CS4617 Computer Architecture

Lecture 8a: Pipelining (continued)

Reference: Appendix C, Hennessy & Patterson

Reference: Hamacher *et al.*

Dr J Vaughan

October 8, 2014

5-stage pipeline

	Clock cycle								
Instr No	**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

Table: Figure C.1: Simple RISC pipeline

Simple RISC Pipeline

- A new instruction is started on each clock cycle
- Must not attempt 2 different ops with same data path resource on same clock cycle
  - e.g., ALU cannot compute ea and subtract at same time

Pipeline as a series of data paths

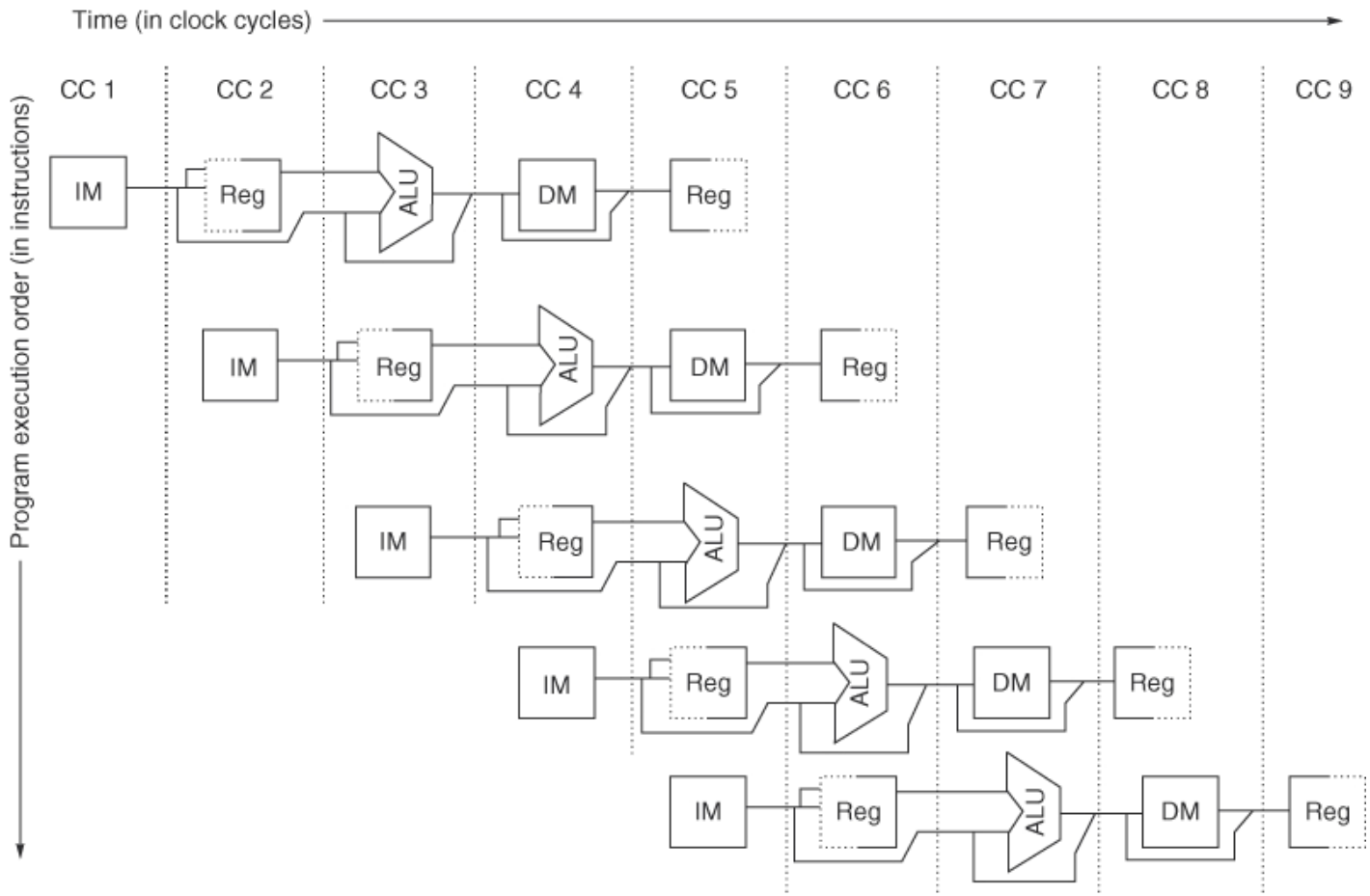

Figure C.2 The pipeline can be thought of as a series of data paths shifted in time. This shows the overlap among the parts of the data path, with clock cycle 5 (CC 5) showing the steady-state situation. Because the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice. We show that it is read in one part of the stage and written in another by using a solid line, on the right or left, respectively, and a dashed line on the other side. The abbreviation IM is used for instruction memory, DM for data memory, and CC for clock cycle.

Pipeline as a series of data paths

There are three reasons for few conflicts: refer to Fig. C2

1. Separate instruction/data memories
   $\Longrightarrow$ No conflict on IF and data memory access
   Note: if pipelined processor has same clock as unpipelined processor, memory system must deliver 5 times the bandwidth.
2. Reg file used for reading in ID and writing in WB
   To handle reads and writes to same reg, perform write in first half of clock cycle and read in second half
3. PC not shown in Fig C2
   IF stage: increment and store PC in every clock cycle
   ID stage: adder must calculate branch target
   Branch does not change PC until ID stage

Simple pipeline practicalities

- Pipeline registers between adjacent stages so that at the end of a clock cycle all results from a stage are stored for use by the following stage on the next clock cycle
  - omitted from figs often but must be present
  - can also be used to carry data between non-adjacent stages
    e.g., Reg value for store is read during ID but not used until MEM – passes through 2 pipeline regs
    Result of an ALU op is computed in EX but not stored until WB – passes through 2 pipeline regs
- Name pipeline regs by connecting stages:
  IF/ID, ID/EX, EX/MEM, MEM/WB
- See Fig C.3

Pipeline registers between stages

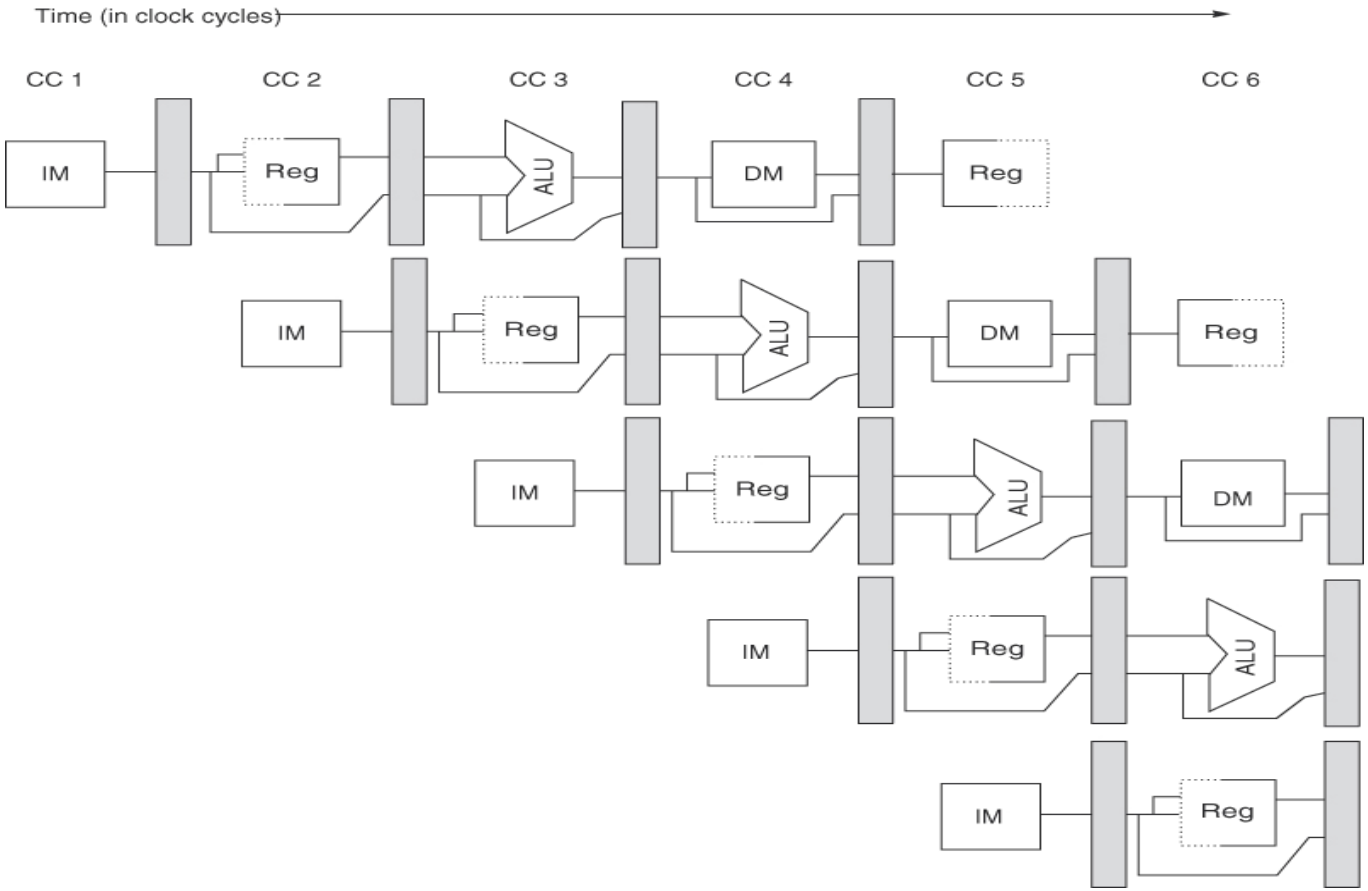

Figure C.3 A pipeline showing the pipeline registers between successive pipeline stages. Notice that the registers prevent interference between two different instructions in adjacent stages in the pipeline. The registers also play the critical role of carrying data for a given instruction from one stage to the other. The edge-triggered property of registers—that is, that the values change instantaneously on a clock edge—is critical. Otherwise, the data from one instruction could interfere with the execution of another!

Pipeline basic issues

- Exec time of individual instructions slightly increased due to pipeline overhead
- Imbalance among pipe stages decreases performance because clock can run no faster than slowest pipe stage
- Overhead is due to pipeline register delay and clock skew
- Pipeline register delay = setup time + propagation delay
- Setup time = time that register input must be stable before the clock signal that triggers a write occurs
  Pipeline registers add setup time and propagation delay to the clock cycle
- Clock skew = max delay between the time the clock arrives at any 2 registers
- Once clock cycle is as small as clock skew + latch overhead, no further pipelining is useful as there is no time left in the cycle for useful work

Example: pipelining

- Unpipelined processor
  - 1ns clock cycle
  - 4 cycles ALU op, branches
  - 5 cycles mem op
  - 40% ALU op, 20% branch, 40% mem op
- Due to clock skew and setup, pipelining adds 0.2 ns to clock
- What is speedup?

Example: pipelining (continued)

- Unpipelined processor
  - Avg instr exec time = clock cycle $\times$ avg CPI
    $= 1\text{ns}\ [(40\% + 20\%)\ 4 + 40\%\ (5)]$
    $= 4.4\text{ns}$
- Pipelined
  - Clock runs at speed of slowest stage and overhead $= 1 + 0.2$
    $= 1.2$ ns
  - Speedup $= \frac{Avg\ \ instr\ \ exec\ \ time\ \ unpipelined}{Avg\ \ instr\ \ exec\ \ time\ \ pipelined}$
    $= \frac{4.4ns}{1.2ns}$
    $= 3.7$
- By Amdahl's Law, overhead limits speed up

Hazards

- Structural: resource conflicts
- Data: data dependencies
- Control: arise from instructions that change the PC (branches/jumps)

Hazard $\implies$ stall pipeline

- All instrs later than stalled instr are also stalled
- All instrs earlier than stalled instr are NOT stalled
  – These must continue in order to clear the hazard
- No new instrs fetched during stall

Performance

- Speed up from pipelining $= \frac{Avg\ \ instr\ \ time\ \ unpipelined}{Avg\ \ instr\ \ time\ \ pipelined}$
  $= \frac{CPI_{up} \times clockcycle_{up}}{CPI_p x clock\ \ cycle_p}$
  $= \frac{CPI_{up}}{CPI_p} \times \frac{Clock\ \ cycle_{up}}{Clock\ \ cycle_p}$ (Ignore cycle time overhead of pipelining)
- CPI_p = Ideal CPI + pipelined stall clock cycles per instr
  = 1 + pipelined stall clock cycles per instr
- Ignore cycle time overhead of pipelining
  $\implies Speedup = \frac{CPI_{up}}{1+Pipeline\ \ stall\ \ cycles\ \ per\ \ instr}$
- If all instrs take same number of cycles
  = number of pipeline stages (also called *depth of pipeline*)
  Then CPI_{up} = Pipeline depth
- $\implies Speedup = \frac{Pipeline\ \ depth}{1+Pipeline\ \ stall\ \ cycles\ \ per\ \ instr}$

Structural hazards

- Pipelining $\Longrightarrow$ overlapped instr execution
  $\Longrightarrow$ Pipeline functional units
  duplicate resources
  $\rightarrow$ allow all possible combinations of instrs in the pipeline
- If $\exists$ instr combination that cannot be allowed due to resource conflict, this causes a structural hazard
- Examples
  1. Functional unit not fully pipelined $\Longrightarrow$ instr sequence using unpipelined unit cannot advance at 1 per clock cycle
  2. Resource duplication not enough to allow all combinations of instrs in the pipeline to execute, e.g,. not enough register file write ports
  3. Single memory for instr and data
     – Data ref conflicts with IF for later instr
     – Fig C4
     – Stall pipeline for 1 clock cycle when data memory access occurs $\Longrightarrow$ pipeline bubble (bubble)

Memory conflict for processor with 1-port memory

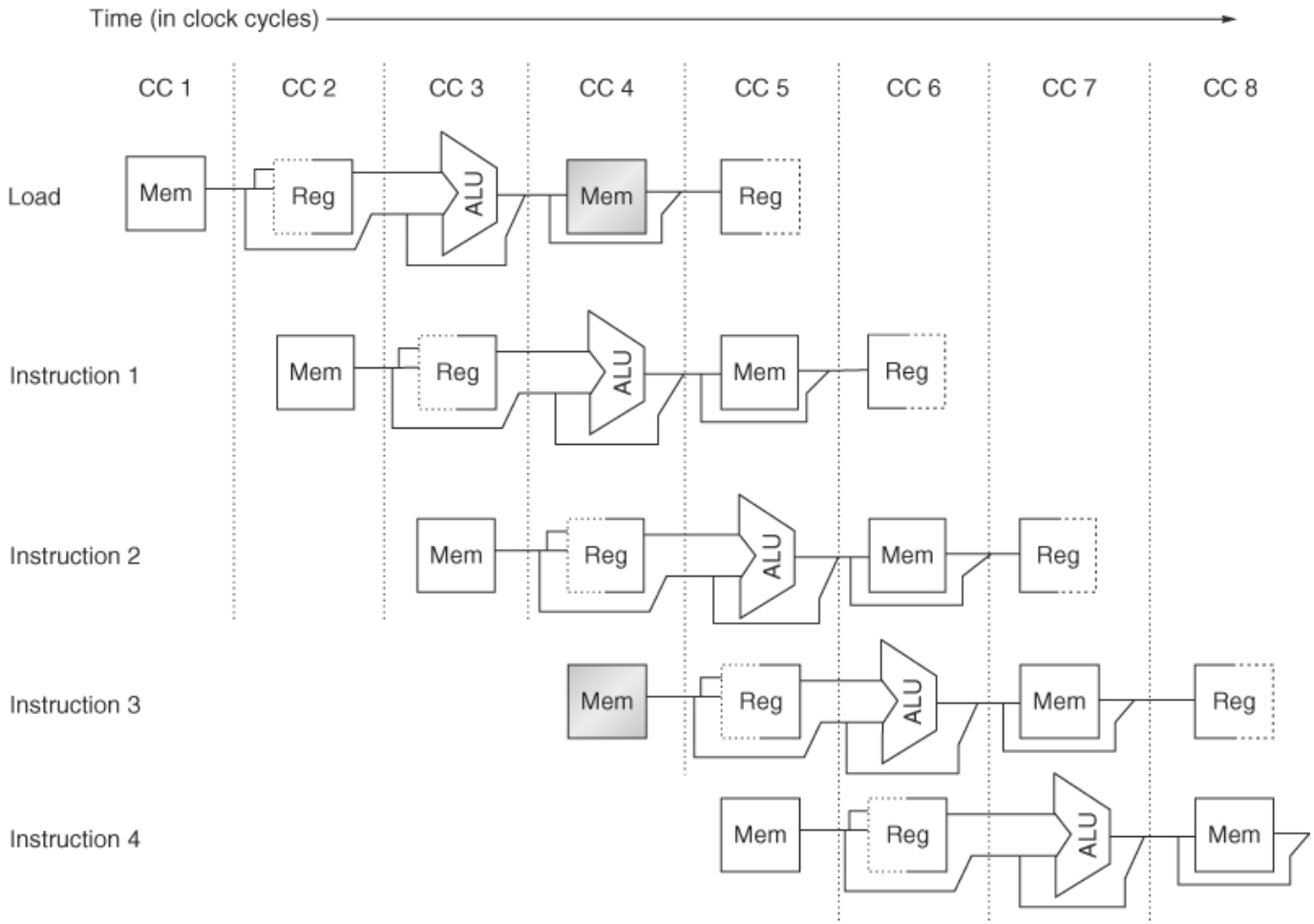

Figure C.4 A processor with only one memory port will generate a conflict whenever a memory reference occurs. In this example the load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory.

Structural hazard stall

	Clock cycle									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load	IF	ID	EX	MEM	WB					
i+1		IF	ID	EX	MEM	WB				
i+2			IF	ID	EX	MEM	WB			
i+3				Stall	IF	ID	EX	MEM	WB	
i+4						IF	ID	EX	MEM	WB
i+5							IF	ID	EX	MEM
i+6								IF	ID	EX

Table: Figure C.5: A pipeline stalled for a structural hazard – a load with one memory port

Example: Cost of load structural hazard

Assume

- Data refs 40%
- $CPI_{pipelined} = 1$
- $Clockrate_{pipelined} = 1.05 \times clockrate_{unpipelined}$
- Is processor faster with or without structural hazard?

Answer

- Avg instr time = CPI $\times$ clock cycle time
- Avg instr time for ideal processor = $clock\ cycle\ time_{ideal}$
- Avg instr time for structural hazard processor = CPI $\times$ clock cycle time $= (1 + 0.4 \times 1) \times \frac{clock\ cycle\ time\ ideal}{1.05}$
  $= 1.3 \times clock\ cycle\ time_{ideal}$
- $\Longrightarrow$ Processor without structural hazard is 1.3 times faster
- $\Longrightarrow$ Either split the cache or use instr buffers
- Designer might allow structural hazards in order to reduce cost of unit

Data Hazards

- Data hazards occur when a pipeline changes the order of read/write accesses to operands so that the order is different to that of an unpipelined processor
- Example code
  - DADD R1, R2, R3
    DSUB R4, R1, R5
    AND R6, R1, R7
    OR R8, R1, R9
    XOR R10, R1, R11
- In Figure C6, DADD writes the value of R1 in the WB stage, but DSUB reads the value during ID
- AND is also affected: Write (R1) does not finish until the end of clock cycle 5
- If AND reads registers in clock cycle 4, it will get wrong result
- XOR and OR operate correctly

Data hazard following DADD

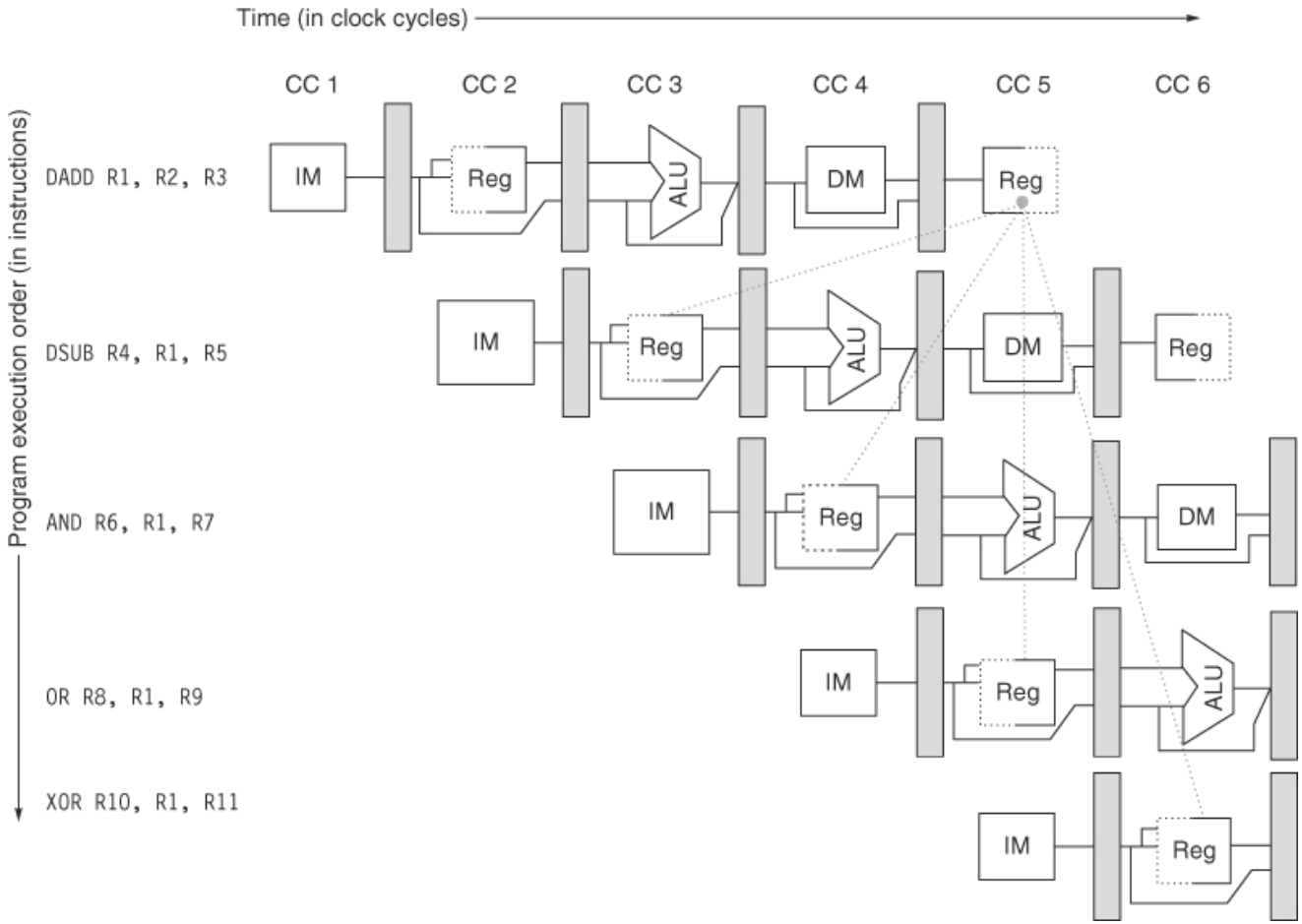

Figure C.6 The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

Forwarding

- The result is not needed by DSUB until after DADD produces it
  1. The ALU result from the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs
  2. If previous ALU op writes a result that is the source for the current ALU op, the forwarded result becomes the ALU input rather than the value read from the register file
- If DSUB stalls, DADD completes and bypass is not activated This also happens if there is an interrupt between DADD and DSUB
- From Figure C6, results may be needed not just from the immediately previous instruction but also possibly from an instruction that started 2 cycles earlier
- Figure C7 shows the example with the bypass in place

Forwarding to deal with data hazard

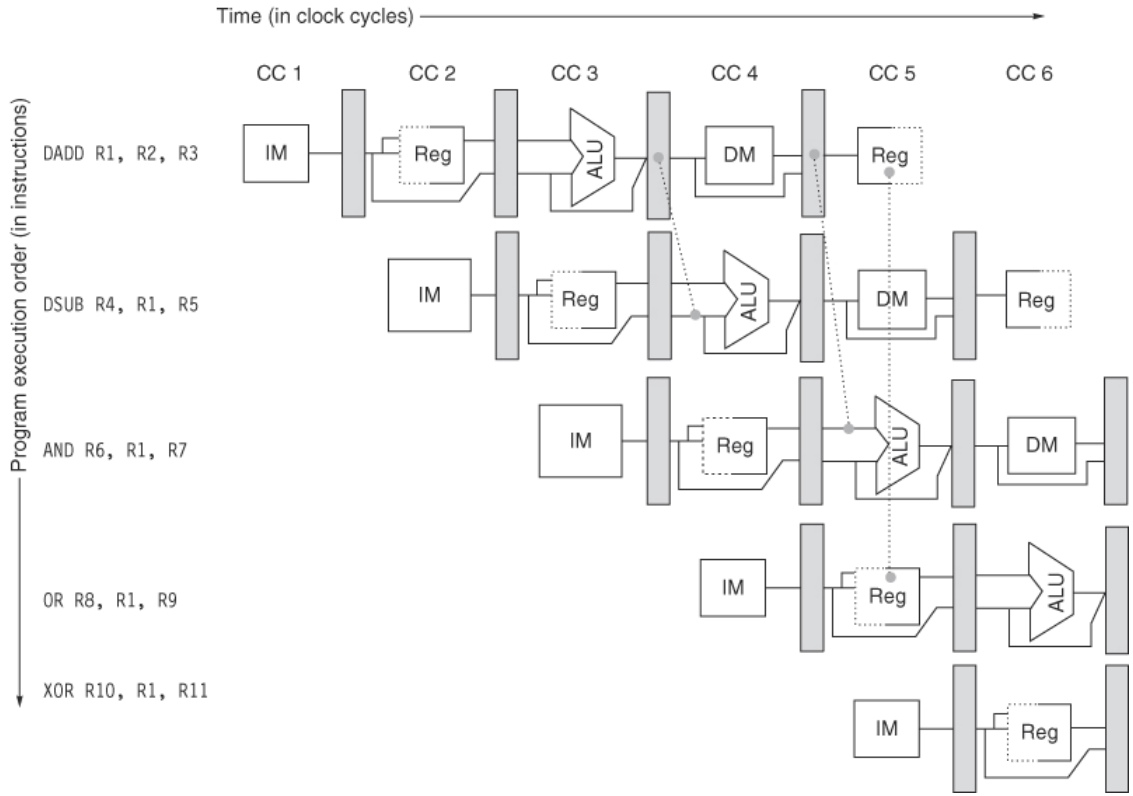

Figure C.7 A set of instructions that depends on the DADD result uses forwarding paths to avoid the data hazard. The inputs for the DSUB and AND instructions forward from the pipeline registers to the first ALU input. The OR receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate. Notice that the forwarded result can go to either ALU input; in fact, both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers. This would occur, for example, if the AND instruction was AND R6,R1,R4.