Design of Digital Circuits

Lecture 18: Branch Prediction

Prof. Onur Mutlu
ETH Zurich

Spring 2018

3 May 2018

Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Other Execution Paradigms

Reminder: Optional Homeworks

- Posted online
  - 3 Optional Homeworks
- Optional
- Good for your learning
- https://safari.ethz.ch/digitaltechnik/spring2018/doku.php?id=homeworks

Readings for Today

- Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

- H&H Chapters 7.8 and 7.9

- Optional:
  - Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.
  - **McFarling, "Combining Branch Predictors," DEC WRL Technical Report, 1993.**

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays

Control Dependence Handling

Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?

- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction

- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?

- In fact, how do we even know whether or not the fetched instruction is a control-flow instruction?

Branch Types

Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Different branch types can be handled differently

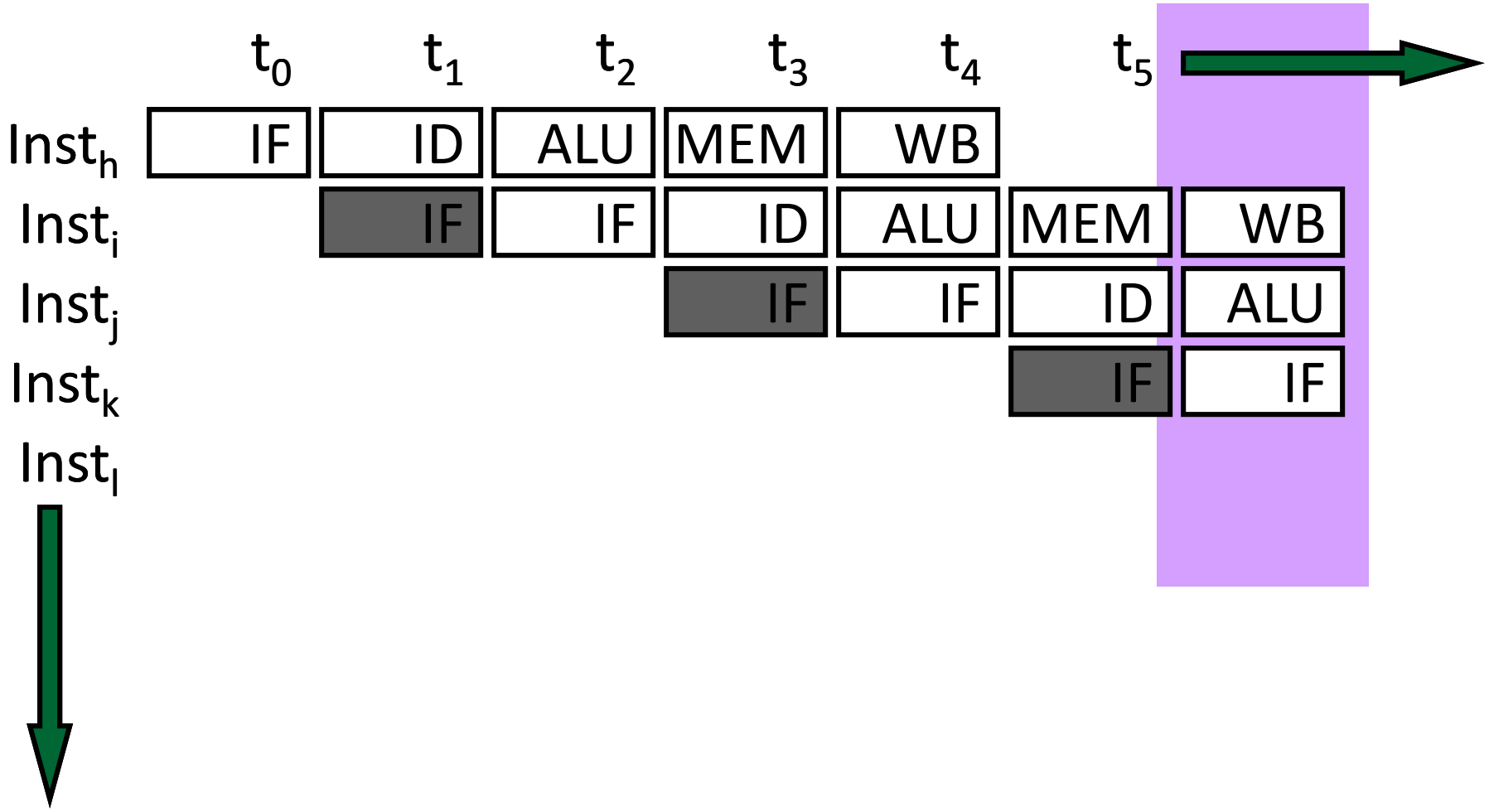
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Stall Fetch Until Next PC is Known: Good Idea?



This is the case with non-control-flow and unconditional br instructions!

The Branch Problem

- Control flow instructions (branches) are frequent
  - 15-25% of all instructions

- Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor
  - N cycles: (minimum) branch resolution latency

- If we are fetching W instructions per cycle (i.e., if the pipeline is W wide)
  - A branch misprediction leads to **N x W** wasted instruction slots

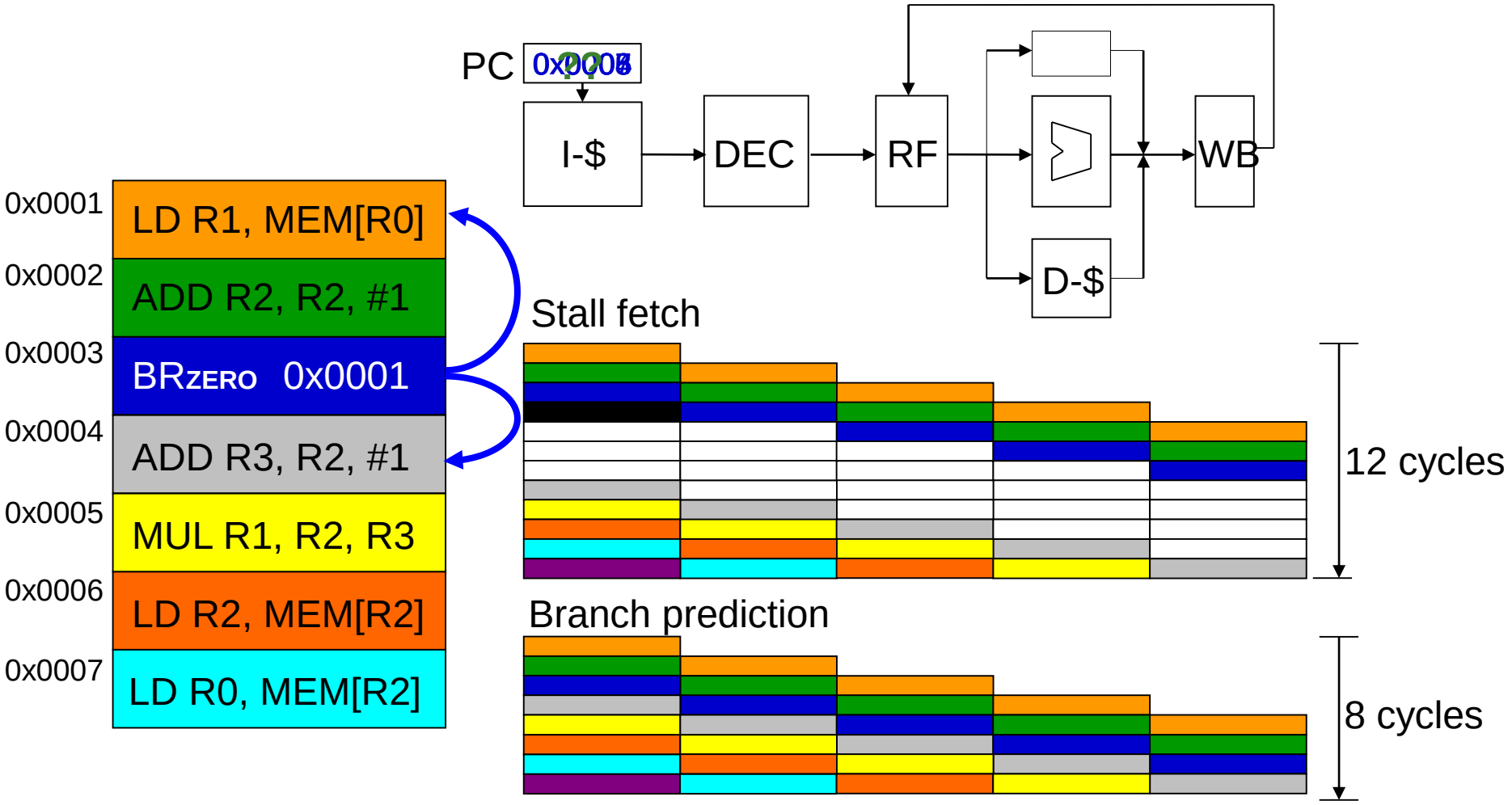
Importance of The Branch Problem

- Assume N = 20 (20 pipe stages), W = 5 (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

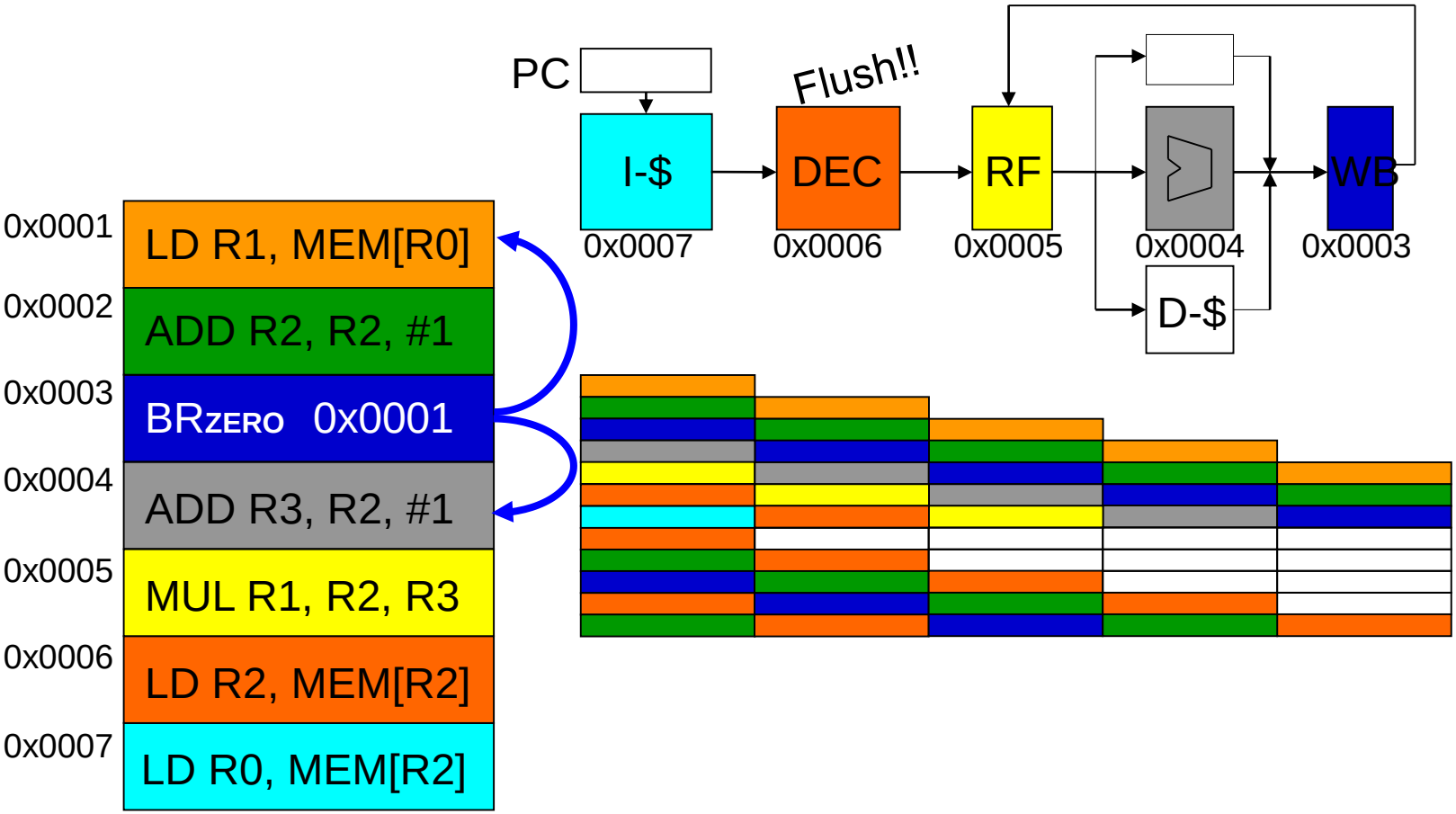
- How long does it take to fetch 500 instructions?
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work; IPC = 500/100
  - 99% accuracy
    - 100 (correct path) + 20 * 1 (wrong path) = 120 cycles
    - 20% extra instructions fetched; IPC = 500/120
  - 90% accuracy
    - 100 (correct path) + 20 * 10 (wrong path) = 300 cycles
    - 200% extra instructions fetched; IPC = 500/300
  - 60% accuracy
    - 100 (correct path) + 20 * 40 (wrong path) = 900 cycles
    - 800% extra instructions fetched; IPC = 500/900

Branch Prediction

Branch Prediction: Guess the Next Instruction to Fetch



Misprediction Penalty



Simplest: Always Guess NextPC = PC + 4

- Always predict the next sequential instruction is the next instruction to be executed
- This is a form of next fetch address prediction (and branch prediction)

- How can you make this more effective?

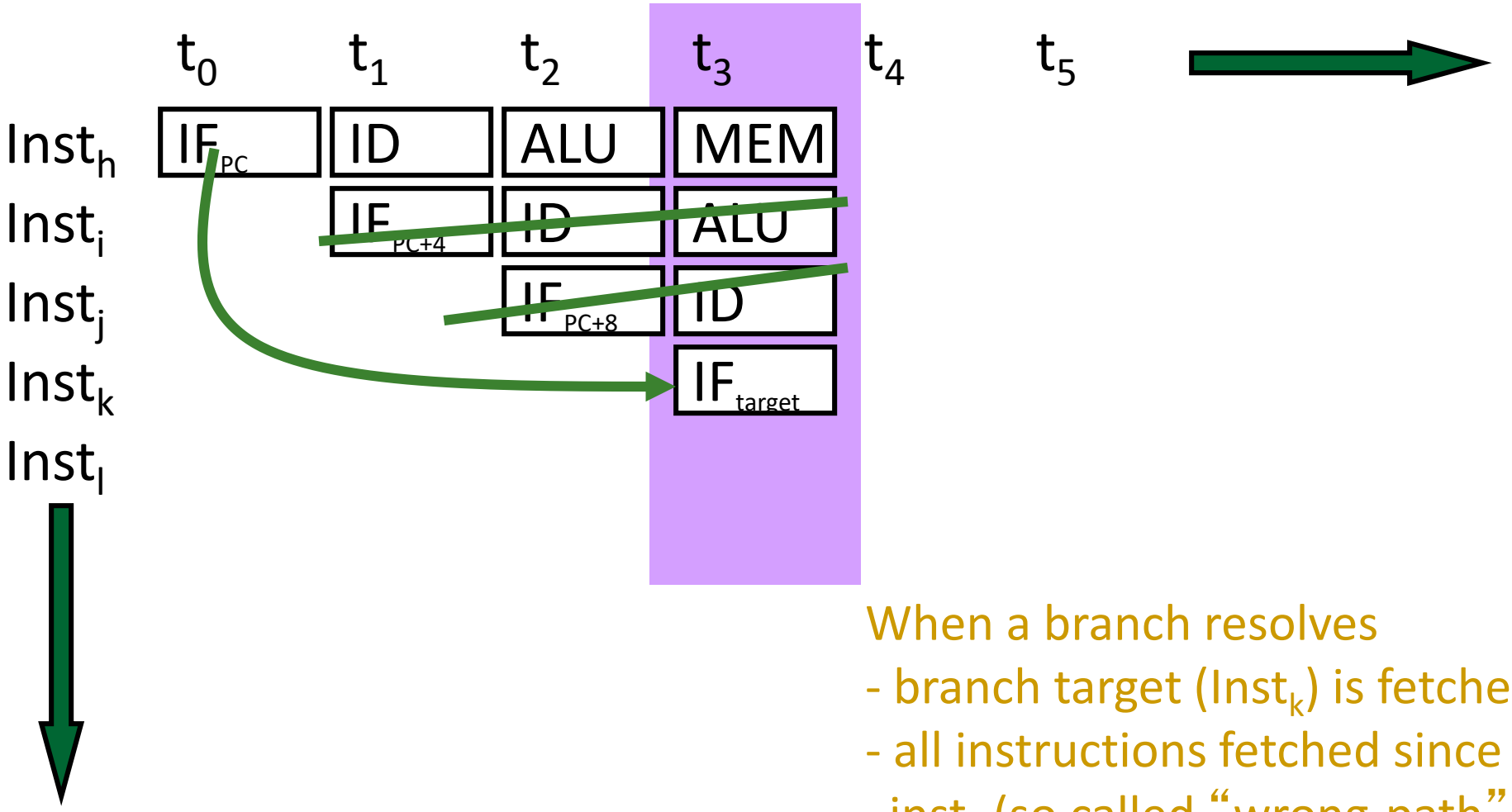
- Idea: Maximize the chances that the next sequential instruction is the next instruction to be executed
  - Software: Lay out the control flow graph such that the "likely next instruction" is on the not-taken path of a branch
    - Profile guided code positioning → Pettis & Hansen, PLDI 1990.
  - Hardware: ??? (how can you do this in hardware…)
    - Cache traces of executed instructions → Trace cache

Guessing NextPC = PC + 4

- How else can you make this more effective?
- Idea: Get rid of control flow instructions (or minimize their occurrence)
- How?

  1. Get rid of unnecessary control flow instructions → combine predicates (predicate combining)
  2. Convert control dependences into data dependences → predicated execution

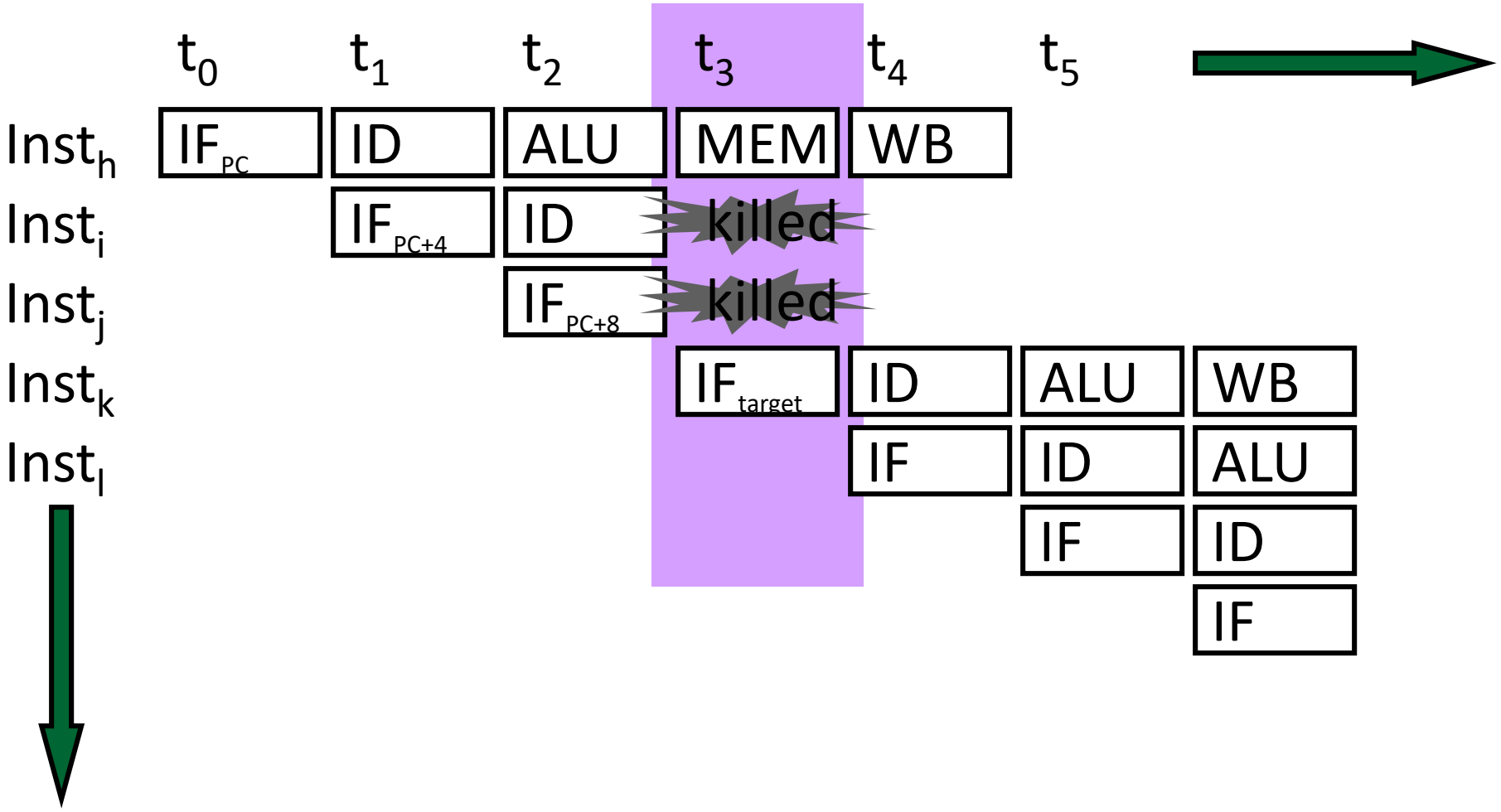
Branch Prediction: Always PC+4



When a branch resolves
- branch target ($Inst_k$) is fetched
- all instructions fetched since $inst_h$ (so called "wrong-path" instructions) must be flushed

$Inst_h$ is a branch

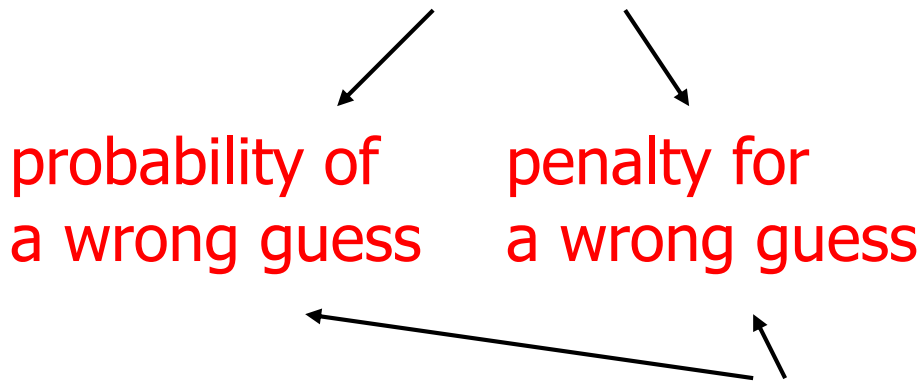
Pipeline Flush on a Misprediction



$Inst_h$ is a branch

Performance Analysis

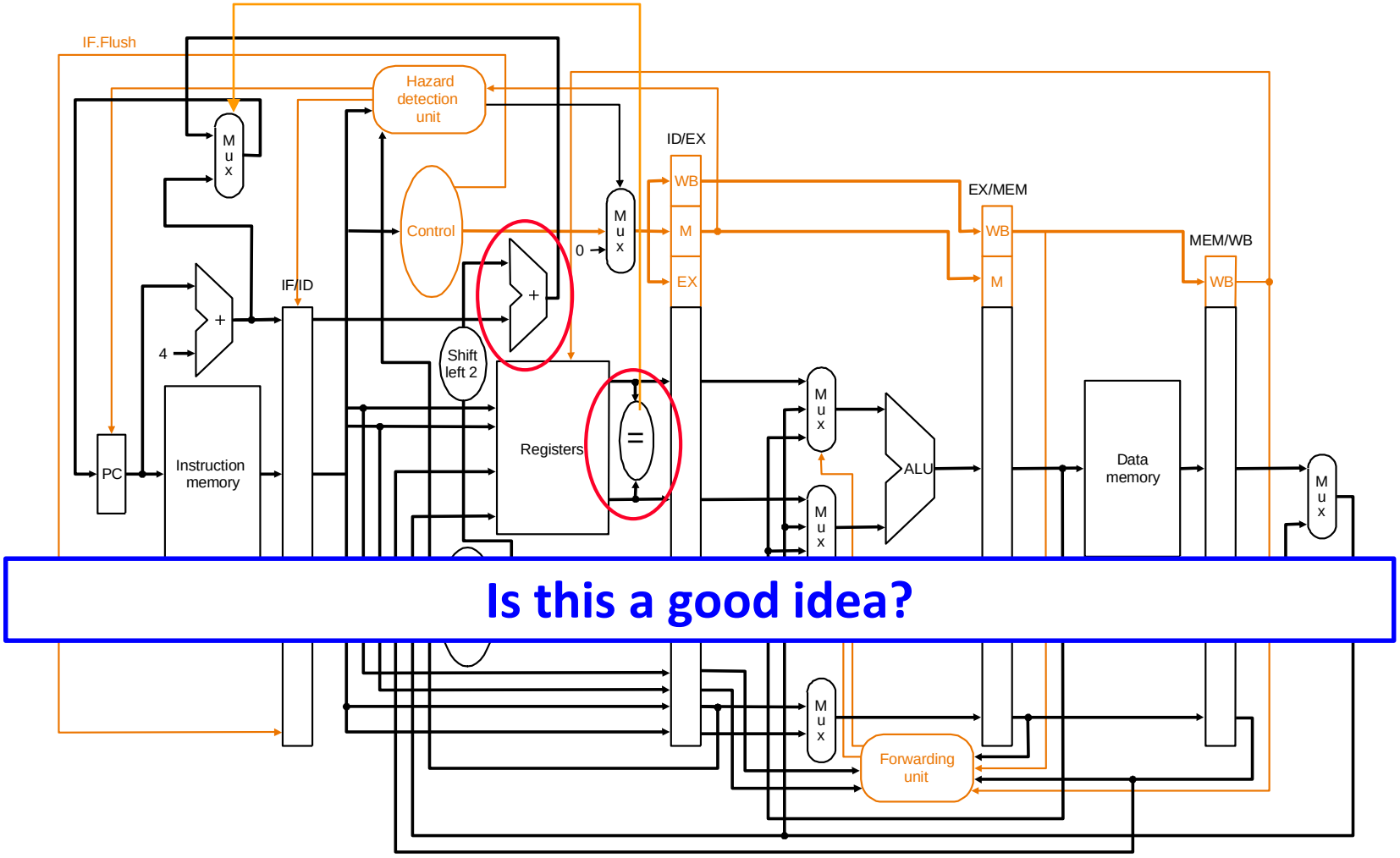
- correct guess ⇒ no penalty ~86% of the time
- incorrect guess ⇒ 2 bubbles
- Assume
  - no data dependency related stalls
  - 20% control flow instructions
  - 70% of control flow instructions are taken
  - CPI = [1 + (0.20*0.7) * 2] =
    = [1 + 0.14 * 2] = 1.28



Can we reduce either of the two penalty terms?

Reducing Branch Misprediction Penalty

- Resolve branch condition and target address early



[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

CPI = [1 + (0.2*0.7) * 1] = 1.14

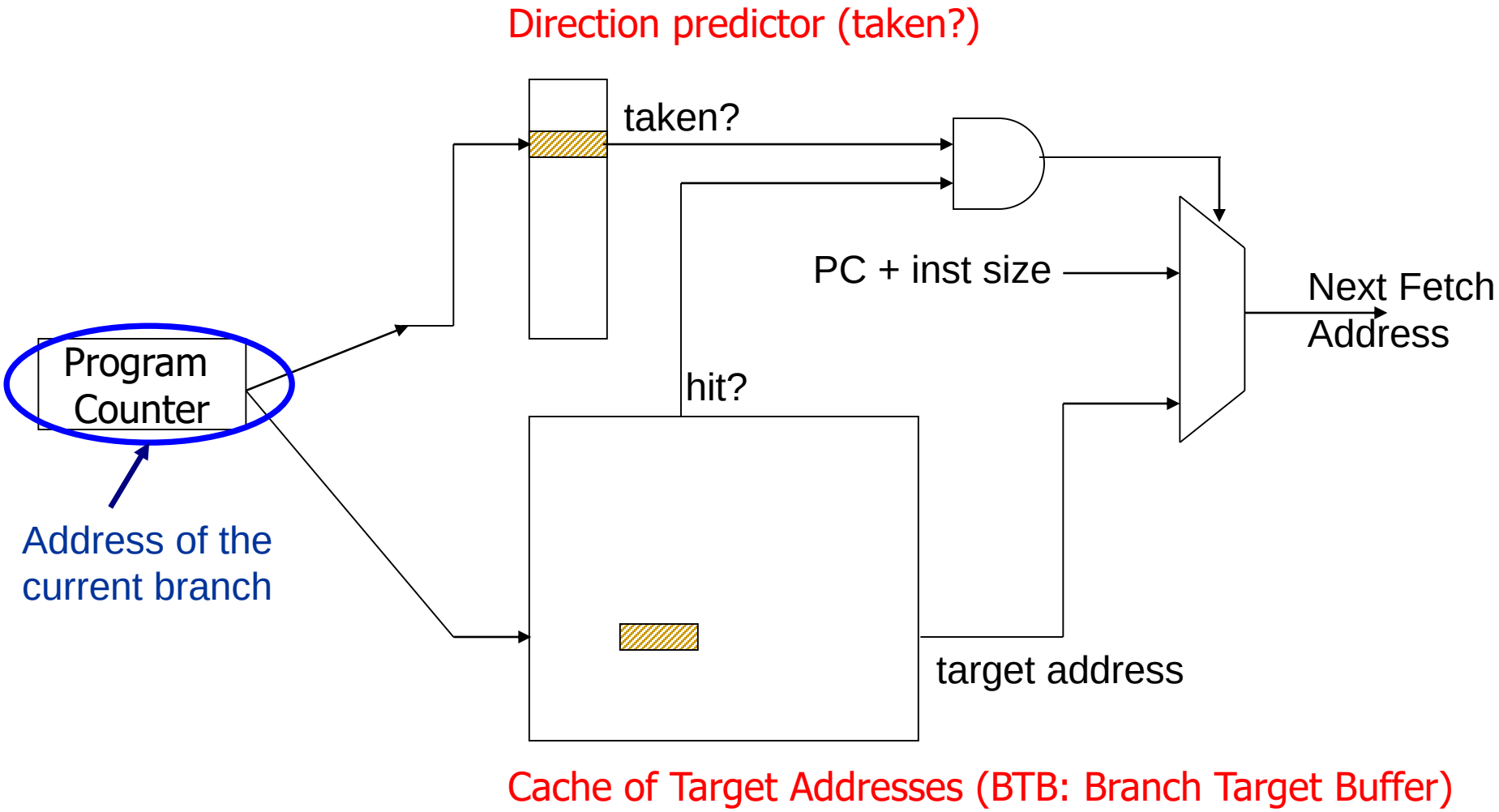
Branch Prediction (A Bit More Enhanced)

- Idea: Predict the next fetch address (to be used in the next cycle)

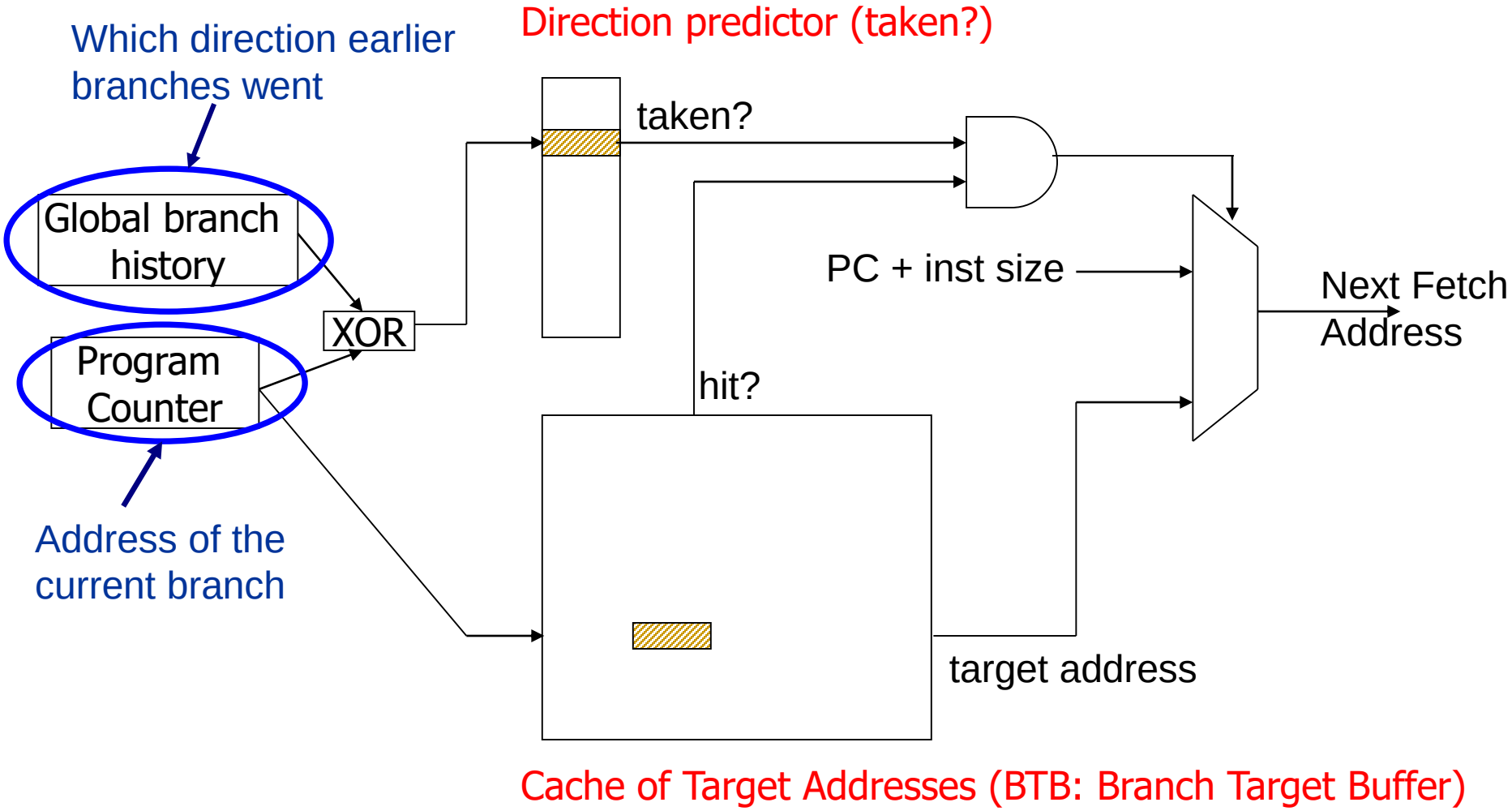
- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)

- Observation: Target address remains the same for a conditional direct branch across dynamic instances
  - Idea: Store the target address from previous instance and access it with the PC
  - Called Branch Target Buffer (BTB) or Branch Target Address Cache

Fetch Stage with BTB and Direction Prediction



More Sophisticated Branch Direction Prediction



Three Things to Be Predicted

- Requires three things to be predicted at fetch stage:
  1. Whether the fetched instruction is a branch
  2. (Conditional) branch direction
  3. Branch target address (if taken)

- Third (3.) can be accomplished using a BTB
  - Remember target address computed last time branch was executed
- First (1.) can be accomplished using a BTB
  - If BTB provides a target address for the program counter, then it must be a branch
  - Or, we can store "branch metadata" bits in instruction cache/memory → partially decoded instruction stored in I-cache
- Second (2.): How do we predict the direction?

Simple Branch Direction Prediction Schemes

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)

More Sophisticated Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid
  - Advanced algorithms (e.g., using perceptrons)

Static Branch Prediction (I)

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40% (for conditional branches)
  - Remember: Compiler can layout code such that the likely path is the "not-taken" path → more effective prediction

- Always taken
  - No direction prediction
  - Better accuracy: ~60-70% (for conditional branches)
    - Backward branches (i.e. loop branches) are usually taken
    - Backward branch: target address lower than branch PC

- Backward taken, forward not taken (BTFN)
  - Predict backward (loop) branches as taken, others not-taken

Static Branch Prediction (II)

- Profile-based
  - Idea: Compiler determines likely direction for each branch using a profile run. Encodes that direction as a hint bit in the branch instruction format.

+ Per branch prediction (more accurate than schemes in previous slide) → accurate if profile is representative!

-- Requires hint bits in the branch instruction format

-- Accuracy depends on dynamic branch behavior:

TTTTTTTTTTNNNNNNNNNN → 50% accuracy
TNTNTNTNTNTNTNTNTNTN → 50% accuracy

-- Accuracy depends on the representativeness of profile input set

Static Branch Prediction (III)

- Program-based (or, program analysis based)
  - Idea: Use heuristics based on program analysis to determine statically-predicted direction
  - Example opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
  - Example loop heuristic: Predict a branch guarding a loop execution as taken (i.e., execute the loop)
  - Pointer and FP comparisons: Predict not equal

+ Does not require profiling

-- Heuristics might be not representative or good

-- Requires compiler analysis and ISA support (ditto for other static methods)

- Ball and Larus, "Branch prediction for free," PLDI 1993.
  - 20% misprediction rate

Static Branch Prediction (IV)

- Programmer-based
  - Idea: Programmer provides the statically-predicted direction
  - Via *pragmas* in the programming language that qualify a branch as likely-taken versus likely-not-taken

+ Does not require profiling or program analysis
+ Programmer may know some branches and their program better than other analysis techniques
-- Requires programming language, compiler, ISA support
-- Burdens the programmer?

Pragmas

- Idea: Keywords that enable a programmer to convey hints to lower levels of the transformation hierarchy

- if (likely(x)) { ... }
- if (unlikely(error)) { ... }

- Many other hints and optimizations can be enabled with pragmas
  - E.g., whether a loop can be parallelized
  - **#pragma omp parallel**
  - **Description**
    - The omp parallel directive explicitly instructs the compiler to parallelize the chosen segment of code.

Static Branch Prediction

- All previous techniques can be combined
  - Profile based
  - Program based
  - Programmer based

- How would you do that?

- What is the common disadvantage of all three techniques?
  - Cannot adapt to dynamic changes in branch behavior
    - This can be mitigated by a dynamic compiler, but not at a fine granularity (and a dynamic compiler has its overheads…)
    - What is a Dynamic Compiler?
      - A compiler that generates code at runtime: Remember Transmeta?
      - Java JIT (just in time) compiler, Microsoft CLR (common lang. runtime)

More Sophisticated Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid
  - Advanced algorithms (e.g., using perceptrons)

Dynamic Branch Prediction

- Idea: Predict branches based on dynamic information (collected at run-time)

- Advantages

  + Prediction based on history of the execution of branches

    + It can adapt to dynamic changes in branch behavior

  + No need for static profiling: input set representativeness problem goes away

- Disadvantages

  -- More complex (requires additional hardware)

Last Time Predictor

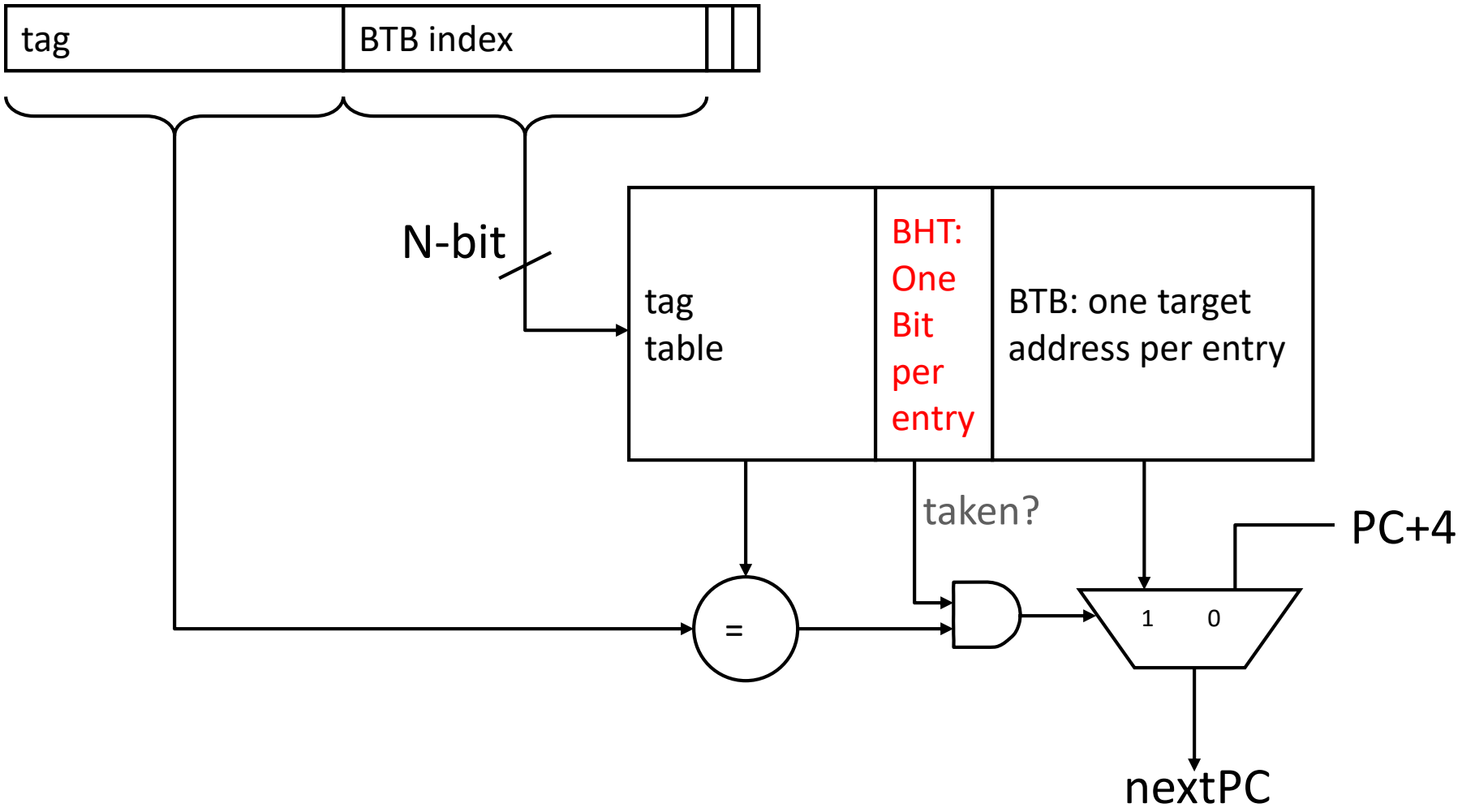
- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed
    TTTTTTTTTTNNNNNNNNNN → 90% accuracy

- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N

+ Loop branches for loops with large N (number of iterations)

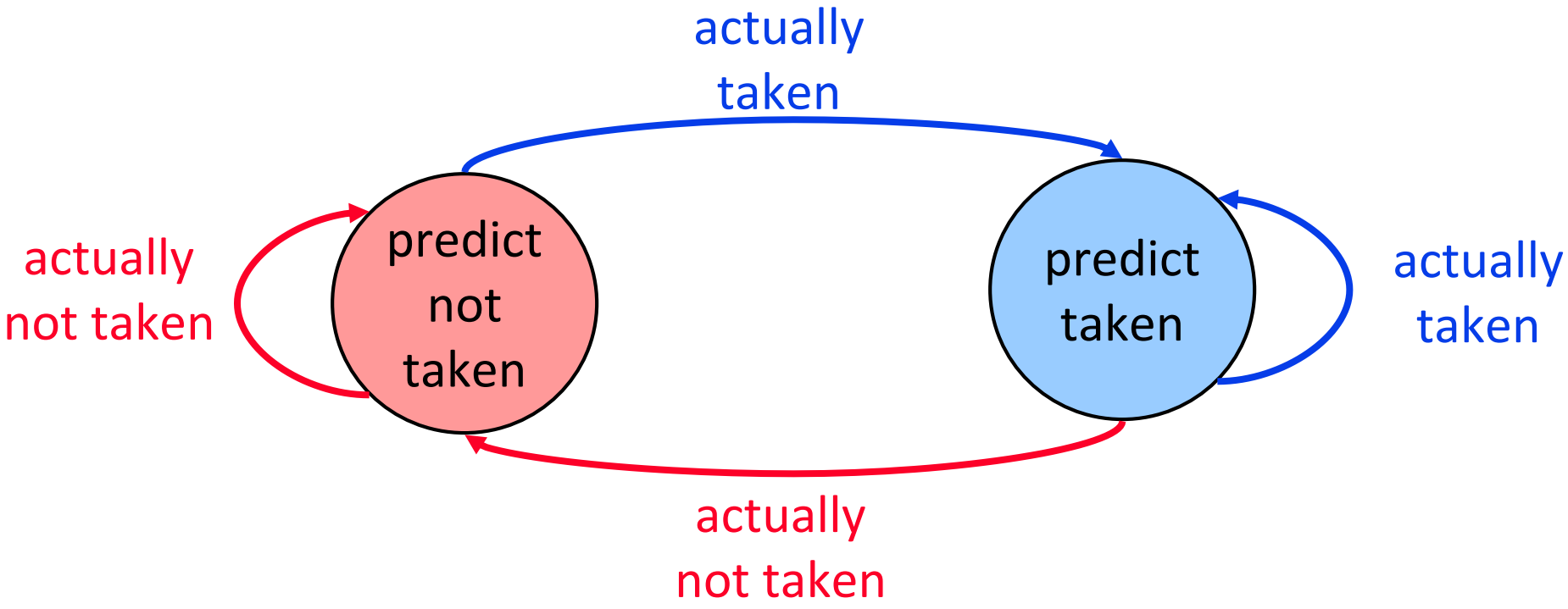
-- Loop branches for loops will small N (number of iterations)
TNTNTNTNTNTNTNTNTNTN → 0% accuracy

Implementing the Last-Time Predictor



The 1-bit BHT (Branch History Table) entry is updated with the correct outcome after each execution of a branch

State Machine for Last-Time Prediction



Improving the Last Time Predictor

- Problem: A last-time predictor changes its prediction from T→NT or NT→T too quickly
  - even though the branch may be mostly taken or mostly not taken
- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for T or NT instead of 1 state for each
- Smith, "A Study of Branch Prediction Strategies," ISCA 1981.

Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome

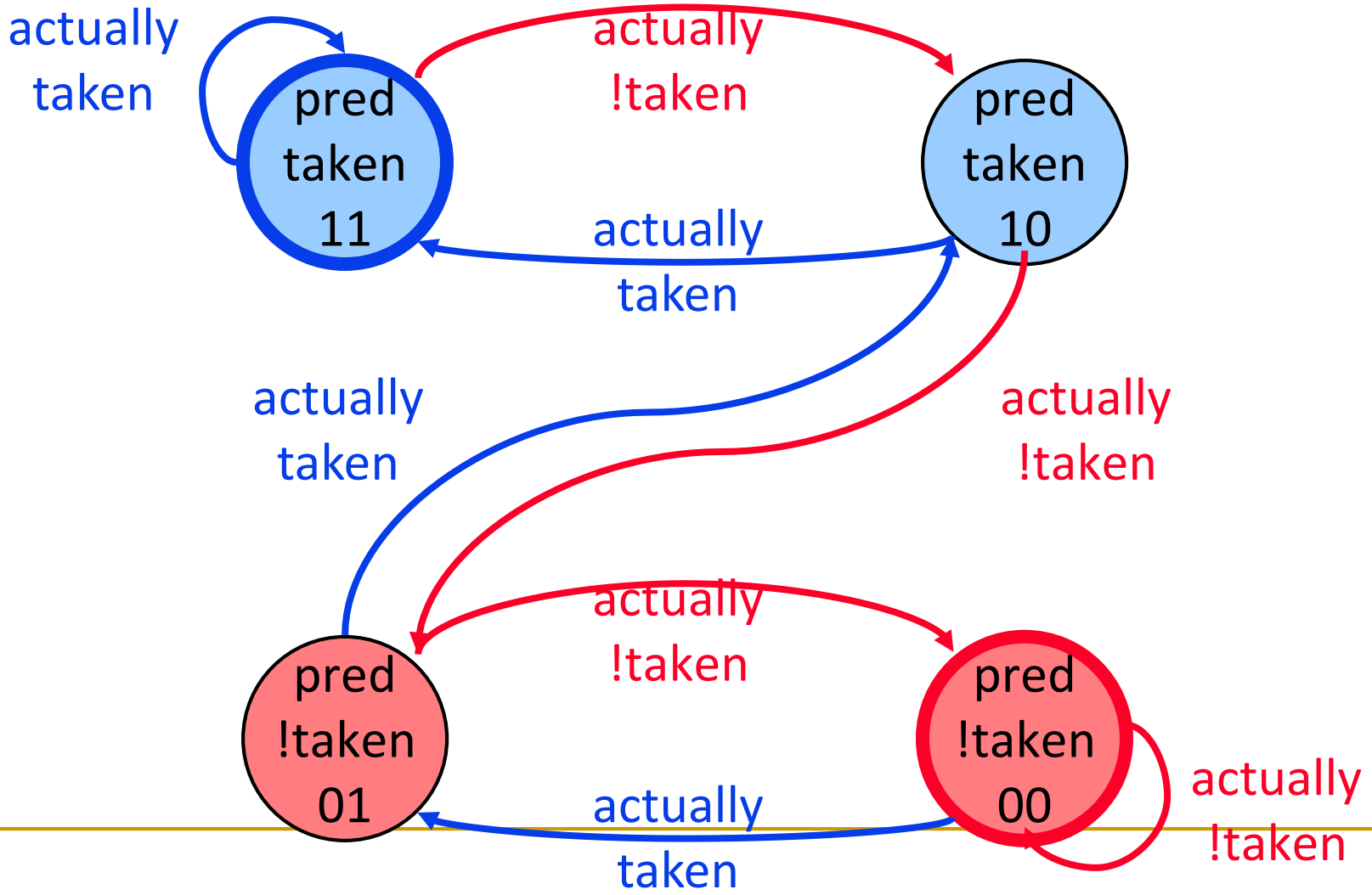
- Accuracy for a loop with N iterations = (N-1)/N
  TNTNTNTNTNTNTNTNTNTNTN → 50% accuracy

(assuming counter initialized to weakly taken)

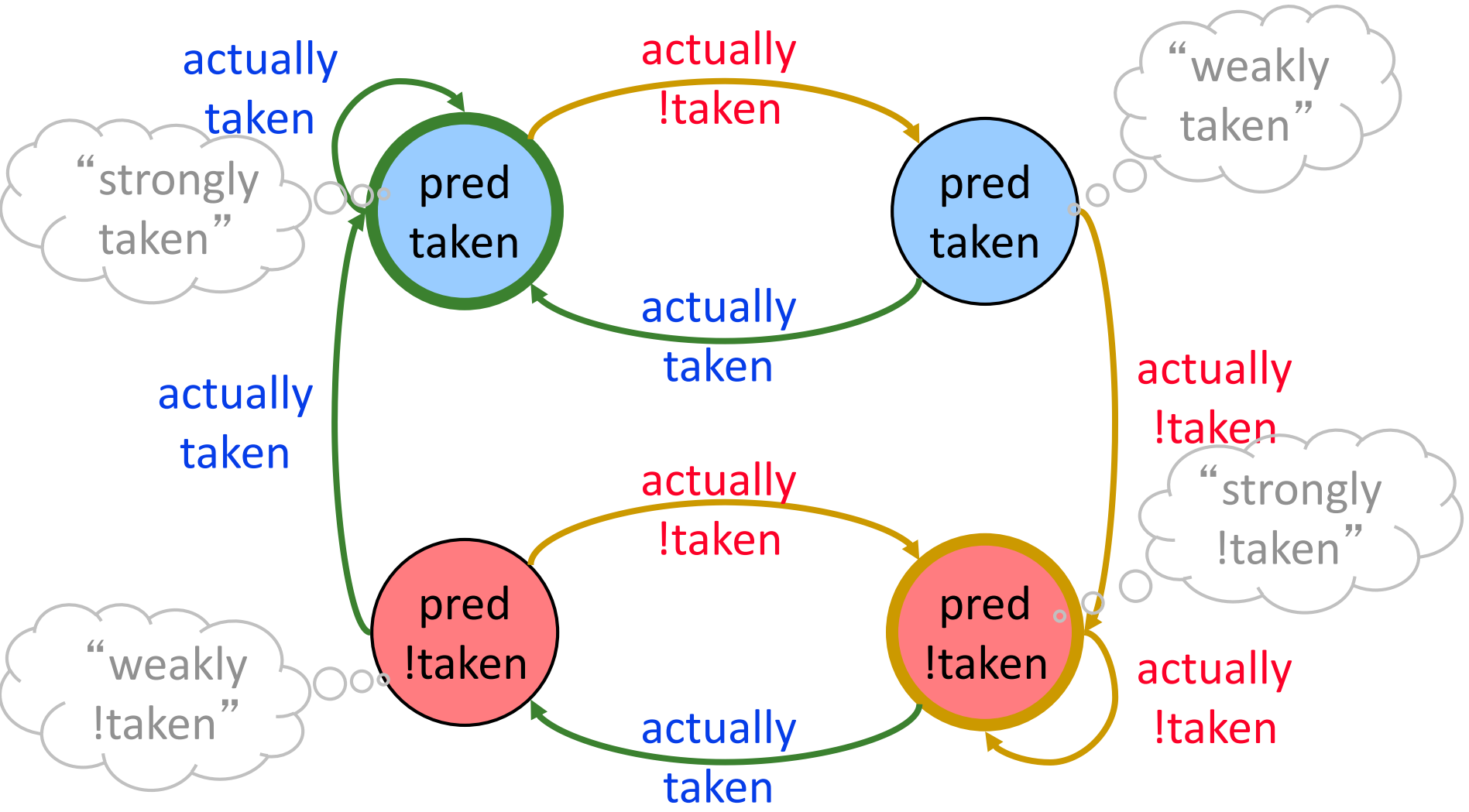
+ Better prediction accuracy
-- More hardware cost (but counter can be part of a BTB entry)

State Machine for 2-bit Saturating Counter

- Counter using *saturating arithmetic*
  - Arithmetic with maximum and minimum values



Hysteresis Using a 2-bit Counter



Change prediction after 2 consecutive mistakes

Is This Good Enough?

- ~85-90% accuracy for **many** programs with 2-bit counter based prediction (also called bimodal prediction)
- Is this good enough?
- How big is the branch problem?

Let's Do the Exercise Again

- Assume N = 20 (20 pipe stages), W = 5 (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

- How long does it take to fetch 500 instructions?
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work; IPC = 500/100
  - 90% accuracy
    - 100 (correct path) + 20 * 10 (wrong path) = 300 cycles
    - 200% extra instructions fetched; IPC = 500/300
  - 85% accuracy
    - 100 (correct path) + 20 * 15 (wrong path) = 400 cycles
    - 300% extra instructions fetched; IPC = 500/400
  - 80% accuracy
    - 100 (correct path) + 20 * 20 (wrong path) = 500 cycles
    - 400% extra instructions fetched; IPC = 500/500

Can We Do Better: Two-Level Prediction

- Last-time and 2BC predictors exploit "last-time" predictability

- Realization 1: A branch's outcome can be correlated with other branches' outcomes
  - Global branch correlation

- Realization 2: A branch's outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch "last-time" it was executed)
  - Local branch correlation

Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path are correlated with the outcome of the next branch

```
if (cond1)
...
if (cond1 AND cond2)
```

- If first branch not taken, second also not taken

```
branch Y: if (cond1) a = 2;
...
branch X: if (a == 0)
```

- If first branch taken, second definitely not taken

Global Branch Correlation (II)

```
branch Y: if (cond1)
...
branch Z: if (cond2)
...
branch X: if (cond1 AND cond2)
```

- If Y and Z both taken, then X also taken
- If Y or Z not taken, then X also not taken

Global Branch Correlation (III)

- Eqntott, SPEC'92: Generates truth table from Boolean expr.

```
if (aa==2)                    ;; B1
        aa=0;
if (bb==2)                    ;; B2
        bb=0;
if (aa!=bb) {                 ;; B3
        ....
  }
```

If B1 is not taken (i.e., aa==0@B3) and B2 is not taken (i.e. bb=0@B3) then B3 is certainly taken

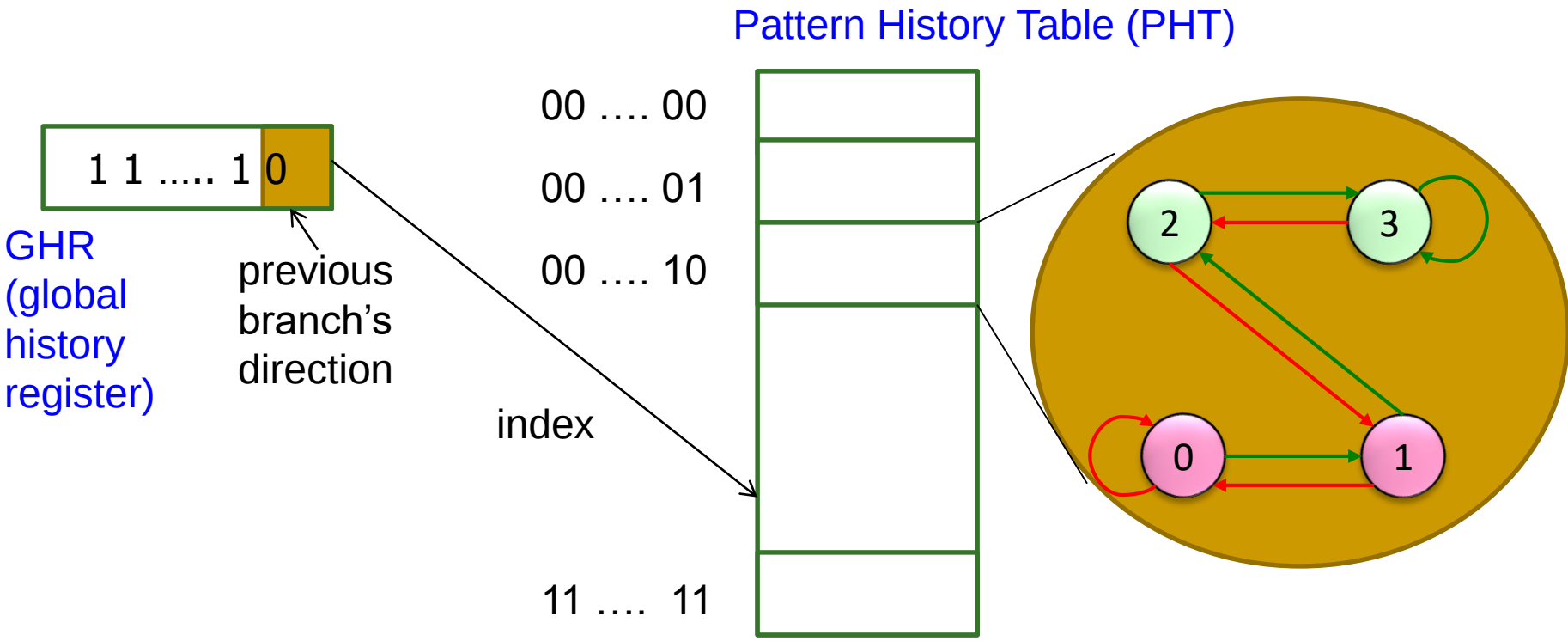
Capturing Global Branch Correlation

- Idea: Associate branch outcomes with "global T/NT history" of all branches
- Make a prediction based on the outcome of the branch the last time the same global branch history was encountered
- Implementation:
  - Keep track of the "global T/NT history" of all branches in a register → Global History Register (GHR)
  - Use GHR to index into a table that recorded the outcome that was seen for each GHR value in the recent past → Pattern History Table (table of 2-bit counters)
- Global history/branch predictor
- Uses two levels of history (GHR + history at that GHR)

Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

Two Level Global Branch Prediction

- First level: Global branch history register (N bits)
  - The direction of last N branches
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen



Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

How Does the Global Predictor Work?

```
for (i=0; i<100; i++)
    for (j=0; j<3; j++)
```

After the initial startup time, the conditional branches have the following behavior, assuming GR is shifted to the left:

test	value	GR	result
j<3	j=1	1101	taken
j<3	j=2	1011	taken
j<3	j=3	0111	not taken
i<100		1110	usually taken

This branch tests i
Last 4 branches test j
History: TTTN
Predict taken for i
Next history: TTNT
(shift in last outcome)

- McFarling, "Combining Branch Predictors," DEC WRL TR 1993.

Intel Pentium Pro Branch Predictor

- Two level global branch predictor
- 4-bit global history register
- Multiple pattern history tables (of 2 bit counters)
  - Which pattern history table to use is determined by lower order bits of the branch address

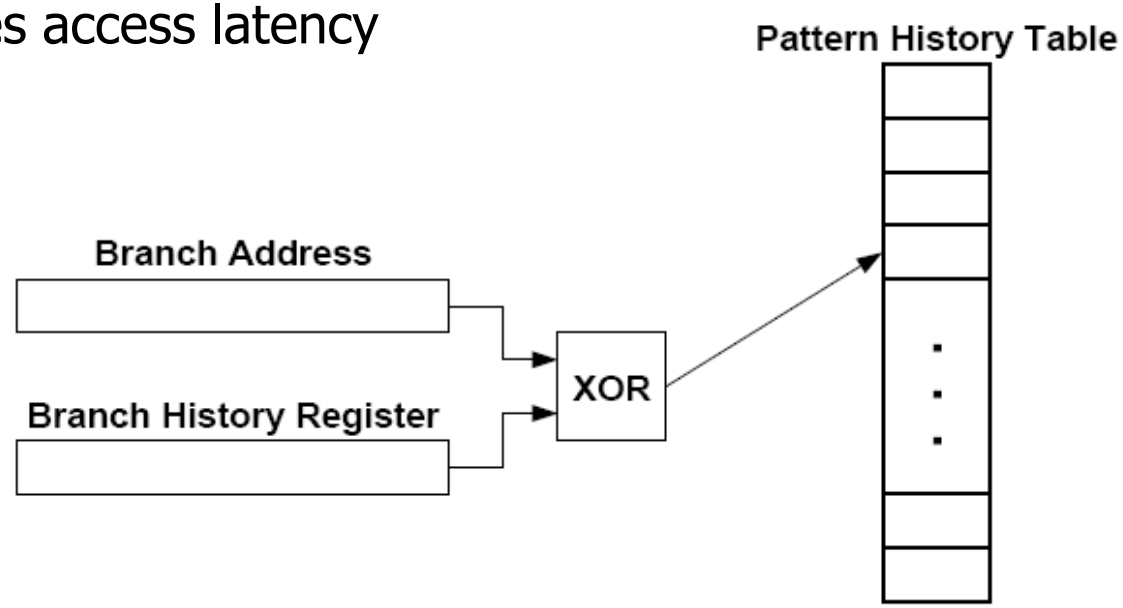
Improving Global Predictor Accuracy

- Idea: Add more context information to the global predictor to take into account which branch is being predicted
  - Gshare predictor: GHR hashed with the Branch PC

  + More context information

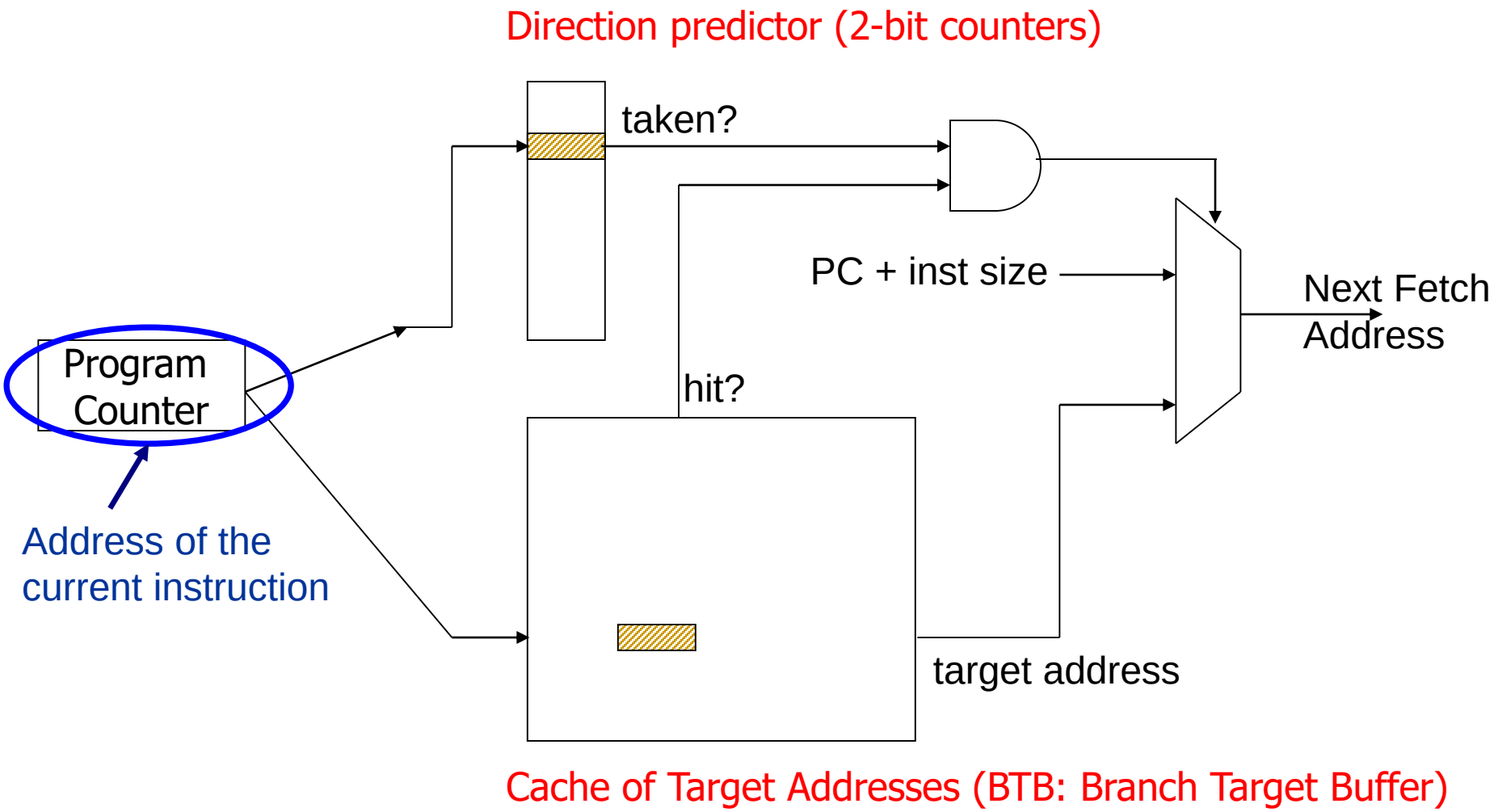
  + Better utilization of PHT

  -- Increases access latency

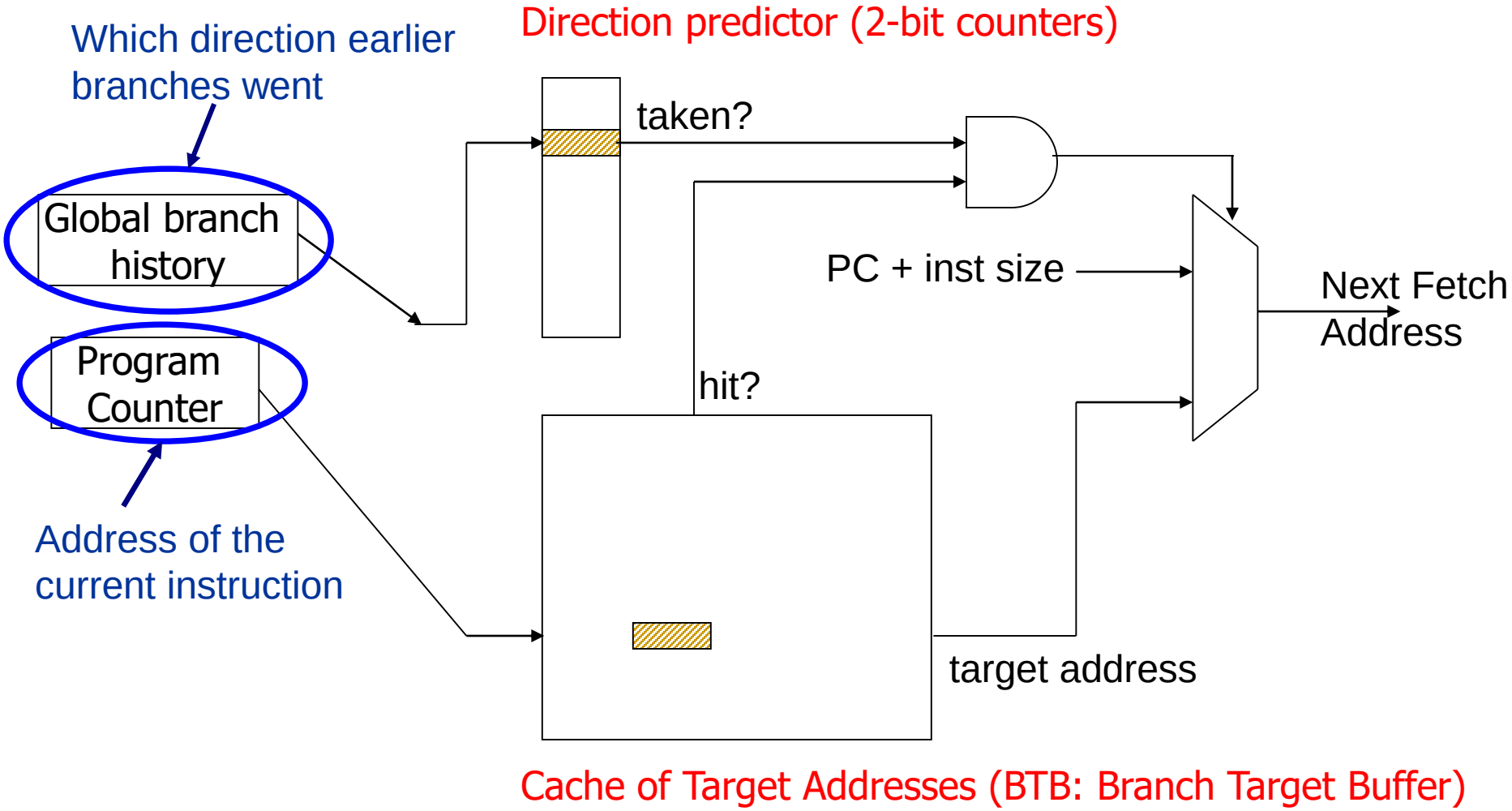


- McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

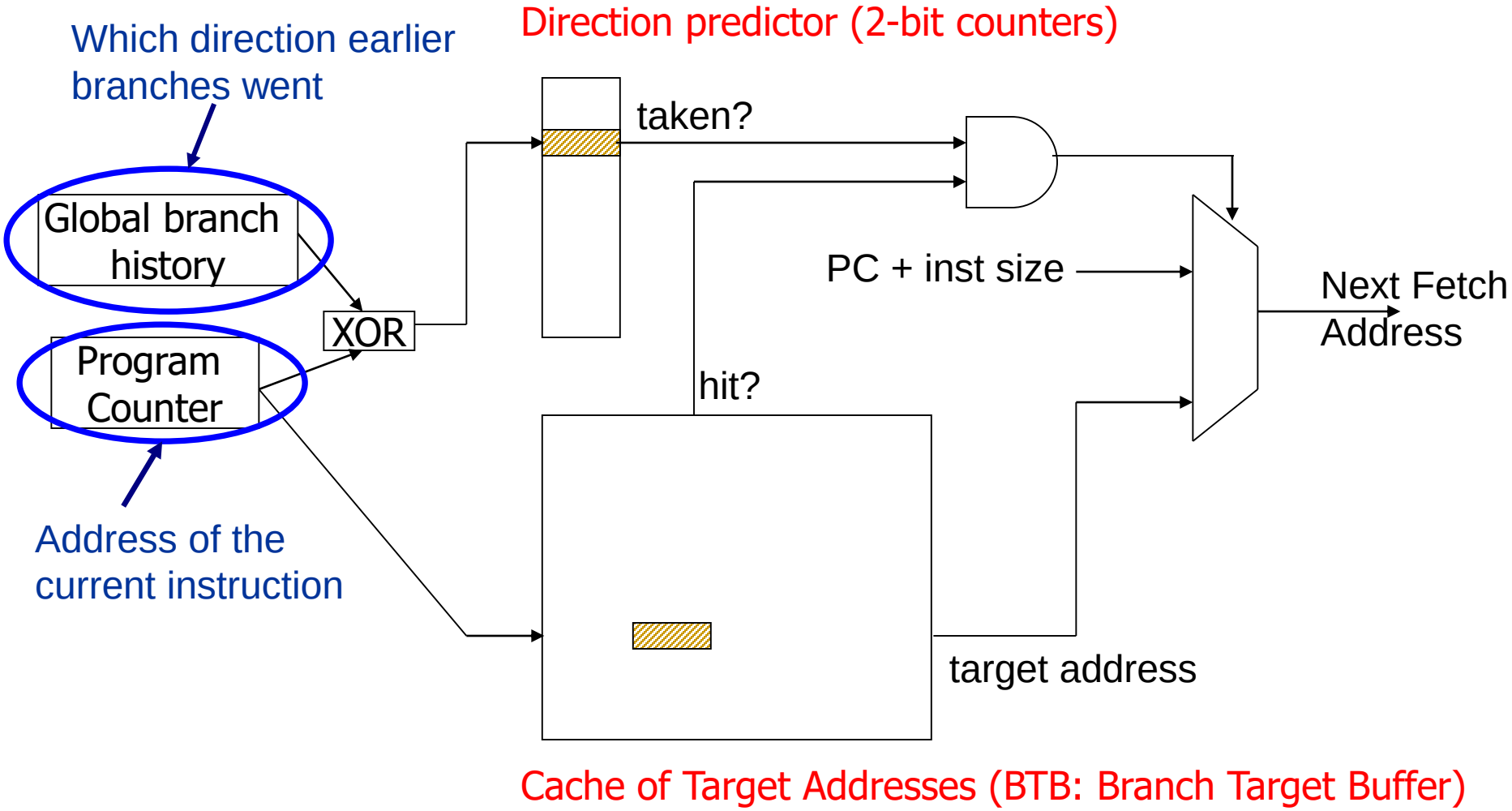
Review: One-Level Branch Predictor



Two-Level Global History Branch Predictor



Two-Level Gshare Branch Predictor



Can We Do Better: Two-Level Prediction

- Last-time and 2BC predictors exploit only "last-time" predictability for a given branch
- Realization 1: A branch's outcome can be correlated with other branches' outcomes
  - Global branch correlation
- Realization 2: A branch's outcome can be correlated with past outcomes of the same branch (in addition to the outcome of the branch "last-time" it was executed)
  - Local branch correlation

Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

Local Branch Correlation

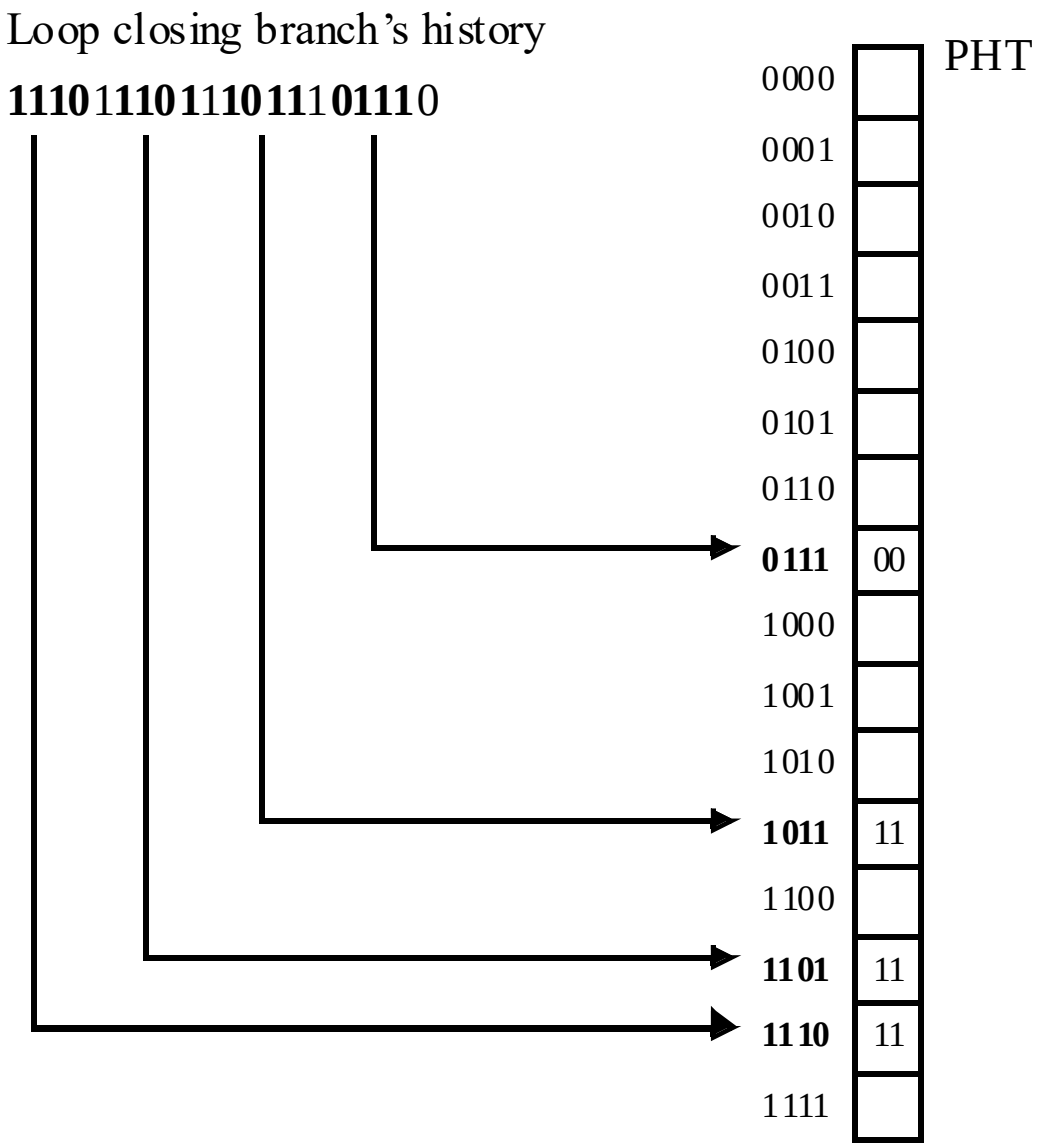
```
for (i=1; i<=4; i++) { }
```

If the loop test is done at the end of the body, the corresponding branch will execute the pattern $(1110)^n$, where 1 and 0 represent taken and not taken respectively, and n is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

- McFarling, "Combining Branch Predictors," DEC WRL TR 1993.

More Motivation for Local History

- To predict a loop branch "perfectly", we want to identify the last iteration of the loop
- By having a separate PHT entry for each local history, we can distinguish different iterations of a loop
- Works for "short" loops



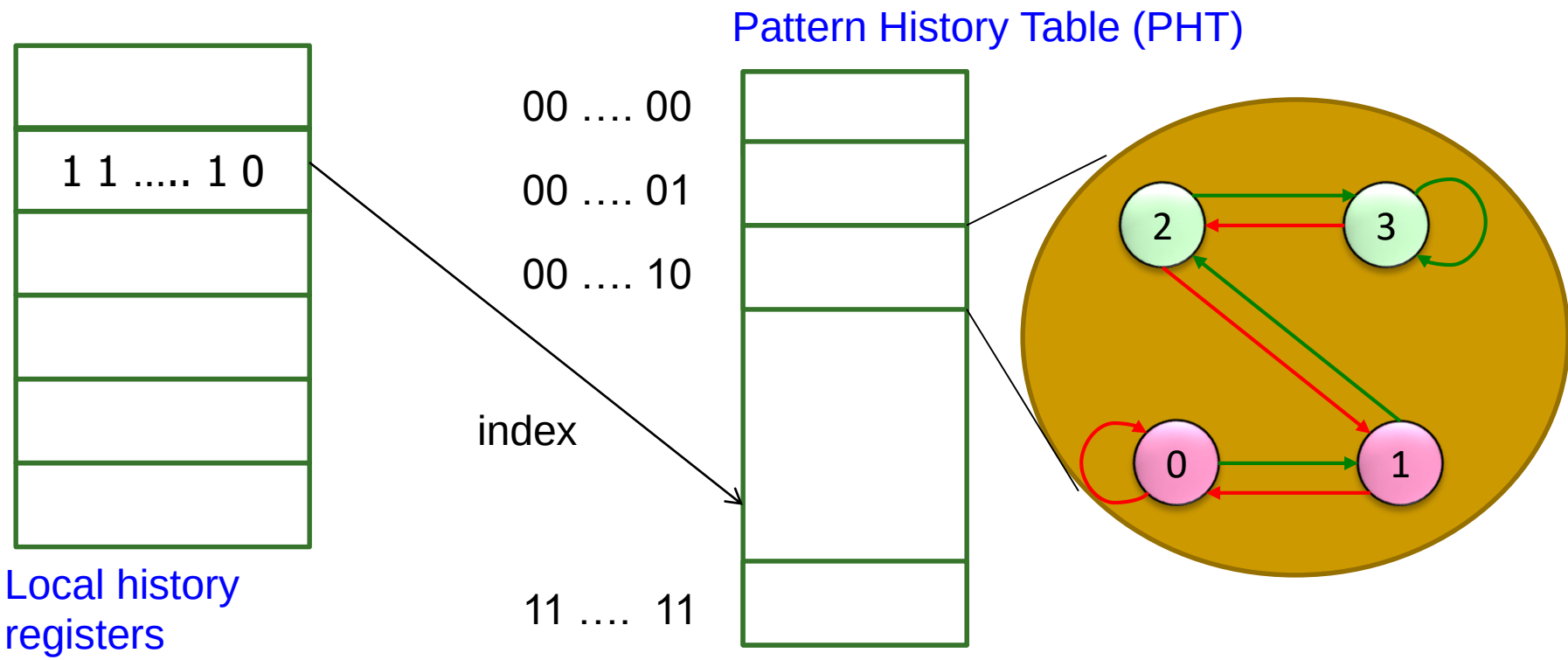
Capturing Local Branch Correlation

- Idea: Have a per-branch history register
  - Associate the predicted outcome of a branch with "T/NT history" of the same branch
- Make a prediction based on the outcome of the branch the last time the same local branch history was encountered

- Called the local history/branch predictor
- Uses two levels of history (Per-branch history register + history at that history register value)

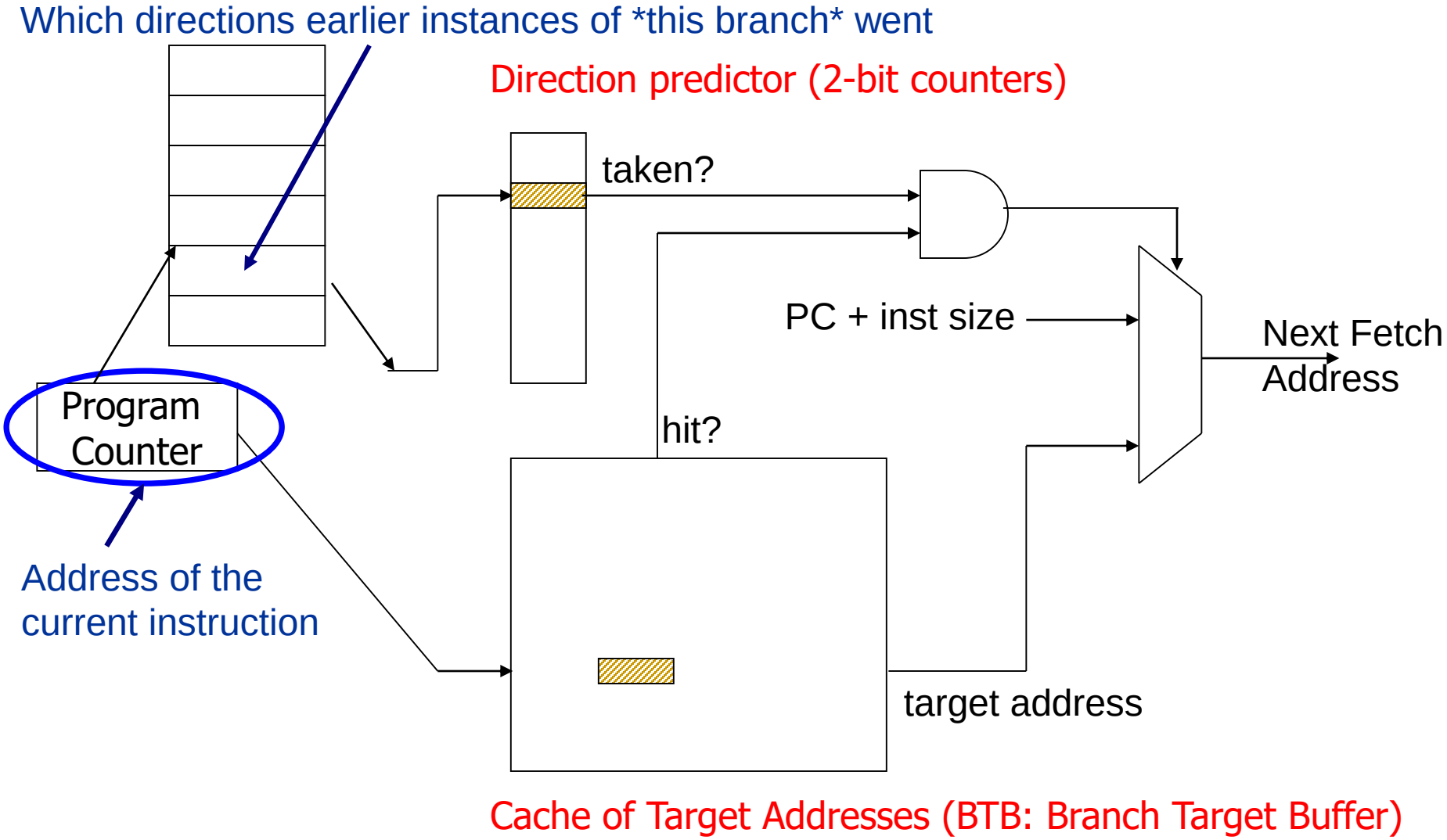
Two Level Local Branch Prediction

- First level: A set of local history registers (N bits each)
  - Select the history register based on the PC of the branch
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen



Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

Two-Level Local History Branch Predictor



Can We Do Even Better?

- Predictability of branches varies

- Some branches are more predictable using local history
- Some using global
- For others, a simple two-bit counter is enough
- Yet for others, a bit is enough

- Observation: There is heterogeneity in predictability behavior of branches
  - No one-size fits all branch prediction algorithm for all branches

- Idea: Exploit that heterogeneity by designing heterogeneous branch predictors

Hybrid Branch Predictors

- Idea: Use more than one type of predictor (i.e., multiple algorithms) and select the "best" prediction
  - E.g., hybrid of 2-bit counters and global predictor

- Advantages:

  + Better accuracy: different predictors are better for different branches

  + Reduced warmup time (faster-warmup predictor used until the slower-warmup predictor warms up)

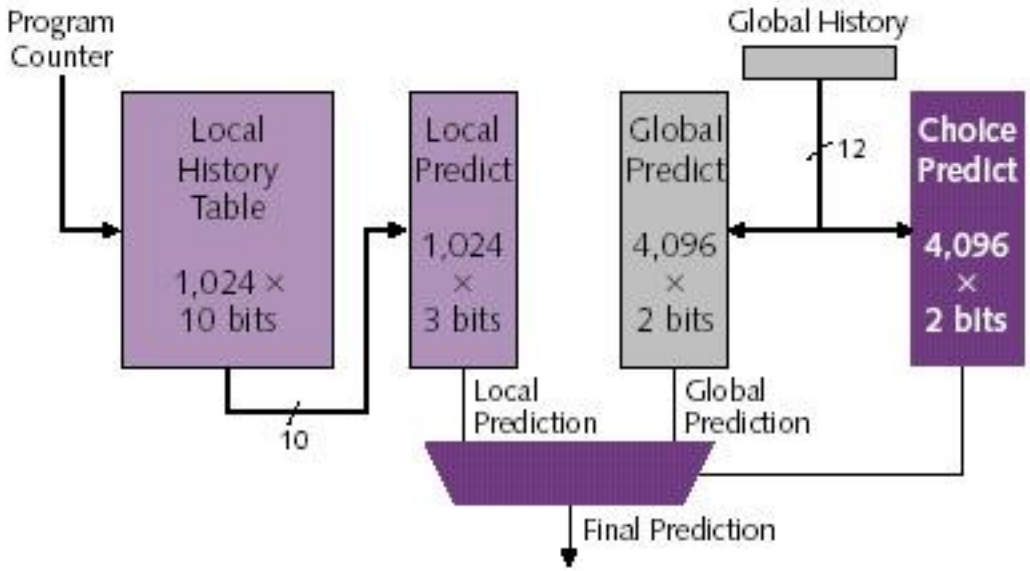
- Disadvantages:

  -- Need "meta-predictor" or "selector"

  -- Longer access latency

  - McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

Alpha 21264 Tournament Predictor



- Minimum branch penalty: 7 cycles
- Typical branch penalty: 11+ cycles
- 48K bits of target addresses stored in I-cache
- Predictor tables are reset on a context switch

- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

Are We Done w/ Branch Prediction?

- Hybrid branch predictors work well
  - E.g., 90-97% prediction accuracy on average

- Some "difficult" workloads still suffer, though!
  - E.g., gcc
  - Max IPC with tournament prediction: 9
  - Max IPC with perfect prediction: 35

Design of Digital Circuits

Lecture 18: Branch Prediction

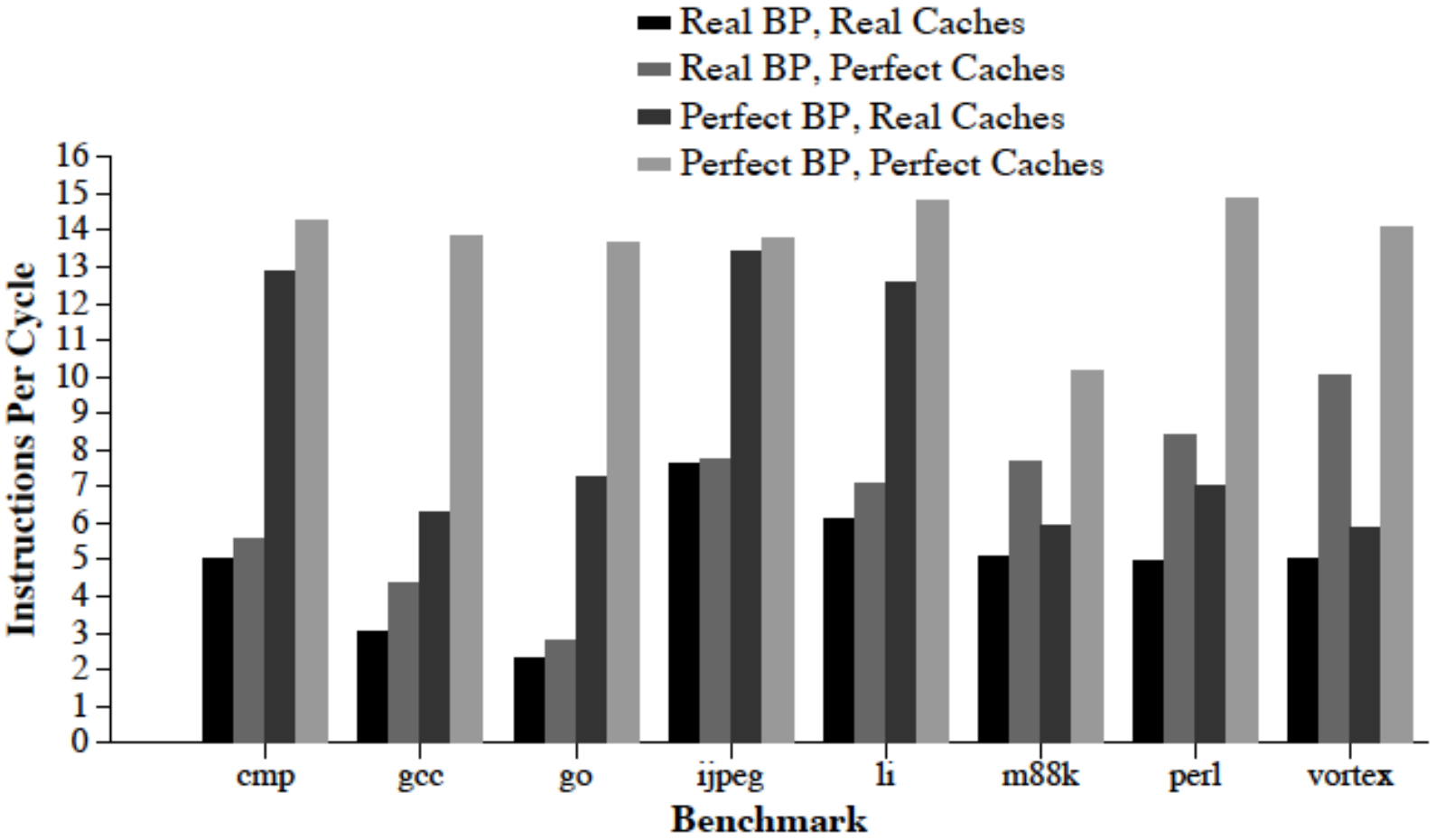
Prof. Onur Mutlu
ETH Zurich

Spring 2018

3 May 2018

We did not cover the following slides in lecture. These are for your benefit and curiosity.

Are We Done w/ Branch Prediction?



Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.

Some Other Branch Predictor Types

- Loop branch detector and predictor
  - Loop iteration count detector/predictor
  - Works well for loops with small number of iterations, where iteration count is predictable
  - Used in Intel Pentium M
- Perceptron branch predictor
  - Learns the *direction correlations* between individual branches
  - Assigns weights to correlations
  - Jimenez and Lin, "Dynamic Branch Prediction with Perceptrons," HPCA 2001.
- Hybrid history length based predictor
  - Uses different tables with different history lengths
  - Seznec, "Analysis of the O-Geometric History Length branch predictor," ISCA 2005.

Intel Pentium M Predictors

The advanced branch prediction in the Pentium M processor is based on the Intel Pentium® 4 processor's [6] branch predictor. On top of that, two additional predictors to capture special program flows, were added: a Loop Detector and an Indirect Branch Predictor.

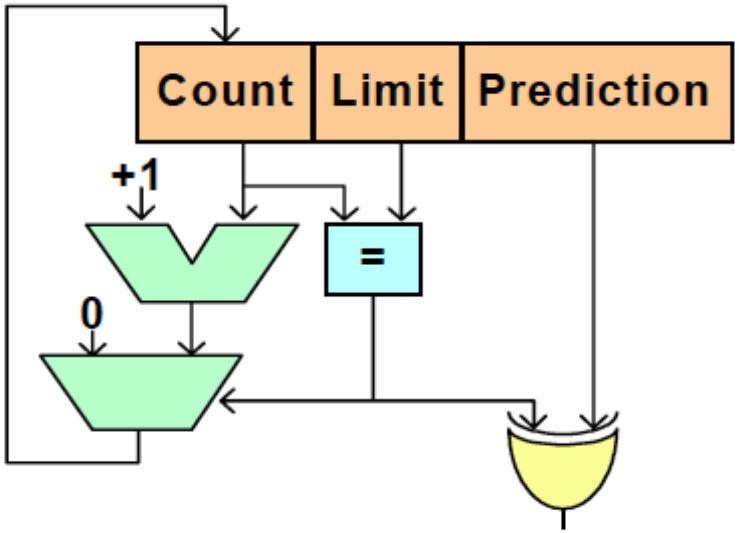


Figure 2: The Loop Detector logic

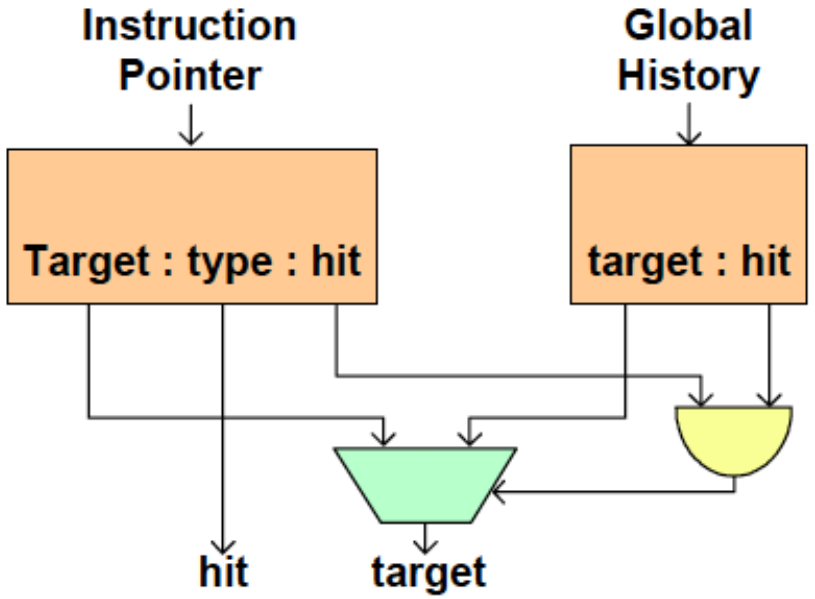
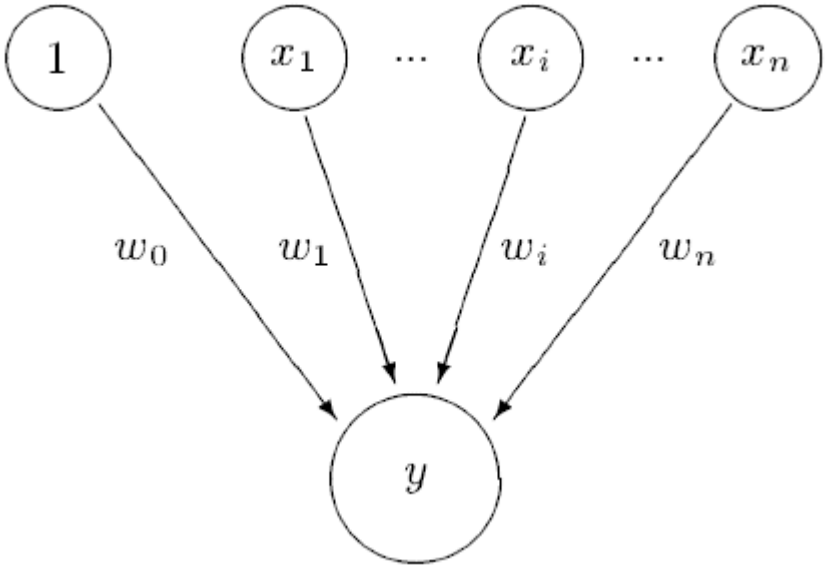


Figure 3: The Indirect Branch Predictor logic

Gochman et al.,
"The Intel Pentium M Processor: Microarchitecture and Performance,"
Intel Technology Journal, May 2003.

Perceptron Branch Predictor (I)

- Idea: Use a perceptron to learn the correlations between branch history register bits and branch outcome
- A perceptron learns a target Boolean function of N inputs



$$y = w_0 + \sum_{i=1}^{n} x_i w_i.$$

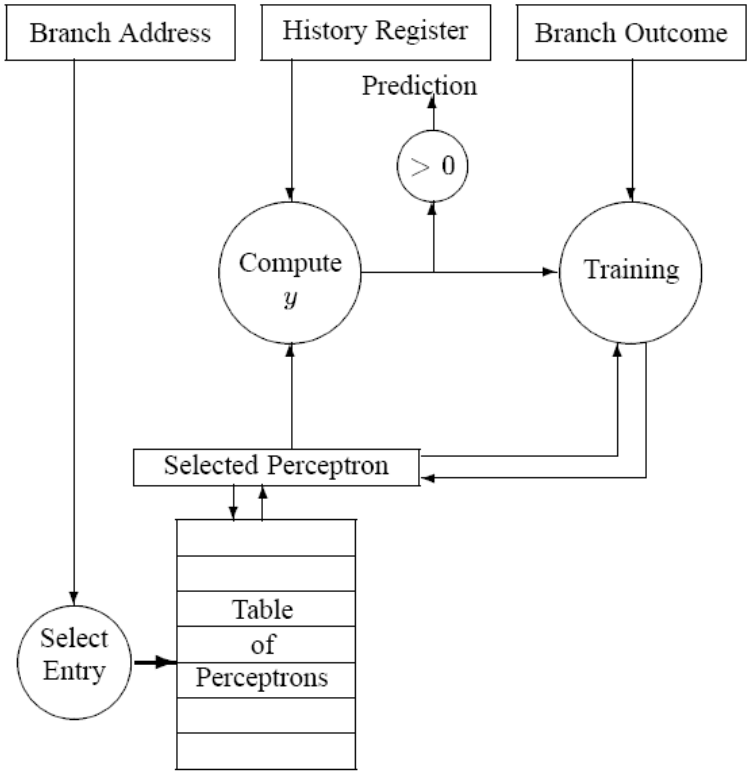
Each branch associated with a perceptron

A perceptron contains a set of weights wi
→ Each weight corresponds to a bit in the GHR
→How much the bit is correlated with the direction of the branch
→ Positive correlation: large + weight
→ Negative correlation: large - weight

Prediction:
→ Express GHR bits as 1 (T) and -1 (NT)
→ Take dot product of GHR and weights
→ If output > 0, predict taken

- Jimenez and Lin, "Dynamic Branch Prediction with Perceptrons," HPCA 2001.
- Rosenblatt, "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms," 1962

Perceptron Branch Predictor (II)



Prediction function:

Dot product of GHR and perceptron weights

$$y = w_0 + \sum_{i=1}^{n} x_i w_i$$

Output compared to 0

Bias weight (bias of branch independent of the history)

Training function:

```
if sign(y_out) ≠ t or |y_out| ≤ θ then
        for i := 0 to n do
                w_i := w_i + t x_i
        end for
end if
```

Perceptron Branch Predictor (III)

- Advantages

  + More sophisticated learning mechanism → better accuracy

- Disadvantages

  -- Hard to implement (adder tree to compute perceptron output)

  -- Can learn only linearly-separable functions

  e.g., cannot learn XOR type of correlation between 2 history bits and branch outcome

Prediction Using Multiple History Lengths

- Observation: Different branches require different history lengths for better prediction accuracy

- Idea: Have multiple PHTs indexed with GHRs with different history lengths and intelligently allocate PHT entries to different branches

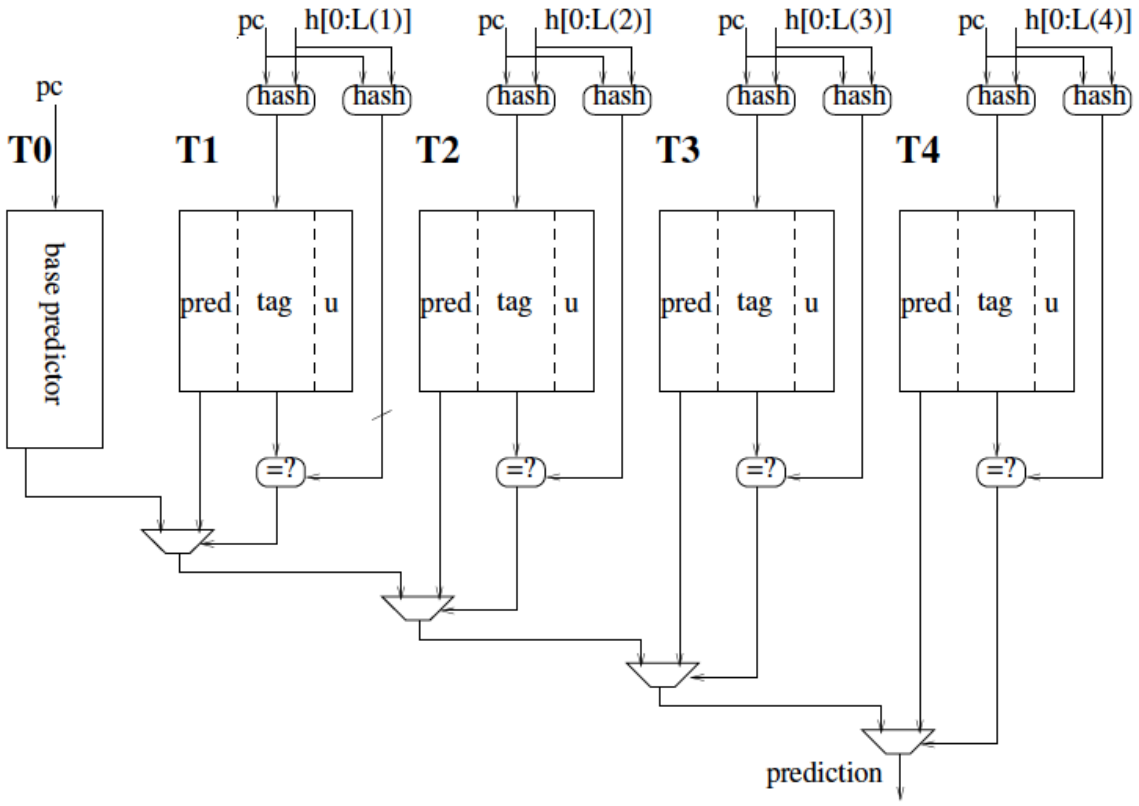


Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

Seznec and Michaud, "A case for (partially) tagged Geometric History Length Branch Prediction," JILP 2006.

State of the Art in Branch Prediction

- See the Branch Prediction Championship
  - https://www.jilp.org/cbp2016/program.html

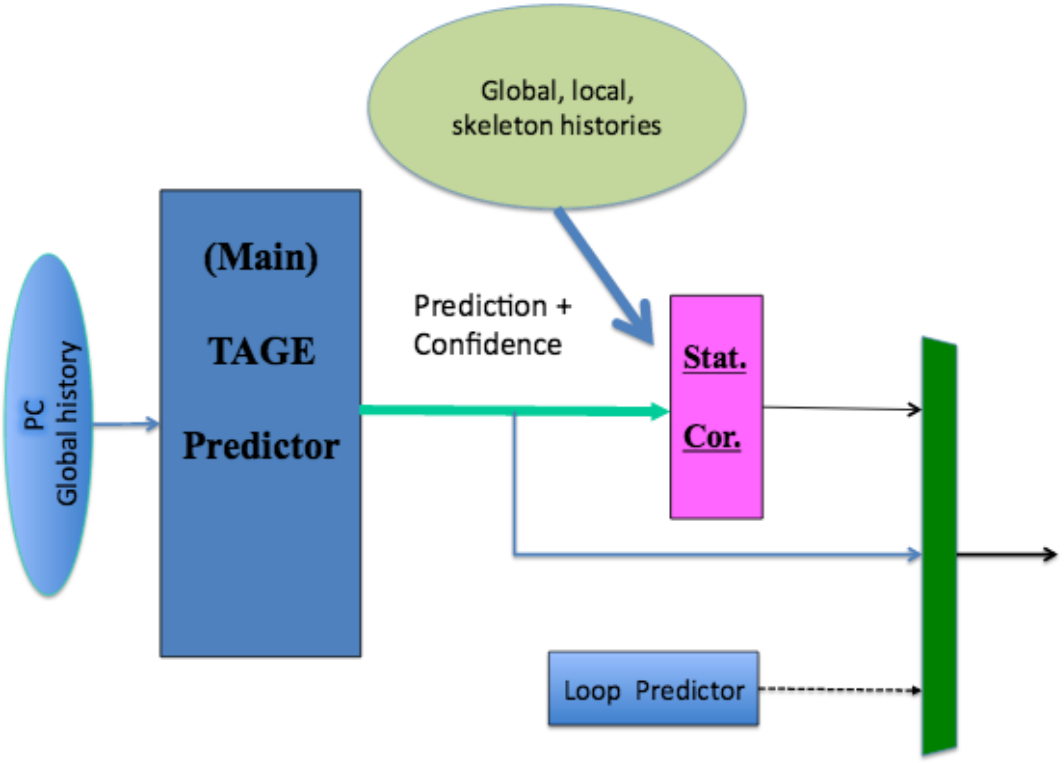


Figure 1. The TAGE-SC-L predictor: a TAGE predictor backed with a Statistical Corrector predictor and a loop predictor

Andre Seznec, "TAGE-SC-L branch predictors," CBP 2014.

Andre Seznec, "TAGE-SC-L branch predictors again," CBP 2016.

Branch Confidence Estimation

- Idea: Estimate if the prediction is likely to be correct
  - i.e., estimate how "confident" you are in the prediction
- Why?
  - Could be very useful in deciding how to speculate:
    - What predictor/PHT to choose/use
    - Whether to keep fetching on this path
    - Whether to switch to some other way of handling the branch, e.g. dual-path execution (eager execution) or dynamic predication
    - …
- Jacobsen et al., "Assigning Confidence to Conditional Branch Predictions," MICRO 1996.