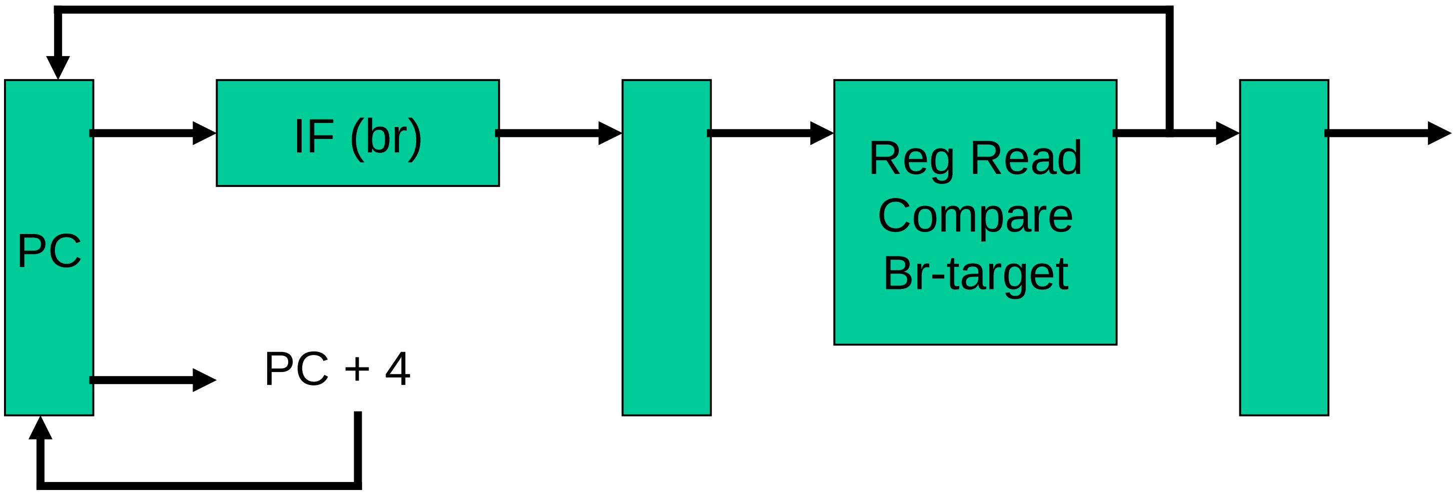
250P: Computer Systems Architecture

Lecture 8: Dynamic ILP Branch prediction

Anton Burtsev
October, 2021

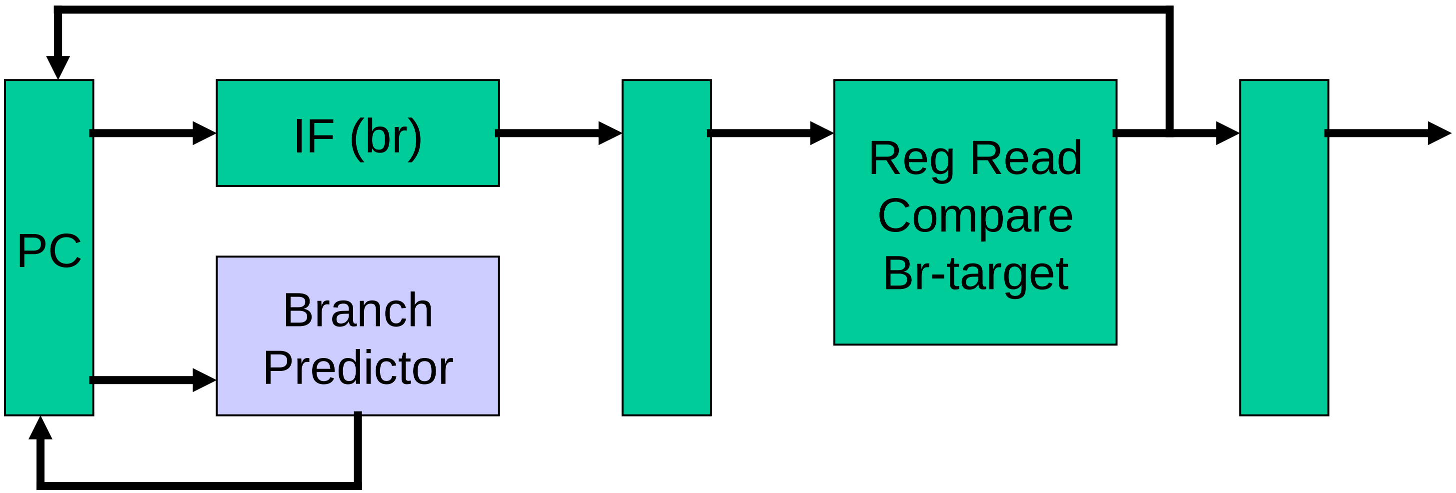
Branch prediction

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)

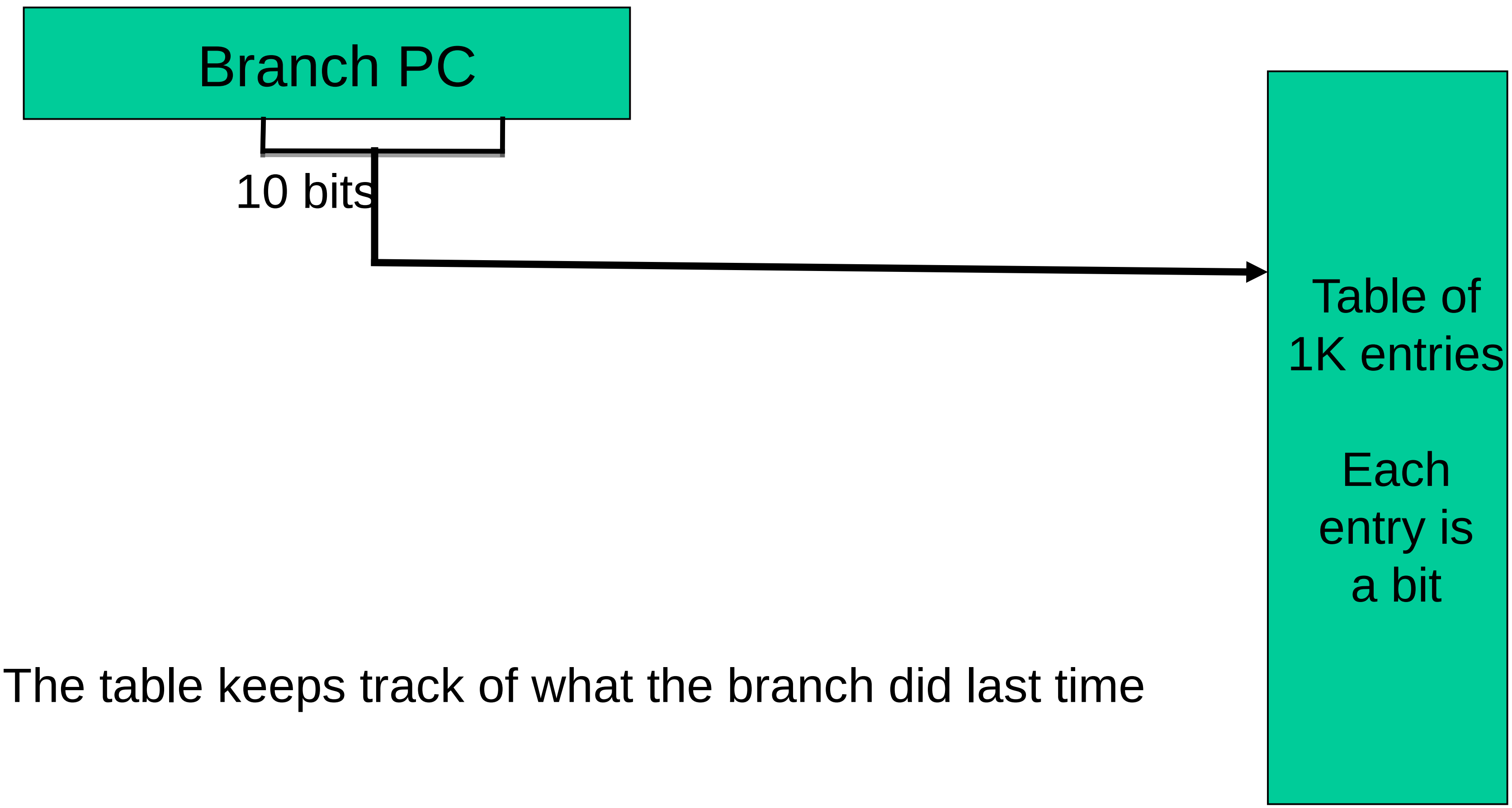
- If (counter >= 2), predict taken, else predict not taken

- 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

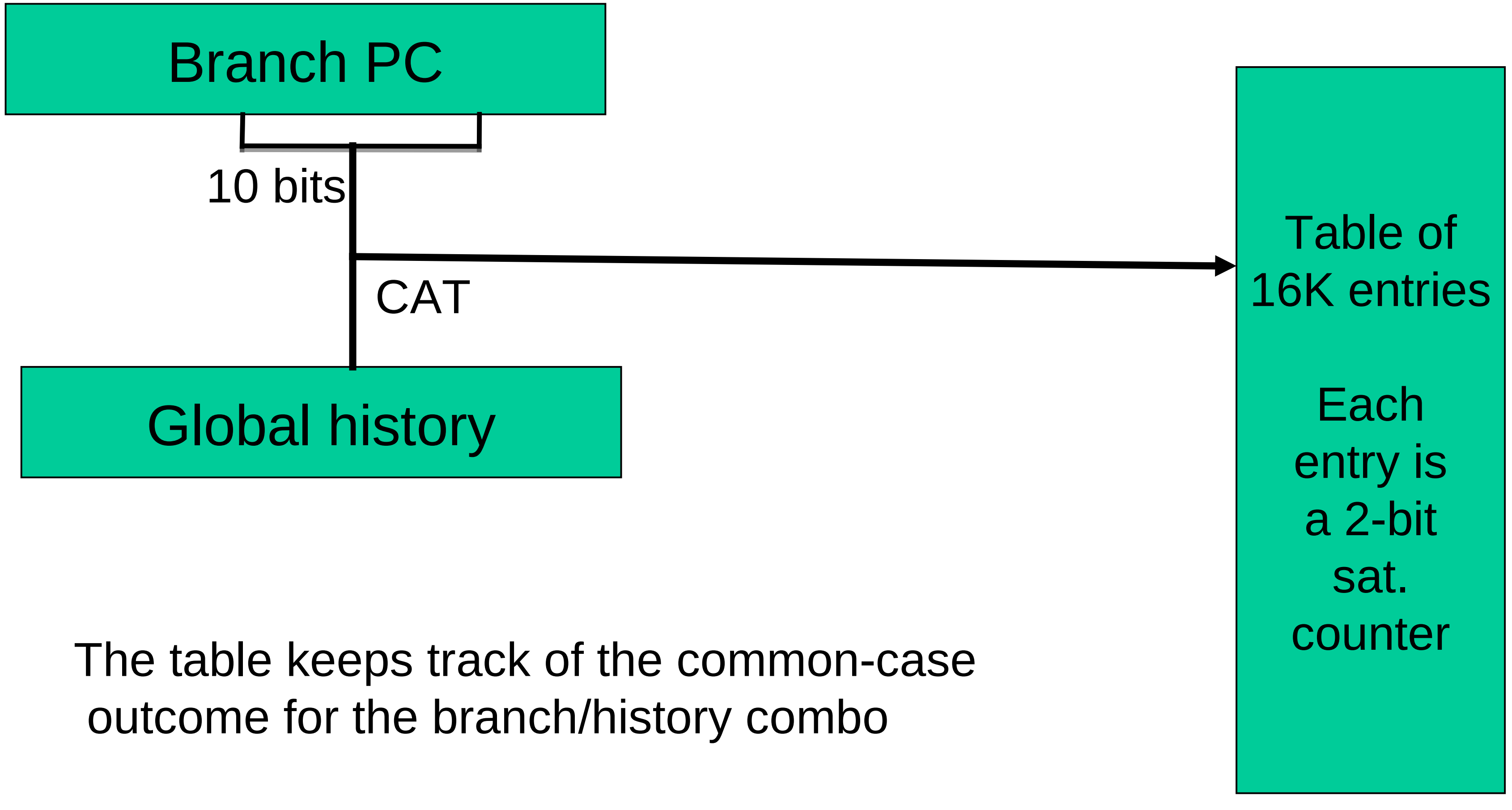
Correlating Predictors

- 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

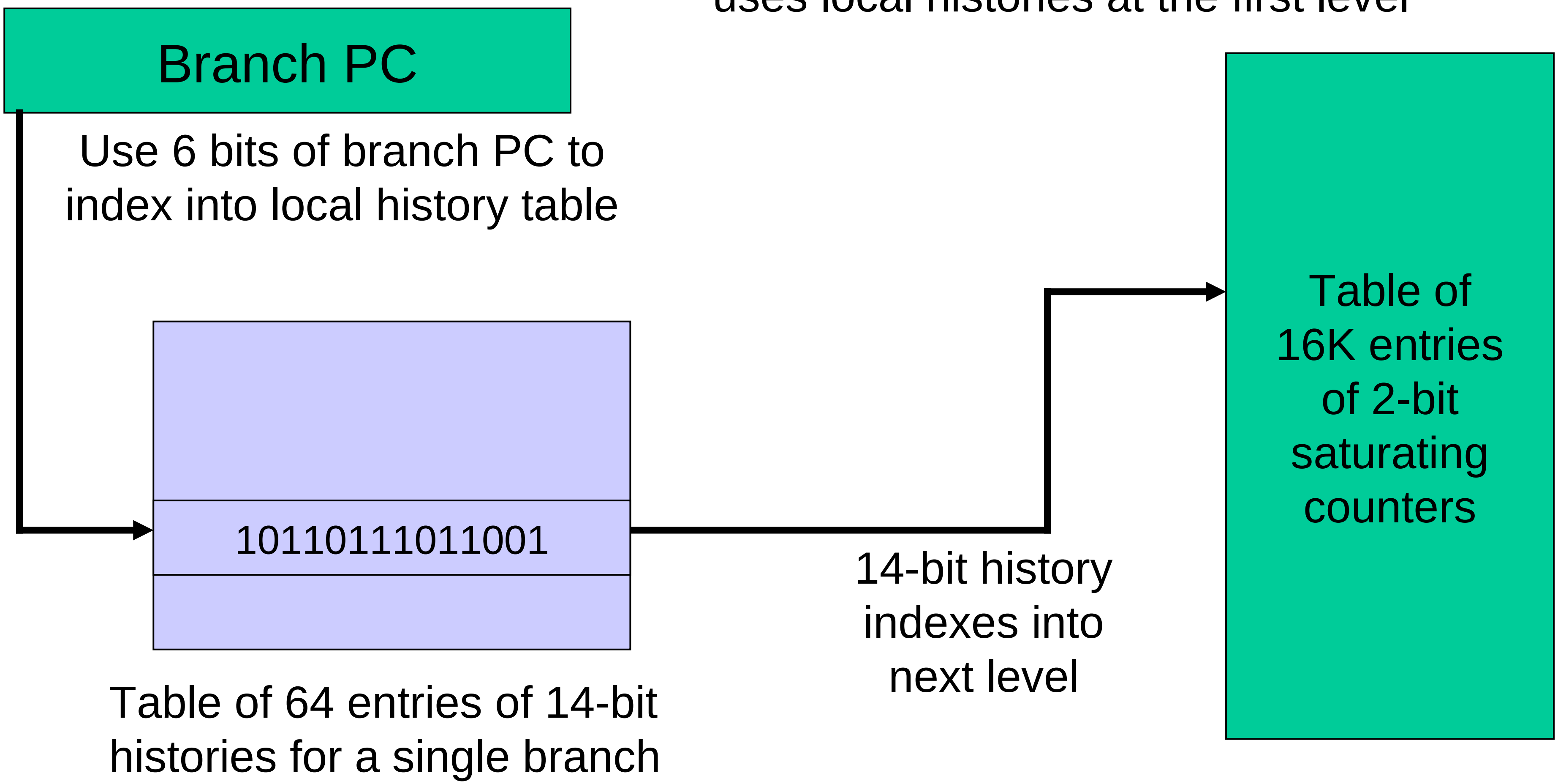
Global Predictor



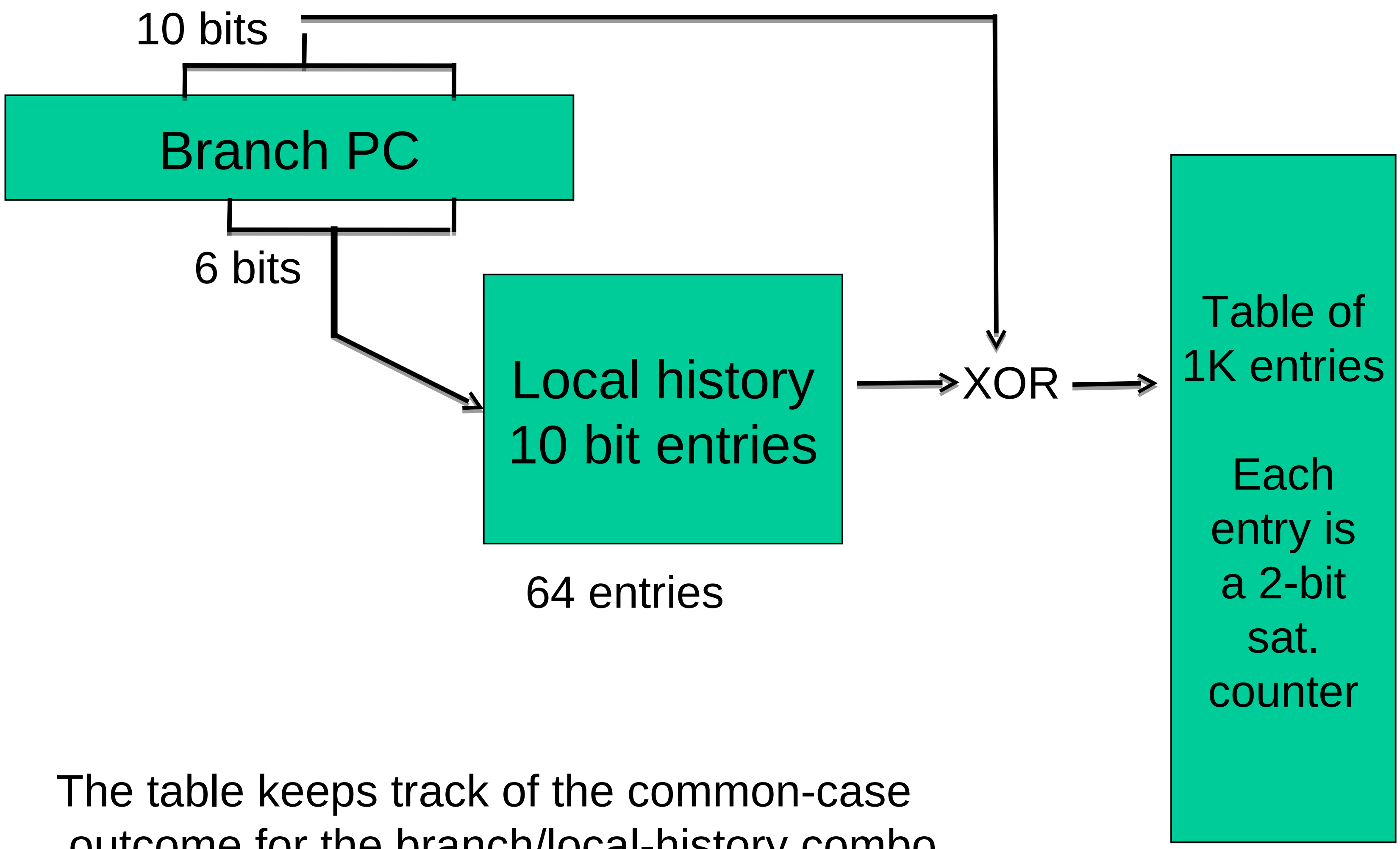
The table keeps track of the common-case outcome for the branch/history combo

Local Predictor

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



Local Predictor



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

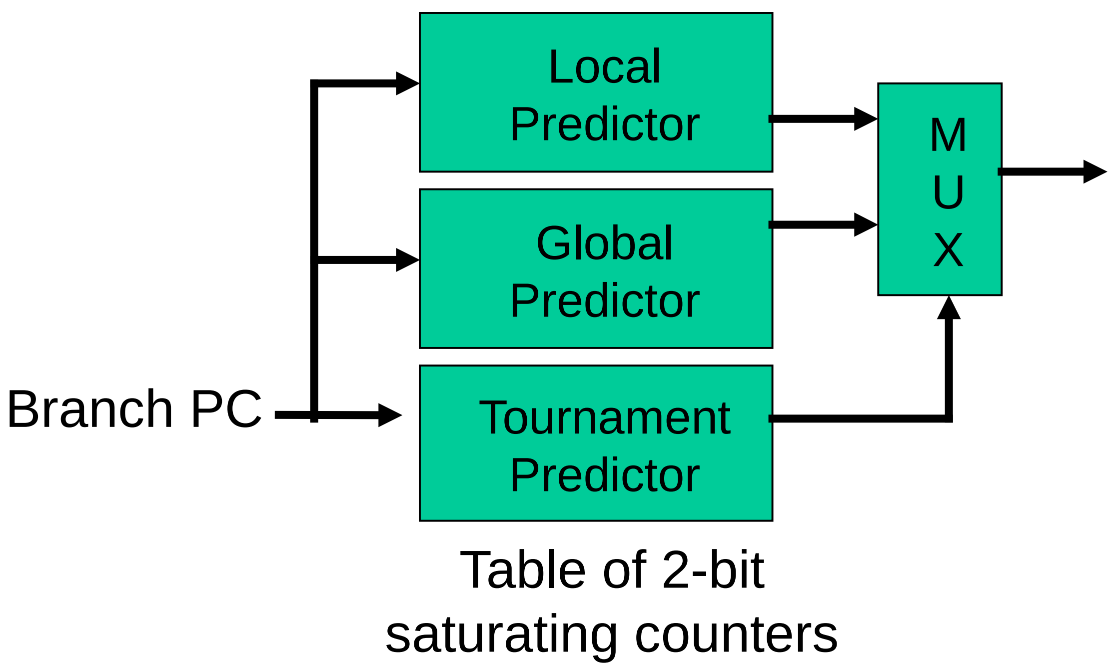
Local/Global Predictors

- 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



Alpha 21264:
1K entries in level-1
1K entries in level-2

4K entries
12-bit global history

4K entries

Total capacity: ?

Predication

- A branch within a loop can be problematic to schedule
- Control dependences are a problem because of the need to re-fetch on a mispredict
- For short loop bodies, control dependences can be converted to data dependences by using predicated/conditional instructions

Predicated or Conditional Instructions

```
if (R1 == 0)
  R2 = R2 + R4
else
  R6 = R3 + R5
  R4 = R2 + R3
```

→

```
R7 = !R1
R2 = R2 + R4   (predicated on R7)
R6 = R3 + R5   (predicated on R1)
R4 = R8 + R3   (predicated on R1)
```

Predicated or Conditional Instructions

- The instruction has an additional operand that determines whether the instr completes or gets converted into a no-op

- Example: lwc R1, 0(R2), R3   (load-word-conditional) will load the word at address (R2) into R1 if R3 is non-zero; if R3 is zero, the instruction becomes a no-op

- Replaces a control dependence with a data dependence (branches disappear) ; may need register copies for the condition or for values used by both directions

```
if (R1 == 0)
   R2 = R2 + R4
else
   R6 = R3 + R5
   R4 = R2 + R3
```

→

```
R7 = !R1 ;
R2 = R2 + R4   (predicated on R7)
R6 = R3 + R5   (predicated on R1)
R4 = R8 + R3   (predicated on R1)
```

Thank you!