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?

Control Hazards

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 pipeline (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)
```

What is wrong and what can be done?

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

Branch condition resolved
### 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</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>2</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>3</td>
<td>no-op</td>
<td>no-op</td>
<td>no-op</td>
<td>no-op</td>
<td>no-op</td>
</tr>
<tr>
<td>4</td>
<td>lw $3, 300($0) or and $3, $2, $1</td>
<td>no-op</td>
<td>no-op</td>
<td>no-op</td>
<td>no-op</td>
</tr>
<tr>
<td>5</td>
<td>lw $3, 300($0) or and $3, $2, $1</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>
</tbody>
</table>

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</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>2</td>
<td>beq $1, $2, 10</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
<td>beq $1, $2, 10</td>
</tr>
<tr>
<td>3</td>
<td>lw $3, 300($0)</td>
<td>beq $1, $2, 10</td>
<td>beq $1, $2, 10</td>
<td>beq $1, $2, 10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>sub $7, $8, $9</td>
<td>lw $3, 300($0)</td>
<td>lw $3, 300($0)</td>
<td>lw $3, 300($0)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>and $3, $2, $1</td>
<td>and $3, $2, $1</td>
<td>and $3, $2, $1</td>
<td>and $3, $2, $1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>beq $1, $2, 10</td>
<td>beq $1, $2, 10</td>
<td>beq $1, $2, 10</td>
<td>beq $1, $2, 10</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>beq $1, $2, 10</td>
<td>beq $1, $2, 10</td>
<td>beq $1, $2, 10</td>
<td>beq $1, $2, 10</td>
<td></td>
</tr>
</tbody>
</table>

Branch condition resolved.
Reducing the number of aborted instructions

Branch condition and address resolved in the EX stage

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

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

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)
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
  \[ \text{CPI} = \text{CPI}_{\text{noch}} + \alpha \times \pi \times K \]
  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
  \[ \text{CPI} = 1 + 2 \times 0.1 \times 0.4 = 1.08 \]
  cycles per instruction (assuming \( \text{CPI}_{\text{noch}} = 1 \))

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,
  \[ \text{CPI} = \text{CPI}_{\text{noch}} + \alpha \times K \]
**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 know (compiler can add no-ops)
- Delayed branch
Delayed branch

• Change the branch semantics such that the “N” instructions after a branch are always executed before branching takes place.
  – The N instructions are executed regardless of the branch outcome
  – N is set to be the number of cycles needed to resolve the branch

• The compiler will try to fill the N slots with useful work
  – If can't find instructions to fill the slots, use NOPs.

• Simplifies hardware (eliminates hazards by changing the semantics)
• Possibly good performance – if slots are filled with useful instructions

• Code size will increase??

Finding instructions to fill the delay slots

a. From before
   add $s1, $s2, $s3
   if $s2 = 0 then
       Delay slot
   Becomes
   if $s2 = 0 then
       add $s1, $s2, $s3

b. From target
   sub $s4, $s5, $s6
   ... add $s1, $s2, $s3
   if $s1 = 0 then
       Delay slot
   Becomes
   sub $s1, $s2, $s3
   add $s1, $s2, $s3
   if $s1 = 0 then
       sub $s4, $s5, $s6

C. From fall-through
   add $s1, $s2, $s3
   if $s1 = 0 then
       Delay slot
   Becomes
   sub $s1, $s2, $s3
   sub $s4, $s5, $s6
Branch prediction

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

- Can we dynamically predict, at runtime, 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 → may have collision
    - Collision → 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 *Predict Taken*, use the *stored target* as the next PC
- If *match* and *Predict not Taken*, 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 and finding that it was mispredicted
  - Undo the damage (flush the wrong instruction)
  - Restore correct PC
  - Update BTB with new information

---

**Shortcoming of 1-bit dynamic predictor**

- Remember the last behavior of the branch

```plaintext
for (i=0; i< 100; i++) {
    A[i] = B[i] * C[i];
    D[i] = E[i] / F[i];
}
```

- How many prediction hits and misses? Prediction accuracy?

```plaintext
for (i=0; j<100; j++) {
    for (i=0; i< 5; i++) {
        A[i] = B[i] * C[i];
        D[i] = E[i] / F[i];
    }
}
```

<table>
<thead>
<tr>
<th>Predicted</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>NT</th>
<th>T</th>
<th>T</th>
<th>NT</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

This branch is predicted wrong twice every inner loop invocation (every 5 branches)
2-bit predictor

- Requires two consecutive miss-predictions to flip direction

![Diagram of 2-bit predictor](image)

2-Bit Branch prediction buffer

![Diagram of 2-bit branch prediction buffer](image)
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)