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

- 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 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)
- 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 $s4, $s5, $s6
add $s1, $s2, $s3
if $s1 = 0 then
  Delay slot
```

c. From fall-through

```
add $s1, $s2, $s3
if $s1 = 0 then
  Delay slot
```

Becomes
```
sub $s4, $s5, $s6
if $s1 = 0 then
  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 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 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

```
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?

```
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>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>
</tr>
</tbody>
</table>

this is a conditional branch

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

- Requires two consecutive miss-predictions to flip direction

2-Bit Branch prediction buffer

- 2-bit counters (keeps state of predictor)

PC

N bits

taken/not taken

indexing

- Taken
- Not taken

Predict taken
Predict not taken
Predict taken
Predict not taken

Correct prediction
Miss-predicted once
Change prediction
Correct prediction
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)