# CMPEN 431
# Computer Architecture
# Fall 2018

## Dealing with Branches and Exceptions

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 & 5th Edition, © 2014 ]

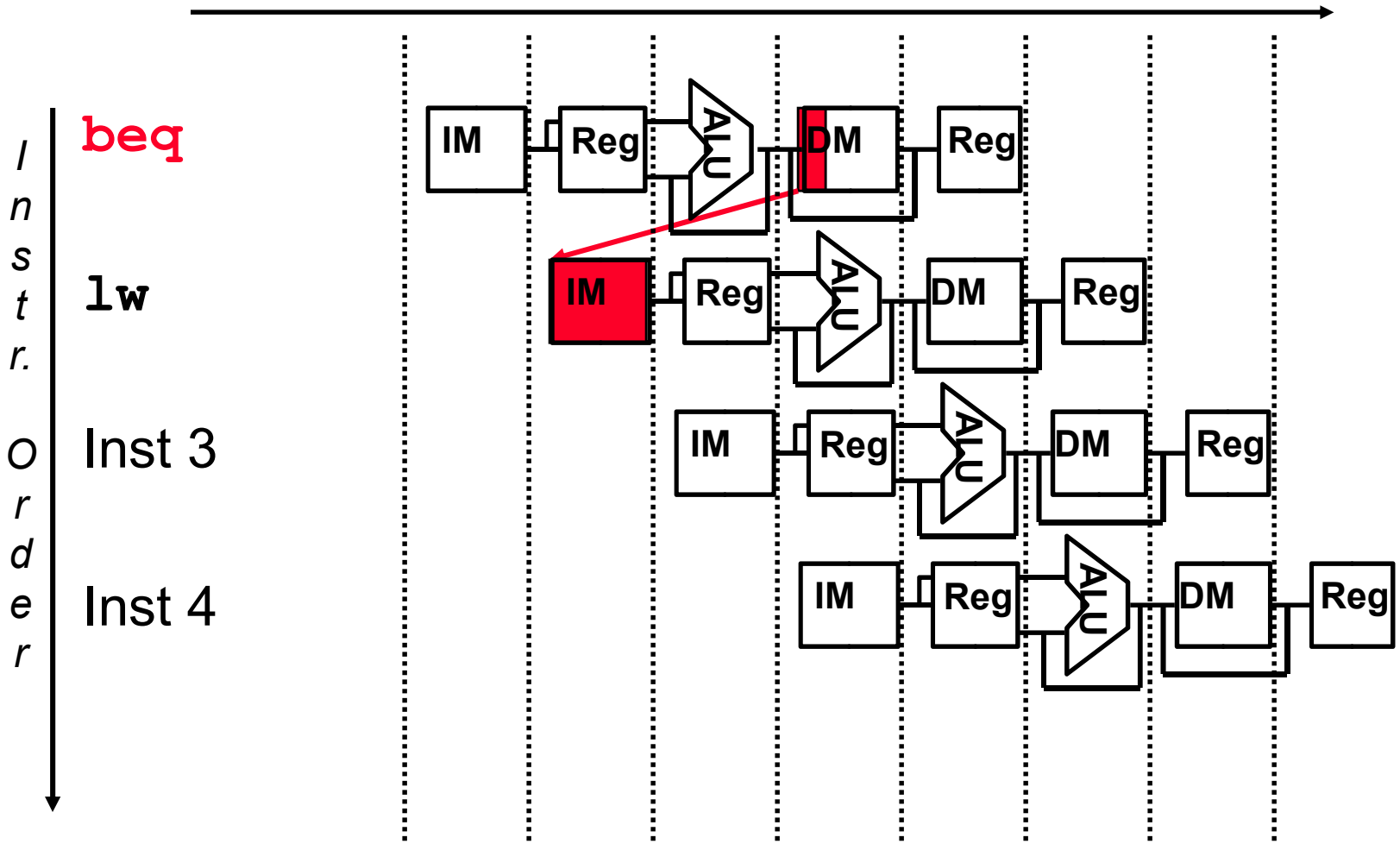
# Review: Pipelining - What Makes it Hard ?

- 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: Control Hazards

- Dependencies backward in time cause hazards



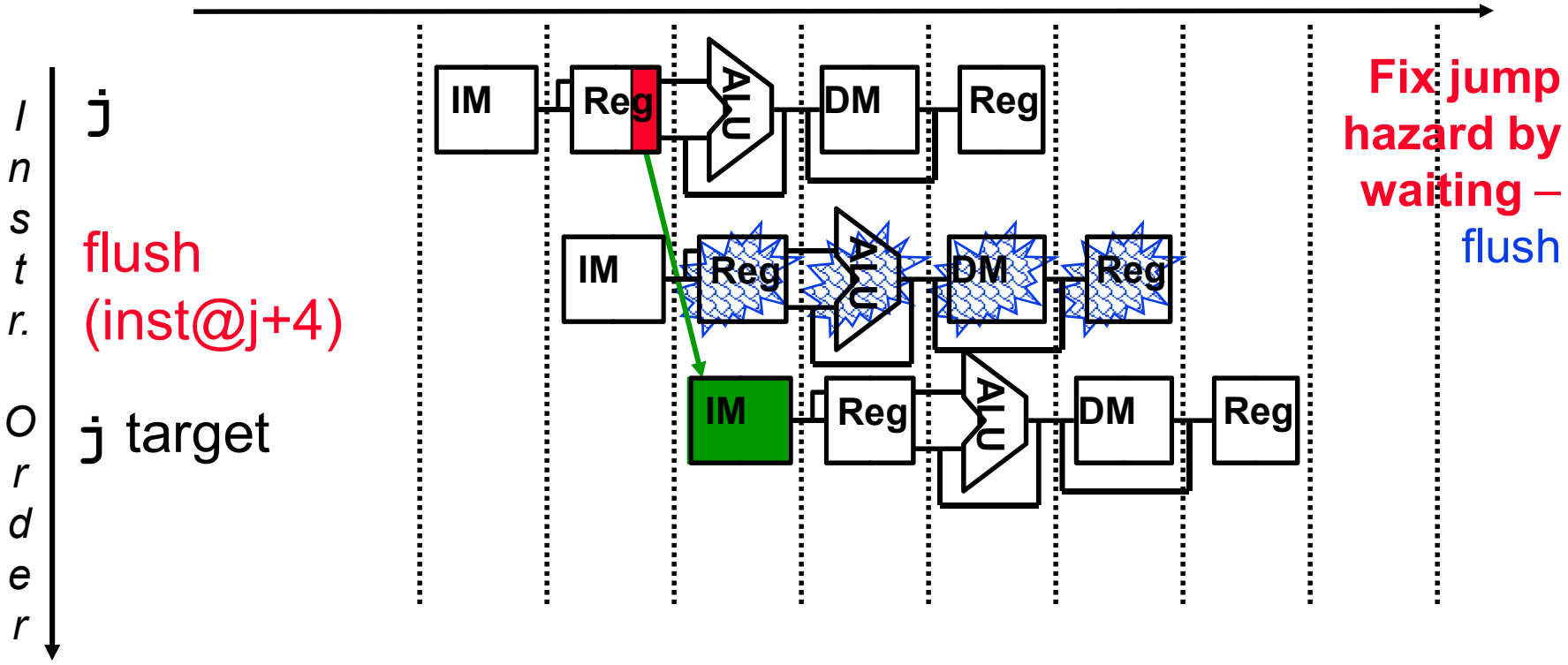

- Which instruction is executed after the branch ?

# To Stall or to Flush ?

- Stall (or bubble) where a `noop` instruction is inserted between two instructions in the pipeline (as done for load-use)
  - 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 to the pipeline registers)
  - Insert `noop` by zeroing the control bits in the pipeline register at the appropriate stage (with `IF/ID.Bubble`)
  - Let the instructions later in the pipeline (earlier in the code) progress normally down the pipeline
- Flush (or instruction squashing) where an instruction in the pipeline is replaced with a `noop` instruction (as done for instructions located sequentially after `j` instructions with `IF/ID.Flush`)
  - Zero the control bits in the IF/ID pipeline register of the instruction to be flushed (the one just after the `j` instruction)

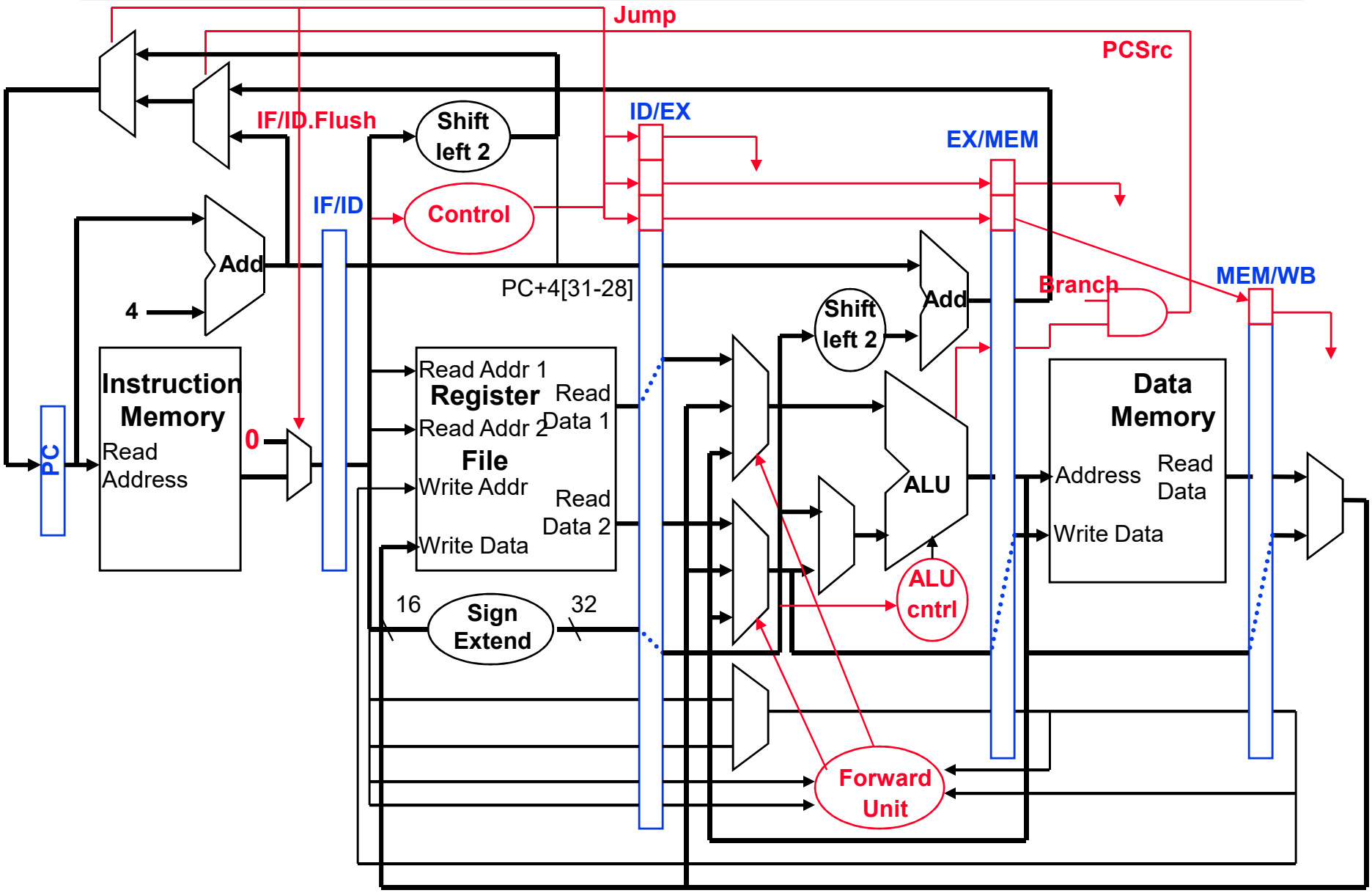
# Aside: Jumps Aren't the Big Worry

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



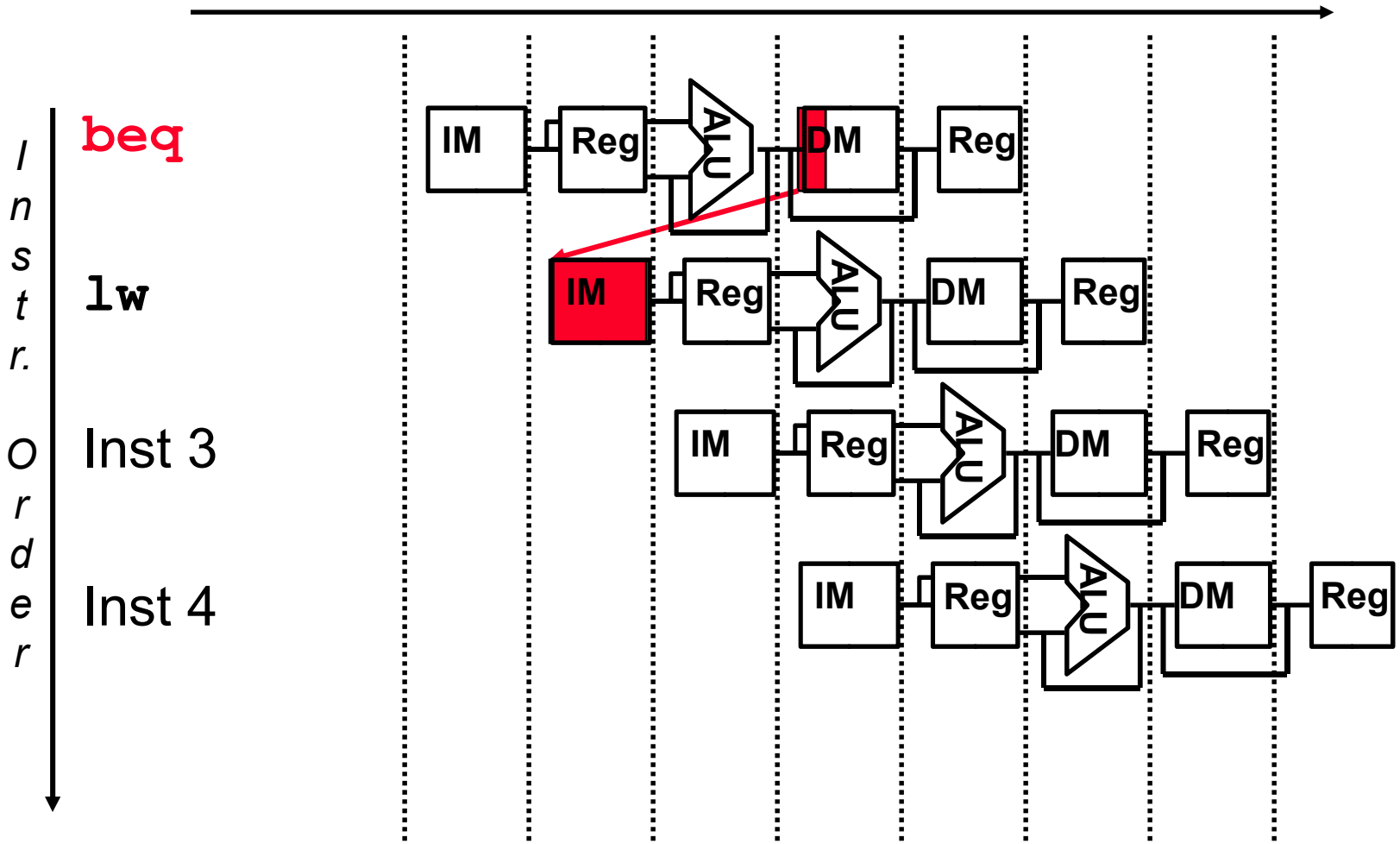

- Fortunately, jumps are very infrequent – only ~3% of the SPECint instruction mix

# Supporting ID Stage Jumps



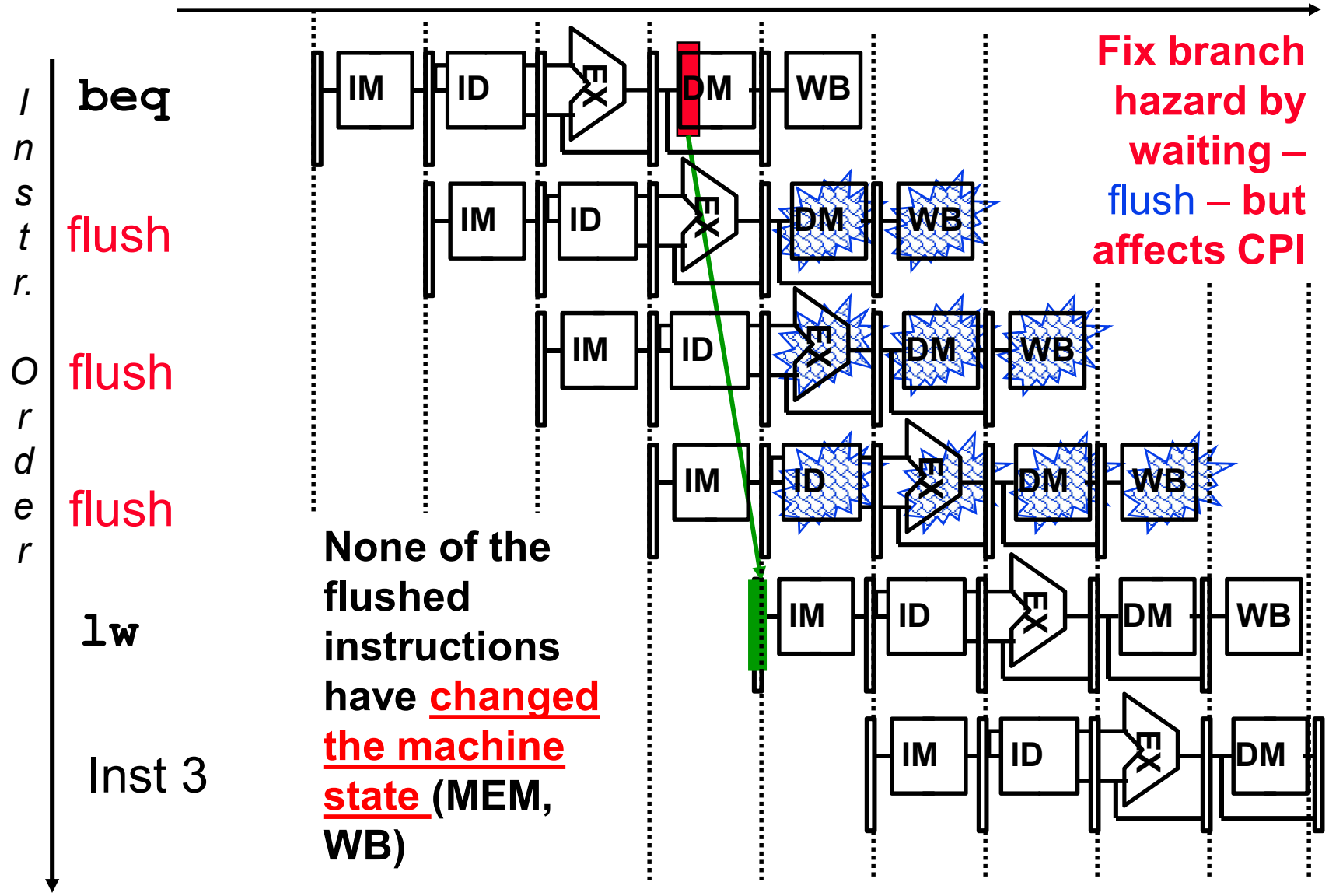
# … and now back to branches

- Dependencies backward in time cause hazards




- Which instruction is executed after the branch ?

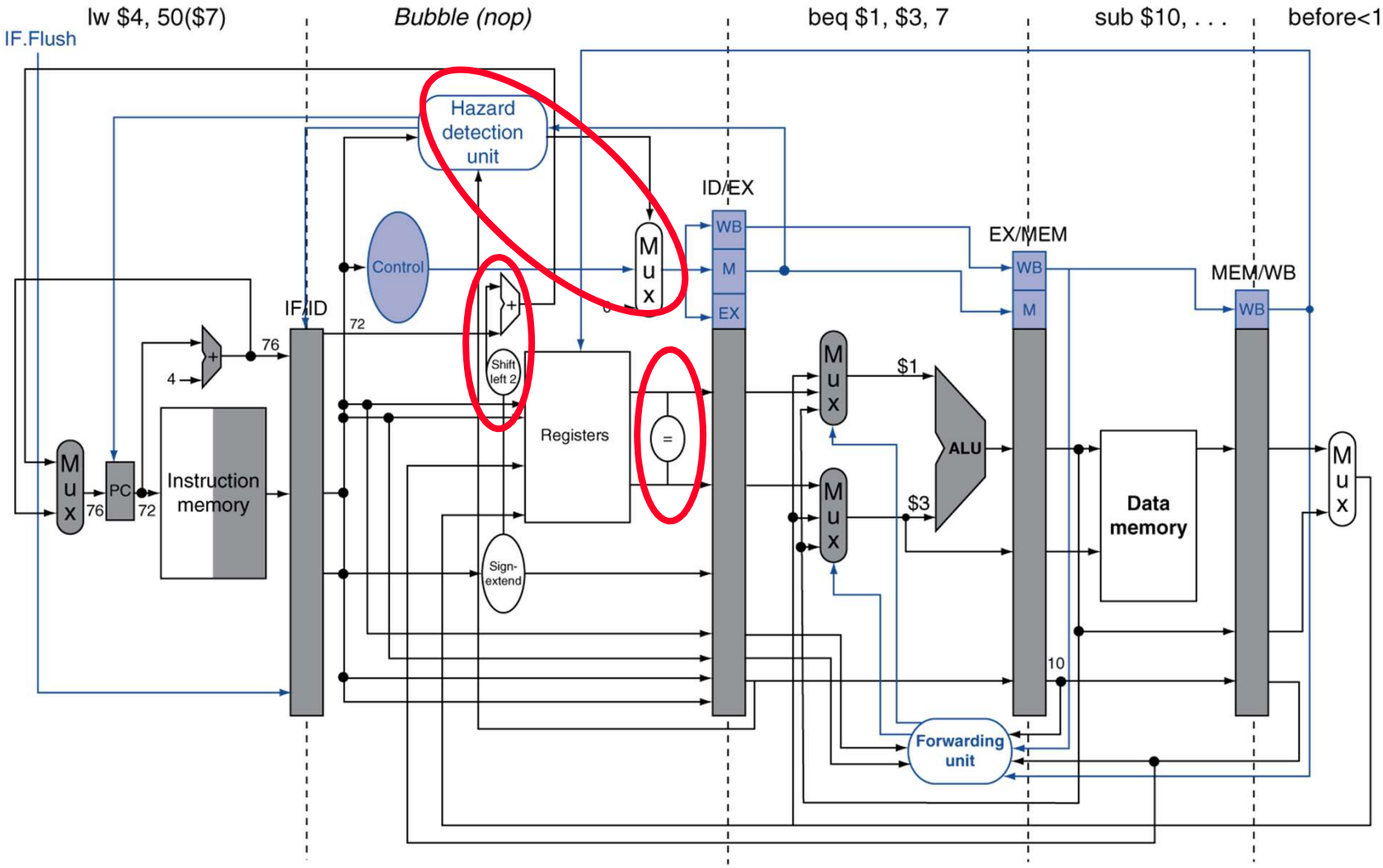
# One Way to "Fix" a Branch Control Hazard



# Reducing the Delay of Branches

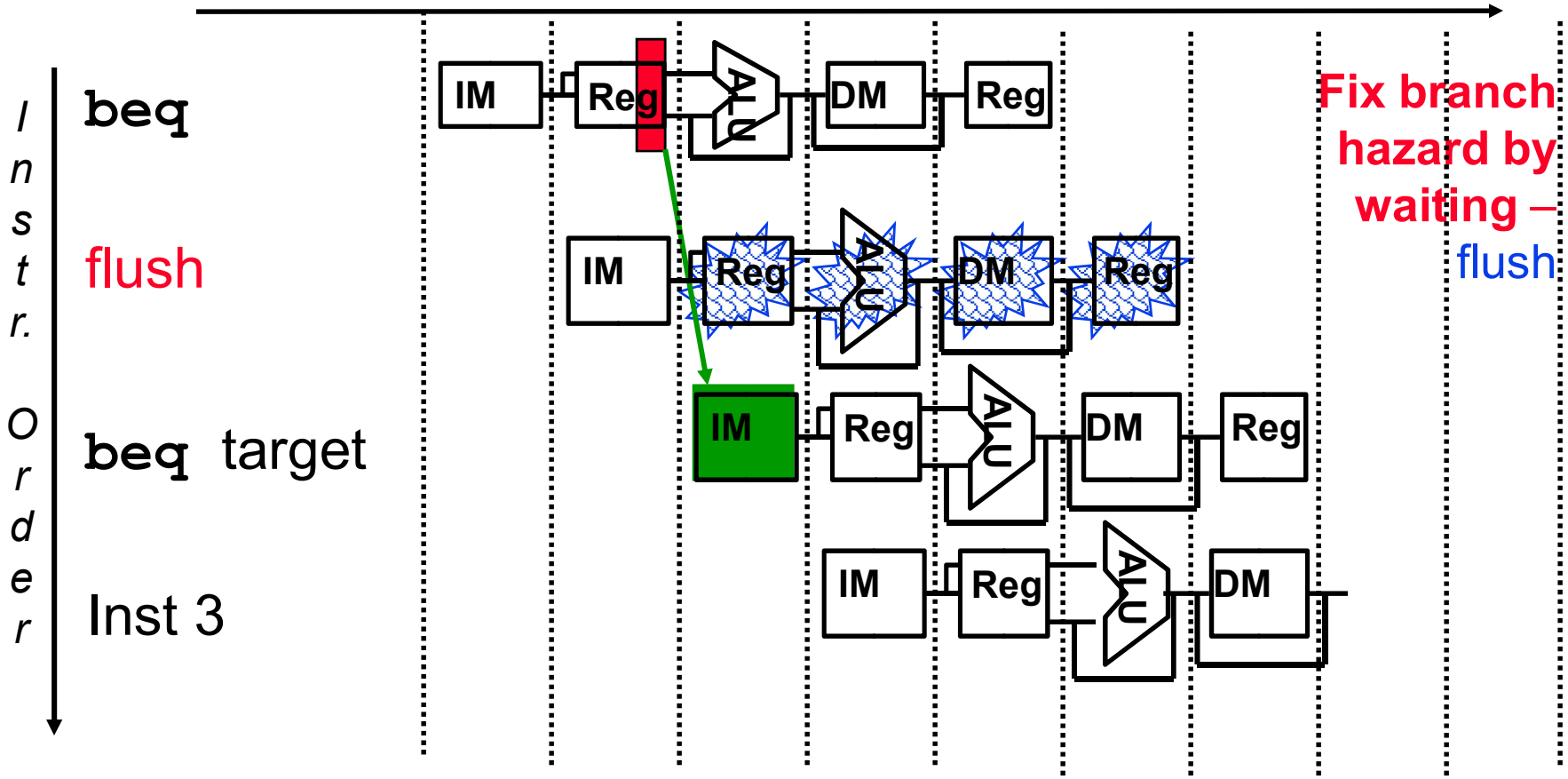
- Move the branch decision hardware back to the EX stage
  - Reduces the number of stall (flush) cycles to two
  - Adds an `and` gate and a 2x1 `mux` to the EX timing path
- Move the branch decision hardware back to the ID stage
  - Reduces the number of stall (flush) cycles to one (like with jumps)
    - 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

# Branch Decision Hardware in ID



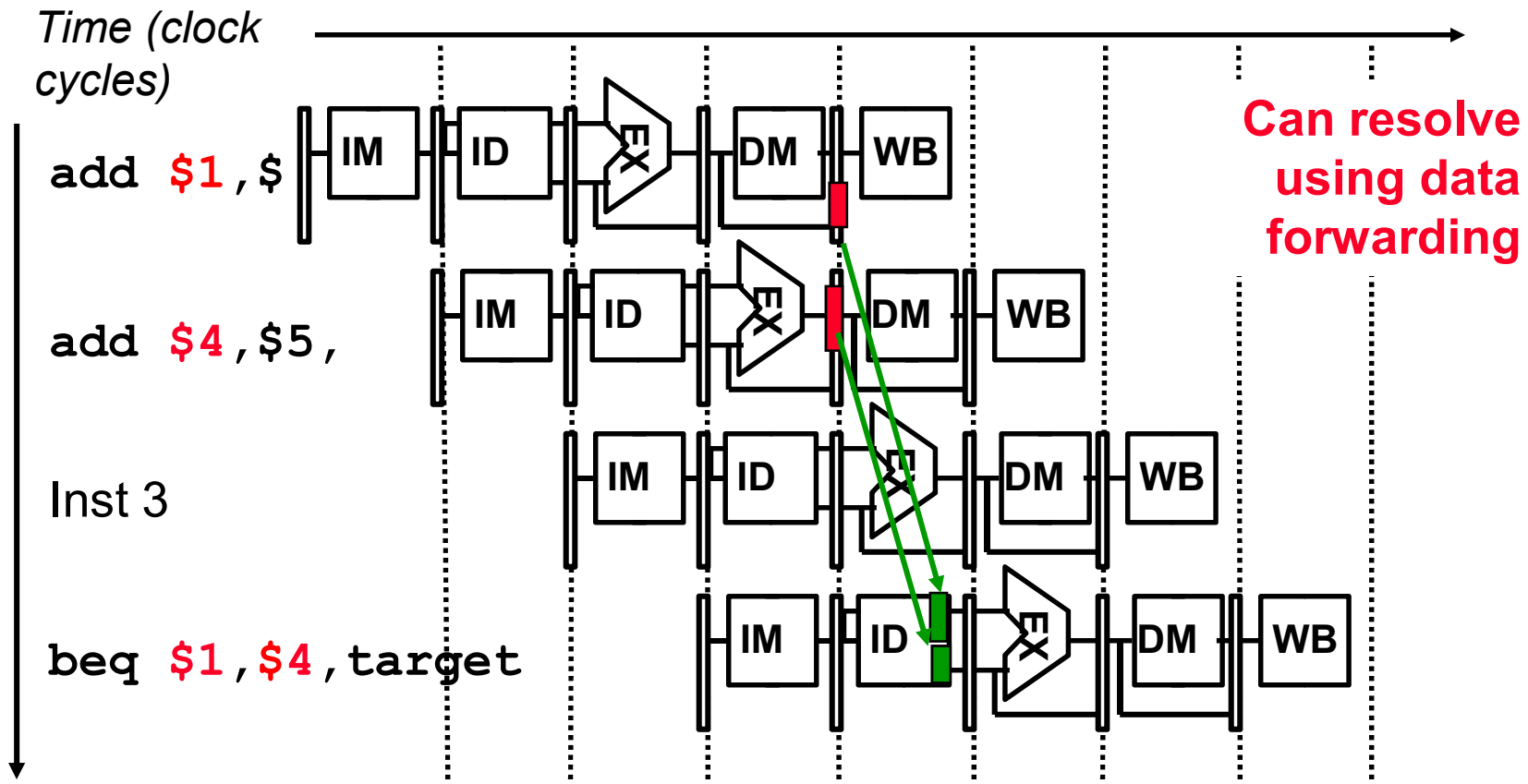
# Branch resolution in ID – Pipeline Diagram

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



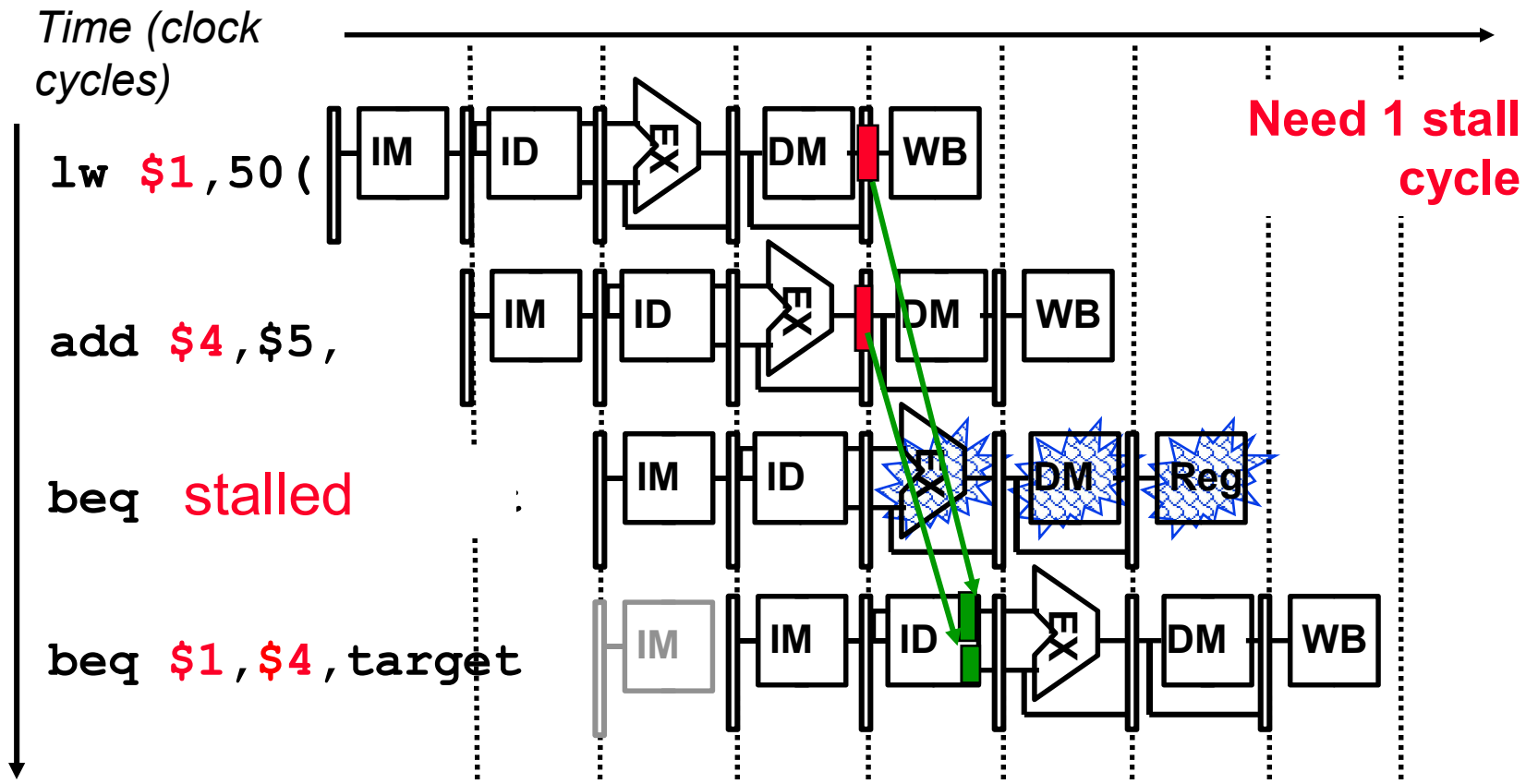
# Yet Another Complication

- What if a comparison register is a destination of the 2nd or 3rd preceding ALU instruction ?



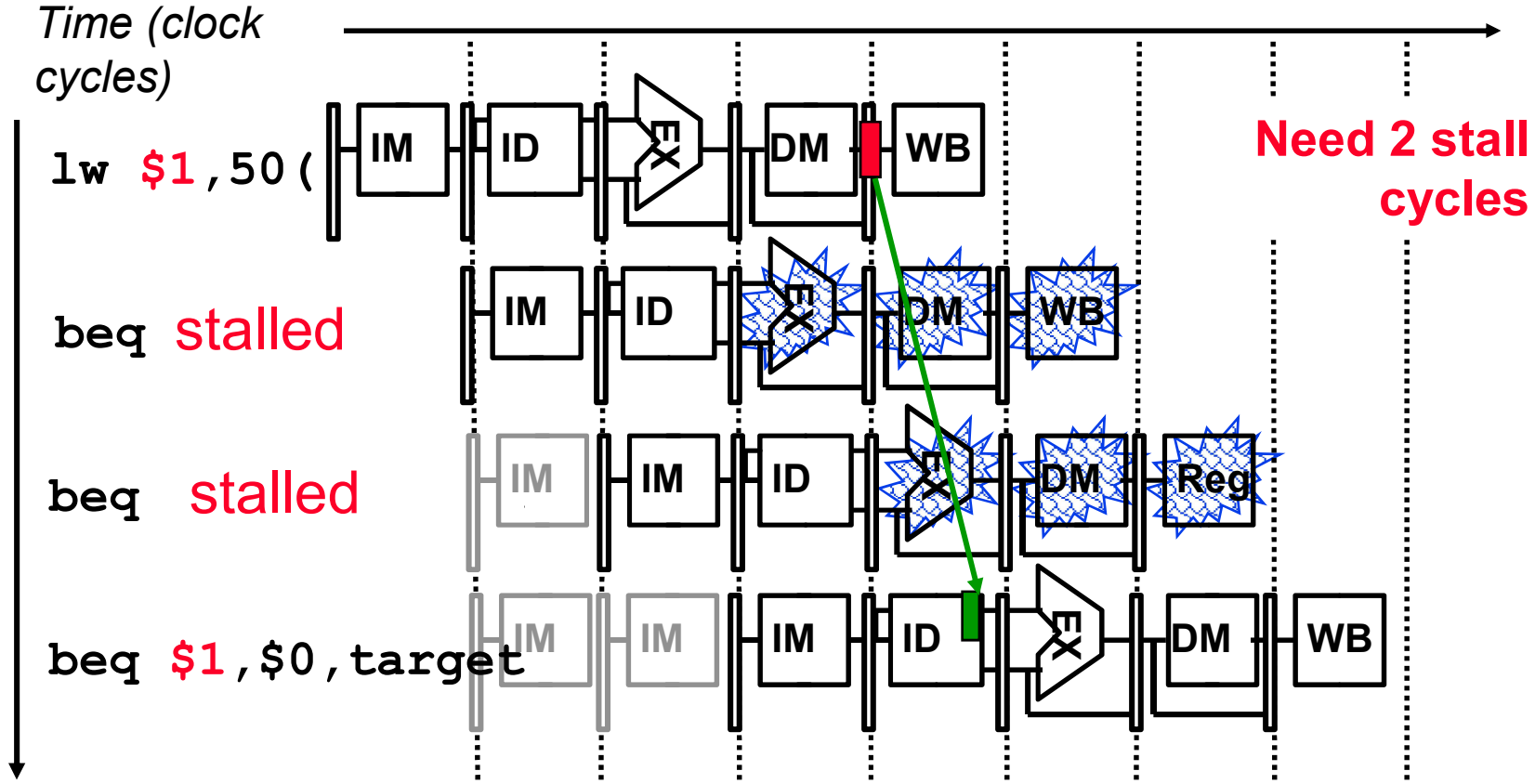
# And Another Branch Data Hazard

- What if a comparison register is a destination of the preceding ALU instruction or the 2nd preceding `lw` instr?



# And Yet Another Branch Data Hazard

- What if a comparison register is a destination of the immediately preceding preceding `lw` instr?



# Revisiting: 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

# Control (Branch, Jump) Hazards

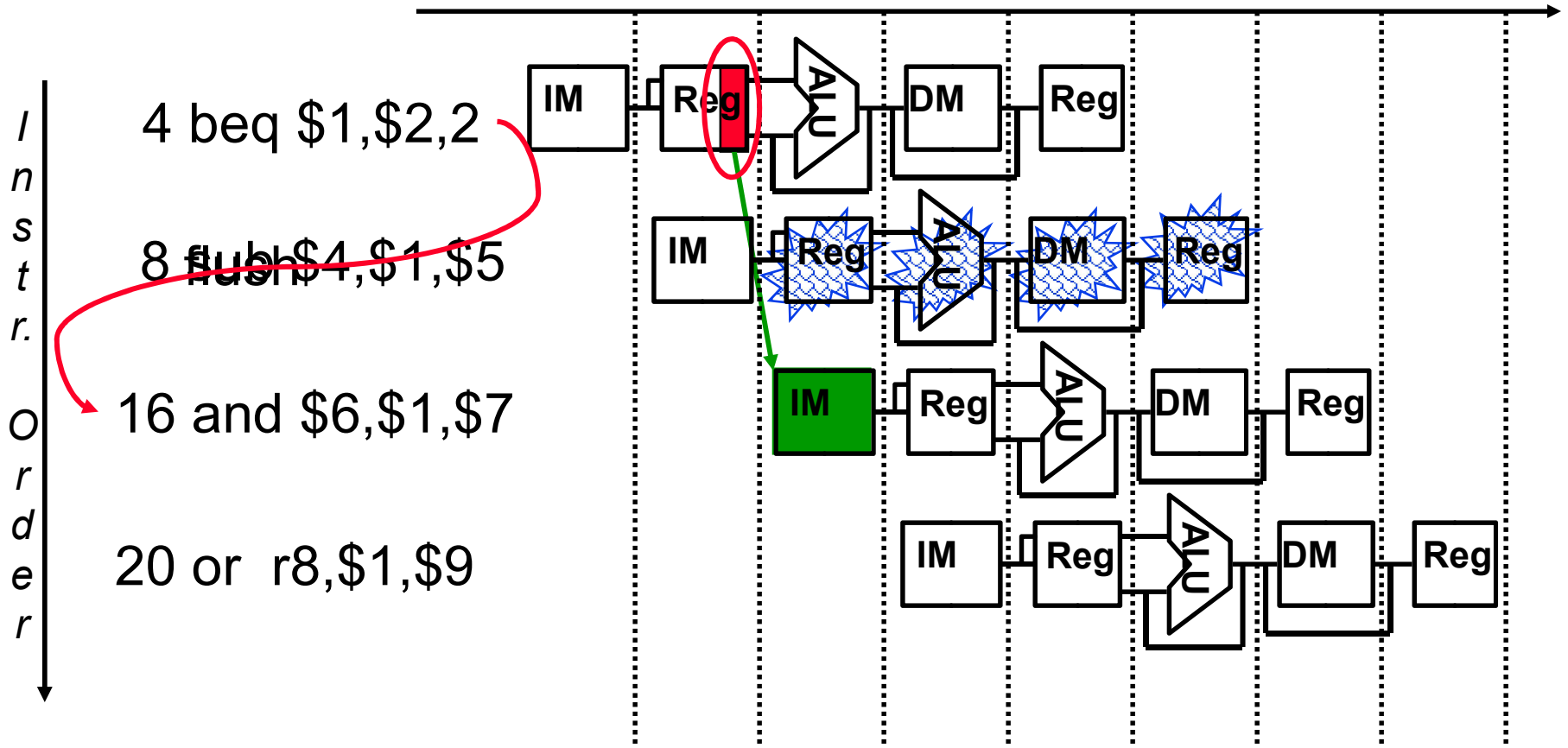
1. When do we want to know the branch decision outcome (the ALU zero flag)?
   - HINT: As early as possible in the pipeline
     - jumps always incur one stall cycle
     - For branches the outcome is computed during EX by the ALU logic, indicted by ALU's zero output which is stored in the EX/MEM pipeline register. Can we do better?
2. 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 early, and were right – this is good
     - we guessed early, and were wrong – now need to fix things
   - Evaluate prediction success rate
     - recent history is often a reasonable predictor of the near future

# 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, later in the code)
     - 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 (Always NT)




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

# Static Branching Structures

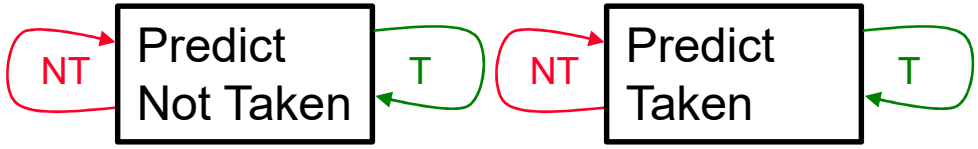
- Always predict NT works well for "top of the loop" branching structures
  - But such loops have jumps at the bottom of the loop to return to the top of the loop – and incur the jump stall overhead every time through the loop

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

- Always predict NT doesn't work well for "bottom of the loop" branching structures
  - Guess wrong every time through the loop except the last time (when we fall out of the loop)

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

# 0-Bit Predictors, Loop for 10 Iterations




| iteration | predict | actual | predict | actual |
|---|---|---|---|---|
| 1 | NT | T | T | T |
| 2 | NT | T | T | T |
| 3 | NT | T | T | T |
| 4 | NT | T | T | T |
| 5 | NT | T | T | T |
| 6 | NT | T | T | T |
| 7 | NT | T | T | T |
| 8 | NT | T | T | T |
| 9 | NT | T | T | T |
| 10 | NT | NT | T | NT |
| | 10% accuracy | | 90% accuracy | |

```
  pre-loop instr
Loop:
  1st loop instr
  2nd loop instr
        .
        .
        .
  last loop instr
  bne $1,$2,Loop
  post-loop instr
```

So use Predict Taken (T)?

# Other Branch Prediction Possibilities

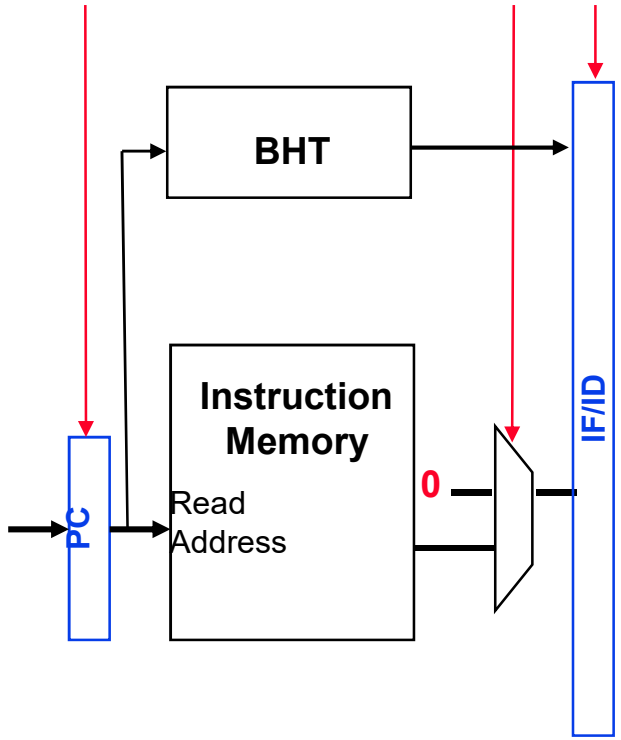
2. Predict taken (T) – predict branches will always be taken
   - BUT predict taken *always* incurs one stall cycle (*if* branch destination hardware has been moved to the ID stage)
   - Is there a way to "cache" the address of the branch target instruction, or *better yet* the actual branch target instruction itself ?? Yes … stay tuned !

- 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 Buffer

- A branch prediction buffer (aka Branch History Table (BHT)) in the IF stage addressed by the low order bits of the PC, contains bit(s) (passed to the ID stage through the IF/ID pipeline register) that tell whether the branch was taken or not the last time it was execute

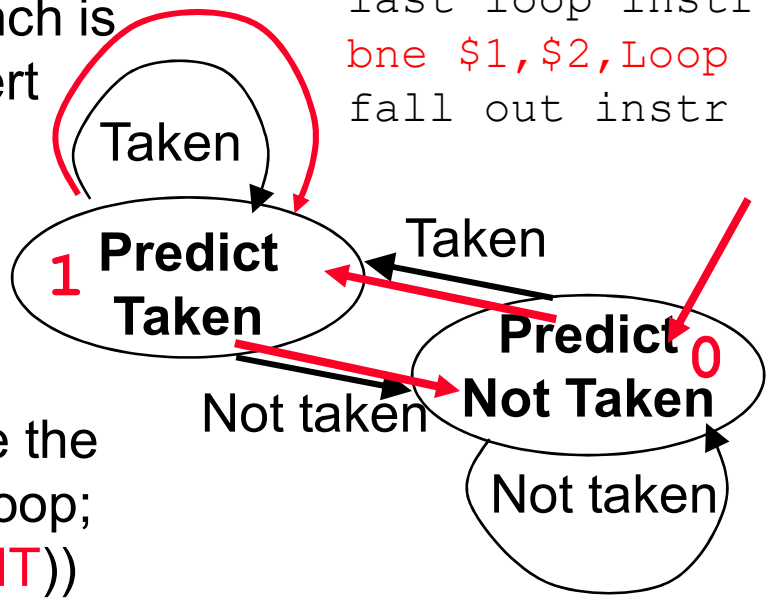
# Branch History Table

- The BHTs prediction bits 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 ID instruction is a branch and checking the prediction bit(s)
- If the prediction is wrong, flush the incorrect instruction(s) in pipeline (the one in IF), restart the pipeline with the right instruction, and invert the prediction bit(s)
- Default BHT size in SimpleScalar is 2048 entries (11 low order bits of the PC)
  - A 4096-bit BHT using 2-level adaptive prediction varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott) misprediction rate, with spice at 9% and gcc at 12%

# 1-bit NT Dynamic Branch Prediction Accuracy

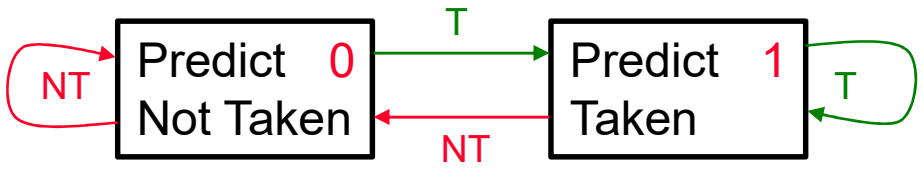
- A 1-bit NT dynamic predictor will be incorrect twice
  - Assume predict_bit = 0 (NT) to start (indicating branch not taken) and loop control is at the bottom of the loop code
  1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1 (T))
  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 (NT))

```
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

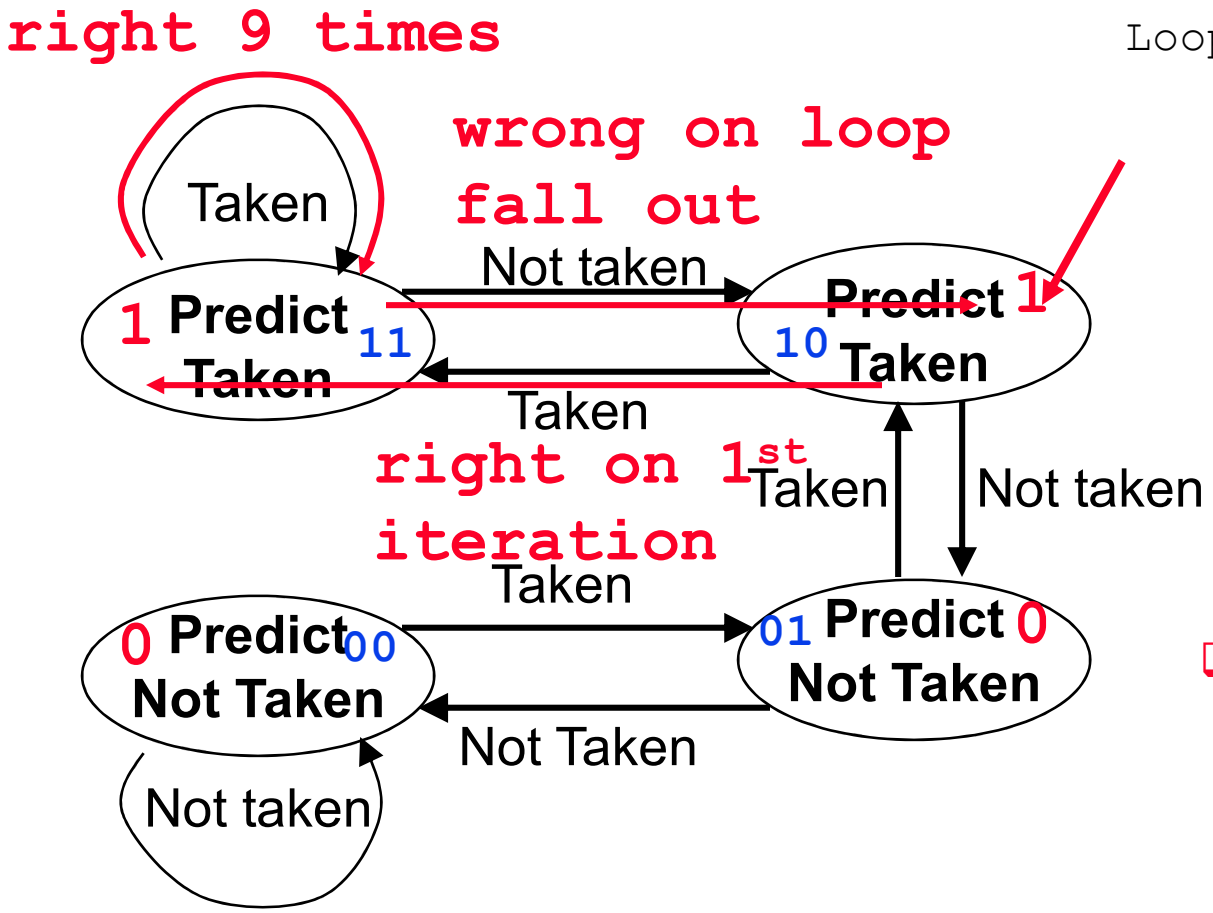
# 1-Bit Dynamic Predictor, Loop for 10 Iterations




| | Initial State = 0 | | | Initial State = 1 | | |
|---|---|---|---|---|---|---|
| iteration | state | predict | actual | state | predict | actual |
| 1 | 0 | NT | T | 1 | T | T |
| 2 | 1 | T | T | 1 | T | T |
| 3 | 1 | T | T | 1 | T | T |
| 4 | 1 | T | T | 1 | T | T |
| 5 | 1 | T | T | 1 | T | T |
| 6 | 1 | T | T | 1 | T | T |
| 7 | 1 | T | T | 1 | T | T |
| 8 | 1 | T | T | 1 | T | T |
| 9 | 1 | T | T | 1 | T | T |
| 10 | 1 | T | NT | 1 | T | NT |
| | 0 | 80% accuracy | | 0 | 90% accuracy | |

# 2-bit Dynamic Branch 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
```

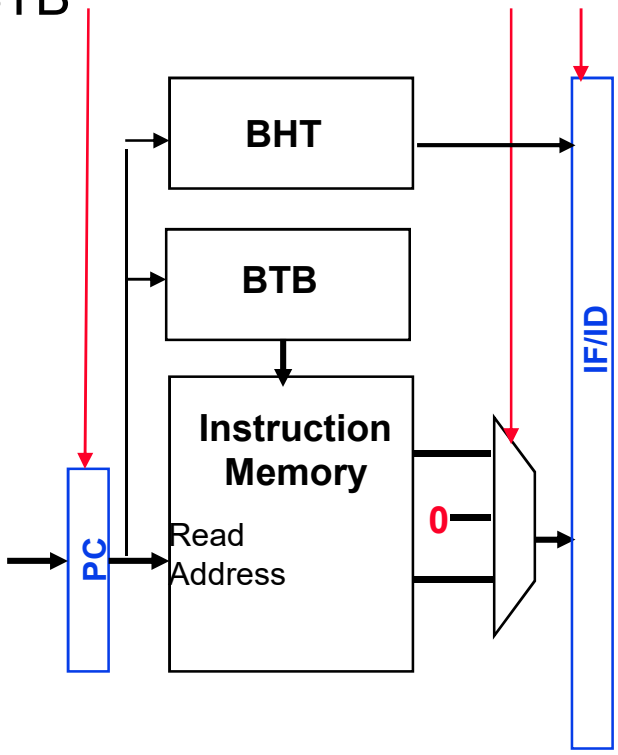
- The BHT also stores the initial FSM state

# Control (Branch, Jump) Hazards, Con't

3. What do we need to know to continue execution ?
   - Next instruction target address, maybe sequential (PC+4) *or*
     - `beq`, PC+4 + (branch instruction's sign-extended offset) which can computed during **ID** (rather than MEM) by moving the branch decision hardware to ID (comparator, zero flag, branch target address adder)
     - `j`, `jal`, constant address field read from IM during IF (26 bits) but don't know it's a jump until **ID**
     - `jr`, `jalr`, read from RF during **ID** (32 bits)
     - `trap` instruction or exception, obtained from table lookup in the OS (32 bits)
   - Want to continue execution with **no** stalls/flushes (at least for branches) on correct prediction if at all possible

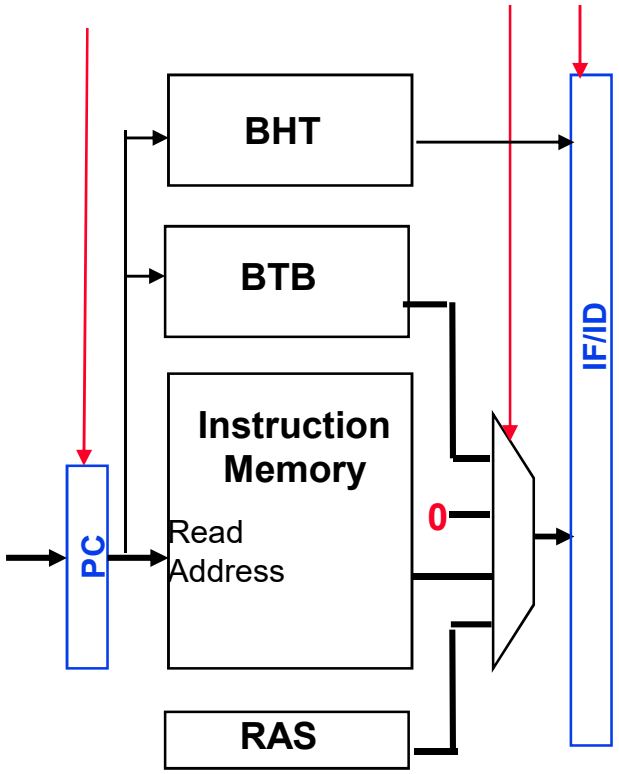
# So Must Know the Branch Target Address

- A Branch Target Buffer (BTB) in the IF stage can cache the branch target address, but remember we also need to fetch the next sequential instruction (at PC+4).
  - Would need a two read port IM to read both the PC+4 instruction and the BTB branch target instruction
  - Then a control signal from the branch in the ID stage can chose to load the PC+4 instruction or the branch target instruction into IF/ID based on the branch outcome
  - Or a no-op in the case of a `j` instruction



# An Alternative BTB

- Or the BTB can cache the actual branch target instruction while the IM is fetching the next sequential instruction
  - Now only need a one read port IM
- And add a Return Address Stack (RAS) which contains the return instruction of the jump (`jr $ra`) in the ID stage
- If we predict correctly, stalls can be avoided no matter which direction the branch goes as well as for some jump instructions



# Summary of 2-bit Dynamic Branch Predictors

- A 2-bit dynamic branch prediction scheme can give 90% accuracy since a prediction must be wrong twice before the prediction is changed
  - In a counter implementation, the counters are incremented when a branch is taken and decremented when not taken (and saturate at 00 or 11).
- BHT stores the initial state of the predictor's Finite State Machine (usually the last state last time through the loop)
- BTB stores the branch target instruction which is "fetched" along with the sequential instruction in the Fetch stage (when the branch is in the Decode stage)
- Since we read the prediction bits on every cycle, a 2-bit predictor will need both a read and a write access port for updating the prediction bits.

# SimpleScalar Branch Prediction

| -bpred<type> | |
|---|---|
| not taken | Always predict not taken |
| taken | Always predict taken |
| perfect | Perfect prediction (but can't build it) |
| bimod | Bimodal predictor using a BTB with 2-bit counters (default with 2048 entry BTB) |
| 2lev | 2-level adaptive predictor |
| comb | Combined predictor (bimodal and 2-level adaptive) |

- Many more possibilities, see http://en.wikipedia.org/wiki/Branch_predictor
- Prediction accuracy improves as the branch predictor grows in complexity

# Dealing with Exceptions

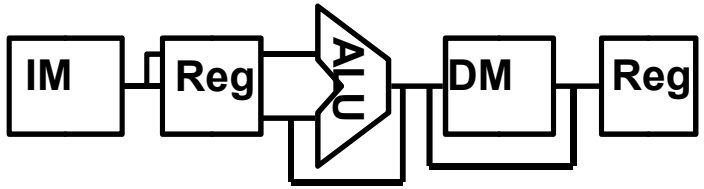
- Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
  - R-type arithmetic overflow
  - Trying to execute an undefined instruction
  - An I/O device request
  - An OS service request (e.g., a page fault, TLB exception)
  - A hardware malfunction
- The pipeline 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

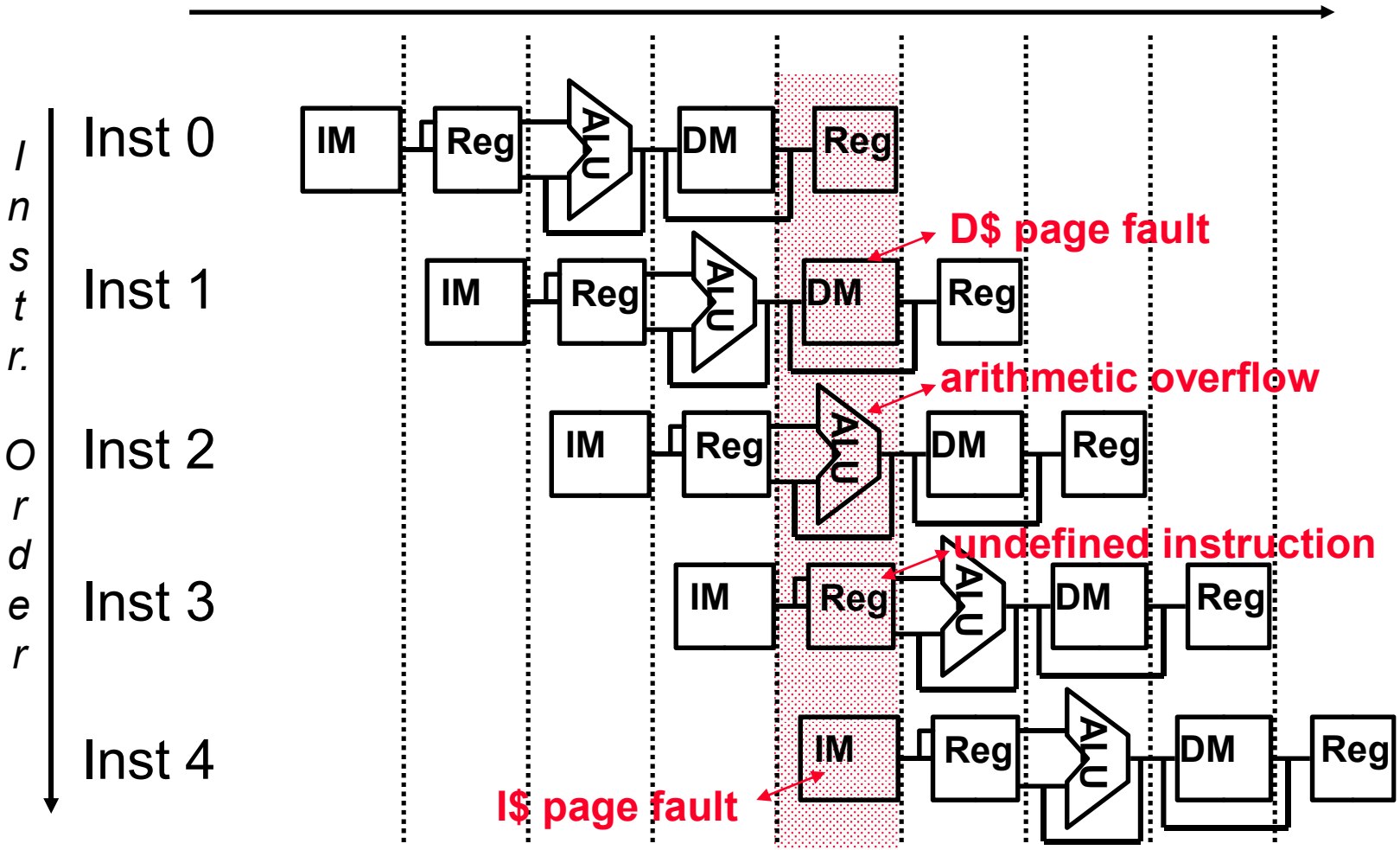
# Where in the Pipeline Exceptions Occur




| | Stage(s)? | Synchronous? |
|---|---|---|
| ❑ Arithmetic overflow | EX | yes |
| ❑ Undefined instruction | ID | yes |
| ❑ TLB or page fault | IF, MEM | yes |
| ❑ I/O service request | any | no |
| ❑ Hardware malfunction | any | no |

- ❑ Be aware that multiple exceptions can occur simultaneously in a *single* clock cycle

# Multiple Simultaneous Exceptions

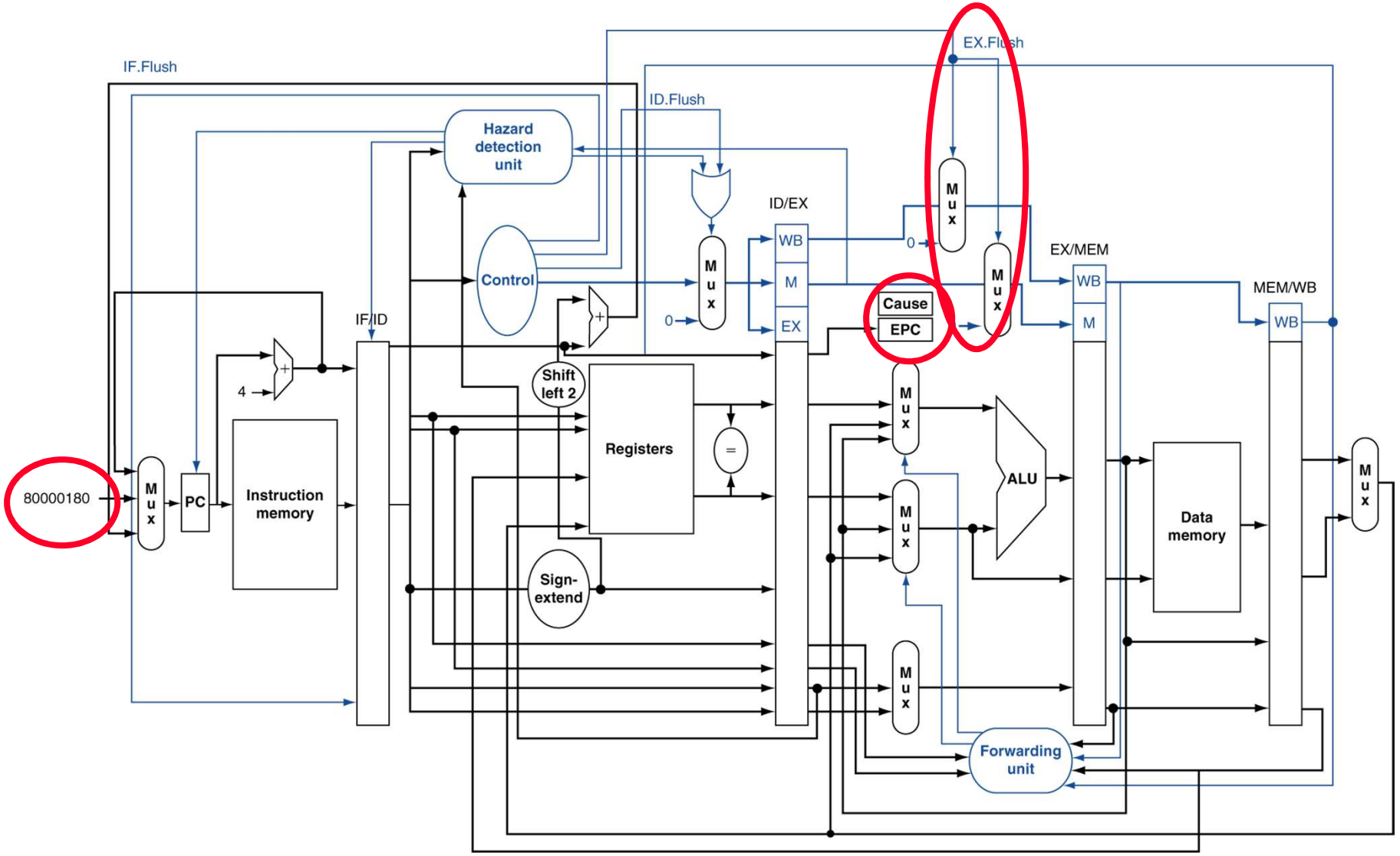



- Hardware sorts the exceptions so that the earliest instruction (D$ page fault) is the one interrupted first

# 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, 8000 $0000_{hex}$ for undefined instruction)
- A way to flush offending instruction and the ones that follow it

# Pipeline with Exception Extensions



# 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