

# Pipeline

---

**Thoai Nam**



# Outline

---

- ❑ Pipelining concepts
- ❑ The DLX architecture
- ❑ A simple DLX pipeline
- ❑ Pipeline Hazards and Solution to overcome

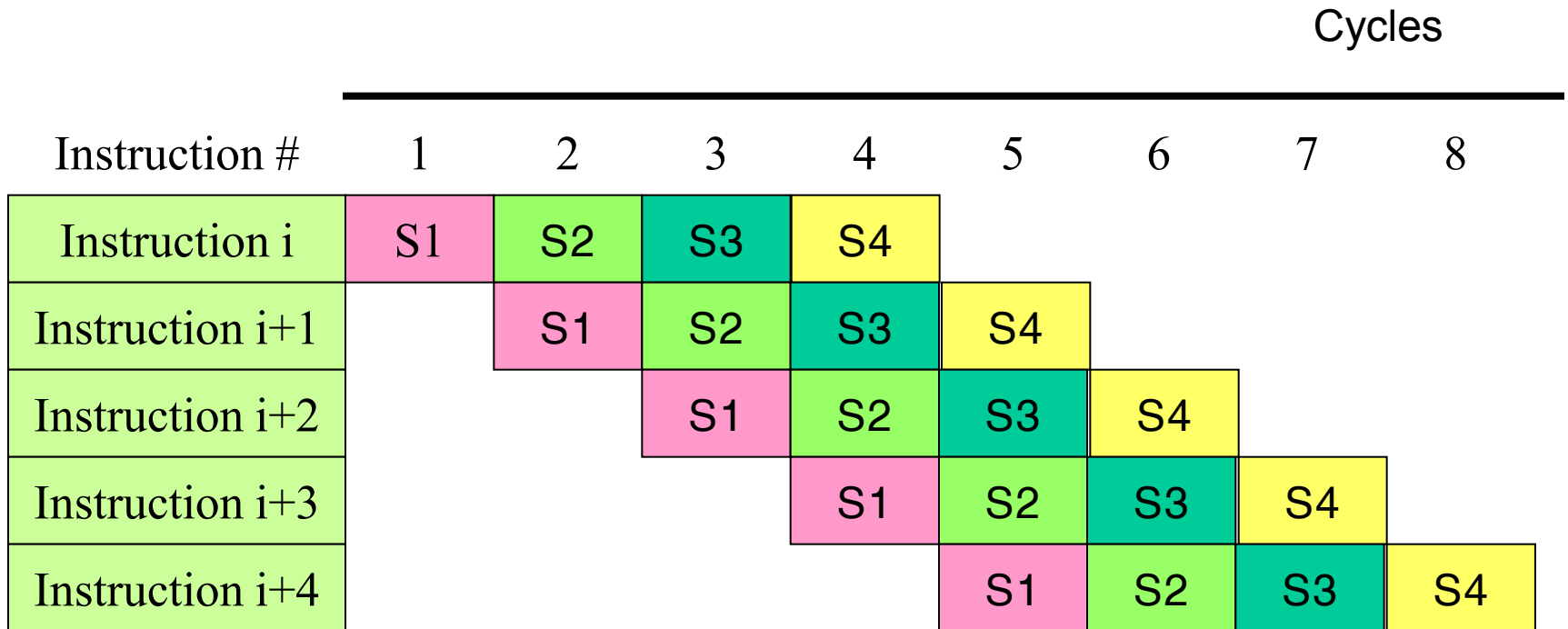
## Reference:

Computer Architecture: A Quantitative Approach,  
*John L Hennessy & David a Patterson*, Chapter 6



# Concepts

- A technique to make fast CPUs by overlapping execution of multiple instructions





# Concepts (cont'd)

---

- ❑ Pipeline throughput
  - Determined by how often an instruction exists the pipeline
  - Depends on the overhead of clock skew and setup
  - Depends on the time required for the slowest pipe stage
- ❑ Pipeline stall
  - Delay the execution of some instructions and all succeeding instructions
  - “Slow down” the pipeline
- ❑ Pipeline Designer’s goal
  - Balance the length of pipeline stages
  - Reduce / Avoid pipeline stalls



# Concepts (cont'd)

---

$$\begin{aligned} \text{Pipeline speedup} &= \frac{\text{Average instruction time without pipeline}}{\text{Average instruction time with pipeline}} \\ &= \frac{\text{CPI without pipelining} * \text{Clock cycle without pipelining}}{\text{CPI with pipelining} * \text{Clock cycle with pipelining}} \end{aligned}$$

( CPI = number of Cycles Per Instruction)

CPI without pipelining = Ideal CPI \* Pipeline depth

CPI with pipelining = Ideal CPI + Pipeline stall clock cycles per instruction

$$\text{Pipeline speedup} = \frac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}}$$



# The DLX Architecture

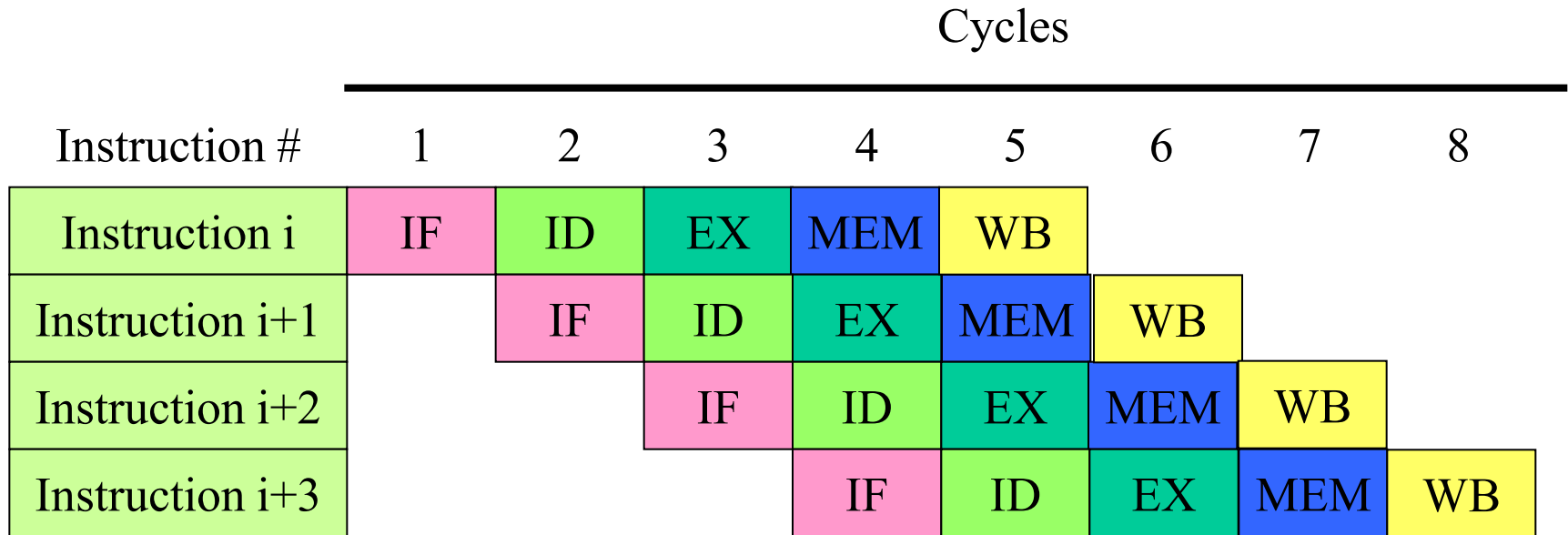
---

- ❑ A mythical computer which architecture is based on most frequently used primitives in programs
- ❑ Used to demonstrate and study computer architecture organizations and techniques
- ❑ A DLX instruction consists of 5 execution stages
  - **IF** – instruction fetch
  - **ID** – instruction decode and register fetch
  - **EX** – execution and effective address calculation
  - **MEM** – memory access
  - **WB** – write back



# A Simple DLX Pipeline

- Fetch a new instruction on each clock cycle
- An instruction step = a pipe stage





# Pipeline Hazards

---

- ❑ Are situations that prevent the next instruction in the instruction stream from executing during its designated cycles
- ❑ Leads to pipeline stalls
- ❑ Reduce pipeline performance
- ❑ Are classified into 3 types
  - Structural hazards
  - Data hazards
  - Control hazards





# Structure Hazard

---

- ❑ Due to resource conflicts
- ❑ Instances of structural hazards
  - Some functional unit is not fully pipelined
    - » a sequence of instructions that all use that unit cannot be sequentially initiated
  - Some resource has not been duplicated enough. Eg:
    - » Has only 1 register-file write port while needing 2 write in a cycle
    - » Using a single memory pipeline for data and instruction
- ❑ Why we allow this type of hazards?
  - To reduce cost.
  - To reduce the latency of the unit



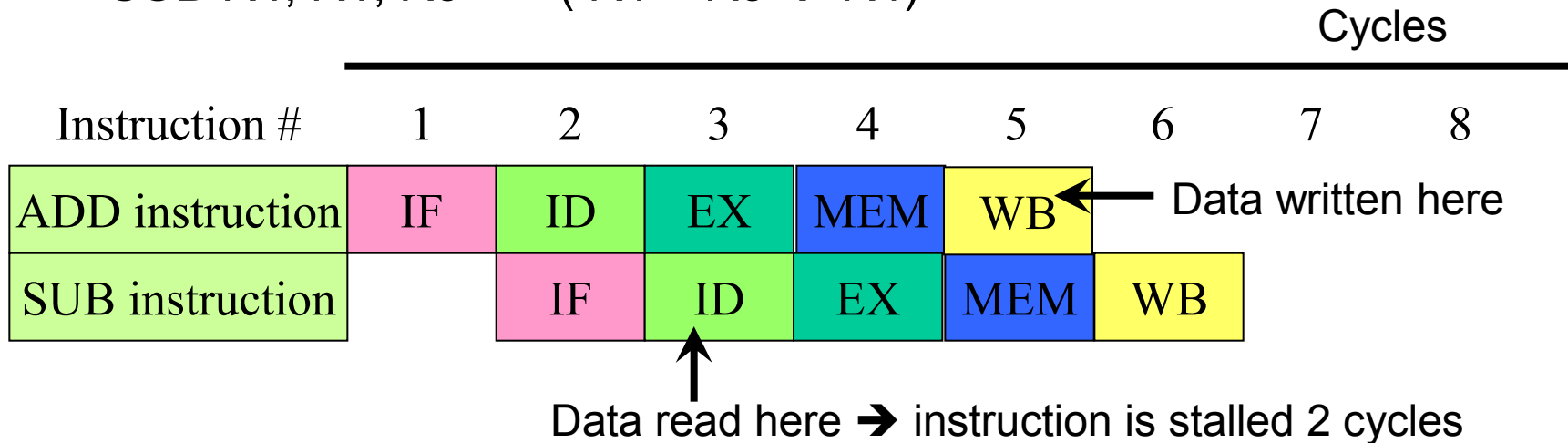
# Data Hazard

- Occurs when the **order of access to operands is changed** by the pipeline, making **data unavailable for next instruction**

- Example: consider these 2 instructions

ADD R1, R2, R3    ( $R2 + R3 \rightarrow R1$ )

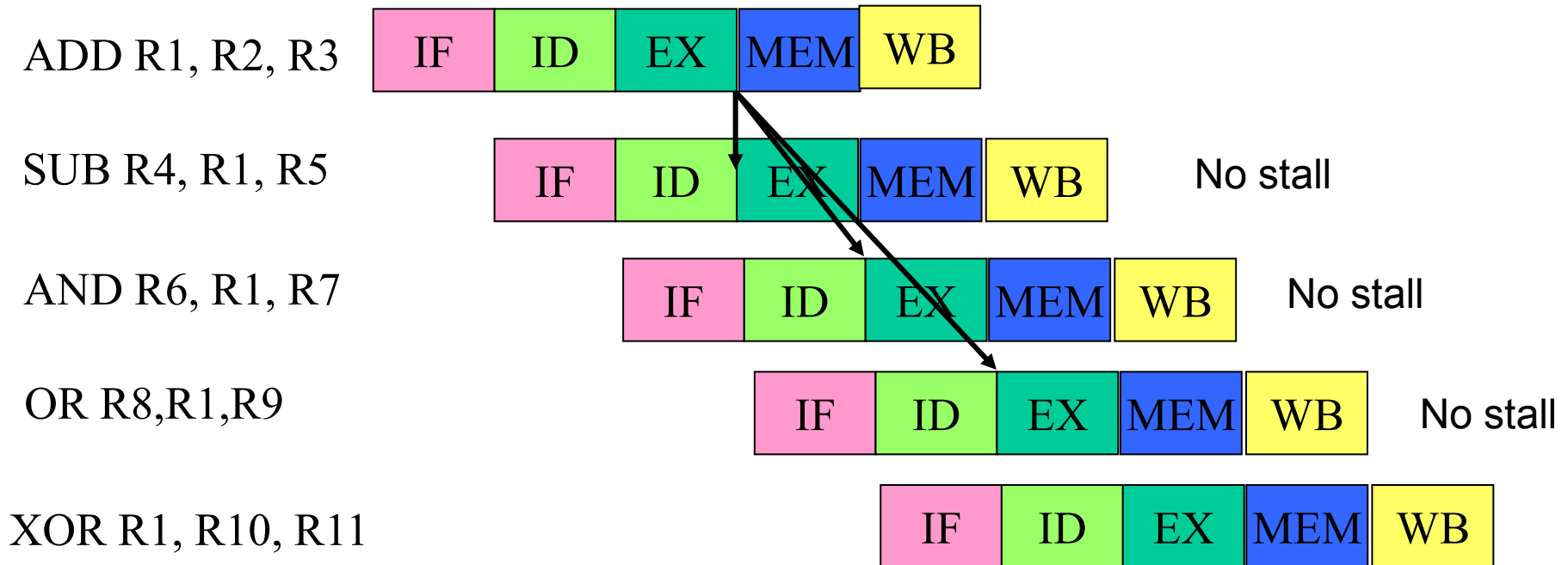
SUB R4, R1, R5    ( $R1 - R5 \rightarrow R4$ )





# Hardware Solution to Data Hazard

- Forwarding (bypassing/short-circuiting) techniques
  - Reduce the delay time between 2 depended instructions
  - The ALU result is fed back to the ALU input latches
  - Forwarding hardware check and forward the necessary result to the ALU input for the **2 next** instructions





# Types of Data Hazards

---

- ❑ RAW(Read After Write)
  - Instruction j tries to read a source before instruction i writes it
  - Most common types
- ❑ WAR(Write After Read)
  - Instruction j tries to write a destination before instruction i read it to execute
  - Can not happen in DLX pipeline. Why?
- ❑ WAW(Write After Write)
  - Instruction j tries to write a operand before instruction i updates it
  - The writes end up in the wrong order
- ❑ Is RAR (Read After Read) a hazard?

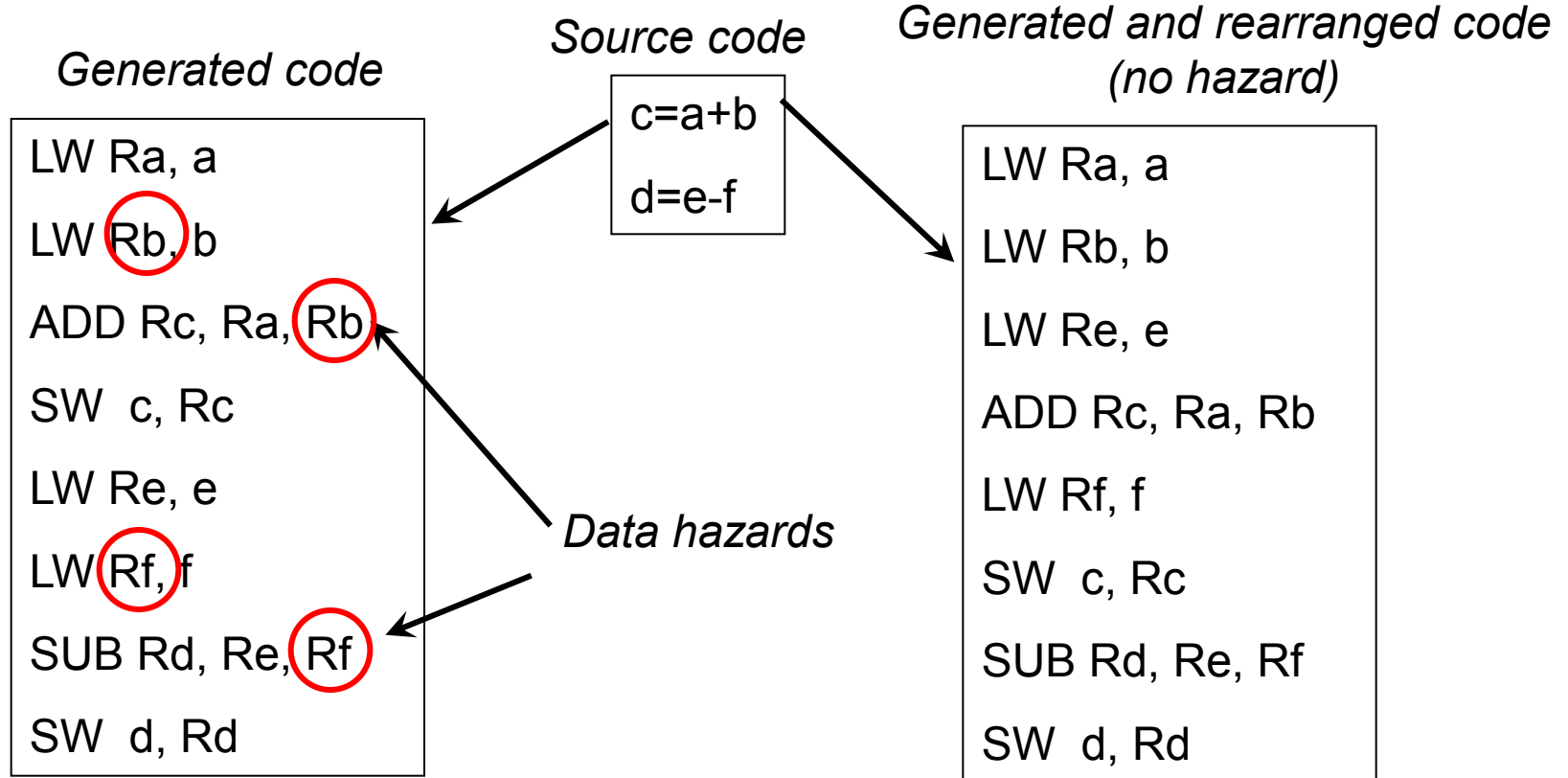


# Software Solution to Data Hazard

## □ Pipeline scheduling (Instruction scheduling)

- Use compiler to rearrange the generated code to eliminate hazard.

Example:





# Control/Branch Hazard

- ❑ Occurs when a branch/jump instruction is taken
- ❑ Causes great performance loss
- ❑ Example:

The PC register changed here

Unnecessary instruction loaded

Branch instruction	IF	ID	EX	MEM	WB					
Instruction i+1	IF	stall	stall		IF	ID	EX	MEM	WB	
Instruction i+2		stall	stall	stall	IF	ID	EX	MEM	WB	
Instruction i+3			stall	stall	stall	IF	ID	EX	MEM..	
Instruction i+4				stall	stall	stall	IF	ID	EX...	
Instruction i+5					stall	stall	stall	IF	ID	
Instruction i+6						stall	stall	stall	IF	



# Reducing Control Hazard Effects

---

- Predict whether the branch is taken or not
- Compute the branch target address earlier
- Use many schemes
  - Pipeline freezing
  - Predict-not-taken scheme
  - Predict-taken scheme (N/A in DLX)
  - Delayed branch



# Pipeline Freezing

---

- ❑ Hold any instruction after the branch until the branch destination is known
- ❑ Simple but not efficient





# Predict-Not-Taken Scheme

---

- Predict the branch as not taken and allow execution to continue
  - Must not change the machine state till the branch outcome is known
- If the branch is not taken: no penalty
- If the branch is taken:
  - Restart the fetch at the branch target
  - Stall one cycle



# Predict-Not-Taken Scheme (cont'd)

## □ Example

Instruction Fetch restarted

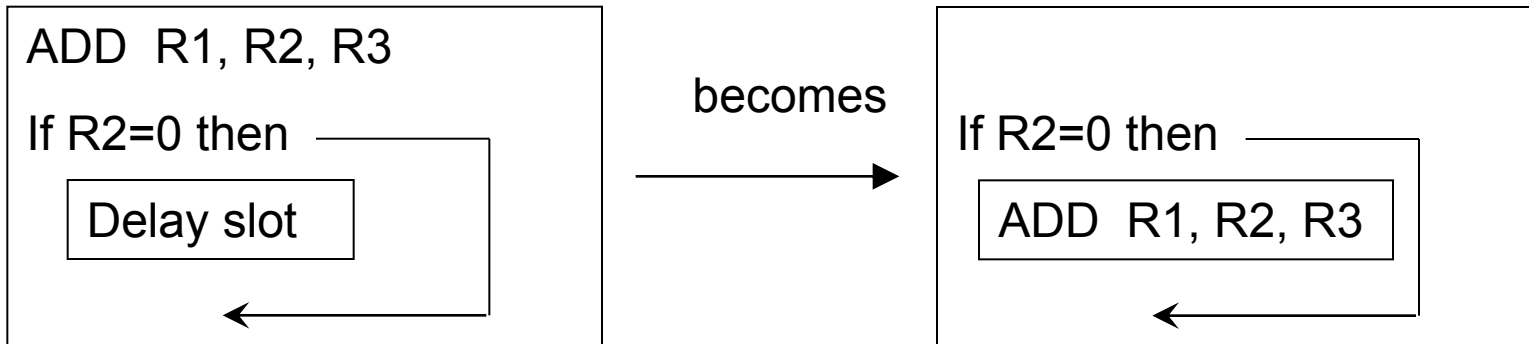
Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction i+1		IF	IF	ID	EX	MEM	WB		
Instruction i+2			stall	IF	ID	EX	MEM	WB	
Instruction i+3				stall	IF	ID	EX	MEM	WB
Instruction i+4					stall	IF	ID	EX	MEM

Right instruction fetched



# Branch Delayed

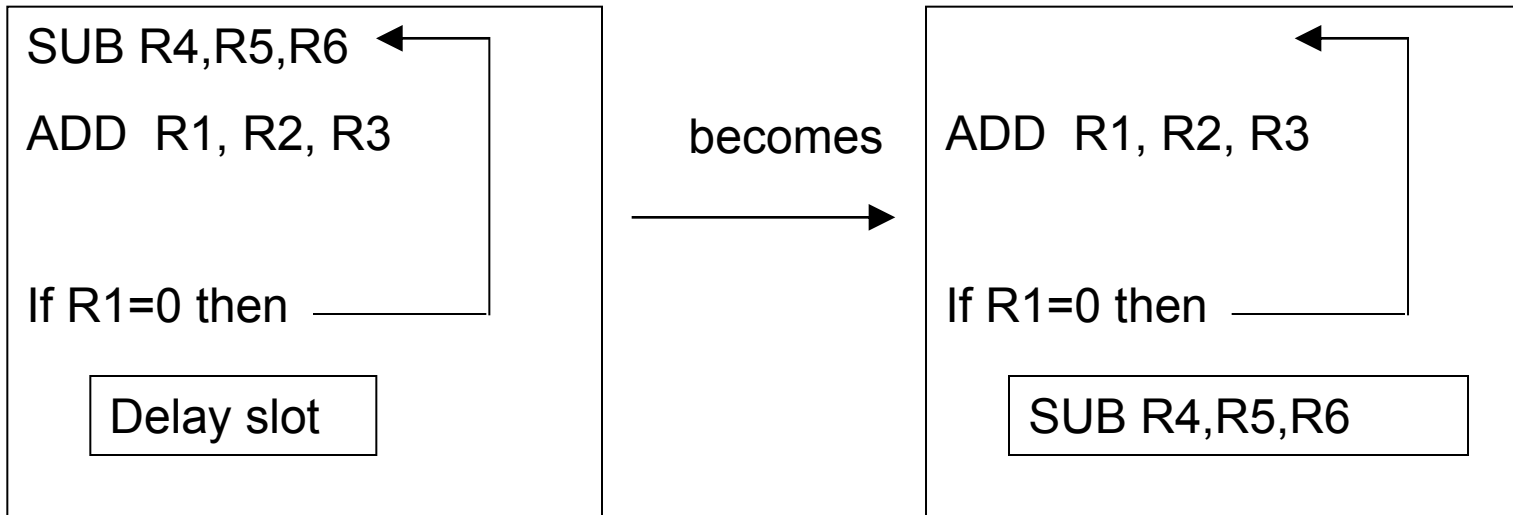
- ❑ Change the order of execution so that the next instruction is always valid and useful
- ❑ “From before” approach





# Branch Delayed (cont'd)

- “From target” approach





# Branch Delayed (cont'd)

- “From fall through” approach

