CS6L047: Advanced Computer Architecture

Branch Prediction

Motivation for Branch Prediction

- Branches resolved in **EX** stage ➡ **2-cycle penalty** for all branches
- Branches resolved in **ID** stage ➡ **1-cycle penalty** for all branches + h/w at ID stage (Adder, Comparator, Wider pipeline register)
- Branches decision in **IF** stage
  - ➡ **0-cycle penalty** for correctly predicted branch
  - ➡ **1-cycle penalty** for mis-predicted branch [if branch resolved at ID]
  - ➡ **2-cycle penalty** for mis-predicted branch [if branch resolved at EX]

Motivation for Branch Prediction

- Branche decision in **IF** stage
  - ➡ **0-cycle penalty** for correctly predicted branch
  - ➡ **1-cycle penalty** for mis-predicted branch [if branch resolved at ID]
  - ➡ **2-cycle penalty** for mis-predicted branch [if branch resolved at EX]

- **Philosophy**: *"Doing something is better than doing nothing."*
  - Instead of stalling the pipeline on a branch, we **speculatively fetch** the next instruction.
  - Even with imperfect accuracy, the **performance gains often outweigh the cost of mispredictions**.
  - This has been a **major research focus** over the last two decades.

Processor Family	Architecture	Branch Prediction Technique
Intel Core (i9/i7/i5)	x86 (Alder Lake, Raptor Lake)	Hybrid dynamic predictors (e.g., TAGE, Bimodal, BTB)
AMD Ryzen (Zen 3/4)	x86 (Zen architecture)	TAGE-like predictors, loop predictors
Apple M1/M2/M3	ARM64 (Apple Silicon)	Dynamic prediction with BTB and global history
ARM Cortex-X2/A78	ARMv9	Dynamic predictors with pattern history tables
IBM POWER10	POWER ISA	Complex dynamic predictors with global/local history
Experimental (TinyBERT)	Research (2024–2025)	Transformer-based ML predictor
Experimental (ESP)	Research (University of Colorado)	Static ML predictor using decision trees & neural nets

“Predict the Future”

- Branch Prediction Championship: ISCA 2025 (Flagship conference, sponsored by ARM)
- **Test of Time Award**
  - Given by Flagship conferences like **HPCA, ISCA, MICRO** to honor papers that have had a **lasting impact** on the field—typically 15 to 20 years after their original publication
  - **Daniel A. Jiménez**, received the **HPCA Test of Time Award in 2019** for his 2001 paper titled *“Dynamic Branch Prediction with Perceptrons*
  - ML models—even simple ones like perceptrons—require **more silicon area**, **power**, and **latency** than traditional predictors.
  - Commercial chips must balance **performance, cost, and energy efficiency**

Branch Prediction Types

- **Static Prediction**:
  - Decided at compile-time or hardcoded in hardware.
  - No adaptation to runtime behavior.
  - Simpler, cheaper, but less accurate.
  - Used in early RISC processors, embedded systems, and compiler-optimized code.
- **Dynamic Prediction**:
  - Learns from actual branch outcomes during execution.
  - Adapts to patterns (e.g., loops, conditionals).
  - Requires more hardware but delivers **much higher accuracy**.
  - Used in modern CPUs (Intel, AMD, Apple) with predictors like **TAGE**, **tournament**, and **hybrid** models.

Fixed/Static Branch Prediction

- A **static prediction strategy** applied **uniformly** to all branch instructions.
- No dynamic learning or history tracking.

Strategy	Hardware Simplicity	Requires Early Target?	Common in Simpler Systems?	Used in Commercial Systems?
Predict Not-Taken	✅ Very simple	❌ No	✅ Yes	✅ Yes
Predict Taken	❌ More complex	✅ Yes	❌ Not usually	✅ Yes (with architectural support)

Branch Prediction Techniques Overview

Technique	Type	Prediction Strategy	Implementation Complexity	Performance Impact
Always Taken	Static	Assume every branch is taken	Simple hardware; good for loop-heavy code	Moderate
Always Not Taken	Static	Assume every branch is not taken	Easiest to implement; fetch next sequential instruction	Moderate
Delayed Branch	Static	Execute instructions after branch	Compiler schedules instructions in delay slots to avoid pipeline stalls	Depends on compiler
Compiler Directed	Static	Use hints (likely/unlikely)	Compiler inserts metadata to guide prediction	Better than fixed
Dynamic Branch Prediction	Dynamic	Learn from runtime behavior	Uses hardware structures (e.g., history tables, saturating counters, TAGE, etc.)	High accuracy

Branch Prediction

- **Static branch prediction.**
  - Always taken, always not taken
  - Delayed Branch
  - Compiler directed (branch likely, branch not likely)

- Dynamic branch prediction

Why Branch Prediction Works?

- Program behavior isn't random— the patterns are predictable.
- **Algorithmic Regularities**
  - Many algorithms (sorting, searching, loops) follow predictable control flows.
  - Example: A loop that runs 1000 times will likely take the branch 999 times and not take it once.
- **Data Regularities**
  - Data often follows patterns (e.g., sorted arrays, structured inputs).
  - Branches based on data comparisons (e.g., if x > threshold) tend to behave consistently across runs..

Dynamic predictor can learn and anticipate branch outcomes with high accuracy.

Predictor Overview

Feature	Static Prediction	Dynamic Prediction
Decision Basis	Fixed rule or compiler hint	Runtime behavior and history
Adaptability	❌ No	✅ Yes
Accuracy	Moderate (~60–70%)	High (often >90% with advanced predictors)
Hardware Complexity	Low	Higher (requires tables, counters, etc.)
Best Use Case	Simple or embedded systems	Modern CPUs, performance-critical apps

Dynamic Branch Prediction Steps

What Happens during IF stage with the Dynamic Branch Prediction enabled?

- Step 1: The CPU retrieves the instruction from memory using the Program Counter (PC).
  - At this point, it doesn't yet know if the instruction is a branch or not.

- Step 2: Branch Prediction Buffer (BPB) Access.

Dynamic Branch Prediction Steps

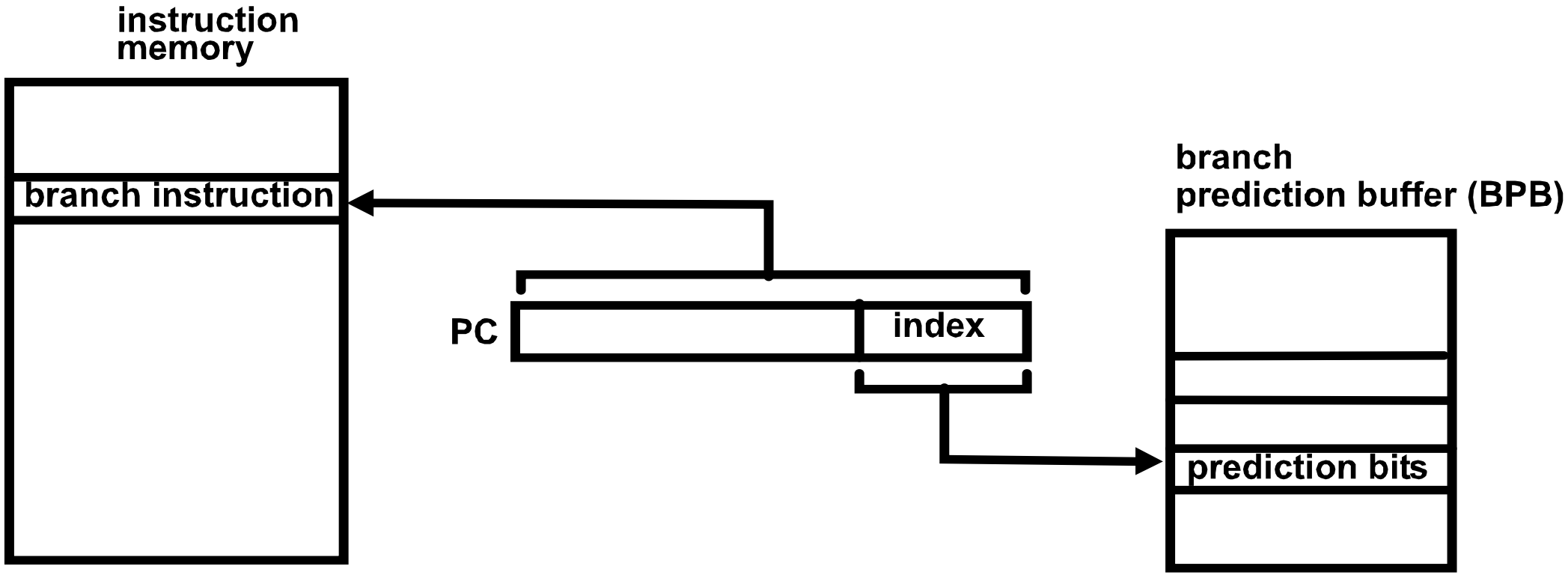
- Step 2: Branch Prediction Buffer (BPB) Access.
  - The PC is used to **index into the BPB** (also called Branch History Table or Branch Target Buffer).
  - If the instruction is a **branch**, the BPB provides:
    - **Direction prediction**: Will the branch be **taken or not**?
    - **Target prediction**: If taken, what is the next PC (**target address**)?
- Step 3: Speculatively Fetch next instruction.--- as per BPB's prediction
- Step 4: Validation: Check if prediction is correct or not (mis-prediction).

1-Bit Branch Predictor

- Why Use Low-Order Bits of PC for indexing the BPB?
  - The CPU keeps a **1-bit history** for each branch instruction.
  - That bit stores the **last outcome**:
    - **1** = last time it was **taken**
    - **0** = last time it was **not taken**
  - When the branch is encountered again, the CPU **predicts** the same outcome as last time. Incase of misprediction, flip the bit in BPB.
- Design/Implementation:
  - In a **Branch Prediction Buffer (BPB)** or **Branch History Table (BHT)**:
    - Each entry holds **1 bit**.
    - The table is indexed using the **low-order bits of the Program Counter (PC)**.

Branch Prediction Buffer (BPB)

- Branch Prediction Buffer (BPB) accessed with Instruction on I-Fetch



- Also called Branch History Table (BHT), Branch Prediction Table (BPT)

1-Bit Branch Predictor

- Why Use Low-Order Bits of PC for indexing the BPB?

  - Reason: The Low-order PC vary more frequently and help differentiate between nearby instructions.
  - Limitation: **multiple branches can map to the same entry**, causing **aliasing** (i.e., shared prediction history).

Loop Branch behavior

```
for (int i = 0; i < 10; i++) {
    // do something
}
```

- loop runs **10 times**:
  - The branch is **taken 9 times** (to repeat the loop).
  - The branch is **not taken once** (to exit the loop).

- What Happens with 1-Bit Predictor?
  - First Iteration…….. Next 8 Iteration …….. Last Iteration
  - Calculate the Prediction accuracy (=correct predictions / Total Branches)

Loop Branch behavior: 1-bit Predictor

- 1-bit predictor: Calculate the Prediction accuracy
  - No history yet → default prediction might be **not taken**
  - But it's actually **taken** → ❌ Misprediction
  - Bit is flipped to **1** (taken)
- **Next 8 iterations**:
  - Bit says "taken" → ✅ Correct prediction
- **Last iteration**:
  - Bit says "taken" → ❌ Misprediction (branch is not taken)
  - Bit is flipped to **0**

Loop Branch behavior: 1-bit Predictor

- 1-bit predictor: Calculate the Prediction accuracy

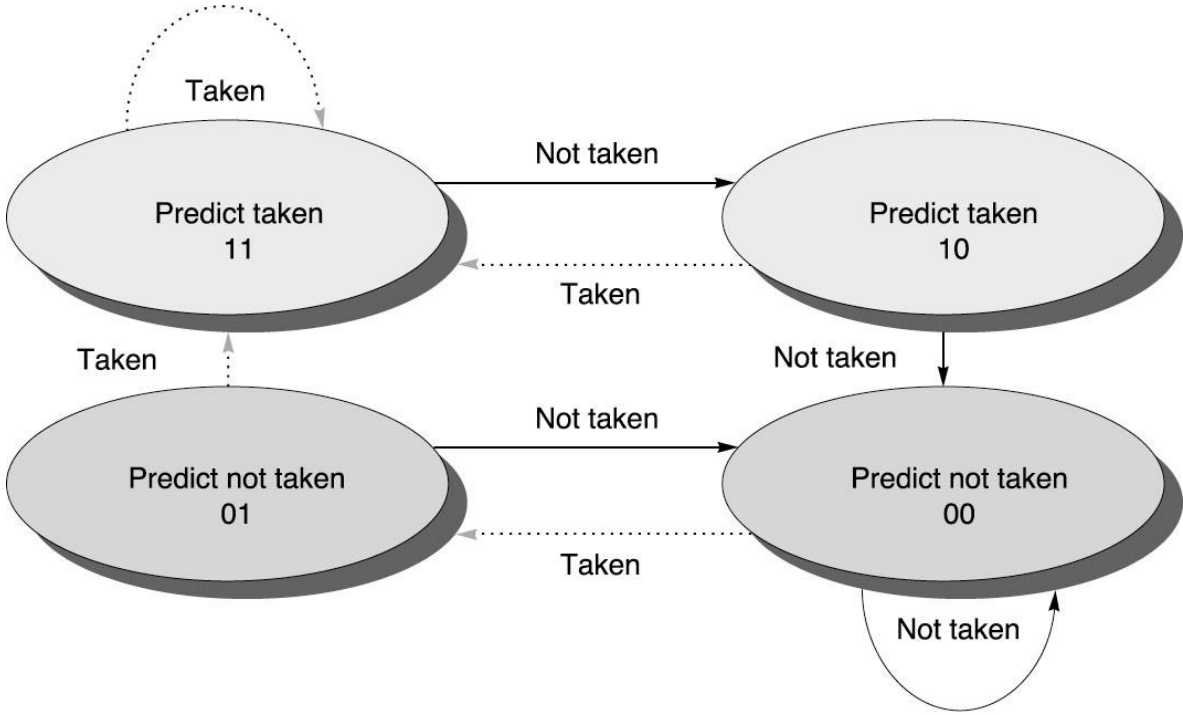
Prediction Accuracy:

- Total branches: 10
- Correct predictions: 8
- Accuracy = 8/10 = 80%

- Even though the branch was **taken 90% of the time**, the predictor only got **80% right**.

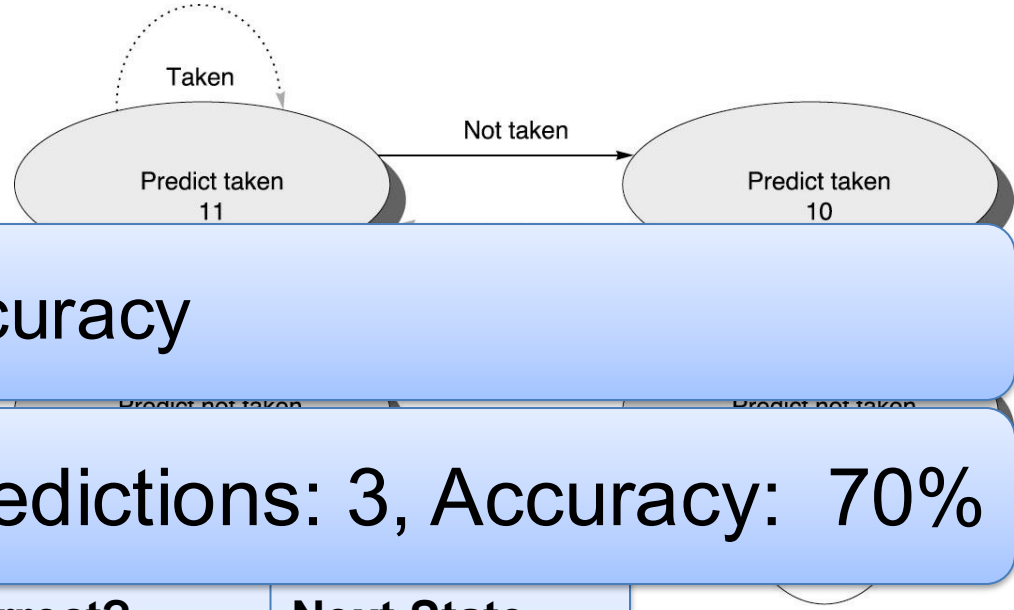
2 mispredictions per loop i.e. Mispredicts at loop boundaries (**first** and **last** iteration)

2-bit Branch Prediction



- Has 4 states instead of 2, allowing for more information about tendencies
- A prediction must miss twice before it is changed
- 2-bit BHT - Also called 2-bit saturating counter
- Can be extended to N-bits (typically N=2)

2-bit Predictor



- 2-bit predictor: Calculate the Prediction accuracy

✅ Correct predictions: 7, ❌ Incorrect predictions: 3, Accuracy: 70%

Iteration	Actual Outcome	Current State	Prediction	Correct?	Next State
1	Taken	00	Not Taken	❌	01
2	Taken	01	Not Taken	❌	11
3	Taken	11	Taken	✅	11
4	Taken	11	Taken	✅	11
5	Taken	11	Taken	✅	11
6	Taken	11	Taken	✅	11
7	Taken	11	Taken	✅	11
8	Taken	11	Taken	✅	11
9	Taken	11	Taken	✅	11
10	Not Taken	11	Taken	❌	10

Alternating Branch Pattern

```
for (int i = 0; i < 20; i++) {
    if (i % 5 == 4) {
        // if statement- body
    } else {
        // Else statement - body
    }
}
```

Calculate the Prediction accuracy for 1-bit and 2-bit, which is better?

Alternating Pattern

```
for (int i = 0; i <
20; i++) {
    if (i % 5 == 4) {}
    else {}
}
```

li $t0, 0	#1
loop_start:	
bge $t0, 20, end	#2
li $t1, 5	#3
rem $t2, $t0, $t1	#4
li $t3, 4	#5
beq $t2, $t3, if_block	**#6**
else_block: # (No instructions here, just a label)	
j loop_continue	#7
if_block: # (No instructions here, just a label)	
loop_continue:	
addi $t0, $t0, 1	#8
j loop_start	#9
end: # (No instructions here, just a label)	

Alternating Branch Pattern

```
for (int i = 0; i < 20; i++) {
    if (i % 5 == 4) {
        // if statement body
    } else {
        // else statement body
    }
}
```

Branch Pattern for PC# 6:

- Iterations 0–3: if **Not Taken**
- Iteration 4: if **Taken**
- Iterations 5–8: : if **Taken**
- Iteration 9: : if **Not Taken**
- ...and so on

Pattern: T, T, T, T, N, T, T, T, T, N, ...

Total Iterations: 20

Taken branches: 16

Not taken branches: 4

Calculate the Prediction accuracy for 1-bit and 2-bit, which is better?

Alternating Branch Pattern (for PC# 6)

Iteration ("i")	Actual	2-bit Predicted	Correct?	State Before	State After
0	N	N	✅	00	00
1	N	N	✅	00	00
2	N	N	✅	00	00
3	N	N	✅	00	00
4	T	N	❌	00	01
5	N	N	✅	01	00
6	N	N	✅	00	00
7	N	N	✅	00	00
8	N	N	✅	00	00
9	T	N	❌	00	01

Assume starting state is 00

✅ Correct Predictions: 16, ❌ Mispredictions: 4

Accuracy: 80%

Alternating Branch Pattern (for PC# 6)

Iter	Actual	Prediction	Correct?	Updated Prediction (1-bit)
0	NT	NT	✅	NT
1	NT	NT	✅	NT
2	NT	NT	✅	NT
3	NT	NT	✅	NT
4	T	NT	❌	T
5	NT	T	❌	NT
6	NT	NT	✅	NT
7	NT	NT	✅	NT
8	NT	NT	✅	NT
9	T	NT	❌	T
10	NT	T	❌	NT
11	NT	NT	✅	NT
12	NT	NT	✅	NT
13	NT	NT	✅	NT
14	T	NT	❌	T
15	NT	T	❌	NT
16	NT	NT	✅	NT
17	NT	NT	✅	NT
18	NT	NT	✅	NT
19	T	NT	❌	T

- Assume starting state is NT (0)
  - ✅ Correct Predictions: 13,
  - ❌ Mispredictions: 7
  - Accuracy: 65%

- Assume starting state is T (1)
  - ✅ Correct Predictions: 12,
  - ❌ Mispredictions: 8
  - Accuracy: 60%

Alternating Branch Pattern

- 1- bit predictor
  - ✅ Correct Predictions: 12 , ❌ Mispredictions: 8
  - Accuracy: 60%

- 2- bit predictor
  - ✅ Correct Predictions: 16, ❌ Mispredictions: 4
  - Accuracy: 80%

Branch Prediction

- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding m co-related branches
- Local predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last *n* occurrences of this branch
- Tournament predictor:
  - Combine correlating predictor with local predictor

Branch Prediction

- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding m co-related branches
- Local predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last *n* occurrences of this branch
- Tournament predictor:
  - Combine correlating predictor with local predictor

Correlating Predictors

- Look at other branches for clues!

```
if (x==2)          -- branch b1
    x=0;
if (y==2)          -- branch b2
    y=0;
if(x!=y) { …   -- branch b3 – Clearly depends on
                     the results of b1 and b2
```

Is the Outcome of the branch “b3” is dependent on the past local-history of the same branch “b3” recorded over the previous iterations?

Branch Prediction

- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding m co-related branches
- Local predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last *n* occurrences of this branch
- Tournament predictor:
  - Combine correlating predictor with local predictor

Branch Prediction

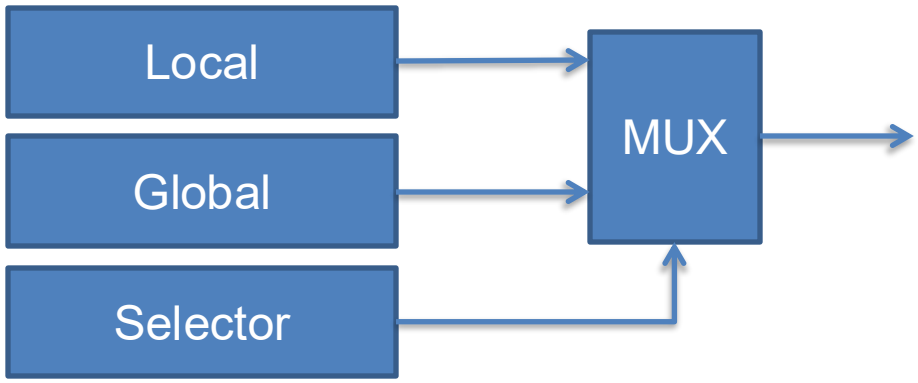
- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding m co-related branches
- Local predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last *n* occurrences of this branch
- Tournament predictor:
  - Combine correlating predictor with local predictor

Branch Prediction

- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding m co-related branches
- Local predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last *n* occurrences of this branch
- Tournament predictor:
  - Combine correlating predictor with local predictor

Tournament Predictors

- Problem: Some branches work well with local predictors, while other branches work well with global predictors
- Solution: Use multiple predictors.
  - One based on global information, one based on local information.
  - Add a selector to pick between predictors for a branch address.
    - Uses a **2-bit saturating counter** to decide which predictor—local or global—has been more accurate recently for a given branch.



Tournament Predictors

1-bit selector would be enough to choose between just two predictors—**local** and **global**.

Why is 2-bit selector used in Tournament Predictors?

- A 1-bit selector flips immediately after a single misprediction. That means:
  - If the currently chosen predictor makes one mistake, the selector switches to the other.
  - This can lead to **oscillation**—rapid flipping between predictors—even when one predictor is generally more accurate.

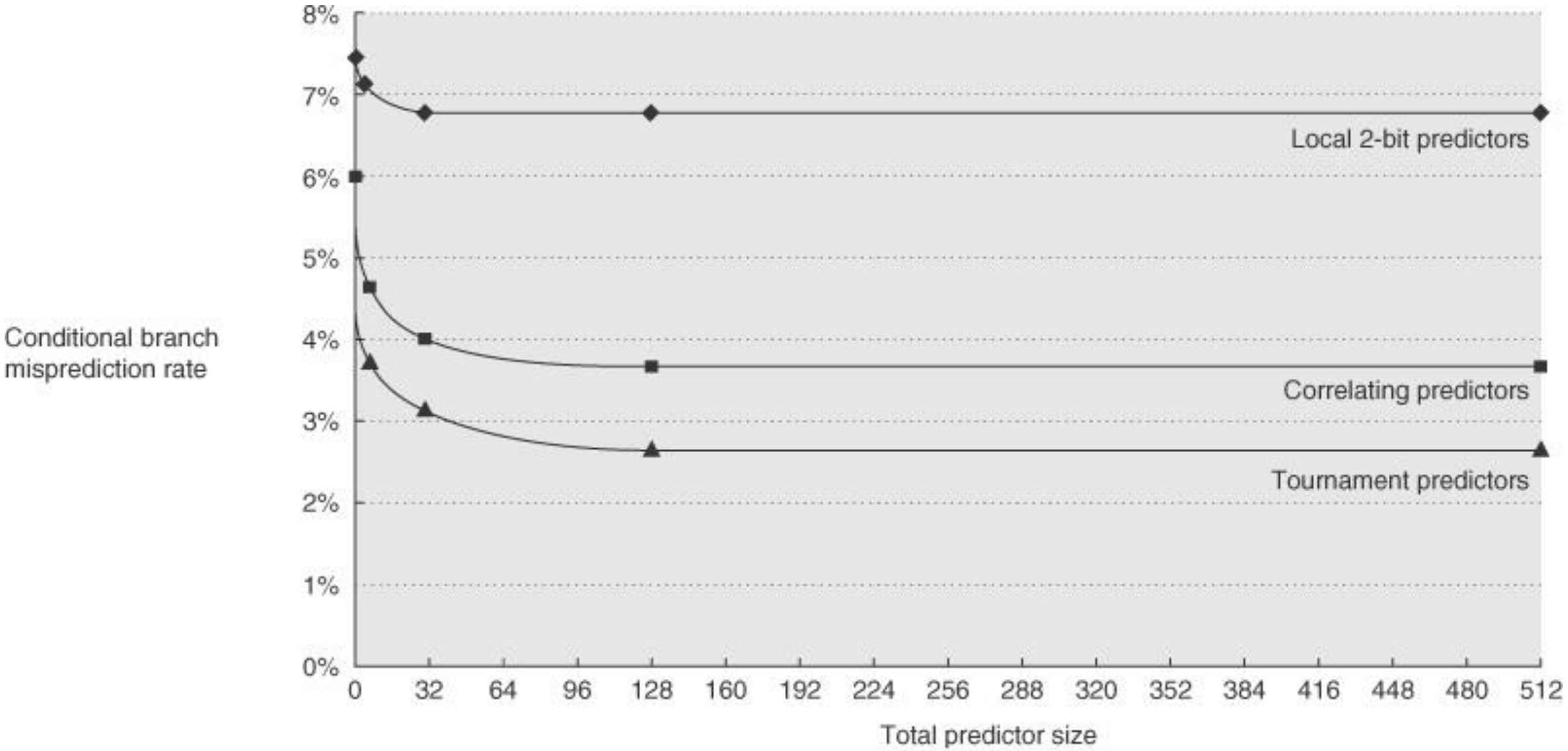
Tournament Predictors

1-bit selector would be enough to choose between just two predictors—**local** and **global**.

Why is 2-bit selector used in Tournament Predictors?

- A **2-bit saturating counter** adds —resistance to change:
  - Stays with the better predictor longer, even if it occasionally mispredicts.
  - Requires two consecutive mispredictions before switching.
  - **More Stable** compared to 1-bit selector

Predictor Accuracy



Review…Dynamic Branch Prediction Steps

- Step 2: Branch Prediction Buffer (BPB) Access.
  - The PC is used to **index into the BPB** (also called Branch History Table or Branch Target Buffer).
  - If the instruction is a **branch**, the BPB provides:
    - **Direction prediction**: Will the branch be **taken or not**?
    - **Target prediction**: If taken, what is the next PC (**target address**)?
- Step 3: Speculatively Fetch next instruction.--- as per BPB's prediction
- Step 4: Validation: Check if prediction is correct or not (mis-prediction).

Branch Target Buffer (BTB)

- The **BTB** is a specialized **cache** that stores:
  - **Branch instruction addresses** (PCs)
  - Their **predicted target addresses** (where to jump if the branch is taken)
- It does **not** store whether a branch is taken or not—that's handled by the **branch predictor**. Instead, the BTB helps with **fetching the next instruction early**.

Is the Current Instruction a Branch?

- During the **Instruction Fetch (IF) stage**, the processor checks the BTB using the current PC.
- If there's a **BTB hit**, it means the instruction at that PC is **likely a branch**.
- This allows the processor to **speculatively fetch** the target instruction **before decoding** the branch.
- This early fetch is crucial for deep pipelines where delays are costly.

What Is the Branch Target Address?

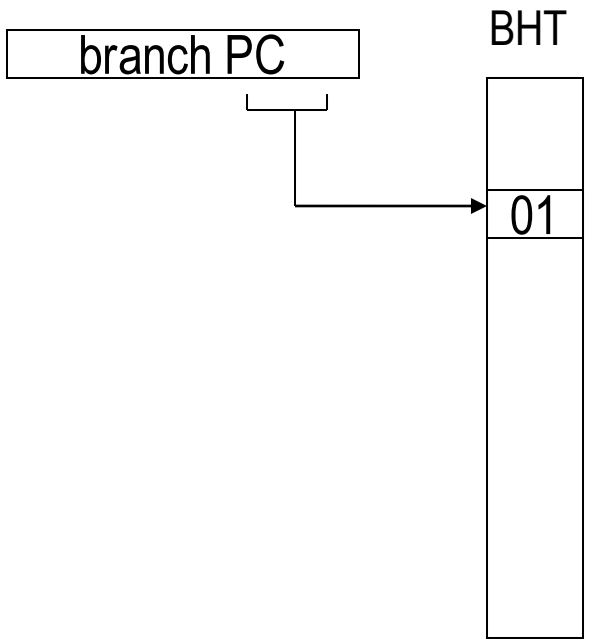
- If the branch is **predicted taken**:
  - The BTB provides the **target address** (e.g., loop start or jump destination).
  - Fetching continues from that target.
- If the branch is **predicted not taken**:
  - The next instruction is simply at **PC + 4** (assuming 4-byte instruction width).

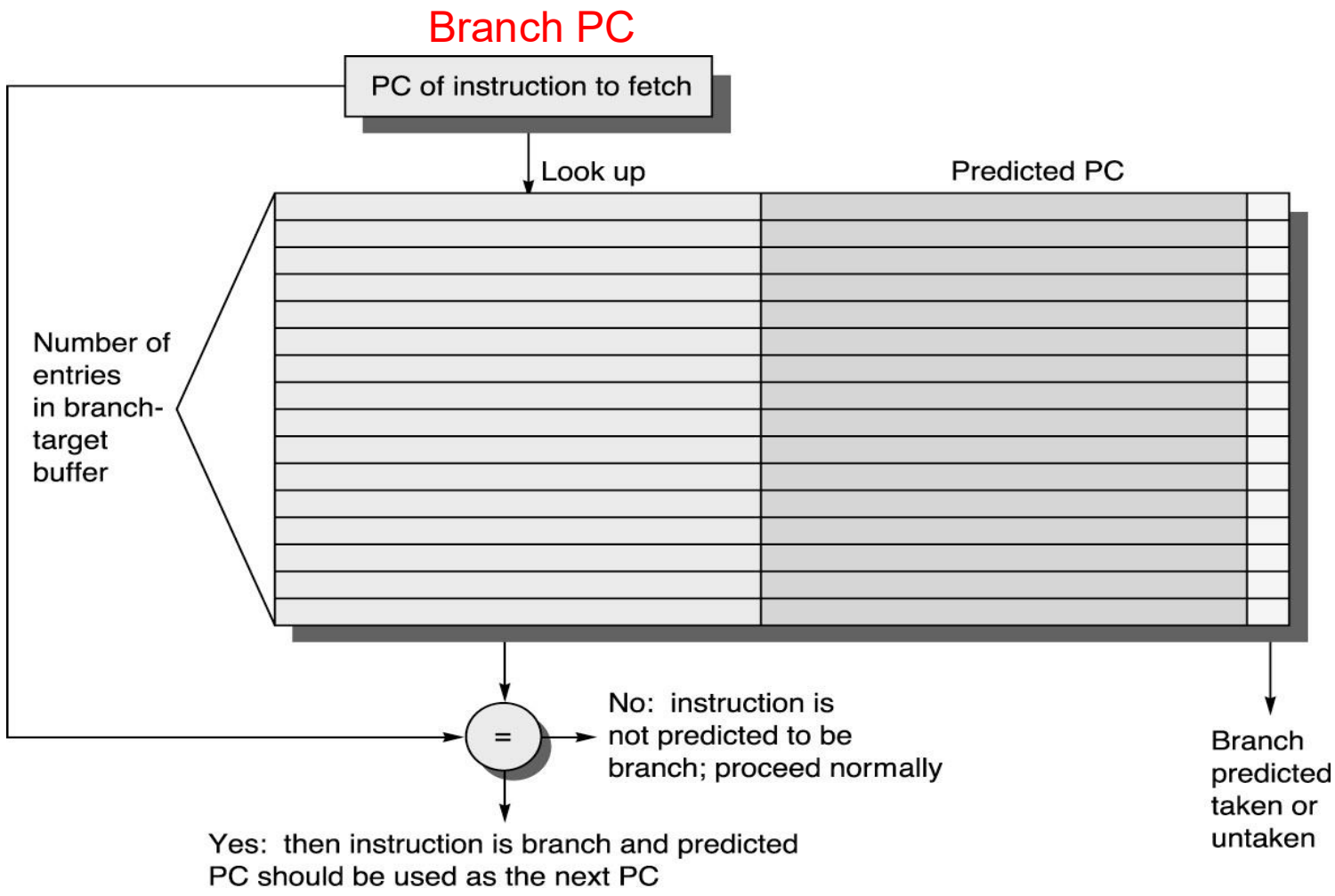
BHT vs BTB

- **BTB** helps the processor know **where** to go if a branch is taken.
- **BHT** helps decide **whether** to take the branch.

- When both work well together, the processor can **speculatively execute** the correct path **without stalling**, improving performance in pipelined architectures.

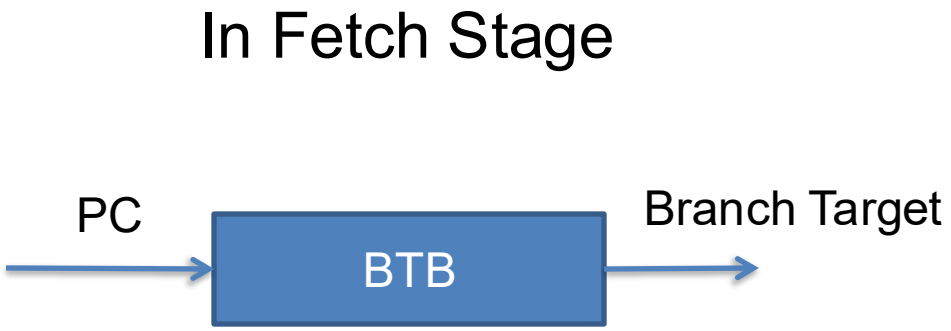
BHT vs BTB


branch PC
BHT
01

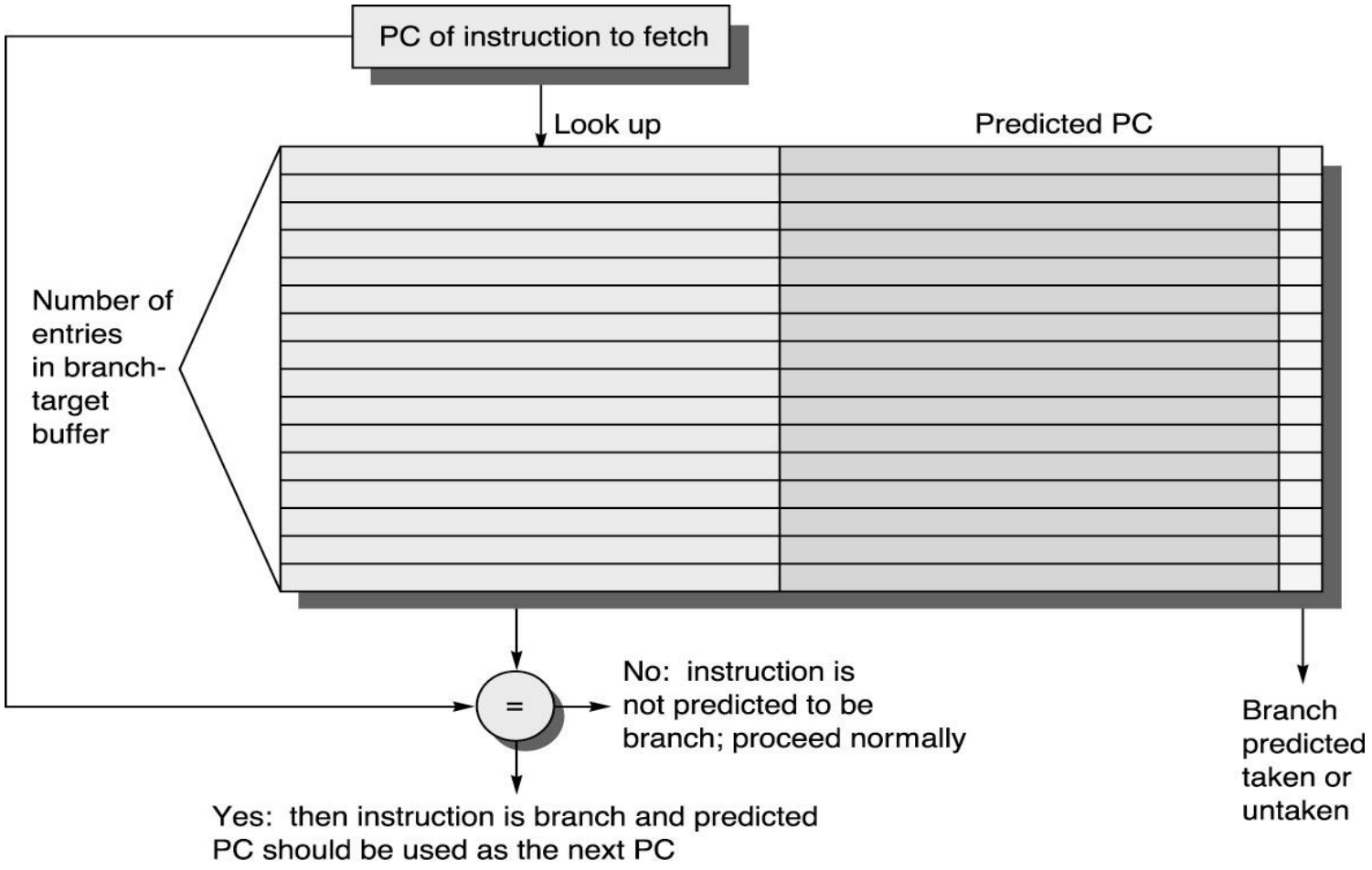

Branch PC
PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Branch predicted taken or untaken
Yes: then instruction is branch and predicted PC should be used as the next PC

Branch Target Buffers (BTB)

- Branch target calculation is costly and stalls instruction fetch
- BTB enable fetching to begin at IF-stage
- BTB cache predicts PC value *(where am I branching to?)*

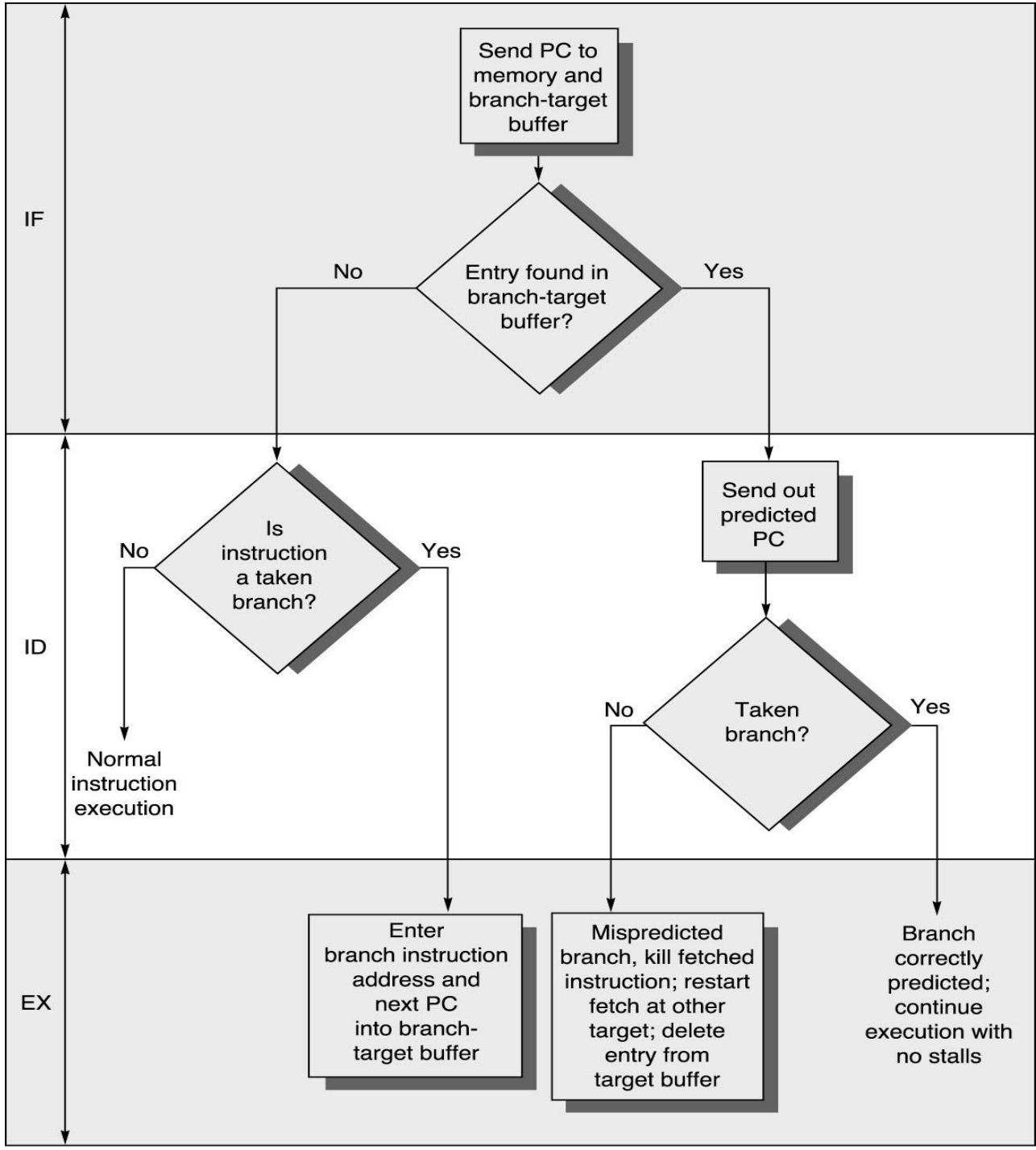


Branch Target Buffers



BTB Algorithm

- BTB hit + predicted taken
  = 0 cycle delay
- BTB hit + misprediction
  = 2 cycle penalty
- BTB miss + misfetch
  = 2 cycle penalty



BTB Algorithm

- **BTB Hit + Predicted Taken = 0 Cycle Delay**
  - The BTB correctly identifies the instruction as a branch and provides the **correct target address**.
  - The branch predictor also predicts **Taken**, so the processor can **immediately fetch** the target instruction in the **IF stage**.
  - **No delay**—this is the ideal case.
- **BTB Hit + Misprediction = 2 Cycle Penalty**
  - The BTB correctly identifies the branch and provides a target.
  - But the **branch predictor was wrong**—it predicted Taken when it should've been Not Taken (or vice versa).
  - The processor has to **flush the pipeline** and **redirect** to the correct path.
  - BTB entry is **corrected** if needed.
  - **2-cycle penalty** due to misprediction recovery.

BTB Algorithm

- **BTB Miss = 2 Cycle Penalty**
  - The BTB **fails to recognize** the instruction as a branch.
  - The processor fetches the next sequential instruction (PC + 4), but later realizes it was a branch.
  - It must **stall**, decode the branch, and **add a new entry** to the BTB.
  - **2-cycle penalty** due to late recognition “as a branch instruction” and redirection.

BTB Performance

- Two things can go wrong
  - BTB miss -> (misfetch)
  - BTB hit -> Fetched wrong target address -> Mispredicted a branch (mispredict)

- Ex. Suppose for branches, BTB hit rate of 85% and predict accuracy of 90%, misfetch penalty of 2 cycles and mispredict penalty of 5 cycles. What is the average branch penalty?
  - 2*(15%) + 5*(85%*10%) = 0.725 cycles

- Branch prediction and BTB can be used together to perform better prediction

Summary

- Branch Prediction Buffer – 1-bit and 2-bit
- Correlating Predictor (Two-level)
  - Incorporates global branch information
- Tournament Predictor
  - Incorporates local branch and global branch info.
  - Selector picks between predictors
- Branch Target Buffers
  - Predicts if instruction is fetch, and branch target address.
  - No more stalls on taken branches!