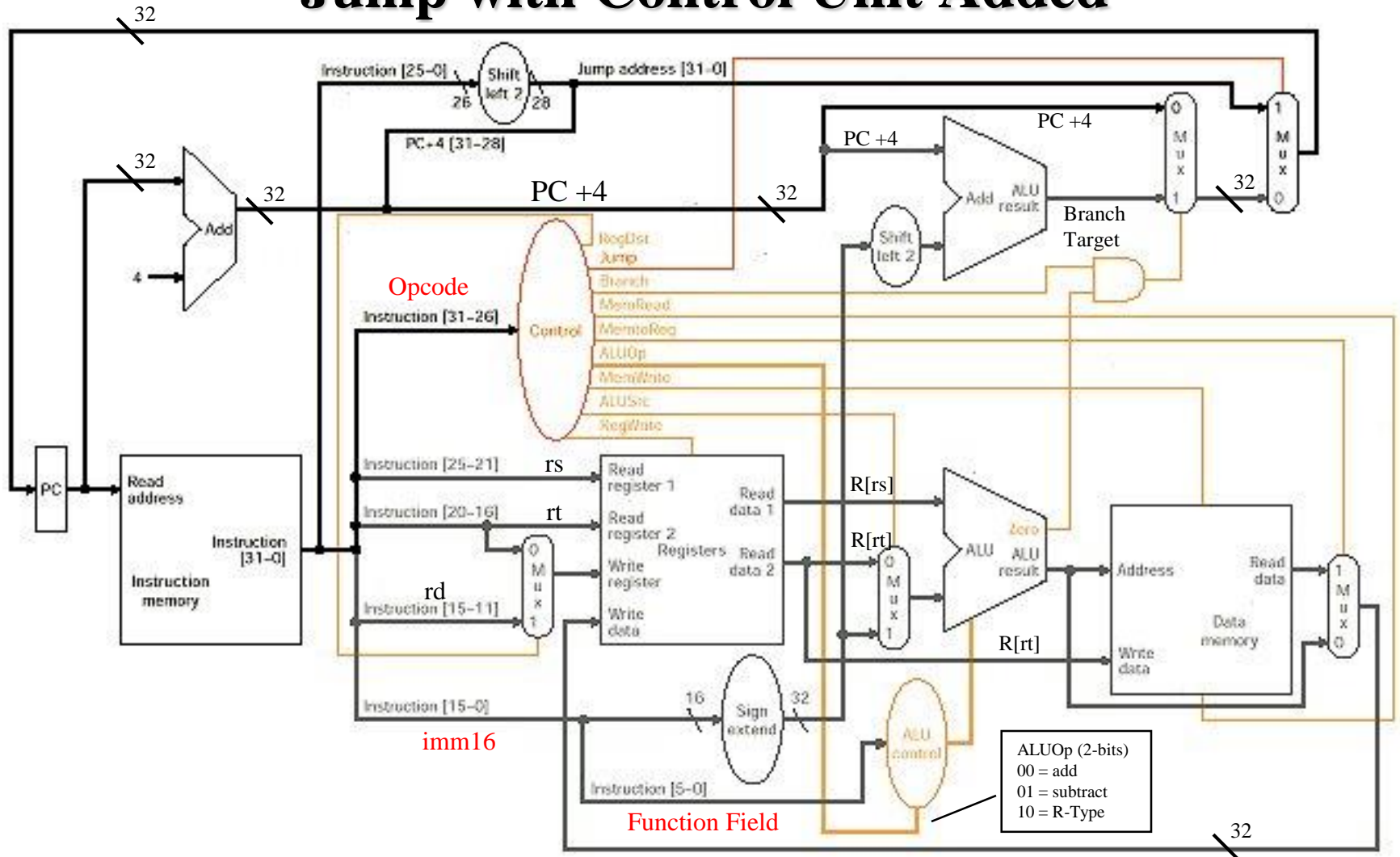
## Lecture 05: CPU Pipelining

**Soontae Kim**

**Spring 2017**

**School of Computing, KAIST**

# Single Cycle MIPS Datapath Extended To Handle Jump with Control Unit Added



This is book version ORI not supported, no zero extend of immediate needed

# Instruction Pipelining Review

- Instruction pipelining is CPU implementation technique where multiple operations of a number of instructions are overlapped.
- An instruction execution pipeline involves a number of steps, where each step completes a part of an instruction. Each step is called *a pipeline stage*.
- The stages or steps are connected in a linear fashion: one stage to the next to form the pipeline -- instructions enter at one end and progress through the stages and exit at the other end.
- The time to move an instruction one step down the pipeline is equal to *the machine cycle* and is determined by the stage with the longest processing delay.
- Pipelining increases the CPU instruction throughput: The number of instructions completed per unit time.
  - Under ideal conditions (no stall cycles), instruction throughput is one instruction per machine cycle, or ideal  $CPI = 1$
- Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called instruction completion latency).
  - Minimum instruction latency =  $n$  cycles, where  $n$  is the number of pipeline stages

The pipeline described here is called **an in-order** pipeline because instructions are executed in the original program order

# MIPS In-Order Single-Issue Integer Pipeline

## Ideal Operation (No stall cycles)

Instruction Number	Time in clock cycles →								
	1	2	3	4	5	6	7	8	9
Instruction I	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

← Time to fill the pipeline →

### MIPS Pipeline Stages:

**IF** = Instruction Fetch  
**ID** = Instruction Decode  
**EX** = Execution  
**MEM** = Memory Access  
**WB** = Write Back

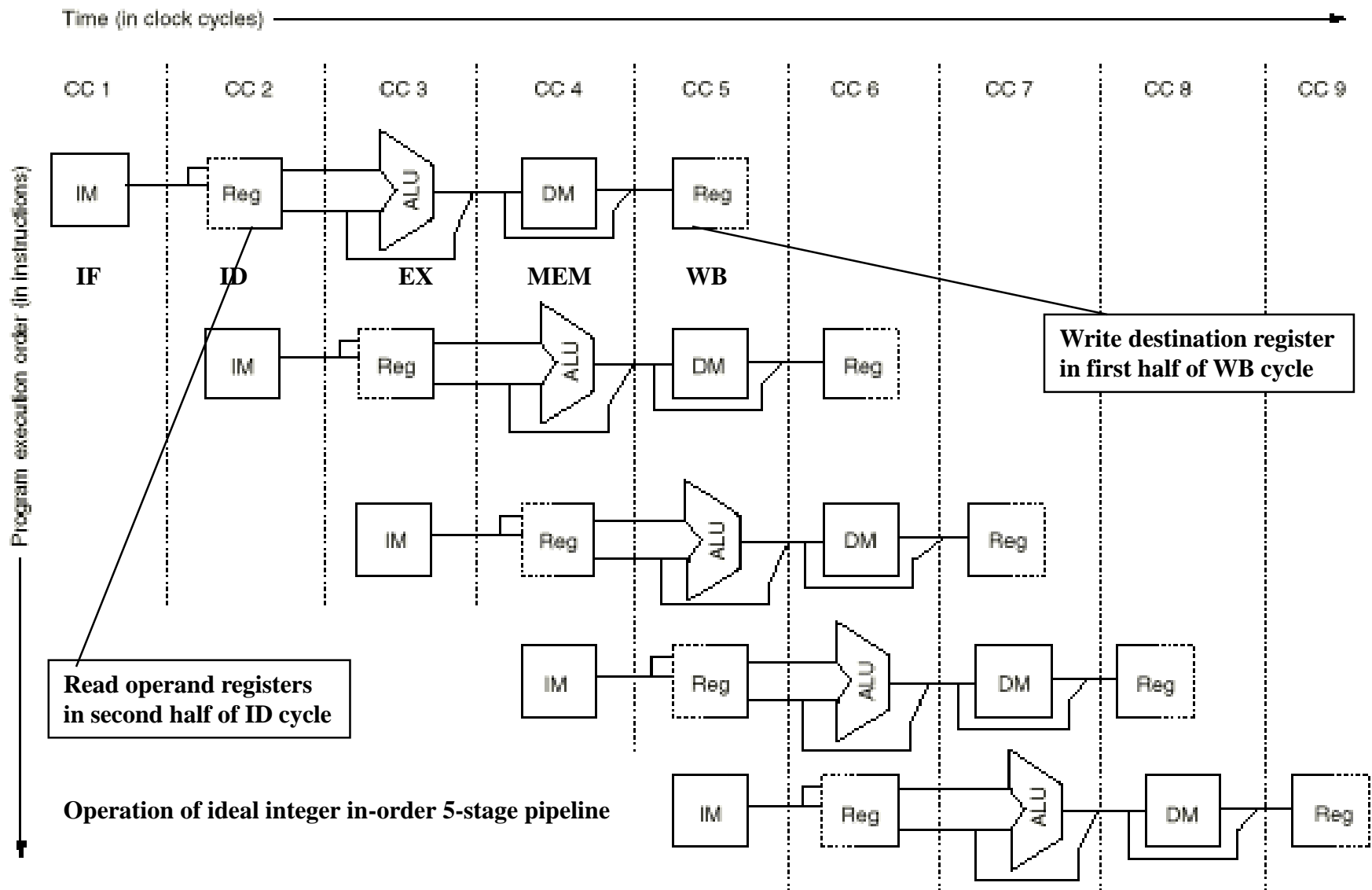
First instruction, I  
Completed

Last instruction,  
I+4 completed

n = 5 pipeline stages

Ideal CPI = 1

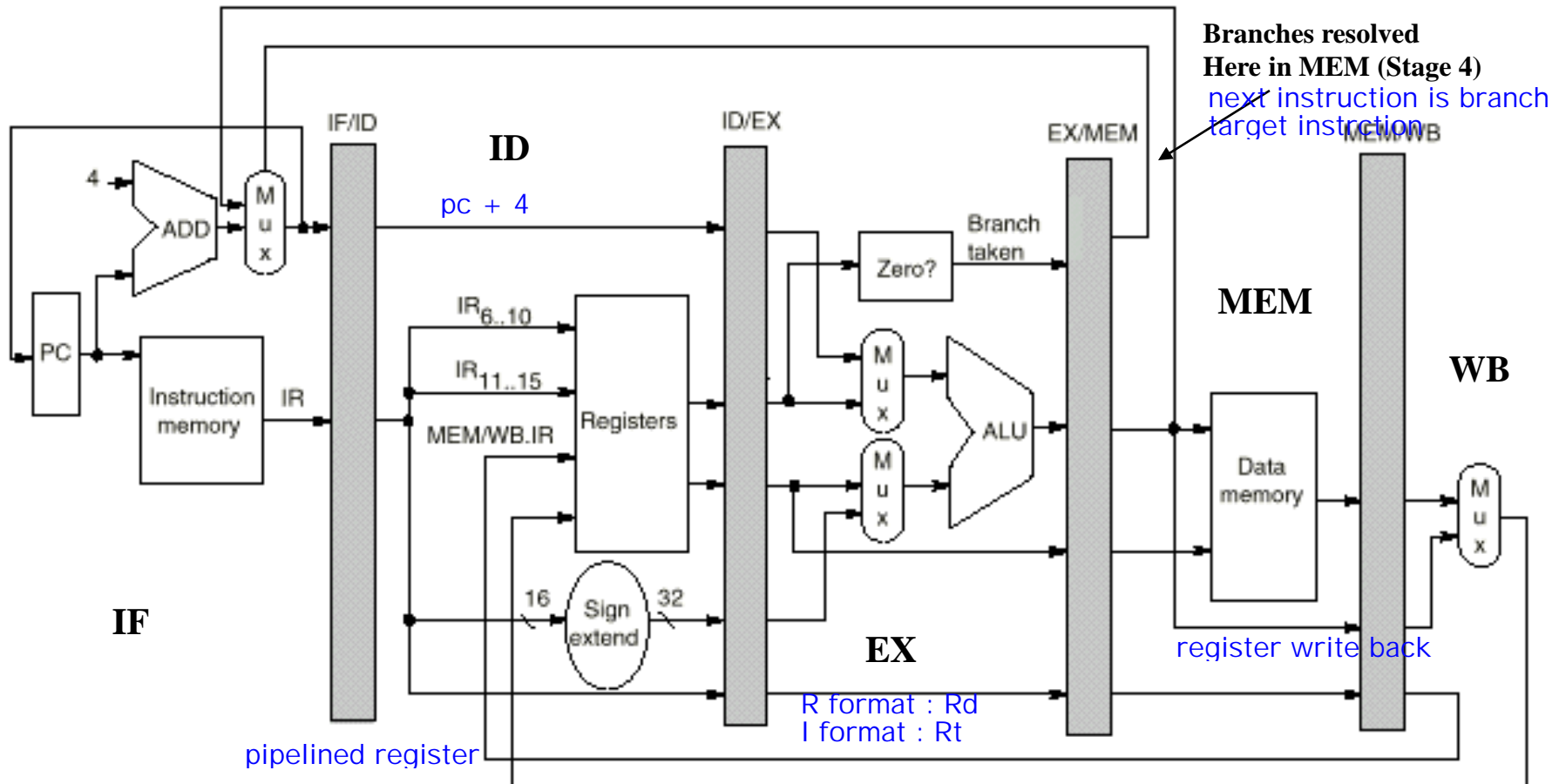
In-order = instructions executed in original program order



The pipeline can be thought of as a series of datapaths shifted in time.

# A Pipelined MIPS Integer Datapath

- Assume register writes occur in first half of cycle and register reads occur in second half.



The datapath is pipelined by adding a set of registers, one between each pair of pipe stages.

Branch Penalty = 4 - 1 = 3 cycles

# Pipeline Hazards

- Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle possibly resulting in one or more stall (or wait) cycles.
- Hazards reduce the ideal speedup (increase  $CPI > 1$ ) gained from pipelining and are classified into three classes:
  - Structural hazards: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions. solve : by adding more hardware
  - Data hazards: Arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline we can't resolve this !
  - Control hazards: Arise from the pipelining of conditional branches and other instructions that change the PC

# Performance of Pipelines with Stalls

- Hazard conditions in pipelines may make it necessary to stall the pipeline by a number of cycles degrading performance from the ideal pipelined CPI of 1.

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall cycles per instruction} \\ &= 1 + \text{Pipeline stall cycles per instruction}\end{aligned}$$

- If pipelining overhead is ignored and we assume that the stages are perfectly balanced then speedup from pipelining over multi-cycle CPU is given by:

$$\begin{aligned}\text{Speedup} &= \text{CPI unpipelined} / \text{CPI pipelined} \\ &= \text{CPI unpipelined} / (1 + \text{Pipeline stall cycles per instruction})\end{aligned}$$

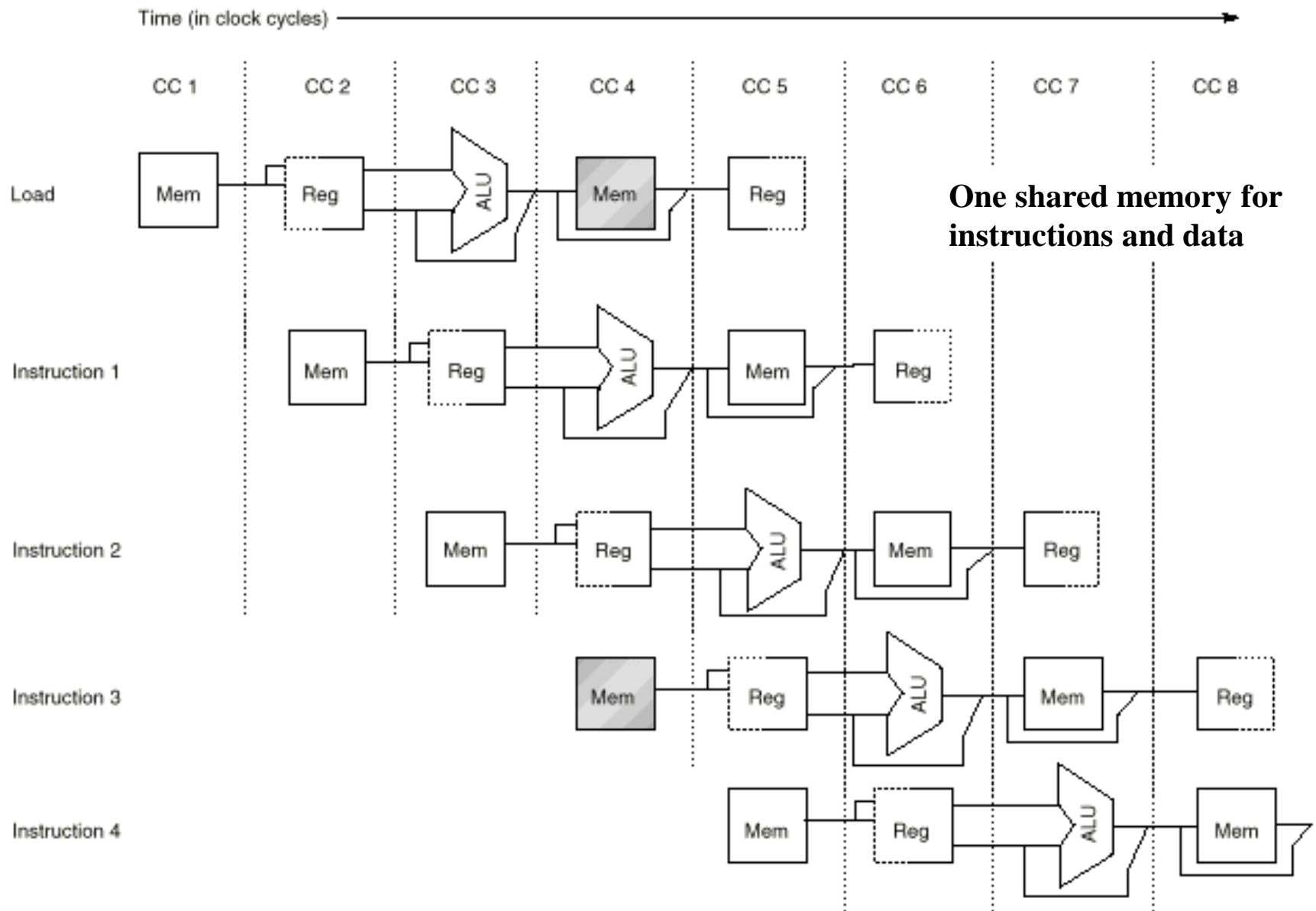
- When all instructions in the unpipelined CPU take the same number of cycles equal to the number of pipeline stages then:

$$\text{Speedup} = \text{Pipeline depth} / (1 + \text{Pipeline stall cycles per instruction})$$



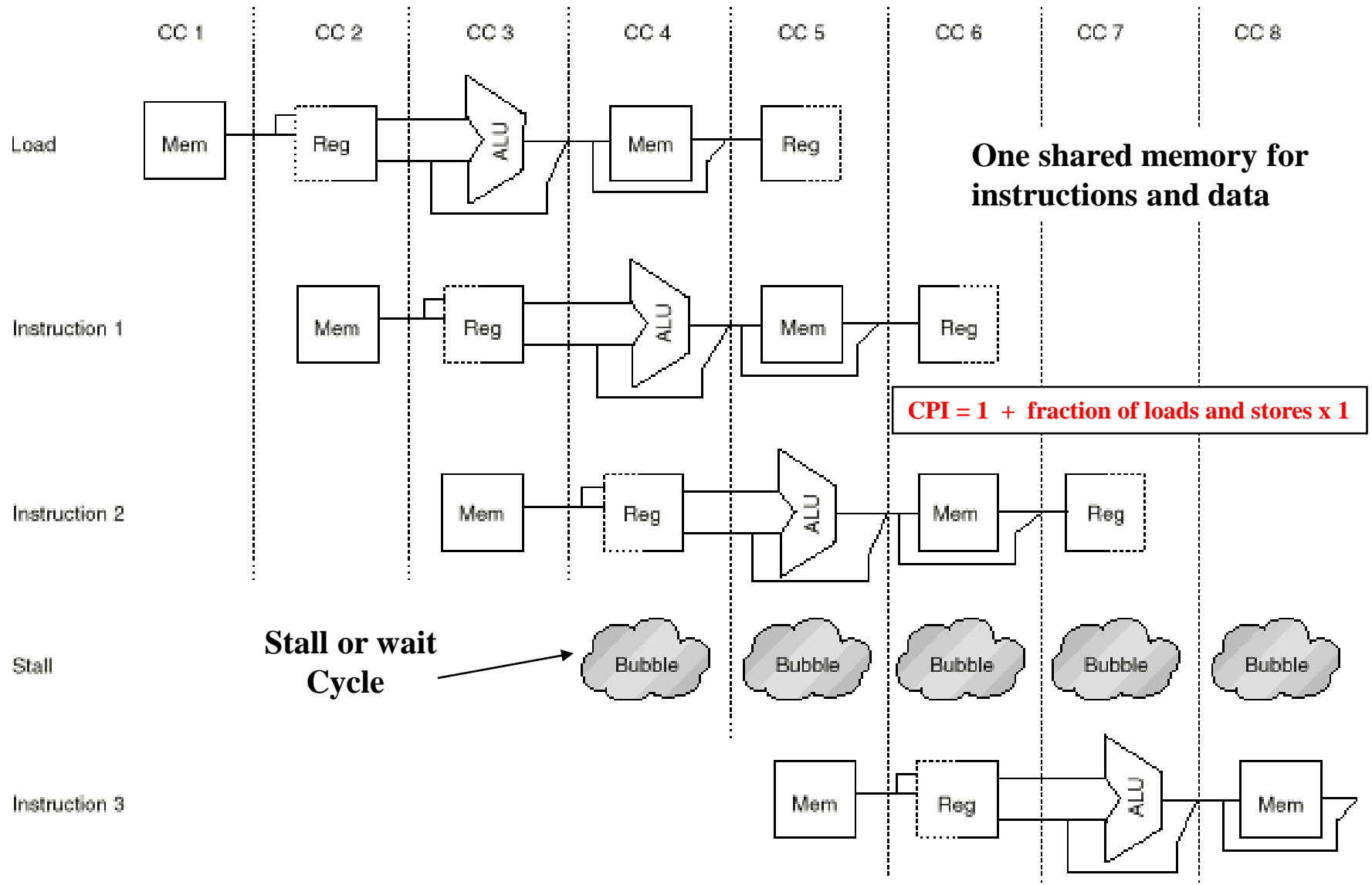
# **Structural Hazards**

- **In pipelined machines overlapped instruction execution requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.**
- **If a resource conflict arises due to a hardware resource being required by more than one instruction in a single cycle, and one or more such instructions cannot be accommodated, then a structural hazard has occurred, for example:**
  - **when a pipelined machine has a shared single-memory for data and instructions.**
  - **stall the pipeline for one cycle for memory data access**



**A machine with only one memory port will generate a conflict whenever a memory reference occurs.**

Time (in clock cycles) →



$$\text{CPI} = 1 + \text{fraction of loads and stores} \times 1$$

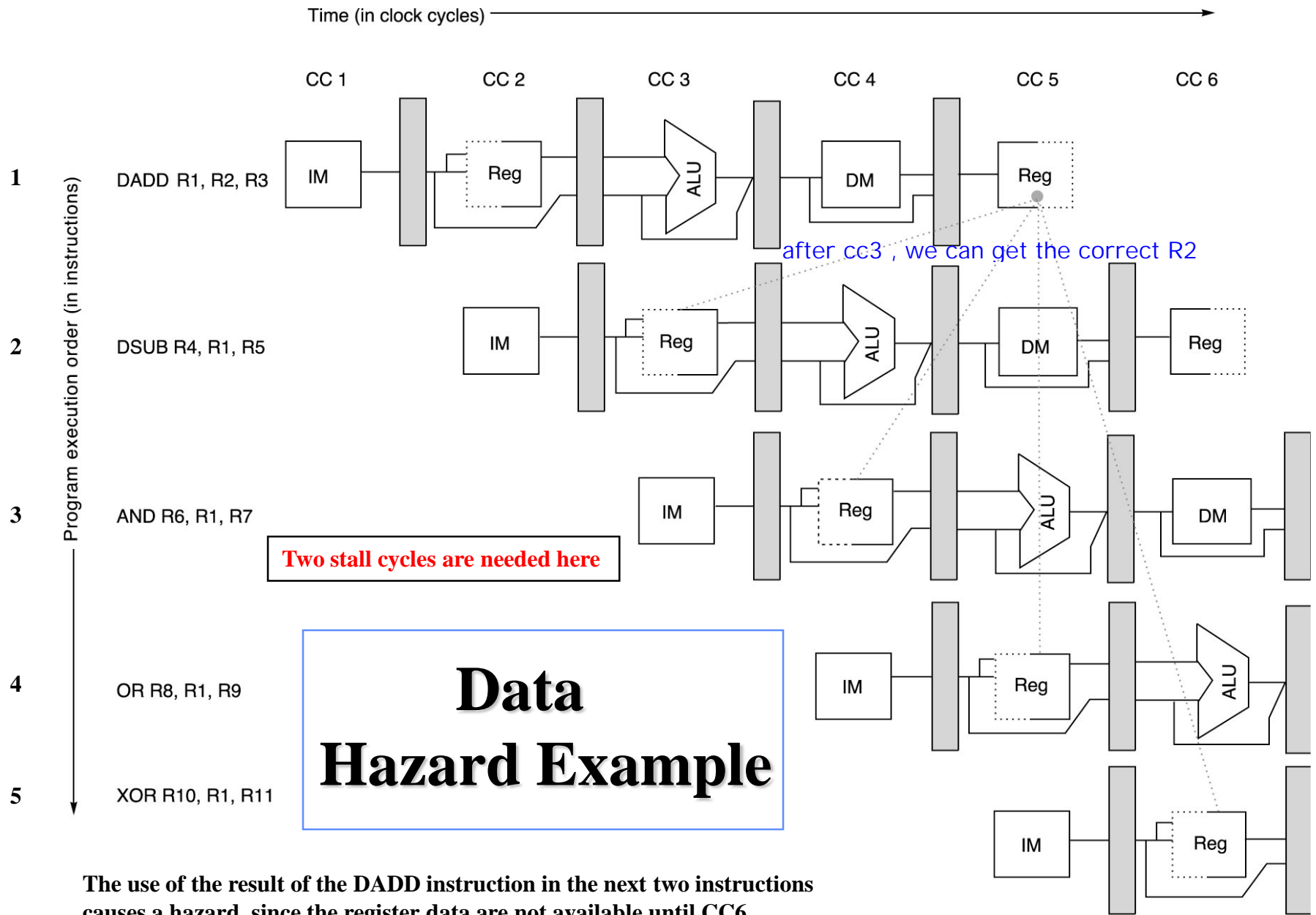
The structural hazard causes pipeline bubbles to be inserted.

# Data Hazards data dependency

- Data hazards occur when the pipeline changes the order of read/write accesses to instruction operands in such a way that the resulting access order differs from the original sequential instruction operand access order of the unpipelined machine resulting in incorrect execution.
- Data hazards may require one or more instructions to be stalled to ensure correct execution.
- Example:

1	DADD R1, R2, R3
2	DSUB R4, R1, R5
3	AND R6, R1, R7
4	OR R8, R1, R9
5	XOR R10, R1, R11

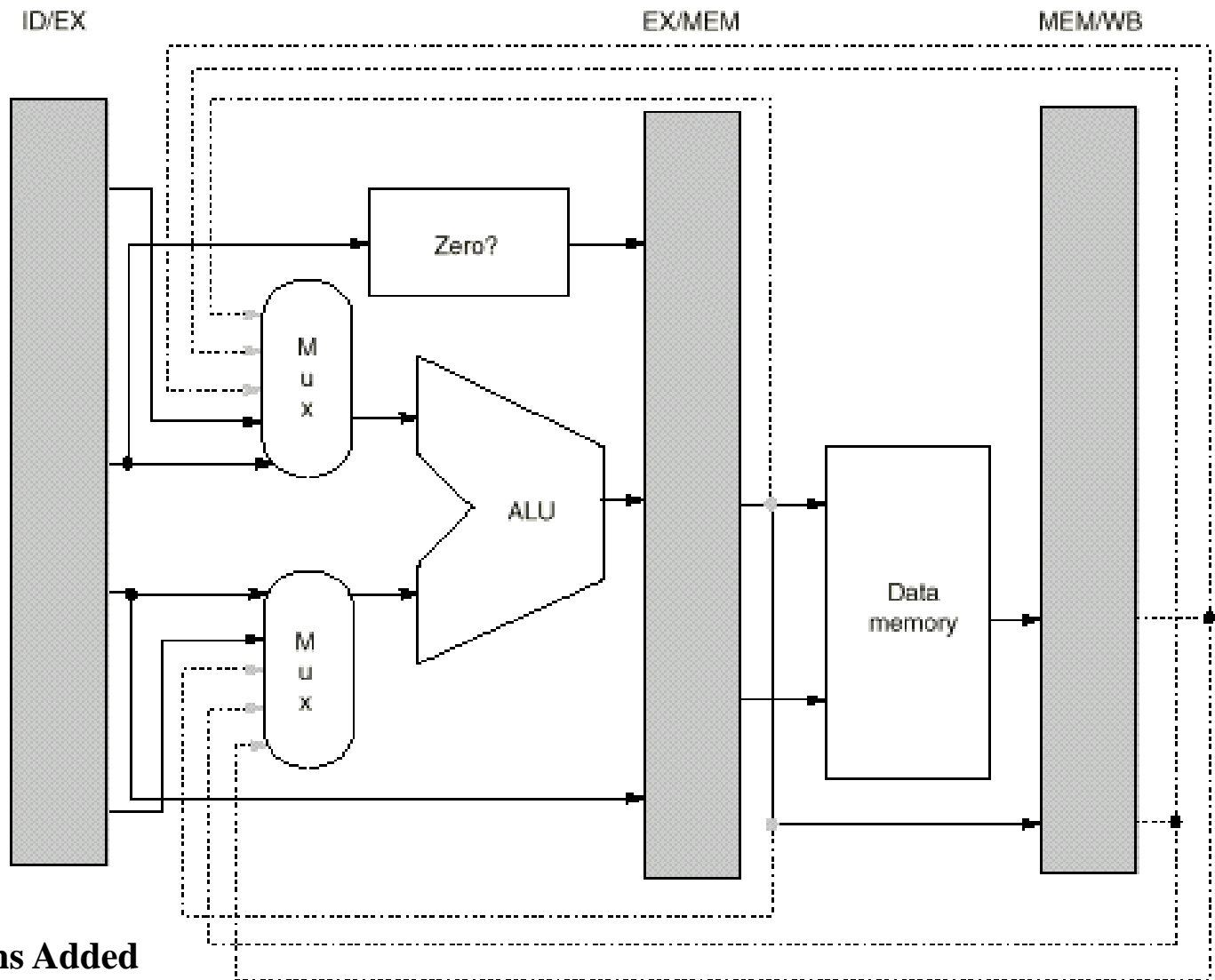
- All the instructions after DADD use the result of the DADD instruction
- DSUB, AND instructions need to be stalled for correct execution.



The use of the result of the DADD instruction in the next two instructions causes a hazard, since the register data are not available until CC6.

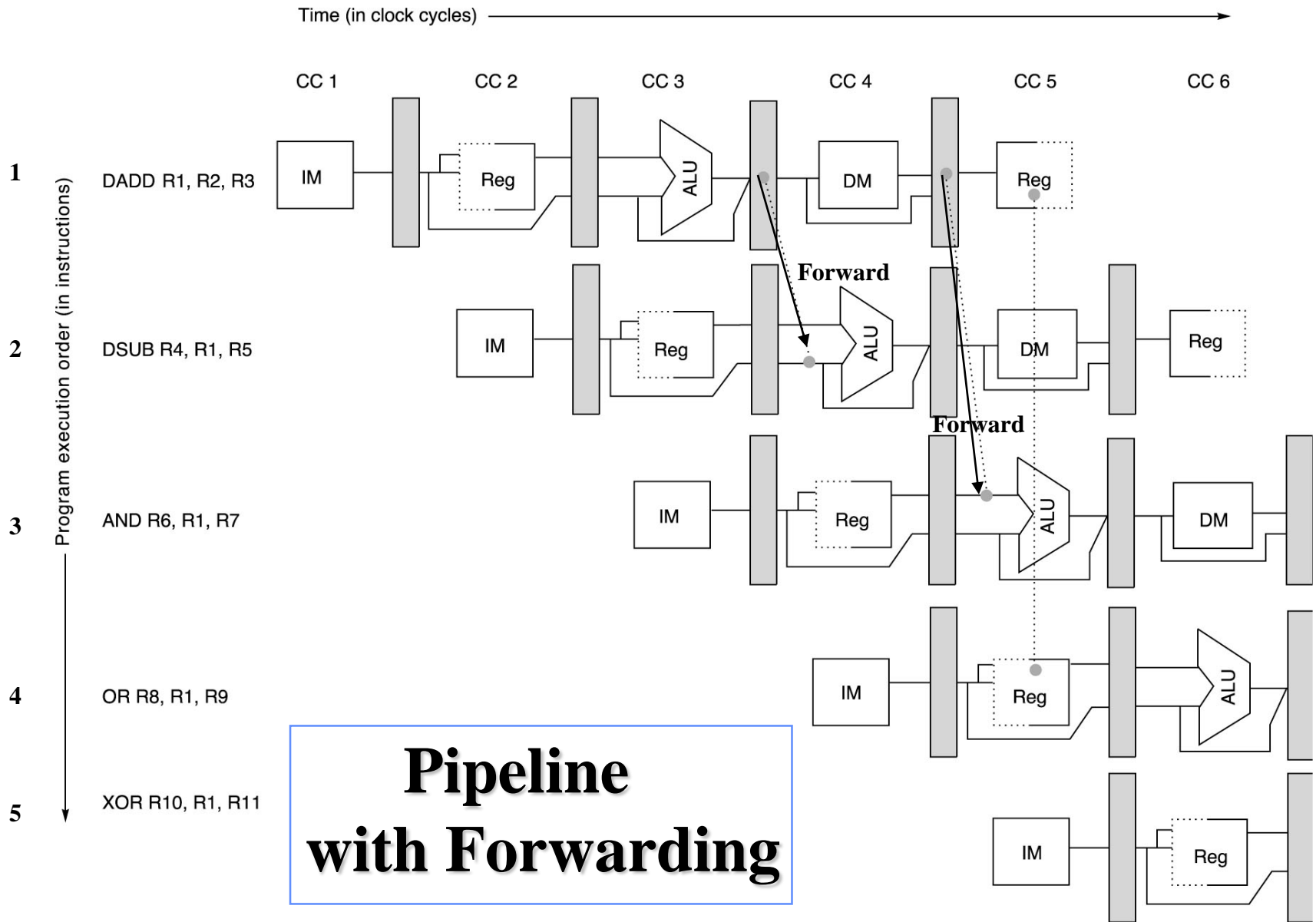
# Minimizing Data hazard Stalls by Forwarding

- Forwarding is a hardware-based technique (also called register bypassing or short-circuiting) used to eliminate or minimize data hazard stalls.
- Using forwarding hardware, the result of an instruction is copied directly from where it is produced (ALU, memory read port etc.), to where it is consumed (ALU input register, memory write port etc.)
- For example, in the MIPS integer pipeline with forwarding:
  - The ALU result from the EX/MEM register may be forwarded or fed back to the ALU input as needed instead of reading the register operand value in the ID stage.
  - Similarly, the Data Memory data from the MEM/WB register may be fed back to the ALU input as needed .
  - If the forwarding hardware detects that a previous ALU operation is to write the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.



## Forwarding Paths Added

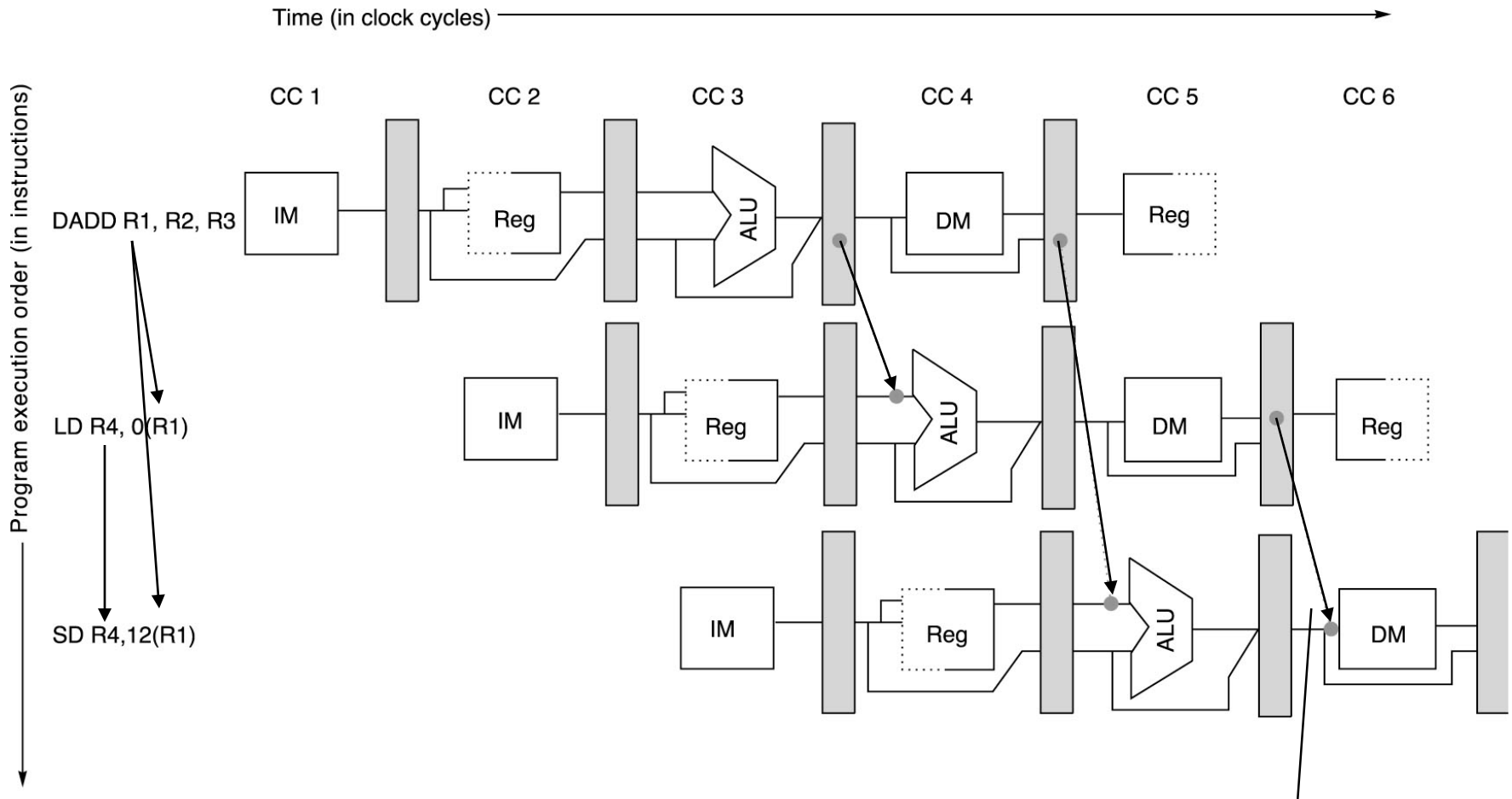
Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs.



**A set of instructions that depend on the DADD result uses forwarding paths to avoid the data hazard**



# Load/Store Forwarding Example

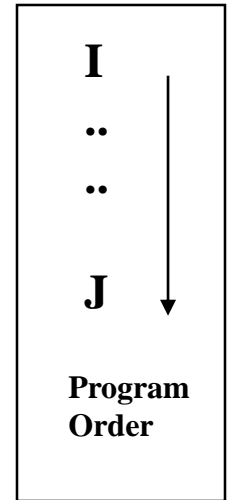


**Forwarding of operand required by store during MEM**

# Data Hazard Classification

Given two instructions  $I$ ,  $J$ , with  $I$  occurring before  $J$  in an instruction stream:

- **RAW (read after write):** *A true data dependence*  
 $J$  tries to read a source before  $I$  writes to it, so  $J$  incorrectly gets the old value.
- **WAW (write after write):** *A name dependence*  
 $J$  tries to write an operand before it is written by  $I$   
The writes are performed in the wrong order.
- **WAR (write after read):** *A name dependence*  
 $J$  tries to write to a destination before it is read by  $I$ ,  
so  $I$  incorrectly gets the new value.
- **RAR (read after read):** Not a hazard.



current MIPS

x

current MIPS

x

# Data Hazards Present in Current MIPS Pipeline

- **Read after Write (RAW) Hazards: Possible?**
  - Results from true data dependencies between instructions.
  - Yes possible, when an instruction requires an operand generated by a preceding instruction.
  - Resolved by:
    - Forwarding or Stalling.
- **Write after Read (WAR) Hazard:**
  - Results when an instruction overwrites the result of an instruction before all preceding instructions have read it.
- **Write after Write (WAW) Hazard:**
  - Results when an instruction writes into a register or memory location before preceding instructions have written its result.
- **Possible? Both WAR and WAW are impossible in the current pipeline.**

**Why?**


  - All instruction operand reads are completed before a following instruction overwrites the operand.
    - Thus WAR is impossible in current MIPS pipeline.
  - All instruction result writes are done in the same program order.
    - Thus WAW is impossible in current MIPS pipeline.

# Data Hazards Requiring Stall Cycles

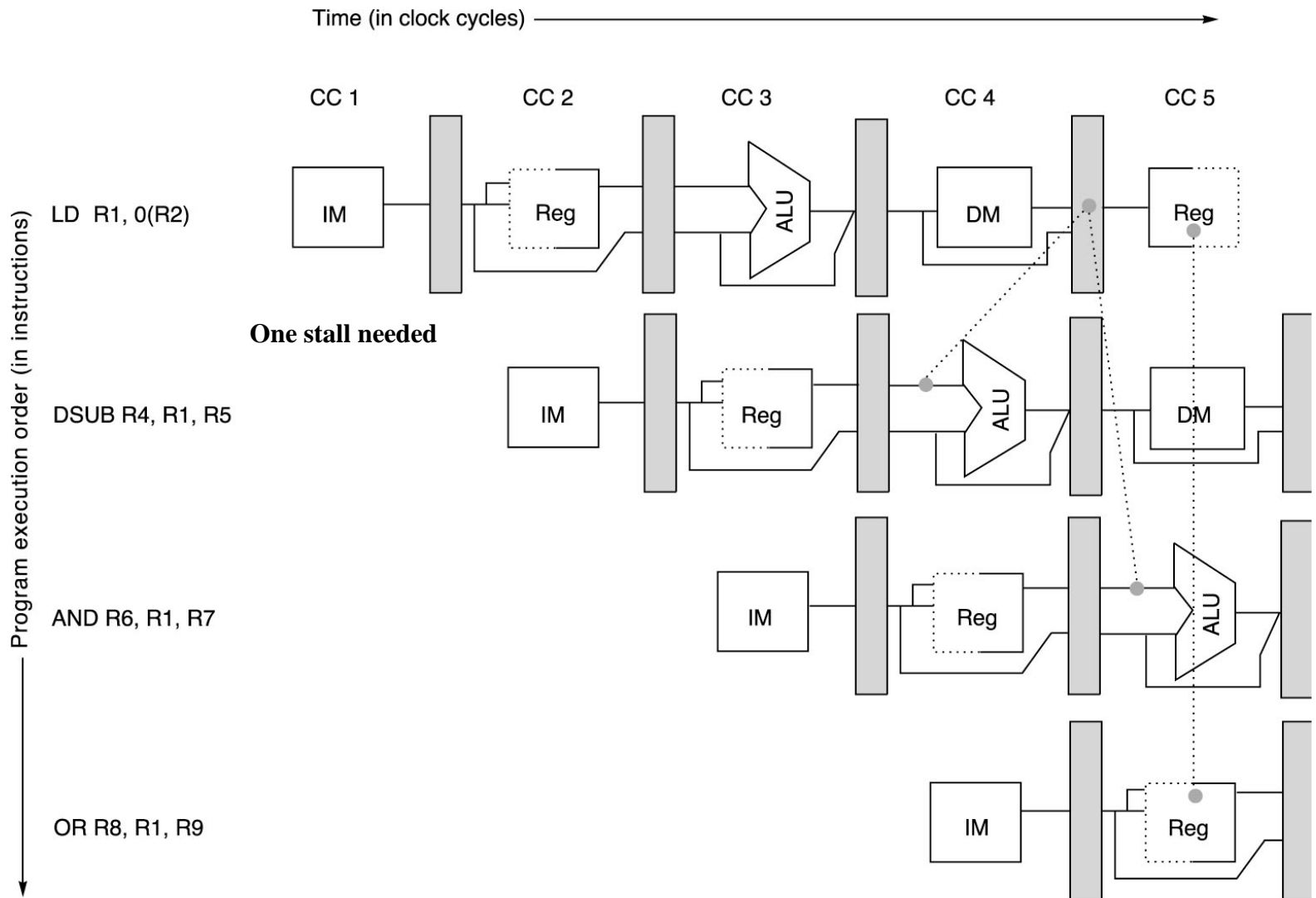
- In some code sequence cases, potential data hazards cannot be handled by forwarding. For example:

load uses hazard

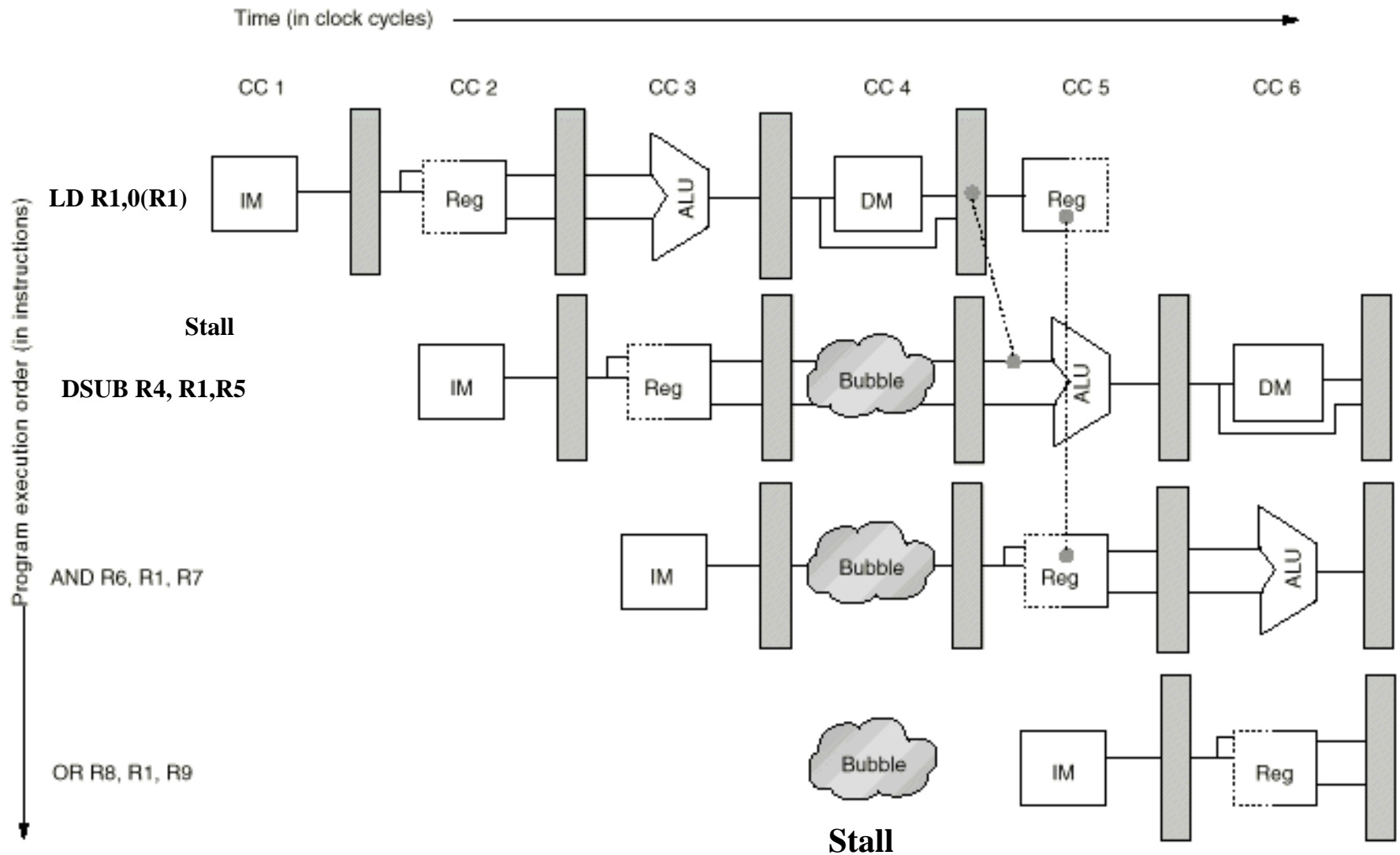
```
LD      R1, 0 (R2)
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
```

A diagram with three arrows pointing from the 'R1' operand of the 'LD' instruction to the 'R1' operand of each of the three subsequent instructions: 'DSUB', 'AND', and 'OR'. This illustrates that all three instructions depend on the value of R1 as written by the 'LD' instruction, creating a data hazard.

- The LD instruction has the data at the end of clock cycle 4 (MEM cycle).
- The DSUB instruction needs the data of R1 in the beginning of that cycle.
- Hazard is prevented by pipeline interlock hardware causing a stall cycle.



The load instruction can bypass its results to the AND and OR instructions, but not to the SUB, since that would mean forwarding the result in "negative time."



The load interlock causes a stall to be inserted at clock cycle 4, delaying the `SUB` instruction and those that follow by one cycle.