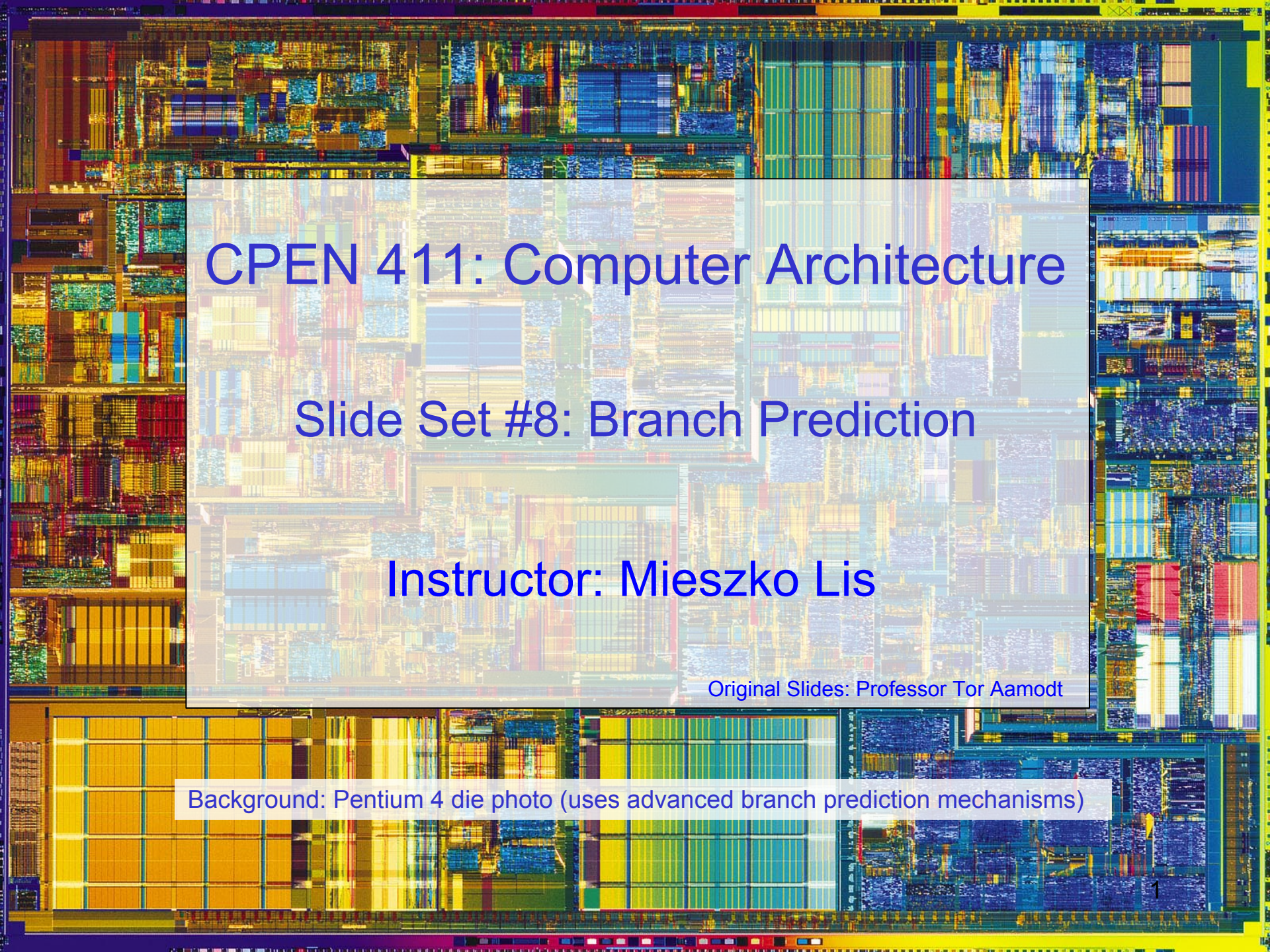


CPEN 411: Computer Architecture

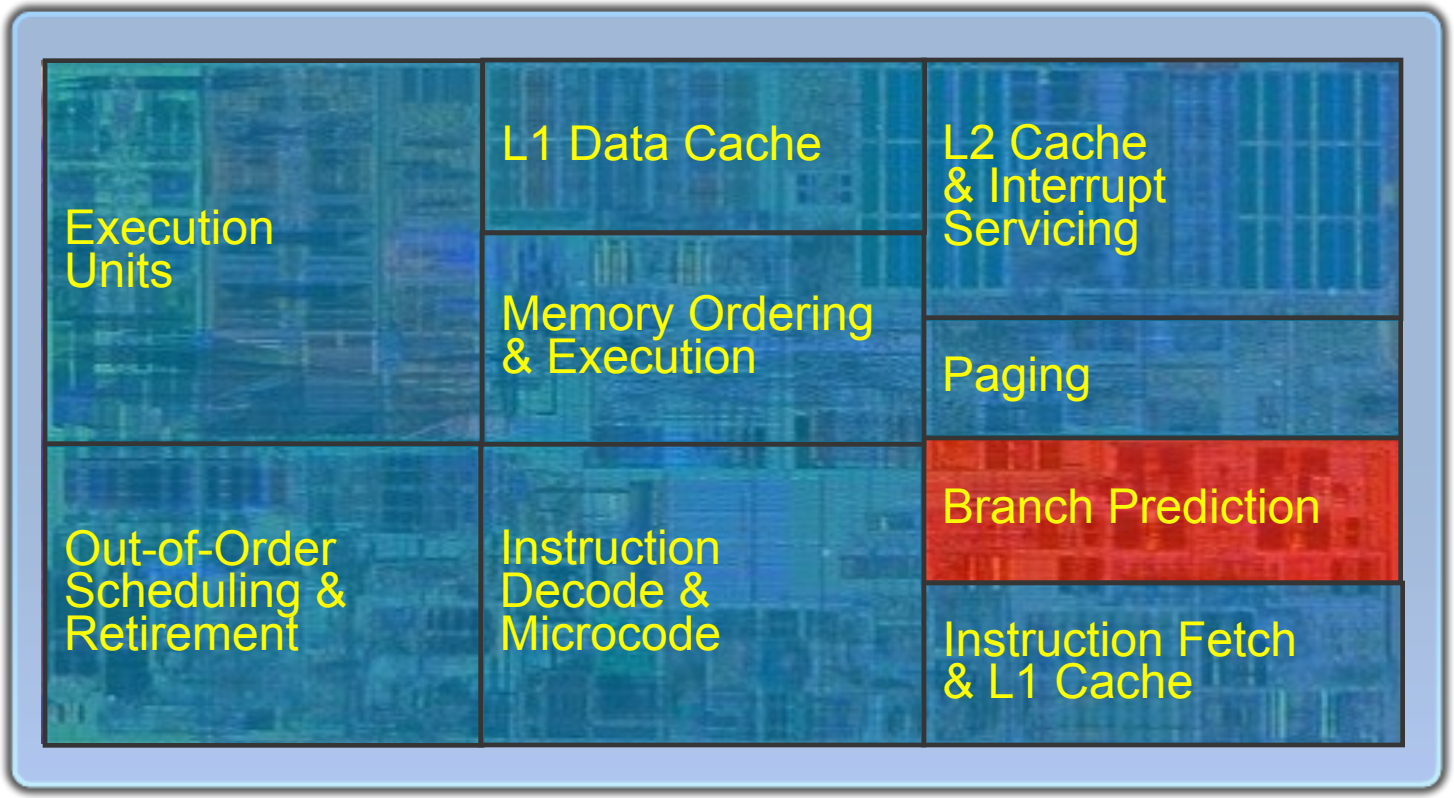
Slide Set #8: Branch Prediction

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

- 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 2.1)

- 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.

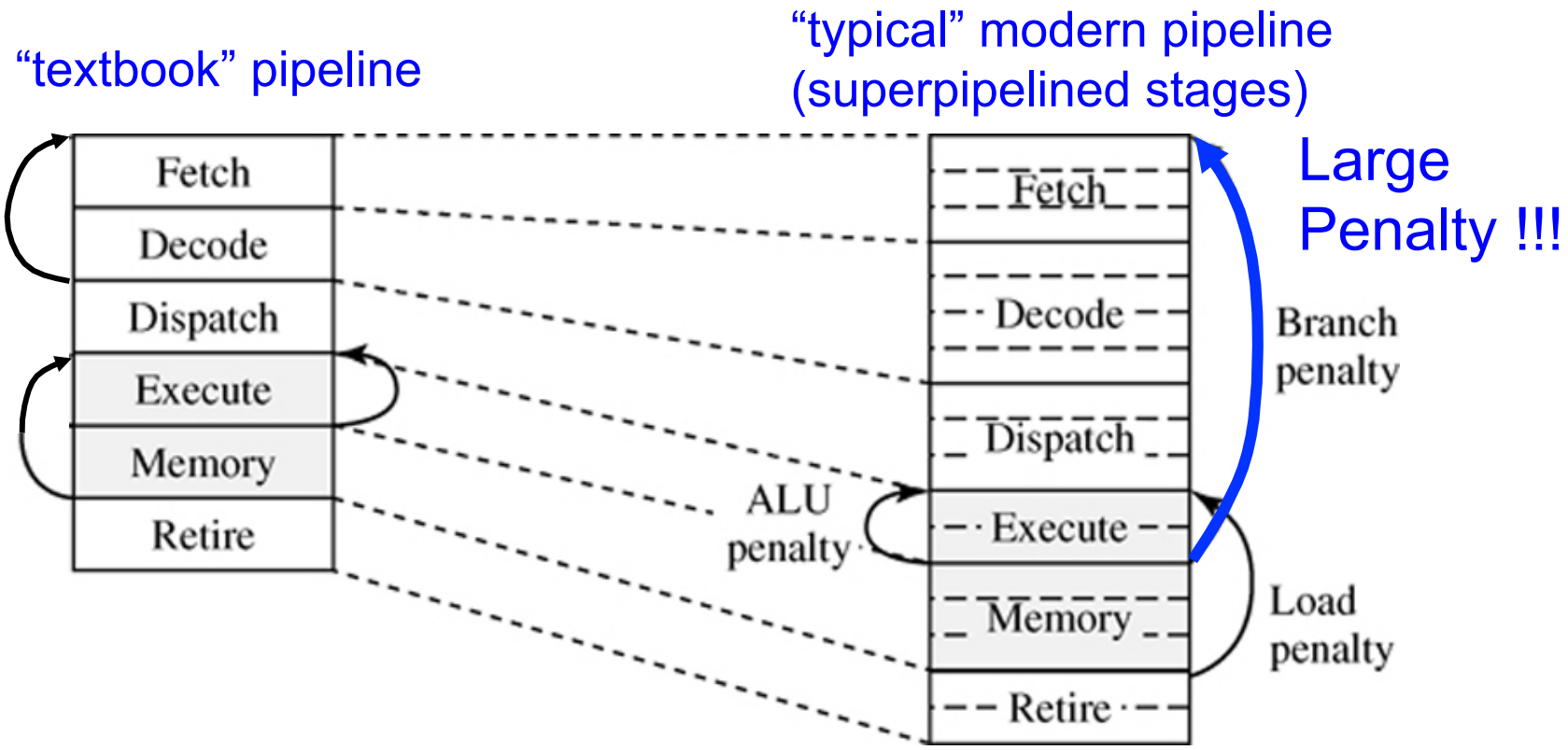
Clock cycle 3: EX stage resolves branch, instructions in ID ("branch+1") and IF ("branch+2") turned into "no-ops"… they are "flushed", correct next instruction ("branch target") fetched on clock cycle 4.

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.

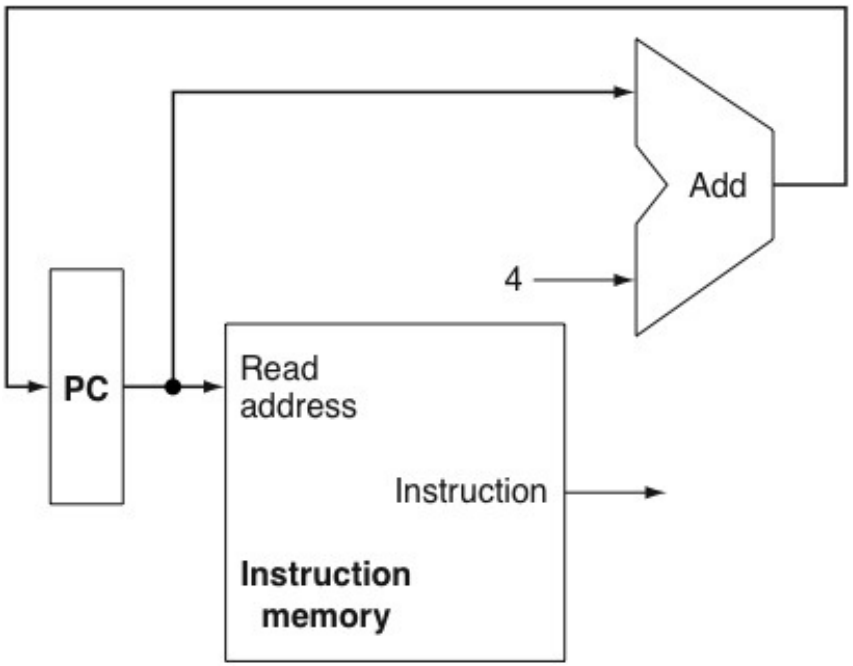
Branch Instructions: Two Problems

- **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:



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".

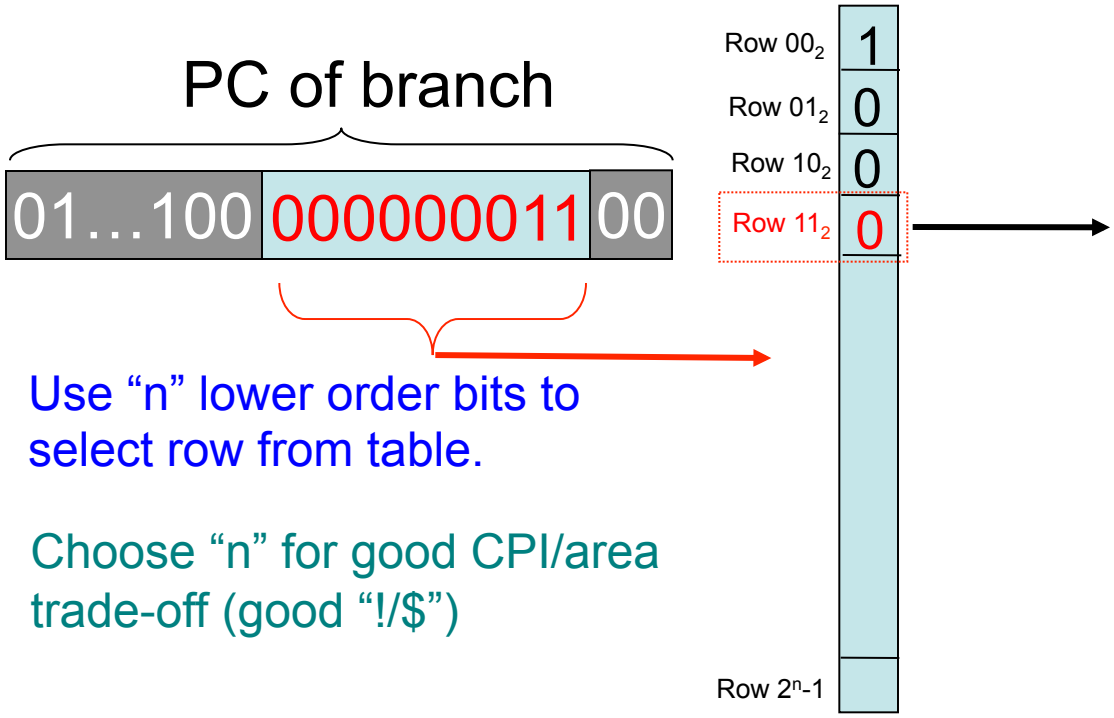
Example Prediction Problem

```
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



Use “n” lower order bits to select row from table.

Choose “n” for good CPI/area trade-off (good “!/$”)

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)

2^n x 1 bit table

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

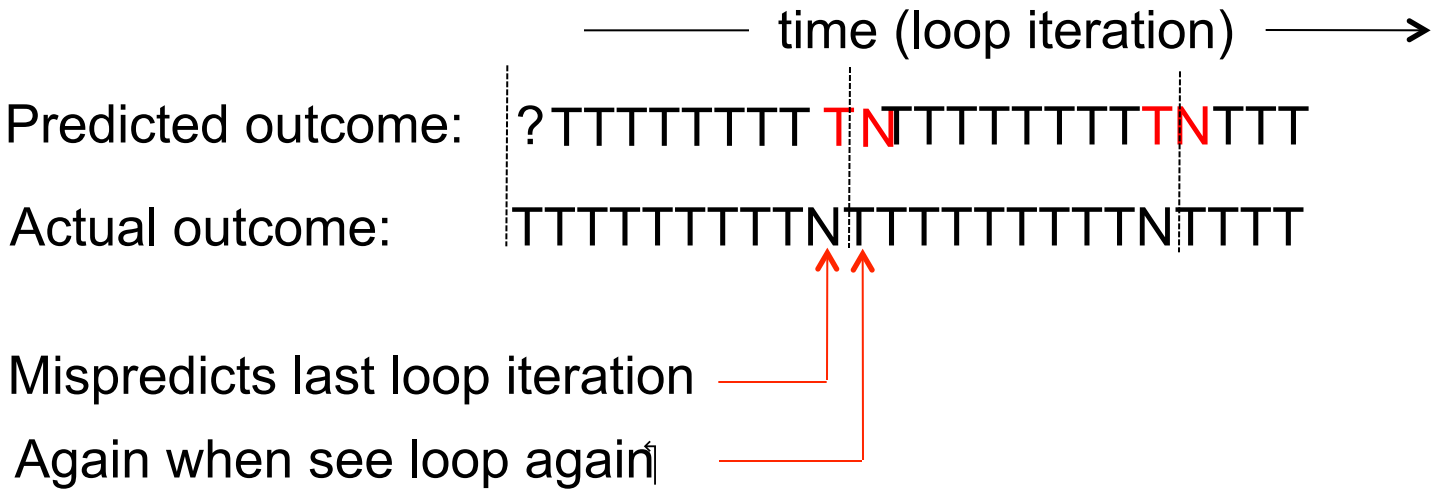
Some of you might have learned about “hash tables” in one of your programming courses. You might note that the hardware used for branch prediction does something similar. (If you don’t remember, don’t worry about it.)

Example, cont'd

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

1-bit Branch Predictor

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

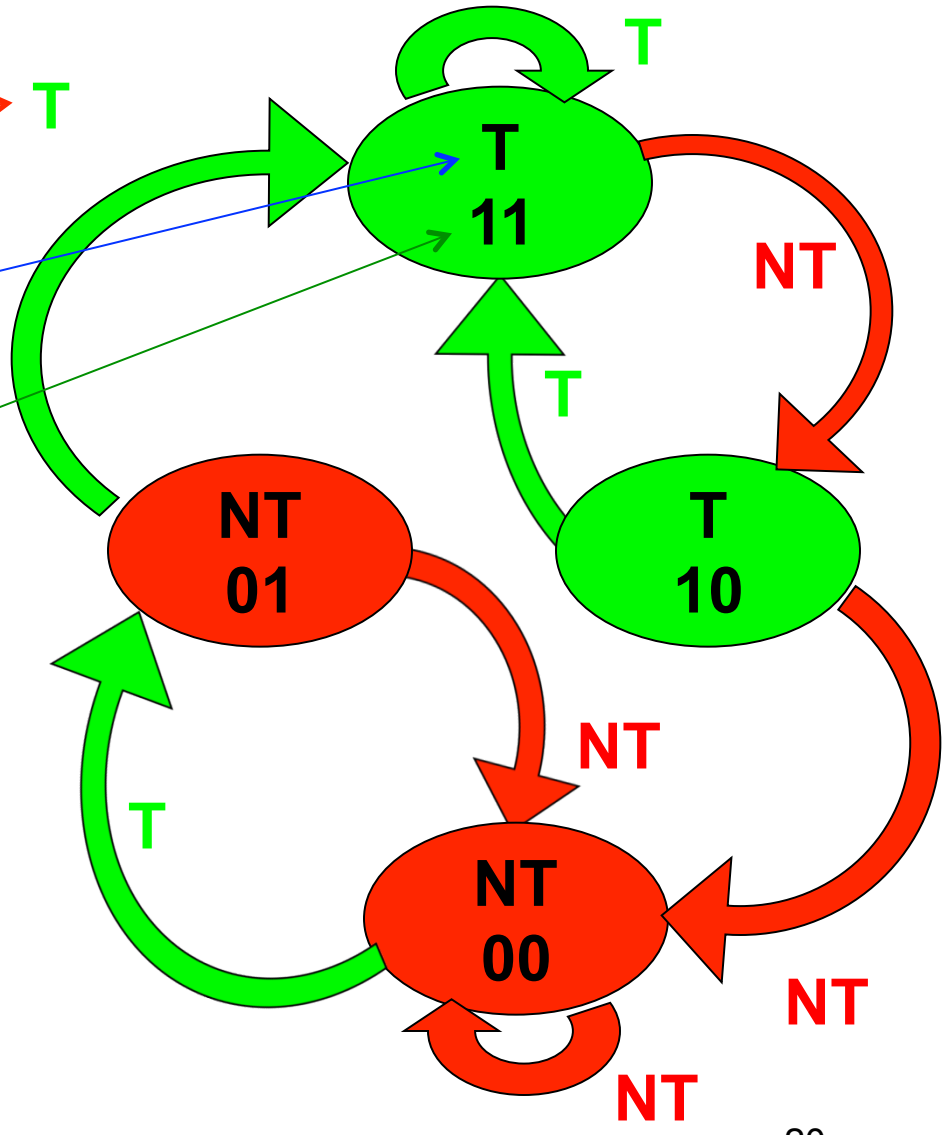
ideas to get rid of mispredictions?

2-bit Branch Predictor

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

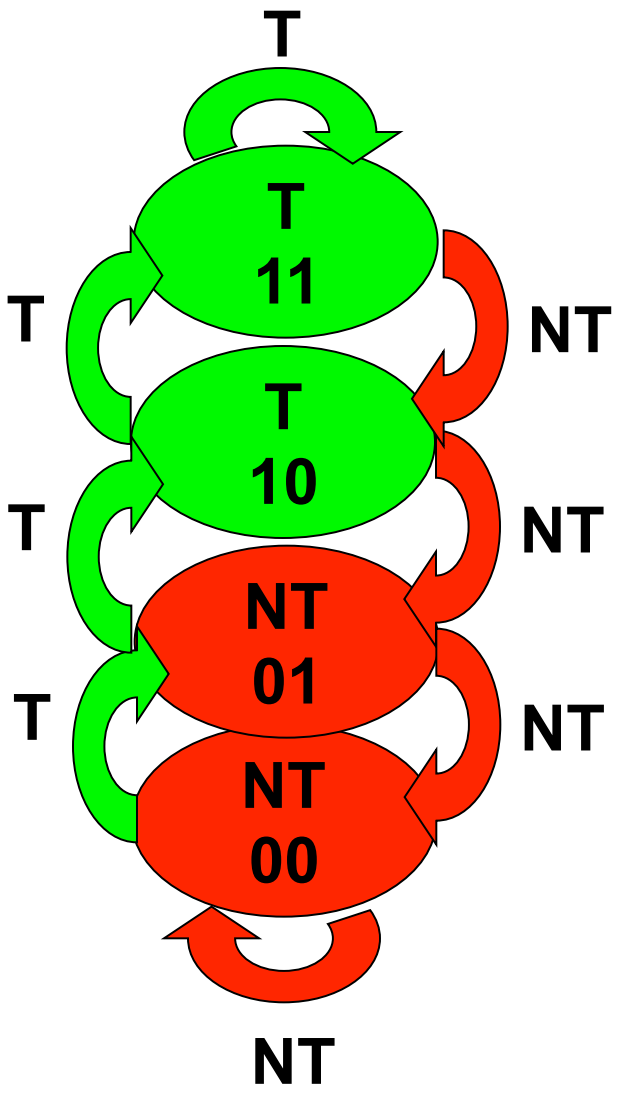
Current state provides prediction:
T= “Taken”/ NT=“not taken”

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

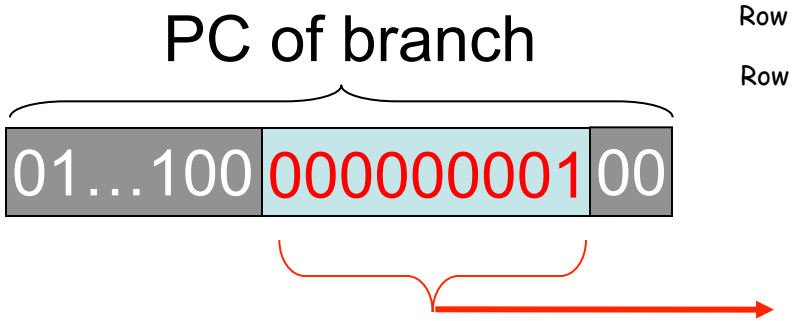
2-bit “Saturating Counter” Predictor

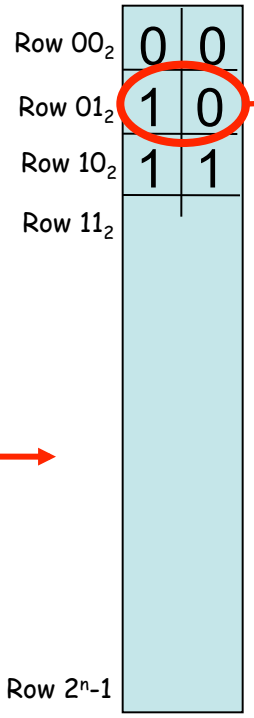


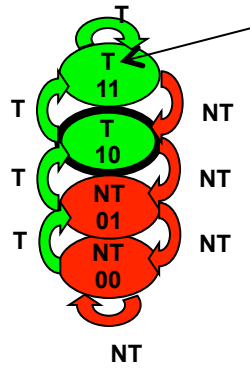
2-bit counter that is incremented after a taken branch, and decrement after a not-taken branch.

Prediction is simply most significant bit.

2-Bit Predictor







prediction based upon current state.

next state depends upon current state and actual outcome.

Contents of each row specify state in state machine. Make prediction based upon current state. *E.g., if in state "10", saturating counter state machine says predict "taken".*

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

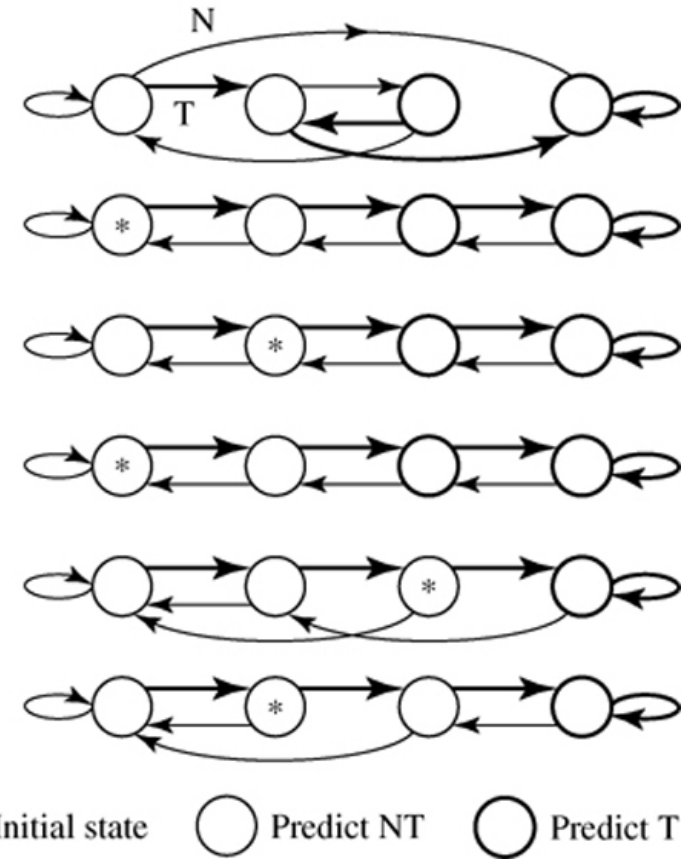
Size of table (in bits) =

2^n x 2 (n=# of PC bits used)

Other 2-bit predictors

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

Benchmark	Optimal	"Counter"
spice2g6	97.2	97.0
doduc	94.3	94.3
gcc	89.1	89.1
espresso	89.1	89.1
li	87.1	86.8
eqntott	87.9	87.2



Study: Ravi Nair, IBM, 1992

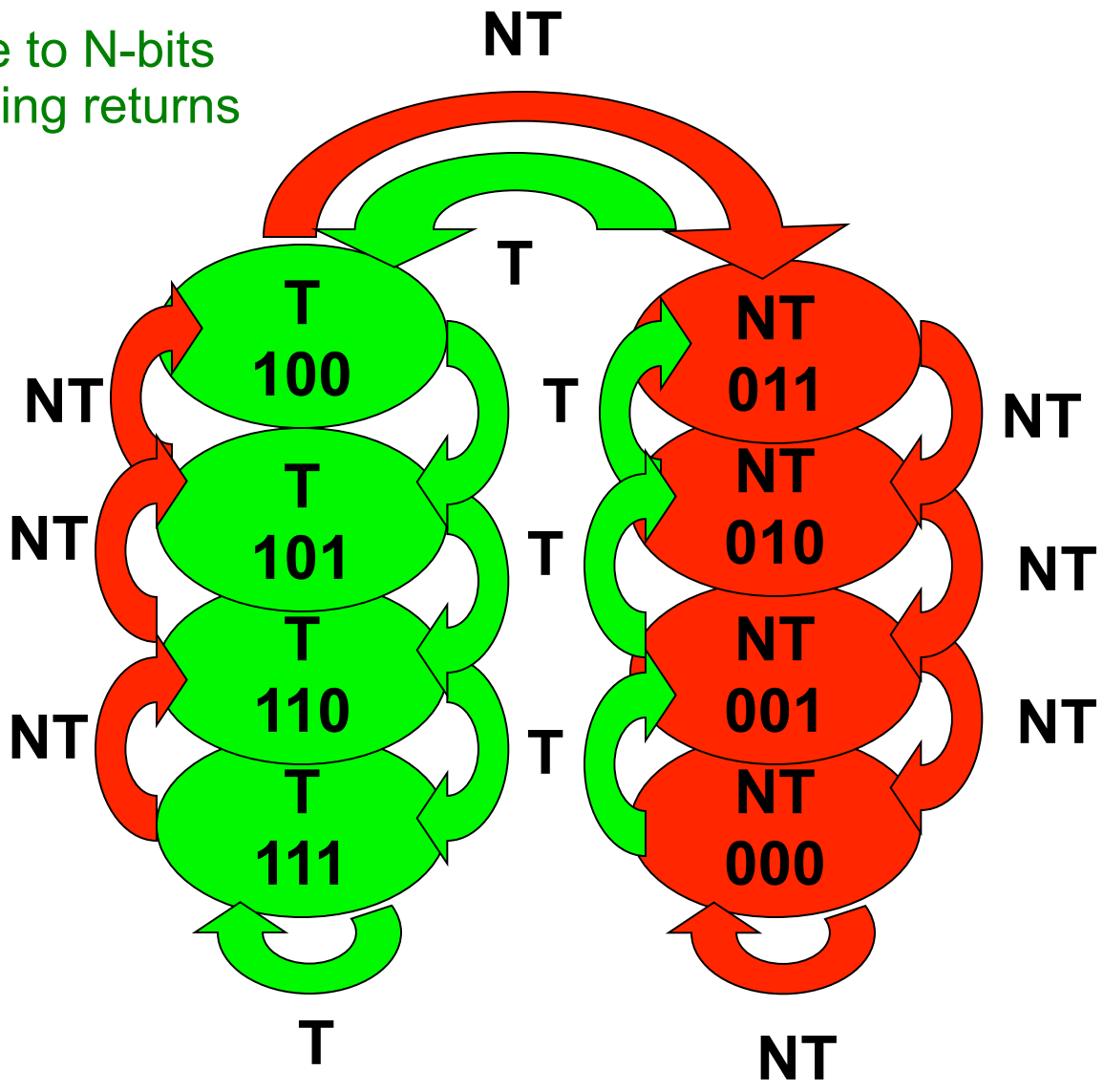
Saturating 2-bit counter is close to "optimal" among all 2-bit state machines (though we will see better predictors).

```
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).

3-bit “Saturating Counter” Predictor

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

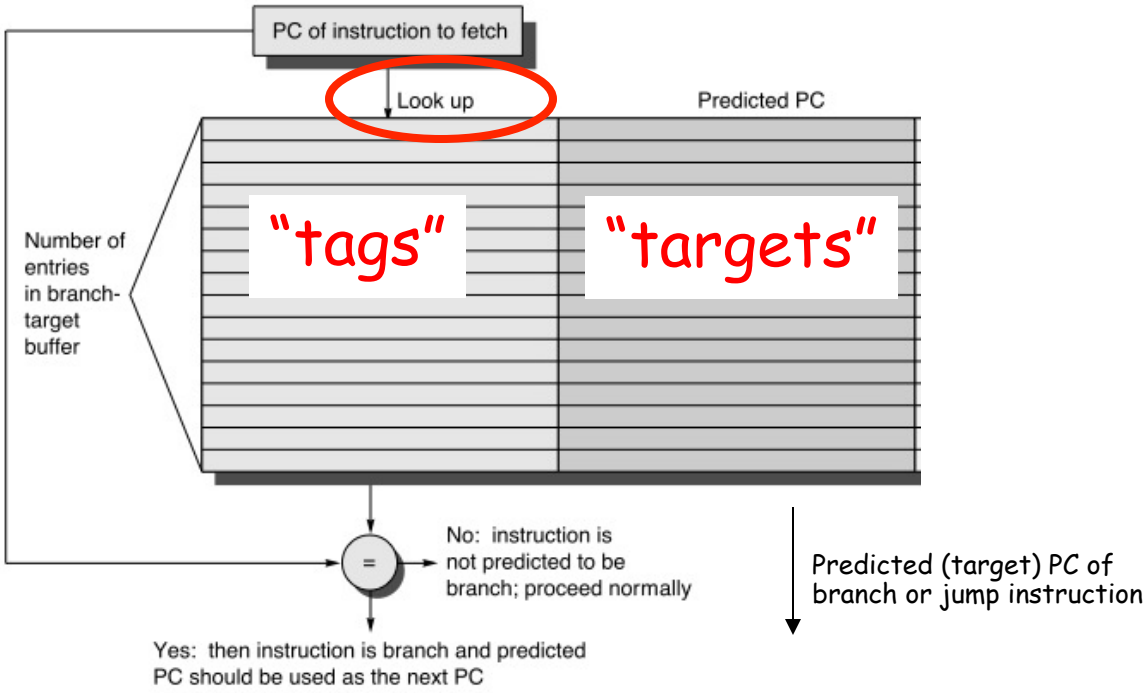


Branch Prediction "FAQ"

- **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)



- 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
    …
}
```


		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

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, ...

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 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:
```

instruction addresses (binary)

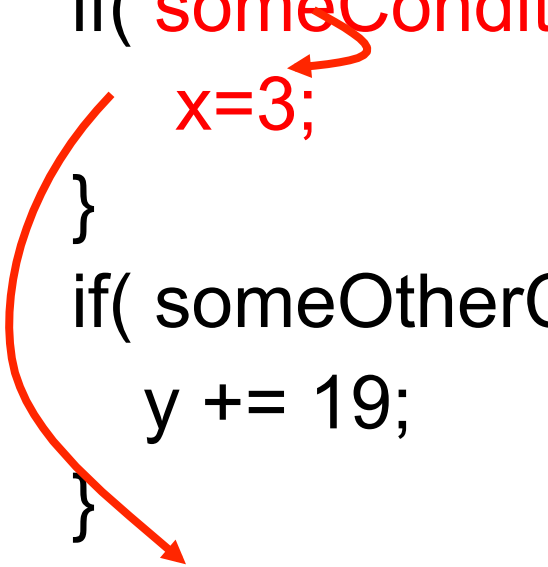
consider initial values for

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 T, likely b2 is T)

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();
}
```



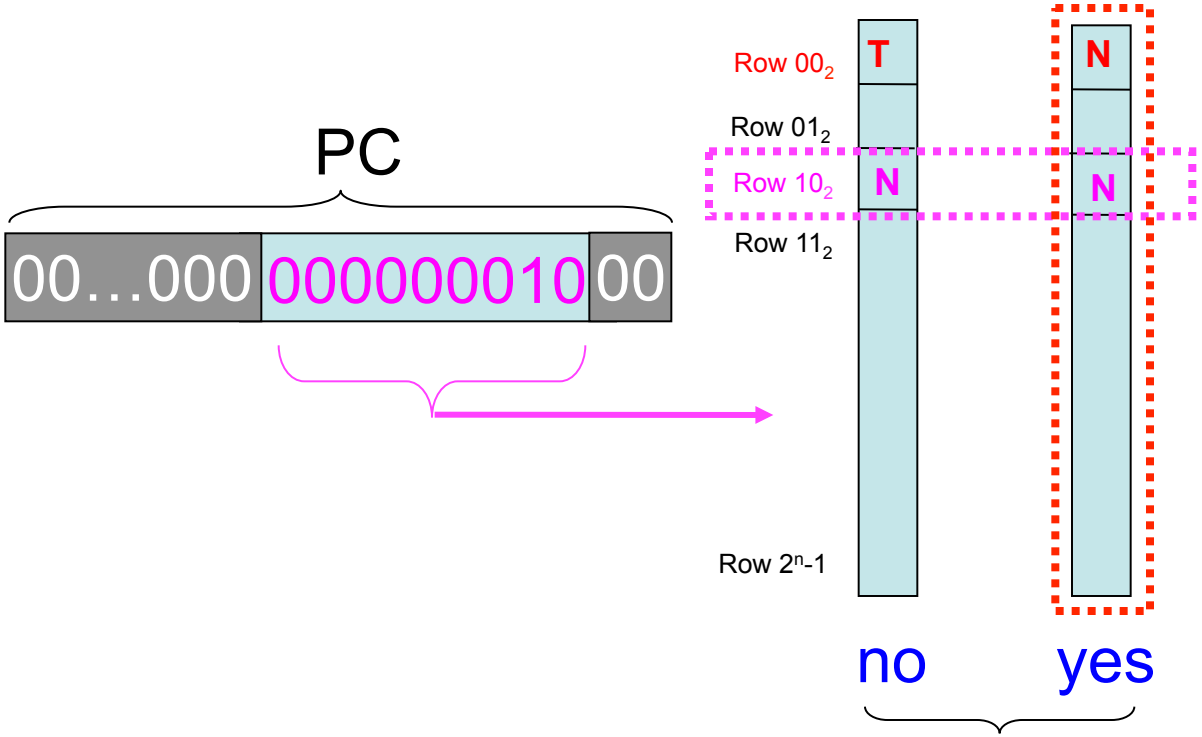
A (Silly) Analogy…

- 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 with1-Bit of Correlation:

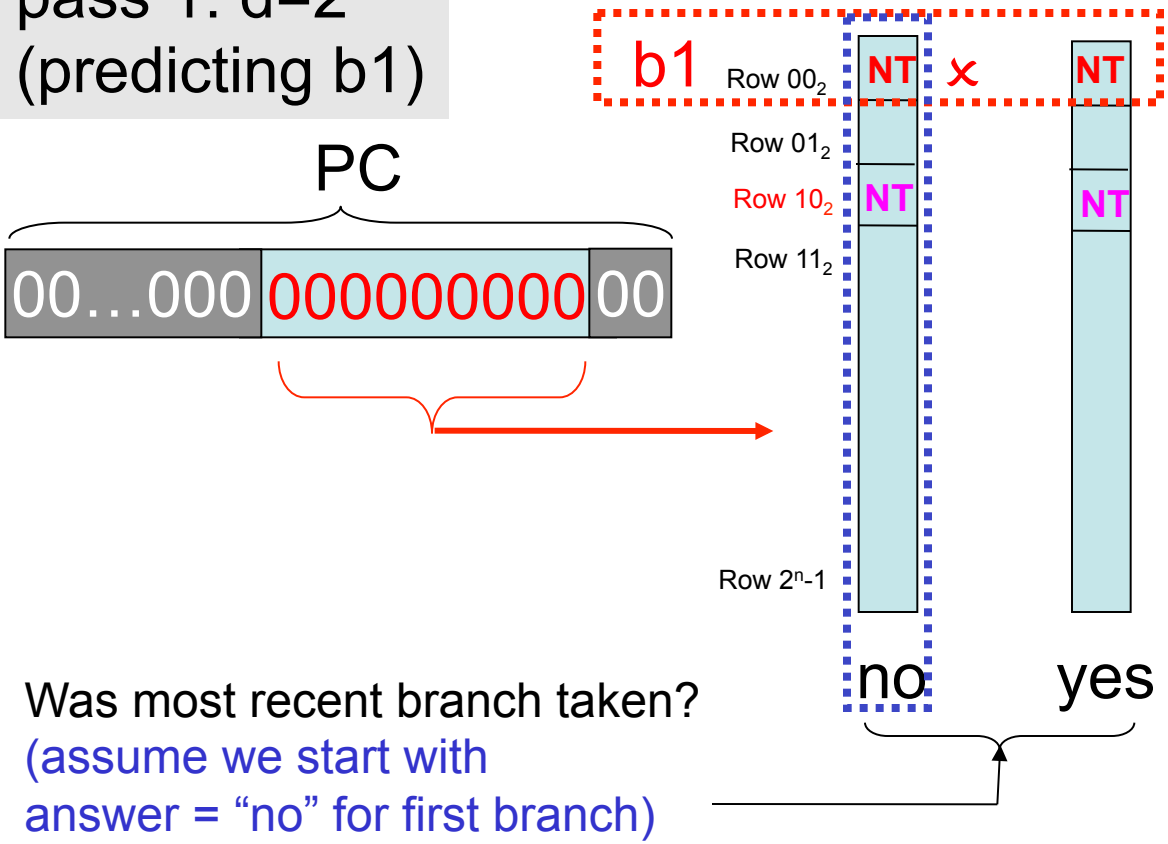



```
        if (d!=0) goto L1
            d=1;
L1:     if (d!=1) goto L2
...
L2:
...
```

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 Predictor with 1-Bit of Correlation:

pass 1: d=2
(predicting b1)



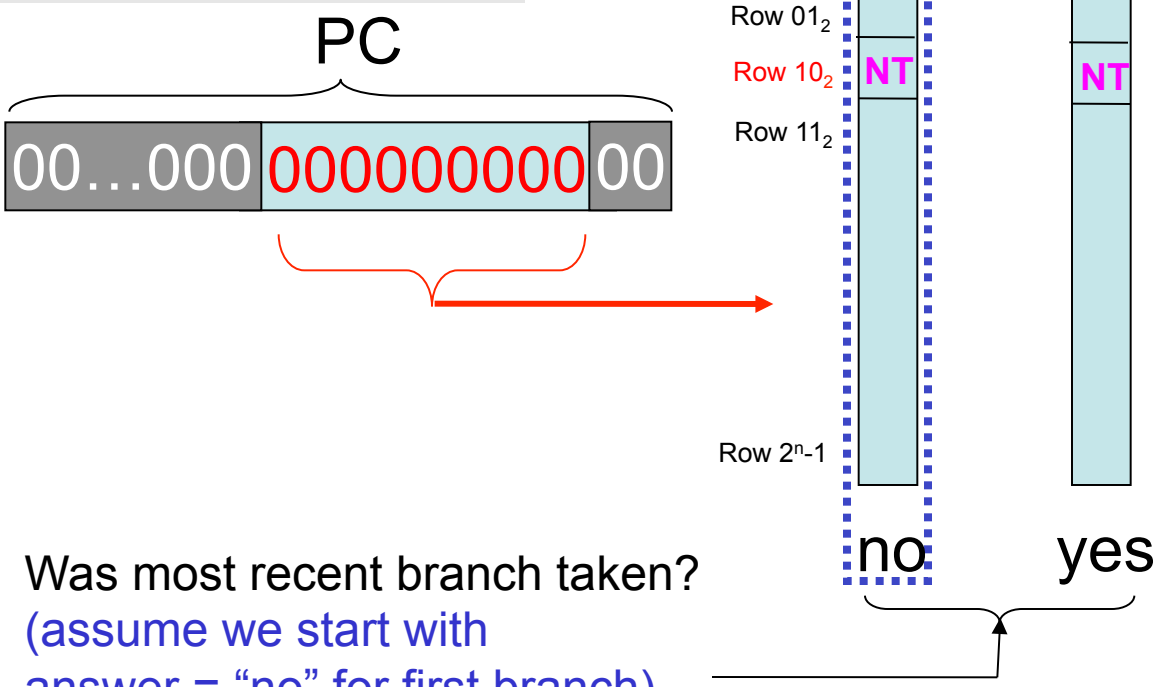

```
        if (d!=0) goto L1;//b1
            d=1;
L1:     if (d!=1) goto L2;//b2
...
L2:
...
```

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,
(after executing b1,
before executing b2)



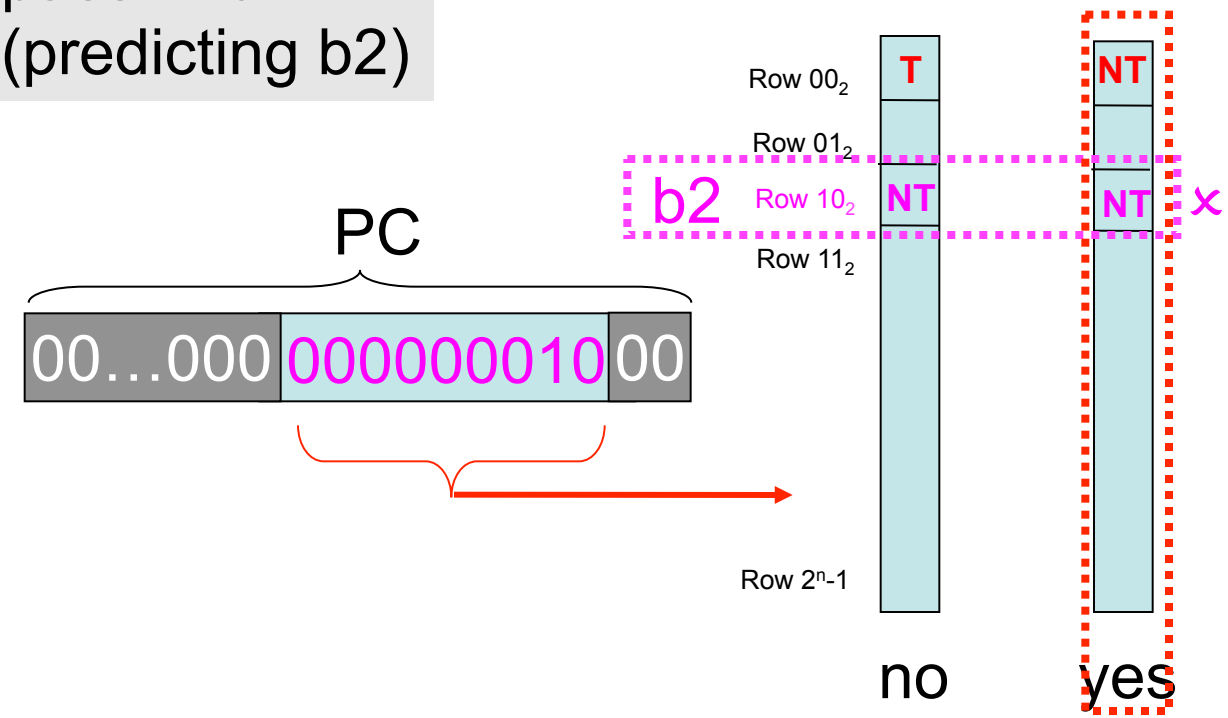

```
        if (d!=0) goto L1;//b1
            d=1;
L1:     if (d!=1) goto L2;//b2
        ...
L2:
        ...
```

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)



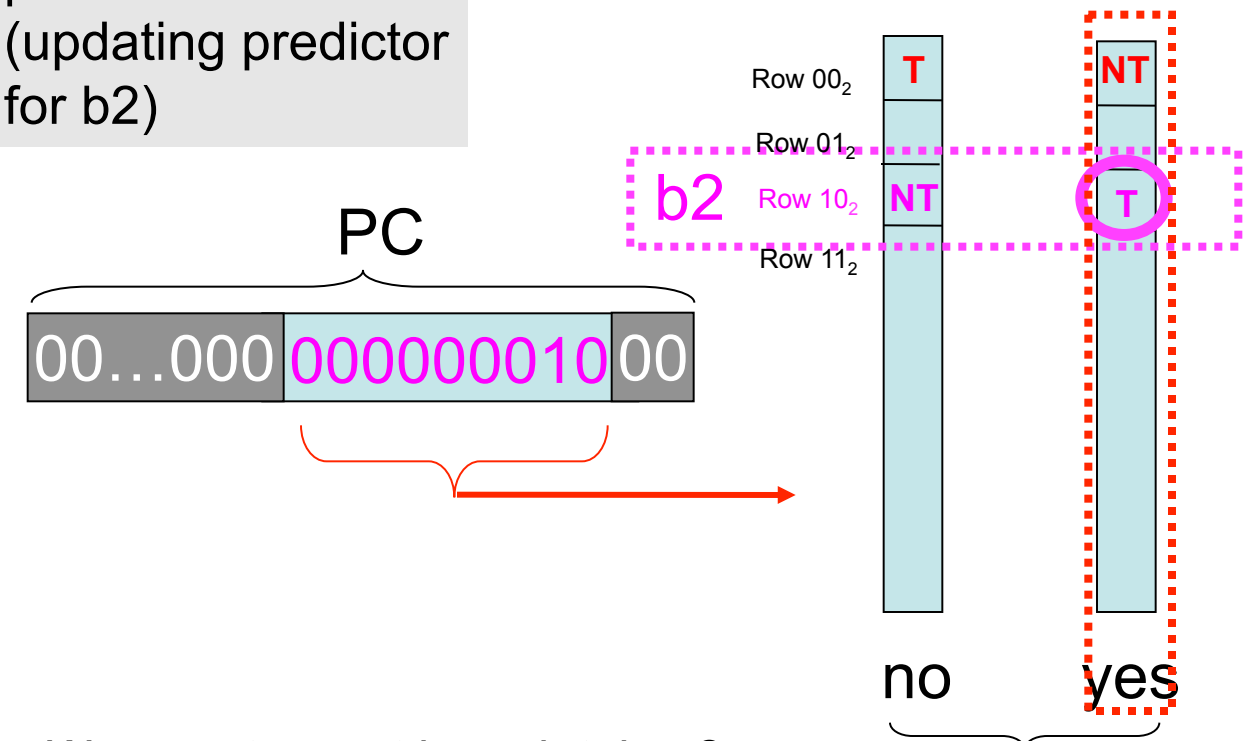

```
        if (d!=0) goto L1;//b1
            d=1;
L1:     if (d!=1) goto L2;//b2
        ...
L2:
        ...
```

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)




```
        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

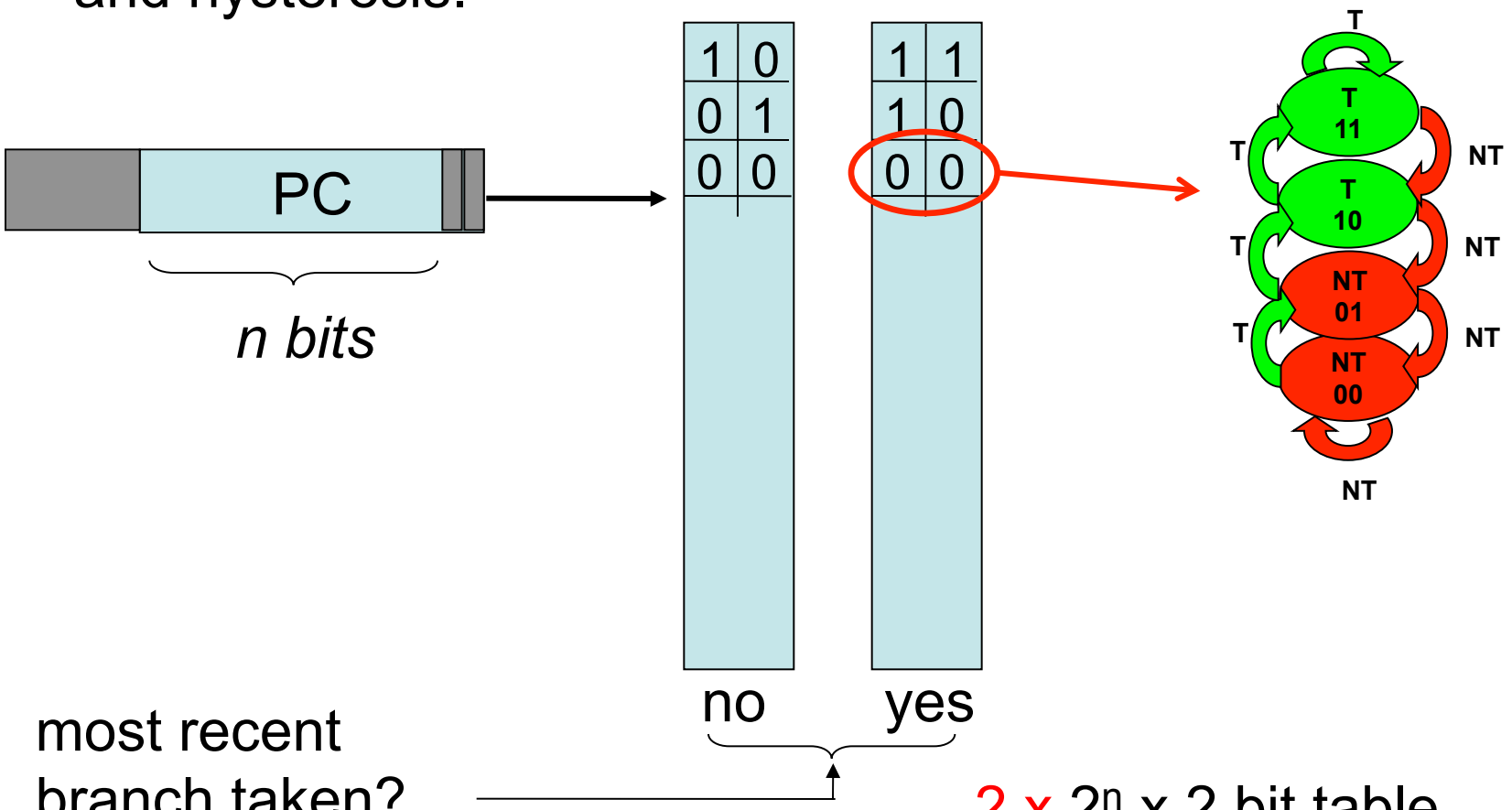
d= ?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	N/N	T	T/N	N/N	T	N/T
0	T/N	N	T/N		N	
2		T			T	
0		N			N	

down: successive passes through the example code

6 out of 8 = 75% correct predictions

2-bit Predictor with 1-bit of Correlation

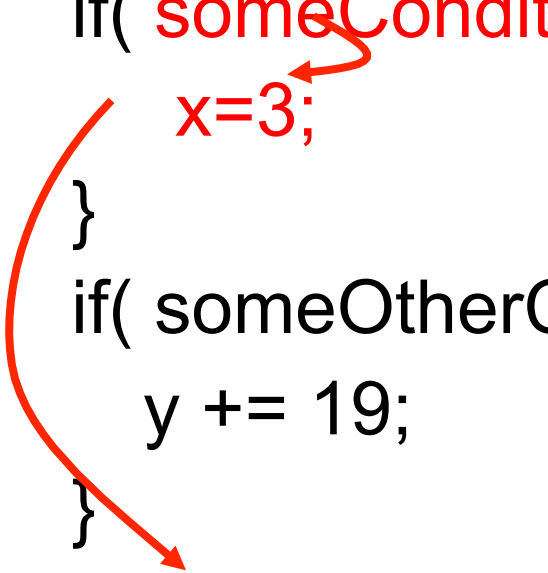
Useful to build a predictor that uses both correlation and hysteresis.



2 x 2^n x 2 bit table

Affector Branches, reconsidered…

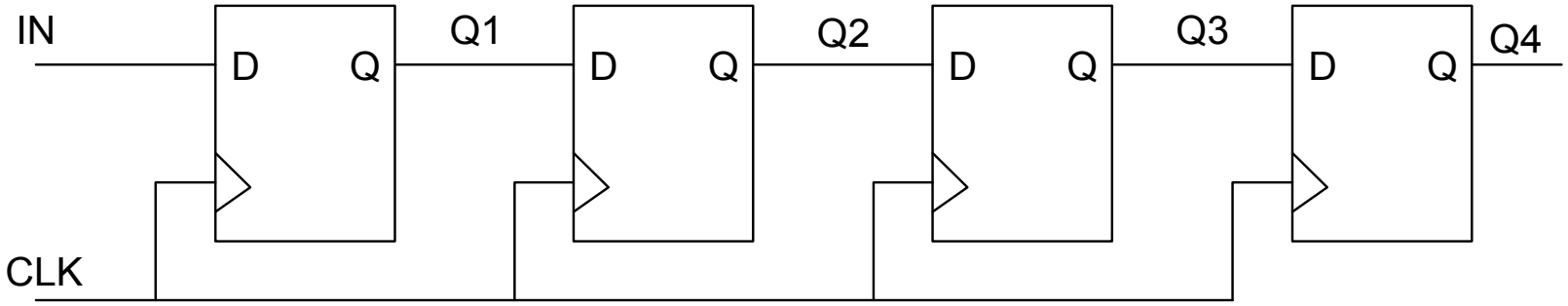
```
x=0;
if( someCondition ) { /* branch A */
    x=3;
}
if( someOtherCondition ) { /* branch B */
    y += 19;
}
if( x <= 0 ) { /* branch C */
    doSomething();
}
```



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



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

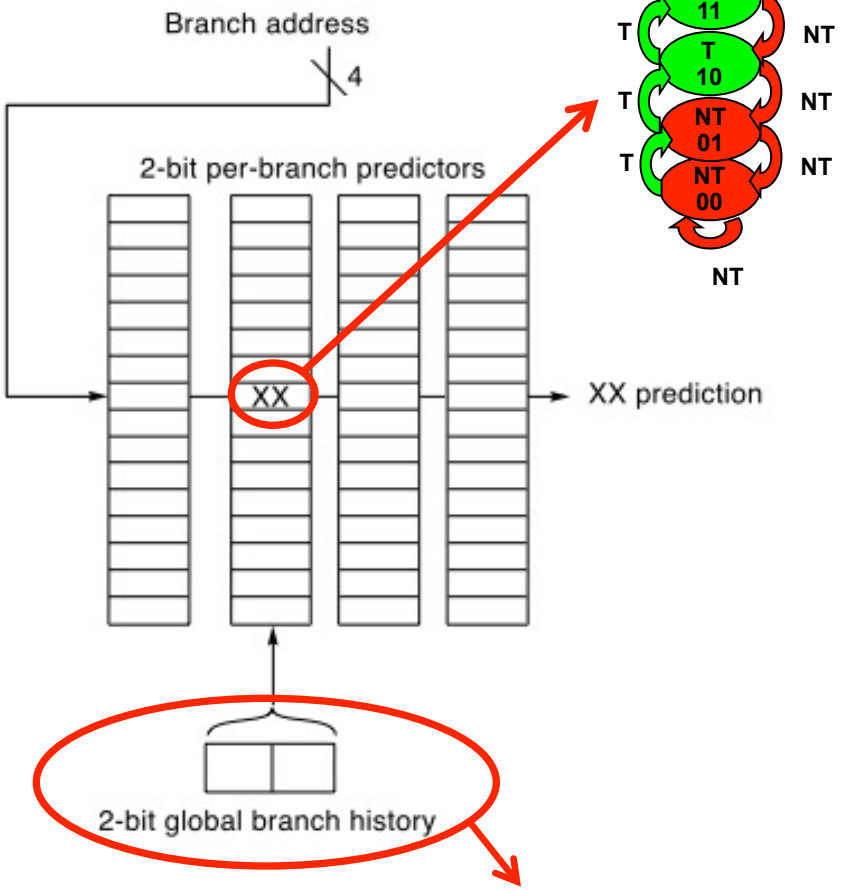
	In	Q1	Q2	Q3	Q4
cycle 0	1	0	0	0	0
cycle 1	0	1	0	0	0
cycle 2	1	0	1	0	0
cycle 3	1	1	0	1	0
cycle 4	0	1	1	0	1
cycle 5	0	0	1	1	0

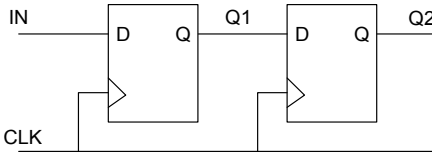
Increase “History Length”
(e.g., correlate against last 2 branches)

Real code example from SPEC89 benchmark “eqnott”:

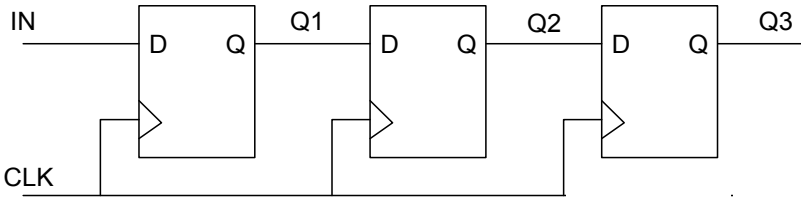
```
if( aa==2 )
    aa=0;
if( bb==2 )
        bb=0;
if( aa!=bb ) {
```

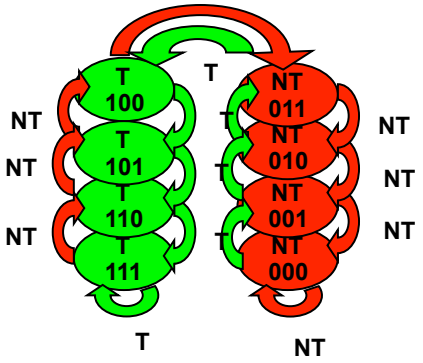
goal: want better prediction for this branch





Track outcome of 3-branches

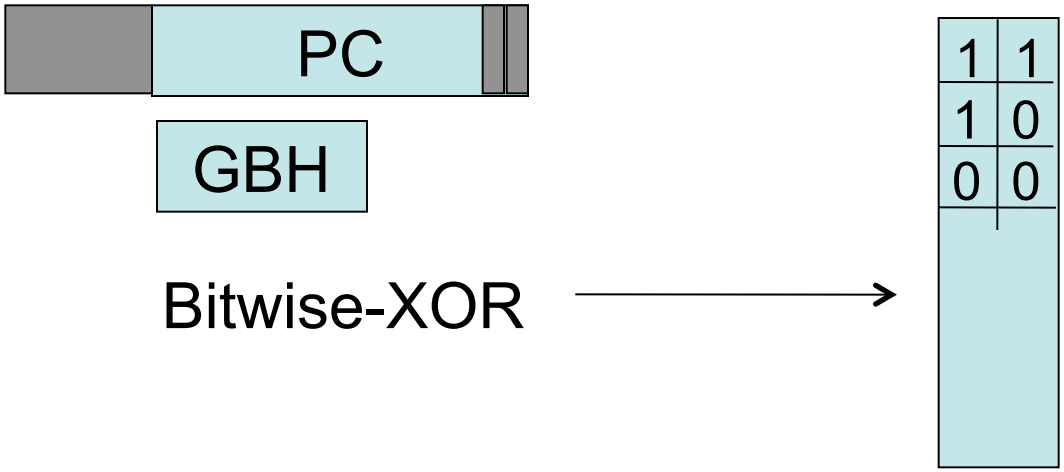




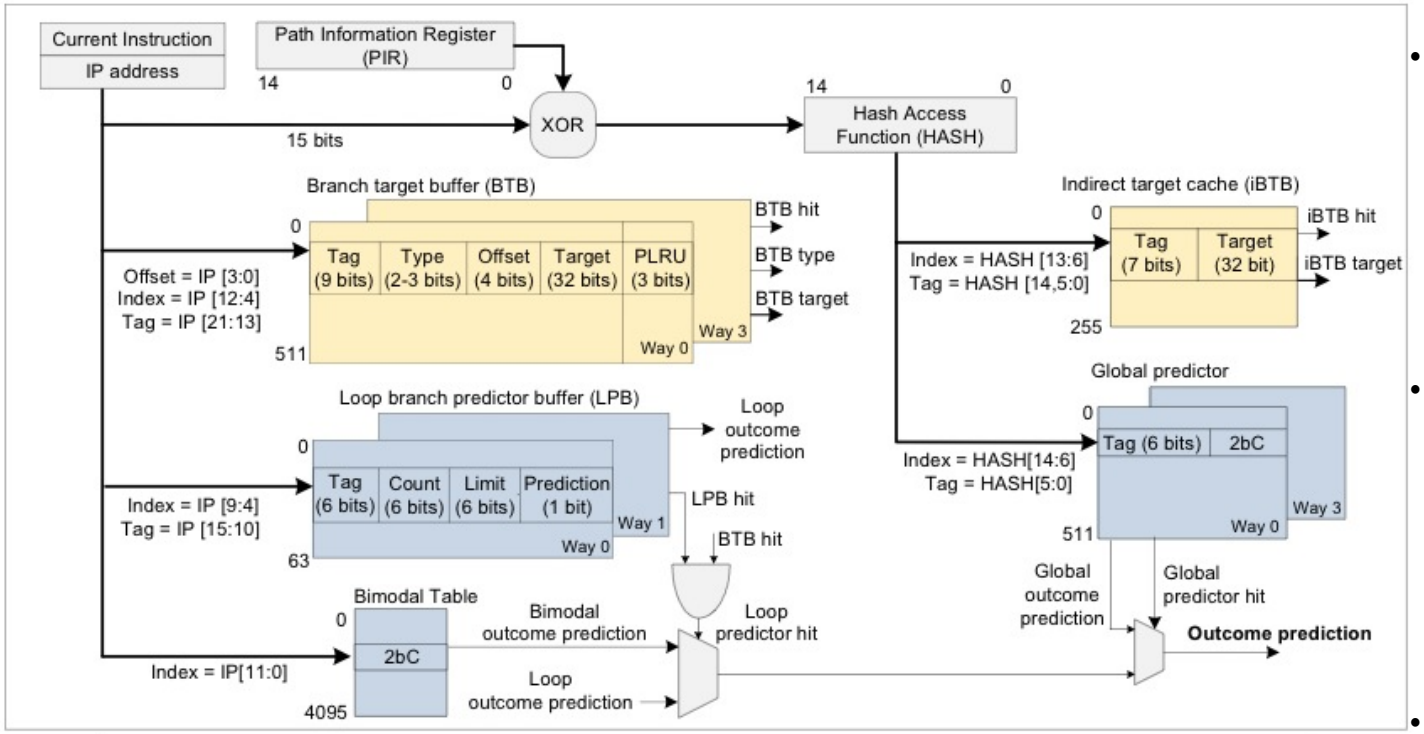
- Which is “correct” one to use?

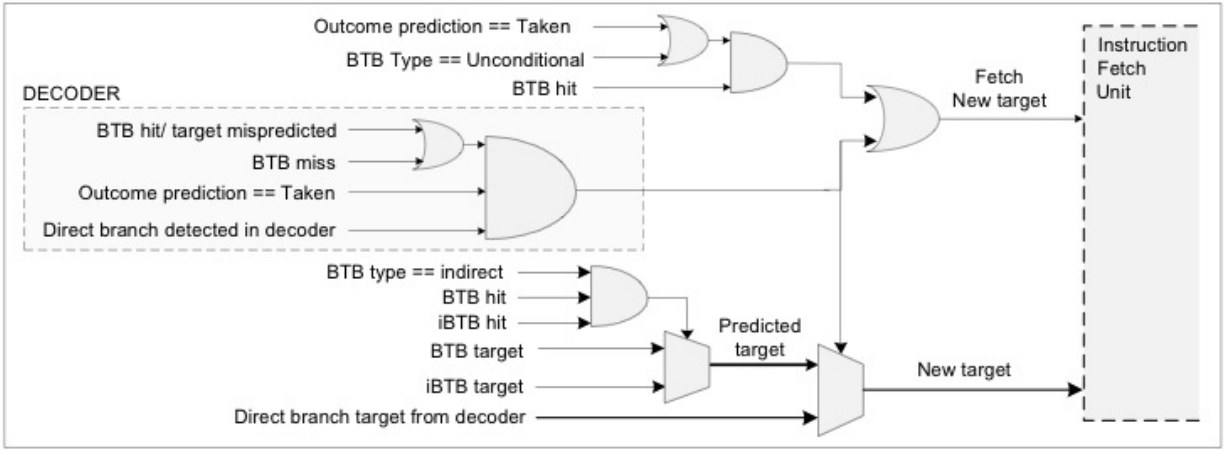
Better Use of Silicon Area

- 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.

Vladimir Uzelac, Aleksandar Milenkovic: *Experiment flows and microbenchmarks for reverse engineering of* branch predictor structures. ISPASS 2009: 207-217

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.