CENG 3420
Computer Organization and Design

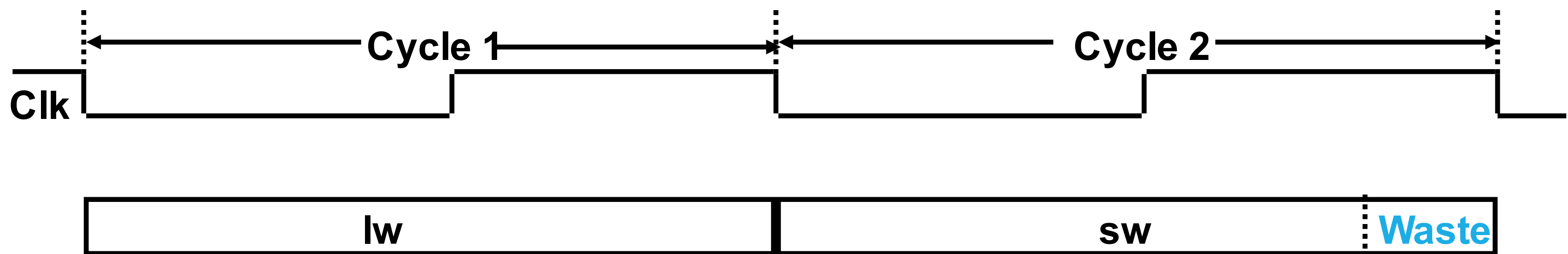
Lecture 07: Pipeline Review

Bei Yu

香港中文大學
The Chinese University of Hong Kong

Review: Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instr
  - especially problematic for more complex instructions like floating point multiply

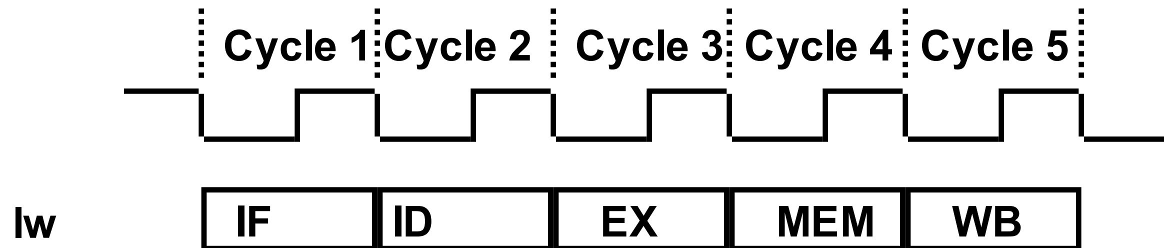


- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

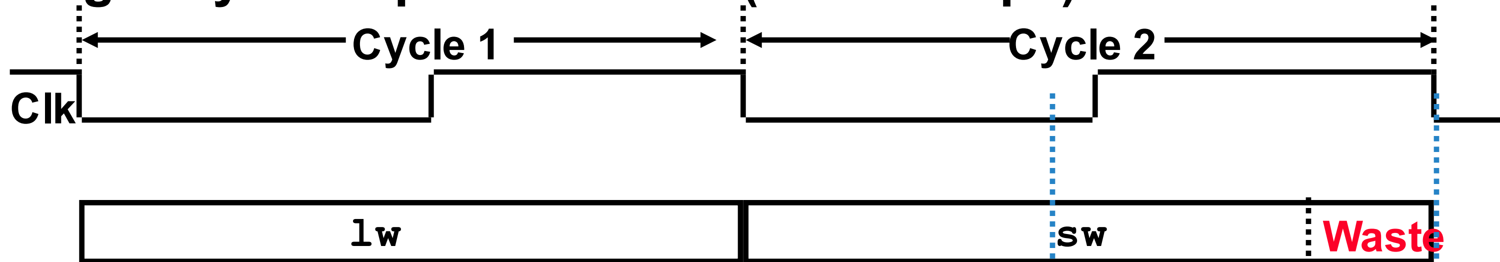
- It is simple and easy to understand

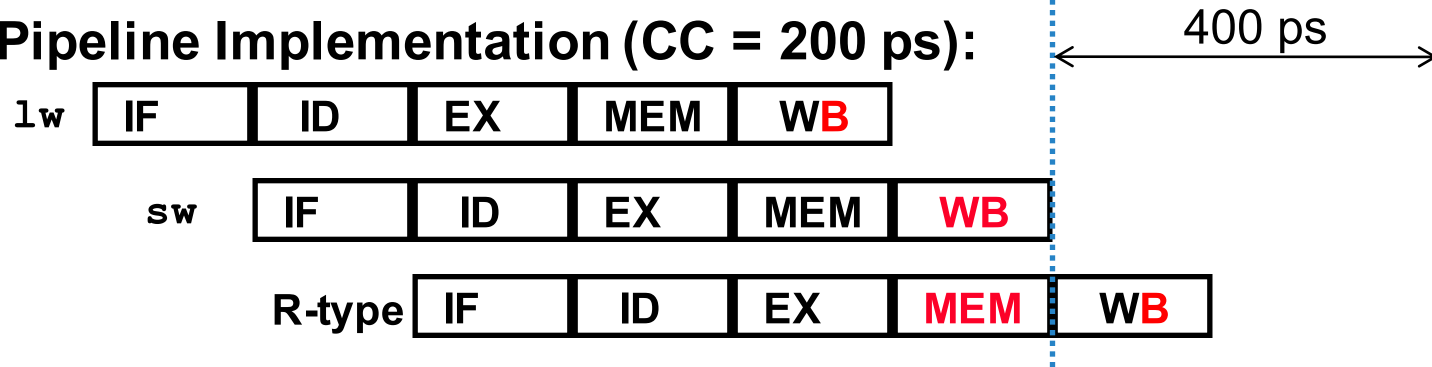
The Five Stages of Load Instruction



- IF: Instruction Fetch and Update PC
- ID: Registers Fetch and Instruction Decode
- EX: Execute R-type; calculate memory address
- MEM: Read/write the data from/to the Data Memory
- WB: Write the result data into the register file

Single Cycle versus Pipeline

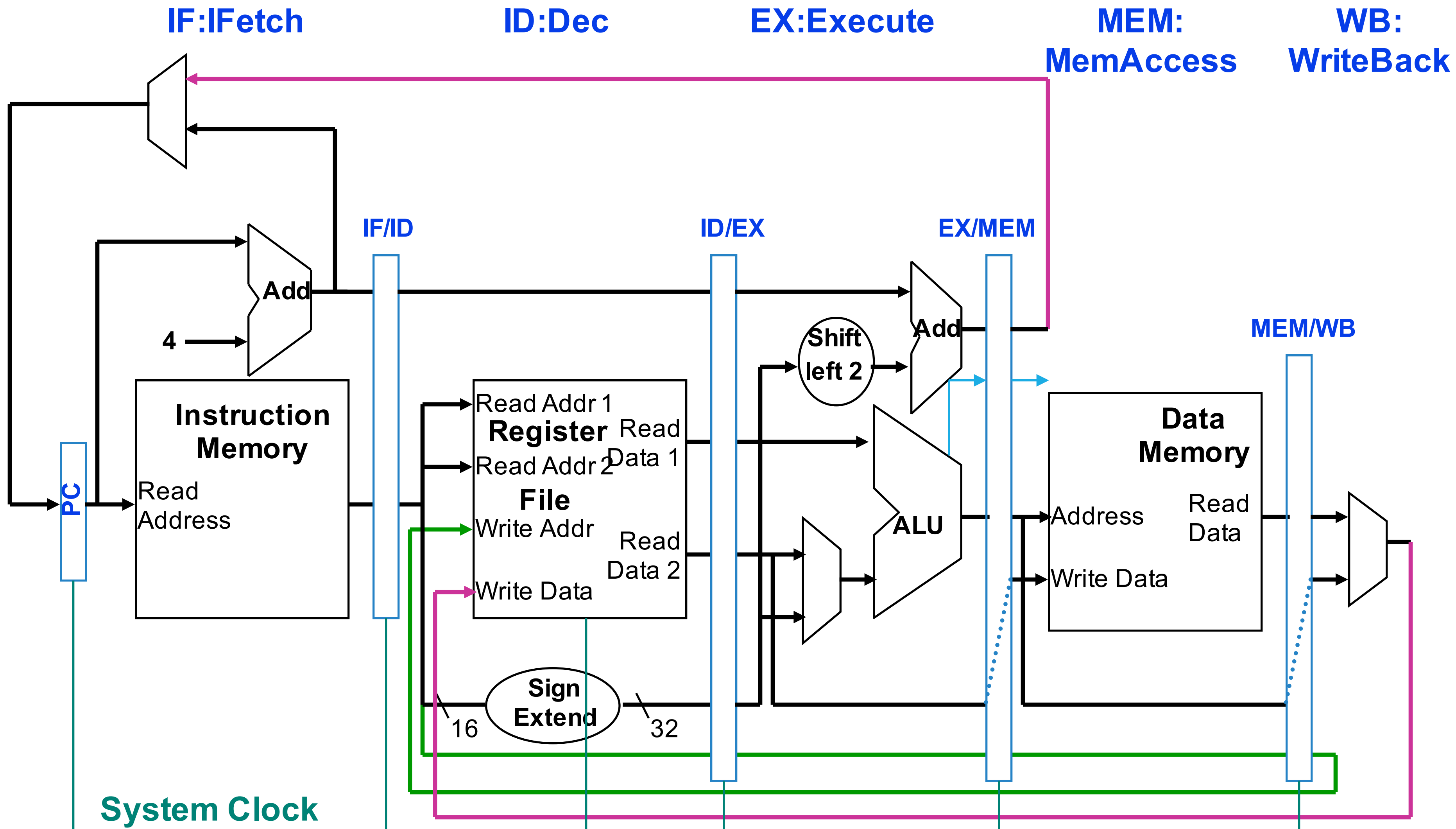




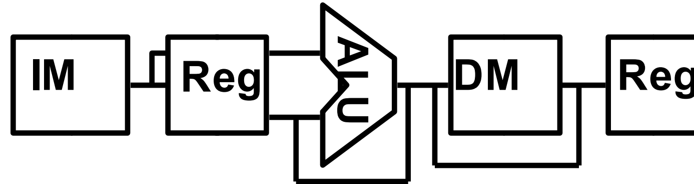
- To complete an entire instruction in the pipelined case takes 1000 ps (as compared to 800 ps for the single cycle case). Why ?
- How long does each take to complete 1,000,000 adds ?

MIPS Pipeline Datapath Additions/Mods

- State registers between each pipeline stage to isolate them



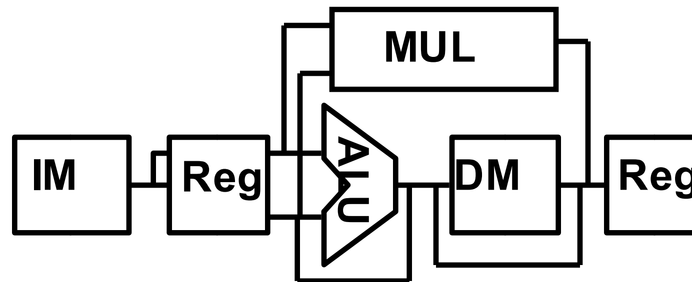
Graphically Representing MIPS Pipeline



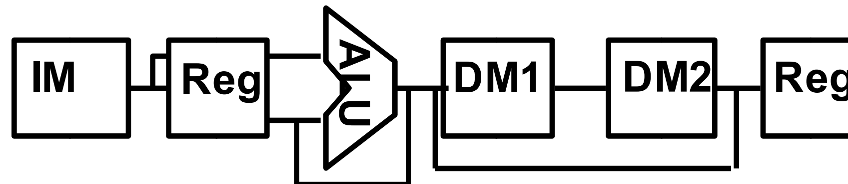
- Can help with answering questions like:
  - How many cycles does it take to execute this code?
  - What is the ALU doing during cycle 4?
  - Is there a hazard, why does it occur, and how can it be fixed?

Other Pipeline Structures Are Possible

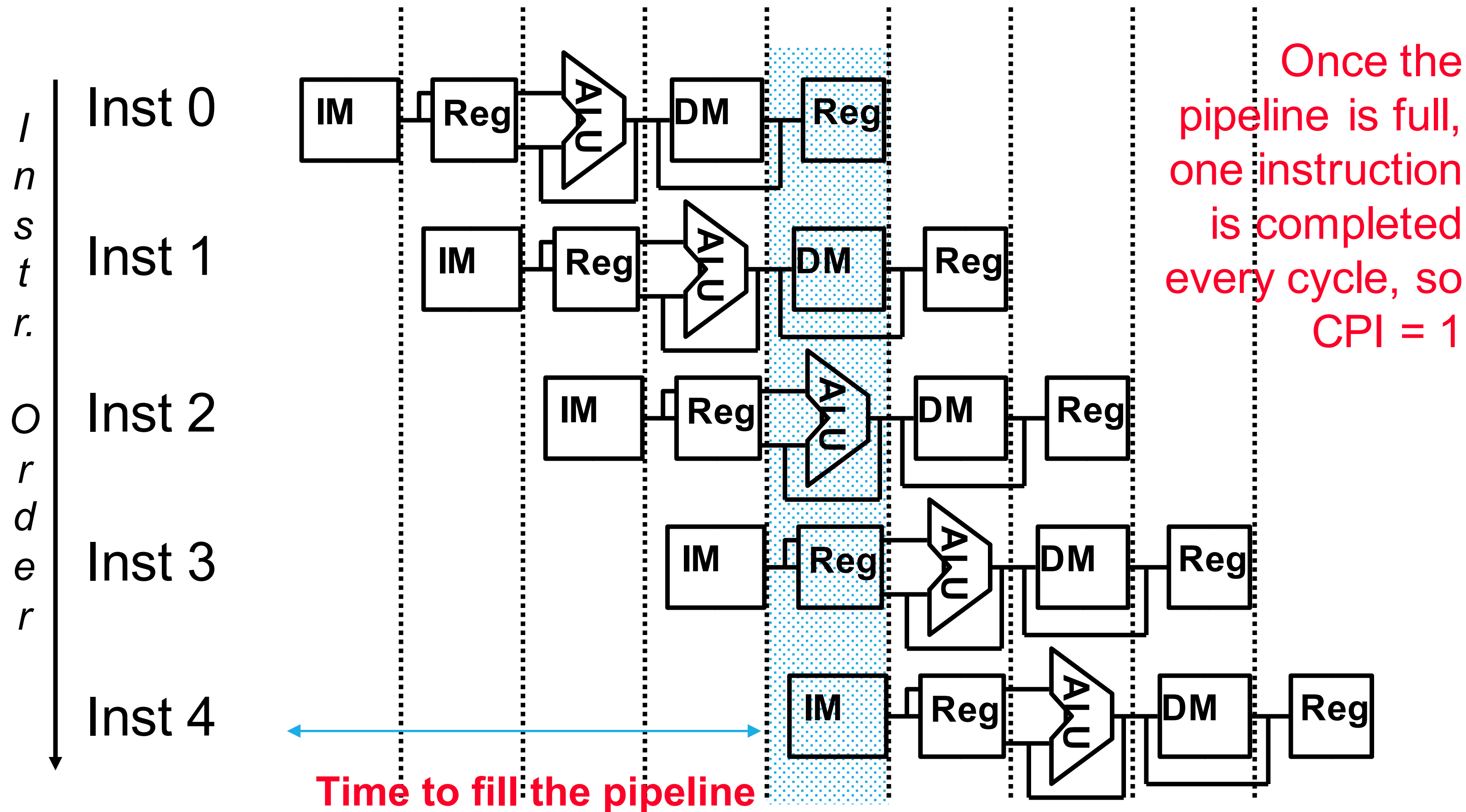
- What about the (slow) multiply operation?
  - Make the clock twice as slow or …
  - let it take two cycles (since it doesn’t use the DM stage)



- What if the data memory access is twice as slow as the instruction memory?
  - make the clock twice as slow or …
  - let data memory access take two cycles (and keep the same clock rate)



Why Pipeline? For Performance!

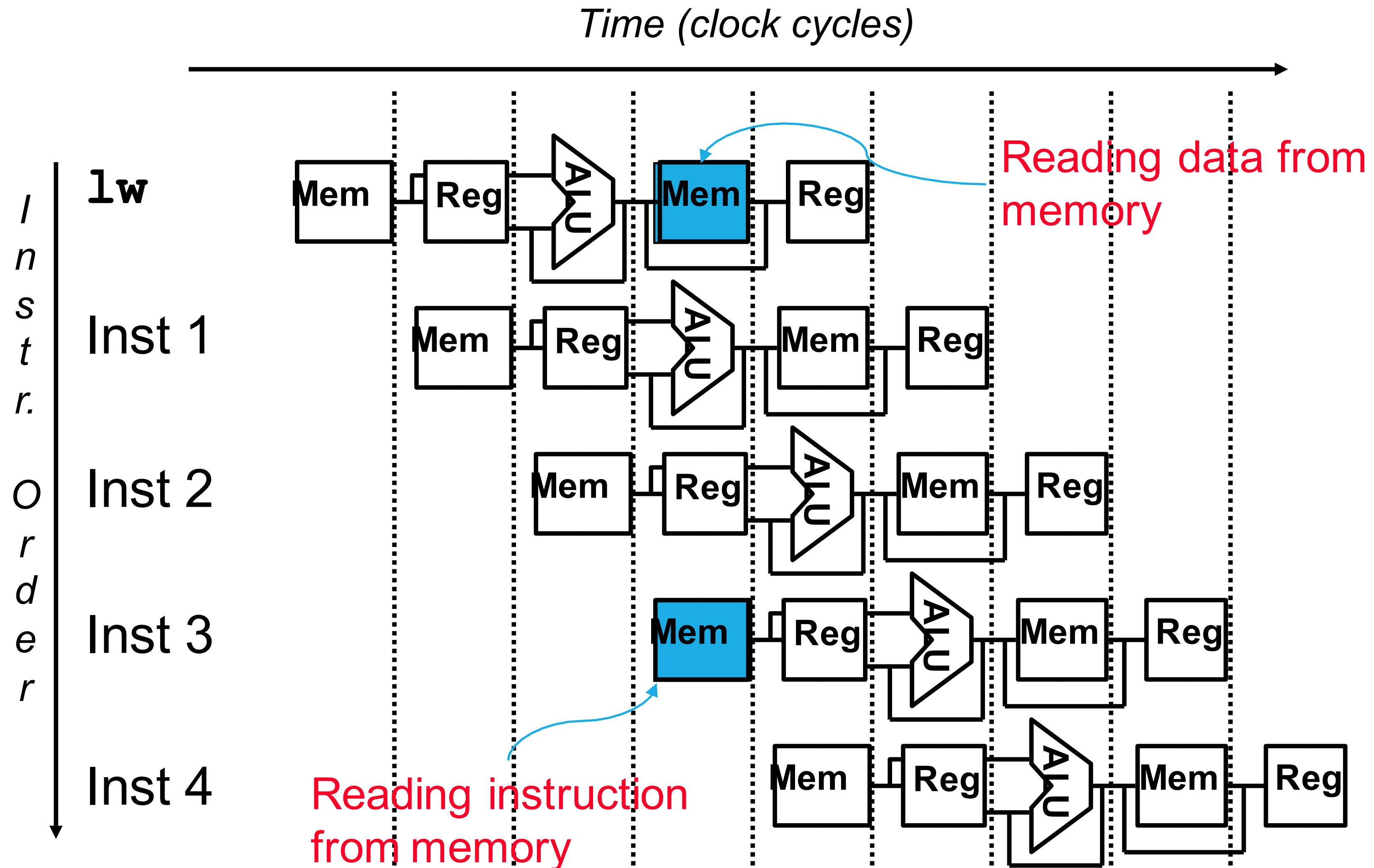


Can Pipelining Get Us Into Trouble?

- Yes: Pipeline Hazards
  - structural hazards:
    - a required resource is busy
  - data hazards:
    - attempt to use data before it is ready
  - control hazards:
    - deciding on control action depends on previous instruction

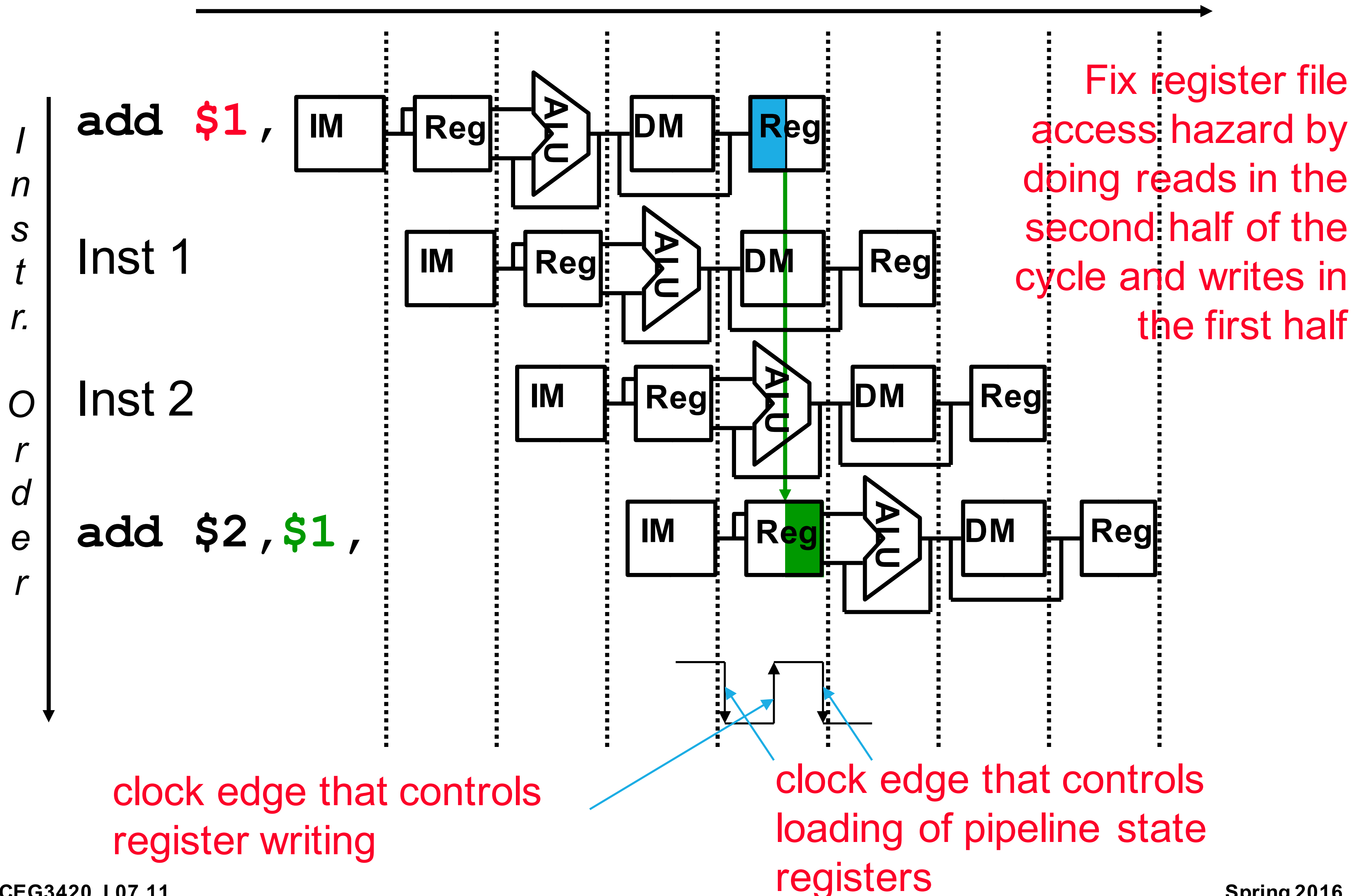
- Can usually resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

Resolve Structural Hazard 1



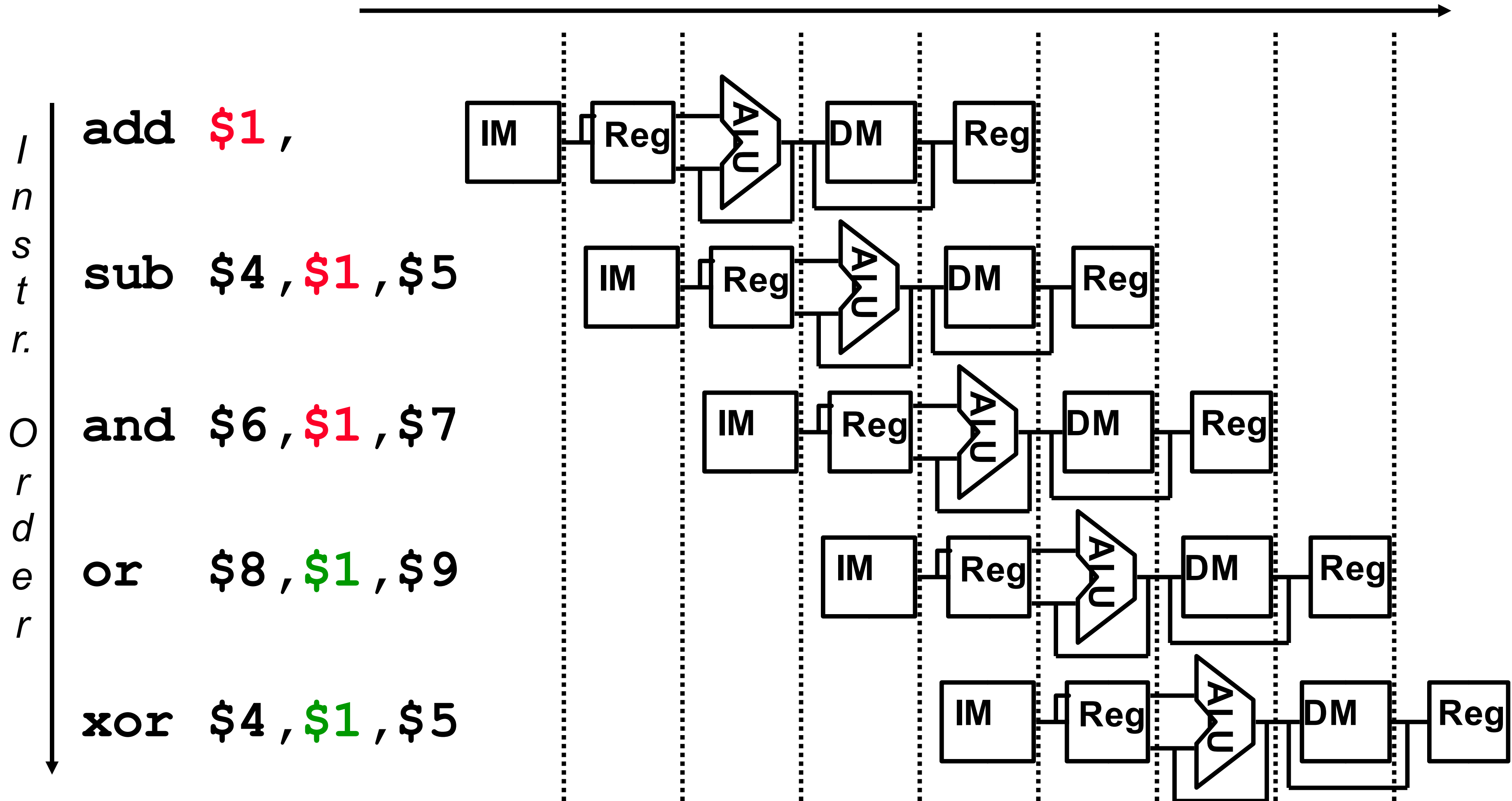
- Fix with separate instr and data memories (I$ and D$)

Resolve Structural Hazard 2



Data Hazards

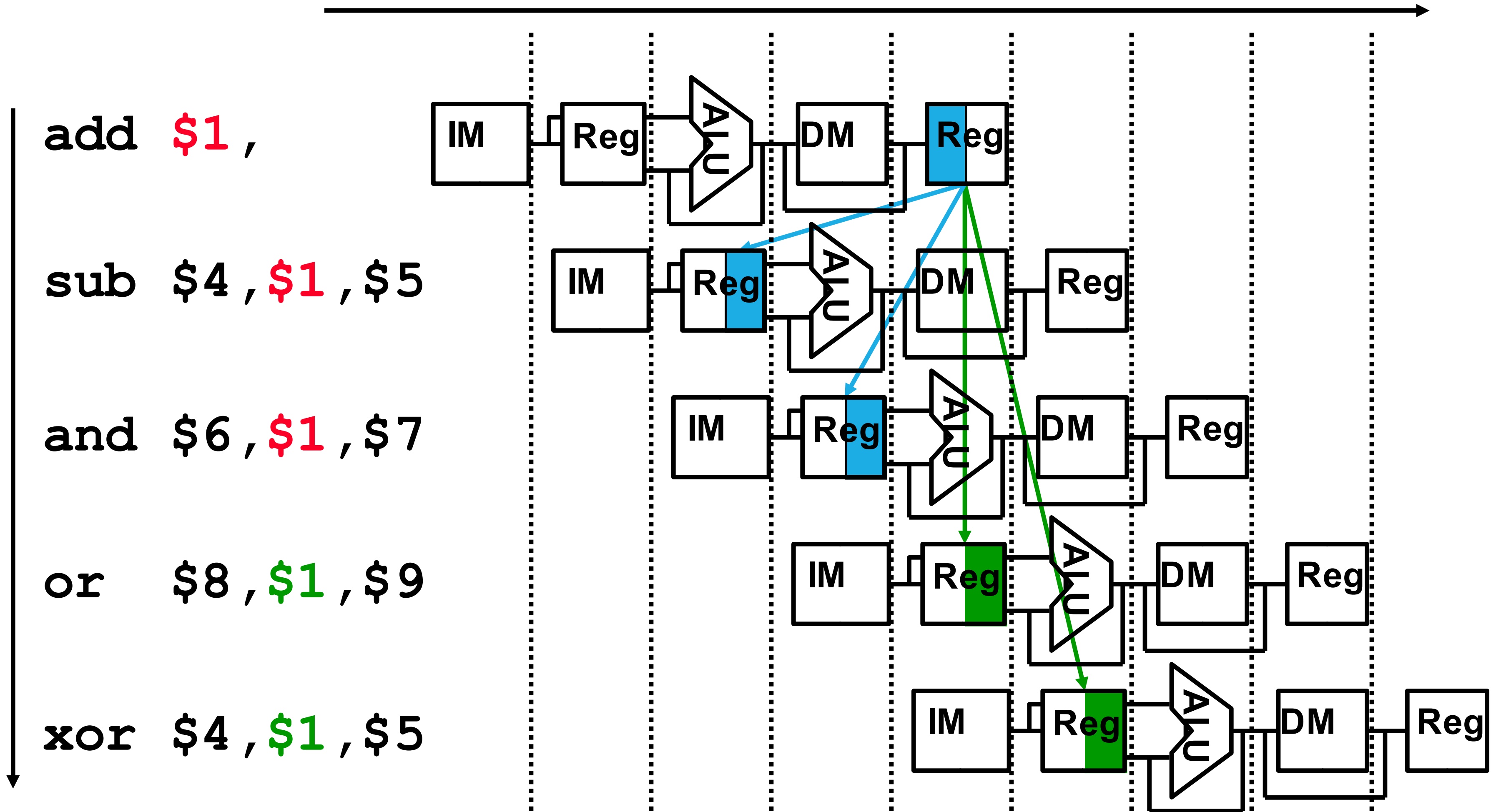
- Dependencies backward in time cause hazards



- Read After Write (RAW) data hazard

Data Hazards: Register Usage

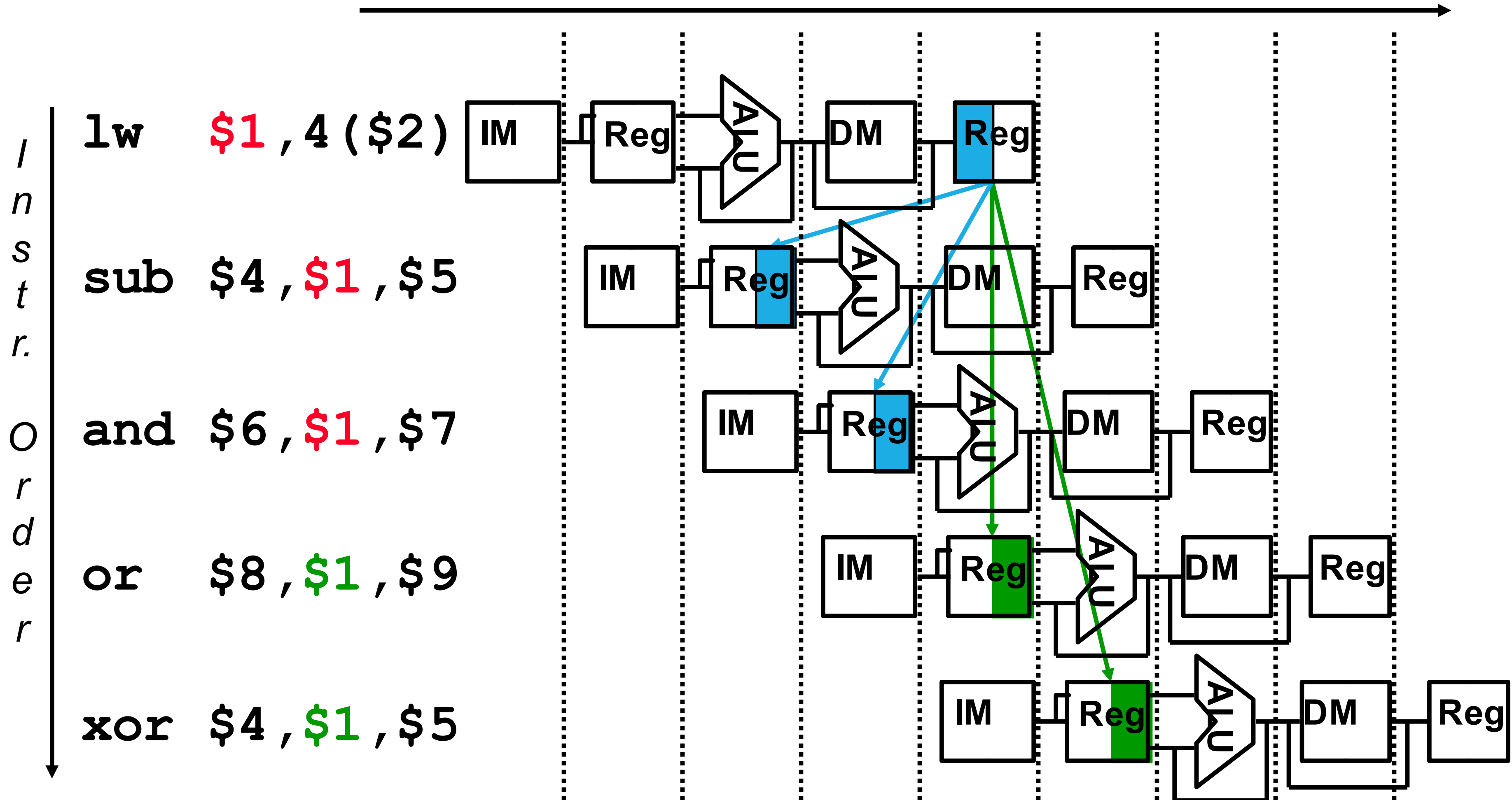
- Dependencies backward in time cause hazards



- Read After Write (RAW) data hazard

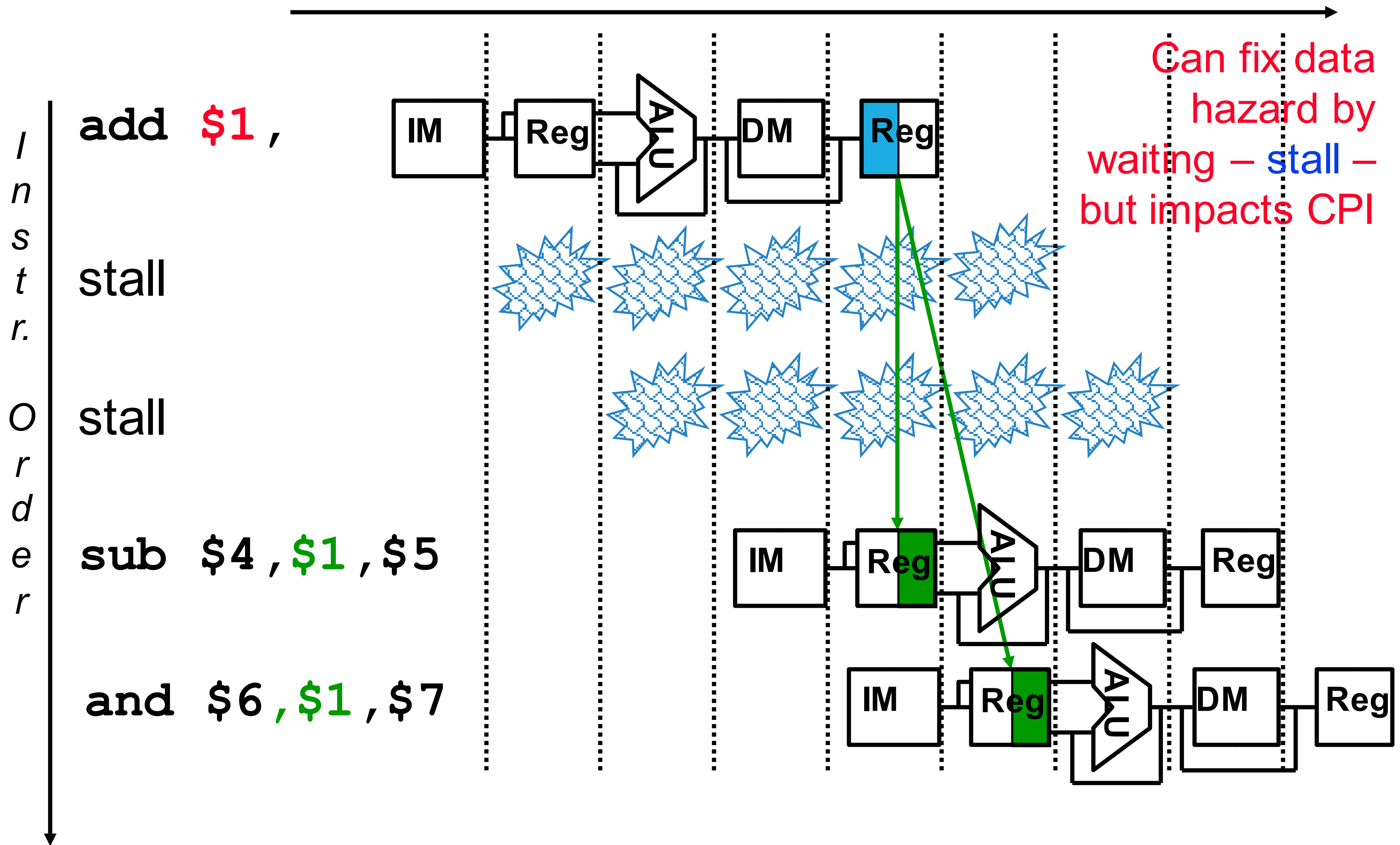
Data Hazards: Load Memory

- Dependencies backward in time cause hazards

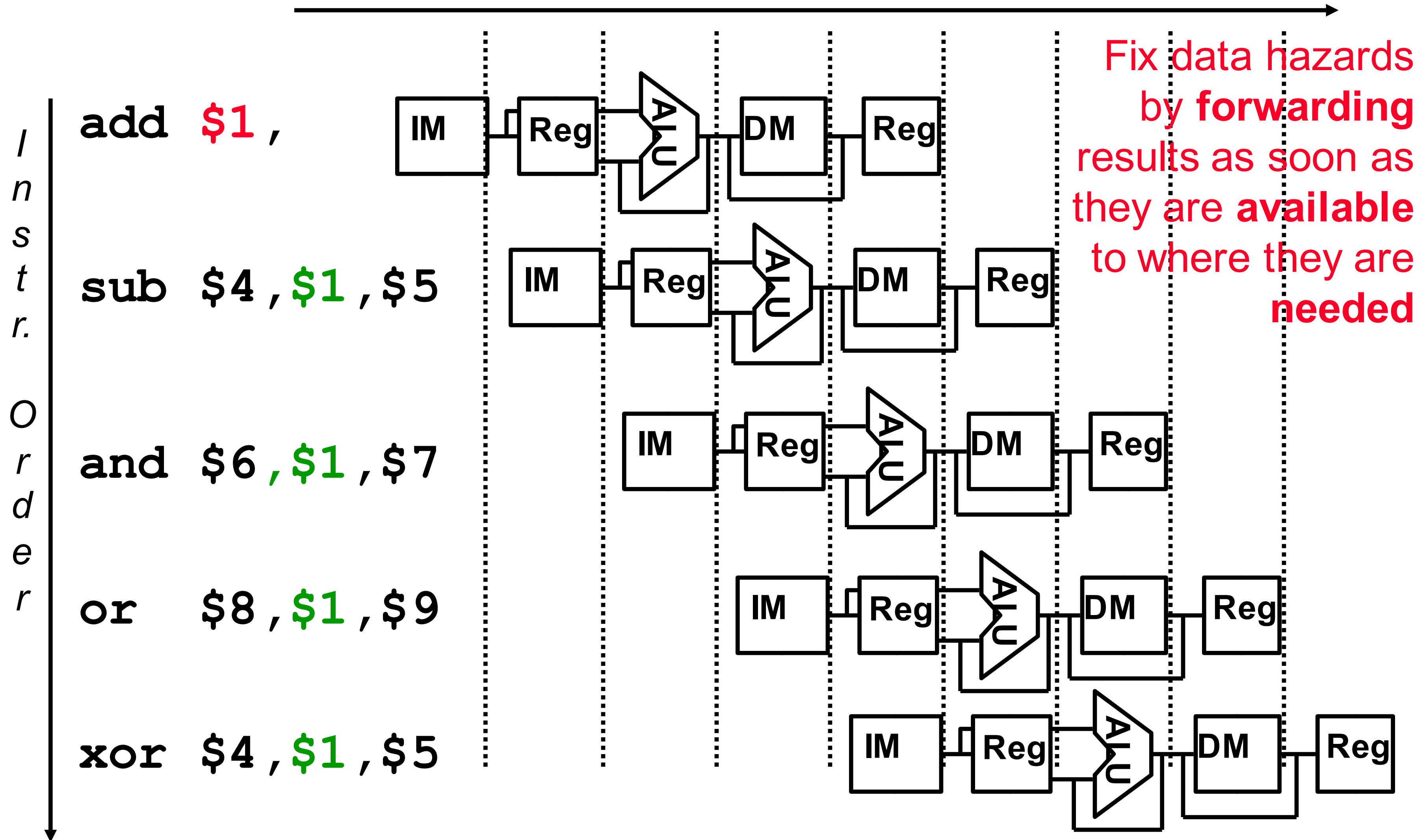


- Load-use data hazard

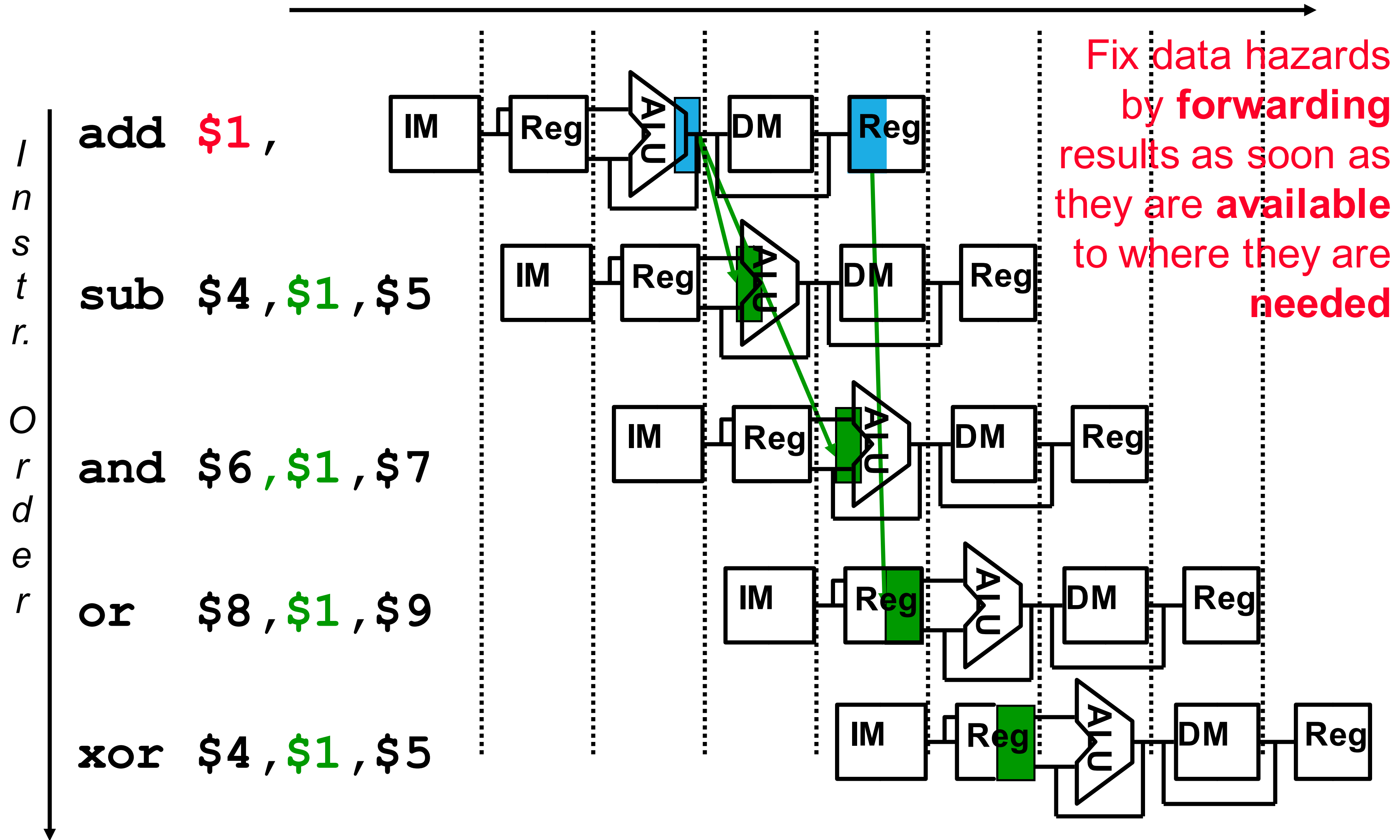
Resolve Data Hazards 1: Insert Stall



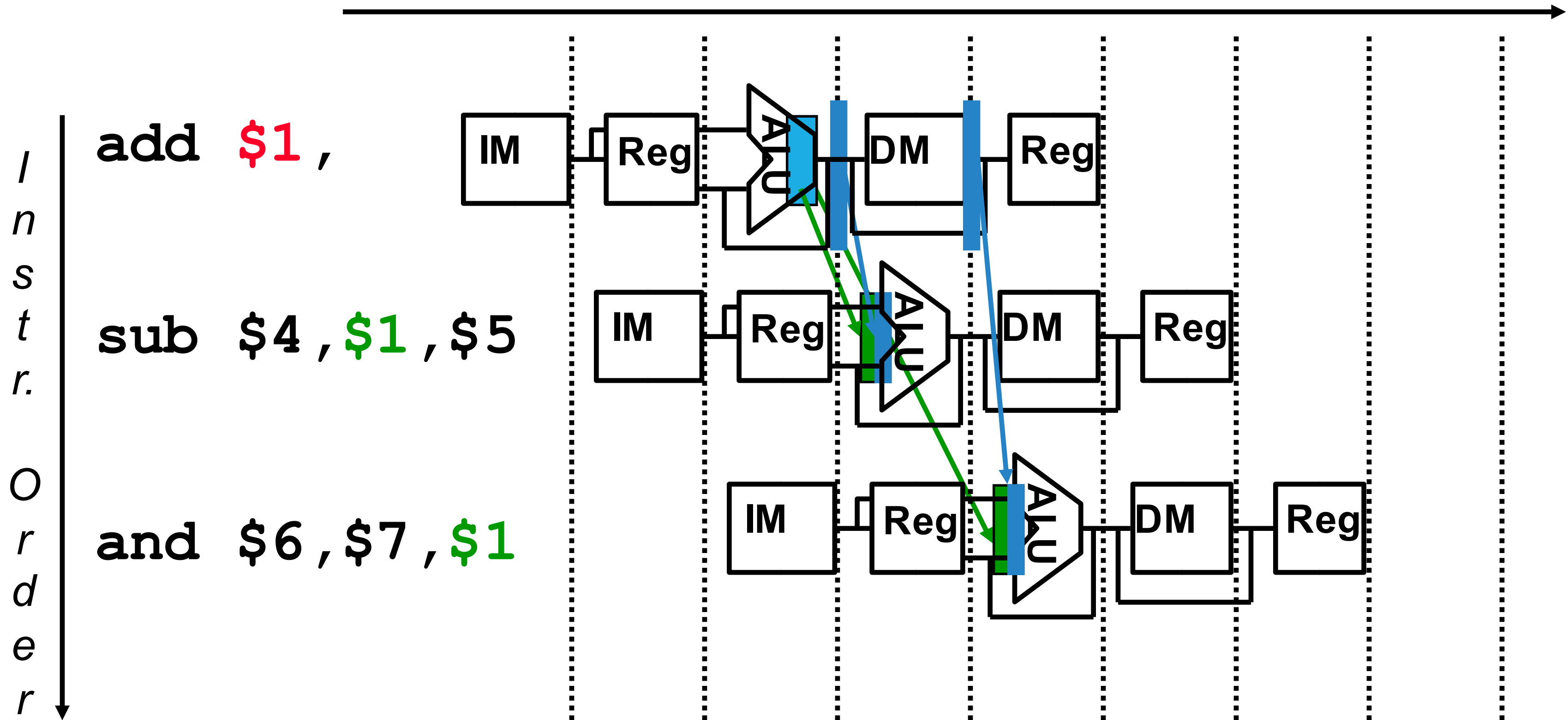
Resolve Data Hazards 2: Forwarding



Resolve Data Hazards 2: Forwarding



Forwarding Illustration

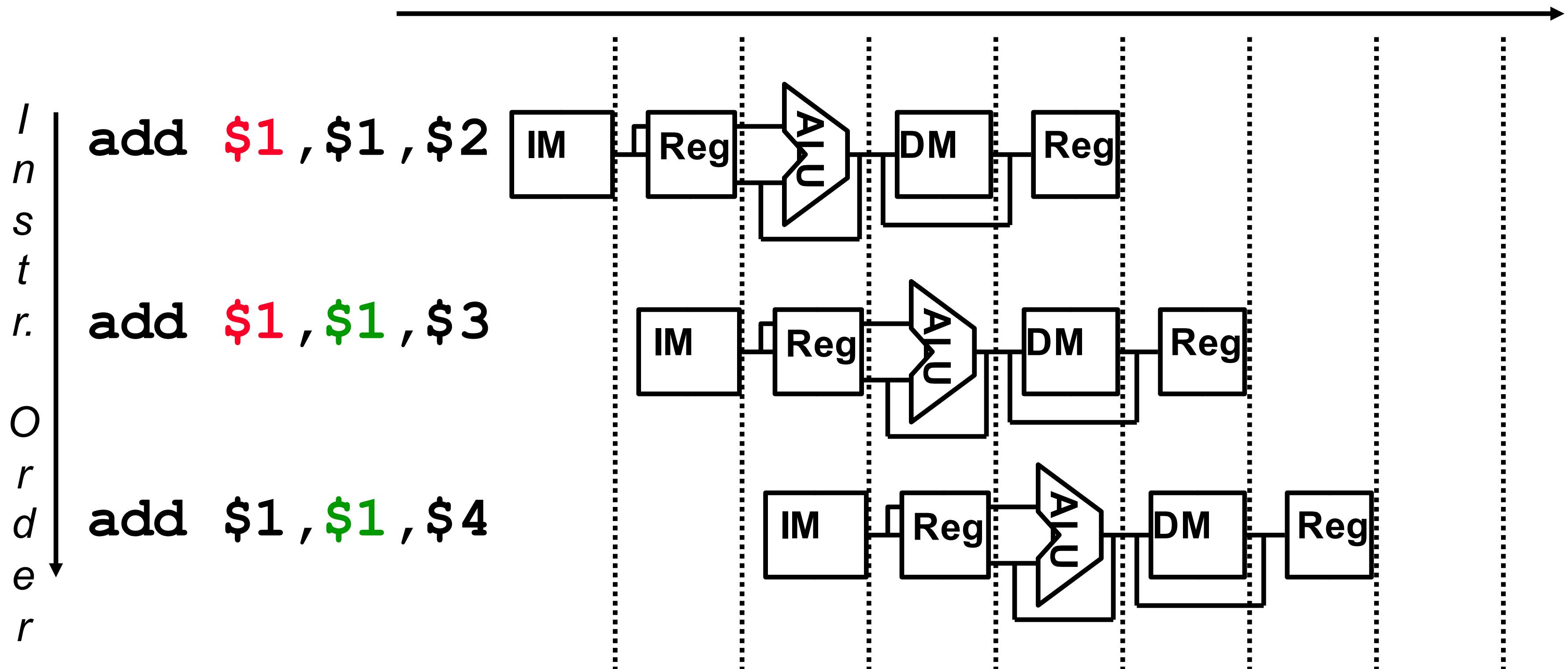


EX forwarding MEM forwarding

Green line: conceptual forwarding
Blue line: real forwarding path (from registers to ALU input)

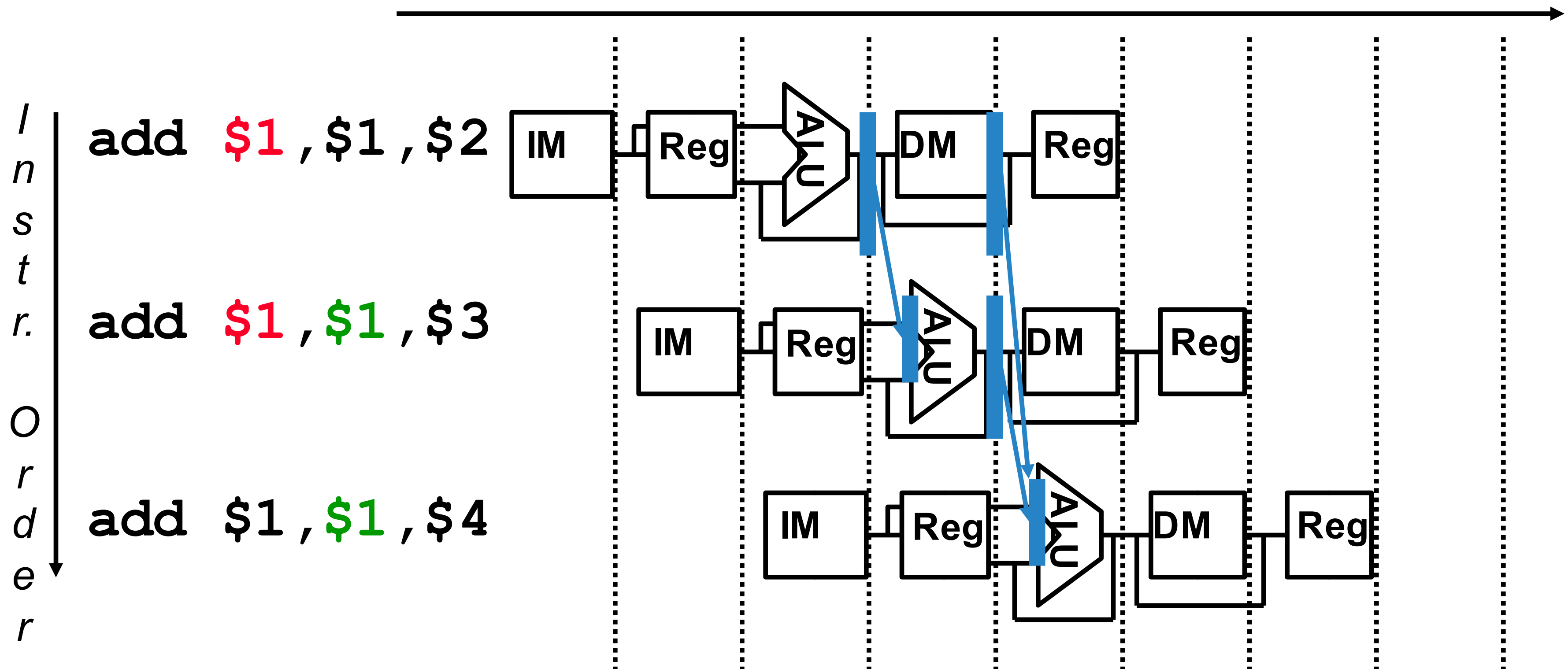
Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?



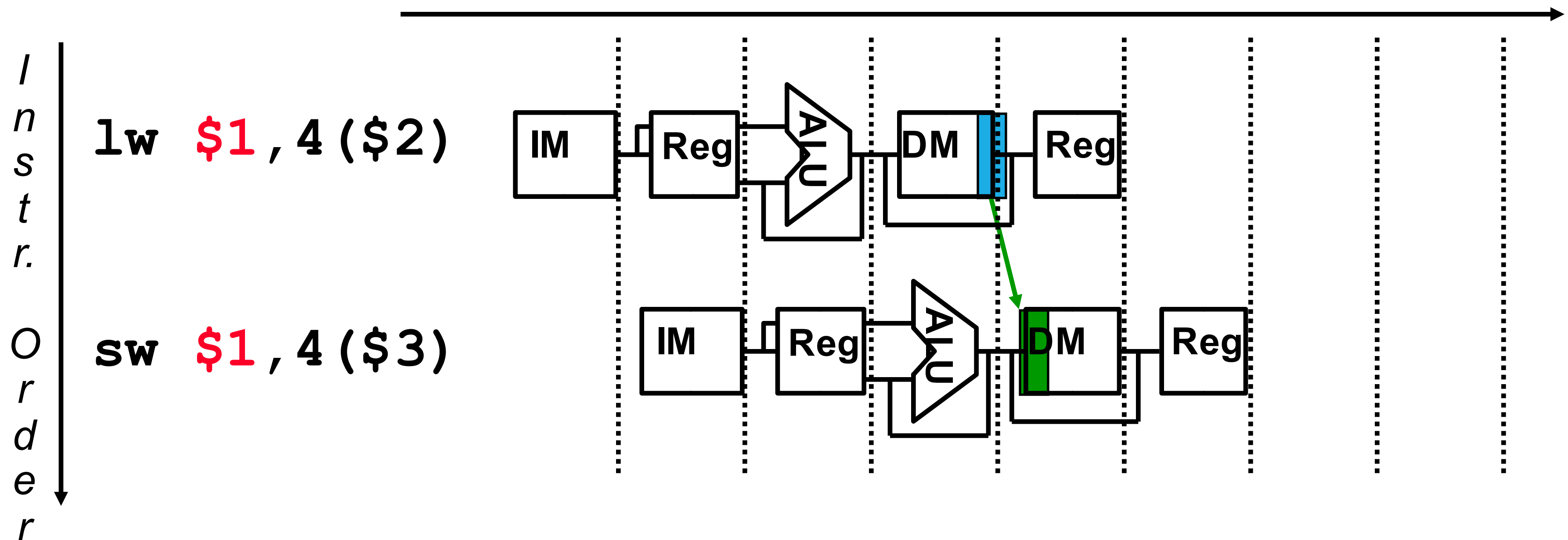
Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?

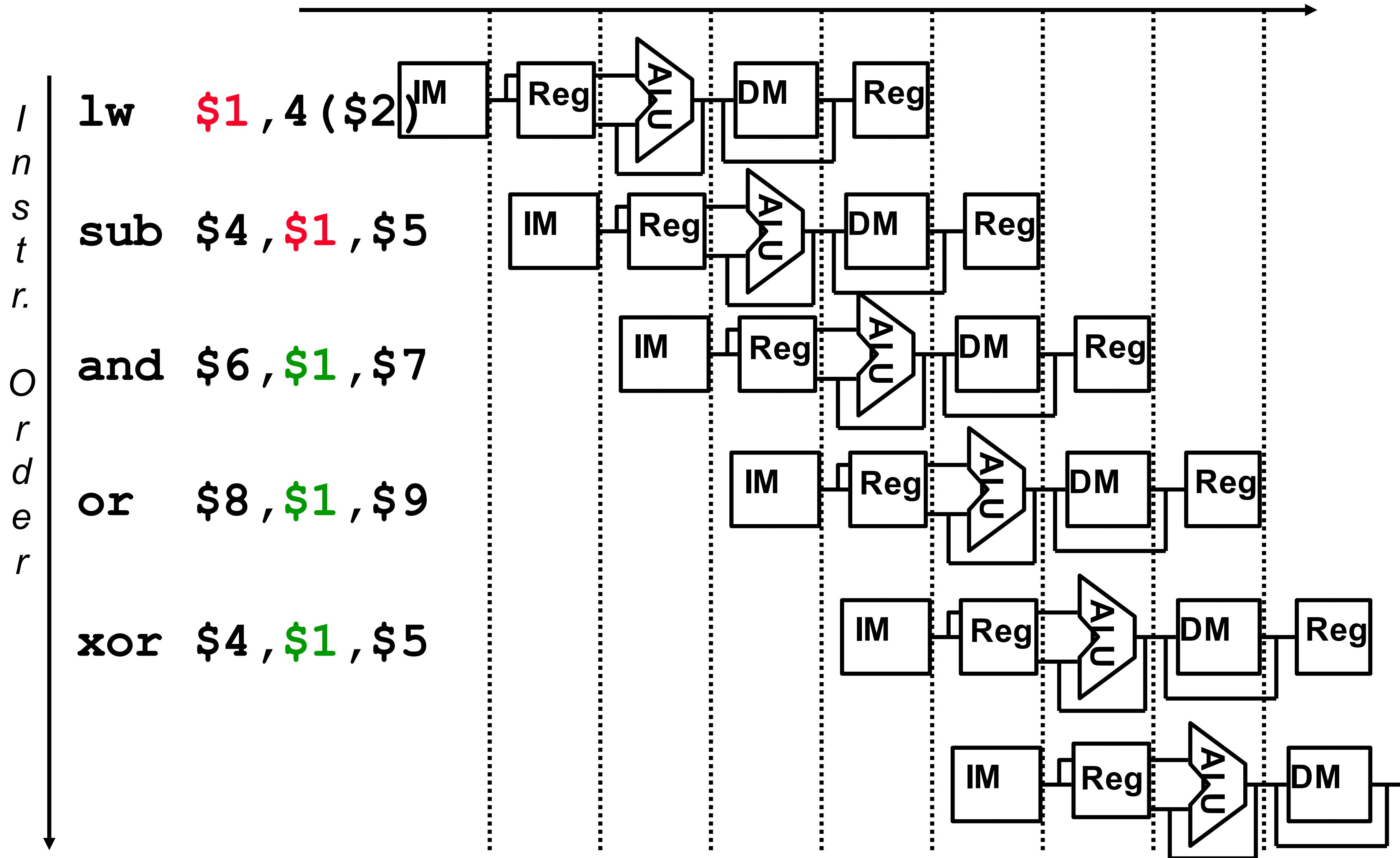


Memory-to-Memory Copies

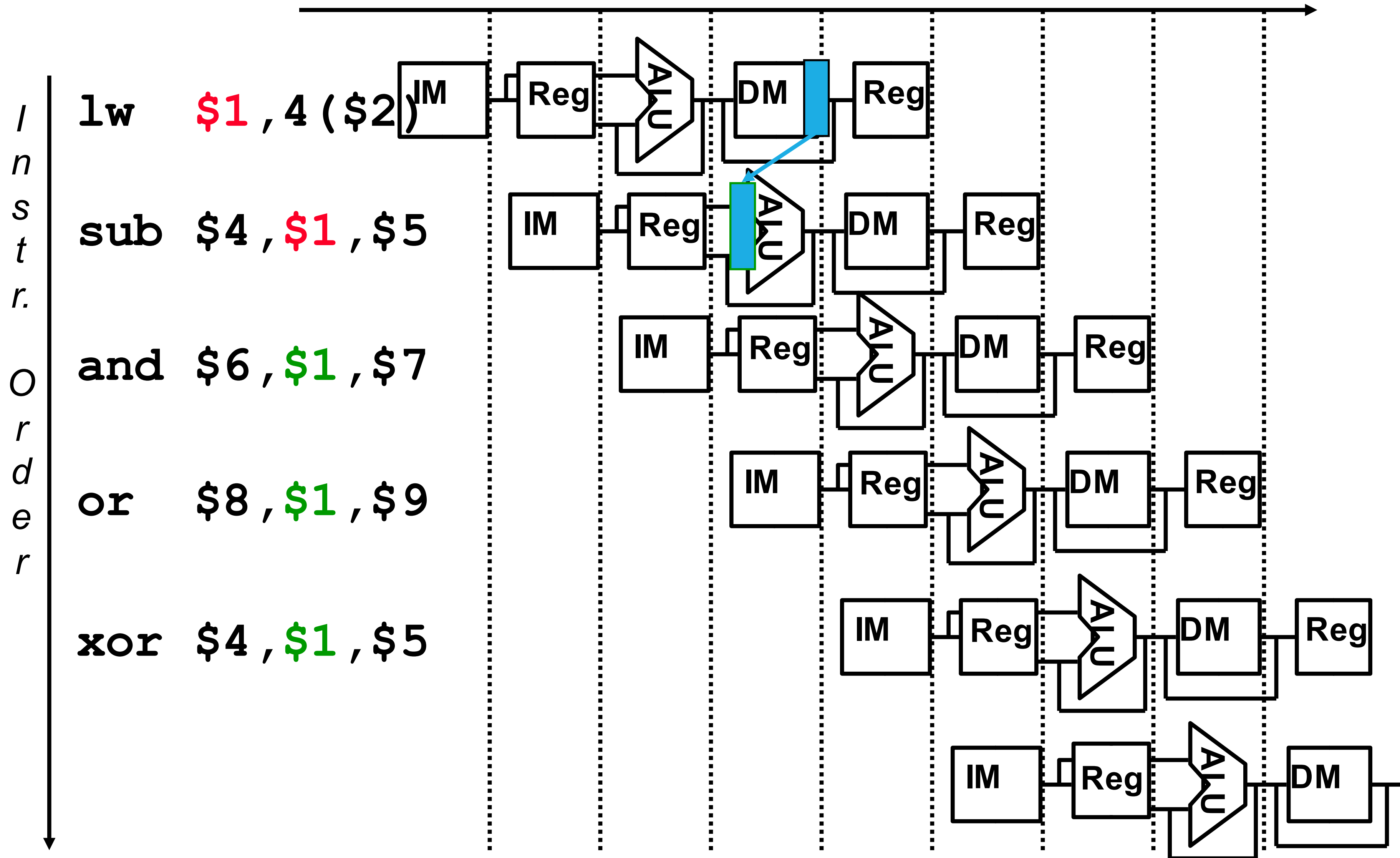
- For loads immediately followed by stores (memory-to-memory copies) can avoid a stall by adding forwarding hardware from the MEM/WB register to the data memory input.
  - Would need to add a Forward Unit and a mux to the MEM stage



Forwarding with Load-use Data Hazards

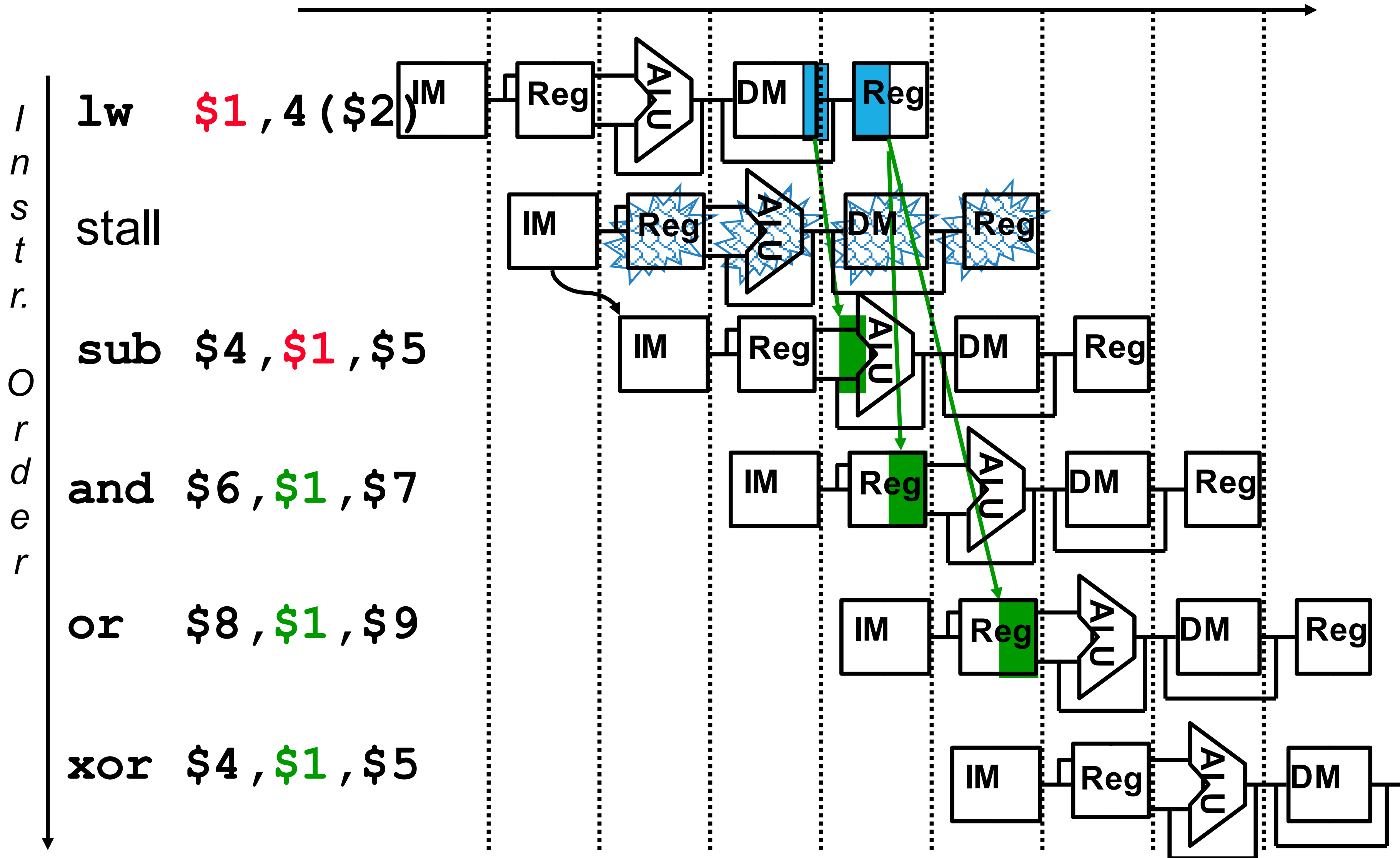


Forwarding with Load-use Data Hazards



- Will still need one stall cycle even with forwarding

Forwarding with Load-use Data Hazards



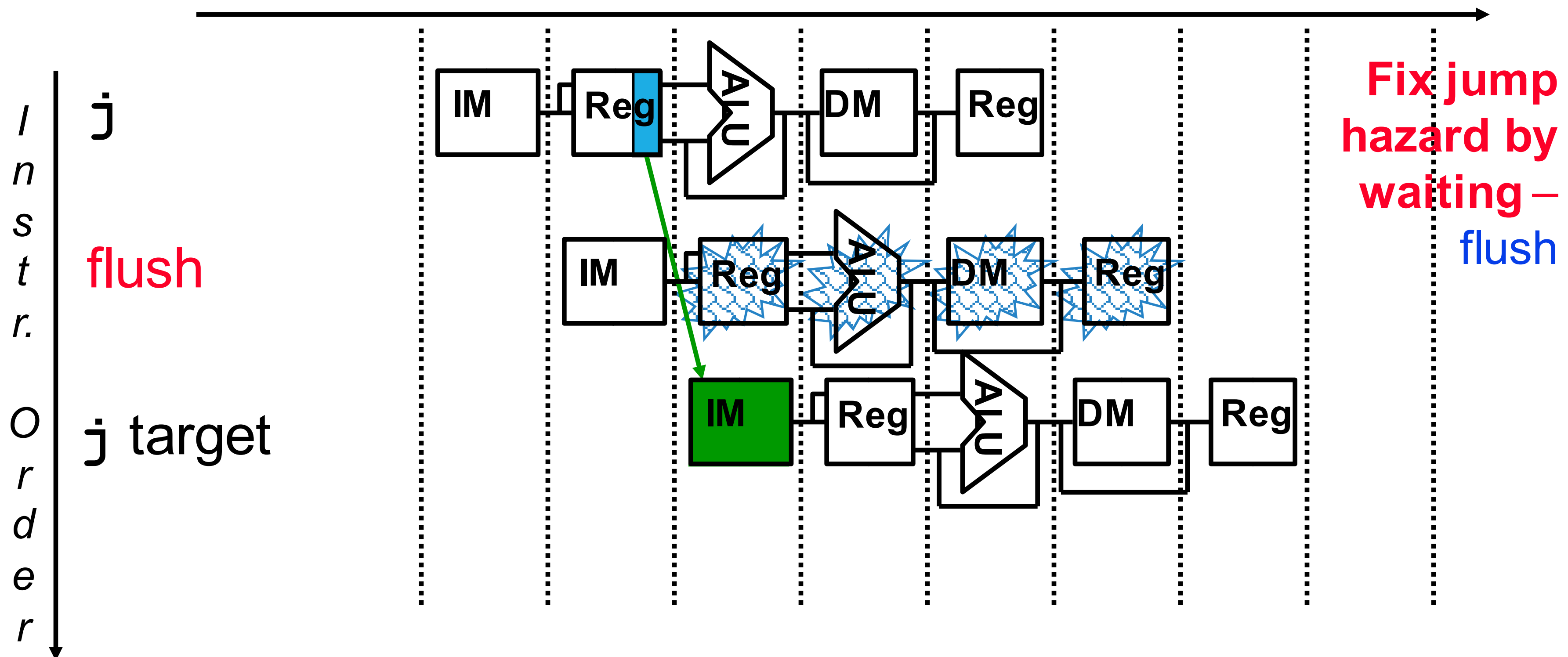
- Will still need one stall cycle even with forwarding

Control Hazards

- When the flow of instruction addresses is not sequential (i.e., PC = PC + 4); incurred by change of flow instructions
  - Unconditional branches (`j, jal, jr`)
  - Conditional branches (`beq`, `bne`)
  - Exceptions
- Possible approaches
  - Stall (impacts CPI)
  - Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Delay decision (requires compiler support)
  - Predict and hope for the best !
- Control hazards occur less frequently than data hazards, but there is *nothing* as effective against control hazards as forwarding is for data hazards

Control Hazards 1: Jumps Incur One Stall

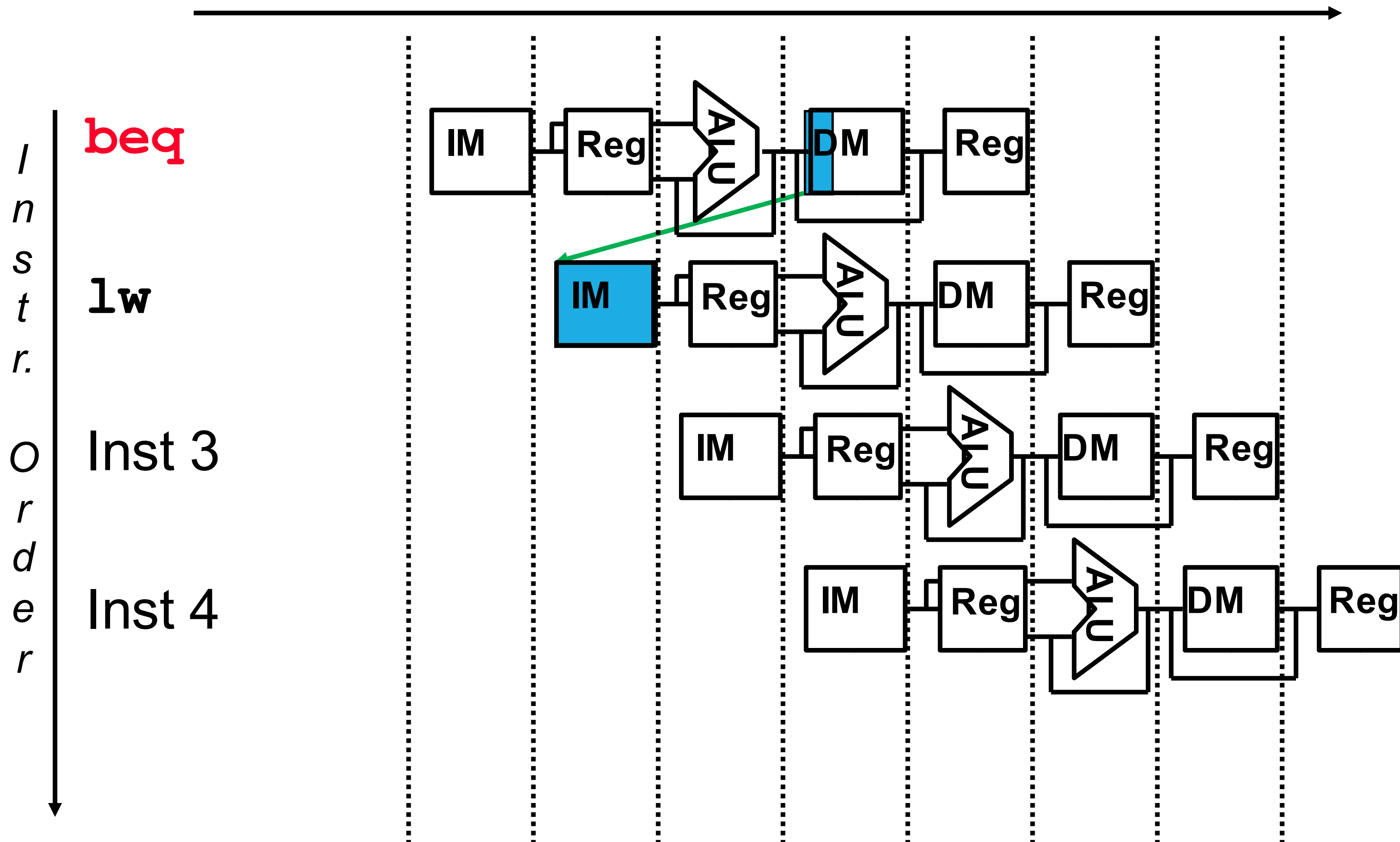
- Jumps not decoded until ID, so one flush is needed
  - To flush, set `IF.Flush` to zero the instruction field of the IF/ID pipeline register (turning it into a `nop`)



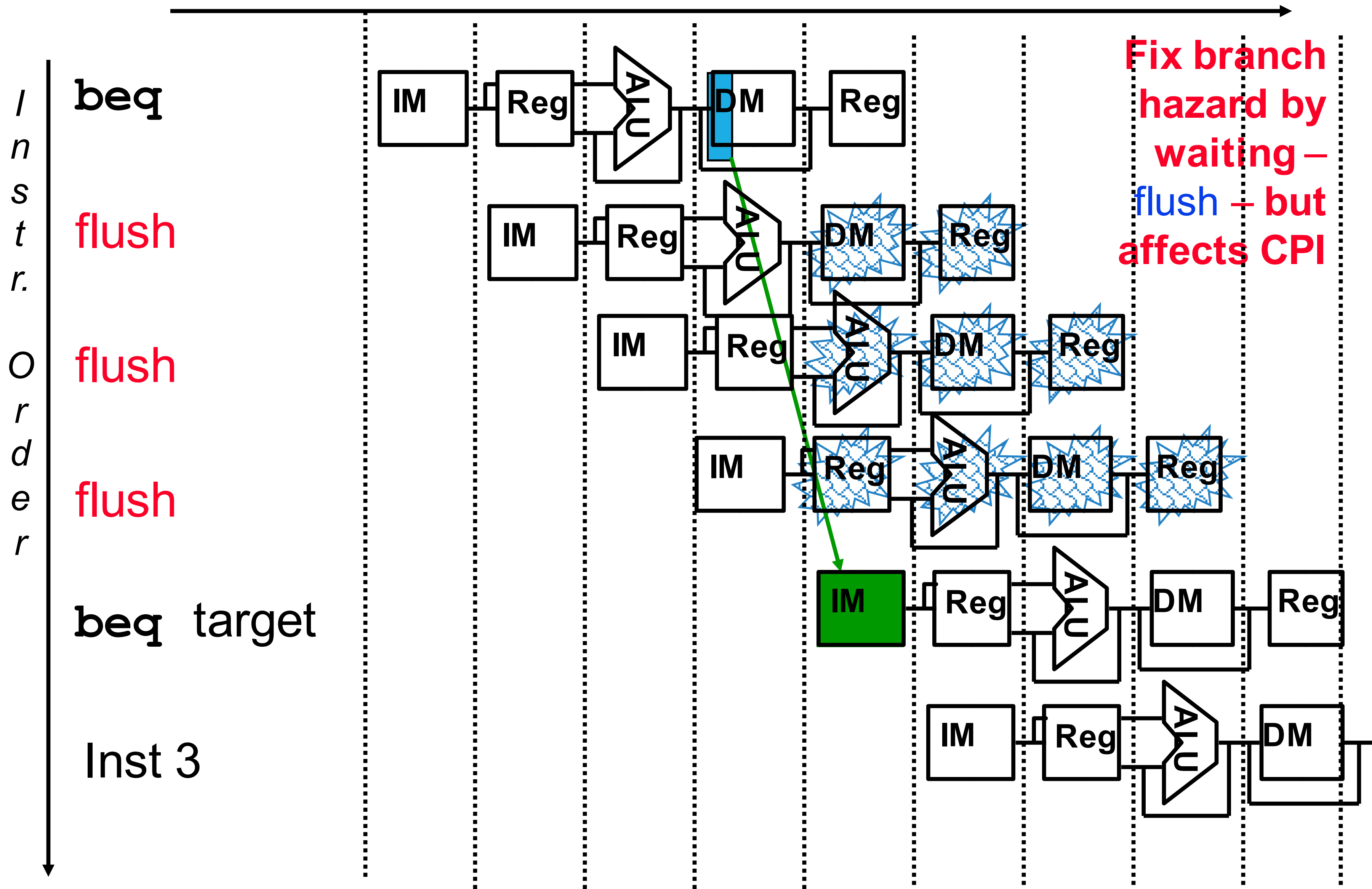
- Fortunately, jumps are very infrequent – only 3% of the SPECint instruction mix

Control Hazards 2: Branch Instr

- Dependencies backward in time cause hazards

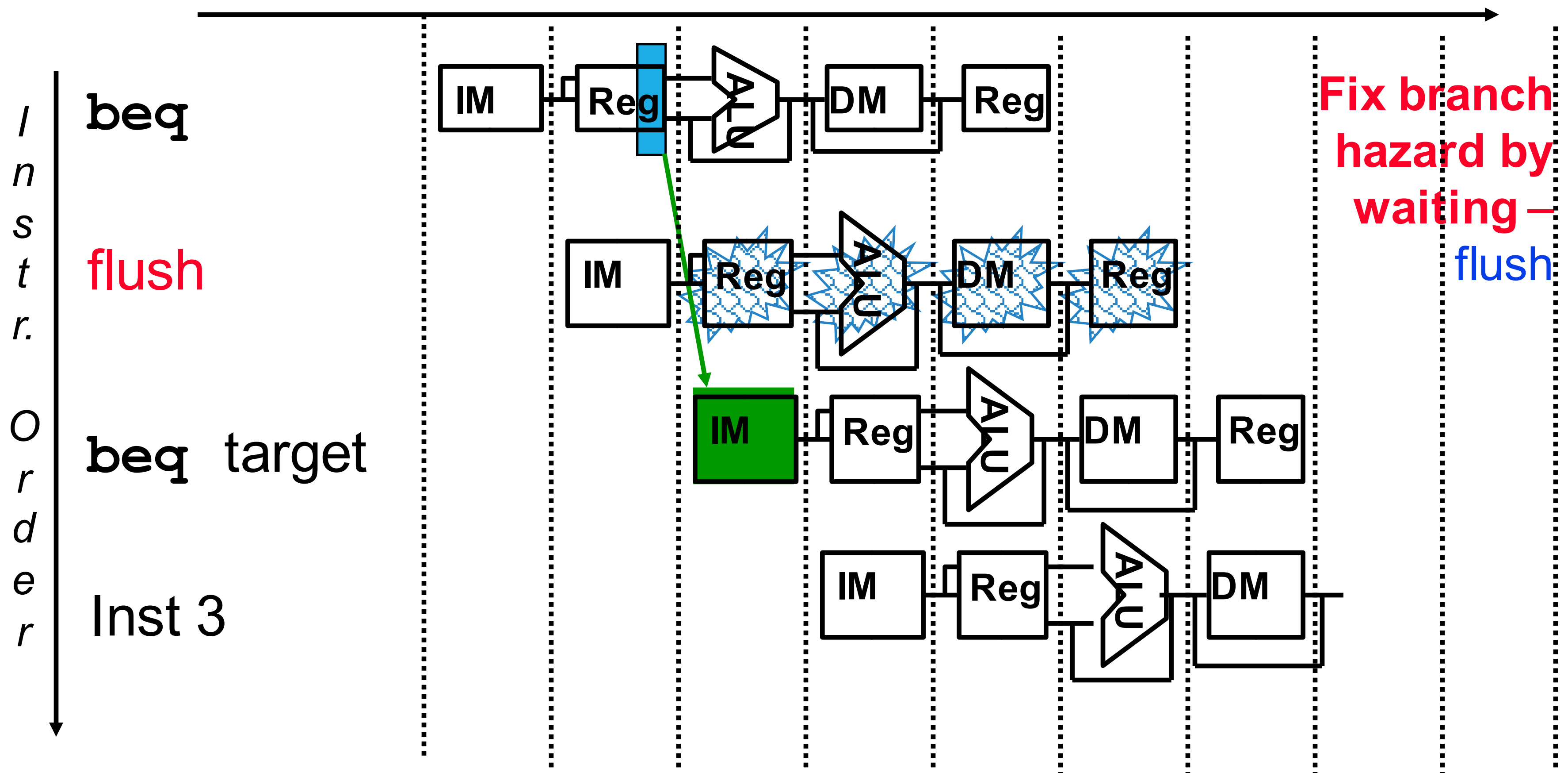


One Way to "Fix" a Branch Control Hazard



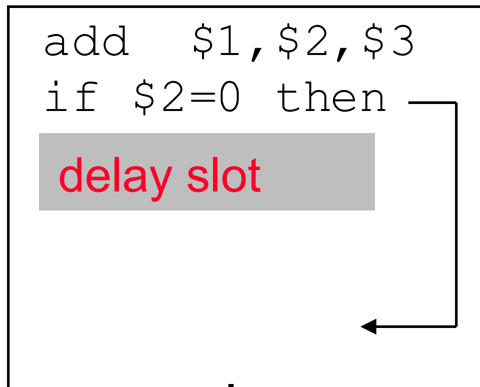
Another Way to "Fix" a Branch Control Hazard

- Move branch decision hardware back to as early in the pipeline as possible – i.e., during the decode cycle

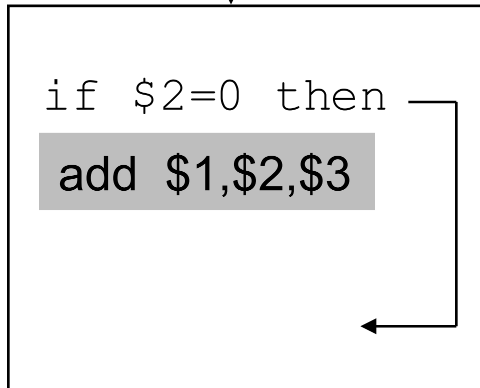


Scheduling Branch Delay Slots

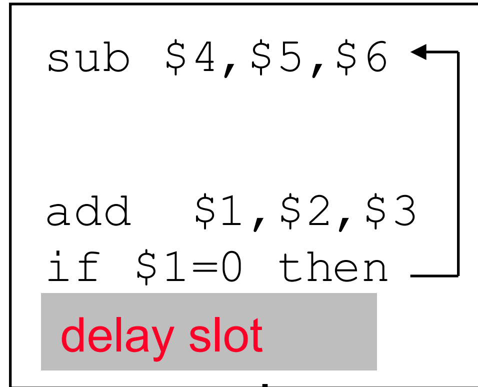
A. From before branch



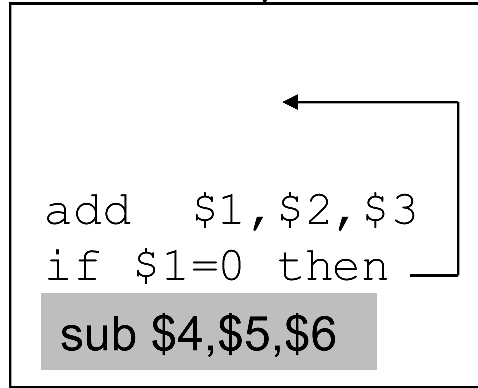
becomes



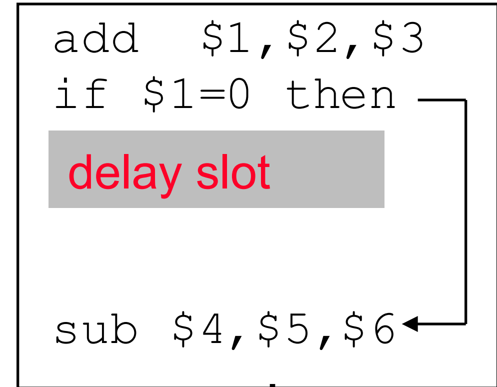
B. From branch target



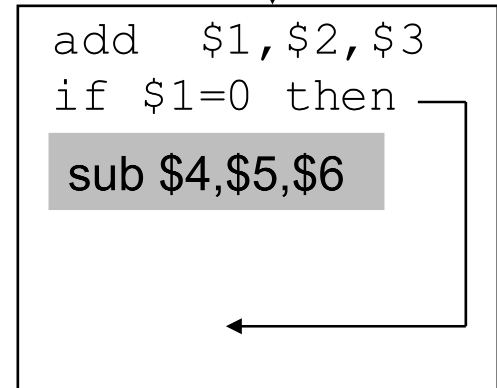
becomes



C. From fall through

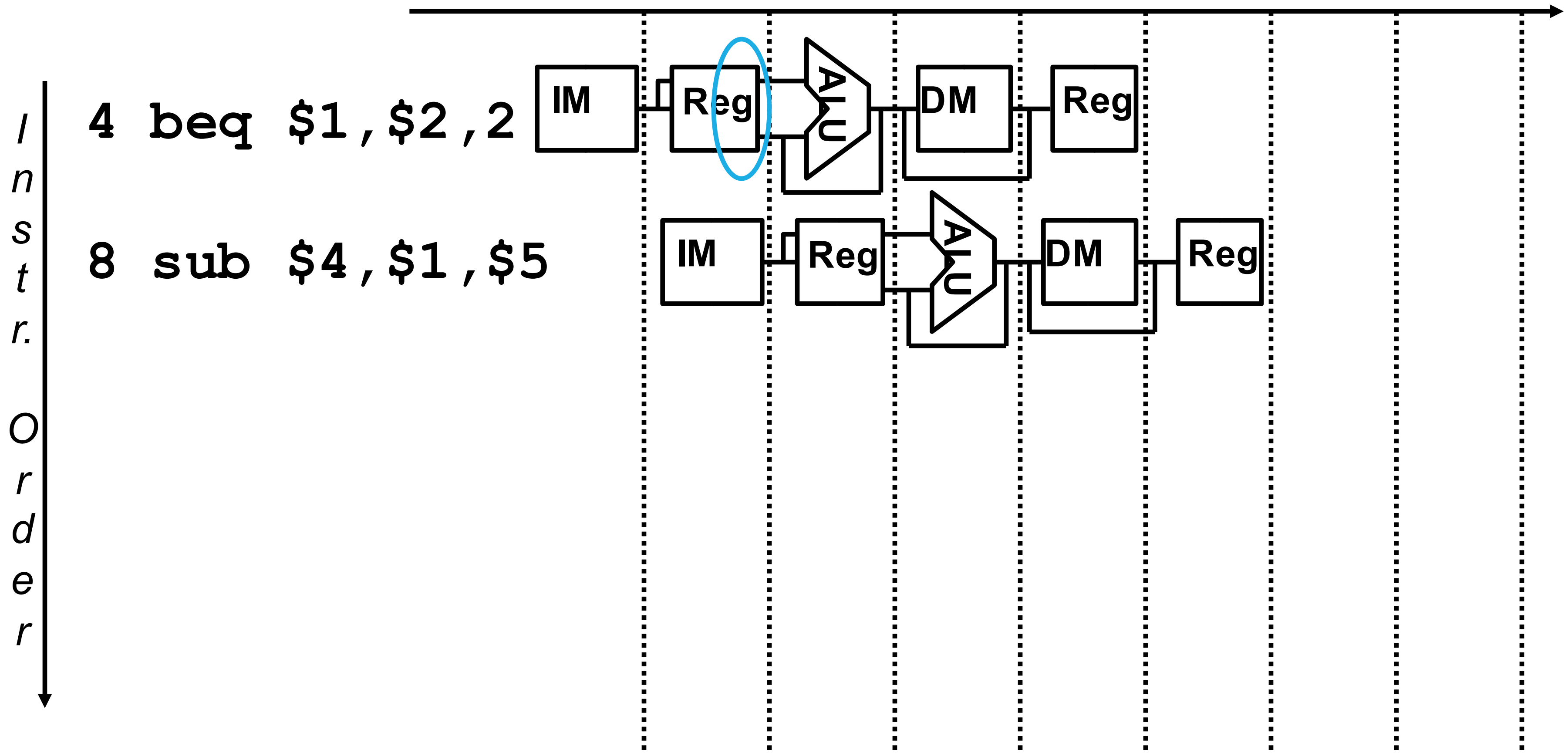


becomes



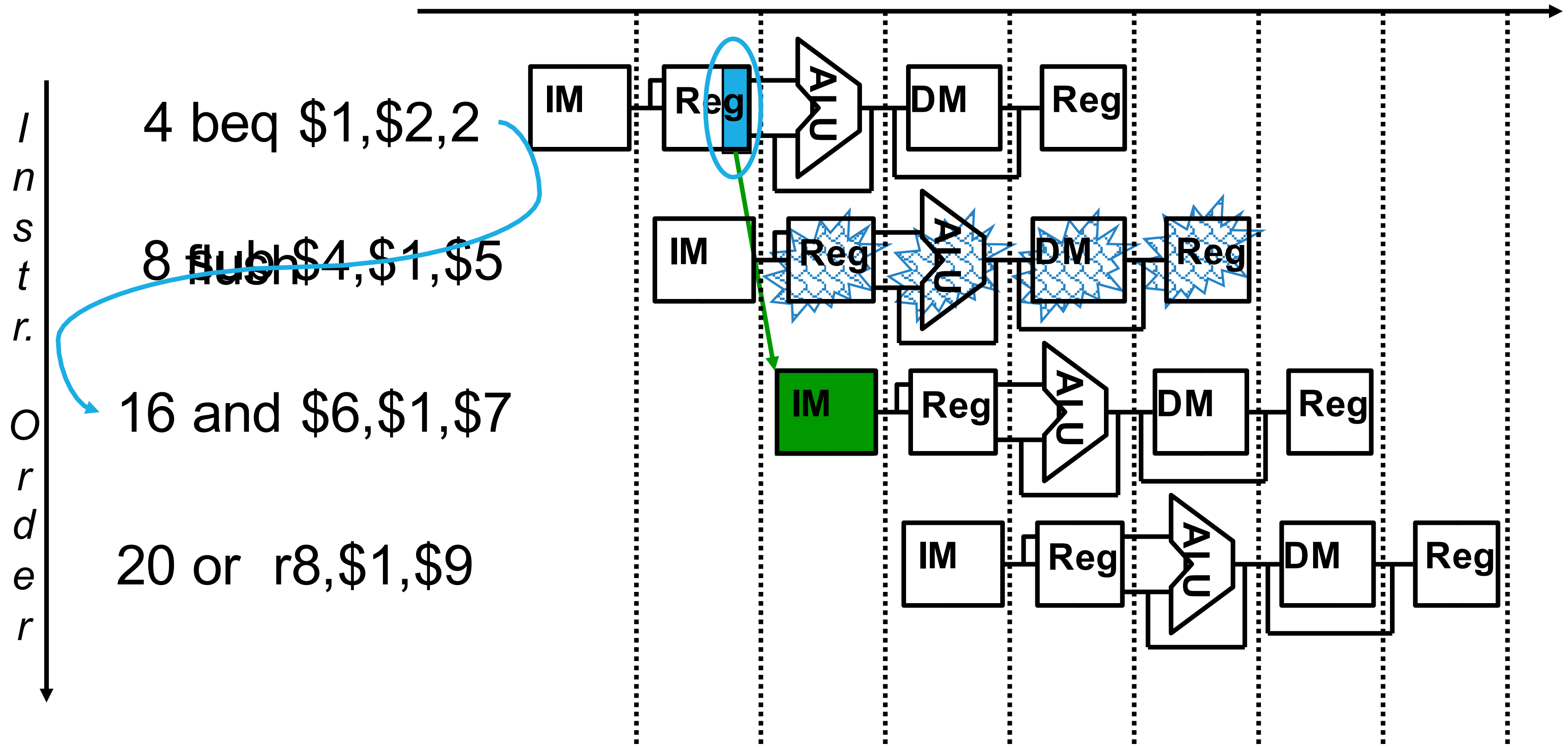
- A is the best choice, fills delay slot and reduces IC
- In B and C, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

Flushing with Misprediction (Not Taken)



- To flush the IF stage instruction, assert `IF.Flush` to zero the instruction field of the IF/ID pipeline register (transforming it into a `nop`)

Flushing with Misprediction (Not Taken)



- To flush the IF stage instruction, assert `IF.Flush` to zero the instruction field of the IF/ID pipeline register (transforming it into a `nop`)

Static Branch Prediction, con't

- Resolve branch hazards by assuming a given outcome and proceeding

2. Predict taken – predict branches will always be taken
   - Predict taken *always* incurs one stall cycle (if branch destination hardware has been moved to the ID stage)
   - Is there a way to "cache" the address of the branch target instruction ??

- As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance. With more hardware, it is possible to try to predict branch behavior dynamically during program execution

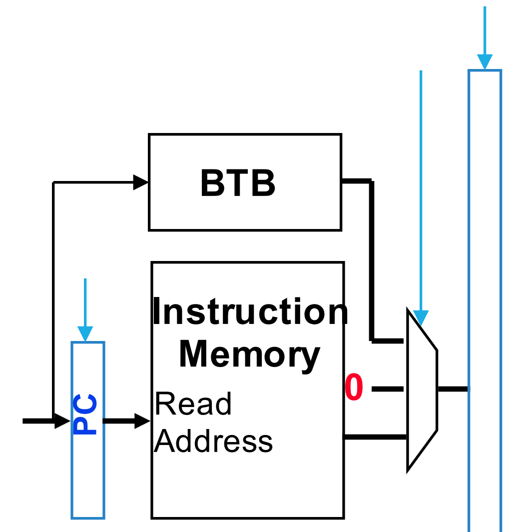
3. Dynamic branch prediction – predict branches at run-time using *run-time* information

Dynamic Branch Prediction

- A branch prediction buffer (aka branch history table (BHT)) in the IF stage addressed by the lower bits of the PC, contains bit(s) passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was execute
  - Prediction bit may predict incorrectly (may be a wrong prediction for this branch this iteration or may be from a different branch with the same low order PC bits) but the doesn't affect correctness, just performance
    - Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit(s)
  - If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit(s)
    - A 4096 bit BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)

Branch Target Buffer

- The BHT predicts *when* a branch is taken, but does not tell *where* its taken to!
  - A branch target buffer (BTB) in the IF stage caches the branch target address, but we also need to fetch the next sequential instruction. The prediction bit in IF/ID selects which "next" instruction will be loaded into IF/ID at the next clock edge
    - Would need a two read port instruction memory
  - Or the BTB can cache the branch taken instruction while the instruction memory is fetching the next sequential instruction



- If the prediction is correct, stalls can be avoided no matter which direction they go

1-bit Prediction Accuracy

- A 1-bit predictor will be incorrect twice when not taken
  - Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code

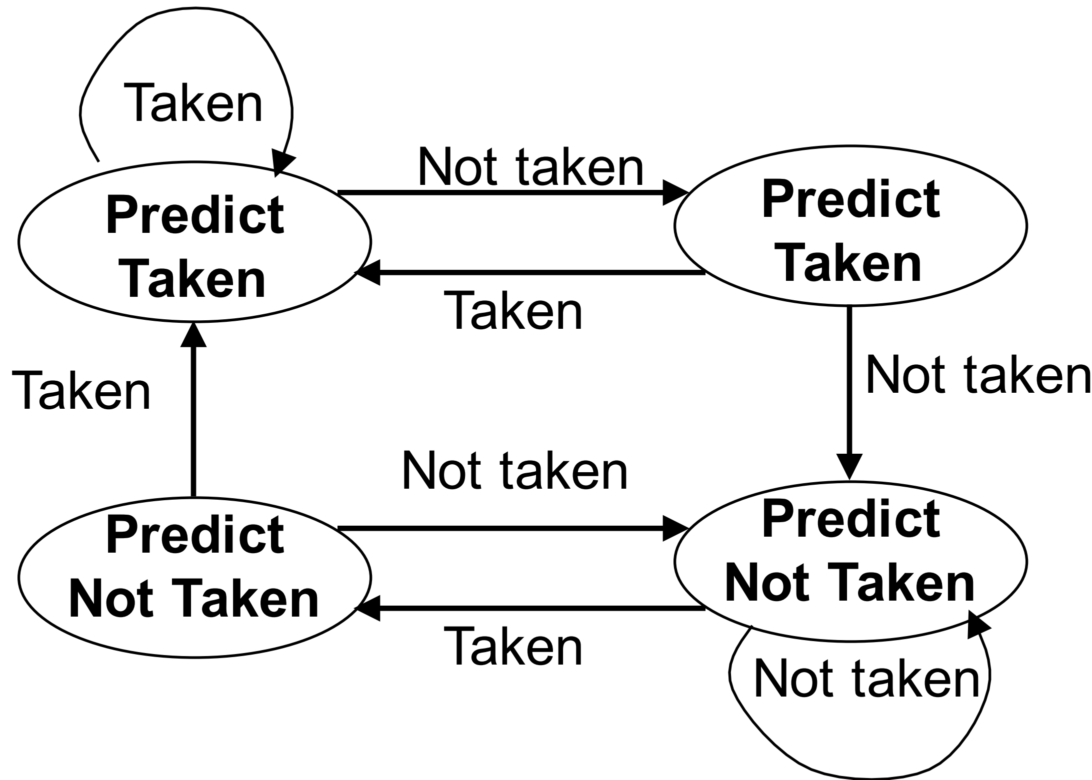
  1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)
  2. As long as branch is taken (looping), prediction is correct
  3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)

```
Loop: 1st loop instr
      2nd loop instr
           .
           .
           .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

2-bit Predictors

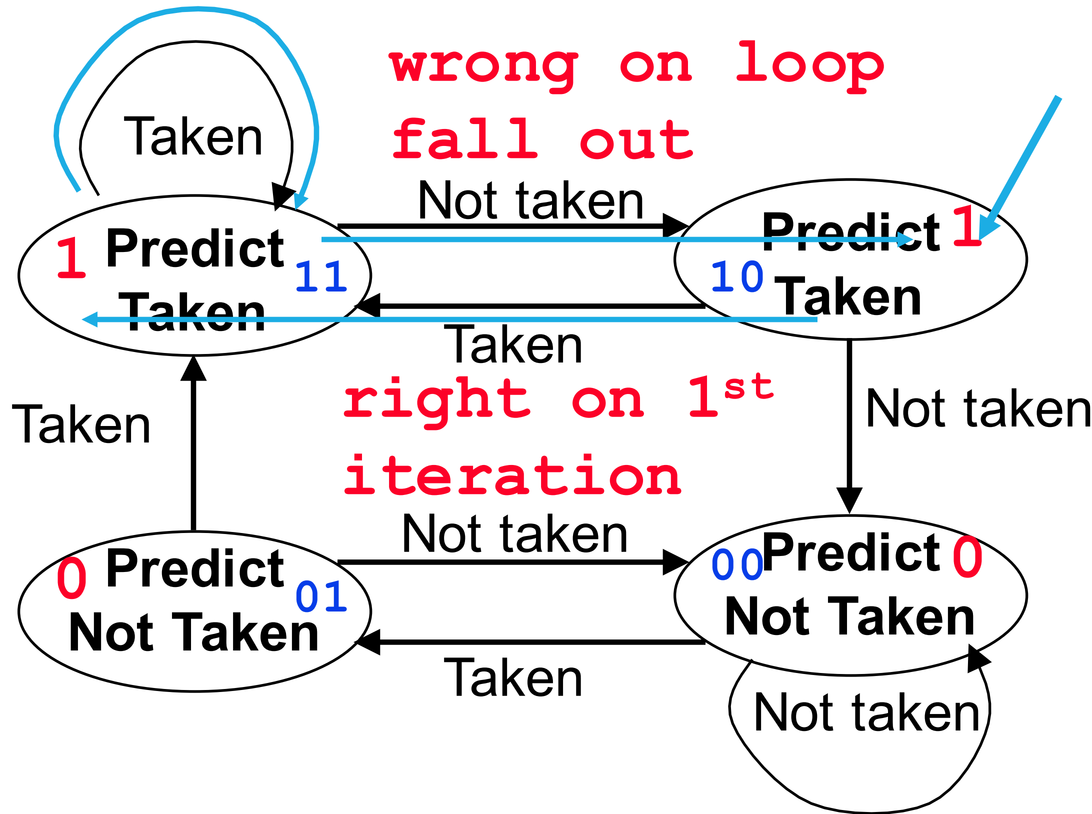
- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed




```
Loop: 1st loop instr
      2nd loop instr
          .
          .
          .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed




```
Loop: 1st loop instr
      2nd loop instr
          .
          .
          .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- BHT also stores the initial FSM state