



Computer Architecture

Lecture 4: Pipelining

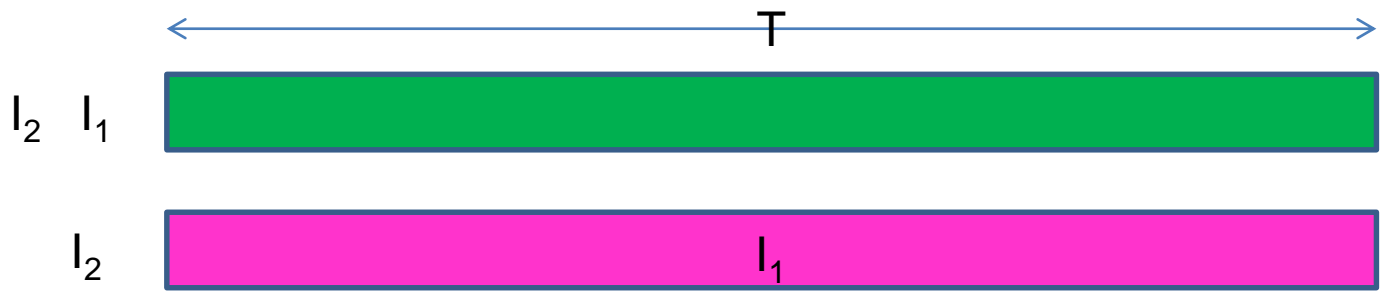
(Appendix C)

Chih-Wei Liu 劉志尉

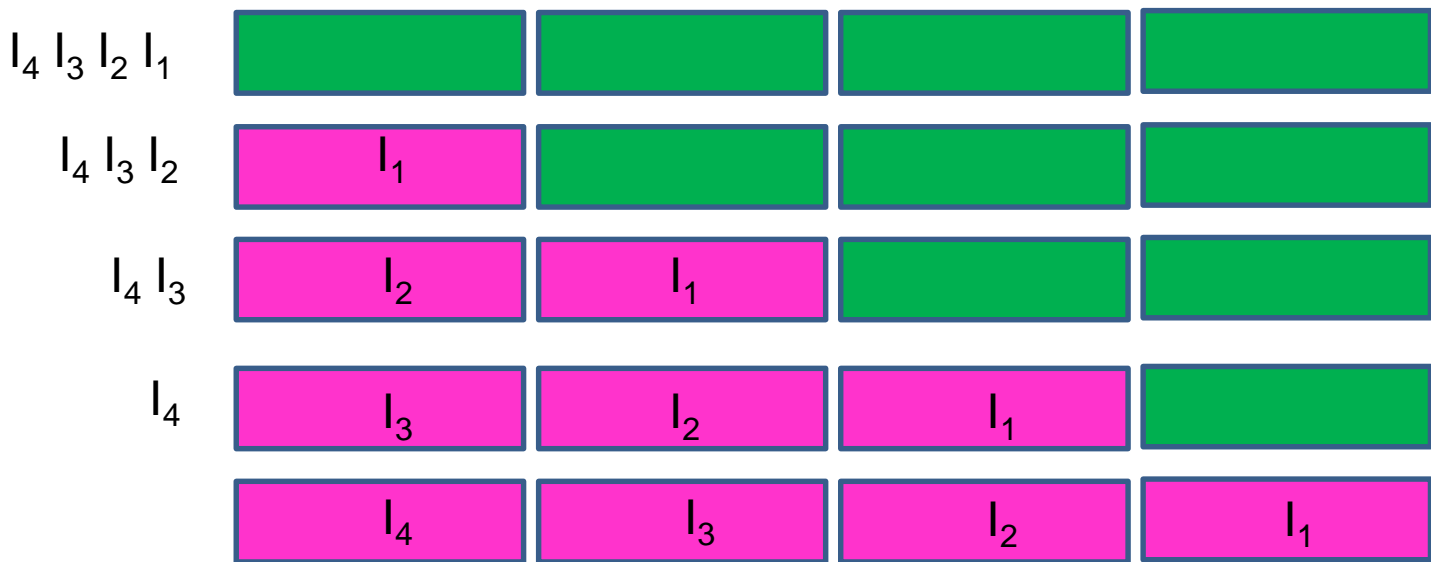
National Chiao Tung University

cwliu@twins.ee.nctu.edu.tw

Why Pipeline?



Throughput = $1/T$



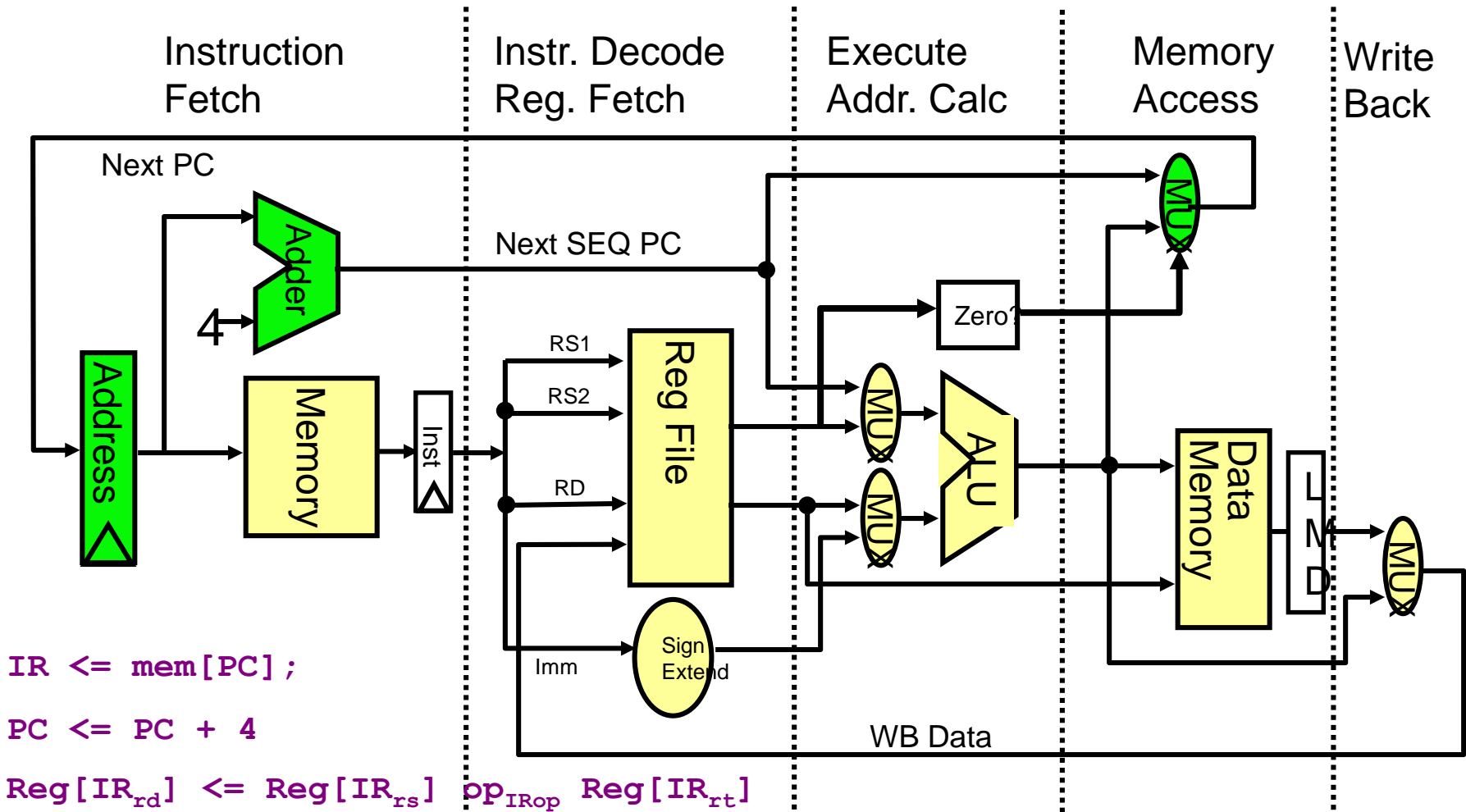
Designing a Pipelined Processor

- Start with ISA
- Examine the datapath and control diagram
 - Starting with single- or multi-cycle datapath?
 - Single- or multi-cycle control?
- Partition datapath into steps
- Insert pipeline registers between successive steps
- Associate resources with steps
- Ensure that flows do not conflict, or figure out how to resolve
- Assert control in appropriate stage

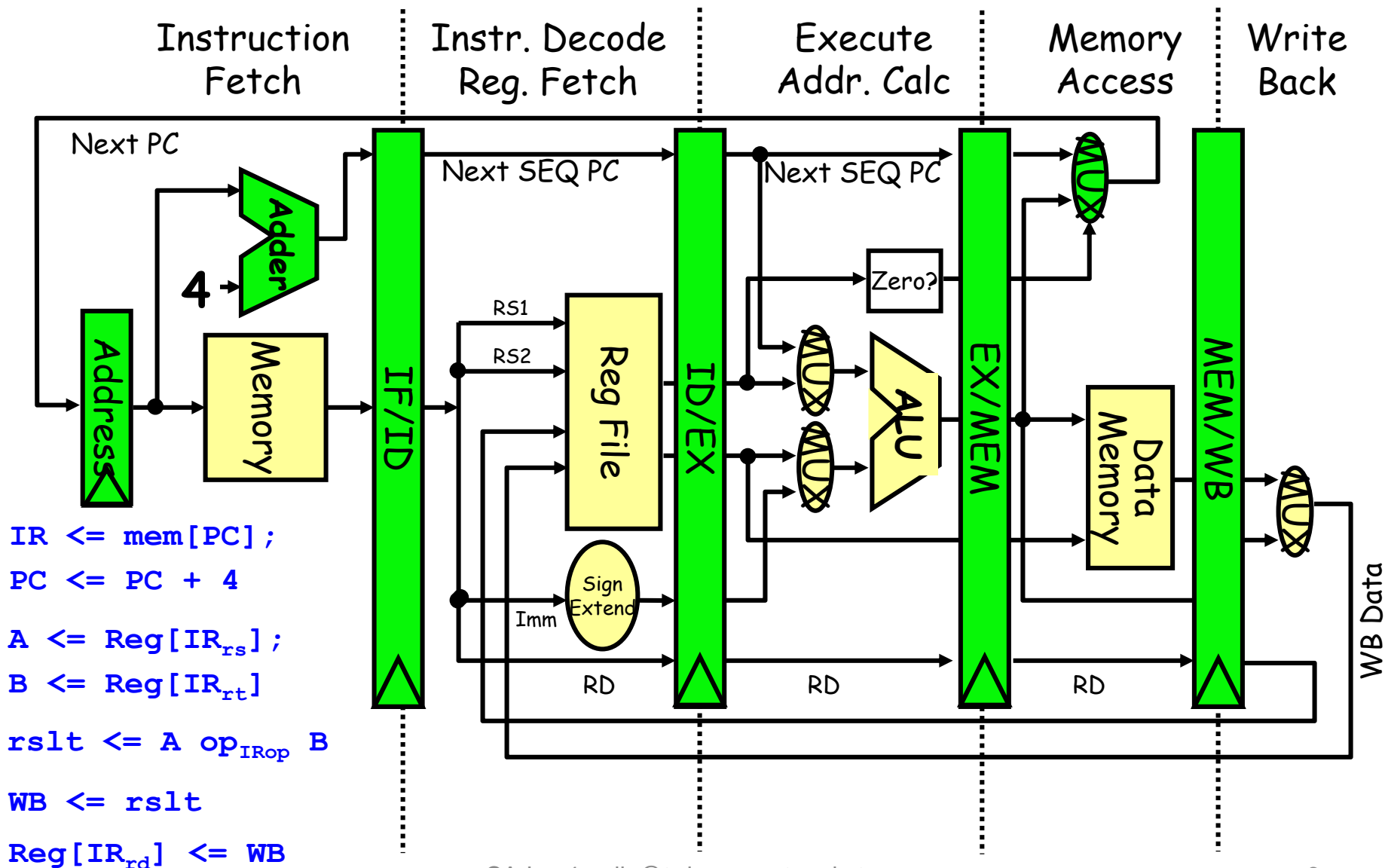
The Basics of the RISC V Instruction Set

- The load-store architecture
 - All operations on data apply to data in registers and typically change the entire register (32 or 64 bits per register).
 - The only operations that affect memory are load and store operations. Load and store operations that load or store less than a full register (e.g., a byte, 16 bits, or 32 bits) are often available.
 - The instruction formats are few in number, with all instructions typically being one size. In RISC V, the register specifiers: rs1, rs2, and rd are always in the same place simplifying the control.
- Simple implementation of a RISC instruction set
 - IF, ID, EX, MEM, WB

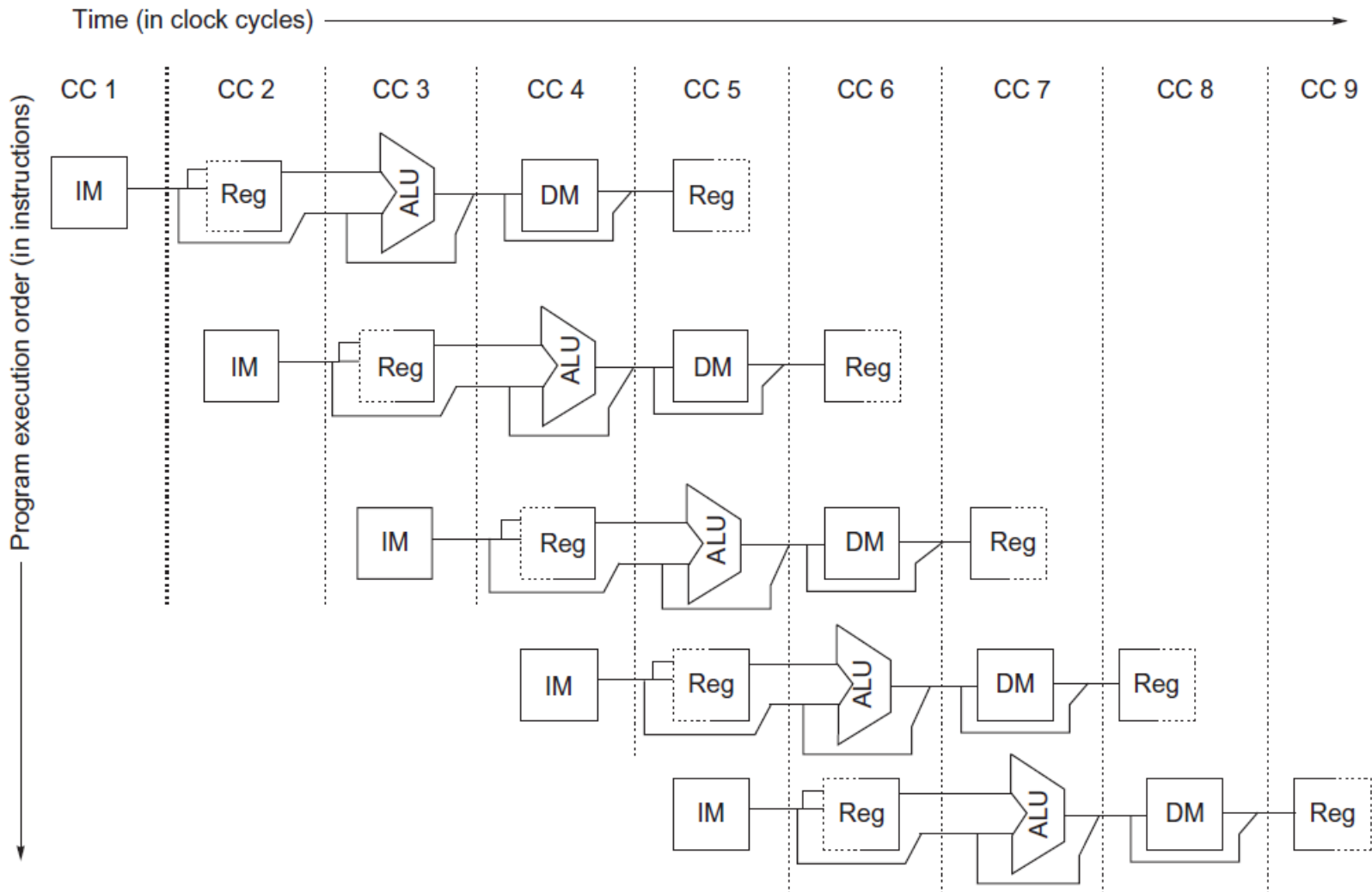
Example: 5 Steps of MIPS Datapath



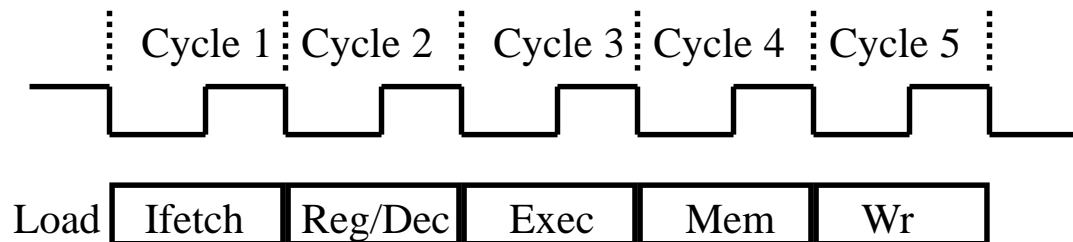
5-Stage Pipelined Datapath



Visualizing Pipelining



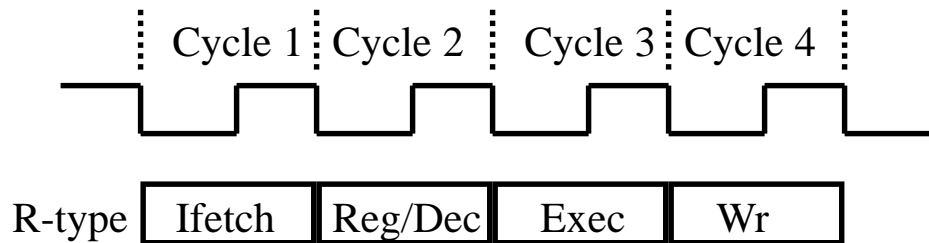
The 5-Step of the Lw-Instruction



- Lw-instruction can be implemented in **5 clock cycles**
- **Ifetch**: Instruction Fetch
 - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Execution and calculate the memory address
- **Mem**: Read the data from the Data Memory
- **Wr**: Write the data back to the register file

Branch requires ? cycles, Store requires ? cycles, others require ? cycles

The 4-Step of R-type Instruction



does not access data memory...

- Ifetch: Instruction Fetch
 - Fetch the instruction from the Instruction Memory
 - Update PC
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec:
 - ALU operates on the two register operands
- Wr: Write the ALU output back to the register file

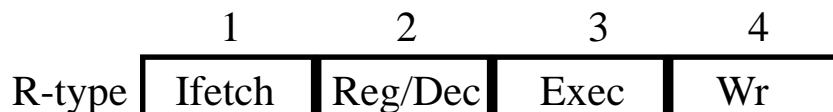
Important Observation

- Each functional unit can only be used **once** per instruction:
 - Load uses Register File's Write Port during its **5th** step



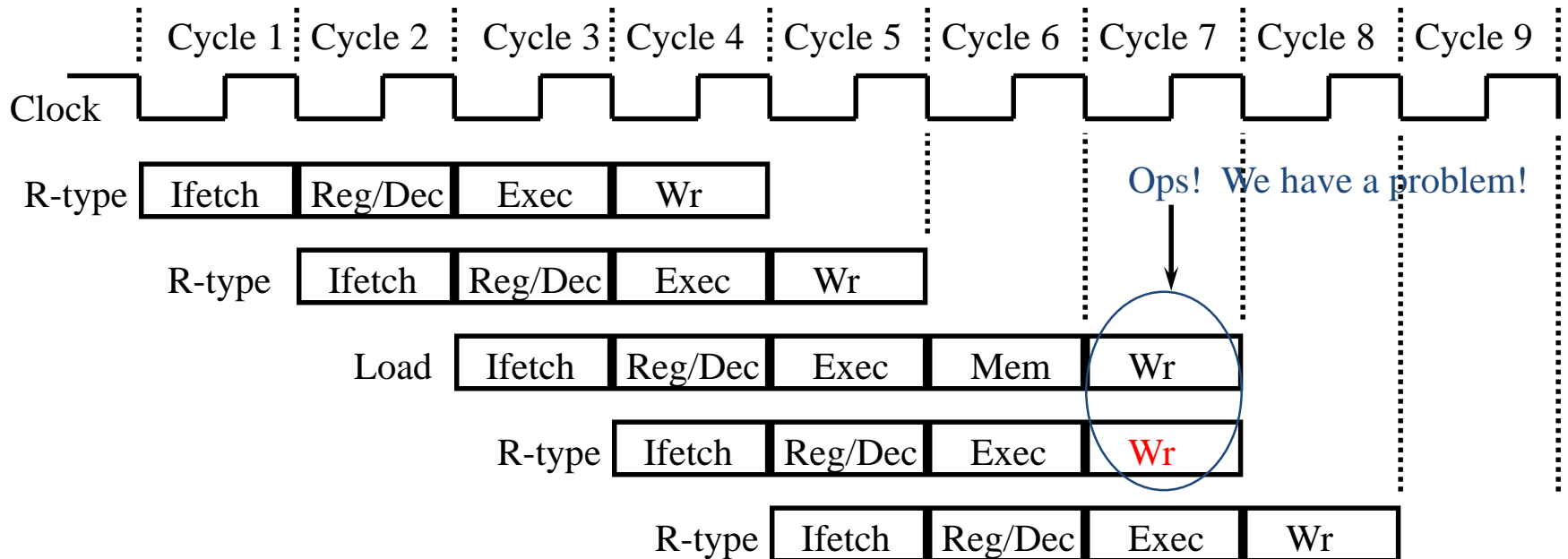
This's what caused the problem

- R-type uses Register File's Write Port during its **4th** step



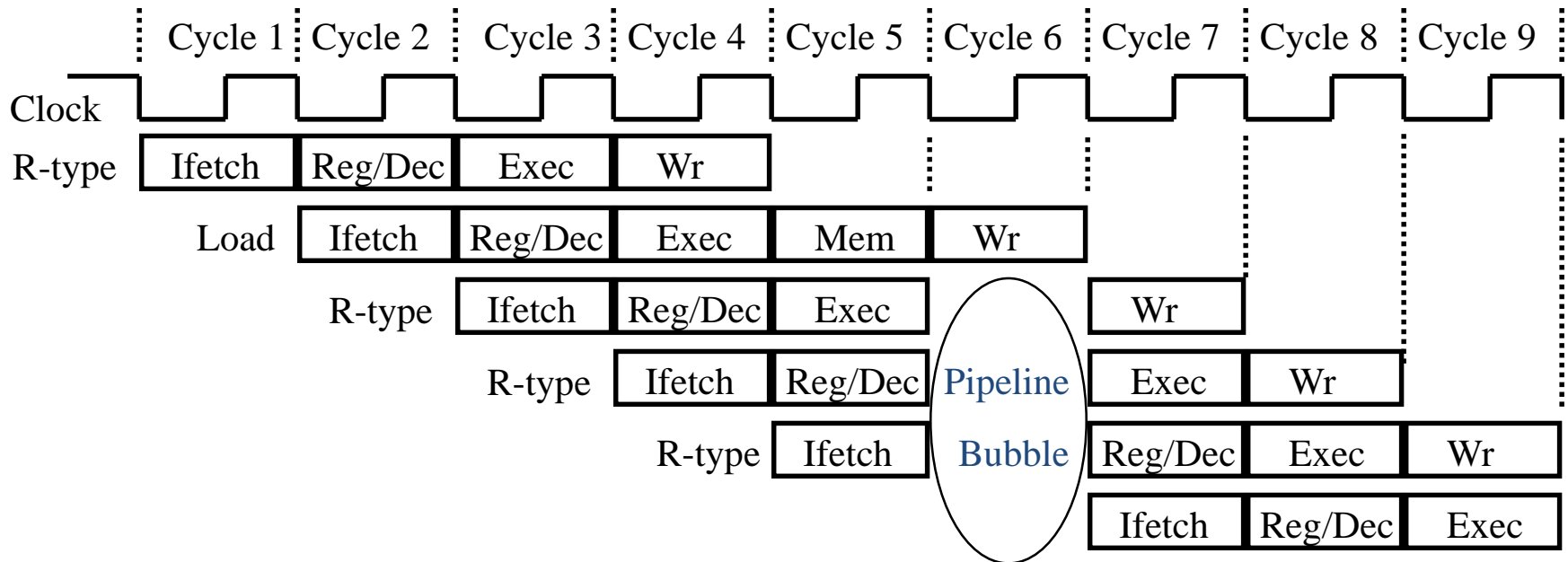
A structural hazard will be happened !!

Pipelining the R-type and Load Instructions



- Structural hazard:
 - Two instructions try to write to the register file at the same time!
 - Only one write port

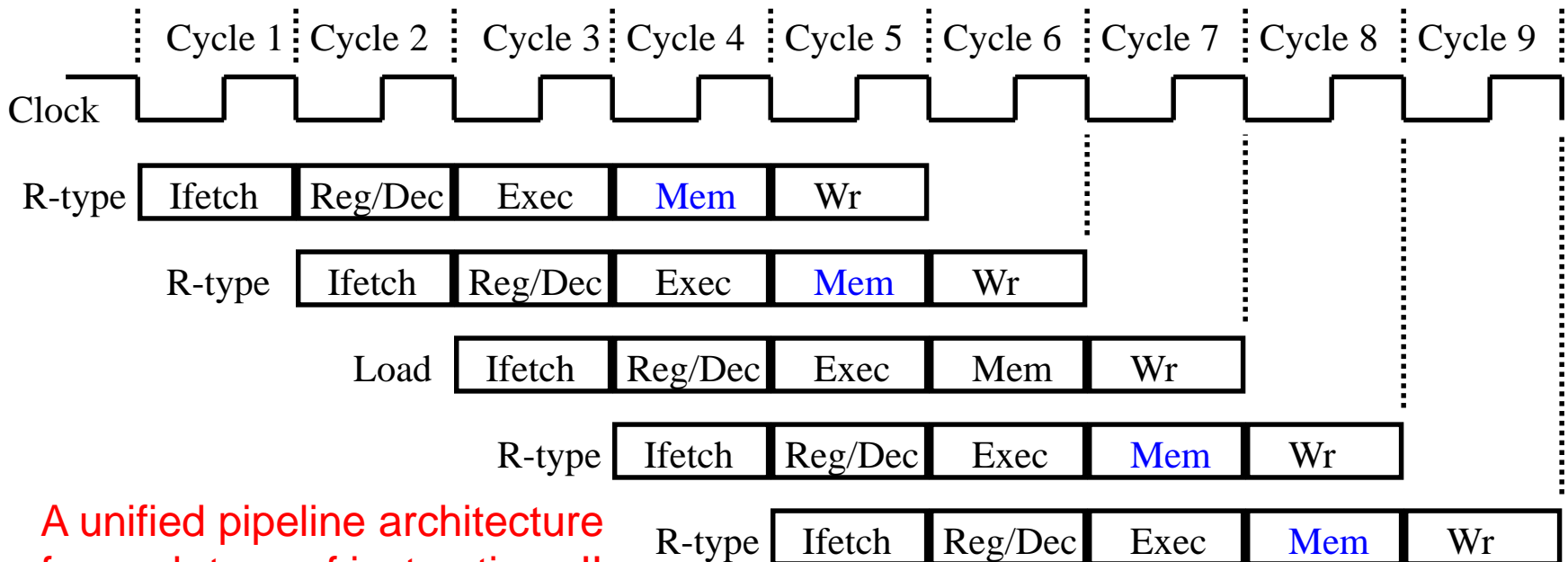
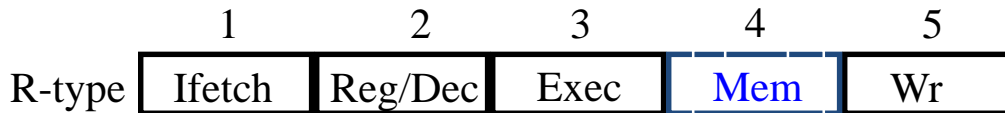
Sol 1: Insert “Bubble” into the Pipeline



- Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle
 - The control logic can be complex.
 - Lose instruction fetch and issue opportunity.

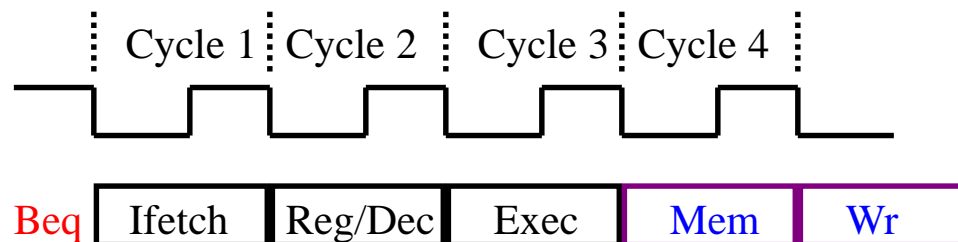
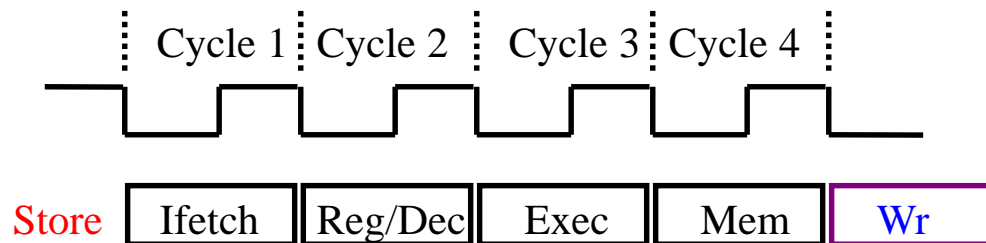
Sol 2: Delay R-type's Write by One Cycle

- 5-step R-type instructions:
 - Mem step for R-type inst. is a NOOP : nothing is being done.



A unified pipeline architecture for each type of instructions !!

Similarly, for Store and Branch Instructions



Pipeline Summary

- A pipeline is like an hooked assembly line.
- Pipelining, in general, is not visible to the programmer (vs ILP)
- Pipelining doesn't help latency of single task, it helps **throughput** of entire workload
- **Pipeline rate limited by slowest pipeline stage**
- Multiple tasks operating simultaneously using different resources
- **Potential speedup = Number pipe stages**, if perfectly balanced stage.
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
- **Pipeline hazard**

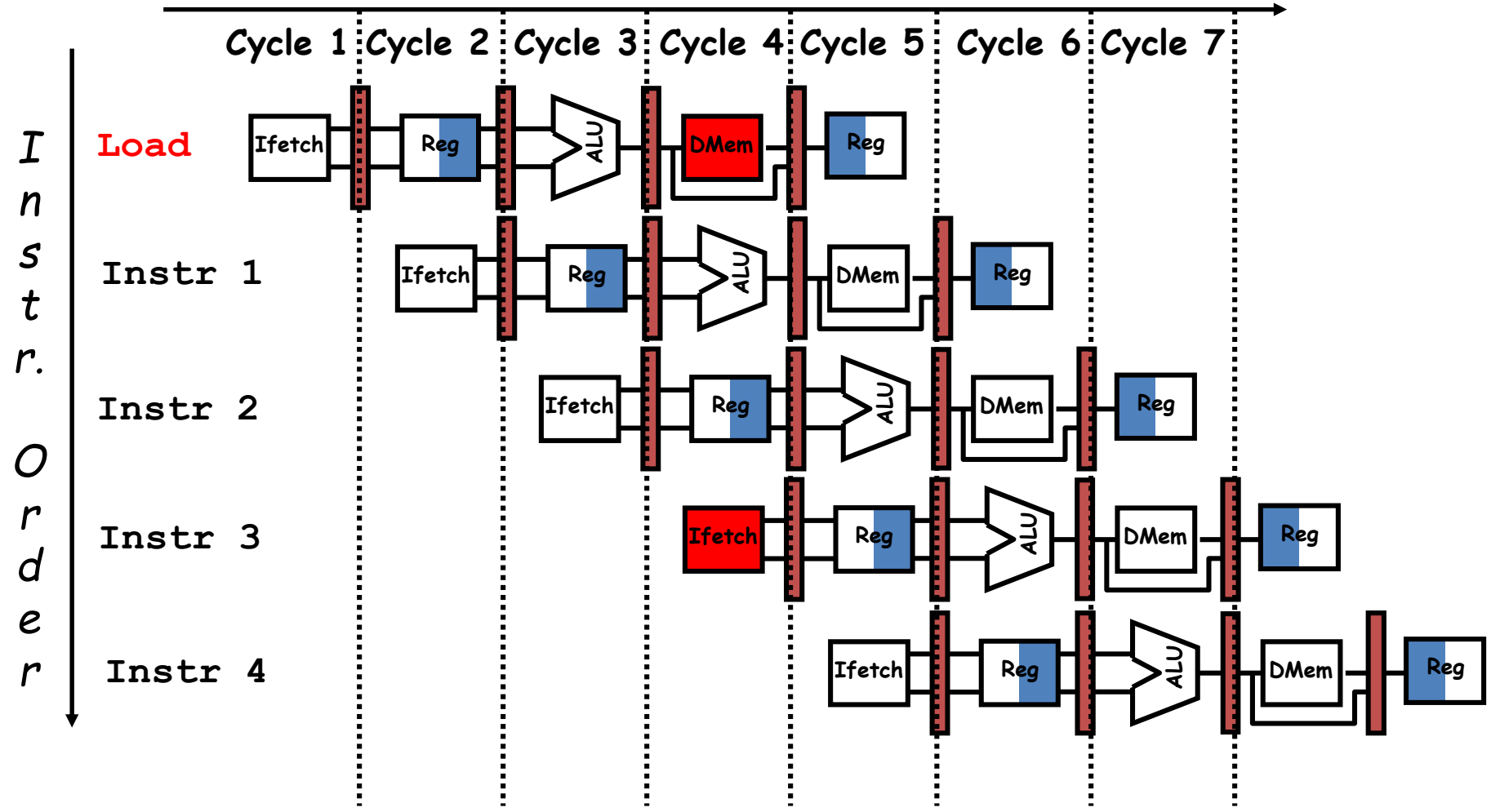
Pipelining is not quite that easy!

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
 - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
 - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
 - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

One Memory Port/Structural Hazards

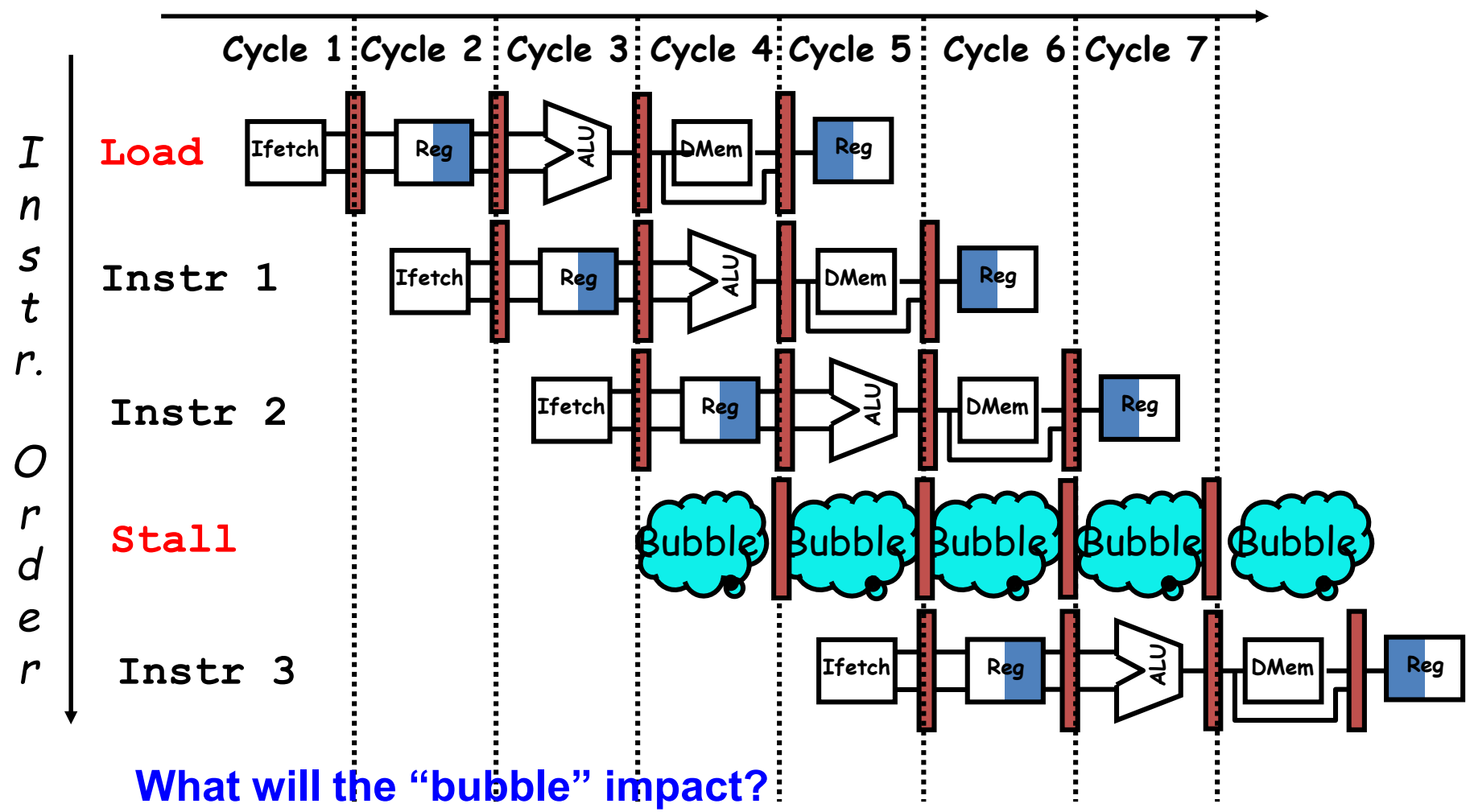
Figure A.4, Page A-14

Time (clock cycles)



One Memory Port/Structural Hazards

Time (clock cycles)



Speed Up Equations for Pipelining

$$\text{Speedup} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{CPI_{\text{unpipelined}}}{CPI_{\text{pipelined}}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$CPI_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

for balanced pipelining

For simple RISC pipeline, $CPI = 1$:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

Example: Dual-port vs. Single-port

- Machine A: Dual ported memory (“Harvard Architecture”)
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Suppose that Loads are 40% of instructions executed

$$\begin{aligned} \text{SpeedUp}_A &= \text{Pipeline Depth} / (1 + 0) \times (\text{clock}_{\text{unpipe}} / \text{clock}_{\text{pipe}}) \\ &= \text{Pipeline Depth} \end{aligned}$$

$$\begin{aligned} \text{SpeedUp}_B &= \text{Pipeline Depth} / (1 + 0.4 \times 1) \times (\text{clock}_{\text{unpipe}} / (\text{clock}_{\text{unpipe}} / 1.05)) \\ &= (\text{Pipeline Depth} / 1.4) \times 1.05 \\ &= 0.75 \times \text{Pipeline Depth} \end{aligned}$$

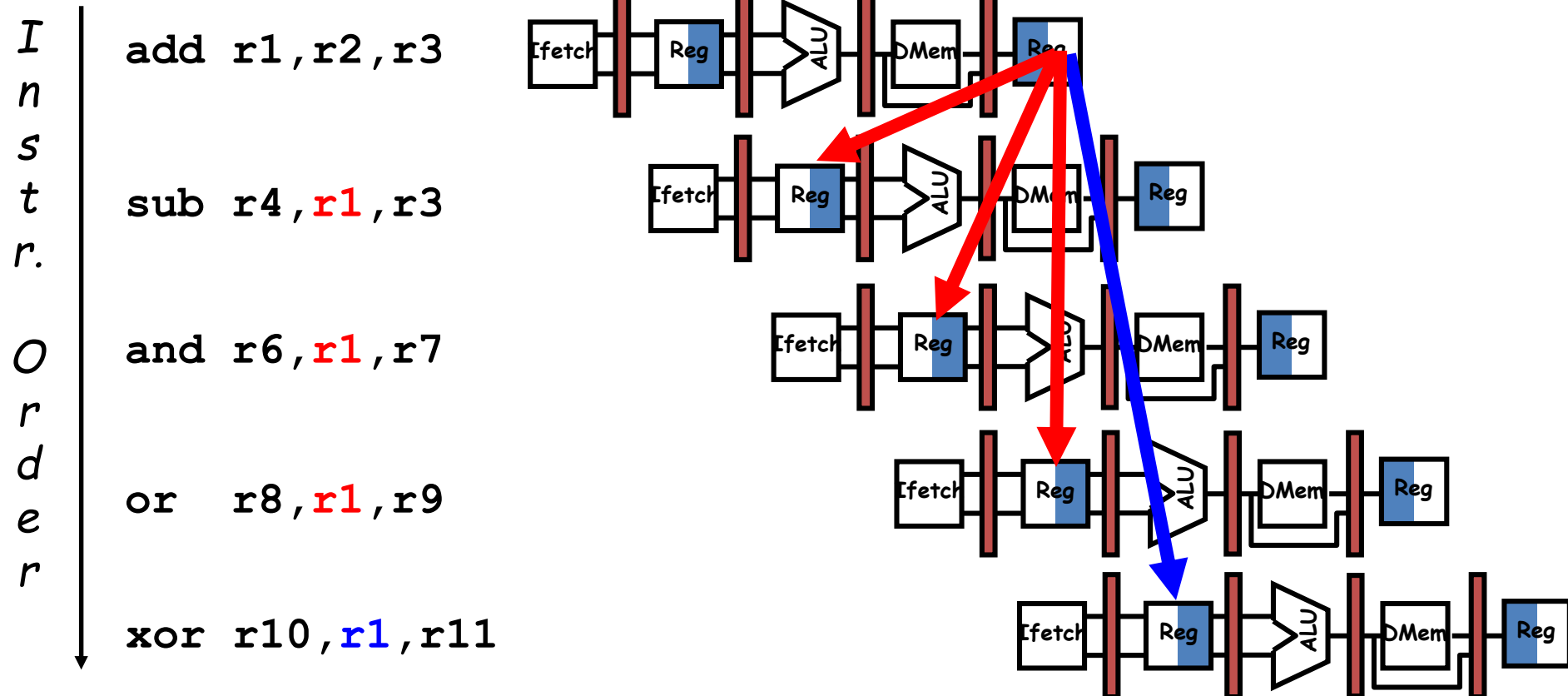
$$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth} / (0.75 \times \text{Pipeline Depth}) = 1.33$$

- Machine A is 1.33 times faster

Data Hazard Problem on r1

Dependencies backwards in time are hazards

Time (clock cycles)

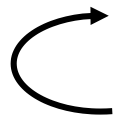


Types of Data Hazards

- RAW (read after write): true data dependence
 - Get wrong propagation result
- WAR (write after read): anti-dependence
 - Get wrong operand
- WAW (write after write): output dependence
 - Leave wrong result

Name Dependence Data Hazards

- Write After Read (WAR)
Instr_j writes operand before Instr_i reads it

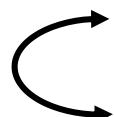
 I: **sub r4, r1, r3**
J: **add r1, r2, r3**
K: **mul r6, r1, r7**

- Called an “anti-dependence” by compiler writers.
- This results from reuse of the name “r1”.
- Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages. Reads are always in stage 2 and Writes are always in stage 5
- Can happen in OOO processor (discussed in chapter 3).

Name Dependence Data Hazards

- Write After Write (WAW)

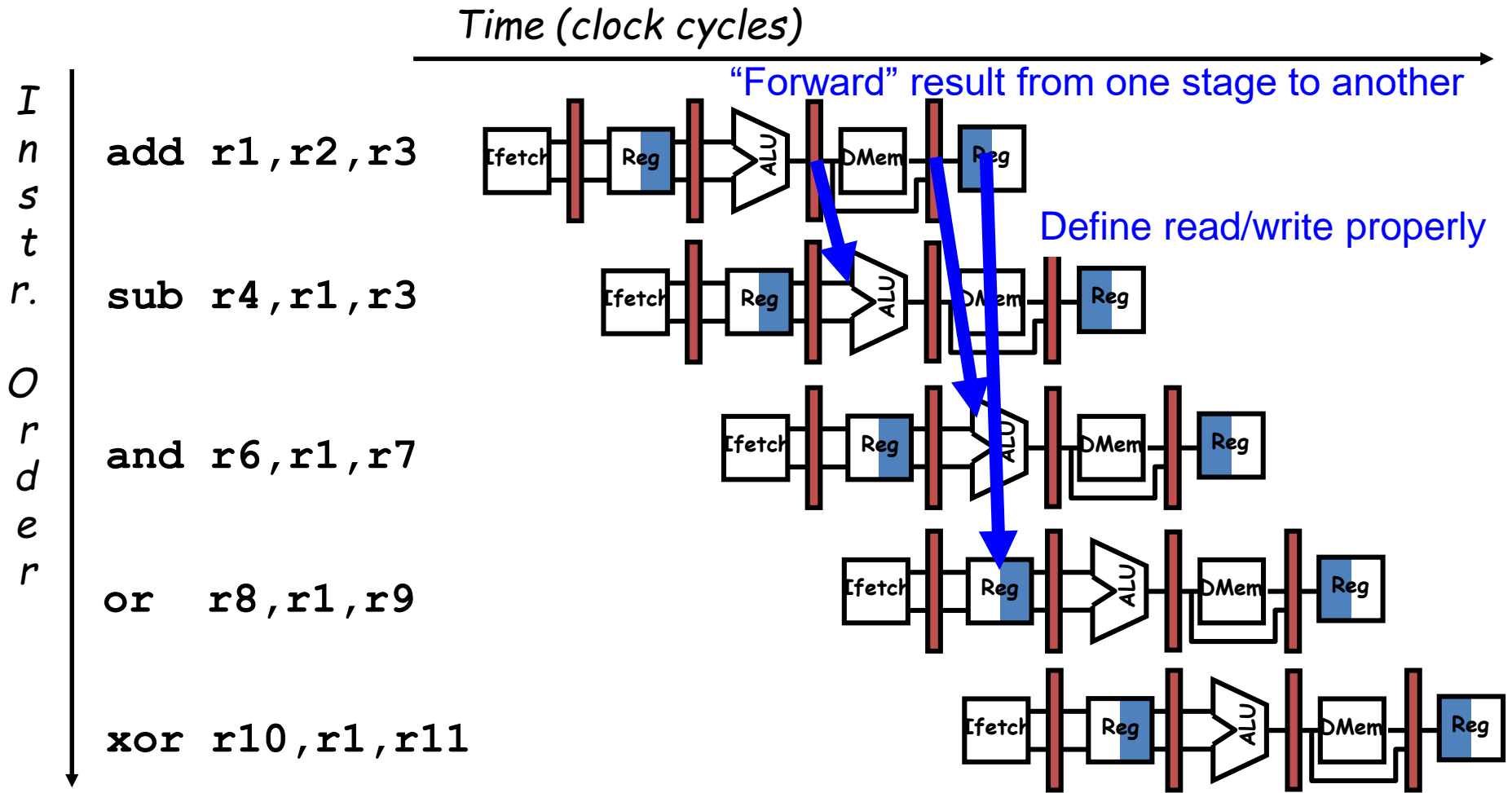
Instr_j writes operand before Instr_i writes it.



```
I: sub r1, r4, r3  
J: add r1, r2, r3  
K: mul r6, r1, r7
```

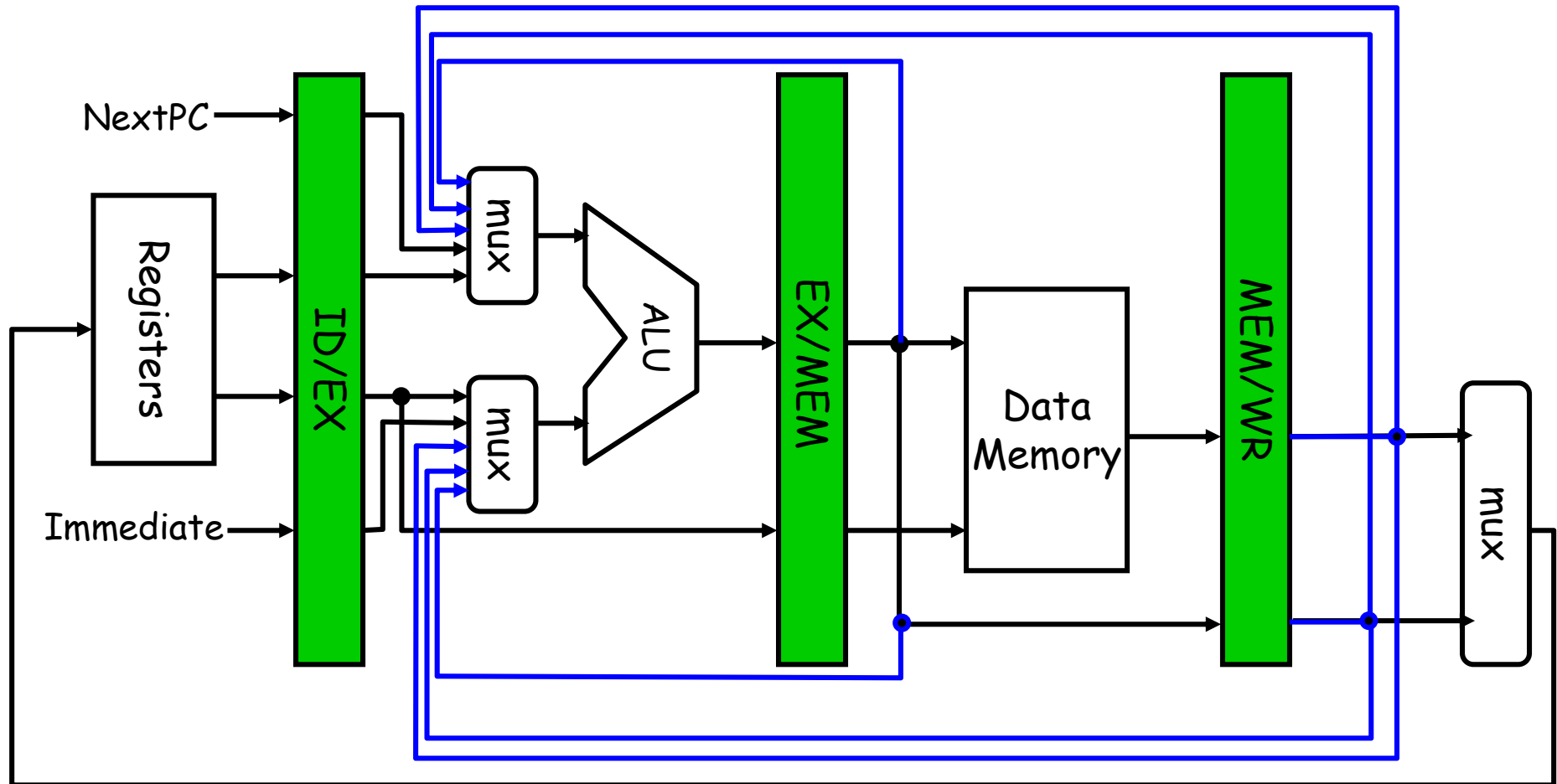
- Called an “output dependence” by compiler writers
- This also results from the reuse of name “r1”.
- Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and Writes are always in stage 5
- Will see WAR and WAW in more complicated pipes and OOO processor

Forwarding to Avoid Data Hazard

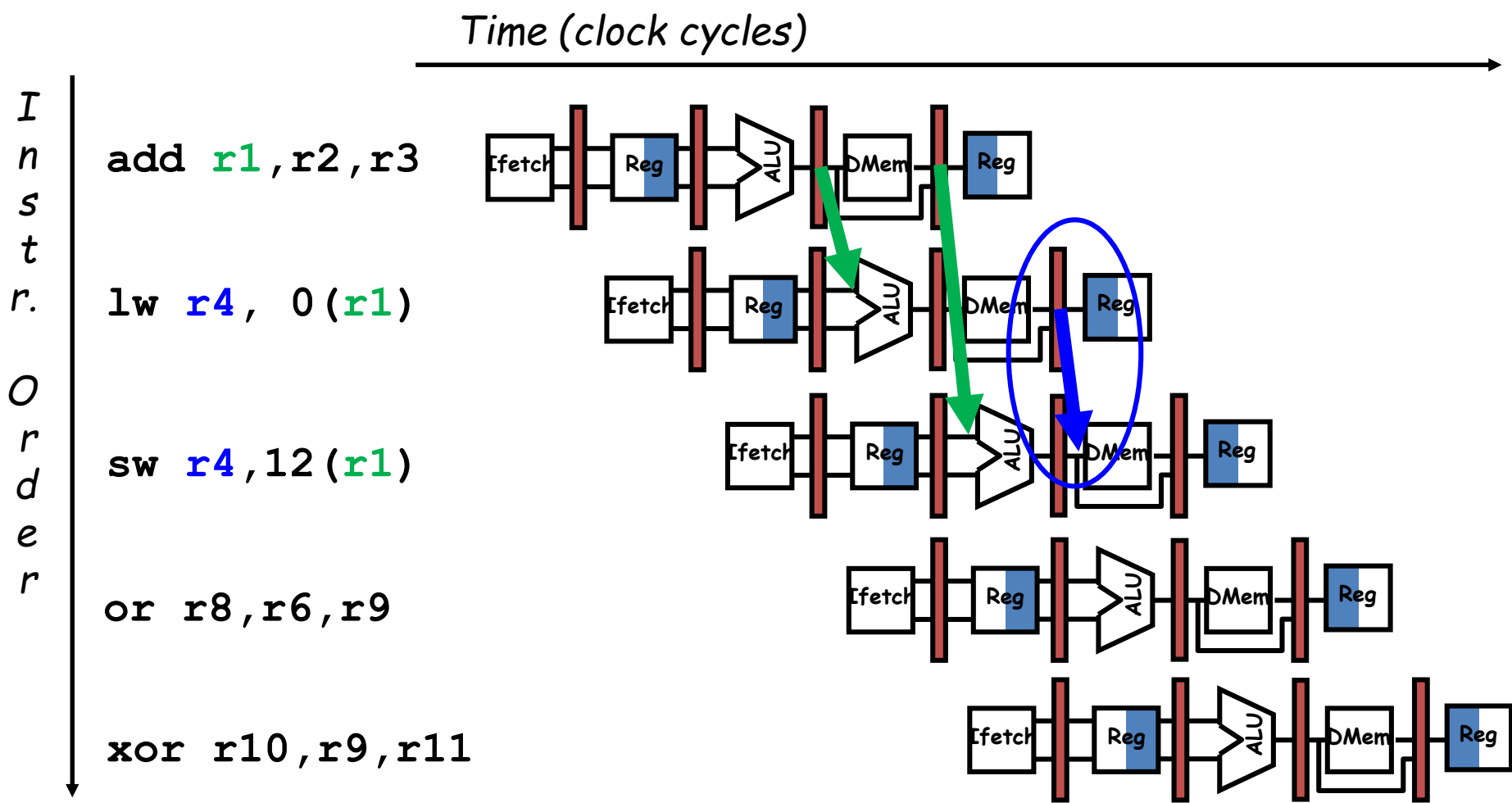


HW Change for Forwarding

Additional hardware is required.



Forwarding to Avoid LW-SW Data Hazard



Data Hazard Even with Forwarding

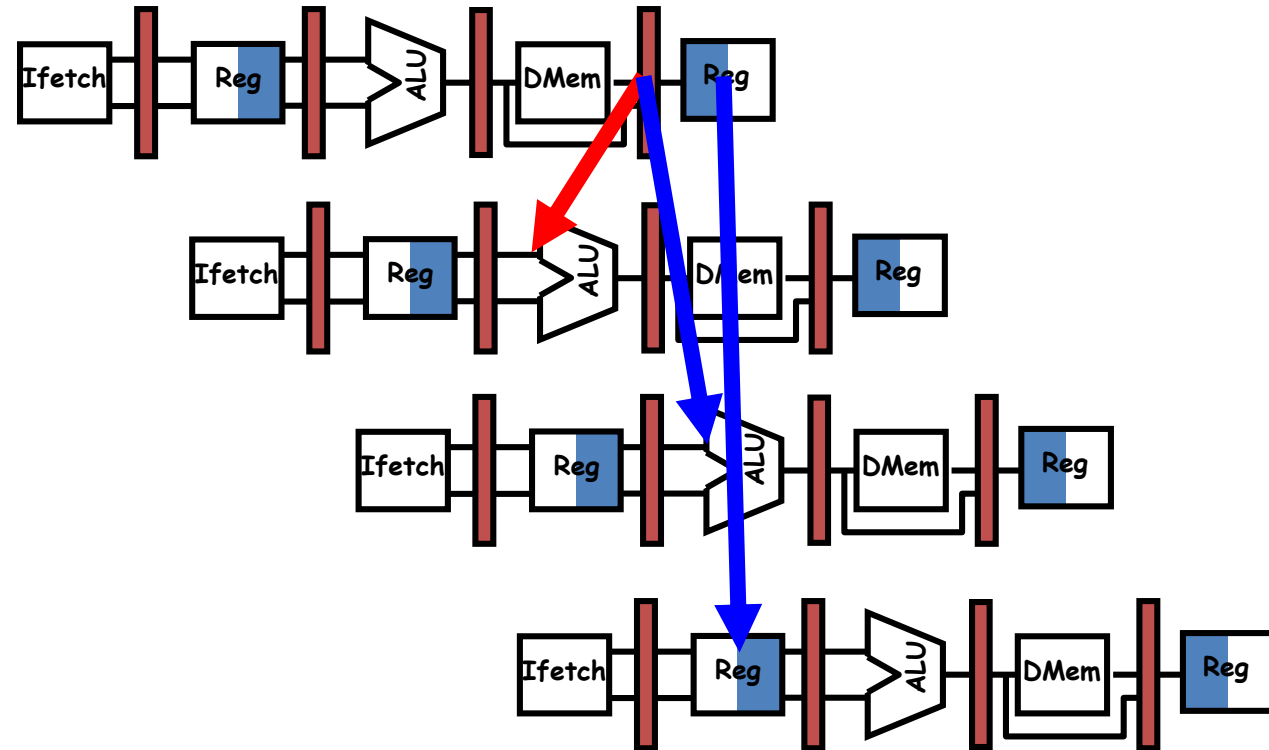
Time (clock cycles)



I
n
s
t
r.

O
r
d
e
r

```
lw r1, 0(r2)
sub r4, r1, r6
and r6, r1, r7
or r8, r1, r9
```



Must delay/stall instruction dependent on loads: Load stall cycles

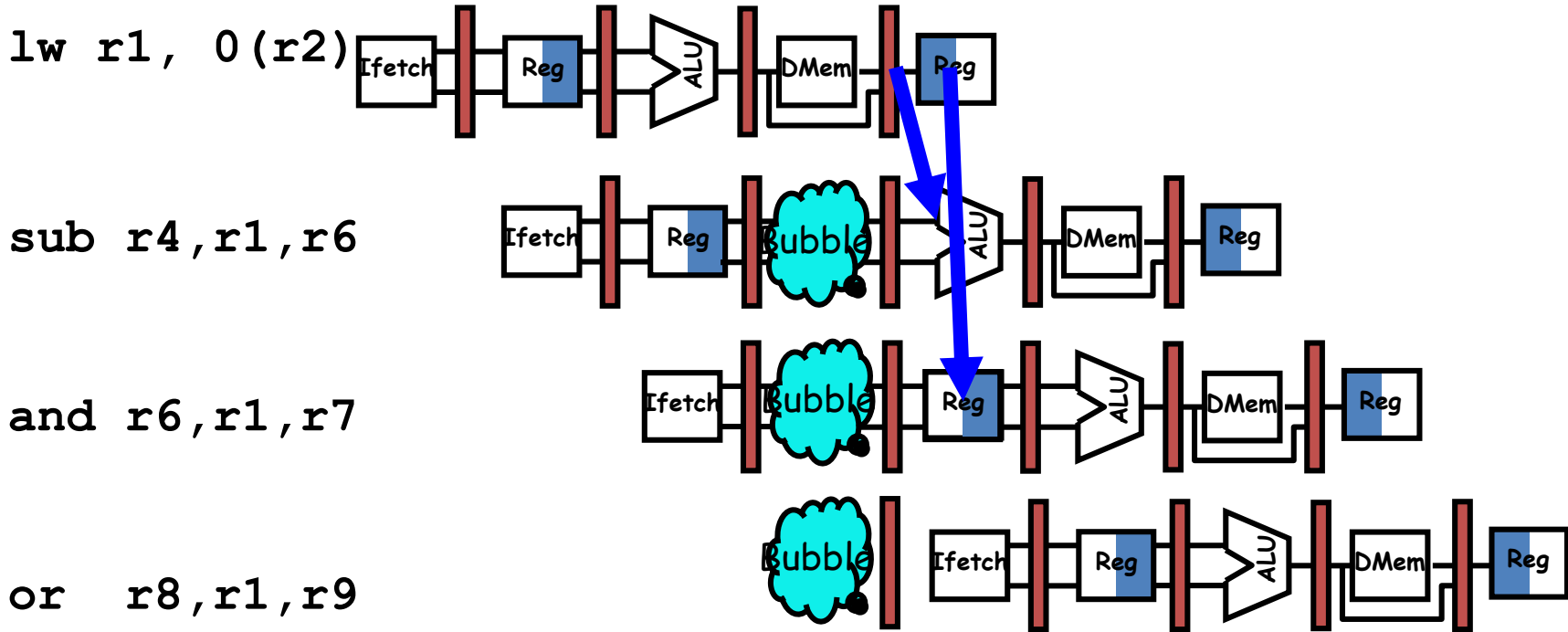
Data Hazard Even with Forwarding

Time (clock cycles)



I
n
s
t
r.

O
r
d
e
r



Software Scheduling to Avoid Load Hazards

Try producing fast code for

$$a = b + c;$$

$$d = e - f;$$

assuming a, b, c, d, e, and f in memory.

Slow code:

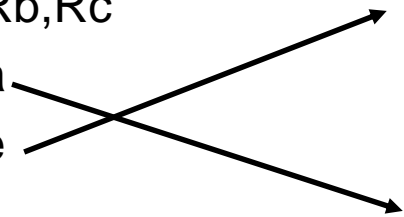
```

LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
    
```

Fast code:

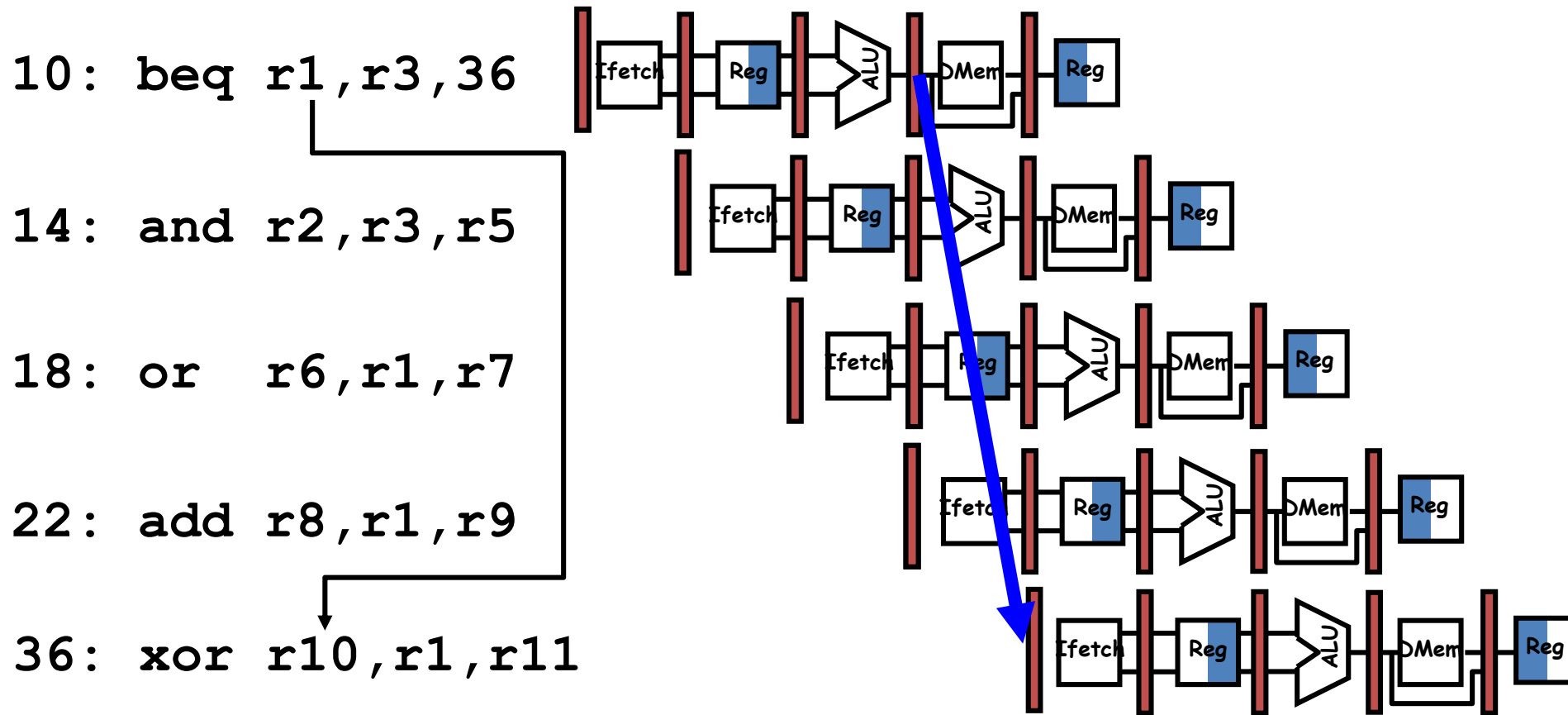
```

LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
    
```



Compiler optimizes for performance. Hardware checks for safety.

Control Hazard on Branches



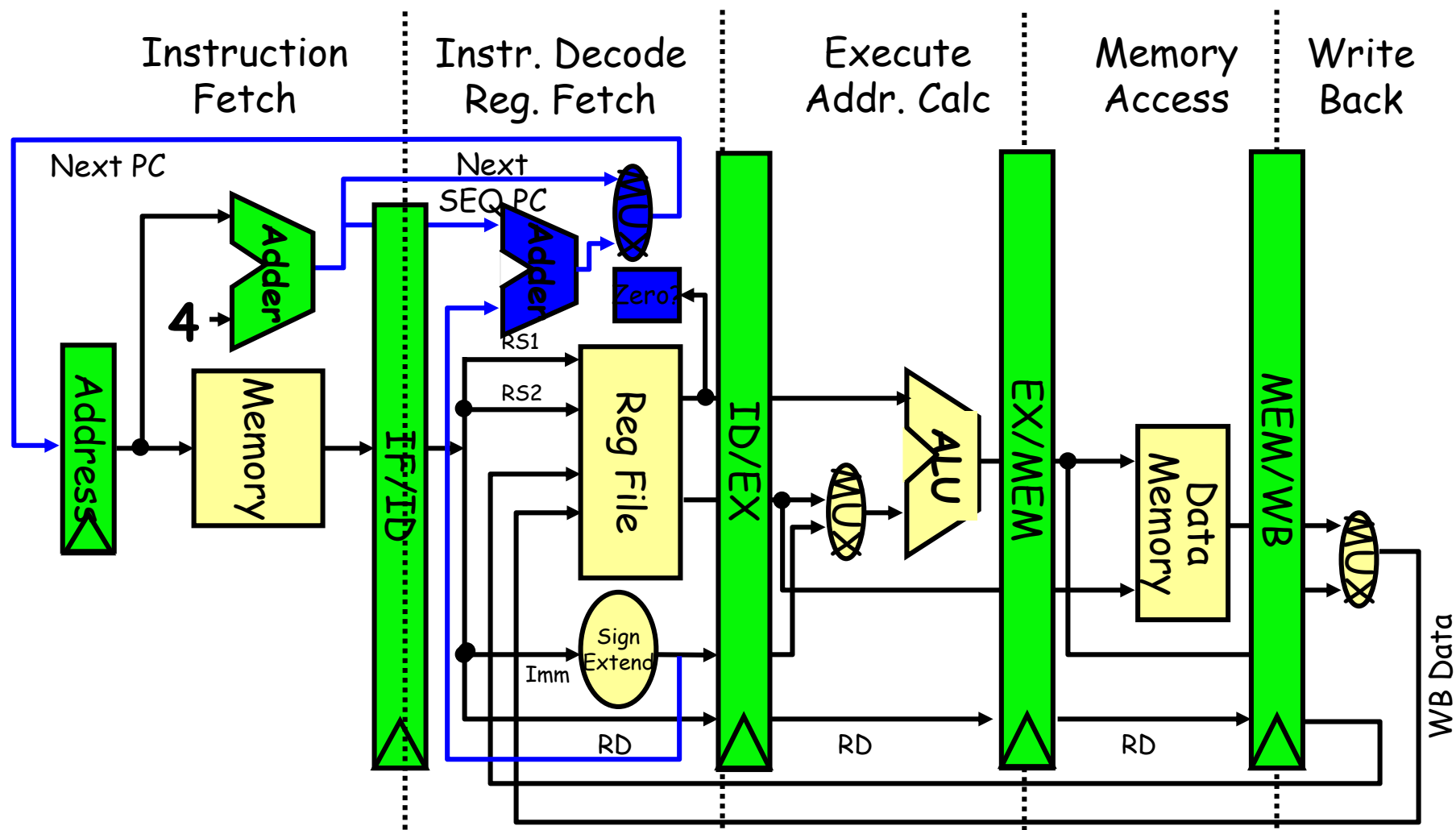
What do you do with the 3 instructions in between?

The simplest solution is to stall the pipeline as soon as a branch instruction is detected

Branch Stall Impact

- If $CPI = 1$, 30% branch,
Stall 3 cycles \Rightarrow new $CPI = 1.9!$
- Two-part solution:
 - Determine branch taken or not sooner, AND
 - Compute taken branch address earlier
- MIPS branch tests if register = 0 or $\neq 0$
- MIPS Solution:
 - Move Zero test to ID/RF stage
 - Adder to calculate new PC in ID/RF stage
 - 1 clock cycle penalty for branch versus 3

Pipelined MIPS Datapath



• Interplay of instruction set design and cycle time.

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- “Squash” instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
 - MIPS still incurs 1 cycle branch penalty
 - Other machines: branch target known before outcome
- What happens when hit not-taken branch?

Four Branch Hazard Alternatives

#4: Delayed Branch – make the stall cycle useful

- Define branch to take place **AFTER** a following instruction

branch instruction

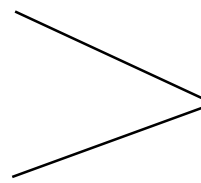
sequential successor₁

sequential successor₂

.....

sequential successor_n

branch target if taken



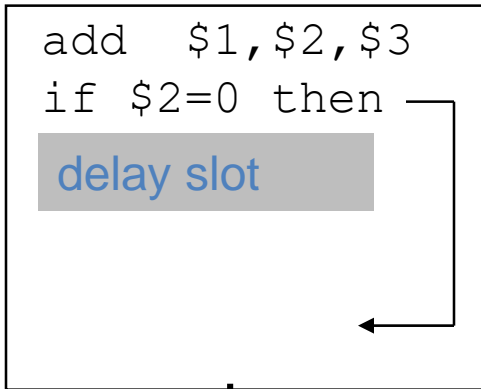
Branch delay of length n

These insts. are executed !!

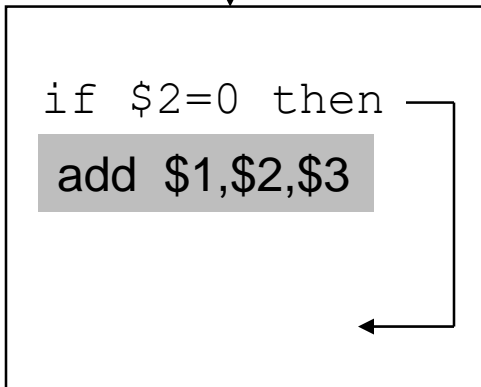
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Scheduling Branch Delay Slots

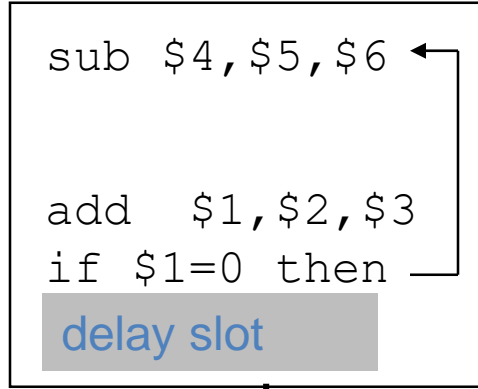
A. From before branch



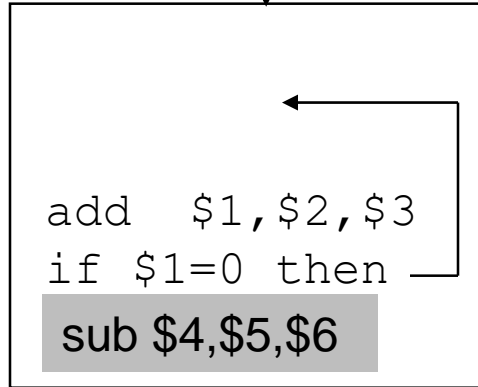
becomes



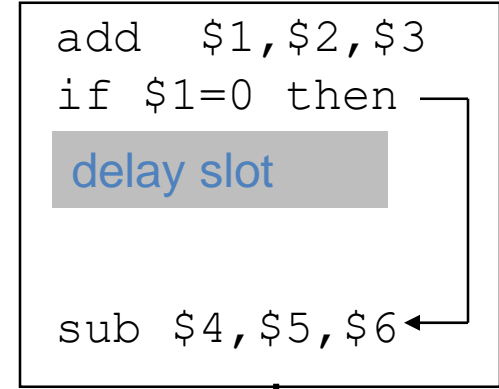
B. From branch target



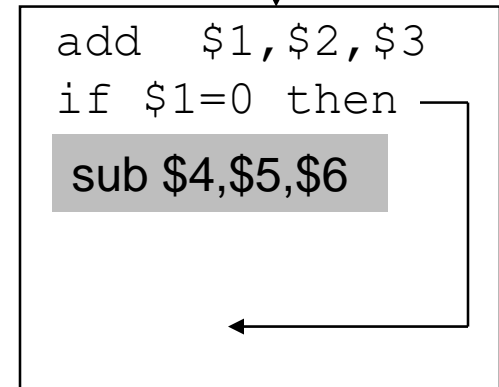
becomes



C. From fall through



becomes



- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails



Delay-Branch Scheduling Schemes and Their Requirements

Scheduling Strategy	Requirements	Improve Performance When?
From before	Branch must not depend on the rescheduled instructions	Always
From target	Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions	When branch is taken. May enlarge program if instructions are duplicated
From fall through	Must be OK to execute instructions if branch is taken	When branch is not taken.

Delayed Branch

- Compiler effectiveness for single branch delay slot:
 - Fills about 60% of branch delay slots
 - About 80% of instructions executed in branch delay slots useful in computation
 - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slots
 - Delayed branch (a static way) has lost popularity compared to more **expensive but more flexible dynamic approaches**
 - Growth in available transistors has made dynamic approaches relatively cheaper

Performance of Branch Schemes Example

- For a deeper pipeline, such as that in a MIPS R4000 and later RISC processors, it takes at least three pipeline stages before the branch-target address is known and an additional cycle before the branch condition is evaluated, assuming no stalls on the registers in the conditional comparison.
- A three-stage delay leads to the branch penalties for the three simplest prediction schemes listed below:

Scheme	Penalty _{Uncond}	Penalty _{untaken}	Penalty _{taken}
Stall	2	3	3
Predicted taken	2	3	2
Predicted not taken	2	0	3

Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Assume: 4% unconditional branch, 6% conditional branch- untaken, 10% conditional branch-taken

<i>Scheduling scheme</i>	<i>Branch penalty</i>	<i>CPI</i>	<i>speedup v. unpipelined</i>	<i>speedup v. stall</i>
Stall pipeline	$2 \times 0.04 + 3 \times 0.06 + 3 \times 0.1$	1.56	n/1.56	1.0
Predict taken	$2 \times 0.04 + 3 \times 0.06 + 2 \times 0.1$	1.46	n/1.46	1.07
Predict ubtaken	$2 \times 0.04 + 0 \times 0.06 + 3 \times 0.1$	1.38	n/1.38	1.13
Delayed branch	$0.5 \times (0.04 + 0.16)$	1.10	n/1.1	1.42

Problems with Pipelining

- **Exception:** An unusual event happens to an instruction during its execution
 - Examples: divide by zero, undefined opcode
- **Interrupt:** Hardware signal to switch the processor to a new instruction stream
 - Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)
- **Problem:** It must appear that the exception or interrupt must appear between 2 instructions (I_i and I_{i+1})
 - The effect of all instructions up to and including I_i is totalling complete
 - No effect of any instruction after I_i can take place
- The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}

Reading Sections

- **C.3 How Is Pipelining Implemented?**
- **C.4 What Makes Pipelining Hard to Implement?**
- **C.5 Extending the RISC V Integer Pipeline to Handle Multicycle Operations**