Lecture: Branch Prediction

- Topics: dynamic branch prediction,
  bimodal/global/local/tournament predictors
  (Chapter 3, notes on class webpage)

Software Pipelining

```
Loop:   L.D       F0, 0(R1)
        ADD.D     F4, F0, F2
        S.D       F4, 0(R1)
        DADDUI    R1, R1,# -8
        BNE       R1, R2, Loop
```

→

```
Loop:   S.D       F4, 16(R1)
        ADD.D     F4, F0, F2
        L.D       F0, 0(R1)
        DADDUI    R1, R1,# -8
        BNE       R1, R2, Loop
```

- Advantages: achieves nearly the same effect as loop unrolling, but without the code expansion – an unrolled loop may have inefficiencies at the start and end of each iteration, while a sw-pipelined loop is almost always in steady state – a sw-pipelined loop can also be unrolled to reduce loop overhead

- Disadvantages: does not reduce loop overhead, may require more registers

Problem 4

LD -> any : 1 stall
FPMUL -> any: 5 stalls
FPMUL -> ST : 4 stalls
IntALU -> BR : 1 stall

```
for (i=1000; i>0; i--)
   x[i] = y[i] * s;
```

Source code

```
Loop:    L.D        F0, 0(R1)        ; F0 = array element
        MUL.D     F4, F0, F2       ; multiply scalar
        S.D        F4, 0(R2)        ; store result
        DADDUI  R1, R1,# -8      ; decrement address pointer
        DADDUI  R2, R2,#-8       ; decrement address pointer
        BNE       R1, R3, Loop    ; branch if R1 != R3
        NOP
```

Assembly code

- Show the SW pipelined version of the code and does it cause stalls?

```
Loop:  S.D      F4, 0(R2)
       MUL     F4, F0, F2
       L.D      F0, 0(R1)
       DADDUI R2, R2, #-8
       BNE       R1, R3, Loop
       DADDUI R1, R1, #-8
```

There will be no stalls

Software Pipelining Reminders

- Note how the store instruction needs an offset in some cases

- Easiest to use more register names to avoid artificial dependences

```
LD    R1 ←                   SD    R1 →
ADD   R1 ← R1       ⇨        ADD   R1 ← R1       ✗
SD    R1 →  [ ]              LD    R1 ←

LD    R1 ←                   SD    R2 →
ADD   R2 ← R1       ⇨        ADD   R2 ← R1
SD    R2 →  [ ]              LD    R1 ←
```

Static vs. Dynamic

- Predication and speculation are other compiler techniques needed to increase performance
- To get high performance with a compiler-based approach, we need support for predication, tables to analyze dependences, etc. Plus, scheduling goes haywire if there are cache misses.
- Difficult to achieve the highest performance with a purely static (compiler-based) approach – it continues to have value for highly simple in-order processors
- For highest performance, dynamic/hardware approaches are most effective, and the compiler can help such processors too

Amdahl’s Law

- Architecture design is very bottleneck-driven – make the common case fast, do not waste resources on a component that has little impact on overall performance/power

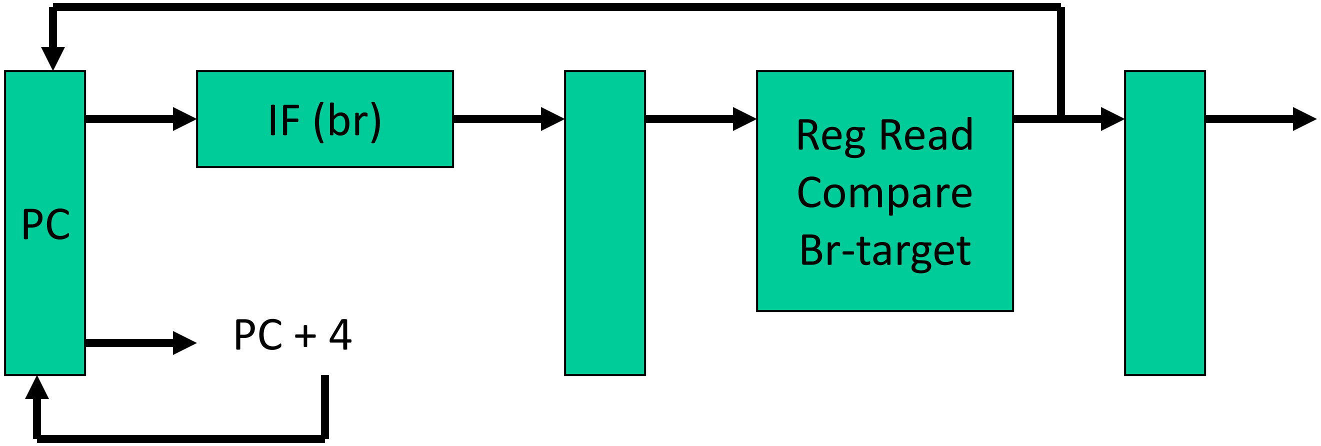
- Amdahl’s Law: performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play

- Example: a web server spends 40% of time in the CPU and 60% of time doing I/O – a new processor that is ten times faster results in a 36% reduction in execution time (speedup of 1.56) – Amdahl’s Law states that maximum execution time reduction is 40% (max speedup of 1.66)

Principle of Locality

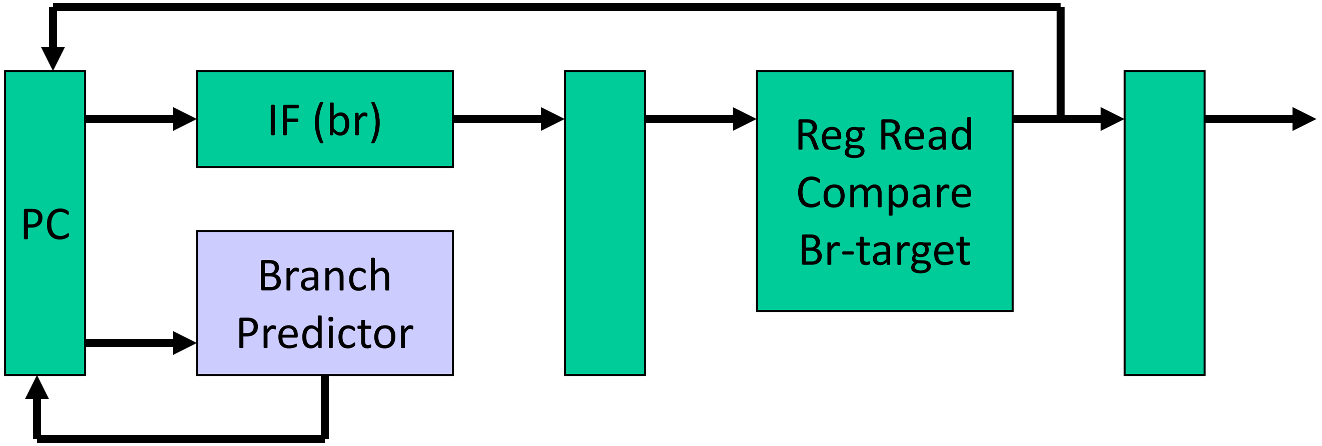
- Most programs are predictable in terms of instructions executed and data accessed
- The 90-10 Rule: a program spends 90% of its execution time in only 10% of the code
- Temporal locality: a program will shortly re-visit X
- Spatial locality: a program will shortly visit X+1

Pipeline without Branch Predictor



In the 5-stage pipeline, a branch completes in two cycles →
If the branch went the wrong way, one incorrect instr is fetched →
One stall cycle per incorrect branch

Pipeline with Branch Predictor



In the 5-stage pipeline, a branch completes in two cycles →
If the branch went the wrong way, one incorrect instr is fetched →
One stall cycle per incorrect branch

1-Bit Bimodal Prediction

- For each branch, keep track of what happened last time and use that outcome as the prediction
- What are prediction accuracies for branches 1 and 2 below:

```
while (1) {
      for (i=0;i<10;i++) {                    branch-1
          …
      }
      for (j=0;j<20;j++) {                    branch-2
         …
      }
}
```

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)

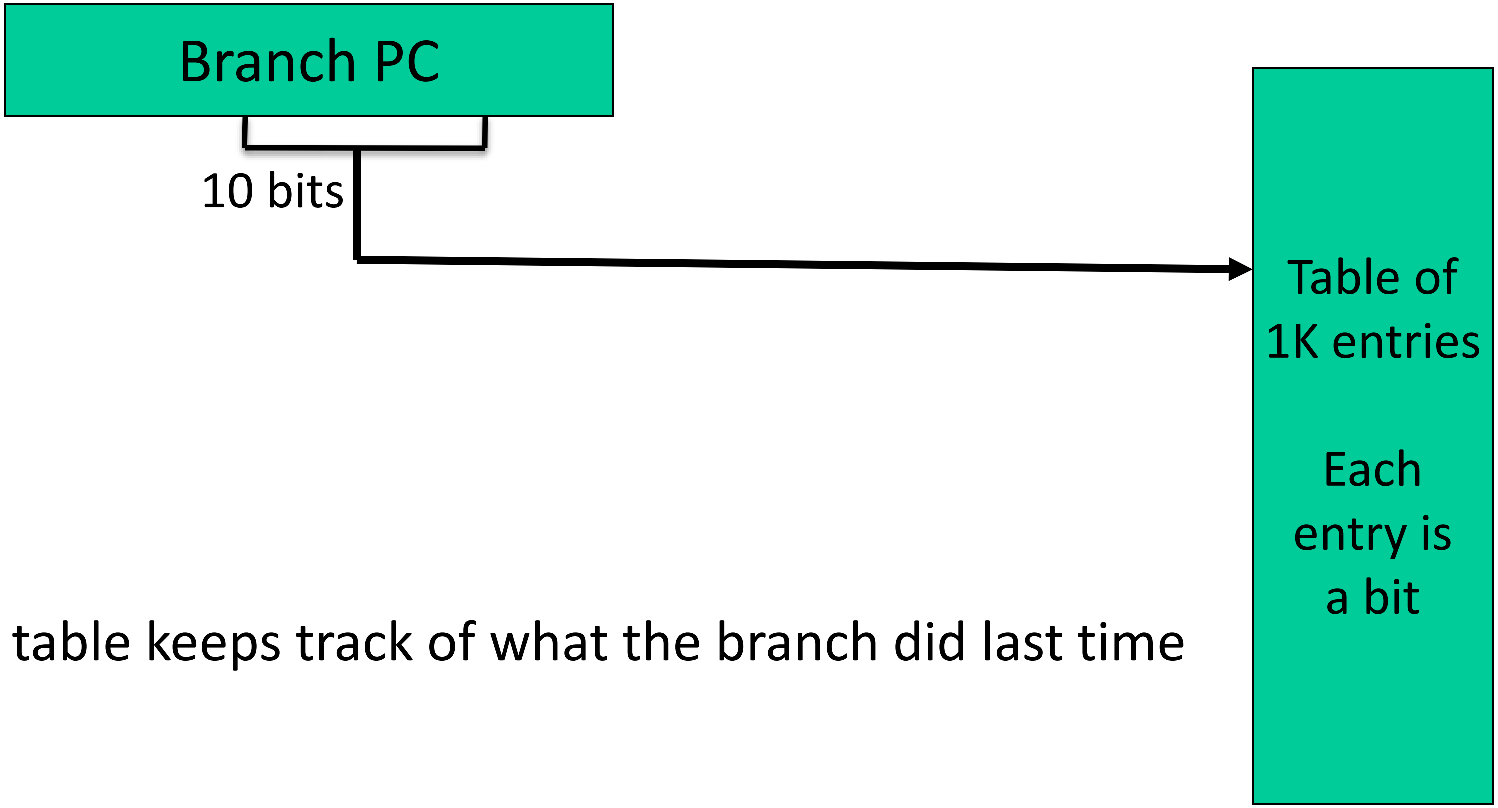
- If (counter >= 2), predict taken, else predict not taken

- 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)

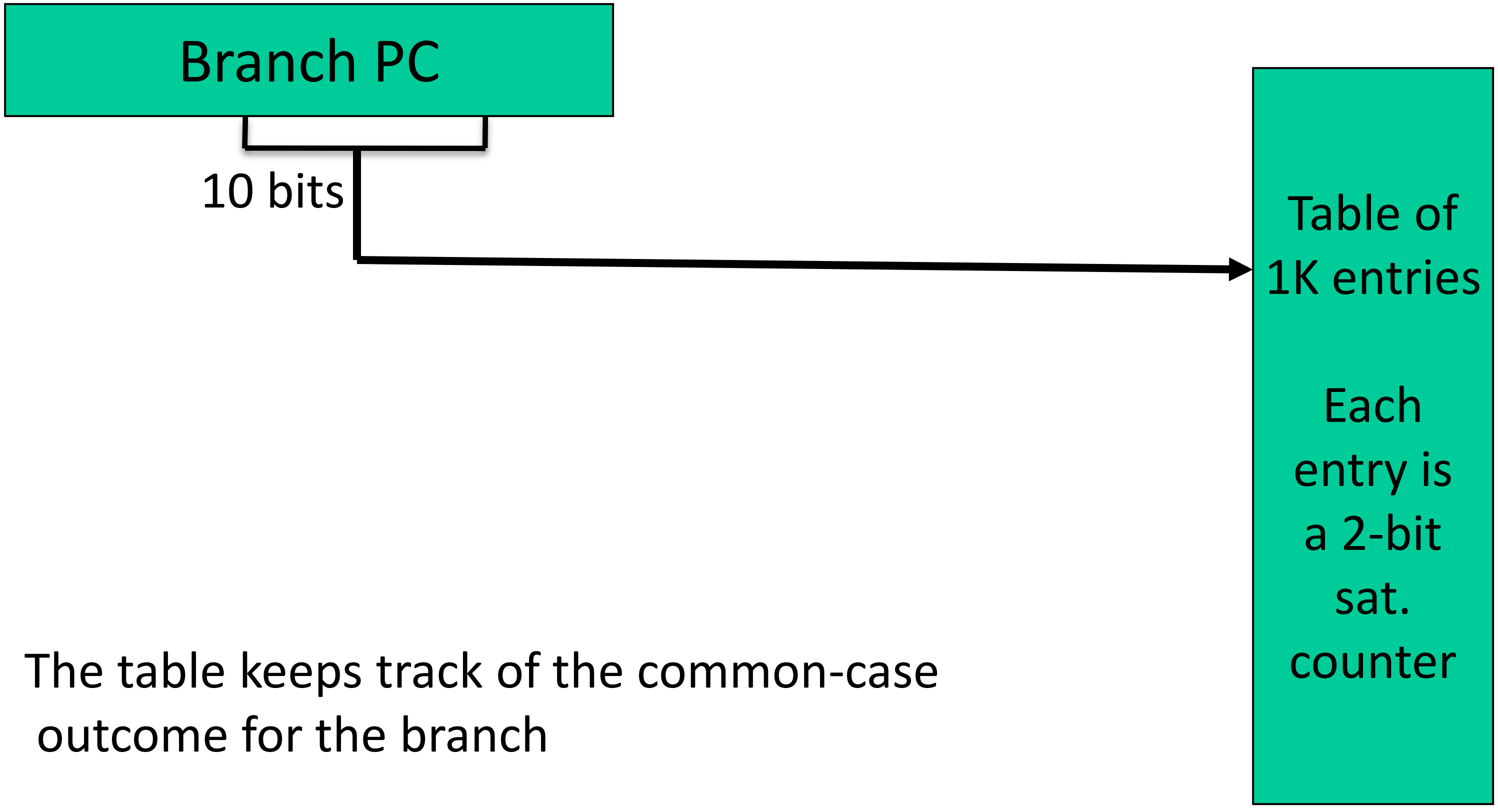
- Can be easily extended to N-bits (in most processors, N=2)

Bimodal 1-Bit Predictor



The table keeps track of what the branch did last time

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

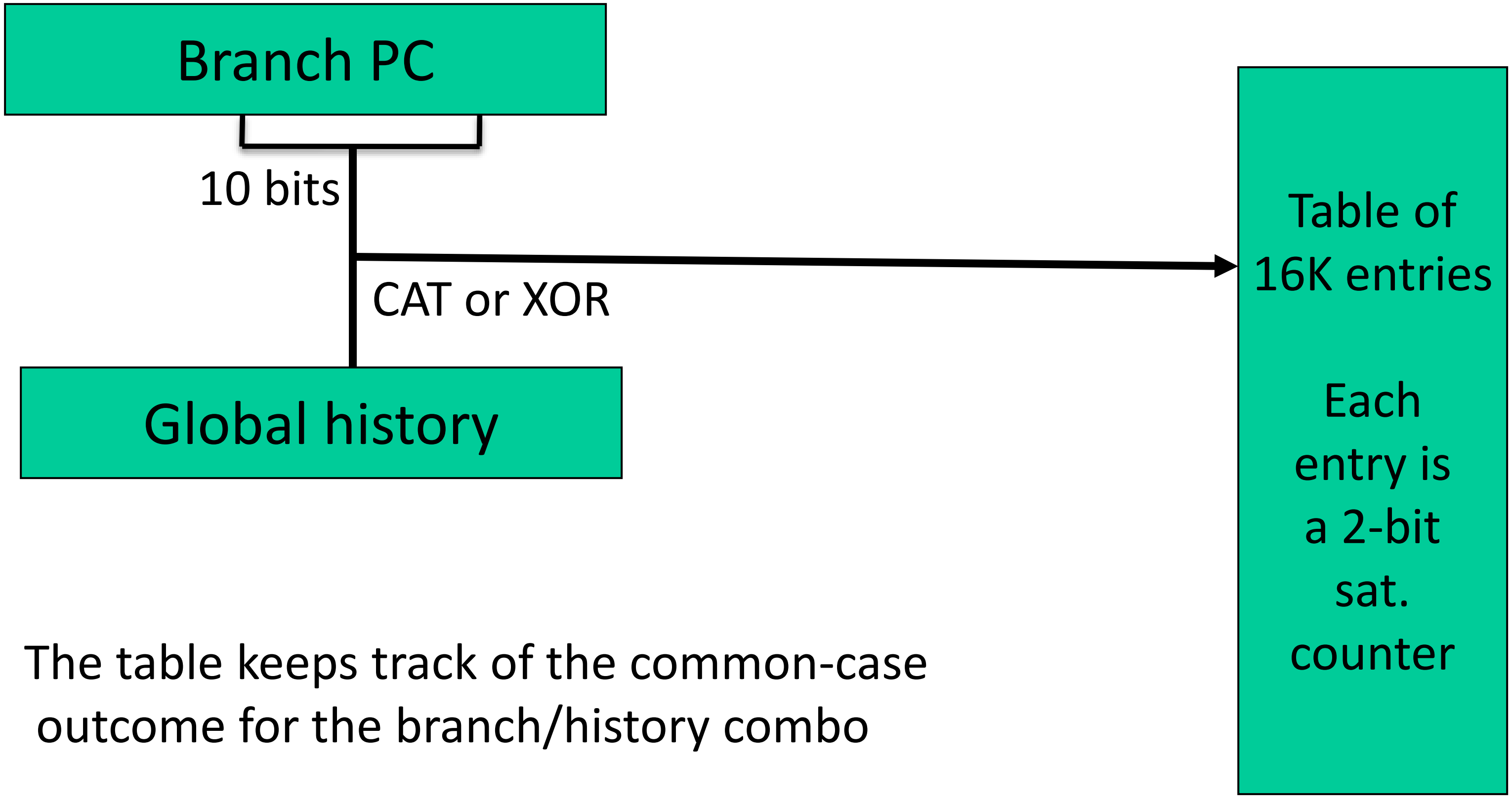
Correlating Predictors

- 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

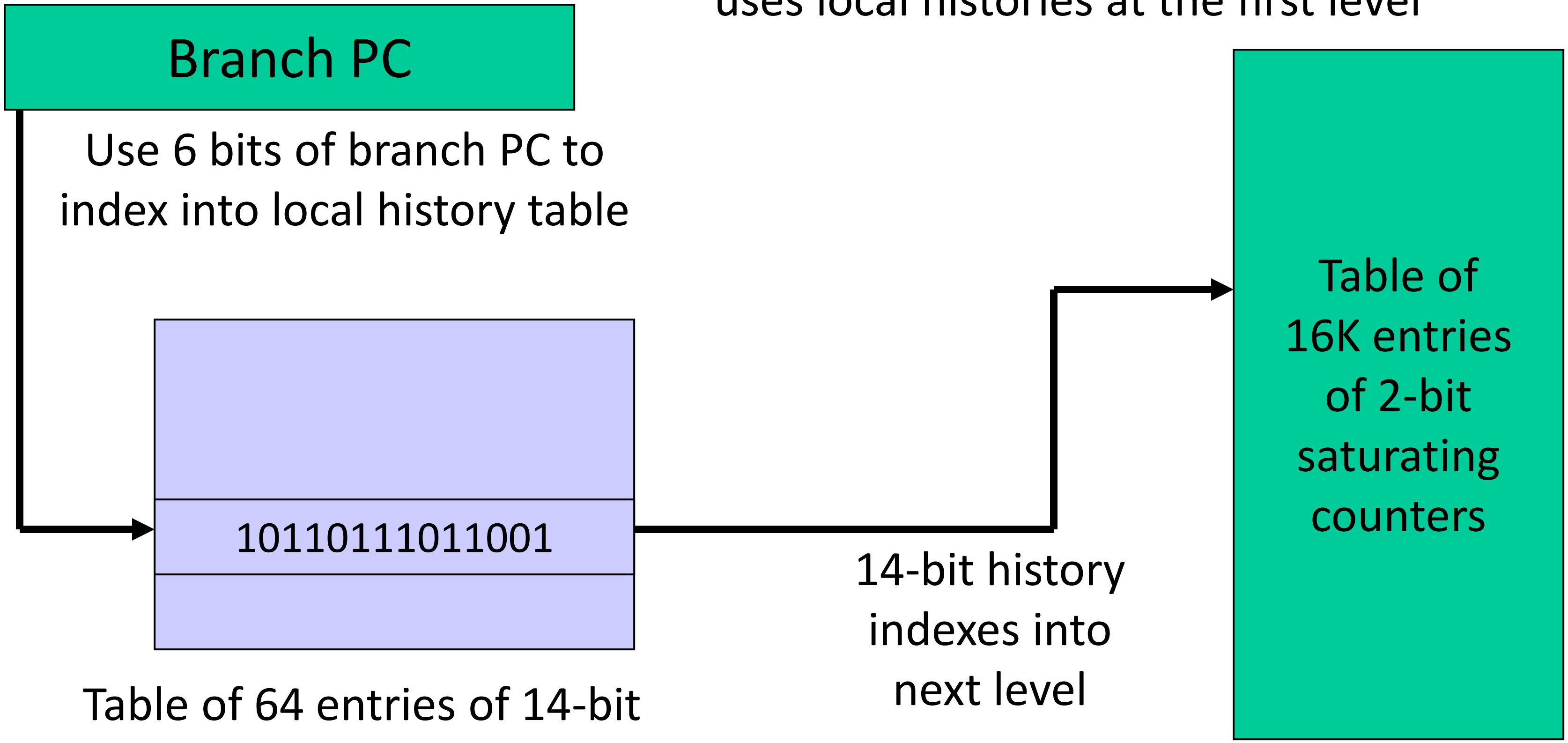
Global Predictor



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

Local Predictor

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



Branch PC

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

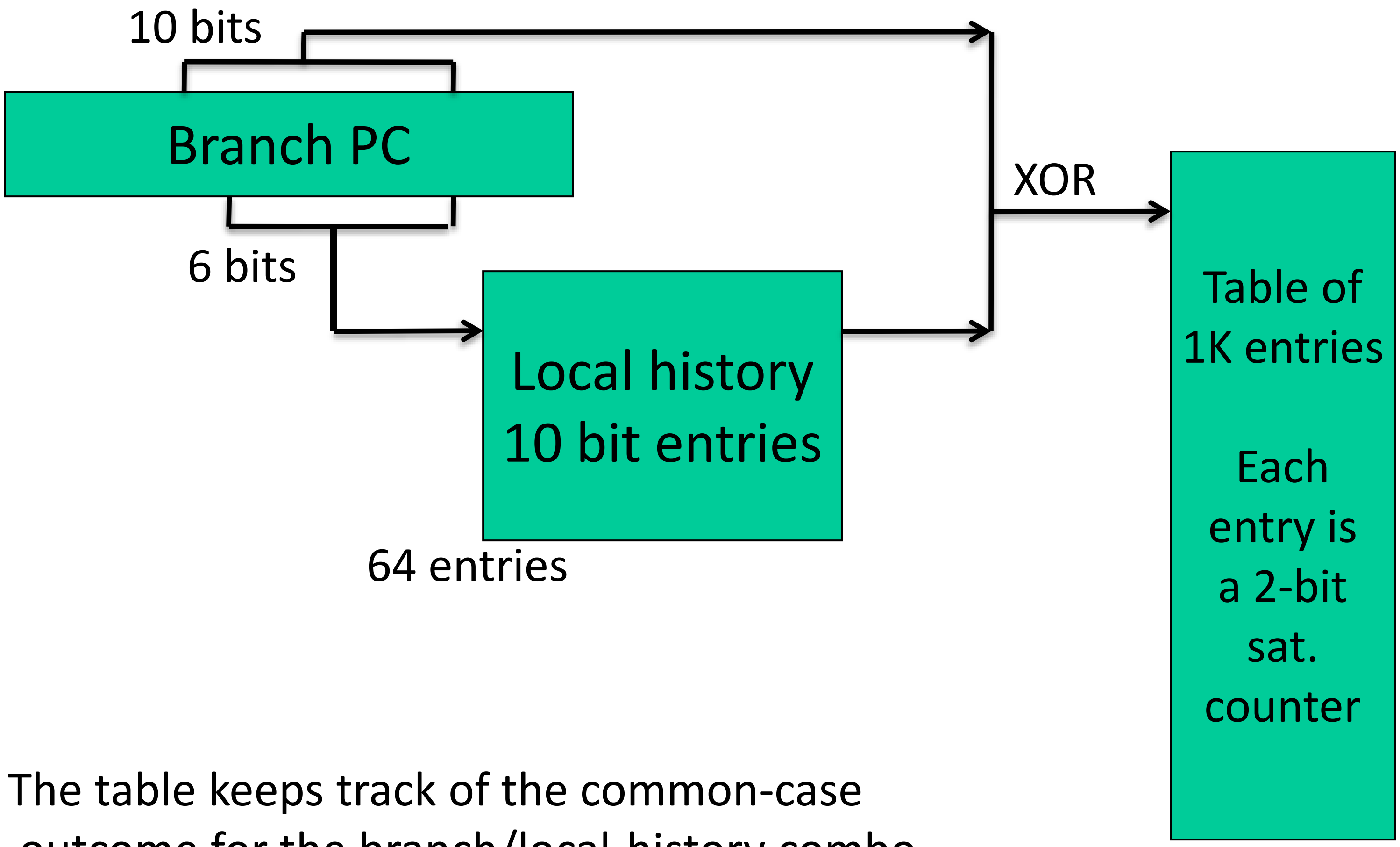
10110111011001

14-bit history indexes into next level

Table of 16K entries of 2-bit saturating counters

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

Local Predictor



10 bits

Branch PC

6 bits

Local history
10 bit entries

64 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

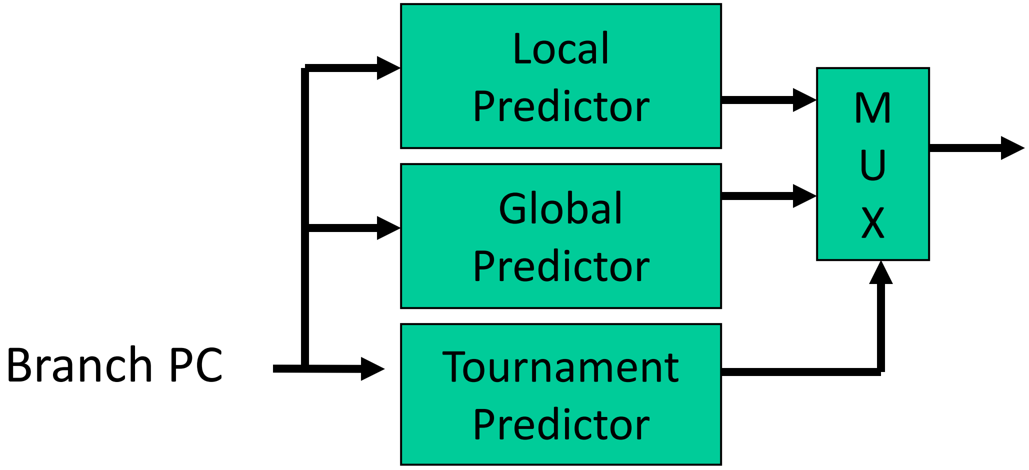


Table of 2-bit saturating counters

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 * 4096 = 12 Kb or 1.5 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.

```
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.

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

PC+4: 2/13 = 15%

1b Bim: (2+6+1)/(4+8+1)
= 9/13 = 69%

2b Bim: (3+7+1)/13
= 11/13 = 85%

Global: (4+7+1)/13
= 12/13 = 92%

Local: (4+7+1)/13
= 12/13 = 92%