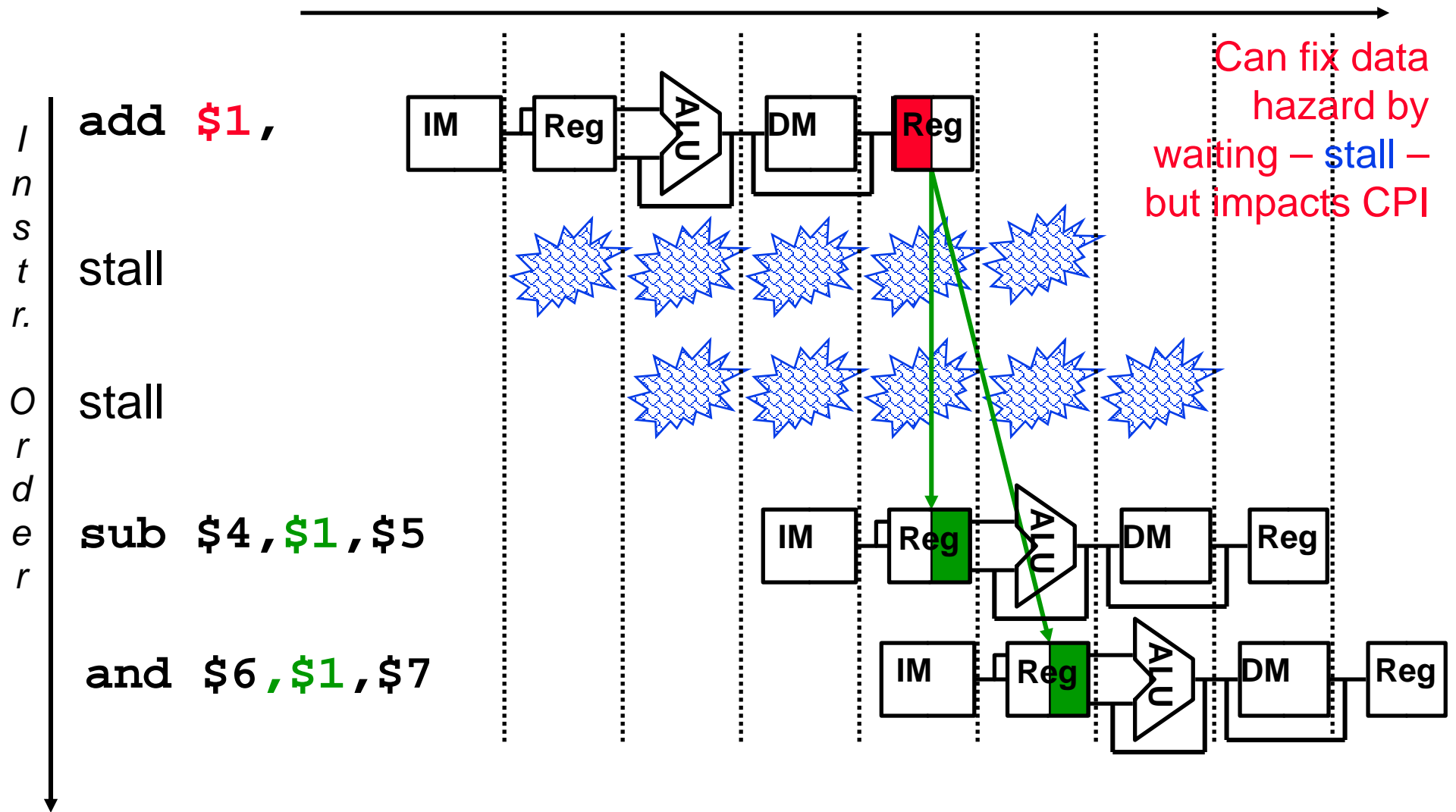
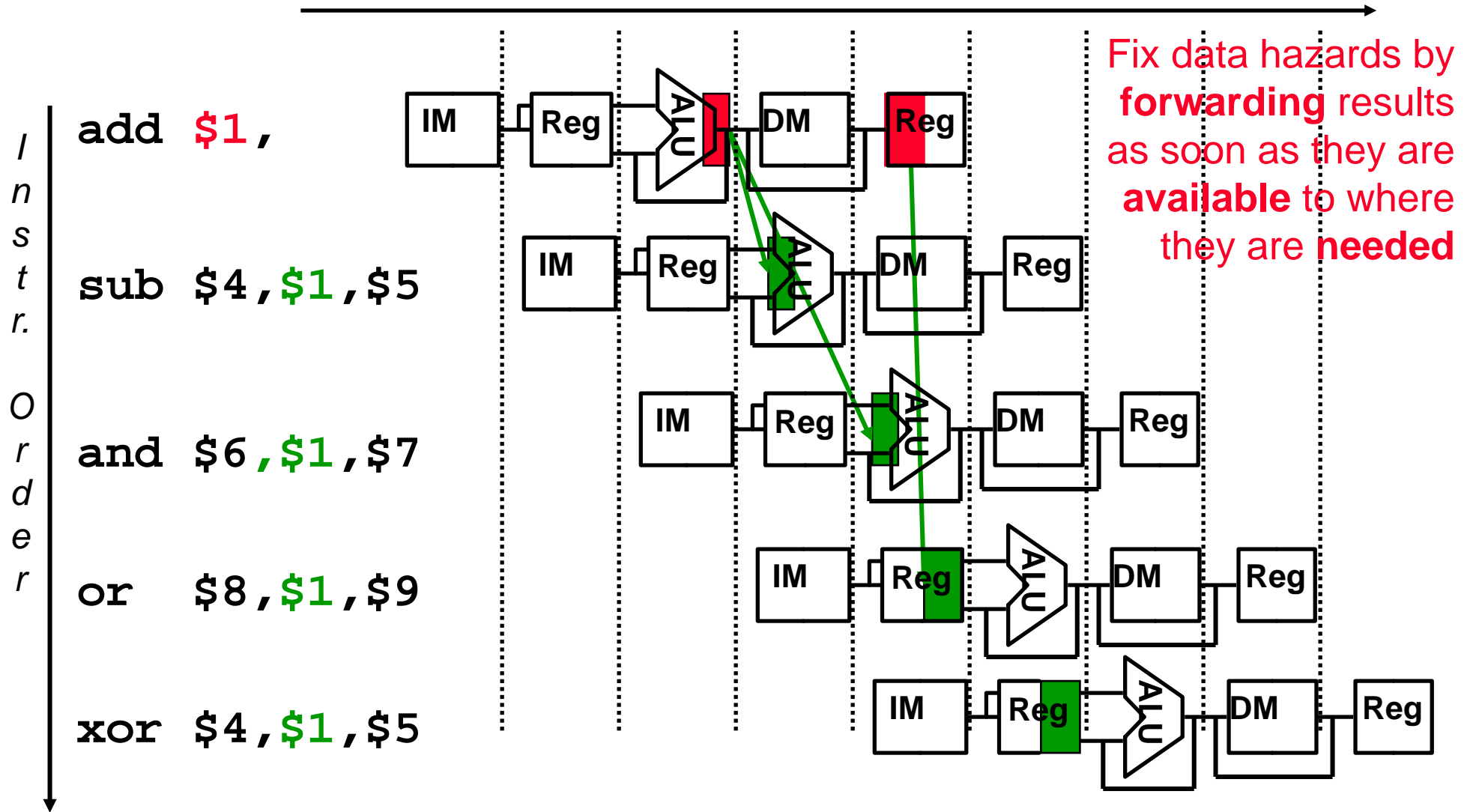


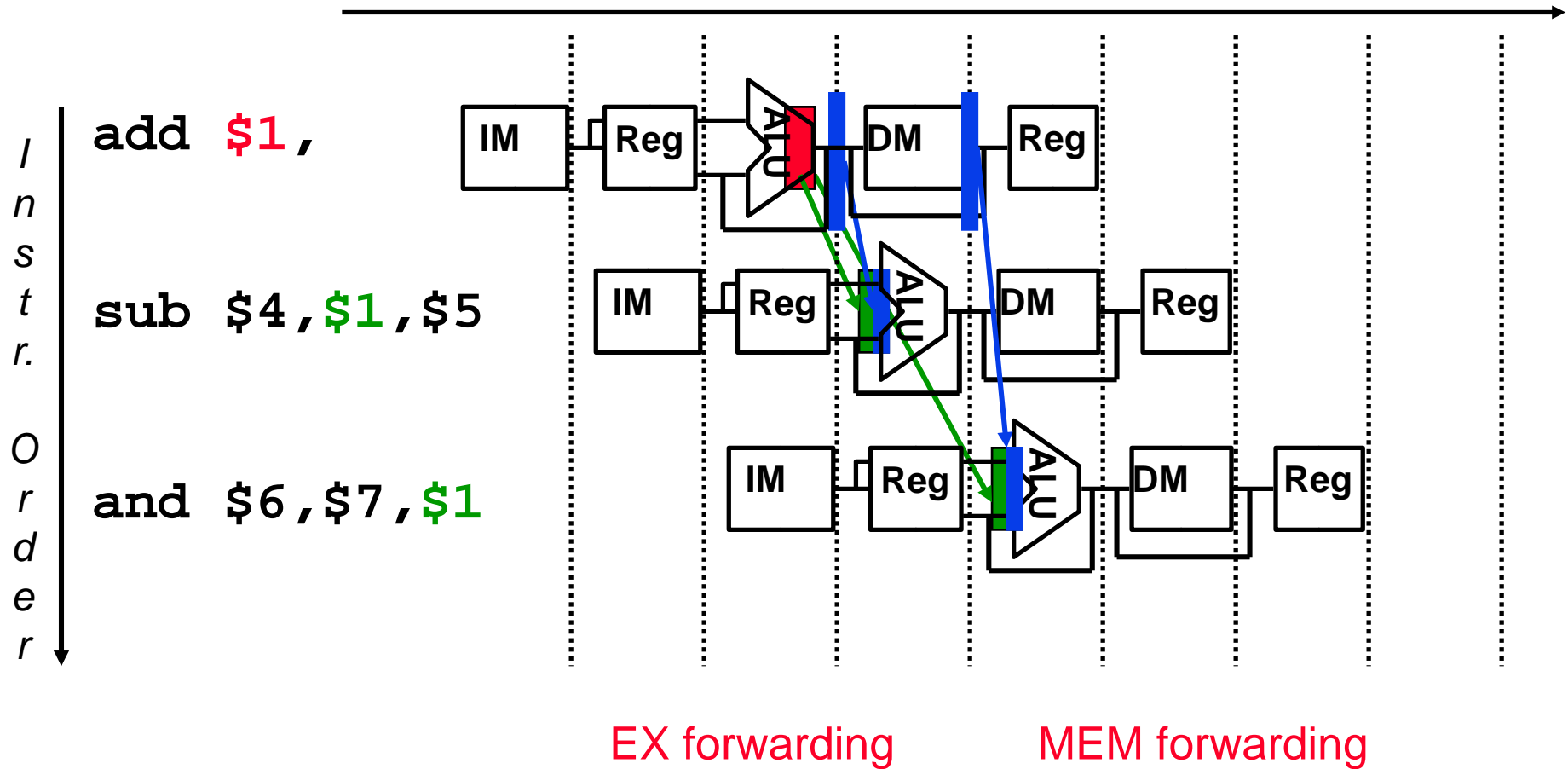
One Way to "Fix" a Data Hazard



Another Way to “Fix” a Data Hazard



Forwarding Illustration



Data Forwarding (aka Bypassing)

- ❑ Take the result from the earliest point that it exists in **any** of the pipeline state registers and forward it to the functional units (e.g., the ALU) that need it that cycle
- ❑ For ALU functional unit: the inputs can come from **any** pipeline register rather than just from 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

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

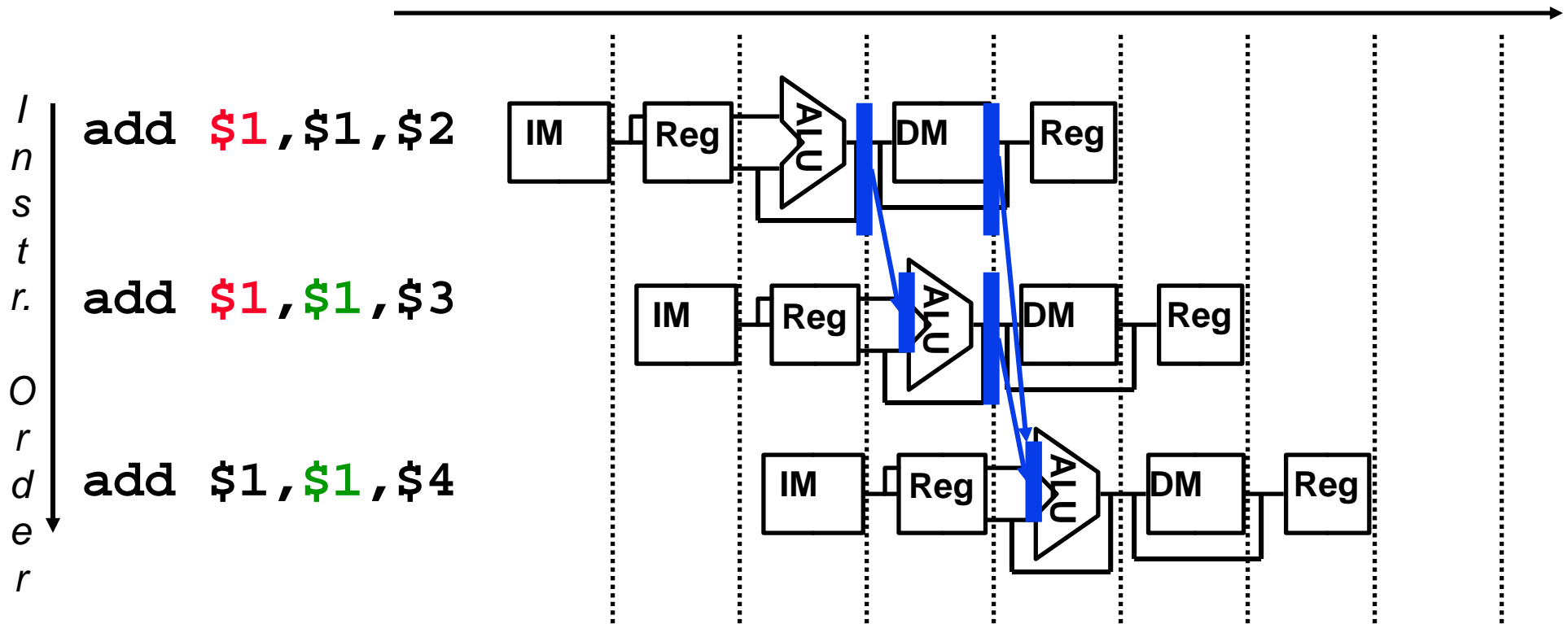
2. MEM Forward Unit:

```
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

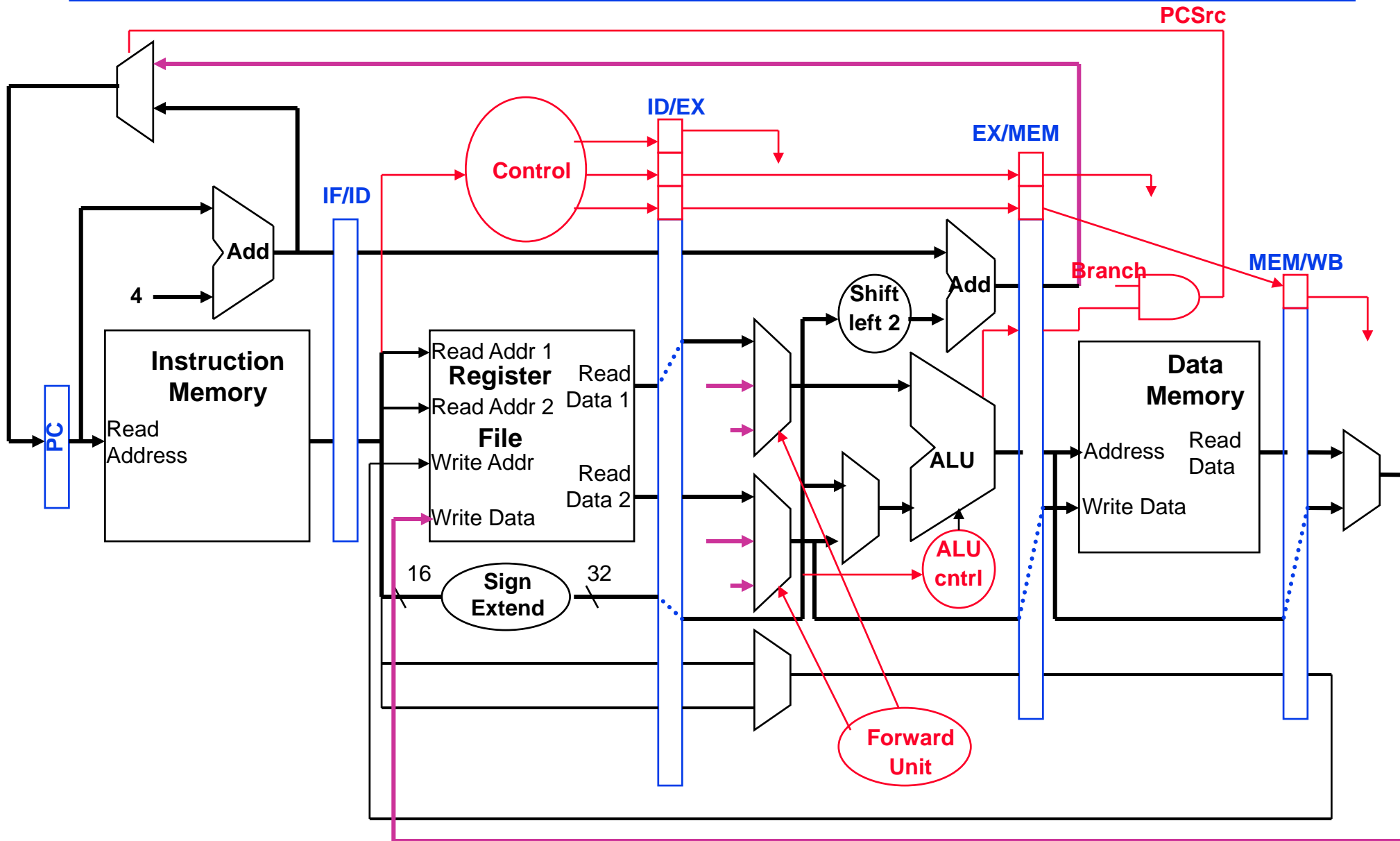
2. MEM Forward Unit:

```
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

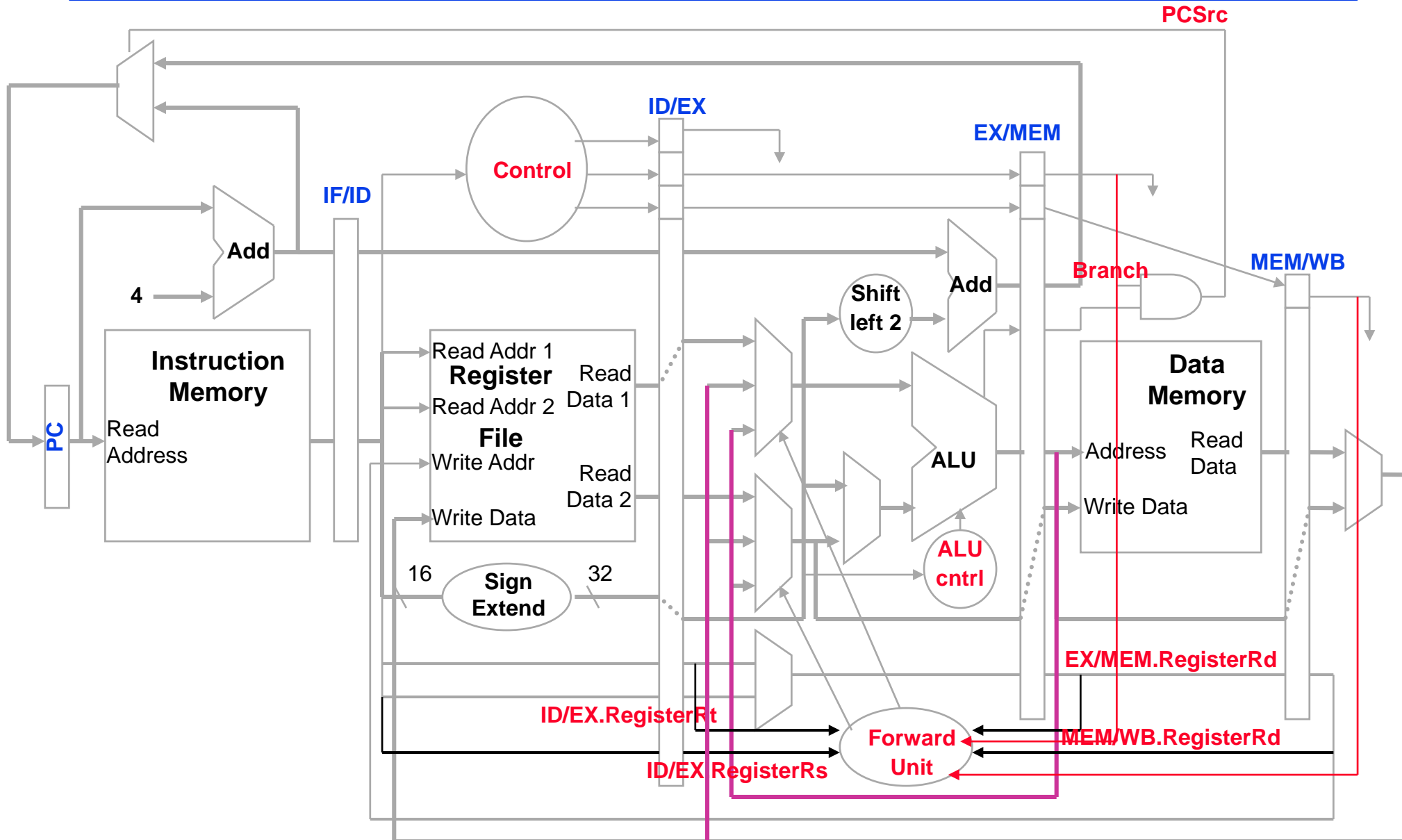
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01
```

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

Datapath with Forwarding Hardware

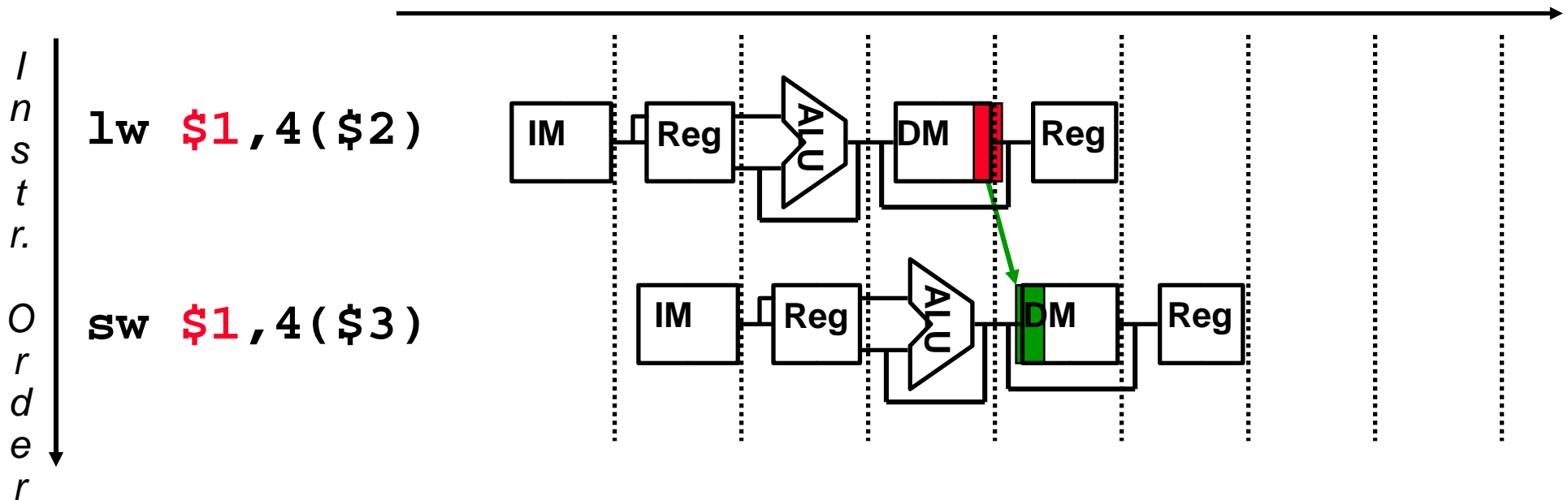


Datapath with Forwarding Hardware

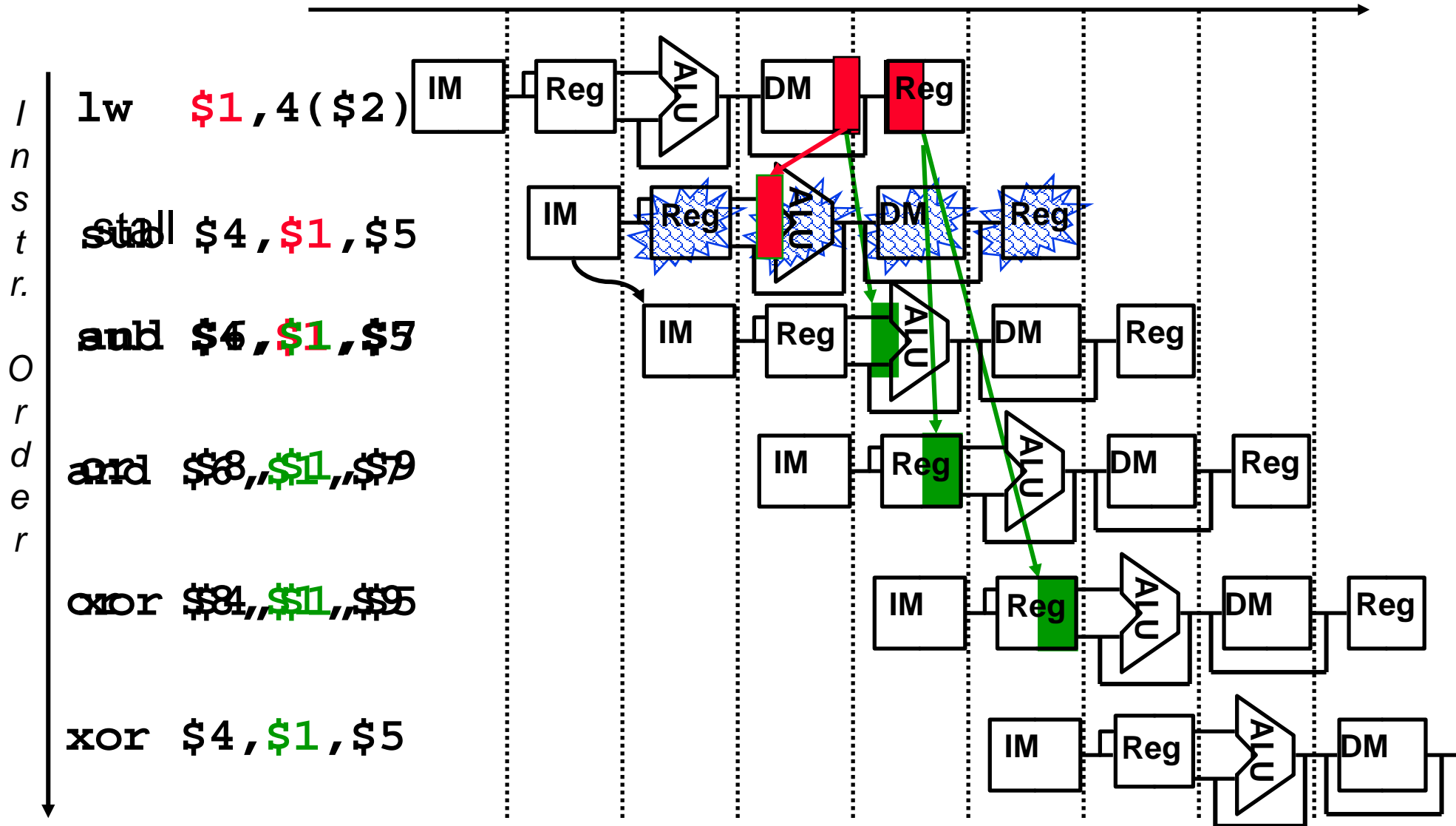


Memory-to-Memory Copies

- ❑ 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



Forwarding with Load-use Data Hazards



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

Load-use 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:

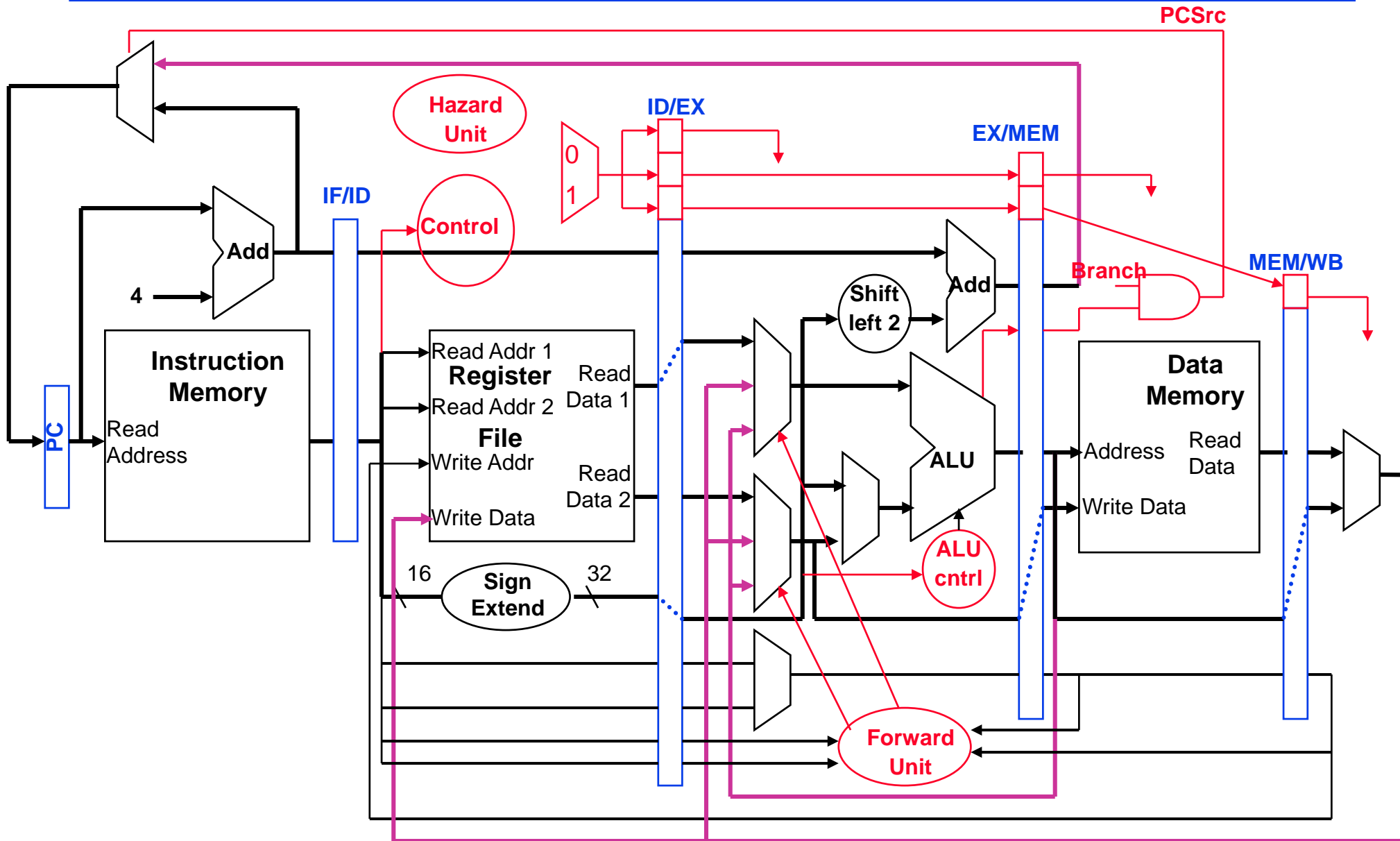
```
if (ID/EX.MemRead  
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)  
or (ID/EX.RegisterRt = IF/ID.RegisterRt)))  
stall the pipeline
```

- ❑ The first line tests to see if the instruction now in the EX stage is a l_w ; the next two lines check to see if the destination register of the l_w 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

Hazard/Stall Hardware

- ❑ Along with the Hazard Unit, we have to implement the stall
- ❑ 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
- ❑ 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` in the execution stream)
 - ❑ 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.
- ❑ Let the `lw` instruction and the instructions after it in the pipeline (before it in the code) proceed normally down the pipeline

Adding the Hazard/Stall Hardware



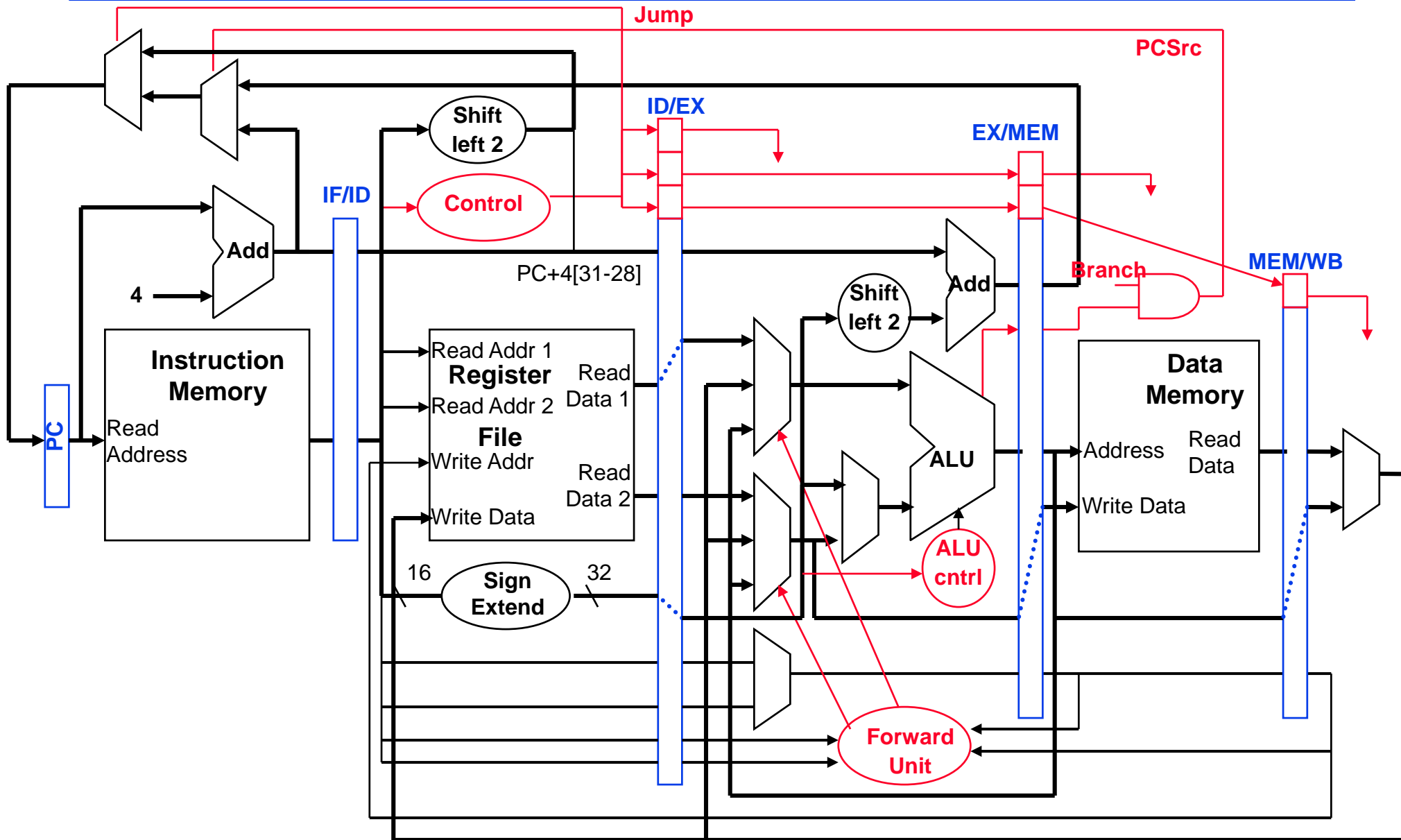
Control Hazards

- ❑ 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 !

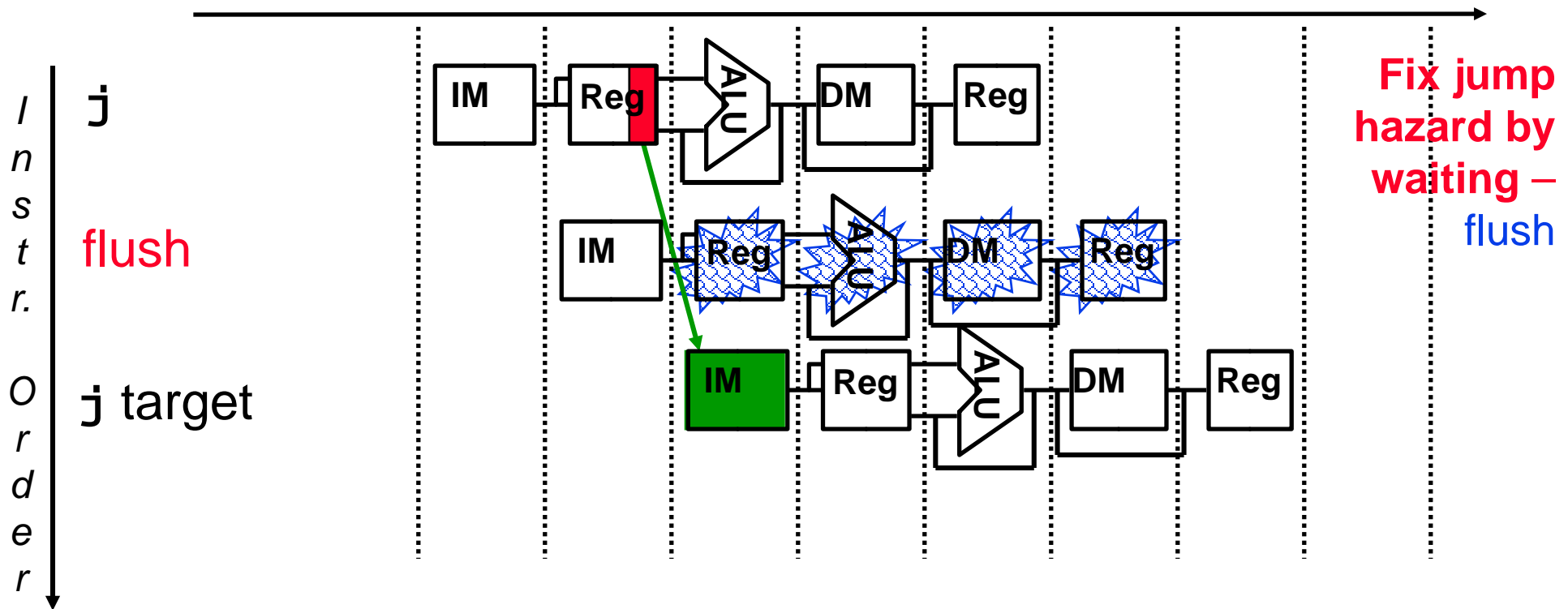
- ❑ Control hazards occur less frequently than data hazards, but there is *nothing* as effective against control hazards as forwarding is for data hazards

Datapath Branch and Jump Hardware



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 `noop`)

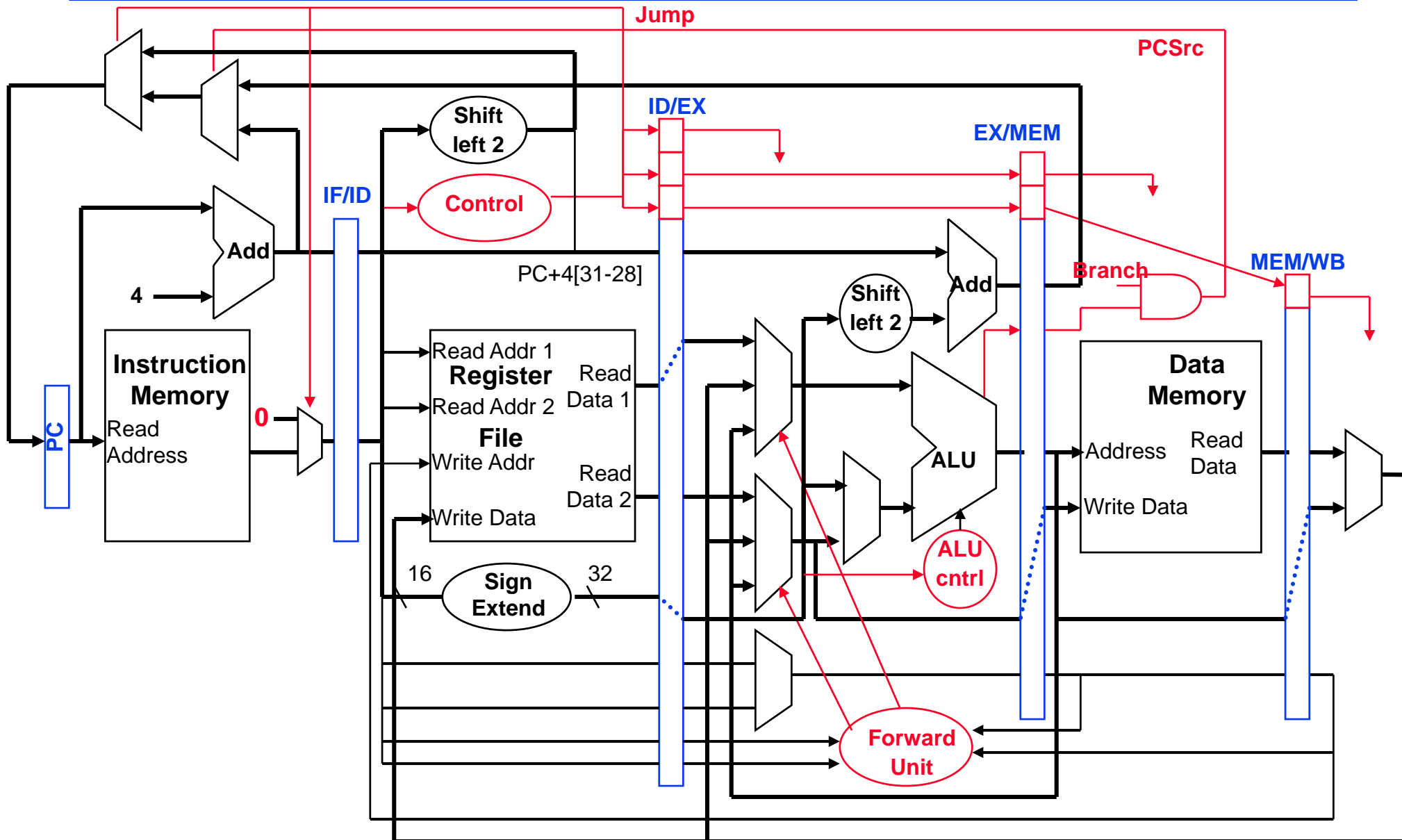


- ❑ Fortunately, jumps are very infrequent – only 3% of the SPECint instruction mix

Two “Types” of Stalls

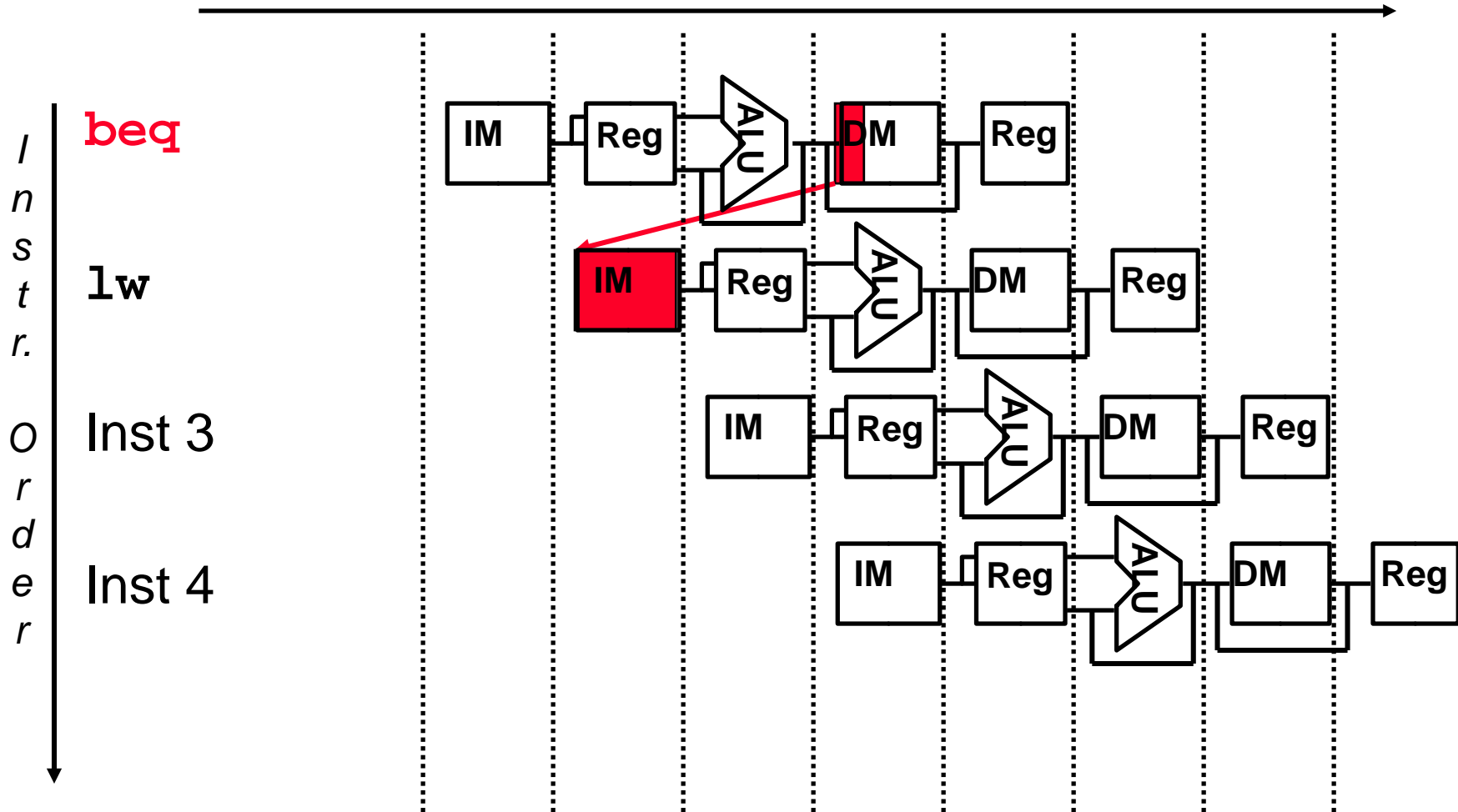
- ❑ `Noop` instruction (or bubble) **inserted** between two instructions in the pipeline (as done for load-use situations)
 - ❑ Keep the instructions *earlier* in the pipeline (later in the code) from progressing down the pipeline for a cycle (“bounce” them in place with write control signals)
 - ❑ Insert `noop` by zeroing control bits in the pipeline register at the appropriate stage
 - ❑ Let the instructions later in the pipeline (earlier in the code) progress normally down the pipeline
- ❑ Flushes (or instruction squashing) where an instruction in the pipeline is **replaced** with a `noop` instruction (as done for instructions located sequentially after `j` instructions)
 - ❑ Zero the control bits for the instruction to be flushed

Supporting ID Stage Jumps

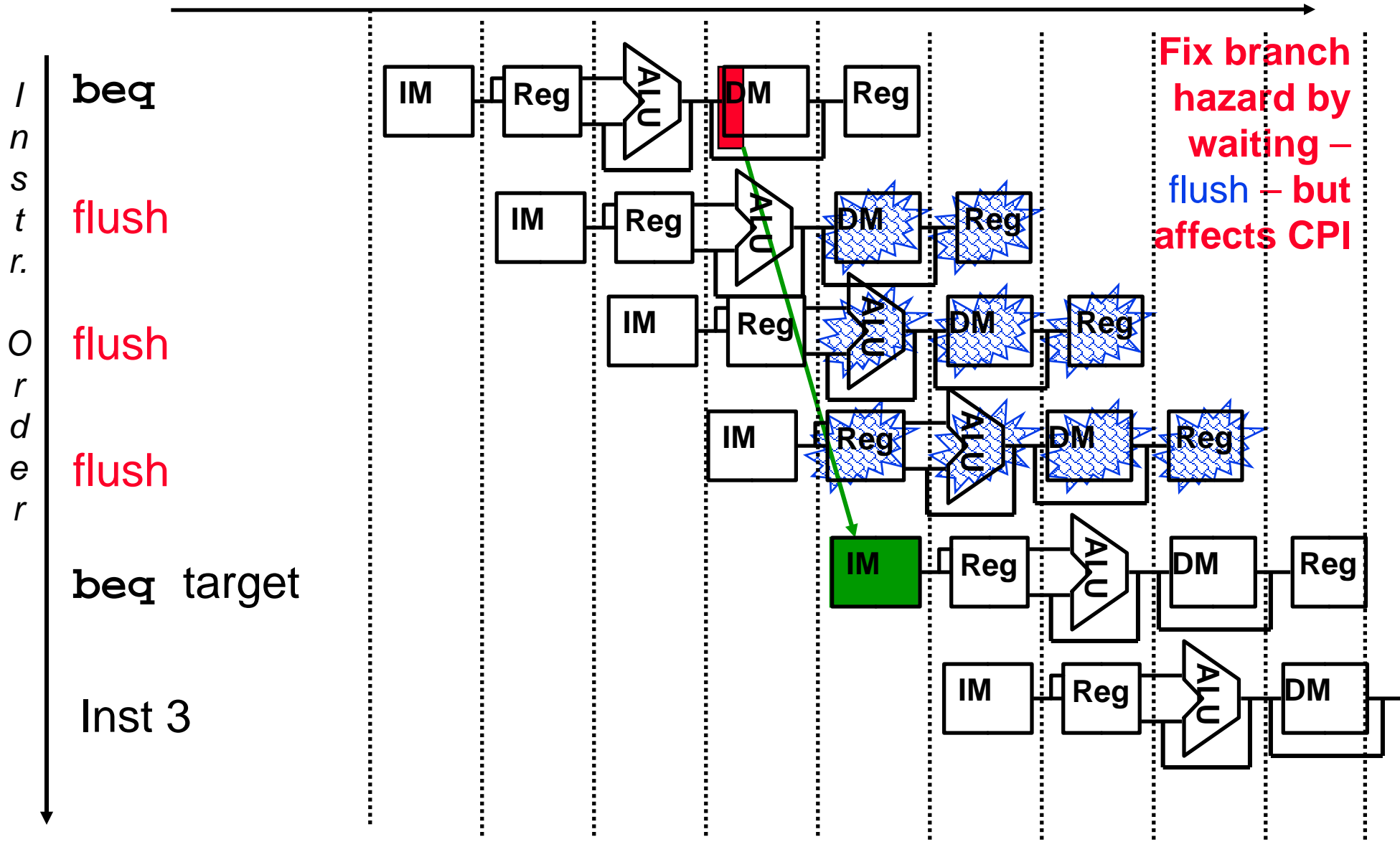


Review: Branch Instr's Cause Control Hazards

- Dependencies backward in time cause **hazards**

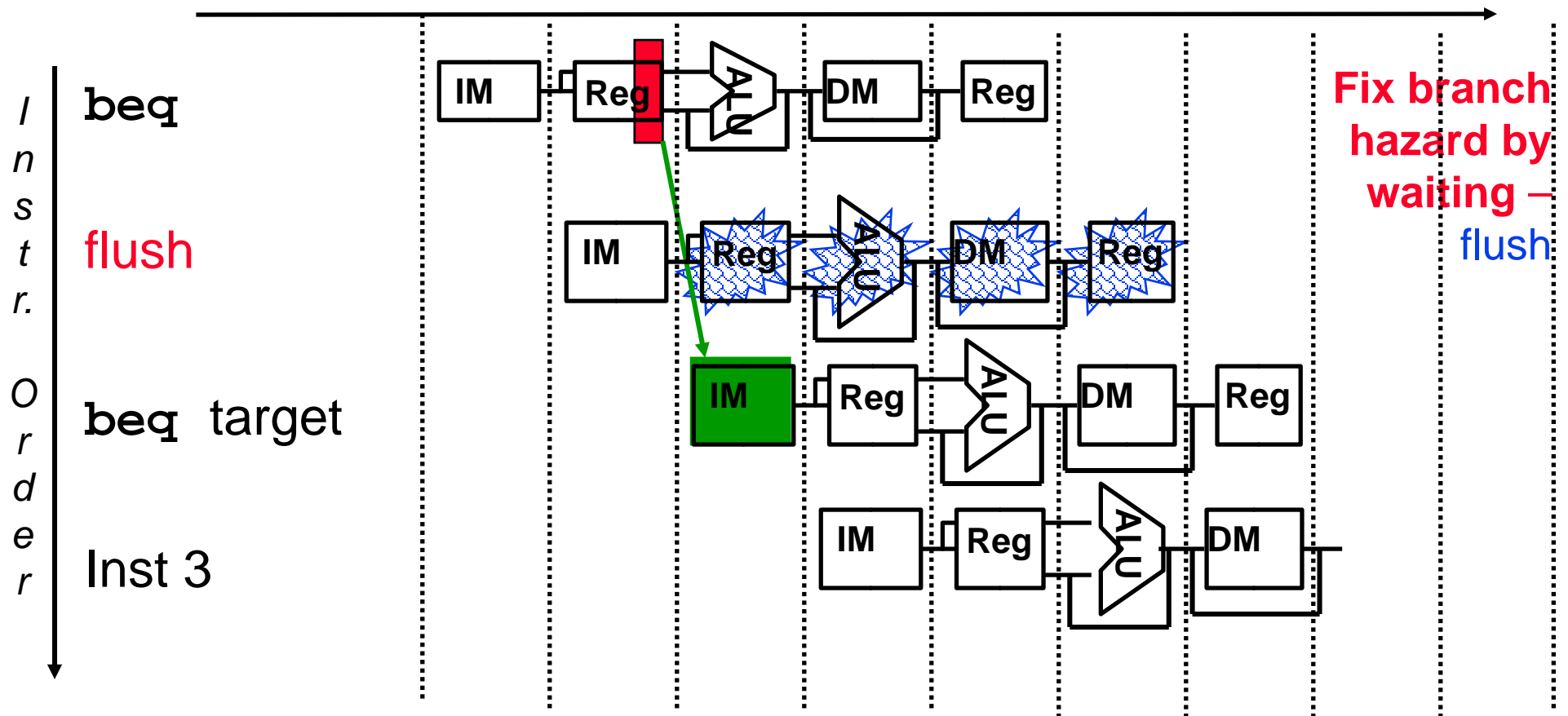


One Way to "Fix" a Branch Control Hazard



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

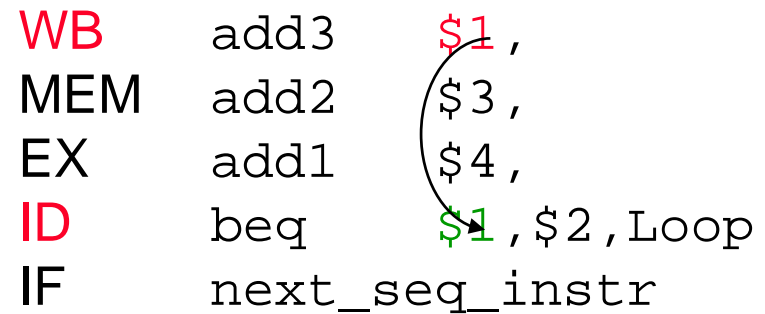


Reducing the Delay of Branches

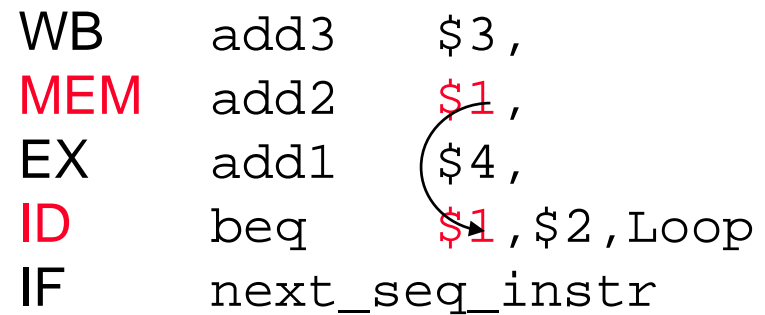
- ❑ Move the branch decision hardware back to the EX stage
 - ❑ Reduces the number of stall (flush) cycles to two
 - ❑ Adds an `and` gate and a `2x1 mux` to the EX timing path
- ❑ Add hardware to compute the branch target address and evaluate the branch decision to the ID stage
 - ❑ Reduces the number of stall (flush) cycles to one (like with jumps)
 - But now need to add **forwarding hardware** in ID stage
 - ❑ Computing branch target address can be done in parallel with RegFile read (done for all instructions – only used when needed)
 - ❑ Comparing the registers can't be done until after RegFile read, so comparing and updating the PC adds a mux, a comparator, and an `and` gate to the ID timing path
- ❑ For deeper pipelines, branch decision points can be even *later* in the pipeline, incurring more stalls

ID Branch Forwarding Issues

- MEM/WB “forwarding” is taken care of by the normal RegFile write before read operation



- Need to forward from the EX/MEM pipeline stage to the ID comparison hardware for cases like

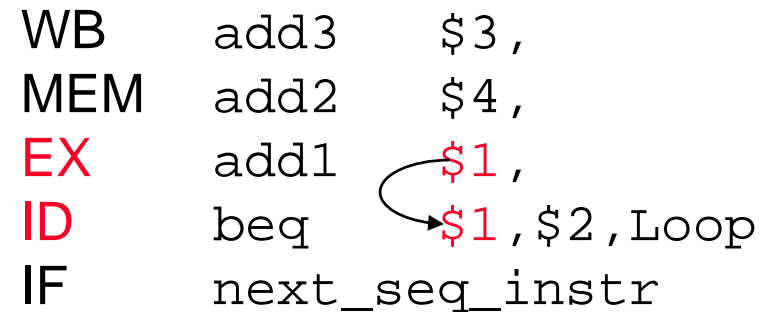


```
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
    ForwardC = 1
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
    ForwardD = 1
```

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

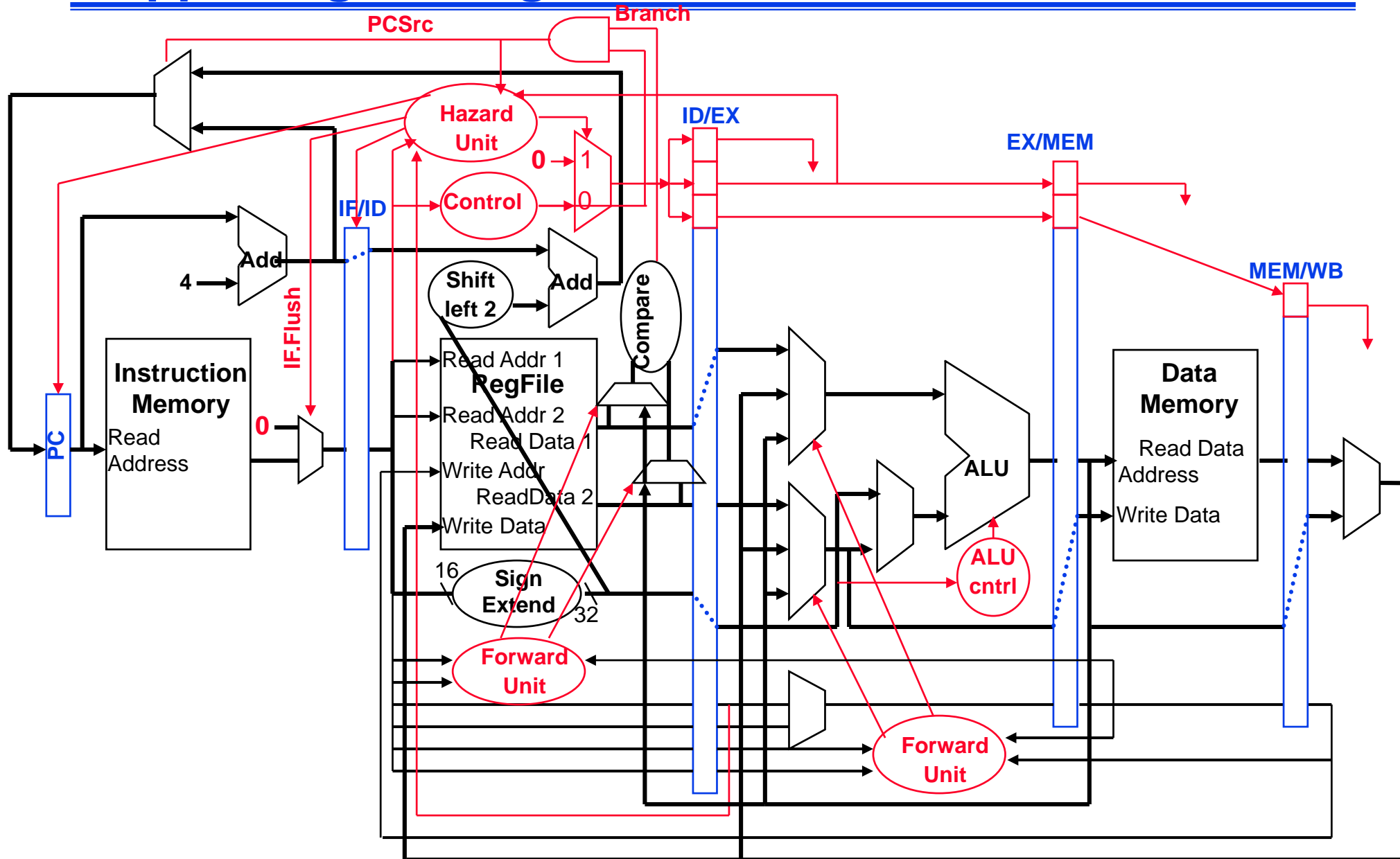
ID Branch Forwarding Issues, con't

- If the instruction immediately before the branch produces one of the branch source operands, then a **stall** needs to be inserted (between the `beq` and `add1`) since the EX stage ALU operation is occurring at the *same time* as the ID stage branch compare operation



- “Bounce” the `beq` (in ID) and `next_seq_instr` (in IF) in place (ID Hazard Unit deasserts `PC.Write` and `IF/ID.Write`)
- Insert a stall between the `add` in the EX stage and the `beq` in the ID stage by zeroing the control bits going into the ID/EX pipeline register (done by the ID Hazard Unit)
- If the branch is found to be taken, then flush the instruction currently in IF (**IF.Flush**)

Supporting ID Stage Branches

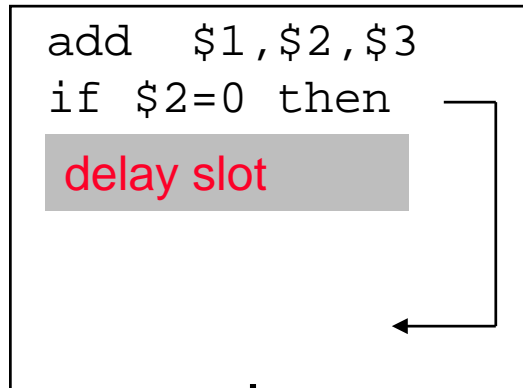


Delayed Branches

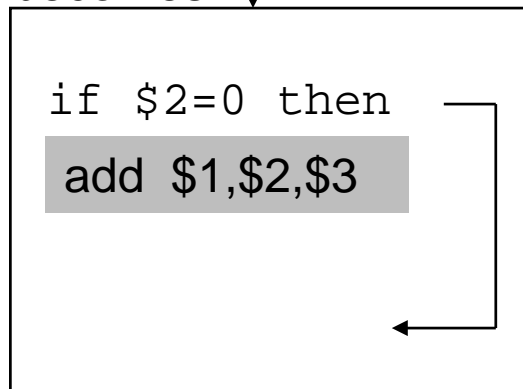
- ❑ If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with **delayed branches** which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect *after* that next instruction
 - ❑ MIPS compiler moves an instruction to immediately after the branch that is not affected by the branch (a **safe** instruction) thereby **hiding** the branch delay
- ❑ With deeper pipelines, the branch delay grows requiring more than one delay slot
 - ❑ Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction
 - ❑ Growth in available transistors has made hardware branch prediction relatively cheaper

Scheduling Branch Delay Slots

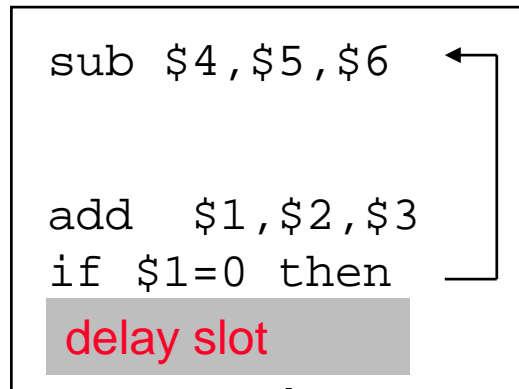
A. From before branch



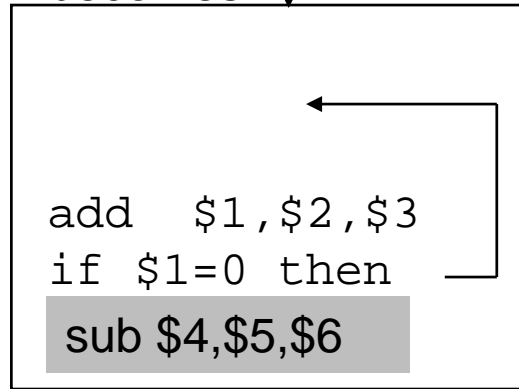
becomes



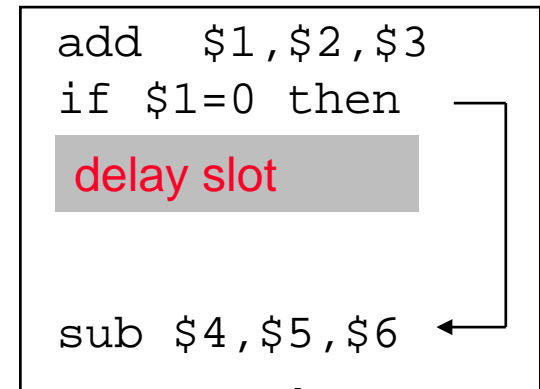
B. From branch target



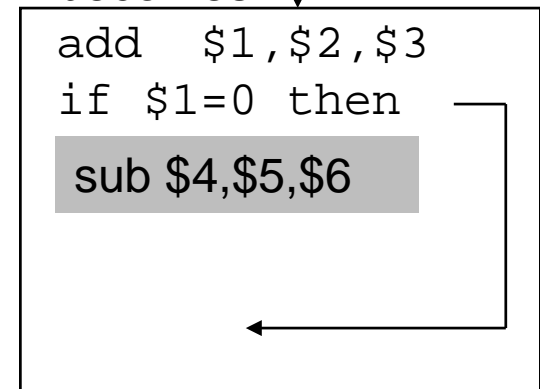
becomes



C. From fall through



becomes

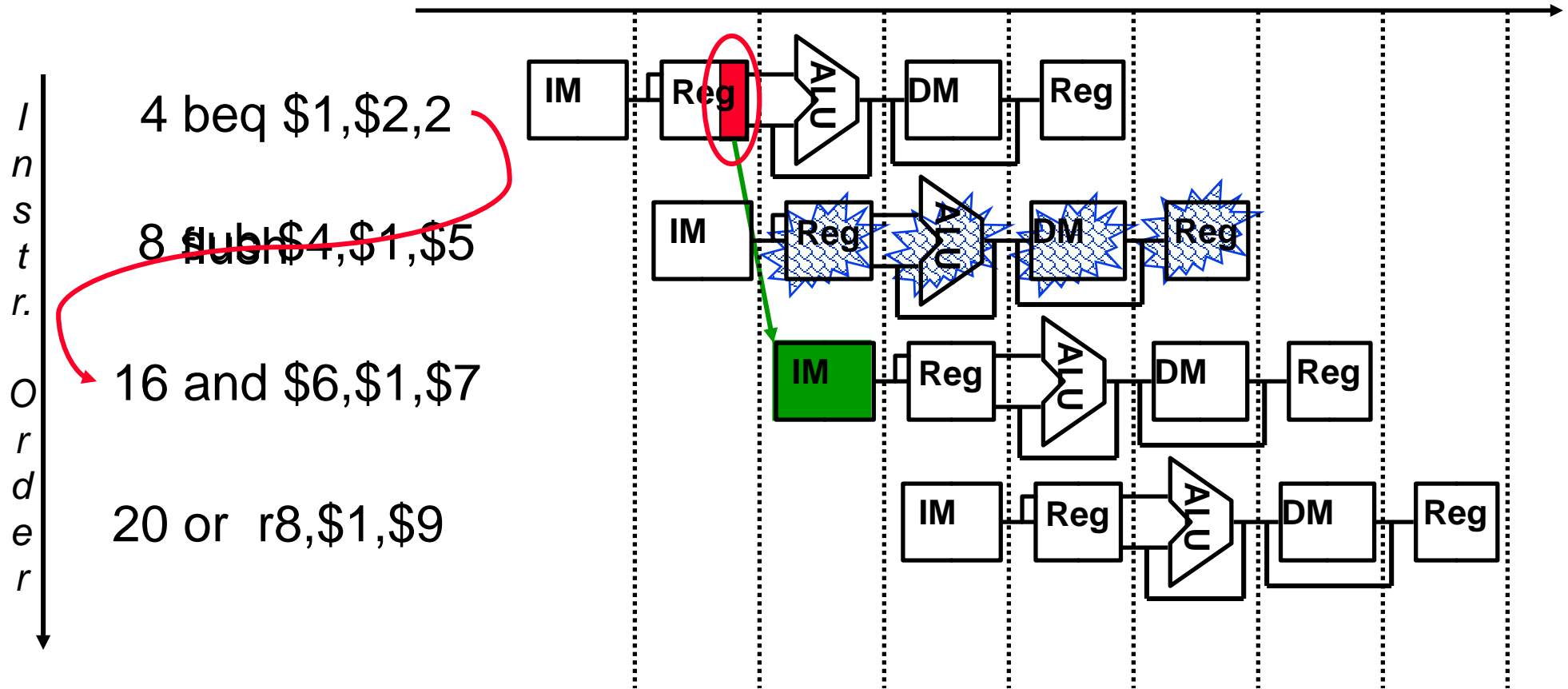


- ❑ A is the best choice, fills delay slot and reduces IC
- ❑ In B and C, the `sub` instruction may need to be copied, increasing IC
- ❑ In B and C, must be okay to execute `sub` when branch fails

Static Branch Prediction

- ❑ Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome
- 1. **Predict not taken** – always predict branches will **not** be taken, continue to fetch from the sequential instruction stream, only when branch *is* taken does the pipeline stall
 - ❑ If taken, **flush** instructions **after** the branch (earlier in the pipeline)
 - in IF, ID, and EX stages if branch logic in MEM – **three** stalls
 - In IF and ID stages if branch logic in EX – **two** stalls
 - in IF stage if branch logic in ID – **one** stall
 - ❑ ensure that those flushed instructions haven't changed the machine state – automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite (in MEM) or RegWrite (in WB))
 - ❑ restart the pipeline at the branch destination

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 `noop`)

Branching Structures

- ❑ Predict not taken works well for “top of the loop” branching structures

- ❑ But such loops have jumps at the bottom of the loop to return to the top of the loop – and incur the jump stall overhead

```
Loop: beq $1,$2,Out
      1nd loop instr
      .
      .
      .
      last loop instr
      j Loop
Out:  fall out instr
```

- ❑ Predict not taken doesn't work well for “bottom of the loop” branching structures

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


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)
- ❑ 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

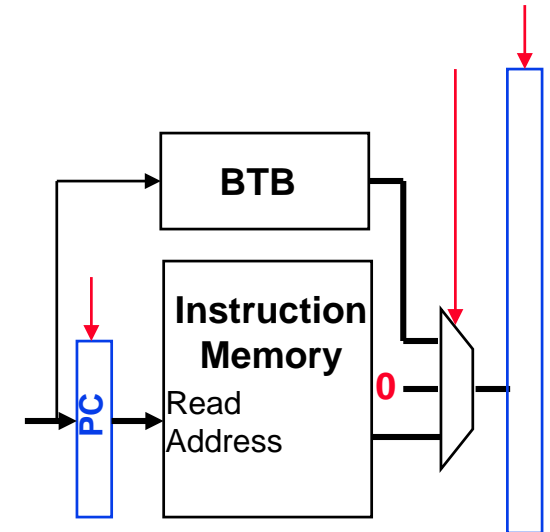
Dynamic Branch Prediction

- 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 executed
 - 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) and gcc at 12%

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`)

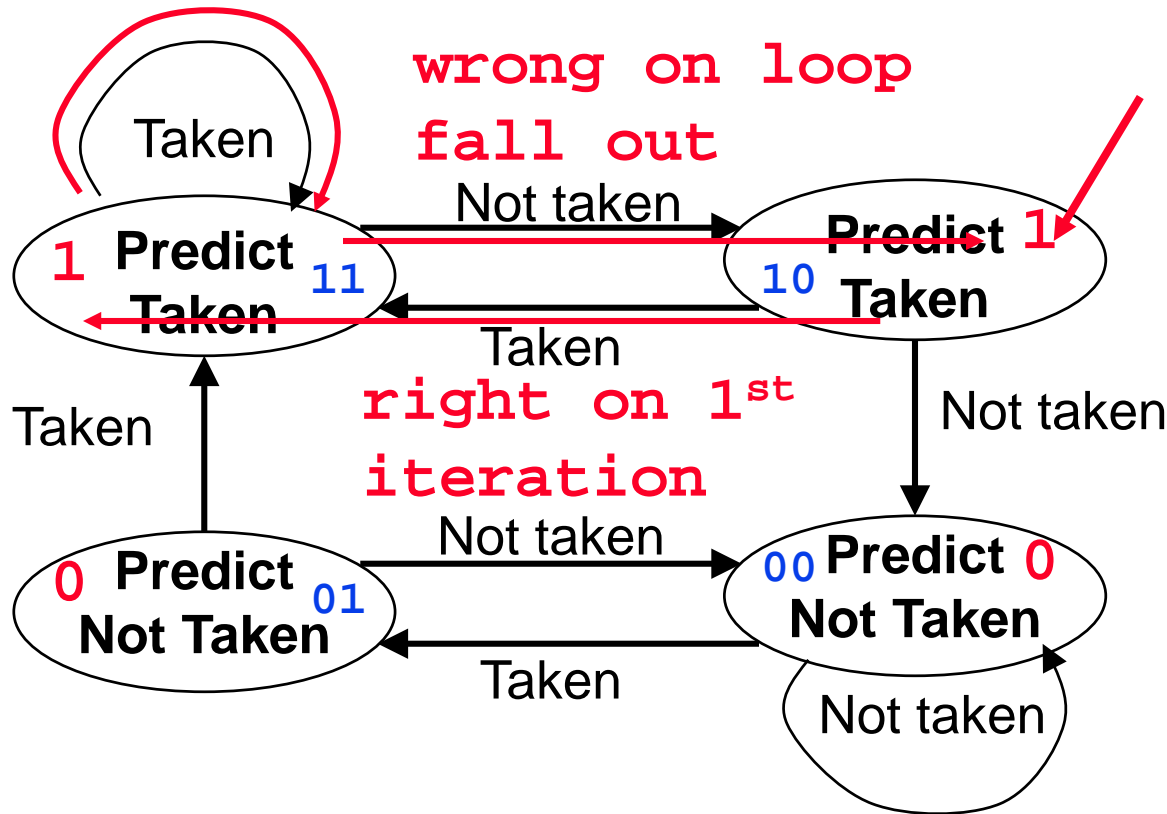
```
Loop: 1st loop instr
      2nd loop instr
      .
      .
      .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

❑ 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

right 9 times



```

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

- BHT also stores the initial FSM state

Dealing with Exceptions

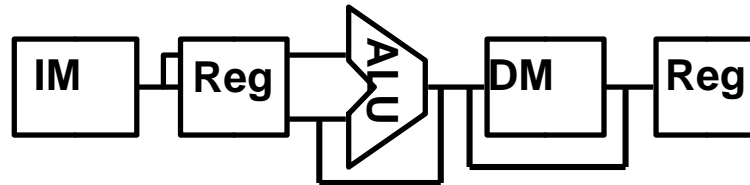
- ❑ Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
 - ❑ R-type arithmetic overflow
 - ❑ Trying to execute an undefined instruction
 - ❑ An I/O device request
 - ❑ An OS service request (e.g., a page fault, TLB exception)
 - ❑ A hardware malfunction
- ❑ The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code)
- ❑ The software (OS) looks at the cause of the exception and “deals” with it

Two Types of Exceptions

- ❑ Interrupts – asynchronous to program execution
 - ❑ caused by **external events**
 - ❑ may be handled **between** instructions, so can let the instructions currently active in the pipeline *complete* before passing control to the OS interrupt handler
 - ❑ simply suspend and resume user program

- ❑ Traps (Exception) – synchronous to program execution
 - ❑ caused by **internal events**
 - ❑ condition must be remedied by the trap handler for **that** instruction, so much stop the offending instruction *midstream* in the pipeline and pass control to the OS trap handler
 - ❑ the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted

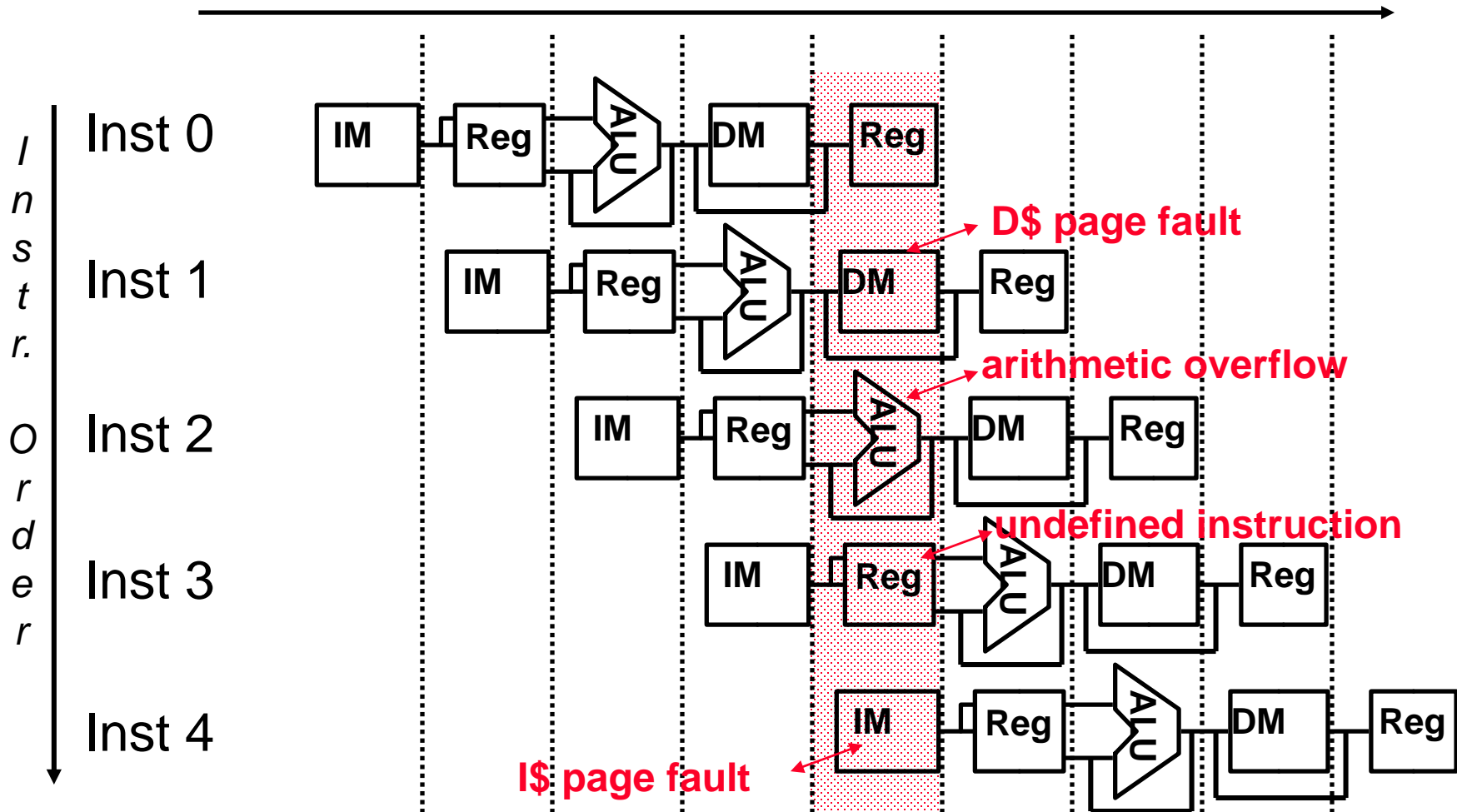
Where in the Pipeline Exceptions Occur



	Stage(s)?	Synchronous?
<input type="checkbox"/> Arithmetic overflow	EX	yes
<input type="checkbox"/> Undefined instruction	ID	yes
<input type="checkbox"/> TLB or page fault	IF, MEM	yes
<input type="checkbox"/> I/O service request	any	no
<input type="checkbox"/> Hardware malfunction	any	no

Beware that multiple exceptions can occur simultaneously in a *single* clock cycle

Multiple Simultaneous Exceptions

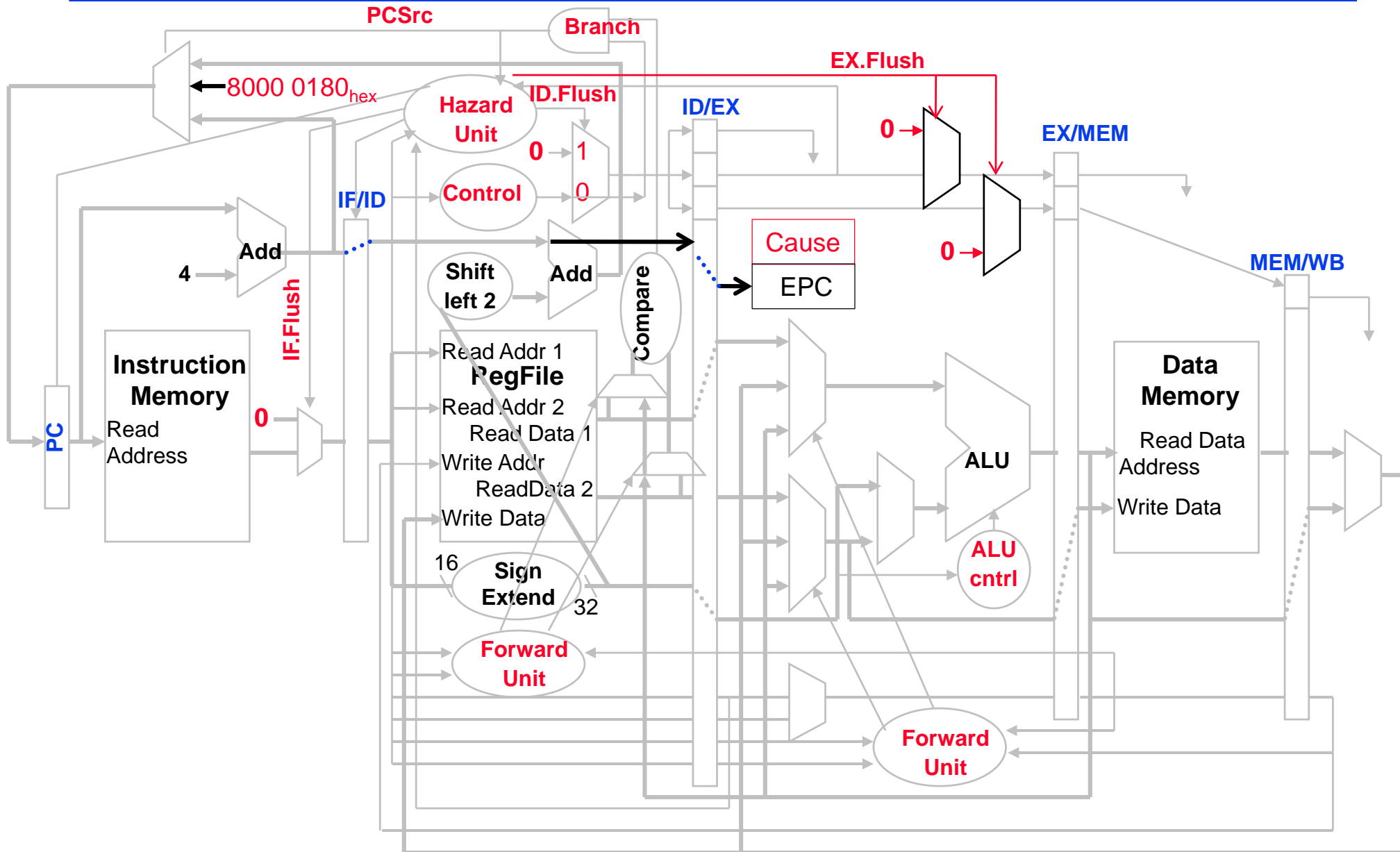


- ❑ Hardware sorts the exceptions so that the earliest instruction is the one interrupted first

Additions to MIPS to Handle Exceptions

- ❑ Cause register (records exceptions) – hardware to record in Cause the exceptions and a signal to control writes to it (CauseWrite)
- ❑ EPC register (records the addresses of the offending instructions) – hardware to record in EPC the address of the offending instruction and a signal to control writes to it (EPCWrite)
 - ❑ Exception software must match exception to instruction
- ❑ A way to load the PC with the address of the exception handler
 - ❑ Expand the PC input mux where the new input is hardwired to the exception handler address - (e.g., 8000 0180_{hex} for arithmetic overflow)
- ❑ A way to flush offending instruction and the ones that follow it

Datapath with Controls for Exceptions



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