

The Processor: Improving the Performance - Control Hazards

Computer Organization

Tuesday, 25 October 16

Many slides adapted from:
Computer Organization and Design,
Patterson & Hennessy
5th Edition, © 2014, MK
and from Prof. Mary Jane Irwin, PSU



TÉCNICO LISBOA

Summary

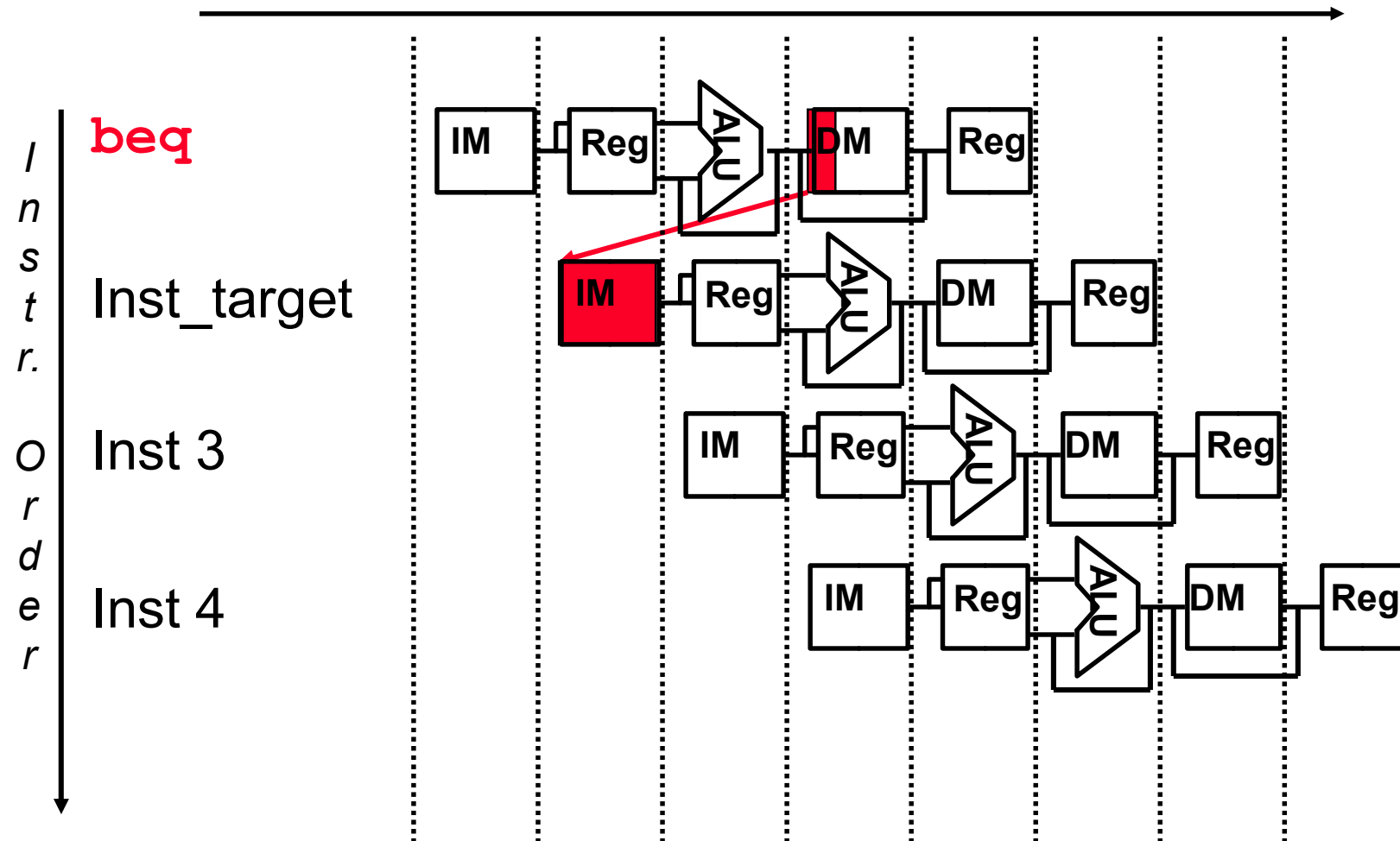
- Previous Class
 - Reducing pipeline data hazards
- Today:
 - Reducing pipeline branch hazards
 - Delayed Branch
 - Branch Prediction
 - Exceptions and Interrupts

Branch Hazards

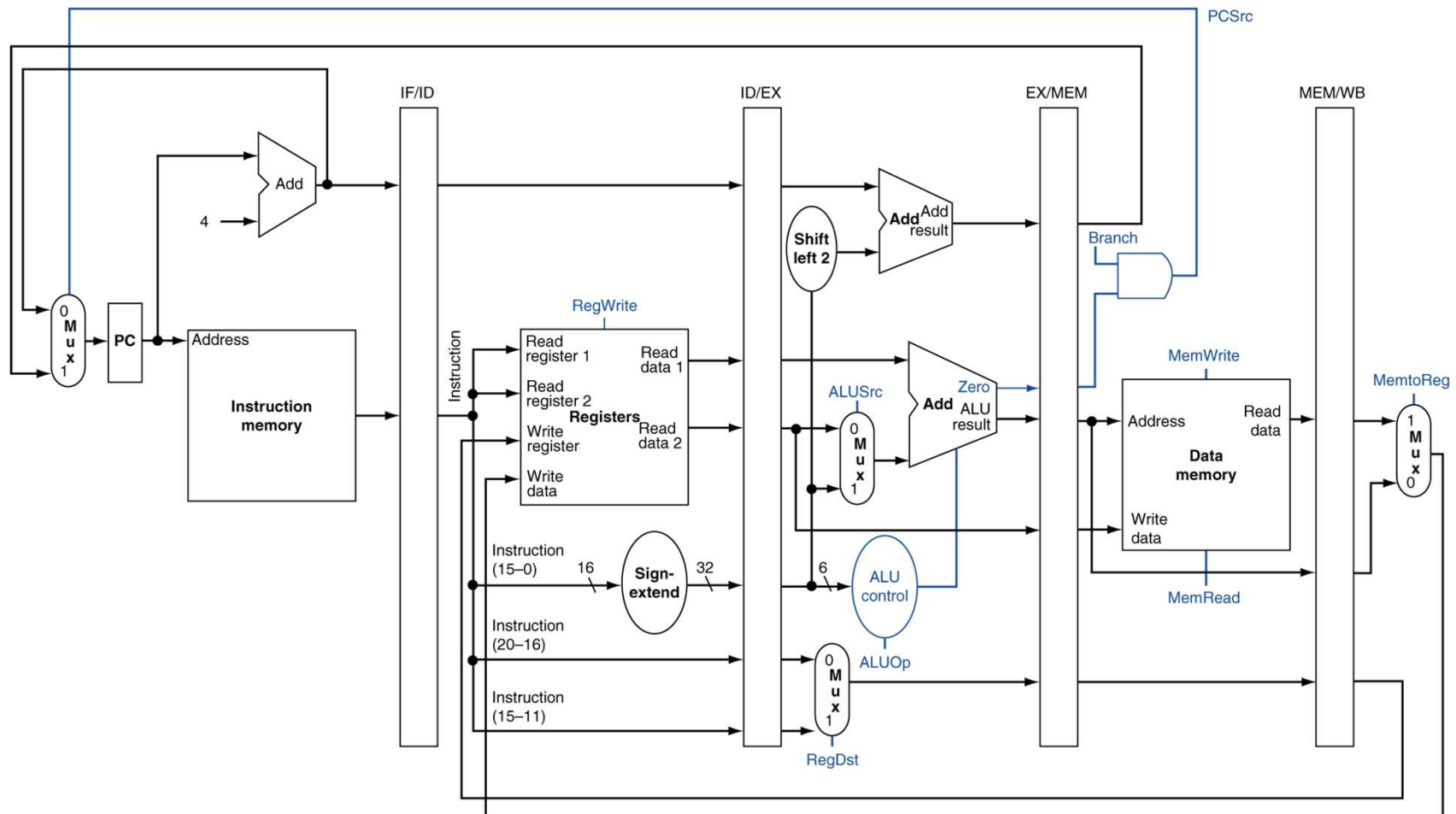
- When the flow of instruction addresses is not sequential (i.e., $PC = PC + 4$); incurred by change of flow instructions
 - Unconditional branches (`j`, `jal`, `jr`)
 - Conditional branches (`beq`, `bne`)
 - Exceptions
- Possible approaches
 - Stall (impacts CPI)
 - Move decision point as early in the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

Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards



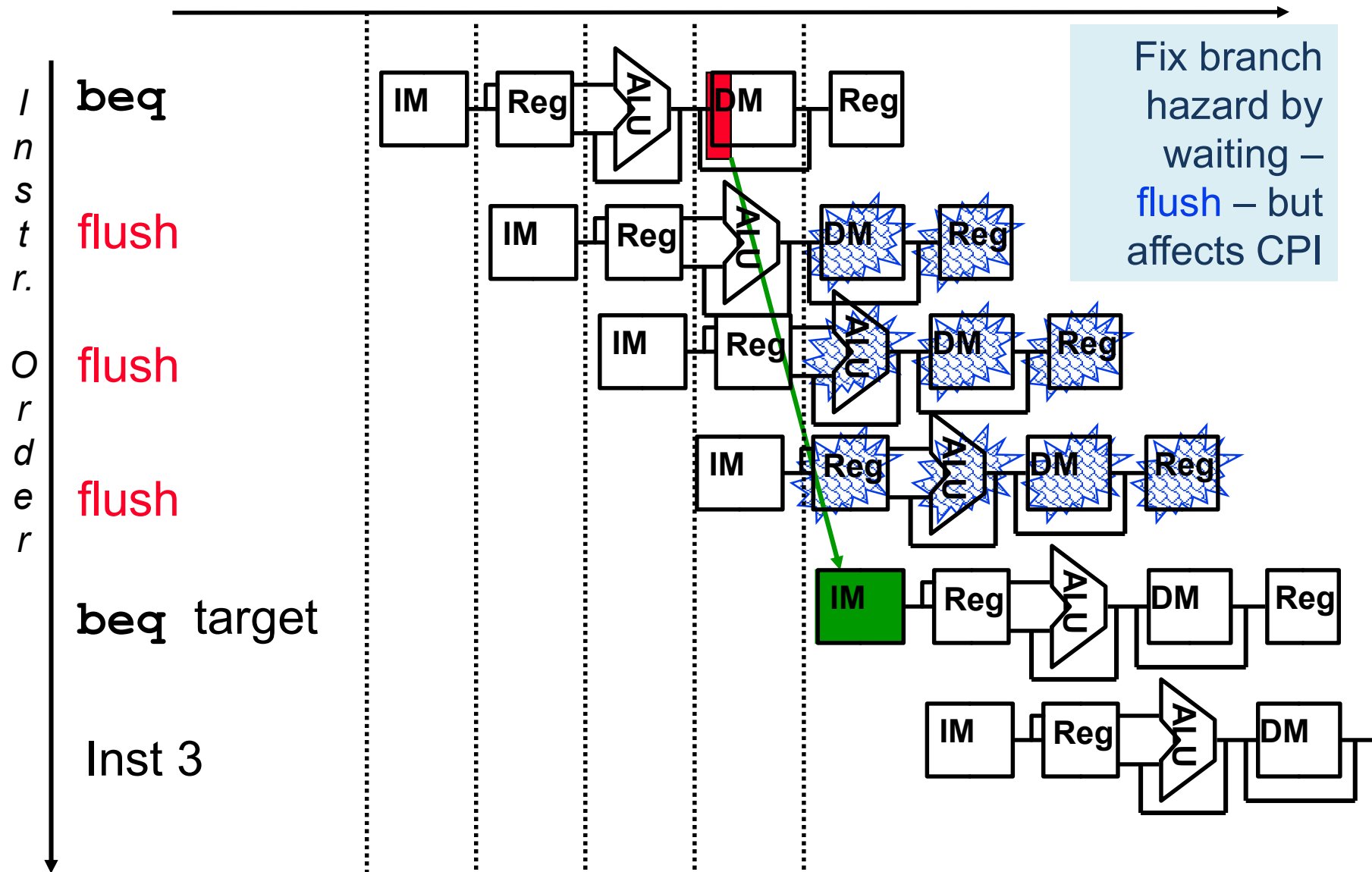
Pipelined Control (Simplified)



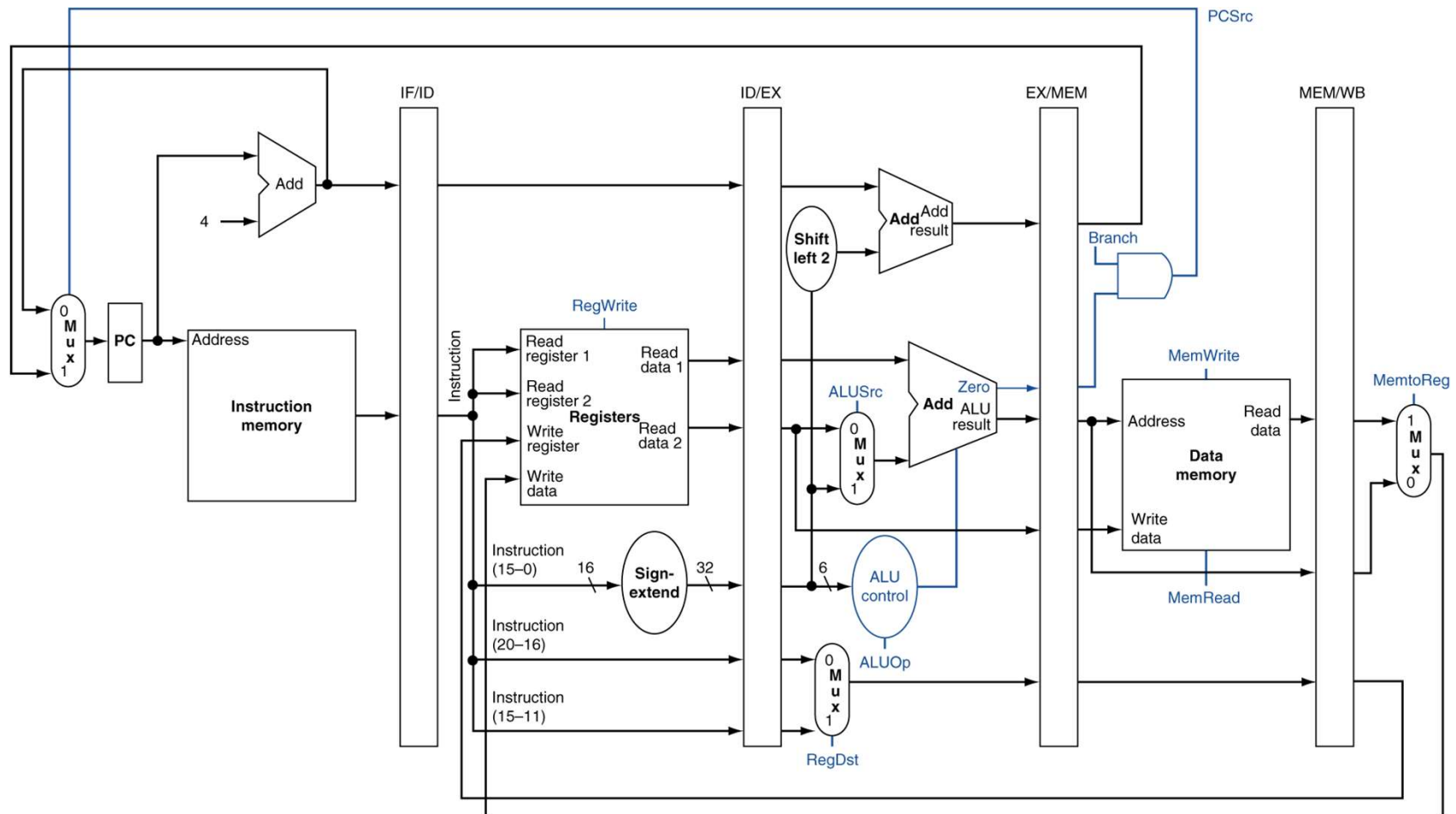
Branch Hazards

- `nop` instruction (or bubble) **inserted** between two instructions in the pipeline
 - 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 `nop` 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 `nop` instruction (as done for instructions located sequentially after `j` instructions)
 - Zero the control bits for the instruction to be flushed

One Way to “Fix” a Branch Control Hazard

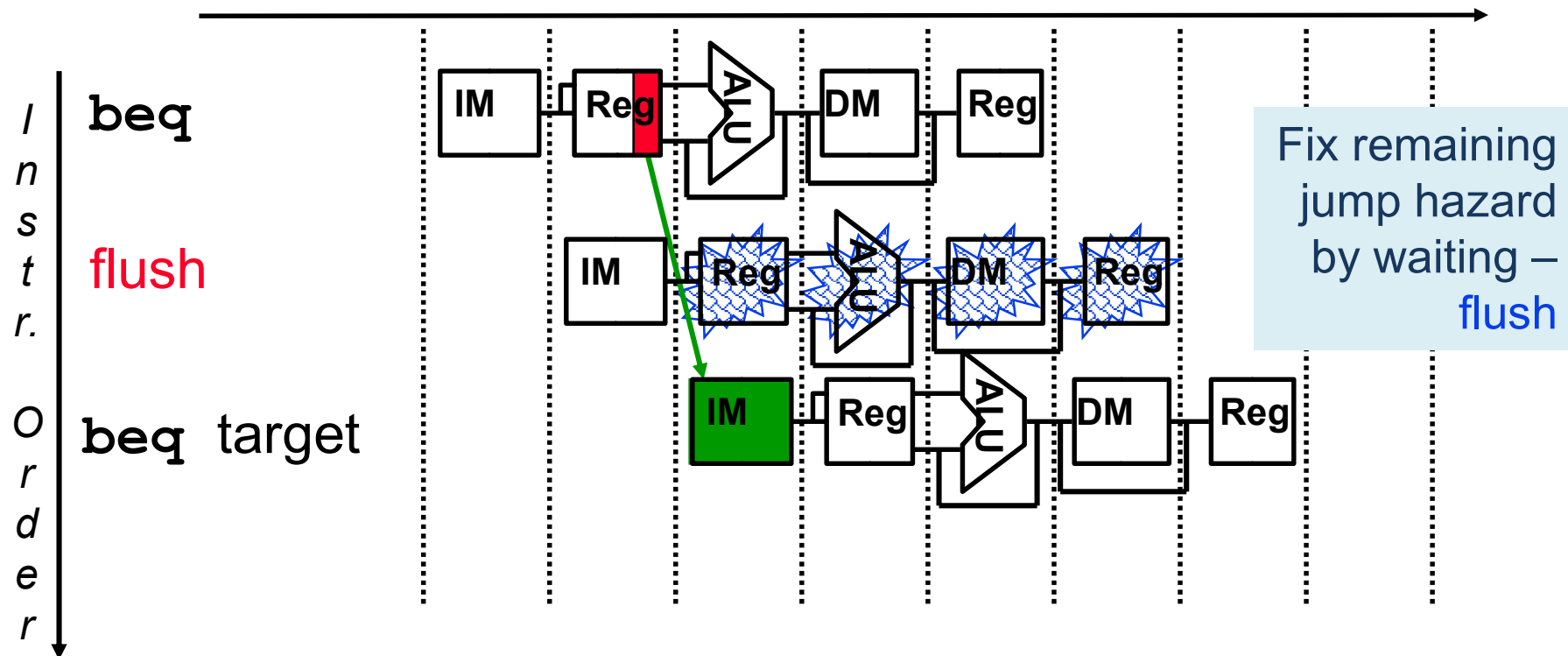


Pipelined Control (Simplified)



Branch Hazards

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

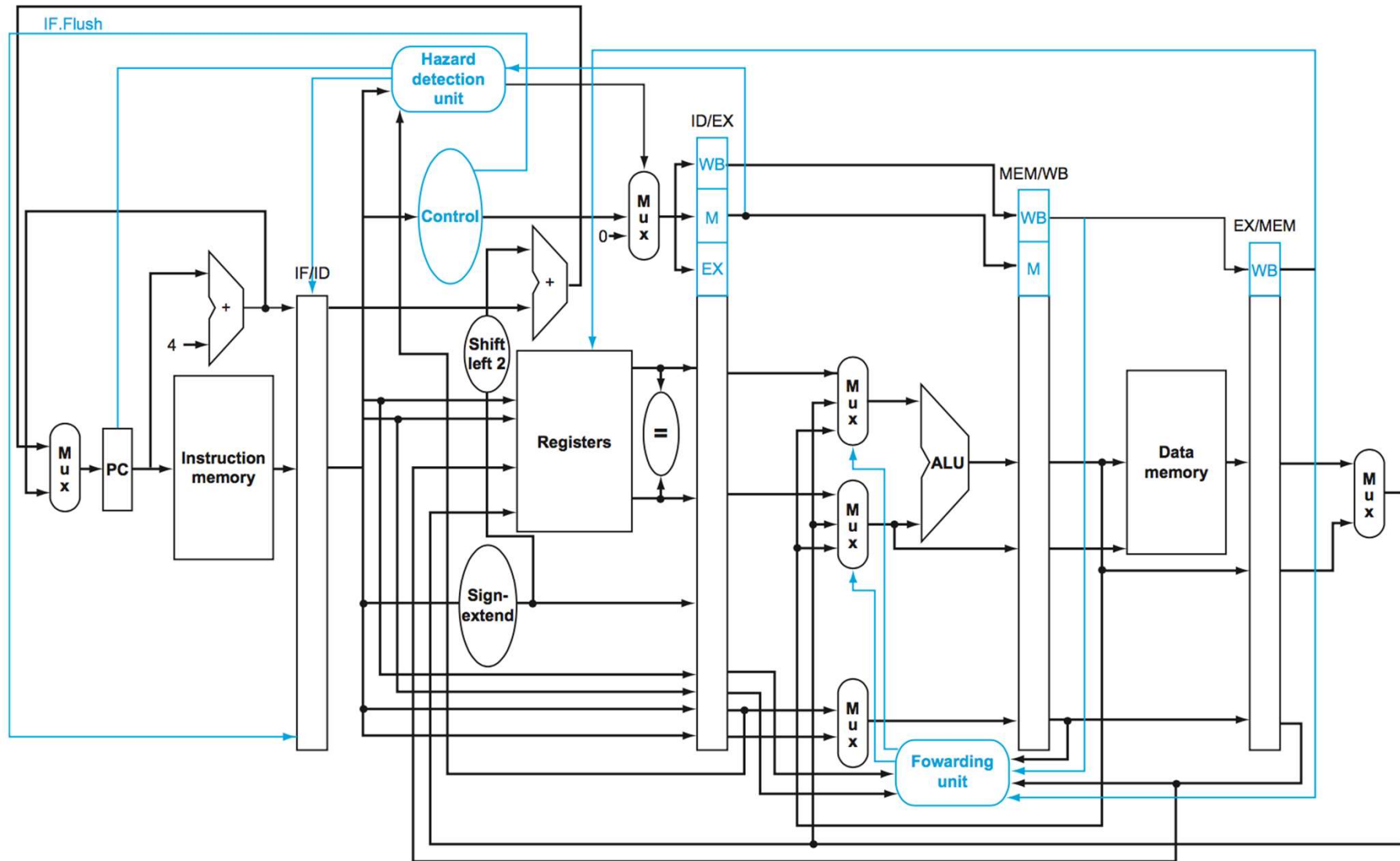


- One stall is still needed

Reducing the Delay of Branches

- Move the branch decision hardware back from the EX stage
- 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
- For deeper pipelines, branch decision points can be even *later* in the pipeline, incurring more stalls

Pipelined Control (Simplified)

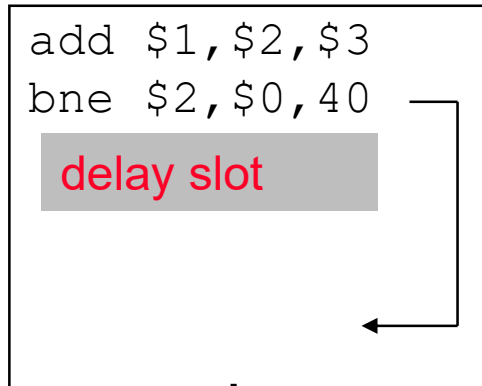


Delayed Branches

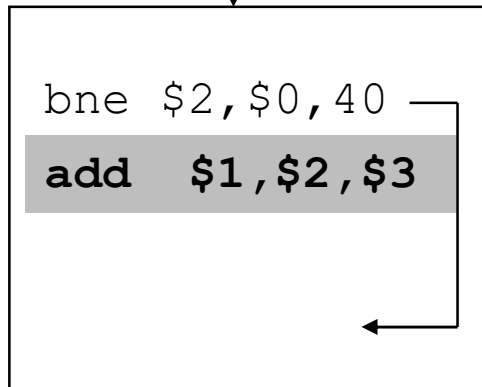
- If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with **delayed branches** which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect *after* that next instruction
 - MIPS compiler moves 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

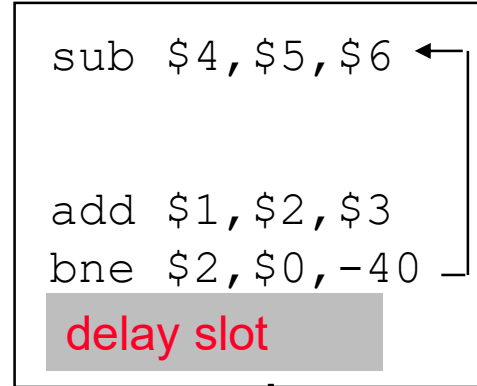
A. From before branch



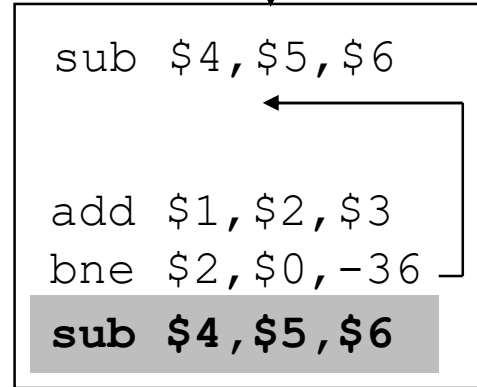
becomes



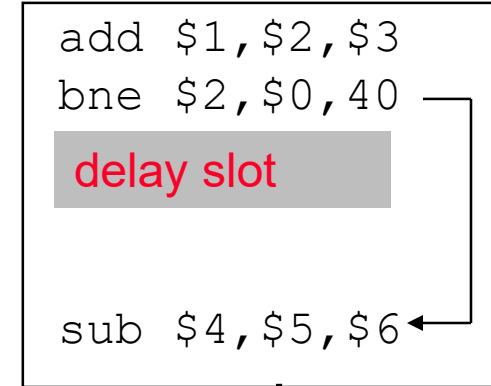
B. From branch target



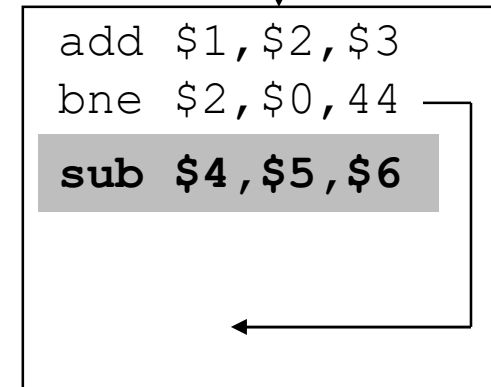
becomes



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

Branch Prediction

- Longer pipelines can't readily determine branch outcome early
 - Stall penalty becomes unacceptable
- Predict outcome of branch
 - Only stall if prediction is wrong
- In MIPS pipeline
 - Can predict branches not taken
 - Fetch instruction after branch, with no delay

Static Branch Prediction

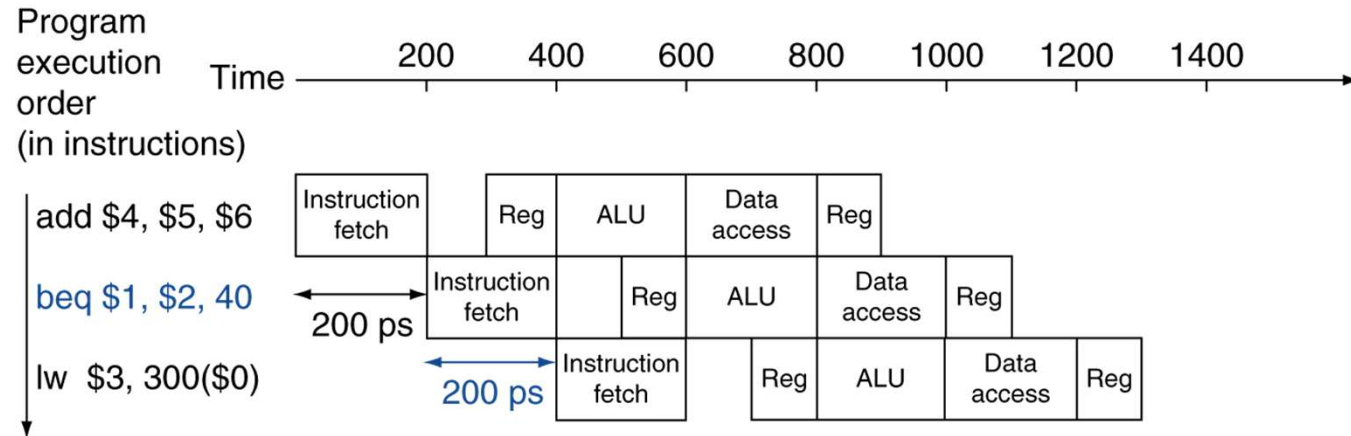
- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome

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

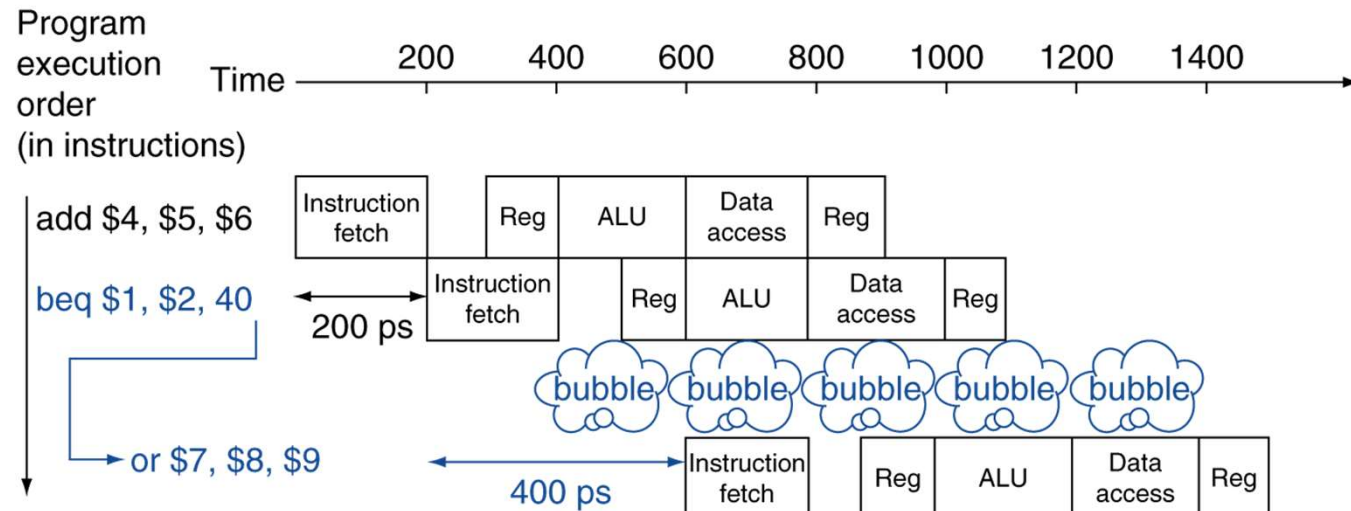
- If taken, **flush** instructions **after** the branch (earlier in the pipeline)
- ensure that those flushed instructions haven't changed the machine state
- restart the pipeline at the branch destination

MIPS with Predict Not Taken

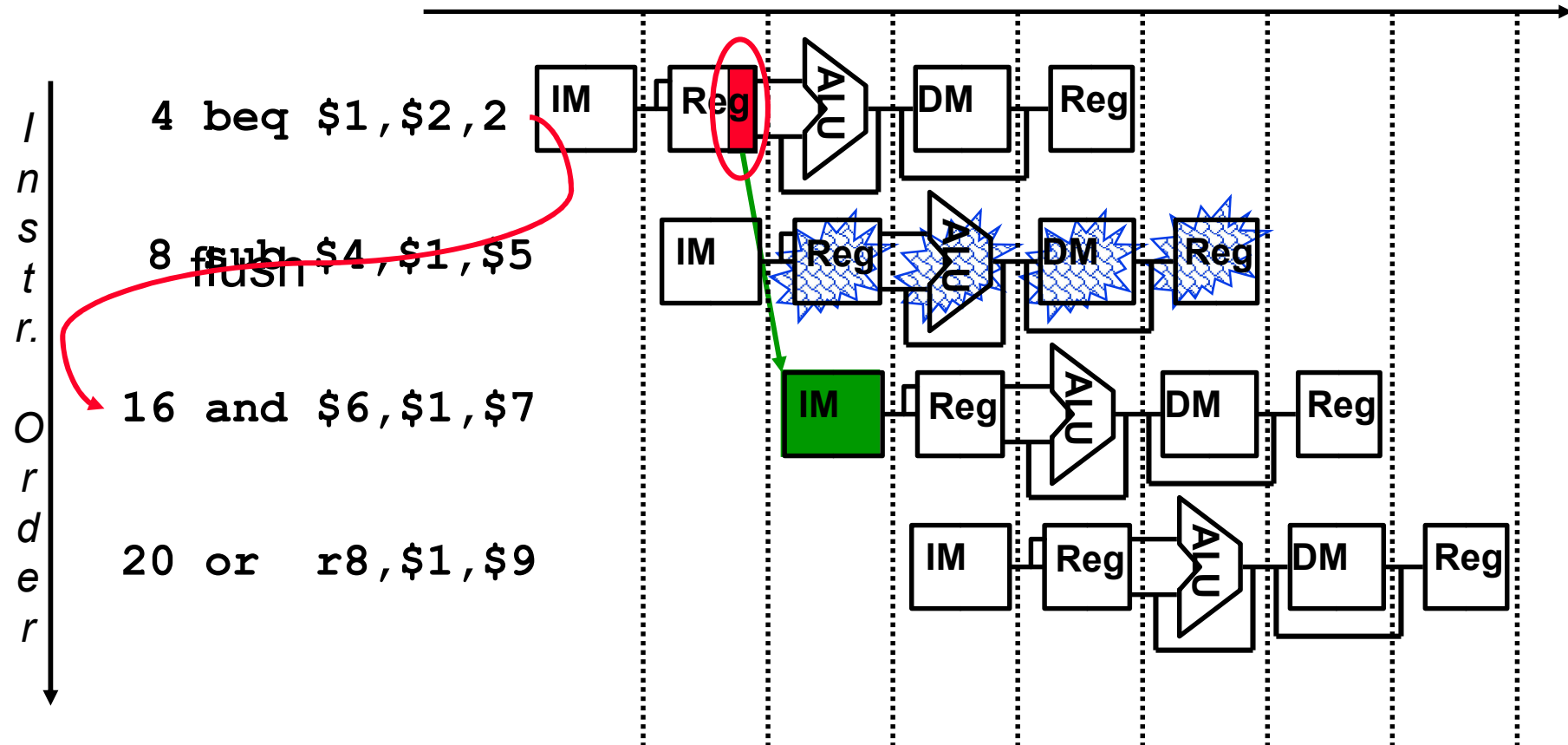
Prediction correct



Prediction incorrect



Flushing with Misprediction (Not Taken)



Static Branch Prediction

- 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

```
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
```

More-Realistic Branch Prediction

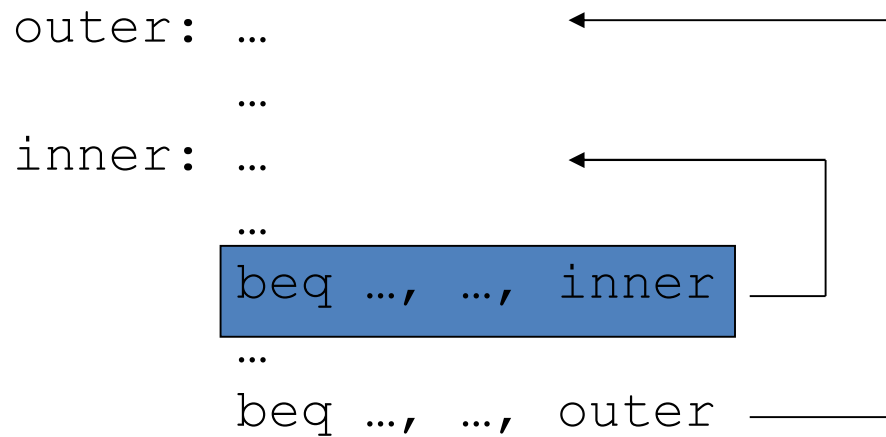
- Static branch prediction
 - Based on typical branch behavior
 - Example: loop and if-statement branches
 - Predict backward branches taken
 - Predict forward branches not taken
- Dynamic branch prediction
 - Hardware measures actual branch behavior
 - e.g., record recent history of each branch
 - Assume future behavior will continue the trend
 - When wrong, stall while re-fetching, and update history

Dynamic Branch Prediction

- Dynamic prediction:
 - Branch prediction buffer (aka branch history table)
 - Indexed by recent branch instruction addresses
 - Stores outcome (taken/not taken)
 - To execute a branch
 - Check table, expect the same outcome
 - Start fetching from fall-through or target
 - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

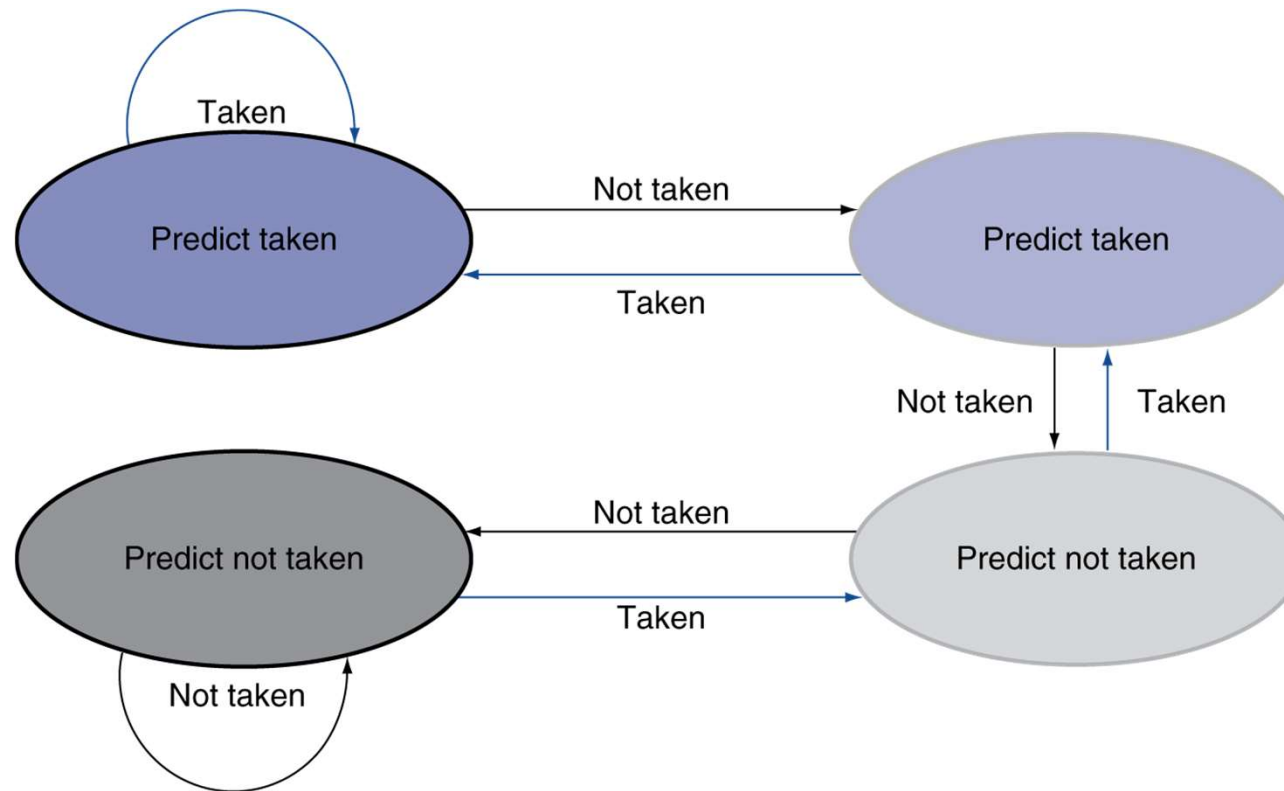
- Inner loop branches mispredicted twice!



- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions



Exceptions and Interrupts

“Unexpected” events requiring change in flow of control

- Different ISAs use the terms differently
- Exception
 - Arises within the CPU
 - e.g., undefined opcode, overflow, syscall, ...
- Interrupt
 - From an external I/O controller

Dealing with them without sacrificing performance is hard.

Handling Exceptions

In MIPS, exceptions managed by a System Control Coprocessor (CP0)

- Save PC of offending (or interrupted) instruction
 - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
 - In MIPS: Cause register
 - We'll assume 1-bit
 - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 0180

An Alternate Mechanism

- Vectored Interrupts
 - Handler address determined by the cause
- Example:
 - Undefined opcode: C000 0000
 - Overflow: C000 0020
 -: C000 0040
- Instructions either
 - Deal with the interrupt, or
 - Jump to real handler

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
 - Take corrective action
 - Use EPC to return to program
- Otherwise
 - Terminate program
 - Report error using EPC, cause, ...

Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on `add` in EX stage
 - `add $1, $2, $1`
 - Prevent `$1` from being clobbered
 - Complete previous instructions
 - Flush `add` and subsequent instructions
 - Set Cause and EPC register values
 - Transfer control to handler
- Similar to mispredicted branch
 - Use much of the same hardware

Exception Properties

- Restartable exceptions
 - Pipeline can flush the instruction
 - Handler executes, then returns to the instruction
 - Refetched and executed from scratch
- PC saved in EPC register
 - Identifies causing instruction
 - Actually $PC + 4$ is saved
 - Handler must adjust

Multiple Exceptions

- Pipelining overlaps multiple instructions
 - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
 - Flush subsequent instructions
 - “Precise” exceptions
- In complex pipelines
 - Multiple instructions issued per cycle
 - Out-of-order completion
 - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
 - Including exception cause(s)
- Let the handler work out
 - Which instruction(s) had exceptions
 - Which to complete or flush
 - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

Conclusion

- Must detect and resolve hazards
 - 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

Next Class

- Multiple issue processors
 - Instruction-Level Parallelism (ILP)

The Processor: Improving the performance - Control Hazards

Computer Organization

Tuesday, 25 October 16

Many slides adapted from:
Computer Organization and Design,
Patterson & Hennessy
5th Edition, © 2014, MK
and from Prof. Mary Jane Irwin, PSU



TÉCNICO LISBOA

Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding accordingly
 - **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 and superscalar pipelines), a simple static prediction scheme will hurt performance.
- **Dynamic branch prediction** – predict branches at run-time using run-time information

Branch History Table

- 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 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
 - 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)

Calculating the Branch Target

- Even with predictor, still need to calculate the target address
 - 1-cycle penalty for a taken branch
- Branch target buffer
 - Cache of target addresses
 - Indexed by PC when instruction fetched
 - If hit and instruction is branch predicted taken, can fetch target immediately

