More pipeline facts

Where to spend your $

The steady state throughput is determined by the slowest stage of the pipeline. There is no point in speeding up the ALU if the memory units are slow.

Performance Enhancement

\[
\text{Speedup} = \frac{\text{Execution time on non-pipelined CPU}}{\text{Execution time on pipelined CPU}}
\]
Control Hazard

If the condition of `beq` is true, then two wrong instructions will enter the pipe, and need to be flushed out.
Solution 1: Insert bubble

lw r1, 32(r2)

L1: beq r1, r3, L2

addi r1, r1, 1
addi r4, r4, -1

L2:

Branch penalty = 2 cycles

But how will the control unit know that the conditional branch will be taken? Use prediction. Since most forward branches are not taken, a common prediction by the control unit is that the branch will not be taken.
**Solution 2: Predict not taken**

**Case 1. Branch not taken**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw r1, 32(r2)</td>
<td>F D X M W</td>
<td></td>
</tr>
<tr>
<td>L1: beq r1, r3, L2</td>
<td>F D X M W</td>
<td></td>
</tr>
<tr>
<td>addi r1, r1, 1</td>
<td>F D X M W</td>
<td></td>
</tr>
<tr>
<td>addi r4, r4, -1</td>
<td>F D X M</td>
<td></td>
</tr>
<tr>
<td>L2:</td>
<td>F D X</td>
<td></td>
</tr>
</tbody>
</table>

**Case 2. Branch taken**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw r1, 32(r2)</td>
<td>F D X M W</td>
<td></td>
</tr>
<tr>
<td>L1: beq r1, r3, L2</td>
<td>F D X M W</td>
<td></td>
</tr>
<tr>
<td>addi r1, r1, 1</td>
<td>F D o o o</td>
<td></td>
</tr>
<tr>
<td>addi r4, r4, -1</td>
<td>F o o o</td>
<td></td>
</tr>
<tr>
<td>L2:</td>
<td>F D X</td>
<td></td>
</tr>
</tbody>
</table>

*2 wrong instructions*
**Flushing the pipe**

The wrong instructions need to be flushed out from the pipeline. One mechanism is to disable the writes (by generating proper control signals), so that the wrong instructions are reduced to NOP.

![Diagram of flushing the pipe]

1. *beq condition is true*
2. *turn off writes*
3. *Branch target*
**Speedup**

Estimate the slowdown now (assume that 15% instructions are conditional branch).

$$\text{Speedup} = \frac{5}{1 + \text{branch frequency} \times \text{branch penalty}}$$

**Reducing Branch Penalty**

Early detection of *branch condition* and computation of the *branch target address* help reduce branch penalty. One can modify the datapath to detect the branch condition in the D-stage and reduce the branch penalty to one cycle.
Compiler Scheduling of Branch Delay Slots

(Assume that branch penalty = 1 cycle)

<table>
<thead>
<tr>
<th>Original program</th>
<th>Transformed version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 1</td>
<td>Instruction 1</td>
</tr>
<tr>
<td>Branch instruction</td>
<td>Branch Instruction</td>
</tr>
<tr>
<td>Instruction 3</td>
<td>Branch delay slot [NOP]</td>
</tr>
<tr>
<td>Instruction 4</td>
<td>Instruction 3</td>
</tr>
<tr>
<td></td>
<td>Instruction 4</td>
</tr>
</tbody>
</table>

Scheduling a NOP after a conditional branch is equivalent to “fetching and flushing an instruction” by the control unit.

An alternative is to schedule some meaningful instruction in the branch delay slot.
Instruction reordering by the compiler

Example 1

\[ r7 := r2 + r3 \]
\[ \text{if (r2=0) then go to L} \]

branch delay slot

\[ \text{instruction} \]

\[ \ldots \ldots \]

\[ L: \text{ do something} \]

\[ \text{if (r2=0) then go to L} \]

\[ r7 := r2 + r3 \]

\[ \text{instruction} \]

\[ \ldots \ldots \]

\[ L: \text{ do something} \]

NOTE: Always improves performance if such an unrelated instruction is available.
Example 2 (Backward branch)

L: r4:= r5 - r6
r1:= r1 - r2
if r1=0 then goto L
branch delay slot

L1: r1:=r1 - r2
if r1=0 then goto L
r4:= r5 - r6
instruction

Performance improves when the branch is taken.
However, it must be OK to execute r4:= r5-r6
when the branch is not taken (or else the
instruction has to be canceled)
**Example 3 (Forward branch)**

\[
\begin{align*}
\text{r1:= } & r2 - r3 & \text{r1:= } & r2 - r3 \\
\text{if } r1=0 \text{ then goto } L & & \text{if } r1=0 \text{ then goto } L \\
\text{branch delay slot} & & \text{r4:= } & r5 - r6 \\
\text{r4:= } & r5 - r6 & \text{L: instruction} & \text{L: instruction}
\end{align*}
\]

Performance improves when the branch is **NOT** taken. However, it must be OK to execute \( \text{r4:= r5-r6} \) when the branch is taken (or else the instruction has to be canceled)
**Brach prediction**

*Static prediction vs. dynamic prediction*

*Static prediction* is done during *compile time*. It is a rule of thumb that *forward branches* are usually *not taken*, and *backward branches* are usually *taken*.

*Dynamic branches* are predicted by profiling the *run-time* behavior of programs.

Ideally we want to have a “crystal ball”, so that we can see ahead of time whether a branch will be *taken* or not.