(Review) Pipeline Hazards

- Limit to pipelining: Hazards
  - Prevent next instruction from executing during its designated clock cycle
- Three classes of hazards
  - **Structural**: Hardware cannot support this combination of instructions - two instructions need the same resource.
  - **Data**: Instruction depends on result of prior instruction still in the pipeline
  - **Control**: Pipelining of branches & other instructions that change the PC
- Common solution is to **stall** the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline
  - To do this, hardware or software must detect that a hazard has occurred

\[
\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
\]
(Review) Pipeline Hazards (B): Data Hazards

Three generic types of data hazards

- **Read After Write (RAW)**
  - Instr\(_J\) tries to read operand before Instr\(_I\) (I < J) writes it
  - Called a dependence

- **Write After Read (WAR)**
  - Instr\(_J\) writes operand before Instr\(_I\) reads it
  - Called an anti-dependence
    - Results from reuse of names

- **Write After Write (WAW)**
  - Instr\(_J\) writes operand before Instr\(_I\) writes it
  - Called an output dependence
    - Also results from name reuse

(Review) Data Hazards and Pipeline Stalls

- Not all data hazards result in a stall

  - For the simple five-stage RISC pipeline
    - Only RAW hazards result in a pipeline stall
      - Instruction reading a register needs to wait until it is written
    - WAR and WAW hazards cannot occur because
      - All instructions take 5 stages
      - Reads happen in the 2\(^{nd}\) stage (ID)
      - Writes happen in the 5\(^{th}\) stage (WB)
      - No way for a write from a subsequent instruction to interfere with the read (or write) of a prior instruction

  - For more complicated pipelines (later in the course)
    - Both WAR and WAW hazards are possible if instructions execute out of order or access (read) data later in the pipeline

RAW Hazards in the 5-stage Pipeline

Absence of WAR and WAW Hazards
Reducing Impact of RAW Hazards: Data Forwarding

- **Data forwarding** (also called **bypassing** or **short-circuiting**)
  - Directly transfers data from each stage to earlier pipeline stages
  - Result is accessible before it gets written into the register file.

  Instr i: \texttt{add r1, r2, r3}  \quad \text{(result ready after EX stage)}

  Instr j: \texttt{sub r4, r1, r5}  \quad \text{(result needed in EX stage)}

- To support data forwarding, additional hardware is required.
  - Multiplexers to allow data to be transferred back
  - Control logic for the multiplexers

Hardware Changes for Forwarding

![Diagram showing hardware changes for data forwarding]

Avoidance of RAW Hazards Using Forwarding

![Diagram illustrating avoidance of RAW hazards using forwarding]

Forwarding Does Not Eliminate All Hazards

![Diagram illustrating forwarding does not eliminate all hazards]

Cope with this by stalling the EXE stage till results are available.
Pipeline Hazards (C): Control Hazards

- Control hazards occur due to instructions changing the PC
  - can result in a large performance loss
- A branch is either
  - Taken: PC $\leftarrow$ PC + 4 + Imm
  - Not Taken: PC $\leftarrow$ PC + 4
- Cannot fetch the next instruction till value of PC is known
- Simplest solution is to stall the pipeline upon detecting a branch
  - ID stage detects the branch
  - Don’t know if the branch is taken until the EX stage
  - If the branch is taken, we need to repeat the IF and ID stages
  - New PC is not changed until the end of the MEM stage, after determining if the branch is taken and the new PC value

(Review) Pipelined Implementation of a RISC ISA

3 Cycle Stall on Branch-Induced Control Hazards

Time (clock cycles)

Instr. order

beq r1,r3,36
and r2,r3,r5
or r6,r1,r7
add r8,r1,r9
xor r10,r1,r11

Impact of Branch Stalls

- If CPI = 1, 30% branches
  - Stall 3 cycles $\Rightarrow$ new CPI = 1.9!
  - 50% of these branches taken $\Rightarrow$ new CPI = 1 + 0.45 + 0.3 = 1.7
- Penalty would be worse for current-day (longer) pipelines
  - IF and ID-like stages are each multiple-cycle
- How do we reduce impact of branch stalls?
  - Two part solution:
    - Determine branch taken or not sooner, AND
    - Compute taken branch address earlier
Pipelined Implementation of a RISC ISA: Reducing Branch Penalty to 1 cycle

Branch Behavior in Programs
- Based on SPEC benchmarks on DLX (CA-AQA, 2nd Edition)
  - Branches occur with a frequency of 14% to 16% in integer programs and 3% to 12% in floating point programs.
  - About 75% of the branches are forward branches
  - 60% of forward branches are taken
  - 80% of backward branches are taken
  - 67% of all branches are taken
- Why are branches (especially backward branches) more likely to be taken than not taken?

Dealing with Branch Stalls
- Approach 1: Stall until branch direction is clear
- Approach 2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - “Squash” instructions in pipeline if branch actually taken
    - Can do this because CPU state not updated till late in the pipeline
  - 33% DLX branches not taken on average
  - PC+4 already calculated, so use it to get next instruction

Dealing with Branch Stalls (cont’d)
- Approach 3: Predict Branch Taken
  - 67% DLX branches taken on average
  - But haven’t yet calculated target address in a 5-stage RISC pipeline
    - So, will still incur a 1-cycle latency
    - Makes sense on machines where branch target is known before outcome
      - (later in the lecture: Branch Target Buffers)
- Approach 4: Delayed Branch
  - Define branch to take place AFTER n following instructions
    - branch instruction
    - sequential successor
    - sequential successor
    - branch target if taken

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Clock Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>i (T)</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>i+1</td>
<td>IF idle idle idle idle</td>
</tr>
<tr>
<td>T</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>T+1</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>T+2</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
Branch Delay Slots

- Instructions in the branch delay slot(s) get executed whether or not branch is taken

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Clock Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>i (T)</td>
<td>1</td>
</tr>
<tr>
<td>D(i+1)</td>
<td>IF</td>
</tr>
<tr>
<td>T</td>
<td>IF</td>
</tr>
<tr>
<td>T+1</td>
<td>IF</td>
</tr>
<tr>
<td>T+2</td>
<td>IF</td>
</tr>
</tbody>
</table>

- Heavily used in early RISC machines
  - 1 delay-slot suffices for a 5-stage pipeline (target available at end of ID)
  - Machines with deep pipelines require additional delay slots to avoid branch penalties
    - Benefits are unclear

Scheduling the Branch Delay Slot

Where does the instruction for the delay slot come from?

- Nullifying or cancelling branches
  - Converts delay slot instruction into a nop

Evaluating Branch Alternatives

Pipeline speedup = \[ \frac{\text{Pipeline depth}}{T + \text{Branch frequency} \times \text{Branch penalty}} \]

- Assumptions
  - 14% of instructions are branches
  - 30% of branches are not taken
  - 50% of delay slots can be filled with useful instructions

<table>
<thead>
<tr>
<th>Scheduling scheme</th>
<th>Branch penalty</th>
<th>CPI</th>
<th>speedup v. unpipelined</th>
<th>speedup v. stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow stall pipeline</td>
<td>3</td>
<td>1.42</td>
<td>3.5</td>
<td>1.0</td>
</tr>
<tr>
<td>Fast stall pipeline</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
<td>1.26</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
<td>1.26</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>0.7</td>
<td>1.10</td>
<td>4.5</td>
<td>1.29</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.07</td>
<td>4.7</td>
<td>1.34</td>
</tr>
</tbody>
</table>

- A Compiler can reorder instructions to further improve speedup

Importance of Avoiding Branch Stalls

- Crucial in modern microprocessors, which issue/execute multiple instructions every cycle
  - Need to have a steady stream of instructions to keep the hardware busy
  - Stalls due to control hazards dominate

- So far, we have looked at static schemes for reducing branch penalties
  - Same scheme applies to every branch instruction

- Potential for increased benefits from dynamic schemes
  - Can choose most appropriate scheme separately for each instruction
    - Branches to top of loop have different behavior (Taken) than “if (x == 0) return;” (Not Taken)
    - Can “learn” appropriate scheme based on observed behavior
  - Dynamic (hardware) branch prediction schemes
    - For both direction (T or NT), target prediction
    - Key element of all modern microprocessors
Dynamic Branch Prediction (1): Branch Prediction (History) Buffer

- Small memory indexed by the low-order bits of the branch instruction
  - Stores a single bit of information: T or NT
  - Starts off as T, flips whenever a branch behaves opposite to prediction
  - Maintained by the IF stage
  - So, a correct prediction implies no branch penalty

- Problems with this simple scheme
  - Prediction value may not correspond to branch being considered
    - Cannot avoid this: Branch Prediction Buffer serves as a cache without tags
  - Does not do a good job of predicting “mostly-taken branches”
    - E.g., a loop: for (i=0; i<10; i++) { ... }
    - Repeated executions of the loop will result in 2 mispredictions
      - Last iteration flips T to NT
      - First iteration flips NT to T
  - So, prediction accuracy of 80%

- Can we do better?

Dynamic Branch Prediction (2): 2-bit Prediction Schemes

- Store 2 bits of information in branch history buffer

  - How does this do on our loop example?
    - 1 misprediction per iteration if we start off in the (11) state
    - 1 misprediction per iteration (plus 2 mispredictions the first time) if we start in (00) state

- Generalization: n-bit saturating counters
  - Increment if taken, decrement if not
  - Predict T if value more than half, else NT
  - Not too much of a win over 2-bit counters

Prediction Accuracy of 2-bit Prediction Schemes

- SPEC89 benchmarks using a 4096-entry 2-bit prediction buffer (Pan, So, and Rameh [1992])

- Is hit-rate in the cache an issue?

Dynamic Branch Prediction (3): 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?

  if (aa == 2)           DSUBUI R3, R1, #2
   aa = 0;              BNEZ R3, L1 ; branch b1
  if (bb == 2)           DADD R1, R0, R0
   bb = 0;              L1: DSUBUI R3, R2, #2
  if (aa != bb)          BNEZ R3, L2 ; branch b2
                      { ... }  
  L2: DSUBU R3, R1, R2
       BEQZ R3, L3    ; branch b3

- 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

if (d == 0)
    BNEZ R1, L1 ; branch b1
if (d == 1)
    DADDUI R1, R0, #1
    L1: DADDUI R3, R1, #1
    { ... }
    BNEZ R1, L1 ; branch b1

L2:

• 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
  - \( X/Y \): X if last branch was NT, Y if last branch was T

<table>
<thead>
<tr>
<th>b1</th>
<th>b1</th>
<th>b2</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>T</td>
</tr>
</tbody>
</table>

These predictions would be correct, irrespective of value of d

(m,n) Correlating Predictors

• Use behavior of the last m branches to choose from among \(2^m\) n-bit predictors (for a single branch)
• 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 T/NT
  - 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

- nasa7: 1%
- matrix300: 1%
- toycomp: 1%
- doduc: 1%
- spice: 9%
- SPECCS benchmarks: 9%
- got: 9%
- espresso: 9%
- esprot: 9%
- ll: 11%

- 4096 entries: 2 bits per entry
- Unlimited entries: 2 bits per entry
- 128 entries: (3,3)

9/25/2002