Branch Hazards

or

“Which way did he go?”

Control Dependence

• Just as an instruction will be dependent on other instructions to provide its operands (data dependence), it will also be dependent on other instructions to determine whether it gets executed or not (branch dependence or control dependence).
• Control dependences are particularly critical with conditional branches.

```plaintext
add $5, $3, $2
sub $6, $5, $2
beq $6, $7, somewhere
and $9, $6, $1
...
somewhere: or $10, $5, $2
add $12, $11, $9
```
Branch Hazards

- Branch dependences can result in branch hazards (when they are too close to be handled correctly in the pipeline).
Dealing With Branch Hazards

- **Hardware**
  - stall until you know which direction
  - reduce hazard through earlier computation of branch direction
  - guess which direction
    - assume not taken (easiest)
    - more educated guess based on history (requires that you know it is a branch before it is even decoded!)

- **Hardware/Software**
  - nops, or instructions that get executed either way (delayed branch).

Stalling for Branch Hazards

```
beq $4, $0, there
and $12, $2, $5
or ...
add ...
sw ...
```
Stalling for Branch Hazards

- Seems wasteful, particularly when the branch isn’t taken.
- Makes all branches cost 4 cycles.

Assume Branch \textit{Not Taken}

- works pretty well when you’re right

\begin{center}
\begin{tikzpicture}
\node (cc1) [cc] {CC1};
\node (cc2) [cc] at (cc1.east) {CC2};
\node (cc3) [cc] at (cc2.east) {CC3};
\node (cc4) [cc] at (cc3.east) {CC4};
\node (cc5) [cc] at (cc4.east) {CC5};
\node (cc6) [cc] at (cc5.east) {CC6};
\node (cc7) [cc] at (cc6.east) {CC7};
\node (cc8) [cc] at (cc7.east) {CC8};

\node (im) [im] at (cc1.north west) {IM};
\node (reg1) [reg] at (im.east) {Reg};
\node (dm) [dm] at (reg1.east) {DM};
\node (reg2) [reg] at (dm.east) {Reg};
\node (im2) [im] at (cc2.north west) {IM};
\node (reg3) [reg] at (im2.east) {Reg};
\node (dm2) [dm] at (reg3.east) {DM};
\node (reg4) [reg] at (dm2.east) {Reg};
\node (im3) [im] at (cc3.north west) {IM};
\node (reg5) [reg] at (im3.east) {Reg};
\node (dm3) [dm] at (reg5.east) {DM};
\node (reg6) [reg] at (dm3.east) {Reg};
\node (im4) [im] at (cc4.north west) {IM};
\node (reg7) [reg] at (im4.east) {Reg};
\node (dm4) [dm] at (reg7.east) {DM};
\node (reg8) [reg] at (dm4.east) {Reg};
\node (im5) [im] at (cc5.north west) {IM};
\node (reg9) [reg] at (im5.east) {Reg};
\node (dm5) [dm] at (reg9.east) {DM};
\node (reg10) [reg] at (dm5.east) {Reg};
\node (im6) [im] at (cc6.north west) {IM};
\node (reg11) [reg] at (im6.east) {Reg};
\node (dm6) [dm] at (reg11.east) {DM};
\node (reg12) [reg] at (dm6.east) {Reg};
\node (im7) [im] at (cc7.north west) {IM};
\node (reg13) [reg] at (im7.east) {Reg};
\node (dm7) [dm] at (reg13.east) {DM};
\node (reg14) [reg] at (dm7.east) {Reg};
\node (im8) [im] at (cc8.north west) {IM};
\node (reg15) [reg] at (im8.east) {Reg};
\node (dm8) [dm] at (reg15.east) {DM};
\node (reg16) [reg] at (dm8.east) {Reg};

\draw (im) edge (reg1)
      (reg1) edge (dm)
      (dm) edge (reg2)
      (reg2) edge (im2)
      (im2) edge (reg3)
      (reg3) edge (dm2)
      (dm2) edge (reg4)
      (reg4) edge (im3)
      (im3) edge (reg5)
      (reg5) edge (dm3)
      (dm3) edge (reg6)
      (reg6) edge (im4)
      (im4) edge (reg7)
      (reg7) edge (dm4)
      (dm4) edge (reg8)
      (reg8) edge (im5)
      (im5) edge (reg9)
      (reg9) edge (dm5)
      (dm5) edge (reg10)
      (reg10) edge (im6)
      (im6) edge (reg11)
      (reg11) edge (dm6)
      (dm6) edge (reg12)
      (reg12) edge (im7)
      (im7) edge (reg13)
      (reg13) edge (dm7)
      (dm7) edge (reg14)
      (reg14) edge (im8)
      (im8) edge (reg15)
      (reg15) edge (dm8)
      (dm8) edge (reg16);
\end{tikzpicture}
\end{center}
Assume Branch \textit{Not Taken}

- same performance as stalling when you’re wrong

```plaintext
beq $4, $0, there
and $12, $2, $5
or ...
add...
there: sub $12, $4, $2
```

Performance depends on percentage of time you guess right.

- Flushing an instruction means to prevent it from changing any permanent state (registers, memory, PC).
  - sounds a lot like a bubble...
  - But notice that we need to be able to insert those bubbles later in the pipeline

\textit{Assume Branch Not Taken}

\textit{Assume Branch Not Taken}

\textit{Assume Branch Not Taken}

\textit{Assume Branch Not Taken}

\textit{Assume Branch Not Taken}

\textit{Assume Branch Not Taken}
Reducing the Branch Delay

• can easily get to 2-cycle stall

Stalling for Branch Hazards

beq $4, $0, there and $12, $2, $5
or ...
add ...
sw ...

CSE 141 Allan Snavely
Reducing the Branch Delay

• Harder, but possible, to get to 1-cycle stall

Stalling for Branch Hazards

beq $4, $0, there
and $12, $2, $5
or ...
add ...
sw ...

CSE 141 Allan Snavely
The Pipeline with flushing for taken branches

• Notice the IF/ID flush line added.

Eliminating the Branch Stall

• There’s no rule that says we have to see the effect of the branch immediately. Why not wait an extra instruction before branching?
• The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches.
• The instruction after a conditional branch is always executed in those machines, regardless of whether the branch is taken or not!
Branch delay slot instruction (next instruction after a branch) is executed even if the branch is taken.

Filling the branch delay slot

add $5, $3, $7
sub $6, $1, $4
and $7, $8, $2
beq $6, $7, there
nop /* branch delay slot */
add $9, $1, $2
sub $2, $9, $5
...
there:
mult $2, $10, $11
Filling the branch delay slot

- The branch delay slot is only useful if you can find something to put there.
- If you can’t find anything, you must put a `nop` to insure correctness.

Branch Delay Slots

- This works great for this implementation of the architecture, but becomes a permanent part of the ISA.
- What about the MIPS R10000, which has a 5-cycle branch penalty, and executes 4 instructions per cycle???
Branch Prediction

- Always assuming the branch is not taken is a crude form of *branch prediction*.
- What about loops that are *taken* 95% of the time?
  - we would like the option of assuming *not taken* for some branches, and *taken* for others, depending on ???

```c
for (i=0;i<10;i++) {
    ...
    ...
}
...
...
add $i, $i, #1
beq $i, #10, loop
```
Two-bit predictors give better loop prediction

for (i=0; i<10; i++) {
  ...
  ...
  add $i, $i, #1
  beq $i, #10, loop
}

Pipeline performance

loop:  lw $15, 1000($2)
       add $16, $15, $12
       lw $18, 1004($2)
       add $19, $18, $12
       beq $19, $0, loop:

Allan Snavely
Control Hazards -- Key Points

- Control (or branch) hazards arise because we must fetch the next instruction before we know if we are branching or where we are branching.
- Control hazards are detected in hardware.
- We can reduce the impact of control hazards through:
  - early detection of branch address and condition
  - branch prediction
  - branch delay slots