Dynamic Hardware Branch Prediction

- Predict the outcome of a branch
- Change the prediction after observing a few iterations
- To achieve good effectiveness we must
  - Have accurate prediction technique
  - Have a low cost for misprediction

Local Prediction: Branch Prediction Buffer

- A table indexed by low bits of branch instruction address
  - It contains a bit indicating whether the branch was recently taken or not
  - If it turns out we have been wrong the bit is inverted

1-bit Branch Prediction Buffer

- Problem – even simplest branches are mispredicted twice

LD R1, #5
Loop: LD R2, 0(R5)
ADD R2, R2, R4
STORE R2, 0(R5)
ADD R5, R5, #4
SUB R1, R1, #1
BNEZ R1, Loop

2-bit Branch Prediction Buffer

- To amend this we will use 2 bits, we must miss twice before we change our prediction

n-bit Branch Prediction Buffer

- We can generalize this technique to n-bit prediction buffers
  - When the counter is $\geq 2^{n-1}$, branch is predicted as taken
  - Those predictors are not much more accurate than 2-bit
Correlating (Global) Branch Predictors

- Assign two prediction bits, one if the previous branch was not taken, the other if it was taken

\[ \begin{align*}
\text{b1: } & \text{ if } (d=0) \\
& \text{if (d==0)} \\
\text{d=1; } & \\
\text{b2: } & \text{ if } (d=1) \\
& \text{if (d==1)} \\
\text{If b1 is taken, b2 is taken} & \\
& \text{……} \\
\text{L2: } & \text{If b1 is not taken, b2 is not taken} \\
& \text{……} \\
\text{One bit indicating what to do} & \text{One bit indicating what to do} \\
& \text{if one previous branch was not taken} & \text{if one previous branch was taken} \\
\end{align*} \]


Correlating Branch Predictors

- Assign two prediction bits, one if the previous branch was not taken, the other if it was taken

\[ \begin{align*}
\text{b1: } & \text{ BNEZ R1, L1} \\
& \text{DADDUI R1, R0, #1} \\
\text{L1: } & \text{DSUBUI R3, R1, #1} \\
\text{b2: } & \text{ BNEZ R3, L2} \\
& \text{……} \\
\text{L2: } & \text{……} \\
\end{align*} \]

This is (1,1) predictor ⇒ it uses outcome of 1 previous branch to do prediction with 1-bit predictor

<table>
<thead>
<tr>
<th>R1=7</th>
<th>b1 prediction</th>
<th>b1 action</th>
<th>New b1 prediction</th>
<th>b2 prediction</th>
<th>b2 action</th>
<th>New b2 prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NT/NT</td>
<td>T</td>
<td>NT/NT</td>
<td>NT/NT</td>
<td>T</td>
<td>NT/NT</td>
</tr>
<tr>
<td>2</td>
<td>T/NT</td>
<td>T</td>
<td>T/NT</td>
<td>NT/NT</td>
<td>T</td>
<td>NT/NT</td>
</tr>
<tr>
<td>0</td>
<td>T/NT</td>
<td>NT</td>
<td>NT/NT</td>
<td>NT/NT</td>
<td>NT</td>
<td>NT/NT</td>
</tr>
</tbody>
</table>

Correlating Branch Predictors (m,n)

- Observe behavior of m previous branches, use n-bit predictor

\[ \begin{align*}
\text{One bit indicating what to do} & \text{One bit indicating what to do} \\
& \text{if one previous branch was not taken} & \text{if one previous branch was taken} \\
\text{One bit indicating what to do} & \text{One bit indicating what to do} \\
& \text{if m previous branches were not taken} & \text{if m previous branches were taken} \\
\text{n bits indicating what to do} & \text{n bits indicating what to do} \\
& \text{if m previous branches were not taken} & \text{if m previous branches were taken} \\
\end{align*} \]

2^m combinations, n-bits each

Correlating Branch Predictors (m,n)

- How many bits do we need for (m,n) predictor?

\[ 2^m + n + 2 \]

Tournament Predictors

- Combine one global and one local predictor with a selector

[Diagram showing tournament predictors]
High-performance Instruction Delivery
- Issue multiple instructions per clock cycle
- It is not enough to predict branches correctly, we also must resolve branch target quickly (end of IF)
- Use branch target buffer, cache branch target address for every branch
- For MIPS pipeline we will resolve branch during ID stage
  - Branch target buffer along with accurate prediction saves 1 cycle as everything is done during IF stage

Branch Target Buffer
- In IF, look up instruction address in BTB
- If we find the address (exact match), it is a branch
- It is predicted taken (otherwise it wouldn’t be there)
- Use cached address in BTB to fetch next instruction

Example
Determine total branch penalty for a BTB assuming the following penalty cycles

<table>
<thead>
<tr>
<th>Instruction in buffer</th>
<th>Prediction</th>
<th>Outcome</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>taken</td>
<td>taken</td>
<td>0</td>
</tr>
<tr>
<td>yes</td>
<td>taken</td>
<td>not taken</td>
<td>2</td>
</tr>
<tr>
<td>no</td>
<td>taken</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>not taken</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Assume 60% of branches are taken, prediction accuracy is 90% and hit rate in BTB is 90%.

Branch Target Buffer
- Sometimes BTB stores a few instructions from the target, instead of target address
- This allows for 0-cycle unconditional branches

Integrated Instruction Fetch Units
- Fetch several instructions per cycle
  - Integrated branch prediction
  - Instruction prefetch and buffering

Return Address Predictors
- Procedure may be called from several places, return addresses will differ
- Use a stack to store return address as procedure is called, pushing address on call, popping on return
Multiple Issue
- We will issue several instructions per cycle
- Superscalar processors issue varying number of MIPS-like instructions per cycle
  - Statically scheduled (in-order execution)
  - Dynamically scheduled (out-of-order execution)
- Very Long Instruction Word (VLIW) processors have long instruction words that contain fixed number of MIPS-like instructions
  - One large instruction or several instructions with explicitly indicated parallelism
  - Statically scheduled (in-order execution)
  - They issue one such word per cycle

Statically Scheduled Superscalar Proc.
- Issue instructions in issue packets
  - From 0 to 8 instructions per issue packet
- All hazards are checked in hardware when instructions are issued (dynamic issue capability)
  - Among issuing instructions in the issue packet
  - Between issuing instructions in the issue packet and the ones still in execution
  - If one instruction from the issue packet cannot be issued due to hazard, only preceding instructions are issued

Statically Scheduled Superscalar MIPS
- Assume 2 instructions issue per cycle
  - One is load, store, branch or integer ALU
  - Other is FP operation
  - This combination reduces risk of hazards between these two instructions
  - Fetch two instructions, check for hazards and issue them
    - If one instruction is load, store or move than we could have data hazard and maybe structural hazard for FP register file ports

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>Int instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>Int instruction</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Problems might arise:
- We will need additional hardware in the pipeline
- Maintaining precise exceptions is hard because instructions may complete out of order
- Hazard penalties are longer
Dynamically Scheduled Superscalar MIPS

- Extend Tomasulo’s algorithm to support issue of 2 instructions per cycle
- We must issue instructions to reservation stations in order
- Issue stage can either be
  - Pipelined – issue one instruction in half cycle, another one in another half
  - Extended – add more hardware and issue instructions simultaneously

### Homework

- Due Tuesday, October 26 by the end of the class
- Submit either in class (paper) or by E-mail (PS or PDF only) or bring the paper copy to my office
- Show scheduling of the following code using Tomasulo’s algorithm
  - Assume two load units, two FP adders, and two FP multipliers. It takes 2 cycles to execute load (one to calculate effective address and one to read from memory), 2 cycles for ADD and SUB and 10 cycles for MUL. Compare the execution time of this code with TA and with scoreboard (homework 5). Which technique is better and why?

```plaintext
LD F2, 0(R2)
ADD F8, F2, F2
MUL F6, F4, F8
SUB F6, F2, F4
```