ECE232: Hardware Organization and Design

Part 11: Pipelining
Chapter 4/6

http://www.ecs.umass.edu/ece/ece232/

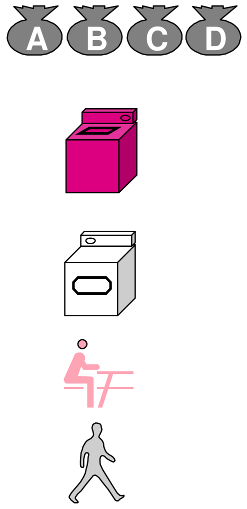
Adapted from *Computer Organization and Design*, Patterson & Hennessy, UCB

CPI Calculation

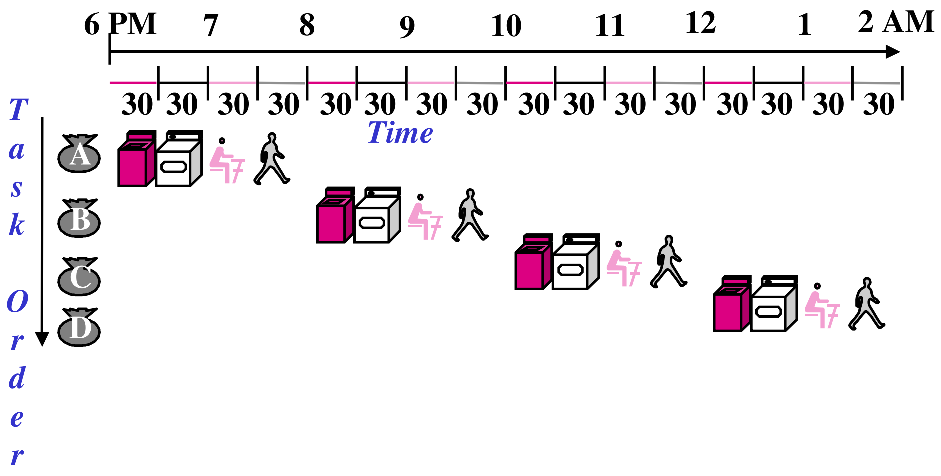
- CPI stands for average number of **Cycles Per Instruction**
- Assume an instruction mix of 24% loads, 12% stores, 44% R-format, 18% branches, and 2% jumps
- CPI = 0.24 * 5 + 0.12 * 4 + 0.44 * 4 + 0.18 * 3 + 0.02 * 3 = 4.04
- Speedup?
- Question: Can we achieve a CPI of 1???

Speeding up through pipelining

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - "Folder" takes 30 minutes
  - "Stasher" takes 30 minutes to put clothes into drawers

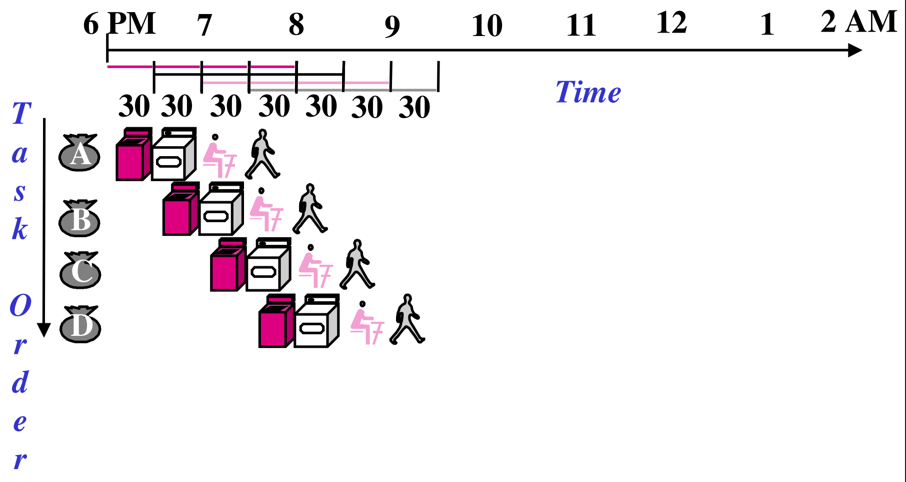


Sequential Laundry



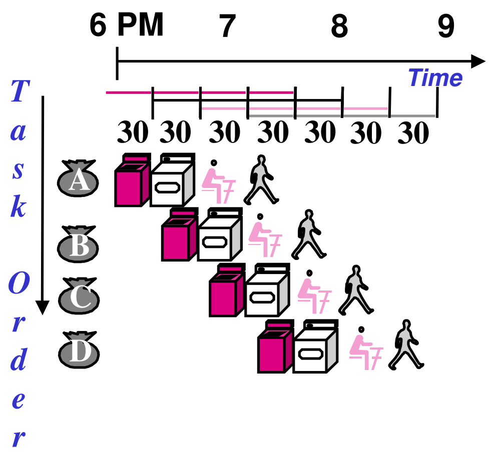
- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



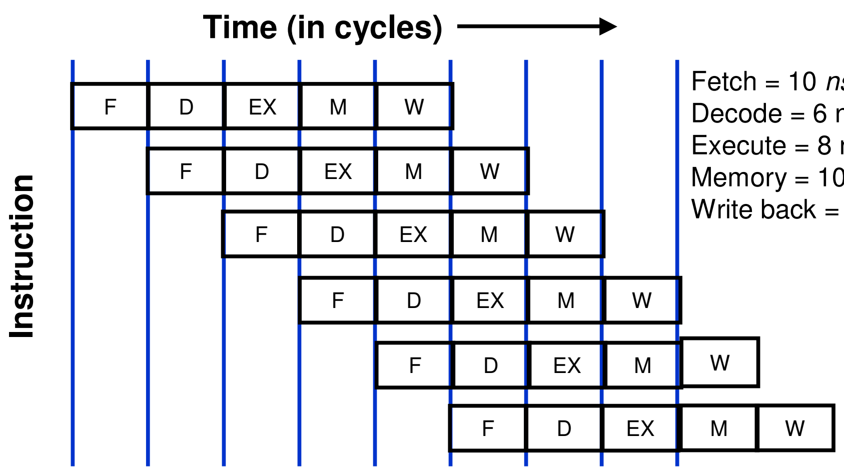
- Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Lessons



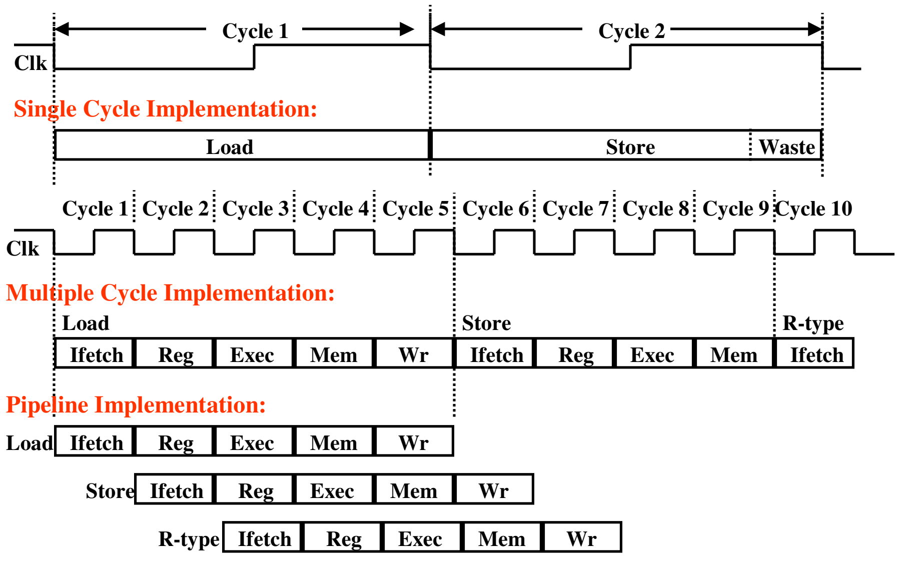
- Pipelining doesn't help *latency* of single task, it helps *throughput* of entire workload
- *Multiple* tasks operating simultaneously using different resources
- Potential speedup = *Number pipe stages*
- Pipeline rate limited by *slowest* pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "*fill*" pipeline and time to "*drain*" it reduces speedup

Pipelining Instructions



Fetch = 10 *ns*
Decode = 6 ns
Execute = 8 ns
Memory = 10 ns
Write back = 6 ns

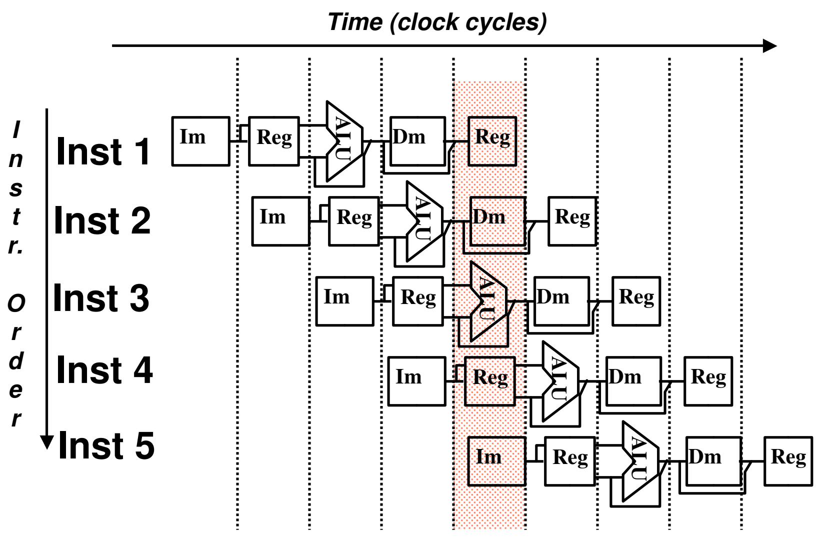
Single Cycle, Multiple Cycle, vs. Pipeline


Cycle 1
Cycle 2
Clk
Single Cycle Implementation:
Load
Store
Waste
Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8 Cycle 9 Cycle 10
Clk
Multiple Cycle Implementation:
Load
Store
R-type
Ifetch Reg Exec Mem Wr Ifetch Reg Exec Mem Ifetch
Pipeline Implementation:
Load Ifetch Reg Exec Mem Wr
Store Ifetch Reg Exec Mem Wr
R-type Ifetch Reg Exec Mem Wr

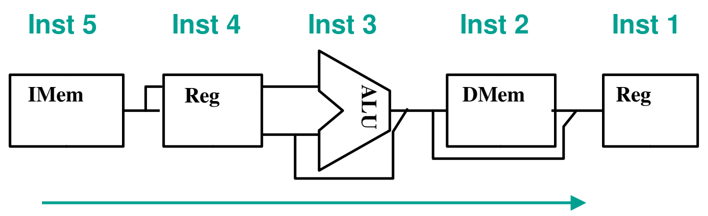
Why Pipeline?

- Suppose we execute 100 instructions
- Single Cycle Machine
  - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
- Multicycle Machine
  - 10 ns/cycle x 4.04 CPI (for the given inst mix) x 100 inst = 4040 ns
  - Instruction mix of 24% loads, 12% stores, 44% R-format, 18% branches, and 2% jumps
- Ideal pipelined machine (with 5 stages)
  - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns
- Speedup=4.33 vs. single-cycle
- 3.88 vs. multi-cycle (for the given inst mix)

Why Pipeline? Because the resources are there!



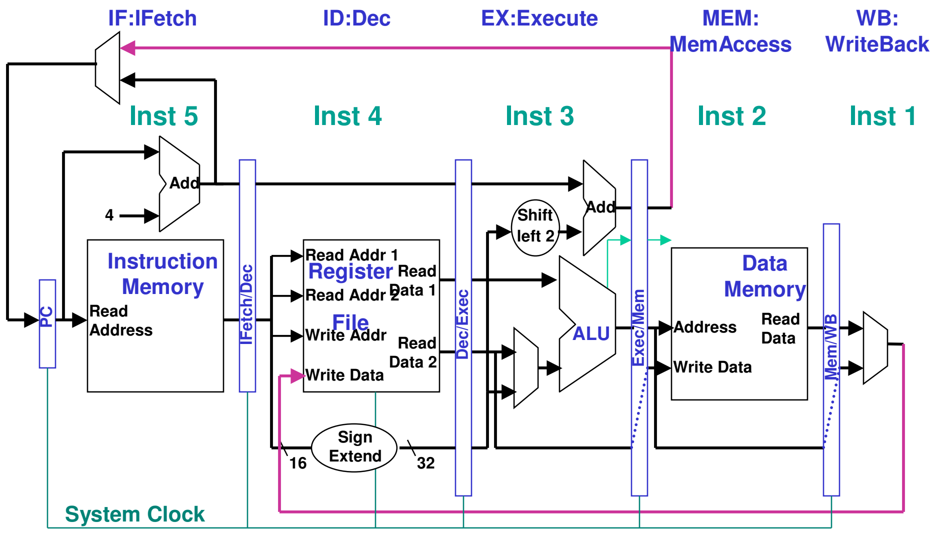
Pipelining Rules



- Forward traveling signals at each stage are latched
- Only perform logic on signals in the same stage
  - signal labeling useful to prevent errors,
    - e.g., IR_R, IR_A, IR_M, IR_W
- Backward travelling signals at each stage represent *hazards*

MIPS Pipelined Datapath

- **State registers** between pipeline stages to **isolate** them



Pipeline Hazards

- Data hazards: an instruction uses the result of a previous instruction (RAW)

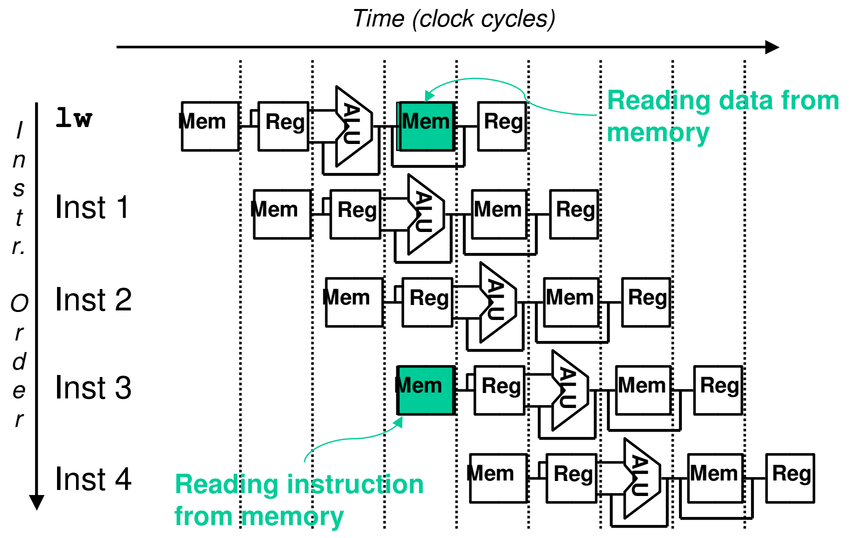
```
ADD    R1, R2, R3     or      SW     R1, 4(R2)
SUB    R4, R1, R5             LW     R3, 4(R2)
```

- Control hazards: the address of the next instruction to be executed depends on a previous instruction

```
          BEQ R1,R2,CONT
          SUB R6,R7,R8
          …
    CONT: ADD R3,R4,R5
```

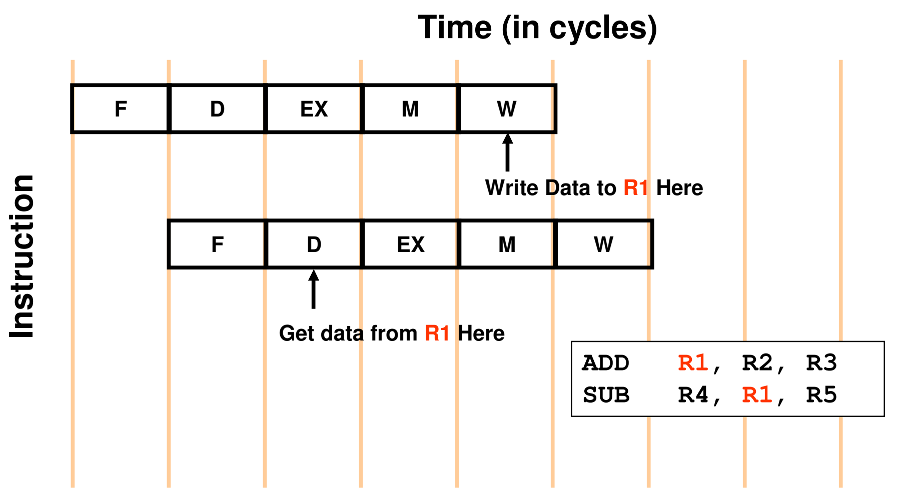
- Structural hazards: two instructions need access to the same resource
  - e.g., single memory shared for instruction fetch and load/store

Structural Hazard



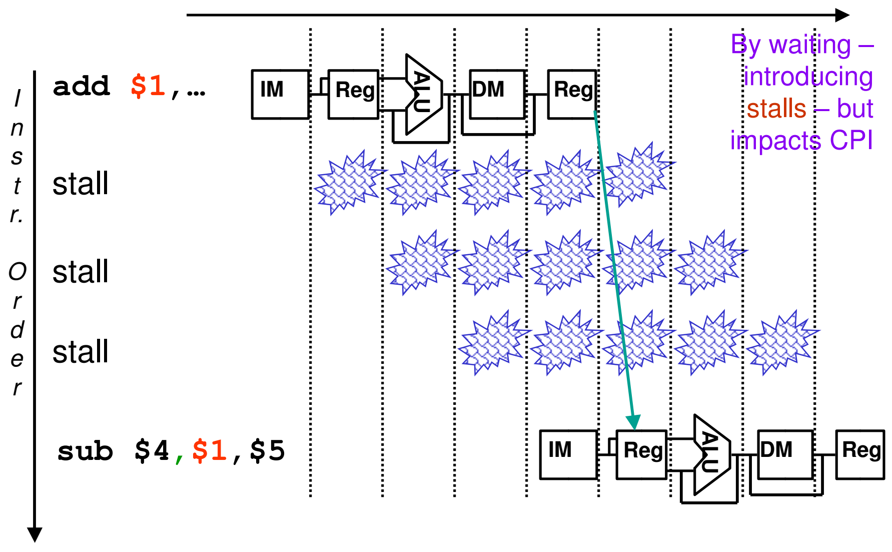
- Fix with separate instruction and data memories (I$ and D$)

Data Hazards (RAW)



```
ADD    R1, R2, R3
SUB    R4, R1, R5
```

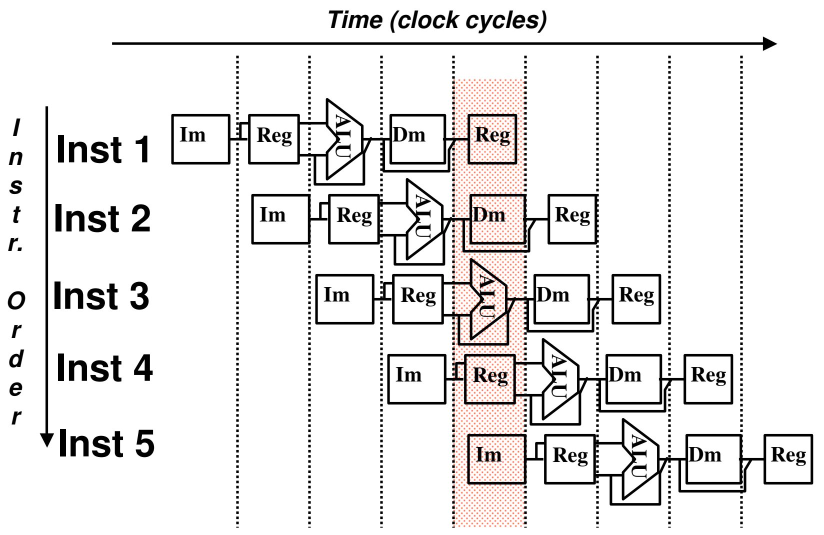
One Way to handle a Data Hazard



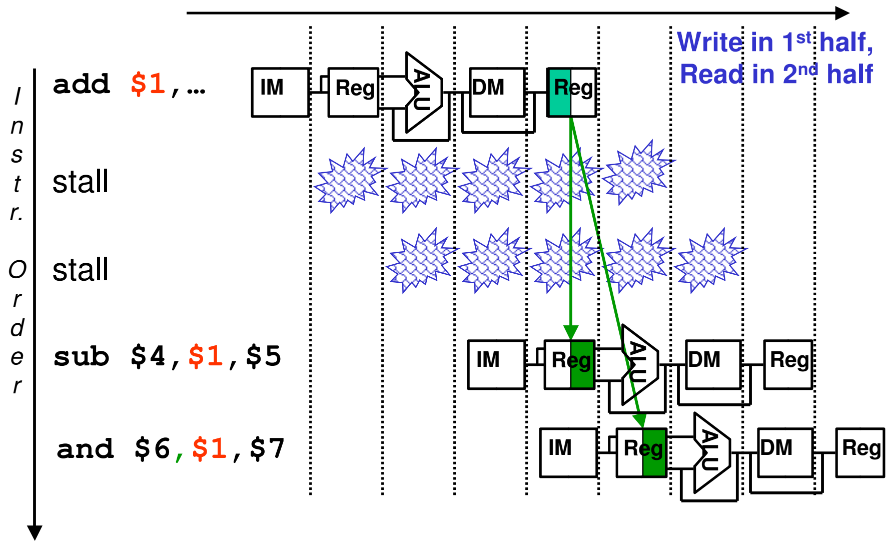
By waiting – introducing stalls – but impacts CPI

Must allow Wr/Rd in REG in same cycle

Split cycle into two halves



Only two stall cycles



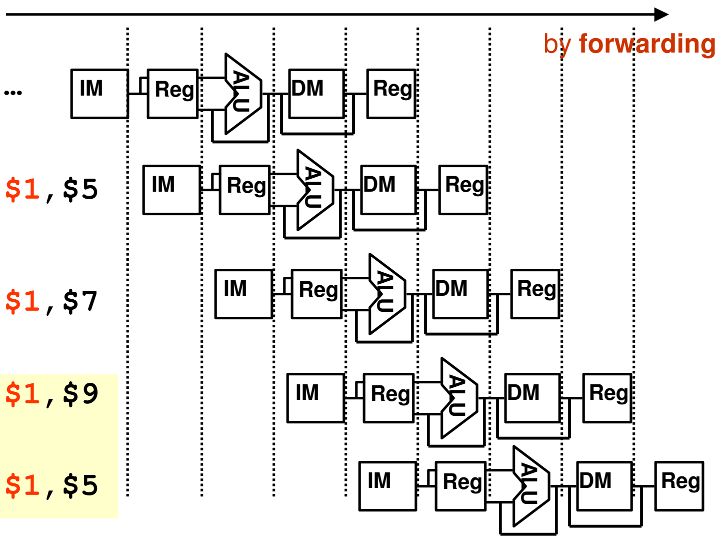
Another Way to “Fix” a Data Hazard

Time

by **forwarding**

Instr. Order

```
add  $1,…
sub  $4,$1,$5
and  $6,$1,$7
or   $8,$1,$9
xor  $4,$1,$5
```



Register File (write and then read)

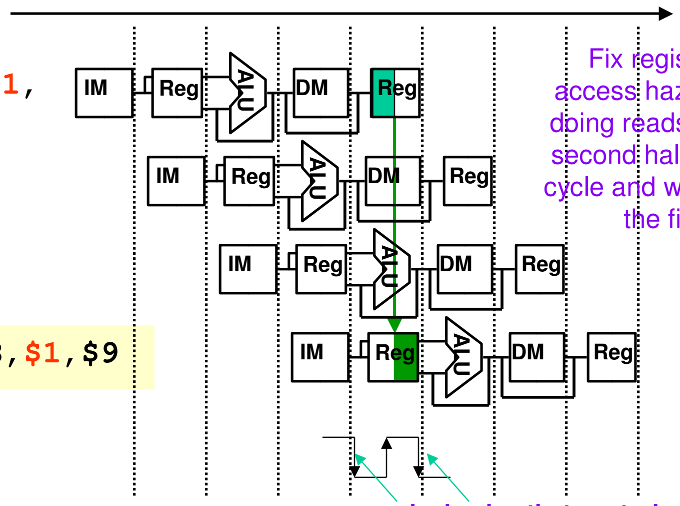
Time (clock cycles)

Instr. Order

```
add $1,
```
Inst 1

Inst 2

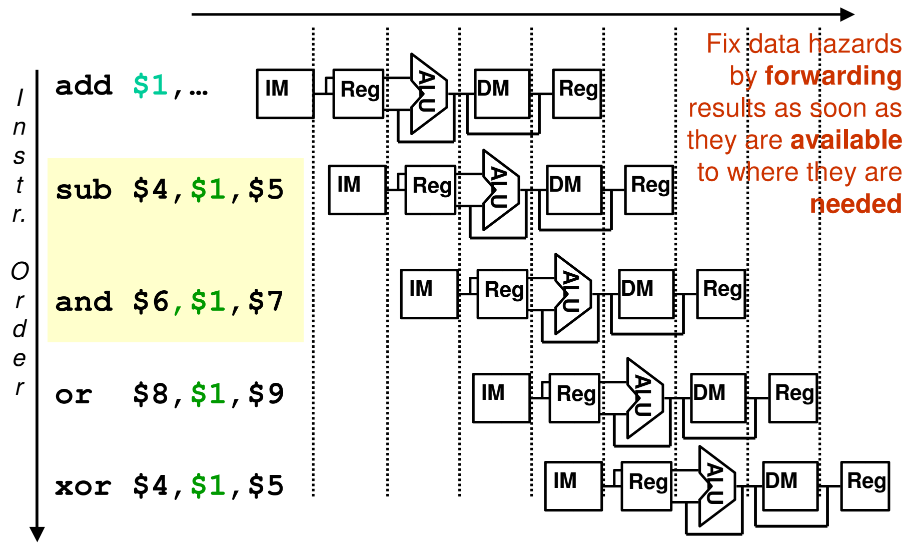
```
or $8,$1,$9
```



Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half

clock edge that controls loading of pipeline state registers

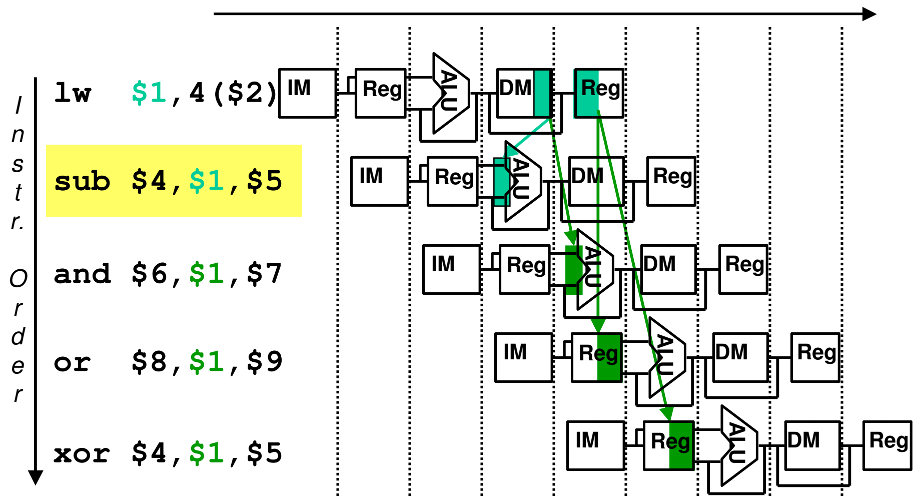
Internal data forwarding



Fix data hazards by **forwarding** results as soon as they are **available** to where they are **needed**

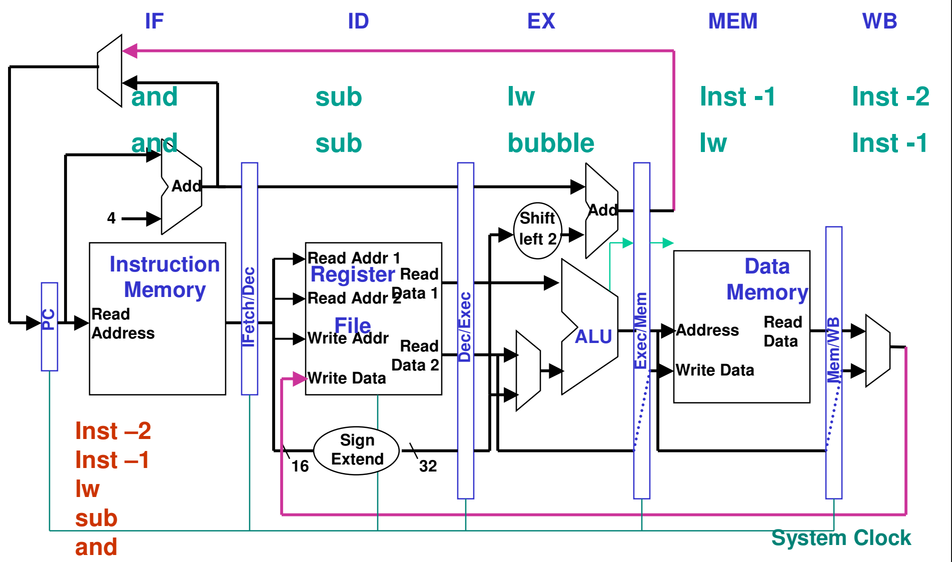
ALU-to-ALU forwarding vs. full forwarding

Forwarding with Load-use Data Hazards



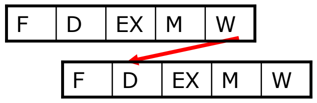
- sub needs to stall
- Will still need one stall cycle even with forwarding

Injecting Bubbles



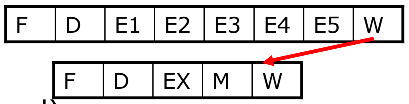
3 Types of Data Hazards

- RAW (read after write)
  - only hazard for 'fixed' pipelines
  - later instruction must *read* after earlier instruction *writes*



add $1,$2,$3
sub $4,$1,$5

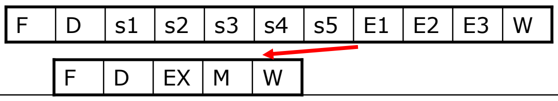
- WAW (write after write)
  - variable-length pipeline
  - later instruction must *write* after earlier instruction *writes*



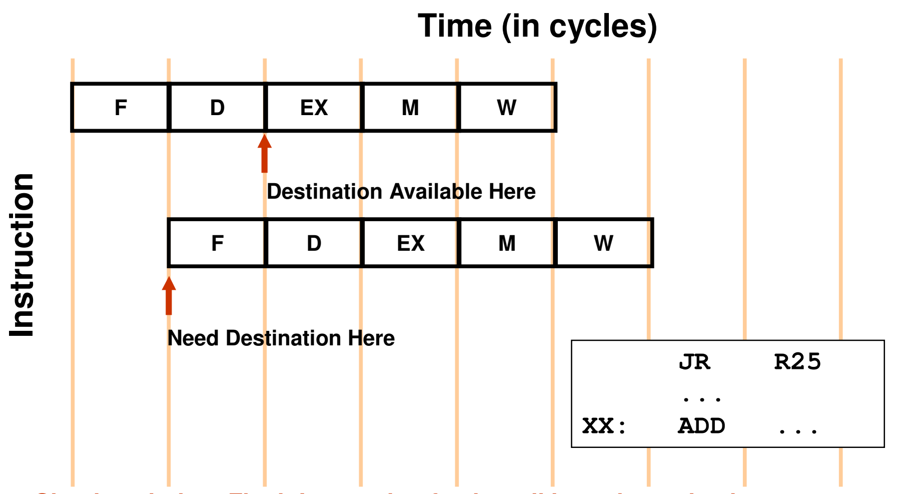
div $1,$4,$3
add $1,$2,$5

- WAR (write after read)
  - instruction with late read (e.g., waiting for an execution unit)
  - later instruction must *write* after earlier instruction *reads*

mlt $4,$1,$3
add $1,$2,$5



Control Hazard



```
      JR     R25
      ...
XX:   ADD    ...
```

Simple solution: Flush Instruction fetch until branch resolved