Advanced Computer Architectures

(High Performance Processors and Systems)

Dynamic Branch Prediction

Politecnico di Milano

v1

Alessandro Verosimile <alessandro.verosimile@polimi.it>
Marco D. Santambrogio <marco.santambrogio@polimi.it>

Outline

- Branch Prediction Techniques
  - Dynamic Branch Prediction
- Performance of Branch Schemes

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 causing the branch prediction can change during the program execution.
- In both cases, care must be taken not to change the processor state until the branch is definitely known.

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:
    - To predict the direction of a branch (i.e. taken or not taken).
  - Branch Target Predictor:
    - To predict the branch target address in case of taken branch.

- These modules are used by the Instruction Fetch Unit to predict the next instruction to read in the I-cache.
  - If branch is not taken ⇒ PC is incremented.
  - If branch is taken ⇒ BTP gives the target address

Dynamic Branch Prediction Schemes

- Dynamic branch prediction is based on two interacting mechanisms:
  - Branch Outcome Predictor:
    - To predict the direction of a branch (i.e. taken or not taken).
  - Branch Target Predictor:
    - To predict the branch target address in case of taken branch.

- These modules are used by the Instruction Fetch Unit to predict the next instruction to read in the I-cache.
  - If branch is not taken $\Rightarrow$ PC is incremented.
  - If branch is taken $\Rightarrow$ BTP gives the target address

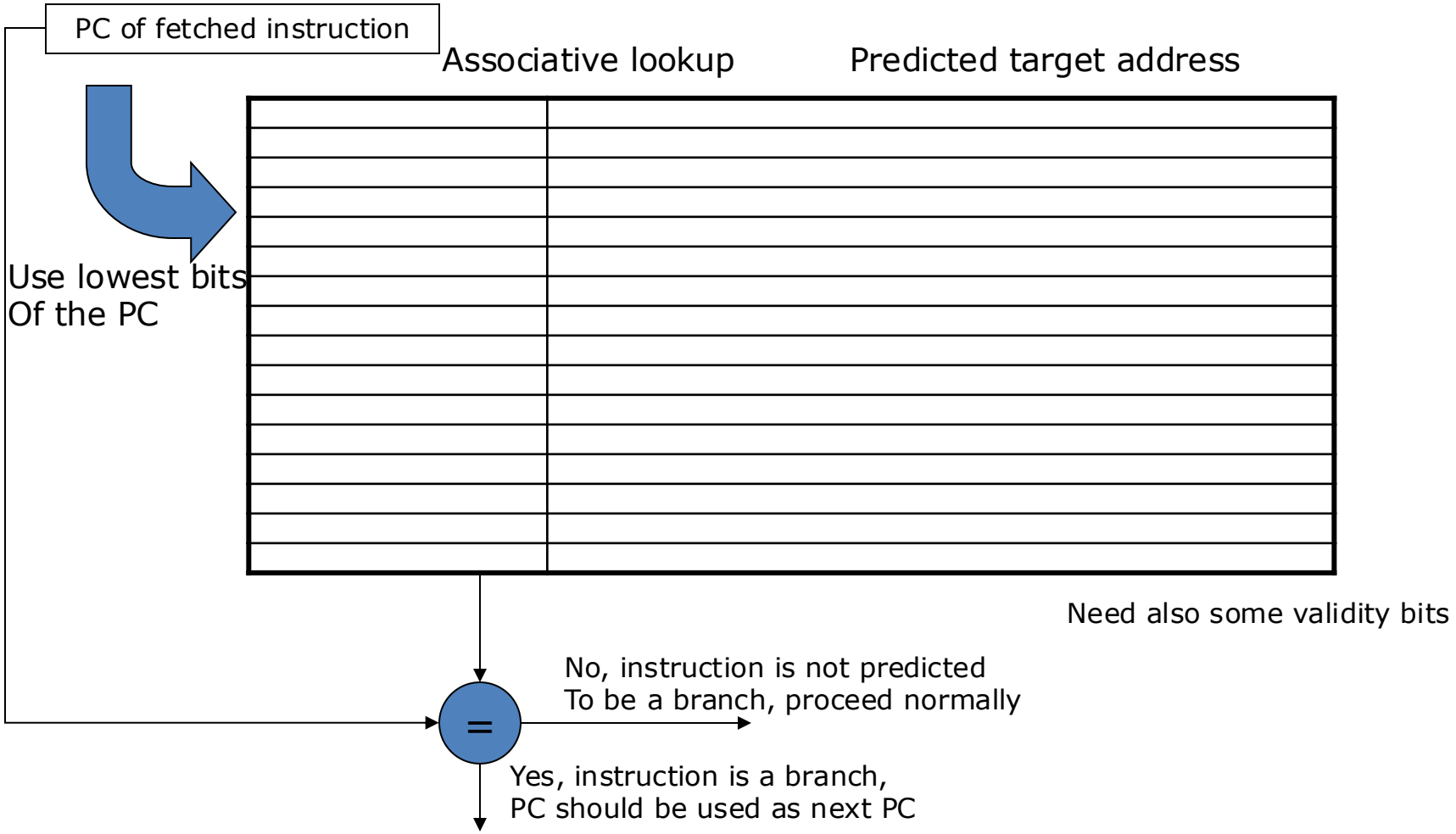
Branch Target Buffer

- 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



Dynamic Branch Prediction Schemes

- Dynamic branch prediction is based on two interacting mechanisms:
  - Branch Outcome Predictor:
    - To predict the direction of a branch (i.e. taken or not taken).
  - Branch Target Predictor:
    - To predict the branch target address in case of taken branch.

- These modules are used by the Instruction Fetch Unit to predict the next instruction to read in the I-cache.
  - If branch is not taken $\Rightarrow$ PC is incremented.
  - If branch is taken $\Rightarrow$ BTP gives the target address

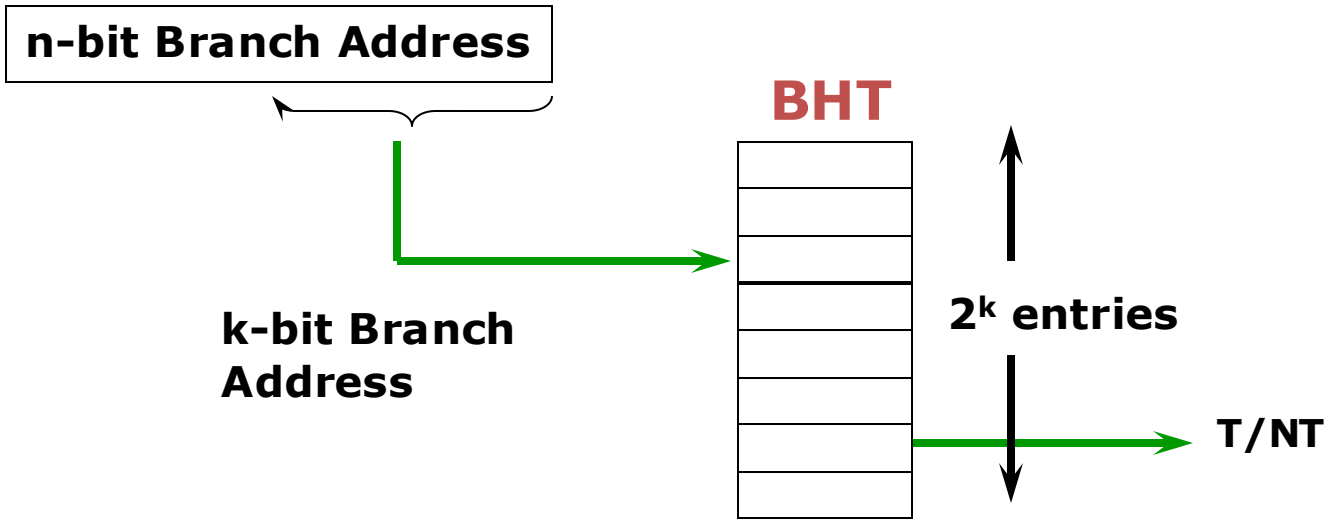
Branch History Table

- 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.

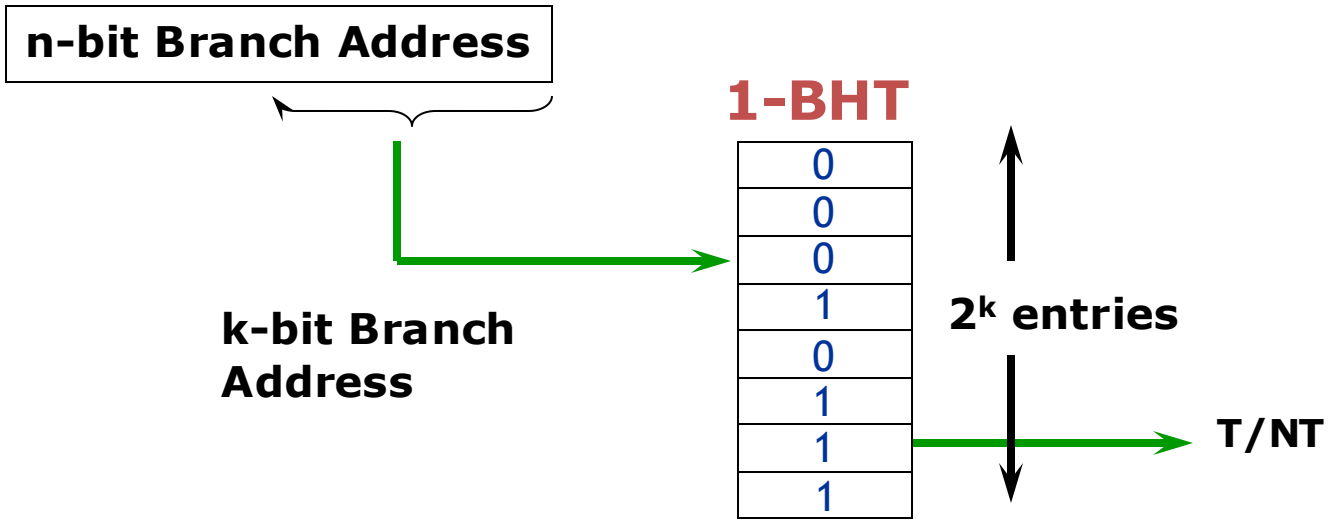
- Prediction: hint that it is assumed to be correct, and fetching begins in the predicted direction.
  - If the hint turns out to be wrong, the prediction bit is inverted and stored back. The pipeline is flushed and the correct sequence is executed.

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

Branch History Table

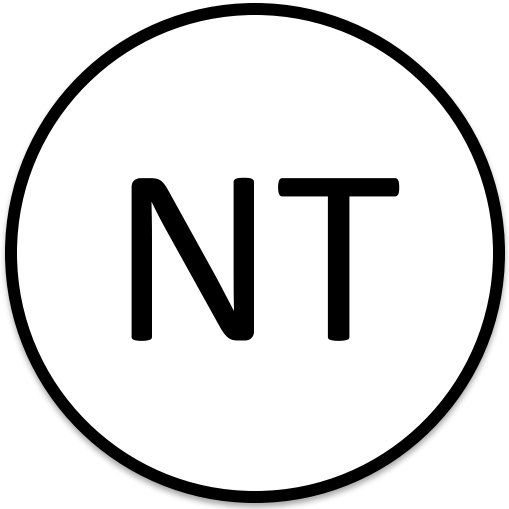


1-bit Branch History Table



«BHT» behavior

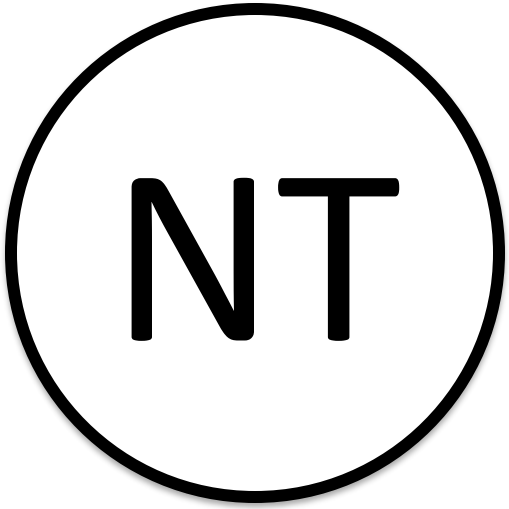
Prediction: NT



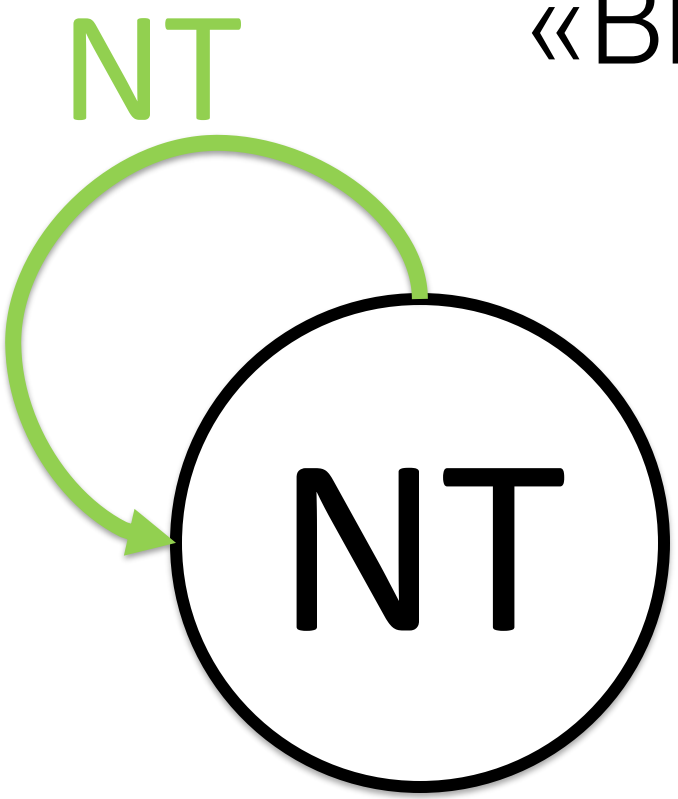
«BHT» behavior

Prediction: NT

Outcome: NT



«BHT» behavior

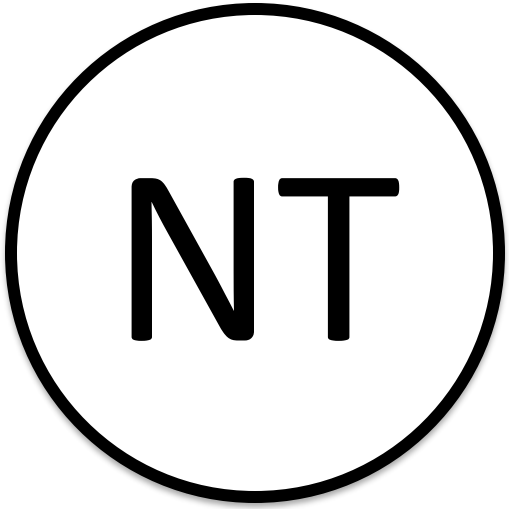


Prediction: NT

Outcome: NT

«BHT» behavior

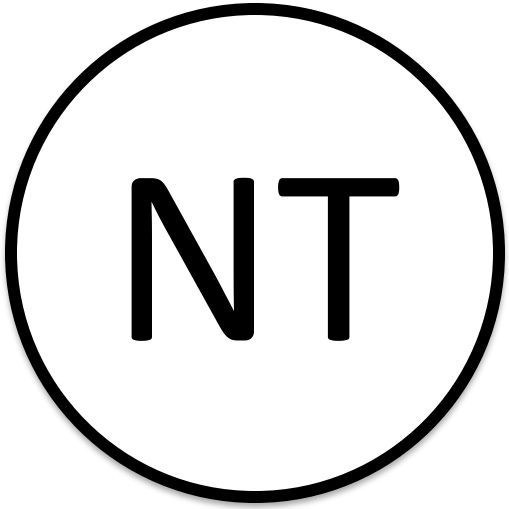
Prediction: NT



«BHT» behavior

Prediction: NT

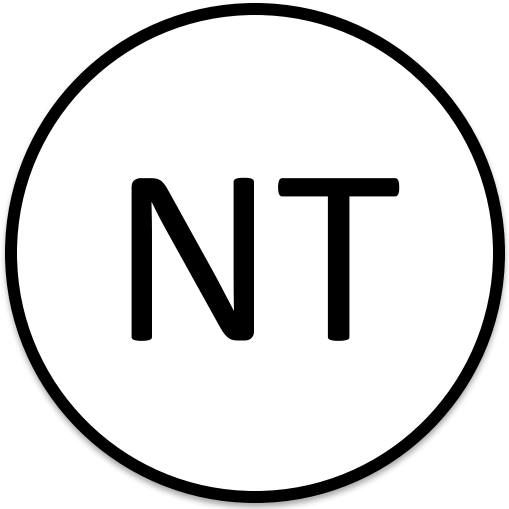
Outcome: T



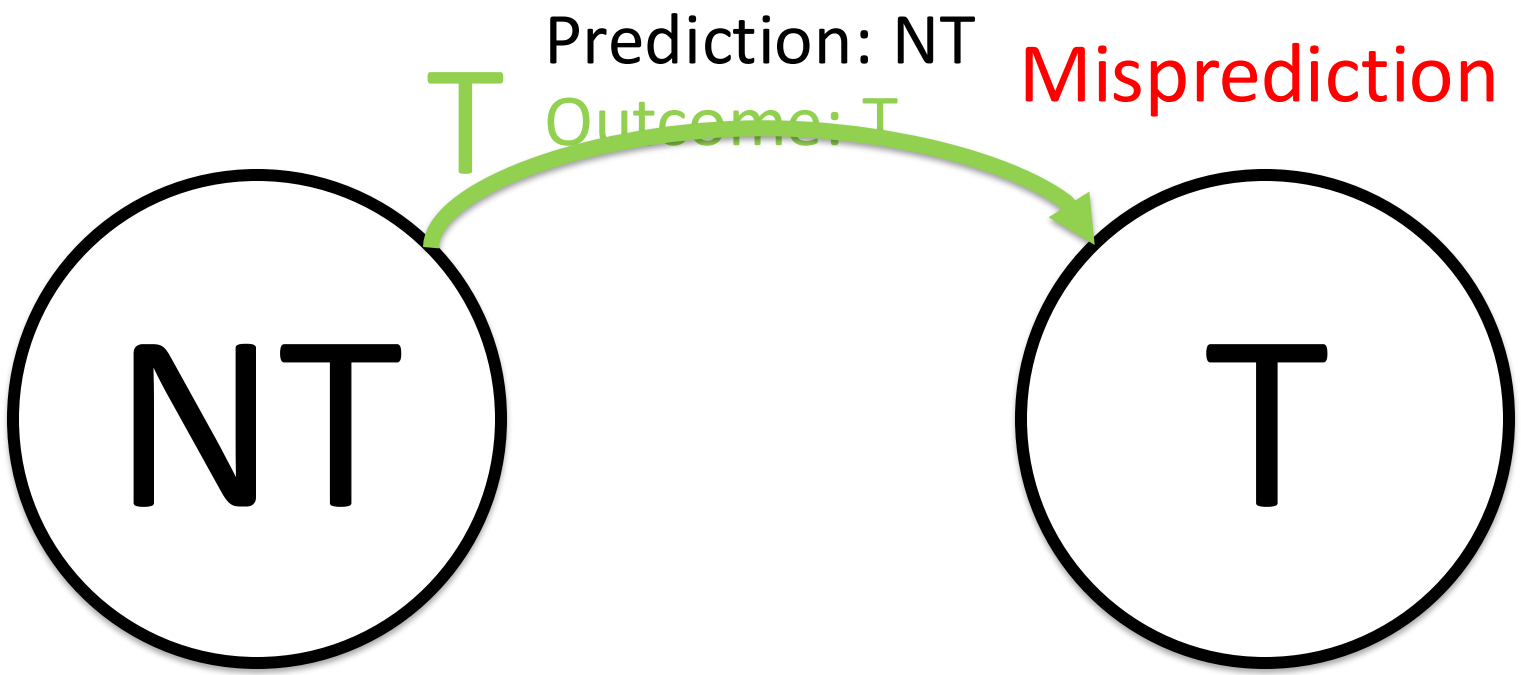
«BHT» behavior

Prediction: NT
Outcome: T

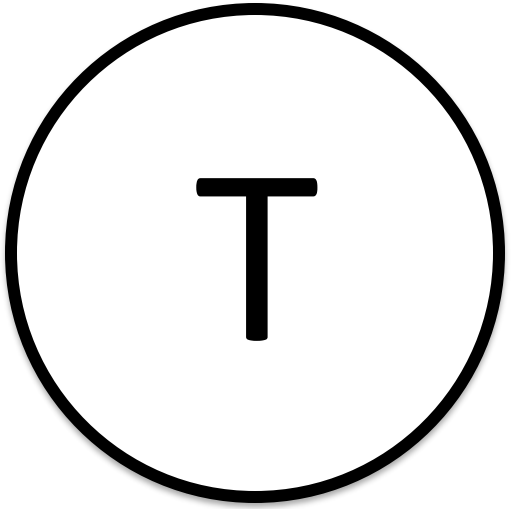
Misprediction



«BHT» behavior

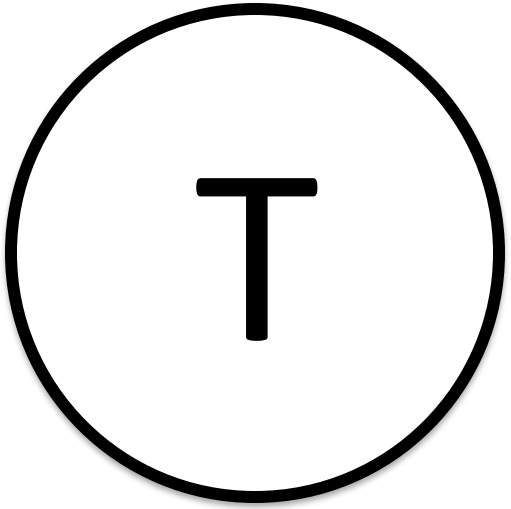


«BHT» behavior



«BHT» behavior

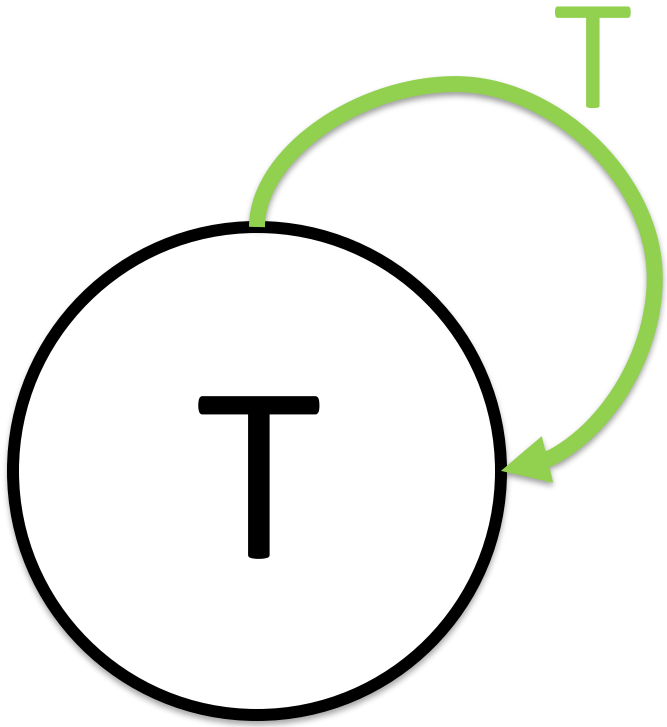
Prediction: T



«BHT» behavior

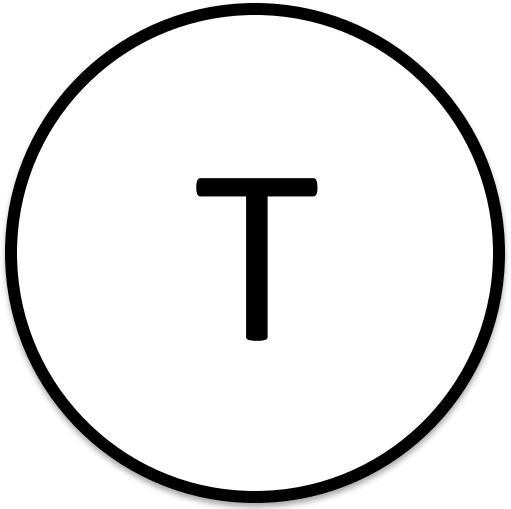
Prediction: T

Outcome: T



«BHT» behavior

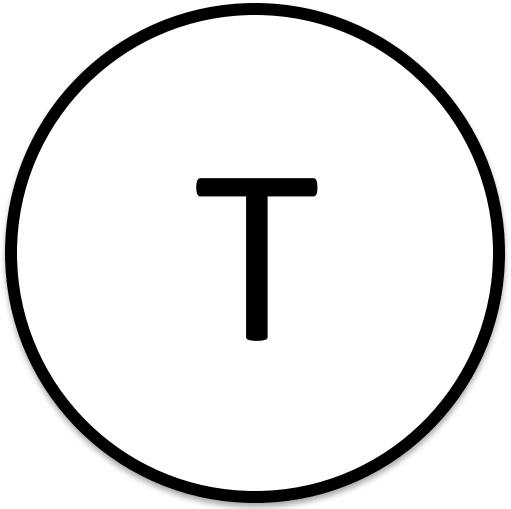
Prediction: T



«BHT» behavior

Prediction: T

Outcome: NT

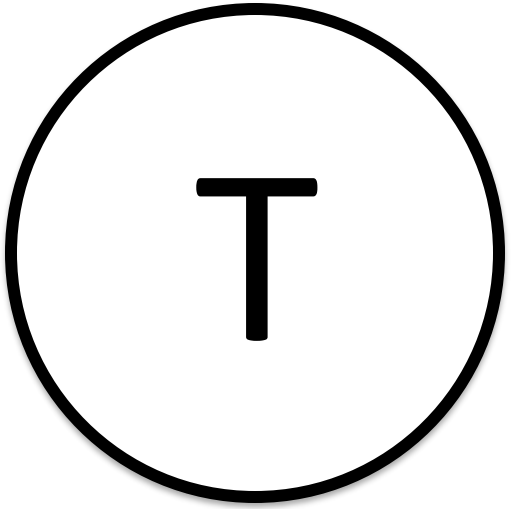


«BHT» behavior

Prediction: T

Outcome: NT

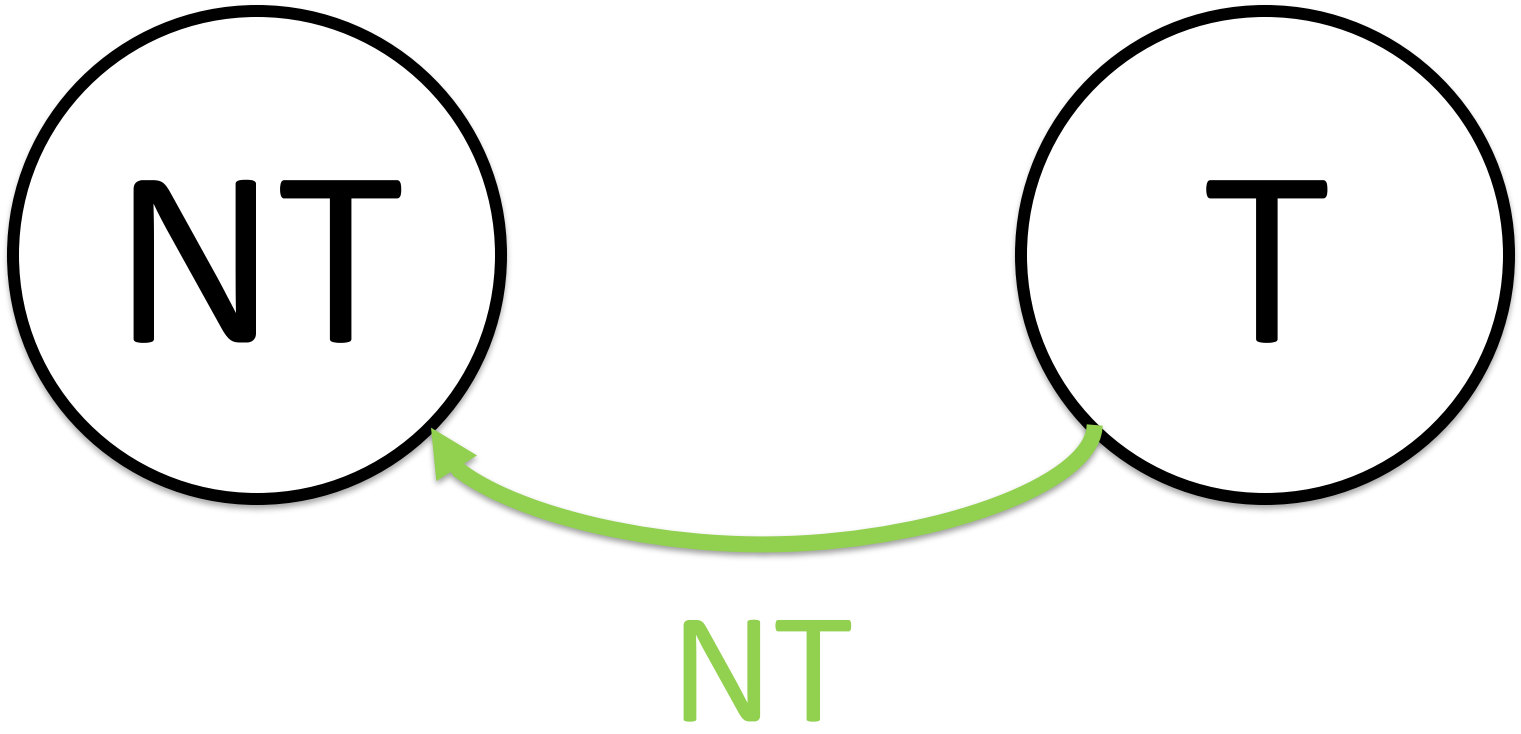
Misprediction



«BHT» behavior

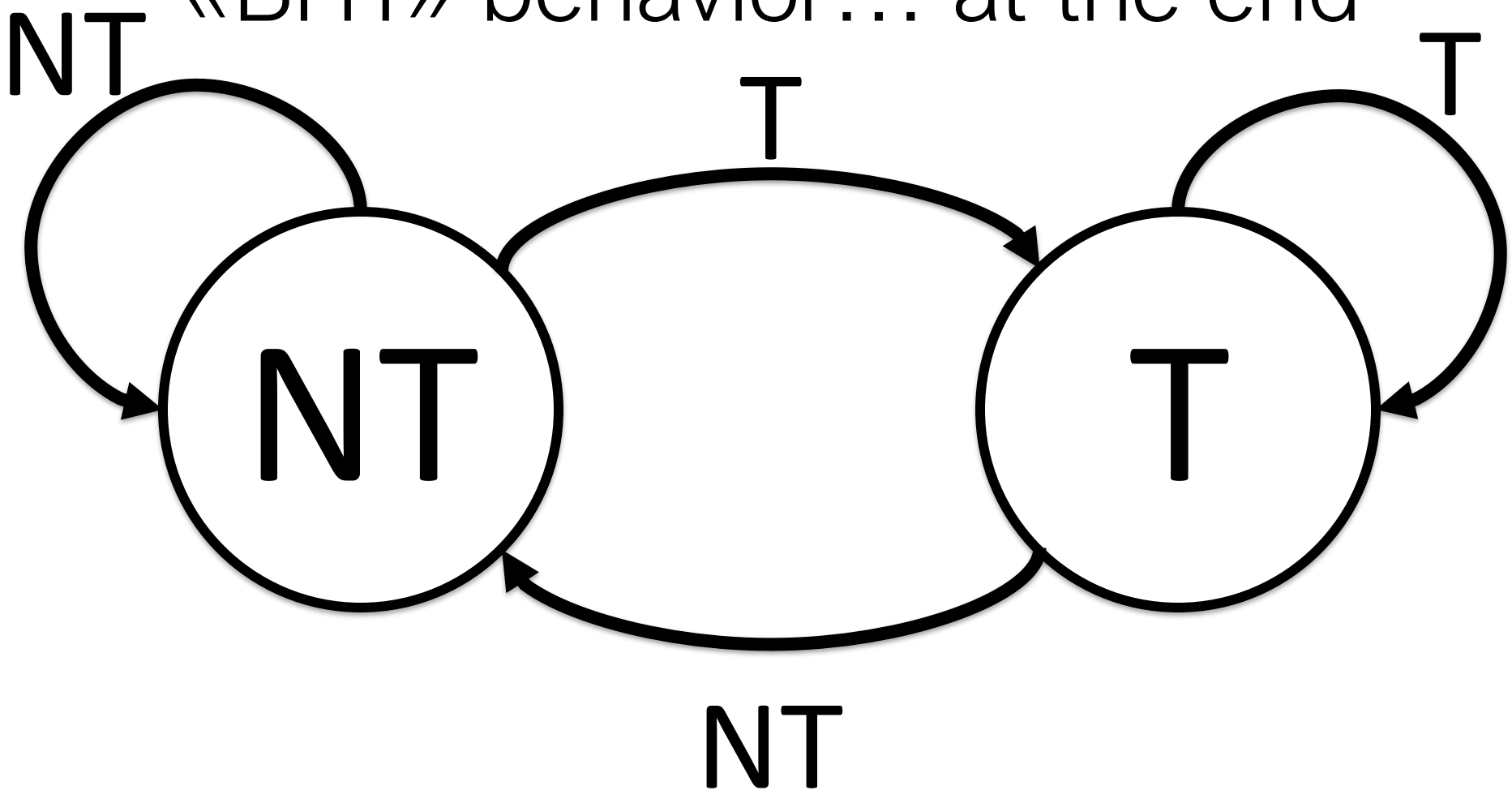
Prediction: T
Outcome: NT

Misprediction

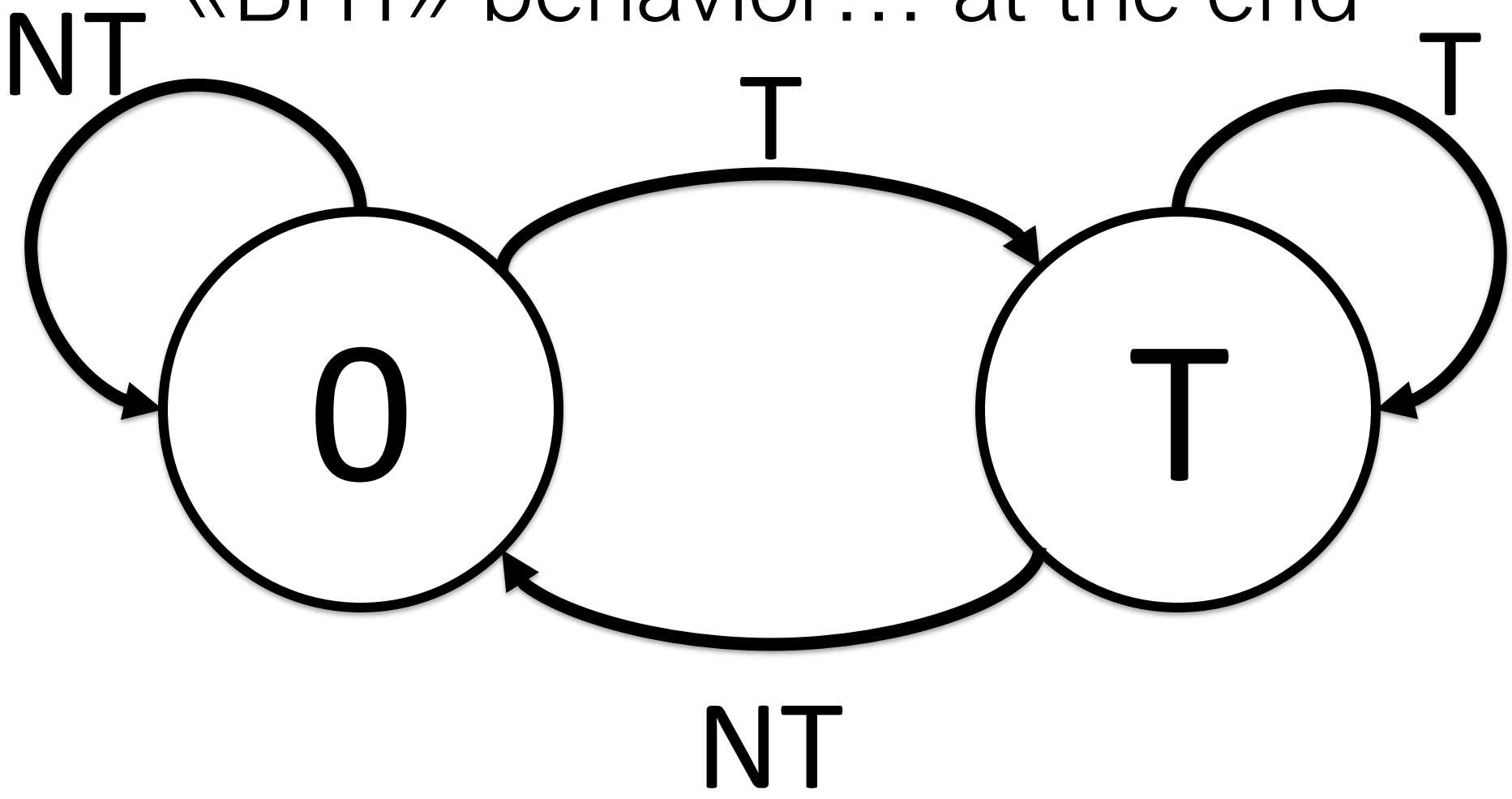


«BHT» behavior… at the end

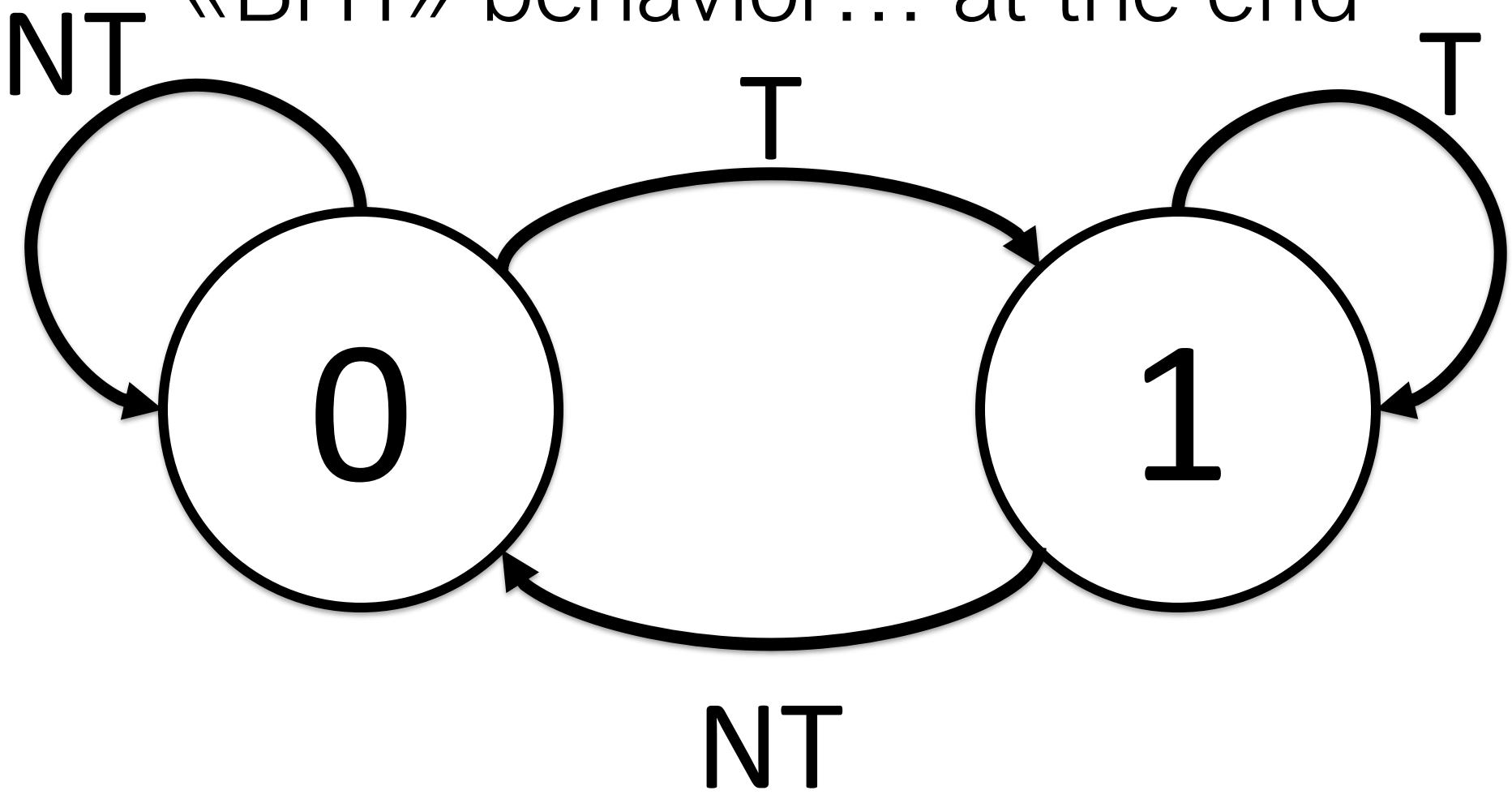
«BHT» behavior… at the end



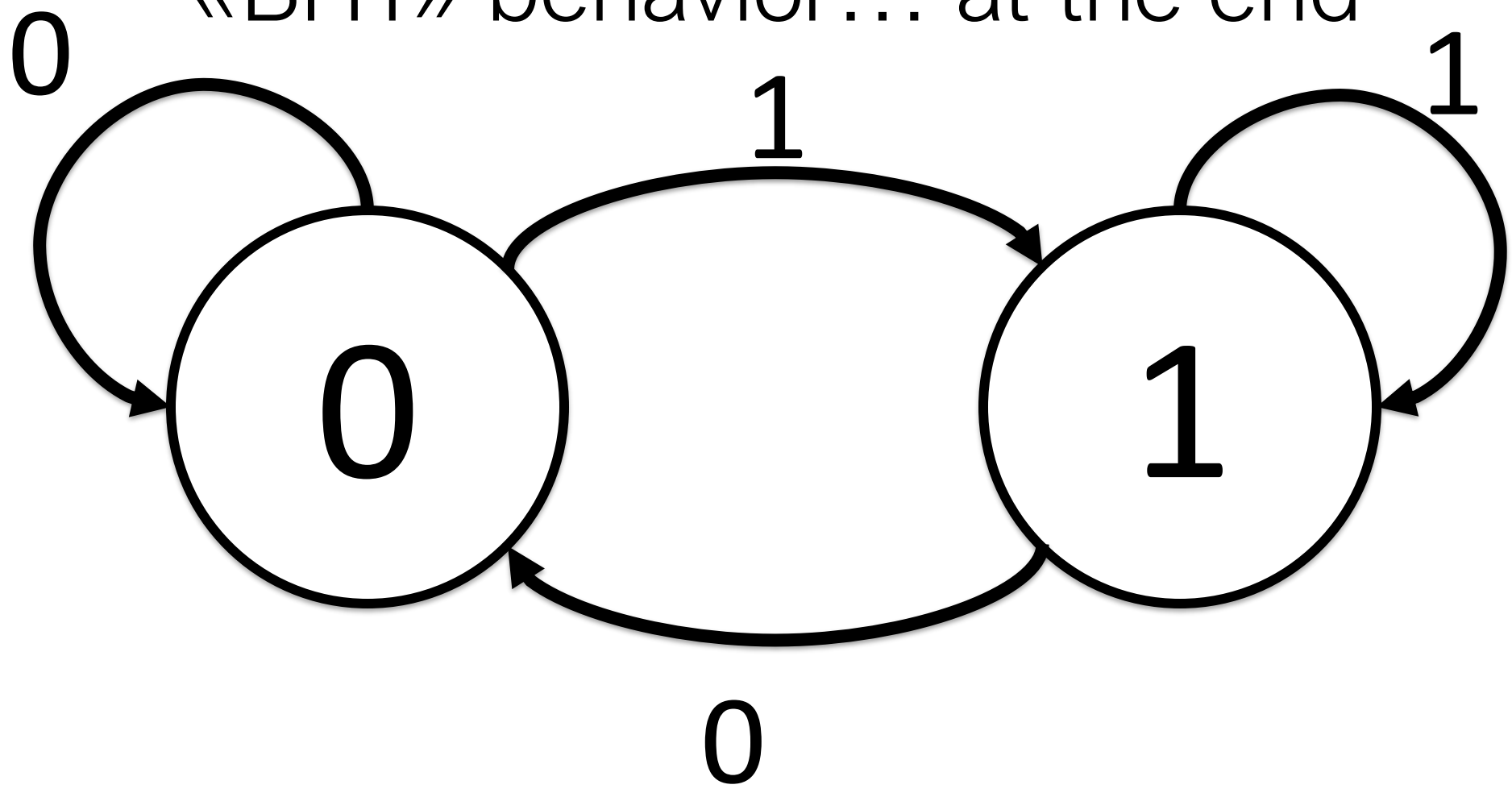
«BHT» behavior… at the end



«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 (such as in GShare).

1-bit BHT: example

R1 is set to 10

```
LOOP:   MULTD   F2   F2   F1
        SUBI    R1   R1   1
        BNEZ    R1   LOOP
```

1-bit BHT: example

R1 is set to 10

```
LOOP:   MULTD   F2   F2   F1
        SUBI    R1   R1   1
        BNEZ    R1   LOOP
```

1-bit prediction: Taken

1-bit BHT: example

R1 is set to 10

```
LOOP:   MULTD   F2   F2   F1
        SUBI    R1   R1   1
        BNEZ    R1   LOOP
```

1-bit prediction: Taken

miss-prediction: 1 (accuracy 90%)

1-bit BHT: example

R1 is set to 10

```
LOOP:   MULTD   F2   F2   F1
        SUBI    R1   R1   1
        BNEZ    R1   LOOP
```

1-bit prediction: Not Taken

1-bit BHT: example

R1 is set to 10

```
LOOP:   MULTD   F2   F2   F1
        SUBI    R1   R1   1
        BNEZ    R1   LOOP
```

1-bit prediction: Not Taken

miss-prediction: 2 (accuracy: 80%)

1-bit BHT: example

R0 is set to 10
R1 is set to 10

```
LOOP1:  LD      F1  0   R0
        ADDD    F2  F1  F1
        ADDI    R1  R1  10
LOOP:   MULTD F2 F2  F1
        SUBI    R1  R1  1
        BNEZ    R1  LOOP
        SUBI    R0  R0  1
        BNEZ    R0  LOOP1
```

1-bit BHT: example

R0 is set to 10
R1 is set to 10

```
LOOP1:  LD      F1  0   R0
        ADDD    F2  F1  F1
        ADDI    R1  R1  10
LOOP:   MULTD F2 F2  F1
        SUBI    R1  R1  1
        BNEZ    R1  LOOP
        SUBI    R0  R0  1
        BNEZ    R0  LOOP1
```

[LOOP]1-bit prediction: Taken
miss-prediction:

1-bit BHT: example

R0 is set to 10
R1 is set to 10

```
LOOP1:  LD      F1  0   R0
        ADDD    F2  F1  F1
        ADDI    R1  R1  10
LOOP:   MULTD F2 F2  F1
        SUBI    R1  R1  1
        BNEZ    R1  LOOP
        SUBI    R0  R0  1
        BNEZ    R0  LOOP1
```

[LOOP]1-bit prediction: Taken
miss-prediction: 1 + 9*2 (accuracy close to 80%)

1-bit BHT: example

R0 is set to 10
R1 is set to 10

```
LOOP1:  LD      F1  0   R0
        ADDD    F2  F1  F1
        ADDI    R1  R1  10
LOOP:   MULTD F2 F2  F1
        SUBI    R1  R1  1
        BNEZ    R1  LOOP
        SUBI    R0  R0  1
        BNEZ    R0  LOOP1
```

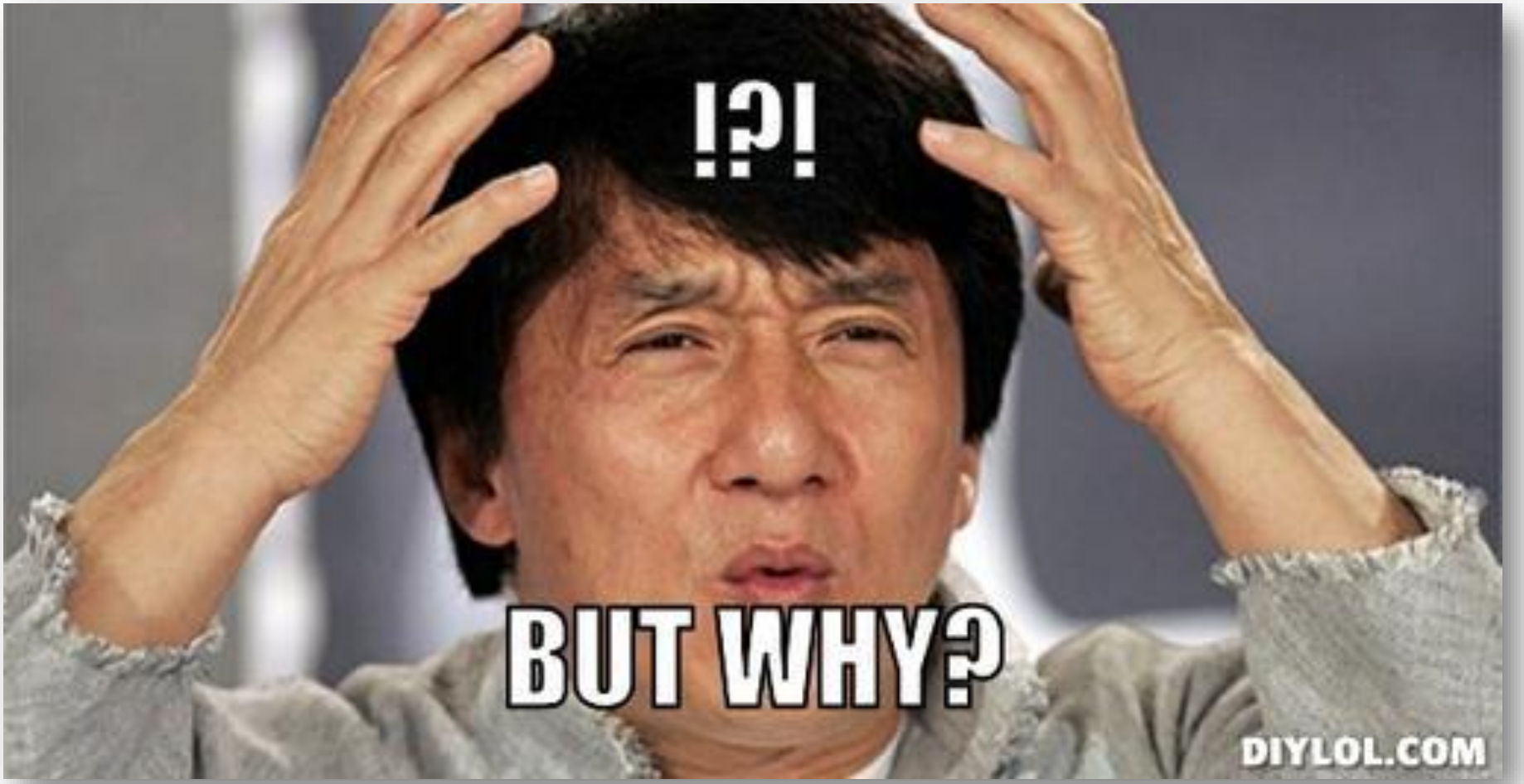
[LOOP]1-bit prediction: Not Taken
miss-prediction:

1-bit BHT: example

R0 is set to 10
R1 is set to 10

```
LOOP1:  LD       F1  0   R0
        ADDD     F2  F1  F1
        ADDI     R1  R1  10
LOOP:   MULTD F2 F2  F1
        SUBI     R1  R1  1
        BNEZ     R1  LOOP
        SUBI     R0  R0  1
        BNEZ     R0  LOOP1
```

[LOOP]1-bit prediction: Not Taken
miss-prediction: 10*2 (accuracy: 80%)

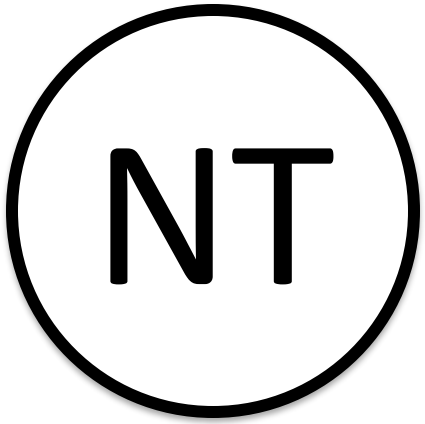

¡?¡
BUT WHY?
DIYLOL.COM

2-bit Branch History Table

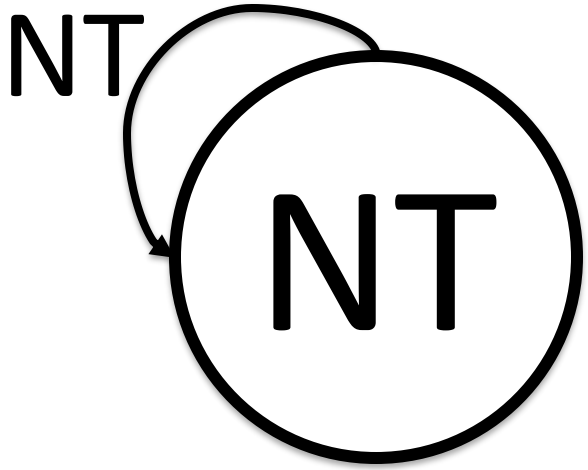
- The prediction must miss twice before it is changed.
- In a loop branch, at the last loop iteration, we do not need to change the prediction.
- For each index in the table, the 2 bits are used to encode the four states of a finite state machine.

2-bit BHT… aka 2nd chance ;)

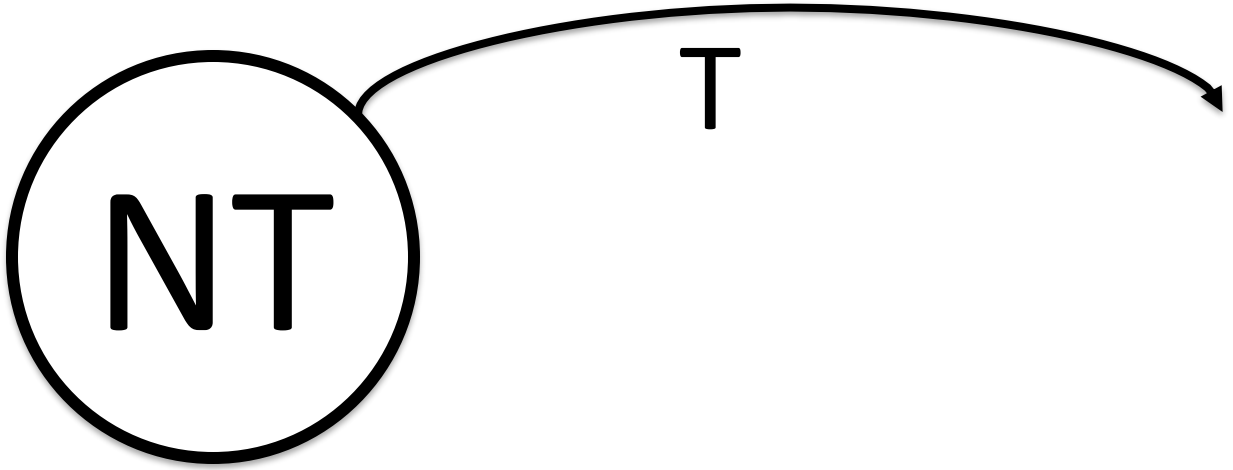
2-bit BHT… aka 2nd chance ;)



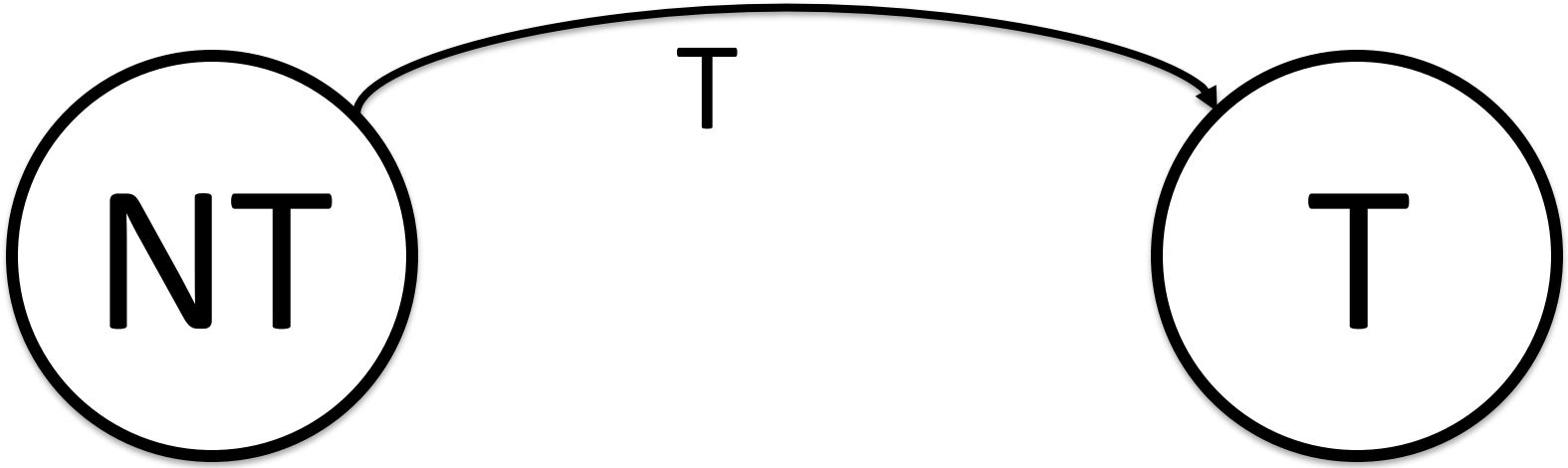
2-bit BHT… aka 2nd chance ;)



2-bit BHT… aka 2nd chance ;)

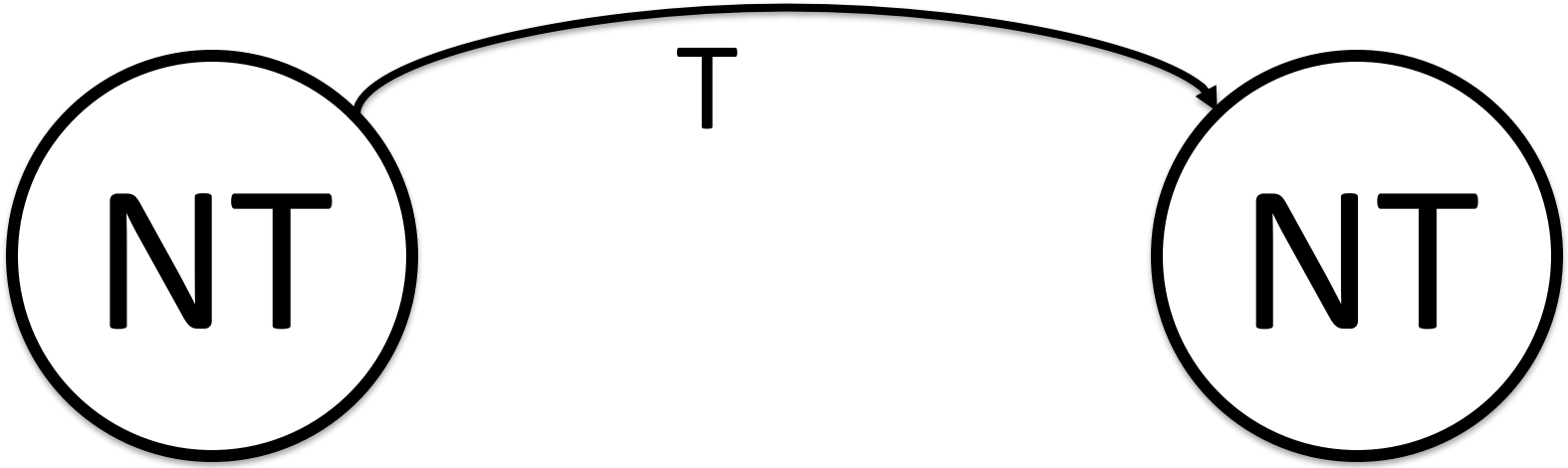


2-bit BHT… aka 2nd chance ;)

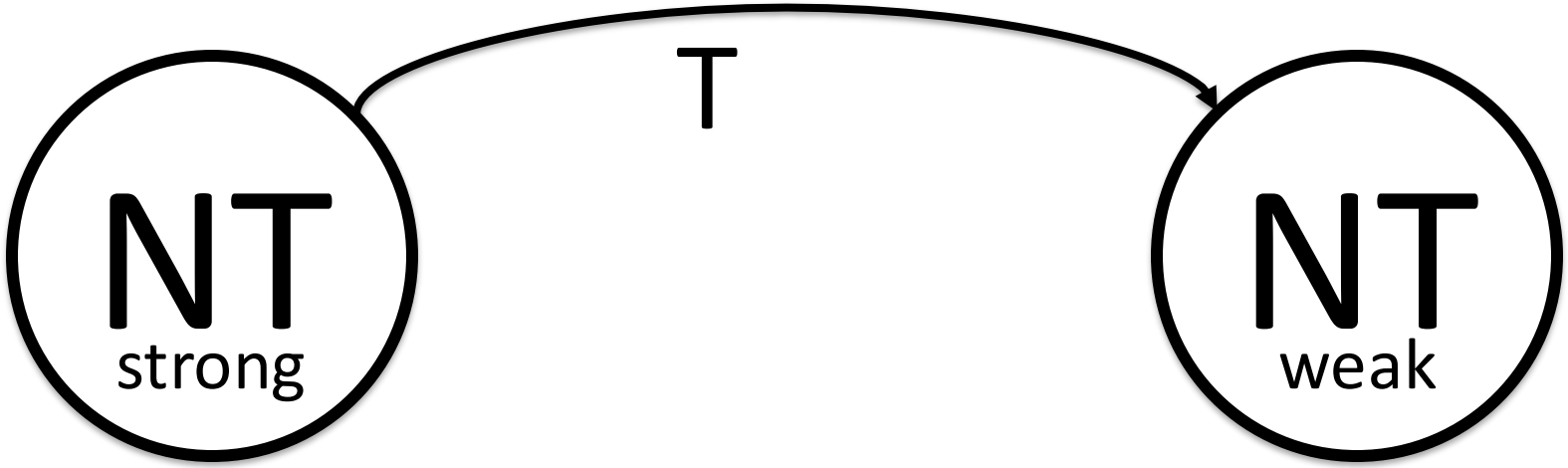


2-bit BHT… aka 2nd chance ;)

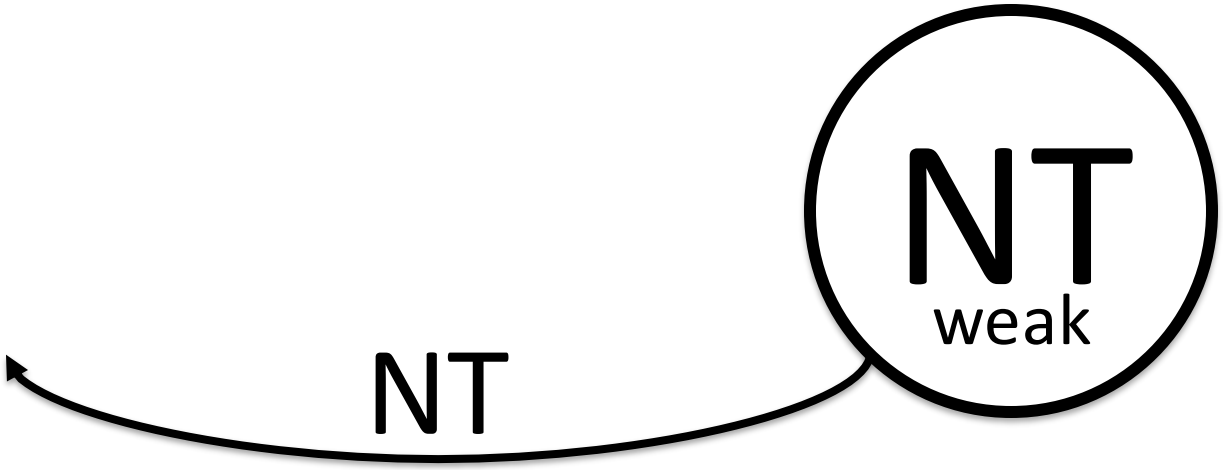
2-bit BHT… aka 2nd chance ;)



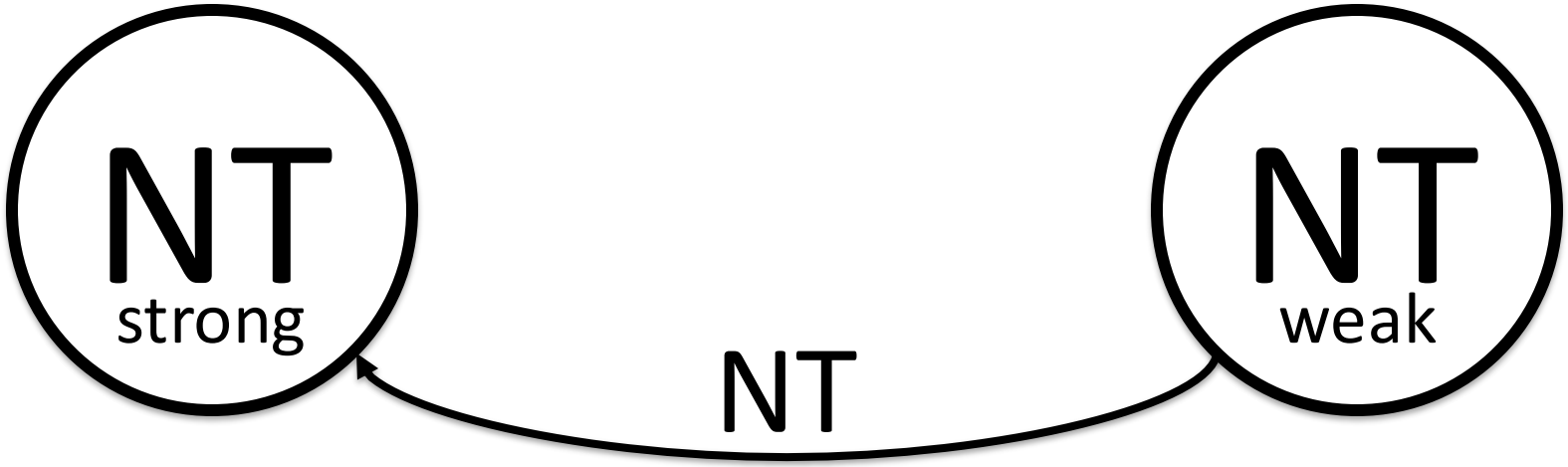
2-bit BHT… aka 2nd chance ;)



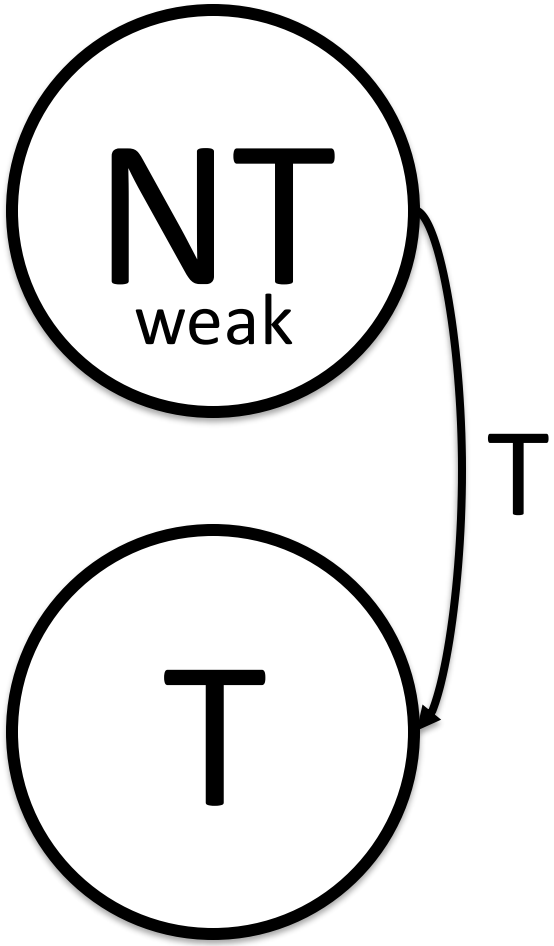
2-bit BHT… aka 2nd chance ;)



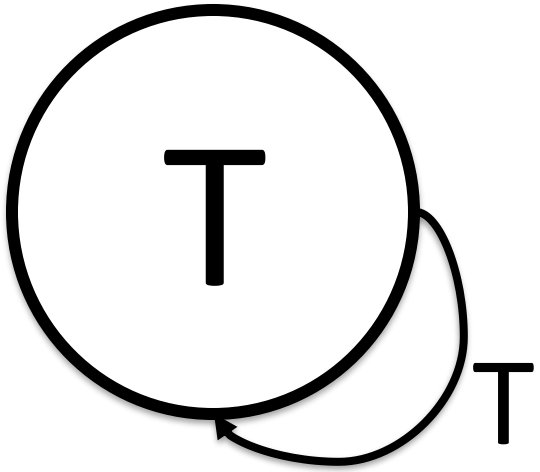
2-bit BHT… aka 2nd chance ;)



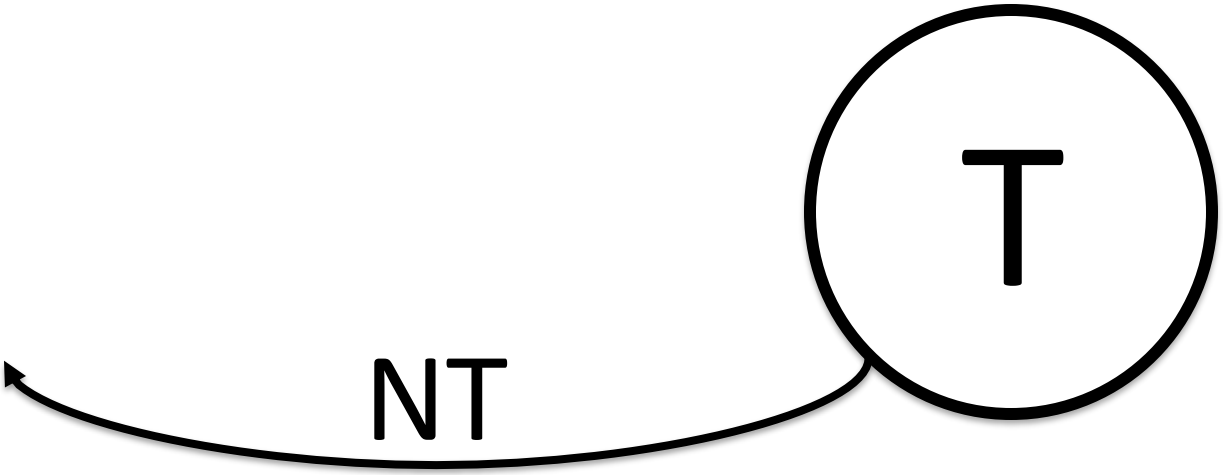
2-bit BHT… aka 2nd chance ;)



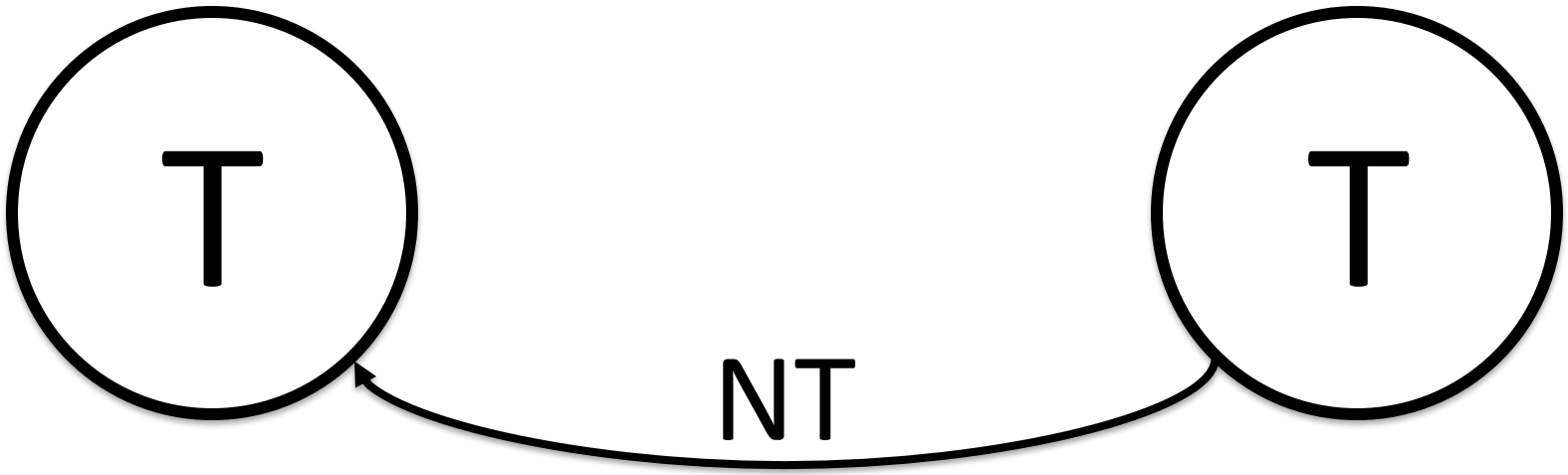
2-bit BHT… aka 2nd chance ;)



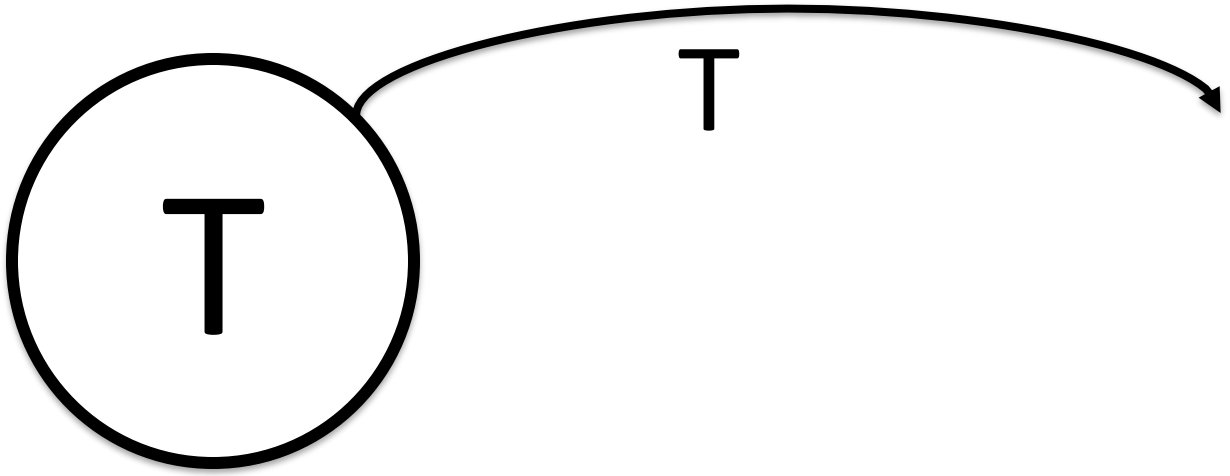
2-bit BHT… aka 2nd chance ;)



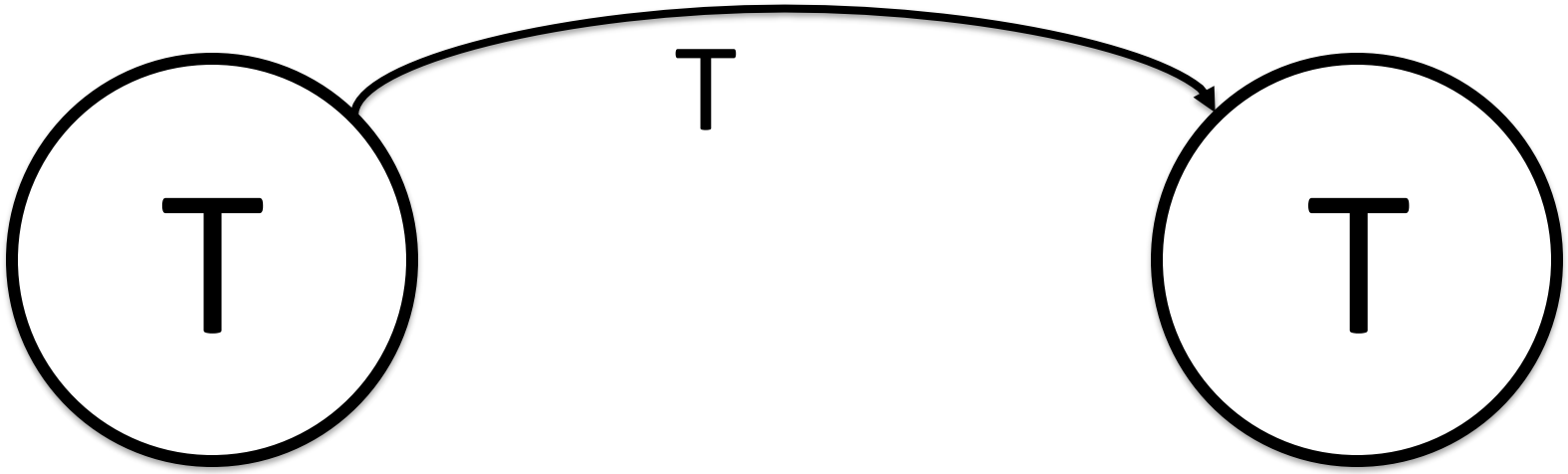
2-bit BHT… aka 2nd chance ;)



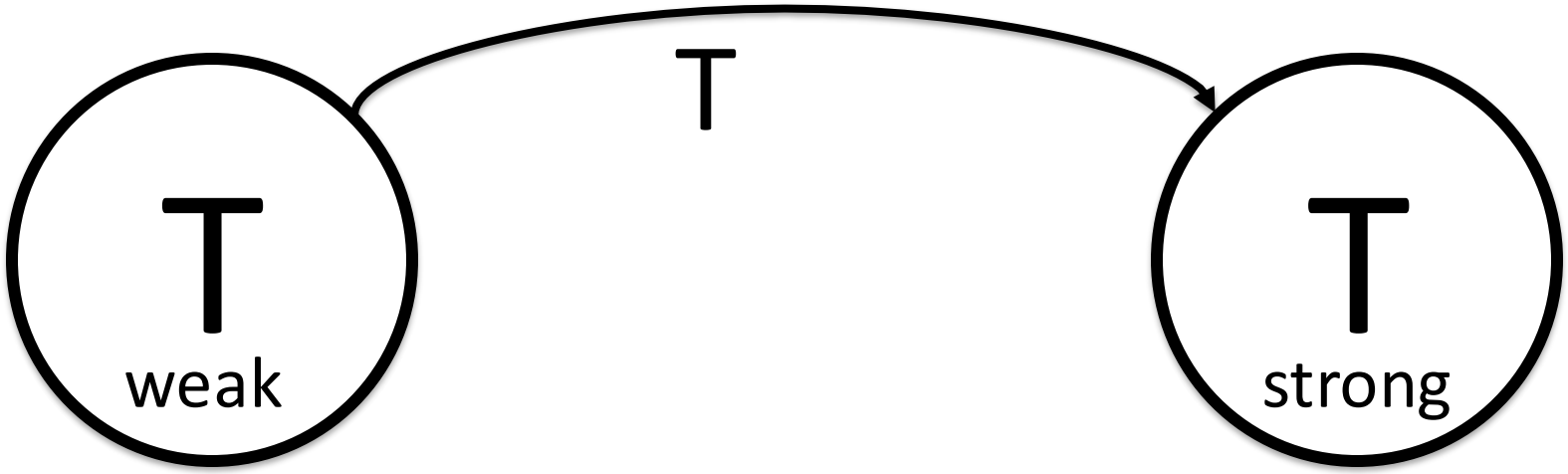
2-bit BHT… aka 2nd chance ;)



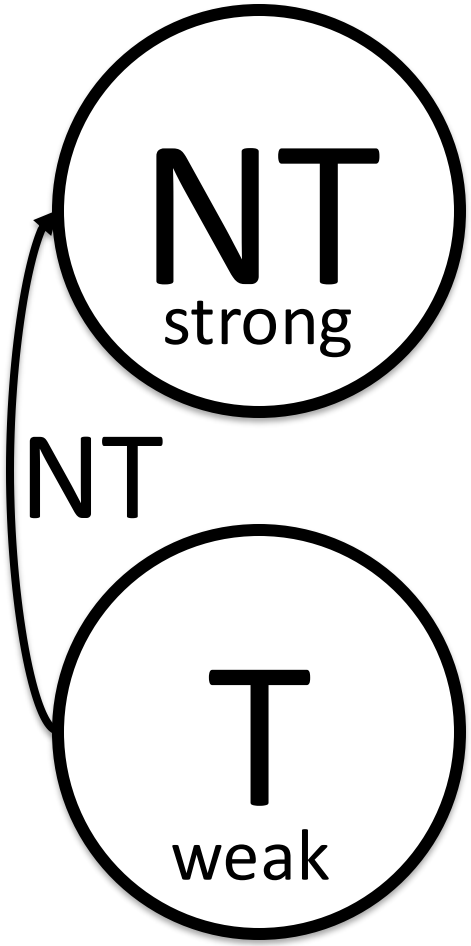
2-bit BHT… aka 2nd chance ;)



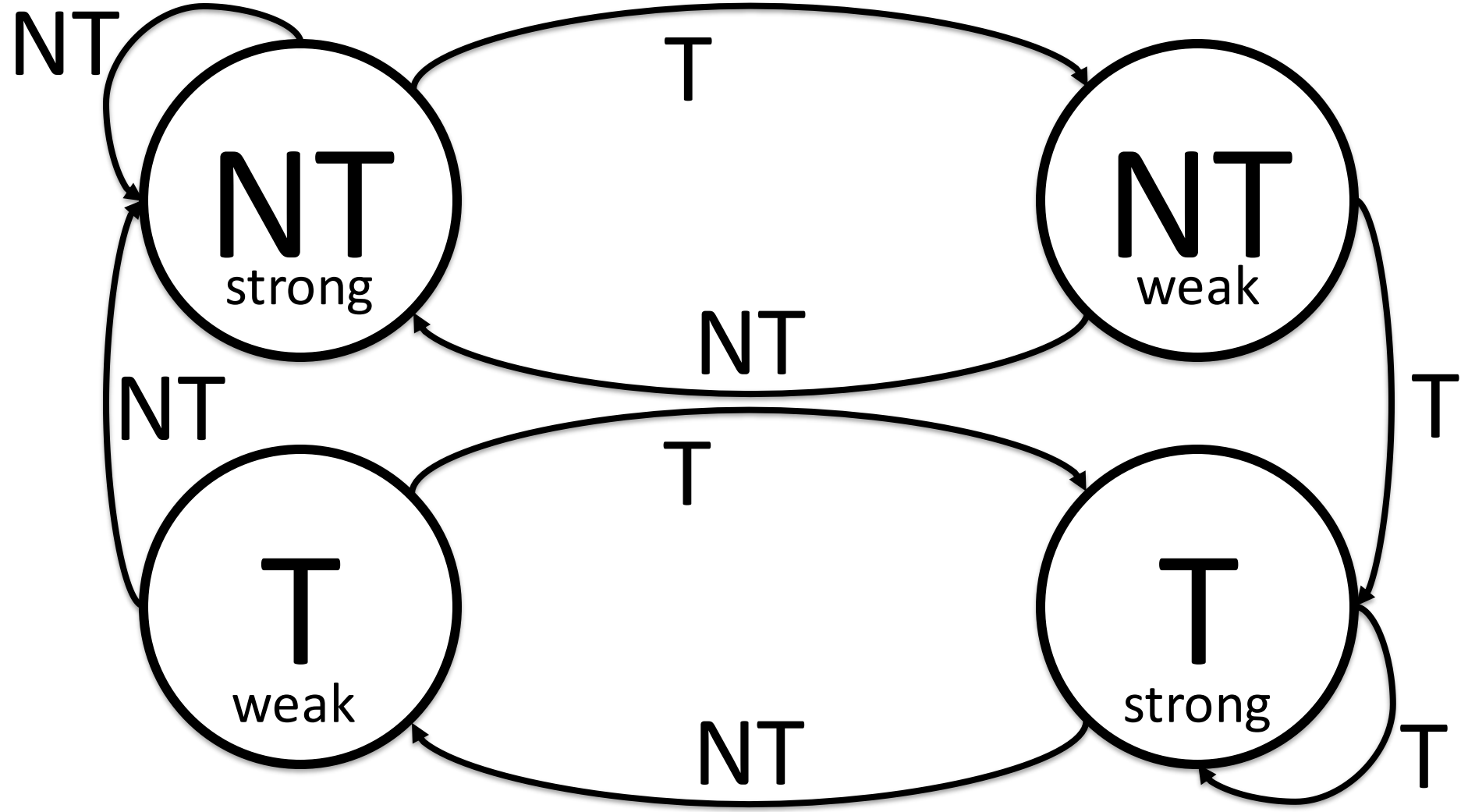
2-bit BHT… aka 2nd chance ;)



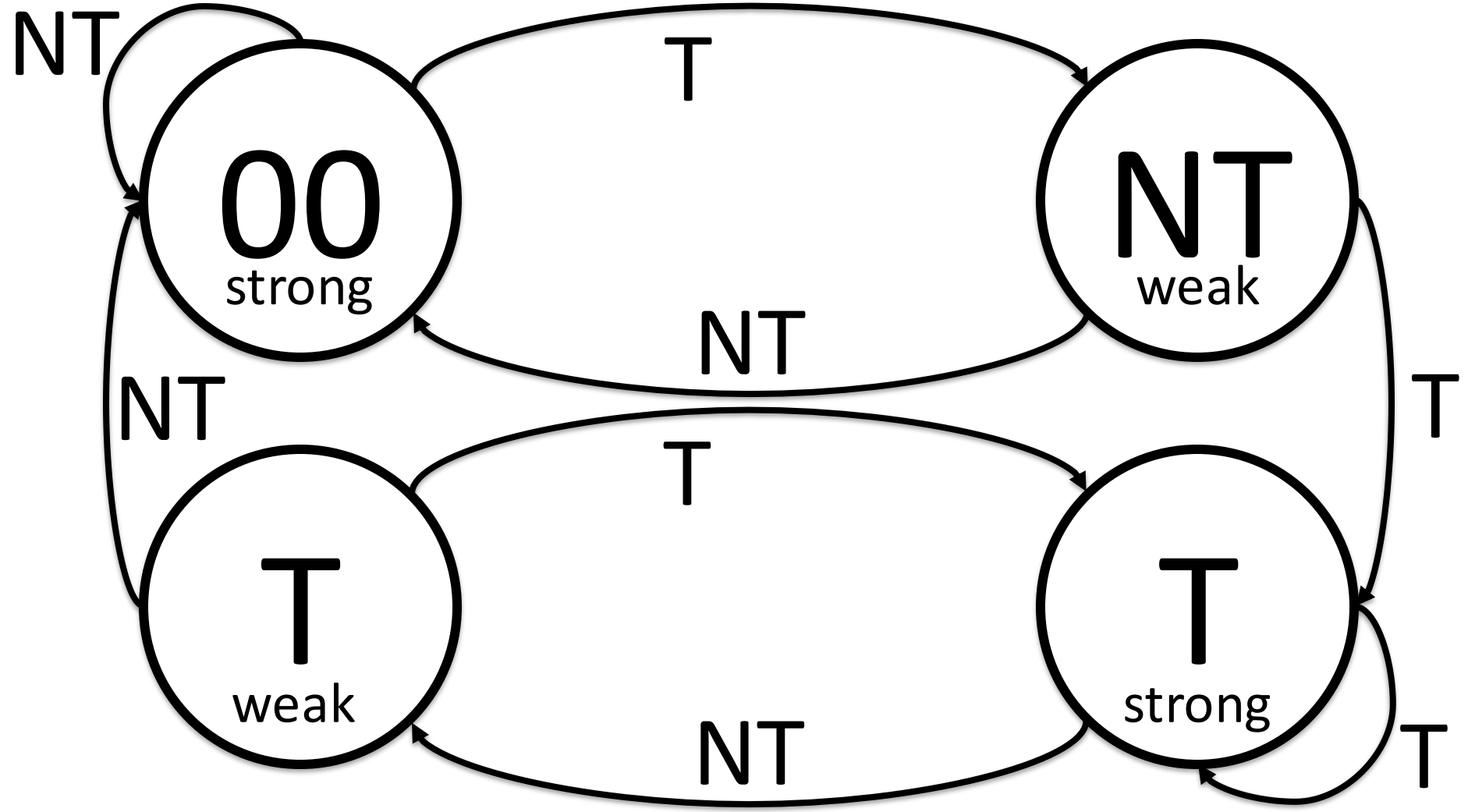
2-bit BHT… aka 2nd chance ;)



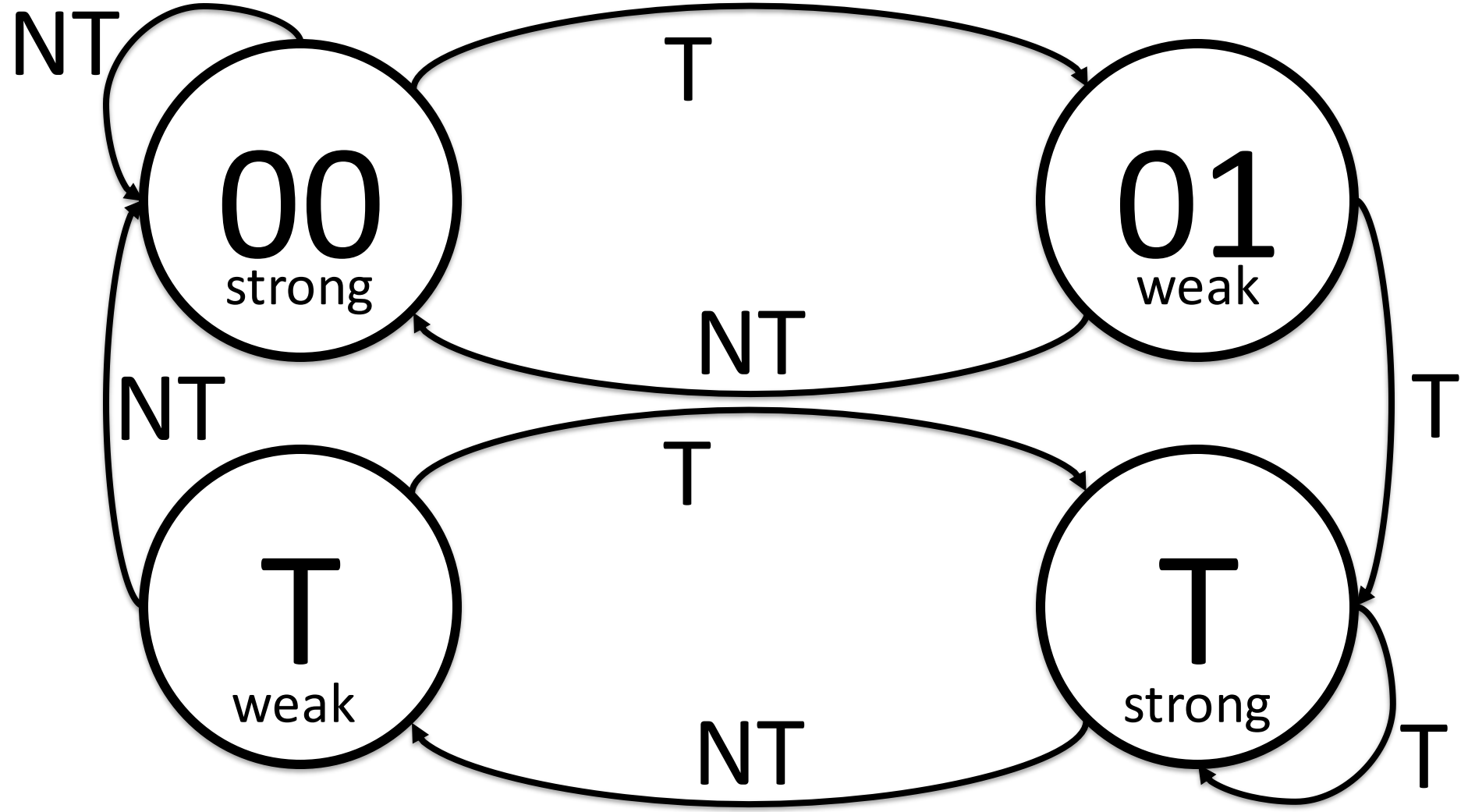
2-bit BHT… all together…



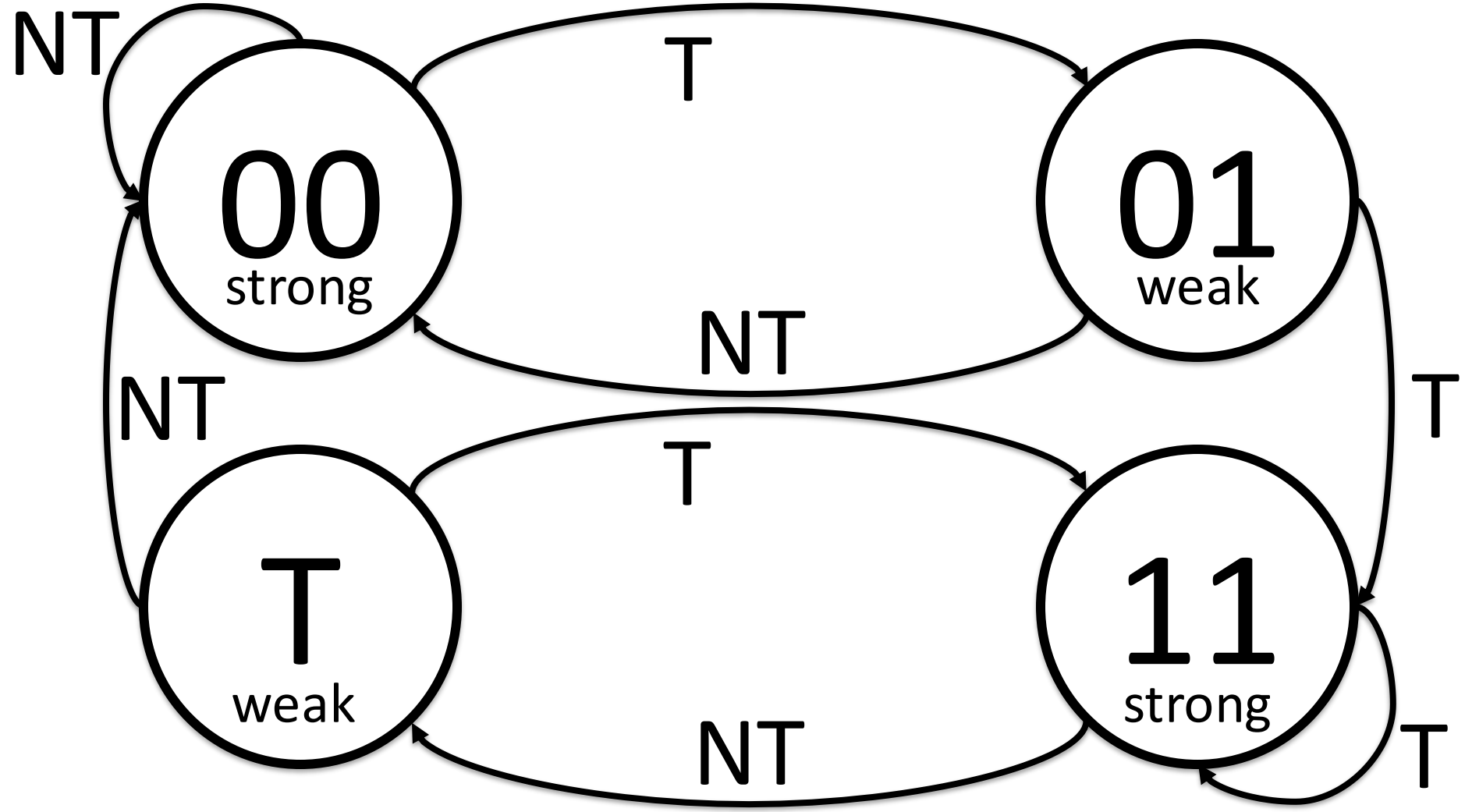
2-bit BHT… all together…



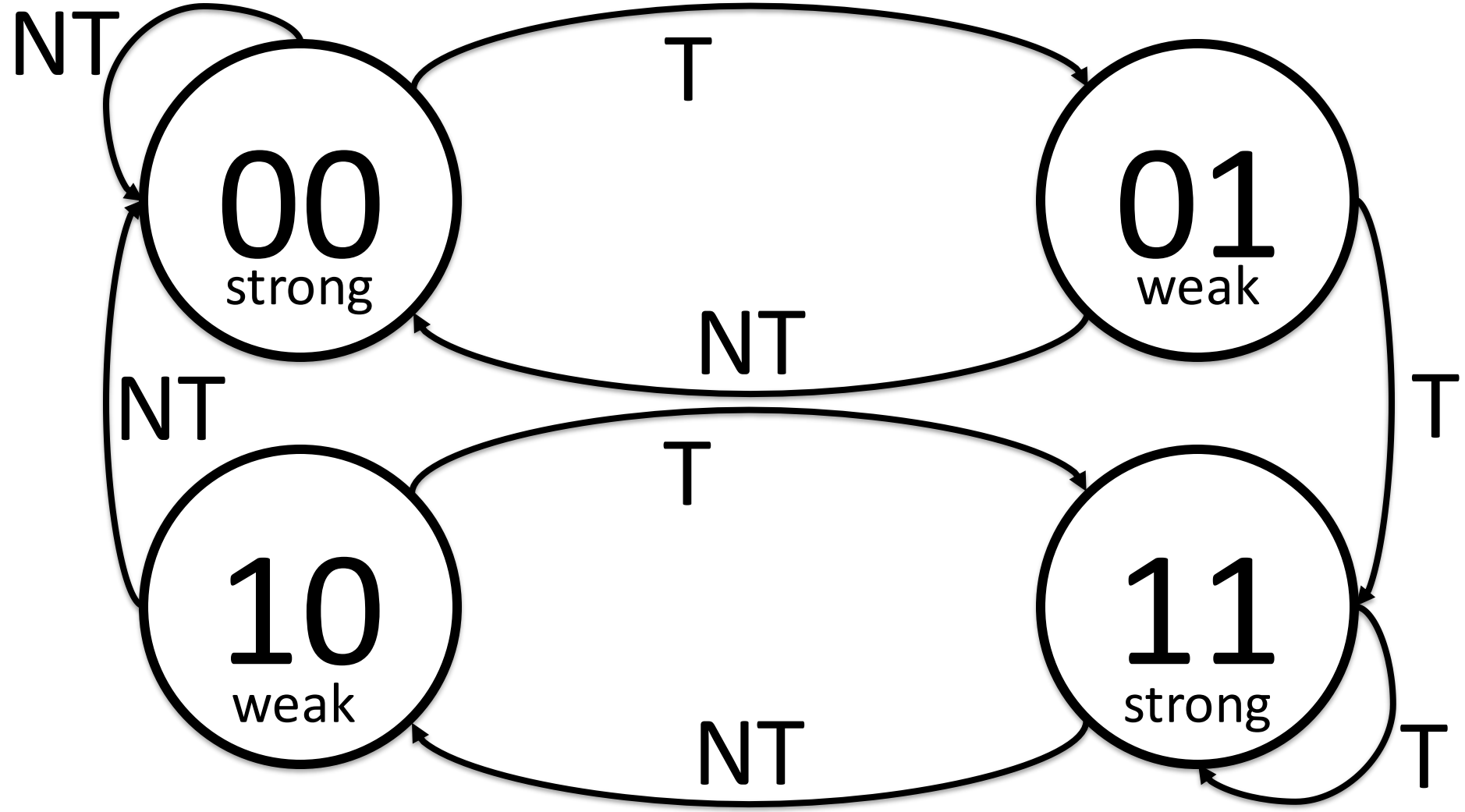
2-bit BHT… all together…



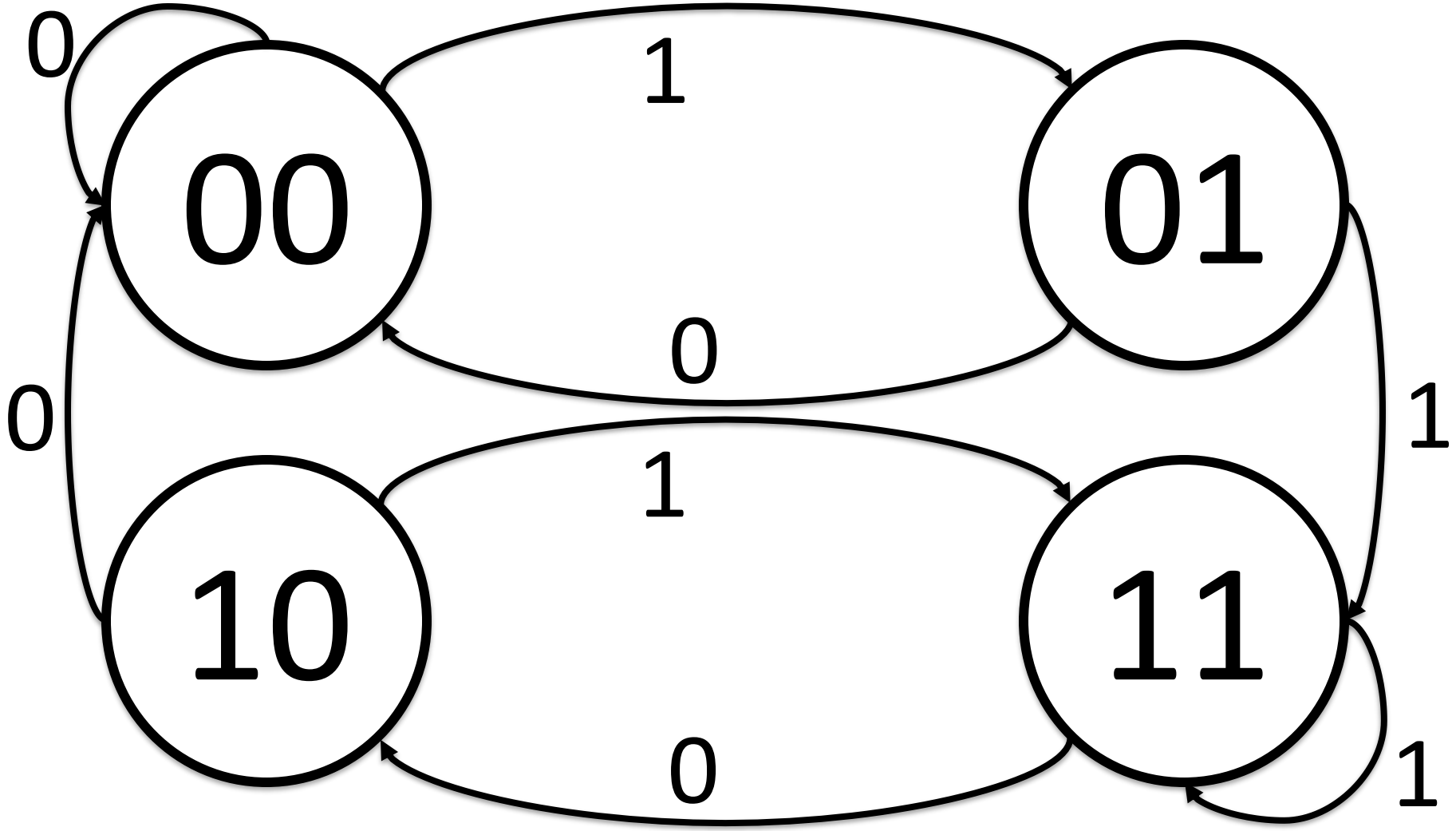
2-bit BHT… all together…



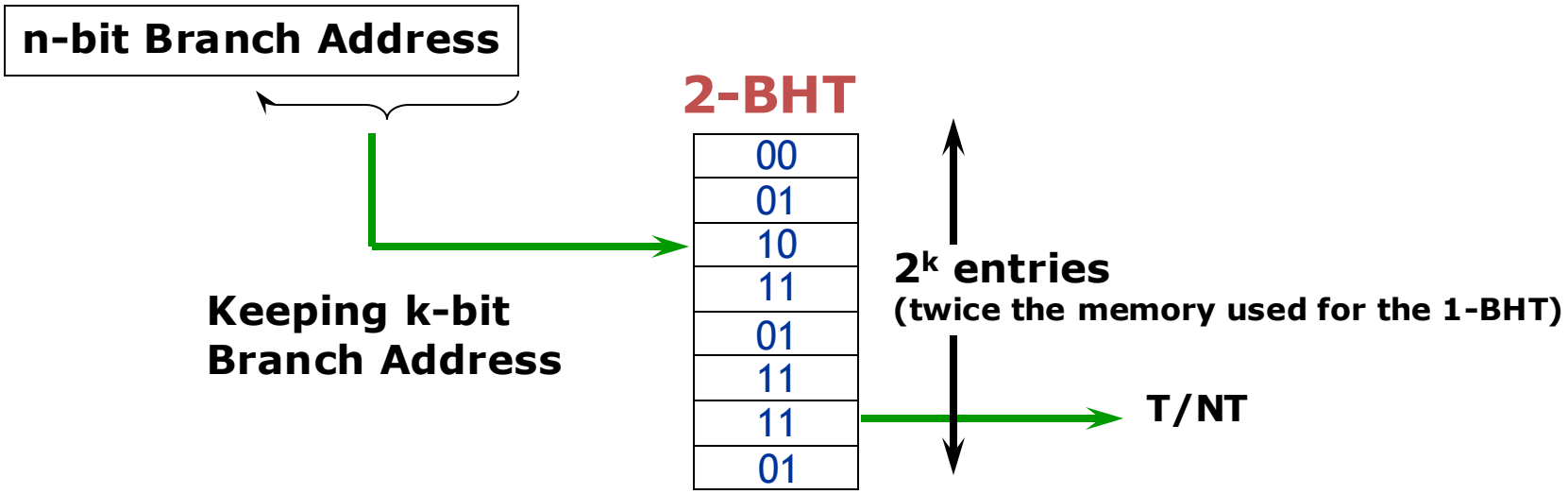
2-bit BHT… all together…



2-bit BHT… all together…



2-bit Branch History Table



2-bit BHT: example

R0 is set to 10
R1 is set to 10

```
LOOP1:  LD      F1  0   R0
        ADDD    F2  F1  F1
        ADDI    R1  R1  10
LOOP:   MULTD   F2  F2  F1
        SUBI    R1  R1  1
        BNEZ    R1  LOOP
        SUBI    R0  R0  1
        BNEZ    R0  LOOP1
```

2-bit BHT: example

R0 is set to 10
R1 is set to 10

```
LOOP1:  LD      F1  0   R0
        ADDD    F2  F1  F1
        ADDI    R1  R1  10
LOOP:   MULTD   F2  F2  F1
        SUBI    R1  R1  1
        BNEZ    R1  LOOP
        SUBI    R0  R0  1
        BNEZ    R0  LOOP1
```

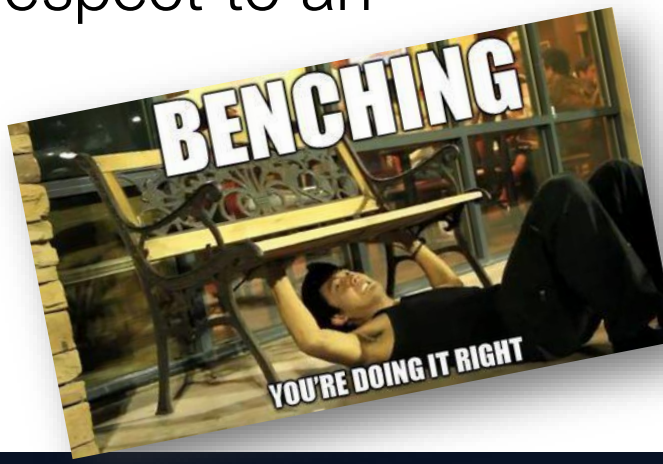
Try this @ home

n-bit Branch History Table

- Generalization: n-bit saturating counter for each entry in the prediction buffer.
  - The counter can take on 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 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.

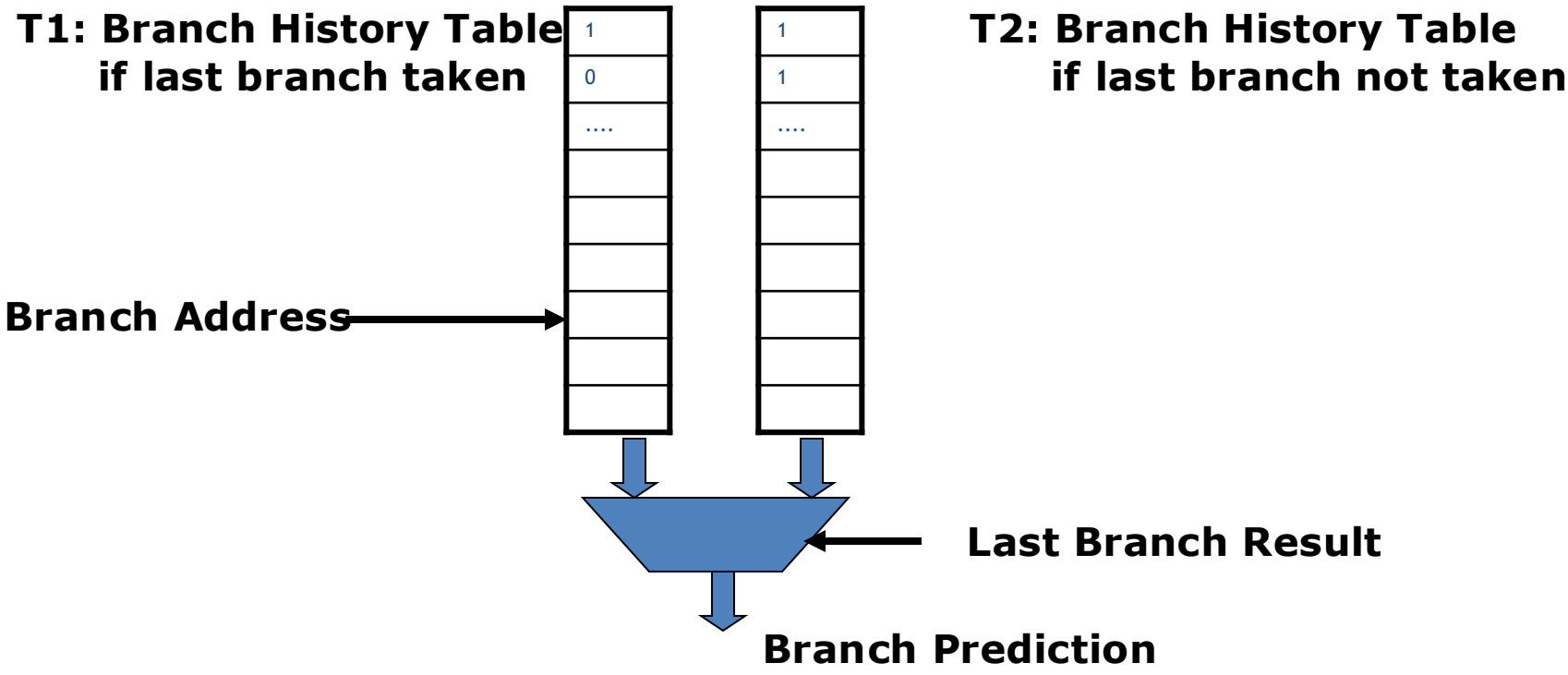
Correlating Branch Predictors

Correlating Branch Predictors

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.

Correlating Branch Predictors: Example



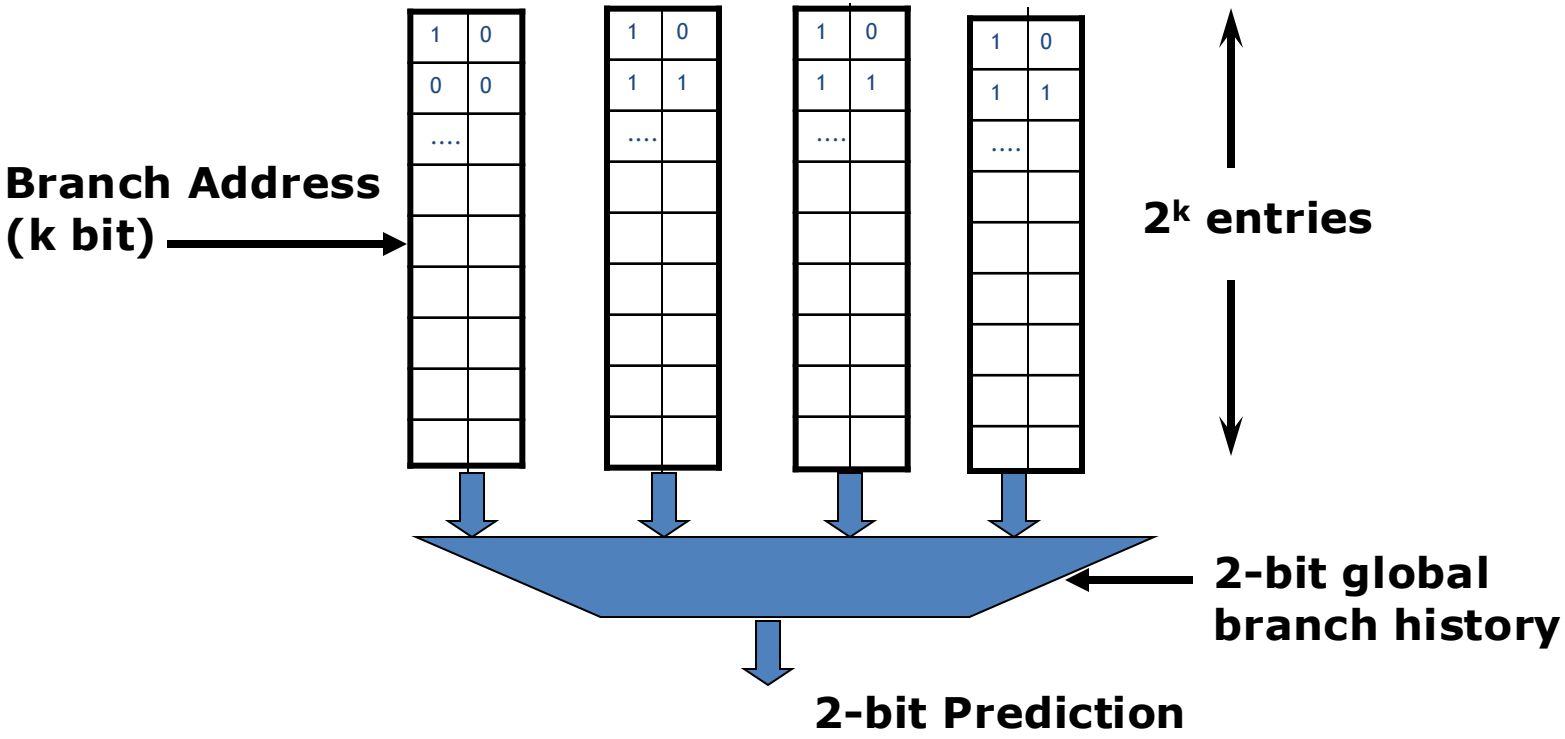
(m, n) Correlating Branch Predictors

- 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).

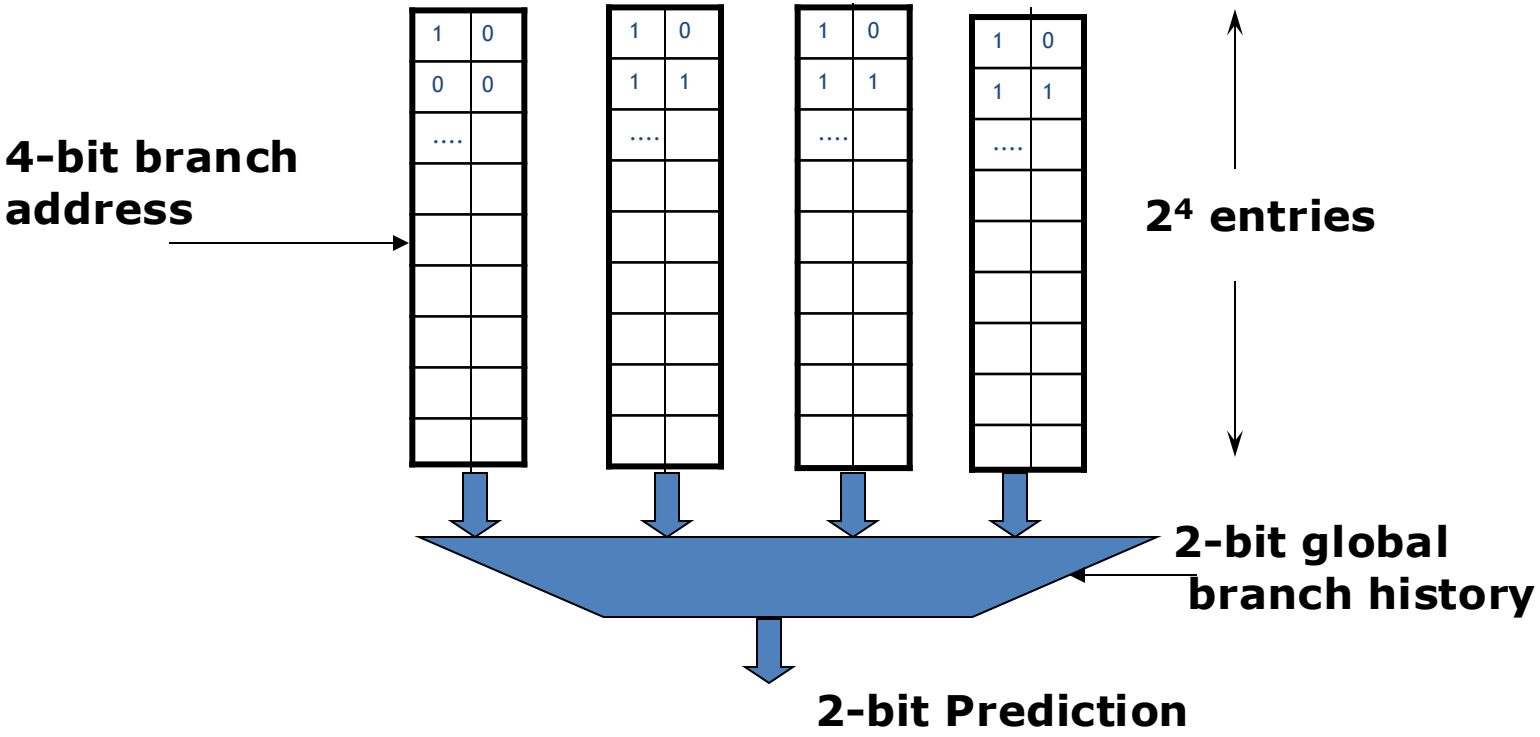
(2, 2) Correlating Branch Predictors

- 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.



(2, 2) Correlating Branch Predictors: example

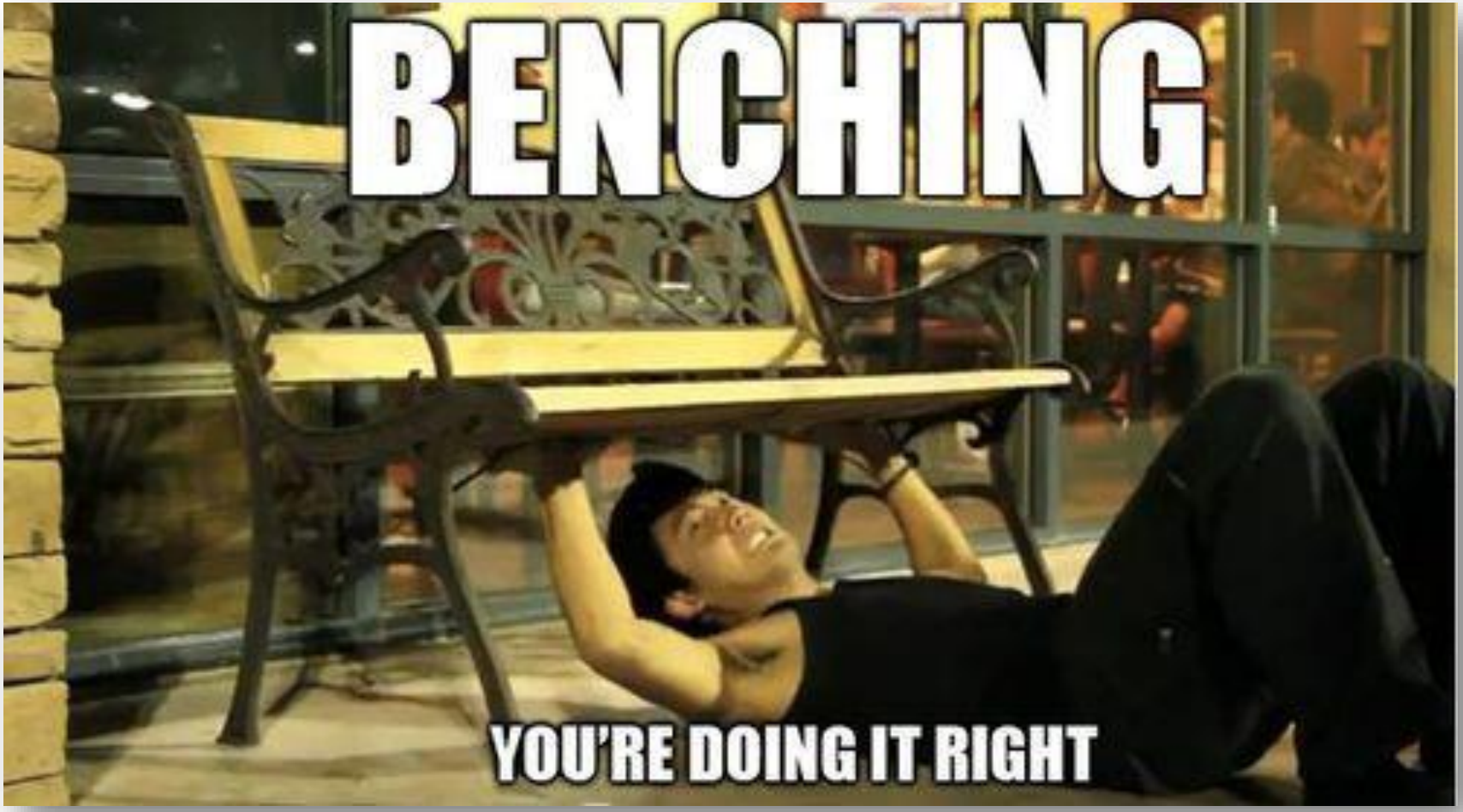
- Example: a (2, 2) correlating predictor with 64 total entries ⇒ 6-bit index composed of: 2-bit global history and 4-bit low-order branch address bits



(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)

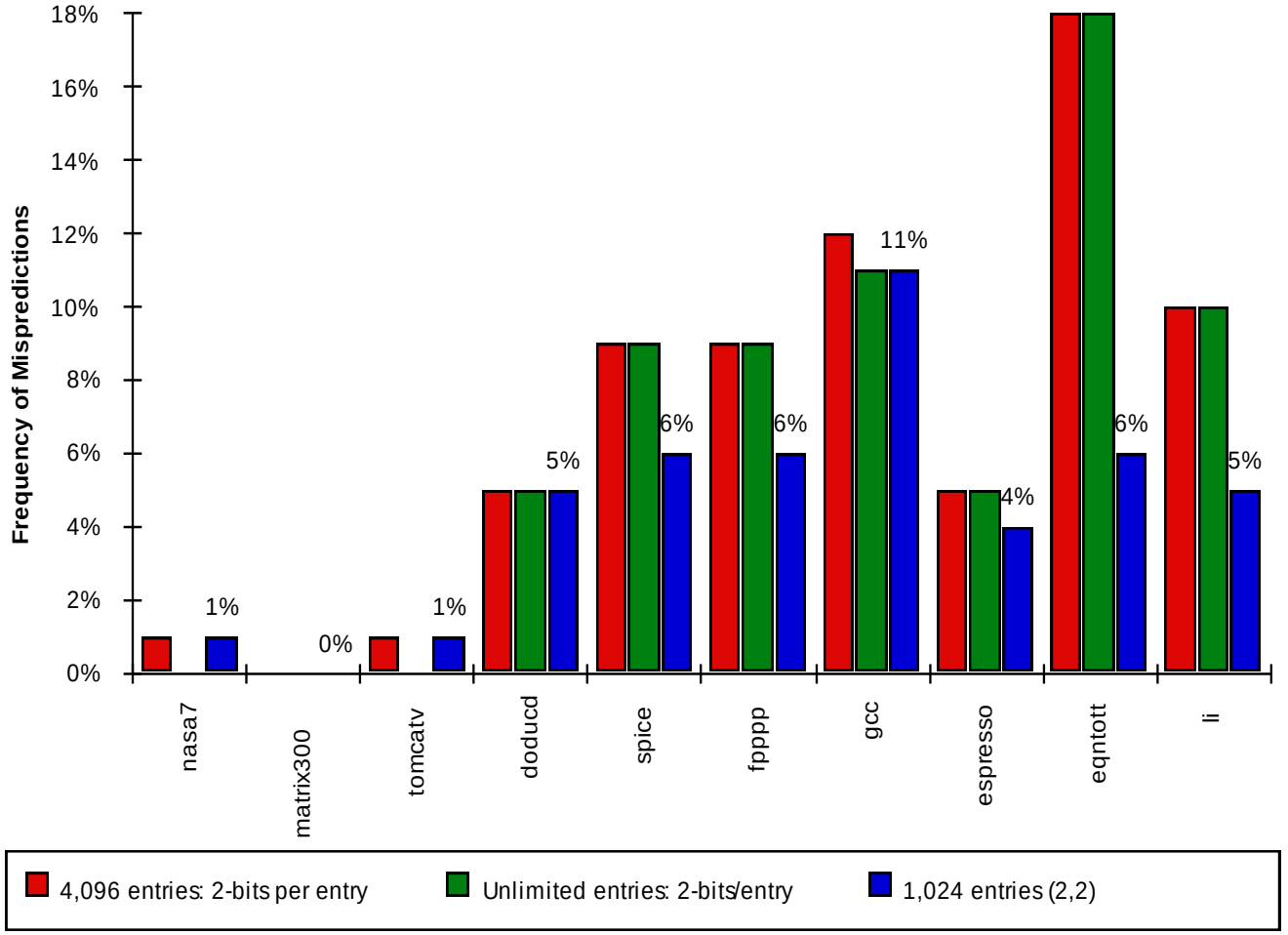
Benchmarks… again…



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 simply 2-bit predictor with the same number of total bits (4K total bits), it often outperforms a 2-bit predictor with an unlimited number of entries.

Accuracy of Correlating Predictors

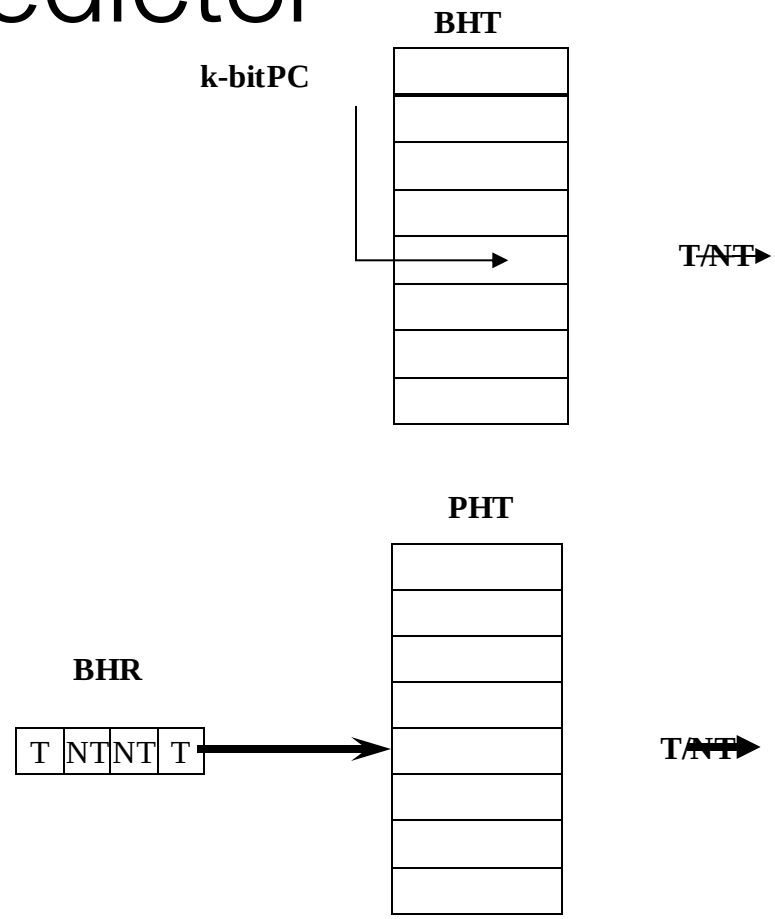


Two-Level Adaptive Branch Predictors

- 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 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 two-bit counter scheme.

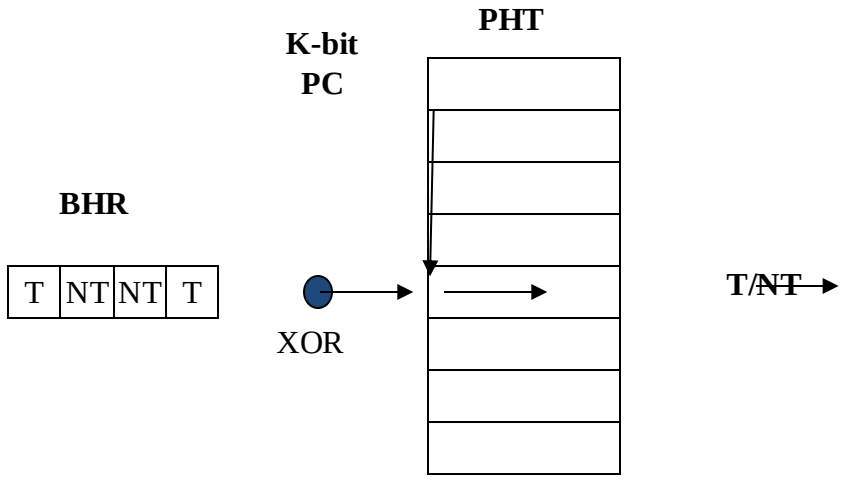
GA Predictor

- BHT: Local predictor
  - Indexed by the low-order bits of PC (branch address)
- GAs: Local and global predictor
  - 2-level predictor: PHT Indexed by the content of BHR (global history)



GShare Predictor

- **GShare:** Local XOR global information
  - Indexed by the exclusive OR of the low-order bits of PC (branch address) and the content of BHR (global history)



References

Championship Branch Prediction (CBP)
2016 edition: http://www.jilp.org/cbp2016/

- An introduction to the branch prediction problem can be found in Chapter 3 of: J. Hennessy and D. Patterson, “Computer Architecture, a Quantitative Approach”, Morgan Kaufmann, third edition, May 2002.
- A survey of basic branch prediction techniques can be found in: D. J. Lalja, “Reducing the Branch Penalty in Pipelined Processors”, Computer, pages 47-55, July 1988.
- A more detailed and advanced survey of the most used branch predictor architectures can be found in: M. Evers and T.-Y. Yeh, “Understanding Branches and Designing Branch Predictors for High-Performance Microprocessors”, Proceedings of the IEEE, Vol. 89, No. 11, pages 1610-1620, November 2001.

THANK YOU
FOR YOUR ATTENTION