Control Dependences (branch/jump) in Pipelines

- ‘Data Dependence’ = next instruction uses data (register/memory) from previous
- ‘Control Dependence’ = which is the next instruction depends on the previous
- Control Dependences arise from ‘Control Transfer Instructions (CTI)’
- Control Transfer Instructions (CTI) are: Jump and Branch Instructions
- ‘Jumps’ are Unconditional CTI’s: they always transfer control
- ‘Branches’ are Conditional CTI’s: whether or not they transfer control depends on the result of a data comparison that they have to perform

Statistics (rough numbers, in a majority of programs, but NOT always so):

- Branches are about 15–16% of all (‘dynamically’) executed instructions in a program  
  – about 2/3 of executed branches are ‘taken’ (successful) = ~10% of all instr.  
  – about 1/3 of executed branches are not taken (unsuccessful) = ~5% of all instr.  
  – most backwards branches appear in loops, and they are about 90% taken
- Jumps are about 4–5% of all executed instructions in a program  
  – procedure calls are about 1%, and returns another ~1%, of all executed instr.
Branch Taken example

- In modern processors, branch latency is quite long
- In our simple pipeline, branch latency is 2 cycles (read registers; compare)
  (with MIPS–style comparisons (beq/bne only) it could even be 1 cycle)
- Example here with 3–cycle branch latency

- need to abort speculative execution before it causes permanent damage: before DM and WB stages

- In this example, each taken branch causes the loss of 3 extra clock cycles
- About 2/3 of all executed branches are taken, so this is a heavy loss
**Branch Not-taken example**

- A not-taken branch is equivalent to a noop instruction
- In the simple fetch–next–below policy that we used up to now, not-taken branches cost NO extra clock cycle

- Good thing that these cost no extra cycle, but they are the minority of branches

- Can we do any better for the majority of branches (taken branches – and jumps) ??
Branch Target known in 1 Cycle

- **Branch Prediction:**
  - Simplest possible prediction, here: branches always taken
  - ~65% accuracy: about 2/3 of executed branches are taken

- Opcode decode and PC+2*Imm computation can both be done in 1 cycle

- In this example, each taken branch causes the loss of 1 extra clock cycle
Branch with failed Prediction example

- Simplest possible prediction, again: branches always taken
  ~65% accuracy: about 2/3 of executed branches are taken
  (reason: loop branches (backwards) ~90% taken)

- In this example, each non-taken branch causes the loss of 2 extra clock cycles
- ~5% jumps + (~15% branches * 2/3 taken) ~= 15% good prediction, versus ~5% bad prediction
Branch Target Buffer (BTB)

- A small table – a cache, like a hash table – containing pairs of (instruction) addresses for which there is statistical evidence that their next–PC is something other than PC+4

PC of a jump or branch–likely instruction;

Target PC to which this instruction usually went, in the past.

- A ‘best approximation’ – not necessarily correct information
- Branches that are believed not–taken are NOT entered into the BTB
- Like IM –the Instruction Cache– this will oftentimes ‘overflow’: old pairs are removed to make room for more recent ones
- May be complemented with a small hardware stack:
  - on every call (jal ra,...), push the return address;
  - on every return (jr ra), pop an address and predict jumpin to that one

- In parallel with each Fetch, search the fetched instruction’s PC value in the BTB

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>260</td>
<td>200</td>
</tr>
<tr>
<td>40</td>
<td>72</td>
</tr>
<tr>
<td>88</td>
<td>120</td>
</tr>
<tr>
<td>180</td>
<td>160</td>
</tr>
</tbody>
</table>

36: add ...
40: beq ..., goto72
44: sd ...
48: and ...
52: or ...
72: ld ...
76: xor ...
compare

PC + 4 (branch not taken)

A

+4

PC

BTB

PredictedNextPC

ActualNextPC

1 = prediction was correct, continue as is
0 = misprediction! ==> Flush all subsequent instructions in the pipeline and fix fetch-address

rs1 + Imm (jalr)
PC + 2*Imm (jal, br taken)
PC + 4 (branch not taken)

return address to rd (jar, jalr)

(*): when IRvalid==0, treat IR as containing a noop instruction

(*) when IRvalid==0, treat IR as containing a noop instruction

decide branch

rs1
compare
rs2

Datapath

IRvalid

control signals
When the BTB prediction is Correct

- When a matching BTB entry is found, use its Prediction; else, fetch from PC+4

- When Prediction is Correct, NO extra clock cycles are lost!
When the BTB prediction is Wrong

- Prediction says: After fetching from 40, fetch from 72
- But this time, the branch ends up going the other way: to 44

- When Mispredicted, branches cost 3 extra clock cycles in this pipeline
1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

```
outer: ...
...
inner: ...
...
beq ..., ..., inner
...
beq ..., ..., outer
```

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around
2-Bit Predictor

- Only change prediction on two successive mispredictions