Now playing — I guess I just feel like (John Mayer)

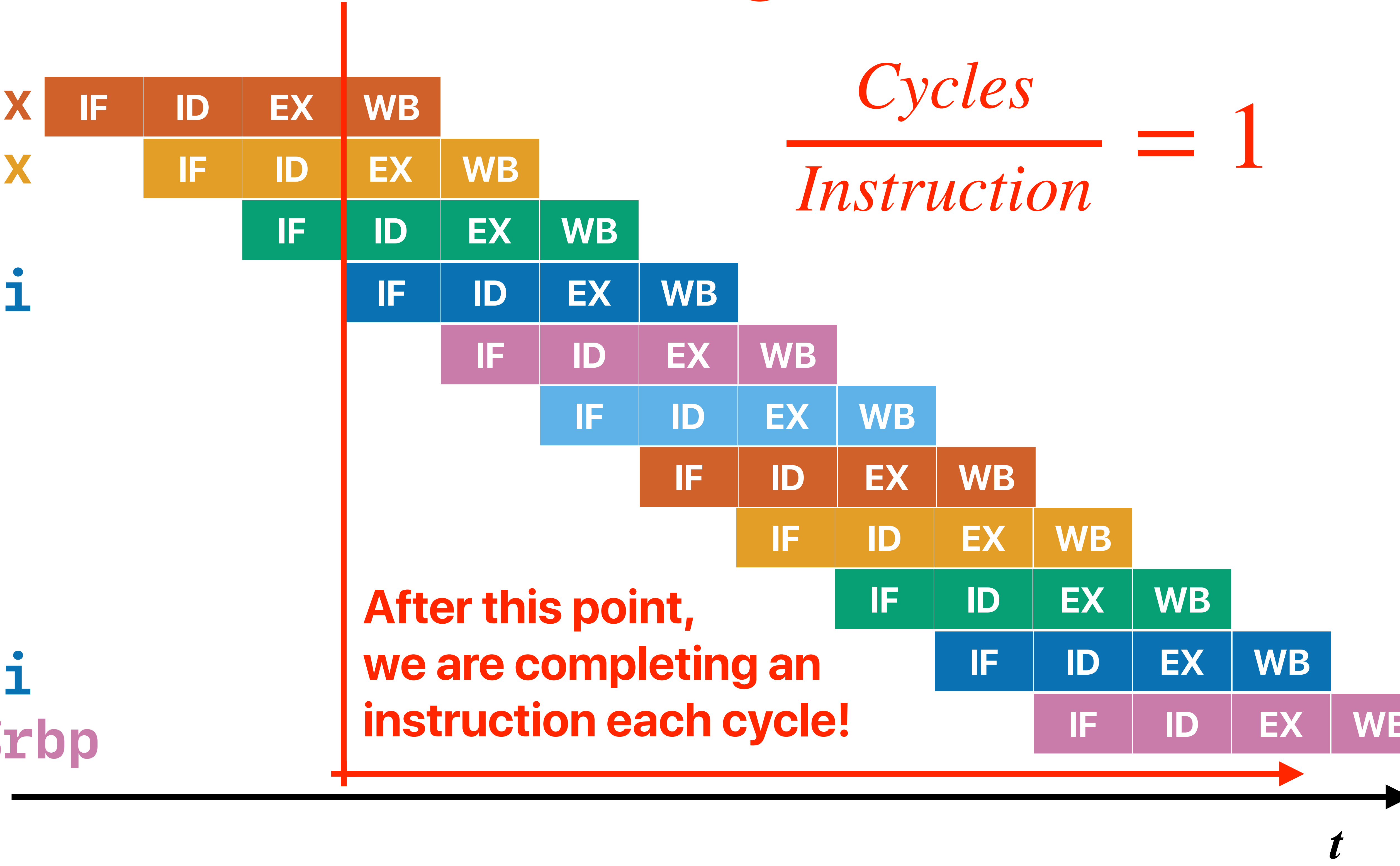
Quiz Alert!

- **Check the schedule in course webpage — due next Tuesday BEFORE the lecture!!**
- No time limitation!
- Start early! We don't accept late submissions
- 15% of your grades!

Recap: Pipelining

```
addl   %eax, %eax
addl   %rdi, %ecx
addq   $4, %r11
testl  %esi, %esi
movl   $10, %edx
pushq  %r12
pushq  %rbp
pushq  %rbx
subq   $8, %rsp
addl   %rsi, %rdi
movslq  %eax, %rbp
```

$$\frac{Cycles}{Instruction} = 1$$

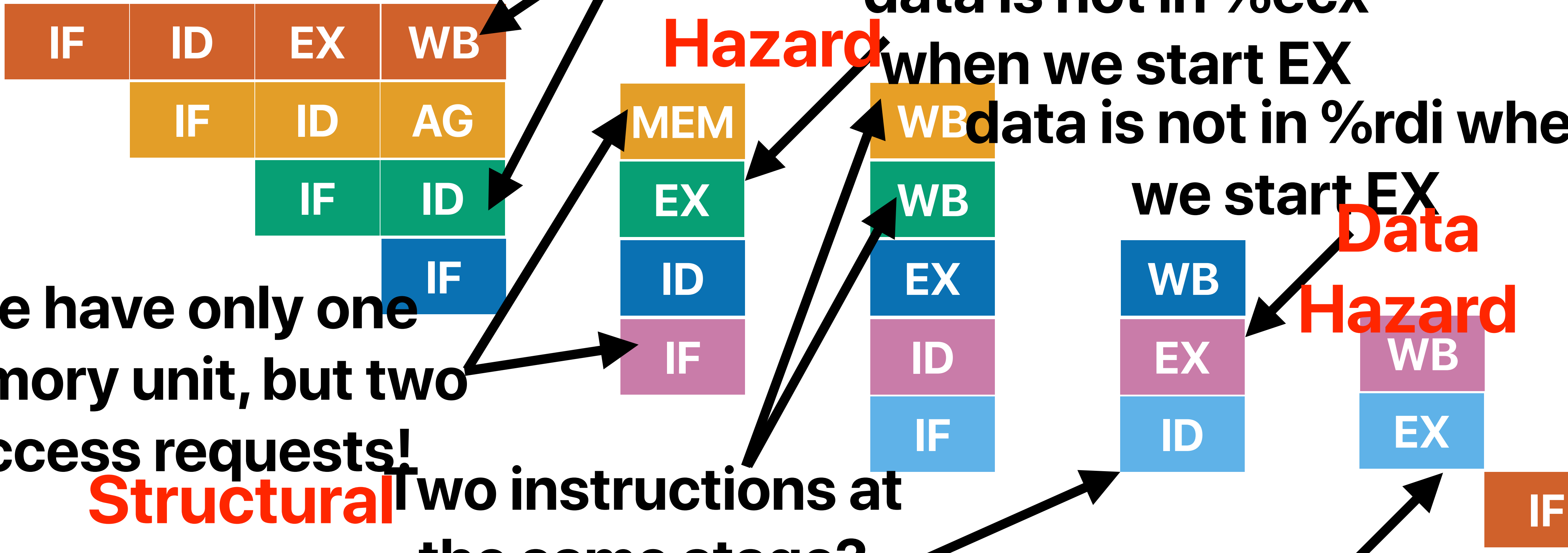


Pipeline Hazards

```
① xorl %eax, %eax
② movl (%rdi), %ecx
③ addl %ecx, %eax
④ addq $4, %rdi
⑤ cmpq %rdx, %rdi
⑥ jne  .L3
⑦ ret
```

- How many of the "hazards" are data hazards?

  A. 0
  B. 1
  $\boxed{\text{C. } 2}$
  D. 3
  E. 4



Structural Hazard

Both (1) and (3) are attempting to access %eax

Data Hazard

data is not in %ecx when we start EX

data is not in %rdi when we start EX

Data Hazard

We have only one memory unit, but two access requests!

Structural Hazard

Structural Hazard

Two instructions at the same stage?

We cannot know if we should fetch (7) or (2) before the EX is done

(6) may not have the outcome from (5)

Control Hazard

Recap: Structural Hazards

- Force later instructions to stall
- Improve the pipeline unit design to allow parallel execution
  - Write-first, read later register files
  - Split L1-Cache
  - Non-blocking, multi-banked cache/memory

3:00

When and how will you make a guess?

Outline

- Why branch prediction for control hazards
- Dynamic branch predictions
  - Local predictor — 2 bit
  - Global predictor — 2-level
  - Hybrid predictors
    - Tournament
    - Perceptron

Control Hazards

How does the code look like?

```
for (unsigned i = 0; i < size; ++i) { // taken when true
    if (data[i] < threshold)          // taken when true
        call_when_true(&a[i]);
    else
        call_when_false(&a[i]);
}
```

Branch taken simply means we are using branch target address as the next address

```
.LFB16:
    endbr64
    testl %esi, %esi
    jle   .L10
    movslq  %esi, %rsi
    pushq %r12
    leaq  (%rdi,%rsi,8), %r12
    pushq %rbp
    movslq  %edx, %rbp
    pushq %rbx
    movq  %rdi, %rbx
    jmp   .L5
    .p2align 4,,10
    .p2align 3
.L15:
    call  call_when_true@PLT
    addq  $8, %rbx
```

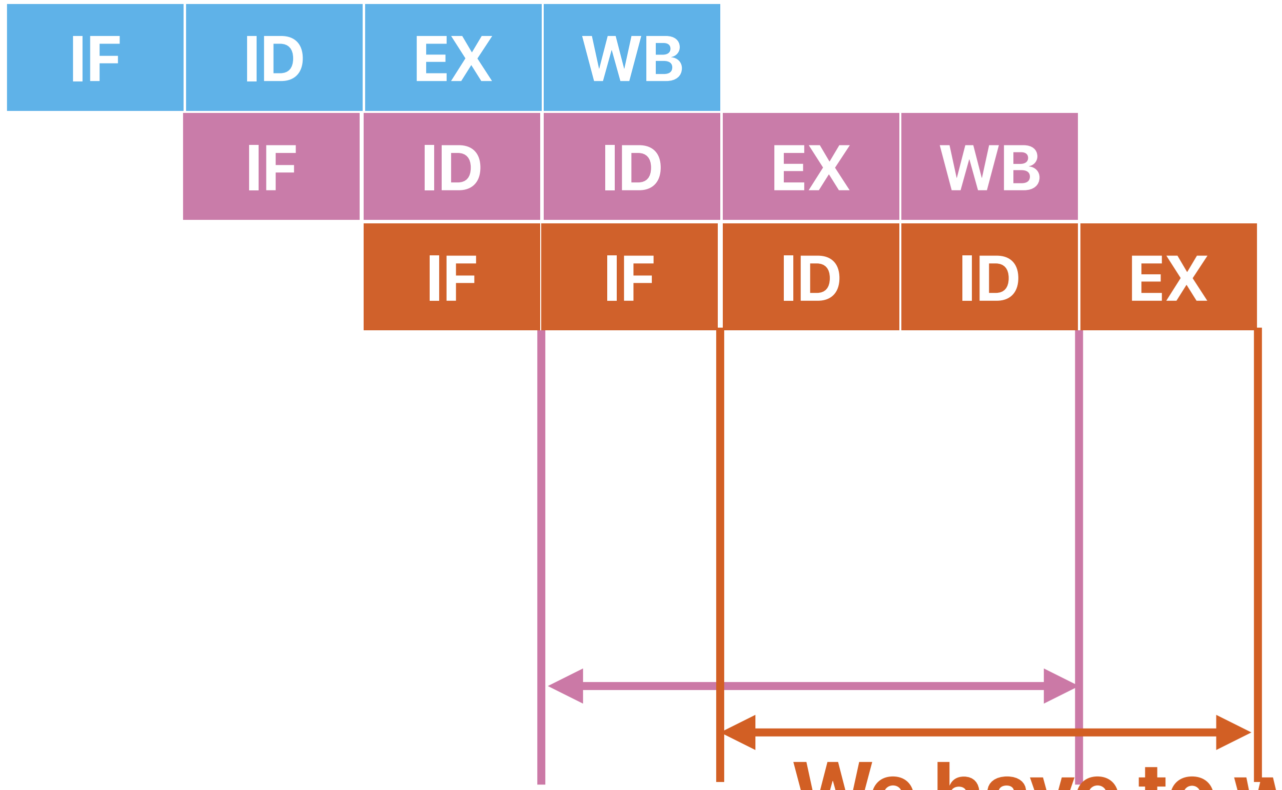
Branch taken

Branch taken

```
    cmpq  %r12, %rbx
    je .L14
.L5:
    movq  %rbx, %rdi
    cmpq  %rbp, (%rbx)
    jl .L15
    call  call_when_false@PLT
    addq  $8, %rbx
    cmpq  %r12, %rbx
    jne   .L5
.L14:
    popq  %rbx
    xorl  %eax, %eax
    popq  %rbp
    popq  %r12
    ret
```

Why is “branch” problematic in performance?

```
① addq $8, %rbx
② cmpq %r12, %rbx
③ jne  .L5
```



The latency of executing the cmpq instruction

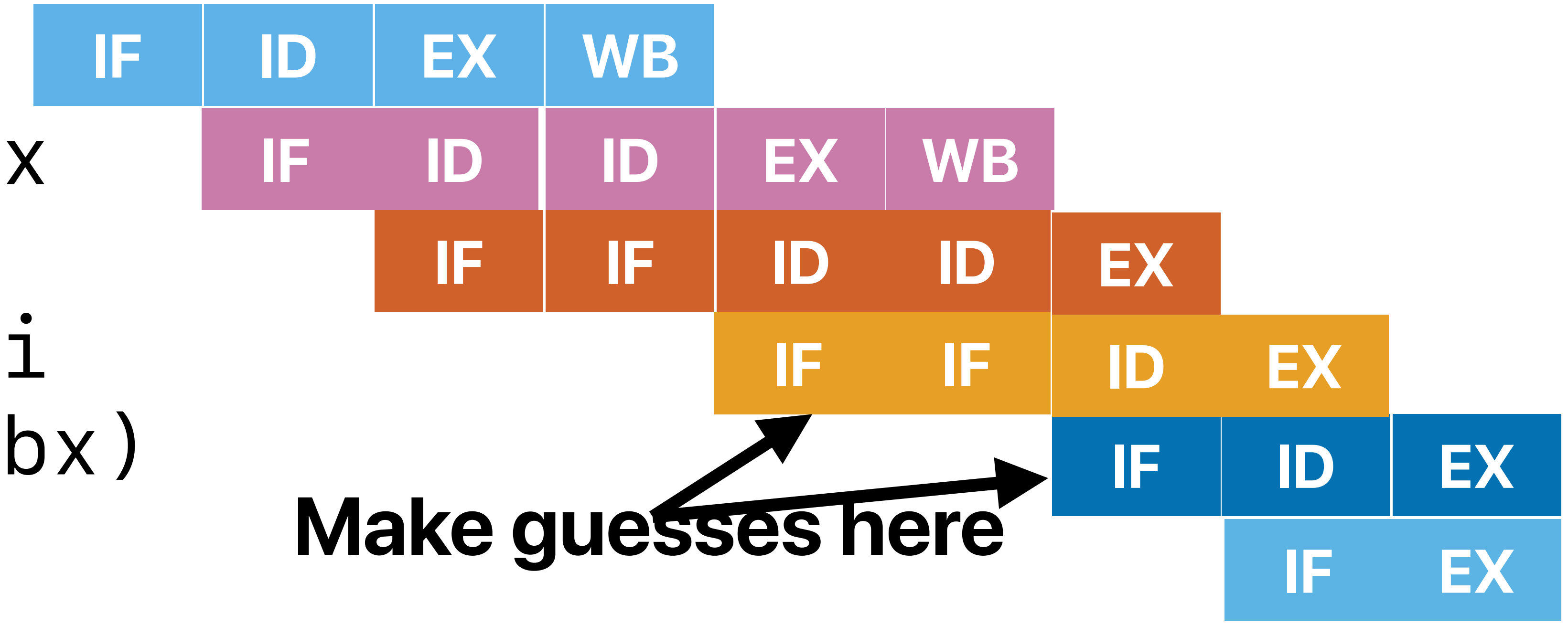
We have to wait almost as long as the latency of the previous instruction to make a decision — we cannot fetch anything before that

Takeaways: branch predictions

- The cost of not to predict a branch is to stall until the data dependency is resolved

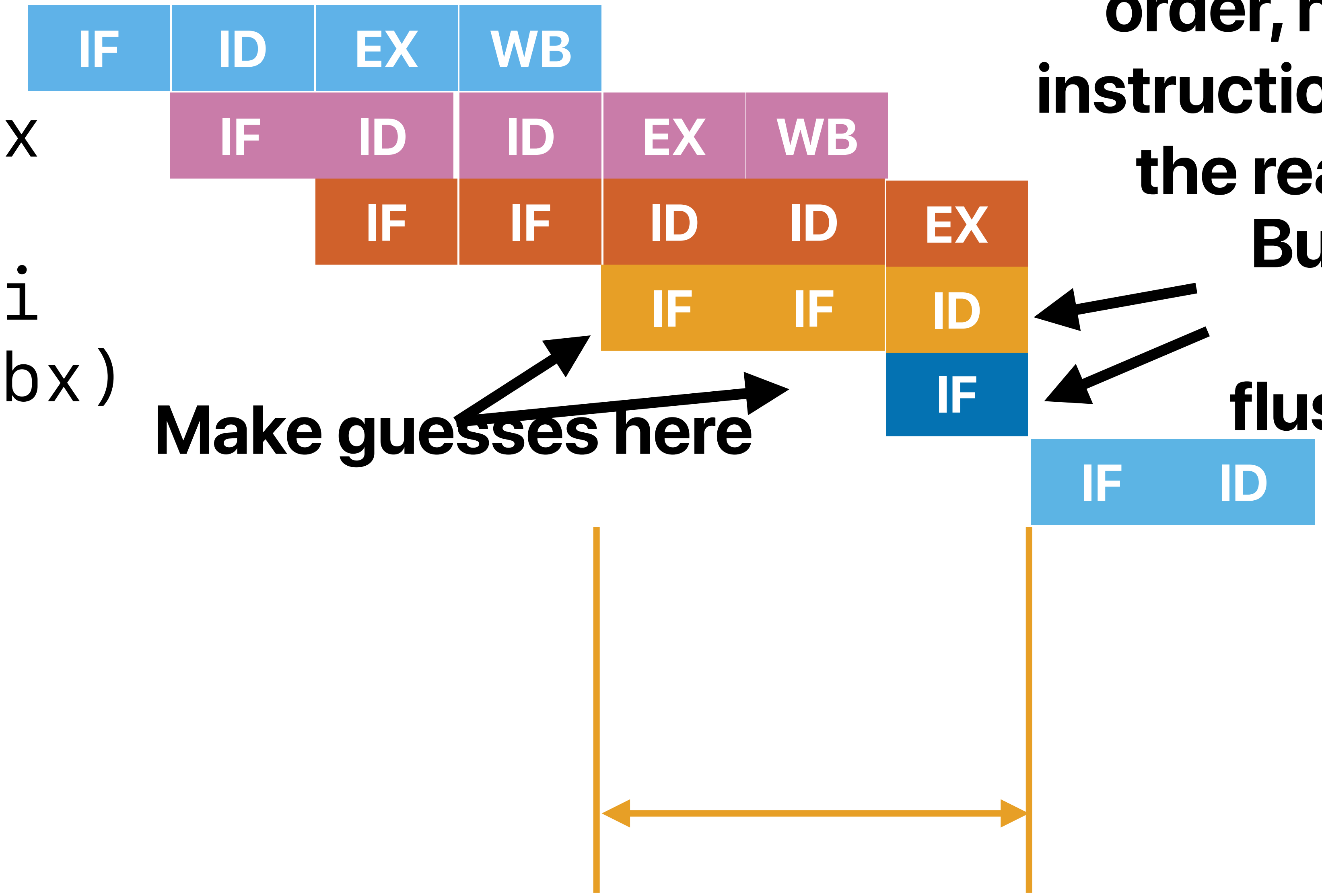
Prediction: What if we guessed right?

```
① 	addq  $8, %rbx
② 	cmpq  %r12, %rbx
③ 	jne   .L5
④ .L5:	movq  %rbx, %rdi
⑤ 	cmpq  %rbp, (%rbx)
```



Prediction: What if we are wrong?

```
①       addq  $8, %rbx
②       cmpq  %r12, %rbx
③       jne   .L5
④ .L5:   movq  %rbx, %rdi
⑤       cmpq  %rbp, (%rbx)
⑥ .L14:  popq  %rbx
```



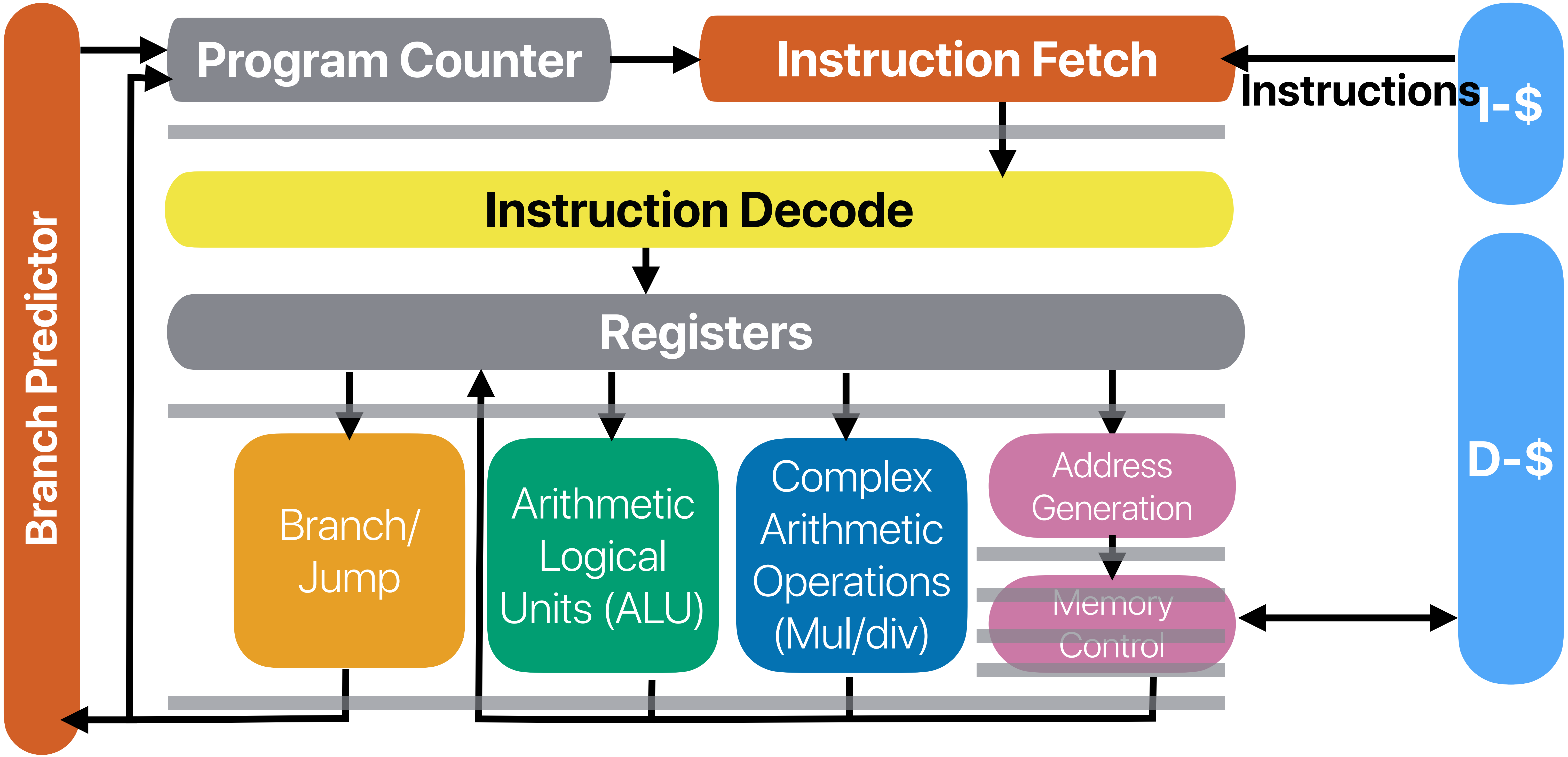
Make guesses here

Since we execute in-order, no predicted instructions will modify the real registers

Bubble the rest stages by flushing pipeline registers

We still only wasted three cycles — same as not doing anything

Microprocessor with a “branch predictor”



Takeaways: branch predictions

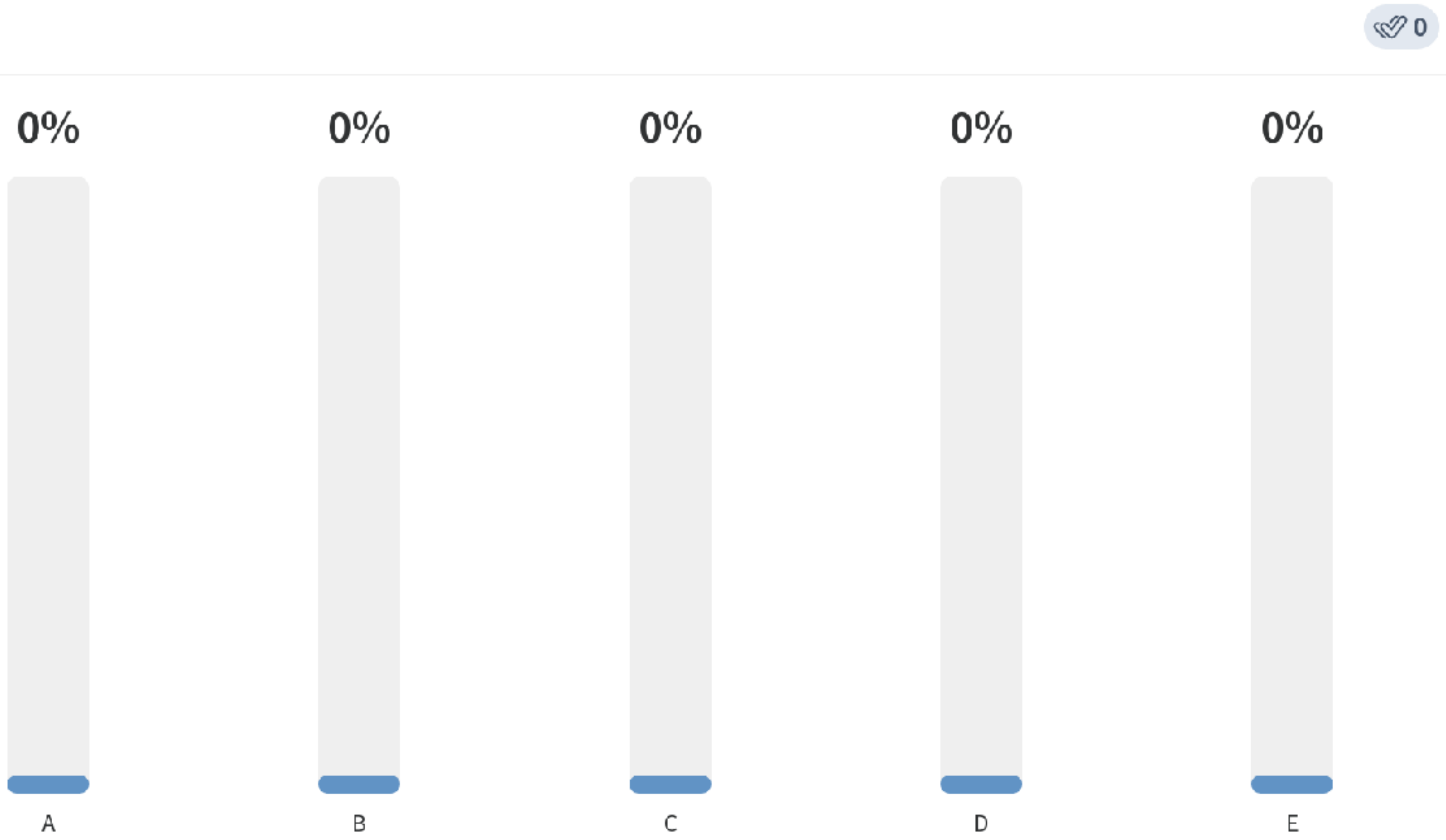
- The cost of not to predict a branch is to stall until the data dependency is resolved
- Branch predictions allow the processor to at least make some progress and hide the stalls if we guessed correctly!

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

What should branch prediction “predict”

- How many of the following statements are true regarding the why is branch can lead to serious performance issues
  1. The result value of the previous instruction generating the input to the branch
  2. The direction of the branch (i.e., taken or not-taken)
  3. The target address of the branch
  4. The forth-through address of the branch

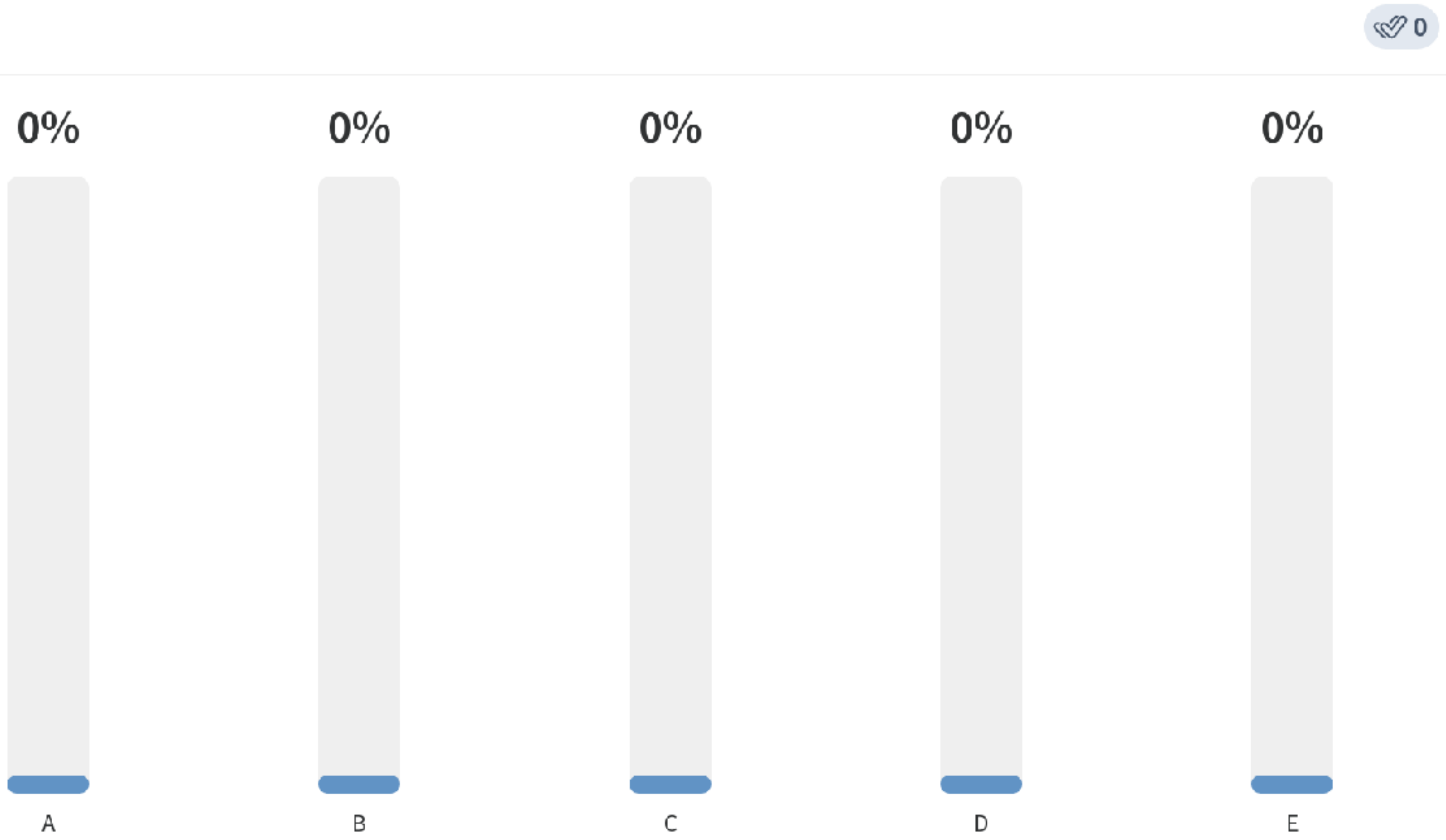
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4


0
0%
0%
0%
0%
0%
A
B
C
D
E

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

What should branch prediction "predict"

- How many of the following statements are true regarding the why is branch can lead to serious performance issues
  ① The result value of the previous instruction generating the input to the branch
  ② The direction of the branch (i.e., taken or not-taken)
  ③ The target address of the branch
  ④ The forth-through address of the branch
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4


0
0%
0%
0%
0%
0%
A
B
C
D
E

What should branch prediction "predict"

- How many of the following statements are true regarding the why is branch can lead to serious performance issues
  1. The result value of the previous instruction generating the input to the branch
  2. The direction of the branch (i.e., taken or not-taken)
  3. The target address of the branch
  4. The forth-through address of the branch

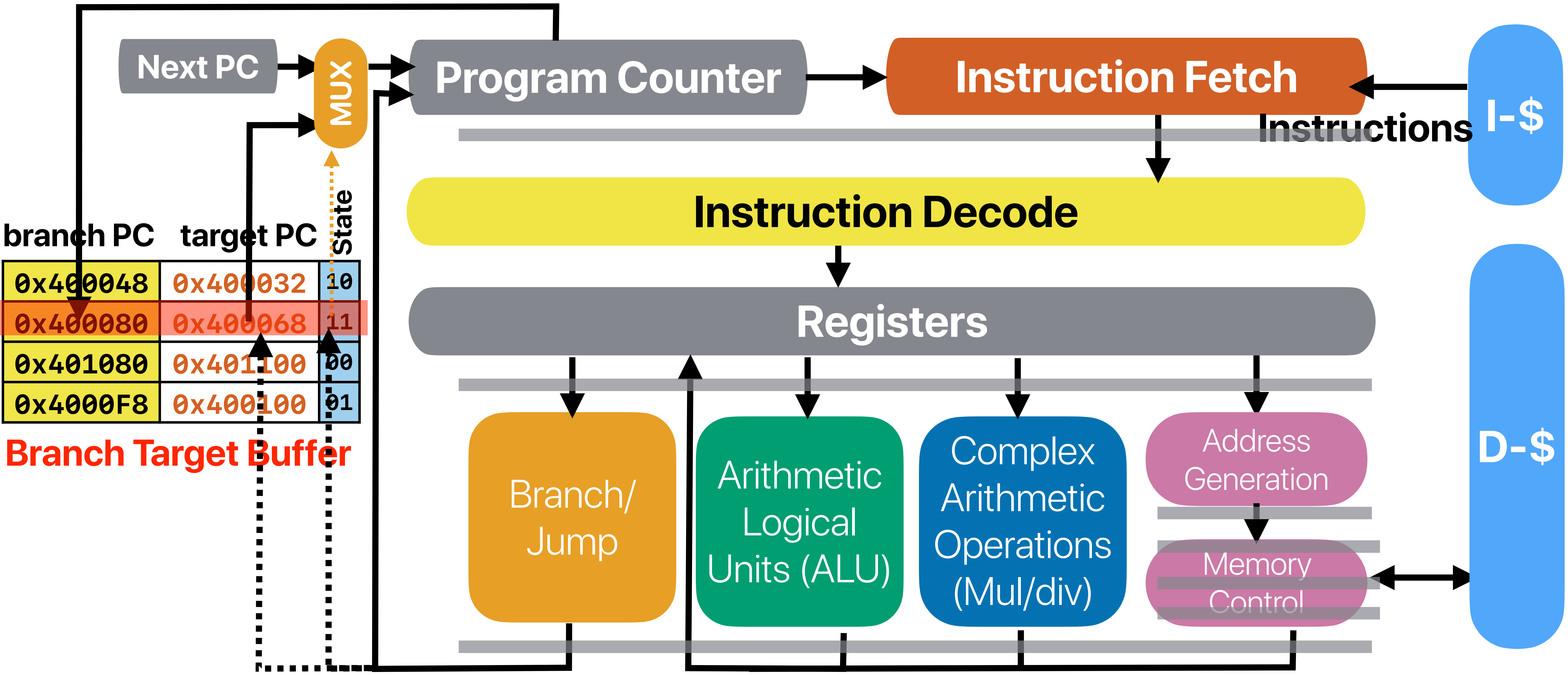
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

What are the "outcome" of the branch?

- **Taken, not-take** **You need to predict that — history/states**
- **Target address, if taken**

You need a cheatsheet for that — branch target buffer

Detail of a basic dynamic branch predictor

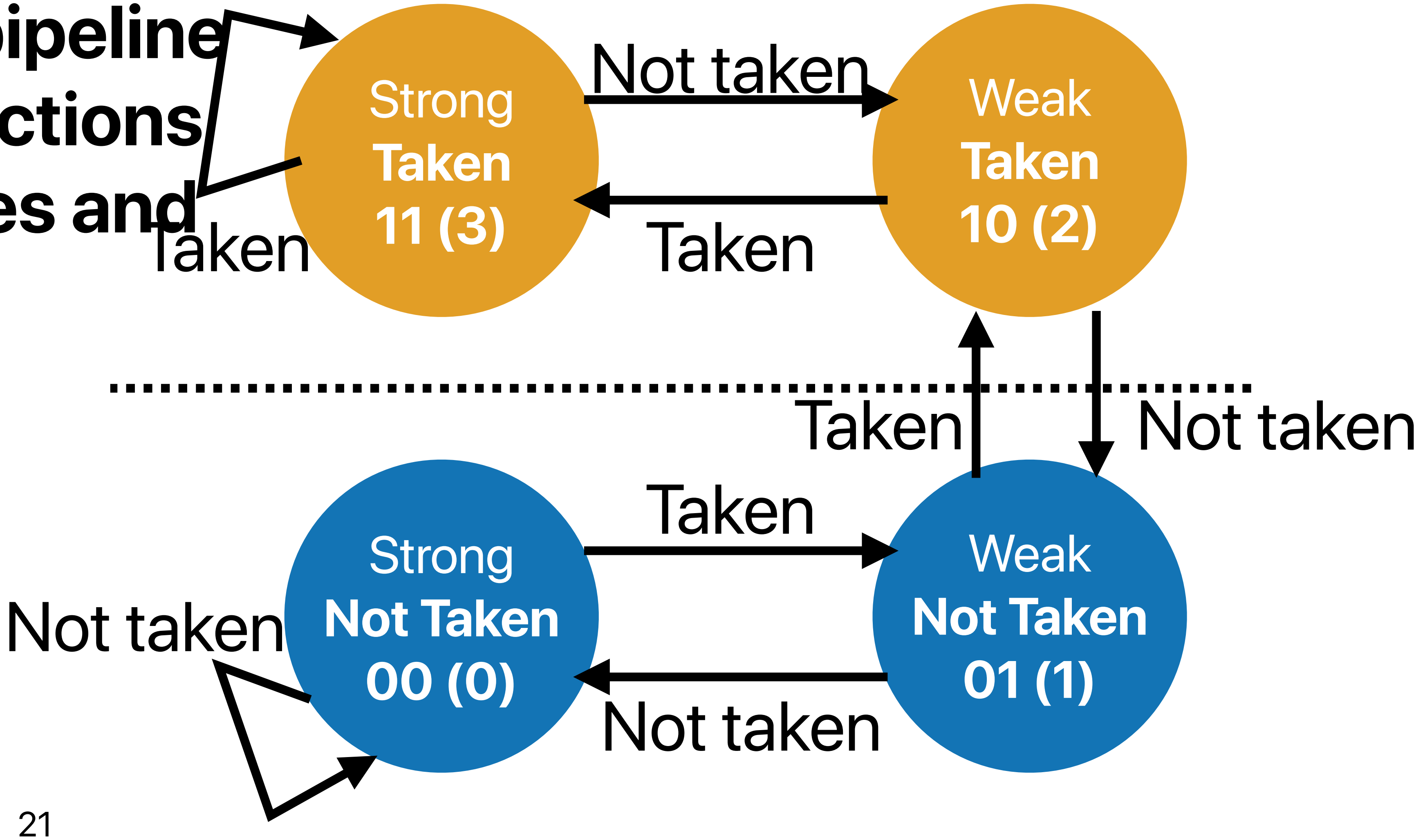


2-bit/Bimodal local predictor

- Local predictor — every branch instruction has its own state
- 2-bit — each state is described using 2 bits
- Change the state based on **actual** outcome
- If we guess right — no penalty
- **If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF and ID stages and reset the PC**

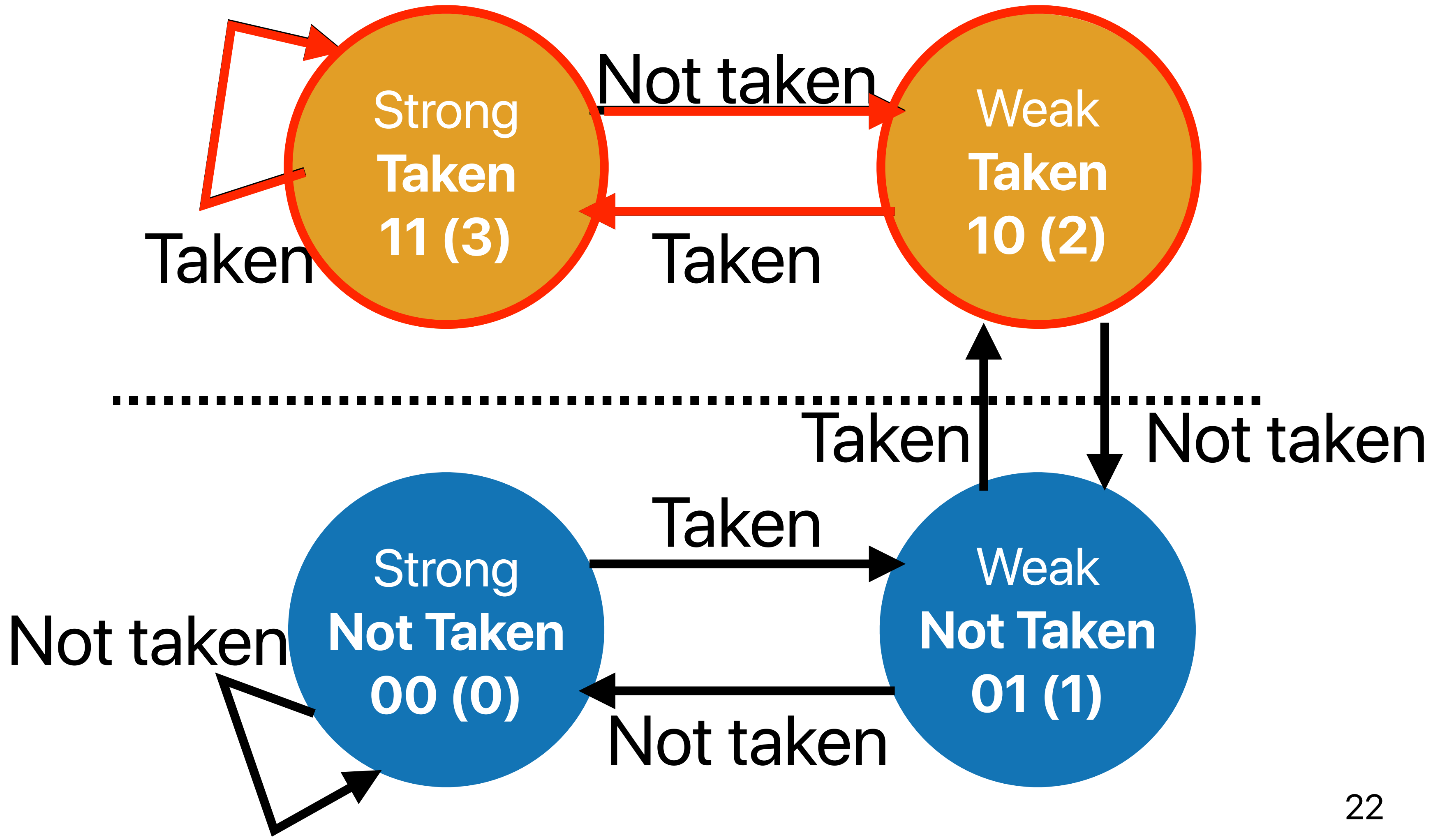
branch PC	target PC	State
0x400048	0x400032	10
0x400080	0x400068	11
0x401080	0x401100	00
0x4000F8	0x400100	01

Predict Taken



2-bit local predictor

```
i = 0;
do {
      sum += a[i];
} while(++i < 10);
```



i	state	predict	actual
1	10	T	T
2	11	T	T
3	11	T	T
4-9	11	T	T
10	11	T	NT

90% accuracy!

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

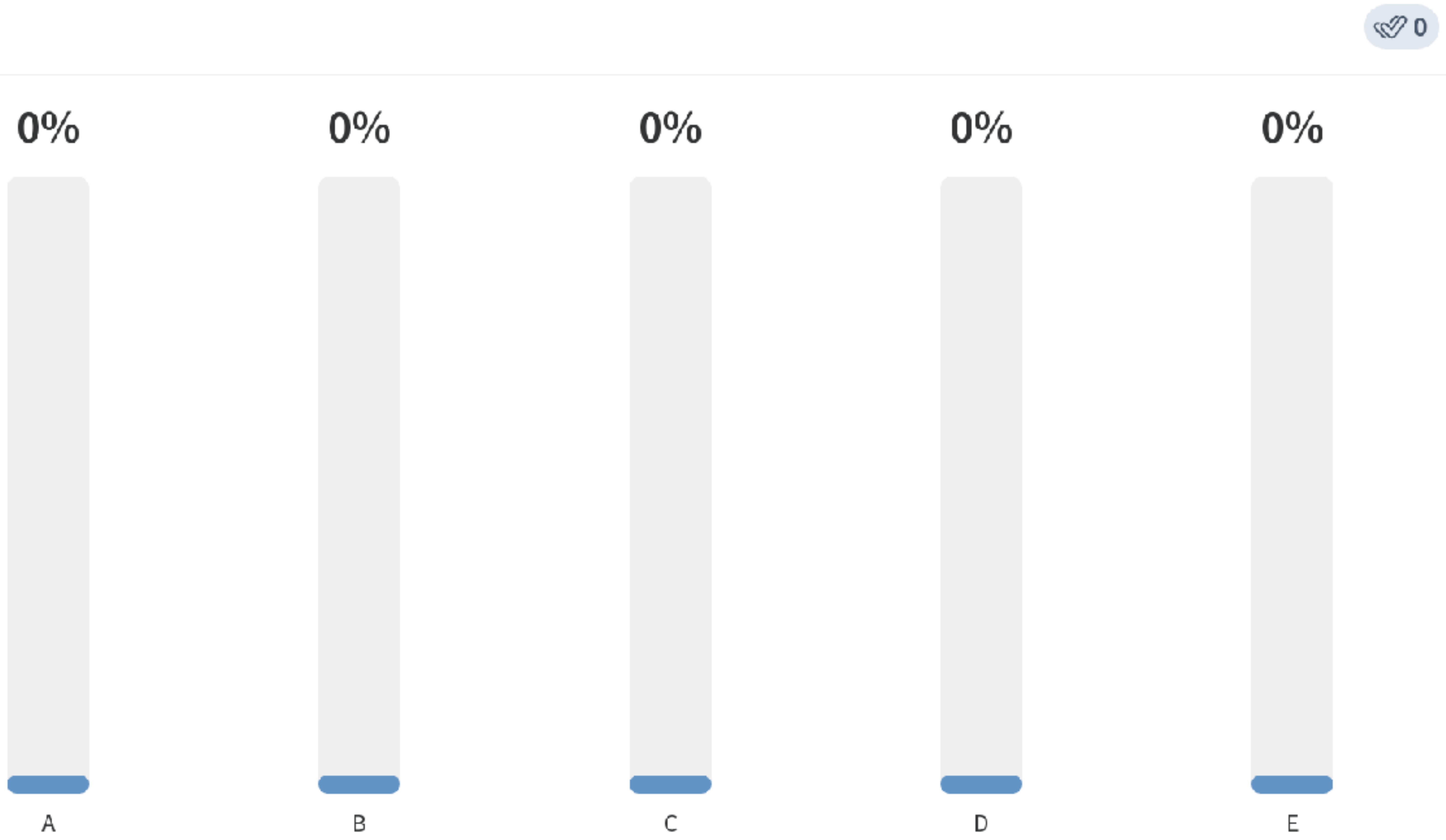
Demo revisited

- Assume that we have a 2-bit local predictor and the values in data is randomly distributed in the number space, what's the branch prediction accuracy of branch X when option is "0" and "1". You may also assume the predictors' states start with 0s.

	Without sorting	After sorting
A	100%	0%
B	50%	0%
C	50%	50%
D	50%	100%
E	0%	100%

```
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)  // Branch X
            sum ++;
    }
}
```


0
0%
0%
0%
0%
0%
A
B
C
D
E

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

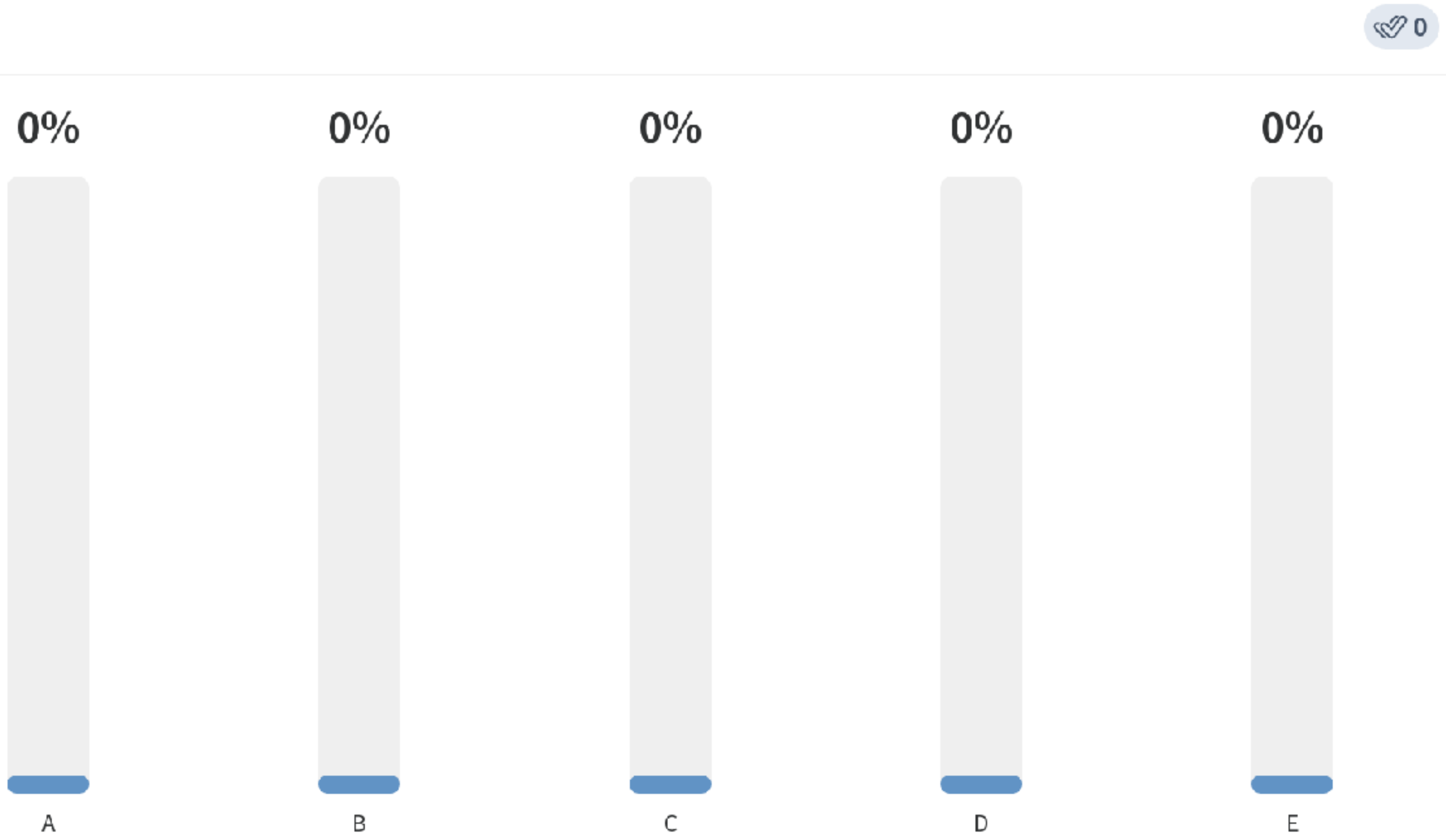
Demo revisited

- Assume that we have a 2-bit local predictor and the values in data is randomly distributed in the number space, what's the branch prediction accuracy of branch X when option is "0" and "1". You may also assume the predictors' states start with 0s.

	Without sorting	After sorting
A	100%	0%
B	50%	0%
C	50%	50%
D	50%	100%
E	0%	100%

```
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)  // Branch X
            sum ++;
    }
}
```


0
0%
0%
0%
0%
0%
A
B
C
D
E

Demo revisited

- Assume that we have a 2-bit local predictor and the values in data is randomly distributed in the number space, what's the branch prediction accuracy of branch X when option is "0" and "1". You may also assume the predictors' states start with 0s.

	Without sorting	After sorting
A	100%	0%
B	50%	0%
C	50%	50%
D	50%	100%
E	0%	100%

```
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)  // Branch X
            sum ++;
    }
}
```

Demo revisited

- Assume that we have a 2-bit local predictor and the values in data is randomly distributed in the number space, what’s the branch prediction accuracy of branch X when option is “0” and “1”. You may also assume the predictors’ states start with 0s.

	Without sorting	After sorting
A	100%	0%
B	50%	0%
C	50%	50%
D	**50%**	**100%**
E	0%	100%

```
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)  // Branch X
```

	Without sorting	With sorting
The prediction accuracy of X before threshold	50%	100%
The prediction accuracy of X after threshold	50%	100%

Demo revisited

If there is no branch predictor on the processor, the code w/ sorting will be slower — but every processor has branch predictors now

	Without sorting	After sorting
A	100%	0%
B	50%	0%
C	50%	50%
D	50%	100%
E	0%	100%

```
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)  // Branch X
```

	Without sorting	With sorting
The prediction accuracy of X before threshold	50%	100%
The prediction accuracy of X after threshold	50%	100%

How can we evaluate 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!

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

23 cycles on AMD Zen 3

Could be more expensive than cache misses

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

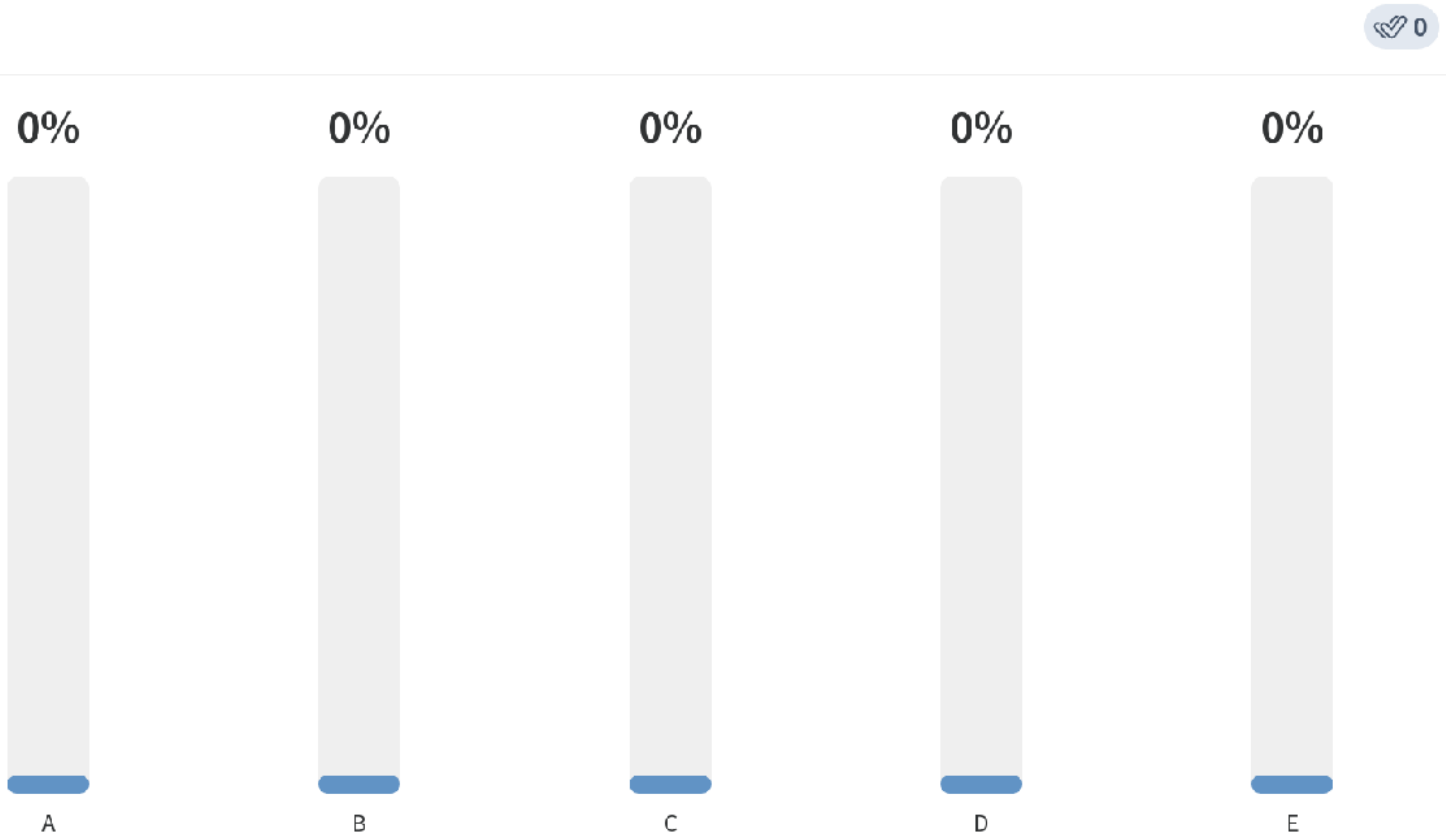
2-bit local predictor

- 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
```

(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%


0
0%
0%
0%
0%
0%
A
B
C
D
E

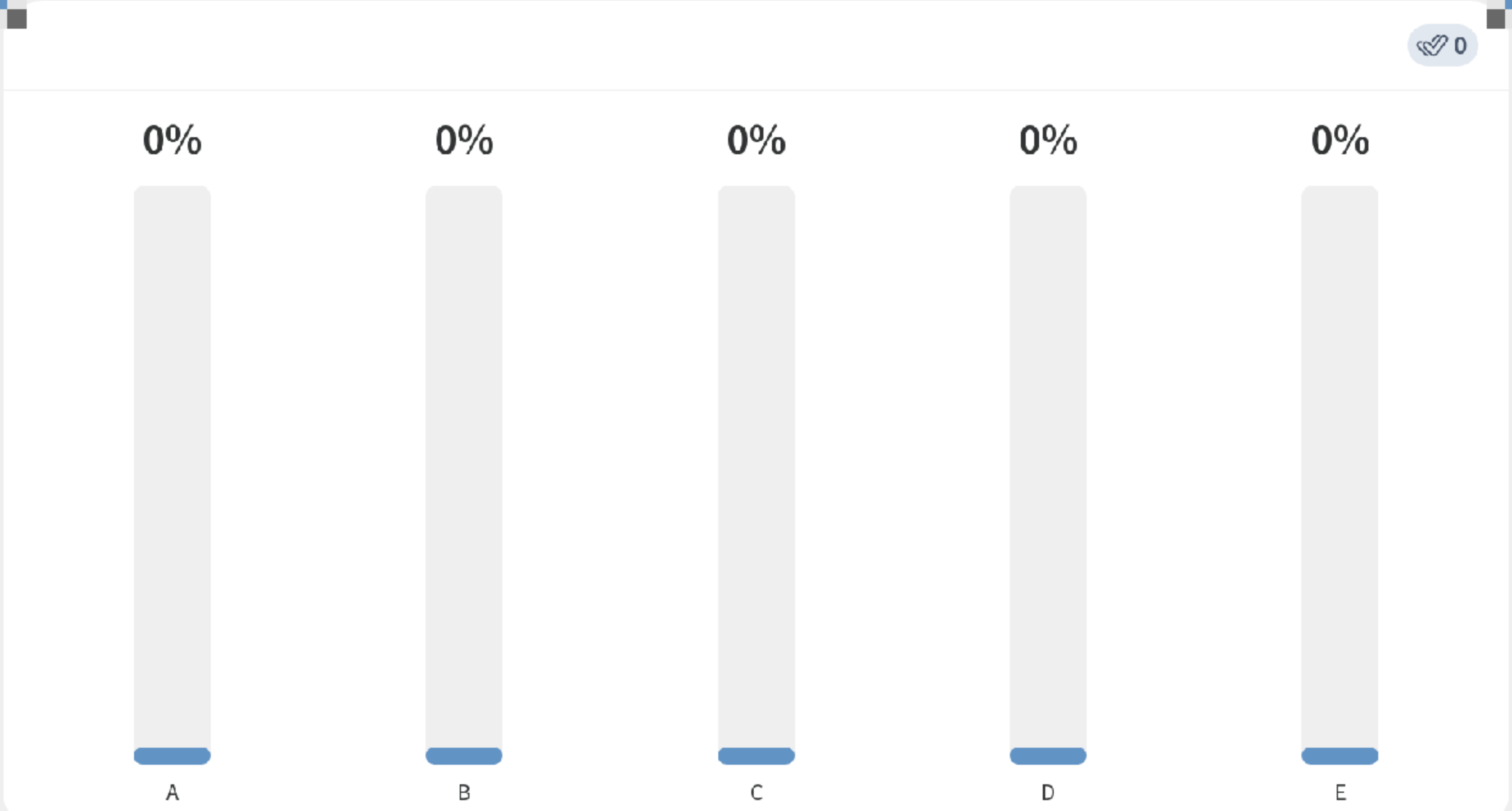
2-bit local predictor

- 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
```

(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%


0
0%
0%
0%
0%
0%
A
B
C
D
E

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%

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 prediction based on the state of each branch instruction

Two-level global predictor

Marius Evers, Sanjay J. Patel, Robert S. Chappell, and Yale N. Patt. 1998. An analysis of correlation and predictability: what makes two-level branch predictors work. In Proceedings of the 25th annual international symposium on Computer architecture (ISCA '98).

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
```

(assume all states started with 00)

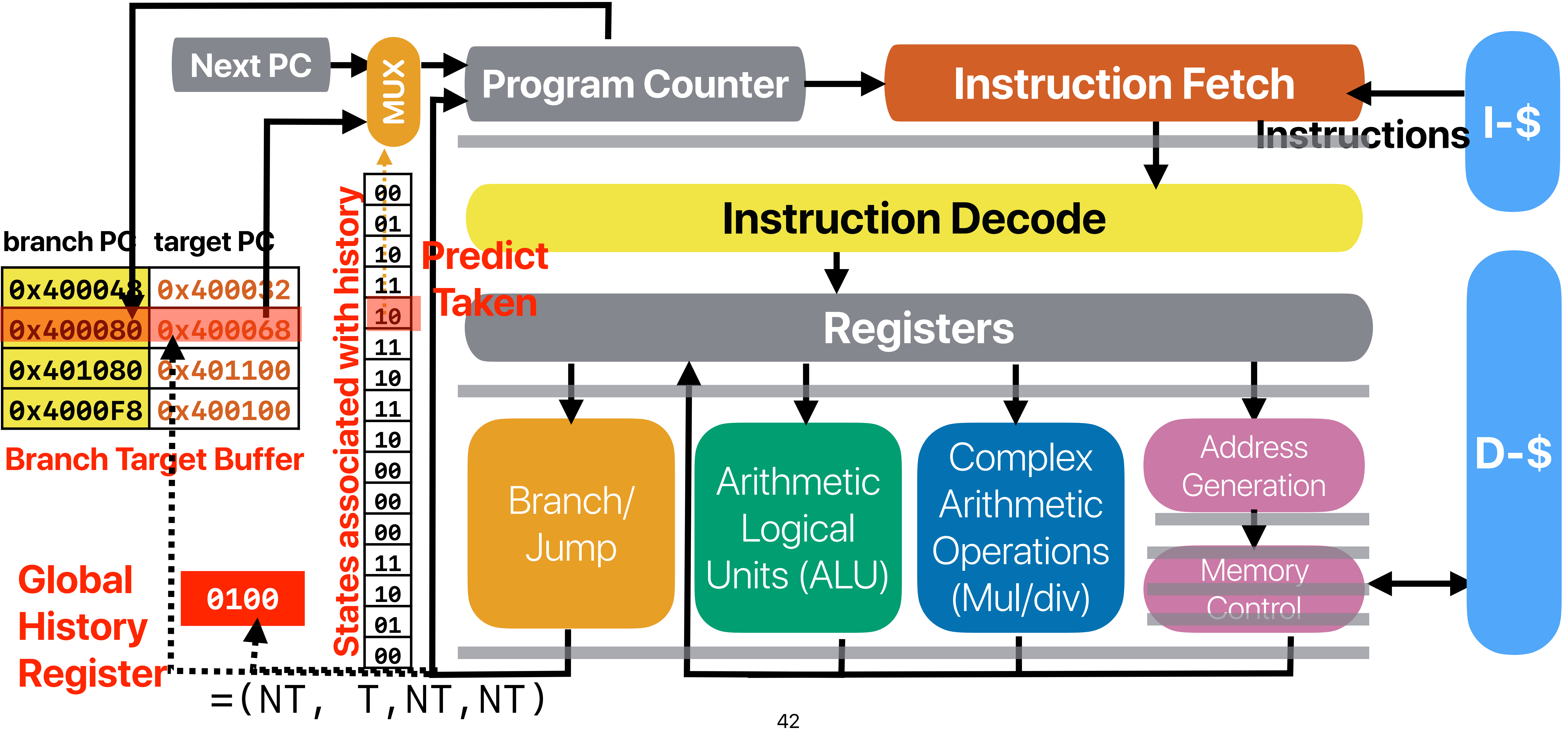
A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%

i	branch?	state	prediction	actual
0	X	00	NT	T
1	Y	00	NT	T
[illegible]	[illegible]	[illegible]1	NT	NT
2	Y	01	NT	T
2	X	00	NT	T
3	Y	10	[illegible]	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

This pattern repeats all the time!

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

Detail of a basic dynamic branch predictor



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
```

Near perfect after this

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

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);
```

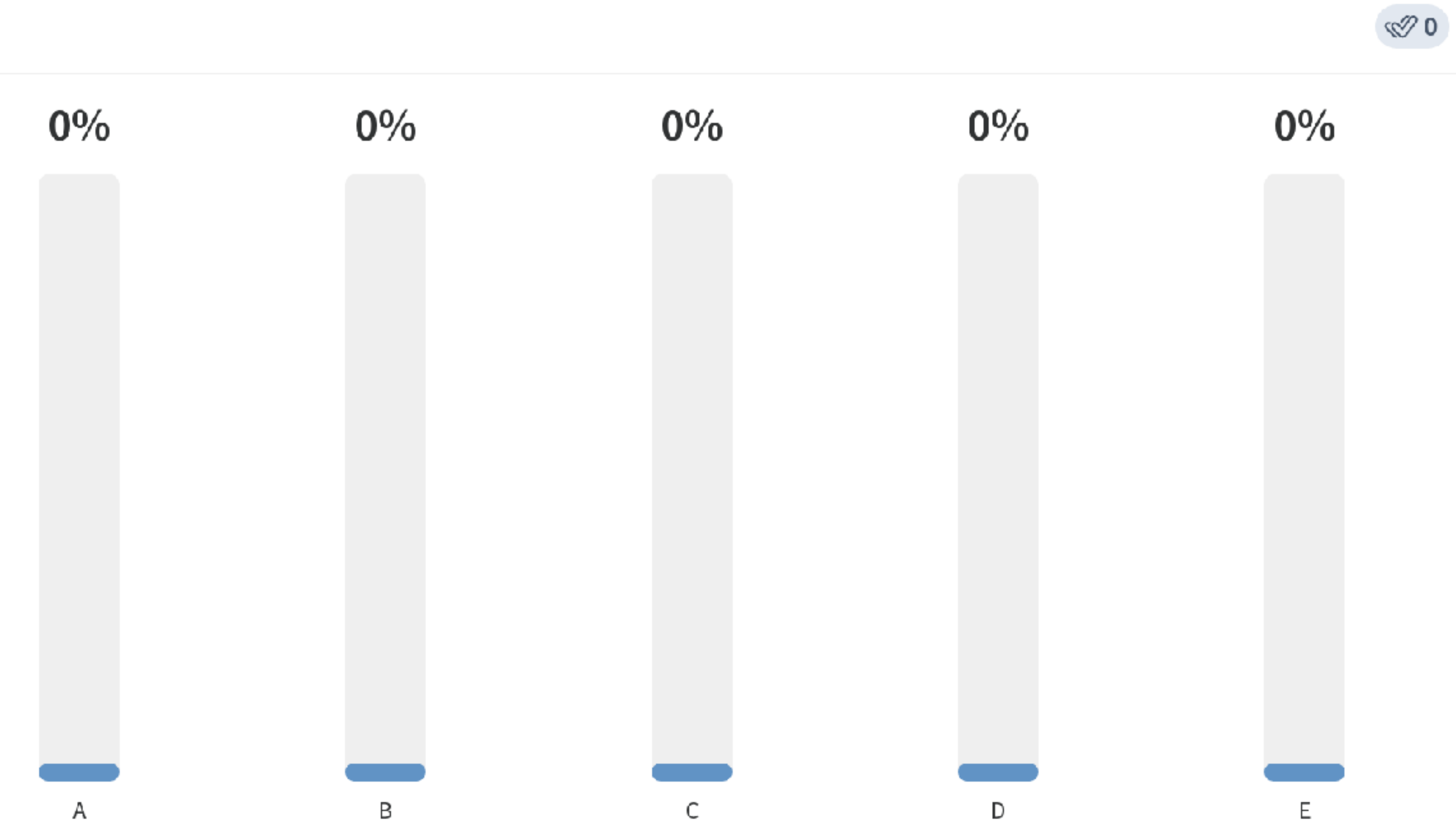
III

```
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


0
0%
0%
0%
0%
0%
A
B
C
D
E

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)?

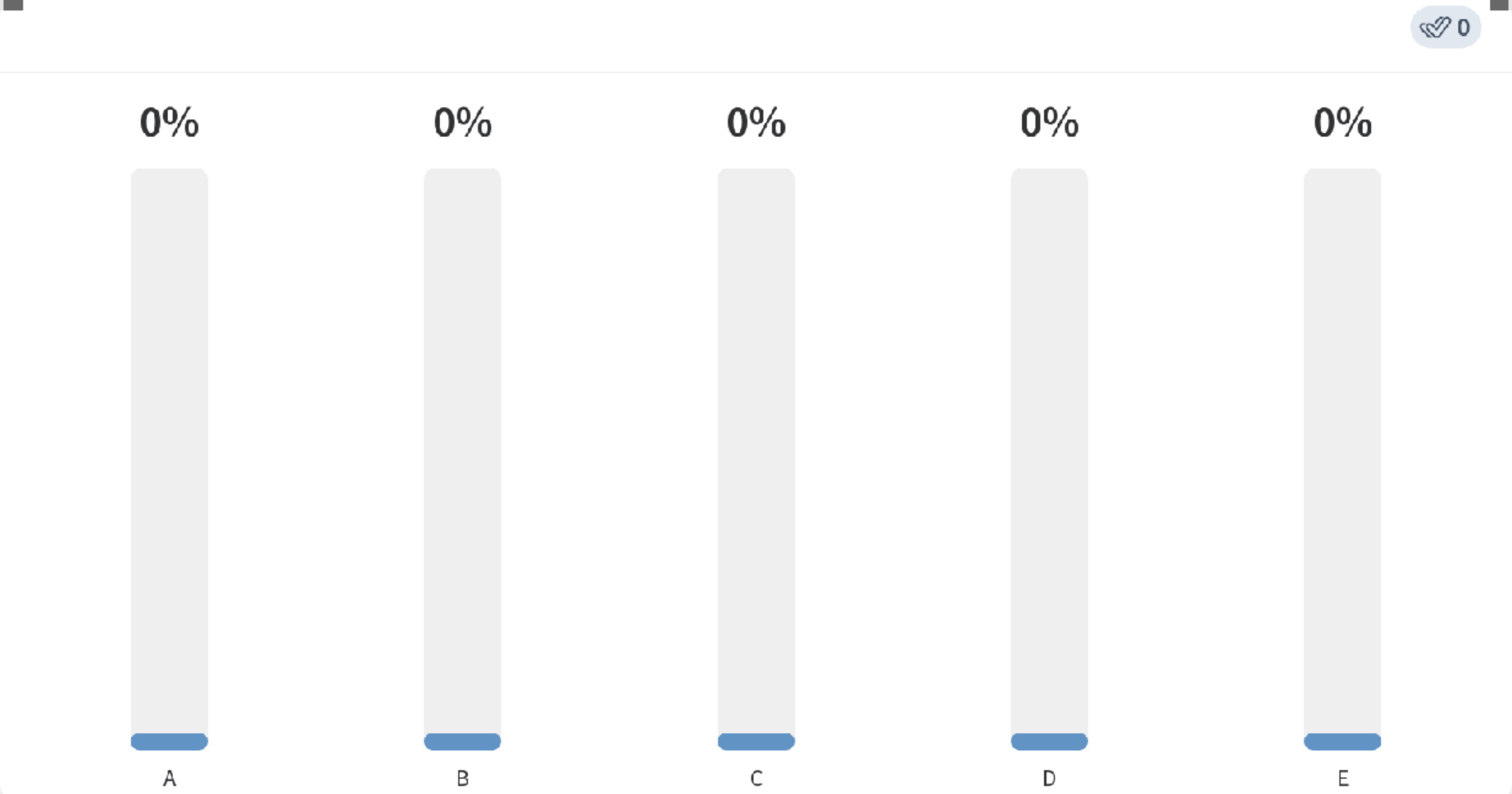
I
```
i = 0;
do {
    if( i % 10 != 0)
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100);
```

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


0
0%
0%
0%
0%
0%
A
B
C
D
E

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)
```

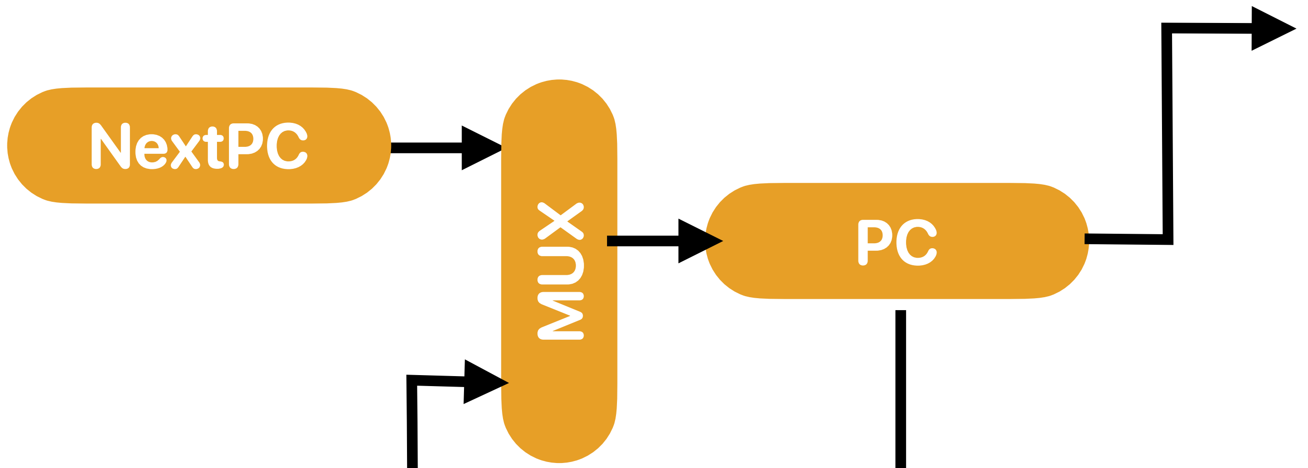
A. 0
B. 1
C. 2
D. 3
E. 4

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 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

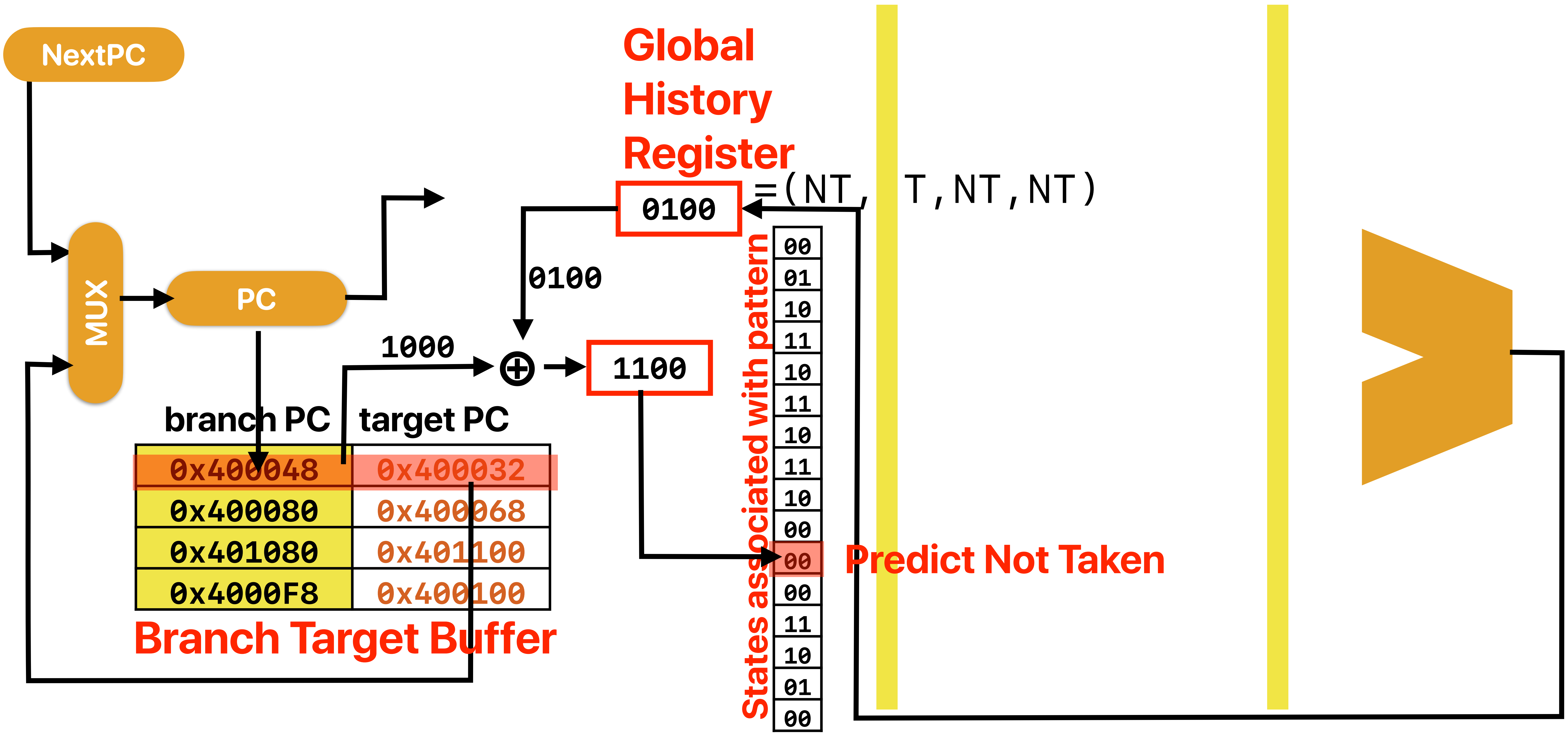
States associated with history

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

gshare predictor



gshare predictor

- Allowing the predictor to identify both branch address but also use global history for more accurate prediction

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

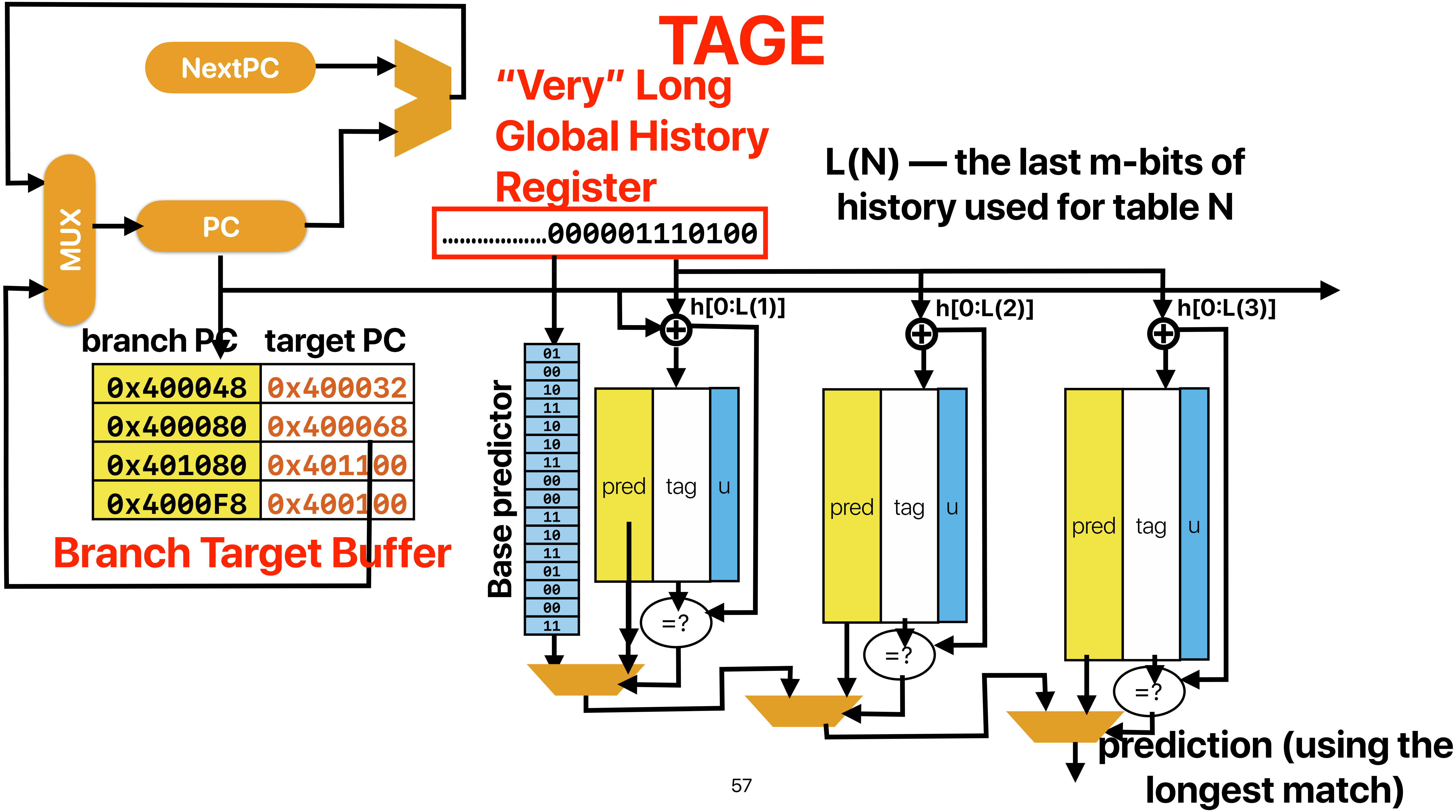
D. 3

E. 4

different branch needs different length of history

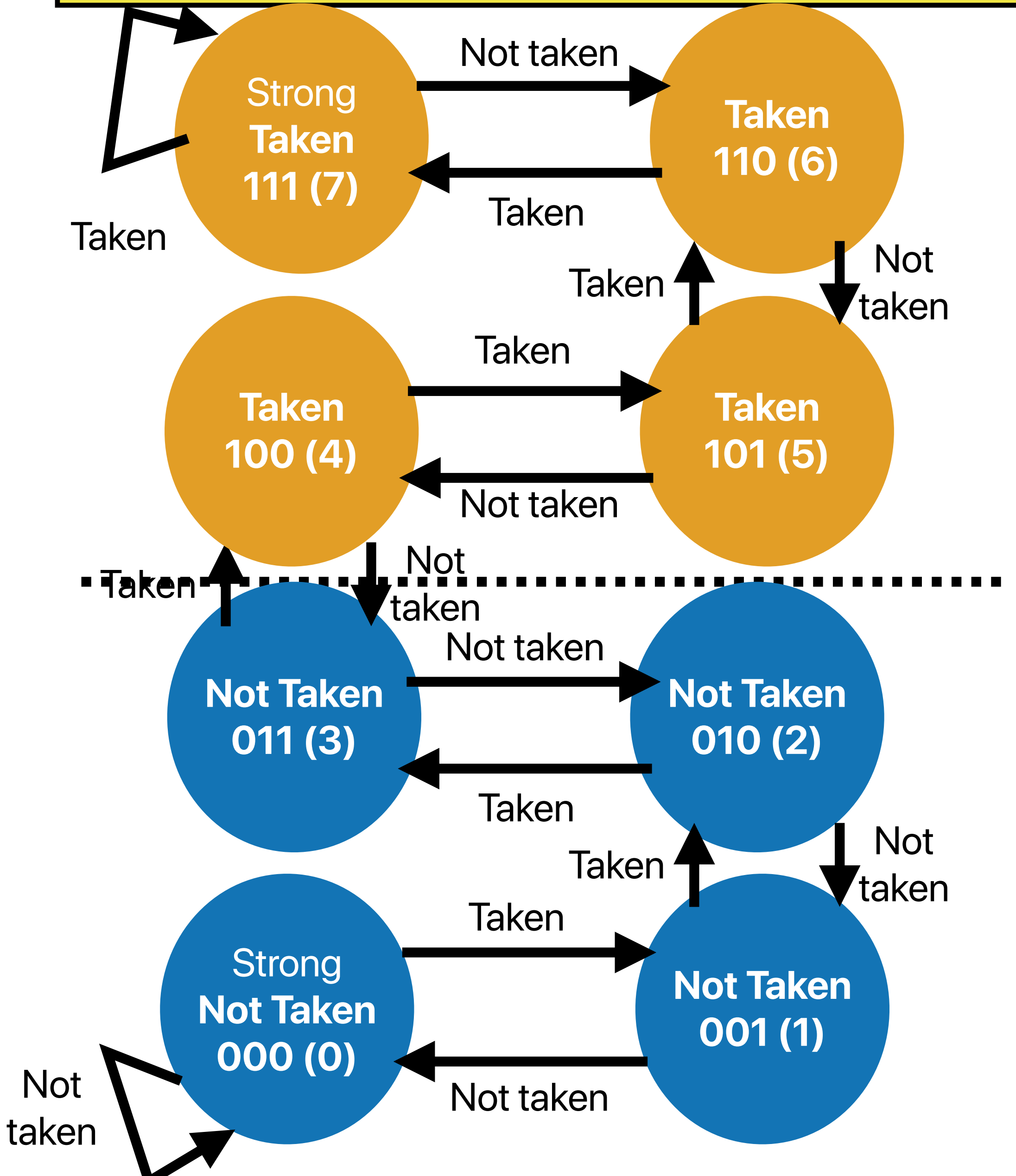
global predictor can work if the history is long enough!

TAGE


NextPC
MUX
PC
"Very" Long
Global History
Register
........................000001110100
L(N) — the last m-bits of
history used for table N
h[0:L(1)]
h[0:L(2)]
h[0:L(3)]
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
pred
tag
u
=?
prediction (using the
longest match)

What's inside each table?

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

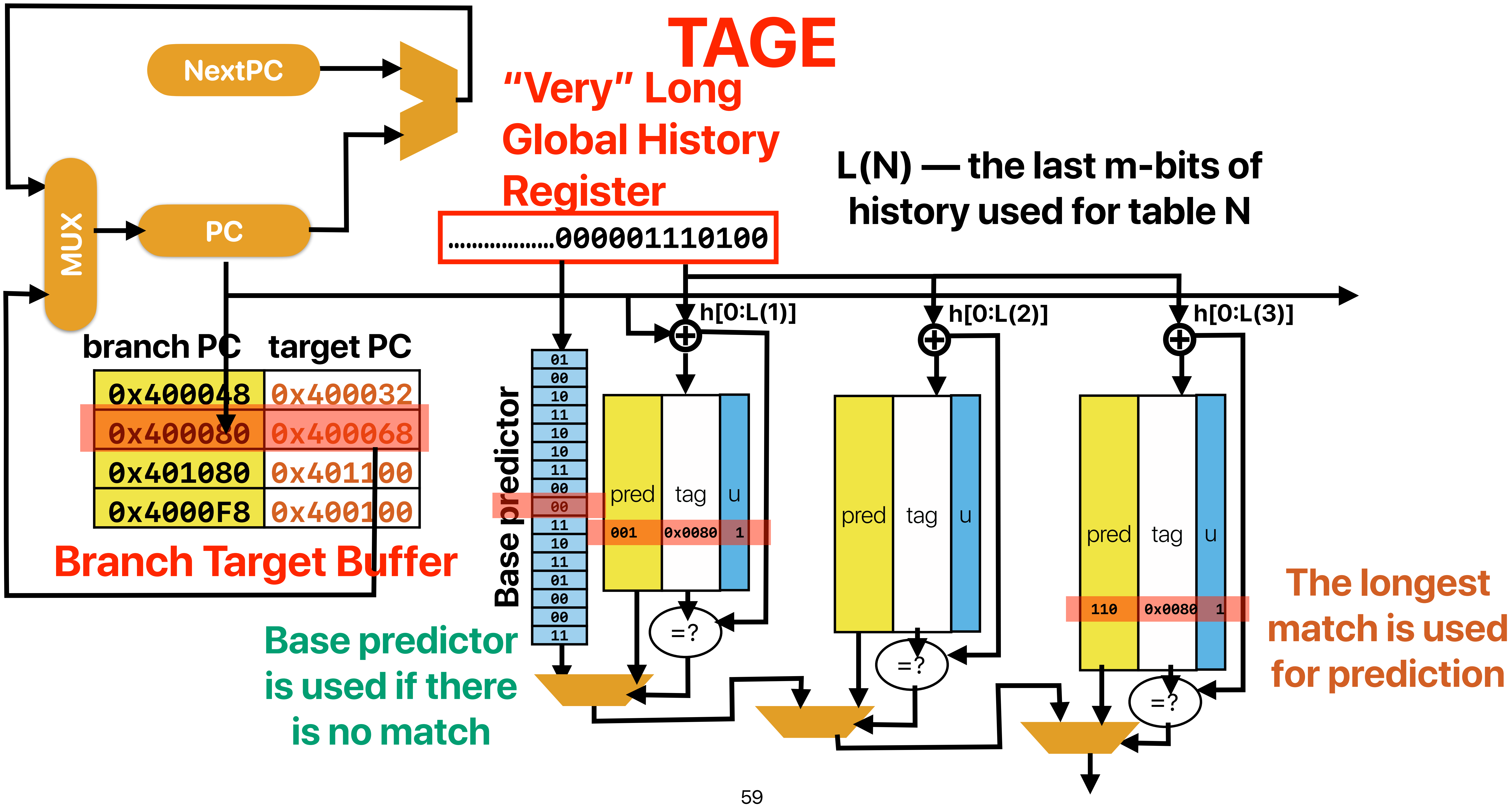


$$if\ prediction(alt\_predictor) \neq prediction(pred):$$

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

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

TAGE


NextPC
MUX
PC
"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
h[0:L(1)]
h[0:L(2)]
h[0:L(3)]
pred
tag
u
001
0x0080
1
110
0x0080
1
=?
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

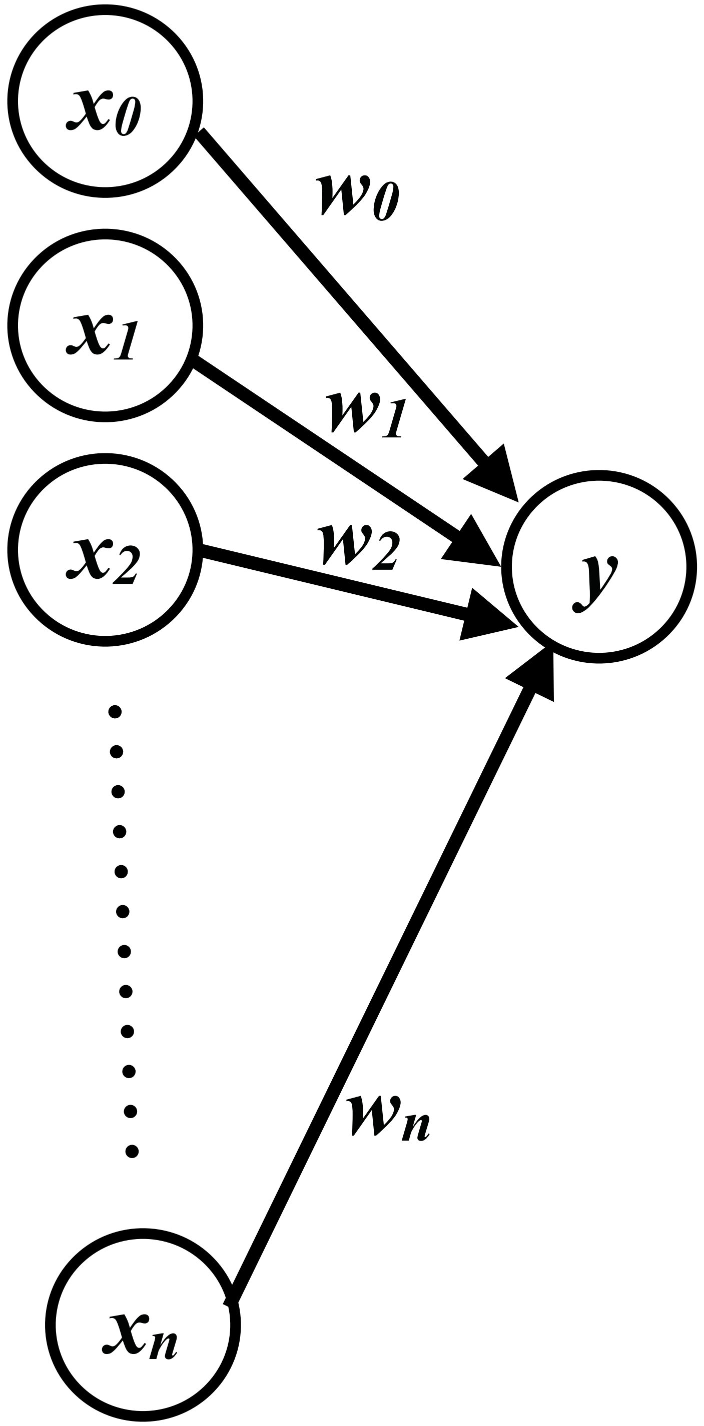
- The machine learns to predict conditional branches
- Artificial neural networks
  - Simple model of neural networks in brain cells
  - Learn to recognize and classify patterns

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



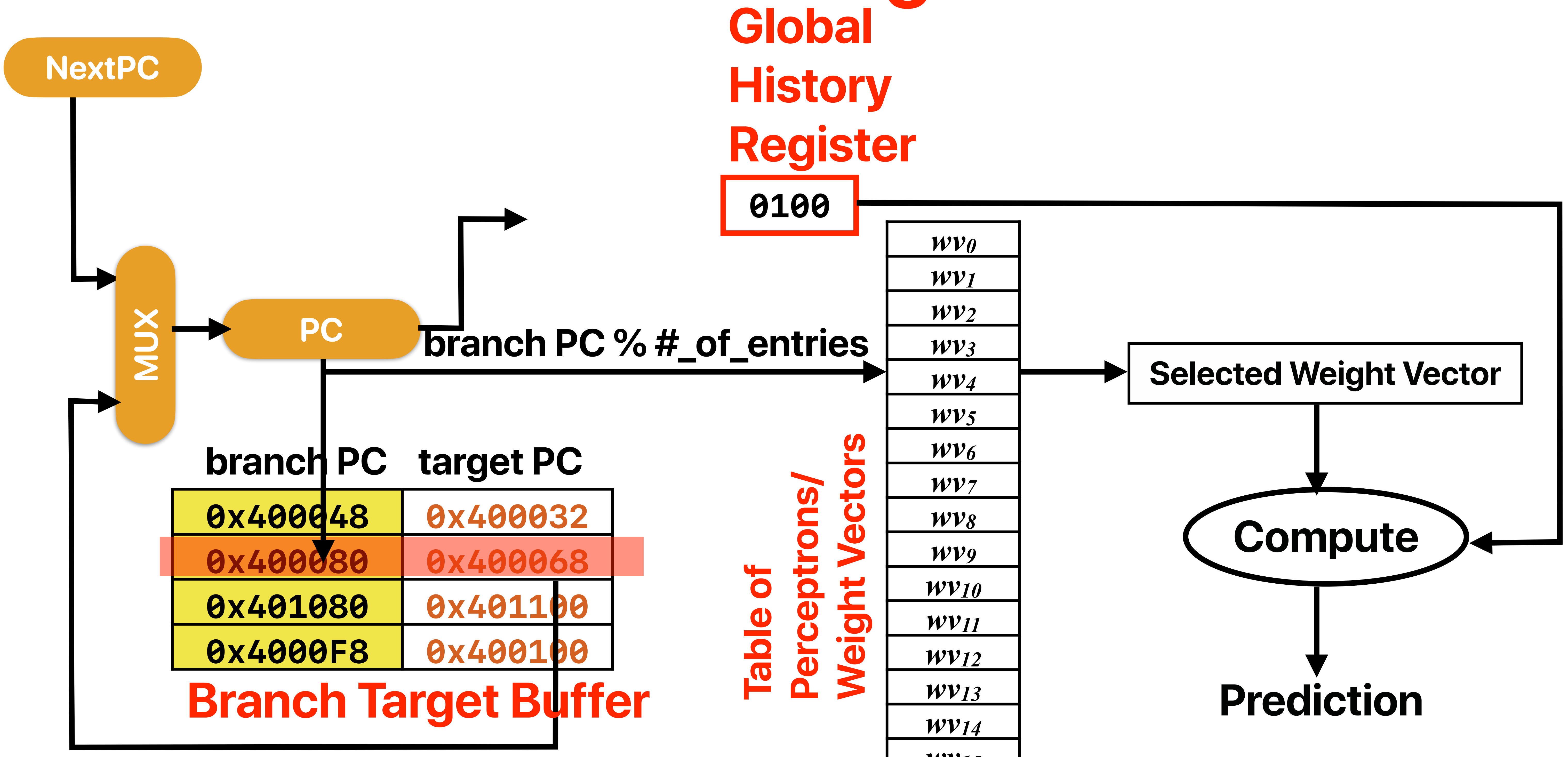
$$y = w_0 + \sum_{i=1}^{n} x_i w_i$$

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 $|y| \leq \theta$ **or** $((y \geq 0) \neq t)$ **then**
    for each $0 \leq i \leq n$ **in parallel**
        if $t = x_i$ **then**
            $w_i := w_i + 1$
        else
            $w_i := w_i - 1$
        end if
    end for
end if

Predictor Organization



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

Design decisions in real practice

- Based on D. Suggs, D. Bouvier, M. Subramony and K. Lepak's "Zen 2", 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 reflects on 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

0

A

0%

B

0%

C

0%

D

0%

E

0%

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

Design decisions in real practice

- Based on D. Suggs, D. Bouvier, M. Subramony and K. Lepak's "Zen 2", 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 reflects on 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 — Group

0

A	0%
B	0%
C	0%
D	0%
E	0%

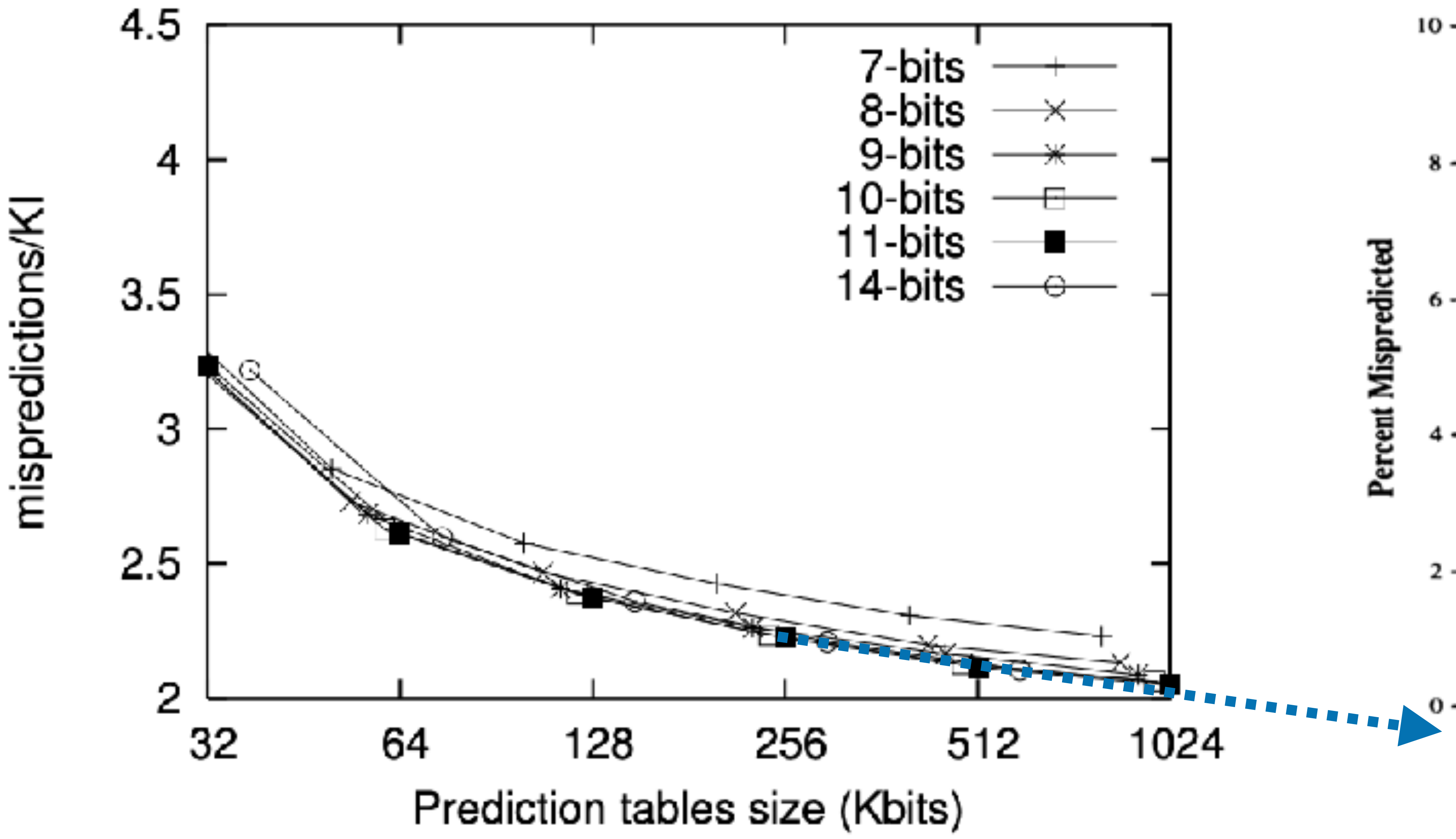
Design decisions in real practice

- Based on D. Suggs, D. Bouvier, M. Subramony and K. Lepak's "Zen 2", 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 reflects on 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

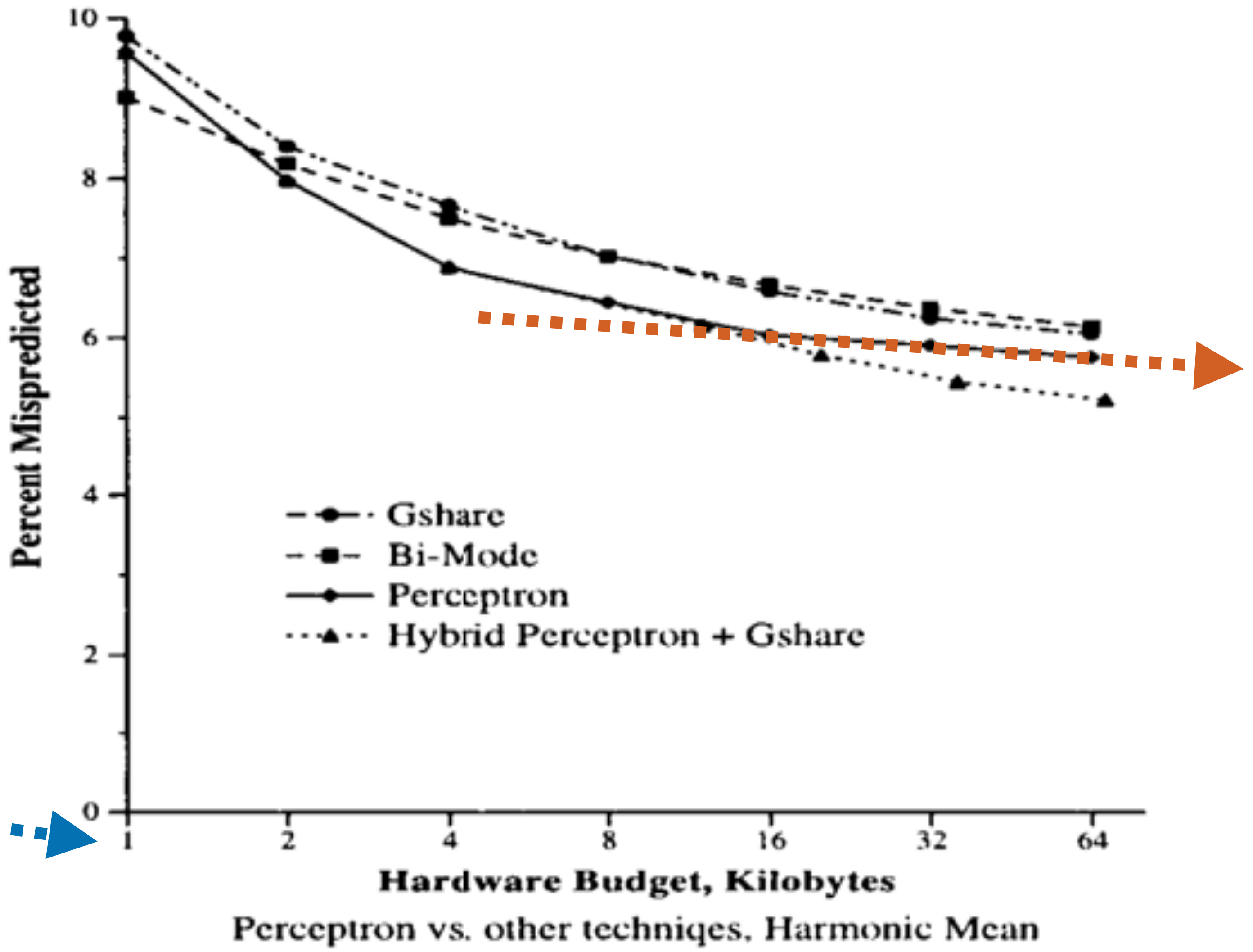
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

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?

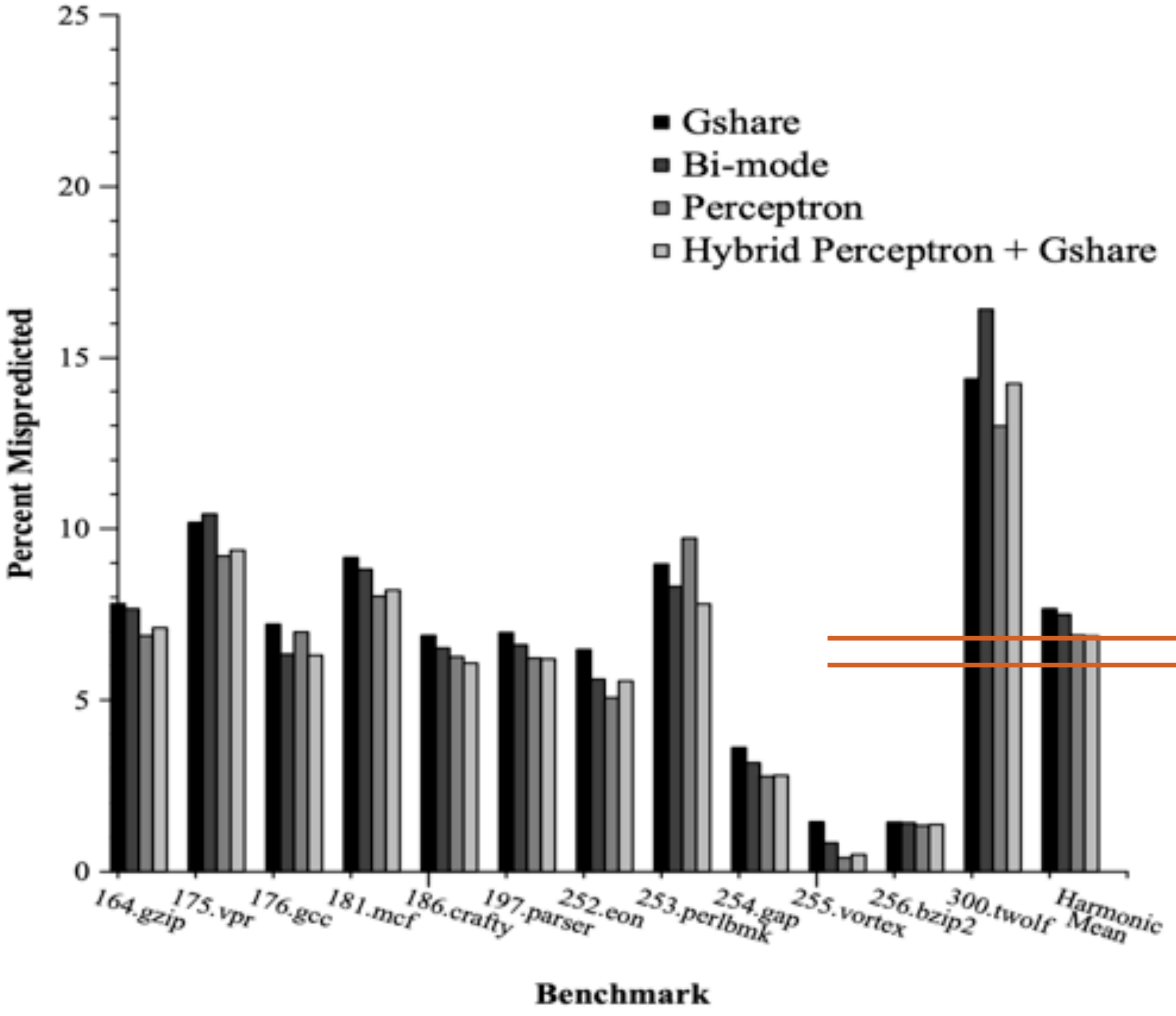


Figure 4: Misprediction Rates at a 4K budget. The perceptron predictor has a lower misprediction rate than *gshare* for all benchmarks except for `186.crafty` and `197.parser`.

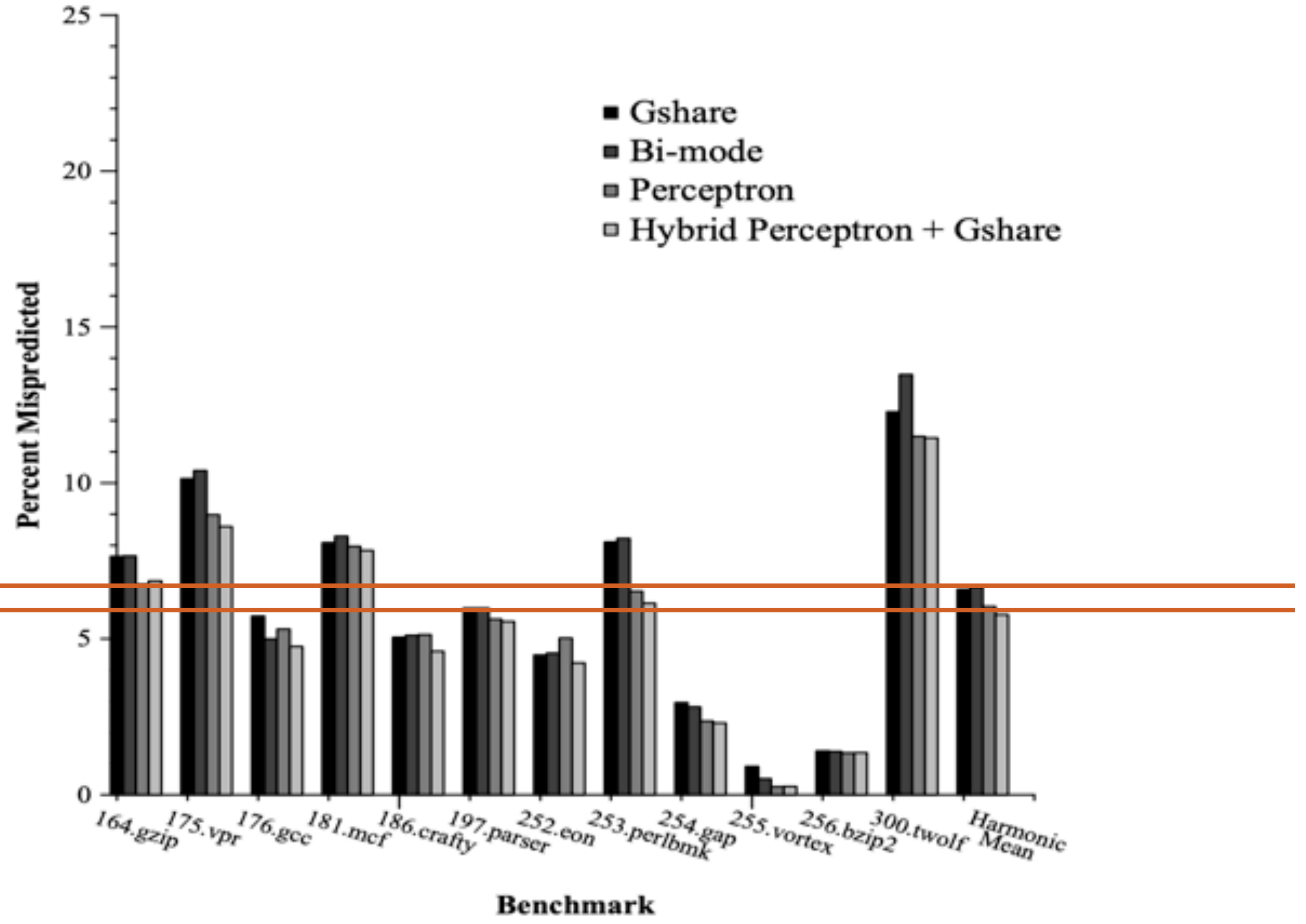
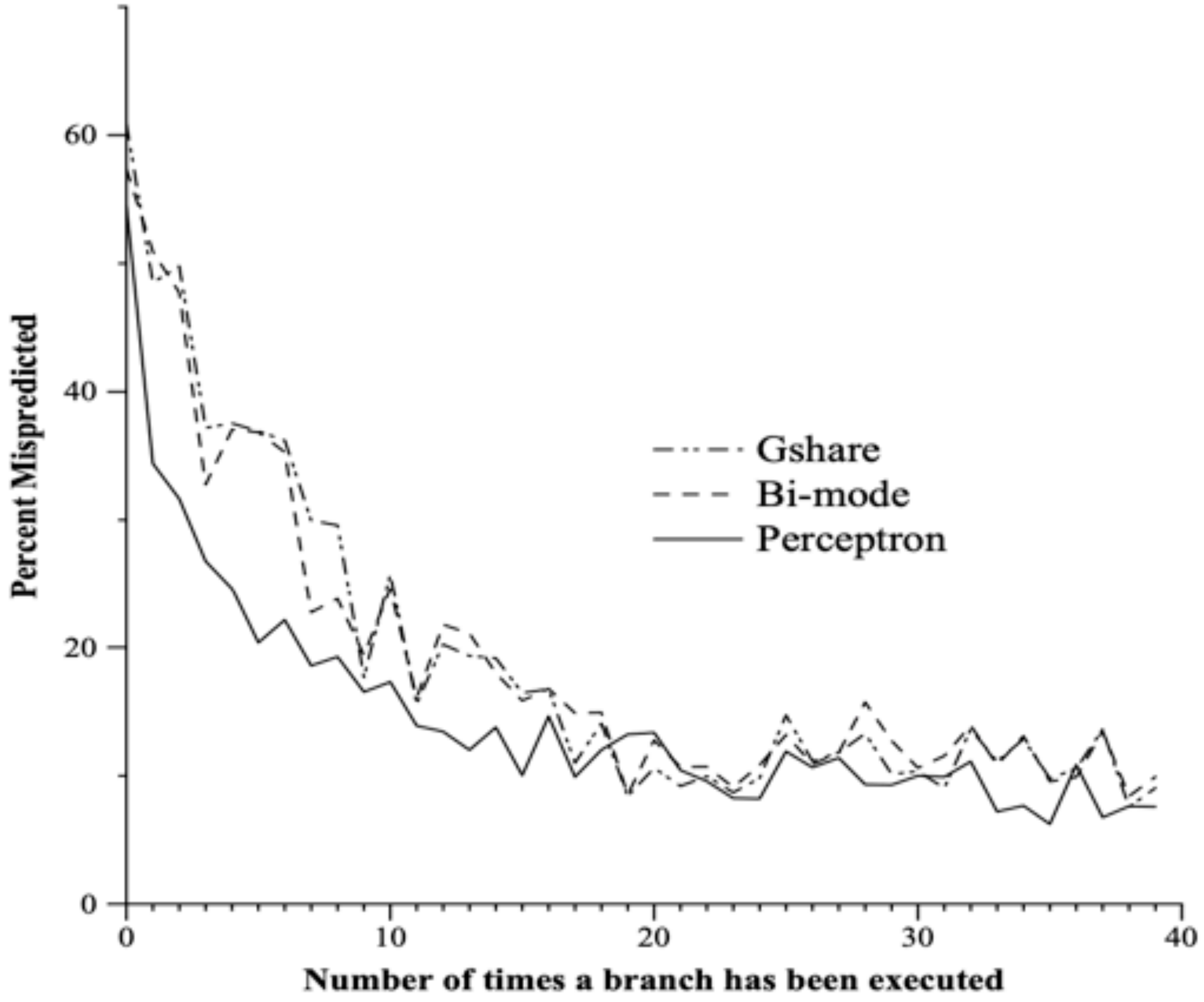


Figure 5: Misprediction Rates at a 16K budget. Gshare outperforms the perceptron predictor only on `186.crafty`. The hybrid predictor is consistently better than the PHT schemes.

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. $\boxed{0}$
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.

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.

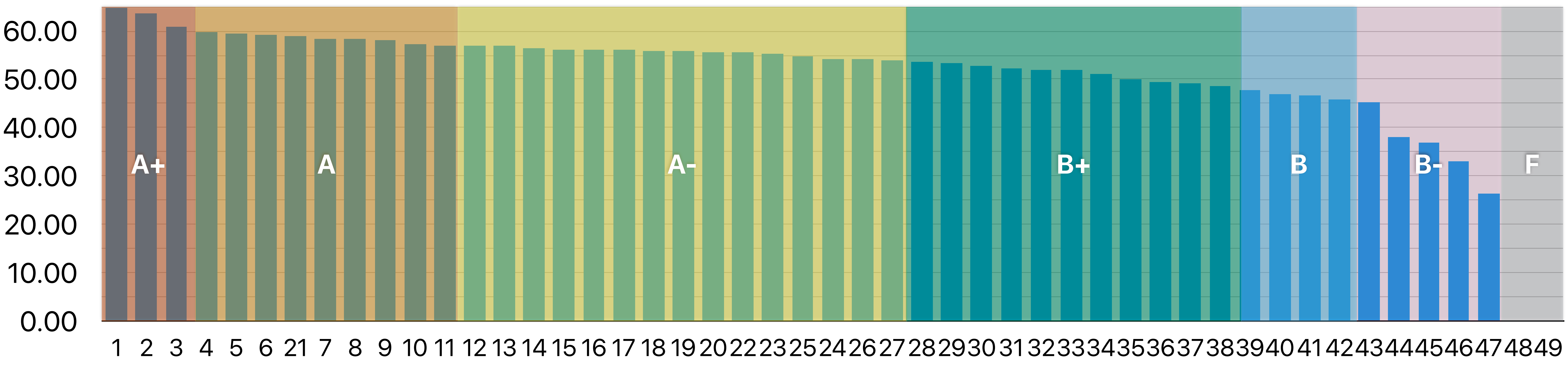
Takeaways: branch predictions

- 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 branch predictors!

Announcements

- Reading quiz #7 due next Tuesday
- Assignment #4 will be up on tonight and due on 11/21
- Your overall grade decides your final letter grade, not just the midterm. Midterm is only 25%
- Midterm and how are you doing so far
  - Midterm average is 69, Max is 100
  - Your overall grade decides your final letter grade, not just the midterm

Current "Total" in myGrades and "Projected" Letter Grades



Computer
Science &
Engineering

203

つづく