

The background of the slide is a high-resolution, colorful photograph of a Pentium 4 microprocessor die. The die is rectangular and densely packed with intricate circuitry, including various colored regions (yellow, green, blue, red) representing different functional blocks and interconnects. The image is oriented horizontally.

CPEN 411: Computer Architecture

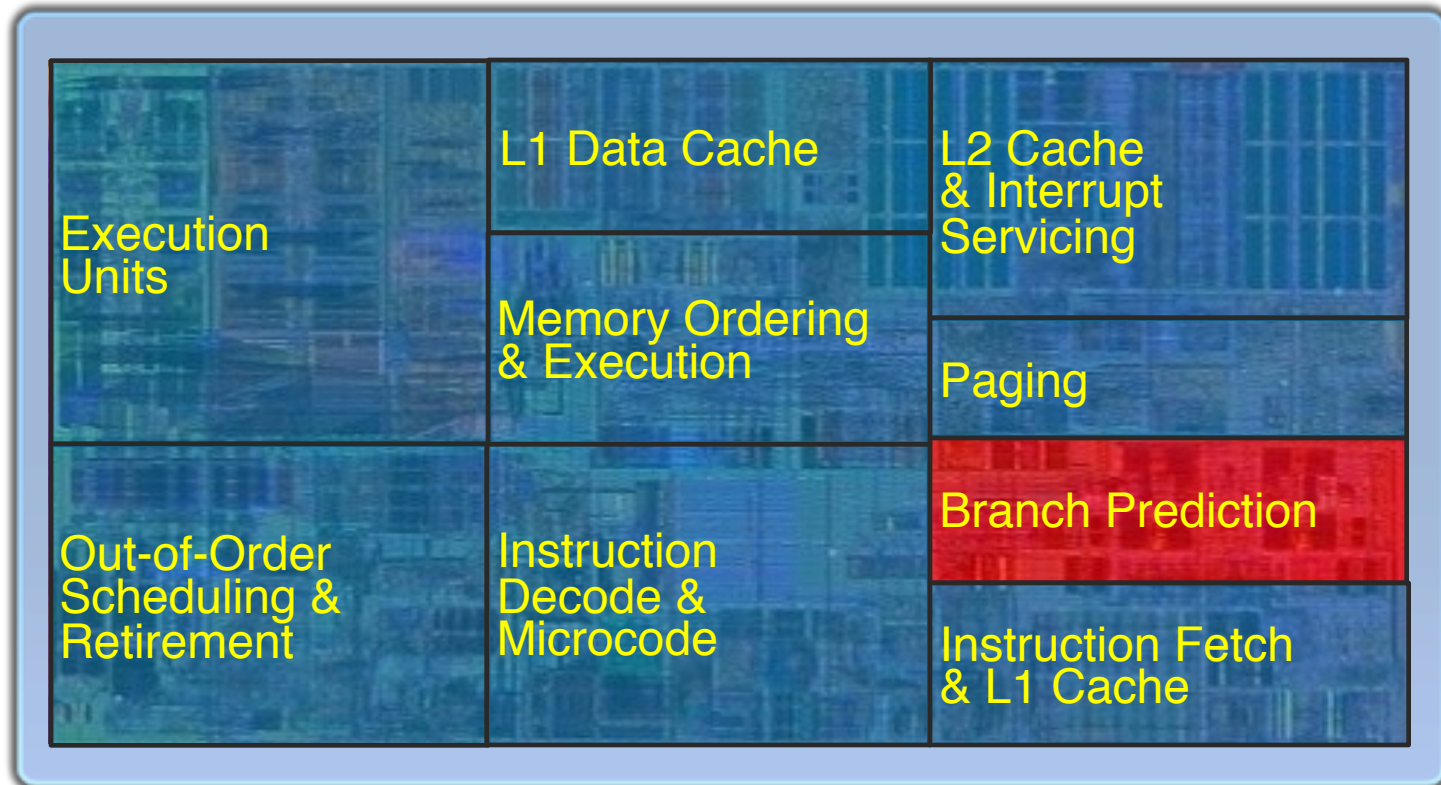
Slide Set #8: Branch Prediction

Instructor: Mieszko Lis

Original Slides: Professor Tor Aamodt

Background: Pentium 4 die photo (uses advanced branch prediction mechanisms)

Intel Core i7



Highlighted above: Significant investment of silicon area to “predict branches”.

Introduction to Slide Set 8

Introduction to Slide Set 8

- In the past couple of slide sets we considered how to execute instructions “out-of-order”.

Introduction to Slide Set 8

- In the past couple of slide sets we considered how to execute instructions “out-of-order”.
- The key to out-of-order execution (with Tomasulo’s algorithm) is finding instructions which do not have true data dependencies.

Introduction to Slide Set 8

- In the past couple of slide sets we considered how to execute instructions “out-of-order”.
- The key to out-of-order execution (with Tomasulo’s algorithm) is finding instructions which do not have true data dependencies.
- **Did not talk about how to handle branches:** Branches limit number of instructions we can consider for dynamic scheduling.

Introduction to Slide Set 8

- In the past couple of slide sets we considered how to execute instructions “out-of-order”.
- The key to out-of-order execution (with Tomasulo’s algorithm) is finding instructions which do not have true data dependencies.
- **Did not talk about how to handle branches:** Branches limit number of instructions we can consider for dynamic scheduling.
- In this slide set, we explore mechanisms for predicting the “outcome” of a branch long before executing the branch.

Introduction to Slide Set 8

- In the past couple of slide sets we considered how to execute instructions “out-of-order”.
- The key to out-of-order execution (with Tomasulo’s algorithm) is finding instructions which do not have true data dependencies.
- **Did not talk about how to handle branches:** Branches limit number of instructions we can consider for dynamic scheduling.
- In this slide set, we explore mechanisms for predicting the “outcome” of a branch long before executing the branch.
- **Requires a way to “recover” when the prediction is incorrect** (our prediction techniques **will** be incorrect sometimes). We learn how to “recover” after incorrect predictions when using Tomasulo’s algorithm in the next slide set. This slide set mainly focuses on how to make the predictions.

Learning Objectives

After we finish this set of slides you should be able to:

- Describe the motivation for and purpose of a branch predictor and branch target buffer.
- Explain why it is possible to predict branch outcomes with relatively good accuracy.
- Describe three types of branch predictor and evaluate their operation in detail.

Control Dependence (H&P 3.1)

- Intuition: An instruction X is control dependent on a branch B if whether instruction X executes is determined by the outcome of branch B.

```
if( p1 ) {  
    S1; // S1 (statement 1) control dependent on p1  
}  
if (p2) {  
    S2; // S2 control dependent on p2, not c.d. on p1  
}
```

Control dependencies lead to control hazards. Earlier, we saw three approaches to dealing with control hazards: (1) wait for the branch to execute, (2) “predict not taken” (3) delayed branches,. In this slide set, we extend the “predict not taken” approach.

Control Hazards

- Predict-not-taken. Expanded pipeline view, showing flushed instructions (assuming branch “resolved in execute”):

Taken Branch	Clock Cycle							
	1	2	3	4	5	6	7	8
BEQZ R1, Label	IF	ID	EX	MEM	WB			
branch +1 (PC+4)		IF	ID	EX	MEM	WB		
branch + 2 (PC+8)			IF	ID	EX	MEM	WB	
branch target (Label)				IF	ID	EX	MEM	WB
branch target. + 1					IF	ID	EX	MEM
branch target. + 2						IF	ID	EX

Clock cycle 1: IF stage predicts “branch+1” is the next instruction to fetch.

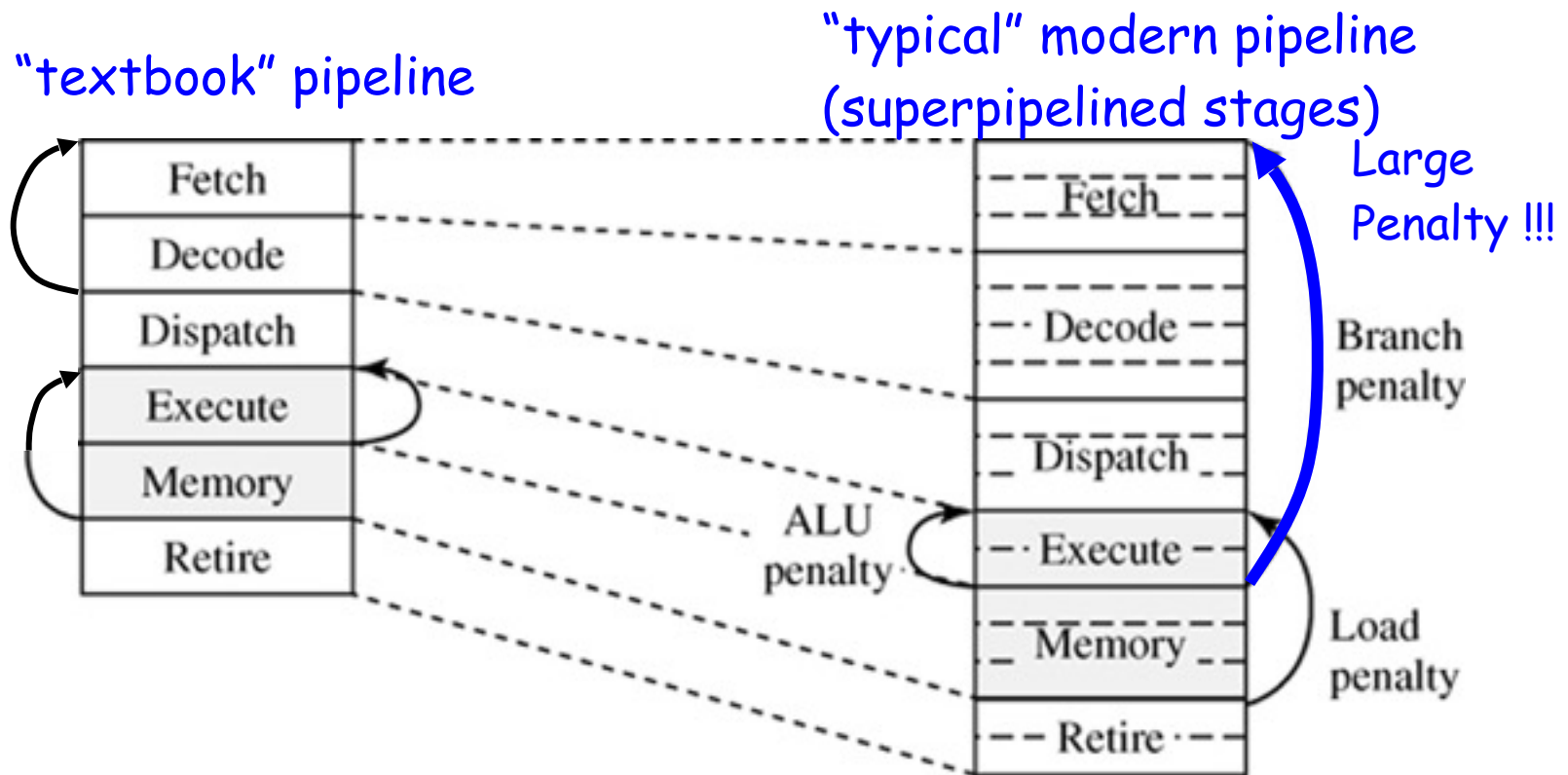
Clock cycle 3: EX stage resolves branch, instructions in ID (“branch +1”) and IF (“branch+2”) turned into “no-ops”... they are “flushed”,

Branch Prediction Impact (5-stage pipeline) – Resolved in EX

Correct Prediction	Clock Number							
	1	2	3	4	5	6	7	8
taken branch	IF	ID	EX	MEM	WB			
branch target		IF	ID	EX	MEM	WB		
branch target. + 1			IF	ID	EX	MEM	WB	
branch target. + 2				IF	ID	EX	MEM	WB

Incorrect Prediction	Clock Number							
	1	2	3	4	5	6	7	8
taken branch	IF	ID	EX	MEM	WB			
predicted next inst.		IF	ID	nop	nop	nop		
predicted next inst.+1			IF	nop	nop	nop	nop	
actual next inst.				IF	ID	EX	MEM	WB

Effect of Deeper Pipelining

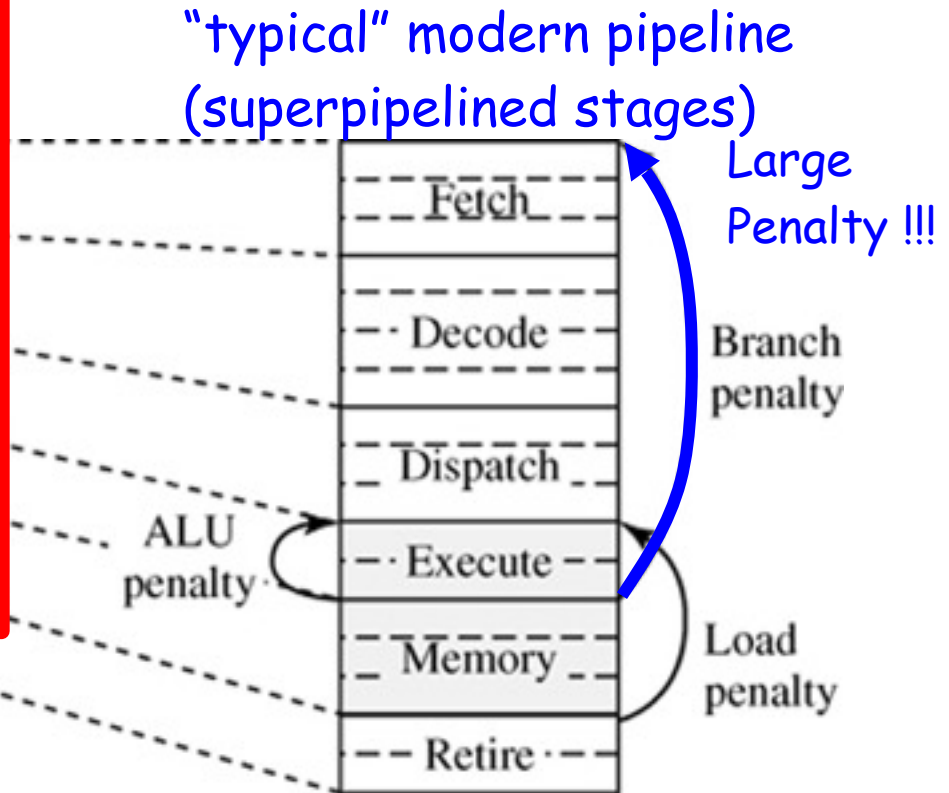


- Waiting for branch to be executed incurs large penalty (related to number of pipeline stages between fetch and execute). If we correctly predict branch outcome, this penalty is “hidden”.
- If branch prediction is wrong, we still pay the large penalty. Thus, important to build predictors that mispredict rarely.

Effect of Deeper Pipelining

Does a processor need to execute instructions out of order to benefit from predicting the correct next instruction to execute after a branch?

- A: Yes, very sure
- B: Yes, but not sure
- C: Not sure
- D: No, but not sure
- E: No, very sure



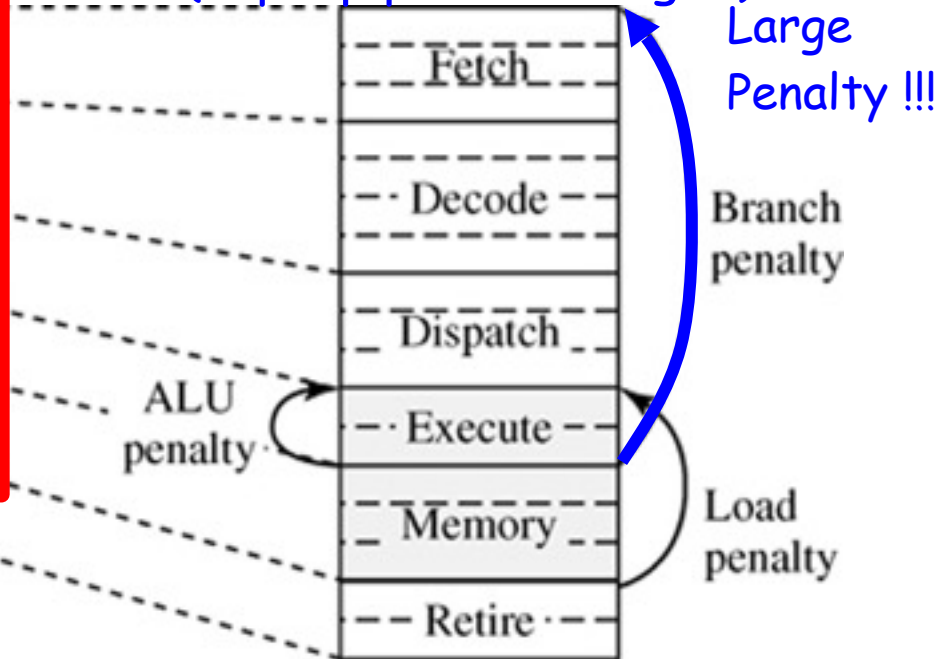
- Waiting for branch to be executed incurs large penalty (related to number of pipeline stages between fetch and execute). If we correctly predict branch outcome, this penalty is "hidden".
- If branch prediction is wrong, we still pay the large penalty. Thus, important to build predictors that mispredict rarely.

Effect of Deeper Pipelining

Does a processor need to execute instructions out of order to benefit from predicting the correct next instruction to execute after a branch?

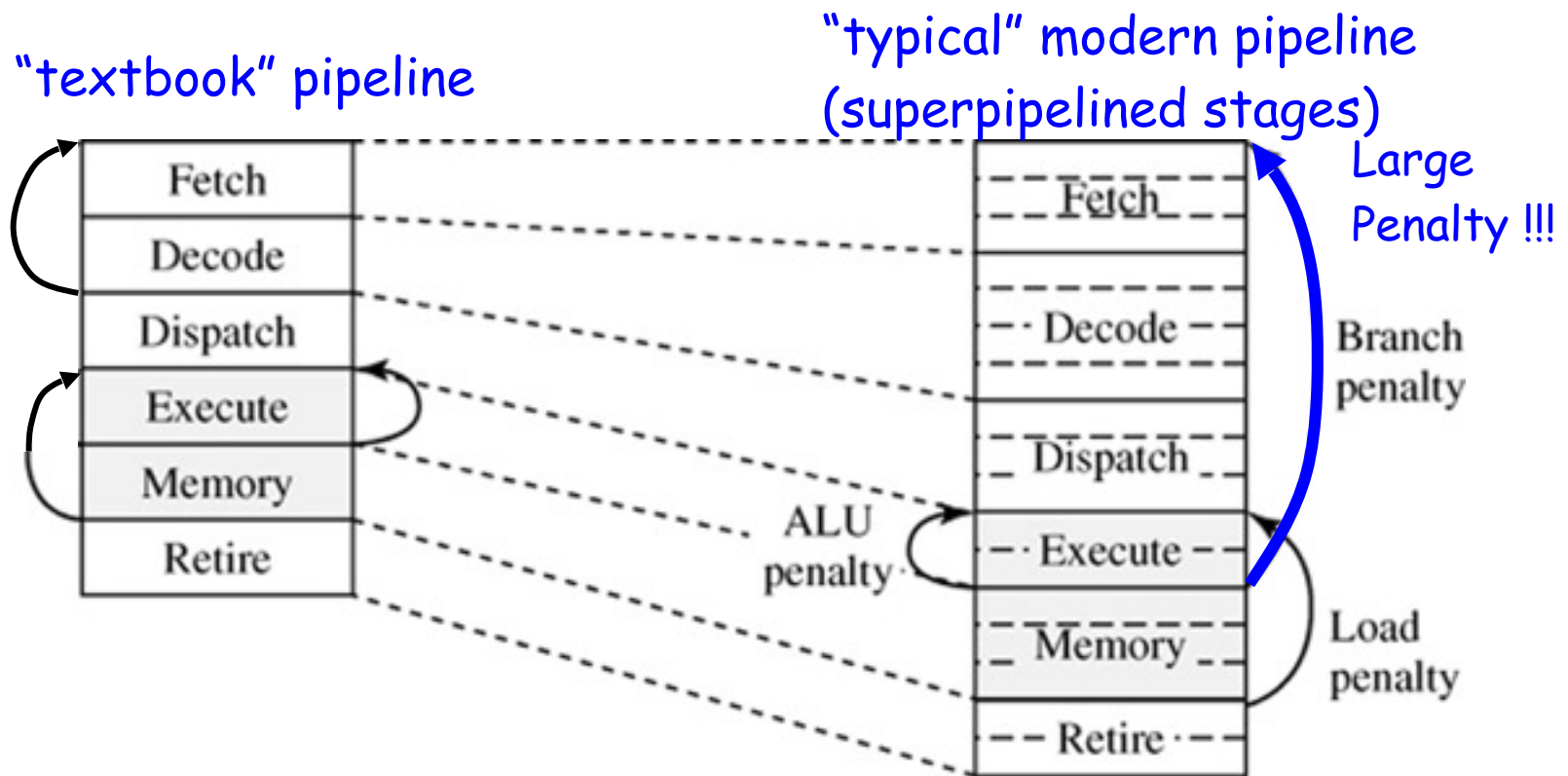
- A: Yes, very sure
- B: Yes, but not sure
- C: Not sure
- D: No, but not sure
- E: No, very sure ✓

“typical” modern pipeline
(superpipelined stages)



- Waiting for branch to be executed incurs large penalty (related to number of pipeline stages between fetch and execute). If we correctly predict branch outcome, this penalty is “hidden”.
- If branch prediction is wrong, we still pay the large penalty. Thus, important to build predictors that mispredict rarely.

Effect of Deeper Pipelining



- Waiting for branch to be executed incurs large penalty (related to number of pipeline stages between fetch and execute). If we correctly predict branch outcome, this penalty is “hidden”.
- If branch prediction is wrong, we still pay the large penalty. Thus, important to build predictors that mispredict rarely.

In five stage pipeline processor we considered earlier is the target of a branch computed before the branch instruction itself is read from the instruction memory?

- A: Yes, very sure
- B: Yes, but not sure
- C: Not sure
- D: No, but not sure
- E: No, very sure

In five stage pipeline processor we considered earlier is the target of a branch computed before the branch instruction itself is read from the instruction memory?

- A: Yes, very sure
- B: Yes, but not sure
- C: Not sure
- D: No, but not sure
- E: No, very sure ✓

Branch Instructions: Two Problems

Branch Instructions: Two Problems

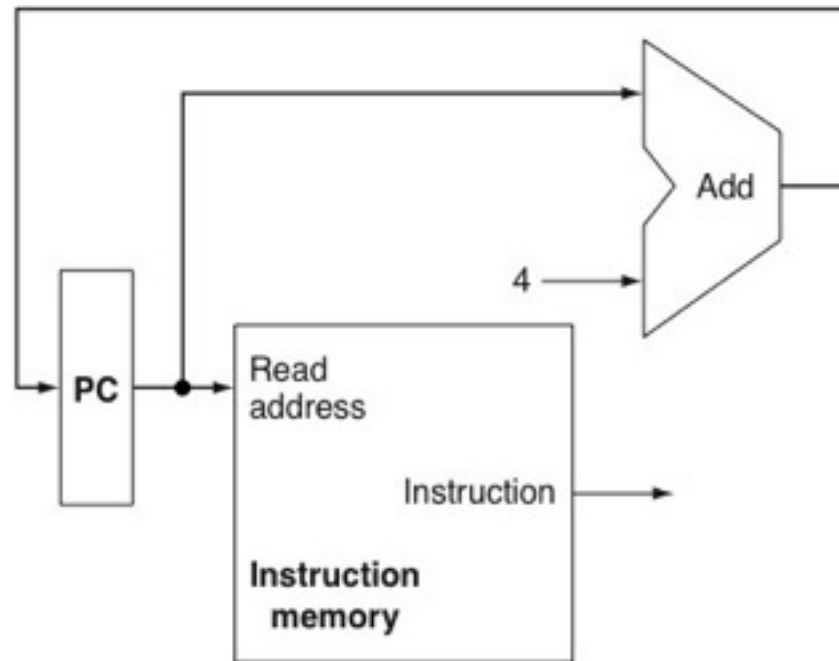
- **Problem 1: Is branch taken? (“direction”)**
 - Hardware to predict this: “Branch Predictor”
 - We will look at several designs

Branch Instructions: Two Problems

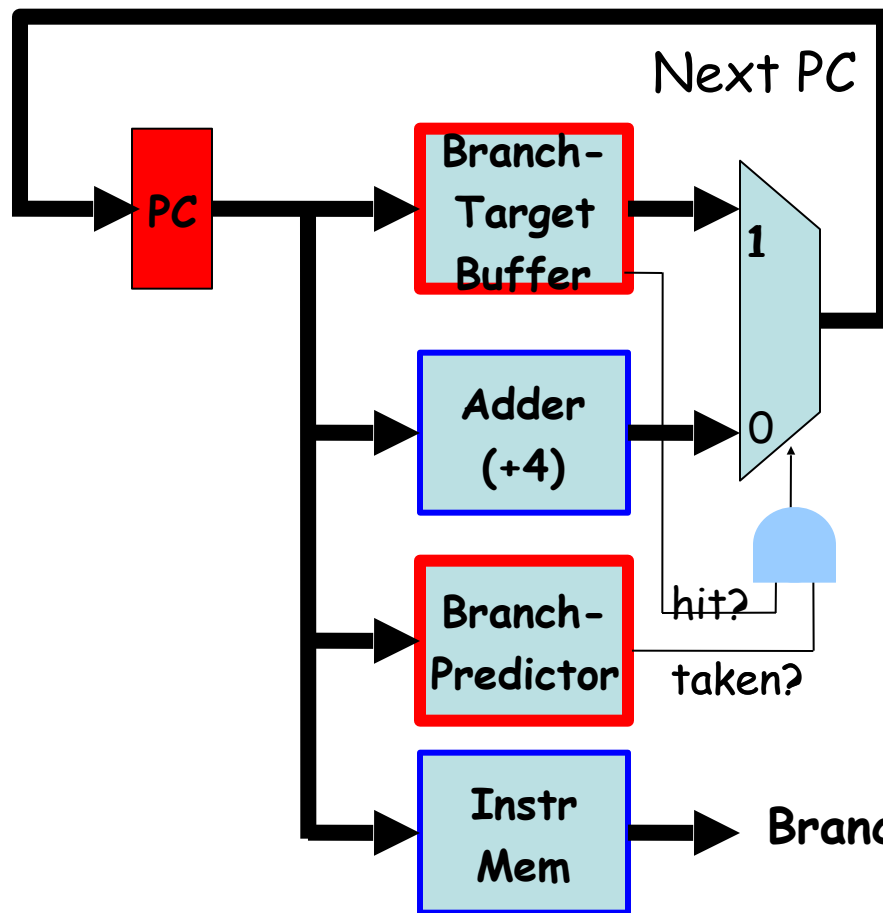
- **Problem 1: Is branch taken? (“direction”)**
 - Hardware to predict this: “Branch Predictor”
 - We will look at several designs
- **Problem 2: If taken, what is target PC?**
 - Hardware to predict this: “Branch Target Buffer”
 - We will look at one design that works fairly well.

Quick Review: How are instructions fetched?

Recall, from Slide Set 4:



Branches Predicted in Fetch Stage



Fetch stage (IF)

Two new hardware structures:

1. Branch Target Buffer
2. Branch Predictor

Our focus:

What is the motivation?

What do they do?

How do we use them?

How do we design them?

Branch Instruction

Decode stage (ID)...

“Next PC” based upon *prediction* of branch outcome.

Predicting the future...

Predicting the future...



Predicting the future...

- Method #1
 - Crystal ball!!



Predicting the future...

- Method #1
 - ~~Crystal ball!!!~~ (doesn't work)



Predicting the future...



- Method #1
 - ~~Crystal ball!!!~~ (doesn't work)
- Method #2
 - Study history since “history repeats itself”.
 - This works amazingly well in computer architecture
 - Past branch “behaviour” => excellent predictor of future branch “behaviour”.

Example Prediction Problem

Loop: ...

...

DSUBI R1,R1,#1

BNEZ R1,Loop ; taken 9 times, not taken once

Observations: For this code, **last branch outcome** (taken, not taken) a good (but not perfect) predictor of next branch outcome. This turns out to be true for many branches.

design hardware to use observation?

1-Bit Branch Predictor

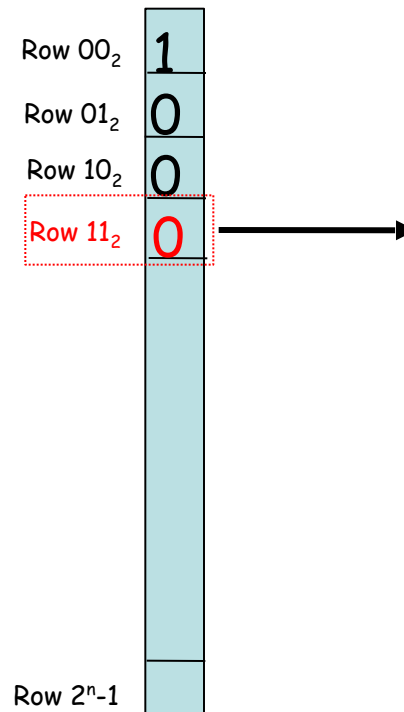
1-Bit Branch Predictor



1-Bit Branch Predictor

Row 00_2	1
Row 01_2	0
Row 10_2	0
Row 11_2	0
Row 2^n-1	

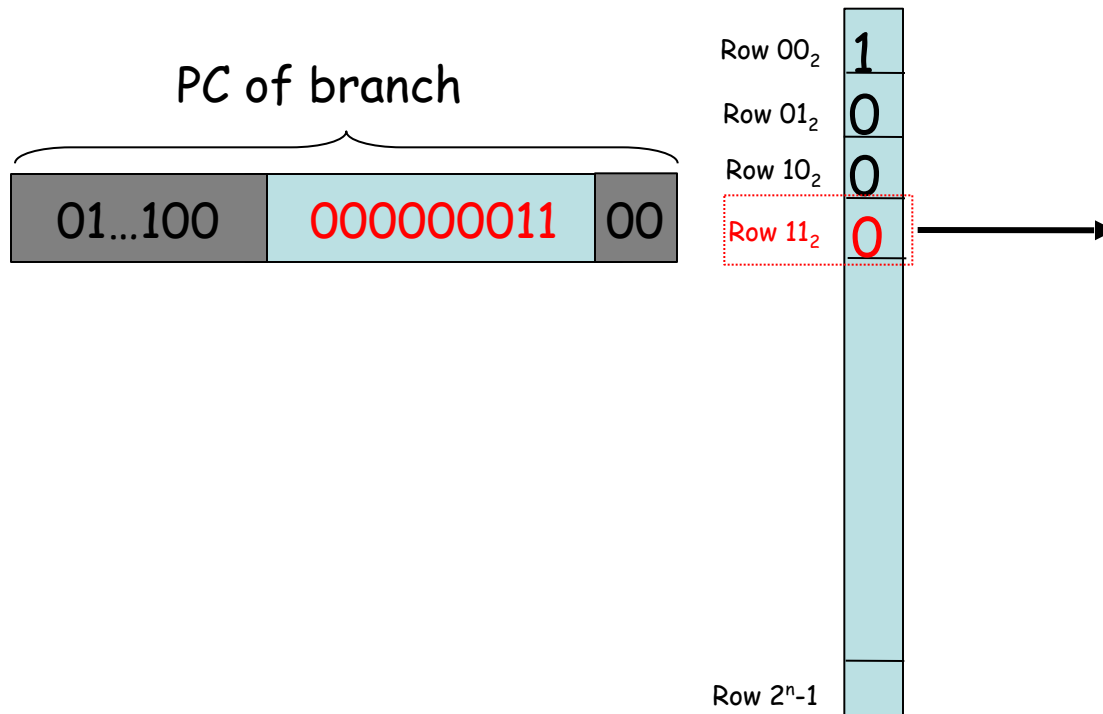
1-Bit Branch Predictor



Output ("0") is previous
actual outcome of branch:

- "1" \Rightarrow last time this branch was "taken" (T)
- "0" \Rightarrow last time, this branch was "not taken" (NT)

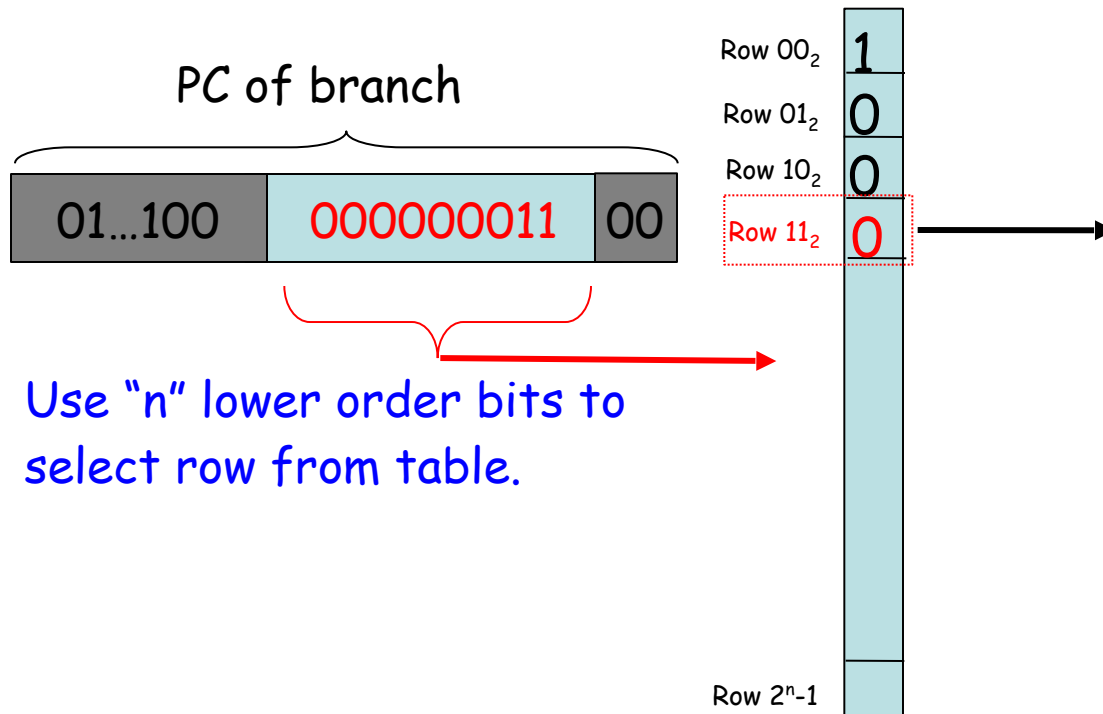
1-Bit Branch Predictor



Output ("0") is previous
actual outcome of branch:

- "1" => last time this branch was "taken" (T)
- "0" => last time, this branch was "not taken" (NT)

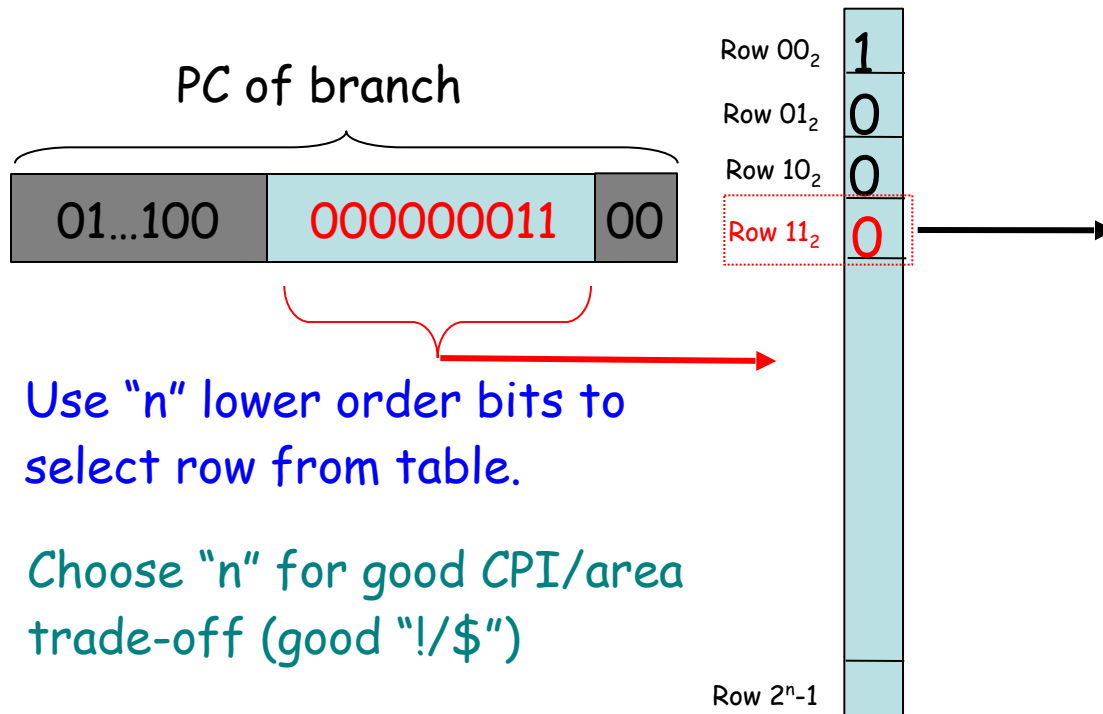
1-Bit Branch Predictor



Output ("0") is previous actual outcome of branch:

- "1" => last time this branch was "taken" (T)
- "0" => last time, this branch was "not taken" (NT)

1-Bit Branch Predictor



Output ("0") is previous actual outcome of branch:

- "1" => last time this branch was "taken" (T)
- "0" => last time, this branch was "not taken" (NT)

Operation of 1-bit Branch Predictor

Operation of 1-bit Branch Predictor

- Step 1: Making a Prediction:
 - Take n-bits of “PC” and use it to select a row in table. If entry in that row is “1”, prediction is “taken” and if entry is “0”, prediction is “not taken”.
 - PC here is address of the branch instruction itself.

Operation of 1-bit Branch Predictor

- Step 1: Making a Prediction:
 - Take n-bits of “PC” and use it to select a row in table. If entry in that row is “1”, prediction is “taken” and if entry is “0”, prediction is “not taken”.
 - PC here is address of the branch instruction itself.
- Step 2: Updating Predictor
 - When we know the correct outcome of the branch update the entry in the table to be “1” if the actual outcome was “taken”, and to “0” if the actual outcome was “not taken”

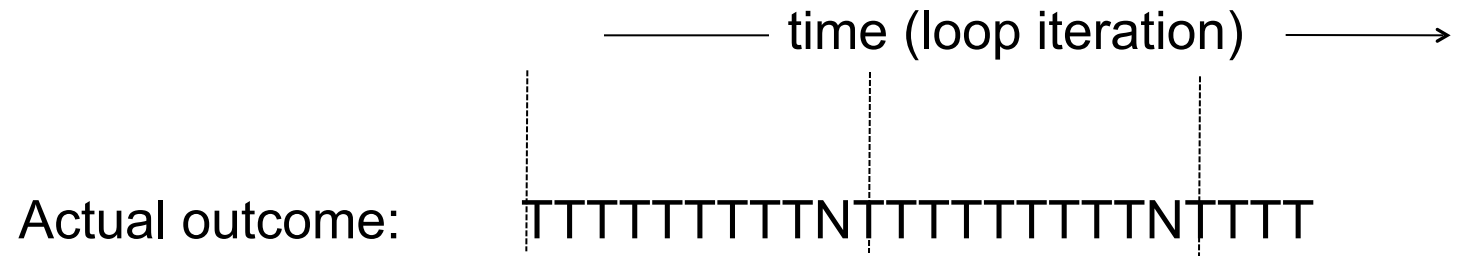
Aside: Hash Tables

Some of you might have learned about “hash tables” in one of your programming courses. You might note that the hardware used for branch prediction does something similar. (If you don’t remember, don’t worry about it.)

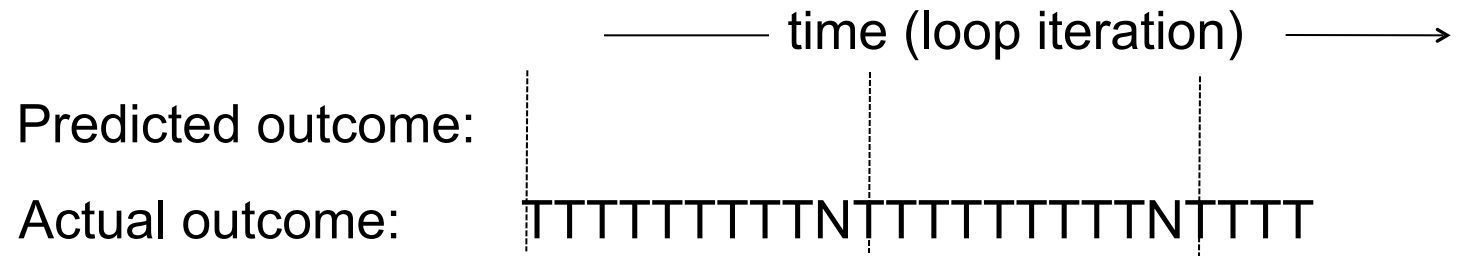
Example, cont'd

```
Loop:  ...  
      ...  
      DSUBI R1,R1,#1  
      BNEZ  R1,Loop
```

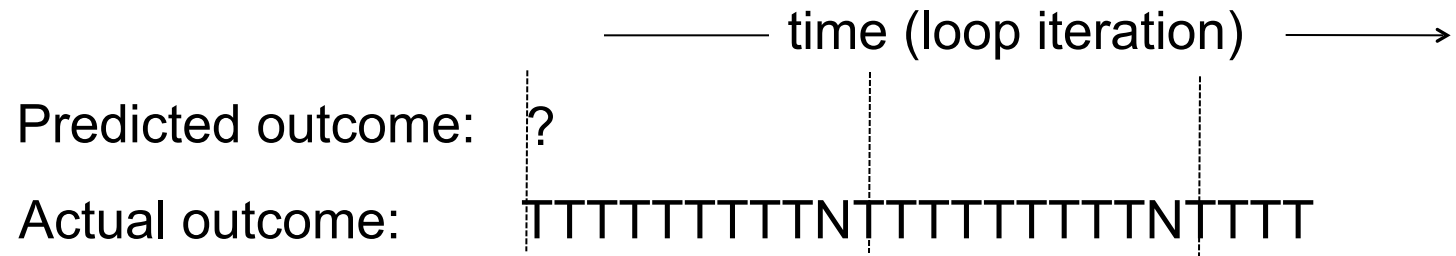
1-bit Branch Predictor



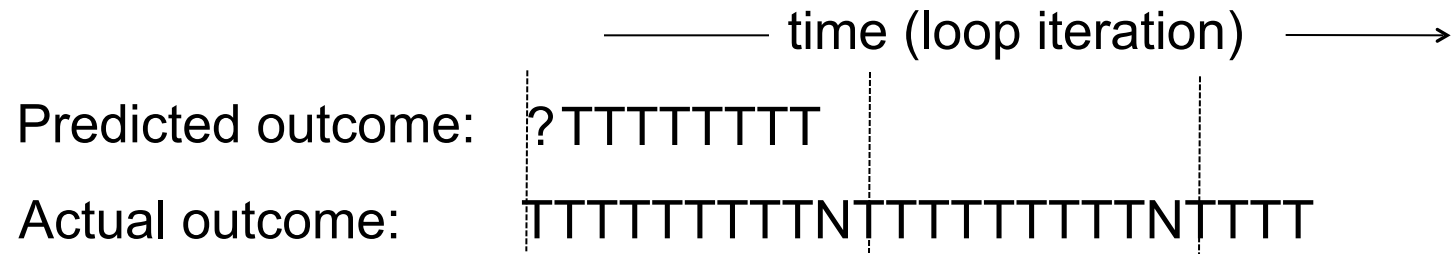
1-bit Branch Predictor



1-bit Branch Predictor



1-bit Branch Predictor



1-bit Branch Predictor

————— time (loop iteration) —————→

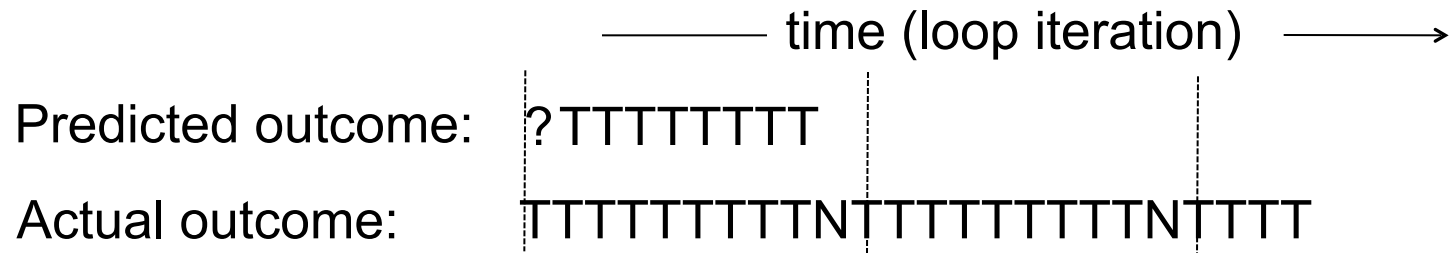
Predicted outcome: ? T T T T T T T T T

Actual outcome: T T T T T T T T T N T T T T T T T T T N T T T T

What happens if branch taken 9 times in a row then not taken and we used 1-bit branch predictor?

- A: Misprediction
- B: Correct prediction
- C: Not sure

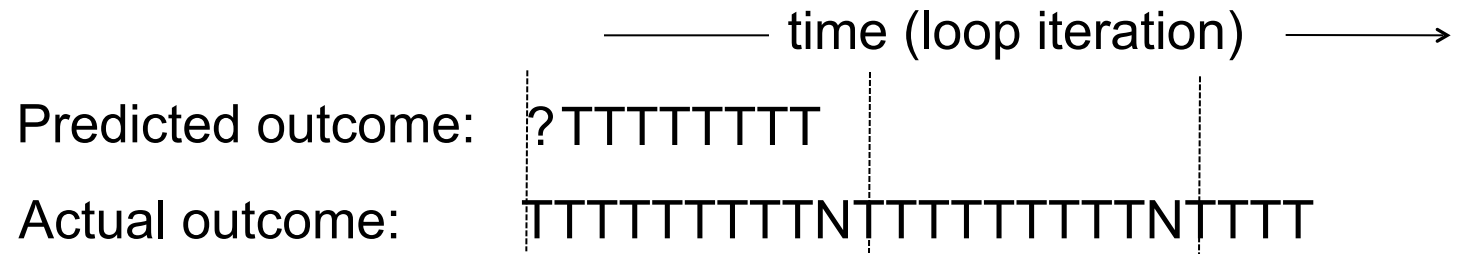
1-bit Branch Predictor



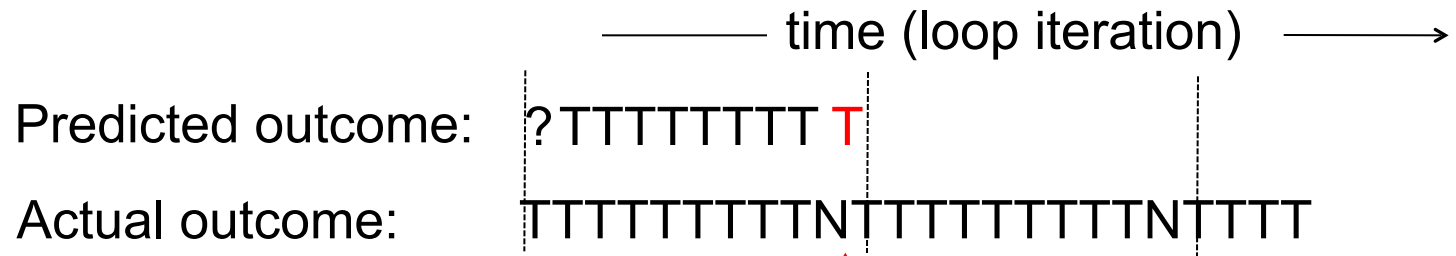
What happens if branch taken 9 times in a row then not taken and we used 1-bit branch predictor?

- A: Misprediction ✓
- B: Correct prediction
- C: Not sure

1-bit Branch Predictor

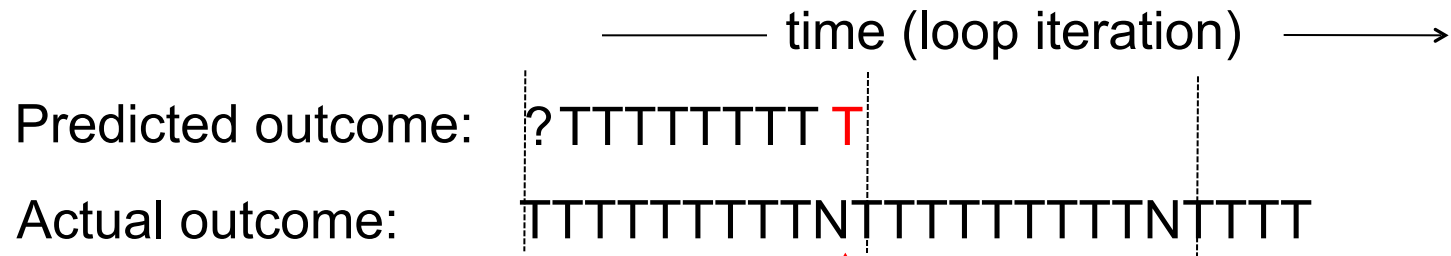


1-bit Branch Predictor



- Mispredicts last loop iteration

1-bit Branch Predictor

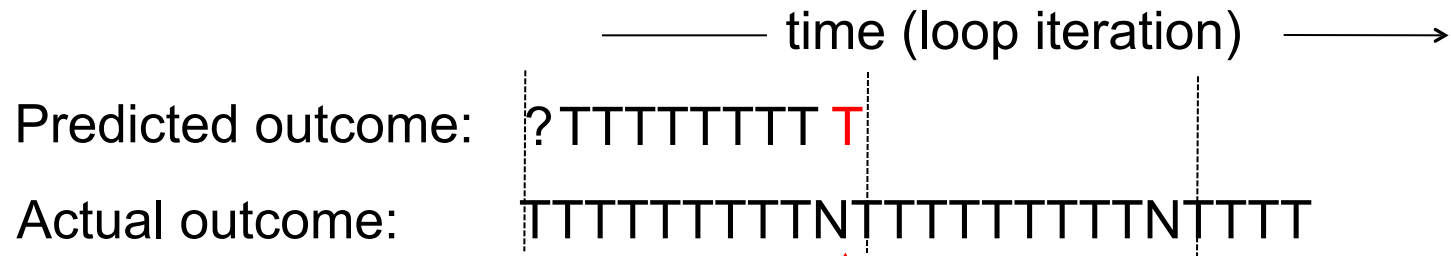


- Mispredicts last loop iteration

What happens if we encounter this loop many times using the 1-bit branch predictor?

- A: 1 Misprediction each time loop is encountered
- B: 2 Mispredictions each time loop is encountered
- C: 9 Mispredictions each time loop is encountered
- D: 10 Mispredictions each time is encountered
- E: Not sure

1-bit Branch Predictor

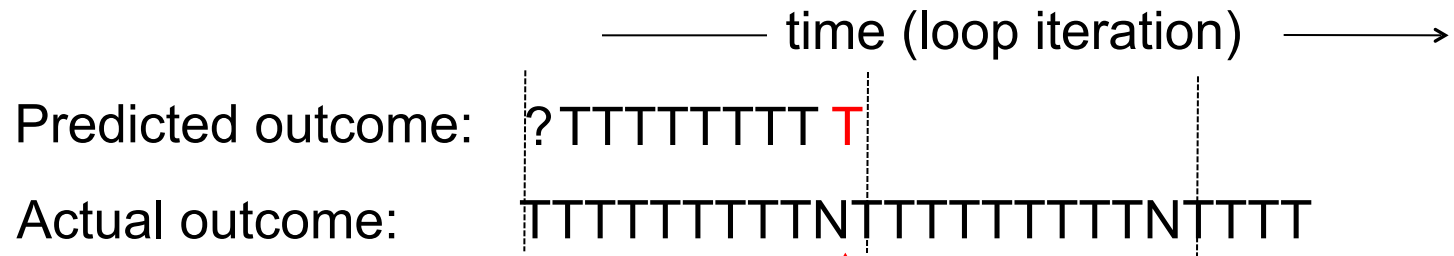


- Mispredicts last loop iteration

What happens if we encounter this loop many times using the 1-bit branch predictor?

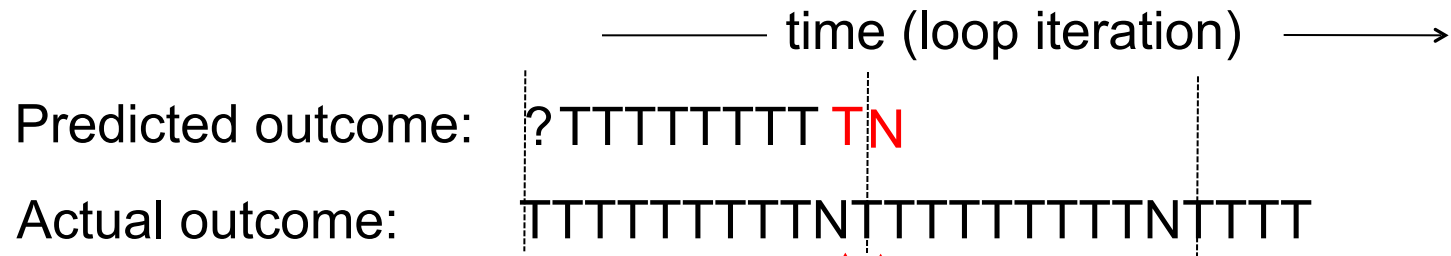
- A: 1 Misprediction each time loop is encountered
- B: 2 Mispredictions each time loop is encountered ✓
- C: 9 Mispredictions each time loop is encountered
- D: 10 Mispredictions each time is encountered
- E: Not sure

1-bit Branch Predictor



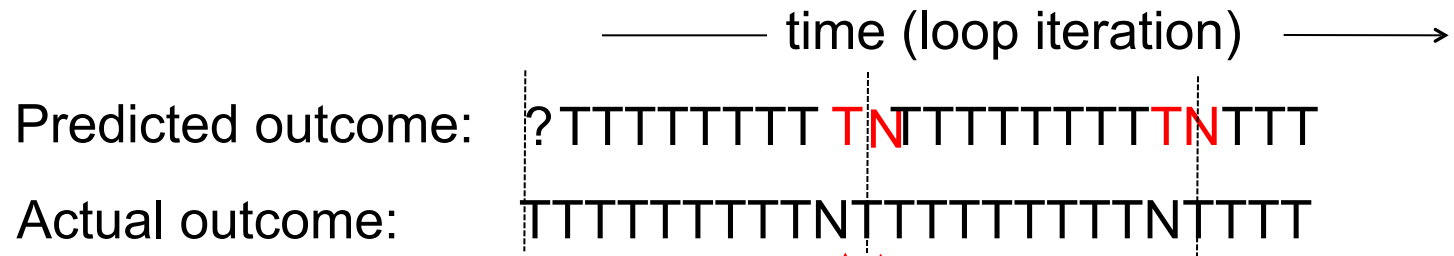
- Mispredicts last loop iteration

1-bit Branch Predictor



- Mispredicts last loop iteration
- Again when see loop again

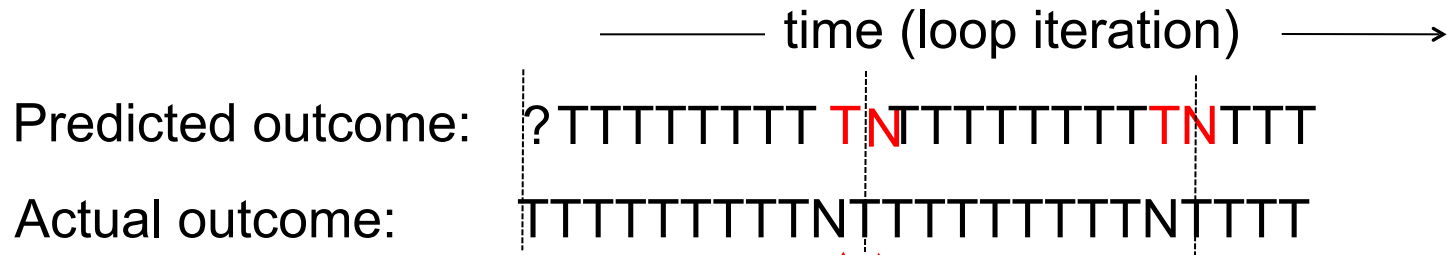
1-bit Branch Predictor



- Mispredicts last loop iteration
- Again when see loop again

1-bit Branch Predictor

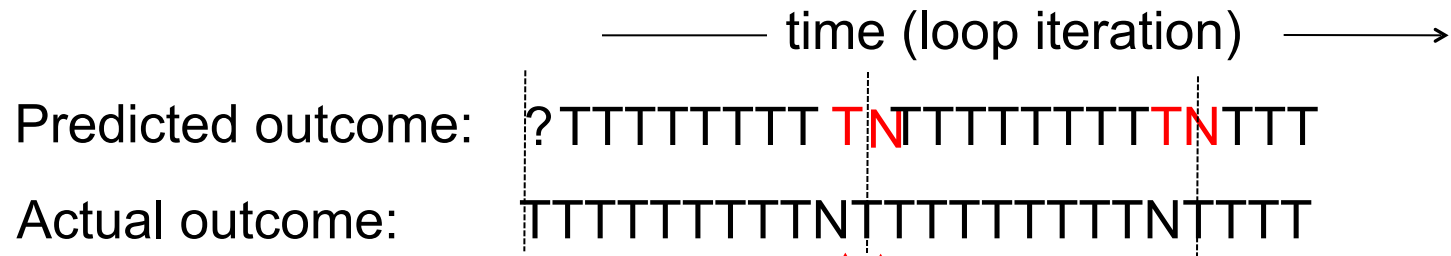
Problem: A branch that is almost always **taken** (**not-taken**) **will mispredict twice** each time branch is **not-taken** (**taken**).



- Mispredicts last loop iteration
- Again when see loop again

1-bit Branch Predictor

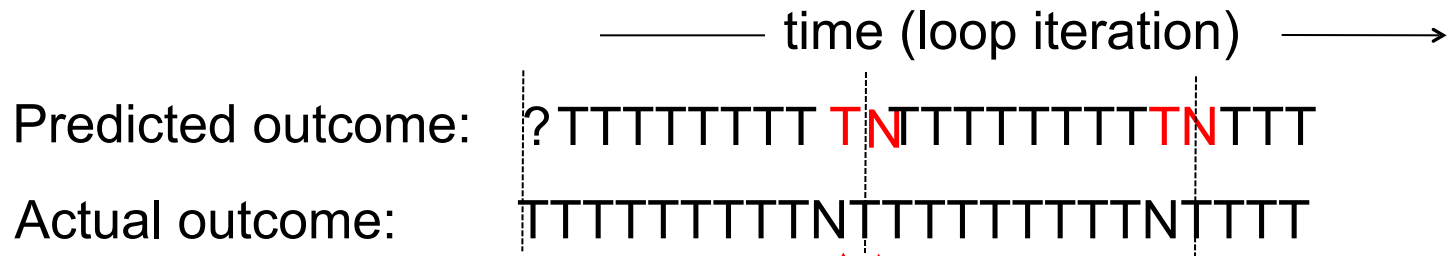
Problem: A branch that is almost always **taken** (**not-taken**) **will mispredict twice** each time branch is **not-taken** (**taken**).



- Mispredicts last loop iteration
- Again when see loop again

1-bit Branch Predictor

Problem: A branch that is almost always **taken** (**not-taken**) **will mispredict twice** each time branch is **not-taken** (**taken**).



- Mispredicts last loop iteration
- Again when see loop again

ideas to get rid of mispredictions?

2-bit Branch Predictor

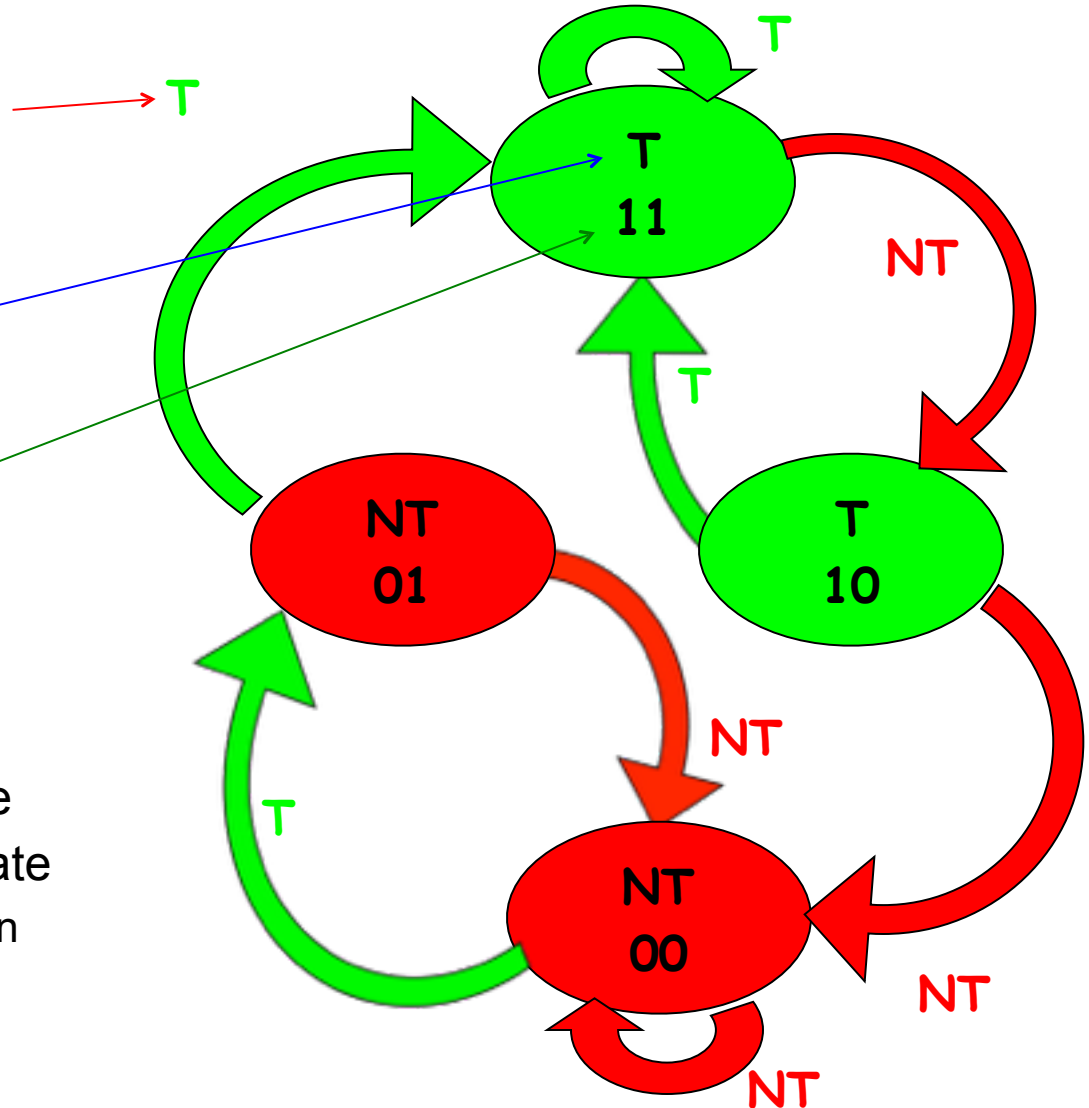
Transitions determined by
actual outcome of branch

T= "Taken"/ NT="not taken"

Current state provides
prediction:

T= "Taken"/ NT="not taken"

State encoding uses
two bits (encoding
is arbitrary)



Observations:

- Repeating T stays in '11' state
- Repeating NT stays in '00' state
- Two-in-a-row to change prediction
 - (T,NT) won't change prediction
 - (NT,T) won't change prediction

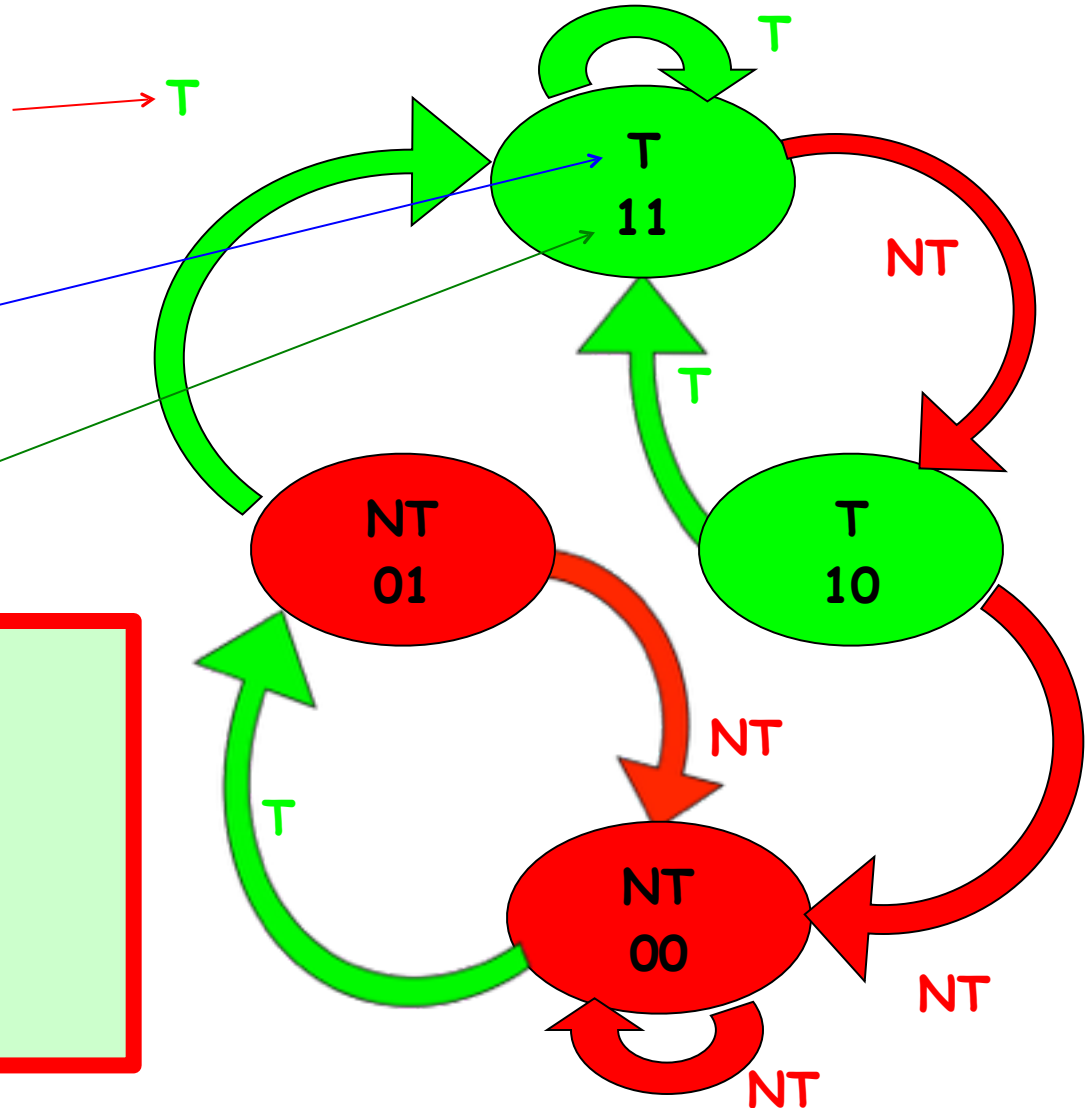
2-bit Branch Predictor

Transitions determined by
actual outcome of branch
T= "Taken"/ NT="not taken"

Current state provides
prediction:

T= "Taken"/ NT="not taken"

State encoding uses
two bits (encoding
is arbitrary)



If in state 01, what is
prediction for next
branch?

- A: Taken
- B: Not taken
- C: Not sure

2-bit Branch Predictor

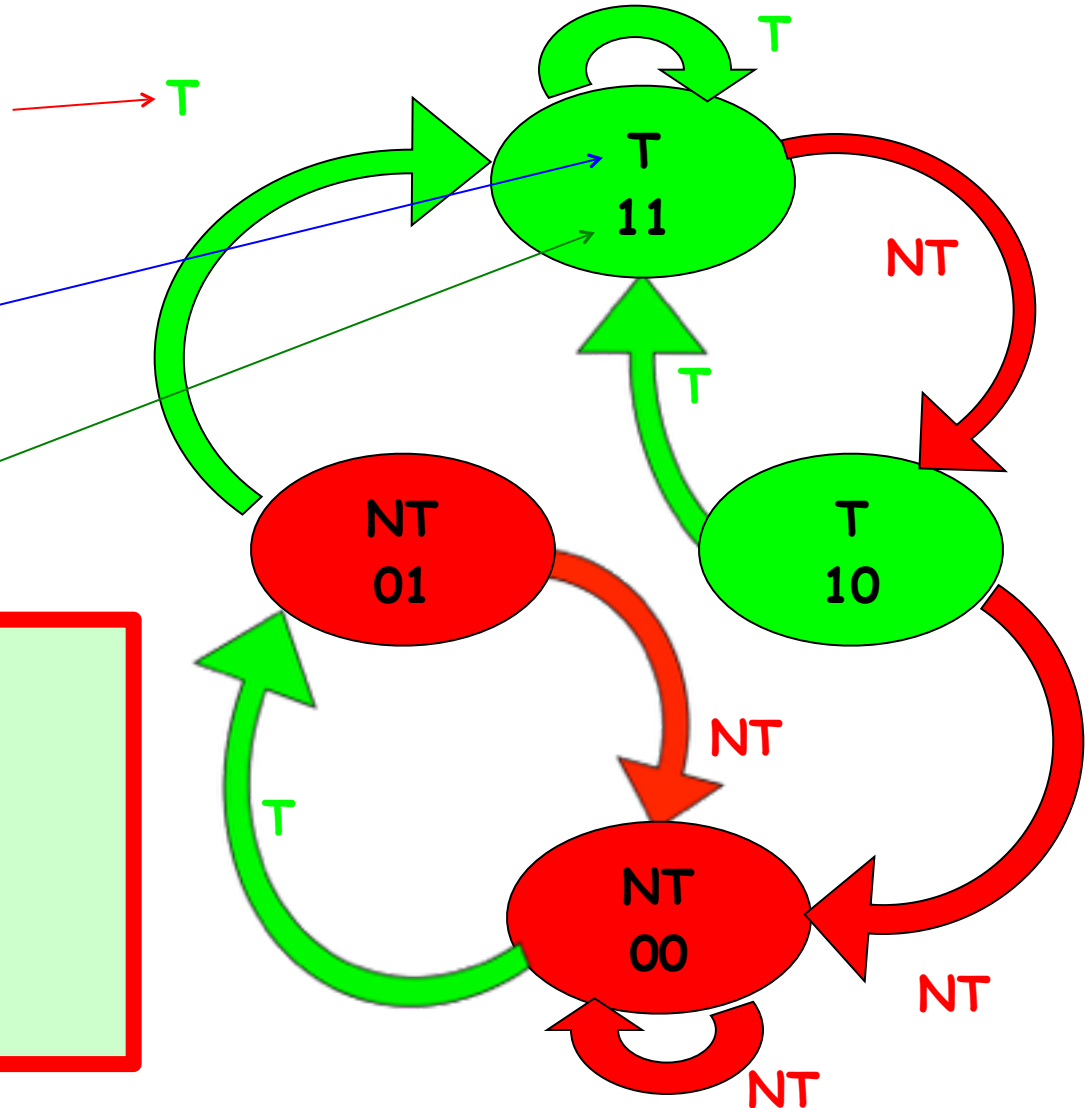
Transitions determined by
actual outcome of branch

T= "Taken"/ NT="not taken"

Current state provides
prediction:

T= "Taken"/ NT="not taken"

State encoding uses
two bits (encoding
is arbitrary)



If in state 01, what is
prediction for next
branch?

A: Taken

B: Not taken ✓

C: Not sure

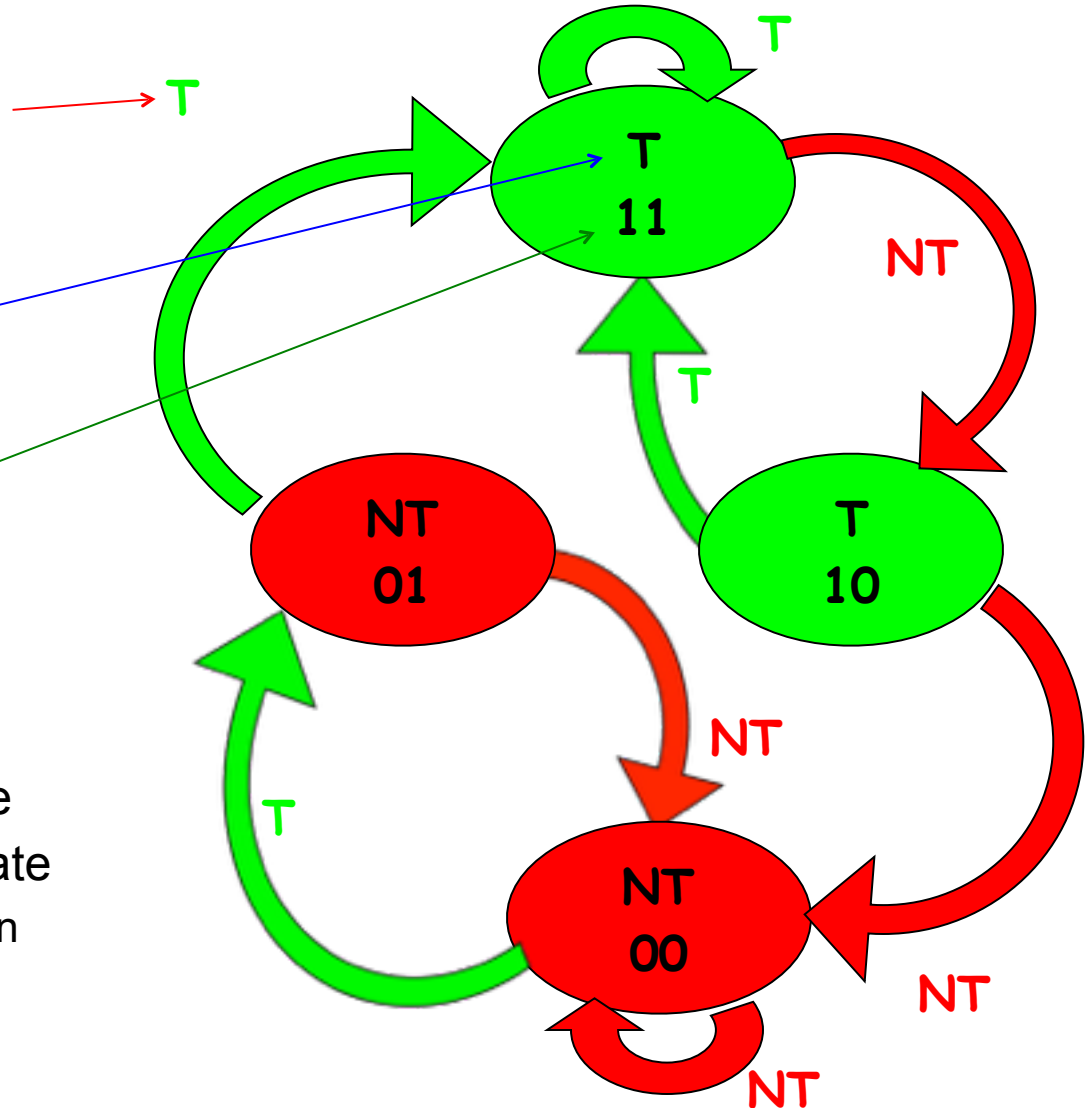
2-bit Branch Predictor

Transitions determined by
actual outcome of branch
T= "Taken"/ NT="not taken"

Current state provides
prediction:

T= "Taken"/ NT="not taken"

State encoding uses
two bits (encoding
is arbitrary)



Observations:

- Repeating T stays in '11' state
- Repeating NT stays in '00' state
- Two-in-a-row to change prediction
 - (T,NT) won't change prediction
 - (NT,T) won't change prediction

2-bit Branch Predictor

Transitions determined by
actual outcome of branch
T= "Taken"/ NT="not taken"

Current state provides
prediction:

T= "Taken"/ NT="not taken"

State encoding uses
two bits (encoding
is arbitrary)

If in state 01, and
predicted outcome is not-
taken but actual branch
outcome is taken, next
state is:

A: 00

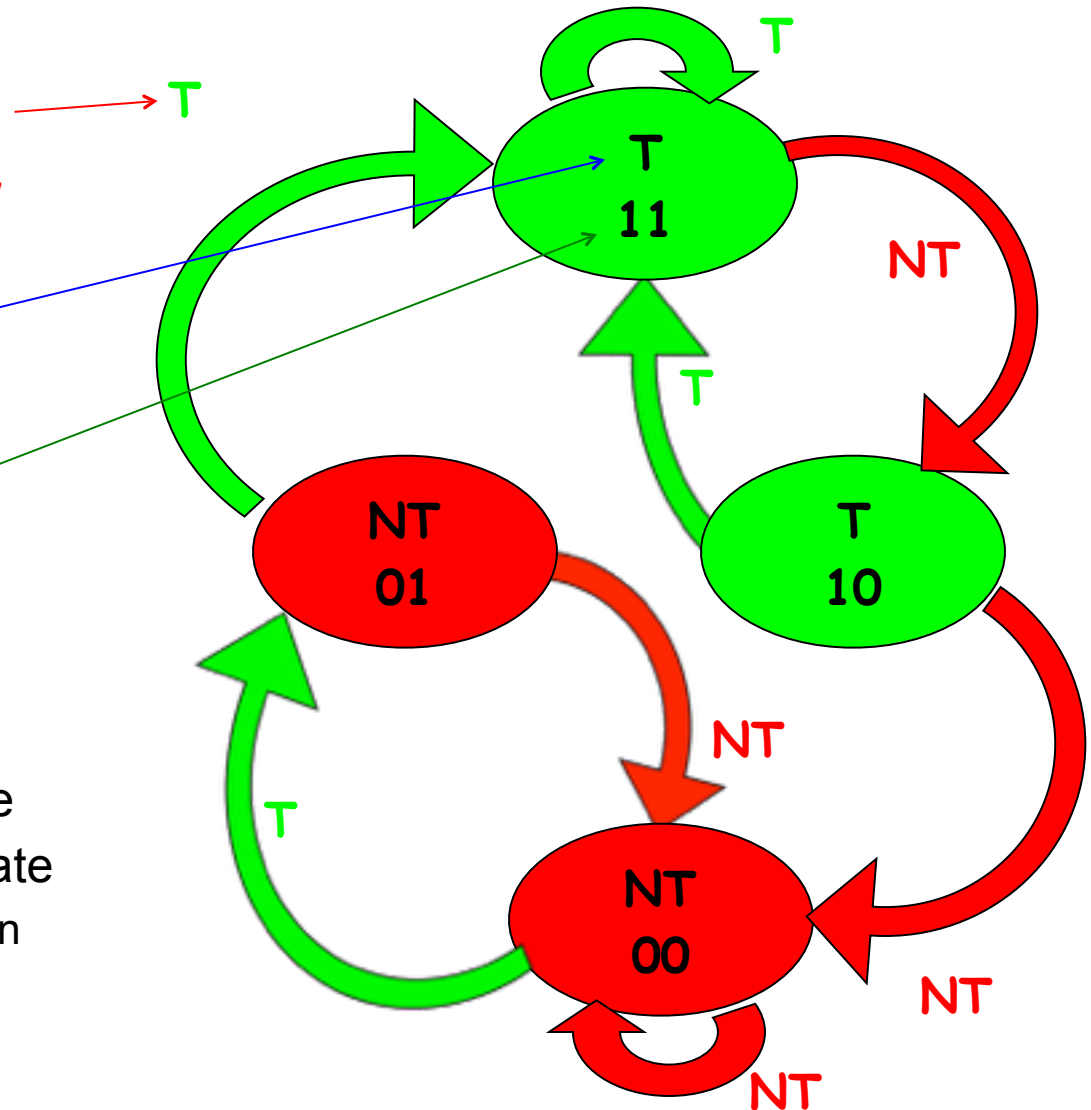
B: 01

C: 10

D: 11

E: NT

'11' state
n '00' state
prediction
prediction
prediction



2-bit Branch Predictor

Transitions determined by
actual outcome of branch
 T= "Taken"/ NT="not taken"

Current state provides
prediction:

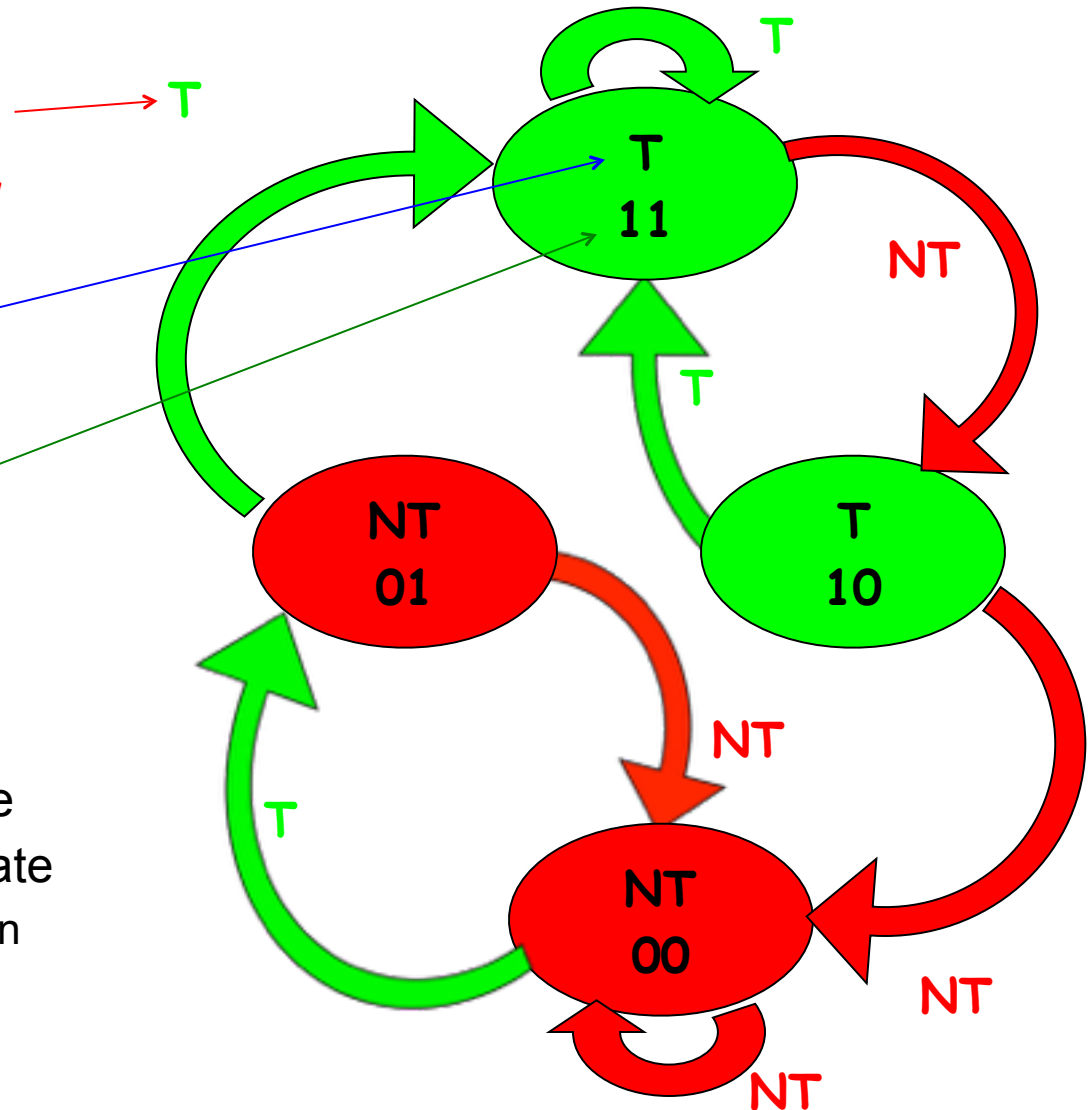
T= "Taken"/ NT="not taken"

State encoding uses
 two bits (encoding
 is arbitrary)

If in state 01, and
 predicted outcome is not-
 taken but actual branch
 outcome is taken, next
 state is:

'11' state
 in '00' state
 prediction
 prediction
 prediction

A: 00
 B: 01
 C: 10
 D: 11 ✓
 E: NT



2-bit Branch Predictor

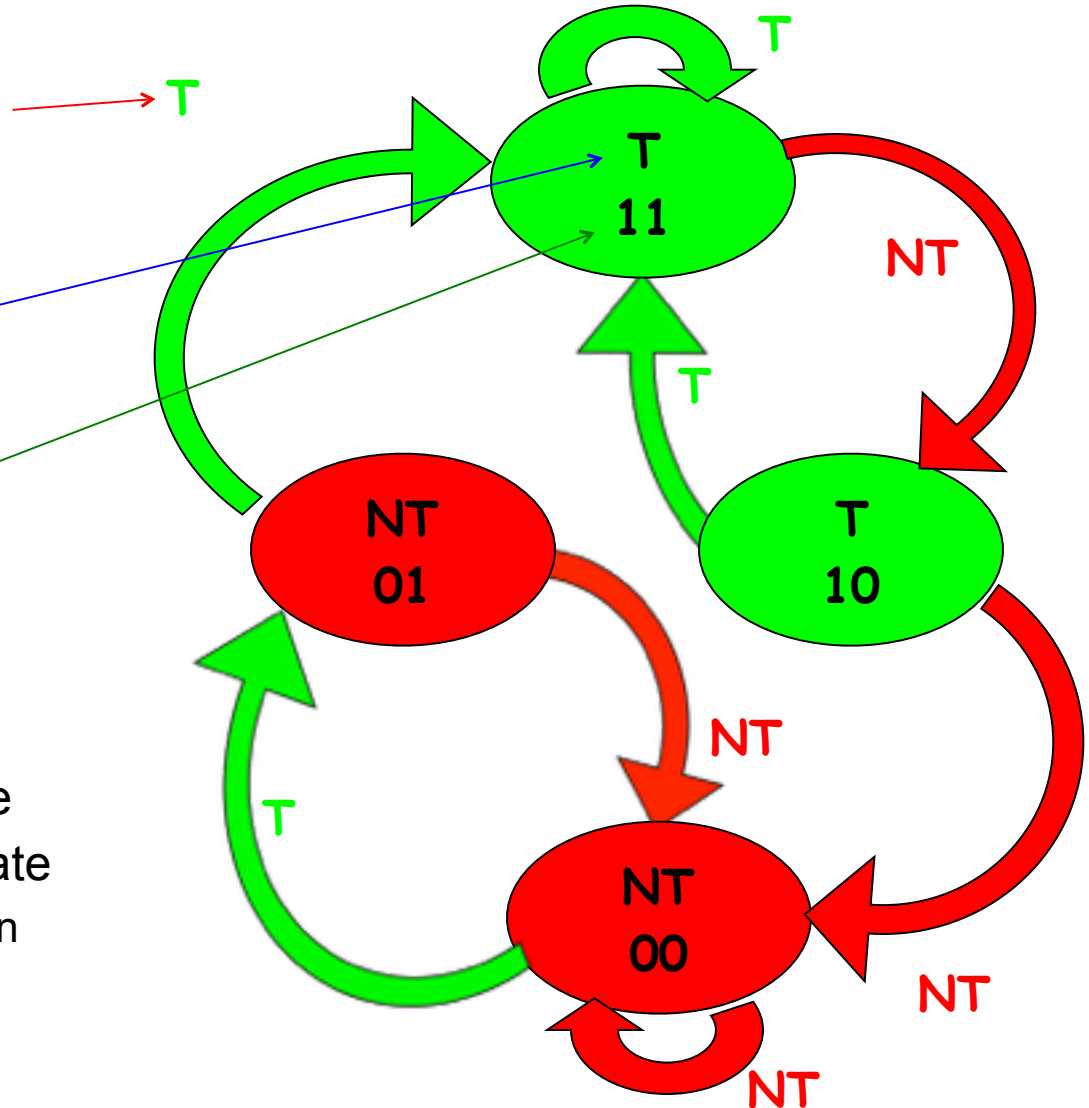
Transitions determined by
actual outcome of branch

T= "Taken"/ NT="not taken"

Current state provides
prediction:

T= "Taken"/ NT="not taken"

State encoding uses
two bits (encoding
is arbitrary)

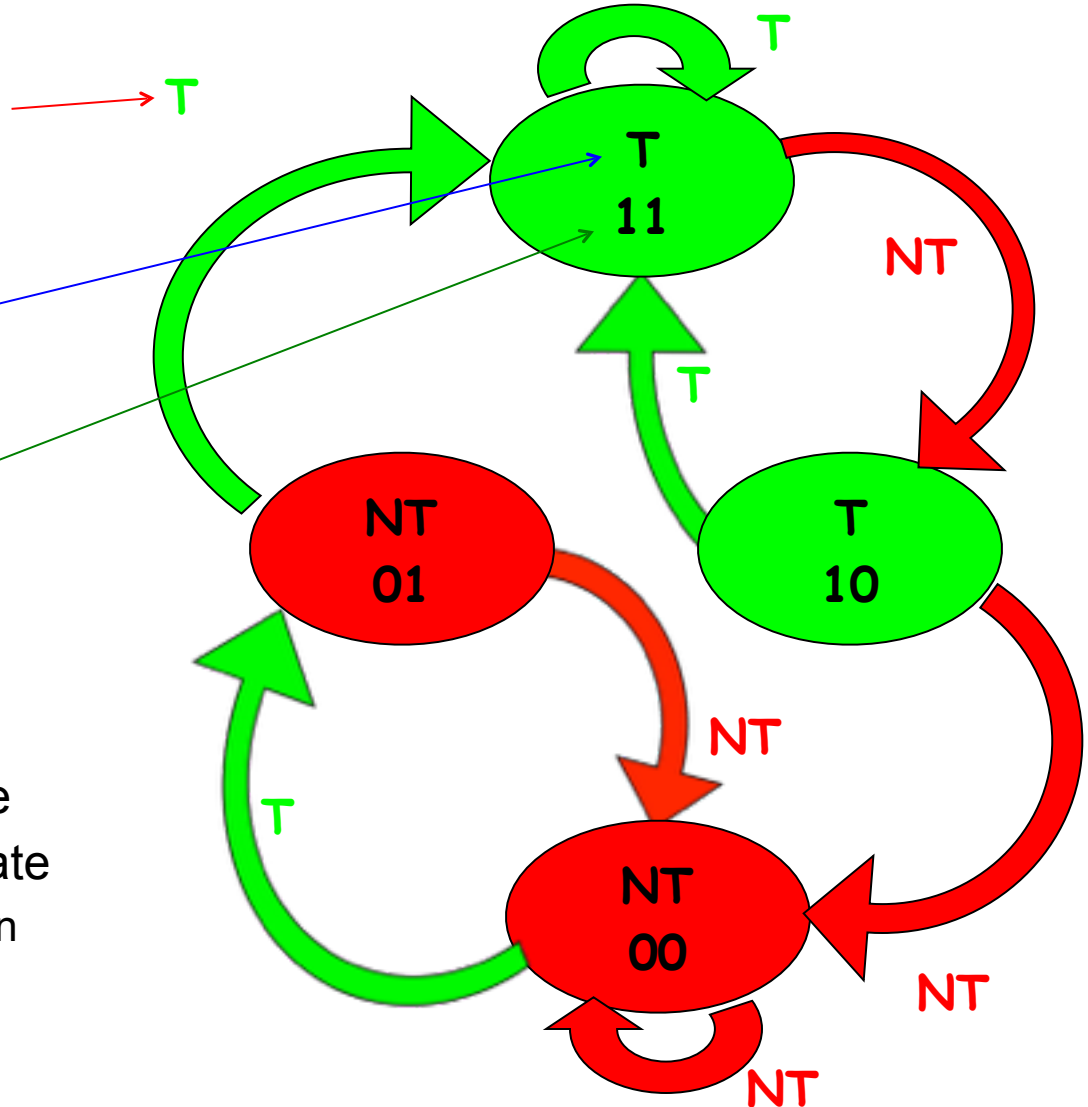


Observations:

- Repeating T stays in '11' state
- Repeating NT stays in '00' state
- Two-in-a-row to change prediction
 - (T,NT) won't change prediction
 - (NT,T) won't change prediction

2-bit Branch Predictor

Transitions determined by
actual outcome of branch
T= "Taken"/ NT="not taken"



Does this state machine
represent a shift register?

- A: Yes
- B: No
- C: Not sure

- Observations:
 - Repeating T stays in '11' state
 - Repeating NT stays in '00' state
 - Two-in-a-row to change prediction
 - (T,NT) won't change prediction
 - (NT,T) won't change prediction

2-bit Branch Predictor

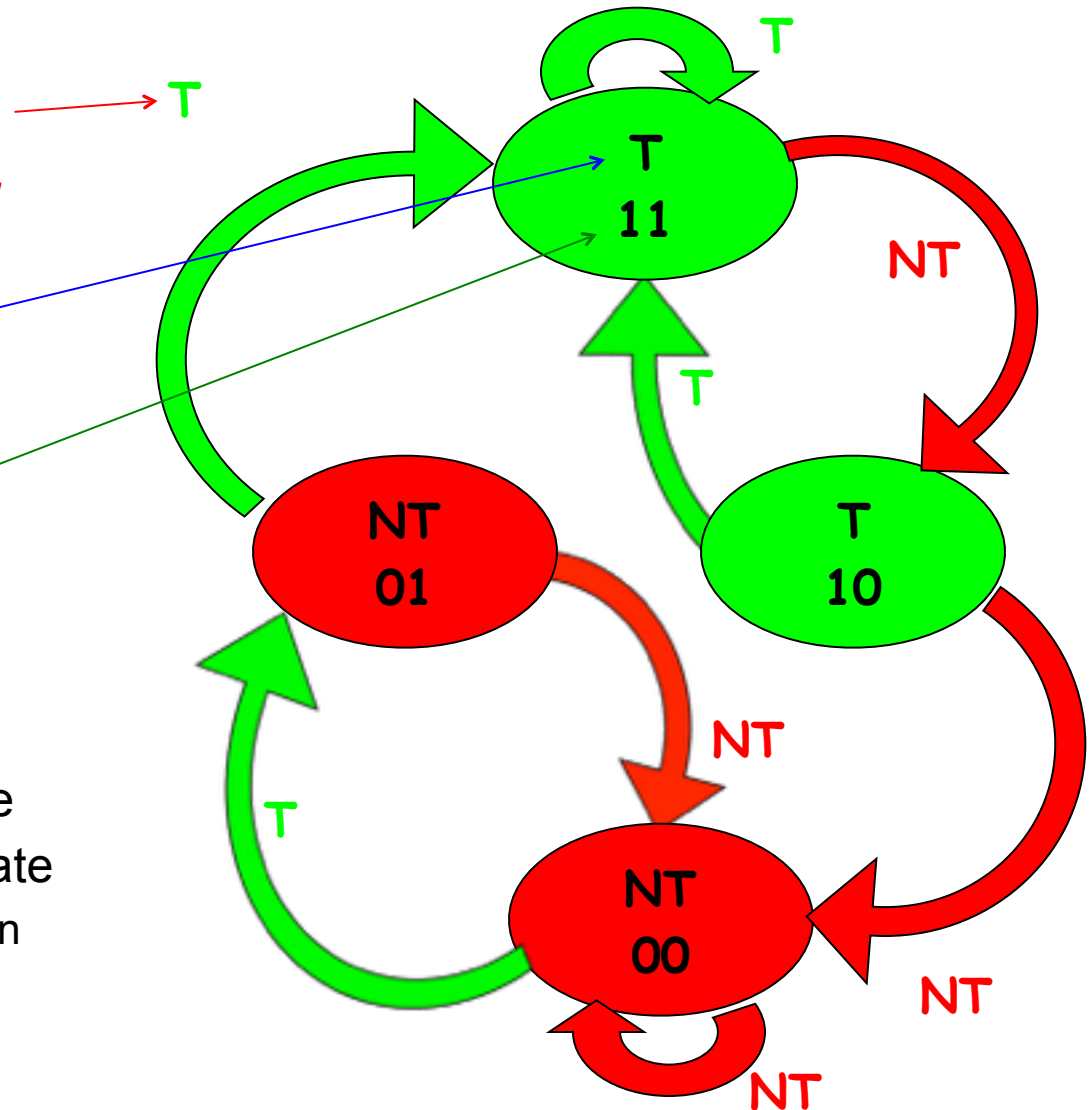
Transitions determined by
actual outcome of branch
T= "Taken"/ NT="not taken"

Does this state machine
represent a shift register?

- A: Yes
- B: No ✓
- C: Not sure

• Observations:

- Repeating T stays in '11' state
- Repeating NT stays in '00' state
- Two-in-a-row to change prediction
 - (T,NT) won't change prediction
 - (NT,T) won't change prediction



2-bit Branch Predictor

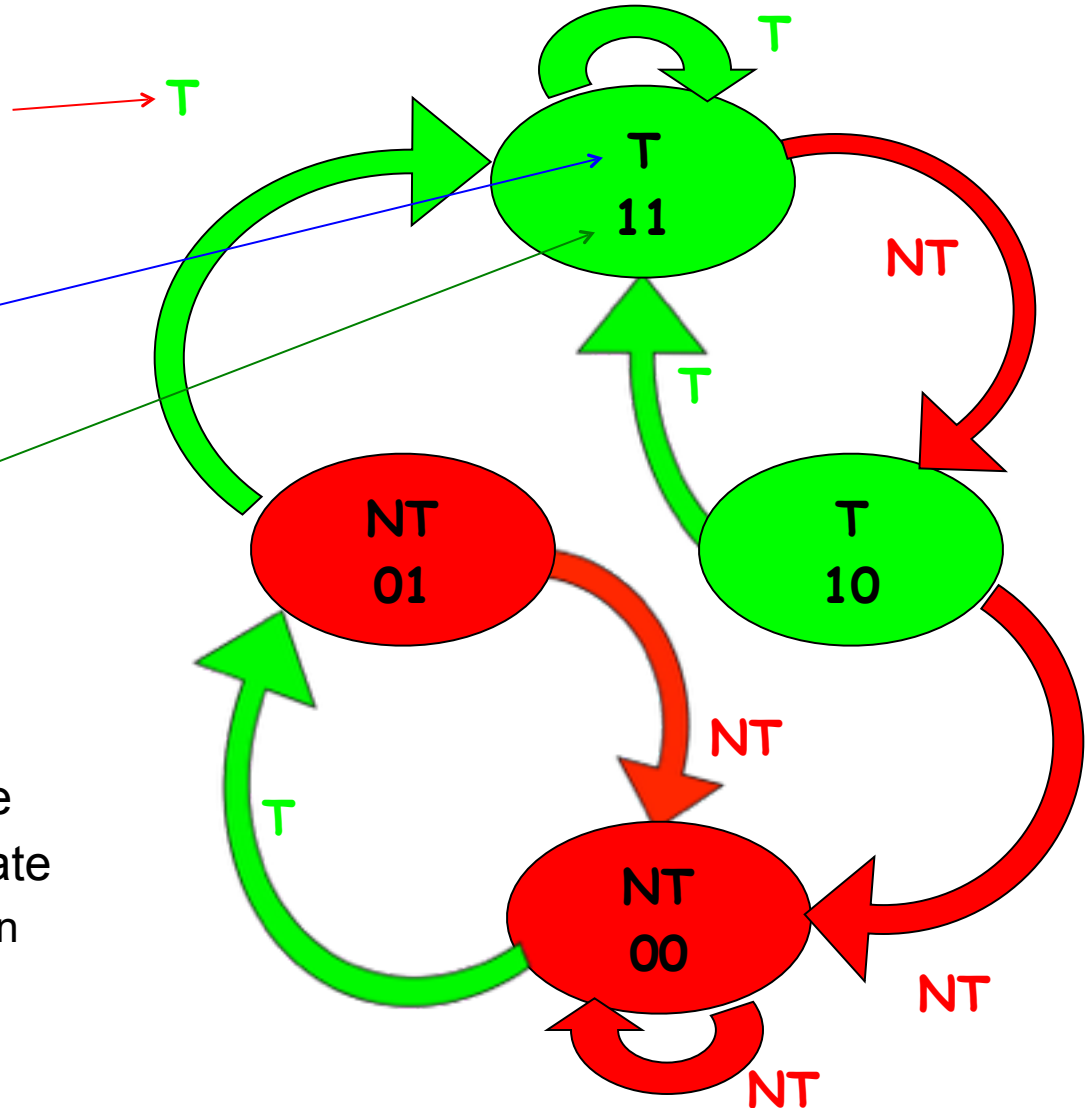
Transitions determined by
actual outcome of branch

T= "Taken"/ NT="not taken"

Current state provides
prediction:

T= "Taken"/ NT="not taken"

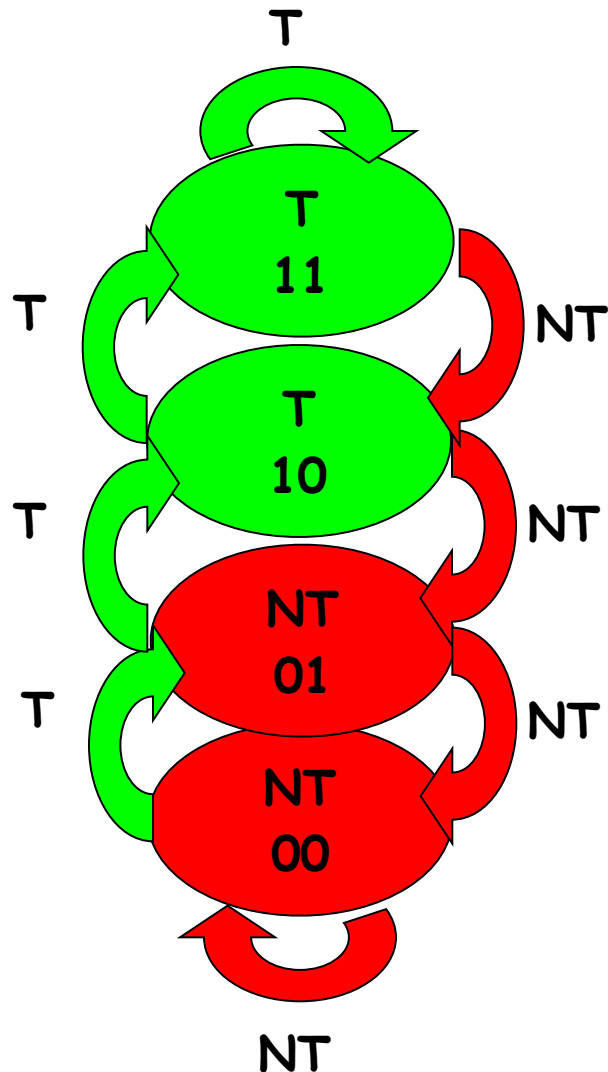
State encoding uses
two bits (encoding
is arbitrary)



• Observations:

- Repeating T stays in '11' state
- Repeating NT stays in '00' state
- Two-in-a-row to change prediction
 - (T,NT) won't change prediction
 - (NT,T) won't change prediction

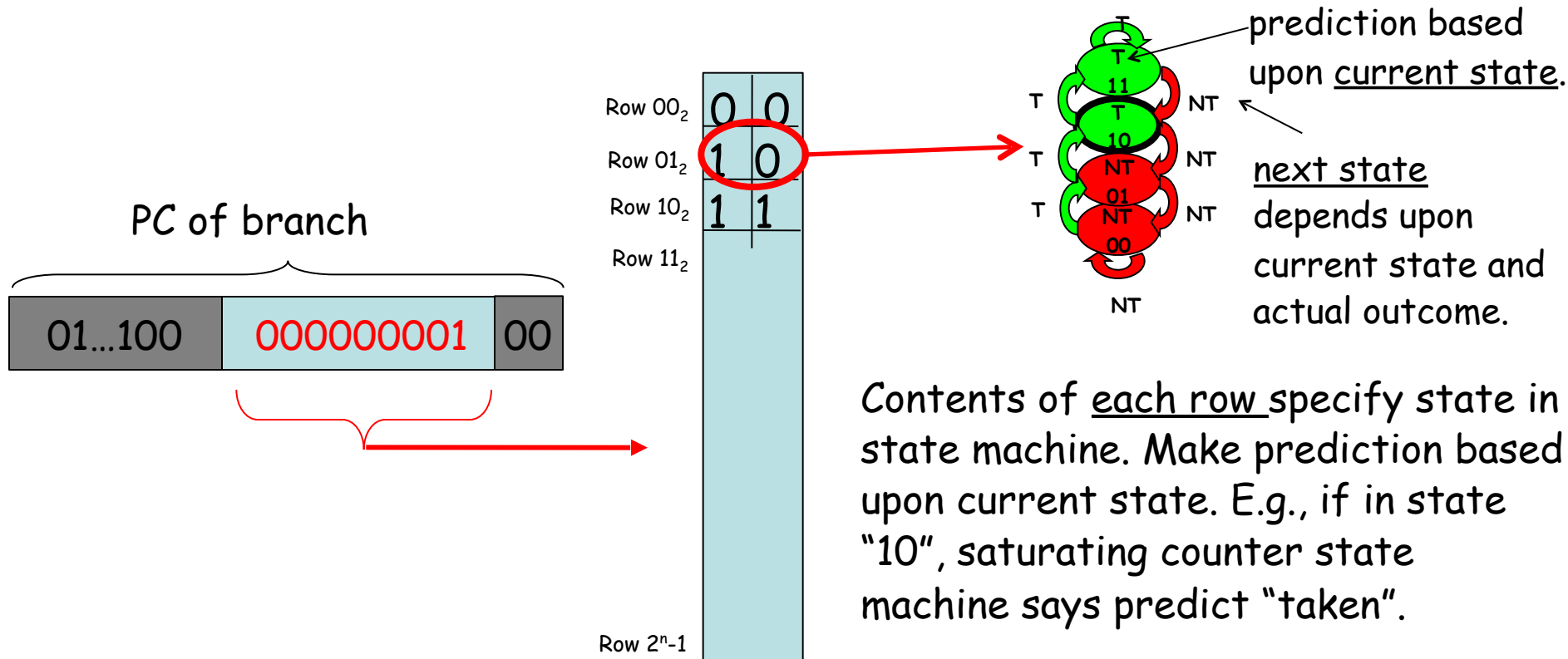
2-bit “Saturating Counter” Predictor



2-bit counter that is incremented after a taken branch, and decrement after a not-taken branch.

Prediction is simply most significant bit.

2-Bit Predictor



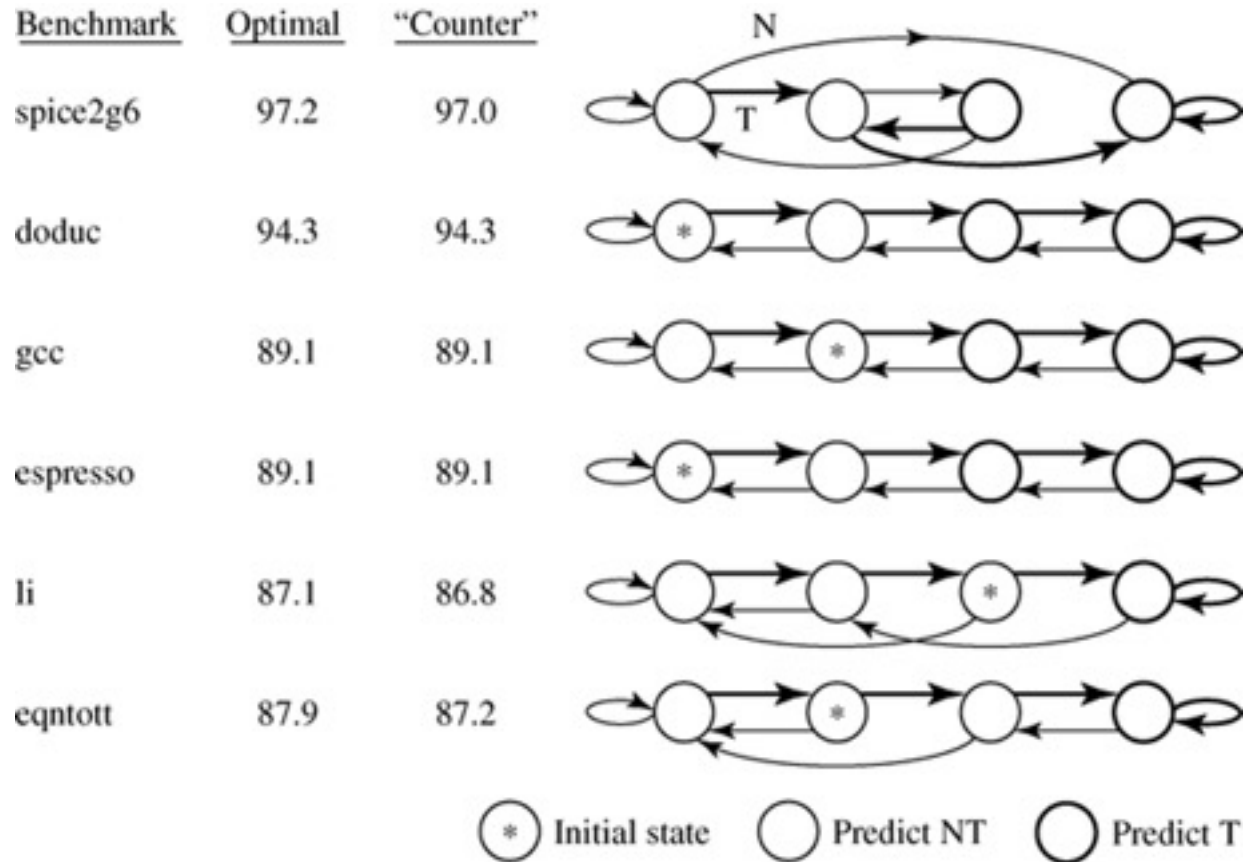
Size of table (in bits) =

$2^n \times 2$ ($n = \#$ of PC bits used)

When actual branch outcome known, update entry in table. E.g., if actual outcome taken new state in row is "11"

Other 2-bit predictors

(there are 1000's of possible 2-bit state machines...)



Study: Ravi Nair, IBM, 1992

Saturating 2-bit counter is close to "optimal" among all 2-bit state machines (though we will see better predictors).

```
Loop:    ...  
        ...  
        DSUBI R1,R1,#1  
        BNEZ  R1,Loop
```

Assume branch is “taken” 9 times in a row then not taken (i.e., loop iterates 10 times before exiting).

Question: What happens to our predictions if we encounter this loop many times using a 2-bit saturating counter branch predictor?

- A: 1 Misprediction each time loop is encountered
- B: 2 Mispredictions each time loop is encountered
- C: 9 Mispredictions each time loop is encountered
- D: 10 Mispredictions each time is encountered
- E: Not sure

```
Loop:  ...  
      ...  
      DSUBI R1,R1,#1  
      BNEZ  R1,Loop
```

Assume branch is “taken” 9 times in a row then not taken (i.e., loop iterates 10 times before exiting).

Question: What happens to our predictions if we encounter this loop many times using a 2-bit saturating counter branch predictor?

A: 1 Misprediction each time loop is encountered ✓

B: 2 Mispredictions each time loop is encountered

C: 9 Mispredictions each time loop is encountered

D: 10 Mispredictions each time is encountered

E: Not sure

```
Loop:  ...
        ...
        DSUBI R1,R1,#1
        BNEZ  R1,Loop
```

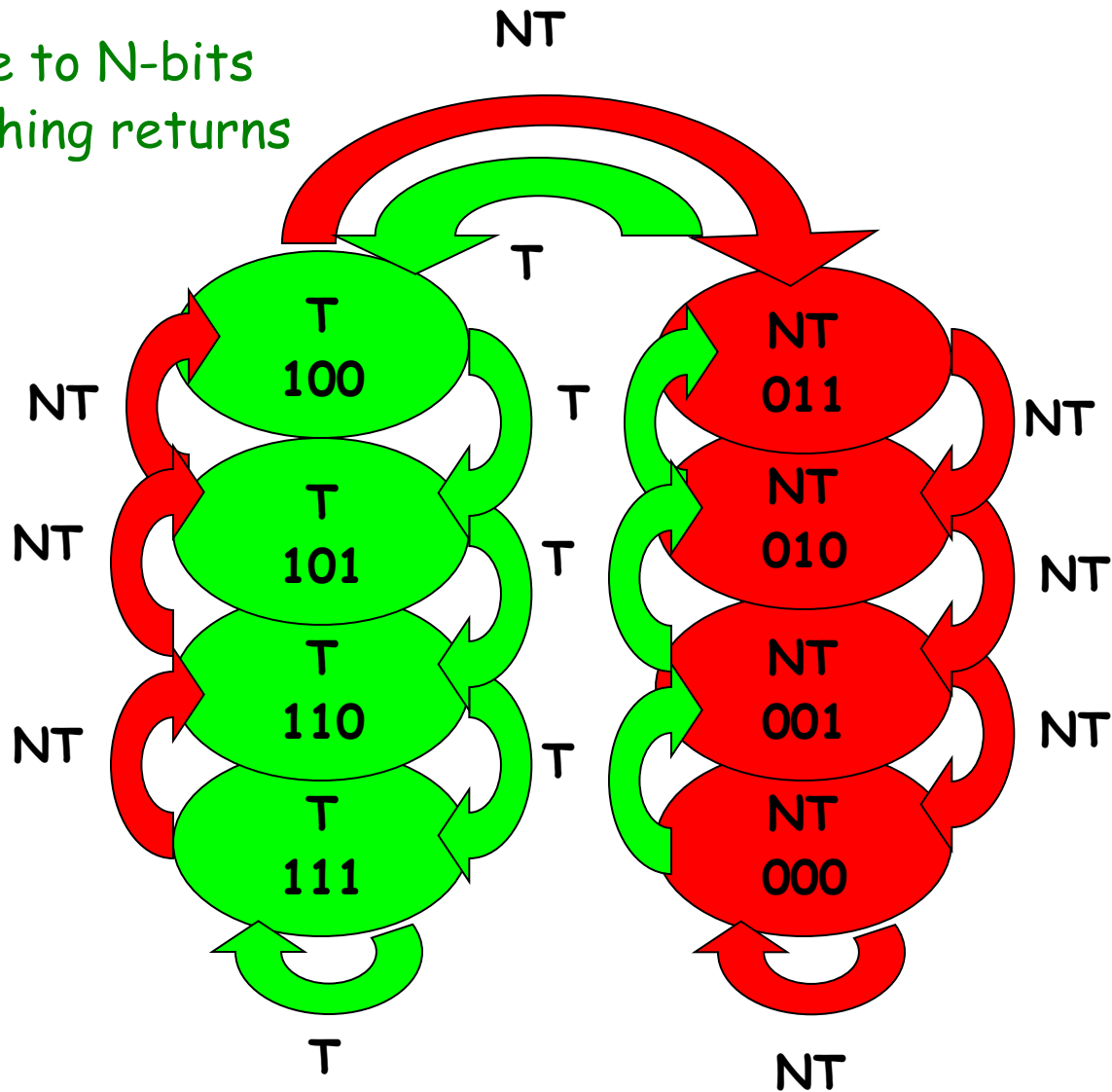
Assume branch is “taken” 9 times in a row then not taken (i.e., loop iterates 10 times before exiting).

```
Loop:    ...  
        ...  
        DSUBI R1,R1,#1  
        BNEZ  R1,Loop
```

Assume branch is “taken” 9 times in a row then not taken (i.e., loop iterates 10 times before exiting).

3-bit “Saturating Counter” Predictor

Can increase to N-bits
(but diminishing returns
as N grows)



Branch Prediction “FAQ”

- **Important:** Branch predictor only makes *predictions*. Correct predictions NOT necessary for correct program execution. Correct predictions improve performance. Hardware must check if predictions were correct when branch is executed. If wrong, “throw out” instructions following branch and start fetching again.
- Branch predictor used to reduce CPI (increase performance)
- There is a trade off between predictor size (silicon area), and the accuracy of the predictor.
- Significant innovation in branch predictor design 1985-2005.
- Typically around 95% correct prediction rate on real programs with sophisticated predictor designs (1-bit predictor: ~70-80%)
- Further improvements? Example: Going from 96% to 99% accuracy (factor of 4 fewer mispredictions) might uncover roughly twice as much instruction level parallelism in a typical program.

Branch Target Buffer (BTB)

Do the “target PC” and the instruction at the memory location pointed to by the “target PC” both have the same binary representation (identical pattern of 1’s and 0’s)?

A: Yes, highly confident (I’d bet \$100 I’m right)

B: Yes, not certain (but I might bet a coffee)

C: Not sure either way (I wouldn’t bet anything)

D: No, not certain (but I might bet a coffee)

E: No, highly confident (I’d bet \$100 I’m right)

Branch Target Buffer (BTB)

Do the “target PC” and the instruction at the memory location pointed to by the “target PC” both have the same binary representation (identical pattern of 1’s and 0’s)?

A: Yes, highly confident (I’d bet \$100 I’m right)

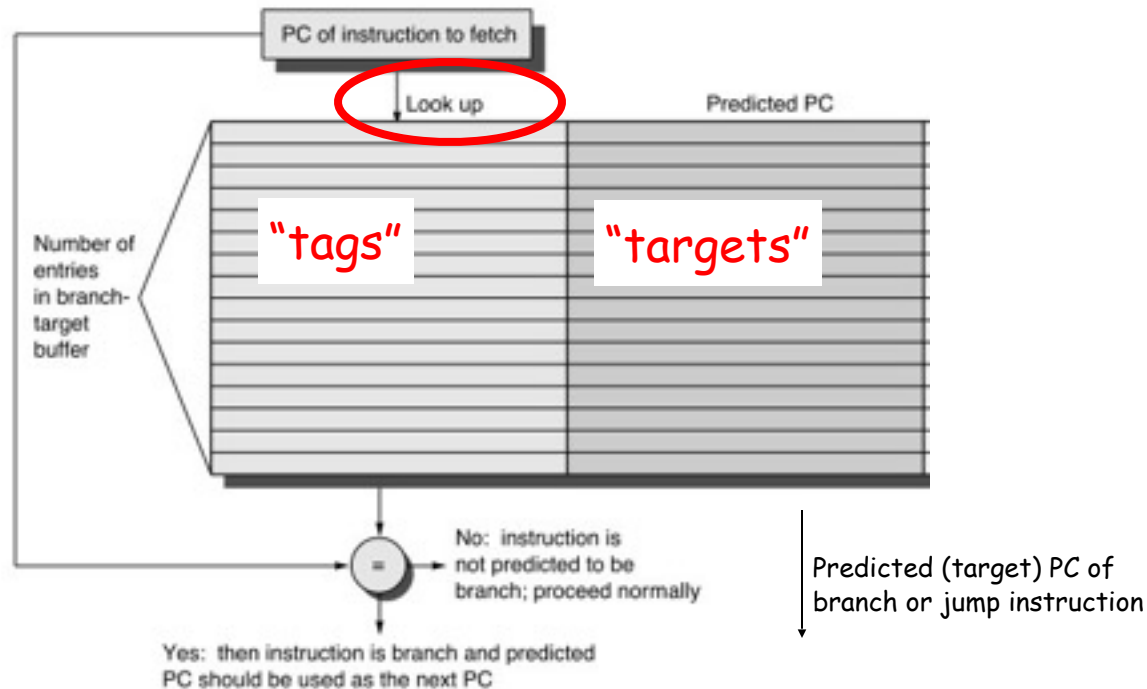
B: Yes, not certain (but I might bet a coffee)

C: Not sure either way (I wouldn’t bet anything)

D: No, not certain (but I might bet a coffee)

E: No ✓

Branch Target Buffer (BTB)



- Many ways to perform “look up” operation
 - Simplest is to use m bits of PC to select a single row to compare left hand side address against entire PC (this is called a “direct mapped” cache structure)
- BTB is a form of cache (study caches in detail later)

BTB Terminology

- In some designs, the BTB is combined with the branch predictor. Often the combined hardware is also called a BTB (even though it corresponds to a BTB and a branch predictor).
- This is not true in all designs
- Textbook uses BTB to mean BTB + branch predictor
- On 476 quiz/midterm/exam BTB means only the part of the hardware that predicts the branch target, not whether the branch is taken.

Another Branch Prediction Example...

C code

```
if (d==0) // b1  
    d=1;  
if (d==1) { // b2  
    ...  
}
```

9

9

-

Another Branch Prediction Example...

C code

```
if (d==0) // b1  
    d=1;  
if (d==1) { // b2  
    ...  
}
```



MIPS64 Assembly

```
BNEZ  R1,L1      ; branch b1 (d!=0)  
DADDI R1,R0,#1   ; d=1  
L1: DADDI R3,R1,#-1 ; R3 = d - 1  
     BNEZ  R3,L2      ; branch b2 (d!=1)  
     ...  
L2:
```

g

g

-

Another Branch Prediction Example...

C code

```
if (d==0) // b1
    d=1;
if (d==1) { // b2
    ...
}
```



MIPS64 Assembly

000000 ₂	BNEZ R1,L1	; branch b1 (d!=0)
000100 ₂	DADDI R1,R0,#1	; d=1
001000 ₂	L1: DADDI R3,R1,#-1	; R3 = d - 1
001100 ₂	BNEZ R3,L2	; branch b2 (d!=1)
...	...	
100100 ₂	L2:	

instruction addresses (binary)

9

9

Another Branch Prediction Example...

C code

```
if (d==0) // b1
    d=1;
if (d==1) { // b2
    ...
}
```



MIPS64 Assembly

000000 ₂	BNEZ R1,L1	; branch b1 (d!=0)
000100 ₂	DADDI R1,R0,#1	; d=1
001000 ₂	L1: DADDI R3,R1,#-1	; R3 = d - 1
001100 ₂	BNEZ R3,L2	; branch b2 (d!=1)
...	...	
100100 ₂	L2:	

instruction addresses (binary)

consider initial values for d = 0, 1, 2:

initial value of d	d==0?	b1	g
0	yes	N	

Another Branch Prediction Example...

C code

```
if (d==0) // b1
    d=1;
if (d==1) { // b2
    ...
}
```



MIPS64 Assembly

000000 ₂	BNEZ R1,L1	; branch b1 (d!=0)
000100 ₂	DADDI R1,R0,#1	; d=1
001000 ₂	L1: DADDI R3,R1,#-1	; R3 = d - 1
001100 ₂	BNEZ R3,L2	; branch b2 (d!=1)
...	...	
100100 ₂	L2:	

instruction addresses (binary)

consider initial values for d = 0, 1, 2:

initial value of d	d==0?	b1	g
0	yes	N	
1	no	T	
2	no	T	

Another Branch Prediction Example...

C code

```
if (d==0) // b1
    d=1;
if (d==1) { // b2
    ...
}
```



MIPS64 Assembly

```

0000002    BNEZ  R1,L1      ; branch b1 (d!=0)
0001002    DADDI R1,R0,#1   ; d=1
0010002 L1: DADDI R3,R1,#-1   ; R3 = d - 1
0011002    BNEZ  R3,L2      ; branch b2 (d!=1)
...
1001002 L2:

```

instruction addresses (binary)

consider initial values for d = 0, 1, 2:

initial value of d	d==0?	b1	value of d before b2	d==1?	b2
0	yes	N	1	yes	N
1	no	T	1	yes	N
2	no	T	2	no	T

Example, cont'd...

First, let's consider behavior of the simple 1-bit predictor from slide 14 (all table entries initialized to "not taken") if d alternates from 2 to 0: $d = 2, 0, 2, 0, \dots$

→
across: behavior as code executes a single time

from previous slide

d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	N	T	T	N	T	T
0	T	N	N		N	N
2		T			T	
0		N			N	

↓
down: successive passes through the example code

Example, cont'

How many mispredictions?

A: 2

B: 4

C: 6

D: 8

E: Not sure

First, let's consider behavior of the simple 1-bit predictor from slide 14 (all table entries initialized to "not taken") if it alternates from 2 to 0: $d = 2, 0, 2, 0, \dots$

from previous slide

across: behavior as code executes a single time

d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	N	T	T	N	T	T
0	T	N	N		N	N
2		T			T	
0		N			N	

down: successive passes through the example code

Example, cont'

How many mispredictions?

A: 2

B: 4

C: 6

D: 8 ✓

E: Not sure

First, let's consider behavior of the simple 1-bit predictor from slide 14 (all table entries initialized to "not taken") if it alternates from 2 to 0: $d = 2, 0, 2, 0, \dots$

from previous slide

across: behavior as code executes a single time

d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	N	T	T	N	T	T
0	T	N	N		N	N
2		T			T	
0		N			N	

down: successive passes through the example code

Example, cont'd...

First, let's consider behavior of the simple 1-bit predictor from slide 14 (all table entries initialized to "not taken") if d alternates from 2 to 0: $d = 2, 0, 2, 0, \dots$

→
across: behavior as code executes a single time

from previous slide

d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	N	T	T	N	T	T
0	T	N	N		N	N
2		T			T	
0		N			N	

↓
down: successive passes through the example code

Example, cont'd...

First, let's consider behavior of the simple 1-bit predictor from slide 14 (all table entries initialized to "not taken") if d alternates from 2 to 0: $d = 2, 0, 2, 0, \dots$

→
across: behavior as code executes a single time

from previous slide

d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	N	T	T	N	T	T
0	T	N	N	T	N	N
2	N	T	T	N	T	T
0	T	N	N	T	N	N

↓
down: successive passes through the example code

Example, cont'd...

First, let's consider behavior of the simple 1-bit predictor from slide 14 (all table entries initialized to "not taken") if d alternates from 2 to 0: $d = 2, 0, 2, 0, \dots$

→
across: behavior as code executes a single time

from previous slide

d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	N	T	T	N	T	T
0	T	N	N	T	N	N
2	N	T	T	N	T	T
0	T	N	N	T	N	N

↓
down: successive passes through the example code

Always mispredicts!

Another look at the code...

C code

```
if (d==0) // b1
    d=1;
if (d==1) { // b2
    ...
}
```



MIPS64 Assembly

```
BNEZ  R1,L1      ; branch b1 (d!=0)
DADDI  R1,R0,#1   ; d=1
L1: DADDI  R3,R1,#-1 ; R3 = d - 1
      BNEZ  R3,L2      ; branch b2 (d!=1)
      ...
L2:
```

g

,)

g

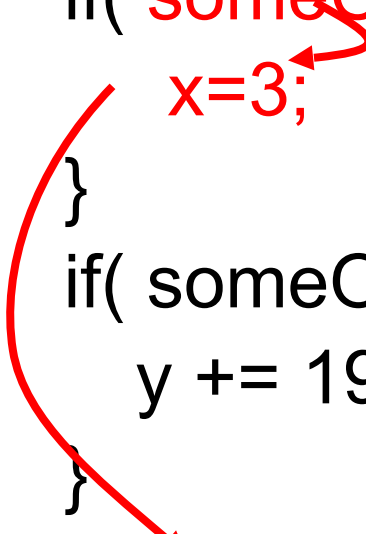
initial value of d	d==0?	b1	value of d before b2	d==1?	b2
0	yes	N	1	yes	N
1	no	T	1	yes	N
2	no	T	2	no	T

Note: branch **b2** correlated with branch **b1** (if b1 is NT, b2 is NT)

Why Are Branch Outcomes Correlated?

Answer: Affector Branches

```
x=0;  
if( someCondition ) { /* branch A */  
    x=3;  
}  
if( someOtherCondition ) { /* branch B */  
    y += 19;  
}  
if( x <= 0 ) { /* branch C */  
    doSomething();  
}
```



A red arrow originates from the assignment `x=3;` inside the first `if` block (branch A) and points to the condition `x <= 0` in the third `if` block (branch C). This illustrates how the outcome of branch A affects the outcome of branch C, demonstrating correlated branch outcomes.

A (Silly) Analogy...

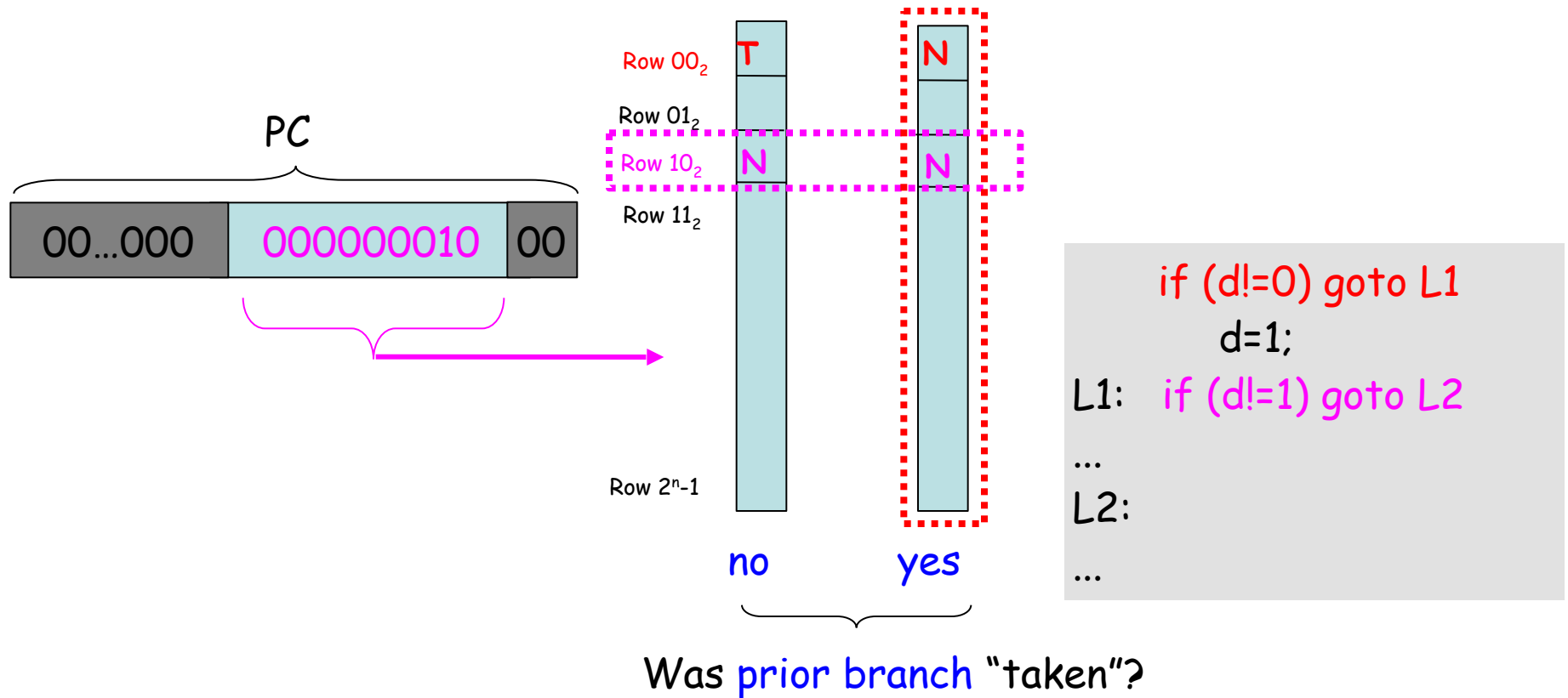


- Which bus route should you take to school tomorrow?
- Goal is to get to school fastest.
- Make prediction based upon:
 - Static information (shortest bus route in km?)
 - History: Most times, route T was faster than route N.
 - Better prediction if consider additional “context”: Route N better than route T on rainy days, but worse on sunny days.

(Silly) Correlating Branch Predictor Analogy...

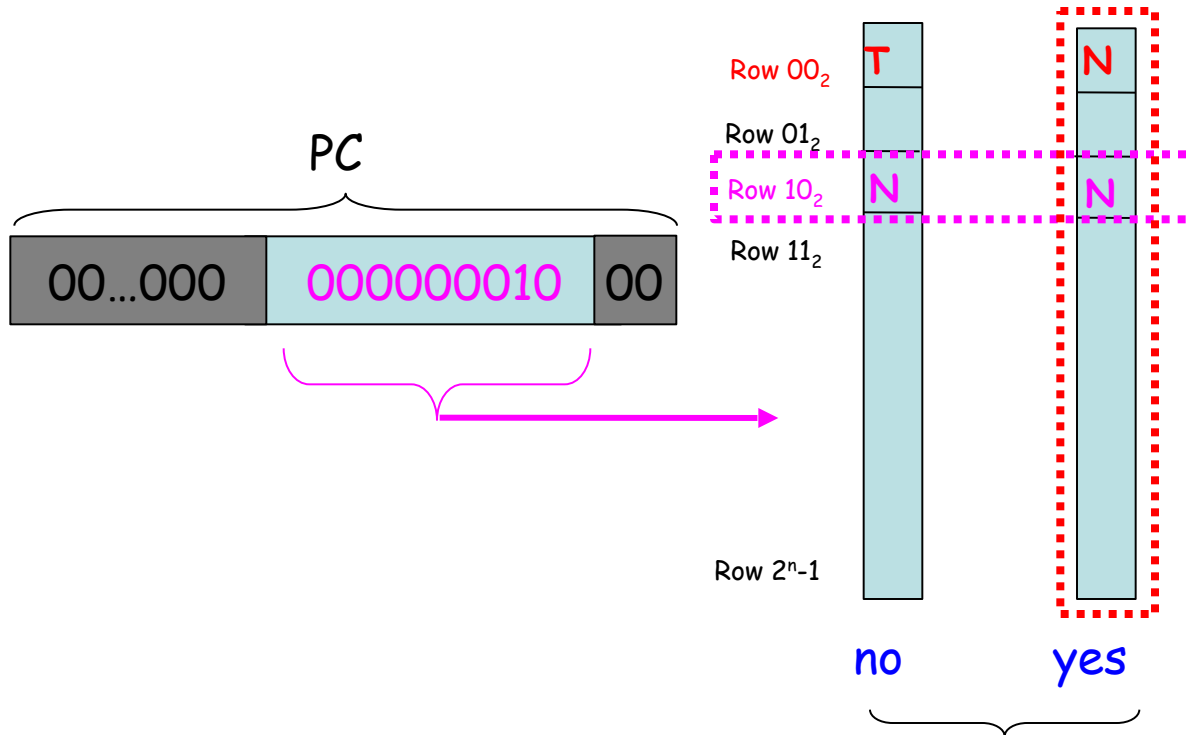
- Problem: You want to predict which bus to take to school.
- You have a feeling that the best route to take seems to depend on the weather. It rains 50% of the time and is sunny 50% of the time.
- You don't yet know it, but "route T" is usually faster than "route N" on sunny days, but is often slower than "route N" on rainy days. Perhaps there are lots of accidents on route T when it rains. You don't care why the bus is slow, you just want to make it to class on time. Besides, you're too busy studying while on the bus to notice why it is faster or slower.
- How are you ever going to notice you should predict N on rainy days, and T on sunny days?
- Solution: Keep two tables: One you record what happens on rainy days, other you record what happens on sunny days.

1-Bit Branch Predictor with 1-Bit of Correlation:



Use prior branch outcomes to select which table to use. Prior branch outcome is like "checking weather" before deciding which bus to. After actual branch outcome is known, update entry used to make prediction with actual branch outcome information.

1-Bit Branch Predictor with 1-Bit



Was prior branch "taken"?

If last branch was "not taken" and PC for branch we want to predict is 0x00...00, then prediction would be?

- A: Taken, very sure
- B: Taken, but not sure
- C: Not sure
- D: Not Taken, but not sure
- E: Not Take, very sure

L1: if (d!=1) goto L2

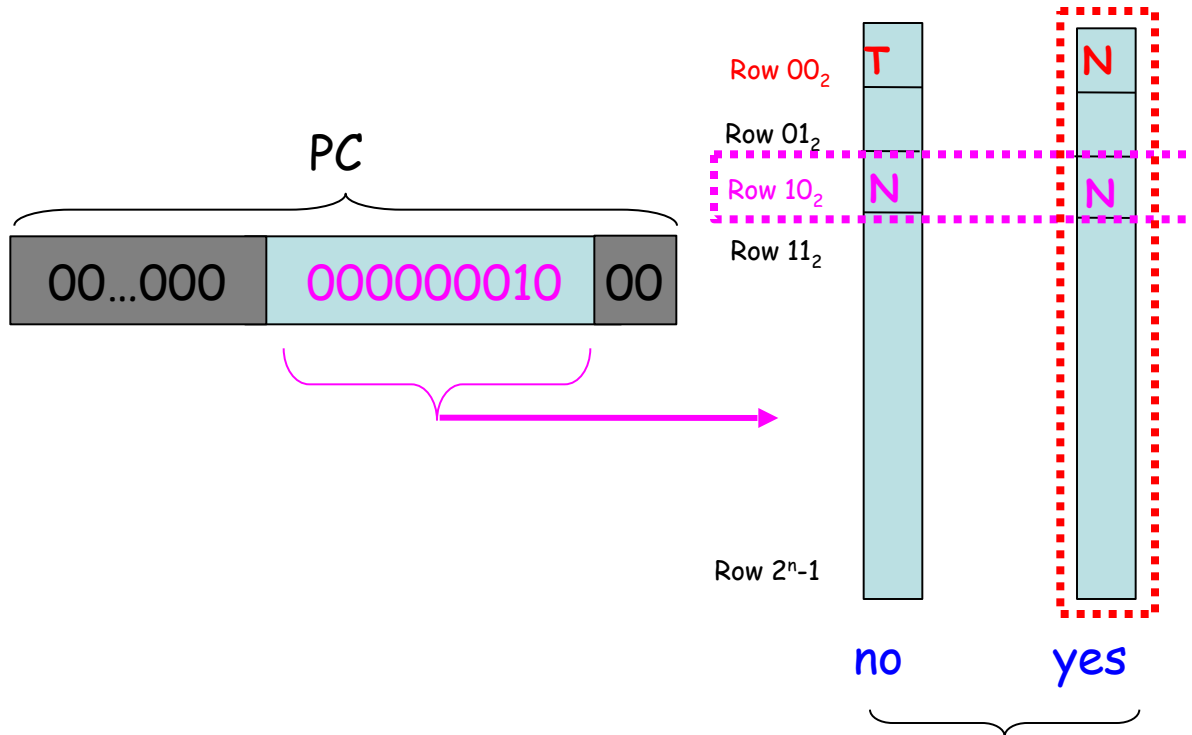
...

L2:

...

Use prior branch outcomes to select which table to use. Prior branch outcome is like "checking weather" before deciding which bus to. After actual branch outcome is known, update entry used to make prediction with actual branch outcome information.

1-Bit Branch Predictor with 1-Bit



Was prior branch "taken"?

If last branch was "not taken" and PC for branch we want to predict is 0x00...00, then prediction would be?

- A: Taken, very sure ✓
- B: Taken, but not sure
- C: Not sure
- D: Not Taken, but not sure
- E: Not Take, very sure

L1: if (d!=1) goto L2

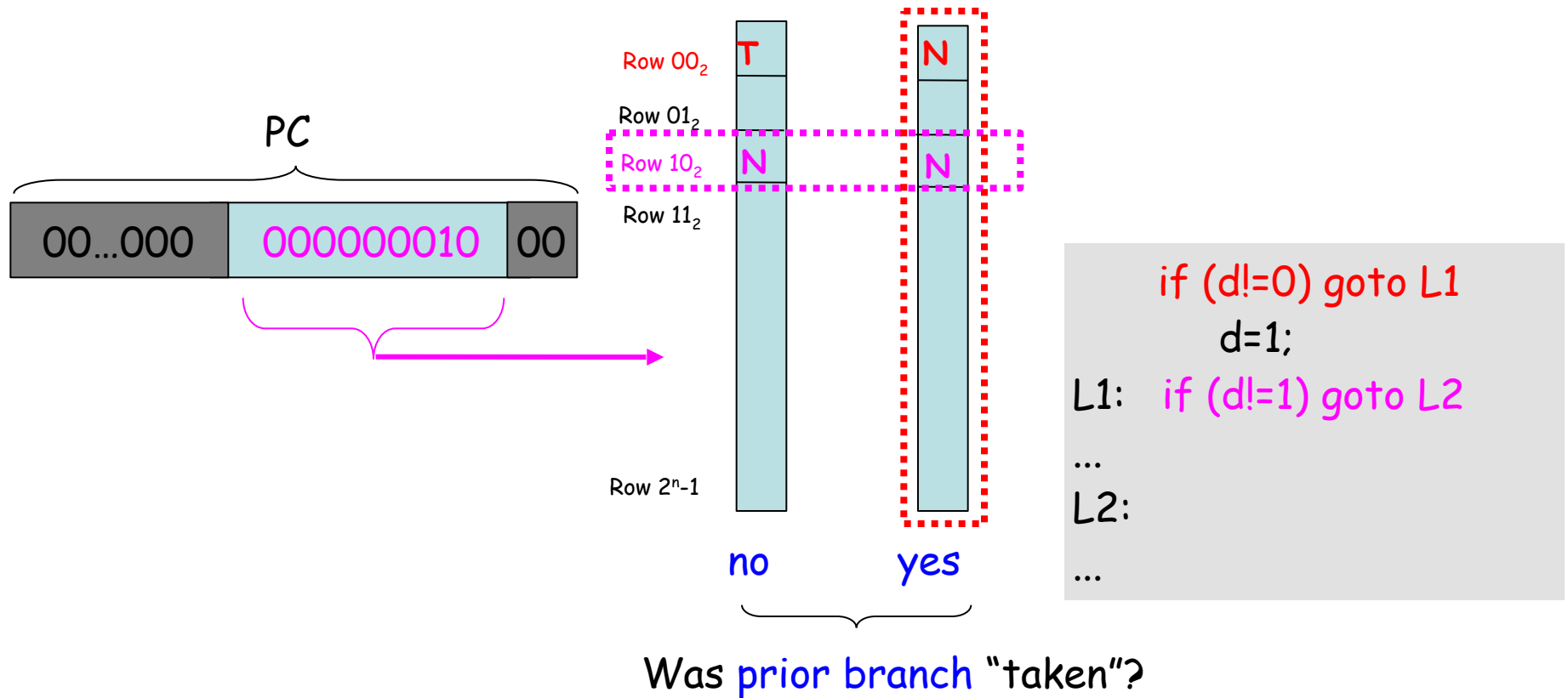
...

L2:

...

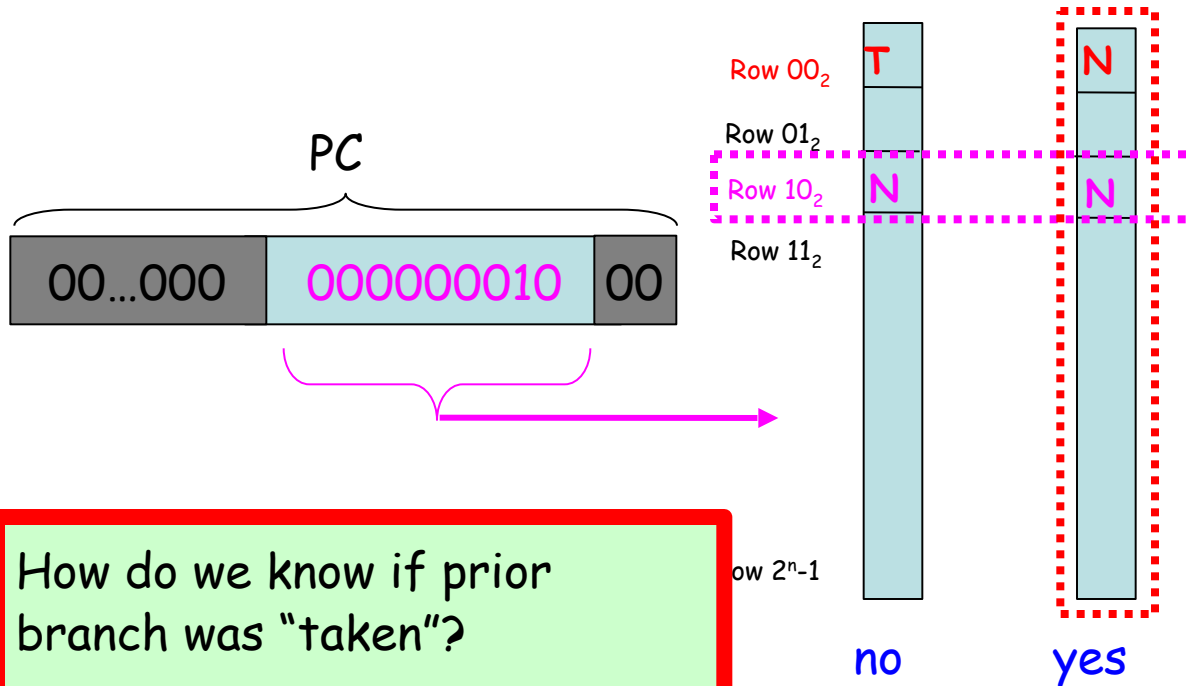
Use prior branch outcomes to select which table to use. Prior branch outcome is like "checking weather" before deciding which bus to. After actual branch outcome is known, update entry used to make prediction with actual branch outcome information.

1-Bit Branch Predictor with 1-Bit of Correlation:



Use prior branch outcomes to select which table to use. Prior branch outcome is like "checking weather" before deciding which bus to. After actual branch outcome is known, update entry used to make prediction with actual branch outcome information.

1-Bit Branch Predictor with 1-Bit of Correlation:



How do we know if prior branch was "taken"?

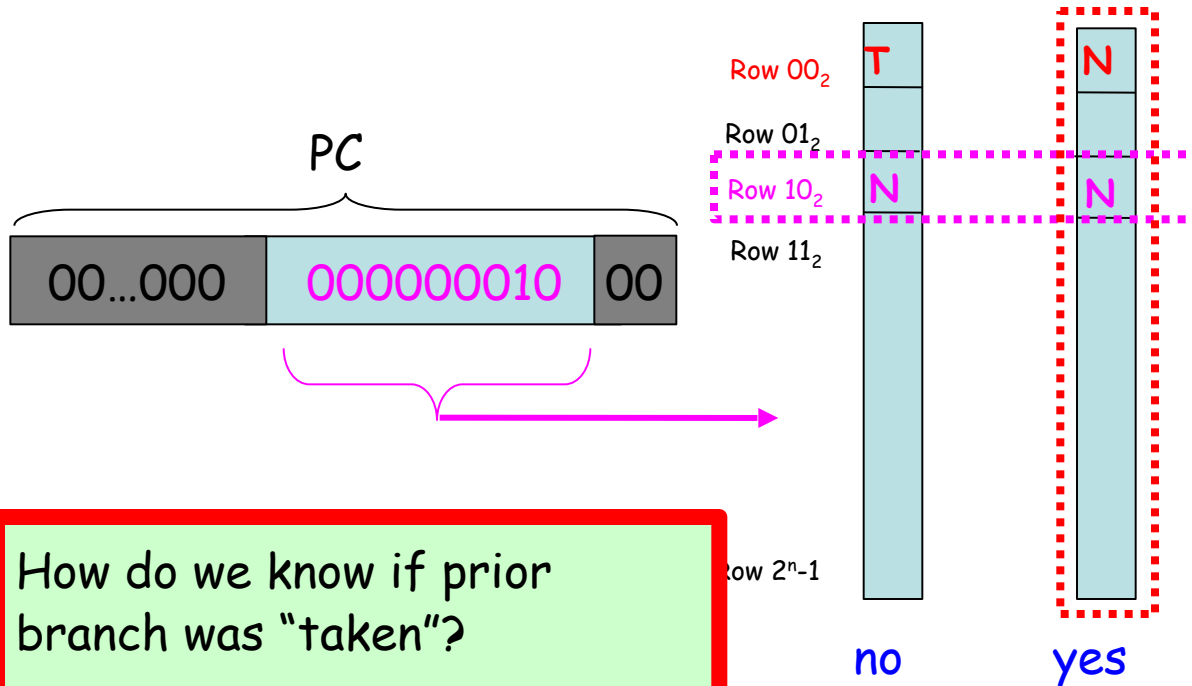
- A: Last prediction
- B: Last actual outcome
- C: Not sure

Was prior branch "taken"?

```
if (d!=0) goto L1
    d=1;
L1:  if (d!=1) goto L2
...
L2:
...
```

select which table to use. Prior branch outcome is like "checking weather" before deciding which bus to. After actual branch outcome is known, update entry used to make prediction with actual branch outcome information.

1-Bit Branch Predictor with 1-Bit of Correlation:



How do we know if prior branch was "taken"?

A: Last prediction

B: Last actual outcome ✓
(pipelining complicates this)

C: Not sure

Was prior branch "taken"?

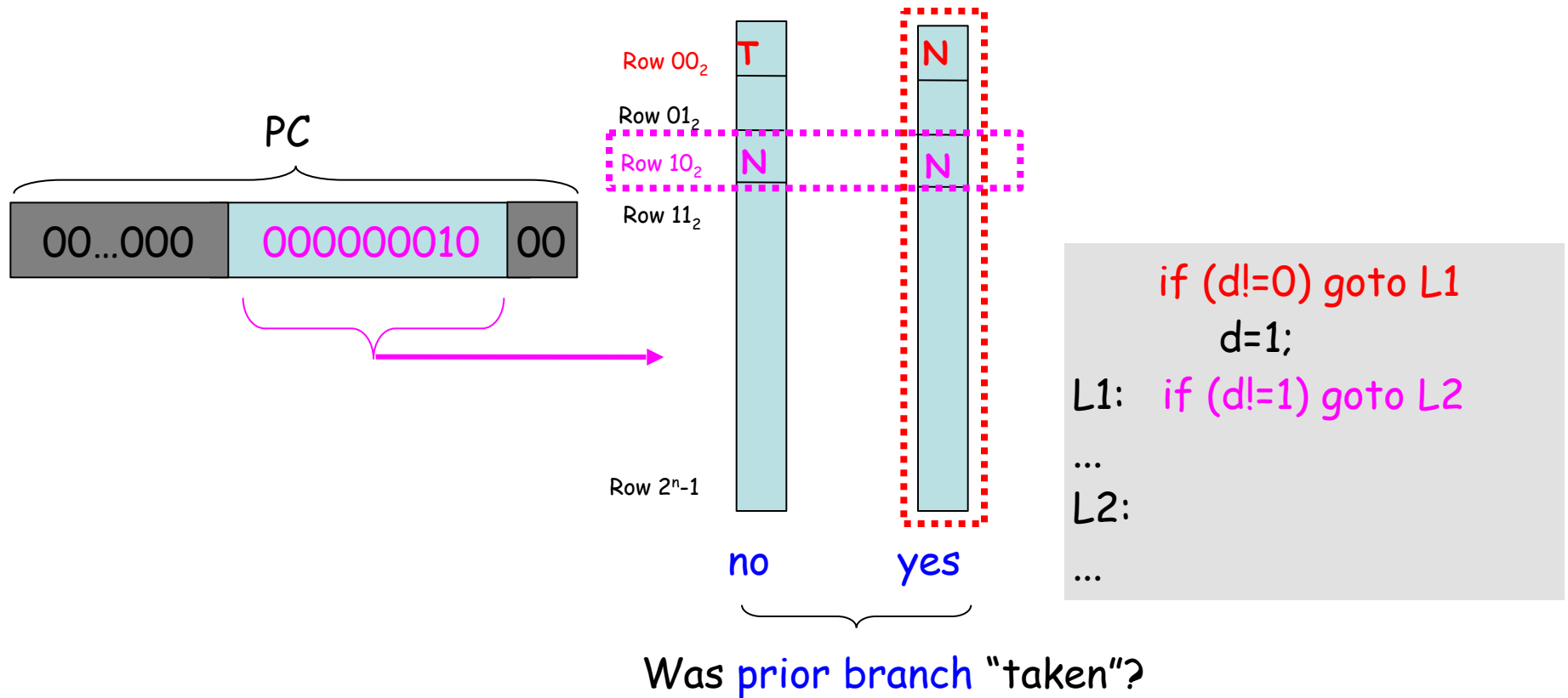
select which table to use. Prior branch
"taken" before deciding which bus to. After
update entry used to make prediction

```

if (d!=0) goto L1
d=1;
L1: if (d!=1) goto L2
...
L2:
...
    
```

with actual branch outcome information.

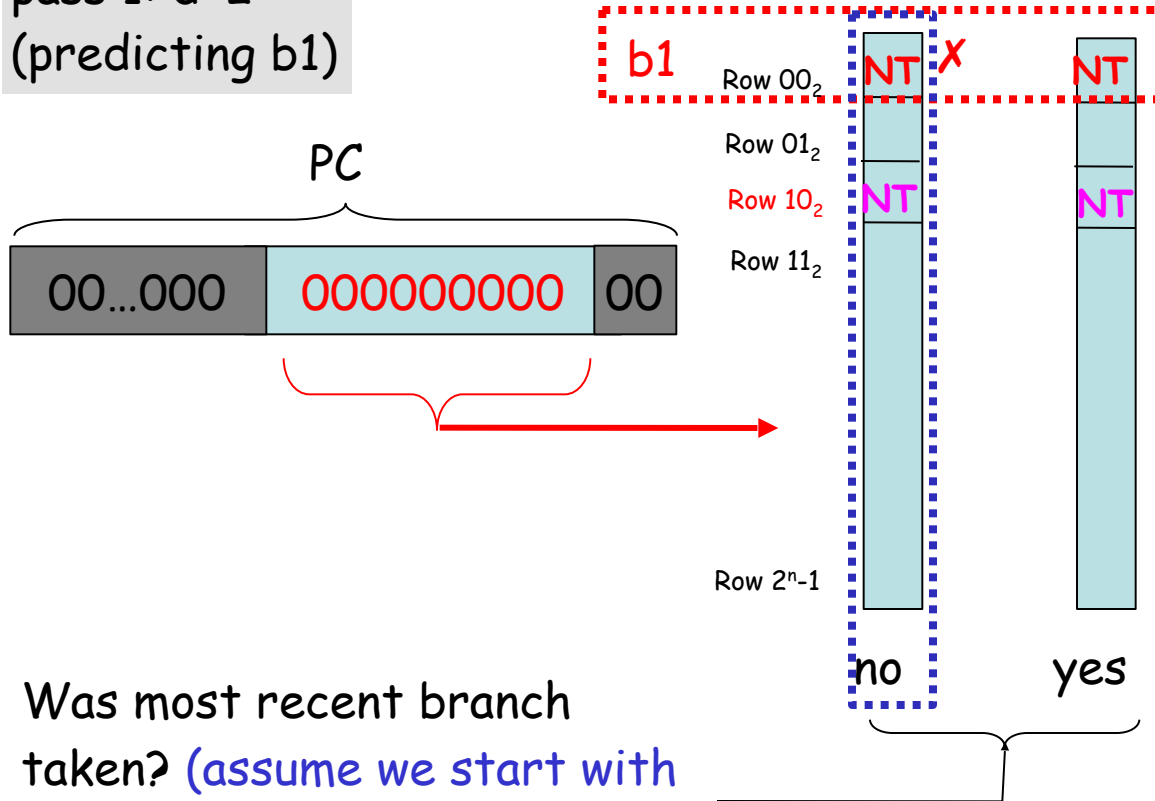
1-Bit Branch Predictor with 1-Bit of Correlation:



Use prior branch outcomes to select which table to use. Prior branch outcome is like "checking weather" before deciding which bus to. After actual branch outcome is known, update entry used to make prediction with actual branch outcome information.

1-Bit Predictor with 1-Bit of Correlation:

pass 1: d=2
(predicting b1)



Was most recent branch
taken? (assume we start with
answer = "no" for first branch)

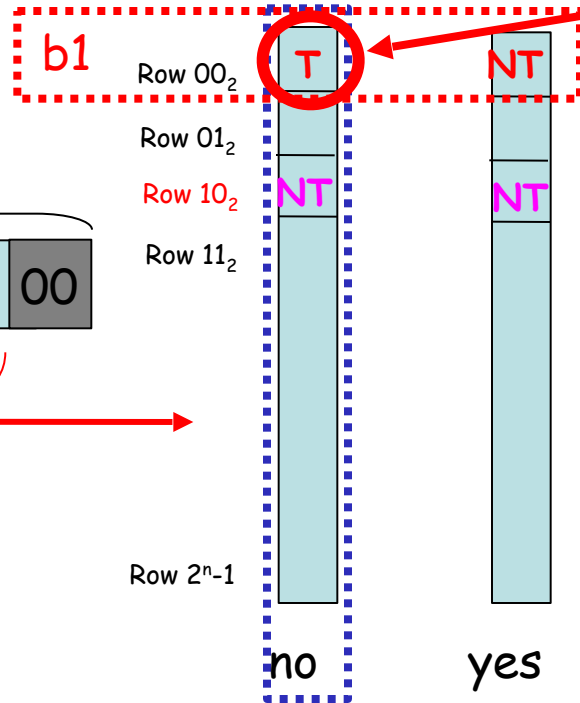
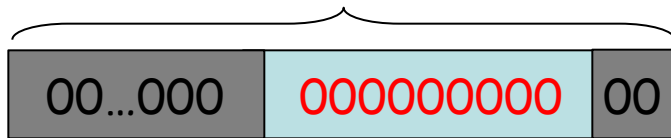
1-bit of "branch history" used to correlate prediction
for current branch with outcome of last branch.

```
if (d!=0) goto L1;//  
b1  
    d=1;  
L1: if (d!=1) goto L2;//  
b2  
...  
L2:  
...
```

1-Bit Predictor with 1-Bit of Correlation:

pass 1: $d=2$,
(after executing b_1 ,
before executing b_2)

PC



Update table after
actual outcome of
 b_1 is known

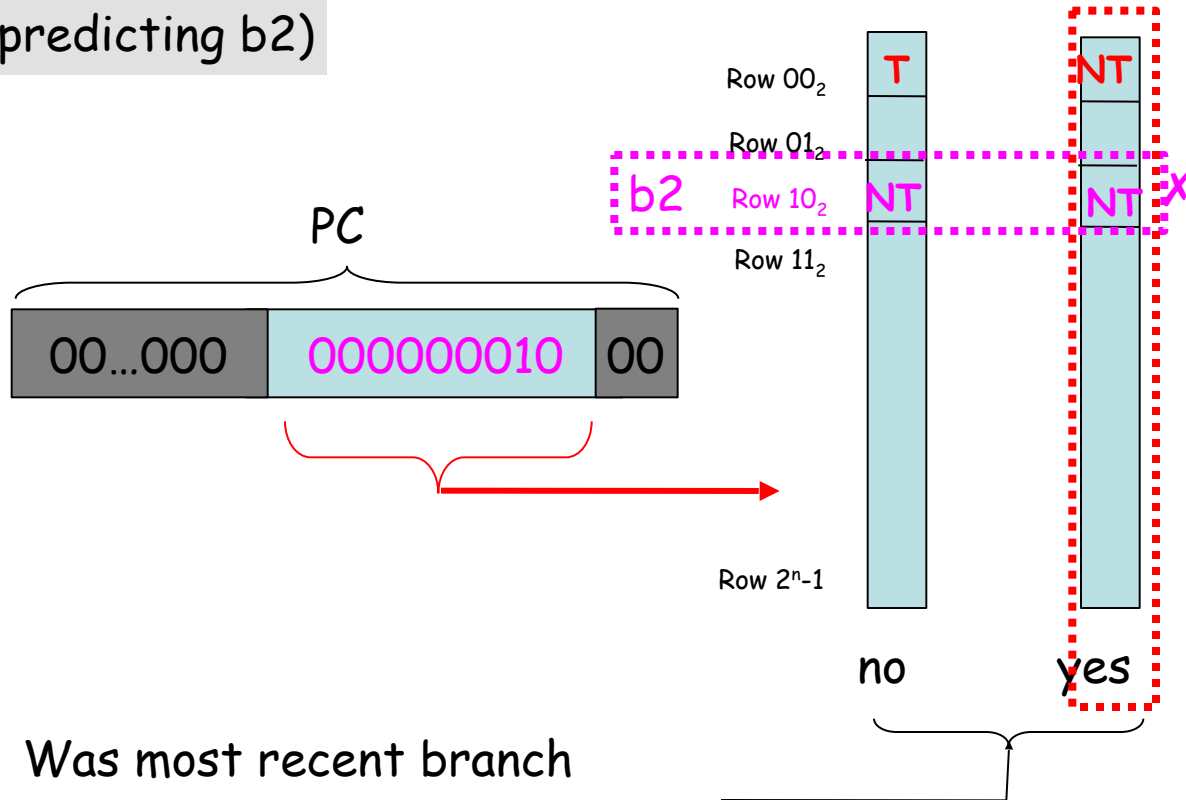
```
if ( $d \neq 0$ ) goto L1; //  
 $b_1$   
     $d=1$ ;  
L1: if ( $d \neq 1$ ) goto L2; //  
 $b_2$   
    ...  
L2:  
    ...
```

Was most recent branch
taken? (assume we start with
answer = "no" for first branch)

1-bit of "branch history" used to correlate prediction
for current branch with outcome of last branch.

1-Bit Predictor with 1-Bit of Correlation:

pass 1: d=2
(predicting b2)



```
if (d!=0) goto L1;//  
b1  
    d=1;  
L1: if (d!=1) goto L2;//  
b2  
    ...  
L2:  
    ...
```

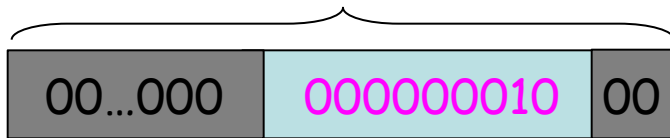
Was most recent branch taken? "Yes" because b1 was taken.

1-bit of "branch history" used to correlate prediction for current branch with outcome of last branch.

1-Bit Predictor with 1-Bit of Correlation:

pass 1: d=2
(updating
predictor
for b2)

PC



Row 00 ₂	T	NT
Row 01 ₂		
Row 10 ₂	NT	T
Row 11 ₂		
	no	yes

```
if (d!=0) goto L1;//  
b1  
    d=1;  
L1: if (d!=1) goto L2;//  
b2  
    ...  
L2:  
    ...
```

Was most recent branch
taken? "Yes" because b1 was
taken.

1-bit of "branch history" used to correlate prediction
for current branch with outcome of last branch.

Example, cont'd...

Consider behavior of a 1-bit predictor with one bit of correlation history... if d alternates from 2 to 0: d = 2, 0, 2, 0, ...

from slide 29

→
across: behavior as code executes a single time

d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	<u>N</u> /N	T	T/N	N/ <u>N</u>	T	N/T
0	T/ <u>N</u>	N	T/N		N	
2		T			T	
0		N			N	

↓
down: successive passes through the example code

Example, cont'

Consider behavior of a 1-bit predictor with one bit alternates from 2 to 0: $d = 2, 0, 2, 0, \dots$

How many mispredictions?

A: 2

B: 4

C: 6

D: 8

E: Not sure

from slide 29

across: behavior as code executes a single time

d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	<u>N</u> /N	T	T/N	N/ <u>N</u>	T	N/T
0	T/ <u>N</u>	N	T/N		N	
2		T			T	
0		N			N	

down: successive passes through the example code

Example, cont'

Consider behavior of a 1-bit predictor with one bit alternates from 2 to 0: $d = 2, 0, 2, 0, \dots$

How many mispredictions?

A: 2 ✓

B: 4

C: 6

D: 8

E: Not sure

from slide 29

across: behavior as code executes a single time

d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	<u>N</u> /N	T	T/N	N/ <u>N</u>	T	N/T
0	T/ <u>N</u>	N	T/N		N	
2		T			T	
0		N			N	

down: successive passes through the example code

Example, cont'd...

Consider behavior of a 1-bit predictor with one bit of correlation history... if d alternates from 2 to 0: d = 2, 0, 2, 0, ...

from slide 29

→
across: behavior as code executes a single time

d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	<u>N</u> /N	T	T/N	N/ <u>N</u>	T	N/T
0	T/ <u>N</u>	N	T/N		N	
2		T			T	
0		N			N	

↓
down: successive passes through the example code

Example, cont'd...

Consider behavior of a 1-bit predictor with one bit of correlation history... if d alternates from 2 to 0: d = 2, 0, 2, 0, ...

from slide 29

across: behavior as code executes a single time

d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	<u>N</u> /N	T	T/N	N/ <u>N</u>	T	N/T
0	T/ <u>N</u>	N	T/N	<u>N</u> /T	N	N/T
2	<u>I</u> /N	T	T/N	N/ <u>I</u>	T	N/T
0	T/ <u>N</u>	N	T/N	<u>N</u> /T	N	N/T

down: successive passes through the example code

Example, cont'd...

Consider behavior of a 1-bit predictor with one bit of correlation history... if d alternates from 2 to 0: d = 2, 0, 2, 0, ...

from slide 29

across: behavior as code executes a single time

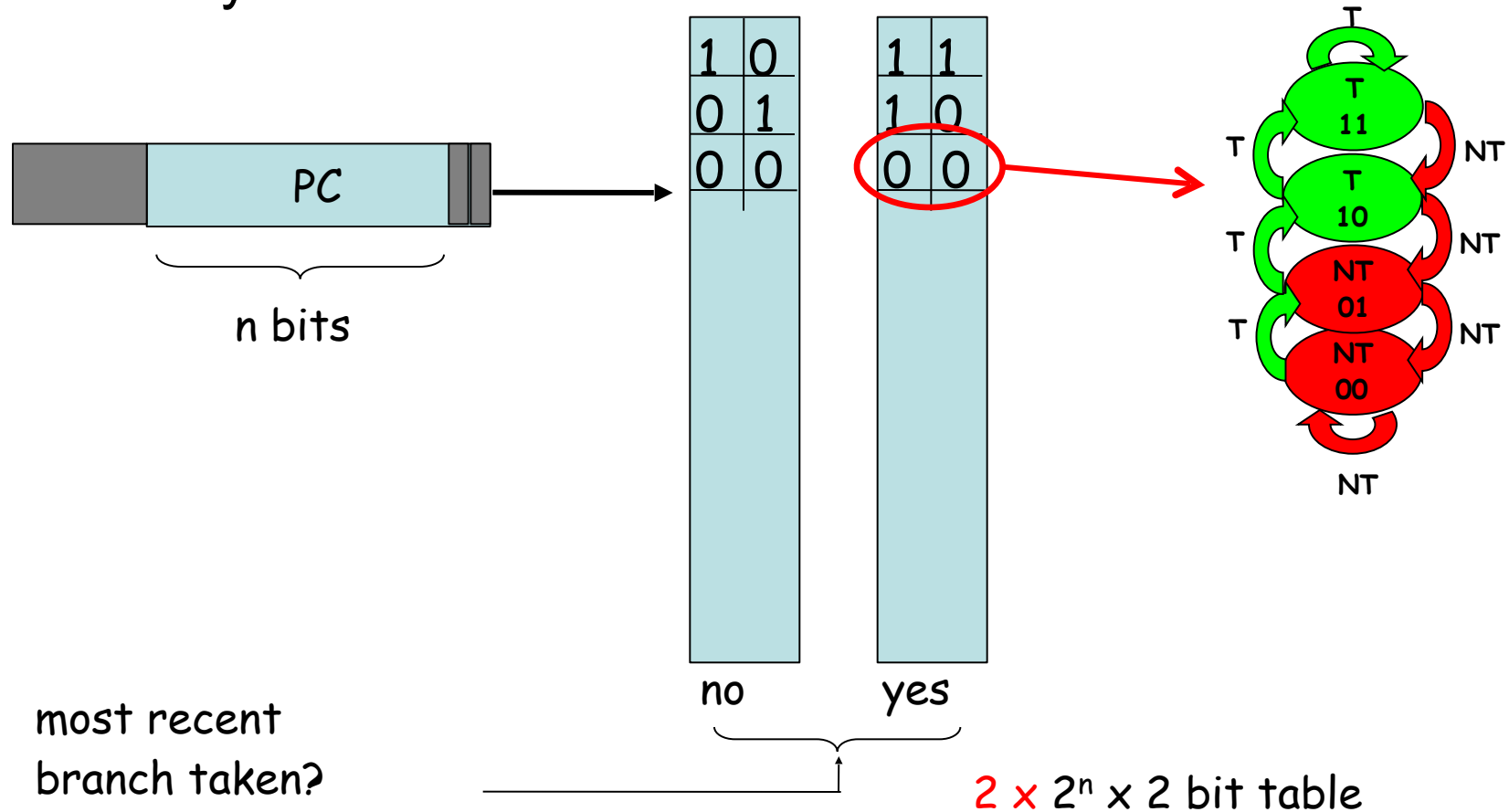
d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	<u>N</u> /N	T	T/N	N/ <u>N</u>	T	N/T
0	T/ <u>N</u>	N	T/N	<u>N</u> /T	N	N/T
2	<u>I</u> /N	T	T/N	N/ <u>I</u>	T	N/T
0	T/ <u>N</u>	N	T/N	<u>N</u> /T	N	N/T

down: successive passes through the example code

6 out of 8 = 75% correct predictions

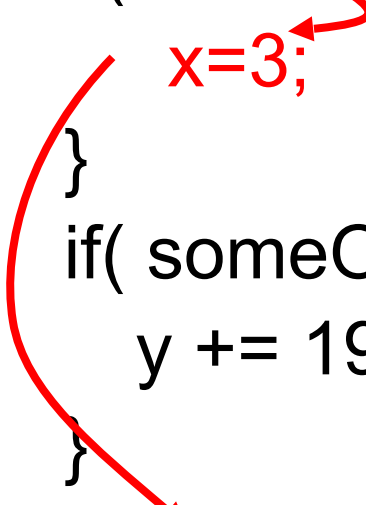
2-bit Predictor with 1-bit of Correlation

Useful to build a predictor that uses both correlation and hysteresis.



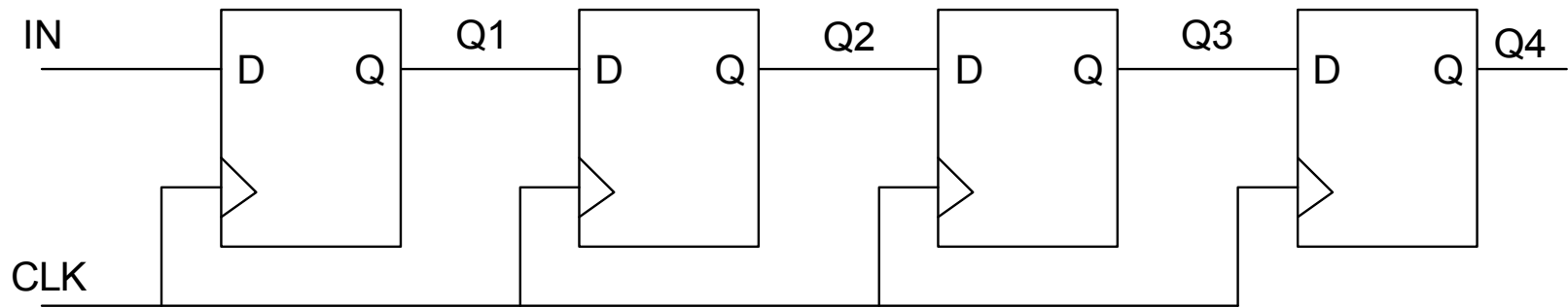
Affector Branches, reconsidered...

```
x=0;  
if( someCondition ) { /* branch A */  
    x=3;  
}  
if( someOtherCondition ) { /* branch B */  
    y += 19;  
}  
if( x <= 0 ) { /* branch C */  
    doSomething();  
}
```

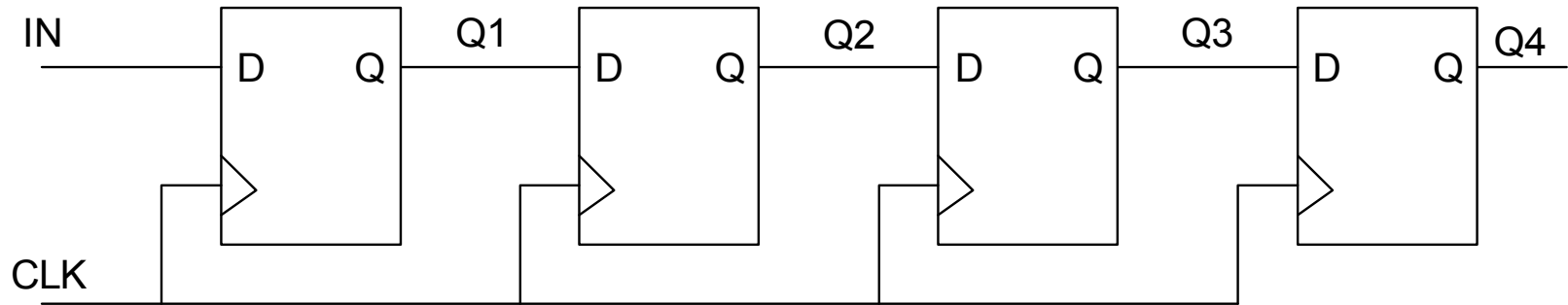
A red curved arrow originates from the 'x=3;' statement in branch A and points to the 'x' variable in the condition 'x <= 0' of branch C, illustrating how a change in 'x' affects the execution of branch C.

Correlating against Multiple Branches

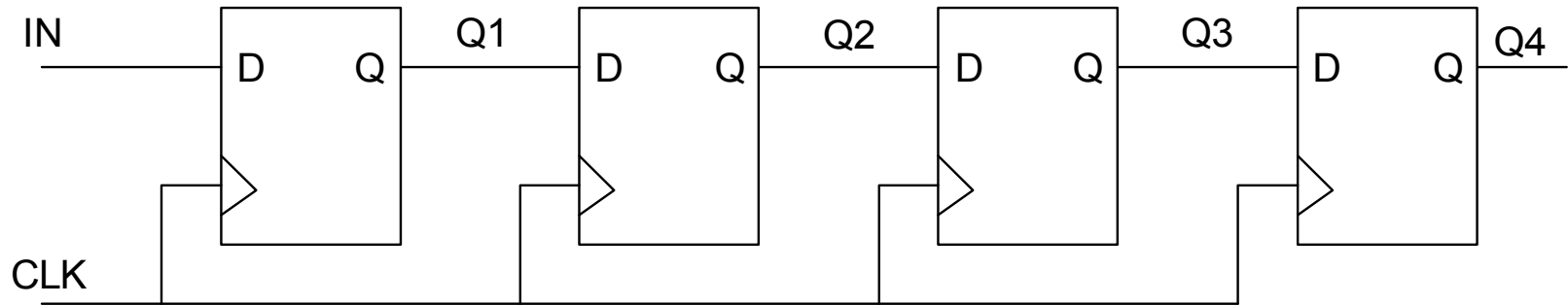
- In the last slide we saw that the outcome of branch C depended in some way upon the outcome of branch A but not Branch B.
- Instead of tracking the outcome of only the last branch and using that to choose between two tables, we will track the outcome of last N branch outcomes and use that to choose between 2^N tables.
- We track the outcome of the last N branches using an N-bit shift register.



Quick Review: Shift Register

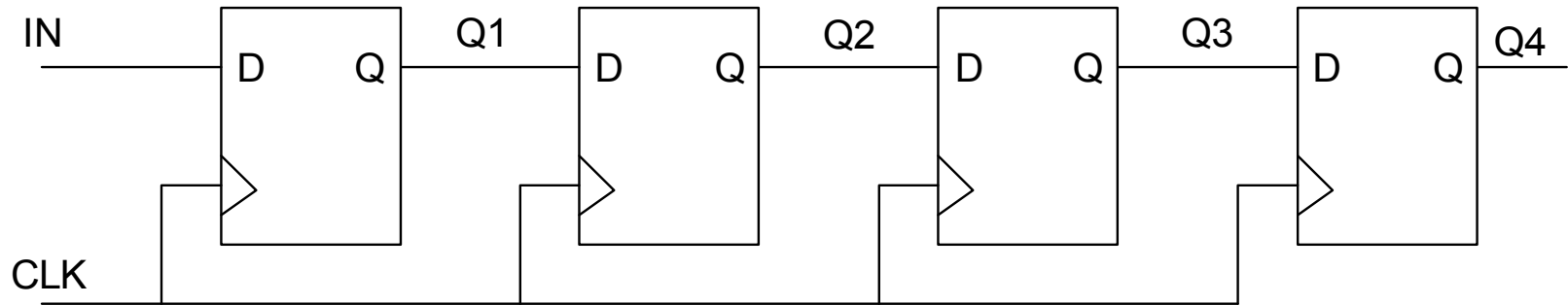


Quick Review: Shift Register



Each cycle, shift contents of this register by one bit:

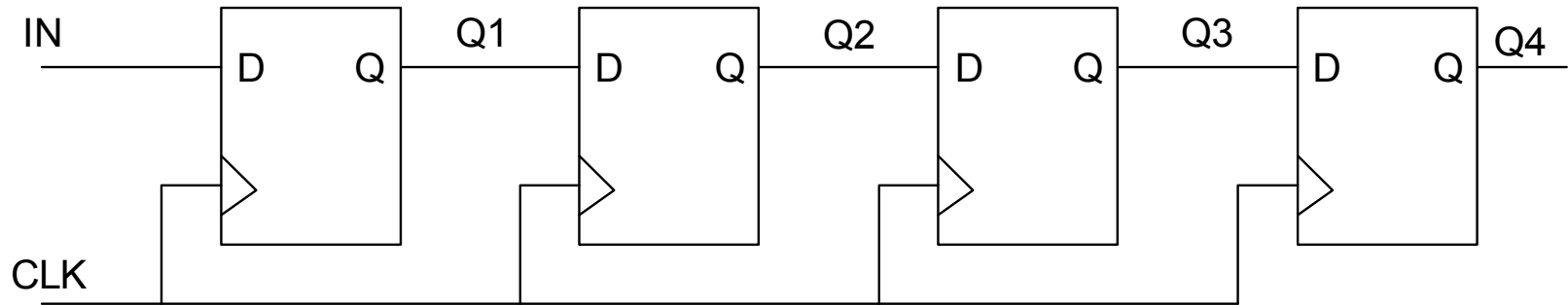
Quick Review: Shift Register



Each cycle, shift contents of this register by one bit:

In	Q1	Q2	Q3	Q4
----	----	----	----	----

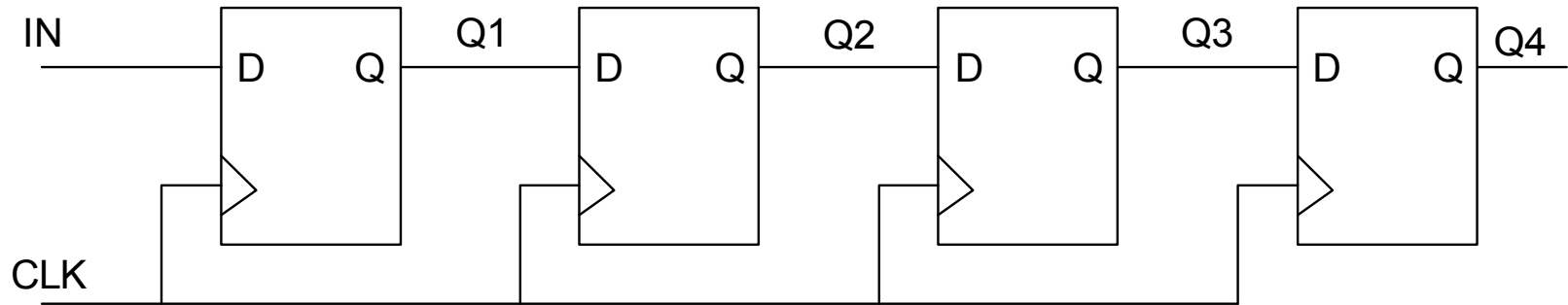
Quick Review: Shift Register



Each cycle, shift contents of this register by one bit:

	In	Q1	Q2	Q3	Q4
cycle 0	1	0	0	0	0

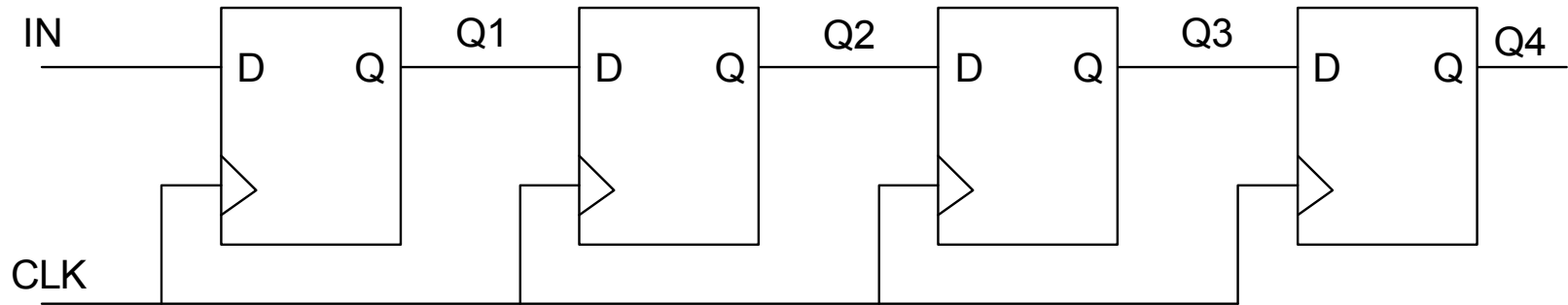
Quick Review: Shift Register



Each cycle, shift contents of this register by one bit:

	In	Q1	Q2	Q3	Q4
cycle 0	1	0	0	0	0
cycle 1	0	1	0	0	0

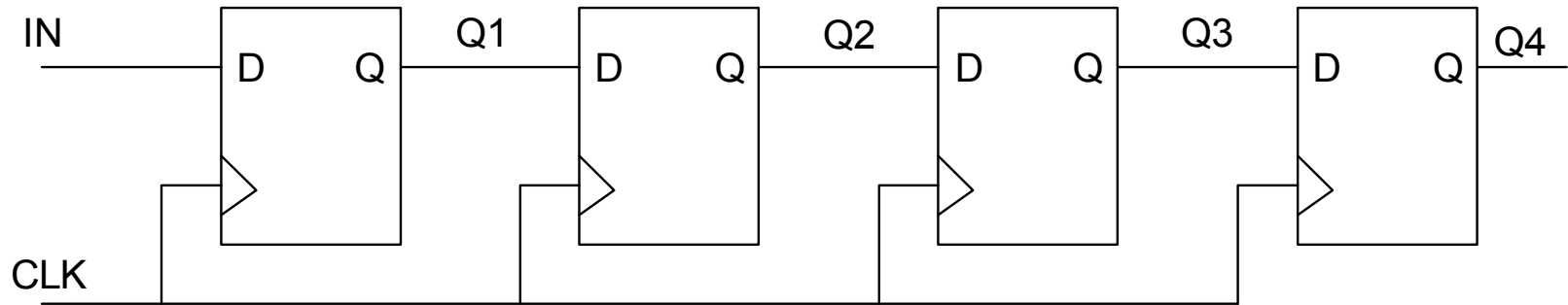
Quick Review: Shift Register



Each cycle, shift contents of this register by one bit:

	In	Q1	Q2	Q3	Q4
cycle 0	1	0	0	0	0
cycle 1	0	1	0	0	0
cycle 2	1	0	1	0	0

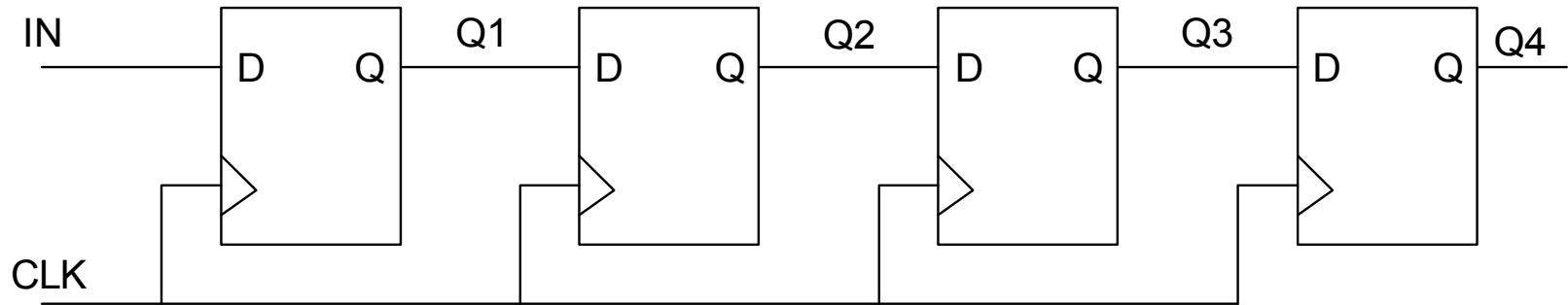
Quick Review: Shift Register



Each cycle, shift contents of this register by one bit:

	In	Q1	Q2	Q3	Q4
cycle 0	1	0	0	0	0
cycle 1	0	1	0	0	0
cycle 2	1	0	1	0	0
cycle 3	1	1	0	1	0

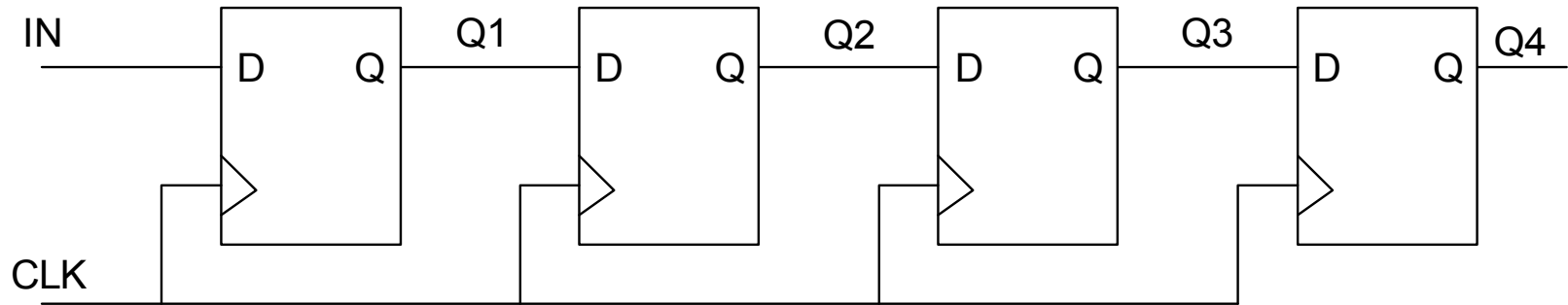
Quick Review: Shift Register



Each cycle, shift contents of this register by one bit:

	In	Q1	Q2	Q3	Q4
cycle 0	1	0	0	0	0
cycle 1	0	1	0	0	0
cycle 2	1	0	1	0	0
cycle 3	1	1	0	1	0
cycle 4	0	1	1	0	1

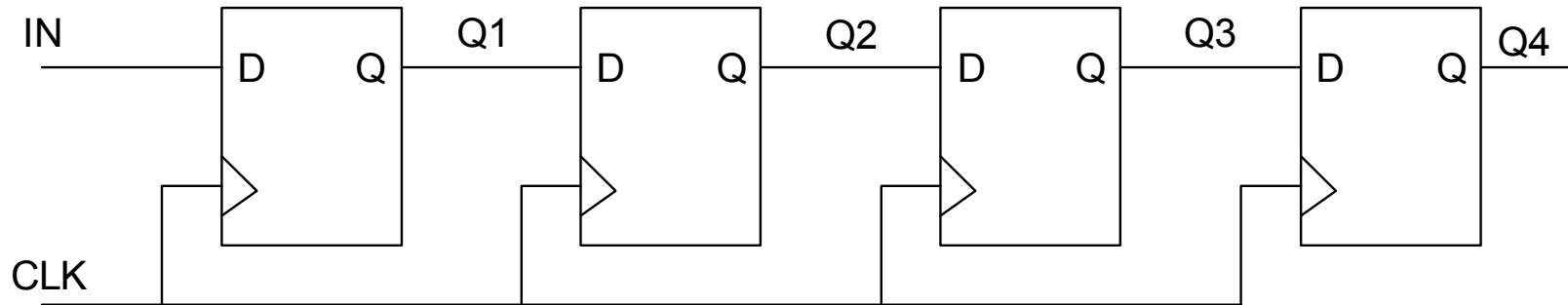
Quick Review: Shift Register



Each cycle, shift contents of this register by one bit:

	In	Q1	Q2	Q3	Q4
cycle 0	1	0	0	0	0
cycle 1	0	1	0	0	0
cycle 2	1	0	1	0	0
cycle 3	1	1	0	1	0
cycle 4	0	1	1	0	1
cycle 5	0	0	1	1	0

Quick Review: Shift Register



Each cycle, shift contents of this register by one bit:

	In	Q1	Q2	Q3	Q4
cycle 0	1	0	0	0	0
cycle 1	0	1	0	0	0
cycle 2	1	0	1	0	0
cycle 3	1	1	0	1	0
cycle 4	0	1	1	0	1
cycle 5	0	0	1	1	0

Is a 2-bit "saturating counter" the same thing as a 2-bit "shift register"?

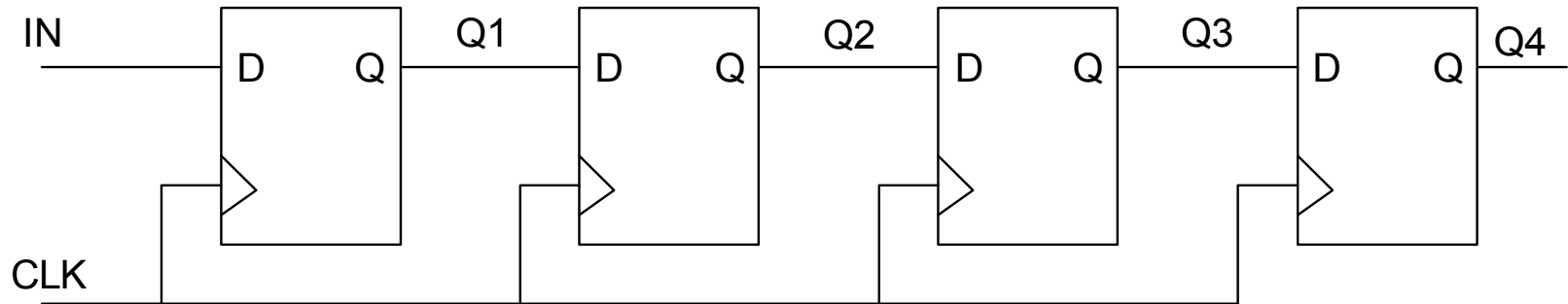
A: Yes, very sure

B: Maybe yes

C: Not sure either way

D: Maybe no

Quick Review: Shift Register



Each cycle, shift contents of this register by one bit:

	In	Q1	Q2	Q3	Q4
cycle 0	1	0	0	0	0
cycle 1	0	1	0	0	0
cycle 2	1	0	1	0	0
cycle 3	1	1	0	1	0
cycle 4	0	1	1	0	1
cycle 5	0	0	1	1	0

Is a 2-bit "saturating counter" the same thing as a 2-bit "shift register"?

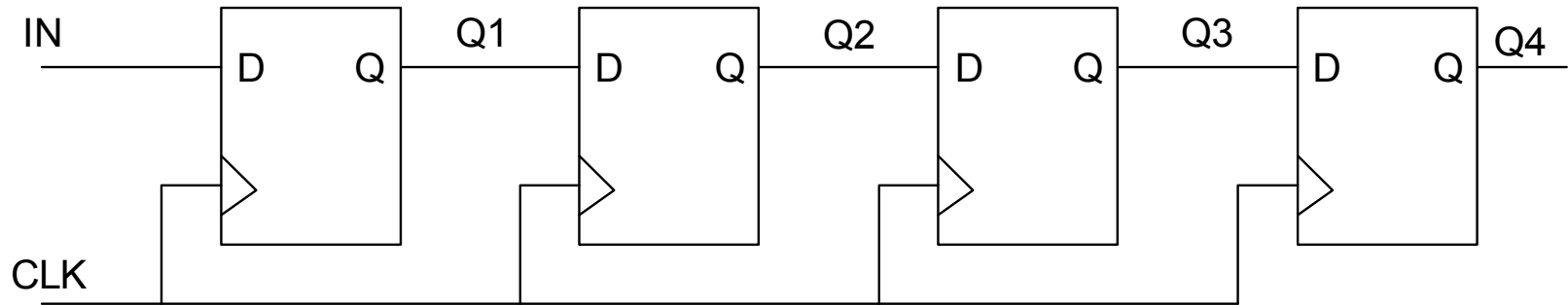
A: Yes, very sure

B: Maybe yes

C: Not sure either way

D: Maybe no

Quick Review: Shift Register



Each cycle, shift contents of this register by one bit:

	In	Q1	Q2	Q3	Q4
cycle 0	1	0	0	0	0
cycle 1	0	1	0	0	0
cycle 2	1	0	1	0	0
cycle 3	1	1	0	1	0
cycle 4	0	1	1	0	1
cycle 5	0	0	1	1	0

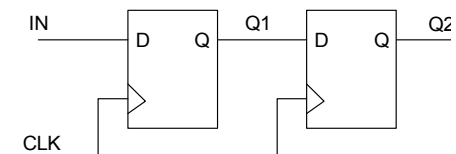
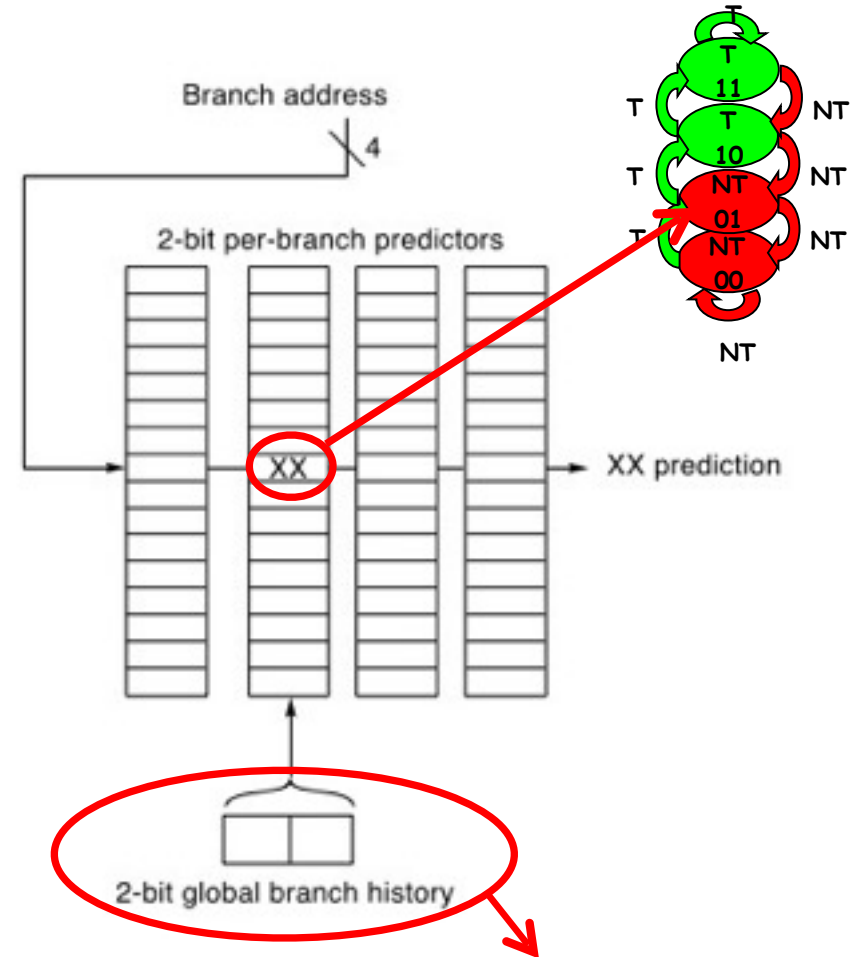
Increase “History Length”

(e.g., correlate against last 2 branches)

Real code example from SPEC89 benchmark “eqnott”:

```
if( aa==2 )  
    aa=0;  
if( bb==2 )  
    bb=0;  
if( aa!=bb ) {
```

goal: want better prediction for this branch

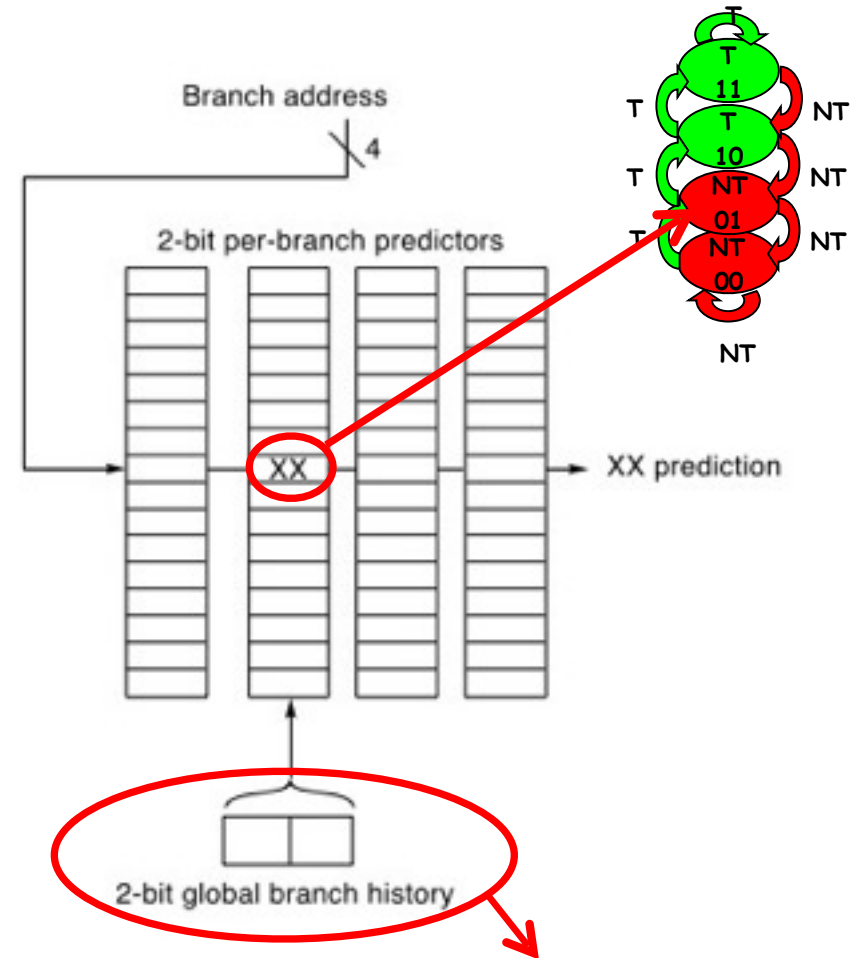


Increase “History Length”

(e.g., correlate against last 2 branches)

Real code example from SPEC89 benchmark “eqnott”:

```
if( aa==2 )  
    aa=0;  
if( bb==2 )  
    bb=0;  
if( aa!=bb ) {
```



True or False? The 2-bit global branch history in the figure is a saturating counter?

- A: True, I'm very certain
- B: True, but I'm not certain
- C: Not sure
- D: False, but I'm not certain

Increase “History Length”

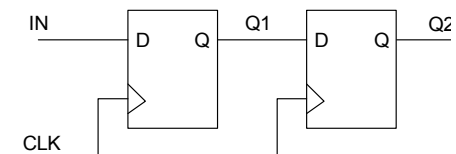
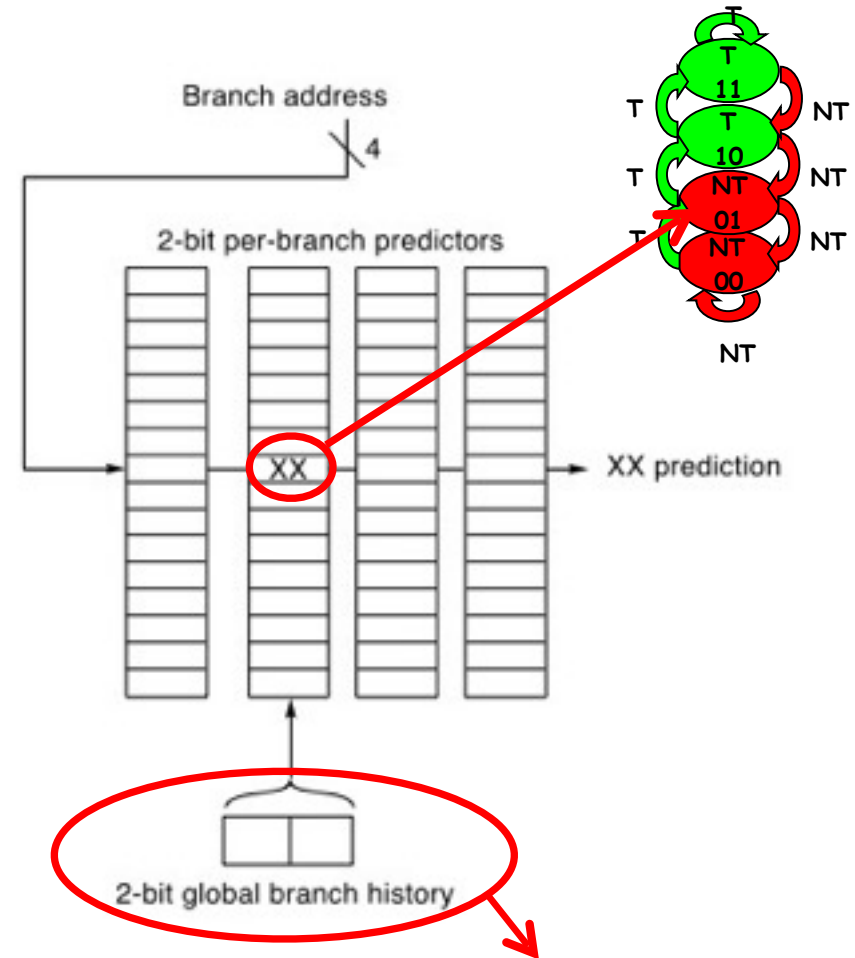
(e.g., correlate against last 2 branches)

Real code example from SPEC89 benchmark “eqnott”:

```
if( aa==2 )
    aa=0;
if( bb==2 )
    bb=0;
if( aa!=bb ) {
```

True or False? The 2-bit global branch history in the figure is a saturating counter?

- A: True, I'm very certain
- B: True, but I'm not certain
- C: Not sure
- D: False, but I'm not certain



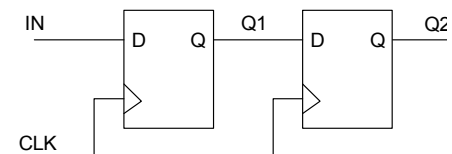
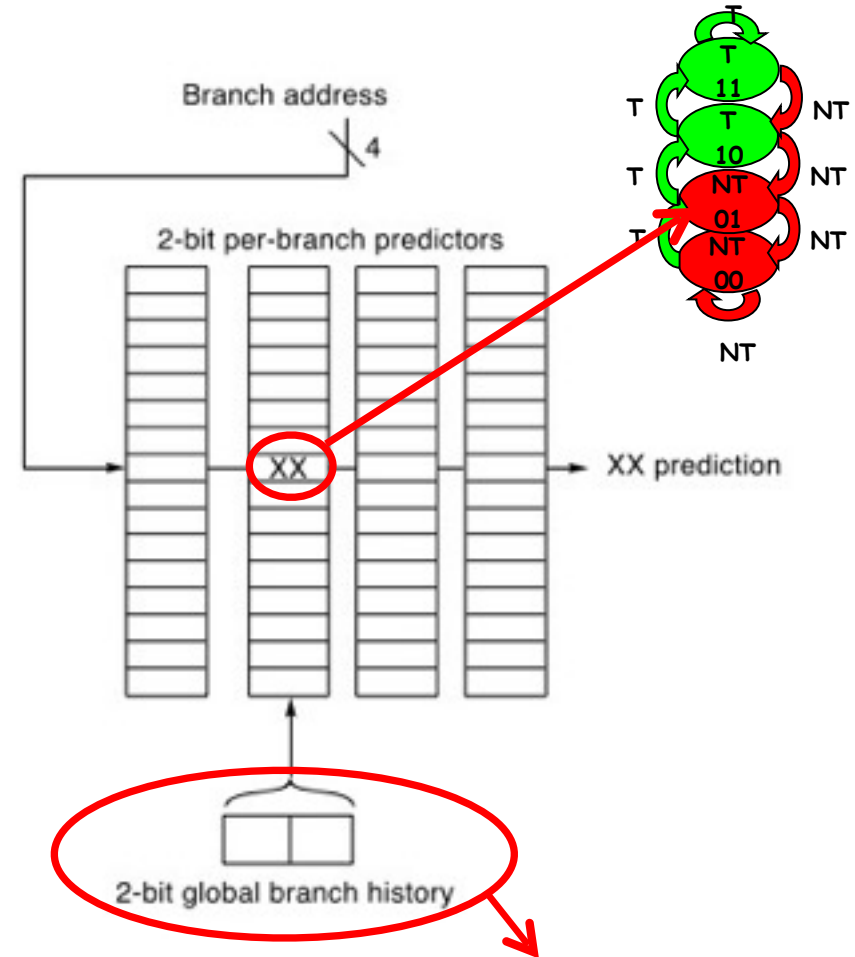
Increase “History Length”

(e.g., correlate against last 2 branches)

Real code example from SPEC89 benchmark “eqnott”:

```
if( aa==2 )  
    aa=0;  
if( bb==2 )  
    bb=0;  
if( aa!=bb ) {
```

goal: want better prediction for this branch



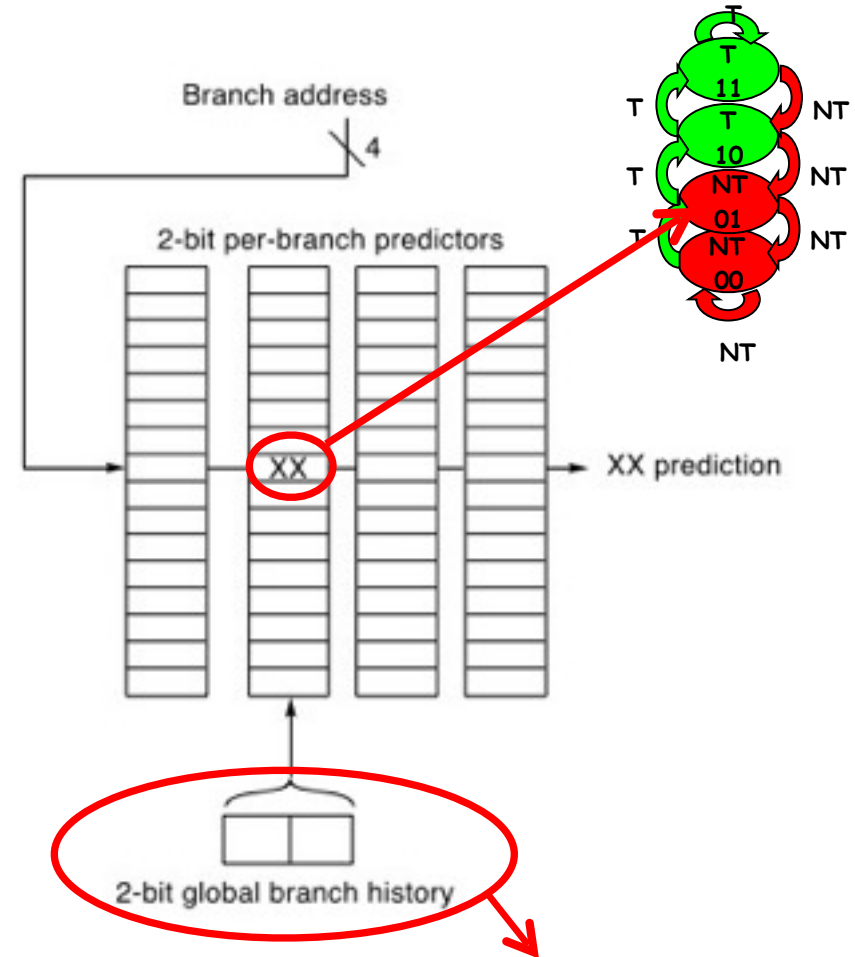
Increase “History Length”

(e.g., correlate against last 2 branches)

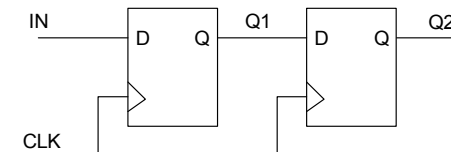
Real code example from SPEC89 benchmark “eqnott”:

```
if( aa==2 )  
    aa=0;  
if( bb==2 )  
    bb=0;  
if( aa!=bb ) {
```

goal: want better prediction for this branch



Global branch history = shift register (recall from 353) “Shift in” branch outcomes of earlier branches.



Increase “History Length”

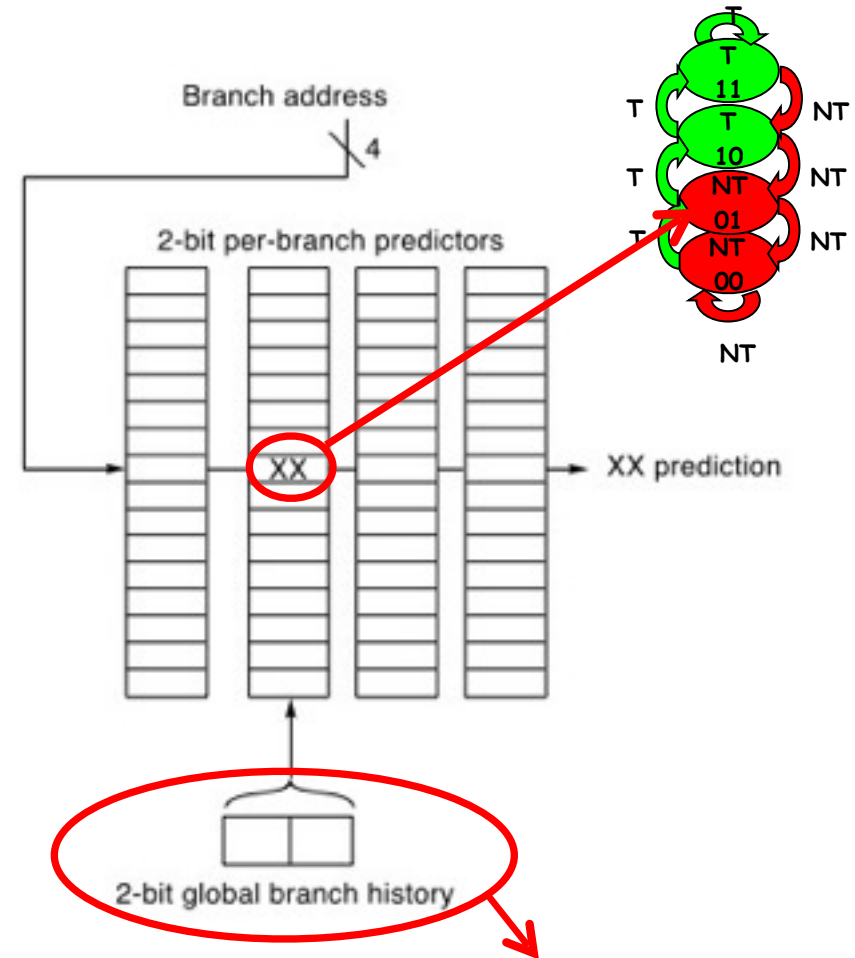
(e.g., correlate against last 2 branches)

Real code example from SPEC89 benchmark “eqnott”:

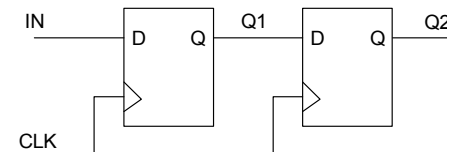
```
if( aa==2 )  
    aa=0;  
if( bb==2 )  
    bb=0;  
if( aa!=bb ) {
```

True or False? A 2-bit saturating counter tracks the outcome of the two most recent branches.

A: True, I'm very certain
B: True, but I'm not certain
C: Not sure
D: False, but I'm not certain



register (recall from
s of earlier branches.



Increase “History Length”

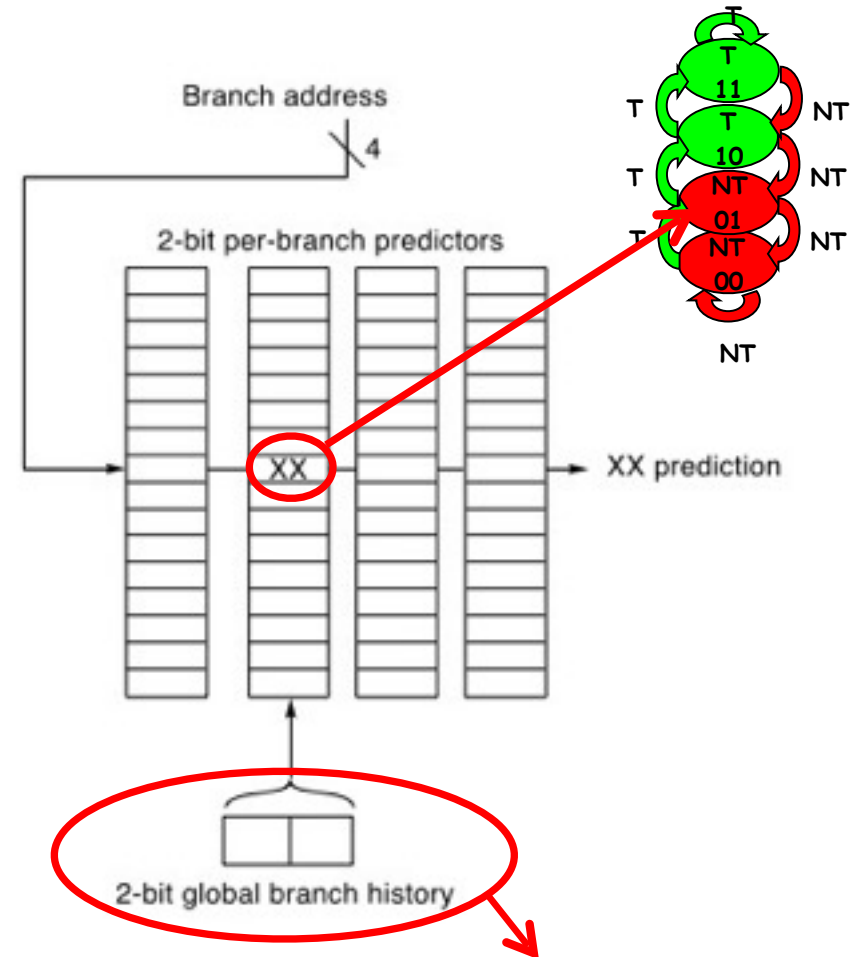
(e.g., correlate against last 2 branches)

Real code example from SPEC89 benchmark “eqnott”:

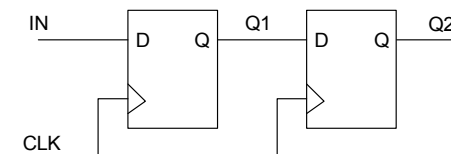
```
if( aa==2 )  
    aa=0;  
if( bb==2 )  
    bb=0;  
if( aa!=bb ) {
```

True or False? A 2-bit saturating counter tracks the outcome of the two most recent branches.

- A: True, I'm very certain
- B: True, but I'm not certain
- C: Not sure
- D: False, but I'm not certain



register (recall from
s of earlier branches.



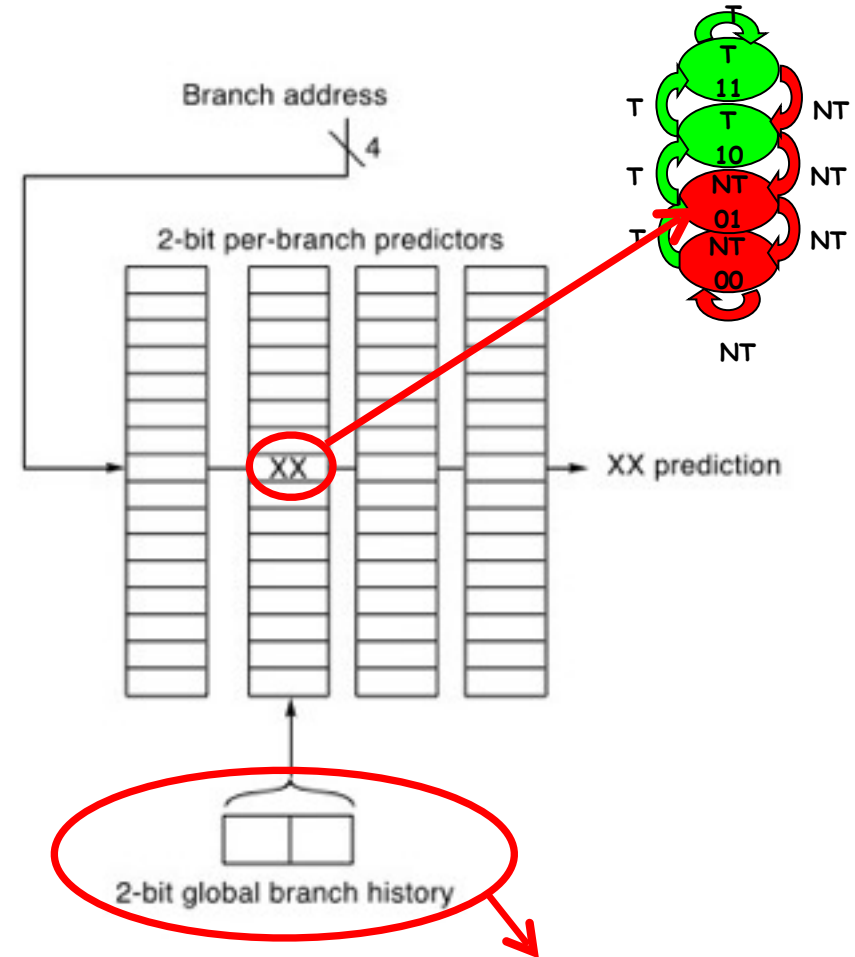
Increase “History Length”

(e.g., correlate against last 2 branches)

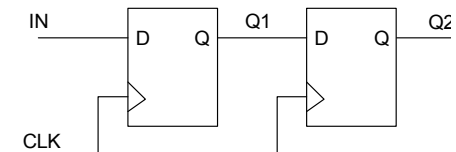
Real code example from SPEC89 benchmark “eqnott”:

```
if( aa==2 )  
    aa=0;  
if( bb==2 )  
    bb=0;  
if( aa!=bb ) {
```

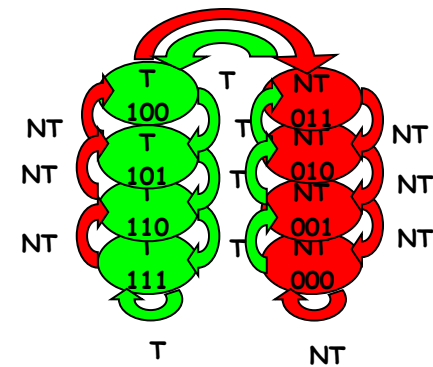
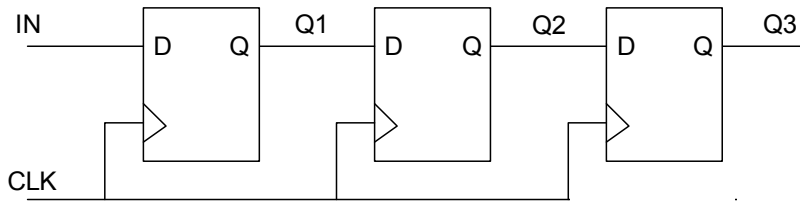
goal: want better prediction for this branch



Global branch history = shift register (recall from 353) “Shift in” branch outcomes of earlier branches.



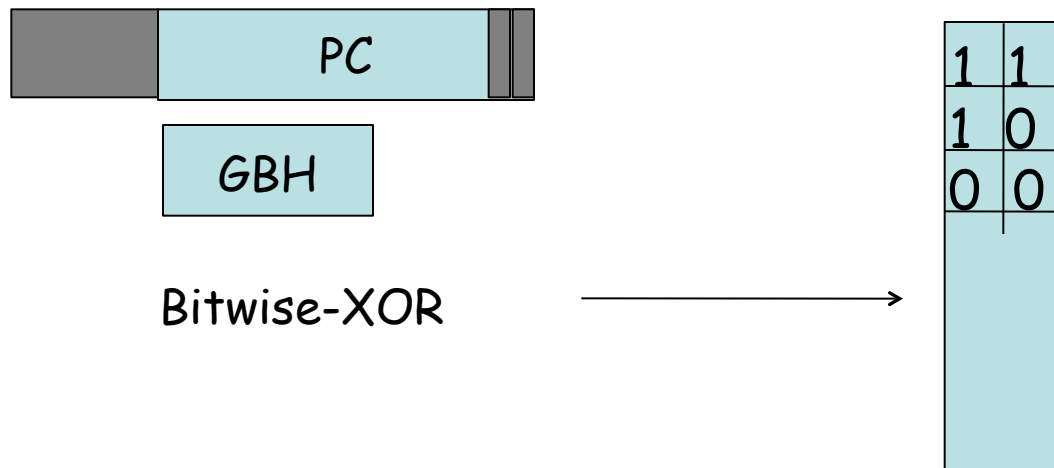
Track outcome of 3-branches



- Which is “correct” one to use?

Better Use of Silicon Area

- Increasing the number of bits of history causes area of table to grow exponentially.
- Better approach is to “hash” PC and global branch history (GBH) bits together using bit-wise XOR (in C, “^” operator).
- This is known as the “GShare” branch predictor

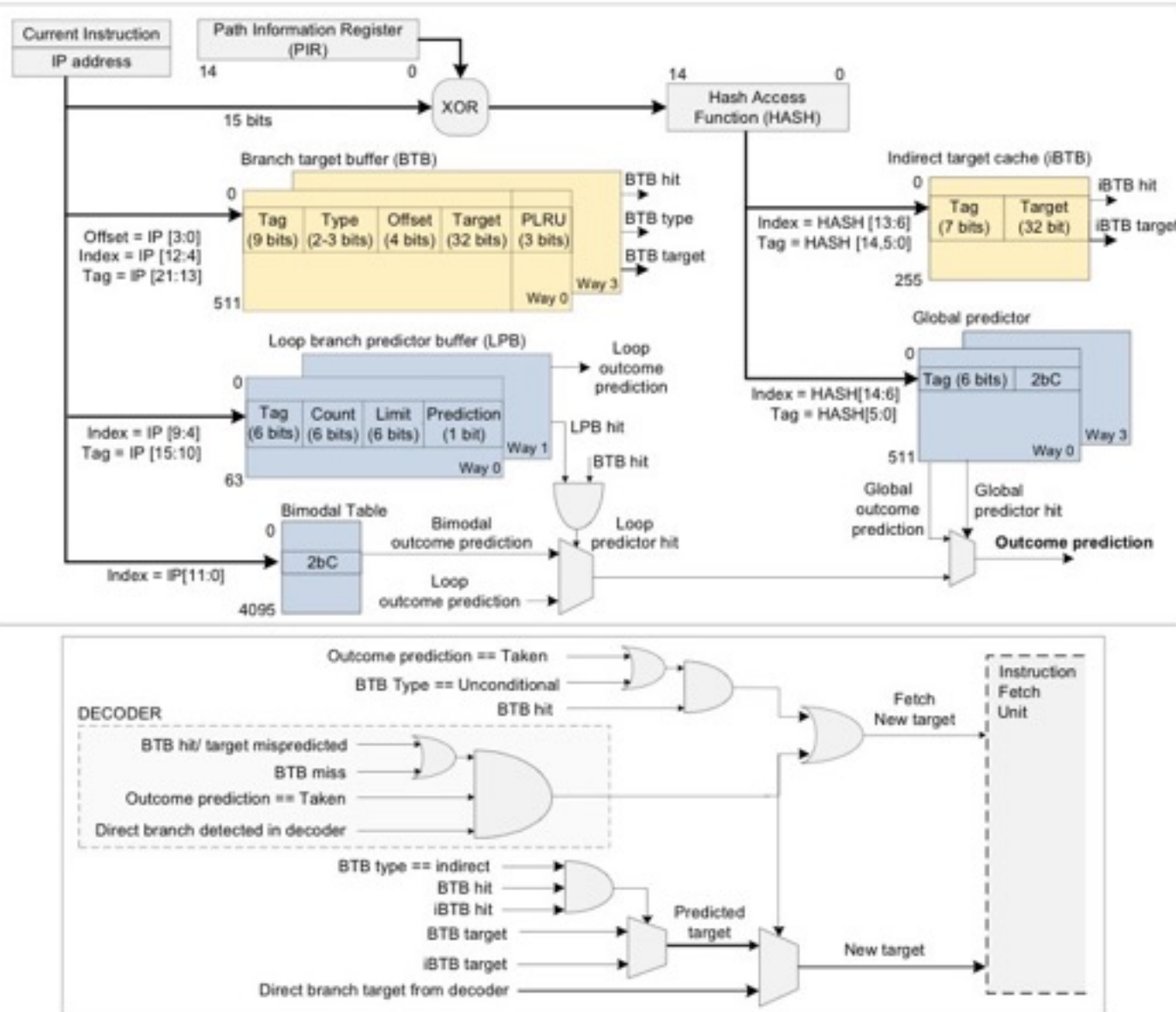


Pentium M Branch Predictor

- Intel/AMD tend not to disclose many details about their branch predictors

- It is possible to infer structure of branch predictor using “microbenchmarks”.

The figure is from a recent academic paper. The authors determined this structure by running microbenchmarks on the Pentium M and using VTune.



Summary of Slide Set 8

- Branch prediction is motivated by need to keep pipeline filled with instructions. This is especially important for processors that use out-of-order execution and deep pipelines. It motivates significant investment of silicon area.
- A 1-bit predictor uses the last outcome of a branch to predict the next outcome.
- A 2-bit predictor uses a state machine. The current state indicates the prediction. The outcome determines the next state.
- A correlating branch predictor uses information about other branch outcomes when making a prediction about the current branch.