

# Digital Design & Computer Arch.

## Lecture 17b: Branch Prediction I

Prof. Onur Mutlu

ETH Zürich

Spring 2021

30 April 2021

# Required Readings

---

## ■ This week

- ❑ Smith and Sohi, “The Microarchitecture of Superscalar Processors,” Proceedings of the IEEE, 1995
- ❑ H&H Chapters 7.8 and 7.9
- ❑ McFarling, “Combining Branch Predictors,” DEC WRL Technical Report, 1993.

# 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

# Approaches to (Instruction-Level) Concurrency

---

- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- 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

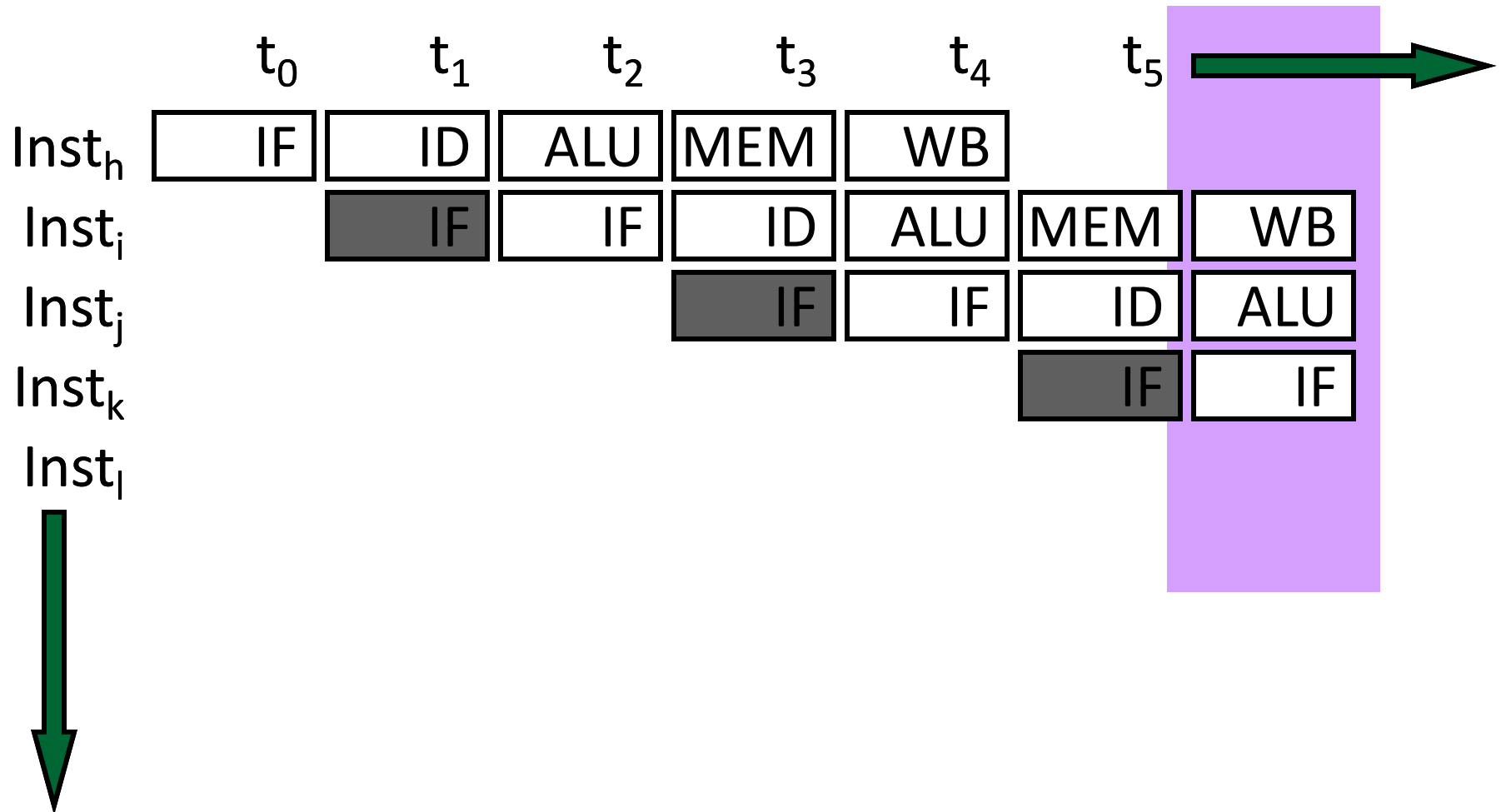
# How to Handle Control Dependences

---

- 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!

# 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)

# 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)

# 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  $N \times W$  wasted instruction slots

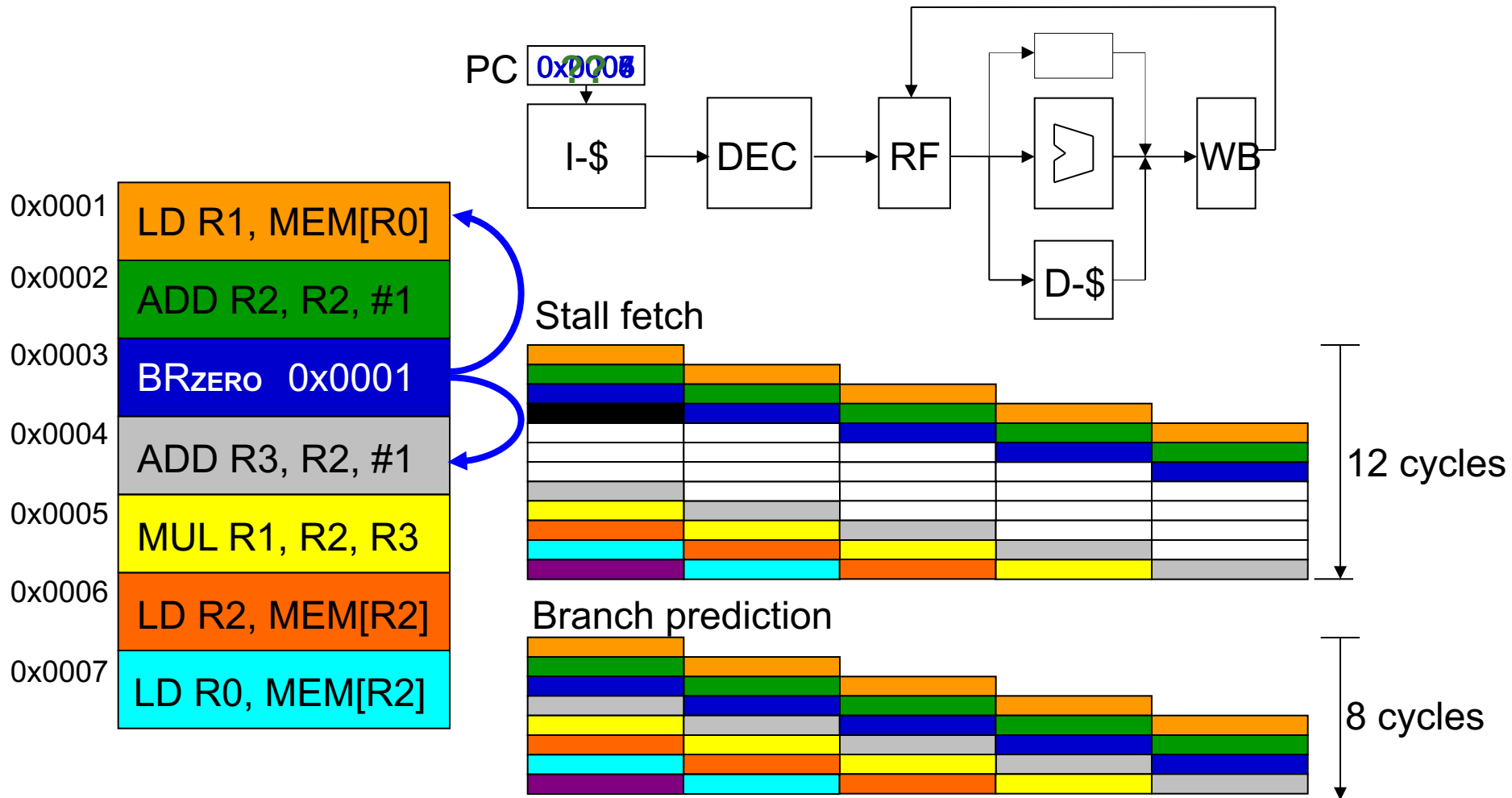
# Importance of The Branch Problem

---

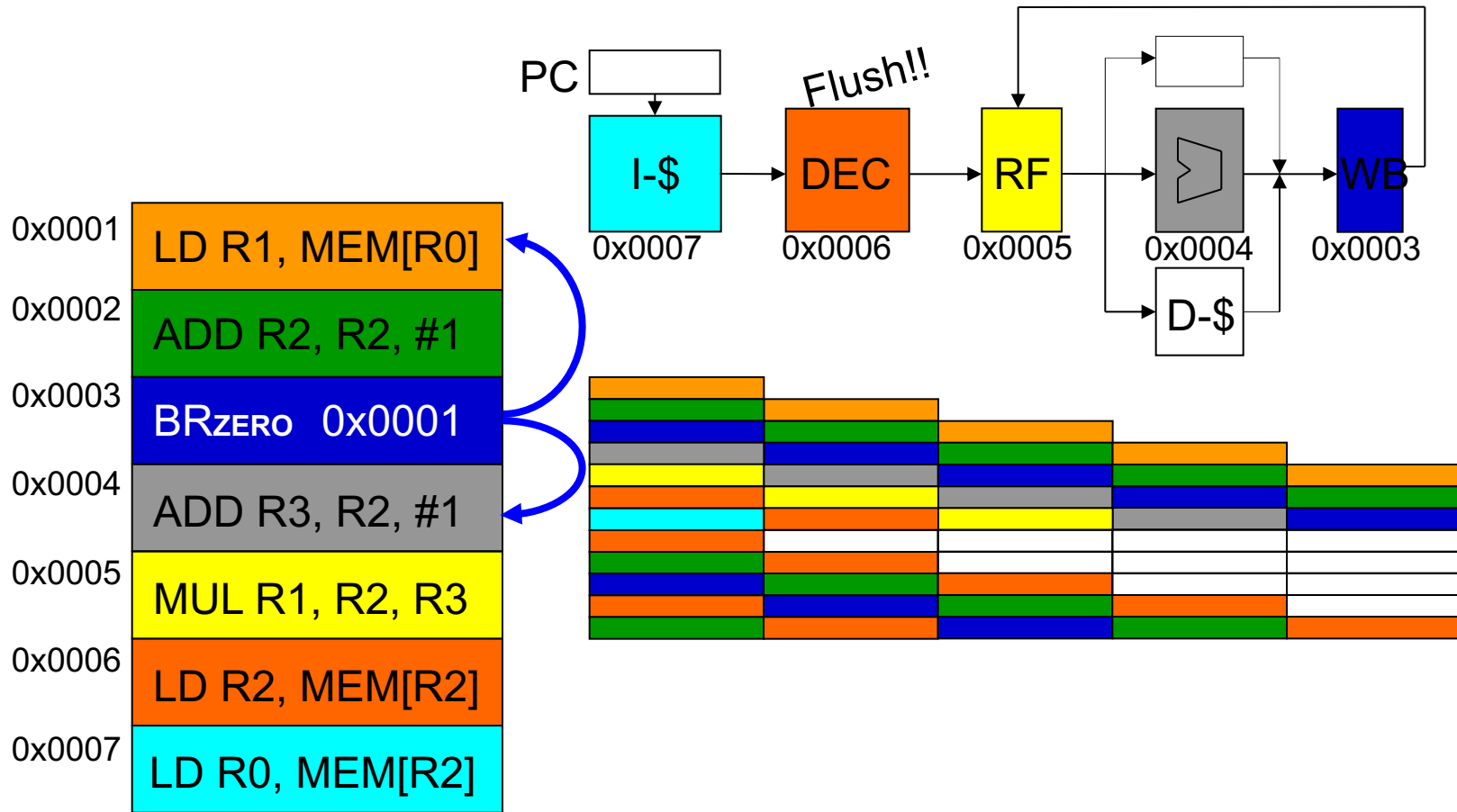
- Assume  $N = 20$  (20 pipe stages),  $W = 5$  (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch
  
- How long does it take to fetch 500 instructions?
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - 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 $\text{NextPC} = \text{PC} + 4$

---

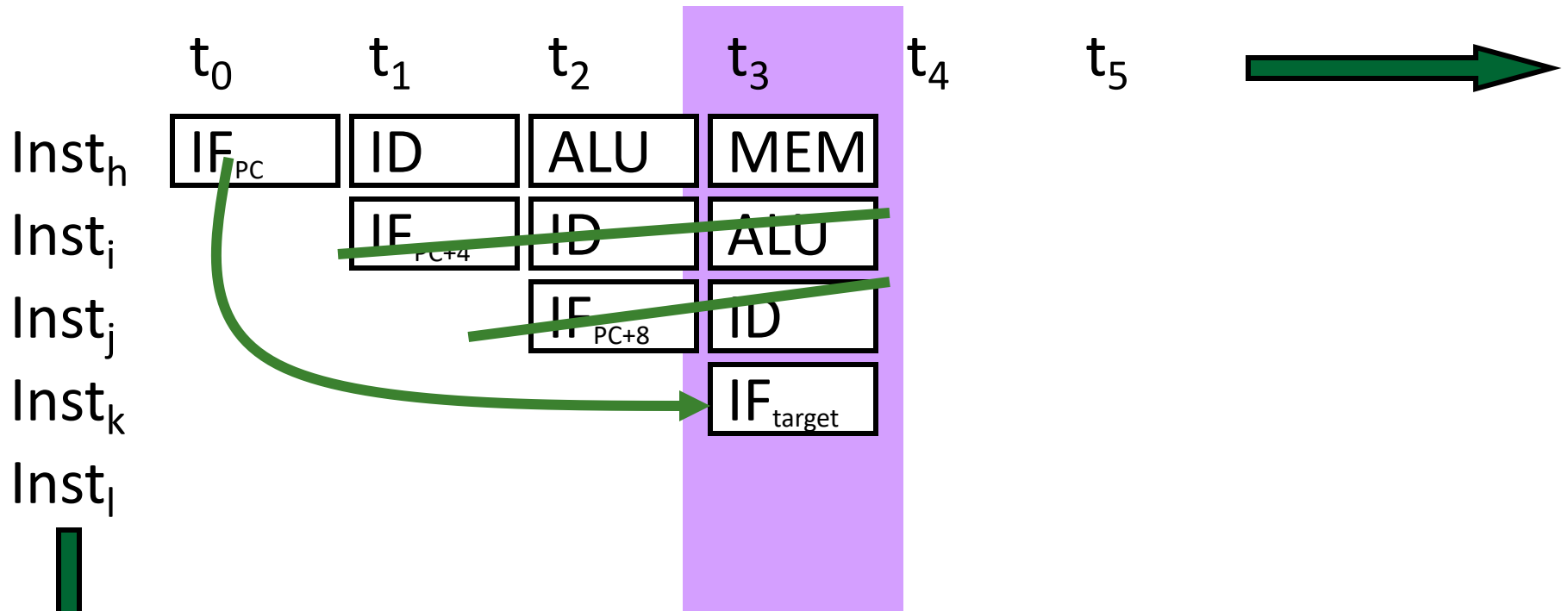
- Always predict the next sequential instruction is the next instruction to be executed
- This is a form of **next fetch address prediction** (and branch prediction)
- How can you make this more effective?
- Idea: **Maximize the chances that the next sequential instruction is the next instruction to 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) and test only once
  2. Convert control dependences into data dependences → predicated execution (aka if-conversion)

# Branch Prediction: Always PC+4

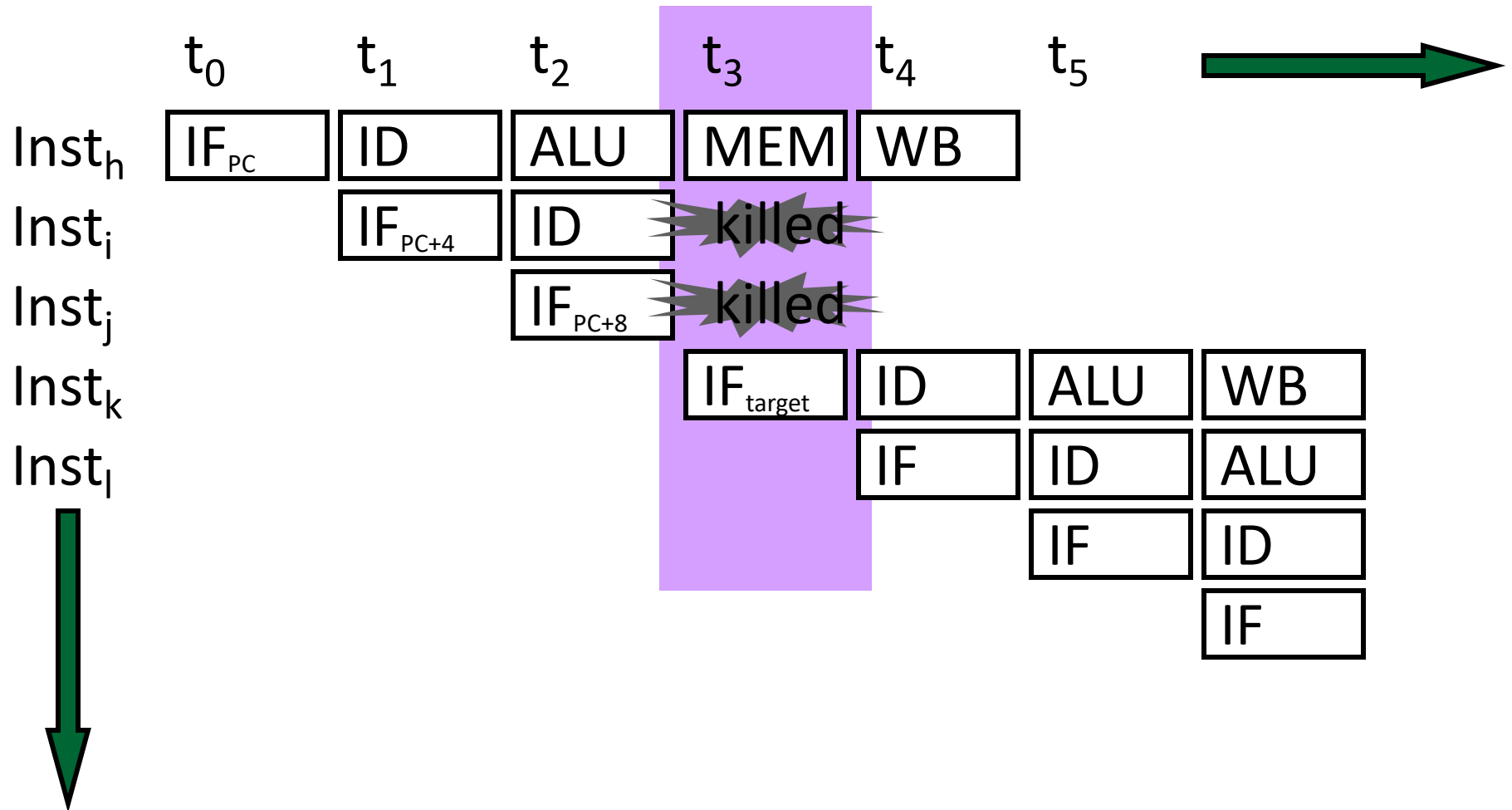


$Inst_h$  is a branch

When a branch resolves

- branch target ( $Inst_k$ ) is fetched
- all instructions fetched since  $inst_h$  (so called “wrong-path” instructions) must be flushed

# Pipeline Flush on a Misprediction



$Inst_h$  is a branch

# Performance Analysis

---

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

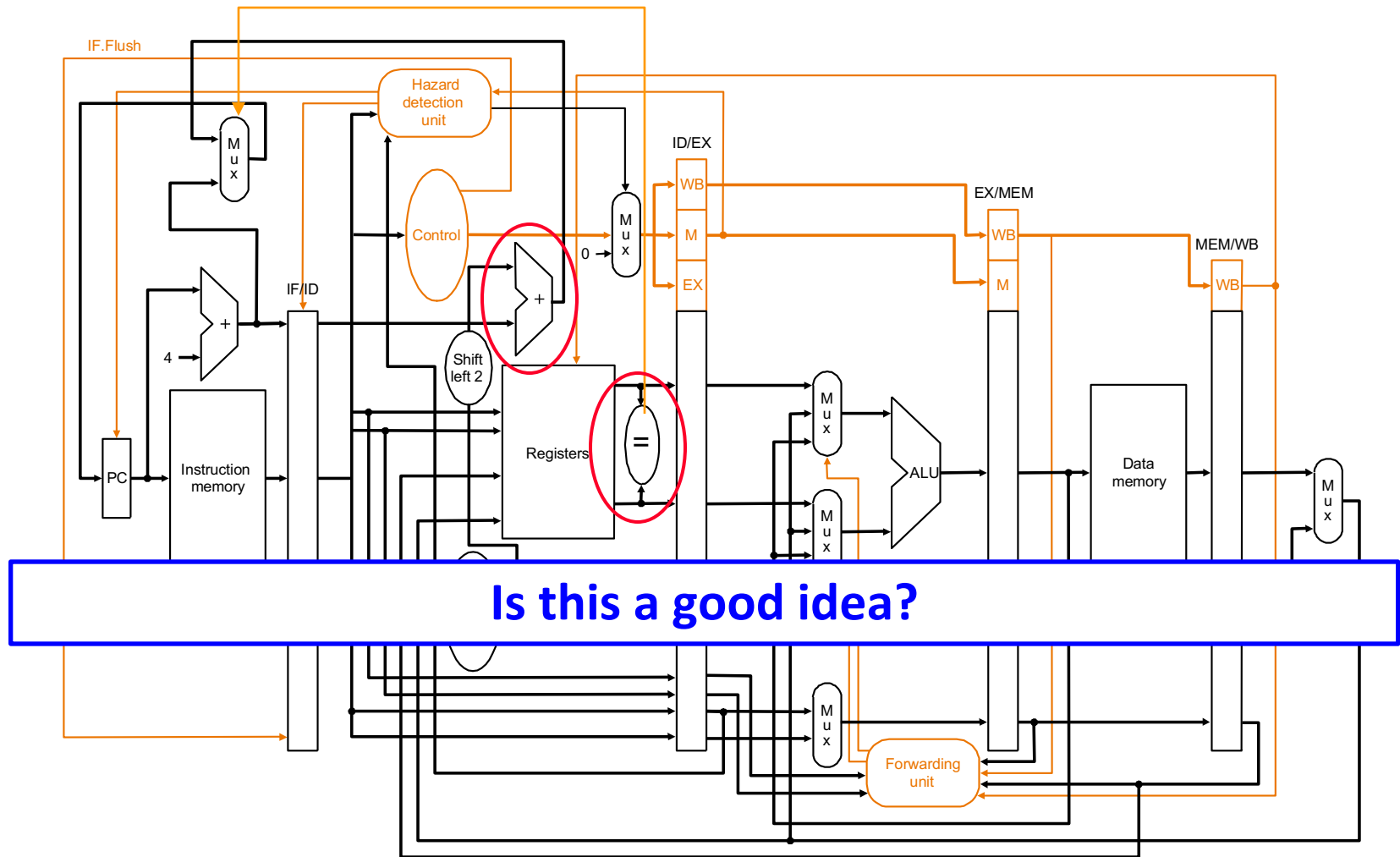
probability of  
a wrong guess

penalty for  
a wrong guess

Can we reduce either of the two penalty terms?

# Reducing Branch Misprediction Penalty

- Resolve branch condition and target address early



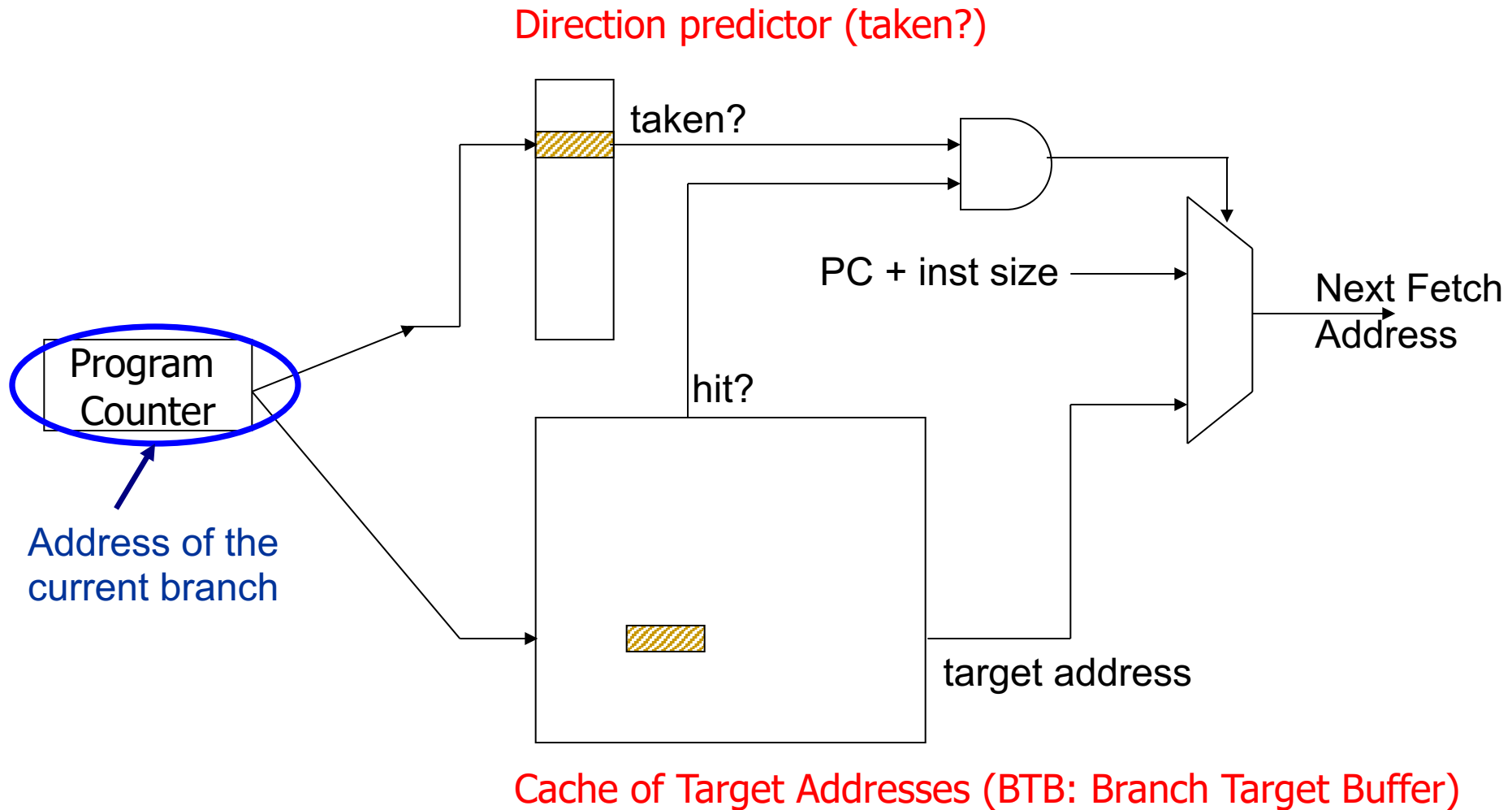
Is this a good idea?

# Branch Prediction (A Bit More Enhanced)

---

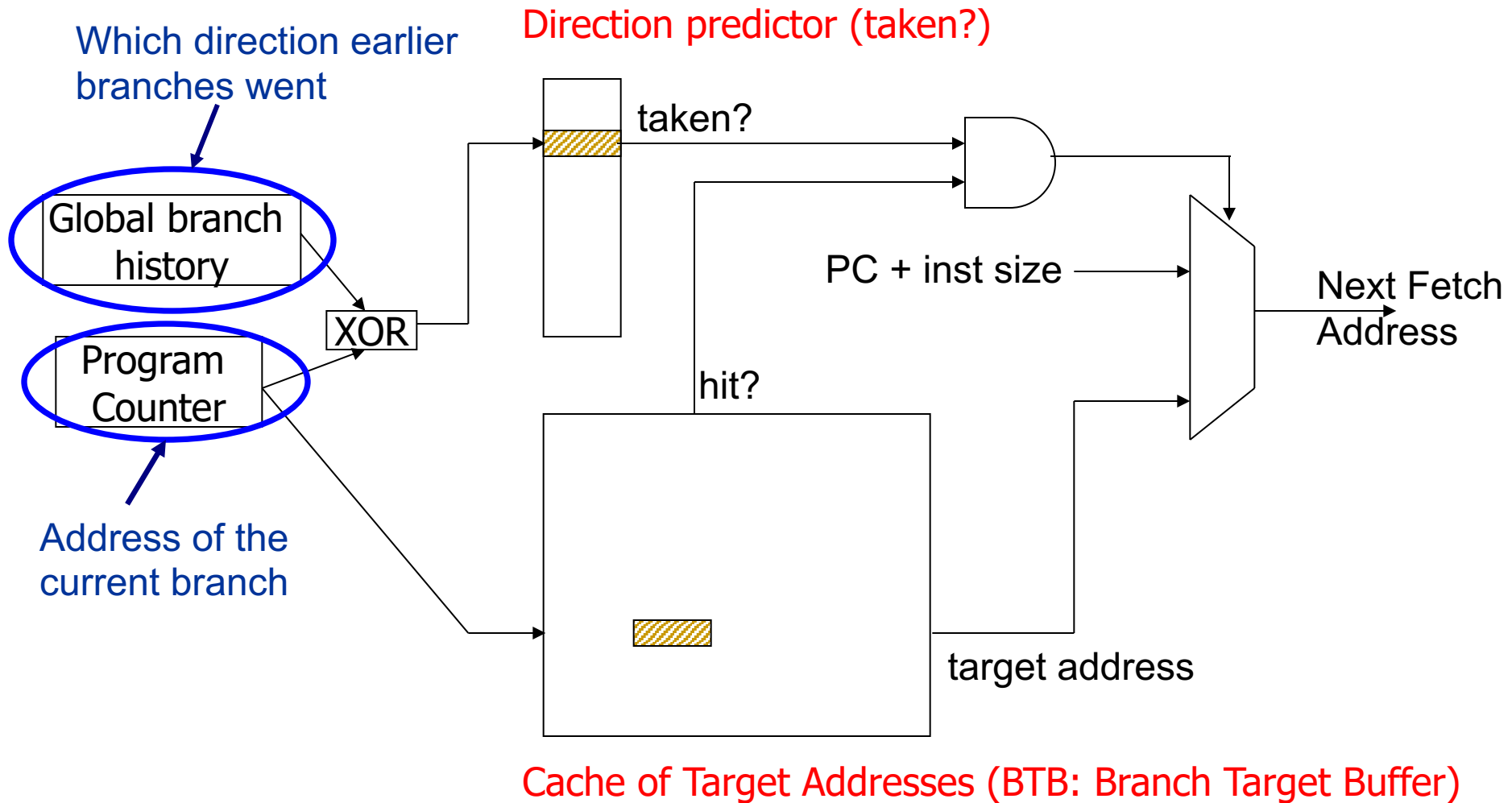
- Idea: Predict the next fetch address (to be used in the next cycle)
- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)
- Observation: Target address remains the same for a conditional 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, geometric history)

# 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:
  - TTTTTTTTTTTTNNNNNNNNNNNN → 50% accuracy
  - TNTNTNTNTNTNTNTNTNTNTN → 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
      - 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)

# Digital Design & Computer Arch.

## Lecture 17b: Branch Prediction I

Prof. Onur Mutlu

ETH Zürich

Spring 2021

30 April 2021

We Will Cover Later Slides  
in Future Lectures.

# 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 =  $(N-2)/N$

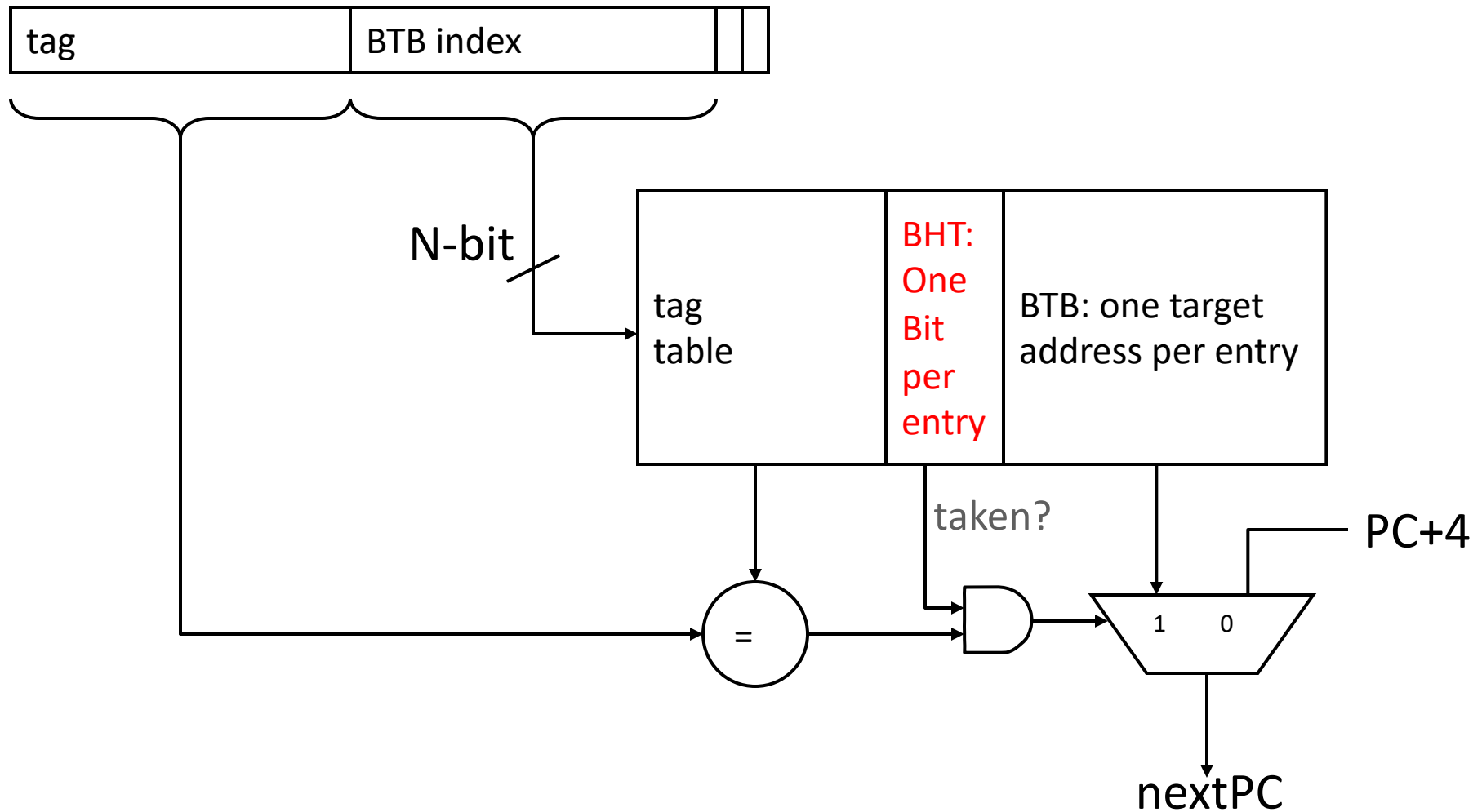
+ Loop branches for loops with large N (number of iterations)

-- Loop branches for loops with small N (number of iterations)

TNTNTNTNTNTNTNTNTN → 0% accuracy



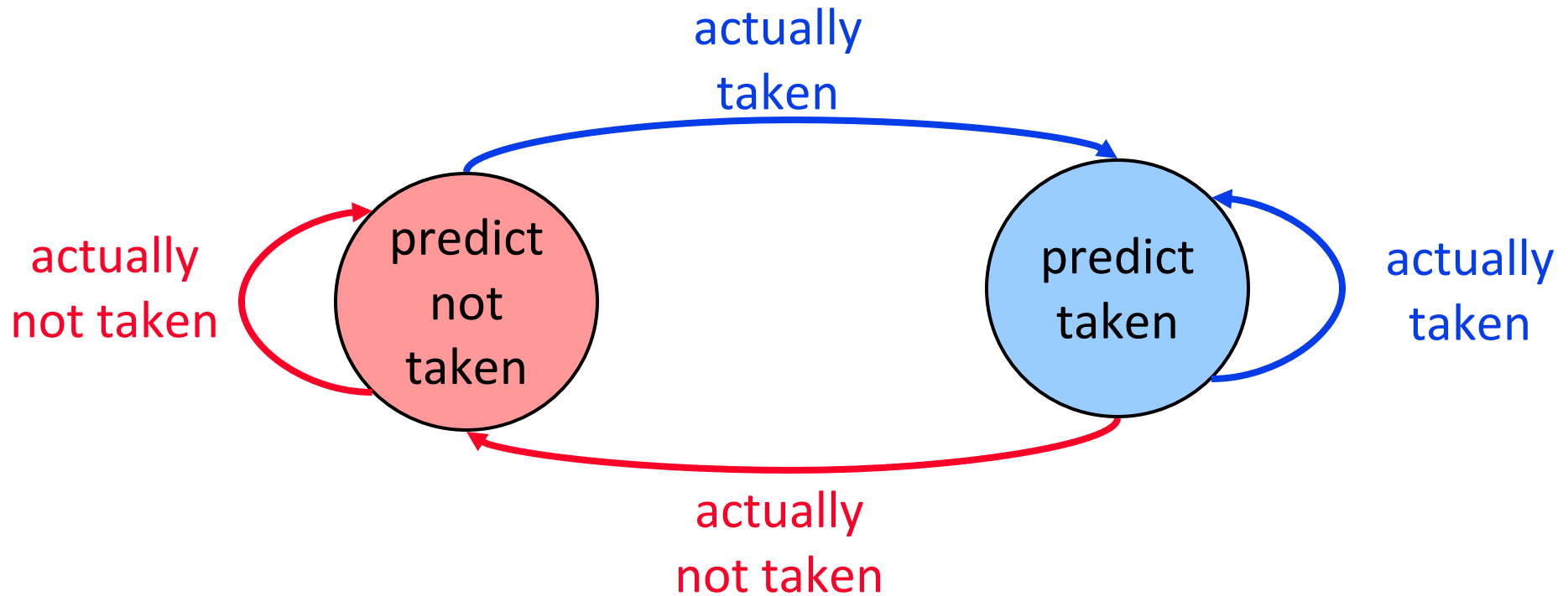
# Implementing the Last-Time Predictor



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  $T \rightarrow NT$  or  $NT \rightarrow T$  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.

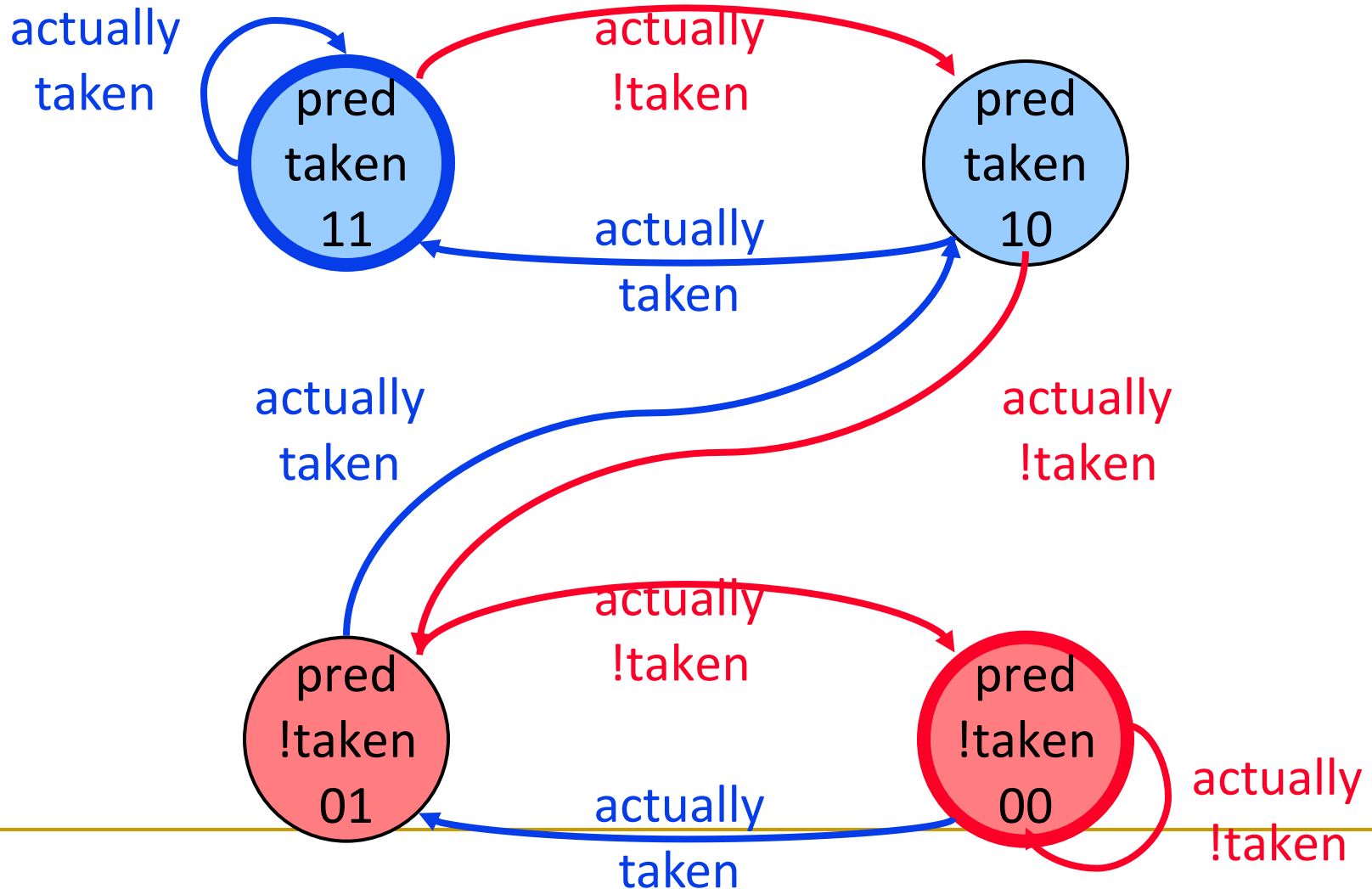
# Two-Bit Counter Based Prediction

---

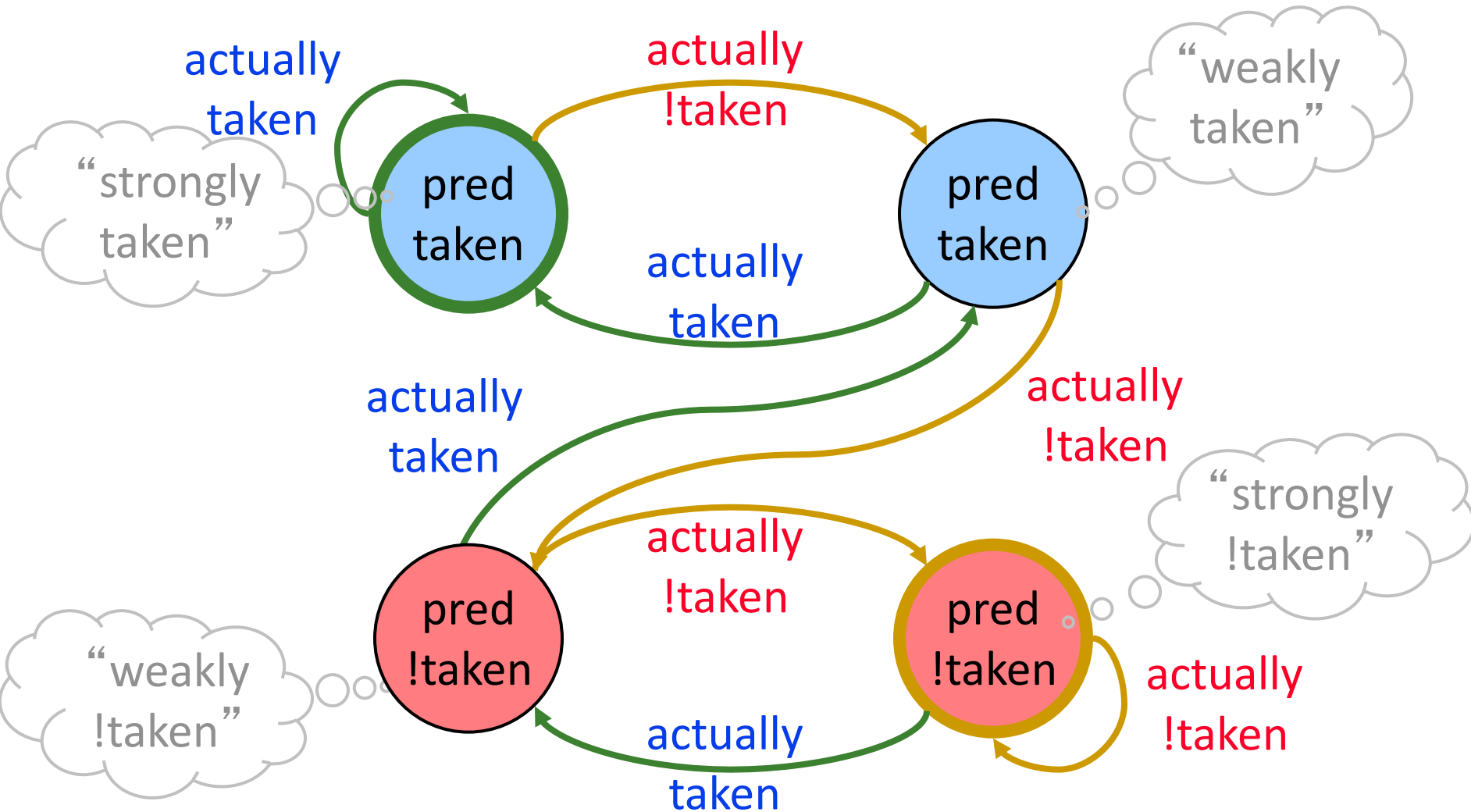
- Each branch associated with a two-bit counter (2BC)
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome
- Also called **bimodal prediction**

# 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

# 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 =  $(N-1)/N$   
TNTNTNTNTNTNTNTNTN → 50% accuracy  
(assuming counter initialized to weakly taken)
- + Better prediction accuracy
- More hardware cost (but counter can be part of a BTB entry)

# 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

# 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

if (x<1) ...	branch Y: if (cond1) a = 2;
if (x>1) ...	...
	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 workload
  - Generates truth table from Boolean expression

```
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

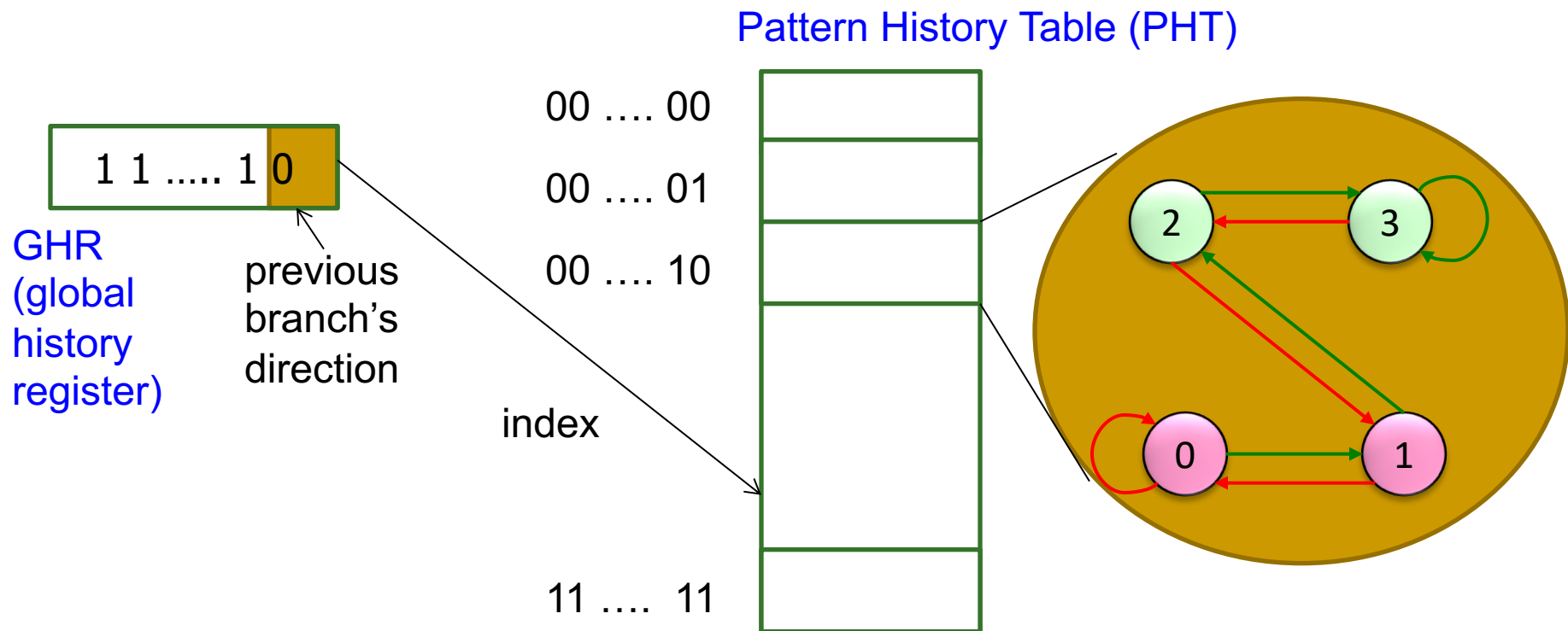
# Capturing Global Branch Correlation

---

- 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)

# 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



# 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 3 branches test j  
History: TTTN  
Predict taken for i  
Next history: TTNT  
(shift in last outcome)

- McFarling, “Combining Branch Predictors,” DEC WRL TR 1993.



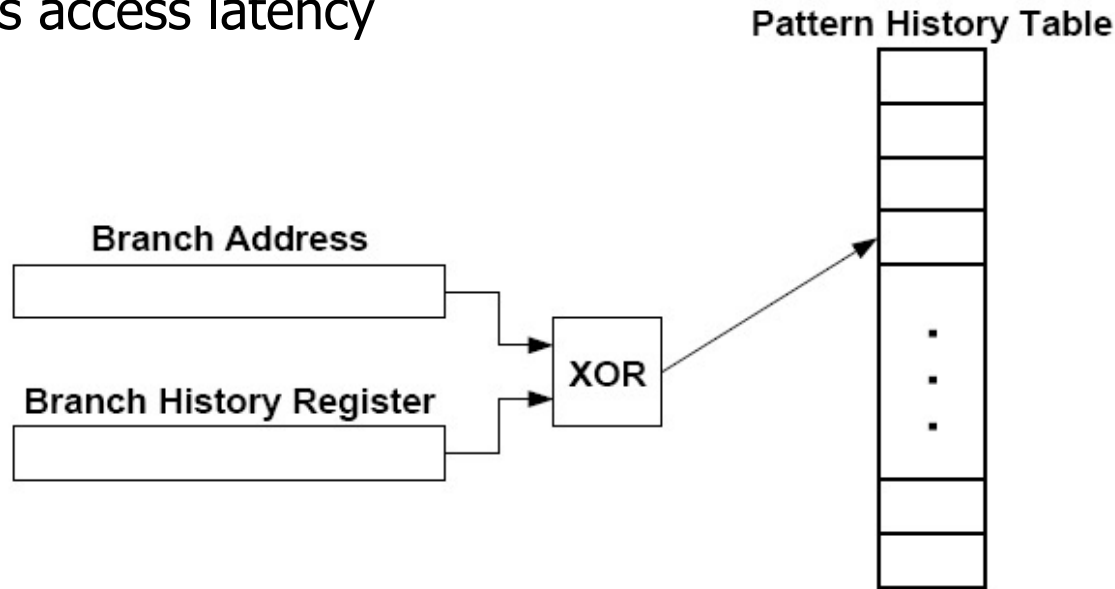
# Intel Pentium Pro Branch Predictor

---

- 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
- First widely commercially successful out-of-order execution machine

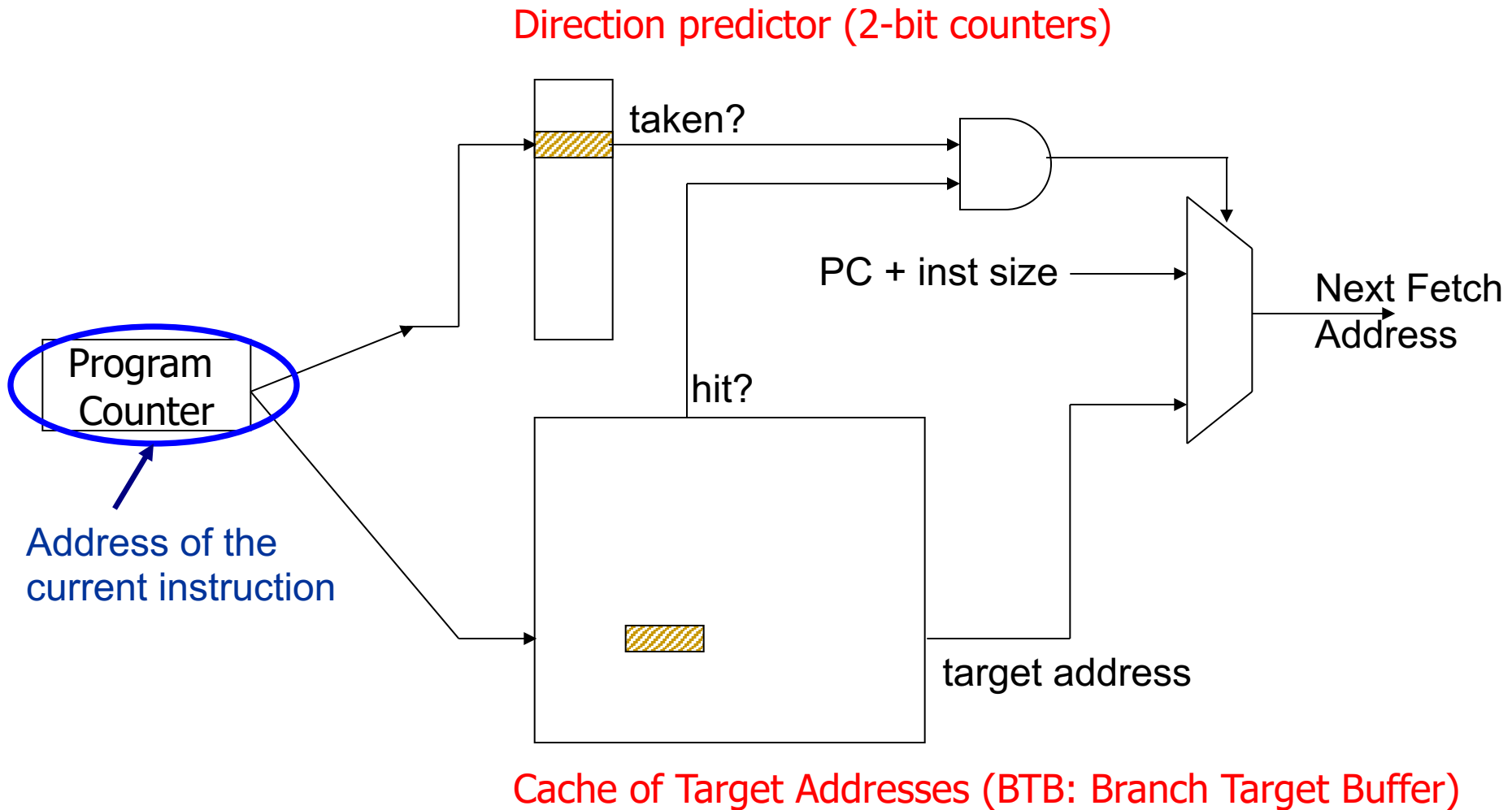
# Improving Global Predictor Accuracy

- 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 used for prediction
    - + Better utilization of the two-bit counter array
    - Increases access latency

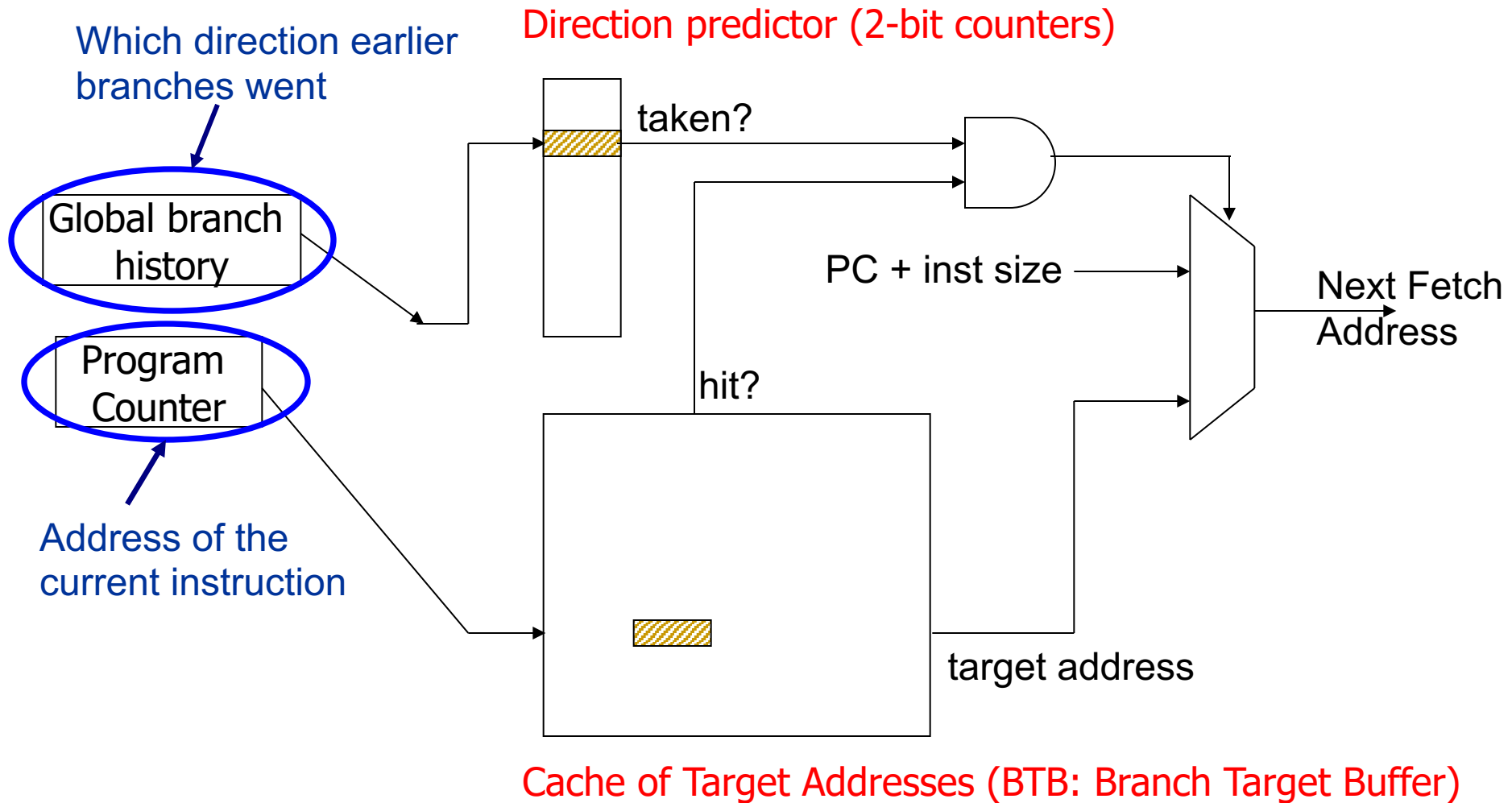


- McFarling, “Combining Branch Predictors,” DEC WRL Tech Report, 1993.

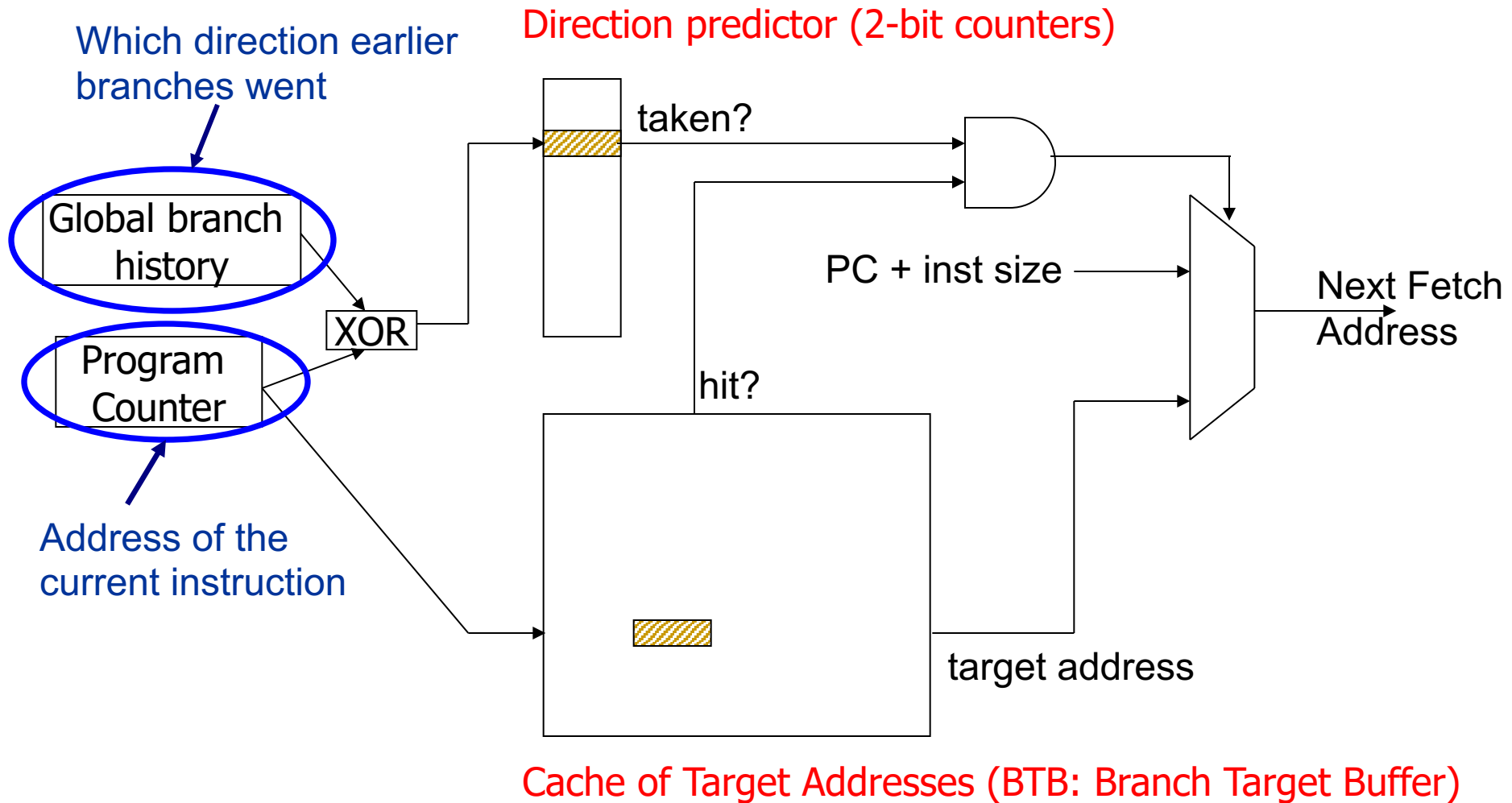
# Review: One-Level Branch Predictor



# 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

# 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)<sup>n</sup>, 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.

# 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.

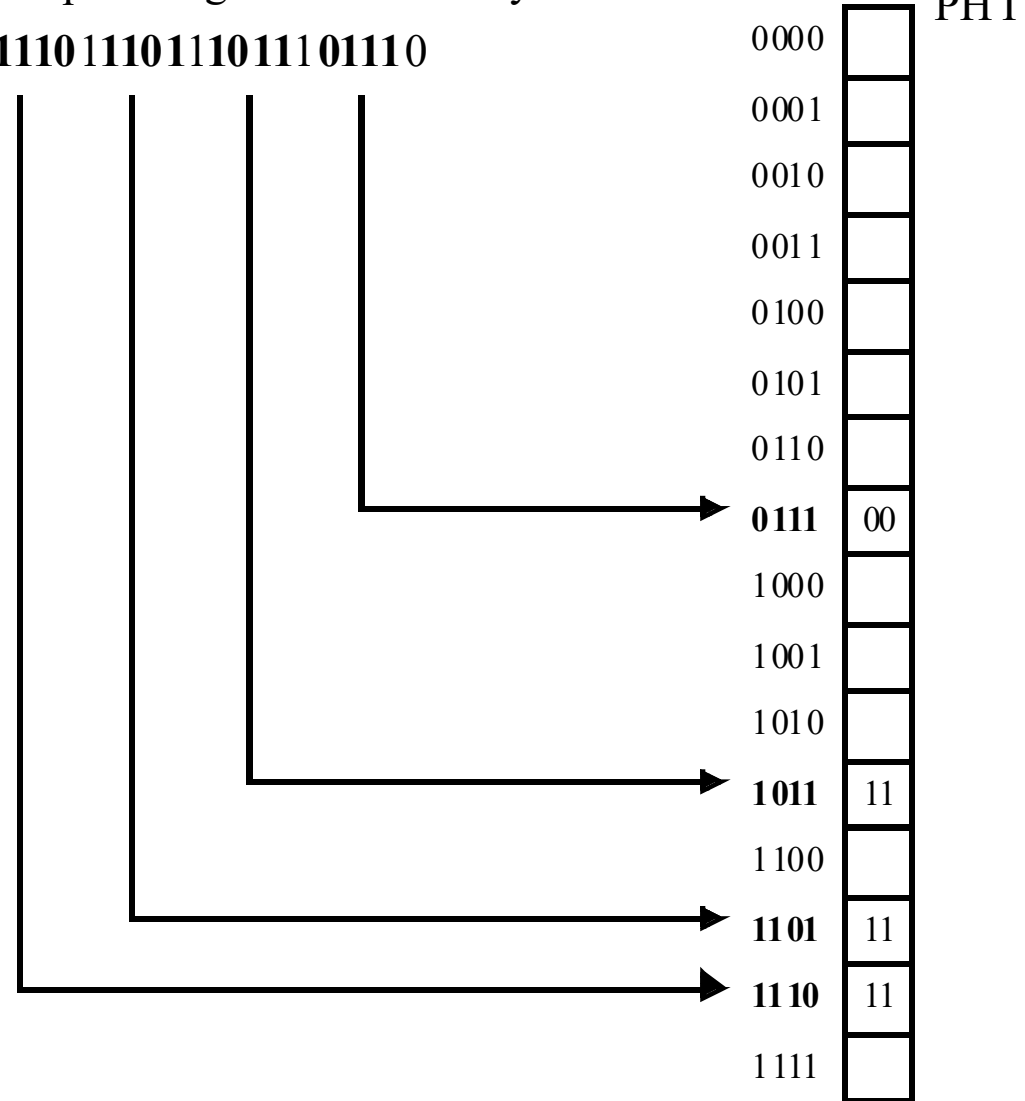


# More Motivation for Local History

- 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

Loop closing branch's history

11101110111011101110



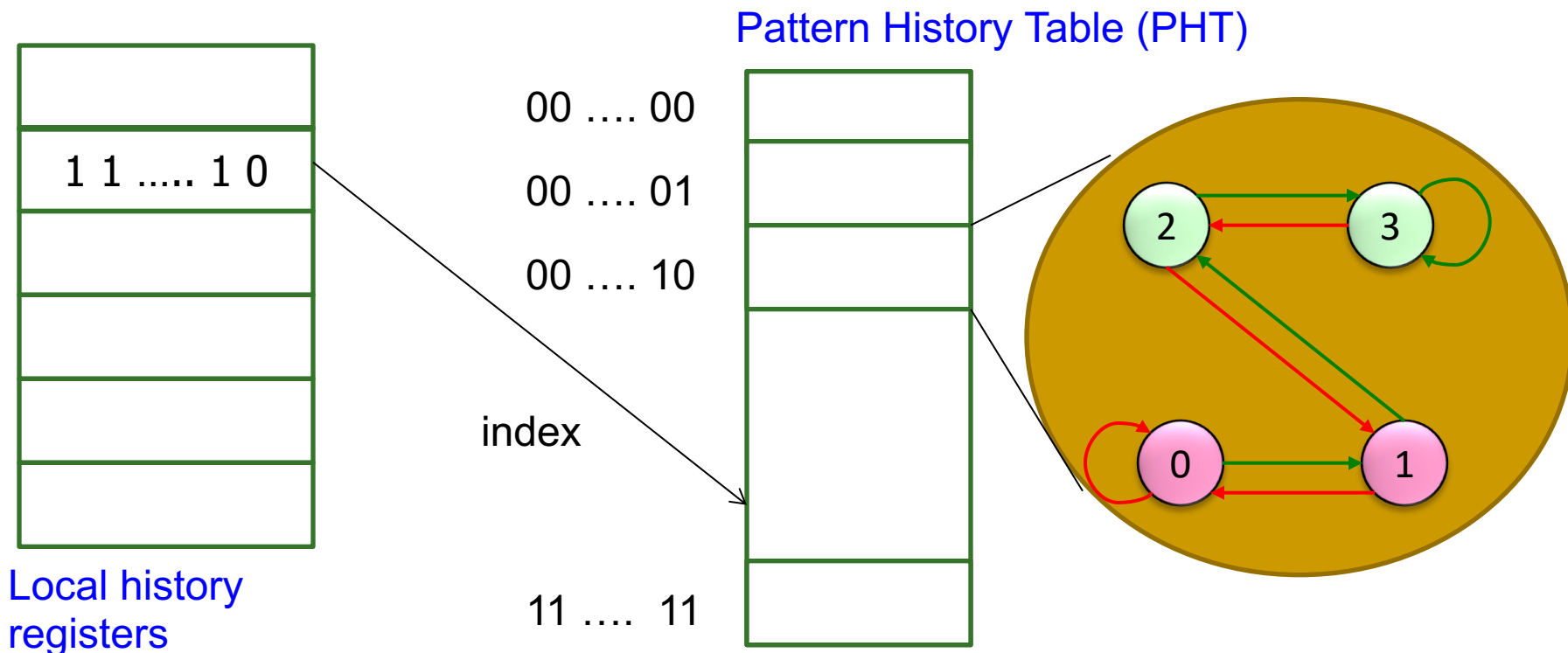
# Capturing Local Branch Correlation

---

- 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)

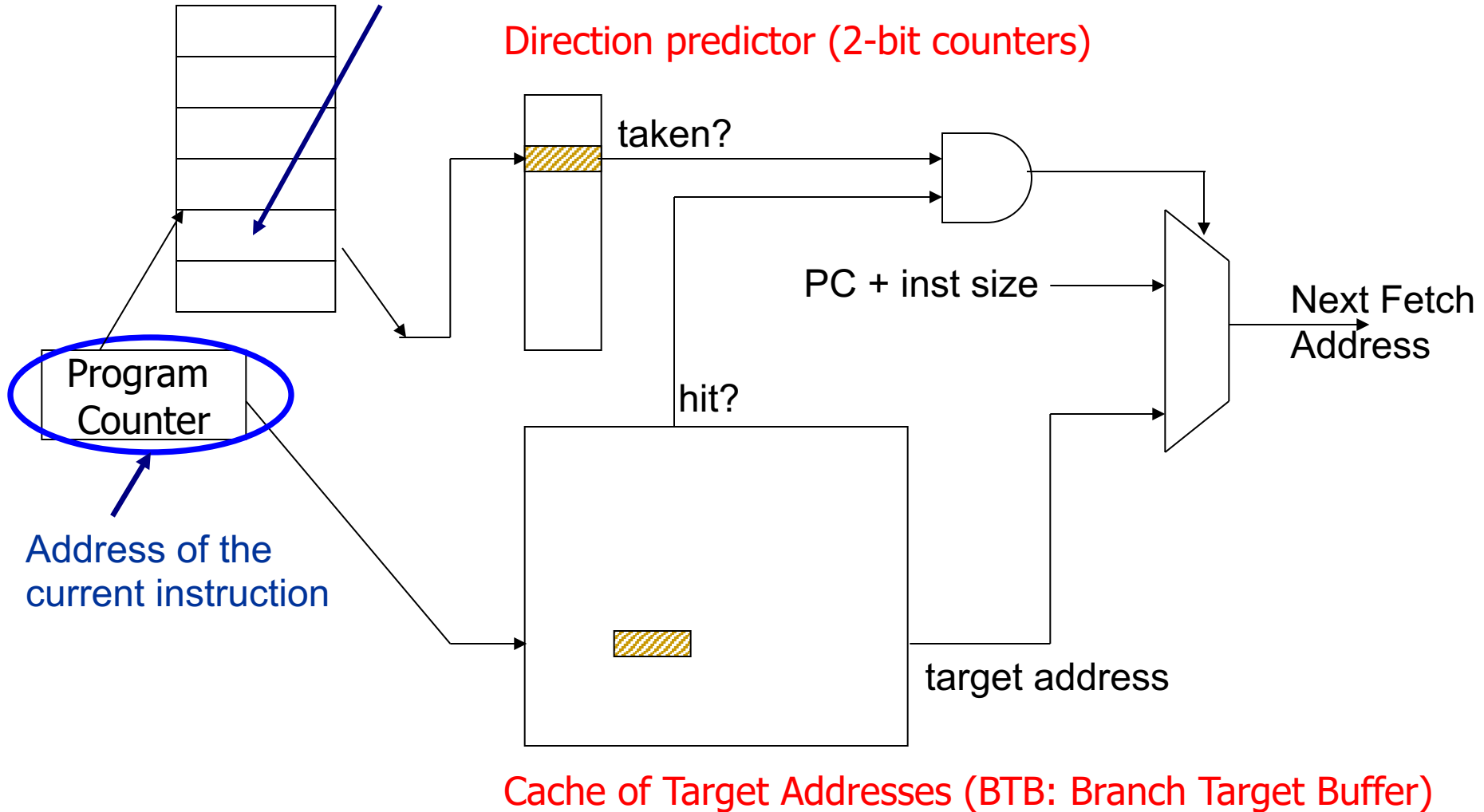
# Two Level Local Branch Prediction

- 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



# Two-Level Local History Branch Predictor

Which directions earlier instances of \*this branch\* went



# Can We Do Even Better?

---

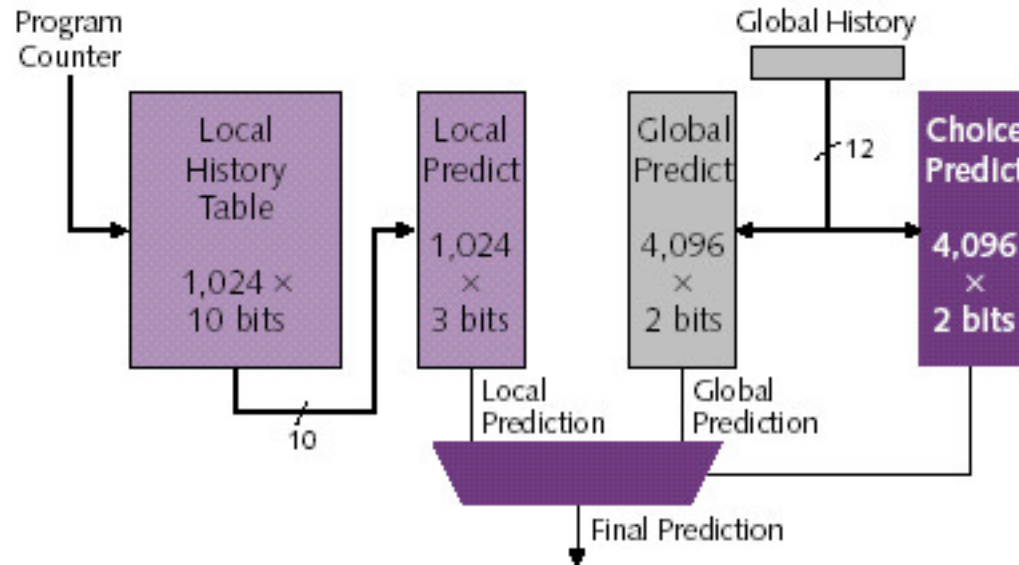
- Predictability of branches varies
- Some branches are more predictable using local history
- Some branches are more predictable using global
- For others, a simple two-bit counter is enough
- Yet for others, a single bit is enough
- Observation: There is heterogeneity in predictability behavior of branches
  - No one-size fits all branch prediction algorithm for all branches
- Idea: Exploit that heterogeneity by designing heterogeneous (hybrid) 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” to decide which predictor to use
  - 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



# 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: Loop and Jump

The advanced branch prediction in the Pentium M processor is based on the Intel Pentium<sup>®</sup> 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.

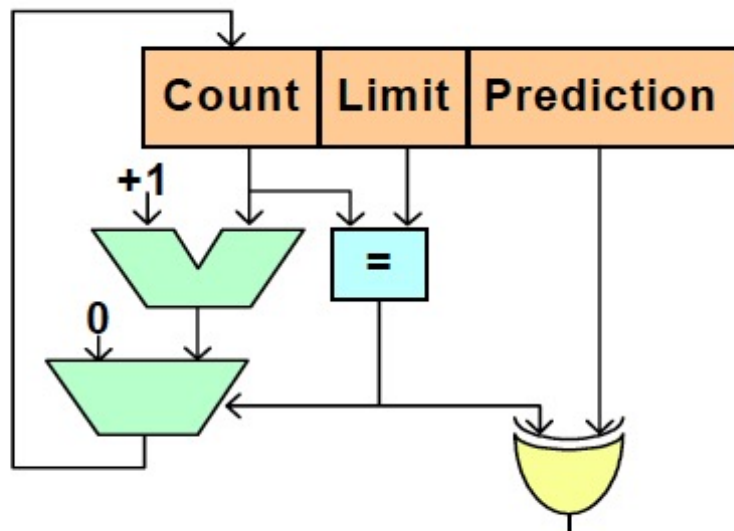


Figure 2: The Loop Detector logic

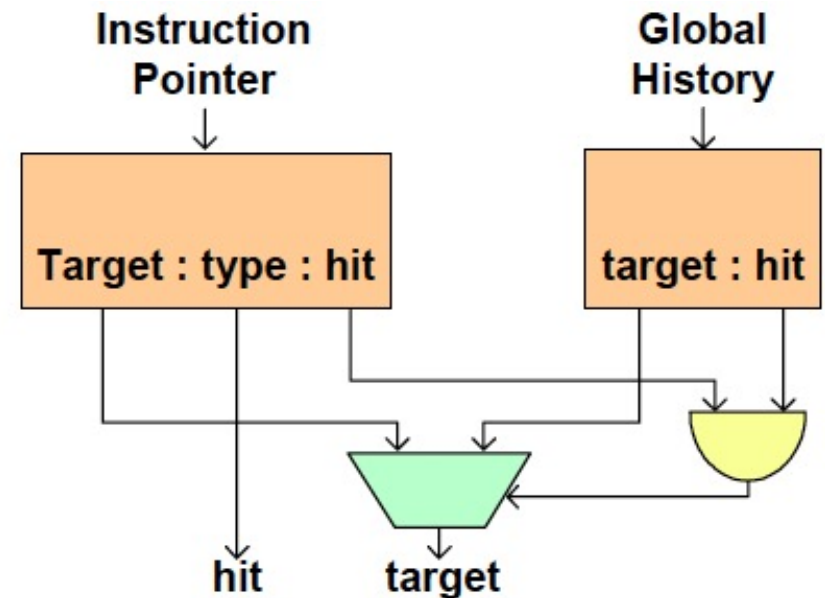


Figure 3: The Indirect Branch Predictor logic

Gochman et al.,

“**The Intel Pentium M Processor: Microarchitecture and Performance,**”

Intel Technology Journal, May 2003.

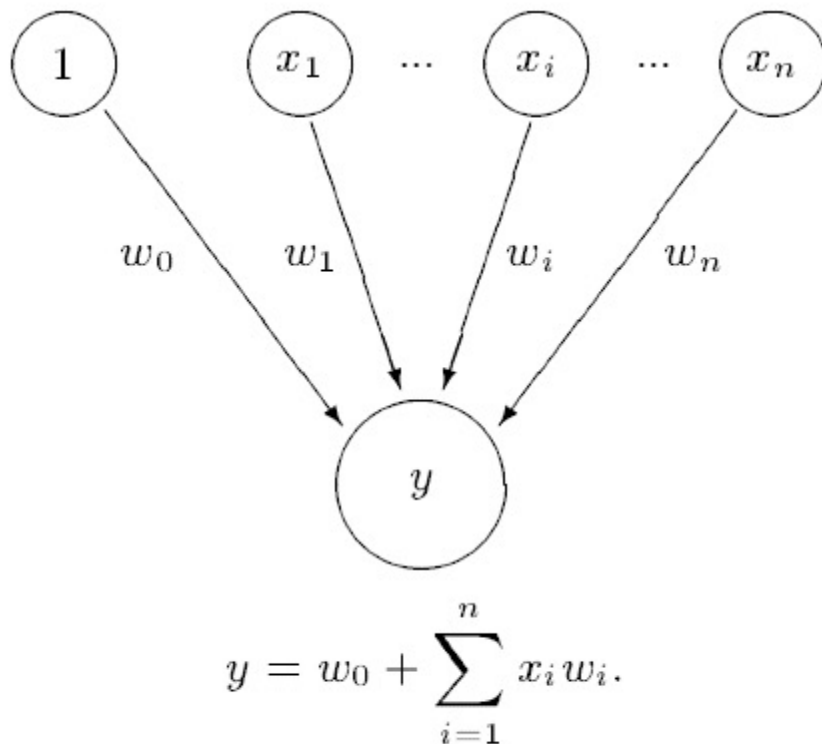
# Perceptrons for Learning Linear Functions

---

- A perceptron is a simplified model of a biological neuron
- It is also a simple **binary classifier**
- A perceptron maps an input vector  $X$  to a 0 or 1
  - Input = Vector  $X$
  - Perceptron learns the linear function (if one exists) of how each element of the vector affects the output (stored in an internal Weight vector)
  - Output =  $\text{Weight} \cdot X + \text{Bias} > 0$
- In the branch prediction context
  - Vector  $X$ : Branch history register bits
  - Output: Prediction for the current branch

# 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



Each branch associated with a perceptron

A perceptron contains a set of weights  $w_i$   
→ 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:

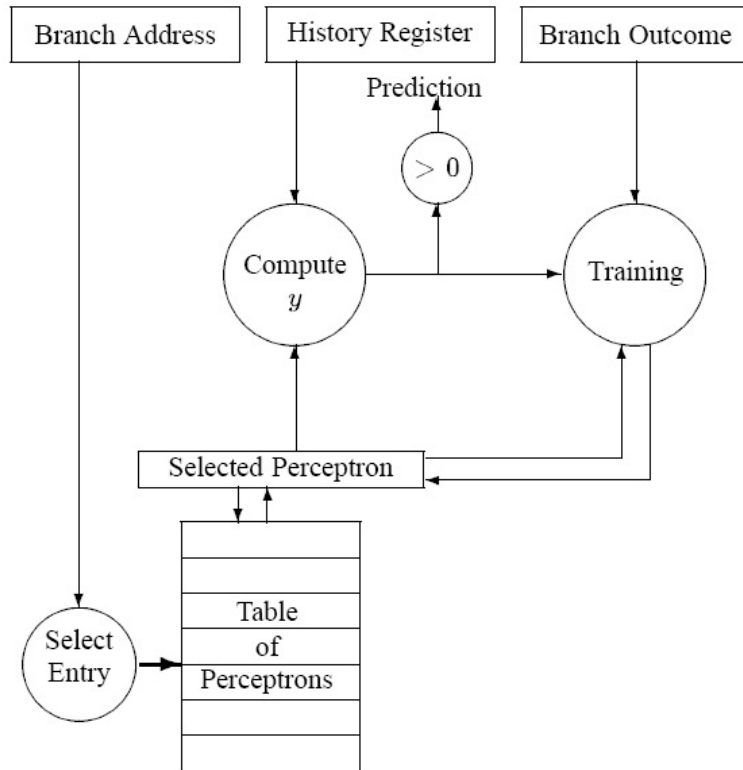
→ Express GHR bits as 1 (T) and -1 (NT)

→ 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( $y_{out}$ )  $\neq t$  or  $|y_{out}| \leq \theta$  then
  for  $i := 0$  to  $n$  do
     $w_i := w_i + tx_i$ 
  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

**A successful example of use of machine learning in processor design**

See, e.g., Grayson+, “[Evolution of the Samsung Exynos CPU Microarchitecture](#),” ISCA 2020.

# 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

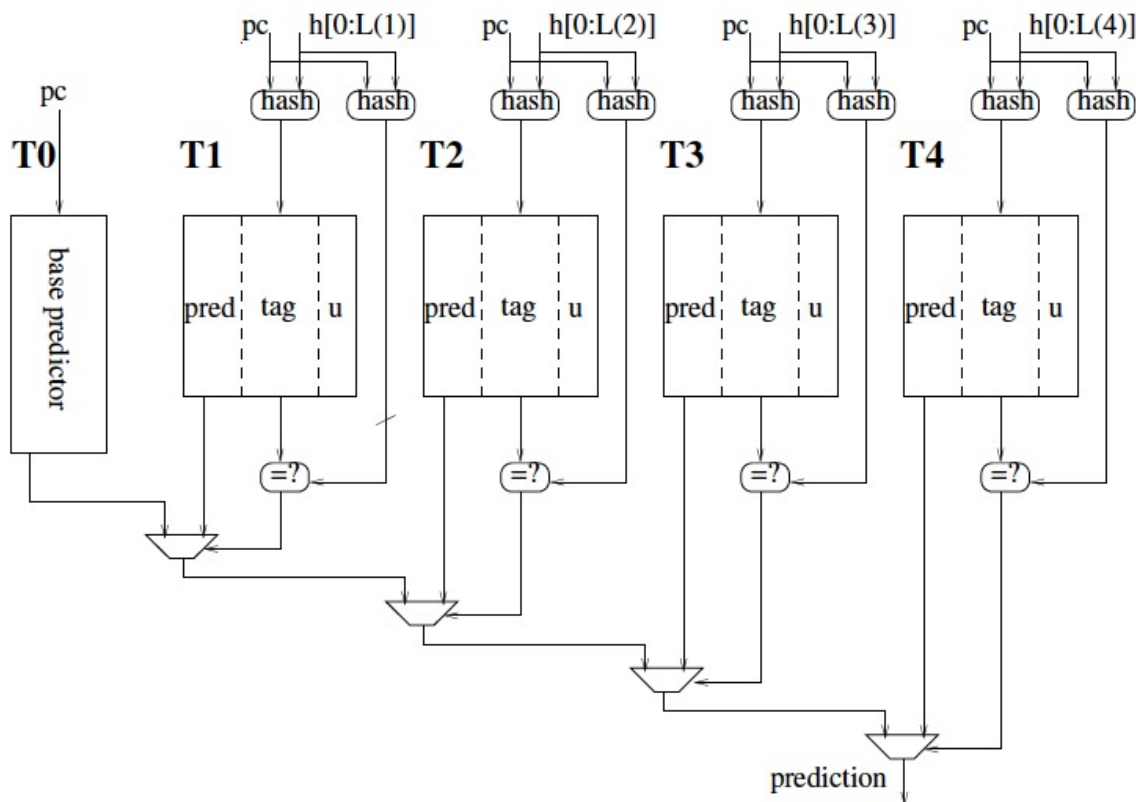
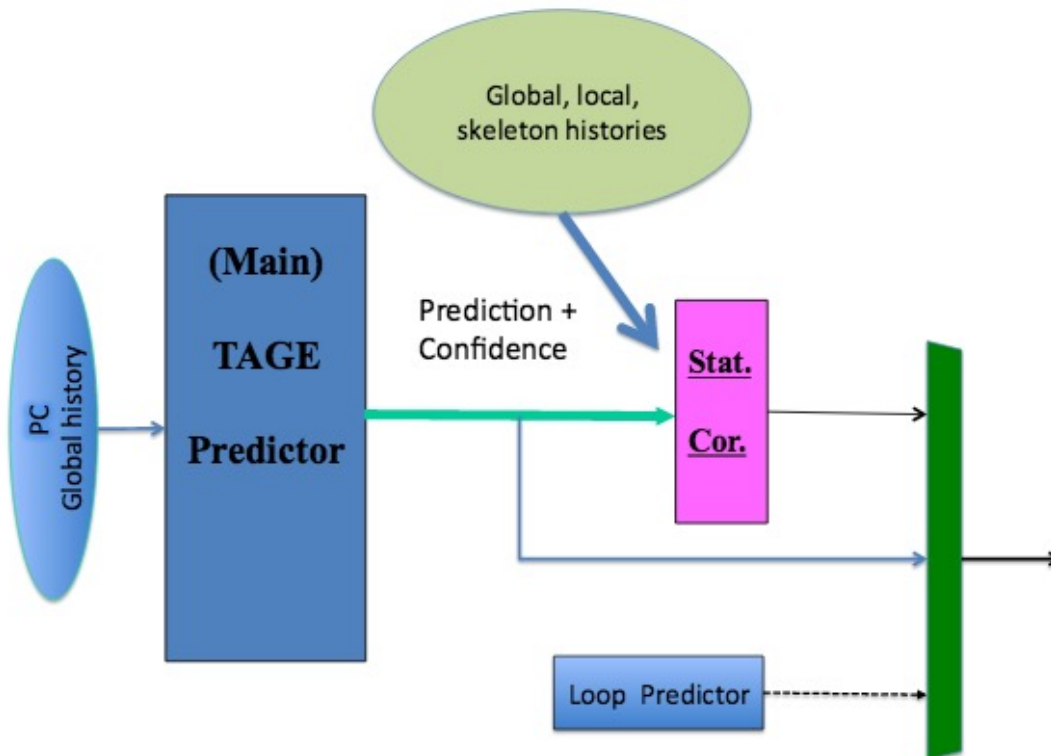


Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

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>



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

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

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



# 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.

# Other Ways of Handling Branches

# 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)

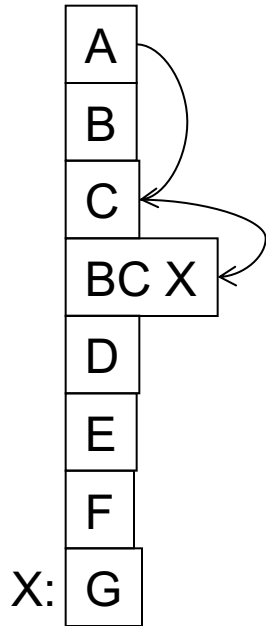
# Delayed Branching (I)

---

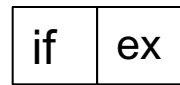
- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles
- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are **always** executed regardless of branch direction.
- Problem: How do you find instructions to fill the delay slots?
  - Branch must be independent of delay slot instructions
- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot

# Delayed Branching (II)

Normal code:



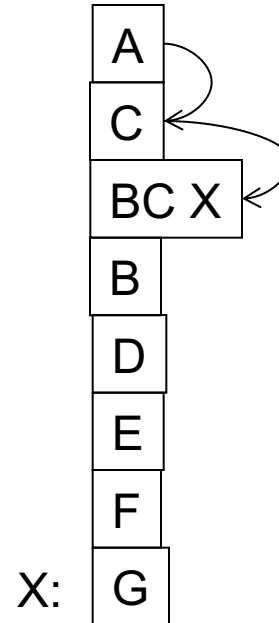
Timeline:



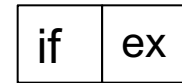
A  
B A  
C B  
BC C  
-- BC  
G --

6 cycles

Delayed branch code:



Timeline:



A  
C A  
BC C  
B BC  
G B

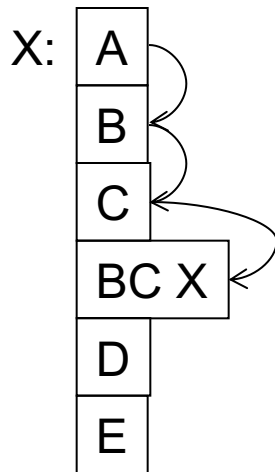
5 cycles

# Fancy Delayed Branching (III)

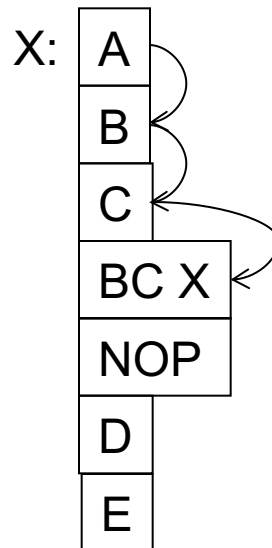
## ■ Delayed branch with squashing

- In SPARC
- Semantics: If the branch falls through (i.e., it is **not taken**), the delay slot instruction is **not** executed
- Why could this help?

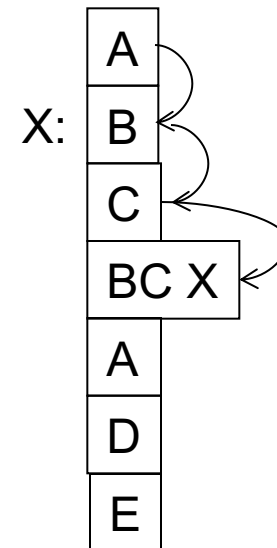
Normal code:



Delayed branch code:



Delayed branch w/ squashing:



# Delayed Branching (IV)

---

## ■ Advantages:

+ Keeps the pipeline full with useful instructions in a simple way assuming

1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves
2. All delay slots can be filled with useful instructions

## ■ Disadvantages:

-- Not easy to fill the delay slots (even with a 2-stage pipeline)

1. Number of delay slots increases with pipeline depth, superscalar execution width
2. Number of delay slots should be variable with variable latency operations. Why?

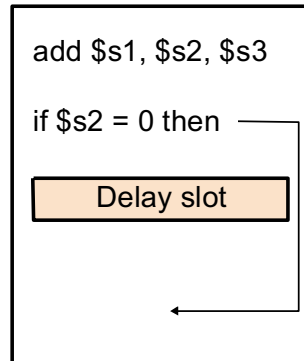
-- Ties ISA semantics to hardware implementation

- SPARC, MIPS, HP-PA: 1 delay slot
- What if pipeline implementation changes with the next design?

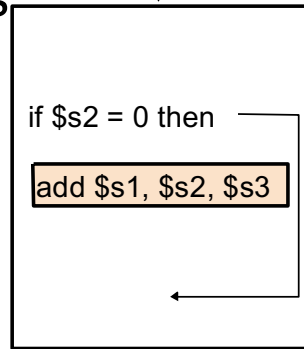
# An Aside: Filling the Delay Slot

reordering data  
independent  
(RAW, WAW,  
WAR)  
instructions  
does not change  
program semantics

a. From before

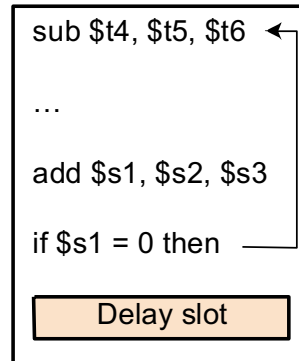


Becomes

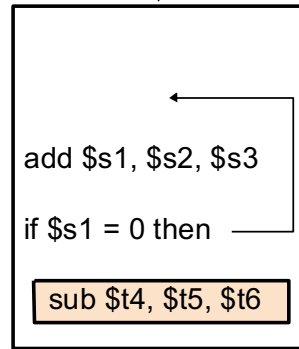


within same  
basic block

b. From target

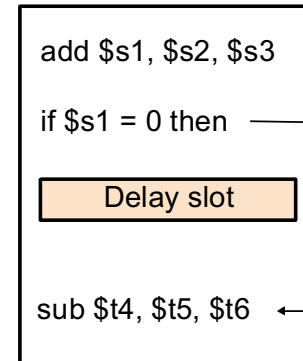


Becomes

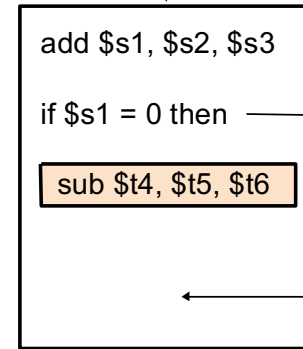


For correctness:  
add a new instruction  
to the not-taken path?

c. From fall through



Becomes



For correctness:  
add a new instruction  
to the taken path?

Safe?



If Time Permits...

# 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)

# Predicate Combining (*not* Predicated Execution)

---

- Complex predicates are converted into multiple branches
  - `if ((a == b) && (c < d) && (a > 5000)) { ... }`
    - 3 conditional branches
- Problem: This increases the number of control dependencies
- Idea: Combine predicate operations to feed a single branch instruction instead of having one branch for each
  - Predicates stored and operated on using condition registers
  - A single branch checks the value of the combined predicate
- + Fewer branches in code → fewer mipredictions/stalls
- Possibly unnecessary work
  - If the first predicate is false, no need to compute other predicates
- Condition registers exist in IBM RS6000 and the POWER architecture

# Predication (Predicated Execution)

---

- Idea: Convert control dependence to data dependence
- Simple example: Suppose we had a Conditional Move instruction...
  - CMOV condition,  $R1 \leftarrow R2$
  - $R1 = (\text{condition} == \text{true}) ? R2 : R1$
  - Employed in most modern ISAs (x86, Alpha)
- Code example with branches vs. CMOVs  
if (a == 5) {b = 4;} else {b = 3;}

CMPEQ condition, a, 5;

CMOV condition, b  $\leftarrow$  4;

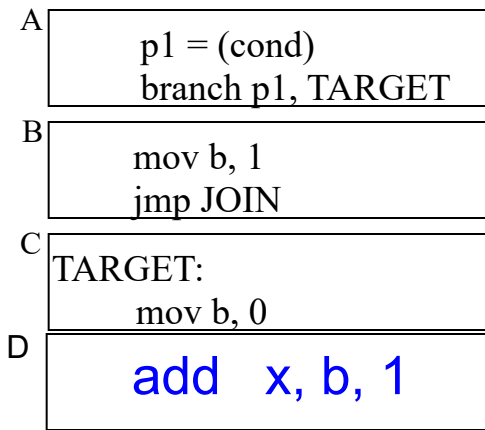
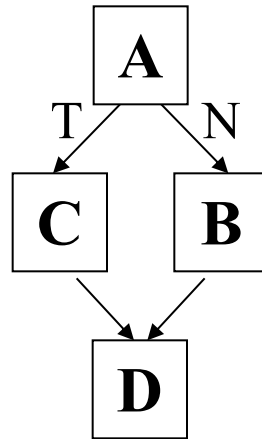
CMOV !condition, b  $\leftarrow$  3;

# Predication (Predicated Execution)

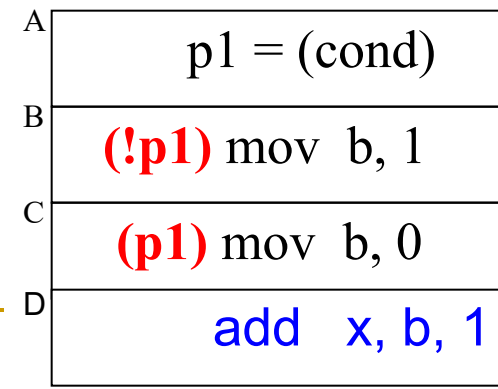
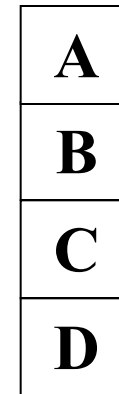
- Idea: Compiler converts control dependence into data dependence → branch is eliminated
  - Each instruction has a predicate bit set based on the predicate computation
  - Only instructions with TRUE predicates are committed (others turned into NOPs)

(normal branch code)

```
if (cond) {  
    b = 0;  
}  
else {  
    b = 1;  
}
```



(predicated code)



# Predicated Execution References

---

- Allen et al., “Conversion of control dependence to data dependence,” POPL 1983.
- Kim et al., “Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution,” MICRO 2005.

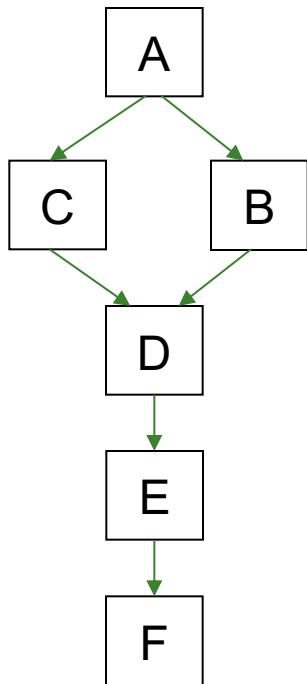
# Conditional Move Operations

---

- Very limited form of predicated execution
- CMOV R1  $\leftarrow$  R2
  - R1 = (ConditionCode == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)

# Predicated Execution (II)

- Predicated execution can be high performance and energy-efficient



## Predicated Execution

Fetch Decode Rename Schedule RegisterRead Execute



*nop*

## Branch Prediction

Fetch Decode Rename Schedule RegisterRead Execute



*Pipeline flush!!*



# Predicated Execution

---

- Eliminates branches → enables straight line code (i.e., larger basic blocks in code)
- Advantages
  - Eliminates hard-to-predict branches
  - Always-not-taken prediction works better (no branches)
  - Compiler has more freedom to optimize code (no branches)
    - control flow does not hinder inst. reordering optimizations
    - code optimizations hindered only by data dependencies
- Disadvantages
  - Useless work: some instructions fetched/executed but discarded (especially bad for easy-to-predict branches)
  - Requires additional ISA (and hardware) support
  - Can we eliminate all branches this way?

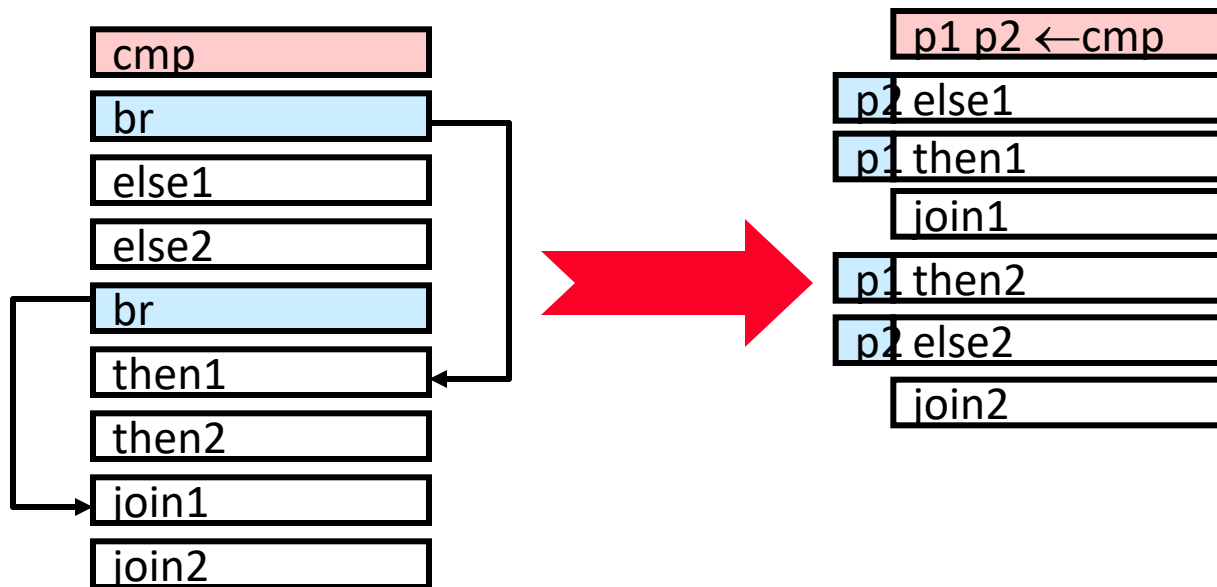
# Predicated Execution vs. Branch Prediction

---

- + Eliminates mispredictions for hard-to-predict branches
  - + No need for branch prediction for some branches
  - + Good if misprediction cost > useless work due to predication
- Causes useless work for branches that are easy to predict
  - Reduces performance if misprediction cost < useless work
  - **Adaptivity**: Static predication is not adaptive to run-time branch behavior. Branch behavior changes based on input set, program phase, control-flow path.

# Predicated Execution in Intel Itanium

- Each instruction can be separately predicated
- 64 one-bit predicate registers
  - each instruction carries a 6-bit predicate field
- An instruction is effectively a NOP if its predicate is false



# Conditional Execution in the ARM ISA

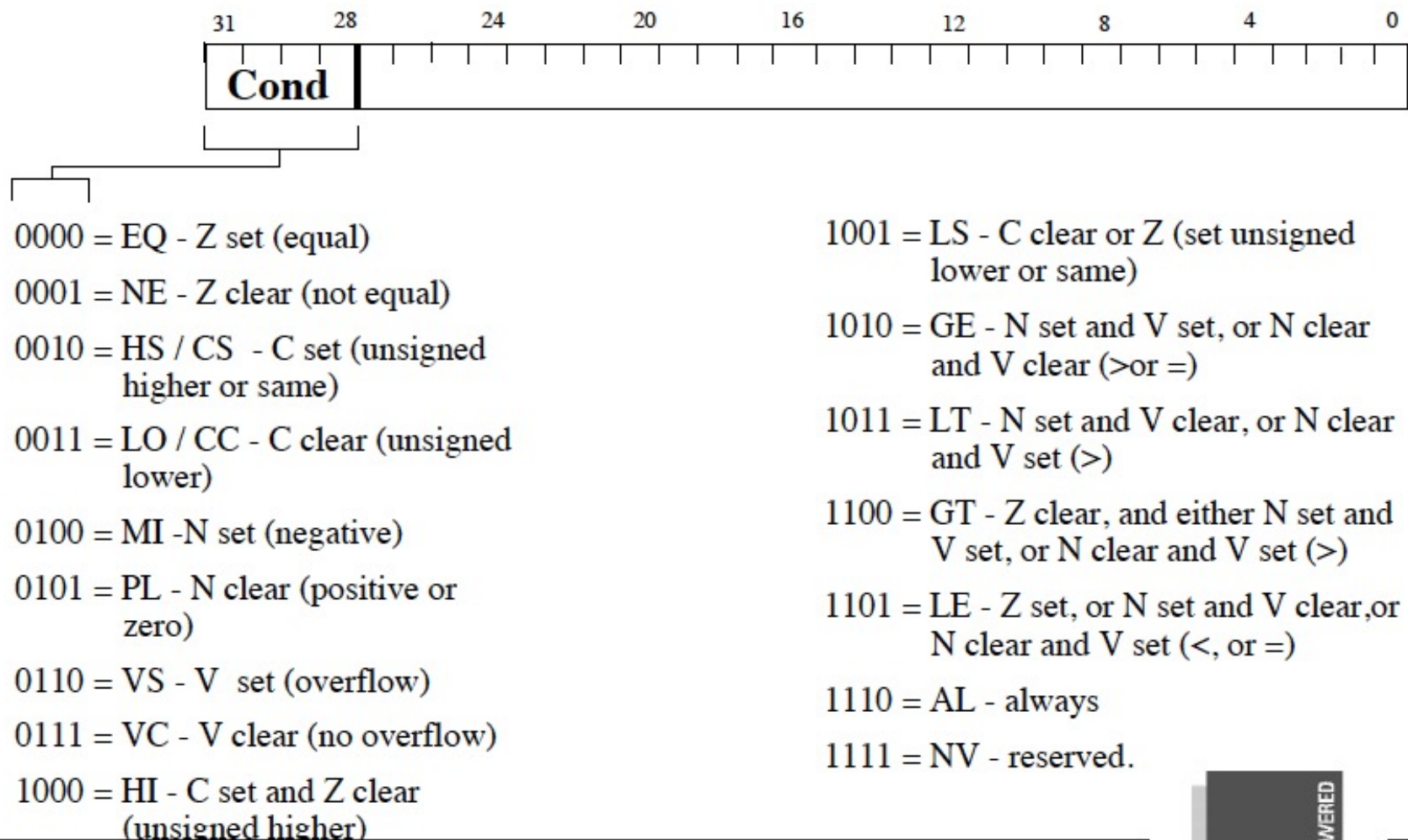
---

- Almost all ARM instructions could include an optional condition code.
  - Prior to ARM v8
- An instruction with a condition code is executed only if the condition code flags in the CPSR meet the specified condition.

# Conditional Execution in ARM ISA

31	2827				1615				87				0				<b>Instruction type</b>															
Cond	0	0	I	Opcode				S	Rn				Rd				Operand2				Data processing / PSR Transfer											
Cond	0	0	0	0	0	0	0	A	S	Rd				Rn				Rs	1	0		0	1	Rm	Multiply							
Cond	0	0	0	0	0	1	U	A	S	RdHi				RdLo				Rs	1	0		0	1	Rm		Long Multiply (v3M / v4 only)						
Cond	0	0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1		Rm	Swap				
Cond	0	1	I	P	U	B	W	L	Rn				Rd				Offset				Load/Store Byte/Word											
Cond	1	0	0	P	U	S	W	L	Rn				Register List									Load/Store Multiple										
Cond	0	0	0	P	U	1	W	L	Rn				Rd				Offset1	1	S	H	1		Offset2				Halfword transfer : Immediate offset (v4 only)					
Cond	0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm	Halfword transfer: Register offset (v4 only)						
Cond	1	0	1	L	Offset															Branch												
Cond	0	0	0	1	0				0	1	0	1				1	1	1	1				1	1	1	0	0	0	1	Rn	Branch Exchange (v4T only)	
Cond	1	1	0	P	U	N	W	L	Rn				CRd				CPNum				Offset				Coproprocessor data transfer							
Cond	1	1	1	0	Op1				CRn				CRd				CPNum				Op2	0	CRm				Coproprocessor data operation					
Cond	1	1	1	0	Op1				L	CRn				Rd				CPNum				Op2	1	CRm				Coproprocessor register transfer				
Cond	1	1	1	1	SWI Number															Software interrupt												

# Conditional Execution in ARM ISA



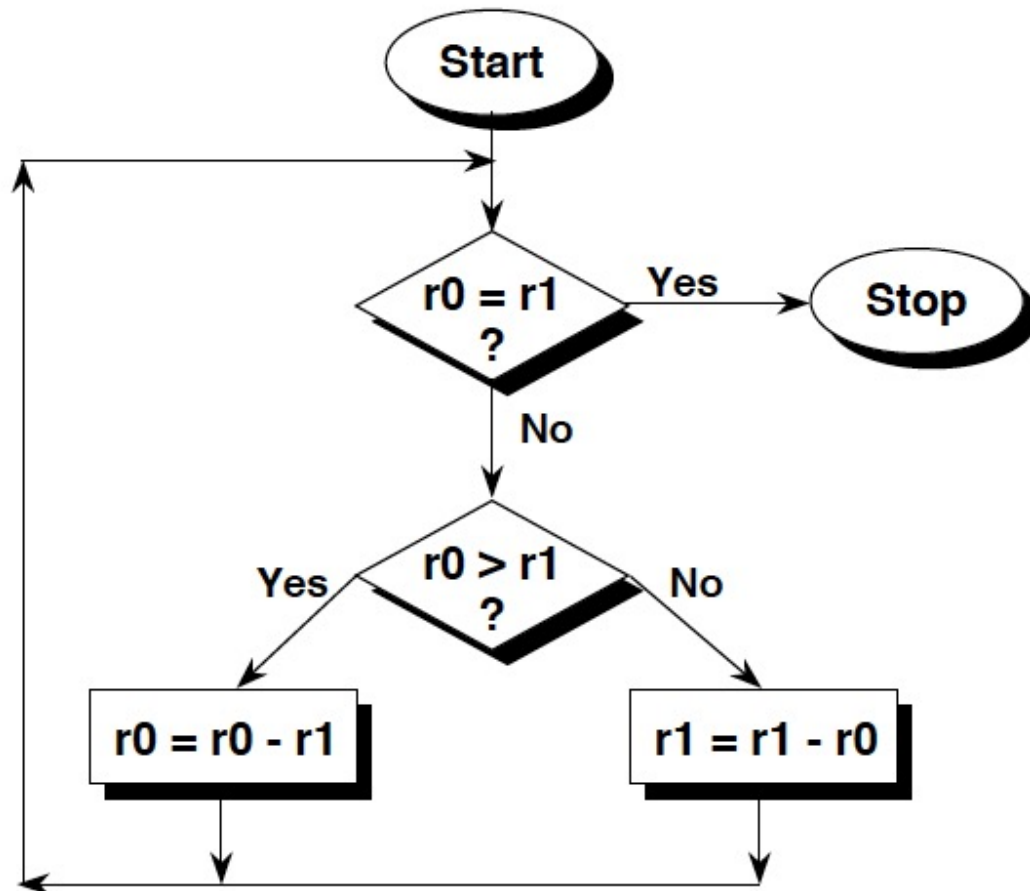


# Conditional Execution in ARM ISA

---

- \* **To execute an instruction conditionally, simply postfix it with the appropriate condition:**
  - For example an add instruction takes the form:
    - `ADD r0,r1,r2` ; `r0 = r1 + r2` (ADDAL)
  - To execute this only if the zero flag is set:
    - `ADDEQ r0,r1,r2` ; If zero flag set then...  
; ... `r0 = r1 + r2`
- \* **By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.**
  - For example to add two numbers and set the condition flags:
    - `ADDS r0,r1,r2` ; `r0 = r1 + r2`  
; ... and set flags

# Conditional Execution in ARM ISA



\* **Convert the GCD algorithm given in this flowchart into**

- 1) “Normal” assembler, where only branches can be conditional.
- 2) ARM assembler, where all instructions are conditional, thus improving code density.

\* **The only instructions you need are `CMP`, `B` and `SUB`.**



# Conditional Execution in ARM ISA

---

## “Normal” Assembler

```
gcd    cmp r0, r1      ;reached the end?
        beq stop
        blt less       ;if r0 > r1
        sub r0, r0, r1  ;subtract r1 from r0
        bal gcd
less   sub r1, r1, r0   ;subtract r0 from r1
        bal gcd
stop
```

---

## ARM Conditional Assembler

```
gcd    cmp    r0, r1      ;if r0 > r1
        subgt r0, r0, r1  ;subtract r1 from r0
        sublt r1, r1, r0  ;else subtract r0 from r1
        bne   gcd        ;reached the end?
```

---

# How to Handle Control Dependences

---

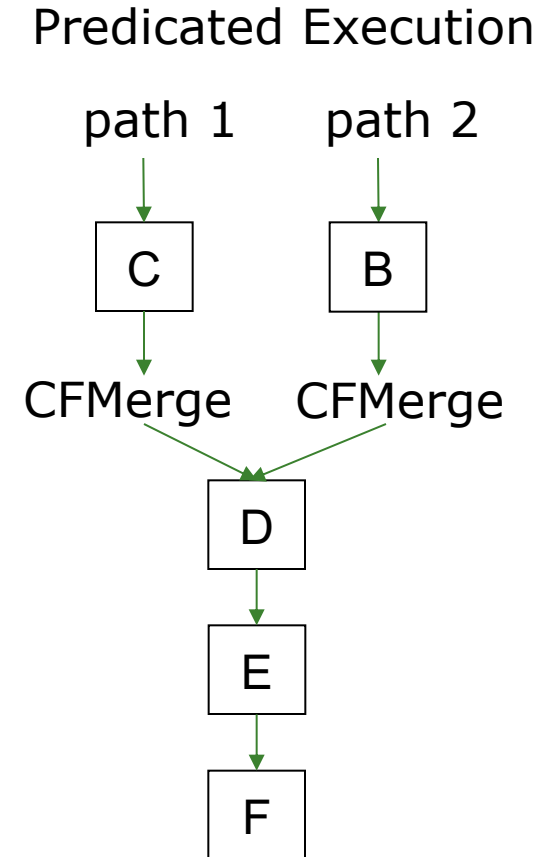
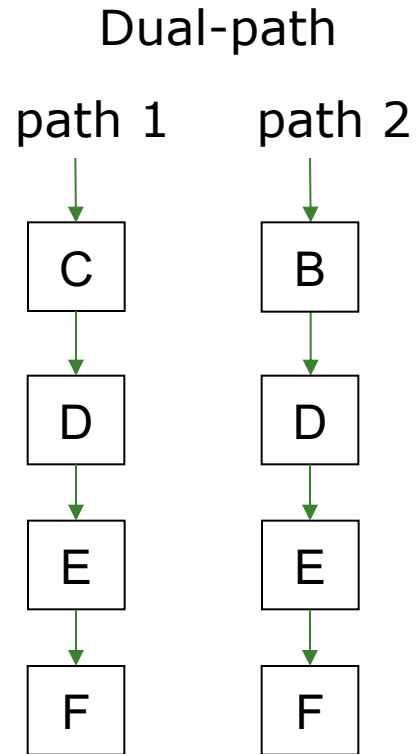
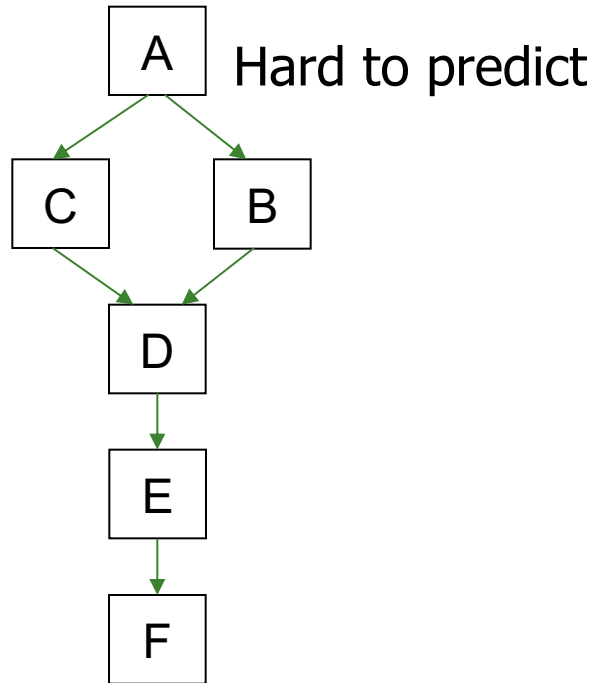
- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- Potential solutions if the instruction is a control-flow instruction:
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address (branch prediction)
  - Employ delayed branching (branch 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)

# Multi-Path Execution

---

- Idea: Execute both paths after a conditional branch
  - For all branches: Riseman and Foster, “The inhibition of potential parallelism by conditional jumps,” IEEE Transactions on Computers, 1972.
  - For a hard-to-predict branch: Use dynamic confidence estimation
  
- Advantages:
  - + Improves performance if misprediction cost > useless work
  - + No ISA change needed
  
- Disadvantages:
  - What happens when the machine encounters another hard-to-predict branch? Execute both paths again?
    - Paths followed quickly become exponential
  - Each followed path requires its own context (registers, PC, GHR)
  - Wasted work (and reduced performance) if paths merge

# Dual-Path Execution versus Predication



# Handling Other Types of Branches

# Remember: 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)

How can we predict an indirect branch with many target addresses?

# Call and Return Prediction

---

## ■ Direct calls are easy to predict

- Always taken, single target
- Call marked in BTB, target predicted by BTB

## ■ Returns are indirect branches

- A function can be called from many points in code
- A return instruction can have many target addresses
  - Next instruction after each call point for the same function
- Observation: Usually a return matches a call
- Idea: Use a stack to predict return addresses (Return Address Stack)
  - A fetched call: pushes the return (next instruction) address on the stack
  - A fetched return: pops the stack and uses the address as its predicted target
  - Accurate most of the time: 8-entry stack → > 95% accuracy

Call X

...

Call X

...

Call X

...

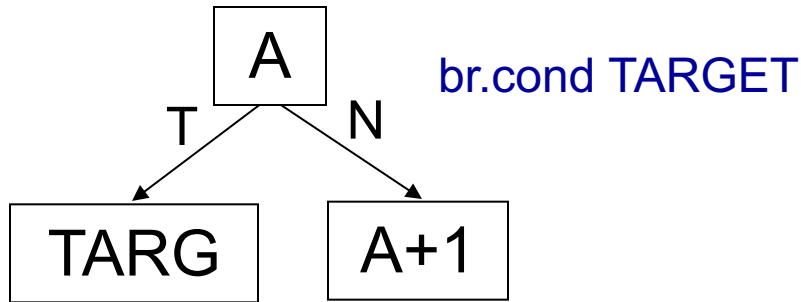
Return

Return

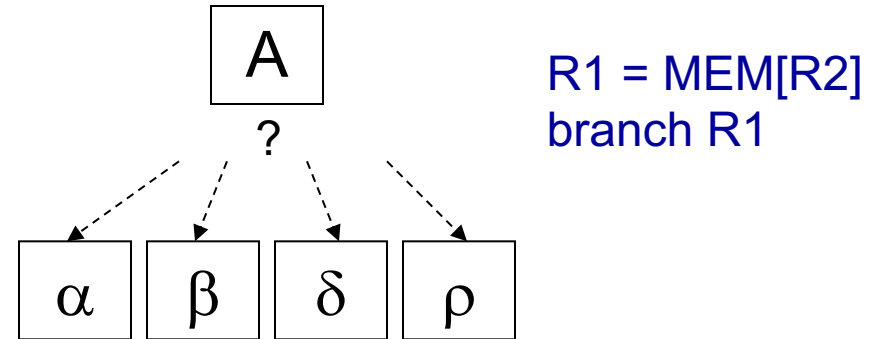
Return

# Indirect Branch Prediction (I)

- Register-indirect branches have multiple targets



Conditional (Direct) Branch



Indirect Jump

- Used to implement
  - ❑ Switch-case statements
  - ❑ Virtual function calls
  - ❑ Jump tables (of function pointers)
  - ❑ Interface calls



# Indirect Branch Prediction (II)

---

- No direction prediction needed
- Idea 1: Predict the last resolved target as the next fetch address
  - + Simple: Use the BTB to store the target address
  - Inaccurate: 50% accuracy (empirical). Many indirect branches switch between different targets
- Idea 2: Use history based target prediction
  - E.g., Index the BTB with GHR XORed with Indirect Branch PC
  - Chang et al., “Target Prediction for Indirect Jumps,” ISCA 1997.
  - + More accurate
  - An indirect branch maps to (too) many entries in BTB
    - Conflict misses with other branches (direct or indirect)
    - Inefficient use of space if branch has few target addresses

# Intel Pentium M Indirect Branch Predictor

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.

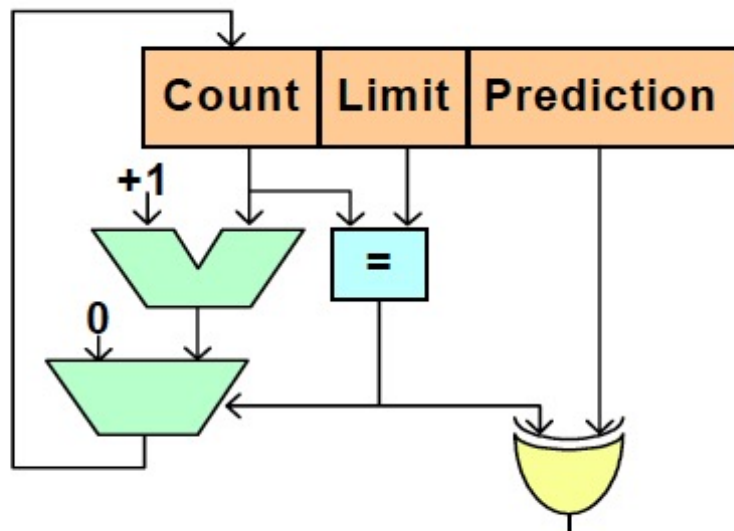


Figure 2: The Loop Detector logic

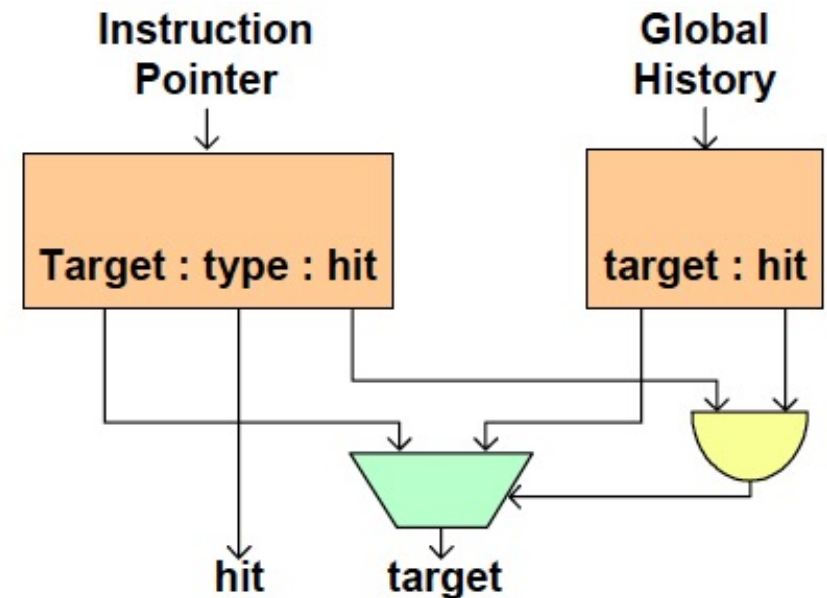


Figure 3: The Indirect Branch Predictor logic

Gochman et al.,

“**The Intel Pentium M Processor: Microarchitecture and Performance,**”

Intel Technology Journal, May 2003.

# Issues in Branch Prediction (I)

---

- Need to identify a branch before it is fetched
- How do we do this?
  - BTB hit → indicates that the fetched instruction is a branch
  - BTB entry contains the “type” of the branch
  - Pre-decoded “branch type” information stored in the instruction cache identifies type of branch
- What if no BTB?
  - Bubble in the pipeline until target address is computed
  - E.g., IBM POWER4

# Latency of Branch Prediction

---

- **Latency:** Prediction is latency critical
  - Need to generate next fetch address for the next cycle
  - Bigger, more complex predictors are more accurate but slower

