Improvement: Correlating Predictors

different branches may be *correlated*

- outcome of branch depends on outcome of other branches
  - makes intuitive sense (programs are written this way)
- e.g., if the first two conditions are true, then third is false

```plaintext
if (aa == 2) aa = 0;
if (bb == 2) bb = 0;
if (aa != bb) { . . . }
```

revelation: prediction = f(branch PC, recent branch outcomes)

- revolution: BP accuracies increased dramatically
- lots of research in designing that function for best BP
Correlating (Two-Level) Predictors

- **branch history shift register (BHR)** holds recent outcomes
  - combination of PC and BHR accesses BHT
  - basically, multiple predictions per branch, choose based on history

**Design space**

- number of BHRs
  - multiple BHRs (“local”, Intel)
  - 1 global BHR (“global”, everyone else)

- PC/BHR overlap
  - full, partial, none (concatenated?)

- popular design: Gshare [McFarling]
  - 1 global BHR, full overlap, \( f = \text{XOR} \)
### Correlating Predictor Example

- example with alternating T,N (1-bit BHT, no correlation)

<table>
<thead>
<tr>
<th>state/prediction</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch outcome</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
</tr>
<tr>
<td>mis-prediction?</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

- add 1 1-bit BHR, concatenate with PC
  - effectively, two predictors per PC
  - top (BHR=N) bottom (BHR=T) active entry

<table>
<thead>
<tr>
<th>state/prediction</th>
<th>N</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch outcome</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
</tr>
<tr>
<td>mis-prediction?</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hybrid/Competitive/Tournament Predictors

*observation:* different schemes work better for different branches

*idea:* multiple predictors, choose on per static-branch basis

**mechanics**

- two (or more) predictors
- chooser
  - if chosen predictor is wrong...
  - ...and other is right...
  - ...flip chooser
- popular design: Gselect [McFarling]
  - Gshare + 2-bit saturating counter

[Diagram of hybrid/predictor architecture]
Branch Target Buffer (BTB)

branch PC $\Rightarrow$ target PC

- target PC available at end of IF stage
  + no bubble for correct predictions

- branch target buffer (BTB)
  - index: branch PC
  - data: target PC (+ T/NT?)
  - tags: branch PC (why are tags needed here and not in BHT?)
    - many more bits per entry than BHT
  - considerations: combine with I-cache? store not-taken branches?

- branch target cache (BTC)
  - data: target PC + target instruction(s)
  - enables “branch folding” optimization (branch removed from pipe)
Jump Prediction

exploit behavior of different kinds of jumps to improve prediction

• function returns
  • use *hardware return address stack (RAS)*
  • call pushes return address on top of RAS
  • for return, predict address at top of RAS and pop
    – trouble: must manage speculatively

• indirect jumps (switches, virtual functions)
  • more than one taken target per jump
  • path-based BTB [Driesen+Holzle]
Branch Issues

issue1: how do we know at IF which instructions are branches?
  • BTB: don’t need to “know”
    • check every instruction: BTB entry ⇒ instruction is a branch

issue2: BHR (RAS) depend on branch (call) history
  • when are these updated?
    • at WB is too late (if another branch is in-flight)
    • at IF (after prediction)
    • must be able to recover BHR (RAS) on mis-speculation (nasty)
Limitations of Branch Prediction

Branch prediction is extremely important for performance
• So, how well can we do?
• Can you think of better branch predictors?
• Can we ever have perfect branch prediction?

Thought experiment:
• Roughly every 5th instruction is a conditional branch
• Assume you predict 90% correctly
• Assume you don’t resolve a branch for 20 cycles
• How many branches must we predict to avoid stalling?
• What’s probability that all predictions were correct?
Reminder of Where We Are

- principles of pipelining
  - pipeline depth: clock rate vs. number of stalls (CPI)
- hazards
  - structural
  - data (RAW, WAR, WAW)
  - control
- multi-cycle operations
  - structural hazards, WAW hazards
- interrupts
  - precise state
Adding Multi-Cycle Operations

RISC tenet #1: “single-cycle operations”

• why was this such a big deal?

• fact: not all operations complete in 1 cycle
  • FP add, int/FP multiply: 2–4 cycles, int/FP divide: 20–50 cycles
  • L1 data cache misses: 10–150 cycles!

• slow clock cycle down to slowest operation?
  – can’t without incurring huge performance loss

• solution: extend pipeline - add pipeline stages to EX
Extended Pipeline

- separate integer/FP, pipe register files
  - loads/stores in integer pipeline only (why?)

- additional, parallel functional units
  - \( E^+ \): FP adder (2 cycles, pipelined)
  - \( E^* \): FP/integer multiplier (4 cycles, pipelined)
  - \( E/ \): FP/integer divider (20 cycles, not pipelined)
Multi-Cycle Example

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>divf</strong> $f_0, f_1, f_2$</td>
<td>F</td>
<td>D</td>
<td>E/</td>
<td>E/</td>
<td>E/</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>mulf</strong> $f_0, f_3, f_4$</td>
<td>F</td>
<td>D</td>
<td>E*</td>
<td>E*</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>addf</strong> $f_5, f_6, f_7$</td>
<td>F</td>
<td>D</td>
<td>E*</td>
<td>E*</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>subf</strong> $f_8, f_6, f_7$</td>
<td>F</td>
<td>D</td>
<td>*</td>
<td>E*</td>
<td>E+</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>mulf</strong> $f_9, f_8, f_7$</td>
<td>F</td>
<td>D</td>
<td>*</td>
<td>*</td>
<td>E*</td>
<td>E*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- write-after-write (WAW) hazards
- register write port structural hazards
- functional unit structural hazards
- elongated read-after-write (RAW) hazards
Another Multi-Cycle Example

example: SAXPY (math kernel)

\[ Z[i] = A*X[i] + Y[i] \quad // \text{single precision} \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldf f2,0(r1)</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul f6,f0,f2</td>
<td>F</td>
<td>D</td>
<td>d*</td>
<td>E*</td>
<td>E*</td>
<td>E*</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldf f4,0(r2)</td>
<td>F</td>
<td>p*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add f8,f6,f4</td>
<td>F</td>
<td>D</td>
<td>d*</td>
<td>d*</td>
<td>E+</td>
<td>E+</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stf f8,0(r3)</td>
<td>F</td>
<td>p*</td>
<td>p*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1,r1,#4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2,r2,#4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r3,r3,#4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

KEY: \( \text{d*} = \text{data stall, p* = stalled behind older stalled instruction} \)
Register Write Port Structural Hazards

where are these resolved?

• multiple writeback ports?
  – not a good idea (why not?)

• in ID?
  • reserve writeback slot in ID (writeback reservation bits)
    + simple, keeps stall logic localized to ID stage
    – won’t work for cache misses (why not?)

• in MEM?
  + works for cache misses, better utilization
    – two stall controls (F/D and M/W) must be synchronized

• in general: cache misses are hard
  • don’t know in ID whether they will happen early enough (in ID)
WAW Hazards

how are these dealt with?

• stall younger instruction writeback?
  + intuitive, simpler
    – lower performance (cascading writeback structural hazards)

• abort (don’t do) older instruction writeback?
  + no performance loss
    – but what if intermediate instruction causes an interrupt? (next)
Dealing With Interrupts

interrupts (aka faults, exceptions, traps)
  • e.g., arithmetic overflow, divide by zero, protection violation
  • e.g., I/O device request, OS call, page fault

classifying interrupts
  • terminal (fatal) vs. restartable (control returned to program)
  • synchronous (internal) vs. asynchronous (external)
  • user vs. coerced
  • maskable (ignorable) vs. non-maskable
  • between instructions vs. within instruction