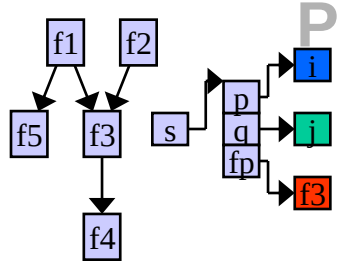
The Big Picture

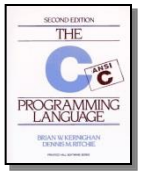
Requirements

Algorithms



Problem Focus

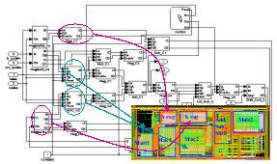
```
f2() {
  f3(s2, &j, &i);
  *s2->p = 10;
  i = *s2->q + i;
}
```



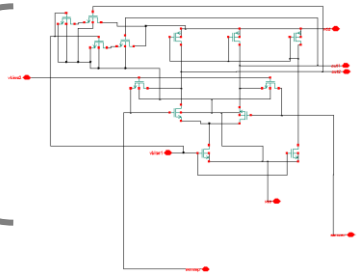
Prog. Lang./OS

ISA

```
i1: ld r1, b          <p1>
i2: ld r2, c          <p1>
i3: ld r5, z          <p3>
i4: mul r6, r5, 3     <p3>
i5: add r3, r1, r2    <p1>
```

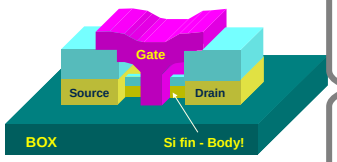


uArch



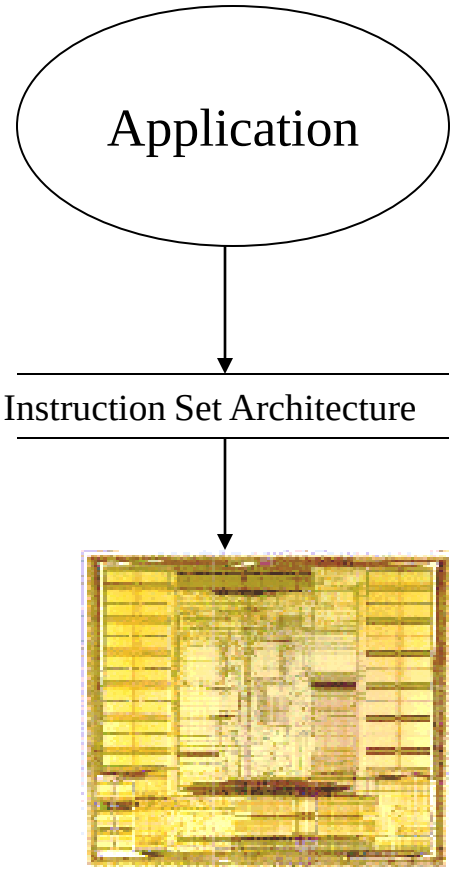
Circuit

Performance Focus



Device

Instruction Set Architecture



…SPARC MIPS ARM x86 HP-PA IA-64…

Intel Pentium X
AMD K6, Athlon, Opteron
Transmeta Crusoe TM5x00

Instruction Set Architecture

- **Strong influence on cost/performance**
- **New ISAs are rare, but versions are not**
  - 16-bit, 32-bit and 64-bit X86 versions
- **Longevity is a strong function of marketing prowess**

Traditional Issues

- Strongly constrained by the number of bits available to instruction encoding
- Opcodes/operands
- Registers/memory
- Addressing modes
- Orthogonality
- 0, 1, 2, 3 address machines
- Instruction formats
- Decoding uniformity

Introduction

A.1 What is Pipelining?

A.2 The Major Hurdle of Pipelining-Structural Hazards

- Data Hazards
- Control Hazards

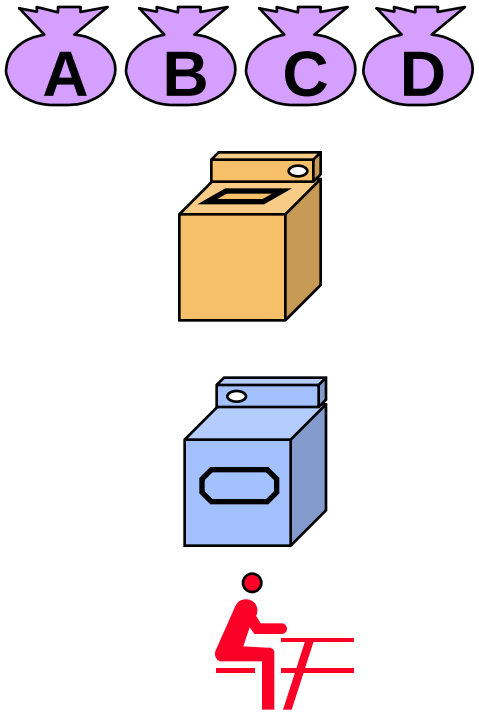
A.3 How is Pipelining Implemented

A.4 What Makes Pipelining Hard to Implement?

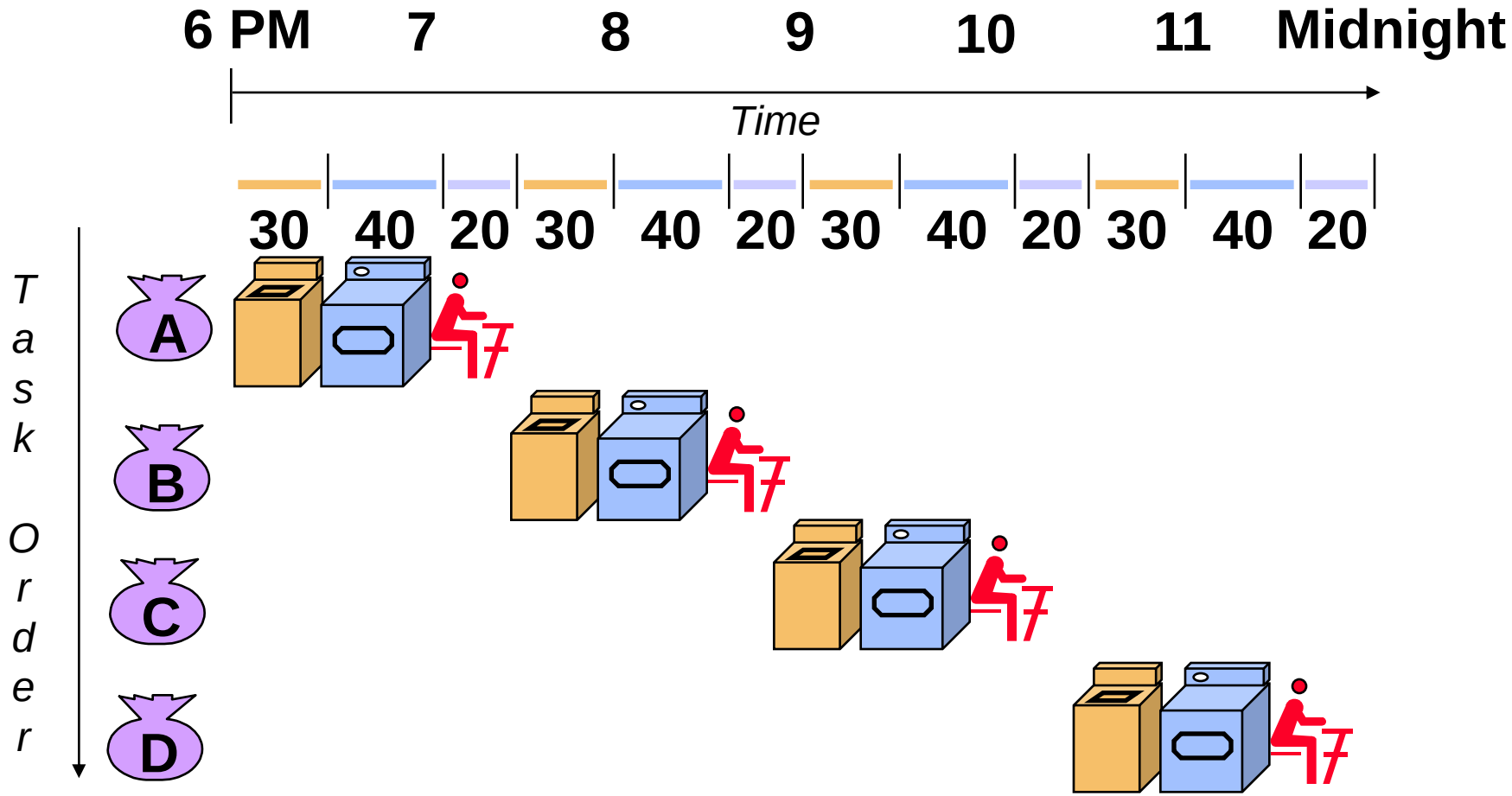
A.5 Extending the MIPS Pipeline to Handle Multi-cycle Operations

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

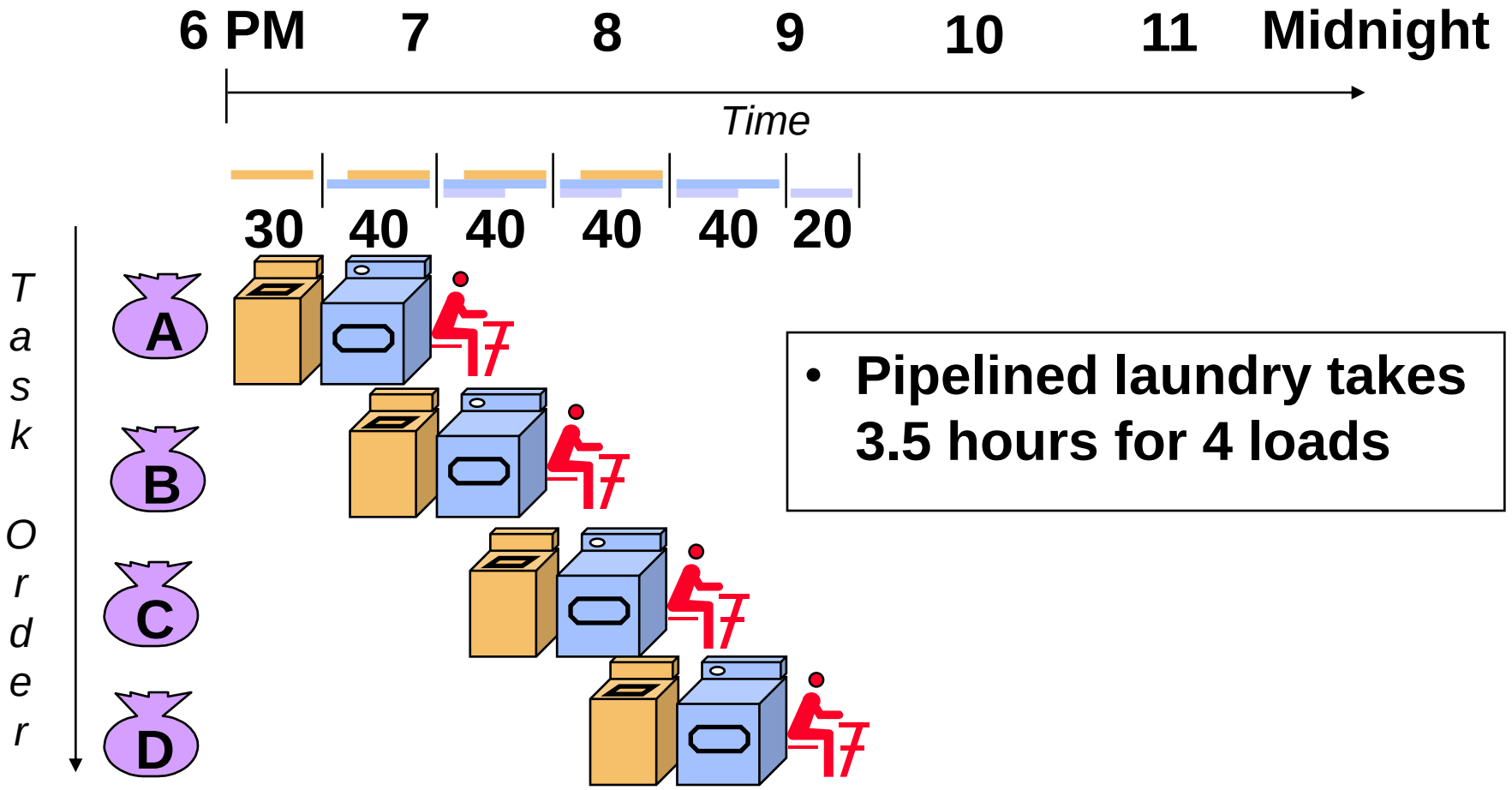


Sequential laundry takes 6 hours for 4 loads

If they learned pipelining, how long would laundry take?

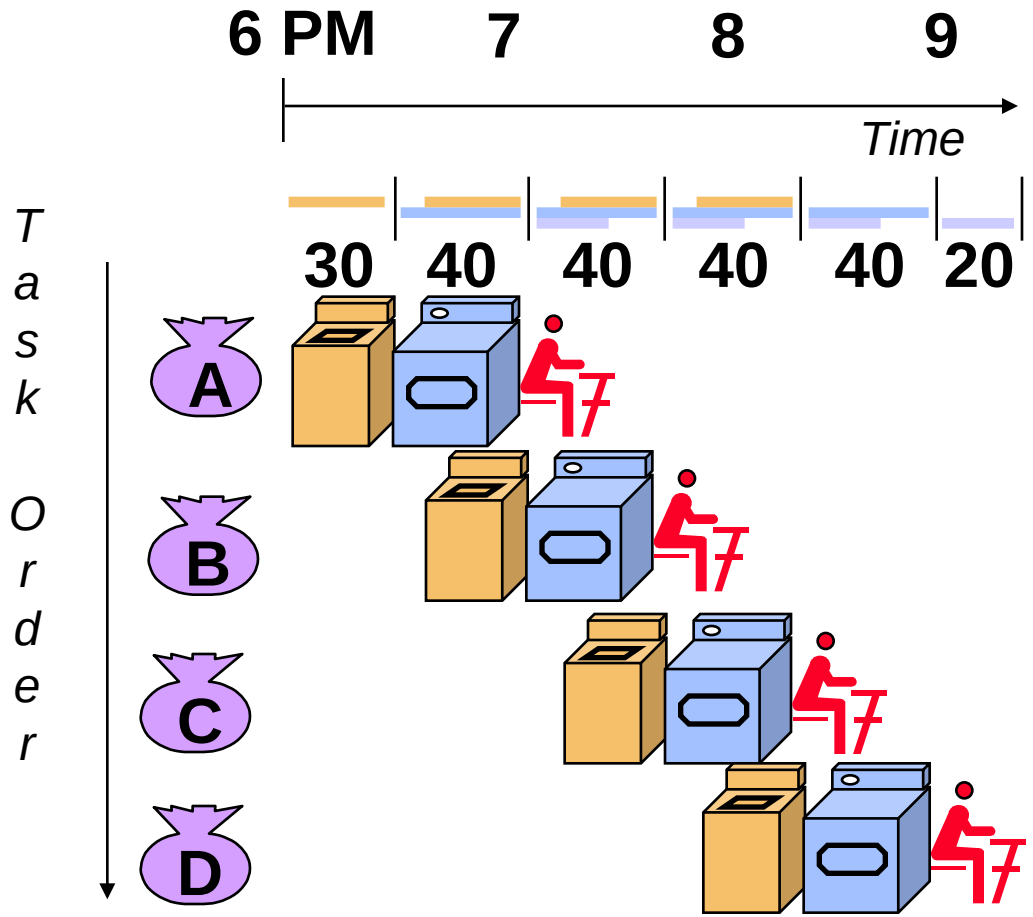
What Is Pipelining

Start work ASAP



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

What Is Pipelining

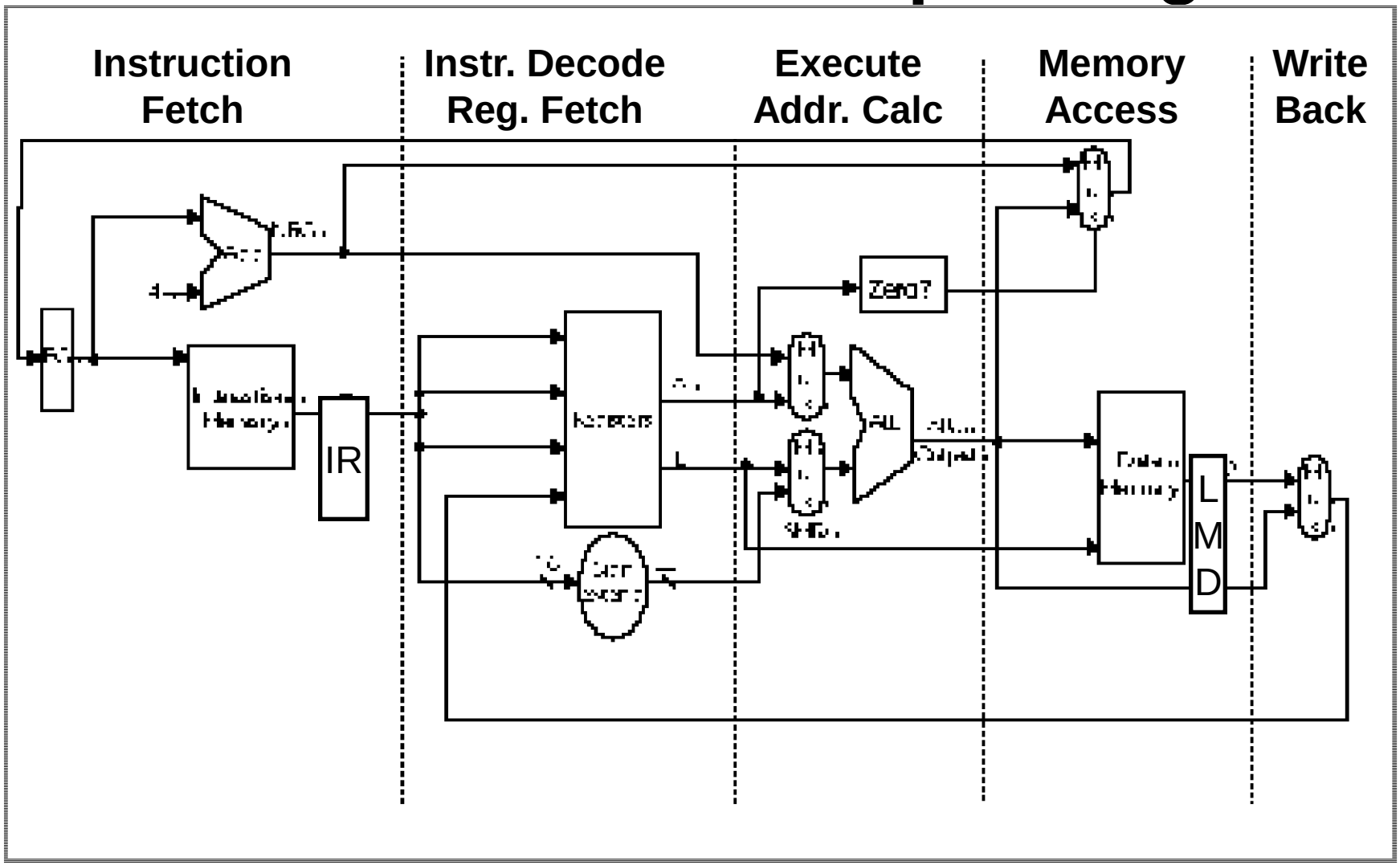


Pipelining Lessons

- 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

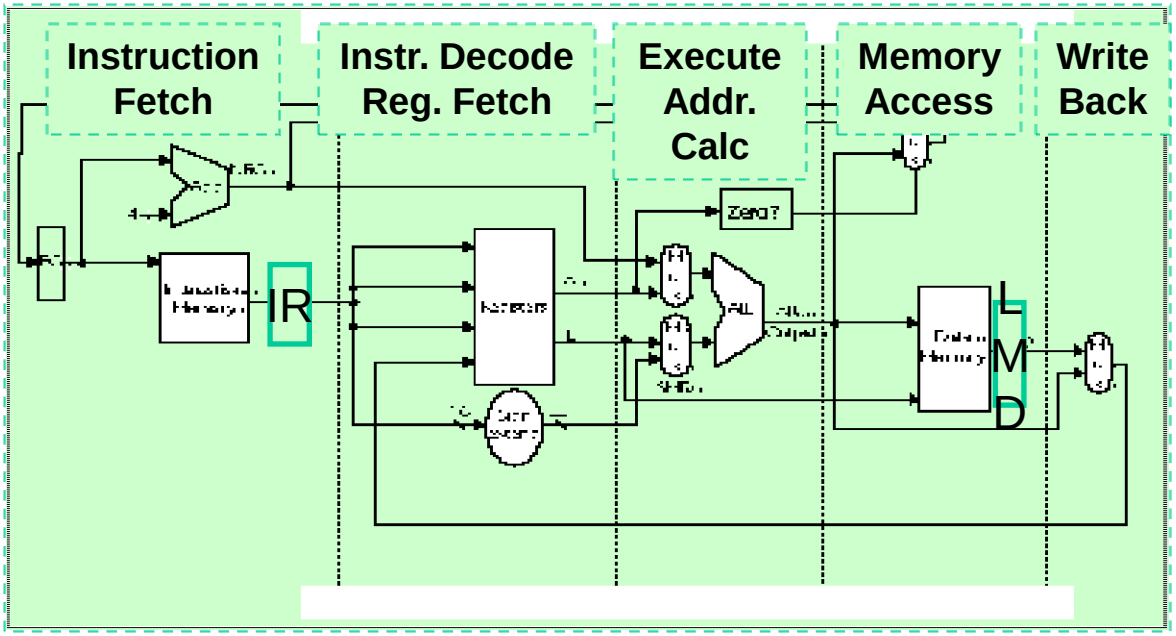
What Is Pipelining

MIPS Without Pipelining



What Is Pipelining

MIPS Functions



Passed To Next Stage

IR <- Mem[PC]

NPC <- PC + 4

Instruction Fetch (IF):

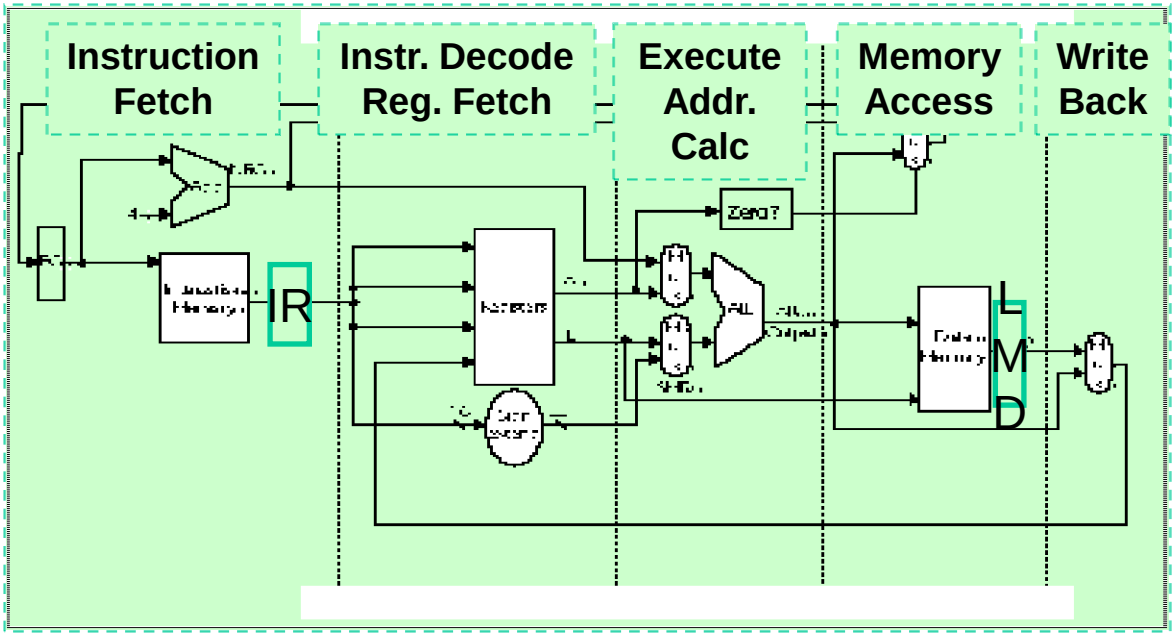
Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC by 4 to address the next sequential instruction.

IR holds the instruction that will be used in the next stage.

NPC holds the value of the next PC.

What Is Pipelining

MIPS Functions



Passed To Next Stage

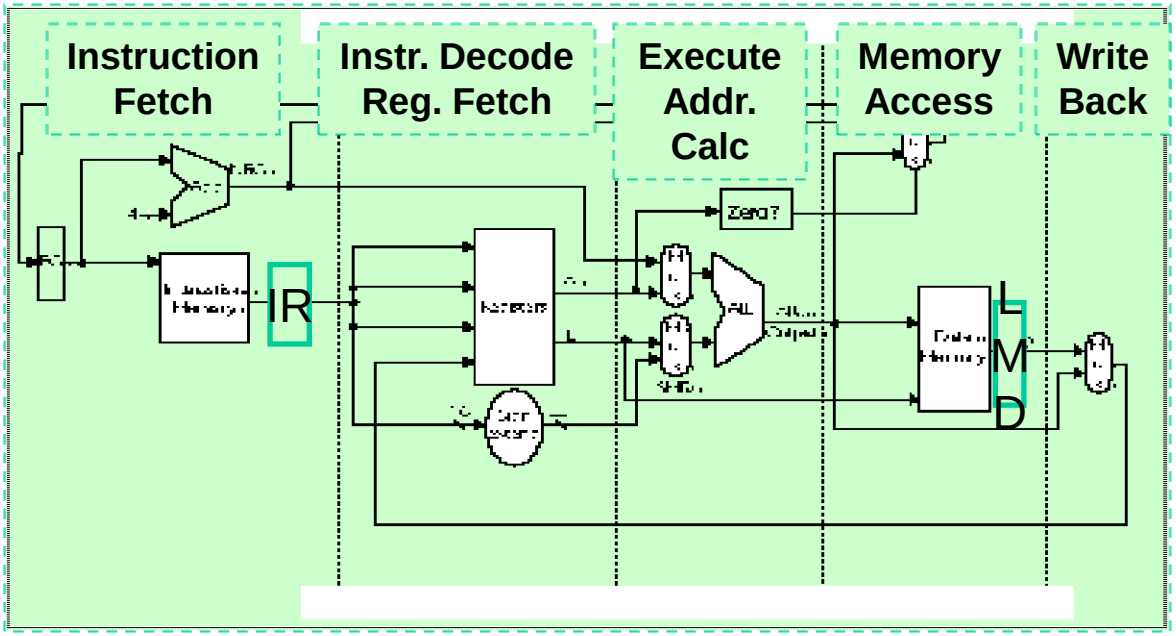
A <- Regs[IR6..IR10];
B <- Regs[IR10..IR15];
Imm <- ((IR16) ##IR16-31

Instruction Decode/Register Fetch Cycle (ID):

Decode the instruction and access the register file to read the registers.
The outputs of the general purpose registers are read into two temporary registers (A & B) for use in later clock cycles.
We extend the sign of the lower 16 bits of the Instruction Register.

What Is Pipelining

MIPS Functions



Passed To Next Stage

A <- A func. B

cond = 0;

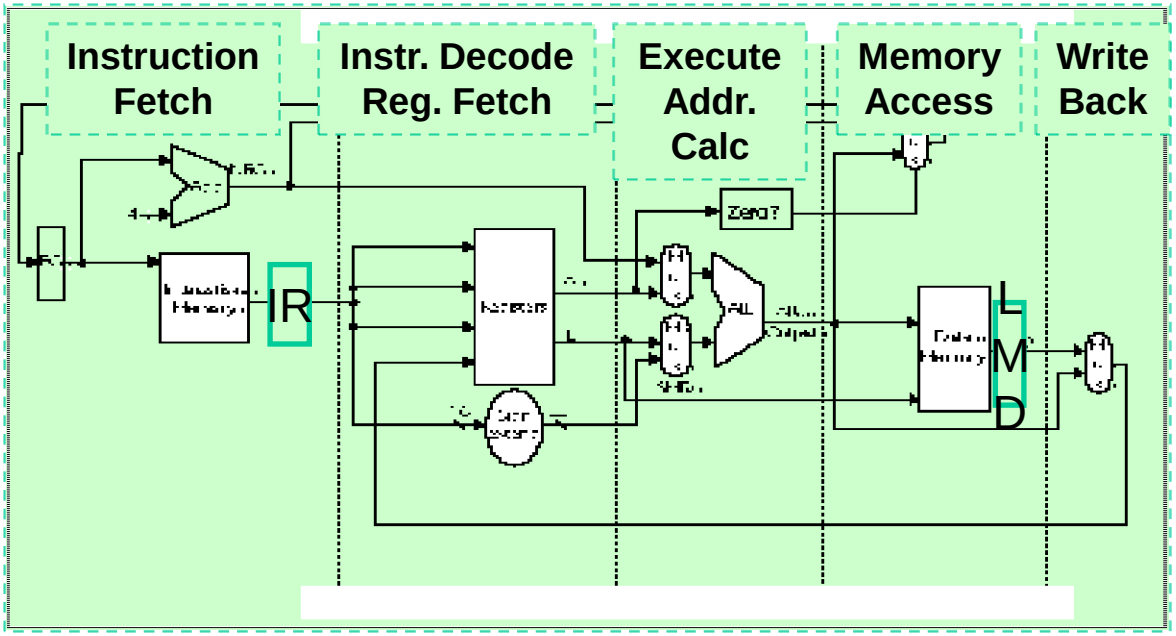
Execute Address Calculation (EX):

We perform an operation (for an ALU) or an address calculation (if it's a load or a Branch).

If an ALU, actually do the operation. If an address calculation, figure out how to obtain the address and stash away the location of that address for the next cycle.

What Is Pipelining

MIPS Functions



Passed To Next Stage

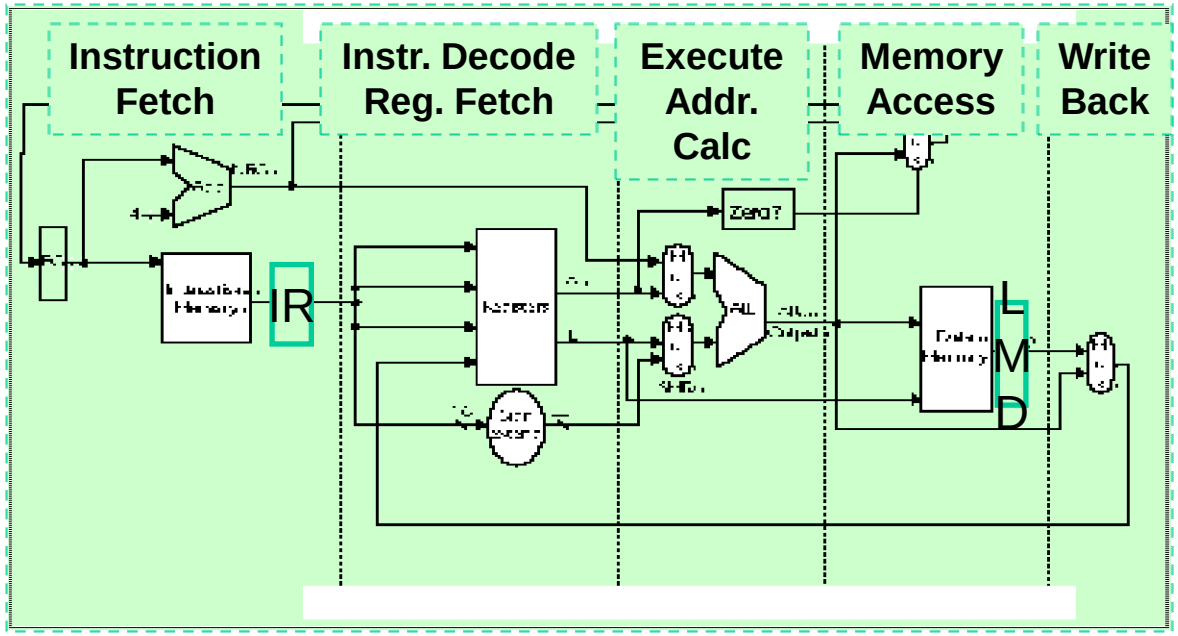
A = Mem[prev. B]
or
Mem[prev. B] = A

MEMORY ACCESS (MEM):

If this is an ALU, do nothing.
If a load or store, then access memory.

What Is Pipelining

MIPS Functions



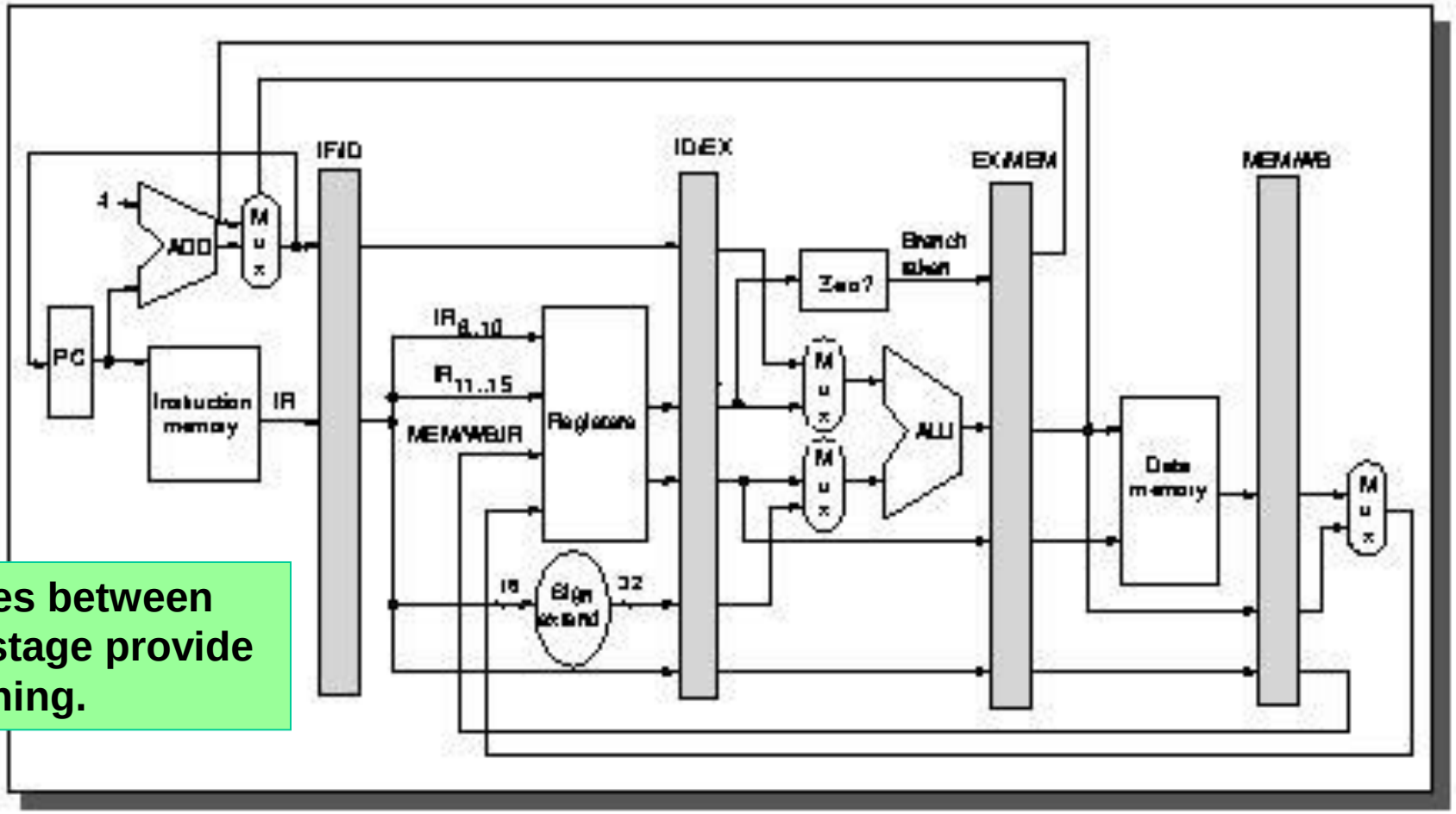
Passed To Next Stage

Regs <- A, B;

WRITE BACK (WB):

Update the registers from either the ALU or from the data loaded.

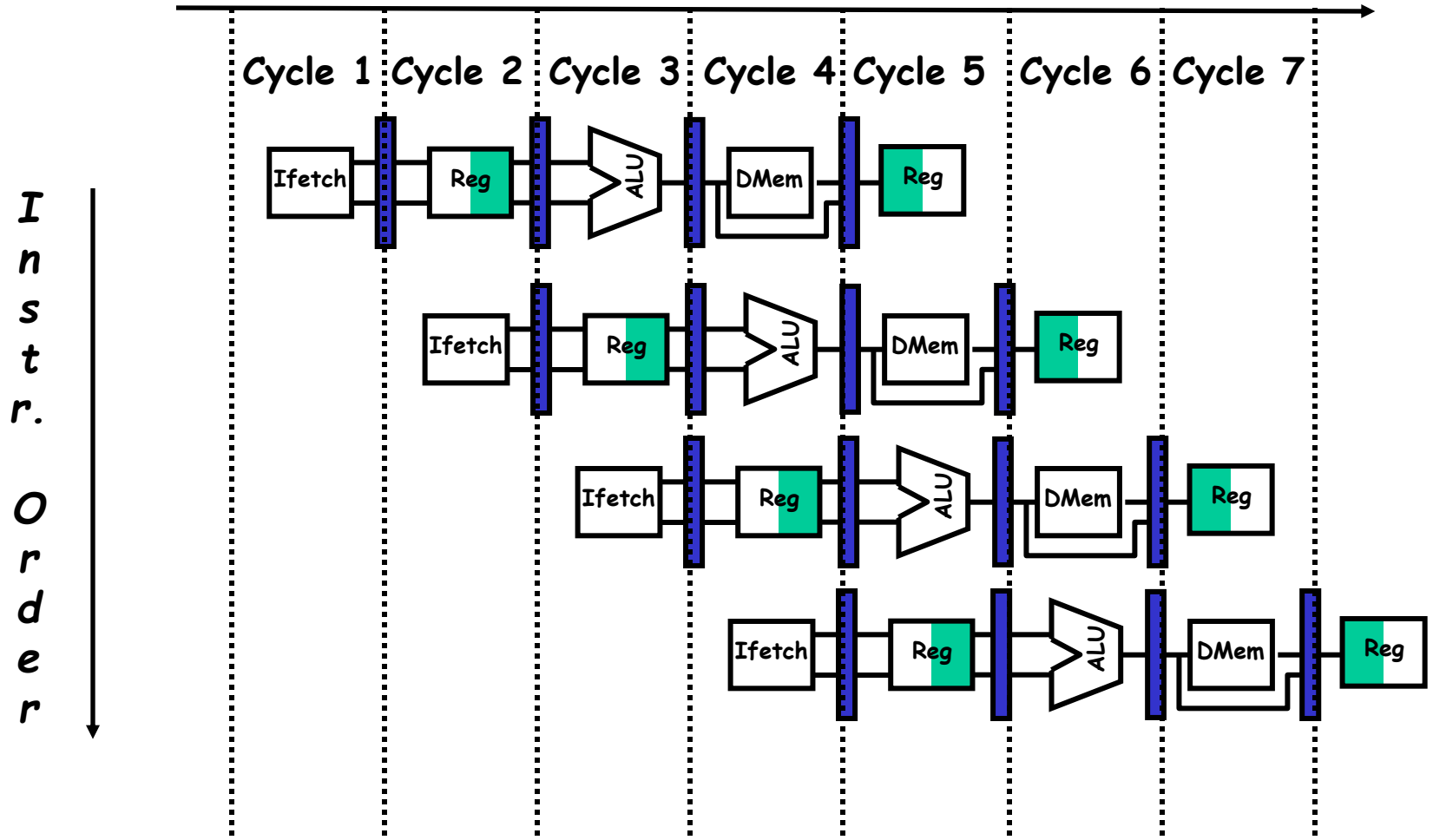
The Basic Pipeline For MIPS



Latches between each stage provide pipelining.

FIGURE 3.4 The datapath is pipelined by adding a set of registers, one between each pair of pipe stages.

The Basic Pipeline For MIPS



Pipeline Hurdles

Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle

- **Structural hazards:** H/W cannot support the 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
  - Current instruction affects next instruction

Pipeline Hurdles

Resolution

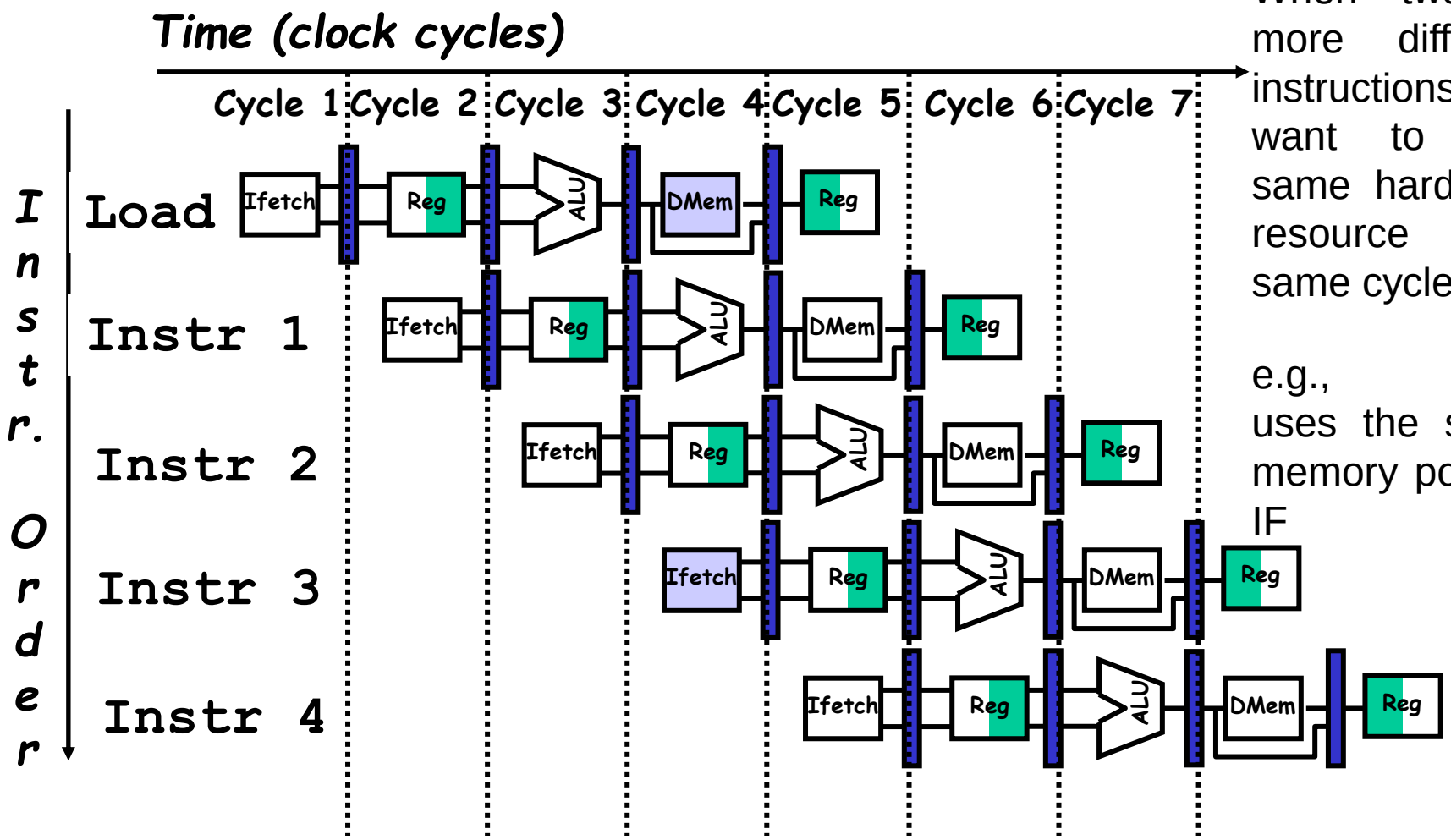
1.Pipeline interlock logic detects hazards and fixes them simple solution: stall

2.increases CPI, decreases performance

better solution: partial stall

some instruction stall, others proceed better to stall early than late

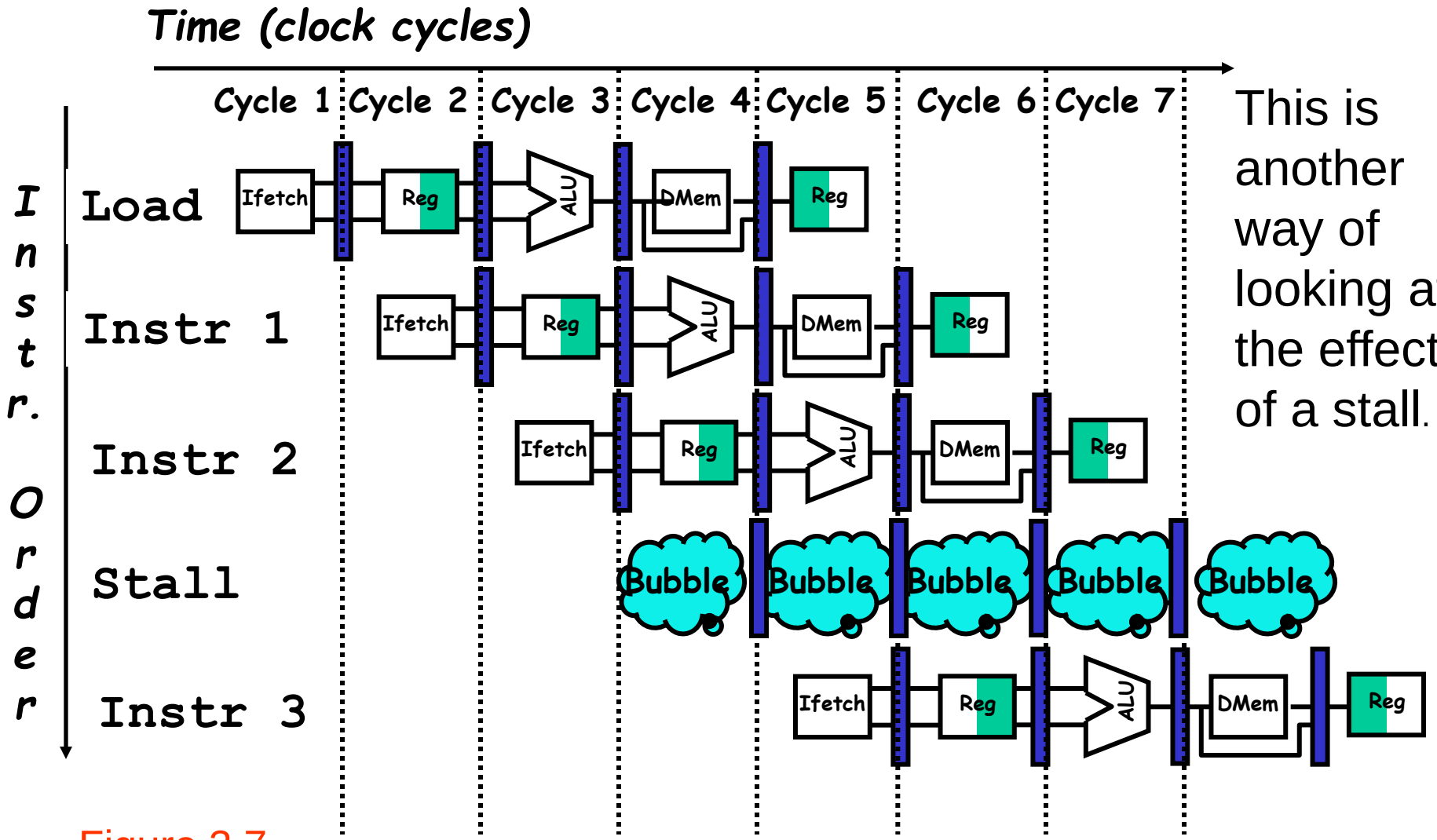
Structural Hazards



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

Structural Hazards



This is another way of looking at the effect of a stall.

Figure 3.7

Structural Hazards

	Clock cycle number									
Instruction	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

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

Structural Hazards

Dealing with Structural Hazards

Replicate resource

- good performance
- increases cost
- 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.

Structural Hazards

Some common Structural Hazards:

- **Memory**
- **Floating point - Since many floating point instructions require many cycles, it's easy for them to interfere with each other.**

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, active instruction that need to access the same data (memory or register) locations.

instruction A
instruction B

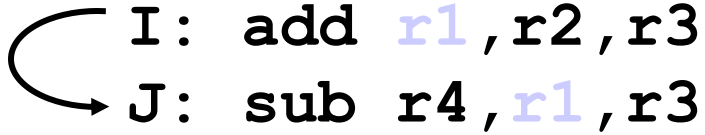
B manipulates (reads or writes) data before A does.

This violates the order of the instructions, since the architecture implies that A completes entirely before B is executed.

Data Hazards

Read After Write (RAW)

Execution Order is:
$Instr_I$
$Instr_J$



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

- **Caused by a "Dependence" .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

```
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

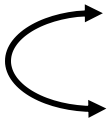
Data Hazards

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

Data Hazards

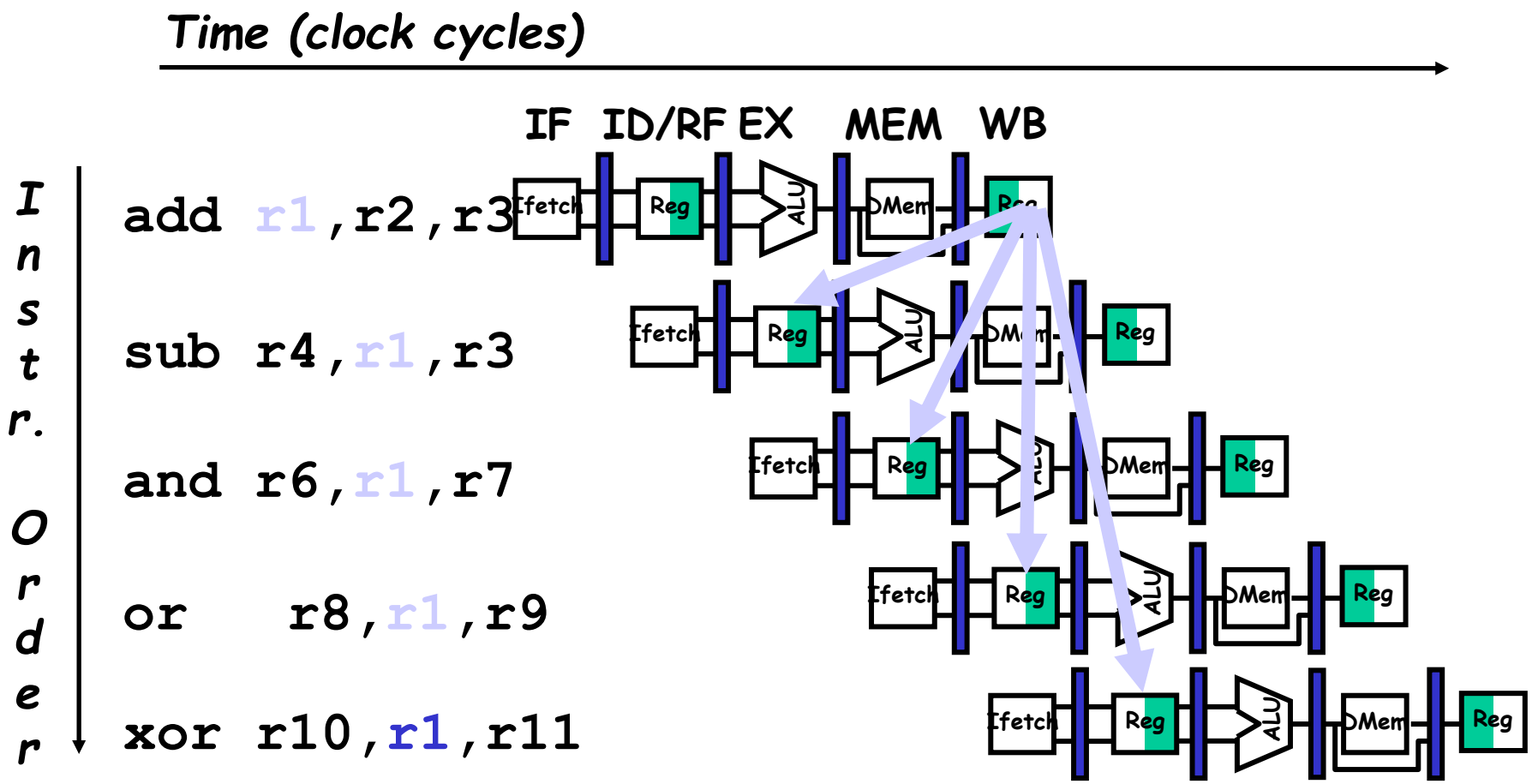
Simple Solution to RAW

- 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
- 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.

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.

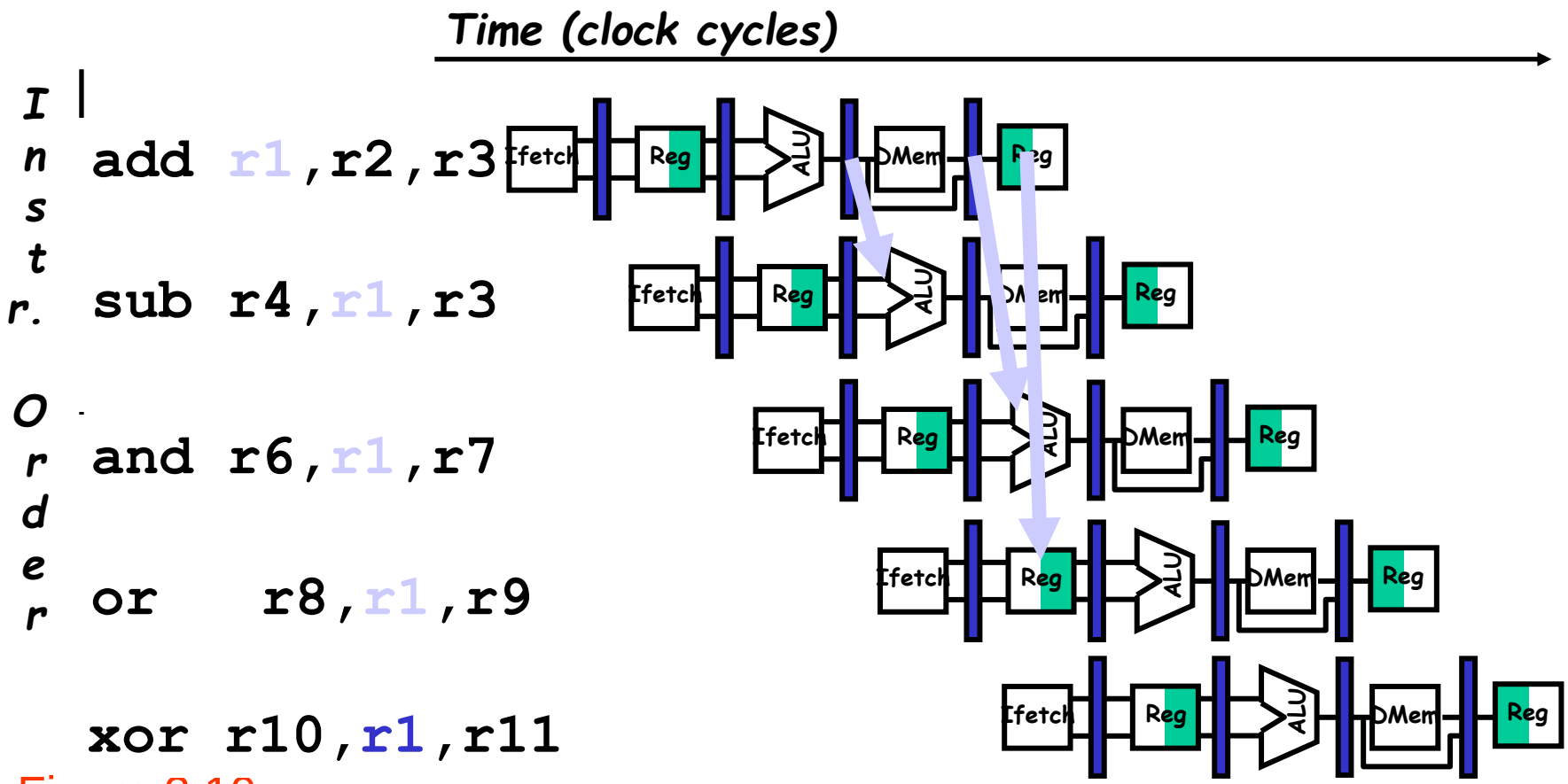
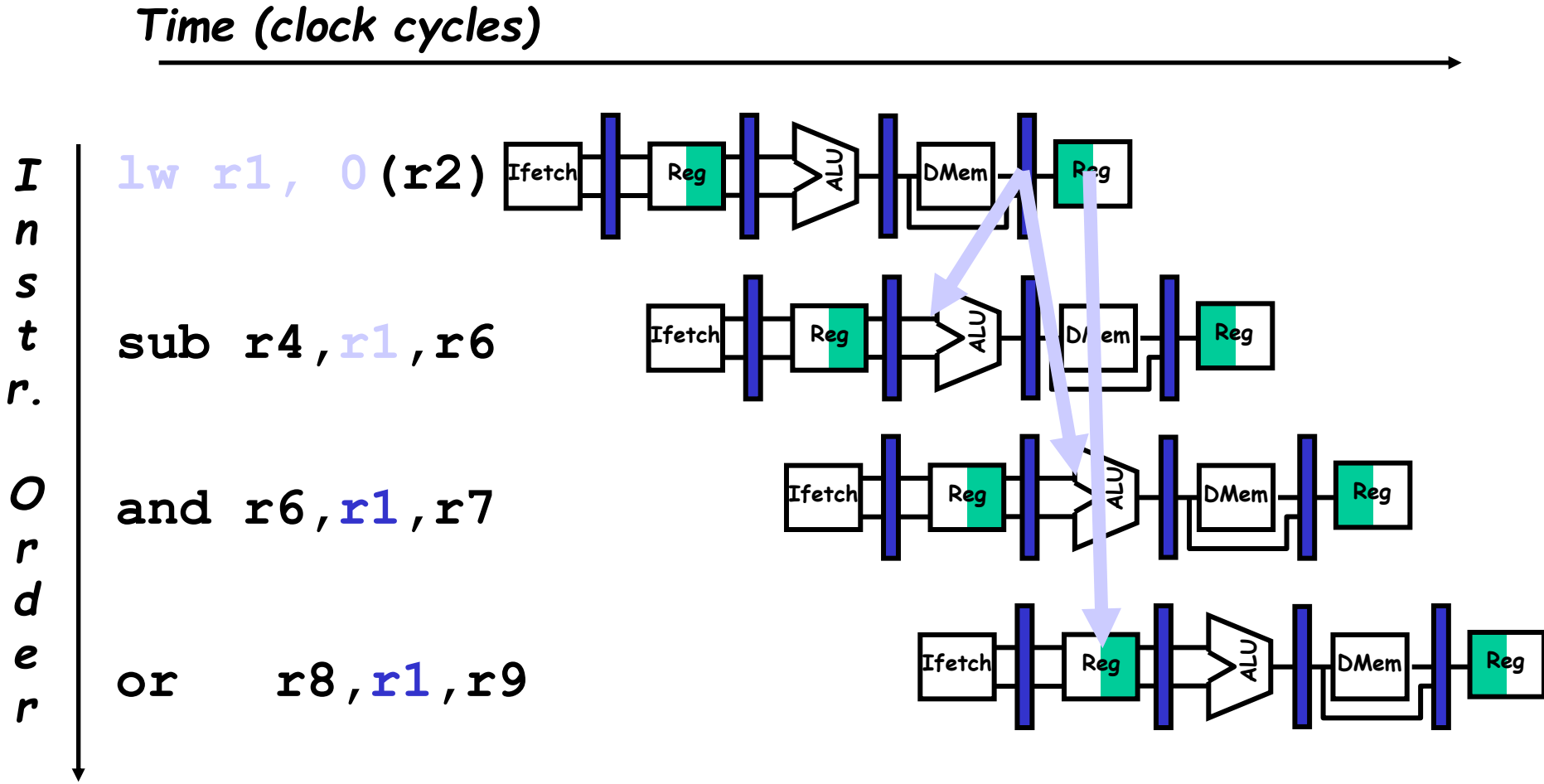


Figure 3.10

Data Hazards

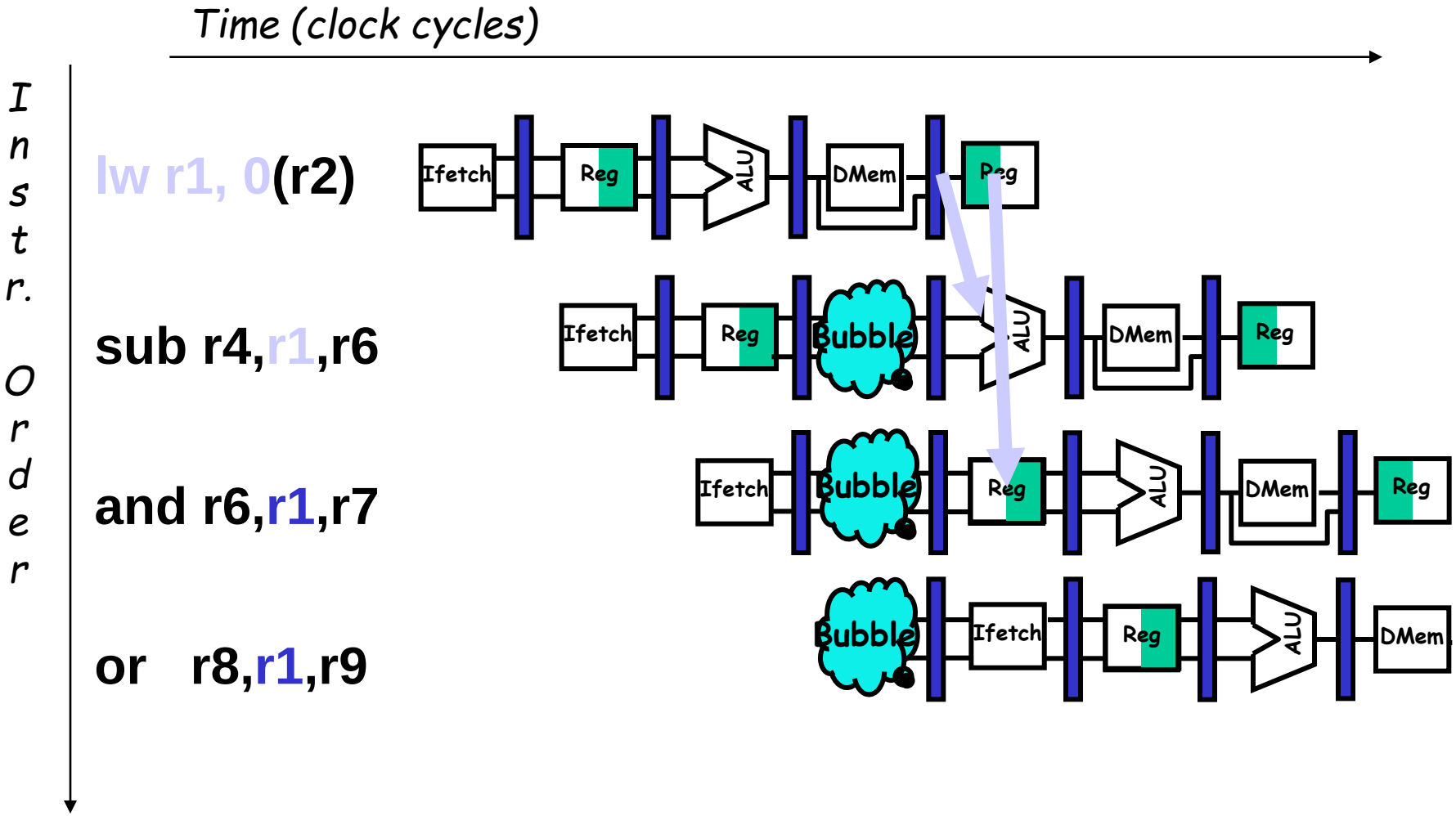
The data isn't loaded until after the MEM stage.



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

Data Hazards

The stall is necessary as shown here.



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

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

Data Hazards

Instruction scheduled by compiler - move instruction in order to reduce stall.

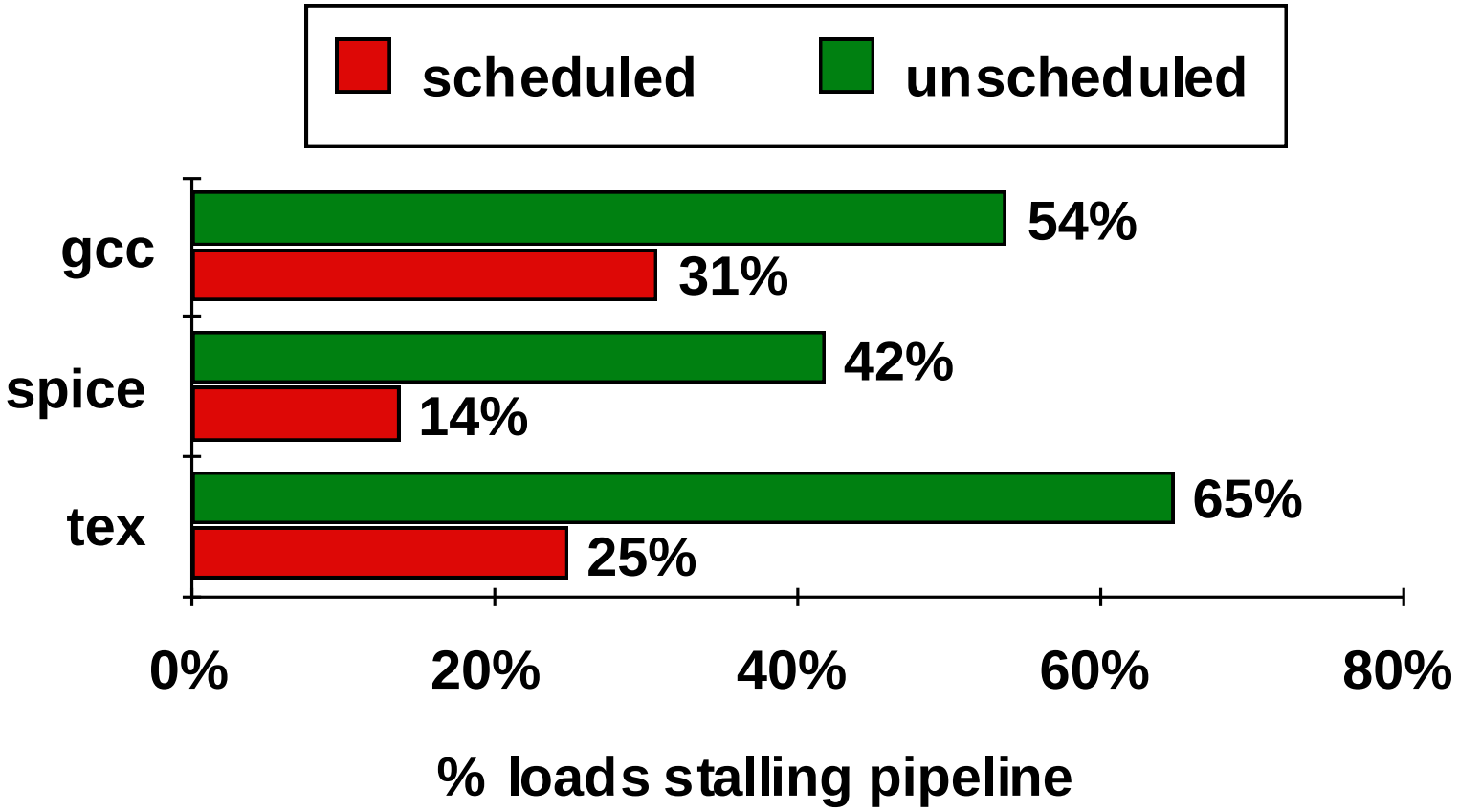
lw Rb, b	code sequence for **a = b+c** before scheduling
lw Rc, c	
Add Ra, Rb, Rc	stall
sw a, Ra	
lw Re, e	code sequence for **d = e+f** before scheduling
lw Rf, f	
sub Rd, Re, Rf	stall
sw d, Rd	

Arrangement of code after scheduling.

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

Data Hazards

Pipeline Scheduling



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

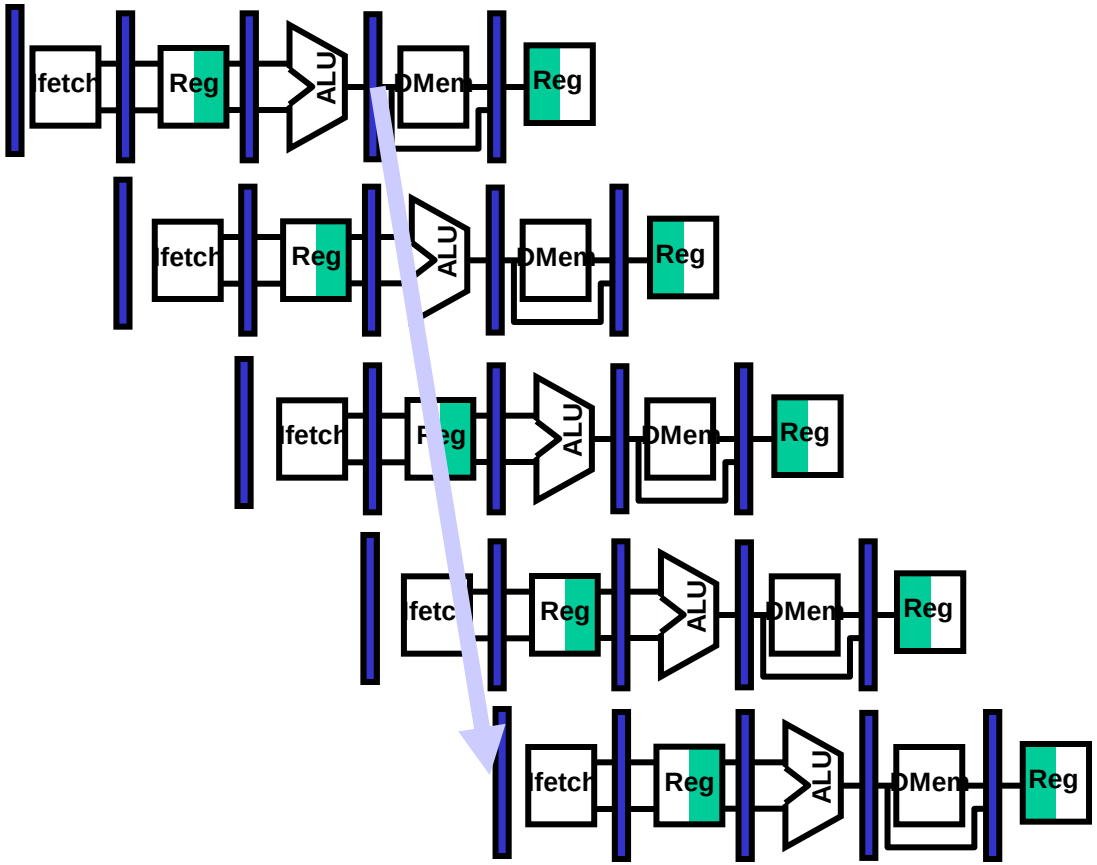
10: beq r1,r3,36

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!**
- **Two part solution to this**
- Determine branch taken or not sooner, AND
- Compute taken branch address earlier

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

#3: Predict Branch Taken

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

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

Evaluating Branch Alternatives

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

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*Speedup v. stall*
Stall pipeline	3	1.42	3.5	1.0
Predict taken	1	1.14	4.4	1.26
Predict not taken	1	1.09	4.5	1.29
Delayed branch	0.5	1.07	4.6	1.31

Conditional & Unconditional = 14%, 65% change PC

Control Hazards

- **Just overlap tasks, and easy if tasks are independent**
- **Speed Up Š Pipeline Depth; if ideal CPI is 1, then:**

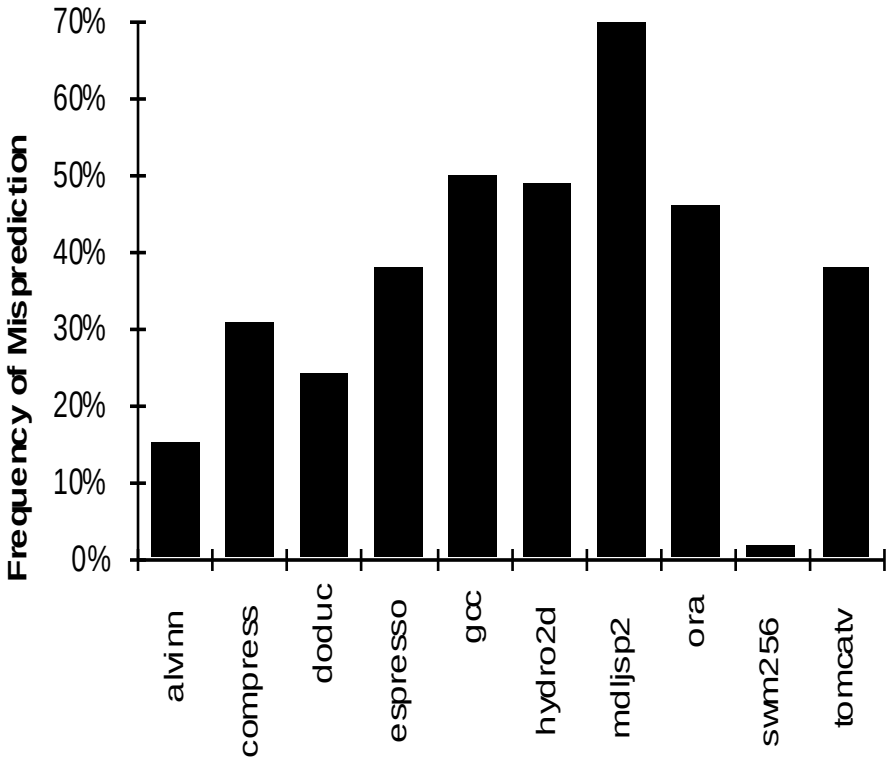
$$\text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle Unpipelined}}{\text{Clock Cycle Pipelined}}$$

- **Hazards limit performance on computers:**
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch, prediction

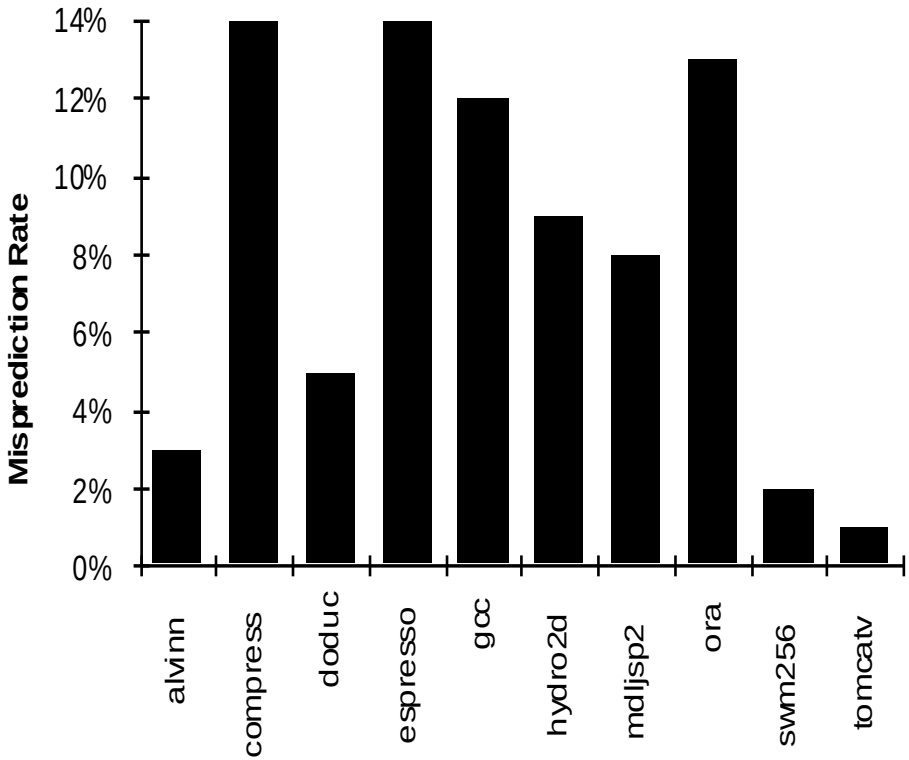
Control Hazards

The compiler can program what it thinks the branch direction will be. Here are the results when it does so.

Compiler "Static" Prediction of Taken/Untaken Branches



Always taken



Taken backwards
Not Taken Forwards

Control Hazards

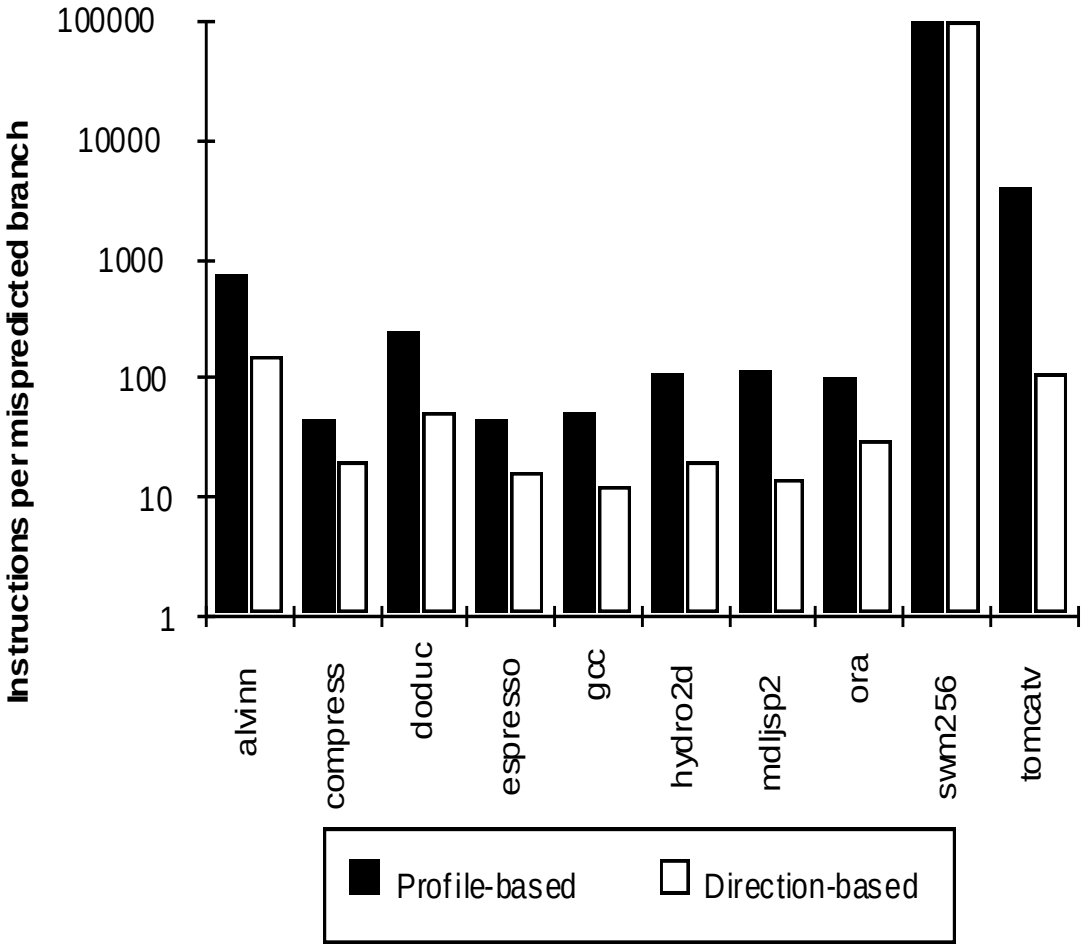
Compiler "Static" Prediction of Taken/Untaken Branches

- **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

Control Hazards

Evaluating Static Branch Prediction Strategies

- **Misprediction ignores frequency of branch**
- **"Instructions between mispredicted branches" is a better metric**



What Makes Pipelining Hard?

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

What Makes Pipelining Hard?

Interrupts cause great havoc!

Examples of interrupts:

- Power failing,
- Arithmetic overflow,
- I/O device request,
- OS call,
- Page fault

Interrupts (also known as: faults, exceptions, traps) often require

- surprise jump (to vectored address)
- linking return address
- saving of PSW (including CCs)
- state change (e.g., to kernel mode)

There are 5 instructions executing in 5 stage pipeline when an interrupt occurs:

- How to stop the pipeline?
- How to restart the pipeline?
- Who caused the interrupt?

What Makes Pipelining Hard?

Interrupts cause great havoc!

What happens on interrupt while in delay slot ?

- **Next instruction is not sequential**

solution #1: save multiple PCs

- **Save current and next PC**
- **Special return sequence, more complex hardware**

solution #2: single PC plus

- **Branch delay bit**
- **PC points to branch instruction**

Stage	*Problem that causes the interrupt*
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic interrupt
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation

What Makes Pipelining Hard?

Interrupts cause great havoc!

- **Simultaneous exceptions in more than one pipeline stage, e.g.,**
  - Load with data page fault in MEM stage
  - Add with instruction page fault in IF stage
  - Add fault will happen BEFORE load fault
- **Solution #1**
  - Interrupt status vector per instruction
  - Defer check until last stage, kill state update if exception
- **Solution #2**
  - Interrupt ASAP
  - Restart everything that is incomplete

Another advantage for state update late in pipeline!

What Makes Pipelining Hard?

Interrupts cause great havoc!

Here's what happens on a data page fault.

```
          1   2   3   4   5   6   7   8   9
i         F   D   X   M   W
i+1           F   D   X   M   W < page fault
i+2               F   D   X   M   W < squash
i+3                   F   D   X   M    W < squash
i+4                       F   D   X    M   W < squash
i+5       trap >                F   D    X   M   W
i+6       trap handler >            F    D   X   M   W
```

What Makes Pipelining Hard?

Complex Addressing Modes and Instructions

- **Address modes: Autoincrement causes register change during instruction execution**
  - Interrupts? Need to restore register state
  - Adds WAR and WAW hazards since writes are no longer the last stage.

- **Memory-Memory Move Instructions**
  - Must be able to handle multiple page faults
  - Long-lived instructions: partial state save on interrupt

- **Condition Codes**

Handling Multi-cycle Operations

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

Multi-cycle instructions also lead to pipeline complexity.

A very lengthy instruction causes everything else in the pipeline to wait for it.

Multi-Cycle Operations

Floating Point

Floating point gives long execution time.

This causes a stall of the pipeline.

It's possible to pipeline the FP execution unit so it can initiate new instructions without waiting full latency. Can also have multiple FP units.

FP Instruction	*Latency*	*Initiation Rate*
Add, Subtract	4	3
Multiply	8	4
Divide	36	35
Square root	112	111
Negate	2	1
Absolute value	2	1
FP compare	3	2

Multi-Cycle Operations

Floating Point

Divide, Square Root take 10X to 30X longer than Add

- Interrupts?
- Adds WAR and WAW hazards since pipelines are no longer same length

	1	2	3	4	5	6	7	8	9	10	11
i	IF	ID	EX	MEM	WB						
I + 1		IF	ID	EX	EX	EX	EX	MEM	WB		
I + 2			IF	ID	EX	MEM	WB				
I + 3				IF	ID	EX	EX	EX	EX	MEM	WB
I + 4					IF	ID	EX	MEM	WB		
I + 5						IF	ID	--	--	EX	EX
I + 6							IF	--	--	ID	EX

Notes:
I + 2: no WAW, but this complicates an interrupt
I + 4: no WB conflict
I + 5: stall forced by structural hazard
I + 6: stall forced by in-order issue

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