Performance and Branch Prediction

Computer Architecture

ECE 6913

Brandon Reagen

NYU TANDON SCHOOL OF ENGINEERING

What we'll cover today

1) Understanding performance
2) Control hazards
3) Branch prediction

Performance metrics!

As we'll see now, comparing and computer
computer performance isn't as trivial as it sounds..

Revisit some definitions

Execution/Response time (Latency)

- Elapsed time between the start and completion of an event
- How long did did your project take?
- Where have we seen this in class?

Throughput (Bandwidth)

- Total amount of work done within some amount of time
- How many lines of code (or bugs) did you write per hour?

Some new definitions

Assume we're running a program on a machine X

Performance:

Performance(X) = 1 / Execution time (x)

It's the inverse of the runtime.

"X is n times faster than Y"

Performance(X) / Performance(Y) => speedup factor of n

On averages..

Arithmetic mean:
Use when using the same units (e.g., time)

$$\frac{1}{n}\sum_{i=1}^{n} Time_i \quad \text{or} \quad \frac{1}{n}\sum_{i=1}^{n} Weight_i * Time_i$$

Harmonic mean:
Use when data values are ratios of variables with different rates (e.g., speedup!)

$$\frac{n}{\sum_{i=1}^{n} \frac{1}{Rate_i}} \quad \text{or} \quad \frac{n}{\sum_{i=1}^{n} \frac{Weight_i}{Rate_i}}$$

Why Harmonic Mean? Driving example

30 mph for the first 10 miles
90 mph for the next 10 miles

Average speed? (30+90)/2 = 60 mph??
Wrong!

Average speed = total distance / total time
(2)/(1/30 + 1/90) = 180/4 = 45 mph

Make the common case fast

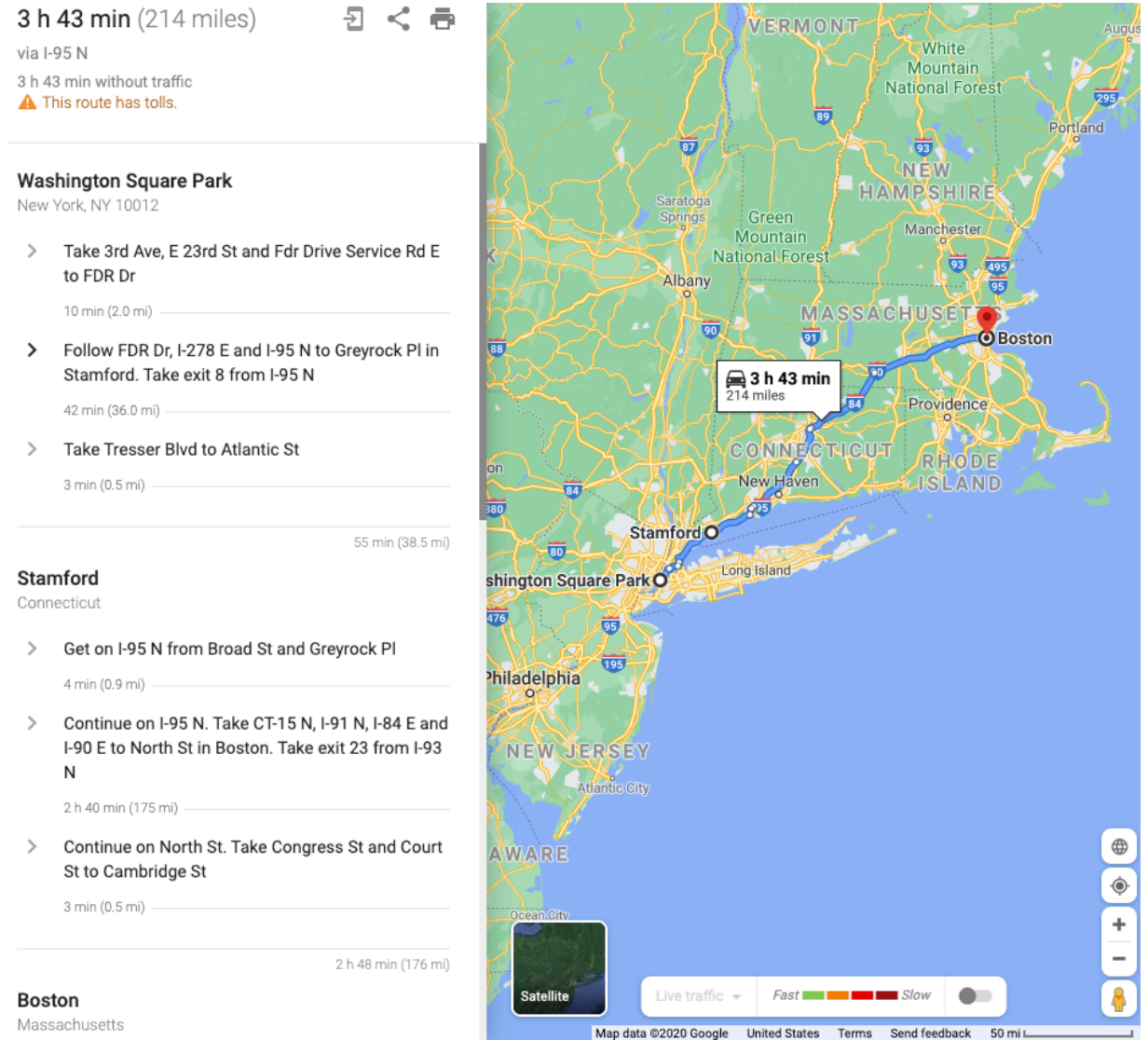
Drive NYC -> Boston
- 130 mph NYC -> Stamford (38 mi)
- 65 mph Stamford -> Boston (176 mi)

- First takes 38/130 = 17.5 minutes
- Second takes 176/65 = 162.5 minutes
- Total = 180 min.

Just go 65 the whole way: 197 minutes

Speedup = 197 / 180 = 1.09x
Not great.. Probably not worth getting arrested.

Make the common case fast!



NYU TANDON SCHOOL OF ENGINEERING

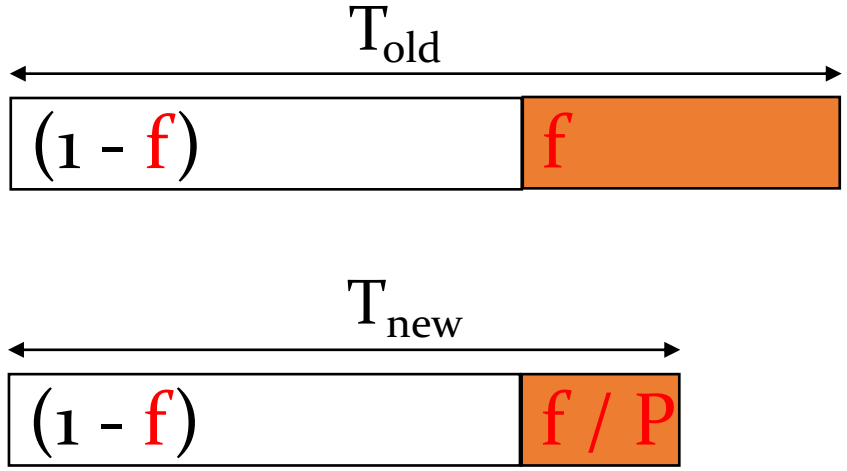
Amdahl's Law (Law of Diminishing Returns)

Speedup is limited by the fraction of time it can be applied

f = fraction of code sped up

P = speedup factor

$$\text{Speedup} = \text{Perf}_{new} / \text{Perf}_{old} = T_{old} / T_{new} = \frac{1}{(1-f) + \frac{f}{P}}$$



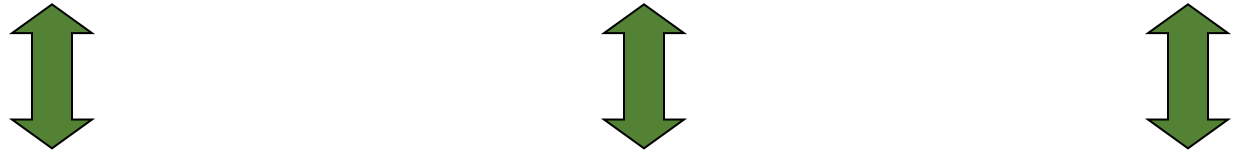
Q: If one function takes 90% of the time,

what's max speedup? (Hint: P -> inf.)

CPU Performance: “Iron law of performance”

Execution Time = Seconds / Program

$$\frac{\text{Instructions}}{\textbf{program}} \times \frac{\text{cycles}}{\text{Instruction}} \times \frac{\textbf{seconds}}{\text{cycle}}$$



- Programmer
- Algorithms
- ISA
- Compilers

- Microarchitecture
- ISA

- Microarchitecture, pipeline depth
- Circuit design
- Technology

Example of Performance Evaluation (I)

Operation	Frequency	Clock cycle count
ALU Ops (reg-reg)	43%	1
Loads	21%	2
Stores	12%	2
Branches	24%	2

Assume 25% of the ALU ops directly use a loaded operand that is not used again.
We propose adding ALU instructions that have one src operand in memory.
These new reg-mem instructions take 2 clock cycles.
Also assume that the extended instruction set increases the
branch inst. clock cycle count by 1 but does not impact to cycle time.
Would this change improve performance ?

Example of Performance Evaluation (I)

Operation	Frequency	Clock cycle count
ALU Ops (reg-reg)	43%	1
Loads	21%	2
Stores	12%	2
Branches	24%	2

Assume 25% of the ALU ops directly use a loaded operand that is not used again.
We propose adding ALU instructions that have one src operand in memory.
These new reg-mem instructions take 2 clock cycles.
Also assume that the extended instruction set increases the branch inst. clock cycle count by 1 but does not impact to cycle time.
Would this change improve performance ?

New inst.

$$Cycles_{old} = 0.43*1+0.21*2+0.12*2+0.24*2 = 1.57$$

$$Cycles_{new} = 0.25*0.43*2+(0.43-0.25*0.43)*1+(0.21-0.25*0.43)*2+0.12*2+0.24*3 = 1.703$$

reg-reg alu

Save load time!

Store and branch stays same.

Example of Performance Evaluation (II)

FP instructions = 25%
Average CPI of FP instructions = 4.0
Average CPI of other instructions = 1.33
FPSQRT = 2% of all instructions, CPI of FPSQRT = 20

Design Option 1: decrease the CPI of FQSQRT to 2

Design Option 2: decease the average CPI of all FP instructions to 2.5

Calculate: original CPI, Des opt 1 CPI, Des. Opt 2 CPI.

Original CPI = 0.25*4 + 1.33*(1-0.25) = 2.0

Option 1 CPI = 2.0 – 2%*(20-2) = 1.64

Option 2 CPI = 0.25*2.5 + 1.33*(1-0.25) = 1.625

Speedup of Option 1 = 2/1.64 = 1.2195
Speedup of Option 2 = 2/1.625 = 1.2308

Optimizing common case a little better than the special case a lot

Control hazards

Hazards!

Hazard (formal) := “when the next instruction cannot execute in the following clock cycle”

Hazard (me) := “you need something you don’t have”

There are three of them..
Think about what goes into a program and how they execute

1) Structural : hardware resources
2) Data : need result from prior computation
3) Control : need to know where to go next

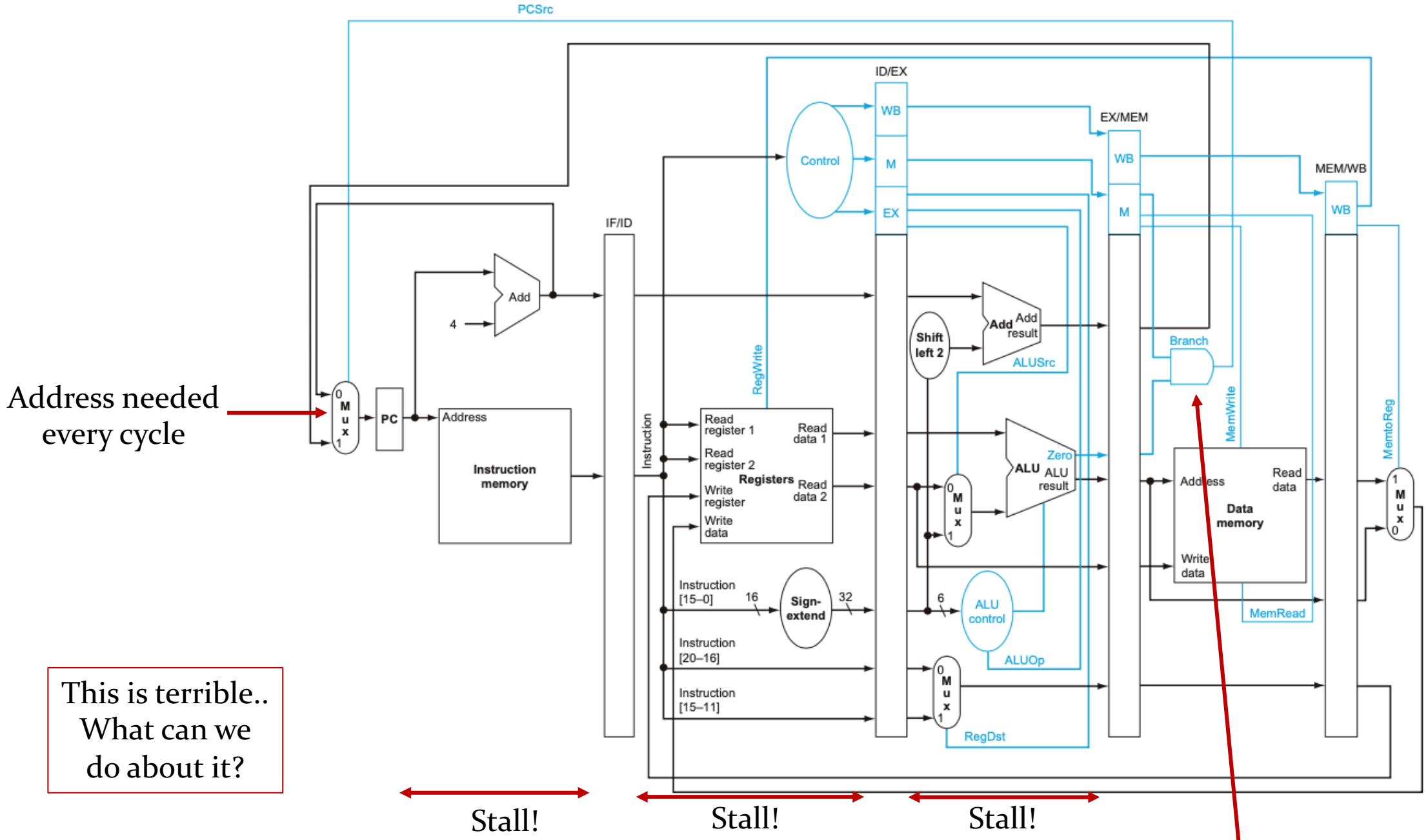
Hazards!

Hazard (formal) := “when the next instruction cannot execute in the following clock cycle”

Hazard (me) := “you need something you don’t have”

There are three of them..
Think about what goes into a program and how they execute

1) Structural : hardware resources : SOLVED!
2) Data : need result from prior computation : Forwarding!
3) Control : need to know where to go next => Branch prediction (later)



Address needed every cycle

This is terrible.. What can we do about it?

Stall!

Stall!

Stall!

Branch resolved

Solution 1: Assume not taken

- Assume not taken doesn't really change the design
  - Fetch instructions each cycle from PC+4
  - Resolve branches in MEM stage
- Pro:
  - What happens to CPI when branch resolves NOT taken?
- Con:
  - What happens to CPI when branch IS taken? (Common case..)
    - How do we implement this?
  - Flush: Discard instructions due to misprediction.
    - Insert NOPs
    - How and where?

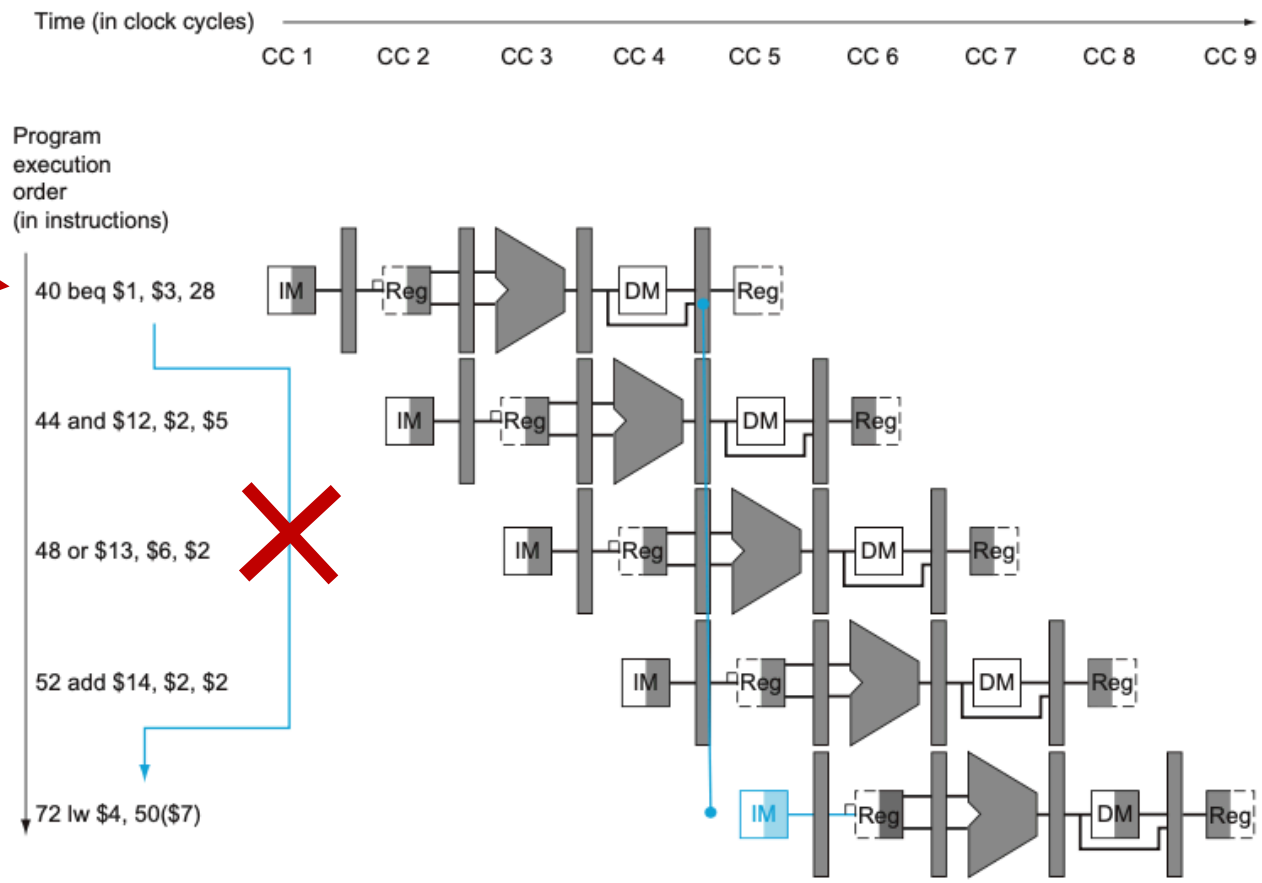
What happens w this design?

Assume that
this branch IS NOT taken.

What's going on here?

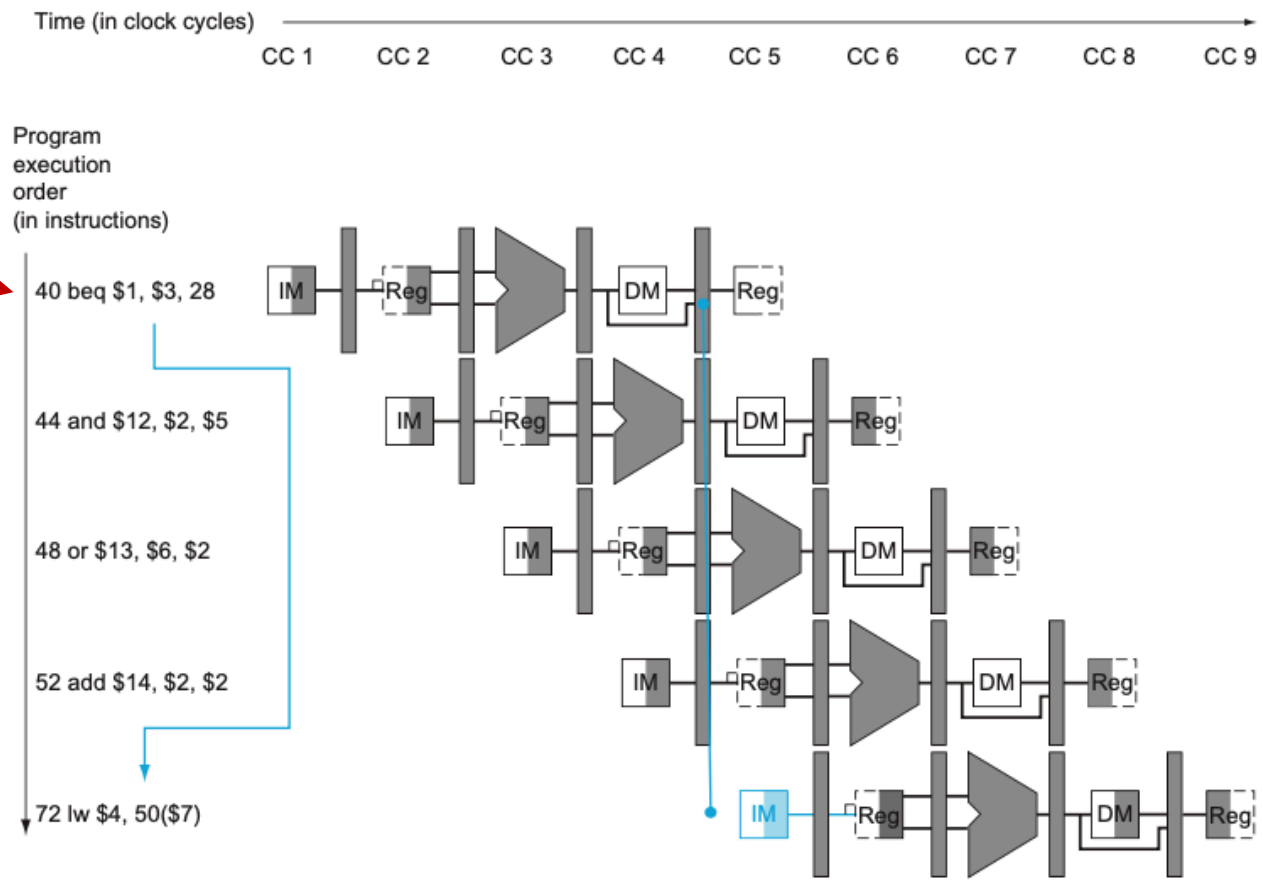
Preserve CPI!

Like the branch instruction
never occurred



What happens w this design?

Assume that
this branch IS taken.
What's going on here?



Don't know the branch outcome until CC 4.

Will execute 3 wrong instructions!
(Assuming we don't flush.)

It'd be so nice if we knew the answer earlier..

Solution 2: Move branch resolution “up”

- Branches resolved in MEM stage, but why?
- We have all the information we need during ID!
  - Know it’s a branch, which registers to compare, value to update PC

- What does this do?
  - Reduces the branch delay
  - When we flush, we waste fewer cycles

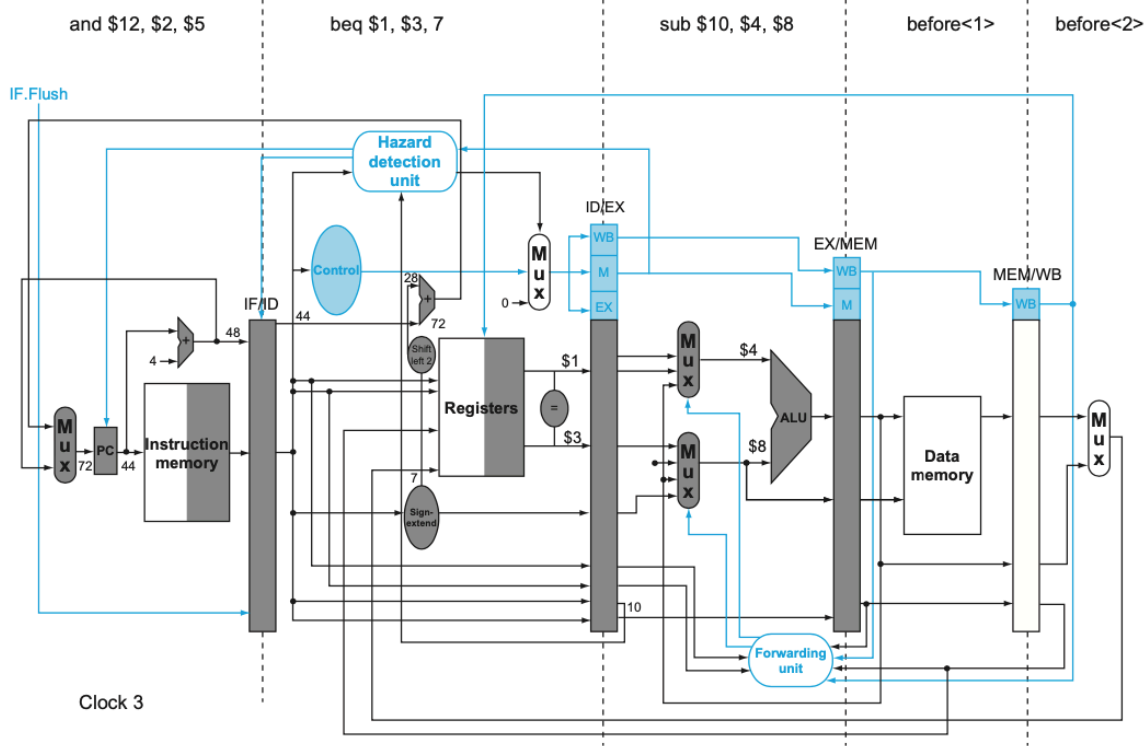
- MIPS branch conditions intentionally kept simple
  - E.g., beq and bne only need to compare values
  - We can do this by adding small circuits to the ID stage

Challenges with early branching (1)

- Computing branch target address
  - First, we already have the info we need in IF/ID pipeline reg:
    PC and branch Immediate
  - Second, we can move the branch adder from EX stage to the ID stage

Challenges with early branching (2)

- Evaluating branch decision
  - New circuits late in the ID stage
    - After we read registers, have to compare values.
    - How can we implement comparators?
  - New bypass/forward logic!
    - Now forward data must also come into ID stage
      - Specifically, to the inputs of the equality unit
    - Can come from EX/MEM or MEM/WB pipeline regs
      - Requires new control logic
      - Cannot come from ID/EX regs, why?
    - Even with forwarding, we can stall. How?
      - If branch depends on previous ALU instruction: 1 cycle
      - If branch depends on previous MEM instruction: 2 cycles



Why times 4?

```
36 sub $10, $4, $8
40 beq $1, $3, 7 # PC-relative branch to 40+4+7*4=72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```

What do we like about this?

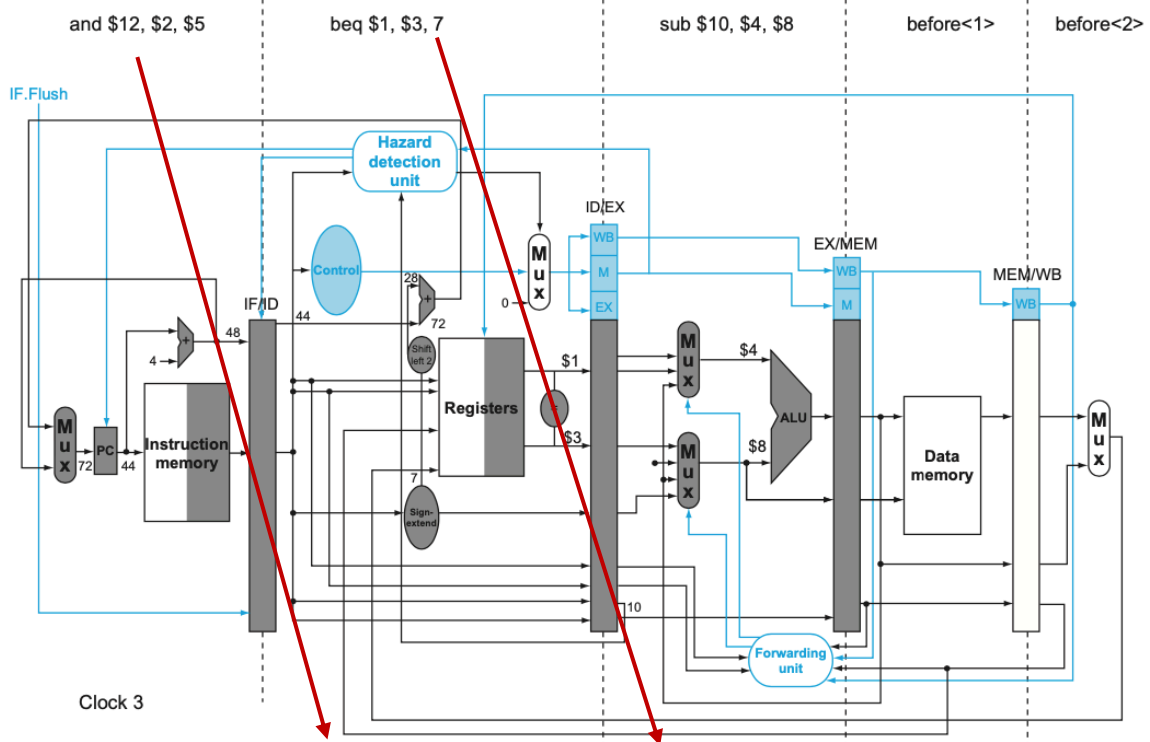
```
36 sub $10, $4, $8
40 beq $1, $3, 7 # PC-relative branch to 40+4+7*4=72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```

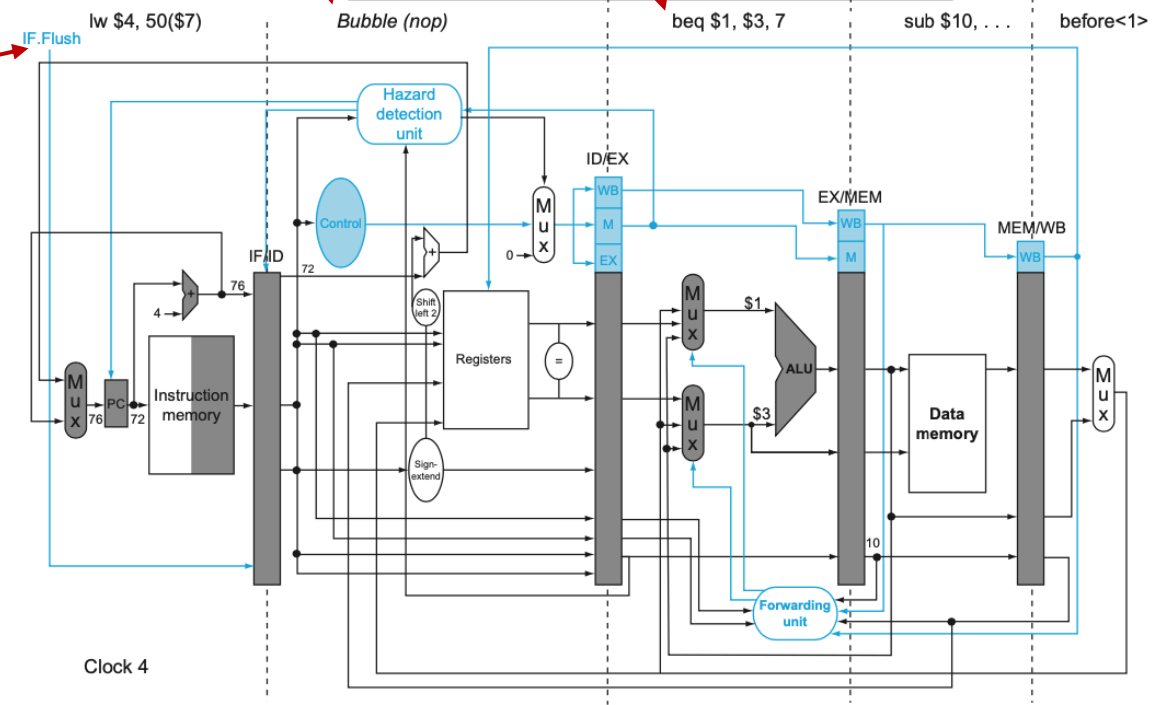
Why times 4?

What's going on here..
What did we have to fix?

1 Stall because we assumed not taken incorrectly.

Is this really that bad?
Do we even need more complex branch prediction??





NYU TANDON SCHOOL OF ENGINEERING

What happens when pipelines get "deeper" and "wider"?

- Assume a pipeline resolves branches in stage 20 (not 2) and fetches 5 inst per cycle
- Also, 20% (1 in 5) instructions are branches, uniform distribution
- How long does it take to fetch 500 instructions?
  - 100% accuracy:
    - 100 cycles, no flushes =? No wasted work!
  - 99% accuracy:
    - 100 (correct) + 20 (penalty) * 1 (wrong frequency*branches) = 120 cycles
    - 20% more cycles..
  - 95% accuracy
    - 100 (correct) + 20 (penalty) * 5 (wrong frequency * num branches) = 200 cycles
    - 100% extra instructions fetched.

Dynamic Hardware Branch Prediction

- Need to do better than always fetching not taken instructions
  - Performs poorly (think about loops)
- Branch prediction
  - **Speculate** which direction to go!
- Dynamic
  - Use results of recently taken branches while program executes
- Hardware
  - Implement with special circuits for speed.
  - Would branch prediction be part of architecture or microarchitecture?
  - Could we do it in software?

Why are Branches Predictable?

```
for (i=0; i<100; i++) {

        ….
}
```

```
addi   r10, r0, 100
add    r1,   r0, r0

L1:
… …
… …
addi   r1, r1,   1
bne    r1, r10, L1
… …
```

```
if (aa==2)
        aa = 0;
if (bb==2)
        bb = 0;
if (aa!=bb)
        ….
```

```
addi  r2,   r0, 2
bne   r10, r2, L_bb
xor   r10, r10, r10

L_bb:
bne   r11, r2, L_xx
xor   r11, r11, r11

L_xx:
beq   r10, r11, L_exit
…
Lexit:
```

Branch predictors have two jobs

This is what we'll focus on today.

- In order to perform task, predictors must
  - Speculate whether the branch is to be taken or not-taken
  - Calculate the target address
    - Target address: Where to go when branch predicted taken
    - What about unconditional jumps?

- Past predicts the future
  - To make informed decisions, we'll track previous branch directions
  - As we'll see, this sounds simple but gets quite complex

FSM of the Simplest Predictor

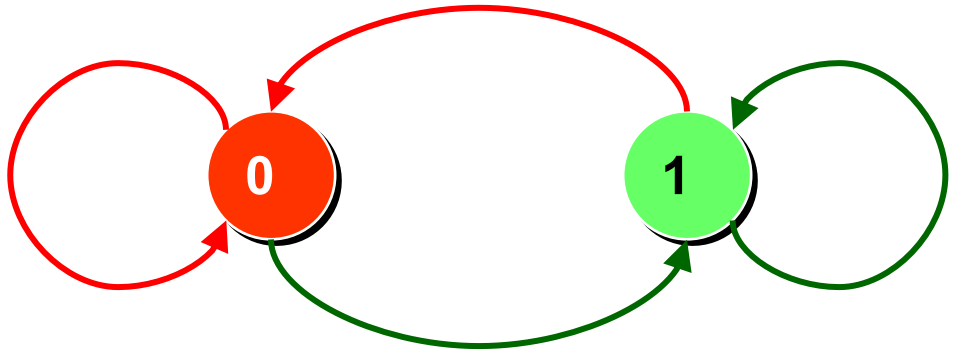
We can save context using simple state machines

States are used to make predictions

Transitions between states occur when branches are resolved

A 2-state machine

Change mind fast



If branch taken

If branch not taken

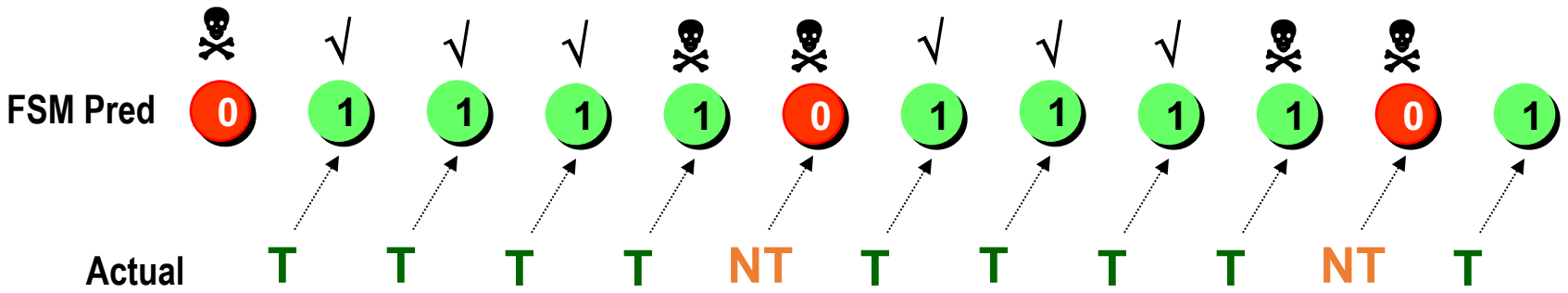
0 Predict not taken

1 Predict taken

Example using 1-bit branch FSM

```
Funct_do_loop()
        for (i=0; i<4; i++) {
                ....
        }
```

```
 addi   r10, r0, 4
 addi   r1,   r0, r0
L1:
 … …
 addi   r1, r1,   1
 bne    r1, r10, L1
```

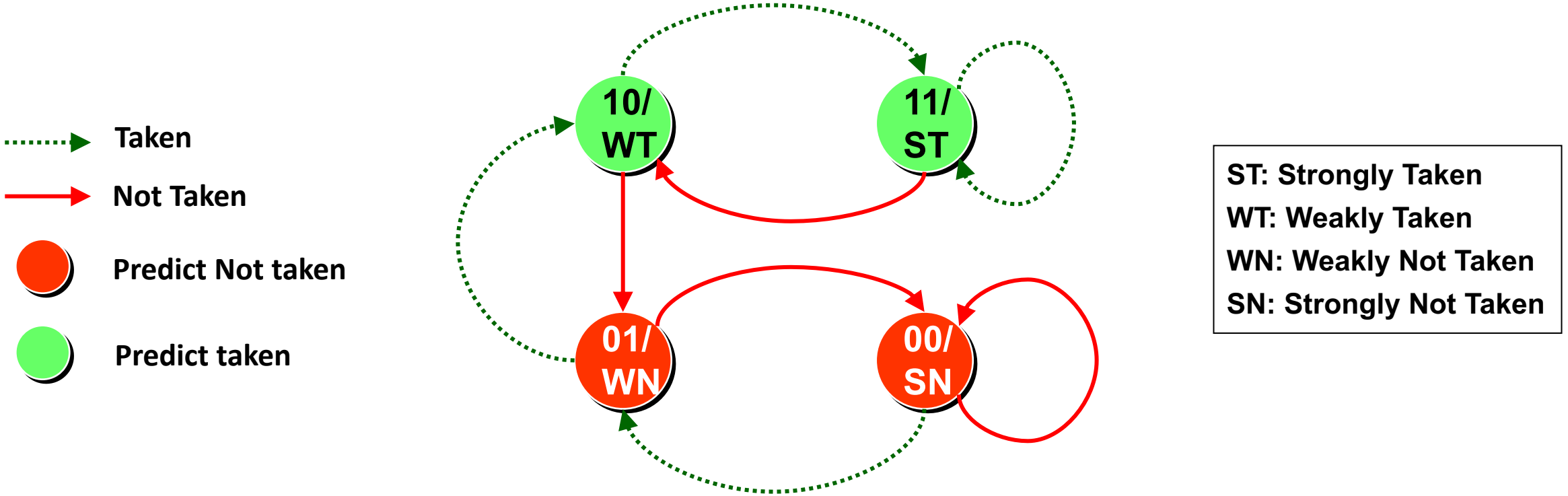


60% accuracy

Pretty bad.. This is an easy loop!

How can we fix this?

2-bit Saturating Up/Down Counter Predictor

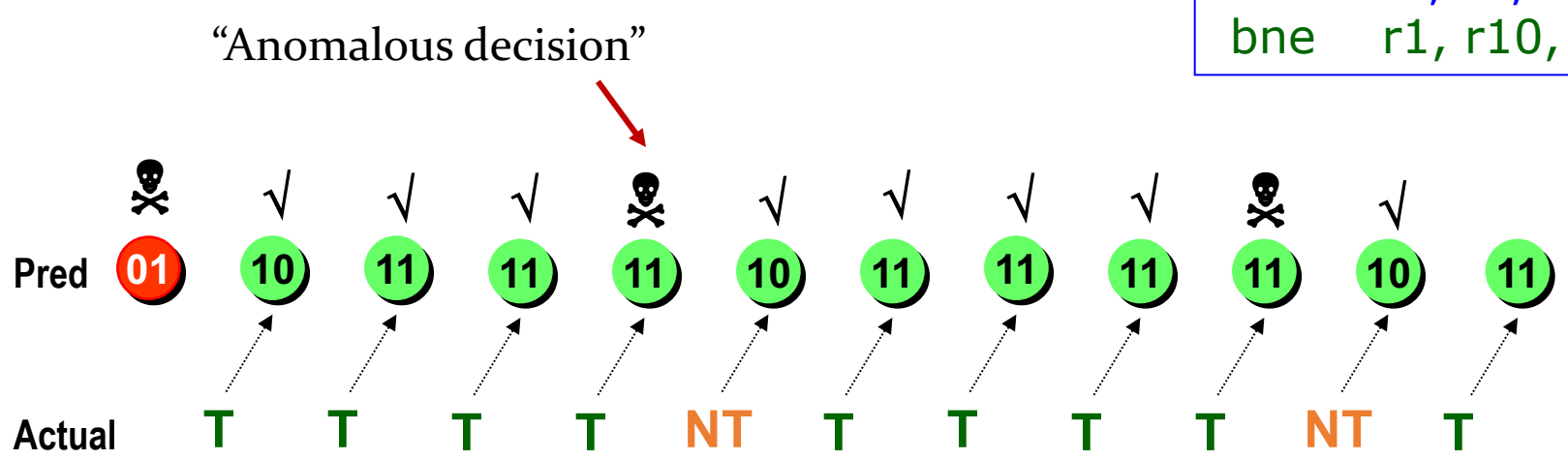


MSB tells us direction (taken/not taken)
LSB servers as hysteresis or inertia

Example using 2-bit up/down counter

```
for (i=0; i<4; i++) {
        ....
}
```

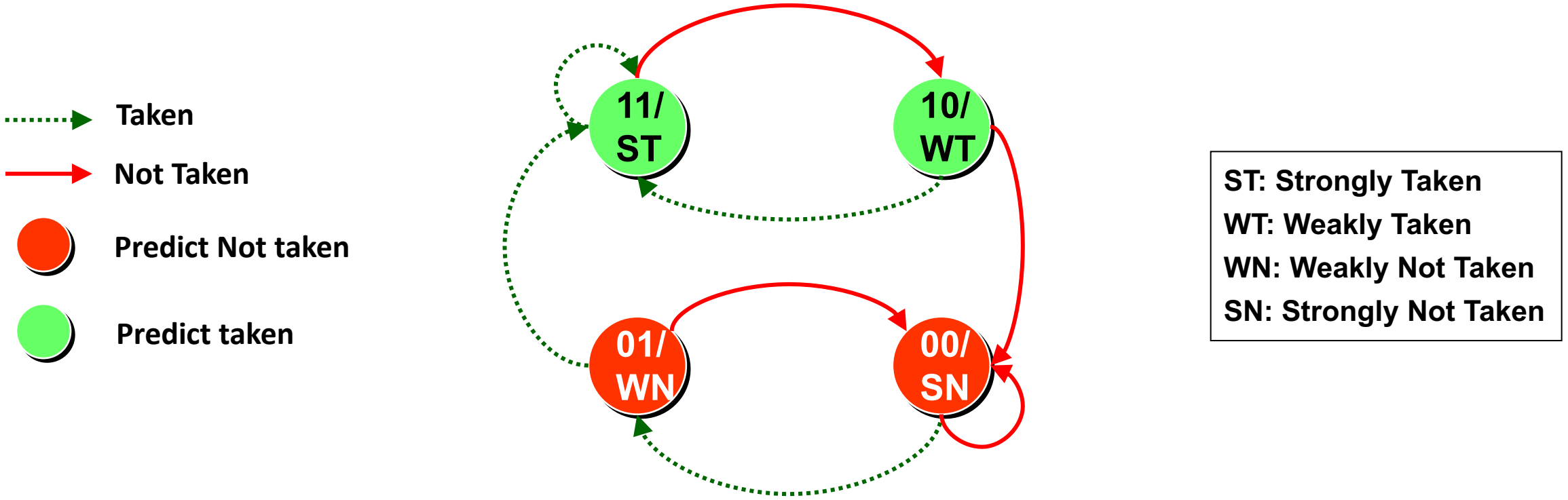
```
 addi   r10, r0, 4
 addi   r1,   r1, r0
L1:
 … …
 addi   r1, r1,   1
 bne    r1, r10, L1
```



80% accuracy

What's happening compared to 1b?

2-bit Counter Predictor (Another Scheme)



Q: What's the difference between the two?
Most ended up using saturating counters

The “Smith Predictor” (1981)

General solution:
- m determines # entries (FSMs)
- k is the number of bits/entry

Total size of the predictor is: k2^m

Rather than just using single 2b counter
for everything, create a table of counters/FSMs

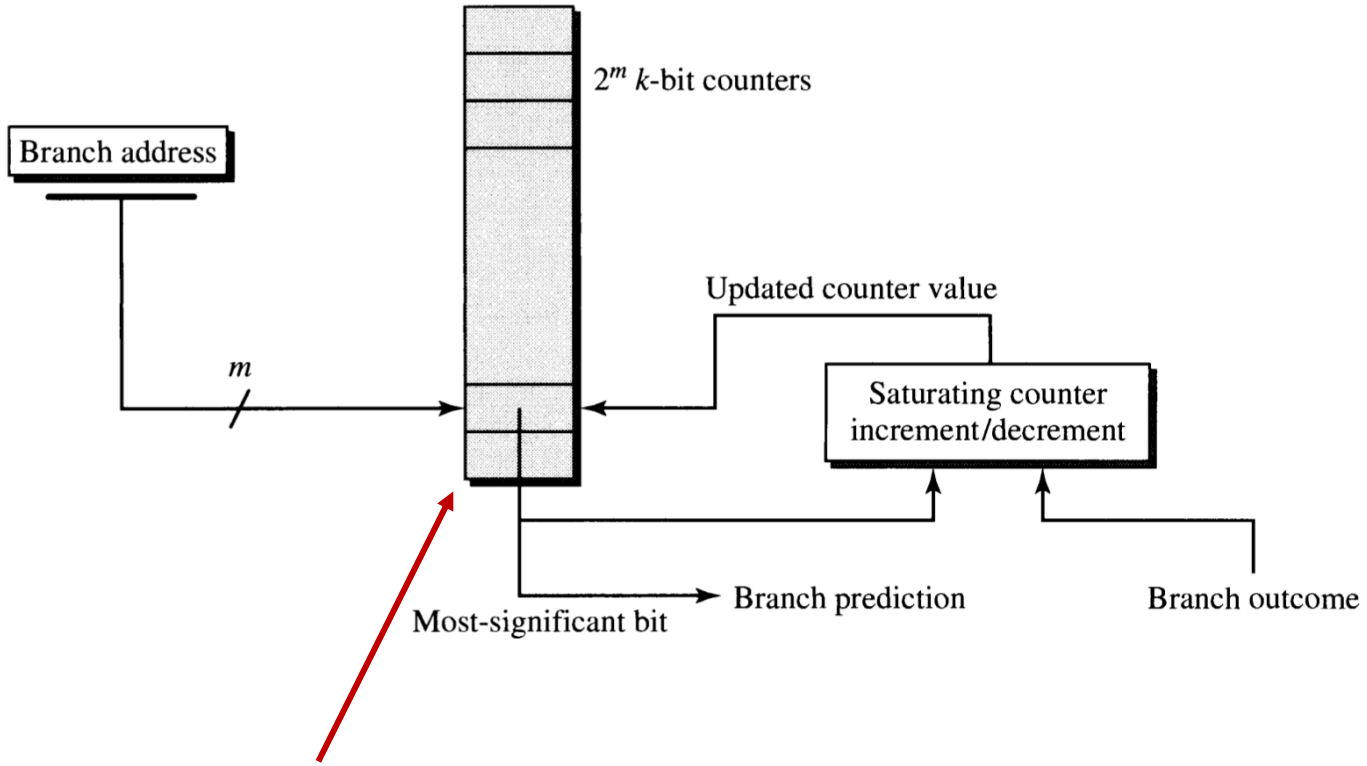
Index this table using the branch address (PC)

Option 1: Use entire branch address
Too expensive!

Option 2: Use 1 entry/branch
Too expensive!

Option 3: Hash and hope collisions
aren’t too bad

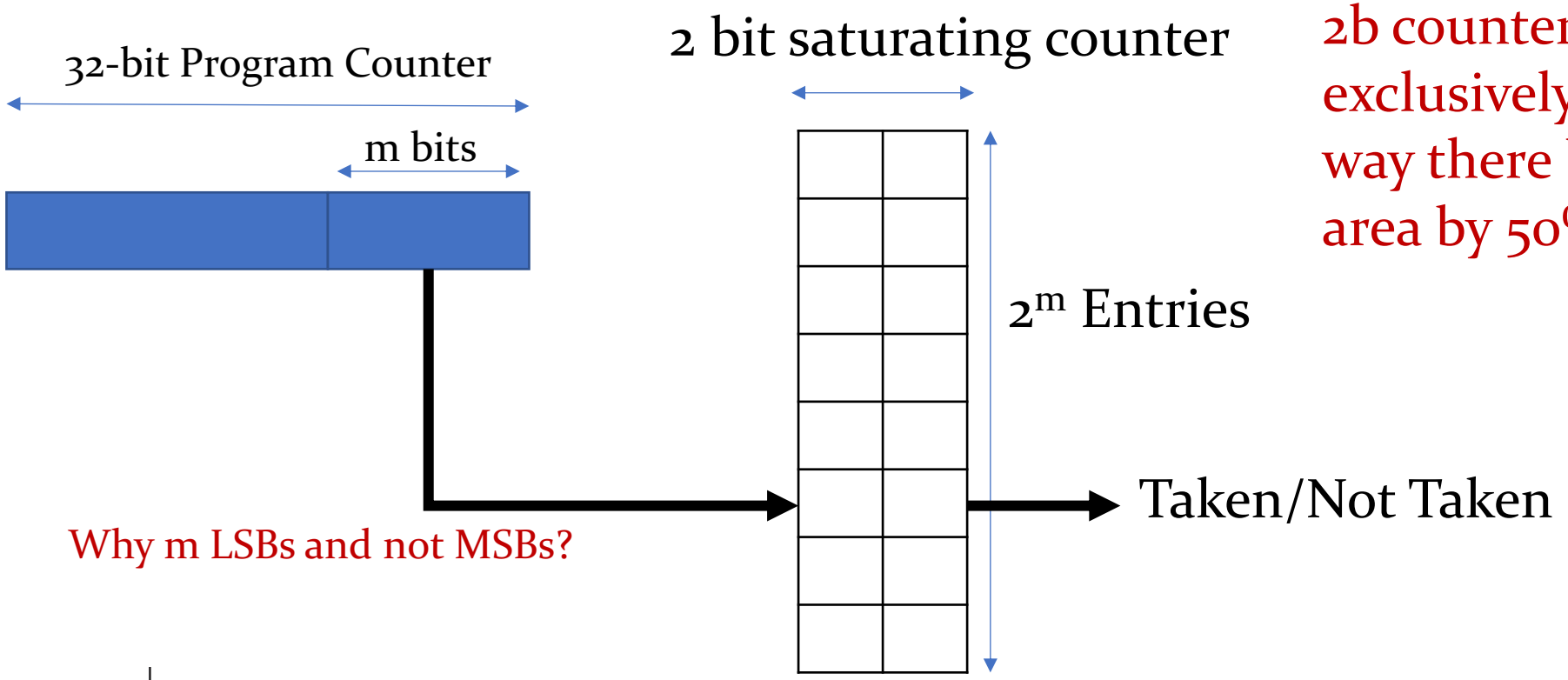
Option 3 is used today. Some uses actual hash functions some take LSBs



Called the Pattern History Table (PHT)

Prediction History Table

Hardware structure that is looked-up during fetch stage to determine Taken/Not Taken



2b counters used almost exclusively. Get most of the way there but 3b increases area by 50%

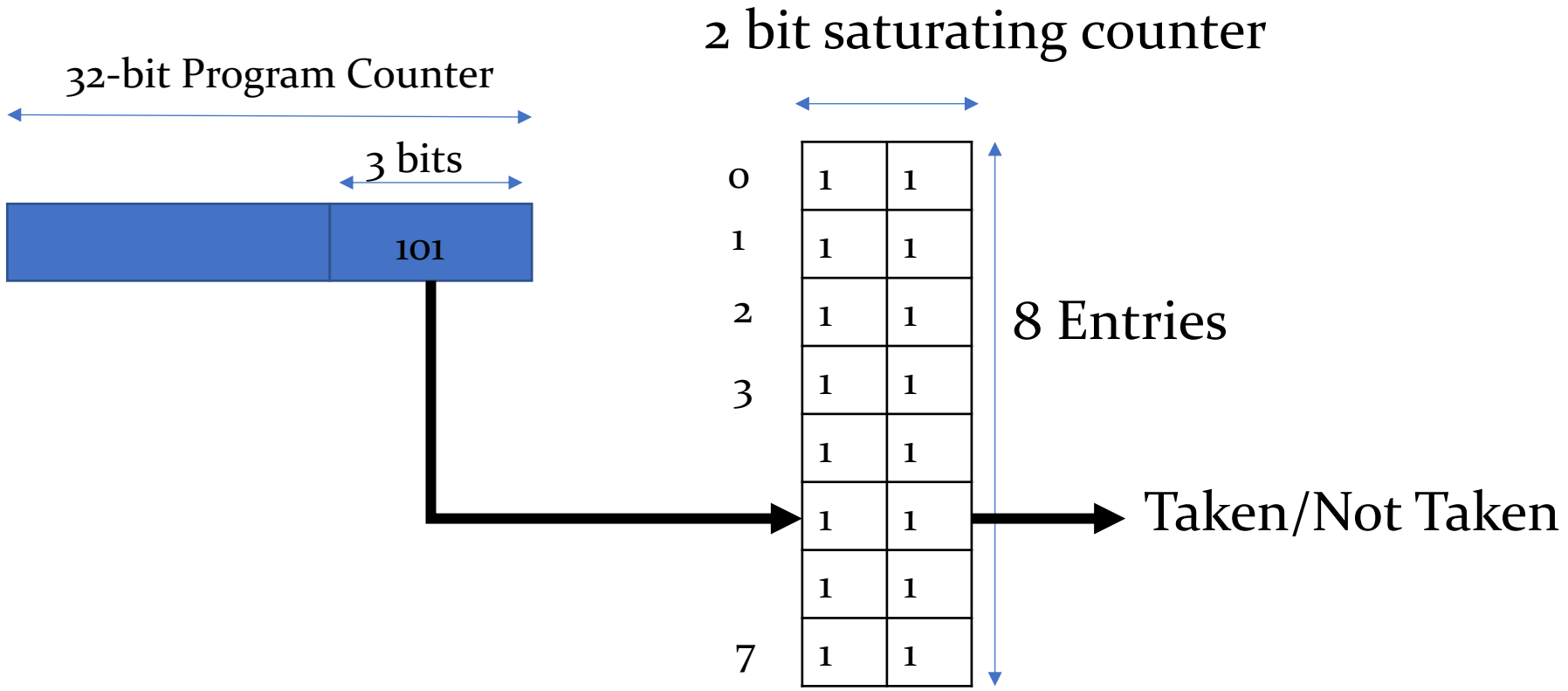
Example

Consider a m=3 and assume that all counters start in 11 state

- 8 2-bit counters indexed from 0…7

Branch Trace <PC, T/NT>

0x00000000 N
0x00000002 N
0x00000007 T
0x00000000 T
0x00000002 N
0x00000007 N
0x00000000 T
0x00000002 N
0x00000007 N

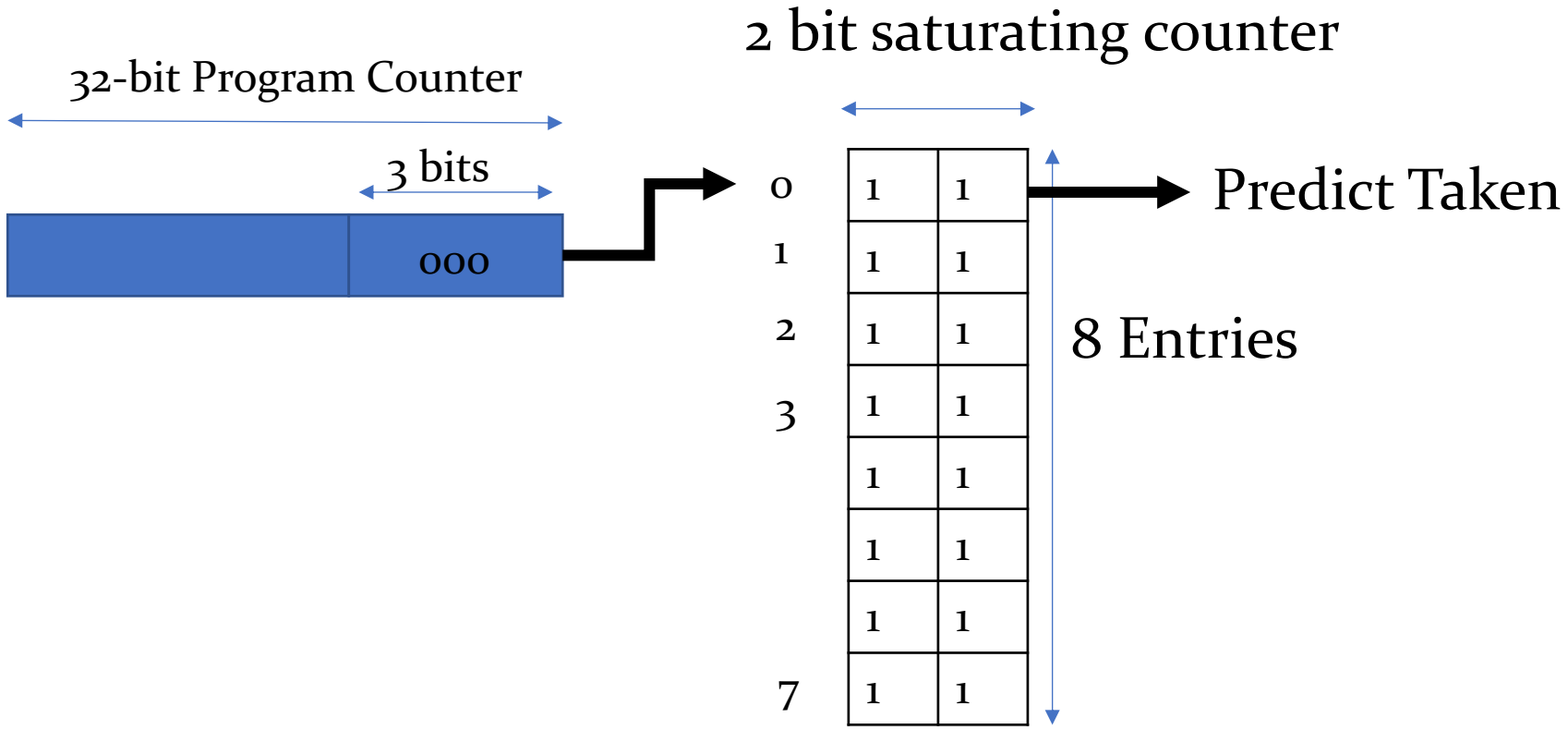


Branch 1

Index into table and predict branch outcome

Branch Trace <PC, T/NT>

0x00000000 N
0x00000002 N
0x00000007 T
0x00000000 T
0x00000002 N
0x00000007 N
0x00000000 T
0x00000002 N
0x00000007 N

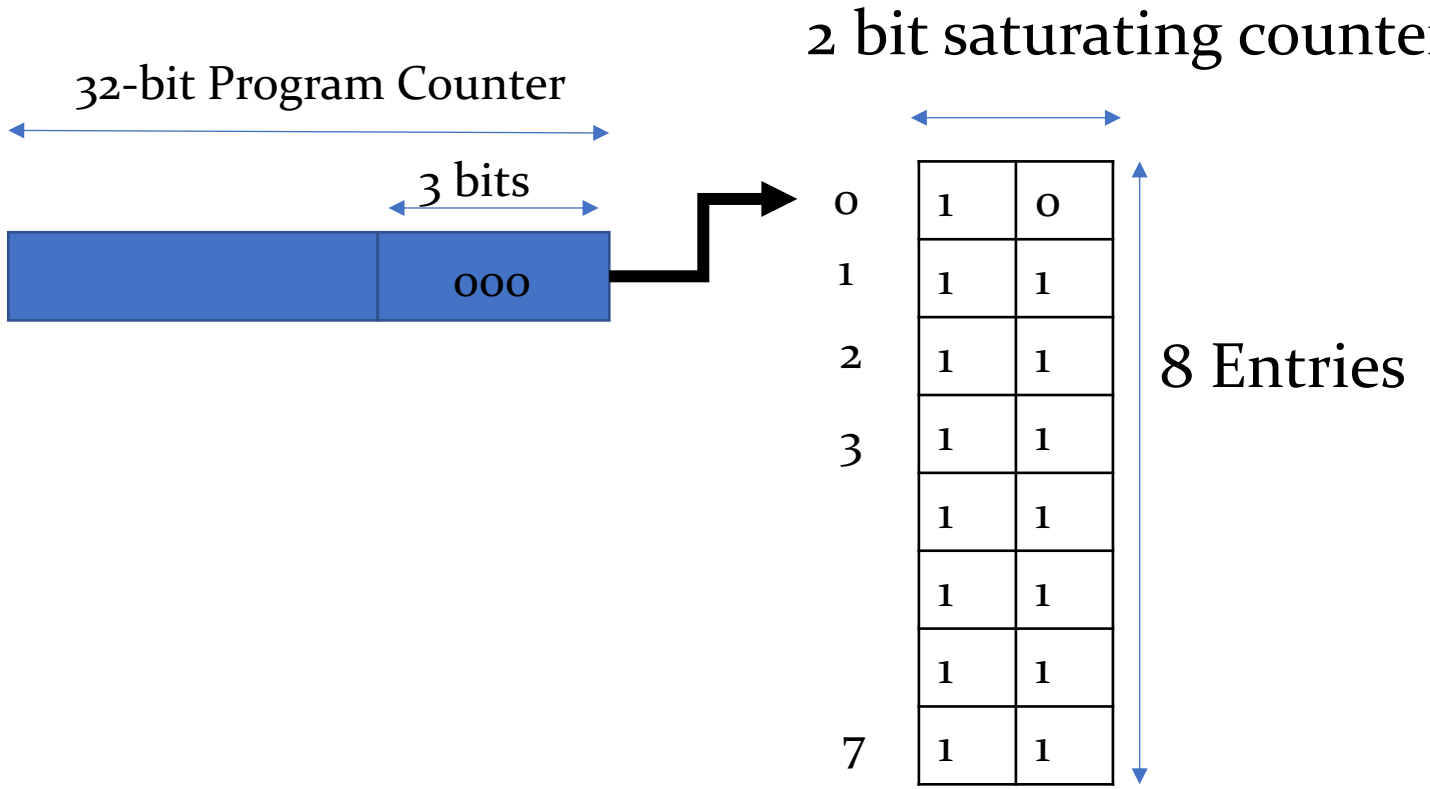


Branch 1

Update counter once actual branch outcome is known

Branch Trace <PC, T/NT>

0x00000000 N
0x00000002 N
0x00000007 T
0x00000000 T
0x00000002 N
0x00000007 N
0x00000000 T
0x00000002 N
0x00000007 N

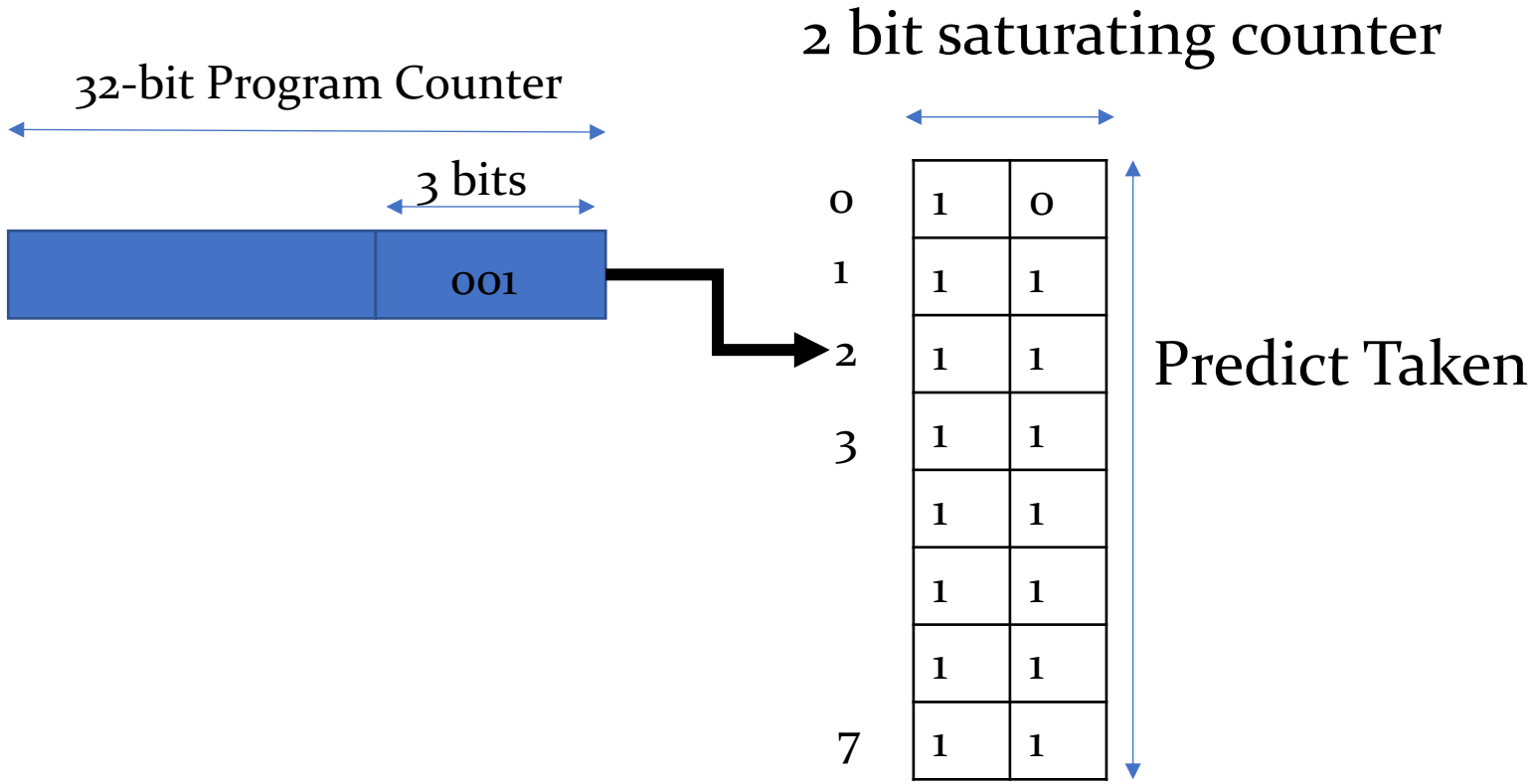


Branch 2

Index into table and predict

Branch Trace <PC, T/NT>

0x00000000 N
0x00000002 N
0x00000007 T
0x00000000 T
0x00000002 N
0x00000007 N
0x00000000 T
0x00000002 N
0x00000007 N

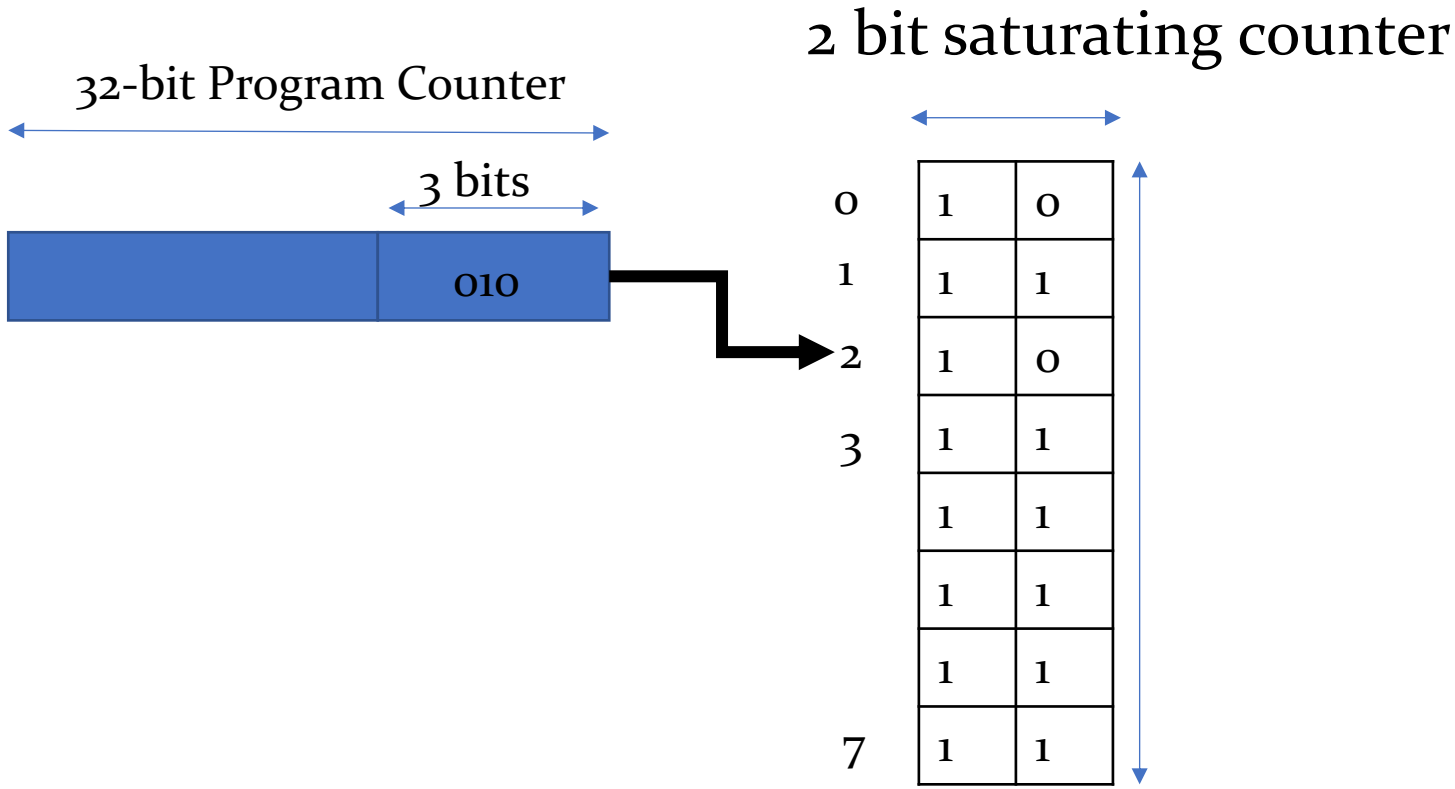


Branch 2

Update counter

Branch Trace <PC, T/NT>

0x00000000 N
0x00000002 N
0x00000007 T
0x00000000 T
0x00000002 N
0x00000007 N
0x00000000 T
0x00000002 N
0x00000007 N

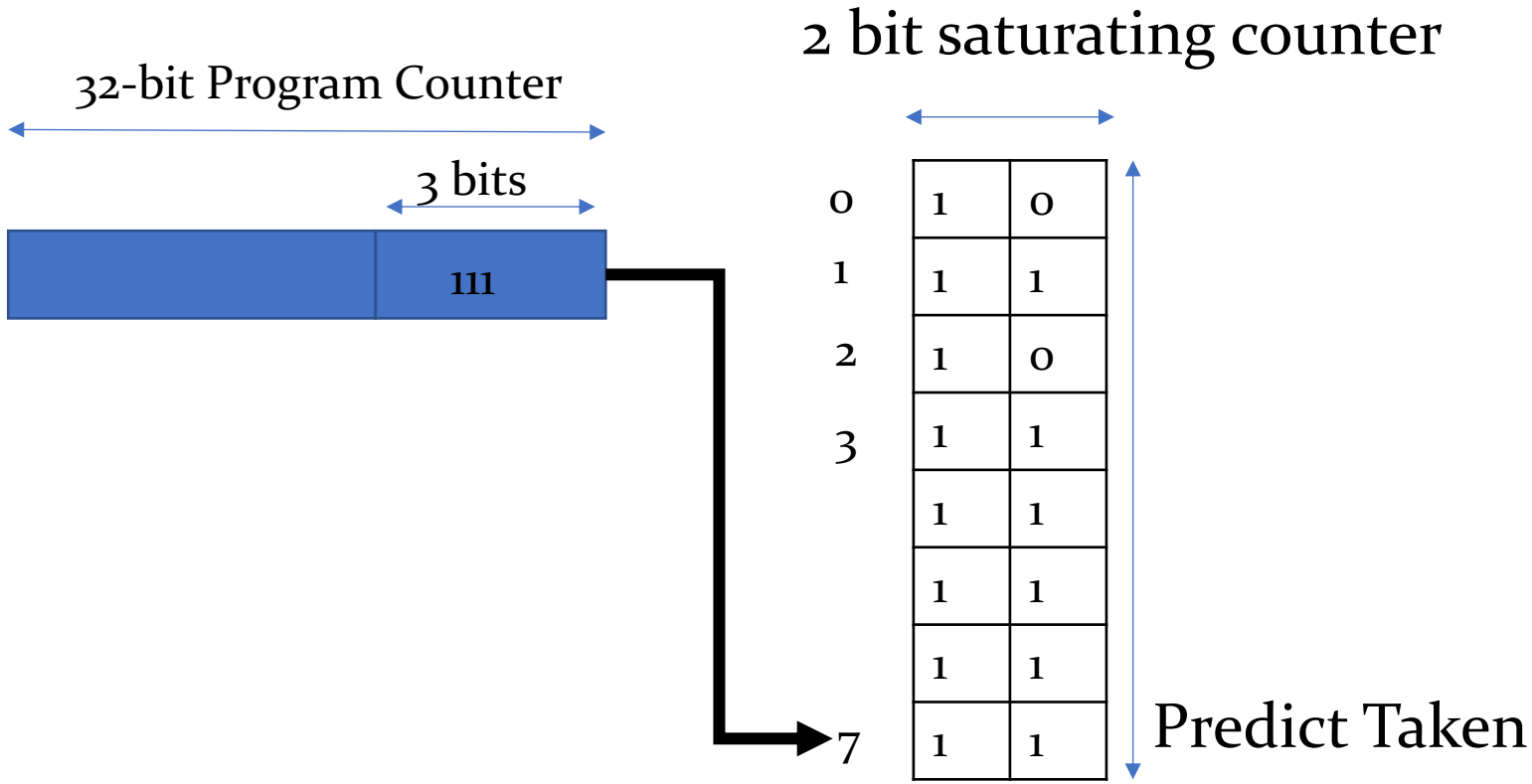


Branch 3

Index into table and predict and update counter

Branch Trace <PC, T/NT>

0x00000000 N
0x00000002 N
0x00000007 T
0x00000000 T
0x00000002 N
0x00000007 N
0x00000000 T
0x00000002 N
0x00000007 N



Two –Level Prediction Tables

More advanced solution: Two–Level adaptive branch prediction

"Two-level" (Yeh and Patt, 91, 92, 93)
"Correlation branch prediction" (Pan, 92)

Key idea: Different branches tend to impact each other. We can make better predictions by using information from other recent branches (both locally and globally).

Smith's predictor – Captures best branch decision

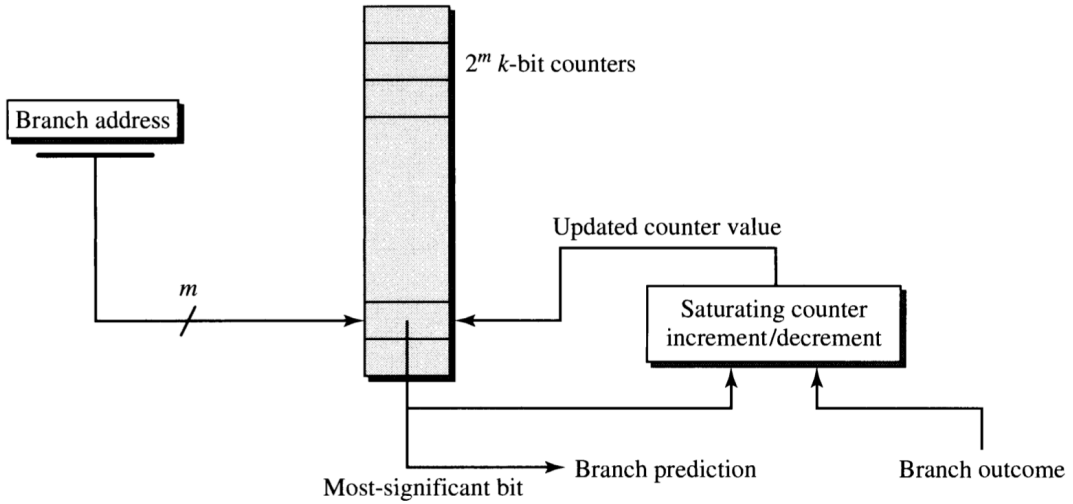
Two-level predictor – Also captures recent decision history

Tracking global branch decisions

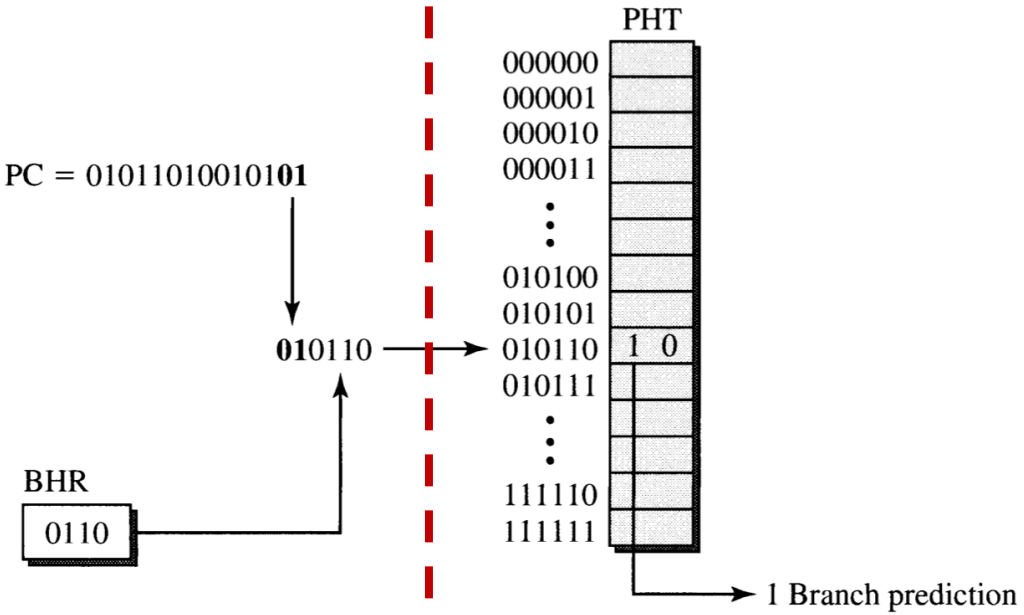
- Branch history register (BHR)
  - Use a single register to keep track of all recent branch outcomes
  - Typically use a shift register
    - When branch resolves, shift in 1 for taken, 0 for not taken
  - Use the BHR or BHR+PC to index PHT
    - Remember, PHT is just a bunch of FSMs
    - The BHR provides a different way of picking PHT entry

Example of a two-level predictor

Smith's



Two-level predictor



Level 1
Track recent outcomes in BHR
Uses h bits, in this case 4
Implemented as Queue (LIFO) right->left

Level 2
Pattern history table (just like before)
Each entry is 2b saturating counter
Still updates on each branch outcome

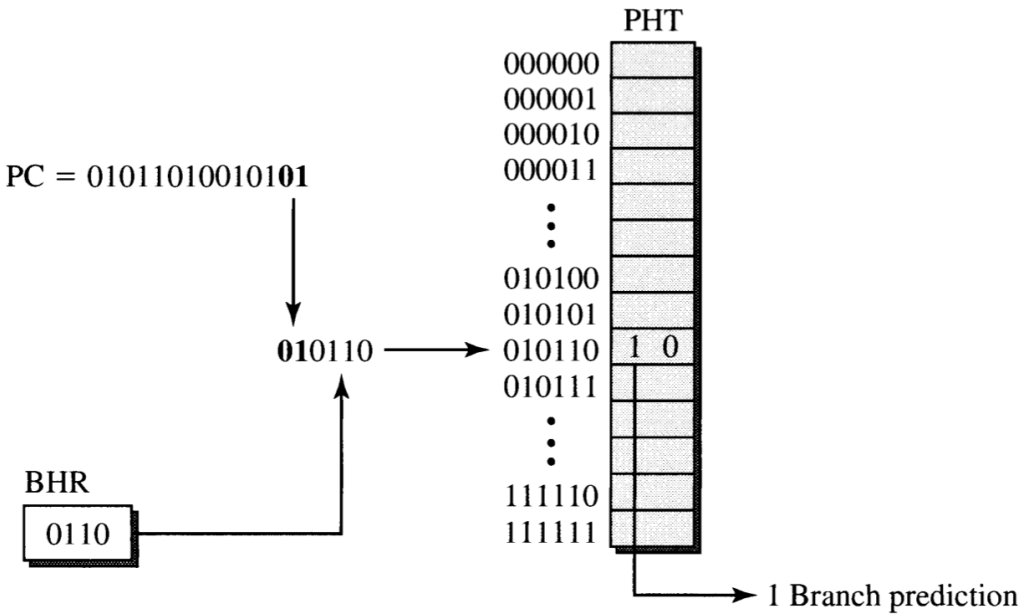
Indexing tradeoffs

How do we decide the PHT address?

More PC bits implies fewer collisions
More BHR bits implies more correlation

Architects job is to balance these competing tradeoffs

What does Smith's predictor favor?



Intuition

```
If (aa==2)
        aa=0
If (bb==2)
        bb=0
If (aa!=bb){
        <do something>
}
```

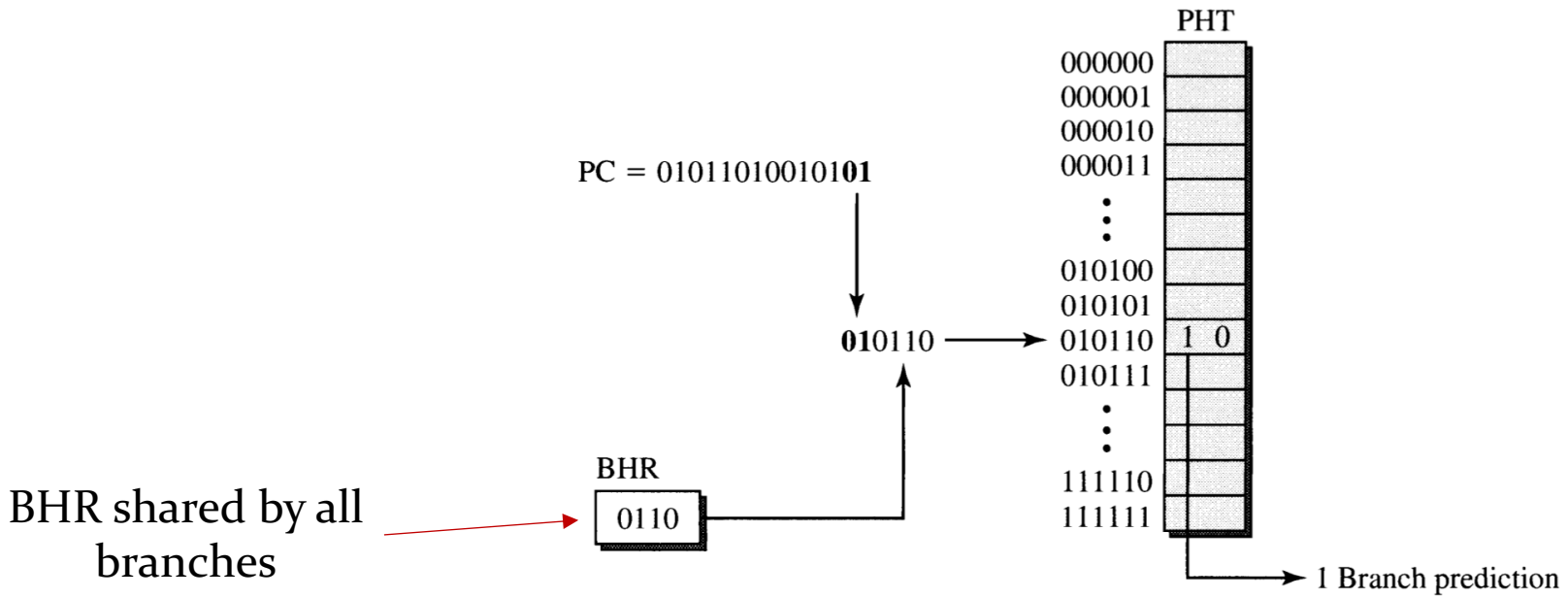
```
X = 0
If (someCondition)
        x=3
If (someOtherCondition)
        y+=19
If (x <= 0){
        <do something>
}
```

High correlation

Low correlation

BHR lets us capture these correlations!

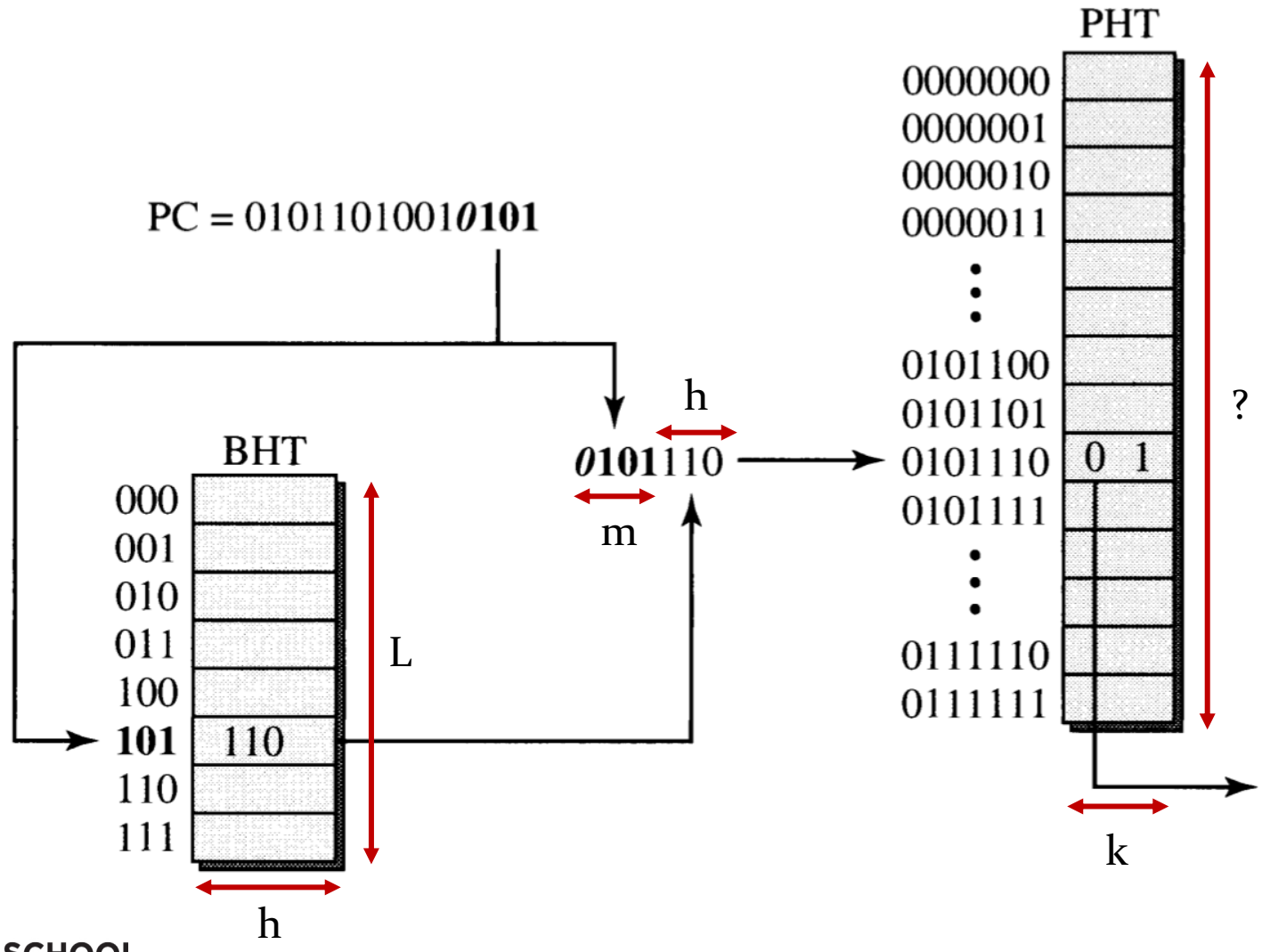
“Global” two-level predictor



“Local-history” two-level predictor

- Global BHR tracks history of all recent branches
- Local tracks outcomes of last several encounters of current branch
  - Generalization of the Global two-level predictor
  - Extend the single BHR to an array of BRHs
  - Called the Branch History Table (BHT)
  - Index the BHT using the LSBs of the PC
    - Each BHR tracks the branch history for that branch
    - Think of it as:
      - BHT index = look up the history pattern for this branch (PC)
      - BHT entry = the last few times this branch occurred, here’s what happened
      - PHT index = Given I saw that pattern for that branch, what should I do?
      - PHT entry = I’ll know what to do looking at the MSB of the 2b counter!

Local-History Two-Level Predictor



How big is this table?

Lh + k2^(m+h)

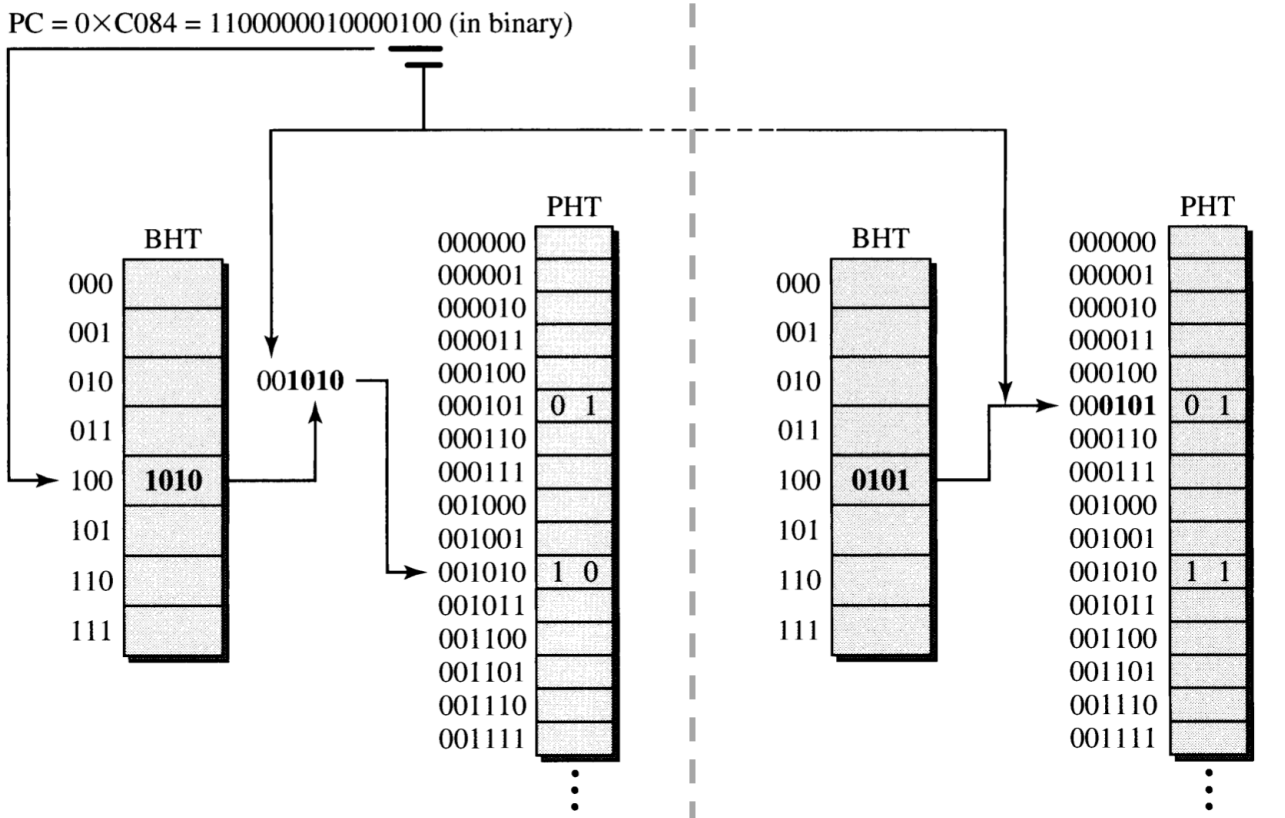
8*3 + 2*2^(7)
= 24+256
=280bits

Example 1: T, N, T, N, …

- Simple pattern, is it simple to predict?
- What happens with 1b predictor initialized to not taken?
  - 0% accuracy, wrong every time!
- What about 2b saturating predictor initialized to Weakly T?
  - 50%
  - What if it's initialized to Strongly NT?

- What about a Locally-History Two-level predictor?

Example 1: Local-History 2-level predictor T, NT, T, NT



First
Guess: taken
Result: taken

Second
Guess: not taken
Result: not taken

What's going on here?

1) Always point to same BHR
2) PHT changes based on local branch history (key!)
3) Different FSMs tell us what to do depending on what we last saw

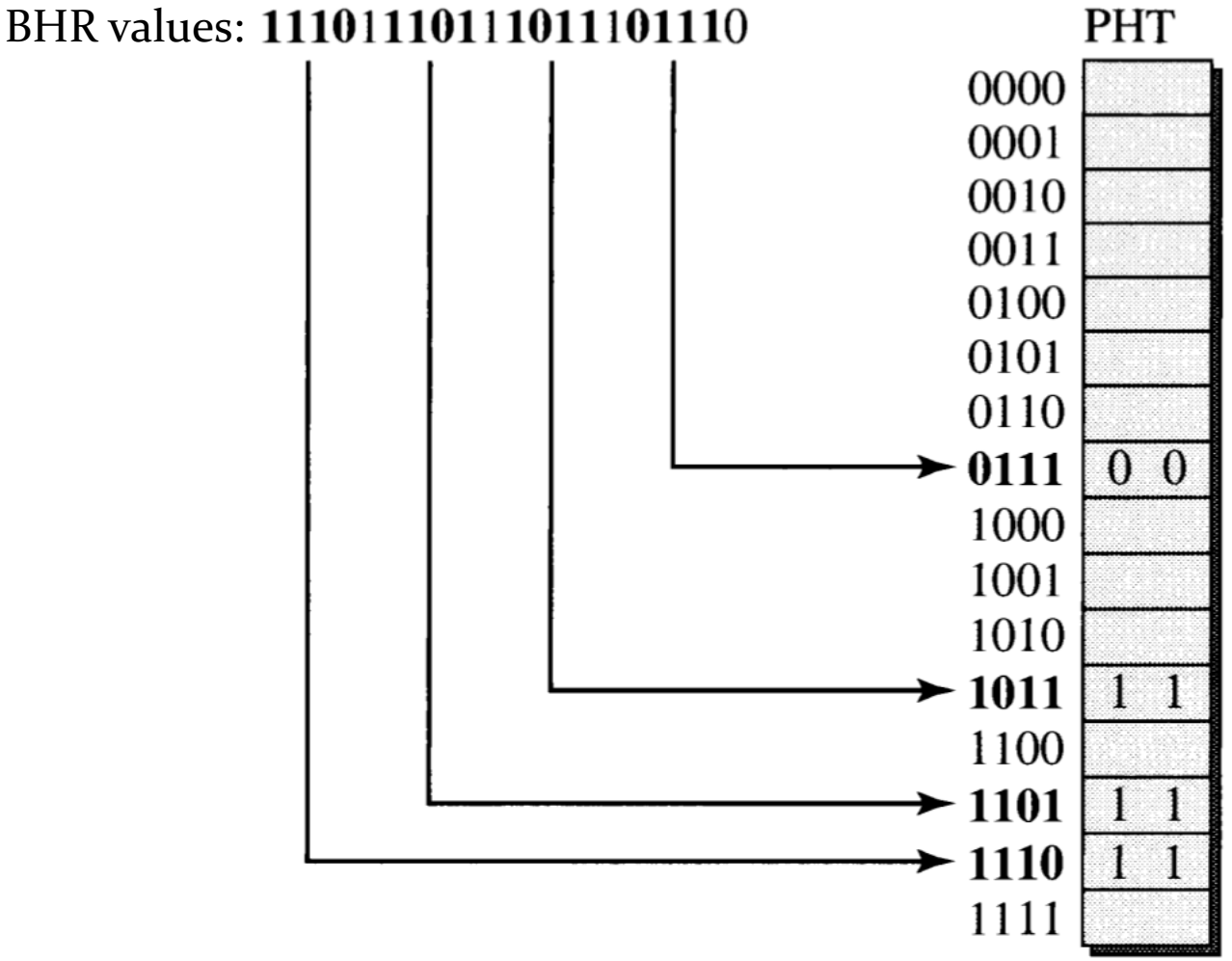
Example 2: 1110111011101

Say we only use BHR to index PHT (no PC bits)

The branch pattern repeats every four branches

If we have a PHT size 16, what happens?

We can learn this pattern exactly!



Taxonomy of branch predictors

Three types of Branch History Tables (BHTs)

1) Global (G): BHT is a single BHR shared by all branches
2) Set (S): Index BHR using hash function
3) Per address (P): Index BHR using PC address

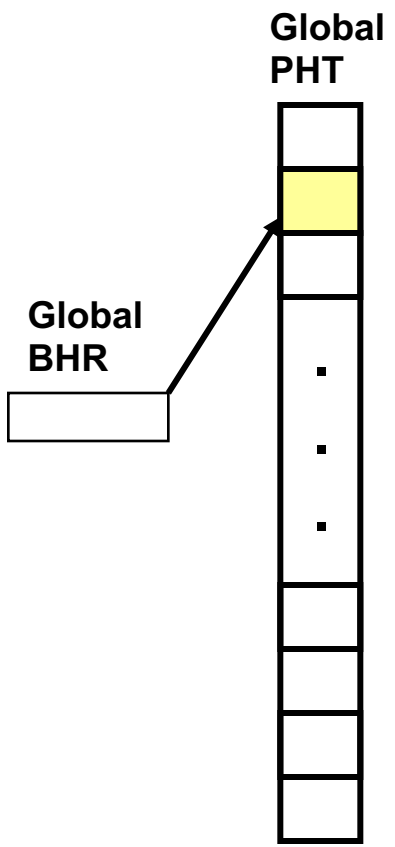
Three types of Pattern History Tables (PHTs)

1) Global (g): 1 shared PHT, indexed by BHR
2) Set (s): PHT selected with branch hash, FSM selected w BHR
3) Per address (p): PHT selected with branch addr, FSM w BHR

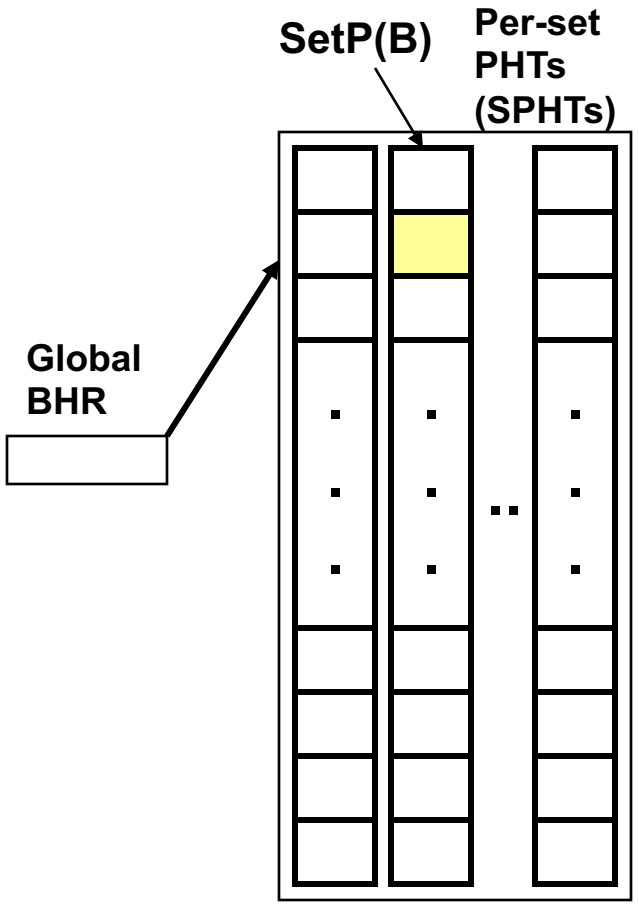
All 9 combos specified by <X>A<y> where X={G,S,P} and Y={g,s,p}

Global History Schemes

GAg

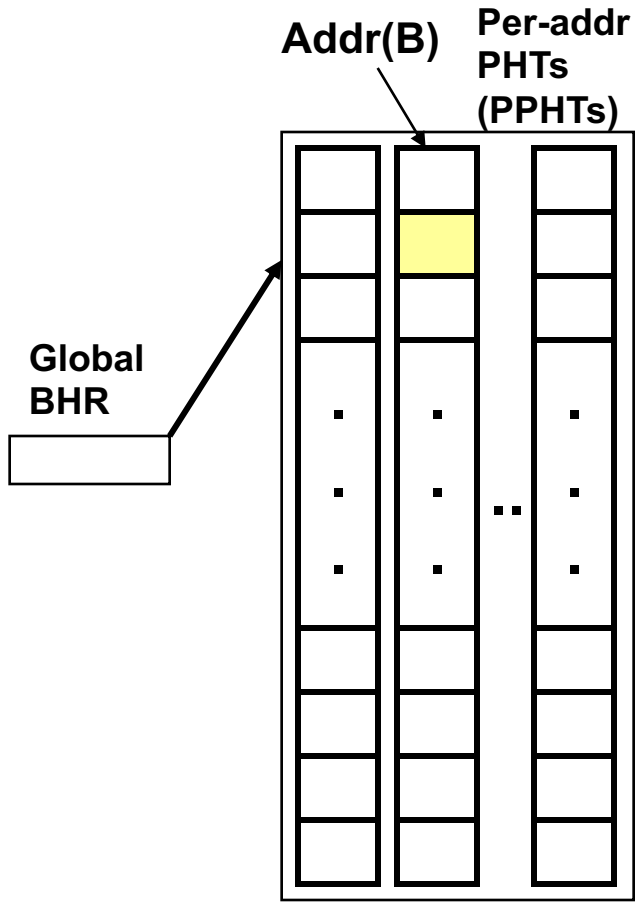


GAs

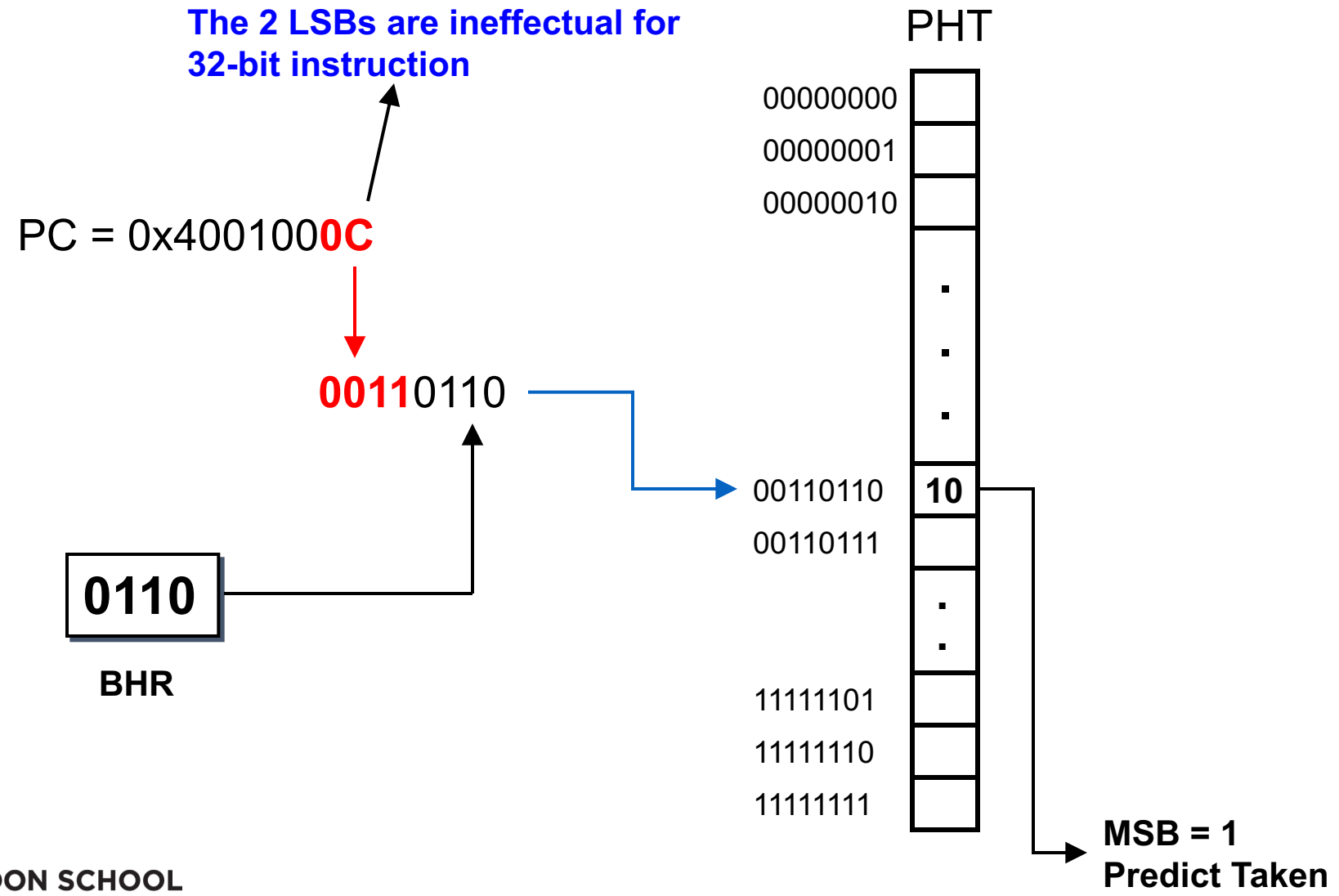


Set can be determined by branch opcode, compiler classification, or branch PC address.

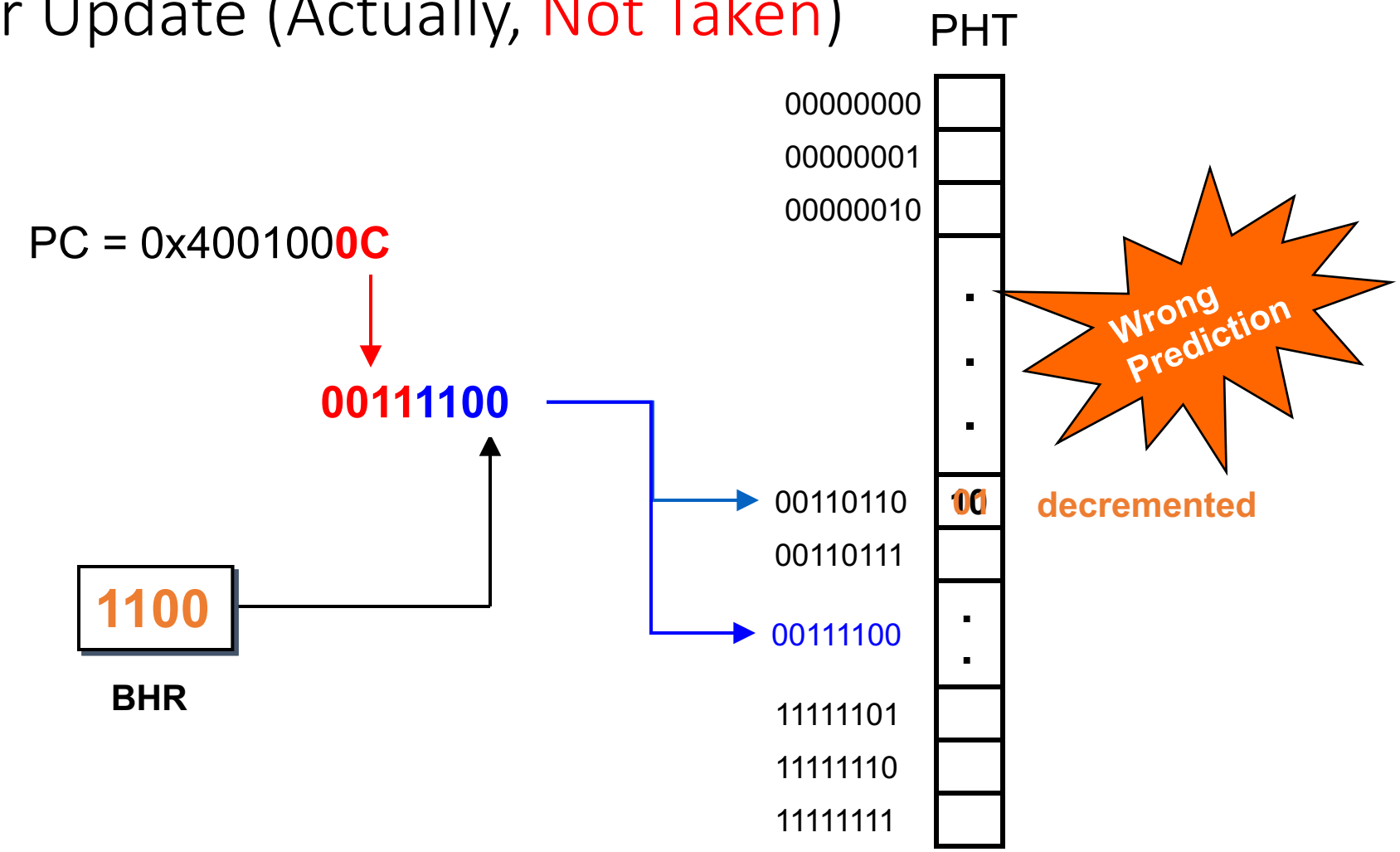
GAp



GAs Two-Level Branch Prediction

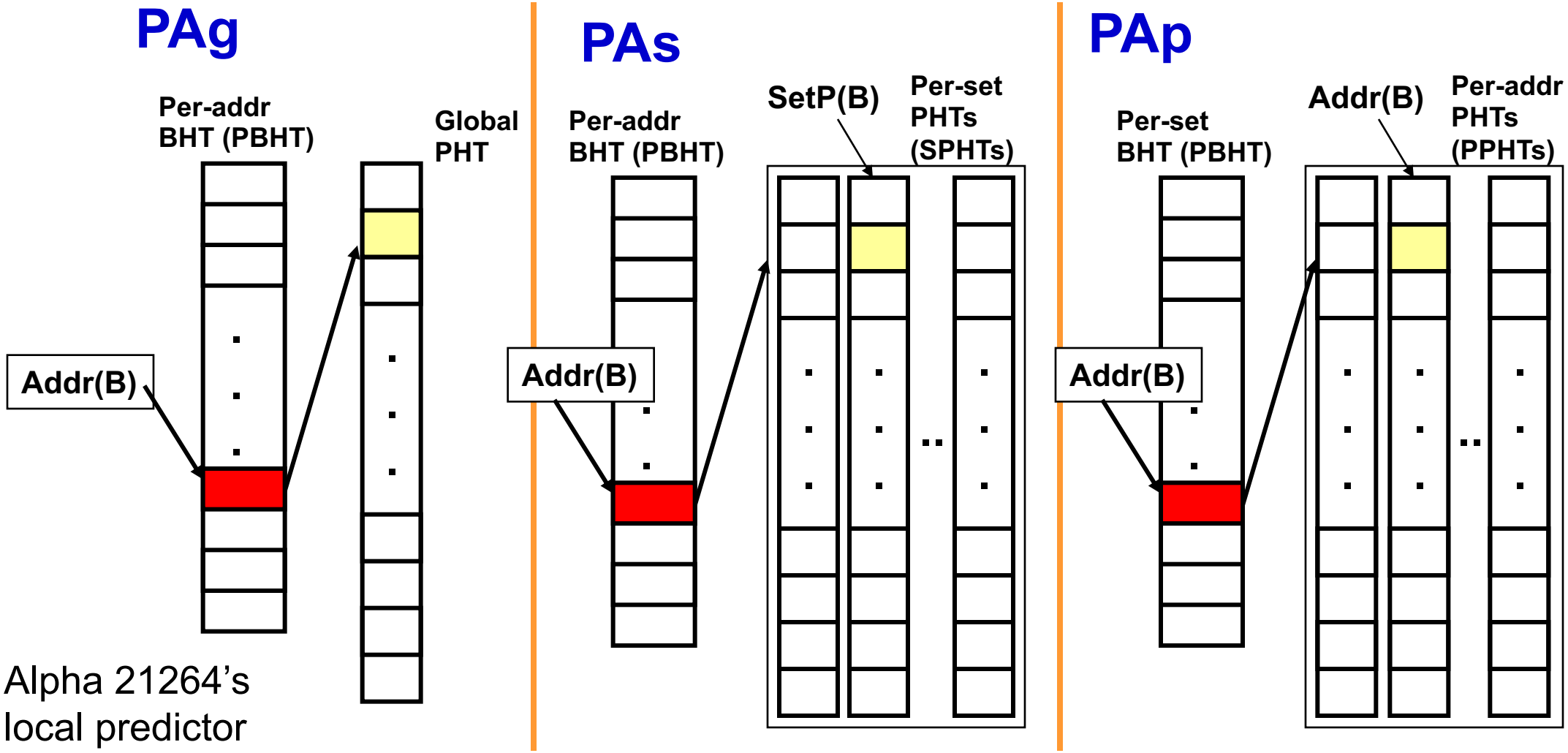


Predictor Update (Actually, Not Taken)



- Update Predictor after branch is resolved

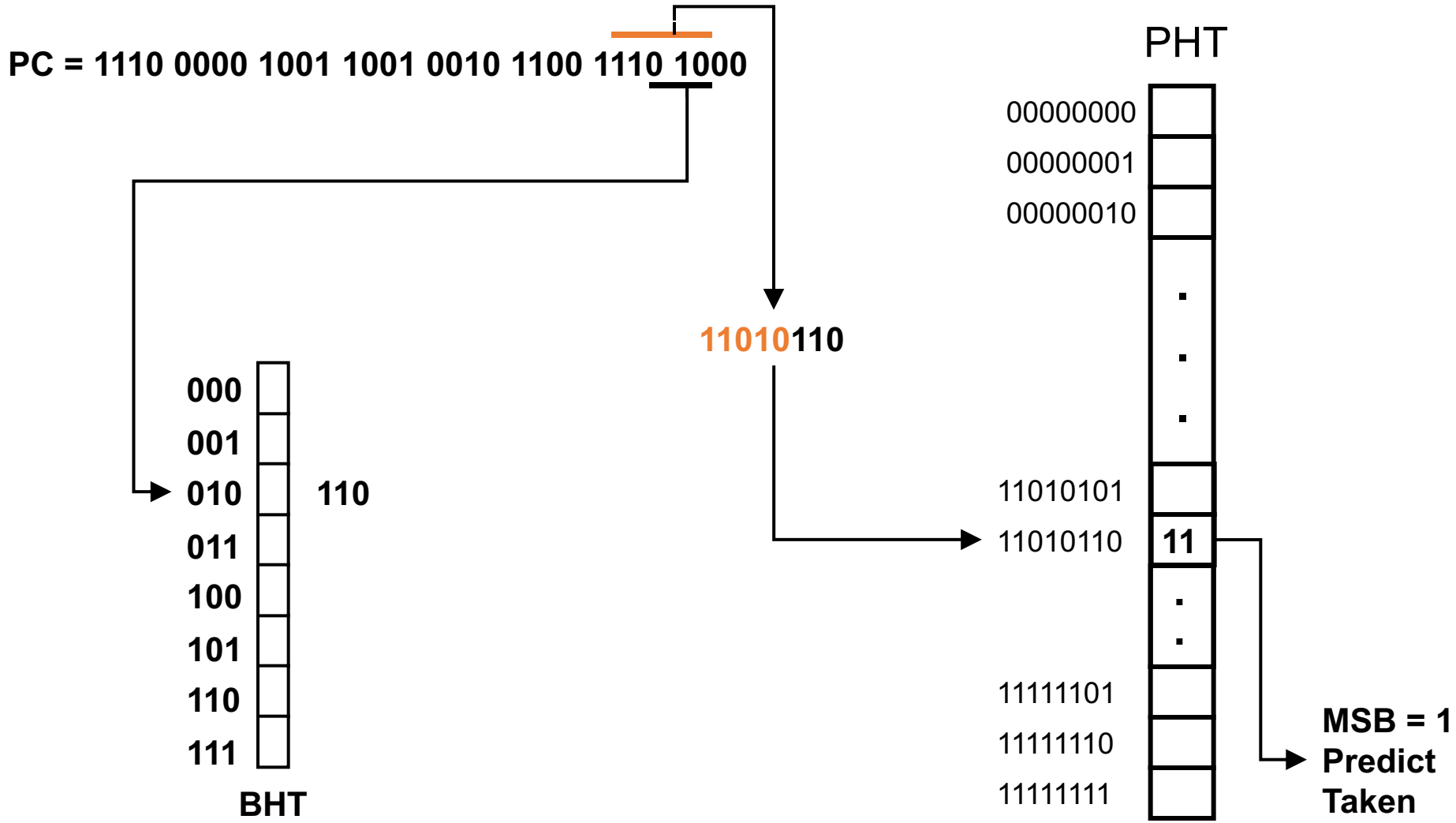
Per-Address History Schemes



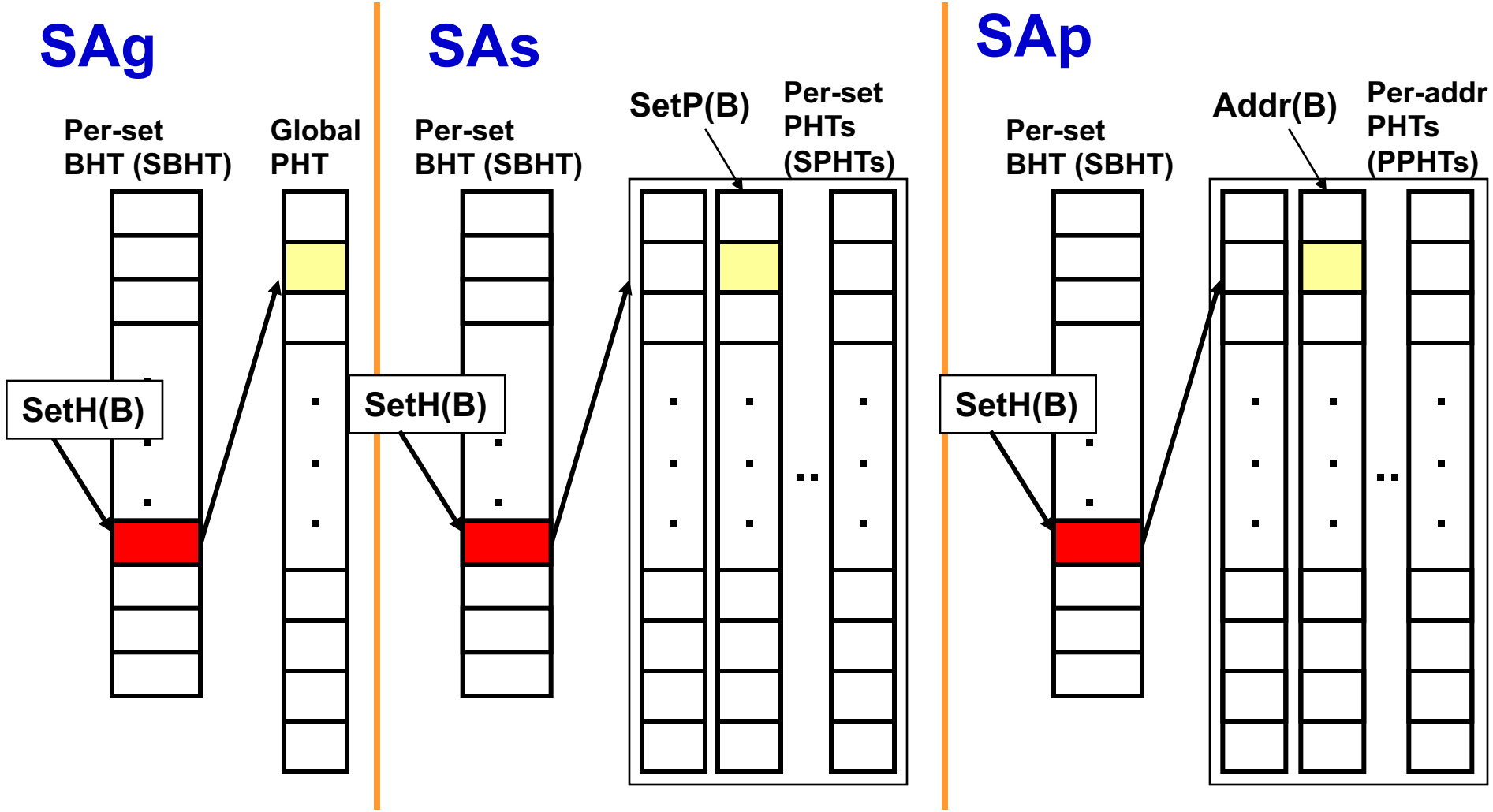
Alpha 21264's local predictor

Ex: P6, Itanium

PAp Two-Level Branch Predictor



Per-Set History Schemes



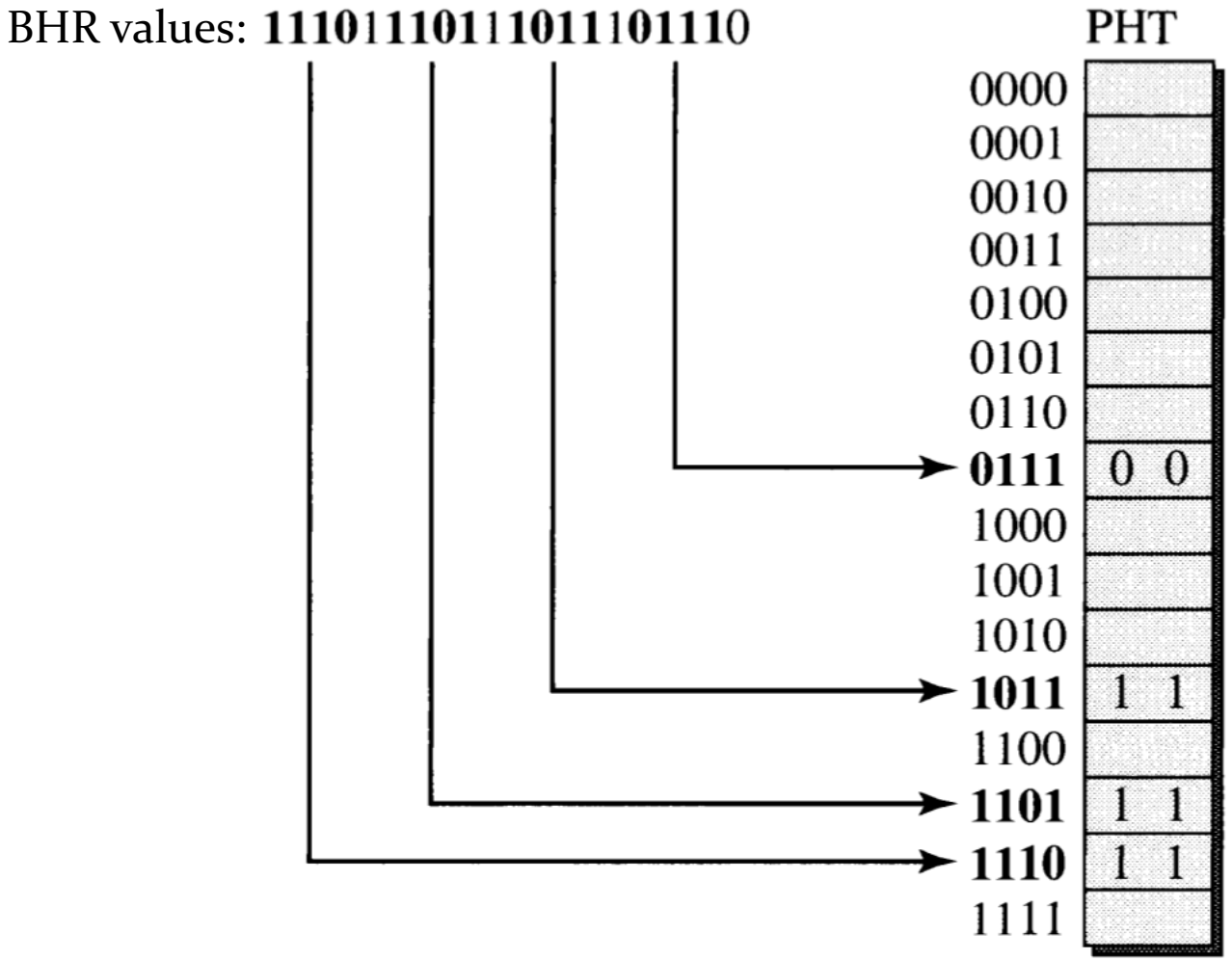
NYU TANDON SCHOOL OF ENGINEERING

More tradeoffs

- How to index PHT?
  - Using more BHR bits enables better correlation
  - Fewer PC bits causes more PHT collisions/aliasing

- BHR-heavy indexing performs well but:
  - Causes aliasing
  - Results in poor PHT utilization
    - Why?
    - Normally only have a few common branch patterns, rest unused

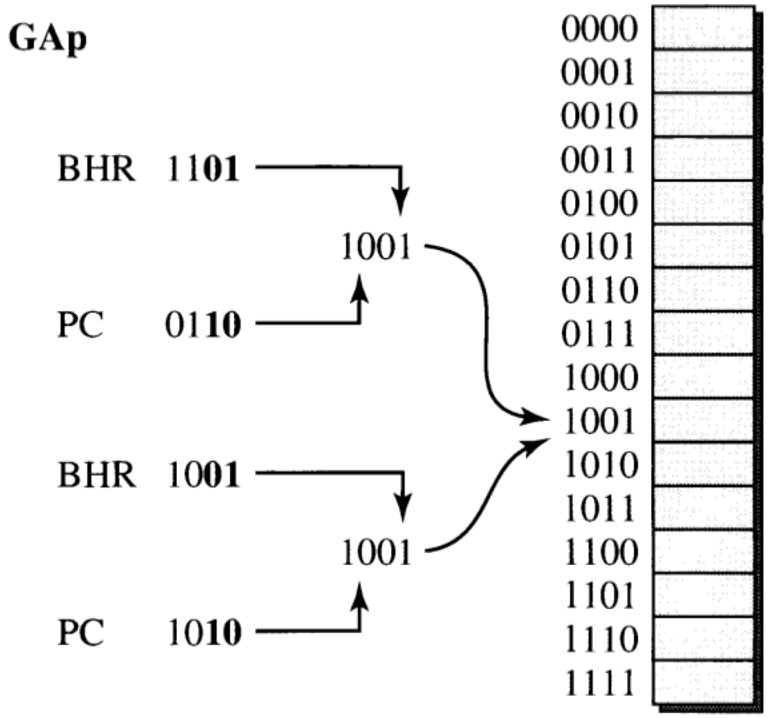
Recall example



Only using 4 of 16 slots..
Poor utilization!
We can do better!

Q: What happens if another branch aliases to the same PHT entries?
Everything works!
"Neutral Interference"

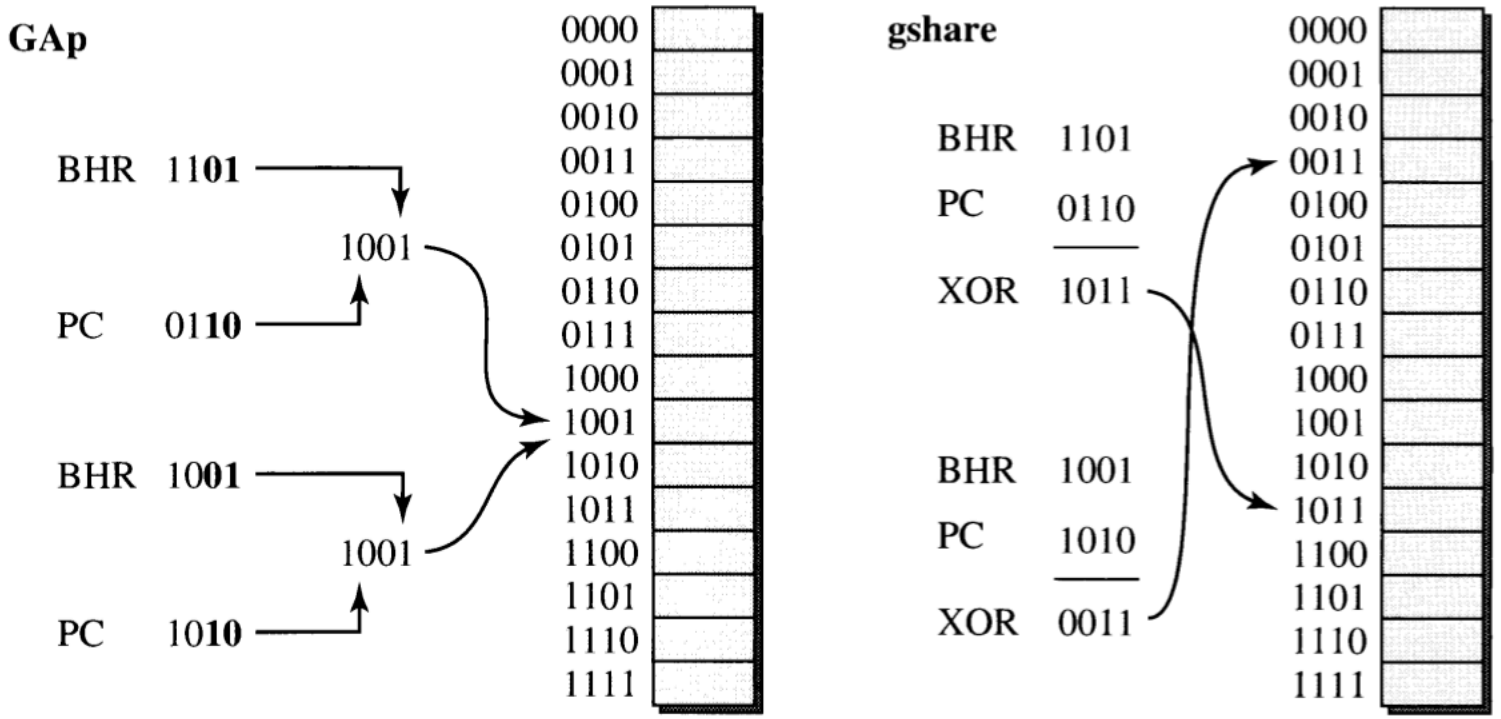
Index-sharing: the gshare predictor



Index-sharing: the gshare predictor

Pros
+ Better utilize PHT
+ Resolves conflicts

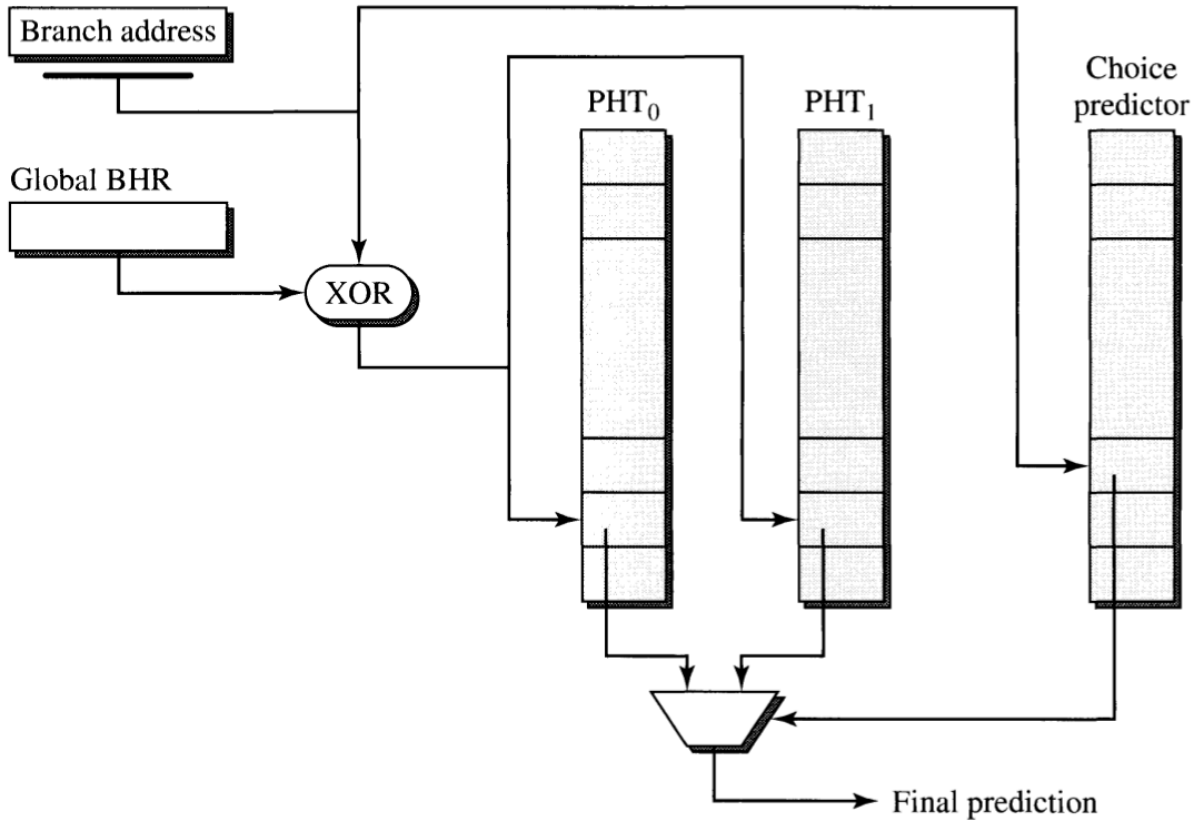
Cons
- Can cause new conflicts
- Minor increase to critical path



Bi-Mode predictor

- Most branches are biased toward T/NT
  - PHTs learn bias
  - Choice predictor learns branch bias
- Turns negative conflicts into neutral
  - If collide in the same PHT,
    Choice made it happen, so probably good

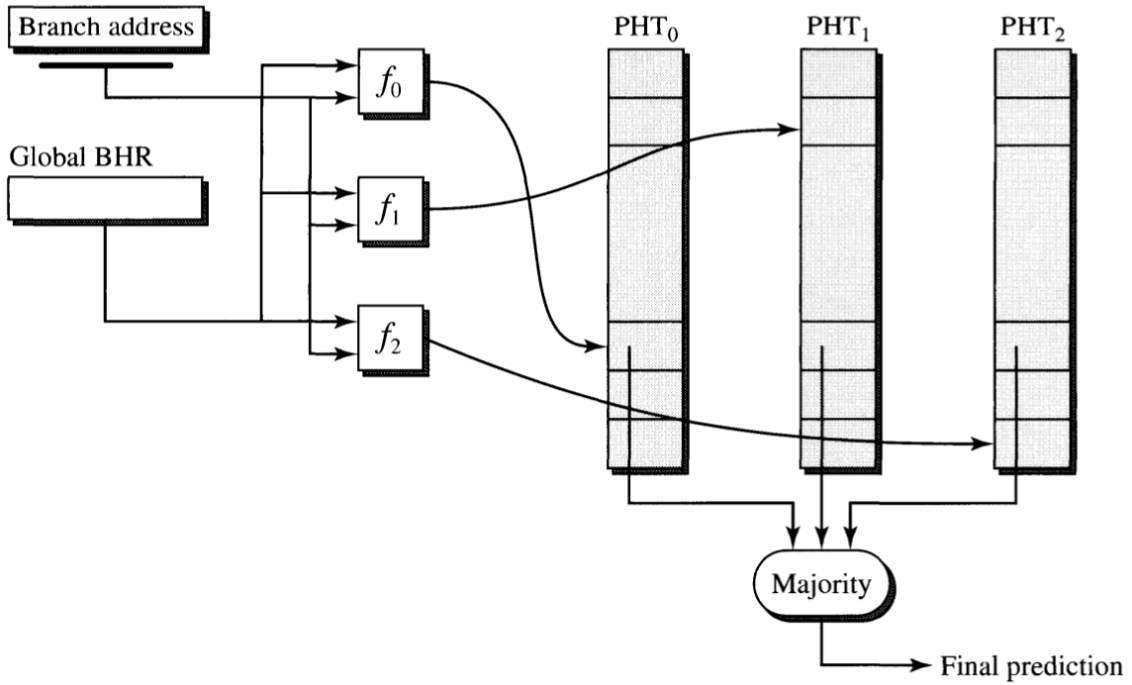
- Updating
  - Selected PHT always updated
  - Not selected bank not updated
  - Choice predictor
    - Always updated using branch direction
    - Unless Choice Pred. outcome was wrong but selected PHT's guess was correct



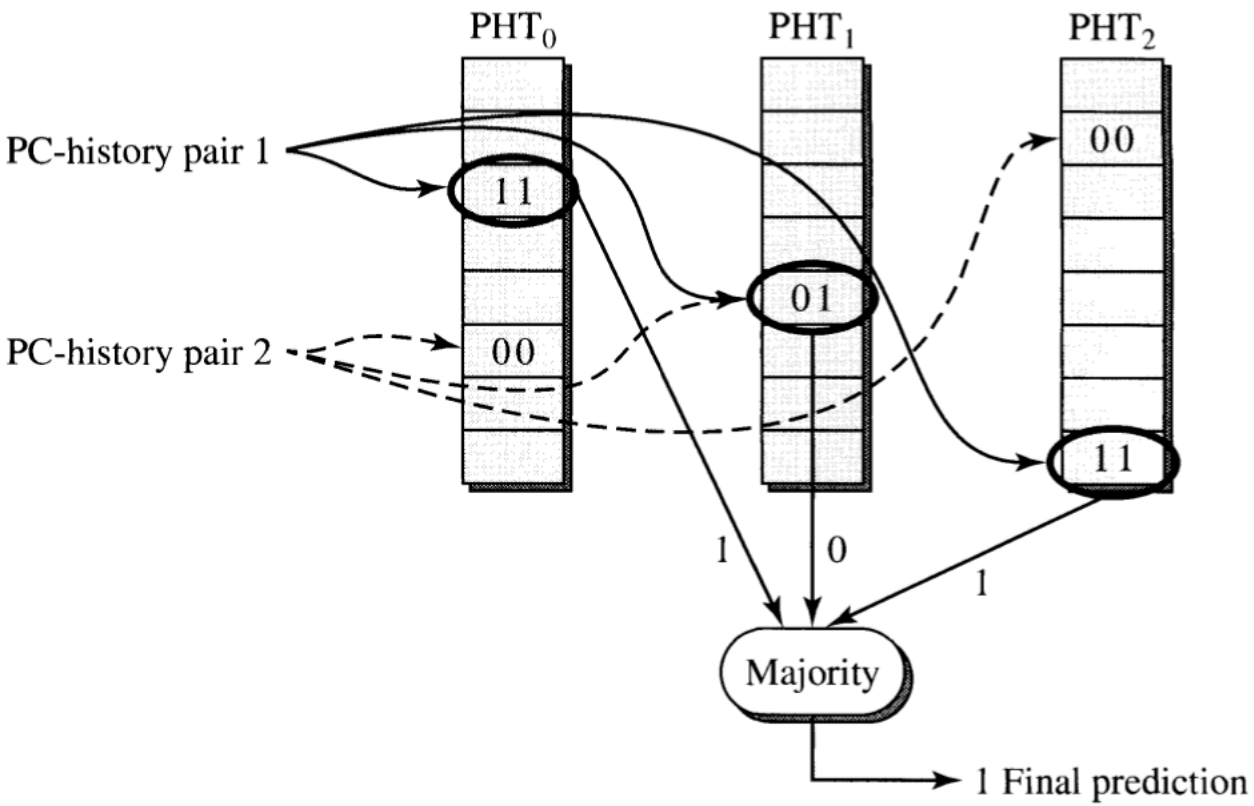
The gskew predictor

- Create 3+ PHT “banks”
- Hash Address-BHR pair
  - How does this compare to Bi-Mode?
- Read predictions from each banks, take majority
- Intuition:
  - Even if you have a conflict, the majority vote will mitigate negative impact
- Clever trick: f0, f1, f2 can be created such that:
  if f0(x) == f1(x), then f1(x) != f2(x), f2(x) != f0(x)

  What does this mean? If you have multiple PHT conflicts they’re from independent branches

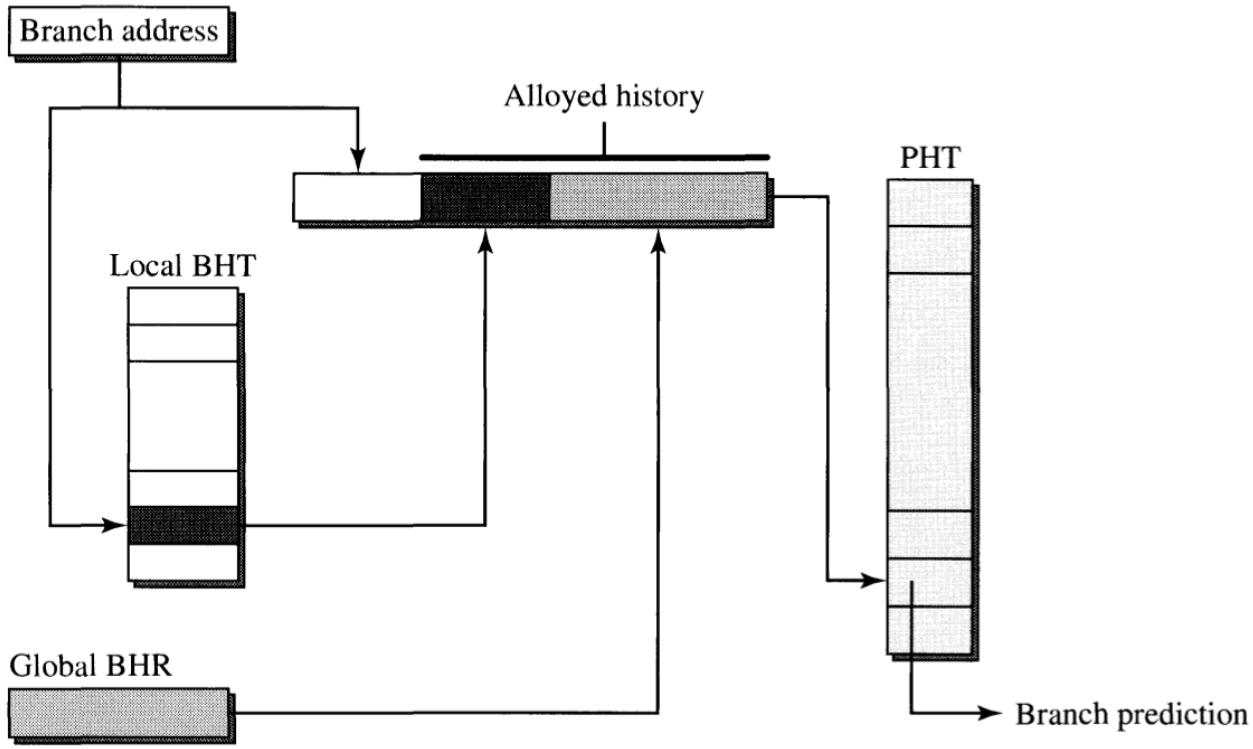


Gskew example



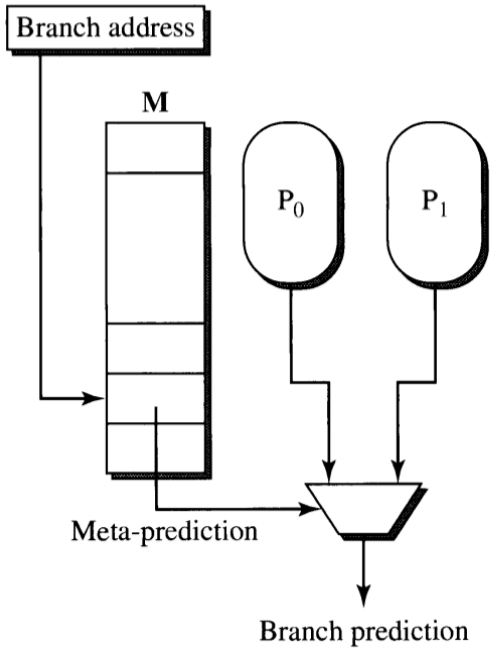
The Alloyed History Predictor

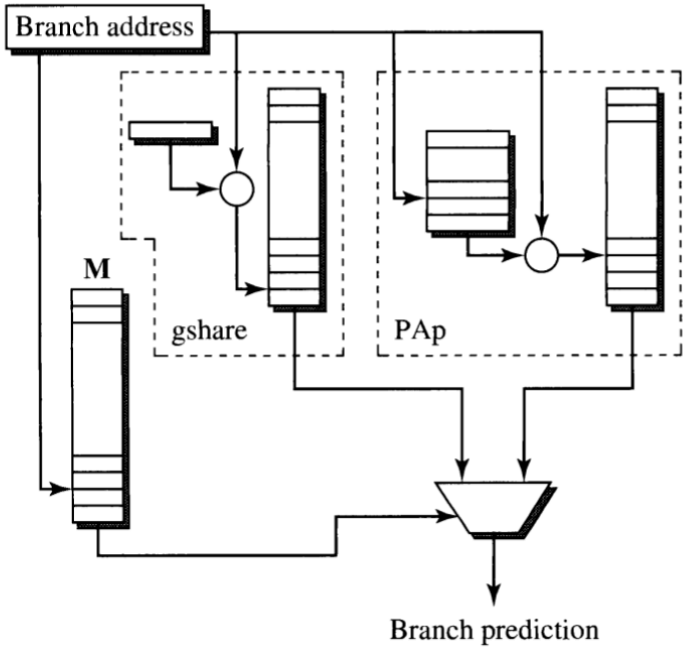
- Some branches need global history (1 BHR) some need local history (BHT)
  What to do?
  - Combine them!
- Alloyed predictor tracks both global and local branch history
- Three benefits:
  - Cheap!
  - Best of both scopes
  - Can correlate even more, since some branches need context from both



Tournament predictors

- Combine multiple BPs
- Use any 2 XAy BPs
- M is 2b counters for MUX select
- After branch resolved, update both P0, P1
- Update M based on predictor mismatch, favoring correct
  - Increment if P0 right, P1 wrong
  - Decrement if P1 right, P1 wrong
  - If bost right/wrong do nothing
- How is this different from Bi-Mode?





What's happening today

Current state-of-the-art (rumored to be what Intel is using)

"TAGE: Tagged Geometric predictor", (Seznac, JILP 2006)

What's research looking into?

Neural predictors!

1) Daniel Jimenez (Texas A&M) did lots of pioneering work
2)Yale Patt(?) has neural prediction paper at MICRO this month:

"BranchNet: A Convolutional Neural Network to Predict Hard-To-Predict Branches"

This sounds interesting.. I want to learn more

1) Highly suggest reading

   Chapter 9 (Advanced Instruction Flow Techniques) in
   "Modern Processor Design" by Mikko Lipasti

2) Read "TAGE" and work backwards through references

3)Try out the branch prediction championchip:

   https://www.jilp.org/cbp2016/

4) If you can beat the SOTA in branch prediction, I'll give you a PhD.