Modern Processor Design (III): Whenever You’re Ready

Hung-Wei Tseng

Recap: how does compiler implement if-else?

```
for(i=0;i<size;i++)
{
    if (data[i] % 2 != 0)
        call_when_true(&data[i]);
    else
        call_when_false(&data[i]);
}
return 0;
```

go to L10 when data[i] % 2 == 1

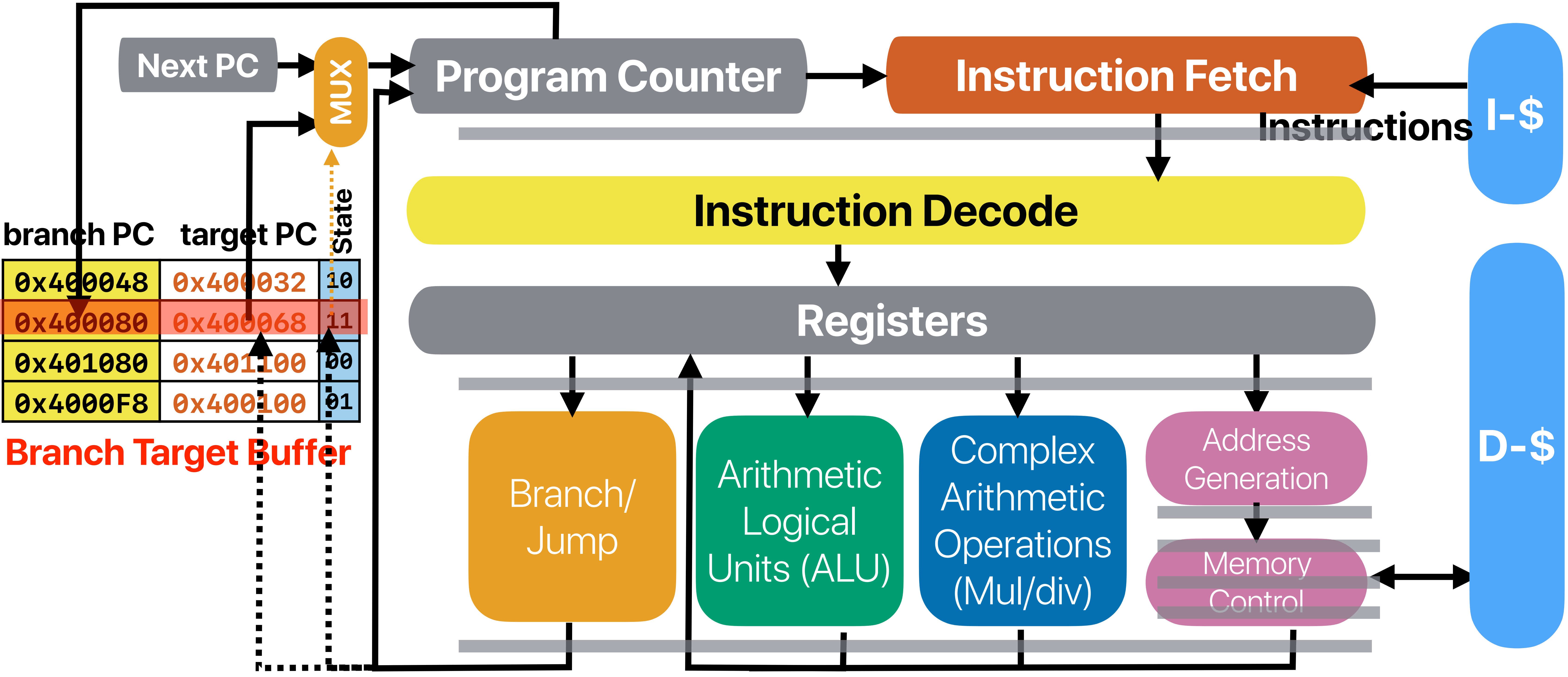
// branch taken if it’s false

branch taken (changing address to L11) if the evaluation is not true

```
.L12:
    movl    -32(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -8(%rbp), %rax
    addq    %rdx, %rax
    movq    (%rax), %rax
    andl    $1, %eax
    testq   %rax, %rax
    je  .L10
    movl    -32(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -8(%rbp), %rax
    addq    %rdx, %rax
    movq    %rax, %rdi
    call    call_when_true
    jmp .L11
.L10:
    movl    -32(%rbp), %eax
    cltq
    leaq    0(,%rax,8), %rdx
    movq    -8(%rbp), %rax
    addq    %rdx, %rax
    movq    %rax, %rdi
    call    call_when_false
.L11:
    addl    $1, -32(%rbp)
.L9:
    movl    -32(%rbp), %eax
    cmpl    -28(%rbp), %eax
    jl  .L12
    movl    $0, %eax
    leave
```

In if-else statement, true can mean “not taken”

Detail of a basic dynamic branch predictor



branch PC	target PC	State
0x400048	0x400032	10
0x400080	0x400068	11
0x401080	0x401100	00
0x4000F8	0x400100	01

Branch Target Buffer

2-bit local predictor

- What's the overall branch prediction (include both branches) accuracy for this nested for loop?

```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```

Can we do a better job?

(assume all states started with 00)

A. ~25%

B. ~33%

C. ~50%

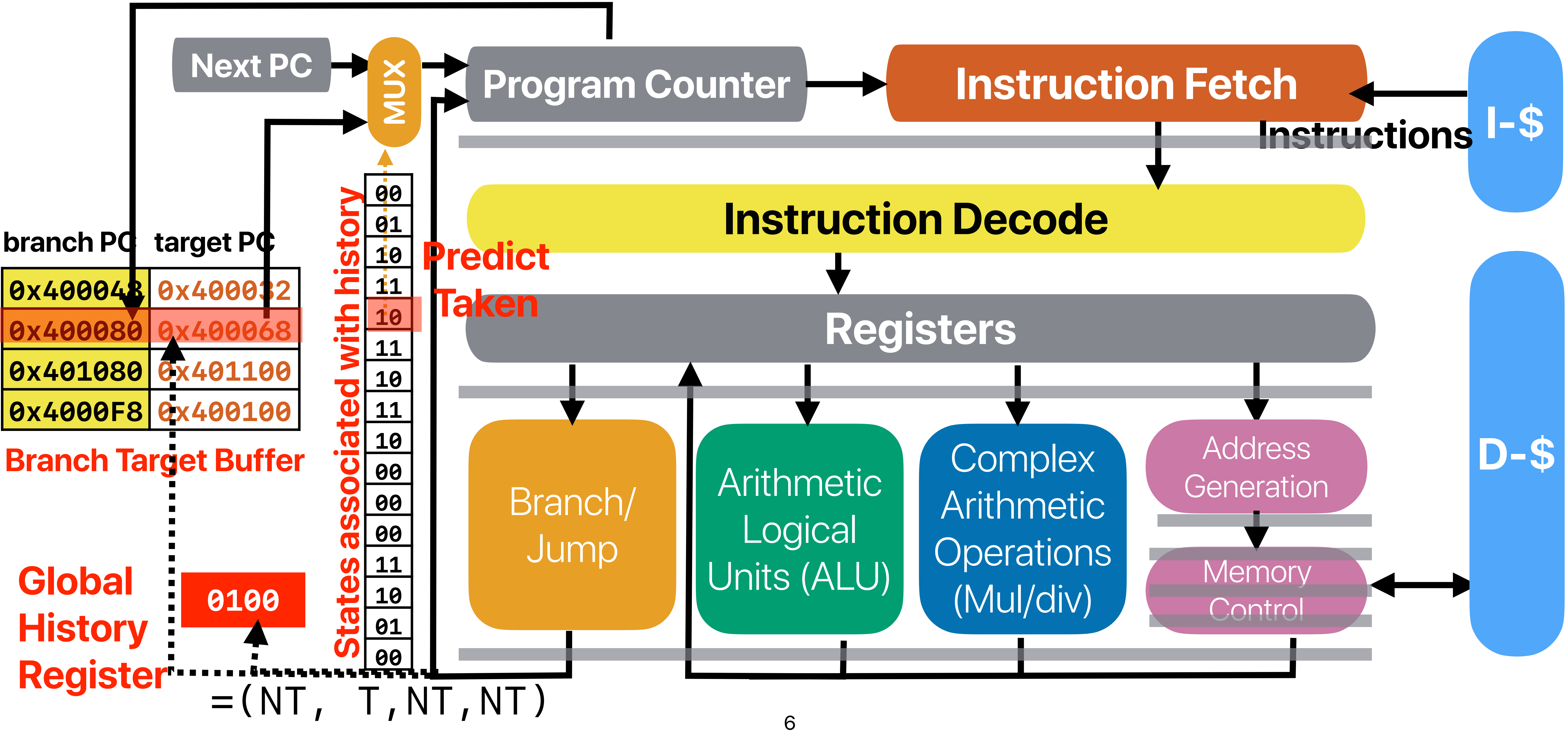
D. ~67%

E. ~75%

i	branch?	state	prediction	actual
0	X	00	NT	T
1	Y	00	NT	T
1	X	01	NT	NT
2	Y	01	NT	T
2	X	00	NT	T
3	Y	10	T	T
3	X	01	NT	NT
4	Y	11	T	T
4	X	00	NT	T
5	Y	11	T	T
5	X	01	NT	NT
6	Y	11	T	T
6	X	00	NT	T
7	Y	11	T	T

For branch Y, almost 100%,
For branch X, only 50%

Detail of a basic dynamic branch predictor



branch PC	target PC
0x400048	0x400032
0x400080	0x400068
0x401080	0x401100
0x4000F8	0x400100

Branch Target Buffer

States associated with history

00
01
10
11
10
11
10
11
10
00
00
00
11
10
01
00

Predict Taken

Global History Register

0100

=(NT, T,NT,NT)

Performance of GH predictor

```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```

i	branch?	GHR	state	prediction	actual
0	X	000	00	NT	T
1	Y	001	00	NT	T
1	X	011	00	NT	NT
2	Y	110	00	NT	T
2	X	101	00	NT	T
3	Y	011	00	NT	T
3	X	111	00	NT	NT
4	Y	110	01	NT	T
4	X	101	01	NT	T
5	Y	011	01	NT	T
5	X	111	00	NT	NT
6	Y	110	10	T	T
6	X	101	10	T	T
7	Y	011	10	T	T
7	X	111	00	NT	NT
8	Y	110	11	T	T
8	X	101	11	T	T
9	Y	011	11	T	T
9	X	111	00	NT	NT
10	Y	110	11	T	T
10	X	101	11	T	T
11	Y	011	11	T	T

Near perfect after this

Demo revisited: evaluating the cost of mis-predicted branches

- Compare the number of mis-predictions
- Calculate the difference of cycles
- We can get the “average CPI” of a mis-prediction!

34 cycles on Intel Alder Lake

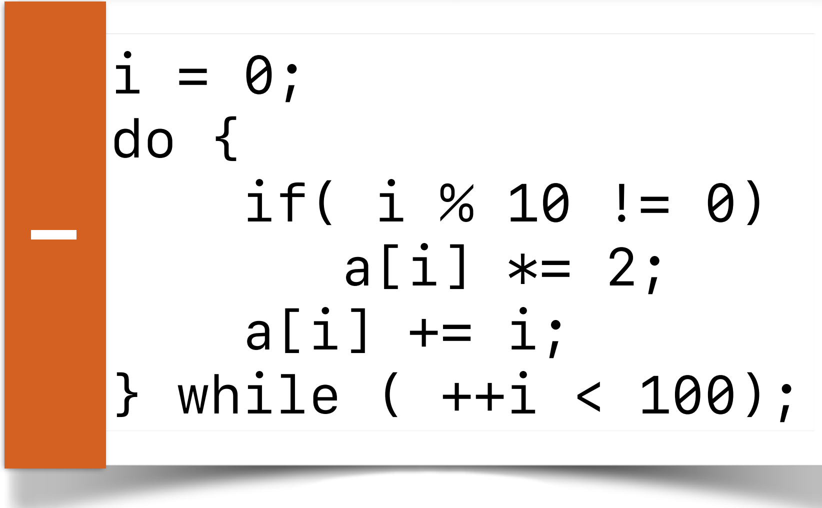
24 cycles on AMD Zen 3

Could be more expensive than cache misses

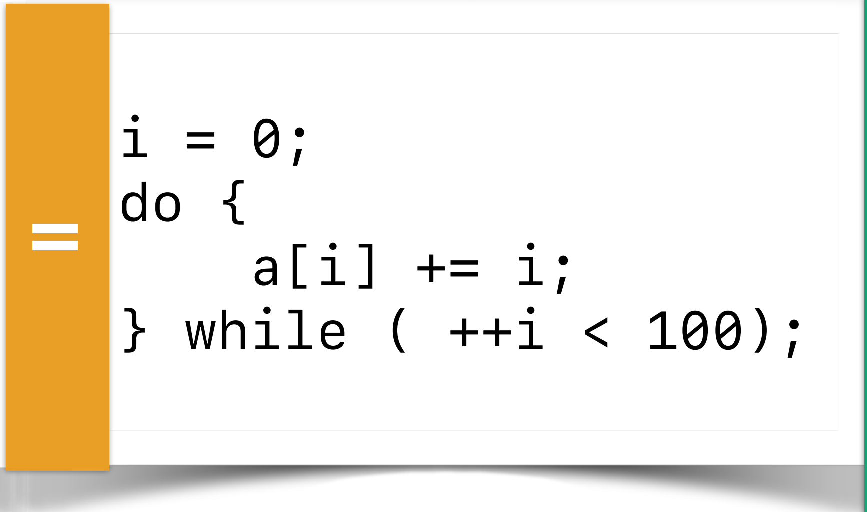
https://www.pollev.com/hungweitseng close in 1:30

Better predictor?

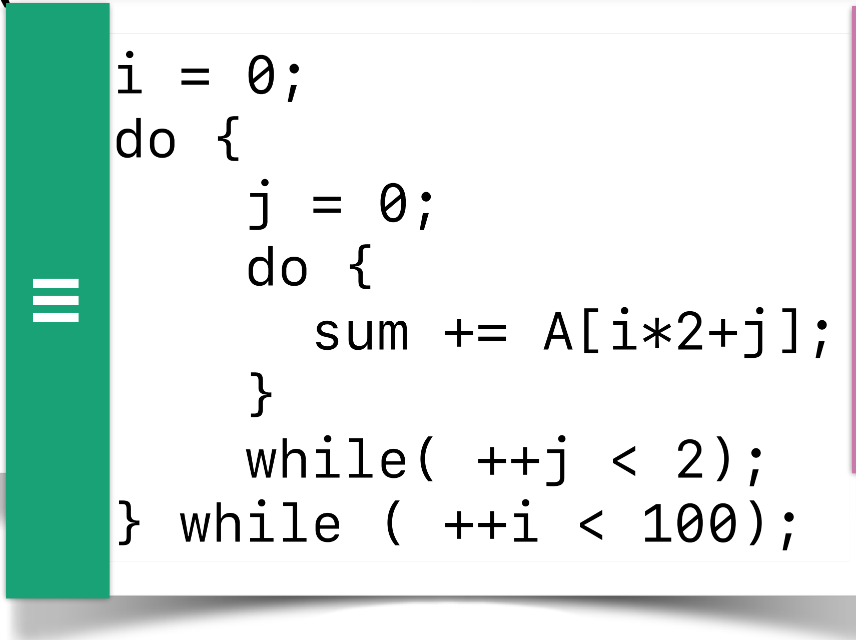
- Consider two predictors — (L) 2-bit local predictor with unlimited BTB entries and (G) 4-bit global history with 2-bit predictors. How many of the following code snippet would allow (G) to outperform (L)?


I

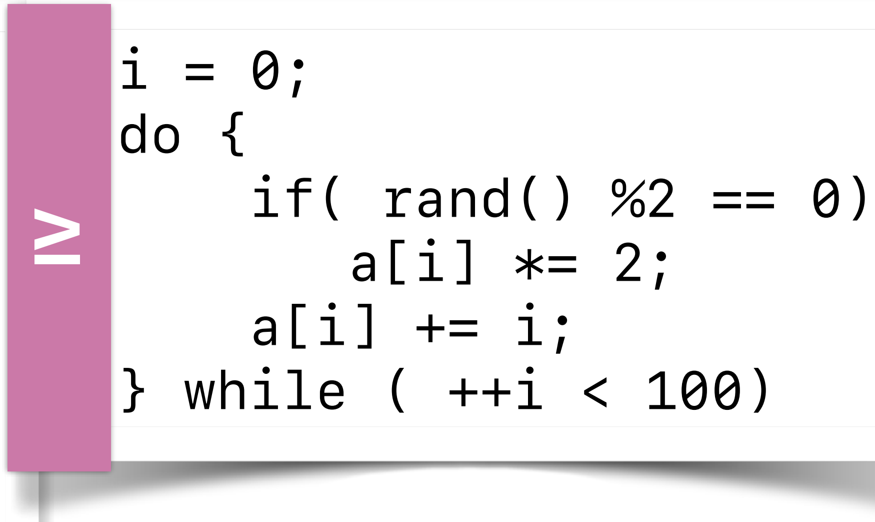
```
i = 0;
do {
    if( i % 10 != 0)
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100);
```


II

```
i = 0;
do {
    a[i] += i;
} while ( ++i < 100);
```


III

```
i = 0;
do {
    j = 0;
    do {
      sum += A[i*2+j];
    }
    while( ++j < 2);
} while ( ++i < 100);
```

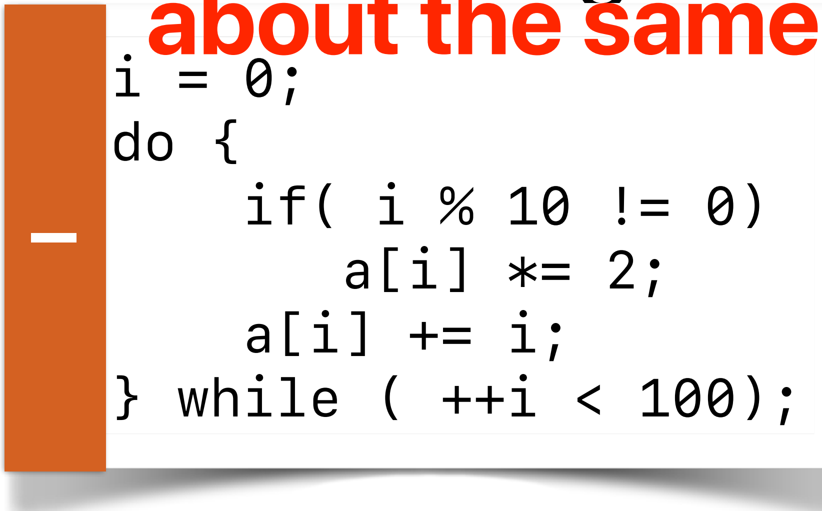

IV

```
i = 0;
do {
    if( rand() %2 == 0)
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)
```

A. 0
B. 1
C. 2
D. 3
E. 4

Better predictor?

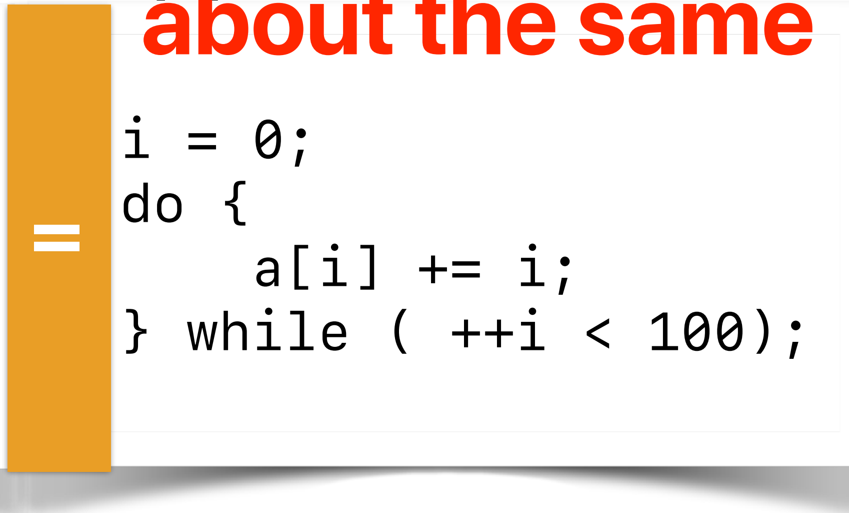
- Consider two predictors — (L) 2-bit local predictor with unlimited BTB entries and (G) 4-bit global history with 2-bit predictors. How many of the following code snippet would allow (G) to outperform (L)?



about the same

I

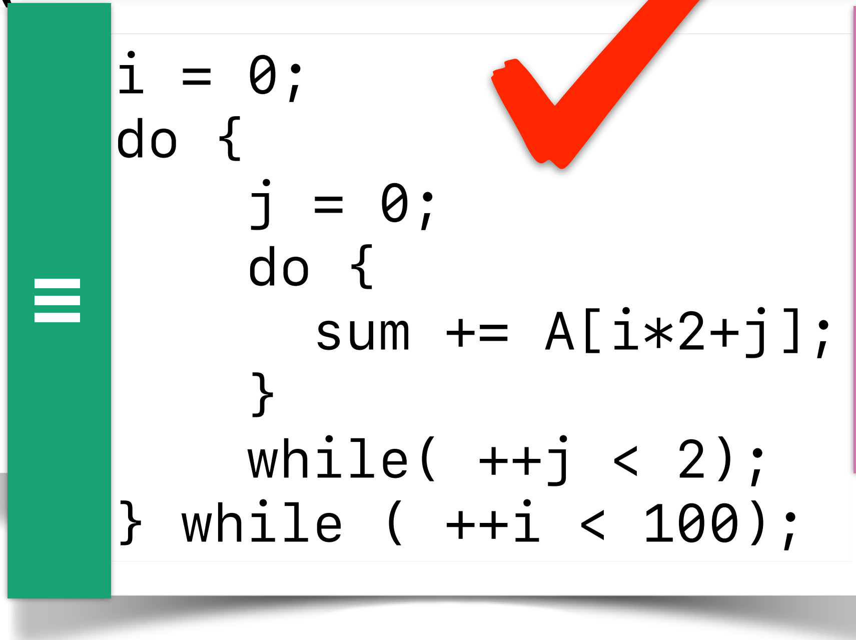
```
i = 0;
do {
    if( i % 10 != 0)
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100);
```



about the same

II

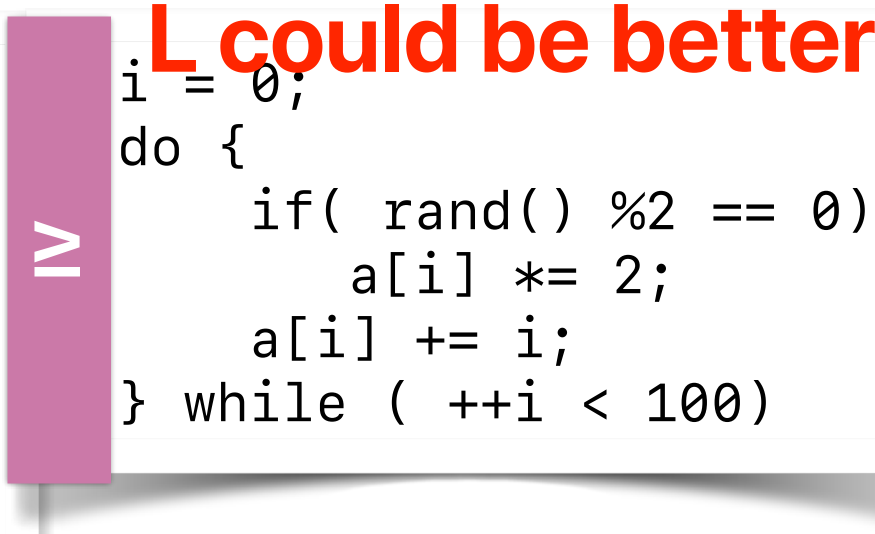
```
i = 0;
do {
    a[i] += i;
} while ( ++i < 100);
```



✓

III

```
i = 0;
do {
    j = 0;
    do {
      sum += A[i*2+j];
    }
    while( ++j < 2);
} while ( ++i < 100);
```



L could be better

IV

```
i = 0;
do {
    if( rand() %2 == 0)
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)
```

A. 0

[✓ B. 1]

C. 2

D. 3

E. 4

Takeaways: branch predictions

- The cost of not to predict a branch is to stall until the data dependency is resolved — 34 cycles on modern intel processors and 24 on AMD processors
- Branch predictions allow the processor to at least make some progress and hide the stalls if we guessed correctly!
- Dynamic branch prediction — predict based on prior history
  - Local predictor — make predictions based on the state of each branch instruction
  - Global predictor — make predictions based on the state from all branches
  - Both are not perfect

Takeaways: branch predictions

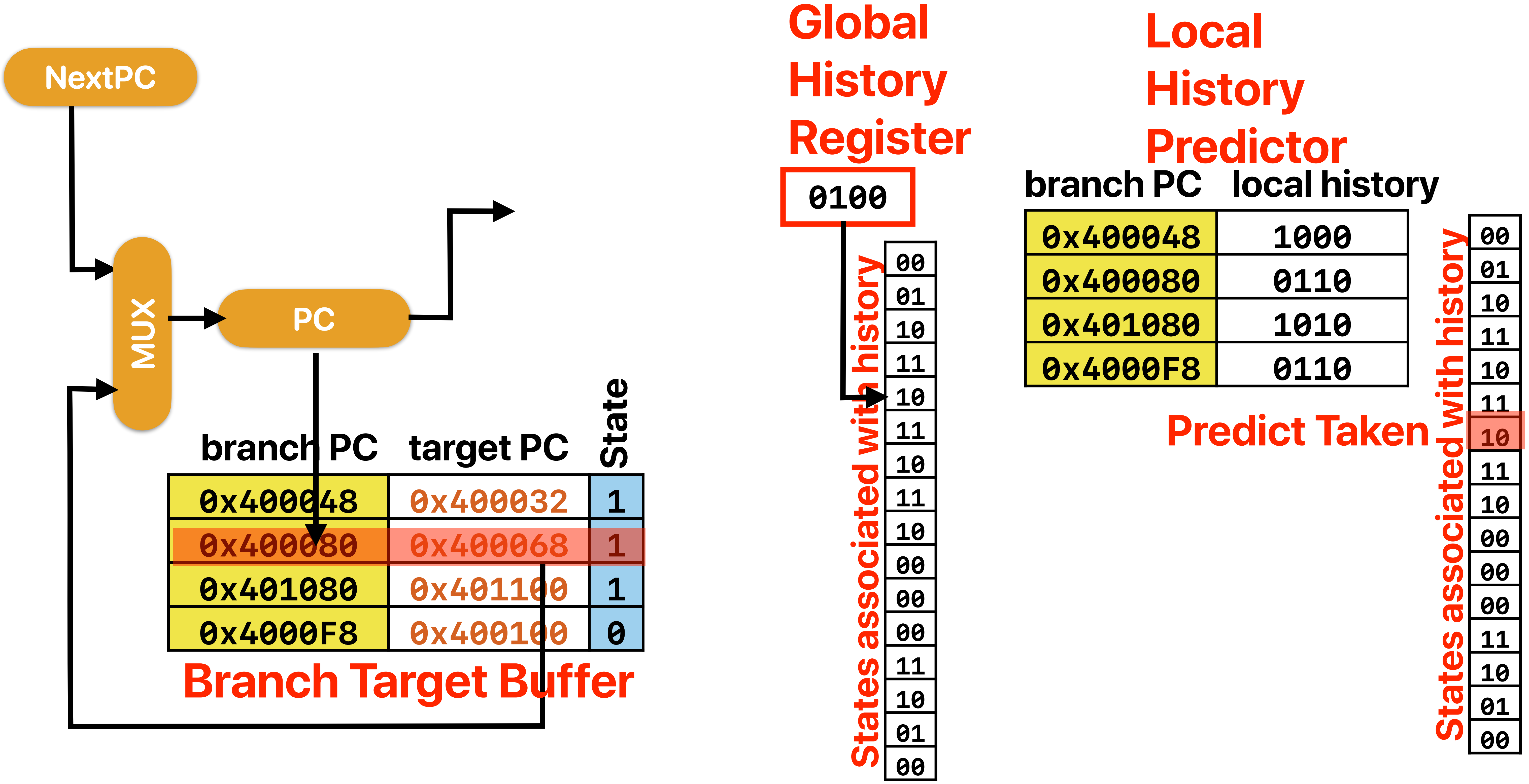
- The cost of not to predict a branch is to stall until the data dependency is resolved — 34 cycles on modern intel processors and 23 on AMD processors
- Branch predictions allow the processor to at least make some progress and hide the stalls if we guessed correctly!
- Dynamic branch prediction — predict based on prior history
  - Local predictor — make predictions based on the state of each branch instruction
  - Global predictor — make predictions based on the state from all branches
  - Both are not perfect

Outline

- Hybrid predictors (cont.)
- Data hazards
- Hardware optimizations for data hazards

Hybrid predictors

Tournament Predictor



branch PC	target PC	State
0x400048	0x400032	1
0x400080	0x400068	1
0x401080	0x401100	1
0x4000F8	0x400100	0

Branch Target Buffer

Global History Register: 0100

States associated with history
00
01
10
11
10
11
10
11
10
00
00
00
11
10
01
00

Local History Predictor

branch PC	local history
0x400048	1000
0x400080	0110
0x401080	1010
0x4000F8	0110

Predict Taken

States associated with history
00
01
10
11
10
11
10
11
10
00
00
00
11
10
01
00

Tournament Predictor

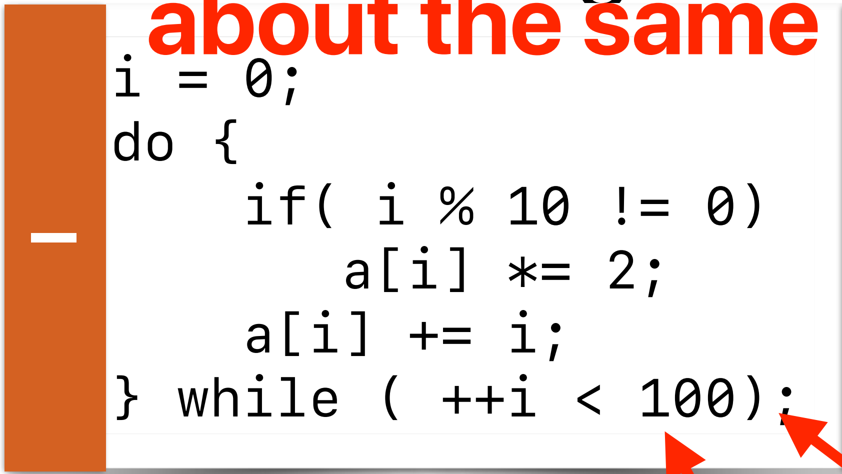
- The state predicts “which predictor is better”
    - Local history
    - Global history
- The predicted predictor makes the prediction
- Tournament predictor is a “hybrid predictor” as it takes both local & global information into account

TAGE

André Seznec. The L-TAGE branch predictor. Journal of Instruction Level Parallelism (http://wwwjilp.org/vol9), May 2007.

Better predictor?

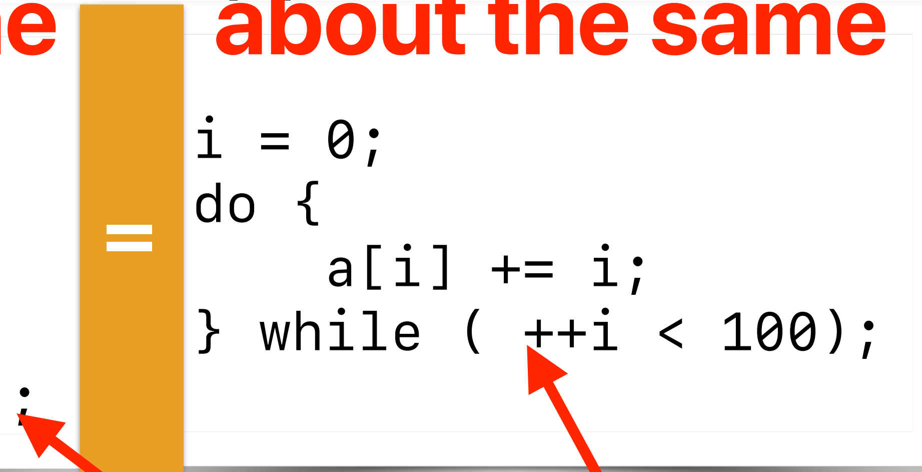
- Consider two predictors — (L) 2-bit local predictor with unlimited BTB entries and (G) 4-bit global history with 2-bit predictors. How many of the following code snippet would allow (G) to outperform (L)?



I

about the same

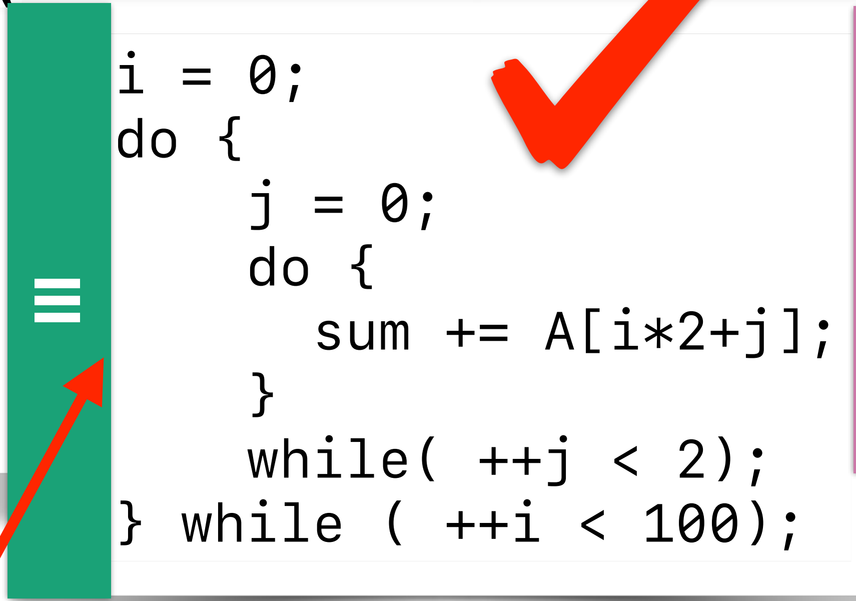
```
i = 0;
do {
    if( i % 10 != 0)
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100);
```



II

about the same

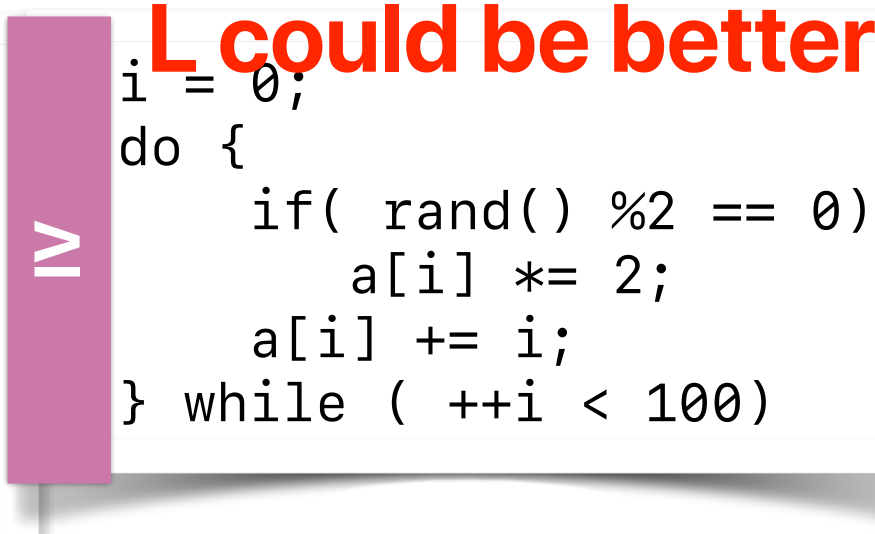
```
i = 0;
do {
    a[i] += i;
} while ( ++i < 100);
```



III

✓

```
i = 0;
do {
    j = 0;
    do {
      sum += A[i*2+j];
    }
    while( ++j < 2);
} while ( ++i < 100);
```



IV

L could be better

```
i = 0;
do {
    if( rand() %2 == 0)
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)
```

A. 0

B. 1

C. 2

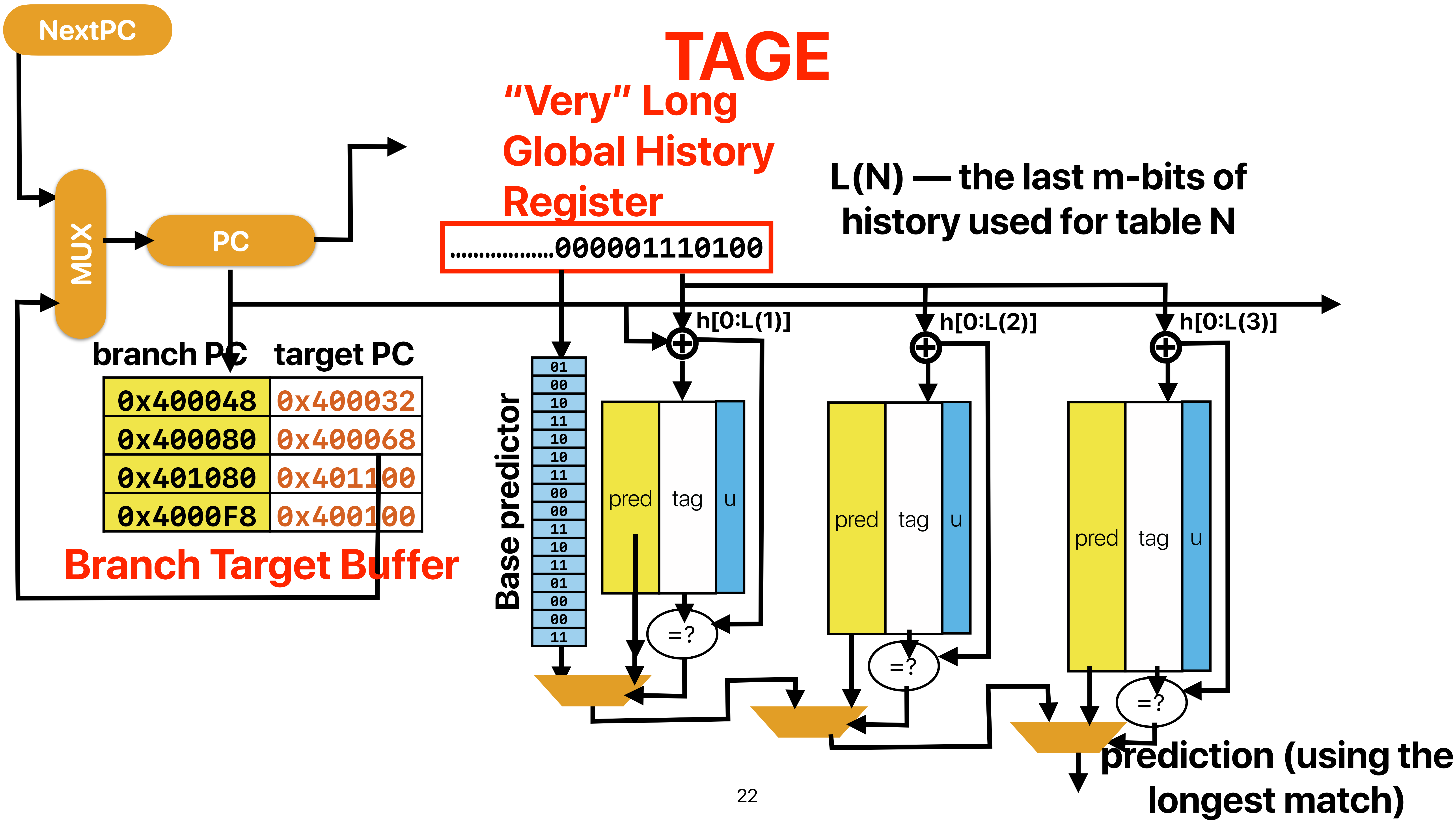
D. 3

E. 4

different branch needs different length of history

global predictor can work if the history is long enough!

TAGE



"Very" Long Global History Register

……………….000001110100

L(N) — the last m-bits of history used for table N

branch PC	target PC
0x400048	0x400032
0x400080	0x400068
0x401080	0x401100
0x4000F8	0x400100

Branch Target Buffer

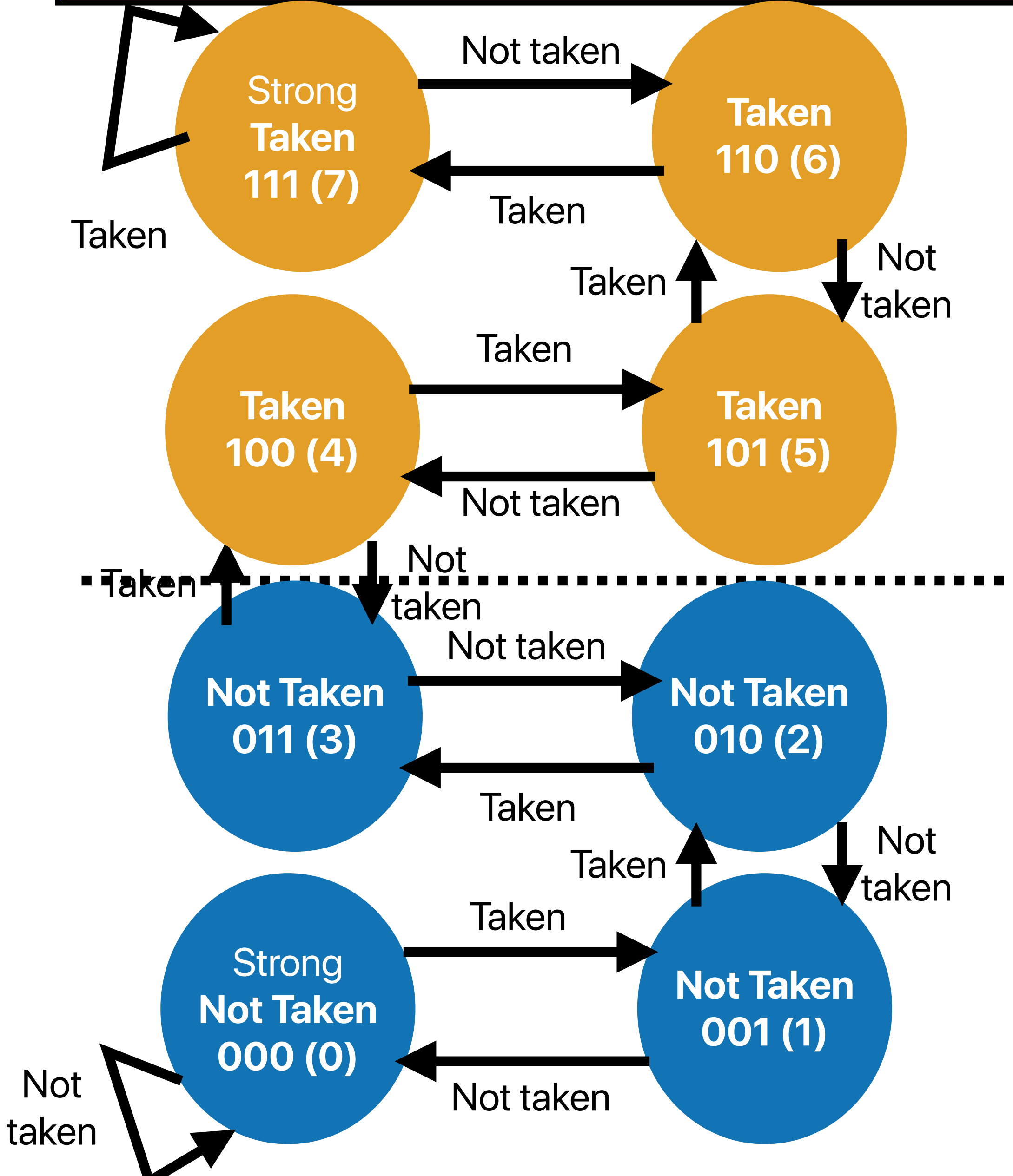
Base predictor

01
00
10
11
10
10
11
00
00
11
10
11
01
00
00
11

prediction (using the longest match)

What's inside each table?

pred **(3-bit counter)**	**tag** **(partial branch PC)**	**u (usefulness)**



$$\mathit{if}\ \mathit{prediction}(\mathit{alt\_predictor}) \neq \mathit{prediction}(\mathit{pred}):$$

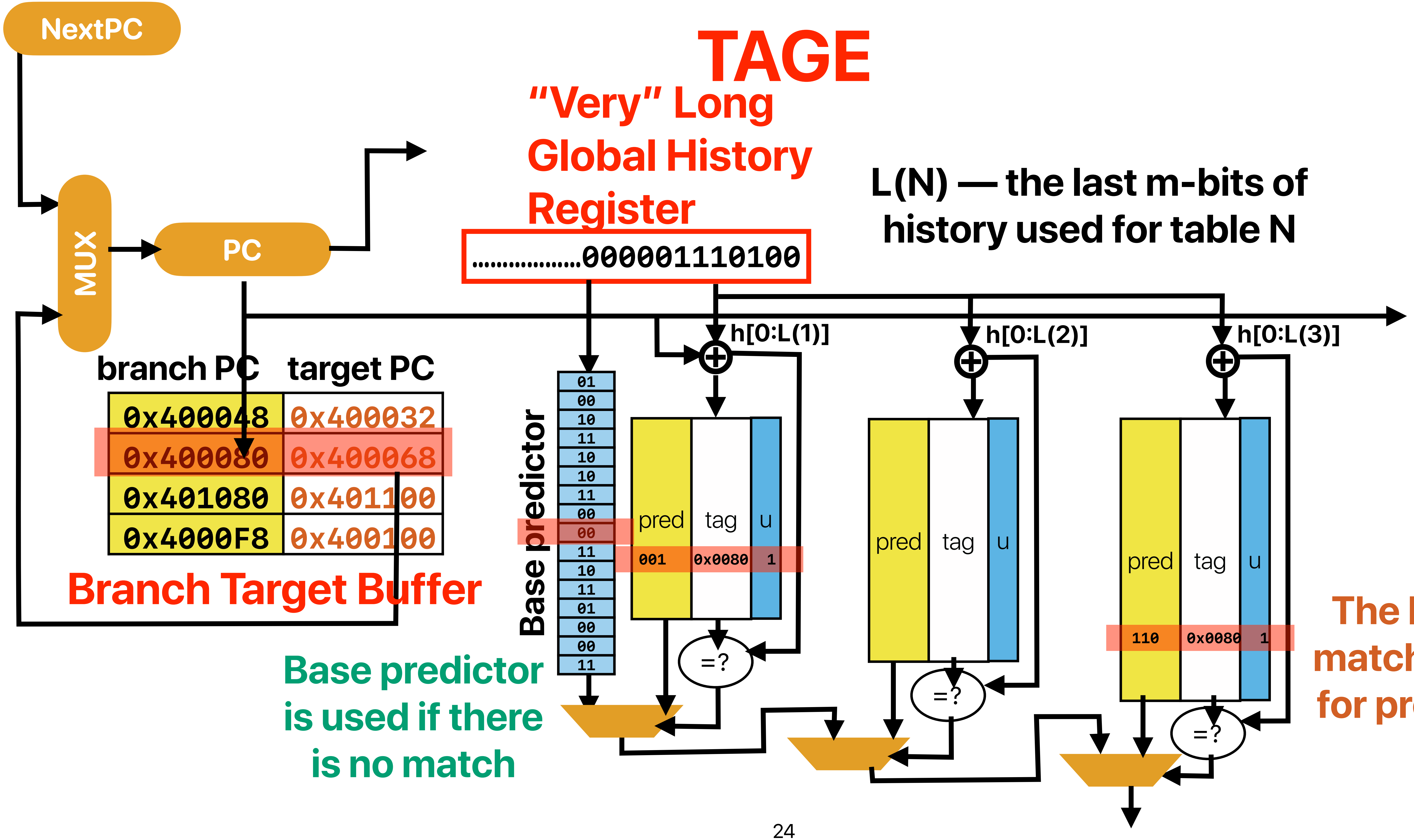
$$\mathit{if}\ \mathit{prediction}(\mathit{pred}) = \mathit{actual\ result}: u = u + 1$$

$$\mathit{if}\ \mathit{prediction}(\mathit{pred}) \neq \mathit{actual\ result}: u = u - 1$$

TAGE

"Very" Long Global History Register

L(N) — the last m-bits of history used for table N



branch PC	target PC
0x400048	0x400032
0x400080	0x400068
0x401080	0x401100
0x4000F8	0x400100

Branch Target Buffer

Base predictor is used if there is no match

The longest match is used for prediction

Perceptron

Jiménez, Daniel, and Calvin Lin. "Dynamic branch prediction with perceptrons." Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture. IEEE, 2001.

The following slides are excerpted from https://www.jilp.org/cbp/Daniel-slides.PDF by Daniel Jiménez

Branch Prediction is Essentially an ML Problem

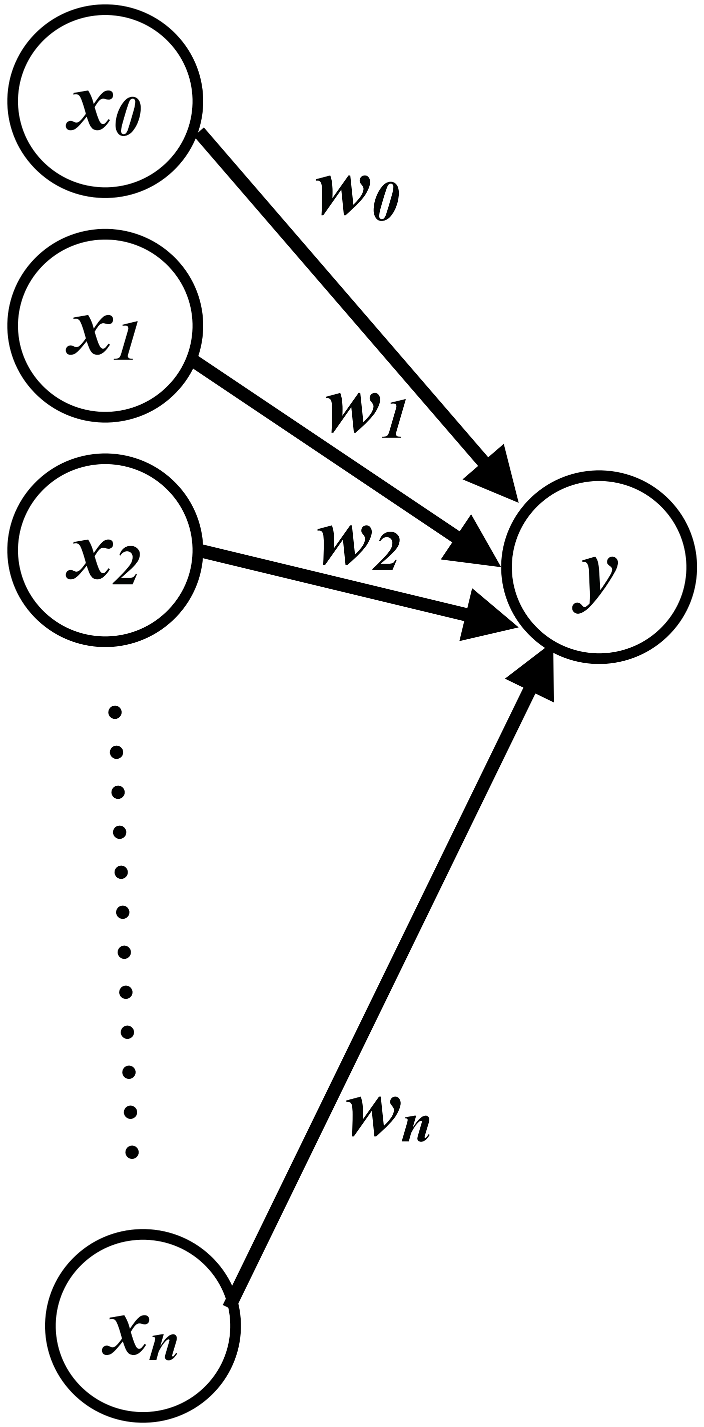
- The machine learns to predict conditional branches
- Artificial neural networks
    - Simple model of neural networks in brain cells
    - Learn to recognize and classify patterns

Mapping Branch Prediction to NN

- The inputs to the perceptron are branch outcome histories
    - Just like in 2-level adaptive branch prediction
    - Can be global or local (per-branch) or both (alloyed)
    - Conceptually, branch outcomes are represented as
        - +1, for taken
        - -1, for not taken
- The output of the perceptron is
    - Non-negative, if the branch is predicted taken
    - Negative, if the branch is predicted not taken
- Ideally, each static branch is allocated its own perceptron

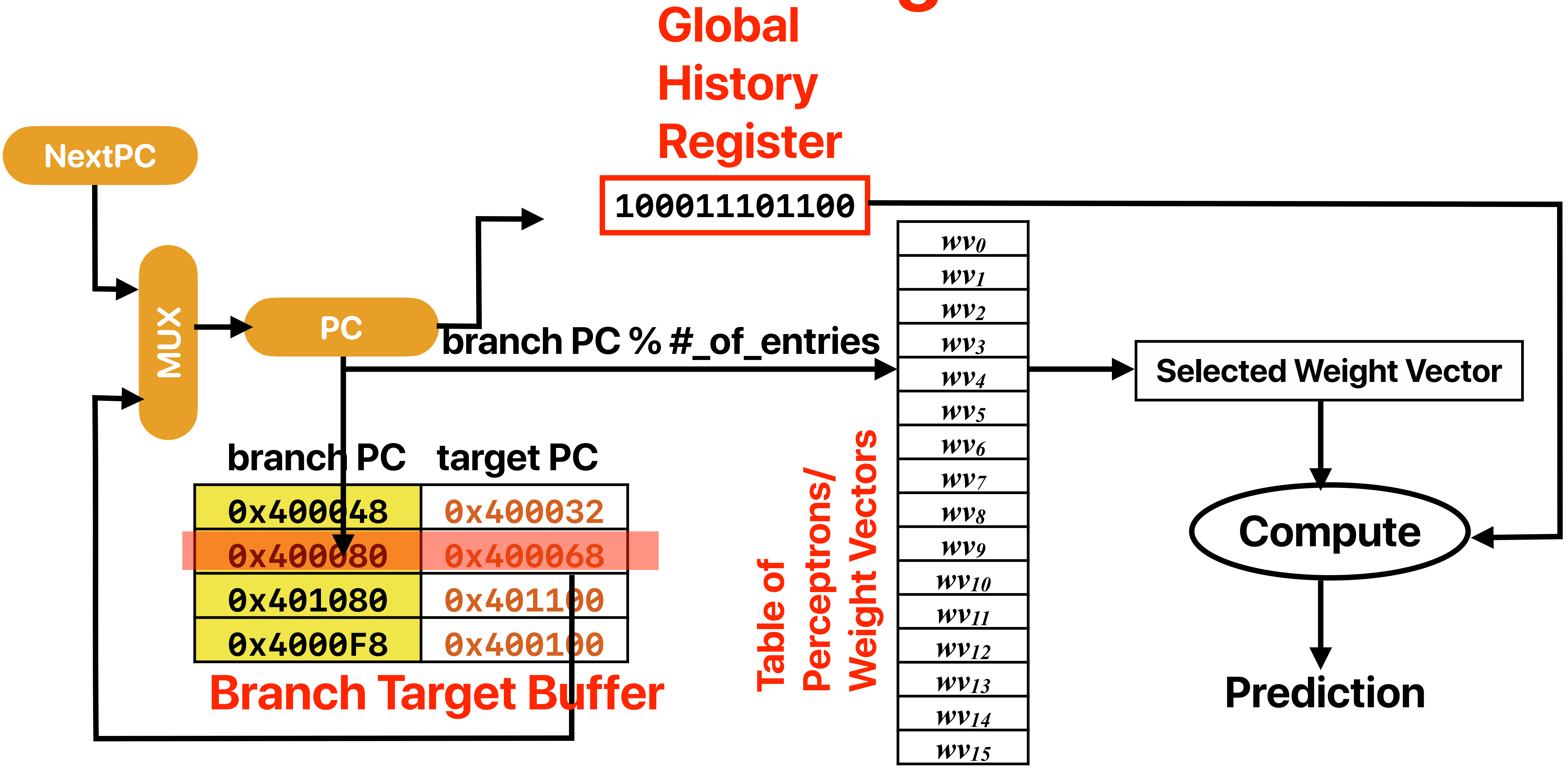
Mapping Branch Prediction to NN (cont.)

- Inputs (x’s) are from branch history and are -1 or +1
- n + 1 small integer weights (w’s) learned by on-line training
- Output (y) is dot product of x’s and w’s; predict taken if y = 0
- Training finds correlations between history and outcome



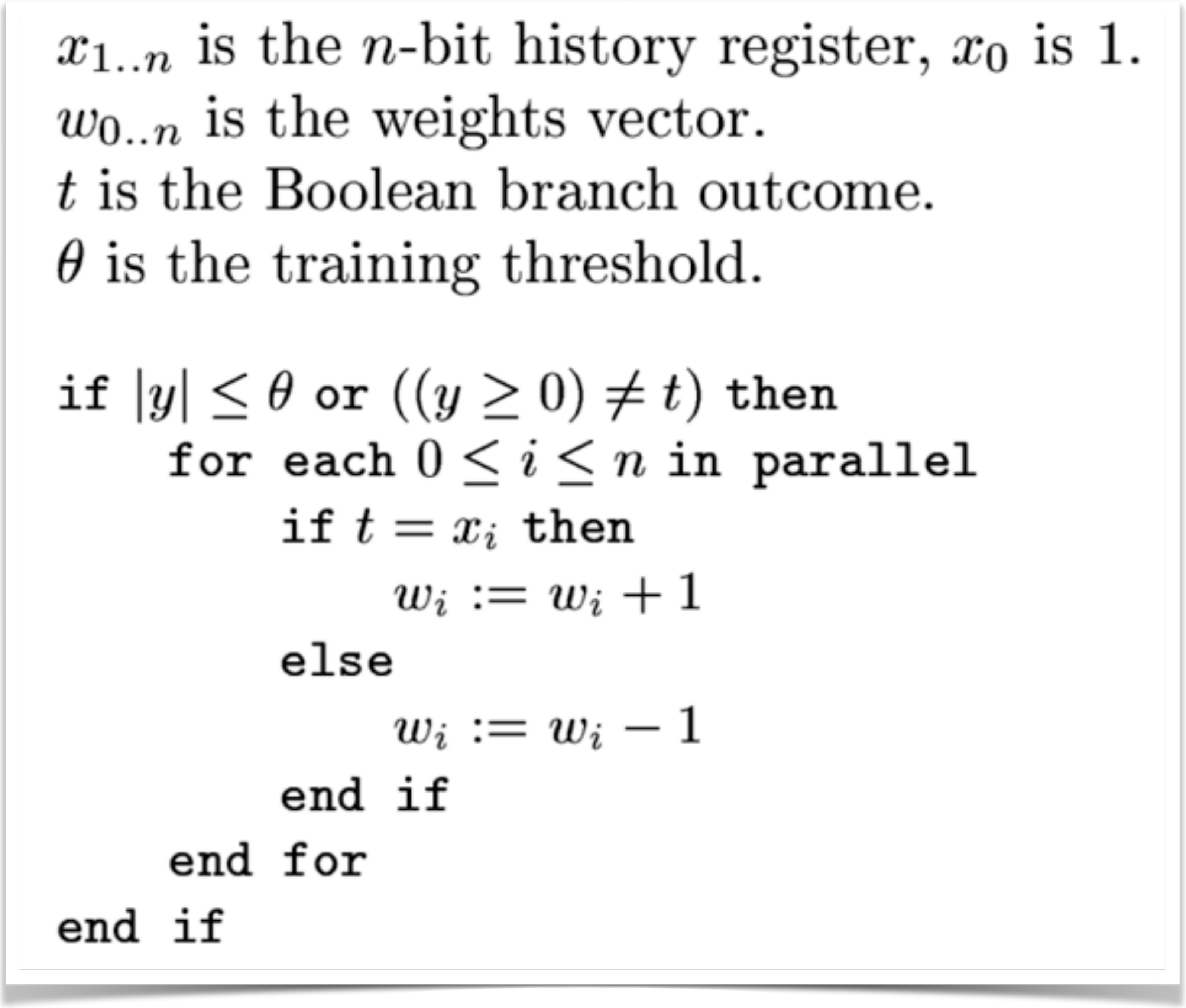
$$y = w_0 + \sum_{i=1}^{n} x_i w_i$$

Predictor Organization



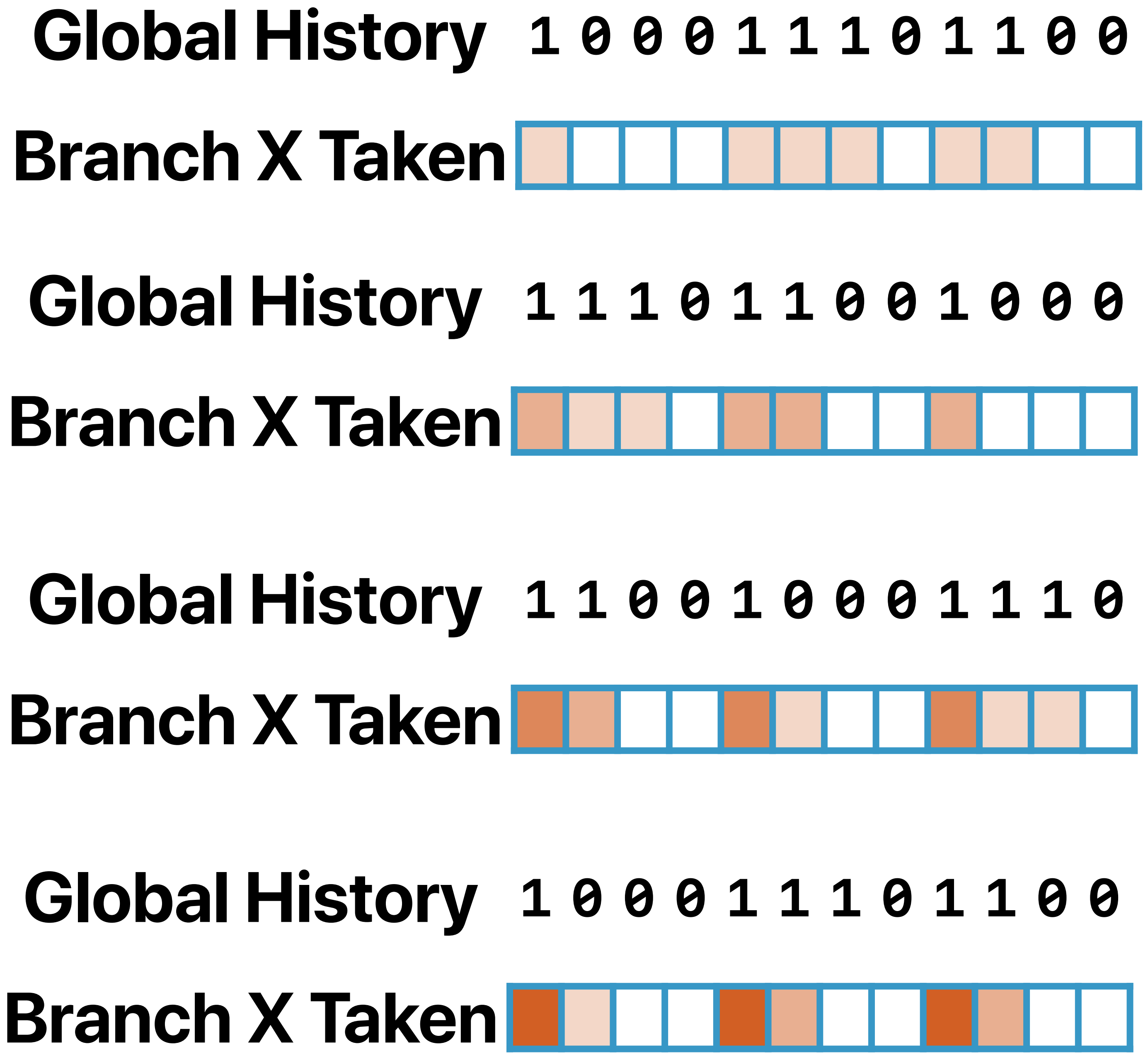
branch PC	target PC
0x400048	0x400032
0x400080	0x400068
0x401080	0x401100
0x4000F8	0x400100

Training Algorithm



$x_{1..n}$ is the n-bit history register, x_0 is 1.
$w_{0..n}$ is the weights vector.
t is the Boolean branch outcome.
θ is the training threshold.

```
if $|y| \le \theta$ or $((y \ge 0) \ne t)$ then
    for each $0 \le i \le n$ in parallel
        if $t = x_i$ then
            $w_i := w_i + 1$
        else
            $w_i := w_i - 1$
        end if
    end for
end if
```



Global History 1 0 0 0 1 1 1 0 1 1 0 0

Branch X Taken

Global History 1 1 1 0 1 1 0 0 1 0 0 0

Branch X Taken

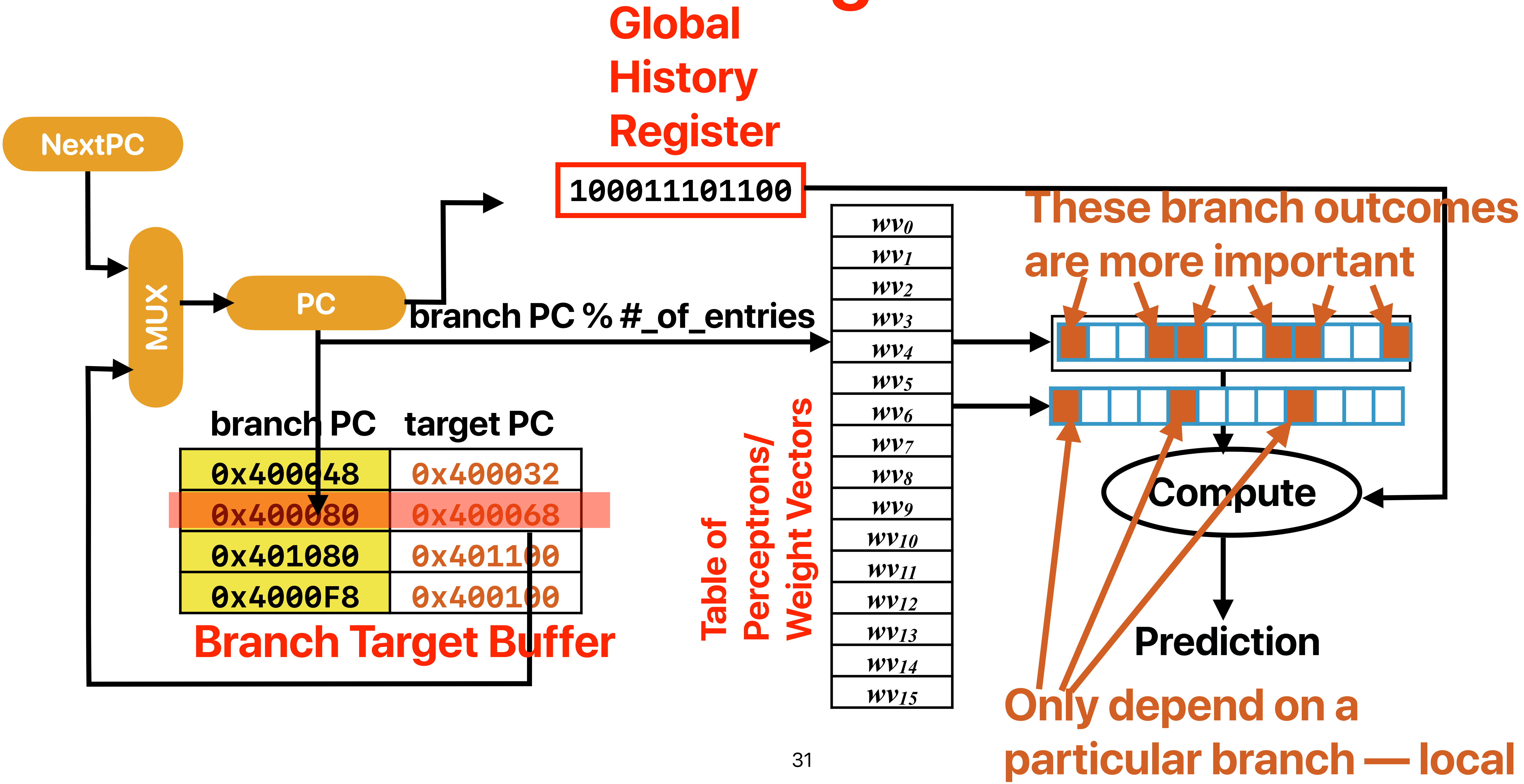
Global History 1 1 0 0 1 0 0 0 1 1 1 0

Branch X Taken

Global History 1 0 0 0 1 1 1 0 1 1 0 0

Branch X Taken

Predictor Organization



branch PC	target PC
0x400048	0x400032
0x400080	0x400068
0x401080	0x401100
0x4000F8	0x400100

Branch Target Buffer

These branch outcomes are more important

Only depend on a particular branch — local

Branch predictors in processors

- The Intel Pentium MMX, Pentium II, and Pentium III have local branch predictors with a local 4-bit history and a local pattern history table with 16 entries for each conditional jump.
- Global branch prediction is used in Intel Pentium M, Core, Core 2, and Silvermont-based Atom processors.
- Tournament predictor is used in DEC Alpha, AMD Athlon processors
- The AMD Ryzen multi-core processor's Infinity Fabric and the Samsung Exynos processor include a perceptron based neural branch predictor.

Demo revisited

```
SELECT count(*) FROM TABLE WHERE val >= A;
```

```
if(option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```

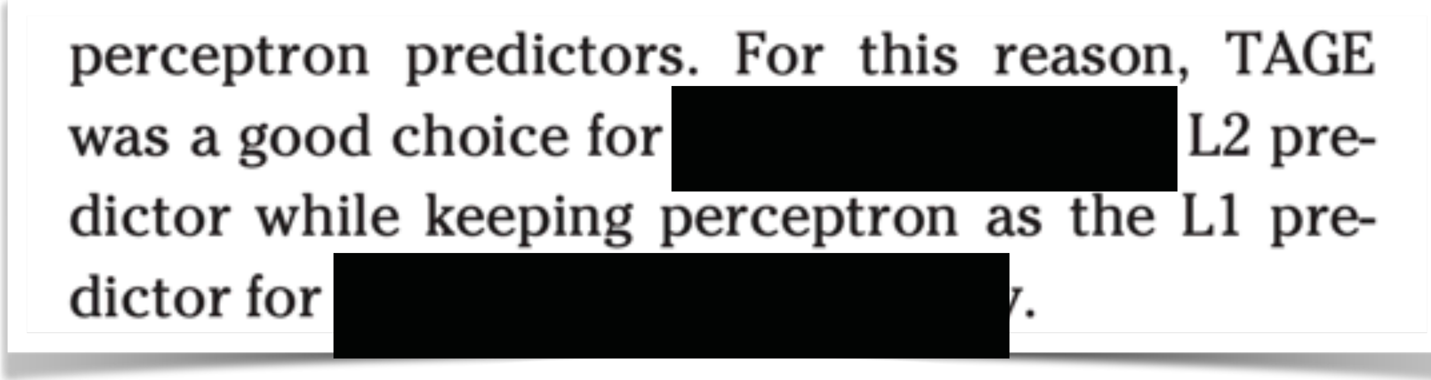
option = 1 is faster!!!, this is just a one-time cost and you don't do this after the data is loaded

	Without sorting	With sorting
The prediction accuracy of X before threshold	60%-70%	100%
The prediction accuracy of X after threshold	60%-70%	100%

https://www.pollev.com/hungweitseng close in 1:30

Design decisions in real practice

- AMD Zen 2 (RyZen 3000 series processors) adopts a design with first level predictor using perceptron and using TAGE for the 2nd level. What such a design decision implies about the characteristics of TAGE and Perceptron?
  ① Perceptron takes longer to train than TAGE
  ② Perceptron takes longer to predict than TAGE
  ③ Perceptron is more accurate than TAGE
  ④ Perceptron's performance improves less given more area
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4



perceptron predictors. For this reason, TAGE was a good choice for [redacted] L2 predictor while keeping perceptron as the L1 predictor for [redacted].

TAGE vs Perceptron

Design decisions in real practice

- AMD Zen 2 (RyZen 3000 series processors) adopts a design with first level predictor using perceptron and using TAGE for the 2nd level. What such a design decision implies about the characteristics of TAGE and Perceptron?
  - ① Perceptron takes longer to train than TAGE
  - ② Perceptron takes longer to predict than TAGE
  - ③ Perceptron is more accurate than TAGE
  - ④ Perceptron's performance improves less given more area

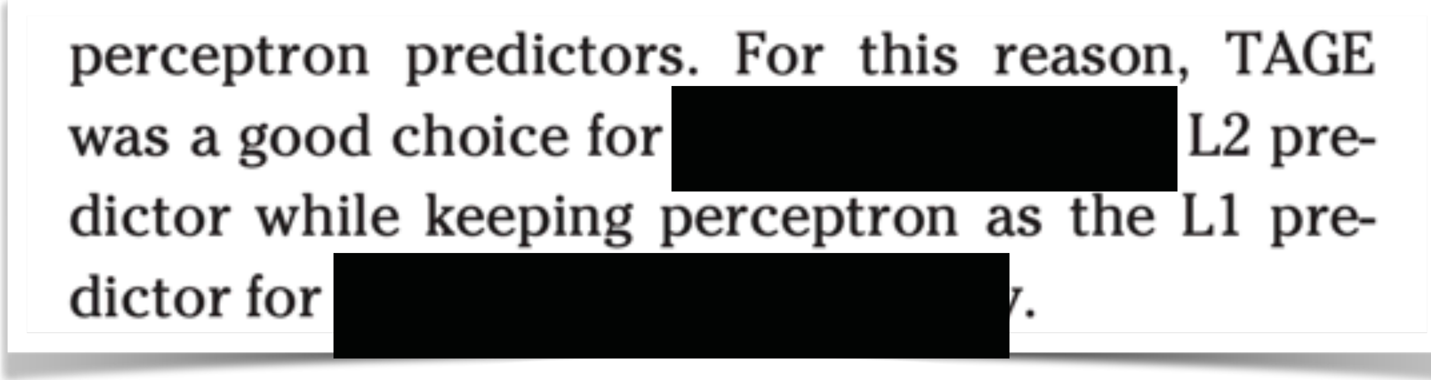
  A. 0

  B. 1

  C. 2

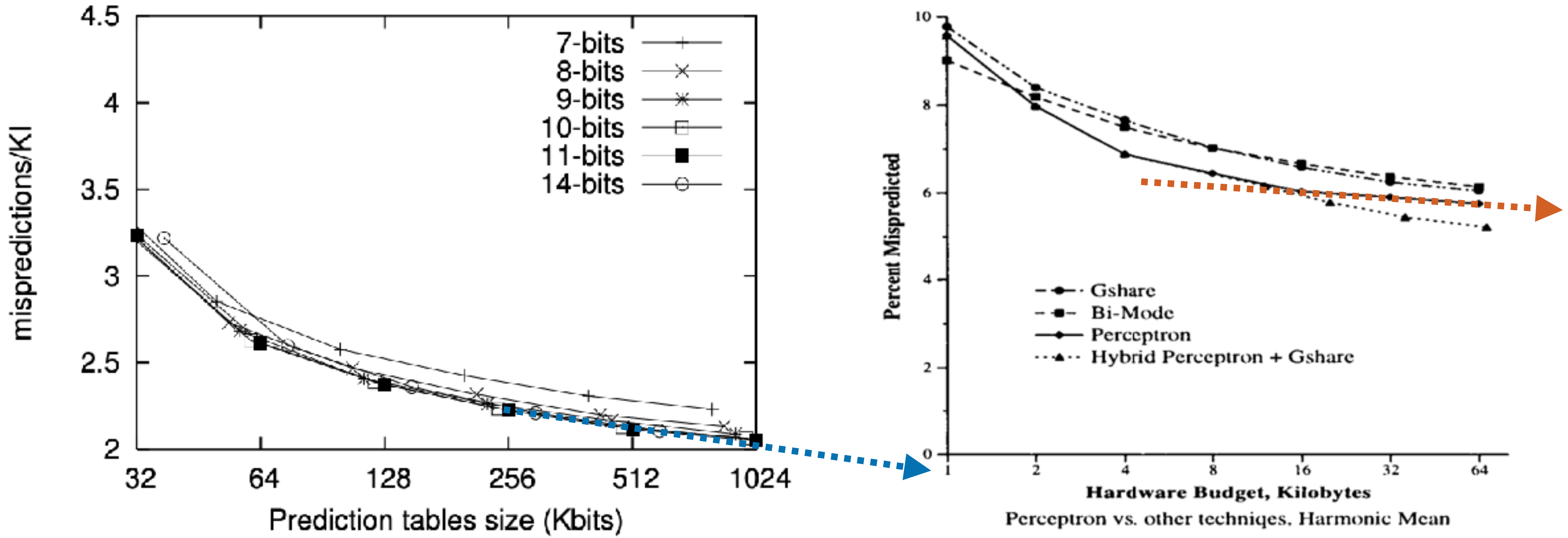
  D. 3

  E. 4



perceptron predictors. For this reason, TAGE was a good choice for [redacted] L2 predictor while keeping perceptron as the L1 predictor for [redacted].

Area efficiency between TAGE and Perceptron



TAGE

Perceptron

How good is prediction using perceptrons?

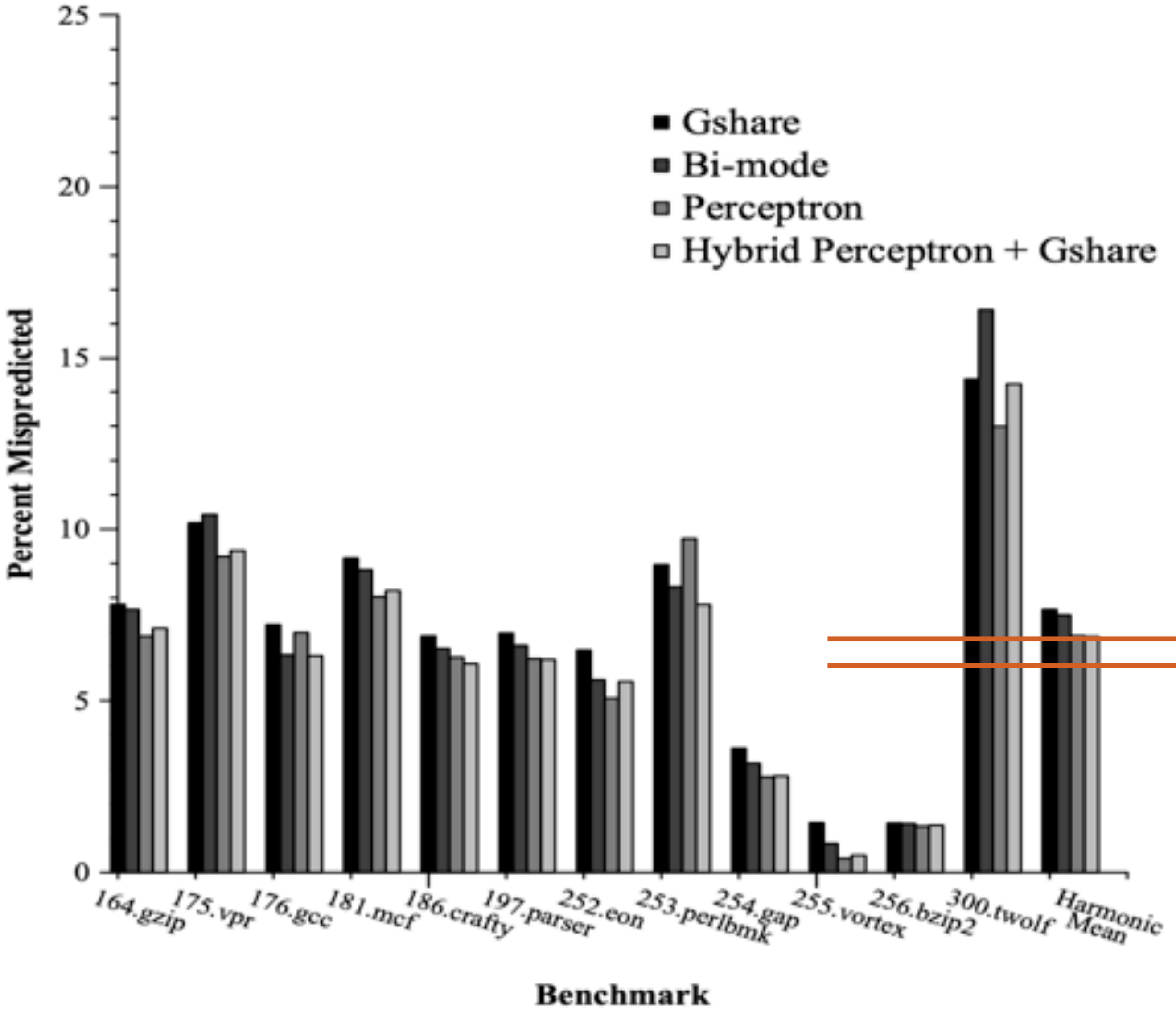


Figure 4: Misprediction Rates at a 4K budget. The perceptron predictor has a lower misprediction rate than *gshare* for all benchmarks except for `186.crafty` and `197.parser`.

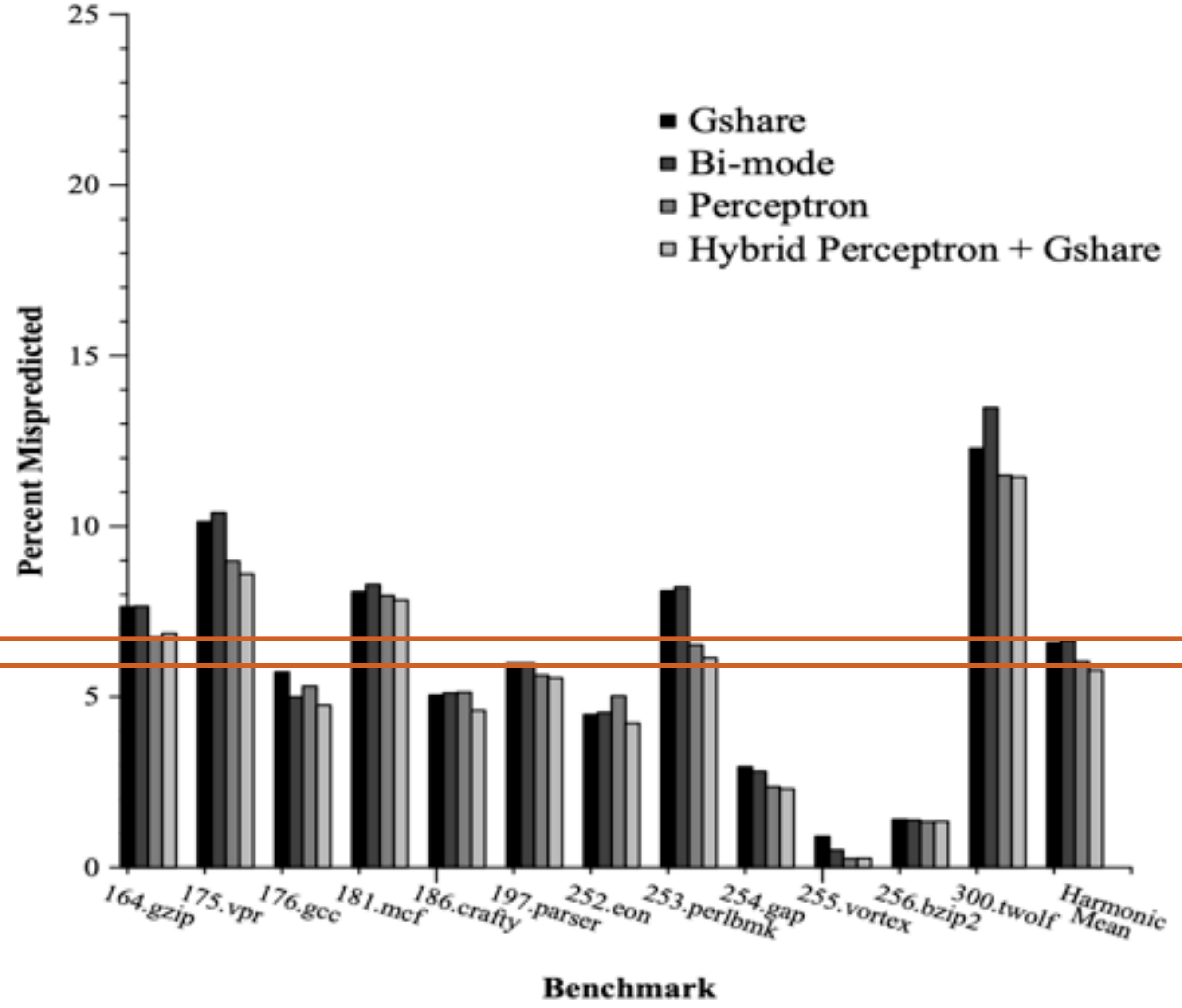
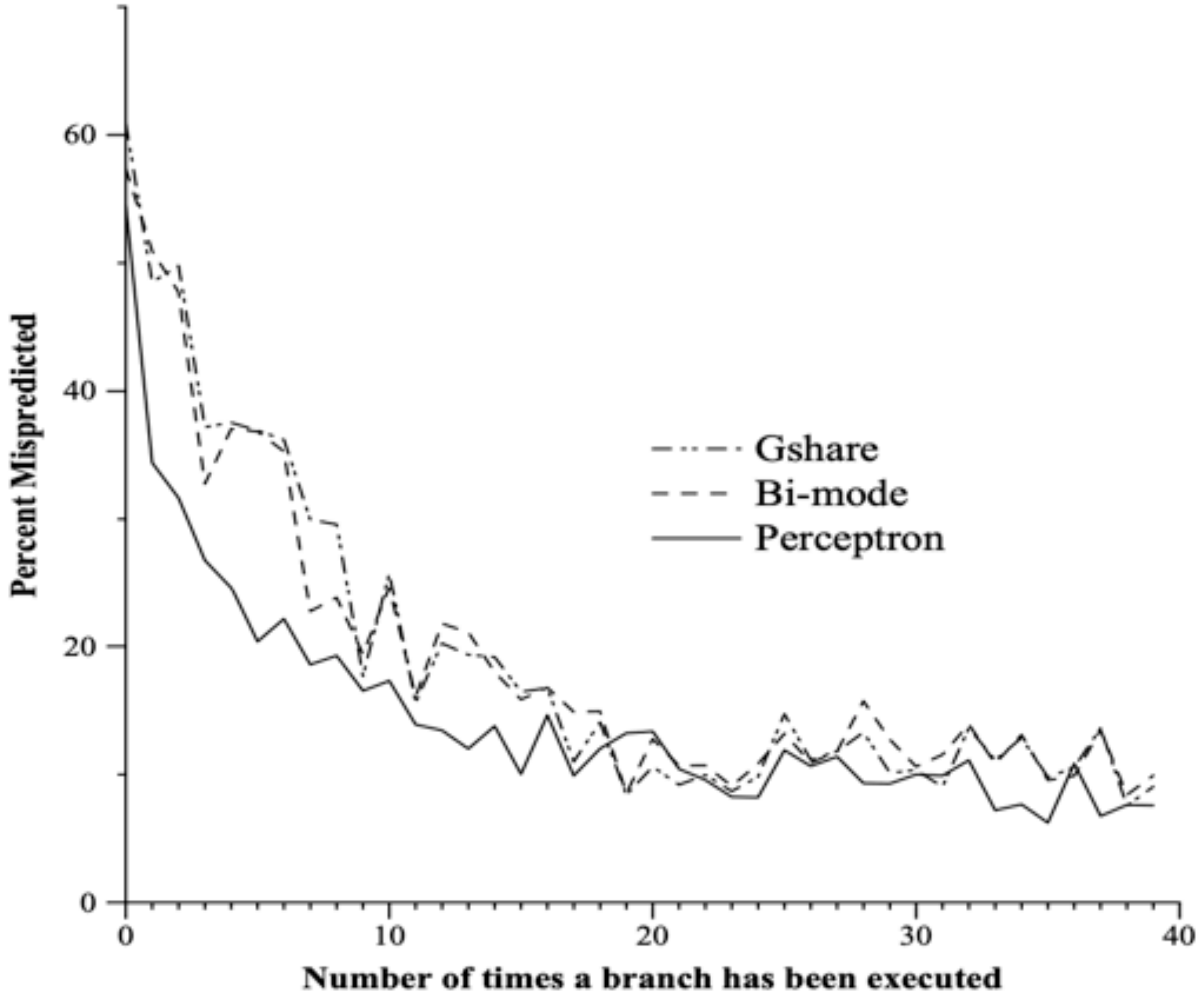
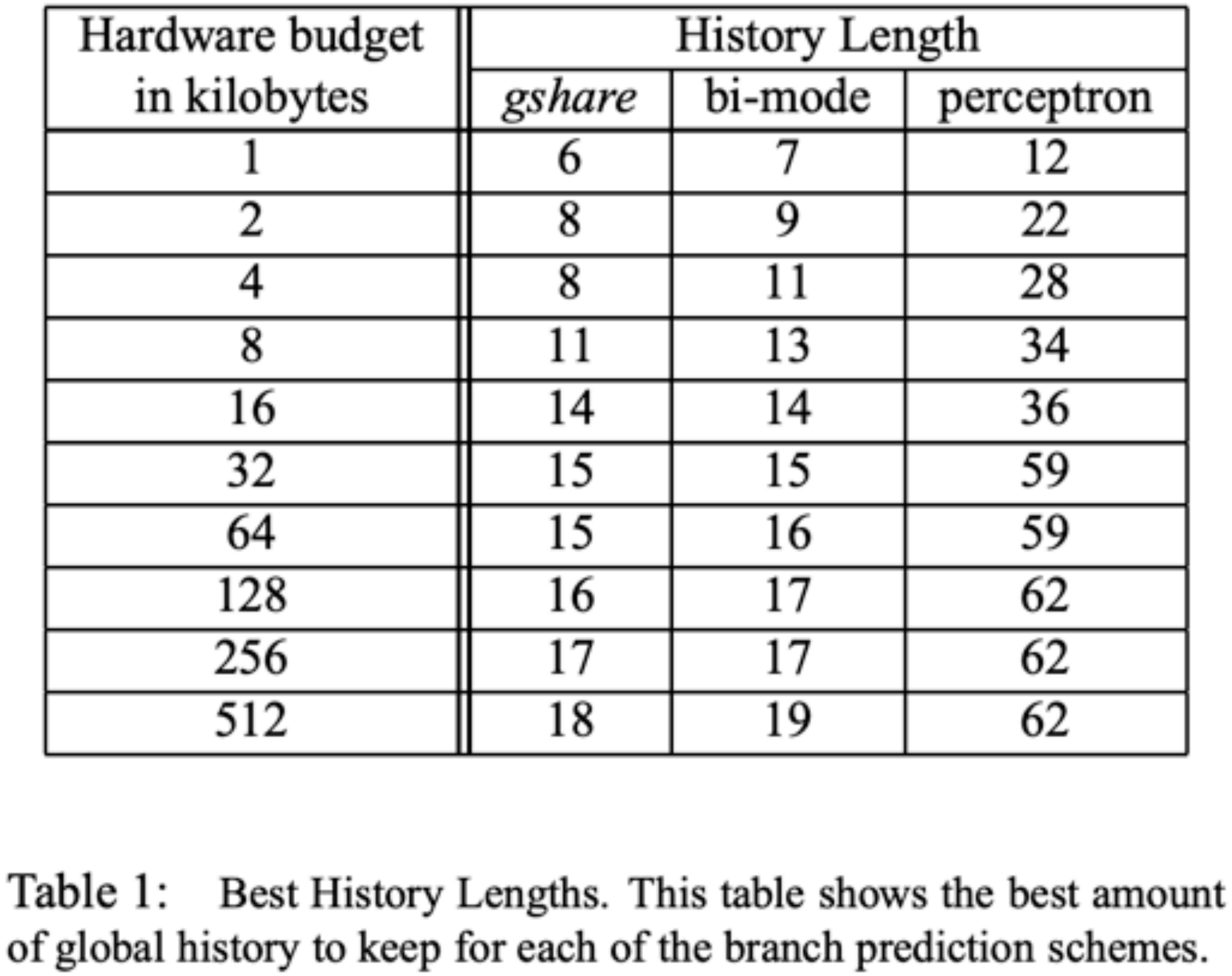


Figure 5: Misprediction Rates at a 16K budget. Gshare outperforms the perceptron predictor only on `186.crafty`. The hybrid predictor is consistently better than the PHT schemes.

History/training for perceptrons

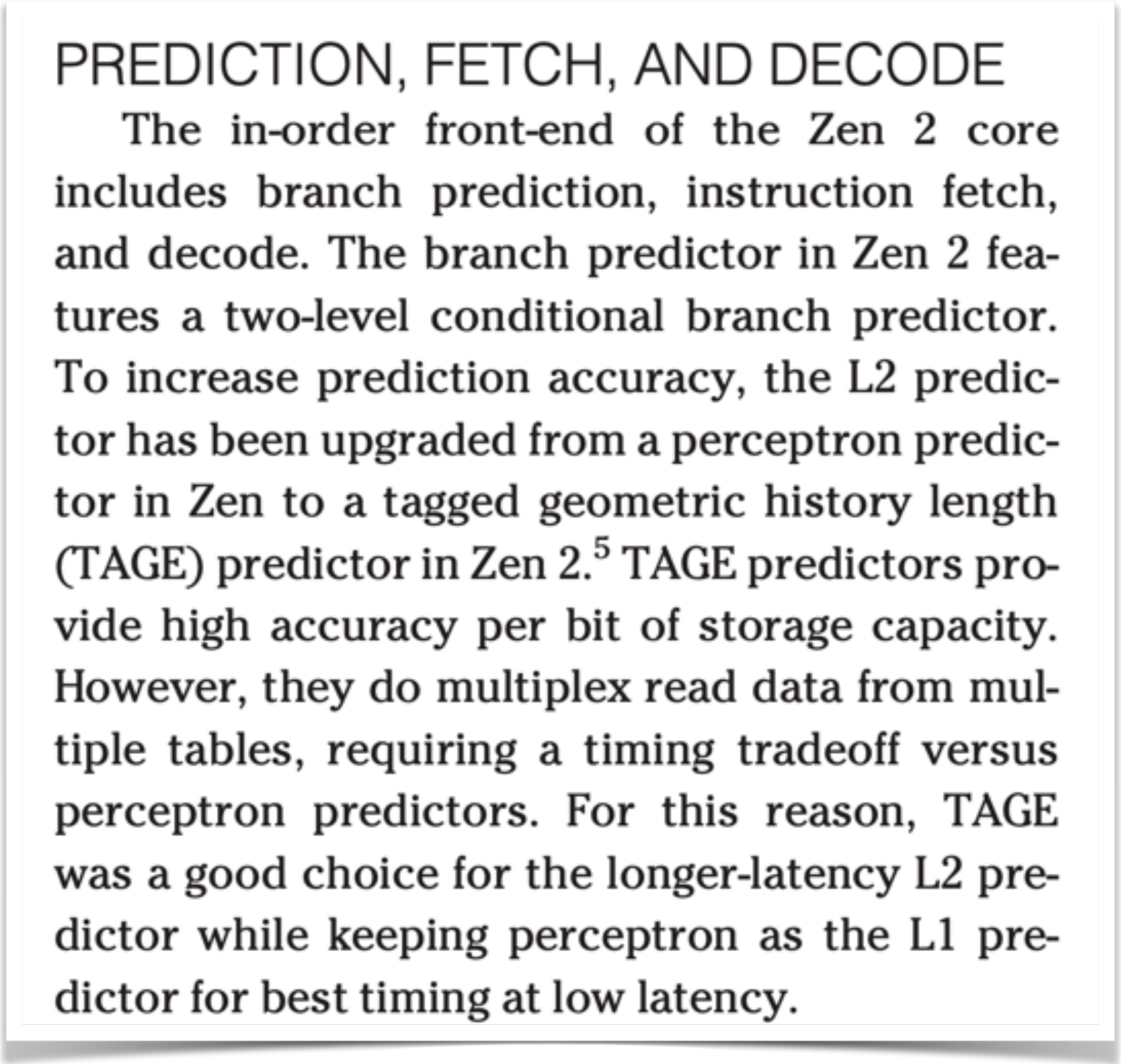




Hardware budget in kilobytes	History Length		
	gshare	bi-mode	perceptron
1	6	7	12
2	8	9	22
4	8	11	28
8	11	13	34
16	14	14	36
32	15	15	59
64	15	16	59
128	16	17	62
256	17	17	62
512	18	19	62

Table 1: Best History Lengths. This table shows the best amount of global history to keep for each of the branch prediction schemes.

AMD Zen 2's design experience



PREDICTION, FETCH, AND DECODE

The in-order front-end of the Zen 2 core includes branch prediction, instruction fetch, and decode. The branch predictor in Zen 2 features a two-level conditional branch predictor. To increase prediction accuracy, the L2 predictor has been upgraded from a perceptron predictor in Zen to a tagged geometric history length (TAGE) predictor in Zen 2.[5] TAGE predictors provide high accuracy per bit of storage capacity. However, they do multiplex read data from multiple tables, requiring a timing tradeoff versus perceptron predictors. For this reason, TAGE was a good choice for the longer-latency L2 predictor while keeping perceptron as the L1 predictor for best timing at low latency.

D. Suggs D. Bouvier M. Subramony and K. Lepak "Zen 2" Hot Chips vol. 31 2019.

Design decisions in real practice

- AMD Zen 2 (RyZen 3000 series processors) adopts a design with first level predictor using perceptron and using TAGE for the 2nd level. What such a design decision implies about the characteristics of TAGE and Perceptron?

① Perceptron takes longer to train than TAGE

no — based on the paper

② Perceptron takes longer to predict than TAGE

short — otherwise won't be in L1

③ Perceptron is more accurate than TAGE

less accurate — otherwise won't need an L2

④ Perceptron's performance improves less given more area

no — based on the paper

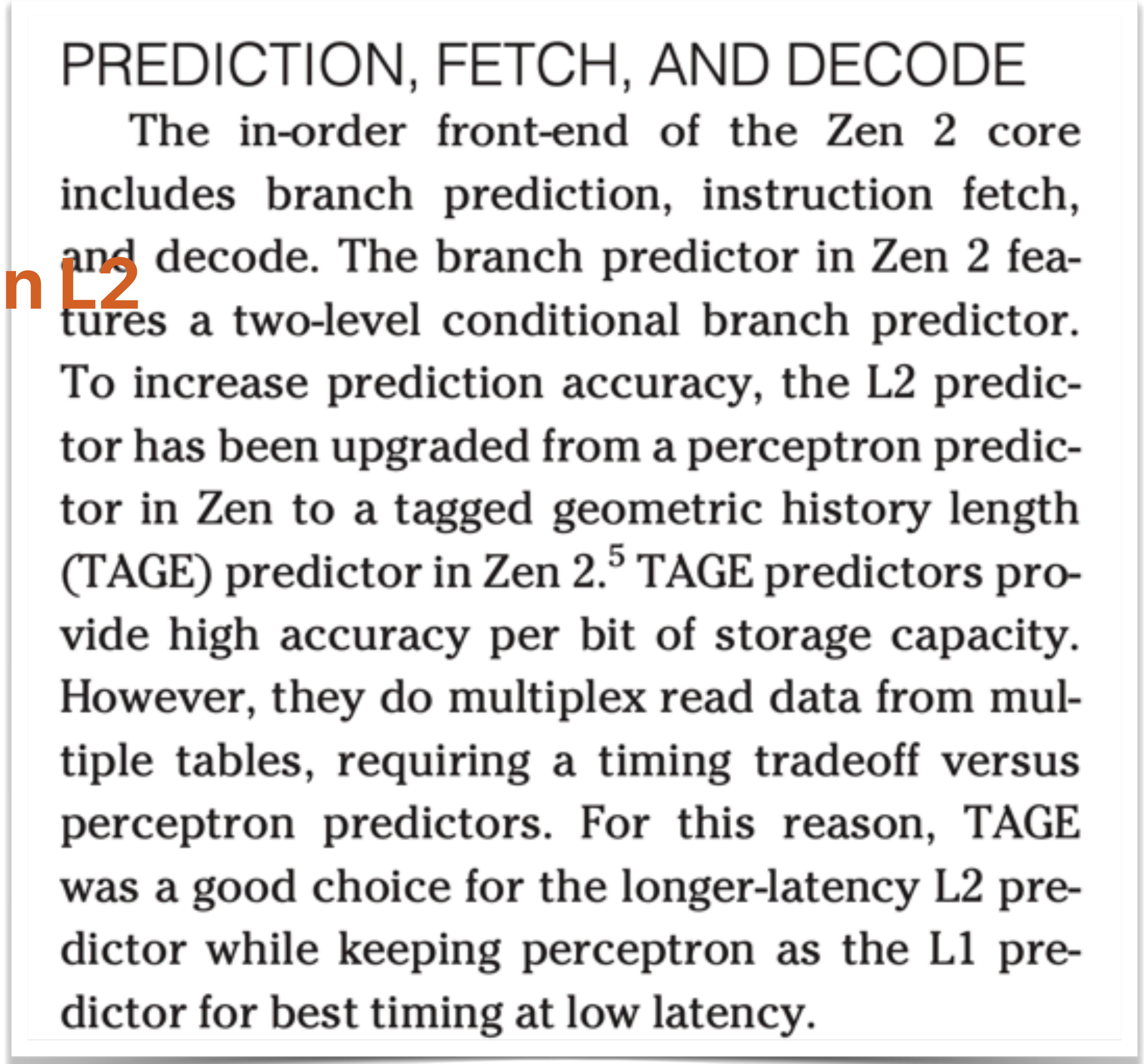
[✓ A. 0]

B. 1

C. 2

D. 3

E. 4



PREDICTION, FETCH, AND DECODE

The in-order front-end of the Zen 2 core includes branch prediction, instruction fetch, and decode. The branch predictor in Zen 2 features a two-level conditional branch predictor. To increase prediction accuracy, the L2 predictor has been upgraded from a perceptron predictor in Zen to a tagged geometric history length (TAGE) predictor in Zen 2.[5] TAGE predictors provide high accuracy per bit of storage capacity. However, they do multiplex read data from multiple tables, requiring a timing tradeoff versus perceptron predictors. For this reason, TAGE was a good choice for the longer-latency L2 predictor while keeping perceptron as the L1 predictor for best timing at low latency.

D. Suggs D. Bouvier M. Subramony and K. Lepak "Zen 2" Hot Chips vol. 31 2019.

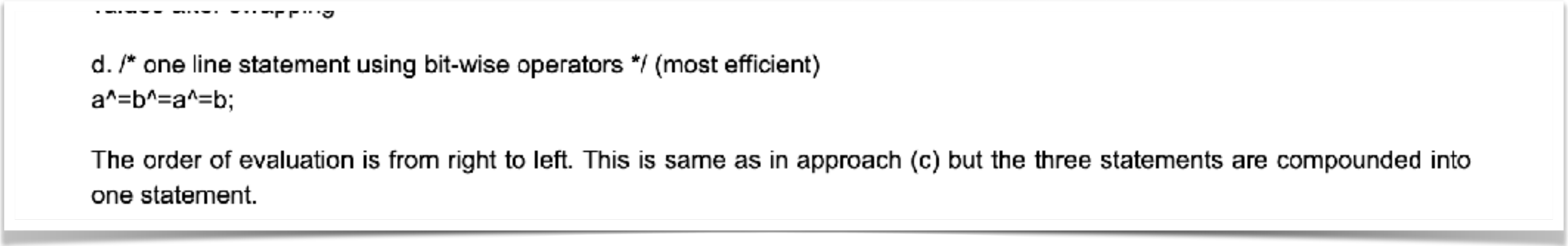
Branch predictors in processors

- The Intel Pentium MMX, Pentium II, and Pentium III have local branch predictors with a local 4-bit history and a local pattern history table with 16 entries for each conditional jump.
- Global branch prediction is used in Intel Pentium M, Core, Core 2, and Silvermont-based Atom processors.
- Tournament predictor is used in DEC Alpha, AMD Athlon processors
- The AMD Ryzen multi-core processor's Infinity Fabric and the Samsung Exynos processor include a perceptron based neural branch predictor.

Takeaways: branch predictions

- The cost of not to predict a branch is to stall until the data dependency is resolved — 34 cycles on modern intel processors and 23 on AMD processors
- Branch predictions allow the processor to at least make some progress and hide the stalls if we guessed correctly!
- Dynamic branch prediction — predict based on prior history
    - Local predictor — make predictions based on the state of each branch instruction
    - Global predictor — make predictions based on the state from all branches
    - Both are not perfect — hybrid predictors
        - Tournament
        - Perceptron
    - All modern processors have pretty accurate branch predictors — if the code itself is predictable

Recap: But A is faster!

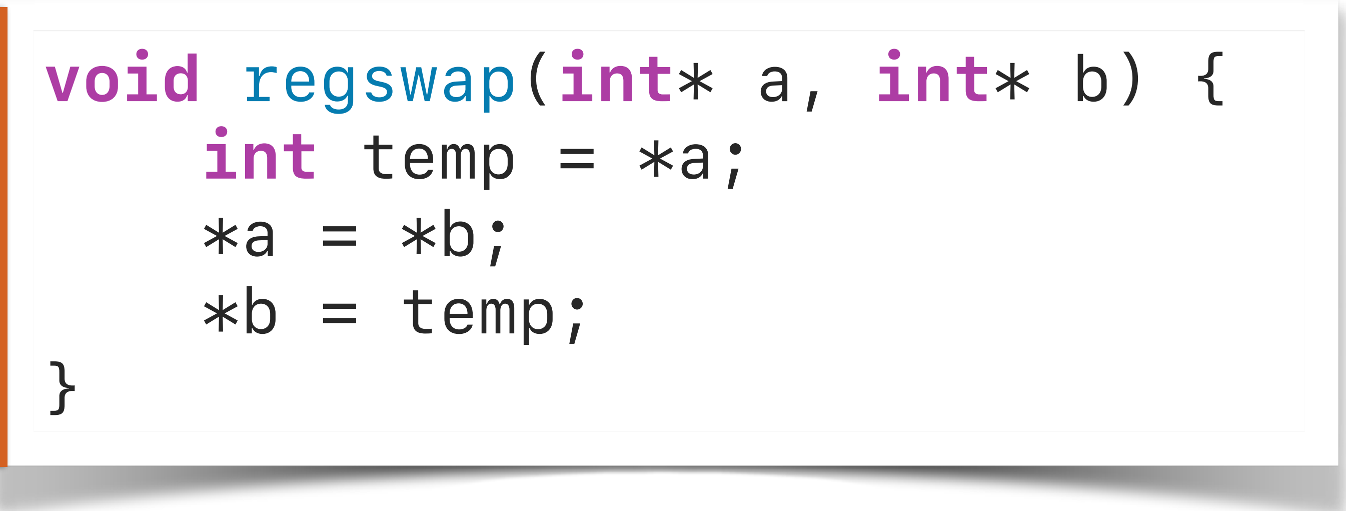


values after swapping

d. /* one line statement using bit-wise operators */ (most efficient)
a^=b^=a^=b;

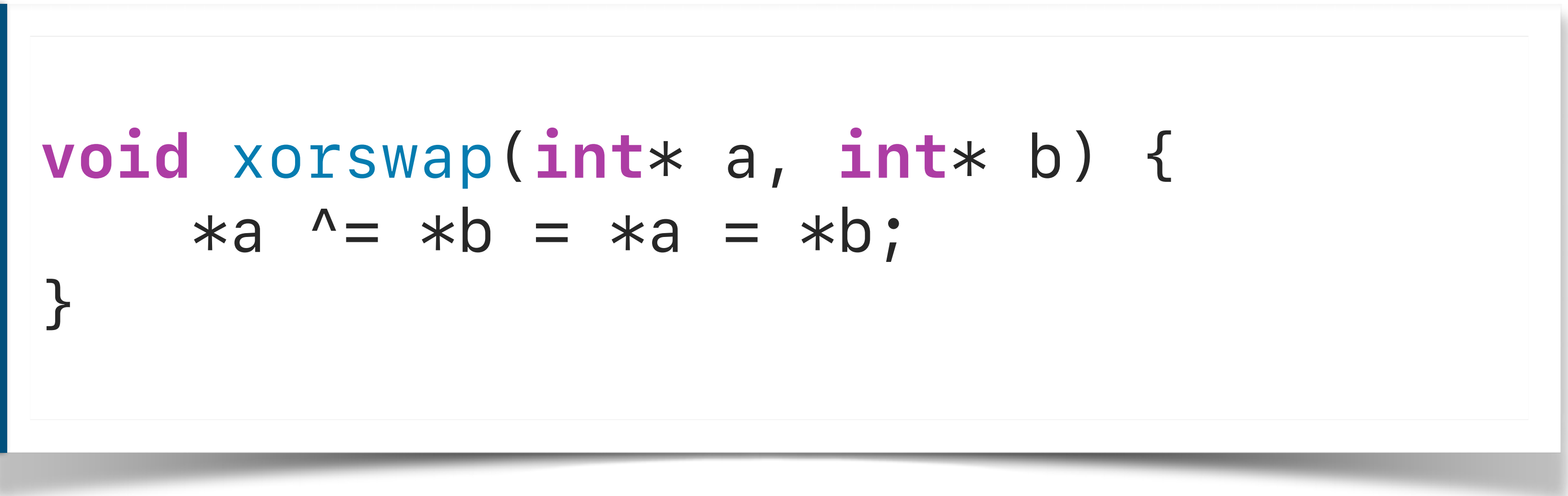
The order of evaluation is from right to left. This is same as in approach (c) but the three statements are compounded into one statement.

A



```
void regswap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

B



```
void xorswap(int* a, int* b) {
    *a ^= *b = *a = *b;
}
```

Data hazards

Data hazards

- An instruction currently in the pipeline cannot receive the “logically” correct value for execution
- Data dependencies
  - The output of an instruction is the input of a later instruction
  - **May sometimes** result in data hazard if the later instruction that consumes the result is still in the pipeline

https://www.pollev.com/hungweitseng close in 1:30

How many data dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?

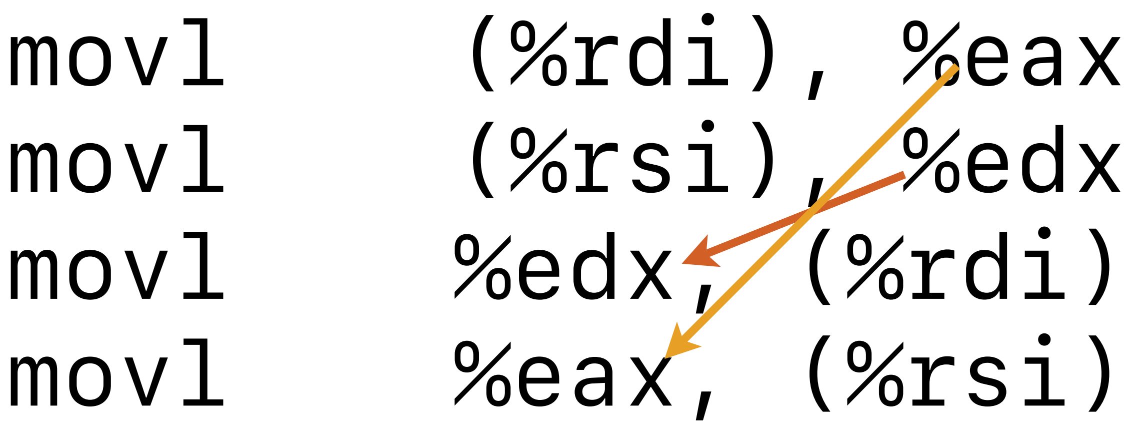
```
movl     (%rdi), %eax
movl     (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```

```
int temp = *a;
*a = *b;
*b = temp;
```

A. 1
B. 2
C. 3
D. 4
E. 5

How many dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?



```
movl     (%rdi), %eax
movl     (%rsi), %edx
movl     %edx, (%rdi)
movl     %eax, (%rsi)
```

```
int temp = *a;
*a = *b;
*b = temp;
```

A. 1

[✓ B. 2]

C. 3

D. 4

E. 5

https://www.pollev.com/hungweitseng close in 1:30

How many data dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?

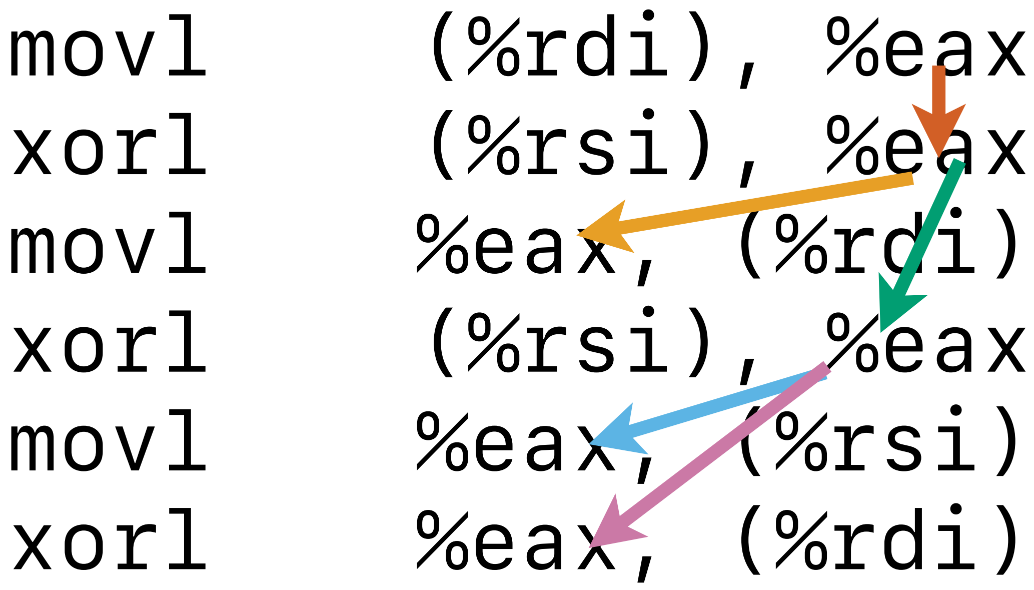
```
movl    (%rdi), %eax
xorl    (%rsi), %eax
movl    %eax, (%rdi)
xorl    (%rsi), %eax
movl    %eax, (%rsi)
xorl    %eax, (%rdi)
```

```
*a ^= *b;
*b ^= *a;
*a ^= *b;
```

A. 1
B. 2
C. 3
D. 4
E. 5

How many dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?



```
movl    (%rdi), %eax
xorl    (%rsi), %eax
movl    %eax, (%rdi)
xorl    (%rsi), %eax
movl    %eax, (%rsi)
xorl    %eax, (%rdi)
```

```
*a ^= *b;
*b ^= *a;
*a ^= *b;
```

A. 1
B. 2
C. 3
D. 4
[✓ E. 5]

Data hazards

```
① movl    (%rdi), %eax
② movl    (%rsi), %edx
③ movl    %edx, (%rdi)
④ movl    %eax, (%rsi)
```

	IF	ID	ALU/BR/AG	M1	M2	M3	M4/XORL	WB/Retire
1	(1)							
2	(2)	(1)						
3	(3)	(2)	(1)					
4	(4)	(3)	(2)	(1)				
5		(4)	(3)	(2)	(1)			
6			(4)	(3)	(2)	(1)		
7				(4)	(3)	(2)	(1)	
8					(4)	(3)	(2)	
9						(4)	(3)	
10							(4)	
11								
12								
13								
14								

%edx does not have our desired value

%eax does not have our desired value

Solution 1: Let’s try “stall” again

- Whenever the input is not ready when the consumer is decoding, just stall — the consumer stays at ID.

https://www.pollev.com/hungweitseng close in 1:30

Data hazards?

- How many cycles do we have to stall in the following x86 instructions to get the expected output if a memory operation (assume 100% cache hit rate) takes 5 cycles?

```
① movl    (%rdi), %eax
② movl    (%rsi), %edx
③ movl    %edx, (%rdi)
④ movl    %eax, (%rsi)
```

A. 1
B. 2
C. 3
D. 4
E. 5

Data hazards?

- How many cycles do we have to stall in the following x86 instructions to get the expected output if a memory operation (assume 100% cache hit rate) takes 5 cycles?

```
① movl    (%rdi), %eax
② movl    (%rsi), %edx
③ movl    %edx, (%rdi)
④ movl    %eax, (%rsi)
```

A. 1

B. 2

C. 3

D. 4

E. 5

	IF	ID	ALU/BR/AG	M1	M2	M3	M4/XORL	WB/Retire
1	(1)							
2	(2)	(1)						
3	(3)	(2)	(1)					
4	(4)	(3)	(2)	(1)				
5	(4)	(3)		(2)	(1)			
6	(4)	(3)			(2)	(1)		
7	(4)	(3)				(2)	(1)	
8	(4)	(3)					(2)	(1)
9	(4)	(3)						(2)
10		(4)	(3)					
11			(4)	(3)				
12				(4)	(3)			
13					(4)	(3)		
14						(4)		(3)
15								(4)

we have the value for %edx already!

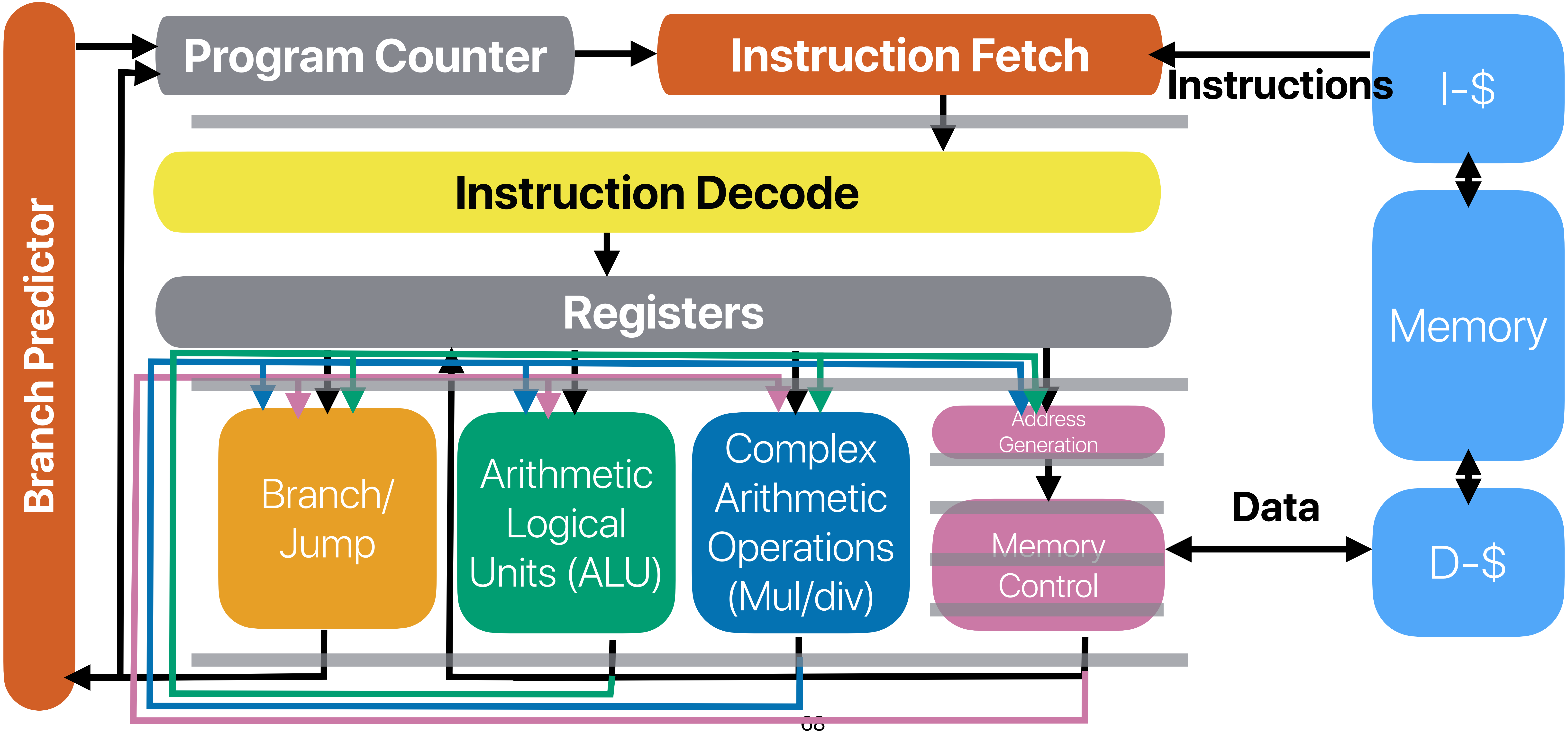
Why another cycle?

5 cycles stalls

Solution 2: Data forwarding

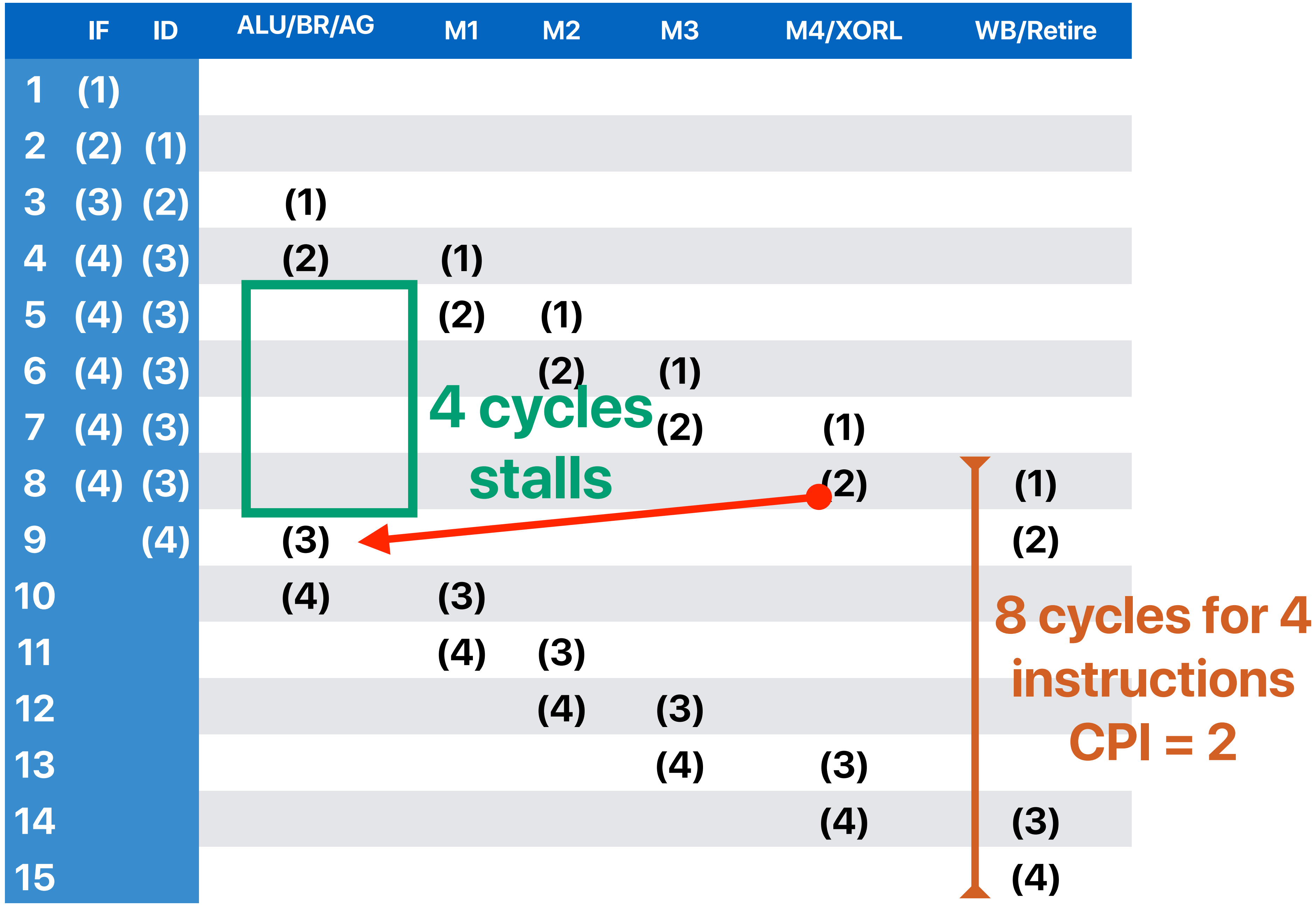
- Add logics/wires to forward the desired values to the demanding instructions

Data "forwarding"



The effect of data forwarding

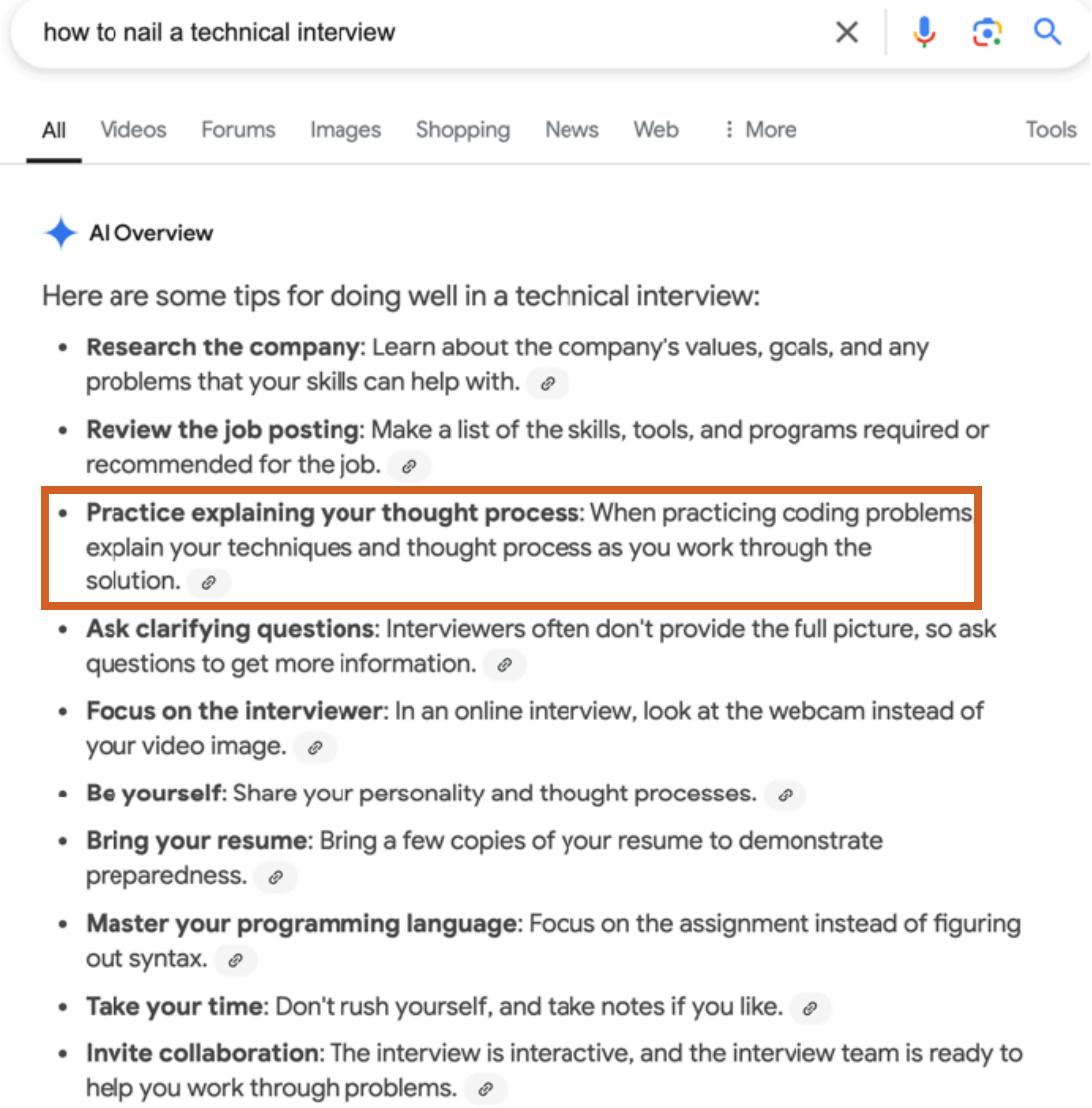
```
① movl    (%rdi), %eax
② movl    (%rsi), %edx
③ movl    %edx, (%rdi)
④ movl    %eax, (%rsi)
```



	IF	ID	ALU/BR/AG	M1	M2	M3	M4/XORL	WB/Retire
1	(1)							
2	(2)	(1)						
3	(3)	(2)	(1)					
4	(4)	(3)	(2)	(1)				
5	(4)	(3)		(2)	(1)			
6	(4)	(3)			(2)	(1)		
7	(4)	(3)				(2)	(1)	
8	(4)	(3)					(2)	(1)
9		(4)	(3)					(2)
10			(4)	(3)				
11				(4)	(3)			
12					(4)	(3)		
13						(4)	(3)	
14							(4)	(3)
15								(4)

4 cycles stalls

8 cycles for 4 instructions CPI = 2



Here are some tips for doing well in a technical interview:

- **Research the company**: Learn about the company's values, goals, and any problems that your skills can help with.
- **Review the job posting**: Make a list of the skills, tools, and programs required or recommended for the job.
- **Practice explaining your thought process**: When practicing coding problems explain your techniques and thought process as you work through the solution.
- **Ask clarifying questions**: Interviewers often don't provide the full picture, so ask questions to get more information.
- **Focus on the interviewer**: In an online interview, look at the webcam instead of your video image.
- **Be yourself**: Share your personality and thought processes.
- **Bring your resume**: Bring a few copies of your resume to demonstrate preparedness.
- **Master your programming language**: Focus on the assignment instead of figuring out syntax.
- **Take your time**: Don't rush yourself, and take notes if you like.
- **Invite collaboration**: The interview is interactive, and the interview team is ready to help you work through problems.

Practice in your assignments and examines!

- Why do I pick/support this solution?
  - Prove it delivers the right result
  - Justify it's better "something" compared against other "alternatives"
- What's the solution?
  - Describe the high-level architecture/idea of your solution
- How does it work out in the problem?

Announcements

- **Assignment 4** due next **Thursday**
- **Reading Quiz 8** due **Tuesday** before the lecture

Computer
Science &
Engineering

203

つづく