

نمودج رقم (2)

COURSE TITLE

Course Code	:30102315
-------------	-----------

Credit Hours	:3
--------------	----

Prerequisite	:30102214
--------------	-----------



Instructor Information

Name	: Dr. Rushdi Saleem Abu Zneit				
Office No.	18, building 17 floor 3				
Tel (Ext)	Contact Telephone(Department Telephone)				
E-mail	dr.rushdizneit@bau.edu.jo				
Office Hours	Online				
Class Times	Building	Day	Start Time	End Time	Room No.
		Sun,Tue, Thu	10:00	11:00	Online



Course description {From course plan}

Course Title; Computer Architecture
Credit Hour(3)

[Pre-req. 30102214

Textbook: Textbook Title

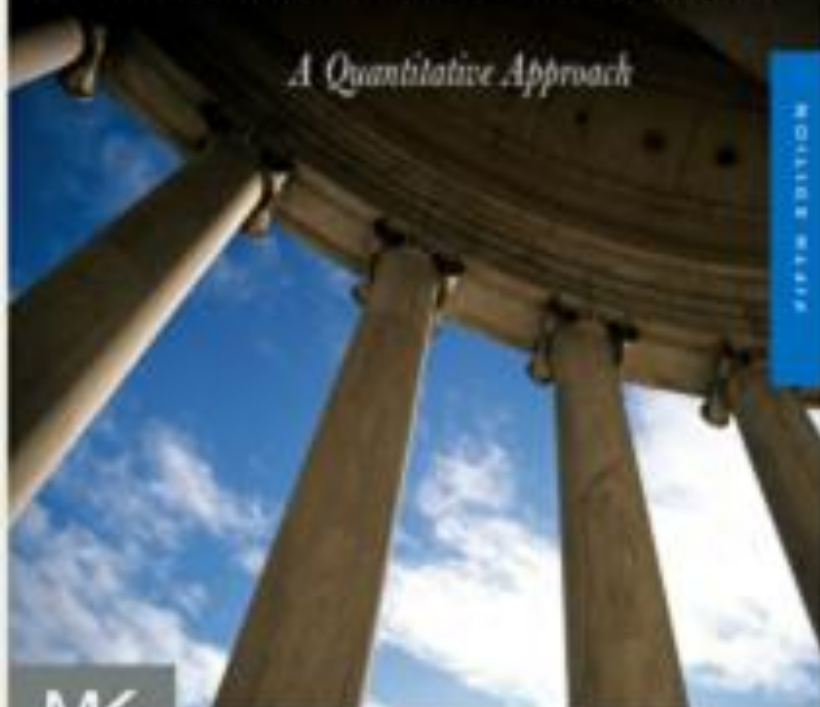
JOHN L. HENNESSY DAVID A. PATTERSON

COMPUTER ARCHITECTURE

A Quantitative Approach

FOURTH EDITION

MK
Morgan Kaufmann





COURSE OBJECTIVES

The module aims to provide students with a fundamental knowledge of computer hardware and computer systems, with an emphasis on system design and performance. The module concentrates on the principles underlying systems organization, issues in computer system design, and contrasting implementations of modern systems. The module is central to the aims of the Computing Systems degree course, for which it is core.

COURSE SYLLABUS

Week	Course Topic
Week 1	Course overview & Introduction
Week 2	Performance, Benchmarks, Measurements
Week 3	Pipelining Review
Week 4	Review: Instruction Set Design & Memory Hierarchy Design
Week 5	Memory Hierarchy Design
Week 6	Pipeline hazards, Instruction Re-scheduling
Week 7	ILP - Static: Loop Unrolling
Week 8	Midterm Exam
Week 9	Introduction to Branch Predictors
Week 10	Dynamic Instruction Level Parallelism: Scoreboarding Technique
Week 11	Dynamic Instruction Level Parallelism: Tomoslus' technique
Week 12	Data-Level Parallelism in Vector, SIMD, and GPU Architectures
Week 13	Data-Level Parallelism in Vector, SIMD, and GPU Architectures
Week 14	Thread-Level Parallelism
Week 15	Warehouse-Scale Computers to Exploit Request-Level and Data-Level Parallelism
Week 16	Final Exam

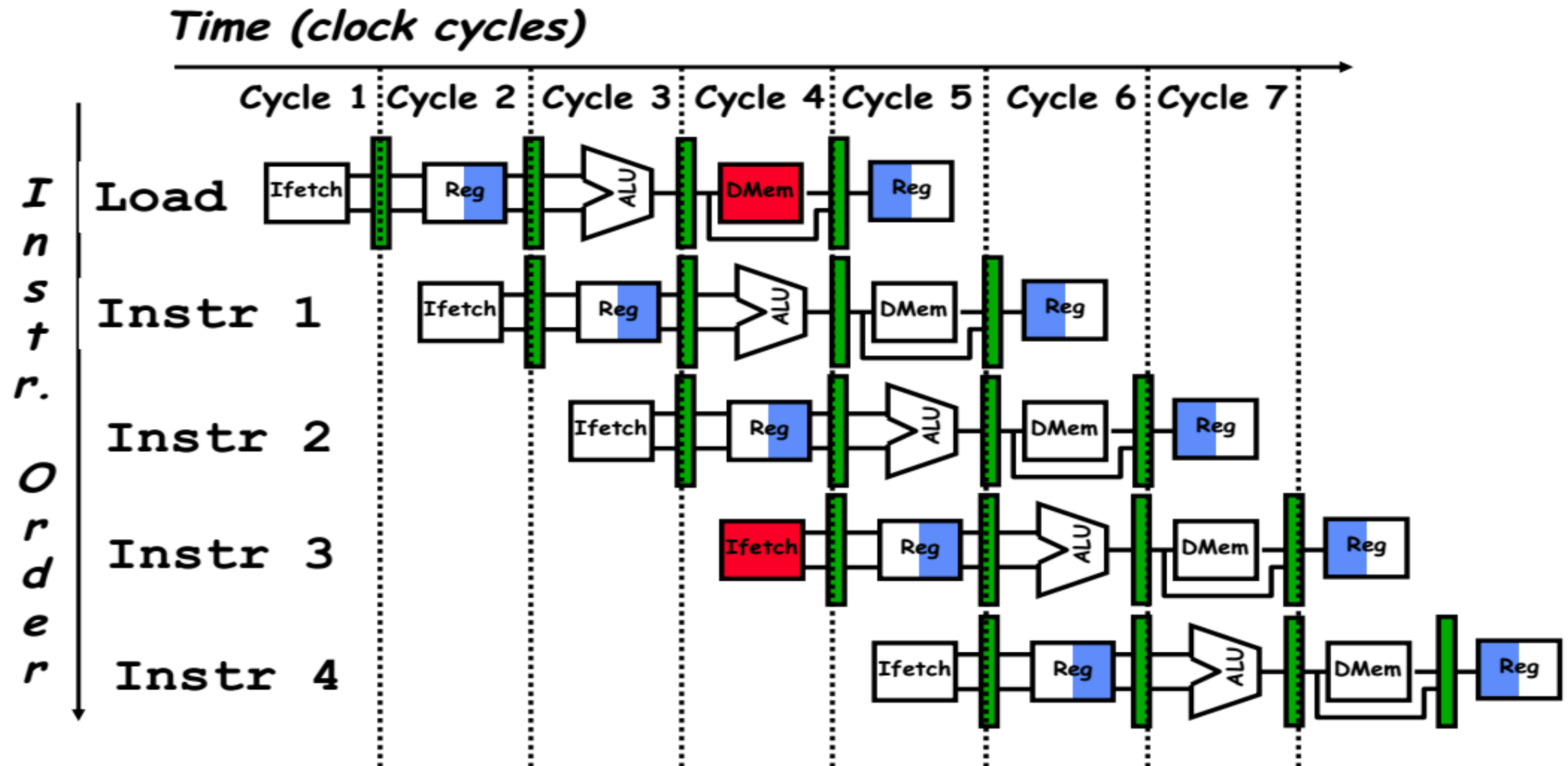
Course Topic
Introduction
Performance, Benchmarks, Measurements
Review: pipelining
Review: Instruction Set Design
Memory Hierarchy Design
Pipeline hazards, Instruction Re-scheduling
ILP – Static: Loop Unrolling
Midterm Exam
Introduction to Branch Predictors
Dynamic Instruction Level Parallelism: Scoreboarding Technique
Dynamic Instruction Level Parallelism: Tomoslus' technique
Data-Level Parallelism in Vector, SIMD, and GPU Architectures
Thread-Level Parallelism
Warehouse-Scale Computers to Exploit Request-Level and Data-Level Parallelism
Final Exam



Week 4

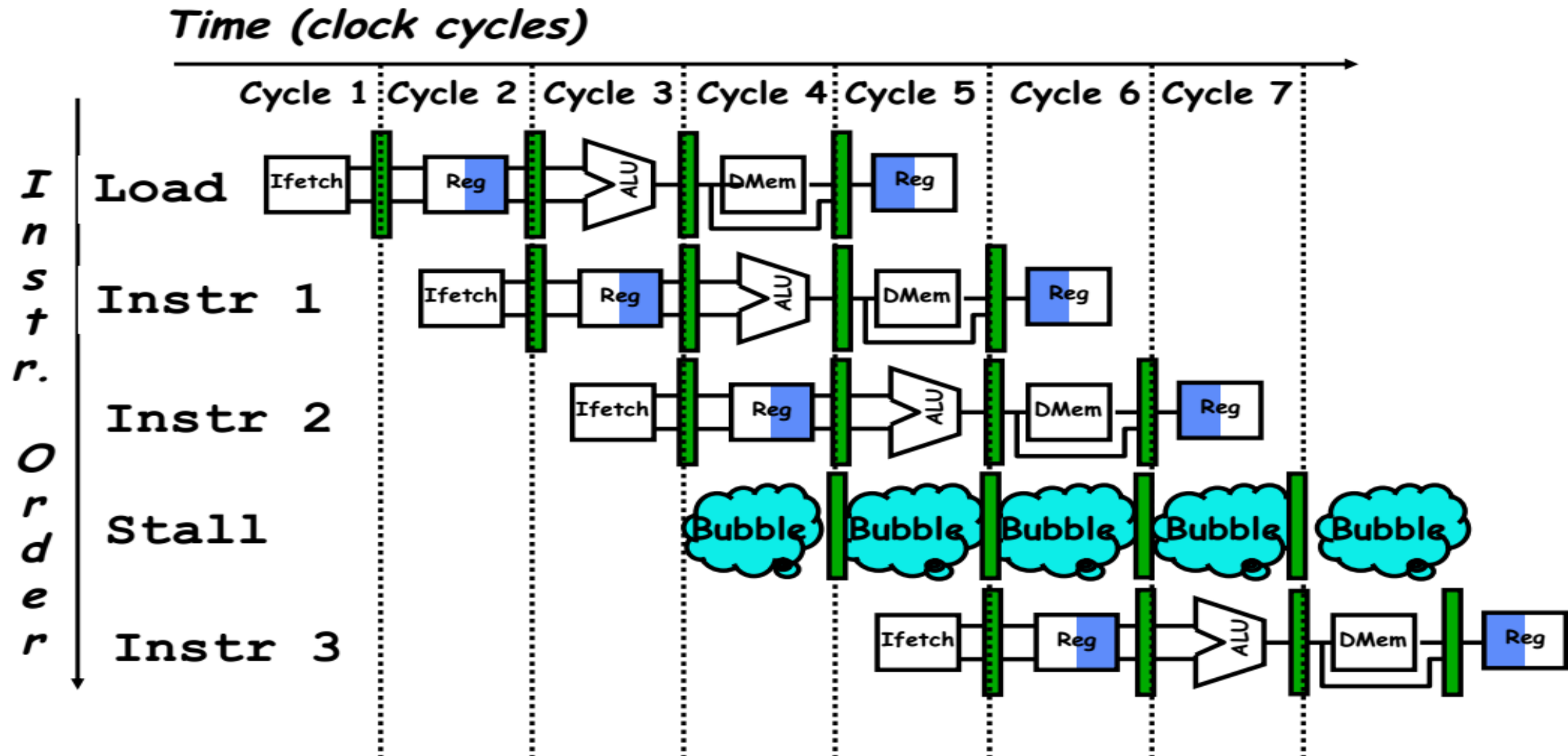
One Memory Port/Structural Hazards

Figure A.4, Page A-14



One Memory Port/Structural Hazards

(Similar to Figure A.5, Page A-15)



How do you “bubble” the pipe?

Speed Up Equation for Pipelining

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

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

For 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
- Loads are 40% of instructions executed

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

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

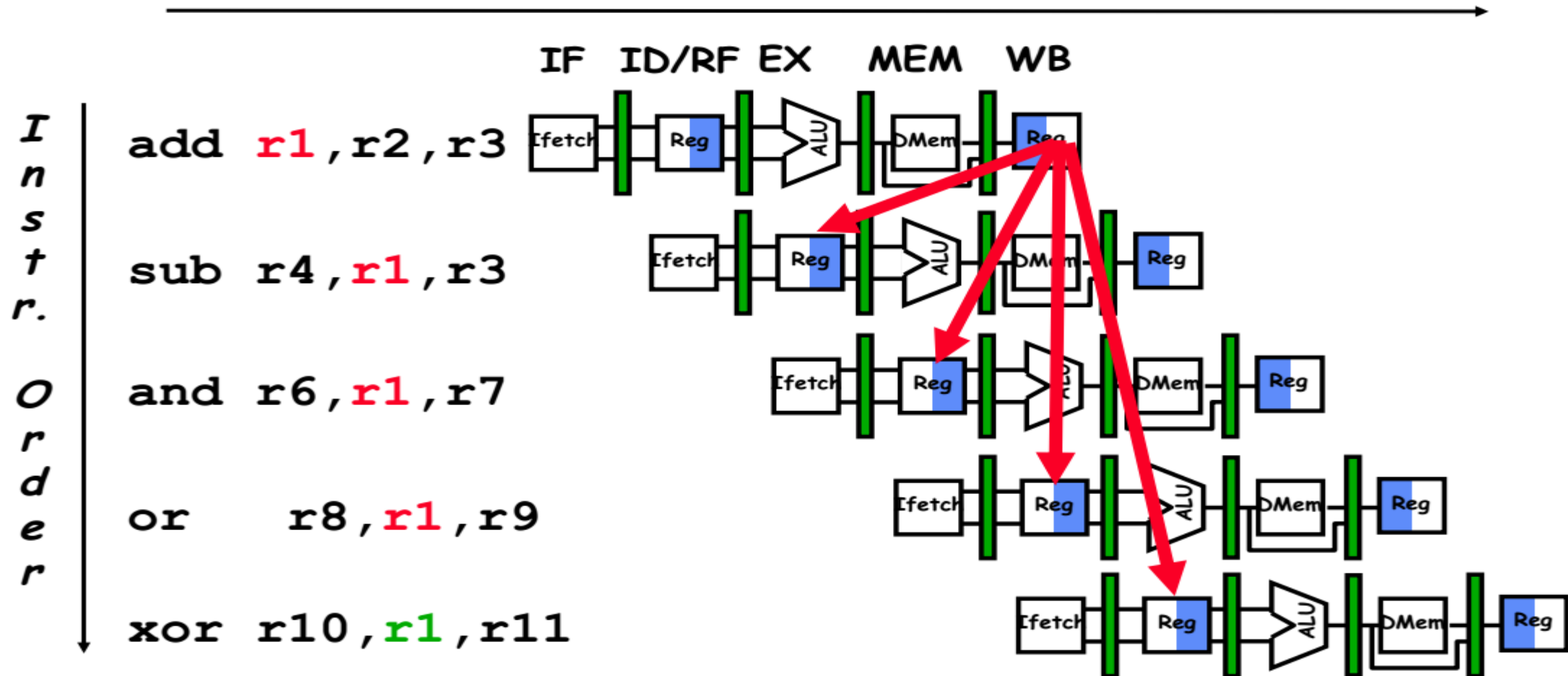
$$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth} / (0.75 \times \text{Pipeline Depth}) = 1.33$$

- Machine A is 1.33 times faster

Data Hazard on R1

Figure A.6, Page A-17


Time (clock cycles)



Three Generic Data Hazards

- **Read After Write (RAW)**

Instr_j tries to read operand before Instr_i writes it


 I: add **r1**, r2, r3
J: sub r4, **r1**, r3

- **Caused by a “Dependence”** (in compiler nomenclature). This hazard results from an actual need for communication.

Three Generic Data Hazards

- **Write After Read (WAR)**

Instr_j writes operand before Instr_i reads it

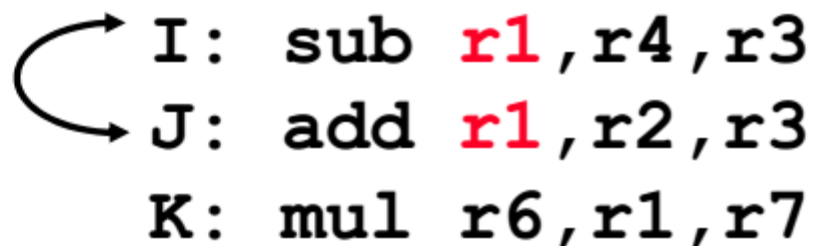
 I: sub r4, **r1**, r3
J: add **r1**, r2, r3
K: mul r6, r1, r7

- 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, and
 - Reads are always in stage 2, and
 - Writes are always in stage 5

Three Generic 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

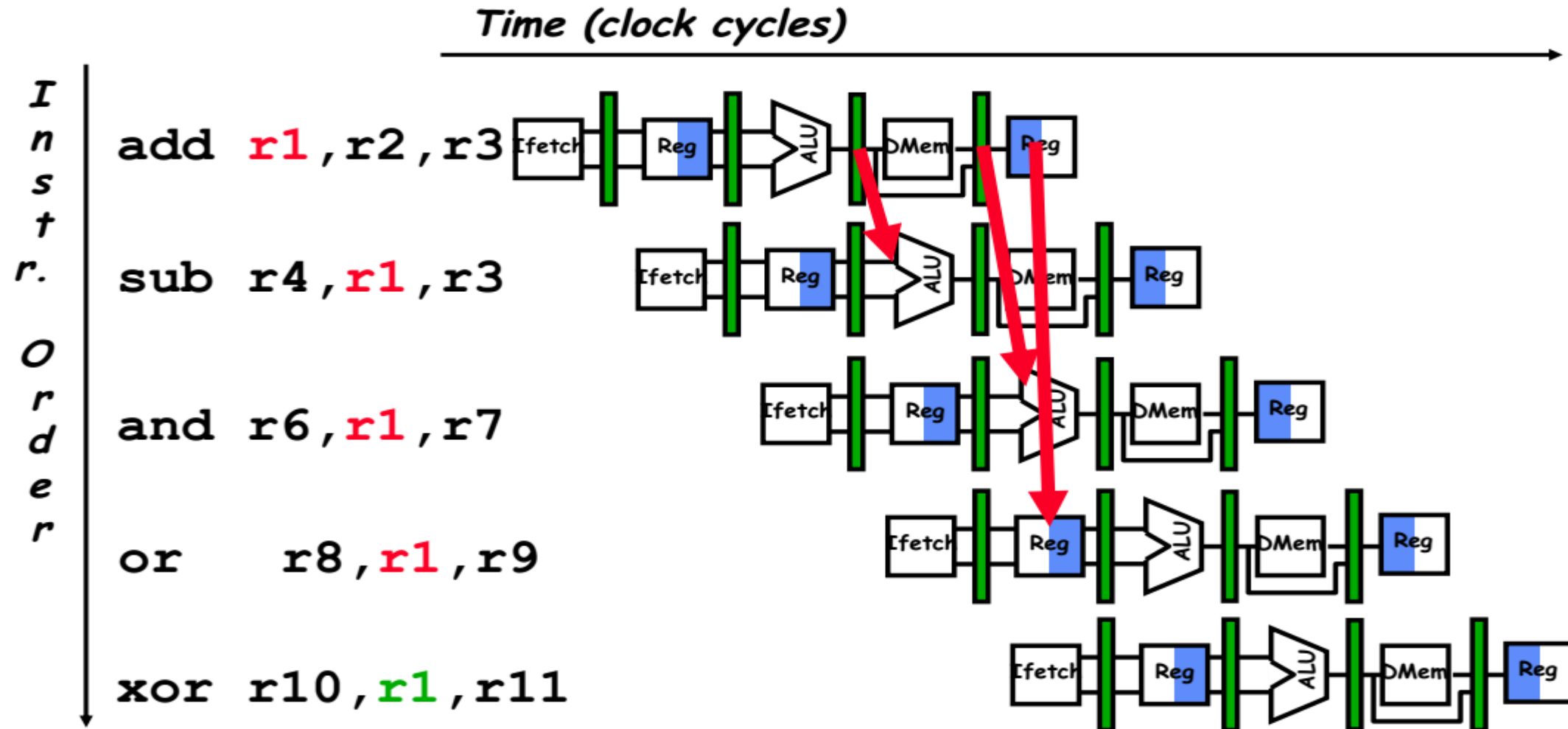
Data Forwarding

- With **data forwarding** (also called **bypassing** or **short-circuiting**), data is transferred back to earlier pipeline stages before it is written into the register file.
 - Instr i: add **r1**,r2,r3 (result ready after EX stage)

 - Instr j: sub r4,**r1**,r5 (result needed in EX stage)
- This either eliminates or reduces the penalty of RAW hazards.
- To support data forwarding, additional hardware is required.
 - Multiplexors to allow data to be transferred back
 - Control logic for the multiplexors

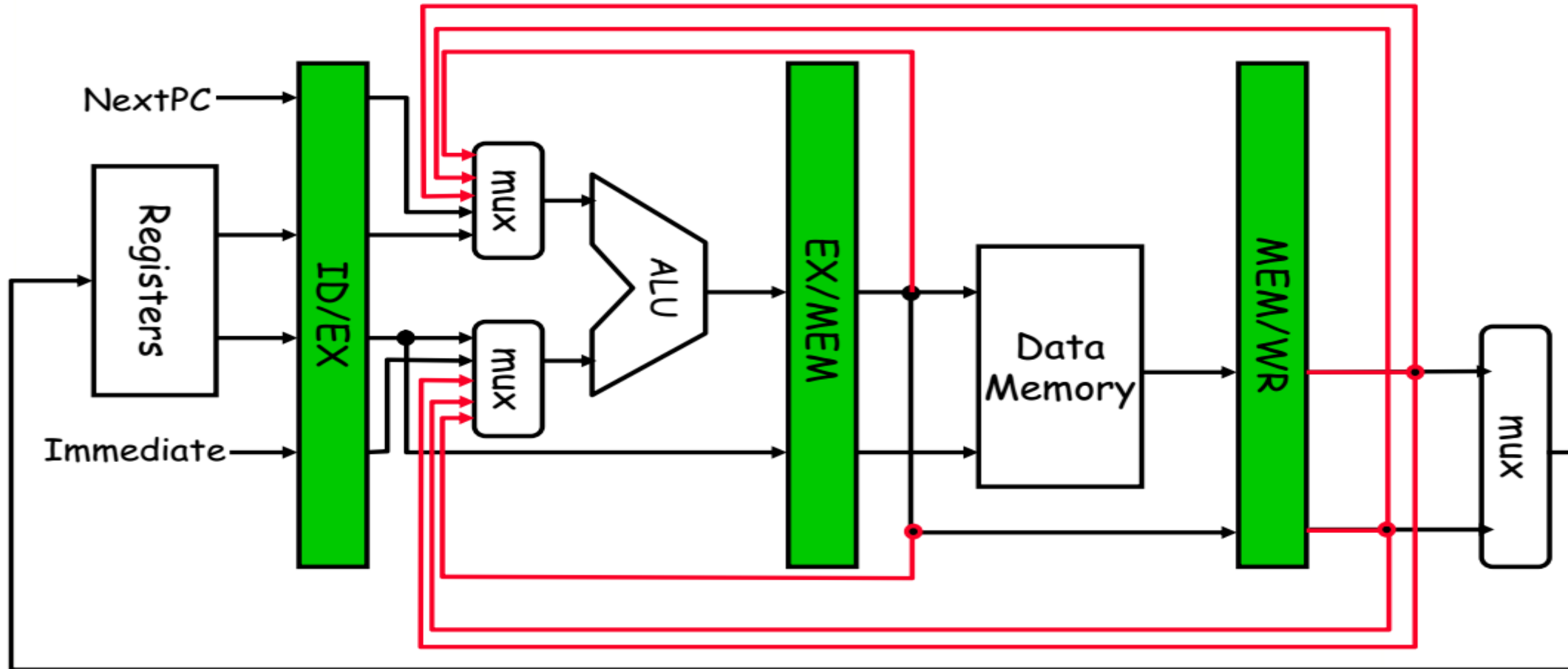
Forwarding to Avoid Data Hazard

Figure A.7, Page A-19



HW Change for Forwarding

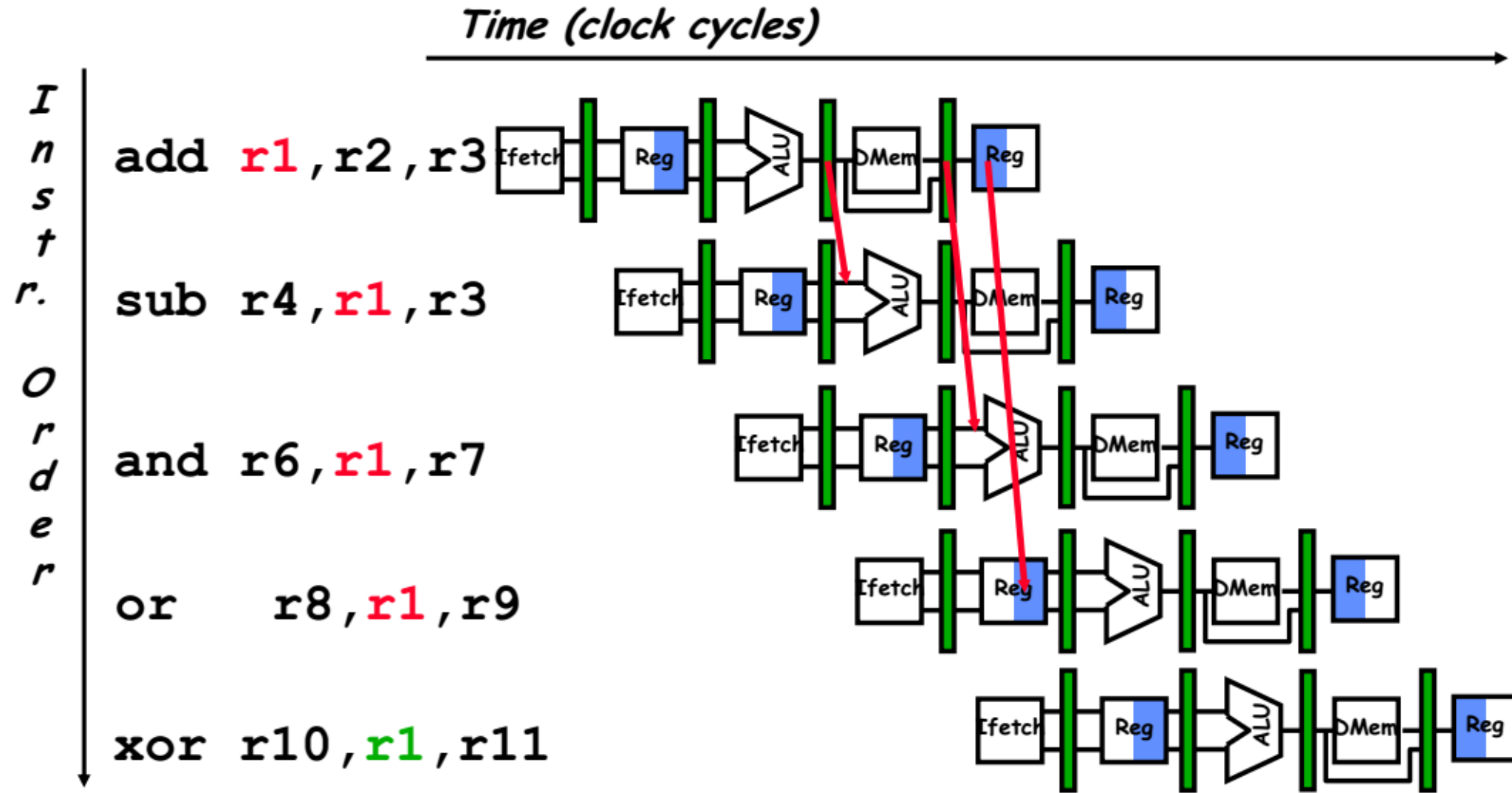
Figure A.23, Page A-37



What circuit detects and resolves this hazard?

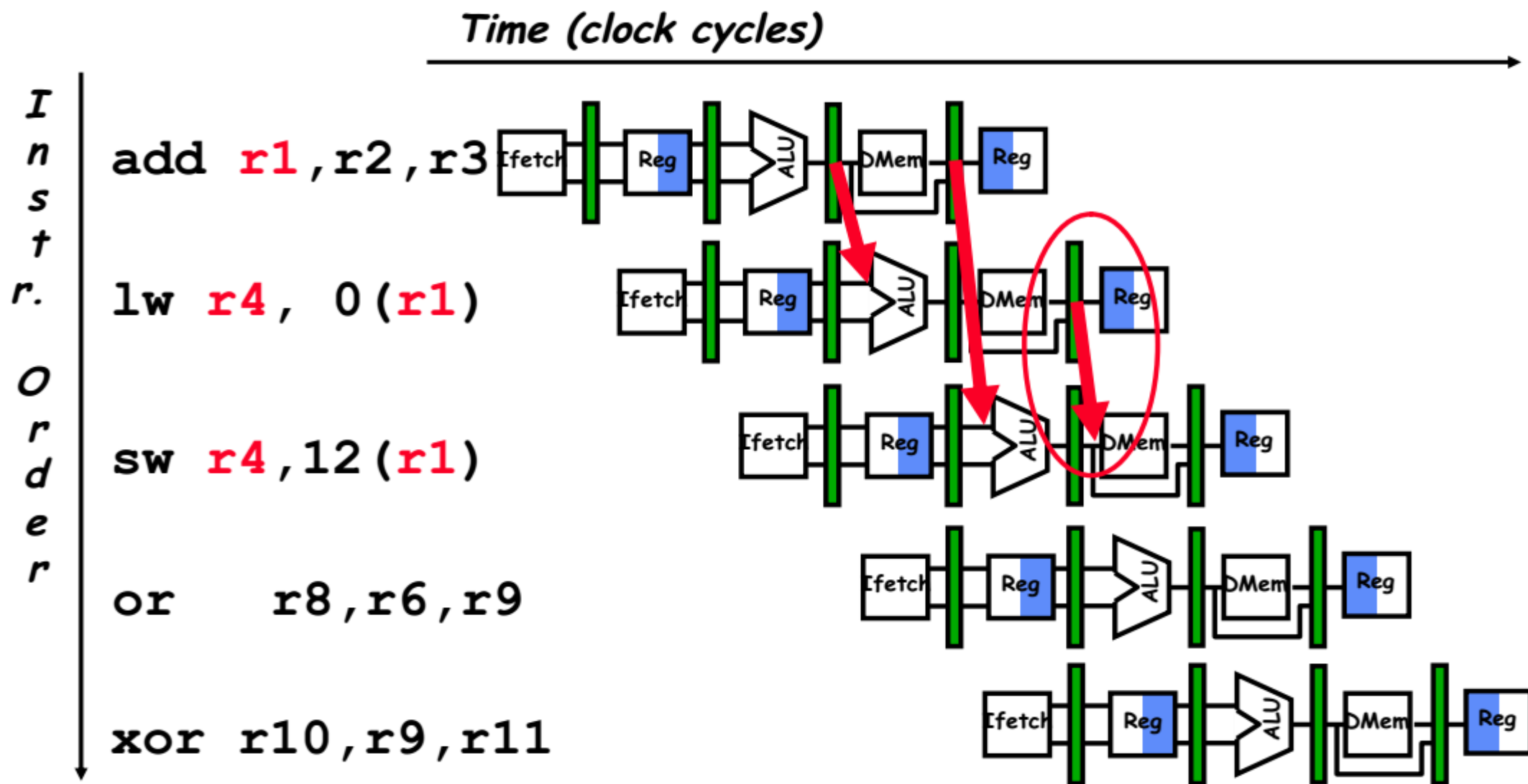
Forwarding to Avoid RAW Hazard

Figure 3.10, Page 149



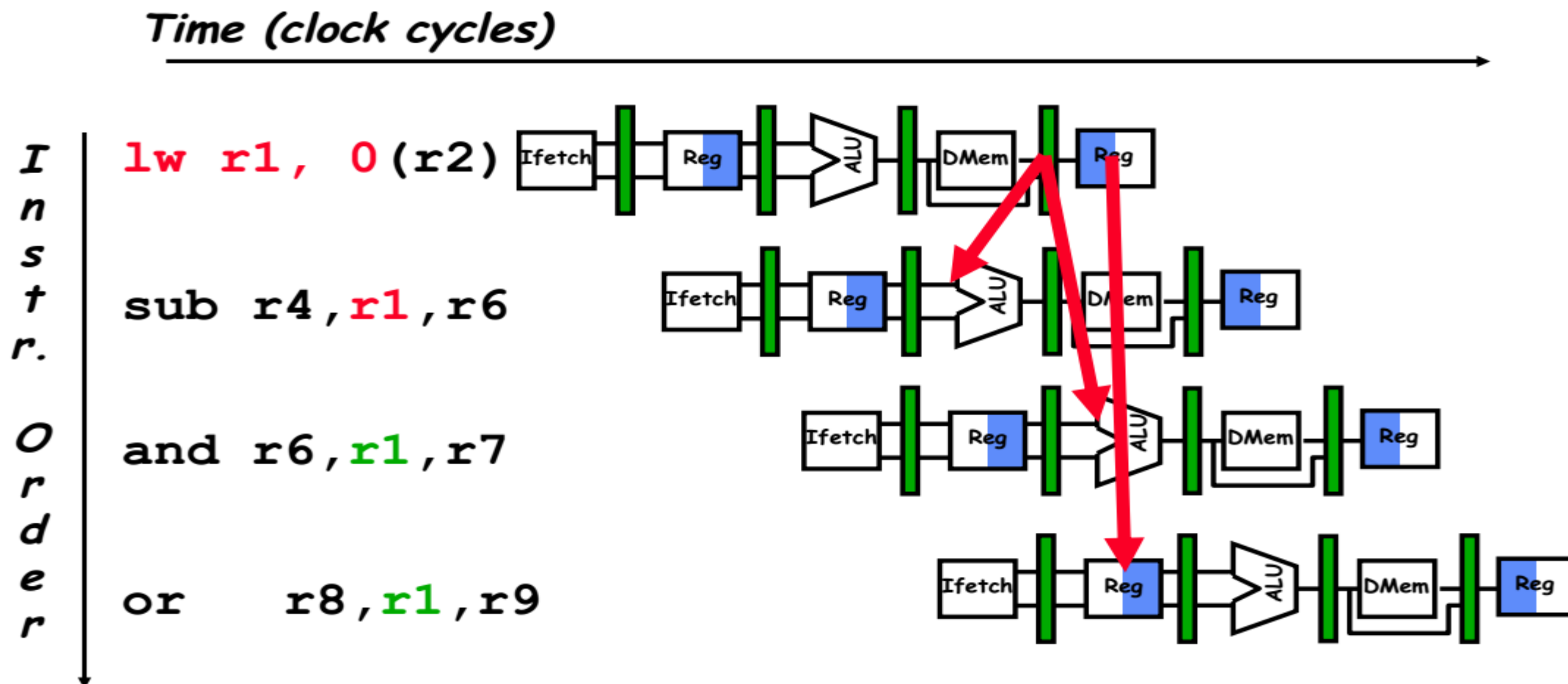
Forwarding to Avoid LW-SW Data Hazard

Figure A.8, Page A-20



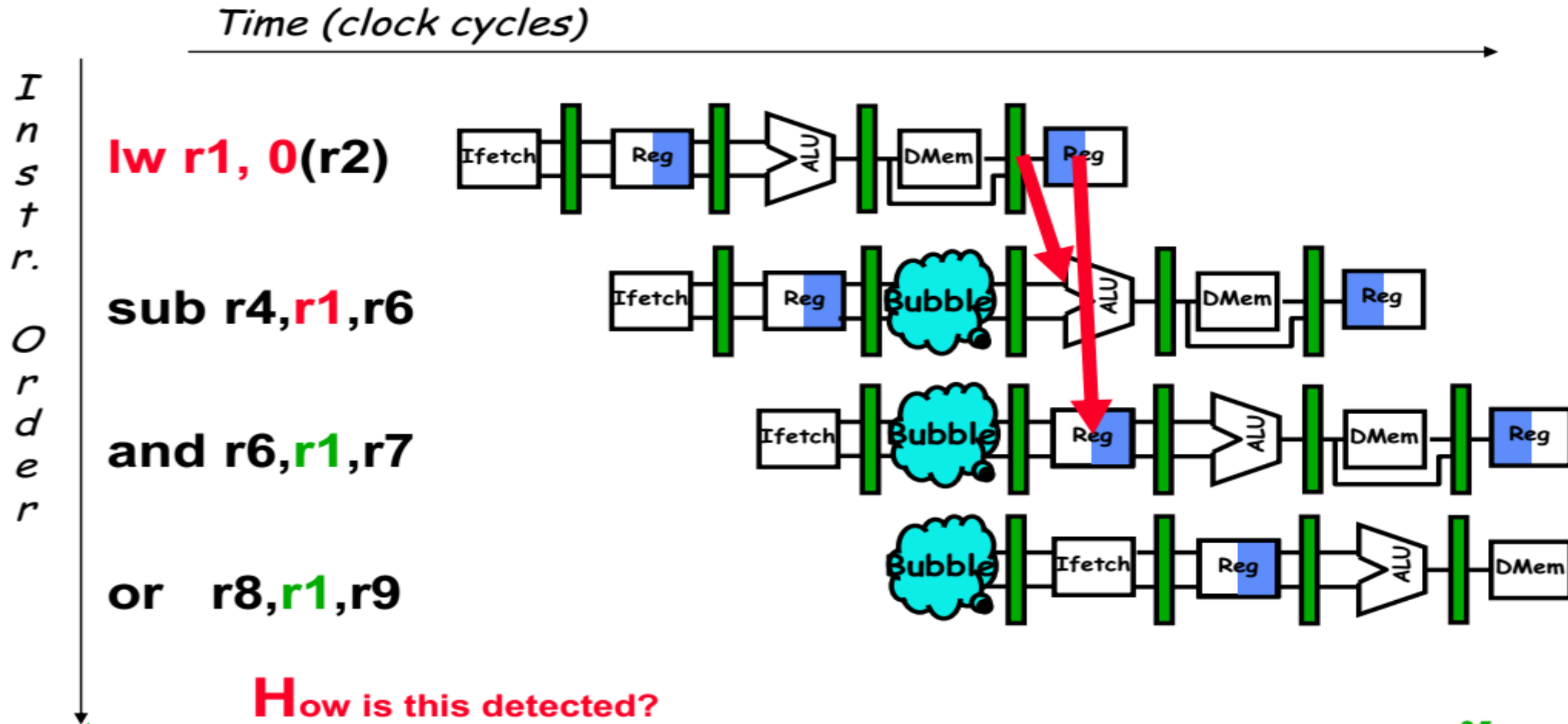
Data Hazard Even with Forwarding

Figure A.9, Page A-21



Data Hazard Even with Forwarding

(Similar to Figure A.10, Page A-21)



Software Scheduling to Avoid Load Hazards

Try producing fast code for

$a = b + c;$

$d = e - f;$

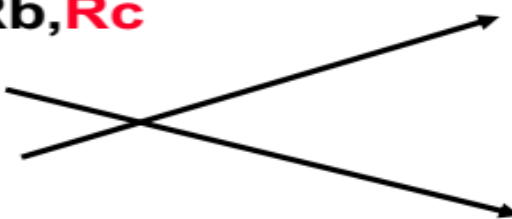
assuming $a, b, c, d, e,$ and f in memory.

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.

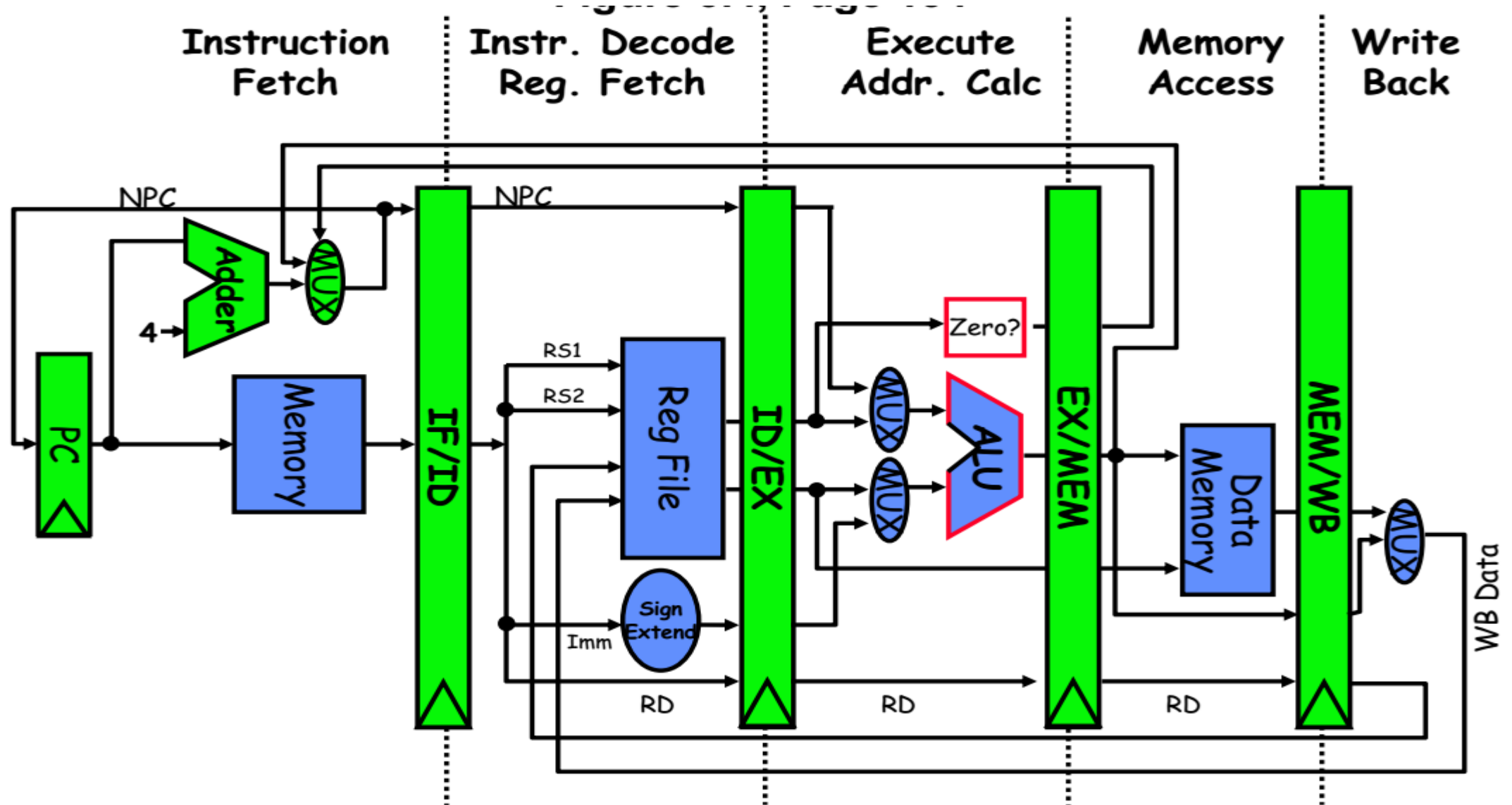
Outline

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- **Branch Schemes**
- **Exceptions and Interrupts**
- **Conclusion**

Control Hazards

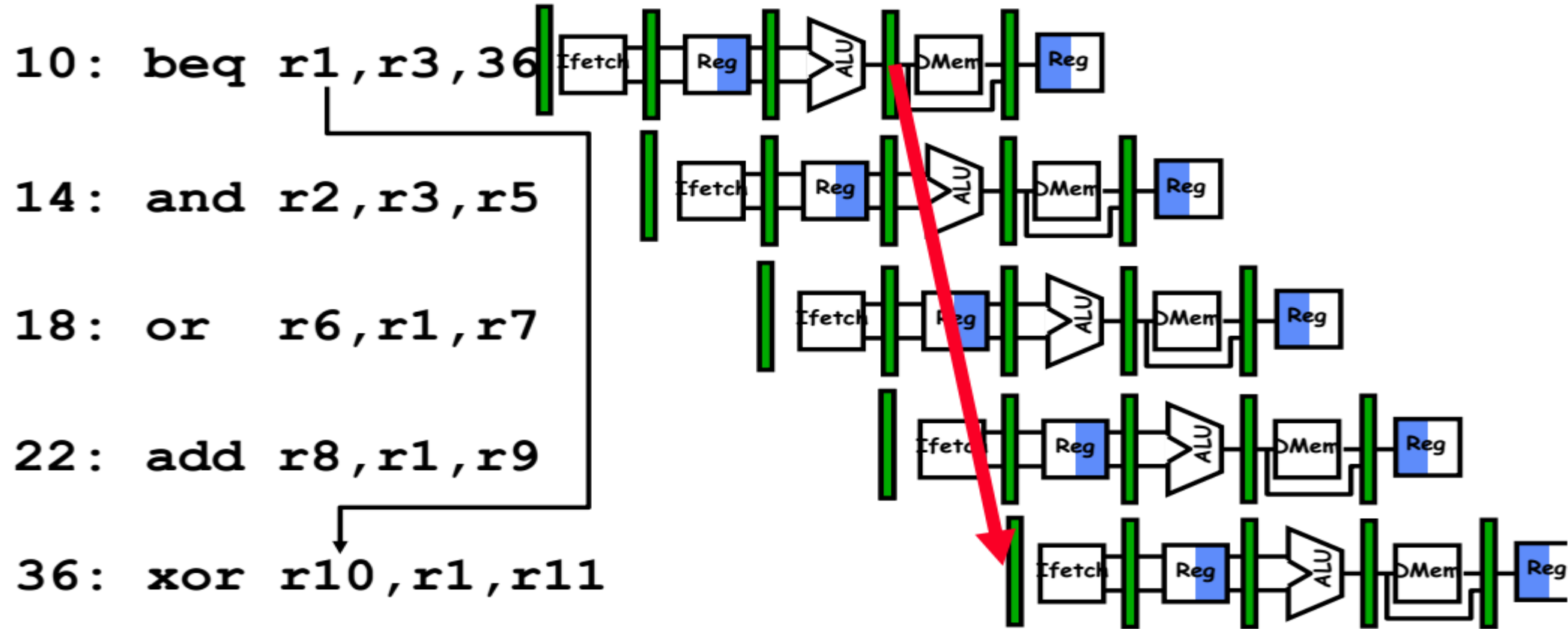
- **Control hazards, which occur due to instructions changing the PC, can result in a large performance loss.**
- **A branch is either**
 - Taken: $PC \leq PC + 4 + Imm$
 - Not Taken: $PC \leq PC + 4$
- **The simplest solution is to stall the pipeline as soon as a branch instruction is detected.**
 - Detect the branch in the ID stage
 - Don't know if the branch is taken until the EX stage
 - If the branch is taken, we need to repeat the IF and ID stages
 - New PC is not changed until the end of the MEM stage, after determining if the branch is taken and computing the new PC value

5 Steps Datapath



Control Hazard on Branches

Three Stage Stall



What do you do with the 3 instructions in between?

How do you do it?

Where is the “commit”?

Control Hazard on Branches

- With our original DLX model, branches have a delay of 3 cycles
- The delay for not-taken branches can be reduced to two cycles, since it is not necessary to fetch the instruction again.

Inst.	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
Branch instr.	IF	ID	EX	MEM	WB				
Branch successor		IF	Stall	Stall	IF	ID	EX	MEM	WB
Branch success+1						IF	ID	EX	MEM WB

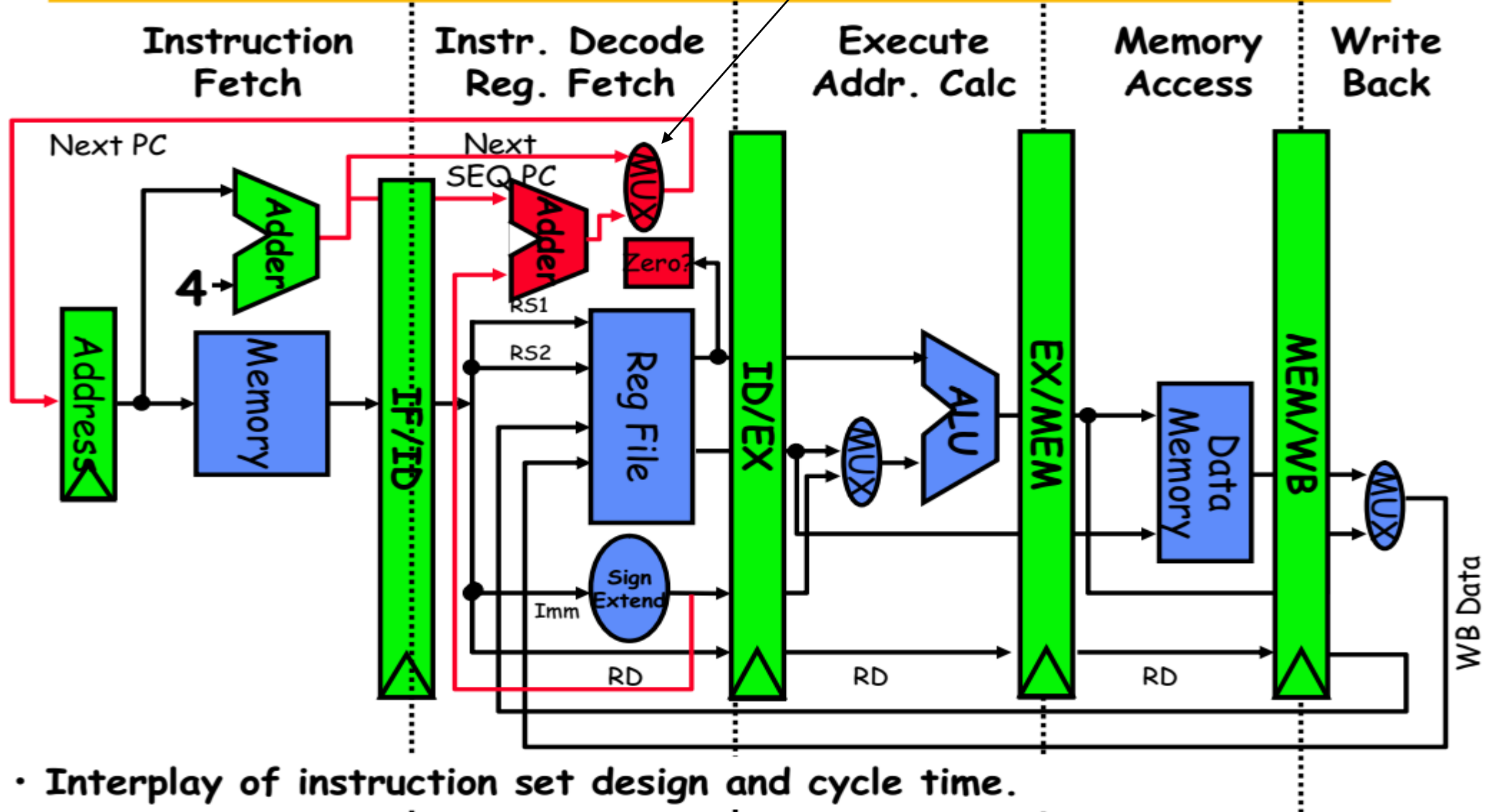
Branch Stall Impact

- 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
 - PC is written at the end of each IF cycle

Pipelined MIPS Datapath

Figure A.24, page A-38

This is the correct 1 cycle latency implementation!



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

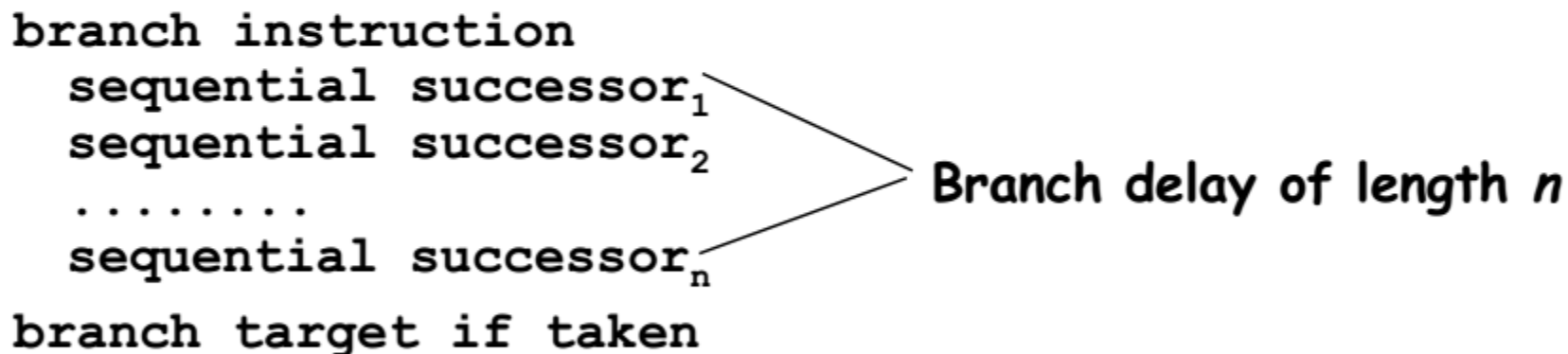
#3: Predict Branch Taken

- 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

Four Branch Hazard Alternatives

#4: Delayed Branch

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

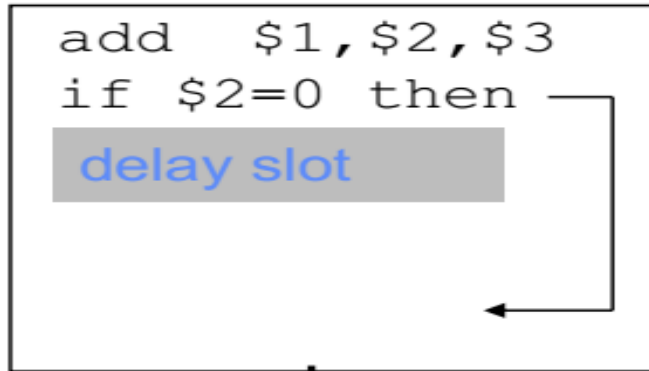


- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

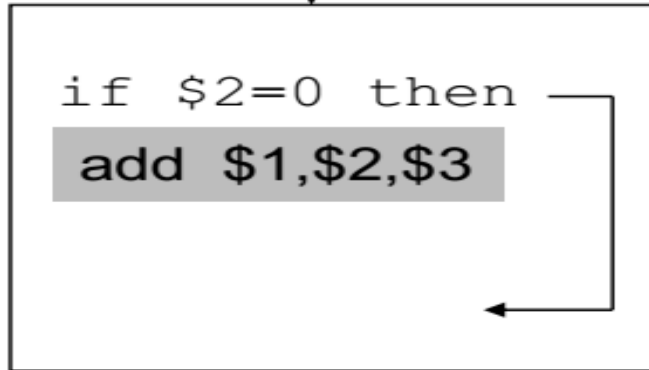
- **Superscalar machines with deep pipelines may require additional delay slots to avoid branch penalties**

Scheduling Branch Delay Slots (Fig A.14)

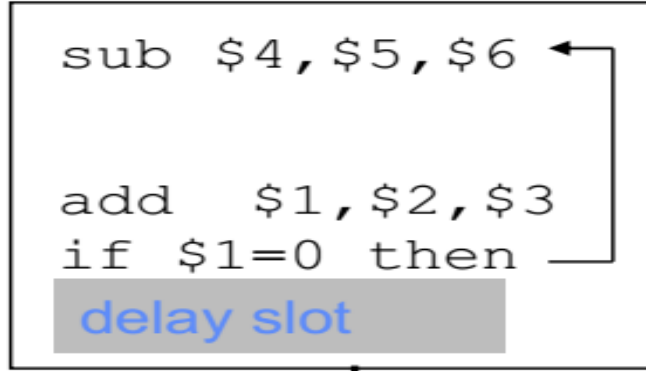
A. From before branch



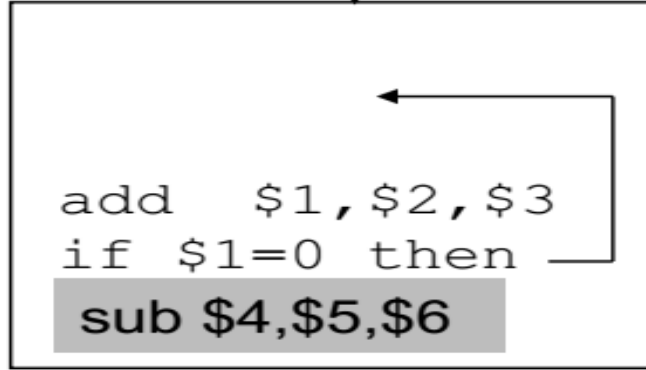
becomes ↓



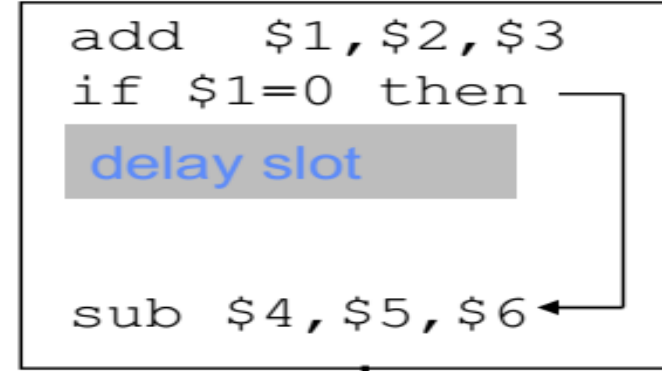
B. From branch target



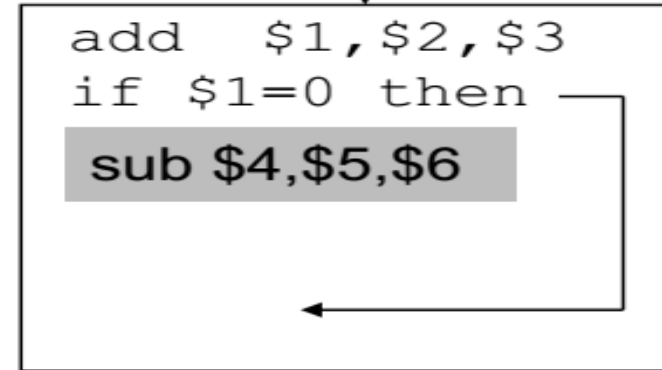
becomes ↓



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**

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% ($60\% \times 80\%$) 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 slot**
 - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
 - Growth in available transistors has made dynamic approaches relatively cheaper

Evaluating Branch Alternatives

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

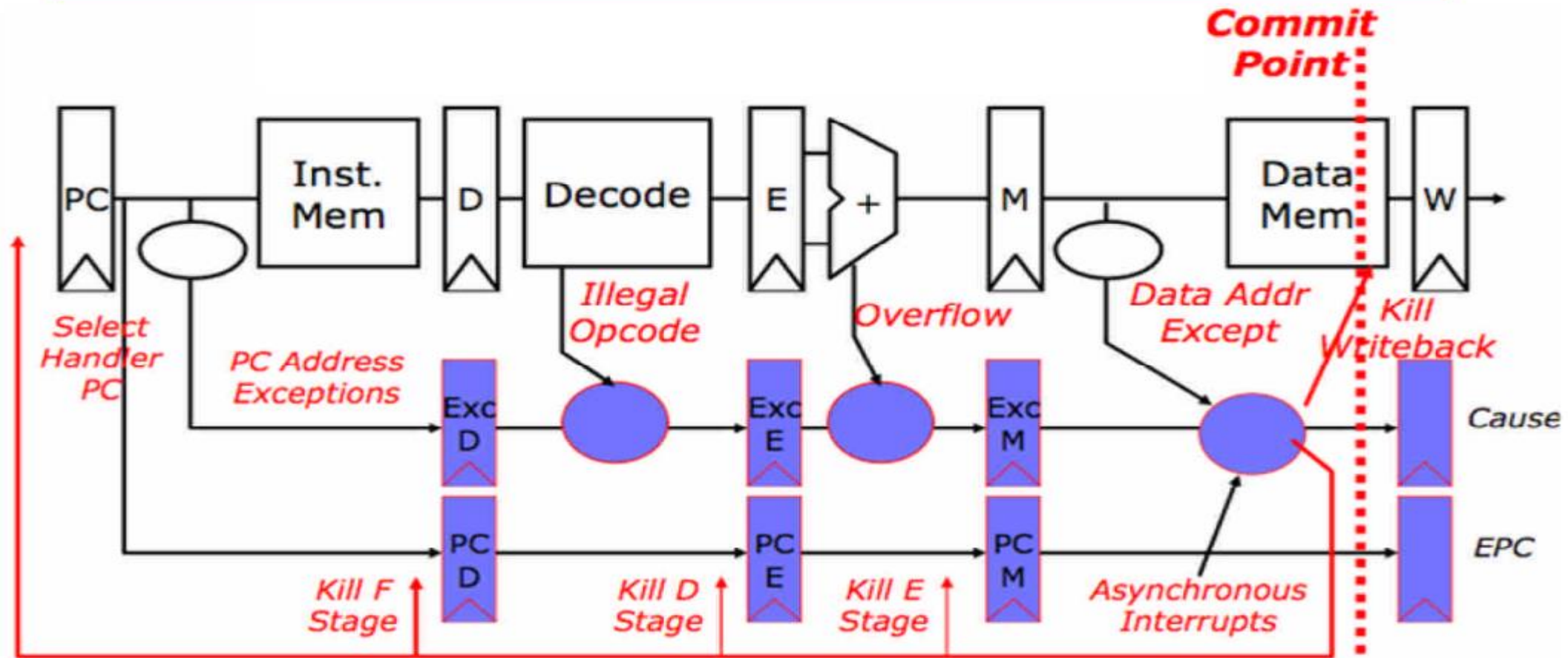
Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

<i>Scheduling scheme</i>	<i>Branch penalty</i>	<i>CPI</i>	<i>speedup v. unpipelined</i>	<i>speedup v. stall</i>
Stall pipeline	3	1.60	3.1	1.0
Predict taken	1	1.20	4.2	1.33
Predict not taken	1	1.14	4.4	1.40
Delayed branch	0.5	1.10	4.5	1.45

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}

Precise Exceptions in Static Pipelines



Key observation: architected state only change in memory and register write stages.

And In Conclusion: Control and Pipelining

- Quantify and summarize performance
 - Ratios, Geometric Mean, Multiplicative Standard Deviation
- F&P: Benchmarks age, disks fail, 1 point fail danger
- Next time: Read Appendix A, record bugs online!
- Control VIA **State Machines** and **Microprogramming**
- Just overlap tasks; easy if tasks are independent
- Speed Up \leq Pipeline Depth; if ideal CPI is 1, then:

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

- Hazards limit performance on computers:
 - Structural: need more HW resources
 - Data (RAW, WAR, WAW): need forwarding, compiler scheduling
 - Control: delayed branch, prediction
- Exceptions, Interrupts add complexity
- Next time: Read Appendix C, record bugs online!