

# EE 660: Computer Architecture

## Branch Prediction

Yao Zheng

Department of Electrical Engineering

University of Hawai'i at Mānoa



UNIVERSITY  
*of* HAWAI'I®  
MĀNOA

Based on the slides of Prof. David Wentzlauff

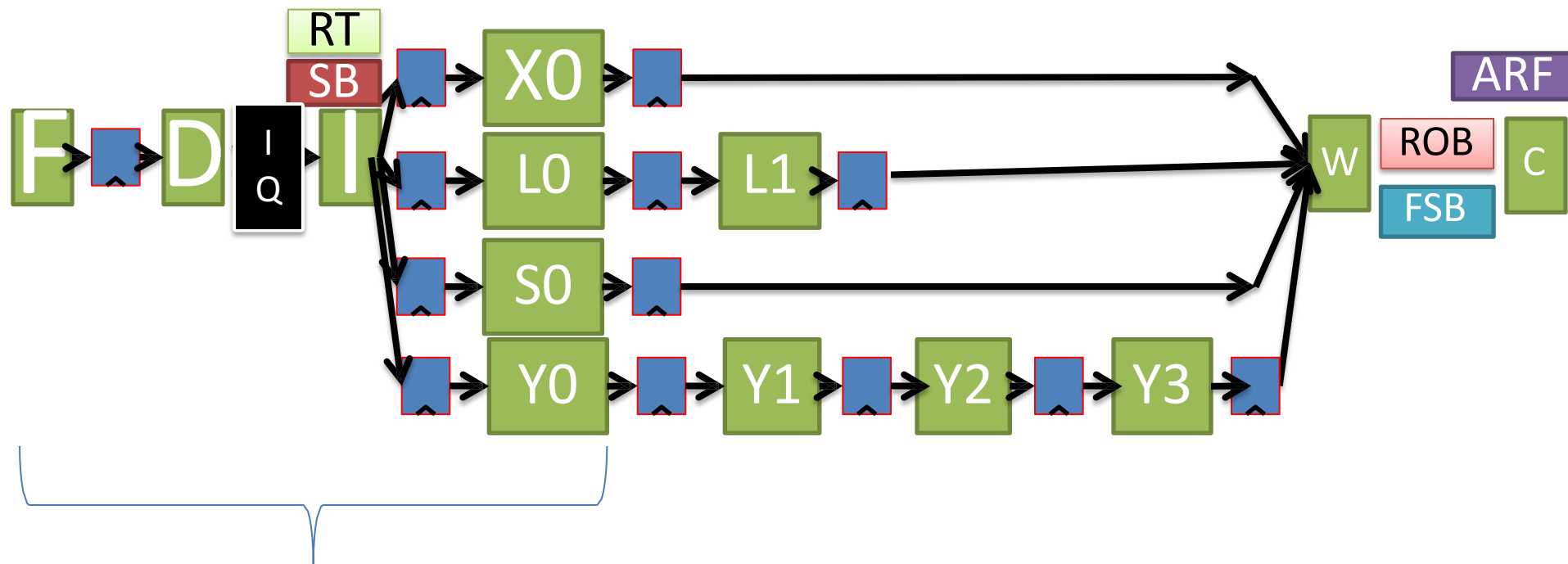
# Agenda

- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

# Agenda

- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

# Longer Frontends Means More Control Flow Penalty



Penalty includes instructions in IQ

# Longer Pipeline Frontends Amplify Branch Cost

<b>Basic Pentium III Processor Misprediction Pipeline</b>									
1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

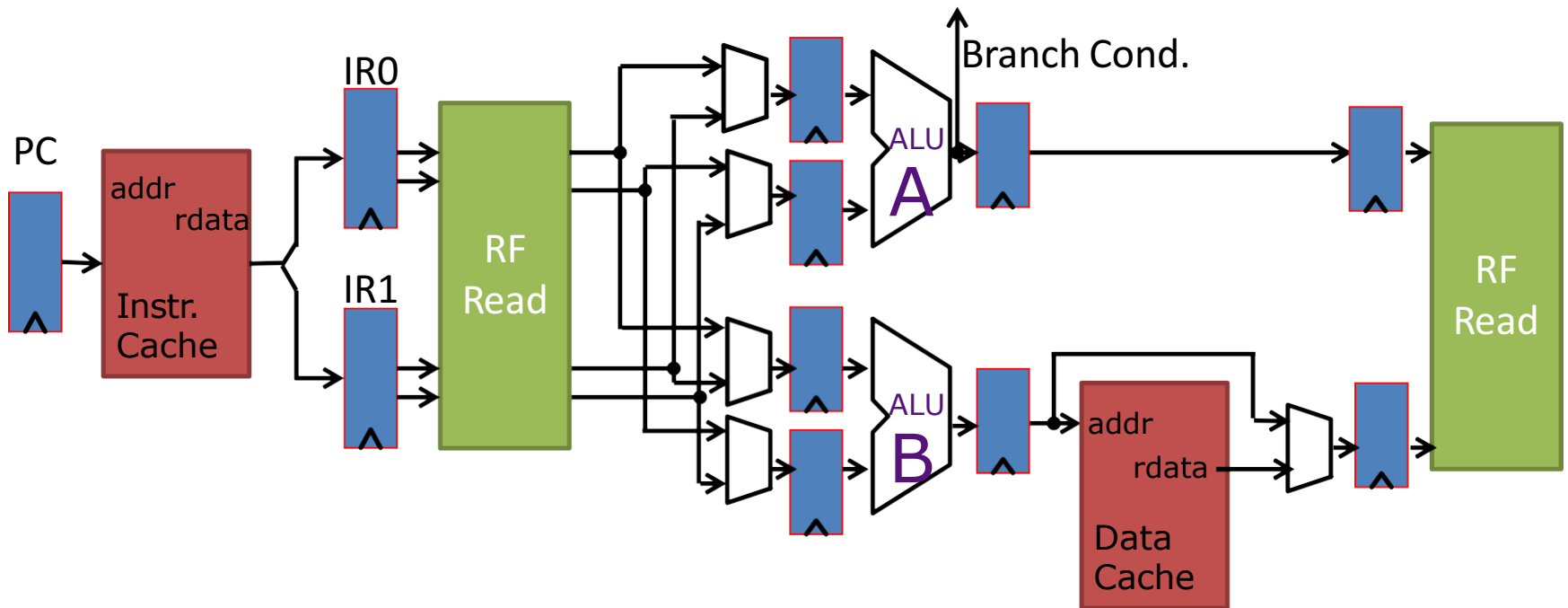
  

<b>Basic Pentium 4 Processor Misprediction Pipeline</b>																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC	Nxt IP	TC	Fetch	Drive	Alloc	Rename	Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive	

Pentium 3: 10 cycle branch penalty

Pentium 4: 20 cycle branch penalty

# Dual Issue and Branch Cost



# Superscalars Multiply Branch Cost

BEQZ	F	D	I	A0	A1	W				
OpA	F	D	I	B0	-	-				
OpB		F	D	I	-	-	-			
OpC		F	D	I	-	-	-			
OpD			F	D	-	-	-	-		
OpE			F	D	-	-	-	-		
OpF				F	-	-	-	-	-	
OpG				F	-	-	-	-	-	
OpH					F	D	I	A0	A1	W
OpI					F	D	I	B0	B1	W

Dual-issue Processor has twice the mispredict penalty

How much work is lost if pipeline doesn't follow correct instruction flow?

~pipeline width x branch penalty

# Agenda

- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

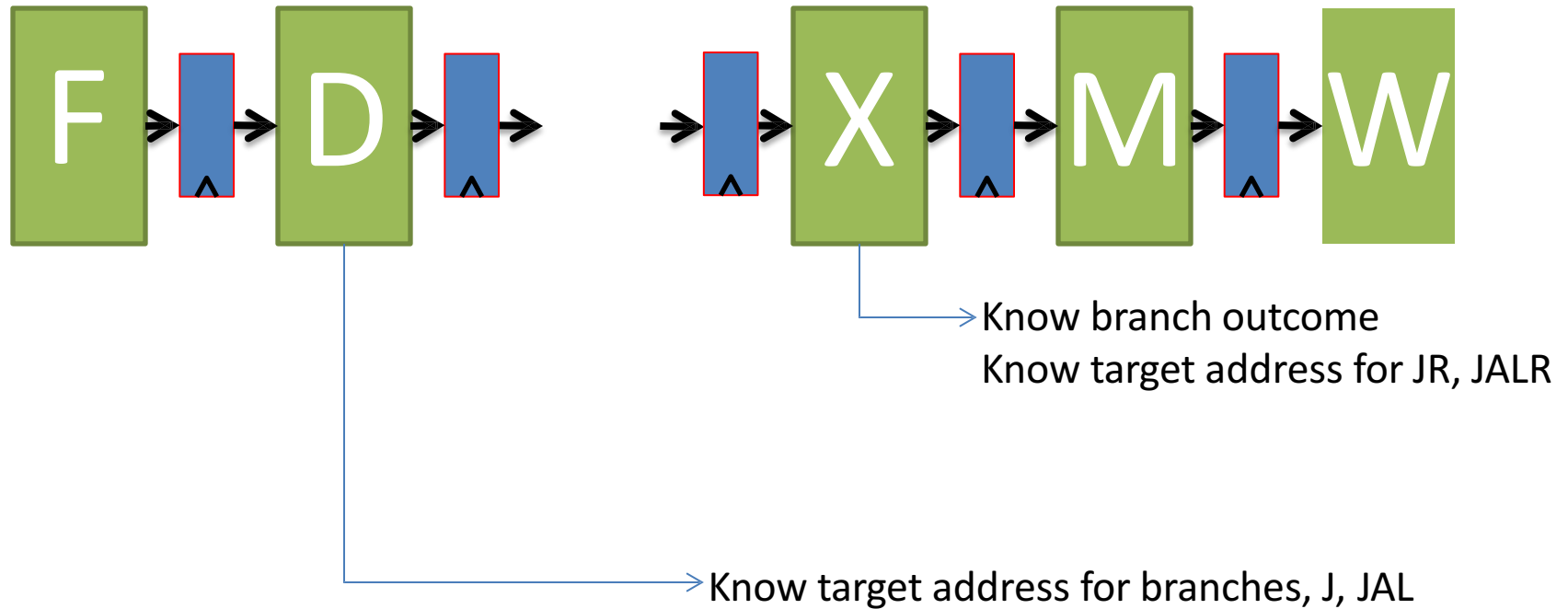
# Branch Prediction

- Essential in modern processors to mitigate branch delay latencies

## Two types of Prediction

1. Predict Branch Outcome
2. Predict Branch/Jump Address

# Where is the Branch Information Known?



# Agenda

- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

# Branch Delay Slots

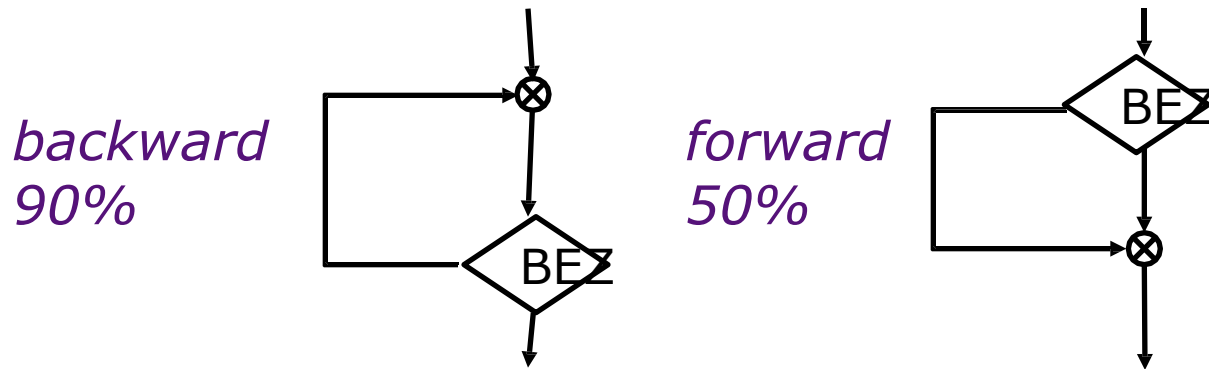
(expose control hazard to software)

- Change the ISA semantics so that the instruction that follows a jump or branch is always executed
  - gives compiler the flexibility to put in a useful instruction where normally a pipeline bubble would have resulted.

I <sub>1</sub>	096	ADD	
I <sub>2</sub>	100	BEQZ r1 +200	
I <sub>3</sub>	104	ADD	← <i>Delay slot instructions executed</i>
I <sub>4</sub>	108	ADD	← <i>regardless of branch outcome</i>
I <sub>5</sub>	304	ADD	

# Static Branch Prediction

Overall probability a branch is taken is  $\sim 60-70\%$  but:



# Static Software Branch Prediction

- Extend ISA to enable compiler to tell microarchitecture if branch is likely to be taken or not (Can be up to 80% accurate)

BR.T F D X M W

OpA F - - - -

Targ F D X M W

BR.N F D X M W

T

OpB F D X M W

OpC F D X M W

What if hint is wrong?

BR.T F D X M W

OpA F - - - -

Targ F - - - -

OpA F D X M W

# Static Hardware Branch Prediction

1. Always Predict Not-Taken
  - What we have been assuming
  - Simple to implement
  - Know fall-through PC in Fetch
  - Poor Accuracy, especially on backward branches
2. Always Predict Taken
  - Difficult to implement because don't know target until Decode
  - Poor accuracy on if-then-else
3. Backward Branch Taken, Forward Branch Not Taken
  - Better Accuracy
  - Difficult to implement because don't know target until Decode

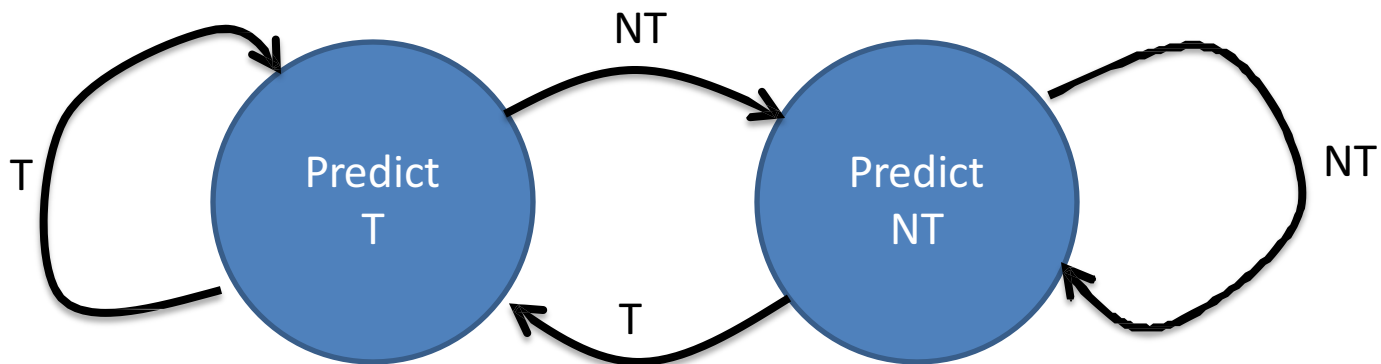
# Agenda

- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

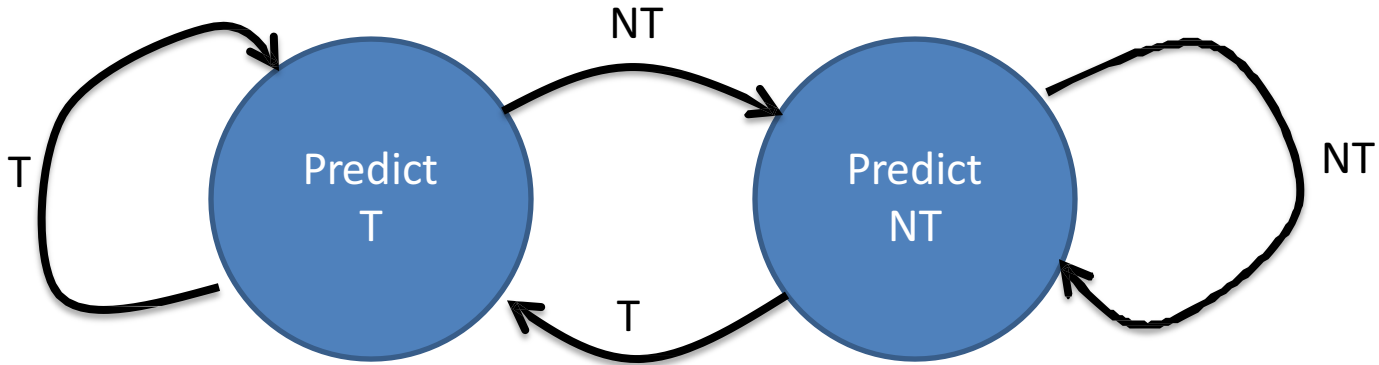
# Dynamic Hardware Branch Prediction: Exploiting Temporal Correlation

- Exploit structure in program: The way a branch resolves may be a good indicator of the way it will resolve the next time it executes (Temporal Correlation)

## 1-bit Saturating Counter



# 1-bit Saturating Counter



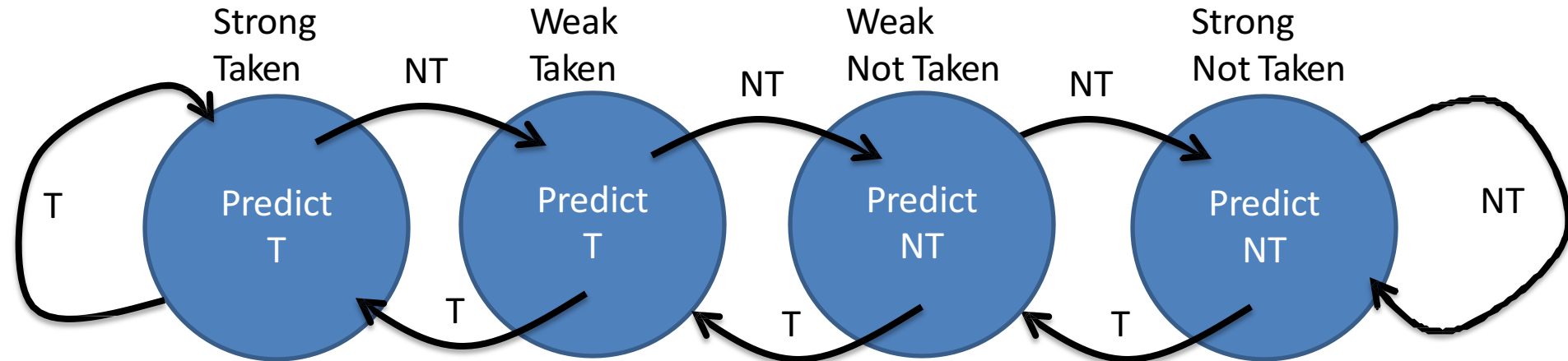
Iteration	Prediction	Actual	Mispredict?
1	NT	T	Y
2	T	T	
3	T	T	
4	T	NT	Y
...			
1	NT	T	Y
2	T	T	
3	T	T	
4	T	NT	Y

For Backward branch in loop

- Assume 4 Iterations
- Assume is executed multiple times

Always 2 Mispredicts

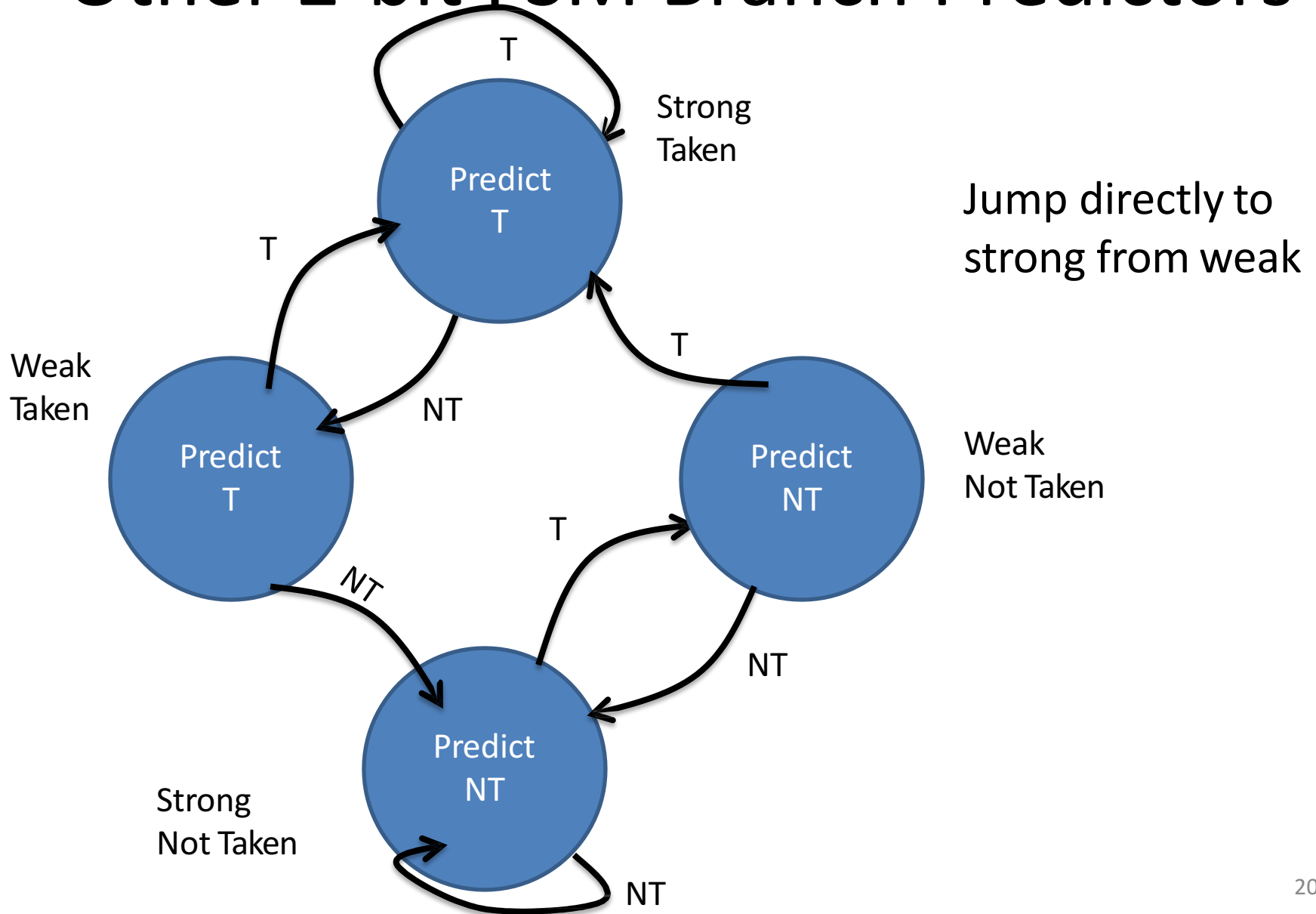
# 2-bit Saturating Counter



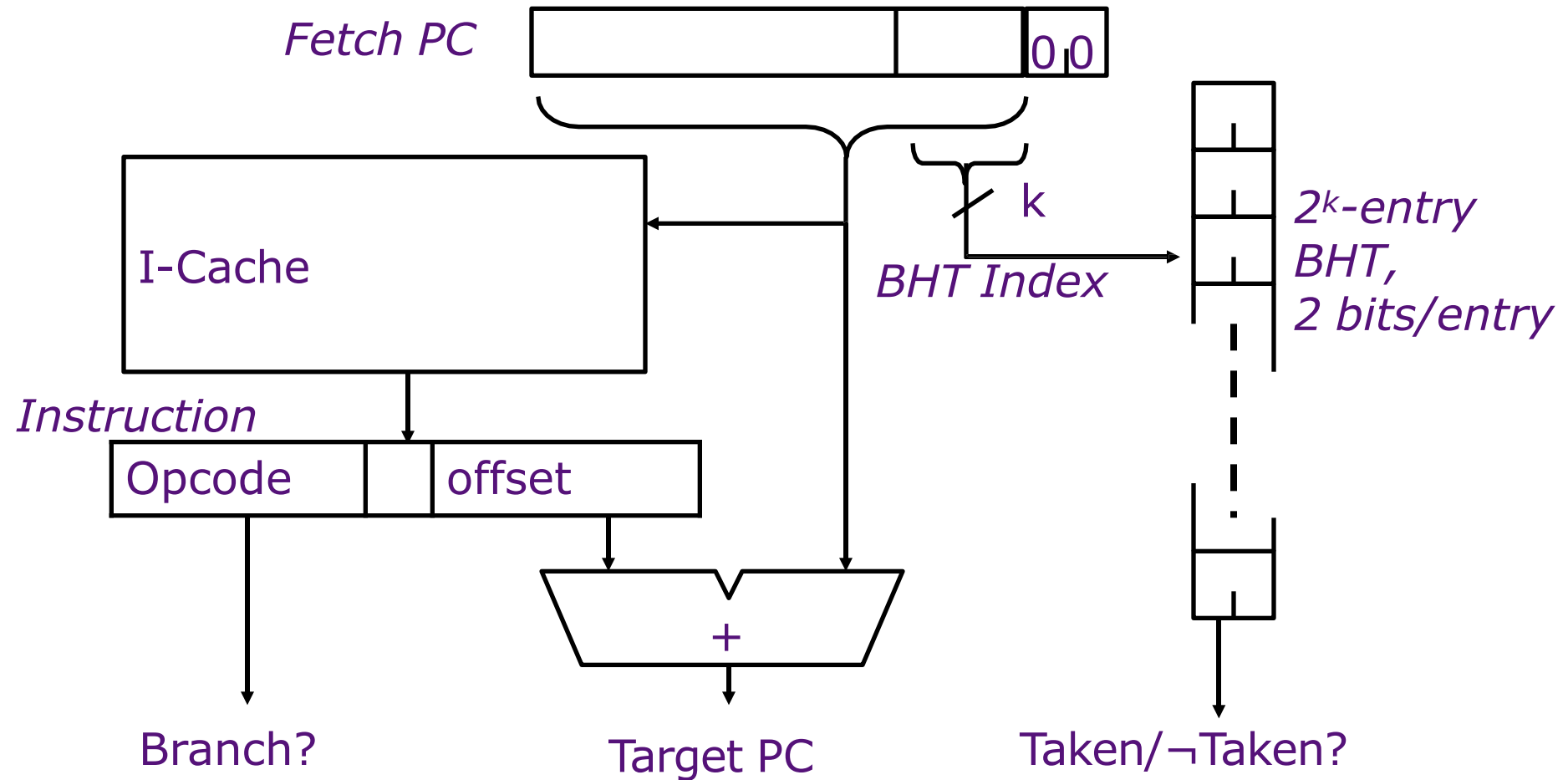
Iteration	Prediction	Actual	Mispredict?	State
1	NT	T	Y	Strong NT
2	NT	T	Y	Weak NT
3	T	T		Weak T
4	T	NT	Y	Strong T
...				
1	T	T		Weak T
2	T	T		Strong T
3	T	T		Strong T
4	T	NT	Y	Strong T

Only 1  
Mispredict

# Other 2-bit FSM Branch Predictors



# Branch History Table (BHT)



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

# Exploiting Spatial Correlation

*Yeh and Patt, 1992*

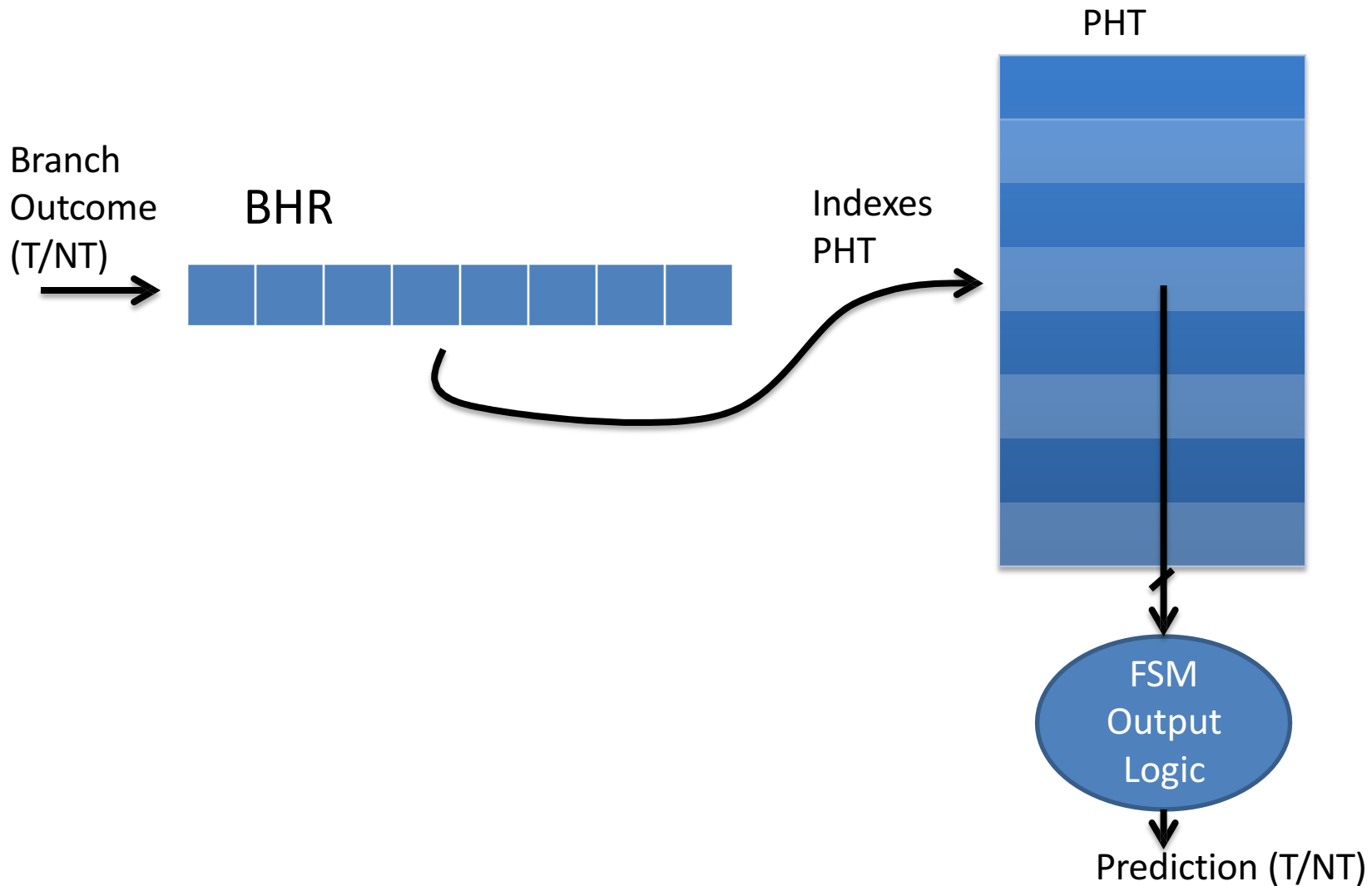
```
if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;
```

If first condition false, second condition also false

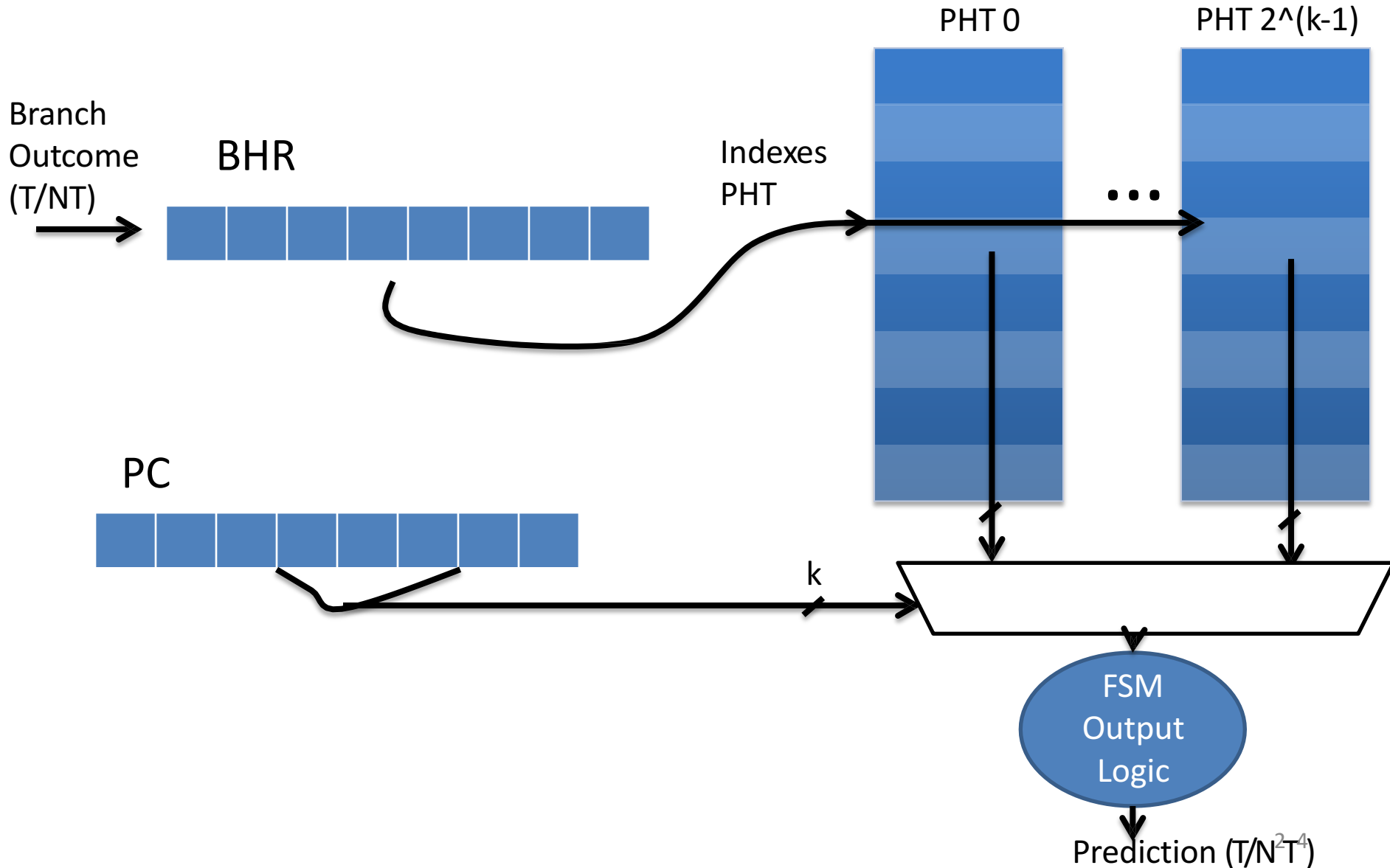
*Branch History Register, BHR*, records the direction of the last N branches executed by the processor (Shift Register)



# Pattern History Table (PHT)



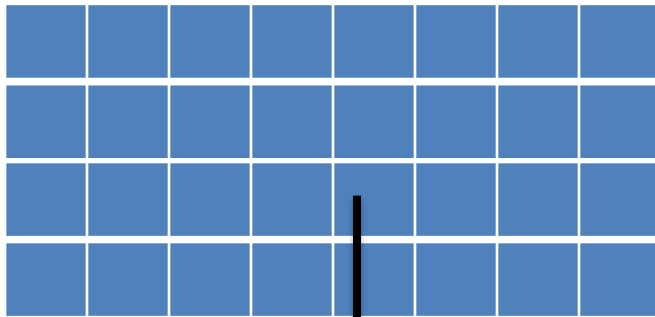
# Two-Level Branch Predictor



# Generalized Two-Level Branch Predictor

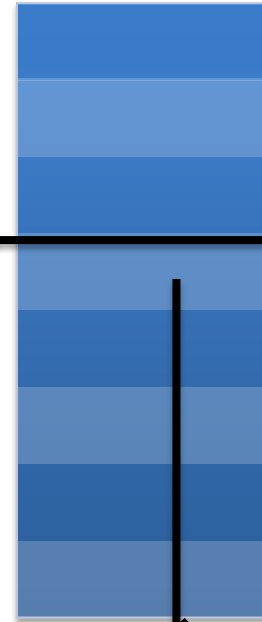
Branch Outcome (T/NT)

BHR



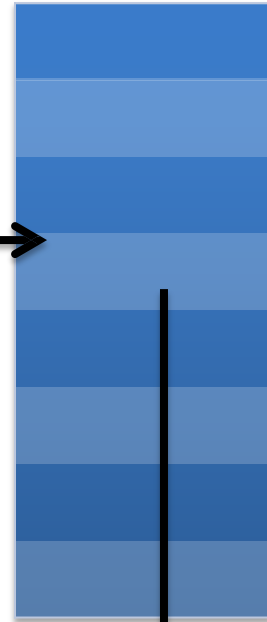
Indexes PHT

PHT 0



...

PHT  $2^{(k-1)}$



PC



k

m

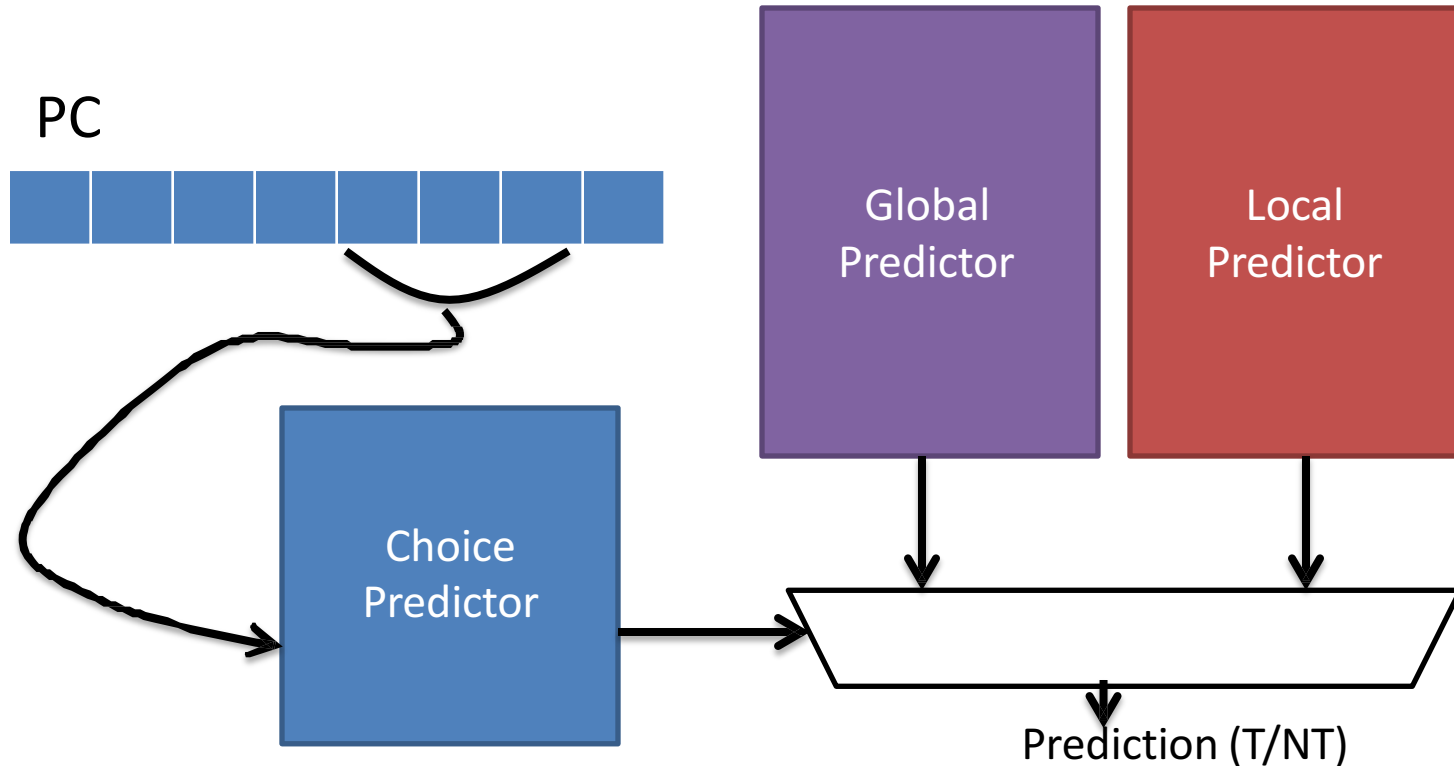


FSM Output Logic

Prediction (T/NT<sup>2-5</sup>)

For non-trivial m and k, > 97% accuracy

# Tournament Predictors (ex: Alpha 21264)

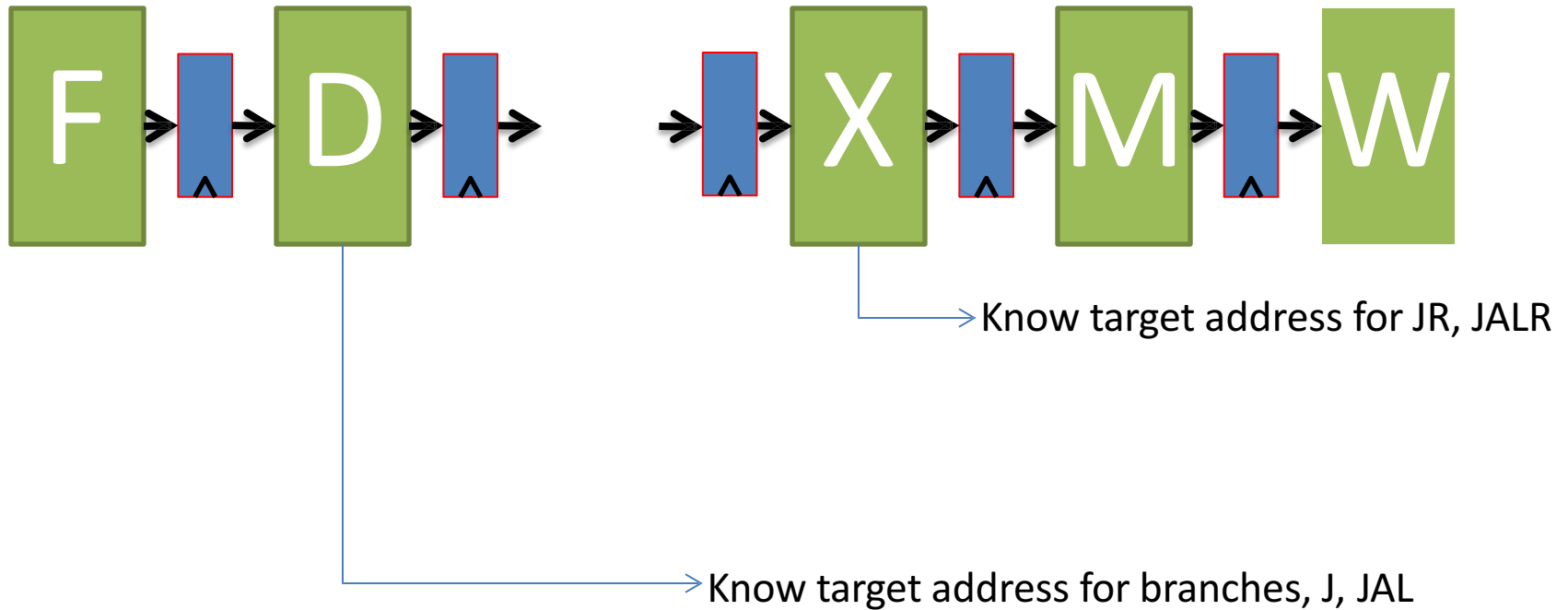


- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications

# Agenda

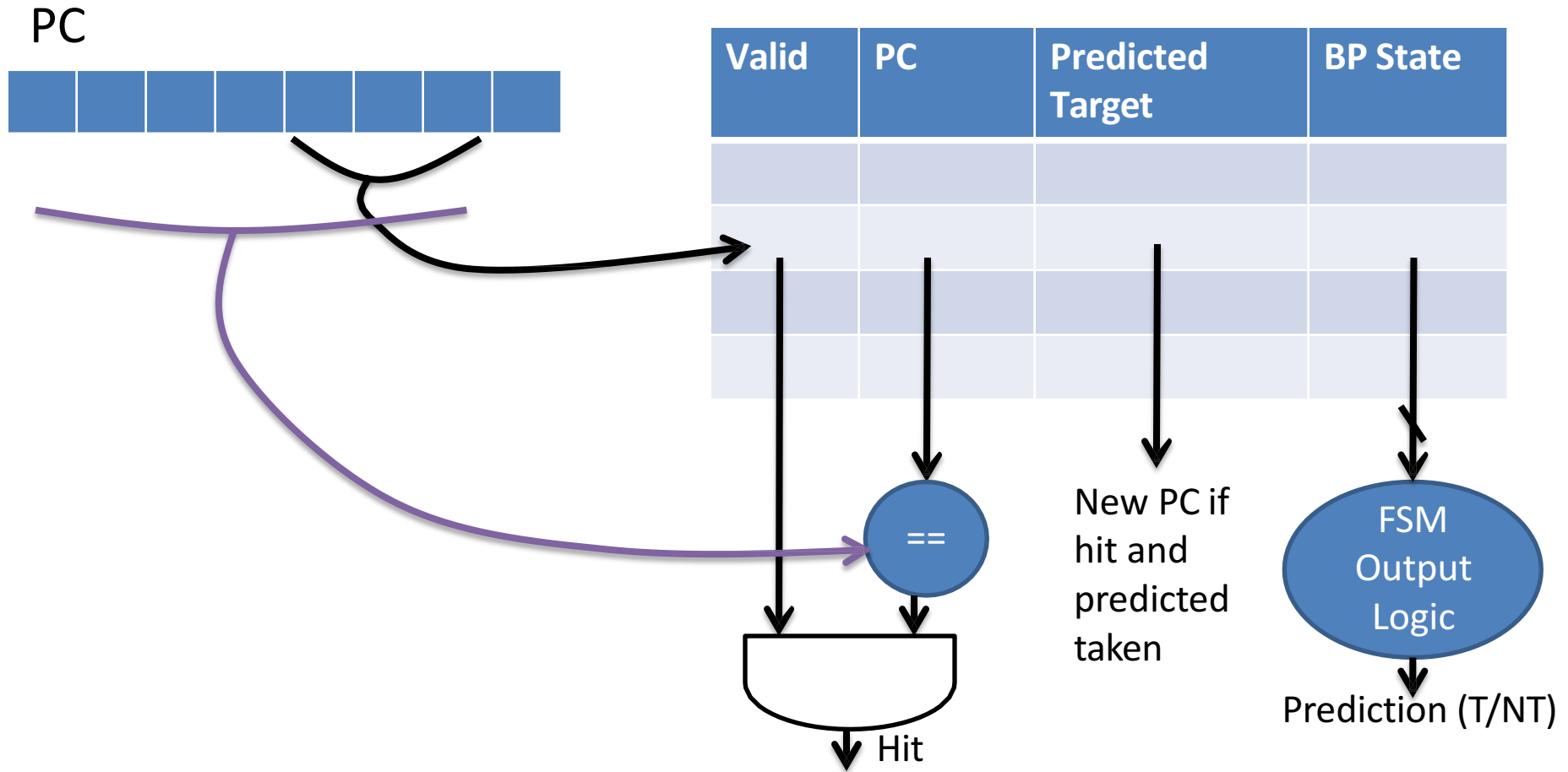
- Branch Cost Motivation
- Branch Prediction
  - Outcome
    - Static
    - Dynamic
  - Target Address

# Predicting Target Address



Even with best possible prediction of branch outcome, still have to wait for branch target address to be determined

# Branch Target Buffer (BTB)



Put BTB in Fetch Stage in parallel with PC+4 Speculation logic

# BTB is only for Control Instructions

- BTB contains useful information for branch and jump instructions only
  - Do not update it for other instructions
- For all other instructions the next PC is PC+4 !

*How to achieve this effect without decoding the instruction?*

*When do we update BTB information?*

# Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)  
BTB works well if same case used repeatedly
- Dynamic function call (jump to run-time function address)  
BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)
- Subroutine returns (jump to return address)  
BTB works well if usually return to the same place  
⇒ *Often one function called from many distinct call* \$

How well does BTB work for each of these cases?

# Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```
fa () { fb () ; }  
fb () { fc () ; }  
fc () { fd () ; }
```

*Push call address when  
function call executed*

*Pop return address  
when subroutine  
return decoded*

**&fd ()**  
**&fc ()**  
**&fb ()**

*k entries  
(typically k=8-16)*

# Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Christopher Batten (Cornell)
- MIT material derived from course 6.823
- UCB material derived from course CS252 & CS152
- Cornell material derived from course ECE 4750

Copyright © 2013 David Wentzlaff