

# What's ILP

- Architectural technique that allows the overlap of individual machine operations ( **add, mul, load, store** ...)
- Multiple operations will execute in **parallel** (simultaneously)
- Goal: Speed Up the execution

# Approach

Approaches to exploiting ILP:

- (1) an approach that relies on **hardware** to help discover and exploit the parallelism dynamically,
- (2) an approach that relies on **software** technology to find parallelism statically at compile time .

# Data Dependences and Hazards

- To exploit instruction-level parallelism we **must determine** which instructions can be executed in parallel.
- If two instructions are parallel (independent), they can execute **simultaneously**.
- If two instructions are dependent, they are not parallel and must be **executed in order**
- Key : Is to determine whether an instruction is dependent on another instruction.

# Types of Dependencies

- Data dependence
- Name dependencies
  - Output dependence
  - Anti-dependence
- Control Dependence

# Data Dependences

- An instruction j is data dependent on instruction i if either of the following hold:
  - instruction i produces a result that may be used by instruction j , or
  - instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i

LOOP LD F0, 0(R1)

ADD F4, F0, F2

SD F4, 0(R1) (STORE)

SUB R1, R1, -8

BNE R1, R2, LOOP

# Data Dependences

- Dependences are a property of programs
- If two instructions are data dependent they can not execute simultaneously
- A dependence results in a hazard and the hazard causes a stall
- Data dependences may occur through registers or memory

A dependence can be overcome in two different ways:

- maintaining the dependence but avoiding a hazard, and
- eliminating a dependence by transforming the code.

# Name Dependences

- A name dependence occurs when two instructions use the same register or memory location, called a name.
- There are two types of name dependences between an **instruction i** that **precedes instruction j** in program order:

## Output dependence

- When instruction **I and J write the same** register or memory location. The ordering must be preserved to leave the correct value in the register

I :add r7,r4,r3

J :div r7,r2,r8



## Anti-dependence

When instruction **j** **writes** a register or memory location that instruction **i** **reads**

i: add r6,r5,r4

j: sub r5,r8,r11

- Solution

Renaming can be more easily done for register operands, where it is called **register renaming**

# Data Hazards

- A **hazard** exists whenever there is a name or data dependence between instructions,
- Because of the dependence, we must preserve **program order**

# Data Hazards

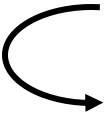
Execution Order is:

**Instr<sub>i</sub>**

**Instr<sub>j</sub>**

## Read After Write (RAW)

**Instr<sub>j</sub> tries to read operand before Instr<sub>i</sub> 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:

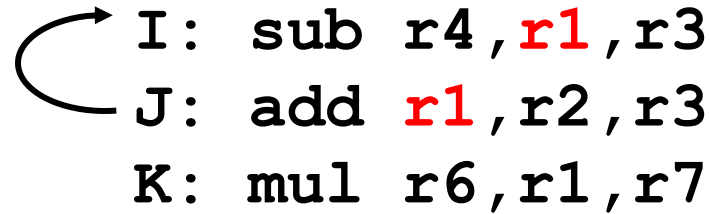
**Instr<sub>i</sub>**

**Instr<sub>j</sub>**

## Write After Read (WAR)

**Instr<sub>j</sub>** tries to write operand before Instr<sub>i</sub> reads it

- 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

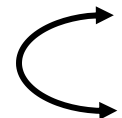
Execution Order is:

**Instr<sub>i</sub>**  
**Instr<sub>j</sub>**

## Write After Write (WAW)

Instr<sub>j</sub> tries to write operand before Instr<sub>i</sub> writes it

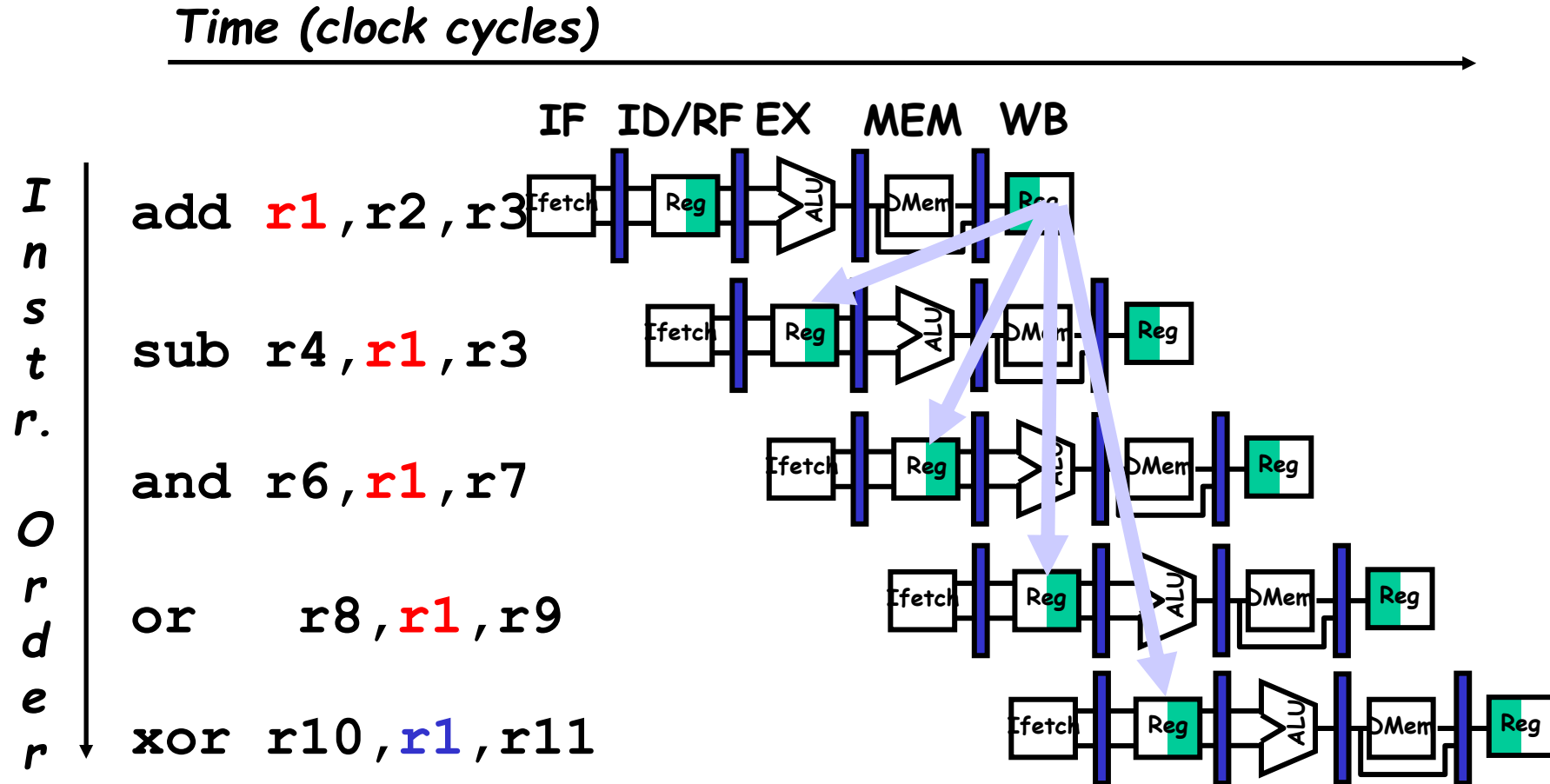
- Leaves wrong result ( Instr<sub>i</sub> not Instr<sub>j</sub> )



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

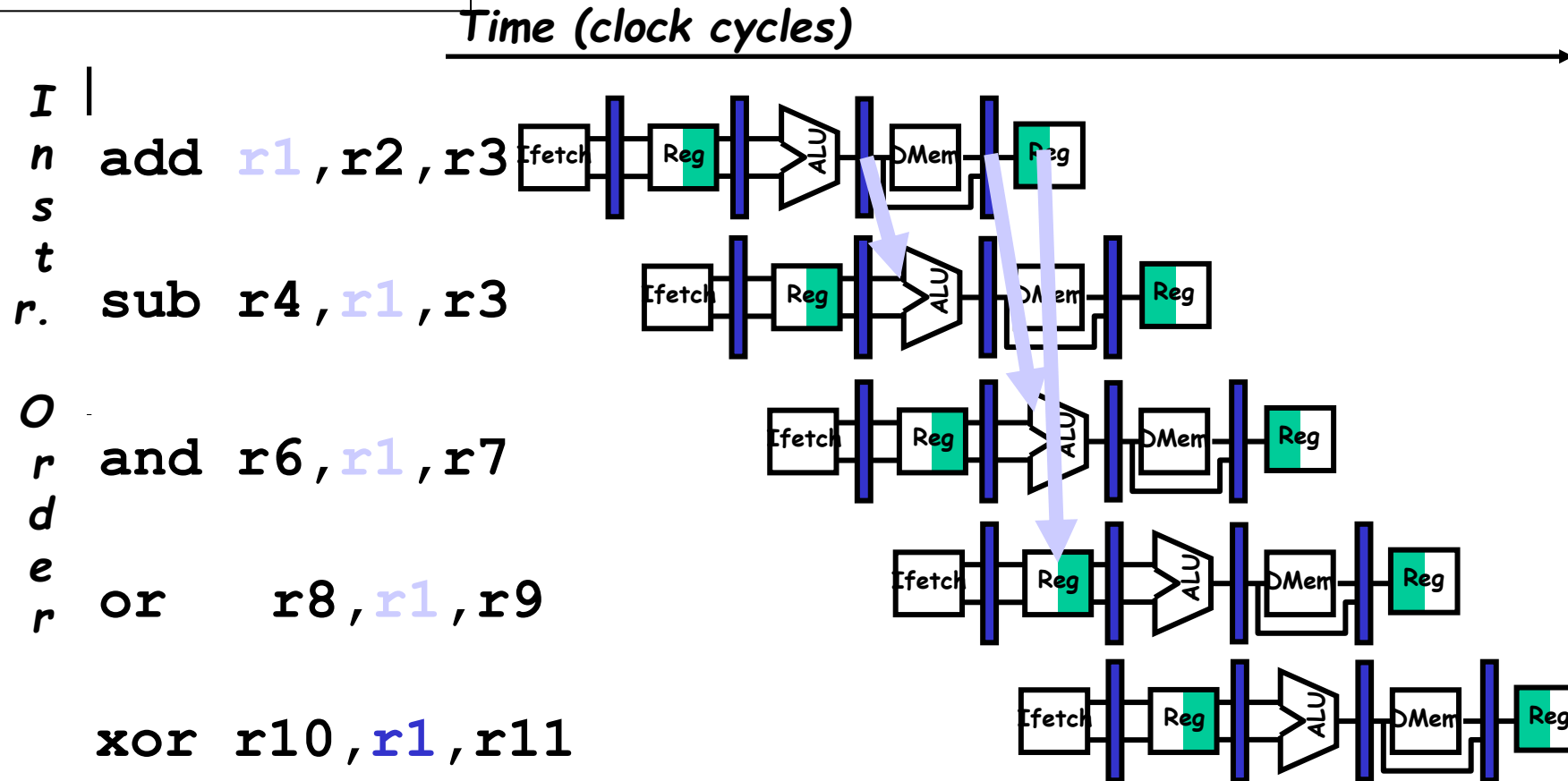


The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

# Data Hazards

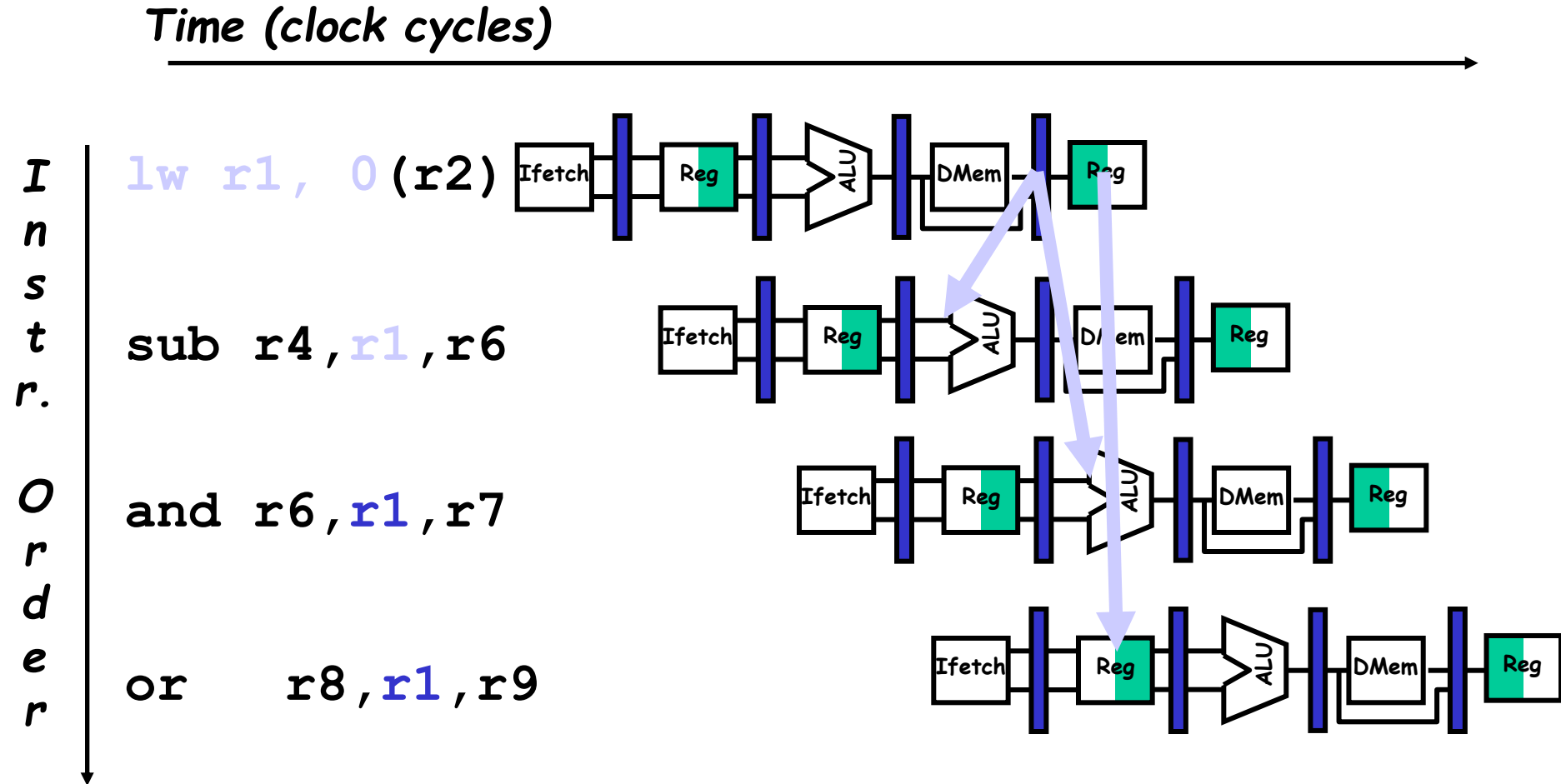
Forwarding To Avoid  
Data Hazard

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.



# Data Hazards

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

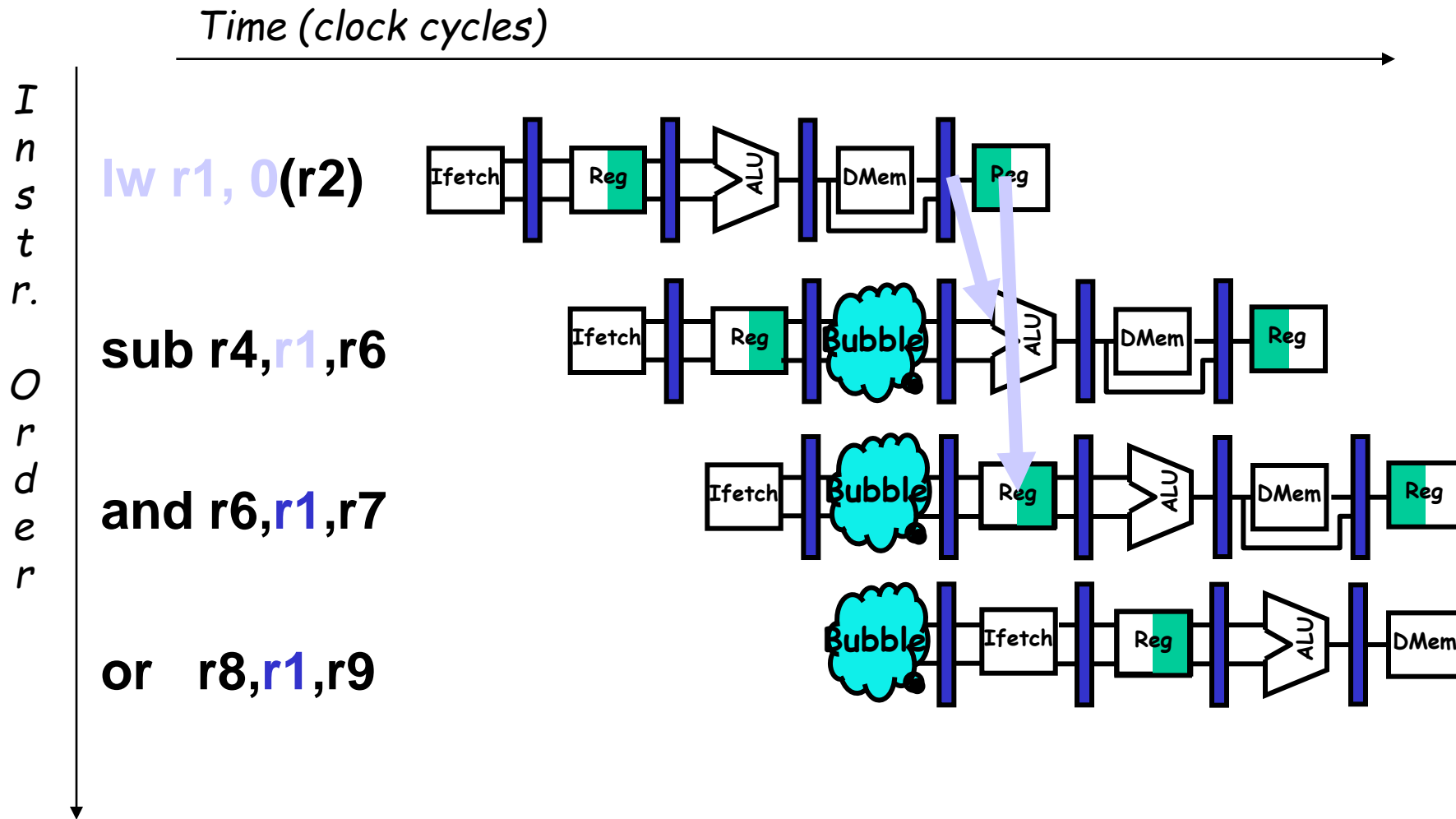


There are some instances where hazards occur, even with forwarding.



# 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

## Pipeline Scheduling

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

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

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
```

# Control Dependences

- A control dependence determines the ordering of an instruction  $i$ , with respect to a branch instruction so that the instruction  $i$  is executed in correct program order.
- Example:

```
    If p1 {  
        S1;  
    };  
    If p2 {  
        S2;  
    };
```

S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1

## Two constraints imposed by control dependences

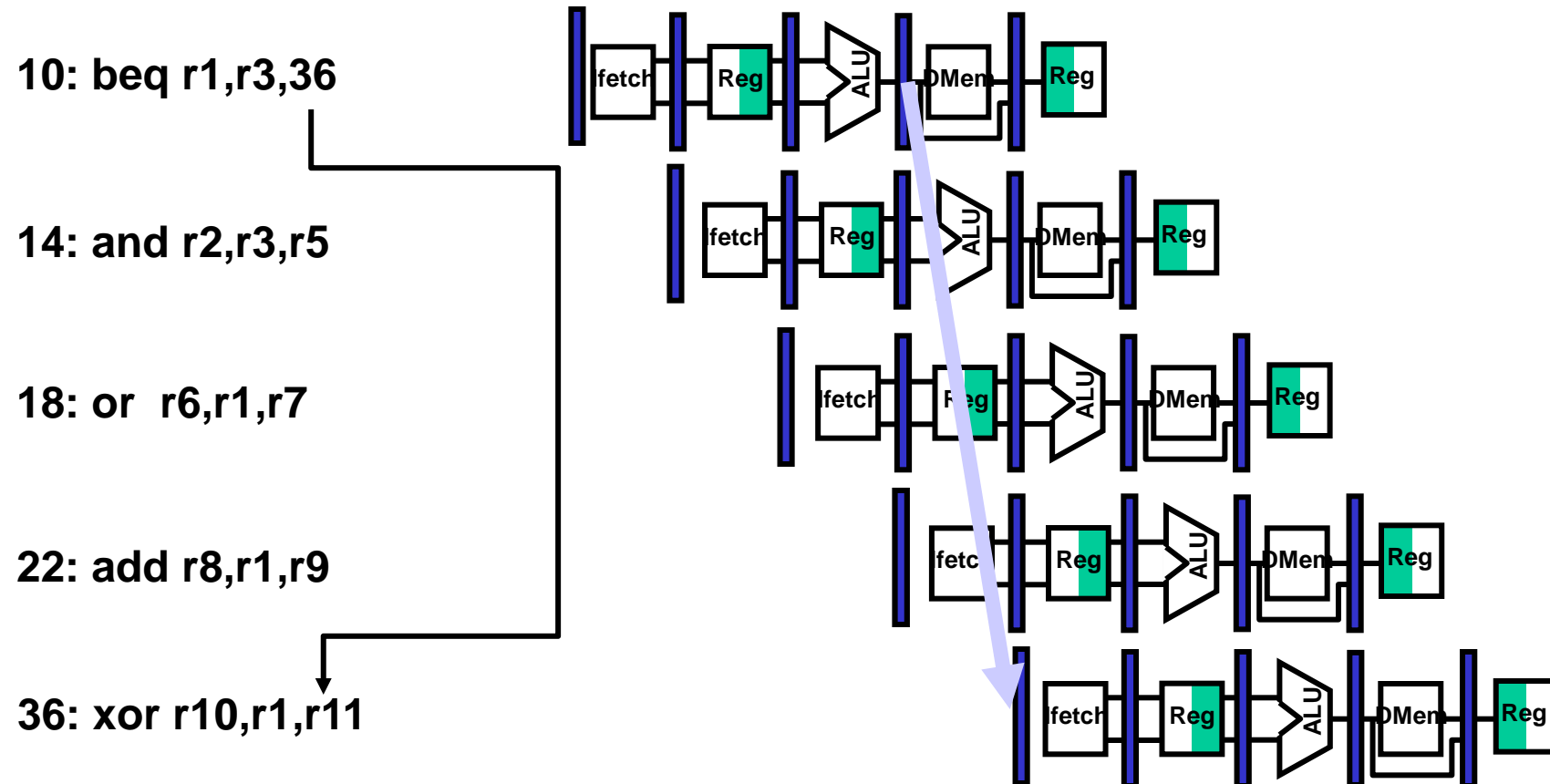
1. An instruction that is control dependent on a branch cannot be moved *before* the branch .

For example, we cannot take an instruction from the then portion of an if statement and move it before the if statement.

2. An instruction that is not control dependent on a branch cannot be moved *after* the branch.

For example, we cannot take a statement before the if statement and move it into the then portion.

# Control Hazards

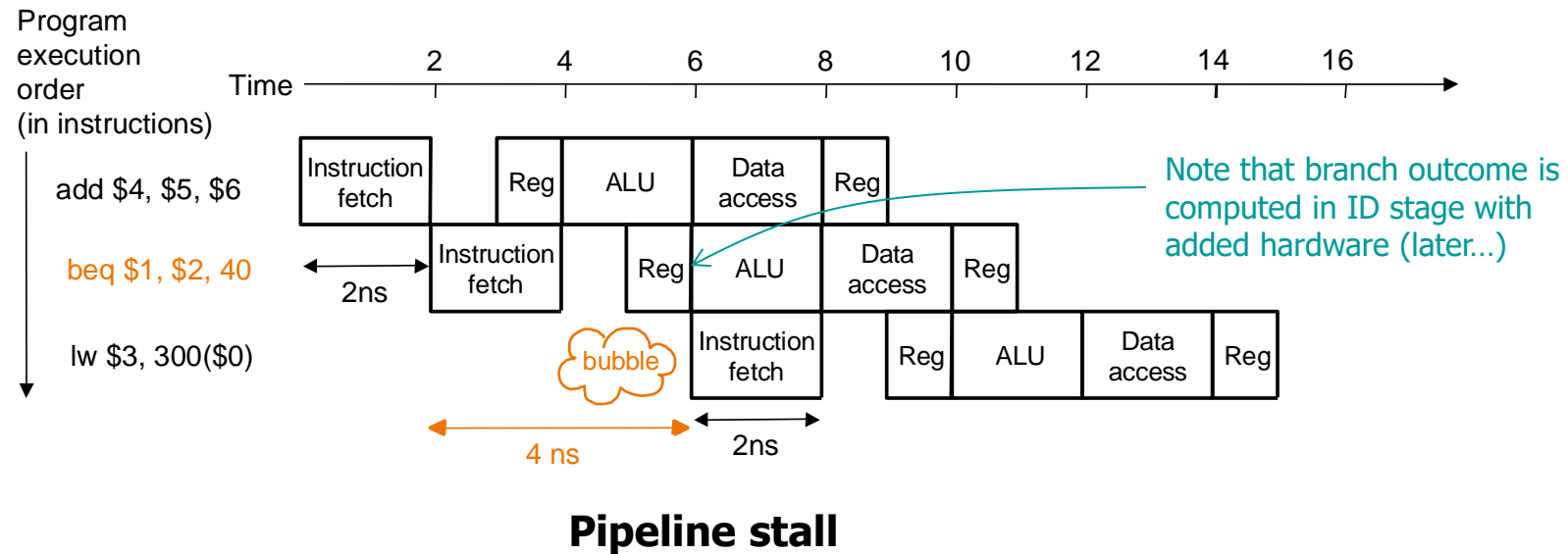


# Instruction Hazards

- Instruction (control) hazard – a delay in the availability of an instruction causes the pipeline to stall.
- Branch Instruction.

# Control Hazards

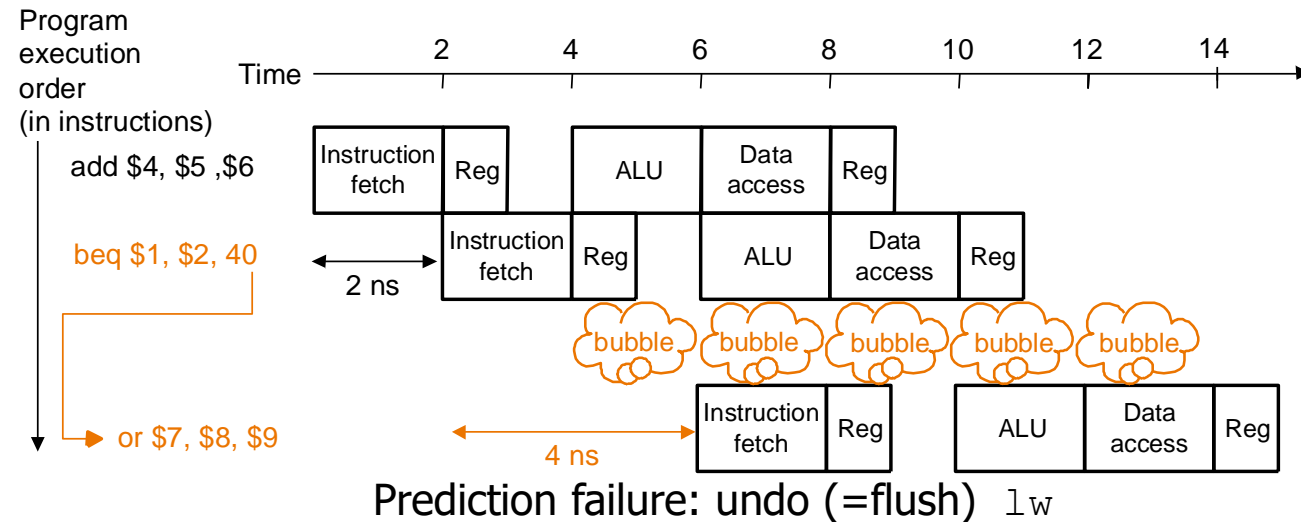
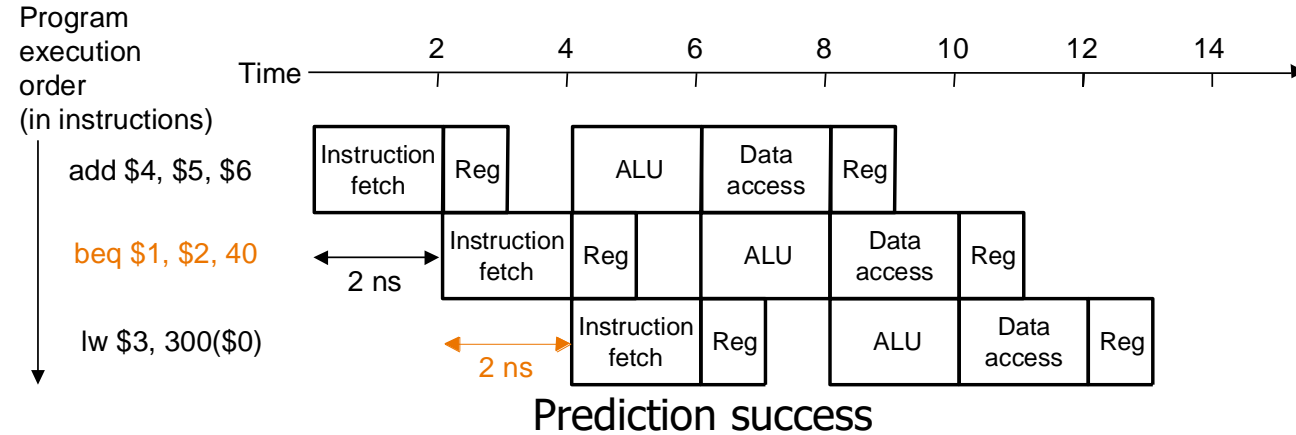
- *Control hazard*: need to make a decision based on the result of a previous instruction still executing in pipeline
- Solution 1 *Stall* the pipeline





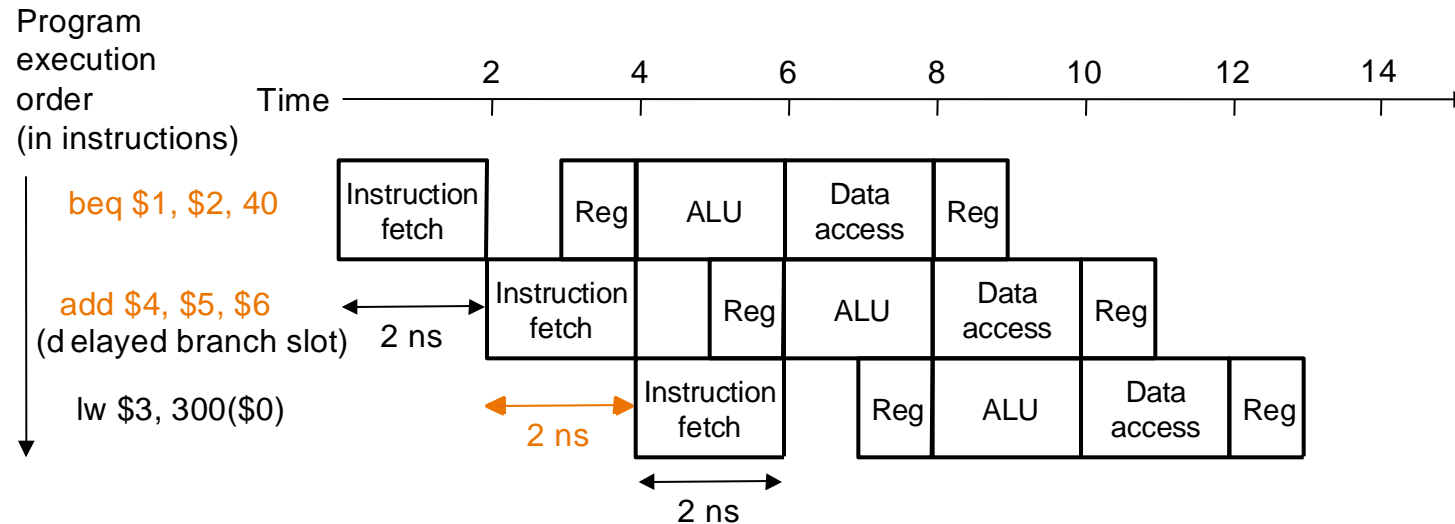
# Control Hazards

- Solution 2 *Predict* branch outcome
  - e.g., predict *branch-not-taken* :



# Control Hazards

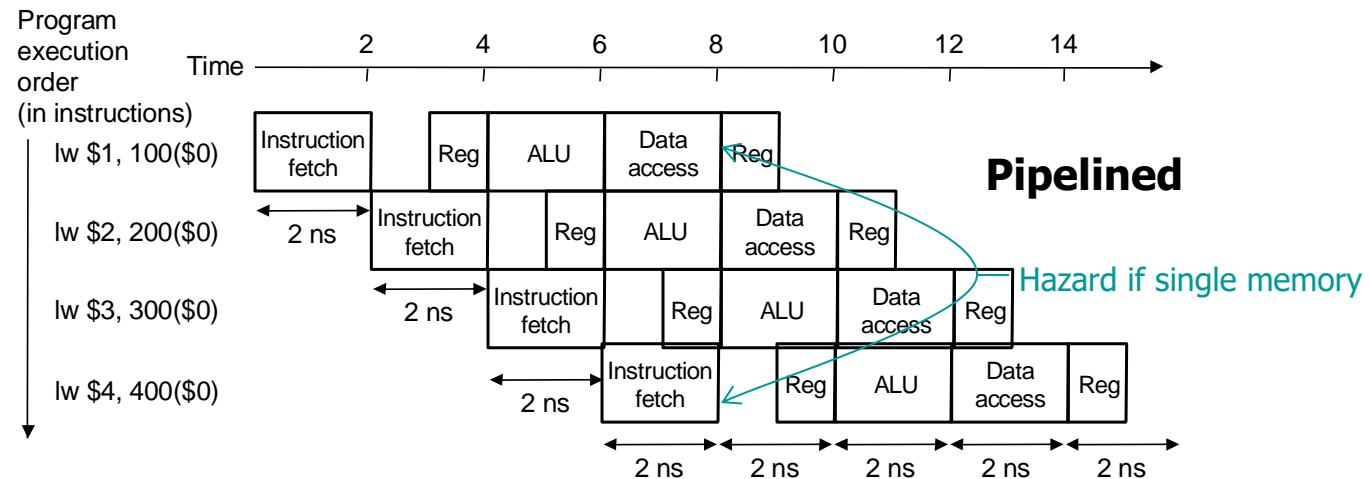
- Solution 3 *Delayed branch*: always execute the sequentially next statement with the branch executing after one instruction delay – compiler's job to find a statement that can be put in the slot that is independent of branch outcome
  - MIPS does this



**Delayed branch** beq is followed by add that is independent of branch outcome

# Structural Hazards

- *Structural hazard*: inadequate hardware to simultaneously support all instructions in the pipeline in the same clock cycle
- E.g., suppose *single – not separate* – instruction and data memory in pipeline below with *one read port*
  - then a structural hazard between first and fourth `lw` instructions



- *MIPS was designed to be pipelined*: structural hazards are easy to avoid!

# Data Hazards: A Classic Example

Identify the data dependencies in the following code. Which of them can be resolved through forwarding? ( Use 5 Stage pipeline).

SUB R2, R1, R3

OR R12, R2, R5

SW R13, 100(R2)

ADD R14, R2, R2

LW R15, 100(R2)

ADD R4, R7, R15