

CS3350B Computer Organization

Chapter 4: Instruction-Level Parallelism

Part 2: Pipeline Hazards

Iqra Batool

Department of Computer Science
University of Western Ontario, Canada

Wednesday March 4, 2024

Outline

1 Overview

2 Structural Hazards

3 Data Hazards

4 Control Hazards

Pros and Cons of Pipelining

- Pipelining overlaps the execution of instructions to keep each stage of the datapath busy at all times.
 - ↳ Improves throughput but not latency.
of single instruction.
number of instruction a unit could process.
 - ↳ Might actually *increase* latency.
- Can increase clock frequency using multi-cycle datapath.
- Ideal speedup can be up to the number of stages. *ideally, increase CPI to 1*
- Ideal speed up never reached.
 - ↳ Fill time and drain time limits speedup.
 - ↳ Must account for dependencies between results of previous instructions and operands of future instructions.
 - ↳ Sometimes the same hardware is needed simultaneously by different pipeline stages and different instructions (e.g. ID and WB stages).
hardware hazard
RAW.

Categorizing Pipeline Hazards

Structural Hazards

- Conflicts in hardware/circuit use.
- Different stages or different instructions attempt to use same piece of hardware at the same time.

Data Hazards

- Dependencies between the result of an instruction and the input to another instruction.
- Data being used before it is finished being computed or written to memory/registers.

Control Hazards

- Ambiguity in the control flow of the program being executed.
- Branch instructions—if/else, loops. *no way to statically know the flow of the program.*
- Take the branch? Don't take the branch? Which instruction follows a branch instruction in the pipeline?

“Resolving” Pipeline Hazards

Not an easy task. Simplest solution: just wait or **stall**.

↳ Any hazard can always be solved by just waiting.

But:

- Ruins potential speedup.
 - ↳ Might end up being slower than a single-cycle datapath.
 - ↳ Since latency can increase in pipelining, with enough stalls becomes slower.
- Increases CPI.
- Works against entire principle of pipelining.
 - ↳ Where's the performance?
- Nonetheless, sometimes it really is the only solution.

Outline

- 1 Overview
- 2 Structural Hazards**
- 3 Data Hazards
- 4 Control Hazards

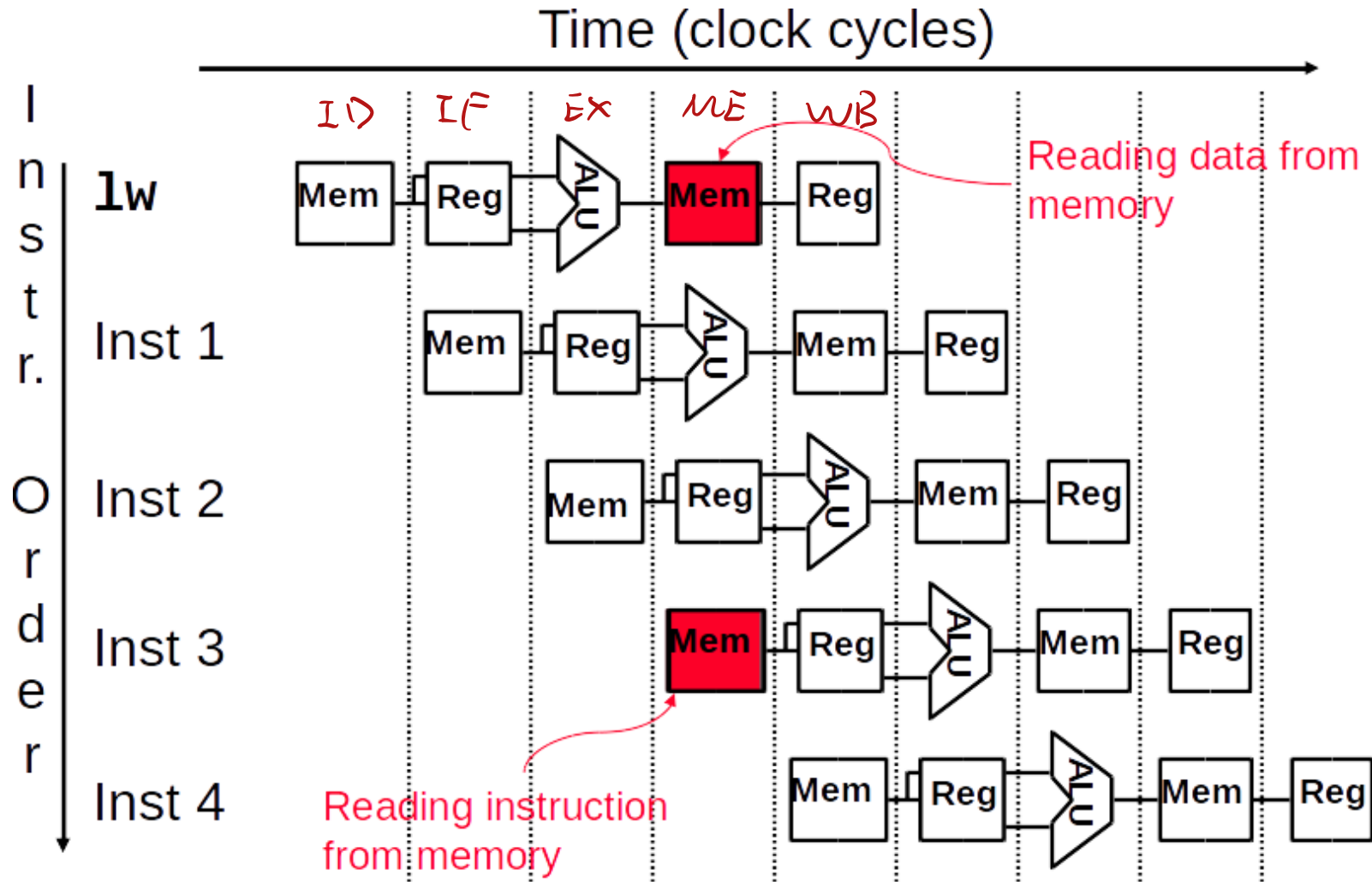
Structural Hazards: Causes and Resolutions

- Structural hazards are caused by two instructions needing to use the same hardware at the same time.
- Easiest to resolve? Just add in redundant hardware.
 - ↳ Works for combinational circuits.
 - ↳ Redundant memory would cause problems in needing to keep both consistent.
- Real structural hazards thus lie in state circuits: registers and memory.
 - ↳ IF stage and MEM stage.
 - ↳ ID stage and WB stage.

Structural Hazards In Memory (1/2)

Store data & instruction together.

Consider a unified L1 cache. Reading instructions and reading/writing data could overlap for pipelined instructions.

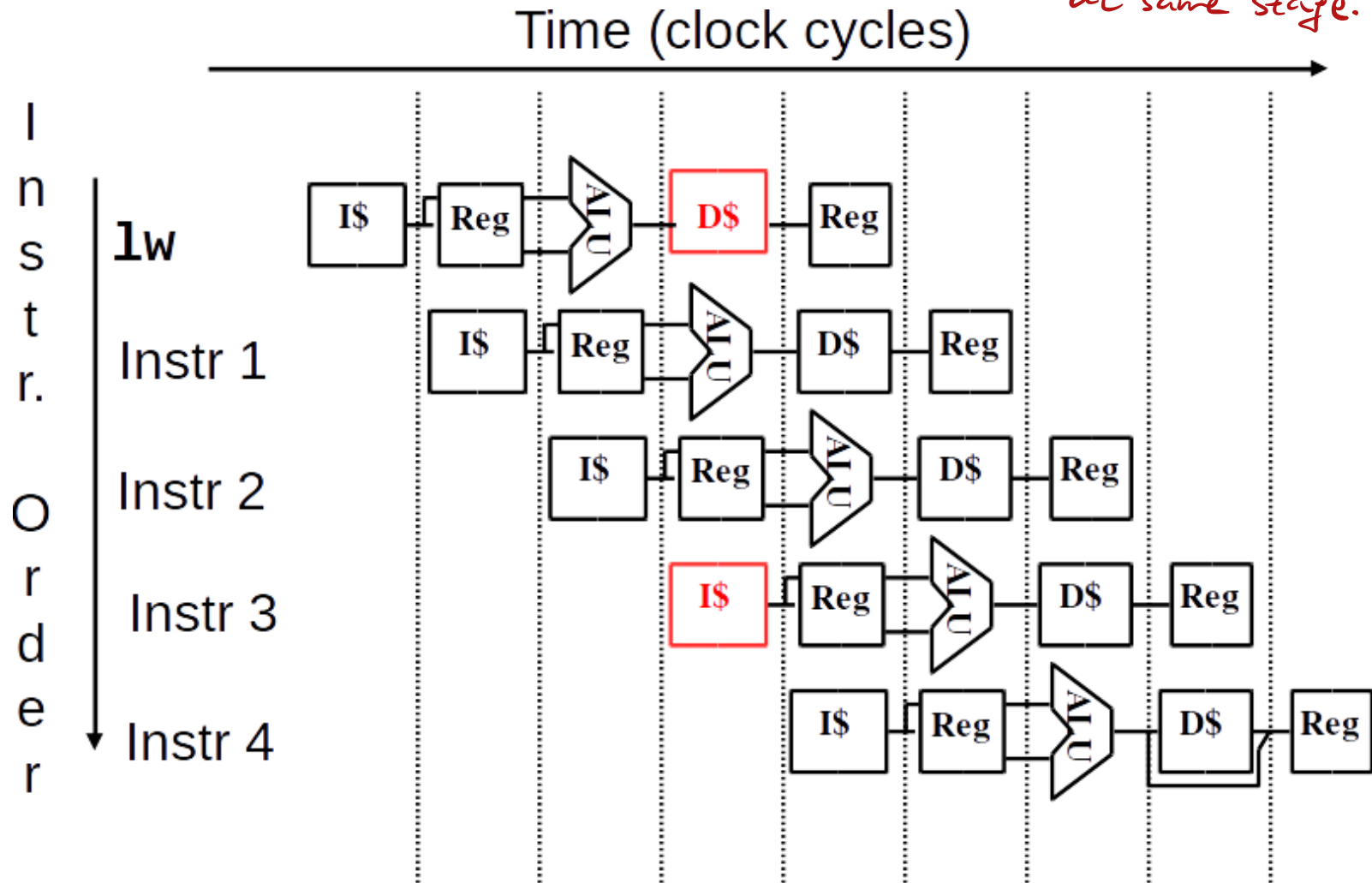


Structural Hazards In Memory (2/2)

Simple fix: separate instruction memory from data memory.

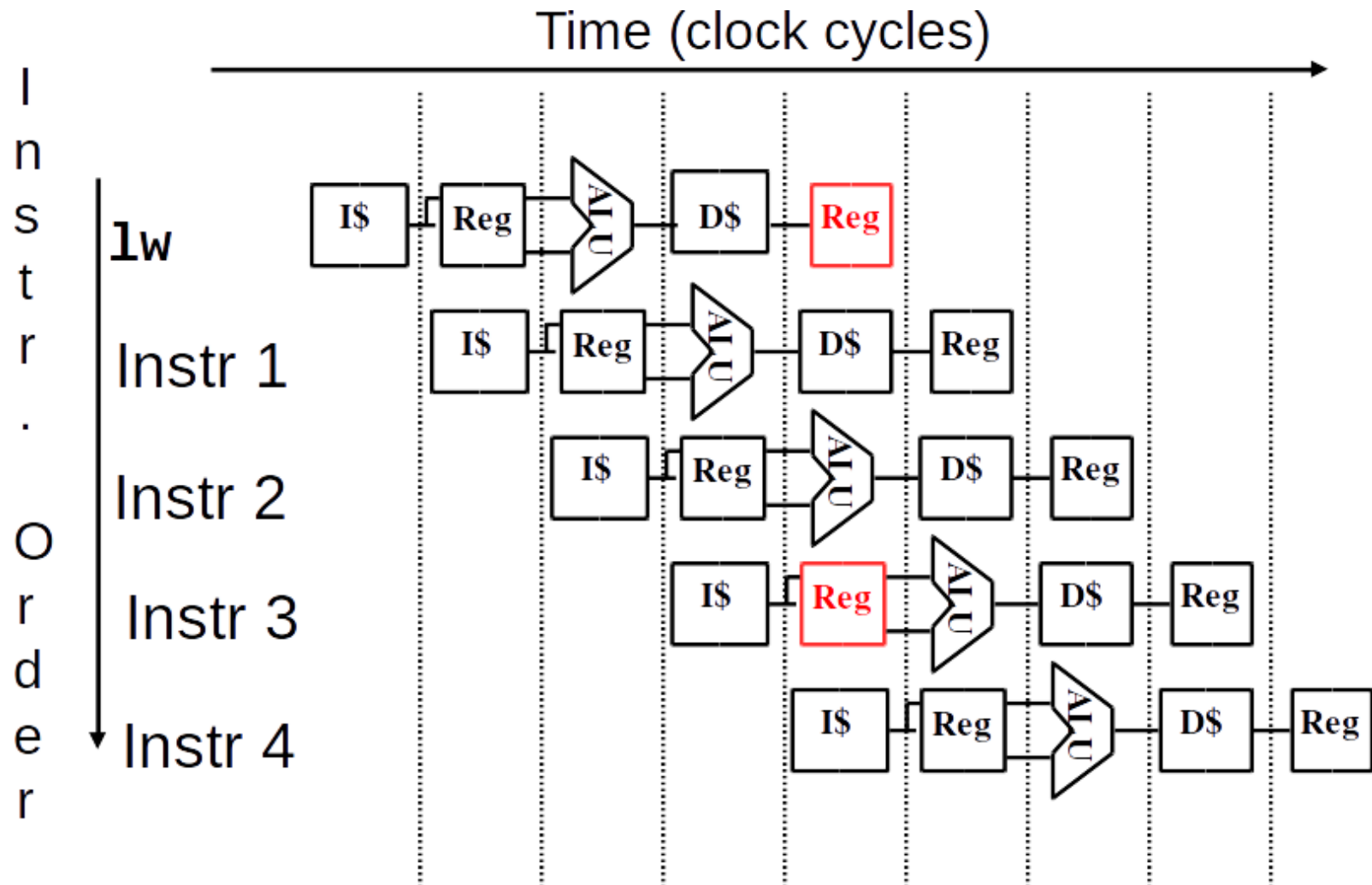
- Can use a banked cache.

*two refs access register
at same stage.*



Structural Hazards In Register File (1/2)

ID stage must read from registers while WB stage must write to registers.

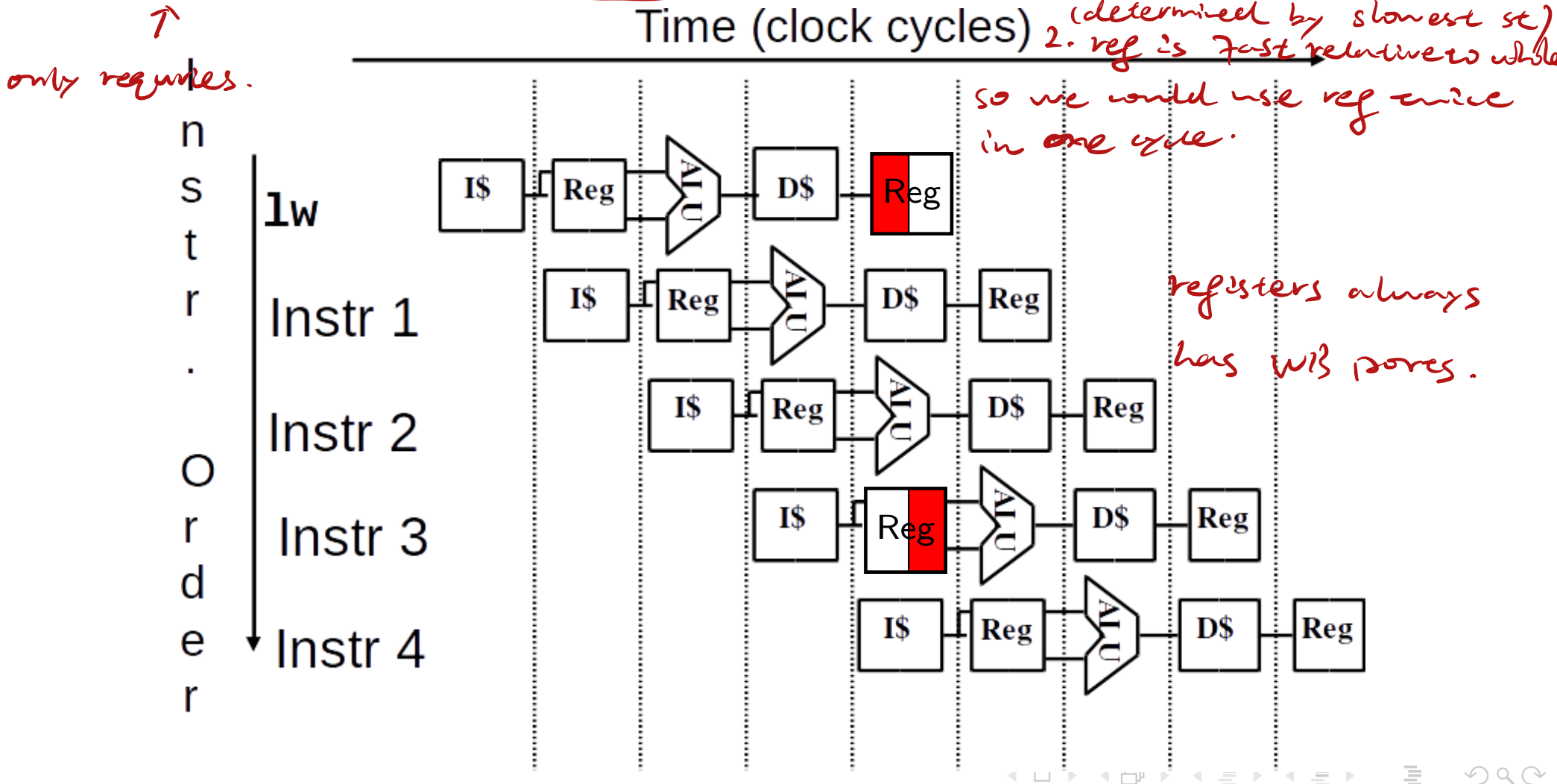


Structural Hazards In Register File (2/2)

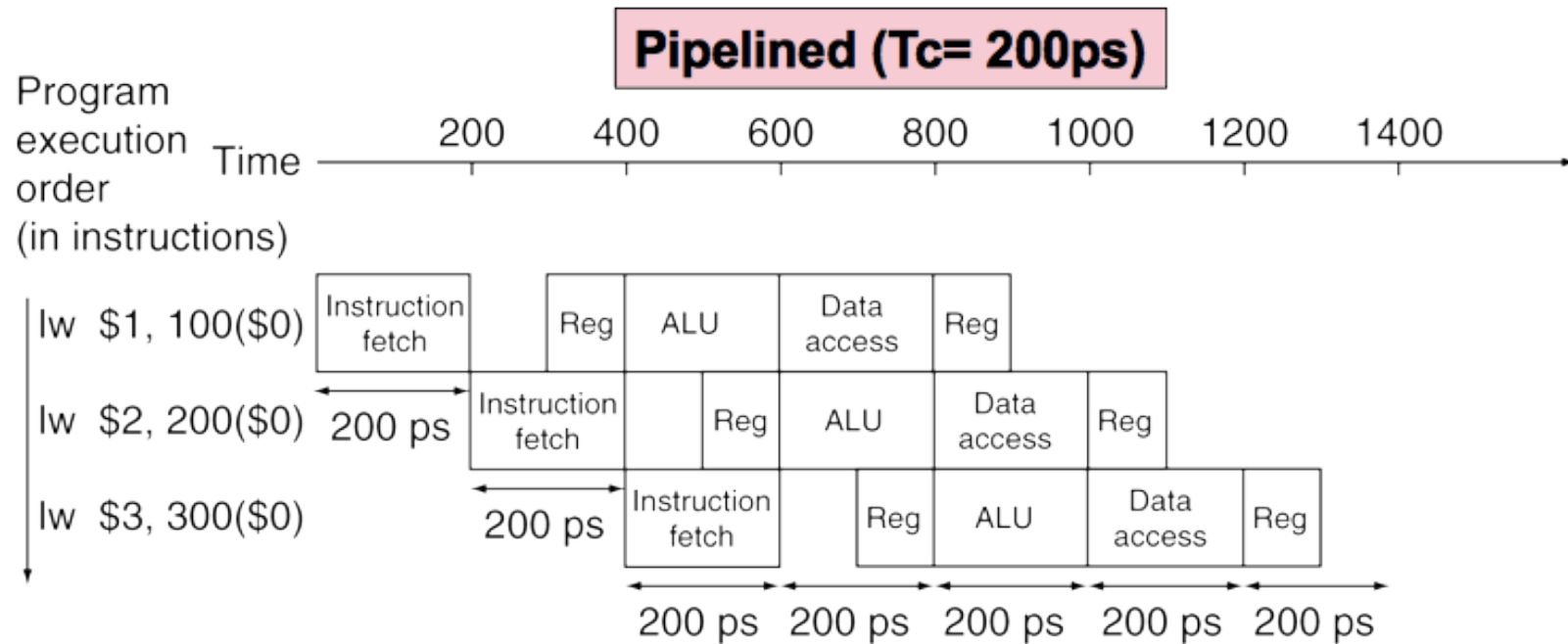
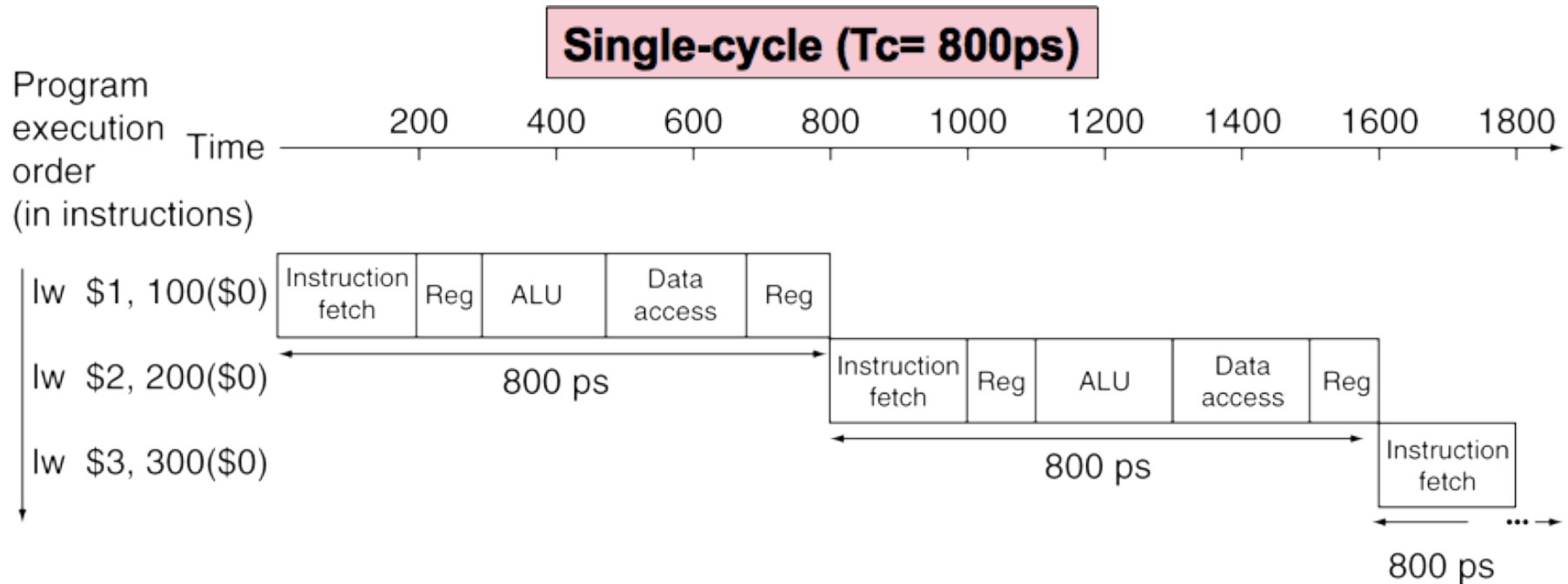
In reality, reading from register file is very fast; clock cycle is long enough to allow both ID and WB to occur within a single clock cycle.

- Needs independent read and write ports.

1. clock cycle is so long
(determined by slowest st)
2. reg is fast relative to whole
so we could use reg twice
in one cycle.



A previous diagram



Outline

- 1 Overview
- 2 Structural Hazards
- 3 Data Hazards**
- 4 Control Hazards

Data Hazards: Causes

Data hazards are caused by **dependencies** between instructions.

- **Read After Write** (RAW) dependency. Using the result of one instruction as an operand to a later instruction. *← this is the only true dependency.*
- **Read After Read** (RAR) dependency. Using the same operand to multiple instructions. *← this is not hazard.*
- **Write After Read** (WAR) dependency. Writing a result to a register after using that register as an operand in some previous instructions. *at most times it does not hazard.*
- **Write After Write** (WAW) dependency. Writing to the same register as a previous instruction writes to.
- RAW is the only true dependency. *← we want the updated data.*
- RAR is never a hazard.
- WAR, WAW only becomes a problem with out-of-order execution. (Lecture 14).

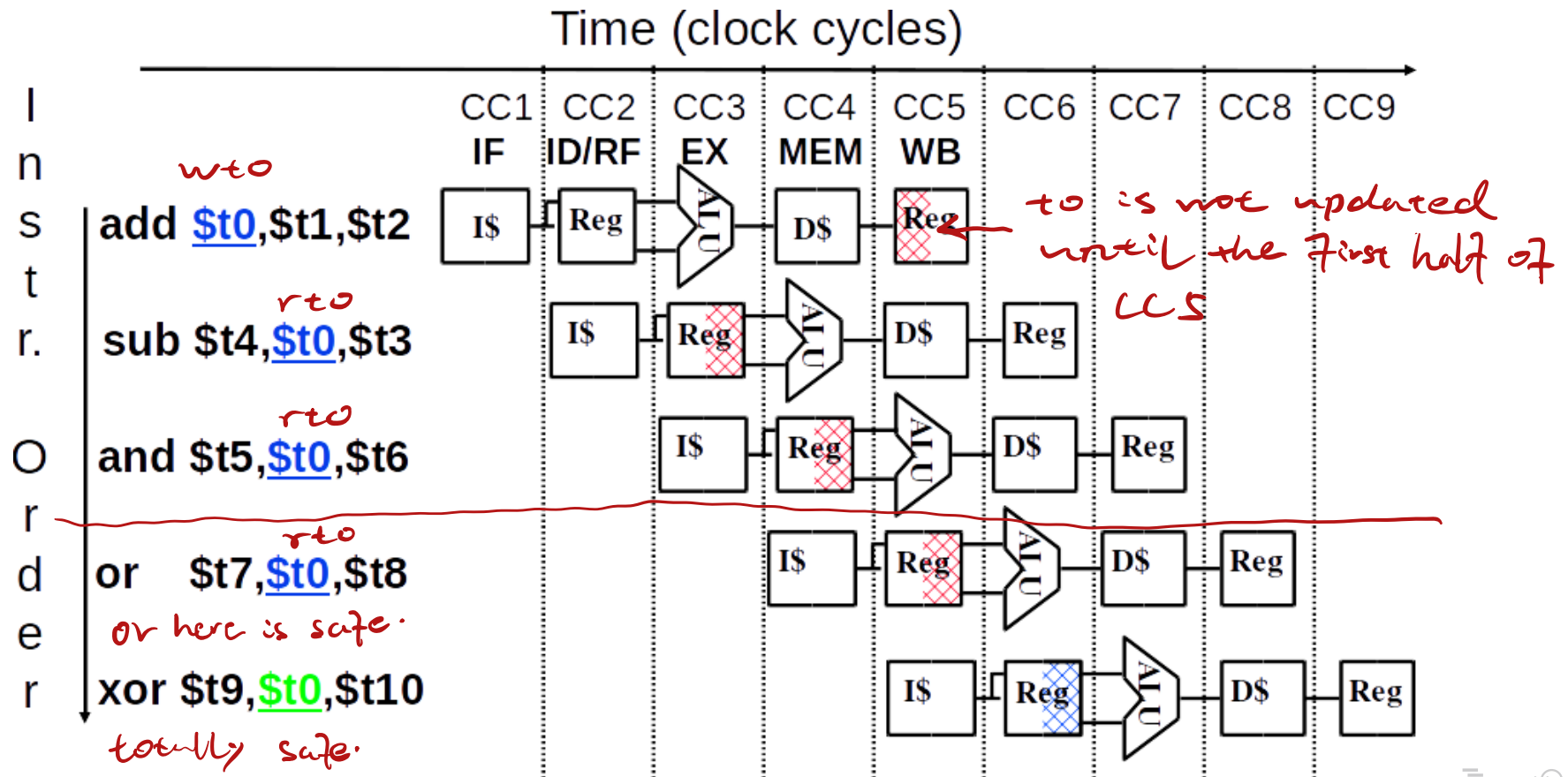
Data Hazards: Resolutions

- **Data hazards** are caused dependencies which *require resolution*.
 - ↳ Not all dependencies create hazards. If the dependent instructions are "far enough away", then there is no hazard.
- Hazards can always be solved by stalling the pipeline. *but it would reduce the performance of PID*
- Hazards can also be solved by special **forwarding** (also called bypass). *so, with special circuits,*
- Data hazards are the most common type of hazard.
 - ↳ It's the logical way to write programs: *locality*.
 - ↳ *Make the common case fast.*
 - ↳ Since locality dictates register re-use, datapath should be handle it efficiently. Typically, through forwarding.

Data Hazard Example 1 (1/3)

add produces a result which is then read by sub, and, or, xor.

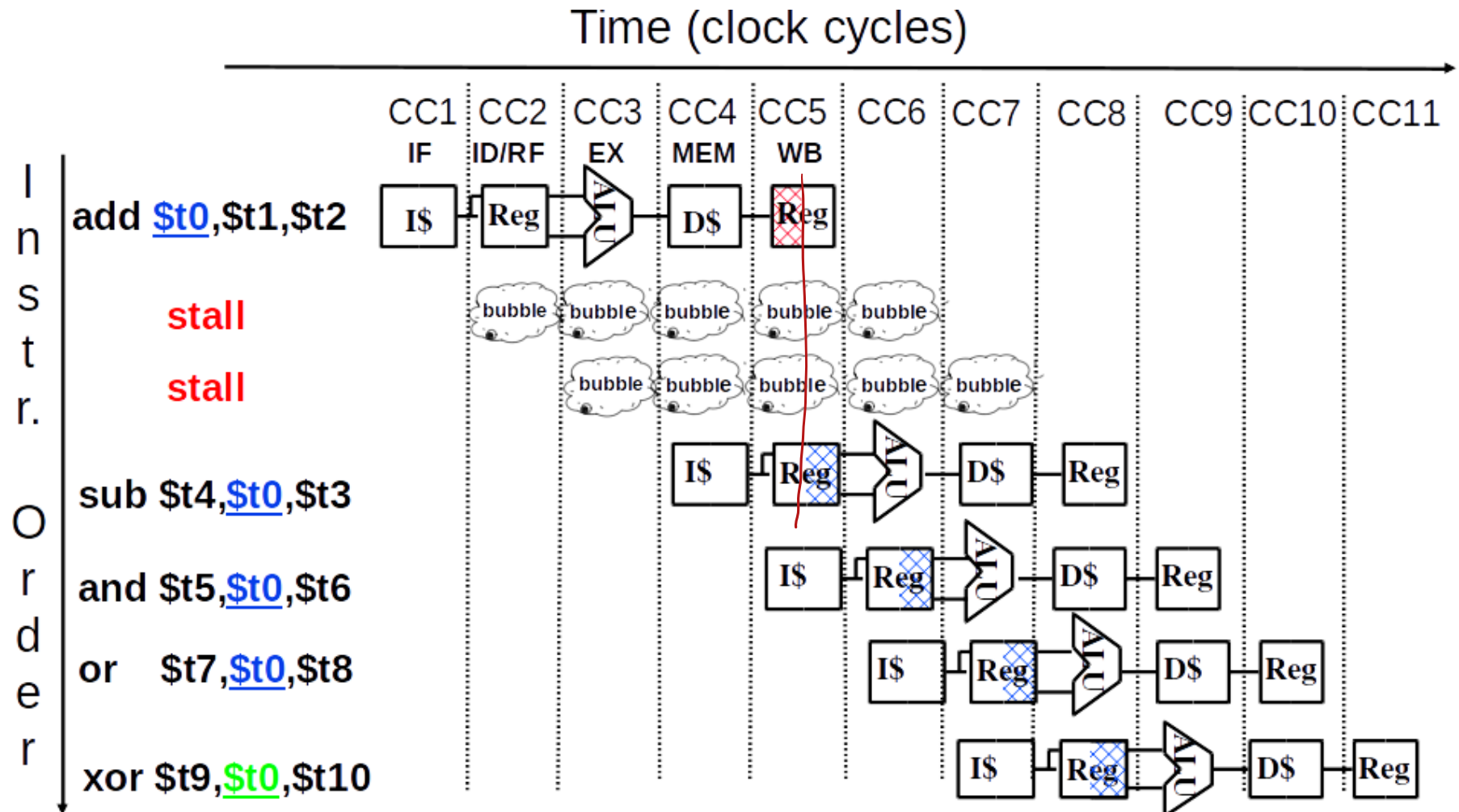
- Read After Write hazard.
- xor is far enough in the future to be okay.
- sub, and need more work.
- or actually already resolved via structural hazard solution



Data Hazard Example 1 (2/3)

Possible (but not great) solution: **stall** the execution.

- sub structural hazard already solved.



Data Hazard Example 1 (3/3)

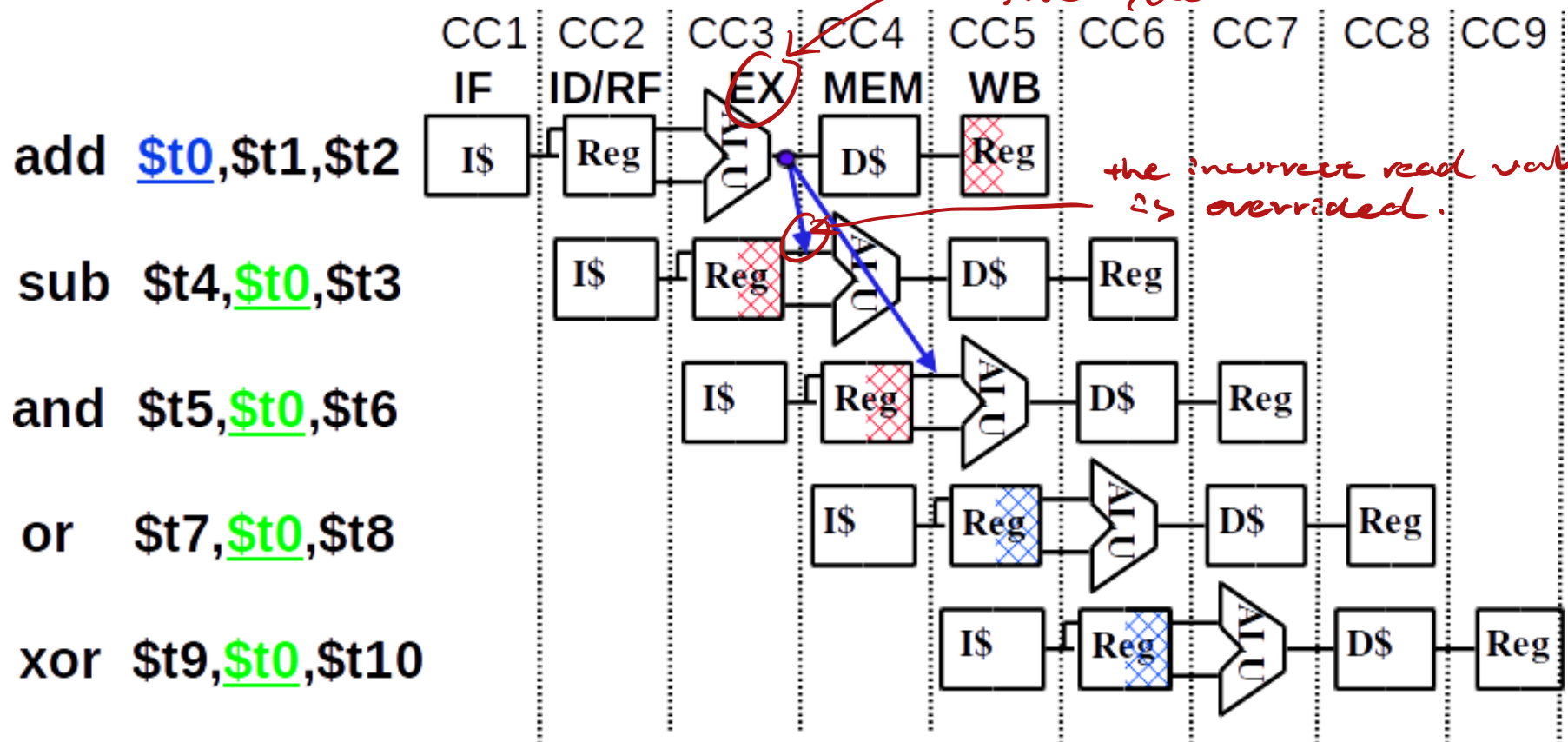
Another possible solution: **forwarding**.

- Implemented via inter-stage registers.
- No more stalls!
- ALU forwarding for add to sub and add to and.
- or structural hazard already solved.

*to implement a forwarding,
we need a feedback loop and
a mux to determine the forwarding*

*the value of t_0 is known after
3rd cycle*

*the incorrect read value
is overridden.*

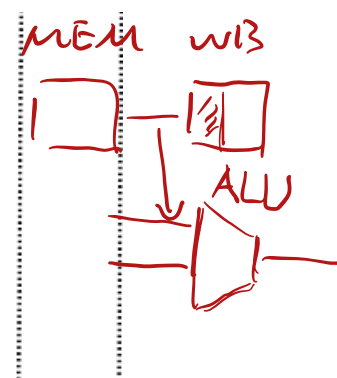
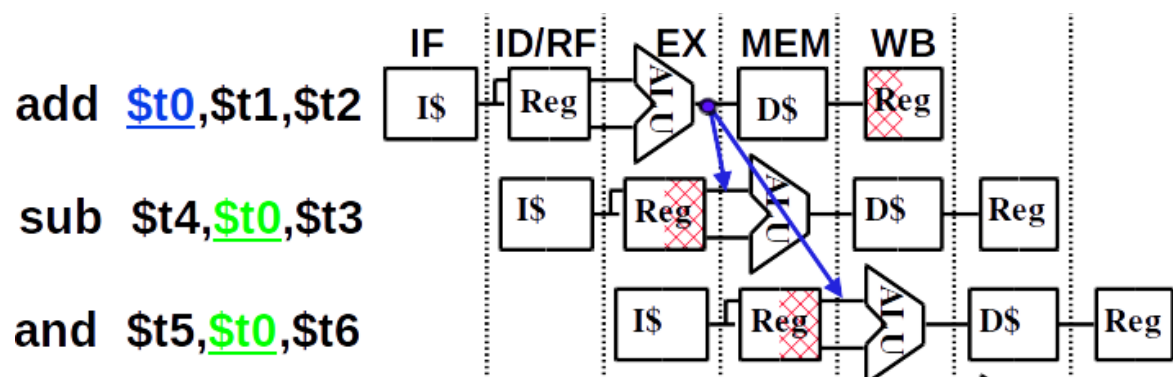


More ALU-ALU Forwarding

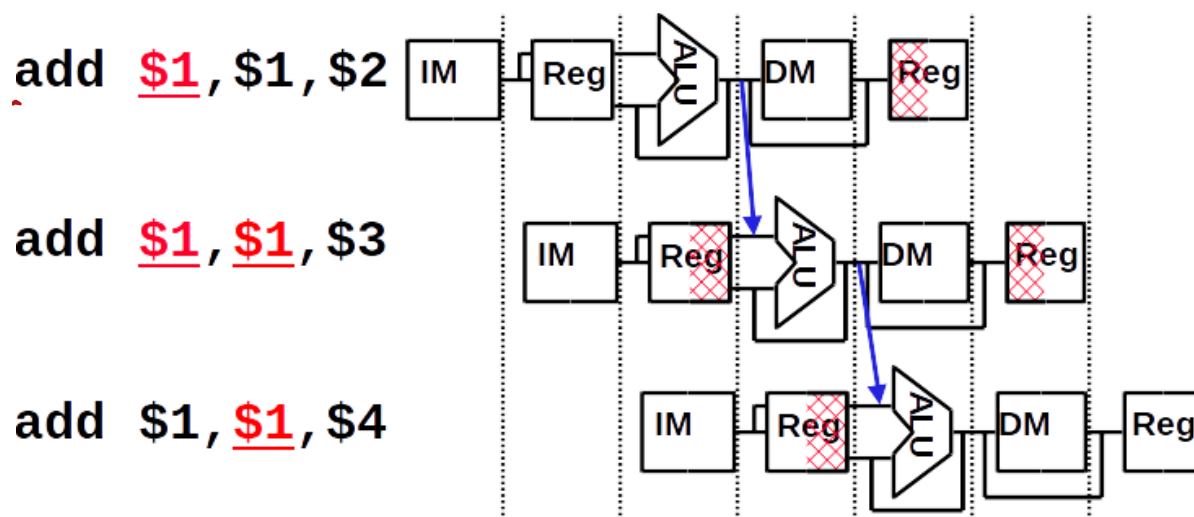
Two kinds of forwarding *into* the ALU:

- From instr. in MEM stage (from ALU/MEM reg): ALU-ALU forwd.
- From instr. in WB stage (from MEM/WB ref): MEM-ALU forwd.
- Which forwarded value to choose? \implies More control, more MUX.

forwarded from buffer to input

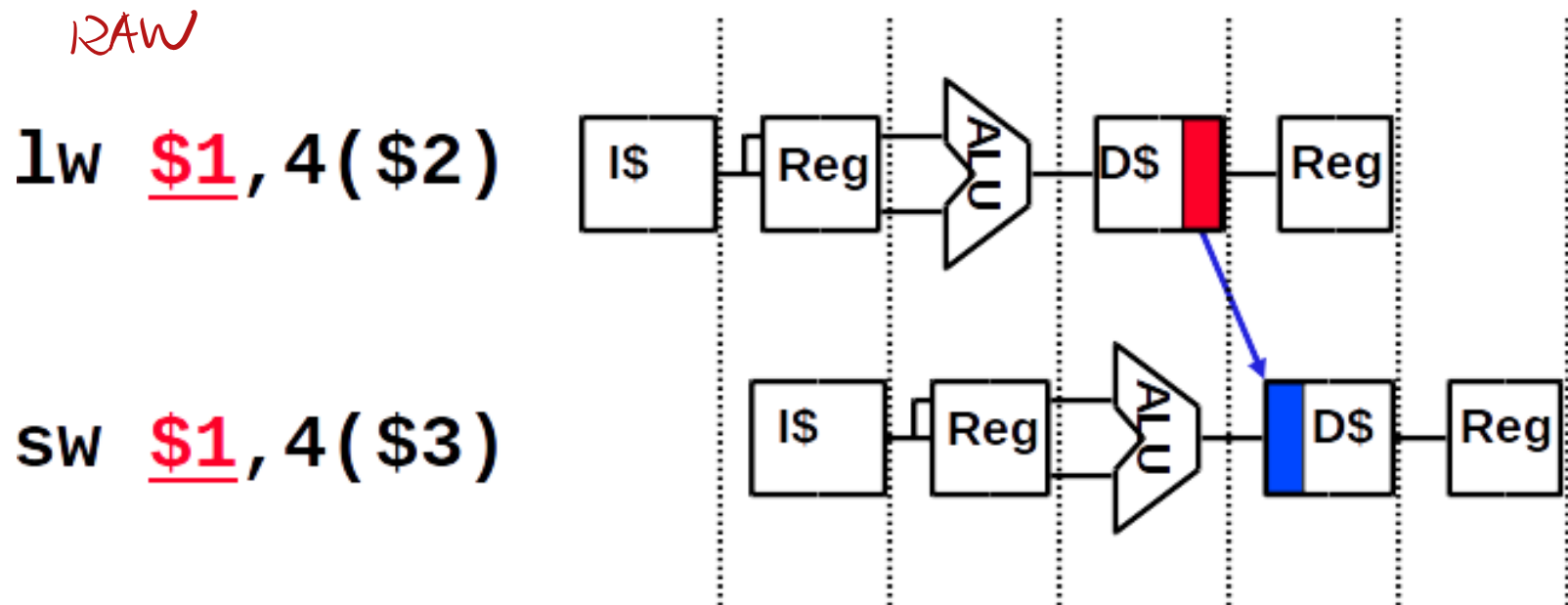


chain forwarding



MEM-MEM Forwarding

- For efficient memory copies (a common operation) this optimization results in no stalls.
 - ↳ Otherwise, two stalls required.
 - ↳ Eight great ideas in computer arch.: make the common case fast.
- MEM-MEM also used for forwarding an ALU result into a sw
 - ↳ `add $t0, $t1, $t4` and `sw $t0 0($t2)`.
 - ↳ `$t0` is MEM-MEM forwarded.

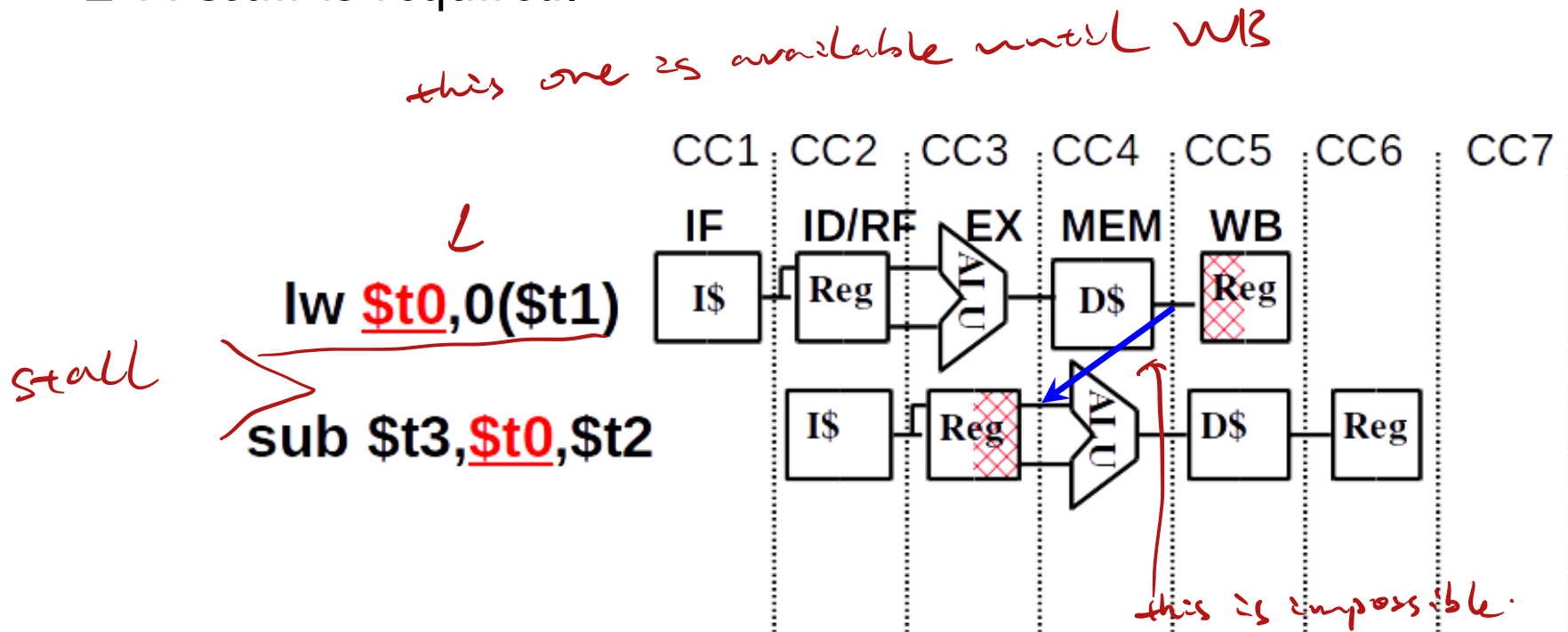


Load-Use Data Hazard

this one cannot be solved with forwarding.

Load-use data hazard, a special kind of RAW hazard.

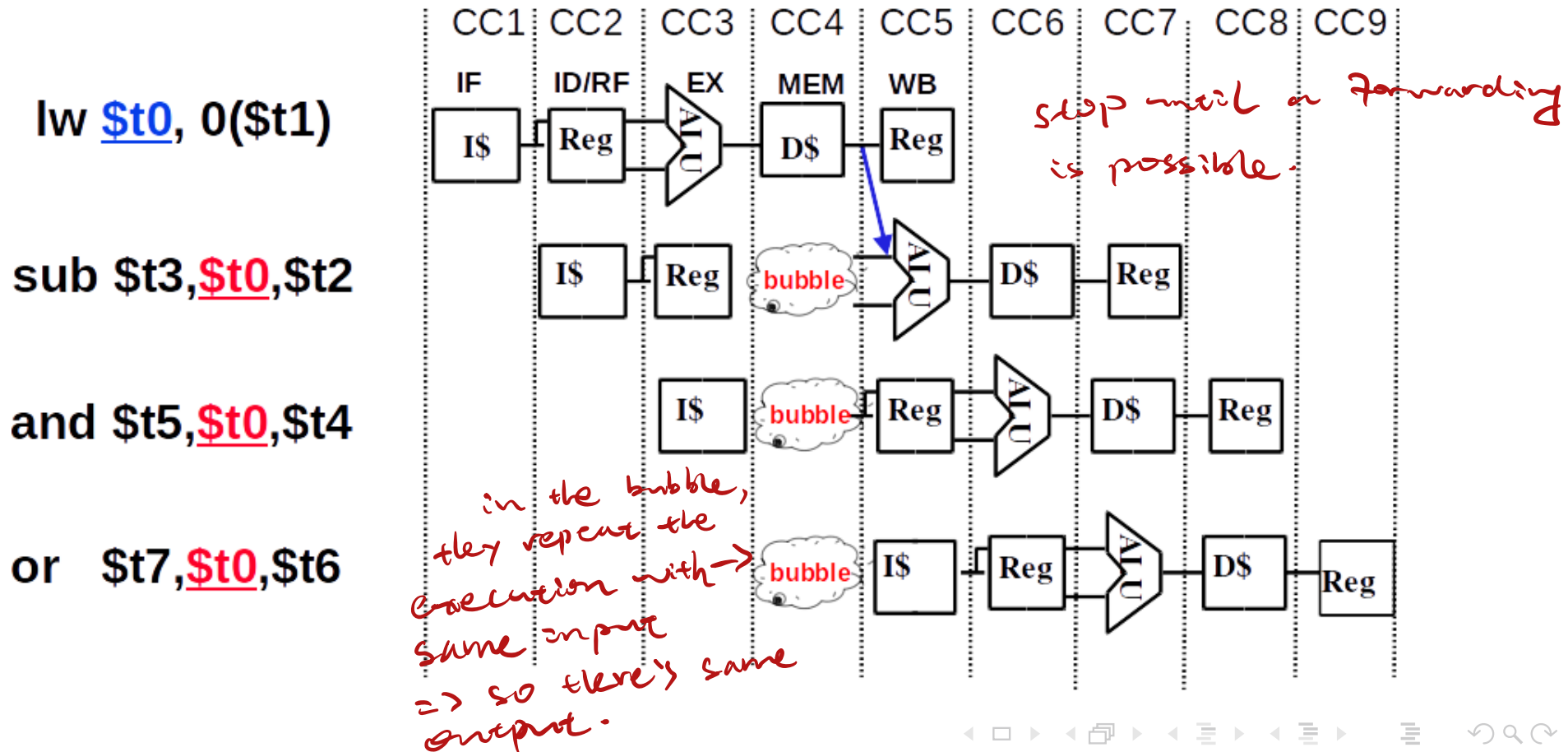
- Forwarding does not help here, still going backwards in time.
- A **stall** is required.



Implementing a Stall: Pipeline Interlock

Pipeline Interlock—hardware detects hazard and stalls the pipeline.

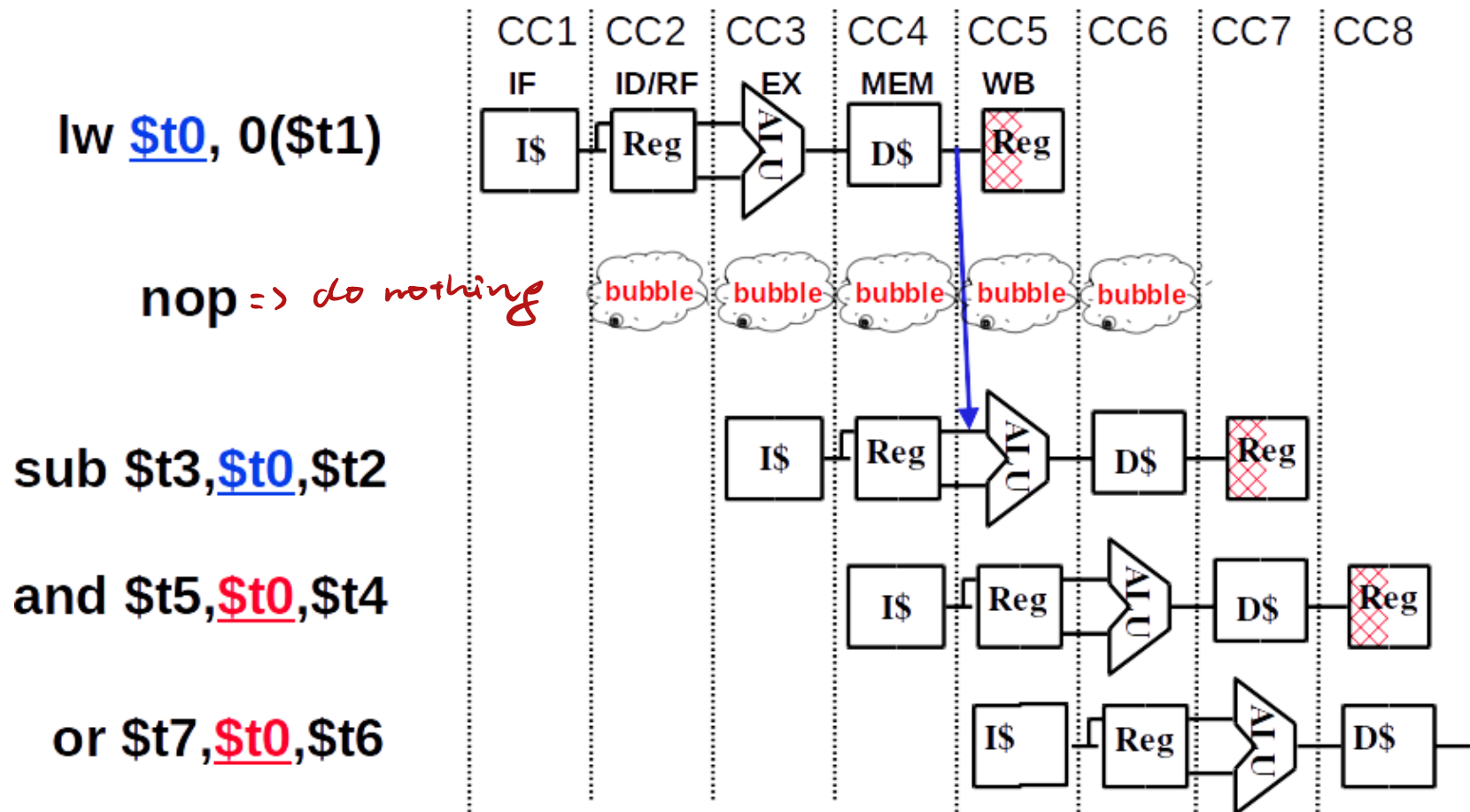
- Quite literally locks the flow of data between stages (locking writes to inter-stage registers).
- Essentially inserts an air bubble into pipeline.



Implementing a Stall: NOP

NOP—a “no operation” special instruction inserted into instruction flow by compiler (or assembler, or assembly programmer).

- Hazards are detected and fixed at compile-time.
- Can be combined with forwarding; MEM-ALU in this case.



Pipeline Interlock vs NOP

- Interlocking requires special circuitry to dynamically detect hazards and stall the datapath. *complex*
- nop requires extra effort at compile time to detect and resolve hazards.
- Inserted nop instructions bloat instruction memory. *this one is actually exist* *but nop keep the datapath simple.*
- More work at compile time for nop insertion but simpler (= faster?) datapath and controller.
- MIPS: Microprocessor without Interlocked Pipelined Stages

Data Hazards and Code Structure

*additional optimization
not necessarily need nops.*

Some data hazards are “fake”.

- Only caused by the order of instructions and not a true dependency.
- Re-order code (if possible) so an independent instruction performed instead of a nop.
 - ↳ Where the nop would be inserted is called the **load delay slot**.
 - ↳ Load delay slot can be filled with a nop or an independent instruction.
- Need at least one instruction between lw and using the loaded word.

<i>these are load-use hazard.</i>	stall	lw	\$t1, 0(\$t0)	<i>instead of inserting NOP, we reorder the instruction to avoid the RAW dependency.</i>	lw	\$t1, 0(\$t0)	
		lw	\$t2 , 4(\$t0)		lw	\$t2 , 4(\$t0)	
		add	\$t3, \$t1, \$t2			lw	\$t4 , 8(\$t0)
		sw	\$t3, 12(\$t0)			add	\$t3, \$t1, \$t2
		lw	\$t4 , 8(\$t0)			sw	\$t3, 12(\$t0)
	stall	add	\$t5, \$t1, \$t4			add	\$t5, \$t1, \$t4
		sw	\$t5, 16(\$t0)			sw	\$t5, 16(\$t0)
13 cycles					11 cycles		

Outline

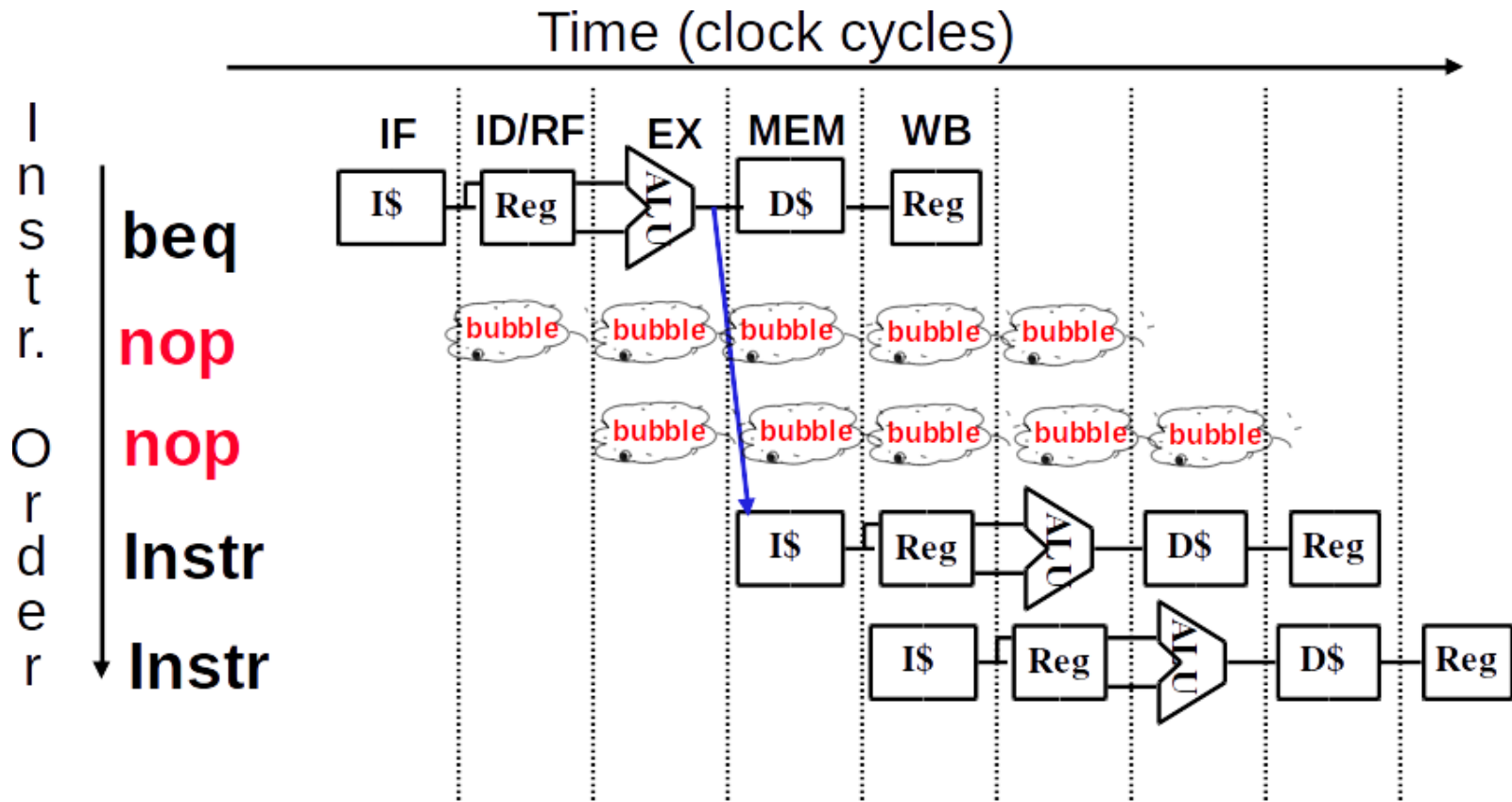
- 1 Overview
- 2 Structural Hazards
- 3 Data Hazards
- 4 Control Hazards**

Control Hazards: Causes and Resolutions

- Control hazards are caused by instructions which change the flow of control.
 - ↳ Branching.
 - ↳ If statements, loops.
- Sometimes called branch hazards.
- Since branch condition (beq, bne) not determined until after EX stage, cannot be *certain* about next instruction to fetch.

Control Hazard Resolution: Wait

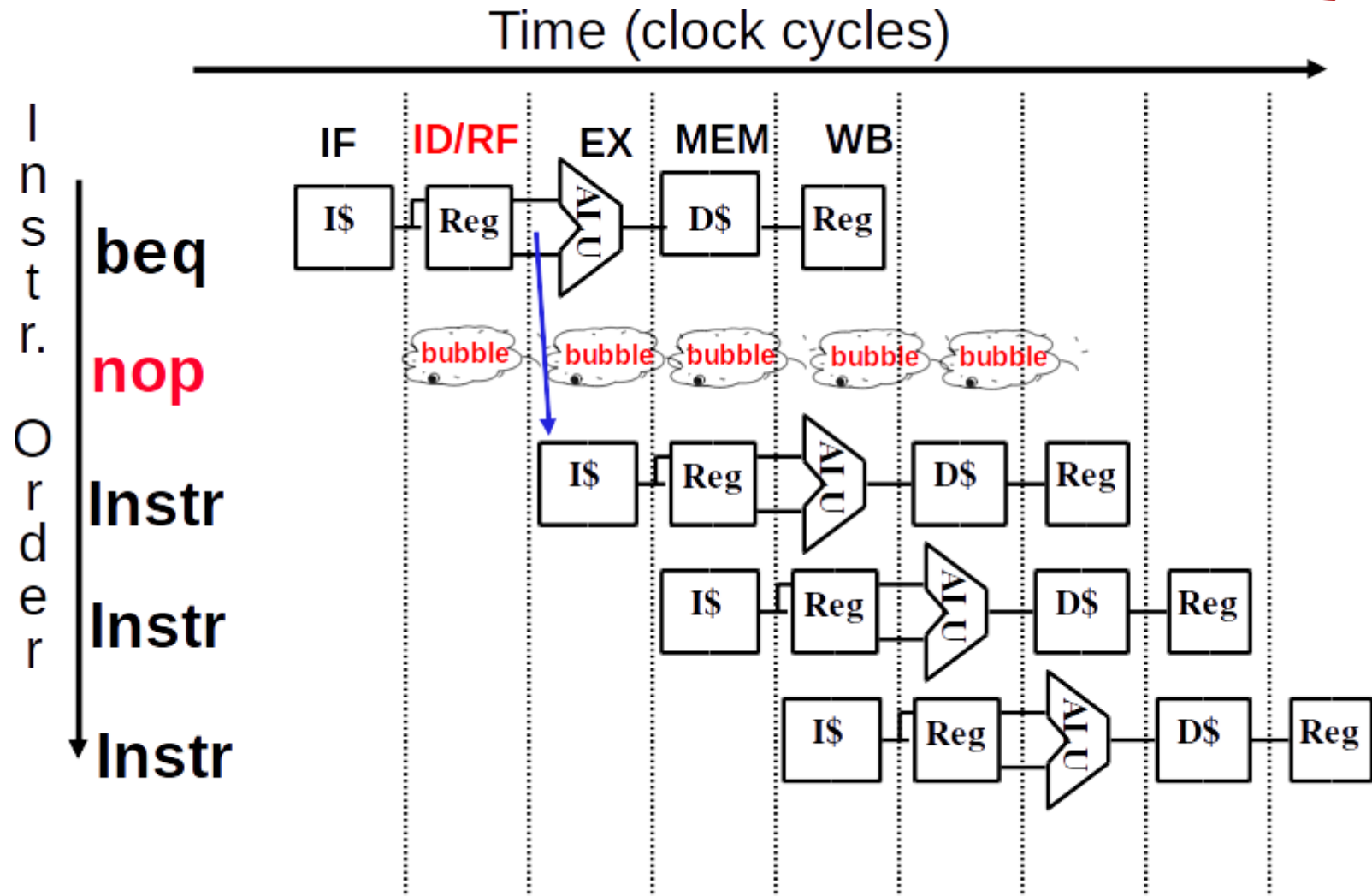
The simplest resolution is to just wait until branch condition is calculated before fetching next instruction.



Control Hazard Resolution: Add a Branch Comparator

Add a special circuit used to calculate branch conditions.

- Now only one stall needed instead of two.
- Similar to load-use hazard we now have a branch delay slot.



Delayed Branching

- The **branch delay slot** is the instruction immediately following a branch. Can be a nop or a useful instruction.
- In **delayed branching** the instruction in the branch delay slot is **always** executed whether or not the branch condition holds.
 - ↳ Used in conjunction with a special branch comparator.
 - ↳ Filling the branch delay slot (and other code re-organization) is usually handled by compiler/assembler.
 - ↳ Cannot fill slot with an instruction that influences branch condition.
- Jump instructions also have a delay slot.

addi \$v0, \$0, 1

+ these are required in branching.
add \$t0, \$s0, \$s1

add \$t1, \$s2, \$s3

beq \$t0, \$t1, L

⋮

L: ...

add \$t0, \$s0, \$s1

add \$t1, \$s2, \$s3

beq \$t0, \$t1, L

addi \$v0, \$0, 1

⋮

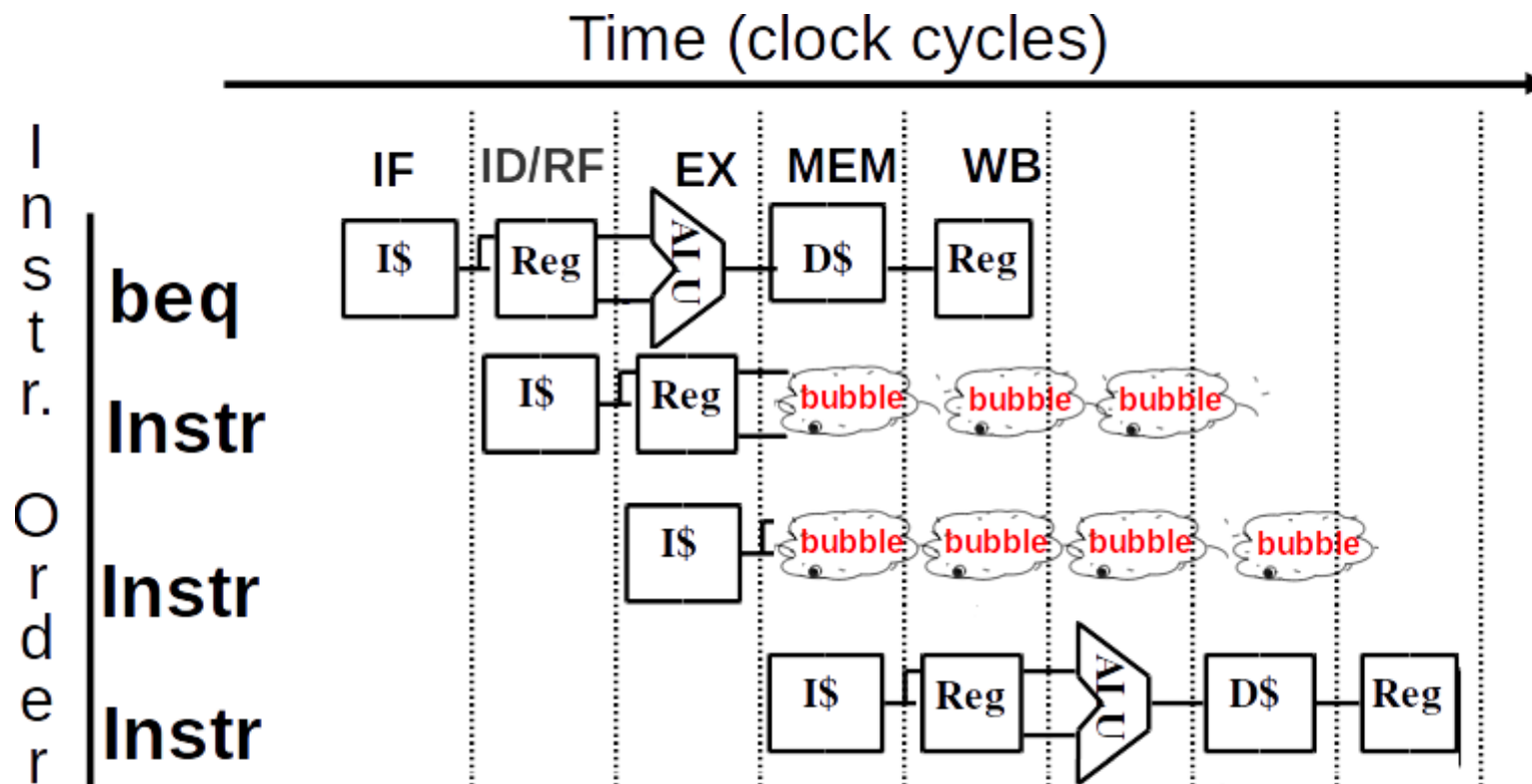
L: ...

addi executed regardless

Control Hazard Resolution: Branch Prediction

Hardware predicts whether branch will occur or not.

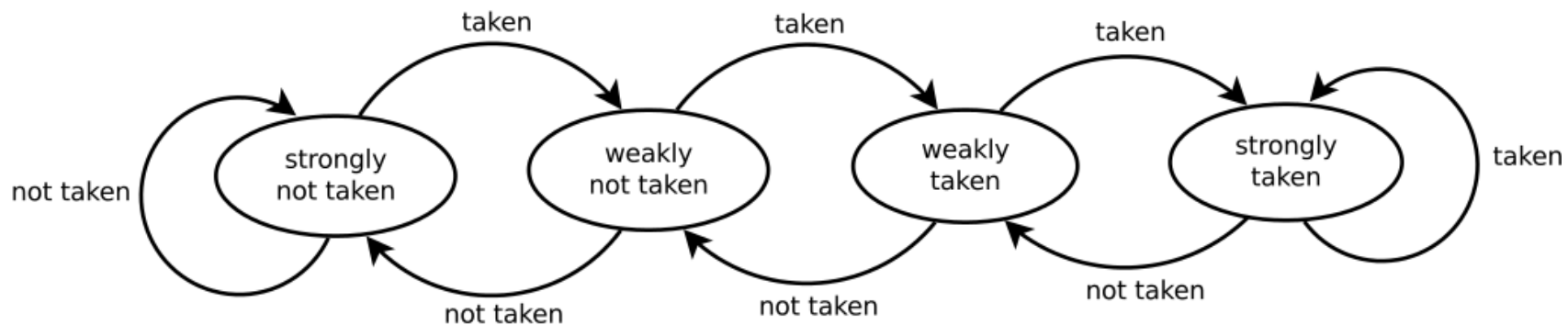
- If the branch condition ends up being opposite of prediction flush the pipeline.
- This flush shows a pipeline *without* a special branch comparator in ID stage. Otherwise, only one instruction needs to be flushed.



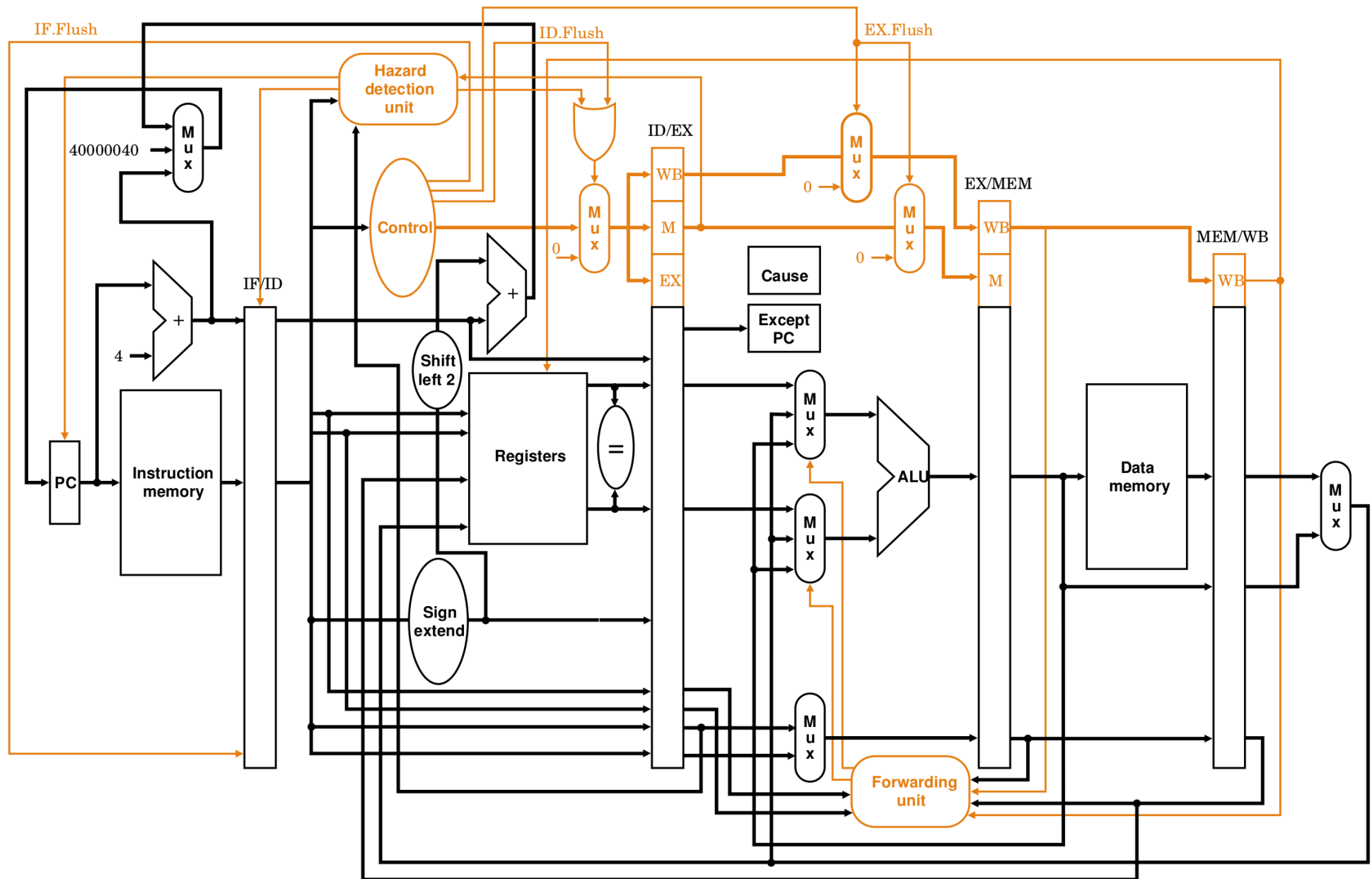
Implementing Branch Prediction

- Branches have exactly two possibilities: taken or not taken.
- In MIPS branches are *statically predicted* to never happen.
- **Dynamic branch prediction** uses run-time information to change prediction between taken or not taken.
 - ↳ Use **branch history** to predict future branches.
 - ↳ Simplest method is to use a *saturated counter*: increment counter if branch actually taken, decrease counter if branch not taken.
 - ↳ Predict based on current count.
 - ↳ More advanced predictors evaluate *patterns* in branch history.
- **Random branch prediction**: statistically 50% correct prediction.

A two-bit saturated counter:



Datapath With Forwarding and Flushing



Hazard Summary

- **Structural hazards** caused by conflicts accessing hardware.
 - ↳ Register access fast enough to happen twice in one clock cycle.
 - ↳ Banked L1 cache for simultaneous instruction and data access.
- **Data hazards** caused by Read After Write (RAW).
 - ↳ ALU-ALU forwarding.
 - ↳ MEM-MEM forwarding (memory copies).
 - ↳ Load-use hazard: stall (load-delay slot) and MEM-ALU forward.
- **Control hazards** caused by branch instructions.
 - ↳ Special branch comparator in ID stage.
 - ↳ Branch delay slot; **delayed branching**.
 - ↳ Branch prediction and **pipeline flush**.
- Compiler handles **nop** insertion to fix hazards.
- Hardware handles fixing hazards with **pipeline interlock**.

Datapath Optimizations Summary

Many possible configurations of datapath.

- Single-cycle to Multi-cycle
- Multi-cycle to Pipelining
- Pipeline with or without banked cache for instr. and data memory
- Pipeline with and without forwarding.
- Pipeline with and without load-use slot.
- Pipeline with and without ID stage branch comparator.
- Pipeline with and without branch delay slot.
- Pipeline with and without branch prediction.

During exercises, read carefully the specification and assumptions of that datapath.