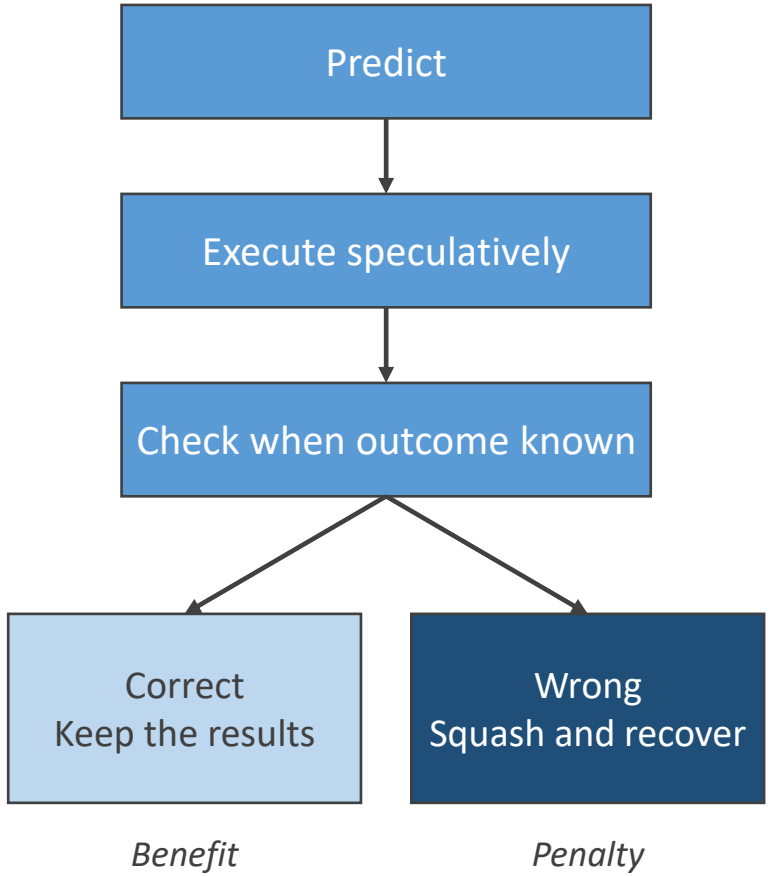
Prediction

Advanced Topics in Computer Architecture

Timothy Jones

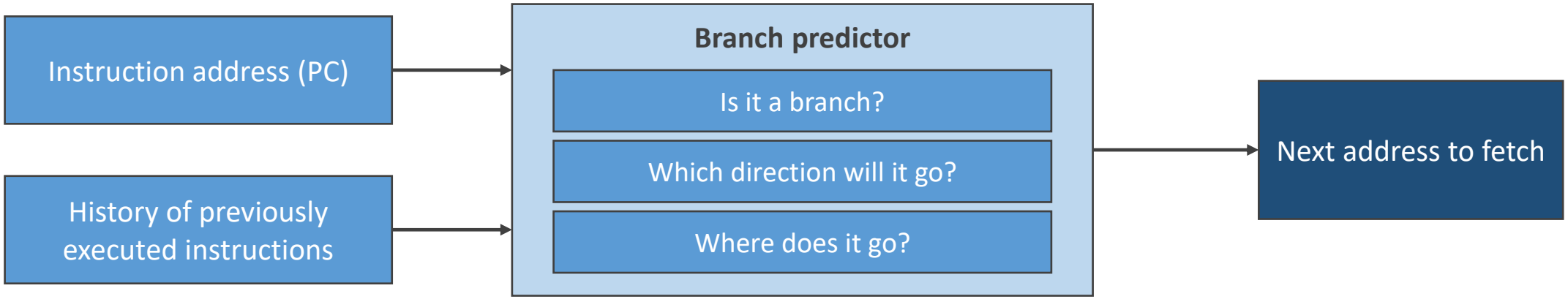
The challenge

- Microprocessors extract ILP to maintain high performance
- Yet the ability to do this is often restricted by dependences between instructions
  - You don’t know which path a conditional branch will follow
  - You don’t know whether a load aliases with an earlier store
- To obtain high performance, we need to make predictions about the future
  - And check afterwards when we know
- When correct, we get the benefits of executing instructions earlier
- When wrong, we pay the penalty of fixing up the processor’s internal state



Branch prediction

- We need to be able to fetch along a path after a branch
  - Otherwise we stall and incur a pipeline bubble
- But we don’t know which direction the branch goes
- Actually we don’t even know it is a branch
- All we have are the instruction’s address and information about instructions that have previously been executed



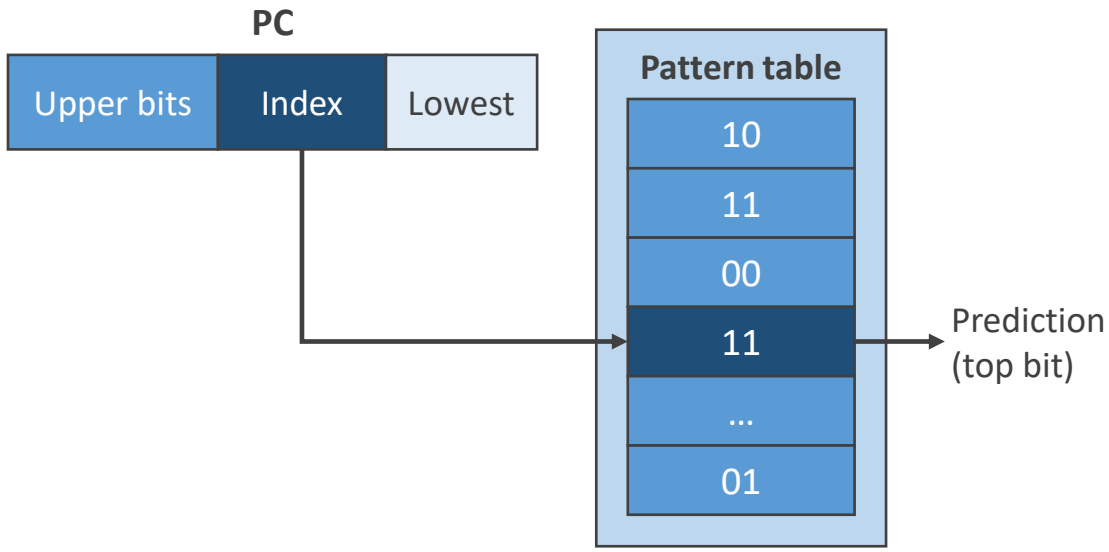
Solutions

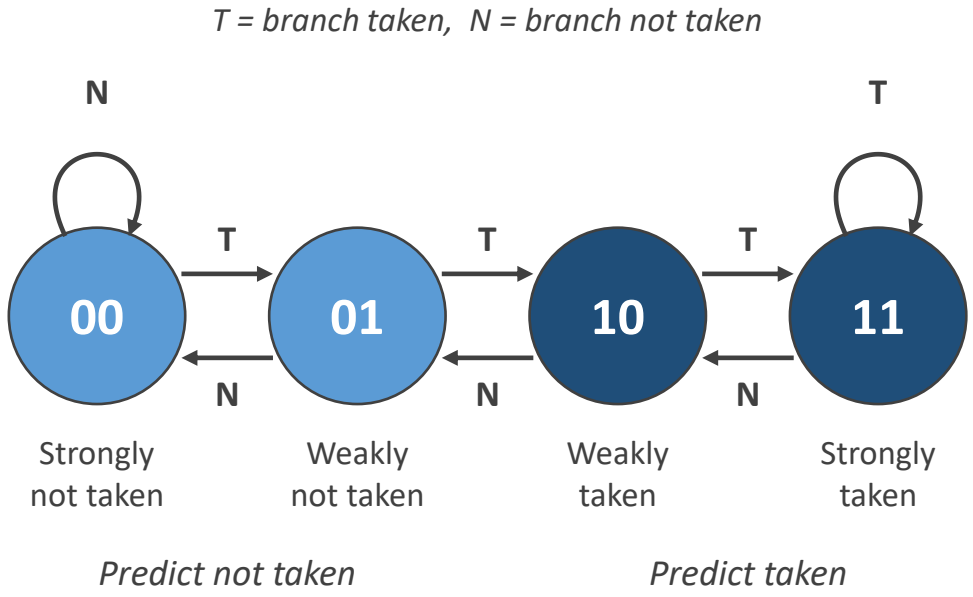
- Waiting is not an option
  - With a branch every ~6 instructions, waiting severely limits the ability to extract ILP
  - We need to break these control dependences
- We could make a decision statically about a branch (e.g. backward taken, forward not-taken)
- We could avoid it with branch delay slots (i.e. let the compiler find independent instructions)
- Or we could dynamically predict the branch's behaviour

Static	Delay slots	Dynamic
Fixed decision for each branch	Instructions after the branch always execute, hiding (some of) the latency	Hardware learns from each branch's past behaviour as the program runs

A simple dynamic branch predictor

- A table indexed by the lower bits of the PC (but typically not the very lowest bits)
  - Each entry is a two-bit saturating counter
- Predicts the last direction of this branch
  - But tolerates one isolated outcome that differs from the norm (useful at the end of loops)



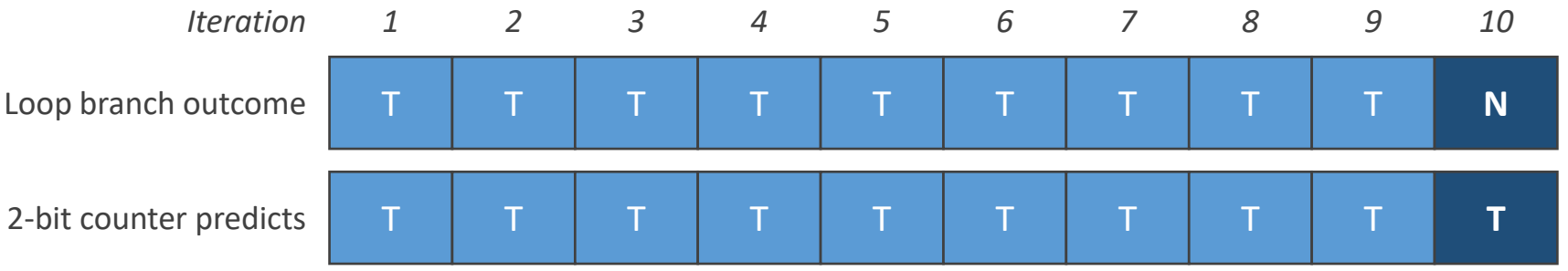


Using history

- Simple predictors work reasonably well but we can do better
- Consider a loop such as

```
for i in 1..10 do
```

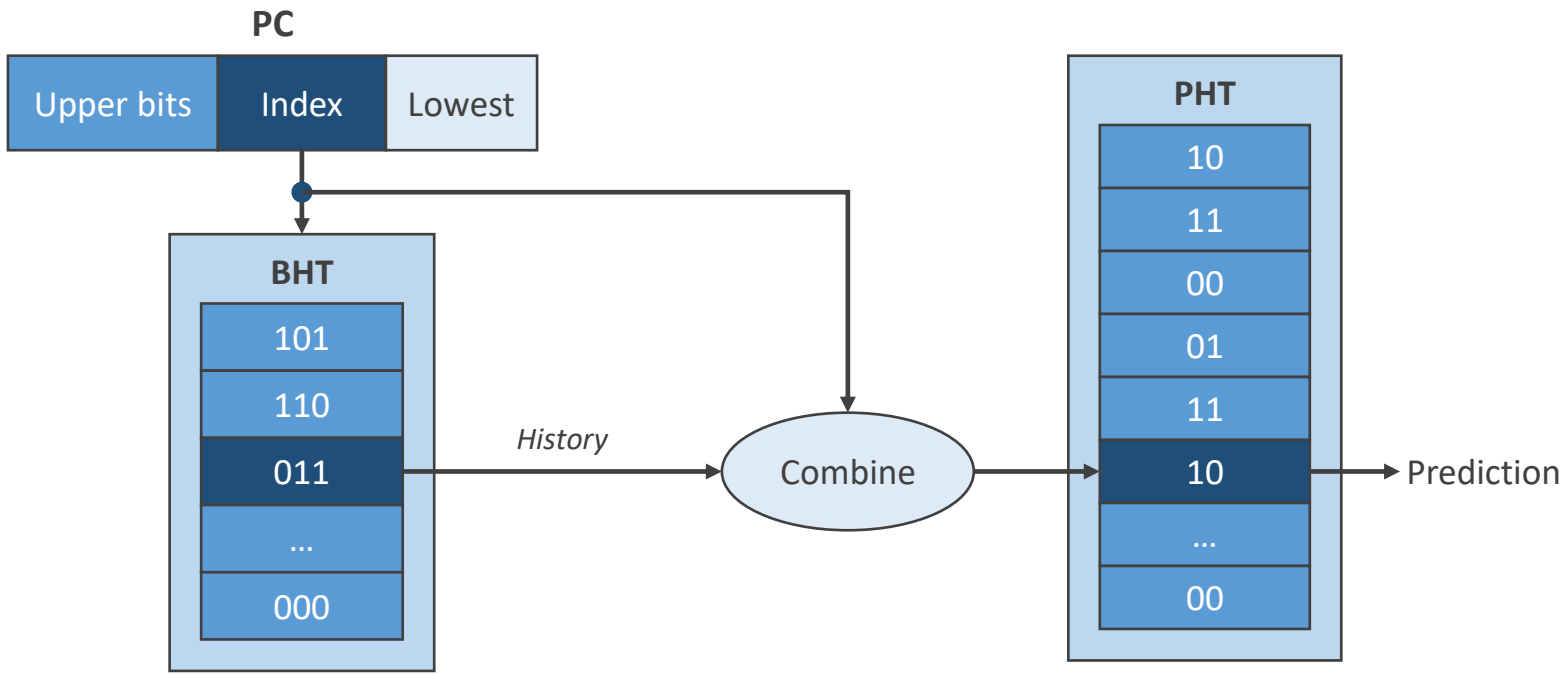
- This will always mispredict the loop exit, but we know it will always execute ten times
- If we knew what had happened to this branch in the past, we could learn this pattern



Mispredict on every loop exit

Local-history predictor

- The BHT (branch history table) is used to get the history for a branch
- The PHT (pattern history table) gives the prediction for a branch when it has a given history



Global history

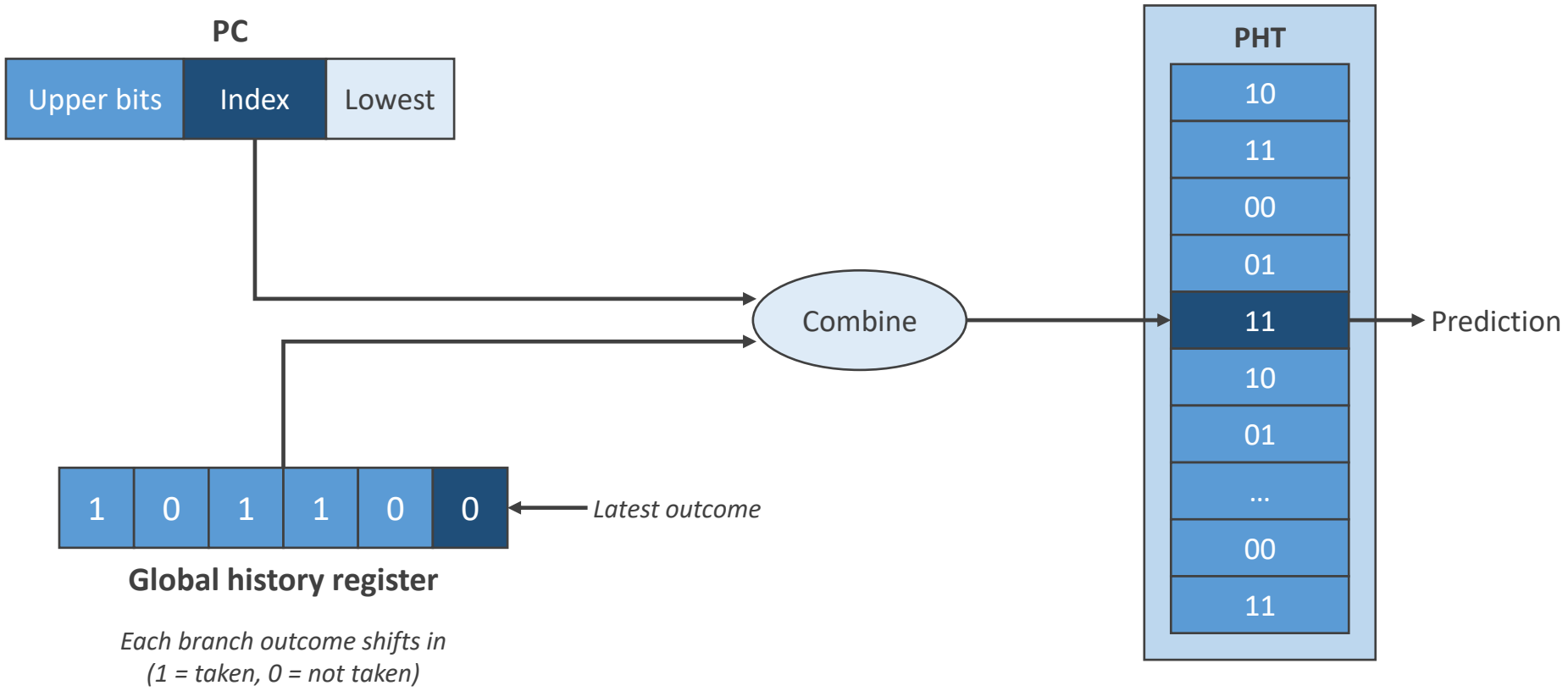
- Some branches don’t have a fixed pattern when considering only their behaviour
- But patterns emerge when you consider other branches too
- For example

```
if (x % 2 == 0)
   ...

...

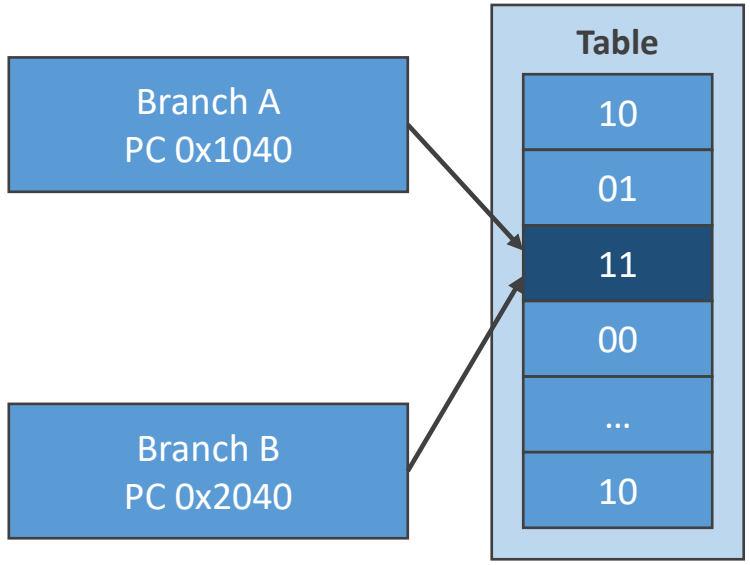
if ((x&1) == 0)  ← will always be the same as the first
   ...
```

Global-history predictor



Considerations

- To avoid pipeline bubbles, predictors must provide a ‘not-taken’ or ‘taken’ signal in one cycle
  - And a target address if taken
- The tables are finite and not tagged
- This means aliasing occurs when multiple branches map to the same entry
  - Can lead to constructive or destructive interference depending on how lucky you are

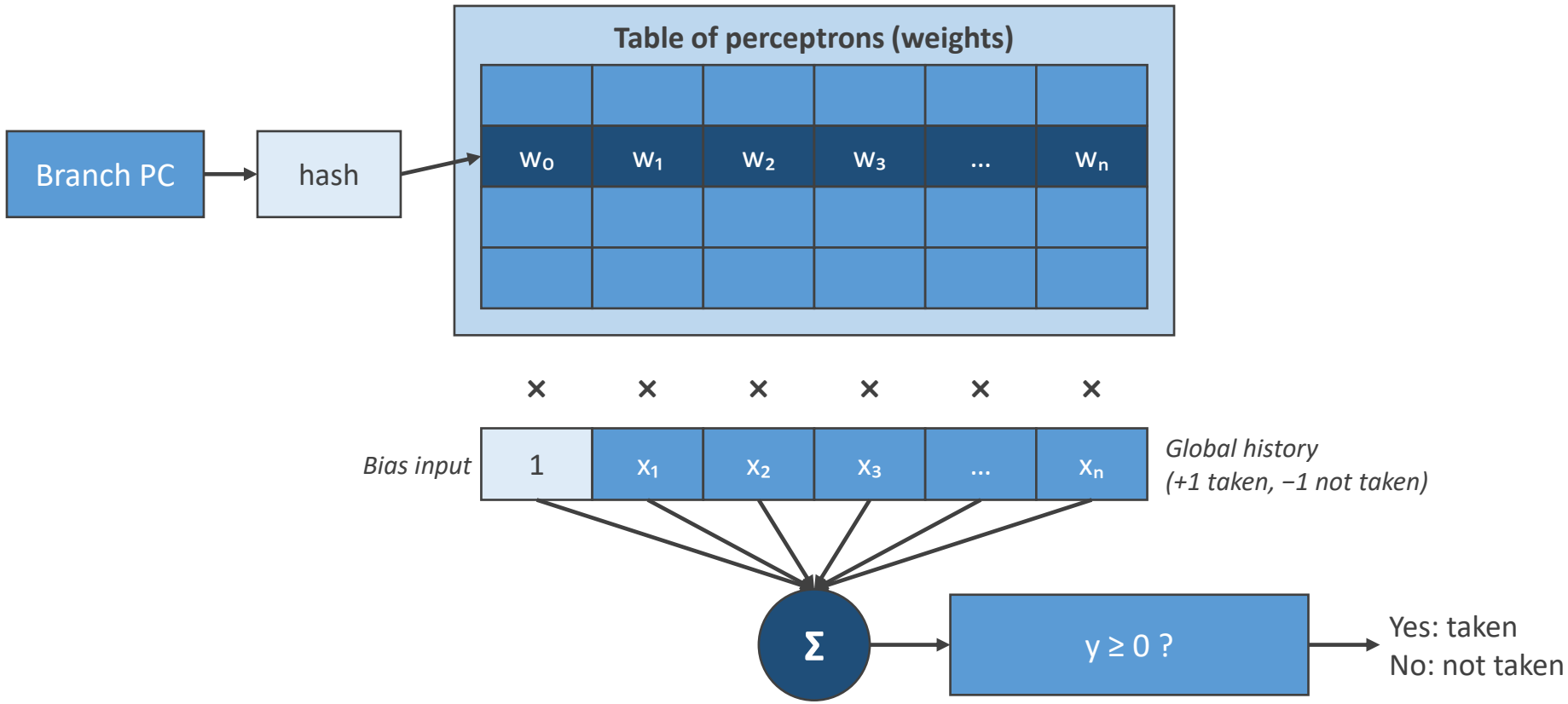


Same index bits, so both branches share one counter

Perceptron

- Use a single-layer one-neuron perceptron for branch prediction
  - Proposed by Jiménez and Lin in HPCA 2001
- Learns the linear target boolean function from a vector of inputs to a binary output
- $y = w_0 + \sum_{i=1}^{n} x_i w_i$
- Each x_i is a bit from the global branch history register
- Each branch is represented by a different set of weights (subject to aliasing)

Perceptron


Table of perceptrons (weights)
Branch PC
hash
w_0 w_1 w_2 w_3 ... w_n
×
Bias input
1 x_1 x_2 x_3 ... x_n
Global history
(+1 taken, −1 not taken)
Σ
y ≥ 0 ?
Yes: taken
No: not taken

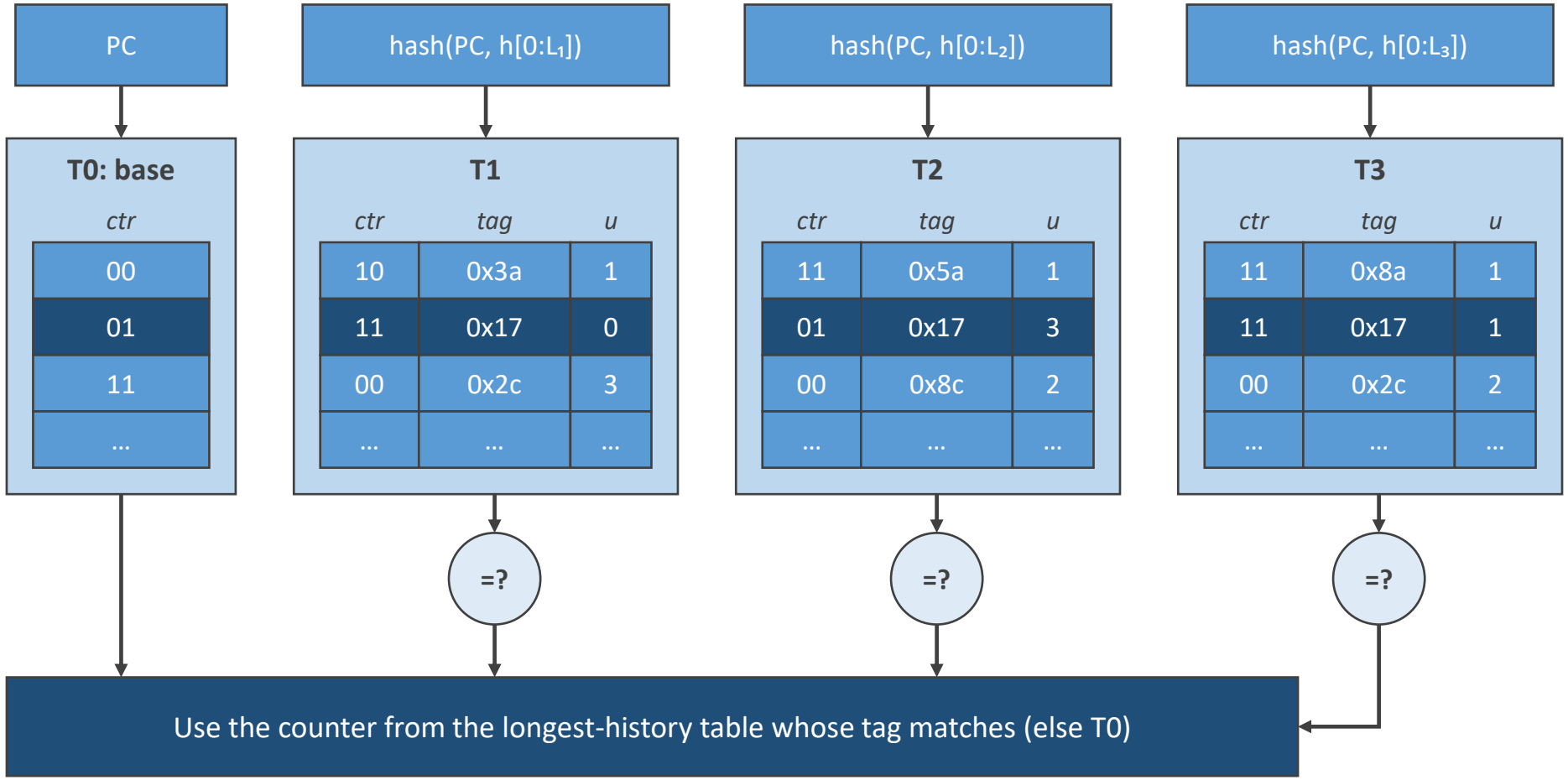
TAGE

- Different branches require different global-history lengths for prediction
- For example

```
if (x > 0) ...   ->                          1 0 1
if (x > 5) ...   ->                    1 1 1 1 0 1
if (x > 5 && y != 0) ...   ->  1 0 0 1 1 1 1 0 1
```

- The key idea is to provide multiple PHTs indexed by varying global-history lengths

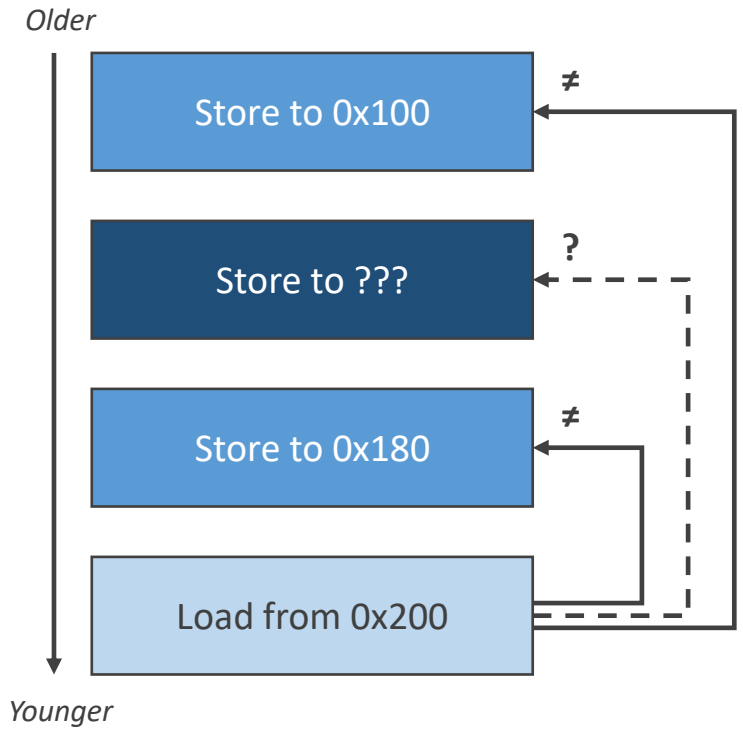
TAGE



$L_1 < L_2 < L_3$ form a geometric series of global-history lengths; u = usefulness counter

Memory-dependence prediction

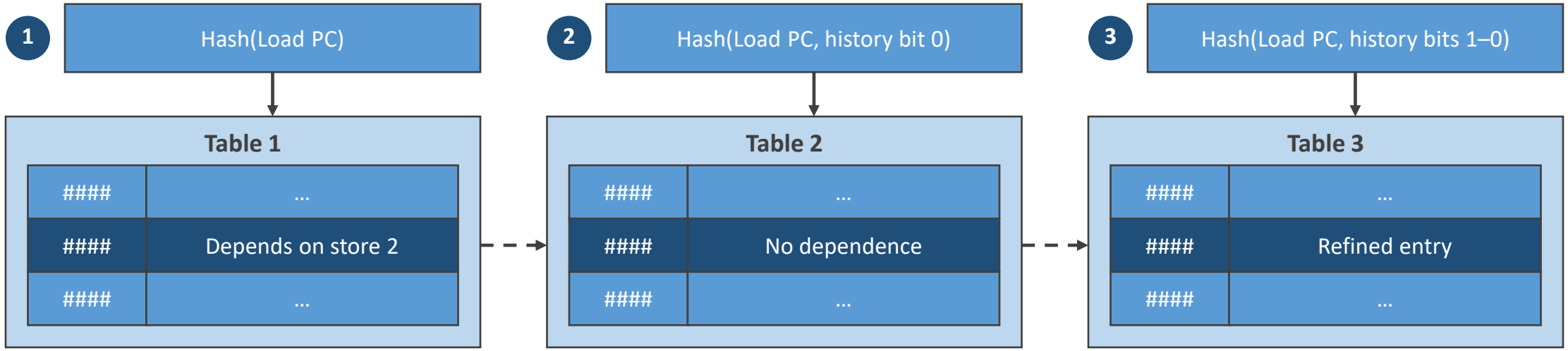
- Another type of predictor is used in conjunction with the load / store queue
- An executing load checks its address against all older stores
- If no match then it's safe to execute before those stores
  - If there is a match it can get its data from the most-recent older store
- But what happens if a store doesn't have its address generated yet?
- Here memory-dependence prediction can help



Predict: does the load depend on the unknown store?

MASCOT

- MASCOT is the state-of-the-art memory-dependence predictor
- It uses TAGE to keep a record of the prior store that a load is dependent on
- Crucially it also records non-dependencies when a load is predicted to depend on a prior store but doesn't depend on any



Global history: 11100110 (bit 0 = most recent branch). A false dependence allocates an entry in the next, higher-context table.

Value prediction

- Value prediction is the only scheme to break true data dependences
  - This can increase the amount of ILP by increasing the number of independent instructions to schedule
- The idea is simple: predict the value an instruction will produce
  - This could be simply the last value produced (or from the last *n* values)
  - Or a learned pattern (e.g. a stride)
  - Or through use of context (e.g. recent values, branch history) hashed to a table of values
- Long-latency instructions are often the target, or those deemed ‘critical’

