CSE 410
Computer Systems

Hal Perkins

Spring 2010

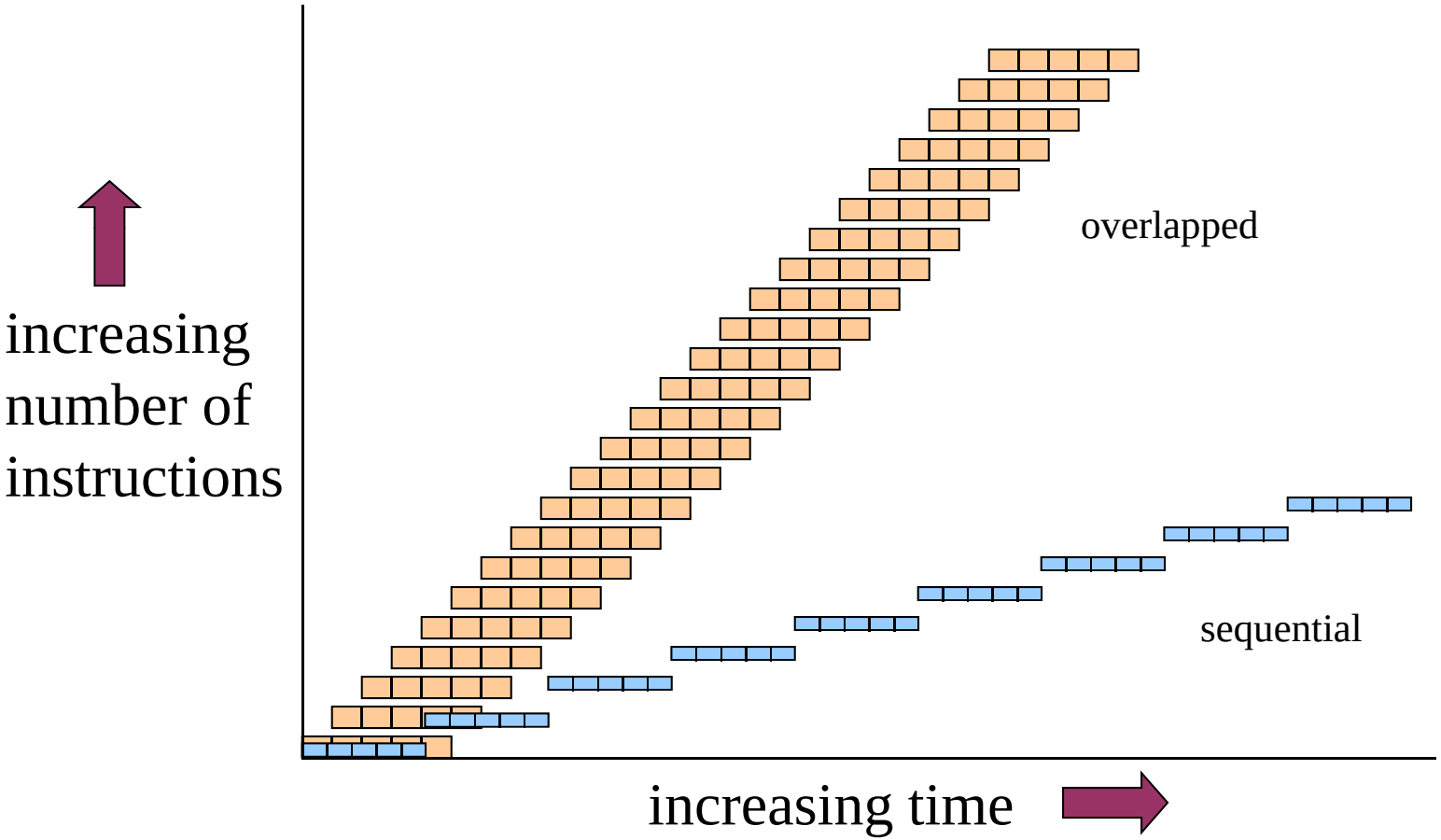
Lecture 10 – Pipelining II

Reading

- Computer Organization and Design
  - Sec. 4.5, Overview of Pipelining
  - Sec. 4.6, Pipeline Datapath and Control, pp. 356-7
  - Sec. 4.7, Data Hazards, pp. 363-367
  - Sec. 4.8, Control Hazards, pp. 375-377, 380-384

  Skim the hardware details, but get the basic ideas of how pipelining and hazards affect instruction execution. Feel free to read the details if you're interested and ask questions offline.

Pipelining Goal: Better Throughput



Can we get that much speedup?

- Any time you get several things going at once, you run the risk of interactions and dependencies
- Unwinding activities after they have started can be very costly in terms of performance
  - drop everything on the floor and start over

Pipeline Hazards

- Structural hazards
  - Instructions in different stages need the same resource, eg, memory controller, arithmetic unit
- Data hazards
  - data not available to perform next operation when needed
- Control hazards
  - data not available to make branch decision soon enough

Structural Hazards

- Example 1: two instructions need the same resource
  - lw instruction in stage four (memory access)
  - another instruction in stage one (instruction fetch)
  - both of these actions require access to memory
- Solution: add more hardware to eliminate problem
  - separate instruction and data memory caches or memory unit ports
- Or stall – do things one at a time
  - Not a good idea, particularly on memory access

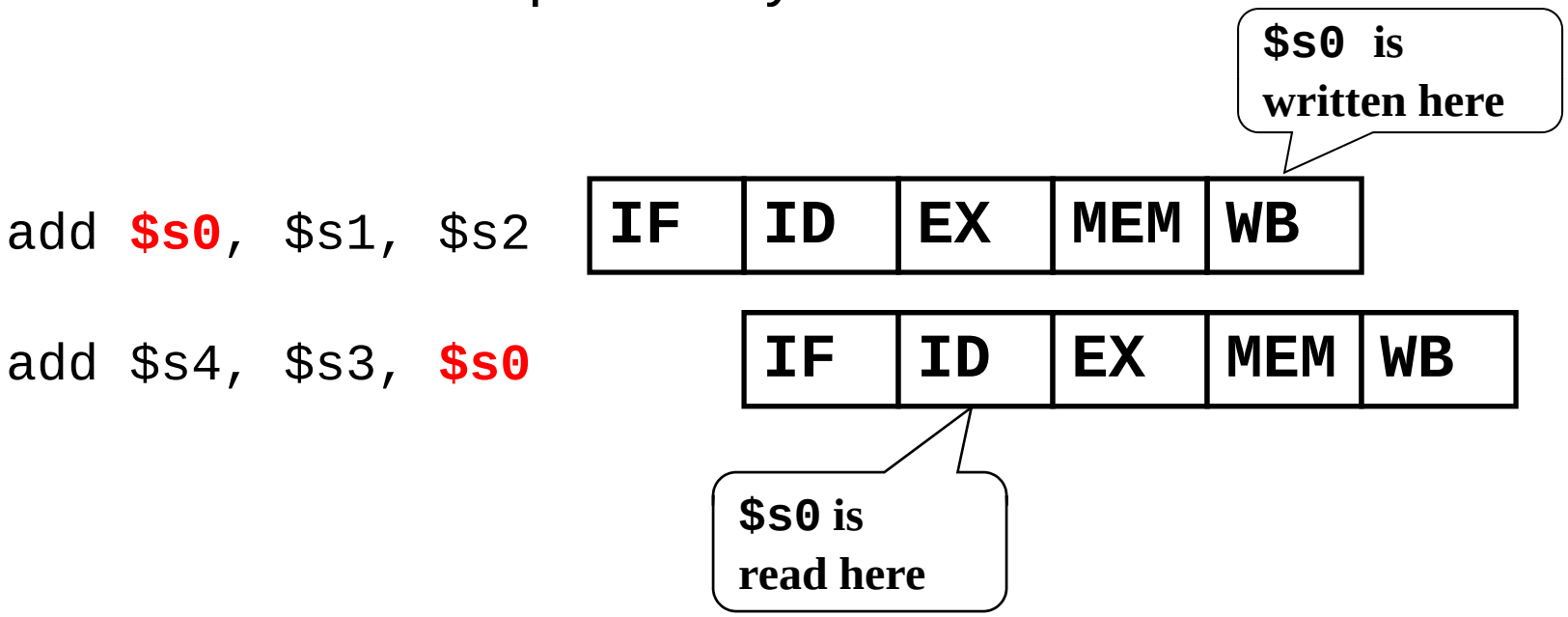
Structural Hazards

- Another example: need to do two additions at once:
  - An add instruction in stage 3 adding two registers
  - Another instruction in stage 1 needs to compute PC+4
- Solution: throw more hardware at the problem
  - Multiple adders in different parts of the CPU

- We'll assume sufficient hardware provided so we don't have to deal with structural hazards

Data Hazards

- When an instruction depends on the results of a previous instruction still in the pipeline
- This is a data dependency

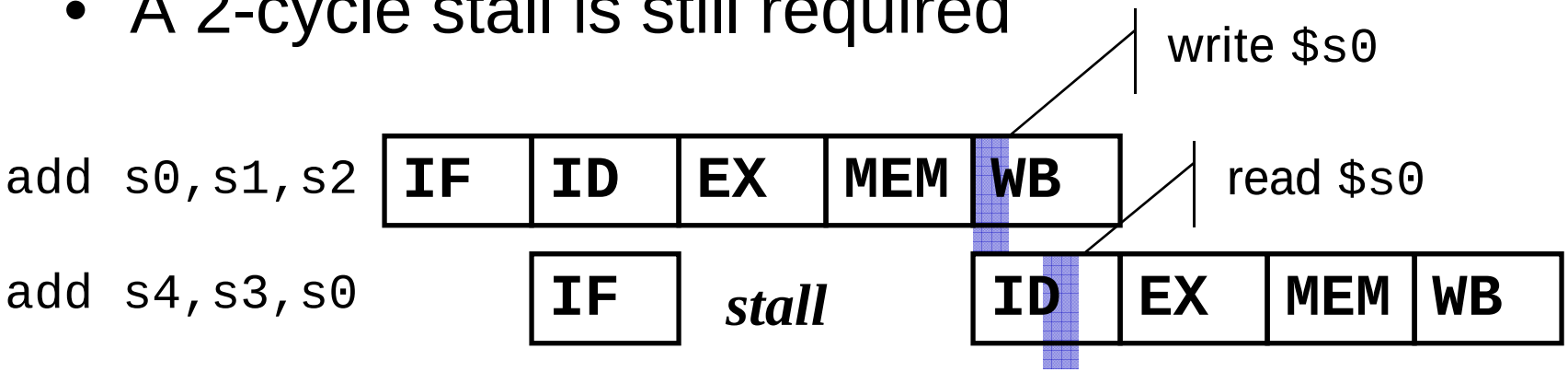


Stall for register data dependency

- One solution: Stall the pipeline until the result is available
  - this would create a 3-cycle *pipeline bubble*

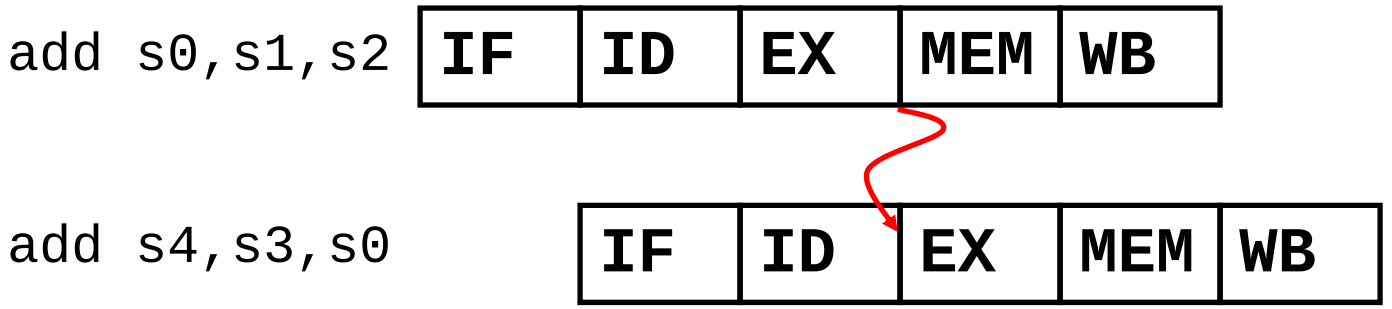
Better: Read & Write in Same Cycle

- Write the register in the first part of the clock cycle
- Read it in the second part of the clock cycle
- Real register files are built like this
- A 2-cycle stall is still required



Solution: Forwarding

- The value of $s0 is known internally after cycle 3 (after the first instruction's EX stage)
- The value of $s0 isn't needed until cycle 4 (before the second instruction's EX stage)
- If we **forward** the result there isn't a stall

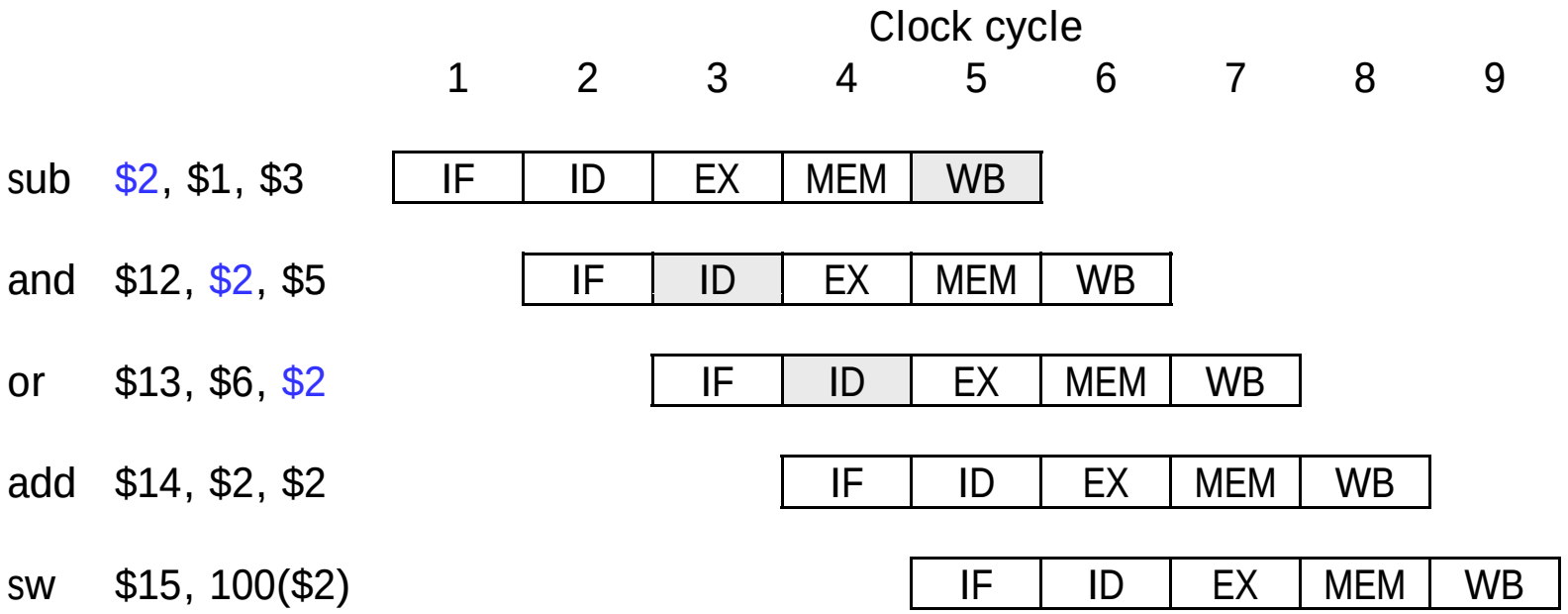


Another example with dependencies

```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

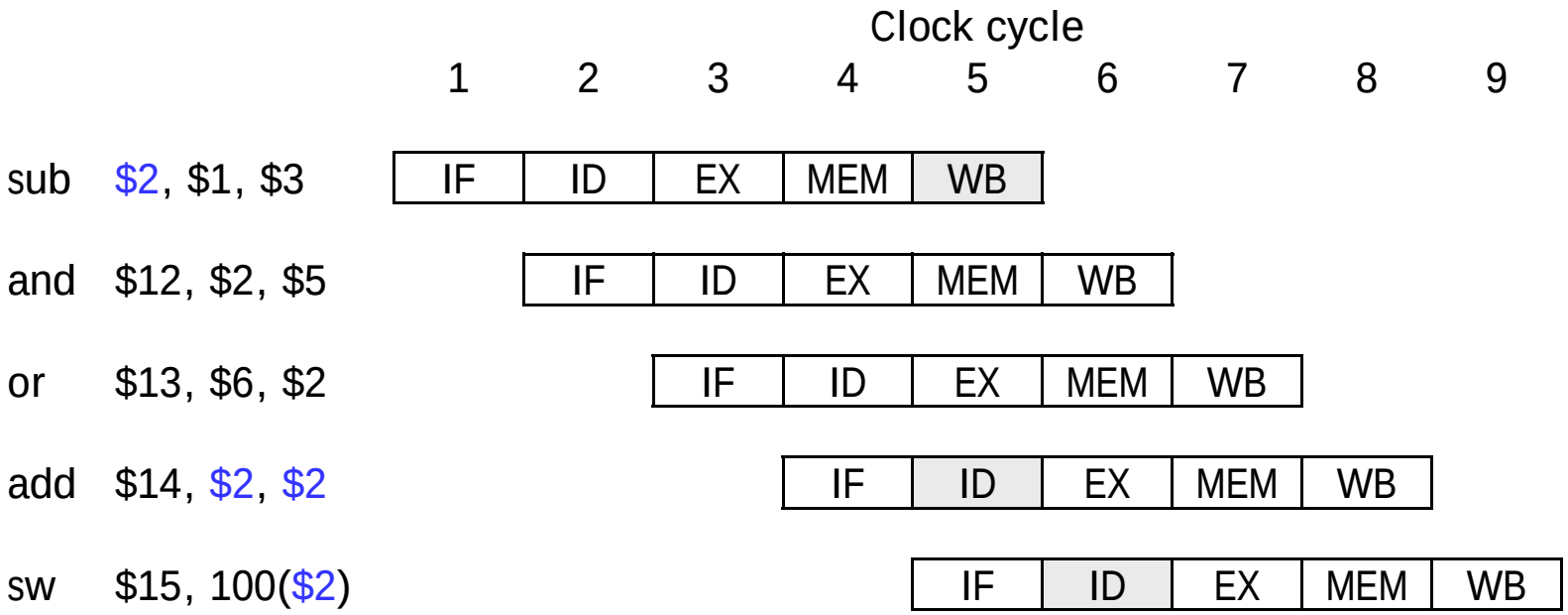
How would this code sequence fare in our pipelined datapath without forwarding?

Data hazards in the pipeline diagram



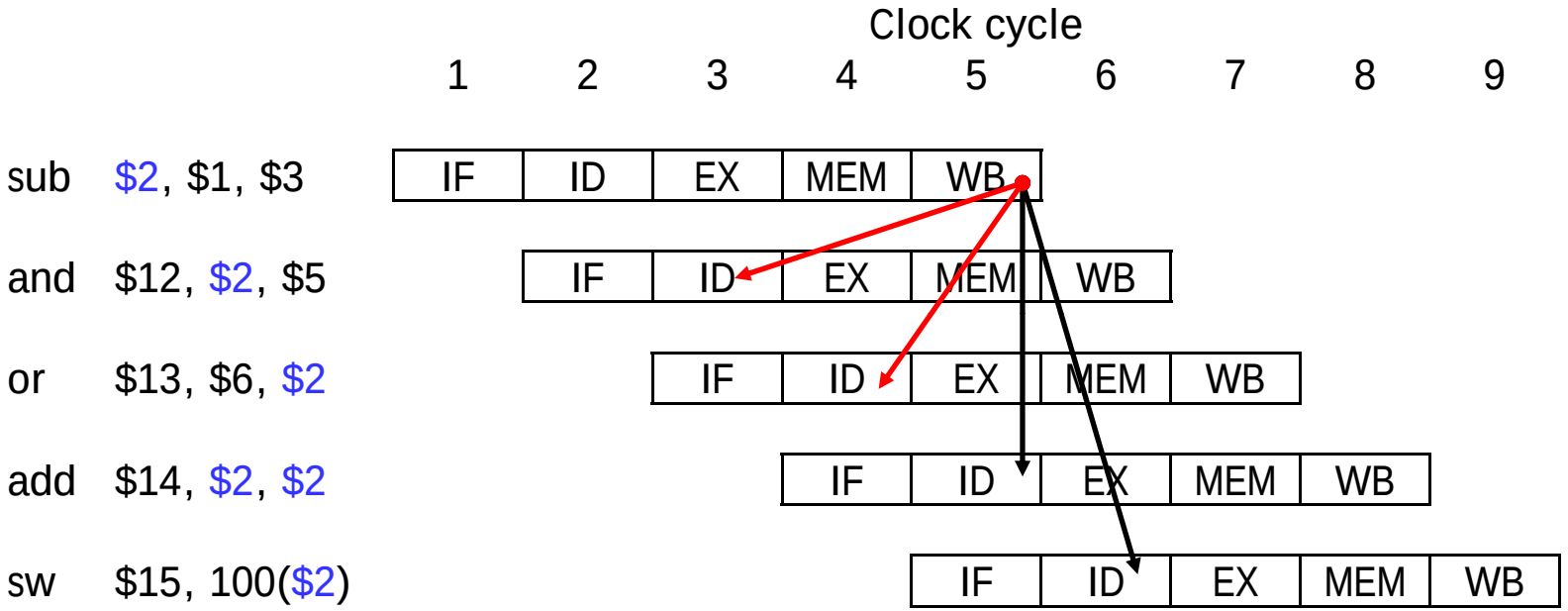
- The SUB instruction does not write to register $2 until clock cycle 5. This causes two data hazards in the pipelined datapath.
  - The AND reads register $2 in cycle 3. Since SUB hasn't modified the register yet, this will be the *old* value of $2, not the new one.
  - Similarly, the OR instruction uses register $2 in cycle 4, again before it's actually updated by SUB.

Things that are okay



- The ADD instruction is okay, because of the register file design.
  - Registers are written at the beginning of a clock cycle.
  - The new value will be available by the end of that cycle.
- The SW is no problem at all, since it reads $2 after the SUB finishes.

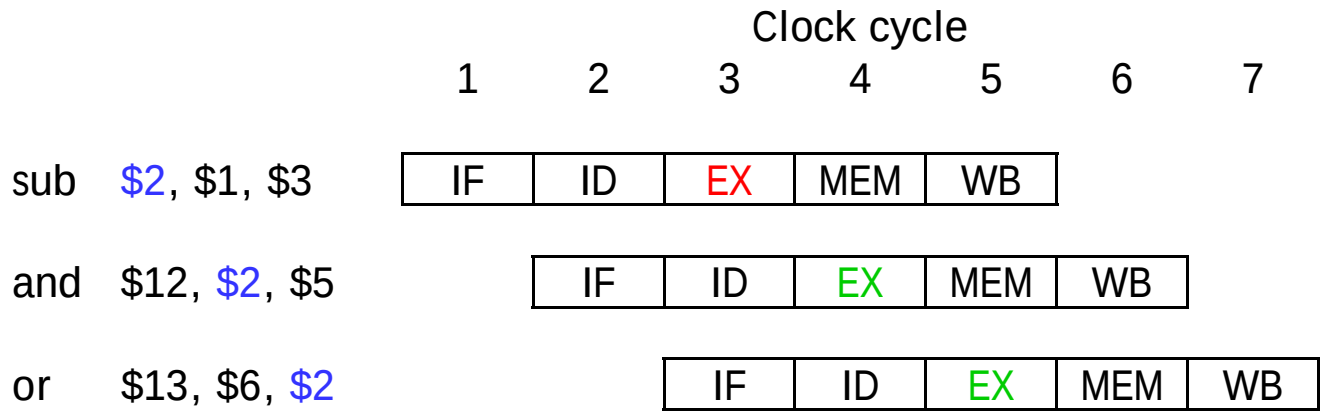
Dependency arrows



- Arrows indicate the flow of data between instructions.
  - The tails of the arrows show when register $2 is written.
  - The heads of the arrows show when $2 is read.
- Any arrow that points backwards in time represents a data hazard in our basic pipelined processor. Here, hazards exist between instructions 1 & 2 and 1 & 3.

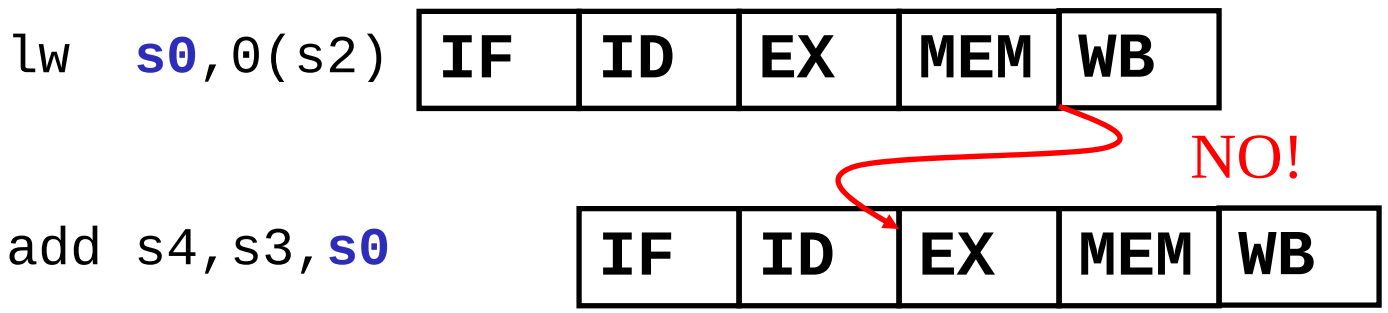
A more detailed look at the pipeline

- We have to eliminate the hazards, so the AND and OR instructions will use the correct value for register $2.
  - The SUB instruction produces its result in its EX stage, during cycle 3 in the diagram below.
  - The AND and OR need the new value of $2 in their EX stages, during clock cycles 4-5 here.
  - We can add forwarding hardware to solve both of these problems



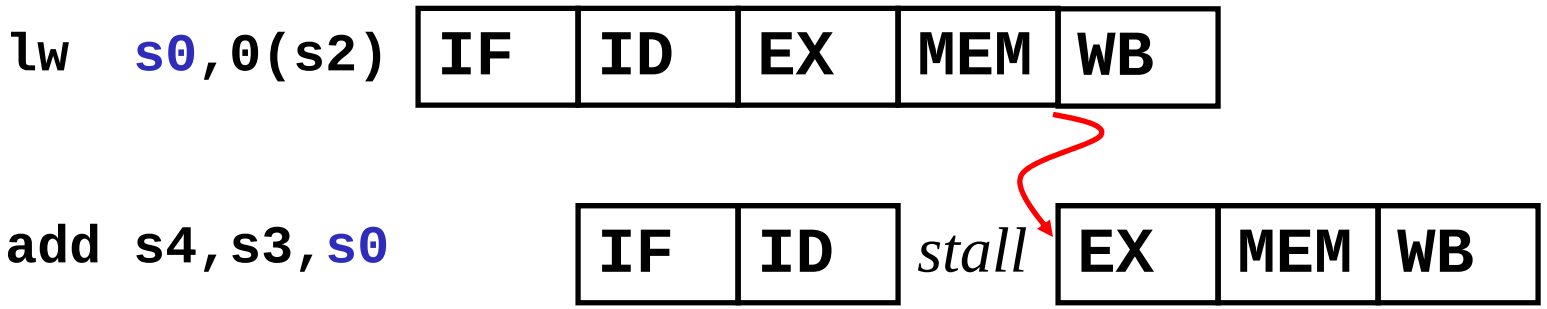
What about loads?

- What if the first instruction is lw?
- This is a true data hazard: s0 isn't known until after the MEM stage
  - We can't forward back into the past
- Either **stall** or **reorder** instructions



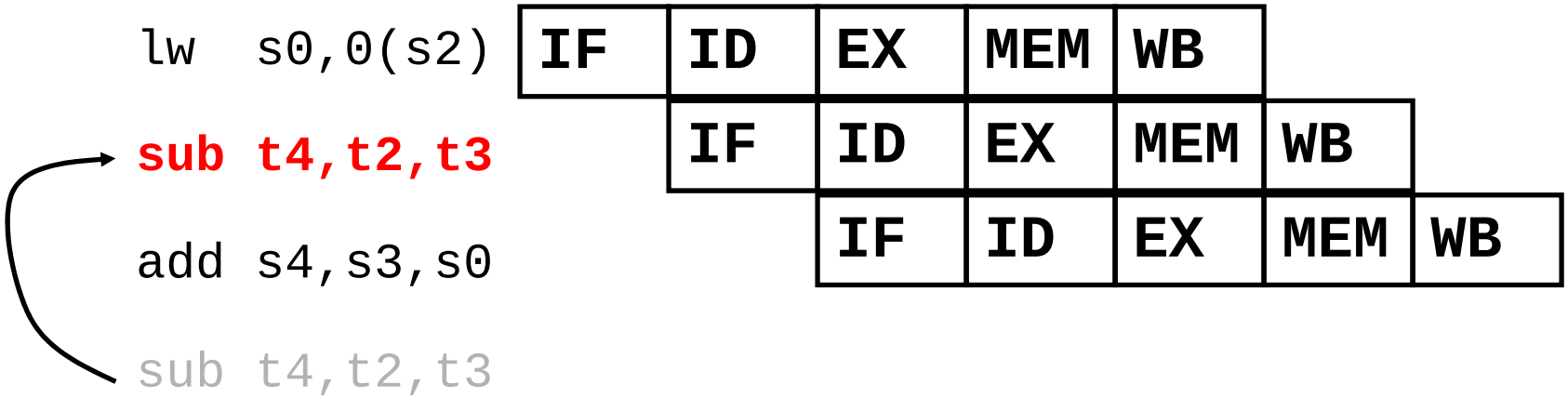
Stall for `lw` hazard

- We can stall for one cycle and then forward
  – but can we avoid the stall?



Instruction Reorder for `lw` hazard

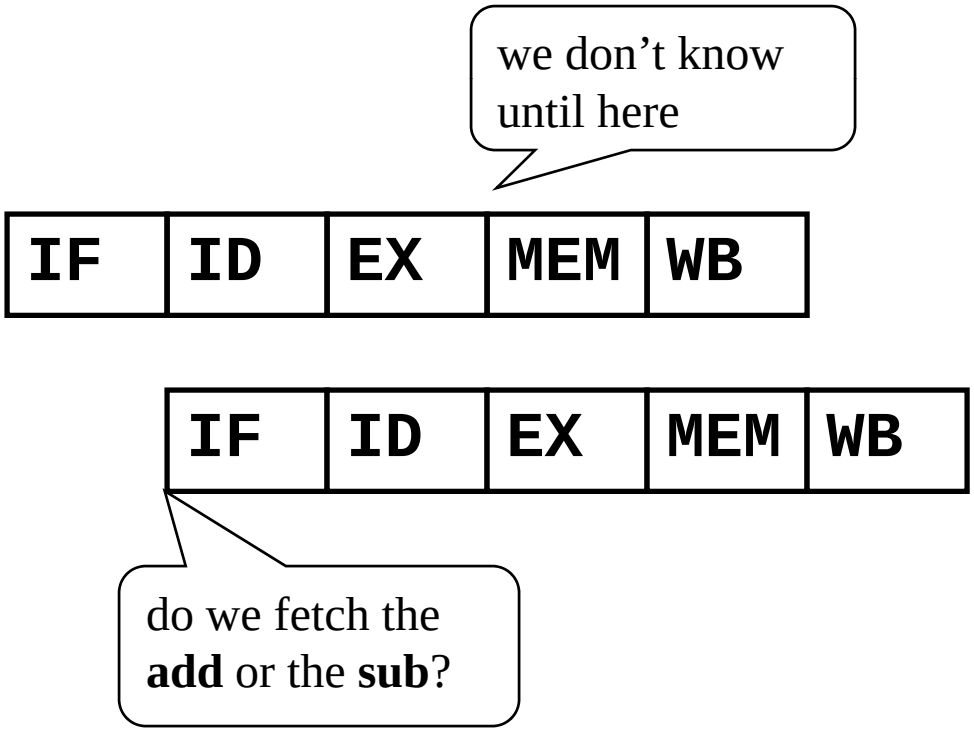
- If some unrelated instruction can be moved, rearrange the order to keep the pipeline busy and mask the stall
  - This is called *instruction scheduling* and all good compilers do it to help out the hardware
  - Many processors do this dynamically during execution even if the compiler doesn't reorder



Control Hazards

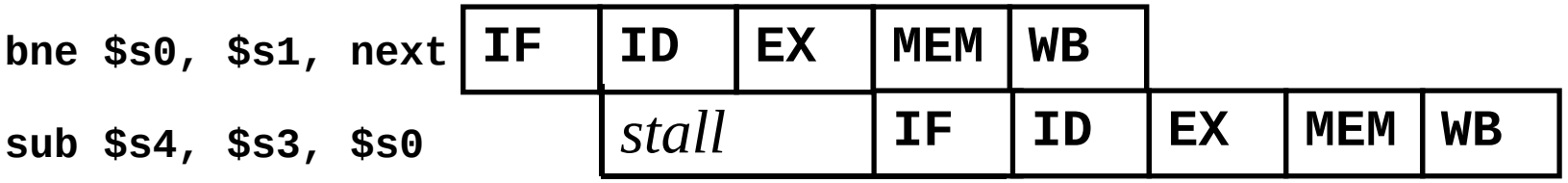
- Branch instructions cause *control hazards* because we don't know which instruction to fetch next

```
  bne $s0, $s1, skip
  add $s4, $s3, $s0
  ...
skip:
  sub $s4, $s3, $s0
```



One solution: Stall

- Stall until we know which instruction to execute next
  - We know the result at the end of the EX state (or earlier if we compare the registers in the ID stage)
  - Still causes a 1 or 2 cycle stall even with forwarding
  - We can't completely avoid the stall unless we know if the branch will be taken



Branch Prediction

- Make a guess! Assume the branch will not be taken
  - Simple to implement: keep fetching sequentially
- If we guessed right, all is well!
  - no bubble at all
- If we guessed wrong, then we lose a little:
  - need to flush the partially completed instructions
    - i.e., an instruction that should not be executed *must not* have any permanent effect
    - i.e., can't write to memory or a register until we know the instruction will actually execute
  - Wasted time, but would have stalled anyway

Branch Prediction

- Static prediction does ok
  - Predict backwards branches taken (e.g., loops)
  - Predict other branches not taken (right ≈ 1/2 time)
- Modern processors use dynamic branch prediction
  - Record history of branches
  - Predict based on history
    - If branch consistently taken/not taken, predict that in the future (but don't change your mind after a single misprediction)
    - More elaborate dynamic schemes in modern processors

Summary

- Three kinds of hazards conspire to make pipelining difficult.
- Structural hazards result from not having enough hardware available to execute multiple instructions simultaneously.
  - These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline.
- Data hazards can occur when instructions need to access registers that haven't been updated yet.
  - Hazards from R-type instructions can be avoided with forwarding.
  - Loads can result in a "true" hazard, which must stall the pipeline.
- Control hazards arise when the CPU cannot determine which instruction to fetch next.
  - We can minimize delays by doing branch tests earlier in the pipeline.
  - We can also take a chance and predict the branch direction, to make the most of a bad situation.