CS 104
Computer Organization and Design

Branch Prediction
Branch Prediction

• Quick Overview
  • Now that we know about SRAMs...
Branch Prediction 10K feet

- Two (separate) tasks:
  - Predict taken/not taken
  - Predict taken target
Branch Prediction 10K feet

• Two (separate) tasks:
  • Predict taken/not taken
  • Predict taken target

• High level solution (both tasks):
  • SRAM “array” to remember most recent behaviors
  • Kind of like a cache, indexed by PC bits, but different
    • Typically no next level (but can have 2 levels)
    • Can skip tag, or use partial tag
      • Predictor: OK to be wrong (as long as we fix it)
Branch Target Buffer (BTB)

- Branch Target Buffer
  - SRAM array, holds recent taken targets
  - Example: 4K entries, direct mapped
  - Can be set-associative
  - Each entry holds partial PC (low order bits)
    - Assume high bits unchanged (why?)
    - Example: 16 bits

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>......</th>
<th>......</th>
<th>4097</th>
<th>4242</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01F3</td>
<td>4242</td>
<td>1234</td>
<td>......</td>
<td>......</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branch Target Buffer (BTB)

- Branch Target Buffer
  - SRAM array, holds recent taken targets
  - Example: 4K entries, direct mapped
  - Can be set-associative
  - Each entry holds partial PC (low order bits)
    - Assume high bits unchanged (why?)
    - Example: 16 bits

- Prediction of taken target:
  - Use PC bits 2—13 to index BTB (why these bits?)
  - Replace PC bits 2—17 with value in BTB
Branch Target Buffer (BTB)

- **Branch Target Buffer**
  - SRAM array, holds recent taken targets
  - Example: 4K entries, direct mapped
  - Can be set-associative
  - Each entry holds partial PC (low order bits)
    - Assume high bits unchanged (why?)
    - Example: 16 bits

- **Prediction of taken target:**
  - Use PC bits 2—13 to index BTB (why these bits?)
  - Replace PC bits 2—17 with value in BTB

- **Update (how do values get into predictor?)**
  - At execute, if branch is taken write target into BTB
  - Use PC bits 2—13 to index for write also (same entry)
Target Prediction: BTB collisions

- PCs may collide in BTB
  - Example: 0x10000000 and 0x20000000 (both index 0)
  - Could use tags (or partial tags)
    - Better to just guess “not taken” than “taken to bogus target”
    - Why?
Target Prediction: BTB collisions

- PCs may collide in BTB
  - Example: 0x10000000 and 0x20000000 (both index 0)
  - Could use tags (or partial tags)
    - Better to just guess “not taken” than “taken to bogus target”
    - Why?
  - What if 0x10000000 is a branch, and 0x20000000 is not?
    - Pipeline may predict bogus next PC for non-branch
      - Fine as long as detected/fixed (extra checking)
      - Usually checked in decode if possible
    - Alternative: pre-decode bits
      - Add bits in I$ to say “is this a branch”
      - Know if not a branch while predicting
      - Bits set on I$ fill path (examine bits coming from L2)
Our branch predictor (so far)

- Missing piece (???): Direction predictor
  - Should we use the taken target (from BTB) or not?
Direction Prediction

- Need to predict “taken” (T) or “not taken” (N)
  - This is typically the hard part, by the way

- Simplest approach: just guess “same as last time”
  - Actually, kind of not bad:
    - Loops: almost always right (taken)
    - Error checks: almost always right (no error)
    - ...etc..

- Implementation:
  - SRAM, indexed by PC bits
  - 1 bit per entry: 1 = taken, 0 = not taken
  - No tags.
  - Collisions? Meh—they happen
Direction Prediction: Example

• Consider:

```c
for (int i = 0; I < 10000000; i++) {
    for (int j = 0; j < 6; j++) {
        //stuff
    }
}

Branches outcomes:
    TTTTTNTTTTTTTNTTTTTTTNTTTTTTTNTTTTTTTNTTTTTTTNT...
Direction Prediction: Example

- Consider:

```java
for (int i = 0; I < 10000000; i++) {
    for (int j = 0; j < 6; j++) {
        //stuff
    }
}
```

Branches outcomes:

```
TTTTTNTTTTTTTNTTTTTTTTTTTTTNTTTTTTTTNT...
```

Predictions:

```
NTTTTTTNTTTTTTTNTTTTTTTNTTTTTTTNTTTTTTTT...
```
Direction Prediction: Can we do better?

Branches outcomes:

```
TTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTTN
```

Predictions:

```
NTTTTTTNTTTTTTNTTTTTTNTTTTTTN
```

- **Problem:**
  - A little too quick to react
  - One-off difference causes two mis-predictions

- **Solution:**
  - Slow down changes in prediction: 2-bit counters
  - T (11), t (10), n (00), N (01)
  - “Strongly” (T/N) and “weakly” (t/n) taken/not taken
  - Updates: taken -> increment, not taken -> decrement
Direction Prediction: Can we do better?

Branches outcomes:

\[ TTTTTN \ldots \]

Predictions:

\[ NTTTTT \ldots \]

- Problem:
  - A little too quick to react
  - One-off difference causes two mis-predictions

- Solution:
  - Slow down changes in prediction: 2-bit counters
  - \( T \ (11), \ t \ (10), \ n \ (00), \ N \ (01) \)
  - “Strongly” (T/N) and “weakly” (t/n) taken/not taken
  - Updates: taken-> increment, not taken -> decrement
Can we do even better still?

- Our branches have a very regular pattern
  - 6Ts, then 1 N
  - We really should be able to get them all right... right?

- Real predictors use **history**
  - Take recent branch outcomes (NTTTTTT = 0111111)
  - XOR with PC to form table index
  - Same PC, different history -> different index -> different counter
  - Would predict previous example perfectly

- Also useful for correlation of branches
  - Nearby branches with related outcomes (why is this common?)
Direction Prediction: Continued..

- Real direction predictors more complex even still
  - Multiple tables with choosers (hybrid history schemes)
- Research ideas too
  - Late 90s/early 2000s: think up bpred idea, publish, repeat
- Big impediment to performance/hard to get well
- Also research ideas for how to get around it
  - Control Independence: predicting reconvergence point easier
Predicting returns

- Previous things don’t work well on “return” instructions
  - jr $ra
  - Why not?
Predicting returns

- Previous things don’t work well on “return” instructions
  - jr $ra
  - Why not?
  - Functions called from many places
    - Previous place to return to, not always current place to return to...
  - But should be predictable: why?
Predicting returns

- Previous things don’t work well on “return” instructions
  - jr $ra
  - Why not?
  - Functions called from many places
    - Previous place to return to, not always current place to return to...
  - But should be predictable: why?
    - Matches up with jal’s PC +4
    - In stack-like fashion
  - So....
Predicting returns

• Previous things don’t work well on “return” instructions
  • jr $ra
  • Why not?
  • Functions called from many places
    • Previous place to return to, not always current place to return to...
  • But should be predictable: why?
    • Matches up with jal’s PC +4
    • In stack-like fashion
  • So....

• “Return Address Stack” (aka “Link Stack”)
  • Predictor tracks a stack of recent jals
  • Encounter a jr $ra? Pop stack for predicted target