Control Hazards

Where are branch conditions and target addresses resolved (when is the PC overwritten with the branch target address)?

When the branch instruction is in MEM stage
• the 3 instructions following the branch are already in the pipeline

When the branch instruction is in EX stage
• the 2 instructions following the branch are already in the pipeline
• How does this affect the cycle time?

Assume that the decision about “branching” takes place in the EX stage. Hence, when a branch decision is made two instructions are already in the pipe (started execution).

Example: consider the execution of the following code segment:

```
add $4, $5, $6
beq $1, $2, 10
lw $3, 300($0)
sub $7, $8, $9
sw $10, 4($8)
and $3, $2, $1
lw $3, 300($0)
beq $1, $2, 10
add $4, $5, $6
```

10 instructions

<table>
<thead>
<tr>
<th>IF stage</th>
<th>ID stage</th>
<th>EX stage</th>
<th>MEM stage</th>
<th>WB stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle 1</td>
<td>$4, $5, $6</td>
<td>$4, $5, $6</td>
<td>$4, $5, $6</td>
<td>$4, $5, $6</td>
</tr>
<tr>
<td>Cycle 2</td>
<td>$1, $2, 10</td>
<td>$1, $2, 10</td>
<td>$1, $2, 10</td>
<td>$1, $2, 10</td>
</tr>
<tr>
<td>Cycle 3</td>
<td>$3, 300($0)</td>
<td>$3, 300($0)</td>
<td>$3, 300($0)</td>
<td>$3, 300($0)</td>
</tr>
<tr>
<td>Cycle 4</td>
<td>$7, $8, $9</td>
<td>$7, $8, $9</td>
<td>$7, $8, $9</td>
<td>$7, $8, $9</td>
</tr>
<tr>
<td>Cycle 5</td>
<td>$10, 4($8)</td>
<td>$10, 4($8)</td>
<td>$10, 4($8)</td>
<td>$10, 4($8)</td>
</tr>
</tbody>
</table>

Branch condition resolved

What is wrong and what can be done?
Adding no-ops (a software solution)

Make the compiler add no-ops after the branch instruction.
- Why is that not ideal??

<table>
<thead>
<tr>
<th>Cycle 2</th>
<th>IF stage</th>
<th>ID stage</th>
<th>EX stage</th>
<th>MEM stage</th>
<th>WB stage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>beq $1, $2, 10</td>
<td>add $4, $5, $6</td>
<td>beq $1, $2, 10</td>
<td>add $4, $5, $6</td>
<td>beq $1, $2, 10</td>
</tr>
<tr>
<td>Cycle 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>no-op</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lw $3, 300($0)</td>
<td>no-op</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 5</td>
<td>lw $3, 300($0)</td>
<td>lw $3, 300($0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add $3, $2, $1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch condition resolved

The branch condition will be resolved in cycle 4 and the correct instruction will enter the pipe in cycle 5

A hardware solution

Introduce a bubble (a no-op introduced by the hardware) to abort unwanted instructions.

<table>
<thead>
<tr>
<th>Cycle 2</th>
<th>IF stage</th>
<th>ID stage</th>
<th>EX stage</th>
<th>MEM stage</th>
<th>WB stage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>beq $1, $2, 10</td>
<td>add $4, $5, $6</td>
<td>beq $1, $2, 10</td>
<td>add $4, $5, $6</td>
<td>beq $1, $2, 10</td>
</tr>
<tr>
<td>Cycle 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lw $3, 300($0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub $7, $8, $9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>and $3, $2, $1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Reducing the number of aborted instructions**

Branch condition and address resolved in the EX stage

Branch address can be resolved in ID stage. Branch condition can also be resolved in the ID stage if we use a comparator.

- why would this help?
- Does this have any effect on the cycle time?

**Resolving the branches**

The mux at the input of PC selects the branch PC when
- the control indicates that the instruction in the ID stage is a `beq`
- the zero output of the comparator is true

When the mux selects the branch PC, then input a no-op to the IF/ID buffer (flush)
Flushing Instructions (creating a bubble)

The effect of control hazard on throughput

- Assume that when the branch is resolved, K instructions following the branch are already in the pipeline.
- If control hazards are dynamically resolved, then each taken branch introduces K bubbles in the pipeline.
- Hence, the average number of clock cycles to execute an instruction is:
  \[ \text{CPI} = \text{CPI}_{\text{noch}} + \alpha \ast \pi \ast K \]
  where
  \( \text{CPI}_{\text{noch}} \) is the CPI with no control hazard
  \( \alpha \) is the fraction of branch instructions in the instruction mix
  \( \pi \) is the probability a branch is actually taken

**Example:** if branches are dynamically resolved in the EX stage, 10% of the instructions are branches and the probability that a branch is taken is 40%, then
\[ \text{CPI} = 1 + 2 \ast 0.1 \ast 0.4 = 1.08 \]
Hence, the average execution time of an instruction is 1.08 * clock cycle time

- For the software solution, where the compiler adds K no-ops after each branch,
  \[ \text{CPI} = \text{CPI}_{\text{noch}} + \alpha \ast K \]
**Pipeline depth vs. branch penalty**

- Today’s processors employ a deep pipeline (possibly more than 20 stages!) to increase the clock rate
  - Many stages means smaller amount of work per stage ⇒ shorter time needed per stage ⇒ higher clock rate!

- But what about branch penalty?
  - Penalty depends on the pipeline length!
  - Branches represent 15~20% of all instructions executed

- Situation is compounded by the increased issue bandwidth (will discuss when we talk about superscalar processors)

---

**Branch handling strategies**

- No prediction (equivalent to predict “NOT TAKEN”)
  - Execute fall-off instructions that follow the branch (at PC+4, PC+8, …)
  - (PC+4) is computed every cycle, so use it to get the fall-off instructions
  - Squash (or cancel) instructions in pipeline if branch is actually taken

- Predict “TAKEN”
  - 67% MIPS branches are taken, on average
  - Start fetching from the branch target as soon as it is available
  - Useful if target address is computed earlier than the branch condition.

- Stall until branch direction is known (compiler can add no-ops)
- Dynamic branch prediction
Dynamic branch prediction

- Goal
  - Predict the branch outcome (taken or not taken) and the branch target address (if taken, where should we go?) at run time.
  - In short, just figure out what is the next PC (target address or PC+4)?

- Can we dynamically predict, at run time, the next PC when we are fetching from the current PC?

- Need to dynamically predict:
  - Branch condition
  - Branch target address

What to predict – T/NT

- Let’s first focus on predicting taken (T)/not taken (NT)

- Branch outcome may be biased for individual branches
  - Example: in loops, branches may be mostly taken or mostly not taken

- Dynamic prediction using a simple 1-bit predictor
  - Remember the last behavior (taken/not-taken) using a hash table
    - $2^N$ entries – index using N bits from PC (for some N)
    - Hashing $\rightarrow$ may have collision
    - Collision $\rightarrow$ problem??
  - When should we check the table?
  - If prediction is wrong
    - Take corrective action
    - Update the prediction buffer
Target prediction (Branch Target Buffer – BTB)

- We would like to check the table while fetching the instruction
  - Are we sure that the BTB entry refers to a branch instruction?
  - Are we sure it refers to the correct branch instruction?
- Should store and match all bits of PC.
- Add the last branch address to the table (to predict the target)

Target prediction with Branch Target Buffer

Records the target address for the last execution of the branch instruction
Target prediction with BTB

- If *match* and “Prediction = taken”, use the *stored target* as the next PC
- If *match* and “Prediction = untaken”, use PC+4 as the next PC
- If no match, then use PC+4 because either
  - the instruction is not a branch
  - It is a branch, but no prediction → assume branch is not taken

- After resolving a branch, if we find that it was mispredicted
  - Undo the damage (flush the wrong instruction)
  - Restore correct PC
  - Update BTB with new information

Branch prediction & pipelining

Assuming that branch condition and target are resolved in ID stage

A similar chart may be drawn if branch condition/target are resolved in EX