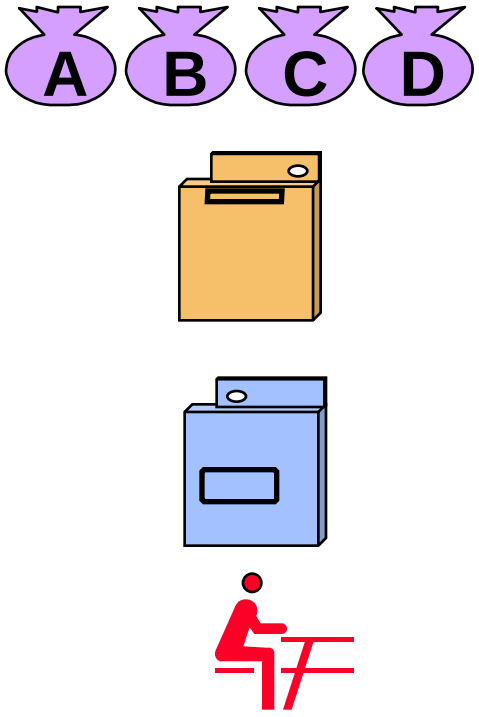
Chapter 17

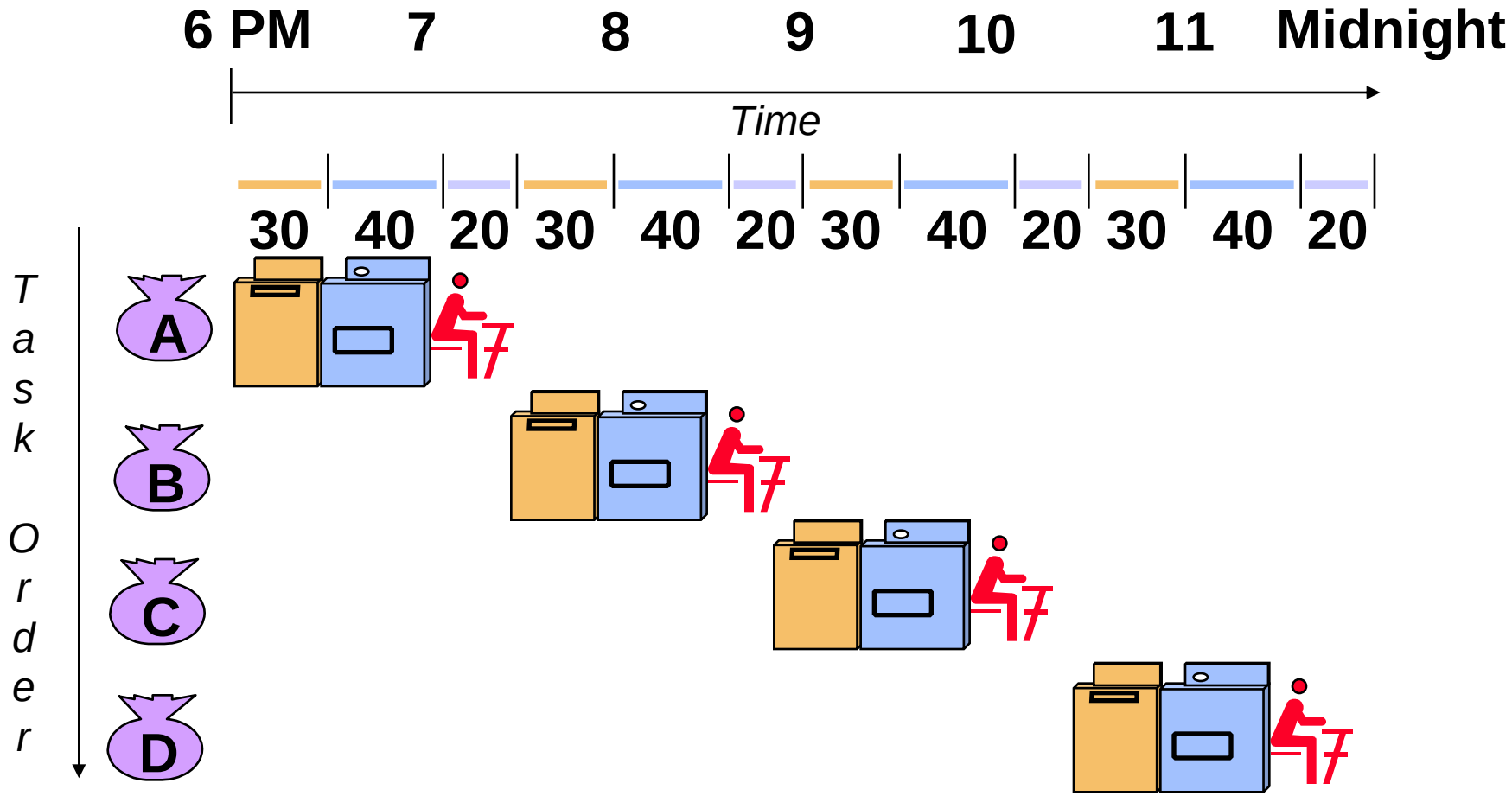
Pipeline Hazards

What Is Pipelining

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes



What Is Pipelining

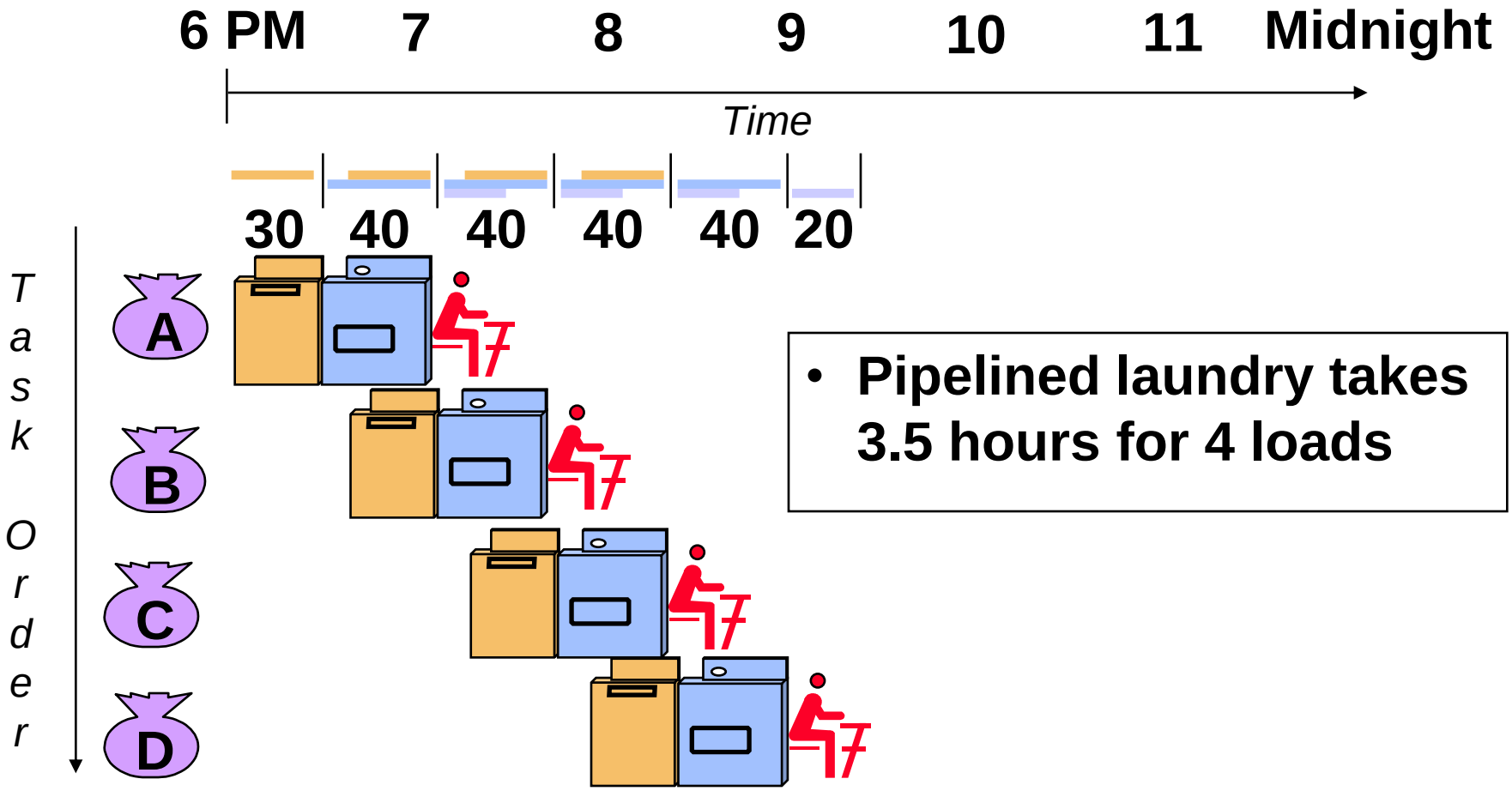


Sequential laundry takes 6 hours for 4 loads

If they learned pipelining, how long would laundry take?

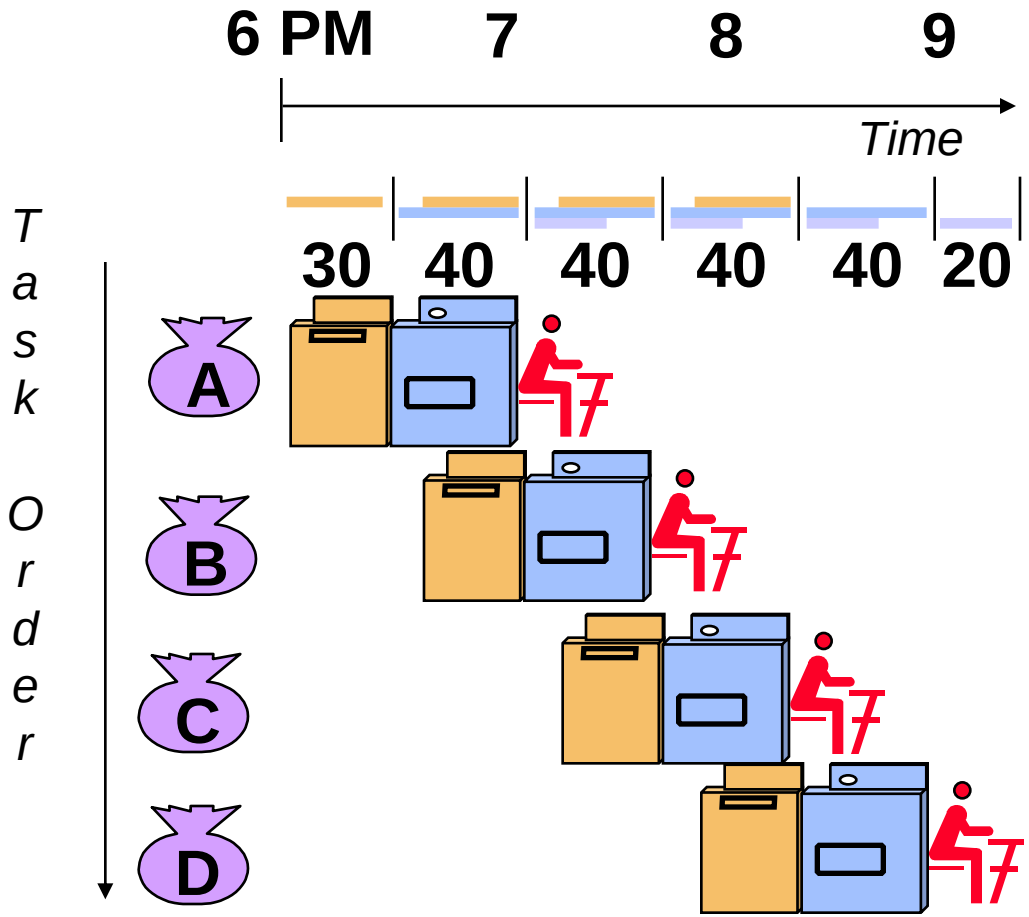
What Is Pipelining

Start work ASAP



- **Pipelined laundry takes 3.5 hours for 4 loads**

What Is Pipelining



Pipelining Lessons

- 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
- Potential speedup = **Number pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to "**fill**" pipeline and time to "**drain**" it reduces speedup

The Basic Pipeline For MIPS

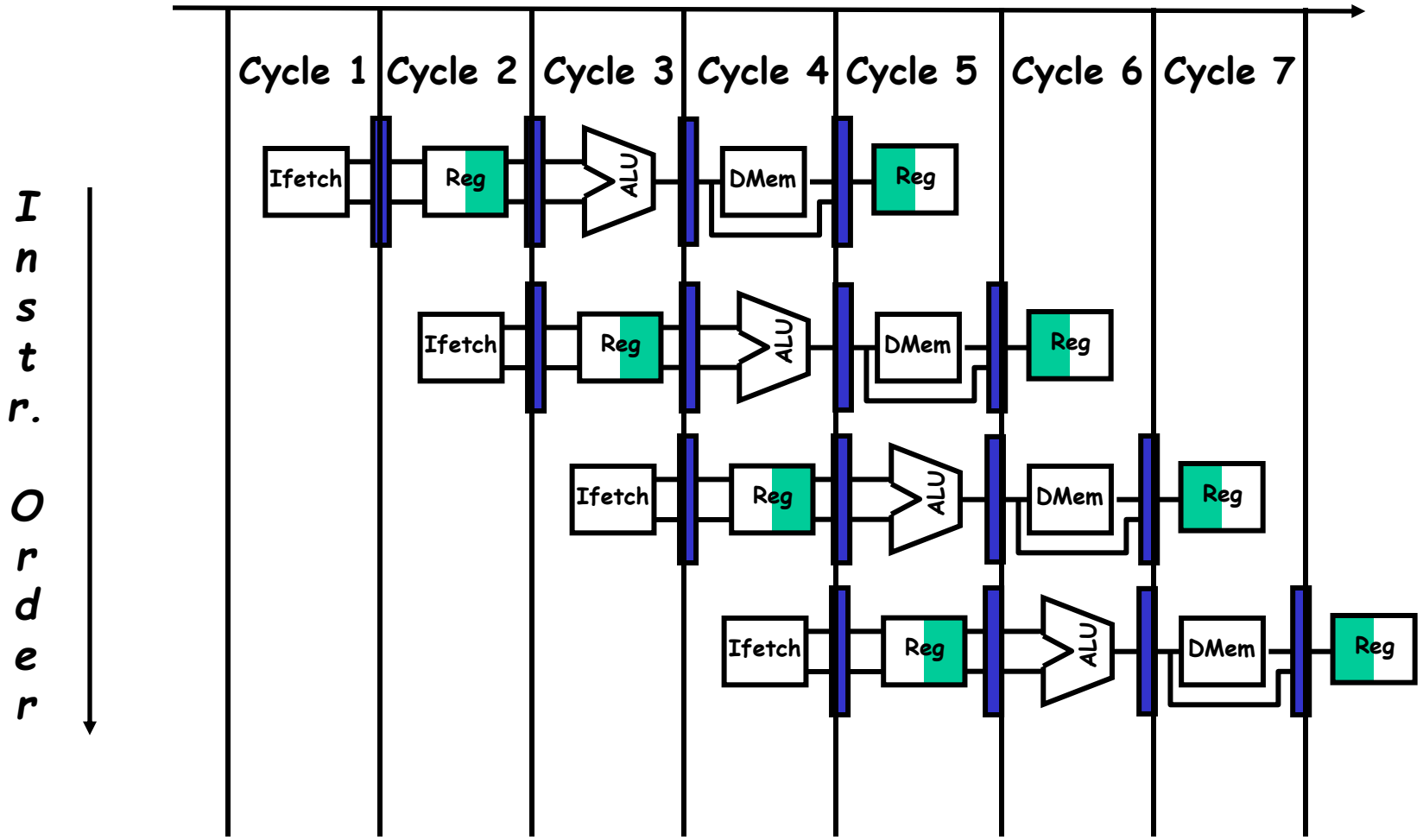


Figure 3.3

Pipeline Hurdles

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:** Pipelining of branches & other instructions that change the PC
- Common solution is to **stall** the pipeline until the hazard is resolved, inserting one or more "**bubbles**" in the pipeline

Pipeline Hurdles

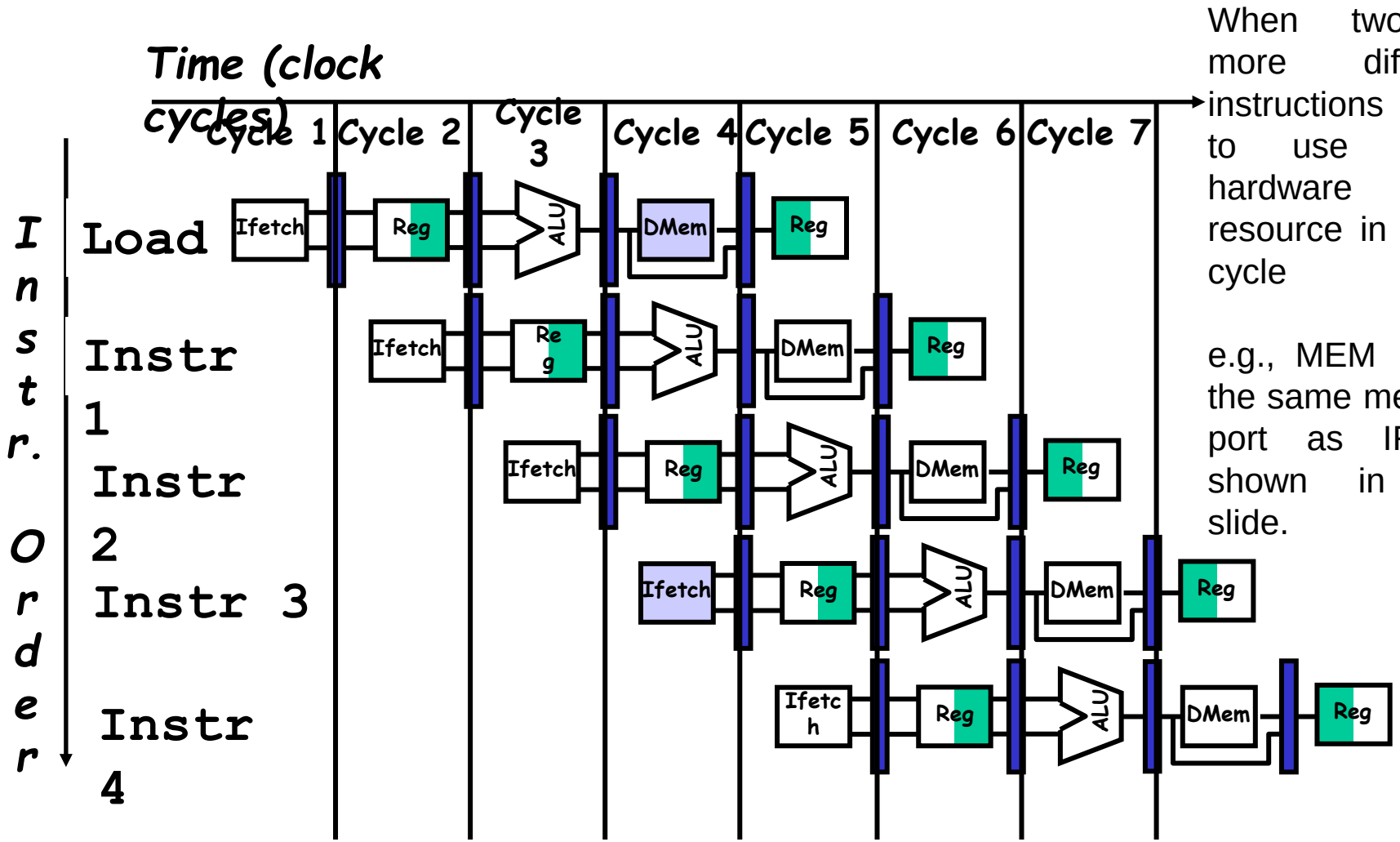
Definition

- conditions that lead to incorrect behavior if not fixed
- Structural hazard
  - two different instructions use same h/w in same cycle
- Data hazard
  - two different instructions use same storage
  - must appear as if the instructions execute in correct order
- Control hazard
  - one instruction affects which instruction is next

Resolution

- Pipeline interlock logic detects hazards and fixes them
- simple solution: stall
- increases CPI, decreases performance
- better solution: partial stall
- some instruction stall, others proceed better to stall early than late

Structural Hazards



When two or more different instructions want to use same hardware resource in same cycle

e.g., MEM uses the same memory port as IF as shown in this slide.

Figure 3.6

Structural Hazards

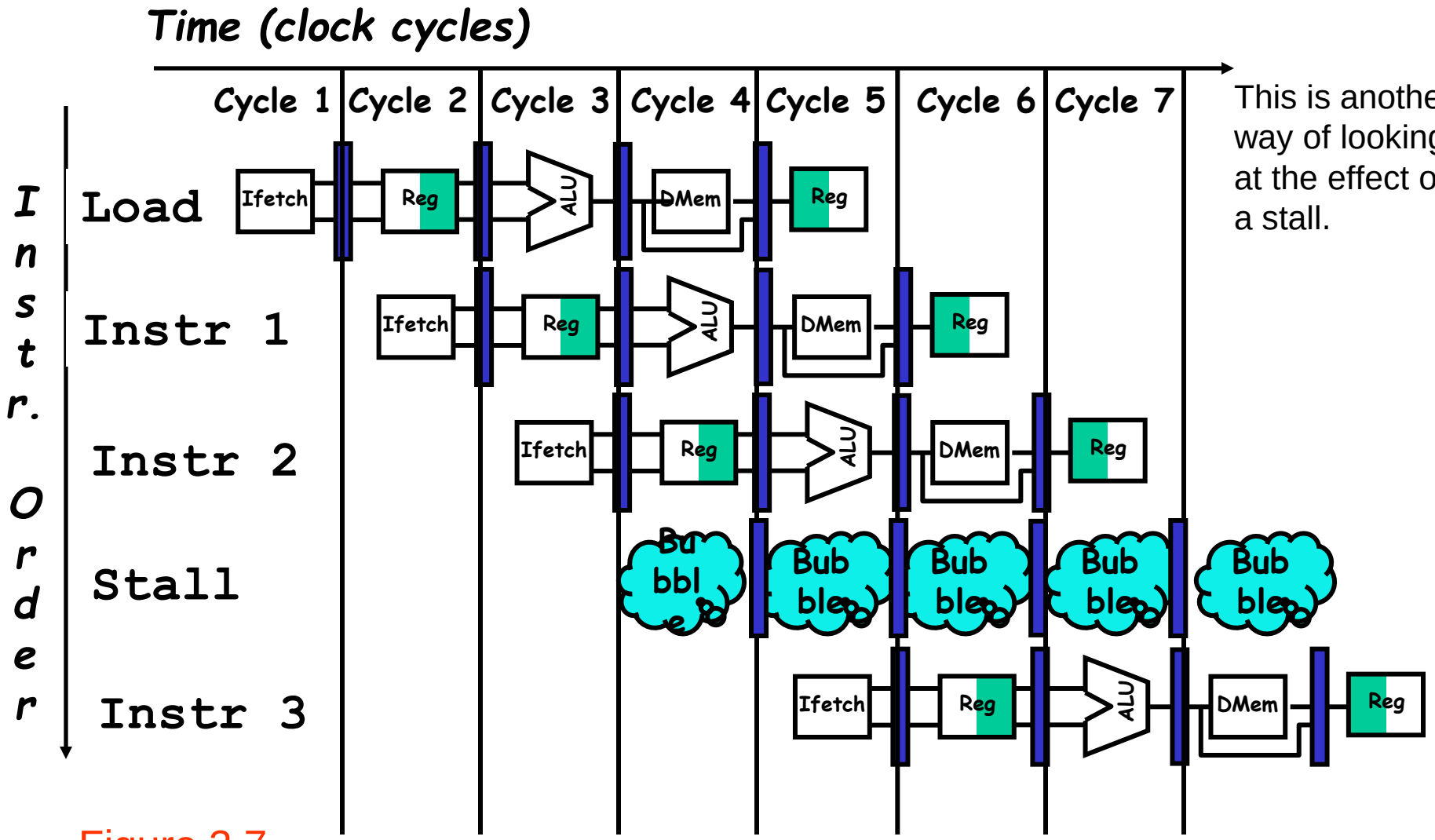


Figure 3.7

Structural Hazards

Instruction	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
Load instruction	IF	ID	EX	MEM	WB					
Instruction i + 1		IF	ID	EX	MEM	WB				
Instruction i + 2			IF	ID	EX	MEM	WB			
Instruction i + 3				stall	IF	ID	EX	MEM	WB	
Instruction i + 4						IF	ID	EX	MEM	WB
Instruction i + 5							IF	ID	EX	MEM
Instruction i + 6								IF	ID	EX

This is another way to represent the stall we saw on the last few pages.

Structural Hazards

Dealing with Structural Hazards

Stall

- low cost, simple
- Increases CPI
- use for rare case since stalling has performance effect

Pipeline hardware resource

- useful for multi-cycle resources
- good performance
- sometimes complex e.g., RAM

Replicate resource

- good performance
- increases cost (+ maybe interconnect delay)
- useful for cheap or divisible resources

Structural Hazards

Structural hazards are reduced with these rules:

- **Each instruction uses a resource at most once**
- **Always use the resource in the same pipeline stage**
- **Use the resource for one cycle only**

Many RISC ISA'a designed with this in mind

Sometimes very complex to do this. For example, memory of necessity is used in the IF and MEM stages.

Some common Structural Hazards:

- **Memory - we've already mentioned this one.**
- **Floating point - Since many floating point instructions require many cycles, it's easy for them to interfere with each other.**
- **Starting up more of one type of instruction than there are resources. For instance, the PA-8600 can support two ALU + two load/store instructions per cycle - that's how much hardware it has available.**

Structural Hazards

We want to compare the performance of two machines. Which machine is faster?

- Machine A: Dual ported memory - so there are no memory stalls
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate

Assume:

- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

$$\text{SpeedUp}_A = \text{Pipeline Depth}/(1 + 0) \times (\text{clock}_{unpipe}/\text{clock}_{pipe})$$
$$= \text{Pipeline Depth}$$
$$\text{SpeedUp}_B = \text{Pipeline Depth}/(1 + 0.4 \times 1) \times (\text{clock}_{unpipe}/(\text{clock}_{unpipe} / 1.05)$$
$$= (\text{Pipeline Depth}/1.4) \times 1.05$$
$$= 0.75 \times \text{Pipeline Depth}$$

SpeedUp_A / SpeedUp_B = Pipeline Depth / (0.75 x Pipeline Depth) = 1.33

- Machine A is 1.33 times faster

Data Hazards

These occur when at any time, there are instructions active that need to access the same data (memory or register) locations.

Where there's real trouble is when we have:

instruction A
instruction B

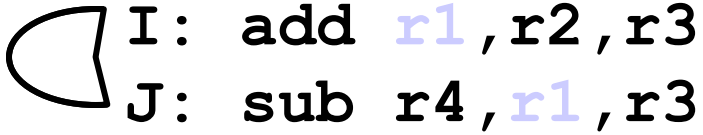
and B manipulates (reads or writes) data before A does. This violates the order of the instructions, since the architecture implies that A completes entirely before B is executed.

Data Hazards

Execution Order is:
$Instr_I$
$Instr_J$

Read After Write (RAW)

$Instr_J$ tries to read operand before $Instr_I$ writes it



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

Data Hazards

Execution Order is:
$Instr_I$
$Instr_J$

Write After Read (WAR)

$Instr_J$ tries to write operand *before* $Instr_I$ reads i

– Gets wrong operand

```
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

Data Hazards

Execution Order is:
$Instr_I$
$Instr_J$

Write After Write (WAW)

$Instr_J$ tries to write operand *before* $Instr_I$ writes it

- Leaves wrong result ($Instr_I$ not $Instr_J$)

```
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 later more complicated pipes**

Data Hazards

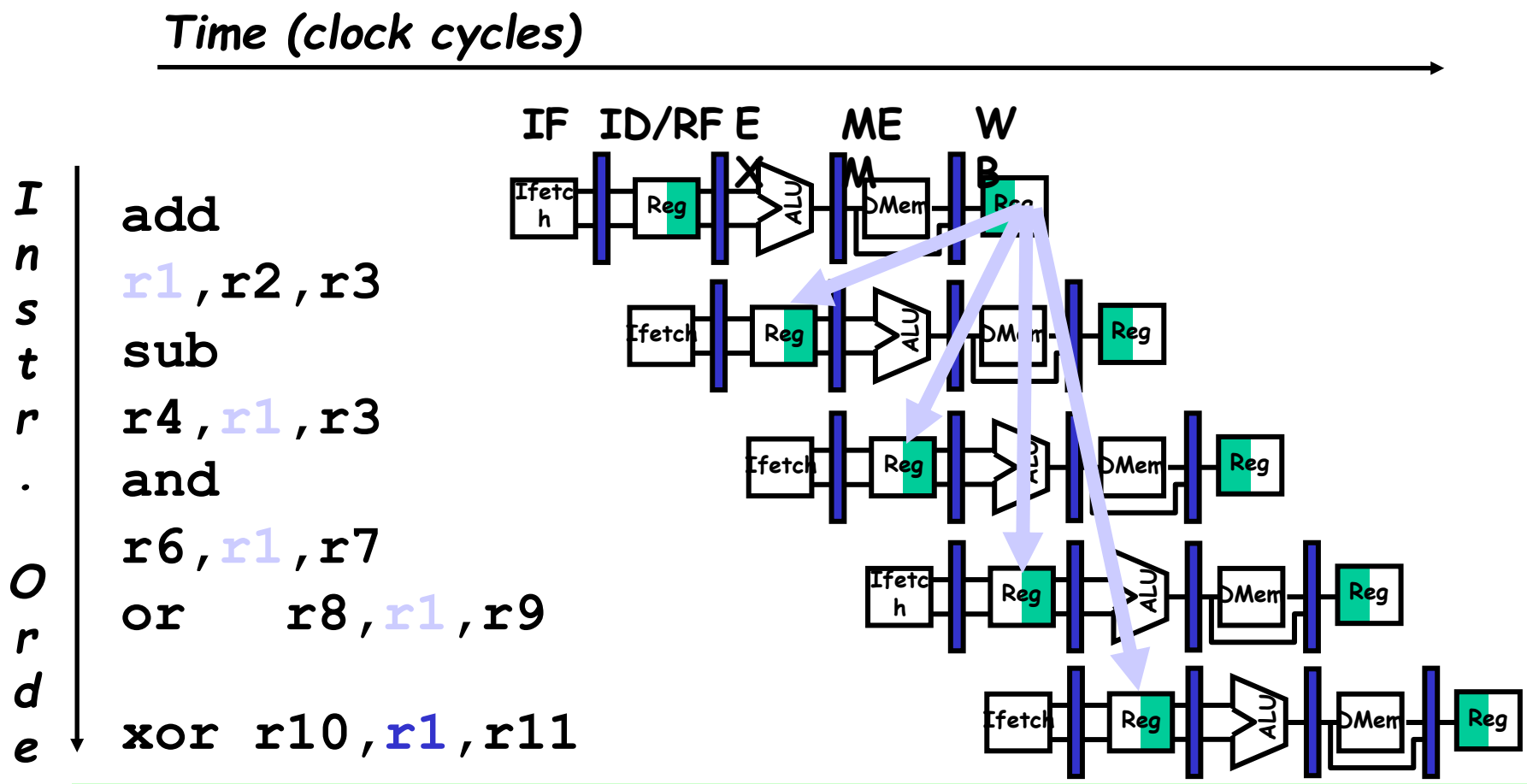
Simple Solution to RAW

- Hardware detects RAW and stalls
- Assumes register written then read each cycle
  + low cost to implement, simple
  -- reduces IPC
- Try to minimize stalls

Minimizing RAW stalls

- Bypass/forward/shortcircuit (We will use the word “forward”)
- Use data before it is in the register
  + reduces/avoids stalls
  -- complex
- Crucial for common RAW hazards

Data Hazards



The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

Figure 3.9

Data Hazards

Forwarding To Avoid Data Hazard

Forwarding is the concept of making data available to the input of the ALU for subsequent instructions, even though the generating instruction hasn't gotten to WB in order to write the memory or registers.

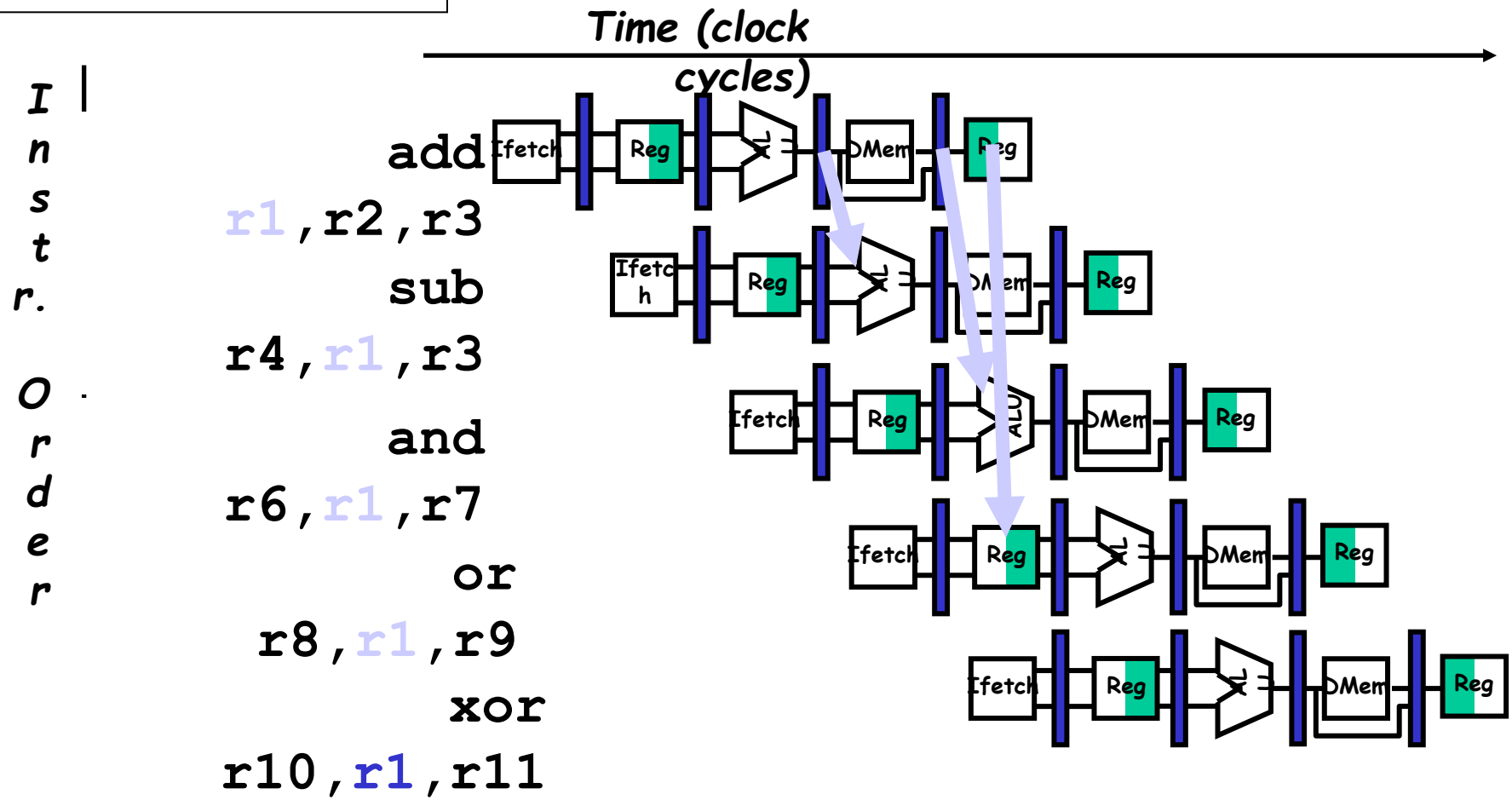
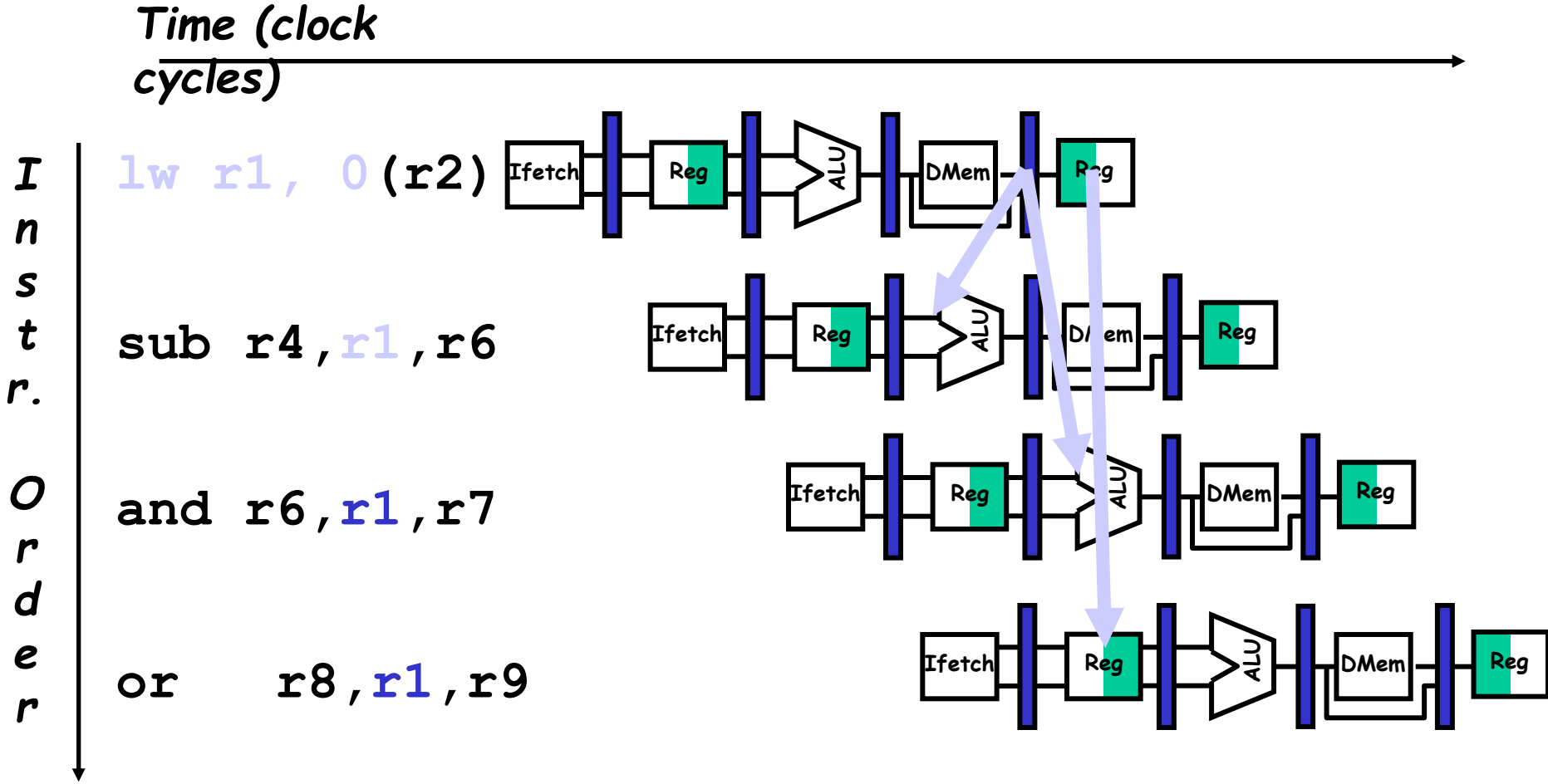


Figure 3.10

Data Hazards

The data isn't loaded until after the MEM stage.

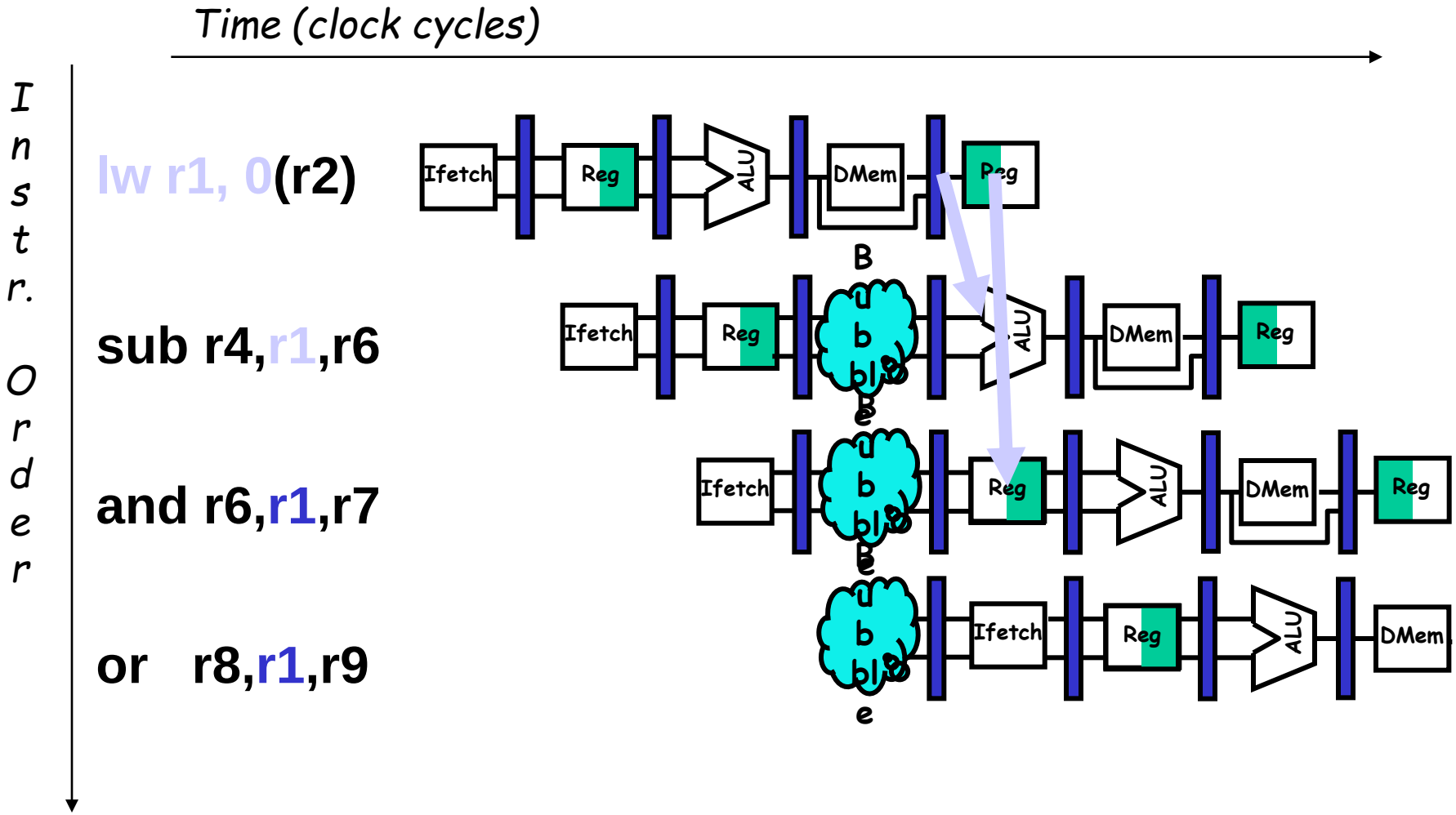


There are some instances where hazards occur, even with forwarding.

Figure 3.12

Data Hazards

The stall is necessary as shown here.



There are some instances where hazards occur, even with forwarding.

Figure 3.13

Data Hazards

This is another representation of the stall.

LW R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

LW R1, 0(R2)	IF	ID	EX	MEM	WB				
SUB R4, R1, R5		IF	ID	stall	EX	MEM	WB		
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB	
OR R8, R1, R9				stall	IF	ID	EX	MEM	WB

Control Hazards

A control hazard is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.

Control Hazards

Control Hazard on Branches Three Stage Stall

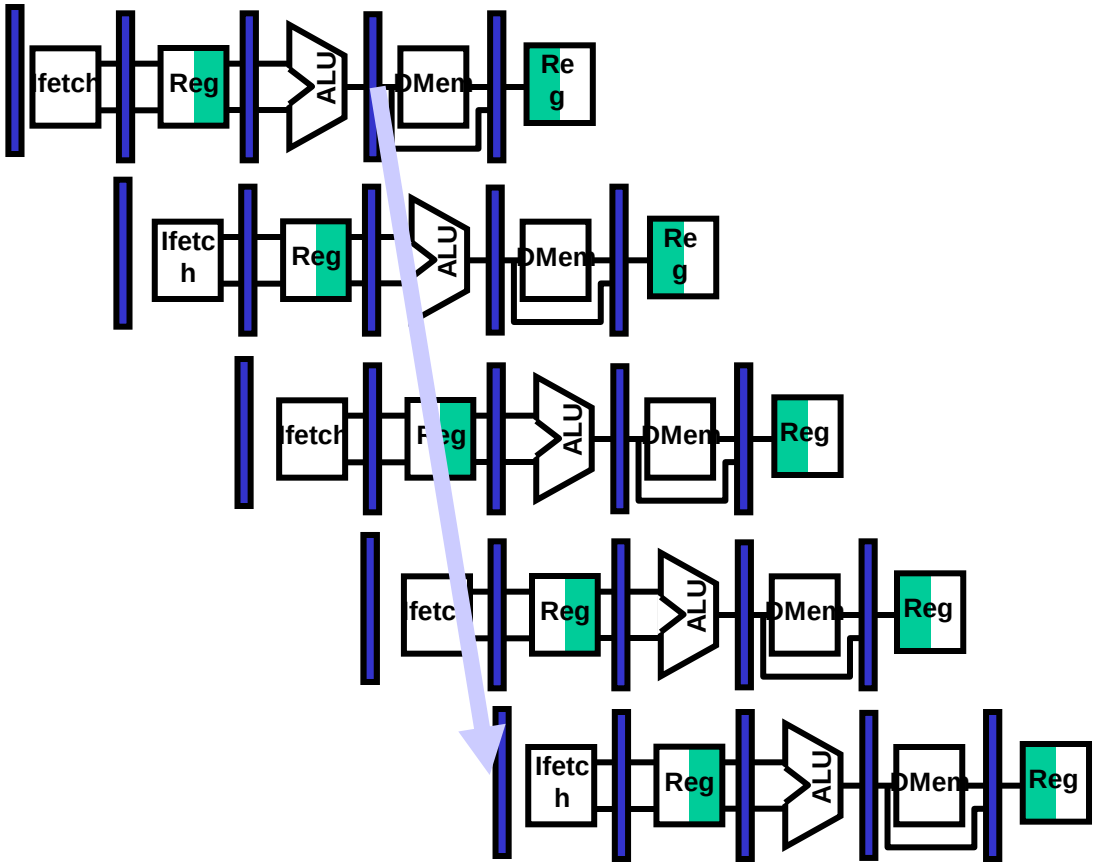
10: beq r1,r3,36

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11



Control Hazards

Branch Stall Impact

- **If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!**
  (Whoa! How did we get that 1.9???)
- **Two part solution to this dramatic increase:**
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier

- **MIPS branch tests if register = 0 or ^ 0**

- **MIPS Solution:**
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
    - must be fast
    - can't afford to subtract
    - compares with 0 are simple
    - Greater-than, Less-than test signbit, but not-equal must OR all bits
    - more general compares need ALU
  - 1 clock cycle penalty for branch versus 3

Control Hazards

Five 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

Control Hazards

Five Branch Hazard Alternatives

#4: Execute Both Paths

#5: Delayed Branch

- Define branch to take place AFTER a following instruction

```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
branch target if taken
```

Branch delay of length *n*

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

Control Hazards

Delayed Branch

- **Where to get instructions to fill branch delay slot?**
  - Before branch instruction
  - From the target address: only valuable when branch taken
  - From fall through: only valuable when branch not taken
  - Cancelling branches allow more slots to be filled

- **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: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)**

Control Hazards

Compiler "Static" Prediction of Taken/Untaken Branches

- **Improves strategy for placing instructions in delay slot**
- **Two strategies**
  - Backward branch predict taken, forward branch not taken
  - Profile-based prediction: record branch behavior, predict branch based on prior run

Summary of Pipelining Basics

- **Hazards limit performance**
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- **Increasing length of pipe increases impact of hazards; pipelining helps instruction bandwidth, not latency**
- **Interrupts, Instruction Set, FP makes pipelining harder**
- **Compilers reduce cost of data and control hazards**
  - Load delay slots
  - Branch delay slots
  - Branch prediction

Summary

A.1 What is Pipelining?

A.2 The Major Hurdle of Pipelining-Structural Hazards

- Data Hazards
- Control Hazards

A.3 How is Pipelining Implemented

A.4 What Makes Pipelining Hard to Implement?

A.5 Extending the MIPS Pipeline to Handle Multi-cycle Operations