Readings and References

• Reading
  – Sections 6.4 through 6.6, Patterson and Hennessy, Computer Organization & Design

• Other References
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
```
Stall for branch hazard

- We could stall to see which instruction to execute next
  - would introduce a 4-cycle pipeline bubble
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

\[
\begin{align*}
\text{bne } & \text{ } s0, \text{ } s1, \text{ } \text{next} \\
\text{sub } & \text{ } s4, \text{ } s3, \text{ } s0
\end{align*}
\]
Reordering 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
Branch Delay Slot

- A branch now causes a stall of one cycle
- Try to execute an instruction instead of stall
- 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

```
bne $s0, $s1, next
add $s3,$s3,1
sub $s4, $s3, $s0
```
beq with delay slot

.set noreorder
.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
```
Assume we will not branch

- 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

\[
\begin{align*}
\text{bne } & \, \$s0,\$zero,\text{Done} \\
\text{addi } & \, \$t0,\$t0,1 \\
\text{addi } & \, \$t0,\$t0,3 \\
\text{Done: move } & \, \$t1,\$t0 \\
\end{align*}
\]

Branch taken

\[
\begin{align*}
\text{bne } & \, \text{IF} \\
\text{addi } & \, \text{ID} \\
\text{addi } & \, \text{EX} \\
\text{move} & \, \text{MEM} \\
\end{align*}
\]
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

- predict: taken
  - 00
    - taken
  - 01
    - not taken
- predict: taken
  - 01
    - taken
  - 11
    - not taken
- predict: not taken
  - 10
    - taken
  - 11
    - not taken
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 the 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