• Computer Organization and Design, Patterson and Hennessy. Feel free to skim or ignore the hardware details. For our purposes, we’re interested in the interaction between the instruction stream and the processor pipeline stages.
  » Section 4.6, pp. 356-357 (graphical pipeline representations)
  » Section 4.7 Data Hazards: Forwarding vs Stalling
  » Section 4.8 Control Hazards
Control Hazards

- Branch instructions cause control hazards (aka branch hazards) because we don’t know which instruction to fetch next

```assembly
bne $s0, $s1, skip
add $s4, $s3, $s0
...
skip:
sub $s4, $s3, $s0
```
Idea: Stall for branch hazard

- Stall until we know which instruction to execute next
  » would introduce a 4-cycle pipeline bubble in the basic pipeline

\[
\begin{array}{l}
\text{bne } $s0, $s1, \text{next} \\
\text{sub } $s4, $s3, $s0 \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\hline
& & & \text{\textit{stall}} & \\
\hline
\end{array}
\]
Idea: Move Branch Logic to ID

• Move the branch hardware to ID stage
  » Hardware to compare two registers is simpler than hardware to add them
• We still have to stall for one cycle
• And we can’t move the branch up any more

```assembly
bne $s0, $s1, next
sub $s4, $s3, $s0
```

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
```
Idea: Reorder Instructions

• Reordering instructions is a common technique for avoiding pipeline stalls

• Static reordering
  » programmer, compiler and assembler do this

• Dynamic reordering
  » modern processors can see several instructions
  » they execute any that have no dependency
  » this is known as *out-of-order execution* and is complicated to implement but effective
Branch Delay Slot

- A branch now causes a stall of one cycle
- Try to execute an instruction instead of nop
- The compiler (assembler, programmer) must find an instruction to fill the branch delay slot
  » 50% of the instructions are useful
  » 50% are \texttt{nop}s which don’t do anything
Branch Delay Slot execution

- Instruction in the branch delay slot always executes, no matter what the branch does
  » it follows the branch in memory
  » but it “piggybacks” and is always executed
  » no bubble at all

 actual instruction sequence after reordering by assembler
beq with delay slot

```
.set    nomacro

.beq     $v0,$zero,$L4

.move    $s1,$s4

.set    macro

.set    reorder
```


jal with delay slot

move $a0,$s3
move $a1,$s0
.set noreorder
.set nomacro
jal QuickSort
move $a2,$s4
.set macro
.set reorder
Idea: Predict the branch action

• For example, assume the branch is not taken
  » Execute the next instruction in memory
• If we guessed right, we’re golden
  » no bubble at all
• If we guessed wrong, then we lose a little
  » squash the partially completed instructions.
  » This is called flushing the pipeline
  » Wasted time, but would have stalled anyway
Squash

• Must be able to completely suppress the effects of guessing wrong
  » An instruction cannot write to memory or a register until we’re sure it should execute
Assume Branch Not Taken

**Branch not taken**

bne  
addi 
addi 
Done: move

**Branch taken**

bne  
addi  
move
Static Branch Prediction

• Most backwards branch are taken (80%)
  » they are part of loops
• Half of forward branches are taken (50%)
  » if statements
• Common static branch prediction scheme is
  » predict backwards branches are taken
  » predict forward branches are not taken
• This does okay (70-80%), but not great
Dynamic Branch Prediction

• Most programs are pretty regular
  » Most of the time only execute a small subset of the program code
  » Same branch instructions execute repeatedly

• A particular branch instruction is usually:
  » taken if it was taken last time
  » not taken if it was not taken last time

• If we keep a history of each branch instruction, then we can predict much better
Dynamic Branch Prediction

• The CPU records what happened last time we executed the branch at this address
• Generally record last two results
  » simple 4-state transition table makes prediction
• Dynamic branch prediction is 92-98% accurate
2-bit prediction scheme
Implementing Branch Prediction

- There is not room to store every branch instruction address
  - so last few bits of the instruction address are used to index into a table
  - some instructions collide like a hash table
  - but that’s okay, it just means we’re wrong once in a while
### Branch Prediction Table

<table>
<thead>
<tr>
<th>Address</th>
<th>state?</th>
<th>Predict</th>
<th>correct?</th>
<th>new state</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x00401234</td>
<td>11</td>
<td>not taken</td>
<td>yes</td>
<td>11</td>
</tr>
<tr>
<td>0x004F0238</td>
<td>00</td>
<td>taken</td>
<td>no</td>
<td>01</td>
</tr>
<tr>
<td>0x0040223C</td>
<td>10</td>
<td>not taken</td>
<td>no</td>
<td>00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Importance of Branch Prediction

• Branches occur very frequently
  » every five instructions on average
• Modern processors execute up to 4 instructions per cycle
  » so a branch occurs every 2 cycles
• Newer pipelines are getting longer
  » 8,9,11,13 cycles
  » error penalty is 3-5 cycles instead of 1 cycle
  » hard to fill branch delay slots
• But even newer pipelines are actually getting shorter! (Intel Atom, multicores, …)
  » It’s all design tradeoffs"

---