Course on: “Advanced Computer Architectures”

Branch Prediction Techniques

Prof. Cristina Silvano
Politecnico di Milano
email: cristina.silvano@polimi.it

Branch Prediction

- A method for solving a ***branch hazard*** that guess the outcome for a branch and proceeds speculatively from that assumption rather than waiting to certify the actual branch outcome.

Branch Prediction Techniques

- There are two types of methods to deal with the performance loss due to branch hazards:
  1. **Static Branch Prediction Techniques:**
     The prediction (taken/untaken) for a branch is fixed at ***compile time*** for each branch during the entire execution of the program.
  2. **Dynamic Branch Prediction Techniques:**
     The prediction (taken/untaken) for a branch can change ***at runtime*** during the program execution.
- Static branch prediction can also be used in ***combination*** with dynamic branch prediction.
- In both prediction cases, we need to do not change the processor state and registers until the branch outcome is fully resolved.

Static Branch Prediction Techniques

Static Branch Prediction Techniques

- **Static Branch Prediction** is a simple method that assumes the prediction is fixed at ***compile time*** by using some heuristics or compiler hints – rather than considering the runtime execution behavior.
- **Static Branch Prediction** is typically an effective method when the branch behavior for the target application is ***highly predictable*** at compile time.
- **Note:** From now on, we will consider the RISC-V processor optimized with early evaluation of the branch ***during the ID stage.***

Types of Static Branch Prediction Techniques

1) Branch Always Not Taken (Predicted-Not-Taken)

2) Branch Always Taken (Predicted-Taken)

3) Backward Taken Forward Not Taken (BTFNT)

4) Profile-Driven Prediction

5) Delayed Branch

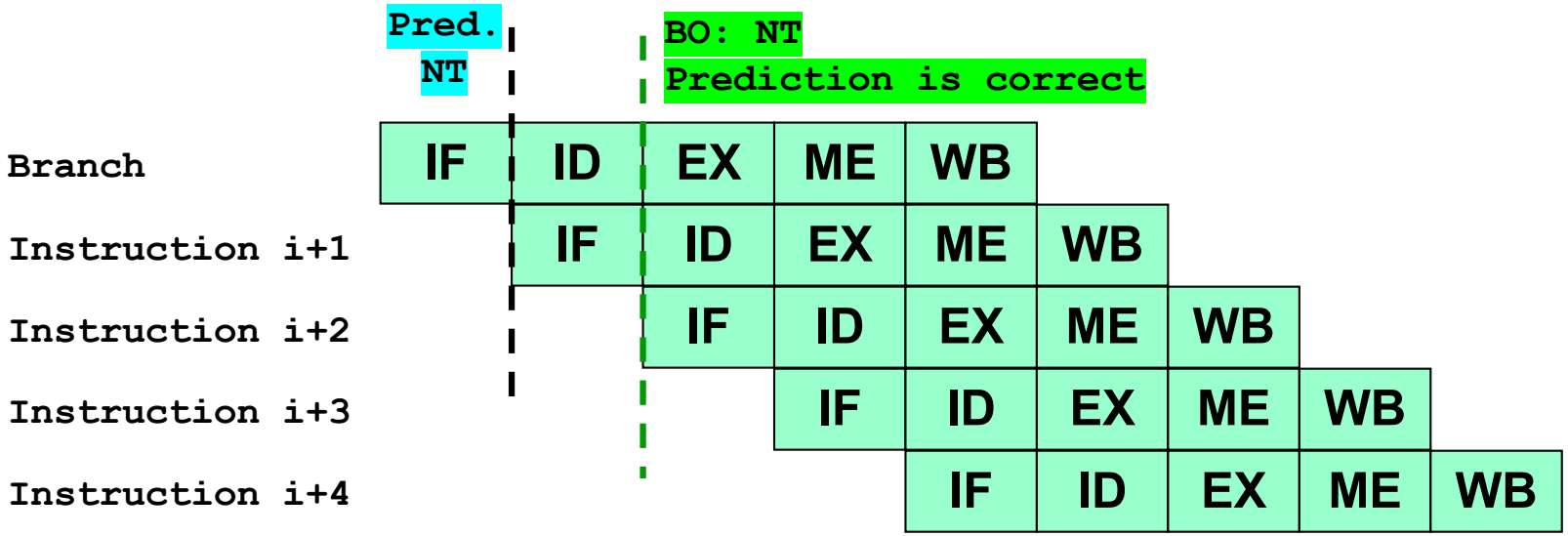
1) Branch Always Not Taken

- **Easiest prediction:** We assume the **branch will be always not taken,** thus the instruction flow can continue sequentially as if the branch condition was **not** satisfied.
- Suitable for **IF-THEN-ELSE** conditional statements, when the **THEN** clause is the most probable and the program will continue sequentially.

```
z = x + y ;
i f ( z > 0)
    w = x ;
e l s e
    w = y ;
```

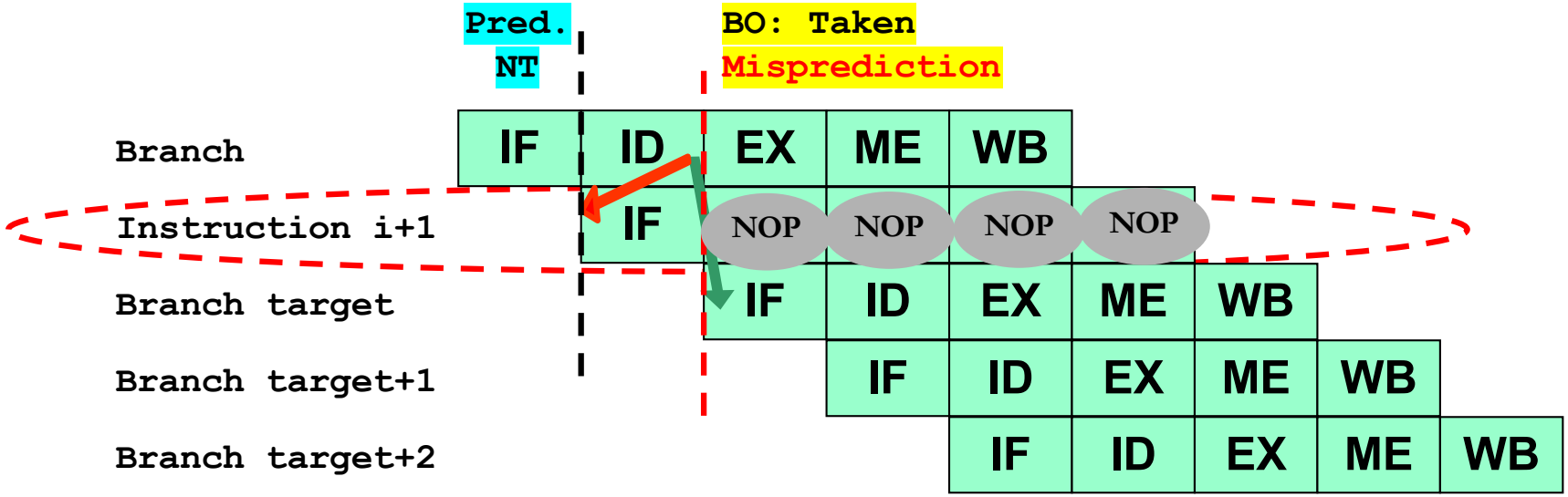
1) Branch Always Not Taken

- Predict the ***branch not taken*** at the end of the **IF** stage.
- Simple to implement (no need to know the Branch Target Address).
- If the Branch Outcome at the end of **ID** stage will be ***not taken*** ⇒ **the prediction was correct ⇒ no branch penalty cycles.**



1) Branch Always Not Taken

- Predict the ***branch not taken*** at the end of the IF stage.
- If the BO at the end of ID stage will be ***taken*** ⇒ **the prediction was *incorrect (misprediction):***
  - Need to **flush** the instruction already fetched (it is turned into a **nop**);
  - Need to fetch the instruction at the Branch Target Address
    ⇒ **One-cycle branch penalty**

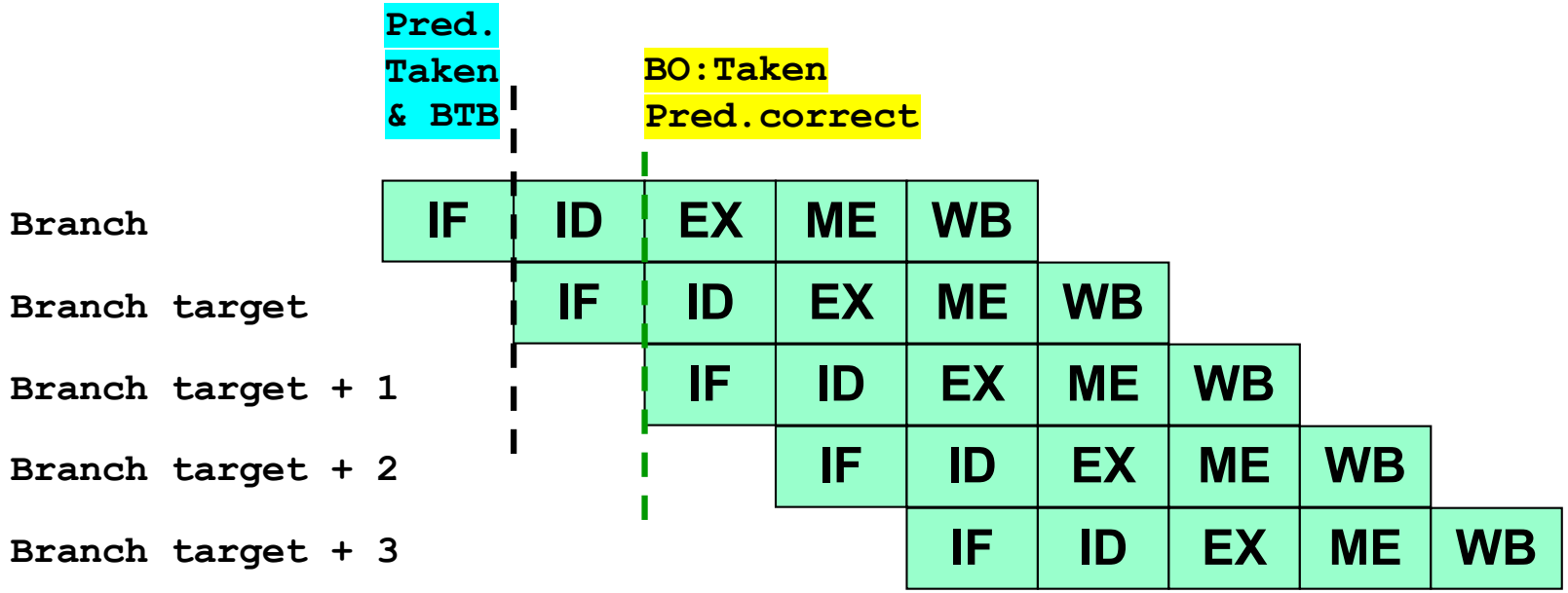


2) Branch Always Taken

- Dual case: We assume every branch as **taken.**
- Suitable for ***backward branches*** such as **`for`** loops and **`do-while`** loops, that are most likely taken.
- **Problem:** we need to know the **Branch Target Address (BTA)** to start fetching and executing instructions at the target
  => In the **IF** stage we need **to add a Branch Target Buffer** where to store the ***Predicted Target Address*** based on the previous branch behavior.

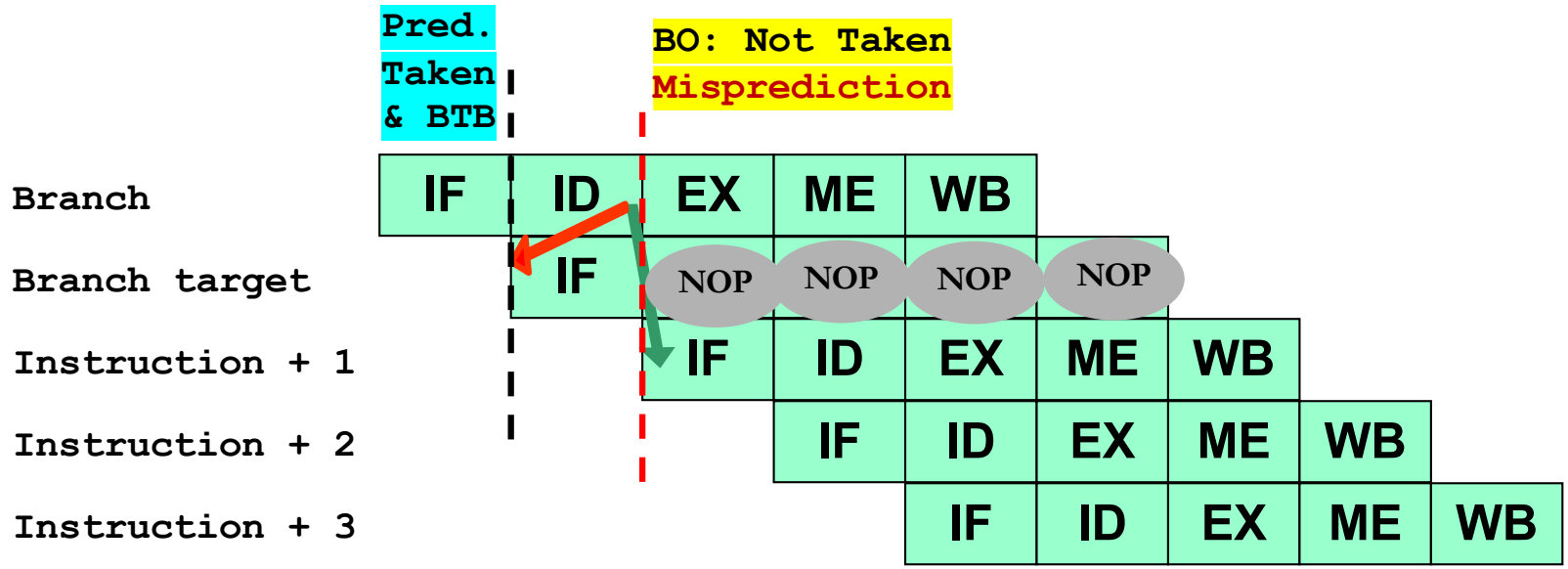
2) Branch Always Taken

- Prediction **taken** at the end of the IF stage.
- If the Branch Outcome at the end of ID stage will be ***taken*** ⇒ **the prediction was *correct* ⇒ no branch penalty cycles.**

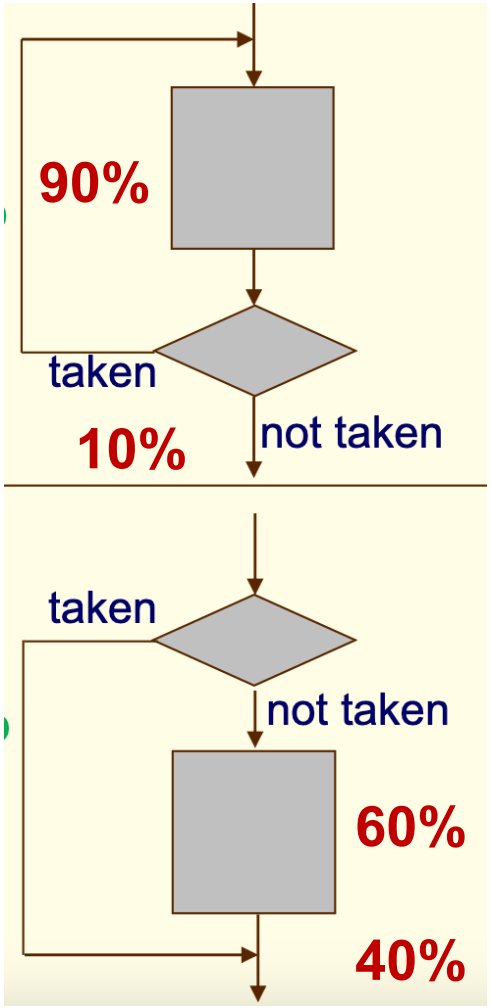


2) Branch Always Taken

- Prediction ***taken*** at the end of the IF stage.
- If the BO at the end of ID stage will be ***not taken*** ⇒ ***misprediction:***
  - Need to **flush** the instruction already fetched (it is turned into a **nop**);
  - Need to fetch the next instruction ⇒ **One-cycle performance penalty**



Taken vs Not Taken Branch Probability: An example



- Backward-going branches are mostly ***taken.***
  - Example: branches at the end of DO-WHILE loops going back at the beginning of the next iteration.

- Forward-going branches are mostly ***not taken.***
  - Example: branches of IF-THEN-ELSE conditional statements when the conditions associated to the ELSE label as less probable.

3) Backward Taken Forward Not Taken (BTFNT)

- The prediction is **based on the branch direction:**
  - Backward-going branches are predicted as ***taken.***
    - Example: loop branches going back at next iteration.
  - Forward-going branches are predicted as ***not taken.***
    - Example: IF-THEN-ELSE branches with ELSE as less probable.
- Used in early processors (e.g., Intel 486, some ARM architectures).

4) Profile-Driven Prediction

- We assume to ***profile*** the behavior of the target application program by several early execution runs by using different data sets.
- The prediction is based on ***profiling information*** collected during earlier runs about the branch behavior. For example:

T T T T T T T T NT NT NT $\Rightarrow$ Taken is the most probable branch outcome

- The profile-driven prediction method requires a ***compiler hint bit*** encoded in the branch instruction format by the compiler:
  - Set to **1** if **Taken** is the most probable branch outcome;
  - Set to **0** if **Not Taken** is the most probable branch outcome;

5) Delayed Branch Technique

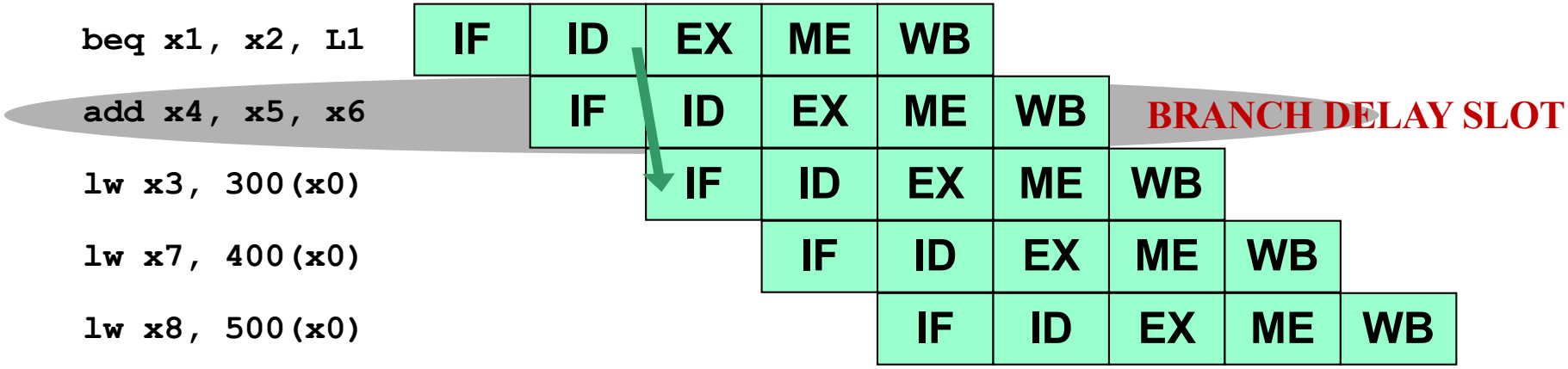
- Scheduling technique: The compiler statically schedules an **independent** instruction in the **Branch Delay Slot.**

- If we assume a branch delay of one-cycle (as for MIPS)
  $\Rightarrow$ we have ***only one-delay slot*** to fill in.

- For deeply pipeline processors, the branch delay slot is typically longer than one cycle: the compiler needs to identify more instructions to fill in.

5) Delayed Branch Technique

- Let's consider the MIPS compiler schedules a **branch independent instruction** after the branch.
- Example: A previous **add** instruction with no effects on the branch condition is scheduled in the **Branch Delay Slot** and it is **always executed,** whether or not the branch will be taken .



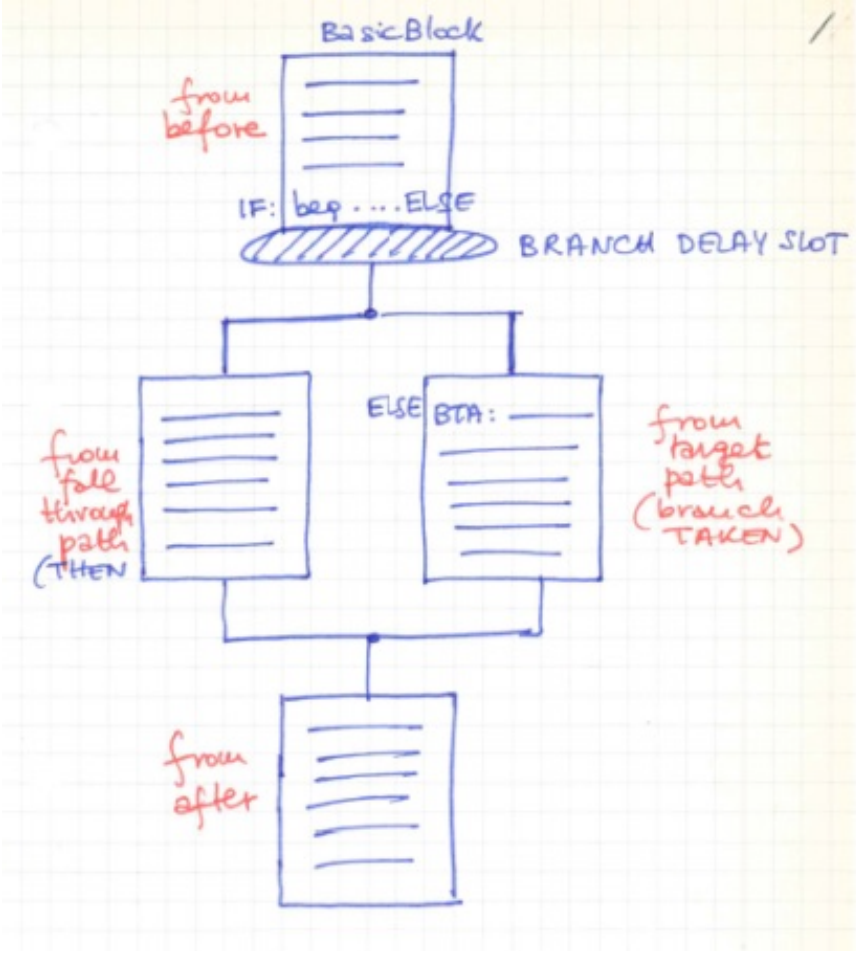
- **Be careful:** The instruction in the slot must be fine to be executed also when the branch goes in the unexpected direction.

5) Delayed Branch Technique

- The job of the compiler is to find a valid and useful instruction to be scheduled in the **branch delay slot.**

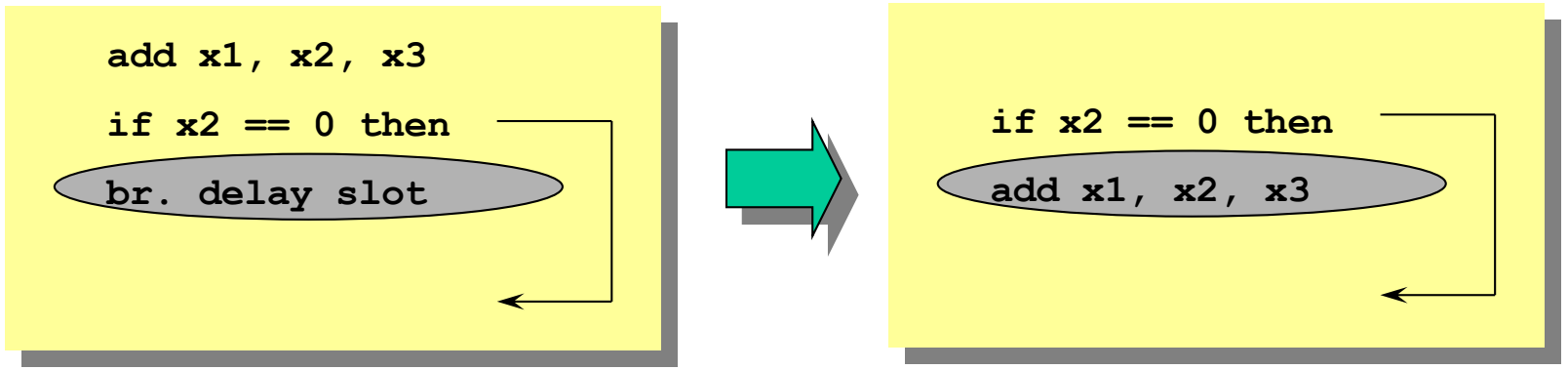
- There are **four ways** to schedule an instruction in the branch delay slot:
  1. **From before**
  2. **From target**
  3. **From fall-through**
  4. **From after**

5) Delayed Branch Technique: IF-THEN-ELSE



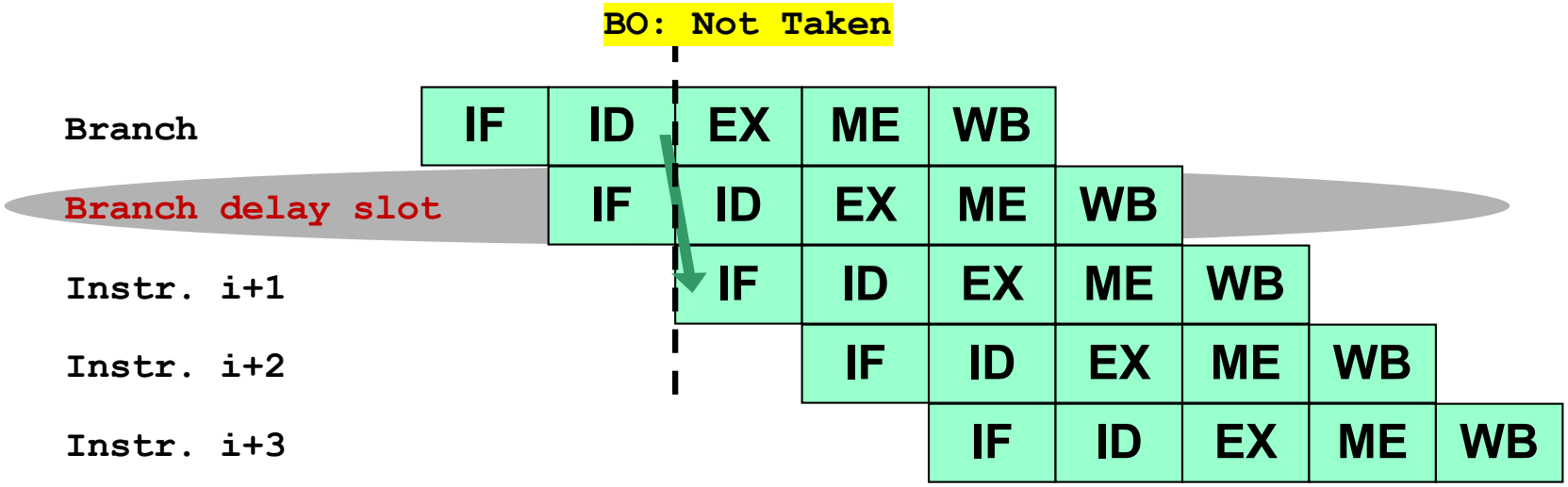
5) Delayed Branch Technique: From Before

- The branch delay slot is scheduled with an **independent** instruction **from before** the branch.
- The instruction in the branch delay slot is **always executed,** whether the branch will be taken or not taken.
- Then execution will continue based on the Branch Outcome in the right direction and the **`add`** instruction in the delay slot will never be flushed.



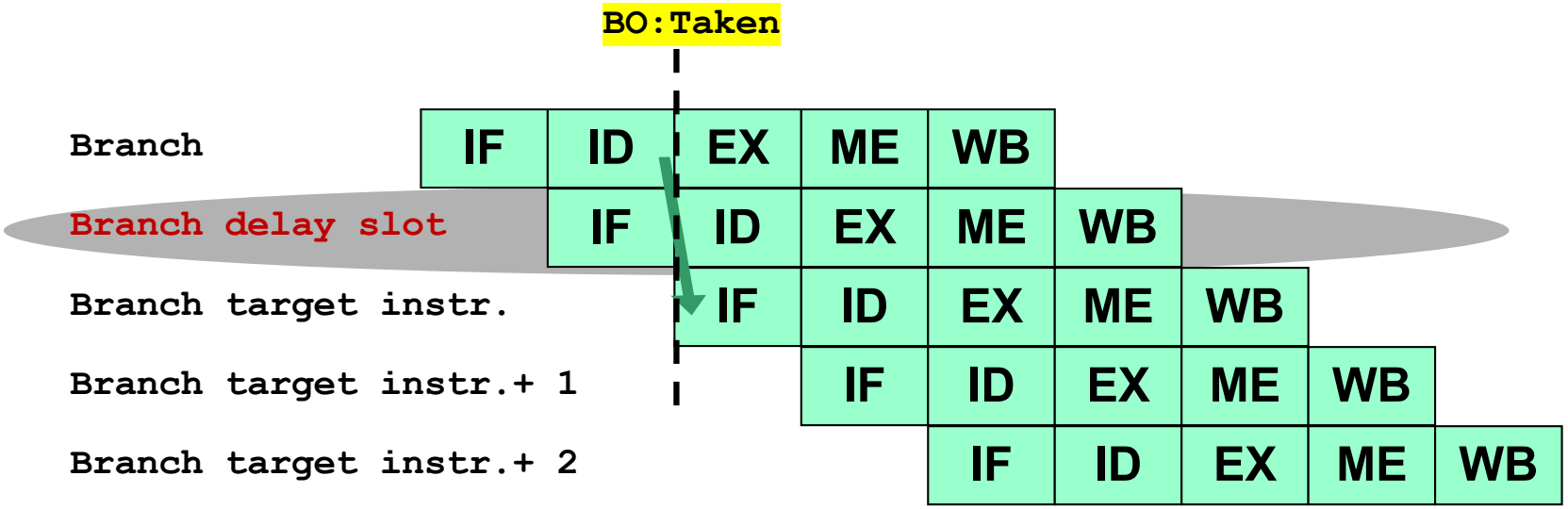
5) Delayed Branch Technique: From Before

- If we assume the MIPS compiler schedules a **branch independent instruction** in the branch delay slot such as it can be always executed, whether or not the branch is taken *(it will never be flushed!)*
  - If the branch is ***not taken*** ⇒ execution continues with the instruction after the branch *(branch delay slot instruction is NOT flushed!)*



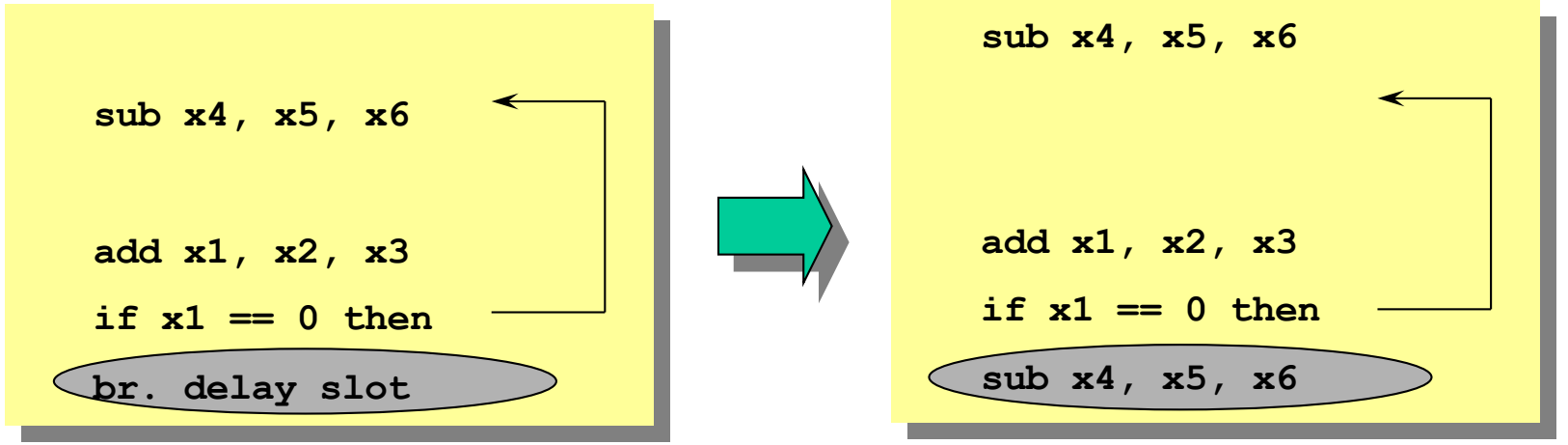
5) Delayed Branch Technique: From Before

- If the branch is ***taken*** ⇒ execution continues at the branch target *(the branch delay slot instruction is NOT flushed!)*



5) Delayed Branch Technique: From Target

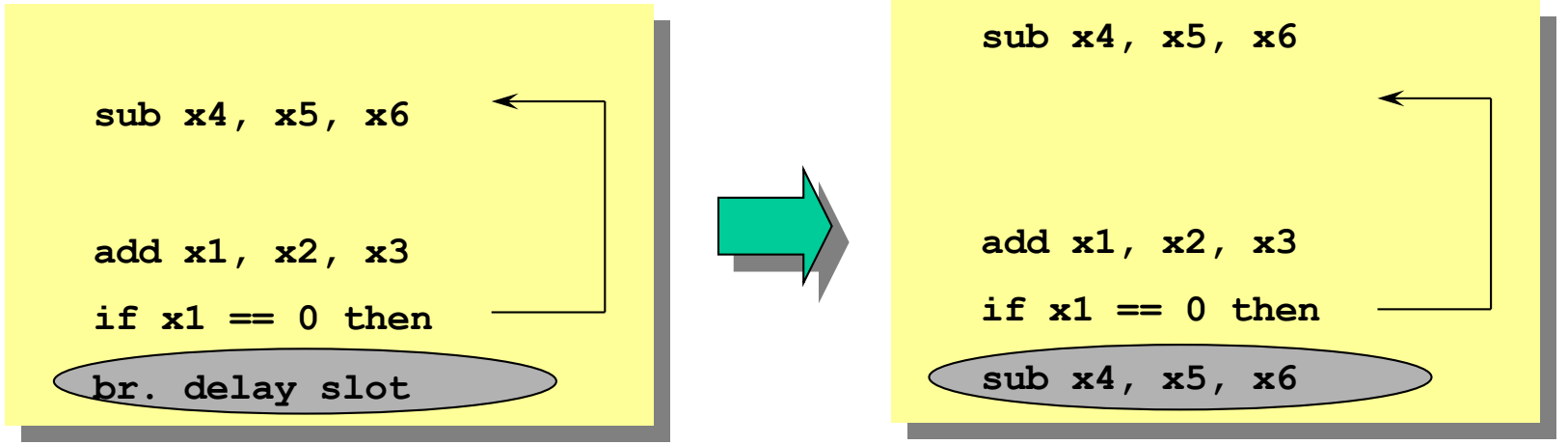
- In this example, the use of **x1** in the branch condition prevents the **`add`** instruction (whose destination is now **x1**) from being moved after the branch.
- The branch delay slot is scheduled with one instruction **from the target** of the branch (branch **taken).**
- **Drawback:** Usually, the target instruction **`sub`** needs to be copied, whenever it can be reached by another path.



5) Delayed Branch Technique: From Target

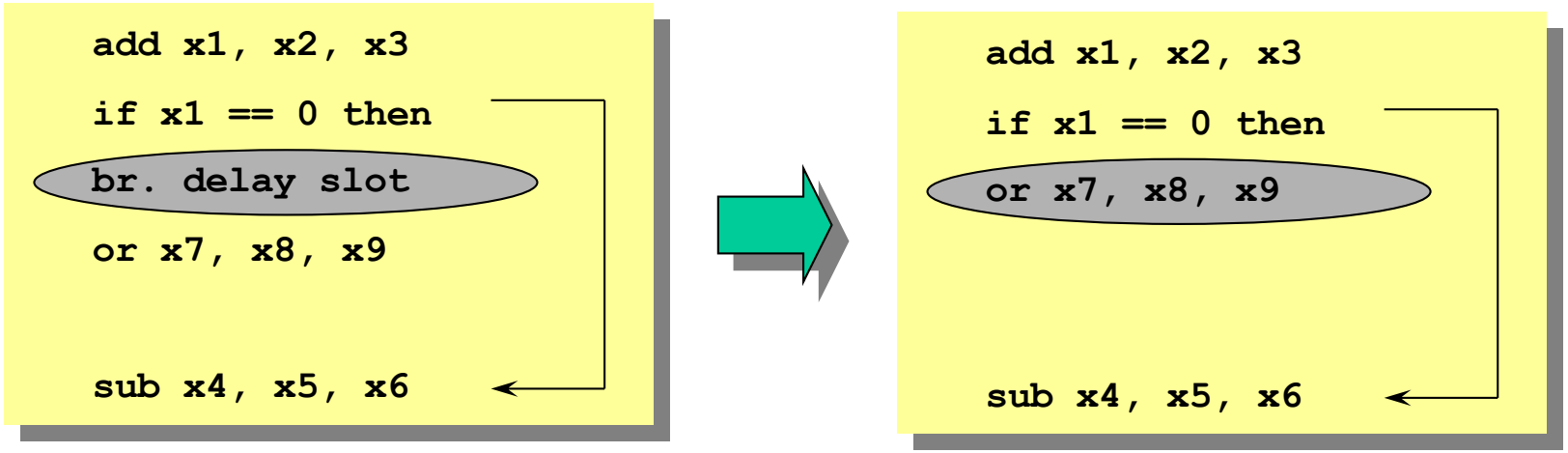
- This strategy is preferred when the branch is **taken** with high probability, such as DO-WHILE loop branches **(backward branches).**

- If the branch is ***untaken (misprediction),*** the **sub** instruction in the delay slot needs to be ***flushed*** or it must be OK to be executed also when the branch goes in the unexpected direction.



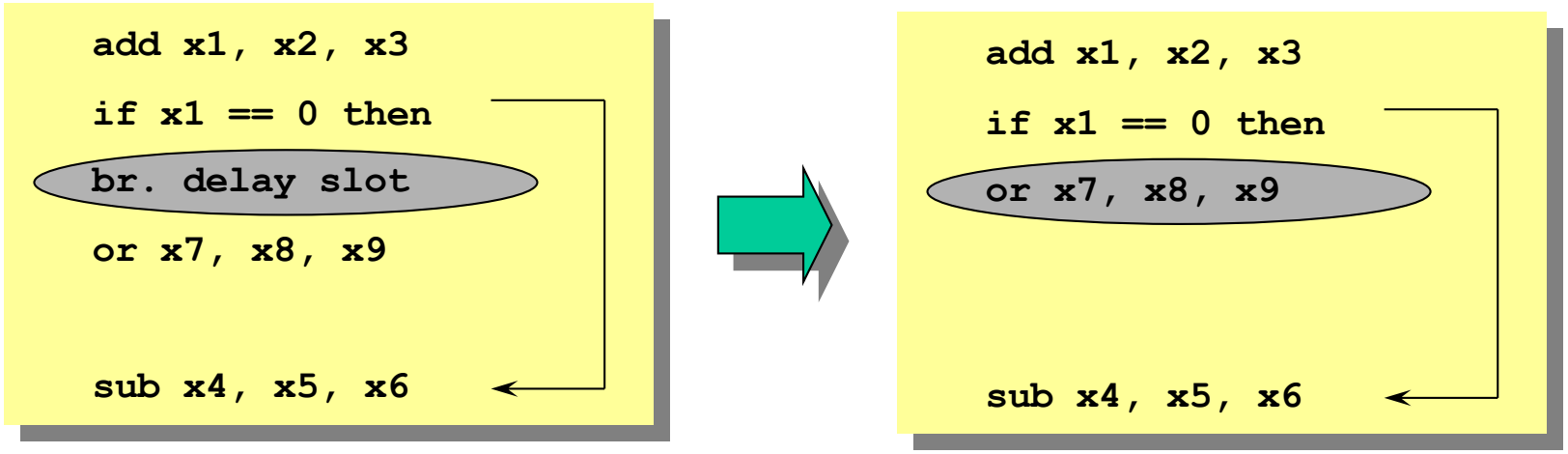
5) Delayed Branch Technique: From Fall-Through

- In this example, the use of **x1** in the branch condition prevents **add** instruction (whose destination is **x1**) from being moved after the branch.
- The branch delay slot is scheduled with one instruction **from the fall-through path** (branch **not taken).**



5) Delayed Branch Technique: From Fall-Through

- This strategy is preferred when the branch is **not taken** with high probability, such as **forwarding branches:** IF-THEN-ELSE statement where the ELSE path is less probable.
- If the branch is **taken (misprediction),** the **or** instruction in the delay slot needs to be ***flushed or*** it must be OK to be executed also when the branch goes in the unexpected direction.



5) Delayed Branch Technique

- To make the optimization legal for the **target** and **fall-through cases,** the instruction in the branch delay slot must be either flushed or it must be OK to be executed also when the branch goes in the unexpected direction.
- By OK we mean that the instruction in the branch delay slot is executed but the work is *wasted* (and the program still execute correctly).
- For example, if the destination register of the instruction in the delay slot is a temporary register unused when the branch goes in the unexpected direction.

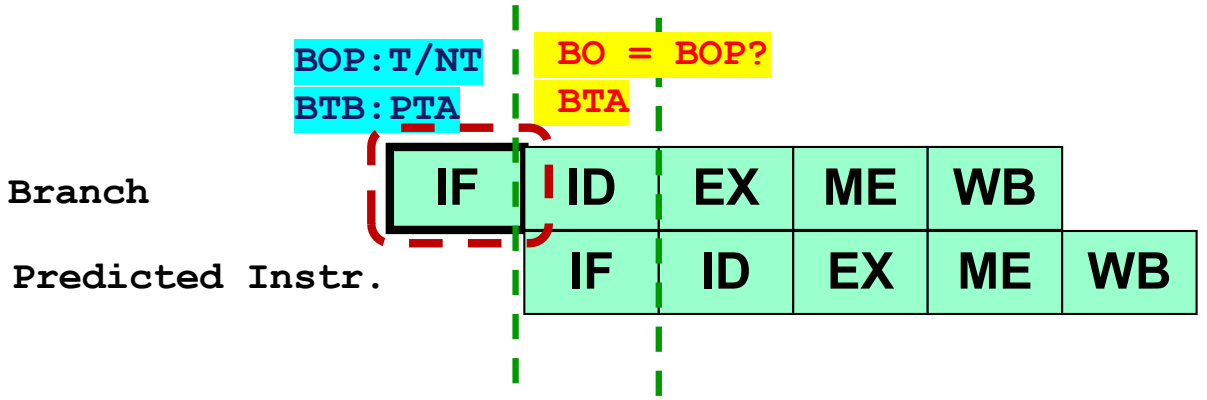
Dynamic Branch Prediction Techniques

Dynamic Branch Prediction

- **Basic Idea:** To use the past branch behavior to predict **at runtime** the future branch behavior.
  - We use the hardware to **dynamically** predict the outcome of a branch.
  - The prediction will depend on the runtime behavior of the branch.
  - The prediction will change at runtime if the branch changes its behavior during execution.

Dynamic Branch Prediction Schemes

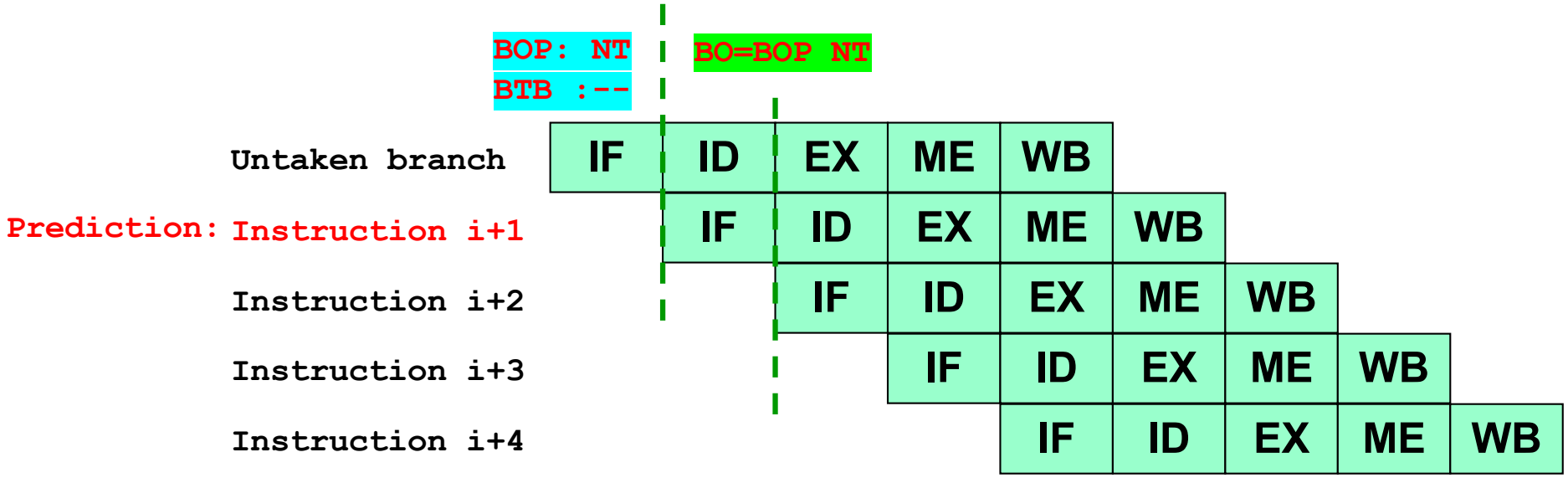
- Dynamic branch prediction is based on **two** interacting hardware blocks:
  1. **Branch Outcome Predictor (BOP) or Branch Prediction Buffer**
     - To predict the direction of a branch (Taken or Not Taken).
  2. **Branch Target Predictor or Branch Target Buffer (BTB):**
     - To predict the branch target address in case of taken branch (Predicted Target Address - PTA)
- **They are placed in the Instruction Fetch stage** to predict the next instruction to read in the Instruction Cache:



What happens when the prediction is Not Taken (1)

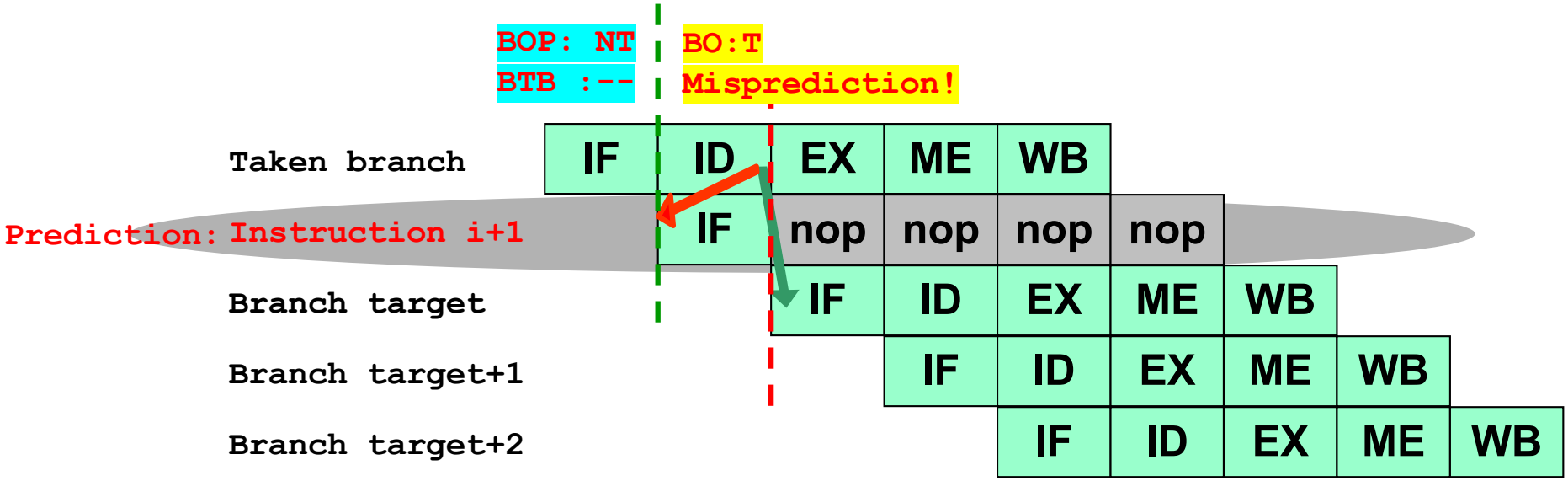
- If branch is predicted by BOP in IF stage as **not taken** ⇒ PC is incremented (BTB not useful in this case):

  1. If the branch outcome at the end of ID stage will result as **not taken** ⇒ **the prediction is correct** ⇒ **no branch penalty cycles.**



What happens when the prediction is Not Taken (2)

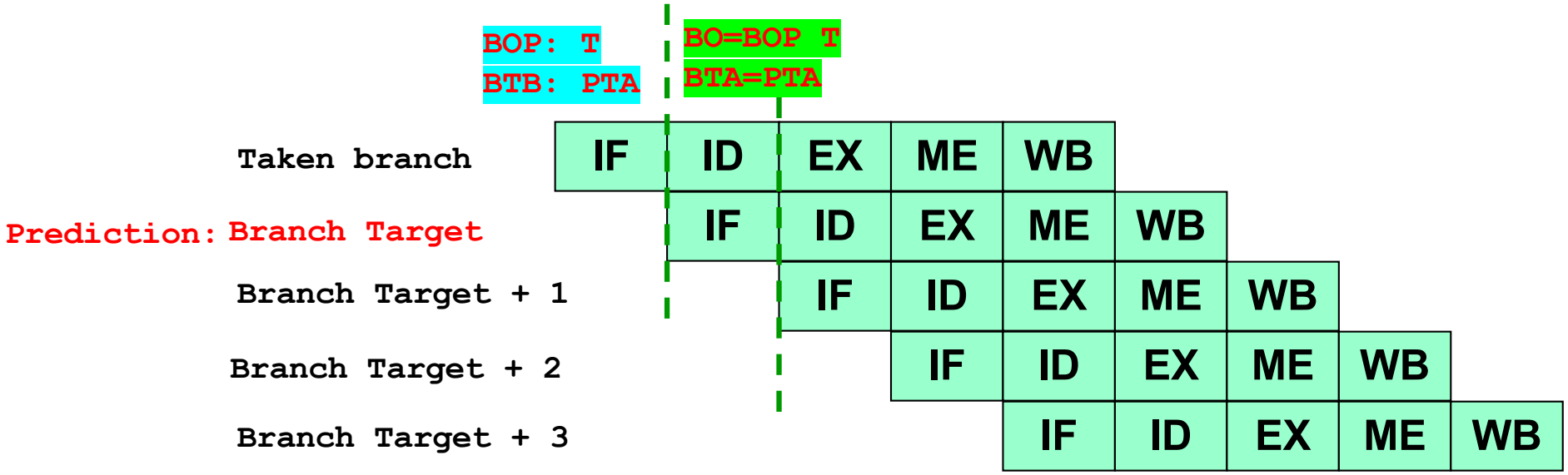
2. If the branch outcome will result **taken** ⇒ **misprediction!**
   - Need to **flush** the next instruction already fetched (the next instruction is turned into a **nop**).
   - Need to restart the execution at the Branch Target Address ⇒ **One-cycle penalty**



What happens when the prediction is Taken (1)

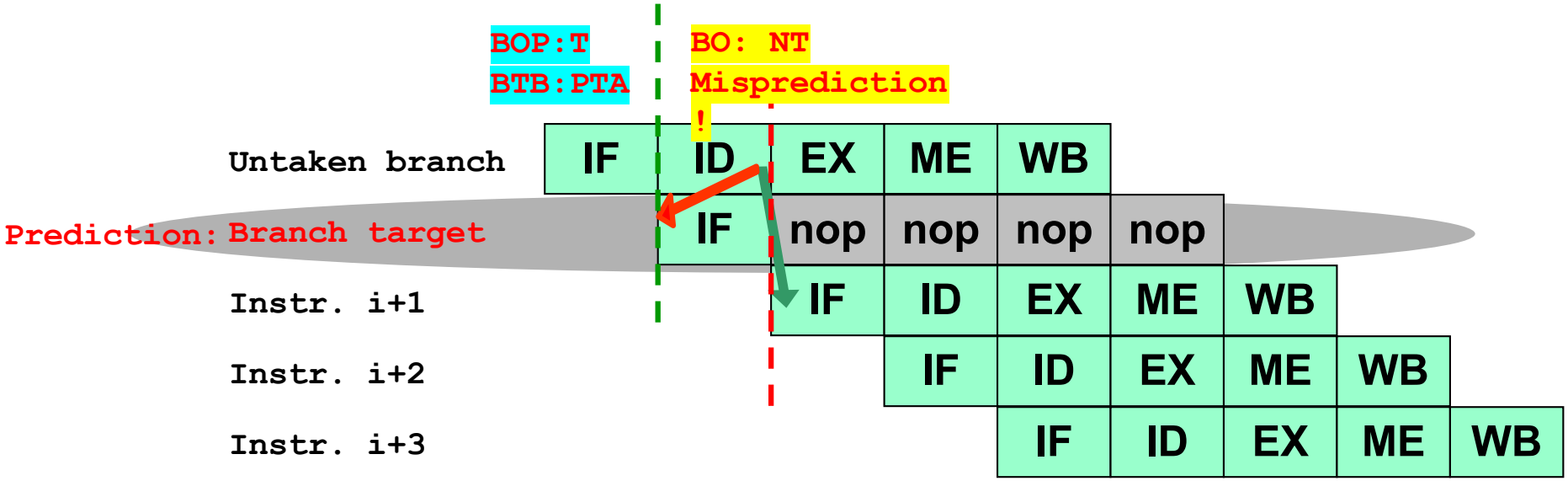
- If branch is predicted by BOP in IF stage as **taken** and BTB gives the Predicted Target Address (PTA):

  1. If the BO at the end of ID stage will result as **taken** ⇒ **the prediction is correct** ⇒ **no branch penalty cycles.**



What happens when the prediction is Taken (2)

2. If the BO at the end of ID stage will result **untaken** ⇒ **misprediction!**
   - Need to **flush** the branch target instruction already fetched (this is turned into a **nop**)
   - Need to restart the execution by fetching at the next instruction ⇒ **One-cycle penalty**



Dynamic Branch Prediction Techniques

1) Branch History Table

2) Correlating Branch Predictors

3) Two-level Adaptive Branch Predictors

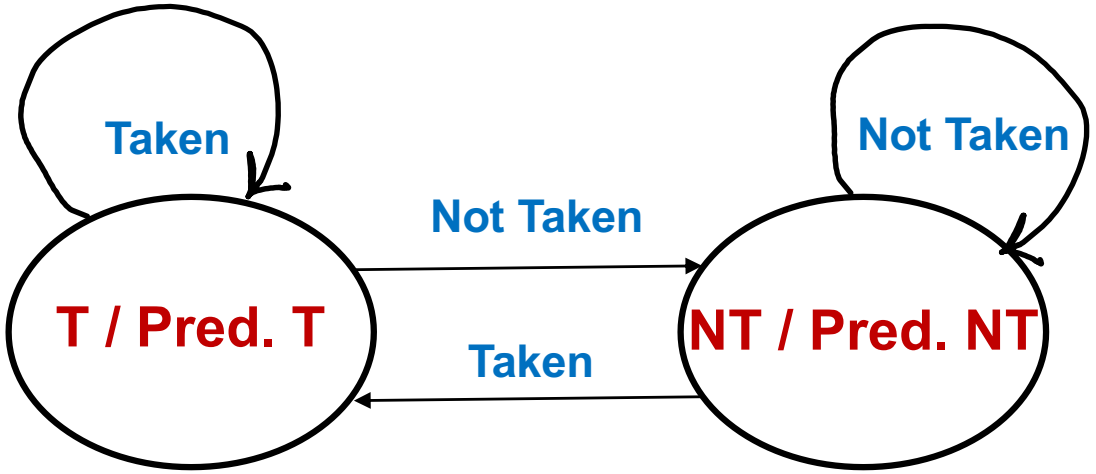
4) Branch Target Buffer

1) Branch History Table

- **Branch History Table (or Branch Prediction Buffer):**
  - Table containing **1 bit** for each branch that says whether the branch was recently **Taken** or **Not Taken.**
  - The behavior is controlled by a **Finite State Machine** with only 1-bit of history (2 states) to remember the last direction taken by the branch to predict the next branch outcome (T/NT).

Behavior of 1-bit Branch History Table

- **Finite State Machine** with only 1-bit of history (2 states) to remember the last direction taken by the branch:
  - If the prediction is correct ⇒ remains in the current status (and branch prediction outcome);
  - If the prediction is not correct ⇒ change status (and branch prediction outcome)

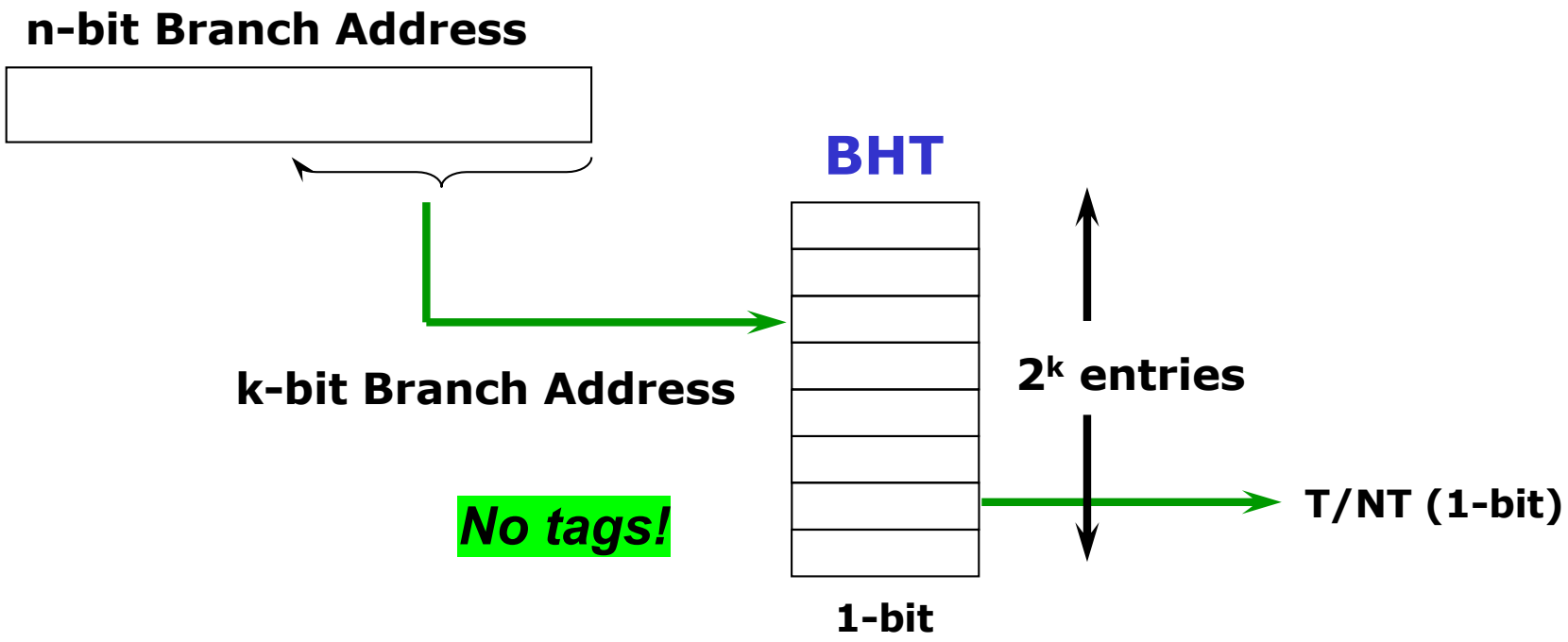


Branch History Table

- **Branch History Table (or Branch Prediction Buffer):**
  - Table containing **1 bit** for each entry that says whether the branch was recently **Taken** or **Not Taken.**
  - Table indexed by the lower portion k-bit of the address of the branch instruction (to keep the size of the table limited)
  - For locality reasons, we would expect that the most significant bits of the branch address are **not** changed

- The table has **no tags** (every access is a hit) and the prediction bit may have been put there by another branch with the same low-order address bits: but it doesn't matter. *The prediction is just a hint!*

1-bit Branch History Table



Accuracy of 1-bit Branch History Table

- A **misprediction** occurs when:
  - The prediction is incorrect for that branch

  or

  - The same index has been referenced by two different branches, and the previous history refers to the other branch (This can occur because there is no tag check!)
    - To reduce this problem, it is enough to increase the number of rows in the BHT (that is to increase k) or to use a hashing function (such as in GShare).

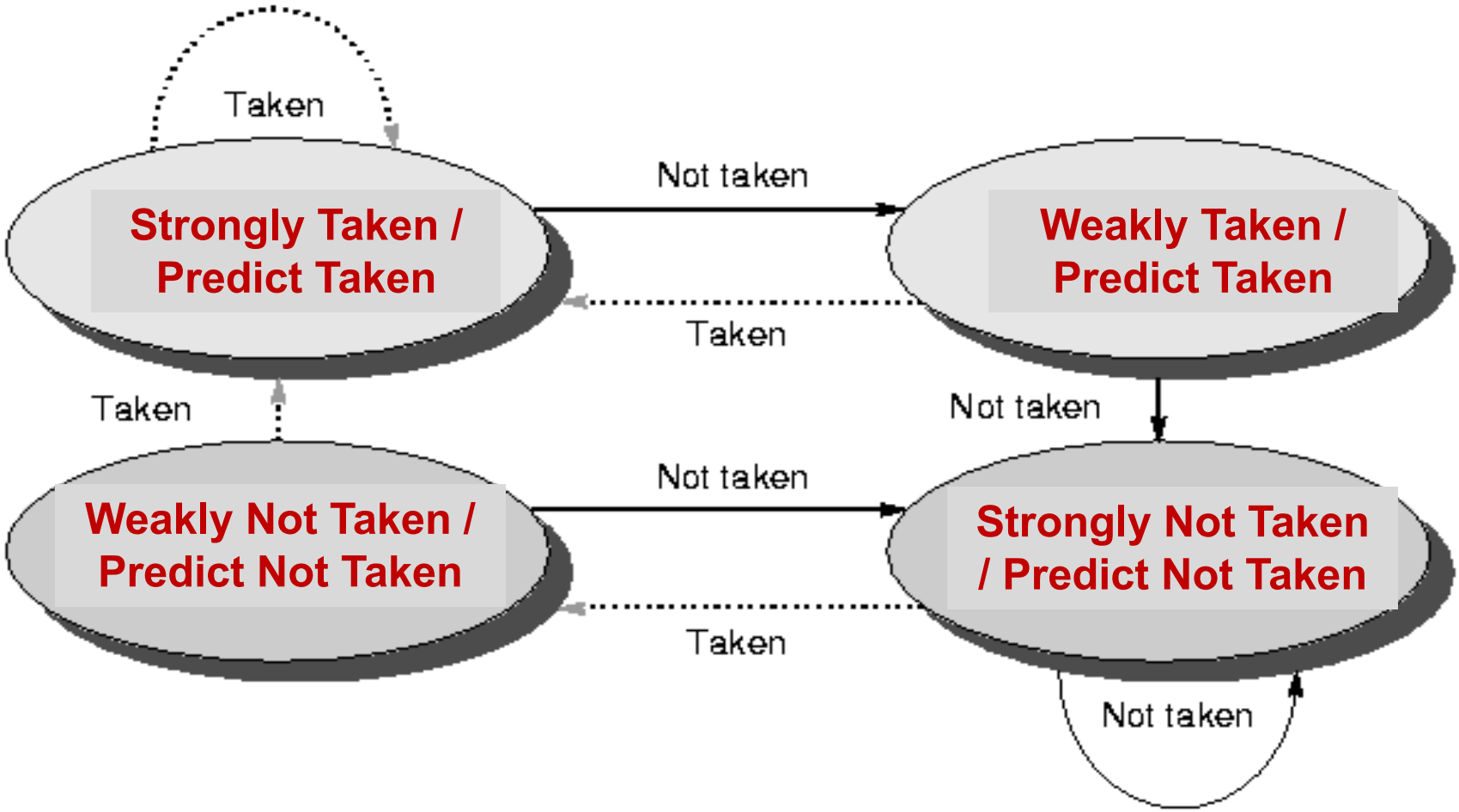
Shortcoming of 1-bit Branch History Table

- **In a loop branch,** a branch is almost always T and then NT once at the exit of the loop.
- Problem: the 1-bit BHT causes 2 mispredictions:
  - **At the last loop iteration,** since the prediction bit is T, while we need to exit from the loop.
  - When we re-enter the loop, **at the first iteration** we need to take the branch to stay in the loop, while the prediction bit was flipped to NT on previous execution of the last iteration of the loop.
- **Example:** In a loop branch of 10 iterations whose behavior is Taken nine times and Not Taken once => the prediction accuracy is only 80% (due to 2 mispredictions at the first and last iterations and 8 correct predictions out of 10 iterations).

2-bit Branch History Table

- **Solution:** use 2-bit BHT used to encode 4 states in order to change the prediction after 2 mispredictions.
  - *The prediction must mispredict twice before it is changed!*
  - *Otherwise, the prediction stays the same.*

Behavior of 2-bit Branch History Table



2-bit Branch History Table

- **Solution:** use 2-bit BHT used to encode 4 states in order to change the prediction after 2 mispredictions.
- *The prediction must mispredict twice before it is changed!*

- Considering a loop branch, at the last loop iteration, we mispredict the branch but we do not need to change the prediction. When re-entering the loop, the branch is correctly predicted as taken.
- **Example:** In a loop branch of 10 iterations whose behavior is Taken nine times and Not Taken once => the prediction accuracy is 90% (due to only 1 misprediction at last iteration and 9 correct predictions out of 10 iterations).

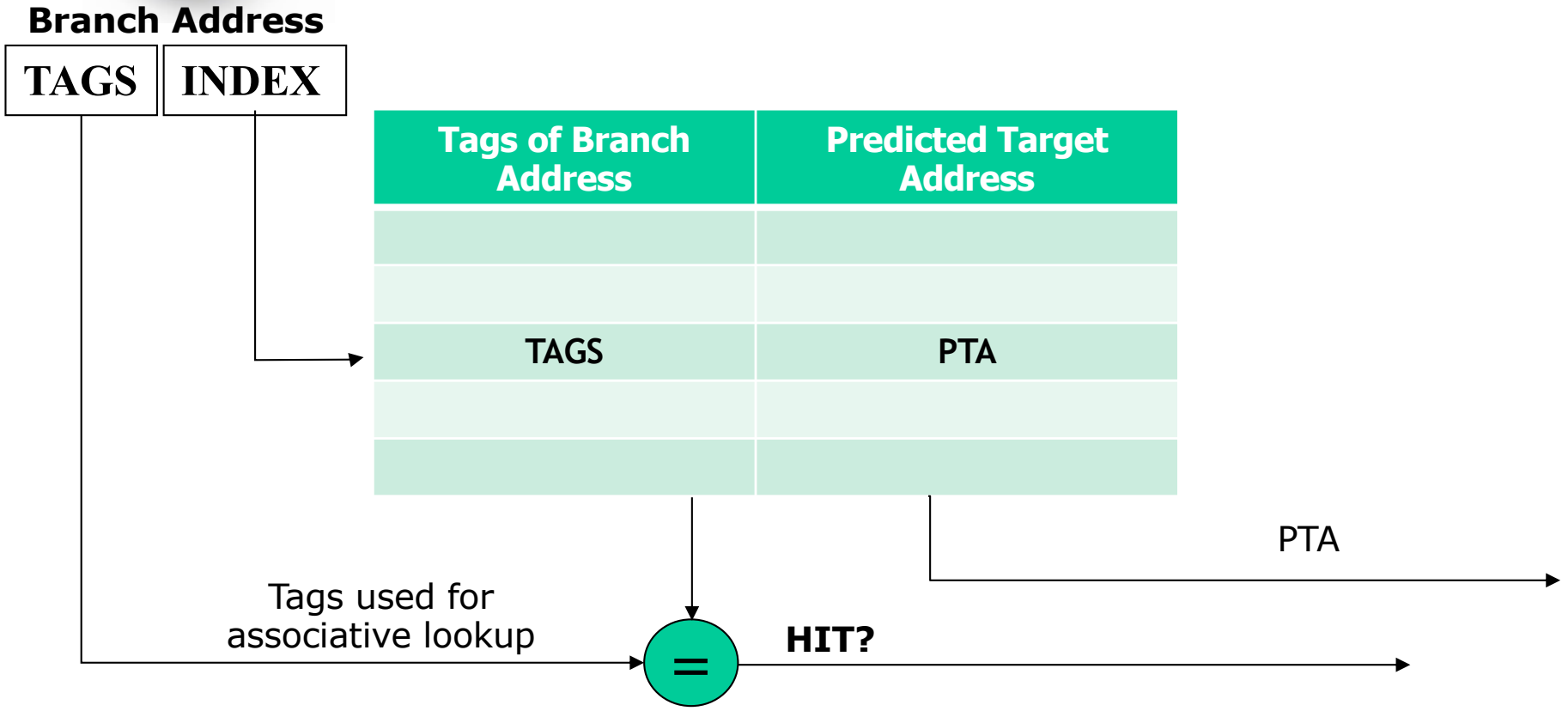
2) Branch Target Buffer

- **Branch Target Buffer (Branch Target Predictor)** is a cache storing the **Predicted Target Address (PTA)** for the taken-branch instructions. The **PTA** is expressed as PC-relative.

- The **BTB** is designed as a direct-mapped cache placed in the **IF stage** by using the address of the fetched branch instruction to index the cache, then **tags** are used for the associative lookup.

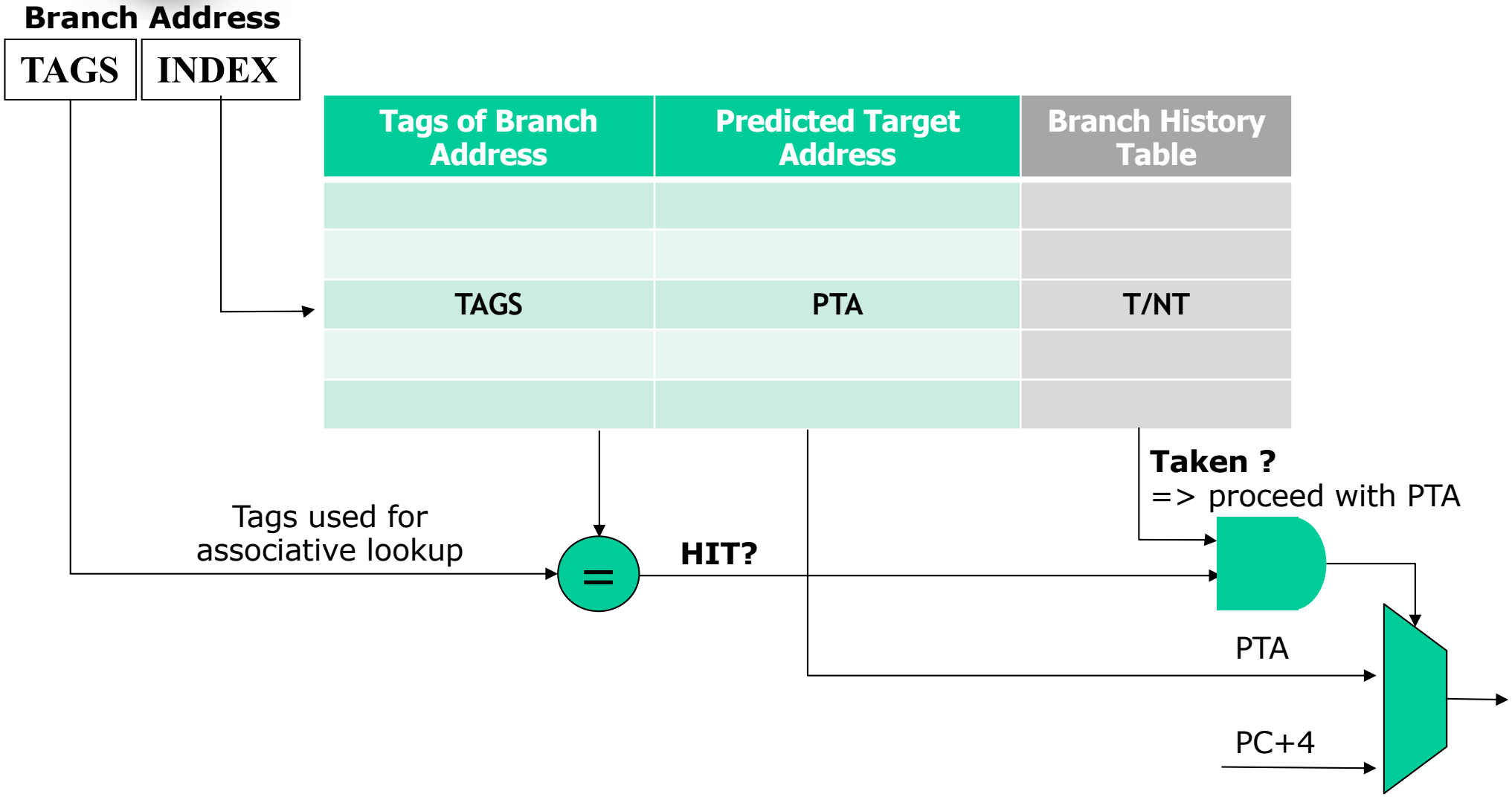
- The **BTB** is used in combination with the **Branch History Table** in the **IF stage.**

2) Structure of a Branch Target Buffer



- Usually, it is combined with a **Branch Outcome Predictor** such as a 1-bit (or 2-bit) Branch History Table.

2) Structure of a Branch Target Buffer



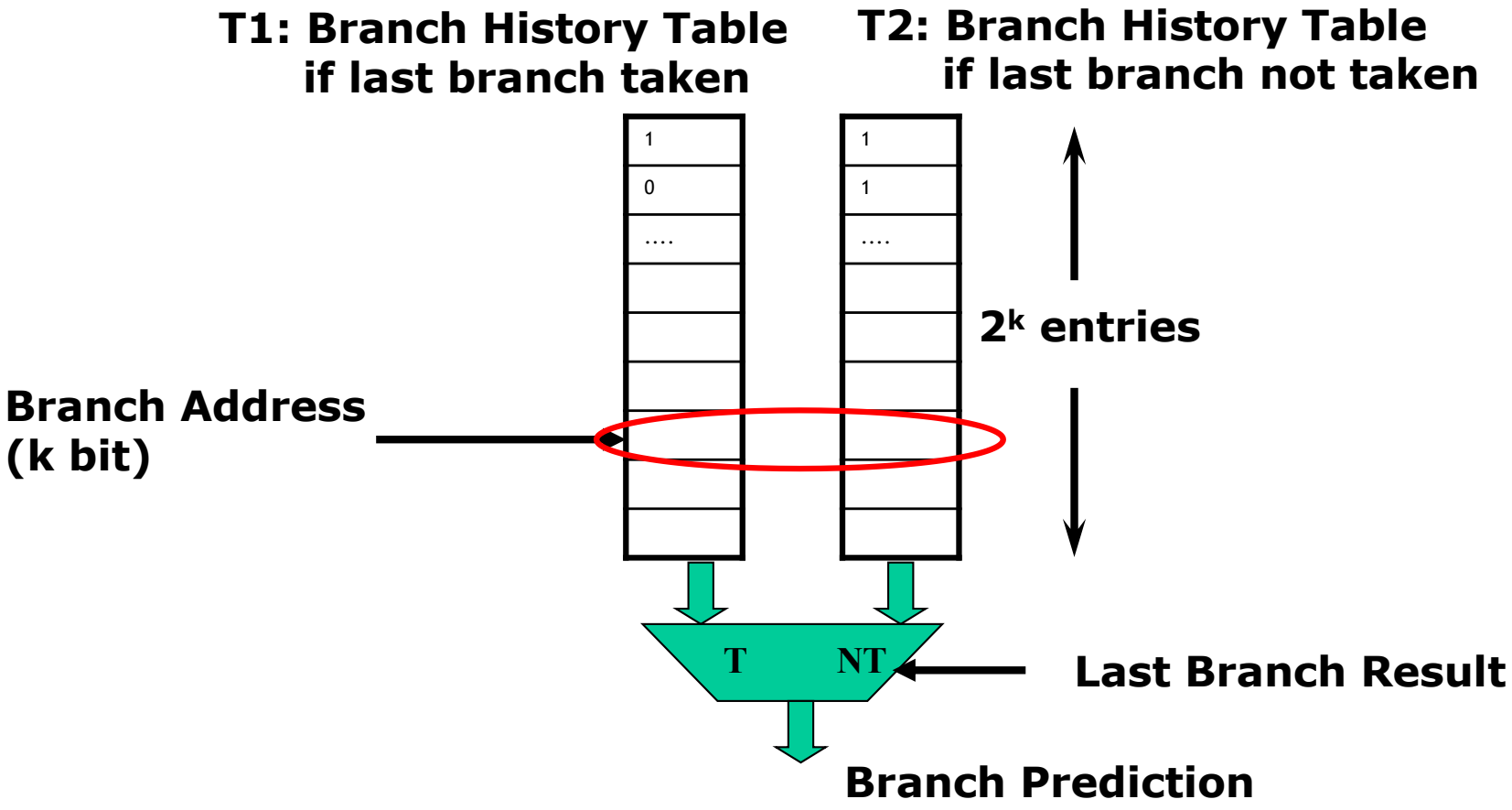
3) Correlating Branch Predictors

- The 2-bit BHT uses only the recent behavior of a single branch to predict the future behavior of that branch.
- **Basic Idea:** the behavior of recent branches are correlated, that is the recent behavior of ***other branches*** rather than just the current branch (we are trying to predict) can influence the prediction of the current branch.
- We try to exploit the correlation existing among different branches: branches are partially based on the same conditions => they can generate information that can influence the behavior of other branches.

3) Correlating Branch Predictors

- Branch predictors that use the behavior of other branches to make a prediction are called **Correlating Predictors** or **2-level Predictors.**
- Example a **(1,1) Correlating Predictors** means a 1-bit predictor with 1-bit of correlation: the behavior of last branch is used to choose among a pair of 1-bit branch predictors.

3) Correlating Branch Predictors: Example



3) Correlating Branch Predictors

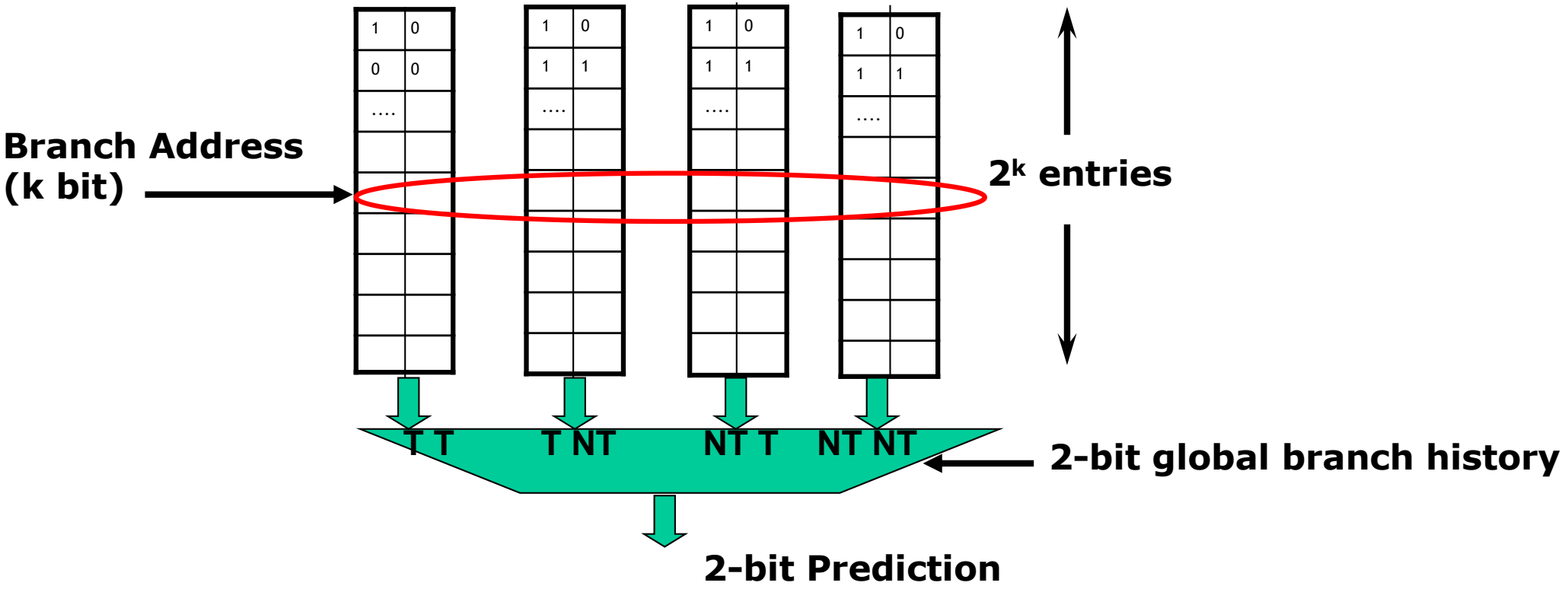
- Record if the most recently executed branches have been ***taken*** o ***not taken***.
- The branch is predicted based on the previous executed branch by selecting the appropriate 1-bit BHT:
  - One prediction is used if the last branch executed was **taken**
  - Another prediction is used if the last branch executed was **not taken.**
- In general, the last branch executed is ***not*** the same instruction as the branch being predicted (although this can occur in simple loops with no other branches in the loops).

(m, n) Correlating Branch Predictors

- In general **(m, n)** correlating predictor records last m branches to choose from 2^m BHTs, each of which is a n-bit predictor.
- The branch prediction buffer can be indexed by using a concatenation of low-order bits from the branch address with m-bit global history (i.e. global history of the most recent m branches).
- A 2-bit BHT predictor with no global history is simply a (0, 2) predictor.

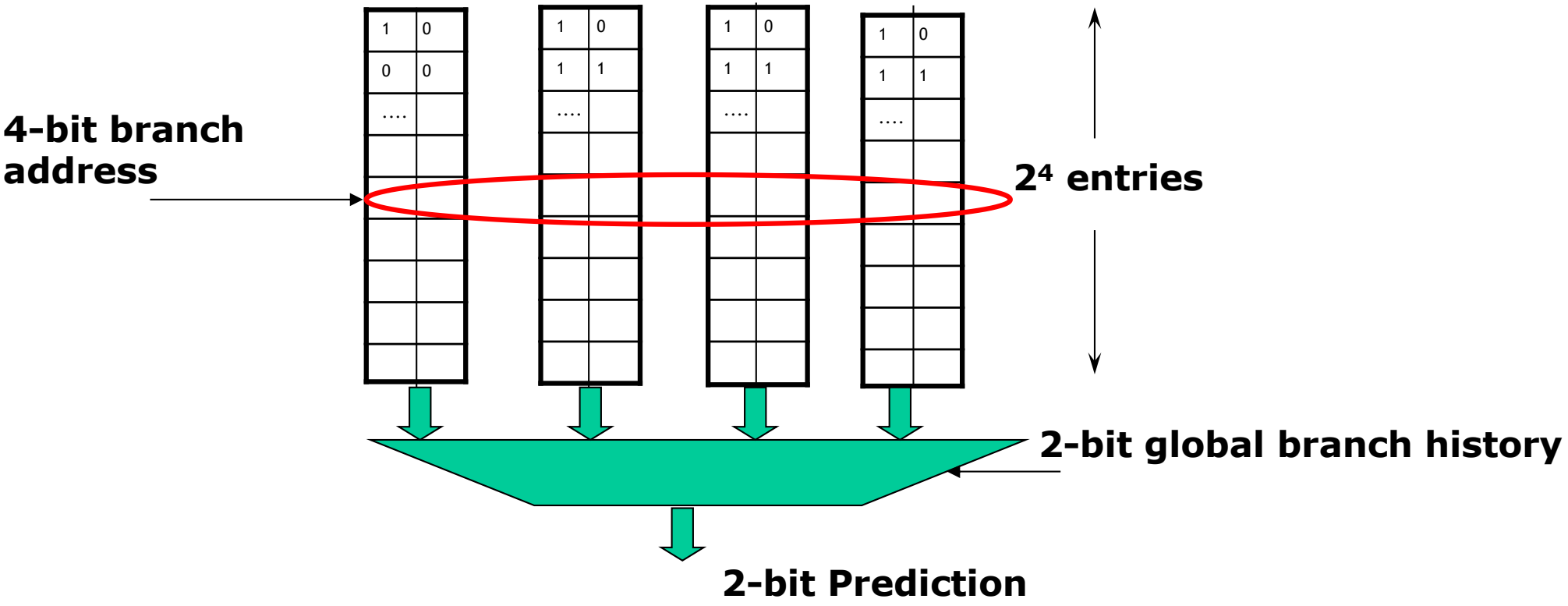
(2, 2) Correlating Branch Predictors

- A **(2, 2)** correlating predictor has 4 2-bit Branch History Tables.
  - It uses the 2-bit global history to choose among the 4 BHTs.



Example of (2, 2) Correlating Predictor

- Example: a **(2, 2)** correlating predictor with 64 total entries
  $\Rightarrow$ 6-bit index composed of: 2-bit global history and 4-bit low-order branch address bits

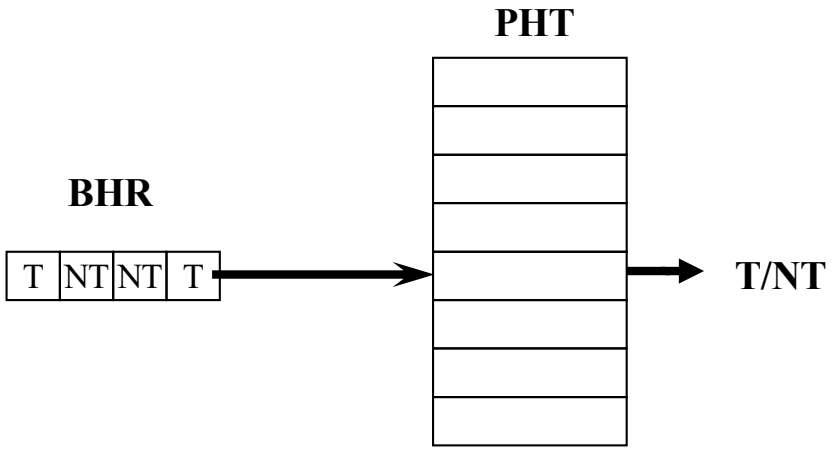


4) Two-Level Adaptive Branch Predictors

- The first level history is recorded in one (or more) k-bit shift register called **Branch History Register (BHR),** which records the outcomes of the k most recent branches (i.e. T, NT, NT, T) ***(used as a global history)***
- The second level history is recorded in one (or more) tables called **Pattern History Table (PHT)** of two-bit saturating counters ***(used as a local history)***
- The BHR is used to index the PHT to select which 2-bit counter to use.
- Once the two-bit counter is selected, the prediction is made using the same method as in the 2-bit counter scheme.

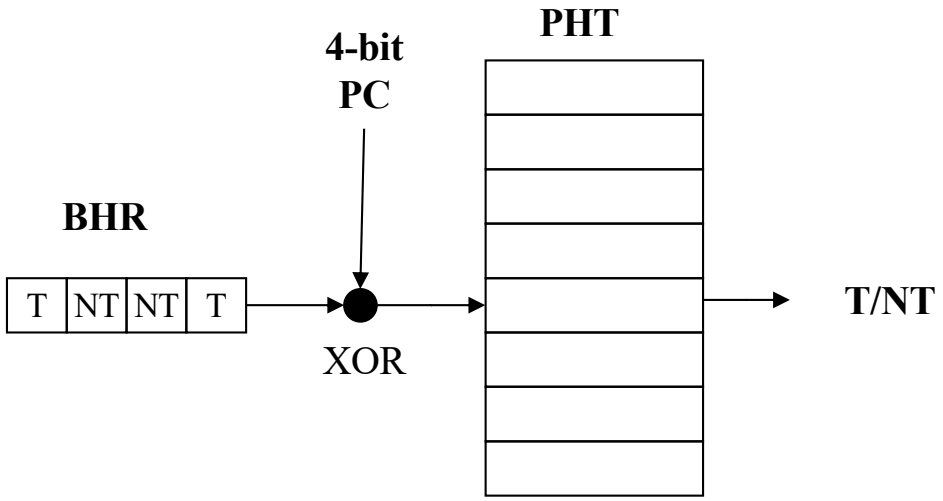
4) Global Adaptive Predictor

- The global 2-level predictor uses the correlation between the current branch and the other branches in the global history to make the prediction
- **GA:** Based on global and local predictor
  - 2-level predictor: PHT (local history) indexed by the content of BHR (global history)



4) GShare Predictor

- Variation of the **GA** predictor where want to correlate the BHR recording the outcomes of the most recent branches (global history) with the low-order bits of the branch address
- **GShare:** We make the XOR of 4-bit BHR (global history) with the low-order 4-bit of PC (branch address) to index the PHT (local history).



Branch Prediction Techniques: Performance

- **Main goal:** try to guess as soon as possible the outcome of a branch instruction.

- The **performance** of a branch prediction technique depends on:
  - **Accuracy** measured in terms of percentage of incorrect predictions given by the predictor.
  - **Cost** of an incorrect prediction measured in terms of time lost to execute useless instructions **(misprediction penalty)** given by the processor architecture: the cost increases for deeply pipelined processors
  - **Branch frequency** given by the application: the importance of accurate branch prediction is higher in programs with higher number of branch instructions.

References

- An introduction to the branch prediction problem can be found in **Appendix A** and **Chapter 3** of the reference book: J. Hennessy and D. Patterson, “Computer Architecture, a Quantitative Approach”, Morgan Kaufmann, Fourth Edition.