



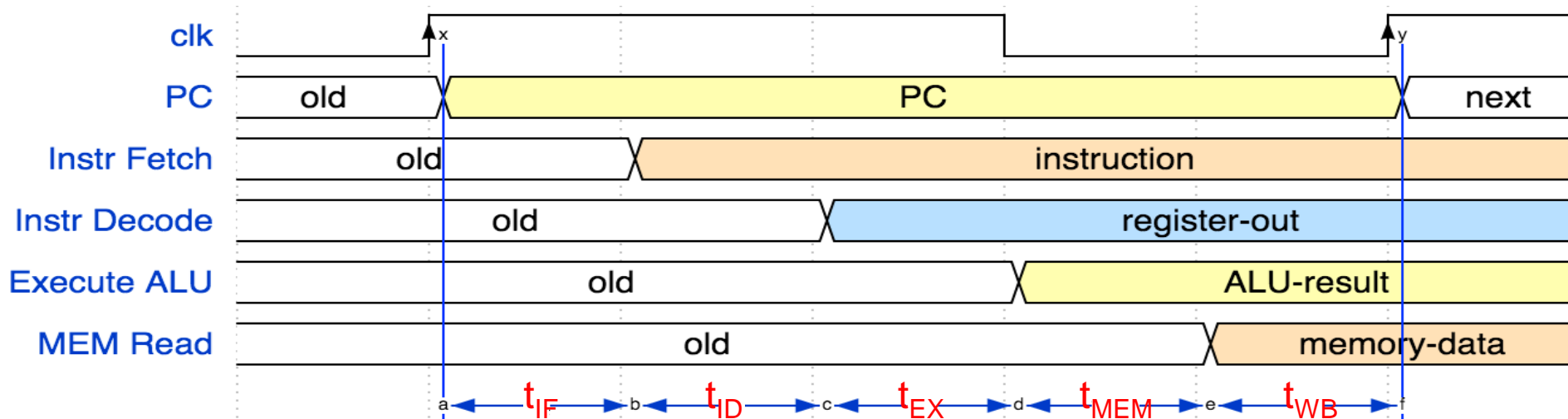
COMPUTER ORGANIZATION

Lecture 9 Pipeline Overview

2025 Spring

This PowerPoint is for internal use only at Southern University of Science and Technology. Please do not repost it on other platforms without permission from the instructor.

Instruction Timing – Single Cycle



Theoretical modeling(ideal single cycle), which could be slightly different with lab implementation

Example:

IF	ID	EX	MEM	WB	Total
I-MEM	Reg Read	ALU	D-MEM	Reg W	
200 ps	100 ps	200 ps	200 ps	100 ps	800 ps

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

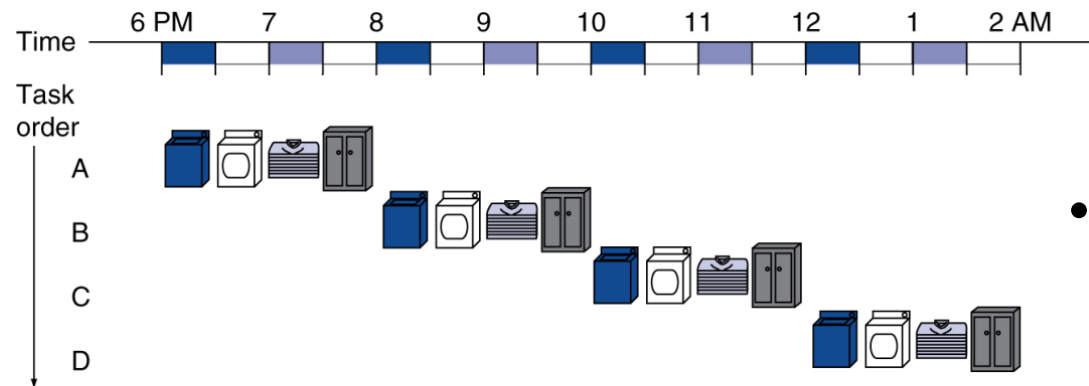
Performance Issues

Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	√	√	√		√	600ps
beq	√	√	√			500ps
lw	√	√	√	√	√	800ps
sw	√	√	√	√		700ps

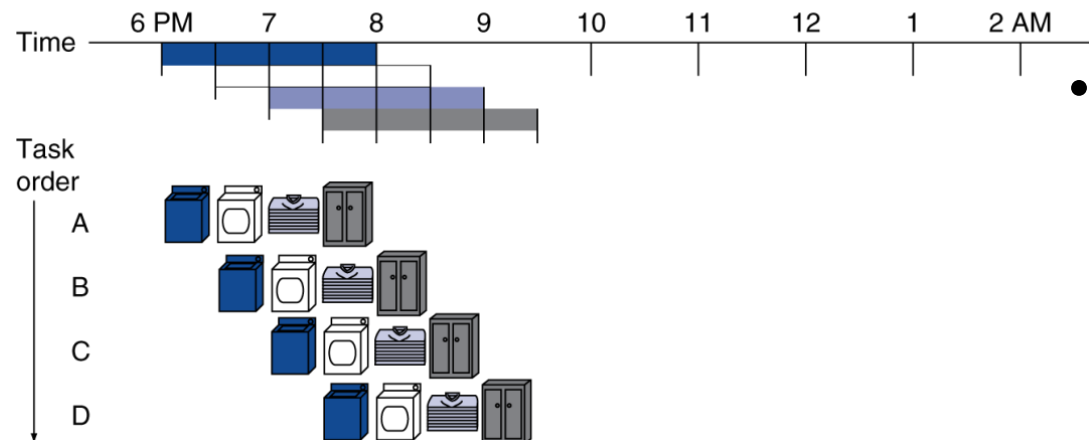
- Longest delay determines clock period
 - Critical path: load instruction
 - Instruction memory → register file → ALU → data memory → register file
 - Maximum clock rate in the above example
 - $f_{\max} = 1/800\text{ps} = 1.25 \text{ GHz}$
- Most blocks idle most of the time
 - E.g.: How can we keep ALU busy all the time?
 - Idea: Factories use three employee shifts - equipment is always busy! i.e., Pipelining

Pipelining Analogy

- Pipelined laundry: overlapping execution
 - Parallelism improves performance



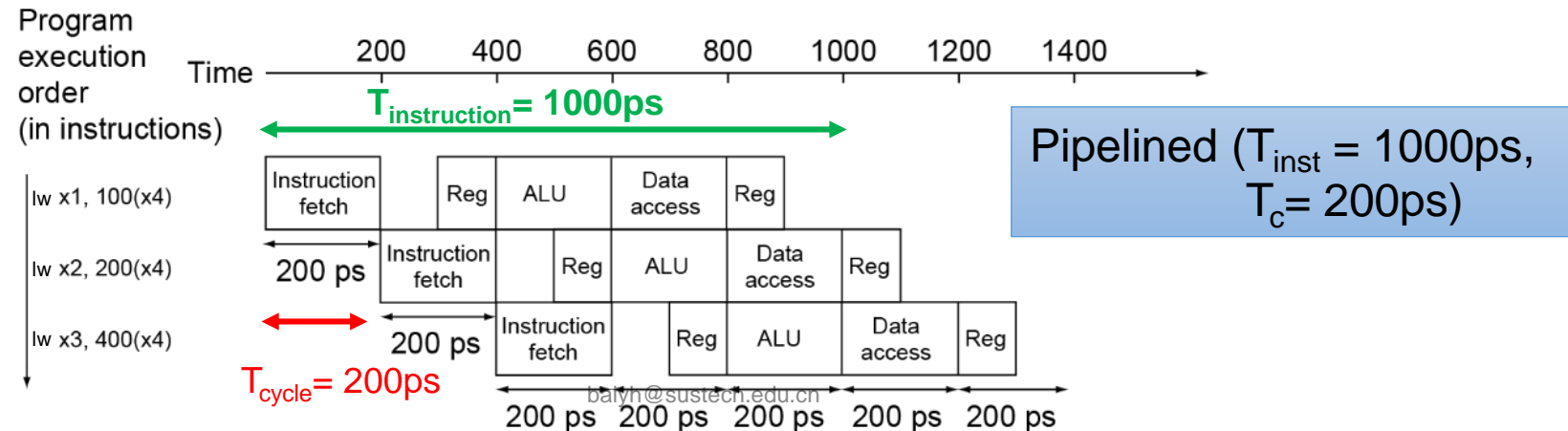
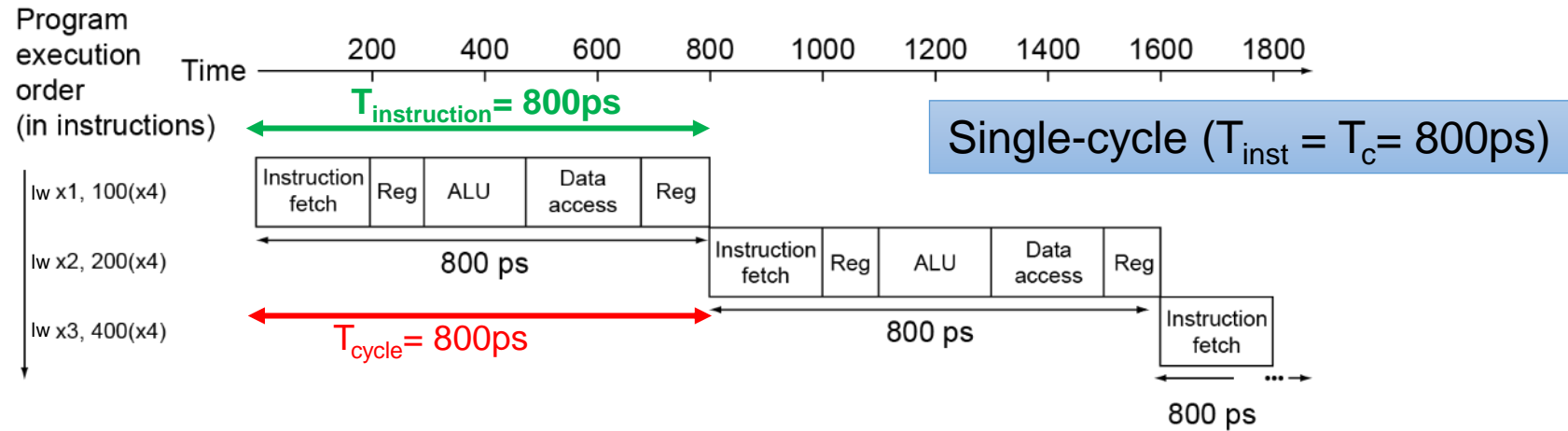
- Four loads(负载):
 - Speedup
 $= 8 / 3.5 = 2.3$



- n tasks:
 - Speedup
 $= 2n / (1.5 + 0.5n) \approx 4$
 $= \text{number of stages}$

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**

Analysis for prev. example	Single Cycle	Pipelining
stage or cycle time	$t_{\text{stage}} = 100 \text{ or } 200 \text{ ps}$ (Register access only 100 ps)	$t_{\text{cycle}} = 200 \text{ ps}$ (All cycles same length)
Instruction latency, t_{instr}	$= t_{\text{cycle}} = 800 \text{ ps}$	$= 5 * t_{\text{cycle}} = 1000 \text{ ps}$
Clock rate, freq	$1/800 \text{ ps} = 1.25 \text{ GHz}$	$1/200 \text{ ps} = 5 \text{ GHz}$
Speedup	1 x	4 x

Why only 4x speedup but not 5?

Pipelining and ISA Design

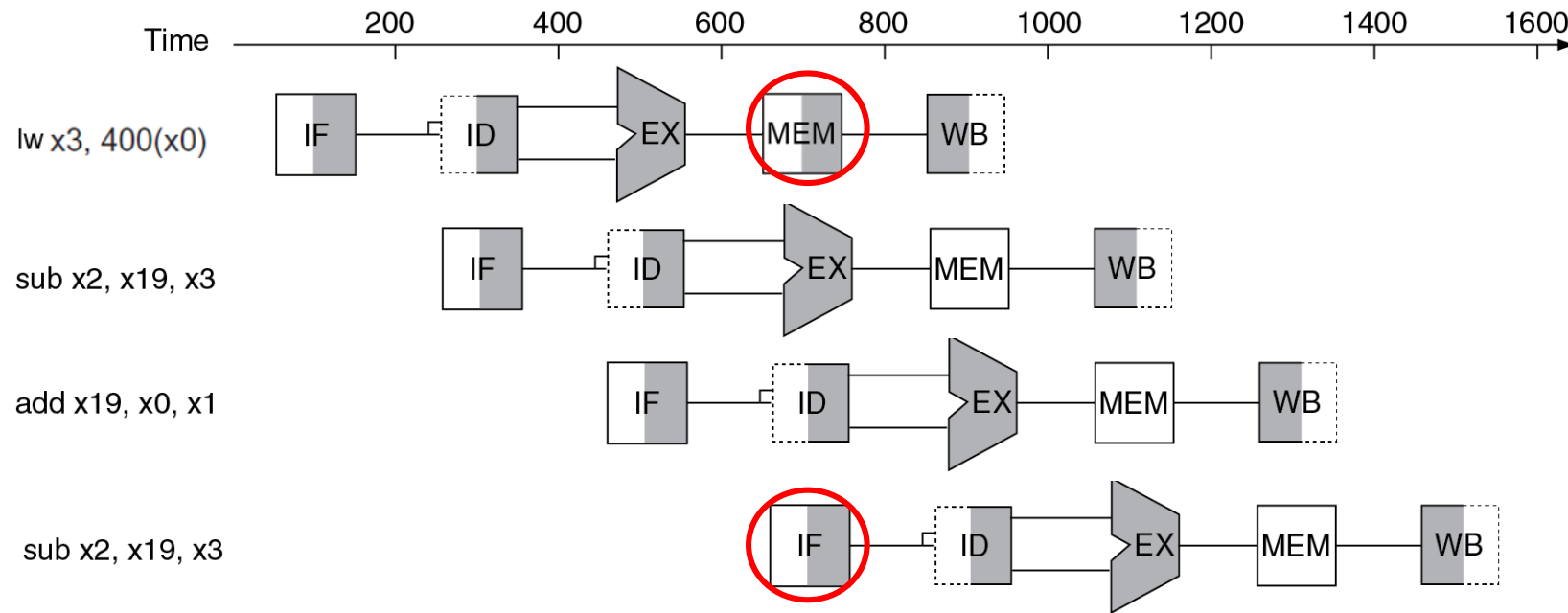
- RISC-V ISA designed for pipelining
 - All instructions are 32-bits
 - Easier to fetch and decode in one cycle
 - c.f. x86: 1- to 17-byte instructions
 - Few and regular instruction formats
 - Can decode and read registers in one step
 - Load/store addressing
 - Can calculate address in 3rd stage, access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

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 **stall (waiting)**
 - pipeline control must detect the hazard
 - and take action to resolve hazards

Structure Hazards

- Conflict for use of a resource
- Instruction and data memory used simultaneously
- In RISC-V, use two separate memories
 - Instruction Mem and Data Mem



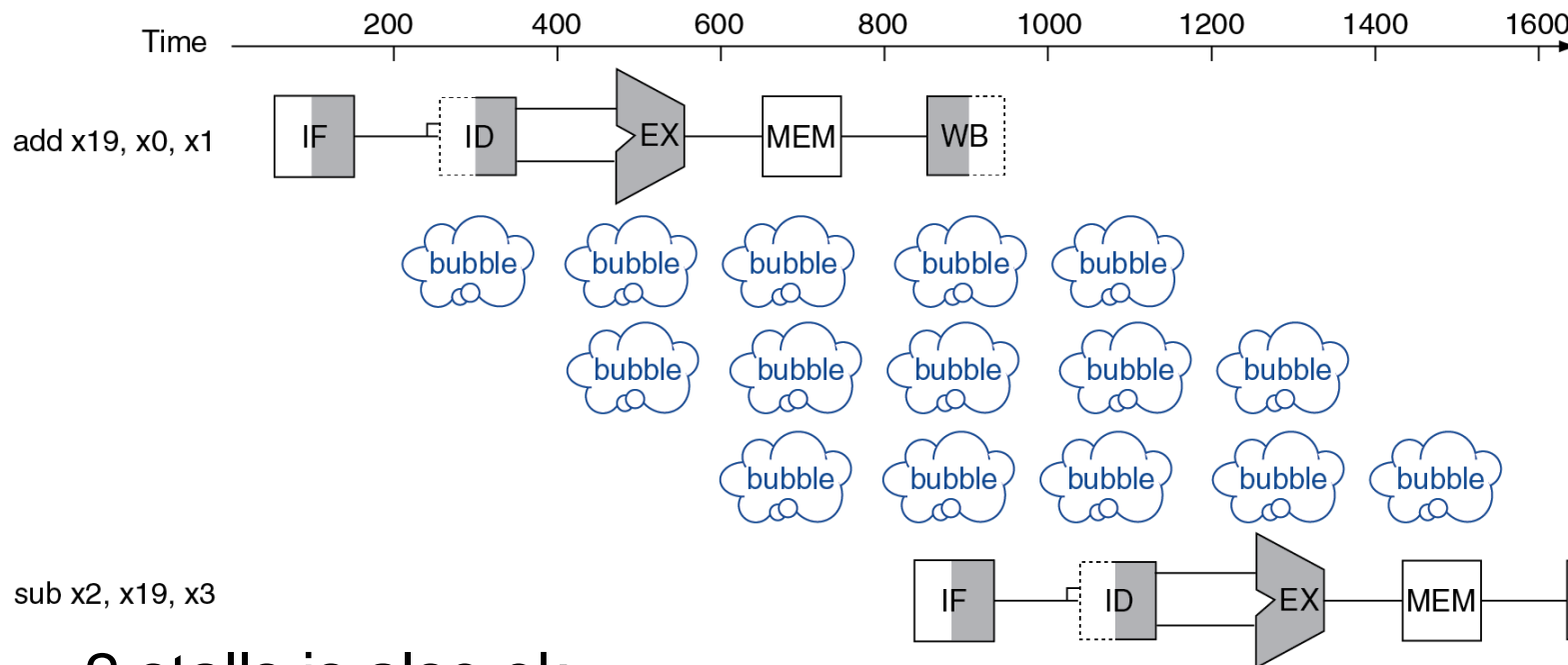
Data Hazards

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

Resolve Data Hazards 1: Stall

- Inserting NOP instructions (stall/bubble)

- add **x19**, x0, x1
 - sub x2, **x19**, x3



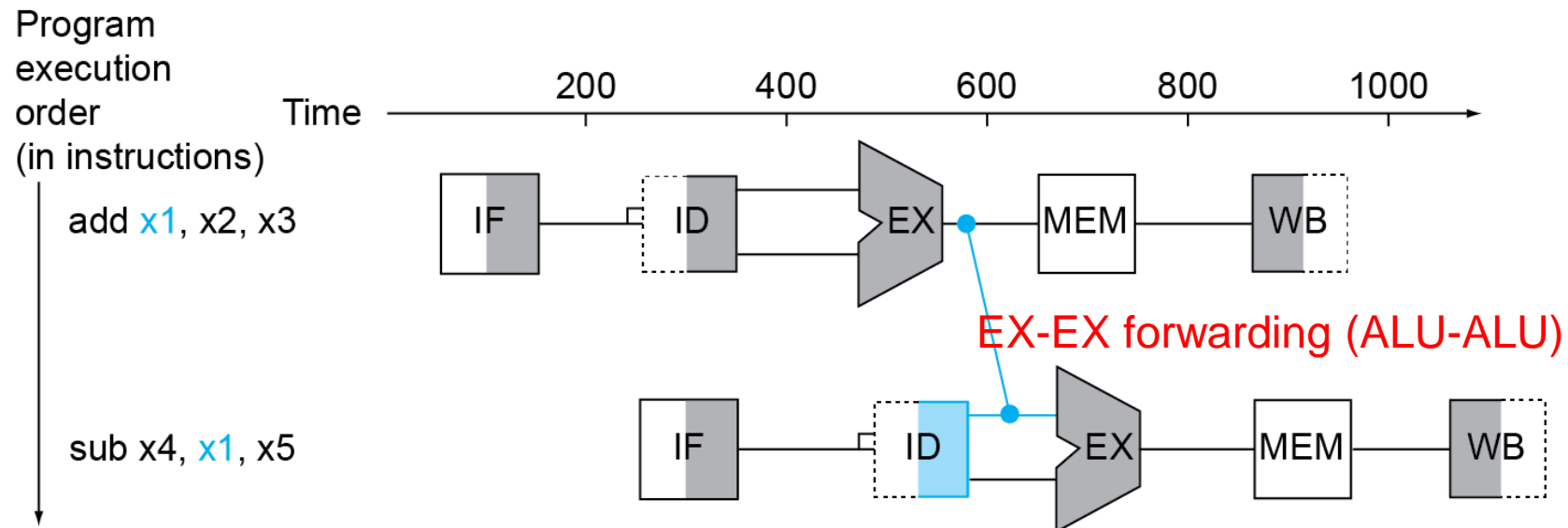
add **x19**, x0, x1
NOP
NOP
NOP
sub x2, **x19**, x3

- 2 stalls is also ok

- right half of ID is shaded because the register file is read
 - left half of WB is shaded because the register file is written

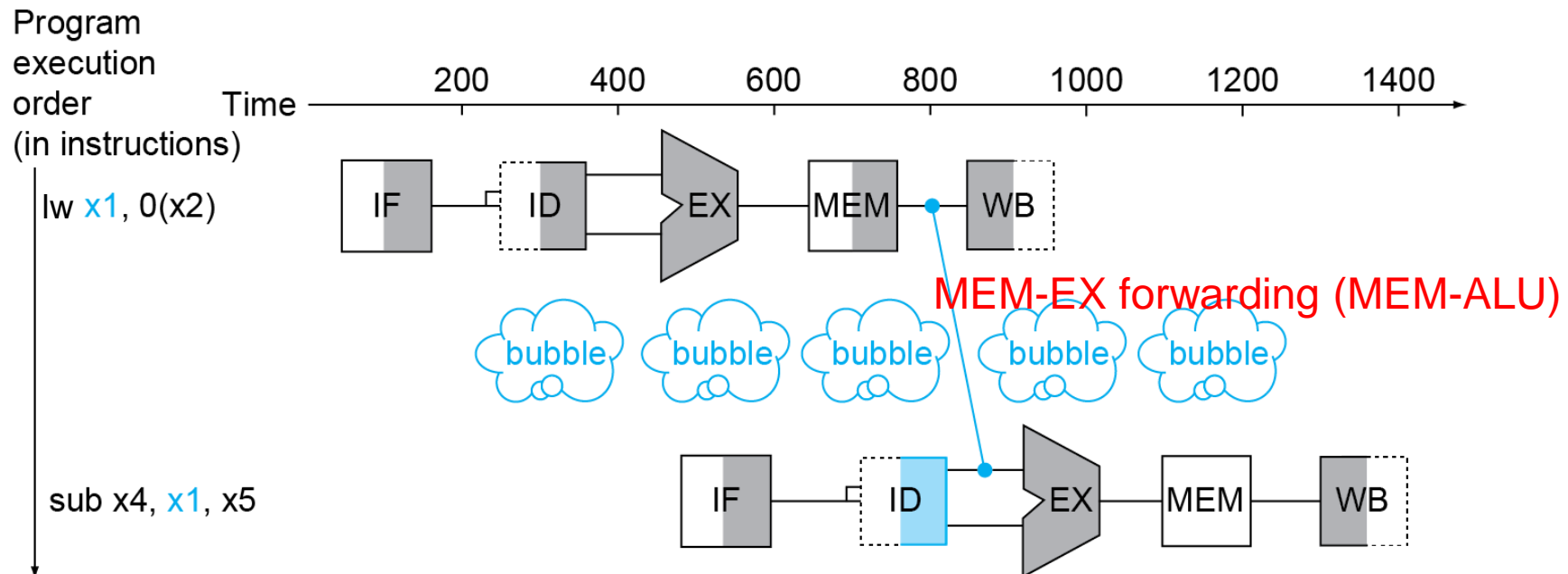
Resolve Data Hazards 2: Forwarding

- **Forwarding** can help to solve data hazard
- Core idea: Use result immediately when it is computed
 - Don't wait for it to be stored in a register
 - Requires extra connections in the datapath
 - Add a **bypassing line** to connect the output of EX to the input



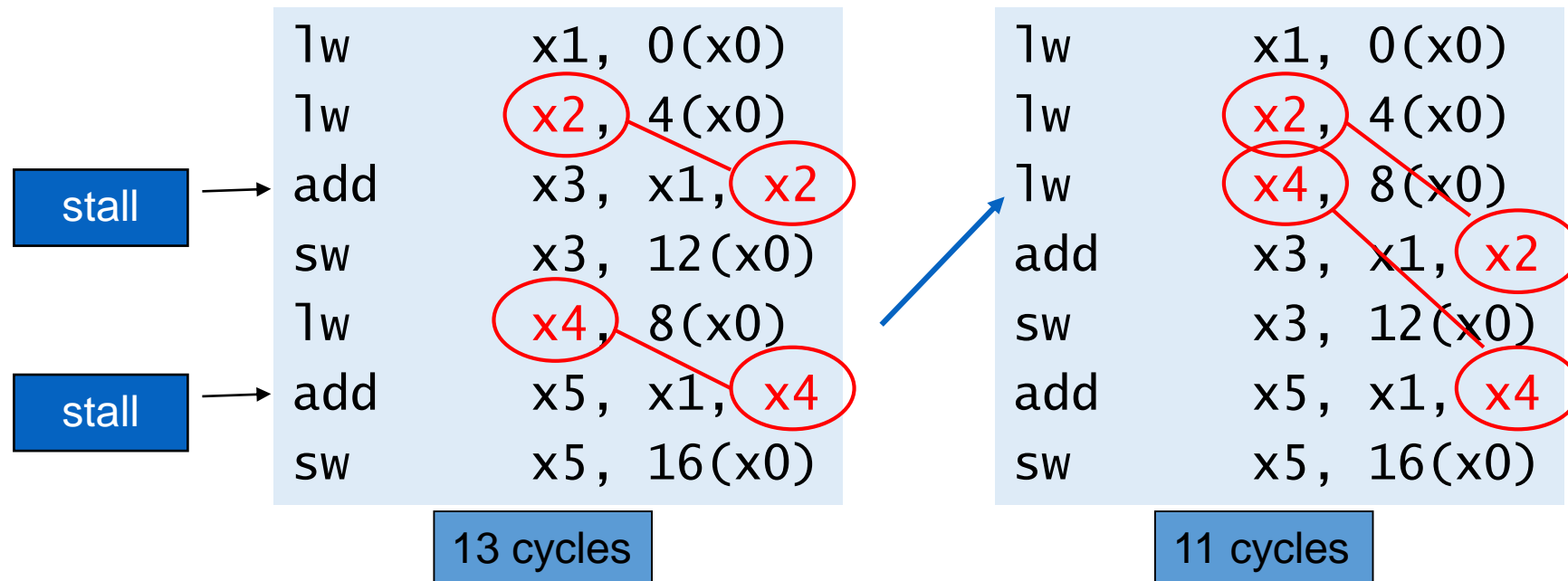
Resolve Data Hazards 3: Stall + forwarding

- For Load-Use Data Hazard, Can't avoid stalls by forwarding
 - If value not computed when needed
 - Can't forward backward in time!
 - E.g. lw **x1**, 0(x2) sub x4, **x1**, x5



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for $a = b + e$; $c = b + f$;
- (x1:b, x2:e, x3:a, x4:f, x5:c)



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

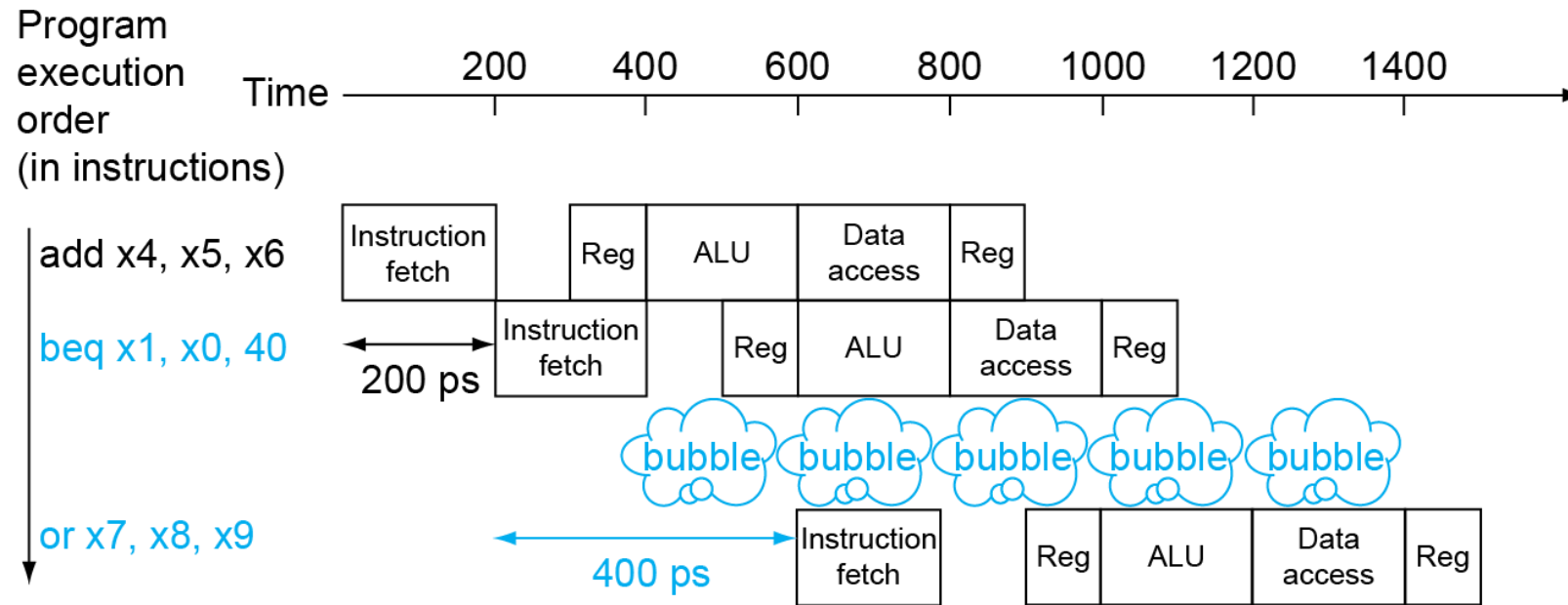
Reorder code to avoid stalls

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 RISC-V 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 before fetching next instruction
 - Note: we assume extra hardware is added to determine the branch outcome early in ID stage



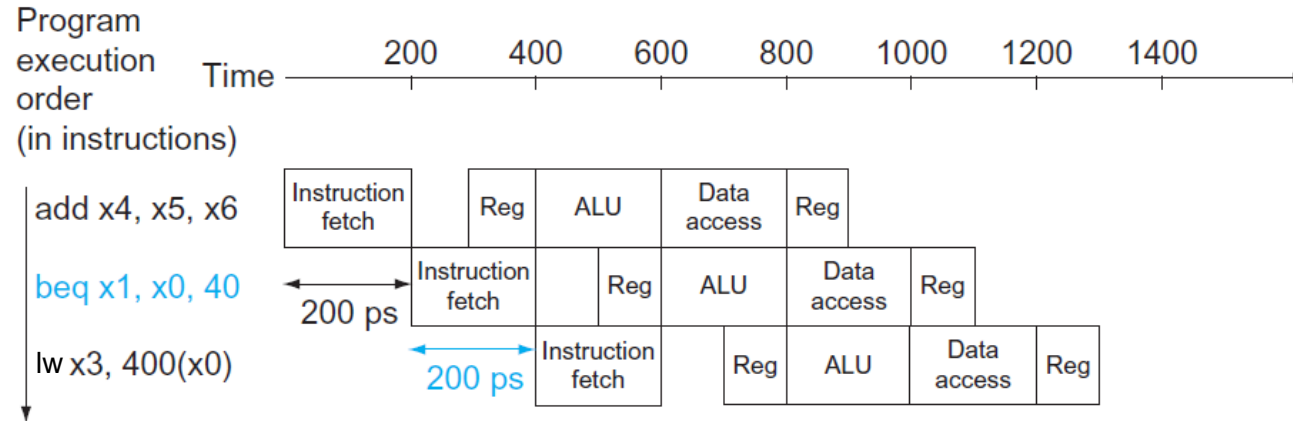
Resolve Control Hazard 2: Branch Prediction

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

RISC-V with Predict Not Taken

Prediction
correct

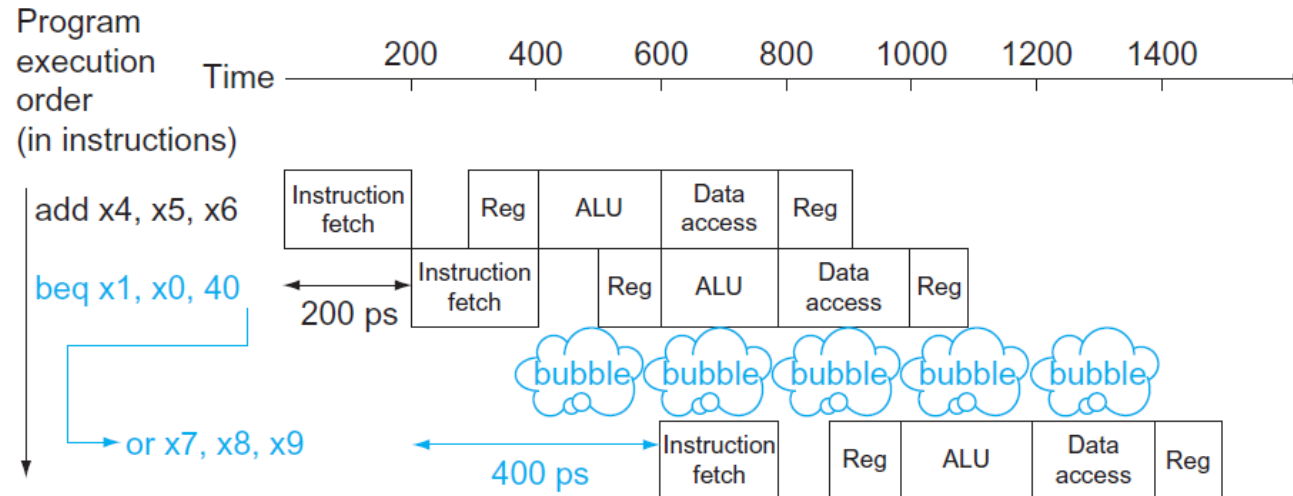
no stall



Note: we assume extra hardware is added to
determine the branch outcome early in ID stage

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
 - Structure, data, control
- Instruction set design affects complexity of pipeline implementation