CMPEN 431
Computer Architecture
Fall 2018

Hazards in the Pipelined Processor

Jack Sampson(www.cse.psu.edu/~sampson)

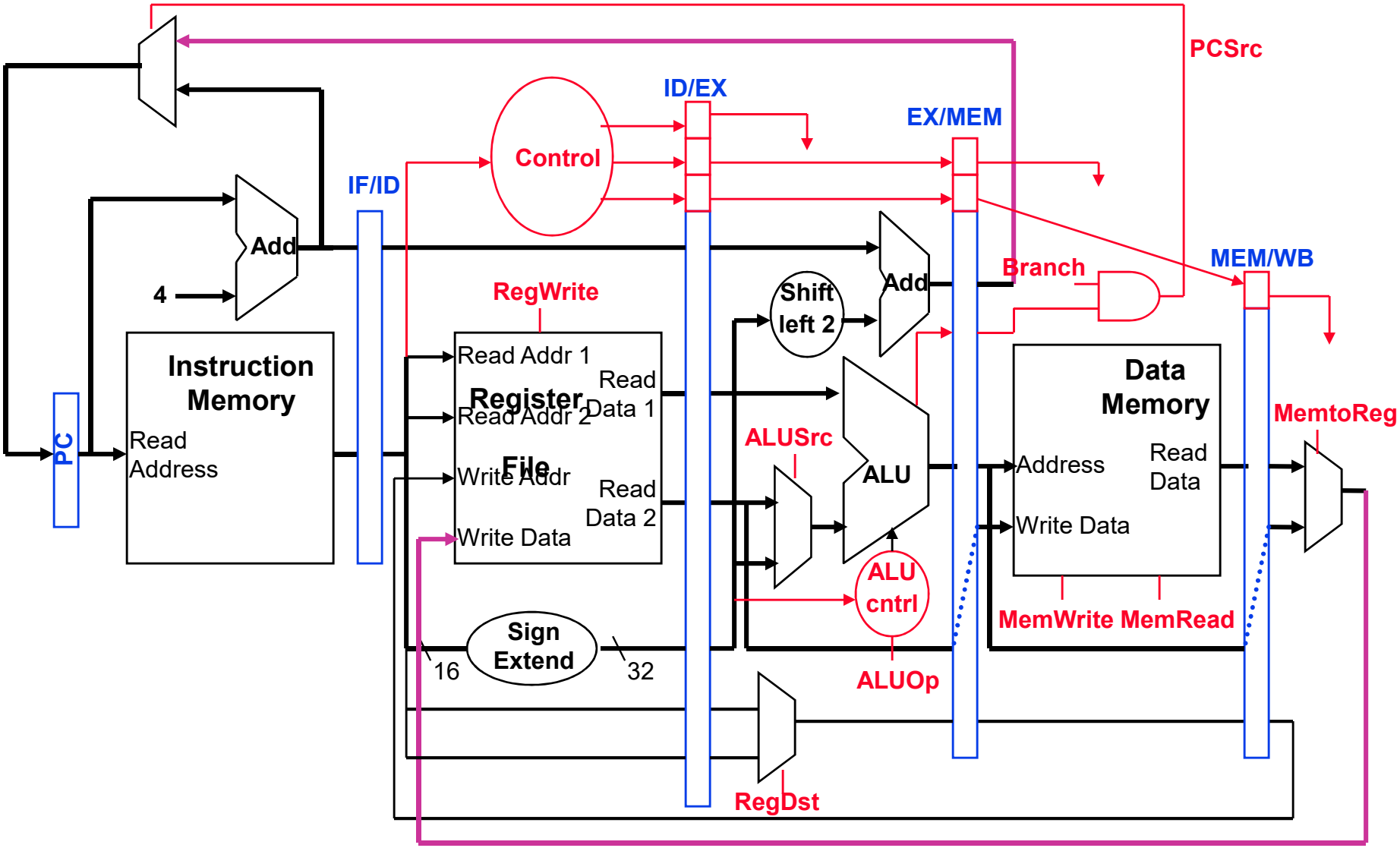
[Slides adapted from work by Mary Jane Irwin, in turn adapted from *Computer Organization and Design, Revised 4th Edition*,

Patterson & Hennessy, © 2011, Morgan Kaufmann]

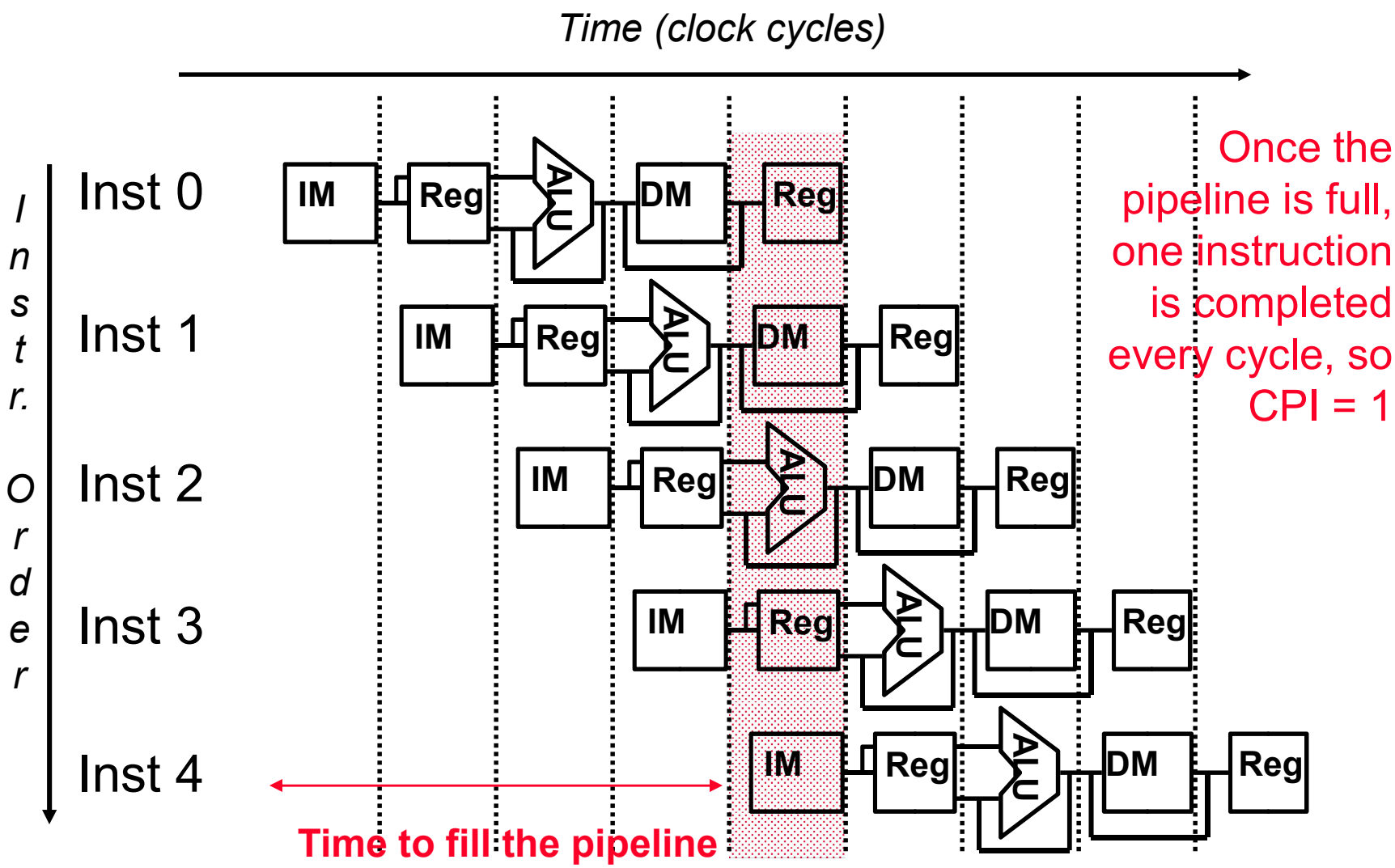
Reminders

- This week
  - Detecting and resolving dependence hazards in a pipelined MIPS pipelined datapath
- Next week
  - Exam 1
  - Then…
    - Reducing branch hazard costs
    - Branch prediction; dealing with exceptions

Review: MIPS Pipeline Data and Control Paths



Review: Why Pipeline? For Performance(& Utilization)!



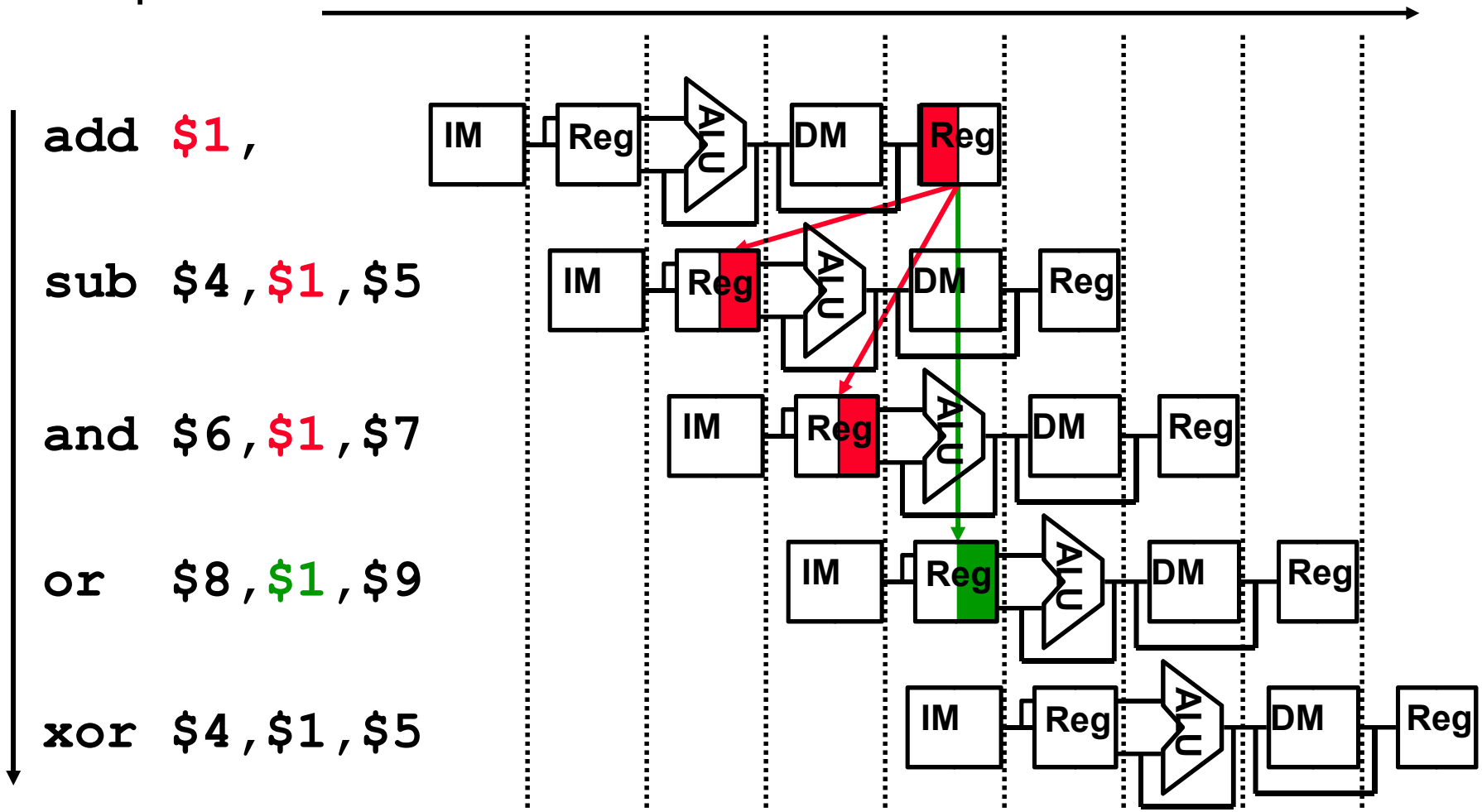
Review: Pipelining - What Makes it Hard ?

- Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch and jump instructions, exceptions

- Pipeline hardware control must detect the hazard and then take action to resolve hazard

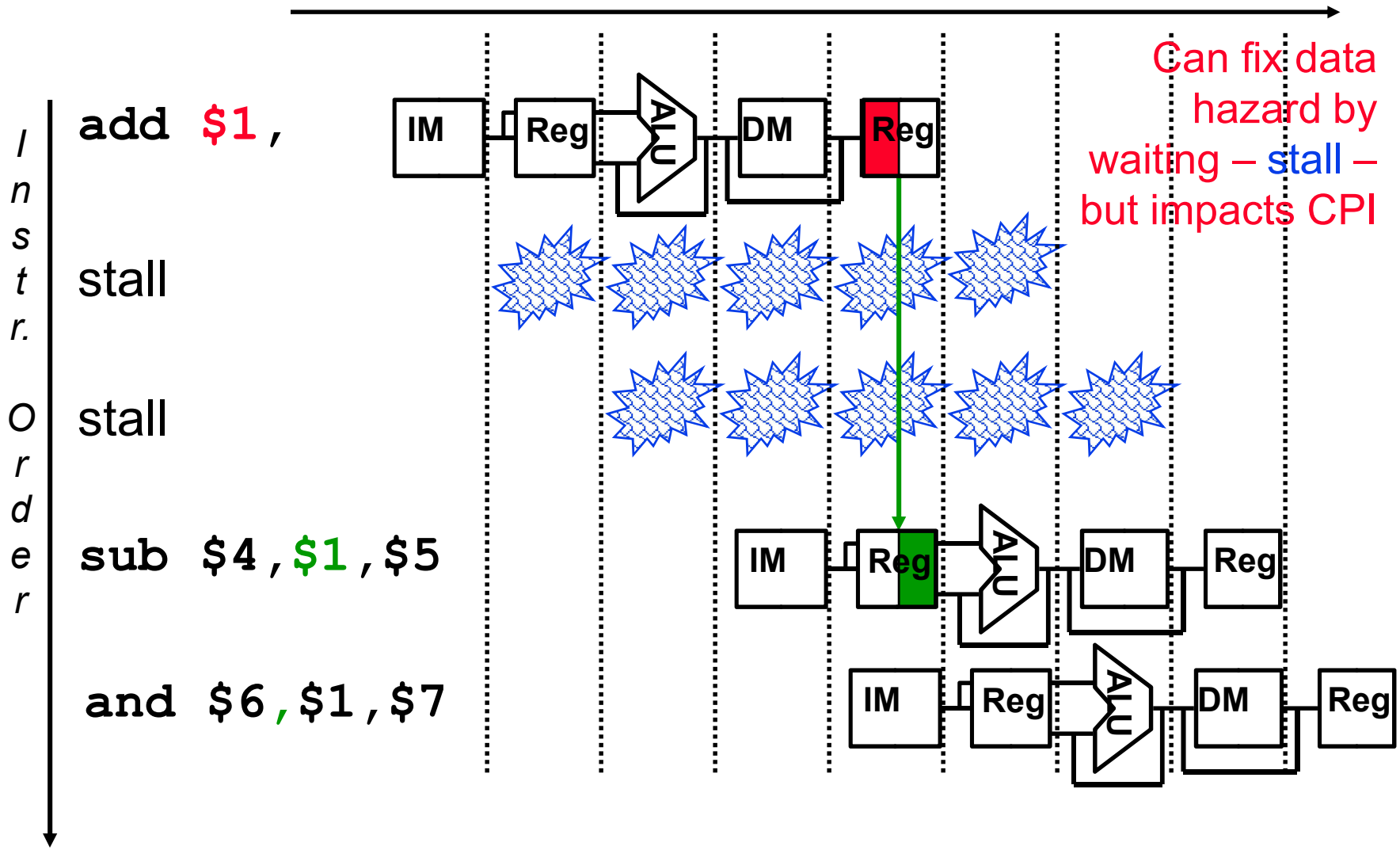
Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards

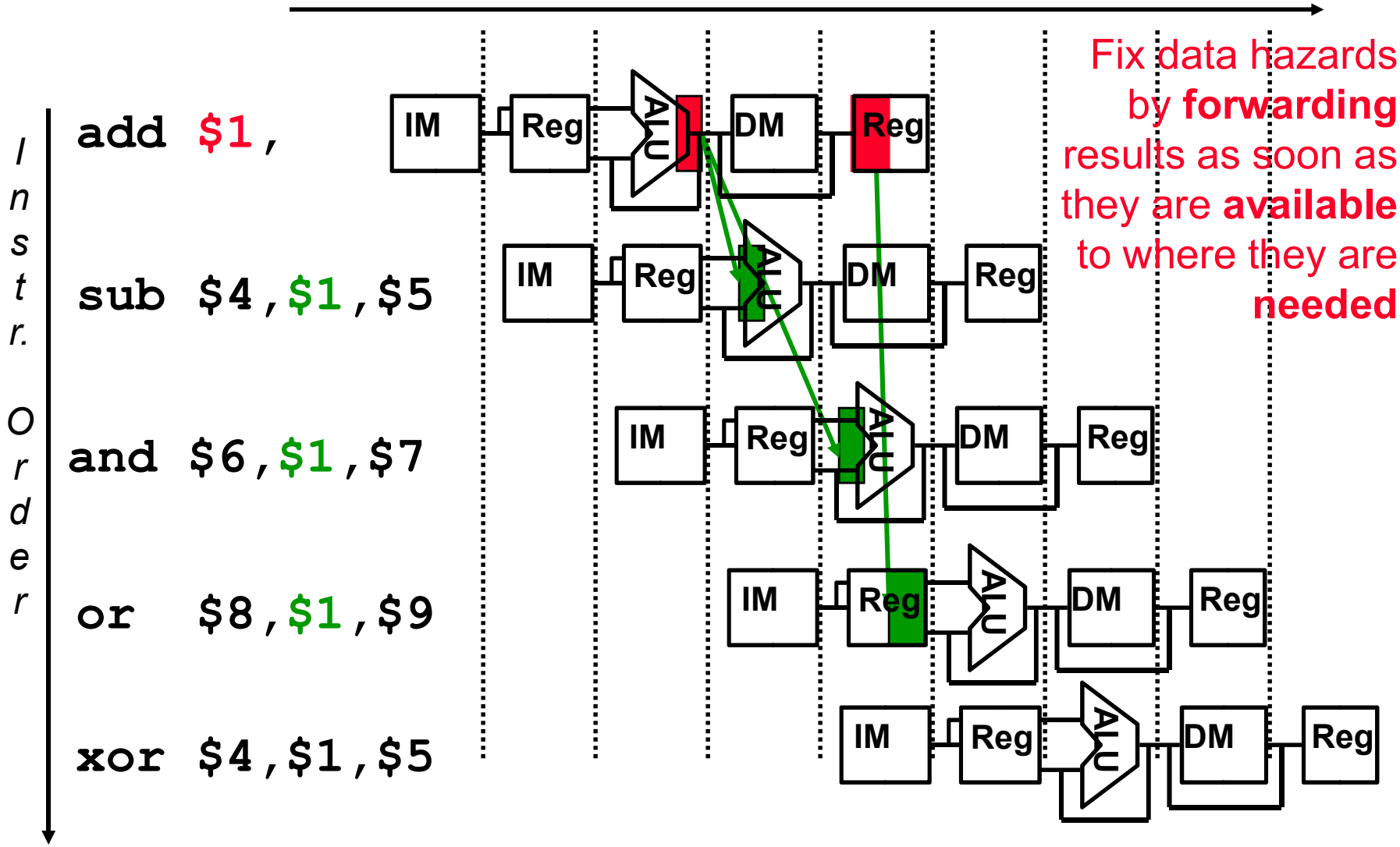


- Read before write data hazard

One Way to "Fix" a Data Hazard



Another Way to "Fix" a Data Hazard

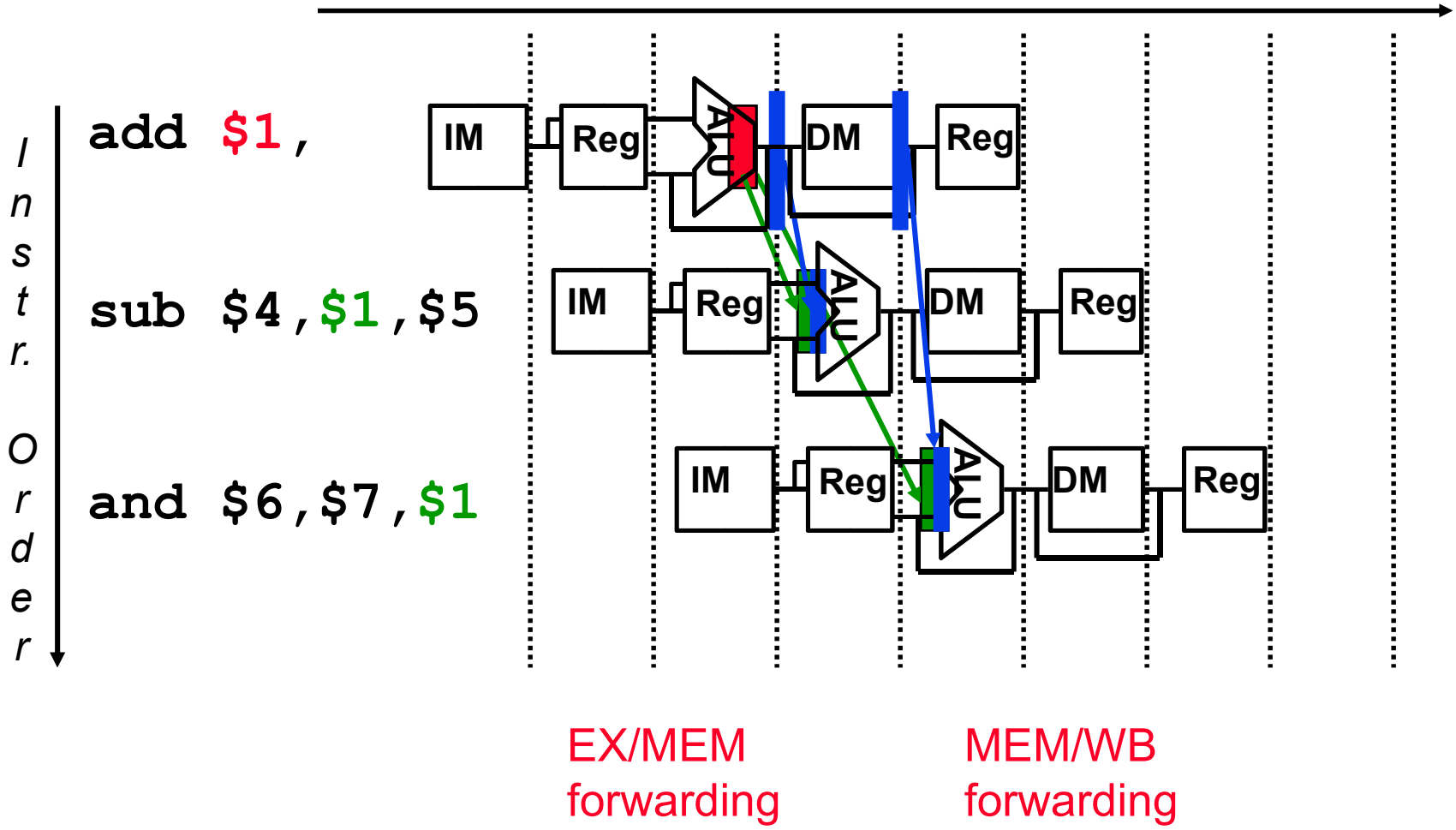


Fix data hazards by **forwarding** results as soon as they are **available** to where they are **needed**

Data Forwarding (aka Bypassing)

- Take the result from a downstream pipeline state register that holds the needed data that cycle and forward it to the functional units (e.g., the ALU) that need that data that cycle
- For ALU functional unit: the inputs can come from other pipeline registers than just ID/EX by
  - adding multiplexors to the inputs of the ALU
  - connecting the Rd write data in EX/MEM or MEM/WB to either (or both) of the EX's stage Rs and Rt ALU mux inputs
  - adding the proper control hardware to control the new muxes
- Other functional units may need similar forwarding logic (e.g., the DM)
- With forwarding can achieve a CPI of 1 even in the presence of data dependencies

Correct Forwarding Illustration



Data Forwarding Control Conditions

1. EX Forward Unit:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU

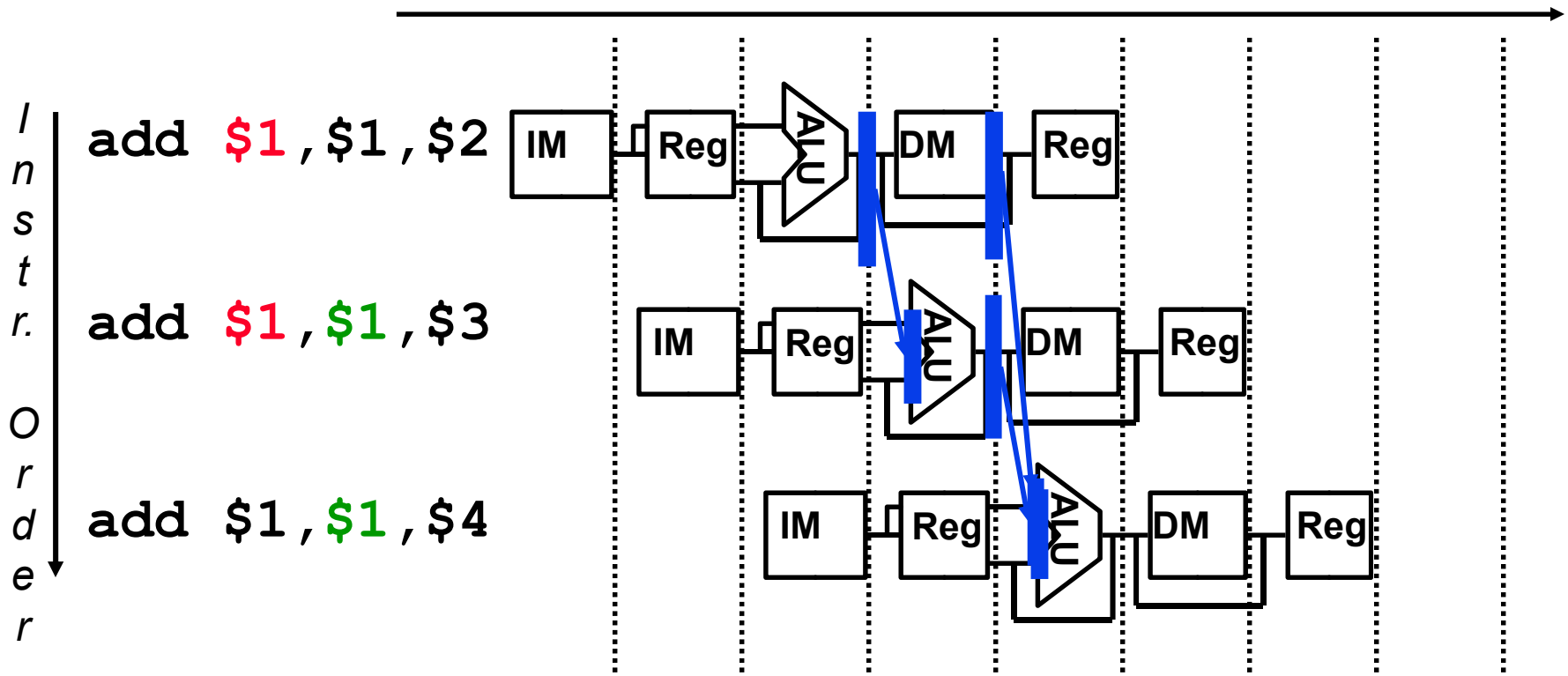
2. MEM Forward Unit:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

Forwards the result from the second previous instr. to either input of the ALU

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?



Corrected Data Forwarding Control Conditions

1. EX Forward Unit:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU

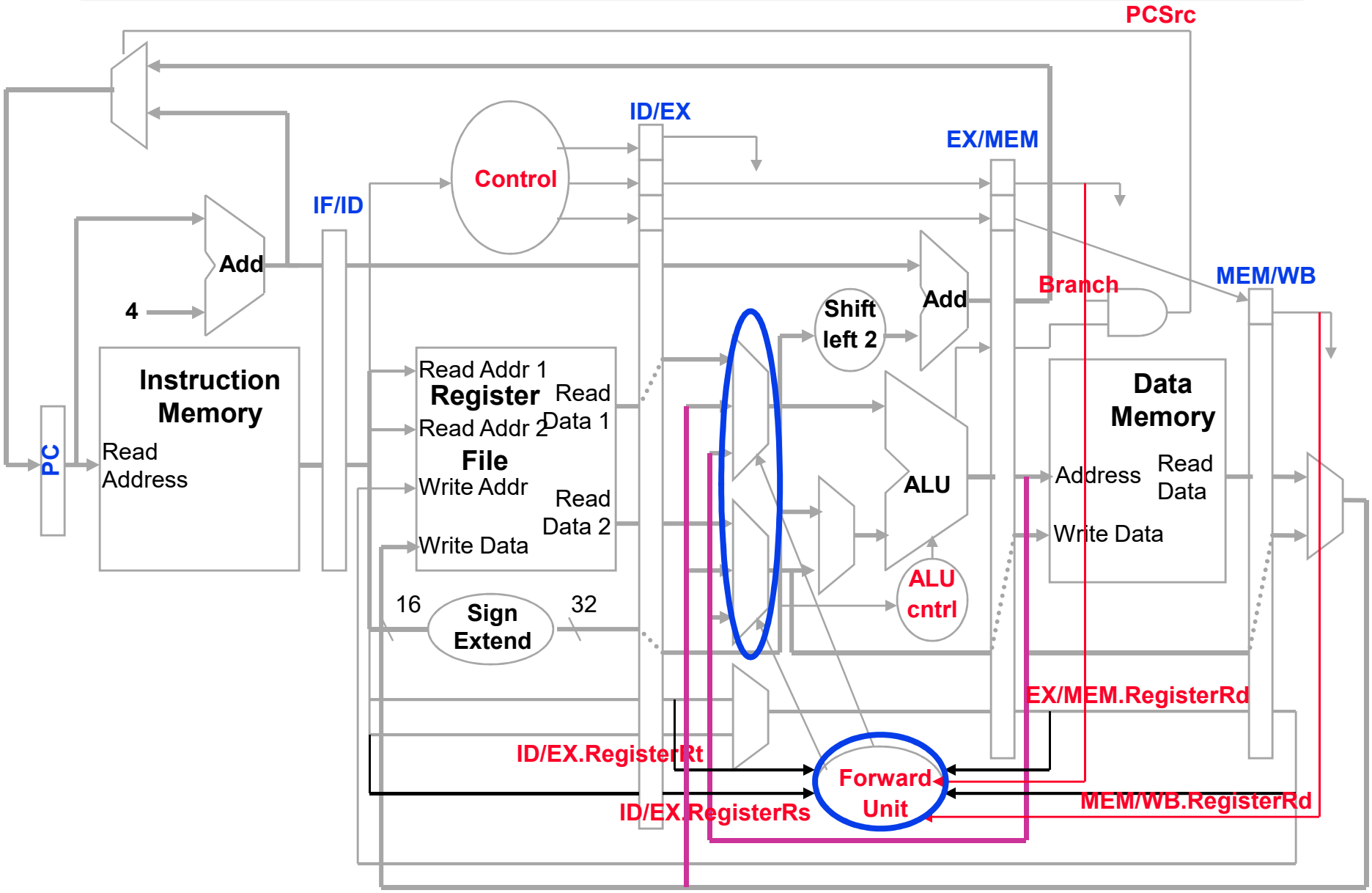
2. MEM Forward Unit:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

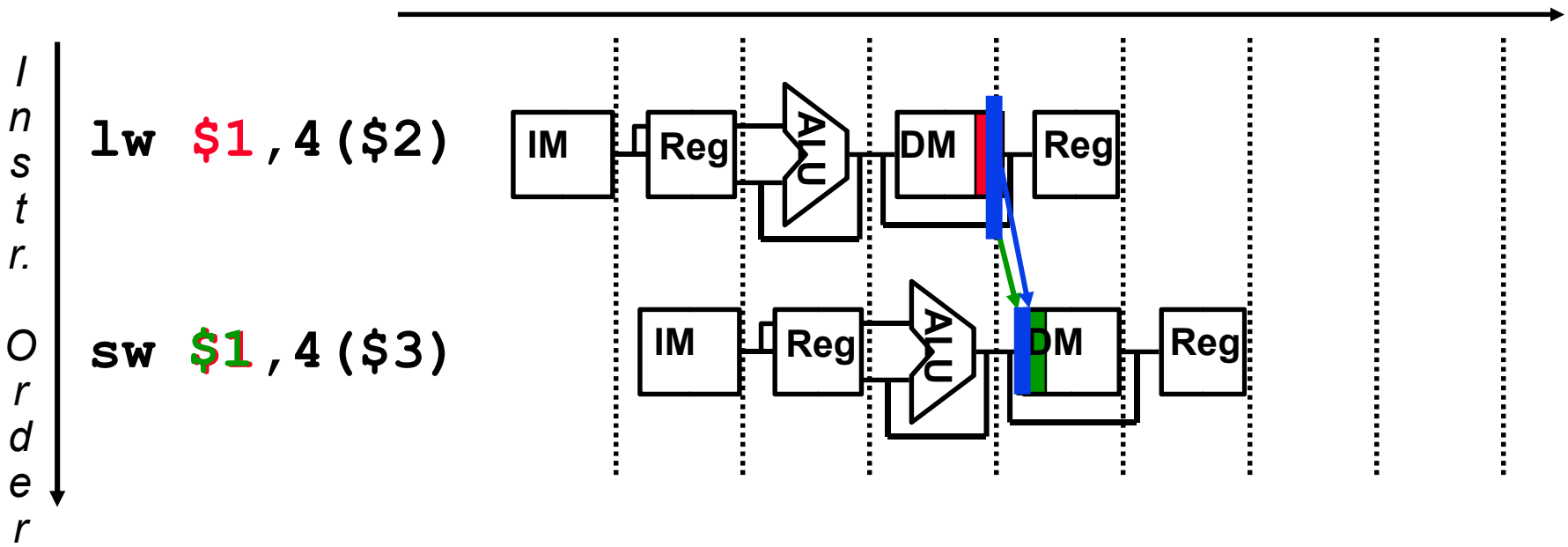
Forwards the result from the previous or second previous instr. to either input of the ALU

Datapath with Forwarding Hardware



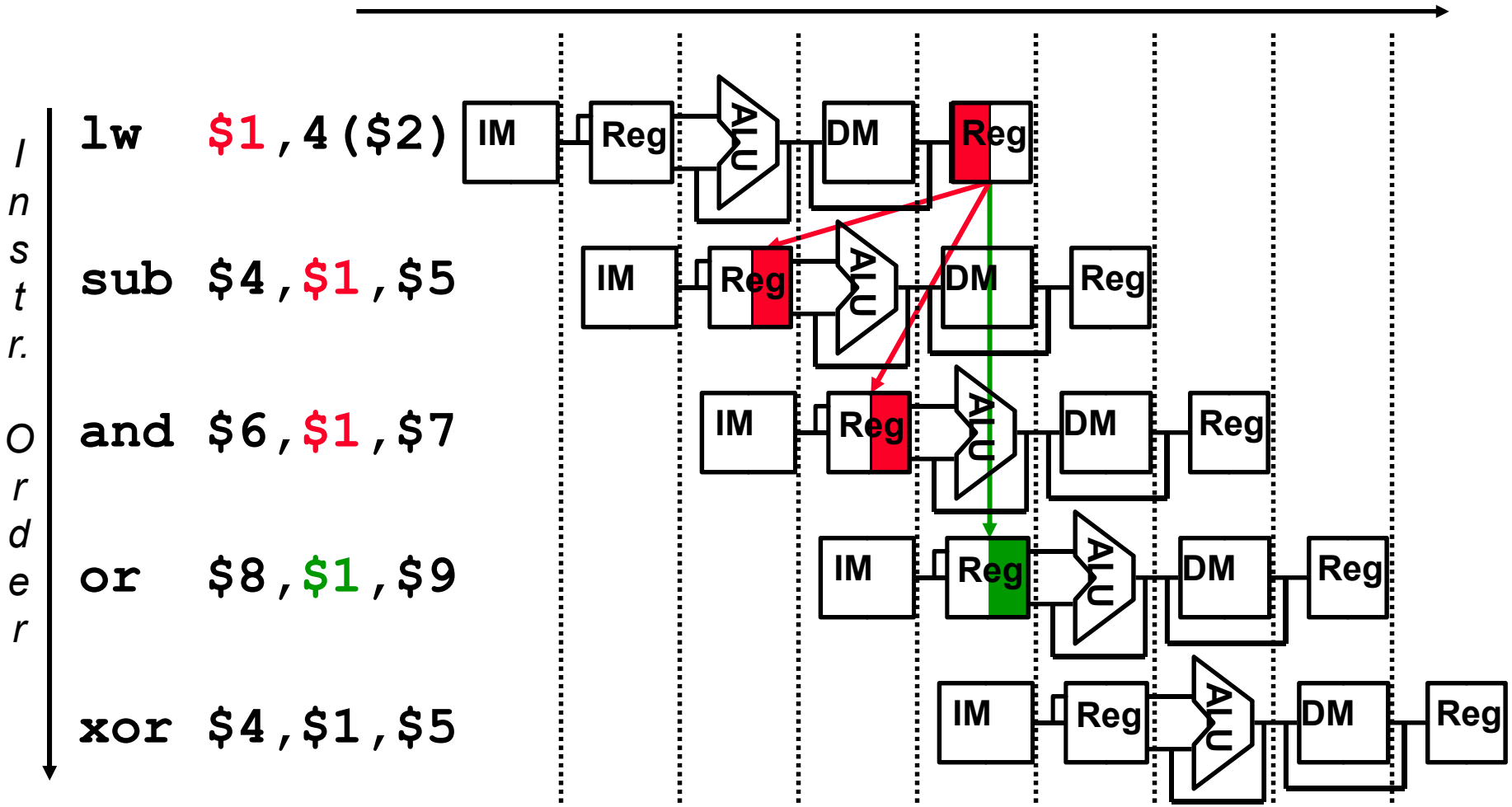
Memory-to-Memory Copies

- 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



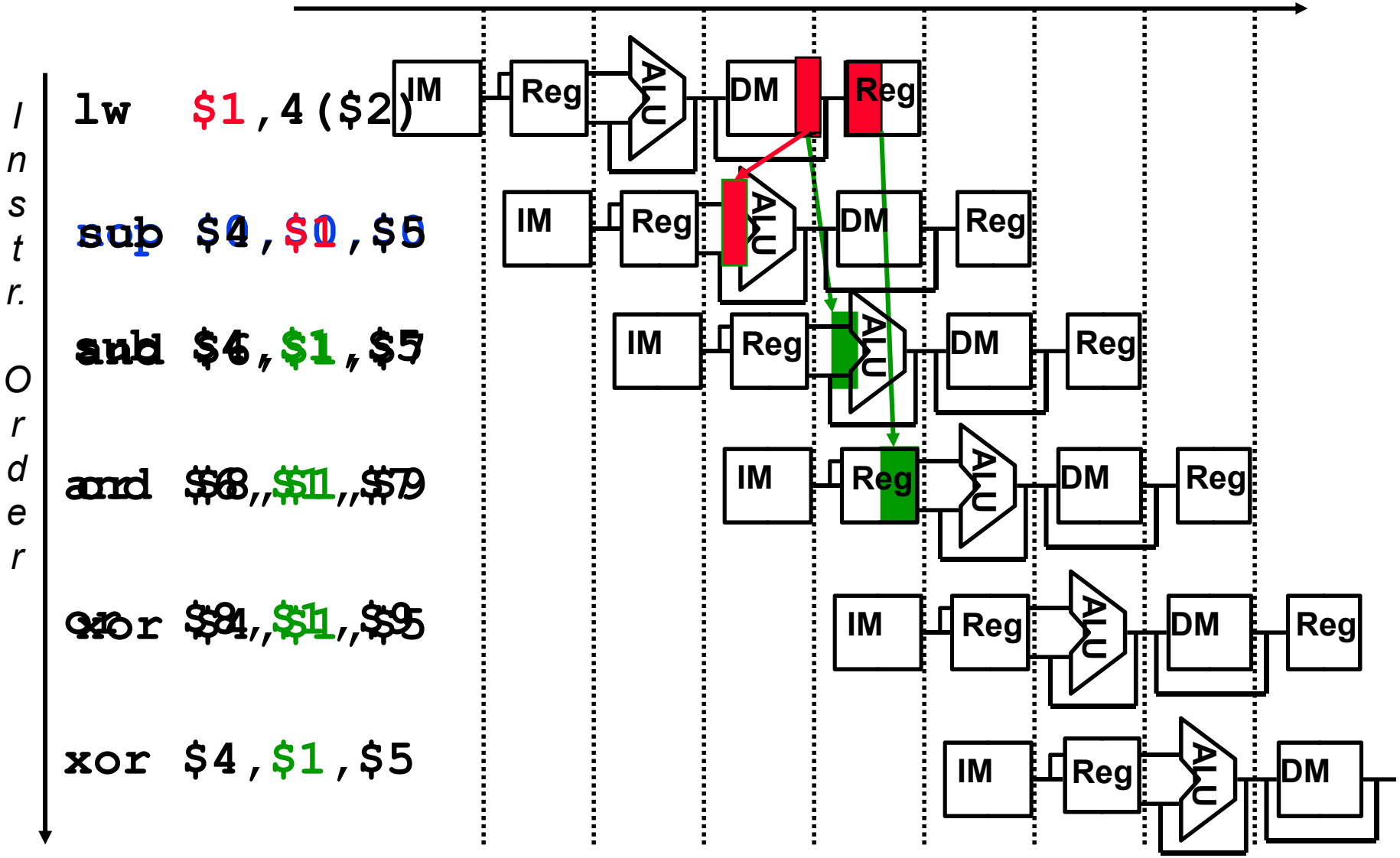
Loads Can Cause Data Hazards

- Dependencies backward in time cause hazards



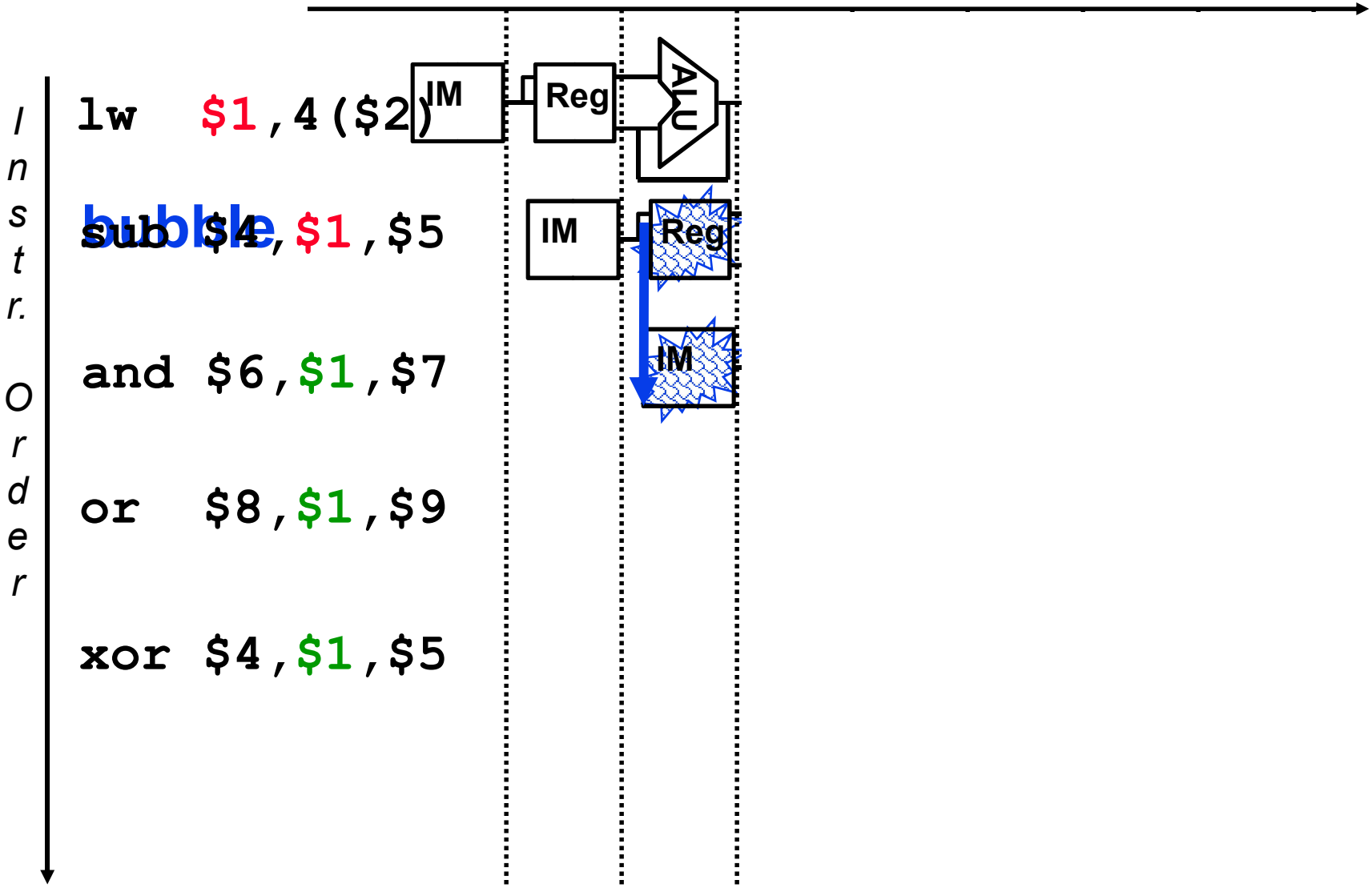
- Load-use data hazard

Avoiding Load-use Data Hazards (statically)



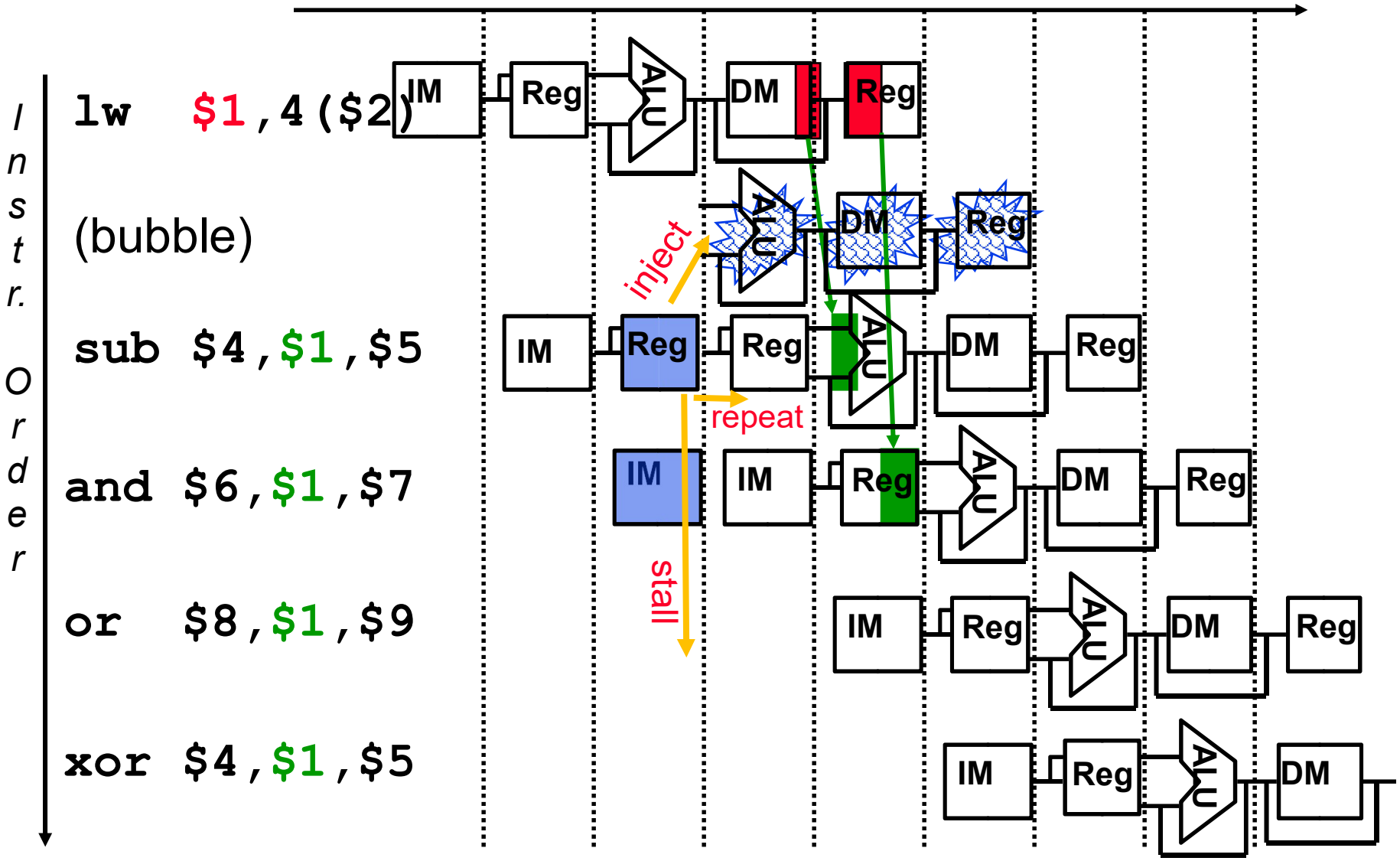
- Will still need to adjust by one cycle *even* with forwarding

Avoiding Load-use Data Hazards (dynamically)



- Introduce a "bubble" in the pipeline

Avoiding Load-use Data Hazards (dynamically)



- Introduce a "bubble" in the pipeline

Load-use Data Hazard Detection Unit

- Need a Hazard detection Unit in the ID stage that inserts a stall between the load and its use

  1. ID Hazard detection Unit:

```
if (ID/EX.MemRead
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
or  (ID/EX.RegisterRt = IF/ID.RegisterRt)))
stall the pipeline
```

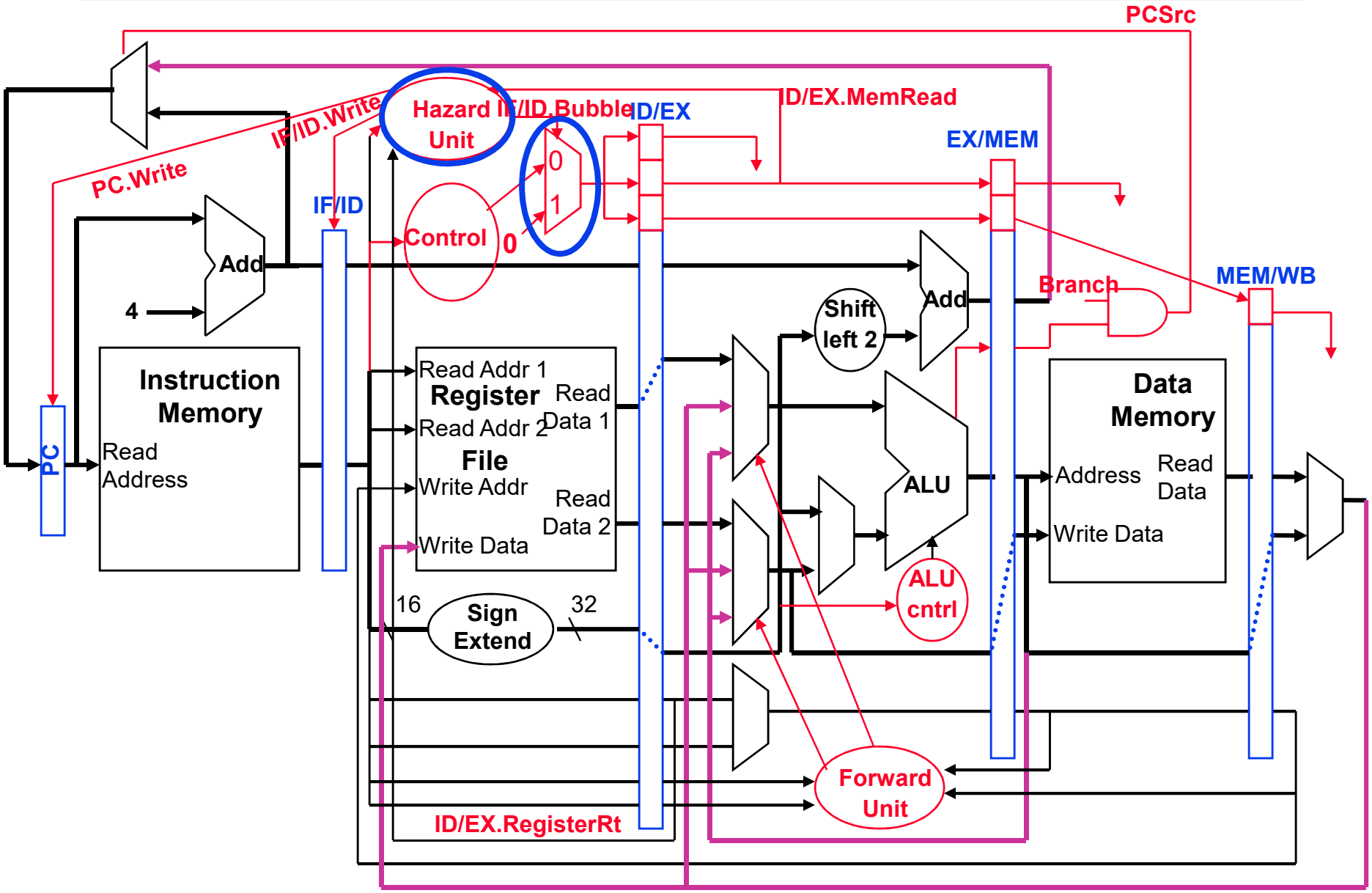
- The first line tests to see if the instruction now in the EX stage is a `lw`; the next two lines check to see if the destination register of the `lw` matches either source register of the instruction in the ID stage (the load-use instruction)
- After this one cycle stall, the forwarding logic can handle the remaining data hazards

Data Hazard/Stall Hardware

Along with the Hazard Unit, we have to *implement* the stall

1. Prevent the instructions in the IF and ID stages from progressing down the pipeline – done by preventing the PC register and the IF/ID pipeline register from changing
   - Hazard detection Unit controls the writing of the PC (`PC.write`) and IF/ID (`IF/ID.Write`) registers
2. Insert a "bubble" between the `lw` instruction (in the EX stage) and the load-use instruction (in the ID stage) (i.e., insert a `noop` with `IF/ID.Bubble`)
   - Set the control bits in the EX, MEM, and WB control fields of the ID/EX pipeline register to 0 (`noop`). The Hazard Unit controls the mux that chooses between the real control values and the 0's.
3. Let the `lw` instruction and the instructions after it in the pipeline (before it in the code) proceed normally down the pipeline

Adding the Data Hazard/Stall Hardware



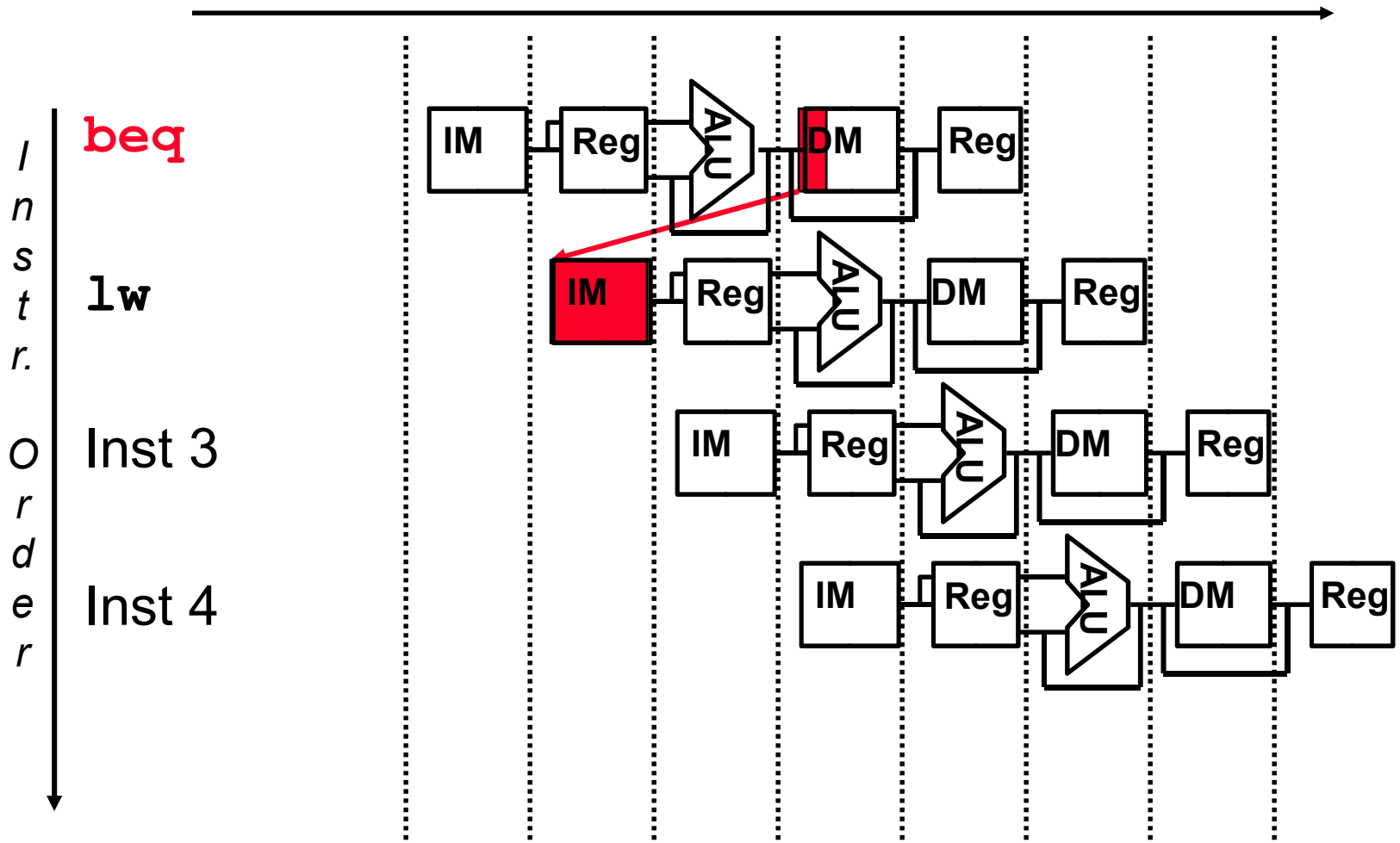
Review: Pipelining - What Makes it Hard ?

- Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch and jump instructions, exceptions

- Pipeline hardware control must detect the hazard and then take action to resolve hazard

Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards



- Which instruction is executed after the branch ?

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 (Branch, Jump) Hazards

- What do we need to know?
  - Next instruction target address, maybe sequential (PC+4) *or*
    - `beq`, PC+4 + sign-extended offset (16 bits)
    - `j`, `jal`, constant address field read from IM during IF (26 bits)
    - `jr`, `jalr`, read from RF during ID (32 bits)
    - `trap` instruction or exception, obtained from table lookup in the OS (32 bits)
  - Branch decision outcome (ALU zero flag)
    - continue sequentially? or jump to the branch target address?
- When do we need to know it?
  - As early as possible in the pipeline
    - if in the ID stage, at worst one stall cycle for the MIPS pipeline

Control (Branch, Jump) Hazards, con't

- When do we act on the decision?
  - As soon as possible
    - we decided too late? We already fetched another instruction, and may need to discard it (flush it)
      - Maybe will have to flush more than one instruction?
    - we guessed, and were right – this is good
    - we guessed, and were wrong – now need to fix things
  - Guesses require an evaluation of the success rate, by simulation (before the fact) or by measurement (after the fact)
    - can measurement improve the quality of the guesses?
    - recent history is often a reasonable predictor of the near future

Summary

- All modern day processors use pipelining for performance
  - Goal: a CPI of 1(/issue-width) and fast a CC
- Pipeline clock rate limited by slowest pipeline stage – so designing a balanced pipeline is important
- Must detect and resolve hazards
  - Structural hazards – resolved by designing the pipeline correctly
  - Data hazards
    - Stall (impacts CPI)
    - Forward (requires hardware support)
  - Control hazards – put the branch decision hardware in as early a stage in the pipeline as possible
    - Stall (impacts CPI)
    - Delay decision (requires compiler support)
    - Static and dynamic prediction (requires hardware support)
- Pipelining complicates exception handling