EEL 5764 Computer Architecture

Sandip Ray
Department of Electrical and Computer Engineering
University of Florida

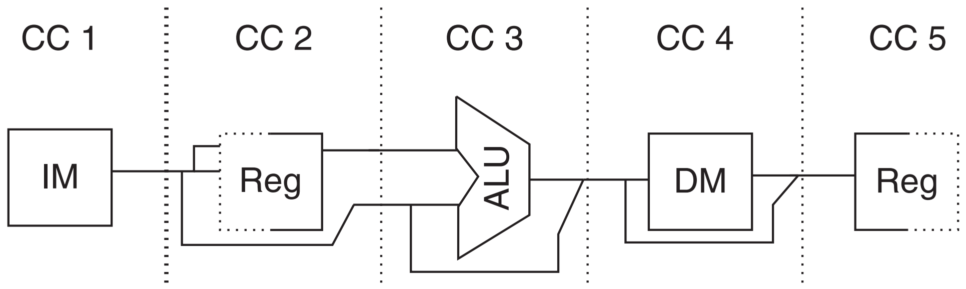
Lecture 18: Pipelining

Announcements

- You should get project feedback by October 9 (Sorry about the delay)
- You should get the graded HW1 back by October 9 (Sorry about the delay)

Instruction Execution of RISC

- Initial State: PC is set to point to the first instruction
- For each instruction, perform the following 5 steps:
  - → Instruction Fetch (**IF**)
  - → Instruction Decode/Register Read (**ID**)
  - → Execution/Effective Address Calculation (**EX**)
  - → Memory Access (**MEM**)
  - → Write Back (**WB**)



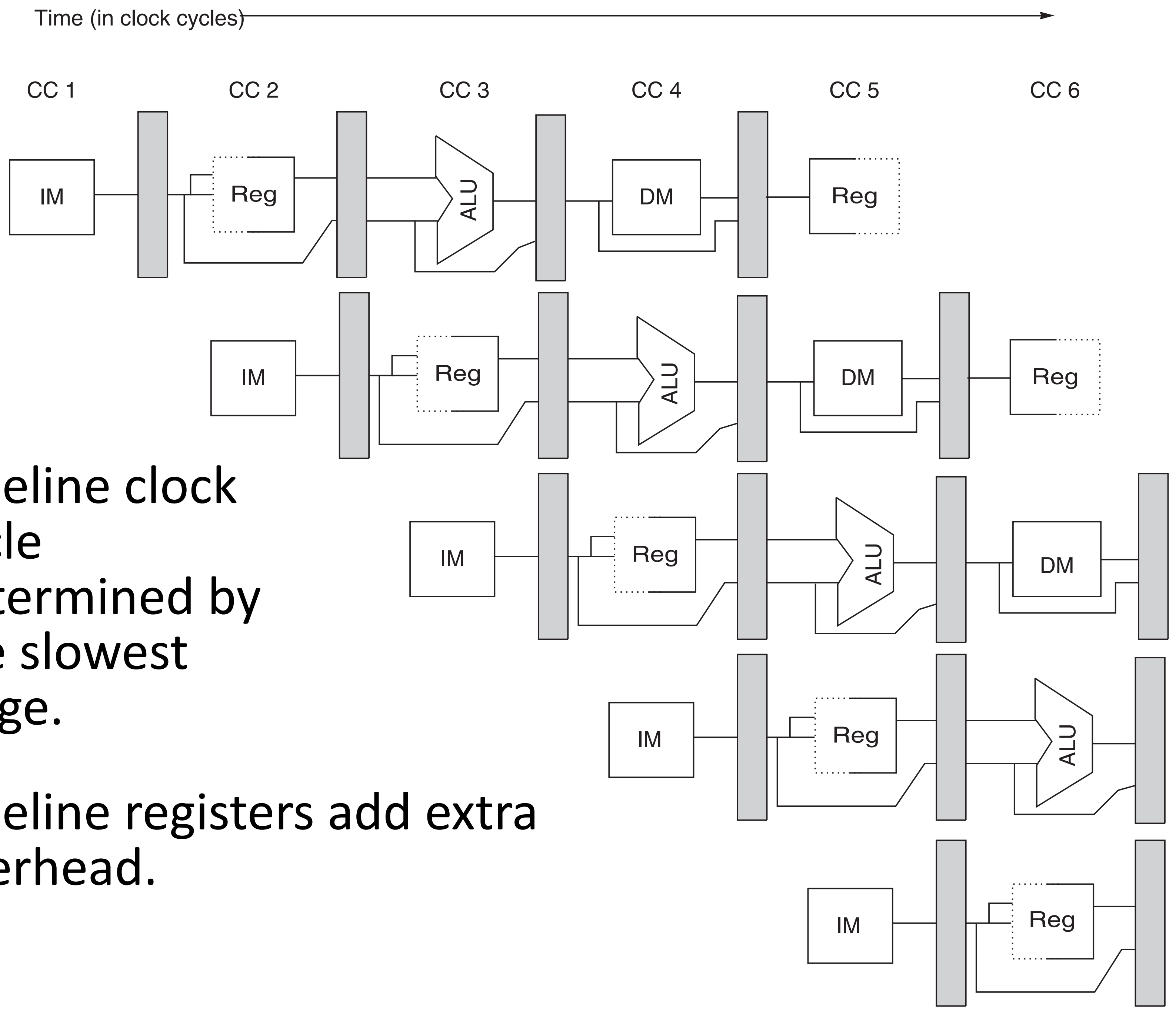
Basic Pipeline

To improve performance, we can make circuit faster, or use …

	Clock number								
Instruction number	**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

Ideal time/instruction =

$$\frac{\text{time/instruction unpipelined}}{\text{\# pipeline stage}}$$



- Pipeline clock cycle determined by the slowest stage.
- Pipeline registers add extra overhead.

Ideal Pipeline and Performance

- Balanced pipeline (each stage has the same delay)
- Zero overhead due to clock skew and pipeline registers
- Ignore pipeline fill and drain overheads

$$Average\ time/instruction = \frac{Average\ time/instruction_{non-pipelined}}{Number\ of\ pipeline\ stages}$$

$$Speedup = \frac{Average\ time/instruction_{non-pipeline}}{Average\ time/instruction_{pipeline}}$$

$$= Number\ of\ pipelinestates$$

Pipeline Performance

- Example: A program consisting of 500 instructions is executed on a 5-stage processor. How many cycles would be required to complete the program. Assume ideal overlap in case of pipelining.
- Without pipelining:
  - → Each instruction will require 5 cycles. There will be no overlap amongst successive instructions.
  - → Number of cycles = 500 * 5 = 2500
- With pipelining:
  - → Each pipeline stage will process a different instruction every cycle. First instruction will complete in 5 cycles, then one instruction will complete in every cycle, due to ideal overlap.
  - → Number of cycles = 5 + ((500-1)*1) = 504
- Speedup for ideal pipelining = 2500/504 = 4.96

Pipeline Performance

- Problem: Consider a non-pipelined processor using the 5-stage datapath with 1 ns clock cycle. Assume that due to clock skew and pipeline registers, pipelining the processor adds 0.2 ns of overhead to the clock speed. How much speedup can we expect to gain from pipelining? Assume a balanced pipeline and ignore the pipeline fill and drain overheads. (A similar ex. in the book)
- Solution:
  - → Without pipelining: Clock period = 1 ns, CPI = 5
  - → With pipelining: Clock period = 1 + 0.2 = 1.2 ns, CPI = 1

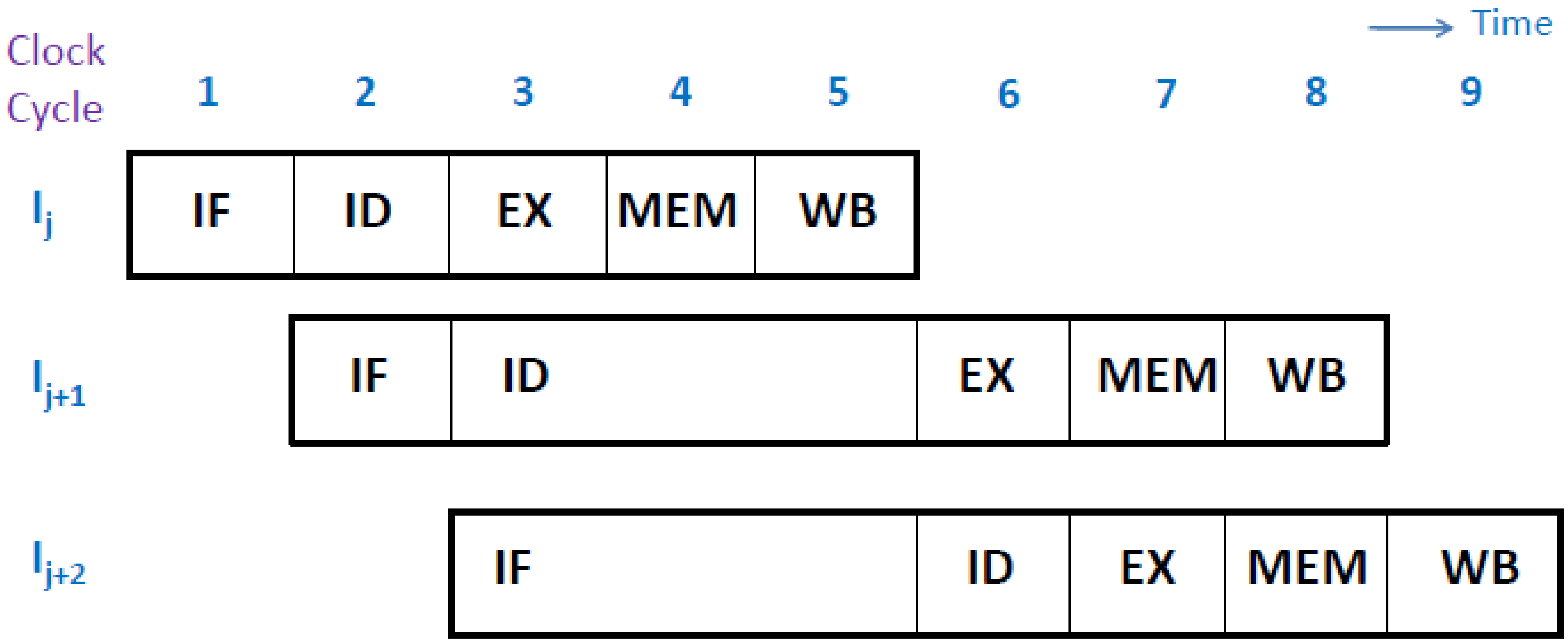
$$\text{Speedup from pipelining} = \frac{(\textit{Average time per instruction})_{non\_pipelined}}{(\textit{Average time per instruction})_{pipelined}}$$

$$= (1\ \text{ns} * 5)\ /\ (1.2\ \text{ns} * 1) = 5\ /\ 1.2 = 4.17$$

Pipeline Performance

- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages
- However, this increase would be achieved only if
  - → all pipeline stages require the same time to complete, and
  - → there is no interruption throughout program execution
- Unfortunately, this is not true
  - → there are times when an instruction cannot proceed from one stage to the next in every clock cycle

Pipeline Performance



- Assume that Instruction I_{j+1} is stalled in the decode stage for two extra cycles
- This will cause I_{j+2} to be stalled in the fetch stage, until I_{j+1} proceeds
- New instructions cannot enter the pipeline until I_{j+2} proceeds past the fetch stage after cycle 5 => execution time increases by two cycles

Pipeline Hazards

Dependences

- Independent instructions can be executed in parallel
- An instruction can depend on another in three ways
  - **Data dependences**
  - **Name dependences**
  - **Control dependences**

Dependences limit how much parallelism can exploited

Pipeline Hazards

- Hazards prevent the next instruction in the instruction stream from executing during its designated clock cycle
- Three types of pipeline hazards
  - Structural hazard – a situation where two (or more) instructions require the use of a given hardware resource at the same time
  - Data hazard – any condition in which either the source or the destination operands of an instruction are not available, when needed in the pipeline
    - Instruction processing will be delayed until operands become available
  - Control hazard – a delay in the availability of an instruction or the memory address needed to fetch the instruction

Pipeline Hazards

- → Hazards may require “stalling” the pipeline allowing some instructions to proceed while others are delayed
  - → With no additional instructions fetched
  - → until the conditions that caused the hazard do not exist anymore
- → This is called “clearing the hazard”
- → Stalling increases the CPI,
  - → reduces the speedup from pipelining

$$Speedup = \frac{Number\ of\ pipeline\ stages}{1 + Stall\ cycles/instruction}$$

Structural Hazards

- A situation where two (or more) instructions require the use of a given hardware resource at the same time
- Example: In the MIPS 5-stage pipeline, both the “IF” and “MEM” stages require memory access
- In the same cycle
  - A new instruction is fetched from memory in the IF stage
  - A “Load” instruction reads data from memory in the MEM stage
- What happens if the memory has only one read port?

Structural Hazards – Stalls

→ Instructions ahead of the stall proceed to completion

→ Stall delays instruction and those following it

	Clock cycle number									
Instruction	**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

Structural Hazards – Cost of Stalls

→ Consider two pipelined processors. Suppose data references represent 40% of the instructions executed and that the ideal CPI of the pipelined processor (no structural hazard) is 1. Assume that the processor with the hazard has a clock rate that is 1.05 times higher than the hazard free processor and incurs a one cycle stall on structural hazards as in our example. Is the pipeline with or without the structural hazard faster, and by how much?

→ Without structural hazard: Clock period = C, CPI = 1

→ With structural hazard: Clock period = C/1.05, CPI = 1 + (0.4 * 1) = 1.4

→ $(Execution\ Time)_{without\ structural\ hazard}/(Execution\ time)_{with\ structural\ hazard} = C * 1/((C/1.05) * 1.4) = 0.75$

→ Processor without the structural hazard is faster

How to Reduce Structural Hazards

- **Key Idea: Add more hardware resources**
- How to avoid structural hazard on memory access during IF and MEM stages?
  - Separate instruction and data caches
- How to avoid structural hazard on register access during ID and WB stages?
  - Dual ported register file
  - Write early in WB stage
  - Read late in ID stage
- Why no structural hazard on ALU during IF (PC ← PC + 4) and MEM stages?
  - PC + 4 in IF stage uses a dedicated adder, not the ALU

Data Hazards

- Any condition in which either the source or the destination operands of an instruction are not available, when needed in the pipeline
- Example

  Add R2, R3, #100 ; R2 <- [R3] + 100

  Subtract R9, R2, #30 ; R9 <- [R2] + 30

  - First instruction writes to register R2 in the WB stage
  - Second instruction reads register R2 in the ID stage
  - If second instruction reads R2 before the first instruction writes R2, the result of second instruction would be incorrect, as it would be based on R2's old value (read-after-write dependence)
  - To obtain the correct result, second instruction needs to wait until the first instruction has written to R2

Data Hazards

$\longrightarrow$ Time

Clock Cycle	1	2	3	4	5	6	7	8
Add R2, R3, #100	IF	ID	EX	MEM	WB			
Subtract R9, R2, #30		IF	ID	stall	stall	EX	MEM	WB

- Subtract is stalled in decode stage for two additional cycles to delay reading R2 until the new value of R2 has been written by Add
  - How is this pipeline stall implemented in hardware?
- Control circuitry recognizes the data dependency when it decodes Subtract (comparing source register ids with dest. register ids of prior instructions)
- During cycles 3 to 5:
  - Add proceeds through the pipe
  - Subtract is held in the ID stage
- In cycle 5
  - **Add** writes R2 in the first half cycle, **Subtract** reads R2 in the second half cycle

Types of Data Hazards

- → Read After Write (RAW) True dependence
  - → Caused by “dependence”. Subsequent instruction has actual need for data produced by earlier instruction
- → Write after Read (WAR) False dependence
  - → Called “anti-dependence”. Can’t occur in in-order pipelines (e.g., our simple MIPS pipeline). We’ll see it later in advanced pipelines
- → Write after Write (WAW) False dependence
  - → Called “output dependence”. Can’t occur in in-order pipelines (e.g., our simple MIPS pipeline). We’ll see it later in advanced pipelines

Types of Data Hazards - RAW

→ Inst_j tries to read operand before Inst_i write it (j > i)

Instr_i: **add r1, r2, r3**

Instr_j: **sub r4, r1, r3**

→ This hazard results from a true dependence: an actual need for communication from the first instruction to the second

Types of Data Hazards - WAR

- Inst$_j$ tries to write operand before Inst$_i$ reads it (j > i)

$$\begin{tikzcd} \text{Instr}_i\text{: }\textbf{sub r4, r1, r3} \arrow[d] \\ \text{Instr}_j\text{: }\textbf{add r1, r2, r3} \end{tikzcd}$$

- This is an anti-dependence (not a true dependence)
  - Arises from the reuse of register “r1”
  - If Instr$_j$ writes to r1 before Instr$_i$ reads r1, then Instr$_i$ will use the value written by Instr$_j$ => Incorrect behavior
- WAR hazards cannot happen in MIPS 5-stage pipeline:
  - Instructions are executed in order
  - Register reads happen earlier (stage-2) than register writes (stage-5)

Types of Data Hazards - WAW

- Instr$_j$ writes to operand before Instr$_i$ writes to it ($j > i$)

  Instr$_i$: **sub r1, r4, r3**

  $\downarrow$

  Instr$_j$: **add r1, r2, r3**

  Instr$_k$: **mul r6, r1, r7**
- This is an output dependence (not a true dependence)
  - Arises from the reuse of register r1
  - If Instr$_j$ writes to r1 before Instr$_i$ then Instr$_k$ will use the value written by Instr$_i$ => violation of program order
- WAW hazards cannot happen in MIPS 5-stage pipeline because:
  - Instructions are executed in order
  - Register writes are always in stage-5

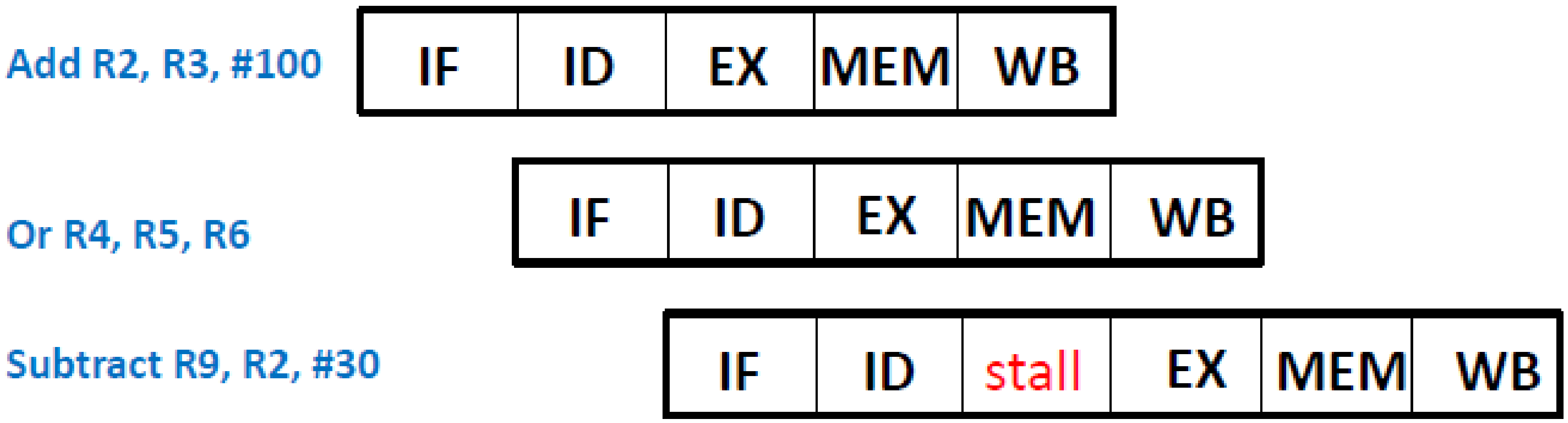
Data Hazards – Stall Penalty

$\longrightarrow$ Time

Clock Cycle	1	2	3	4	5	6	7	8
Add R2, R3, #100	IF	ID	EX	MEM	WB			
Subtract R9, R2, #30		IF	ID	stall	stall	EX	MEM	WB

→ If the dependent instruction follows right after the producer instruction, we need to stall the pipe for 2 cycles (Stall penalty = 2)

Data Hazards – Stall Penalty



→ What happens if the producer and dependent instructions are separated by an intermediate instruction?

→ In this case, stall penalty = 1

→ Stall penalty depends upon the **distance** between the producer and dependent instruction