Lecture: Static ILP, Branch Prediction

- Topics: compiler-based ILP extraction, branch prediction,
  bimodal/global/local/tournament predictors
  (Section 3.3, notes on class webpage)

Problem 1

- Use predication to remove control hazards in this code

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

Problem 1

- Use predication to remove control hazards in this code

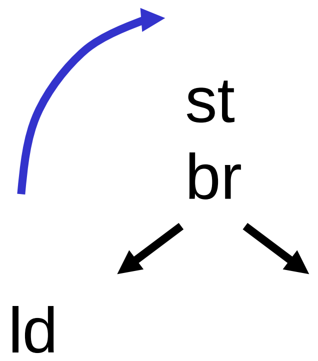
```
if (R1 == 0)
   R2 = R5 + R4
   R3 = R2 + R4
else
   R6 = R3 + R2
```

→

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

Support for Speculation

- When re-ordering instructions, we need hardware support
  - to ensure that an exception is raised at the correct point
  - to ensure that we do not violate memory dependences



Detecting Exceptions

- Some exceptions require that the program be terminated (memory protection violation), while other exceptions require execution to resume (page faults)
- For a speculative instruction, in the latter case, servicing the exception only implies potential performance loss
- In the former case, you want to defer servicing the exception until you are sure the instruction is not speculative
- Note that a speculative instruction needs a special opcode to indicate that it is speculative

Program-Terminate Exceptions

- When a speculative instruction experiences an exception, instead of servicing it, it writes a special NotAThing value (NAT) in the destination register

- If a non-speculative instruction reads a NAT, it flags the exception and the program terminates (it may not be desireable that the error is caused by an array access, but the segfault happens two procedures later)

- Alternatively, an instruction (the *sentinel*) in the speculative instruction’s original location checks the register value and initiates recovery

Memory Dependence Detection

- If a load is moved before a preceding store, we must ensure that the store writes to a non-conflicting address, else, the load has to re-execute

- When the speculative load issues, it stores its address in a table (Advanced Load Address Table in the IA-64)

- If a store finds its address in the ALAT, it indicates that a violation occurred for that address

- A special instruction (the *sentinel*) in the load's original location checks to see if the address had a violation and re-executes the load if necessary

Problem 2

- For the example code snippet below, show the code after the load is hoisted:

```
Instr-A
Instr-B
ST  R2 → [R3]
Instr-C
BEZ R7, foo
Instr-D
LD R8 ← [R4]
Instr-E
```

Problem 2

- For the example code snippet below, show the code after the load is hoisted:

	LD.S R8 ← [R4]
Instr-A	Instr-A
Instr-B	Instr-B
ST R2 → [R3]	ST R2 → [R3]
Instr-C	Instr-C
BEZ R7, foo	BEZ R7, foo
Instr-D	Instr-D
LD R8 ← [R4]	LD.C R8, rec-code
Instr-E	Instr-E

rec-code: LD R8 ← [R4]

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

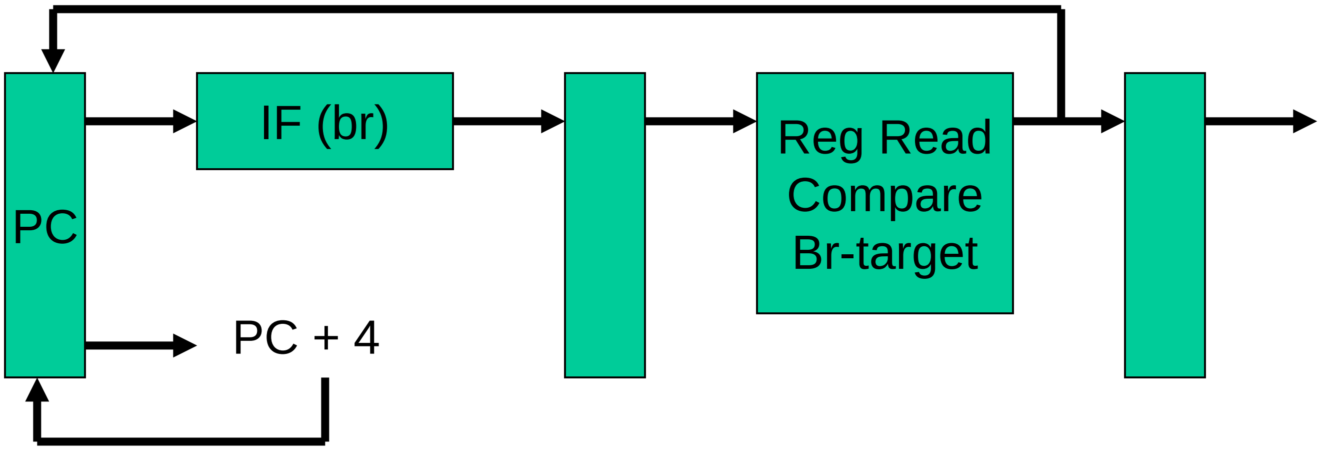
- Amdahl's Law: performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play

- 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

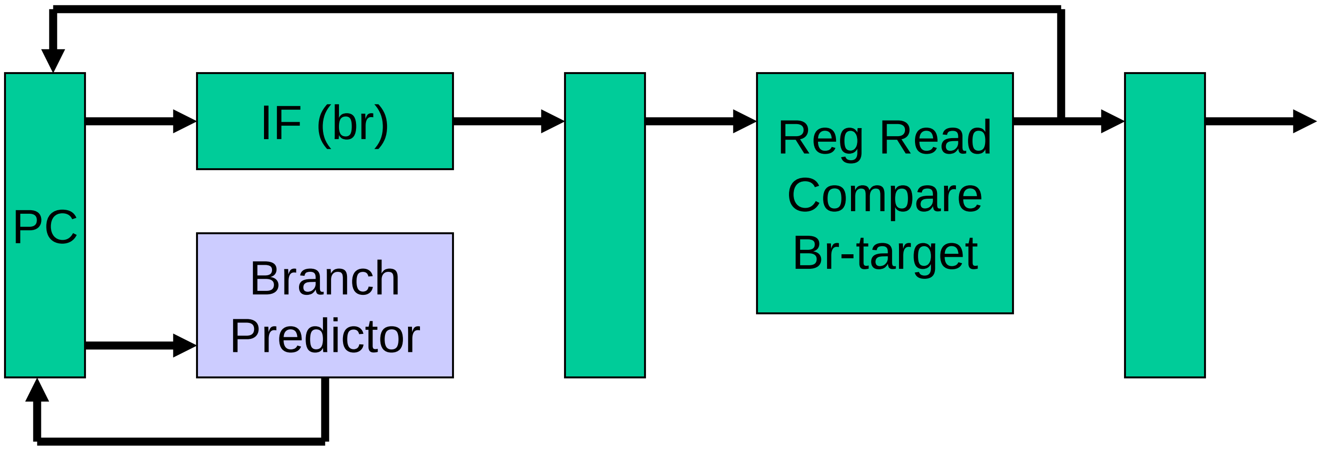
- Most programs are predictable in terms of instructions executed and data accessed
- The 90-10 Rule: a program spends 90% of its execution time in only 10% of the code
- Temporal locality: a program will shortly re-visit X
- Spatial locality: a program will shortly visit X+1

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)

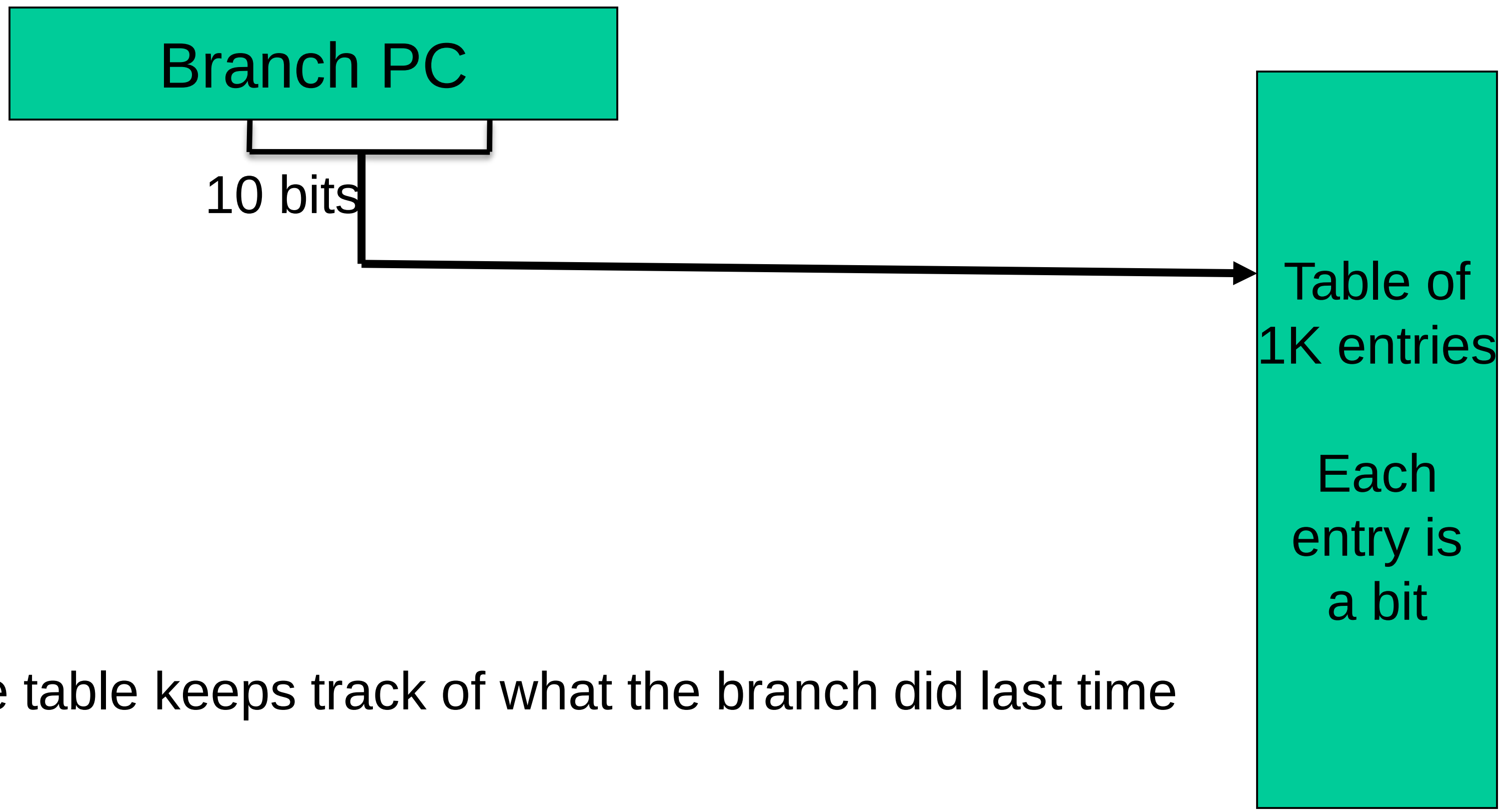
- If (counter >= 2), predict taken, else predict not taken

- 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)

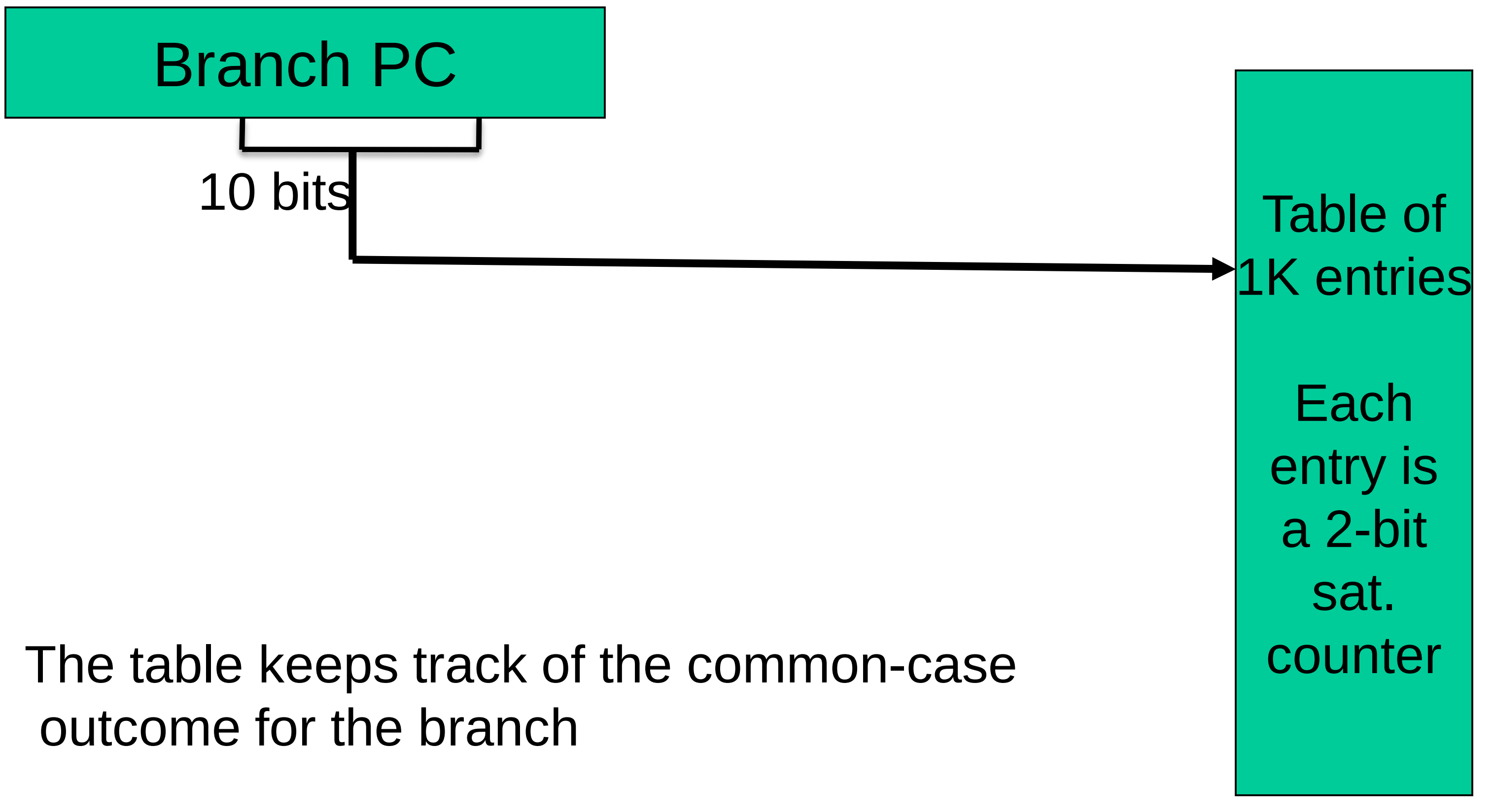
- Can be easily extended to N-bits (in most processors, N=2)

Bimodal 1-Bit Predictor



The table keeps track of what the branch did last time

Bimodal 2-Bit Predictor



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

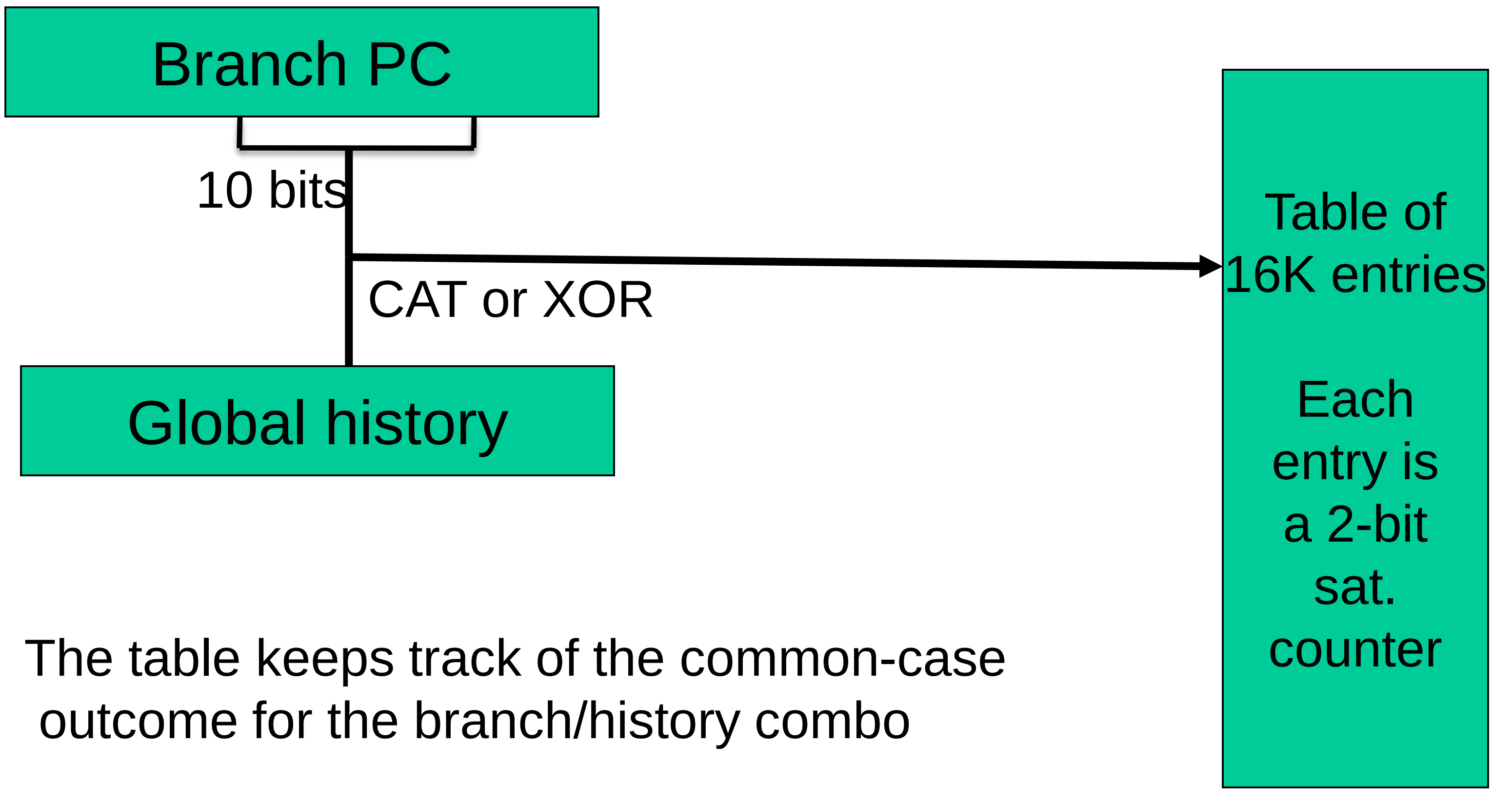
Correlating Predictors

- 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

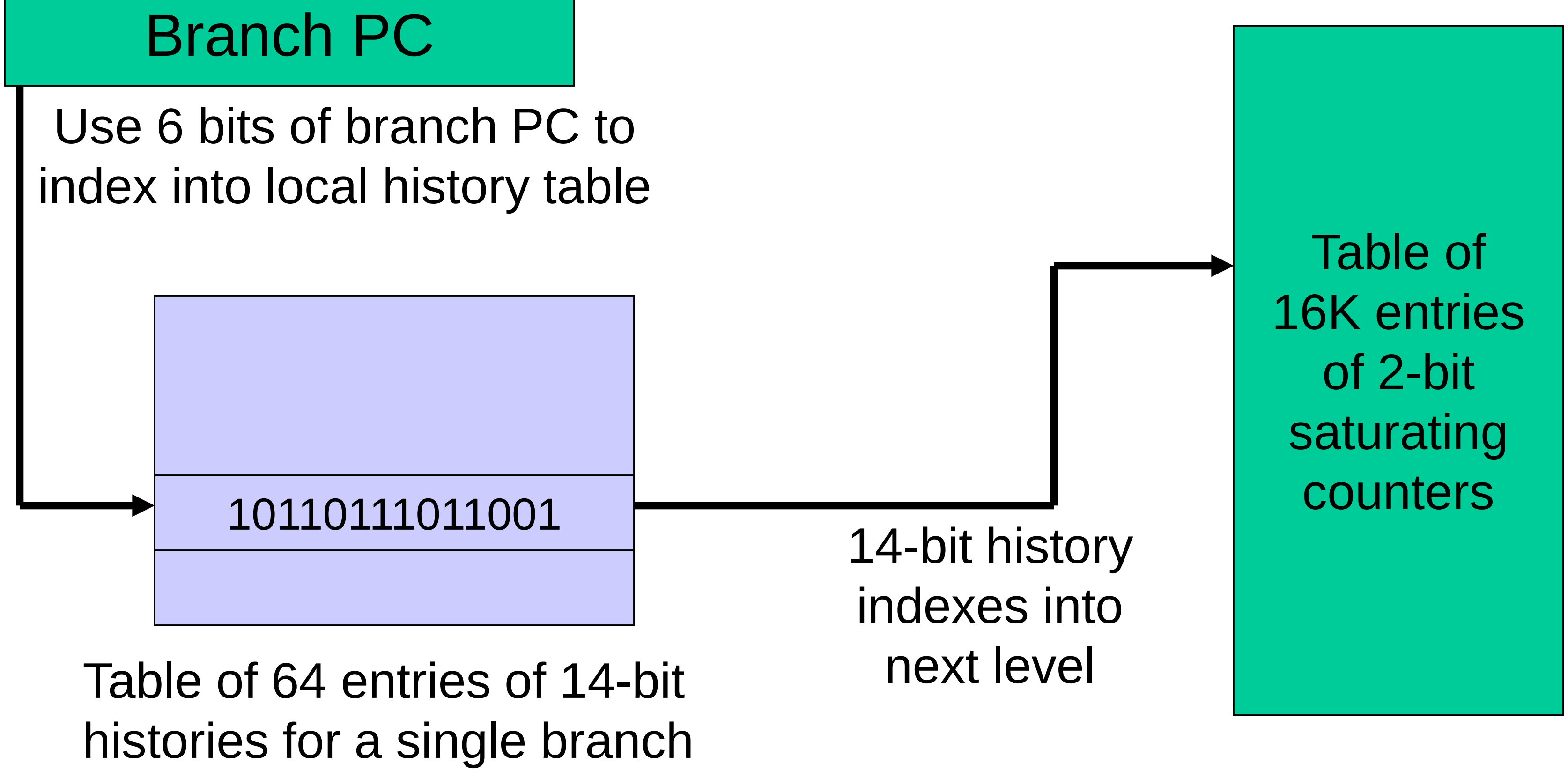
Global Predictor



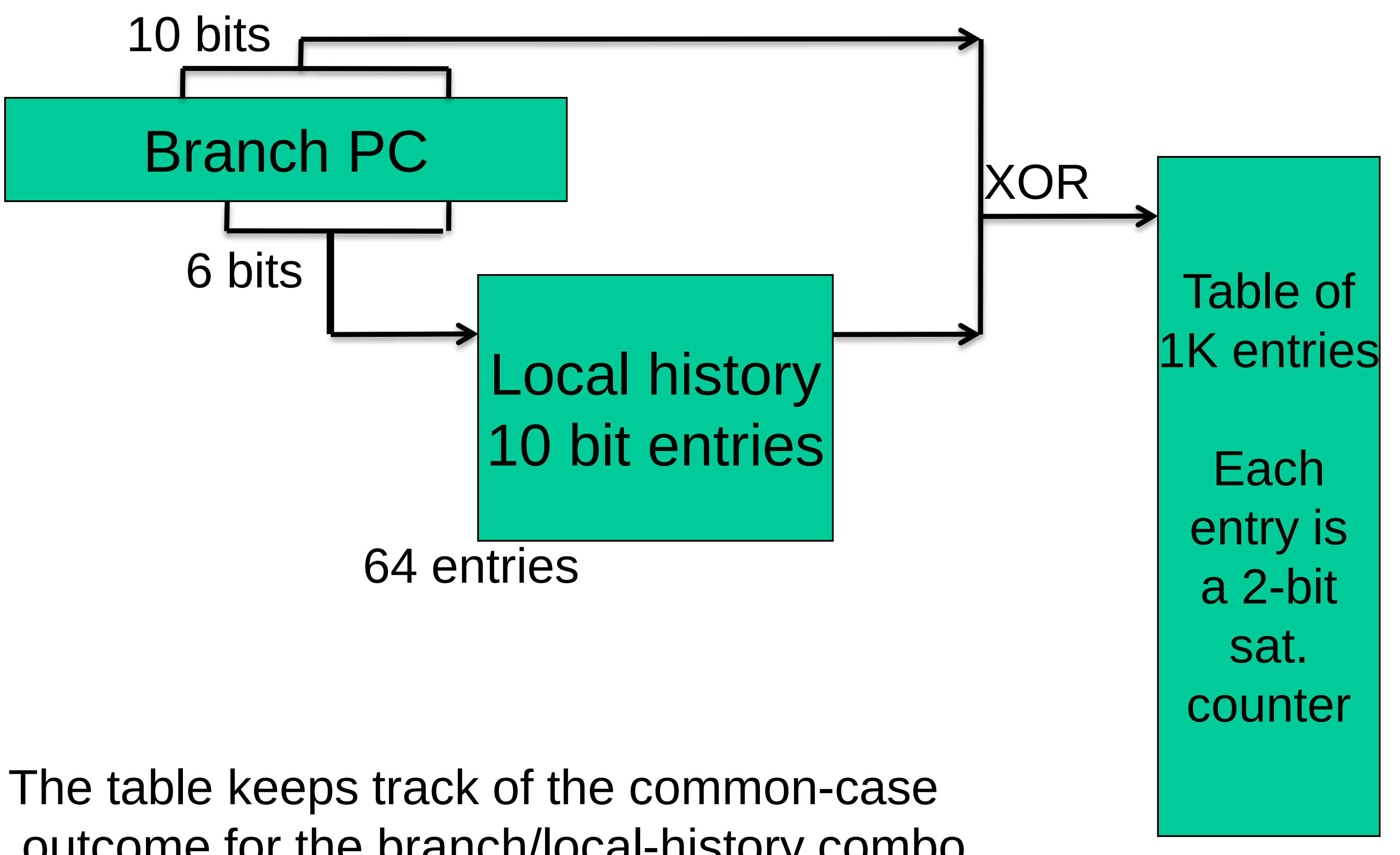
The table keeps track of the common-case outcome for the branch/history combo

Local Predictor

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



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

Title

- Bullet