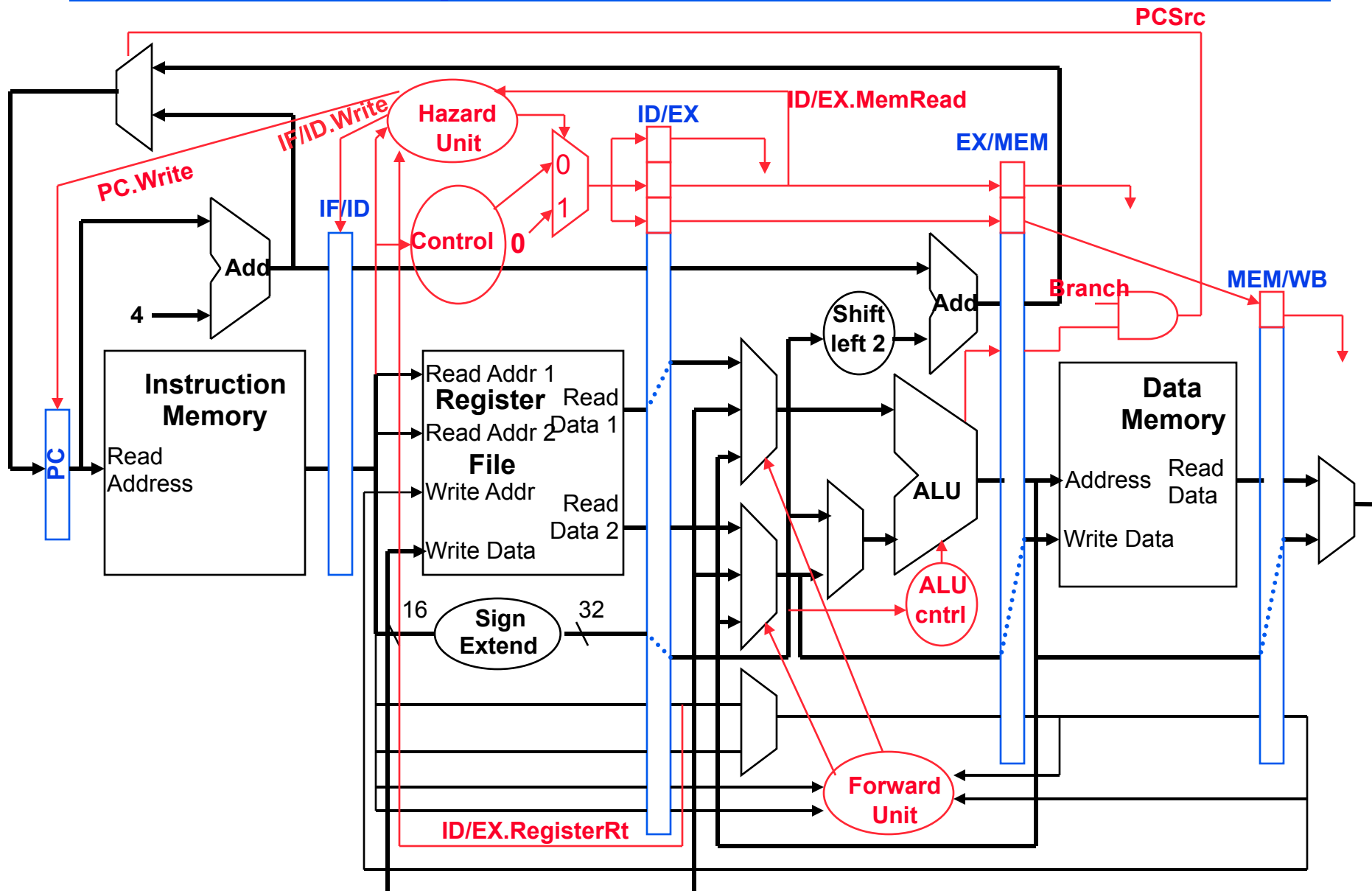


---

# Containing Control Hazards

[Adapted from Mary Jane Irwin for  
*Computer Organization and Design*,  
Patterson & Hennessy, © 2005, UCB]

# Review: Datapath with Data Hazard Control

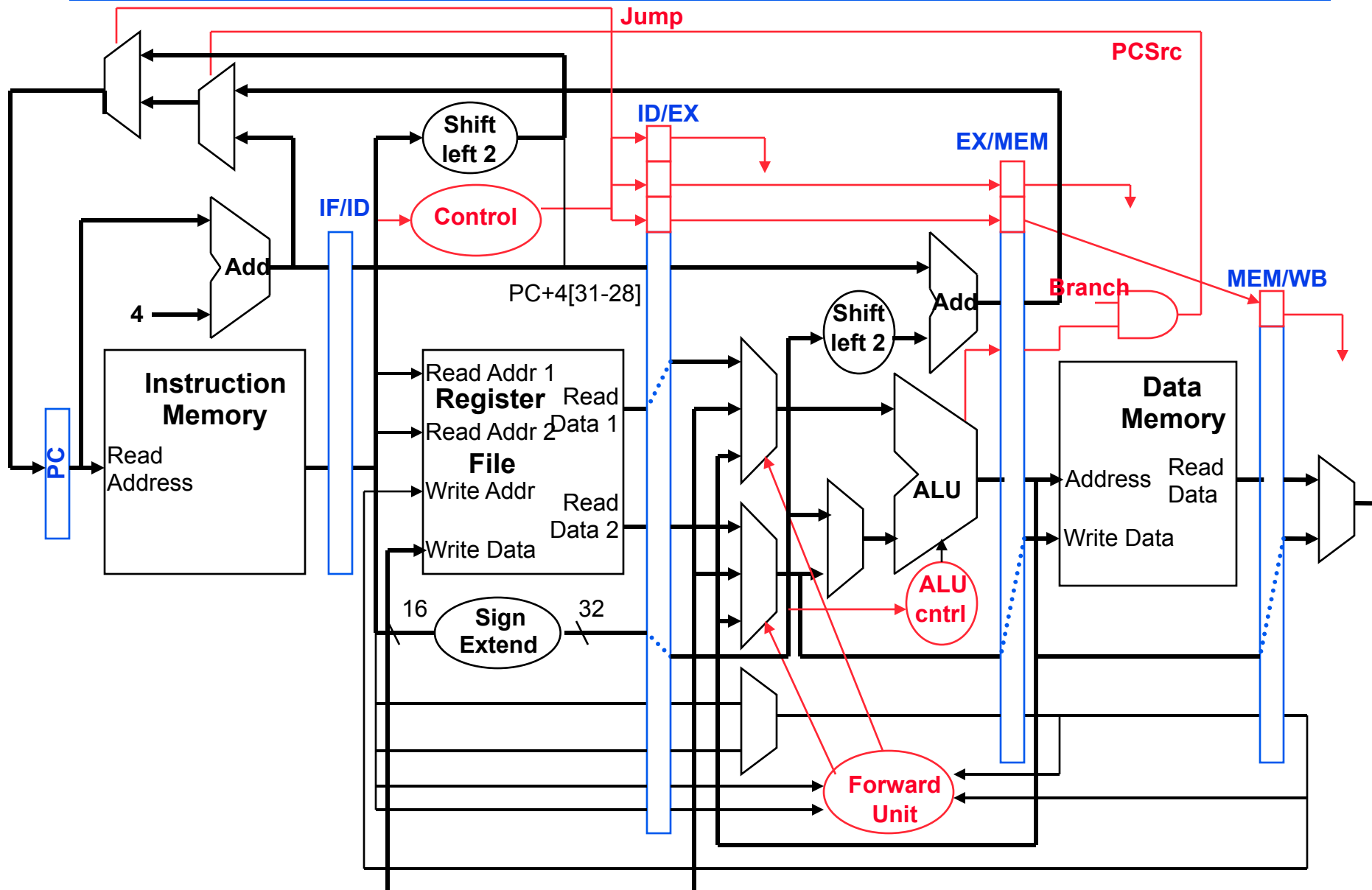


# Control Hazards

---

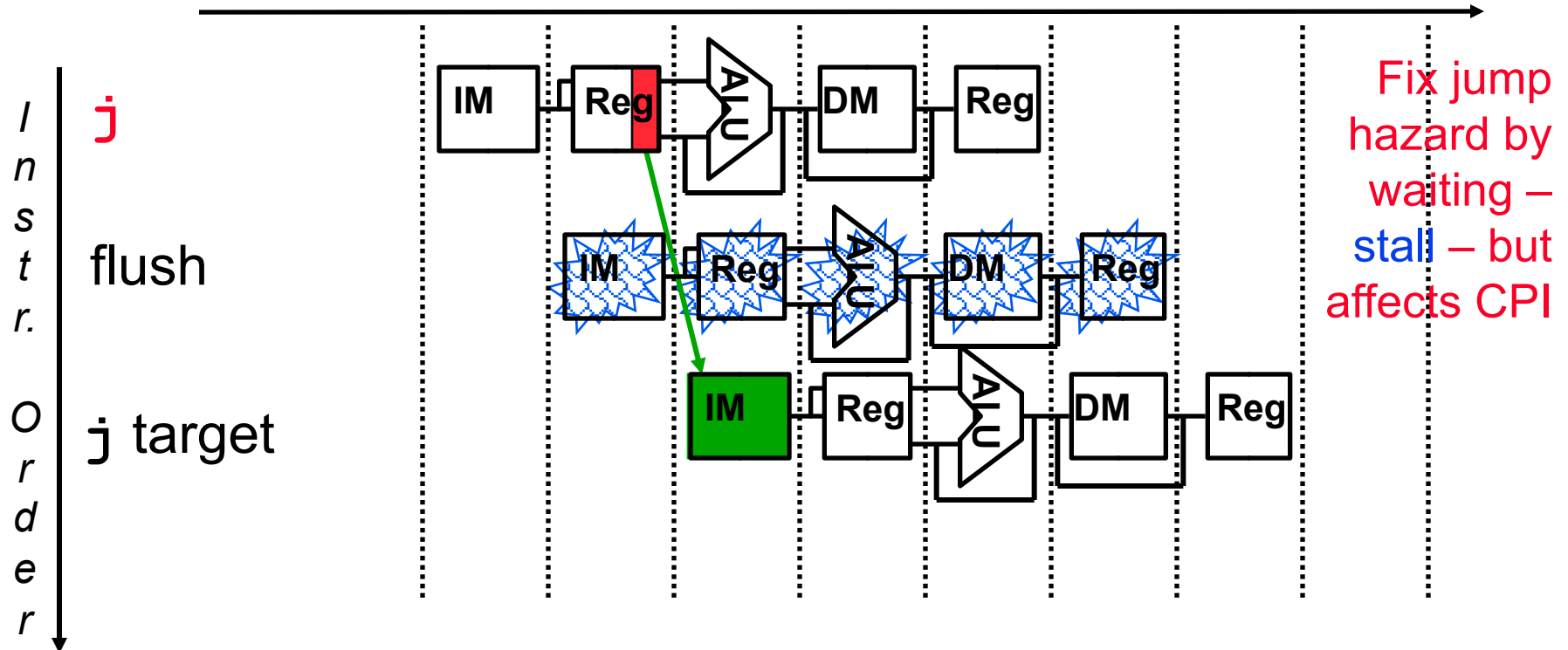
- ❑ When the flow of instruction addresses is not sequential (i.e.,  $PC = PC + 4$ ); incurred by change of flow instructions
  - Conditional branches (`beq`, `bne`)
  - Unconditional branches (`j`, `jal`, `jr`)
  - Exceptions
- ❑ Possible approaches
  - Stall (impacts CPI)
  - Move decision point as 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



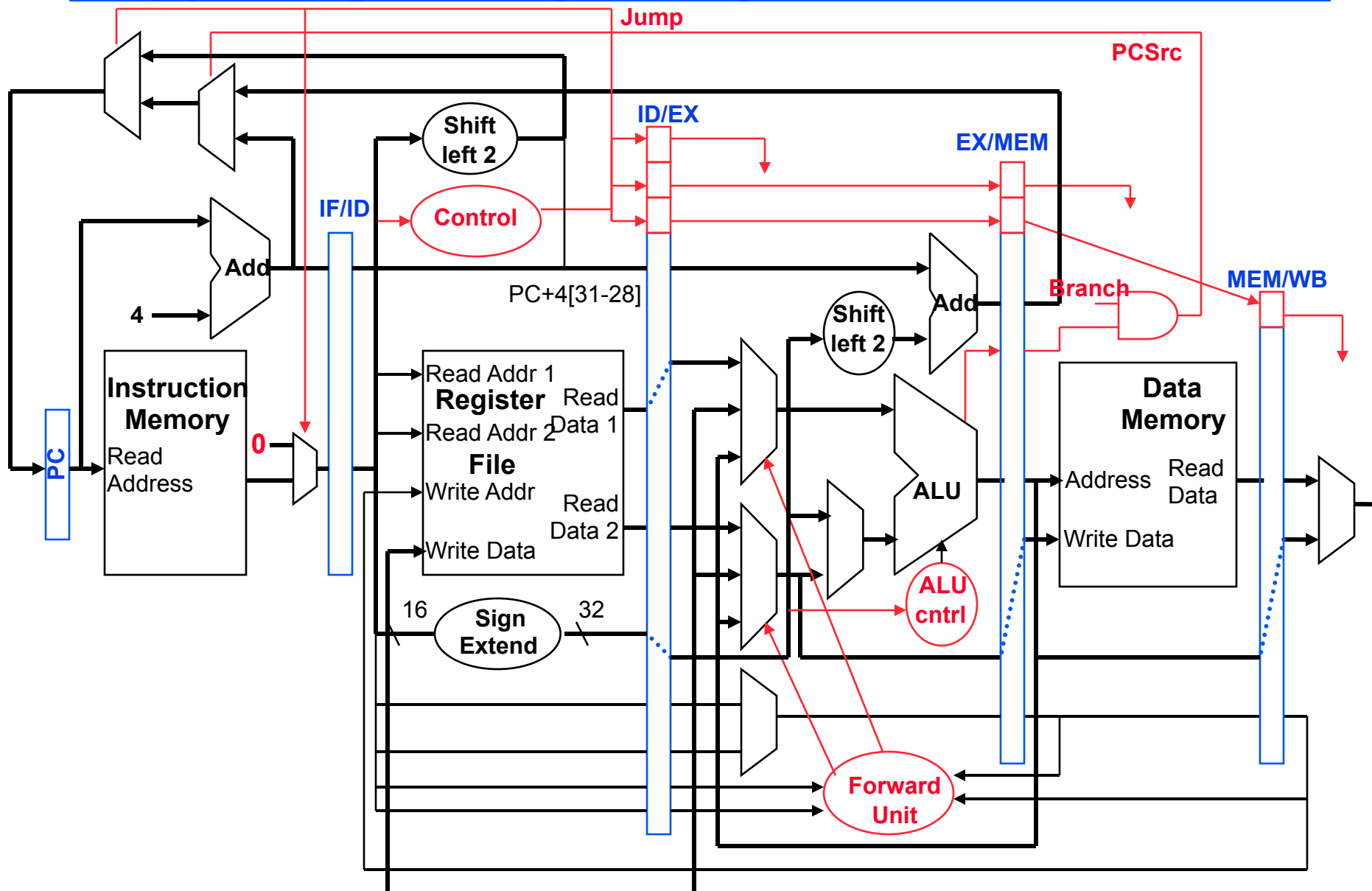
# Jumps Incur One Stall

- ❑ Jumps not decoded until ID, so one **flush** is needed



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

# Supporting ID Stage Jumps

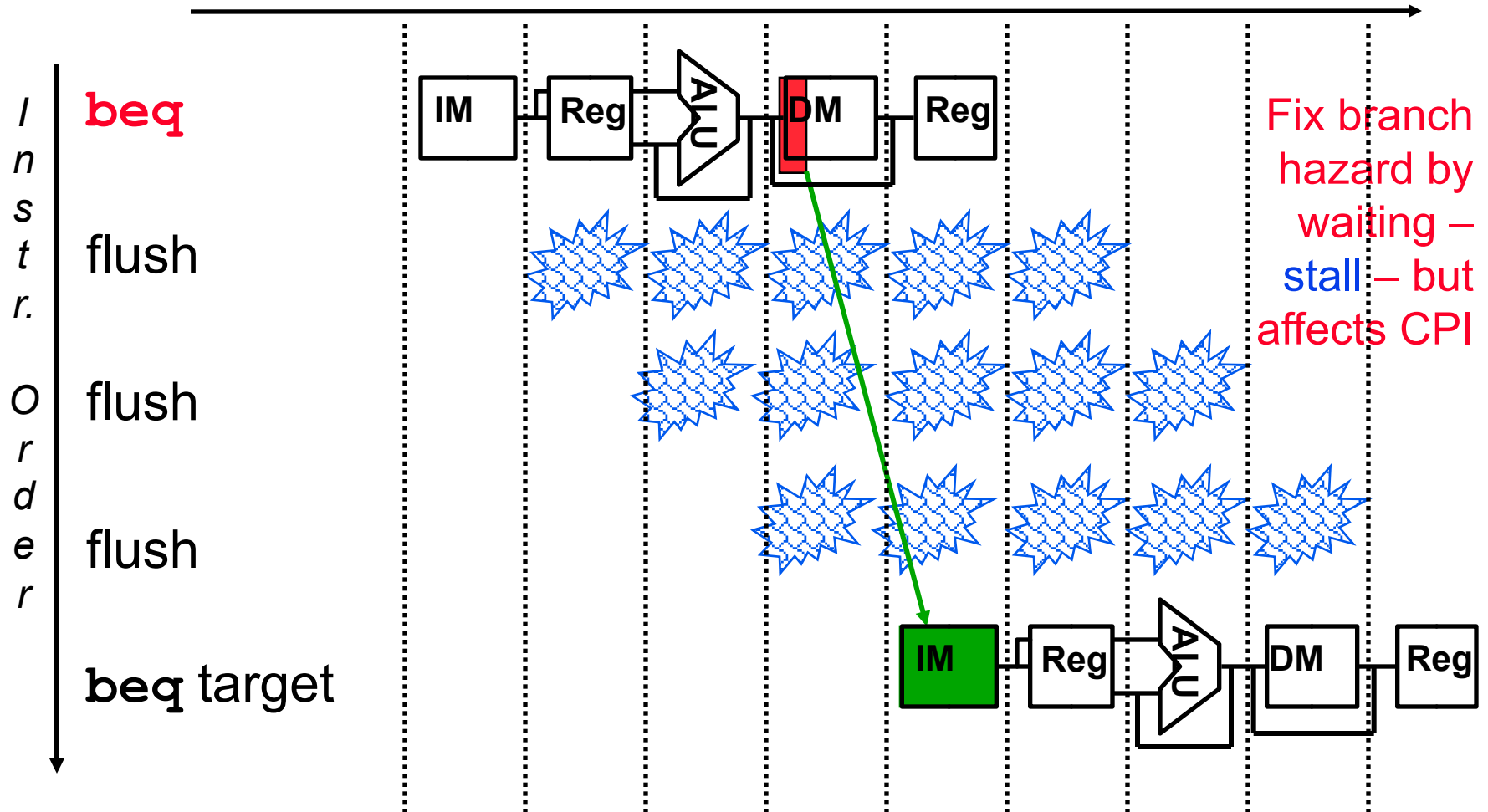


# Two “Types” of Stalls

---

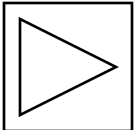
- ❑ Noop instruction (or bubble) **inserted** between two instructions in the pipeline (as done for load-use situations)
  - Keep the instructions *earlier* in the pipeline (later in the code) from progressing down the pipeline for a cycle (“bounce” them in place with write control signals)
  - Insert `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

# Review: Branches Incur Three Stalls





# Moving Branch Decisions Earlier in Pipe

- ❑ 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

WB	add3	\$1,
MEM	add2	\$3,
EX	add1	\$4,
ID	beq	\$1, \$2, Loop
IF	next_seq_instr	

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

WB	add3	\$3,
MEM	add2	\$1,
EX	add1	\$4,
ID	beq	\$1, \$2, Loop
IF	next_seq_instr	

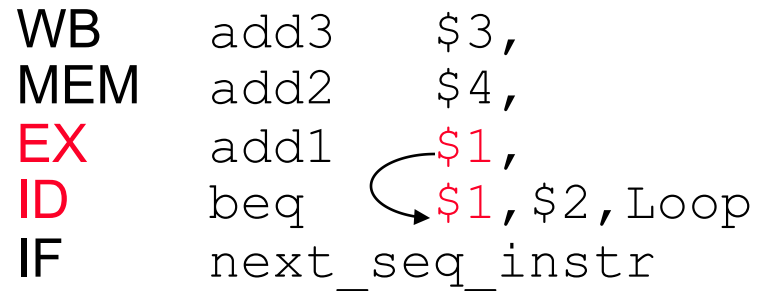
```

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

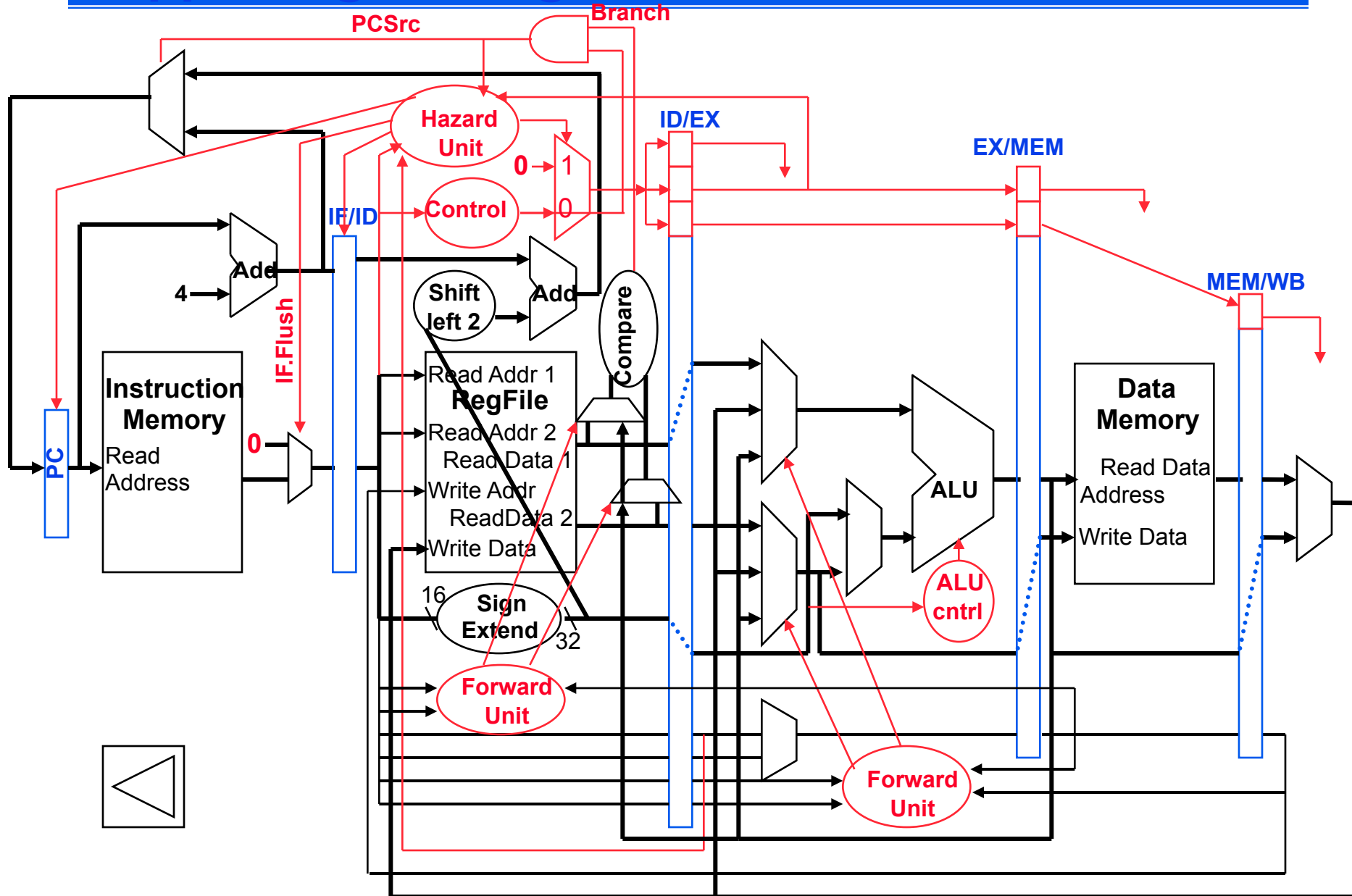
# ID Branch Forwarding Issues, con' t

- ❑ 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**)

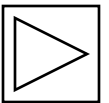
# Supporting ID Stage Branches



# Delayed Decision

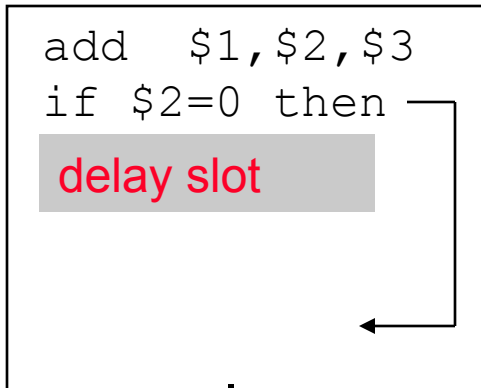
---

- ❑ 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

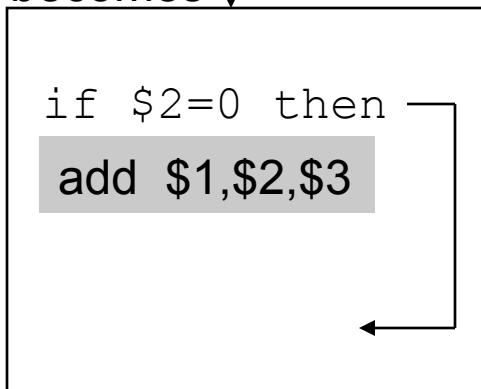


# Scheduling Branch Delay Slots

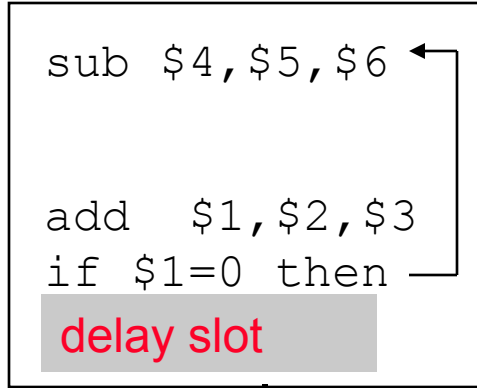
A. From before branch



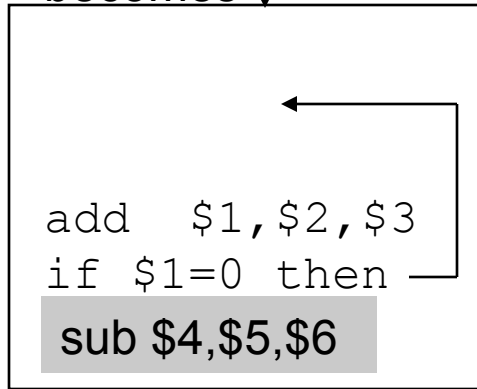
becomes



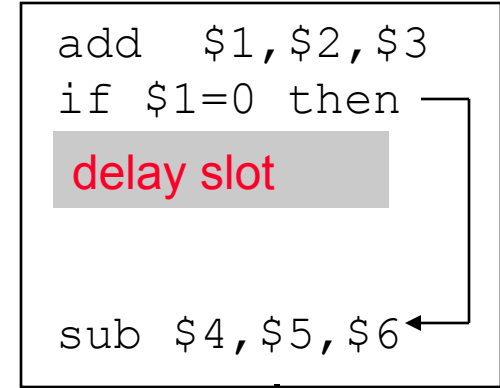
B. From branch target



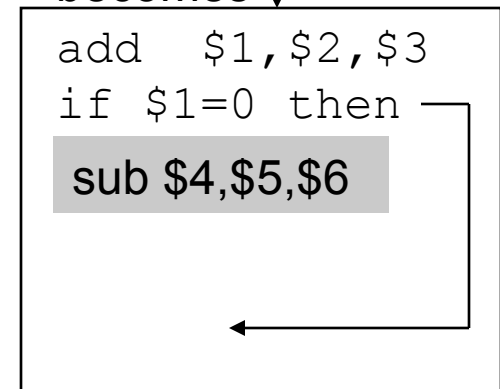
becomes



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

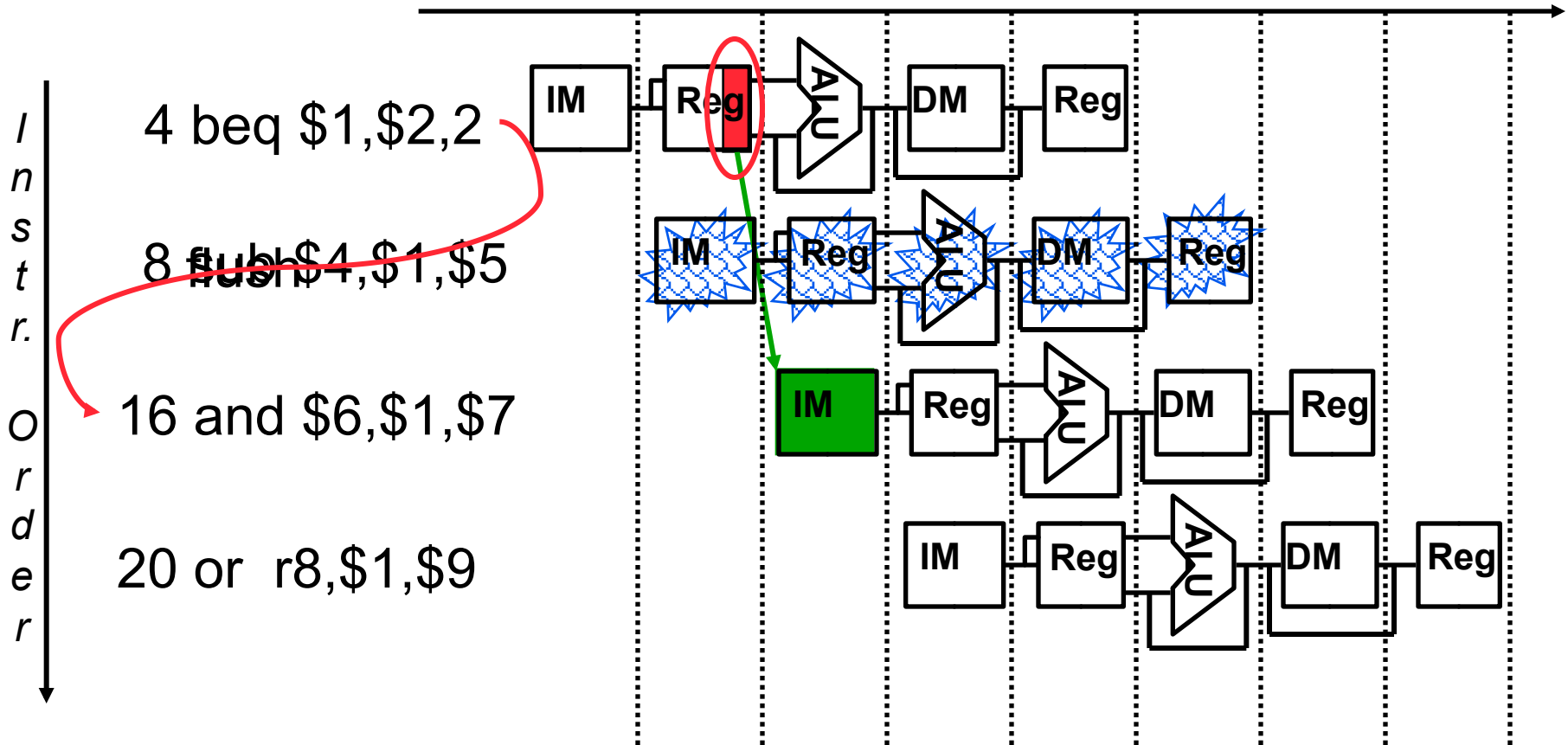


# Static Branch Prediction

---

- ❑ 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
      1st 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)
  - 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

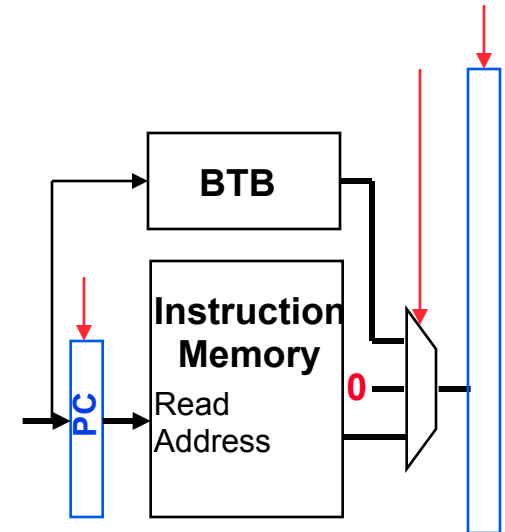
---

- ❑ A **branch prediction buffer** (aka branch history table (**BHT**)) in the IF stage addressed by the lower bits of the PC, contains a bit 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
  - If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit
    - 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 can cache 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

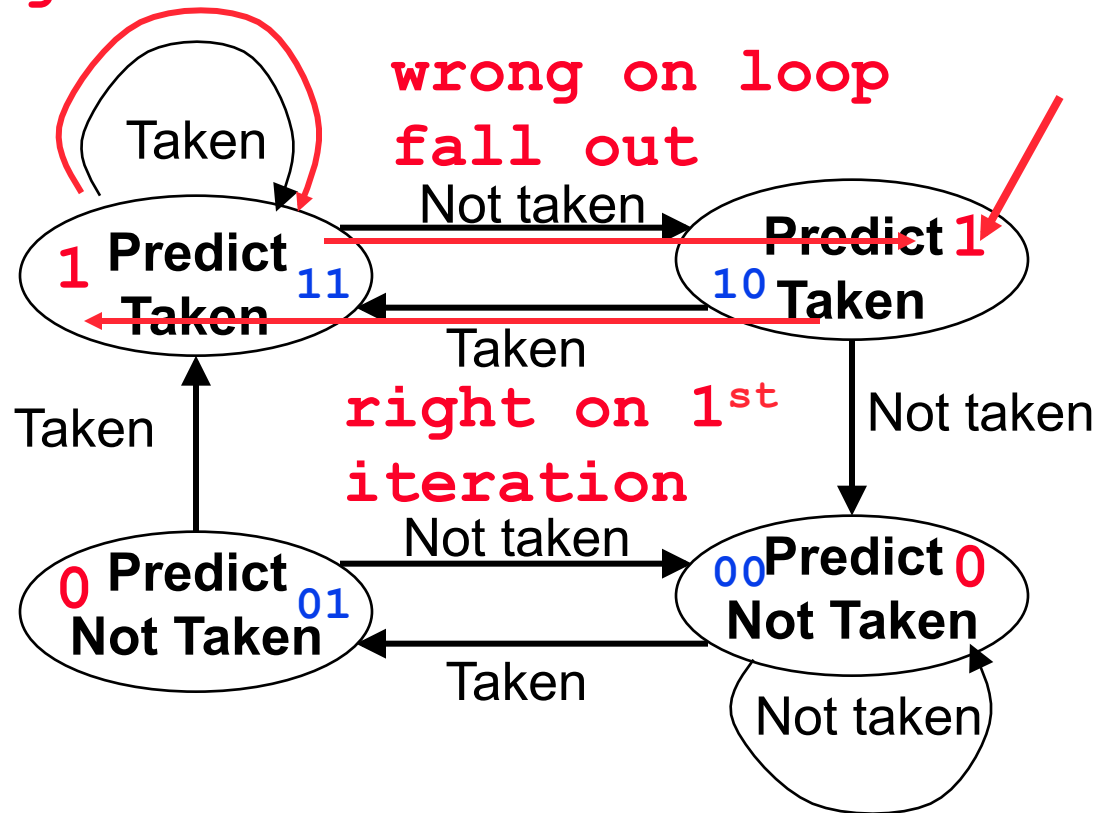
- ❑ A 1-bit predictor will be incorrect twice when not taken
  - Assume `predict_bit = 0` to start (indicating branch not taken) and loop control is at the bottom of the loop code
  - 1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the 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`)
- ❑ For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

```
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

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

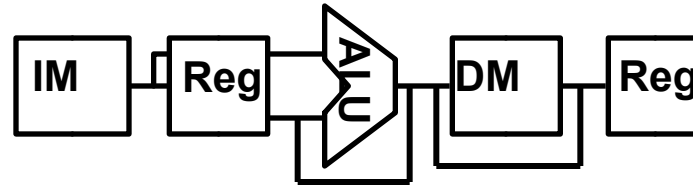
# Two Types of Exceptions

---

- ❑ 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



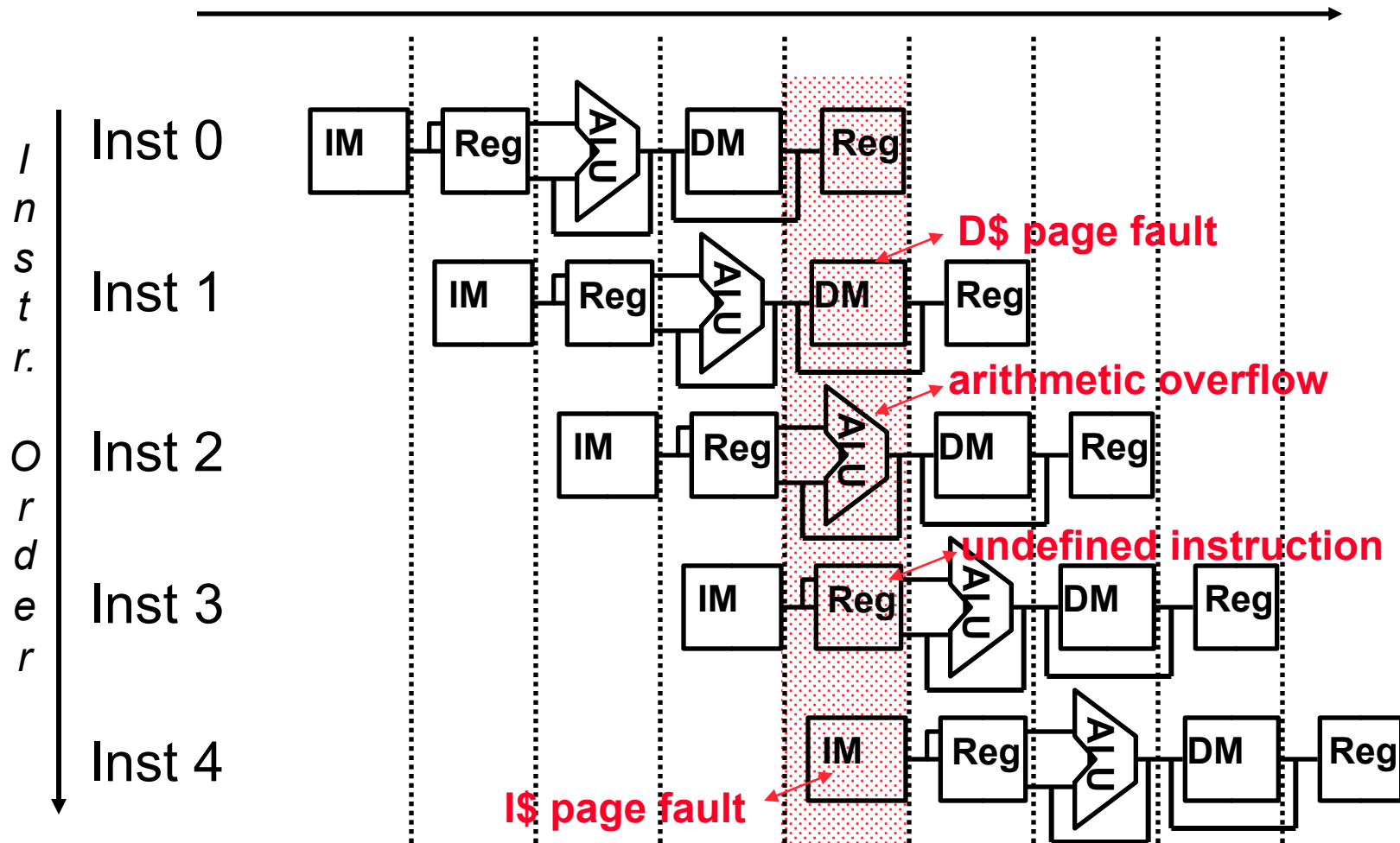
# Where in the Pipeline Exceptions Occur



	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<sub>hex</sub> for arithmetic overflow)
- ❑ A way to flush offending instruction and the ones that follow it

# 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)