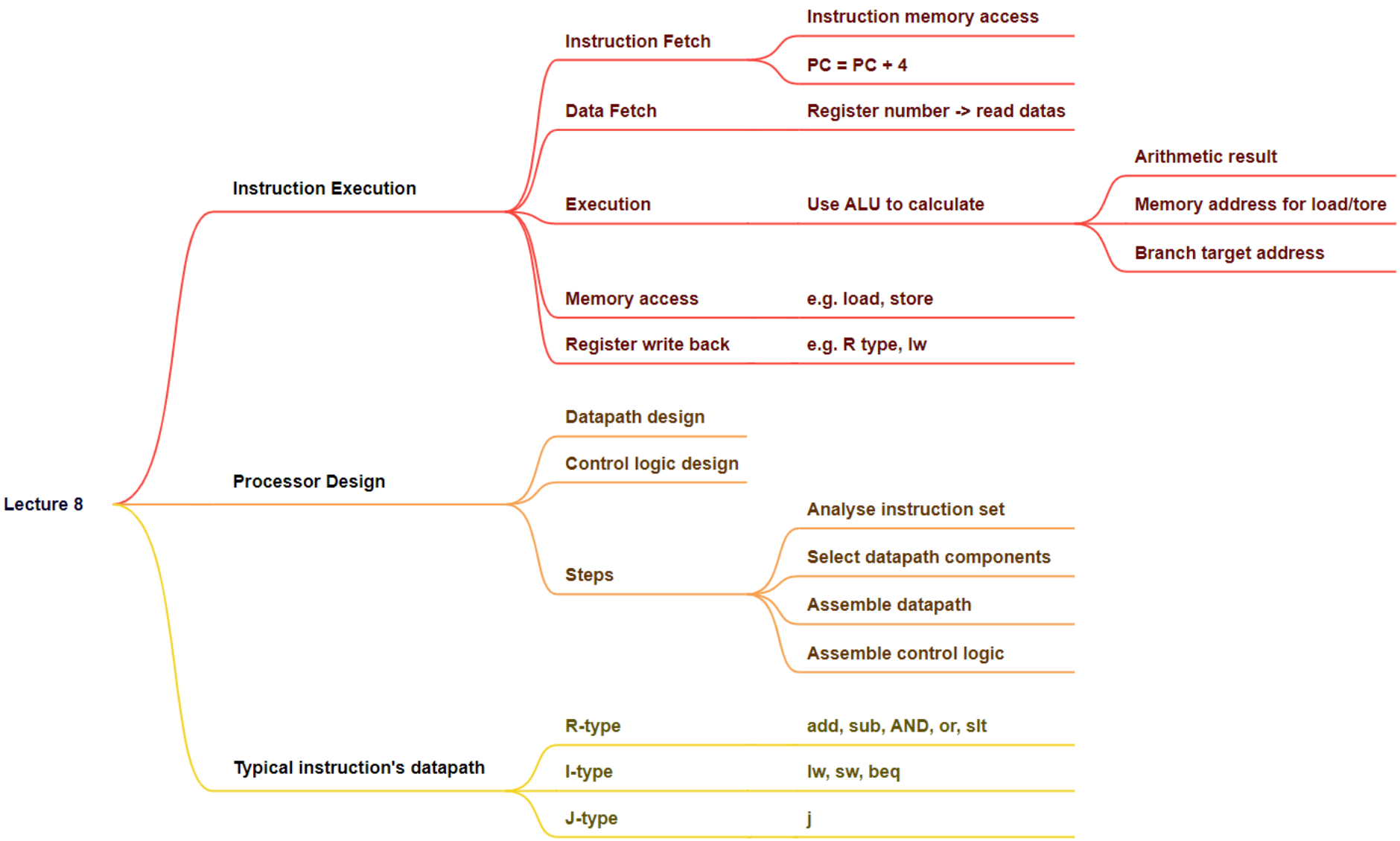
Lecture 9
Pipeline Overview

CS202 2023 Spring

Today's Agenda

- Recap
  - Implementation overview
  - Logic design basics
  - Single-Cycle Processor Datapath
- Context
  - Pipeline Performance
  - Introduction of Hazard
- Reading: Textbook 4.5
- Self-Study: Textbook 4.6-4.9

Recap

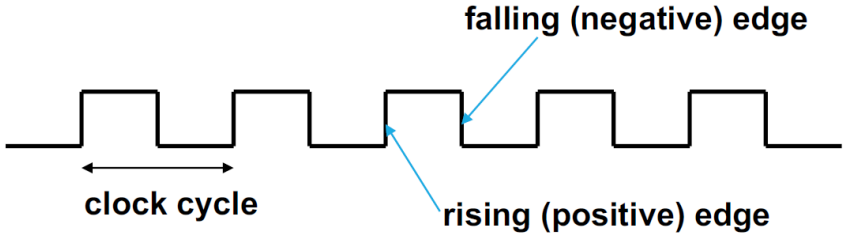

Lecture 8
Instruction Execution
Instruction Fetch
Instruction memory access
PC = PC + 4
Data Fetch
Register number -> read datas
Execution
Use ALU to calculate
Arithmetic result
Memory address for load/tore
Branch target address
Memory access
e.g. load, store
Register write back
e.g. R type, lw
Processor Design
Datapath design
Control logic design
Steps
Analyse instruction set
Select datapath components
Assemble datapath
Assemble control logic
Typical instruction's datapath
R-type
add, sub, AND, or, slt
I-type
lw, sw, beq
J-type
j

Outline

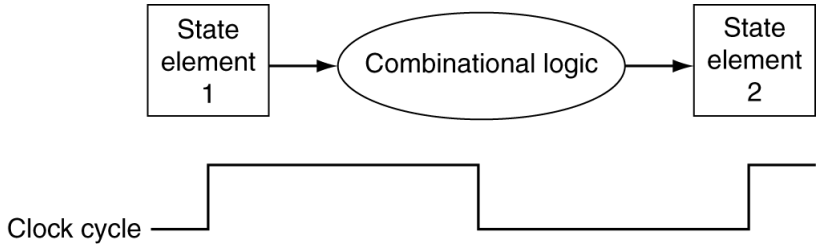
- **Pipeline Motivation**
- Pipeline Hazards

Clocking Methodologies

- Clocking methodology defines when signals can be read and when they can be written



- An edge-triggered methodology
  - all signals must propagate
    - from state element 1
    - through the combinational logic
    - and to state element 2 in the time of one clock cycle



Instruction Critical Paths

- Calculate cycle time assuming:
  - 100ps for register read or write
  - 200ps for other stages

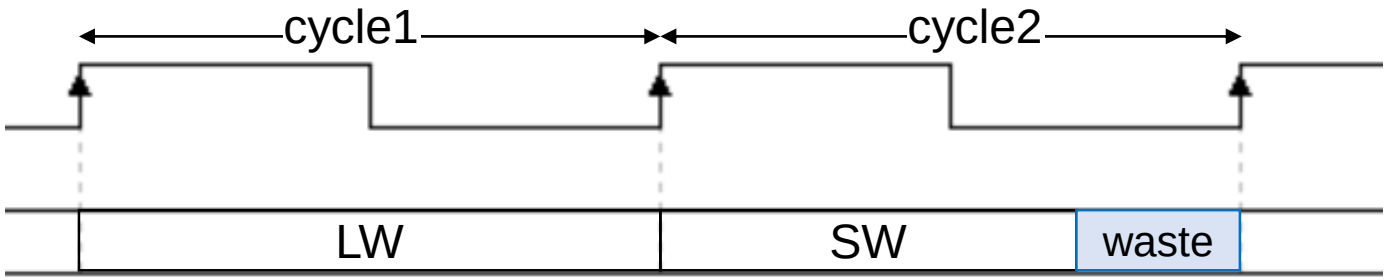
Critical Path

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file

Single Cycle issue

- Pros:
  - It is simple and easy to understand
- Cons:
  - Uses the clock cycle inefficiently
    - The clock cycle must be timed to accommodate the **slowest** inst.



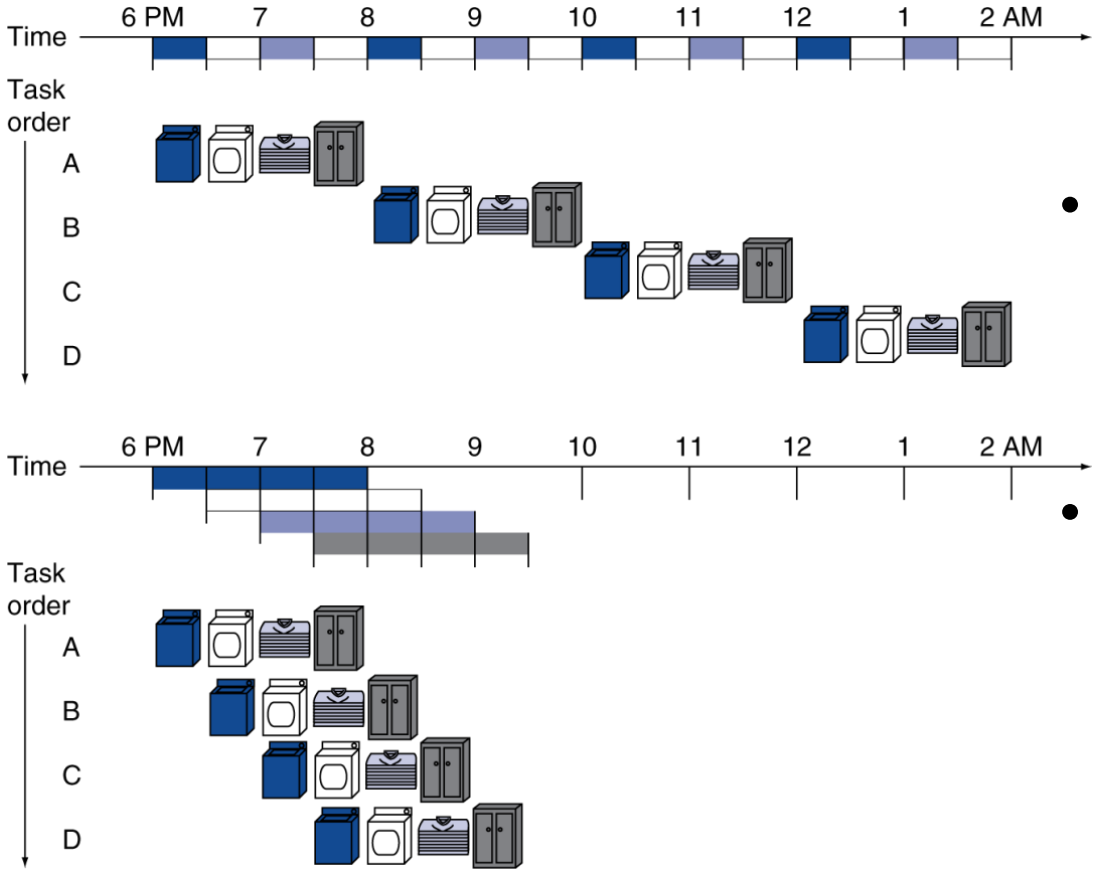
  - May be wasteful of area
  - Violates design principle
    - Making the common case fast

How Can We Make It Faster

- 1st method: Start fetching and executing the next instruction before the current one has completed
  - **Pipelining** – modern processors are pipelined for performance
  - Must know
    - Pipelining improves instruction throughput, but not the execution time of an individual instruction
    - Three different hazards need to be dealt with within a pipeline
- 2nd method: Fetch (and execute) more than one instruction at a time
  - next lecture

Pipelining Analogy

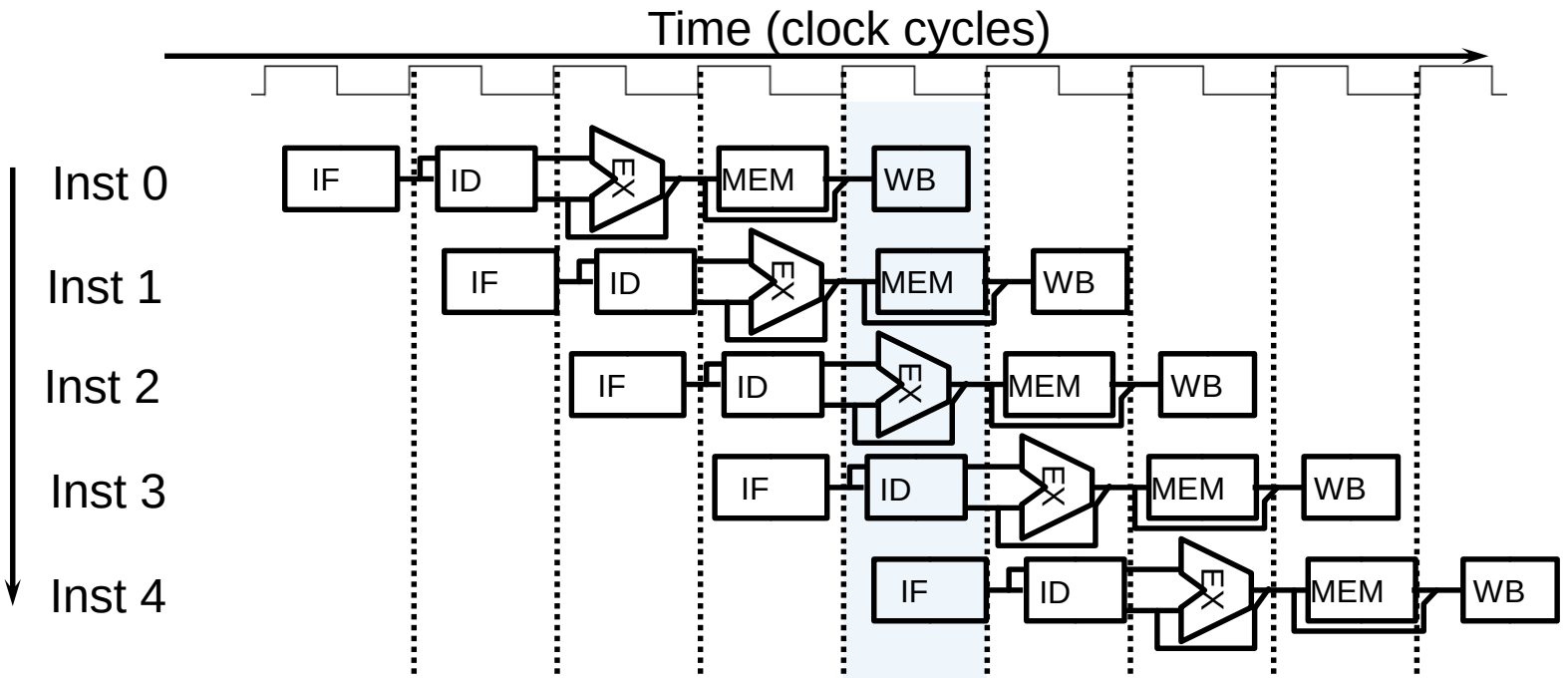
- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- 4 loads:
  - Speedup
    = 8/3.5 = 2.3
- n loads:
  - Speedup
    = 2n/(1.5 + 0.5n) ≈ 4
    = number of stages

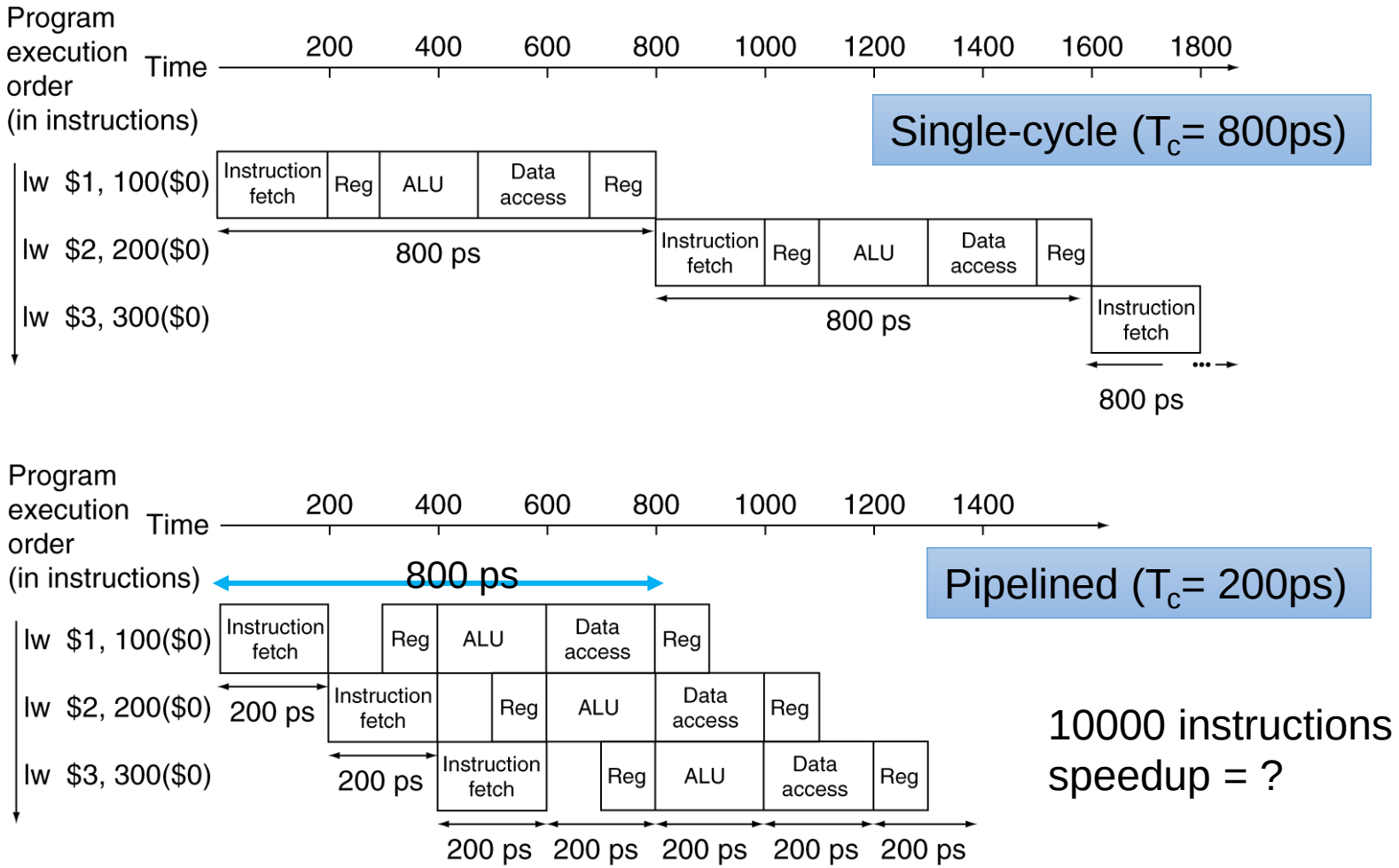
MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register



Pipeline Performance

- Single-cycle: each instruction takes one clock cycle
- Pipelined: each stage takes one clock cycle



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time

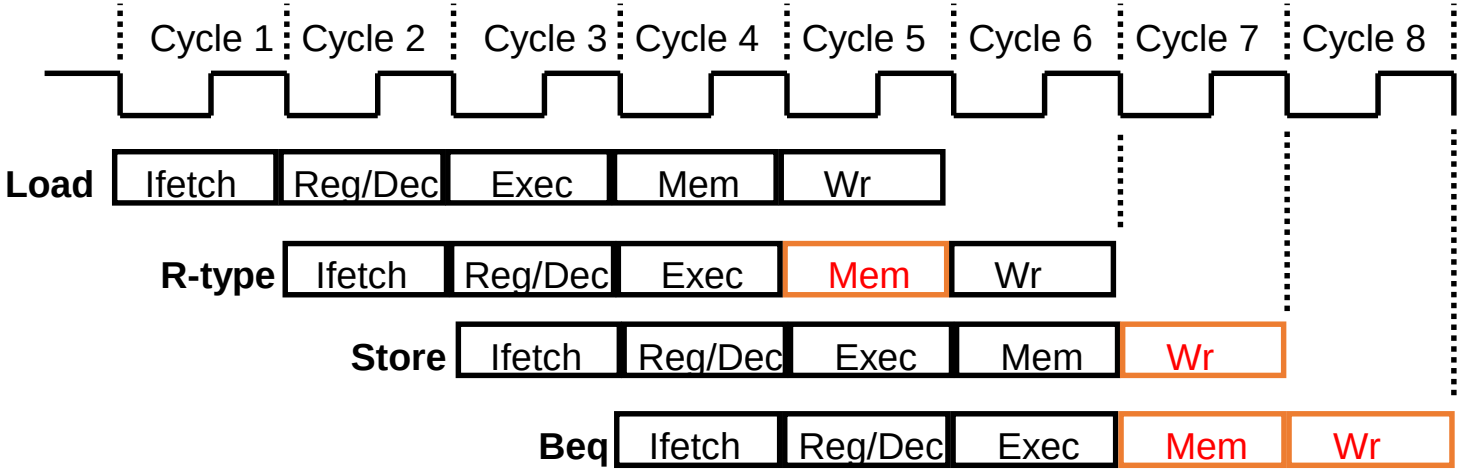
$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

- If not balanced, speedup is less
- Speedup due to **increased throughput**
  - Latency (time for each instruction) does not decrease

Unify instruction length

	IF	ID	EX	MEM	WB
R-type	Fetch Mem[PC] PC = PC+4	fetch Reg[rs],Reg[rt]	ALUOut = A op B	-	Reg[rd] = ALUout
lw		signExt(imm)	ALUout = A + signExt(imm)	Load Mem[ALUout]	Write back to Reg[rt]
sw				Store Mem[ALUout]=B	-
beq		PC=PC+4+signExt(imm)<<2	If (A==B) then Zero = 1	-	-

- R-type: 4 stages, Store: 4 stages, Beq: 3 stages, unbalanced
  - Add non-operation stages to unify instruction length



Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Outline

- Pipeline Motivation
- **Pipeline Hazards**

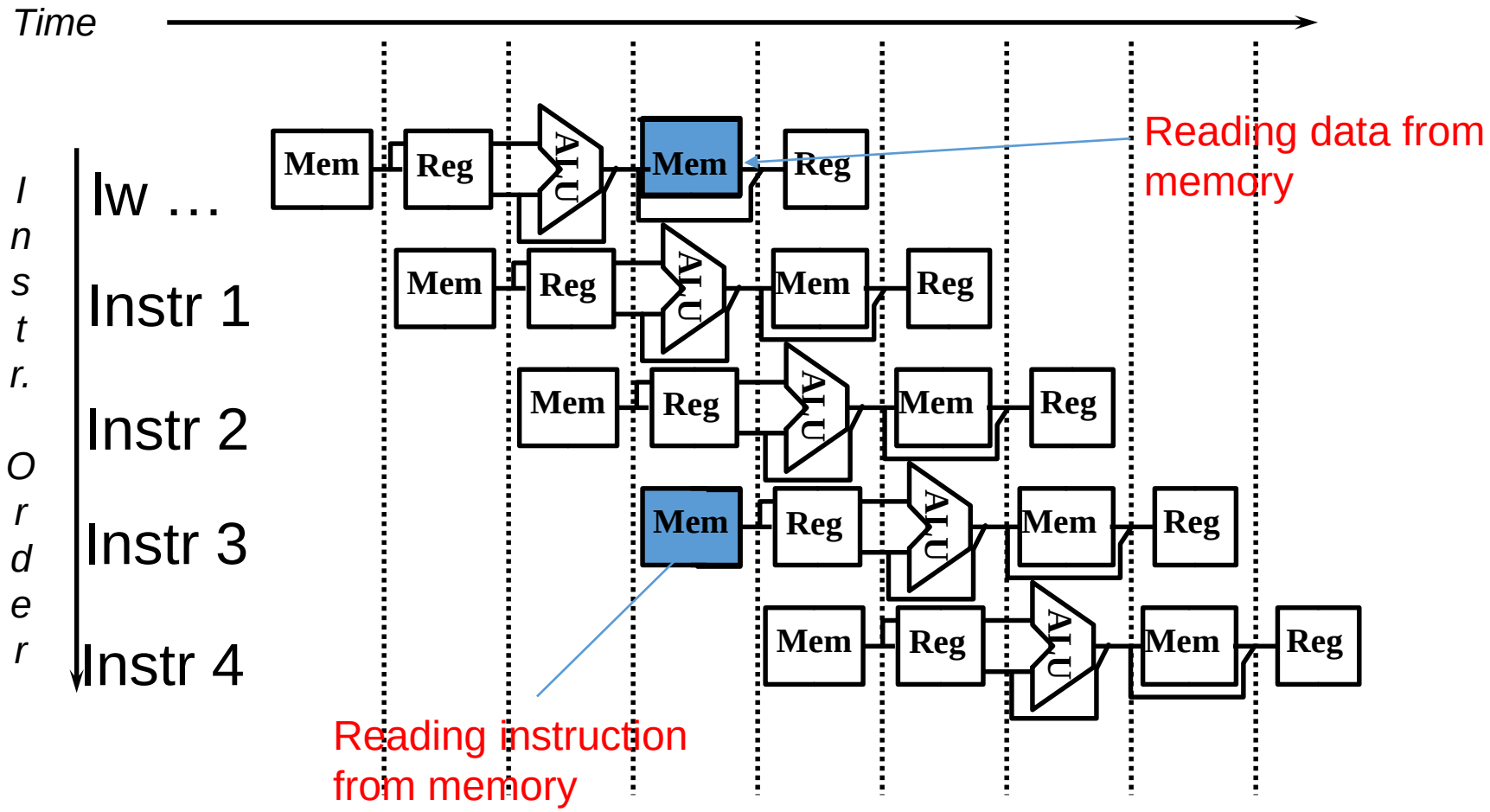
Hazards

- Situations that prevent starting the next instruction in the next cycle
  - Structure hazards
    - A required resource is busy
  - Data hazard
    - Need to wait for previous instruction to complete its data read/write
  - Control hazard
    - Deciding on control action depends on previous instruction
- Can usually resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

Structure Hazards

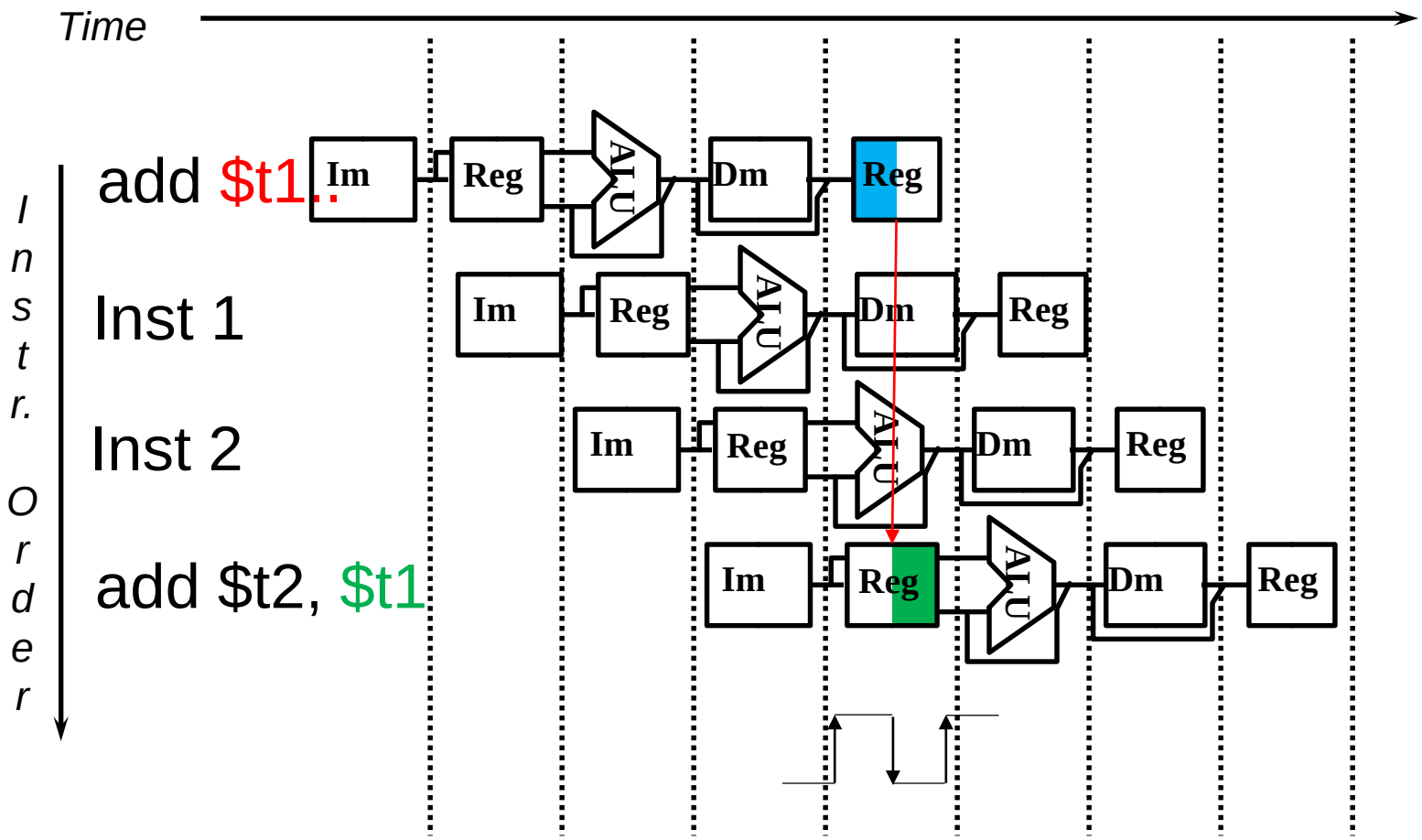
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to **stall** for that cycle
    - Would cause a pipeline "**bubble**"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
- MIPS instruction set was designed to be pipelined
  - making it fairly easy to avoid structural hazards

Resolve Structure Hazard



- Fix with separate inst. and data memories

Register access



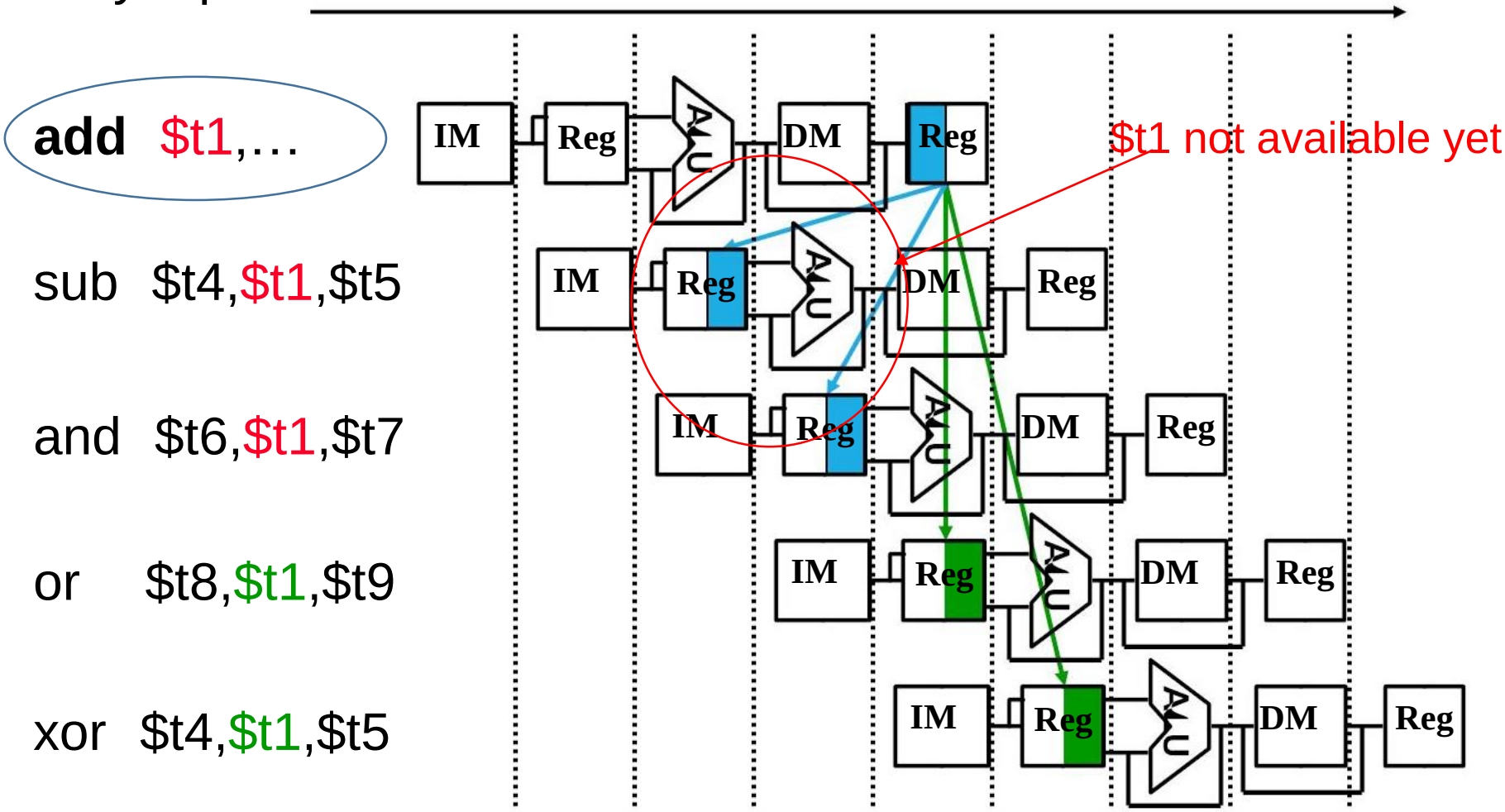
- No structural hazard since register file is accessed by doing reads in the second half of the cycle and writes in the first half

Data Hazards

- When data needed to execute the instruction is not yet available.
  - Register usage
  - Load-Use
- Solution
  - Stall
  - Forwarding 提前拿出来
  - Stall + Forwarding
  - Code scheduling

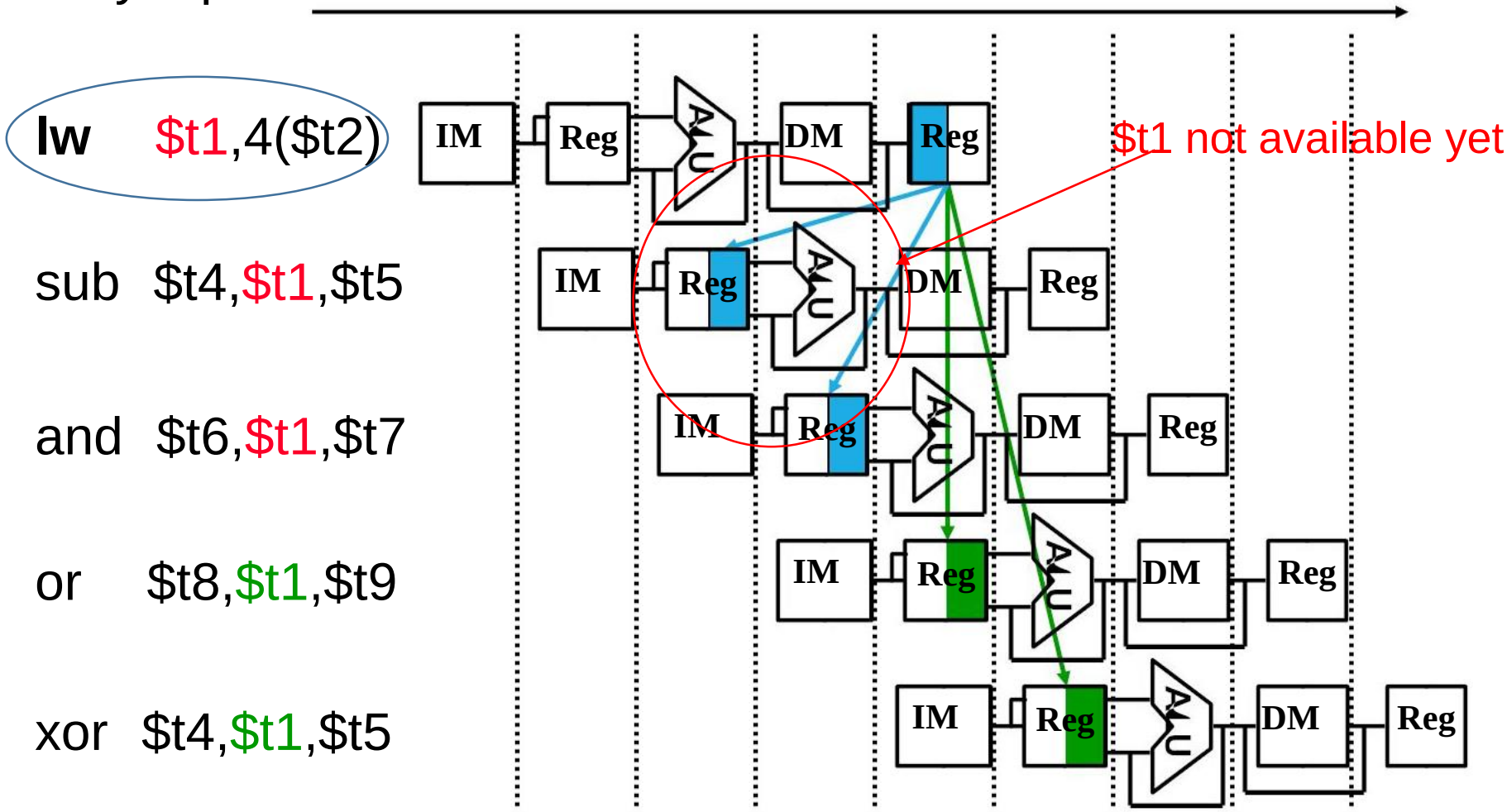
Data Hazards: Register Usage

- An instruction depends on completion of data access by a previous instruction



Data Hazards: Load-Use

- An instruction depends on completion of data access by a previous instruction

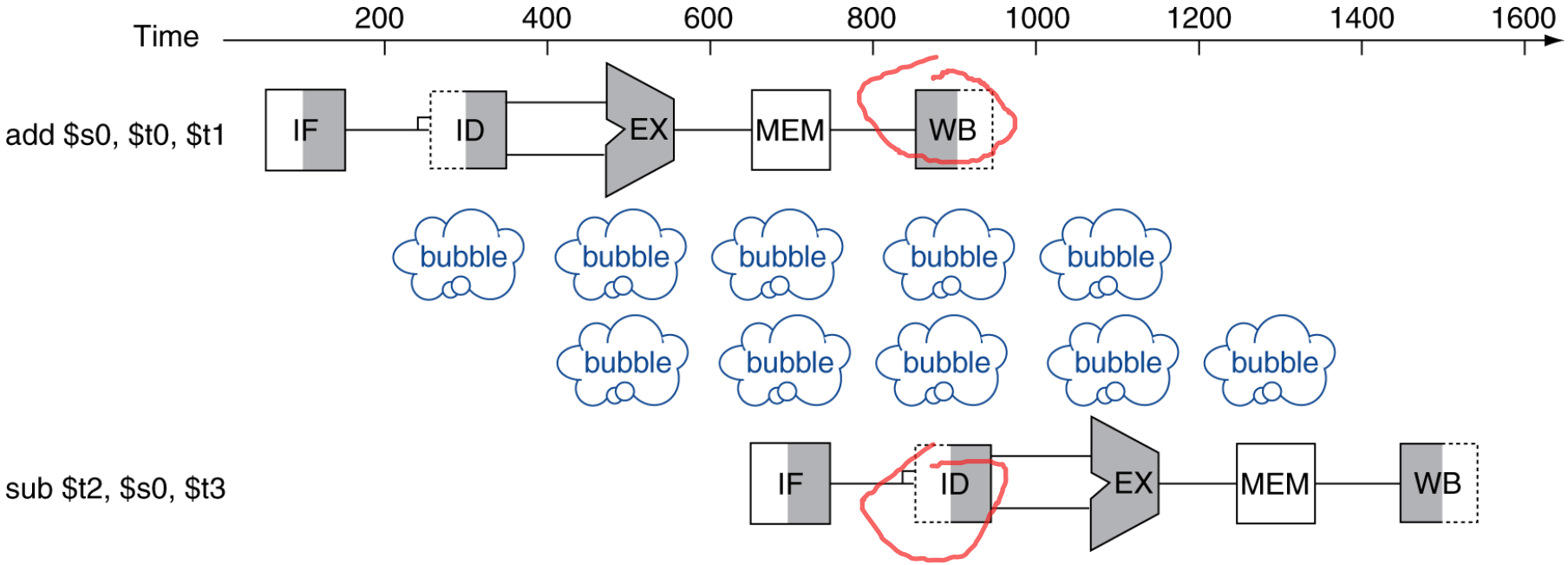


Resolve Data Hazards 1: Stall

- Can fix data hazard by waiting
  - Inserting NOP instructions (stall/bubble)
  - but impacts CPI

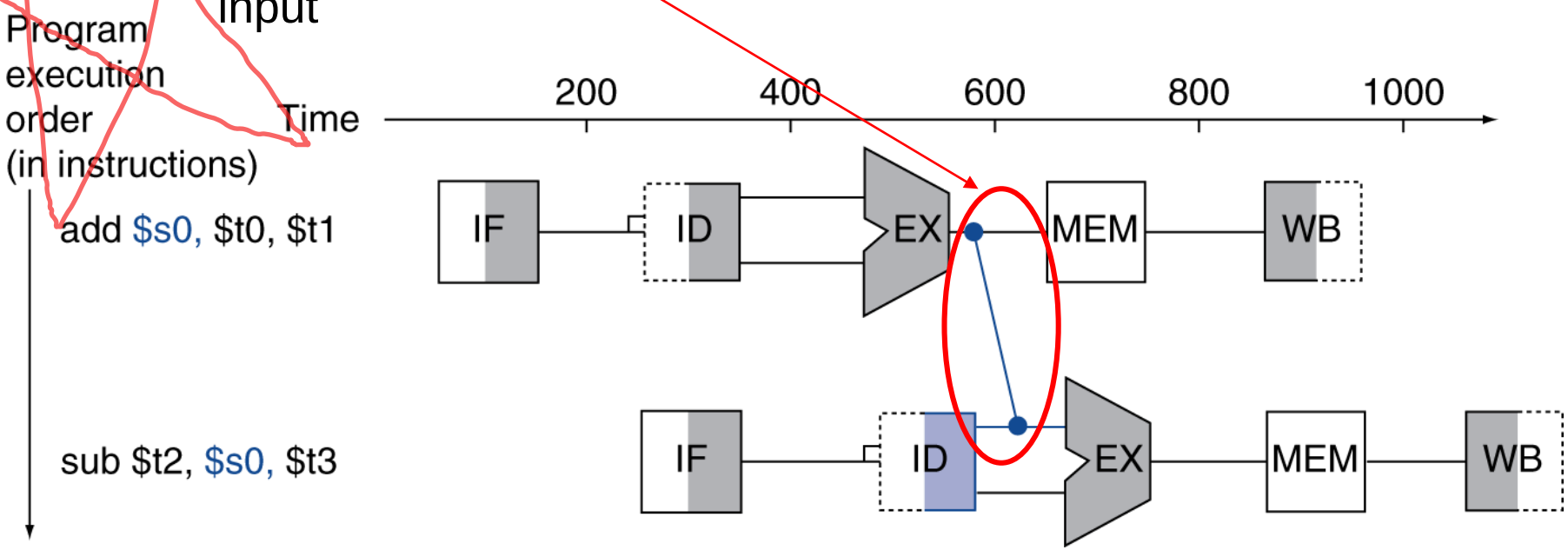
```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

```
add   $s0, $t0, $t1
NOP
NOP
sub   $t2, $s0, $t3
```



Resolve Data Hazards 2: Forwarding

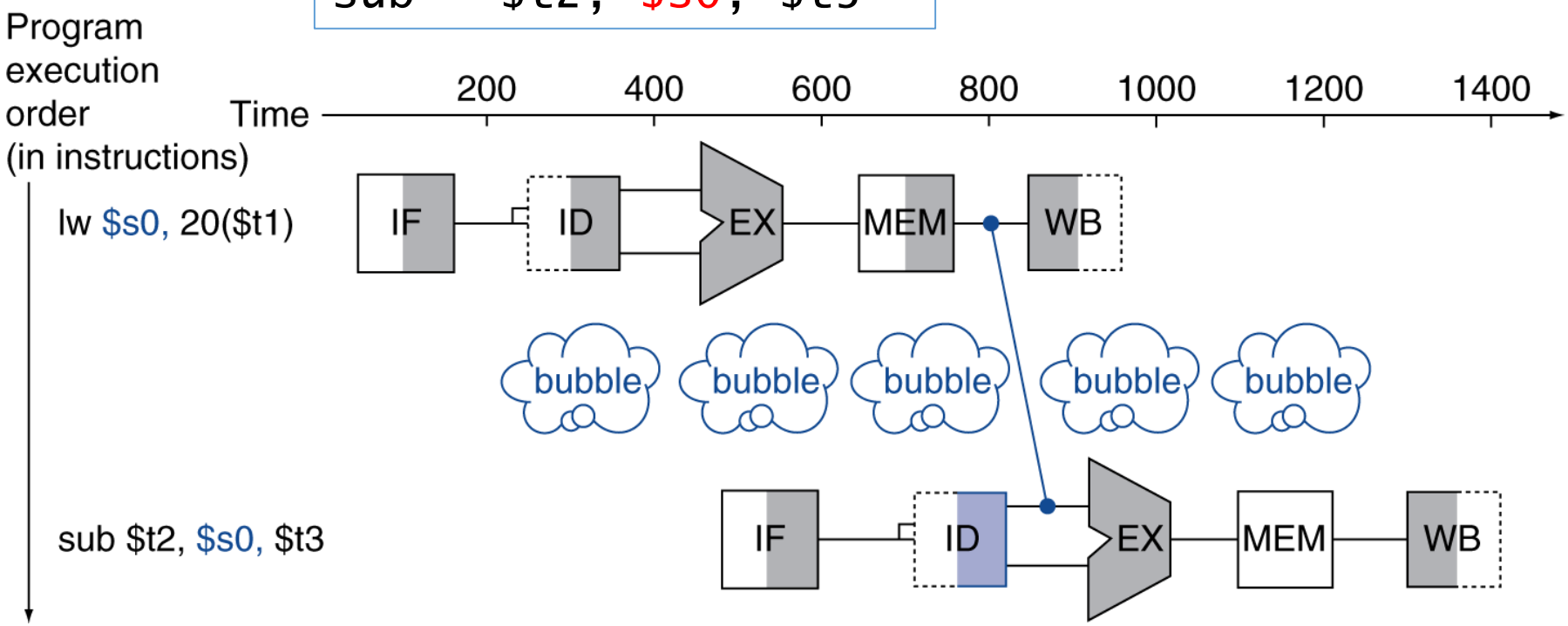
- Fix data hazards by forwarding (aka. bypassing) results as soon as they are available to where they are needed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath
    - e.g. Add a bypassing line to connect the output of EX to the input



Resolve Data Hazards 3: Stall + forwarding (Load-Use)

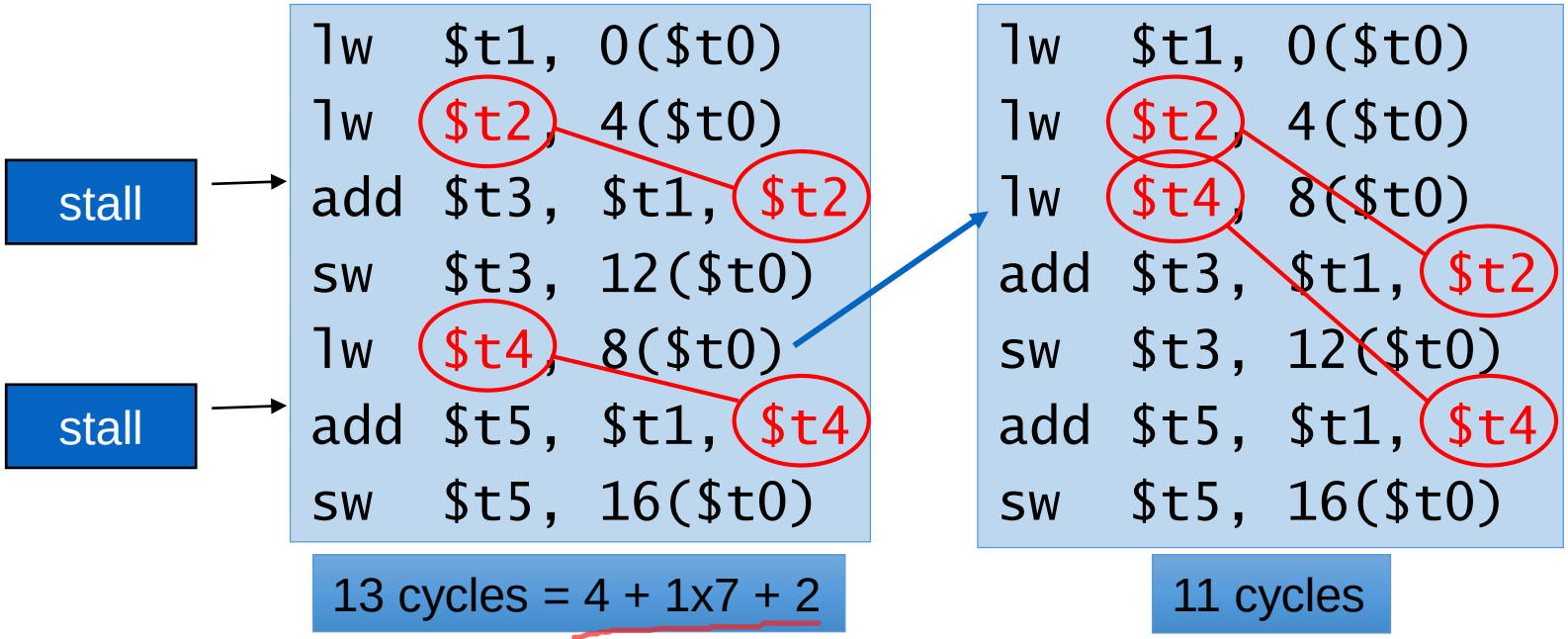
- Can't always avoid stalls by forwarding for Load-Use data hazard
  - If value not computed when needed, we will still need one stall cycle even with forwarding

```
lw    $s0, 20($t1)
sub   $t2, $s0, $t3
```



Resolve Data Hazards 4: Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction (avoid “Load+Ex” combination)
- C code for A = B + E; C = B + F;
- (t1:B, t2:E, t3:A, t4:F, t5:C)



Assume forwarding is available, we still need 2 stalls to resolve data hazard

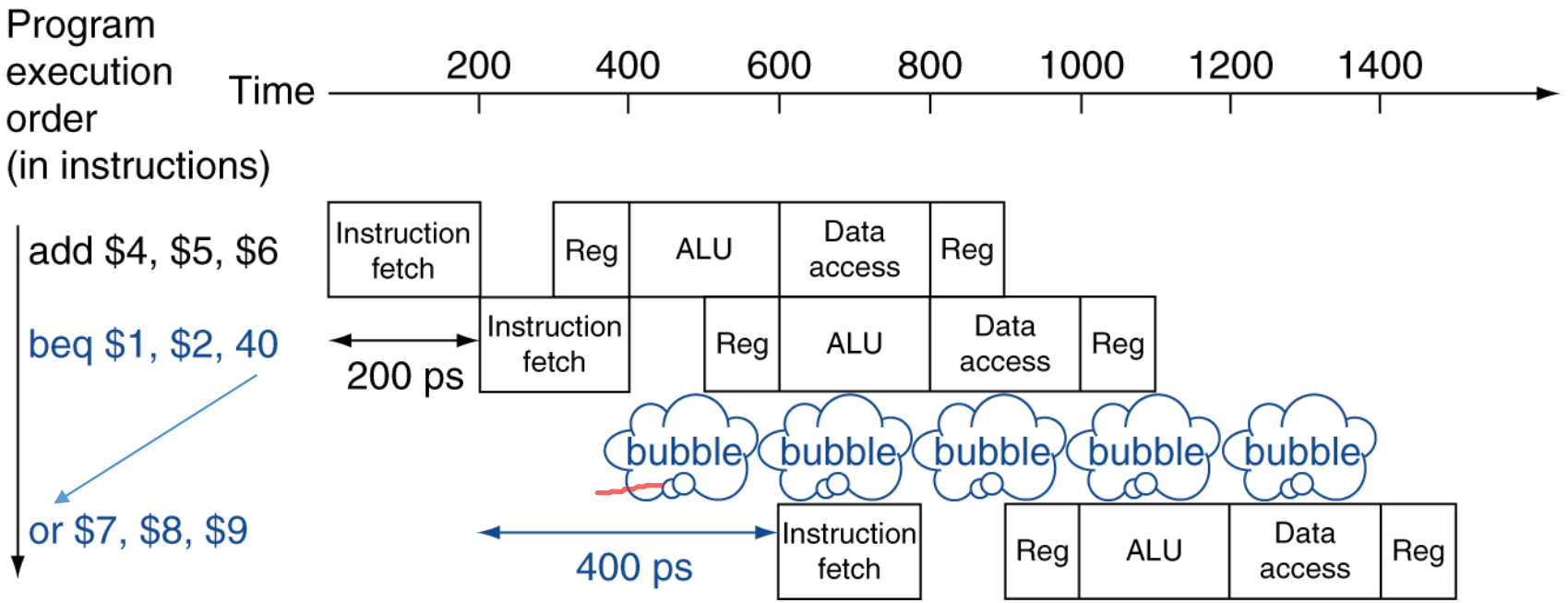
Reorder code to avoid stalls

Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Resolve Control Hazard 1: Stall on Branch

- Wait until branch outcome determined(in ID stage) before fetching next instruction



- Question: Estimate the impact on $CPI_{overall}$ of stalling on branches.
  - Branches are 17% of the instructions executed in SPECint2006
  - $CPI_{overall}$ = CPU clock cycles / instructions counts
    = (83x1 + 17x2) / 100 = 1.17

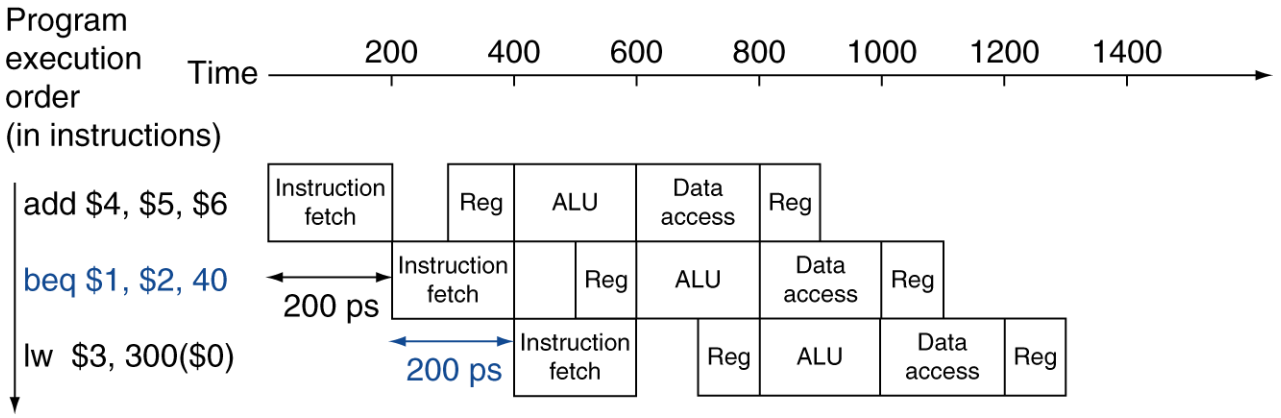
Resolve Control Hazard 2: Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

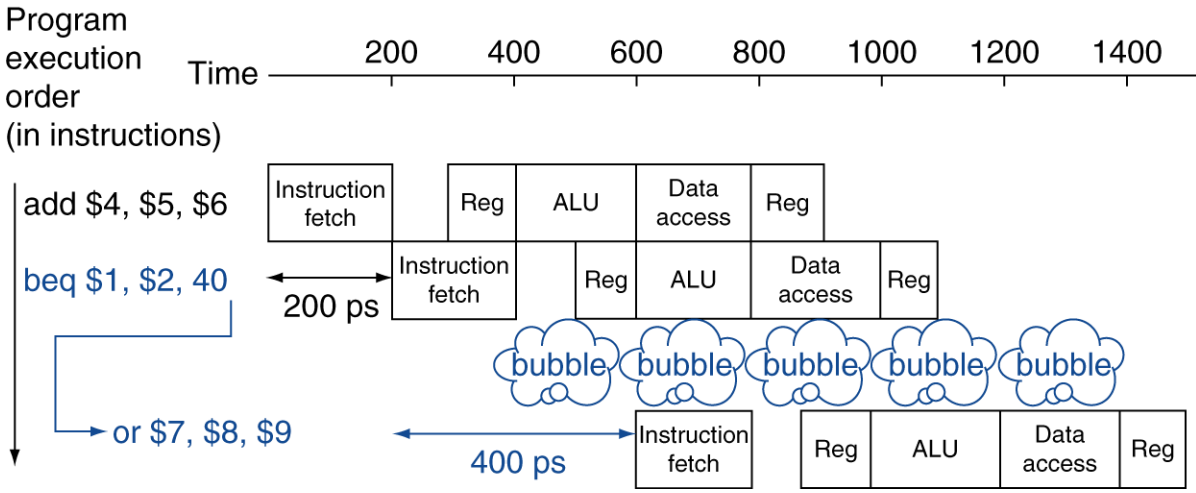
Prediction correct

no stall



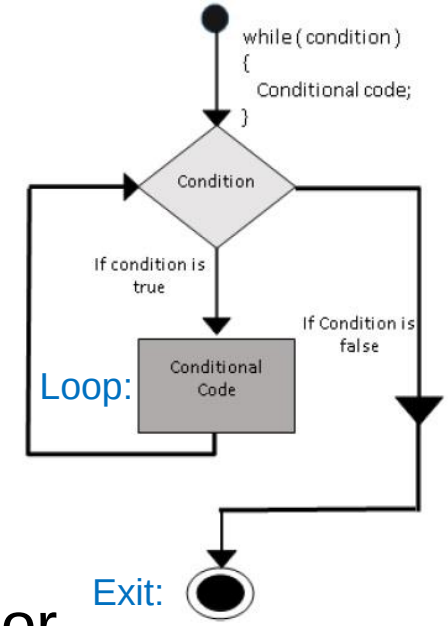
Prediction incorrect

one stall



More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history



Pipeline Summary

- Pipelining improves performance by increasing instruction **throughput**
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards

Type	Reason	Solution
Data hazards	Attempt to use data before it's ready	Forwarding(Bypassing) Stall (NOPs, Bubbles)
Control hazards	Attempt to make decision before condition is evaluated	Stall Branch prediction
Structural hazards	Attempt to use the same hardware	Build more hardware

- Instruction set design affects complexity of pipeline implementation