POLITECNICO MILANO 1863 | DIPARTIMENTO DI ELETTRONICA INFORMAZIONE E BIOINGEGNERIA

Advanced Computer Architectures

Branch prediction

A.Y. 2024/2025 I Christian Pilato (christian.pilato@polimi.it)

POLITECNICO MILANO 1863 | DIPARTIMENTO DI ELETTRONICA INFORMAZIONE E BIOINGEGNERIA

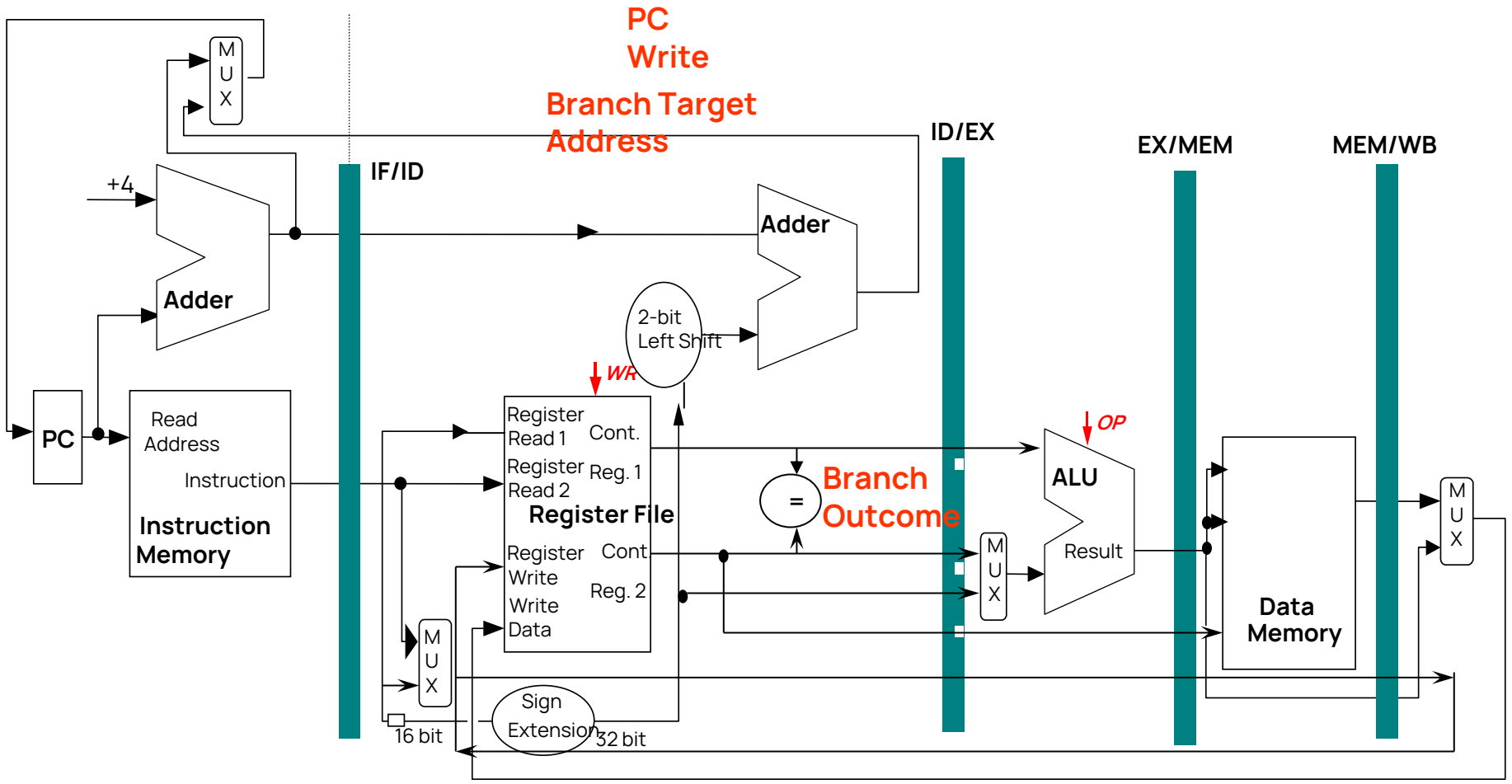
Outline

Branch Prediction Techniques

- Static Branch Prediction
- Dynamic Branch Prediction

Performance of Branch Schemes

Effects of branches on the pipeline



Each branch costs **one stall** to fetch the correct instruction flow: (PC+4) or the branch target address

Branch Prediction Techniques

Main goal: try to predict as early 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 deep pipelined processors (more stages to flush)
- **Branch frequency** in the application: the importance of accurate branch prediction is higher in programs with higher branch frequency

Branch Prediction Techniques

There are many methods to deal with the performance loss due to branch hazards:

- **Static Branch Prediction Techniques**: The actions for a branch are fixed for each branch during the entire execution. The actions are fixed at compile time
- **Dynamic Branch Prediction Techniques**: The decision of the branch prediction can change during the program execution

In both cases, it is important not to change the processor state until the branch is effectively known

Static Branch Prediction Techniques

Static Branch Prediction is used in processors where the expectation is that the branch behavior is highly predictable at compile-time

Static Branch Prediction can also be used to assist dynamic predictors

Static Branch Prediction Techniques

Branch Always Not Taken (Predicted-Not-Taken)

Branch Always Taken (Predicted-Taken)

Backward Taken Forward Not Taken (BTFNT)

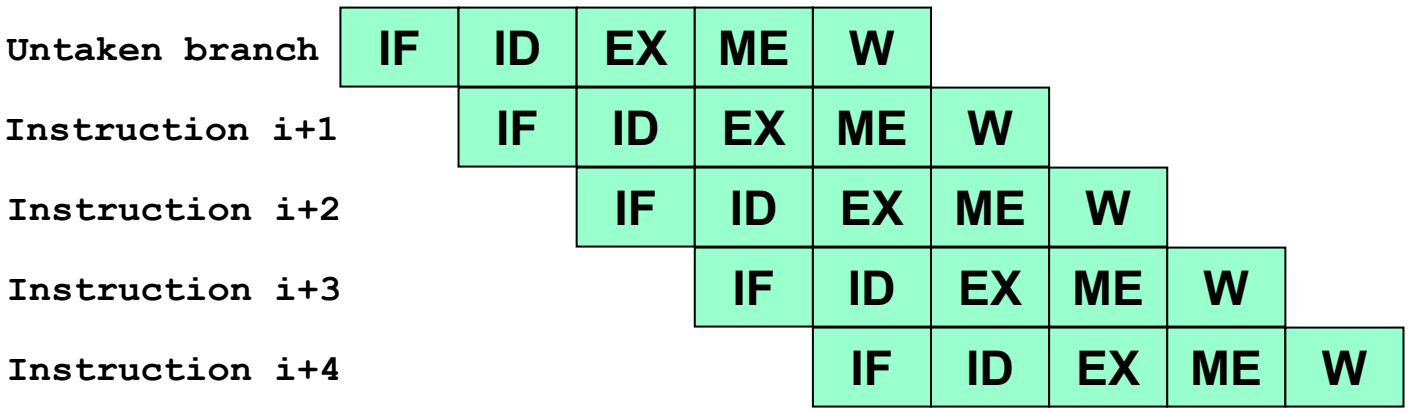
Profile-Driven Prediction

Delayed Branch

Branch Always Not Taken

We assume the **branch will not be taken**

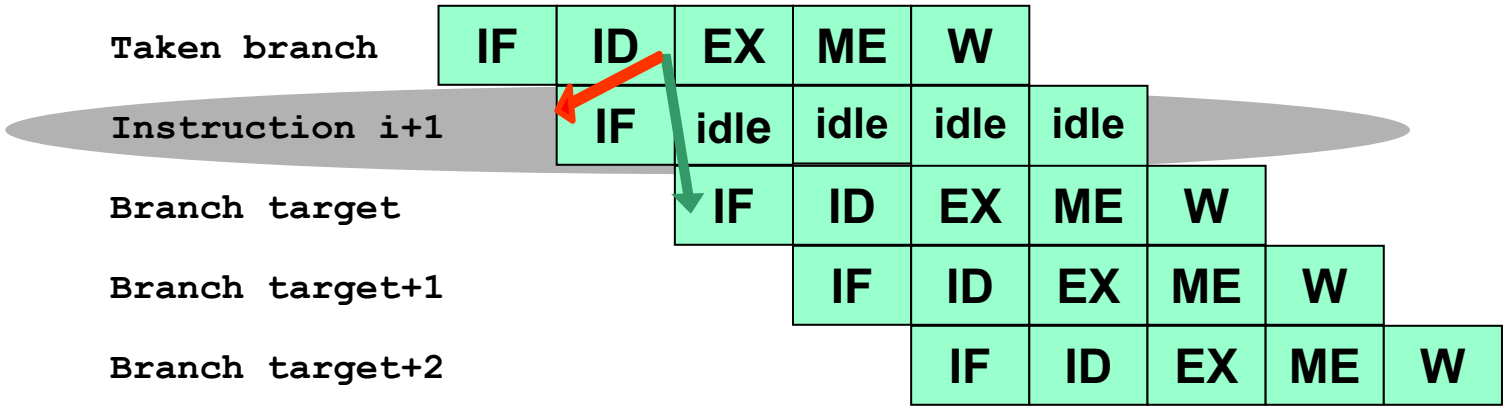
- The sequential instruction flow we have fetched can continue as if the branch condition was not satisfied
- If the condition in stage ID is not satisfied **(the prediction is correct)**, we can preserve the performance



Branch Always Not Taken

If the condition in stage ID is satisfied **(the prediction is incorrect)**, the branch is **taken:**

- We need to **flush** the next instruction already fetched (the fetched instruction is turned into a **`nop`**)
- We restart the execution by fetching the instruction at the branch target address ⇒ **One-cycle penalty**



Branch Always Taken

An alternative scheme is to consider **every branch as taken**:

- as soon as the branch is decoded and the branch target address is computed, we assume the branch to be taken
- we begin fetching and executing the instruction at the target address

The predicted-taken scheme makes sense for pipelines where the branch target is known before the branch outcome

In the MIPS pipeline, we don't know the branch target address earlier than the branch outcome, so this approach has no advantage for this pipeline

Backward Taken Forward Not Taken (BTFNT)

The prediction is based on the branch direction:

- **Backward-going branches** are predicted as taken
  - Example: the branches at the end of loops go back at the beginning of the next loop iteration
    $\Rightarrow$ we assume the backward-going branches are always taken
- **Forward-going branches** are predicted as not taken

Profile-Driven Prediction

The branch prediction is based on profiling information collected from earlier runs

The method can use compiler hints

Delayed Branch Technique

The compiler statically schedules an independent instruction in the **branch delay slot**

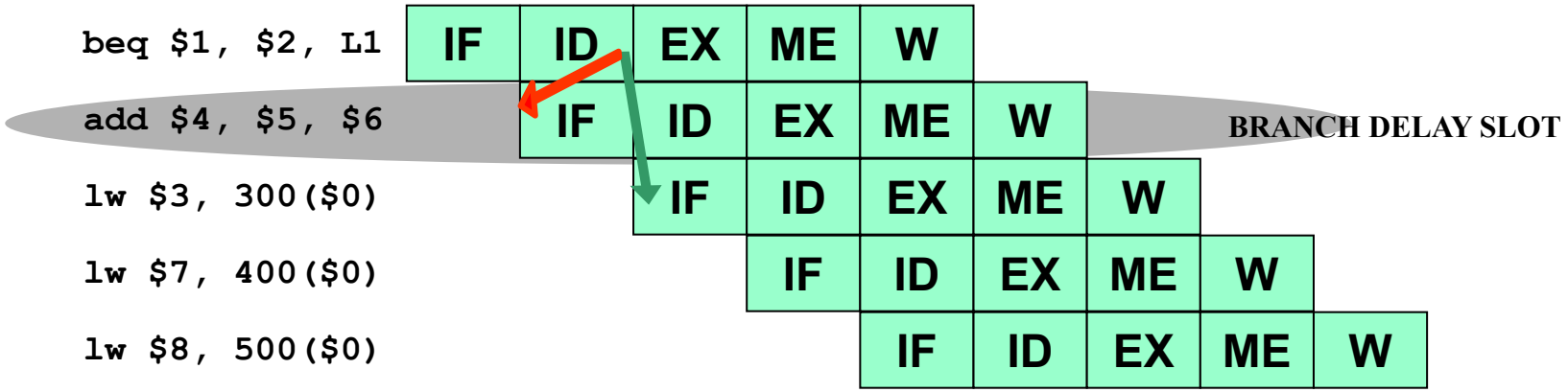
The instruction in the branch delay slot is executed whether the branch is taken or not

- If we assume a branch delay of one cycle (as for MIPS)
  $\Rightarrow$ we have only **one delay slot**
- In some deep pipeline processors, it is possible to have a branch delay longer than one cycle
  $\Rightarrow$ almost all processors with delayed branches have a **single delay slot** (since it is usually difficult for the compiler to fill in more than one delay slot)

Delayed Branch Technique

The MIPS compiler always schedules a branch-independent instruction after the branch

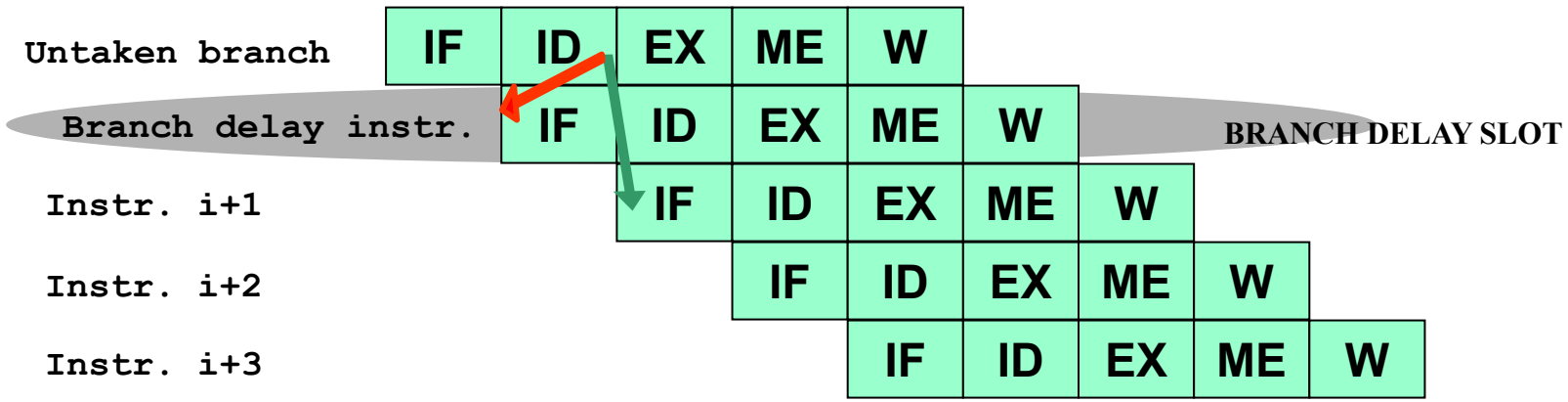
Example: A previous **`add`** instruction without any effects on the branch is scheduled in the **Branch Delay Slot**



Delayed Branch Technique

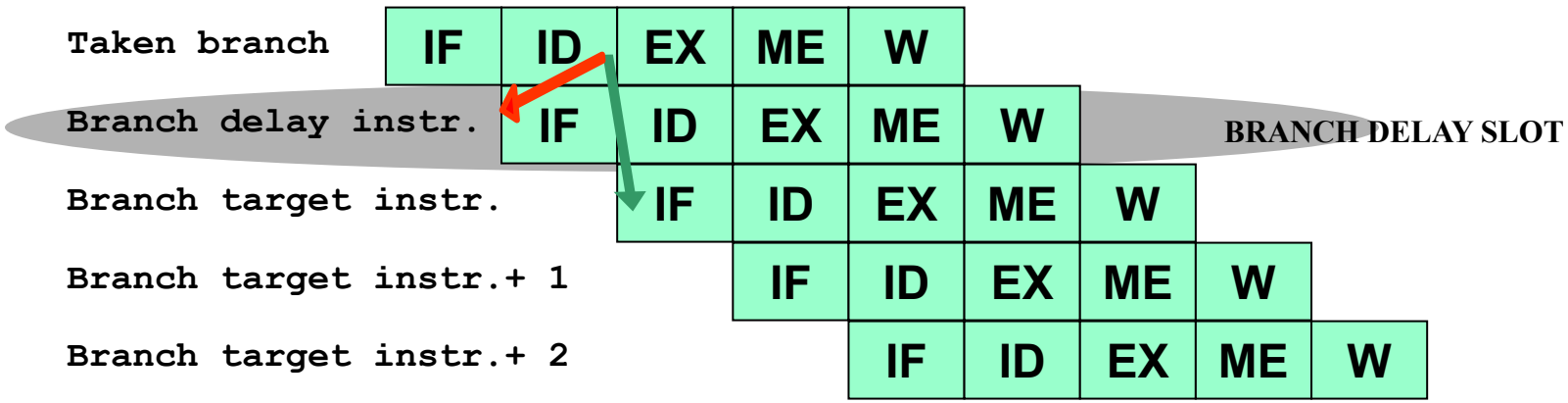
The behavior of the delayed branch is the same whether the branch is taken or not

- If the branch is **untaken** ⇒ execution continues with the instruction after the branch



Delayed Branch Technique

If the branch is **taken** ⇒ execution continues at the branch target



Delayed Branch Technique

The job of the compiler is to place a valid and useful instruction in the branch delay slot

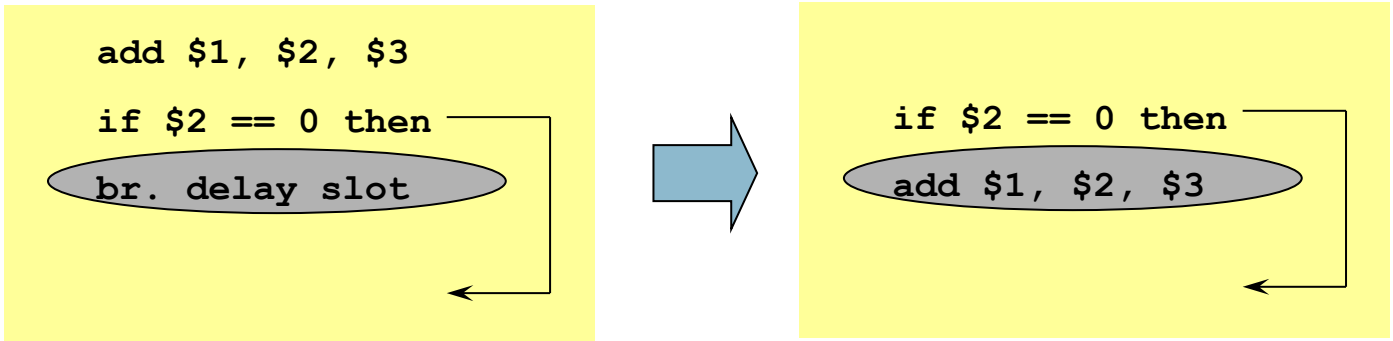
There are three ways in which the branch delay slot can be scheduled:

1. **From before**
2. **From target**
3. **From fall-through**

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 is taken or untaken)

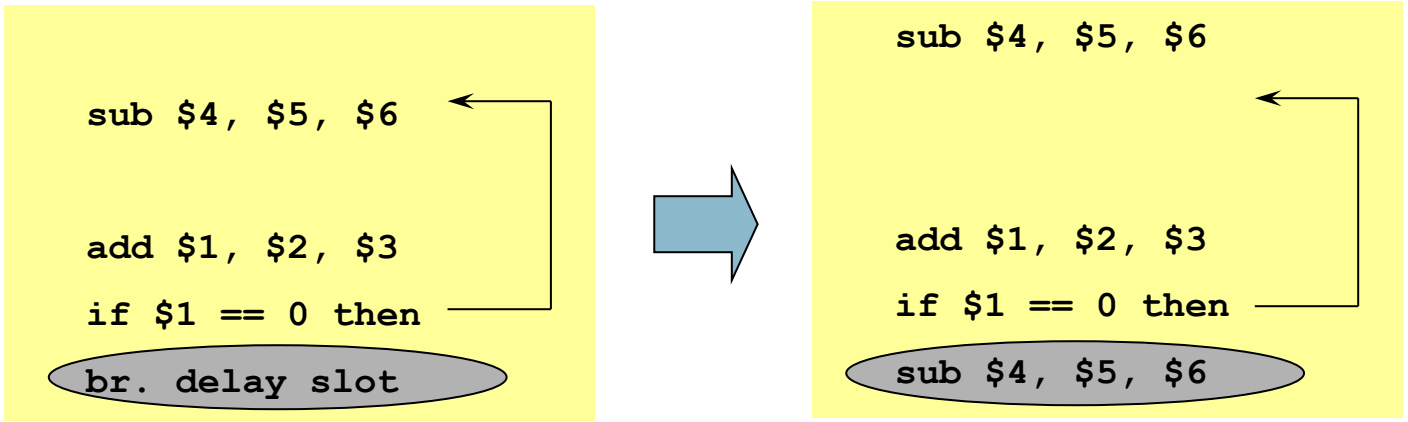


Delayed Branch Technique: From Target

The use of **`$1`** in the branch condition prevents **`add`** instruction (whose destination is **`$1`**) from being moved after the branch

The branch delay slot is scheduled from the target of the branch (usually, the target instruction will need to be copied because it can be reached by another path)

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

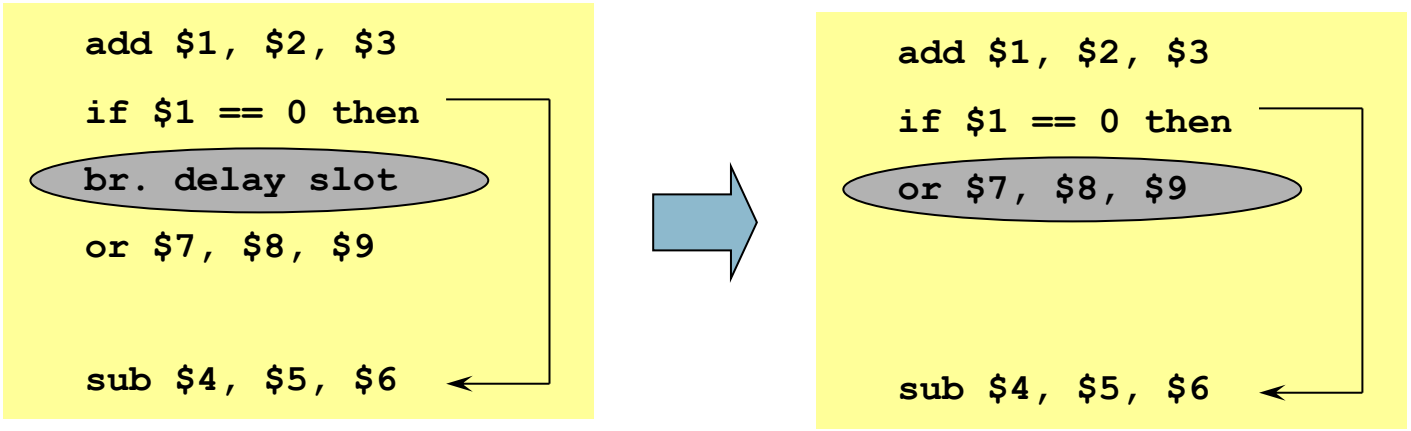


Delayed Branch Technique: From Fall-Through

The use of **$1** in the branch condition prevents add instruction (whose destination is **$1**) from being moved after the branch

The branch delay slot is scheduled from the **not-taken fall-through path**

This strategy is preferred when the branch is not taken with high probability, such as **forward branches**



Delayed Branch Technique

To make the optimization legal for the target and fall-through cases, it must be OK to execute the moved instruction 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 (the program will still execute correctly)

Example: the destination register is an unused temporary register when the branch goes in the “unexpected” direction

Delayed Branch Technique

In general, the compilers are able to fill about 50% of delayed branch slots with valid and useful instructions, while the remaining slots are filled with **`nops`**

In deep pipelined processors, the delayed branch is longer than one cycle: many slots must be filled for every branch

- it is more challenging to fill all the slots with useful instructions

Delayed Branch Technique

The main limitations on delayed branch scheduling arise from:

- The restrictions on the instructions that can be scheduled in the delay slot
- The ability of the compiler to statically predict the outcome of the branch

Delayed Branch Technique

To improve the ability of the compiler to fill the branch delay slot ⇒ most processors have introduced a **canceling or nullifying branch**: the instruction includes the direction in which the branch was predicted

- When the branch behaves as predicted ⇒ the instruction in the branch delay slot is executed normally
- When the branch is incorrectly predicted ⇒ the instruction in the branch delay slot is turned into a `nop` (flushed)

In this way, the compiler does not need to be conservative when filling the delay slot

Dynamic Branch Prediction

Basic Idea: To use the past branch behavior to predict the future

- We use **hardware** to dynamically predict the **outcome of a branch**: the prediction will depend on the behavior of the branch at run time and will change if the branch changes its behavior during execution

- We start with a simple branch prediction scheme and then examine approaches that increase the branch prediction accuracy

Dynamic Branch Prediction Schemes

Dynamic branch prediction is based on two interacting mechanisms:

- **Branch Outcome Predictor (BOP):**

  To predict the direction of a branch (i.e., taken or not taken)

- **Branch Target Predictor (BTP):**

  To predict the branch target address in the case of taken branch

The **Instruction Fetch Unit** uses these modules to predict the next instruction to read in the I-cache

If the branch is not taken ⇒ PC is incremented

If the branch is taken ⇒ BTP gives the target address

Branch Target Buffer

Branch Target Buffer (Branch Target Predictor) is a cache storing the predicted branch target address for the next instruction after a branch

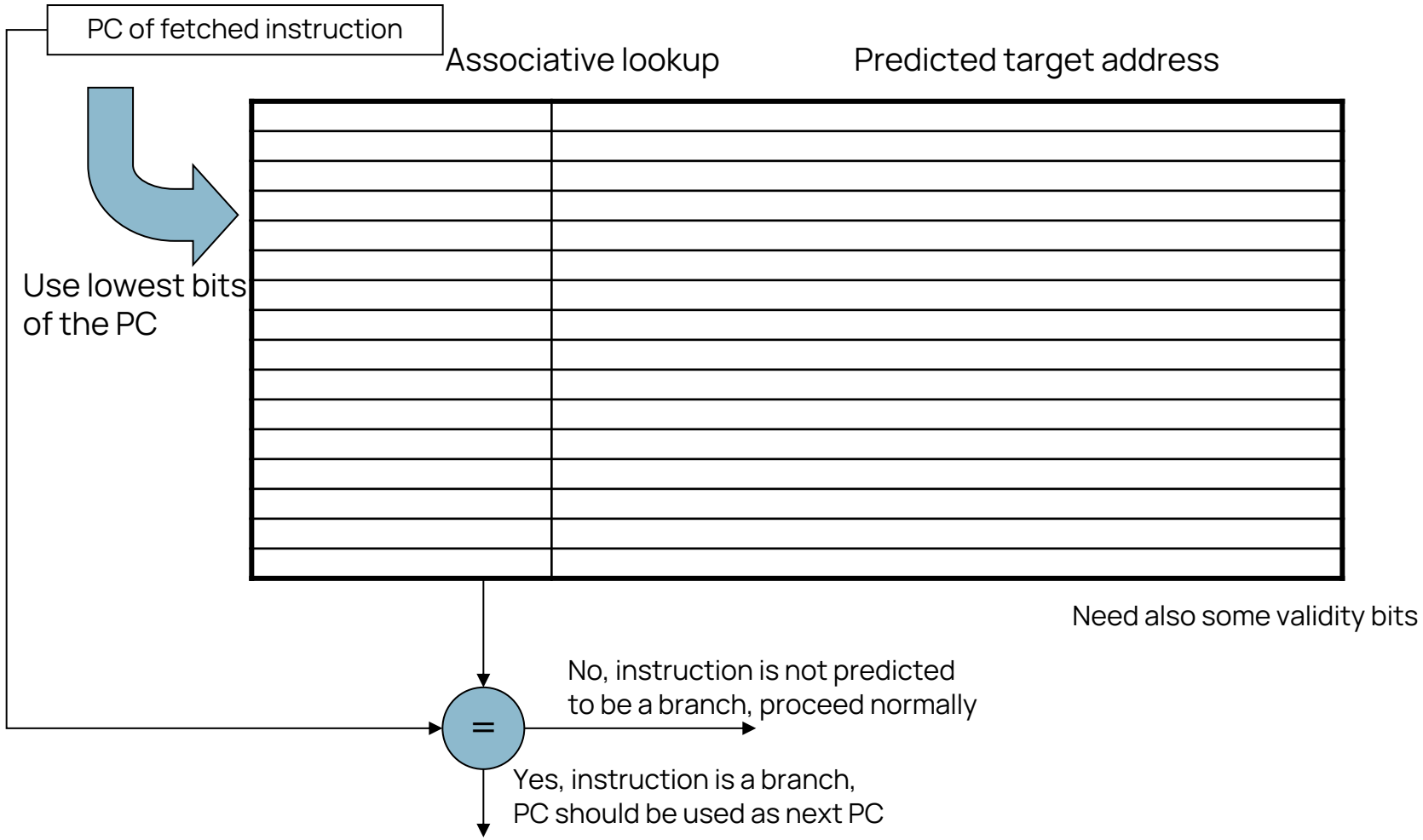
We access the BTB in the IF stage using the instruction address of the fetched instruction (a possible branch) to index the cache.

Typical entry of the BTB:

Exact Address of a Branch	Predicted target address

The predicted target address is expressed as PC-relative

Structure of a Branch Target Buffer



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
- Table indexed by the lower portion of the address of the branch instruction

Prediction: hint that it is assumed to be correct, fetching begins in the predicted direction

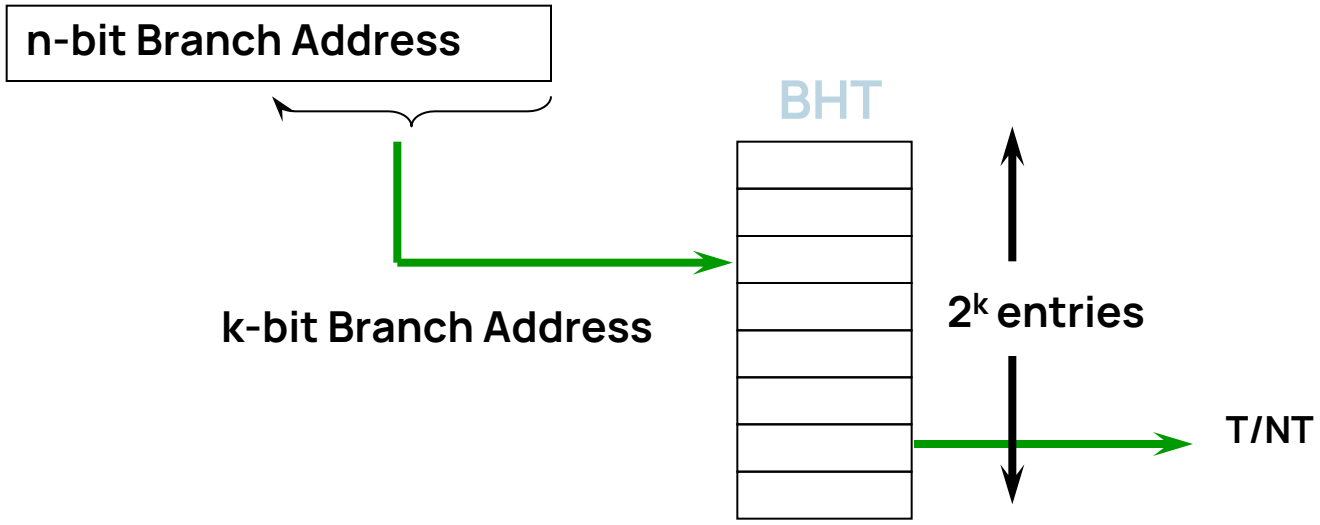
- If the hint turns out to be wrong, the prediction bit is inverted and stored back. The pipeline is flushed, and the correct sequence is executed

The table has no tags (every access is a hit)

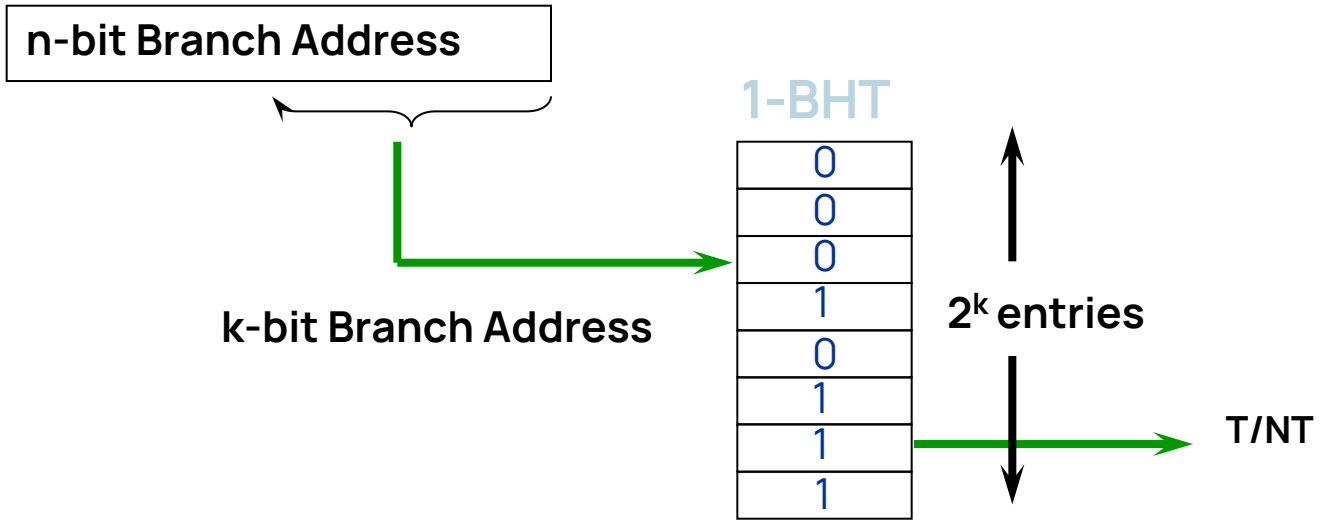
The prediction bit could have been put there by another branch with the same low-order address bits

- it does not matter. The prediction is just a hint!

Branch History Table

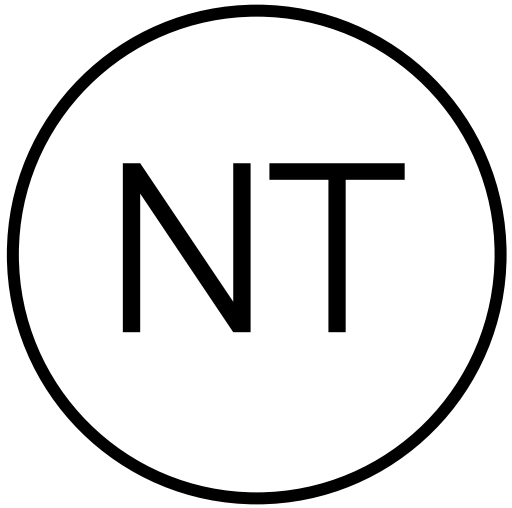


1-bit Branch History Table

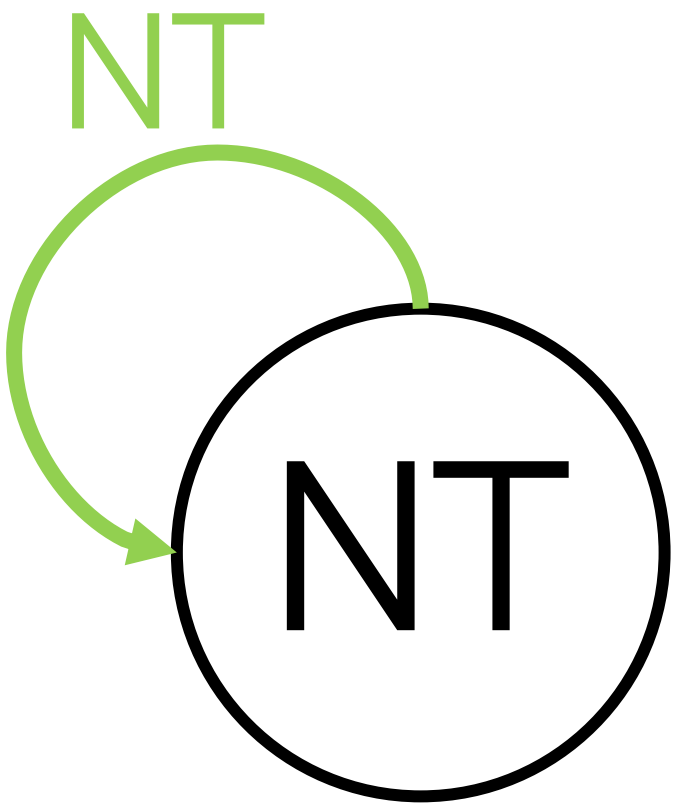


«BHT» behavior

Prediction: NT



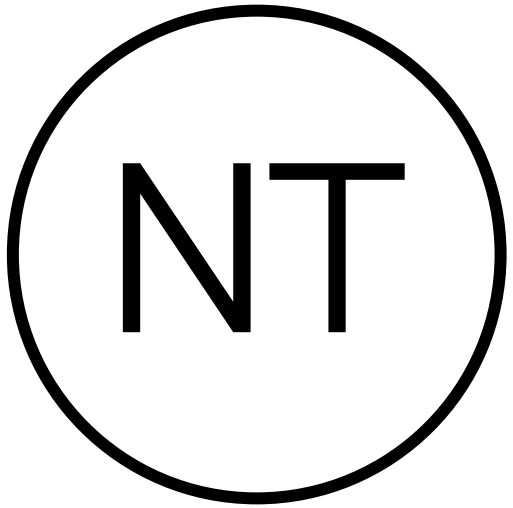
«BHT» behavior



Prediction: NT
Outcome: NT

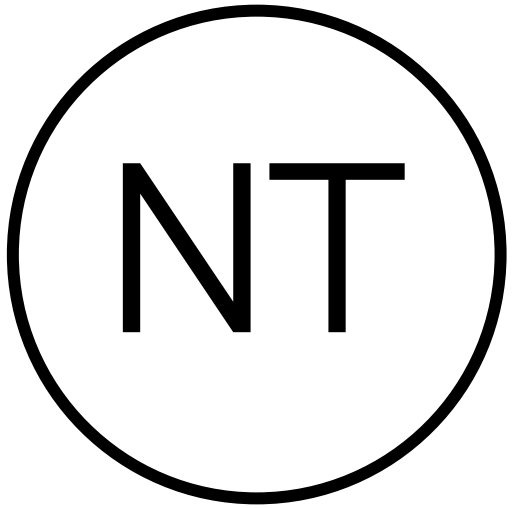
«BHT» behavior

Prediction: NT



«BHT» behavior

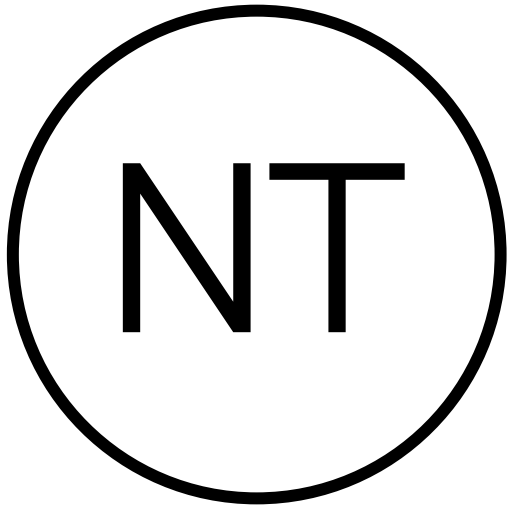
Prediction: NT
Outcome: T



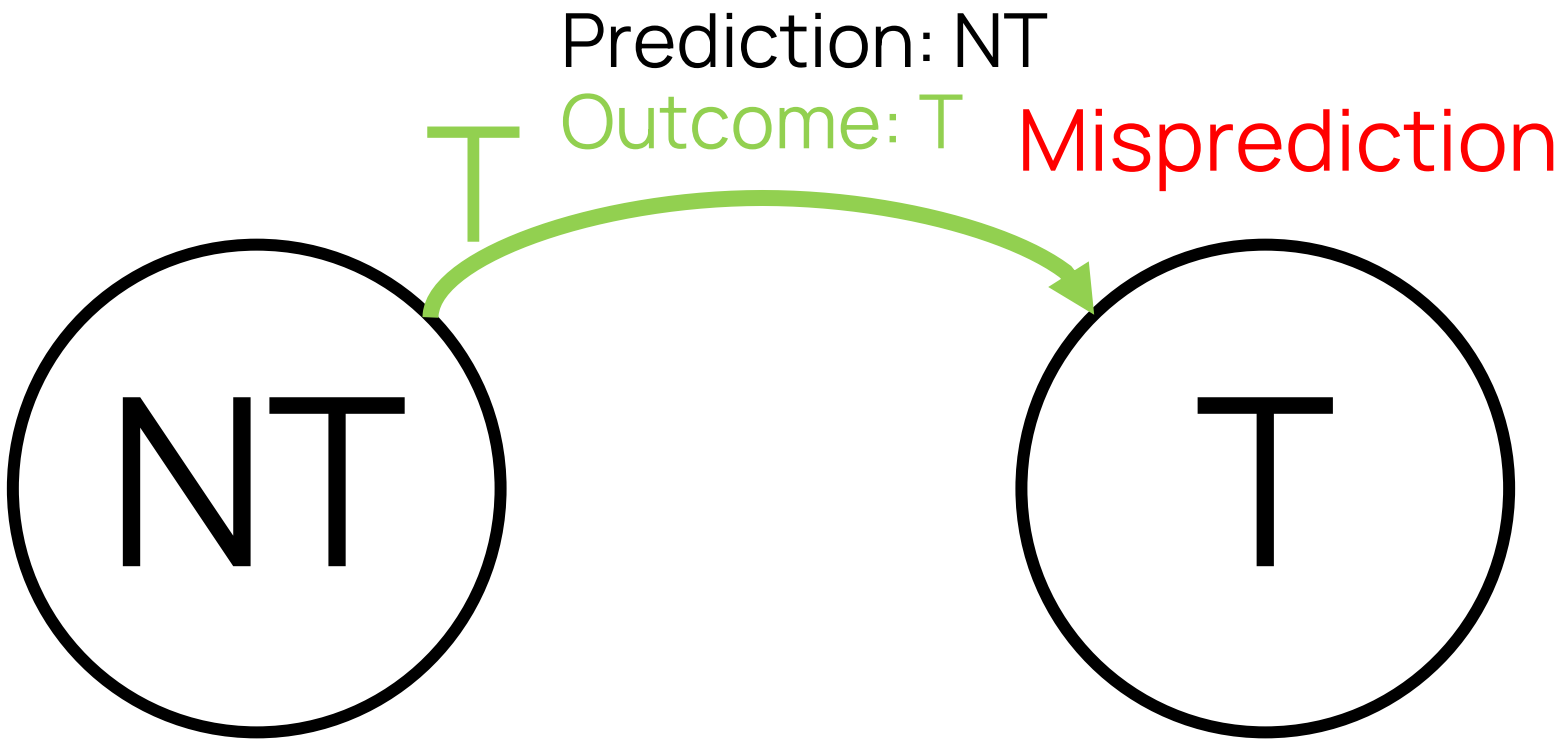
«BHT» behavior

Prediction: NT

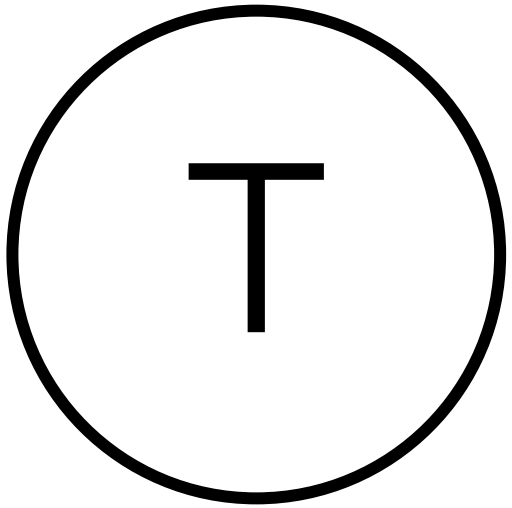
Outcome: T Misprediction



«BHT» behavior

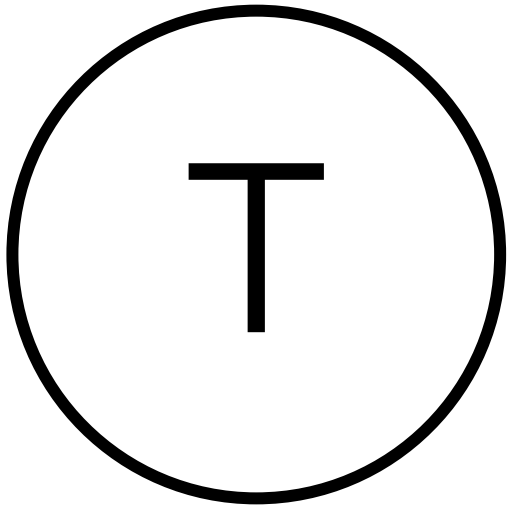


«BHT» behavior



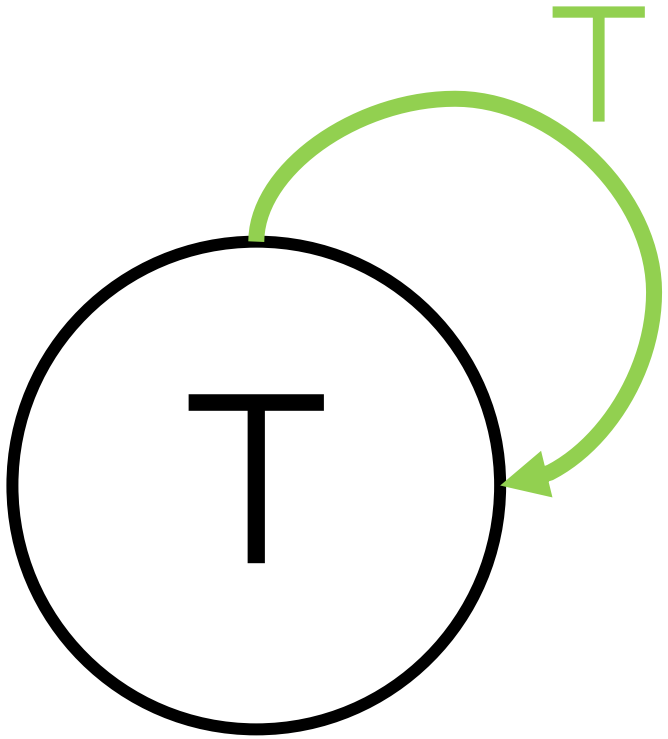
«BHT» behavior

Prediction: T



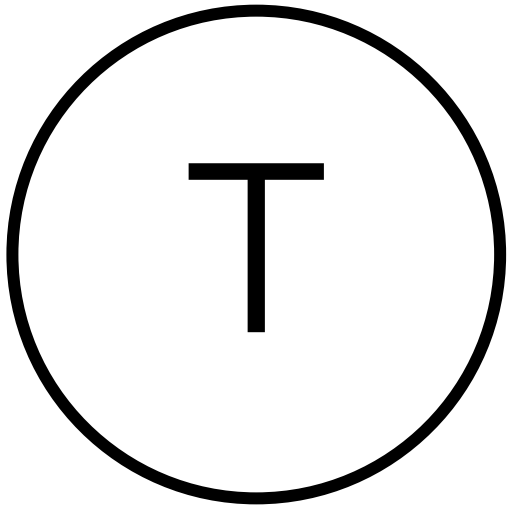
«BHT» behavior

Prediction: T
Outcome: T



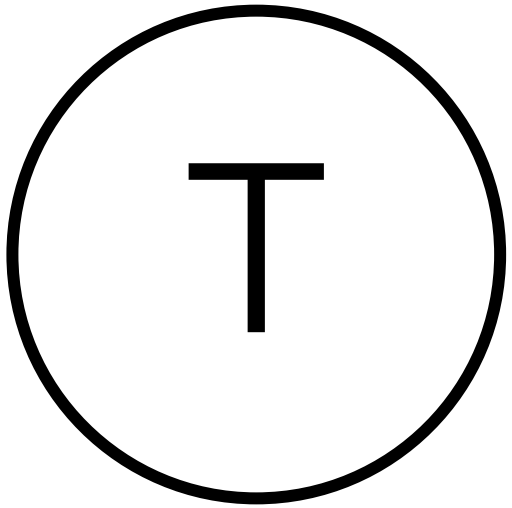
«BHT» behavior

Prediction: T



«BHT» behavior

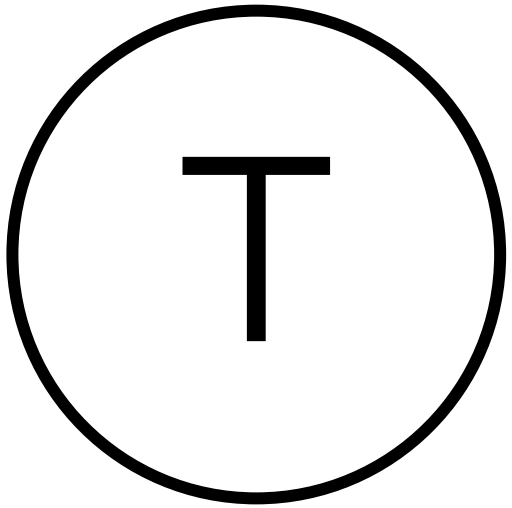
Prediction: T
Outcome: NT



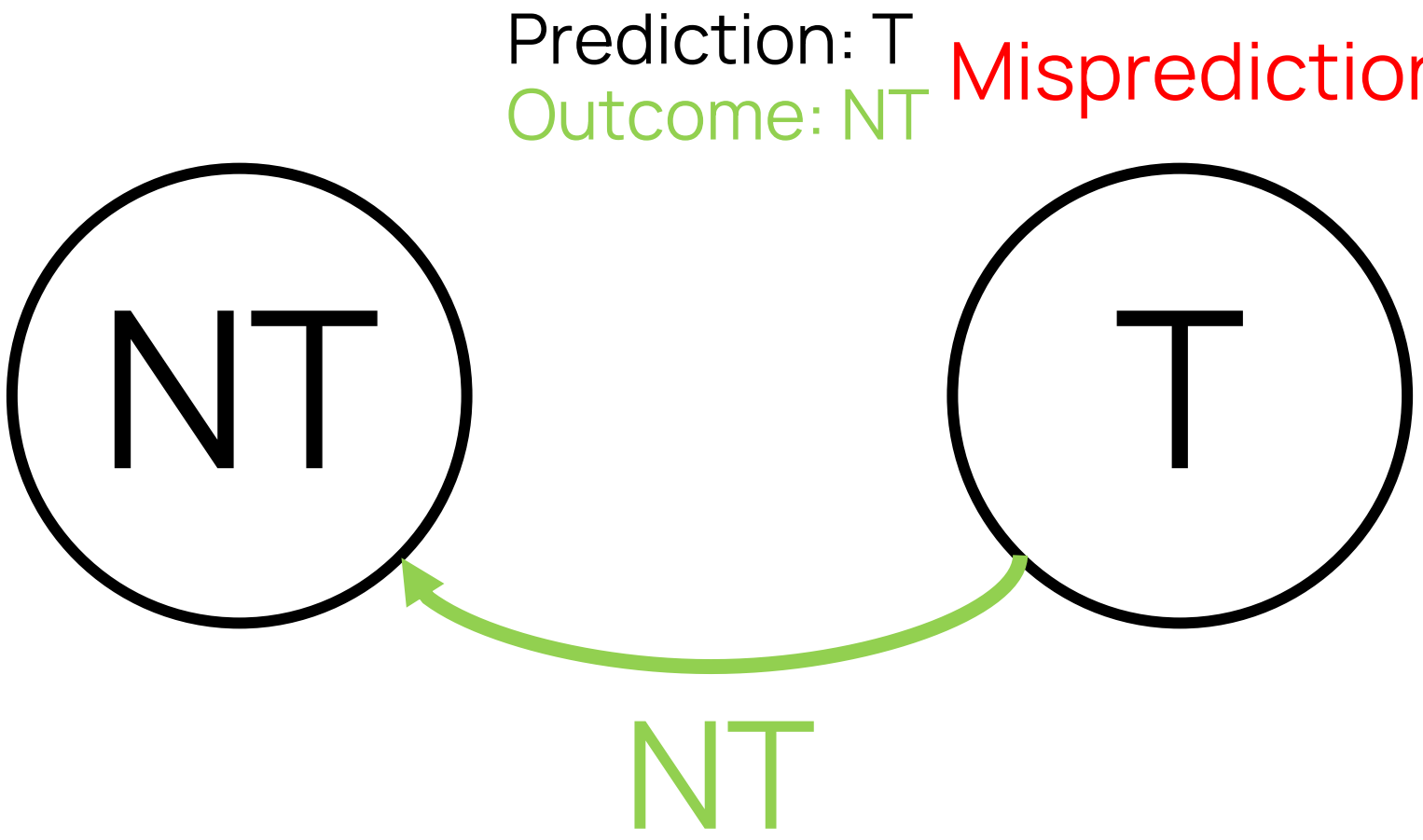
«BHT» behavior

Prediction: T
Outcome: NT

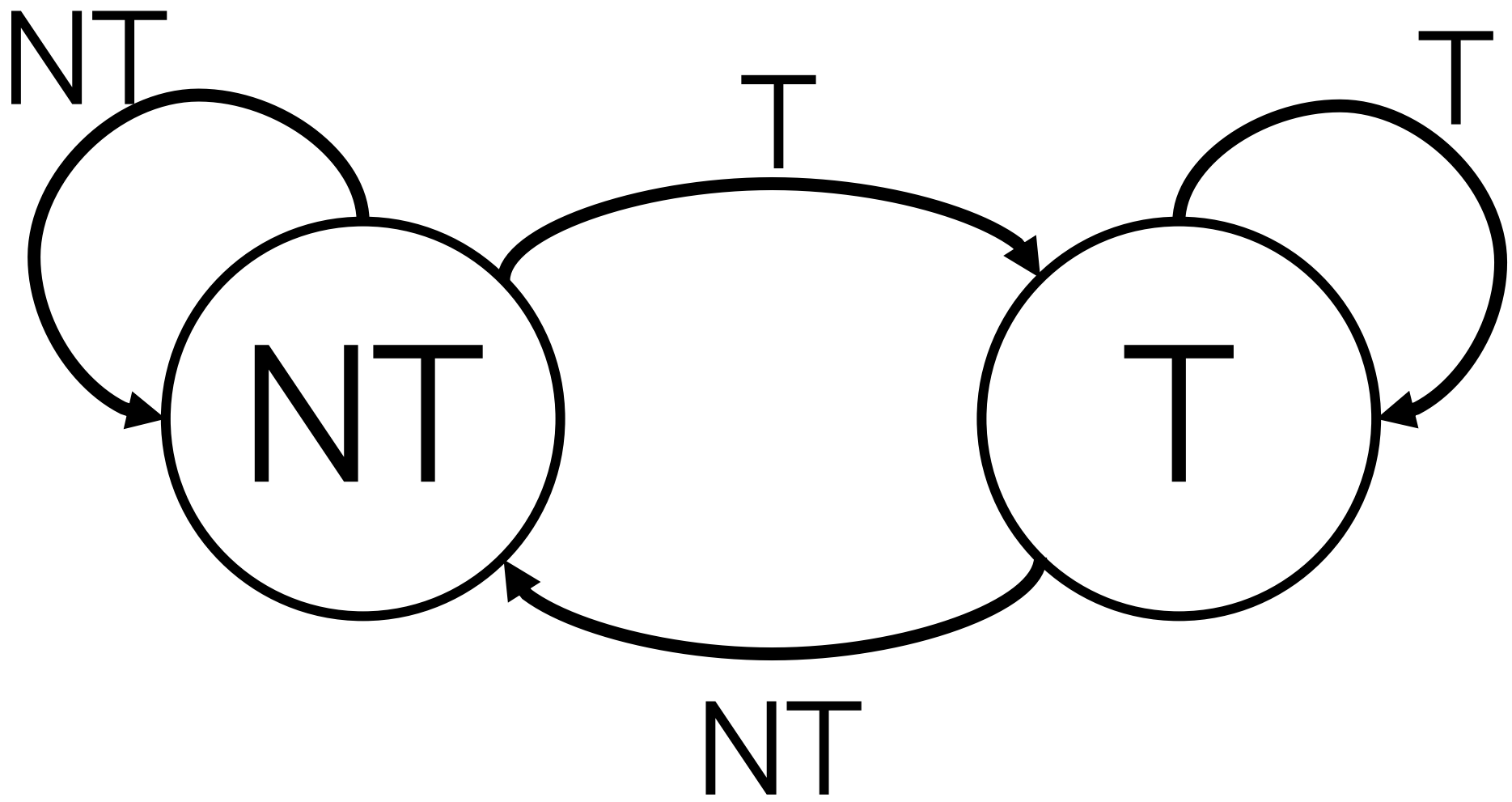
Misprediction



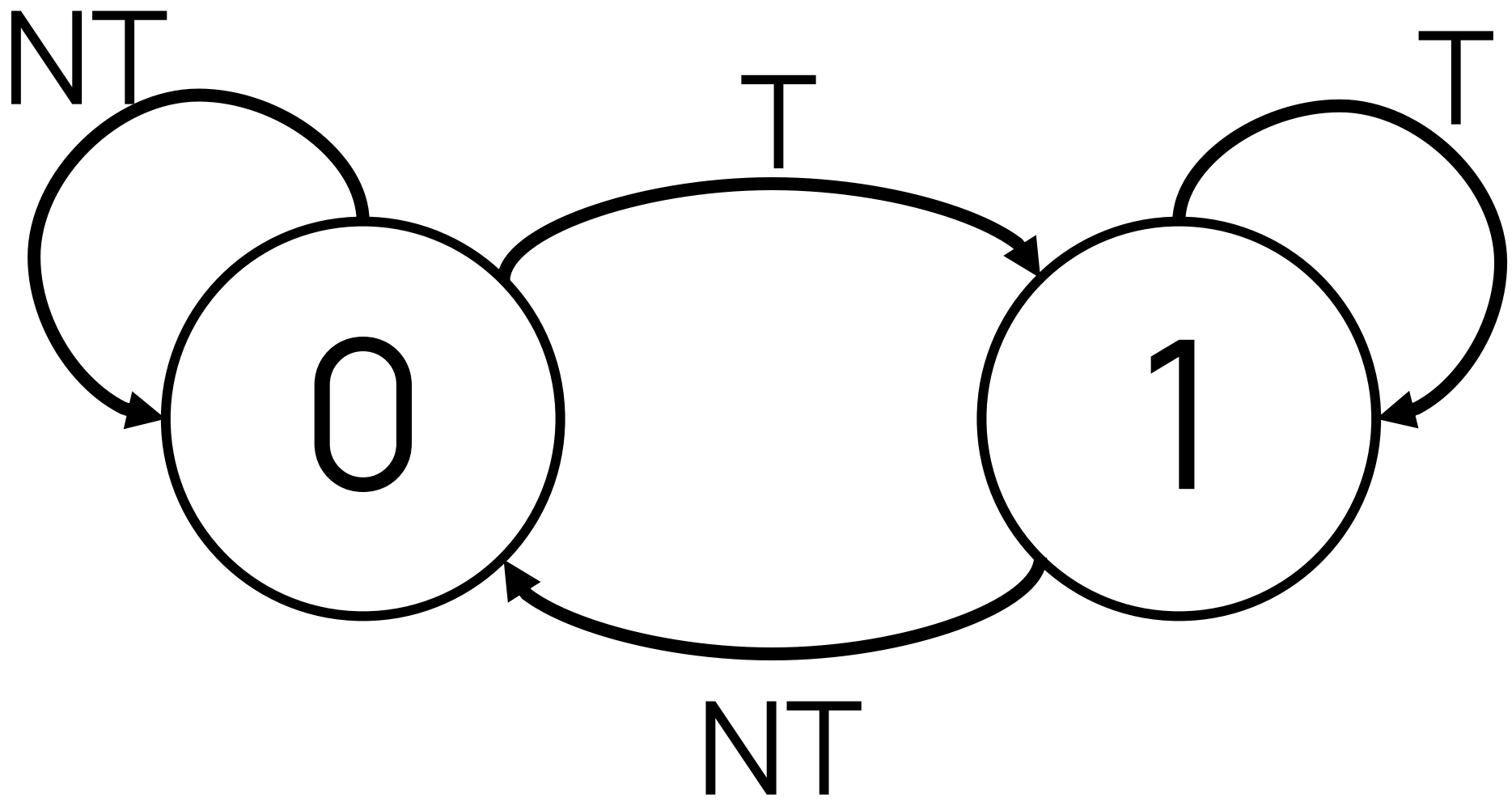
«BHT» behavior



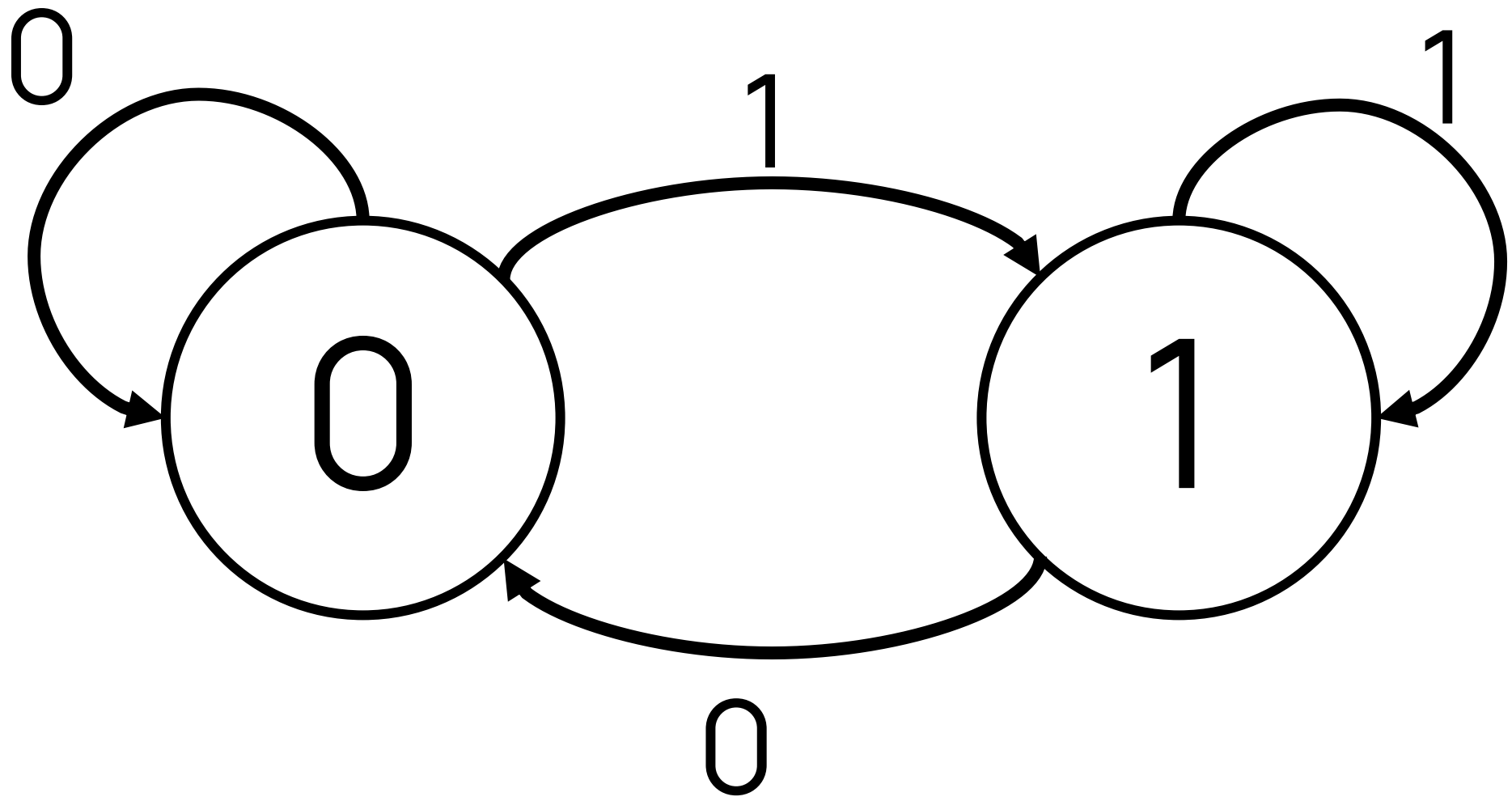
«BHT» behavior… at the end



«BHT» behavior… at the end



«BHT» behavior… at the end



Accuracy of the 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

To solve this problem, it is enough to **increase the number of rows** in the BHT or to use a **hashing function**

1-bit Branch History Table

A shortcoming of the 1-bit BHT: In a loop branch, even if a branch is almost always taken and then not taken once, the 1-bit BHT will mispredict twice (rather than once) when it is not taken

That scheme causes **two** wrong predictions:

- At the last loop iteration, since the prediction bit will say “taken”, we need to exit from the loop
- When we re-enter the loop, at the end of the first loop iteration, we need to take the branch to stay in the loop, while the prediction bit says to exit from the loop, since the prediction bit was flipped on the previous execution of the last iteration of the loop

For example, if we consider a loop branch whose behavior is taken nine times and not taken once, the prediction accuracy is only 80% (due to two incorrect predictions and eight correct ones)

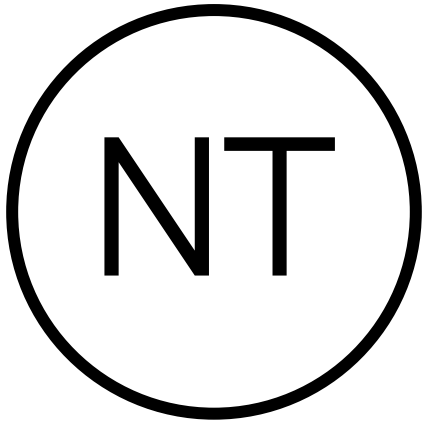
2-bit Branch History Table

The prediction must be wrong twice before it is changed

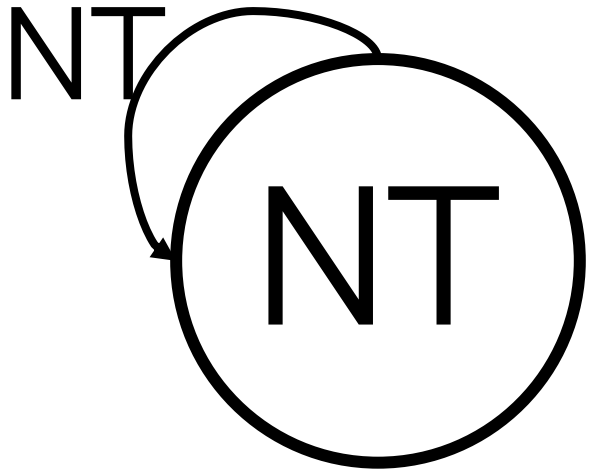
In a loop branch, at the last loop iteration, we do not need to change the prediction

For each index in the table, 2 bits are used to encode the four states of a finite state machine

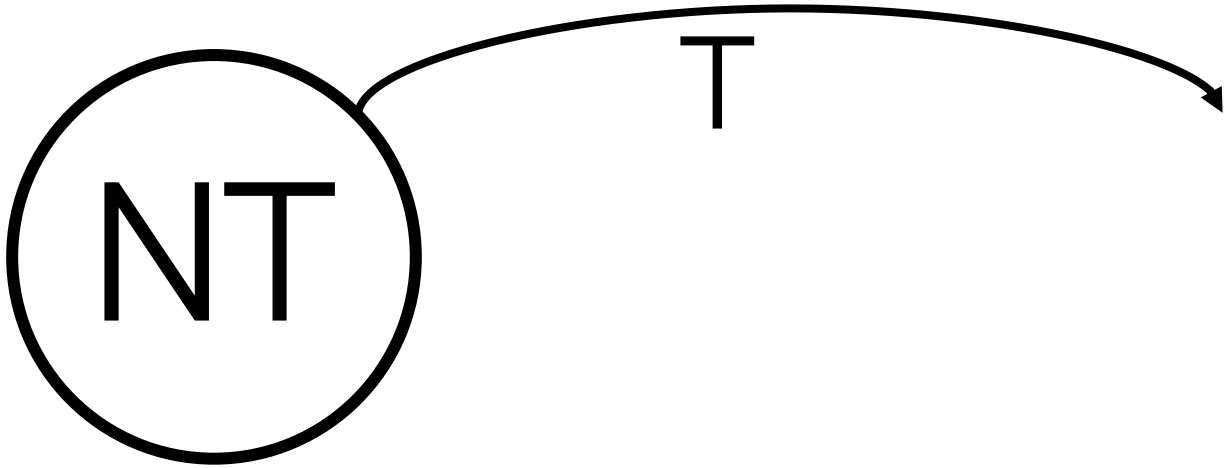
2-bit BHT… aka 2nd chance ;)



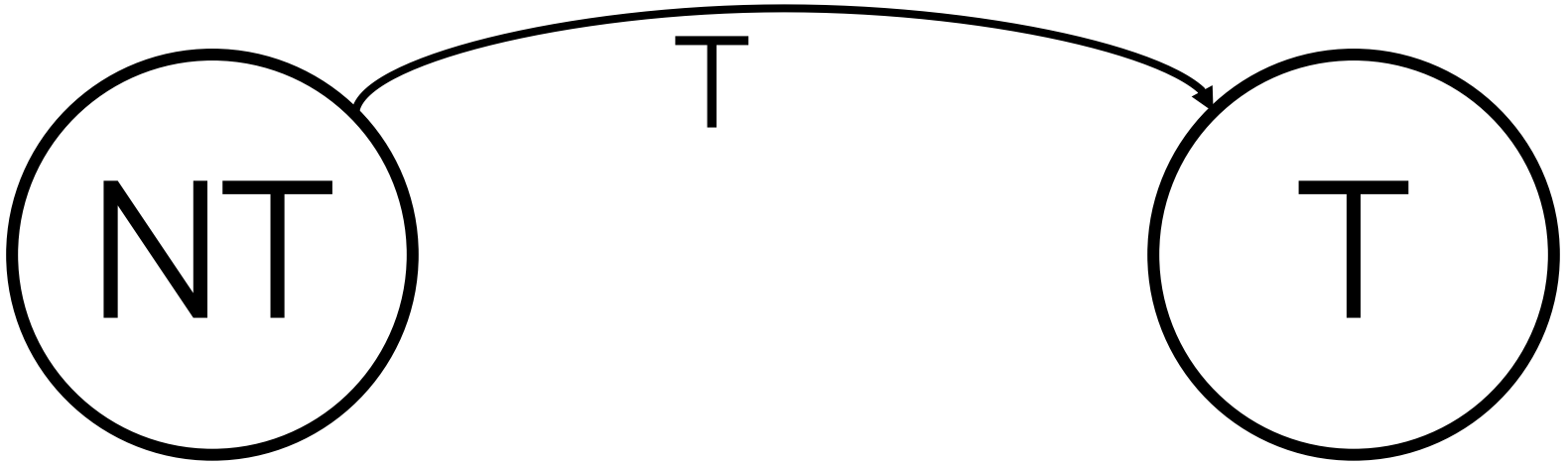
2-bit BHT… aka 2nd chance ;)



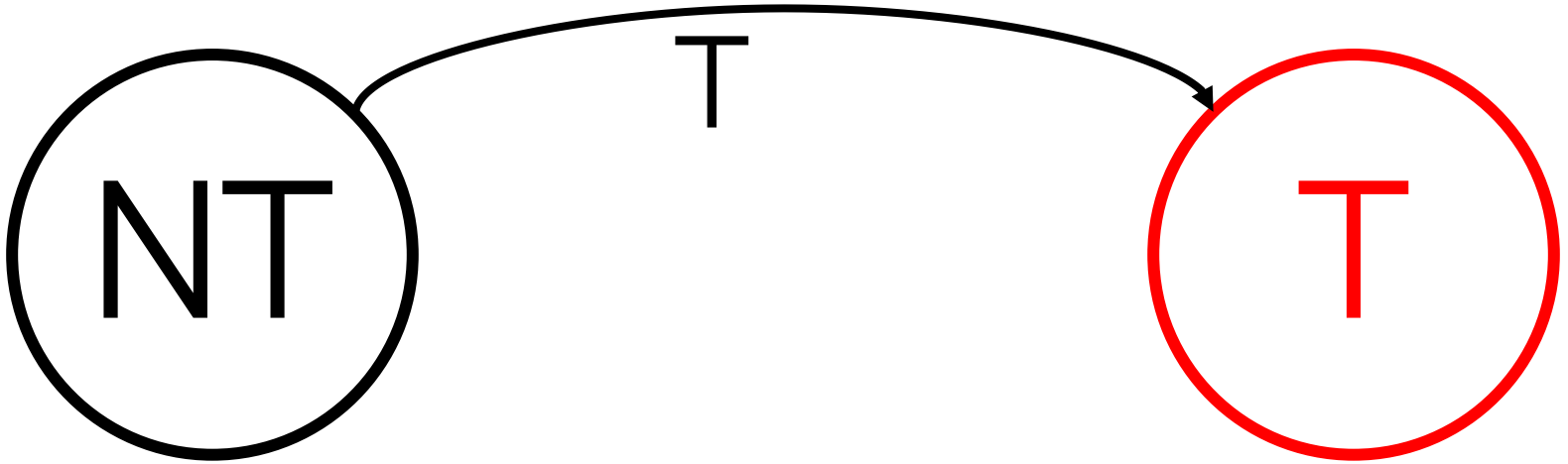
2-bit BHT… aka 2nd chance ;)



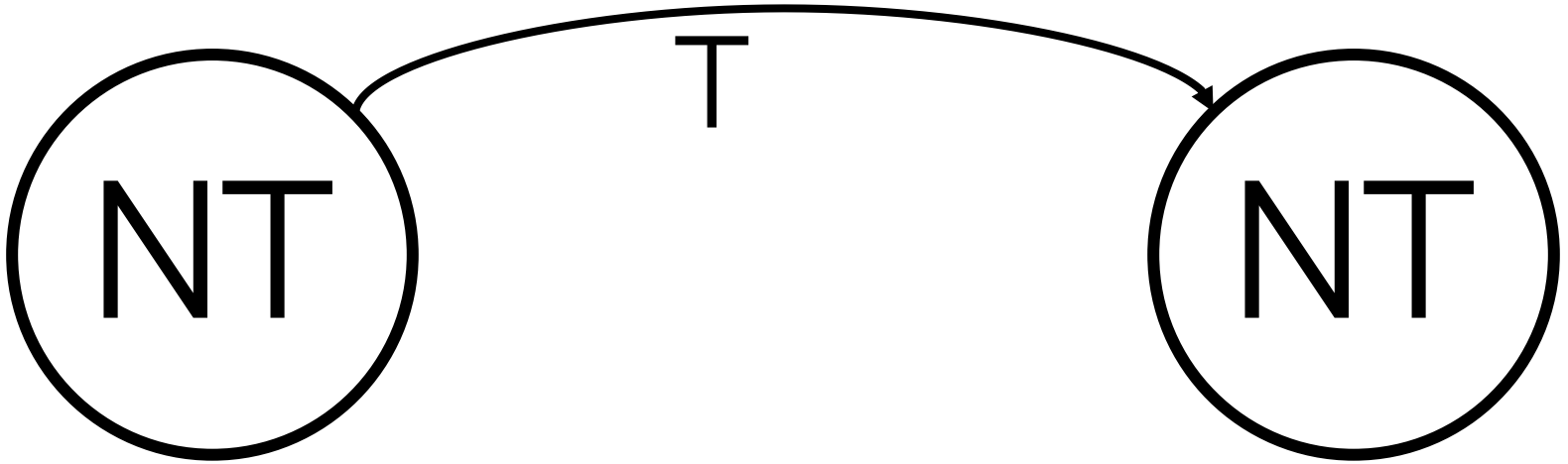
2-bit BHT… aka 2nd chance ;)



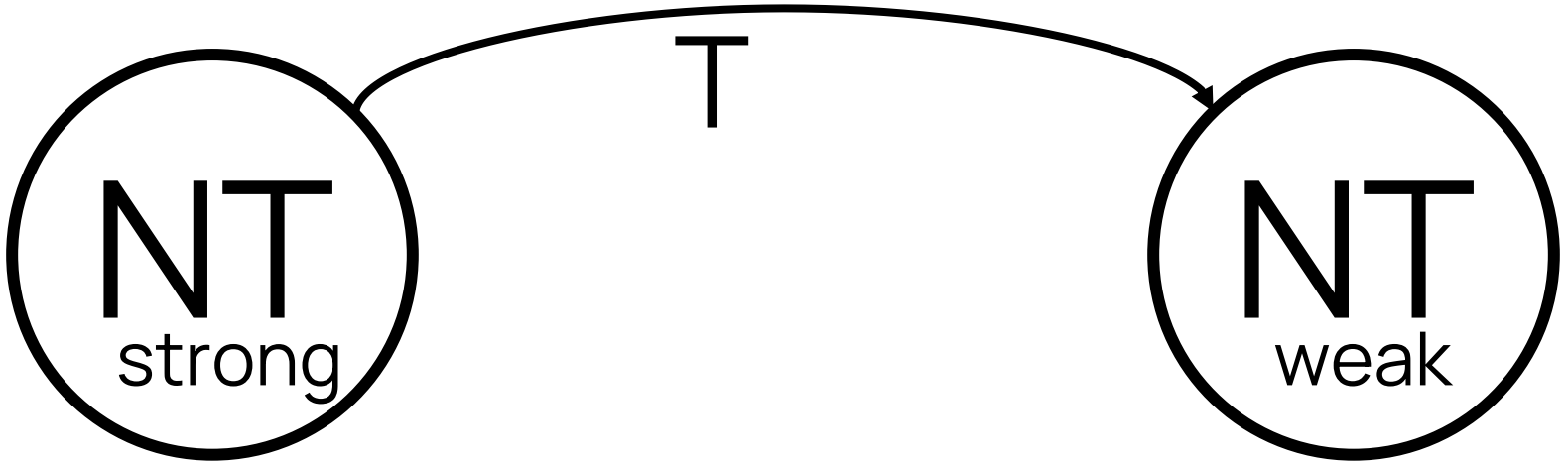
2-bit BHT… aka 2nd chance ;)



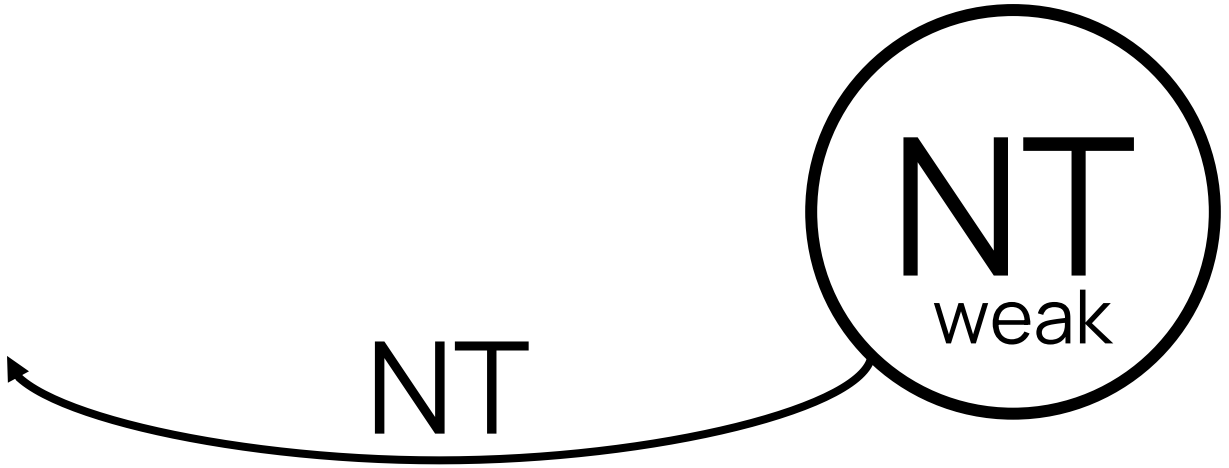
2-bit BHT… aka 2nd chance ;)



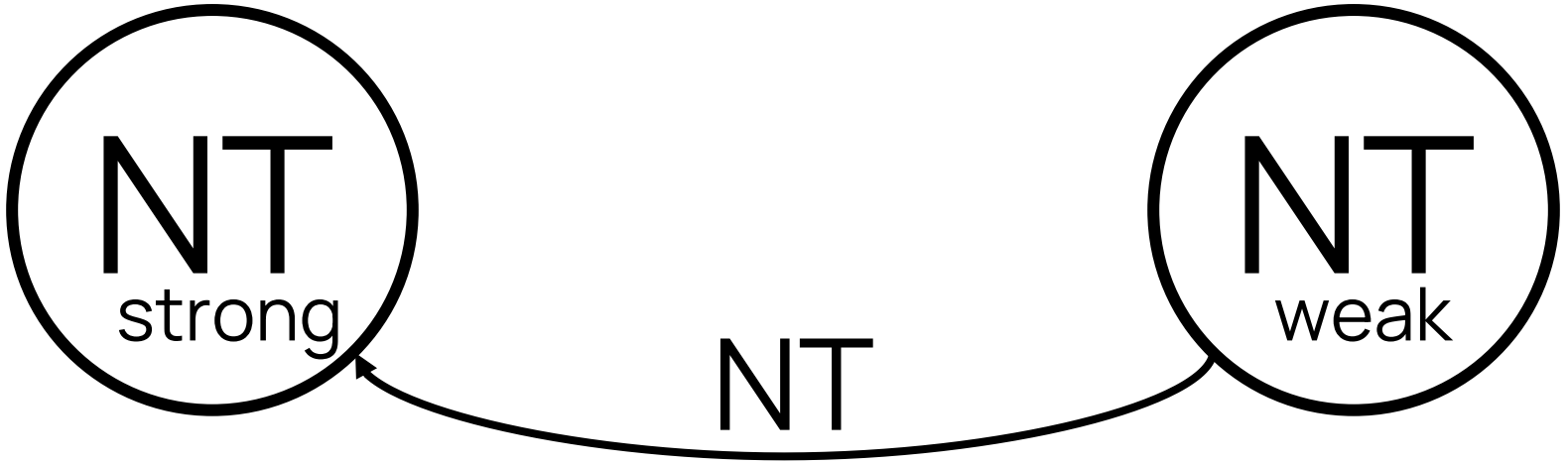
2-bit BHT… aka 2nd chance ;)



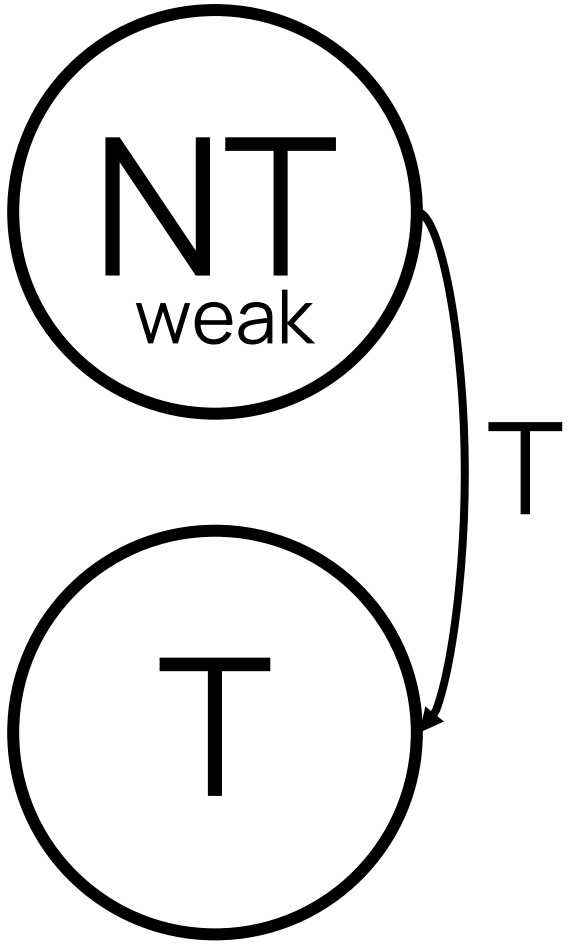
2-bit BHT… aka 2nd chance ;)



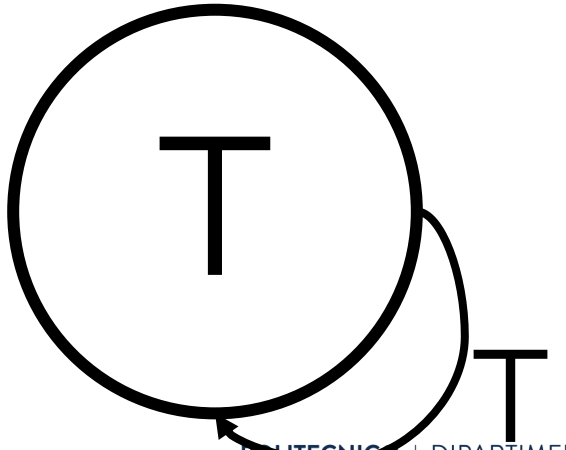
2-bit BHT… aka 2nd chance ;)



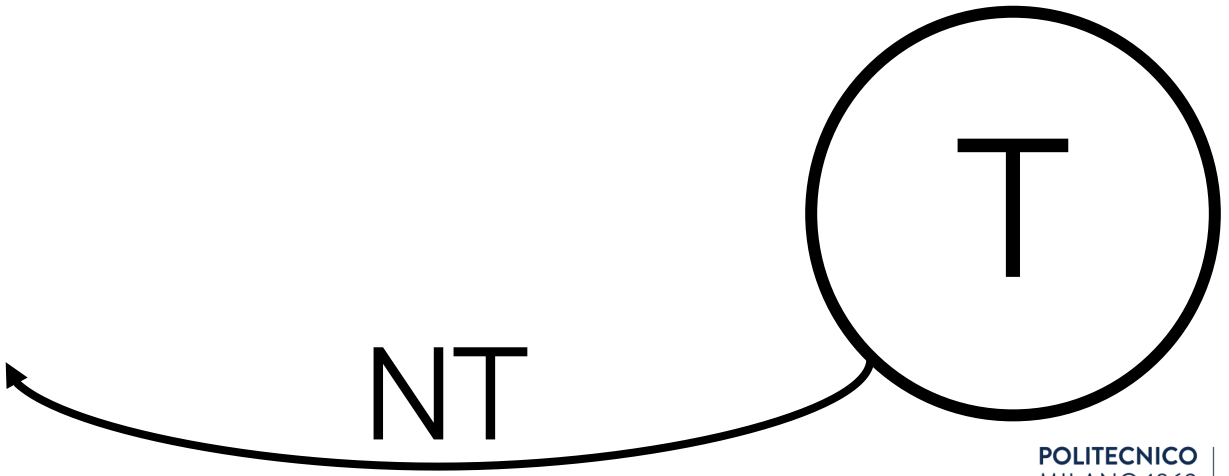
2-bit BHT… aka 2nd chance ;)



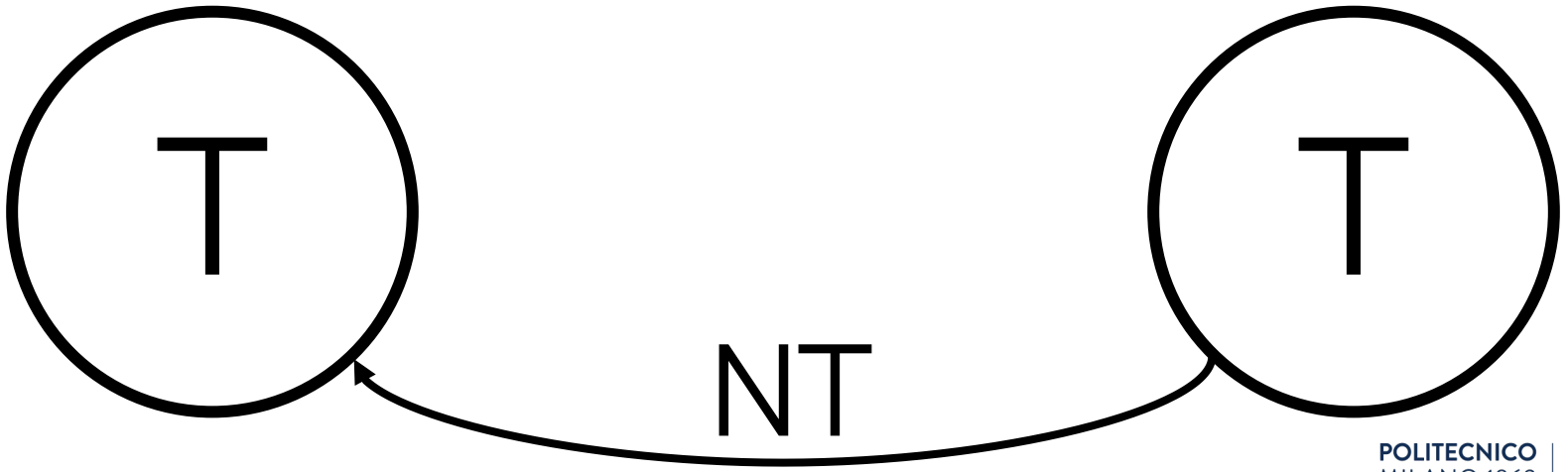
2-bit BHT… aka 2nd chance ;)



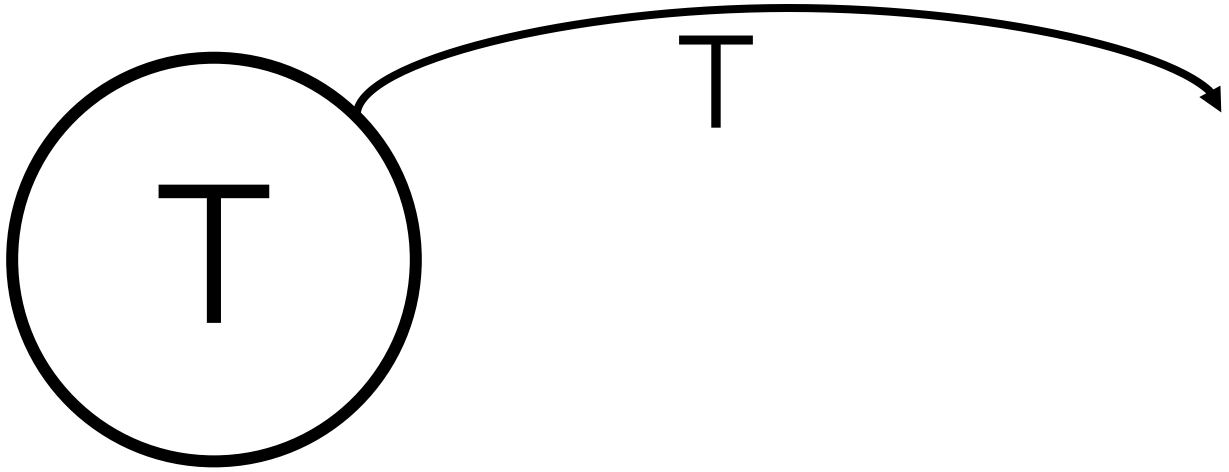
2-bit BHT… aka 2nd chance ;)



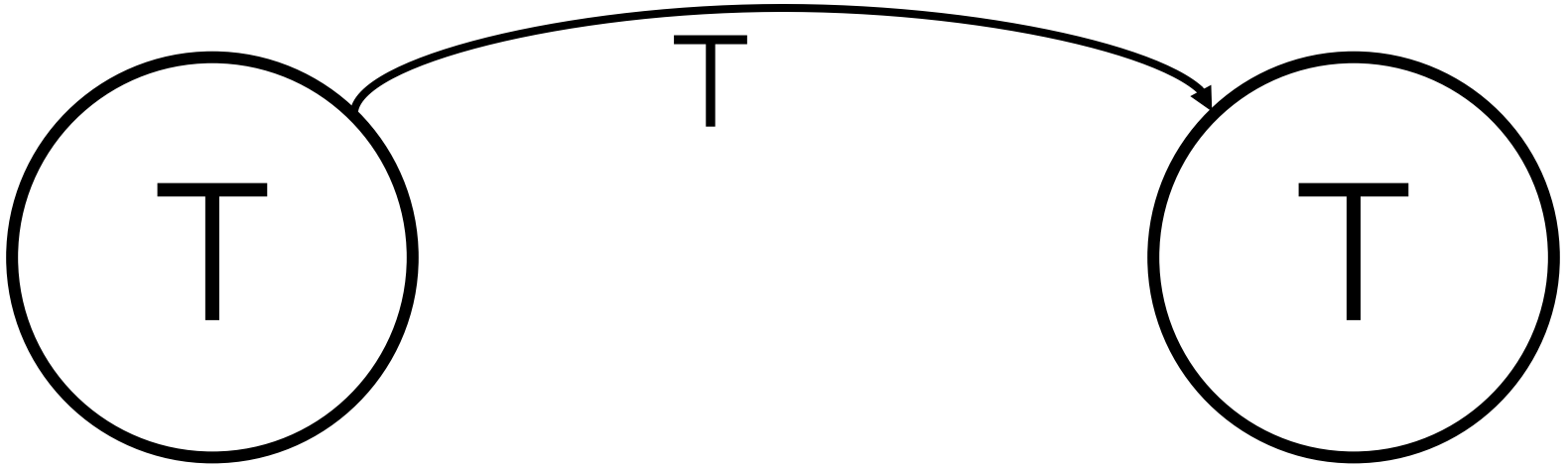
2-bit BHT… aka 2nd chance ;)



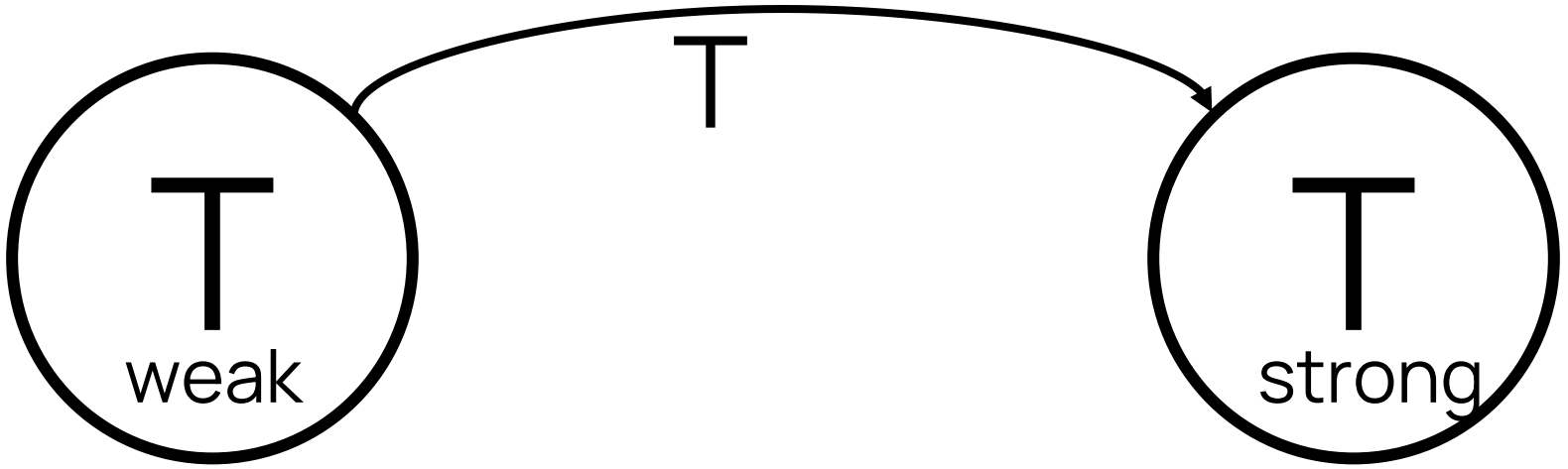
2-bit BHT… aka 2nd chance ;)



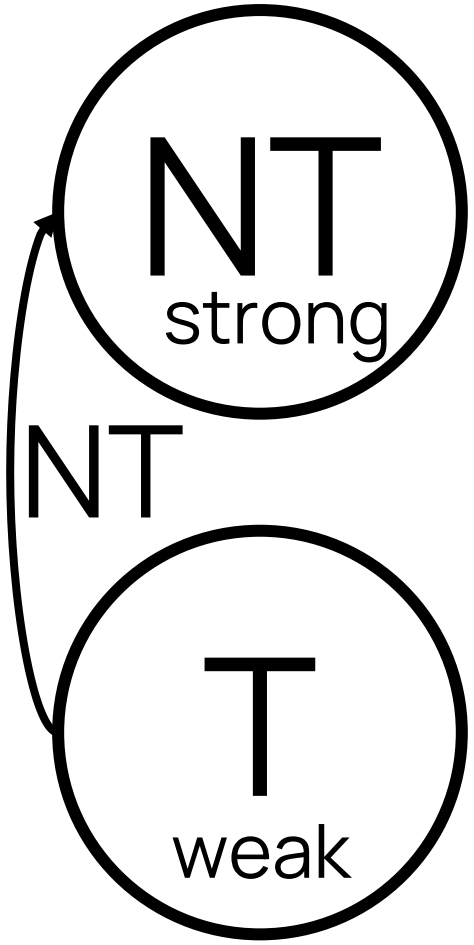
2-bit BHT… aka 2nd chance ;)



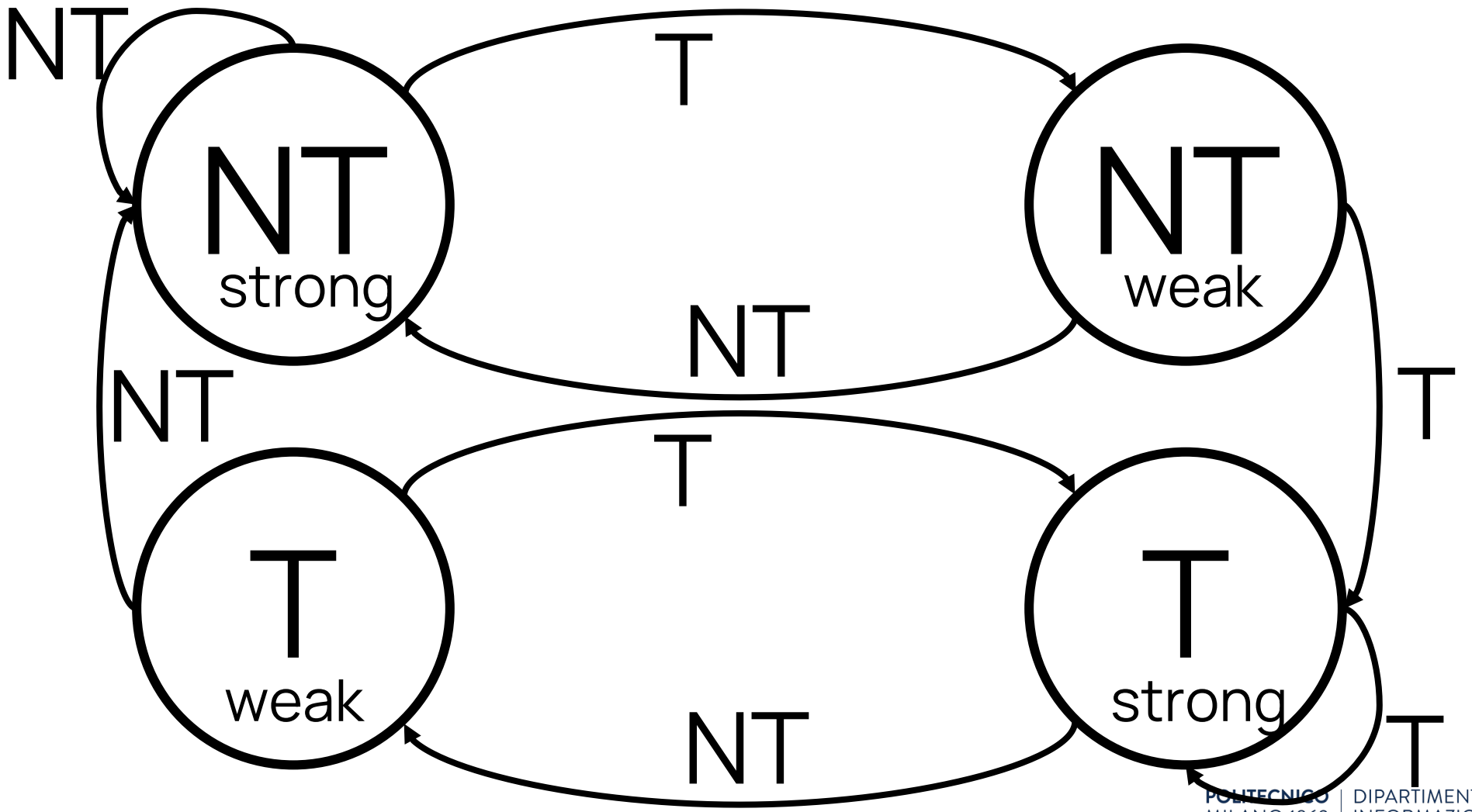
2-bit BHT… aka 2nd chance ;)



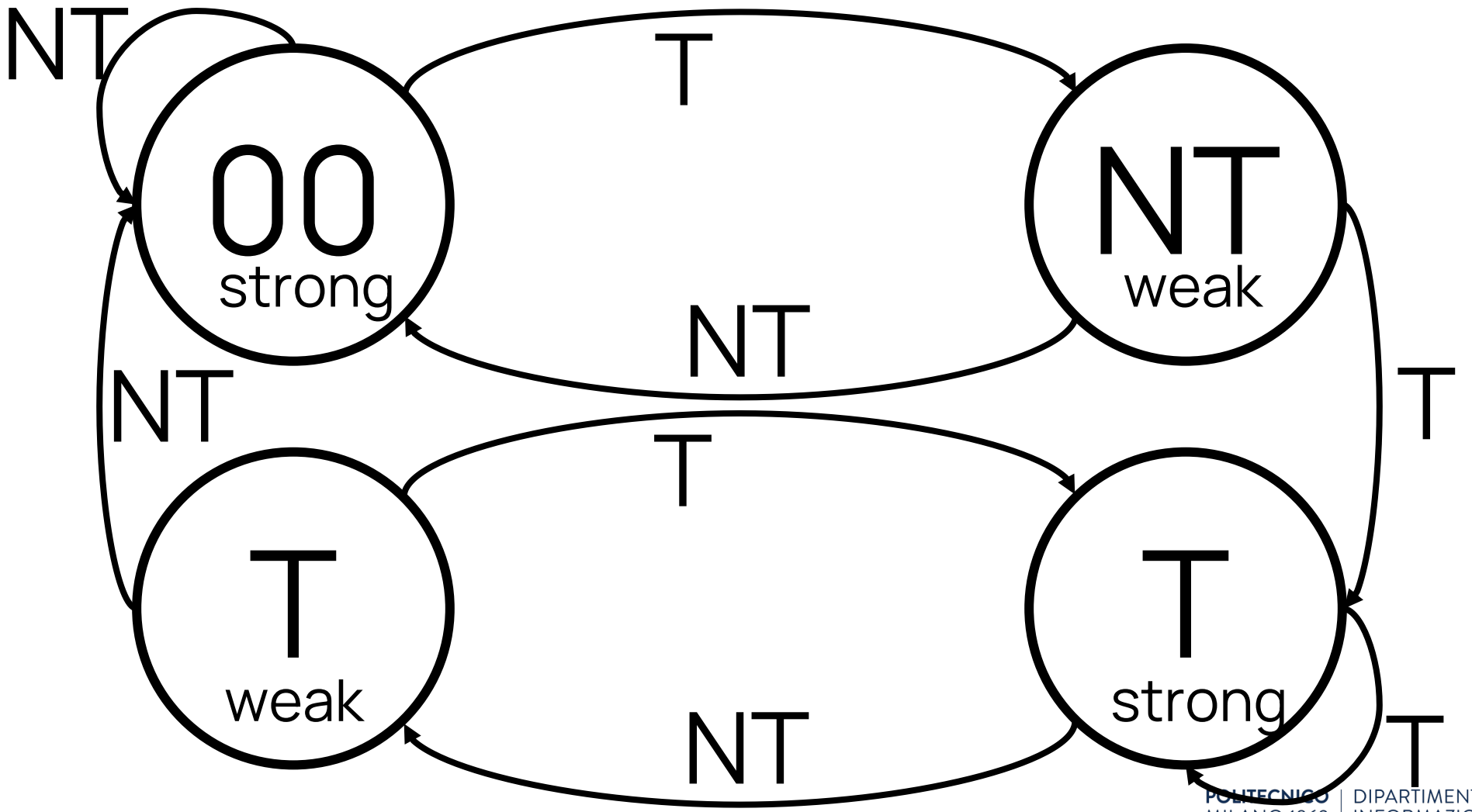
2-bit BHT… aka 2nd chance ;)



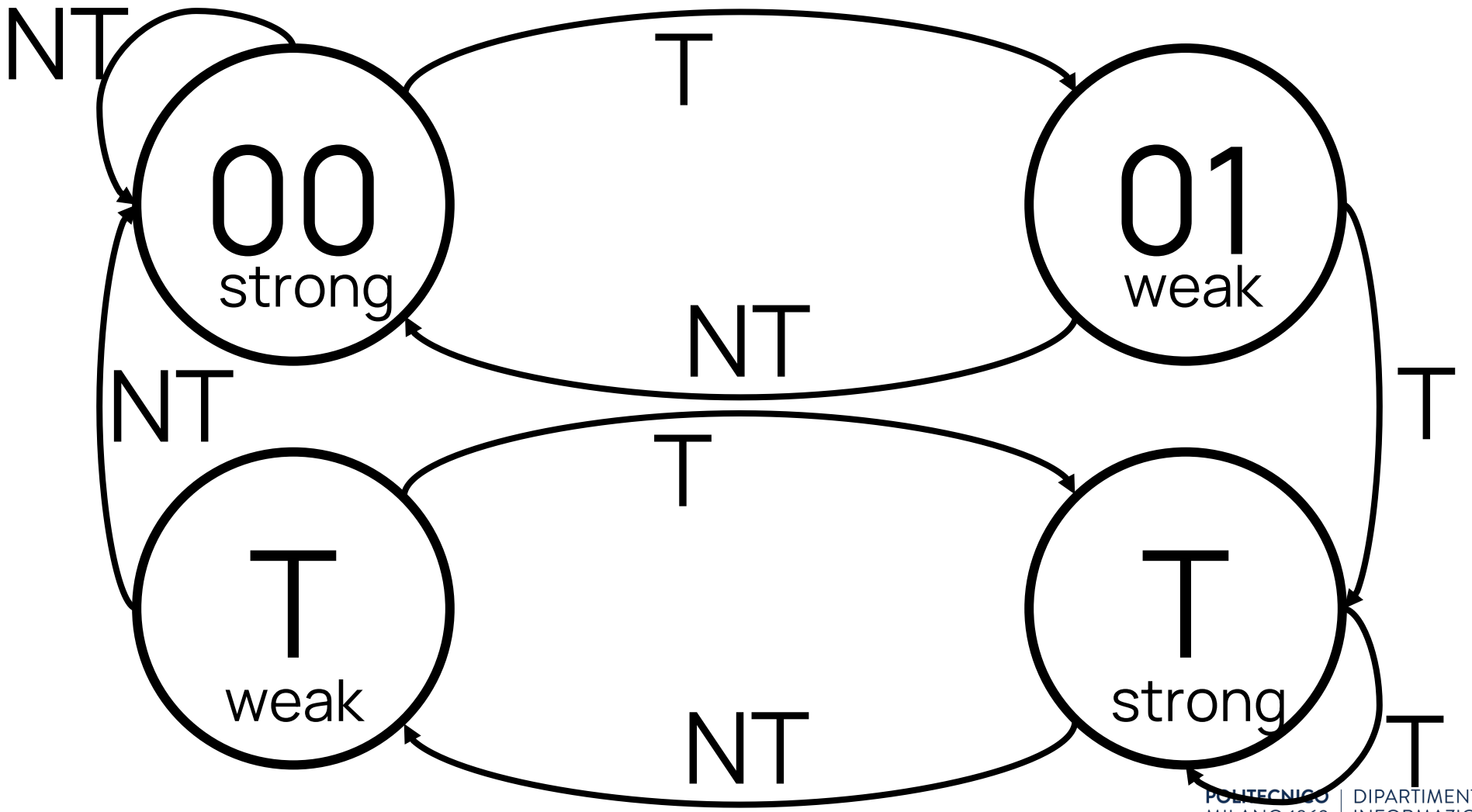
2-bit BHT… all together…



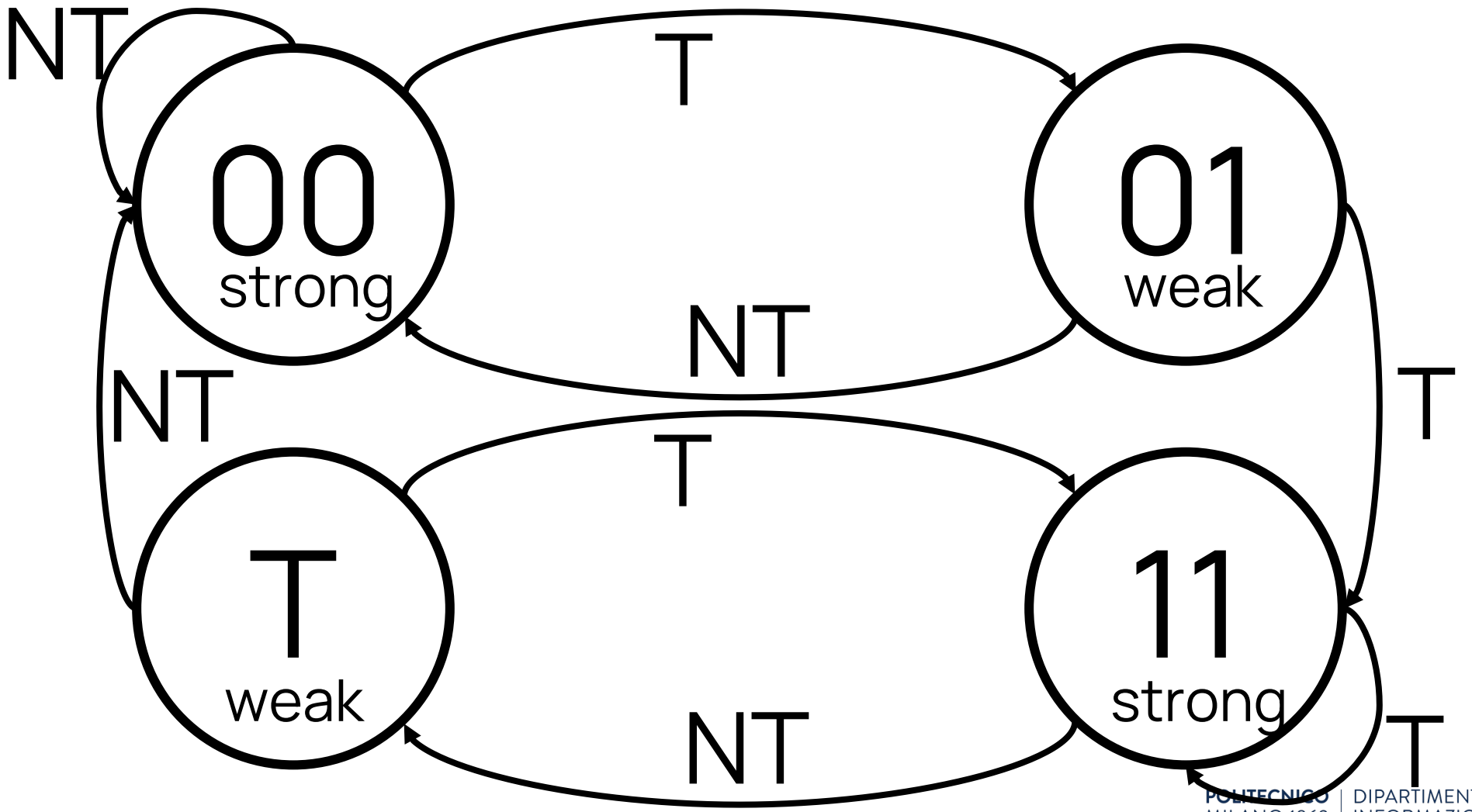
2-bit BHT… all together…



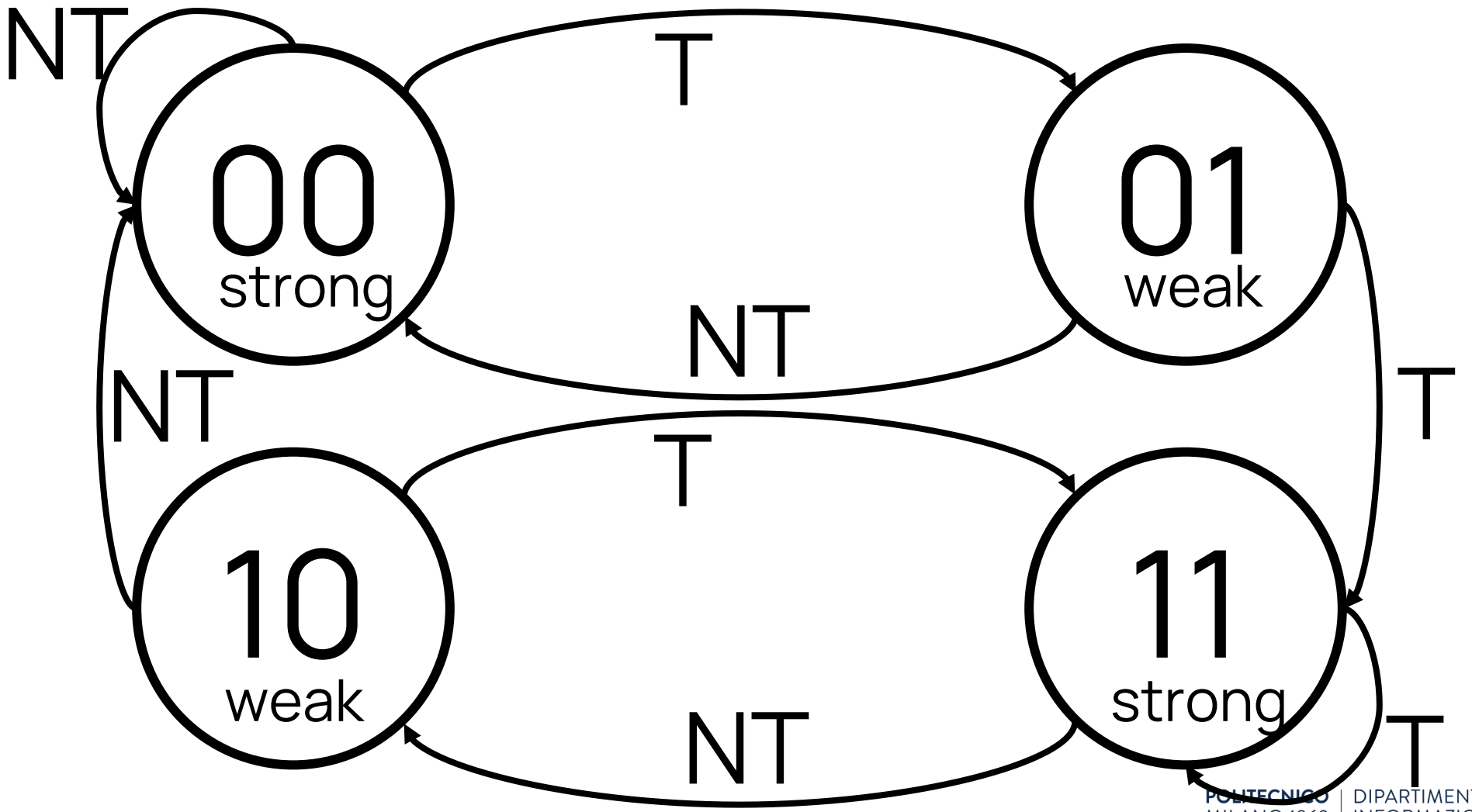
2-bit BHT… all together…



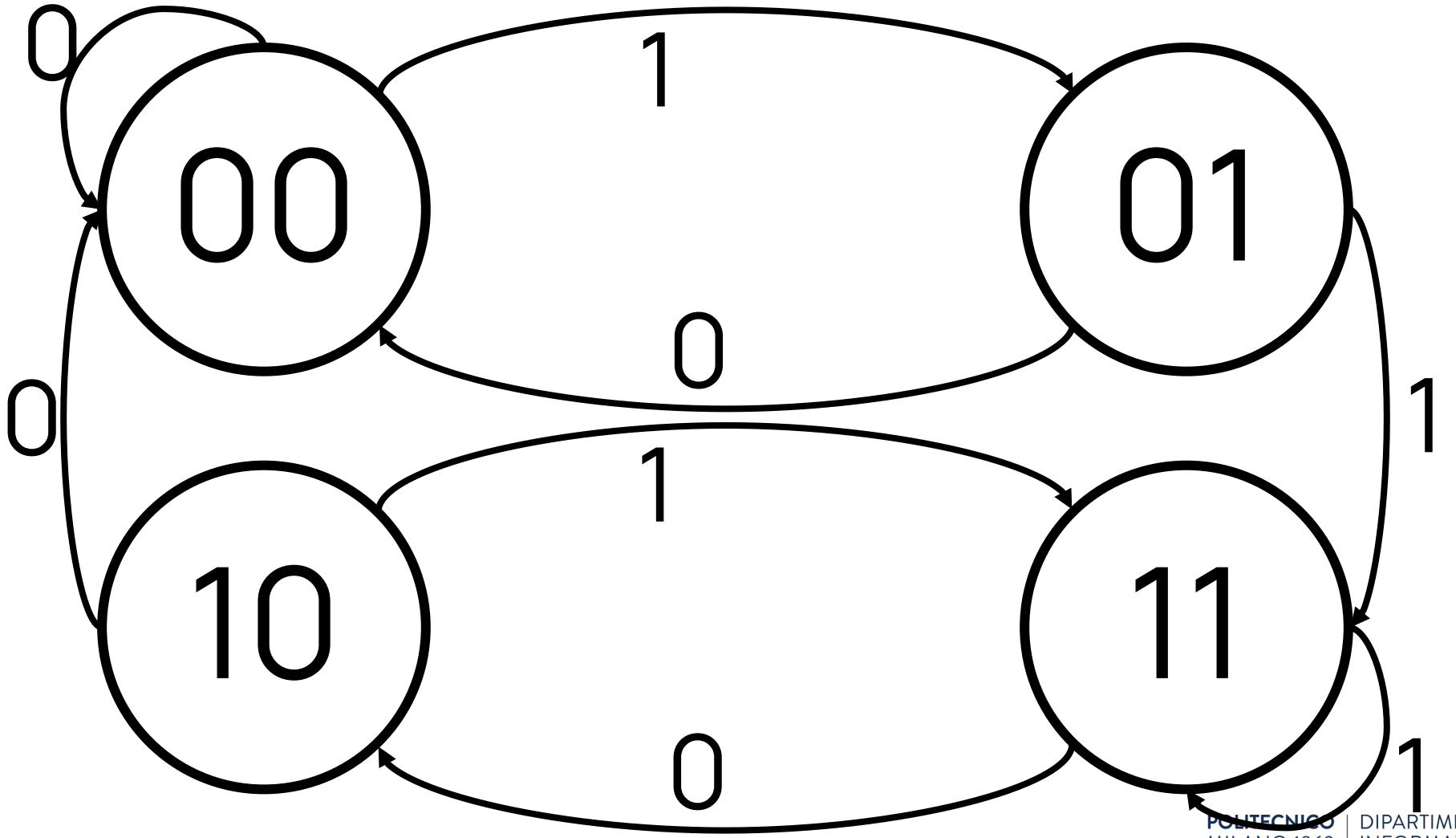
2-bit BHT… all together…



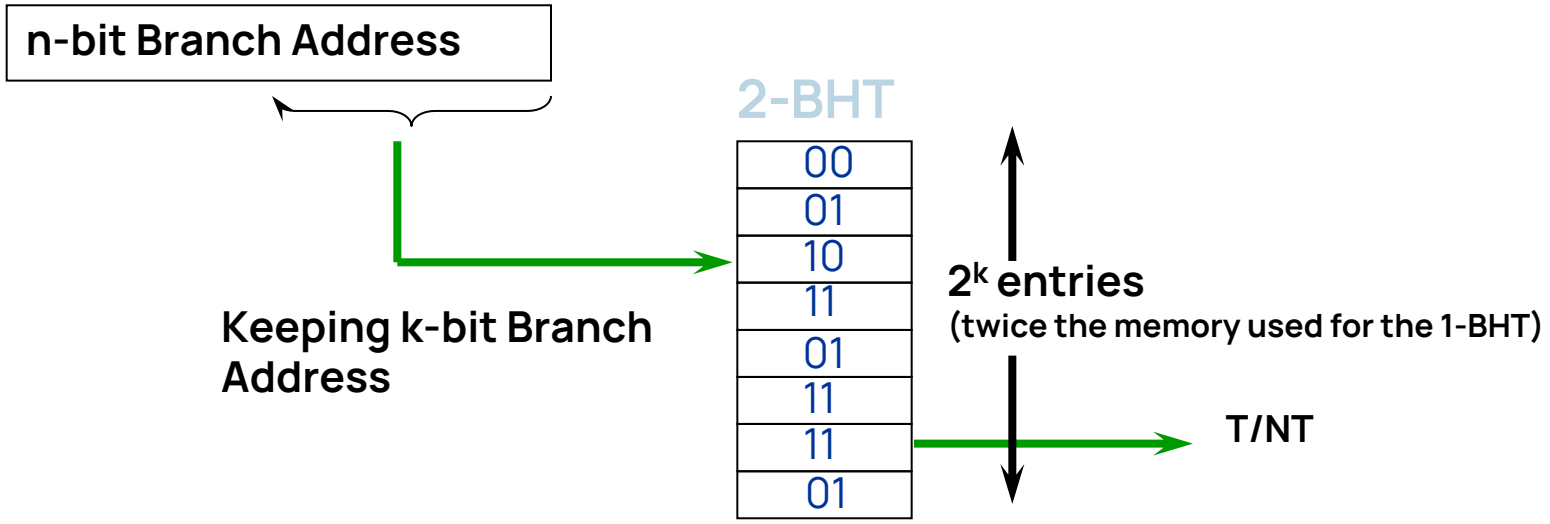
2-bit BHT… all together…



2-bit BHT… all together…



2-bit Branch History Table



n-bit Branch History Table

Generalization: **n-bit saturating counter** for each entry in the prediction buffer

- The counter can take values between 0 and 2^n-1
- When the counter is greater than or equal to one-half of its maximum value (2^n-1), the branch is predicted as taken
- Otherwise, it is predicted as untaken

As in the 2-bit scheme, the counter is **incremented on a taken branch** and **decremented on an untaken branch**

Studies on n-bit predictors have shown that 2-bit predictors behave almost as well

Accuracy of 2-bit Branch History Table

For IBM Power architecture executing SPEC89 benchmarks, a 4K-entry BHT with 2-bit per entry results in:

- Prediction accuracy from 99% to 82% (i.e., misprediction rate from 1% to 18%)
- Almost similar performance with respect to an infinite buffer with 2-bit per entry

Correlating Branch Predictors

BHT predictors use only the recent behavior of a single branch to predict the future behavior of that branch

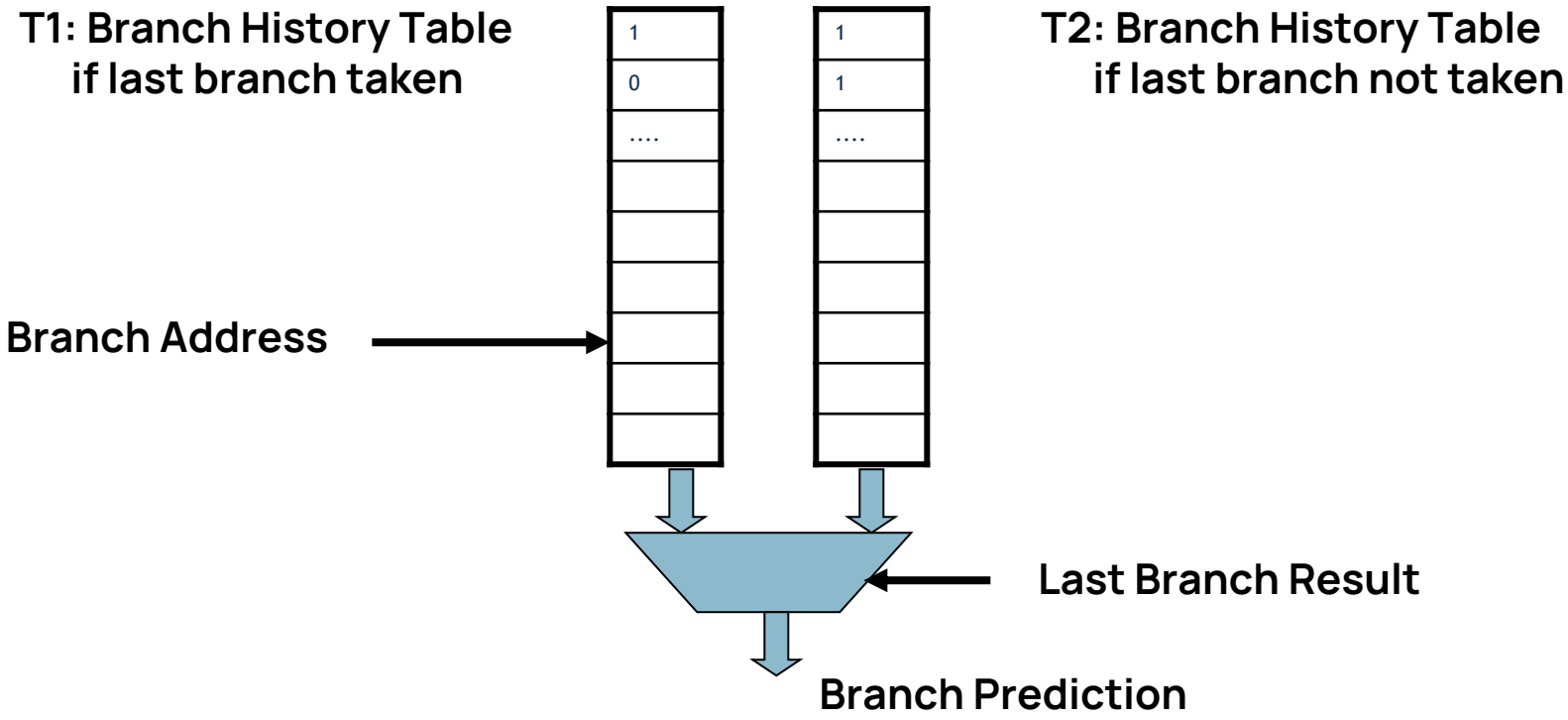
Basic Idea: the behavior of recent branches is correlated $\Rightarrow$ the recent behavior of other branches (rather than just the current branch we are trying to predict) can influence the prediction of the current branch

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 the last branch is used to choose among a pair of 1-bit branch predictors

Correlating Branch Predictors: Example



(m, n) Correlating Branch Predictors

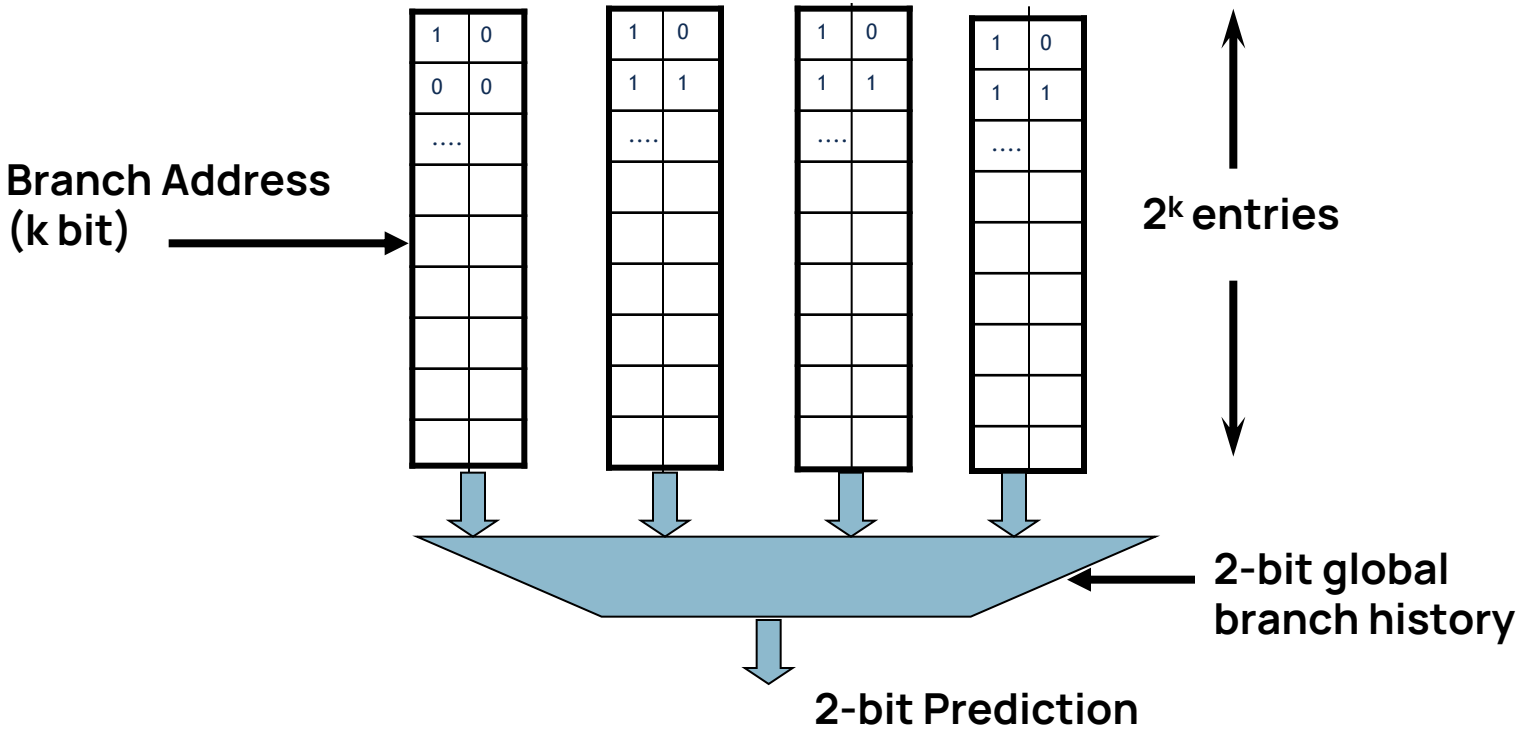
Generally, ***(m, n)* correlating predictor** records the last *m* branches to choose from *2m* BHTs, each of which is an *n*-bit predictor

The branch prediction buffer can be indexed 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)

(2, 2) Correlating Branch Predictors

A **(2, 2) correlating predictor** has four 2-bit Branch History Tables

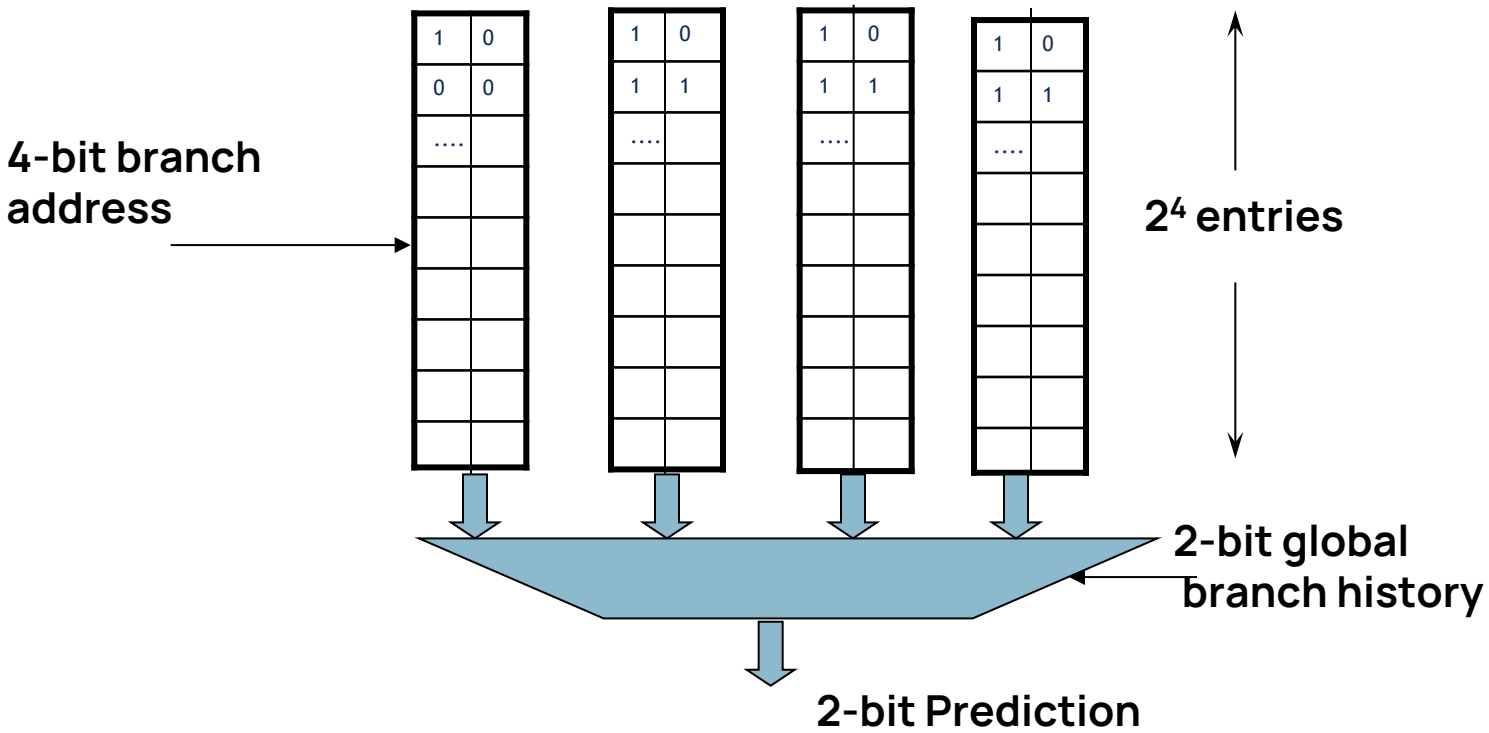
- It uses the 2-bit global history to choose among the four BHTs



(2, 2) Correlating Branch Predictors: example

Example: a **(2, 2) correlating predictor** with 64 total entries ⇒ 6-bit index composed of:

- 2-bit global history
- 4-bit low-order branch address bits



(2, 2) Correlating Branch Predictors: example

Each BHT is composed of 16 entries of 2-bit each

- The 4-bit branch address is used to choose four entries (a row)
- 2-bit global history is used to choose one of four entries in a row (one of four BHTs)

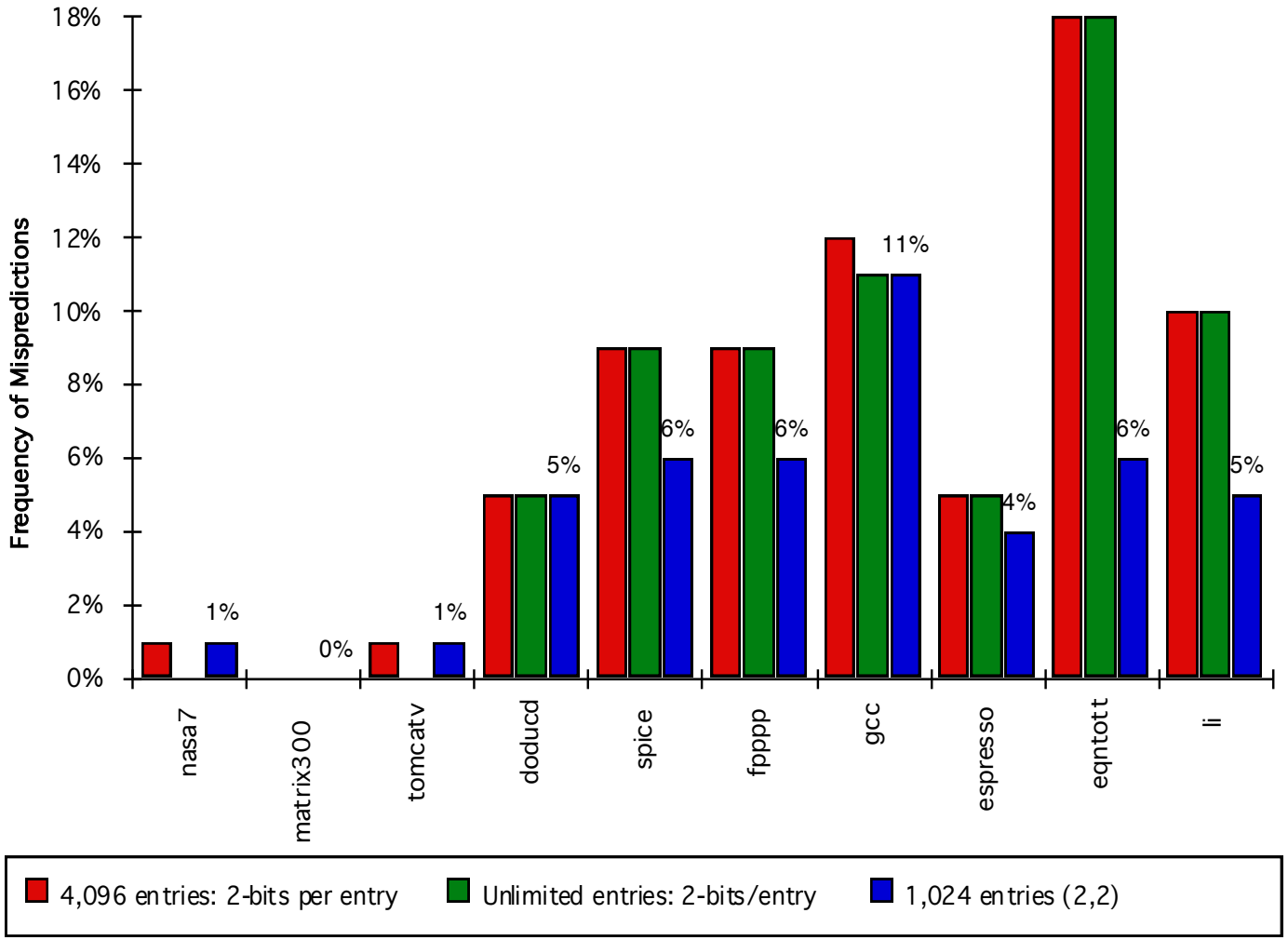
Accuracy of Correlating Predictors

A **2-bit predictor** with no global history is simply a **(0, 2) predictor**

By comparing the **performance** of a 2-bit simple predictor with 4K entries and a (2,2) correlating predictor with 1K entries

- The (2,2) predictor not only outperforms the simple 2-bit predictor with the same number of total bits (4K total bits), but it often outperforms a 2-bit predictor with an unlimited number of entries

Accuracy of Correlating Predictors



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

The **second-level history** is recorded in one (or more) tables called *Pattern History Table* (PHT) of two-bit saturating counters

The BHR is used to index the PHT and 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 two-bit counter scheme

Questions?