Quantitative techniques in computer design

Nilanjan Byabarta

UEM Kolkata

Agenda

measuring and reporting performance

Performance Measurement

Recall performance is function of

- CPI: cycles per instruction
- Clock cycle
- Instruction count

Reducing any of the 3 factors will lead to improved performance

Computer performance is the amount of work accomplished by a computer system. The word performance in computer performance means “How well is the computer doing the work it is supposed to do?”. It basically depends on response time, throughput and execution time of a computer system.

Response time is the time from start to completion of a task. This also includes:

- Operating system overhead.
- Waiting for I/O and other processes
- Accessing disk and memory
- Time spent executing on the CPU or execution time.

Throughput is the total amount of work done in a given time.

CPU execution time is the total time a CPU spends computing on a given task. It also excludes time for I/O or running other programs. This is also referred to as simply CPU time.

Performance is determined by execution time as performance is inversely proportional to execution time.

```
Performance = (1 / Execution time)

And,

(Performance of A / Performance of B) = (Execution Time of B / Execution
Time of A)
```

Execution time = CPU clock cycles x clock cycle time

Since clock cycle time and clock rate are reciprocals, so,

Execution time = CPU clock cycles / clock rate

The number of CPU clock cycles can be determined by,

CPU clock cycles = (No. of instructions / Program) x (Clock cycles / Instruction) = Instruction Count x CPI

```
Execution time = Instruction Count x CPI x clock
cycle time

= Instruction Count x CPI / clock rate
```

How to Improve Performance?

To improve performance you can either:

•Decrease the CPI (clock cycles per instruction) by using new Hardware.

•Decrease the clock time or Increase clock rate by reducing propagation delays or by use pipelining.

•Decrease the number of required cycles or improve ISA or Compiler.

A Few Words About Where We Are Headed

Performance = 1 / Execution time   simplified to   1 / CPU execution time

CPU execution time = Instructions × CPI / (Clock rate)

Performance = Clock rate / (Instructions × CPI)

Try to achieve CPI = 1 with clock that is as high as that for CPI > 1 designs; is CPI < 1 feasible?

Design memory & I/O structures to support ultrahigh-speed CPUs

Define an instruction set; make it simple enough to require a small number of cycles and allow high clock rate, but not so simple that we need many instructions, even for very simple tasks ✓

Design hardware for CPI = 1; seek improvements with CPI > 1

Design ALU for arithmetic & logic ops ✓

Performance Measurement

- First step is to apply concept of pipelining to the instruction execution process
  - Overlap computations
- What does this do?
  - Decrease clock cycle
  - Decrease effective CPU time compared to original clock cycle

Introduction

Principles that are useful in design and analysis of computers:

- Make the common case fast !
    - If a design trade-off is necessary, favor the frequent case (which is often simpler) over the infrequent case.
    -
    - For example, given that overflow in addition is infrequent, favor optimizing the case when no overflow occurs.

Introduction

Typical performance metrics:

- Response time
- Throughput

Speedup of X relative to Y

- Execution time$_Y$ / Execution time$_X$

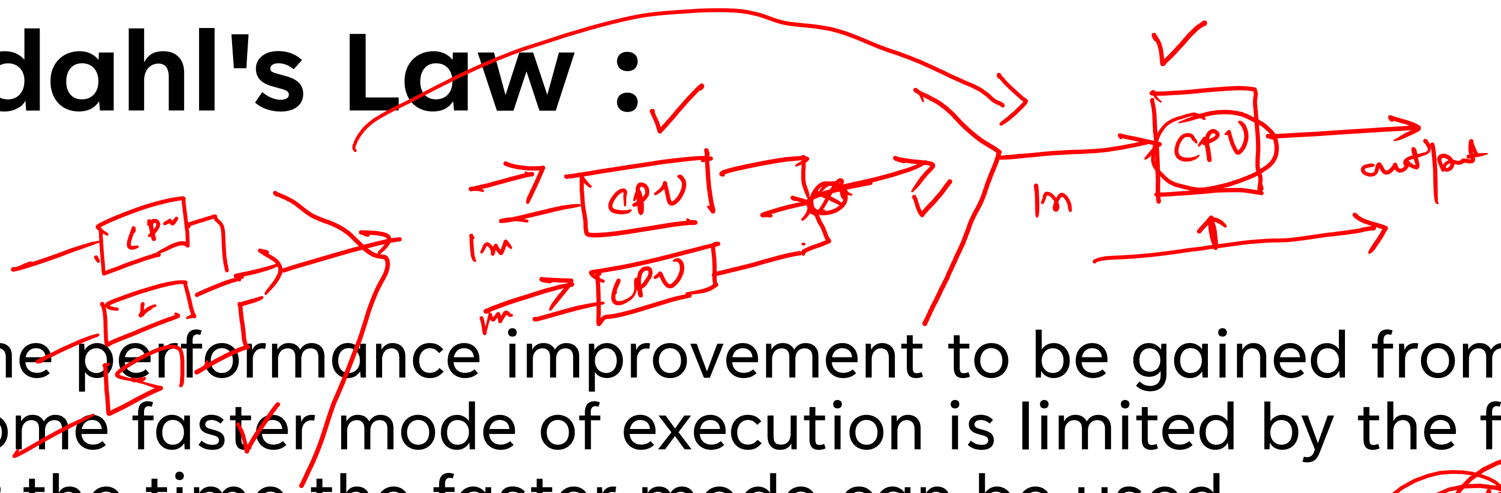
Execution time

- Wall clock time: includes all system overheads
- CPU time: only computation time

Objective:

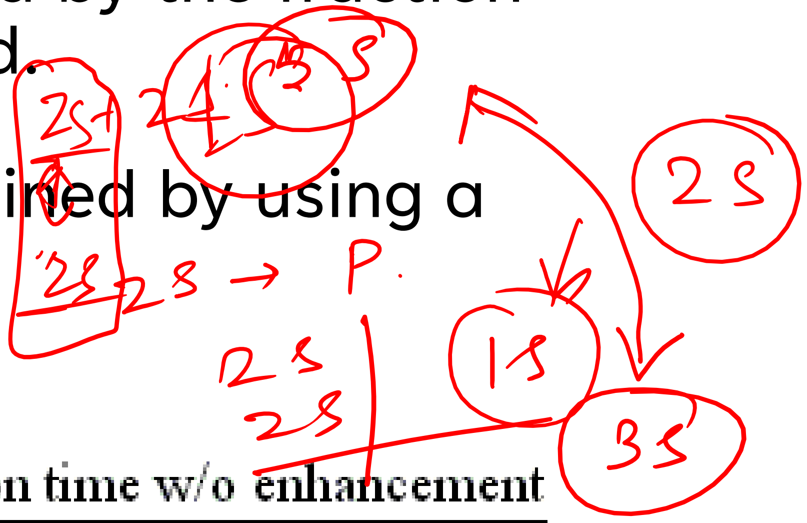
- Determine the frequent case.
- Determine how much improvement in performance is possible by making it faster.
- Amdahl's law can be used to quantify the latter given that we have information concerning the former.

Amdahl's Law :



- The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

- Amdahl's law defines the speedup obtained by using a particular feature:



$$\text{Speedup} = \frac{\text{Performance with enhancement}}{\text{Performance without enhancement}} = \frac{\text{Execution time w/o enhancement}}{\text{Execution time with enhancement}}$$

Areas of growth

- Two factors:

•Fraction $_{\text{enhanced}}$: Fraction of compute time in original machine that can be converted to take advantage of the enhancement.

  - Always <= 1.

•Speedup $_{\text{enhanced}}$: Improvement gained by enhanced execution mode:

$$\frac{\text{Time of original mode}}{\text{Time of enhanced mode}} \qquad \text{Always} > 1$$

• ”

- *Execution time using original machine with enhancement:*

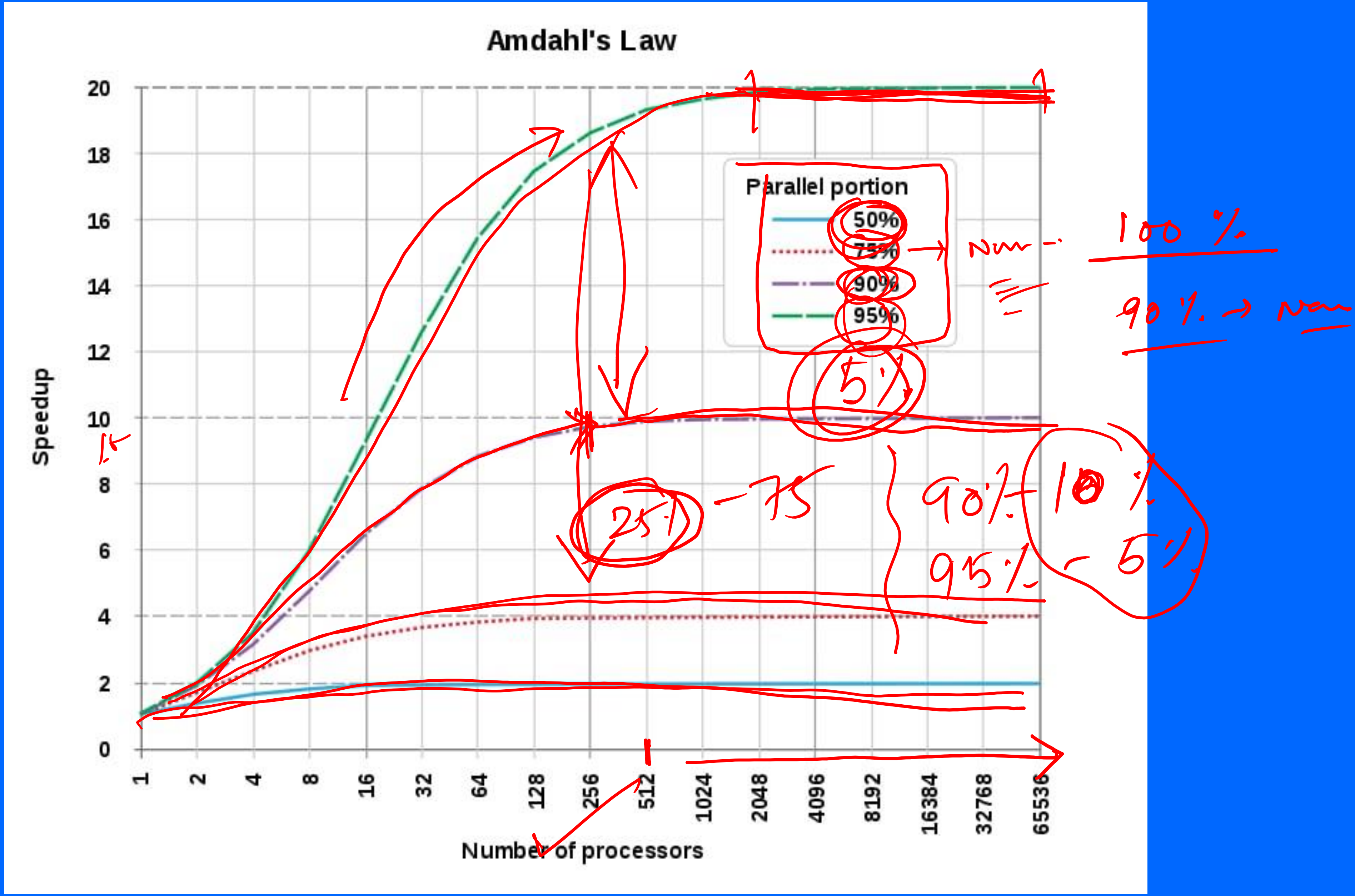
$P,$

$$\text{Exec time}_{new} = \underbrace{\text{Exec time}_{old} \times \left((1 - \mathit{Fraction}_{enhanced})\right.}_{\text{Time spent using UNenhanced mode}} \left. + \frac{\mathit{Fraction}_{enhanced}}{\mathit{Speedup}_{enhanced}} \right)$$

S

Time spent using enhancement

$$= T \times \left\{ \left\{ (1-P) \right\} + \frac{P}{S} \right\}$$


Amdahl's Law
Speedup
Number of processors
Parallel portion
50%
75%
90%
95%
0
2
4
6
8
10
12
14
16
18
20
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
5%
25%
75
90% - 10%
95% - 5%
Non - 100%
90% → Non

T = Total execution time of the system some modules

→ both enhancement Part + Non-enhancement Mode

total time with an with all the enhancement

P = Part of Program where enhancement happened.

→ $(1-P)$ = Part of Program where enhancement did not happen.

S = speed up value / enhancement portion.

$$T = PT + (1-P)T$$

$$= \underline{\underline{(1-P)T}} + \boxed{P \cdot T}$$

S = speed up $\rightarrow$ $P.T$ $\rightarrow$ $\frac{PT}{S} = \frac{P}{S} \cdot T$

$\downarrow$ After the enhancement has happened.

$$T_S = (1-P)T + \frac{P}{S}T$$

$P = 10$

$s = 2$

$\frac{P}{S} = \frac{10}{2} = \boxed{5}$

$$\frac{T.W}{T_S W} = \frac{T}{T_{SW}} = \frac{\cancel{T}}{\cancel{T}}$$

$$= \frac{T}{(1-P)T + \frac{P}{S}T}$$

Speed up ~~latency~~ $= \boxed{\frac{1}{(1-P) + \frac{P}{S}}} \rightarrow$ enhancement

$(\downarrow) \rightarrow$ Parallelization as happen.

speed up = 30% $\longrightarrow$ $\boxed{.33}$ P $\rightarrow$ $(1-P)$

$= (1-.33)$

$= .77$

twice $\qquad S = 2$

1.

$\underline{\underline{1.086}}$

$= \text{latency} = \dfrac{1}{.7 + \dfrac{.3}{2}}$

$= \dfrac{1}{.7 + .015}$

$= [1.2]$

$\boxed{1.3}$

$100\left(1 - \dfrac{1}{\text{latency}}\right)$

$= x/o$

•*Speedup* overall *using Amdahl's Law:*

$$Speedup_{overall} = \frac{ExecTime_{old}}{ExecTime_{new}} = \frac{1}{\left((1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}\right)}$$

CPU Performance Equation:

- Often it is difficult to measure the improvement in time using a new enhancement directly.

- A second method that decomposes the CPU execution time into three components makes this task simpler.

- CPU Performance Equation:

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

•*where, for example, Clock cycle time = 2ns for a 500MHz Clock rate.*

CPU Performance Equation:

- An alternative to "number of clock cycles" is "number of instructions executed" or Instruction Count (IC).

- Given both the "number of clock cycles" and IC of a program, the average Clocks Per Instruction (CPI) is given by:

$$\text{CPI} = \frac{\text{CPU clock cycles of a program}}{\text{IC}}$$

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{Clock cycle time} = \frac{\text{IC} \times \text{CPI}}{\text{Clock rate}}$$

CPU Performance Equation:

- One difficulty: It is difficult to change one in isolation of the others:
- Clock cycle time: Hardware and Organization.
- CPI: Organization and Instruction set architecture.
- Instruction count: Instruction set architecture and Compiler technology.

$$\text{CPU time} = \left(\sum_{i=1}^{n} CPI_i \times IC_i \right) \times \text{Clock cycle time}$$

- *where IC $_i$ represents number of time instruction i is executed in a program and CPI $_i$ represents the average number of clock cycles for instruction i.*

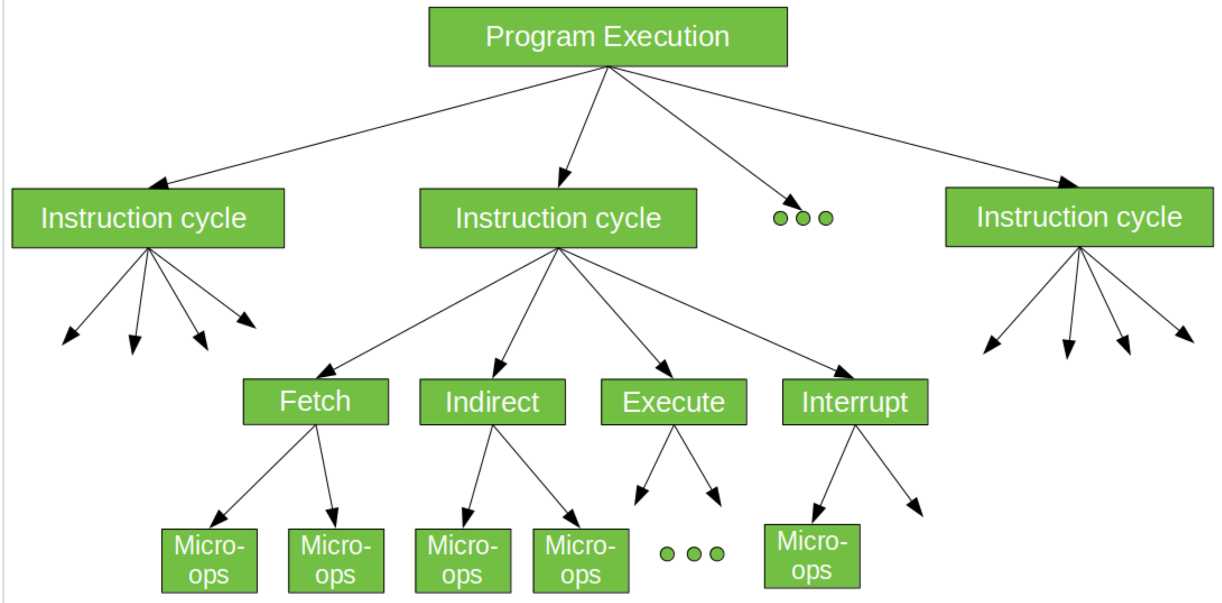
- *Why isn't CPI $_i$ a constant*

(million instruction per second) is NOT an alternative metric to time.

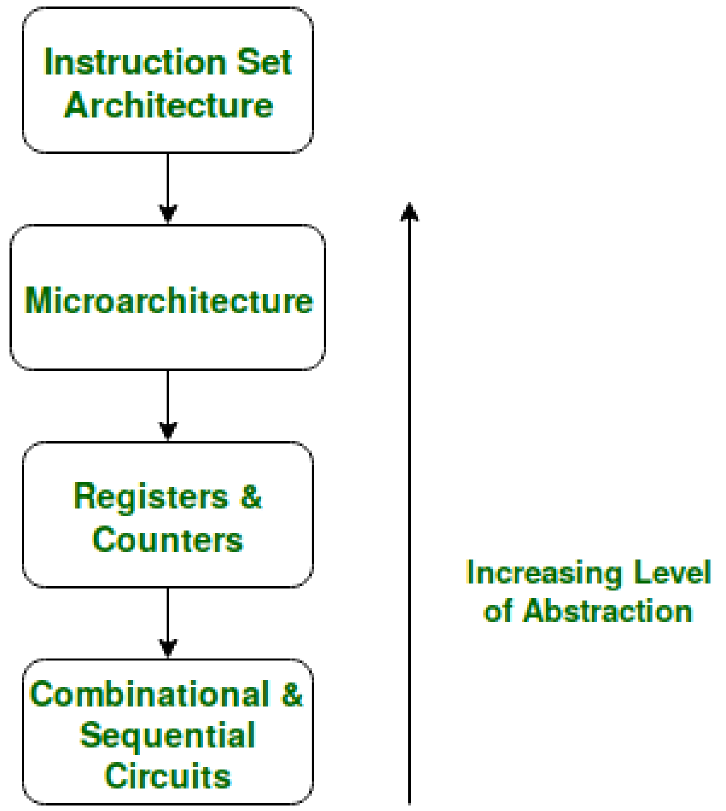
$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Exec Time} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

ISA(Instruction set Architecture)

In computer science, an instruction set architecture (ISA), also called computer architecture, is **an abstract model of a computer**. A device that executes instructions described by that ISA, such as a central processing unit (CPU), is called an implementation.

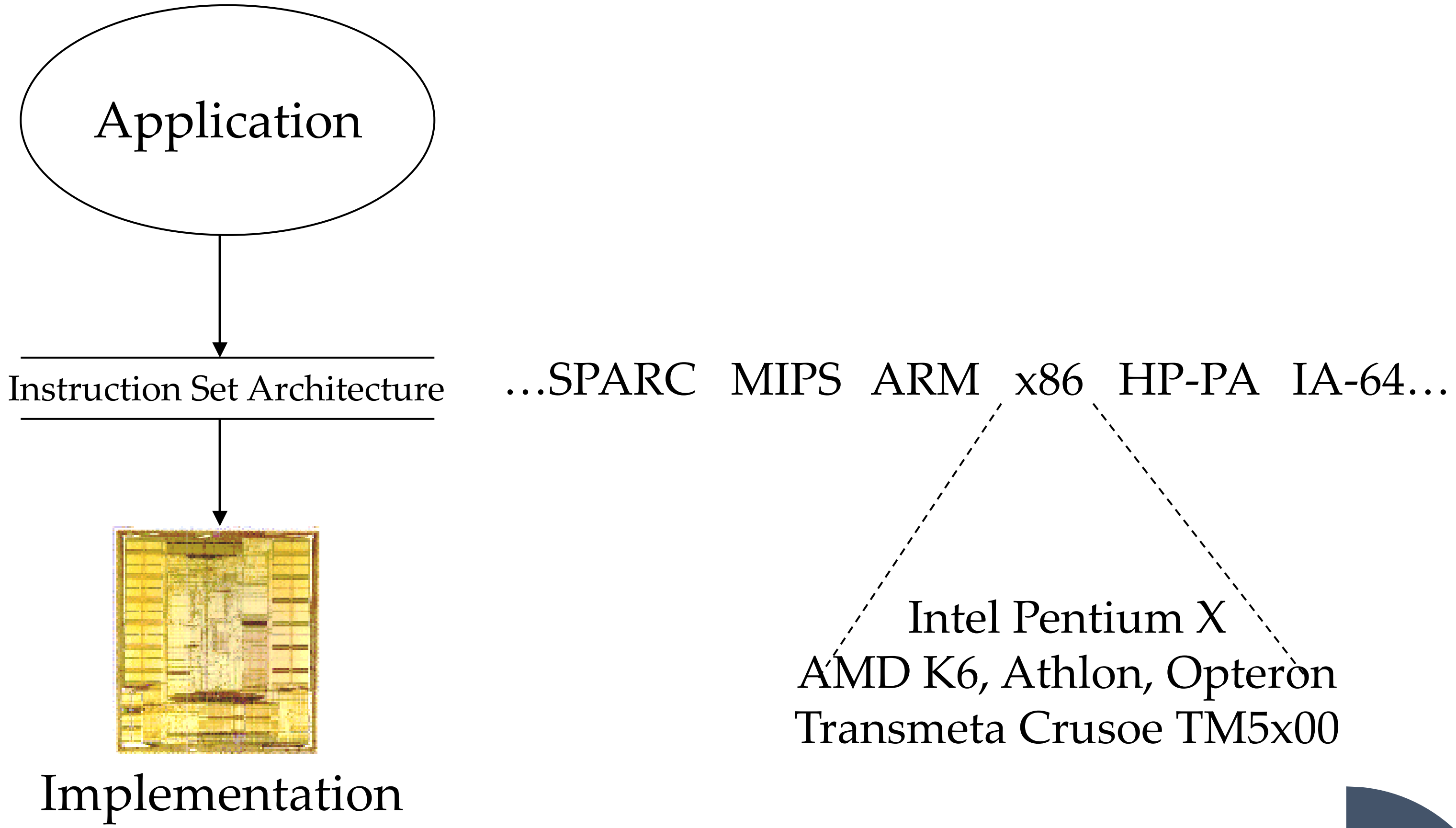


ISA describes the **design of a Computer** in terms of the **basic operations** it must support. The ISA is not concerned with the implementation-specific details of a computer. It is only concerned with the set or collection of basic operations the computer must support.


Instruction Set Architecture
Microarchitecture
Registers & Counters
Combinational & Sequential Circuits
Increasing Level of Abstraction

- The ISA defines the **types of instructions** to be supported by the processor. Based on the type of operations they perform MIPS Instructions are classified into 3 types:
- **Arithmetic/Logic Instructions:**
These Instructions perform various Arithmetic & Logical operations on one or more operands.
- **Data Transfer Instructions:**
These instructions are responsible for the transfer of instructions from memory to the processor registers and vice versa.
- **Branch and Jump Instructions:**
These instructions are responsible for breaking the sequential flow of instructions and jumping to instructions at various other locations, this is necessary for the implementation of *functions* and *conditional statements*.

Instruction Set Architecture



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

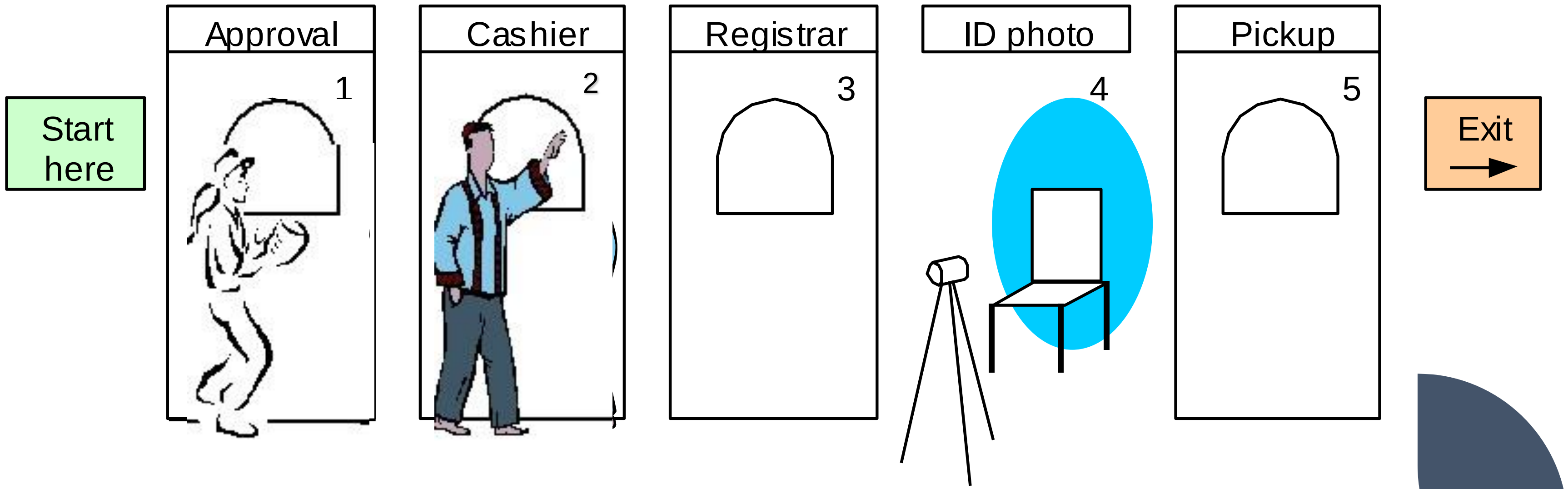
Decoding uniformity

Pipelining Concepts

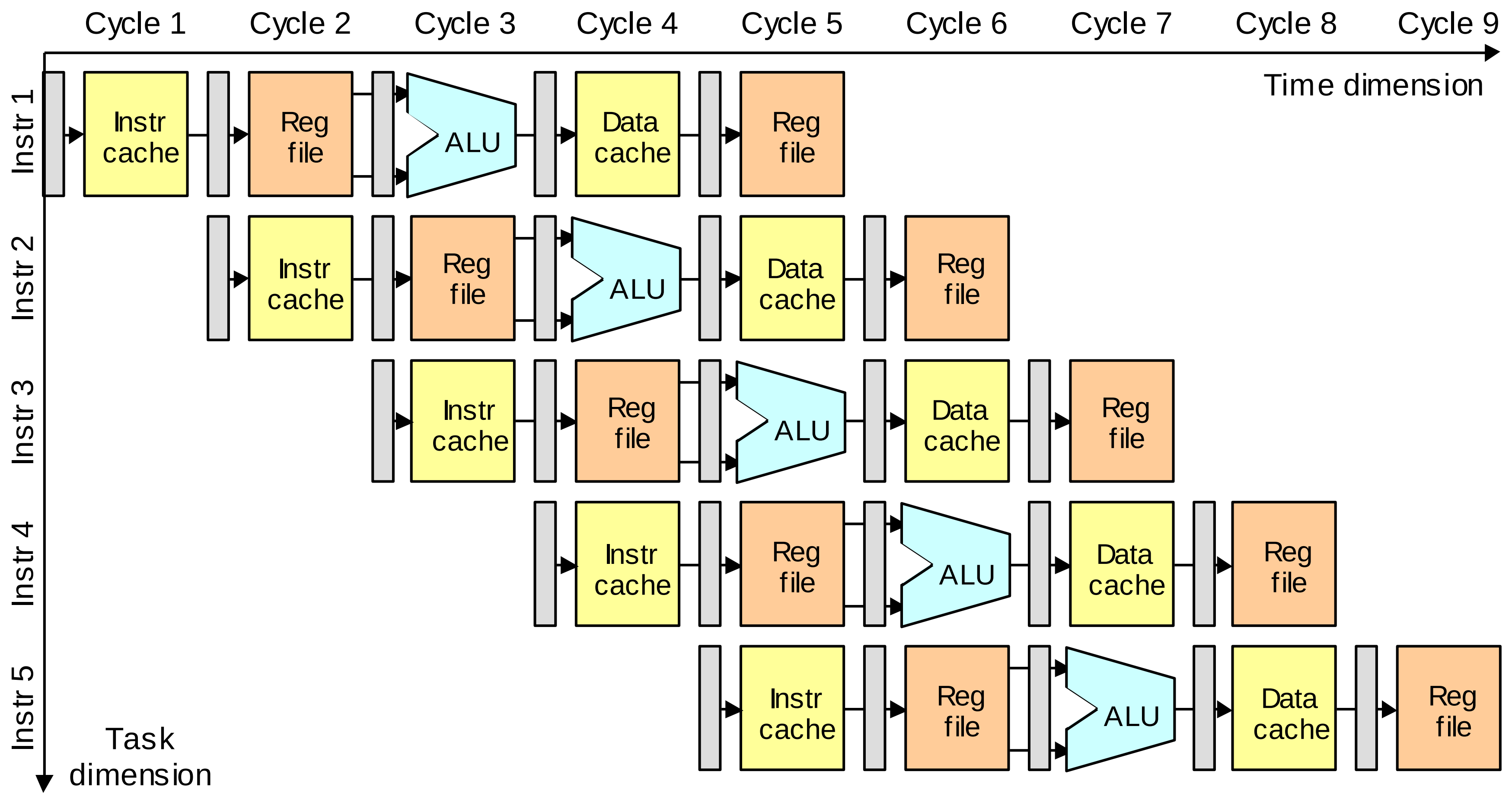
Strategies for improving performance

1 – Use multiple independent data paths accepting several instructions that are read out at once: *multiple-instruction-issue* or *superscalar*

2 – Overlap execution of several instructions, starting the next instruction before the previous one has run to completion: *(super)pipelined*

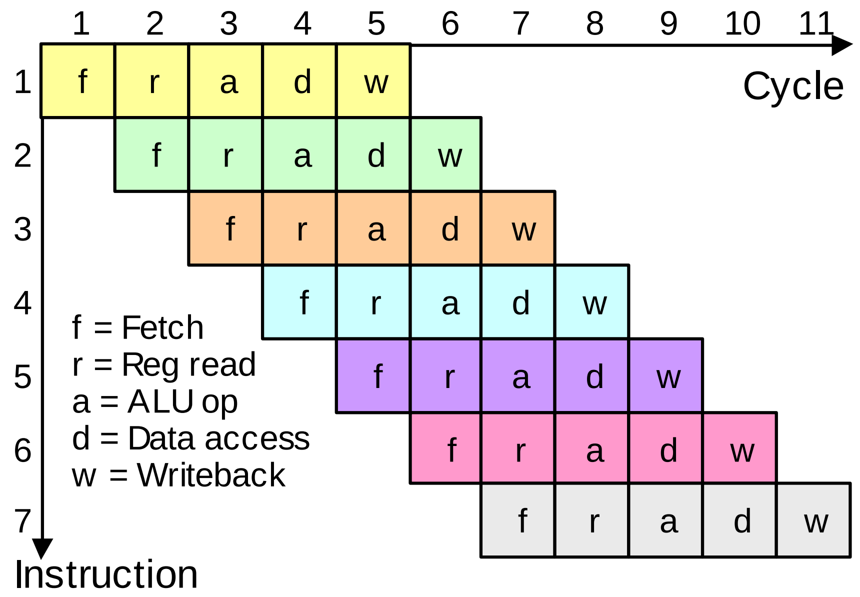


Pipelined Instruction Execution

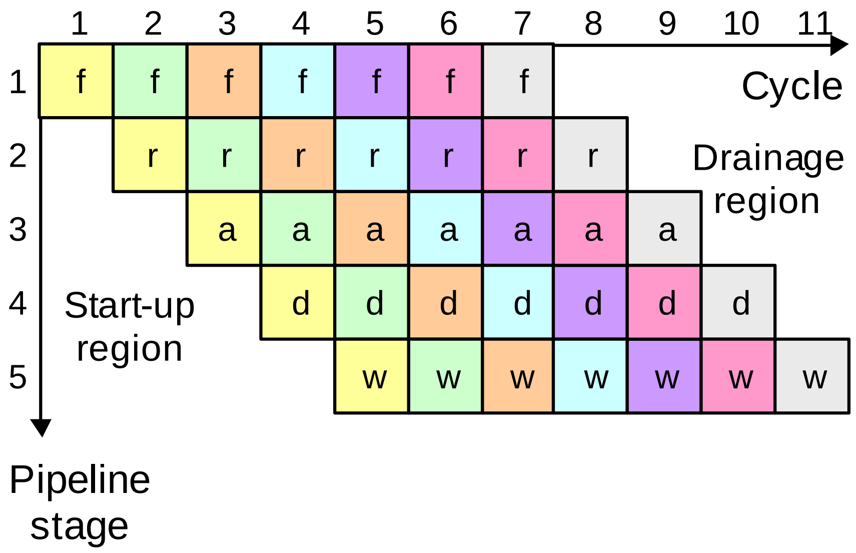


Alternate Representations of a Pipeline

Except for start-up and drainage overheads, a pipeline can execute one instruction per clock tick; IPS is dictated by the clock frequency



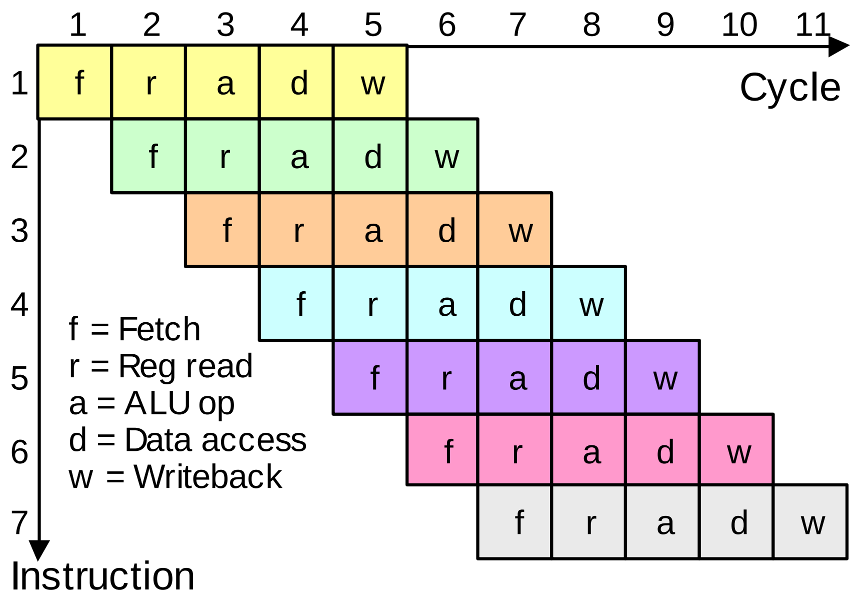
(a) Task-time diagram



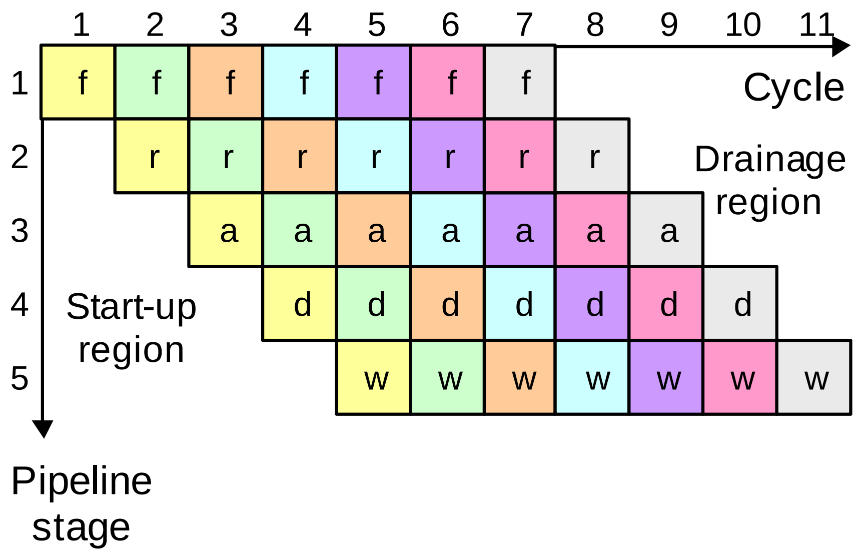
(b) Space-time diagram

Alternate Representations of a Pipeline

Except for start-up and drainage overheads, a pipeline can execute one instruction per clock tick; IPS is dictated by the clock frequency



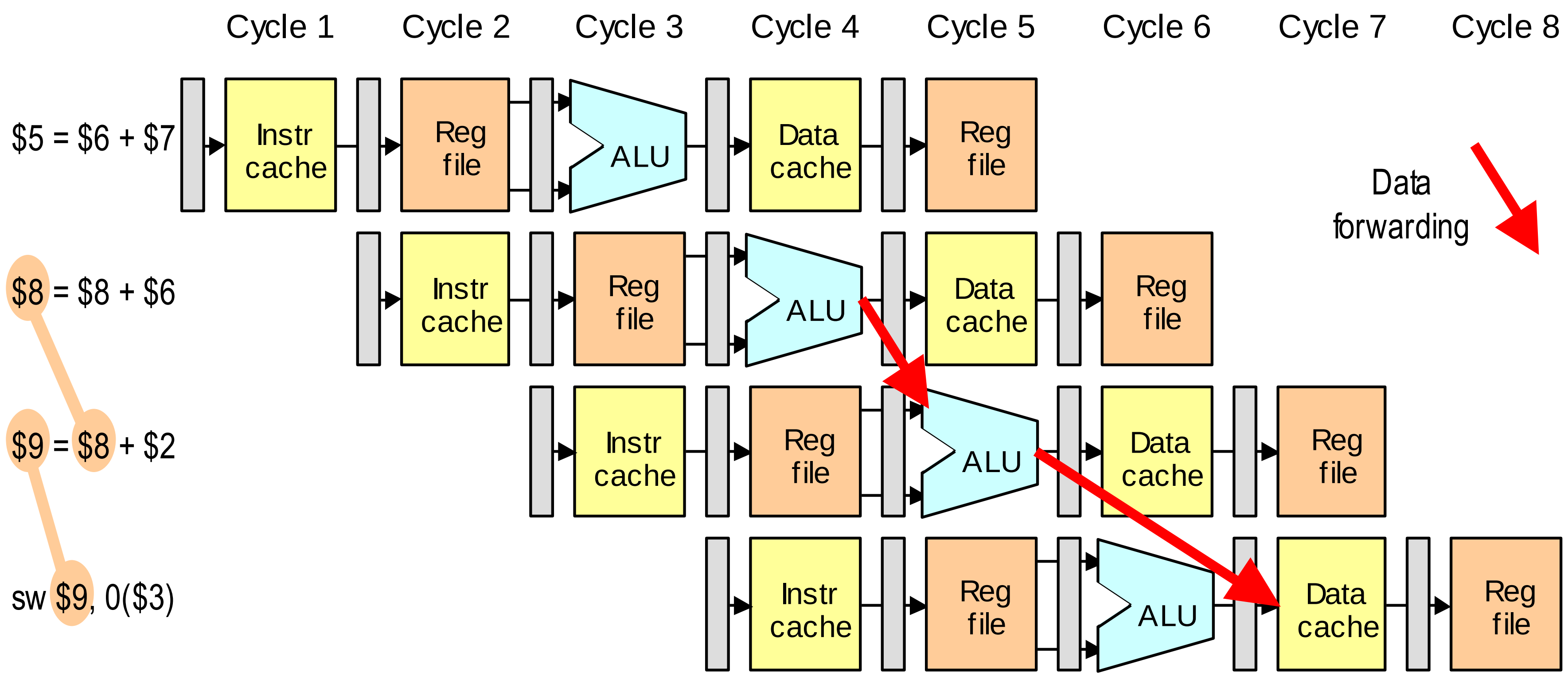
(a) Task-time diagram

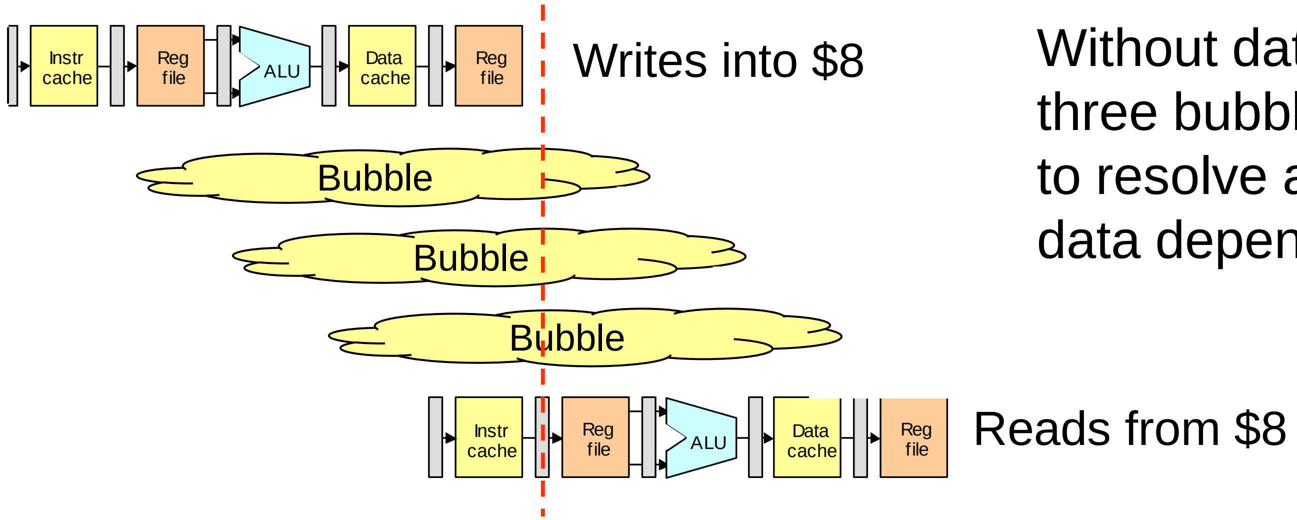


(b) Space-time diagram

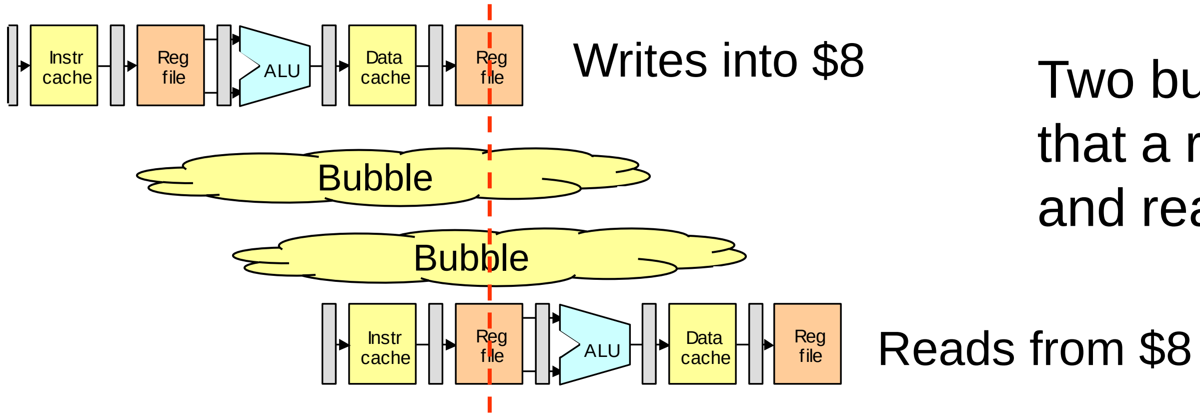
15.2 Pipeline Stalls or Bubbles

First type of data dependency



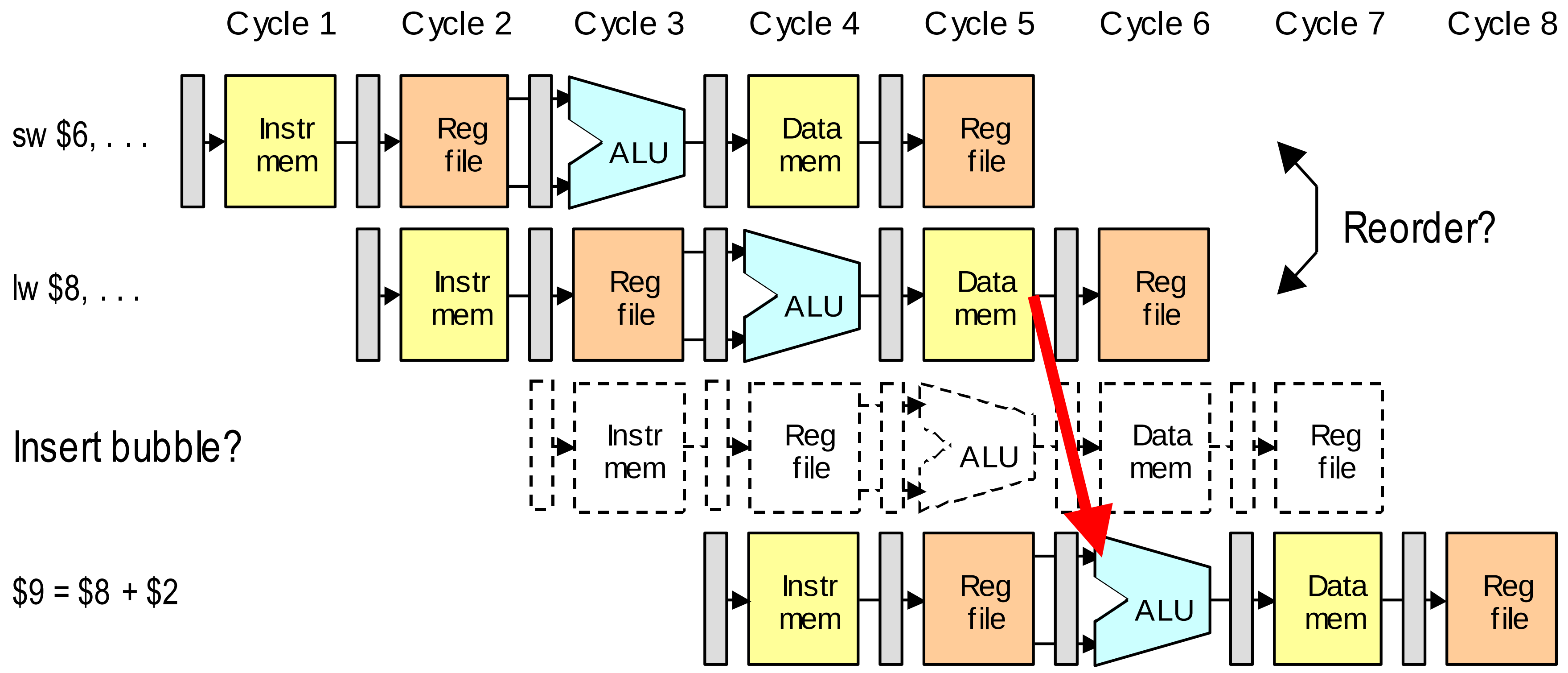


Without data forwarding, three bubbles are needed to resolve a read-after-write data dependency



Two bubbles, if we assume that a register can be updated and read from in one cycle

Second Type of Data Dependency



Without data forwarding, three (two) bubbles are needed to resolve a read-after-load data dependency

Read-after-load data dependency and its possible resolution through bubble insertion and data forwarding.

Introduction

What Is Pipelining

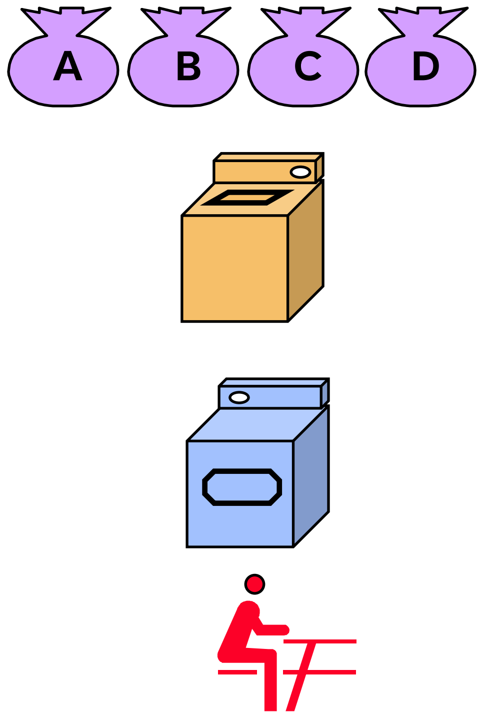
Laundry Example

Ann, Brian, Cathy, Dave
each have one load of clothes
to wash, dry, and fold

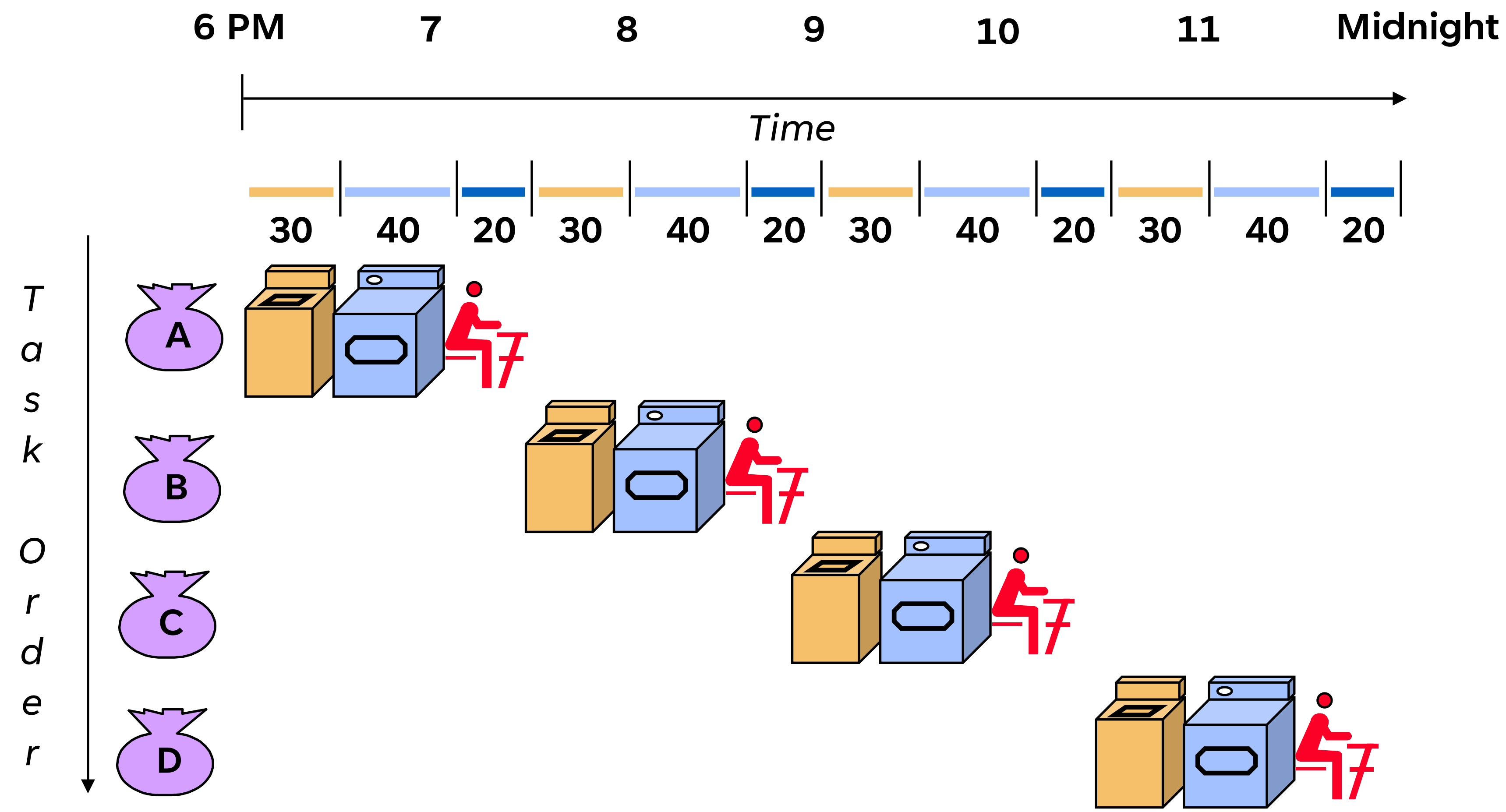
Washer takes 30 minutes

Dryer takes 40 minutes

“Folder” takes 20 minutes



What Is Pipelining

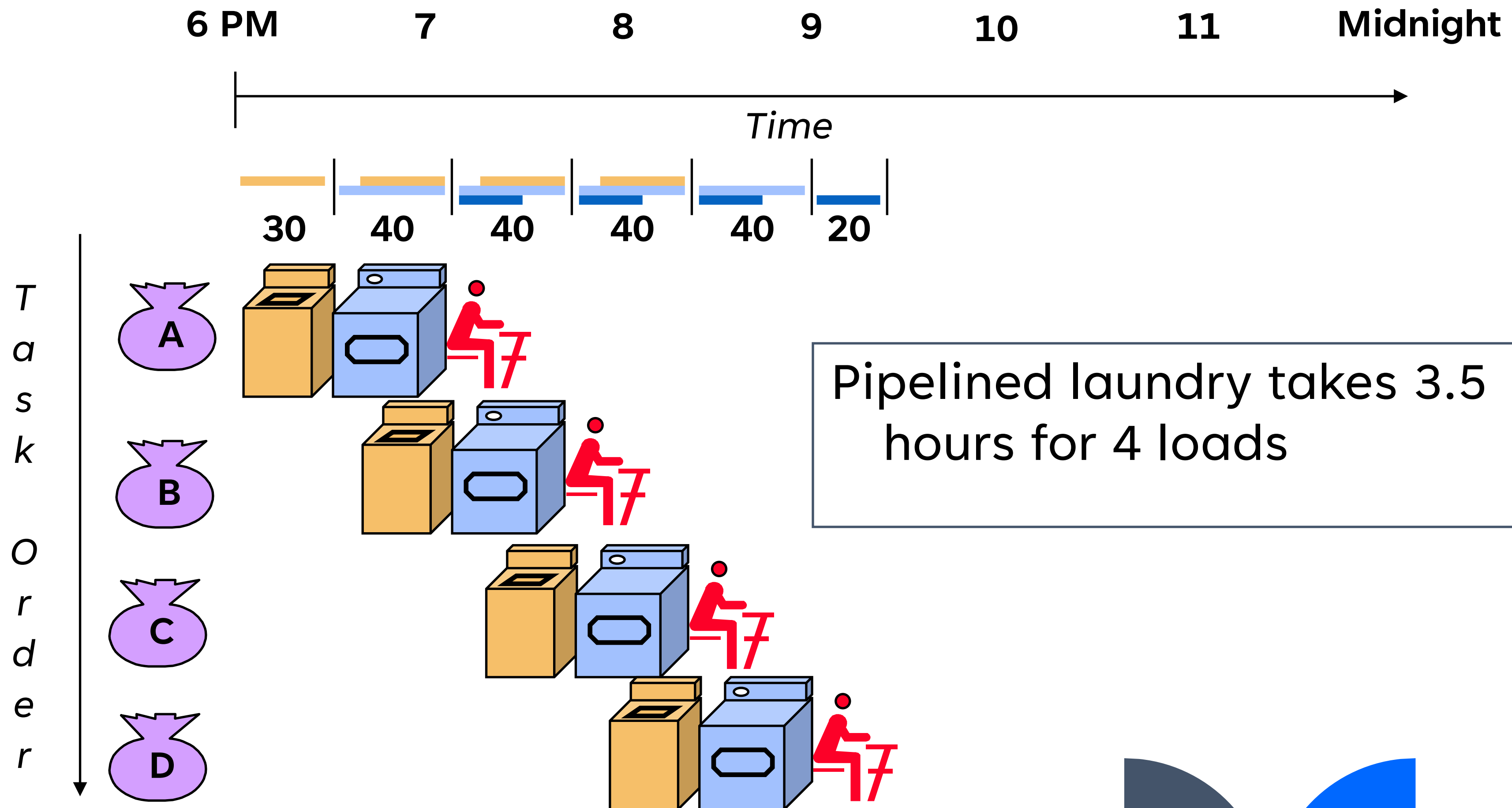


Sequential laundry takes 6 hours for 4 loads

If they learned pipelining, how long would laundry take?

What Is Pipelining

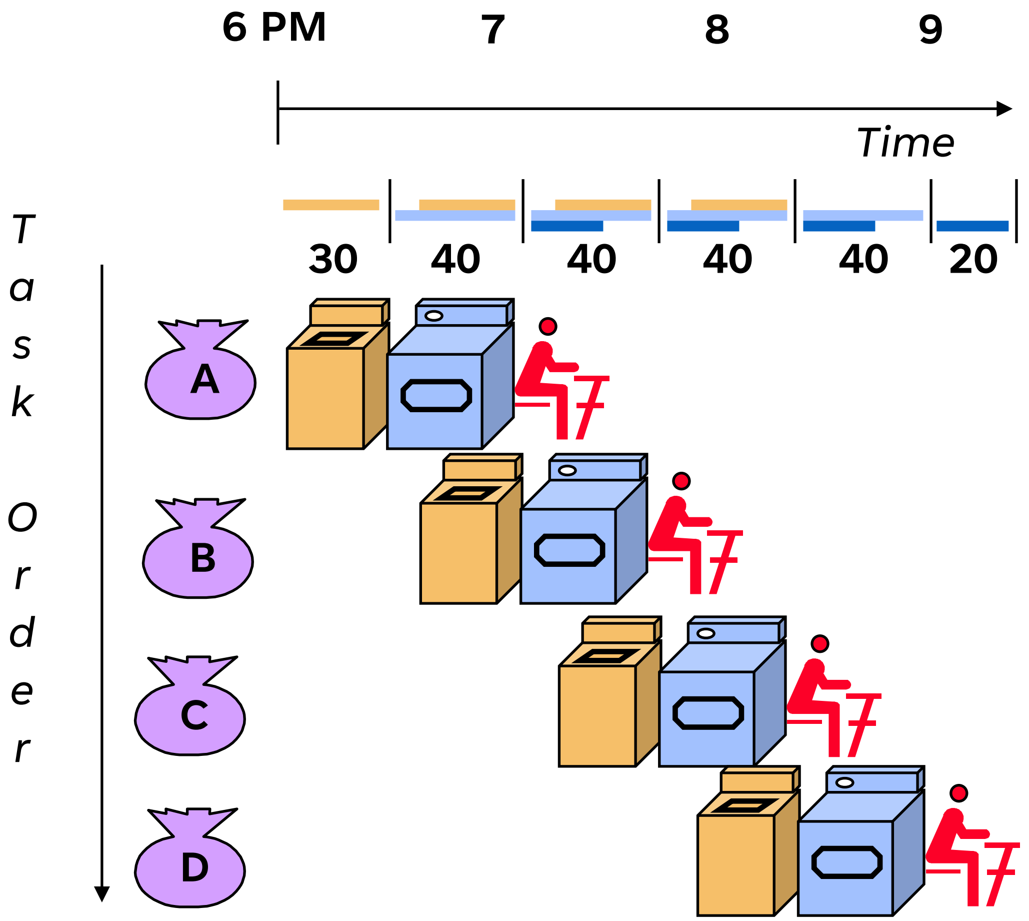
Start work ASAP



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

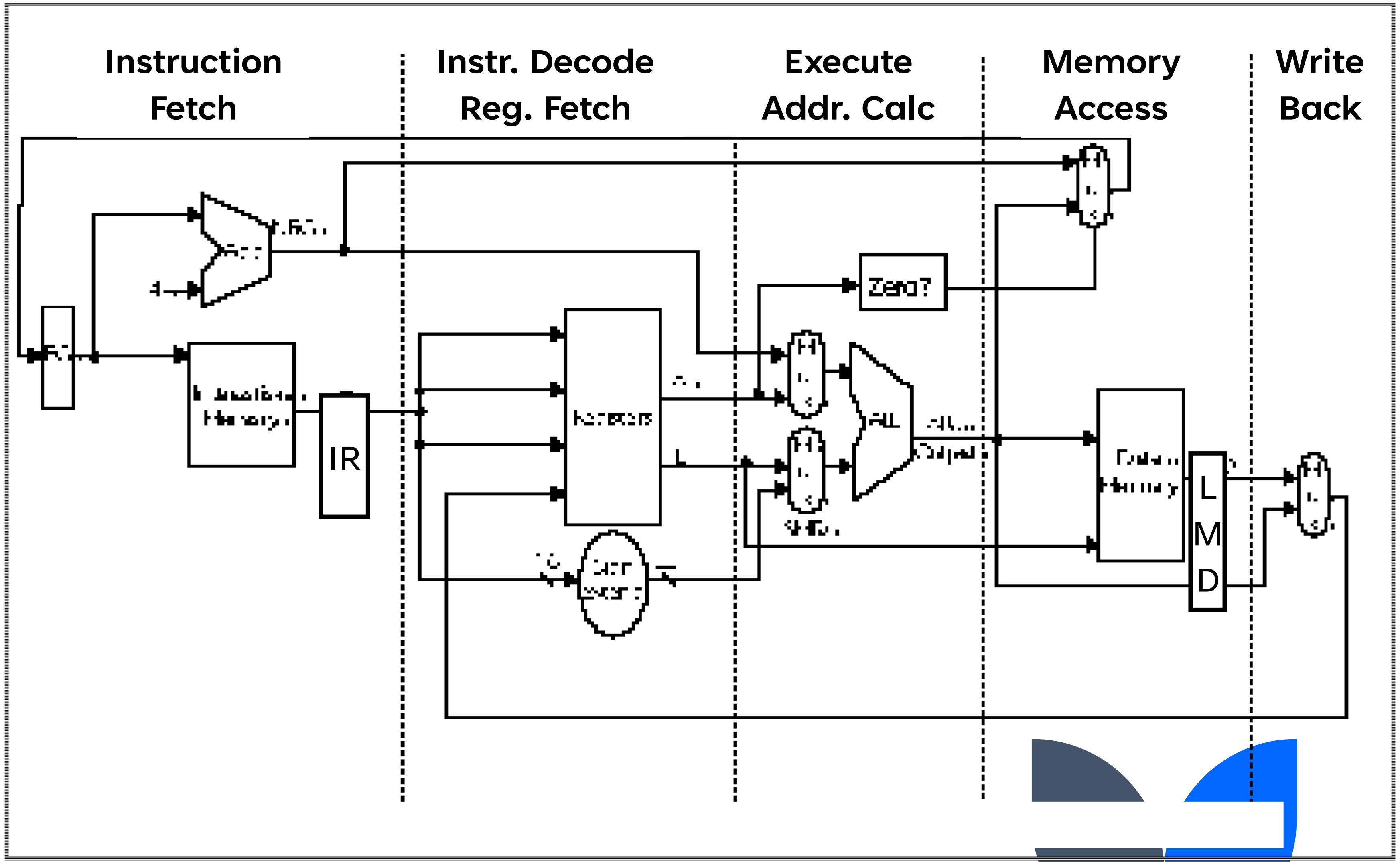
Potential speedup = Number pipe stages

Unbalanced lengths of pipe stages reduces speedup

Time to "fill" pipeline and time to "drain" it reduces speedup

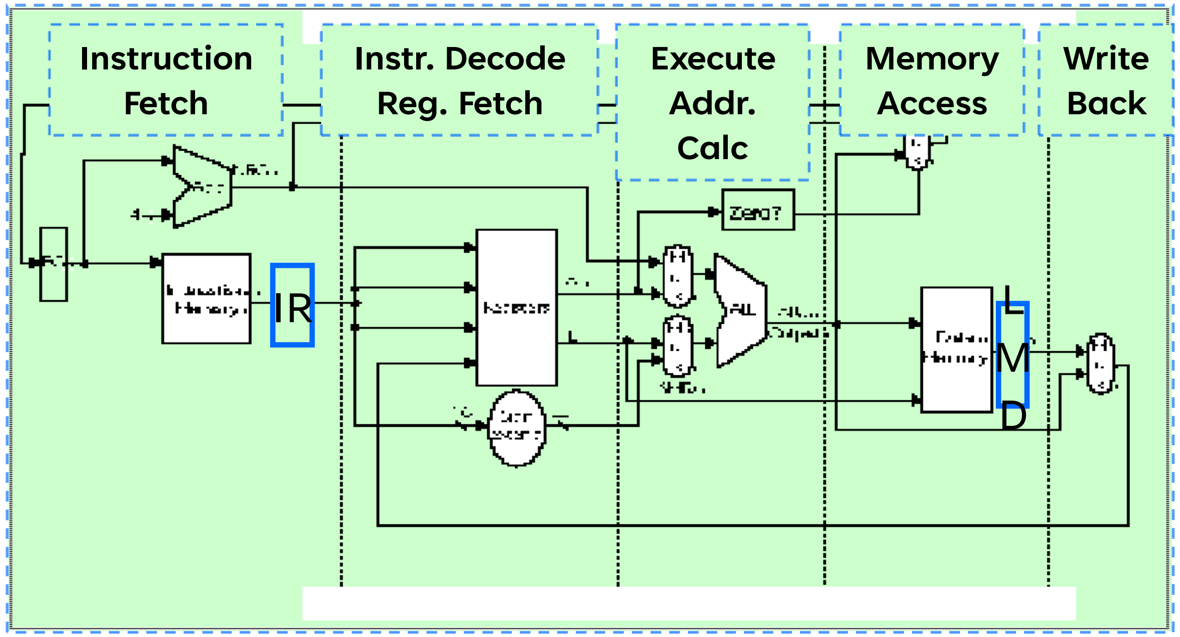
What Is Pipelining

MIPS Without Pipelining



What Is Pipelining

MIPS Functions



Passed To Next Stage

IR <- Mem[PC]

NPC <- PC + 4

Instruction Fetch (IF):

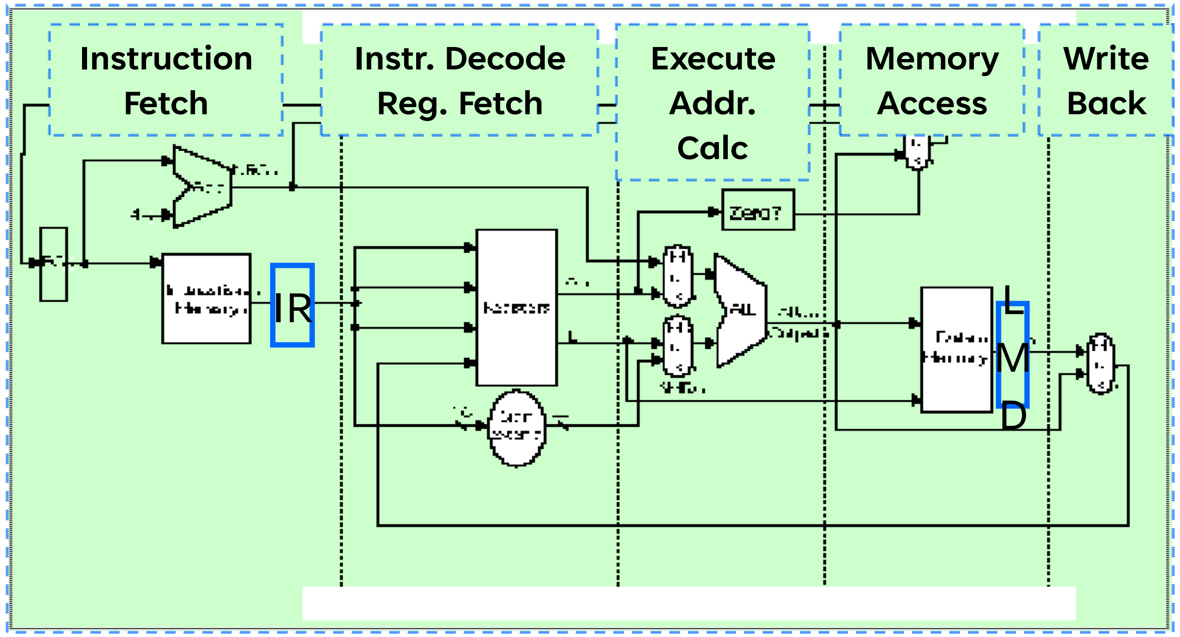
Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC by 4 to address the next sequential instruction.

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



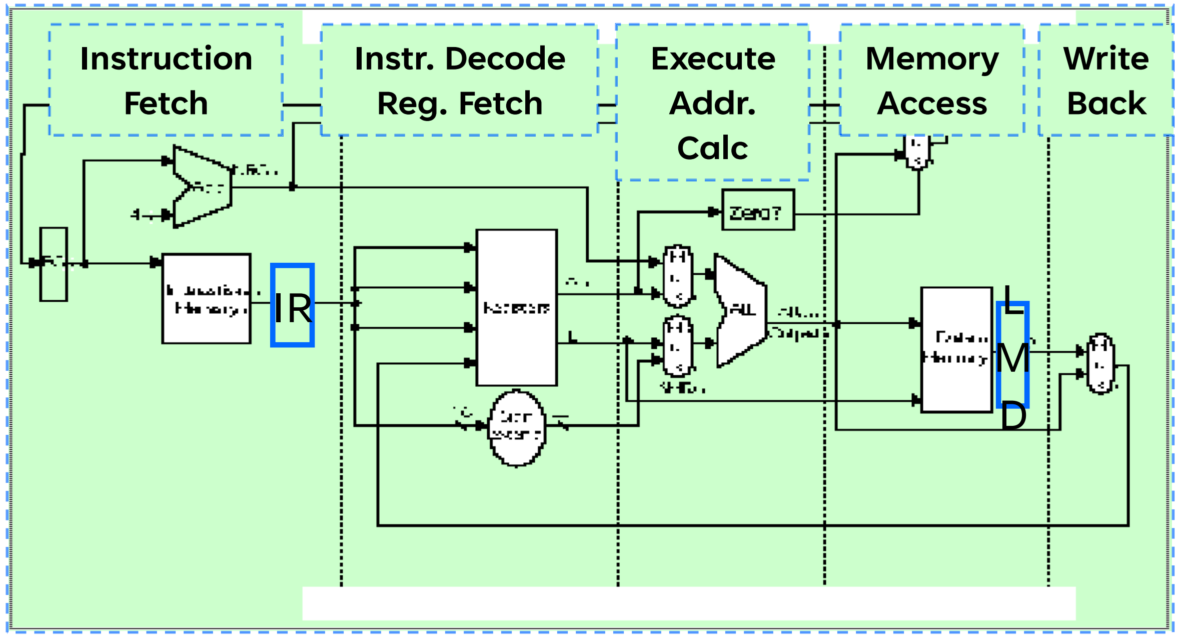
Passed To Next Stage

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;

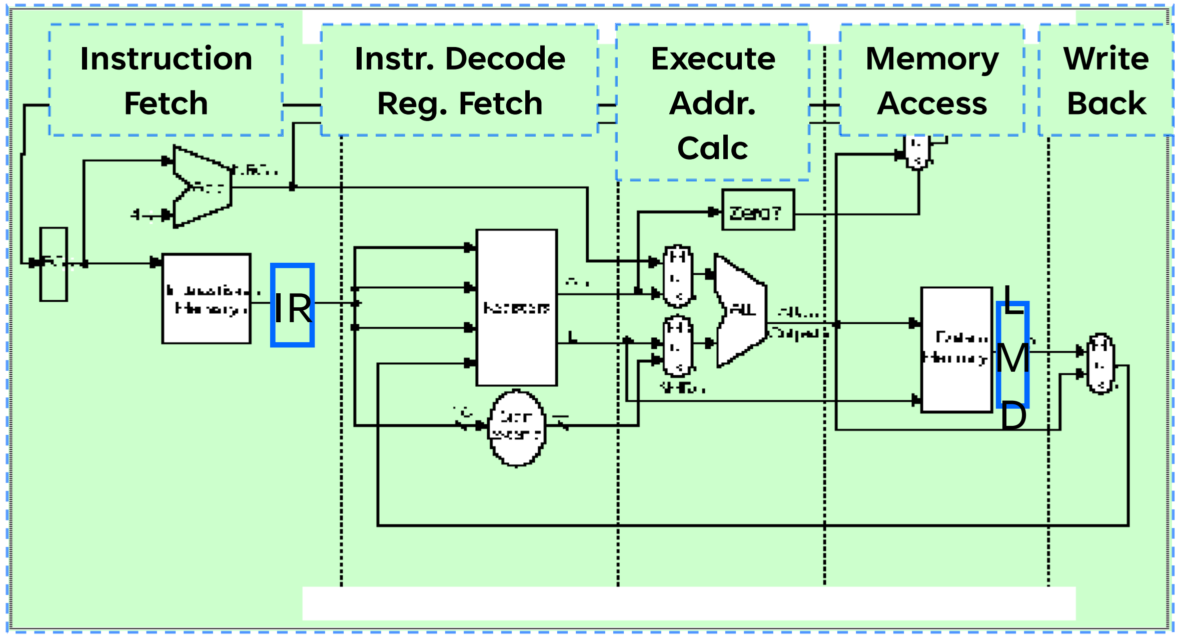
Execute Address Calculation (EX):

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]

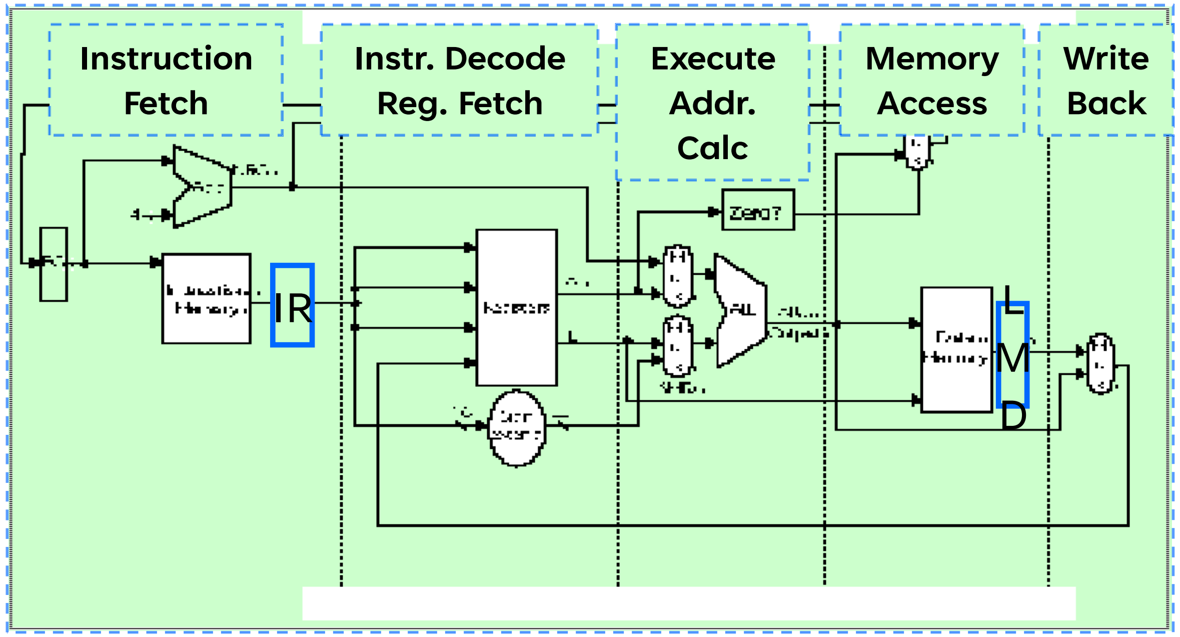
or

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

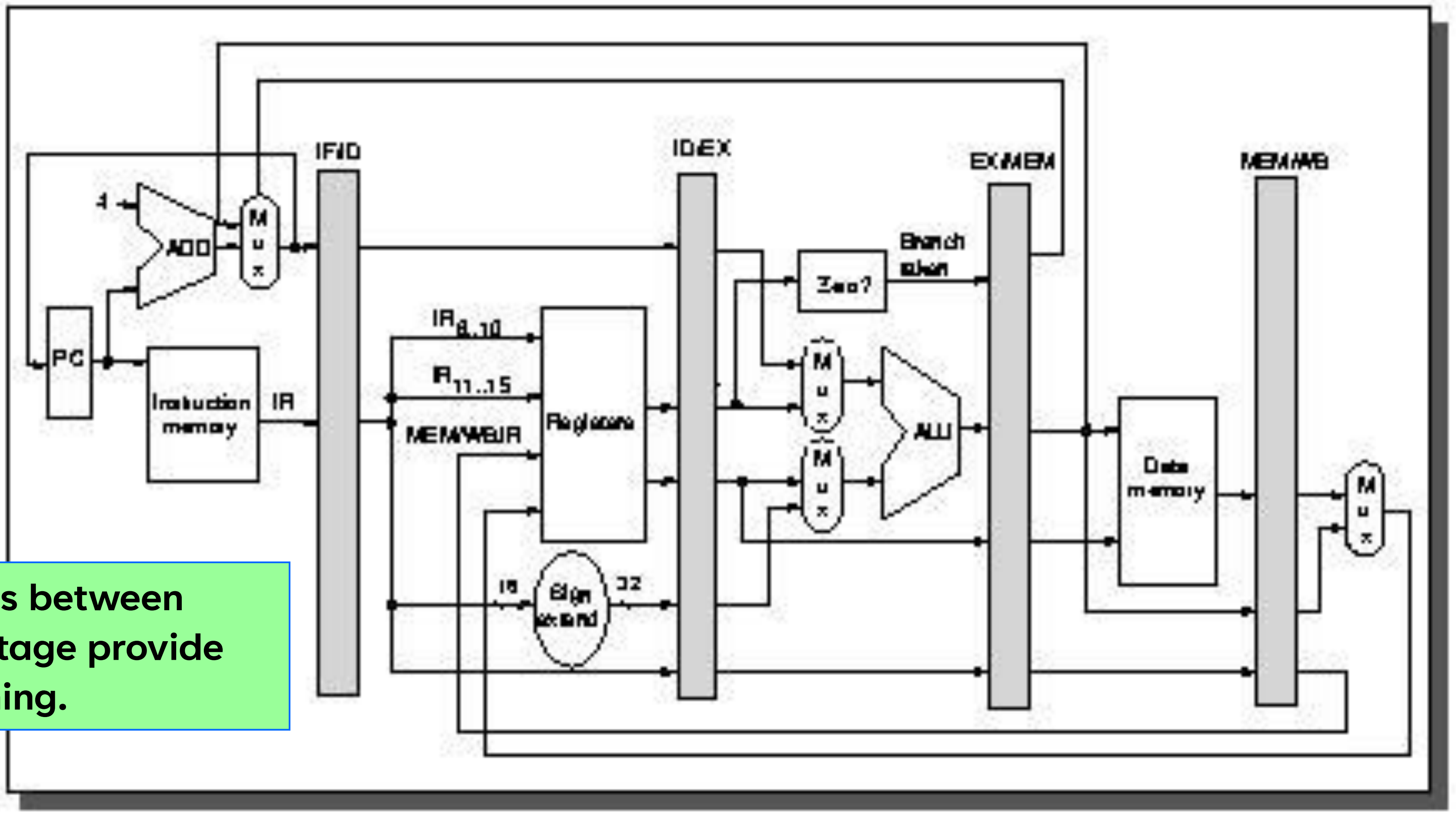


Passed To Next Stage

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

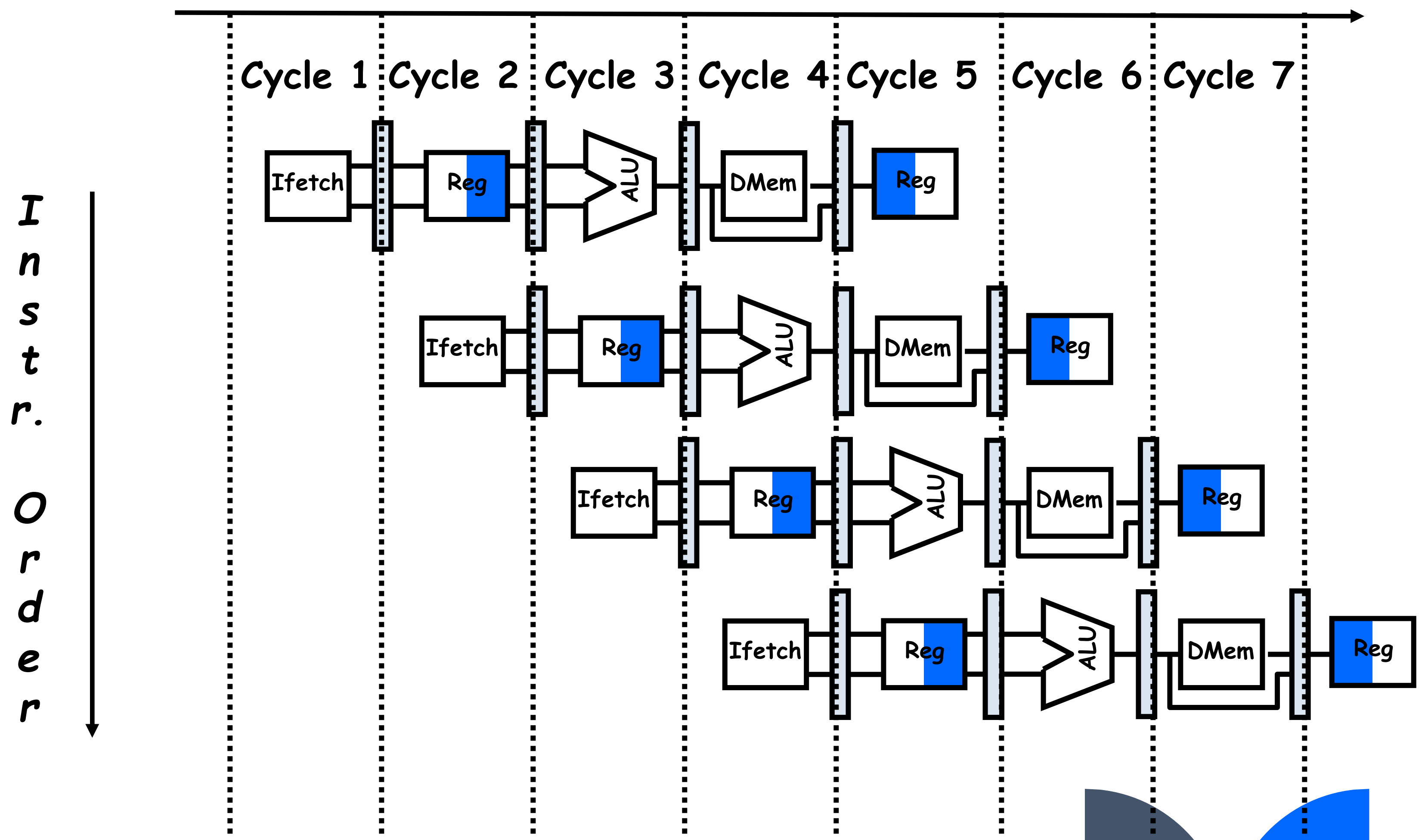


Figure 3.3

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

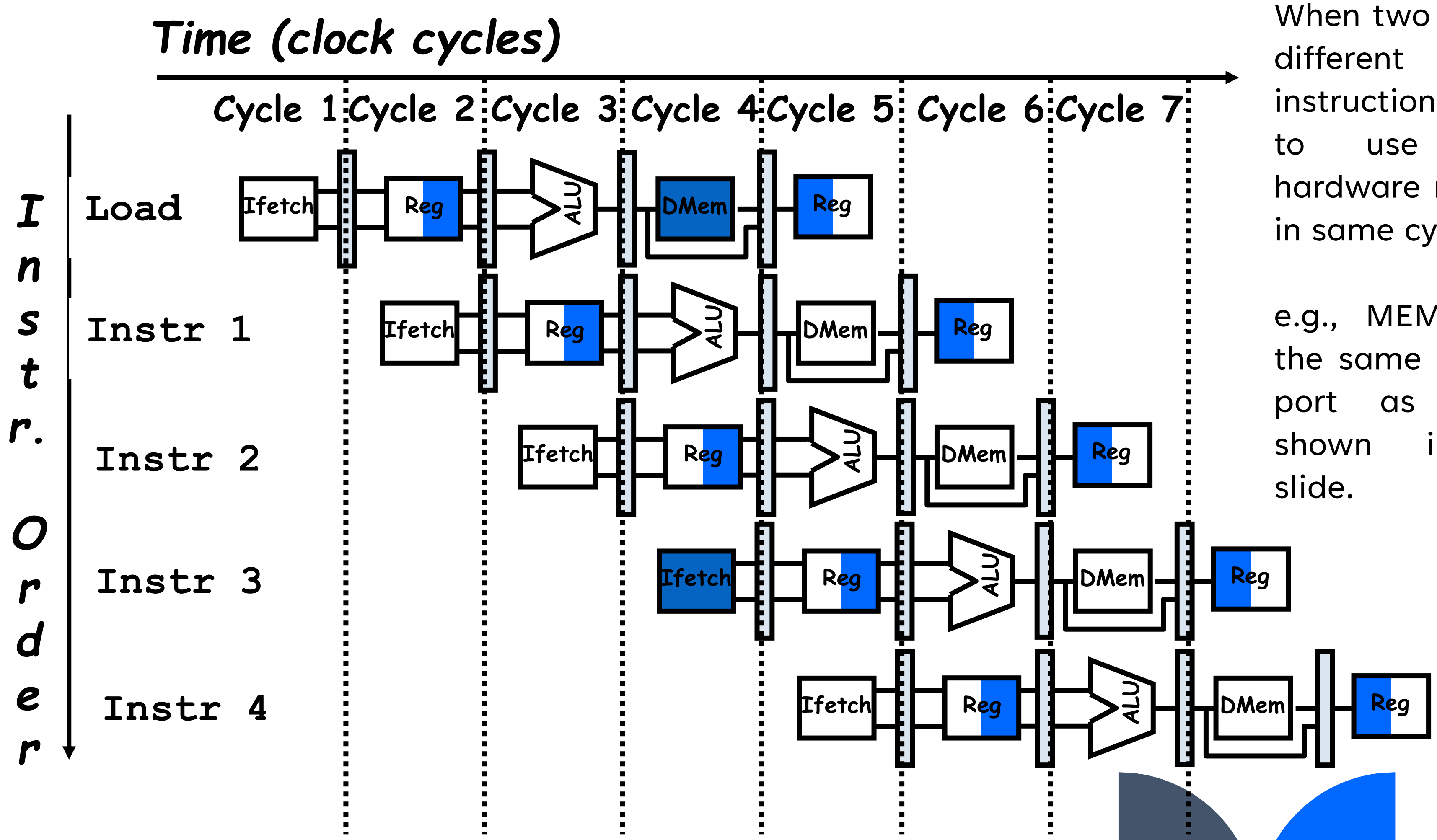
simple solution: stall

increases CPI, decreases performance

better solution: partial stall

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

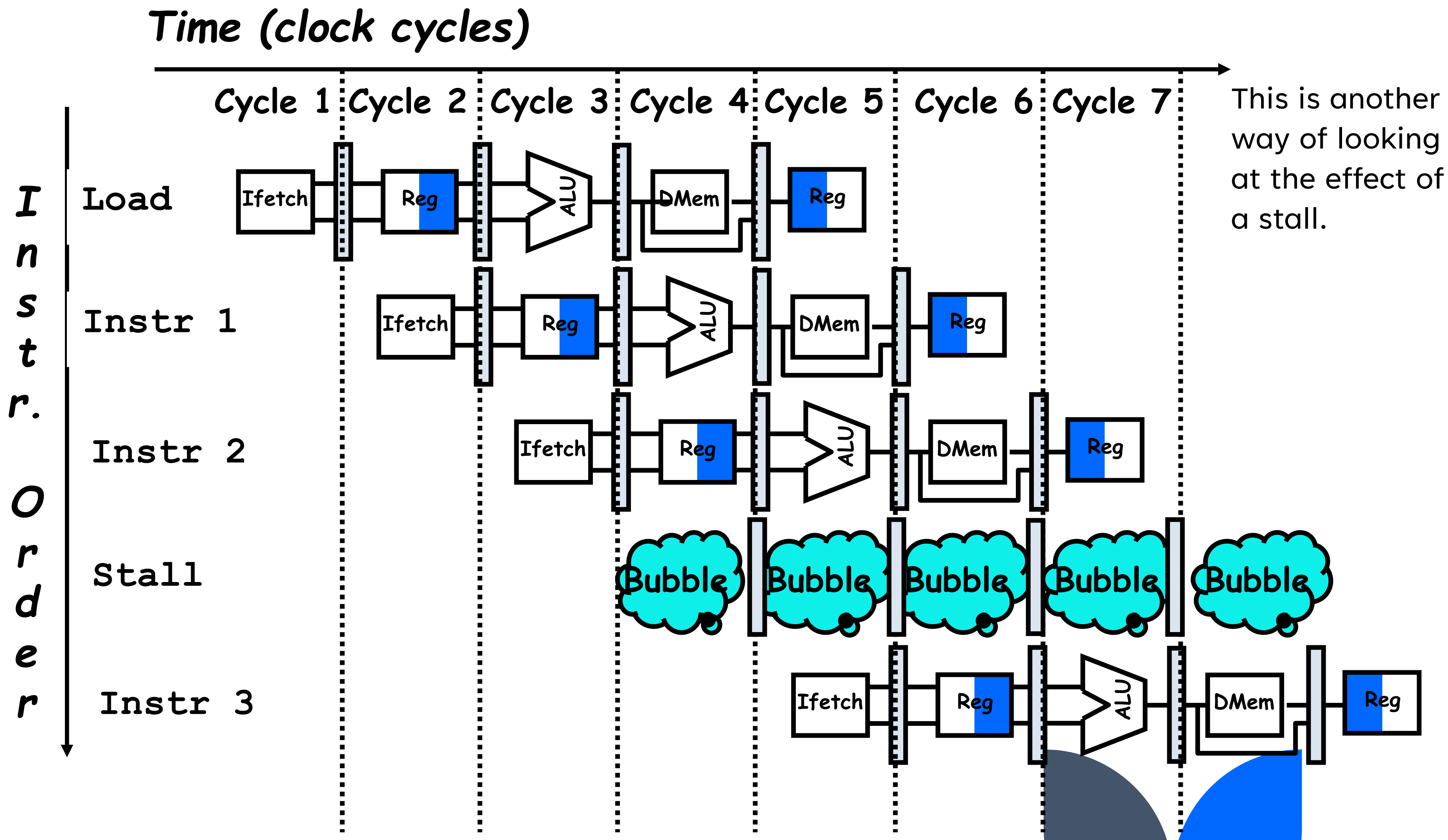
Structural Hazards



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

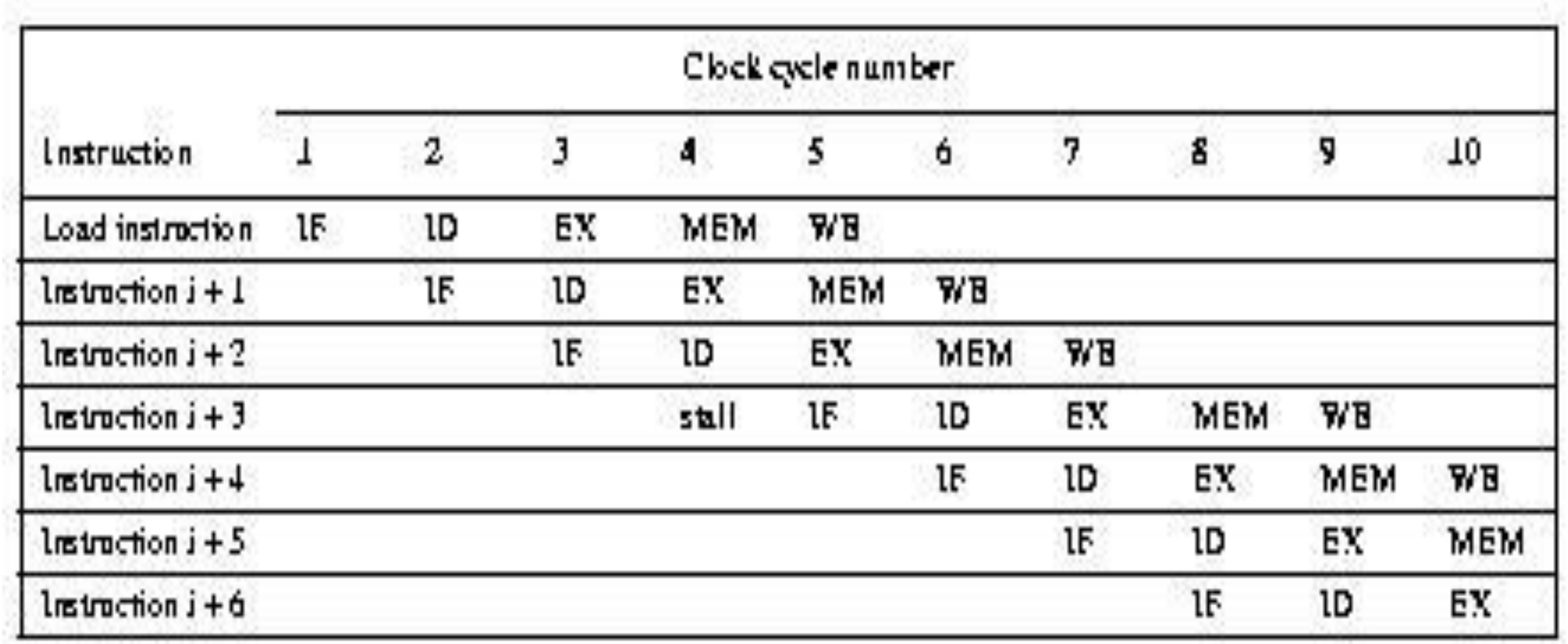
e.g., MEM uses the same memory port as IF as shown in this slide.

Structural Hazards



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

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.**

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

```
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.

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

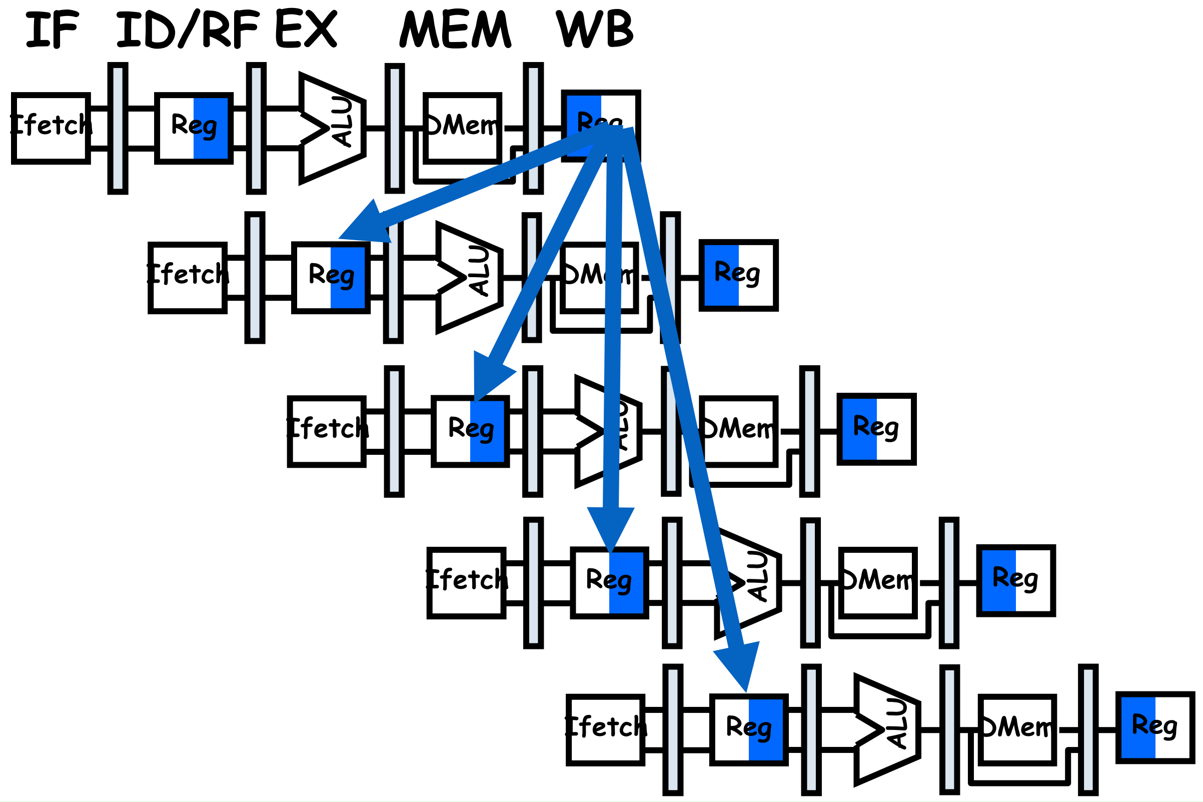
- Bypass/forward/shortcircuit (We will use the word “forward”)
- Use data before it is in the register
  + reduces/avoids stalls
  -- complex
- Crucial for common RAW hazards

Data Hazards

Time (clock cycles)

Instr. Order

```
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or   r8,r1,r9
xor r10,r1,r11
```



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

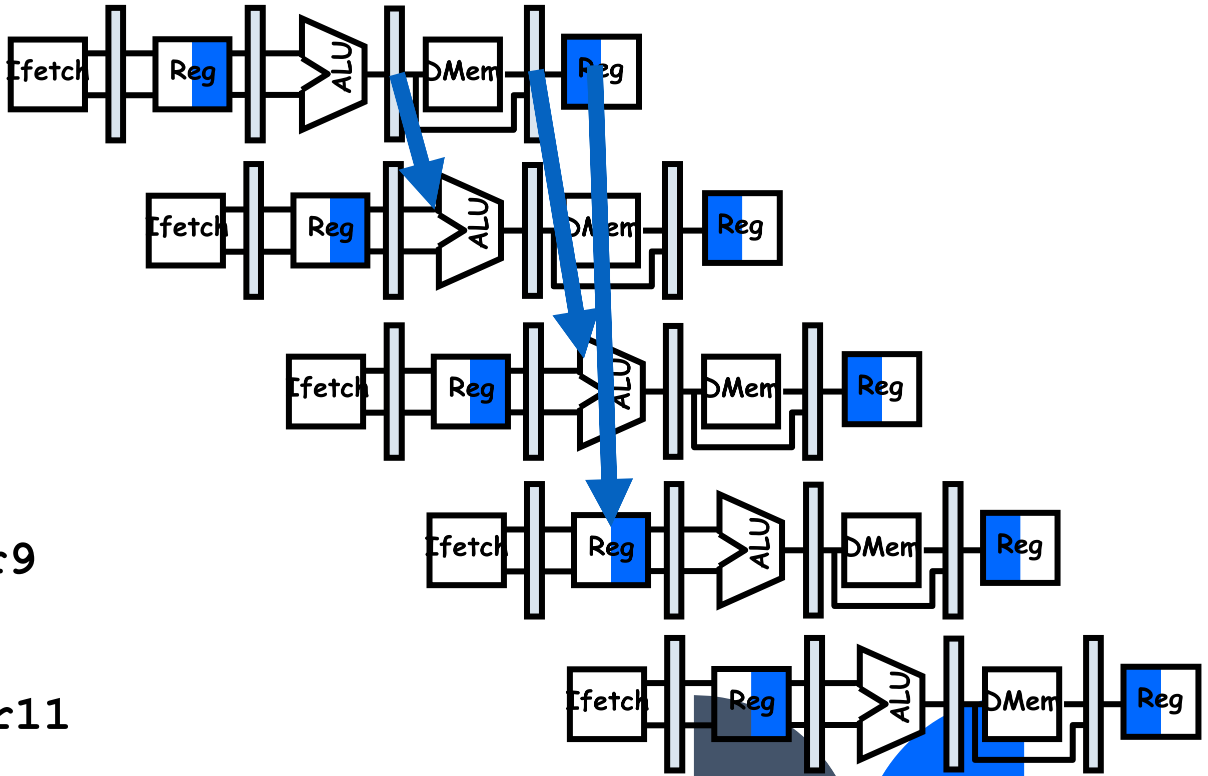
Forwarding To Avoid Data Hazard

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.

Time (clock cycles)

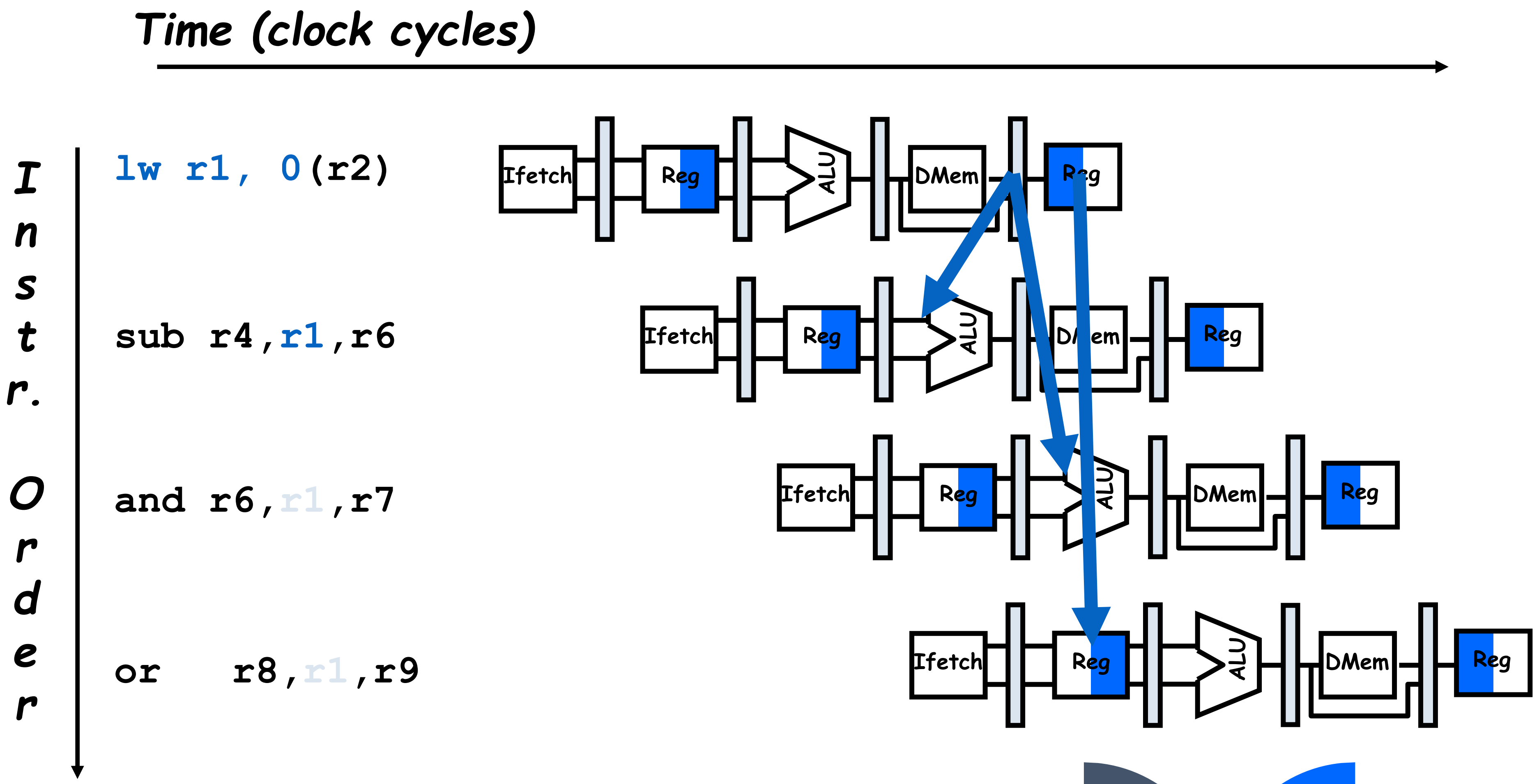
I n s t r. O r d e r

```
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or   r8,r1,r9
xor r10,r1,r11
```



Data Hazards

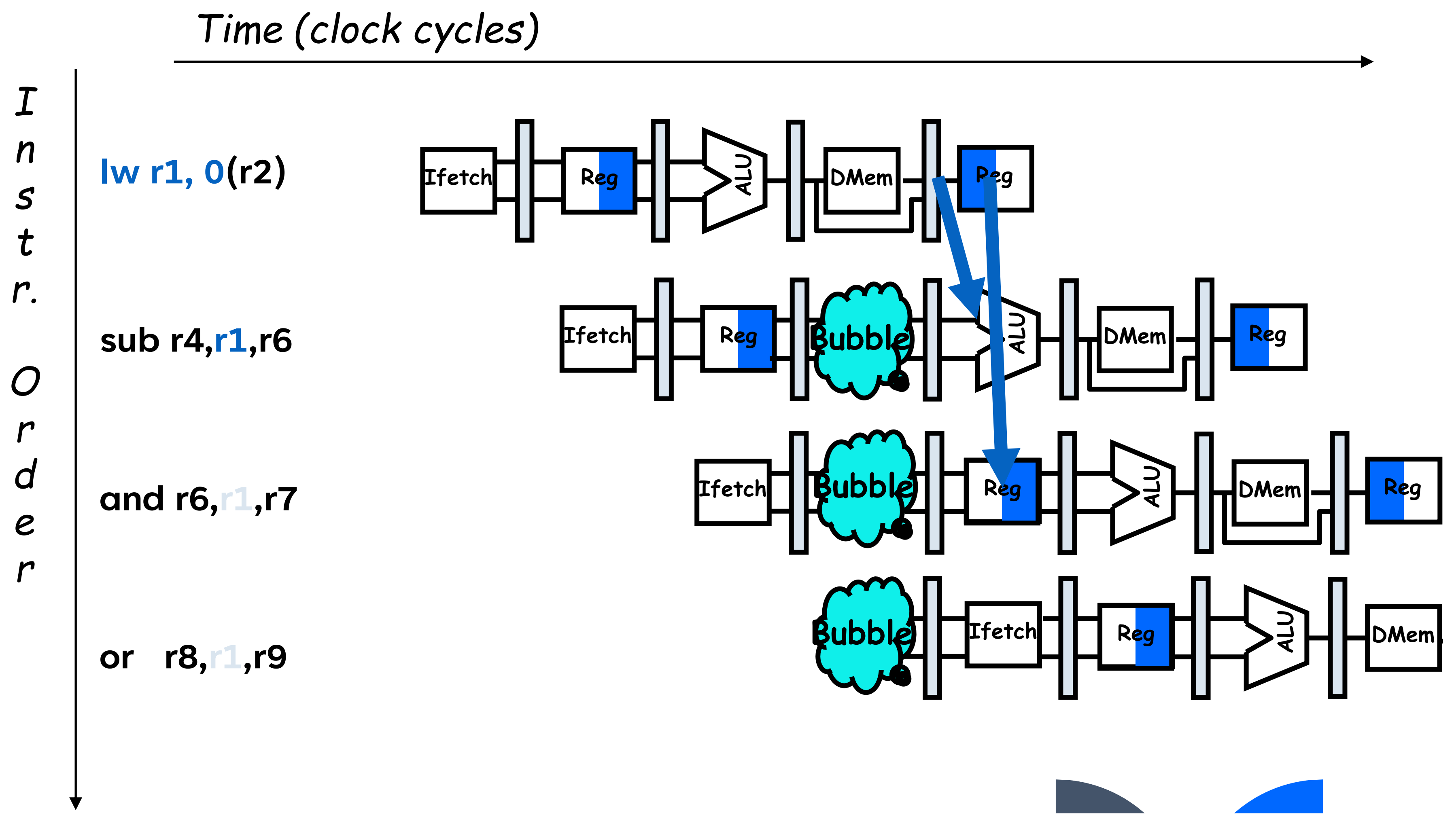
The data isn’t loaded until after the MEM stage.



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.

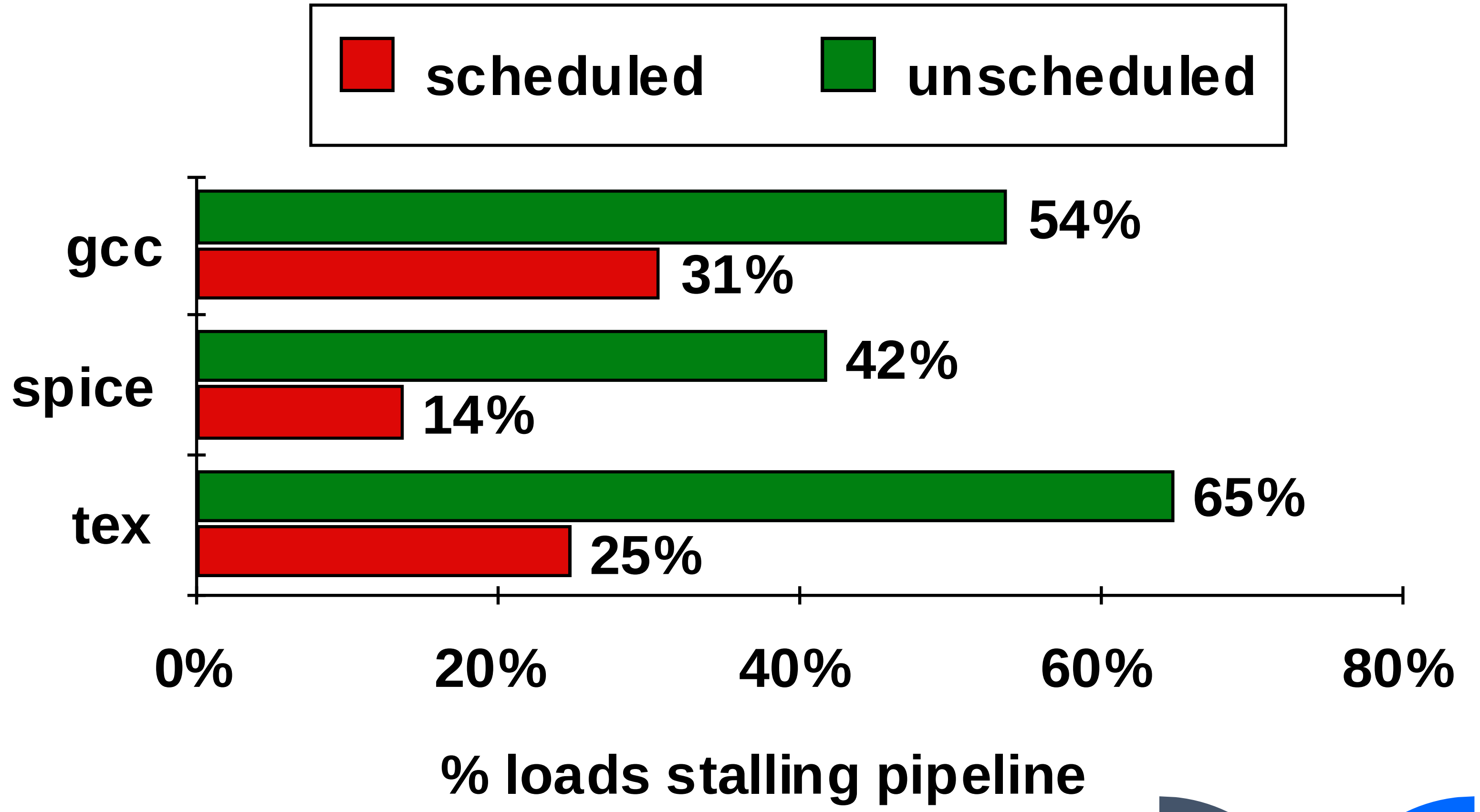
lw Rb, b	code sequence for **a = b+c** before scheduling
lw Rc, c	
Add Ra, Rb, Rc	stall
sw a, Ra	
lw Re, e	code sequence for **d = e+f** before scheduling
lw Rf, f	
sub Rd, Re, Rf	stall
sw d, Rd	

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

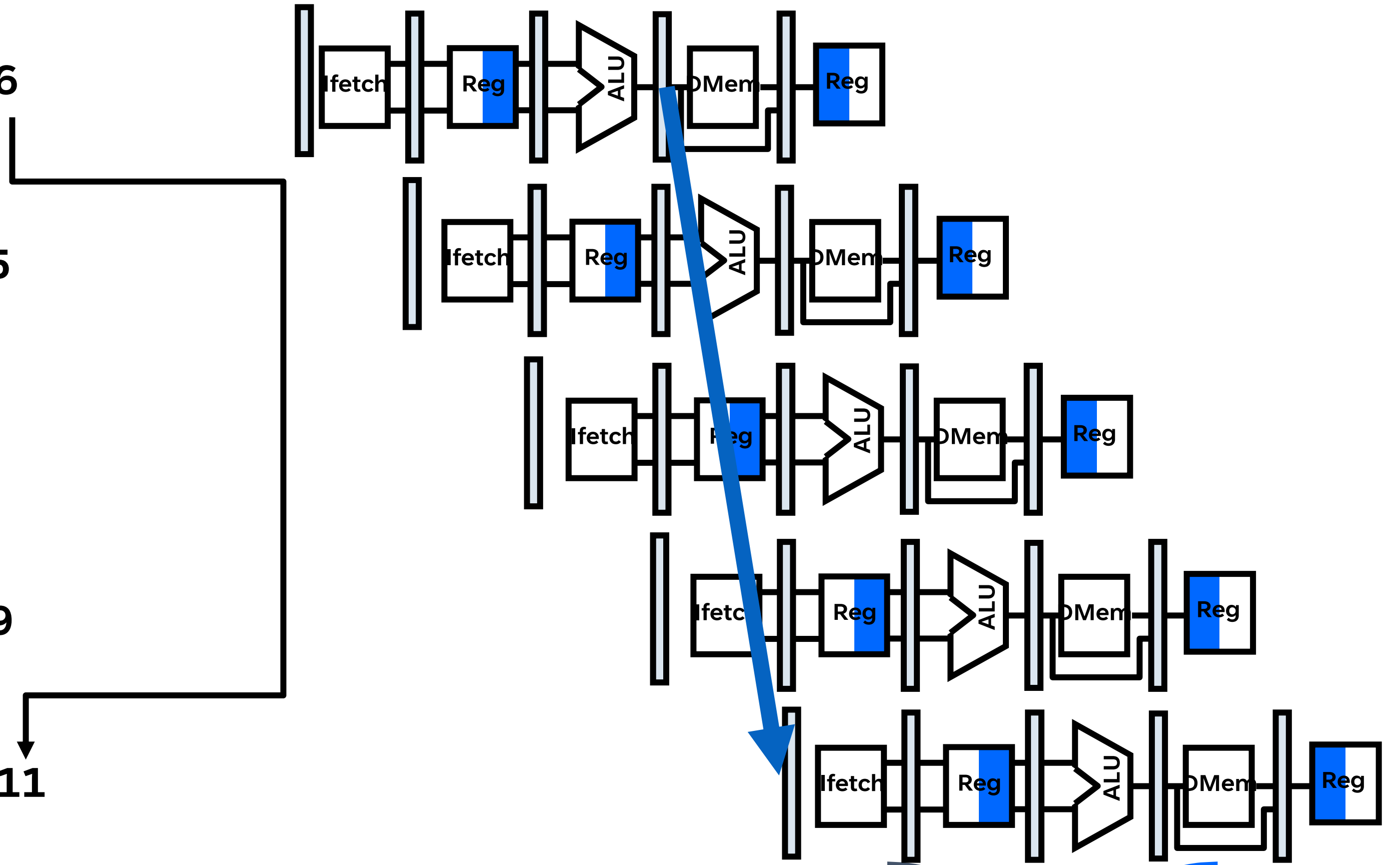
10: beq r1,r3,36

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!

(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

In the next chapter, we'll look at ways to avoid the branch all together.

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 successor_1
  sequential successor_2
  ....
  sequential successor_n
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

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

Pipelining Introduction Summary

Just overlap tasks, and easy if tasks are independent

Speed Up Š Pipeline Depth; if ideal CPI is 1, then:

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

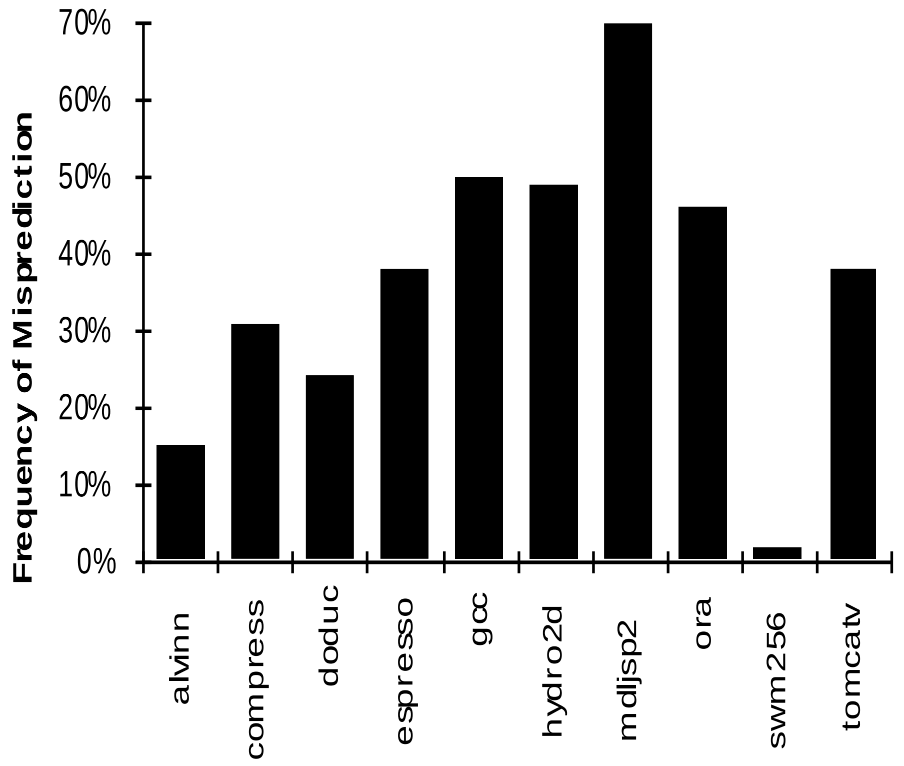
Hazards limit performance on computers:

- Structural: need more HW resources
- Data (RAW,WAR,WAW): need forwarding, compiler scheduling
- Control: delayed branch, prediction

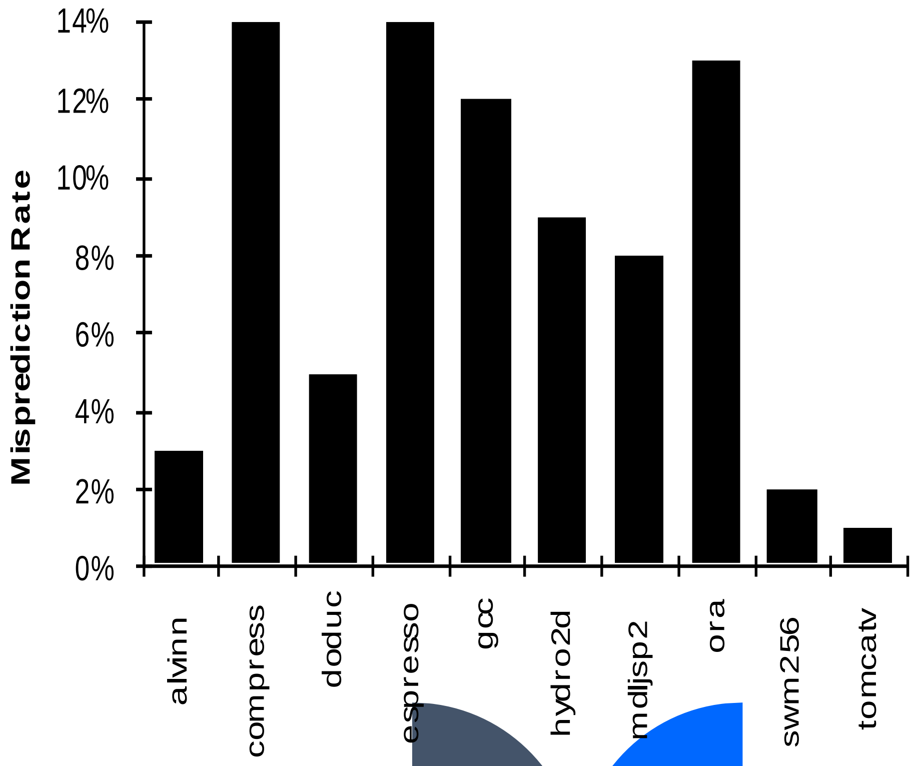
Control Hazards

Compiler “Static” Prediction of Taken/Untaken Branches

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



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

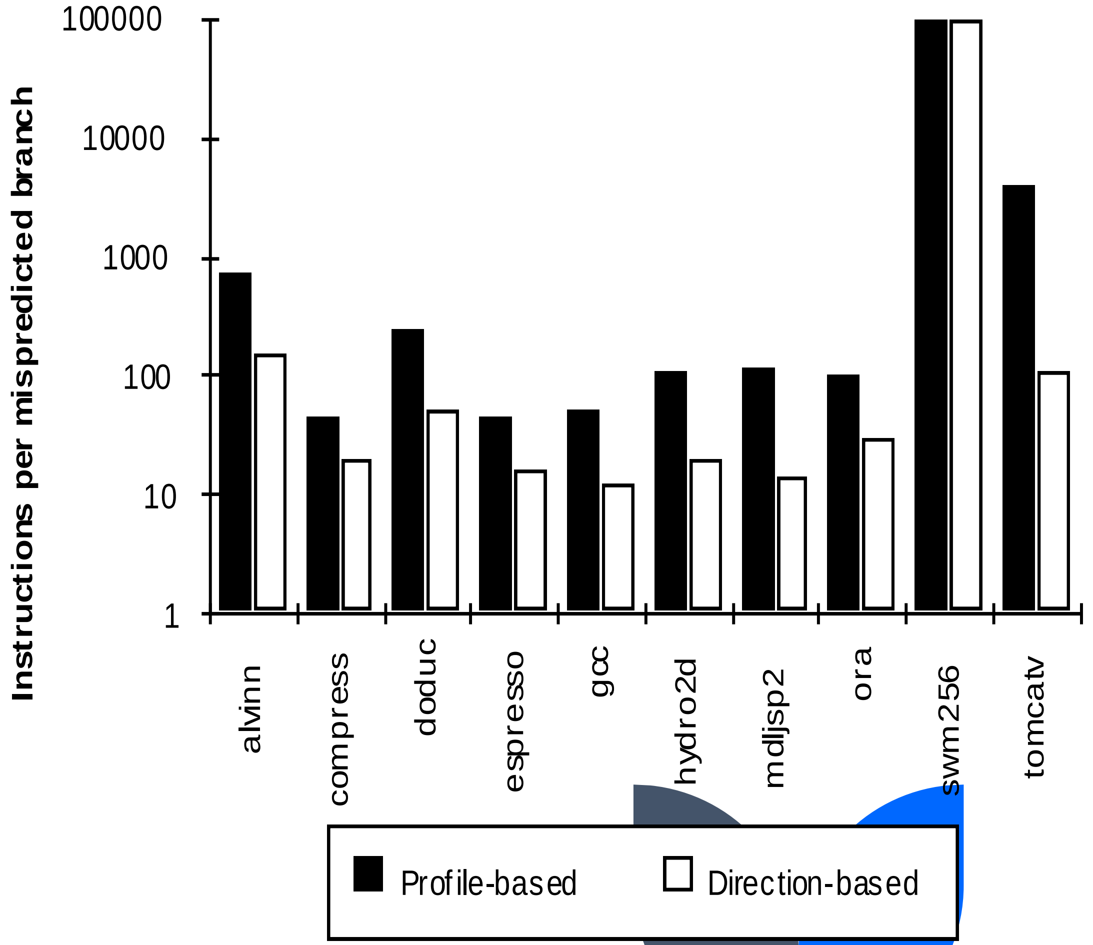
- Backward branch predict taken, forward branch not taken
- Profile-based prediction: record branch behavior, predict branch based on prior run

Control Hazards

Evaluating Static Branch Prediction Strategies

Misprediction ignores frequency of branch

"Instructions between mispredicted branches" is a better metric



What Makes Pipelining Hard?

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 Instructions

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

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

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