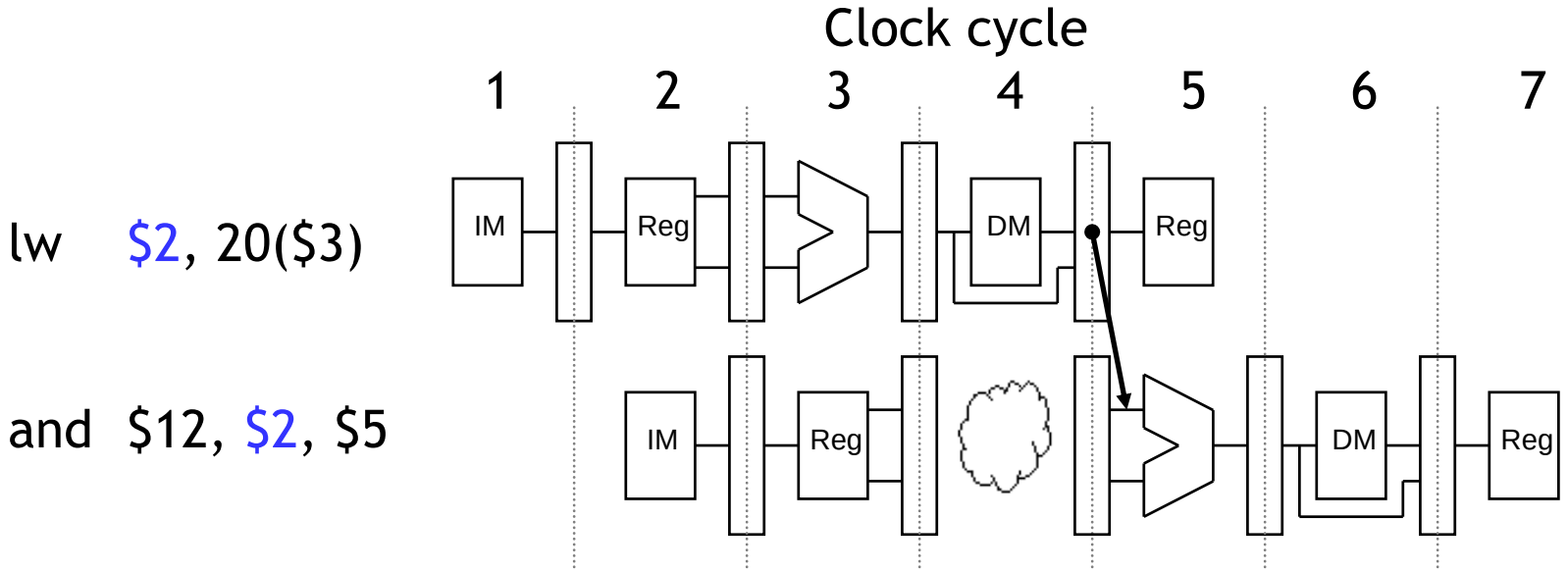
Stalling

- The easiest solution is to stall the pipeline

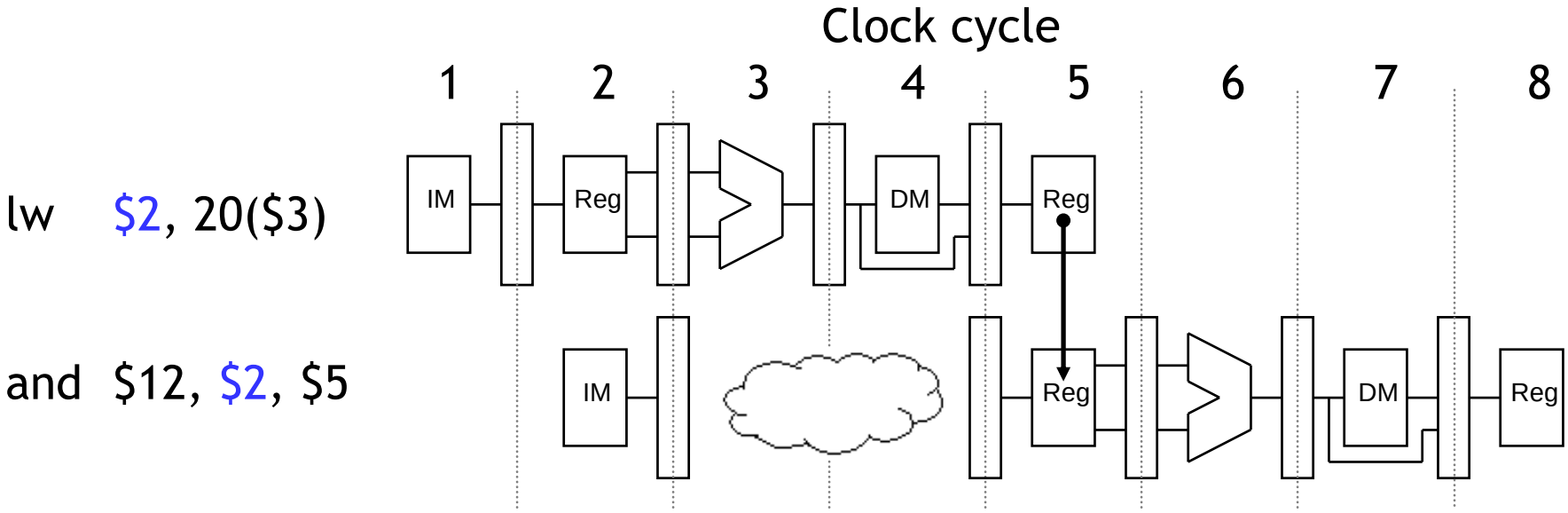
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a bubble



- Notice that we're still using forwarding in cycle 5, to get data from the MEM/WB pipeline register to the ALU

Stalling and forwarding

- Without forwarding, we'd have to stall for *two* cycles to wait for the LW instruction's writeback stage



- In general, you can always stall to avoid hazards—but dependencies are very common in real code, and stalling often can reduce performance by a significant amount

Load-Use Hazard Detection

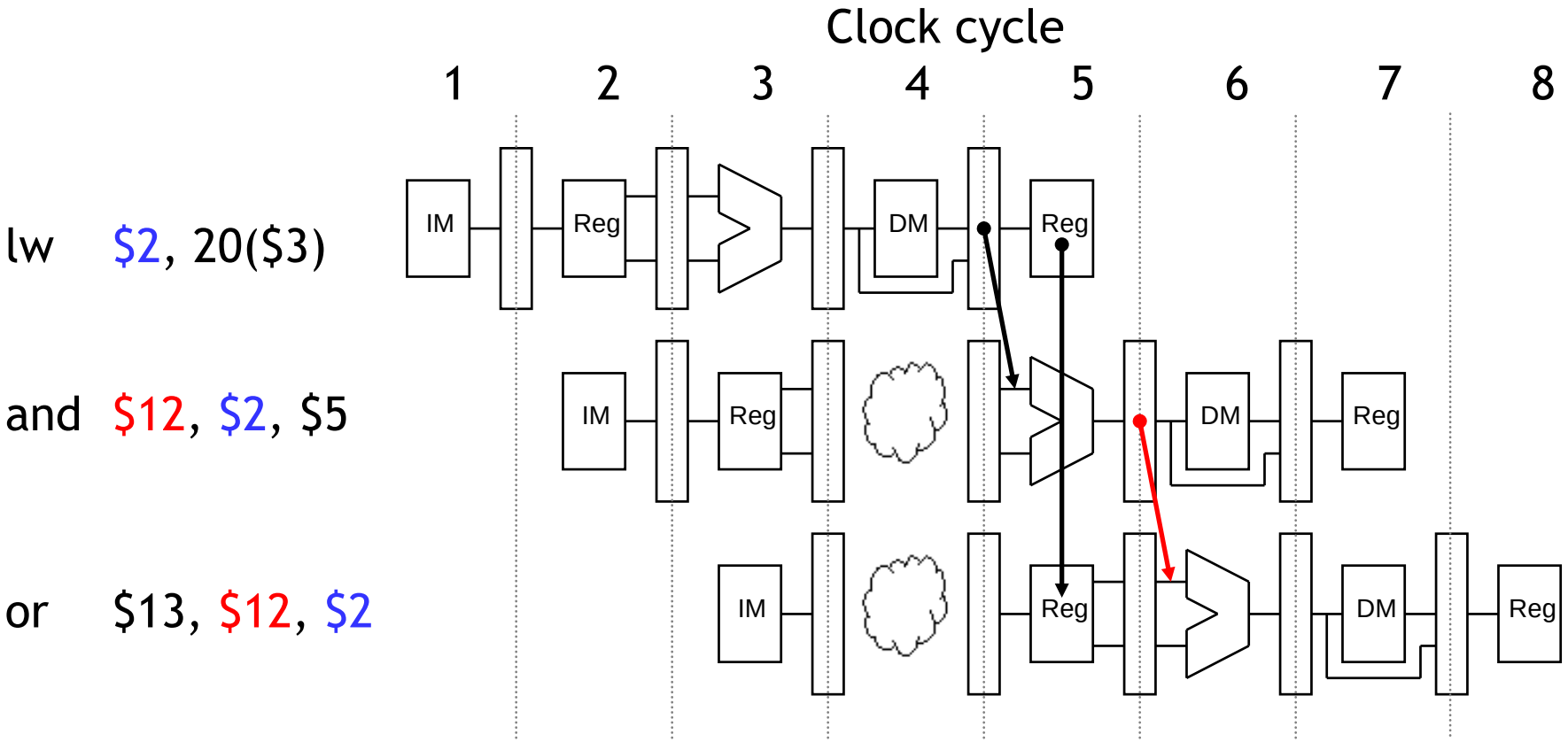
- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble

How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for `lw`
    - Can subsequently forward to EX stage

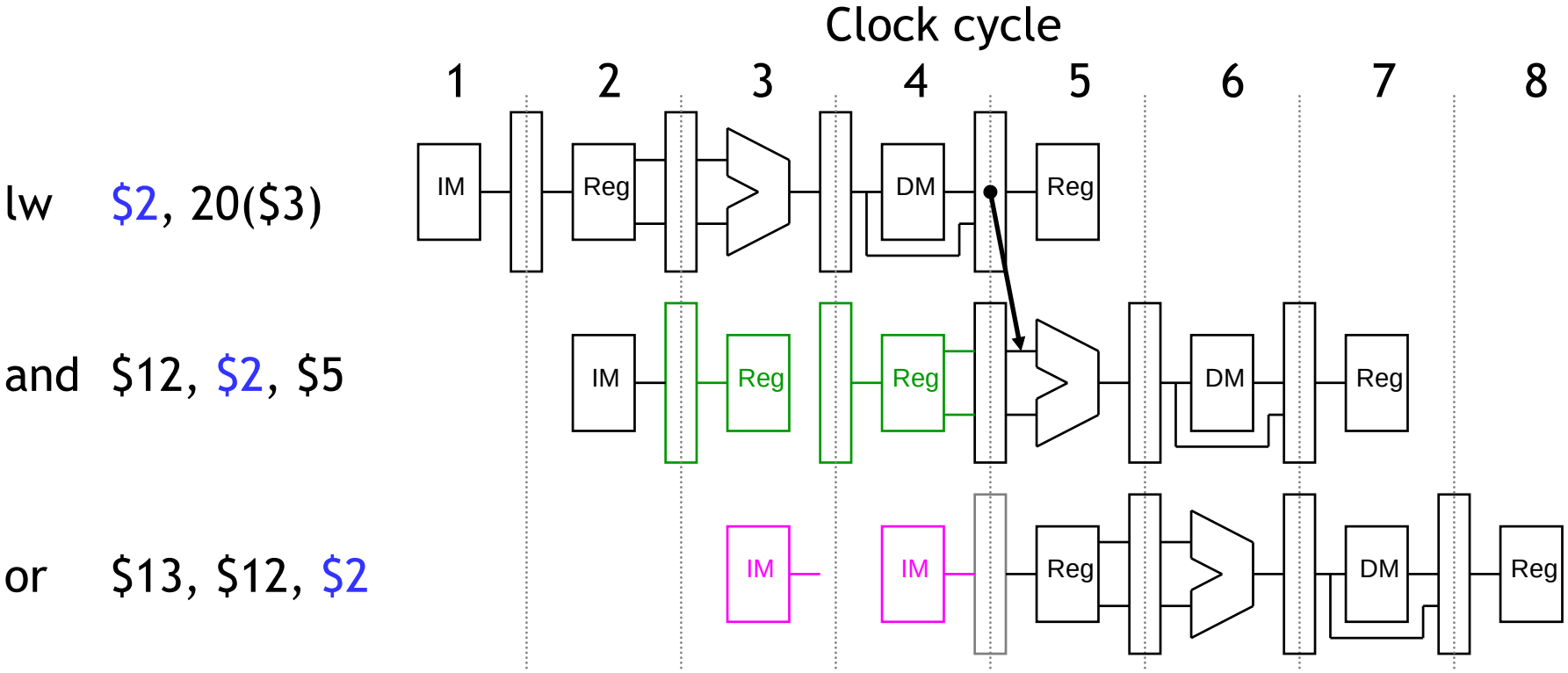
Stalling delays the entire pipeline

- If we delay the second instruction, we'll have to delay the third one too
  - This is necessary to make forwarding work between AND and OR
  - It also prevents problems such as two instructions trying to write to the same register in the same cycle



What about EX, MEM, WB

- But what about the ALU during cycle 4, the data memory in cycle 5, and the register file write in cycle 6?



- Those units aren't used in those cycles because of the stall, so we can set the EX, MEM and WB control signals to all 0s.

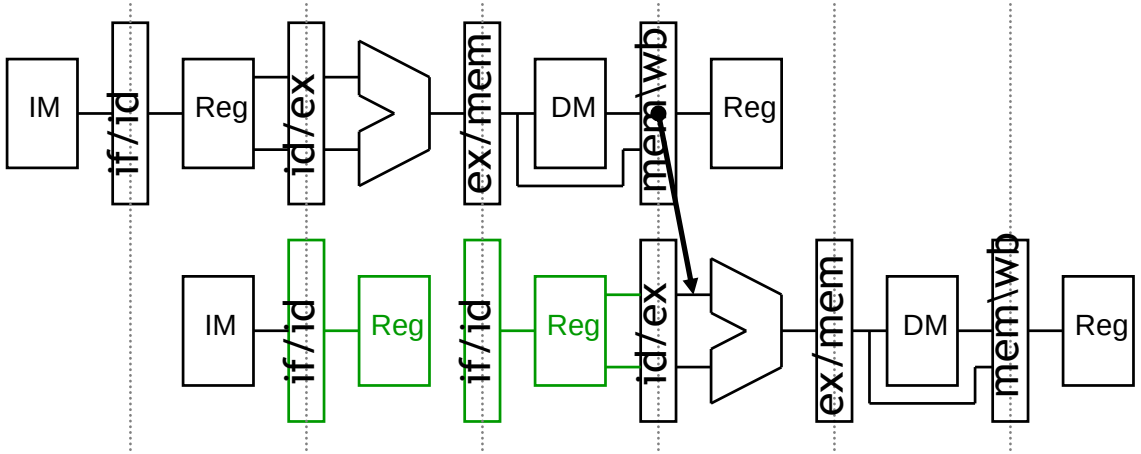
Detecting Stalls, cont.

- When should **stalls** be detected?

  EX stage (of the instruction causing the stall)

lw $2, 20($3)

and $12, $2, $5

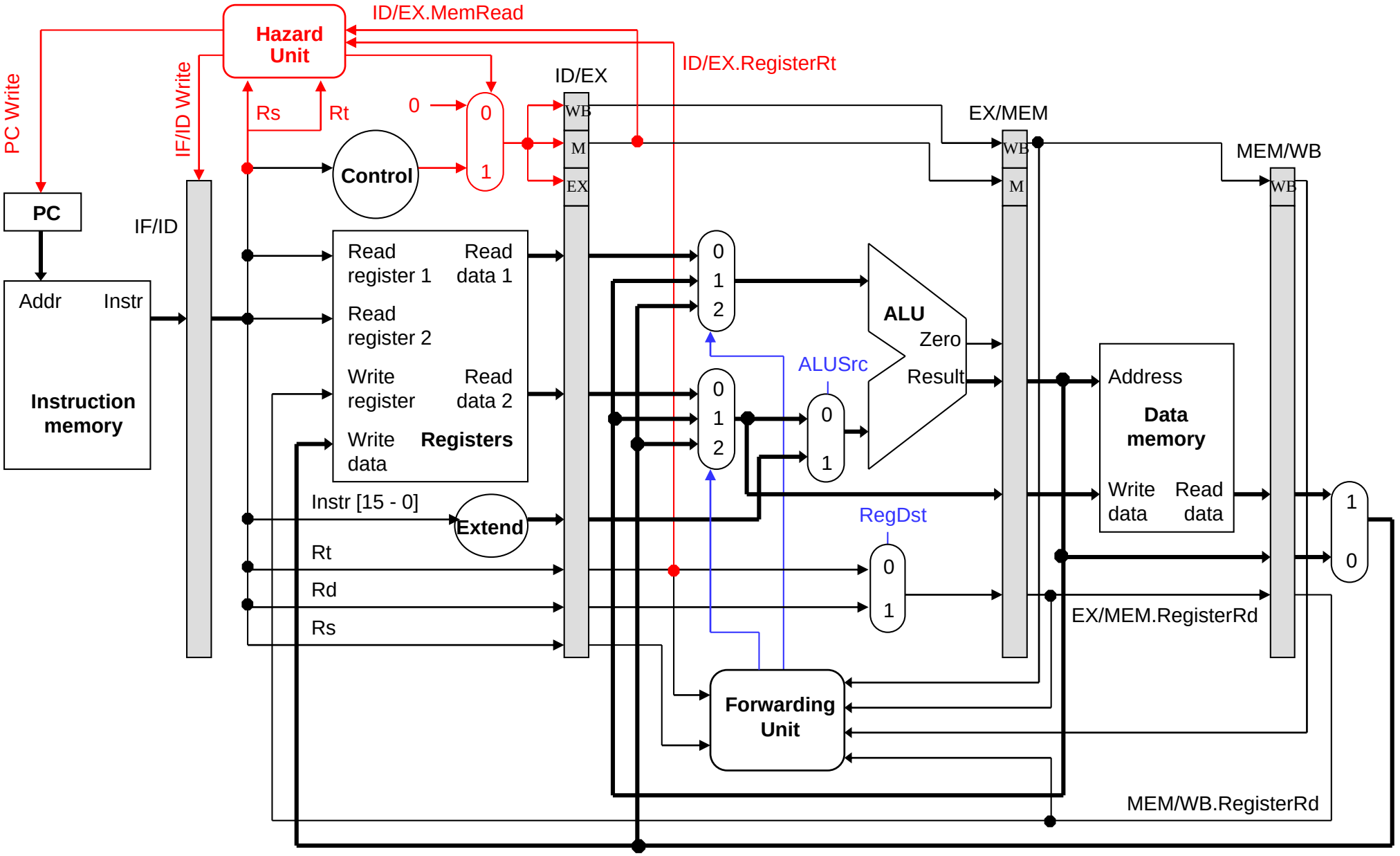


- What is the stall condition?

  if (ID/EX.MemRead = 1 and (ID/EX.rt = IF/ID.rs or ID/EX.rt = IF/ID.rt))

  then stall

Adding hazard detection to the CPU



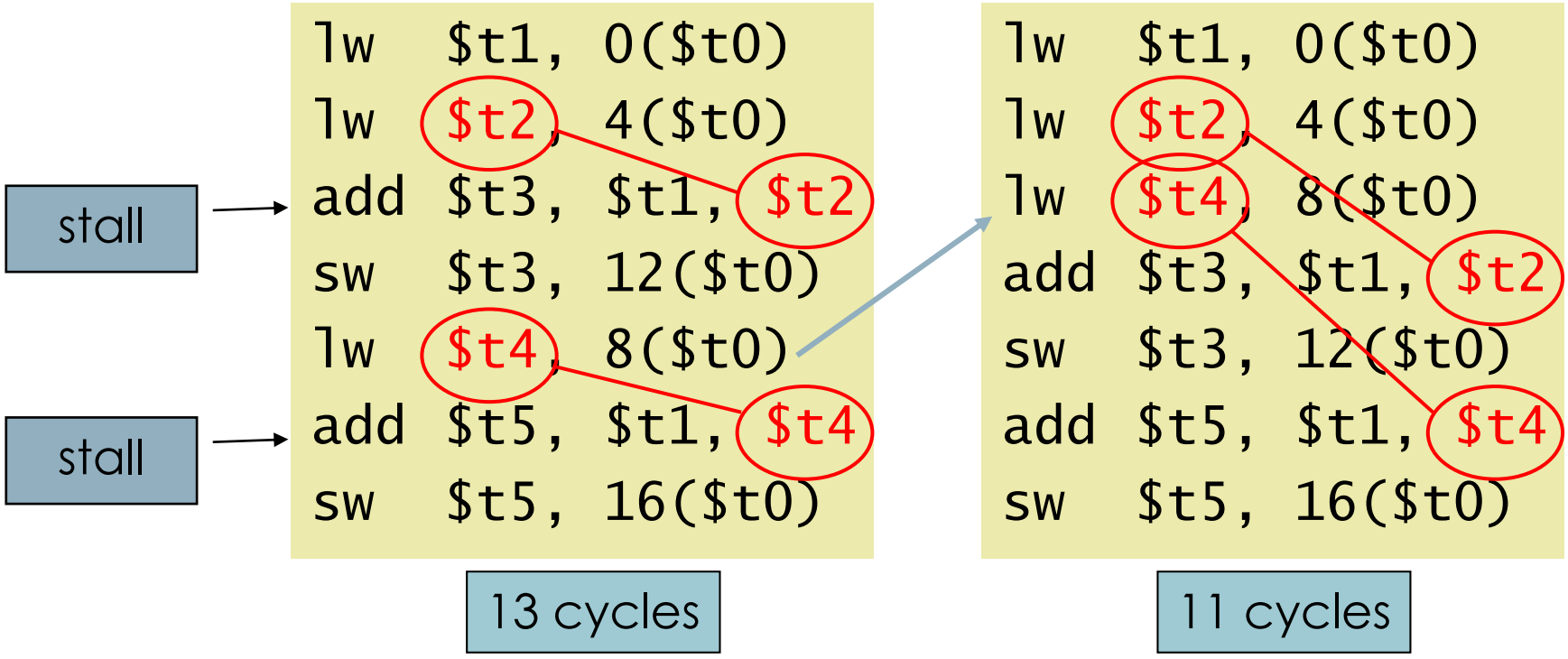
Stalls and Performance

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

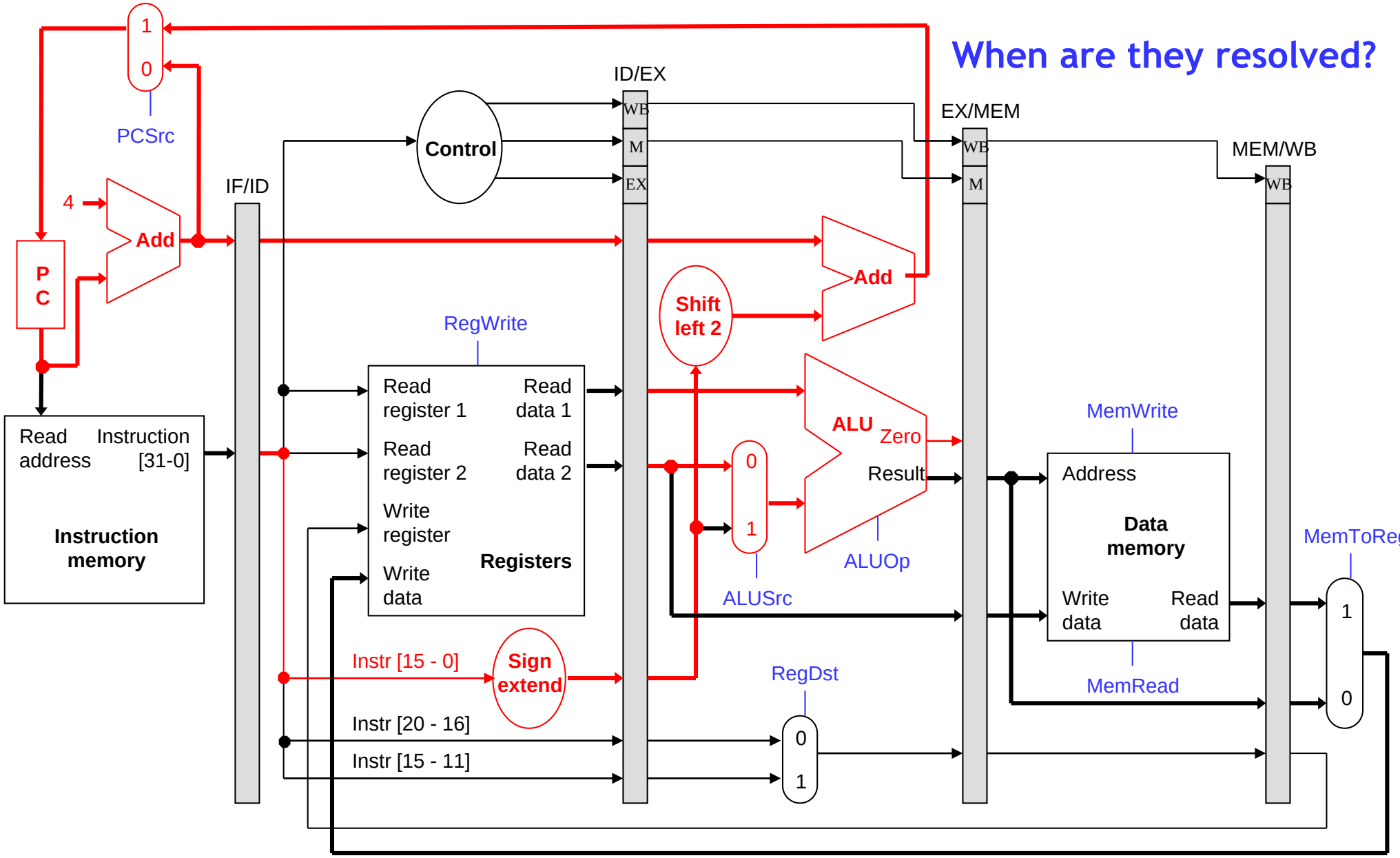
Code Scheduling to Avoid Stalls

Reorder code to avoid use of load result in the next instruction

Ex: c code for `A = B + E; C = B + F;`

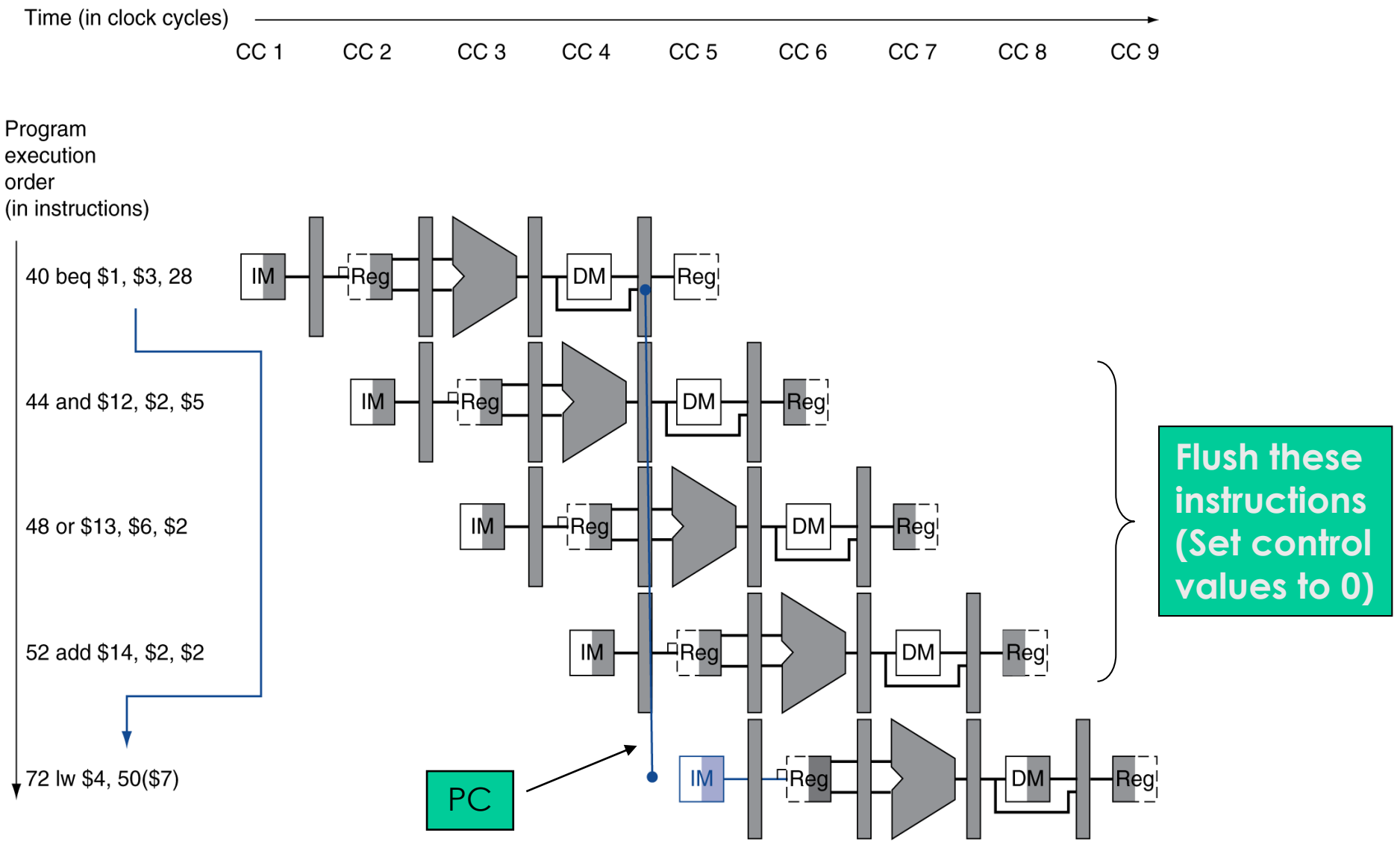


Branches in the original pipelined datapath



Branch Hazards

If branch outcome determined in MEM:



Reducing Branch Delay

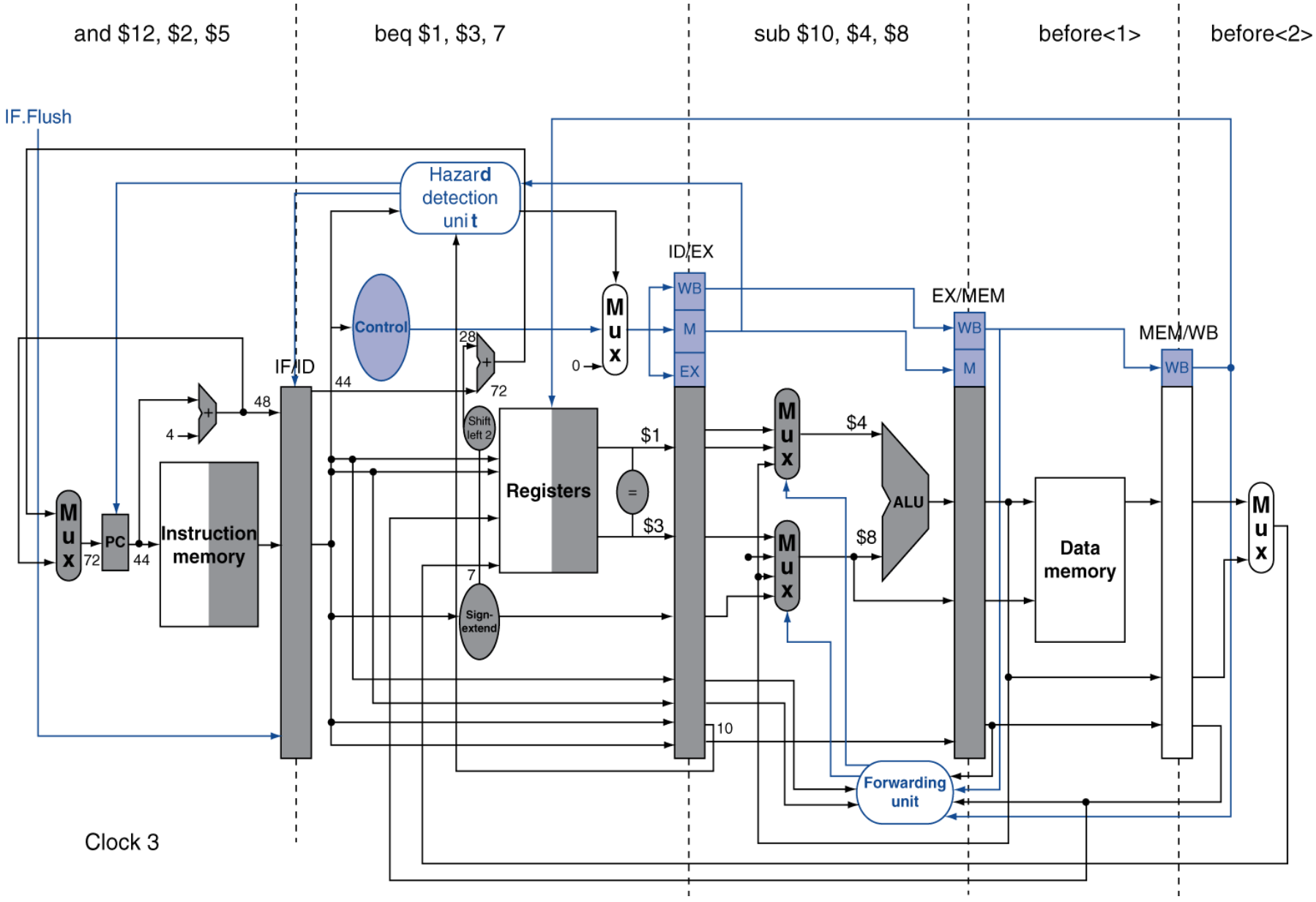
Move hardware to determine outcome to ID stage

- Target address adder
- Register comparator

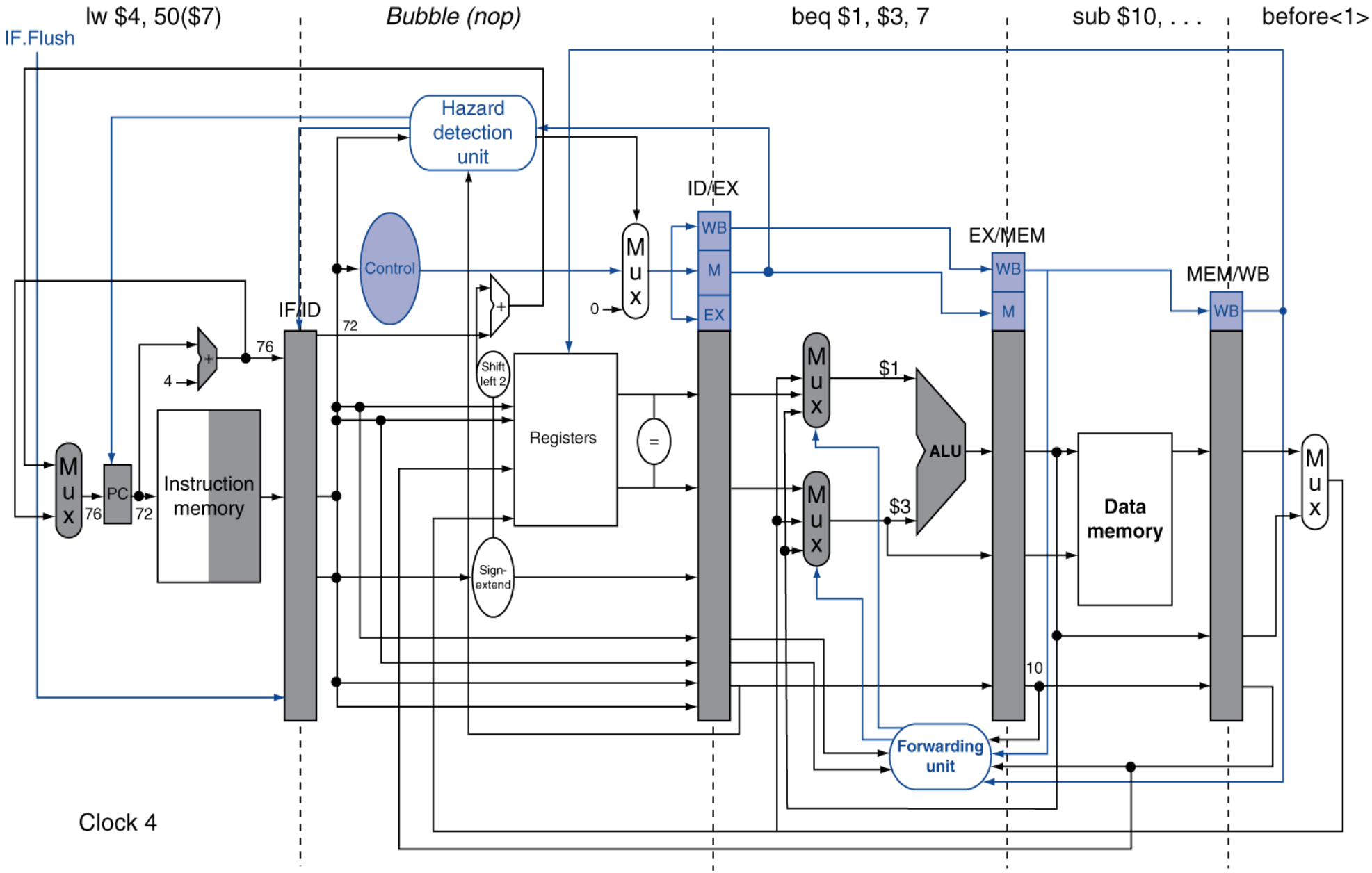
Example: branch taken

```
36:  sub  $10, $4, $8
40:  beq  $1,  $3, 7
44:  and  $12, $2, $5
48:  or   $13, $2, $6
52:  add  $14, $4, $2
56:  slt  $15, $6, $7
     ...
72:  lw   $4, 50($7)
```

Example: Branch Taken



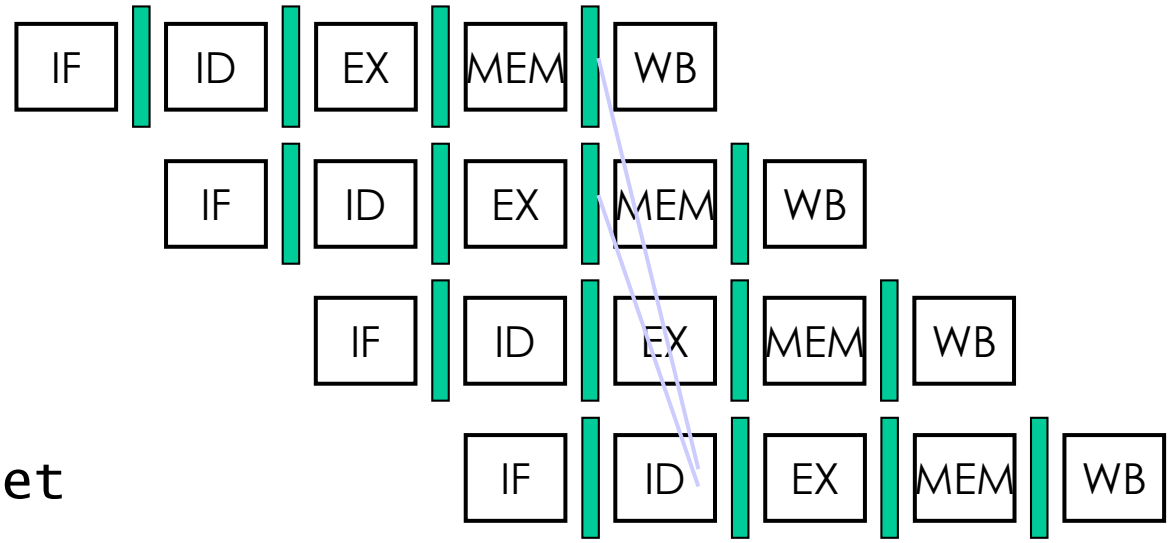
Example: Branch Taken



Data Hazards for Branches

If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```

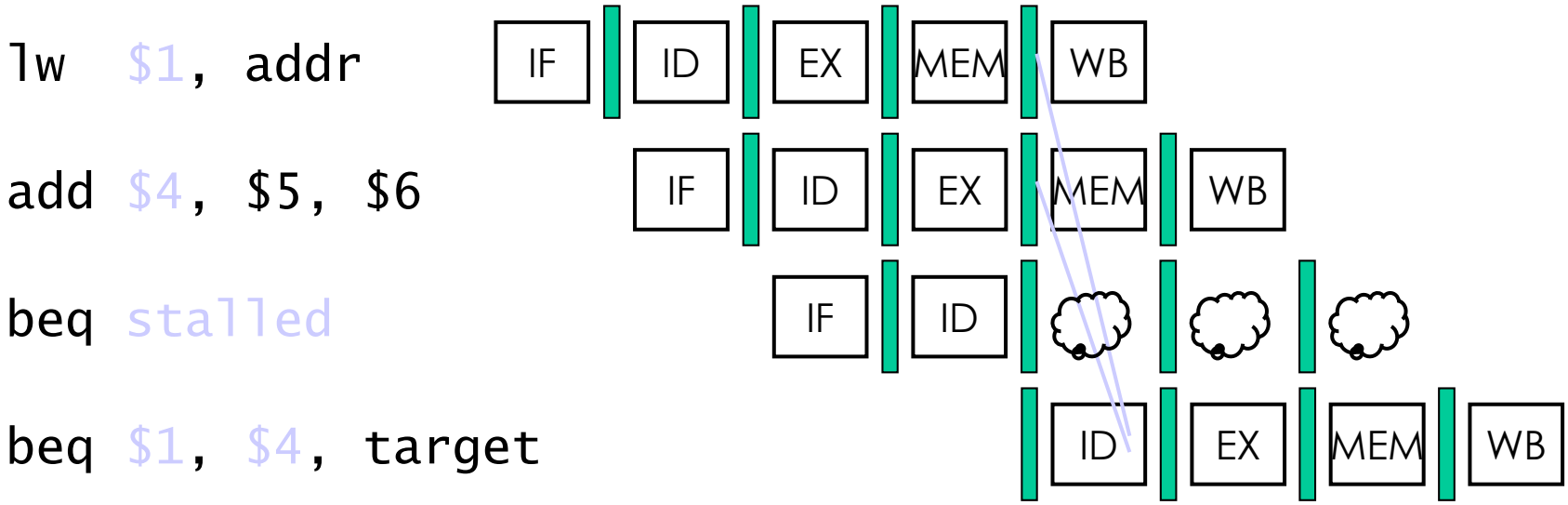


Can resolve using forwarding

Data Hazards for Branches

If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction

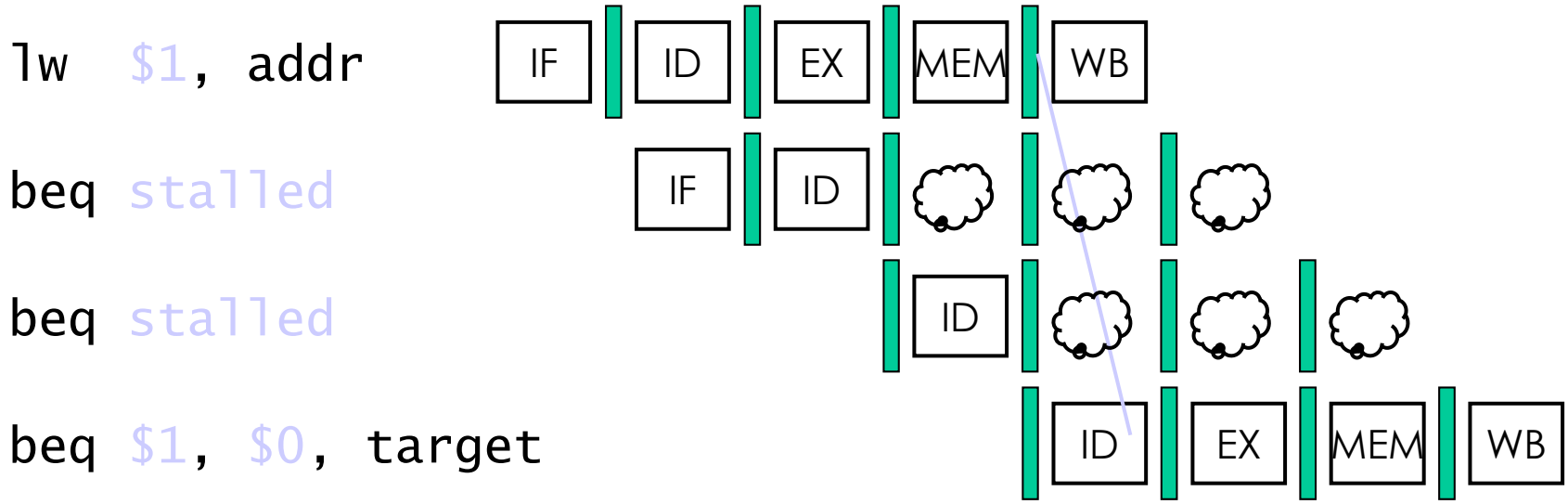
Need 1 stall cycle



Data Hazards for Branches

If a comparison register is a destination of immediately preceding load instruction

- Need 2 stall cycles



Branch Prediction

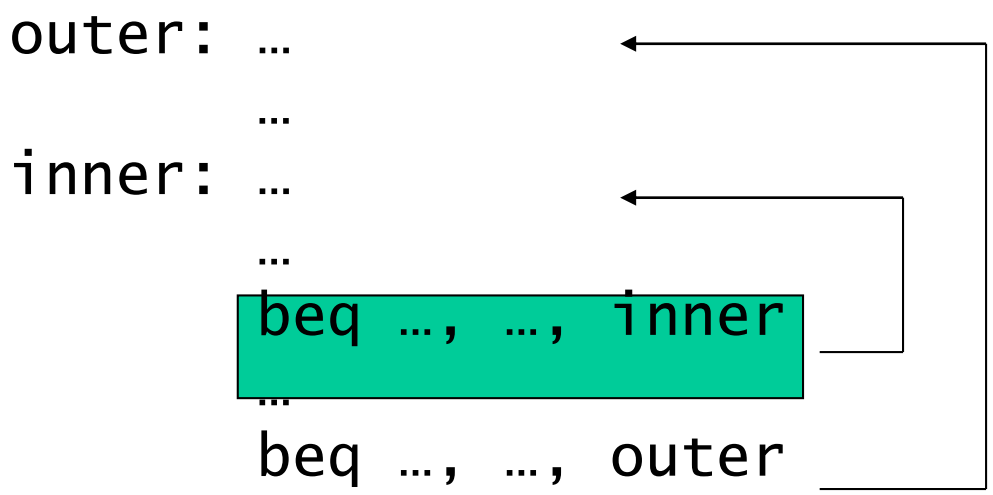
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict (i.e., guess) outcome of branch
  - Only stall if prediction is wrong
- Simplest prediction strategy
  - predict branches not taken
  - Works well for loops if the loop tests are done at the start.
  - Fetch instruction after branch, with no delay

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

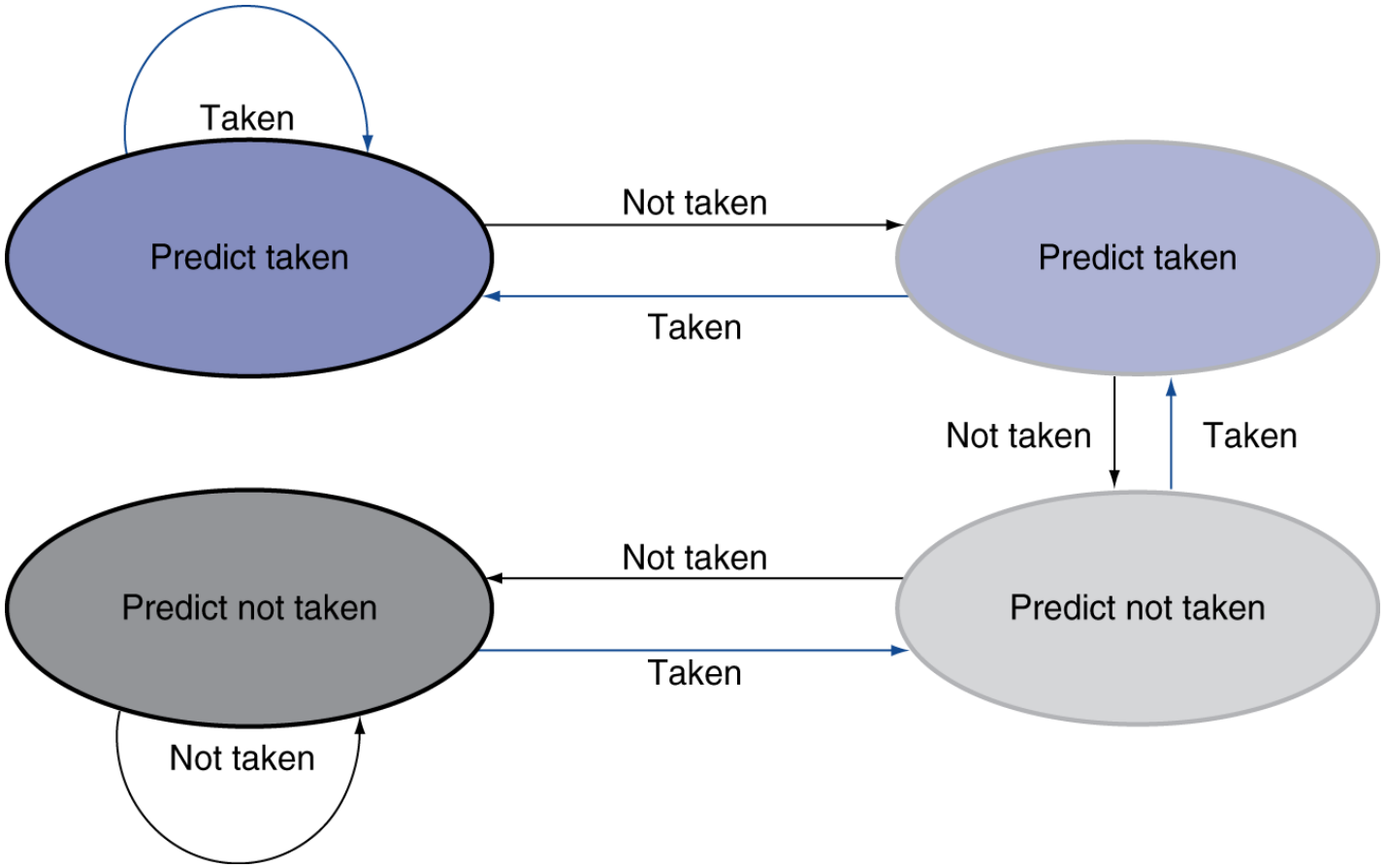
Inner loop branches mispredicted twice!



- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

Only change prediction on two successive mispredictions



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
  - Main additions in hardware:
    - forwarding unit
    - hazard detection and stalling
    - branch predictor
    - branch target table