CSCI 564 Advanced Computer Architecture

Lecture 09: Handling Branches

Dr. Bo Wu (with modifications by Sumner Evans)

March 27, 2021

Colorado School of Mines

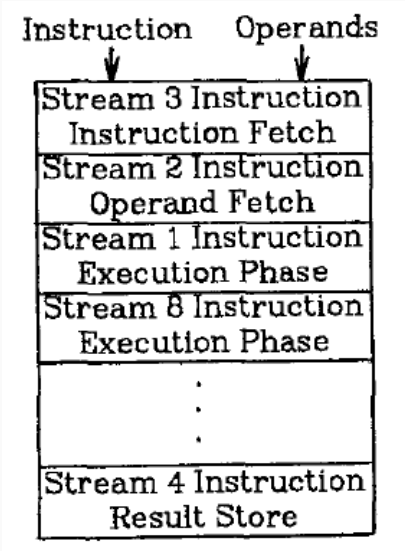
Fine-Grained Multithreading

Fine-Grained Multithreading I

Idea: hardware has multiple thread contexts. Each cycle, the fetch engine fetches from a different thread.

- By the time the fetched branch/instruction resolves, no instruction has been fetched from the same thread.
- The branch/instruction resolution latency is overlapped with execution of other threads' instructions.

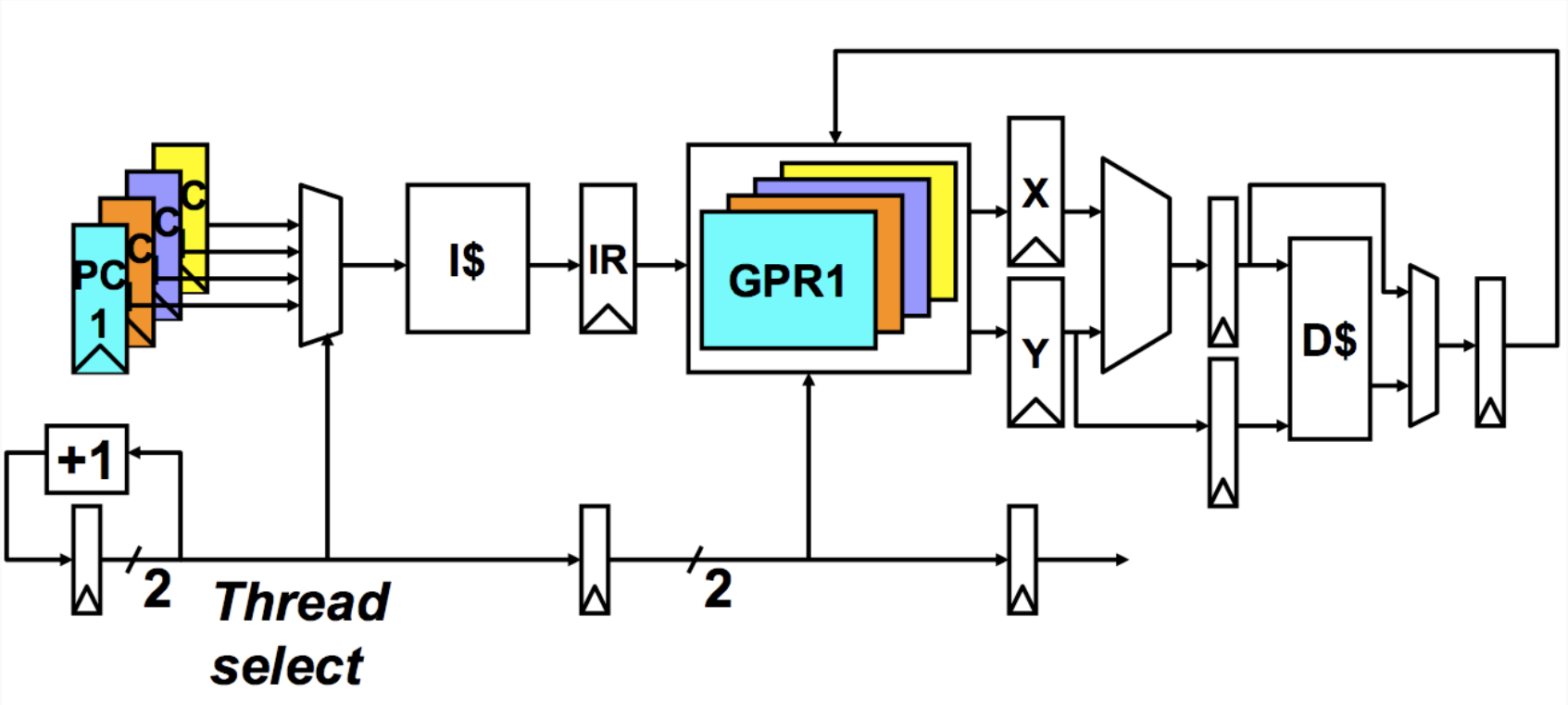
- Pro: No logic is needed for handling control and data dependencies within a thread
- Con: Single thread performance suffers
- Con: Extra logic for keeping thread contexts
- Con: Latencies do not overlap if there are not enough threads to cover the whole pipeline.



Fine-Grained Multithreading II

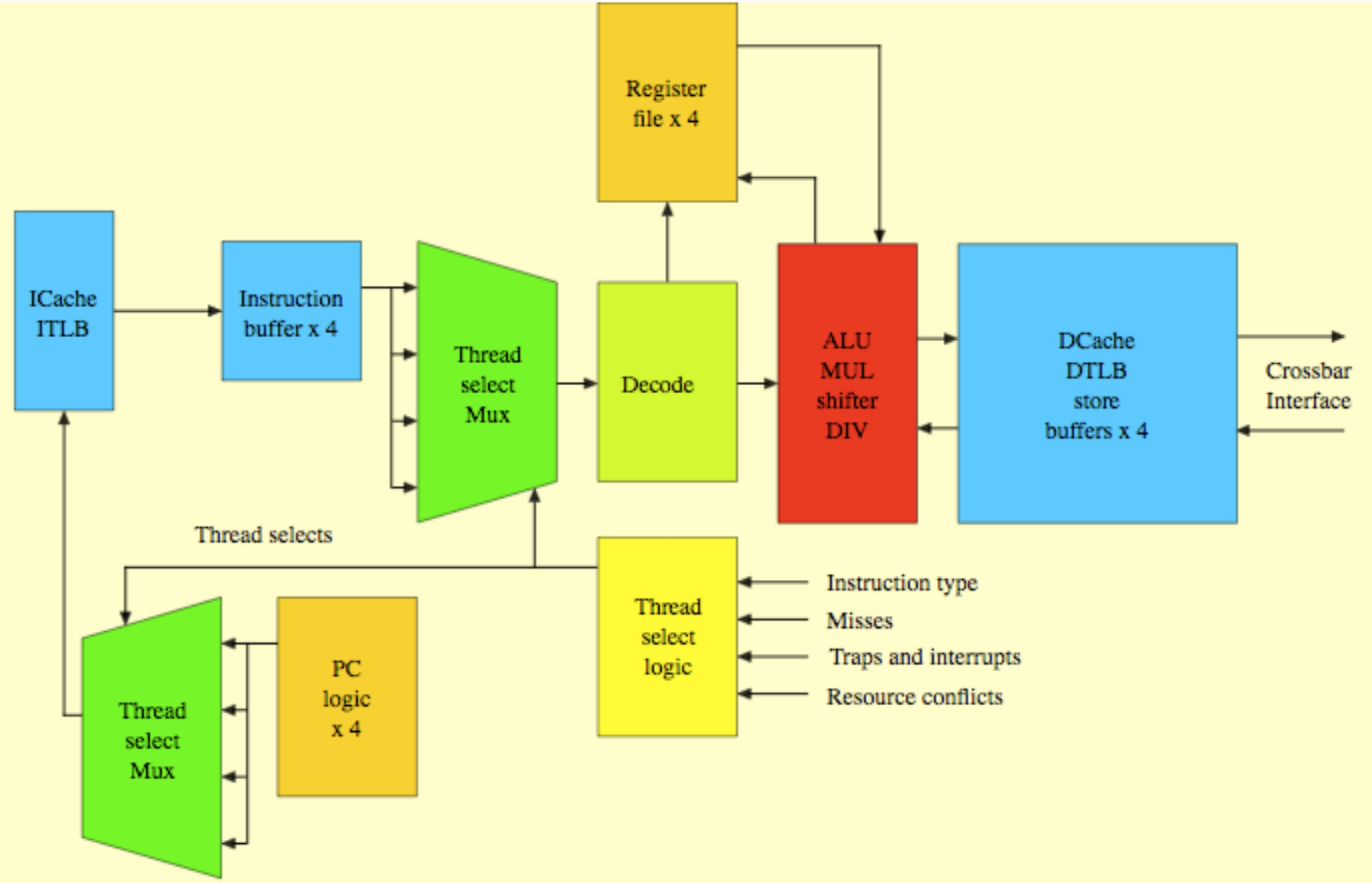
- CDC 6600's peripheral processing unit is fine-grained multithreaded
  - Thornton, *Parallel Operation in the Control Data 6600*, AFIPS 1964.
  - Processor executes a different I/O thread every cycle
  - An operation from the same thread is executed every 10 cycles
- Denelcor HEP (Heterogeneous Element Processor)
  - Smith, *A pipelined, shared resource MIMD computer*, ICPP 1978.
  - 120 threads/processor
  - available queue vs. unavailable (waiting) queue for threads
  - each thread can have only 1 instruction in the processor pipeline; each thread independent
  - **to each thread, processor looks like a non-pipelined machine**
  - **system throughput vs. single thread performance tradeoff**

Multithreading Pipeline Example



Credit: Joel Emer (MIT)

Sun Niagara Multithreaded Pipeline



Kongetira et al., *Niagara: A 32-Way Multithreaded Sparc Processor*, IEEE Micro 2005.

Fine-Grained Multithreading: Pros and Cons

Pros:

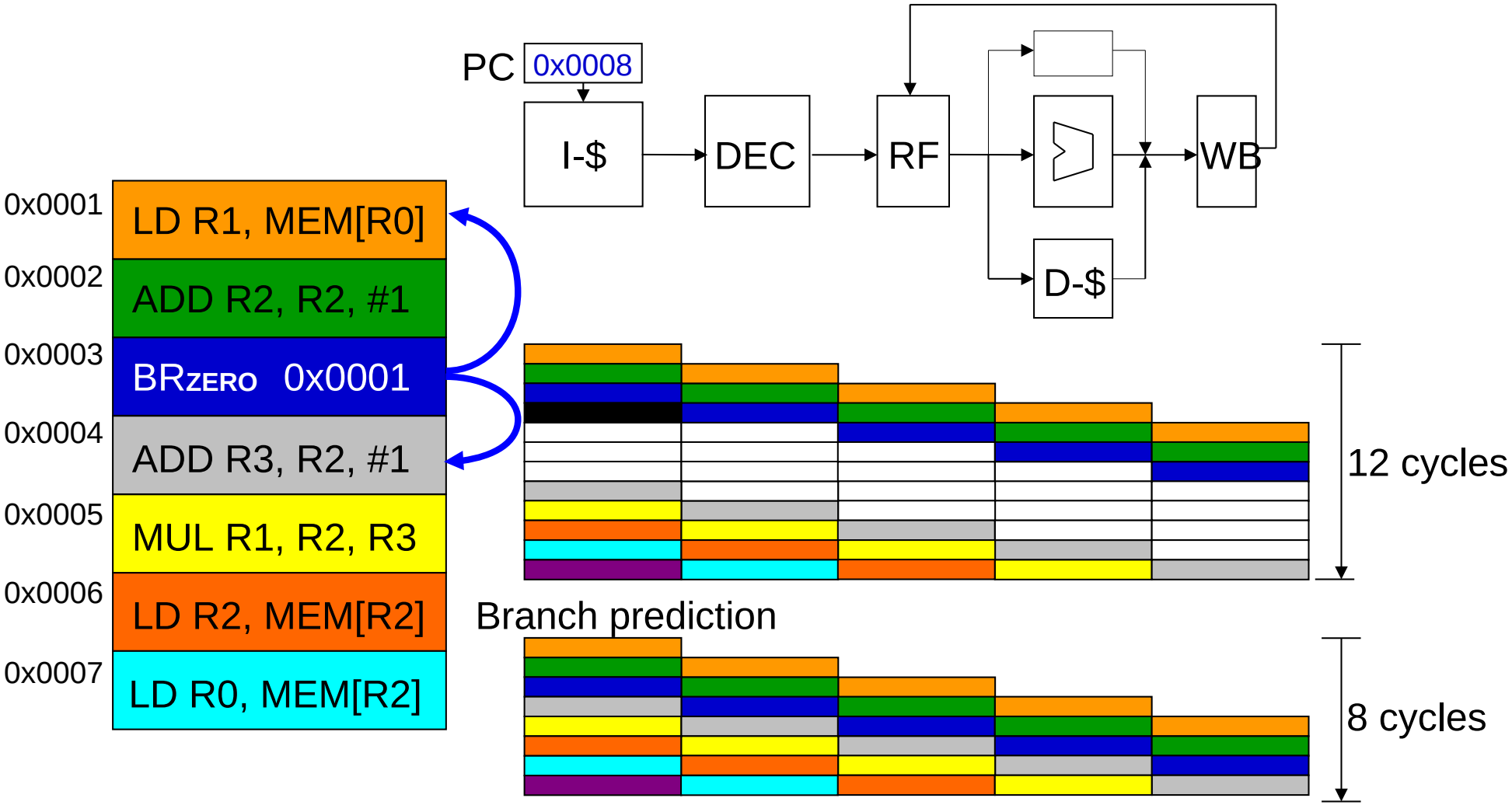
- No need for dependency checking between instructions (only one instruction in pipeline from a single thread).
- No need for *branch prediction* logic.
- Cycles that would have to be bubbles/stalls can be used for executing useful instructions from different threads.
- Improved system throughput, latency tolerance, and utilization.

Cons:

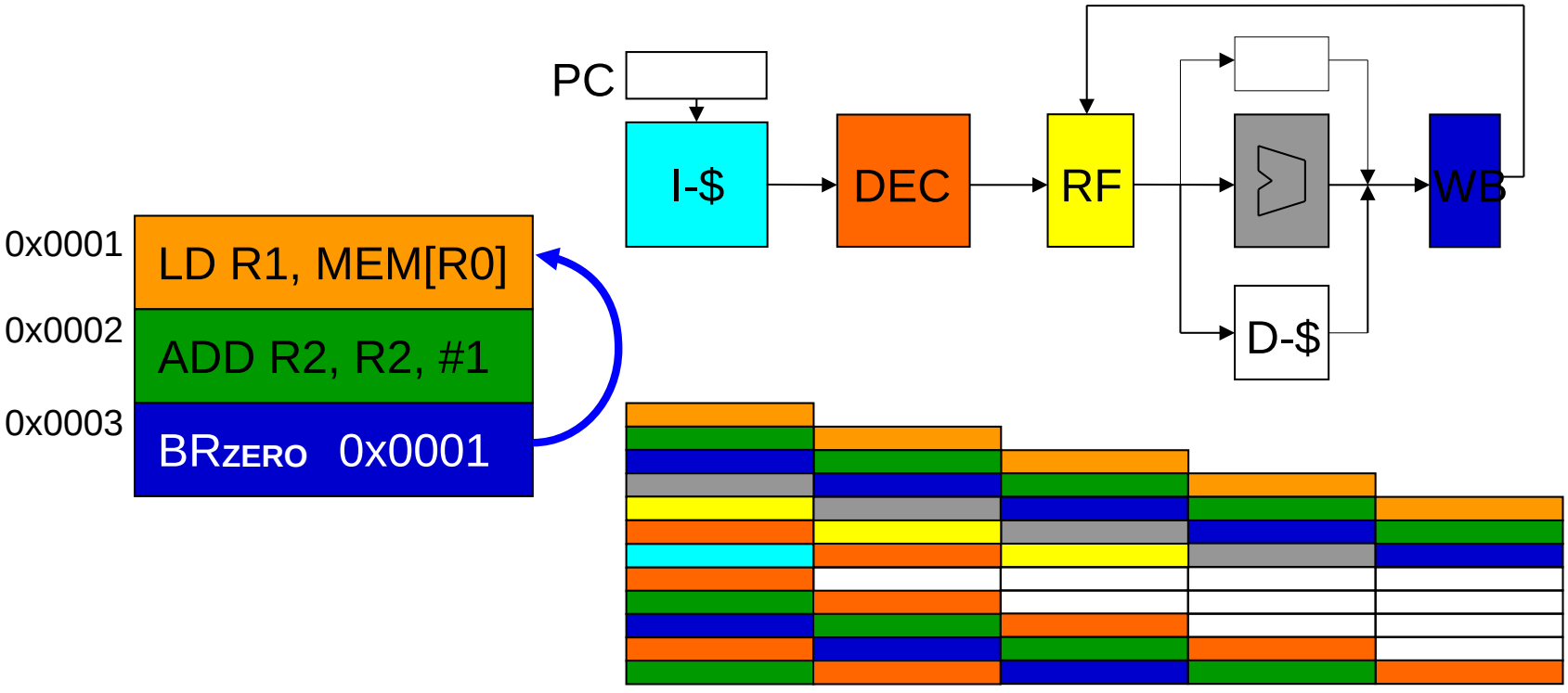
- Extra hardware complexity: multiple hardware contexts (PCs, register files, ...), thread selection logic.
- Reduced single thread performance (one instruction fetched every N cycles from the same thread).
- Resource contention between threads in caches and memory.
- Some dependency checking logic *between* threads remains (load/store)

Branch Prediction

Branch Prediction: Guess the Next Instruction to Fetch

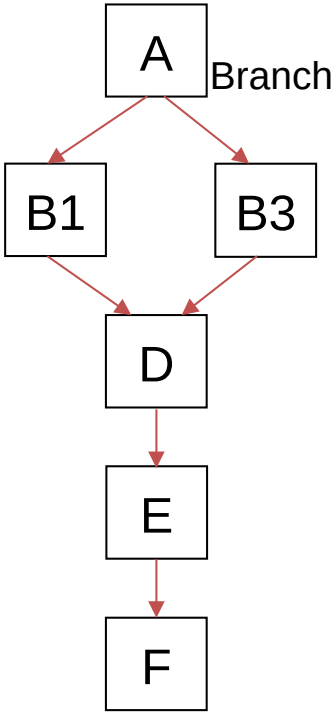


Misprediction Penalty

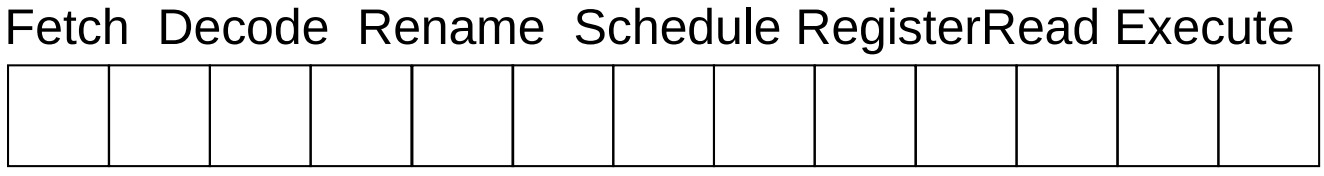


Branch Prediction

- Processors are pipelined to increase concurrency
- How do we keep the pipeline full in the presence of branches?
  - Guess the next instruction when a branch is fetched
  - Requires guessing the direction and target of a branch

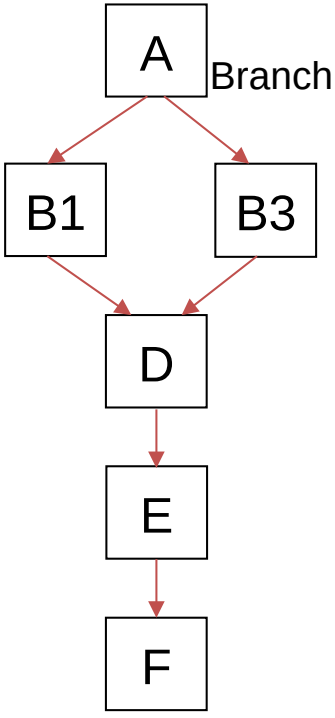


Pipeline

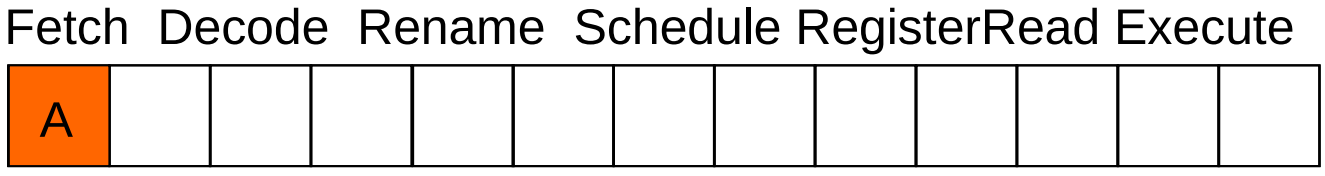


Branch Prediction

- Processors are pipelined to increase concurrency
- How do we keep the pipeline full in the presence of branches?
  - Guess the next instruction when a branch is fetched
  - Requires guessing the direction and target of a branch



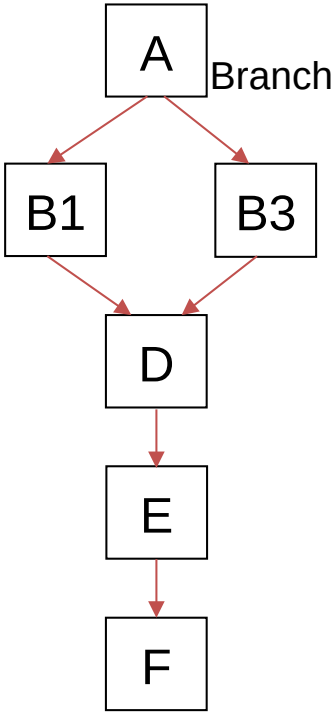
Pipeline



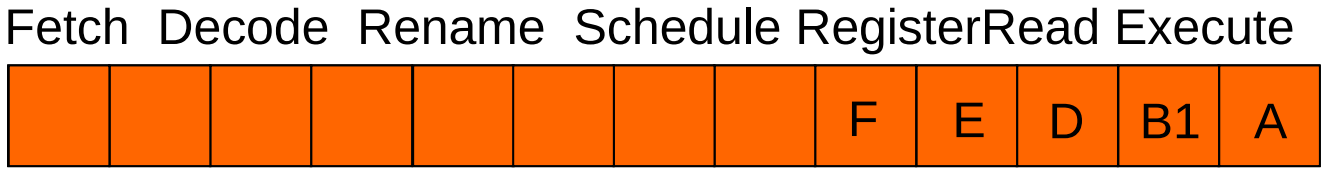
What to fetch next?

Branch Prediction

- Processors are pipelined to increase concurrency
- How do we keep the pipeline full in the presence of branches?
  - Guess the next instruction when a branch is fetched
  - Requires guessing the direction and target of a branch



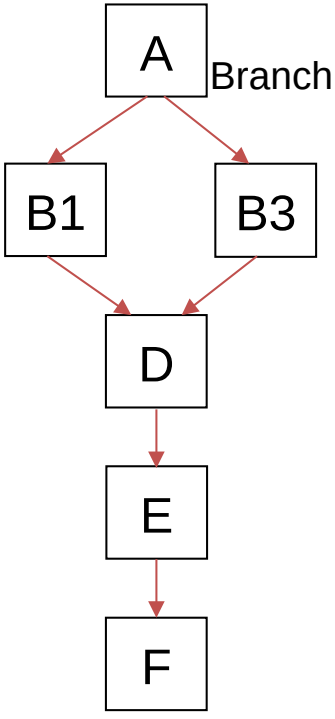
Pipeline



Verify the Prediction

Branch Prediction

- Processors are pipelined to increase concurrency
- How do we keep the pipeline full in the presence of branches?
  - Guess the next instruction when a branch is fetched
  - Requires guessing the direction and target of a branch



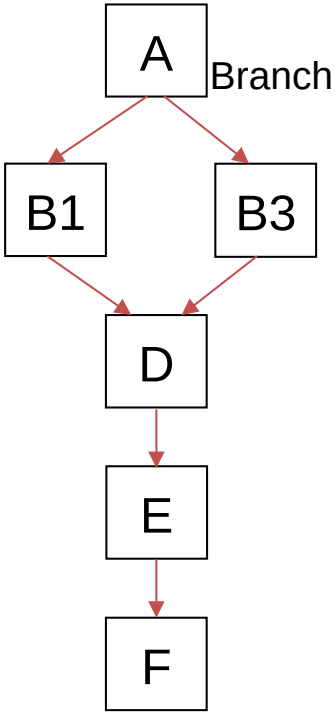
Pipeline

Fetch Decode Rename Schedule RegisterRead Execute

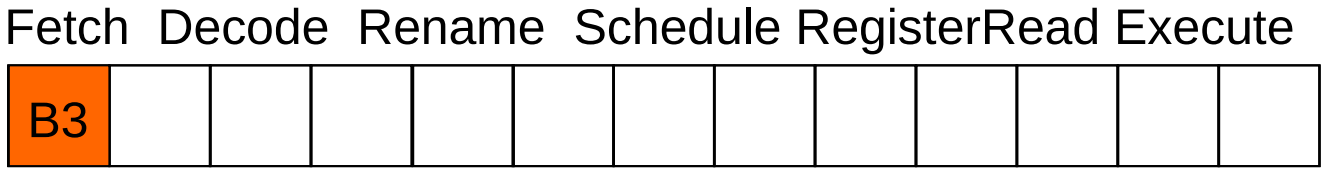
Target Misprediction Detected! Flush the pipeline

Branch Prediction

- Processors are pipelined to increase concurrency
- How do we keep the pipeline full in the presence of branches?
  - Guess the next instruction when a branch is fetched
  - Requires guessing the direction and target of a branch

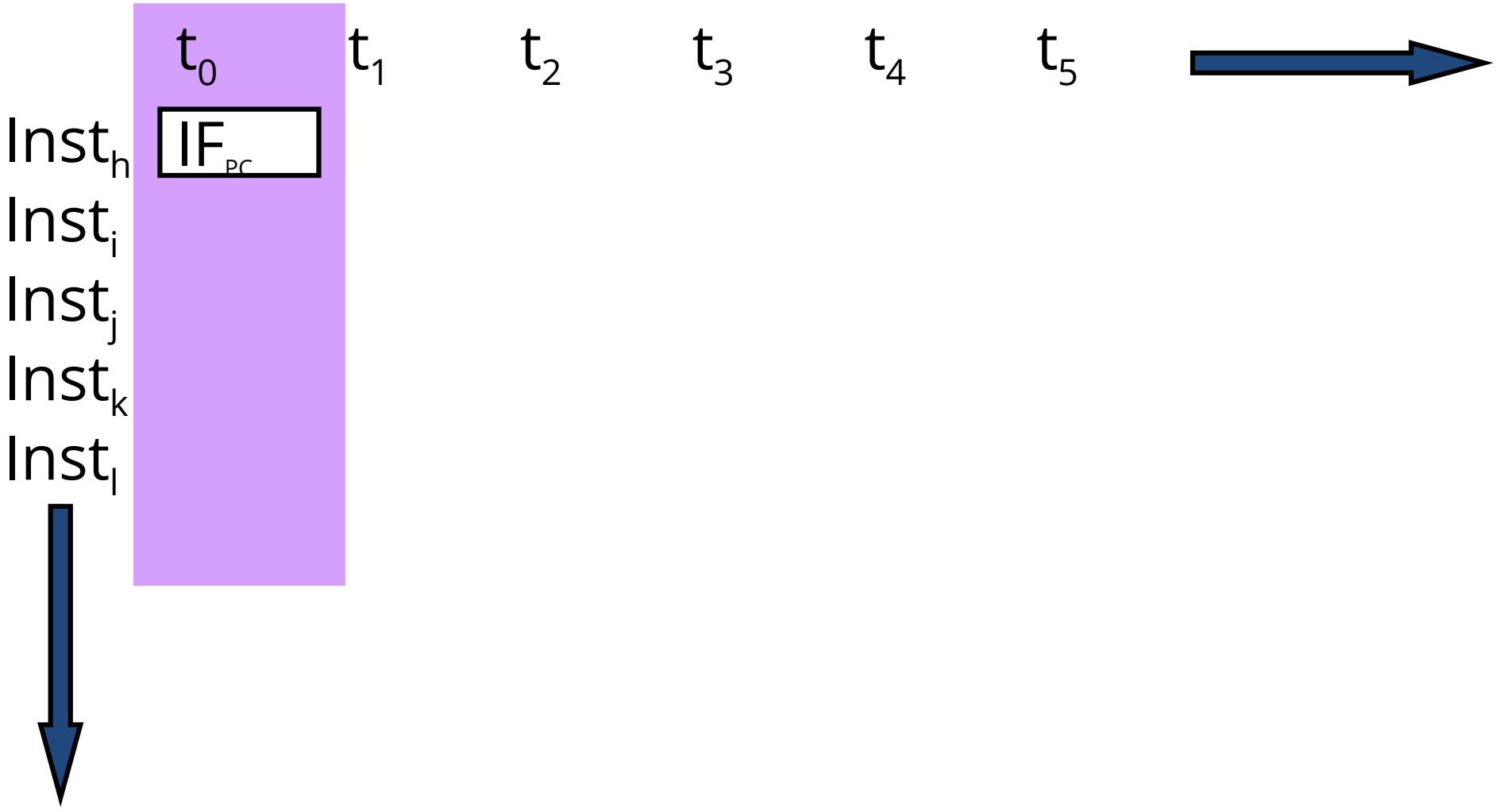


Pipeline



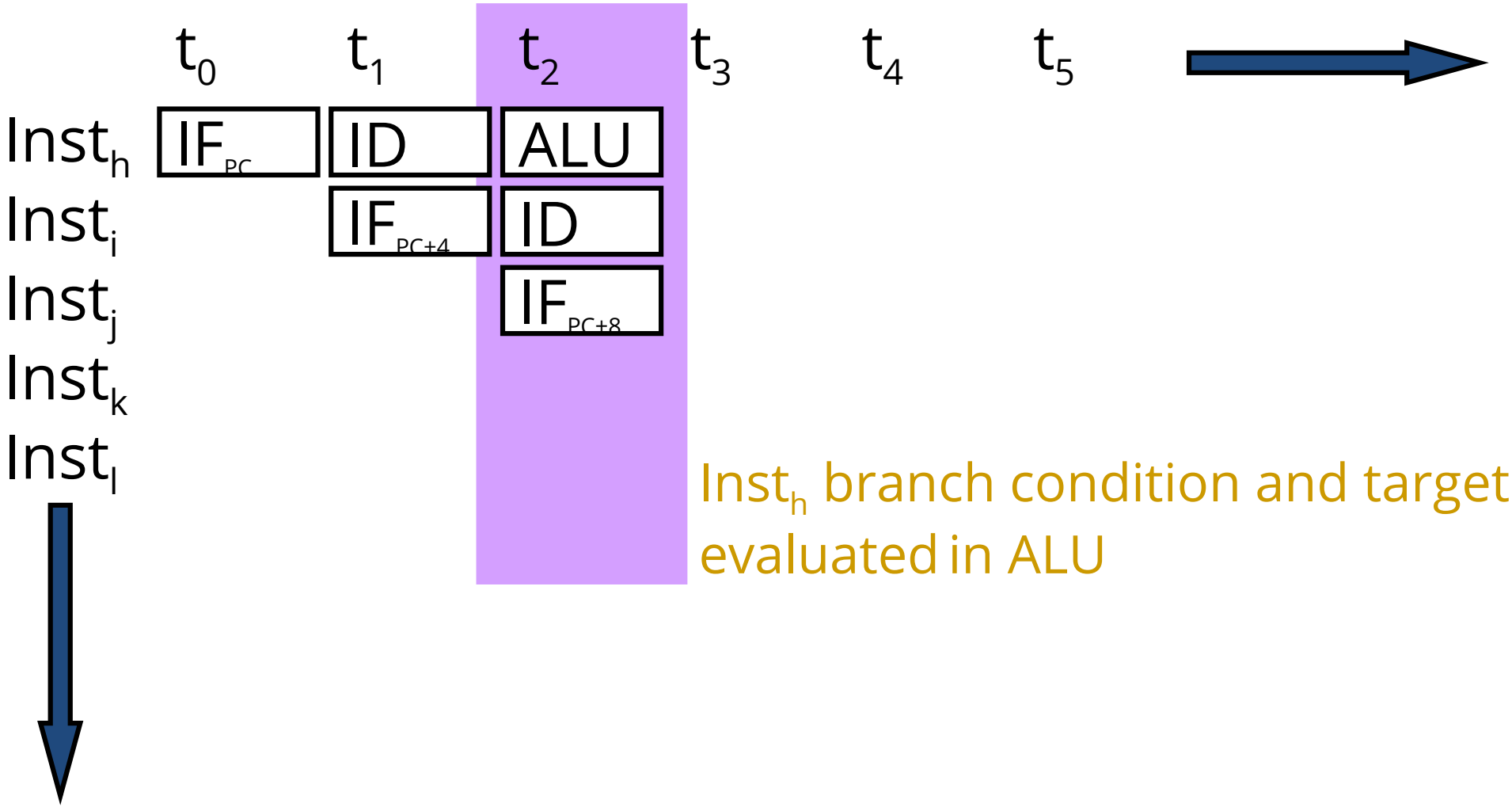
Fetch from the correct target

Branch Prediction: Always PC+4

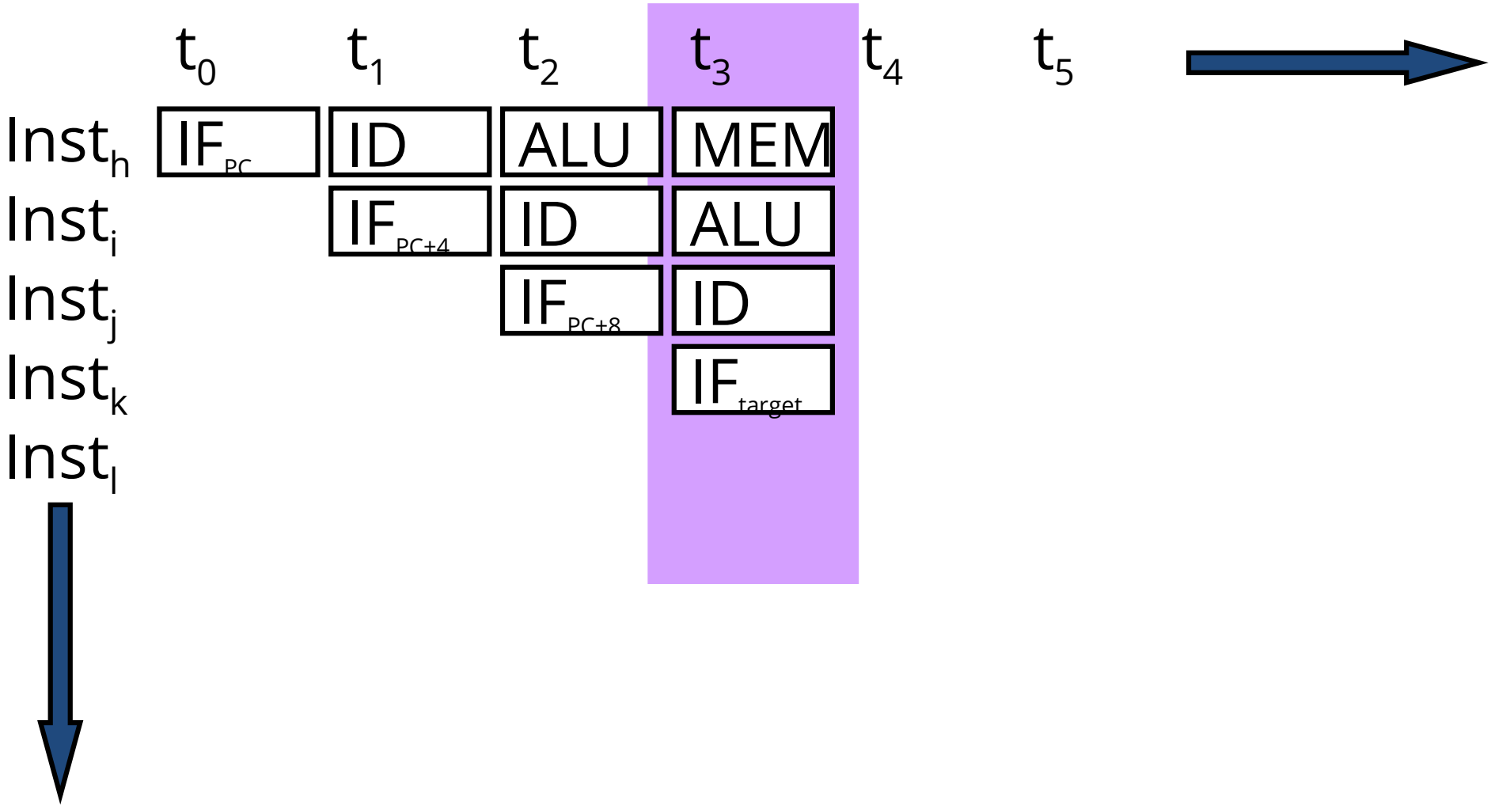


Insth is a branch

Branch Prediction: Always PC+4

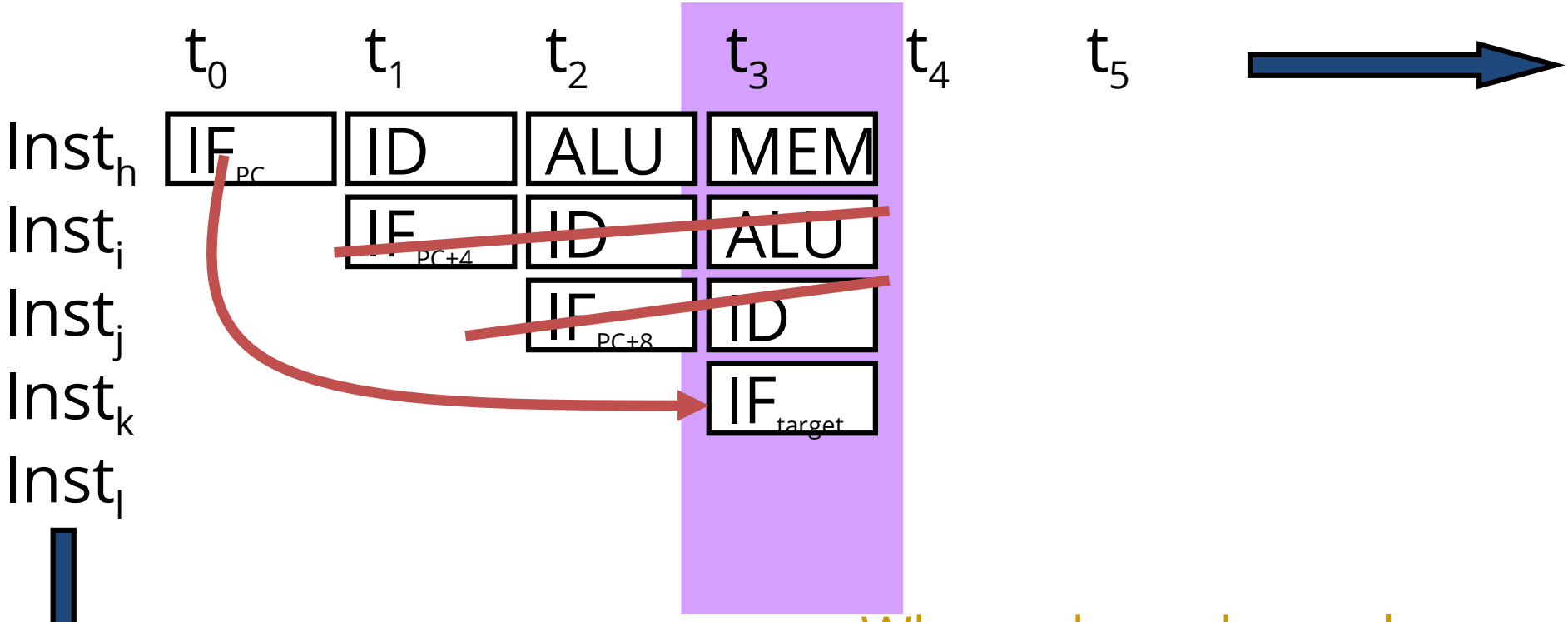


Branch Prediction: Always PC+4



$Inst_h$ is a branch

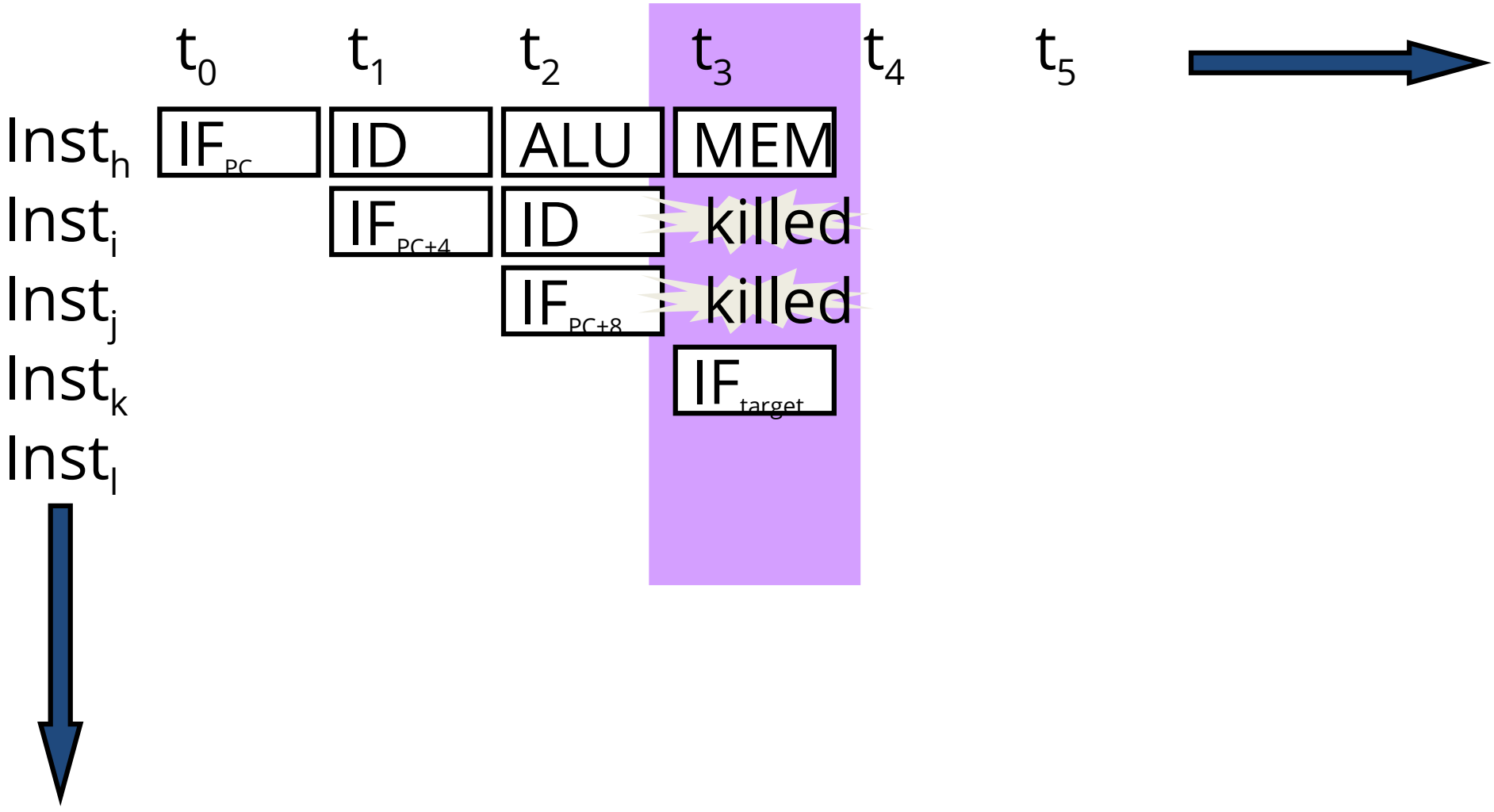
Branch Prediction: Always PC+4



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

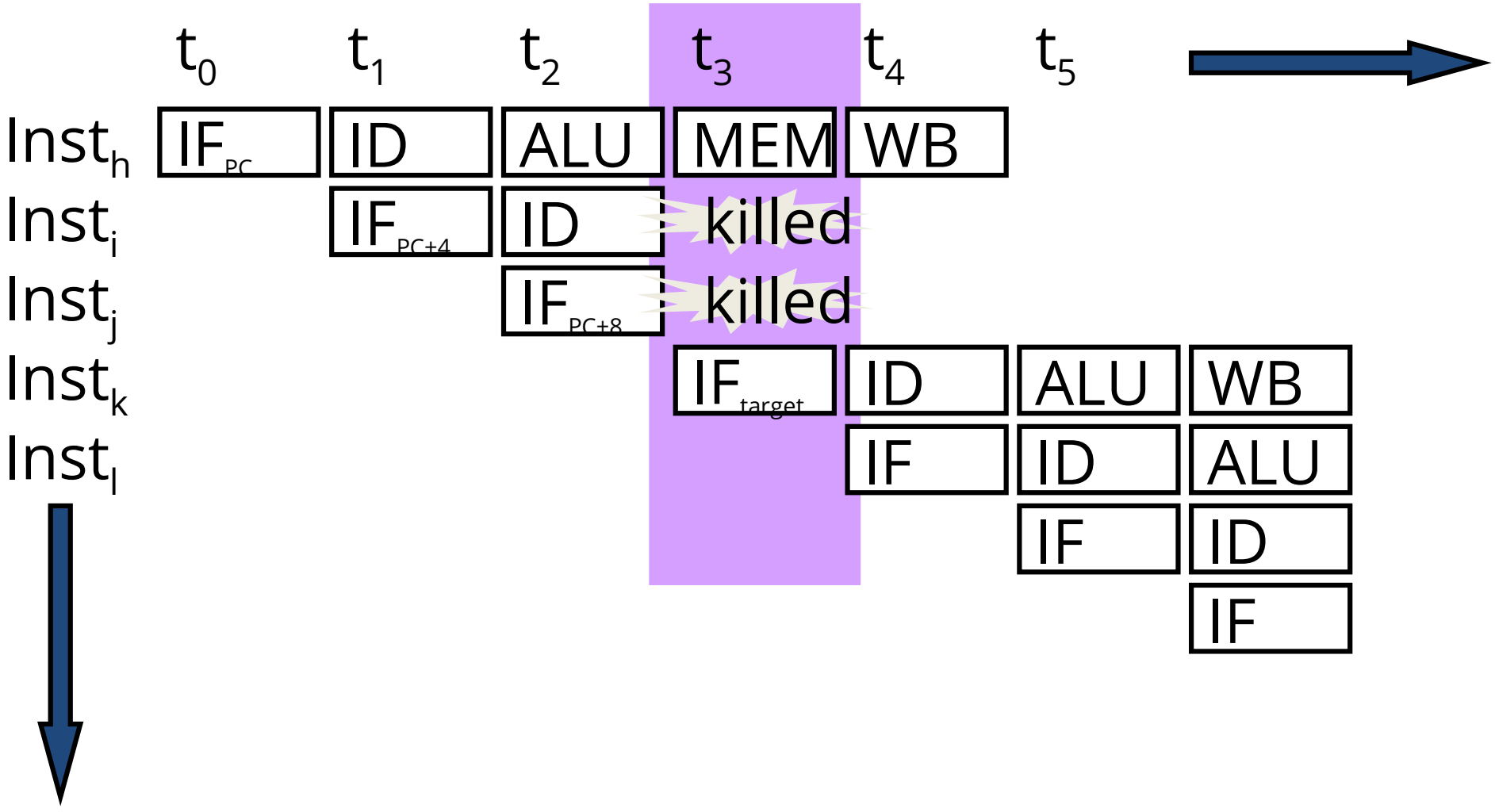
$Inst_h$ is a branch

Pipeline Flush on a Misprediction



Insth is a branch

Pipeline Flush on a Misprediction



$Inst_h$ is a branch

Performance Analysis

- Correct guess $\rightarrow$ no penalty
- Incorrect guess $\rightarrow$ two bubbles
- So, if we assume
  - no data-dependency-related stalls
  - 20% control flow instructions
  - 70% of control flow instructions are *taken*

  we can calculate the CPI:

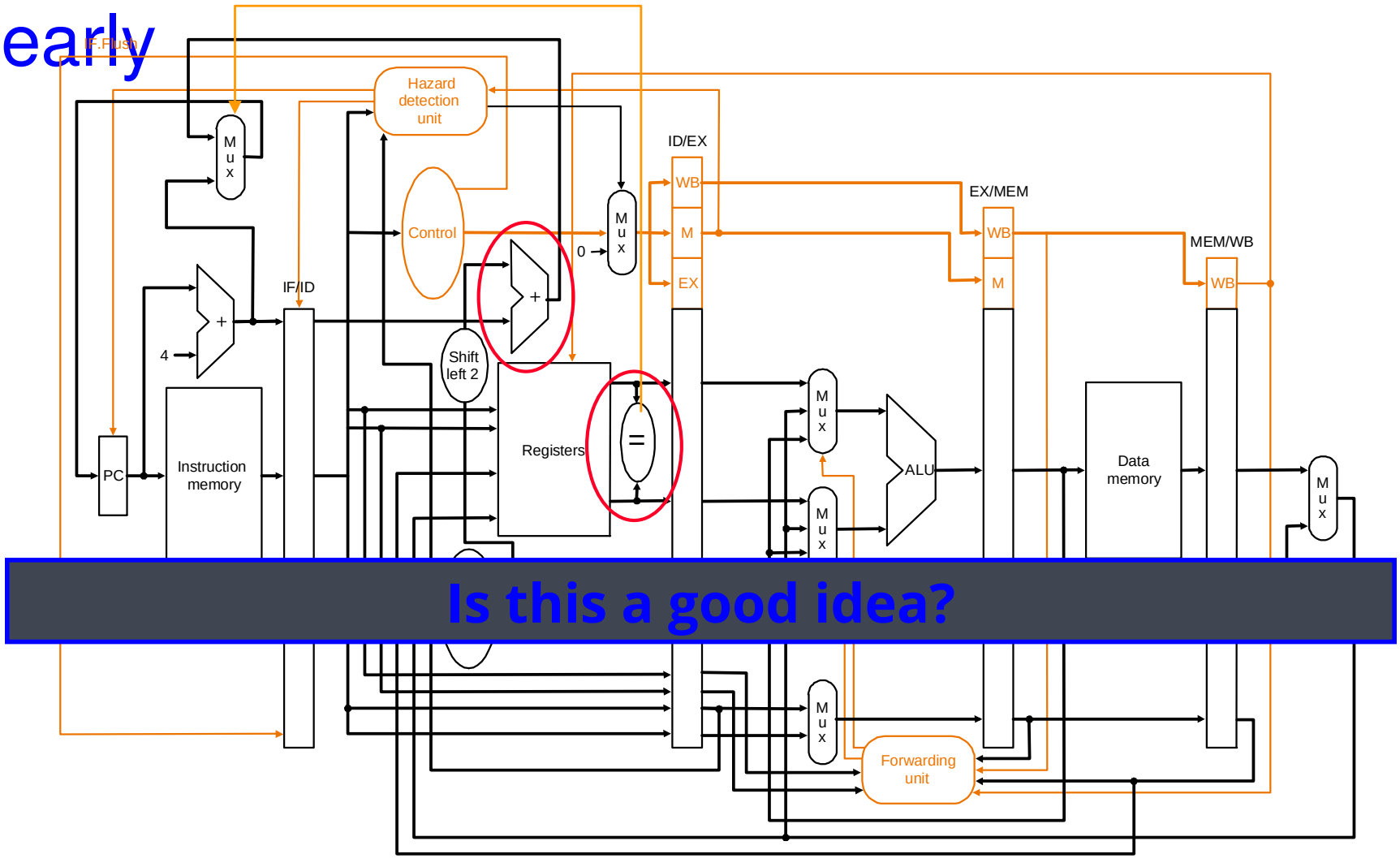
$$CPI = 1 + \underbrace{(0.2 \times 0.7)}_{\text{probability of wrong guess}} \times \underbrace{2}_{\text{penalty for wrong guess}}$$

$$= 1 + 0.14 \times 2 = \boxed{1.28}.$$

Can we reduce either of these 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 (Enhanced)

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

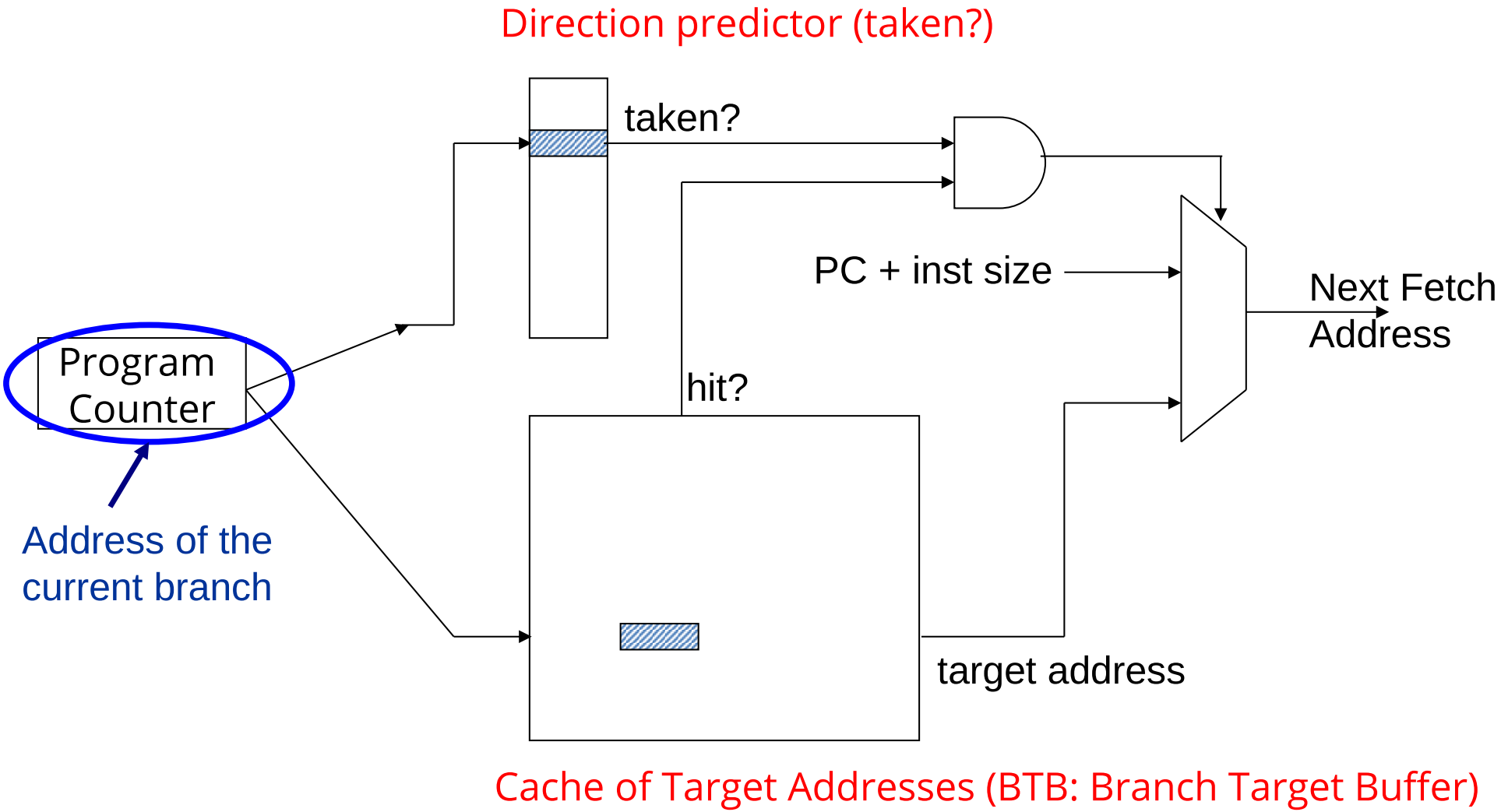
This requires three things to be predicted at the fetch stage:

- Whether the fetched instruction is a branch
- (Conditional) branch *direction*
- Branch *target address* (if taken)

Observation: the target address remains the same for a conditional direct branch across instances.

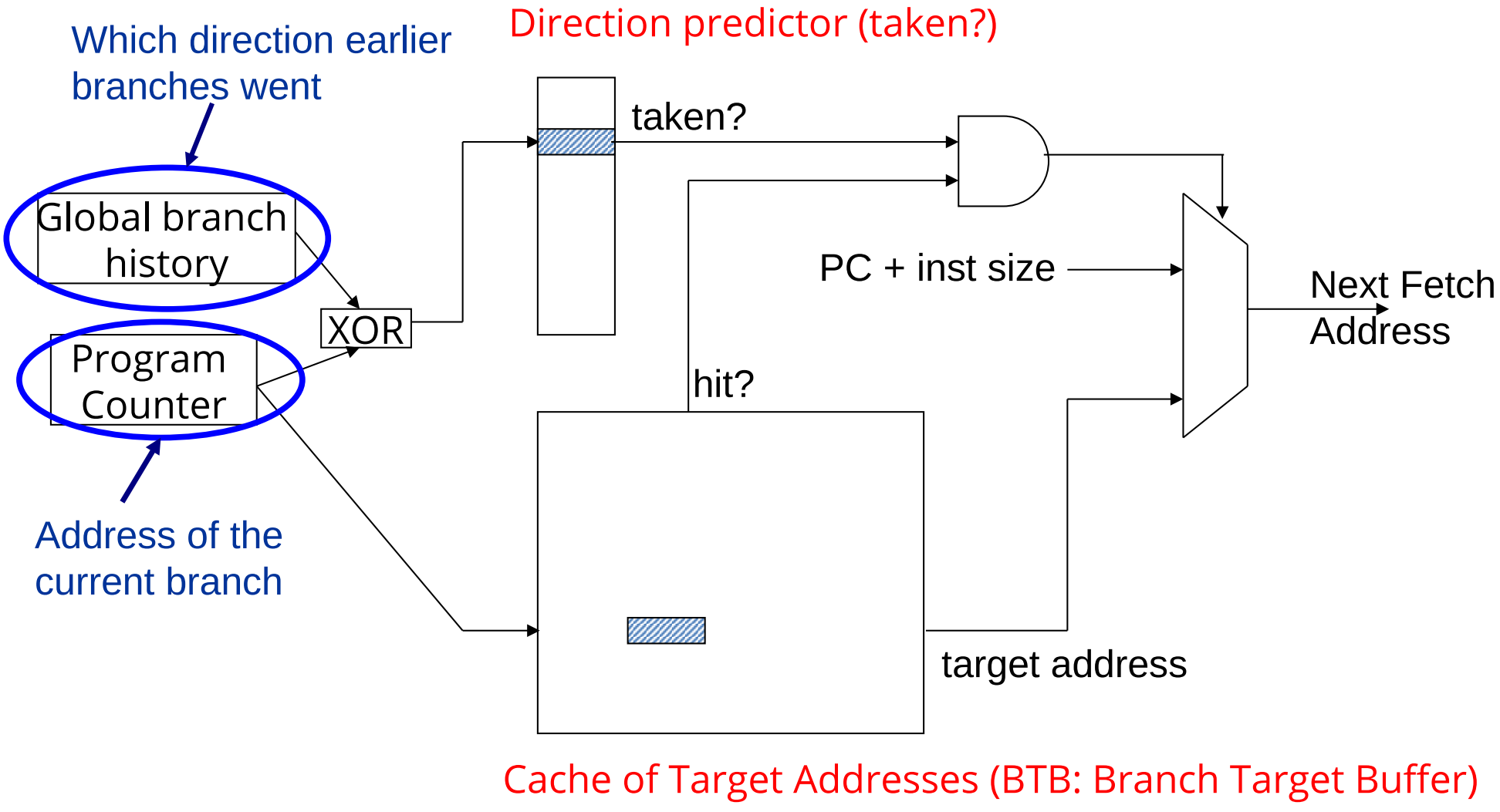
- **Idea: add another cache!**
- Store the target address from previous instance and access it with the PC.
- This is called the *Branch Target Buffer (BTB)* or the *Branch Target Address Cache*.

Fetch Stage with BTB and Direction Prediction



Always taken CPI = [1 + (0.20*0.3) * 2] = 1.12 (70% of branches taken)

More Sophisticated Branch Direction Prediction



Three Things to be Predicted

Recall there are three things which need to be predicted at the fetch stage:

- Whether the fetched instruction is a branch
- (Conditional) branch *direction*
- Branch *target address* (if taken)

We can accomplish (3) using a BTB

- We remember the target address computed the last time the branch was executed

We can accomplish (1) using a BTB

- If the BTB has an entry for the program counter, then it must be a branch.
- Alternatively, we can store *branch metadata* bits in the instruction cache

But how do we accomplish (2)?

Branch Direction Prediction

- Compile-time (static)
  - Always not taken
  - Always taken
  - BTFN (backwards taken, forward not taken)
  - Profile-based (likely direction)
  - More advanced: program-analysis-based (likely direction)
  - Programmer-based
- Run time (dynamic)
  - Last time prediction (single bit)
  - More advanced: two-bit counter based prediction
  - More advanced: two-level prediction (global vs. local)
  - Hybrid

Static Branch Prediction

Simple Schemes

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: $\sim 30 - 40\%$ (for conditional branches)
  - The compiler can layout code such that the likely path is the "not-taken" path which will lead to more effective prediction.
- Always taken
  - No direction prediction
  - Better accuracy: $\sim 60 - 70\%$ (for conditional branches)
    - Backwards branches (mostly loop branches) are usually taken
    - Backwards branch: target address is lower than branch instruction's PC
- Backwards taken, forward not-taken (BTFN)
  - Predict backwards (loop) branches are taken, others are not-taken

Practice: Static Branch Prediction: Worksheet Problem 1

Consider the following code once it reaches steady-state:

```
do {
    for (int i = 0; i < 4; i++) {
        // increment something
    }
    for (int j = 0; j < 8; j++) {
        // increment something
    }
    k++;
} while (k < 1000000000)
```

a. What is the branch prediction accuracy for an always not-taken (PC+4 prediction) branch predictor?

b. What is the branch prediction accuracy for an always taken branch predictor?

Profile Based

Idea: the compiler determines the likely direction for each branch using a profiling run. Then it encodes the direction as a *hint bit* in the branch instruction format.

- Pro: per-branch prediction (more accurate than simple schemes) are accurate if profile is representative. :
- Con: requires hint bits in the branch instruction format
- Con: accuracy depends on dynamic branch behavior:
  - `TTTTTTTTTTNNNNNNNNNN` $\rightarrow$ 50% accuracy
  - `TNTNTNTNTNTNTNTNTNTN` $\rightarrow$ 50% accuracy
- Con: accuracy depends on the representativeness of the profile input set

Program-Based

Idea: use heuristics based on program analysis to determine statically-predicted direction

- Example opcode heuristic: predict `BLEZ` as not-taken (negative integers used as error values in many programs)
- Example loop heuristic: predict a branch guarding a loop execution as taken (i.e., predict the loop will execute)
- Pointer and floating-point comparisons: predict not equal

- Pro: does not require profiling
- Con: heuristics may not be representative or could just be incorrect
- Con: requires compiler analysis and ISA support

Ball and Larus, *Branch prediction for free*, PLDI 1993. (20% misprediction rate)

Programmer-Based

Idea: the programmer provides the statically-predicted direction

This can be done via *pragmas* in the programming language that qualify a branch as likely-taken versus not-likely-taken.

- Pro: does not require profiling or program analysis
- Pro: programmer may know some branches in their program better than other analysis techniques
- Con: requires programming language, complier, and ISA support
- Con: burdens the programmer

Sidebar: Pragmas

Pragmas are keywords that enable a programmer to convey hints to lower levels of the transformation hierarchy.

For example:

```
if (likely(x)) { ... }
if (unlikely(x)) { ... }
```

Many other hints an optimizations can be enabled with pragmas. For example, whether or not a loop can be parallelized. For example:

```
#pragma omp parellel
```

explicitly instructs the compiler to parallelize the given segment of code.

Combining Approaches

All of the previous techniques can be combined: profile-, program-, and programmer-based.

How might you do that?

All of these techniques have one main disadvantage in common: *they 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 overhead).
- What is a dynamic compiler?
  - Self-optimizing code
  - Java JIT (just-in-time) compiler, the .NET CLR (common language runtime), V8 (JavaScript JIT), PyPy (Python JIT)

Dynamic Branch Prediction

Dynamic Branch Prediction

Idea: predict branches based on *dynamic* information (collected at runtime).

Pros

- 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 problems go away!

Disadvantages

- More complex (requires more hardware)

Dynamic Branch Prediction: Last-Time Predictor

A *last time predictor* stores a single bit per branch in the BTB indicating which direction the branch went last time it executed.

`TTTTTTTTTTNNNNNNNNNN` branch pattern results in 90% accuracy.

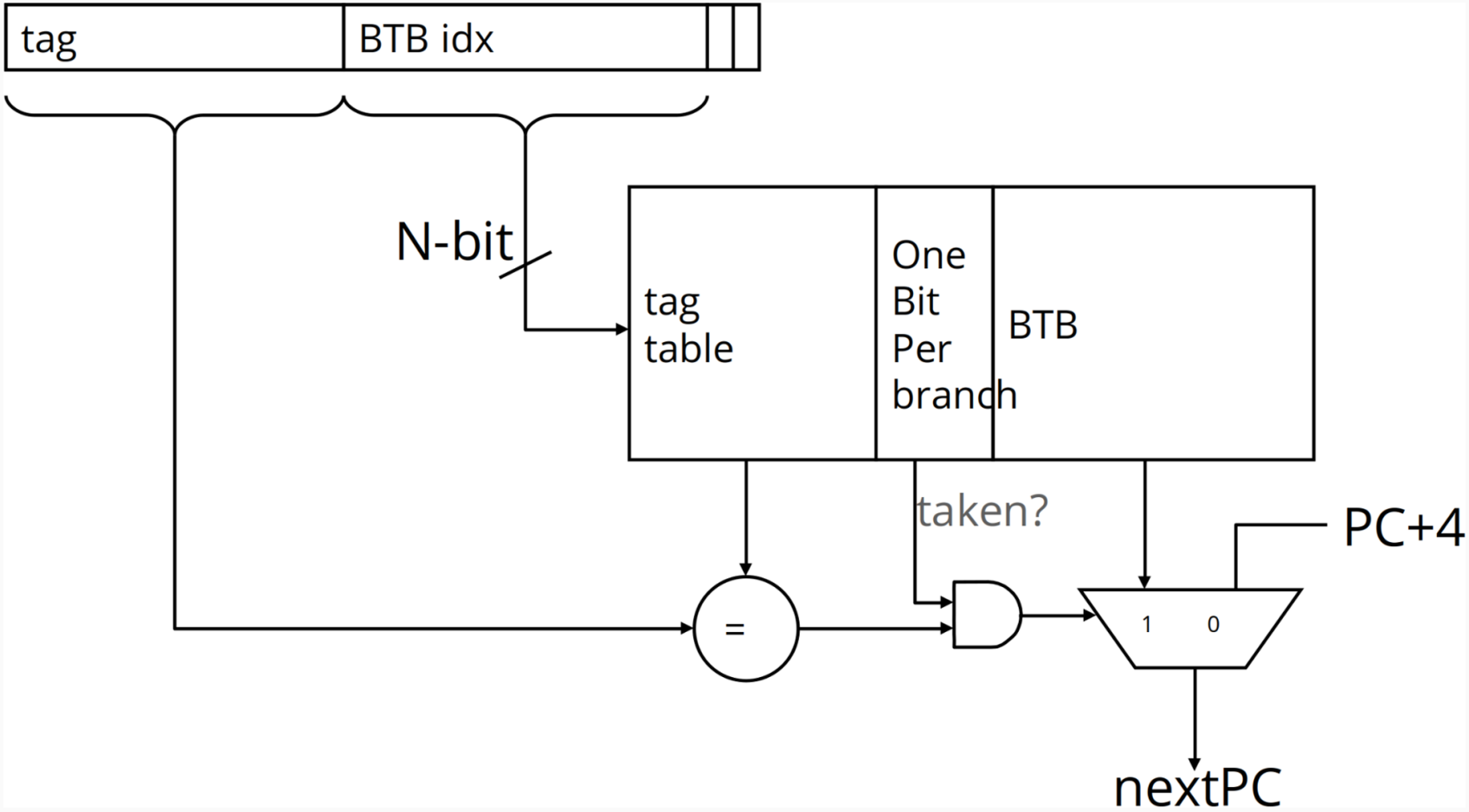
This predictor *always* mispredicts the last iteration and the first iteration of a loop branch. Thus, the accuracy for a loop with N iterations is $(N-2)/N$.

Pro: loop branches for loops with large number of iterations

Con: loop branches for loops with small number of iterations. `TTTTTTTTTTNNNNNNNNNN` branch pattern results in 90% accuracy.

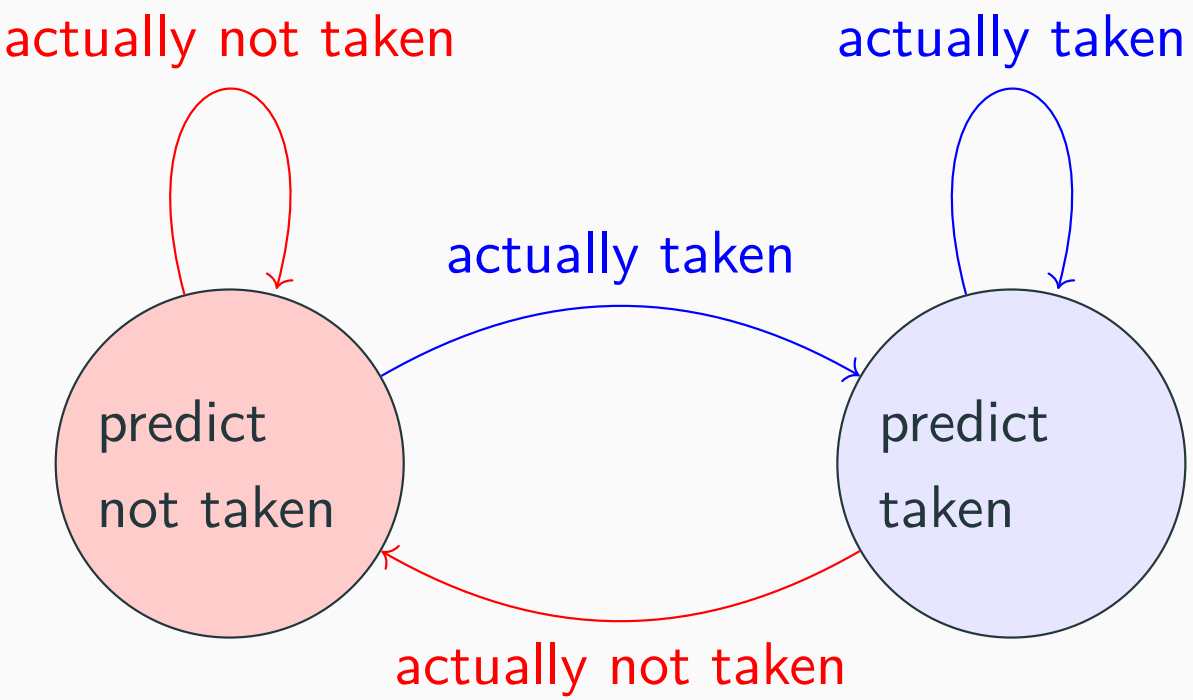
CPI: $1 + (0.2 \times \underbrace{0.15}_{\text{assume 85\% accuracy}}) \times 2 = 1.06$

Last-Time Predictor: Implementation



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

Last-Time Predictor: State Machine



Practice: Dynamic Branch Prediction: Worksheet Problem 2

Consider the following code once it reaches steady-state:

```
do {
    for (int i = 0; i < 4; i++) {
        // increment something
    }
    for (int j = 0; j < 8; j++) {
        // increment something
    }
    k++;
} while (k < 1000000000)
```

a. What is the branch prediction accuracy for a 1-bit branch predictor?

Last-Time Predictor: Problem

A last-time predictor changes its prediction from `T` to `NT` or `NT` to `T` too quickly (even if the branch is mostly taken or mostly not-taken).

Idea: add hysteresis to the predictor so that prediction doesn't change on a single different outcome.

- Use two bits to track history of predictions for a branch instead of a single bit
- Can have two states for `T` or `NT` instead of one state for each

(See: Smith, *A Study of Branch Prediction Strategies*, ISCA 1981.)

Dynamic Branch Prediction: Two-Bit Counter (2BC) Predictor

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

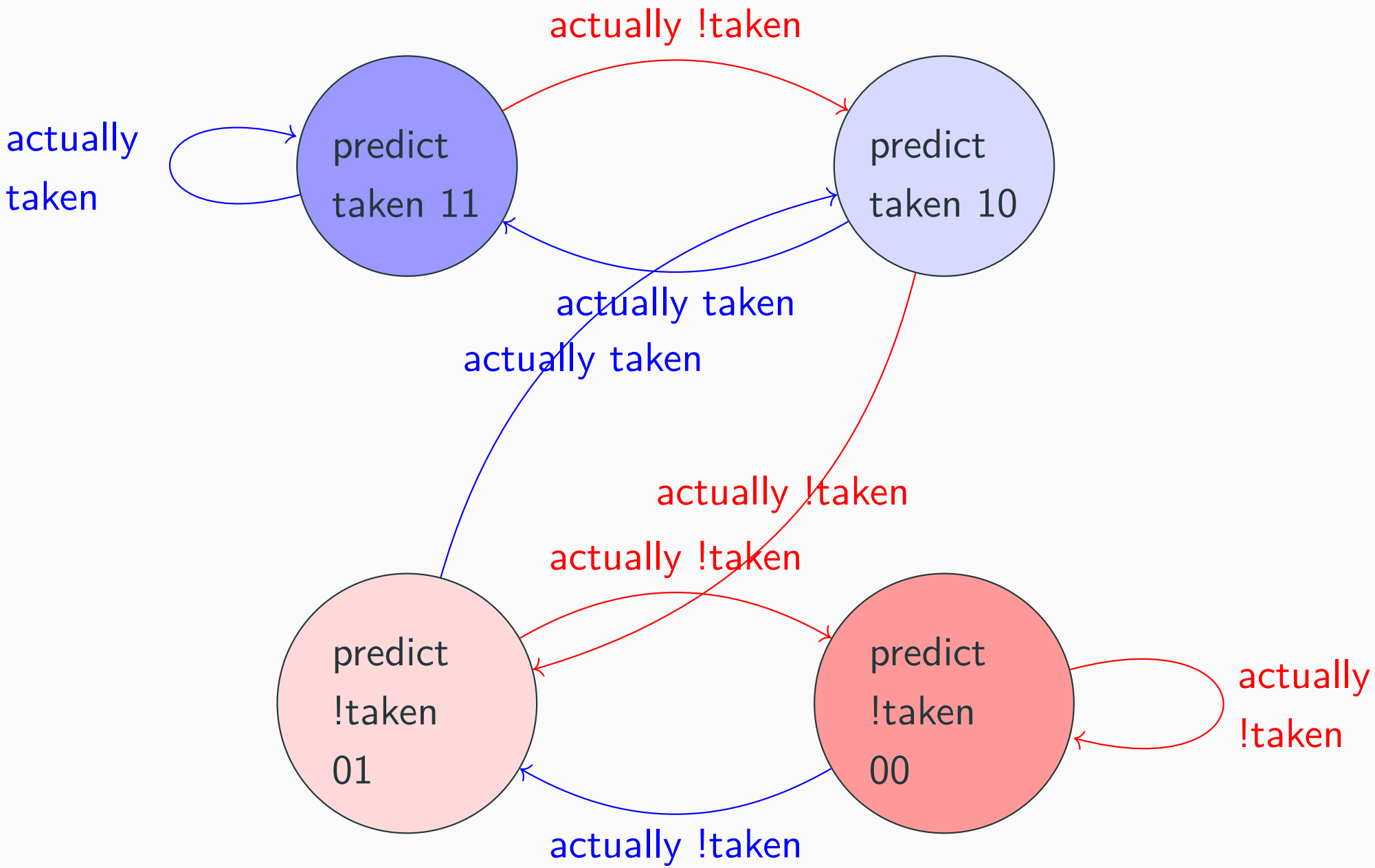
Accuracy for a loop with N iterations $= (N-1)/N$. And the `TNTNTNTNTNTNTNTNTNTN` pattern results in 50% accuracy (assuming counter is initialized to weakly taken)

Pro: better prediction accuracy

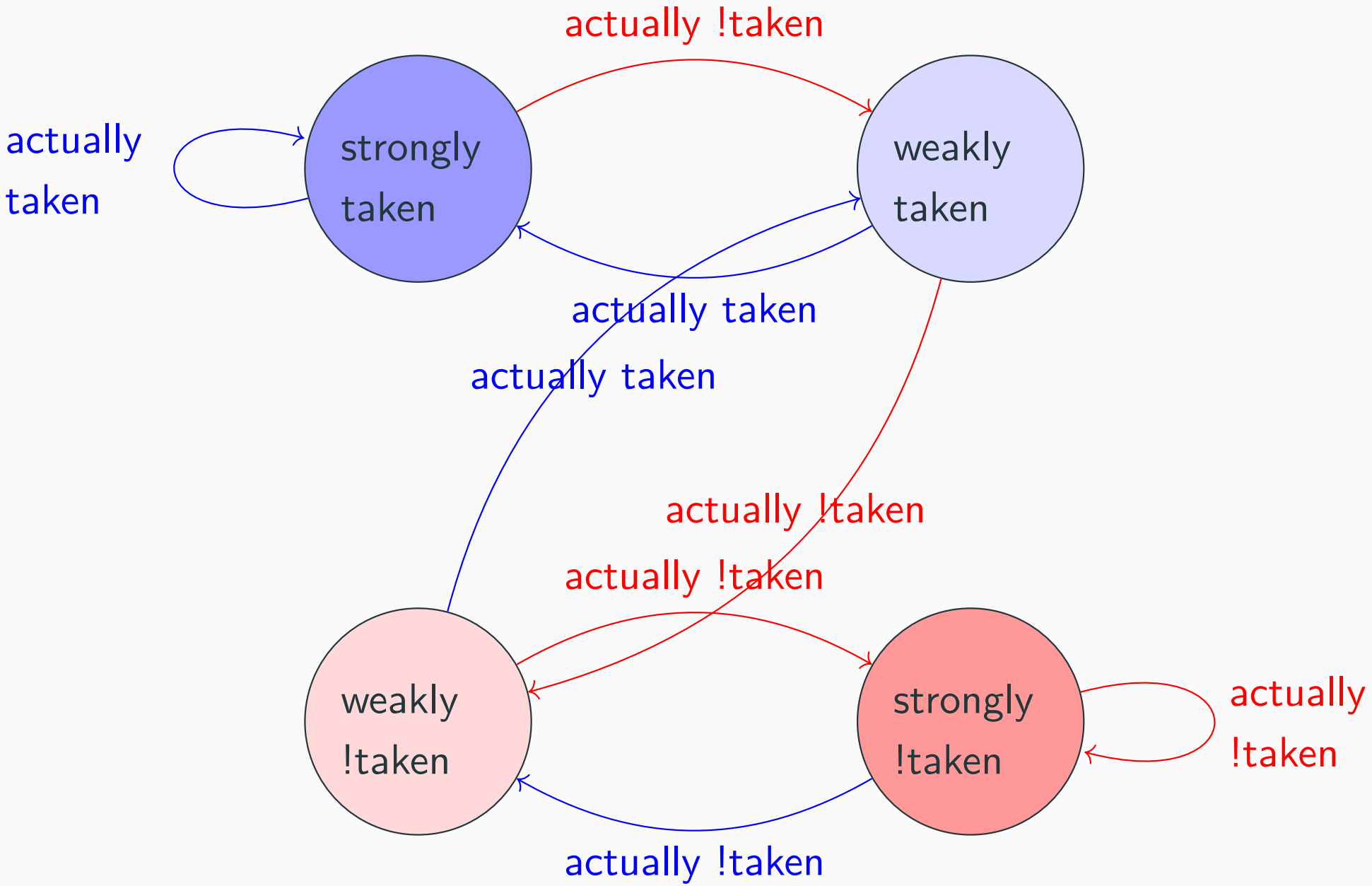
- CPI: $1 + (0.2 \times 0.1) \times 2 = 1.04$ (asuming 90% accuracy)

Con: more hardware cost

Two-Bit Predictor: State Machine



Two-Bit Predictor: State Machine



Practice: Dynamic Branch Prediction: Worksheet Problem 2

Consider the following code once it reaches steady-state:

```
do {
    for (int i = 0; i < 4; i++) {
        // increment something
    }
    for (int j = 0; j < 8; j++) {
        // increment something
    }
    k++;
} while (k < 1000000000)
```

b. What is the branch prediction accuracy for a 2-bit branch predictor?

Two-Bit Predictor: Is This Enough?

- We can achieve $\sim 85 - 90\%$ accuracy for many programs with two-bit counter based prediction
- Is this good enough?
- Let's quantify the branch problem.

Quantifying the Branch Problem

Back to the Drawing Board

Control flow instructions (branches) are frequent (15-20%) of all instructions.

Problem: next fetch address after a control-flow instruction is determined after N cycles in a pipelined processor.

- N cycles: (minimum) branch resolution latency
- Stalling on a branch wastes instruction processing bandwidth (that is, it reduces IPC)
- $N \times W$ instruction slots are wasted (W: pipeline width)

So, we use branch prediction to try and fill up the pipeline after the branch.

Problem: we need to determine the **next fetch address** when the branch is fetched (to avoid a pipeline bubble)

Importance of the Branch Problem

Assume a 5-wide superscalar pipeline with 20-cycle branch resolution latency.

- How long does it take to fetch 500 instructions?
- 100% accuracy
  - 100 cycles (all instructions fetched on the correct path)
  - No wasted work
- 99% accuracy
  - 100 (correct path) + 20 (wrong path) = 120 cycles
  - 20% extra instructions fetched
- 98% accuracy
  - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
  - 40% extra instructions fetched
- 95% accuracy
  - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
  - 100% extra instructions fetched

Global and Local Branch Prediction

Global and Local Branch Prediction: Intuition

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 to the outcome of the next branch.

Example: if first branch taken (false), then second branch taken (false):

```
if (cond1)
    ...
if (cond1 && cond2)
    ...
```

Example: if first branch not taken (true), then second branch is definitely taken (false):

```
if (cond1) a = 2;
...
if (a == 0) ...
```

Global Branch Correlation II

```
if (cond1)  // branch X
...
if (cond2)  // branch Y
...
if (cond1 && cond2)  // branch Z
```

- if X or Y are taken, then Z is also taken
- if X and Y are not taken, then Z is also not taken

Global Branch Correlation III

From Eqntott, SPEC 1992

```
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.

In other words: make a prediction based on the outcome of the branch the last time the same global branch history was encountered.

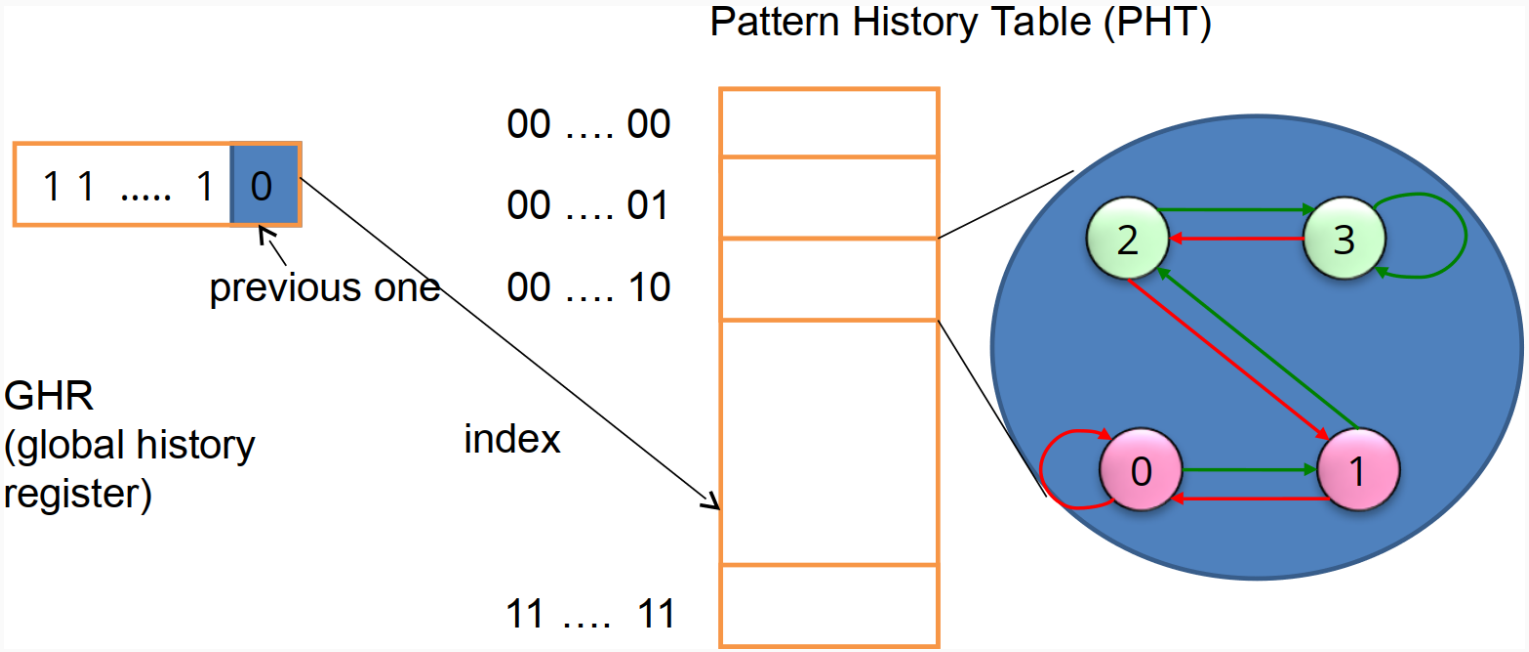
Implementation:

- Keep track of the "global T/NT history" of all branches in a *Global History Register (GHR)*.
- Use GHR to index into a *Pattern History Table* that recorded the outcome that was seen for each GHR value in the recent past.

Now we have a global history/branch predictor. It uses two levels of history (GHR + history at that GHR).

Two-Level Global Branch Prediction

- Global Branch History Register (N bits) — stores the direction of the last N branches.
- Table of saturating counters for each history entry — stores 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?

Consider the following code:

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

After the initial startup time, the conditional branches have the following behaviour (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

McFarling, *Combining Branch Predictors*, DEC WRL TR 1993.

Practice: Global vs. Local Branch Prediction: Worksheet Problem 3

Consider the following code once it reaches steady-state:

```
do {
    for (int i = 0; i < 4; i++) {
        // increment something
    }
    for (int j = 0; j < 8; j++) {
        // increment something
    }
    k++;
} while (k < 1000000000)
```

Assume that the PHT contains 2-bit counters.

a. What is the branch prediction accuracy for a global branch predictor with a 5-bit history?

Global Branch Predictor: How Large Must it Be?

Question: How much storage will it take to store a global predictor that uses 3-bit counters and that produces an index by XOR-ing 12 bits of branch PC with 12 bits of global history?

The index is 12 bits wide, so the table has 2^{12} counters. Each counter is 3 bits wide. Thus, the total required is:

$$3 \times 2^{12} = 12 \text{ Kib or } 1.5 \text{ KiB}$$

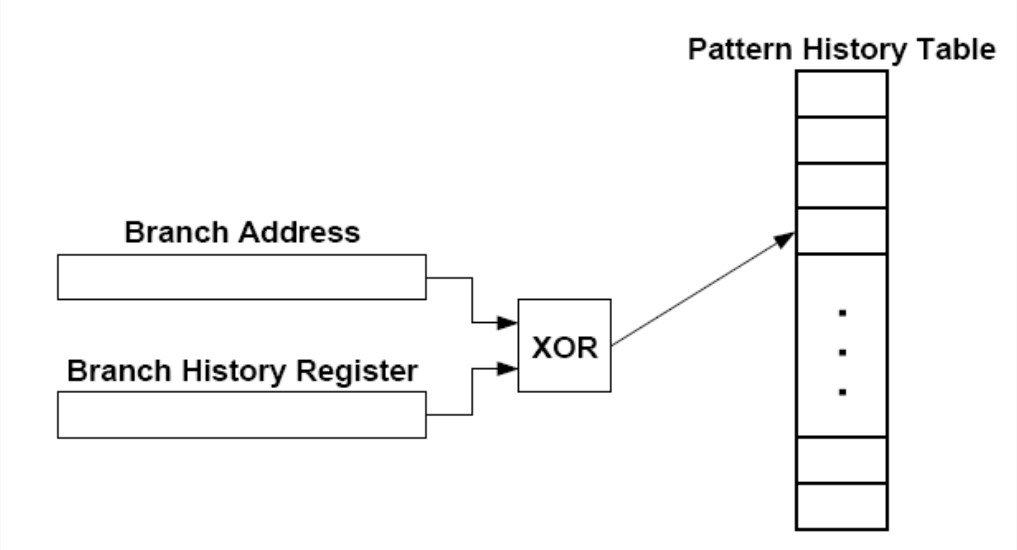
Intel Pentium Pro Branch Predictor

- 4-bit global history register
- Multiple pattern history tables (of 2-bit counters)
  - The pattern history table to use is determined by lower-order bits in the branch address.

Improving Global Predictor Accuracy

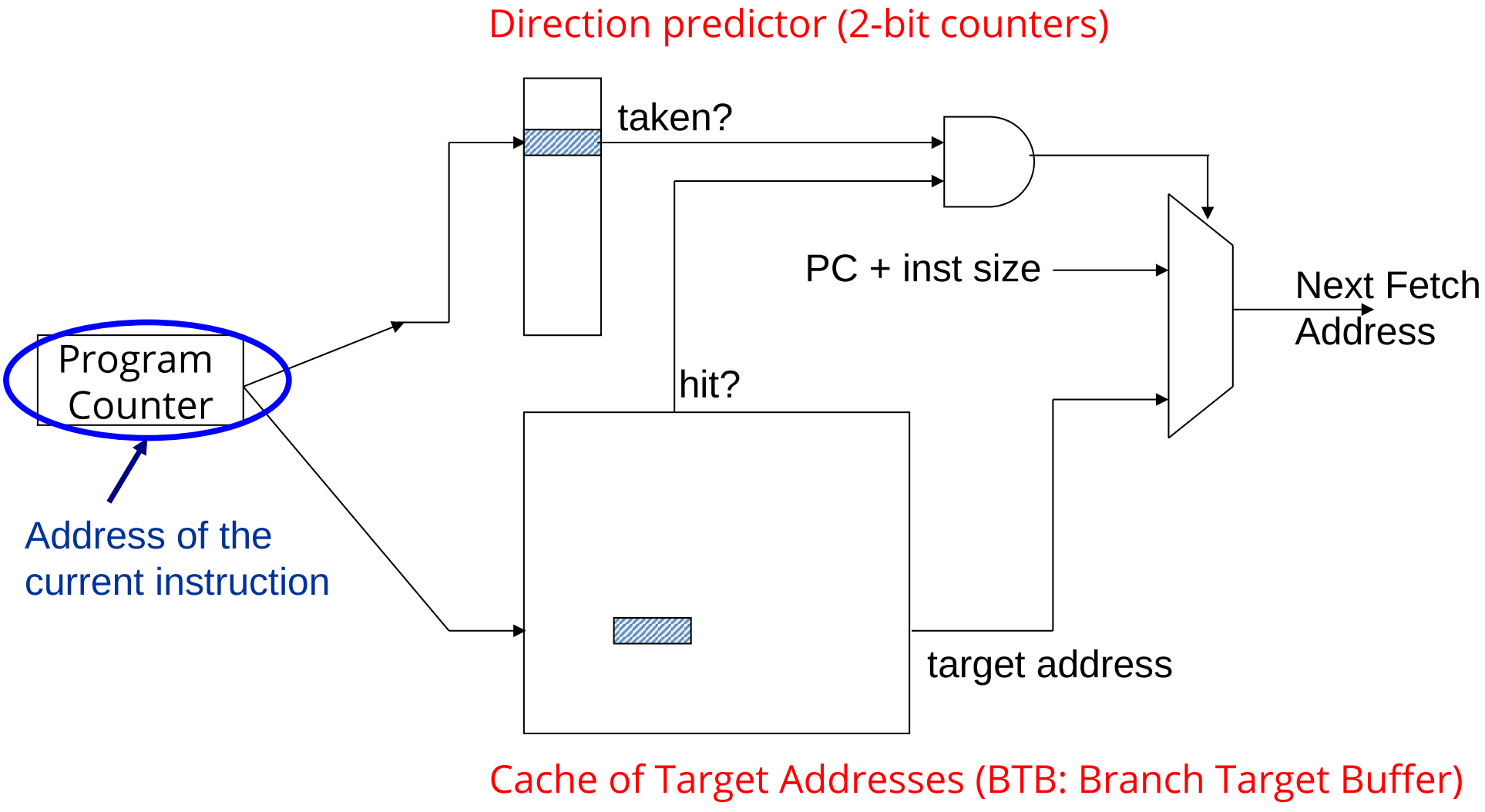
Idea: add more context information to the global predictor to take into account which branch is being predicted.

- *Gshare predictor* because the GHR and branch PC are XORed.
- Pro: more context information
- Pro: better utilization of the PHT
- Con: increases access latency

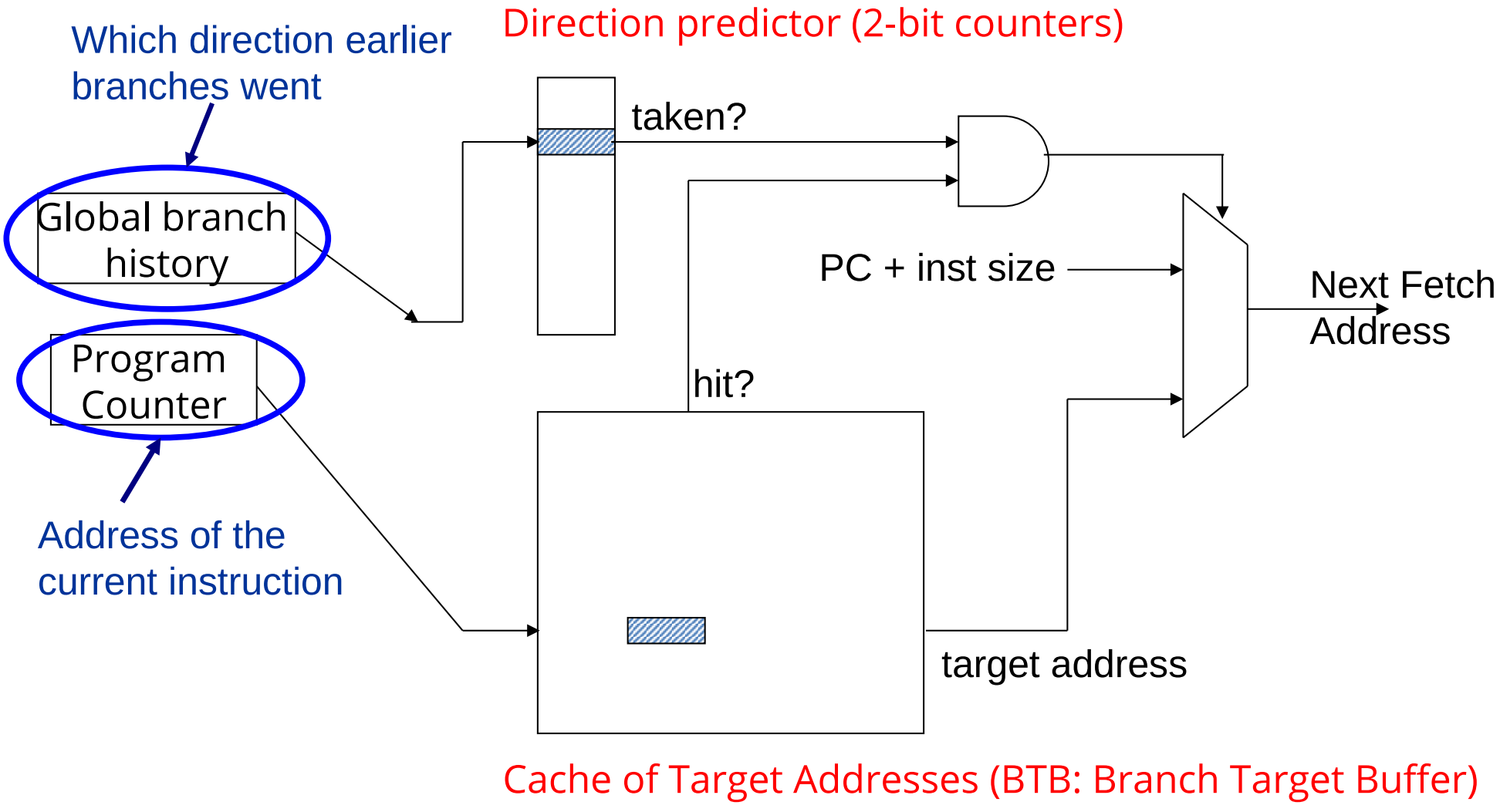


McFarling, *Combining Branch Predictors*, DEC WRL TR 1993.

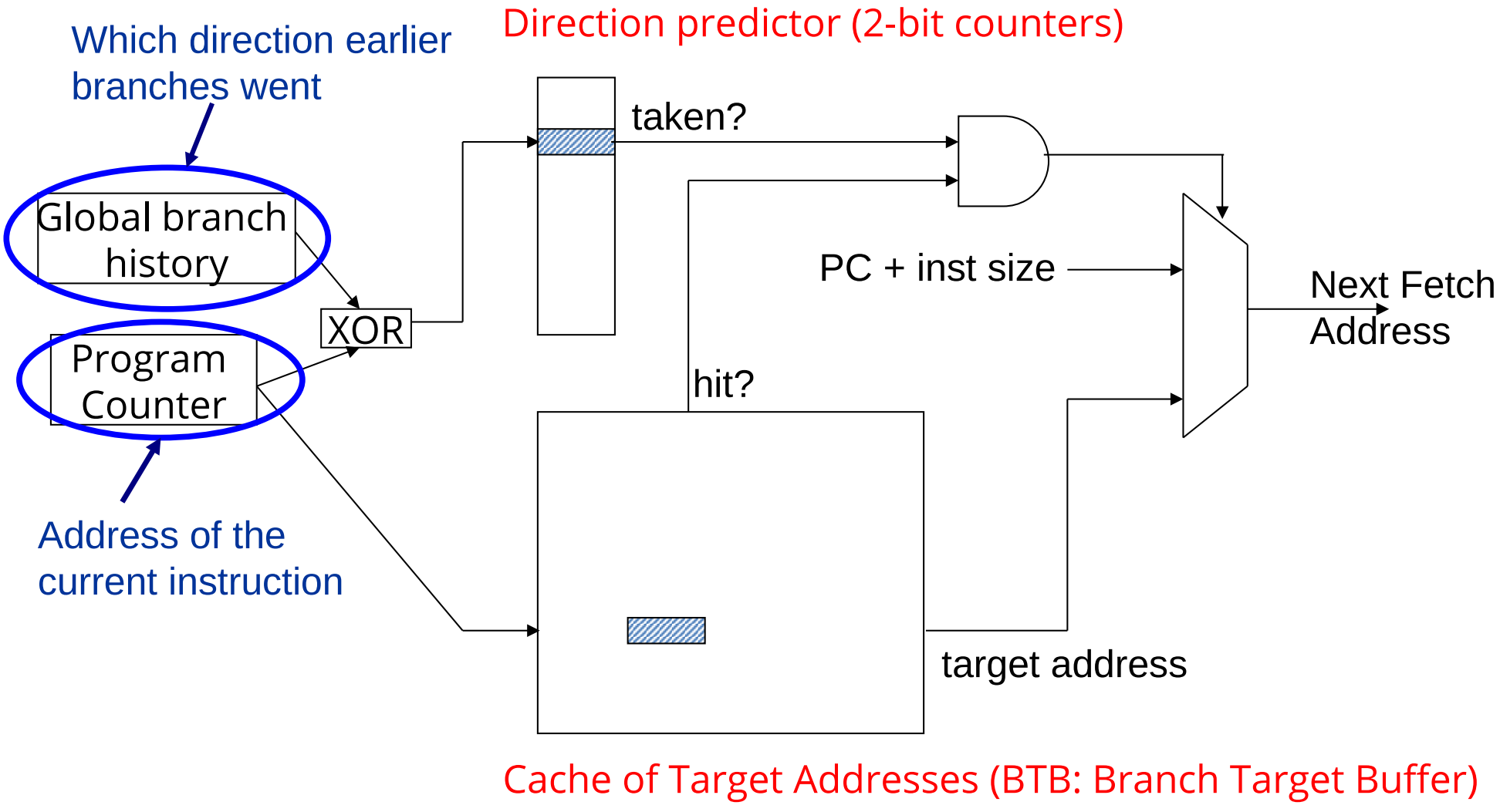
Review: One-Level Branch Predictor



Two-Level Global History Branch Predictor



Two-Level Gshare Branch Predictor



Can We Do Better?

- 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.
  This leads to favouring **global branch prediction**.
- **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).
  This leads to favouring **local branch prediction**.

Local Branch Correlation

Consider the following code:

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

If the loop test is done at the end of the body, the corresponding branch will execute the pattern will be `TTTN` (`T` = taken, `N` = not taken).

Say we run this loop n times. We don't know how the loop behaves the first time the loop is run. However, on subsequent loop runs, **if we know 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.

Capturing Local Branch Correlation

Idea: have a per-branch history register

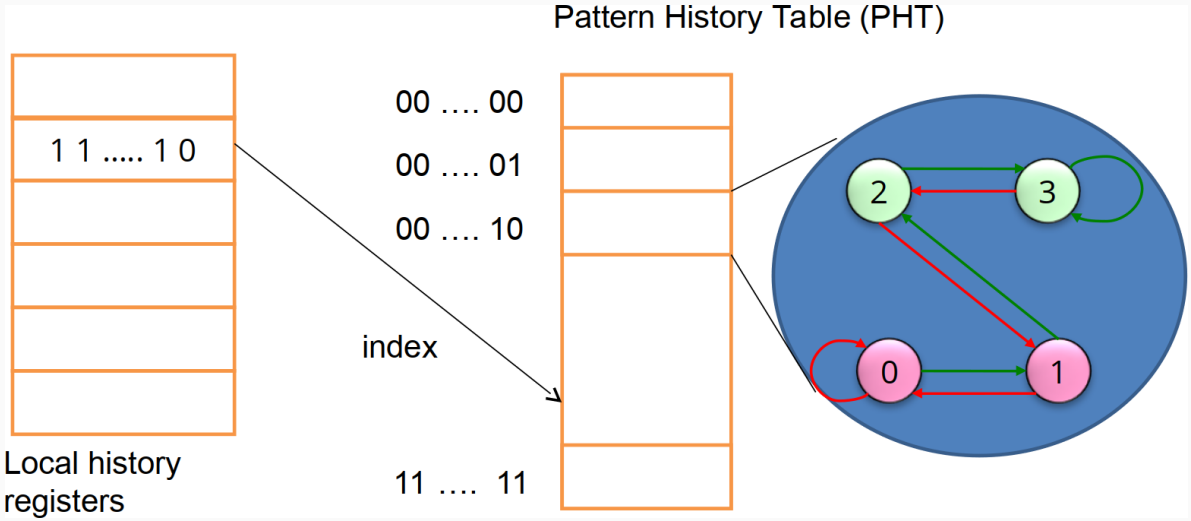
- Similar to the global history register.
- Associate the predicted outcome of a branch with the "T/NT history" of the same branch.

This will make a prediction based on the outcome of the branch last time the same local branch history was encountered.

- This is called the *local history branch predictor*
- It requires two levels of history:
  1. per-branch history register
  2. history at that history register value

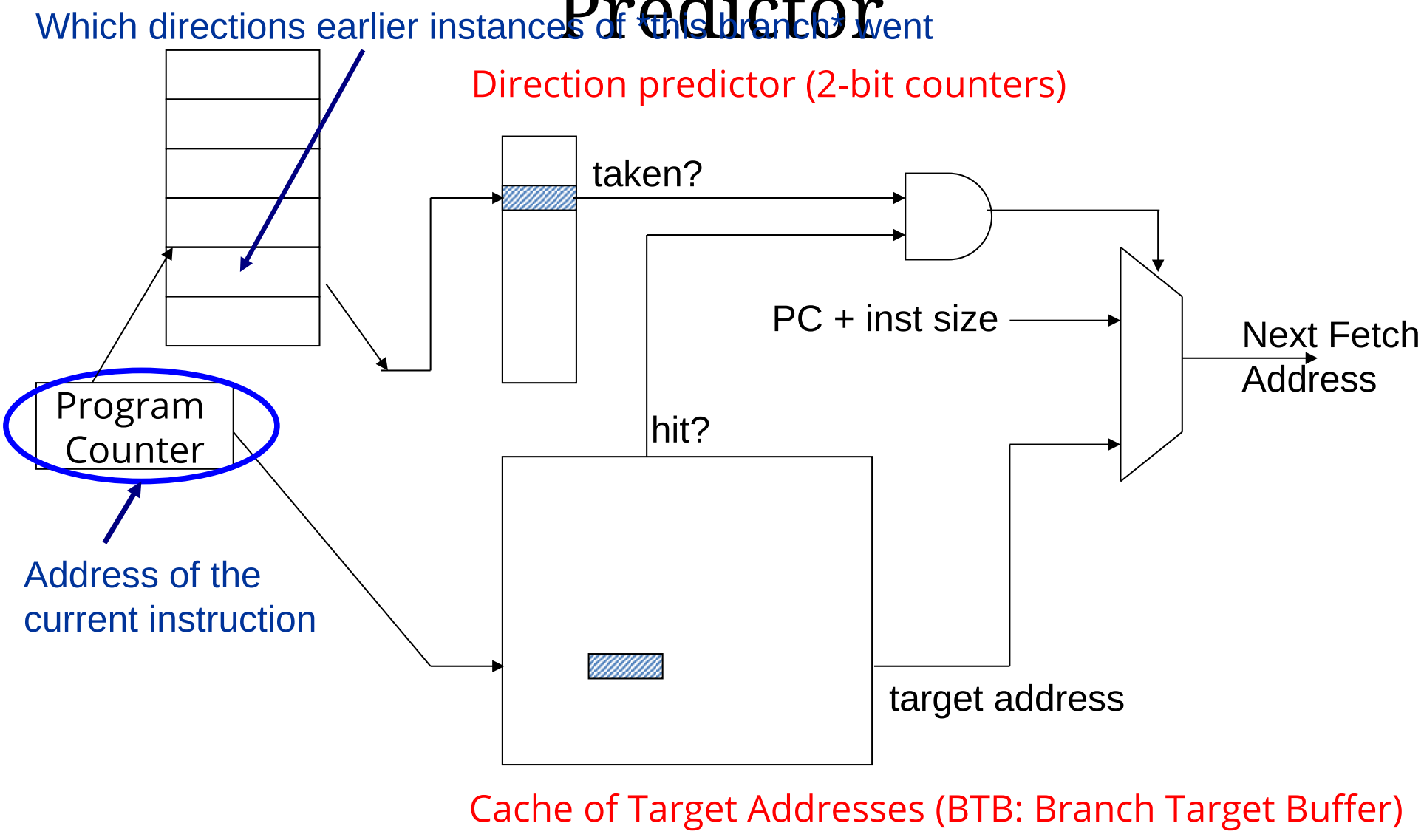
Two-Level Local Branch Prediction

- **First level:** A set of *local history registers* (N bits each)
  We select which local history register to use based on the PC of the branch.
- **Second level:** table of saturating counters for each history item.
  We use this to see 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



Practice: Global vs. Local Branch Prediction: Worksheet Problem 3

Consider the following code once it reaches steady-state:

```
do {
    for (int i = 0; i < 4; i++) {
        // increment something
    }
    for (int j = 0; j < 8; j++) {
        // increment something
    }
    k++;
} while (k < 1000000000)
```

Assume that the PHT contains 2-bit counters.

b. What is the branch prediction accuracy for a local branch predictor with a 5-bit history?

Additional Considerations

Hybrid Branch Predictors

Idea: use more than one type of predictor (multiple algorithms) and select the "best" prediction.

For example, a hybrid of 2-bit counters and a global predictor.

Pros:

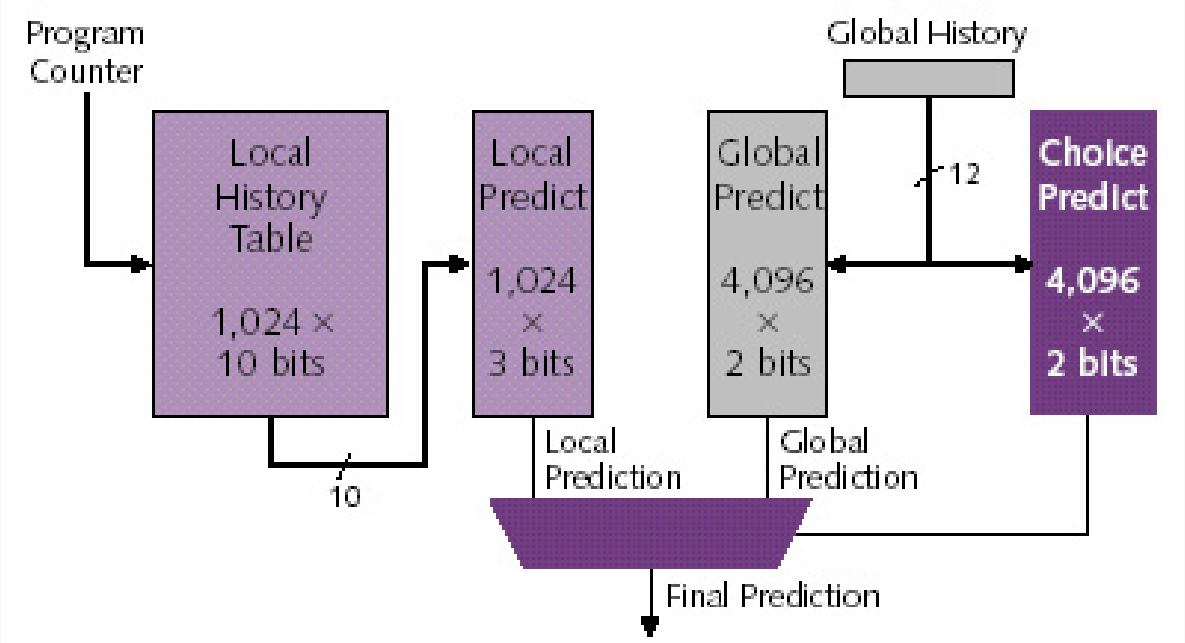
- Better accuracy: different predictors are better for different branches.
- Reduced *warmup* time (faster-warmup predictor used until the slower-warmup predictor is warm)

Cons:

- Need *meta-predictor* or *selector*
- Results in longer access latency

McFarling, *Combining Branch Predictors*, DEC WRL TR 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.

Branch Prediction Accuracy (Example)

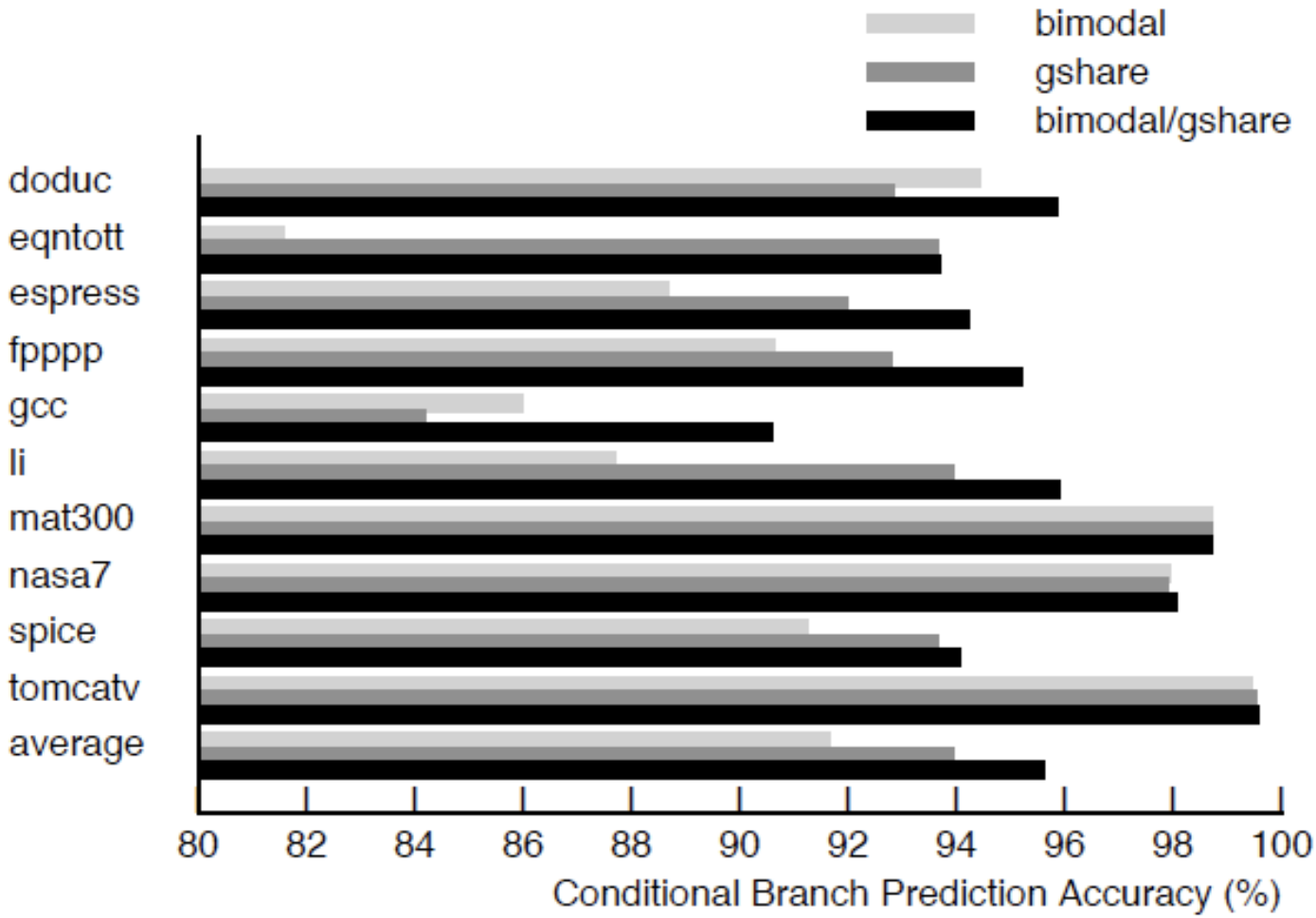


Figure 13: Combined Predictor Performance by Benchmark

Biased Branches

Observation: many branches are biased in one direction (example: 99% taken)

Problem: these branches *pollute* the branch prediction structures. This makes prediction of the other branches more difficult by causing "interference" in branch prediction tables and history registers.

Solution: detect such biased branches, and predict them with a simpler predictor.

Chang et al., *Branch classification: a new mechanism for improving branch predictor performance*, MICRO 1994.

Predicated Execution: What is Predication? i

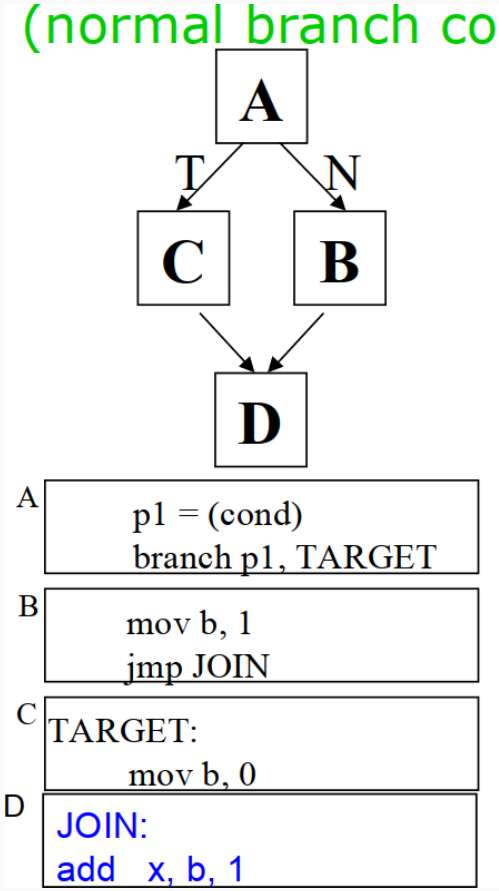
Idea: compiler converts control dependence into data dependence, thus eliminating the branch!

- Each instruction must have a predicate bit set based on the predicate computation.
- Only instructions with *true* predicates are committed (others turned into NOPs).

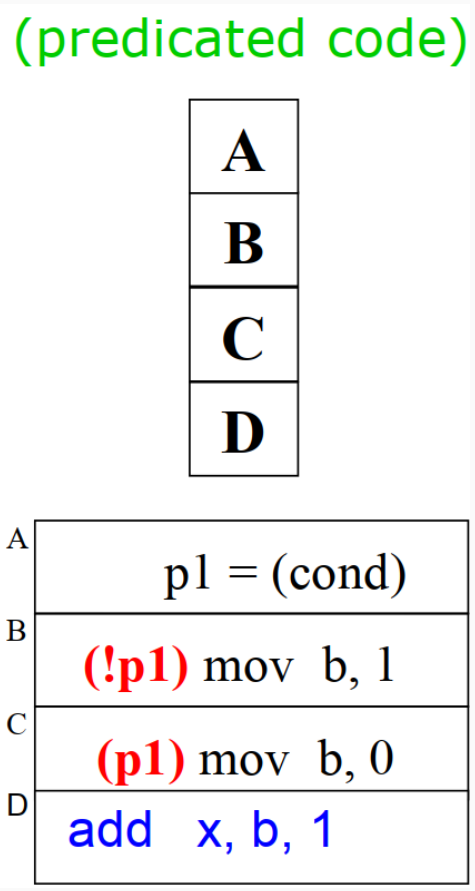
Predicated Execution: What is Predication? ii

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

(normal branch code)

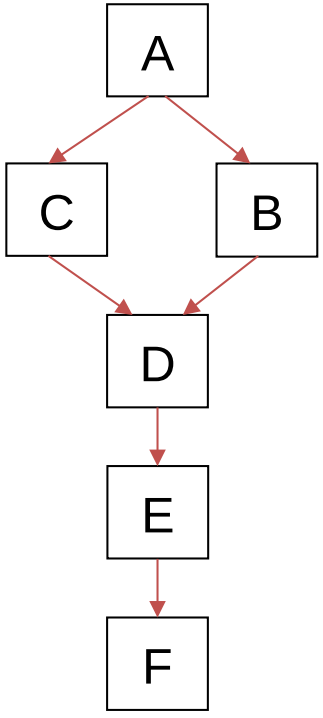


(predicated code)



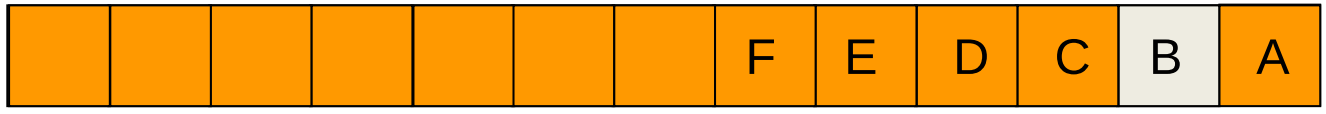
Predicated Execution (II)

- 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: Pros and Cons

Pros:

- Eliminates mispredictions for hard-to-predict branches.

Cons:

- Causes useless work for branches that are easy to predict.
- Additional hardware and ISA support.
- Cannot eliminate all hard-to-predict branches.

Conditional Execution in the ARM ISA

- Almost all ARM instructions can include an optional condition code.
- An instruction with a condition code is executed only if the condition code flags meet the specified condition.

Wish Branches: Best of Both Worlds?

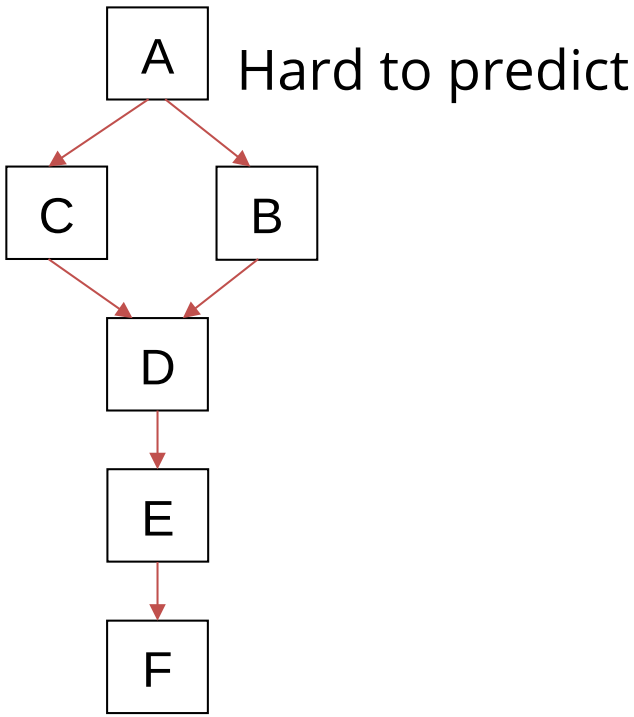
Wouldn't it be nice if the branch is eliminated (predicated) only when it would actually be mispredicted, and is predicted when it would actually be correctly predicted?

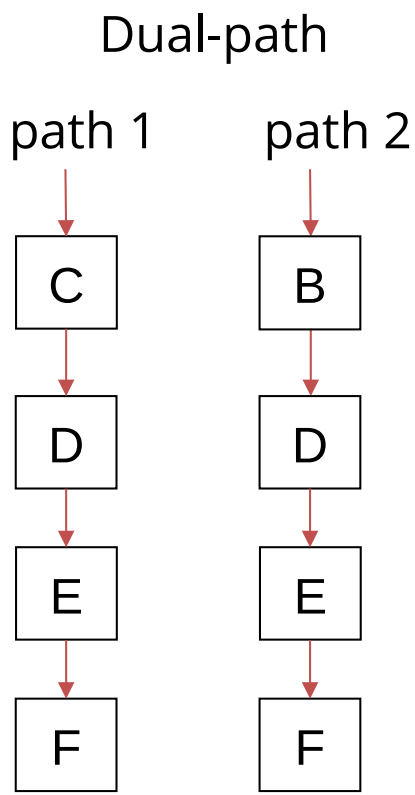
Enter **wish branches**.

- The *compiler* generates code (with wish branches) that can be executed *either* as predicated *or* non-predicated code (normal branch code).
- The *hardware decides* to execute predicated code or normal branch code at run-time based on the confidence of the branch prediction.
- Easy-to-predict $\rightarrow$ use normal branch code
- Hard-to-predict $\rightarrow$ use predicated code

Kim et al., *Wish Branches: Enabling Adaptive and Aggressive Predicated Execution*, MICRO 2006, IEEE Micro Top Picks, Jan/Feb 2006.

Dual-Path Execution versus Predication





Dual-Path Execution versus Predication