Modern Processor Design (III): I Just Can't Wait

Hung-Wei Tseng
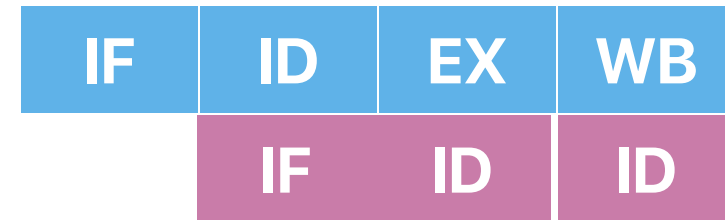
Recap: Pipelining

```
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
```

| IF | ID | EX | WB |

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

You must continuously fetch instructions!

After this point, we are completing an instruction each cycle!

t

2
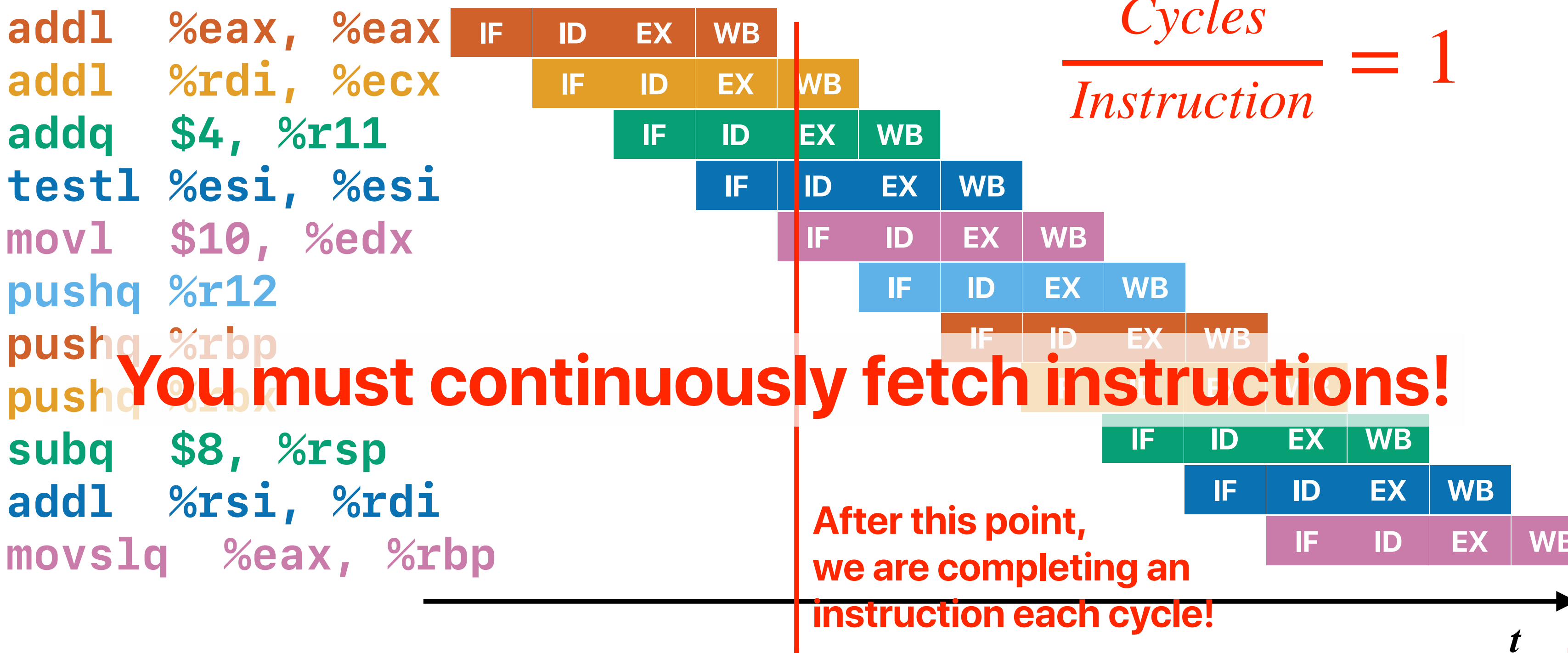
Control Hazard

① `cmpq %rdx, %rdi`
② `jne  .L3`
③ `ret`

We need the EX stage to calculate the address of .L3 if we are going to .L3

IF	ID	EX	WB

IF	ID	ID	EX

IF	ID	EX	WB

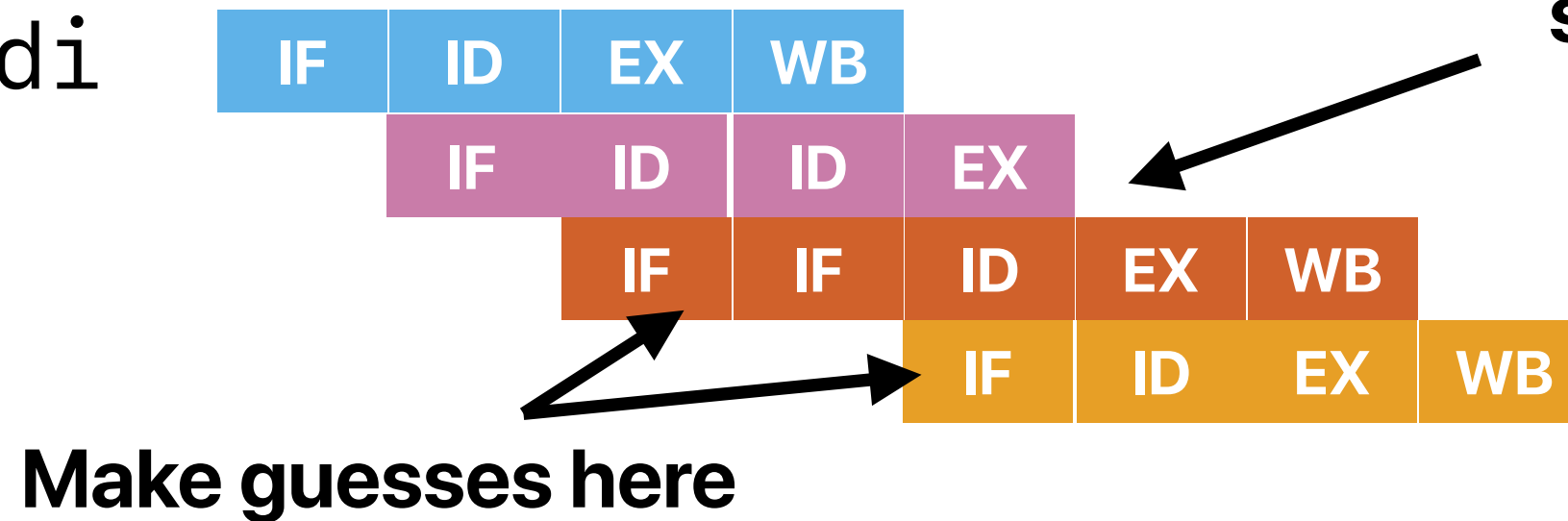**We cannot know if we should fetch "ret" or instruction at .L3 before cmpq finishes**

We wasted three fetching cycles — that means we will also have no output in 3 cycle

Microprocessor with a "branch predictor"
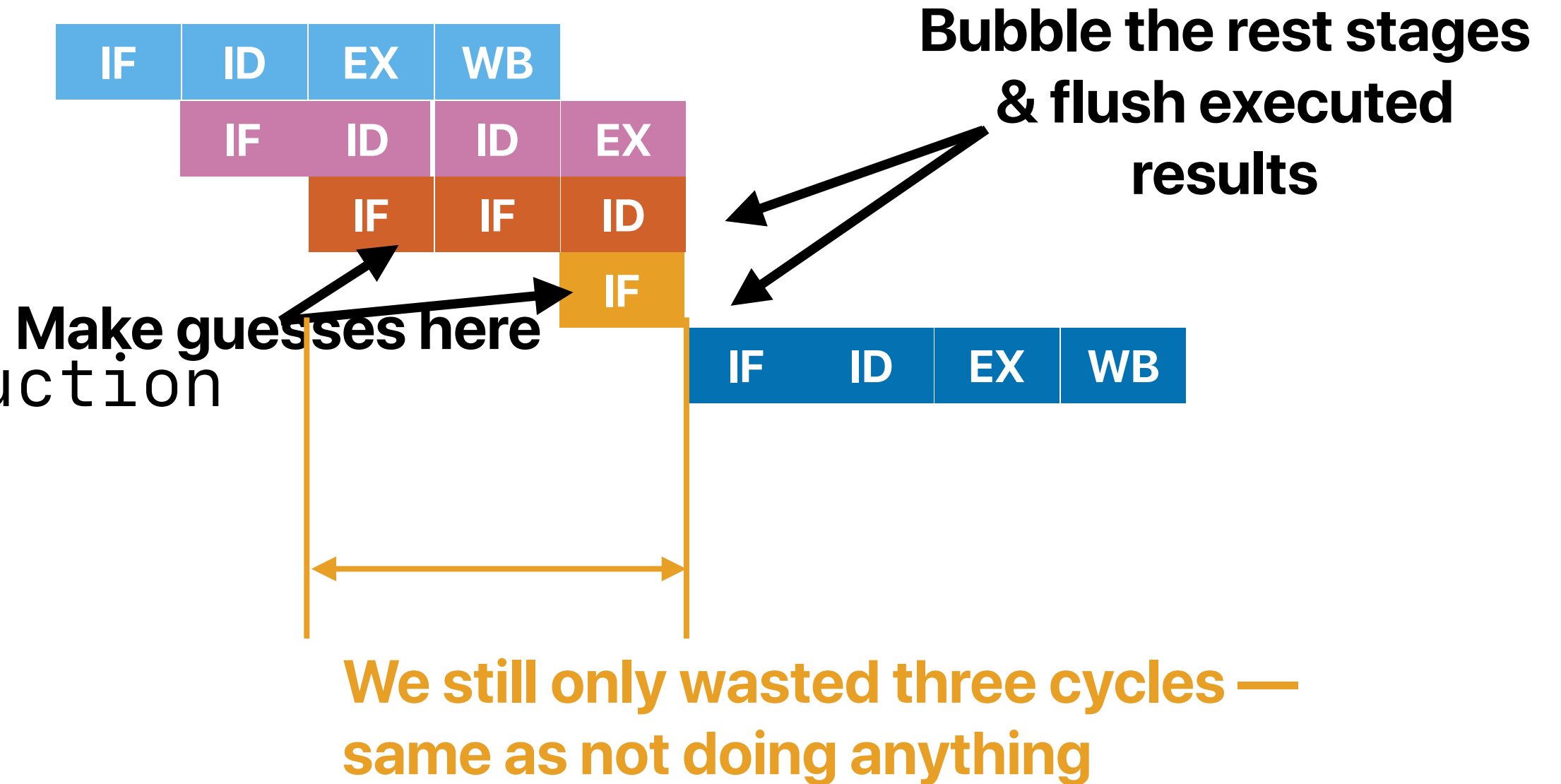
Branch Predictor

Program Counter → Instruction Fetch ← **Instructions** I-$

Instruction Decode

Registers

Branch/ Jump | Arithmetic Logical Units (ALU) | Complex Arithmetic Operations (Mul/div) | Memory Operations ↔ D-$

4
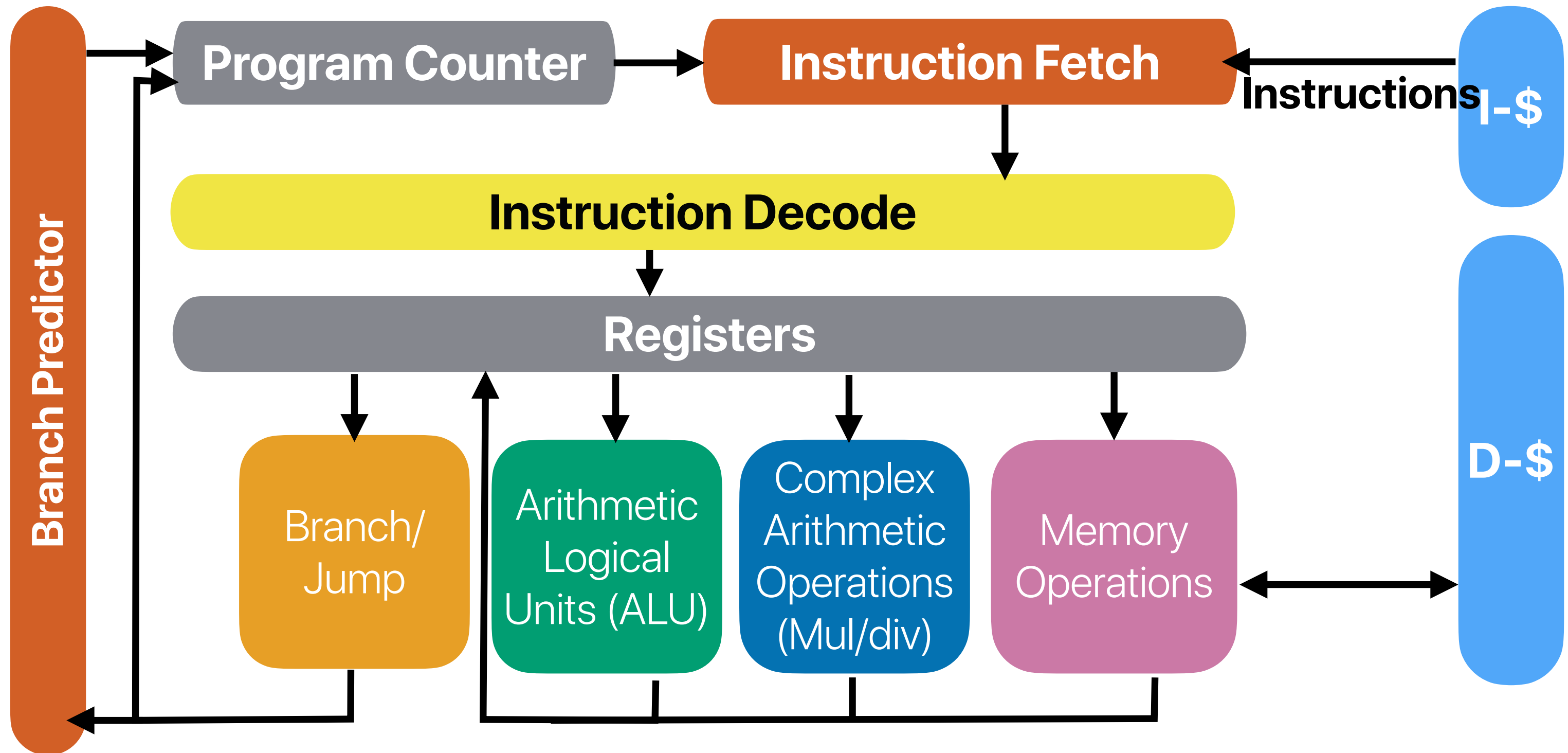
Recap: Branch Prediction

We can execute smoothly if we guess right!

① `cmpq %rdx, %rdi`
② `jne  .L3`
③ `ret`
④ `something …`

IF	ID	EX	WB

IF	ID	ID	EX

IF	IF	ID	EX	WB

IF	ID	EX	WB

Make guesses here

Recap: What if we are wrong?

① `cmpq %rdx, %rdi`
② `jne  .L3`
③ `ret`
④ `something …`
⑤ `the right instruction`

IF	ID	EX	WB

IF	ID	ID	EX

IF	IF	ID

IF

IF	ID	EX	WB

Make guesses here

Bubble the rest stages & flush executed results

We still only wasted three cycles — same as not doing anything
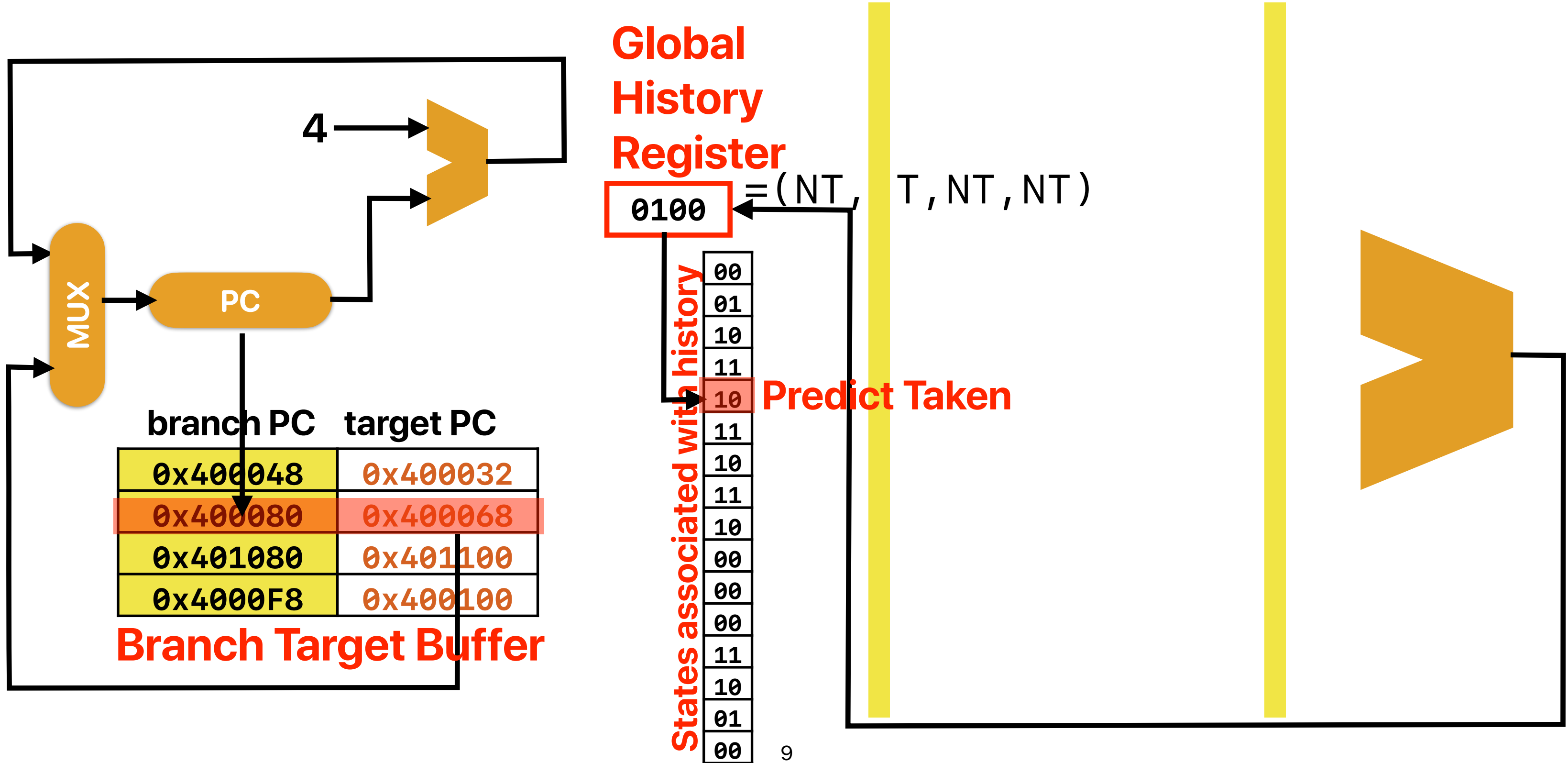
Recap: branch predictors

- 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**
- Global
  - Predictors do not keep states for each branch instructions
  - Predictors do not rely on the outcome of single branch instructions to predict outcome
  - Example: 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).
- Local
  - Predictors keep states for each branch instructions
  - Predictors rely on the outcome of single branch instructions to predict outcome
- n-bit — the number of bits in the state machine

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

8
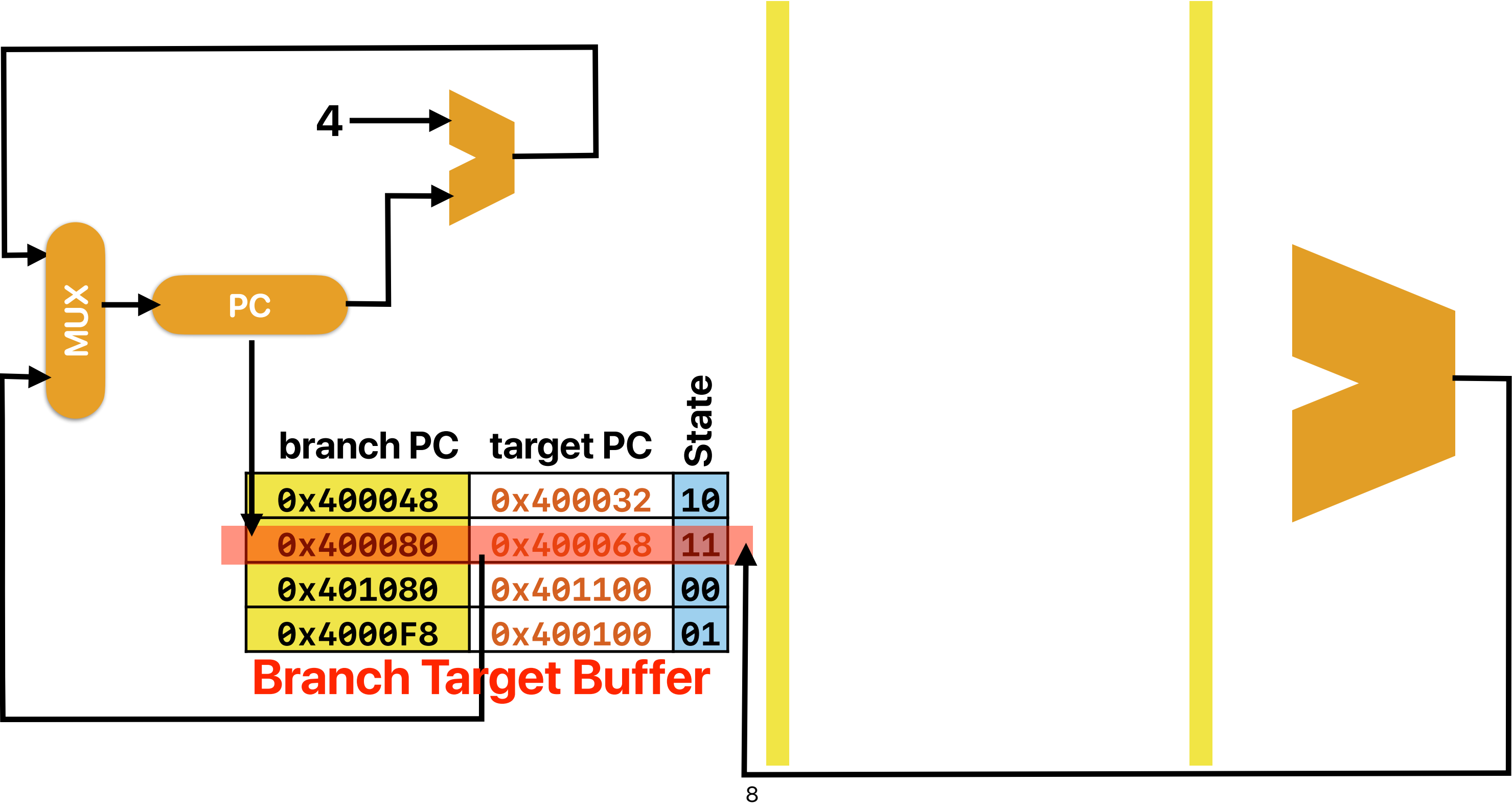
Recap: Global history (GH) predictor

Global History Register

$= (NT, T, NT, NT)$

0100

States associated with history

00	
01	
10	
11	
10	**Predict Taken**
11	
10	
11	
10	
00	
00	
00	
11	
10	
01	
00	

4

MUX

PC

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

Branch Target Buffer

Recap: 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

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

about the same

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

✓

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

L could be better

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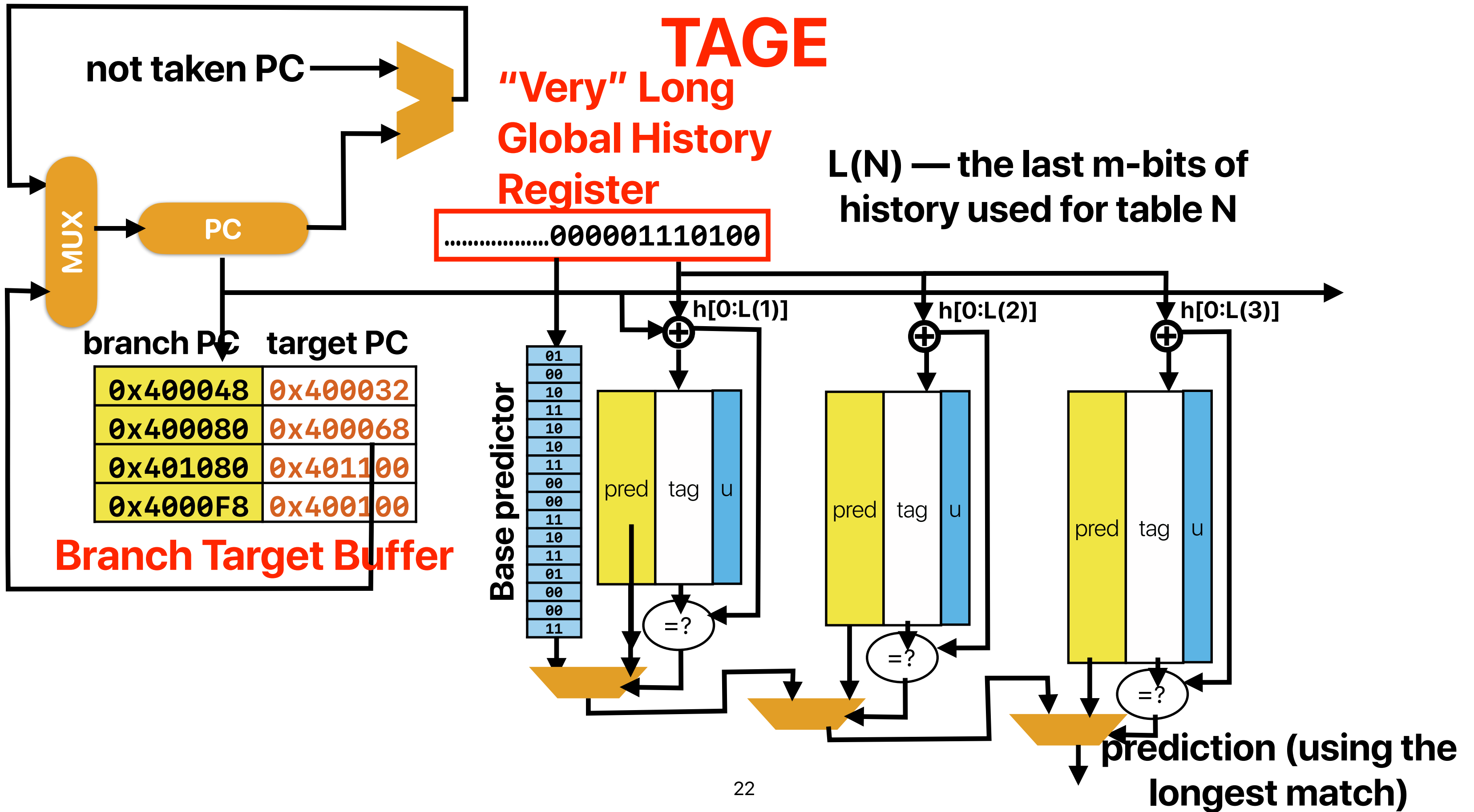
A. 0

B. 1

C. 2

D. 3

E. 4

Outline

- Dynamic branch prediction
- Data hazards
  - Stalls
  - Data forwarding

Hybrid predictors

Tournament Predictor
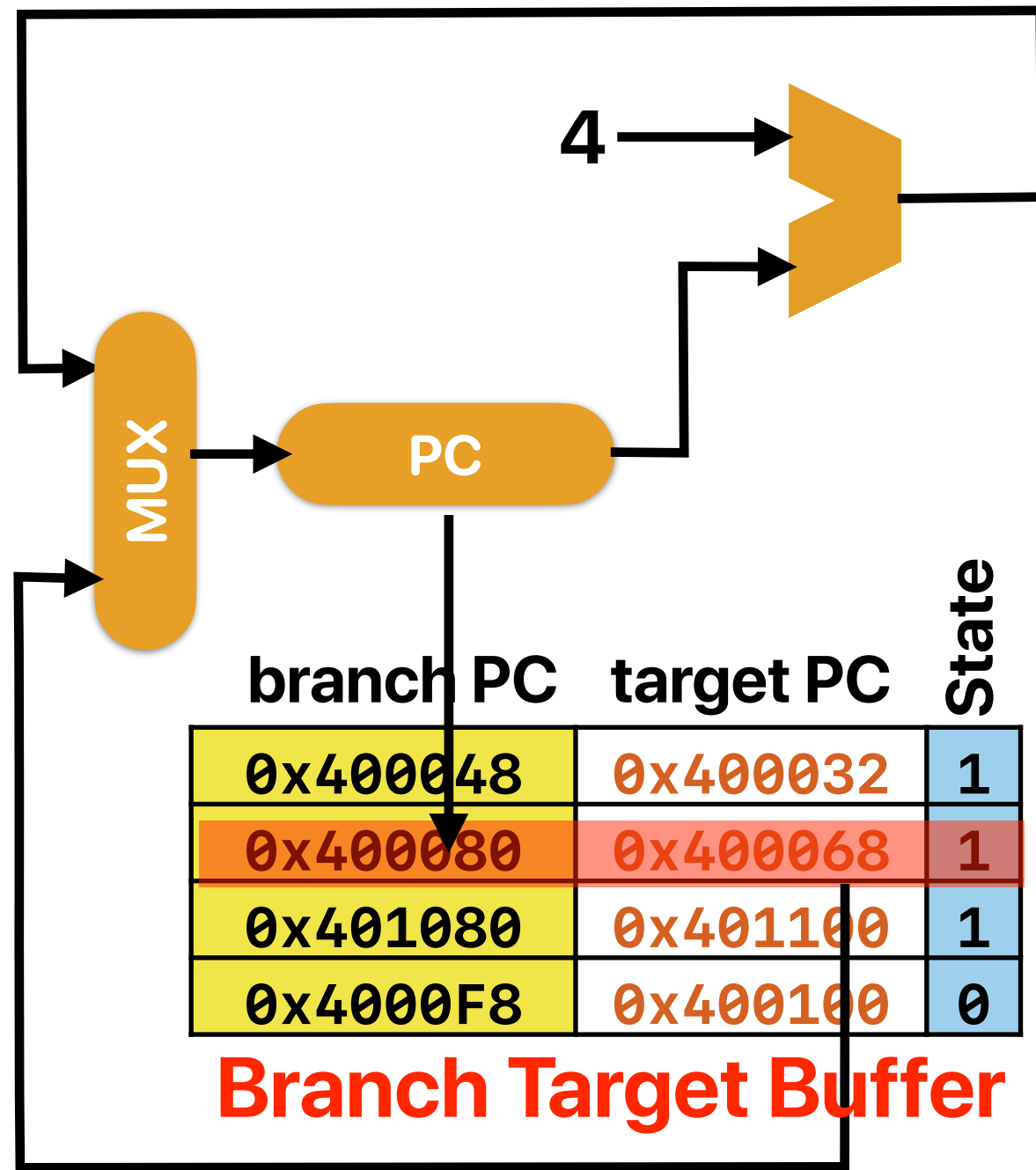
Global History Register

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
00
00
00
11
10
01
00

4

MUX

PC

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

Branch Target Buffer

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)?

about the same   **about the same**   **L could be better**

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

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

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

```
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

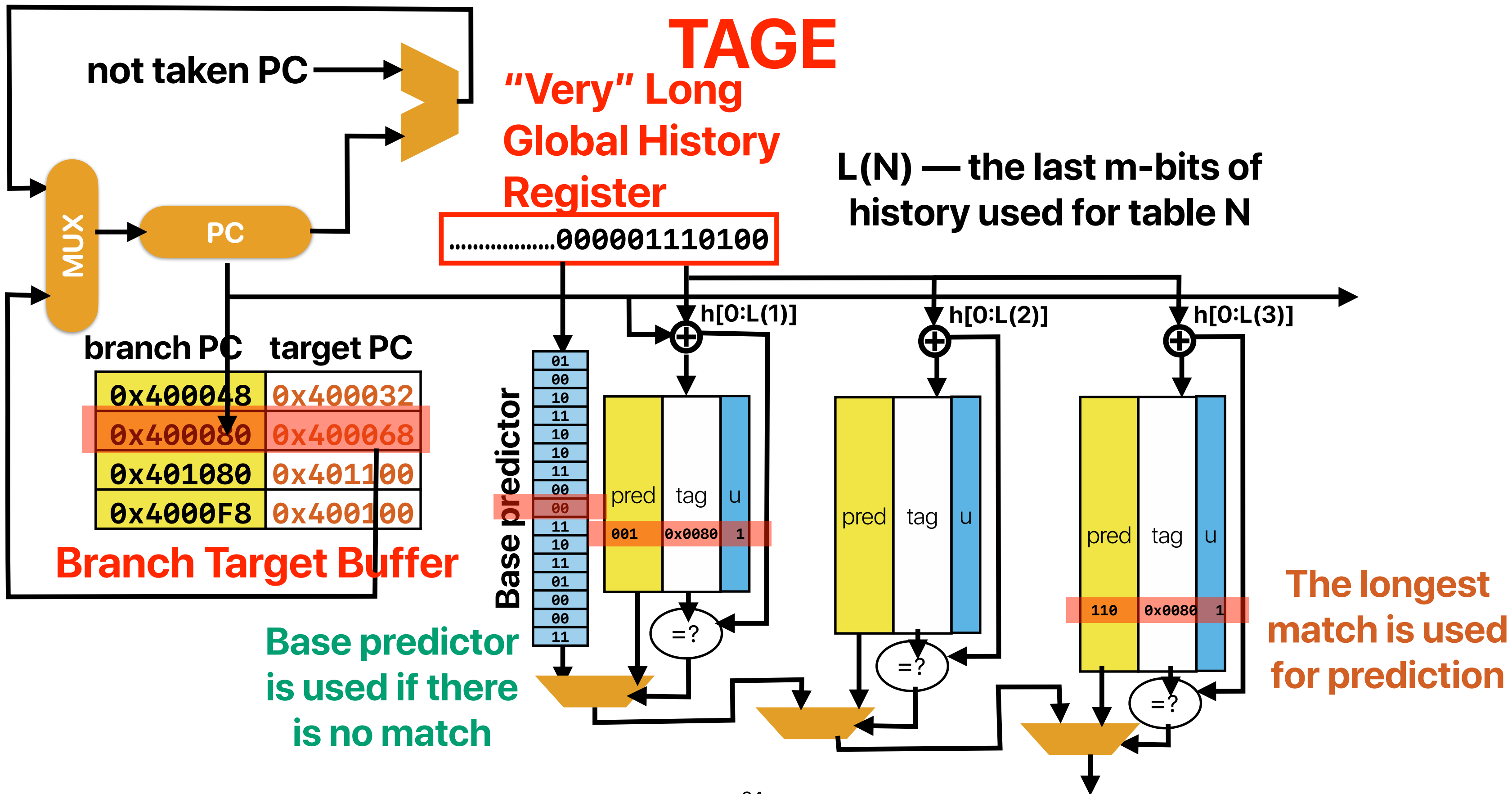"Very" Long Global History Register

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

……………000001110100

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

Branch Target Buffer

not taken PC

MUX

PC

Base predictor

h[0:L(1)]   h[0:L(2)]   h[0:L(3)]

pred tag u   pred tag u   pred tag u

=?   =?   =?

prediction (using the longest match)

22

What's inside each table?

pred (3-bit counter)	tag (partial branch PC)	u (usefulness)
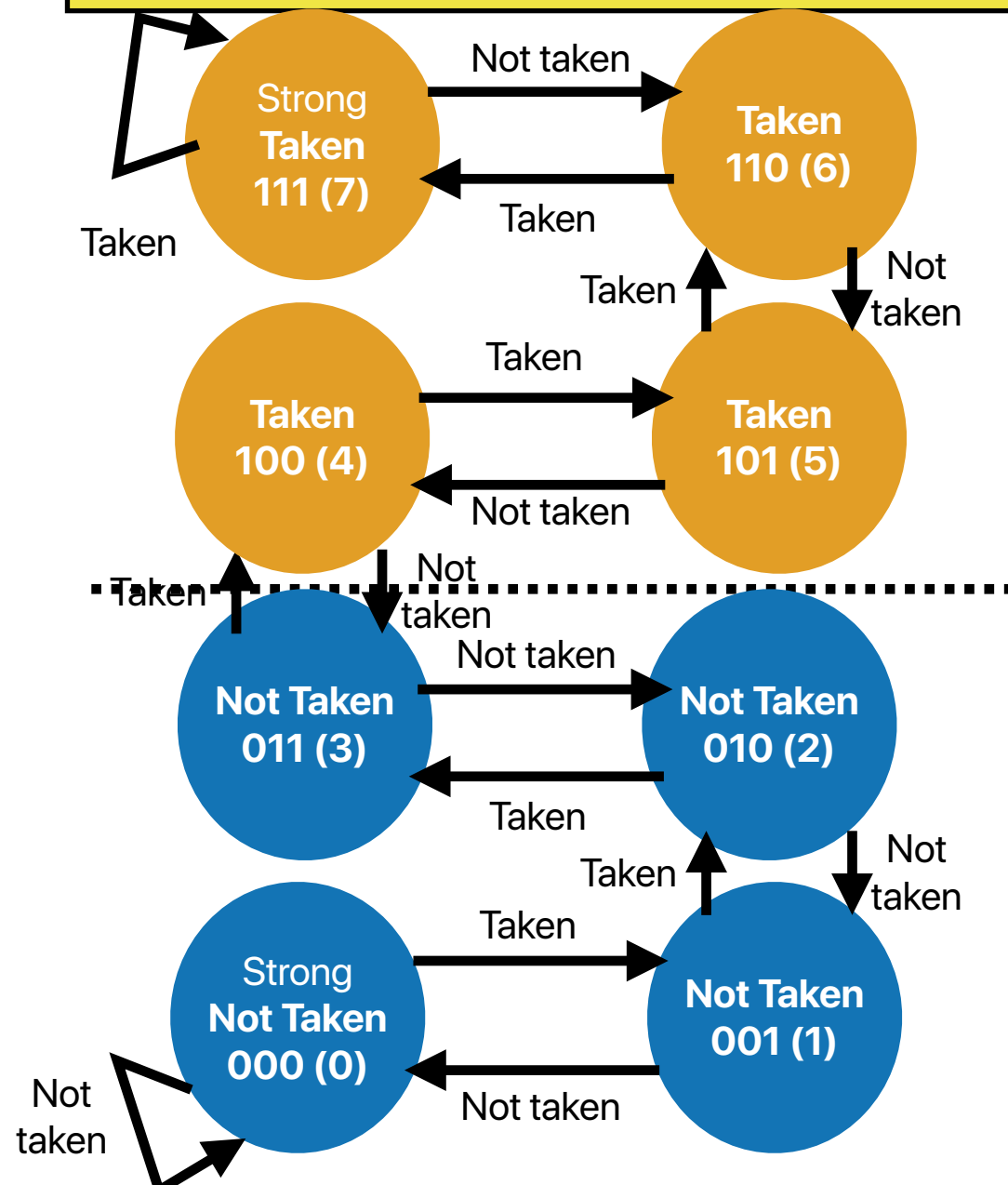
$$if\ prediction(alt\_predictor) \neq prediction(pred):$$

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

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

23

TAGE

"Very" Long Global History Register

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

not taken PC

MUX

PC

…………………000001110100

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
001	0x0080	1

=?

pred	tag	u

=?

pred	tag	u
110	0x0080	1

=?

Base predictor is used if there is no match

The longest match is used for prediction

24

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
- The formation of the problem
{branch address, branch history, some other attributes.. } —
{"taken", "not taken"}
- 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

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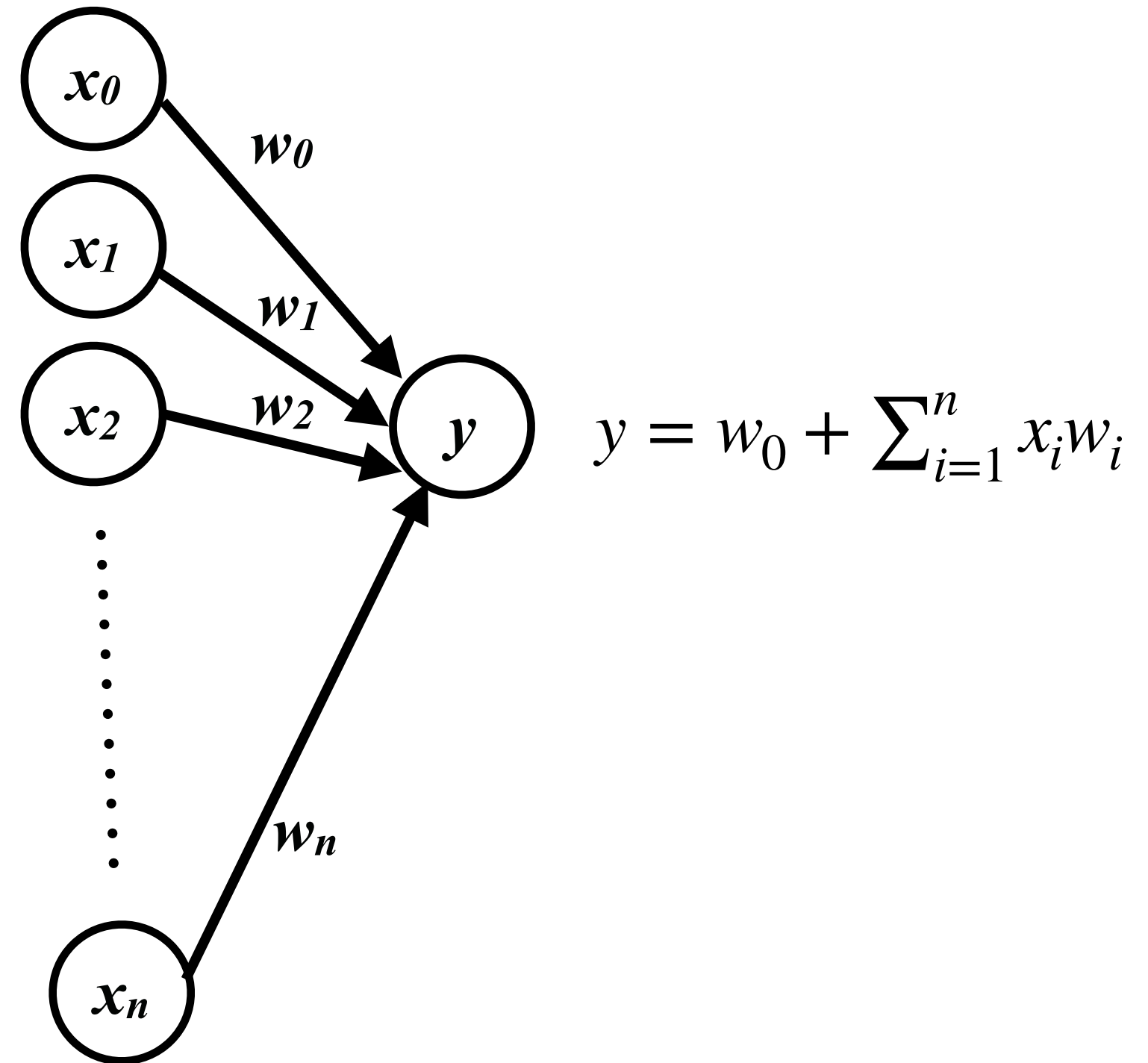
Training Algorithm

$x_{1..n}$ 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| ≤ θ or ((y ≥ 0) ≠ t) then
     for each 0 ≤ i ≤ n in parallel
          if t = xᵢ then
               wᵢ := wᵢ + 1
          else
               wᵢ := wᵢ - 1
          end if
     end for
end if
```

Predictor Organization

Global History Register

Table of Perceptrons/ Weight Vectors

Branch Target Buffer

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

about the same

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

✔

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

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

Perceptron can discount this when predicting while

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 characteristics of TAGE and Perceptron justify such a design decision?

  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 ███████████ L2 predictor while keeping perceptron as the L1 predictor for ████████████████.

32

Area efficiency between TAGE and Perceptron
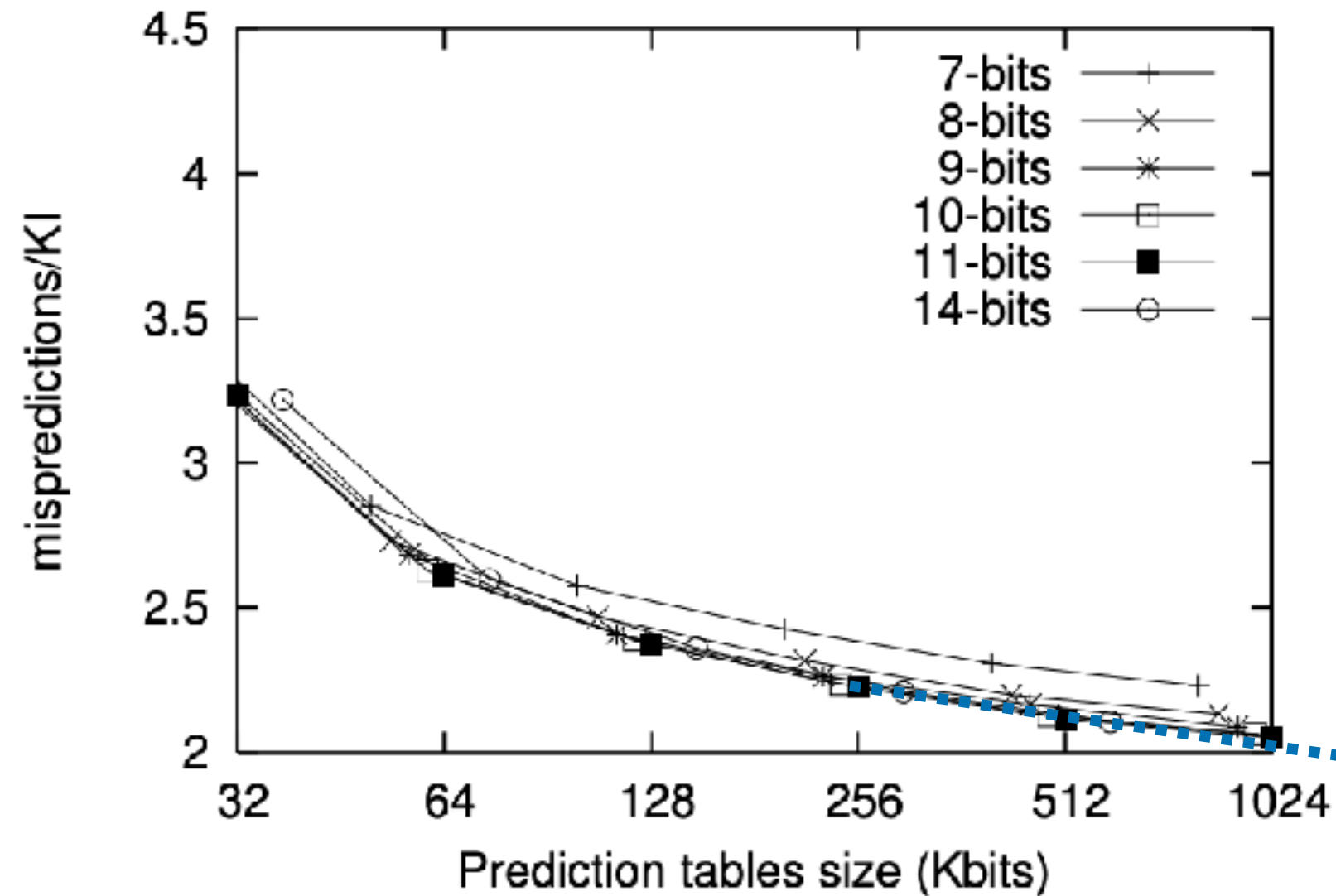
TAGE
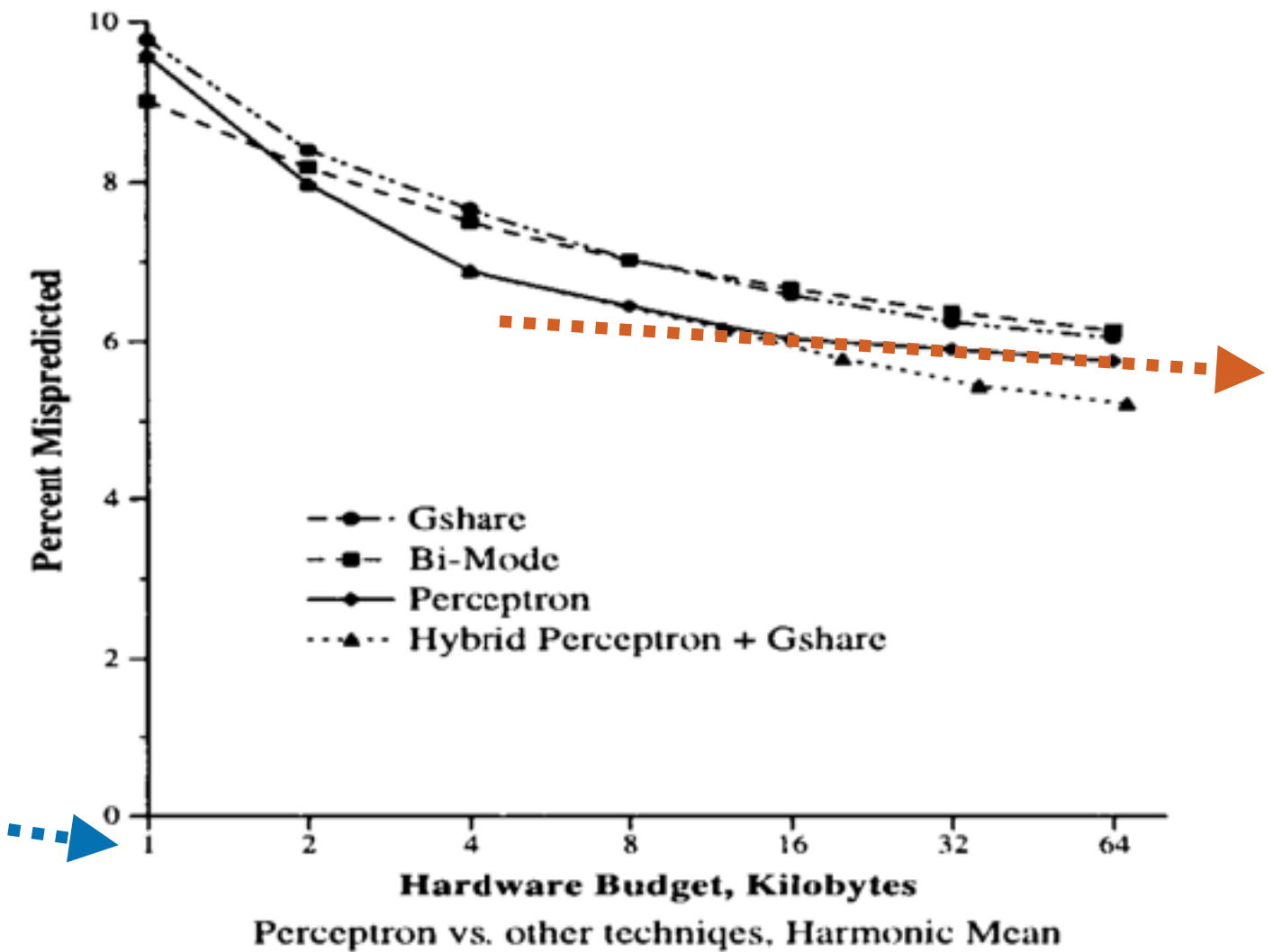
Perceptron

How good is prediction using perceptrons?
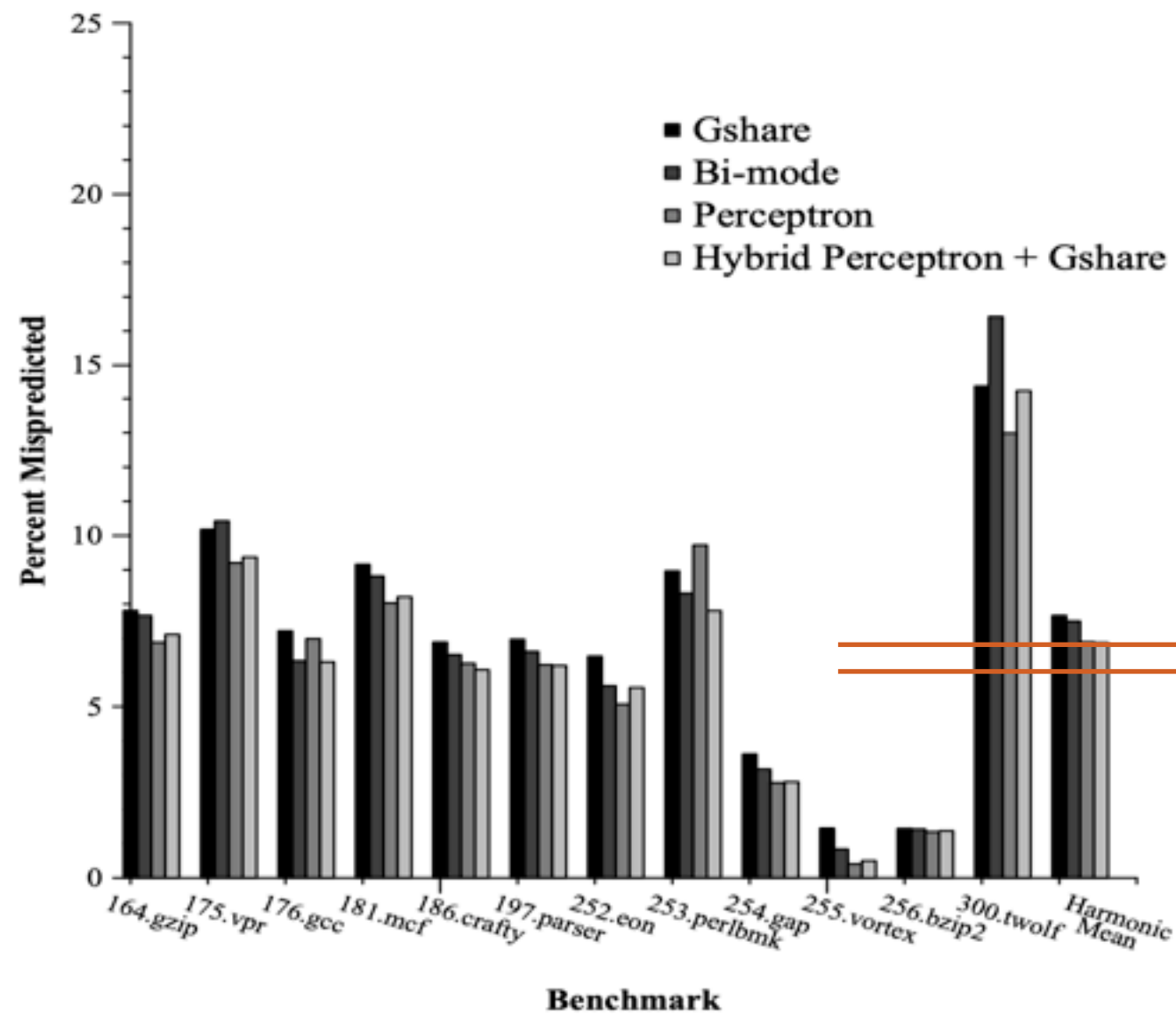
Figure 4: Misprediction Rates at a 4K budget. The perceptron predictor has a lower misprediction rate than *gshare* for all benchmarks except for 186.crafty and 197.parser.
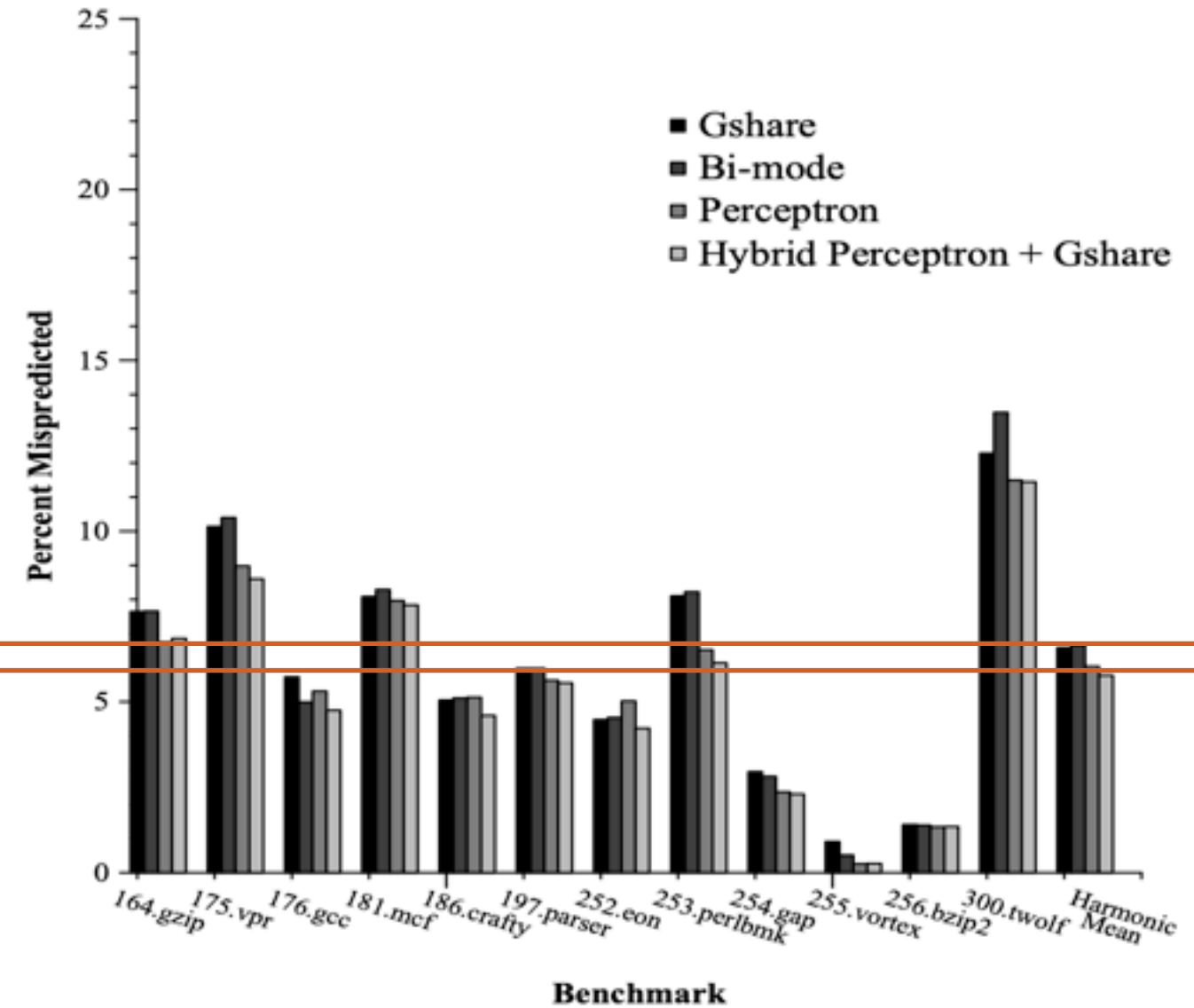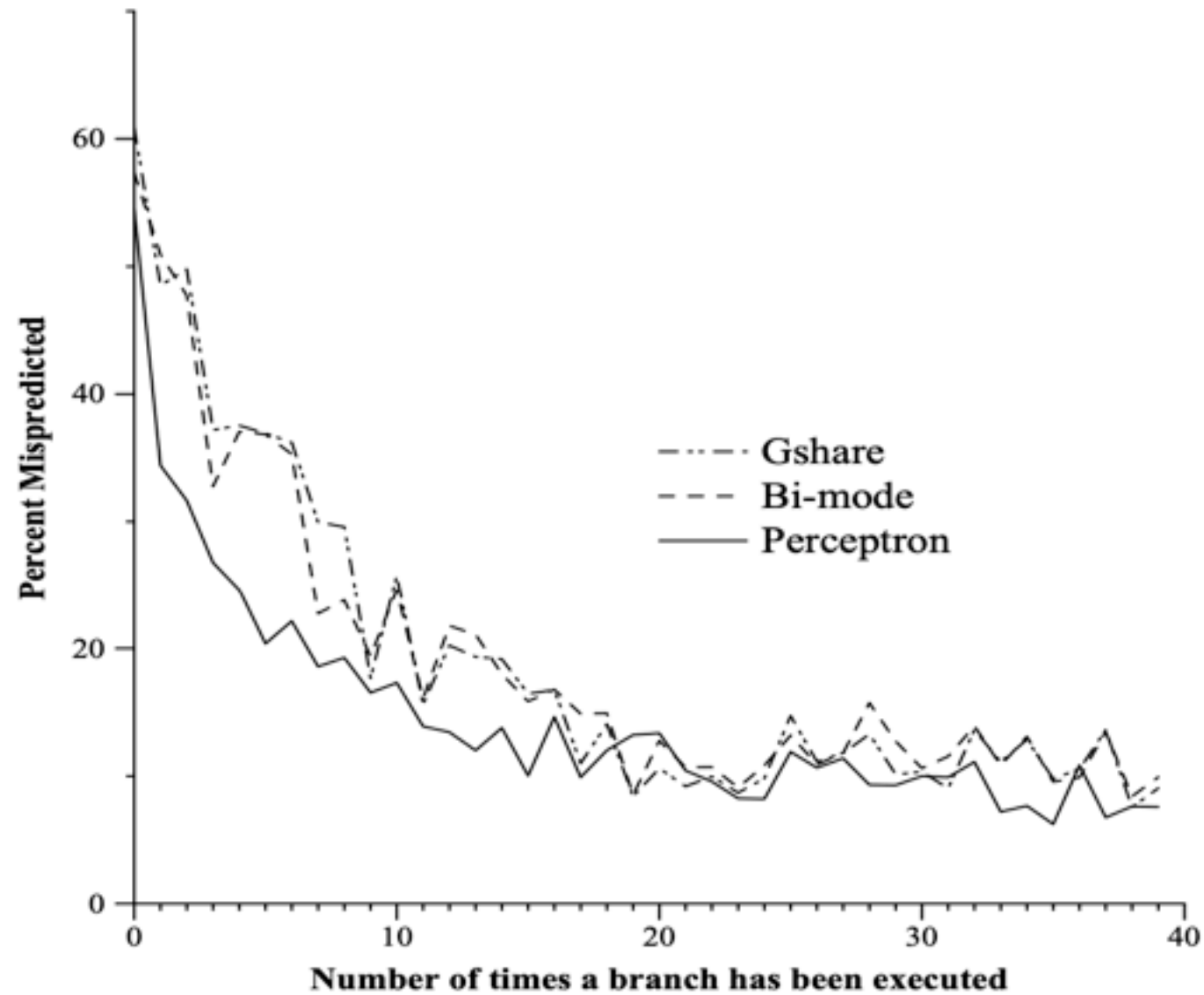
Figure 5: Misprediction Rates at a 16K budget. Gshare outperforms the perceptron predictor only on 186.crafty. The hybrid predictor is consistently better than the PHT schemes.

History/training for perceptrons

Hardware budget	History Length		
in kilobytes	*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.

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 characteristics of TAGE and Perceptron justify such a design decision?

   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

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

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.

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 and programming

Demo revisited

```cpp
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!!!

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

Demo revisited

- Why the performance is better when option is not "0"

  ① The amount of dynamic instructions needs to execute is a lot smaller

  ② The amount of branch instructions to execute is smaller

  ③ The amount of branch mis-predictions is smaller
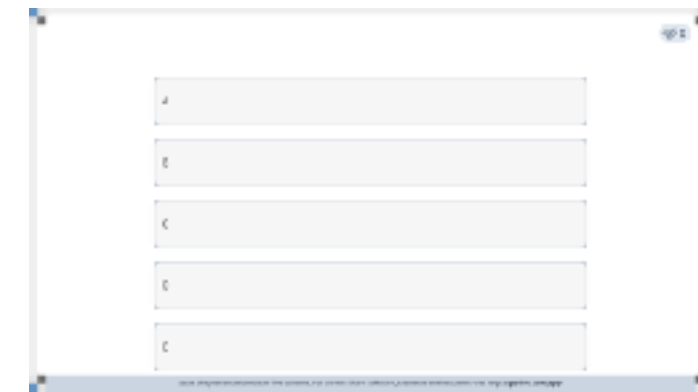
  ④ The amount of data accesses is smaller

A. 0

B. 1

C. 2

D. 3

E. 4

```cpp
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 ++;
    }
}
```

44

Demo revisited

- Why the performance is better when option is not "0"
  - ① The amount of dynamic instructions needs to execute is a lot smaller
  - ② The amount of branch instructions to execute is smaller
  - ✓③ The amount of branch mis-predictions is smaller
  - ④ The amount of data accesses is smaller

A. 0

B. 1

C. 2

D. 3

E. 4

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

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

Recap: Which swap is faster?

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

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

- Both version A and B swaps content pointed by a and b correctly. Which version of code would have better performance?
  A. Version A
  B. Version B
  C. They are about the same (sometimes A is faster, sometimes B is)

50

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
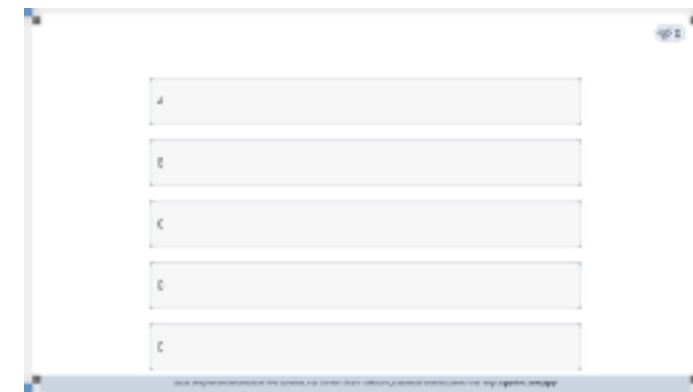 - May result in data hazard if the later instruction that consumes the result is still in the pipeline

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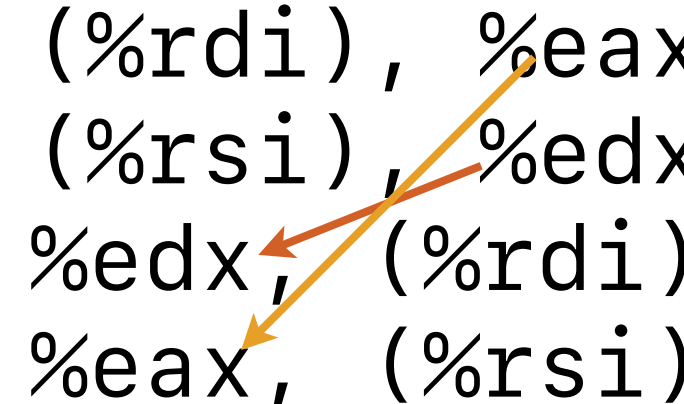
```
int temp = *a;
*a = *b;
*b = temp;
```

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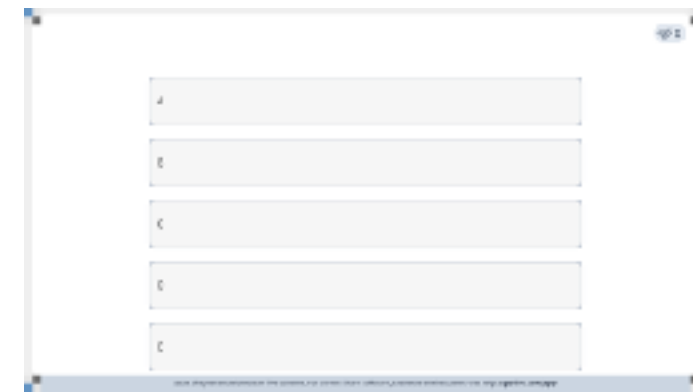
A. 1
B. 2
C. 3
D. 4
E. 5

57

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

58

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

62

Data hazards?

- How many pairs of data dependences in the following x86 instructions will result in data hazards if a memory operation (assume 100% cache hit rate) takes 4 cycles?

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

A. 0

B. 1

C. 2

D. 3

E. 4

Data hazards?

- How many pairs of data dependences in the following x86 instructions will result in data hazards if a memory operation (assume 100% cache hit rate) takes 4 cycles?
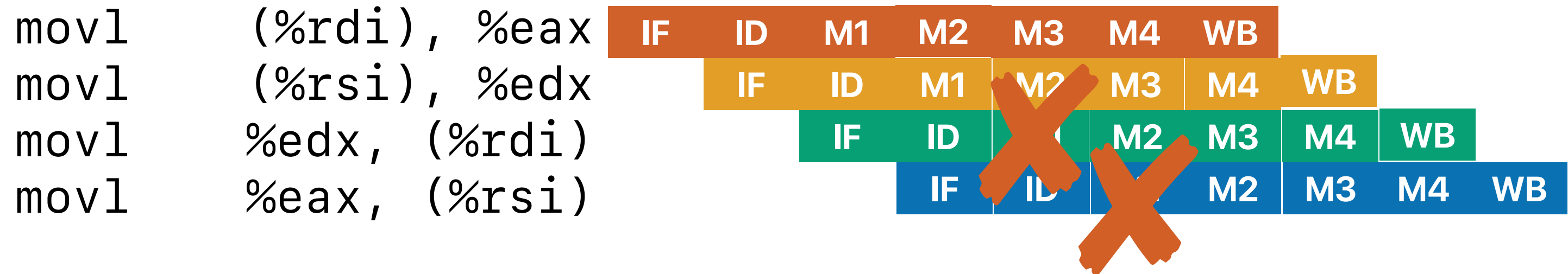
```
movl    (%rdi), %eax
movl    (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```


A. 0

B. 1

C. 2

D. 3

E. 4

Data hazards

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

	IF	ID	ALU/BR/M1	M2	M3	M4	WB
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							

%eax does not have our desired value

%edx does not have our desired value

Computer
Science &
Engineering

つづく