Computer Architecture

Hossein Asadi
Department of Computer Engineering
Sharif University of Technology
asadi@sharif.edu

Today's Topics

- Pipelining
- Pipelining Hazards

Copyright Notice

- Parts (text & figures) of this lecture adopted from:
  - Computer Organization & Design, The Hardware/Software Interface, 3rd Edition, by D. Patterson and J. Hennessey, MK publishing, 2005.
  - "Intro to Computer Architecture" handouts, by Prof. Hoe, CMU, Spring 2009.
  - "Computer Architecture & Engineering" handouts, by Prof. Kubiatowicz, UC Berkeley, Spring 2004.
  - "Intro to Computer Architecture" handouts, by Prof. Hoe, UWisc, Spring 2019.
  - "Computer Arch I" handouts, by Prof. Garzarán, UIUC, Spring 2009.
  - "Intro to Computer Organization" handouts, by Prof. Mahlke & Prof. Narayanasamy, Winter 2008.

Instruction Latencies and Throughput

•Single-Cycle CPU

Load	IF	ID/RR	Exec	Mem	Wr

•Multiple Cycle CPU

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5

Load	IF	ID/RR	Exec	Mem	Wr

•Pipelined CPU

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8

Load	IF	ID/RR	Exec	Mem	Wr
Load	Ifetch	ID/RR	Exec	Mem	Wr
Load	Ifetch	ID/RR	Exec	Mem	Wr
Load	Ifetch	ID/RR	Exec	Mem	Wr
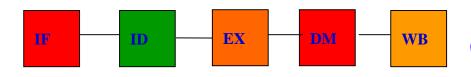
Introduction of Pipeline

Machine assembly line
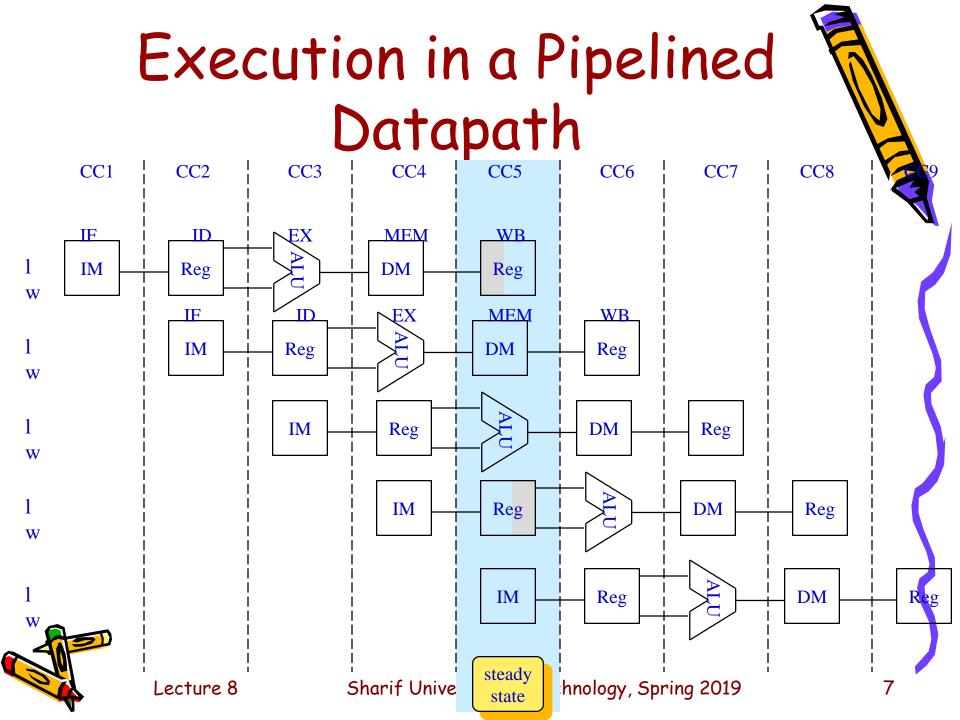
Datapath of pipeline

IF	ID	EX	DM	WB

Execution in a Pipelined Datapath

CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9

Execution in a Pipelined Datapath

lw   IF [IM]   ID [Reg]   EX [ALU]   MEM [DM]   WB [Reg]

lw   IF [IM]   ID [Reg]   EX [ALU]   MEM [DM]   WB [Reg]

lw   IF [IM]   ID [Reg]   EX [ALU]   MEM [DM]   WB [Reg]

lw   IF [IM]   ID [Reg]   EX [ALU]   MEM [DM]   WB [Reg]

lw   IF [IM]   ID [Reg]   EX [ALU]   MEM [DM]   WB [Reg]

steady state

Mixed Instructions in Pipeline

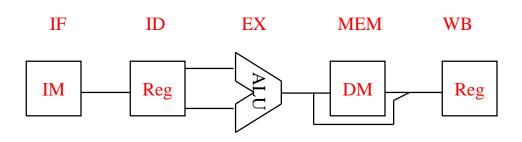|	CC1	CC2	CC3	CC4	CC5	CC6

l
w

IM — Reg — ALU — DM — Reg

add

IM — Reg — ALU — Reg
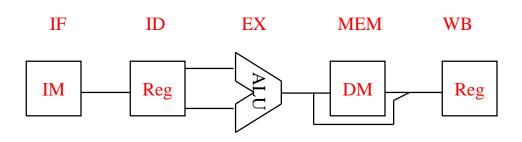
Pipeline Principles

- Principles
  - All instructions that share a pipeline must have same *stages* in same *order*
    - ➔ *add* does nothing during Mem stage
    - ➔ *sw* does nothing during WB stage

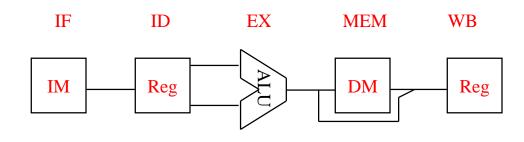| IF	ID	EX	MEM	WB
IM	Reg	ALU	DM	Reg

Pipeline Principles (cont.)

- Principles
  - All intermediate values must be latched each cycle
  - No functional block reuse for an instruction
    - E.g. need 2 adders and ALU (like in single-cycle)

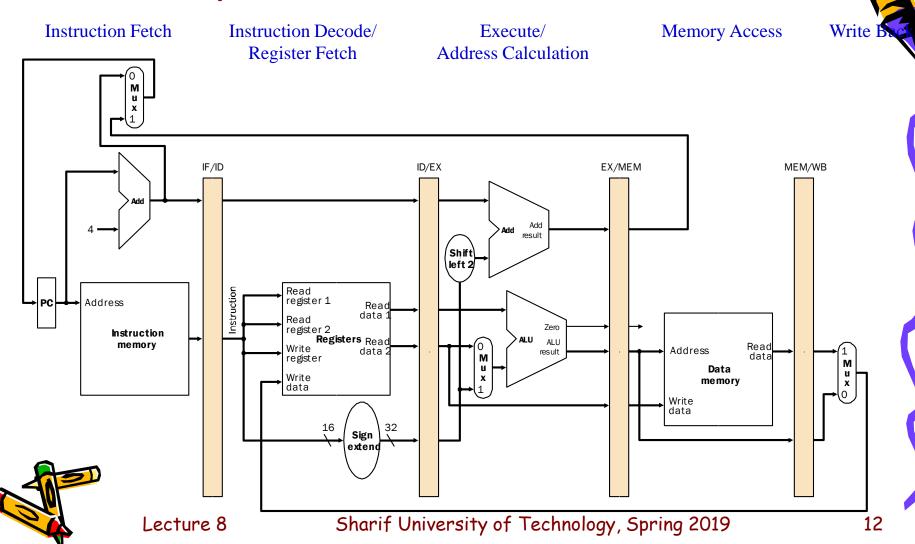IF   ID   EX   MEM   WB

| IM | Reg | ALU | DM | Reg |
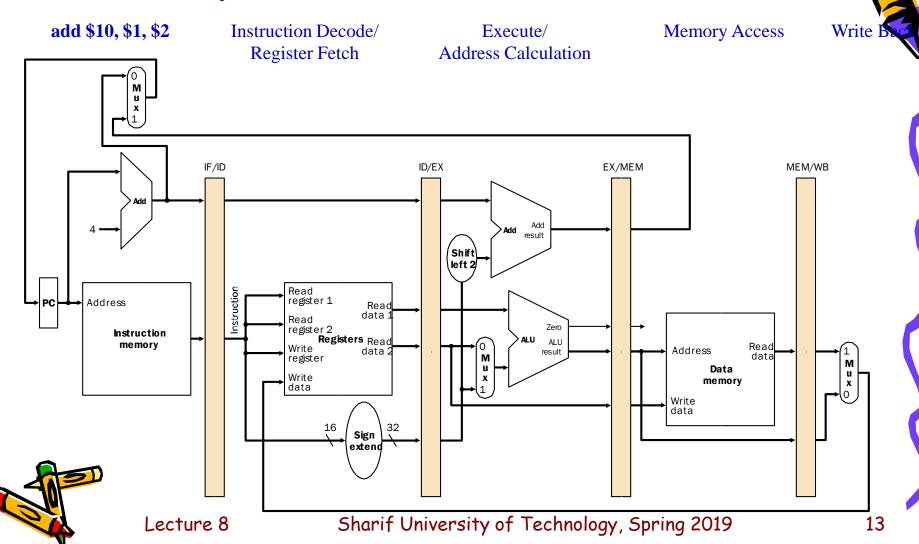
Pipeline Principles
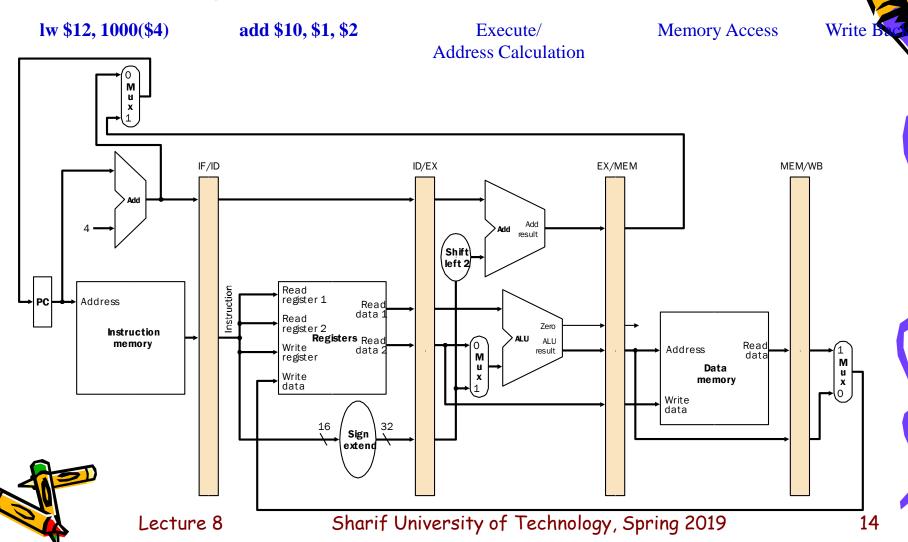
- Question:
  - What if we wanted to bypass a stage for an instruction?
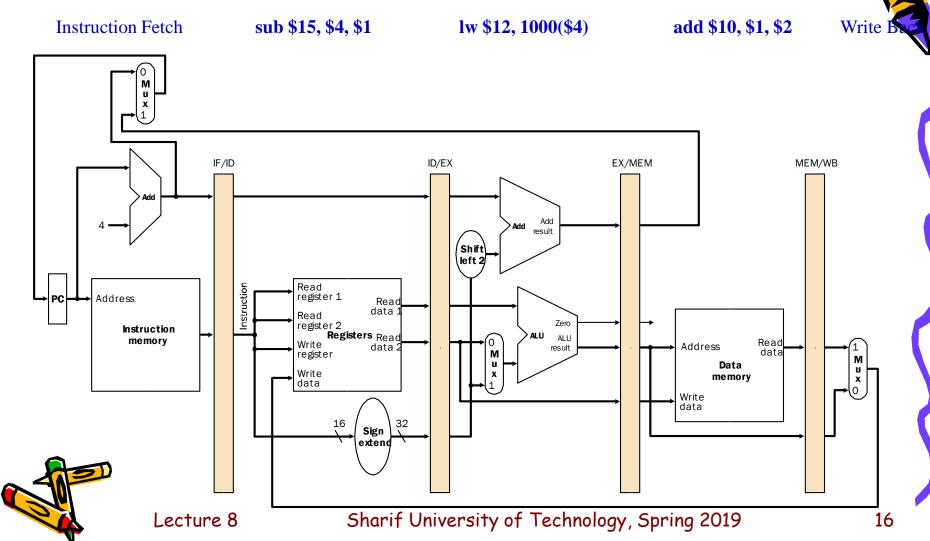
IF	ID	EX	MEM	WB
IM	Reg	ALU	DM	Reg

Pipeline in Execution

Instruction Fetch — Instruction Decode/Register Fetch — Execute/Address Calculation — Memory Access — Write Back

Pipeline in Execution

add $10, $1, $2   Instruction Decode/   Execute/   Memory Access   Write Back
    Register Fetch   Address Calculation

Pipeline in Execution

lw $12, 1000($4)   **add $10, $1, $2**   Execute/
                                                                    Address Calculation   Memory Access   Write Back

Pipeline in Execution

sub $15, $4, $1   **lw $12, 1000($4)**   **add $10, $1, $2**   Memory Access   Write Back

Pipeline in Execution

Instruction Fetch   **sub $15, $4, $1**   **lw $12, 1000($4)**   **add $10, $1, $2**   Write Back

Pipelining Performance

- ET = IC * CPI * CT
- Achieve High *throughput*
  - Without reducing instruction *latency*
- Example:
  - A CPU that takes 5ns to execute an instruction pipelined into 5 equal stages
    - Latch between each stage has a delay of 0.25 ns
  1. Min. clock cycle time of this arch?
  2. Max. speedup that can be achieved by this arch (compared to single cycle arch)?

Issues With Pipelining

- Pipelining Creates Potential Hazards

  - What happens if two instructions need a same structure?
    - Structure hazard
  - What happens when an instruction needs result of another instruction?
    - Data hazard
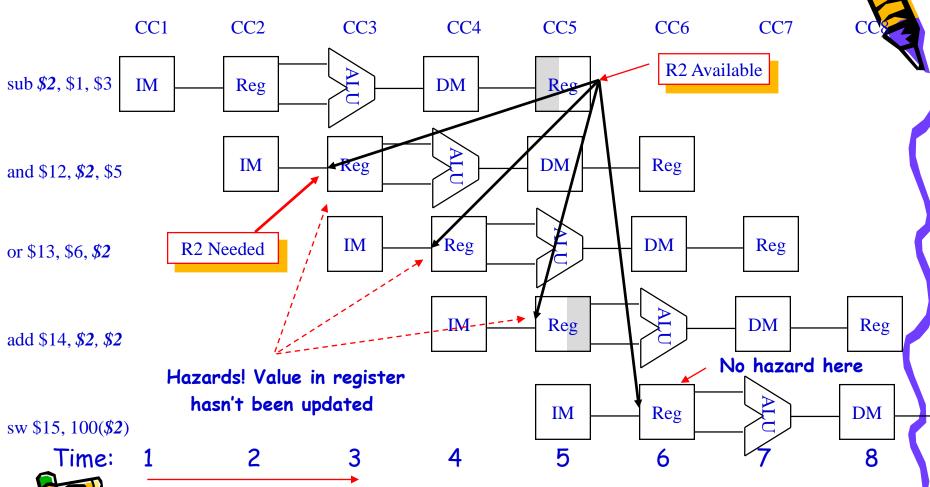 - What happens on a branch?
    - Control hazard

Structural Hazards

- How?
  - Two instructions require use of a given hardware resource at same time

- Access to Memory
  - Separate instruction and data caches

- Access to Register File
  - Multiple port register file

Data Hazards

CC1   CC2   CC3   CC4   CC5   CC6   CC7   CC8

sub **$2**, $1, $3 — IM — Reg — ALU — DM — Reg

R2 Available

and $12, **$2**, $5 — IM — Reg — ALU — DM — Reg

R2 Needed

or $13, $6, **$2** — IM — Reg — ALU — DM — Reg

add $14, **$2**, **$2** — IM — Reg — ALU — DM — Reg

Hazards! Value in register hasn't been updated

No hazard here

sw $15, 100(**$2**) — IM — Reg — ALU — DM

Time: 1   2   3   4   5   6   7   8

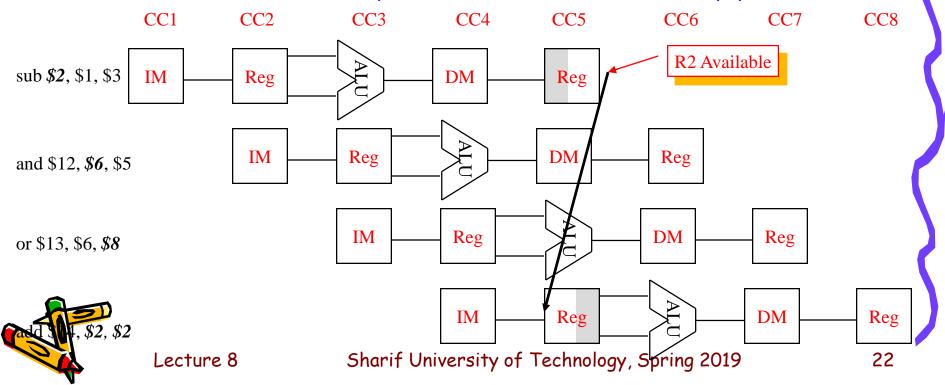Data Hazard: result needed before it is available

Handling Data Hazards

- ## SW Solution

  - Insert independent instructions

  - No-ops instructions

- ## HW Solutions

  - Insert bubbles (i.e. stall pipeline)

  - Data forwarding

  - Latch-based GPR

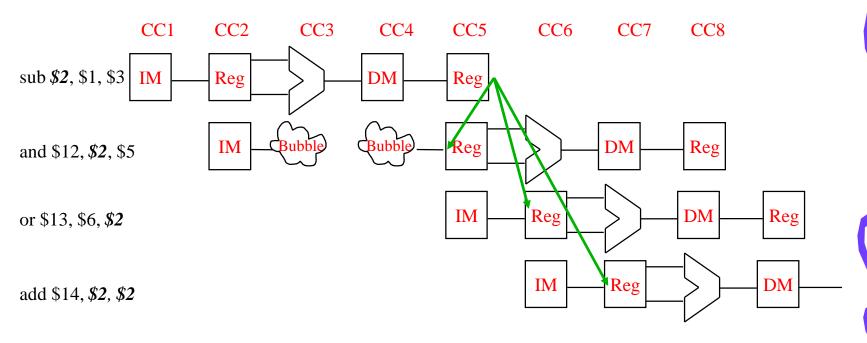# Dealing With Data Hazards

- Using Transparent Register File
  - Use latches rather than flip-flops in RF
    - First half-cycle : register loaded
    - Second half-cycle: new value is read into pipeline state

CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8

sub $2, $1, $3

and $12, $6, $5

or $13, $6, $8

add $14, $2, $2

R2 Available
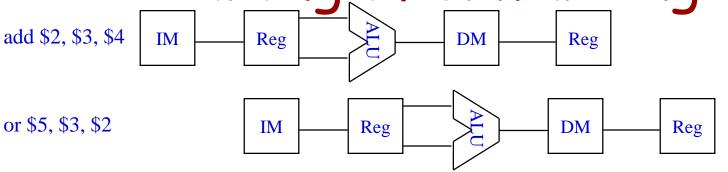
Handling Data Hazards in Hardware: Stall pipeline

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8

sub **$2**, $1, $3 — IM — Reg — [ALU] — DM — Reg

and $12, **$2**, $5 — IM — Bubble — Bubble — Reg — [ALU] — DM — Reg

or $13, $6, **$2** — IM — Reg — [ALU] — DM — Reg

add $14, **$2**, **$2** — IM — Reg — [ALU] — DM
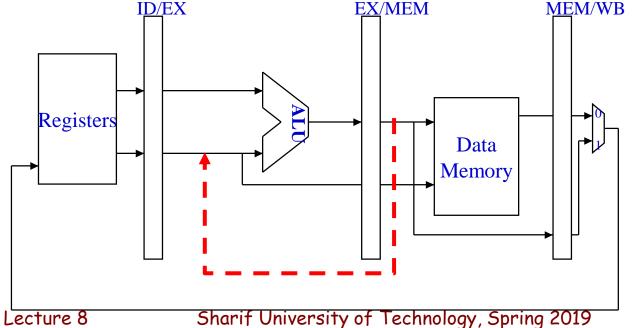
Reducing Data Hazards Through Forwarding

add $2, $3, $4

| IM | | Reg | | ALU | | DM | | Reg |

or $5, $3, $2

| | IM | | Reg | | ALU | | DM | | Reg |
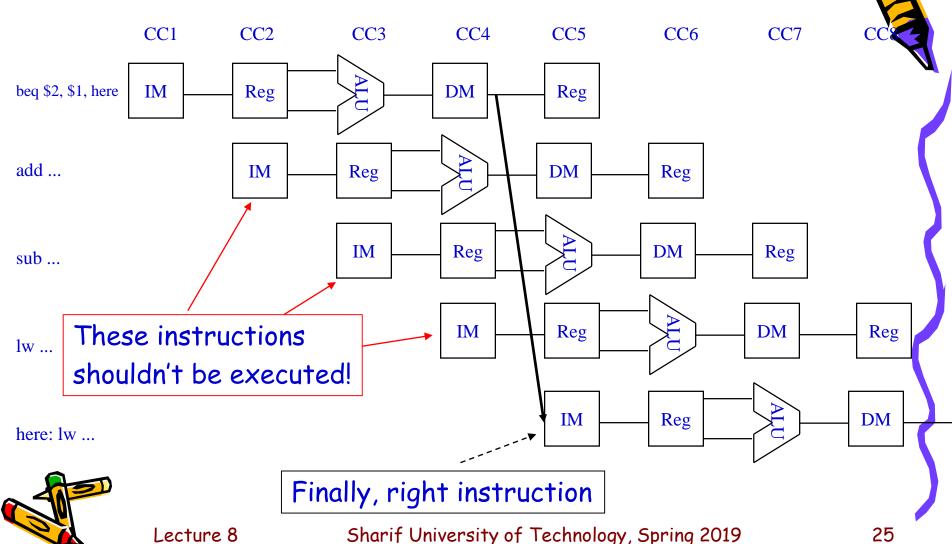
Avoid stalling by forwarding ALU output from "add" to ALU input for "or"

ID/EX   EX/MEM   MEM/WB

Registers → ALU → Data Memory
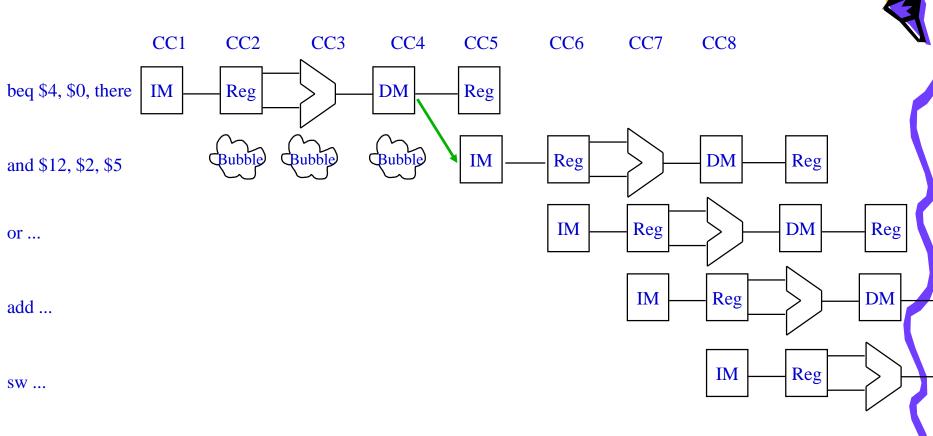
Control or Branch Hazard

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8

beq $2, $1, here — IM — Reg — ALU — DM — Reg

add ... — IM — Reg — ALU — DM — Reg

sub ... — IM — Reg — ALU — DM — Reg

lw ... — IM — Reg — ALU — DM — Reg

here: lw ... — IM — Reg — ALU — DM

These instructions shouldn't be executed!

Finally, right instruction

Stalling for Branch Hazards

CC1   CC2   CC3   CC4   CC5   CC6   CC7   CC8

beq $4, $0, there | IM | | Reg | > | DM | Reg |

and $12, $2, $5   Bubble Bubble Bubble | IM | Reg | > | DM | Reg |

or ... | IM | Reg | > | DM | Reg |

add ... | IM | Reg | > | DM |

sw ... | IM | Reg | > |
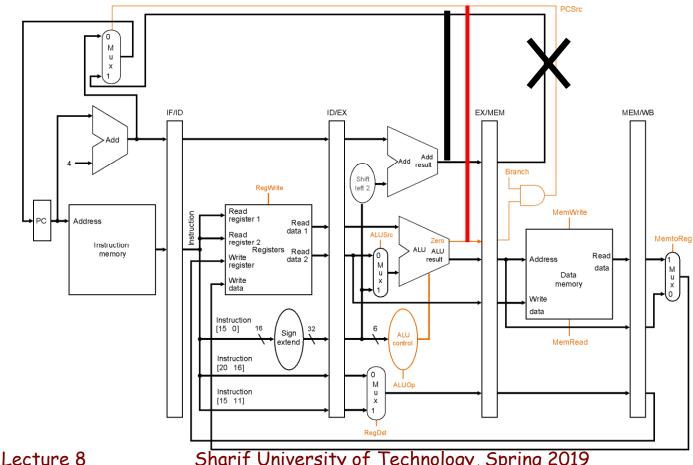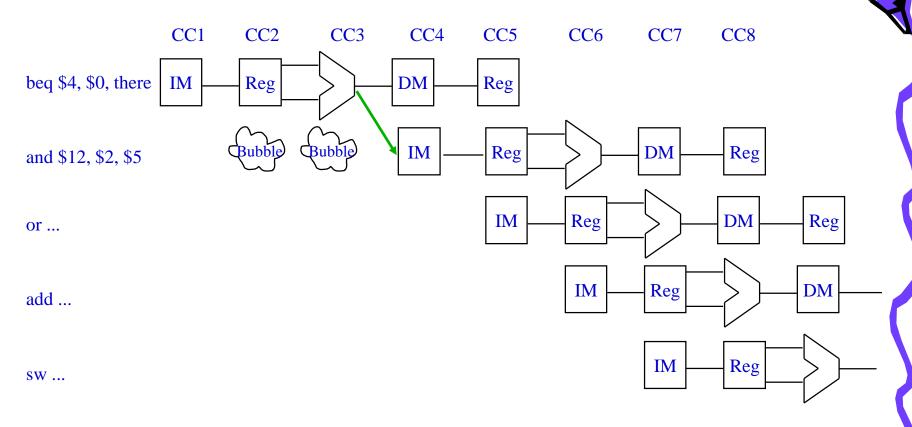
Reducing Branch Delay

It's easy to reduce stall to 2-cycles

Stalling for Branch Hazards

CC1   CC2   CC3   CC4   CC5   CC6   CC7   CC8

beq $4, $0, there | IM | Reg | > | DM | Reg |

Bubble   Bubble

and $12, $2, $5 | IM | Reg | > | DM | Reg |

or ... | IM | Reg | > | DM | Reg |

add ... | IM | Reg | > | DM |

sw ... | IM | Reg | > |
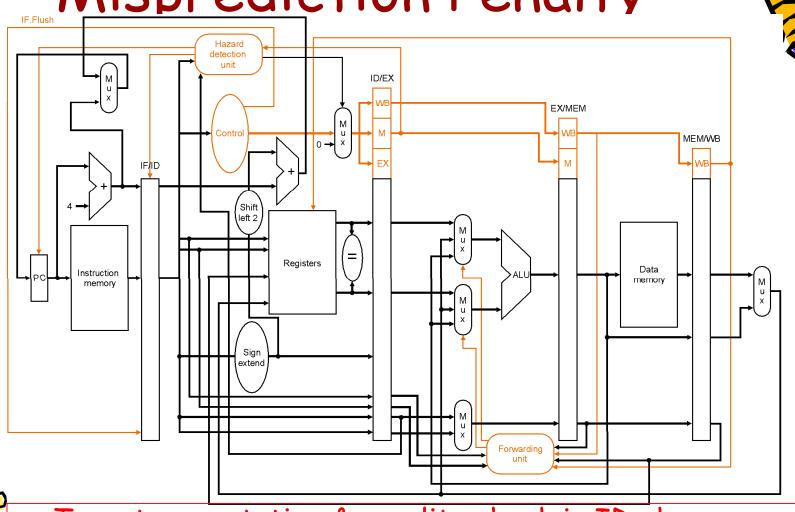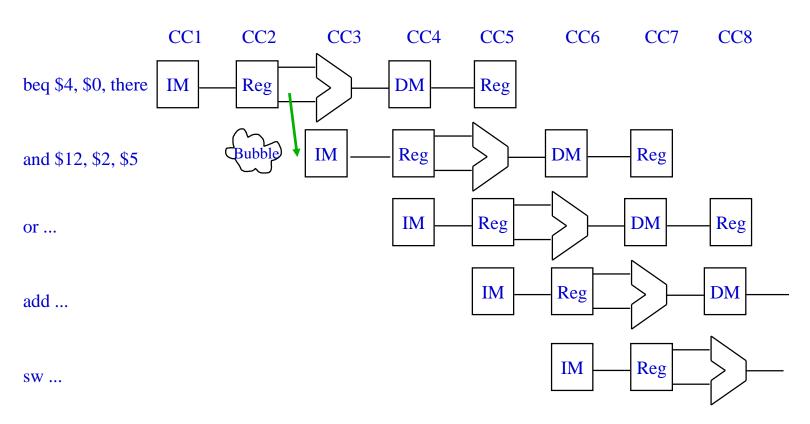
One-Cycle Branch Misprediction Penalty

- Target computation & equality check in ID phase

Stalling for Branch Hazards

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8

beq $4, $0, there IM — Reg — > — DM — Reg

and $12, $2, $5 Bubble IM — Reg — > — DM — Reg

or ... IM — Reg — > — DM — Reg

add ... IM — Reg — > — DM

sw ... IM — Reg — >

Practice
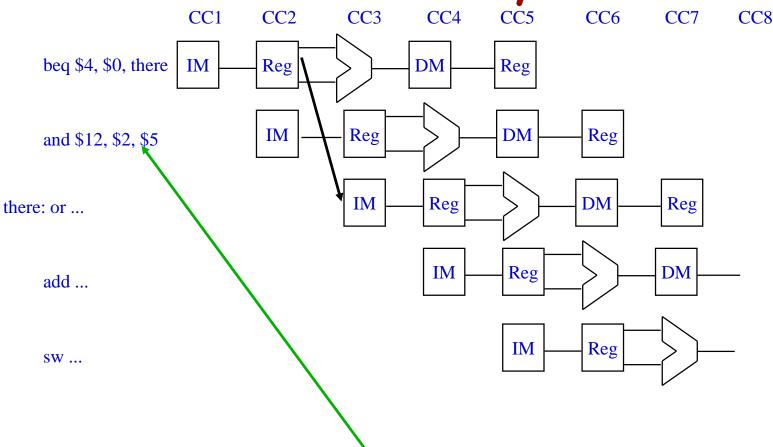
- Which program has more IC?

- Which one has less bubbles?

- Which one runs faster?

- How many clock cycles?

```
        move $s0,$zero              move $s0,$zero
        li $s1, 100                 li $s1, 100
   L1:  beq $s0, $s1, L2       L1:  add $s2,$s2,$s0
        add $s2,$s2,$s0             addi $s0,$s0,1
        addi $s0,$s0,1             bne $s0, $s1, L1
        j L1                  L2:
   L2:
```

Eliminating Branch Stall

- SPARC and MIPS
  - Use a single *branch delay slot* to eliminate single-cycle stalls after branches
  - Instruction after a conditional branch is *always executed* in those machines
    - Regardless of whether branch is taken or not!

Branch Delay Slot

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
beq $4, $0, there	IM	Reg		DM	Reg			
and $12, $2, $5		IM	Reg		DM	Reg		
there: or ...			IM	Reg		DM	Reg	
add ...				IM	Reg		DM	
sw ...					IM	Reg		

Branch delay slot instruction (next instruction after a branch) is executed even if the branch is taken.

Filling Branch Delay Slot

```
add  $5, $3, $7
sub  $6, $1, $4
and  $7, $8, $2
beq $6, $7, there
nop   /* branch delay slot */
add  $9, $1, $2
sub  $2, $9, $5
...
there:
mult $2, $10, $11
```

Backup