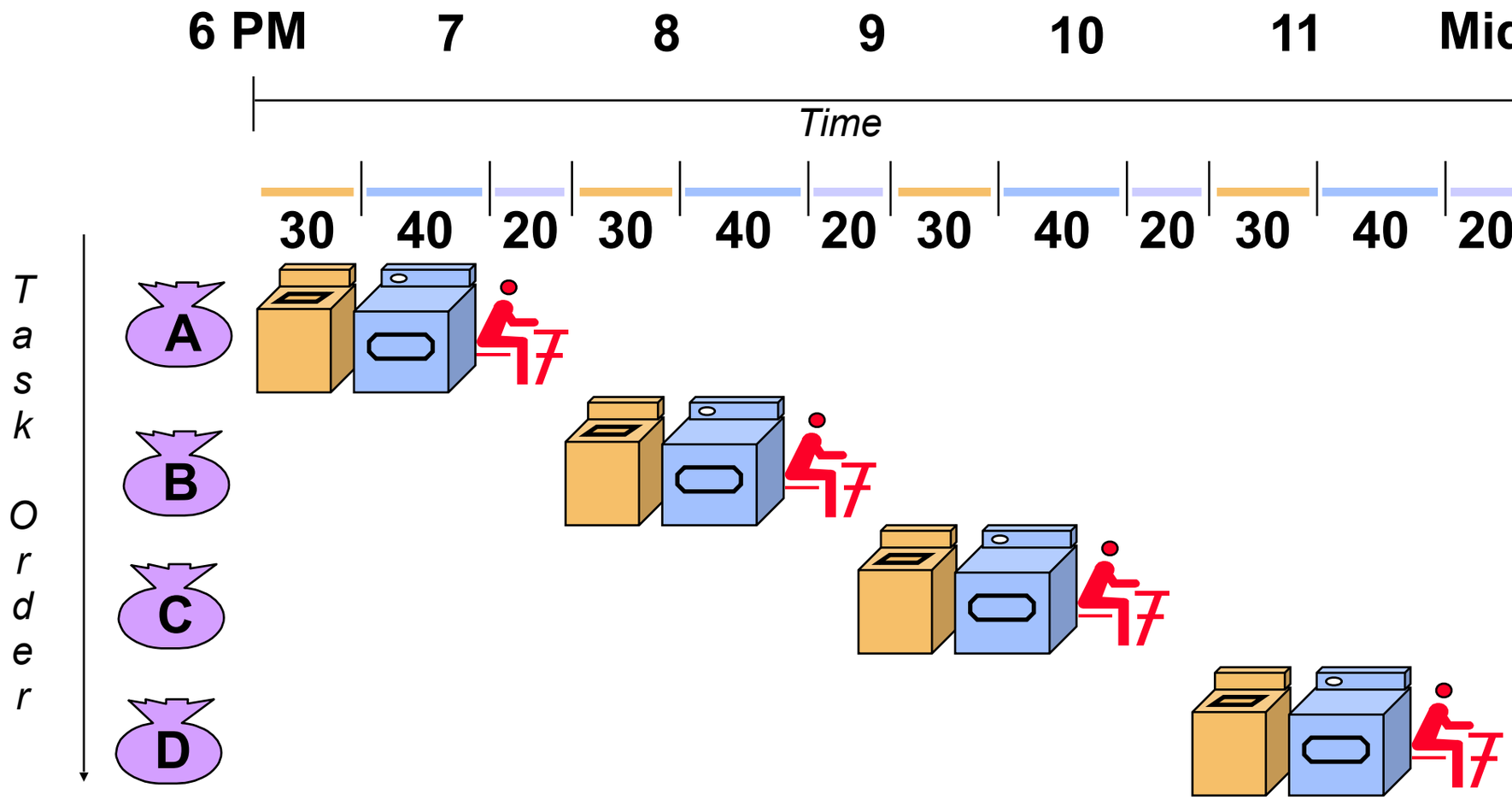
Lecture 15: Pipelining

Spring 2024
Jason Tang

Topics

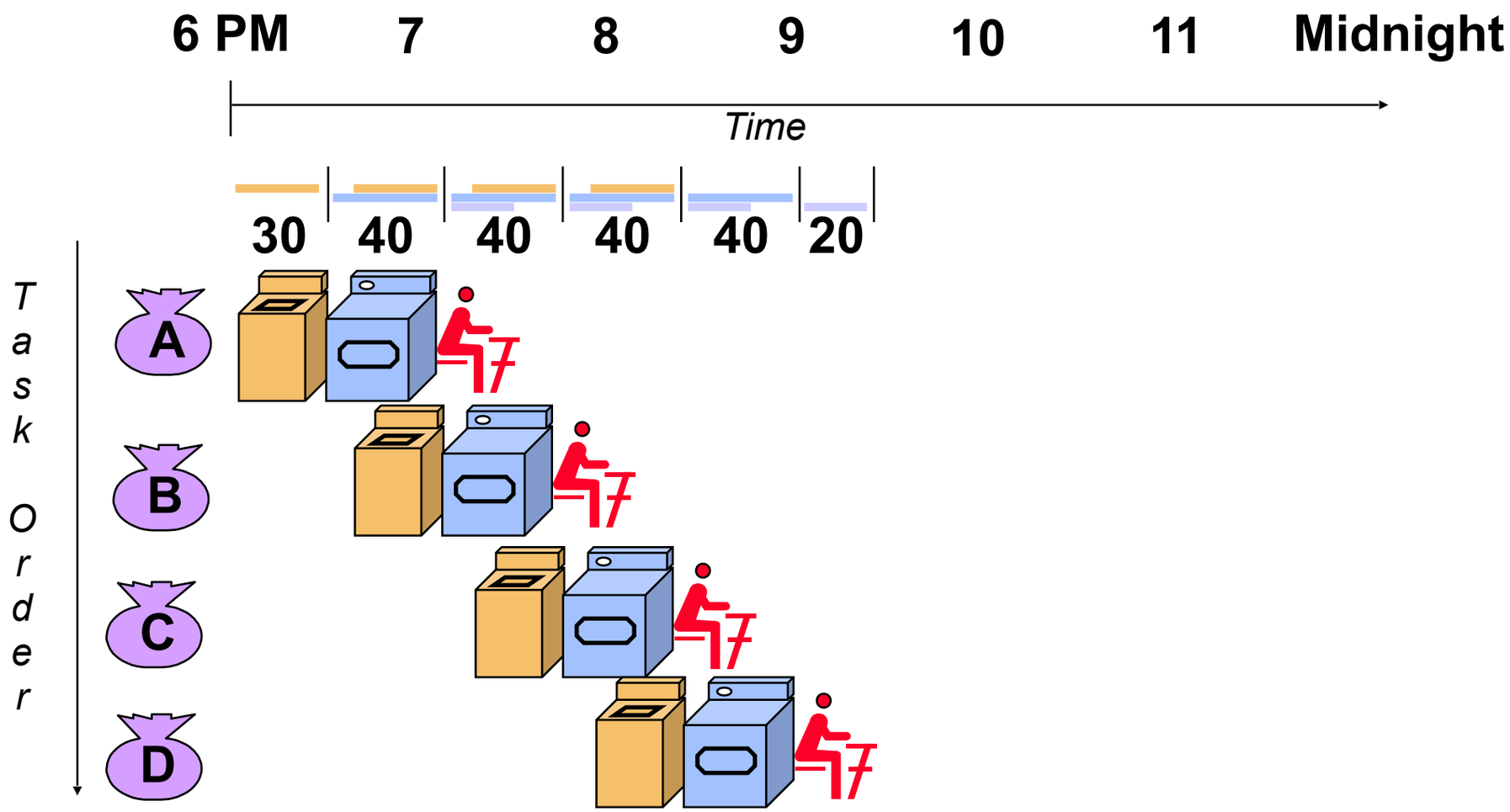
- Overview of pipelining
- Pipeline performance
- Pipeline hazards

Sequential Laundry



- A clothes washer takes 30 minutes, dryer takes 40 minutes, and folding takes 20 minutes
  - Sequential laundry would thus take **6 hours** for 4 loads

Pipelined Laundry



- Pipelining means start work as soon as possible
  - Pipelined laundry would thus take **3.5 hours** for 4 loads

Pipelining

- Does not improve latency of a single task, but improves throughput of entire workload

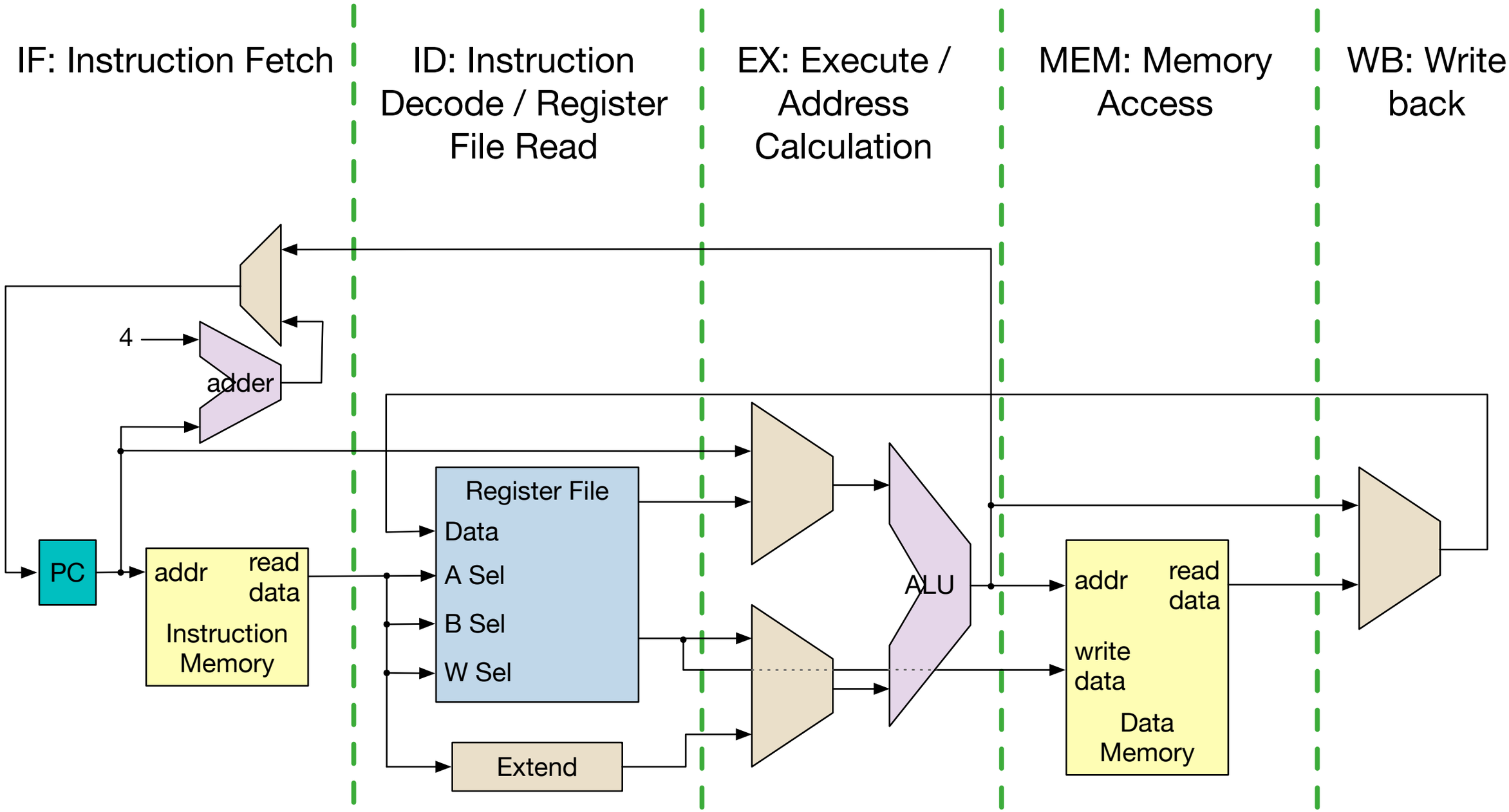
- Pipeline rate limited by slowest pipeline stage

- Multiple tasks operating simultaneously using different resources

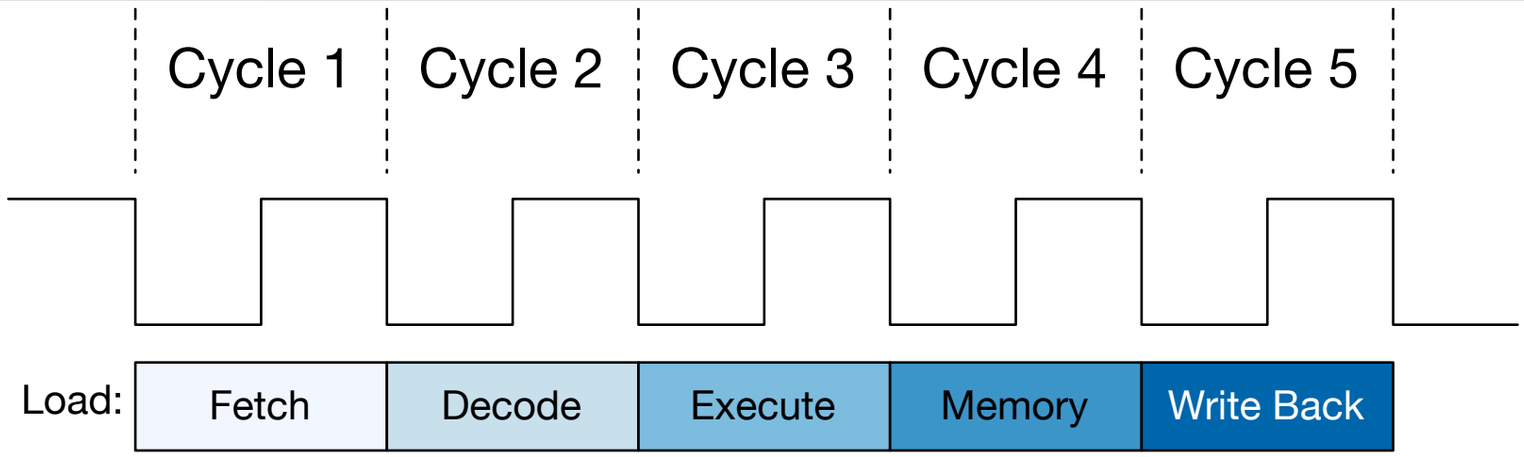
- Speedup correlates to the number of pipe stages

    - Actual speedup reduced by: unbalanced lengths of pipe stages, time to fill pipeline, time to drain pipeline, and stalling due to dependencies

Multi-Cycle Instruction Execution



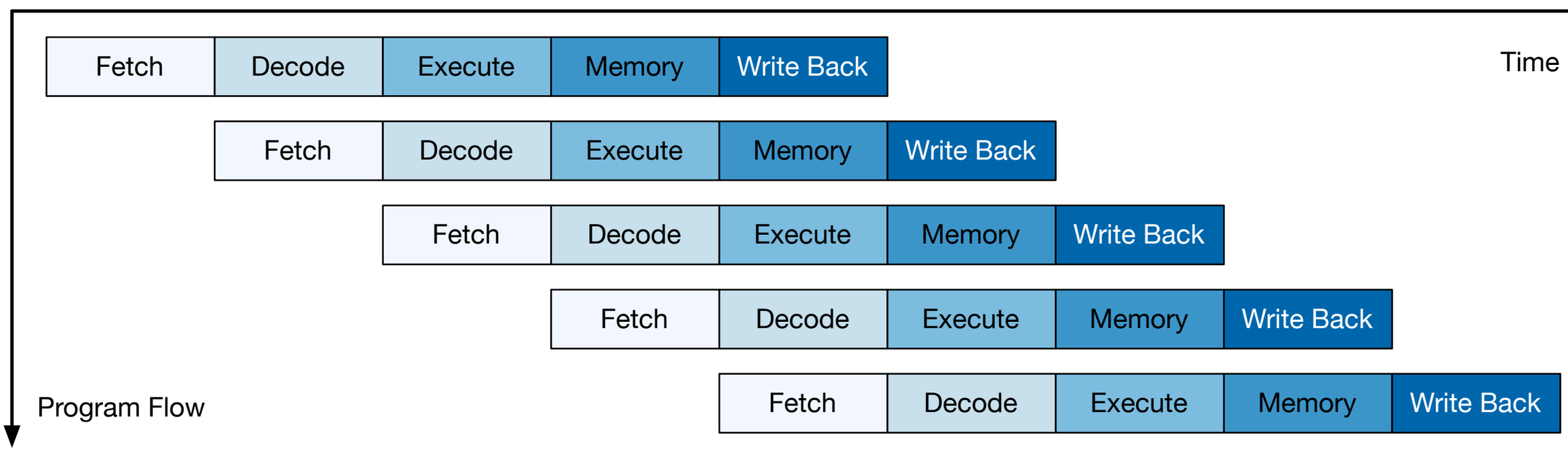
Stages of Instruction Execution



- As mentioned earlier, load instructions take the longest to process

- All instructions follow at most these five stages:
  - Fetch: fetch instruction from Instruction Memory at `PC`
  - Decode: fetch registers and decode instruction
  - Execute: calculate results
  - Memory: read/write data from/to Data Memory
  - Write Back: write data back to register file

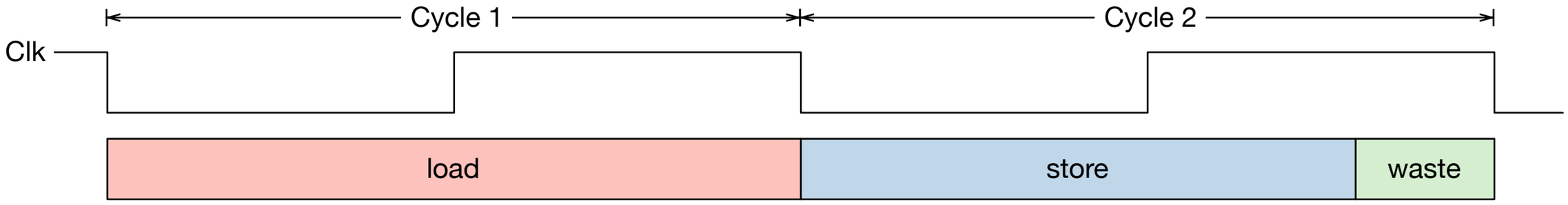
Instruction Pipelining

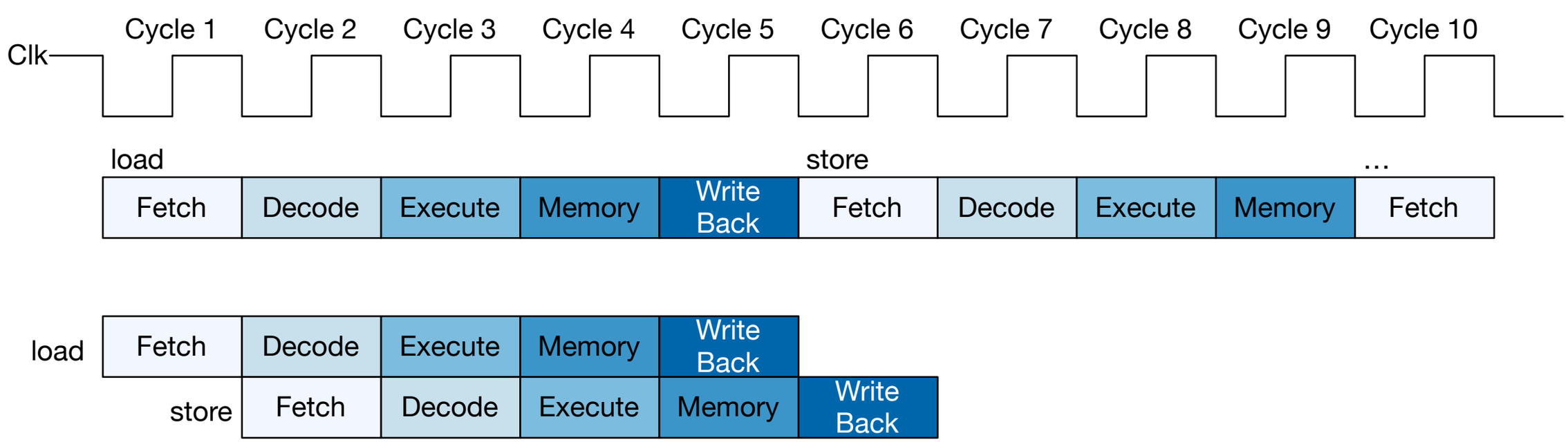


- Start handling next instruction while current instruction is in progress
- Pipelining is feasible when different parts of CPU are used at different stages of instruction execution
- Pipelined instruction throughput = $\frac{non - pipelined\ time}{number\ of\ stages}$

Datapath Comparisons

- Example program flow: a load instruction followed by a store instruction





Example Pipeline Performance

Fetch	Decode	Execute	Memory	Write Back
200 ps	100 ps	200 ps	200 ps	100 ps

- Given the instruction sequence {load, store, R-type}, what is the clock frequency and how long to finish executing all three instructions, for a single-cycle datapath? For a multi-cycle datapath? For a pipelined datapath (ignoring all hazards)?

- How long would it take to execute 1000 consecutive loads for: single-cycle, multi-cycle, and pipeline datapaths?

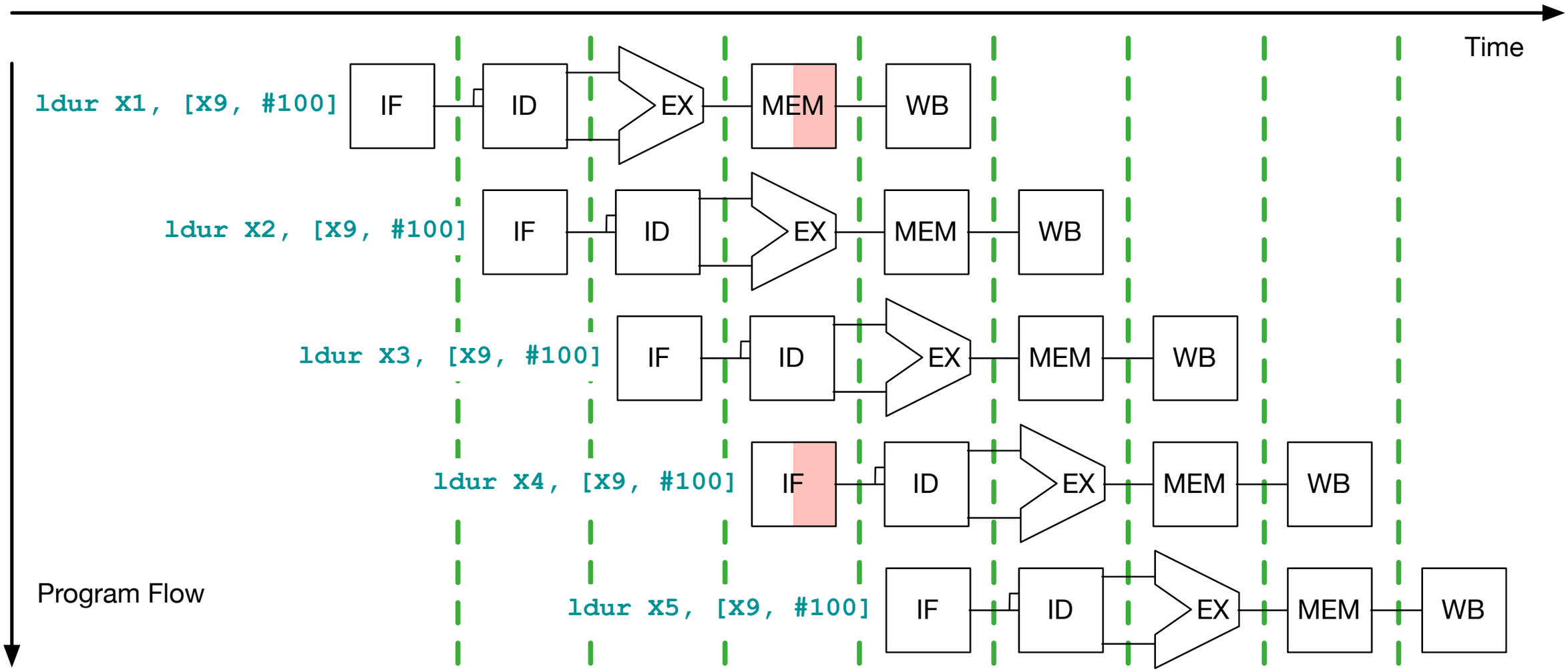
Designing Instruction Sets for Pipelining

- How bits are represented within an instruction affects pipeline performance
- Simplifying instruction fetch:
    - RISC architectures [generally] have same sized instructions
    - CISC architectures have varying length instructions
- Simplifying memory access:
    - ARMv8-A has limited load and store instructions
    - x86-64 allows memory to be used as operands to ALU

Pipeline Hazards

- Situation that prevents next instruction from executing on next clock cycle
- Structural hazard: attempt to use a resource two different ways at same time
  - Example: all-in-one washer/dryer
- Data hazard: attempt to use item before it is ready
  - Example: ready to fold socks, but one sock is still in washer
- Control hazard: attempt to make a decision before condition is evaluated
  - Example: choosing laundry detergent based upon previous load

Structural Hazard



- Combined instruction/data memory can cause conflicting accesses

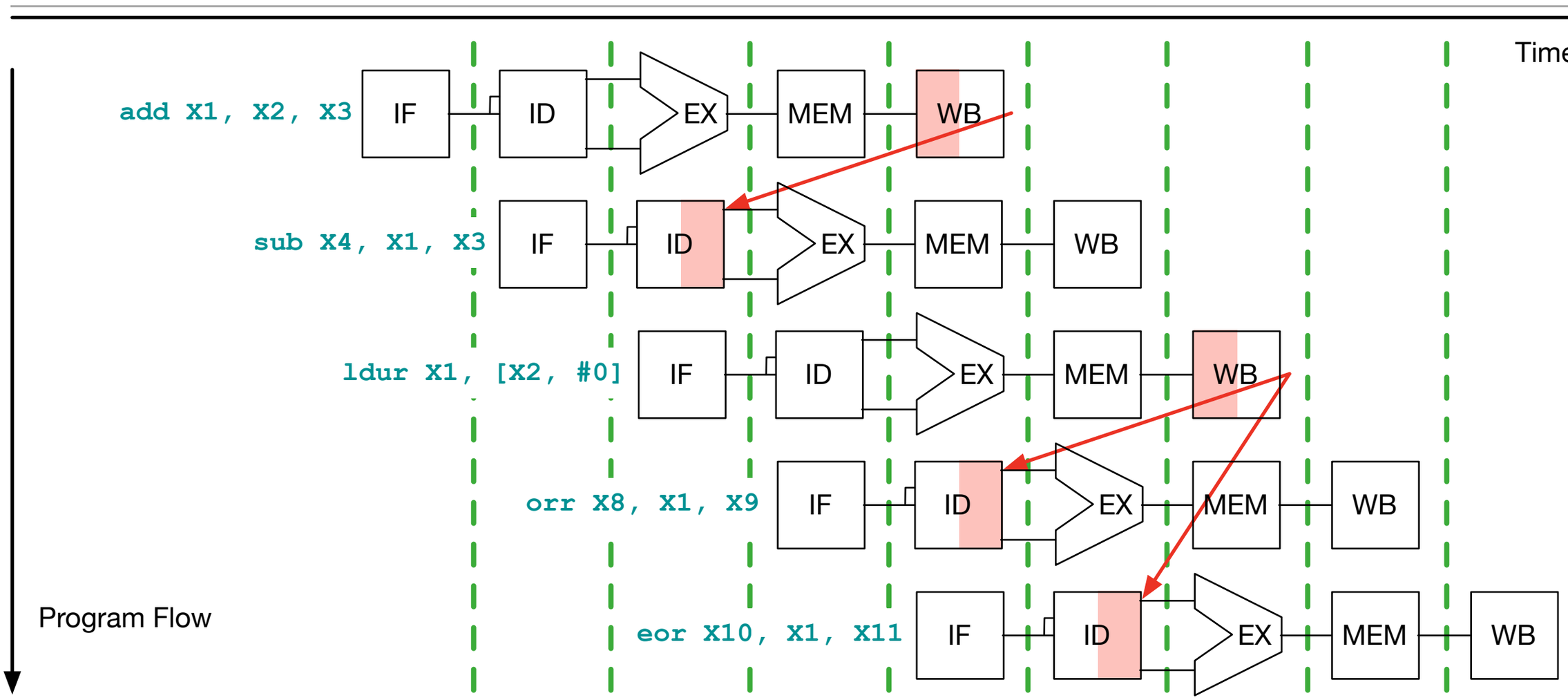
- Can be resolved by adding an idle cycle before fetching fourth **`ldur`**

Memory Architectures

- von Neumann (also known an Princeton) Architecture: single combined memory bus for both data and instructions
    - Simpler to build, allows for self-modifying code, but leads to the von Neumann bottleneck
- Harvard Architecture: separate memory buses for data and instructions
    - Allows parallel access to data and instructions, allows different memory technologies used, but much more complicated to build, prevents self-modifying code
    - Modified Harvard Architecture: has split caches, but unified main memory

http://ithare.com/modified-harvard-architecture-clarifying-confusion/

Data Hazard



- Later instruction is dependent upon previous instruction's execution

- Can be resolved by code reordering, forwarding, or stalling

Code Reordering

- Clever compilers (specifically, code generators) can reorder generated assembly instructions to avoid data hazards

- Example:

```
a = b + e;
c = b + f;
```

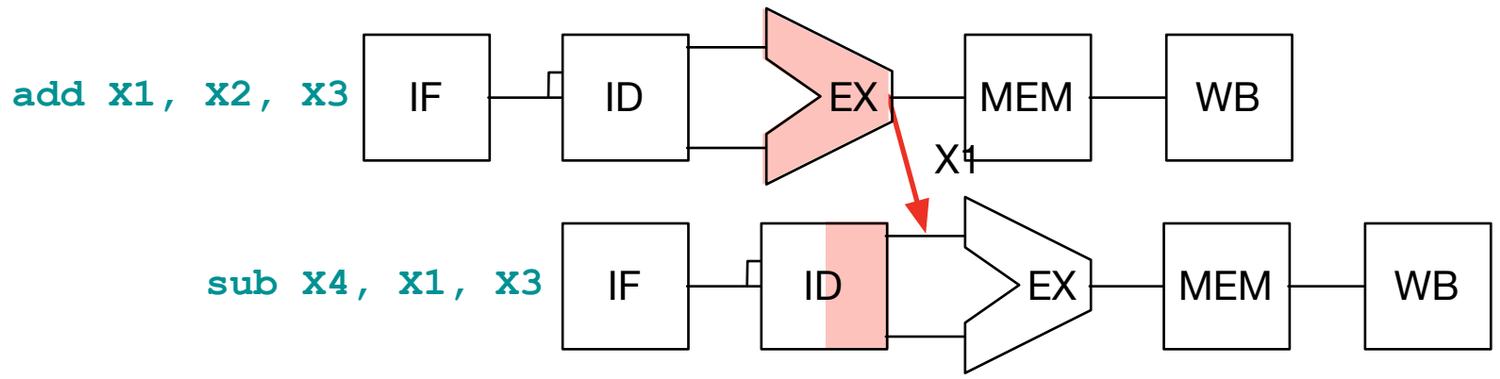
literal, unoptimized

```
ldur X1, [X0, #0]   // load b
ldur X2, [X0, #8]   // load e
add X3, X1, X2      // b + e
ldur X4, [X0, #12]  // load f
add X5, X1, X4      // b + f
```

reordered, optimized

```
ldur X1, [X0, #0]   // load b
ldur X2, [X0, #8]   // load e
ldur X4, [X0, #12]  // load f
add X3, X1, X2      // b + e
add X5, X1, X4      // b + f
```

Forwarding

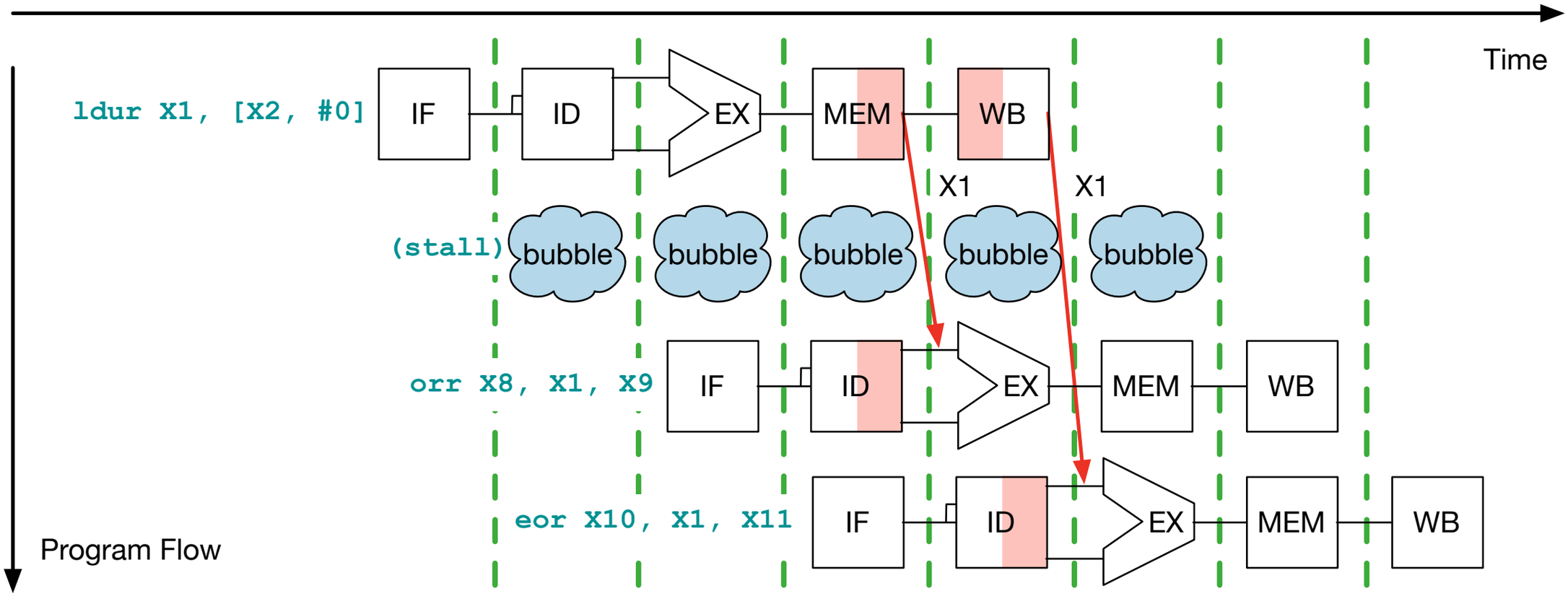


- Add hardware to retrieve missing data from an internal buffer instead of from programmer-visible registers or memory

- Only works for forward paths, later in time

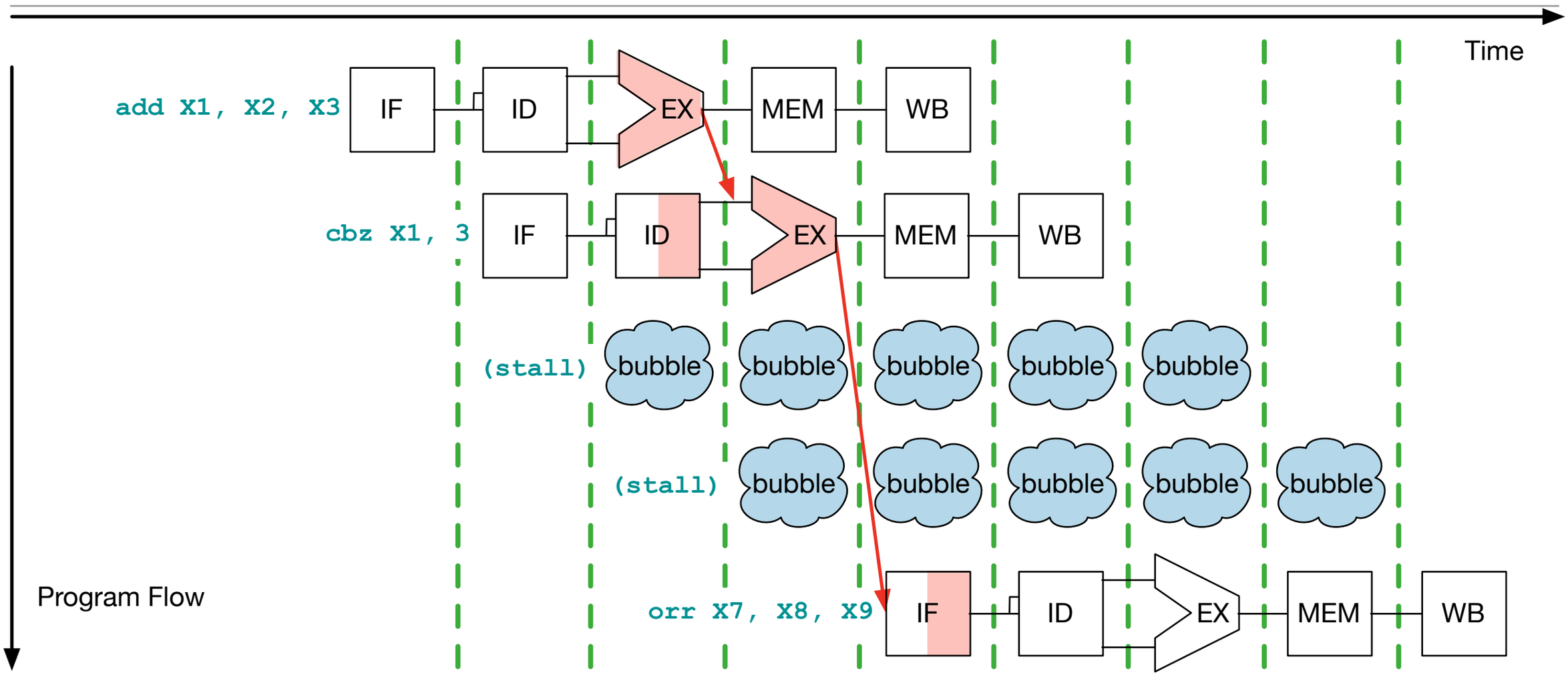
- Does not work for a load immediately followed by an instruction that uses that result (a load-use data hazard)

Stalling



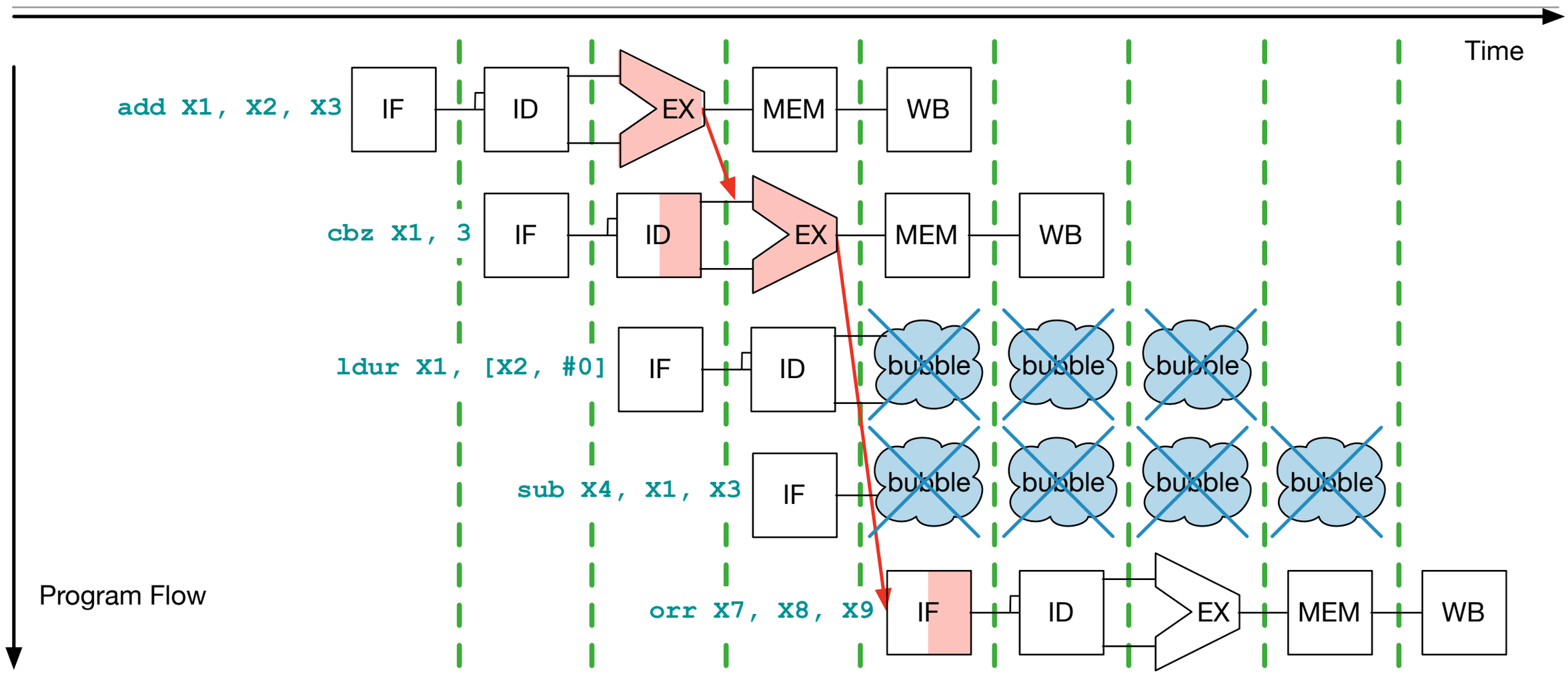
- When code reordering and forwarding is insufficient, then intentionally stall pipeline by adding bubbles
  - Many ways to detect when stalling is needed and how many bubbles to induce

Control Hazard



- Upon branching, the PC for the next instructor is unknown until after decode (for unconditional branches) or after execution (for conditional branches)
  - One solution is to always induce stall(s) when a branch instruction is detected, until after branch is resolved

Branch Prediction



- In simple case, assume that branch will never be taken

- If branch is taken, then flush pipeline, restarting with correct instruction