Computer Architecture

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

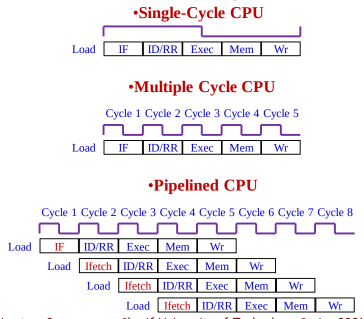
Today's Topics

- 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 2021.
  - "Computer Arch I" handouts, by Prof. Garzarán, UIUC, Spring 2009.
  - "Intro to Computer Organization" handouts, by Prof. Mahlke & Prof. Narayanasamy, Winter 2008.

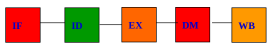
Instruction Latencies and Throughput


•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

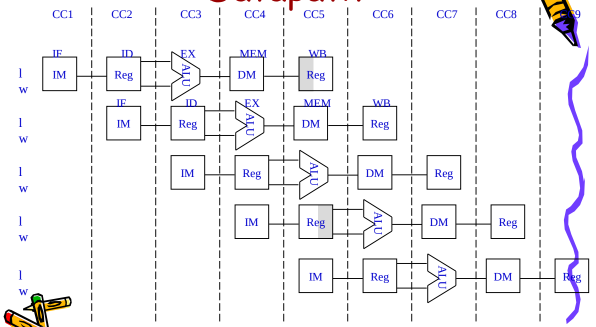
Introduction of Pipeline

Machine assembly line

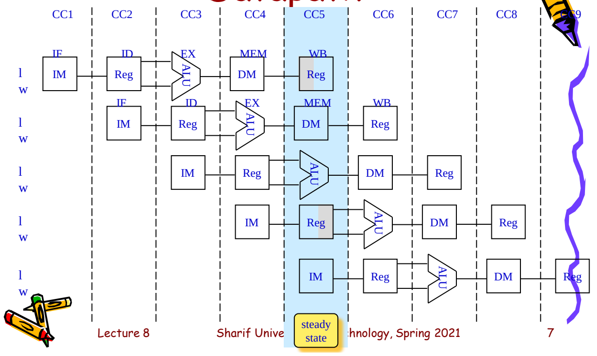
Datapath of pipeline



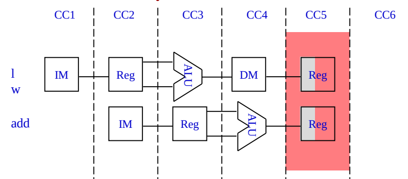
Execution in a Pipelined Datapath



Execution in a Pipelined Datapath

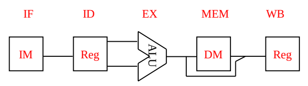


Mixed Instructions in Pipeline



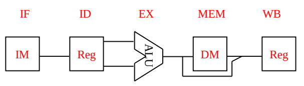
Pipeline Principles

- 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



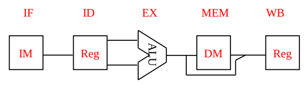
Pipeline Principles (cont.)

- 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)

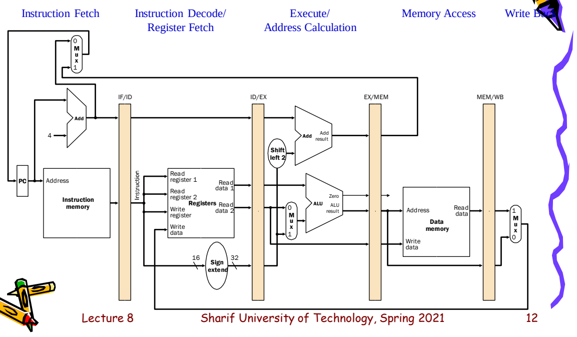


Pipeline Principles

- Question:
  - What if we wan to bypass a stage for an instruction?

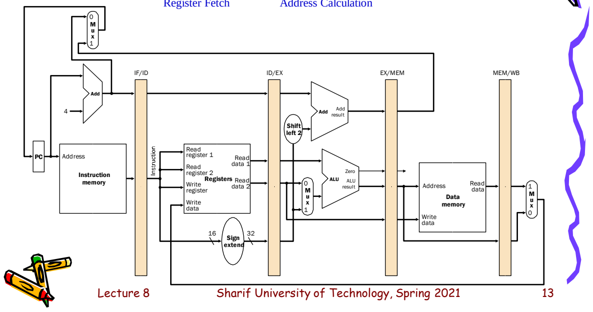


Pipeline in Execution



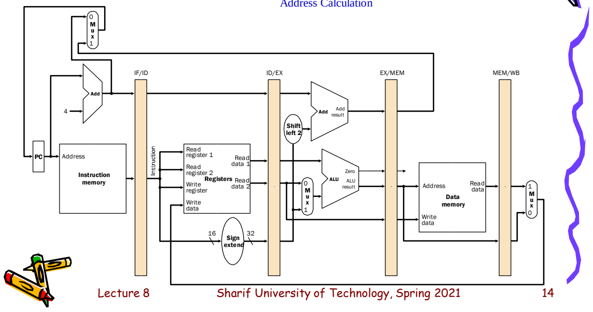
Pipeline in Execution

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



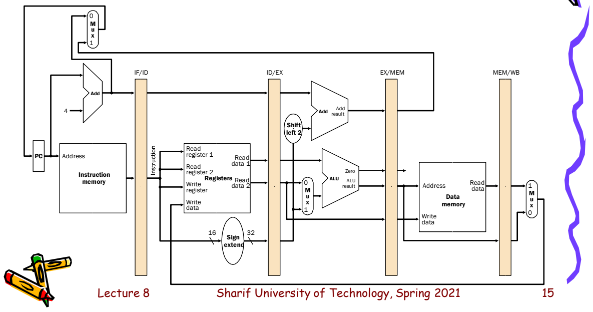
Pipeline in Execution

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



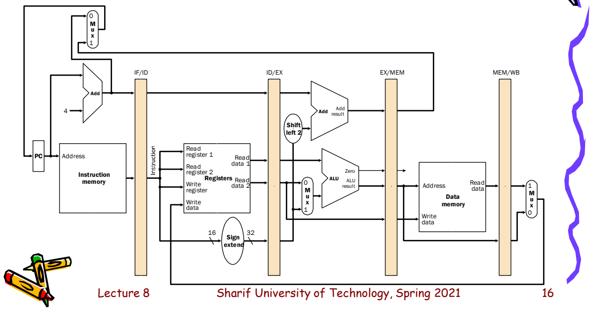
Pipeline in Execution

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



Pipeline in Execution

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



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)?

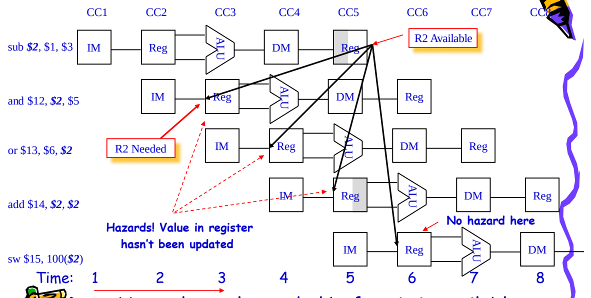
Issues With Pipelining

- 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



Data Hazard: result needed before it is available

Handling Data Hazards

- SW Solution
  - Insert independent instructions
  - No-ops instructions
  - Code reordering
- HW Solutions
  - Insert bubbles (i.e. stall pipeline)
  - Data forwarding
  - Latch-based GPR

Dealing With Data Hazards

- Reorder code to avoid use of load result in next instruction
- C code for A = B + E; C = B + F;

stall → add $t3, $t1, $t2

stall → add $t5, $t1, $t4

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

13 cycles

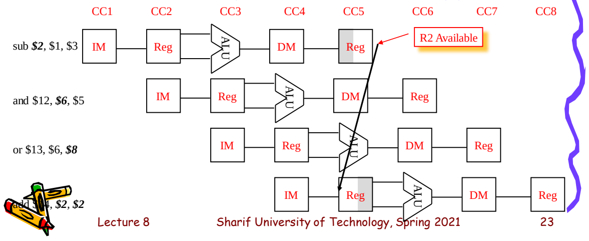
```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

11 cycles

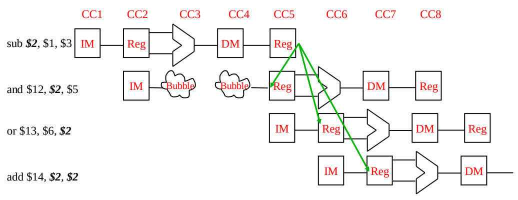
add $14, $2, $2

Dealing With Data Hazards

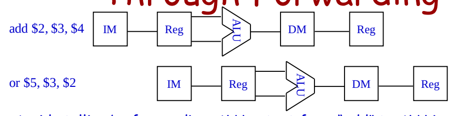
- Using Transparent Register File
  - Use latches rather than flip-flops in RF
    - First half-cycle : register loaded
    - Second half-cycle: new value is read into pipeline state



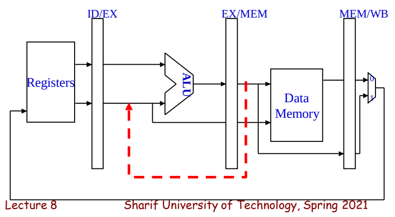
Handling Data Hazards in Hardware: Stall pipeline



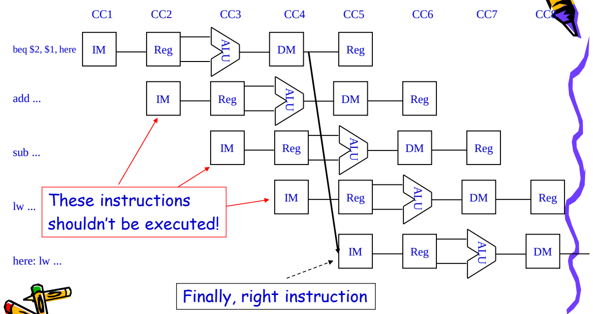
Reducing Data Hazards Through Forwarding



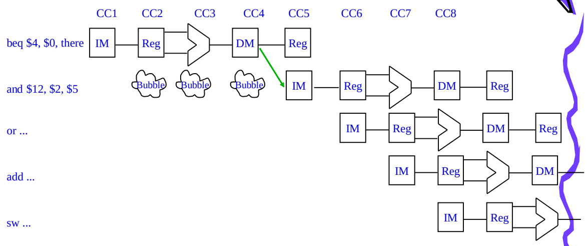
Avoid stalling by forwarding ALU output from "add" to ALU input for "or"



Control or Branch Hazard

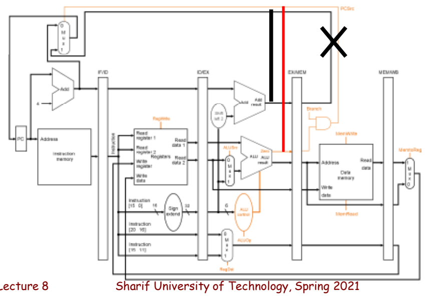


Stalling for Branch Hazards

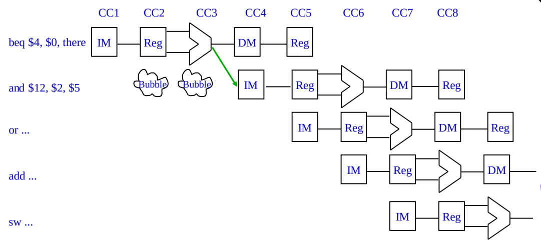


Reducing Branch Delay

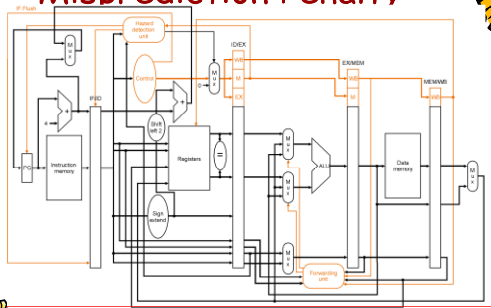
It's easy to reduce stall to 2-cycles



Stalling for Branch Hazards

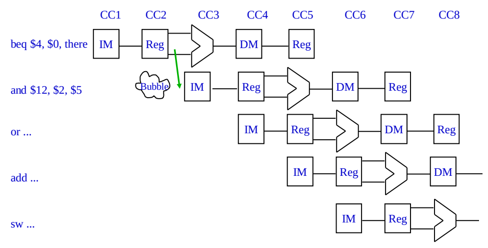


One-Cycle Branch Misprediction Penalty



- Target computation & equality check in ID phase

Stalling for Branch Hazards



Practice

- Which program has more IC?
- Which one has less bubbles?
- Which one runs faster?
- How many clock cycles?

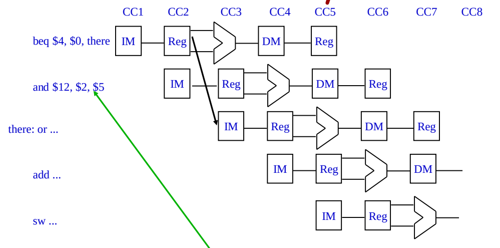
```
     move $s0,$zero
     li $s1, 100
L1:  beq $s0, $s1, L2
     add $s2,$s2,$s0
     addi $s0,$s0,1
     j L1
L2:
```

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



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
```

More-Realistic Branch Prediction

- **Static** Branch Prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- **Dynamic** Branch Prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue trend
    - When wrong ➔ stall while re-fetching & update history

Backup