COMP3710 (Class # 5176)
Special Topics in Computer Science
Computer Microarchitecture

Convener: Shoaib Akram
shoaib.akram@anu.edu.au

Australian
National
University

Plan & Progress

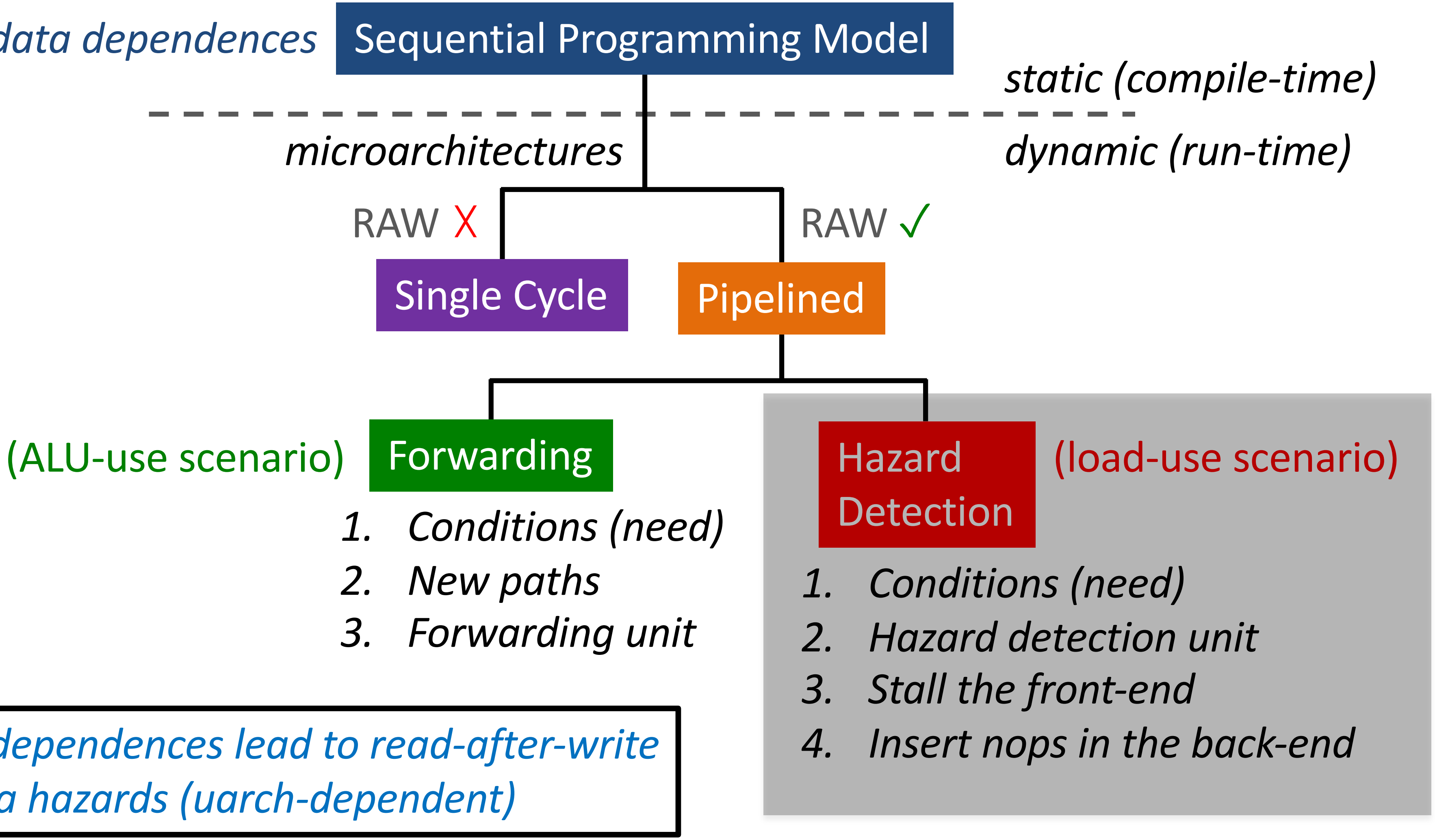
Week 3: Pipelined In-Order (IO) processor

Week 3: Add forwarding to avoid data hazards

This Week: Add hazard detection to avoid load-use hazards

This Week: Mitigate branch hazards + branch prediction

Big Picture



data dependences Sequential Programming Model

static (compile-time)

microarchitectures

dynamic (run-time)

RAW ✗ Single Cycle

RAW ✓ Pipelined

(ALU-use scenario) Forwarding

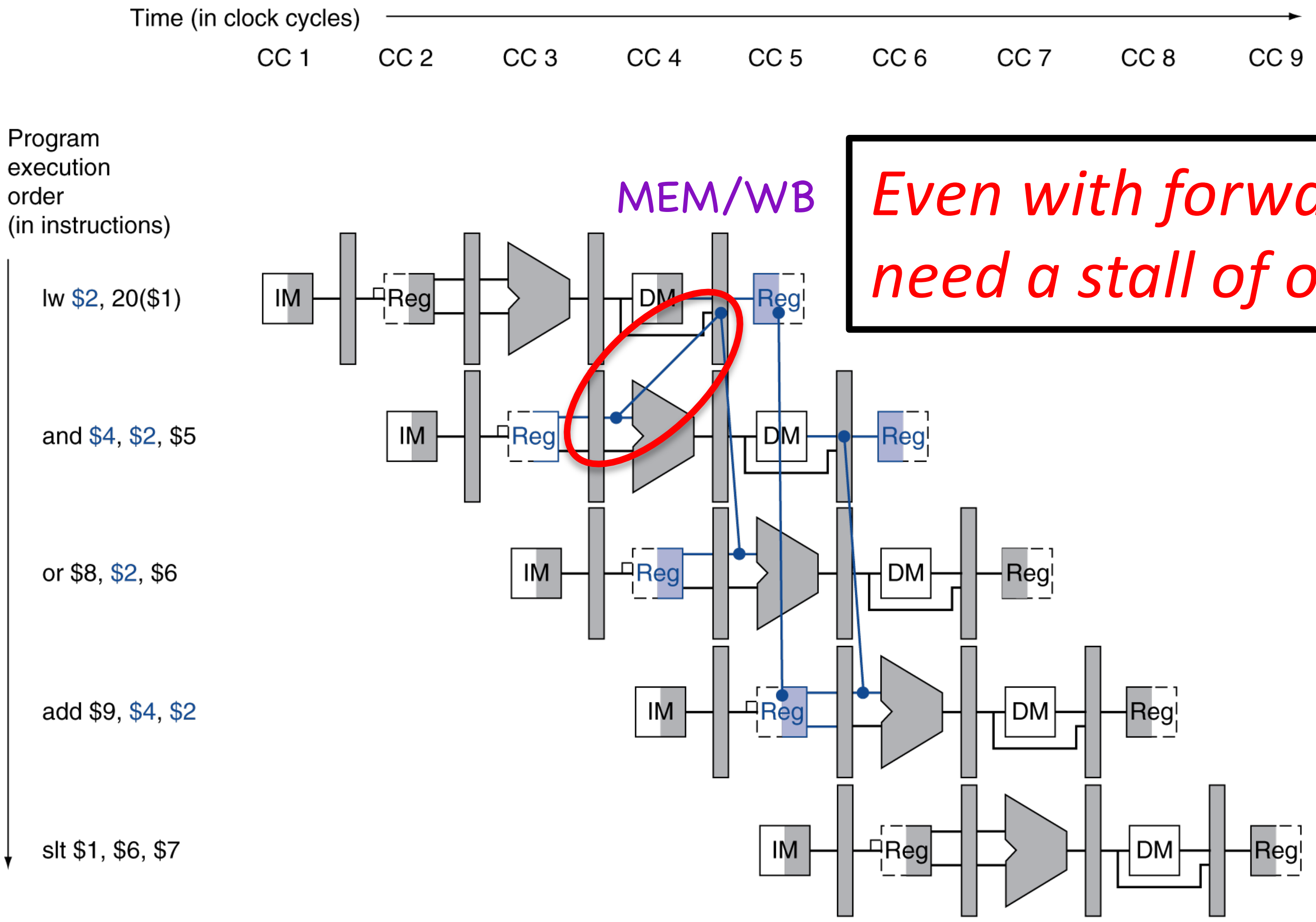
1. *Conditions (need)*
2. *New paths*
3. *Forwarding unit*

Hazard Detection (load-use scenario)

1. *Conditions (need)*
2. *Hazard detection unit*
3. *Stall the front-end*
4. *Insert nops in the back-end*

Note: *True dependences lead to read-after-write or RAW data hazards (uarch-dependent)*

Load-Use Data Hazard



Even with forwarding, we need a stall of one cycle

Detecting Load-Use Hazards

The instruction in the EX stage is a load

And its destination register = following instruction's source

```
if (ID/EX.MemRead
and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt))
      stall the pipeline
```

Perform this check when the "using" instruction is in the decode stage of the pipeline

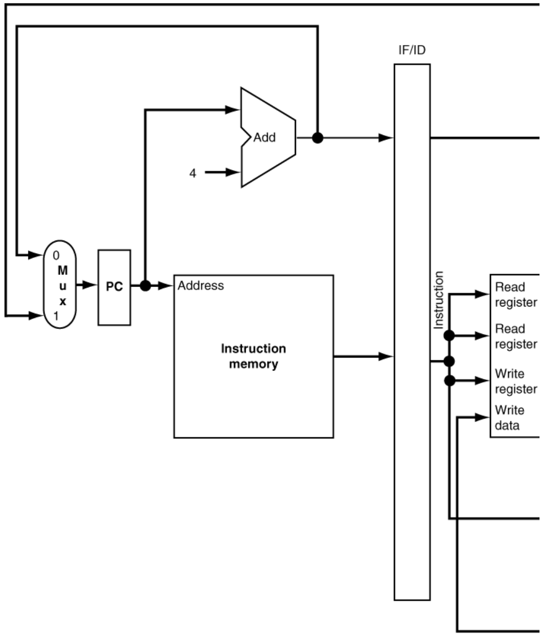
lw $2, 20($1) → ID/EX

and $4, $2, $5 → IF/ID

Stalling the Pipeline

Front-end:

1. Prevent the IF/ID pipeline register from changing
2. Prevent the PC register from changing



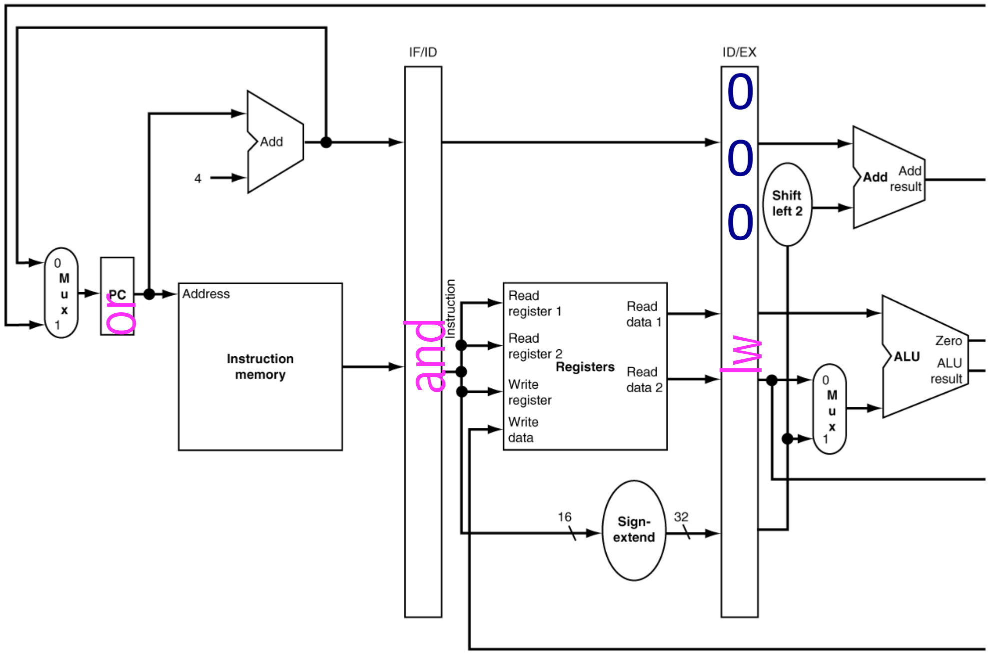
lw $2, 20($1) → ID/EX
and $4, $2, $5 → IF/ID
or $8, $2, $6 → PC

Stalling the Pipeline

Front-end:

1. Prevent the IF/ID pipeline register from changing
2. Prevent the PC register from changing

cycle # 3



Back-end: Send nops

Executing instruction that have no effect (make all control lines 0)

lw $2, 20($1) → ID/EX

and $4, $2, $5 → IF/ID

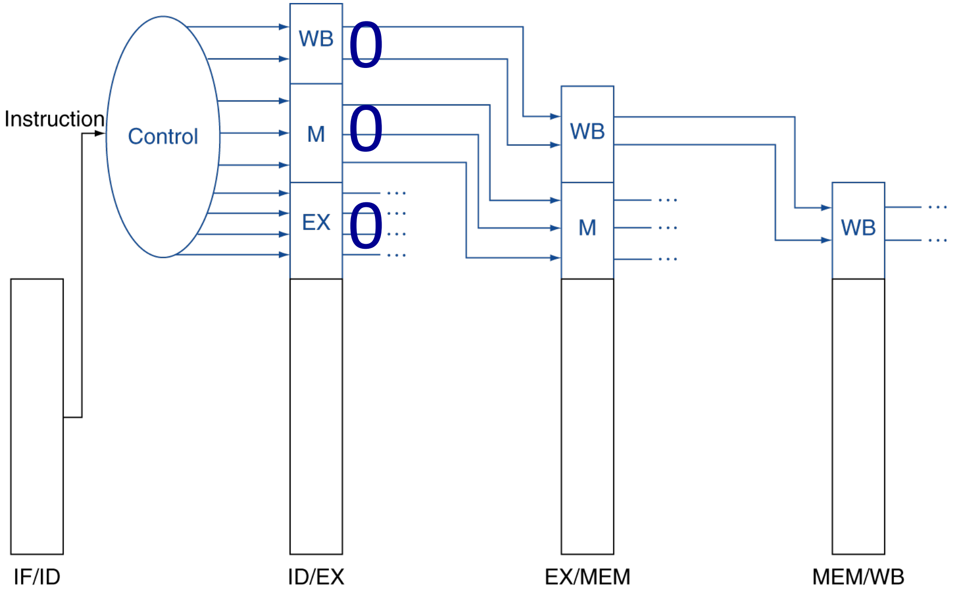
or $8, $2, $6 → PC

Stalling the Pipeline

Back-end:

Deassert all control signals in the EX, MEM, and WB stages

(*Equivalent to inserting a nop)*



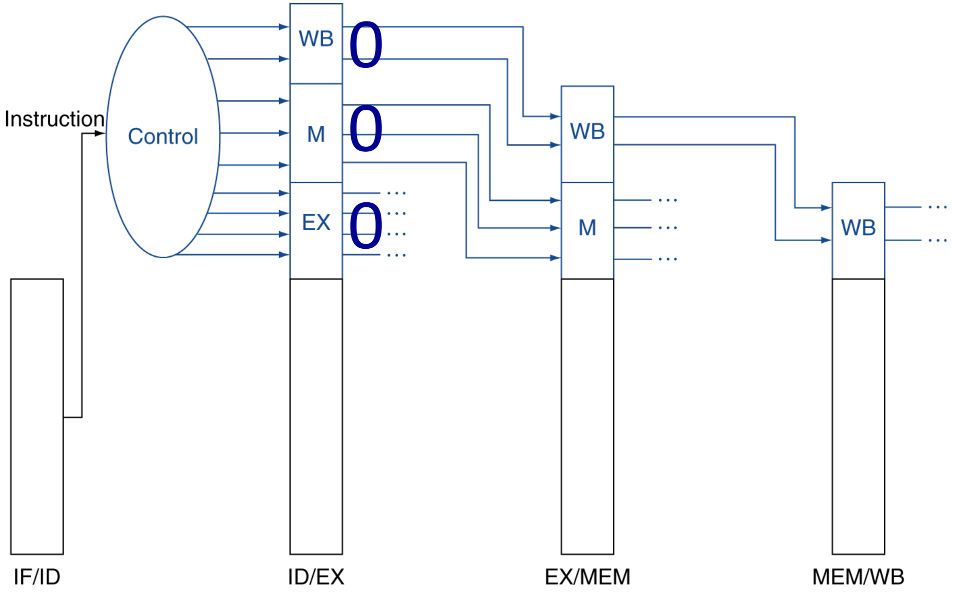
Question: Do we need to deassert *all control signals or deasserting some of them is sufficient?*

Stalling the Pipeline

Back-end:

Deassert all control signals in the EX, MEM, and WB stages

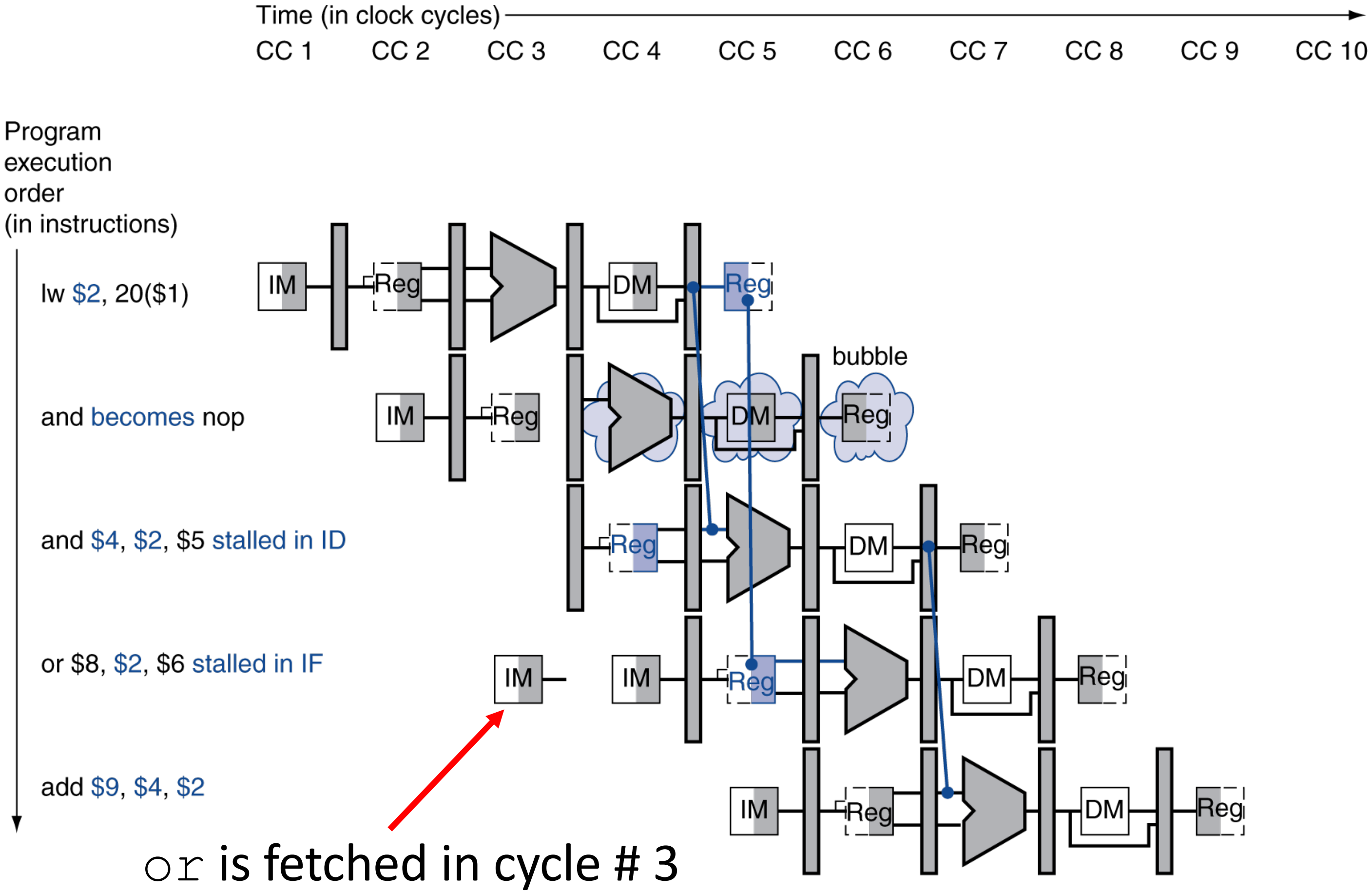
(*Equivalent to inserting a nop)*



Question: Do we need to deassert *all control signals or deasserting some of them is sufficient?*

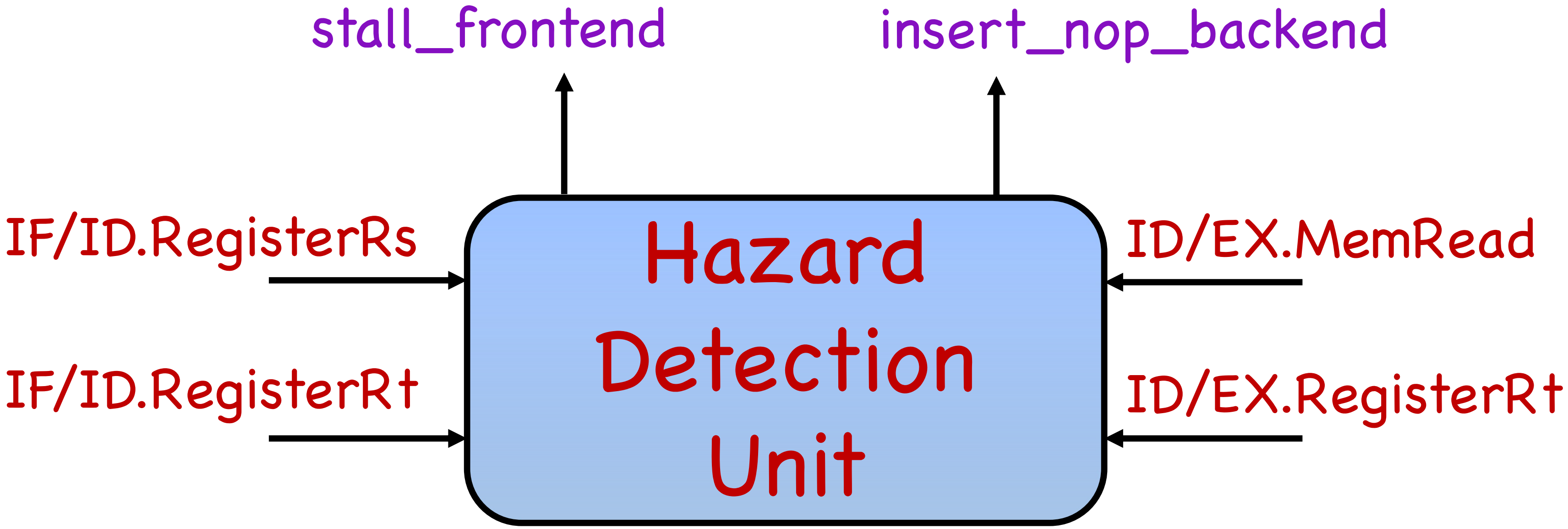
RegWrite, MemRead, and MemWrite

Example: Inserting a Stall



`or` is fetched in cycle # 3

Hazard Detection Unit



stall_frontend

insert_nop_backend

IF/ID.RegisterRs

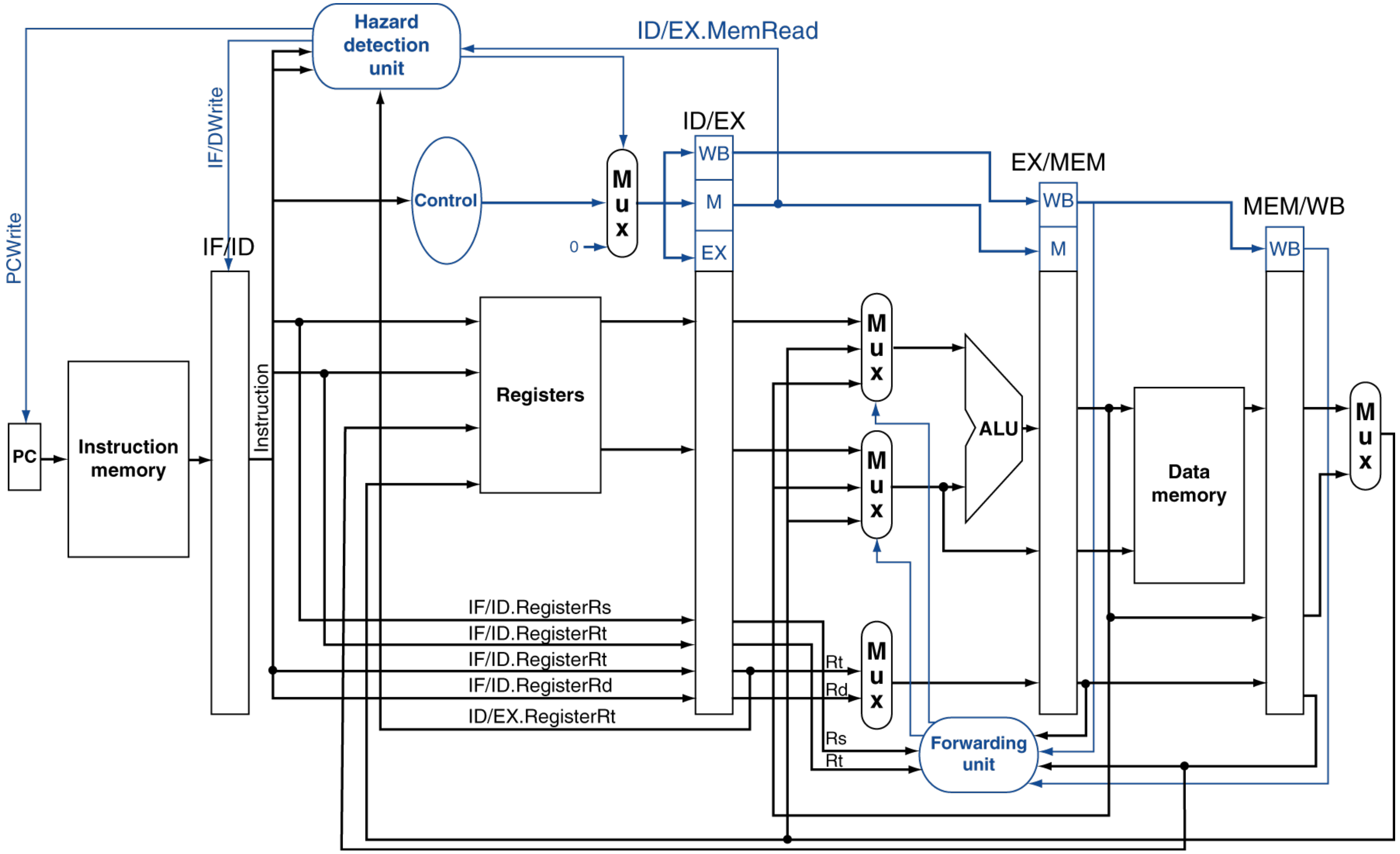
IF/ID.RegisterRt

Hazard Detection Unit

ID/EX.MemRead

ID/EX.RegisterRt

Datapath with Hazard Det.



Stalls and Performance

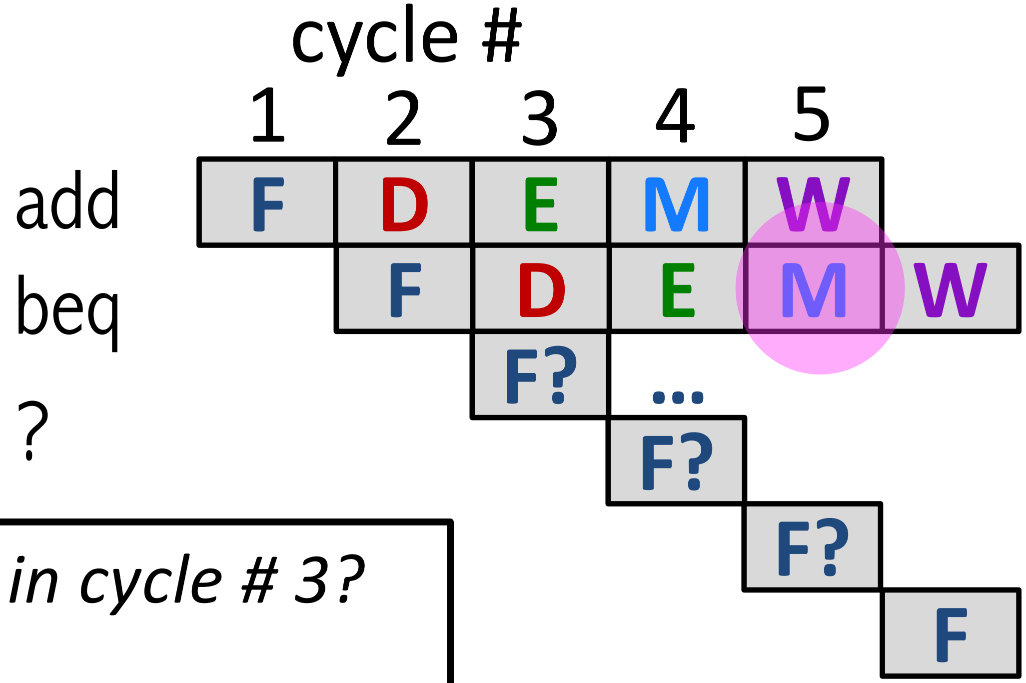
Compiler relies on hardware for correct execution

Compiler should still be aware of the microarchitecture for optimally organizaing/reordering the instruction sequence

Control Hazards

Consider the following instruction sequence

```
        add   $4   $5       $6
        beq   $1   $2       40
        lw    $3   300($0)
40      or    $7   $8       $9
```



Which instruction should we fetch in cycle # 3?

1. *PC + 4*
2. *40*

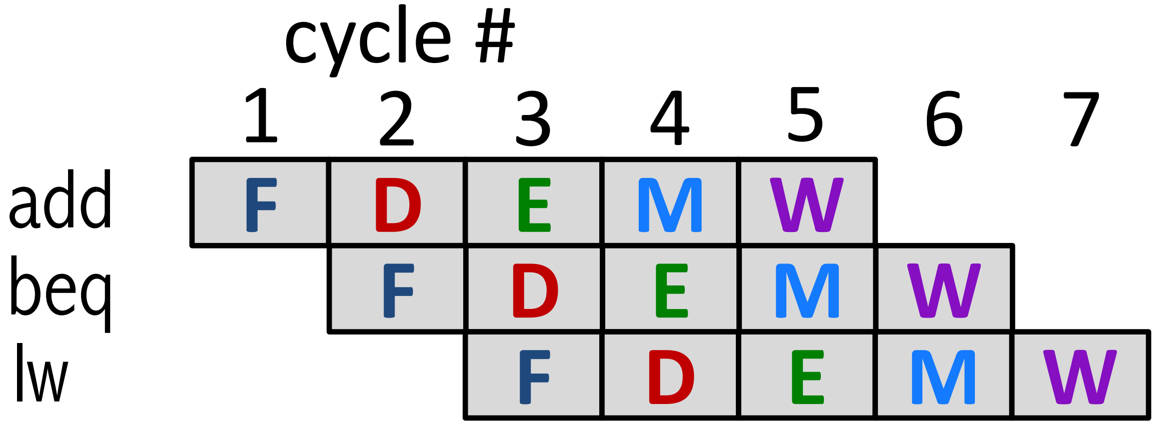
Control Hazards

Problem: *The pipeline cannot determine the next instruction to fetch, since it only just received the branch instruction from memory (residing in the IF/ID PPR)*

Control or branch hazard: *When the proper instruction cannot execute in the proper pipeline cycle because the instruction that was fetched is not the one that is needed*

1. *Stall the pipeline until branch outcome is ready (high penalty)*
2. *Predict outcome and **flush** instructions if wrong*
3. *Seek compiler support (branch delay slot)*

Execution with Taken and Not Taken

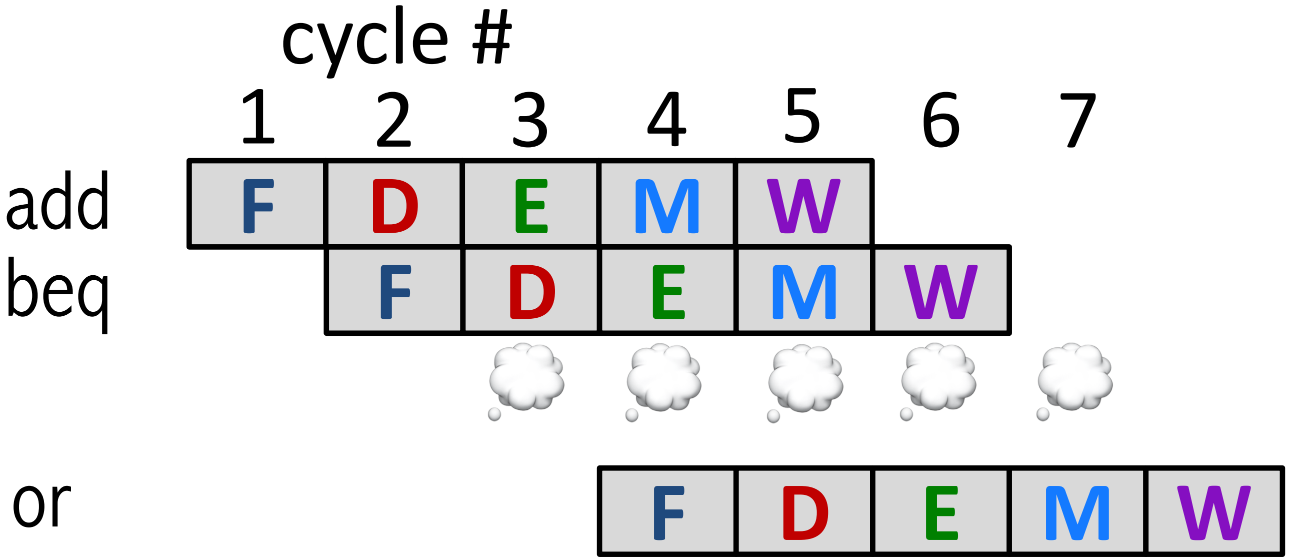


cycle #

	1	2	3	4	5	6	7
add	F	D	E	M	W		
beq		F	D	E	M	W	
lw			F	D	E	M	W

If we assume branch is not taken and the branch is not taken, we do not incur performance penalty

Assuming we add extra hardware support to compute the branch outcome in the ID stage, we still incur a 1-cycle penatly when the branch is taken



cycle #

	1	2	3	4	5	6	7	
add	F	D	E	M	W			
beq		F	D	E	M	W		
or				F	D	E	M	W

Exercise: Performance Penalty

In the SPECint2006 benchmark suite, branches are 17% (on average) of the total instructions. Assuming branch outcomes are computed in the decode stage, and we stall before fetching the next instruction, what the performance impact of branches on the average CPI. (Assume non-branch instructions have a CPI of 1.)

Answer: **17%**

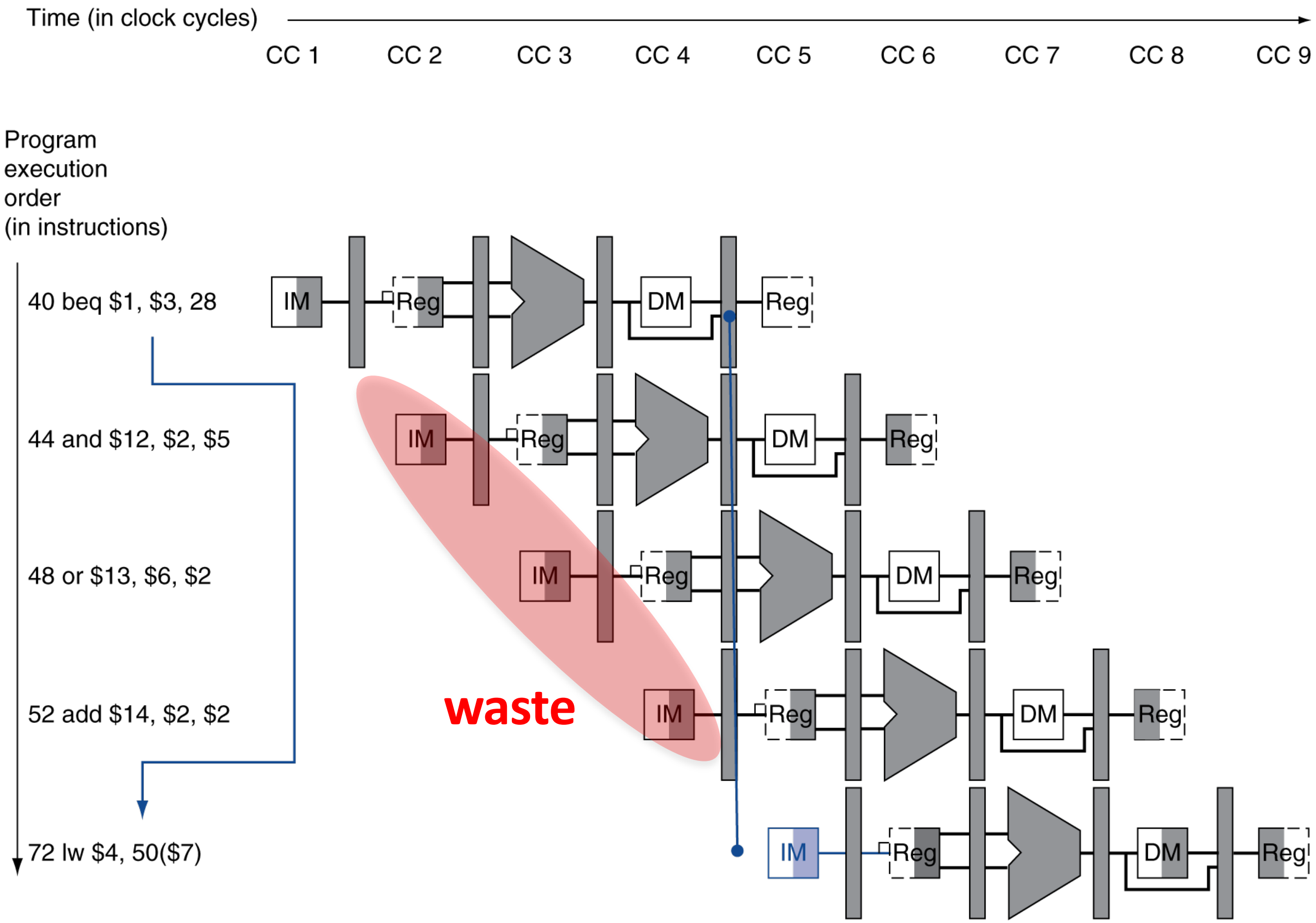
In deeper pipelines, branches are not resolved much later, and the perforamnce penalty from stalling is much higher

Control Hazards

Understanding the cost of *branch not taken* in the absence of hardware support for *early branch resolution*

Adding extra hardware for *early branch resolution*

Control Hazards



Branch Not Taken

Outcome matches prediction: *Noting to do*

Outcome does not match prediction:

1. *Discard instructions (change the control values to 0)*
2. *Flush the pipeline (EX, ID, IF)*
3. *Update the PC with the correct target address*

This policy is called static branch prediction:
Used in Intel i486

Extra Hardware for Taken Branches

How can we reduce the cost of the taken branch?

→ *Branch resolution in MEM stage leads to 3 insts of wasted work*
→ *What if we resolve the ID stage? (1 wasted instruction)*

Moving the branch in Decode requires two early actions:

→ *Branch decision*
→ *Branch target (easy, PC + constant field from IF/ID)*

Branch decision

→ *Equality testing (easy, XOR all bits, and then OR then together)*
→ *The difficult part is new data hazards in the Decode stage*

Branch Decision in ID Stage

Branch decision

- → *Another forwarding unit in ID stage (formerly only in EX stage)*
- → *Hazard detection*

Flushing the instruction

- → *Add an extra control line (IF.Flush) that zeros IF/ID PPR (transform to nop)*

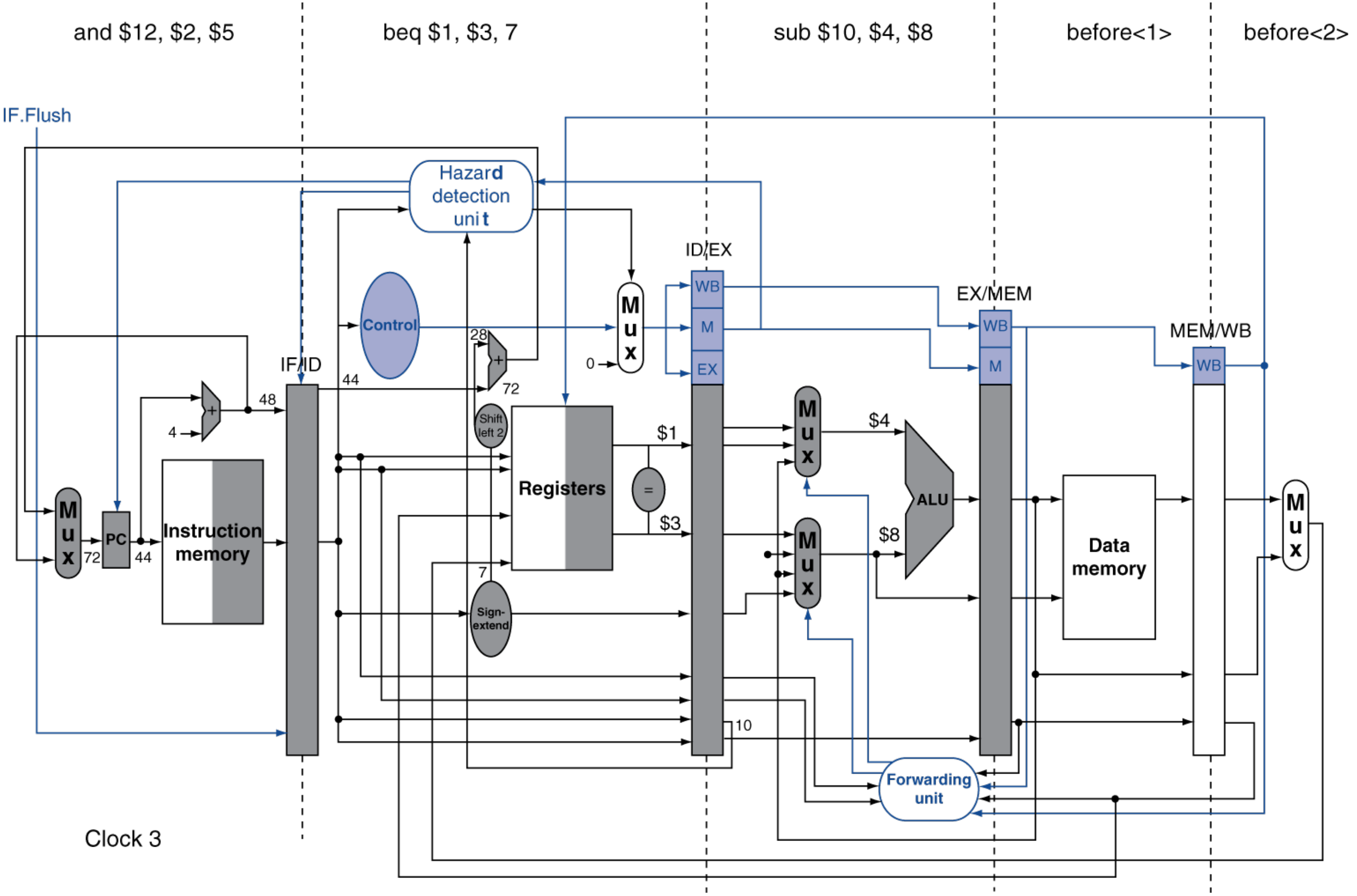
Exercise

What happens when the branch is taken in the instruction sequence, assuming the pipeline is optimized for branches that are not taken and that we moved the branch execution to the ID stage?

```
36      sub     $10     $4      $8
40      beq     $1      $3      7
44      and     $12     $2      $5
48      or      $13     $2      $6
52      add     $14     $4      $2
56      slt     $15     $6      $7
...
72      lw      $4      50($70
```

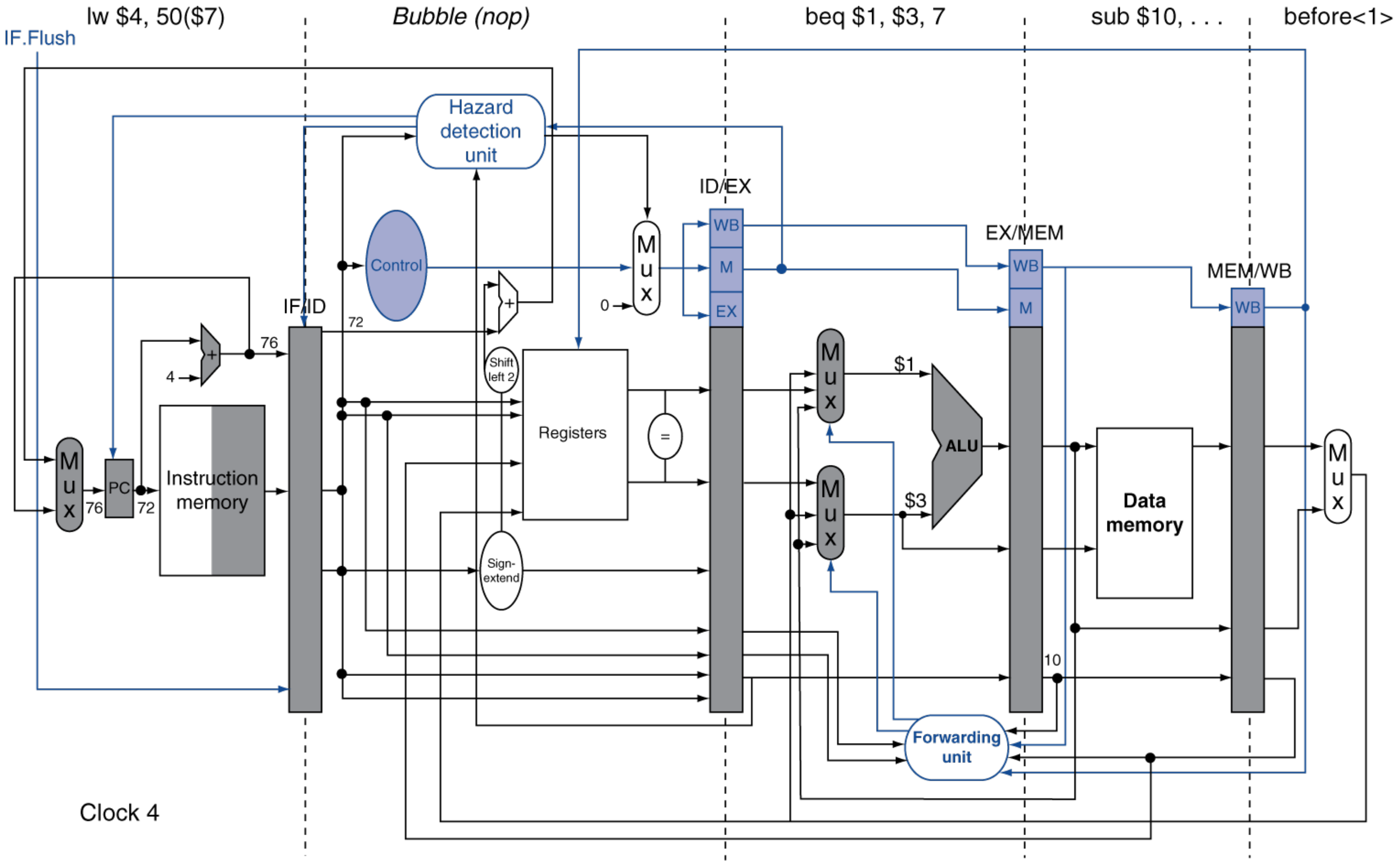
target = 40 + 4 + 7*4 = 72

Cycle 3



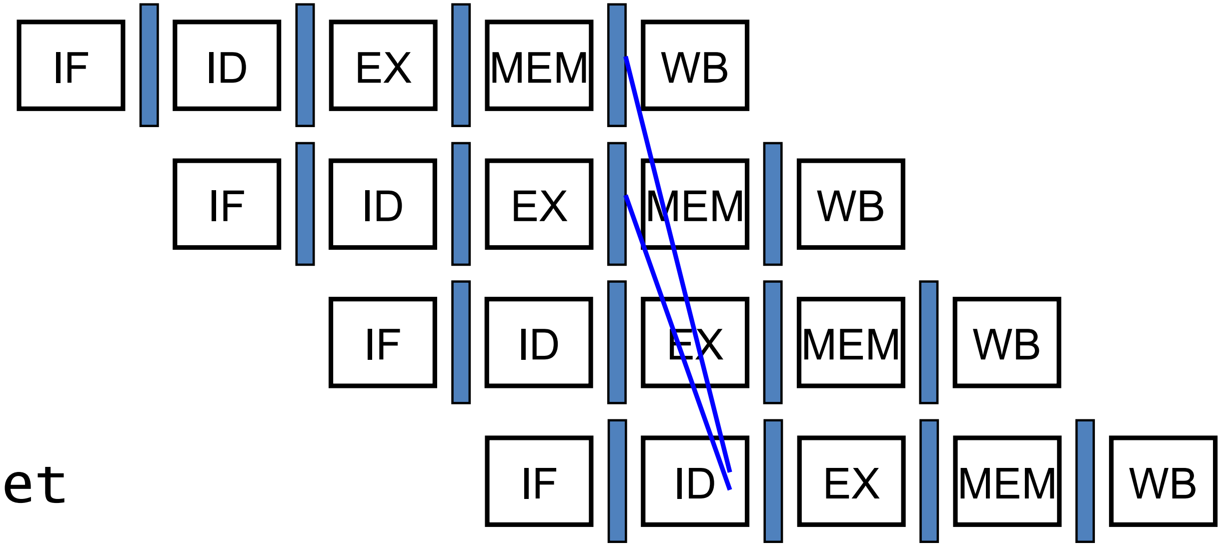
and $12, $2, $5
beq $1, $3, 7
sub $10, $4, $8
before<1>
before<2>
IF.Flush
Hazard detection unit
ID/EX
EX/MEM
MEM/WB
WB
M
EX
WB
M
WB
Control
28
0
Mux
IF/ID
44
72
48
4
Shift left 2
Registers
$1
=
$3
$4
$8
ALU
Data memory
Instruction memory
Mux
PC
72
44
7
Sign-extend
Mux
Mux
10
Forwarding unit
Mux
Clock 3

Cycle 4


lw $4, 50($7)
Bubble (nop)
beq $1, $3, 7
sub $10, . . .
before<1>
IF.Flush
Hazard detection unit
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
M u x
0
72
76
4
+
Shift left 2
Registers
=
Instruction memory
PC
76
72
Sign-extend
$1
$3
ALU
Data memory
10
Forwarding unit
Clock 4

Data Hazards for Branches

```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```



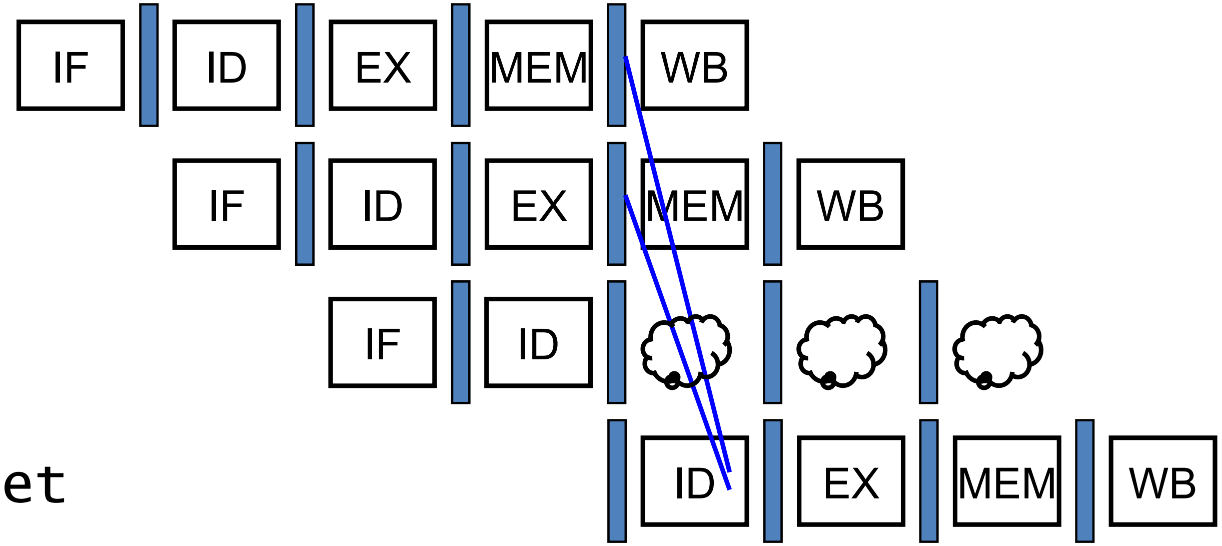
How many stall cycles?

0

If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

Data Hazards for Branches

```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



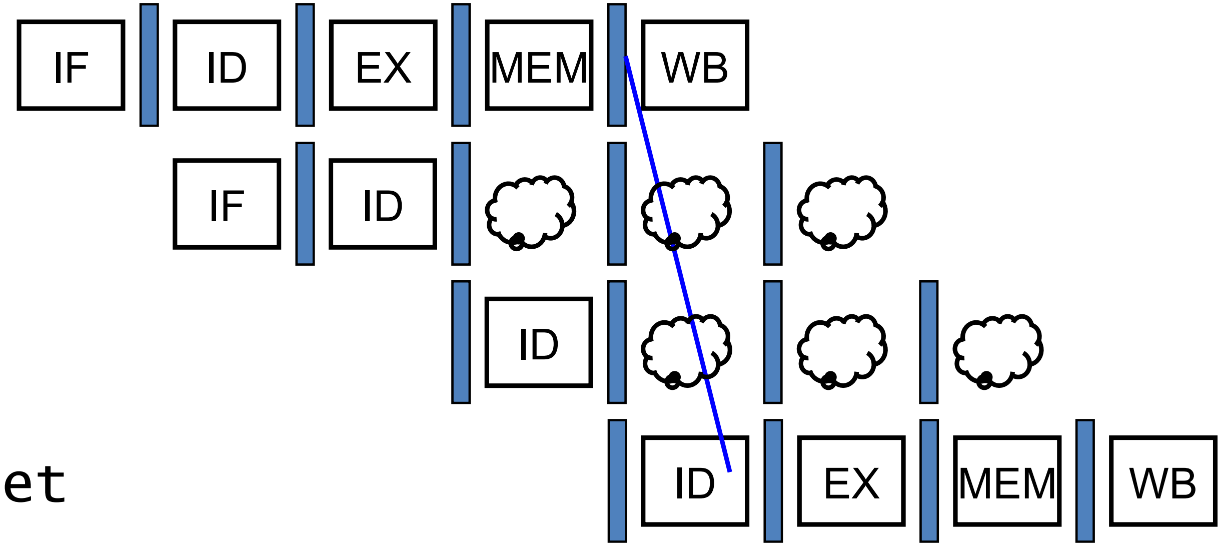
How many stall cycles?

1

If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction

Data Hazards for Branches

```
lw  $1, addr
beq stalled
beq stalled
beq $1, $0, target
```



How many stall cycles?

2

If a comparison register is a destination of immediately preceding load instruction

Dynamic Branch Prediction





What is the similarity in these pics?

What is different about them?

Imagine your co-pilot has poor navigation skills. What will you do to go at full-speed? What if you drive every day and the decision to go right/left changes over time?

Fork Context

Cherry tree: *take a left*

name:most-recent-turn → this-turn

Cherry tree,
Bavaria, Germany

Wind mills,
Retzstadt, Franconia, Germany

Wind mills: *take a right*

Dynamic Branch Prediction

Predict the outcome of a branch instruction (in fetch stage) based on the recent behavior of the branch

What do we need?

1. Branch identification (name/id, uarch-level)
2. Branch behavior (taken/untaken last time)

Branch Identification & Behavior

At the microarchitecture-level, what is the *simplest* way to name a previously encountered branch

→ *The address of the branch instruction in memory*

→ *Instructions are laid out next to each other 4-bytes apart*

What is the *simplest* way to record branch behavior

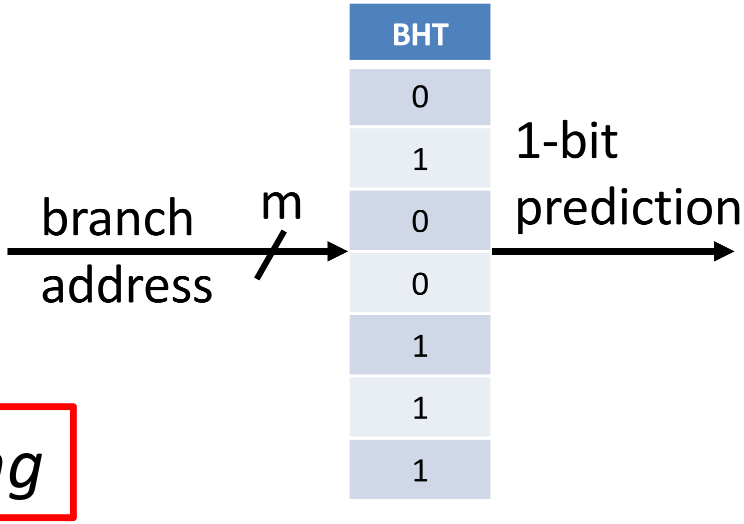
→ *Outcome (grab it from the ALU)*

Branch History Table

Branch History Table (BHT) or Branch Prediction Buffer

→ *A small amount of memory indexed by the low-order bits of branch address*

→ *Keep a single bit that says branch was recently taken or not*



2^m entries → aliasing

Operation

Placement & Access: Fetch stage

Predicted untaken: Fetch PC + 4

Predicted taken: Predict/compute target address and fetch target

Outcome matches prediction: *Noting to do*

Outcome does not match prediction:

1. *Flip the entry in the BHT*
2. *Flush the pipeline (EX, ID, IF), update PC*

Is correctness affected by misprediction?
Is performance affected by misprediction?

Accuracy/Perf of 1-bit Predictor

Consider the following loop:

```
for (i = 1; i < n; i++)
{
        do something;
}
```

i =	1	2	3	4	5	6	7	8	9	10
Direction	T	T	T	T	T	T	T	T	T	NT
State/Prediction	NT	T	T	T	T	T	T	T	T	T
New State	T	T	T	T	T	T	T	T	T	NT

What is the prediction accuracy of a 1-bit branch predictor if n is equal to 10?

→ *Accuracy is 80% for a branch that is taken 90% of the time*

This example is the reason why forward untaken/backwards taken was used as backup in Intel Pentium Pro, II, III, 4

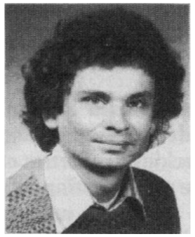
Anomalous Decision

When branches that are *strongly biased* toward one direction suddenly takes a different path/direction

E.g., Branches at the end of a loop are usually taken (backward) except for the case of the loop exit (forward)

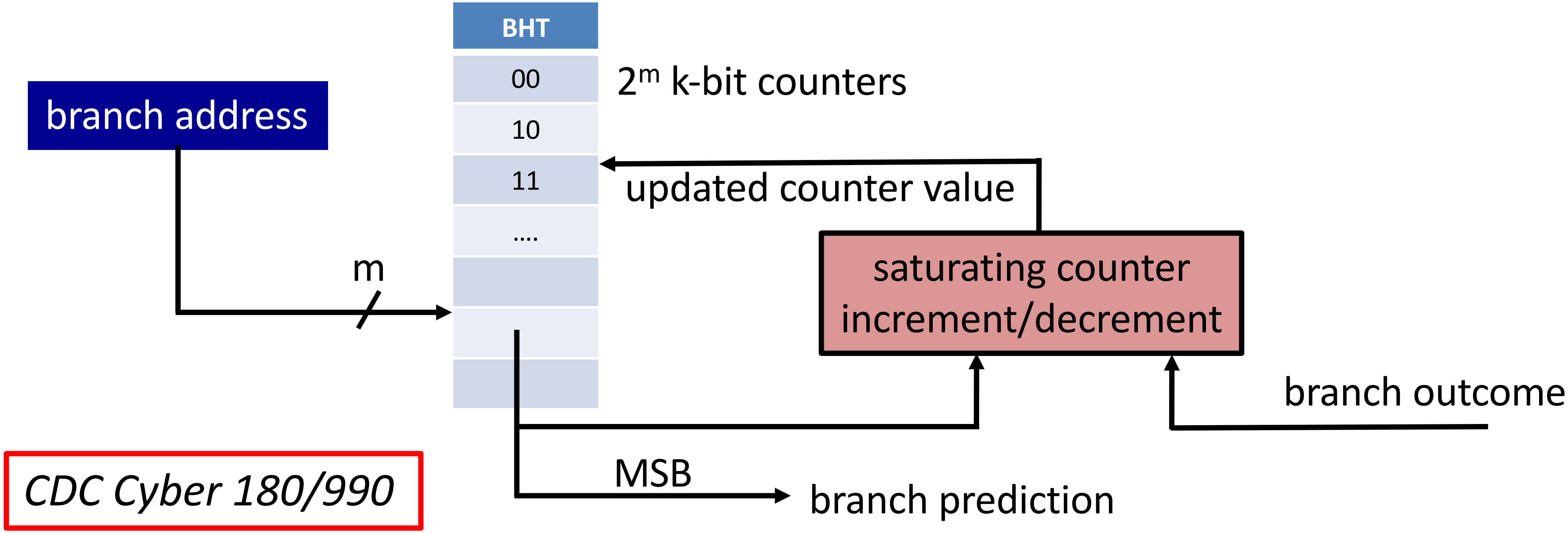
A 1-bit predictor is "thrown off" by a single anomolous decision

Smith's Algorithm



1979: *James E. Smith patents branch prediction at Control Data*

Notices the performance pathology of 1-bit predictor at loop termination
Key insight: Add hysterisis (inertia) to the predictor's state



BHT

00
10
11
….

2^m k-bit counters

branch address

m

updated counter value

saturating counter increment/decrement

branch outcome

MSB

branch prediction

CDC Cyber 180/990

k = 2

A saturating counter maps the outcomes of several recent branches on to a counter with different states

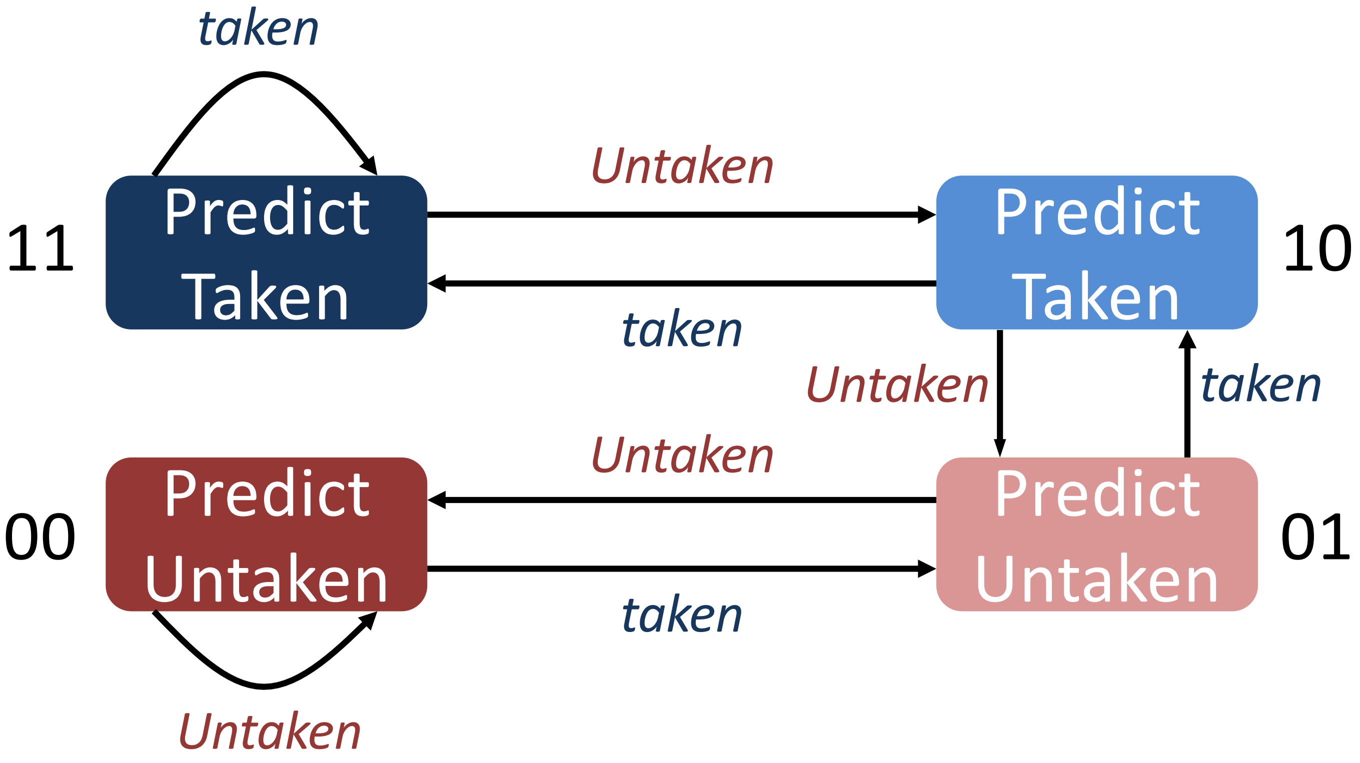
Four states:

1. *Strongly not-taken (SN or **0**0)*
2. *Weakly not-taken (WN or **0**1)*
3. *Weakly taken (WT or **1**0)*
4. *Strongly taken (ST or **1**1)*

The same outcome must occur multiple times to reach the strong states

State Diagram with k = 2

The state transitions show the *bimodal* behavior of Smith predictor



Accuracy of Smith's Predictor

Accuracy of Smith$_1$ (1-bit counter) and Smith$_2$ (2-bit counter) on a sequence of branches with a single anomolous decision

Branch	**Branch Direction**	**Smith$_1$ (k = 1)**		**Smith$_2$ (k = 2)**	
		State	Prediction	State	Prediction
A	1	1	1	11	1
B	1	1	1	11	1
C	0 *anomaly*	1	1 (misprediction)	11	1 (misprediction)
D	1 *anomaly*	0	0 (misprediction)	10	1
E	1	1	1	11	1
F	1	1	1	11	1

END: Week 4, Part 1

What is interesting about B1, B2, and B3?

```
if (aa == 2)          B1
        aa = 0;
if (bb == 2)          B2
        bb = 0
if (aa != bb)         B3
        {...}
```

```
if (!done && (counter == 15))          B1
{
        counter = 1;
        reset = 1;
}
...
...
if (reset == 1)                        B2
        {...}
if (counter < 2)                       B3
        {...}
```

Branch Correlation

What is interesting about B1, B2, and B3?

```
if (aa == 2)        B1
      aa = 0;
if (bb == 2)        B2
      bb = 0
if (aa != bb)       B3
      {…}
```

(B1, B2, B3)
(T, T, ?)
(T, F, ?)

```
if (!done && (counter == 15))      B1
{
      counter = 1;
      reset = 1;
}
…
…
if (reset == 1)                    B2
      {…}
if (counter < 2)                   B3
      {…}
```

(B1, B2, B3)
(T, ?, ?)
(F, ?, ?)

Behavior of one branch is correlated with that of another

Branch Correlation

Insight # 1, Global Branch Correlation: *Behavior of one branch is often correlated with the behavior of other branches*

Insight # 2, Local Branch Correlation: *Behavior of a branch is often correlated with the past outcomes of the same branch other than the outcome of the branch the last time it was executed*

```
for (i = 1; i < n; i++)
{
        do something;
}
```

n = 3 : 11011011011011011011

n = 4 : 11101110111011101110

Capturing four recent branch outcomes in a window reveal distinct patterns

n = 3 : 11011011011011011011

n = 4 : 11101110111011101110

Correlating Branch Predictors

Branch predictors that use the behavior (outcomes) of other branches to make a prediction

A predictor that uses the outcomes of only a single branch to predict the behavior of that branch does not capture correlation b/w branches

Also called two-level adaptive branch prediction

→ *Two separate levels of branch history to make the prediction*

Recording Branch History

The global history of the most recent m branches can be recorded in an m-bit shift register (*branch history register or BHR*)

state of shift register over time

2-bit shift register

0	0	*initial state*
0	1	*B1 taken*
1	1	*B2 taken*
1	0	*B3 not taken*

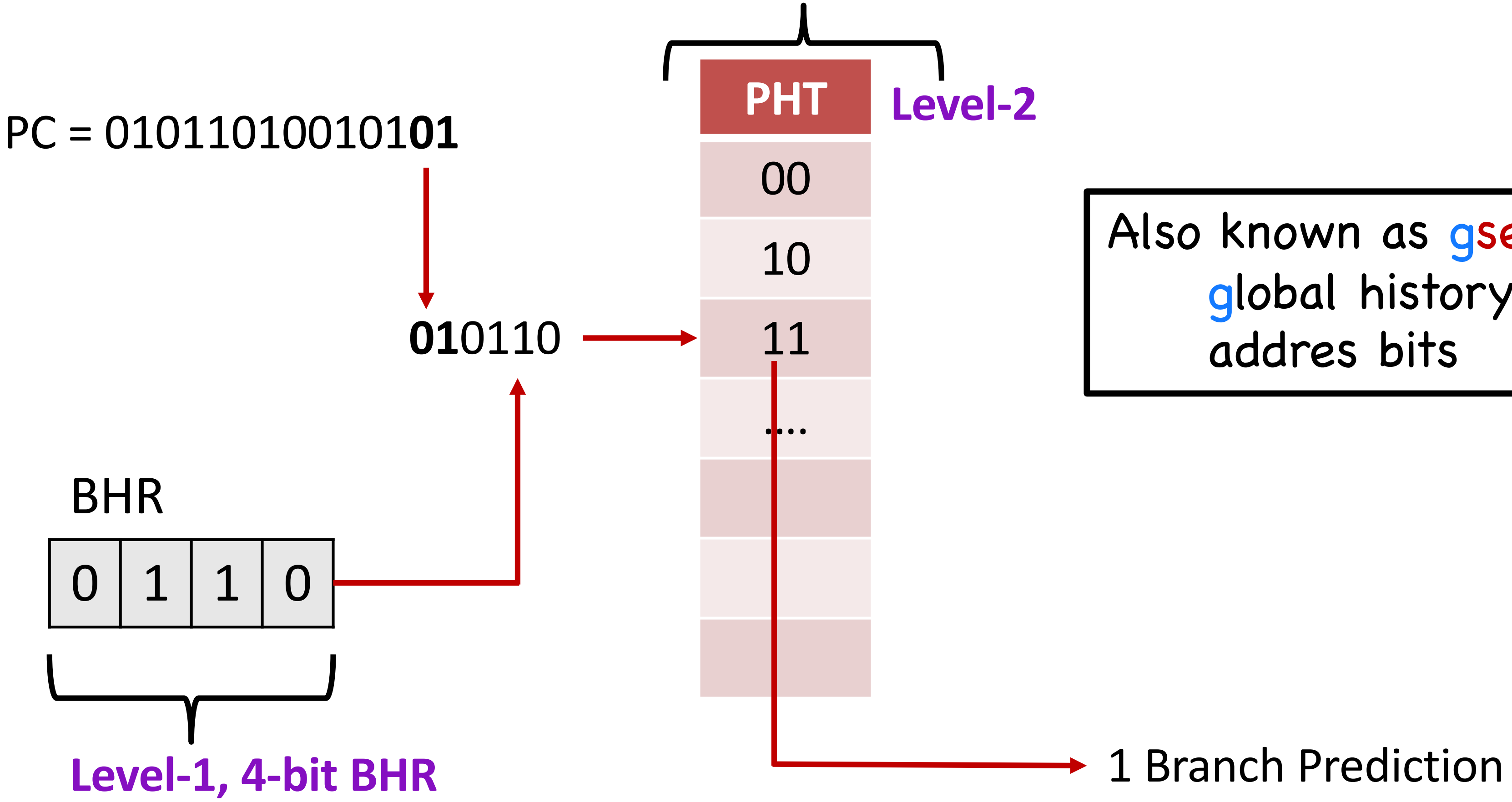
3-bit shift register

0	0	0	*initial state*
0	0	1	*B1 taken*
0	1	1	*B2 taken*
1	1	0	*B3 not taken*

Shift the *actual* outcome at one end and *discard* the oldest outcome

Global-History Two-Level Predictor

Pattern History Table (PHT): *A table of saturating 2-bit counters. Indexed with a concatenation of BHR and a selection of branch address bits*



PC = 0101101001010**01**

PHT

Level-2

00

10

11

….

010110

1 Branch Prediction

BHR

0	1	1	0

Level-1, 4-bit BHR

Also known as gselect:
global history with selected
addres bits

Operation of gselect

The counters in the indexed PHT provides prediction (and are updated) in a manner similar to Smith's algorithm

The contents of BHR are shifted with the correct outcome of the branch

PHT is indexed with a concatenation of address and history bits

Remembering gselect: Global branch history + Per-address PHT indexing

Can ignore PC and just use BHR for indexing

Intuition for gselect

Behavior of one branch is correlated with that of an earlier branch

1. The branches test conditions involving the same variable
2. One instruction gaurds a variable that another one tests

Irrelevant Branches & Training Time

```
x = 0
if (something) /* branch A */
        x = 3;
if (someothercondition) /* branch B */
        y += 19;
if (x <= 0) /* branch C */
        dosomething();
```

C	B	A
1	0	0
0	0	1
1	1	0
0	1	1

1. *Tracking A could help achieve perfect prediction for C*
2. *The “irrelevant” branch B increases the training time of gselect (predictor must* ***learn*** *to ignore these irrelevant history bits)*

Size of PHT

For a harware budget of 4 KB, how many entries can reside in the PHT at once? (Assume 2-bit saturating counters)

*General formula: For an X KB PHT, the PHT contains 4*X 2-bit counters (Each byte can hold 4 2-bit saturating counters)*

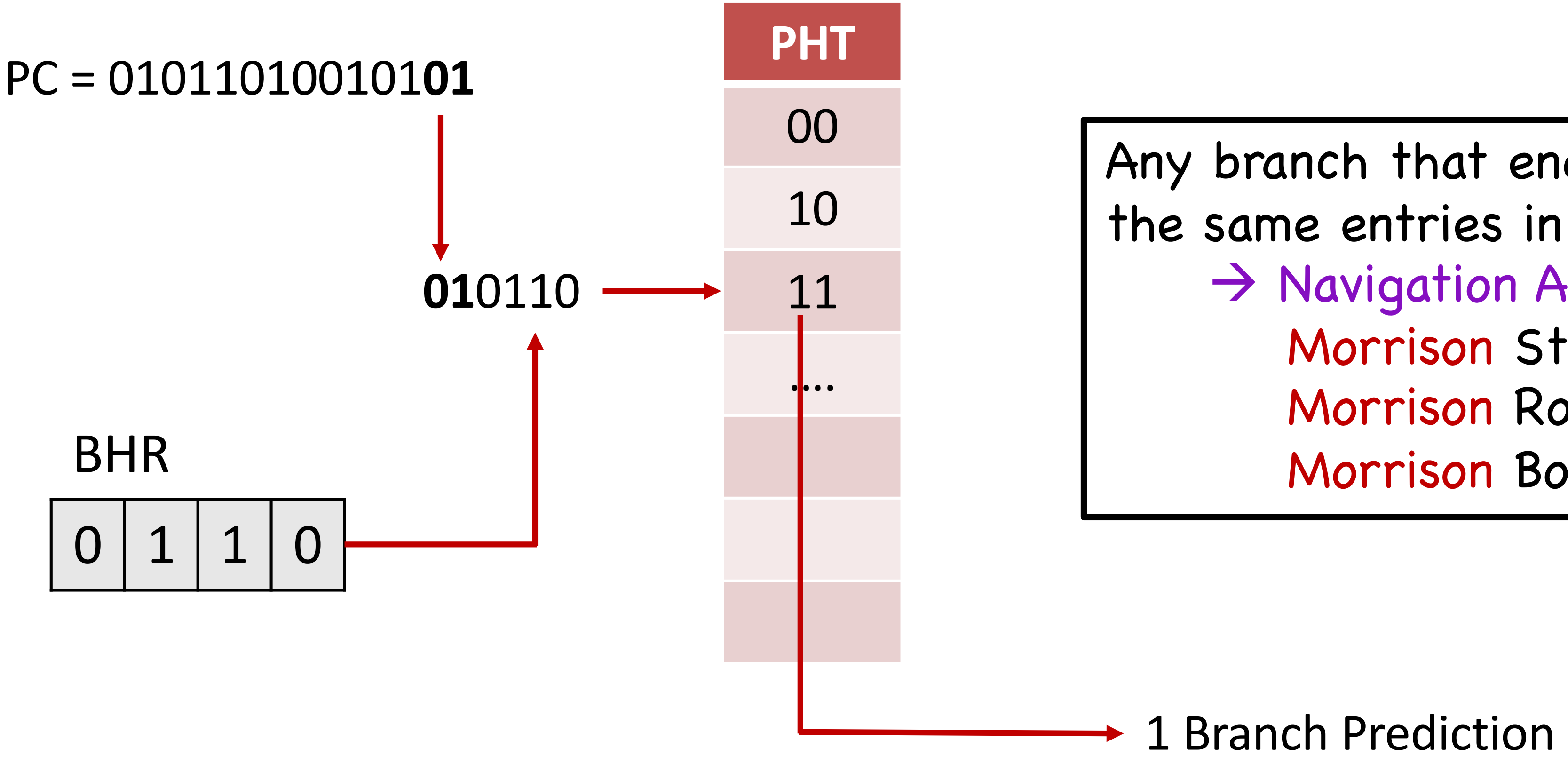
Indexing the PHT

History bits: h
Address bits: m
entries in the PHT: 2^{h+m}

For a 4 KB PHT with 16,384 entries, we have 14 bits (total)

Distributing 14 bits across history and address exposes interesting tradeoffs

PHT Indexing: # Address Bits Matter



PC = 010110100101**01**

010110

PHT

00

10

11

....

BHR

0	1	1	0

1 Branch Prediction

Any branch that ends in **01** share the same entries in the PHT

→ Navigation Analogy:

Morrison Street,
Morrison Road,
Morrison Boulevard

PHT Indexing: # History Bits Matter

Using more history bits allows the predictor to correlate against more complex branch history patterns

Optimally balancing the address vs. history bits depends on: (1) compiler's arrangement of the code, (2) program, (3) ISA, and (4) program inputs (anything else?)

Local-History Two-Level Predictor

Global History: Track outcomes of several branches encountered

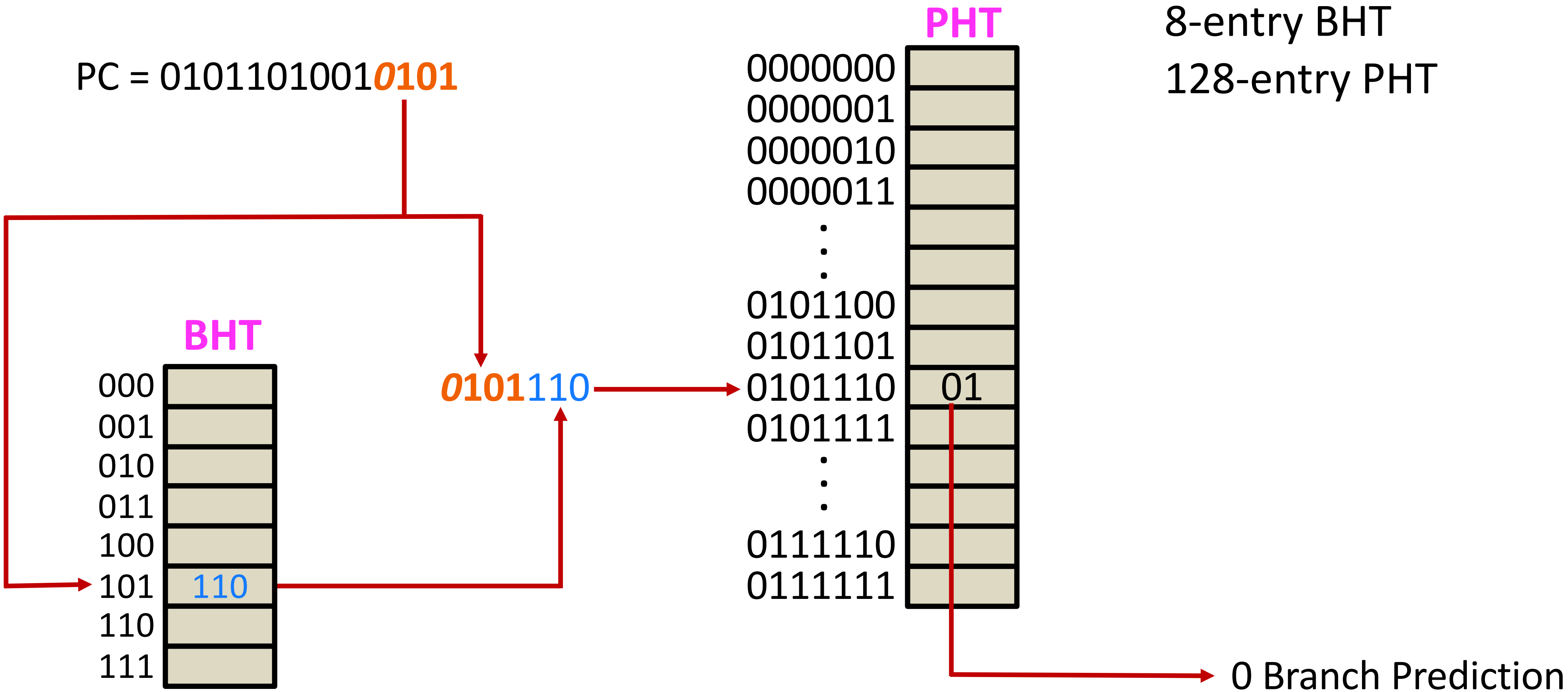
Local History: Track the outcomes of the last several outcomes of only the current branch

Navigation Analogy

- → Intersection R, weekdays *right* (work 😔), weekends *left* (downtown 😀)
- → Turns made to get to the intersection (i.e., global history) are the same
- → If R,R,R,L is the local (intersection R) history, then what day is next?

Remembering driving decision history for each intersection requires more information than the local history

Local-History Two-Level Predictor



8-entry BHT

128-entry PHT

PC = 0101101001**0101**

BHT

000
001
010
011
100
101 110
110
111

0101110

PHT

0000000
0000001
0000010
0000011
:
0101100
0101101
0101110 01
0101111
:
0111110
0111111

0 Branch Prediction

Local-History Two-Level Predictor

Replace global BHR with a one BHR per branch

→ *BHR = BHT with one entry*

Combine BHR contents with selected PC bits to index the PHT

Smith’s saturating counters work as before

Shift the most recent branch outcome into the selected entry into the BHT

Per-address branch history + Per-address PHT indexing

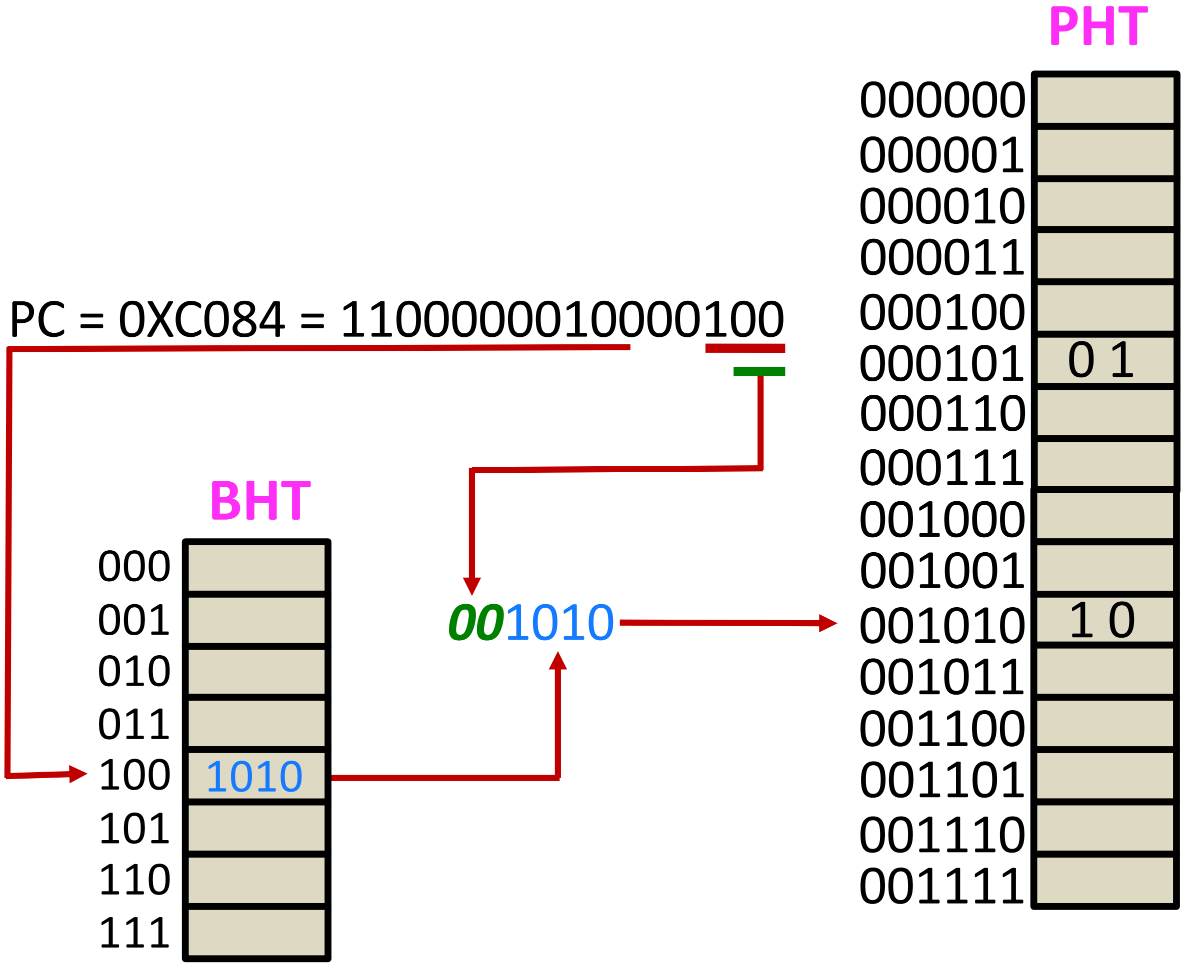
Can ignore PC and just
use BHT entries for indexing

Sizing Tradeoffs

1. Balancing the # history and # non-history (address) bits for the PHT index
2. For a given budget, how many bits for the BHT and how many for the PHT
3. BHT: # entries and the width of each entry (history length)

Sizing Formula: *For a predictor with an L-entry BHT and an h-bit history, and one that uses m bits of the branch address for the PHT index requires a total size of* $L*h + 2^{h+m+1}$

Working Example



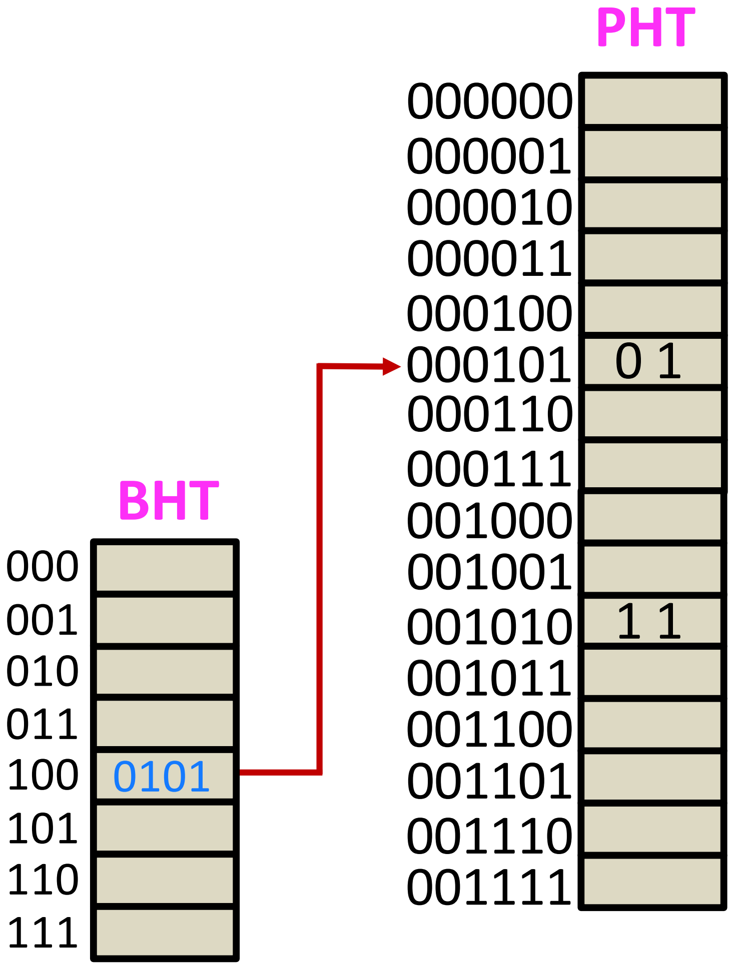
PC = 0XC084 = 1100000010000100

*00*1010

	BHT
000	
001	
010	
011	
100	1010
101	
110	
111	

	PHT
000000	
000001	
000010	
000011	
000100	
000101	0 1
000110	
000111	
001000	
001001	
001010	1 0
001011	
001100	
001101	
001110	
001111	

Outcome = Taken



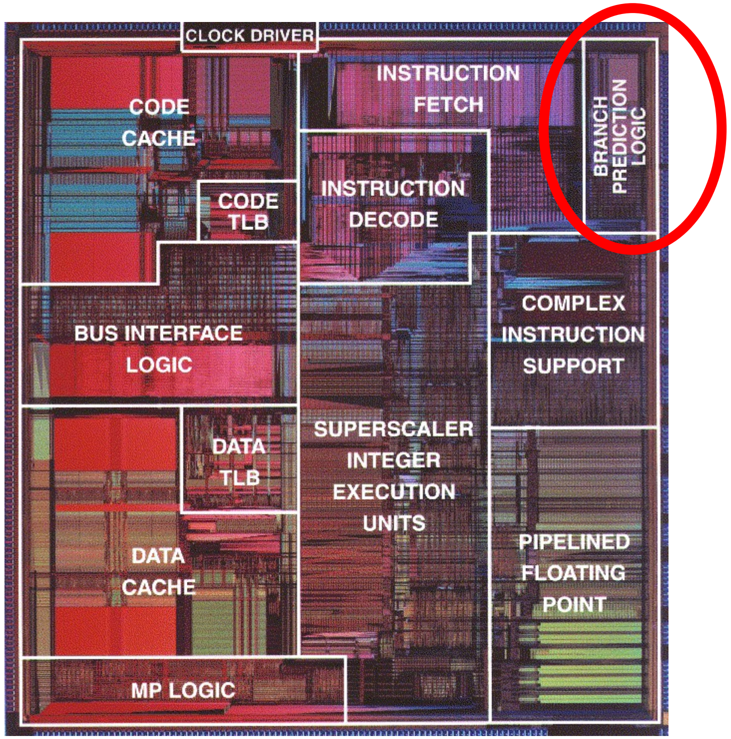
	BHT
000	
001	
010	
011	
100	0101
101	
110	
111	

	PHT
000000	
000001	
000010	
000011	
000100	
000101	0 1
000110	
000111	
001000	
001001	
001010	1 1
001011	
001100	
001101	
001110	
001111	

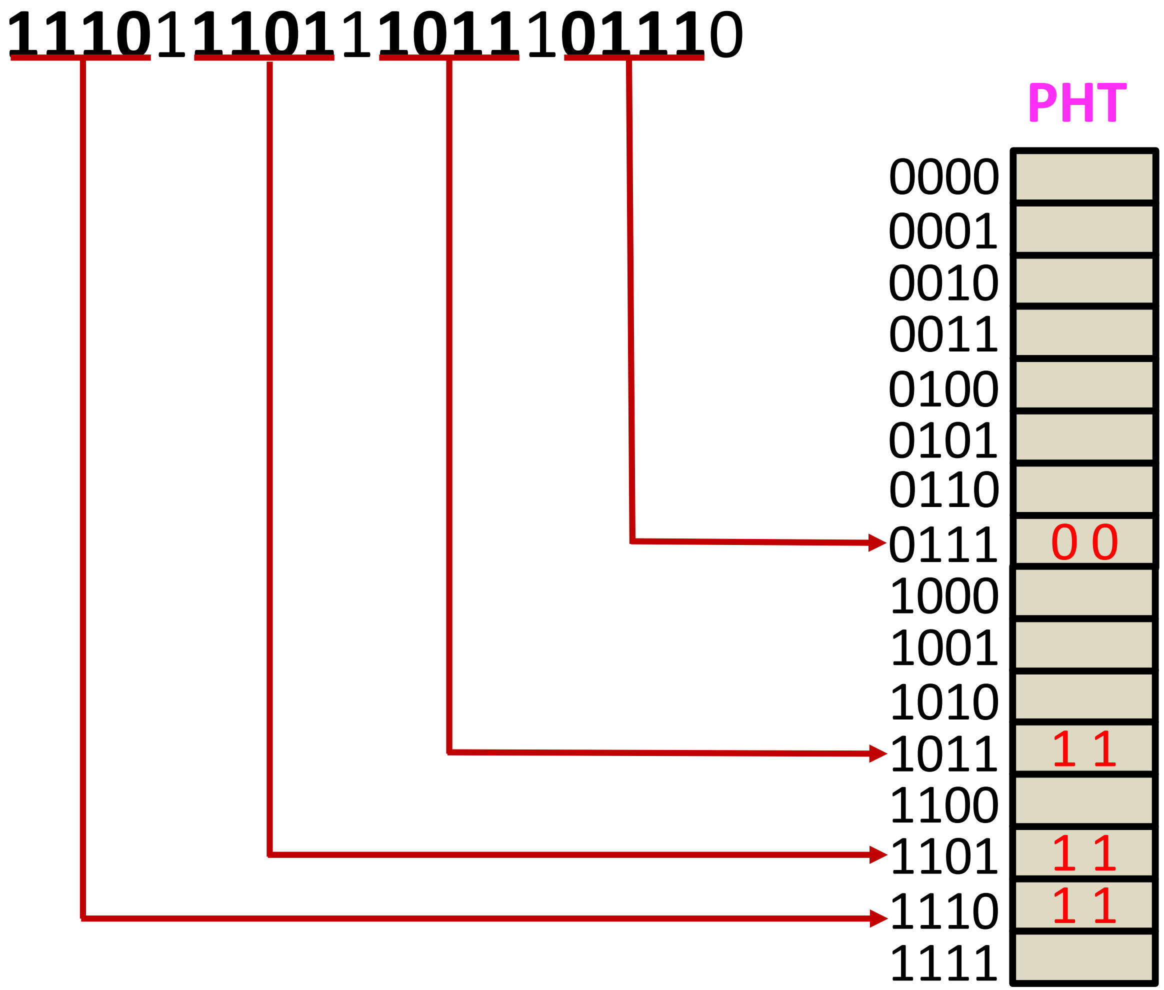
Intel P6

Uses Local History 2-level predictor with a history length of 4

… with "slight" complications



Loop Closing Example



11101**1101**1**1011**1**0111**0

PHT

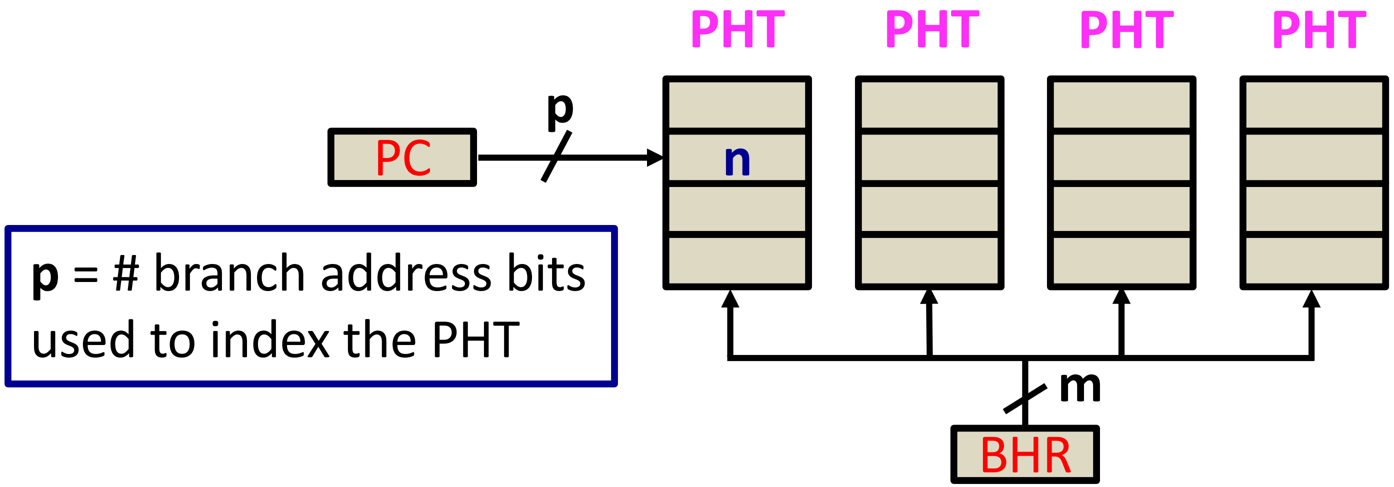
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	0 0
1000	
1001	
1010	
1011	1 1
1100	
1101	1 1
1110	1 1
1111	

The (m,n) View

So far, we have considered a monolithic view of the PHT

An alternative view is the (m,n) view shown below

There are 2^m PHTs, and each entry of the PHT is n-bits wide



p = # branch address bits used to index the PHT

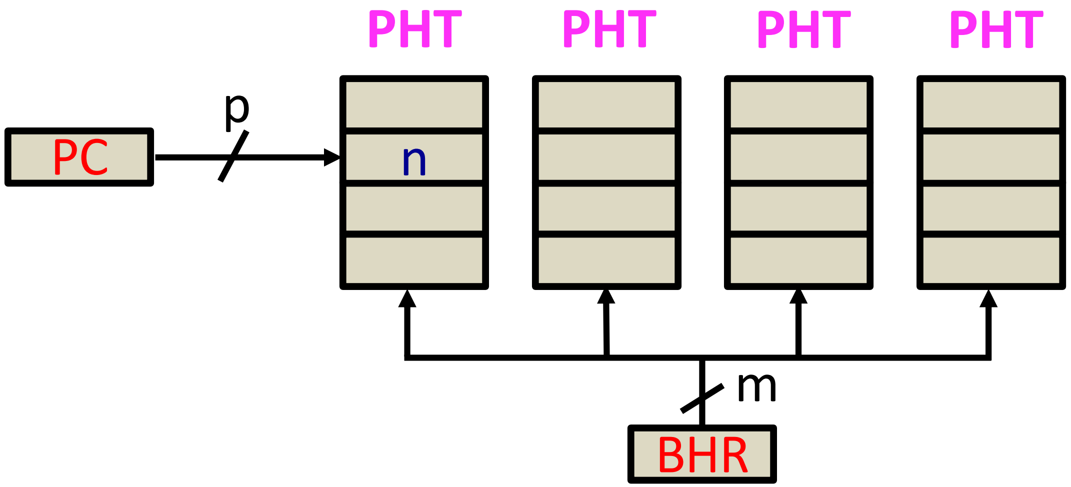
What is a (2,2) predictor? # PHTs? Smith$_k$ (k =?)

What is a (0,2) predictor? What about (0,1)?

Exercise

How many bits are in (0,2) branch predictor with 4K entries?

How many branch-selected entries are in a (2,2) predictor that has a total of 8K bits in the prediction buffer (monolithic PHT)?



Solution

How many bits are in (0,2) branch predictor with 4K entries?

- *A (0,2) branch predictor does not use any history at all since m = 0. The (0,2) predictor also uses 2-bit saturating counters. It is essentially a Smith*$_2$ *predictor. Since each entry of the branch prediction buffer is 2 bits (counter) and there are 4 K entries, the total number of bits in a (0,2) predictor equals 8 Kbits.*

Solution

How many branch-selected (i.e., 2^p) entries are in a (2,2) predictor that has a total of 8K bits in the prediction buffer (monolithic PHT)?

- *A (2,2) predictor uses 2 history bits and therefore the number of PHTs is equal to 4. The size of each PHT is then 2 Kbits. Since each entry of the PHT is 2-bits wide (i.e., n = 2), there are 1 K entries in the PHT. Thus, the branch selected entries in the PHT is equal to 1 K.*

Taxonomy of 2-Level Predictors

What type of branch historty to use?

- Global (G)
- Per-address (P)

Where does the non-history bits in the index come from?

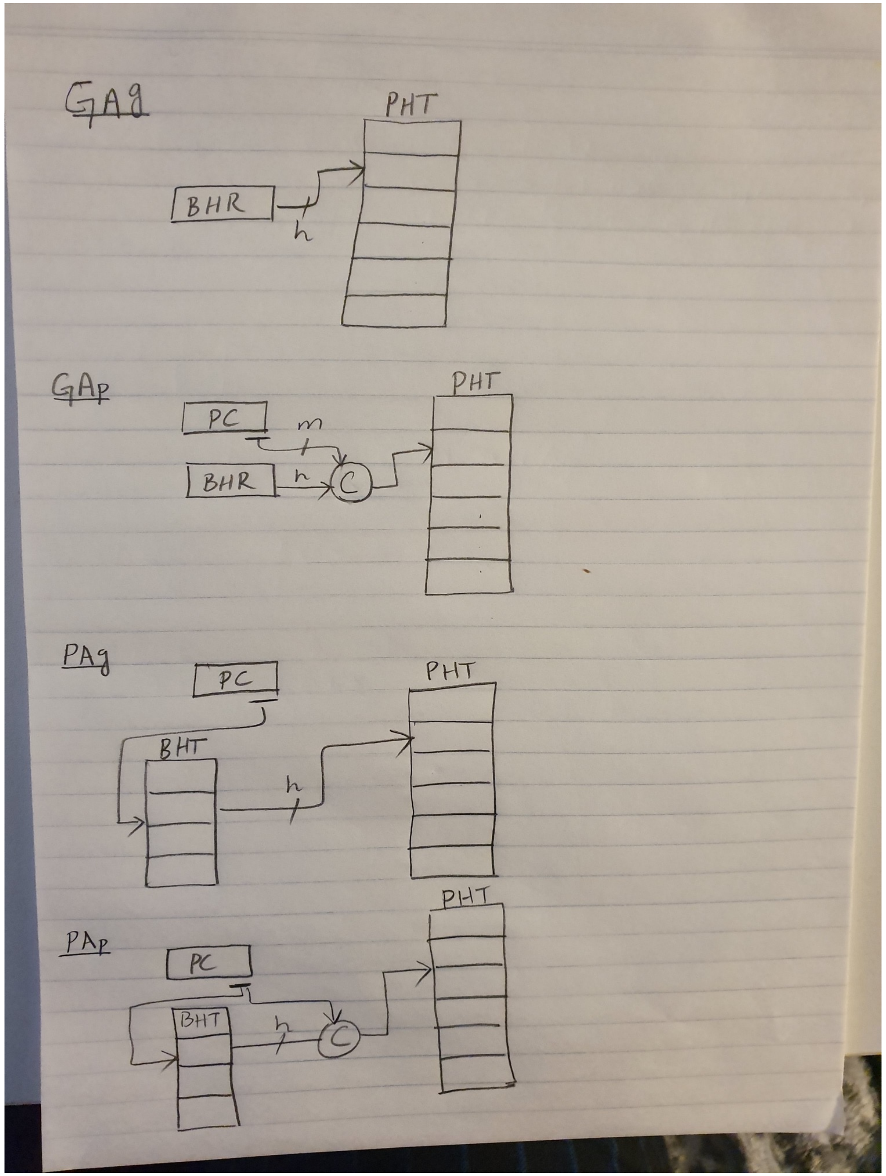
- Ignore PC, use BHR only (g) → *global branch history table* (gPHT)
- Use low-order bits of the address (p) → *per-add branch history table* (pPHT)

Four predictors

- GAg
- GAp
- PAg
- PAp

Four predictors

- GAg
- GAp
- PAg
- PAp



Index Sharing Predictors

Motivation: A drawback of 2-level algorithms is that the designer must make a tradeoff b/w BHR width and # branch address bits to index the PHT

Example: Consider the loop closing branch pattern again. The history length is 4 bits. With 4 bits, we can access 16 PHT entries. But there are only four frequently occuring patterns

Index formation in gselect and other 2-level correlating predictors introduces inefficiencies

McFarling’s gshare Predictor

McFarling (1993) proposed a variation of the global-history 2-level predictor

Key Insight: *Make better use of index bits by hashing the BHR and PC together to index the PHT*

Why does it works? *Combining BHR and PC contains more information due to the non-uniform distribution of branch addresses and history bits **(index sharing)***

Indexing: GAp vs. gshare

GAp: 4-bit index (2 bits from BHR and 2 low order bits from PC)

BHR	1	1	0	1
PC	0	1	1	0
Index	?	?	?	?

BHR	1	0	0	1
PC	1	0	1	0
Index	?	?	?	?

GAp

BHR	1	1	0	1
PC	0	1	1	0
Index	?	?	?	?

BHR	1	0	0	1
PC	1	0	1	0
Index	?	?	?	?

gshare

Indexing: GAp vs. gshare

GAp: 4-bit index (2 bits from BHR and 2 low order bits from PC)

BHR	1	1	0	1
PC	0	1	1	0
Index	1	0	0	1

BHR	1	0	0	1
PC	1	0	1	0
Index	1	0	0	1

GAp

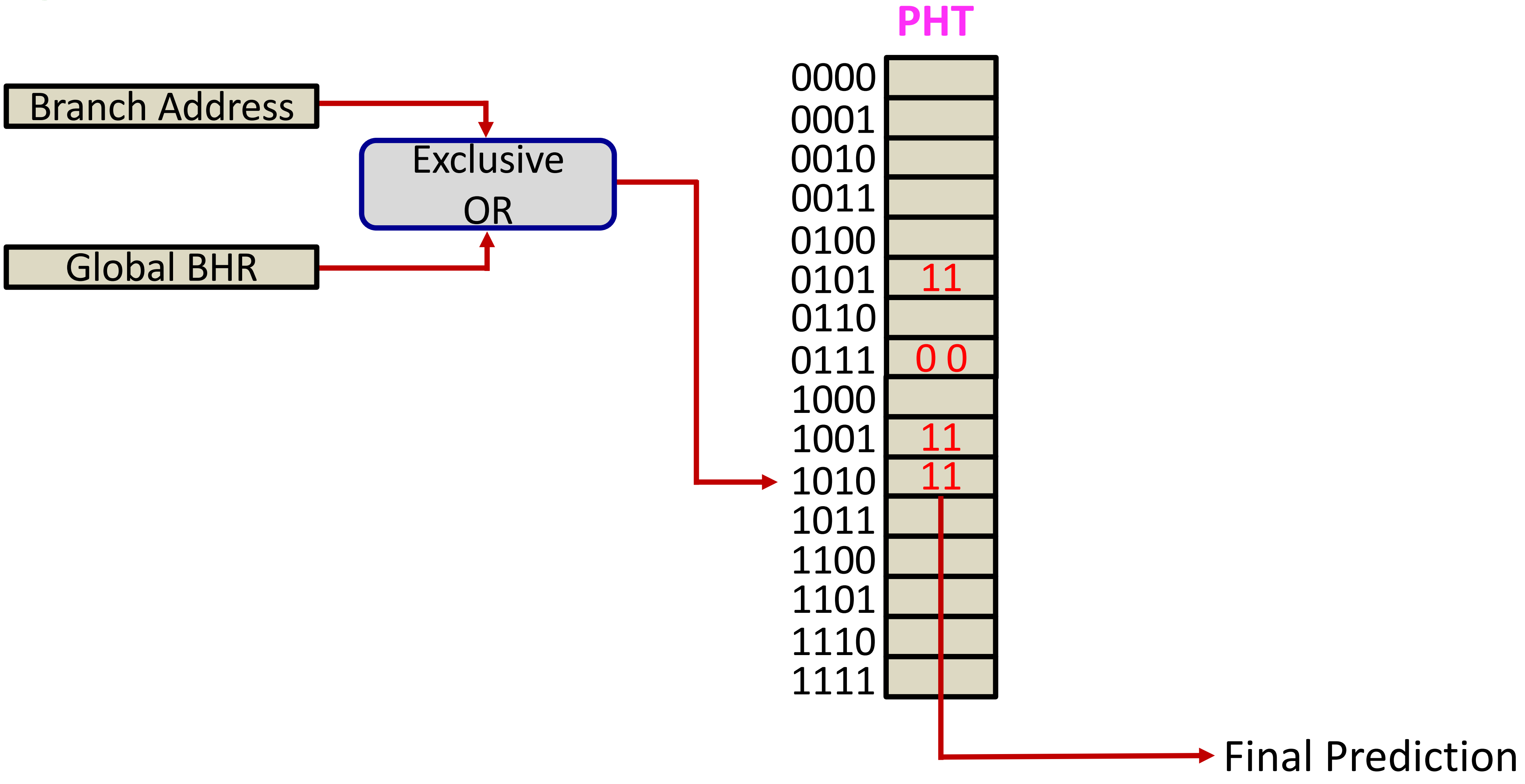
Information loss during index formation

BHR	1	1	0	1
PC	0	1	1	0
Index	1	0	1	1

BHR	1	0	0	1
PC	1	0	1	0
Index	0	0	1	1

gshare

gshare Circuit



Branch Address

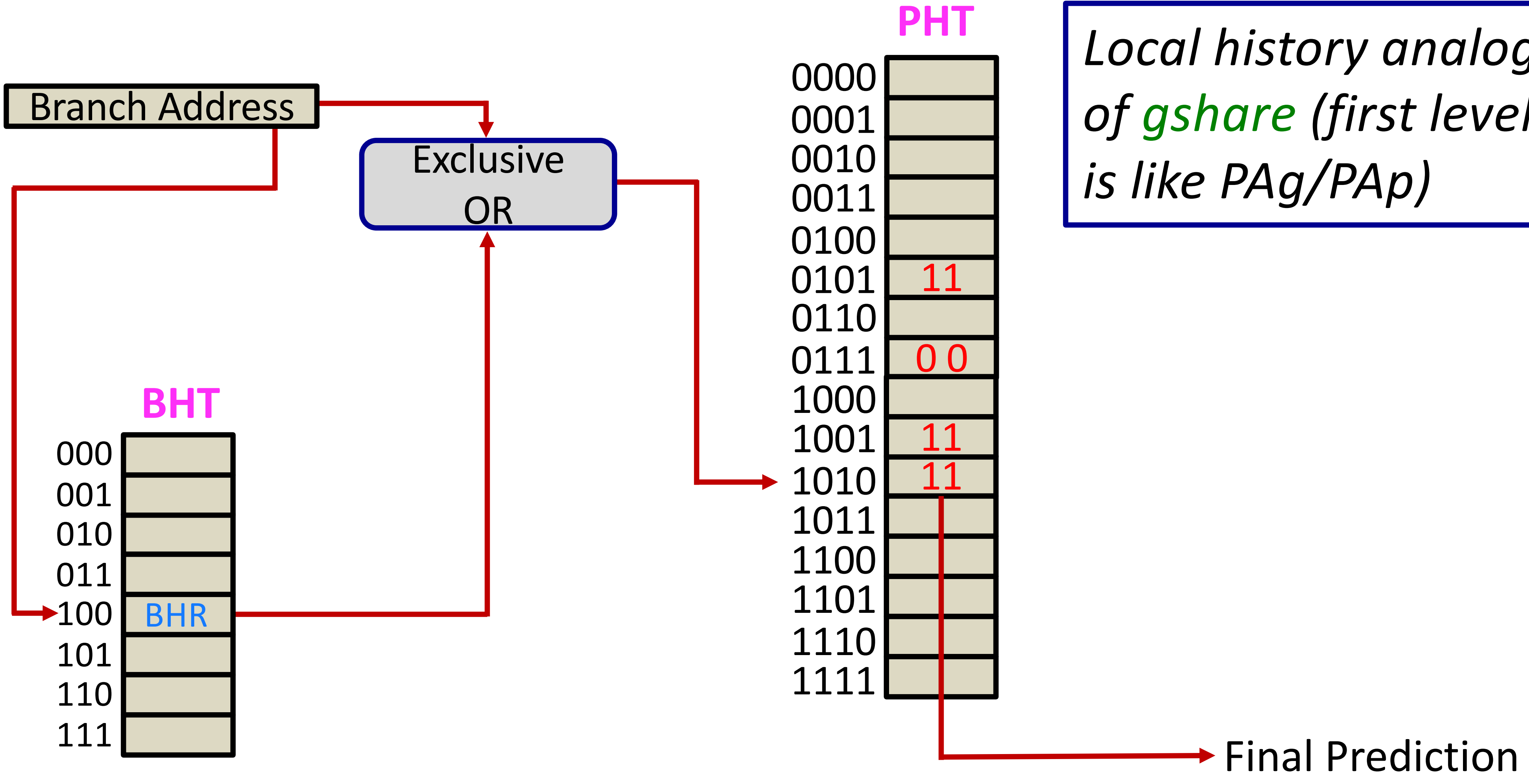
Global BHR

Exclusive OR

	PHT
0000	
0001	
0010	
0011	
0100	
0101	11
0110	
0111	0 0
1000	
1001	11
1010	11
1011	
1100	
1101	
1110	
1111	

Final Prediction

pshare Circuit



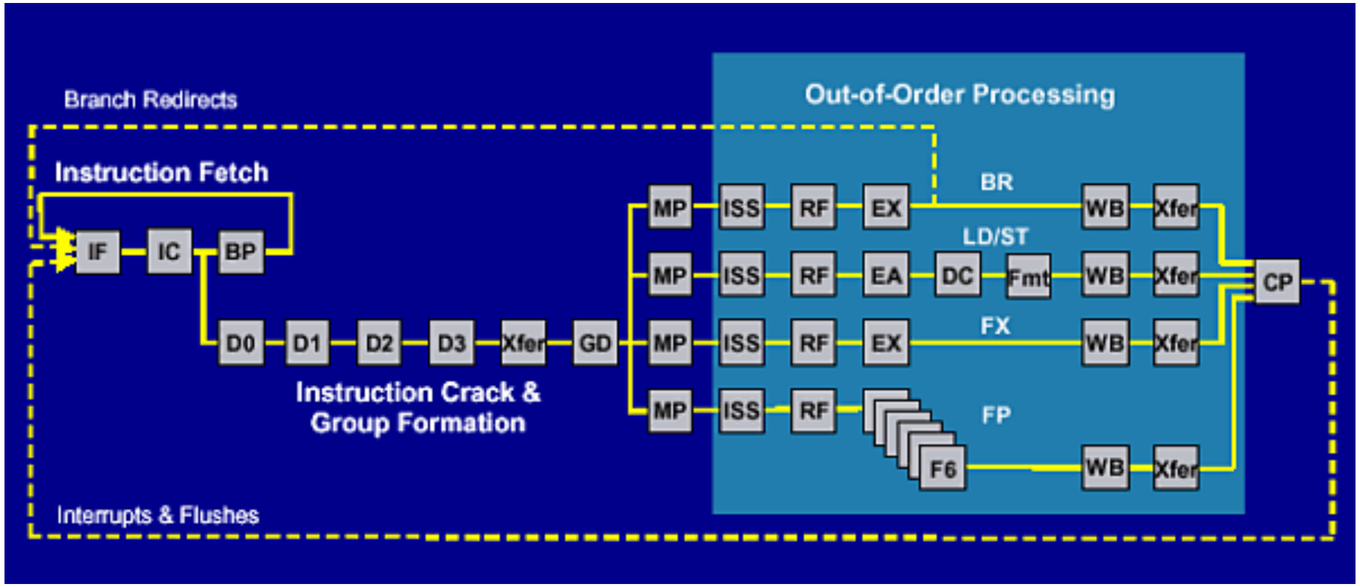
Local history analog of gshare (first level is like PAg/PAp)

More Real-World Examples

IBM Power 4

- Global history predictor
- 11-bit BHR
- 16,384 entry PHT
- 2nd level uses Smith$_1$

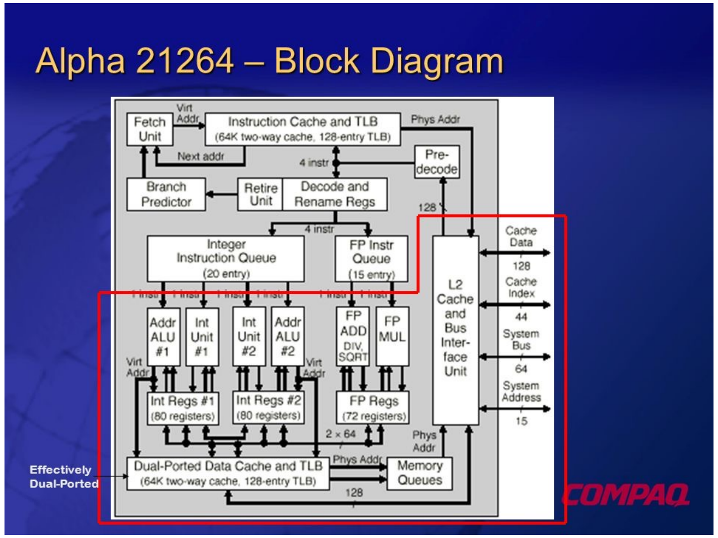
Power 4 Core: *http://ixbtlabs.com/articles/ibmpower4/*



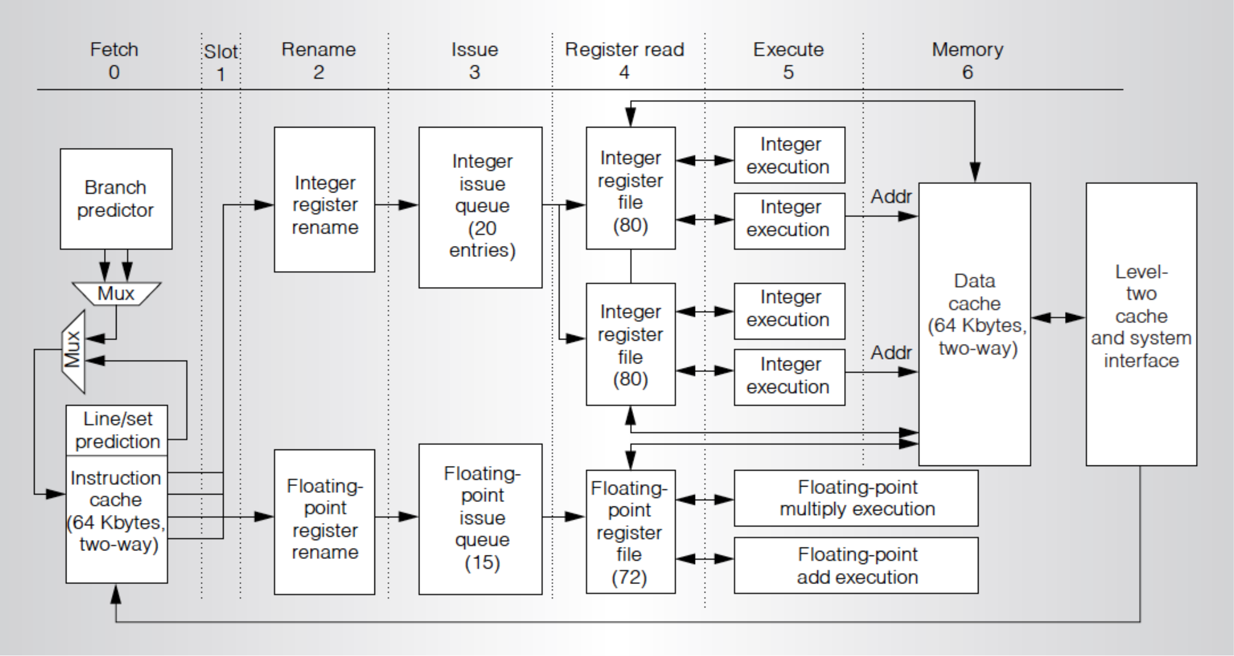
More Real-World Examples

Alpha 21264

- → Global history predictor
- → 12-bit BHR
- → 4096 entry PHT



Alpha 21264 – Block Diagram



R. E. Kessler, "*The Alpha 21264 Microprocessor,*" *IEEE MICRO*

Questions to Ponder

```
int random = rand() % 100;  /* naïve/poor way to get a random no. */
if  (some_condition_involving_random)
{
        do something;
}
```

How should we deal with probalistic branches (*increasingly common*)?

→ *On Alpha 21264, the misprediction penalty is not 1 cycle, but 14 cycles*

→ Is it worth flushing the pipeline for a misprediction on a probablistic branch? What does the program semantics demands? What is programmer’s intention?

Questions to Ponder: Role of ML

- Should we pay attention to machine learning for building more accurate branch predictors? What has changed in recent times?
- Movement away from desktop to the cloud
  - Are the workloads in the cloud converging to a few classes? Is *user diversity* equal to *workload diversity*? How can we exploit these trends for optimizing processors?
- Big *Profiling* Data, Huge and diverse datasets

Next Two Weeks

Tournament Predictor

Tagged Hybrid Predictor (TAGE)

The Perceptron Predictor (time permiting)

Branch Target Buffer

Exception Handling (MIPS Pipeline)

Week 6: In-Order to Out-of-Order Transformation

Plan

Week 4: Data and branch hazards, branch prediction

Week 5: Correlating predictors (via an example)

Week 5: Hybrid, Neural, and Tag-based predictors

Week 5: BTBs, Exception handling, Multiscalar Pipelines

Homework Solution

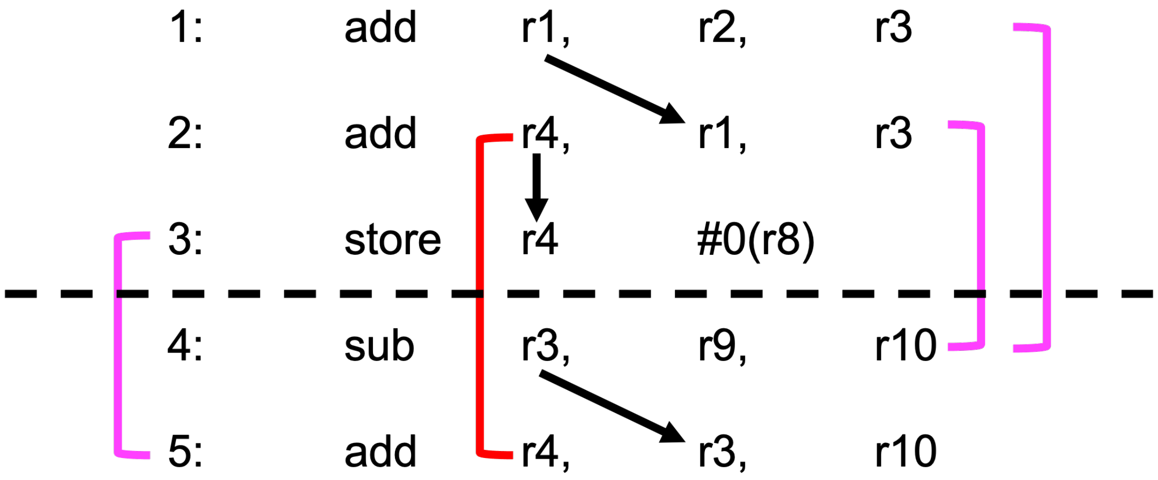
Dependence (hazard)

Output **(WAW)**
Anti **(WAR)**
True/Data **(RAW)**

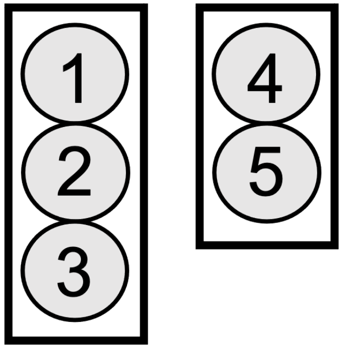
Hazards:

Write-After-Write
Write-After-Read
Read-After-Write

- i2 reads after i1 writes
- i5 writes after i3 has read



```
1:   add    r1,   r2,     r3
2:   add    r4,   r1,     r3
3:   store  r4    #0(r8)
4:   sub    r3,   r9,     r10
5:   add    r4,   r3,     r10
```



Accuracy Comparison

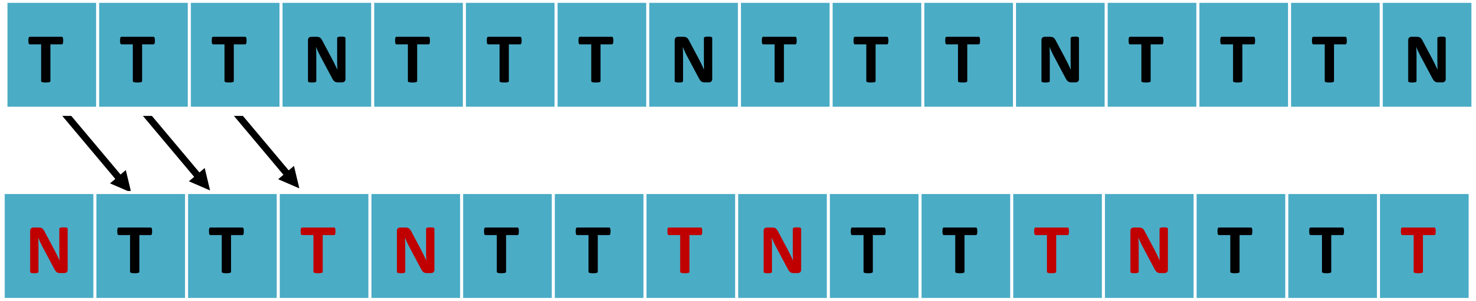
T	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**

Loop Pattern

Compare the accuracy of three predictors

	Smith$_1$	**Smith$_2$**	**PAg**
Accuracy	?	?	?

Smith$_1$

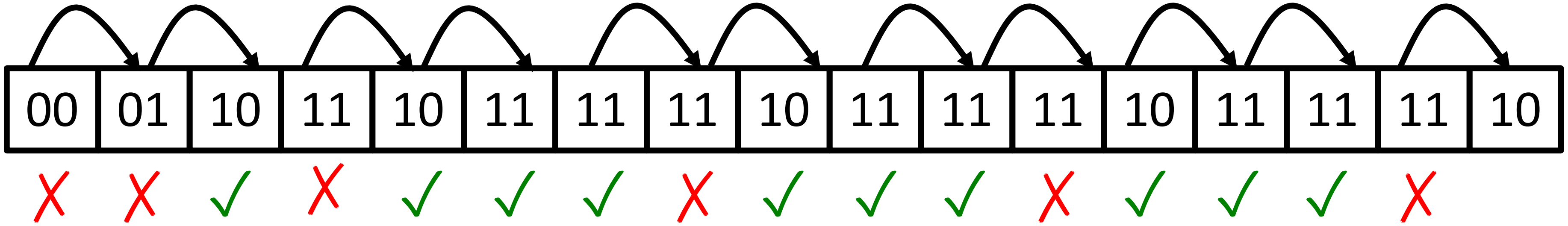


T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N	Actual Direction
N	T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	State/Prediction

	Smith$_1$	Smith$_2$	PAg
Accuracy	50%	?	?

Smith$_2$

T	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**	Actual Direction



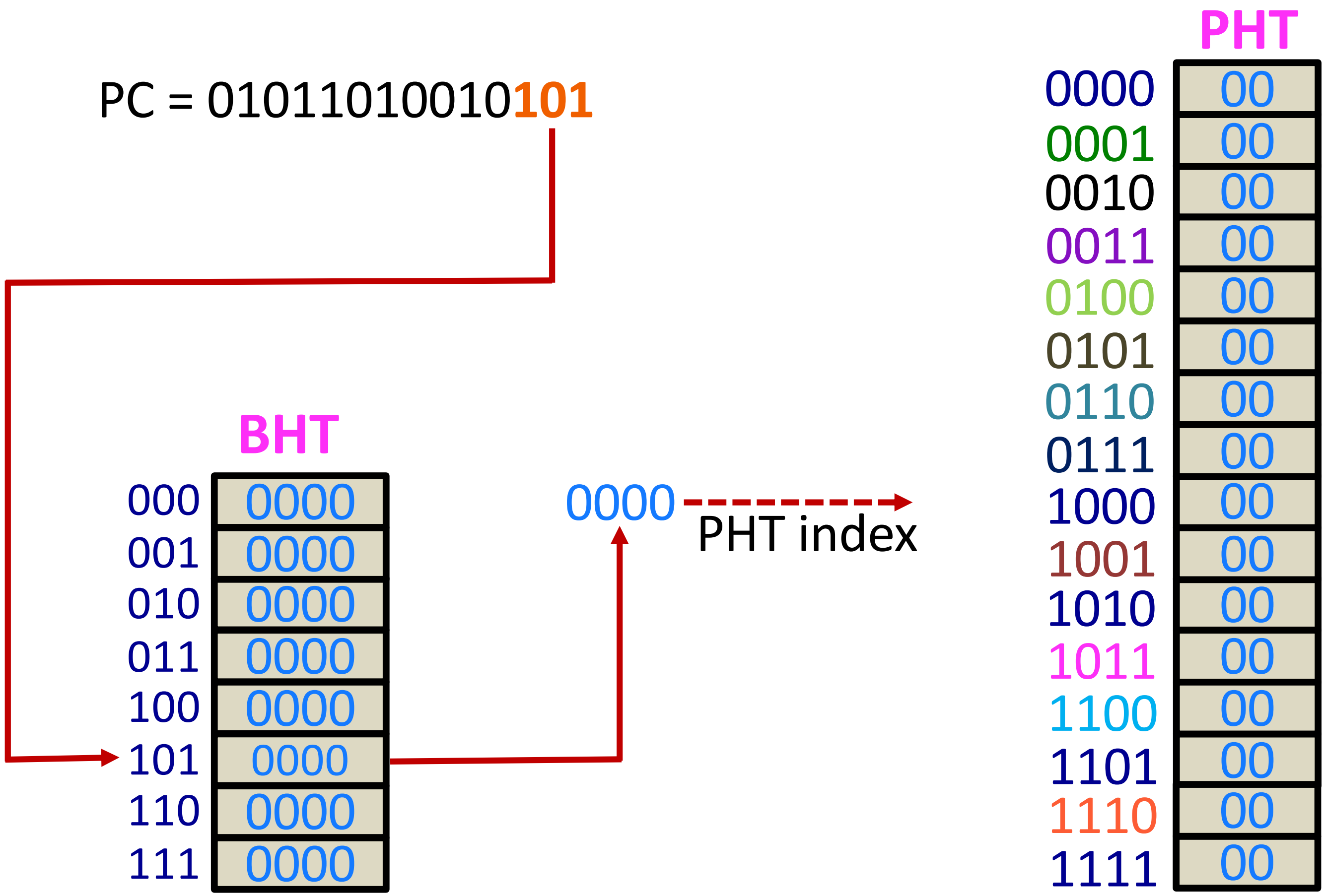
00	01	10	11	10	11	11	11	10	11	11	11	10	11	11	11	10
✗	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗	

	Smith$_1$	**Smith$_2$**	**PAg**
Accuracy	50%	62.5%	?

1st iteration: 25%
Steady state: 75%

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0000
110	0000
111	0000

0000

PHT index

PHT

0000	00
0001	00
0010	00
0011	00
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00
1110	00
1111	00

Iteration # 1

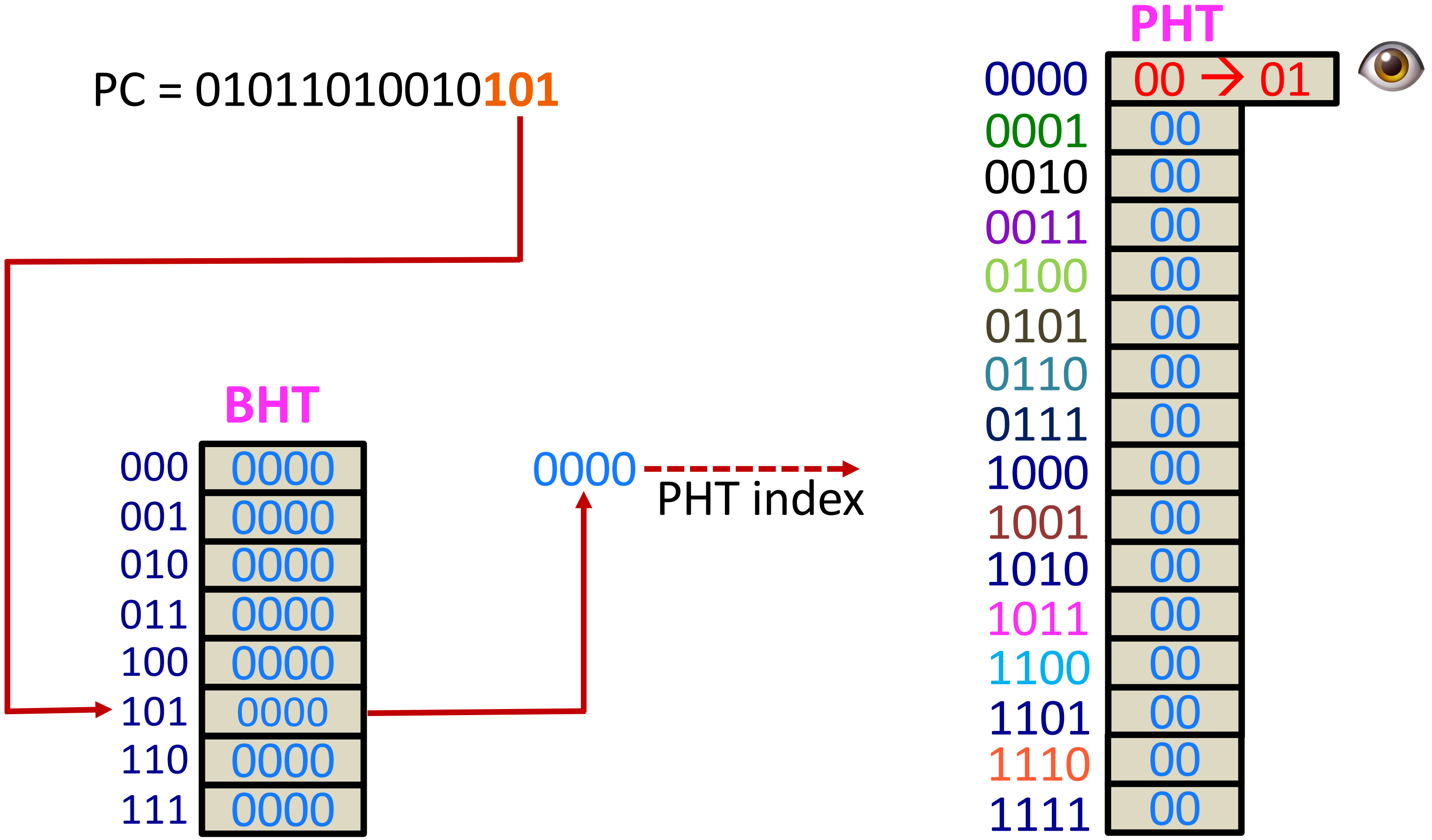
Prediction: ?

Outcome: ?

Score: ?

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0000
110	0000
111	0000

0000 PHT index

PHT

0000	00 → 01
0001	00
0010	00
0011	00
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00
1110	00
1111	00

Iteration # 1

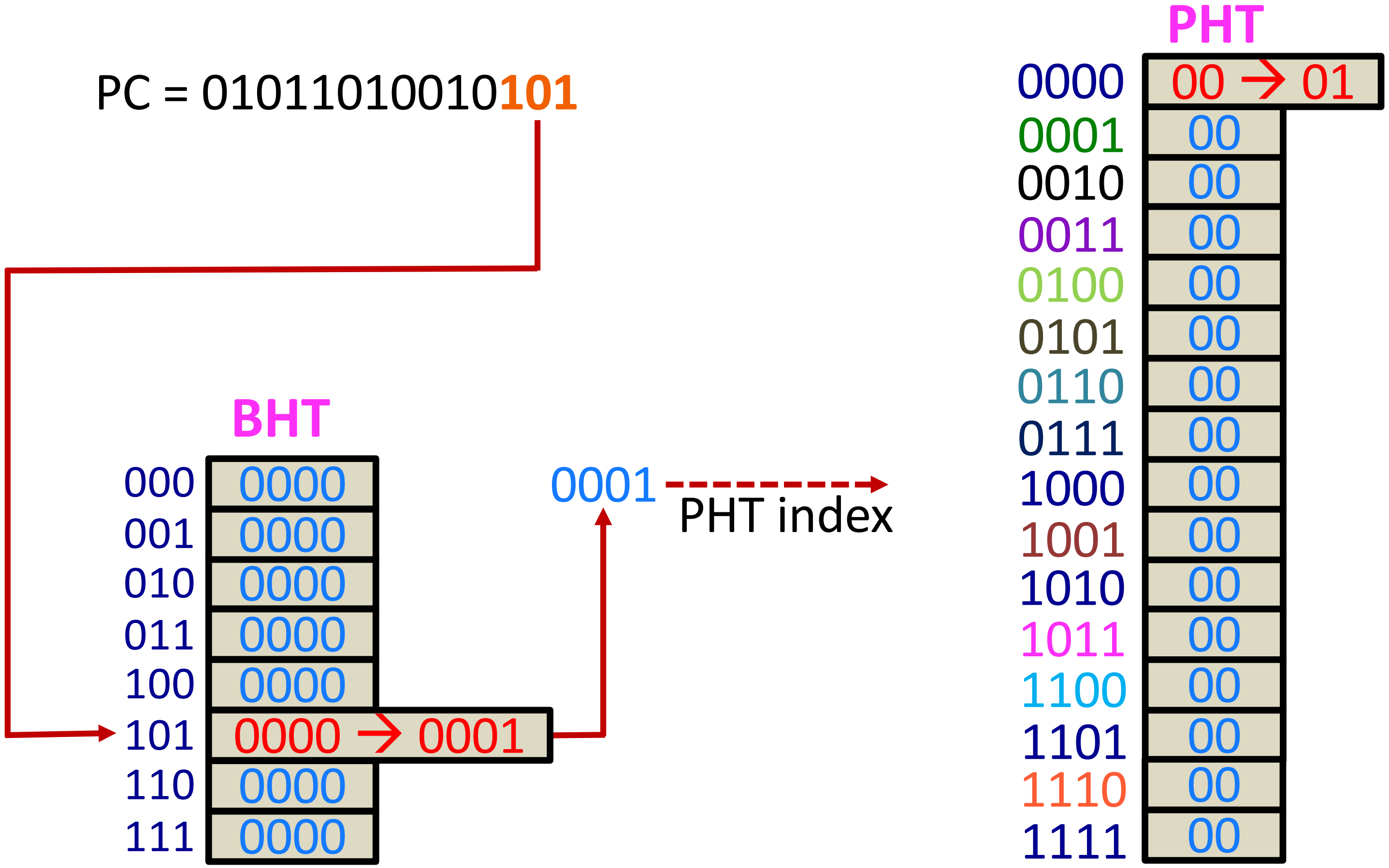
Prediction: NT

Outcome: T

Score: 0/1

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010101

BHT	
000	0000
001	0000
010	0000
011	0000
100	0000
101	0000 → 0001
110	0000
111	0000

0001 PHT index

PHT	
0000	00 → 01
0001	00
0010	00
0011	00
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00
1110	00
1111	00

Iteration # 1

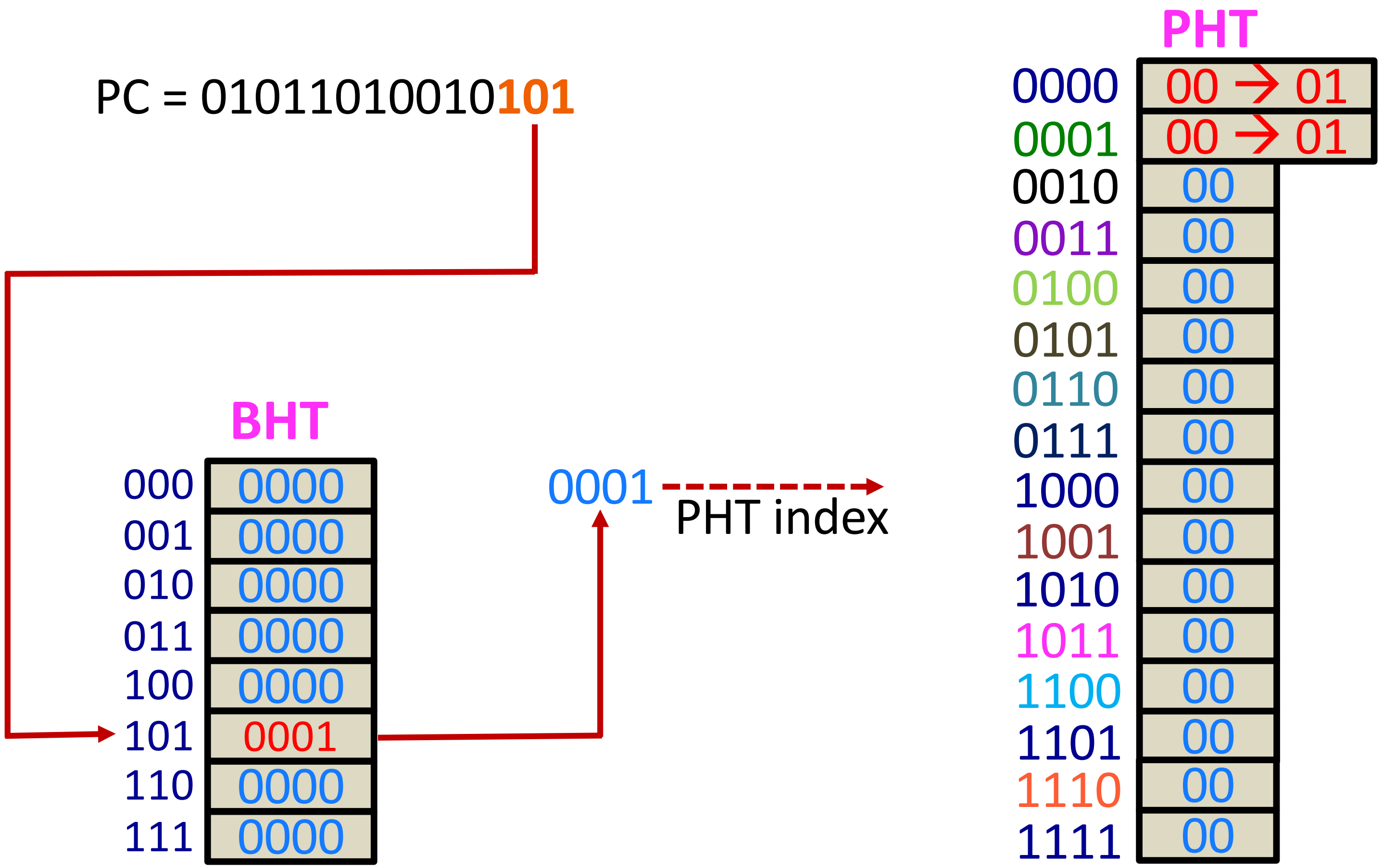
Prediction: NT

Outcome: T

Score: 0/1

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0001
110	0000
111	0000

0001 PHT index

PHT

0000	00 → 01
0001	00 → 01
0010	00
0011	00
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00
1110	00
1111	00

Iteration # 1

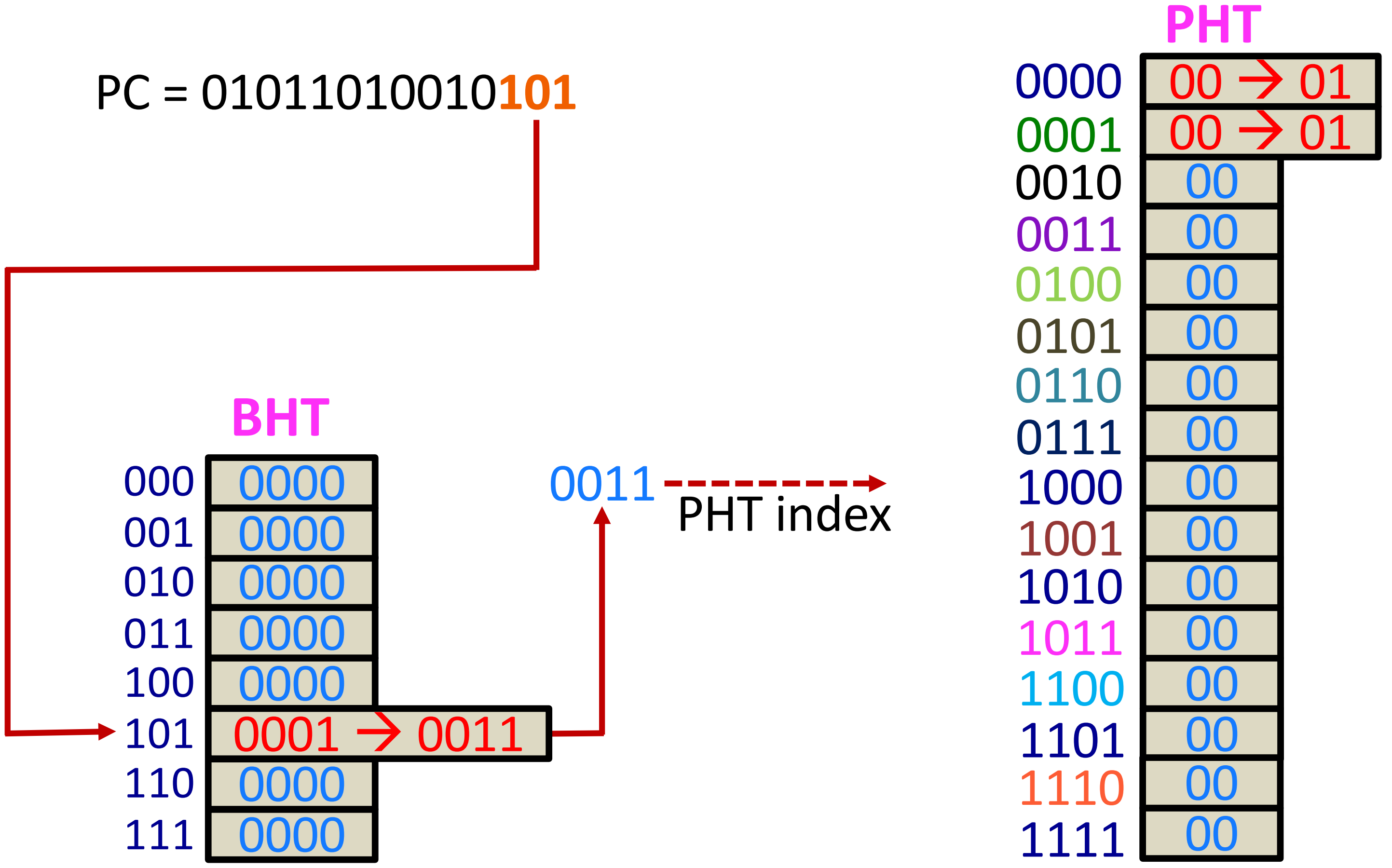
Prediction: NT

Outcome: T

Score: 0/2

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0001 → 0011
110	0000
111	0000

0011 PHT index

PHT

0000	00 → 01
0001	00 → 01
0010	00
0011	00
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00
1110	00
1111	00

Iteration # 1

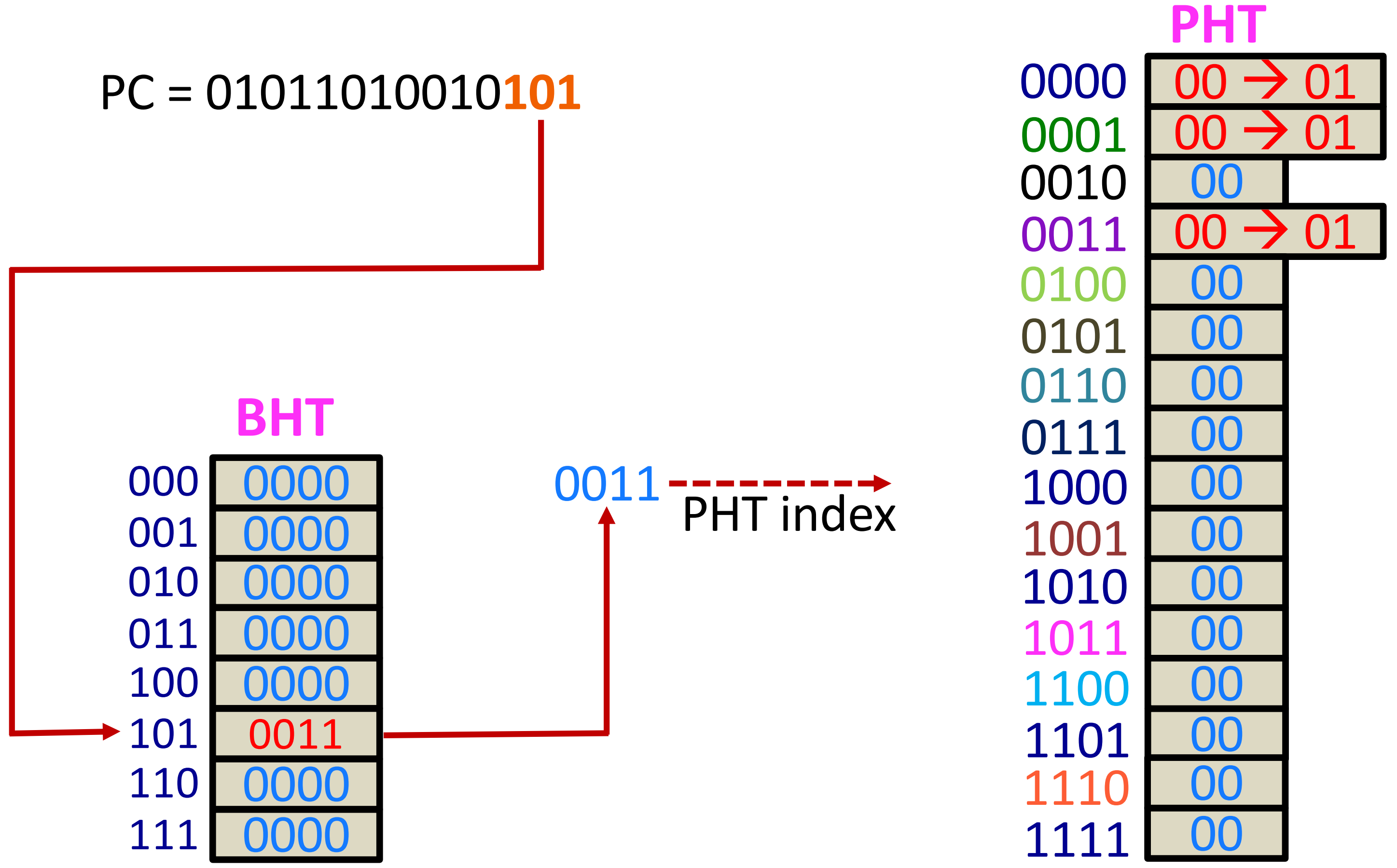
Prediction: NT

Outcome: T

Score: 0/2

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010101

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0011
110	0000
111	0000

0011

PHT index

PHT

0000	00 → 01
0001	00 → 01
0010	00
0011	00 → 01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00
1110	00
1111	00

Iteration # 1

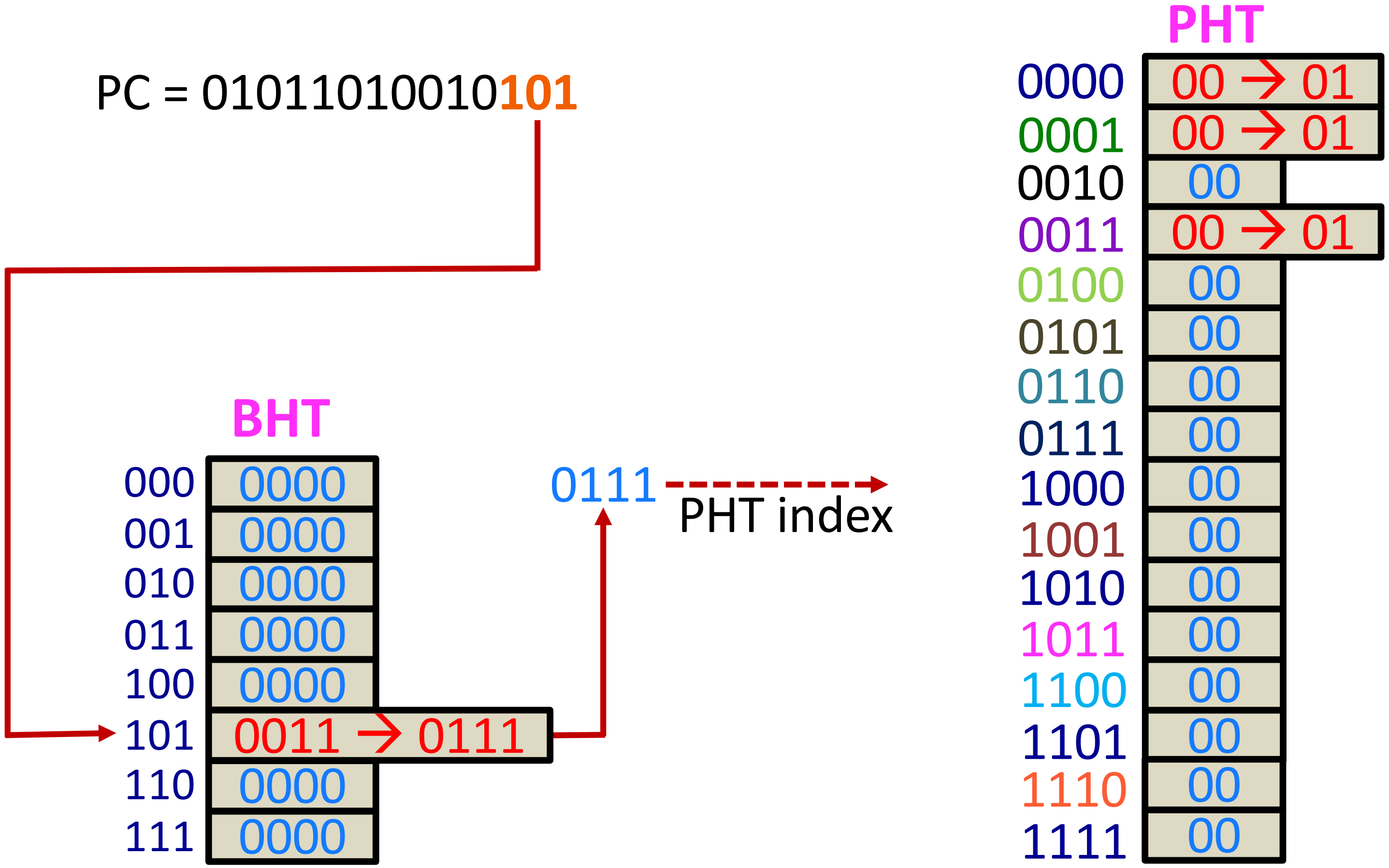
Prediction: NT

Outcome: T

Score: 0/3

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N


PC = 01011010010101

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0011 → 0111
110	0000
111	0000

0111 PHT index

PHT

0000	00 → 01
0001	00 → 01
0010	00
0011	00 → 01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00
1110	00
1111	00

Iteration # 1

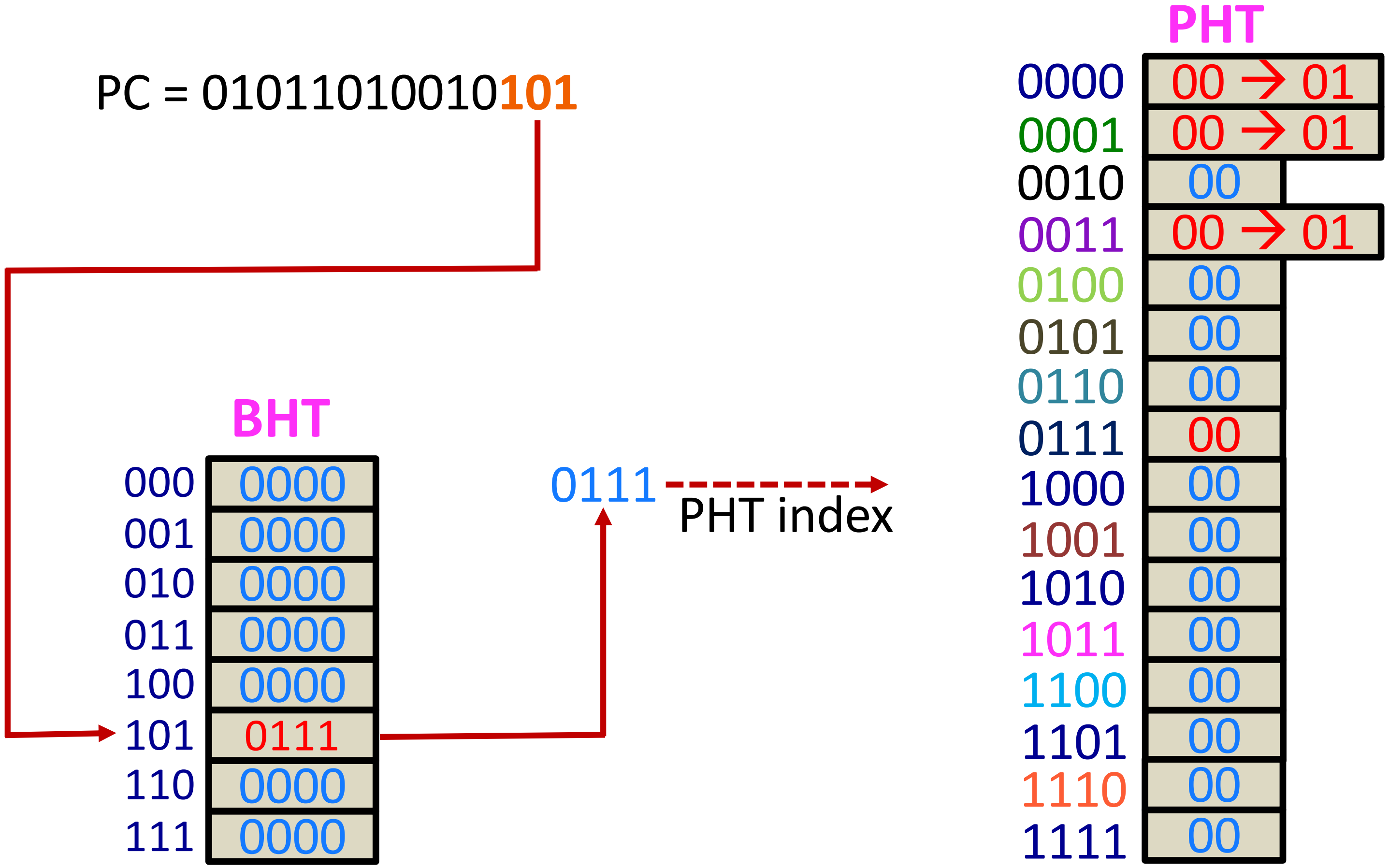
Prediction: NT

Outcome: T

Score: 0/3

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010101

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0111
110	0000
111	0000

0111 PHT index

PHT

0000	00 → 01
0001	00 → 01
0010	00
0011	00 → 01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00
1110	00
1111	00

Iteration # 1

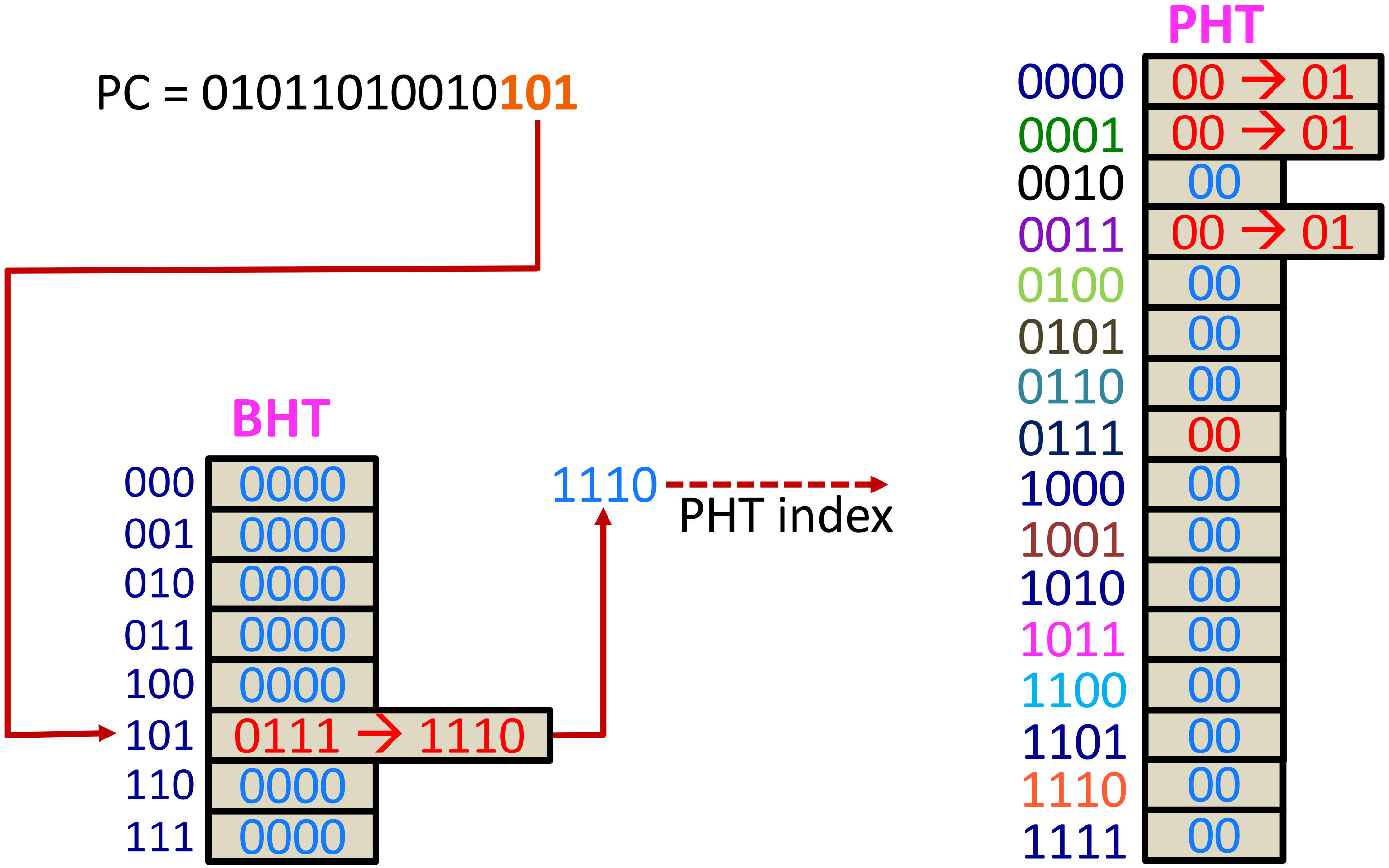
Prediction: NT

Outcome: NT

Score: 1/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0111 → 1110
110	0000
111	0000

1110 PHT index

PHT

0000	00 → 01
0001	00 → 01
0010	00
0011	00 → 01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00
1110	00
1111	00

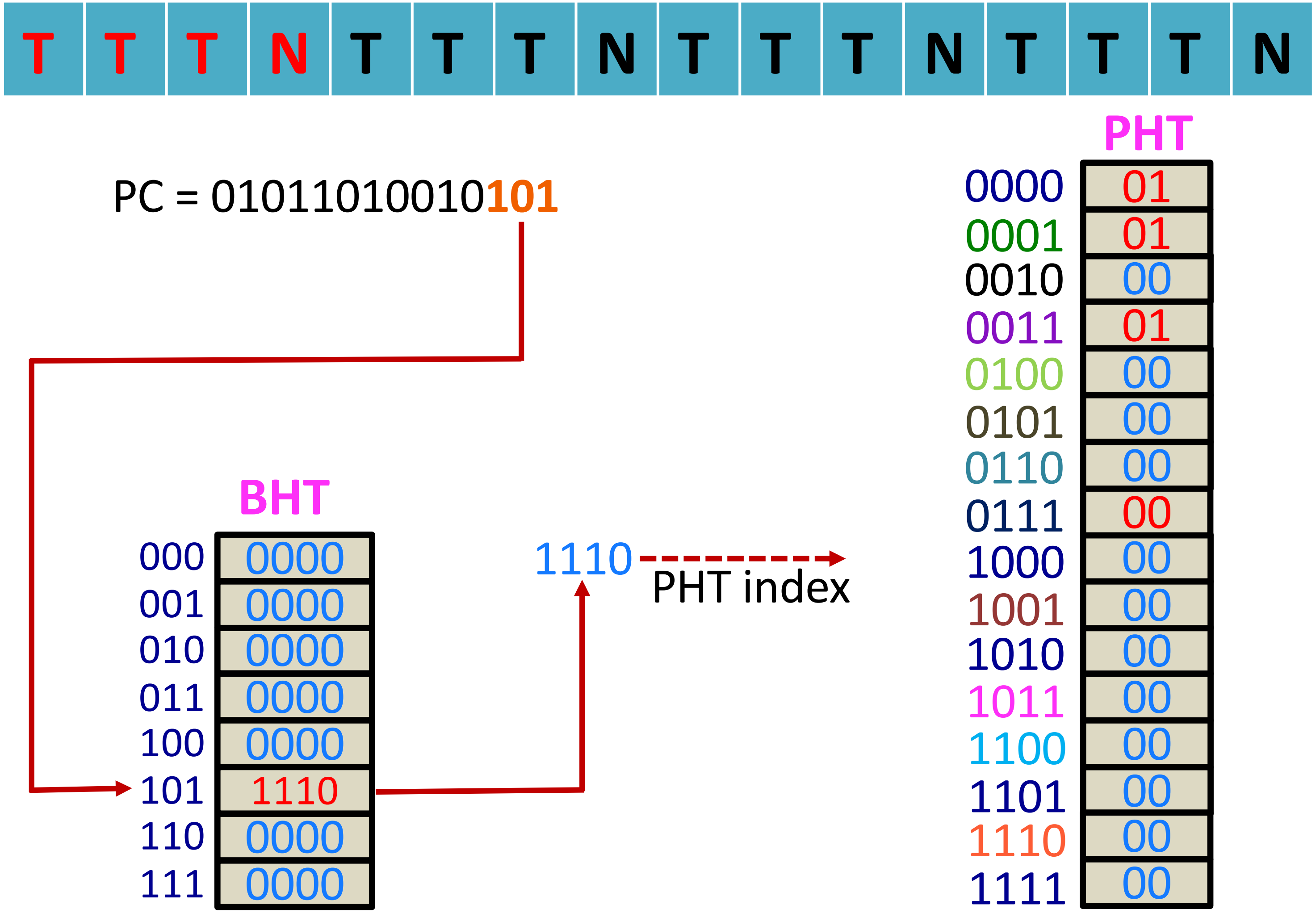
Iteration # 1

Prediction: NT

Outcome: NT

Score: 1/4

2-Level PAg predictor



T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N

PC = 01011010010101

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1110
110	0000
111	0000

1110 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00
1110	00
1111	00

Iteration # 1

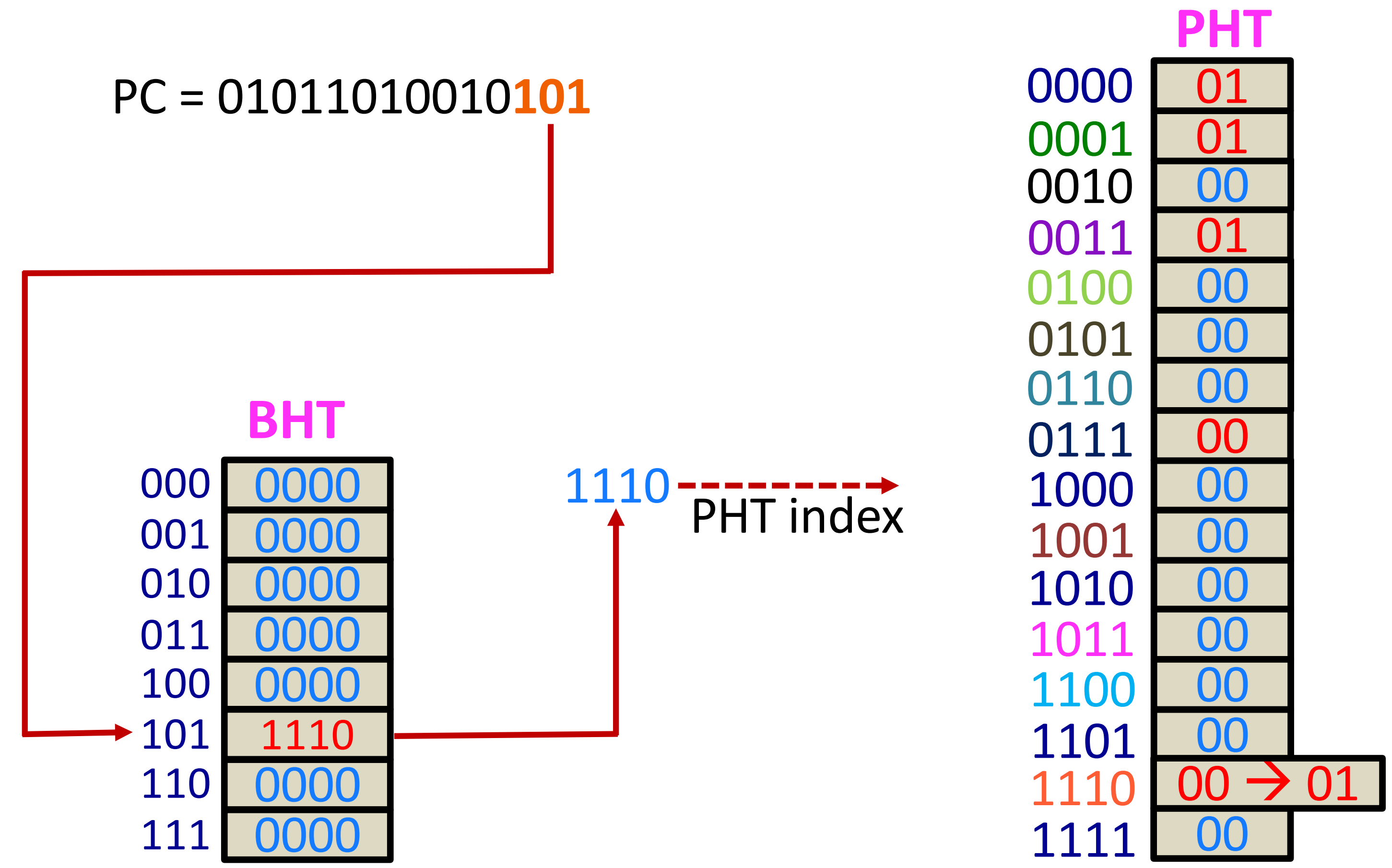
Prediction: NT

Outcome: NT

Score: 1/4

2-Level PAg predictor

T	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1110
110	0000
111	0000

1110 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00
1110	00 → 01
1111	00

Iteration # 2

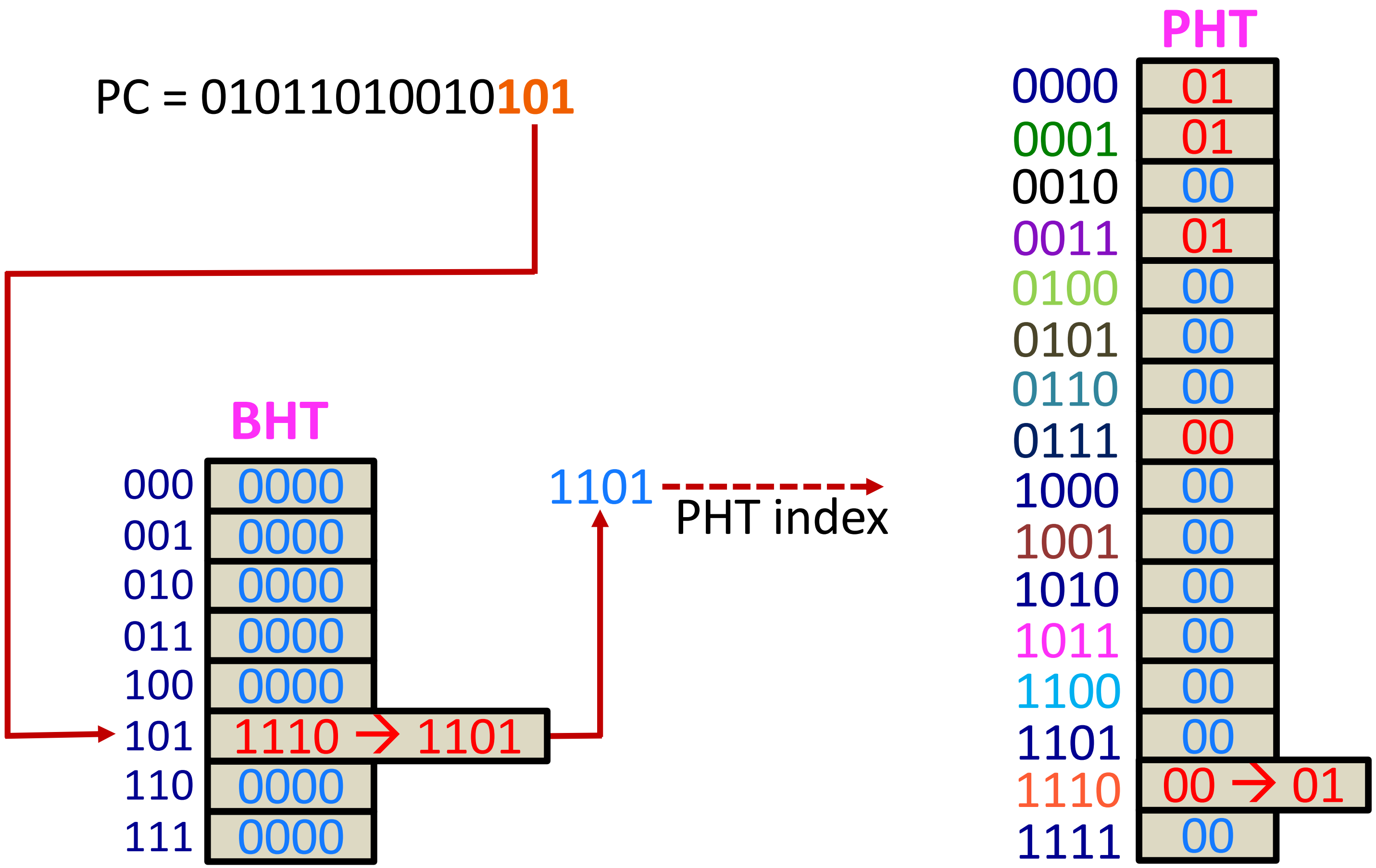
Prediction: NT

Outcome: T

Score: 0/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1110 → 1101
110	0000
111	0000

1101 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00
1110	00 → 01
1111	00

Iteration # 2

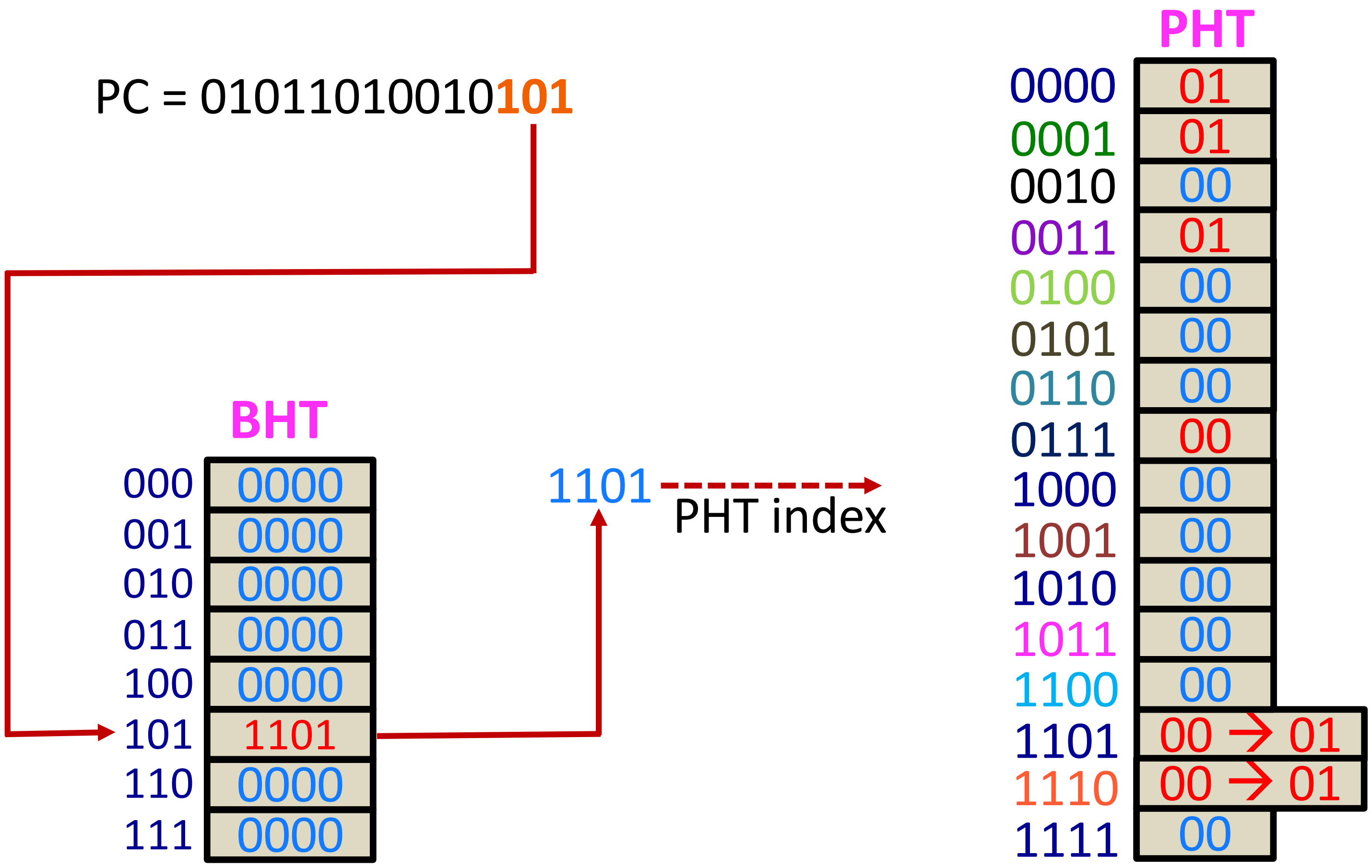
Prediction: NT

Outcome: T

Score: 0/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1101
110	0000
111	0000

1101

PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00 → 01
1110	00 → 01
1111	00

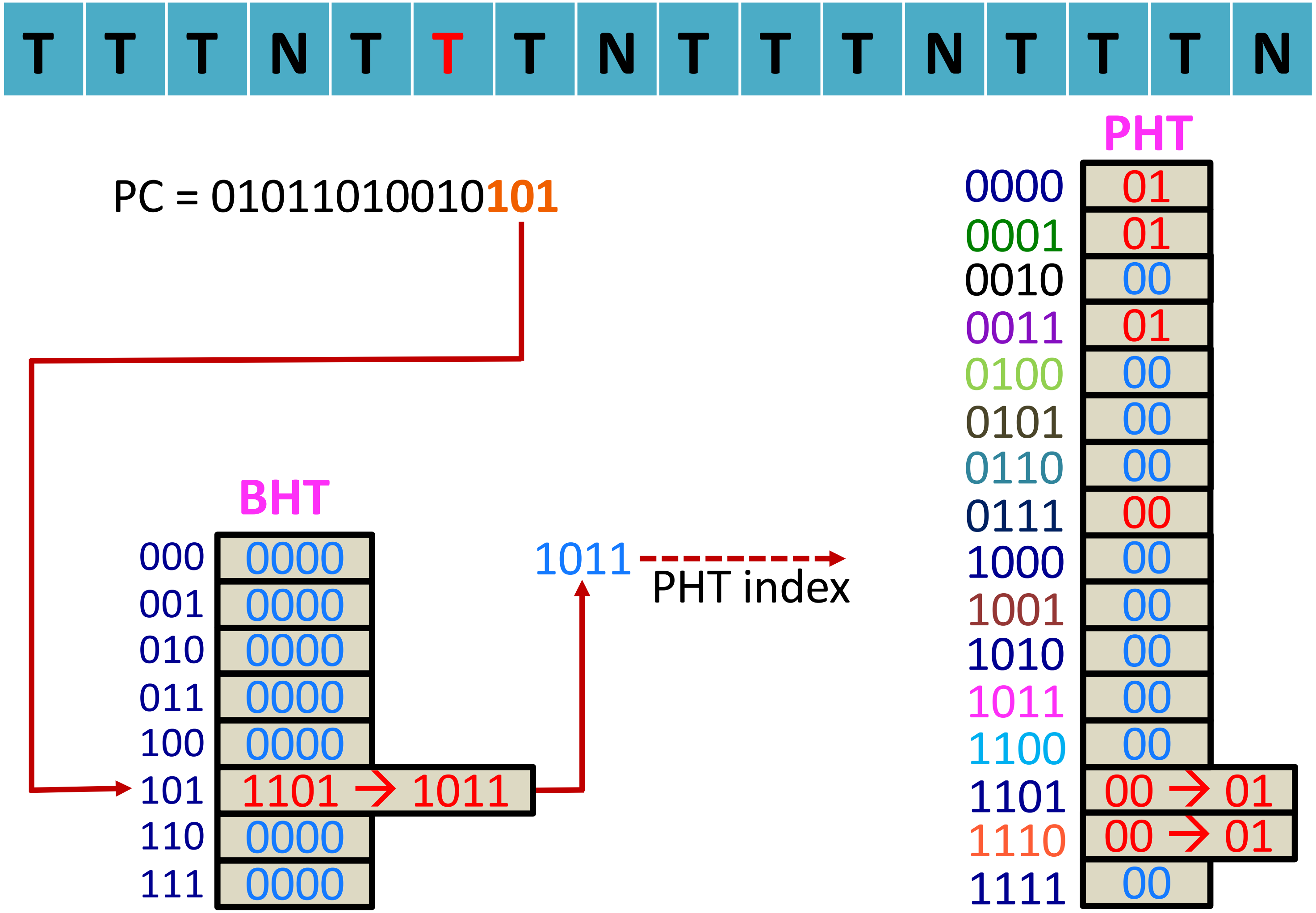
Iteration # 2

Prediction: NT

Outcome: T

Score: 0/4

2-Level PAg predictor



T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N

PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1101 → 1011
110	0000
111	0000

1011 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00
1100	00
1101	00 → 01
1110	00 → 01
1111	00

Iteration # 2

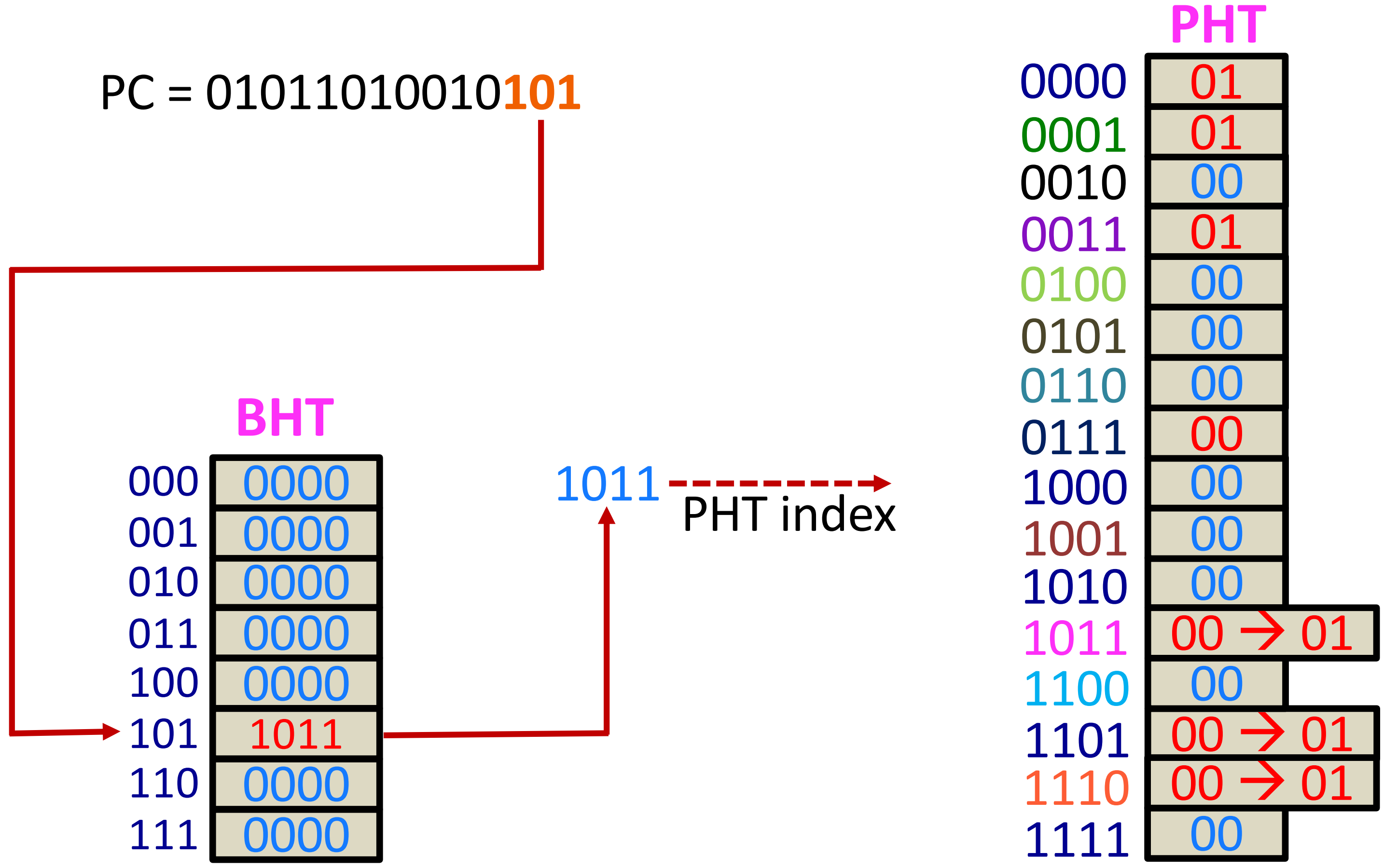
Prediction: NT

Outcome: T

Score: 0/4

2-Level PAg predictor

T	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1011
110	0000
111	0000

1011 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00 → 01
1100	00
1101	00 → 01
1110	00 → 01
1111	00

Iteration # 2

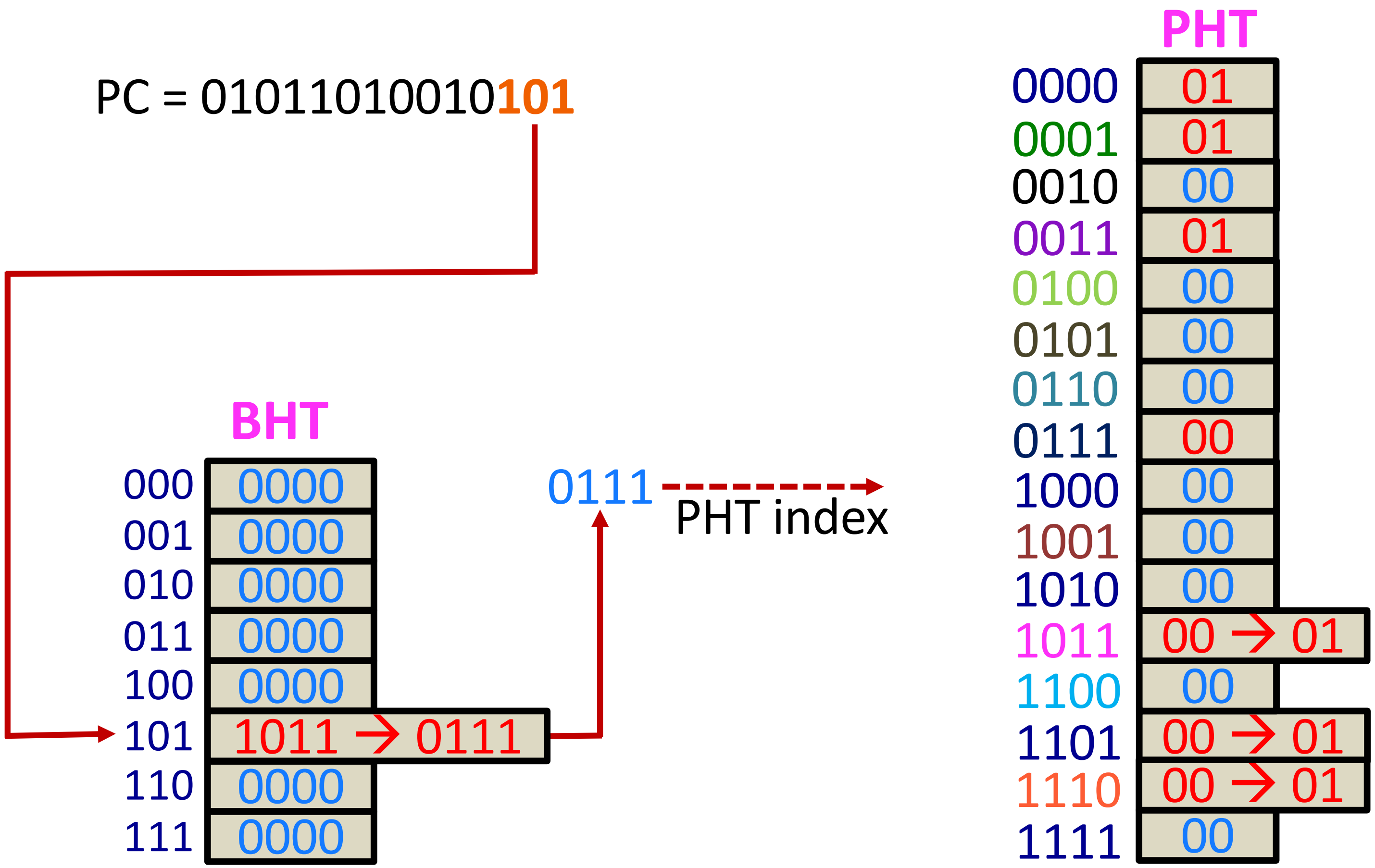
Prediction: NT

Outcome: T

Score: 0/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1011 → 0111
110	0000
111	0000

0111 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00 → 01
1100	00
1101	00 → 01
1110	00 → 01
1111	00

Iteration # 2

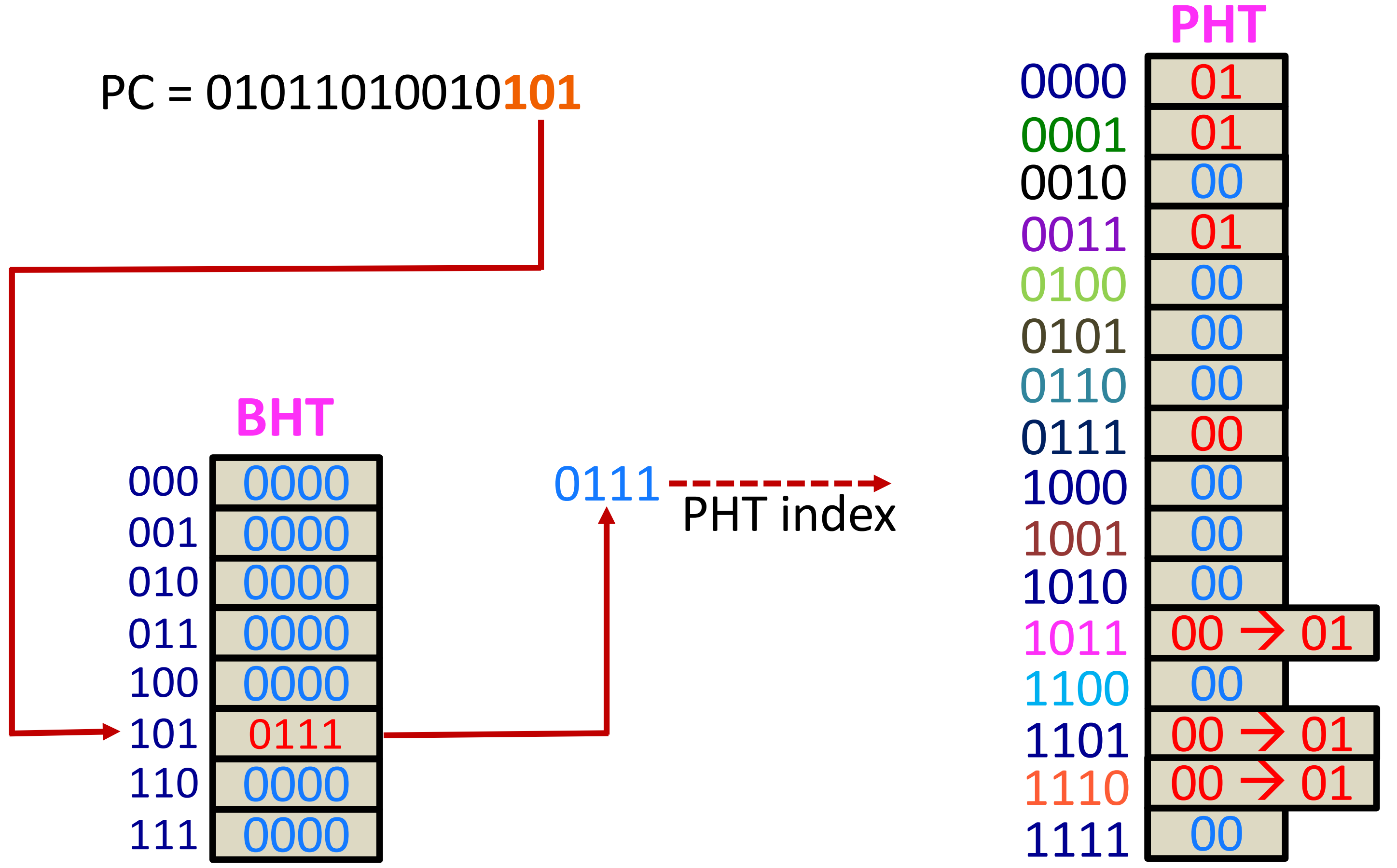
Prediction: NT

Outcome: T

Score: 0/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010101

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0111
110	0000
111	0000

0111 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00 → 01
1100	00
1101	00 → 01
1110	00 → 01
1111	00

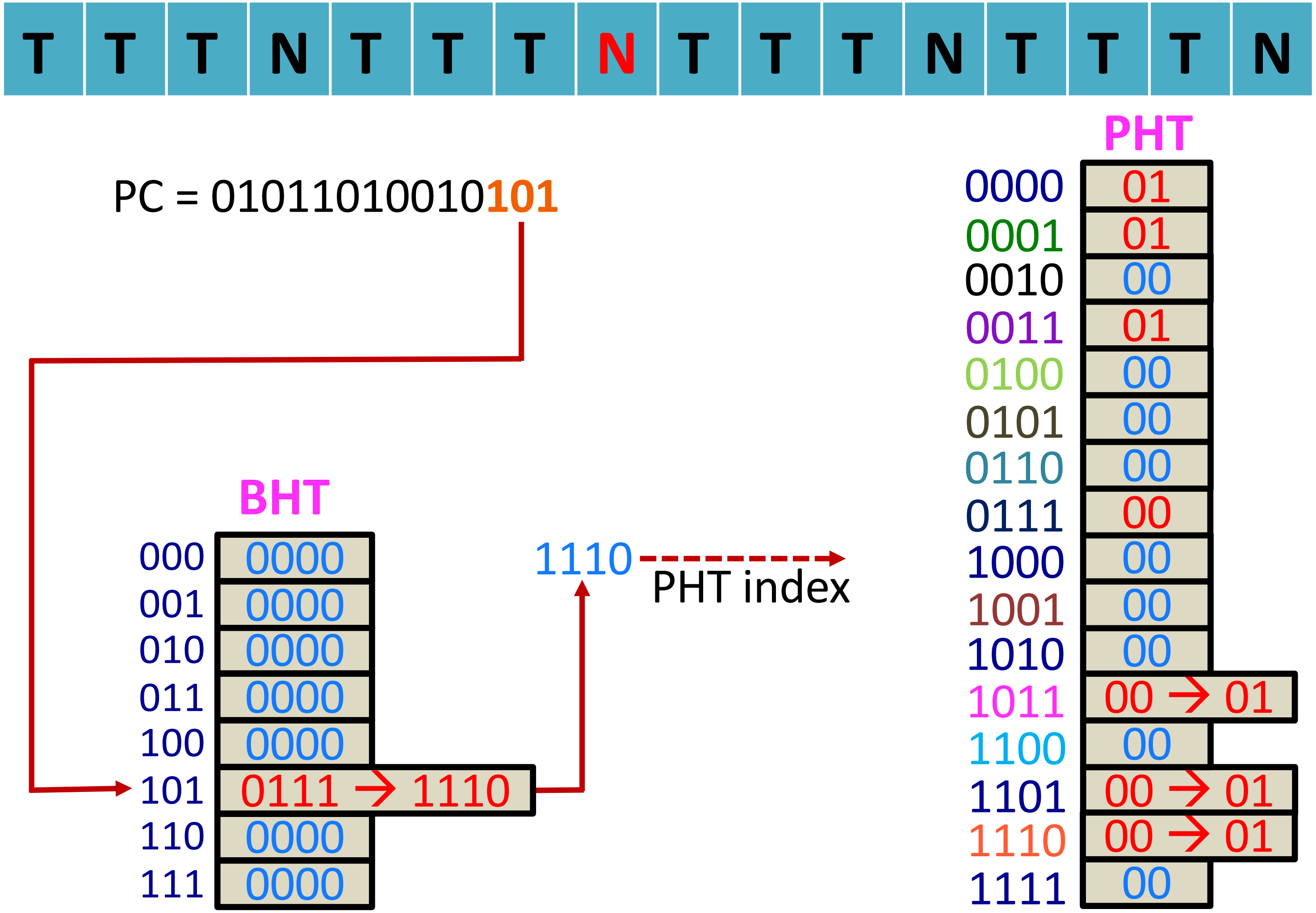
Iteration # 2

Prediction: NT

Outcome: NT

Score: 1/4

2-Level PAg predictor



T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N

PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0111 → 1110
110	0000
111	0000

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	00 → 01
1100	00
1101	00 → 01
1110	00 → 01
1111	00

Iteration # 2

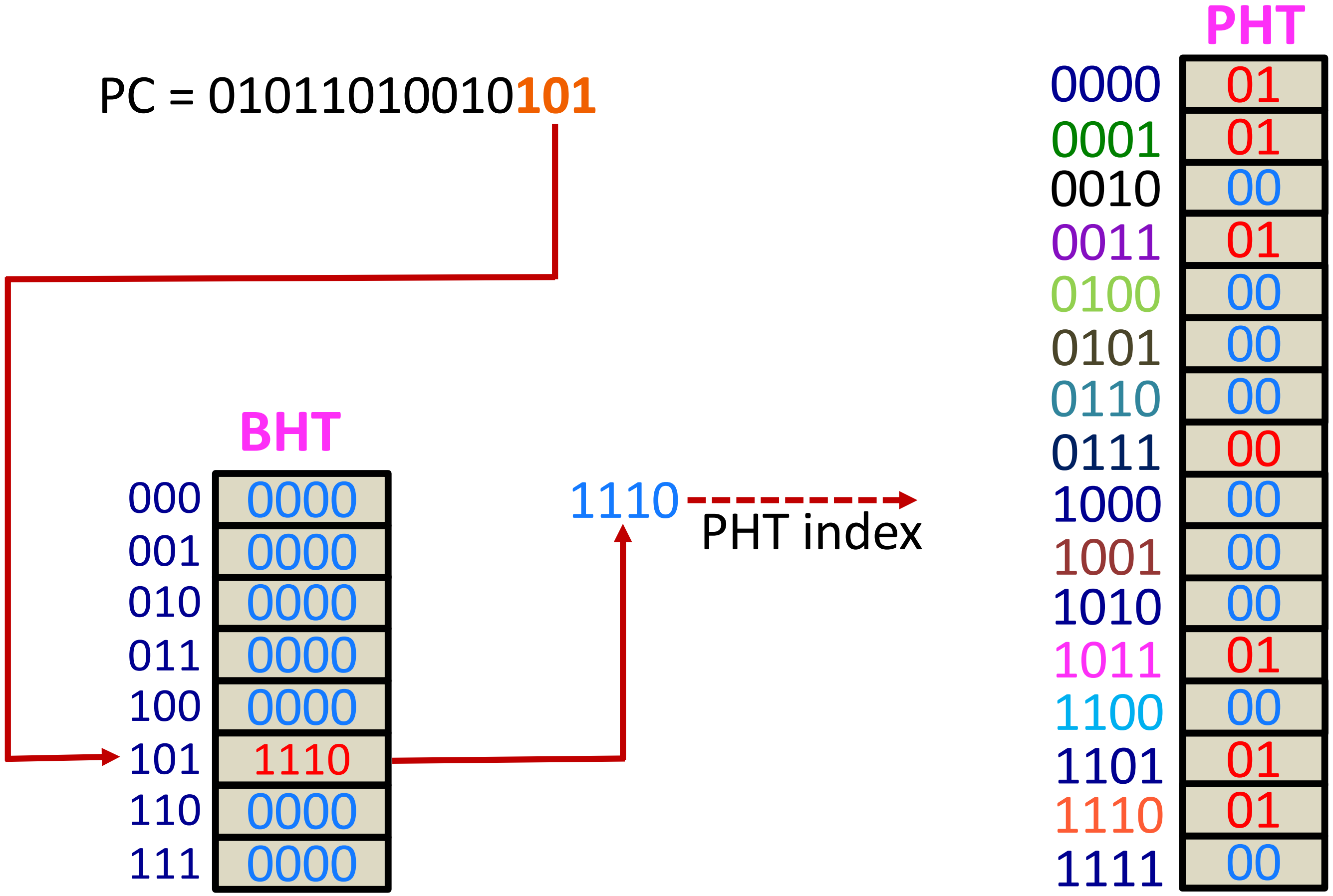
Prediction: NT

Outcome: NT

Score: 1/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1110
110	0000
111	0000

1110

PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	01
1100	00
1101	01
1110	01
1111	00

Iteration # 2

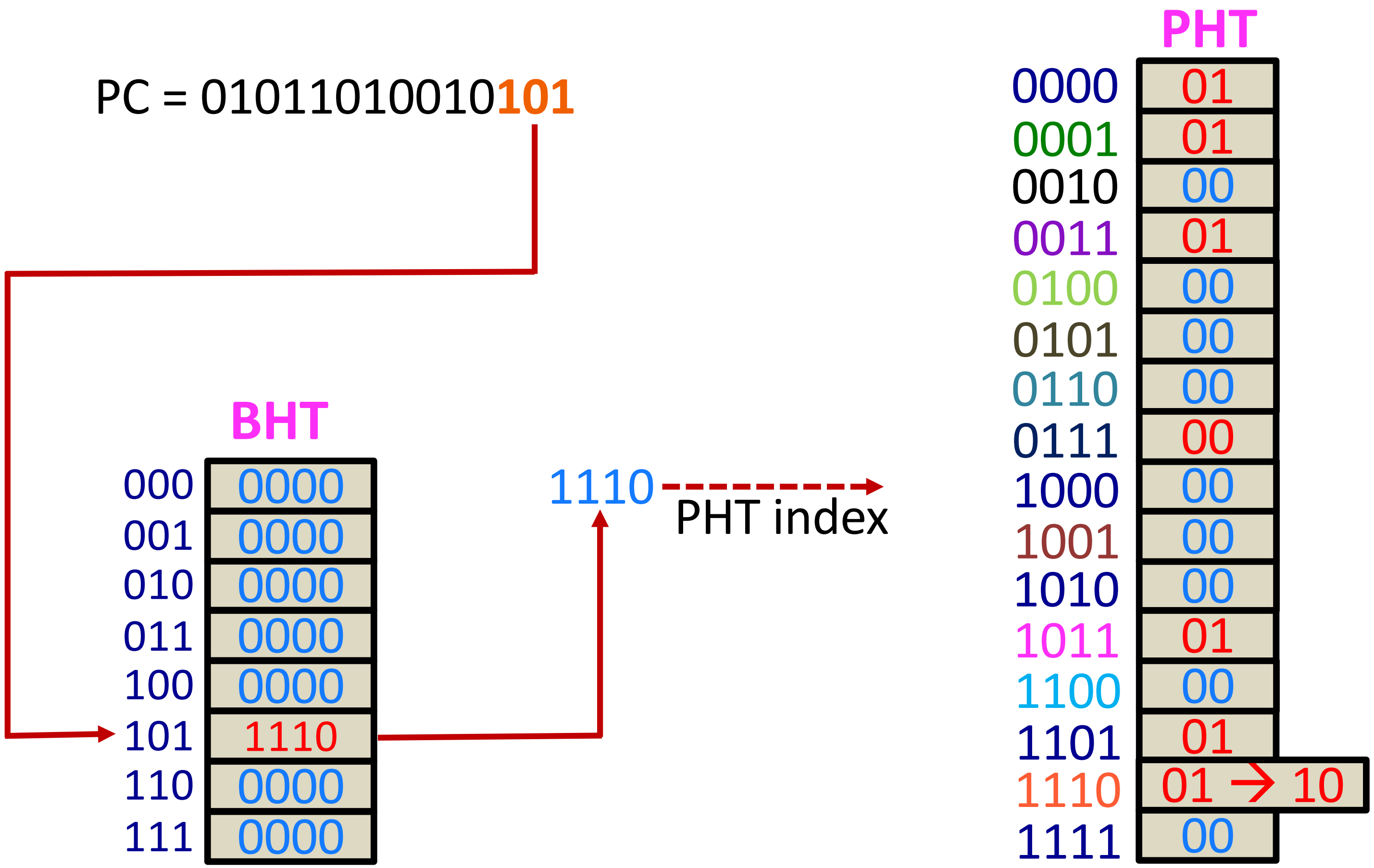
Prediction: NT

Outcome: NT

Score: 1/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010101

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1110
110	0000
111	0000

1110

PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	01
1100	00
1101	01
1110	01 → 10
1111	00

Iteration # 3

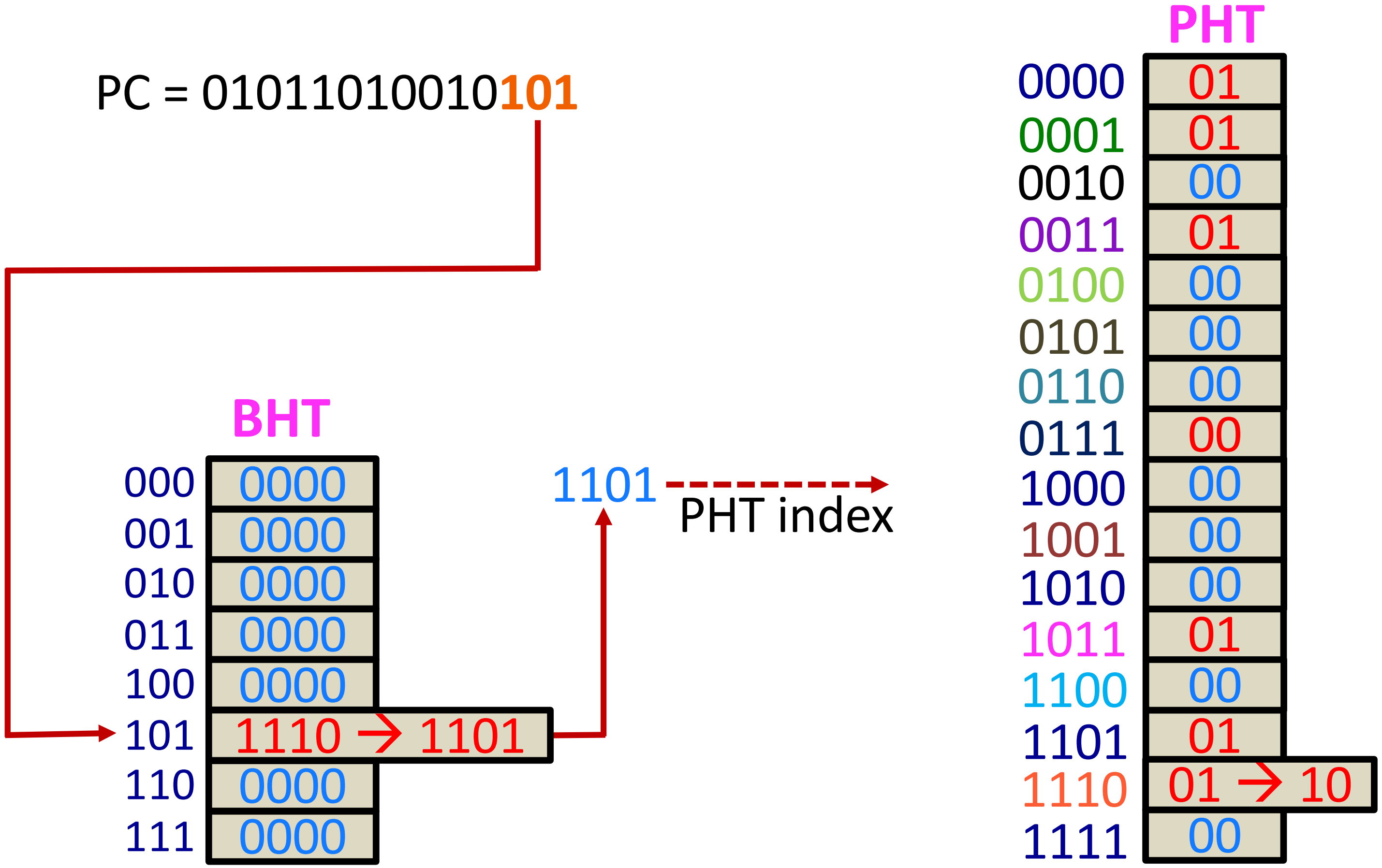
Prediction: NT

Outcome: T

Score: 0/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1110 → 1101
110	0000
111	0000

1101 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	01
1100	00
1101	01
1110	01 → 10
1111	00

Iteration # 3

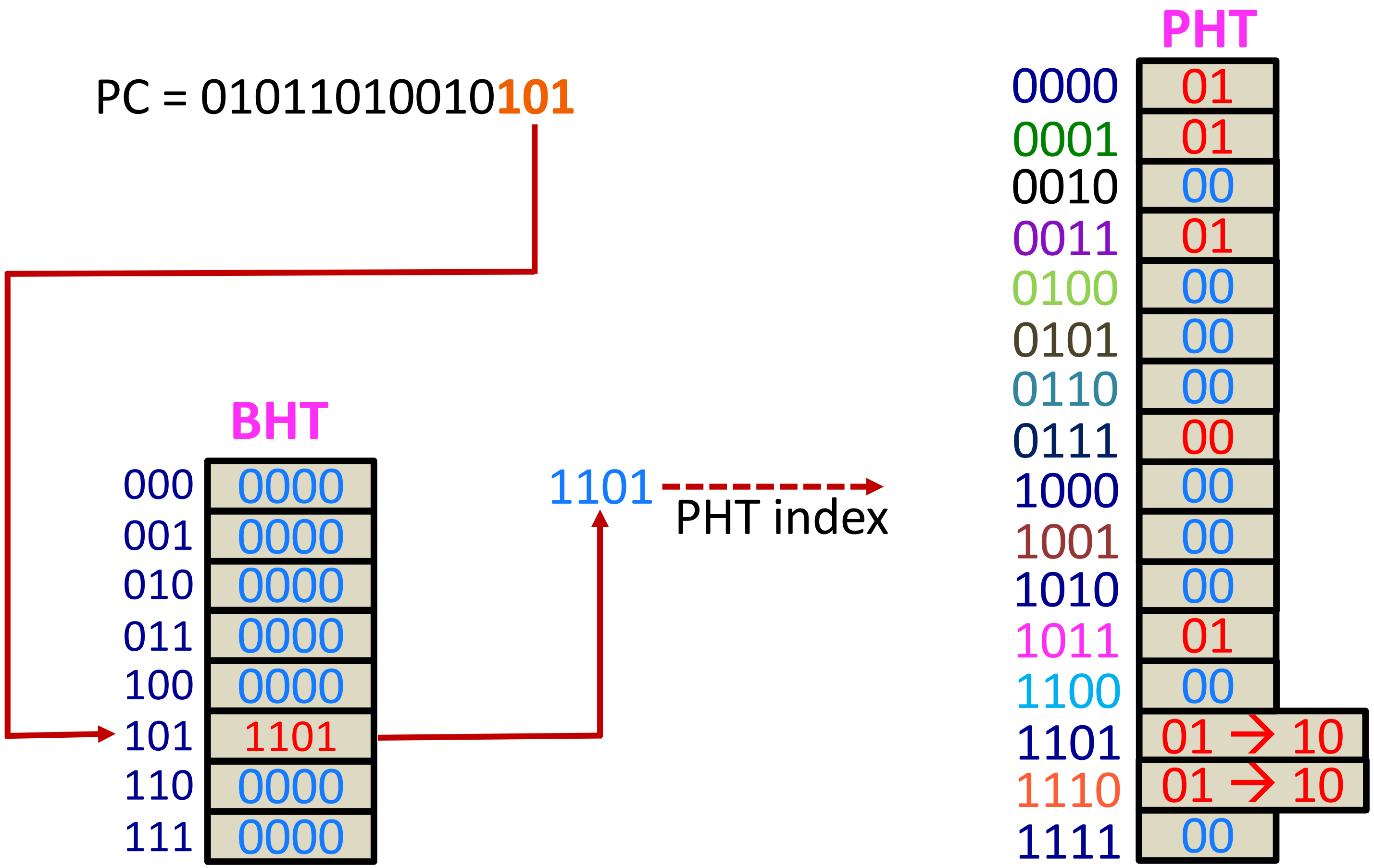
Prediction: NT

Outcome: T

Score: 0/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1101
110	0000
111	0000

1101 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	01
1100	00
1101	01 → 10
1110	01 → 10
1111	00

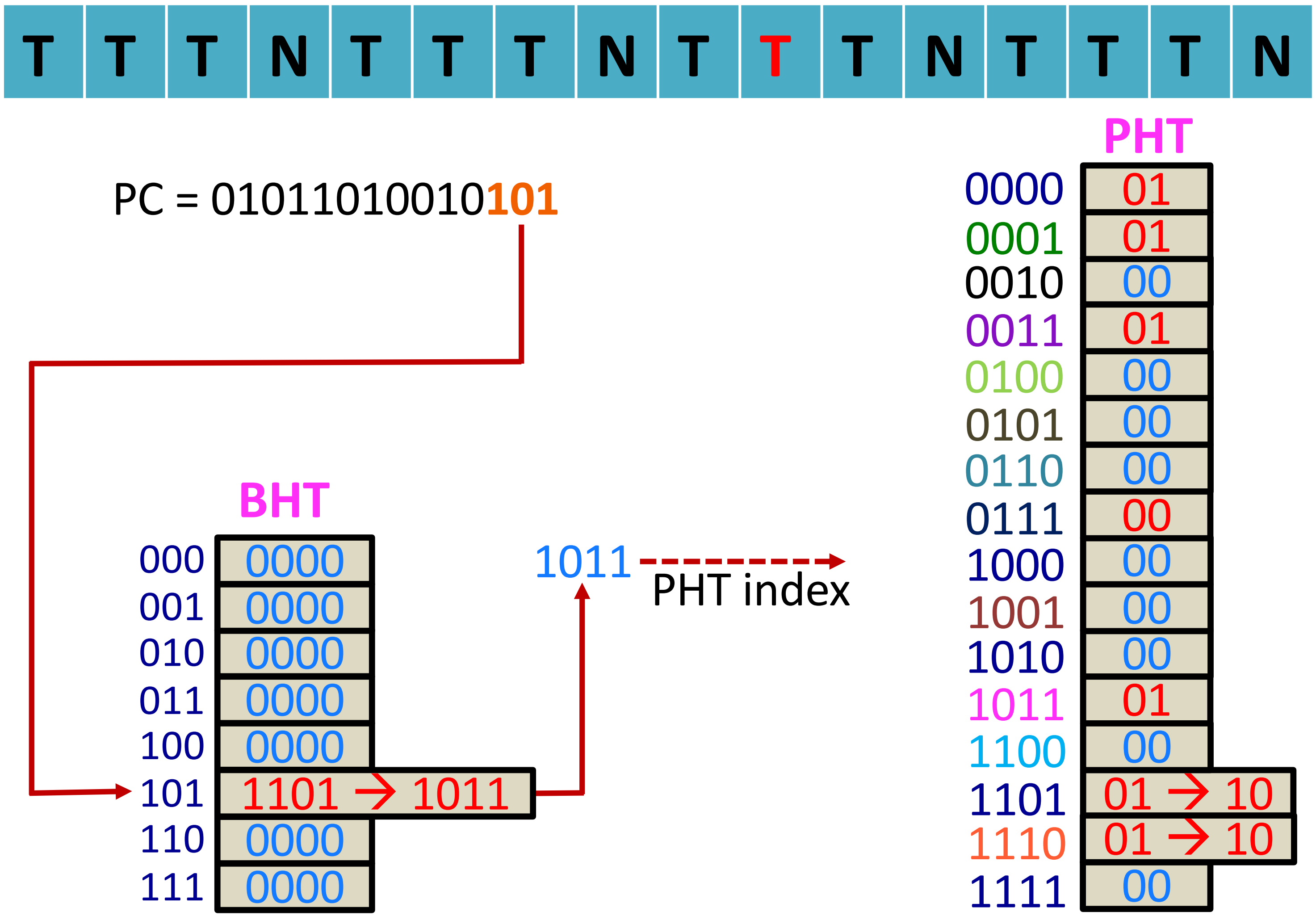
Iteration # 3

Prediction: NT

Outcome: T

Score: 0/4

2-Level PAg predictor



T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N

PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1101 → 1011
110	0000
111	0000

1011 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	01
1100	00
1101	01 → 10
1110	01 → 10
1111	00

Iteration # 3

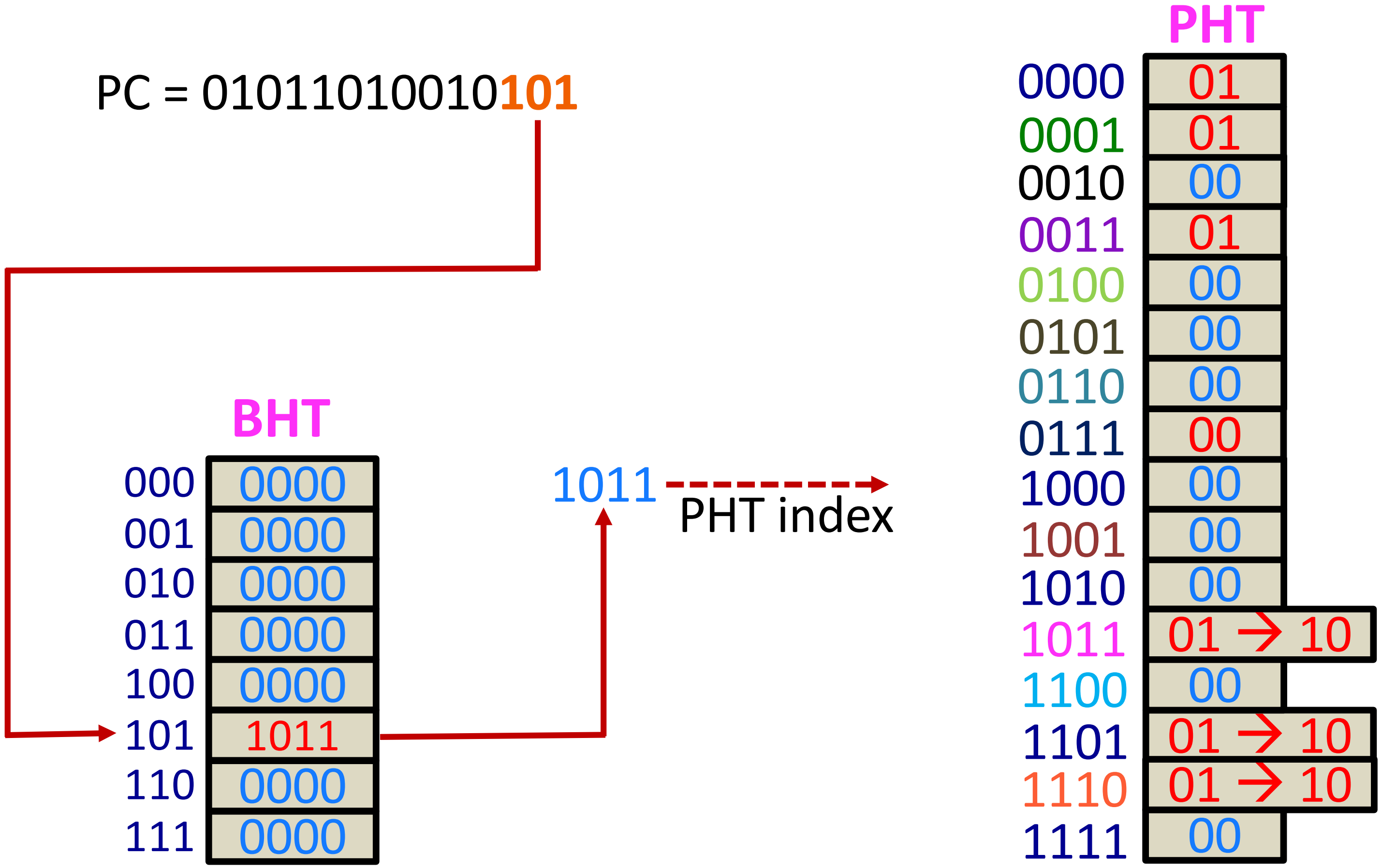
Prediction: NT

Outcome: T

Score: 0/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1011
110	0000
111	0000

1011 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	01 → 10
1100	00
1101	01 → 10
1110	01 → 10
1111	00

Iteration # 3

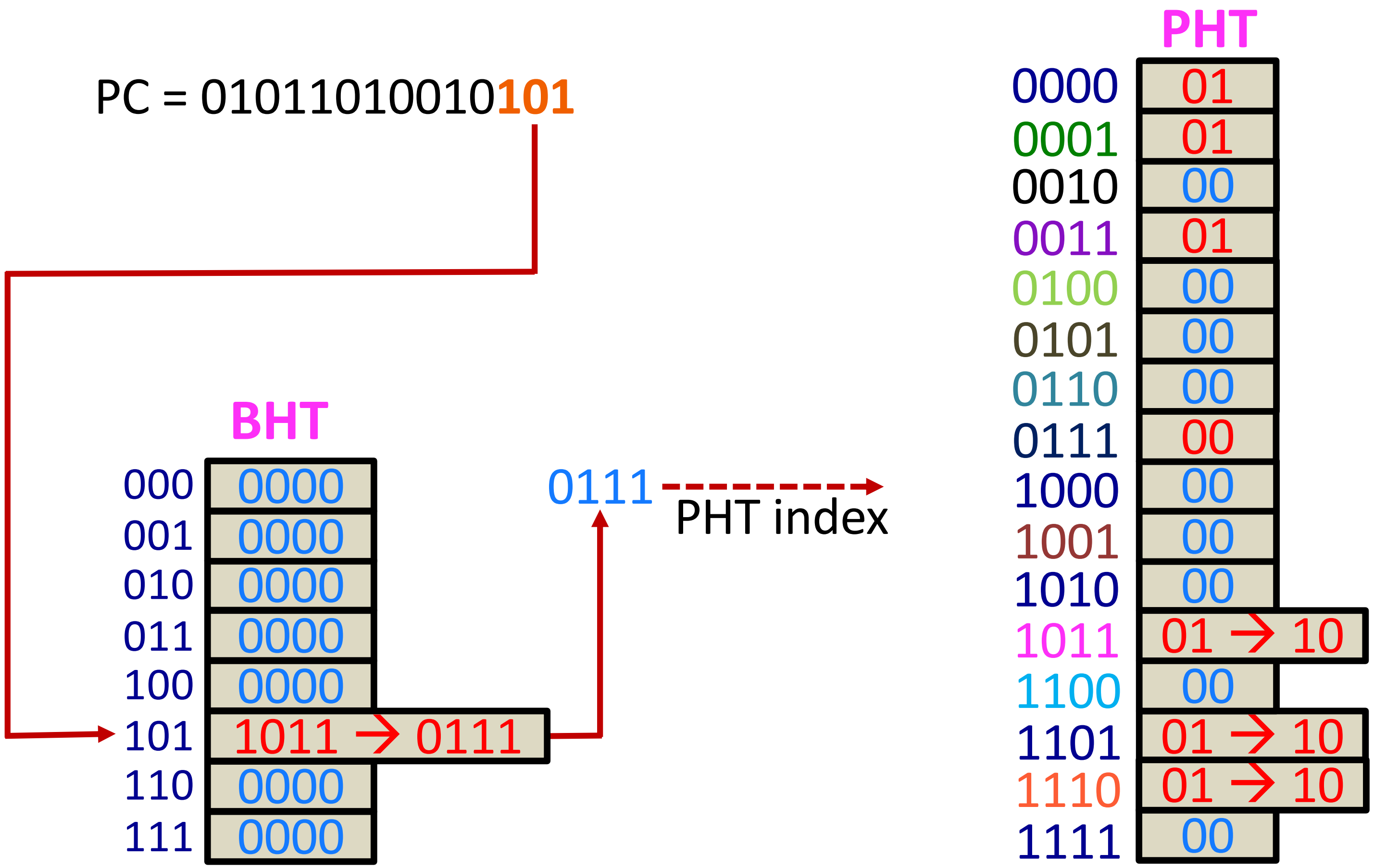
Prediction: NT

Outcome: T

Score: 0/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1011 → 0111
110	0000
111	0000

0111 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	01 → 10
1100	00
1101	01 → 10
1110	01 → 10
1111	00

Iteration # 3

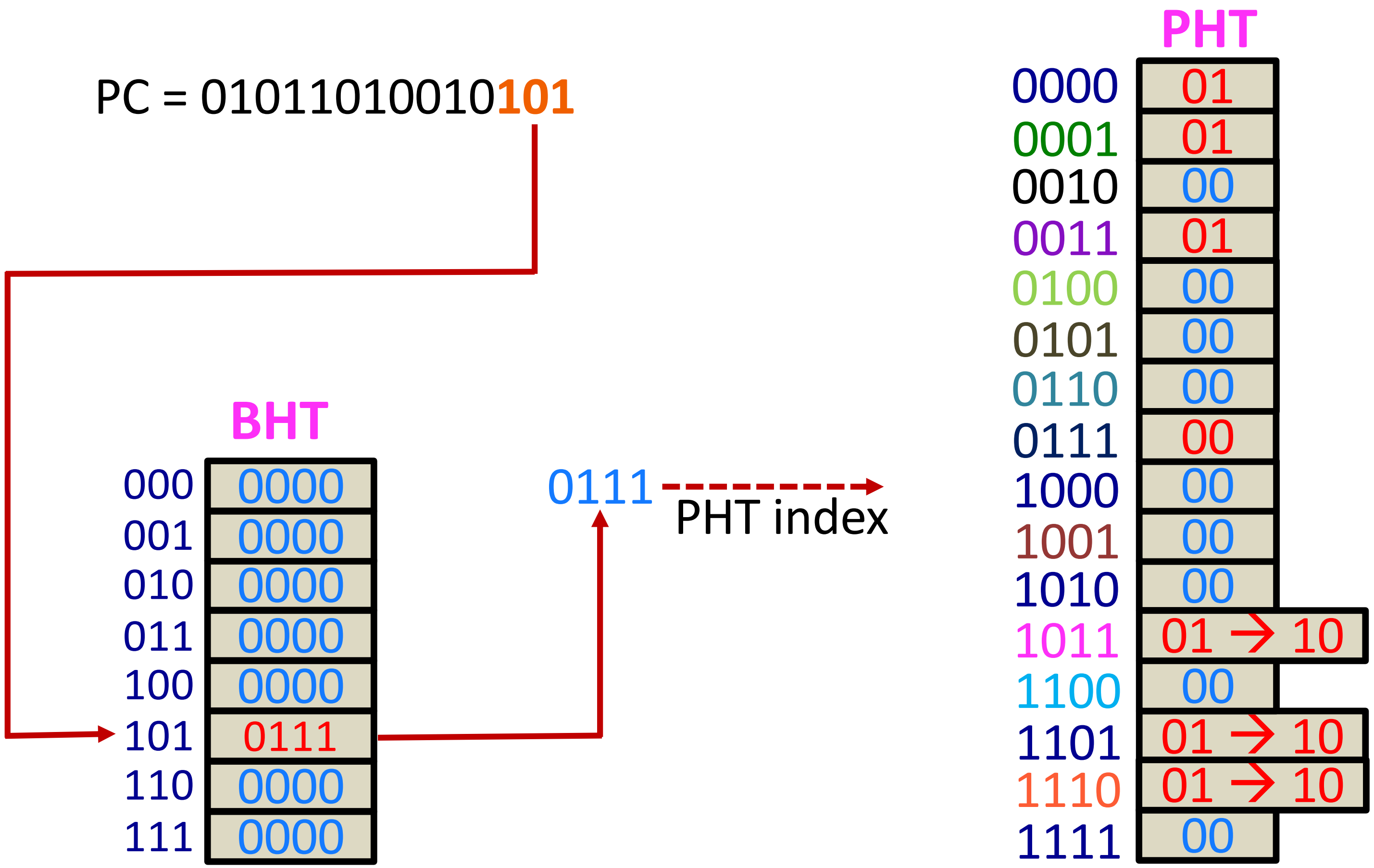
Prediction: NT

Outcome: T

Score: 0/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0111
110	0000
111	0000

0111 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	01 → 10
1100	00
1101	01 → 10
1110	01 → 10
1111	00

Iteration # 3

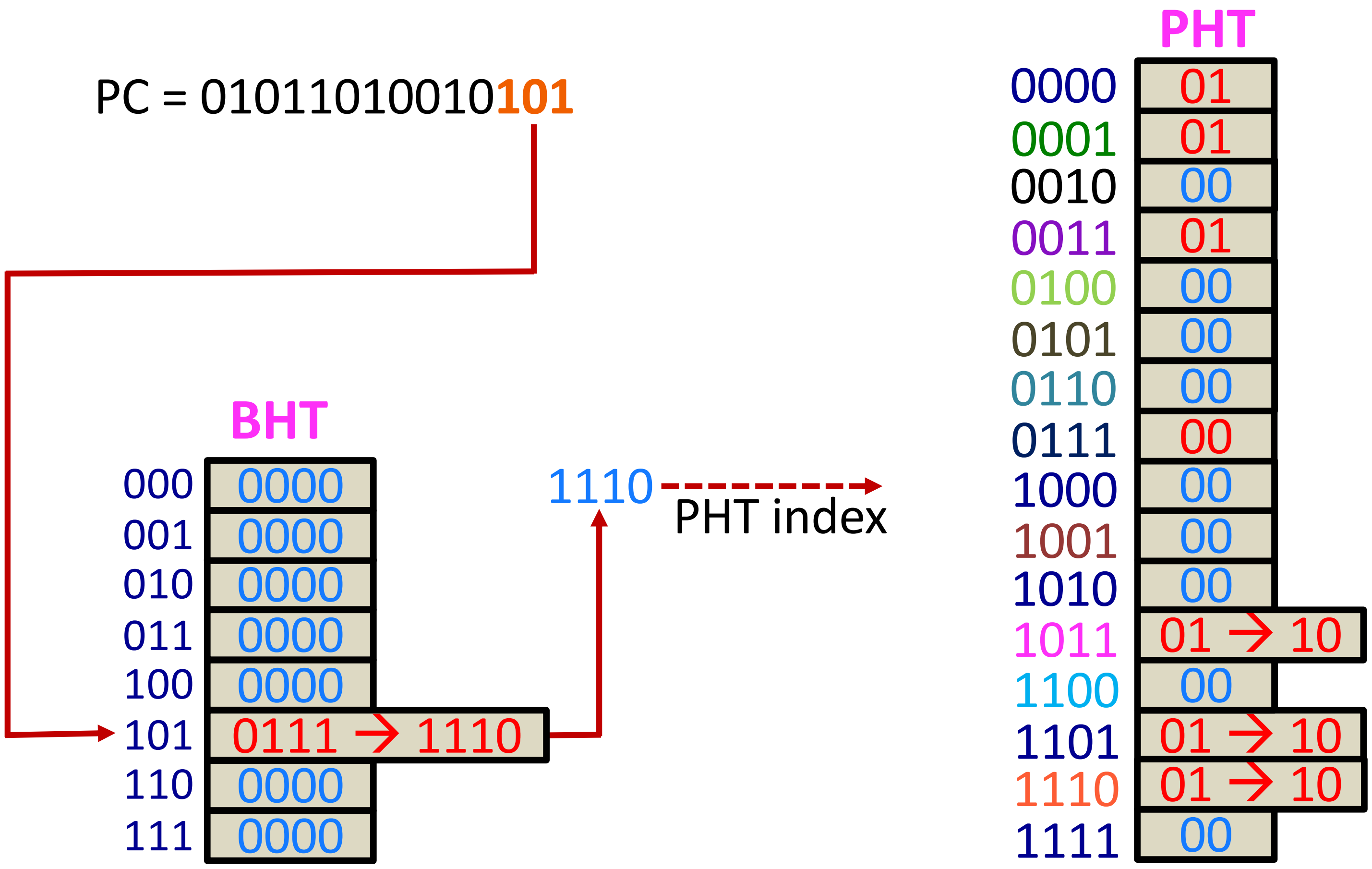
Prediction: NT

Outcome: NT

Score: 1/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0111 → 1110
110	0000
111	0000

1110 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	01 → 10
1100	00
1101	01 → 10
1110	01 → 10
1111	00

Iteration # 3

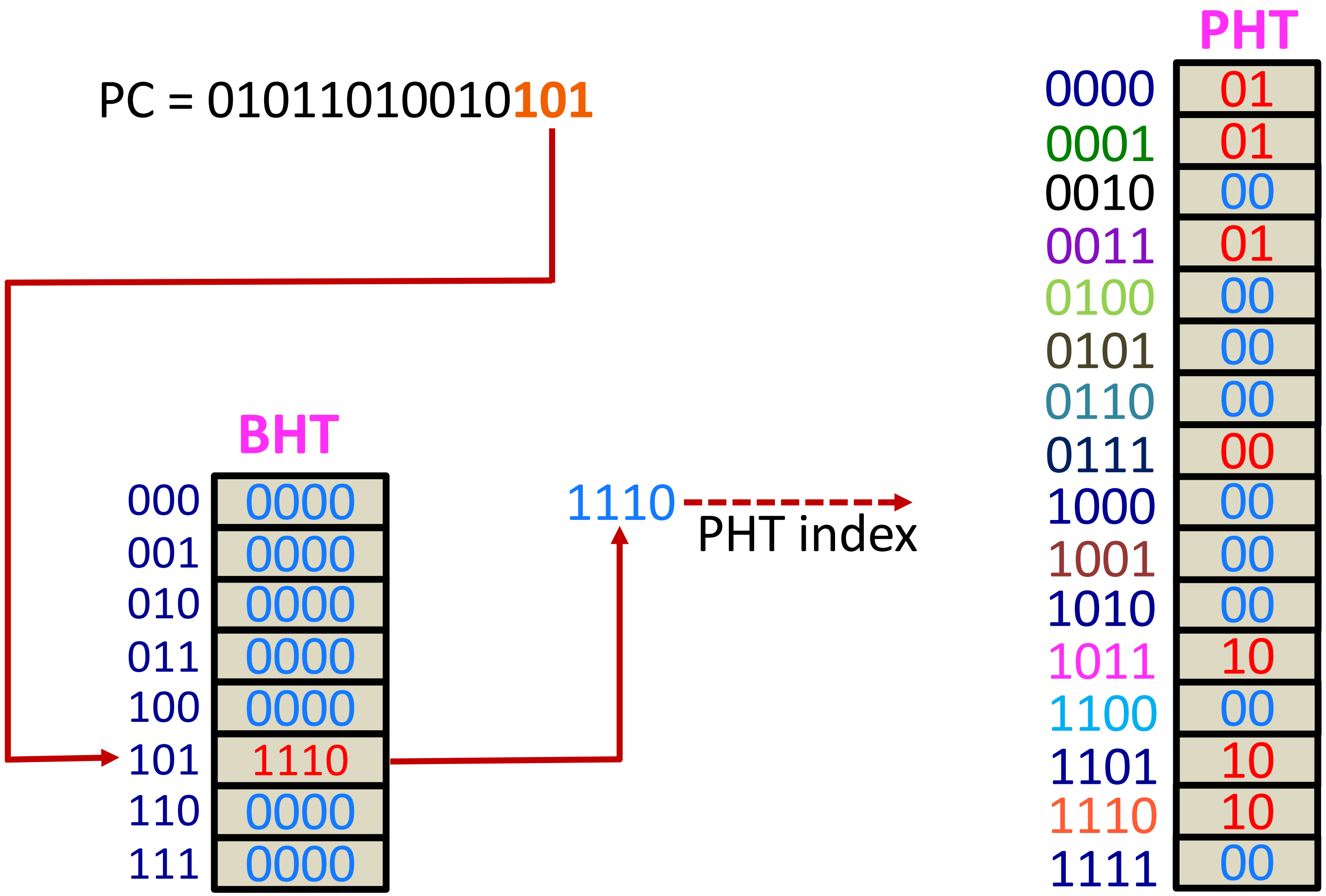
Prediction: NT

Outcome: NT

Score: 1/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1110
110	0000
111	0000

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	10
1100	00
1101	10
1110	10
1111	00

Iteration # 3

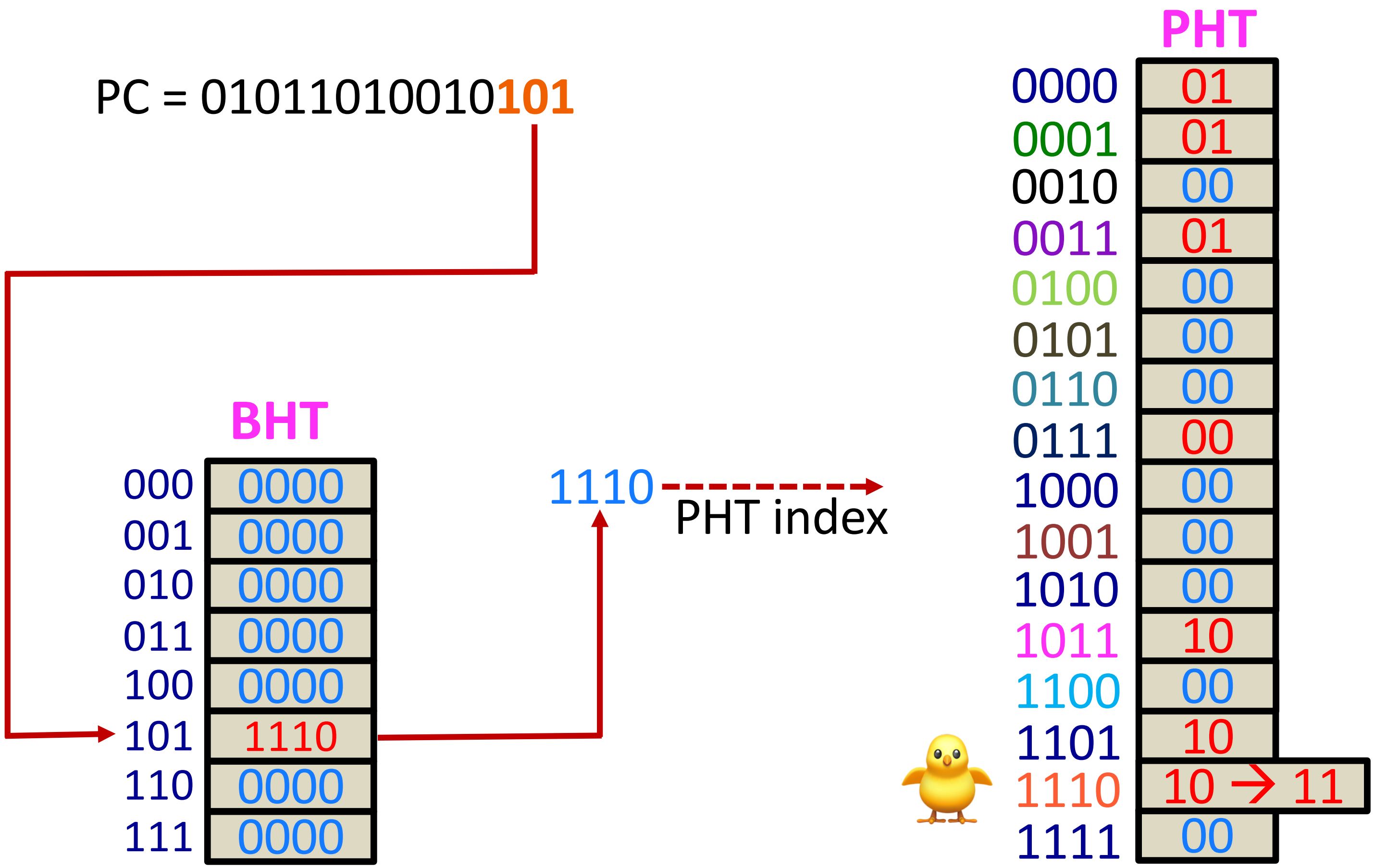
Prediction: NT

Outcome: NT

Score: 1/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1110
110	0000
111	0000

1110 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	10
1100	00
1101	10
🐥 1110	10 → 11
1111	00

Iteration # 4

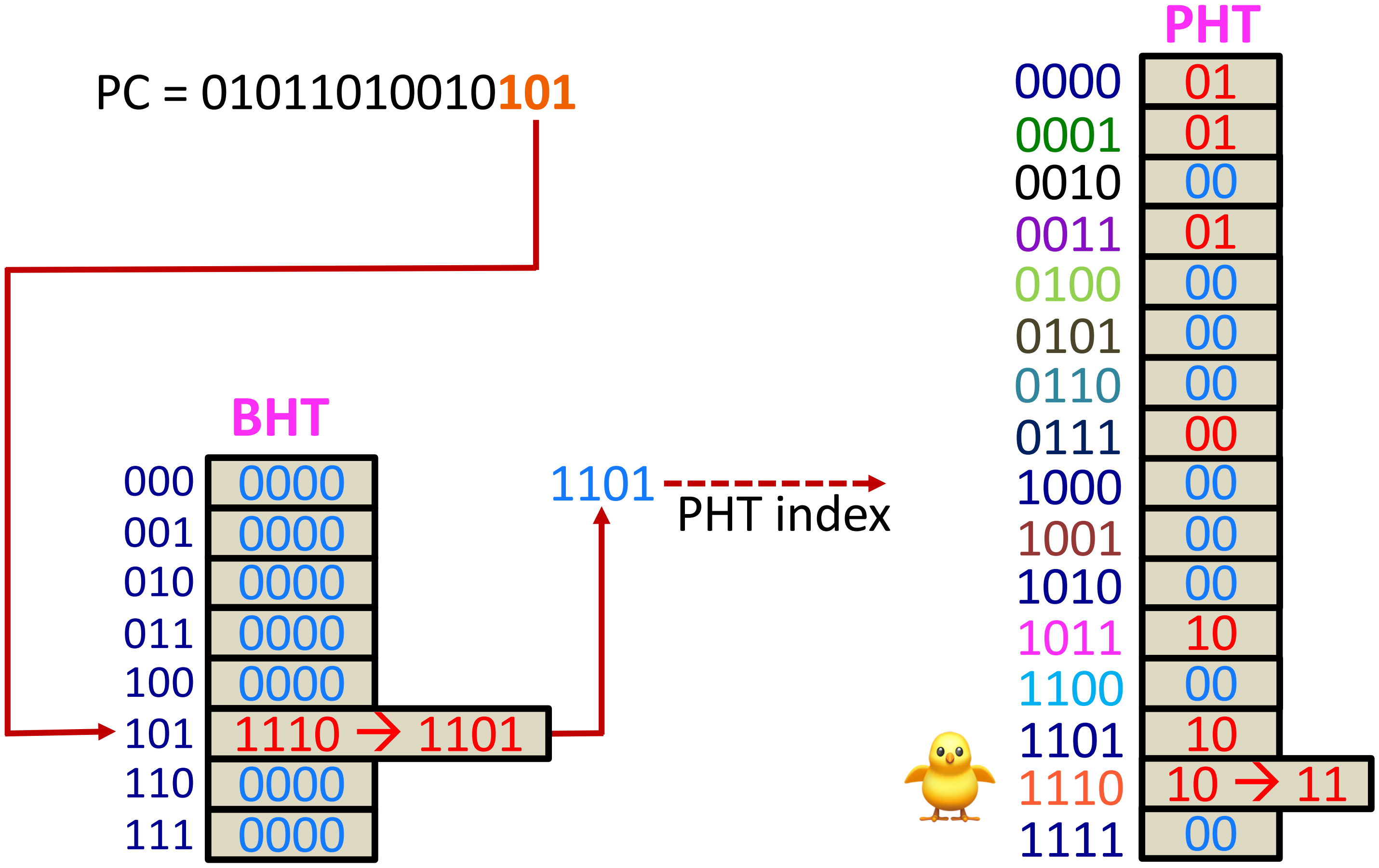
Prediction: T

Outcome: T

Score: 1/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010101

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1110 → 1101
110	0000
111	0000

1101 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	10
1100	00
1101	10
🐥 1110	10 → 11
1111	00

Iteration # 4

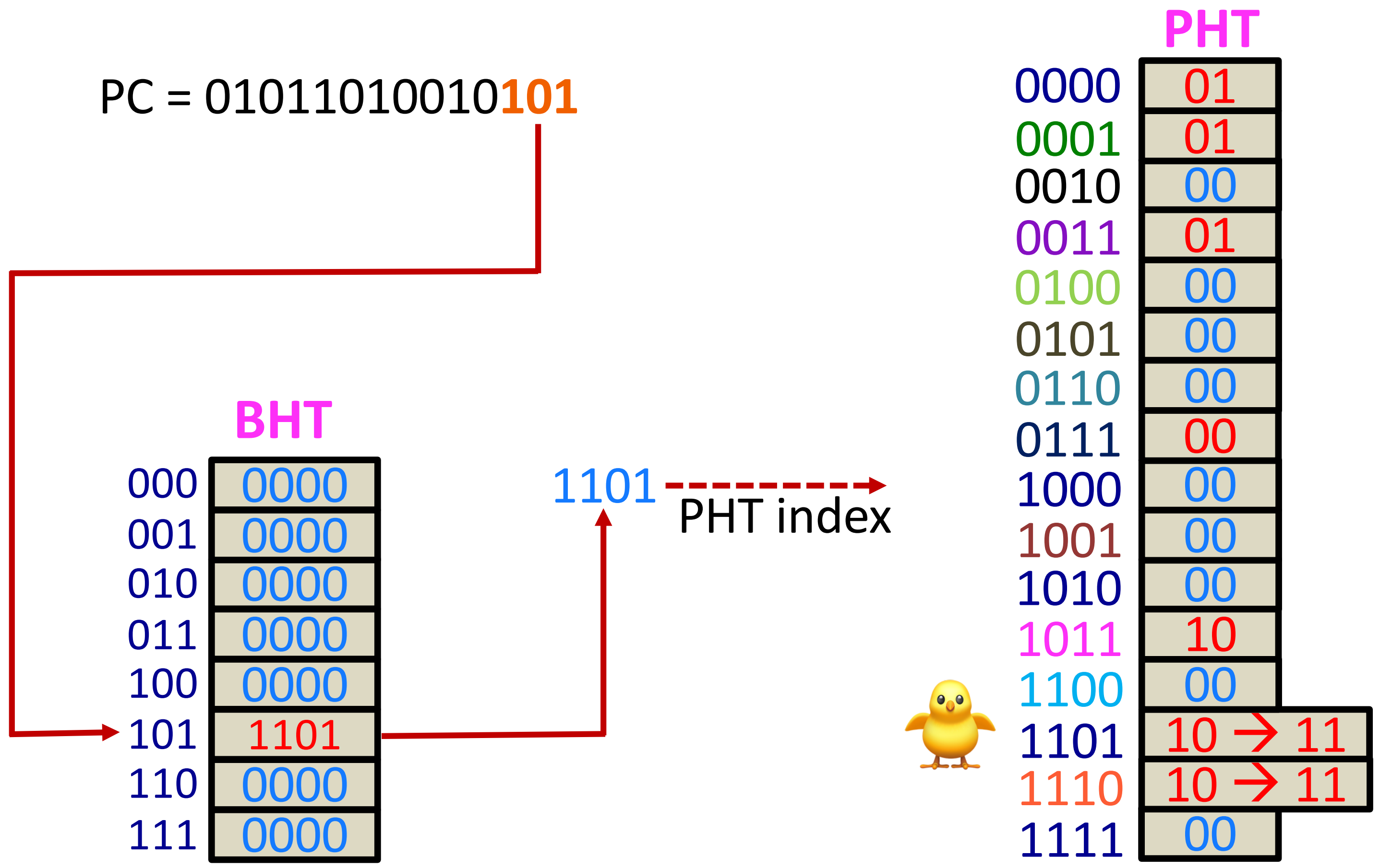
Prediction: T

Outcome: T

Score: 1/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1101
110	0000
111	0000

1101 → PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	10
1100	00
1101	10 → 11
1110	10 → 11
1111	00

Iteration # 4

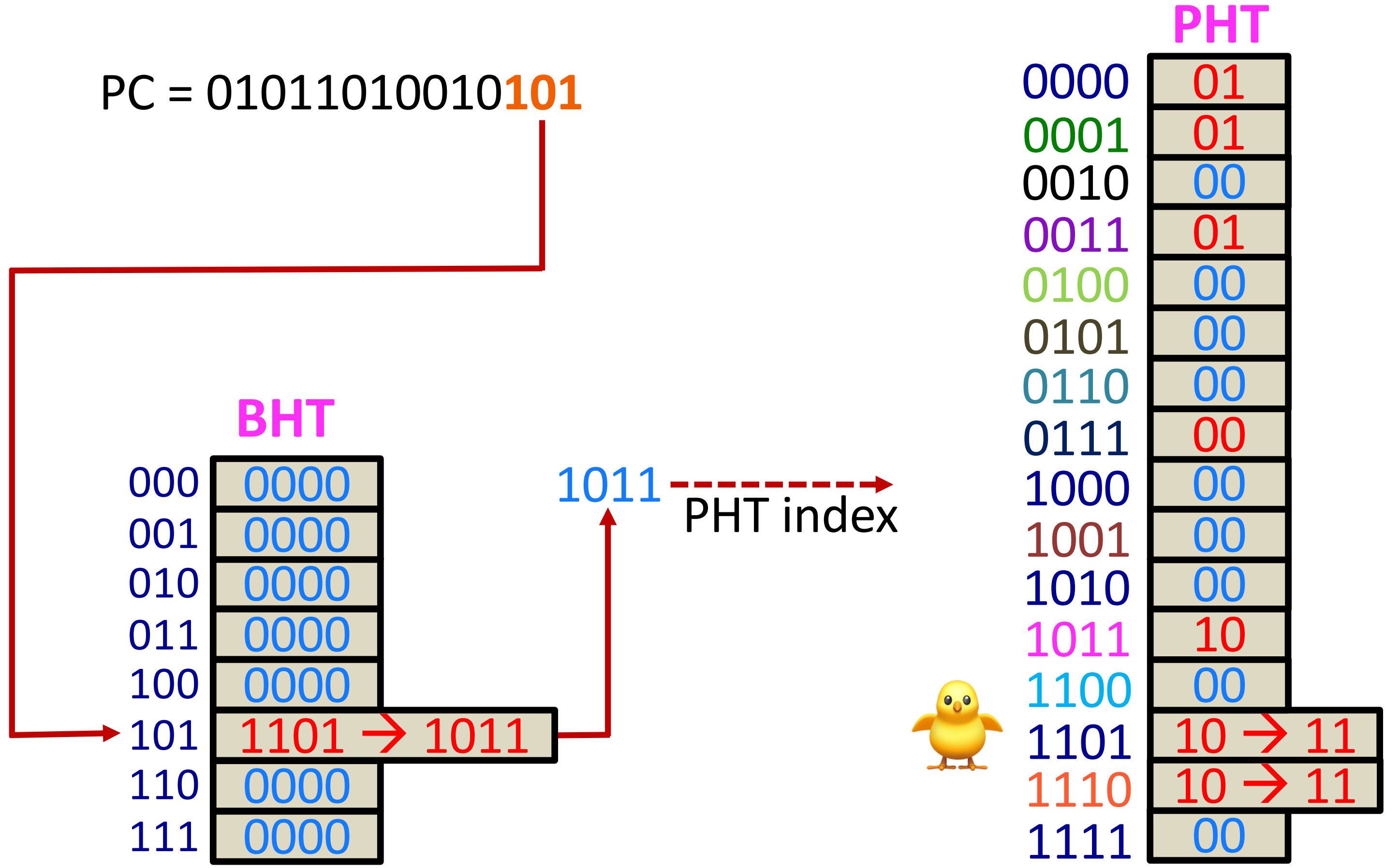
Prediction: T

Outcome: T

Score: 2/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010101

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1101 → 1011
110	0000
111	0000

1011 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	10
1100	00
1101	10 → 11
1110	10 → 11
1111	00

Iteration # 4

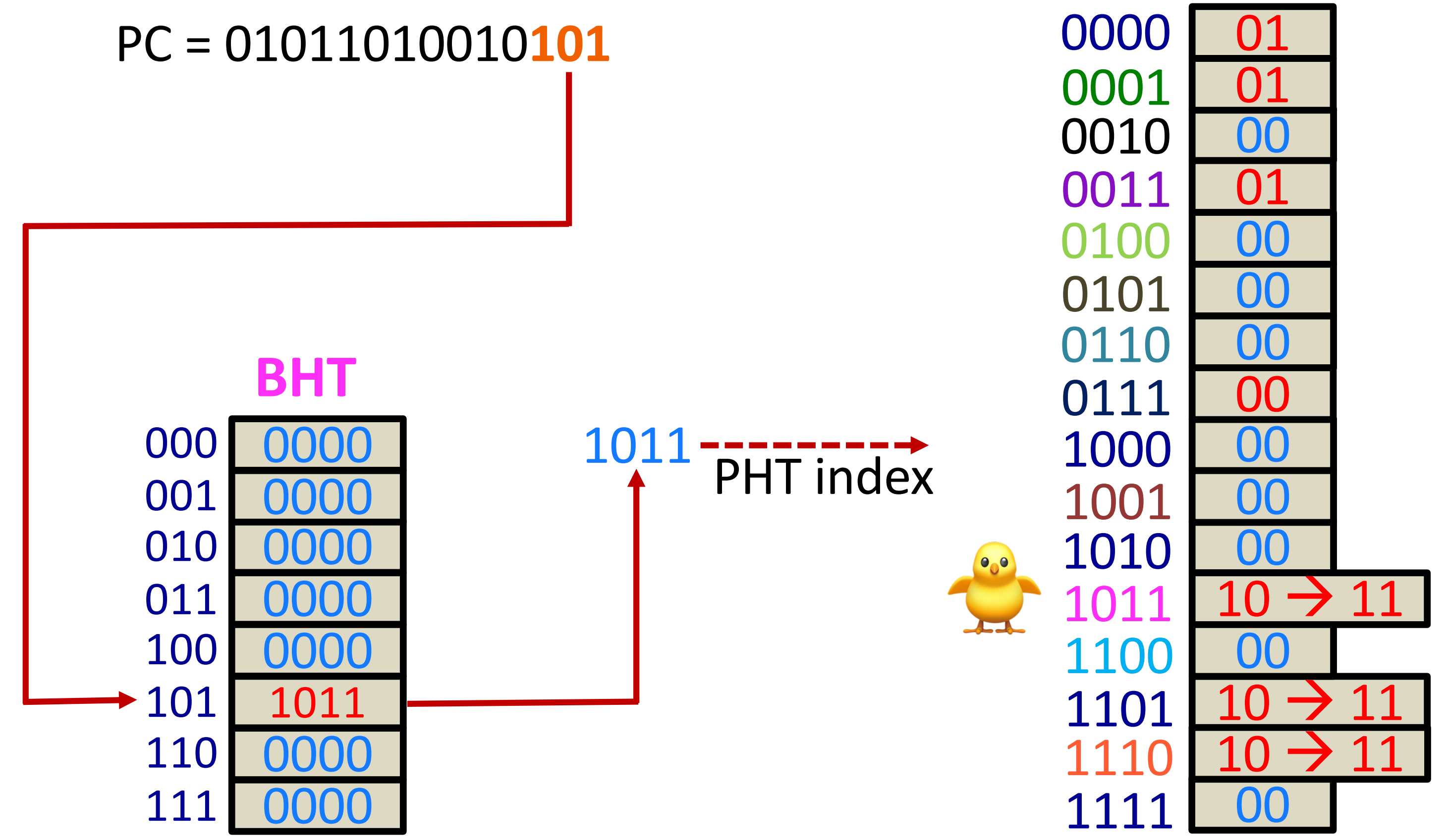
Prediction: T

Outcome: T

Score: 2/4

2-Level PAg predictor

T	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1011
110	0000
111	0000

1011 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	10 → 11
1100	00
1101	10 → 11
1110	10 → 11
1111	00

Iteration # 4

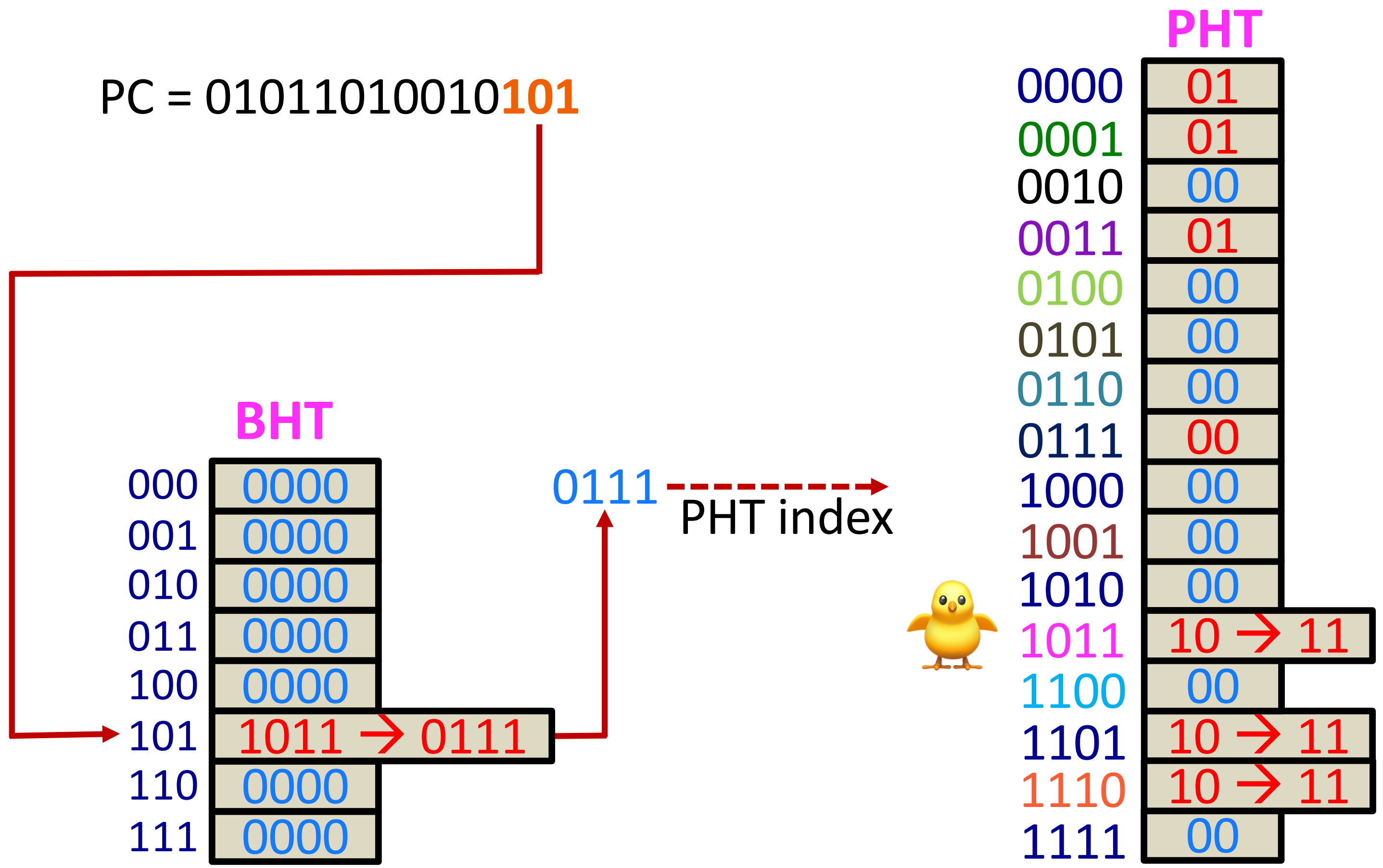
Prediction: T

Outcome: T

Score: 3/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	1011 → 0111
110	0000
111	0000

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	10 → 11
1100	00
1101	10 → 11
1110	10 → 11
1111	00

Iteration # 4

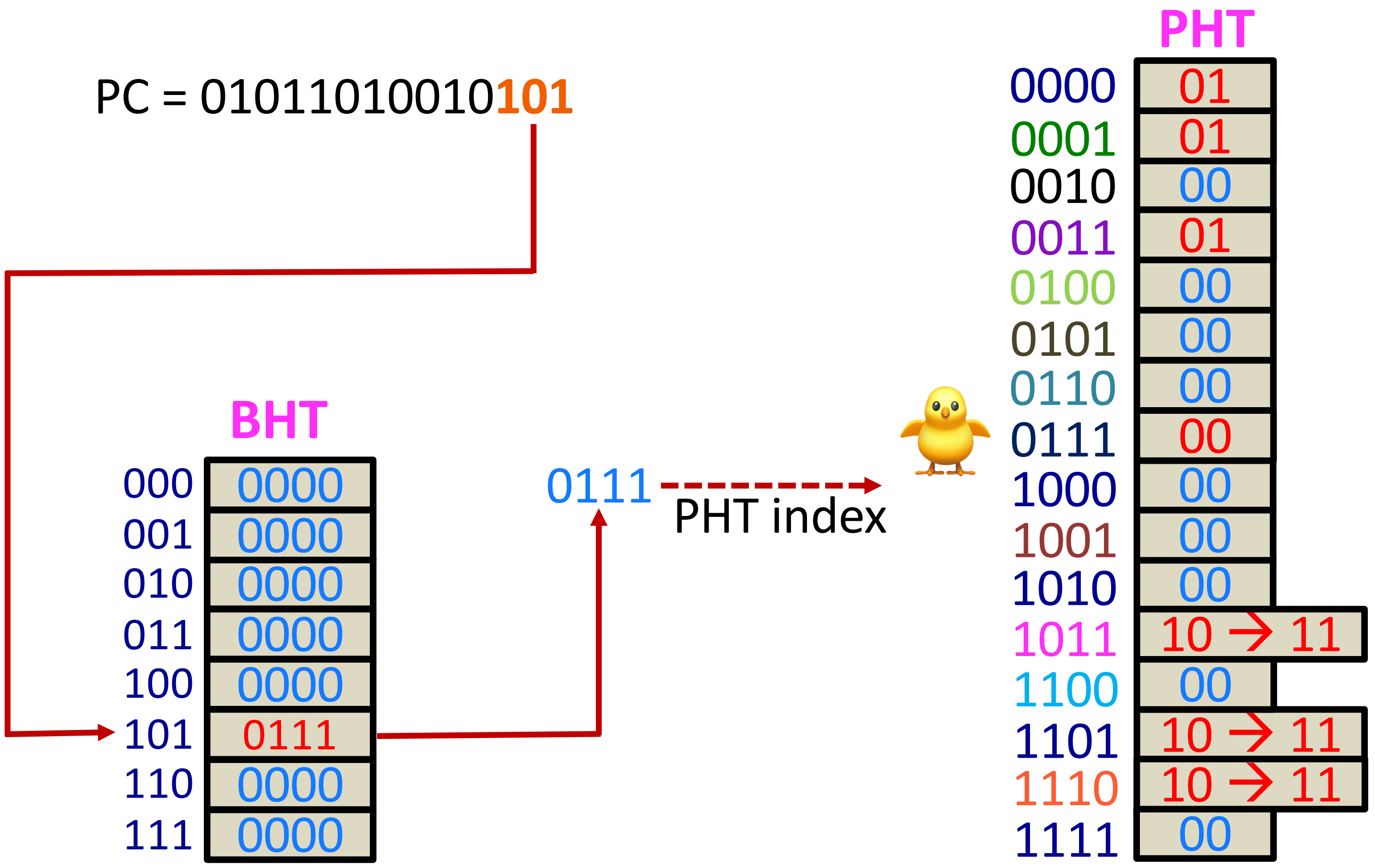
Prediction: T

Outcome: T

Score: 3/4

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010101

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0111
110	0000
111	0000

0111

PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	10 → 11
1100	00
1101	10 → 11
1110	10 → 11
1111	00

Iteration # 4

Prediction: NT

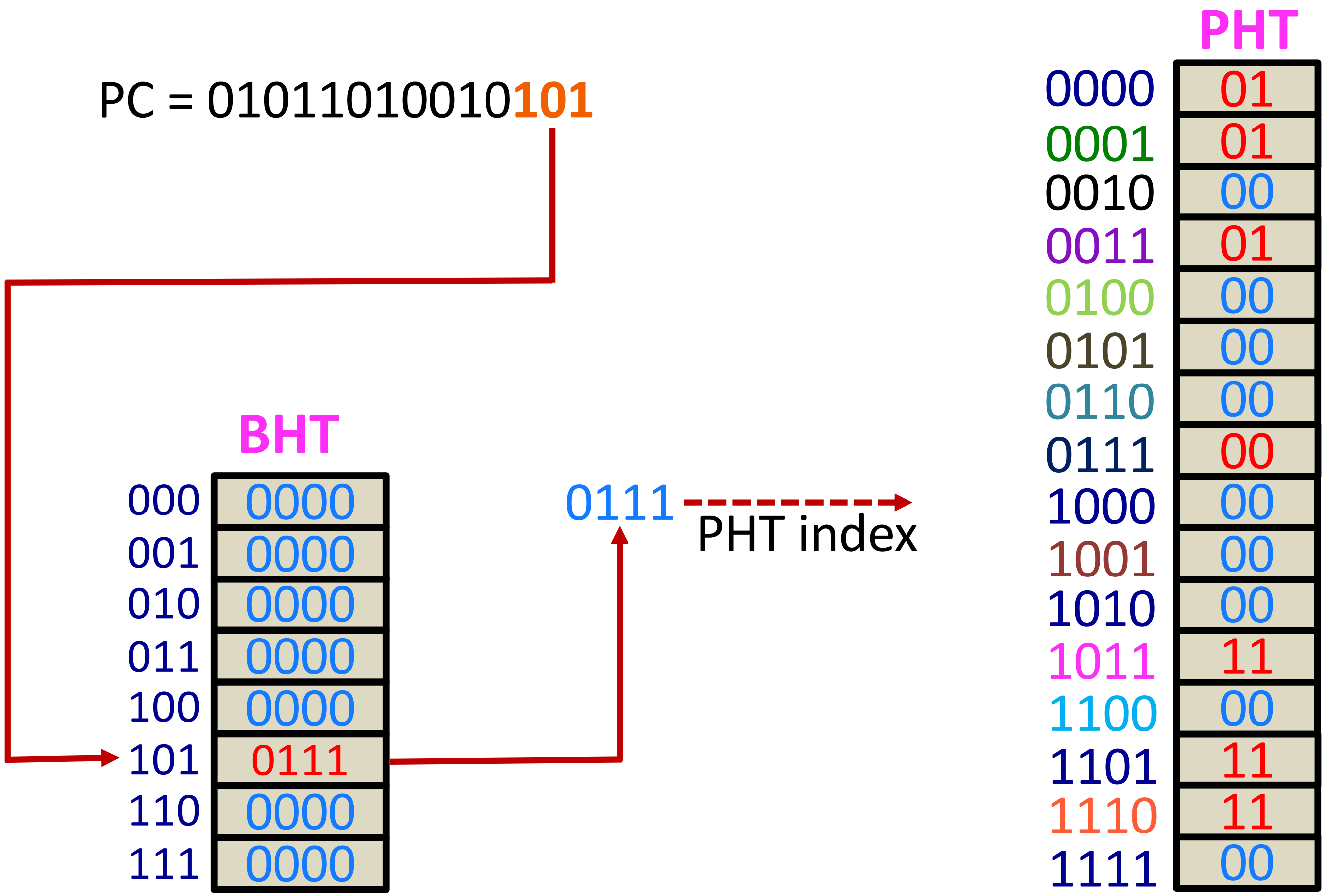
Outcome: NT

Score: 4/4

100% accuracy !

2-Level PAg predictor

T	T	T	N	T	T	T	N	T	T	T	N	T	T	T	N



PC = 01011010010**101**

BHT

000	0000
001	0000
010	0000
011	0000
100	0000
101	0111
110	0000
111	0000

0111 PHT index

PHT

0000	01
0001	01
0010	00
0011	01
0100	00
0101	00
0110	00
0111	00
1000	00
1001	00
1010	00
1011	11
1100	00
1101	11
1110	11
1111	00

Iteration # 4

Prediction: NT

Outcome: NT

Score: 4/4

100% accuracy !

Accuracy Comparison

T	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**	Loop Pattern

Compare the accuracy of three predictors (**4 iterations**)

	Smith$_1$	**Smith$_2$**	**PAg**
Accuracy	50%	62.5%	43.75%

Accuracy Comparison

T	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**N**	Loop Pattern

Compare the accuracy of three predictors (**100 iterations**)

	Smith$_1$	**Smith$_2$**	**PAg**
Accuracy	50%	74.5%	97.75%

Reasons for Mispredictions

Unseen (cold) branches → training time

- *Relearning due to phase behavior*
- *2^n history patterns for a BHR of size n*

Randomized/cryptographic algorithms

Lack of information

- Global history cannot see local correlation
- Not enough history bits

Negative interference/aliasing

- Contrast with neutral interference

Types of Aliasing

The three-C's model: Compulsory Capacity Conflict

Compulsory aliasing

- First use of address-history pair (approx. 1% mispredictions)

Capacity aliasing

- Size of working set is greater than the size of PHT
- 128-entry PHT, 129 branches in the program

Conflict aliasing

- Two different address-history pairs map to the same PHT entry
- 128-entry PHT, 2 branches in the program with addresses 131 and 259

Interference-Reducing Predictors

The next two predictors try to reduce the negative interference due to a shared PHT

gskewed Predictor

Bi-Mode Predictor

Branch Filtering

gskewed

Operation

- Divide the PHT into multiple banks
- Each bank is indexed with a different hash
- Combine the results with a majority function
- Total update: update all PHTs with the correct outcome
- Partial update: Do not update the mispredicted bank if overall prediction is correct

Intuition: *If two branch-history pairs conflict in one PHT, then they are unlikely to conflict in the other two PHTs*

Is gshare not sufficient?

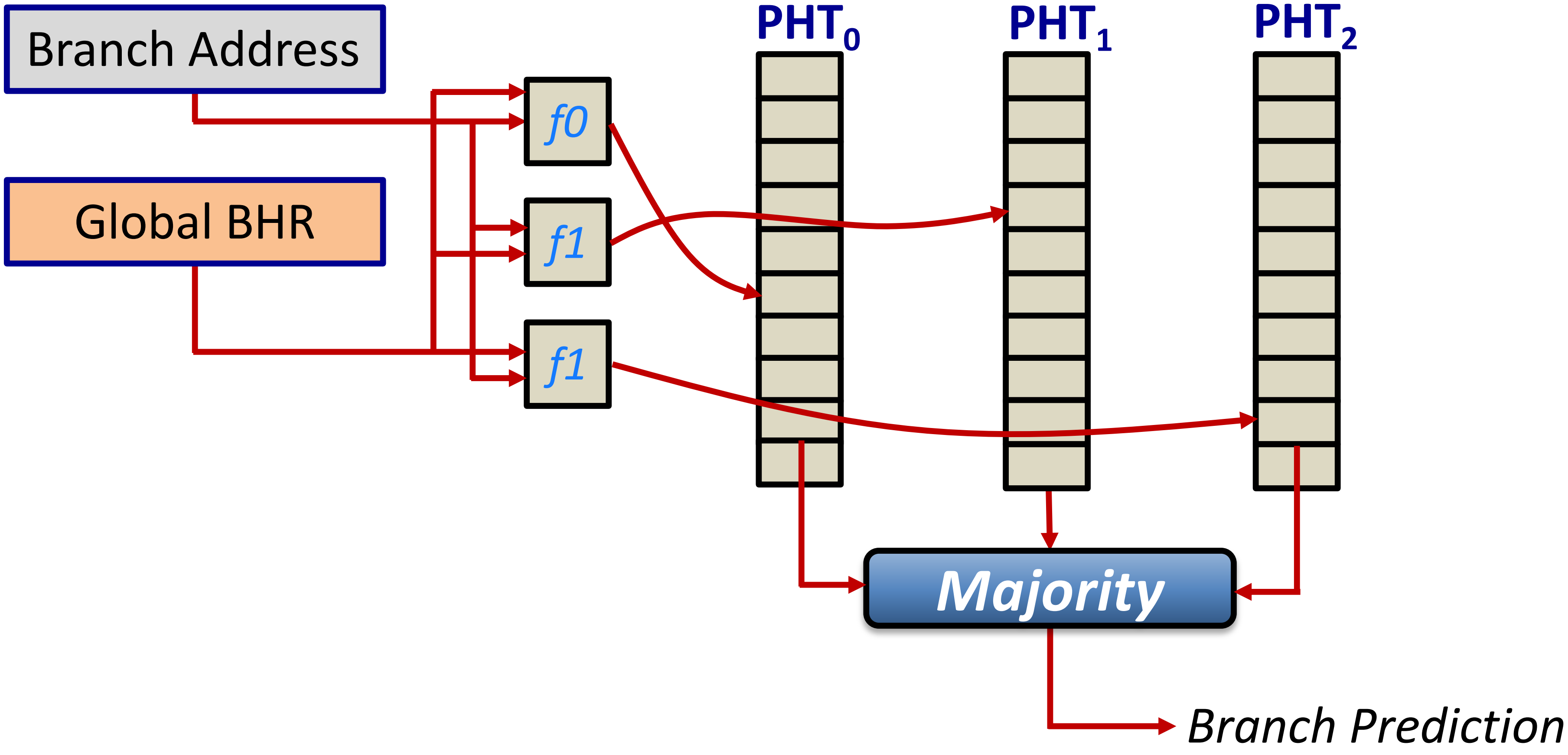
Consider the following branch-history pairs:

BHR	0	1	1	0
PC	1	1	0	0
Index	1	0	1	0

BHR	1	1	0	1
PC	0	1	1	1
Index	1	0	1	0

gshare

gskewed Predictor



gskewed Predictor

$f_0(x,y) = H(y) \veebar H^{-1}(x) \veebar x$

$f_1(x,y) = H(y) \veebar H^{-1}(x) \veebar y$

$f_2(x,y) = H^{-1}(y) \veebar H(x) \veebar x$

- (x,y) are n-bits long
- H^{-1} is inverse of H

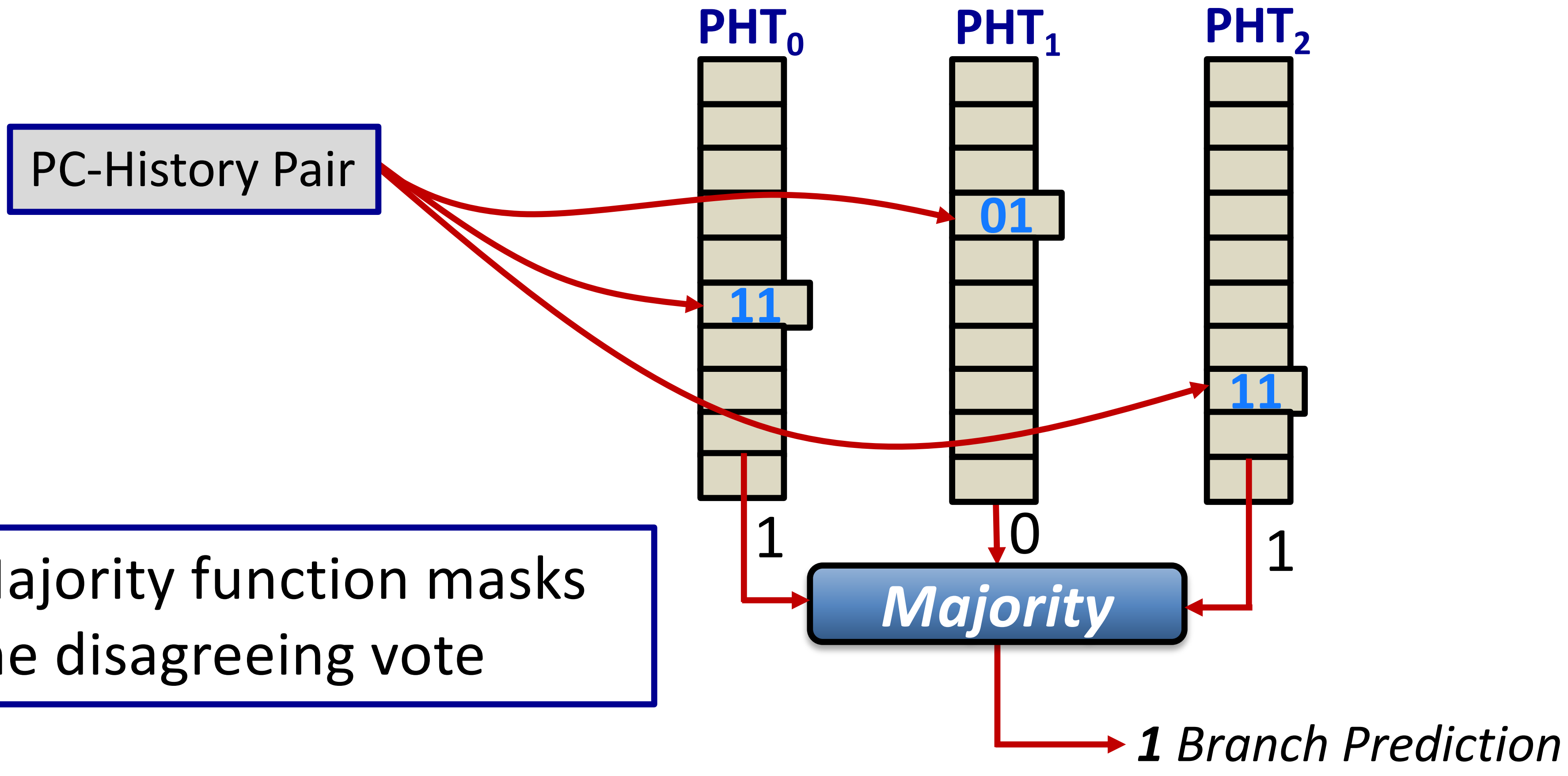
$f_0(x1) = f_0(x2)$ then $f_1(x1)$ **!=** $f_1(x2)$ and $f_2(x1)$ **!=** $f_2(x2)$

$H(b_n, b_{n-1},\ldots,b_3,b_2,b_1) = (b_n \text{ XOR } b_1, b_n, b_{n-1},\ldots,b_3,b_2)$

gskewed Predictor

Used in Alpha EV8

Never realized



Majority function masks the disagreeing vote

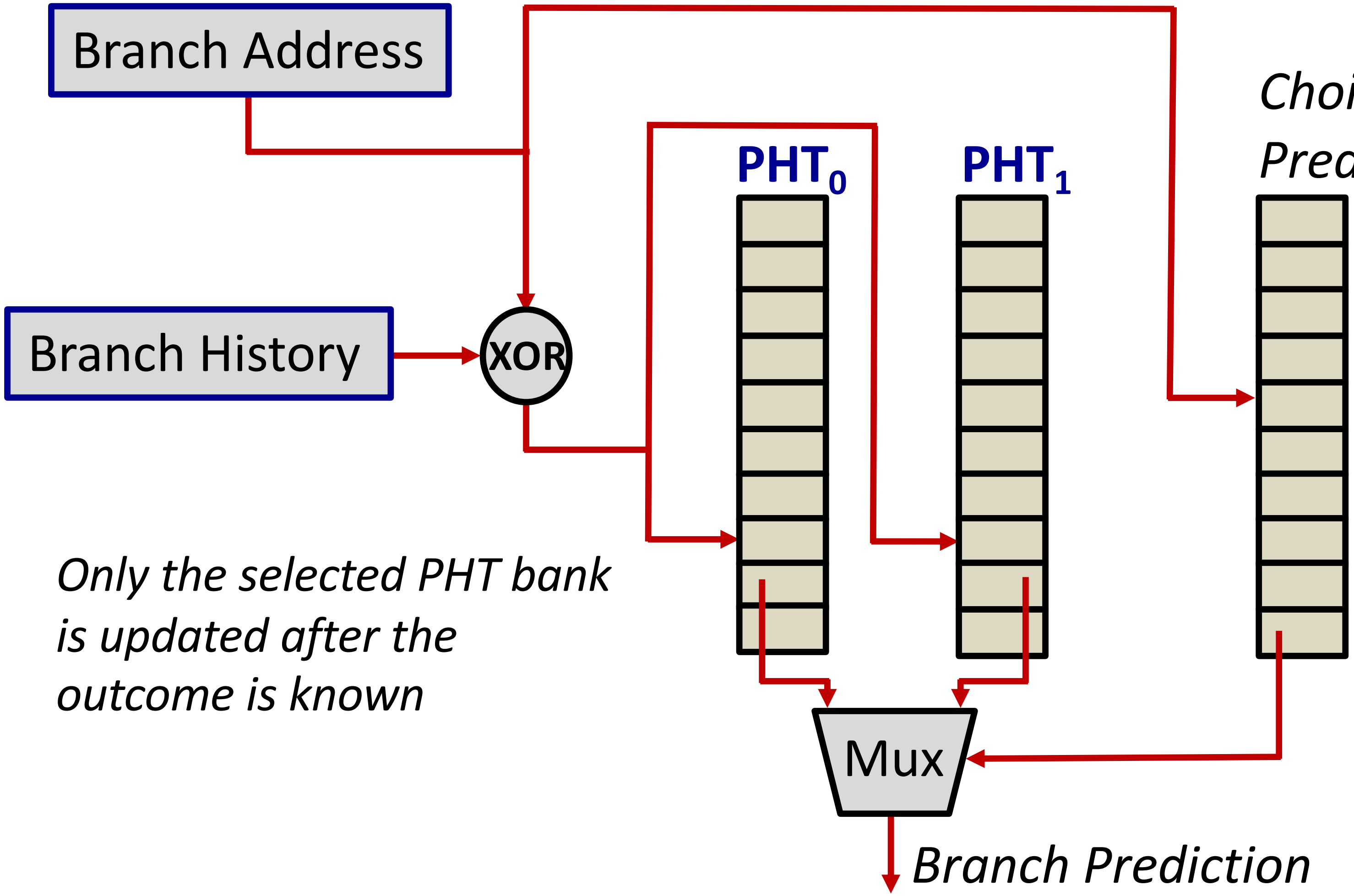
Bi-Mode Predictor

Operation

- Split branches into two groups (ST and SNT)
- Use two PHTs (direction predictors) and index with the same address-history hash
- ST branches map to one PHT, and SNT branches to the other
- A meta-predictor (*choice predictor*) selects the PHT bank
- Index the choice predictor with the branch address

Intuition: *Branches have a bias (ST or SNT). Separating them into two PHT mitigates –ve interference. If two branches map to the same entry in the choice predictor, they are unlikely to harm each other*

Bi-Mode Predictor



NNNNNNNTNNTN

- Which PHT?
- Entries of PHT?

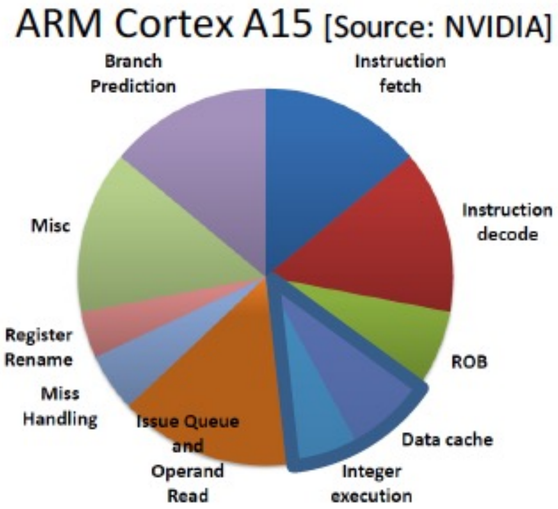
Only the selected PHT bank is updated after the outcome is known

Choice predictor is updated with the branch outcome

Bi-Mode Predictor

15% of Cortex A15 power

Sizing more complicated as one needs to tune PHT sizes and that of the choice predictor



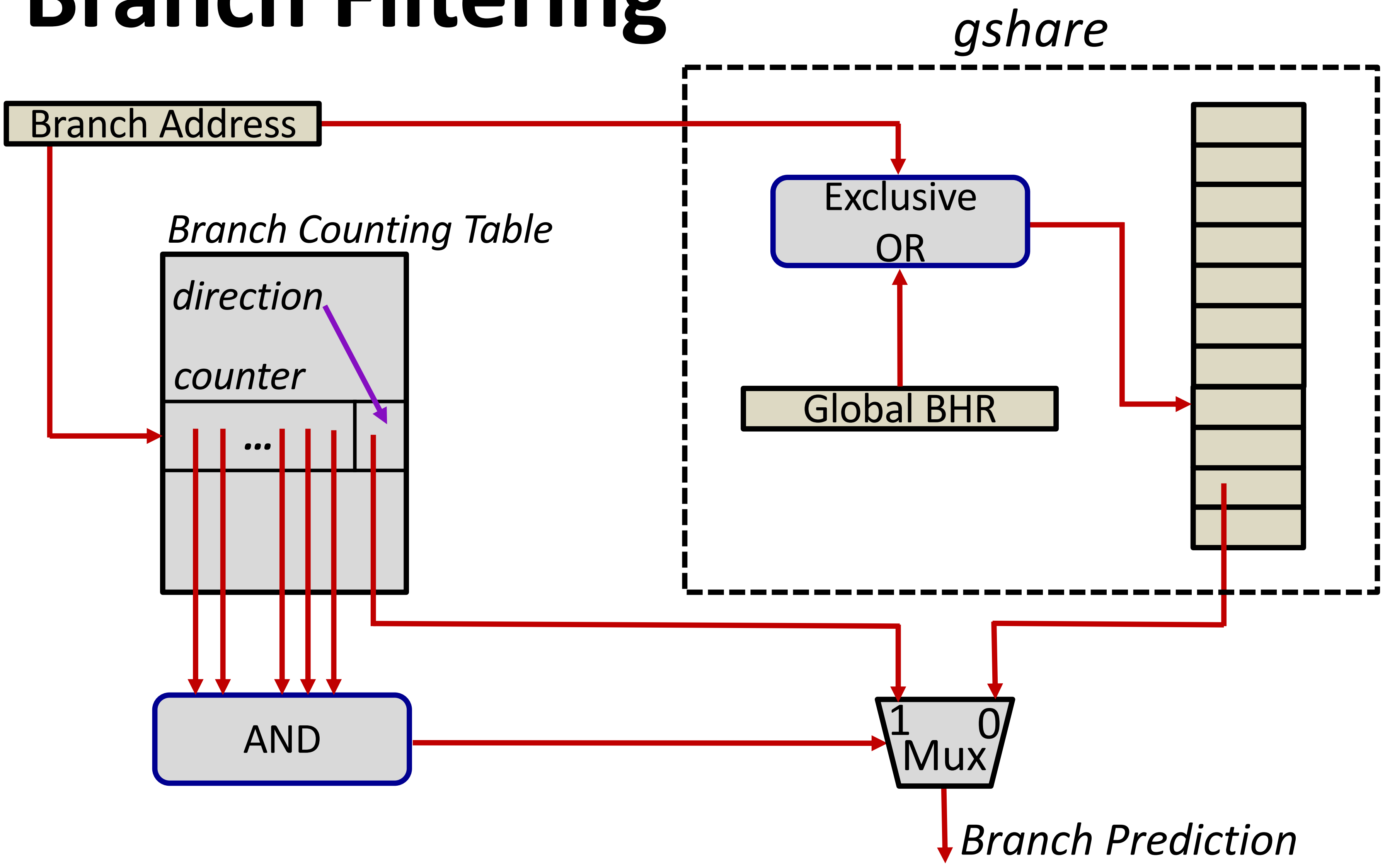
Branch Filtering

Intuition: *Reduce the # branches stored in the PHT by removing highly biaseed branches from the PHT*

Operation

- Track how many times a branch has gone in the same direction
- Beyond a threshold, a branch is “filtered” and no longer updates the PHT
- If the direction changes, reset the counter, and note the new direction

Branch Filtering



Alternative Context Predictors

Tradeoffs in choosing the branch prediction context

- Local or global history
- Length of branch history register
- How many bits of the branch address?

Motivation: *Can we combine all of the above into a single context? Can we use per-branch-type information? Can we use additional information to form context?*

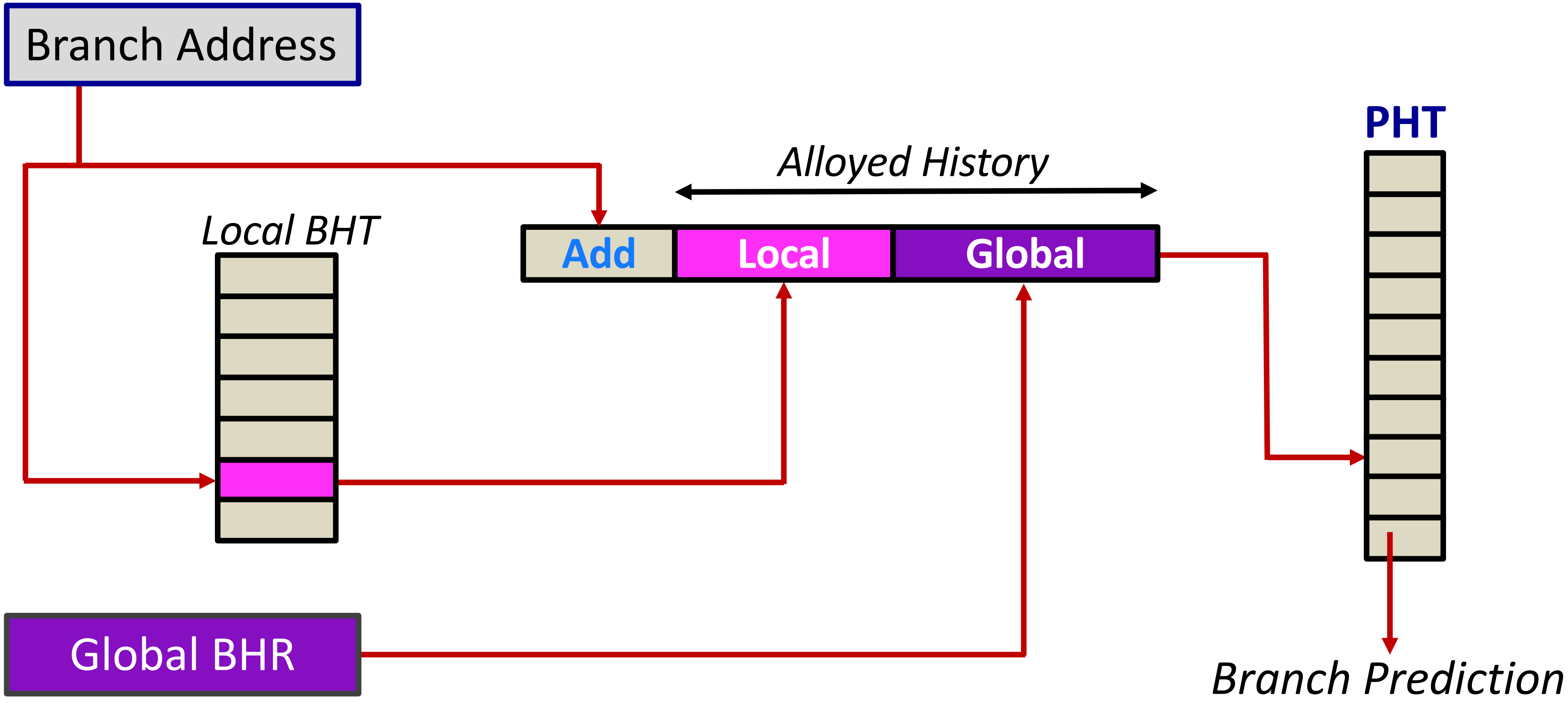
Alloyed History Predictor

Some mispredictions are due to

- Wrong type of history (*wrong-history misprediction*)
- Some branches prefer local, some global, and some both

Motivation: *Distinguish the local and global correlations with the same structure*

Alloyed History Predictor



Loop Counting Predictors

If we want to accurately predict loops, what size BHR do we need for a loop that iterates n times?

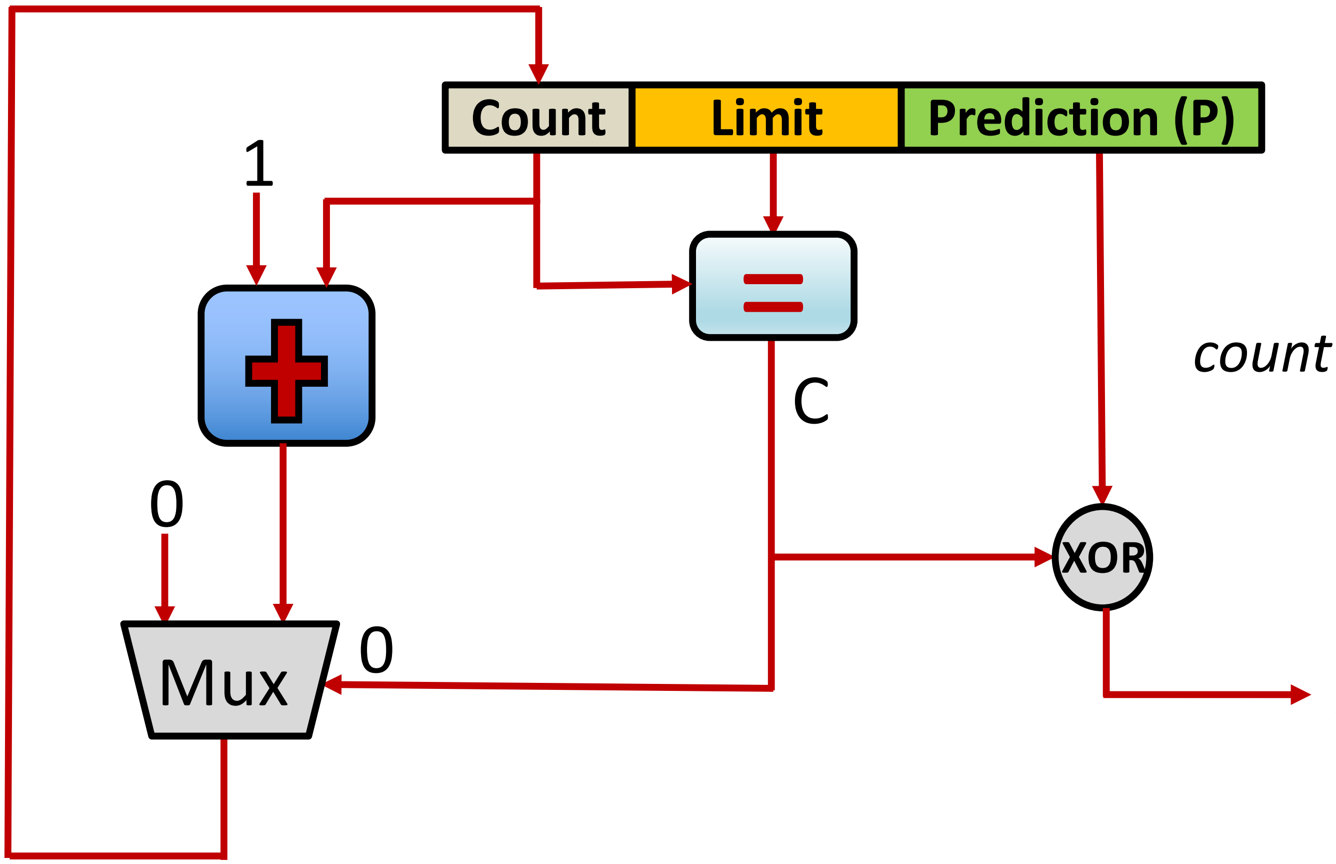
PHT size is exponential in the history length

Loop predictor in Pentium M

- \# iterations (limit)
- Current count
- Direction
- Can detect 11101110 and 00010001

Loop Counting Predictors

The Pentium-M Loop Predictor Table (One entry)



count < limit

P	C	F
0	**0**	0
0	1	1
1	**0**	1
1	1	0

The Perceptron Predictor

Motivation: Increasing the # history bits

- Exponentially increases the PHT size
- Many patterns are irrelevant (training noise)

Question: Can we use more history bits without the exponential increase in area?

- Use perceptron for training the branch predictor
- Use branch history as a feature vector (*not index*)

https://www.youtube.com/watch?v=5g0TPrxKK6o&ab_channel=Udacity

Perceptron

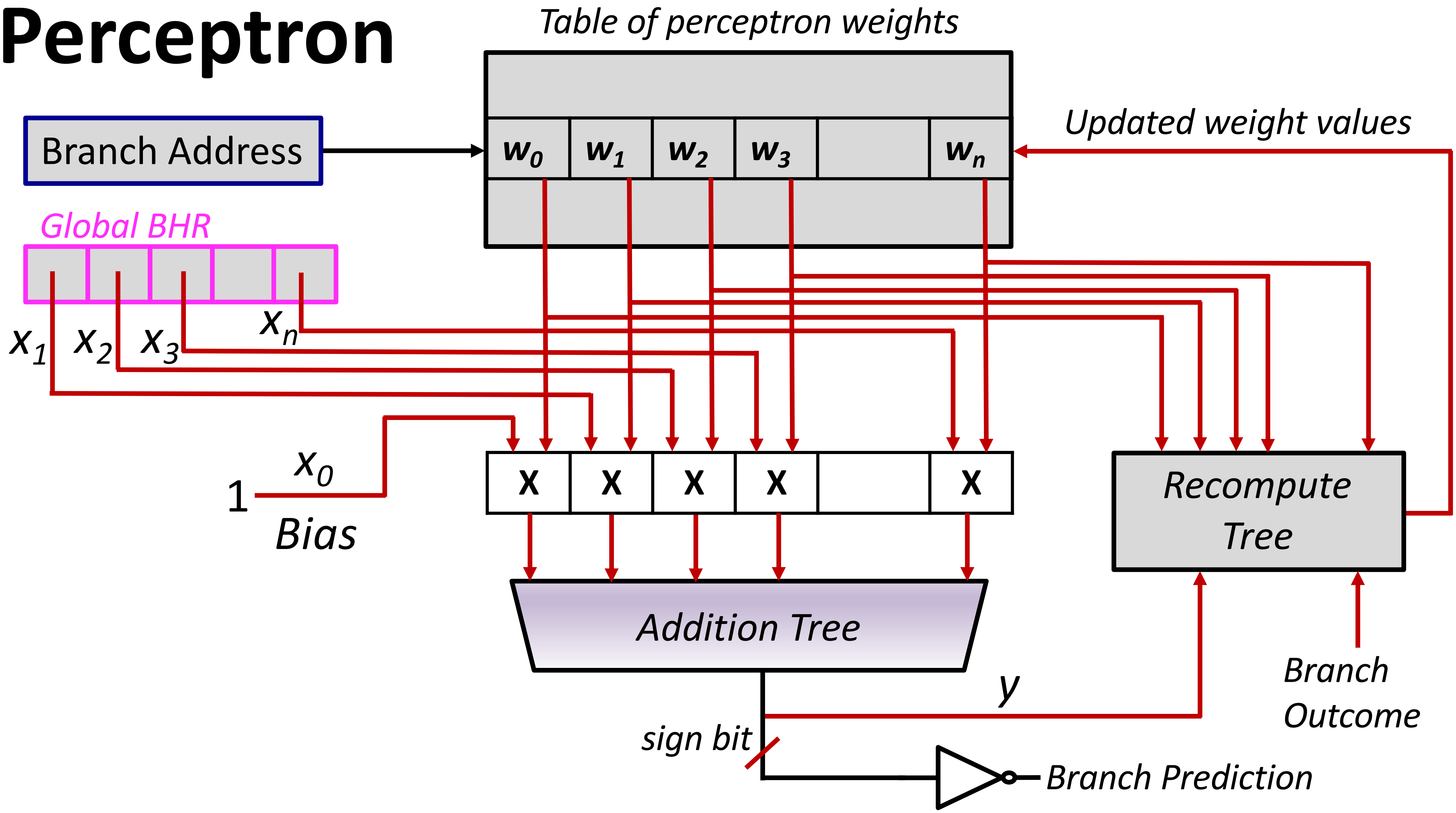


Table of perceptron weights
Branch Address
w_0
w_1
w_2
w_3
w_n
Updated weight values
Global BHR
x_1
x_2
x_3
x_n
x_0
1
Bias
X
X
X
X
X
Recompute Tree
Addition Tree
y
sign bit
Branch Outcome
Branch Prediction

Hybrid Branch Predictors

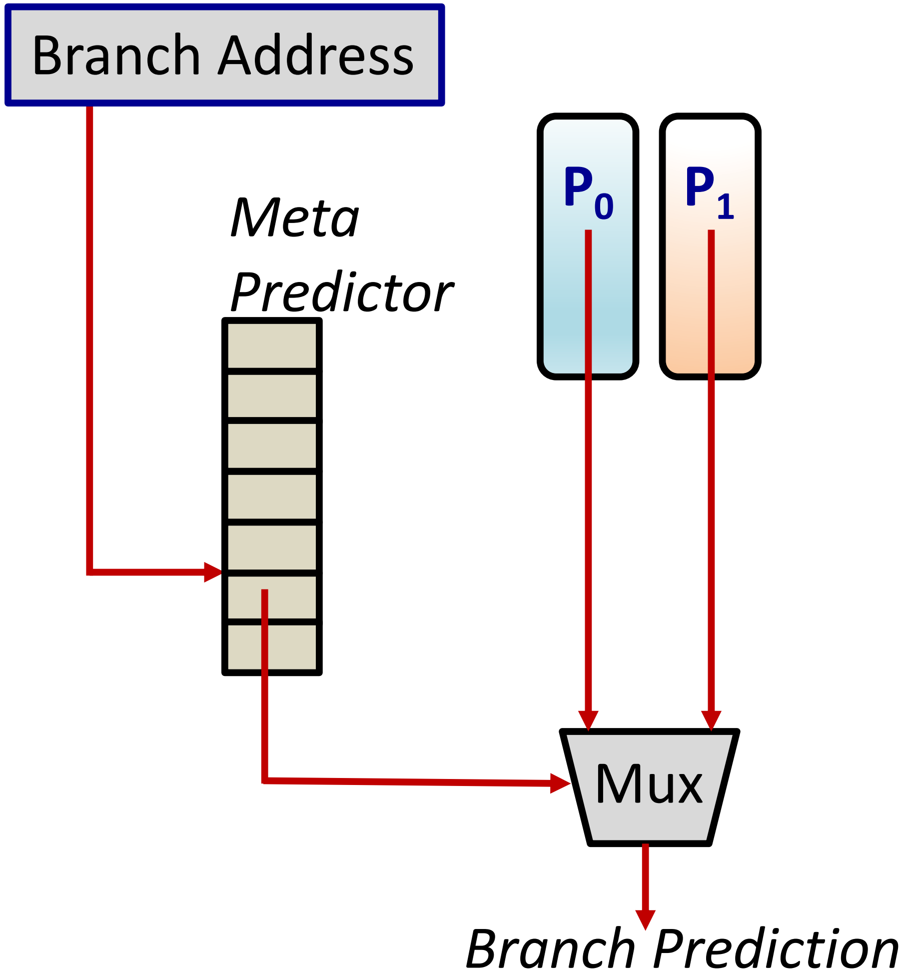
Motivation: *Programs contain a mix of branch types. Different branches may be strongly correlated with different types of history (i.e., global vs local)*

Hybrid branch predictors employ two or more single-scheme branch prediction algorithms

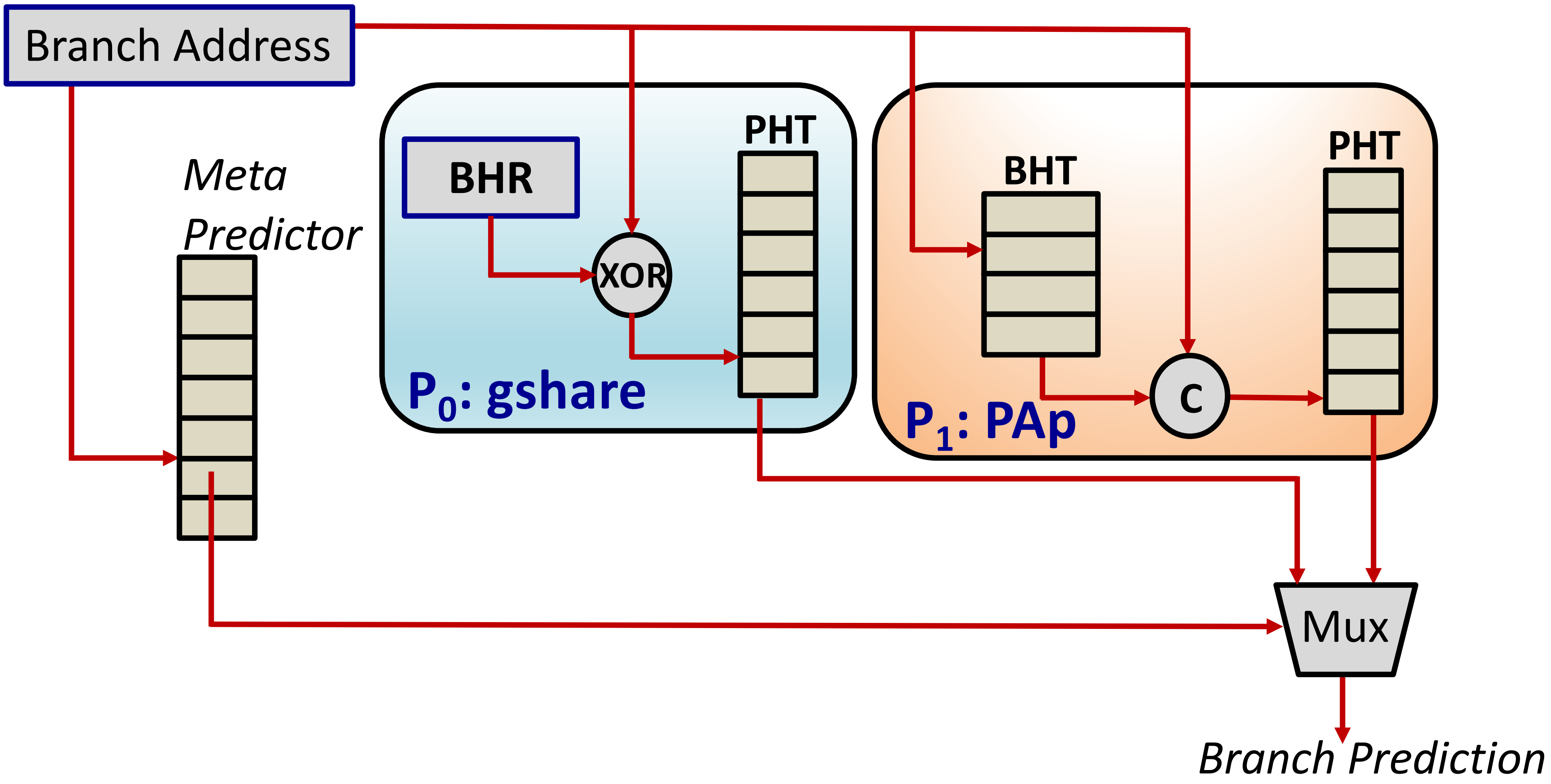
- Combine multiple predictions to make one final prediction

McFarling (1993) proposed the multi-scheme **tournament predictor**

Tournament Predictor

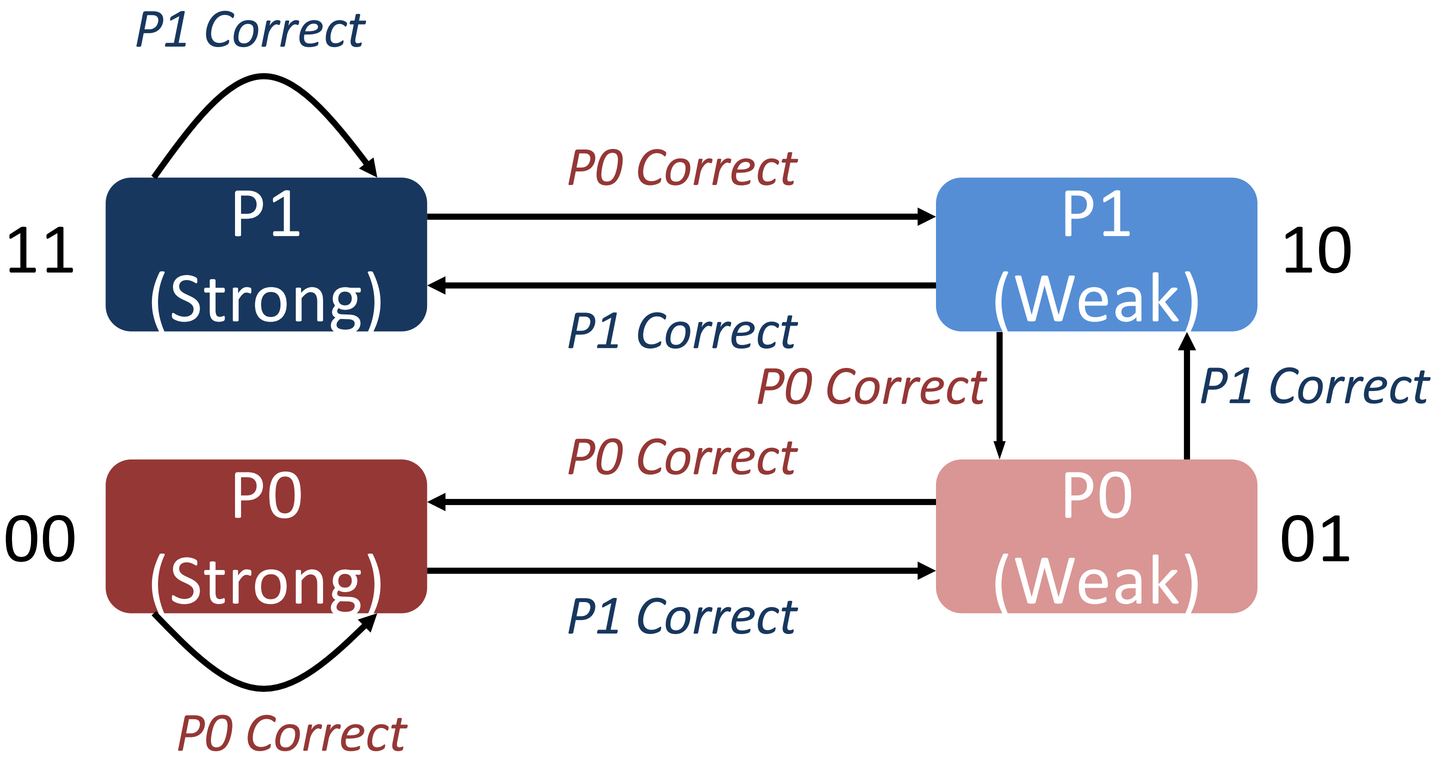


Tournament Predictor



Tournament Meta-Predictor

P0/P1 both correct/incorrect: state unchanged



P1 Correct

11

P1 (Strong)

P0 Correct

P1 Correct

P1 (Weak)

10

P0 Correct

P1 Correct

00

P0 (Strong)

P0 Correct

P1 Correct

P0 (Weak)

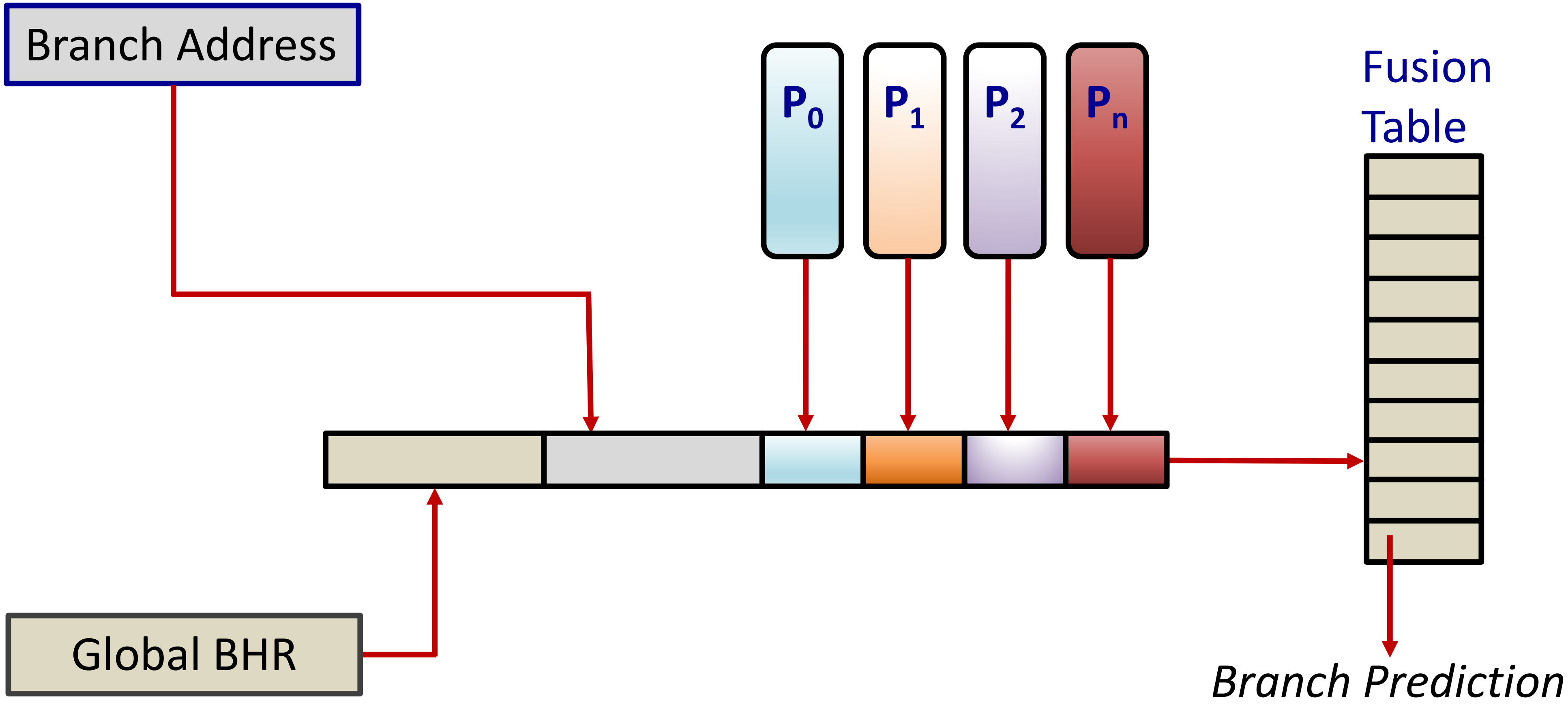
01

P0 Correct

Fusion-Based Hybrid Predictor

Motivation: *Do not throw away the output from any predictor*

Fusion-Based Hybrid Predictor



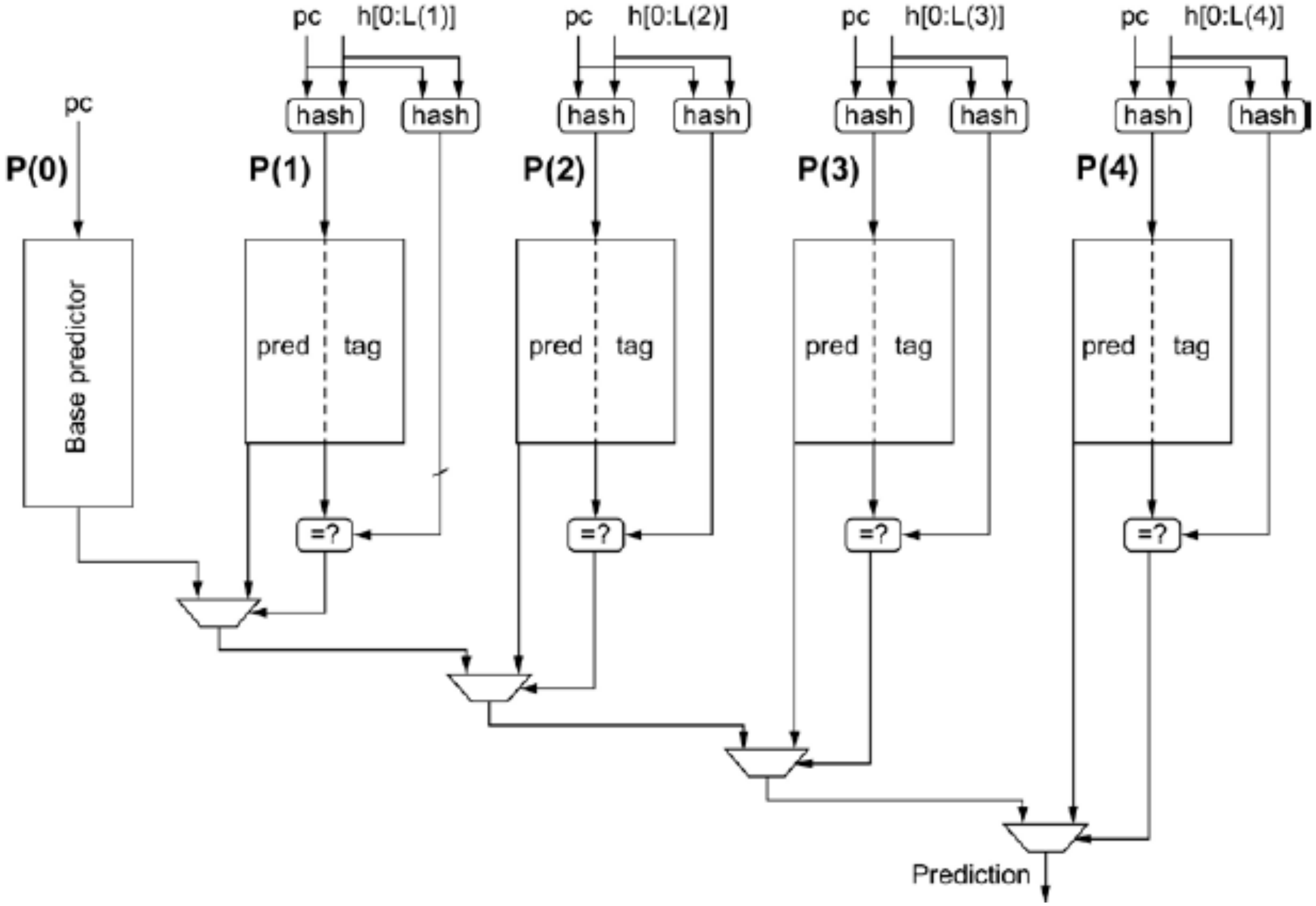
TAGE

TAgged **GE**ometric Predictors
(state-of-the-art)

Three innovations

- Use multiple history lengths
- *History lengths make a geometric series*
- Use tags to alleviate aliasing
- Use bits

Tagged Hybrid Predictors



Pipeline Tradeoff Analysis

Peter Kogge, 1981

$$C = G + k * L$$

$$P = \frac{1}{(\frac{T}{k} + S)}$$

P = Performance

T = non-pipelined-delay

k = depth of the pipeline (# stages)

S = Delay due to the addition of a latch (pipeline register)

C = Cost

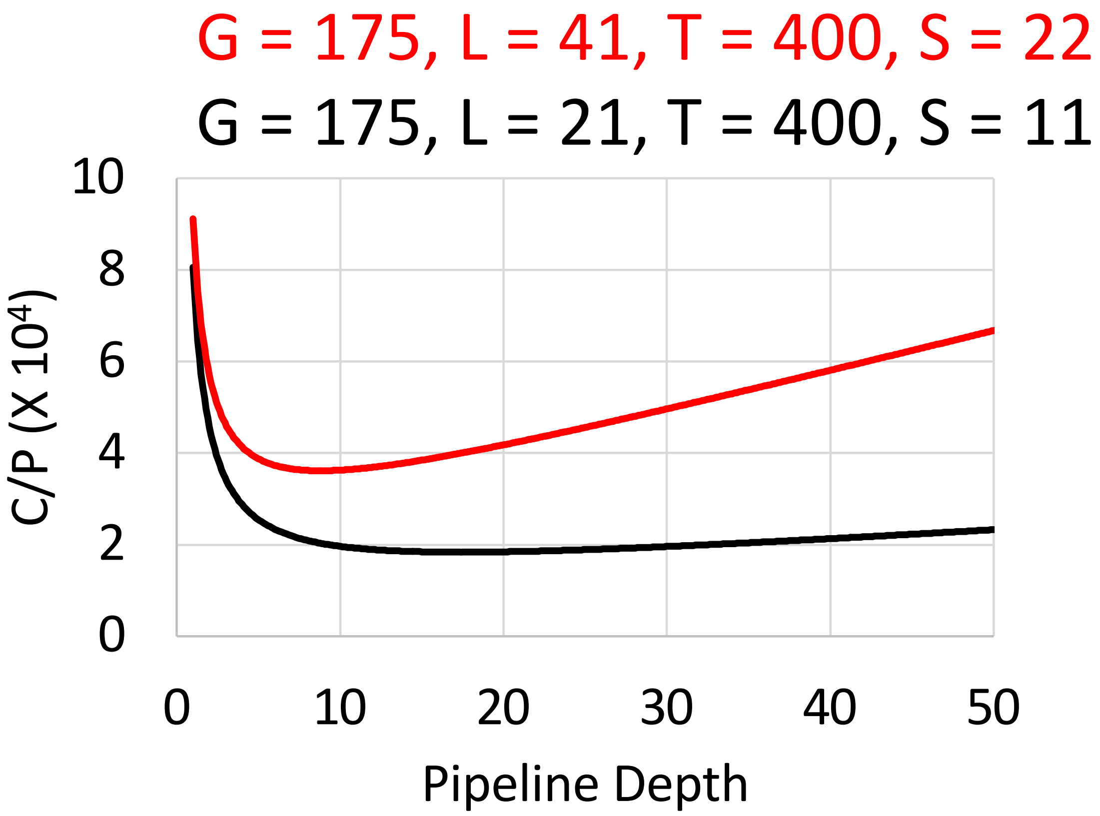
G = Cost in terms of gates in non-pipelined

L = Cost of adding each latch

We can find C/P, the cost to performance ratio, and then take the first derivative, set it to 0, and solve for the value of k that minimizes the cost/performance ratio

$$k_{opt} = \sqrt{\frac{GT}{LS}}$$

Pipeline Tradeoff Analysis



More Recent Study

A. Harstein and Thomas R. Puzak, "The Optimum Pipeline Depth for a Microprocessor," ISCA, 2002