Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.

Seoul National University

Fall 2019

# Pipelining

Chap. 4.5

# Introduction to Pipelining

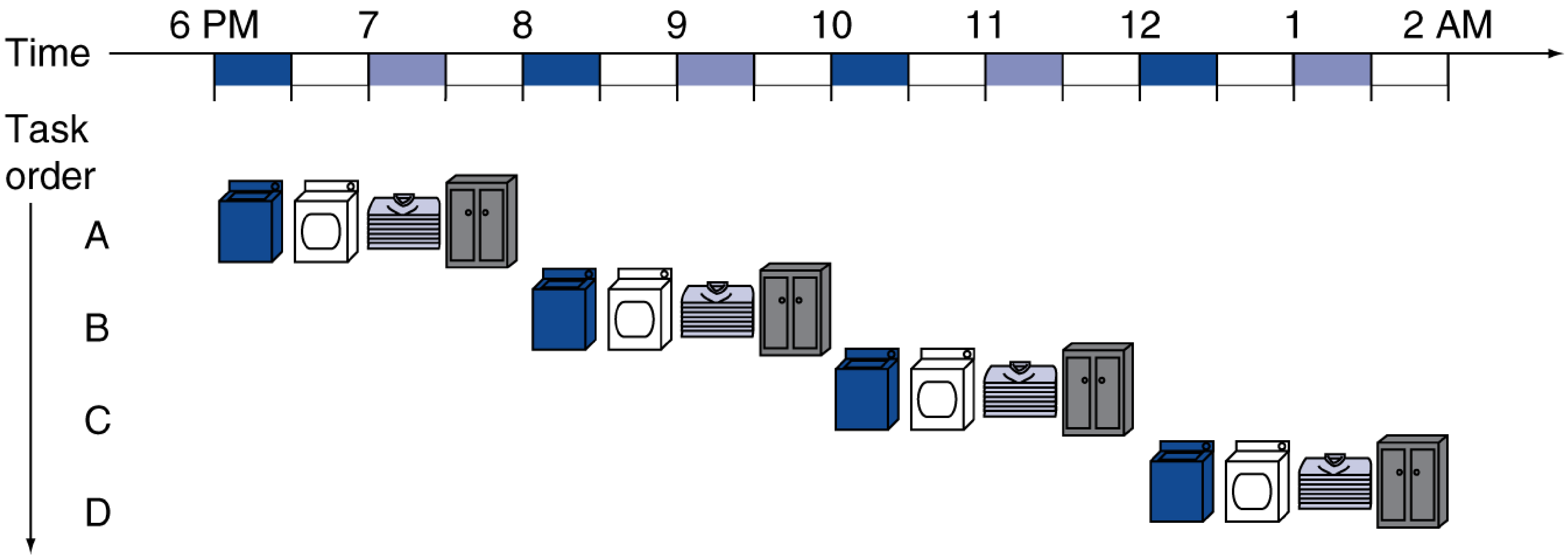
# Sequential Processing

# Parallel Processing (Multi-core)

# Pipelining

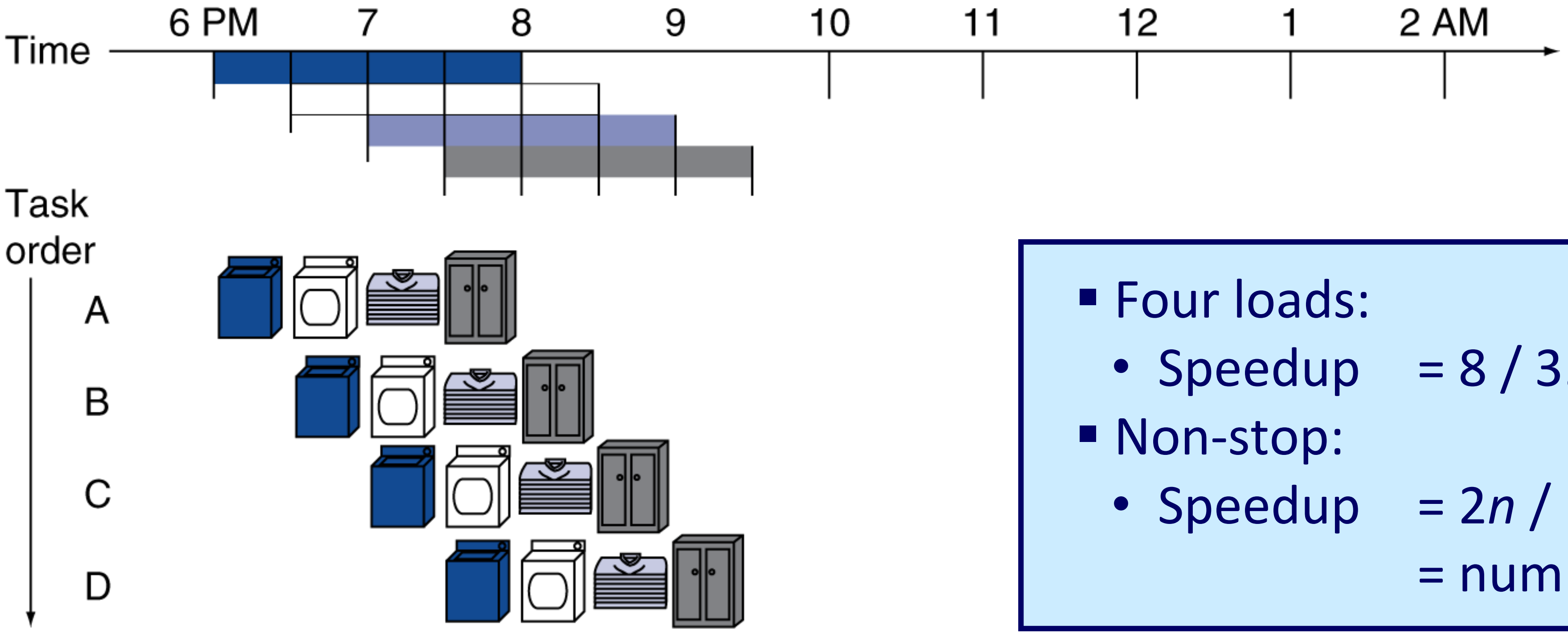
# Laundry Example

- Sequential processing: Wash-Dry-Fold-Store



# Pipelined Laundry Example

- Overlapping execution
- Parallelism improves performance




- Four loads:
  - Speedup = 8 / 3.5 = 2.3
- Non-stop:
  - Speedup = $2n / (0.5n + 1.5) \simeq 4$
    = number of stages

# A RISC-V Pipeline

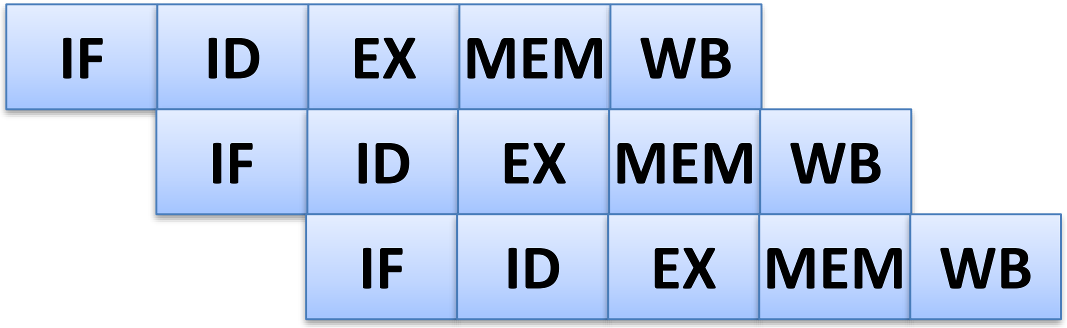
- Five stages, one step per stage

- IF: Instruction fetch from memory
- ID: Instruction decode & register read
- EX: Execute operation or calculate address
- MEM: Access memory operand
- WB: Write result back to register

# Pipelined Instruction Execution

- Sequential execution


- Pipelined execution




```
add x10, x11, x12
    sub x13, x14, x15
      and x5, x6, x7
```

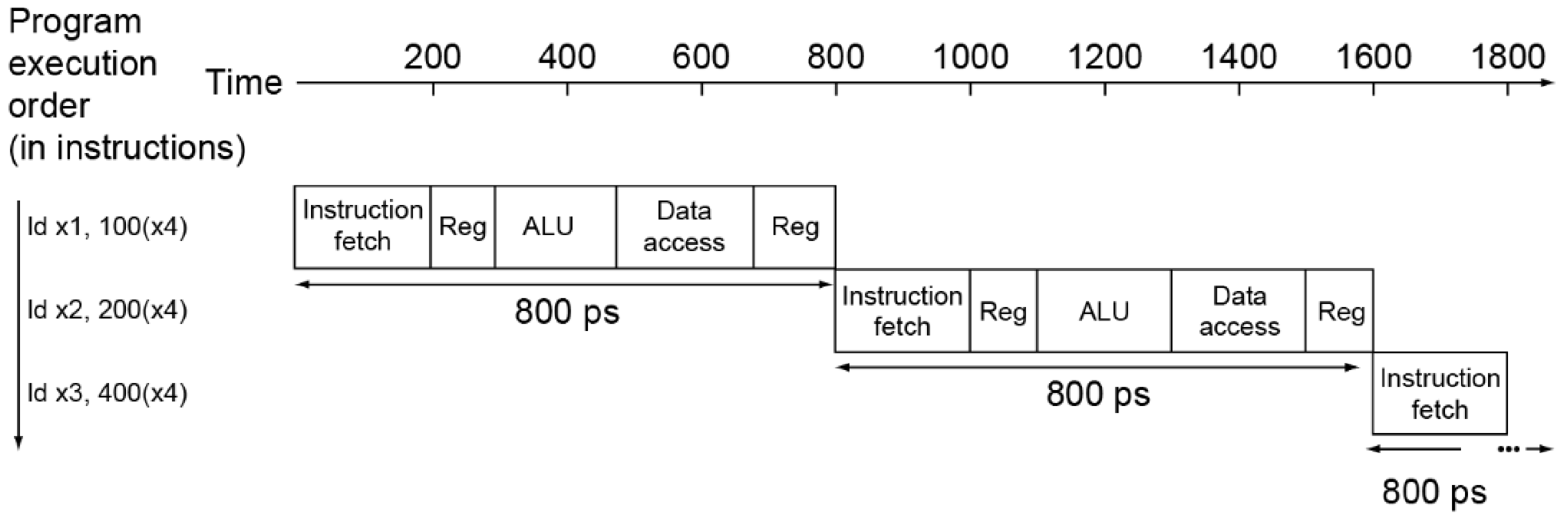
# Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare piplined datapath with single-cycle datapath

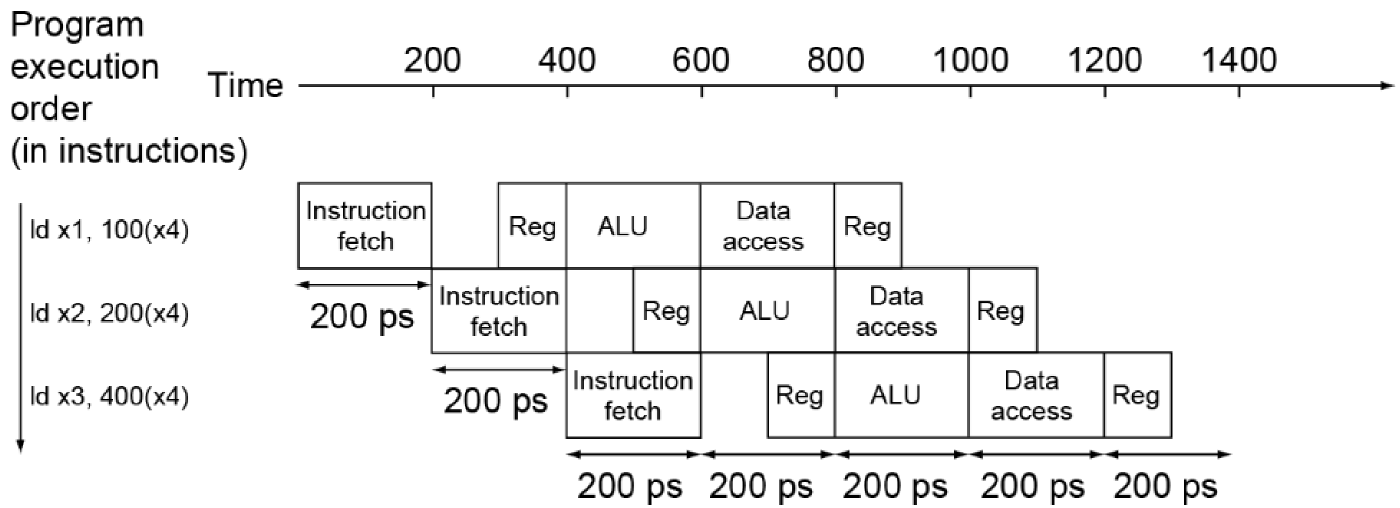
| Inst. | Inst. fetch | Register read | ALU op. | Memory access | Register write | Total time |
|---|---|---|---|---|---|---|
| `ld` | 200ps | 100ps | 200ps | 200ps | 100ps | 800ps |
| `sd` | 200ps | 100ps | 200ps | 200ps | | 700ps |
| R-type | 200ps | 100ps | 200ps | | 100ps | 600ps |
| `beq` | 200ps | 100ps | 200ps | | | 500ps |

# Pipeline Performance (cont'd)

**Single-cycle**
**($T_c$ = 800ps)**




**Pipelined**
**($T_c$ = 200ps)**



# Pipeline Speedup

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

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

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

# Pipelining and ISA Design

- RISC-V ISA designed for pipelining

- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - cf. x86: 1- to 17-byte instructions
- Few and regular instruction formats
  - Source and destination register fields located in the same place
  - Can decode and read registers in one step
- Load/store addressing
  - Can calculate address in 3rd EX stage, access memory in 4th MEM stage

# Pipeline Hazards

# Hazards

- Situations that prevent starting the next instruction in the next cycle

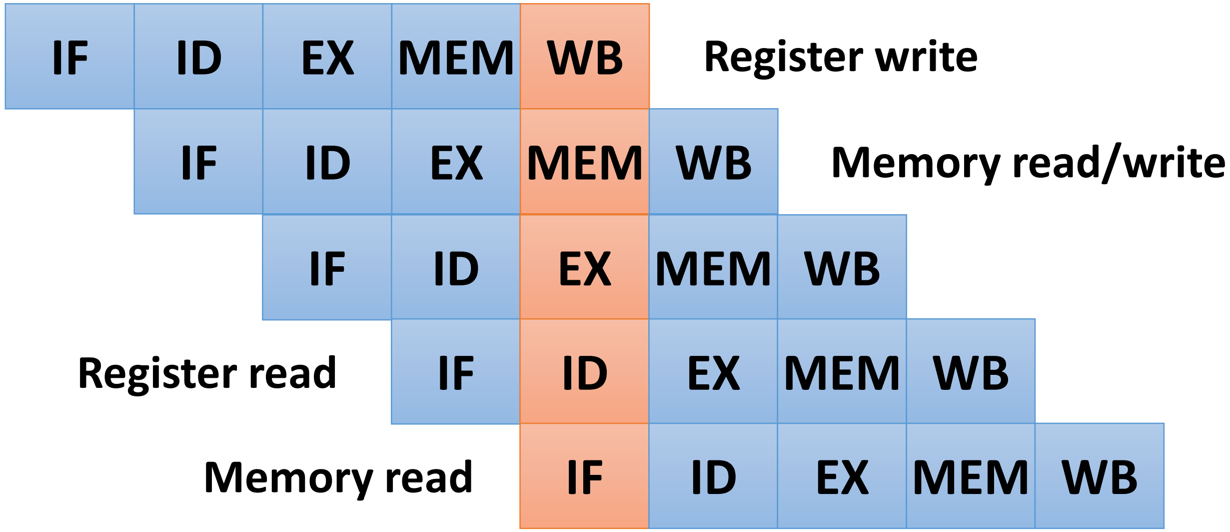
- Structural hazard
  - A required resource is busy

- Data hazard
  - Need to wait (or *stall*) for previous instruction to complete its data read/write

- Control hazard
  - Deciding on control action depends on previous instruction

# Structural Hazard

- Conflict for use of a resource
- In RISC-V 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)
- Register file also requires multiple ports (for 2 reads and 1 write)

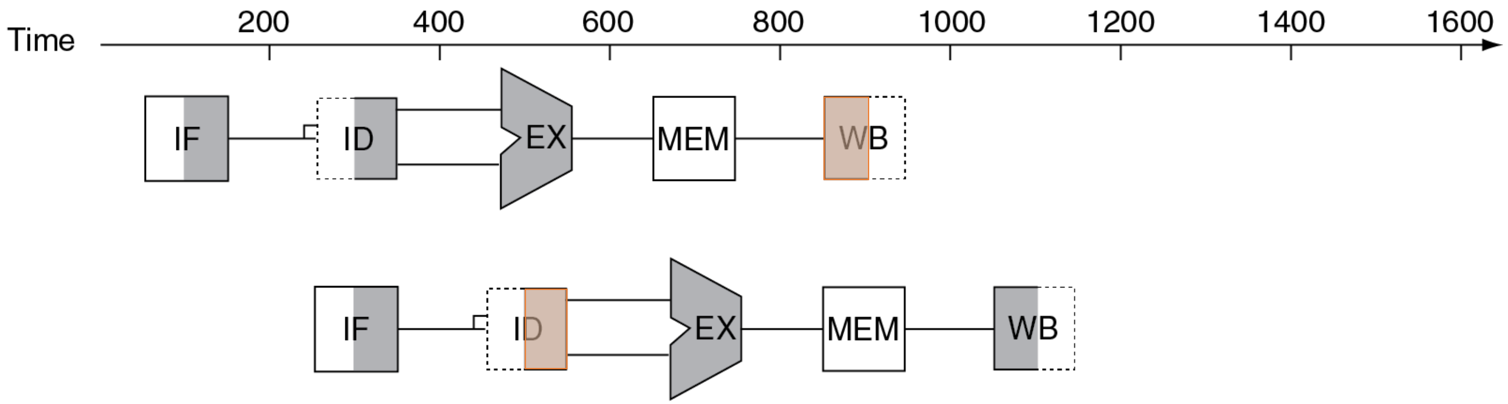


# Data Hazard

- An instruction depends on completion of data access by a previous instruction
- Also called "Read-After-Write (RAW)" hazard
- This hazard results from an actual need for communication

```
add  x19, x0, x1

sub  x2, x19, x3
```



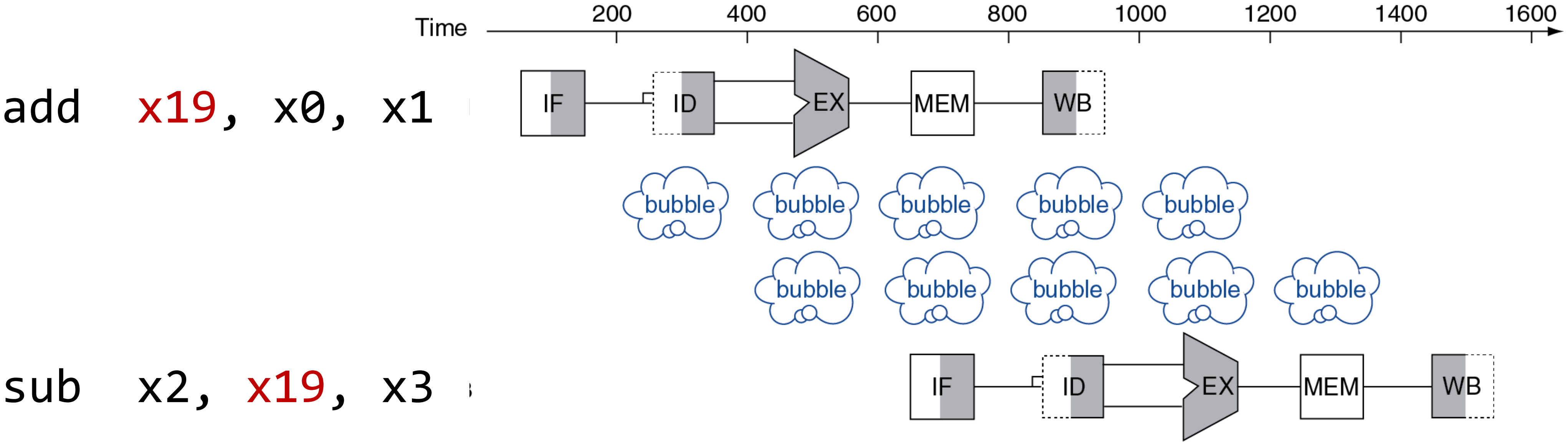
# Solutions to Data Hazard

- Freezing the pipeline
- Forwarding
- Compiler scheduling
- Out-Of-Order execution (discussed later)

# Freezing the Pipeline

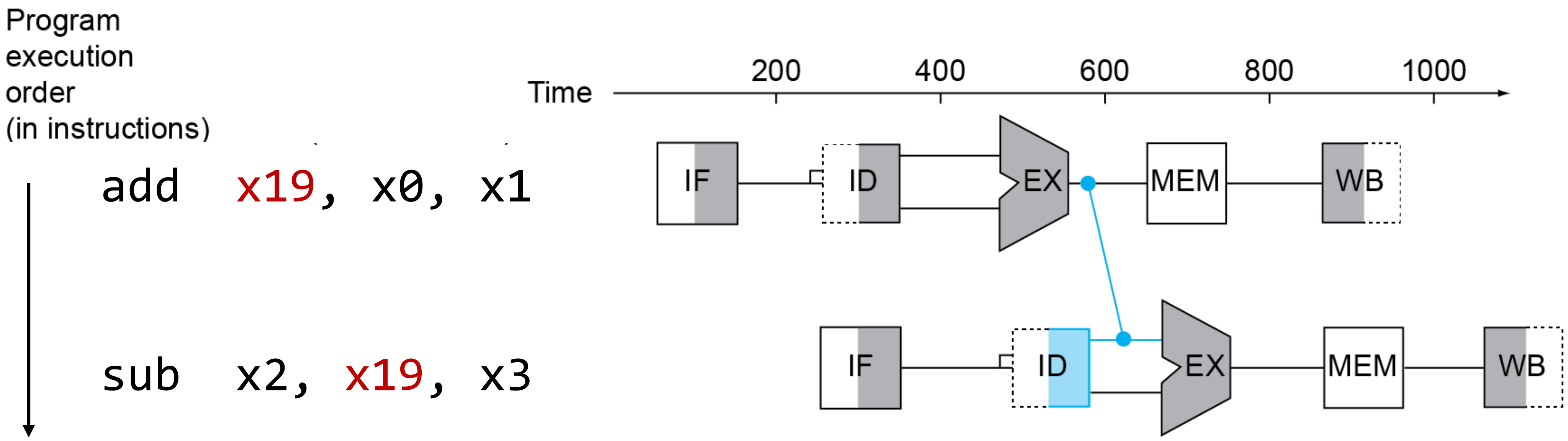
- Stall the pipeline until dependences are resolved

- ALU result to next instruction (2 stalls)



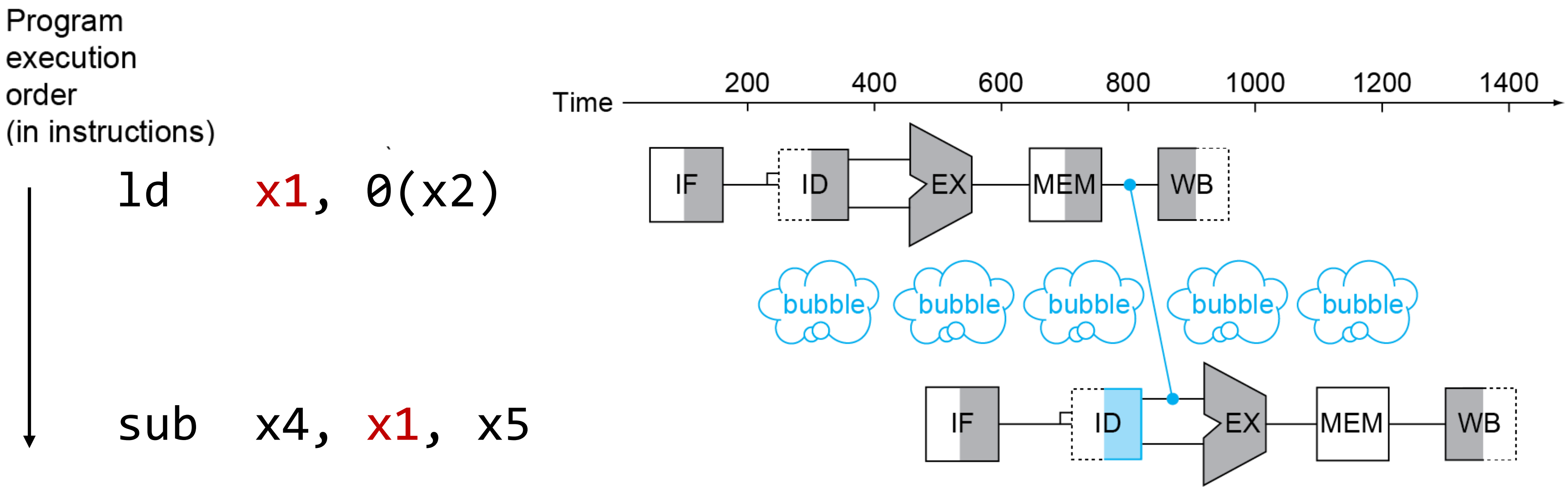
# Forwarding (or Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



# Forwarding: Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



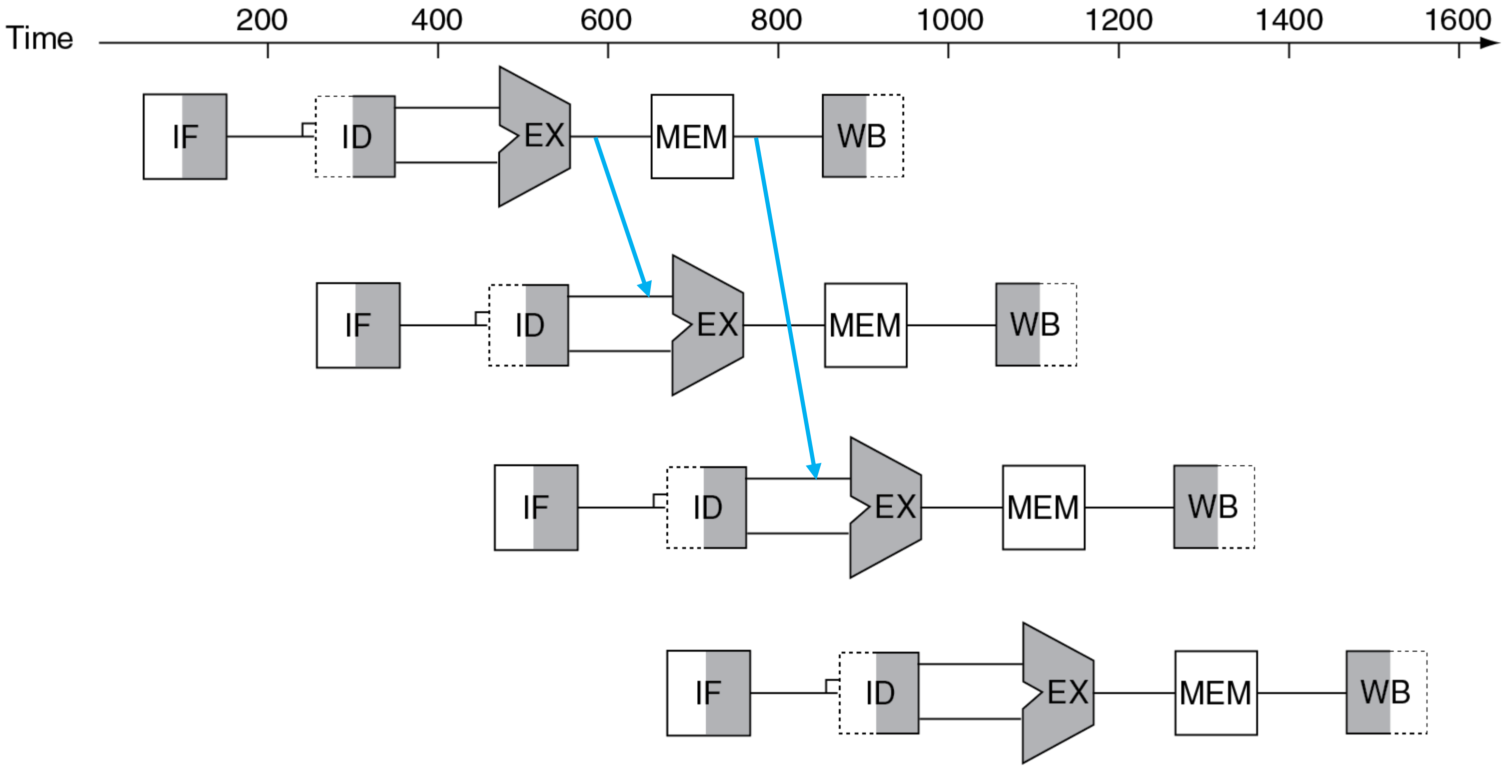
# Forwarding: Multiple Readers

```
add  x10, x4, x5

sub  x6, x10, x4

and  x7, x10, x0

xor  x8, x10, x3
```



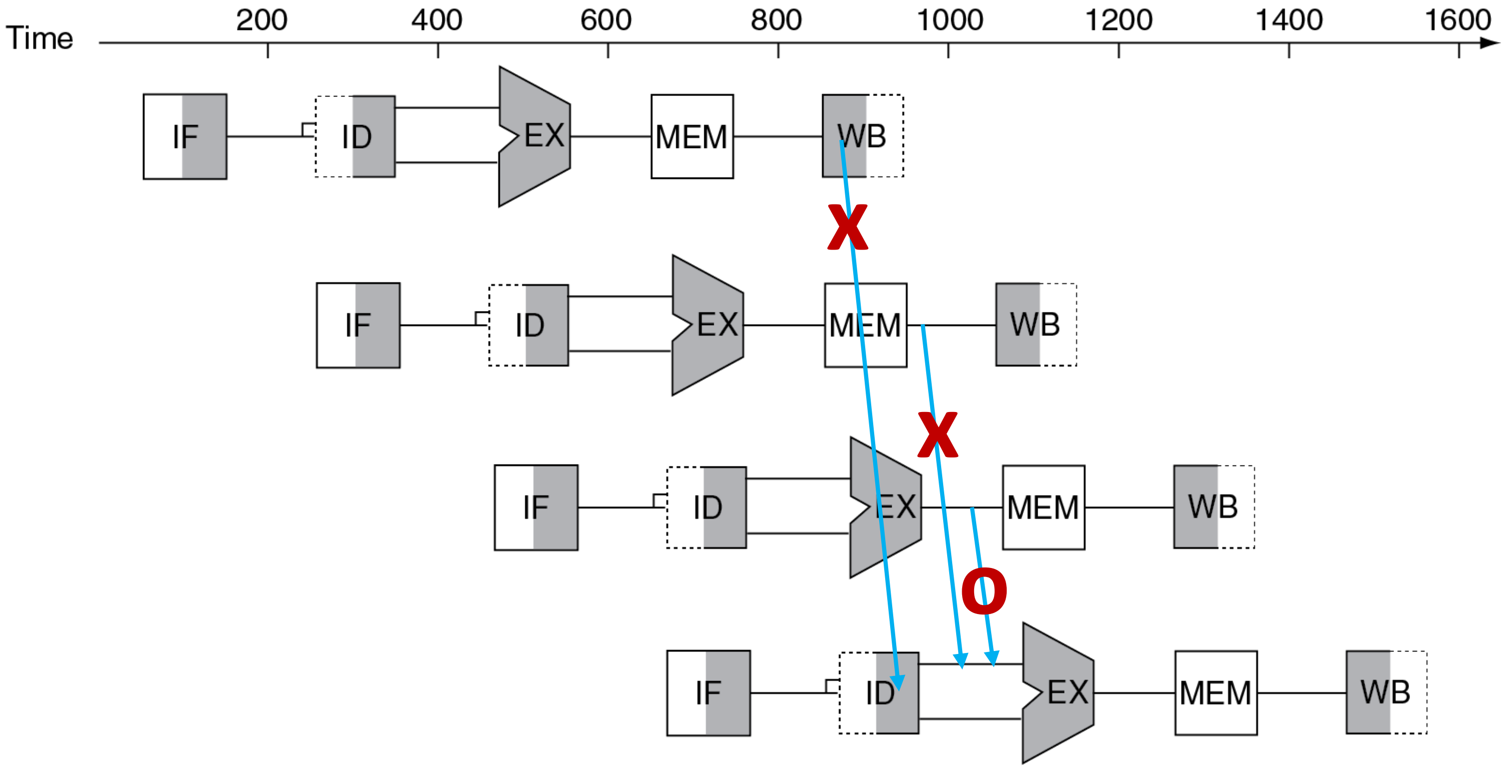
# Forwarding: Multiple Writers

```
add  x10, x4, x5

sub  x10, x5, x6

addi x10, x2, 1

xor  x8, x10, x7
```



# Compiler Scheduling

- Reorder code to avoid use of load result in the next instruction
- C code for `v[3] = v[0] + v[1];  v[4] = v[0] + v[2];`

```
        ld    x1, 0(x0)
        ld    x2, 8(x0)
Stall → add   x3, x1, x2
        sd    x3, 24(x0)
        ld    x4, 16(x0)
Stall → add   x5, x1, x4
        sd    x5, 32(x0)
```

**13 cycles**

```
ld    x1, 0(x0)
ld    x2, 8(x0)
ld    x4, 16(x0)
add   x3, x1, x2
sd    x3, 24(x0)
add   x5, x1, x4
sd    x5, 32(x0)
```

**11 cycles**

# Control Hazard

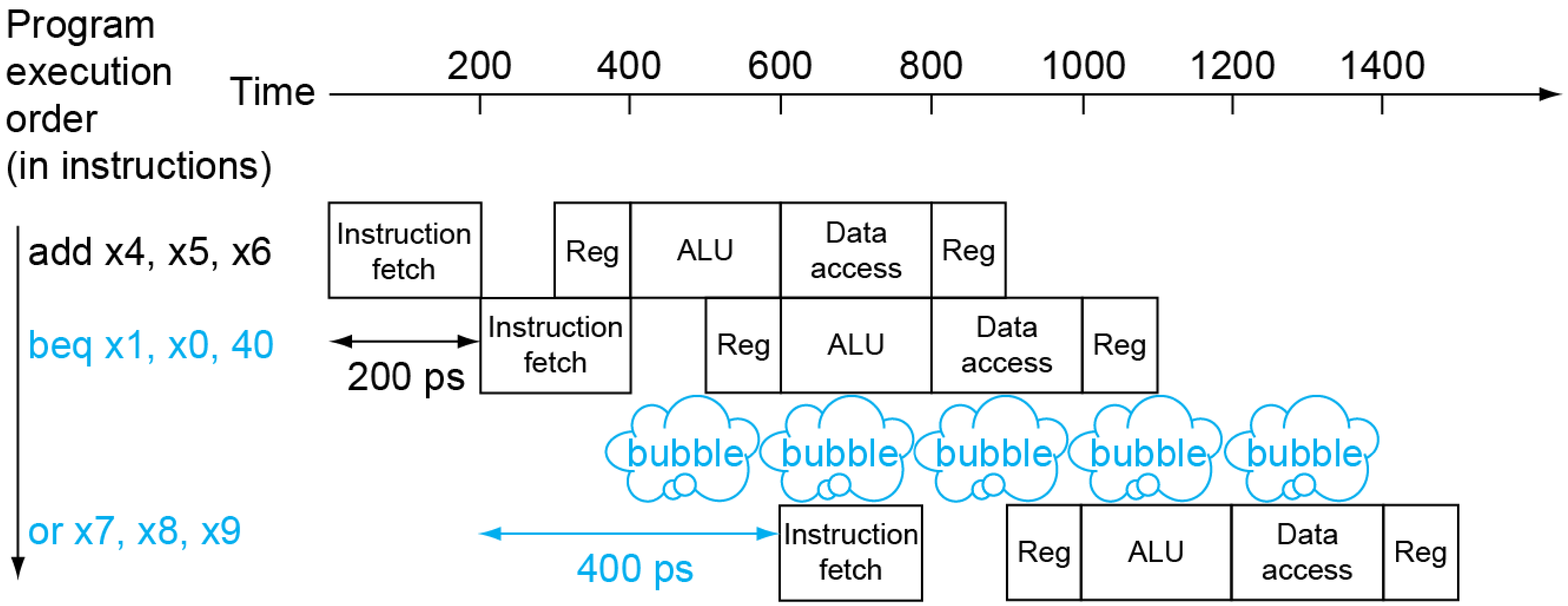
- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction: still working on ID stage of branch
- In RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in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

# Solutions to Control Hazard

- Stall on branch
- Branch prediction
- Delayed branch (compiler scheduling to avoid stalls)

# Stall on Branch

- Wait until branch outcome determined before fetching next instruction



# 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
  - Cancel the fetched instruction if the prediction was wrong

# 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

# Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structural, data, control
- Instruction set design affects complexity of pipeline implementation