

# CSE 502:

# Computer Architecture

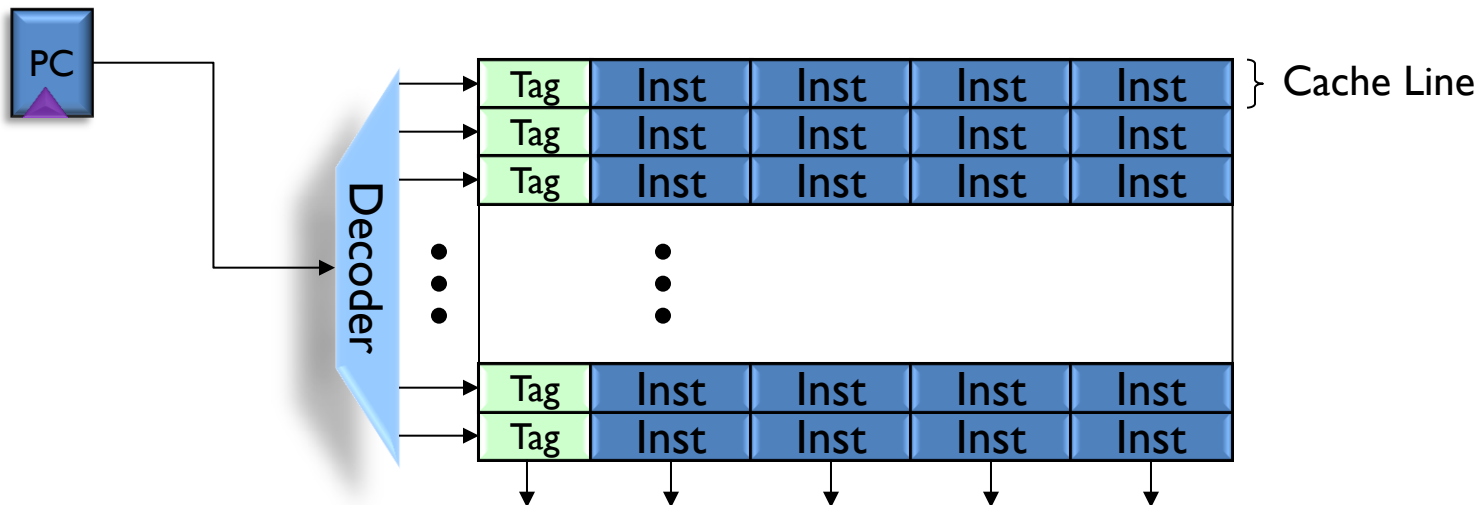
Instruction Fetch and Branch Prediction

# Fetch Rate is an ILP Upper Bound

- Instruction fetch limits performance
  - To sustain IPC of  $N$ , must sustain a fetch rate of  $N$  per cycle
    - If you consume 1500 calories per day, but burn 2000 calories per day, then you will eventually starve.
  - Need to fetch  $N$  on average, not on every cycle
- $N$ -wide superscalar *ideally* fetches  $N$  insns. per cycle
- This doesn't happen in practice due to:
  - Instruction cache organization
  - Branches
  - ... and interaction between the two

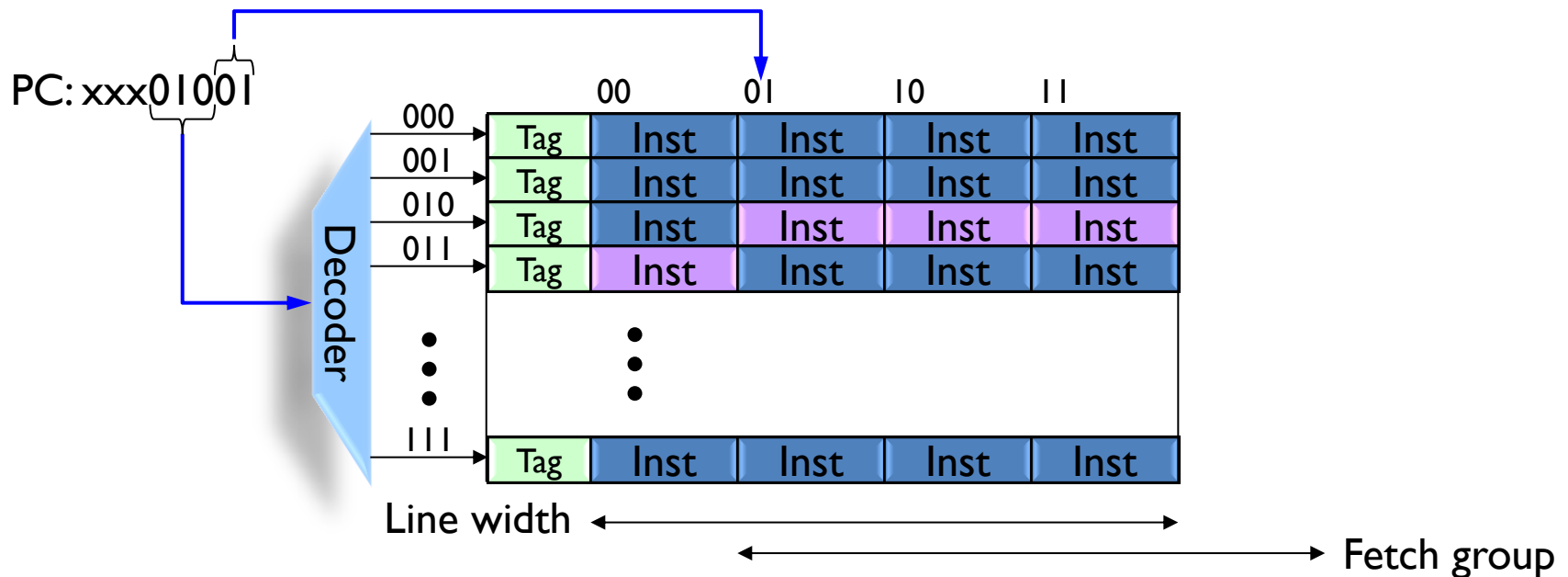
# Instruction Cache Organization

- To fetch N instructions per cycle...
  - L1-I line must be wide enough for N instructions
- PC register selects L1-I line
- A fetch group is the set of insns. starting at PC
  - For N-wide machine,  $[PC, PC+N-1]$



# Fetch Misalignment (1/2)

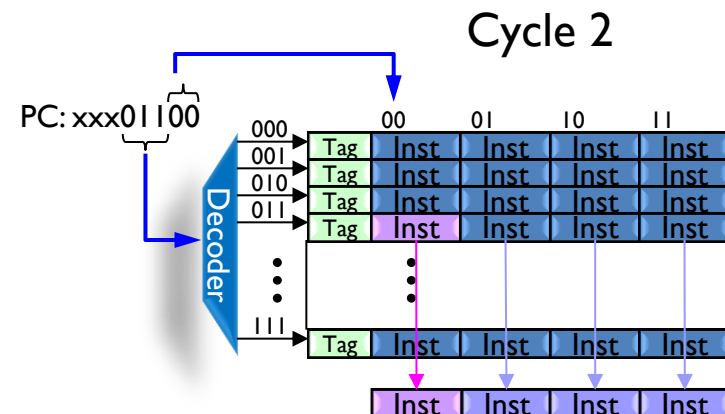
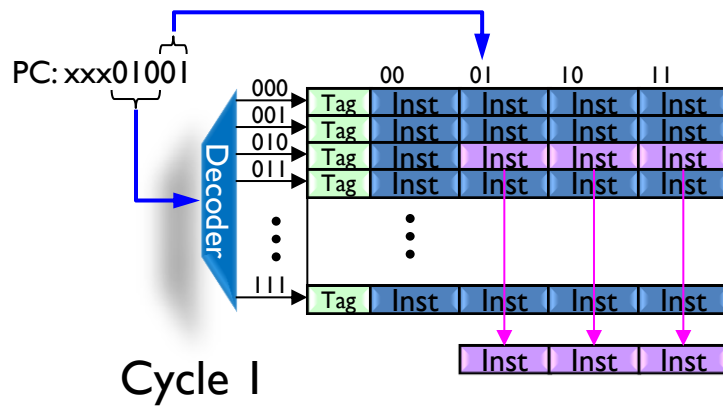
- If PC = xxx01001, N=4:
  - Ideal fetch group is xxx01001 through xxx01100 (inclusive)



Misalignment reduces fetch width

# Fetch Misalignment (2/2)

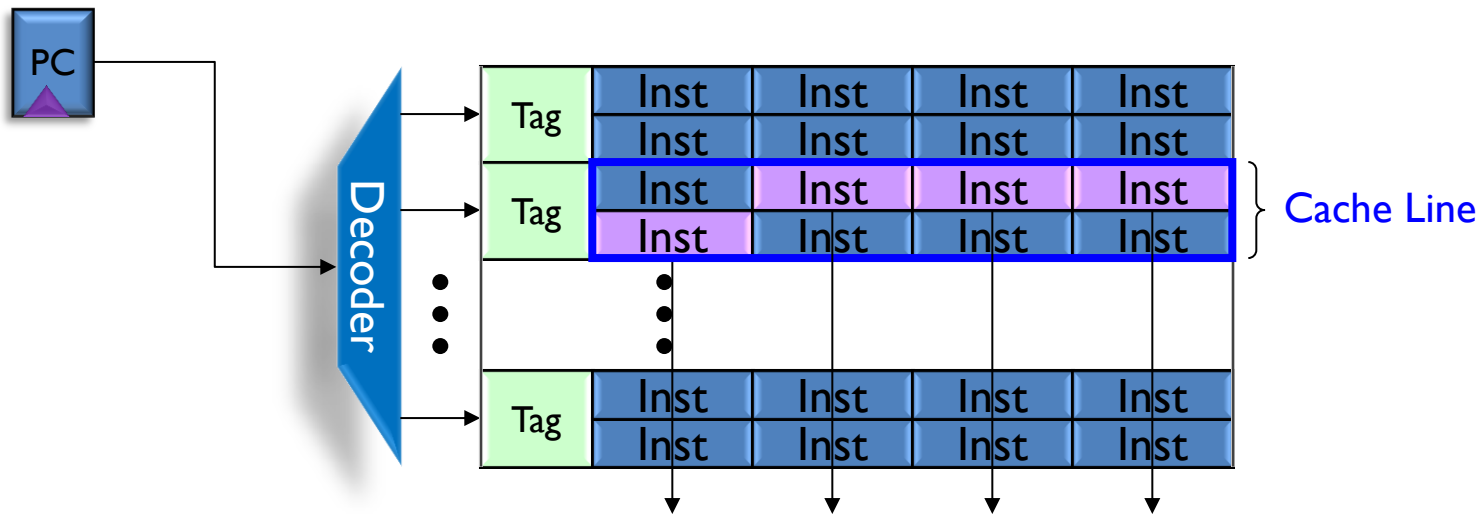
- Now takes two cycles to fetch N instructions
  - $\frac{1}{2}$  fetch bandwidth!



Might not be  $\frac{1}{2}$  by combining with the next fetch

# Reducing Fetch Fragmentation (1/2)

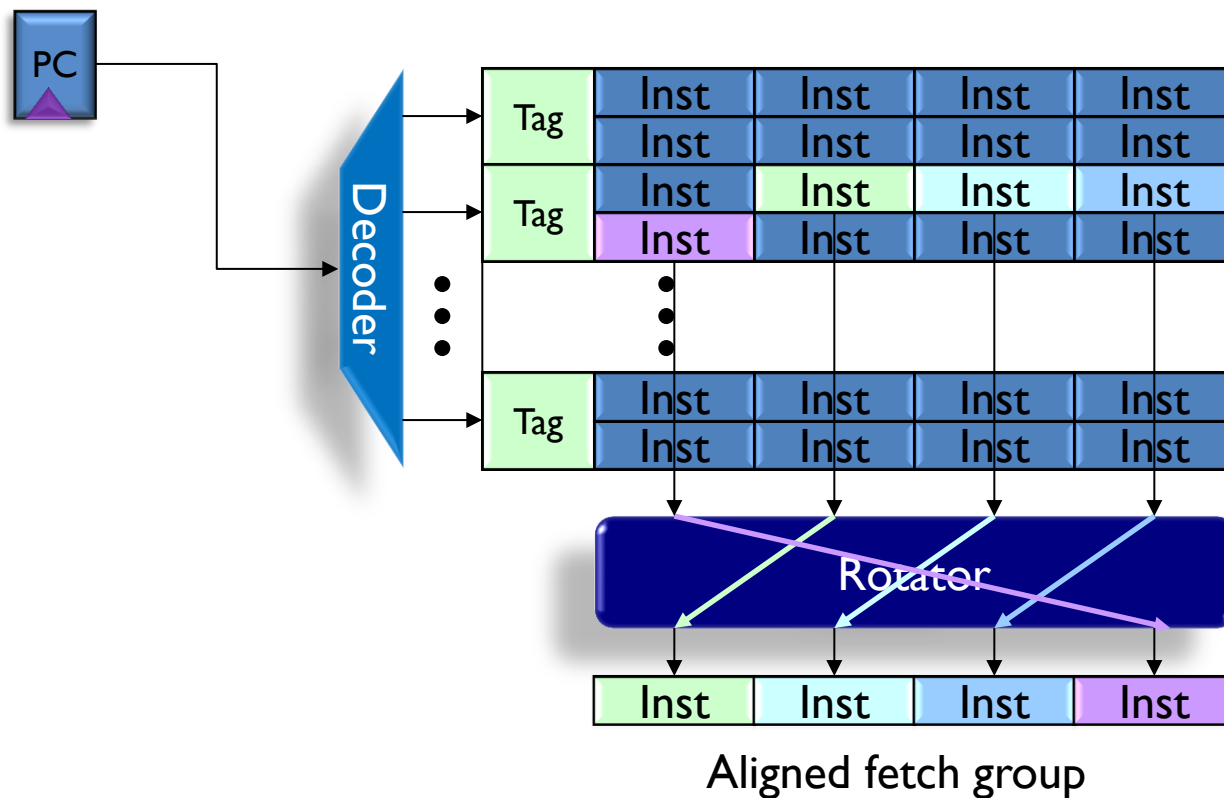
- Make  $|\text{Fetch Group}| < |\text{L1-L Line}|$



Can deliver N insns. when  $PC > N$  from end of line

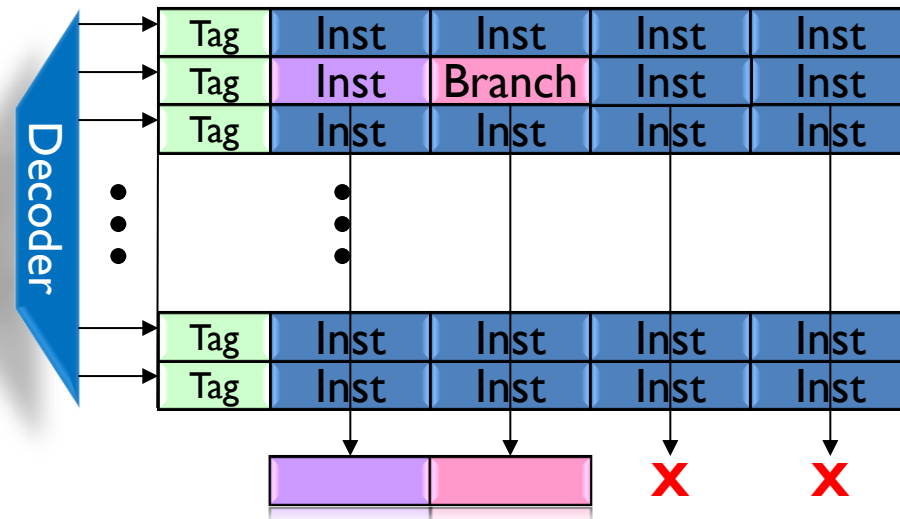
# Reducing Fetch Fragmentation (2/2)

- Needs a “rotator” to decode insns. in correct order



# Fragmentation due to Branches

- Fetch group is aligned, cache line size > fetch group
  - Taken branches still limit fetch width





# Types of Branches

- Direction:
  - Conditional vs. Unconditional
- Target:
  - PC-encoded
    - PC-relative
    - Absolute offset
  - Computed (target derived from register)

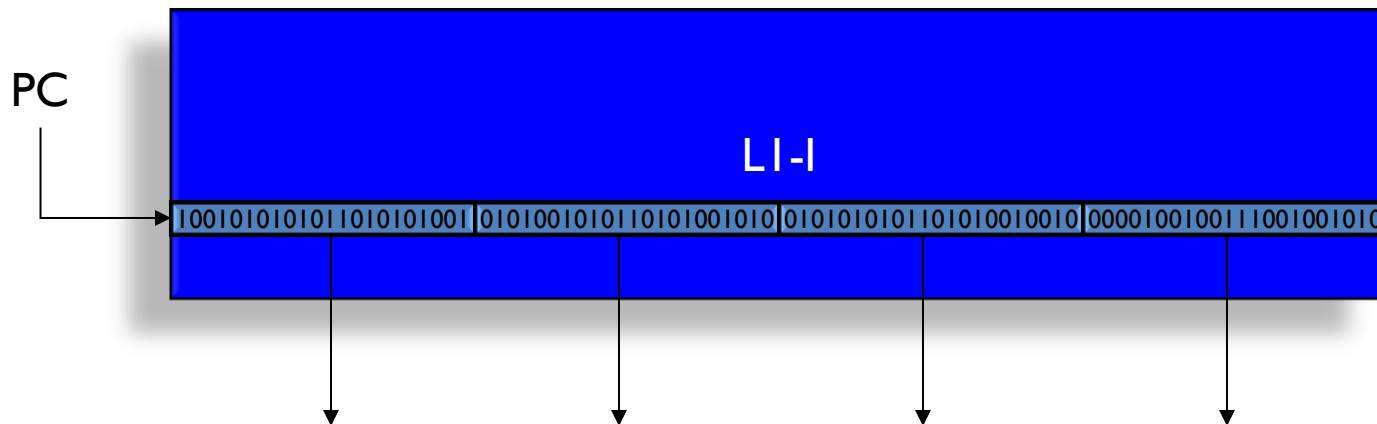
Need direction and target to find next fetch group

# Branch Prediction Overview

- Use two hardware predictors
  - Direction predictor guesses if branch is taken or not-taken
  - Target predictor guesses the destination PC
- Predictions are based on history
  - Use previous behavior as indication of future behavior
  - Use historical context to disambiguate predictions

# Where Are the Branches?

- To predict a branch, must find the branch

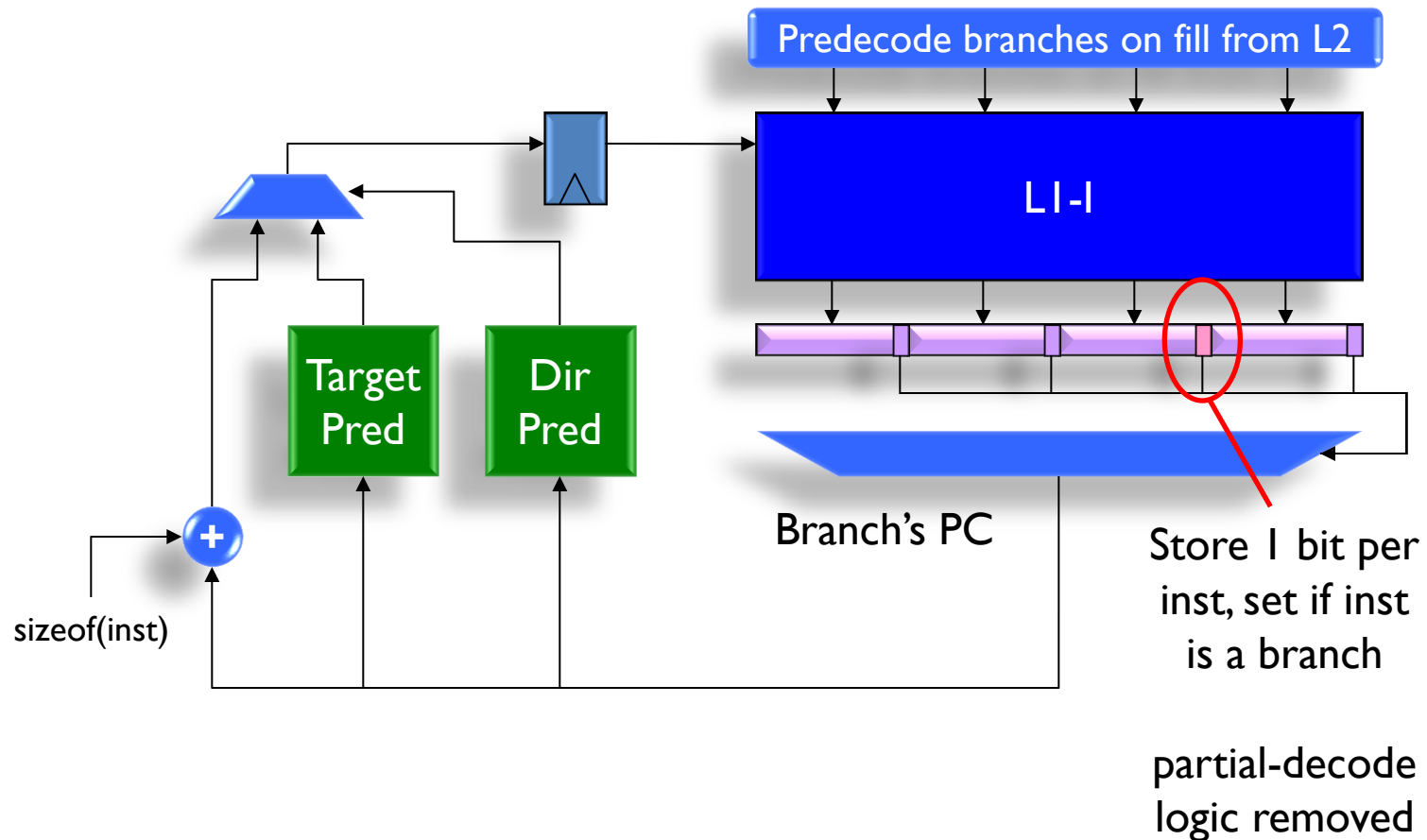


Where is the branch in the fetch group?

The diagram illustrates a Branch Target Predictor (BTP) architecture. It features a large green curved arrow representing a feedback loop. The **Branch's PC** is input to a **Fetch PC** block and a **LI-I** (Local Instruction Index) block. The **Fetch PC** block outputs the **Target PC** (PC + **sizeof(inst)**). The **LI-I** block outputs to a multi-ported structure containing **PD** (Prediction Data) blocks. The **Target PC** is used to calculate the **Target Pred** and **Dir Pred** (Direction Prediction). These predictions are then used to update the **LI-I** and the **PD** blocks. The **PD** blocks are part of a multi-ported structure that also receives the **Branch's PC** and outputs the **Target PC**.

## Huge latency (reduces clock frequency)

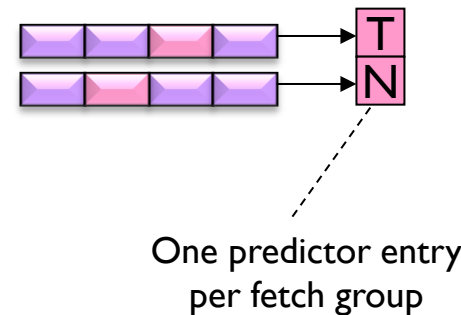
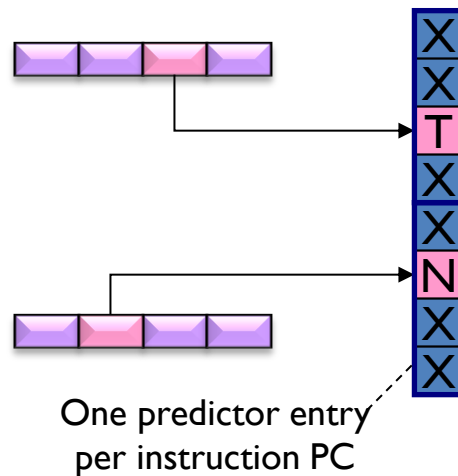
# Branch Identification



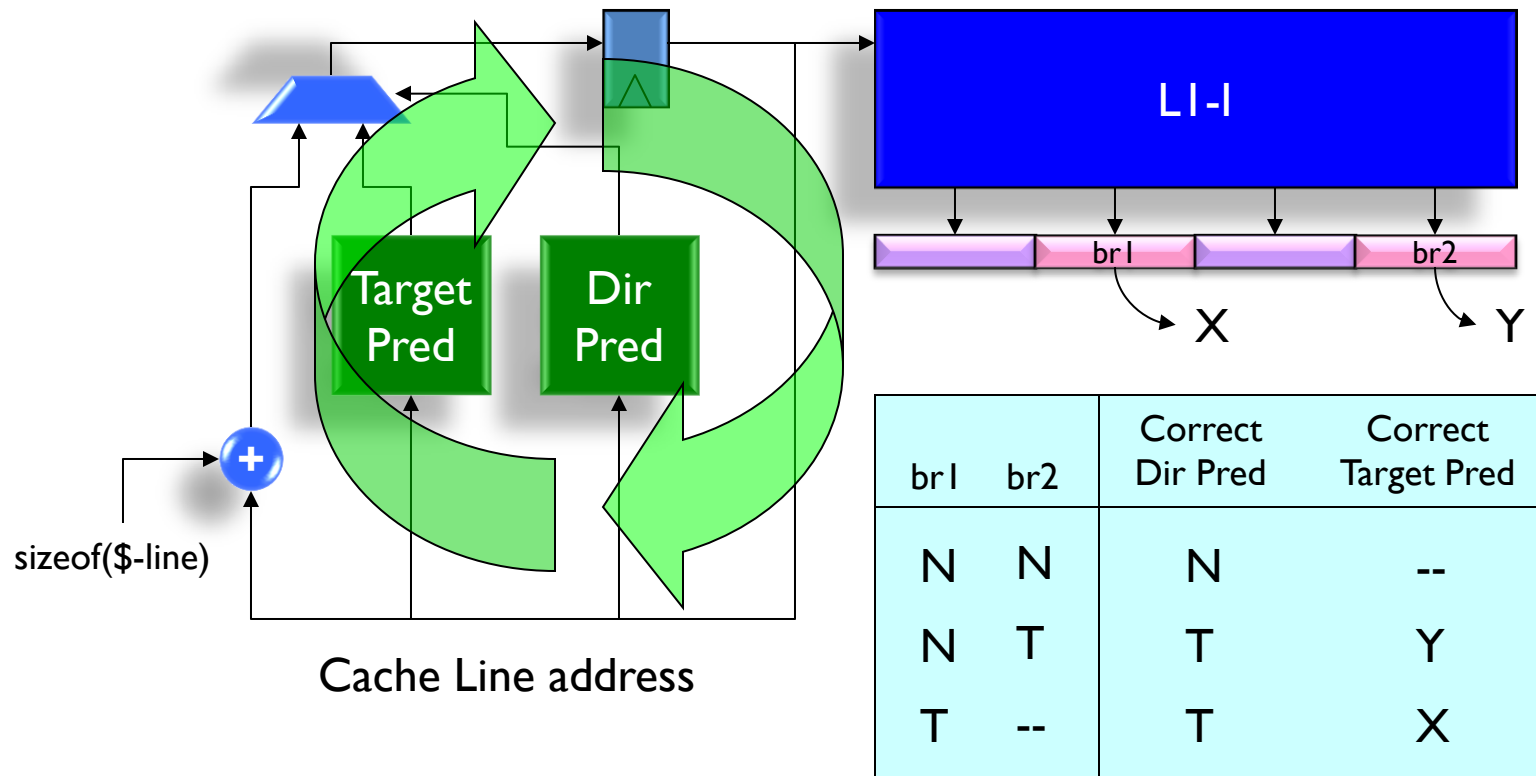
High latency (L1-I on the critical path)

# Line Granularity

- Predict fetch group without location of branches
  - With one branch in fetch group, does it matter where it is?



# Predicting by Line



This is still challenging: we may need to choose between multiple targets for the same cache line

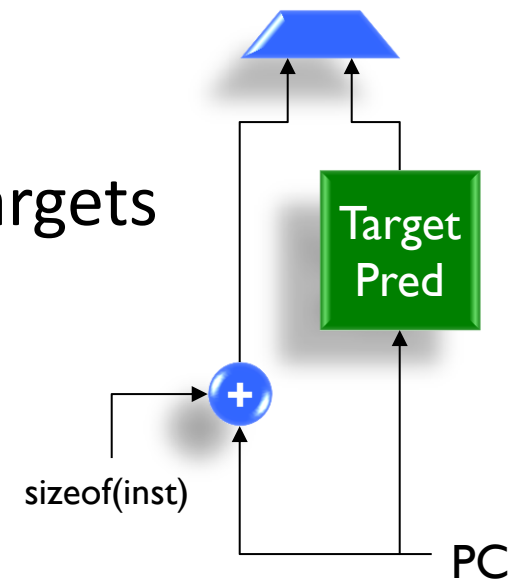
Latency determined by branch predictor

[illegible]

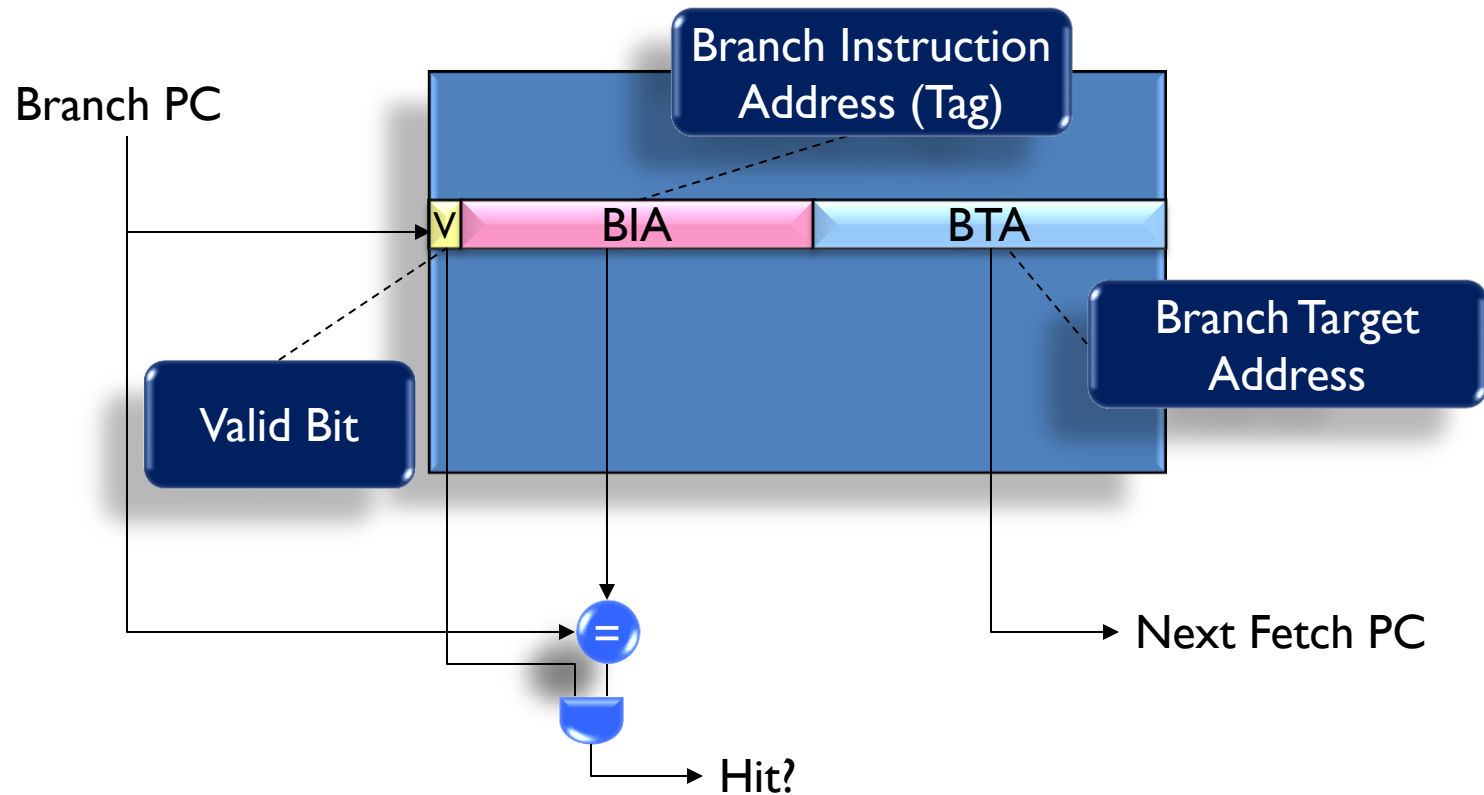


# Direction vs. Target Prediction

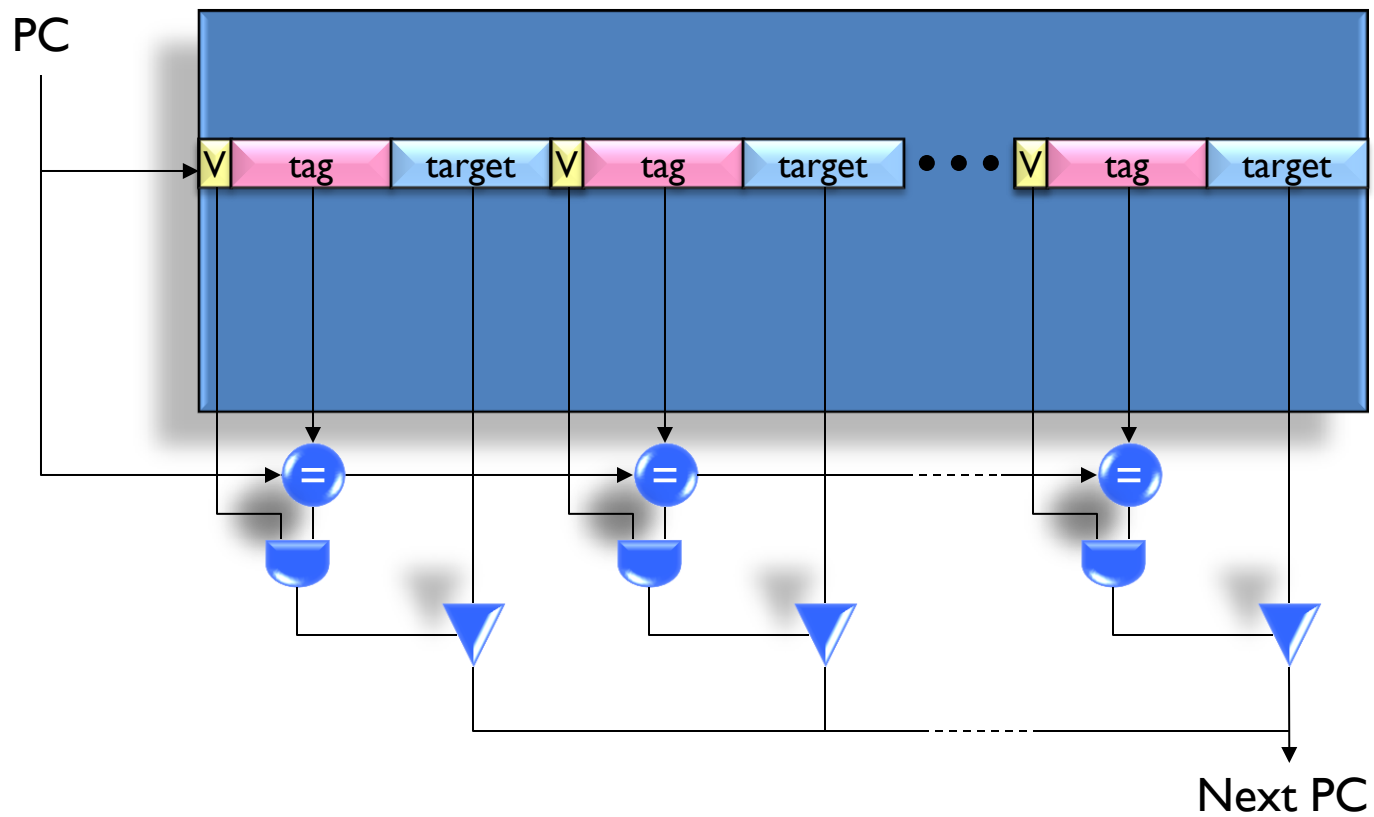
- Direction: 0 or 1
- Target: 32- or 64-bit value
- Turns out targets are generally easier to predict
  - Don't need to predict N-t target
  - T target doesn't usually change
- Only need to predict taken-branch targets
- Prediction is really just a “cache”
  - Branch Target Buffer (BTB)



# Branch Target Buffer (BTB)



# Set-Associative BTB

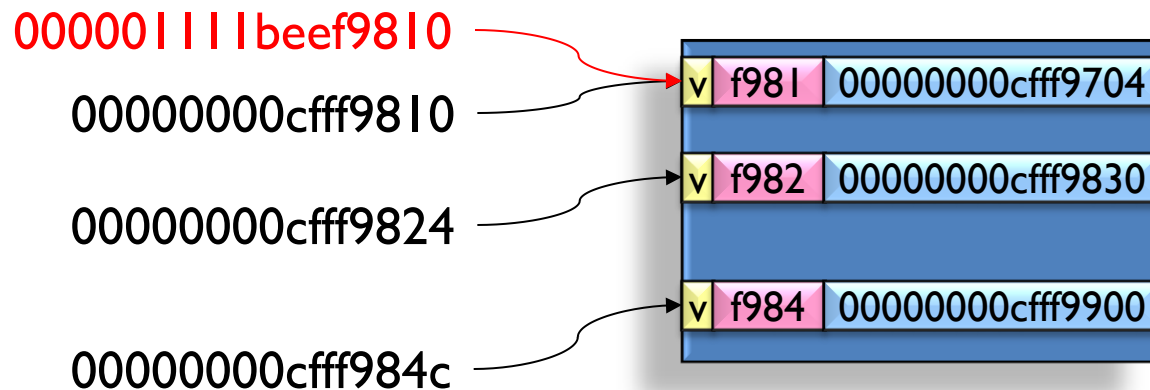
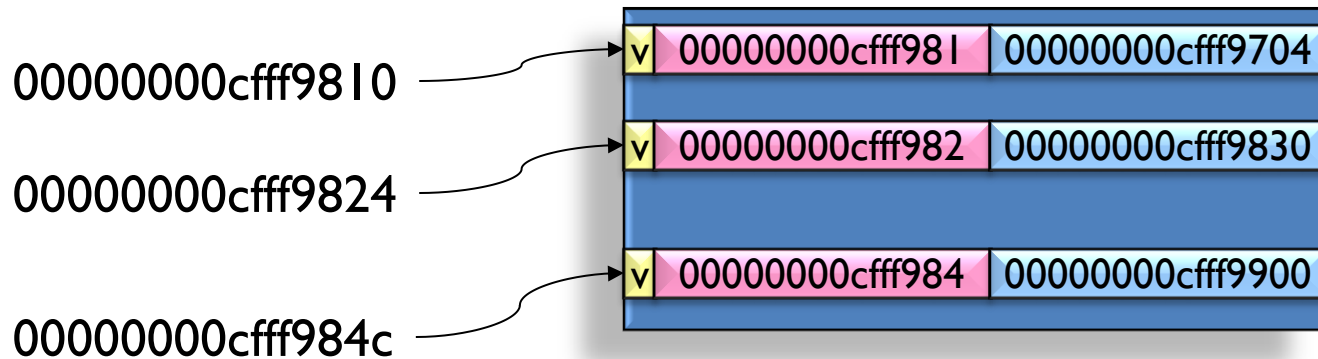


# Making BTBs Cheaper

- Branch prediction is permitted to be wrong
  - Processor has ways to detect mispredictions
  - Correctness of execution is always preserved
  - Performance may be affected

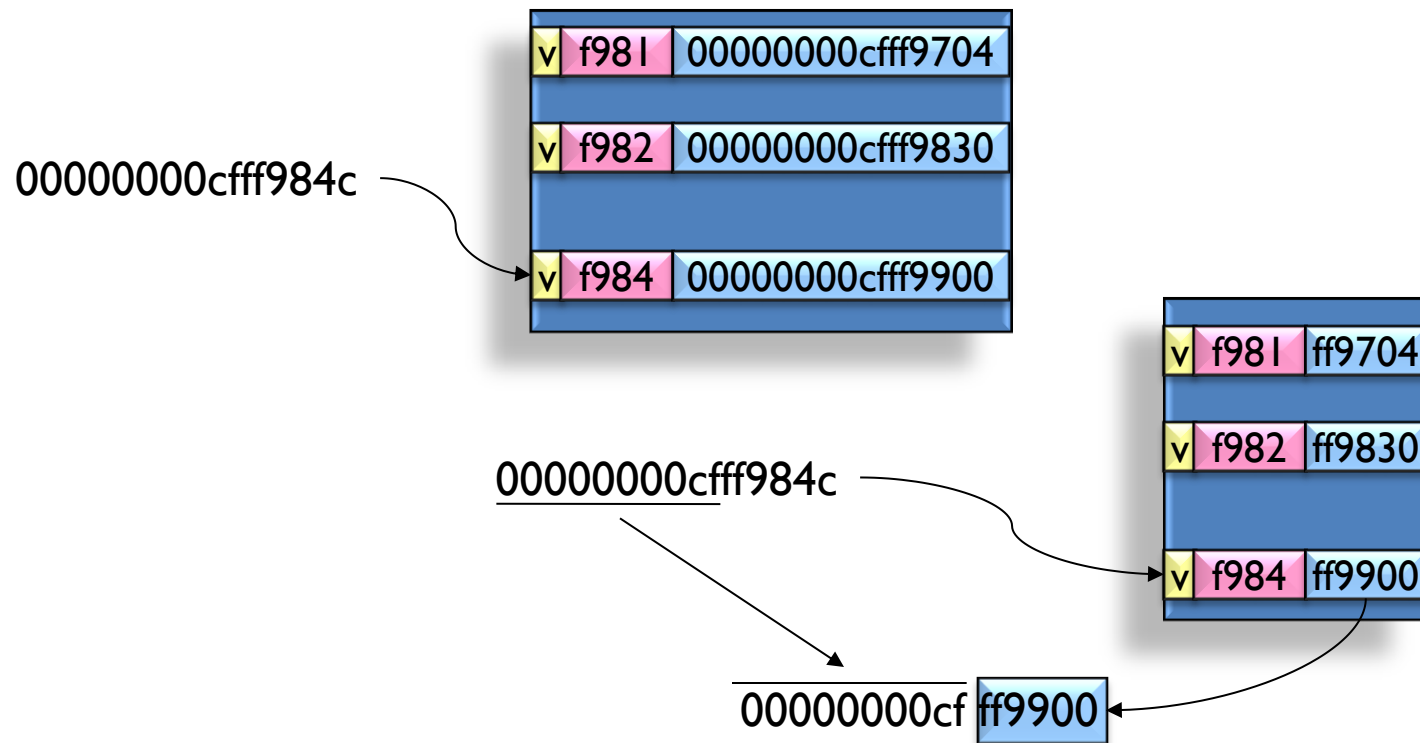
Can tune BTB accuracy based on cost

# BTB w/Partial Tags



Fewer bits to compare, but prediction may alias

# BTB w/PC-offset Encoding

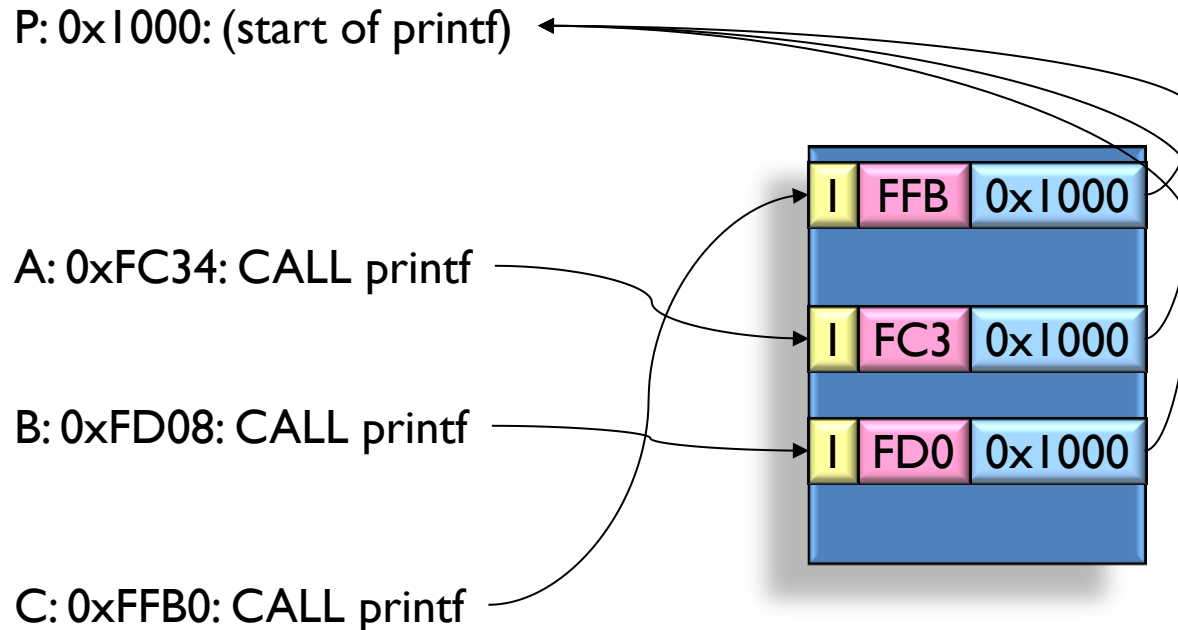


If target too far or PC rolls over, will mispredict

## BTB Miss?

- Dir-Pred says “taken”
- Target-Pred (BTB) misses
  - Could default to fall-through PC (as if Dir-Pred said N-t)
    - But we know that’s likely to be wrong!
- Stall fetch until target known ... when’s that?
  - PC-relative: after decode, we can compute target
  - Indirect: must wait until register read/exec

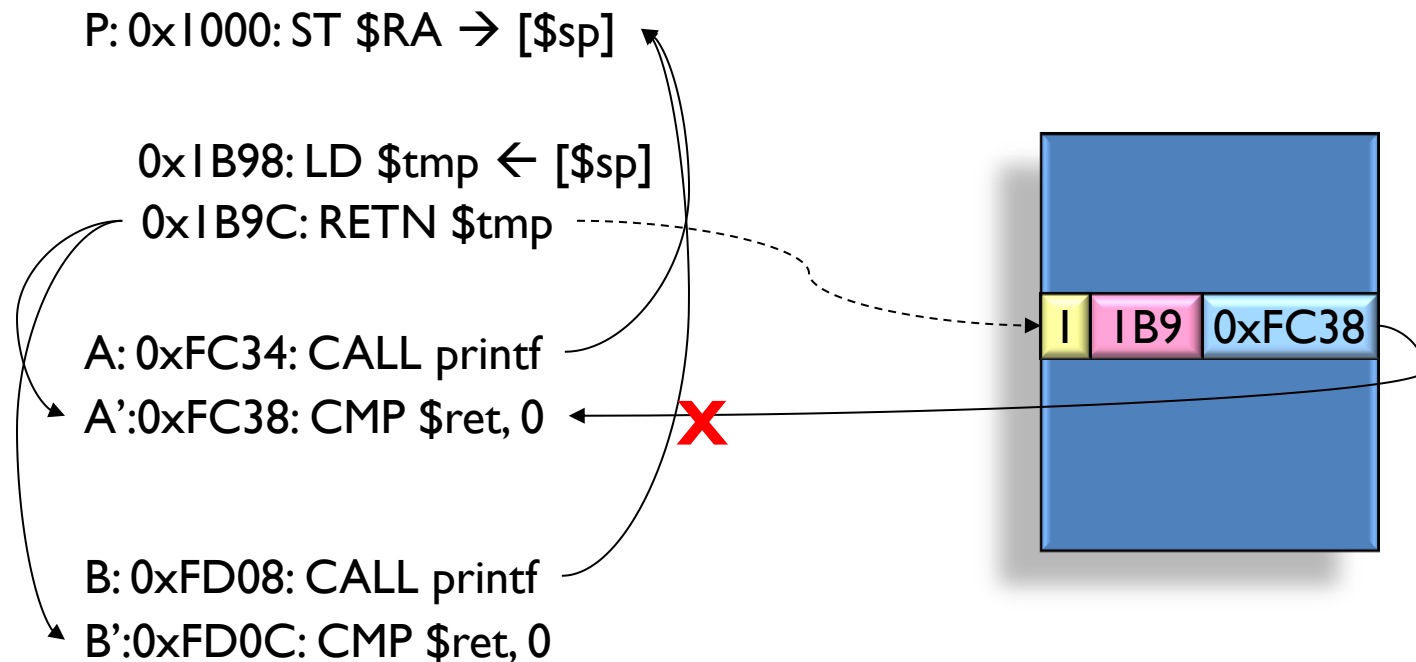
# Subroutine Calls



BTB can easily predict target of calls



# Subroutine Returns



BTB can't predict return for multiple call sites

# Return Address Stack (RAS)

- Keep track of call stack

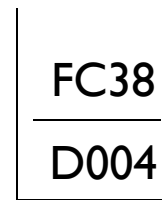
A: 0xFC34: CALL printf  
FC38

P: 0x1000: ST \$RA → [\$sp]

...

0x1B9C: RETN \$tmp

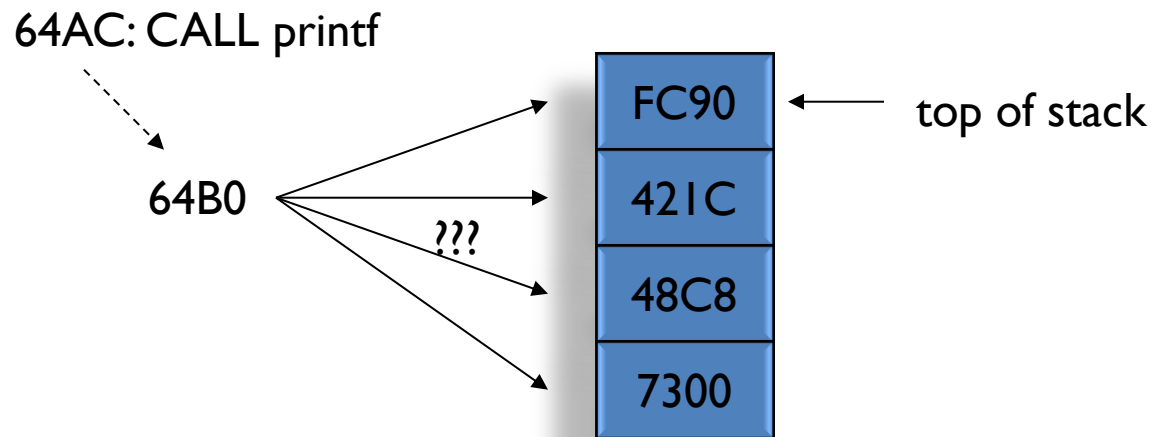
A': 0xFC38: CMP \$ret, 0



FC38

# Return Address Stack Overflow

1. Wrap-around and overwrite
  - Will lead to eventual misprediction after four pops
2. Do not modify RAS
  - Will lead to misprediction on next pop



# Branches Have Locality

- If a branch was previously taken...
  - There's a good chance it'll be taken again

```
for(i=0; i < 100000; i++)  
{  
    /* do stuff */  
}
```




This branch will be taken  
99,999 times in a row.

# Simple Direction Predictor

- Always predict N-t
  - No fetch bubbles (always just fetch the next line)
  - Does horribly on loops
- Always predict T
  - Does pretty well on loops
  - What if you have if statements?

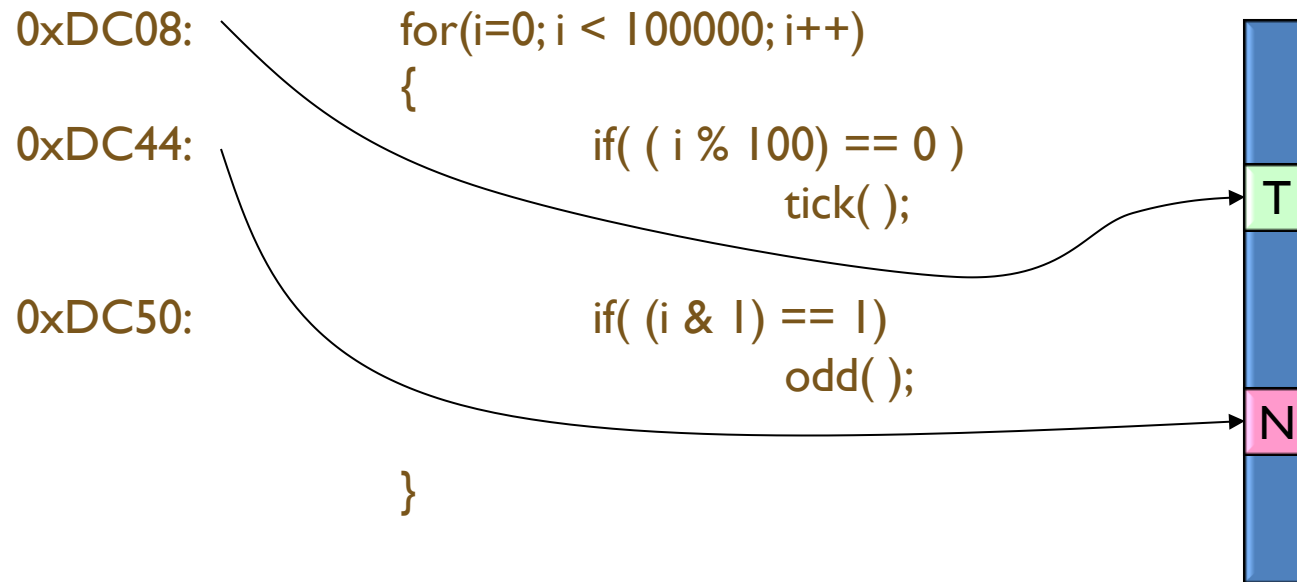
```
p = calloc(num, sizeof(*p));  
if(p == NULL)  
    error_handler( );
```



This branch is practically never taken

# Last Outcome Predictor

- Do what you did last time



0xDC08:TTTTTTTTTTTTT ... TTTTTTTTTTTNTTTTTTTTTT ...

100,000 iterations

TN

NT

2 / 100,000

99.998%  
Prediction  
Rate

**0xDC44:**TTTTT ... TNTTTTT ... TNTTTTT ...

2 / 100 ◀

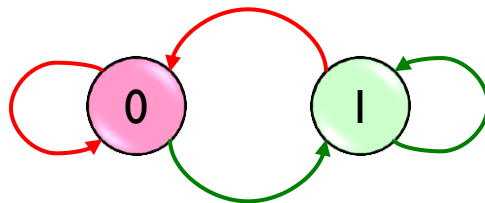
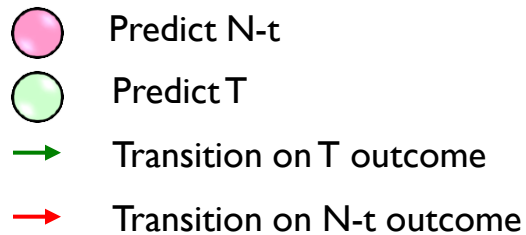
98.0%

0xDC50:TNTNTNTNTNTNTNTNTNTNTNTNTNTNTNT...

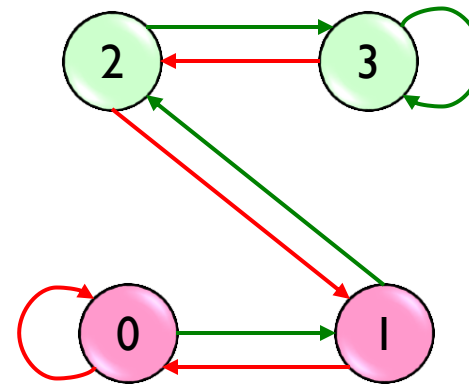
2 / 2 ◀

0.0%

# Saturating Two-Bit Counter



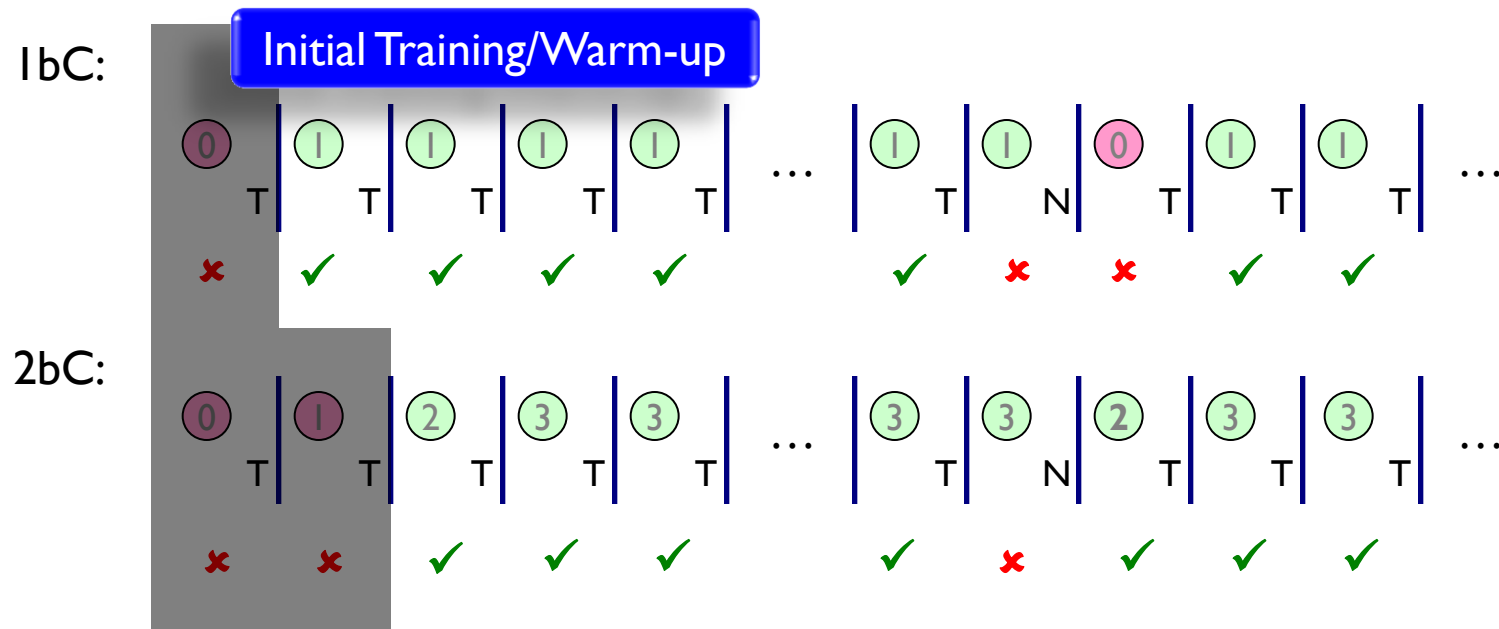
FSM for Last-Outcome  
Prediction



FSM for 2bC  
(2-bit Counter)



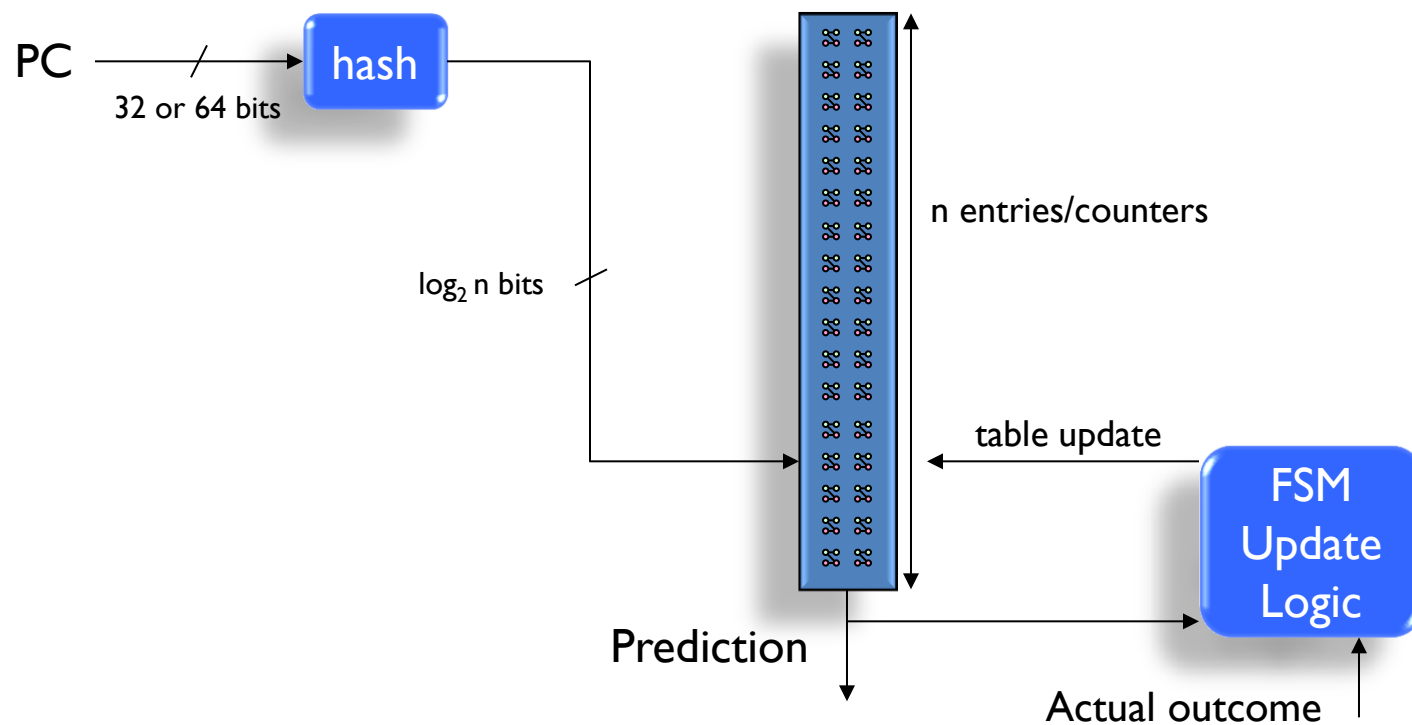
# Example



Only 1 Mispredict per N branches now!  
DC08: 99.999%    DC04: 99.0%

2x reduction in misprediction rate

# Typical Organization of 2bC Predictor



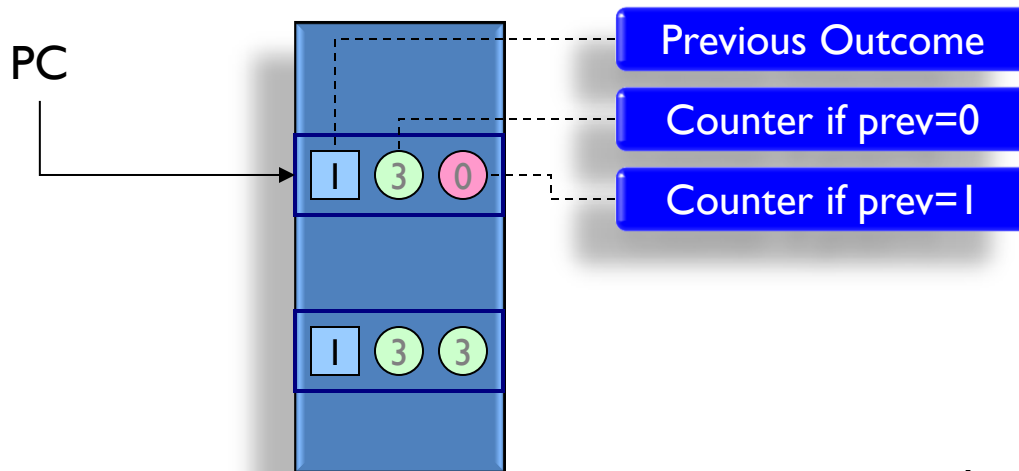
# Typical Branch Predictor Hash

- Take the  $\log_2 n$  least significant bits of PC
- May need to ignore some bits
  - In RISC, insns. are typically 4 bytes wide
    - Low-order bits zero
  - In CISC (ex. x86), insns. Can start anywhere
    - Probably don't want to shift

# Dealing with Toggling Branches

- Branch at 0xDC50 changes on every iteration
  - 1bc and 2bc don't do too well (50% at best)
  - But it's still obviously predictable
- Why?
  - It has a repeating pattern: (NT)\*
  - How about other patterns? (TTNTN)\*
- Use branch correlation
  - Branch outcome is often related to previous outcome(s)

# Track the *History* of Branches



prev = 1    (3) (3)    prediction = T ✖

prev = 0    (3) (2)    prediction = T

prev = 1    (3) (2)    prediction = T

prev = 1    (3) (3)    prediction = T

prev = 1    (3) (0)    prediction = N

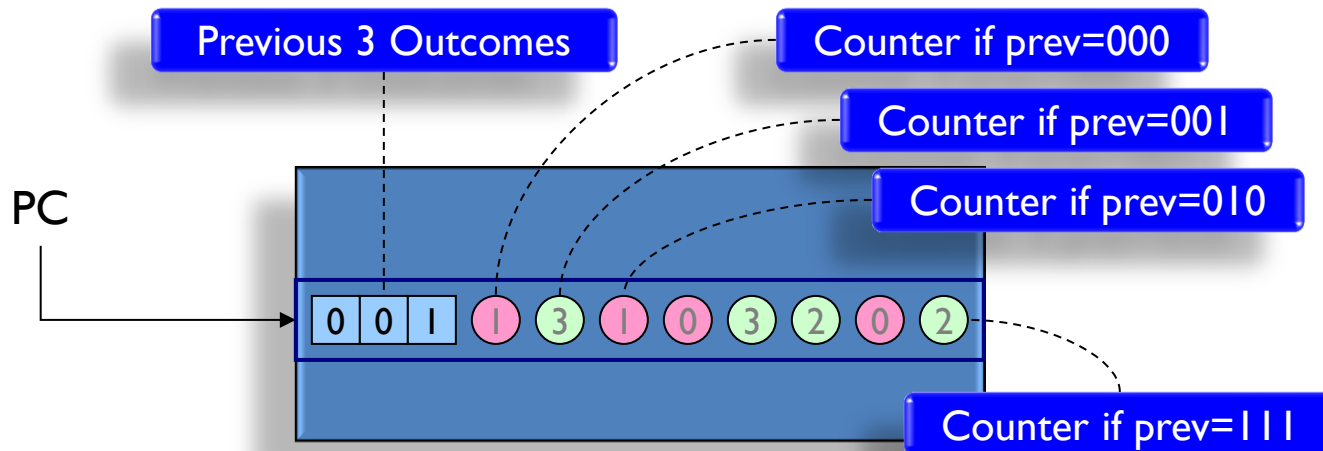
prev = 0    (3) (0)    prediction = T

prev = 1    (3) (0)    prediction = N

prev = 0    (3) (0)    prediction = T

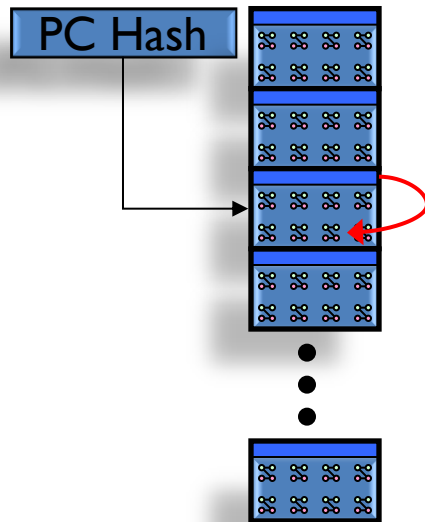
# Deeper History Covers More Patterns

- Counters learn “pattern” of prediction

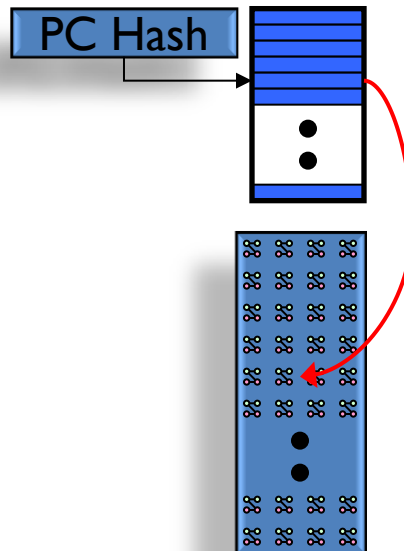


001  $\rightarrow$  1; 011  $\rightarrow$  0; 110  $\rightarrow$  0; 100  $\rightarrow$  1  
 00110011001... (0011)\*

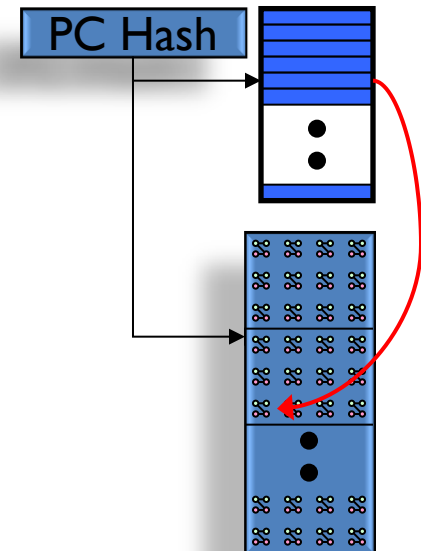
# Predictor Organizations



Different pattern for each branch PC




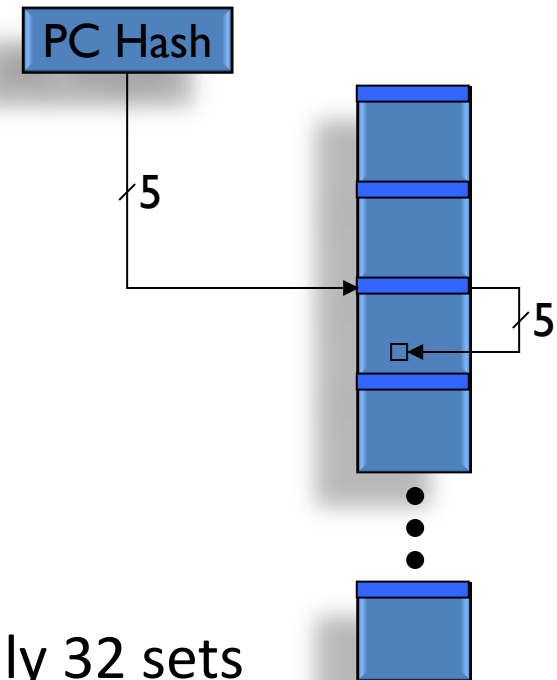
Shared set of patterns



Mix of both


# Branch Predictor Example (1/2)

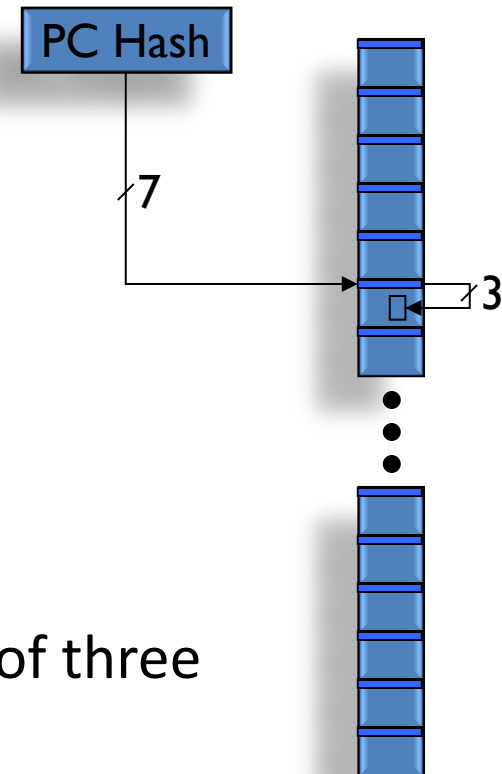
- 1024 counters ( $2^{10}$ )
  - 32 sets ()
    - 5-bit PC hash chooses a set
  - Each set has 32 counters
    - $32 \times 32 = 1024$
    - History length of 5 ( $\log_2 32 = 5$ )
- Branch collisions
  - 1000's of branches collapsed into only 32 sets





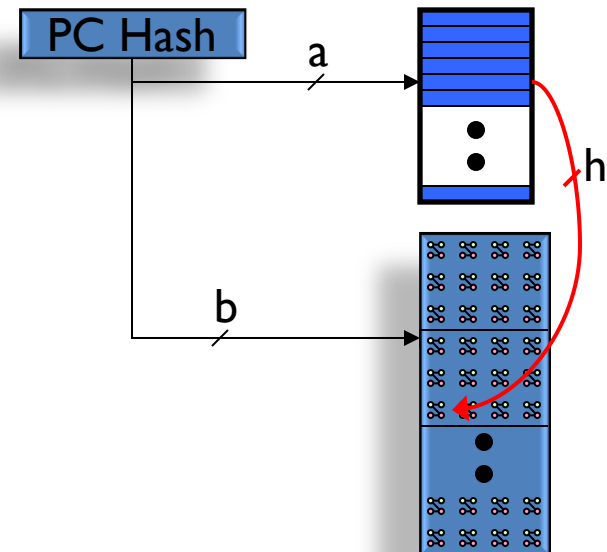
# Branch Predictor Example (2/2)

- 1024 counters ( $2^{10}$ )
  - 128 sets (  )
    - 7-bit PC hash chooses a set
  - Each set has 8 counters
    - $128 \times 8 = 1024$
    - History length of 3 ( $\log_2 8 = 3$ )
- Limited Patterns/Correlation
  - Can now only handle history length of three



# Two-Level Predictor Organization

- Branch History Table (BHT)
  - $2^a$  entries
  - $h$ -bit history per entry
- Pattern History Table (PHT)
  - $2^b$  sets
  - $2^h$  counters per set
- Total Size in bits
  - $h \times 2^a + 2^{(b+h)} \times 2$



Each entry is a 2-bit counter

# Classes of Two-Level Predictors

- $h = 0$  or  $a = 0$  (Degenerate Case)
  - Regular table of  $2^b C$ 's ( $b = \log_2 \text{counters}$ )
- $h > 0, a > 0$ 
  - “Local History” 2-level predictor
  - Predict branch from ***its own*** previous outcomes
- $h > 0, a = 0$ 
  - “Global History” 2-level predictor
  - Predict branch from previous outcomes of ***all*** branches

# Why Global Correlations Exist

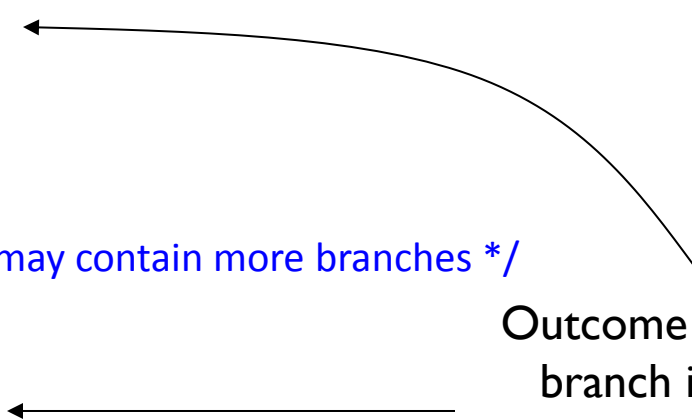
Example: related branch conditions

```
p = findNode(foo);
```

```
A: if ( p is parent )  
    do something;
```

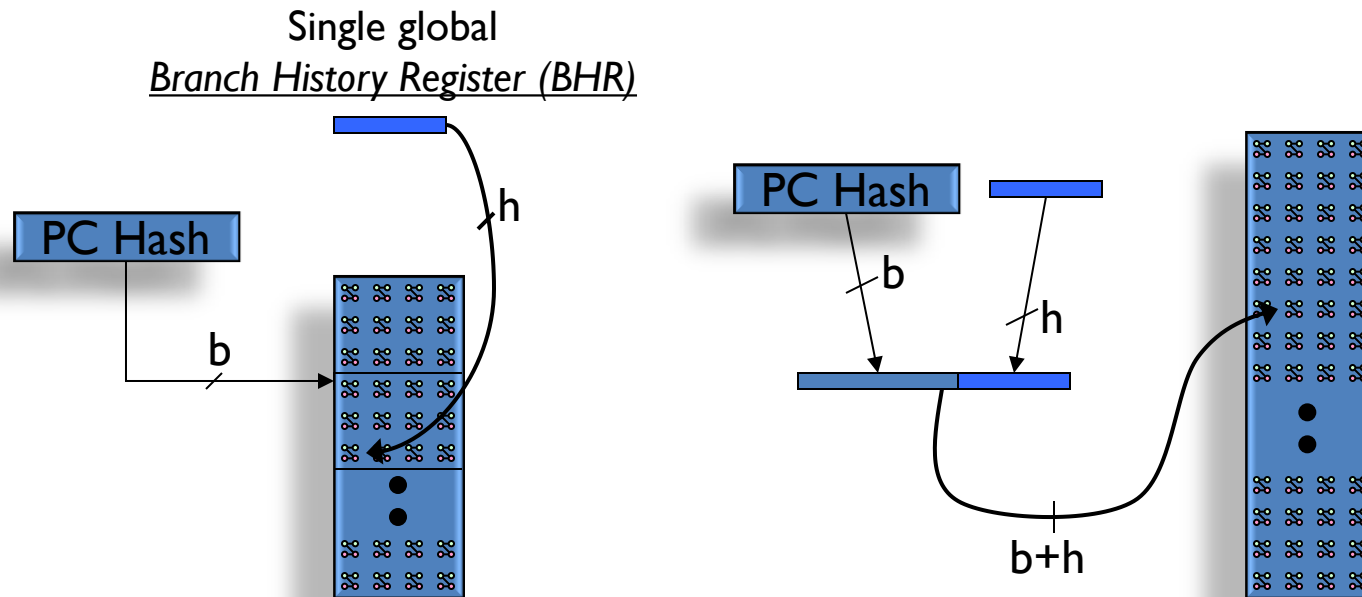
```
do other stuff; /* may contain more branches */
```

```
B: if ( p is a child )  
    do something else;
```



Outcome of second  
branch is always  
opposite of the first  
branch

# A Global-History Predictor

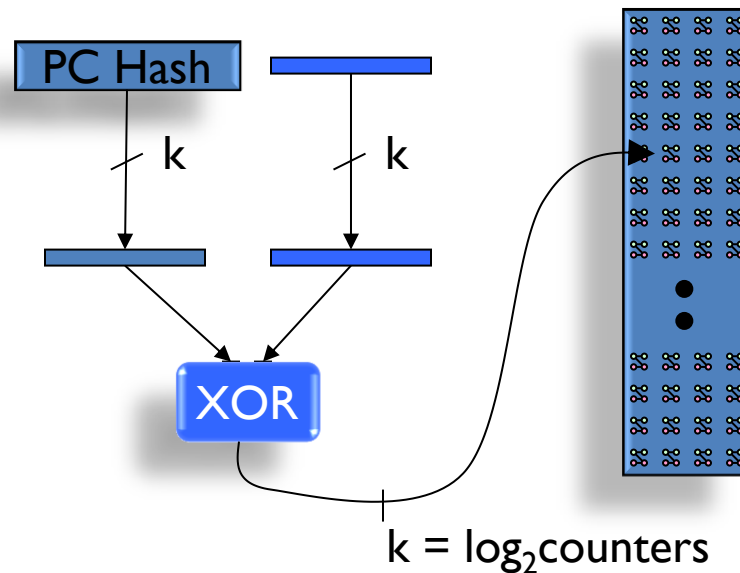


# Tradeoff Between B and H

- For fixed number of counters
  - Larger  $h \rightarrow$  Smaller  $b$ 
    - Larger  $h \rightarrow$  longer history
      - Able to capture more patterns
      - Longer warm-up/training time
    - Smaller  $b \rightarrow$  more branches map to same set of counters
      - More interference
  - Larger  $b \rightarrow$  Smaller  $h$ 
    - Just the opposite...

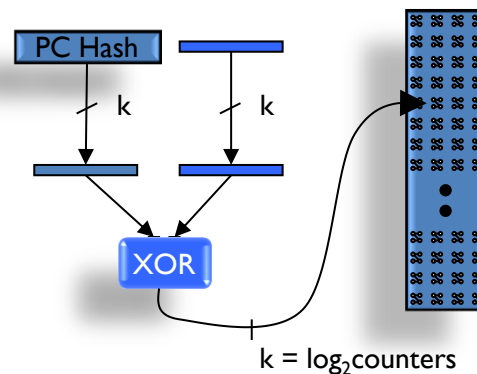
# Combined Indexing (1/2)

- “gshare” (S. McFarling)



## Combined Indexing (2/2)

- Not all  $2^h$  “states” are used
  - (TTNN)\* uses  $\frac{1}{4}$  of the states for a history length of 4
  - (TN)\* uses two states regardless of history length
- Not all bits of the PC are uniformly distributed
- Not all bits of the history are uniformly correlated
  - More recent history more likely to be strongly correlated

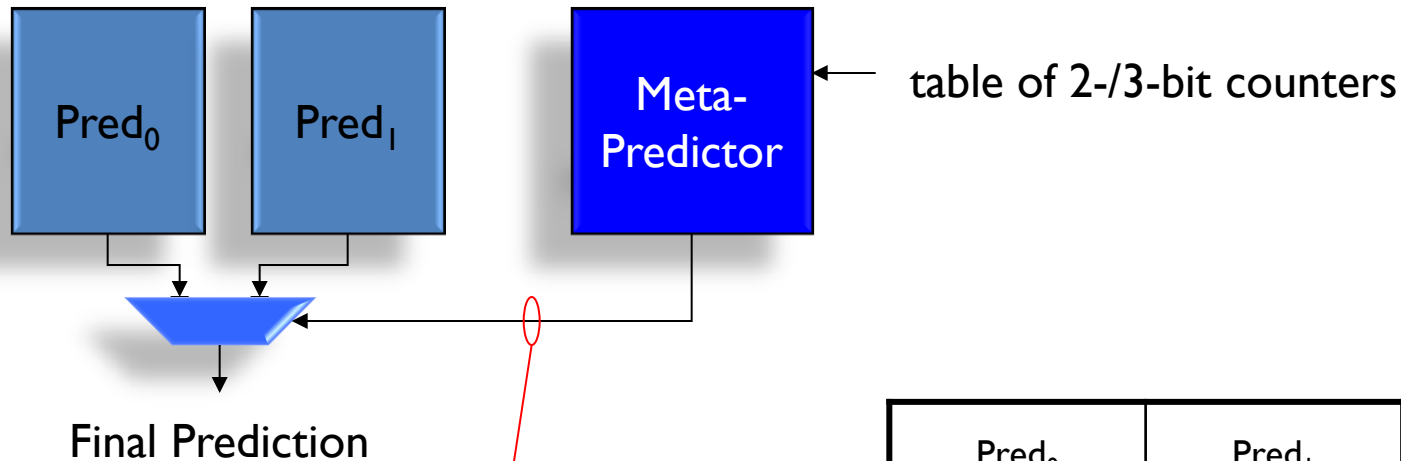




# Combining Predictors

- Some branches exhibit local history correlations
  - ex. loop branches
- Some branches exhibit global history correlations
  - “spaghetti logic”, ex. if-elsif-elsif-elsif-else branches
- Global and local correlation often exclusive
  - Global history hurts locally-correlated branches
  - Local history hurts globally-correlated branches

# Tournament Hybrid Predictors



If meta-counter MSB = 0,  
use  $\text{pred}_0$  else use  $\text{pred}_1$

$\text{Pred}_0$	$\text{Pred}_1$	Meta Update
x	x	---
x	✓	Inc
✓	x	Dec
✓	✓	---

# Pros and Cons of Long Branch Histories

- Long global history provides *context*
  - More potential sources of correlation
- Long history incurs costs
  - PHT cost increases exponentially:  $O(2^h)$  counters
  - Training time increases, possibly decreasing accuracy

# Predictor Training Time

- Ex: prediction equals opposite for 2<sup>nd</sup> most recent

- Hist Len = 2

- 4 states to train:

NN → T

NT → T

TN → N

TT → N

- Hist Len = 3

- 8 states to train:

NNN → T

NNT → T

NTN → N

NTT → N

TNN → T

TNT → T

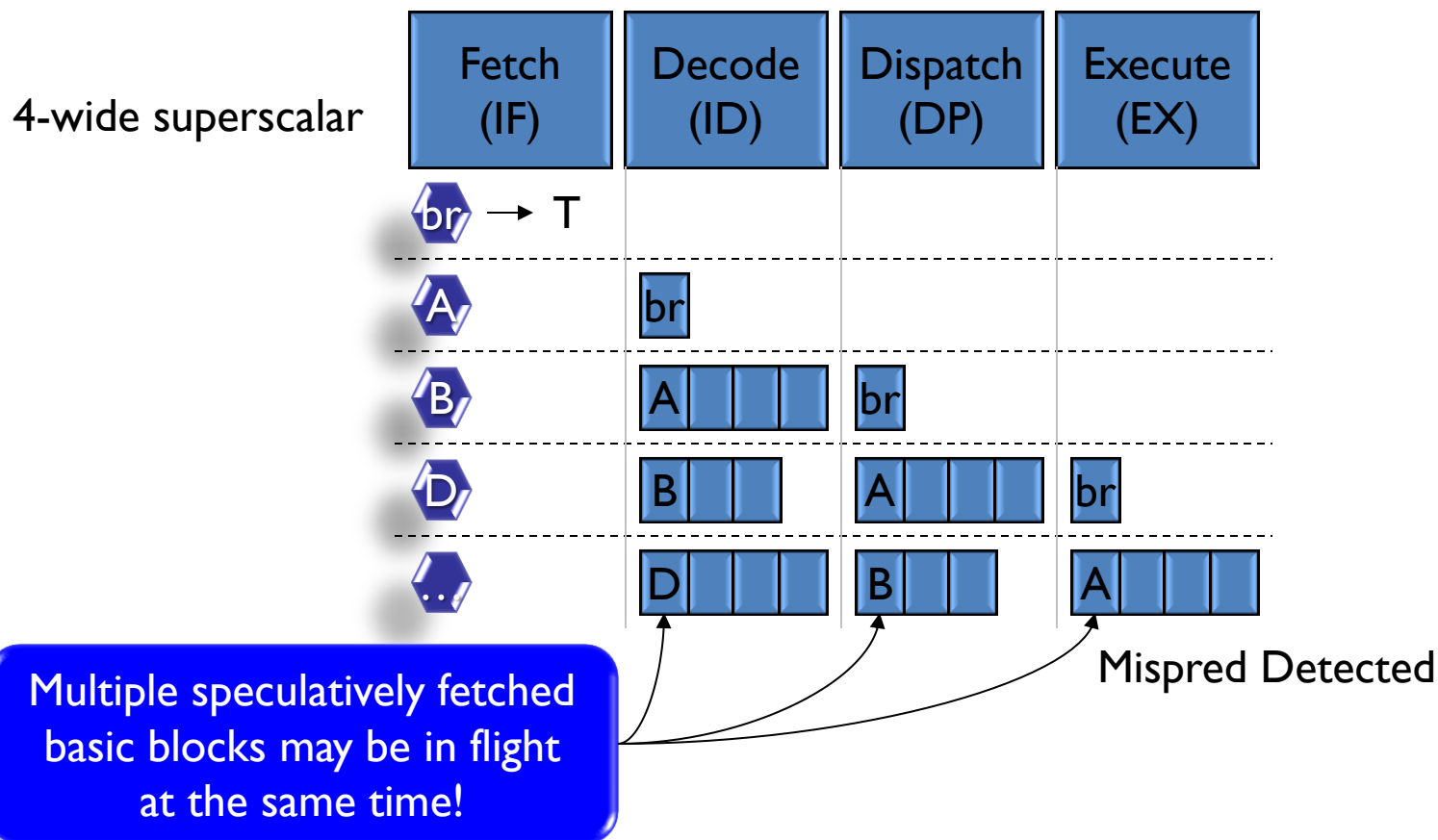
TTN → N

TTT → N

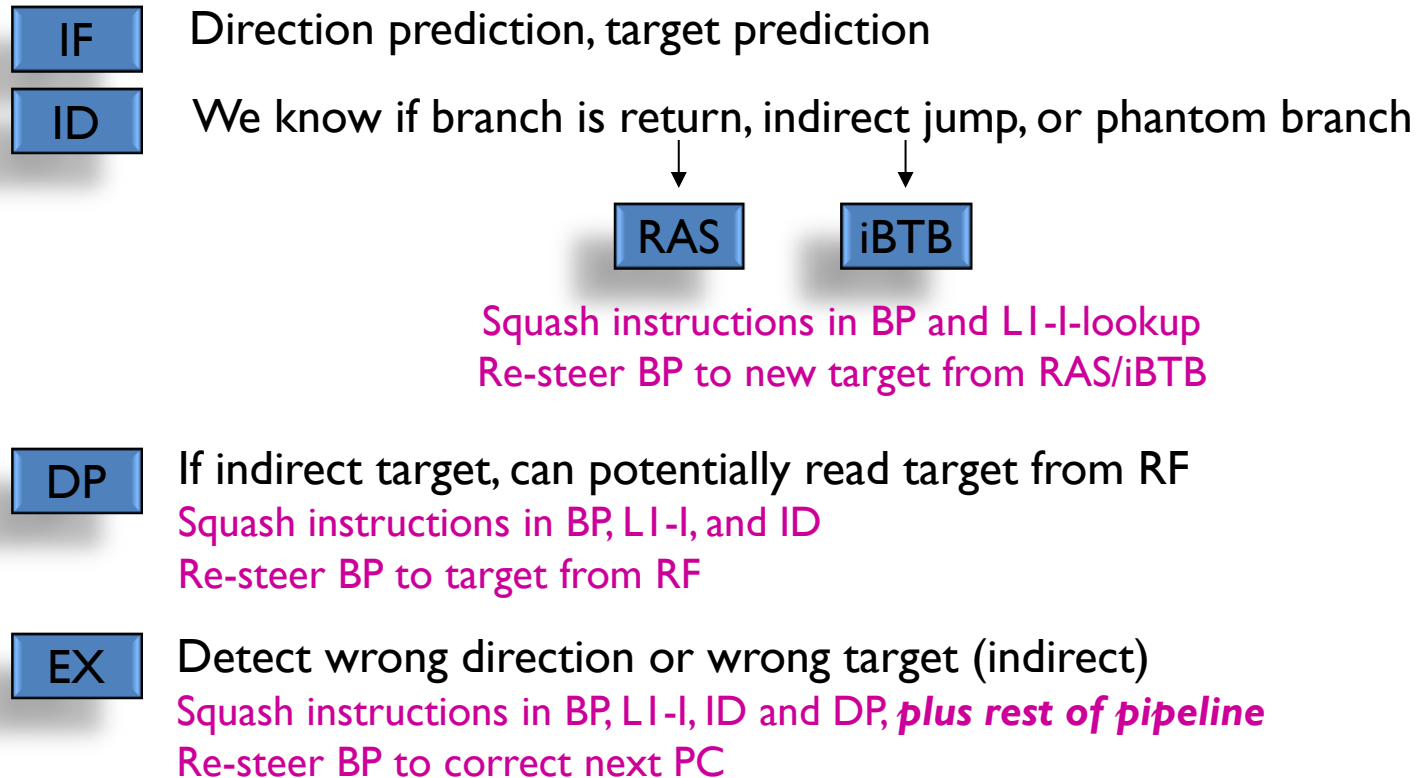
# Branch Predictions Can Be Wrong

- How/when do we detect a misprediction?
- What do we do about it?
  - Re-steer fetch to correct address
  - Hunt down and squash instructions from the wrong path

# Branch Mispredictions in the Pipeline (1/2)

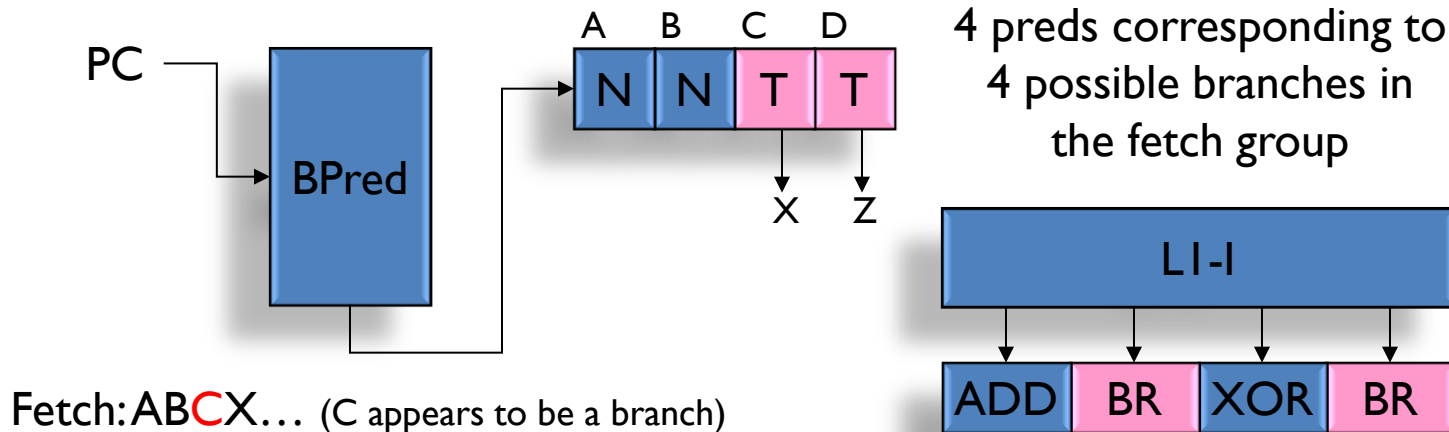


# Branch Mispredictions in the Pipeline (2/2)



# Phantom Branches

- May occur when performing multiple bpreds

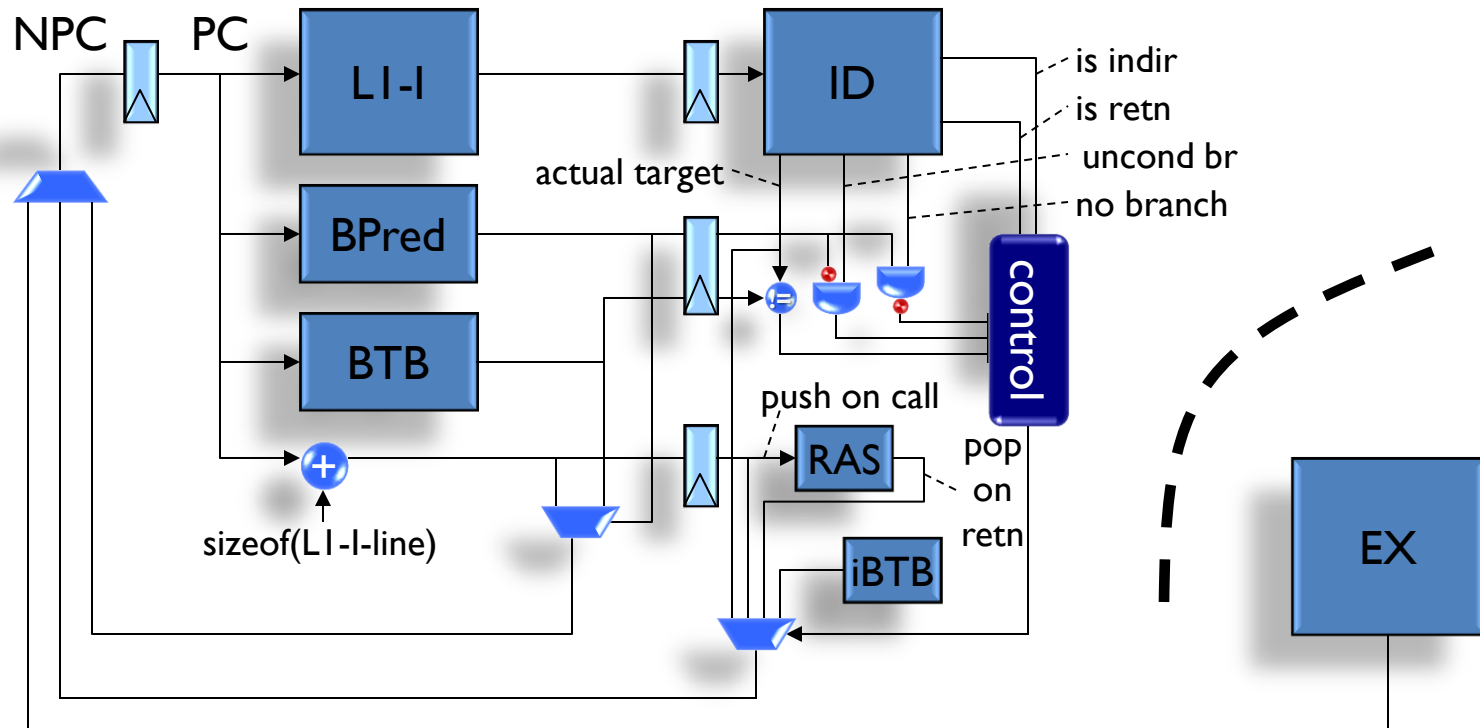


After fetch, we discover C cannot be taken because it is not even a branch! This is a phantom branch.

Should have fetched: ABCDZ...



# Front-End Hardware Organization



# Speculative Branch Update (1/3)

- Ideal branch predictor operation
  1. Given PC, predict branch outcome
  2. Given actual outcome, update/train predictor
  3. Repeat
- Actual branch predictor operation
  - Streams of predictions and updates proceed parallel

Predict:



Update:



time

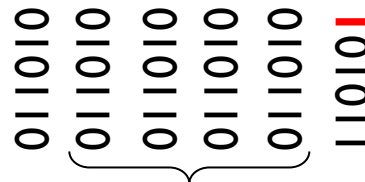
Can't wait for update before making new prediction

# Speculative Branch Update (2/3)

- BHR update cannot be delayed until commit
  - But outcome not known until commit

Predict: 

Update: 

BHR: 

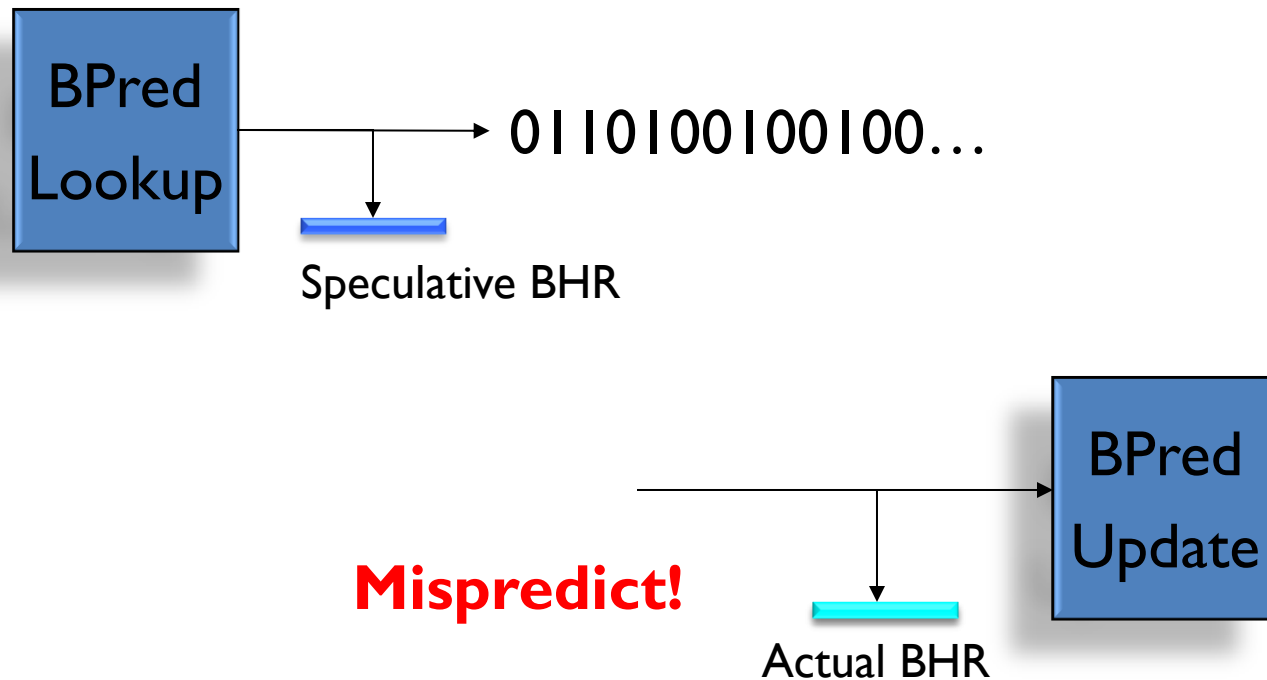
Branches B-E all predicted with

The same stale BHR value

# Speculative Branch Update (3/3)

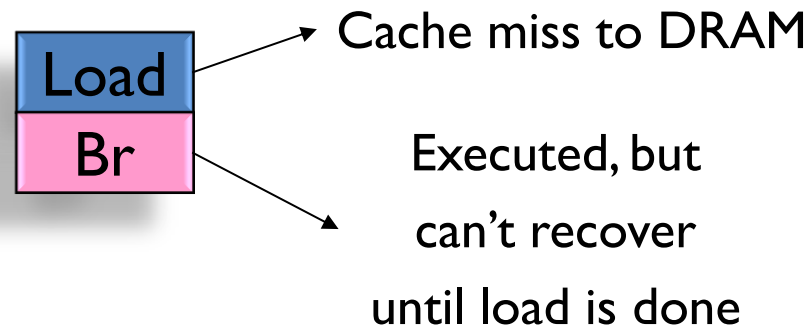
- Update branch history using predictions
  - *Speculative* update
- If predictions are correct, then BHR is correct
- What happens on a misprediction?
  - Commit-time BHR recovery
  - Execution-time BHR recovery

# Commit-time BHR recovery



# Execution-time BHR recovery

- Commit-time may delay misprediction recovery



- Instead, “checkpoint” BHR at time of prediction
  - Roll back to checkpoint for recovery
  - Must track where to roll back to
  - In-flight branches limited by number of checkpoints

# Overriding Branch Predictors (1/2)

- Use two branch predictors
  - 1<sup>st</sup> one has single-cycle latency (fast, medium accuracy)
  - 2<sup>nd</sup> one has multi-cycle latency, but more accurate
  - Second predictor can ***override*** the 1<sup>st</sup> prediction

Get speed without full penalty of low accuracy

# Overriding Branch Predictors (2/2)

