



Modern Computer Architecture

Lecture2 Pipelining: Basic and Intermediate Concepts

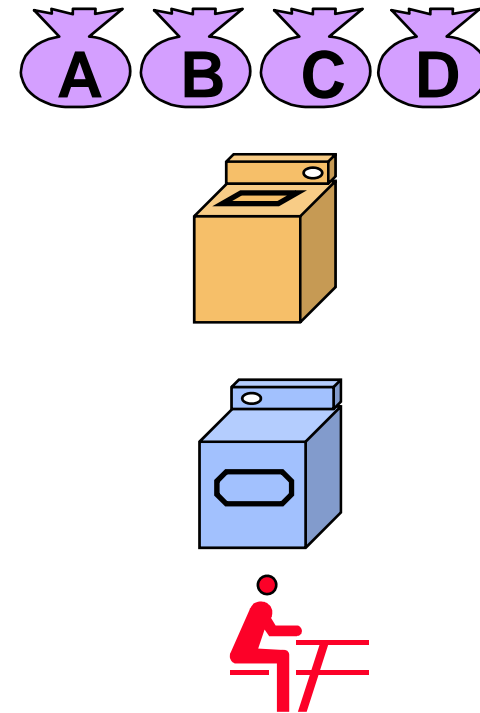
Hongbin Sun

国家集成电路人才培养基地

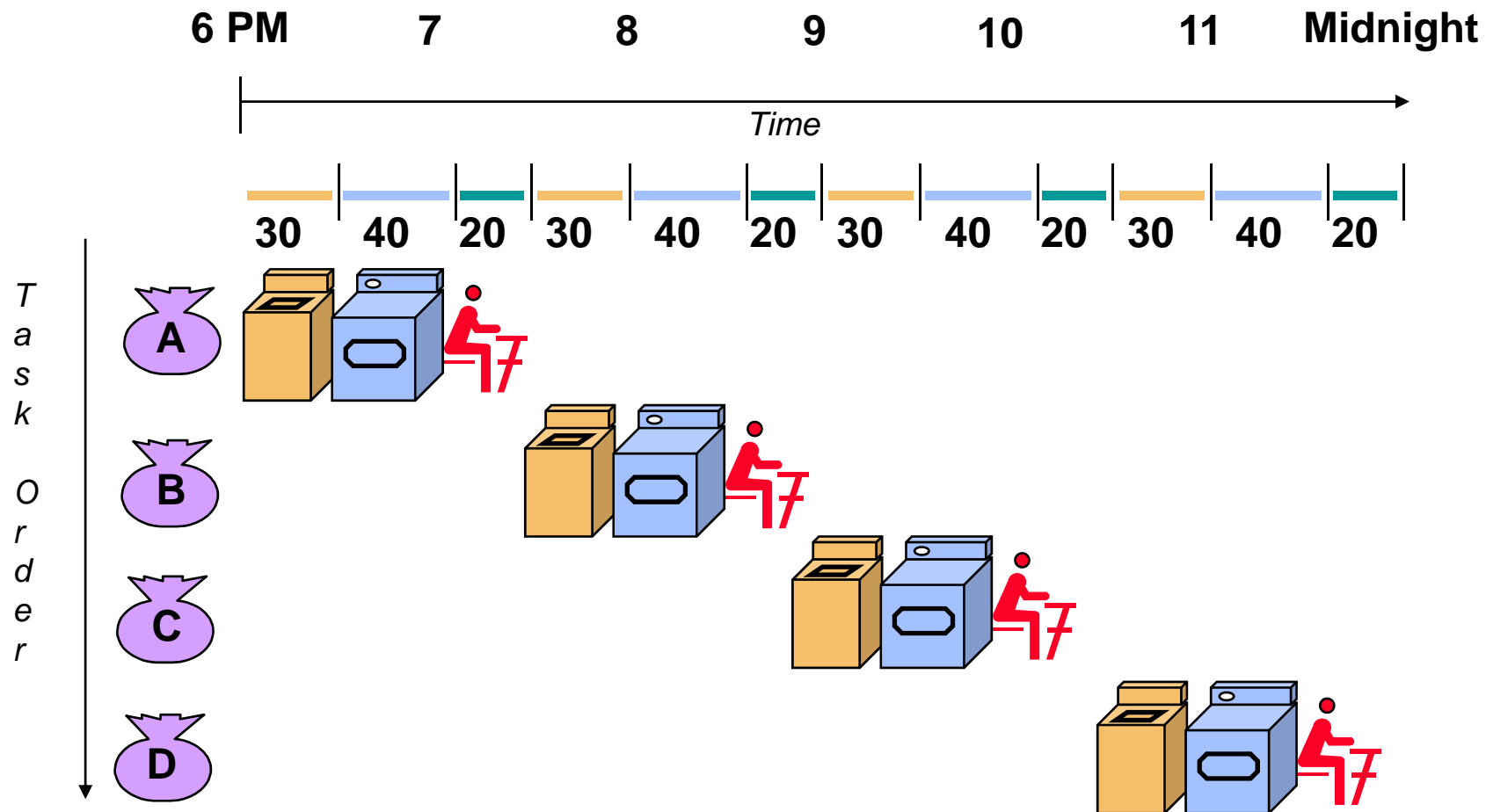
Xi'an Jiaotong University

Pipelining: Its Natural!

- 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

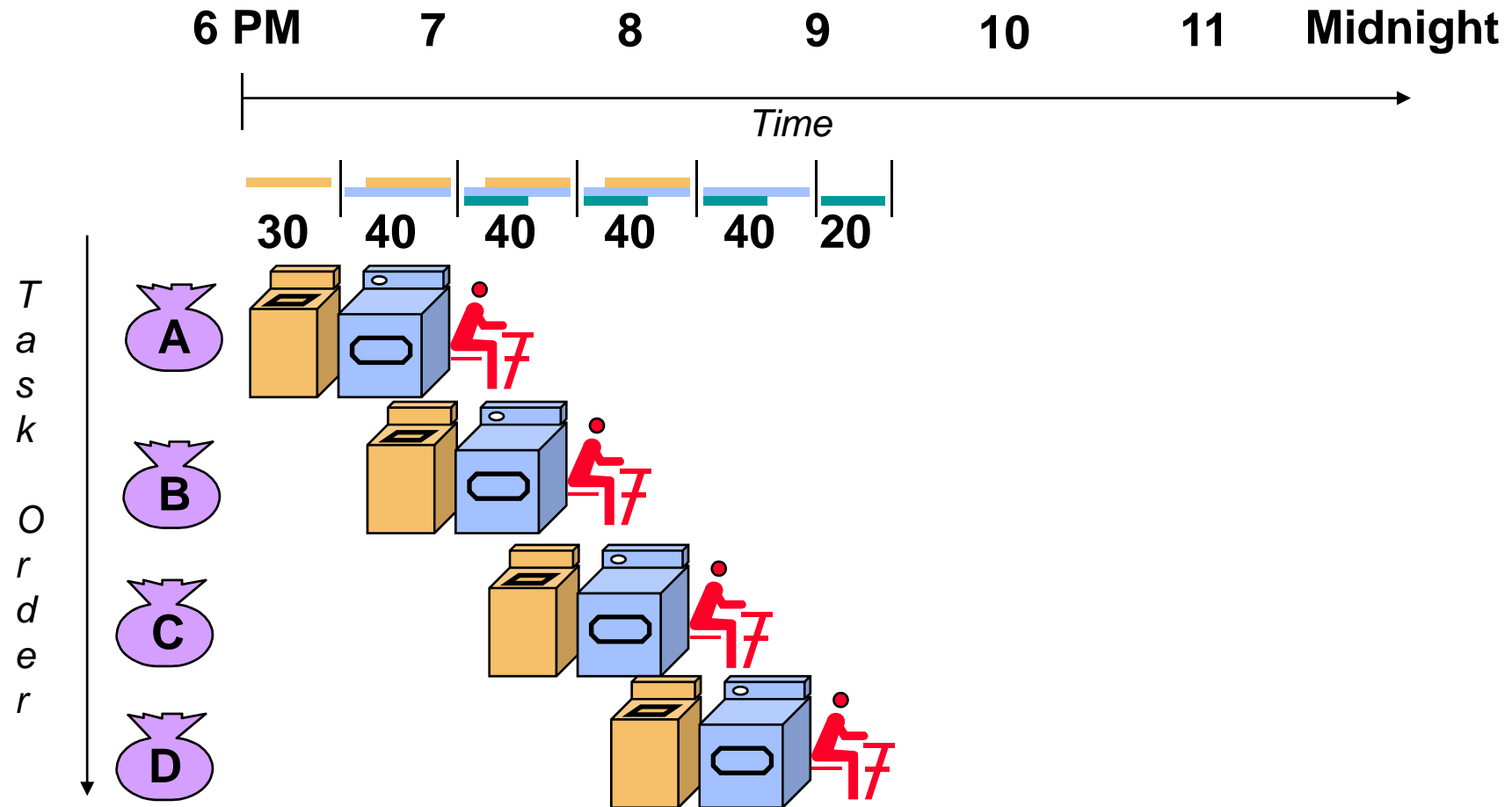


Sequential Laundry



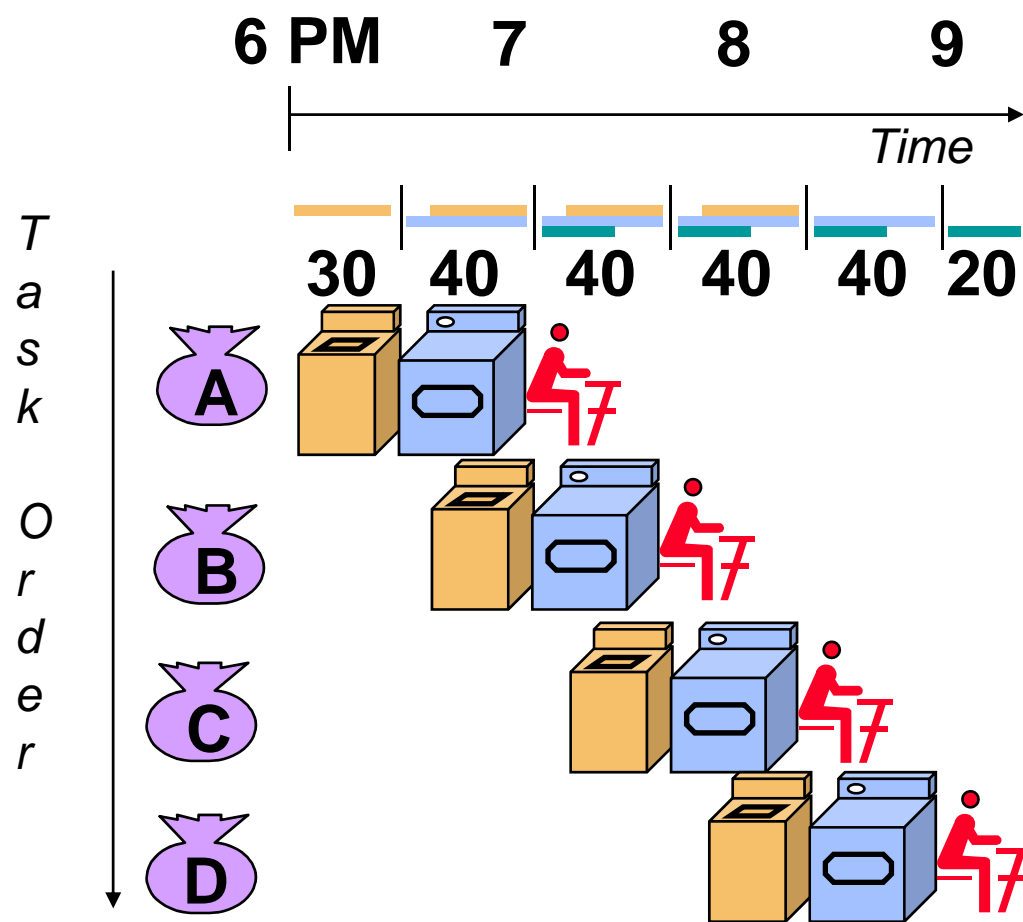
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry



- Pipelined laundry takes 3.5 hours for 4 loads

Pipeline 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

Computer Pipeline

- Execute billions of instructions, so **throughput** is what matters
- What is desirable in instruction sets for pipelining?
 - Variable length instructions vs. all instructions same length?
 - Memory operands part of any operation vs. memory operands only in loads or stores?
 - Register operand many places in instruction format vs. registers located in same place?

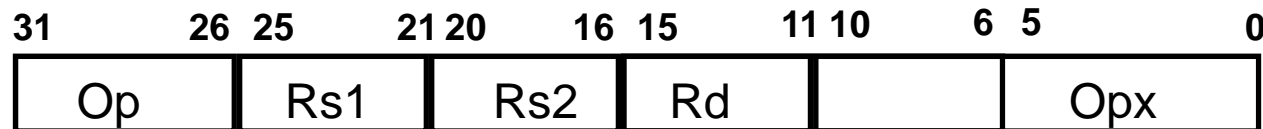
A "Typical" RISC

- **32-bit fixed format instruction (3 formats)**
- **Memory access only via load/store instructions**
- **32 32-bit GPR (R0 contains zero, DP take pair)**
- **3-address, reg-reg arithmetic instruction; registers in same place**
- **Single address mode for load/store:
base + displacement**
 - no indirection
- **Simple branch conditions**
- **Delayed branch**
 - see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

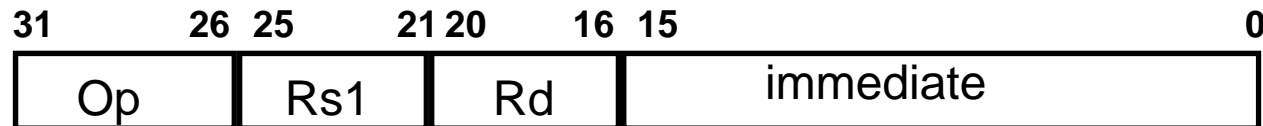
Example: MIPS

(Note register location)

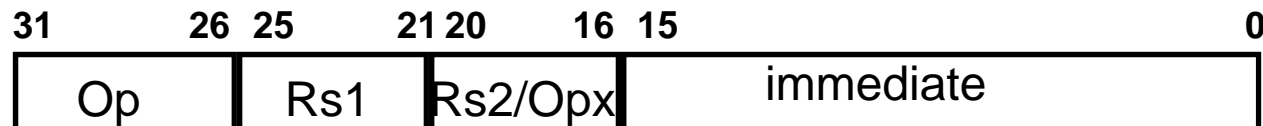
Register-Register



Register-Immediate



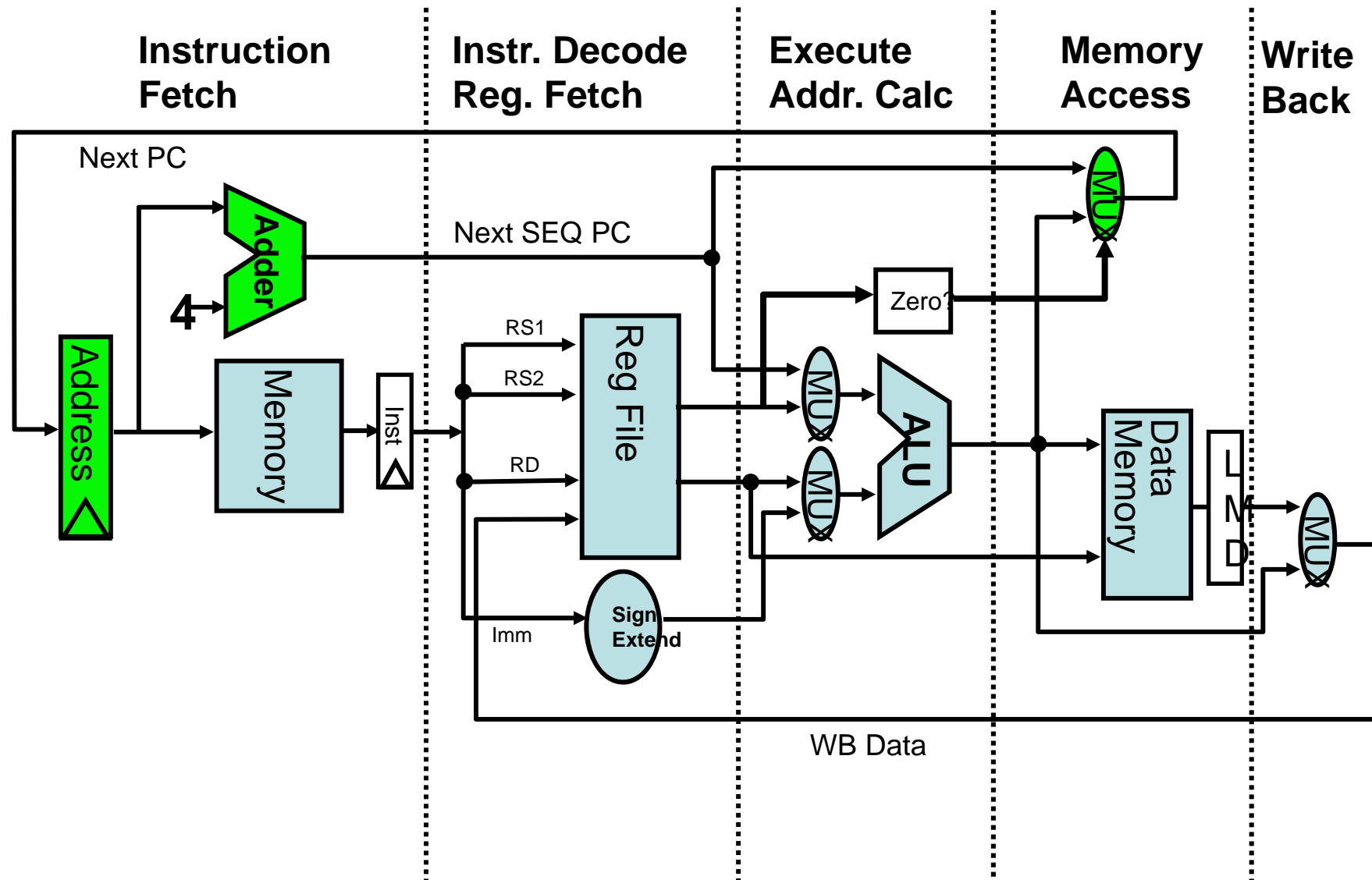
Branch



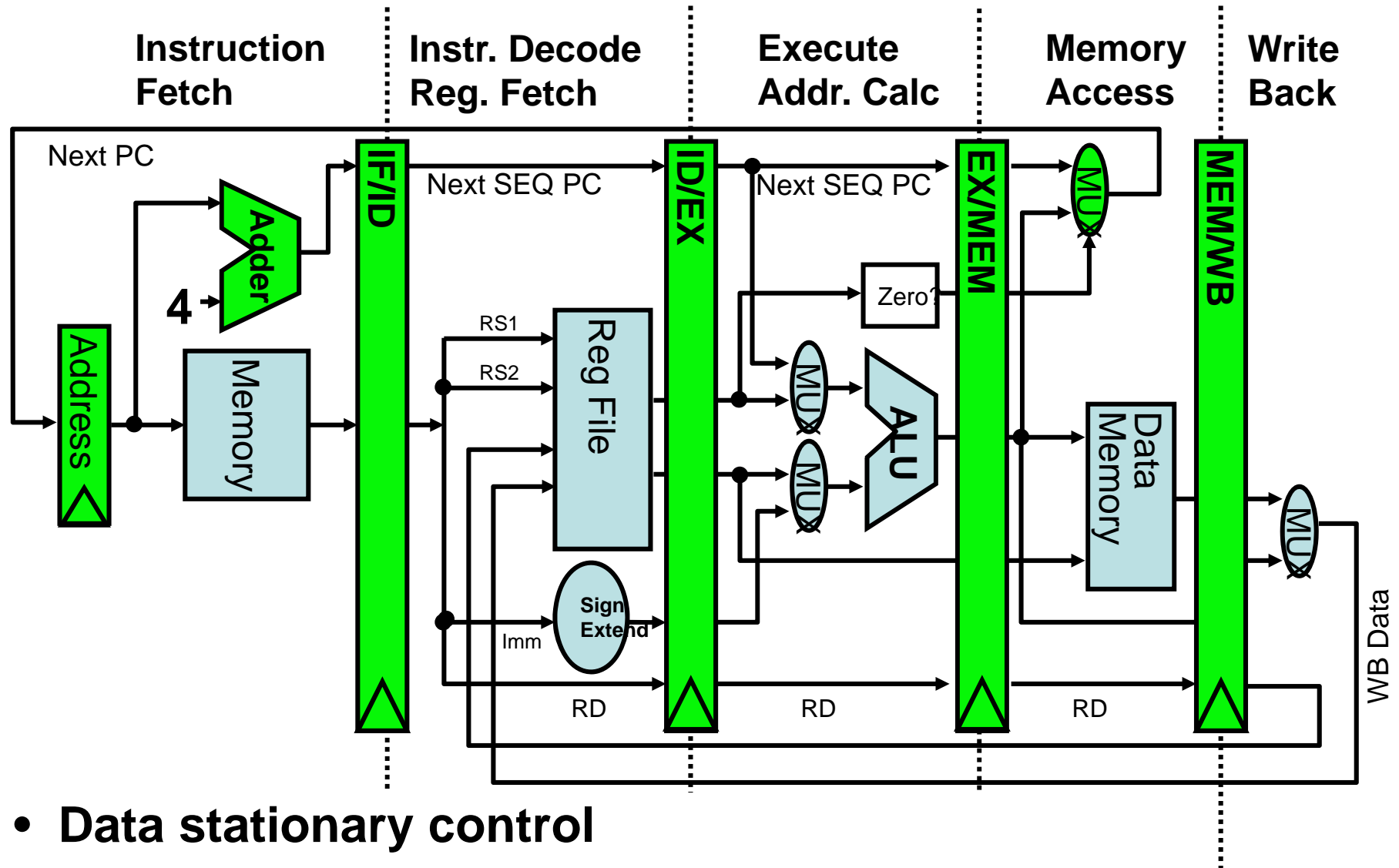
Jump / Call



5 Steps of MIPS Datapath

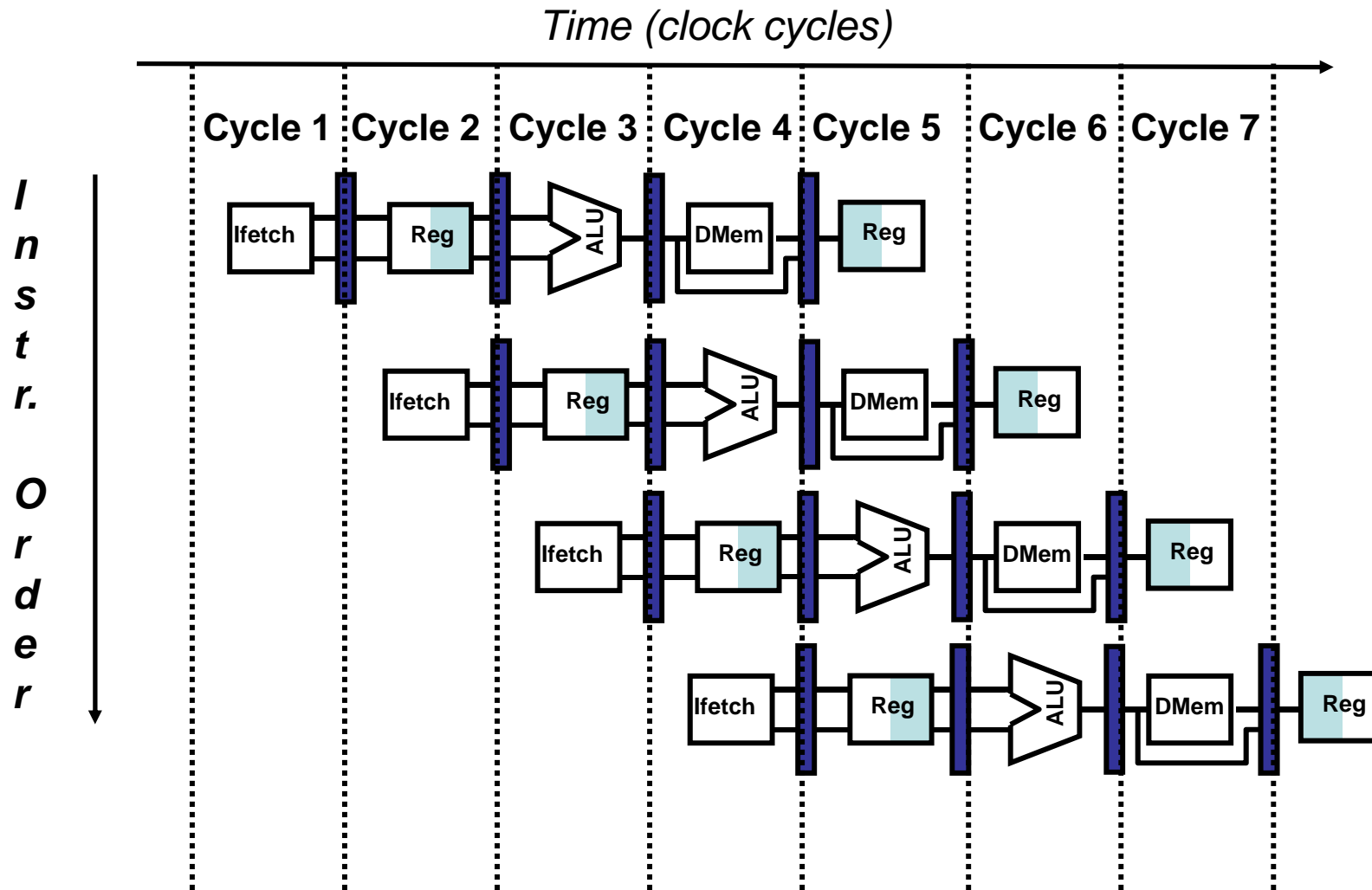


5 Steps of MIPS Datapath



- **Data stationary control**
 - local decode for each instruction phase / pipeline stage

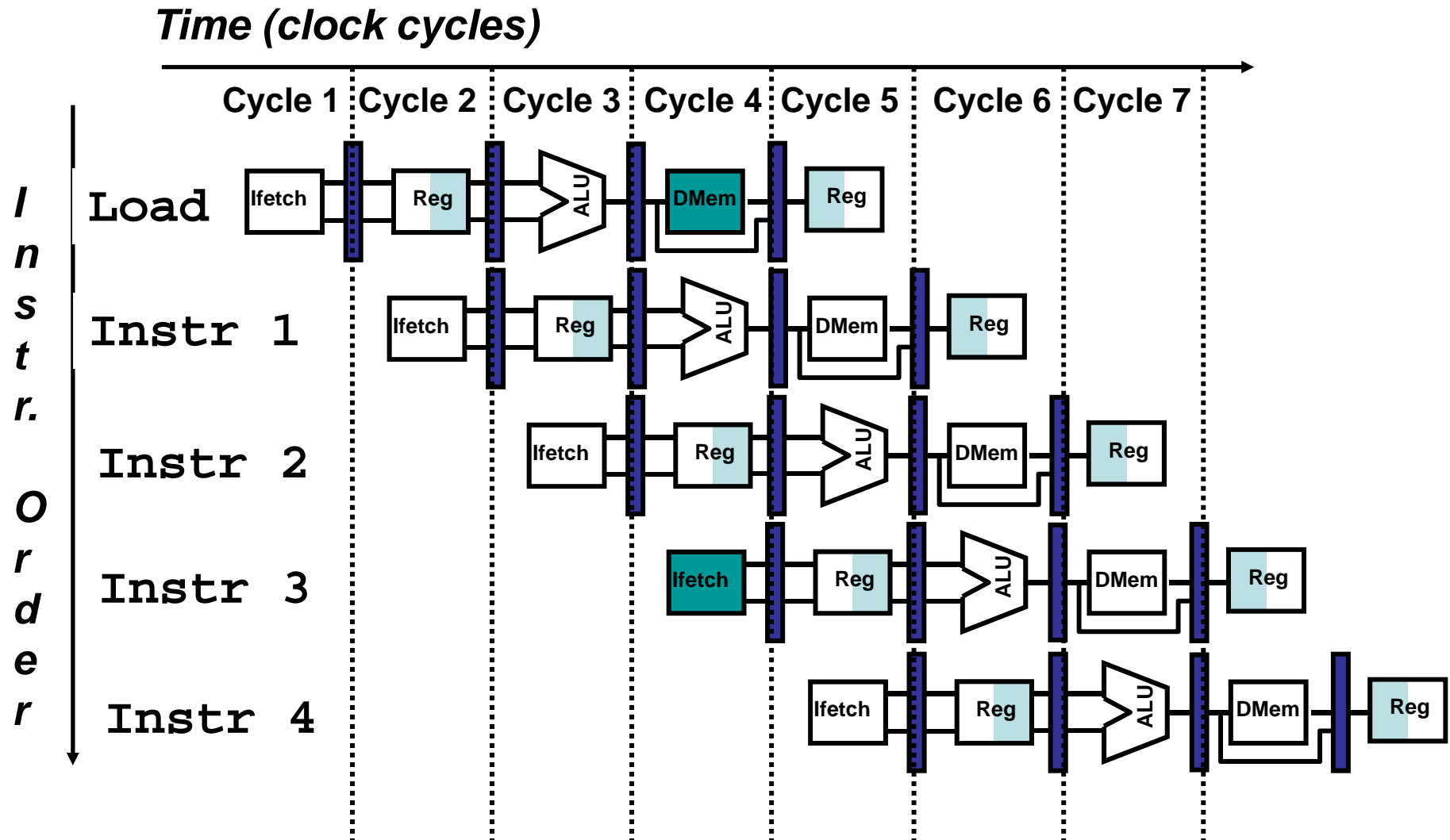
Visualizing Pipelining



Its Not That Easy for Computers

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

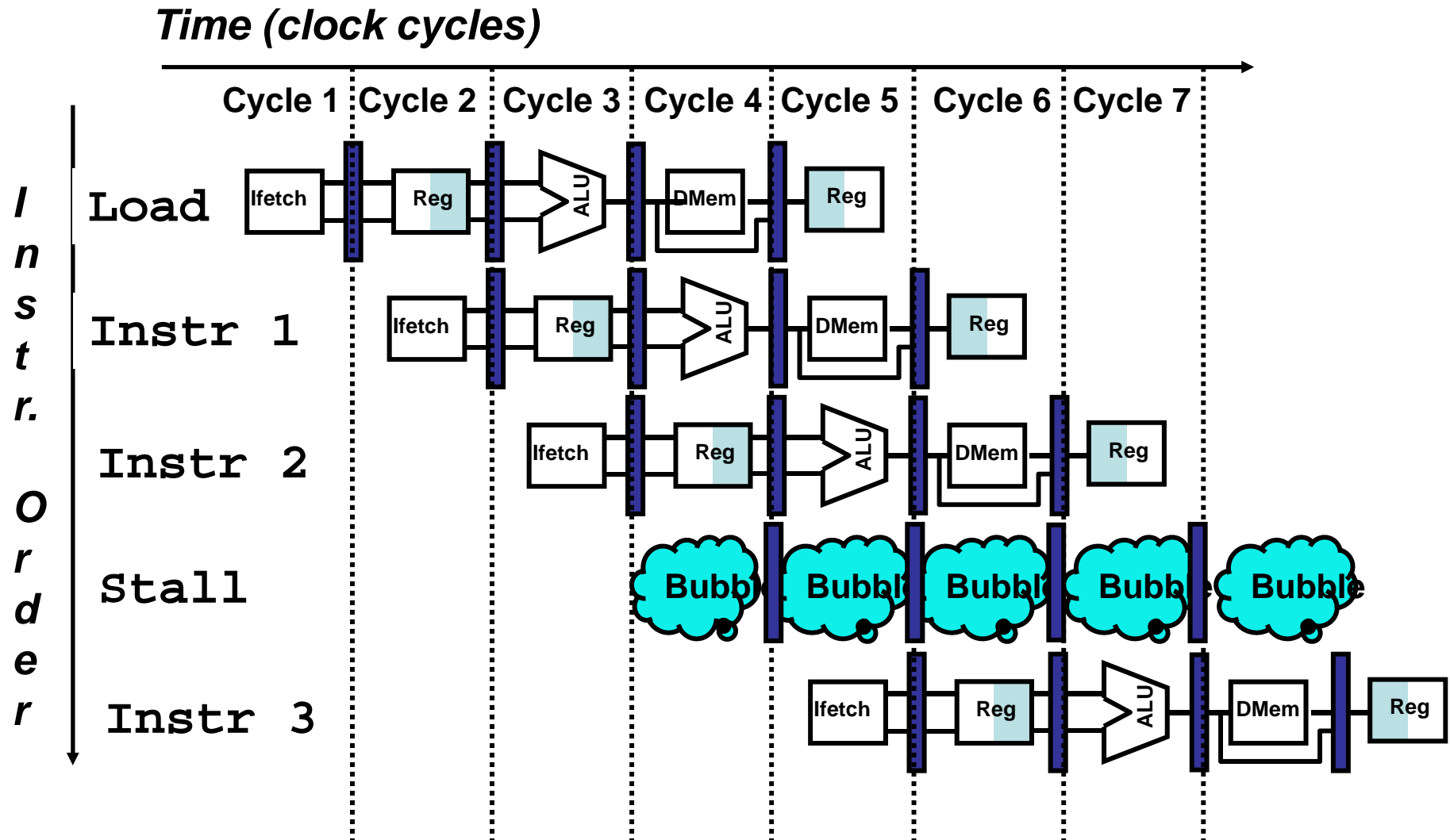
One Memory Port/Structural Hazards



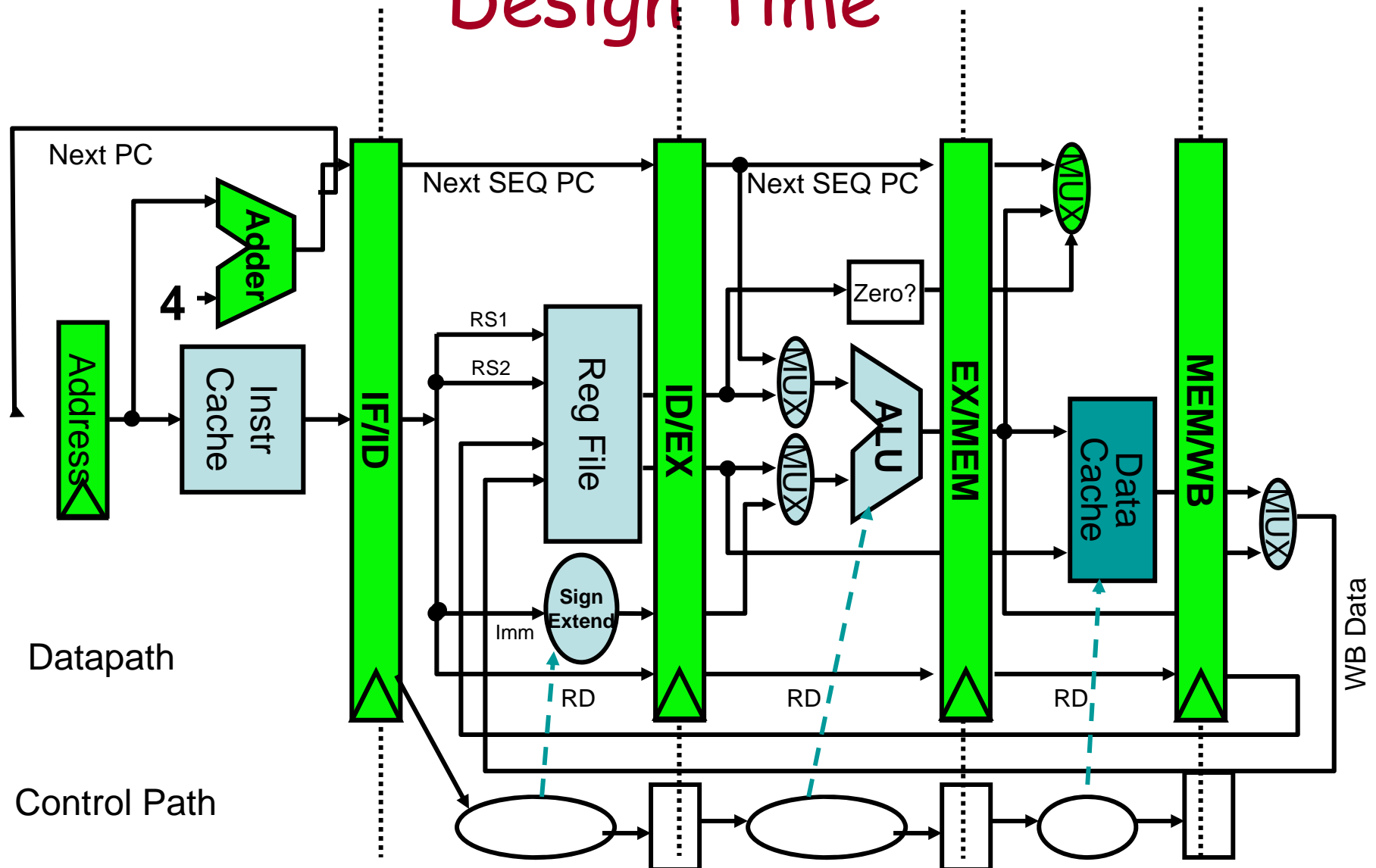
Resolving structural hazards

- **Define: attempt to use same hardware for two different things at the same time**
- **Solution 1: Wait**
 - ⇒ must detect the hazard
 - ⇒ must have mechanism to stall
- **Solution 2: Throw more hardware at the problem**

One Memory Port/Structural Hazards



Eliminating Structural Hazards at Design Time

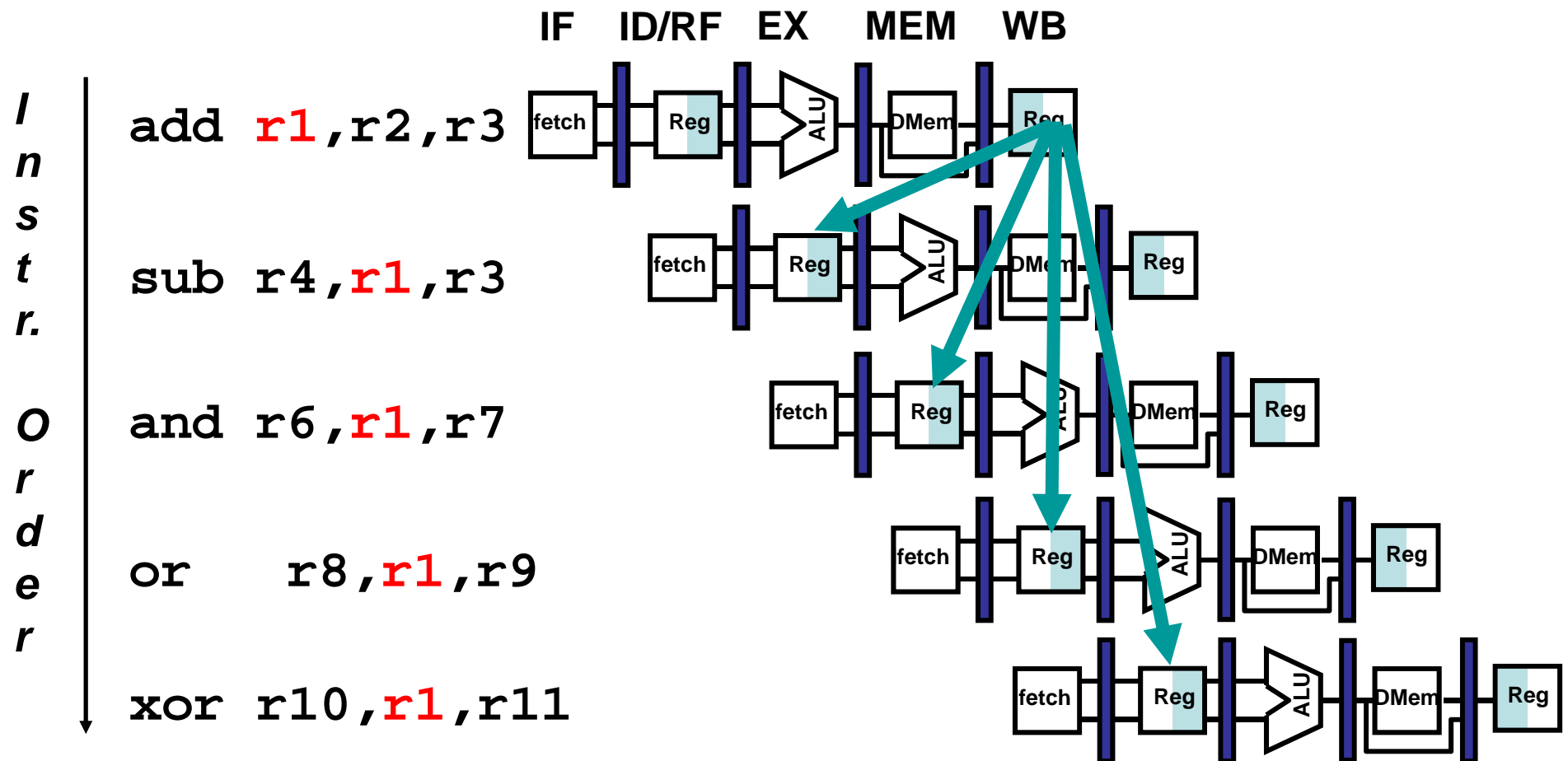


Role of Instruction Set Design in Structural Hazard Resolution

- **Simple to determine the sequence of resources used by an instruction**
 - opcode tells it all
- **Uniformity in the resource usage**
- **Compare MIPS to IA32?**
- **MIPS approach => all instructions flow through same 5-stage pipelining**

Data Hazard on R1

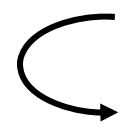
Time (clock cycles)



Three Generic Data Hazards

- **Read After Write (RAW)**

Instr_j tries to read operand before Instr_i writes it

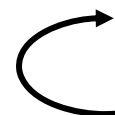
 I: add **r1**, r2, r3
J: sub r4, **r1**, r3

- 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



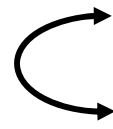
```
I: sub r4, r1, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

- 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)**

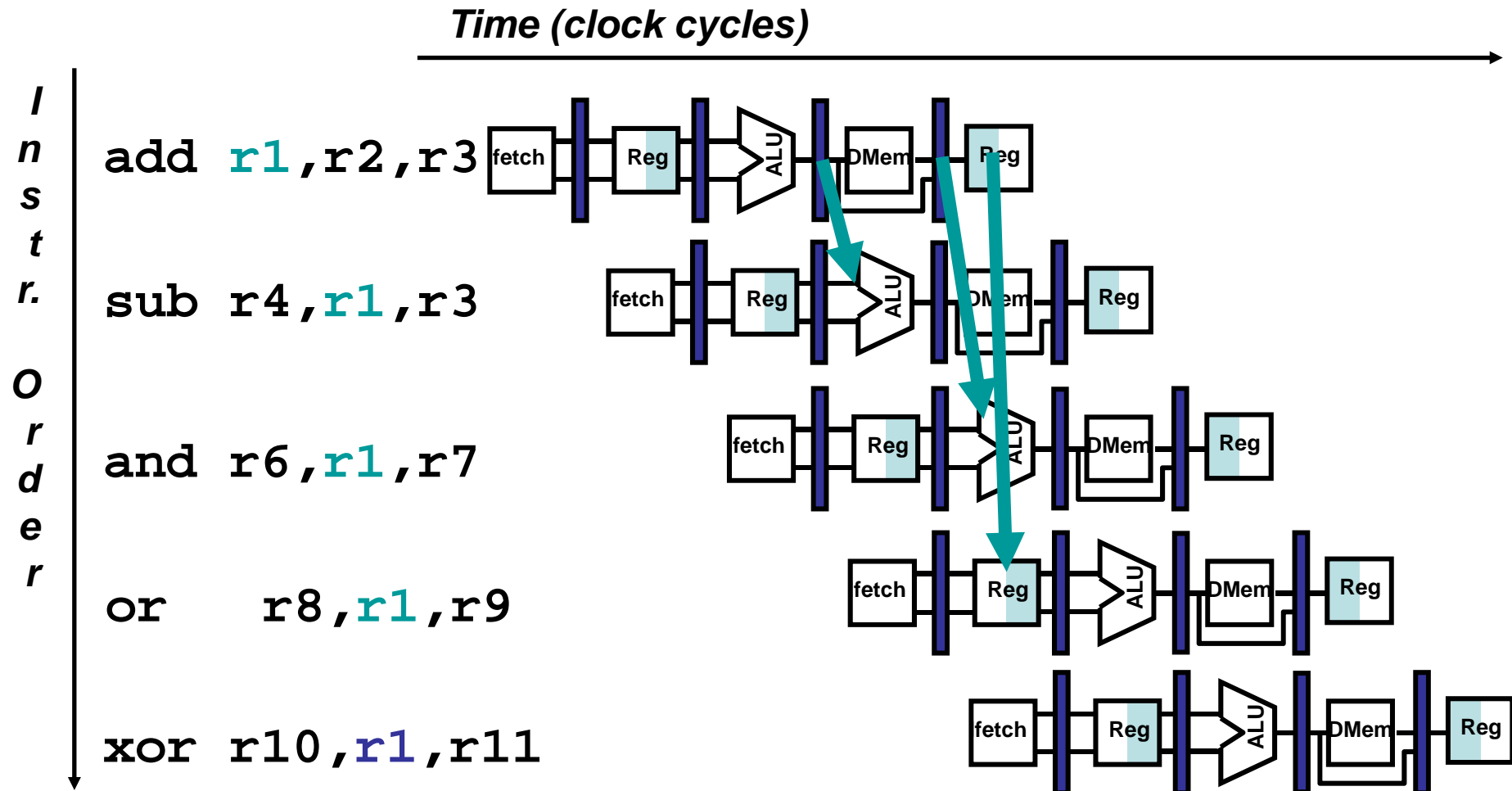
Instr_j writes operand *before* Instr_i writes it.



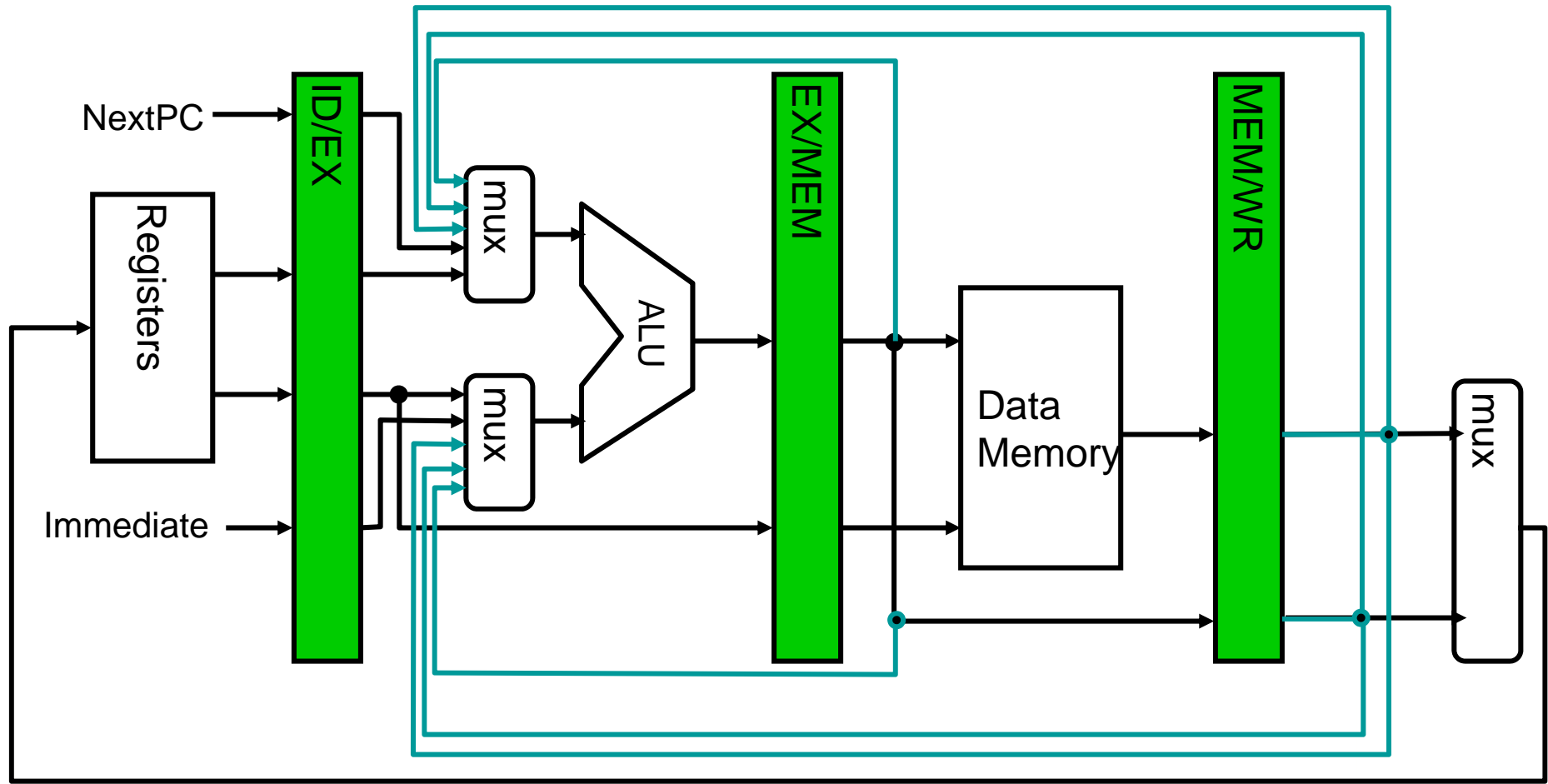
```
I: sub r1, r4, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

- 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 later more complicated pipes

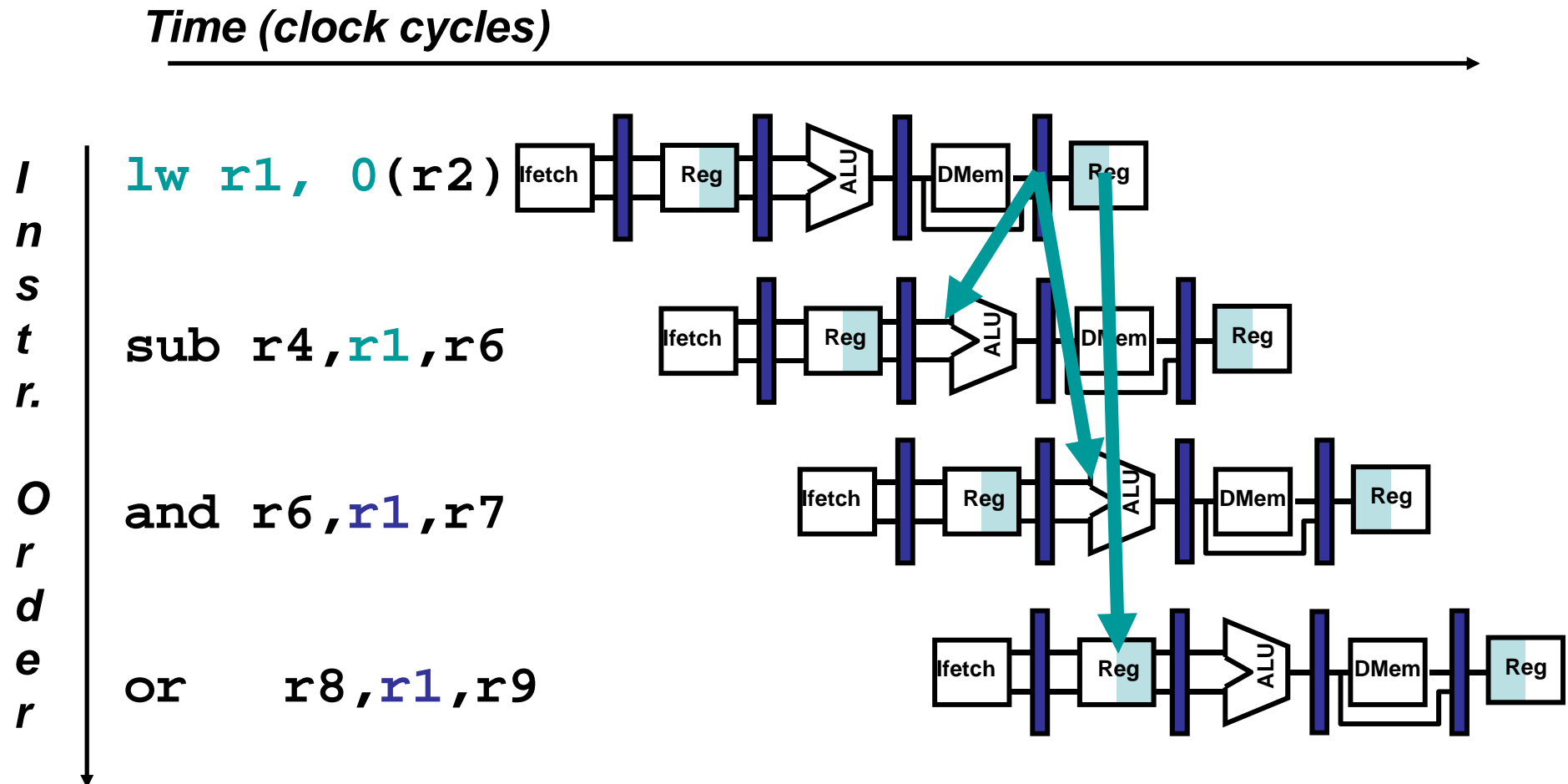
Forwarding to Avoid Data Hazard



HW Change for Forwarding



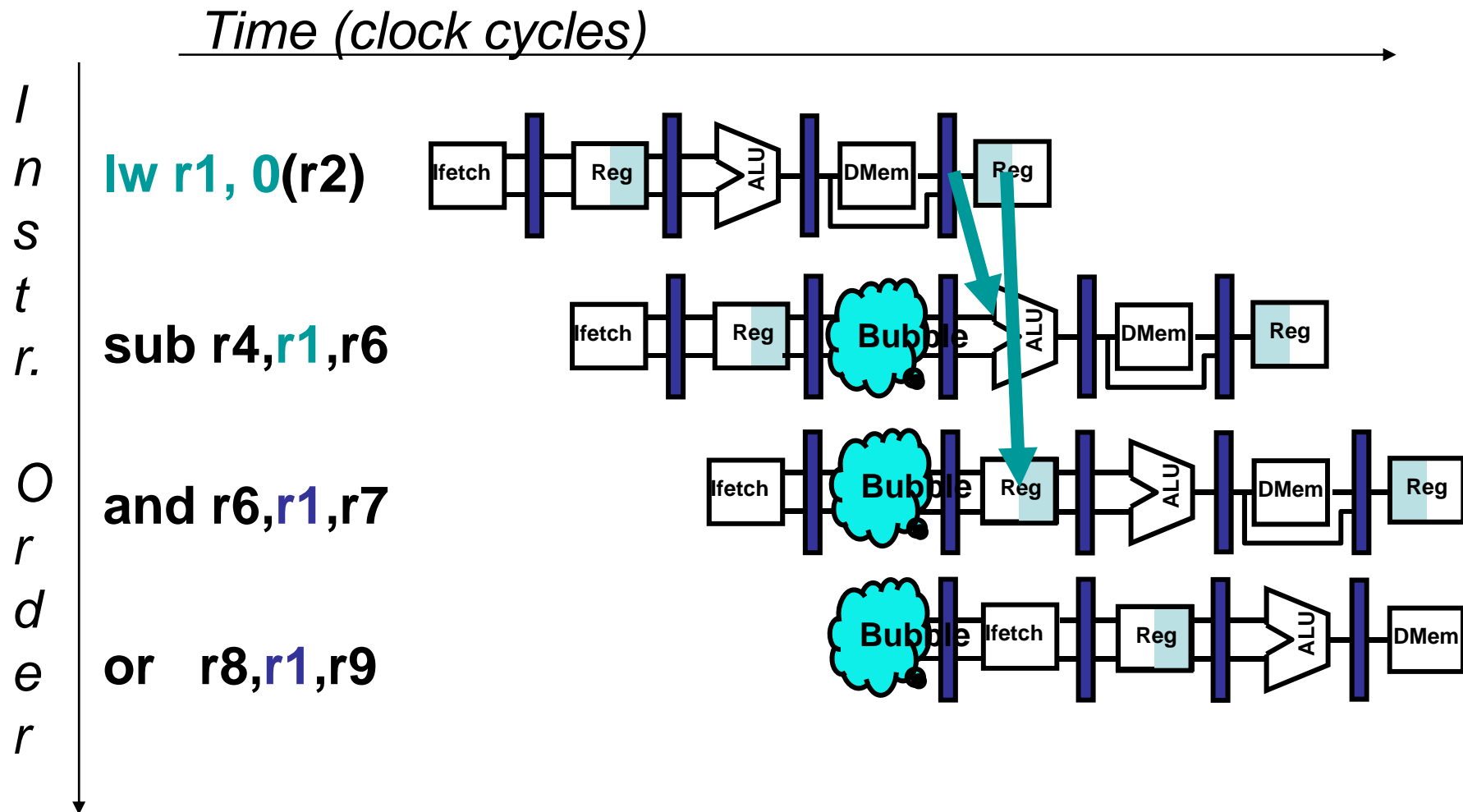
Data Hazard Even with Forwarding



Resolving this load hazard

- Adding hardware? ... not
 - Detection?
 - Compilation techniques?
-
- What is the cost of load delays?

Data Hazard Even with Forwarding



Software Scheduling to Avoid Load Hazards

Try producing fast code for

$a = b + c;$

$d = e - f;$

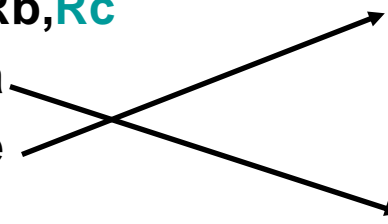
assuming $a, b, c, d, e,$ and f in memory.

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
```

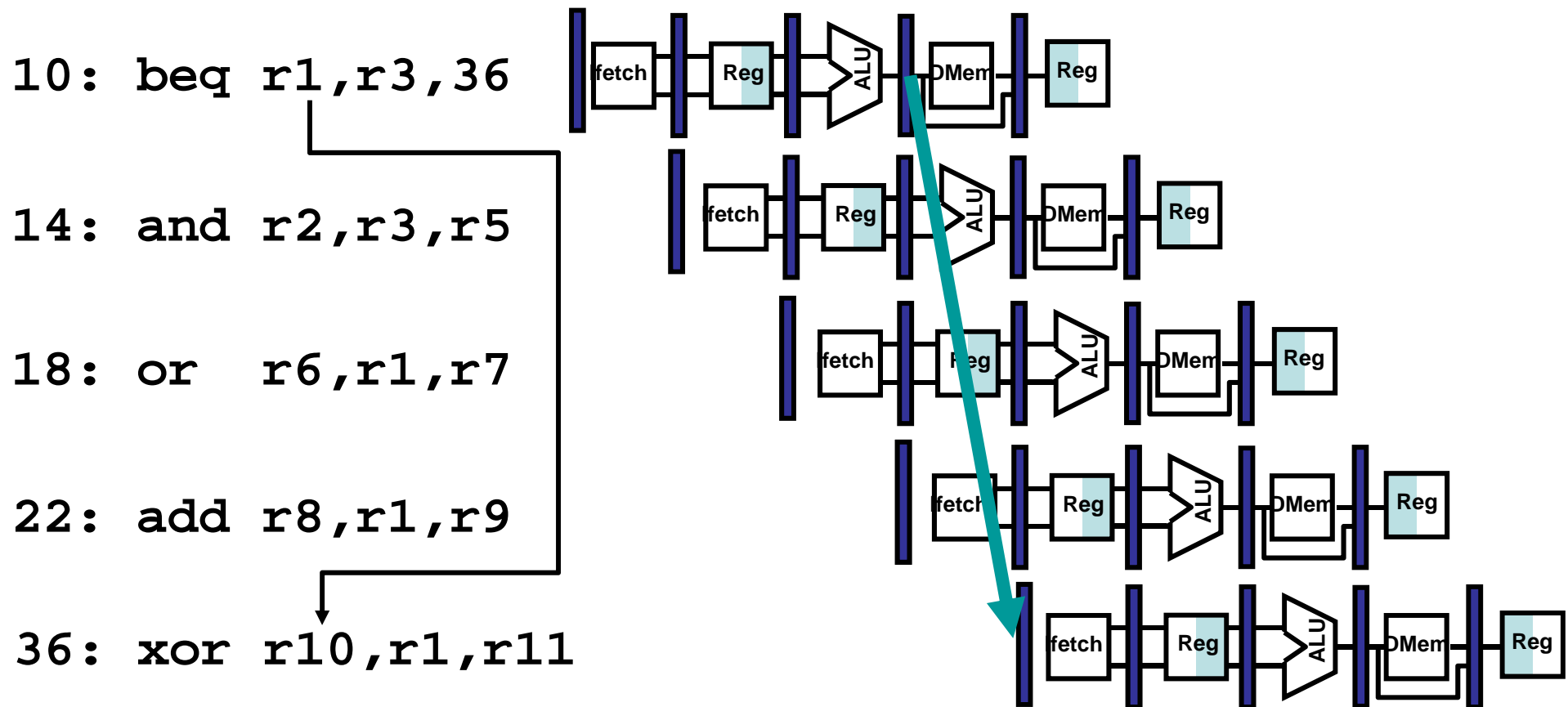
Fast code:

```
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
```



Control Hazard on Branches

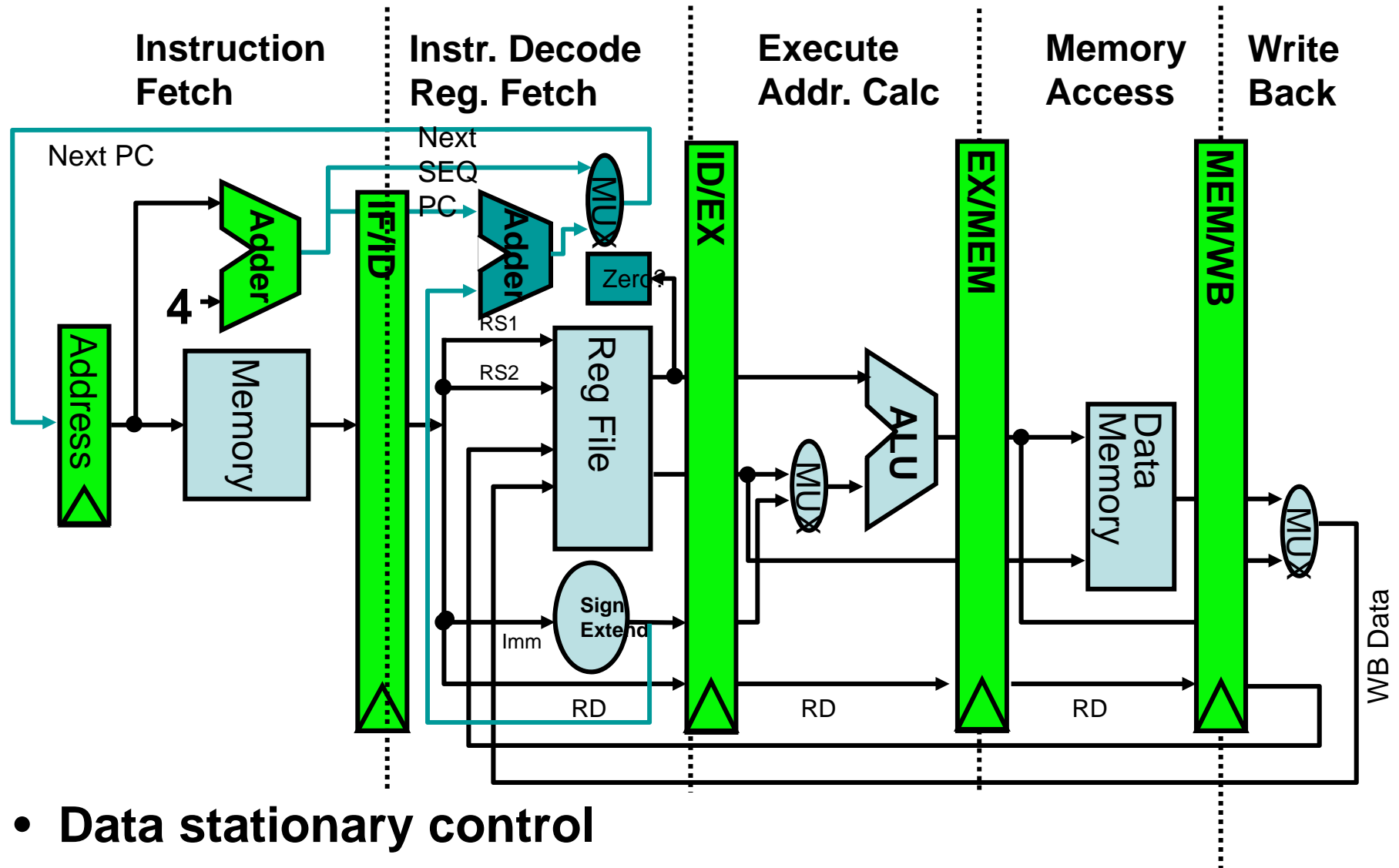
Three Stage Stall



Example: Branch Stall Impact

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

Pipelined MIPS Datapath



- **Data stationary control**
 - local decode for each instruction phase / pipeline stage

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

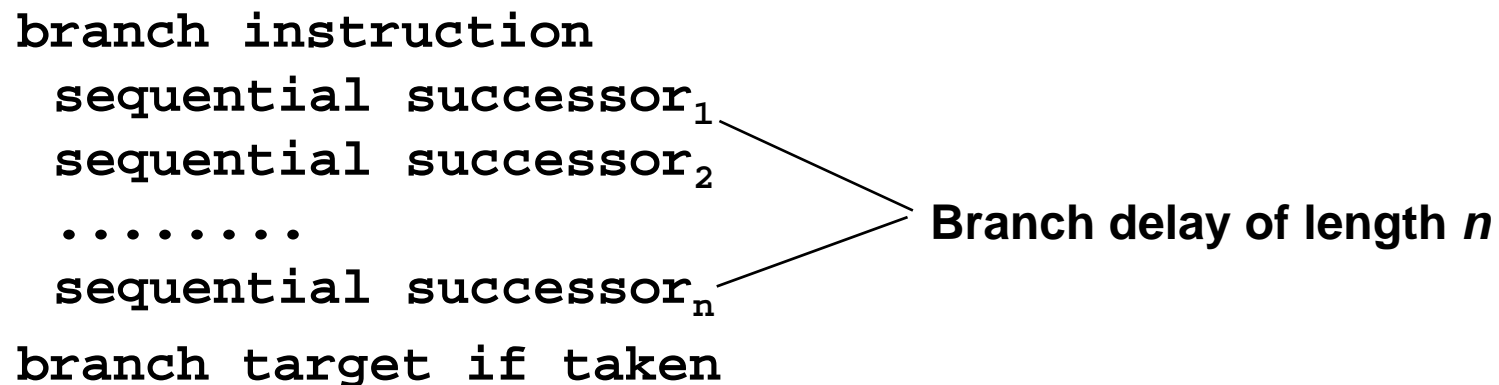
#3: Predict Branch Taken

- 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

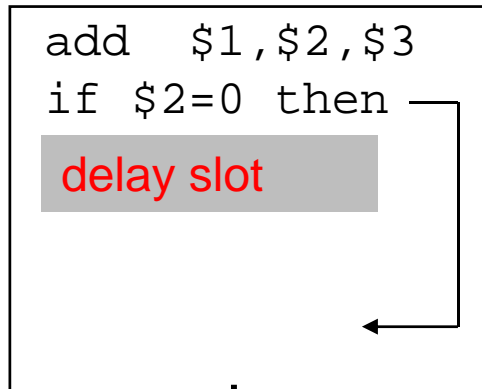
- Define branch to take place **AFTER** a following instruction



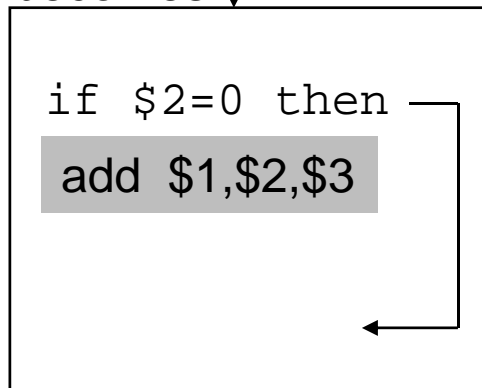
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Scheduling Branch Delay Slots (Fig A.14)

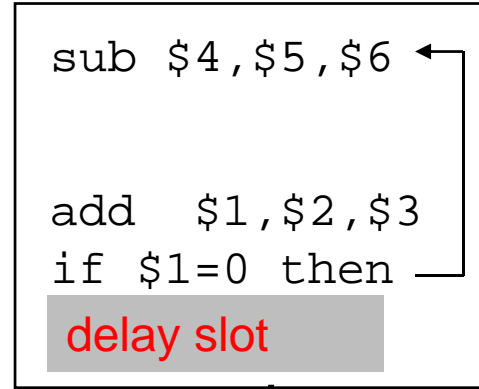
A. From before branch



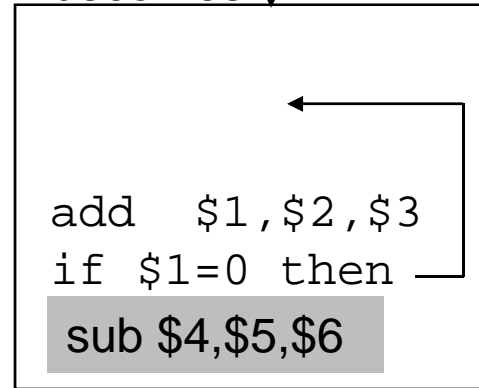
becomes



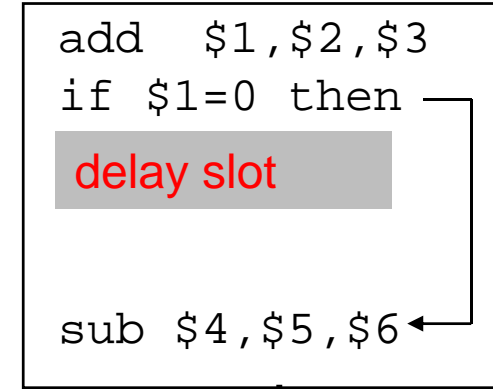
B. From branch target



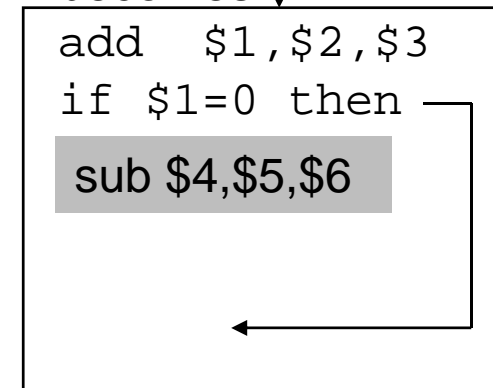
becomes



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% x 80%) of slots usefully filled
- **Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay 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

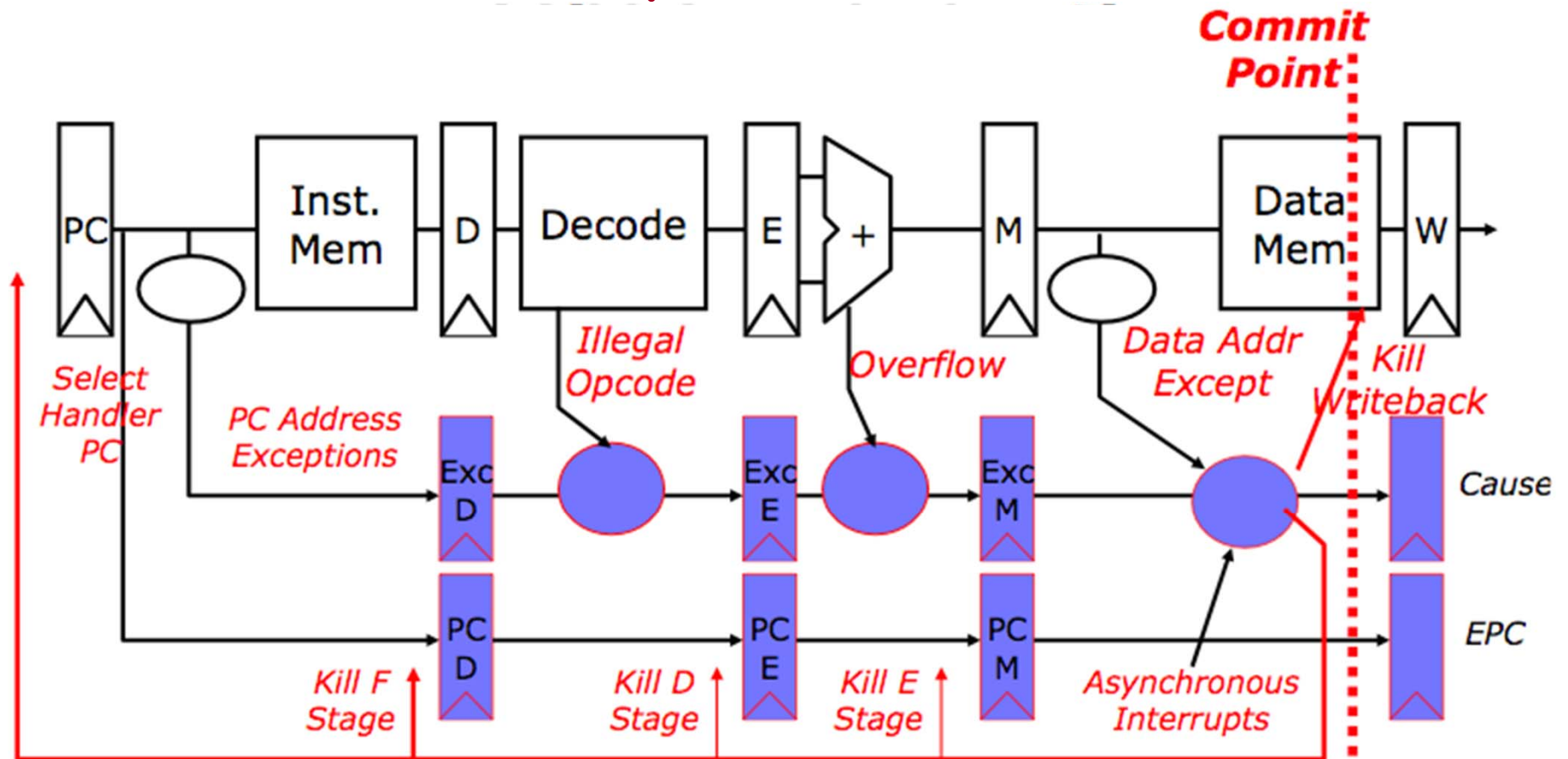
Delayed Branch

- **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
 - Canceling 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)**

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-Order Pipelines



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

Recall: 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: Evaluating Branch Alternatives

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

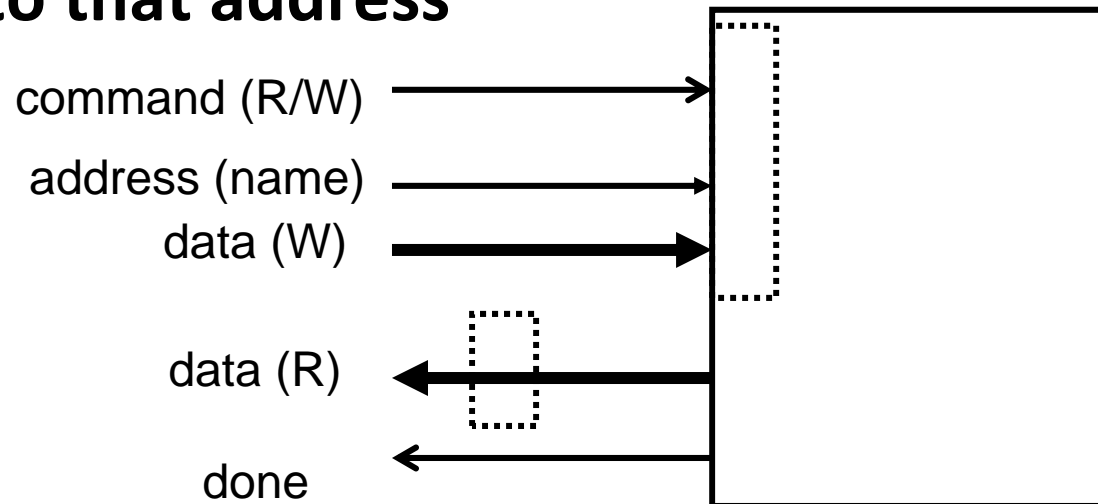
Assume:

**Conditional & Unconditional = 14%, 65%
change PC**

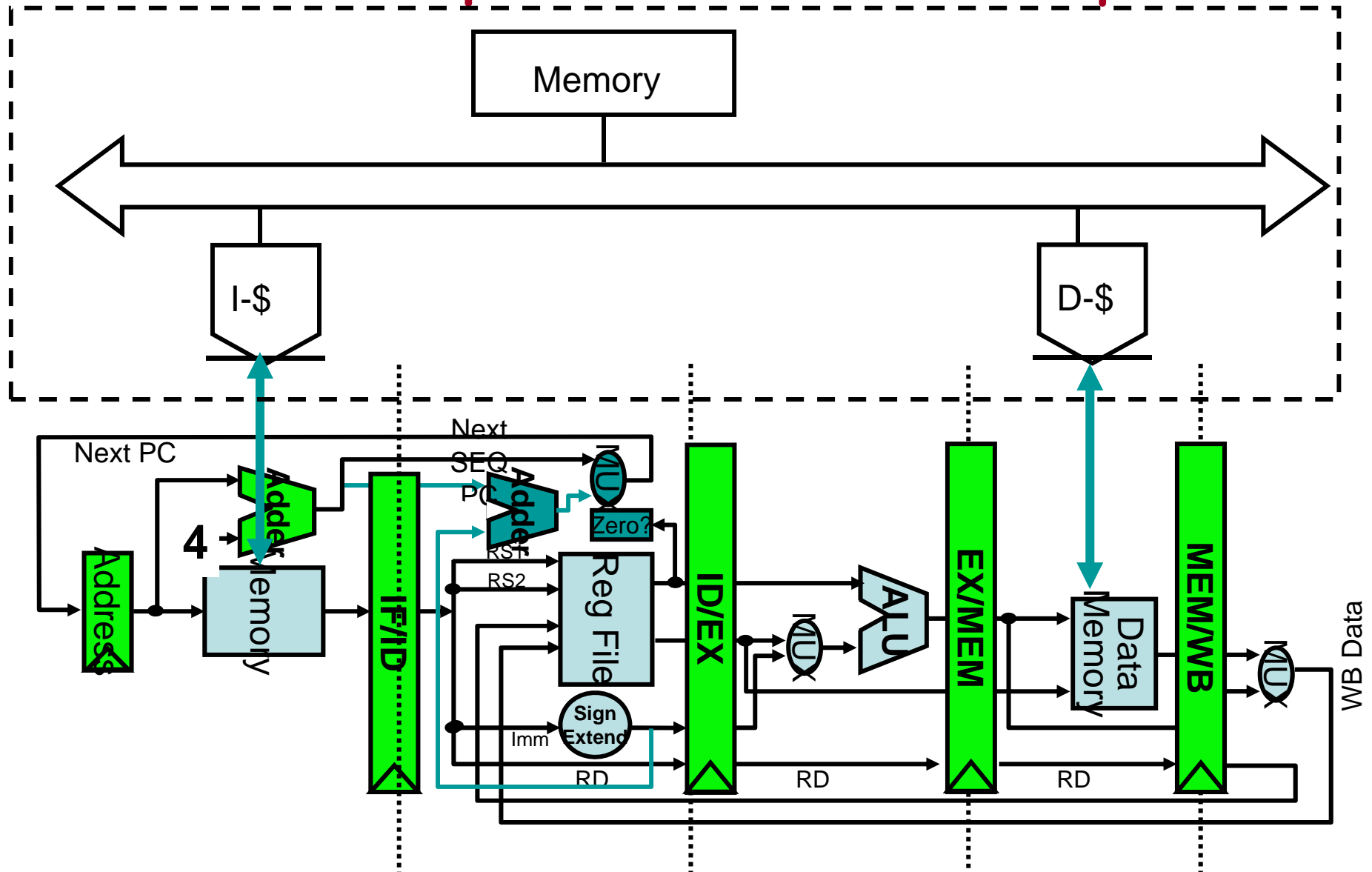
<i>Scheduling scheme</i>	<i>Branch penalty</i>	<i>CPI</i>	<i>speedup v. stall</i>
Stall pipeline	3	1.42	1.0
Predict taken	1	1.14	1.26
Predict not taken	1	1.09	1.29
Delayed branch	0.5	1.07	1.31

The Memory Abstraction

- **Association of <name, value> pairs**
 - typically named as byte addresses
 - often values aligned on multiples of size
- **Sequence of Reads and Writes**
- **Write binds a value to an address**
- **Read of addr returns most recently written value bound to that address**



Relationship of Caches and Pipeline



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