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, js)
  - 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 not decoded until ID, so one flush is needed
  - Jumps incur one stall
  - Fix jump hazard by waiting – stall – but affects CPI

- 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) 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
  - The instructions later in the pipeline (earlier in the code) progress normally down the pipeline

- Flushes (or instruction squashing) were an instruction in the pipeline is replaced with a **noop** instruction (as done for instructions located sequentially after a branch)
  - Zero the control bits for the instruction to be flushed

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)
  - 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

```plaintext
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

- "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

- add $1,$2,$3
- if $2=0 then
  - delay slot

becomes

- add $1,$2,$3
- if $2=0 then
  - sub $4,$5,$6

- $4, $5, $6

B. From branch target

- add $1,$2,$3
- if $1=0 then
  - delay slot

becomes

- add $1,$2,$3
- if $1=0 then
  - sub $4,$5,$6

- $4, $5, $6

C. From fall through

- add $1,$2,$3
- if $1=0 then
  - delay slot

becomes

- add $1,$2,$3
- if $1=0 then
  - sub $4,$5,$6

- $4, $5, $6

- $4, $5, $6

- $4, $5, $6

Flush with Misprediction (Not Taken)

- 4 beq $1,$2,2
- 8 sub $4,$1,$5

- 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).

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

Flush with Misprediction (Not Taken)

- 4 beq $1,$2,2
- 8 sub $4,$1,$5

- 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
- Predict not taken doesn’t work well for "bottom of the loop" branching structures

**Static Branch Prediction, con’t**
- Resolve branch hazards by assuming a given outcome and proceeding
  - 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
  - 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 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
  - 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
    - 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)
    - As long as branch is taken (looping), prediction is correct
    - 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

**2-bit Predictors**
- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed
2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed.

<table>
<thead>
<tr>
<th>Branch Predictors</th>
<th>Taken</th>
<th>Not Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predict Taken</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Predict Not Taken</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- BHT also stores the initial FSM state.

Prediction Accuracy of 2-bit Prediction Schemes

- SPEC99 benchmarks using a 4096-entry 2-bit prediction buffer [Pan, So, and Rameh (1992)].

Correlating Branch Predictors

- 2-bit predictor uses only the recent behavior of a single branch to predict its future behavior.

- Is branch direction affected by more "global" properties?

- Behavior of b3 is correlated with that of b1 and b2
  - if both b1 and b2 are NT, b3 will be T
  - Can (how do) we predict such branches?

A (1,1) Correlating Predictor

- Behavior of a 1-bit predictor for repeated executions of above with values of d=2, 0, 2, 0, ...

- 1-bit predictor that uses 1 bit of correlation
  - X? X if last branch was NT, Y if last branch was T

(m,n) Correlating Predictors

- Use behavior of the last m branches to choose from among 2^m n-bit predictors (for a single branch)
  - Makes the predictor context-sensitive

- Yields improved prediction accuracy for small hardware cost
  - History of last m branches can be kept as a shift register
  - Each bit records whether corresponding branch was NT/NTT
  - Branch prediction buffer can then be indexed by concatenating the lower-order bits of address with the m-bit history

Prediction Accuracy of Correlating Predictors

- Is this a fair comparison?
  - Metric: Number of bits
  - 4096 entries (2-bit): 8K
  - 1024 entries (2,2): 22 × 2 × 1024 = 8K
Tournament Predictors

- Adaptively combine local and global predictors
  - Make more effective use of large numbers of prediction bits
- Generalization: Multilevel branch predictors
  - Several levels of branch prediction tables
    - Combination of local (with different bit/branch) and global (different m,n)
  - Selector algorithm that chooses which predictor to apply
- A selector should be adaptive (“learn” from its mistakes)
- Saturating counters as in local predictors
  - Increment/decrement based on which predictor is correct (when the other is incorrect)

Performance of Tournament Predictors

- Misprediction rate on SPEC99 as total number of bits is increased
  - “optimal” correlating predictor chosen at each point

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

- Arithmetic overflow
- Undefined instruction
- TLB or page fault
- I/O service request
- Hardware malfunction

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.

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

Additions to MIPS to Handle Exceptions (Fig 6.42)

- Cause register (records exceptions) – hardware to record the cause of the exceptions in Cause register 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.

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).

Next Lecture and Reminders

- Next lecture:
  - Introduction to Multiple Issue.
- Reminders:
  - HW4 due today February 21st.
  - Lab 1 due in two weeks March 8th.