Lecture: Branch Prediction, Out-of-order Processors

- Topics: branch predictors, out-of-order intro, register renaming
2-Bit Bimodal Prediction

- For each branch, maintain a 2-bit saturating counter:
  if the branch is taken: counter = min(3,counter+1)
  if the branch is not taken: counter = max(0,counter-1)

- If (counter >= 2), predict taken, else predict not taken

- Advantage: a few atypical branches will not influence the prediction (a better measure of “the common case”)

- Especially useful when multiple branches share the same counter (some bits of the branch PC are used to index into the branch predictor)

- Can be easily extended to N-bits (in most processors, N=2)
Bimodal 2-Bit Predictor

Branch PC

10 bits

Table of 1K entries

Each entry is a 2-bit sat. counter

The table keeps track of the common-case outcome for the branch
Correlating Predictors

- Basic branch prediction: maintain a 2-bit saturating counter for each entry (or use 10 branch PC bits to index into one of 1024 counters) – captures the recent “common case” for each branch

- Can we take advantage of additional information?
  - If a branch recently went 01111, expect 0; if it recently went 11101, expect 1; can we have a separate counter for each case?
  - If the previous branches went 01, expect 0; if the previous branches went 11, expect 1; can we have a separate counter for each case?

Hence, build correlating predictors
Global Predictor

Branch PC

10 bits

CAT or XOR

Global history

Table of 16K entries
Each entry is a 2-bit sat. counter

The table keeps track of the common-case outcome for the branch/history combo
Local Predictor

Use 6 bits of branch PC to index into local history table

Table of 64 entries of 14-bit histories for a single branch

Also a two-level predictor that only uses local histories at the first level

Table of 16K entries of 2-bit saturating counters

14-bit history indexes into next level

Branch PC
Local Predictor

- Branch PC: 6 bits
- Local history: 10 bit entries
- XOR
- Table of 1K entries
  - Each entry is a 2-bit sat. counter

The table keeps track of the common-case outcome for the branch/local-history combo.
Local/Global Predictors

• Instead of maintaining a counter for each branch to capture the common case,

→ Maintain a counter for each branch and surrounding pattern
→ If the surrounding pattern belongs to the branch being predicted, the predictor is referred to as a local predictor
→ If the surrounding pattern includes neighboring branches, the predictor is referred to as a global predictor
Tournament Predictors

- A local predictor might work well for some branches or programs, while a global predictor might work well for others.

- Provide one of each and maintain another predictor to identify which predictor is best for each branch.

![Diagram]

Table of 2-bit saturating counters

**Local Predictor**
- 4K entries

**Global Predictor**
- 12-bit global history

**Tournament Predictor**
- 4K entries

**MUX**

**Branch PC**

**Alpha 21264:**
- 1K entries in level-1
- 1K entries in level-2
- 4K entries
- 12-bit global history
- 4K entries
- Total capacity: ?
Branch Target Prediction

• In addition to predicting the branch direction, we must also predict the branch target address

• Branch PC indexes into a predictor table; indirect branches might be problematic

• Most common indirect branch: return from a procedure – can be easily handled with a stack of return addresses
Problem 1

• What is the storage requirement for a global predictor that uses 3-bit saturating counters and that produces an index by XOR-ing 12 bits of branch PC with 12 bits of global history?
Problem 1

- What is the storage requirement for a global predictor that uses 3-bit saturating counters and that produces an index by XOR-ing 12 bits of branch PC with 12 bits of global history?

  The index is 12 bits wide, so the table has $2^{12}$ saturating counters. Each counter is 3 bits wide. So total storage
  
  \[ = 3 \times 4096 = 12 \text{ Kb} \text{ or } 1.5 \text{ KB} \]
Problem 2

• What is the storage requirement for a tournament predictor that uses the following structures:
  ▪ a “selector” that has 4K entries and 2-bit counters
  ▪ a “global” predictor that XORs 14 bits of branch PC with 14 bits of global history and uses 3-bit counters
  ▪ a “local” predictor that uses an 8-bit index into L1, and produces a 12-bit index into L2 by XOR-ing branch PC and local history. The L2 uses 2-bit counters.
Problem 2

- What is the storage requirement for a tournament predictor that uses the following structures:
  - a “selector” that has 4K entries and 2-bit counters
  - a “global” predictor that XORs 14 bits of branch PC with 14 bits of global history and uses 3-bit counters
  - a “local” predictor that uses an 8-bit index into L1, and produces a 12-bit index into L2 by XOR-ing branch PC and local history. The L2 uses 2-bit counters.

Selector = 4K * 2b = 8 Kb
Global = 3b * 2^14 = 48 Kb
Local = (12b * 2^8) + (2b * 2^12) = 3 Kb + 8 Kb = 11 Kb
Total = 67 Kb
Problem 3

- For the code snippet below, estimate the steady-state bpred accuracies for the default PC+4 prediction, the 1-bit bimodal, 2-bit bimodal, global, and local predictors. Assume that the global/local preds use 5-bit histories.

```c
int i, j, k;

do {
    for (i=0; i<4; i++) {
        increment something
    }
    for (j=0; j<8; j++) {
        increment something
    }
    k++;
} while (k < some large number)
```
Problem 3

- For the code snippet below, estimate the steady-state bpred accuracies for the default PC+4 prediction, the 1-bit bimodal, 2-bit bimodal, global, and local predictors. Assume that the global/local preds use 5-bit histories.

```c
for (i=0; i<4; i++) {
    increment something
}
for (j=0; j<8; j++) {
    increment something
}
k++;
} while (k < some large number)
```

- PC+4: \( \frac{2}{13} = 15\% \)
- 1b Bim: \( \frac{2+6+1}{4+8+1} = \frac{9}{13} = 69\% \)
- 2b Bim: \( \frac{3+7+1}{13} = \frac{11}{13} = 85\% \)
- Global: \( \frac{4+7+1}{13} = \frac{12}{13} = 92\% \)
- Local: \( \frac{4+7+1}{13} = \frac{12}{13} = 92\% \)
An Out-of-Order Processor Implementation

Branch prediction and instr fetch

Instr Fetch Queue

R1 ← R1+R2
R2 ← R1+R3
BEQZ R2
R3 ← R1+R2
R1 ← R3+R2

Decode & Rename

Instr Fetch Queue

Instr 1  Instr 2  Instr 3  Instr 4  Instr 5  Instr 6
T1      T2      T3      T4      T5      T6

Reorder Buffer (ROB)

Instr 1  Instr 2  Instr 3  Instr 4  Instr 5  Instr 6
T1      T2      T3      T4      T5      T6

Register File
R1-R32

Issue Queue (IQ)

T1 ← R1+R2
T2 ← T1+R3
BEQZ T2
T4 ← T1+T2
T5 ← T4+T2

ALU  ALU  ALU

Results written to ROB and tags broadcast to IQ

17
Problem 1

- Show the renamed version of the following code:
  Assume that you have 4 rename registers T1-T4

\[
\begin{align*}
R1 & \leftarrow R2 + R3 \\
R3 & \leftarrow R4 + R5 \\
BEQZ & \quad R1 \\
R1 & \leftarrow R1 + R3 \\
R1 & \leftarrow R1 + R3 \\
R3 & \leftarrow R1 + R3 \\
\end{align*}
\]
Problem 1

• Show the renamed version of the following code:
  Assume that you have 4 rename registers T1-T4

\[
\begin{align*}
R1 & \leftarrow R2+R3 & T1 & \leftarrow R2+R3 \\
R3 & \leftarrow R4+R5 & T2 & \leftarrow R4+R5 \\
\text{BEQZ } R1 & & \text{BEQZ } T1 \\
R1 & \leftarrow R1 + R3 & T4 & \leftarrow T1+T2 \\
R1 & \leftarrow R1 + R3 & T1 & \leftarrow T4+T2 \\
R3 & \leftarrow R1 + R3 & T2 & \leftarrow T1 +R3
\end{align*}
\]
Design Details - I

• Instructions enter the pipeline in order

• No need for branch delay slots if prediction happens in time

• Instructions leave the pipeline in order – all instructions that enter also get placed in the ROB – the process of an instruction leaving the ROB (in order) is called commit – an instruction commits only if it and all instructions before it have completed successfully (without an exception)

• To preserve precise exceptions, a result is written into the register file only when the instruction commits – until then, the result is saved in a temporary register in the ROB
Design Details - II

• Instructions get renamed and placed in the issue queue – some operands are available (T1-T6; R1-R32), while others are being produced by instructions in flight (T1-T6)

• As instructions finish, they write results into the ROB (T1-T6) and broadcast the operand tag (T1-T6) to the issue queue – instructions now know if their operands are ready

• When a ready instruction issues, it reads its operands from T1-T6 and R1-R32 and executes (out-of-order execution)

• Can you have WAW or WAR hazards? By using more names (T1-T6), name dependences can be avoided
Design Details - III

• If instr-3 raises an exception, wait until it reaches the top of the ROB – at this point, R1-R32 contain results for all instructions up to instr-3 – save registers, save PC of instr-3, and service the exception

• If branch is a mispredict, flush all instructions after the branch and start on the correct path – mispredicted instrs will not have updated registers (the branch cannot commit until it has completed and the flush happens as soon as the branch completes)

• Potential problems: ?