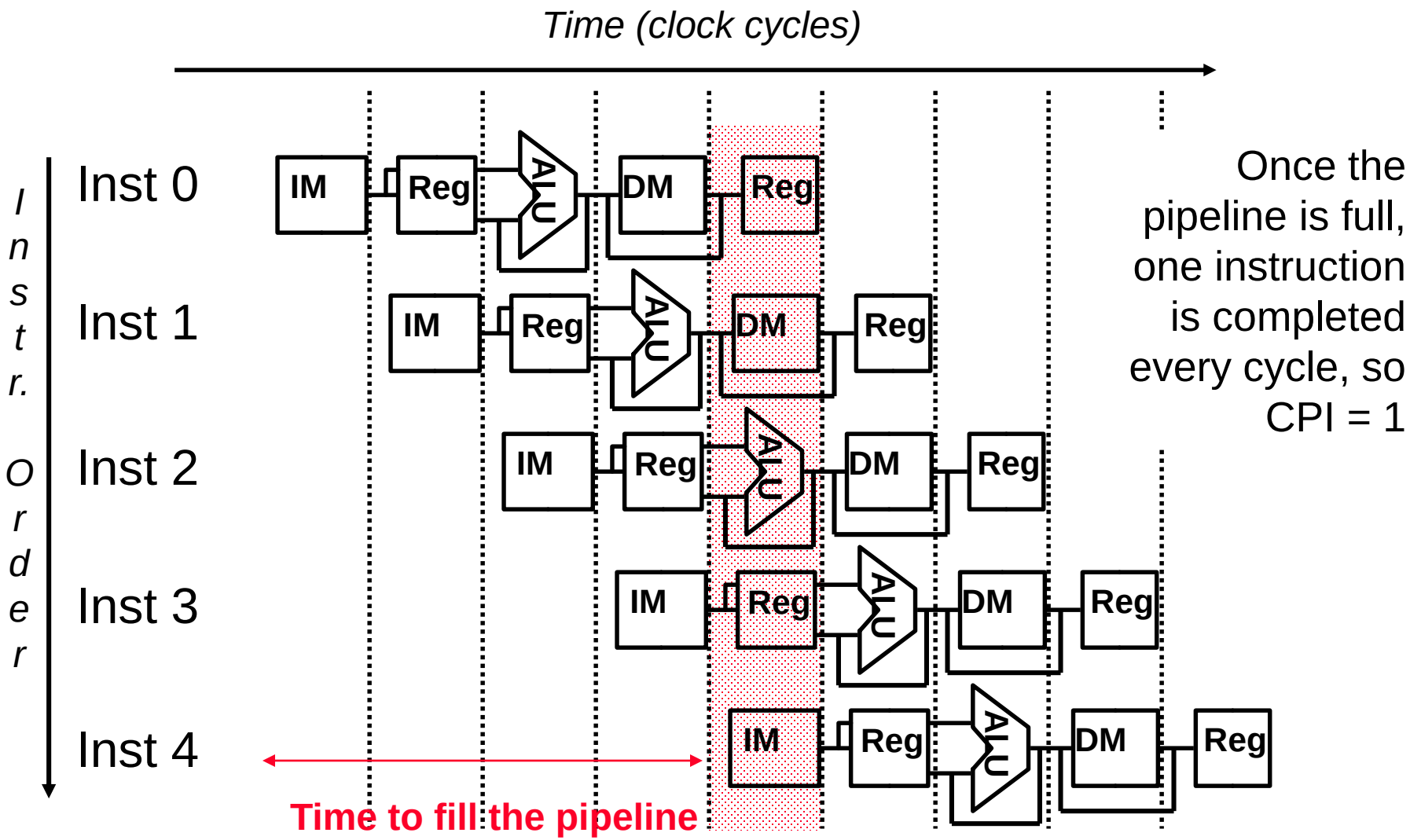
CSE 431
Computer Architecture
Fall 2017

The Pipelined Processor
Part B

Mahmut Taylan Kandemir (www.cse.psu.edu/~kandemir)

[Adapted from *Computer Organization and Design, 5th Edition*,
Patterson & Hennessy, © 2014, Morgan Kaufmann]

Review: Why Pipeline? For Performance!



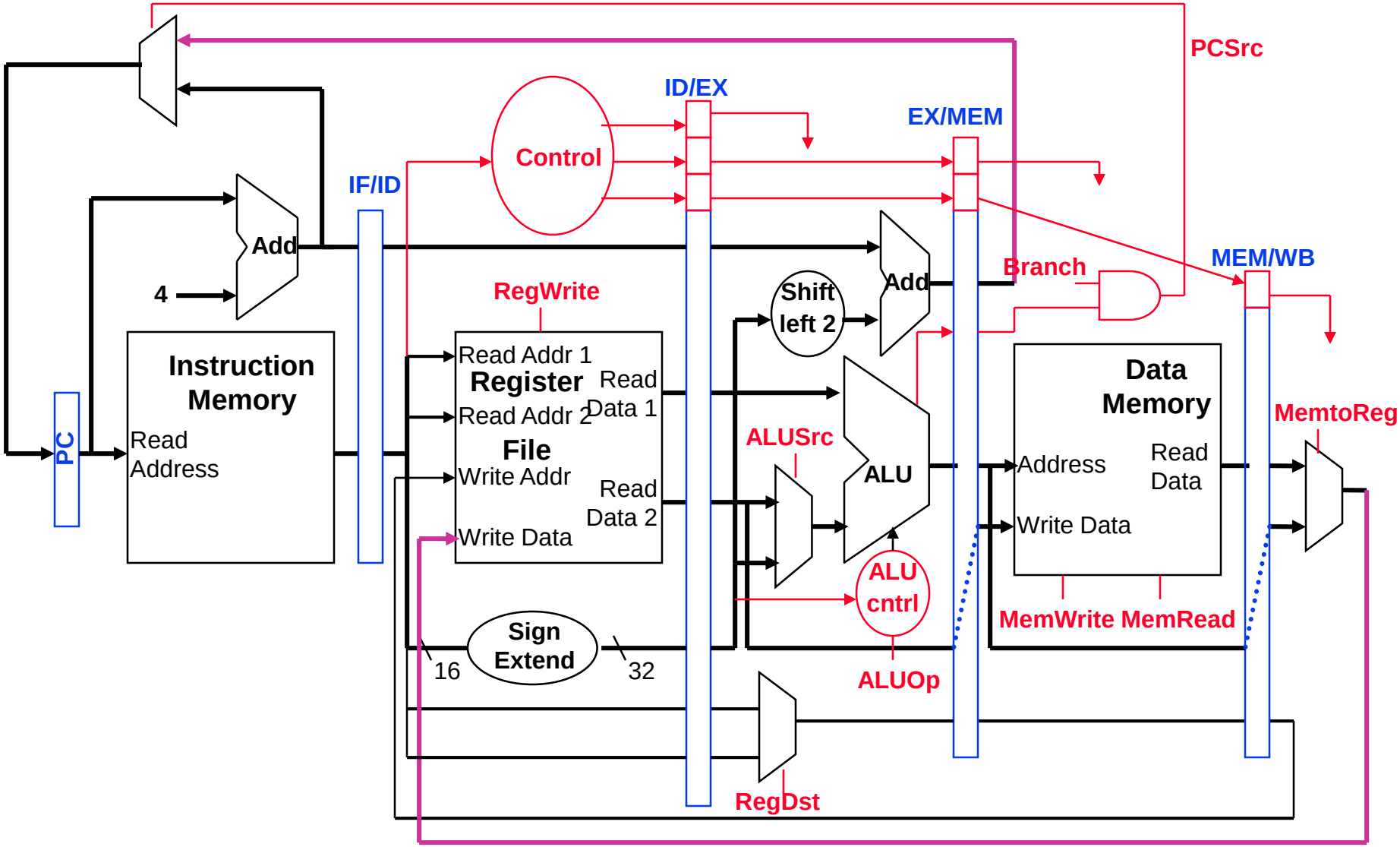
Once the pipeline is full, one instruction is completed every cycle, so CPI = 1

Review: Pipelining - What Makes it Hard ?

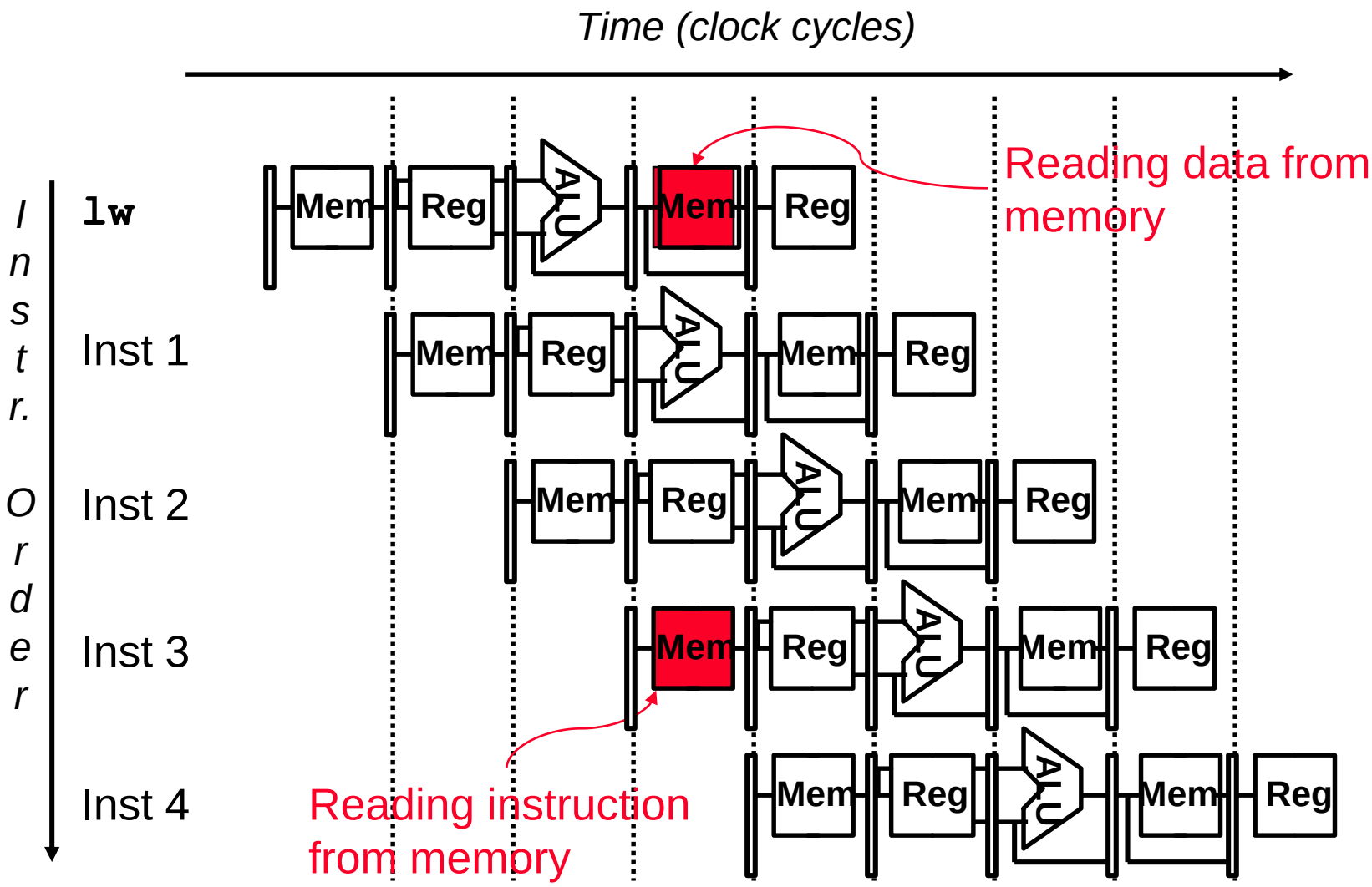
- Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt 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

Review: MIPS Pipeline Data and Control Paths

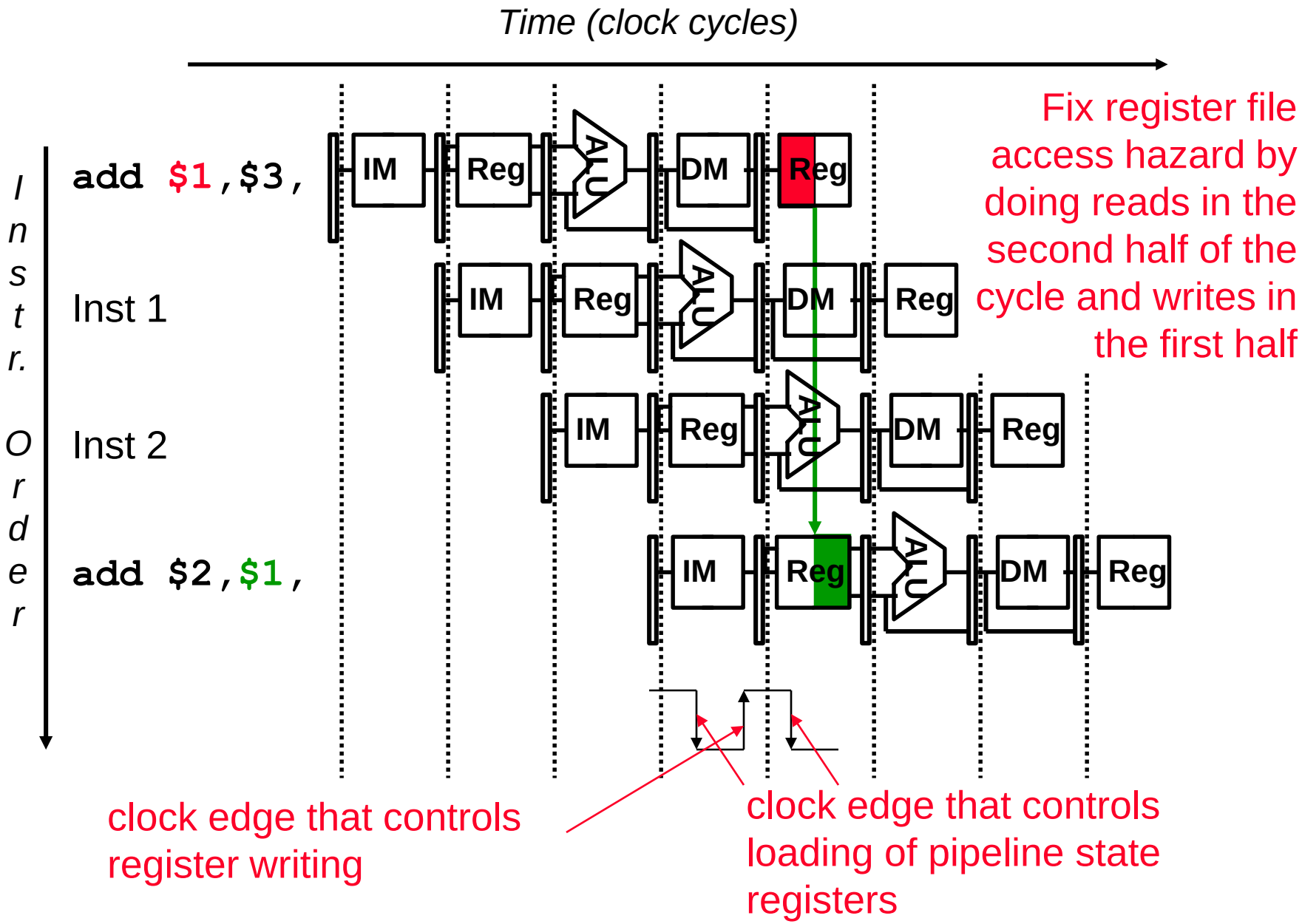


A Single Memory Would Be a Structural Hazard



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

How About Register File Access?



Structural Hazards

Some common Structural Hazards:

- Memory/RF:
  - we've already mentioned this one.
- Floating point:
  - Since many floating point instructions require many cycles, it's easy for them to interfere with each other.
- Starting up more of one type of instruction than there are resources.
  - For instance, the PA-8600 can support two ALU + two load/store instructions per cycle – that's how much hardware it has available.

Dealing with Structural Hazards

Stall

- low cost, simple
- Increases CPI (reduces IPC)
- use for rare case since stalling has performance effect

Pipeline hardware resource

- useful for multi-cycle resources
- good performance
- sometimes complex e.g., RAM

Replicate resource

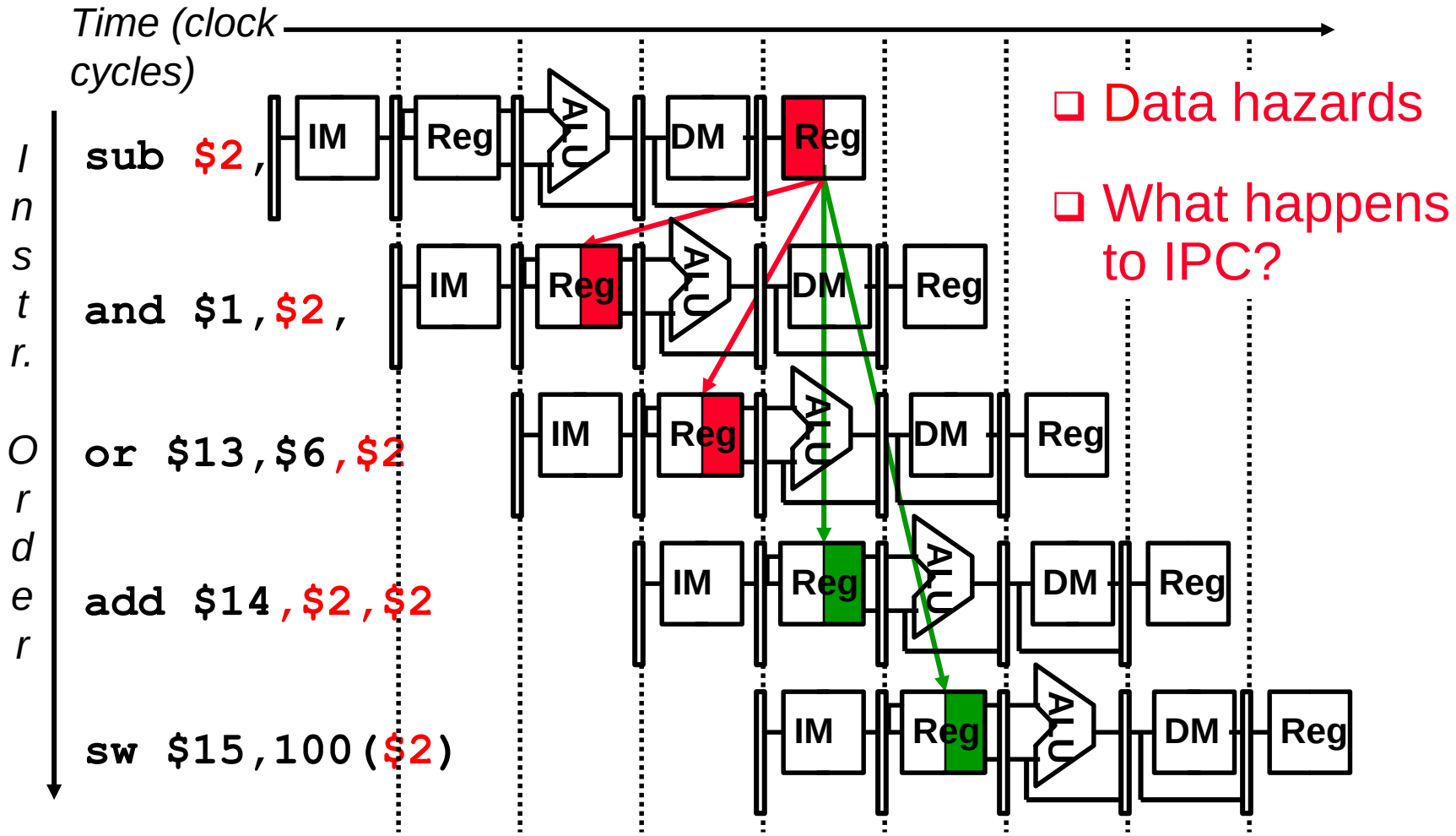
- good performance
- increases cost (+ maybe interconnect delay)
- useful for cheap or divisible resources

Structural Hazards

- Structural hazards are usually reduced with these rules:
  - Each instruction uses a resource at most once
  - Always use the resource in the same pipeline stage
  - Use the resource for one cycle only
- Many RISC ISAs are designed with this in mind
- Sometimes very difficult to do this.
  - For example, memory of necessity is used in the IF and MEM stages.

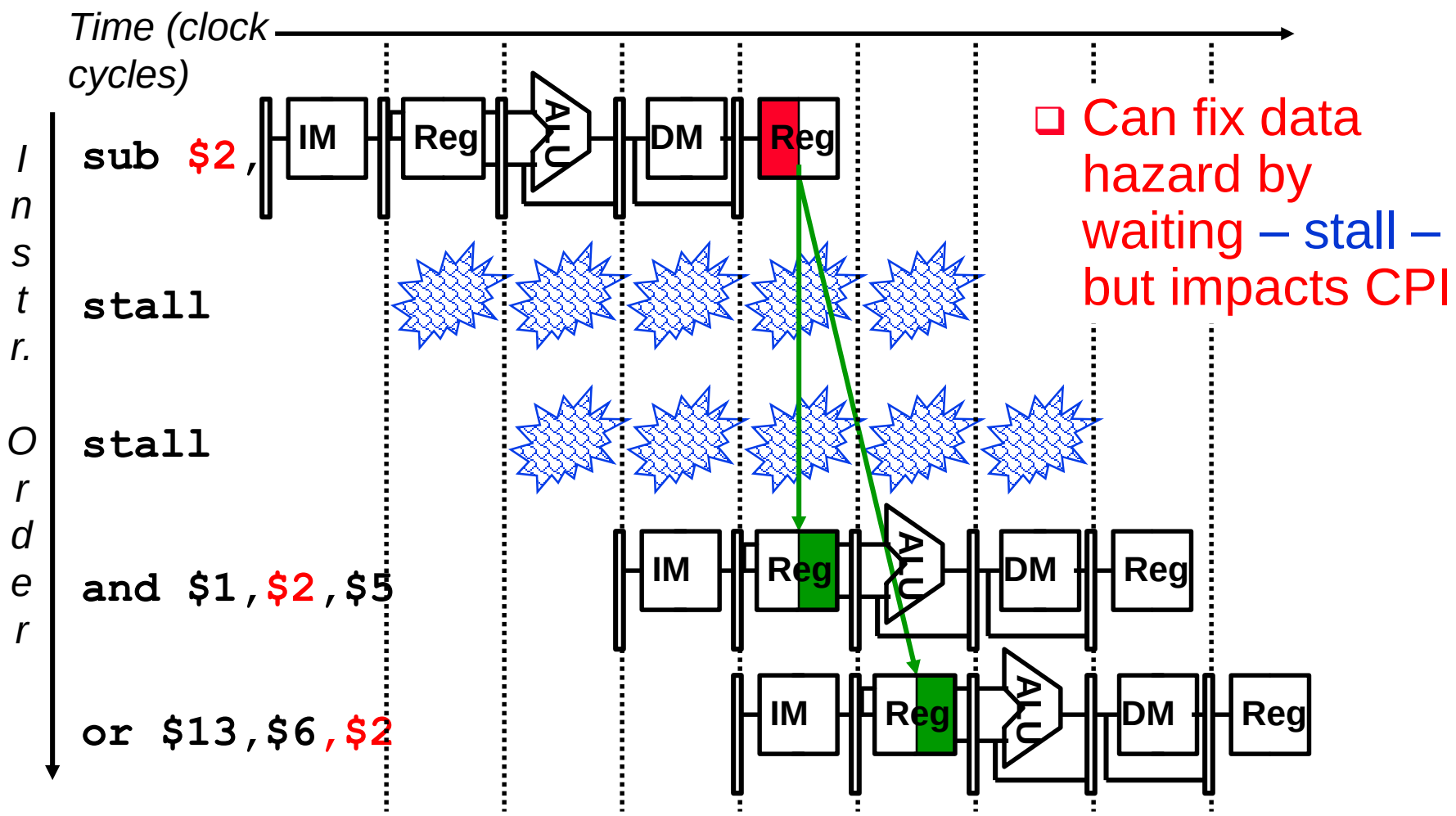
Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards



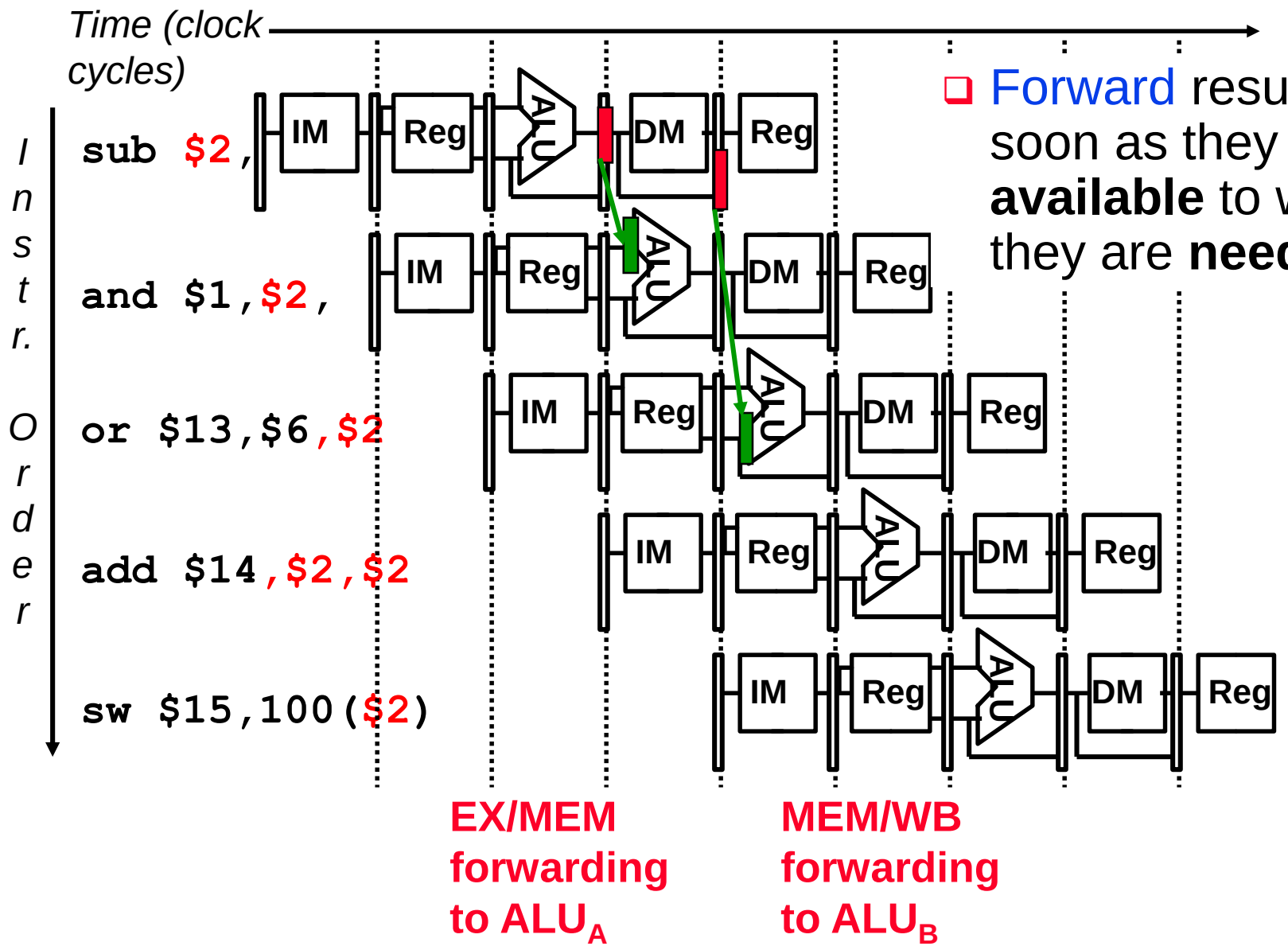
- Data hazards
- What happens to IPC?

One Way to "Fix" a Data Hazard



- Can fix data hazard by waiting – stall – but impacts CPI

Another Way to "Fix" a Data Hazard

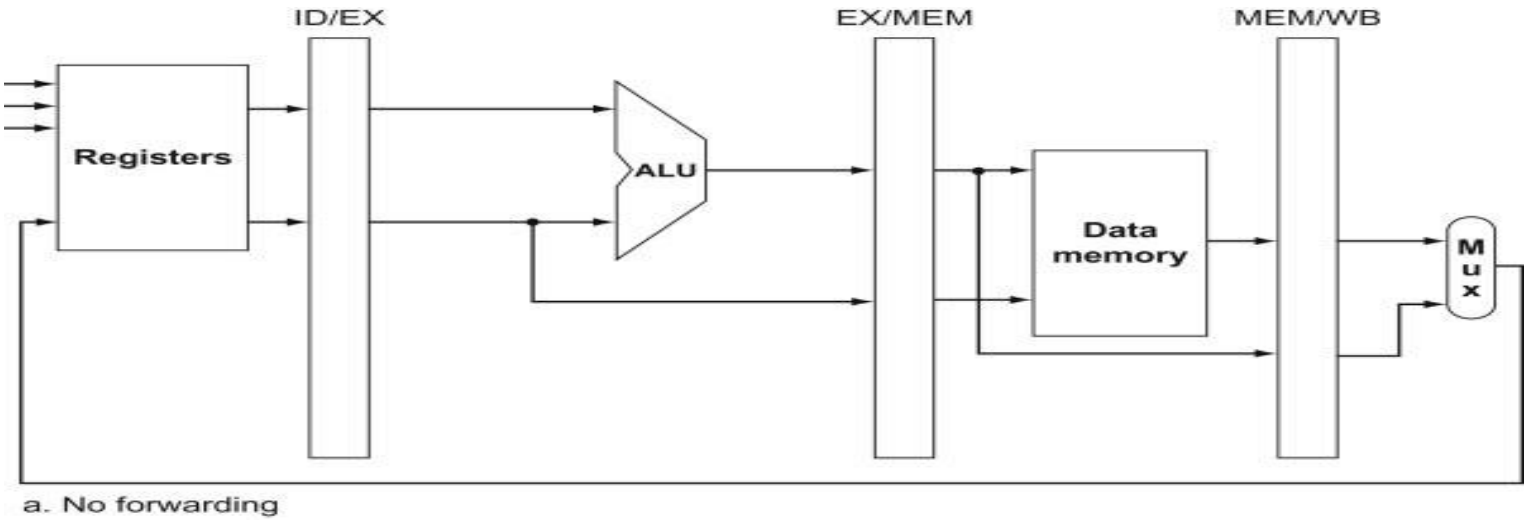


- Forward results as soon as they are **available** to where they are **needed**

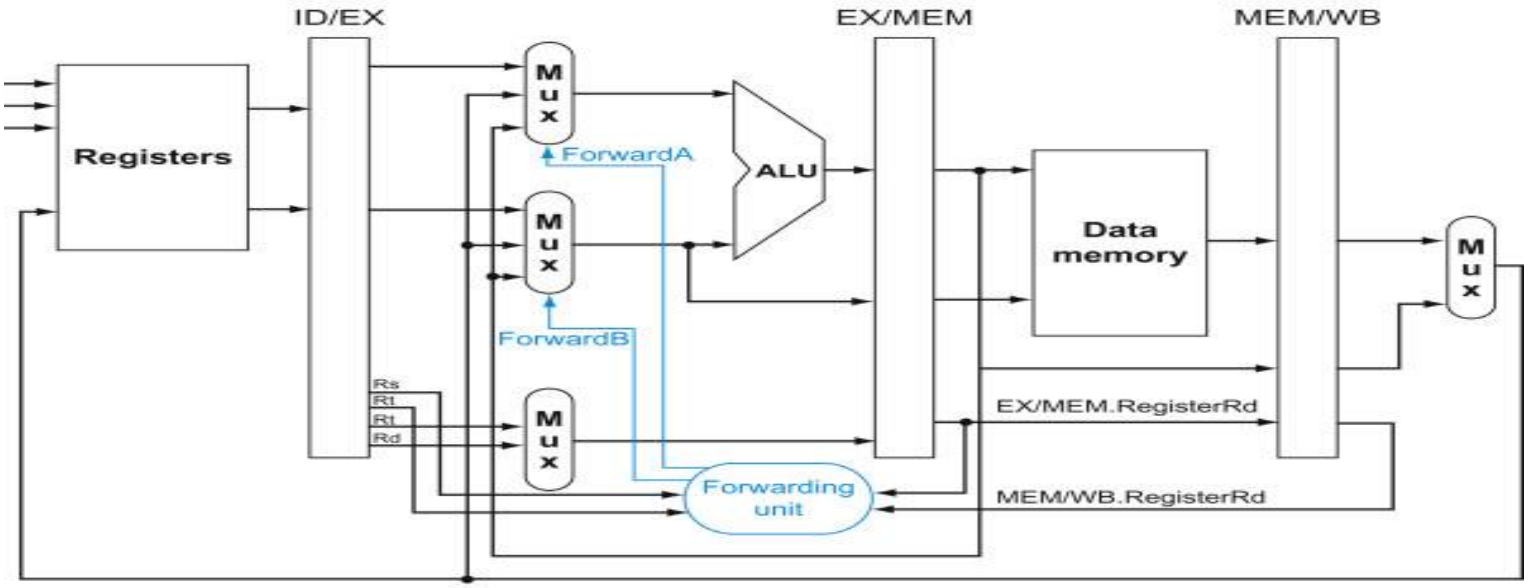
Data Forwarding (aka Bypassing)

- 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, one can achieve a CPI of 1 even in the presence of data dependencies**

Before and After Forwarding

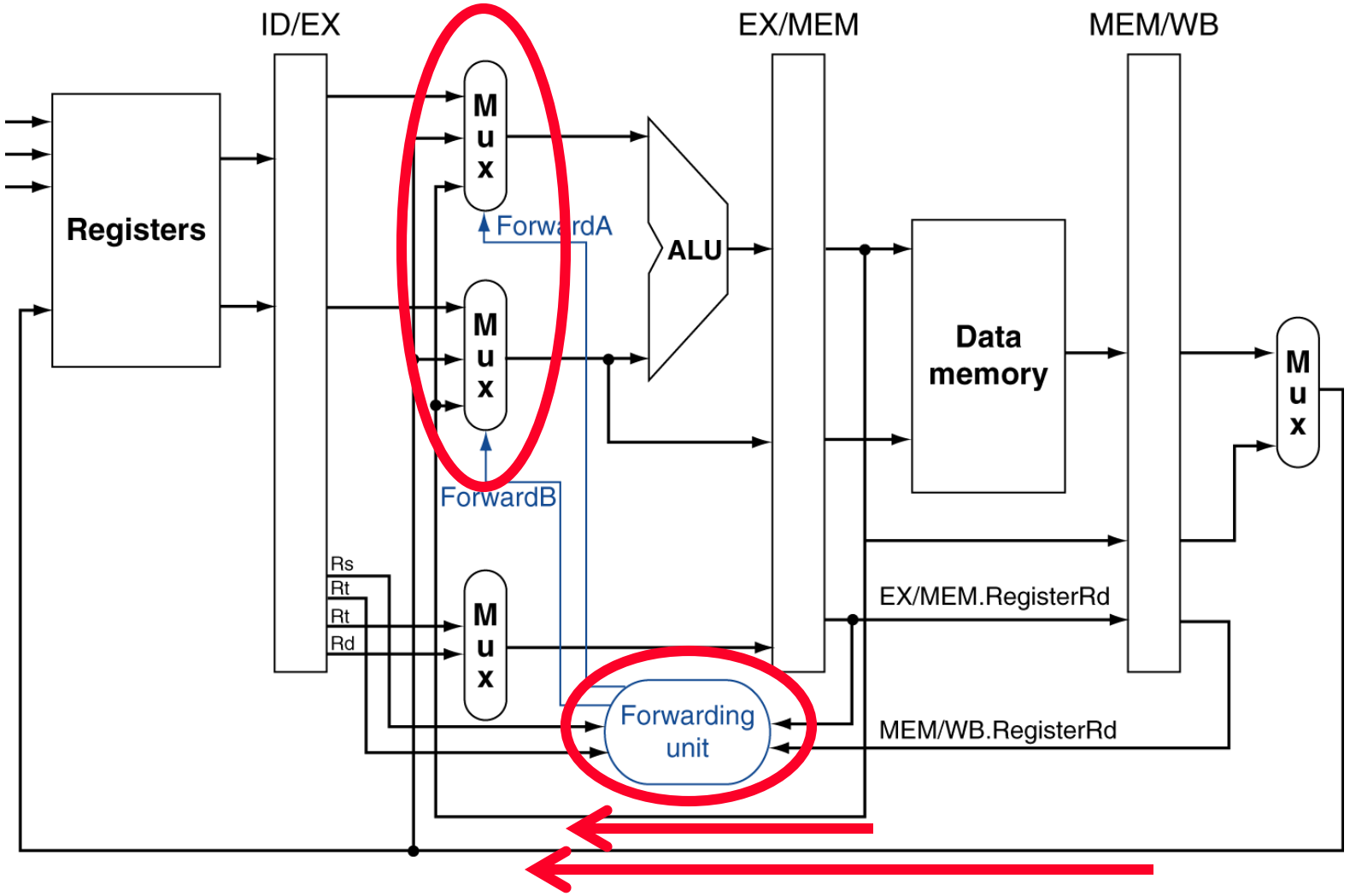


a. No forwarding



b. With forwarding

Forwarding Logic



Control Values for the Forwarding Multiplexors

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Data Forwarding Control Conditions

1. EX/MEM 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

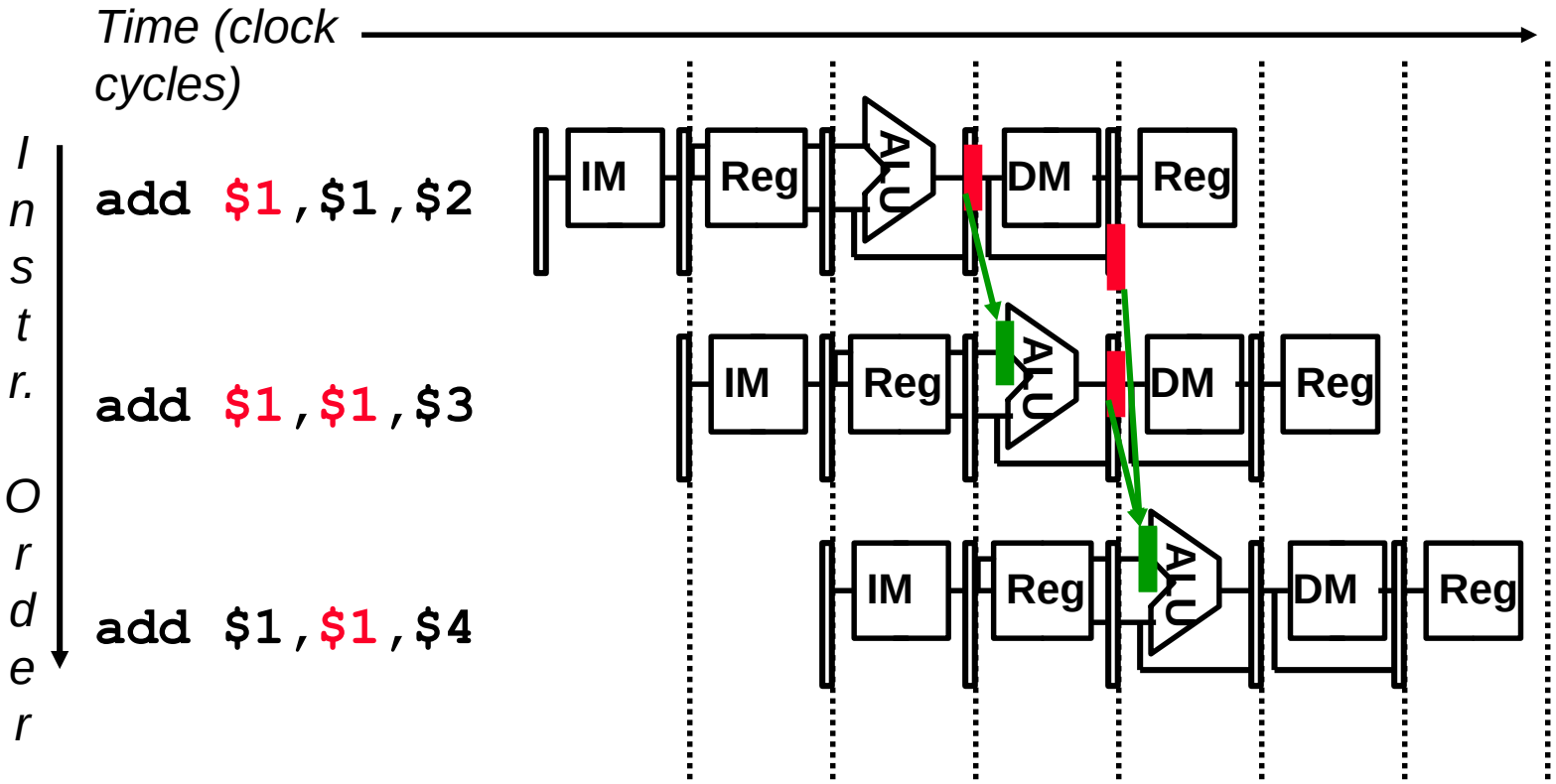
2. MEM/WB Forward Unit:

```
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/MEM Forward Unit:

```
if (EX/MEM.RegWrite
…
```

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

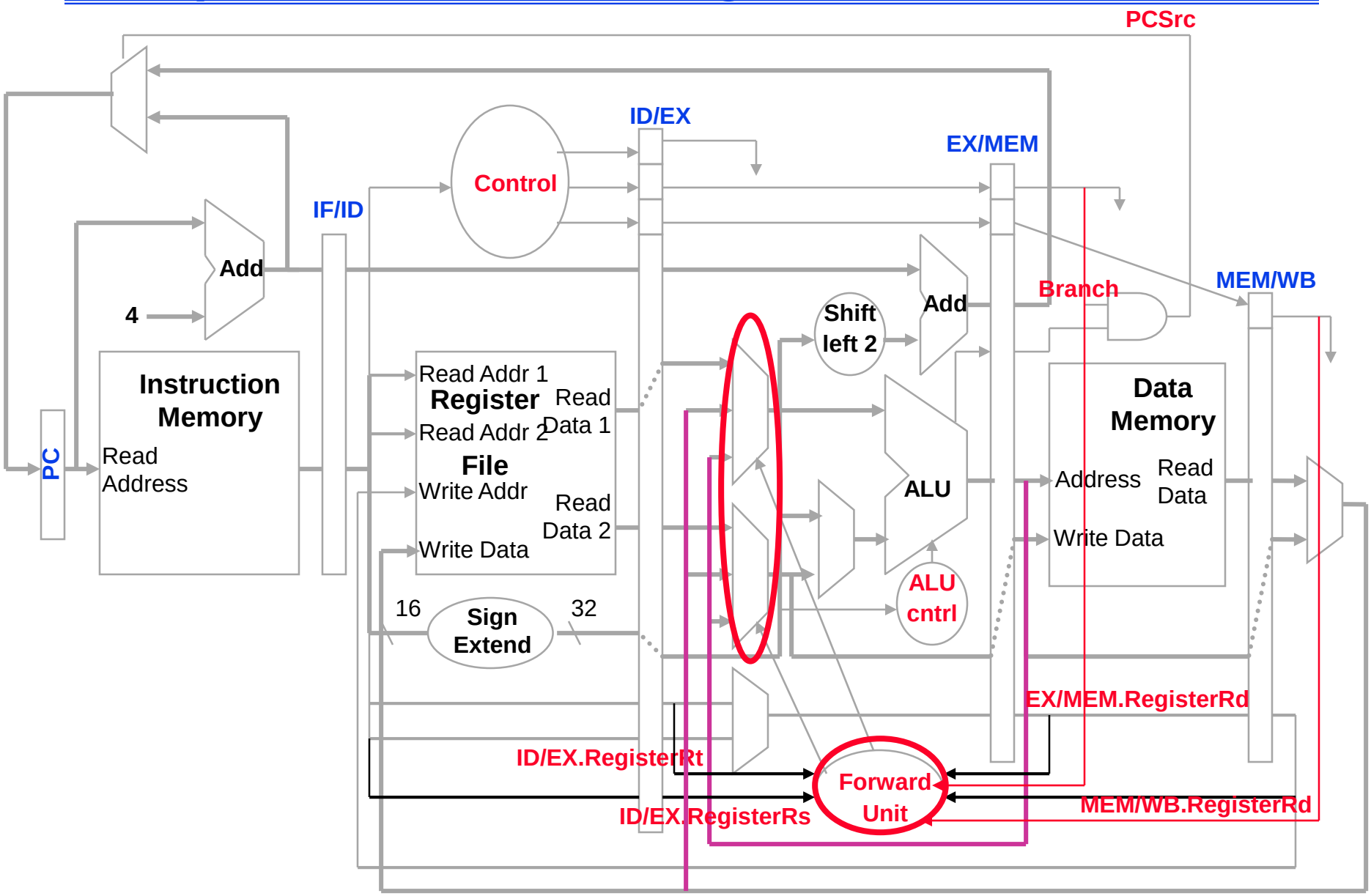
2. MEM/WB Forward Unit:

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

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

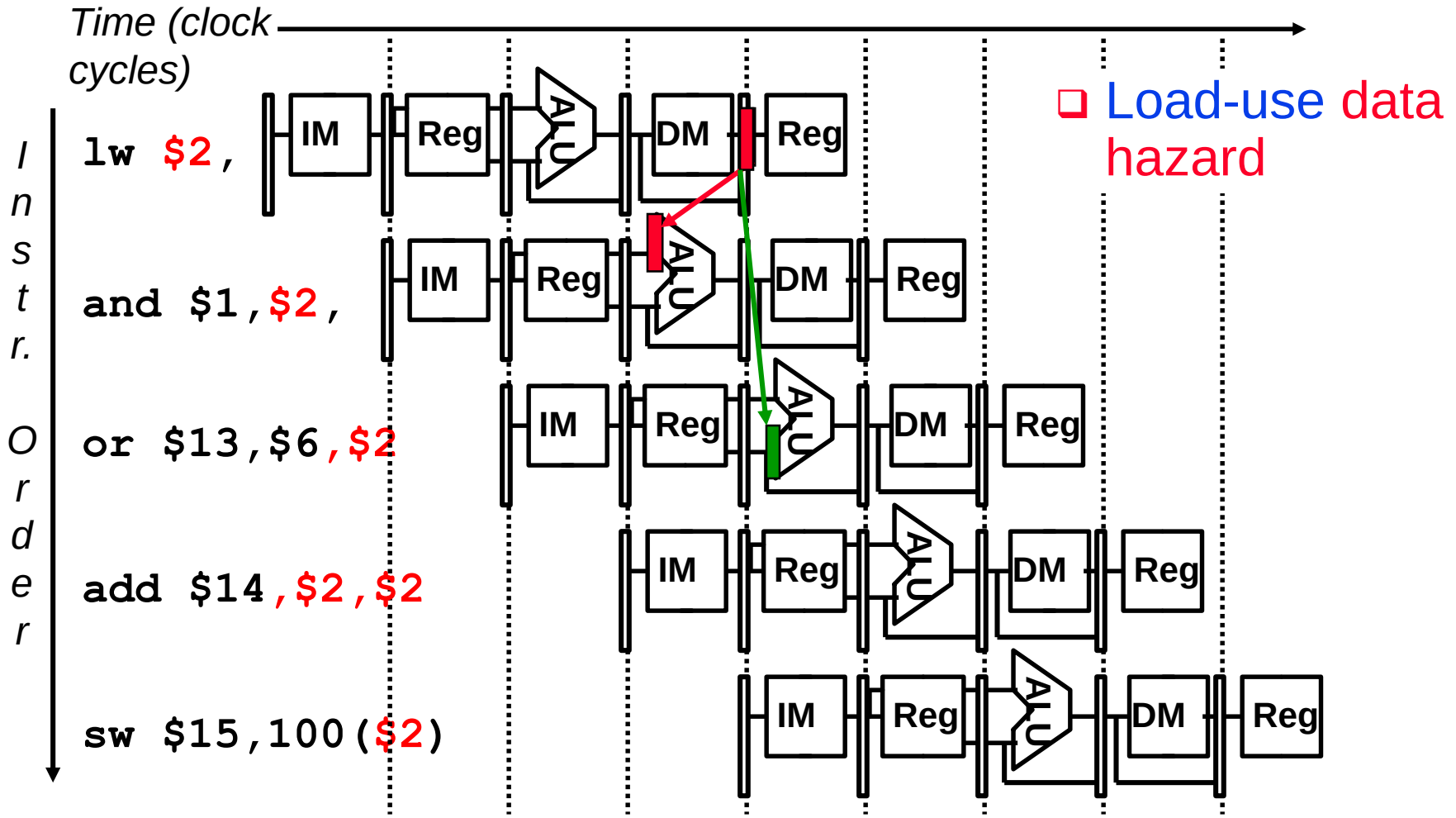
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and !(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

Datapath with Forwarding Hardware



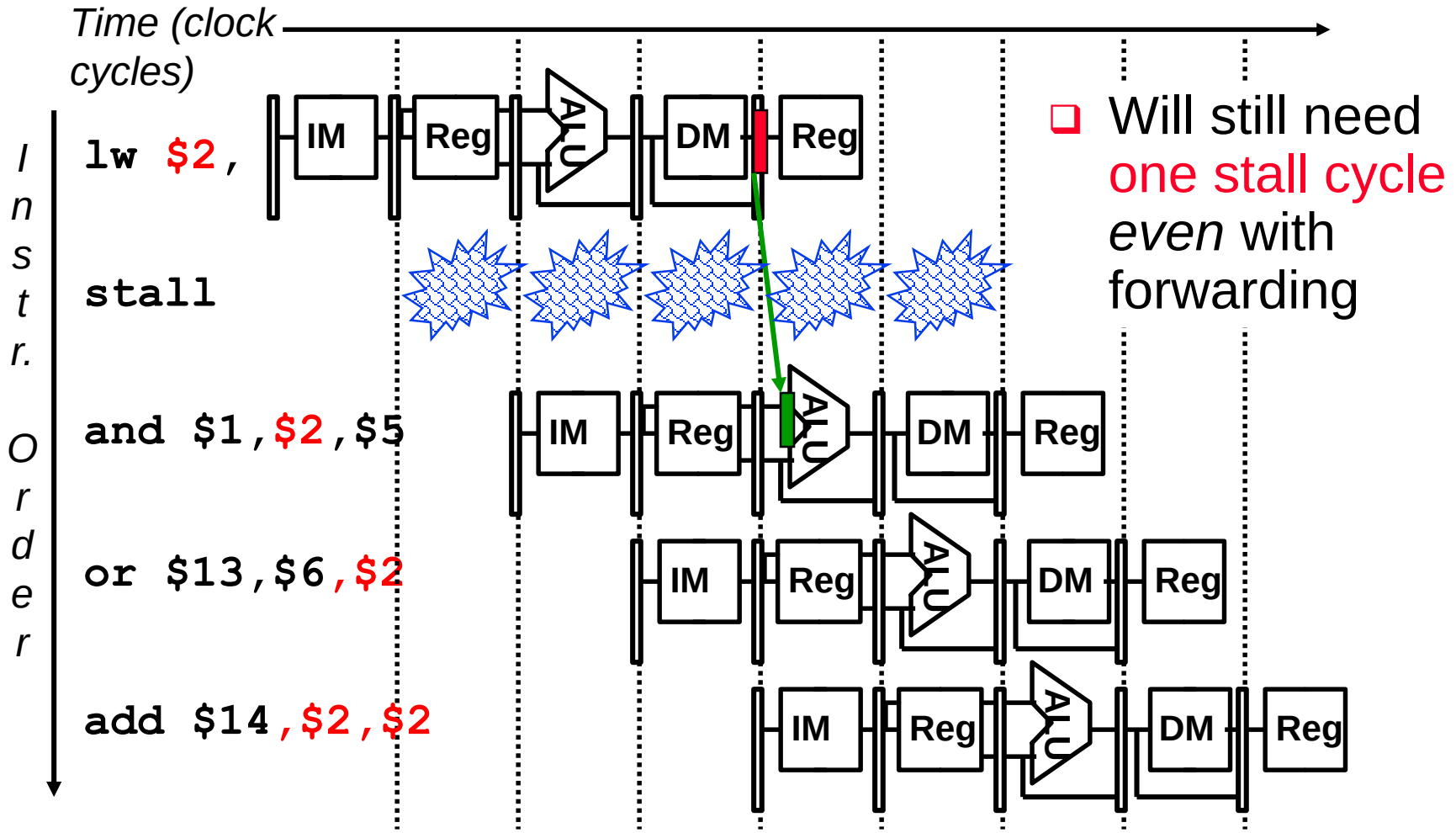
Load-Use Can Cause Data Hazards

- Dependencies backward in time cause hazards



Load-Use Can Cause Data Hazards

- Dependencies backward in time cause hazards



- Will still need one stall cycle *even* with forwarding

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

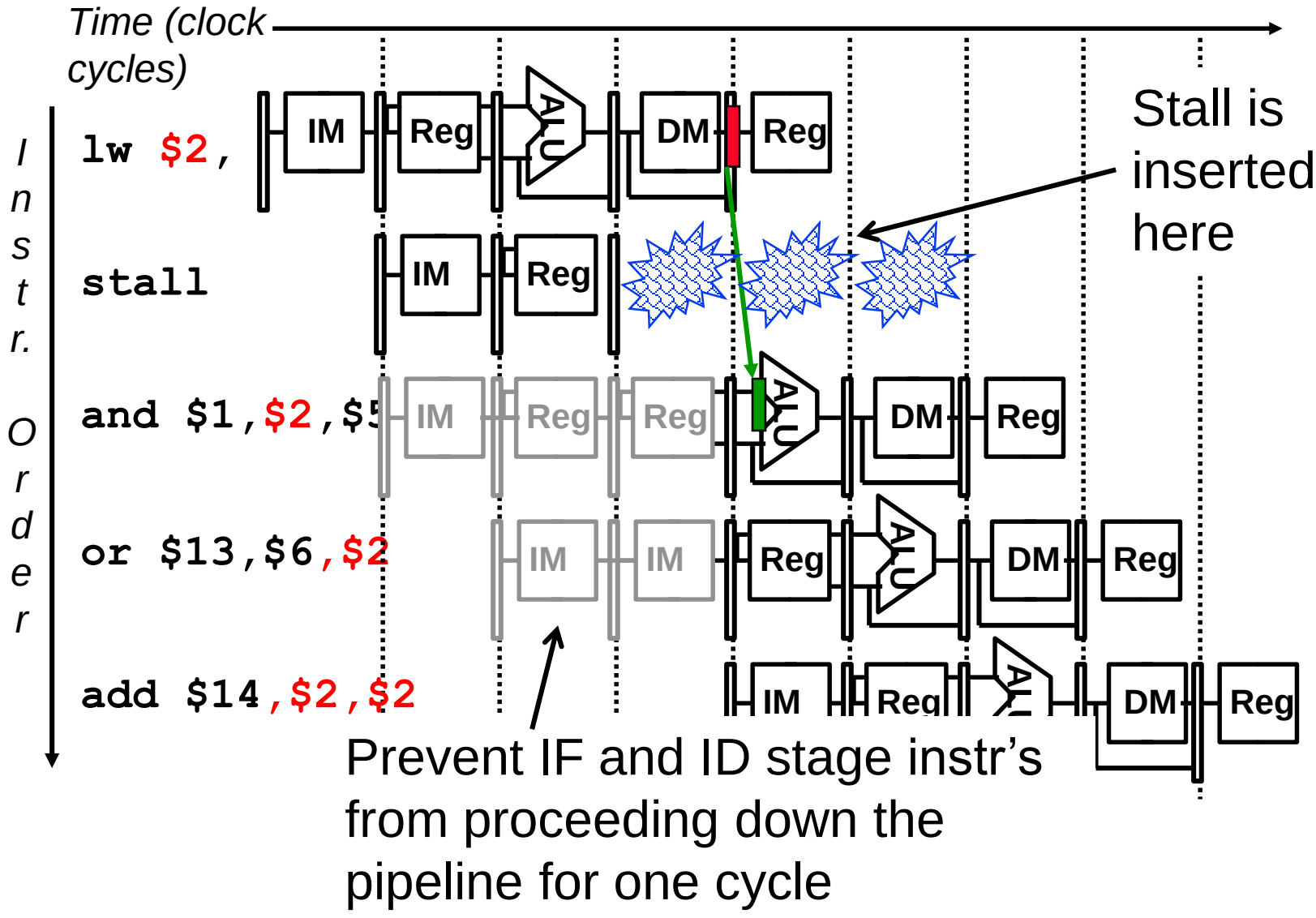
Data Hazard/Stall Hardware

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

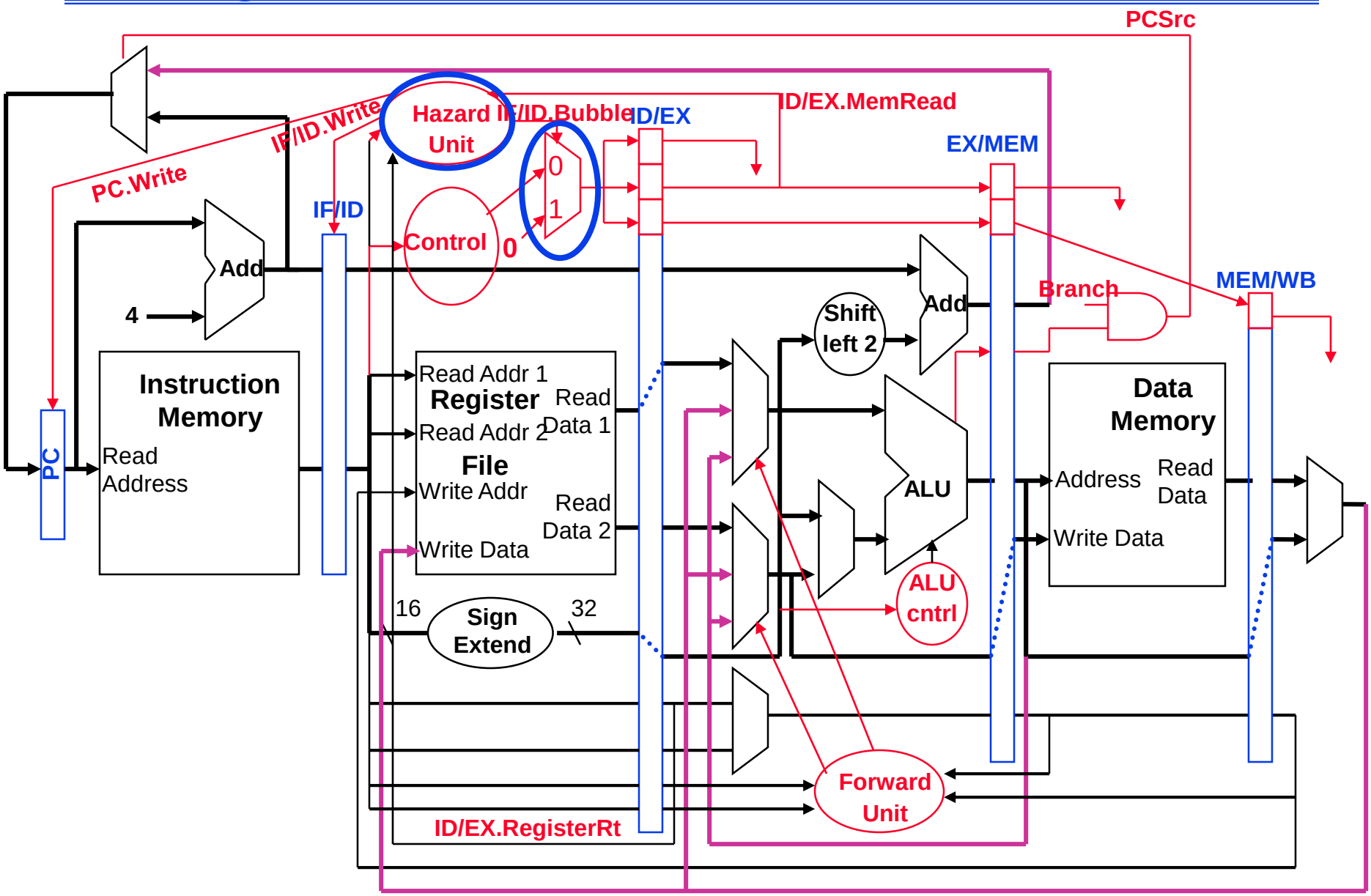
1. Insert a "bubble" between the `lw` instruction (in the EX stage) and the load-use instruction (in the ID stage) (i.e., insert a `noop` with `IF/ID.Bubble`)
   - Set the control bits in the EX, MEM, and WB control fields of the ID/EX pipeline register to 0 (`noop`). The Hazard Unit controls the mux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

Load-Use Can Cause Data Hazards

- Dependencies backward in time cause hazards



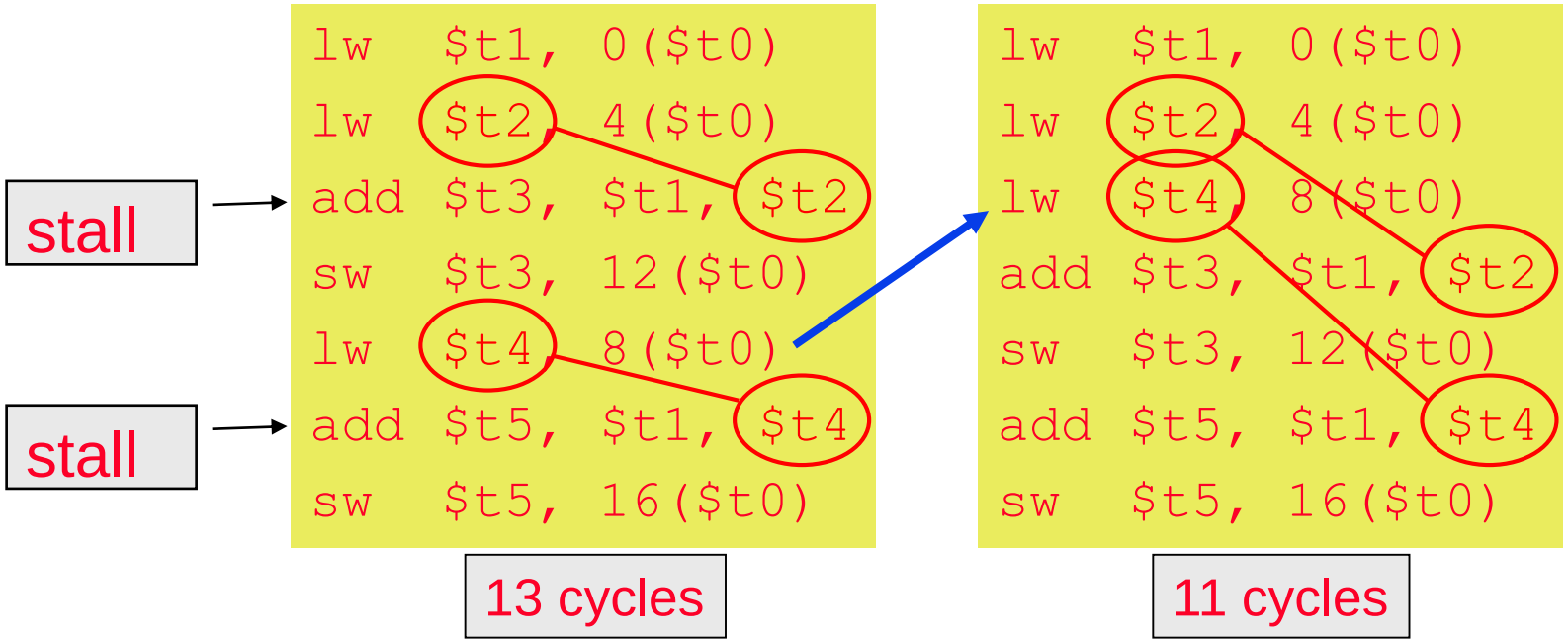
Adding the Data Hazard/Stall Hardware



Use Code Scheduling to Avoid L-U Stalls

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

  C code for `A = B + E; C = B + F;`



Summary

- All modern day processors use pipelining for performance (a CPI of 1 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 (more on this later)

Reading Assignment

- Read Sections 4.5, 4.6 and 4.7 from HP