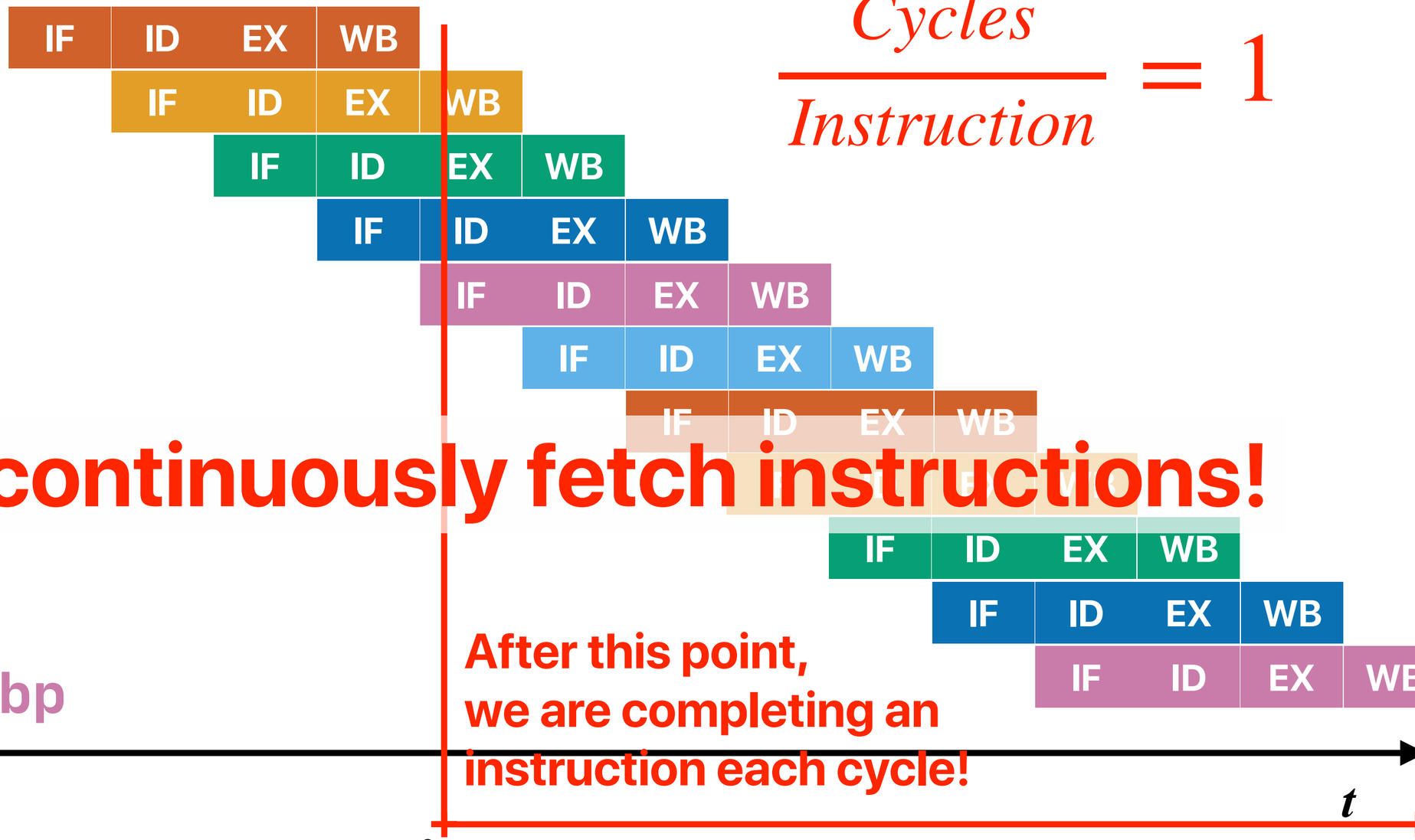
Modern Processor Design (III): Come to your Senses

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
```

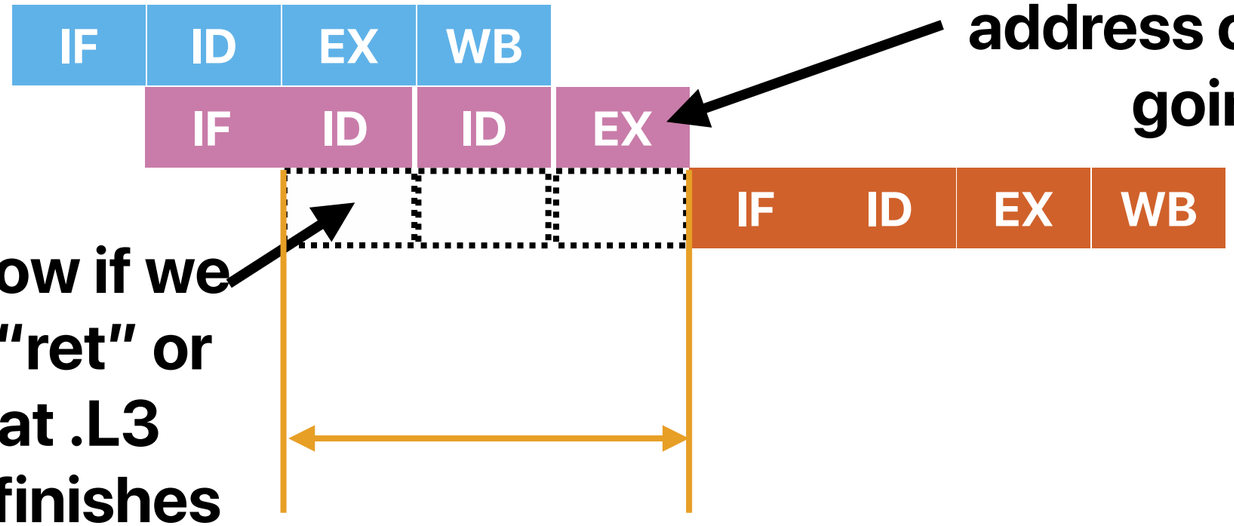


$$\frac{Cycles}{Instruction} = 1$$

You must continuously fetch instructions!

Control Hazard

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

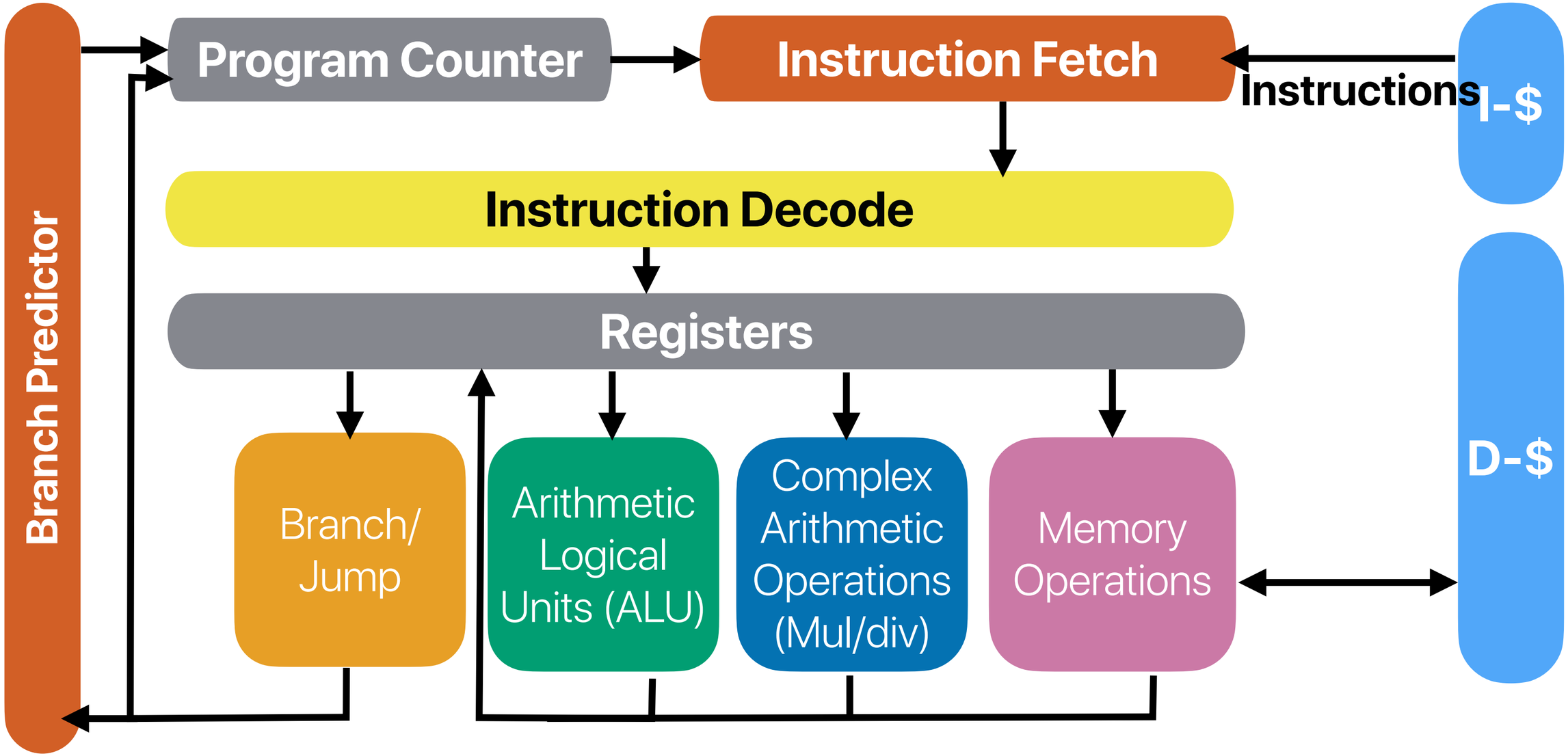


We need the EX stage to calculate the address of .L3 if we are going to .L3

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

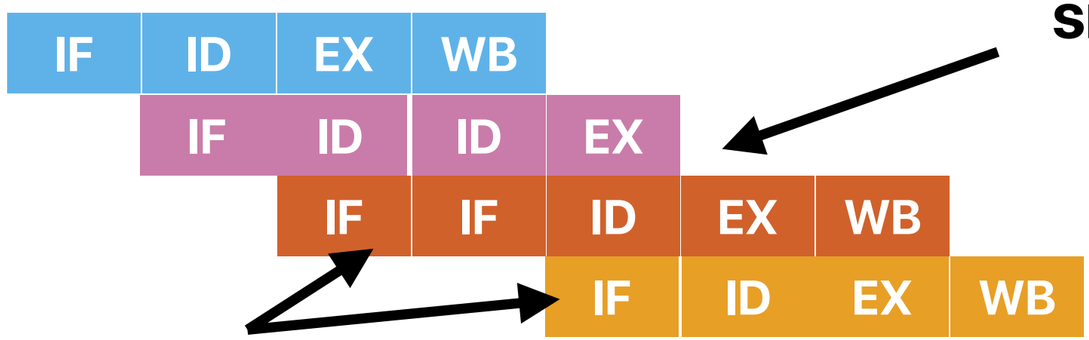
We wasted three fetching cycles —
that means we will also have no output in 3 cycle

Microprocessor with a “branch predictor”



Recap: Branch Prediction

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

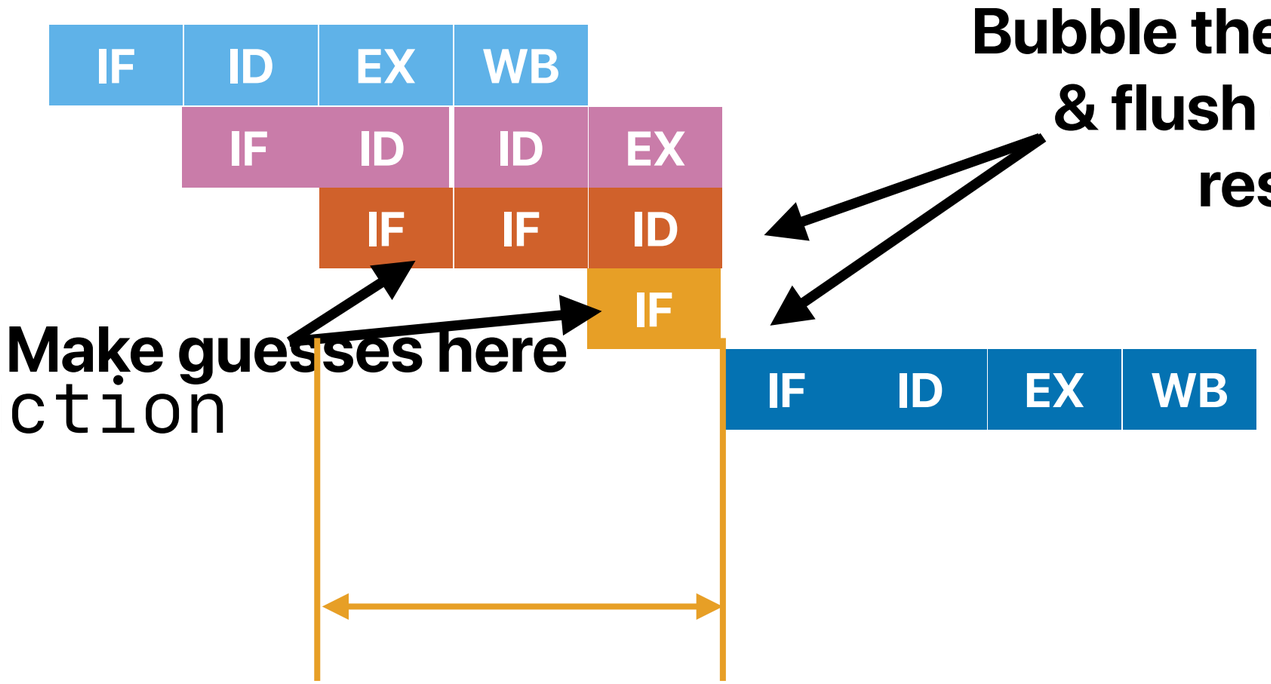


We can execute smoothly if we guess right!

Make guesses here

Recap: What if we are wrong?

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



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

Demo revisited: evaluating the cost of mis-predicted branches

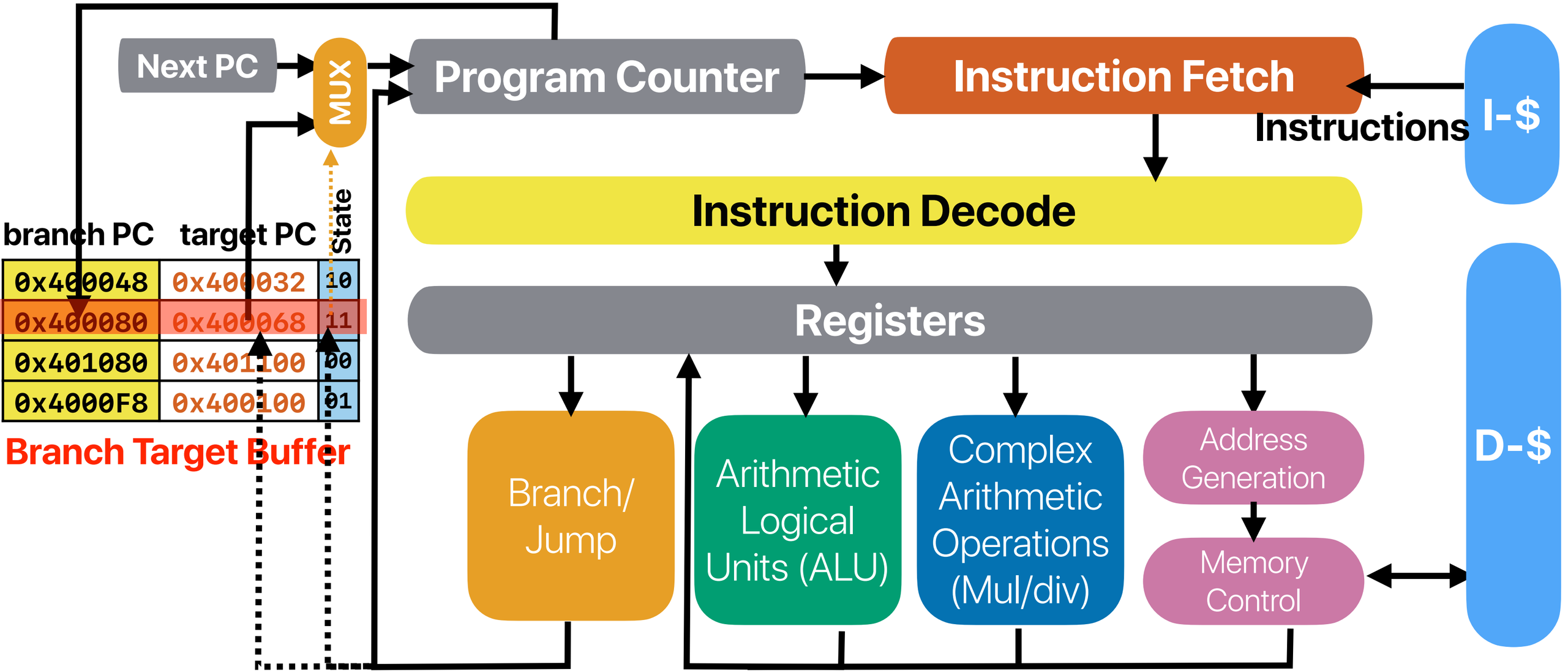
- Compare the number of mis-predictions
- Calculate the difference of cycles
- We can get the “average CPI” of a mis-prediction!

34 cycles!!!

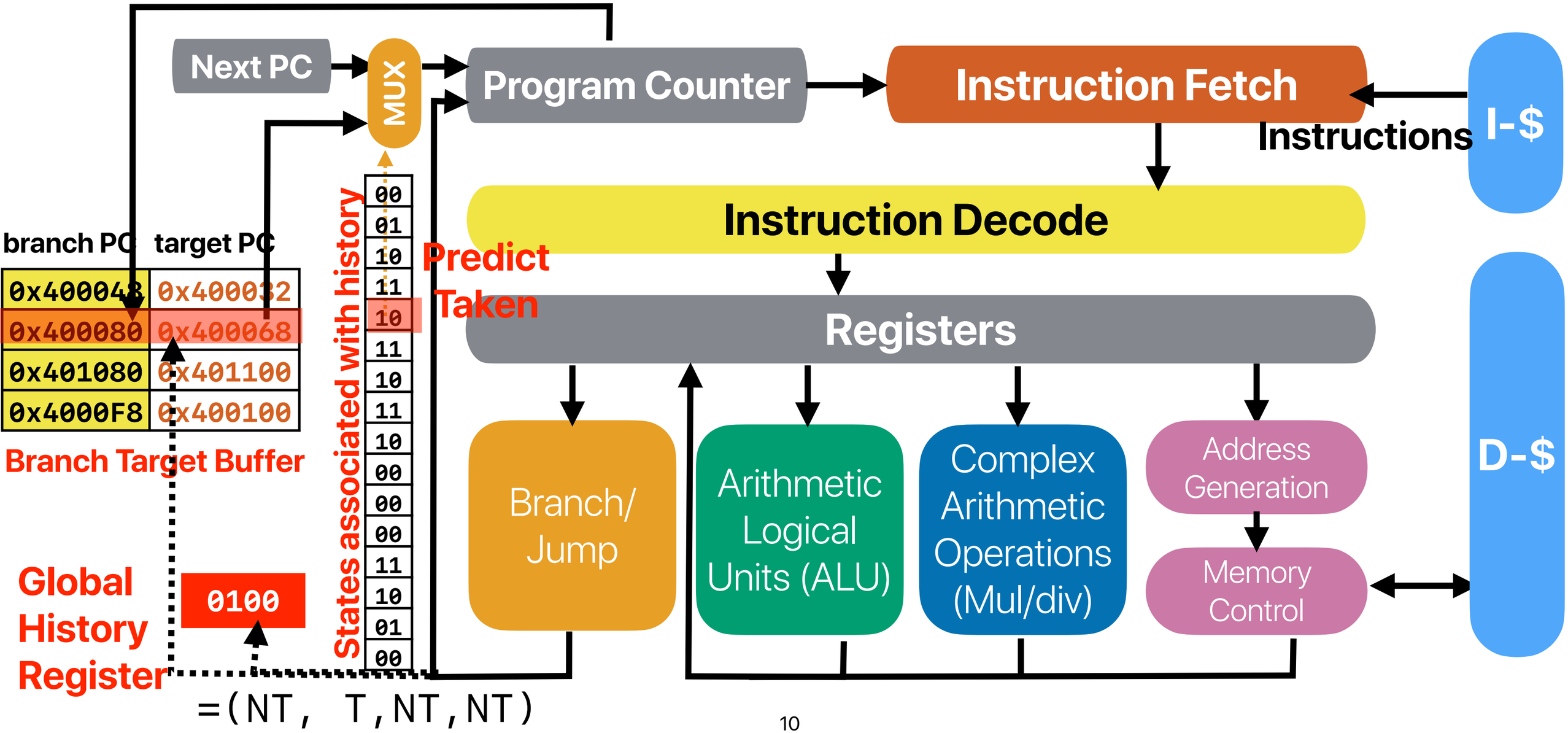
Recap: branch predictors

- 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 2-bit local predictor



Detail of a two-level global predictor



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
```

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

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

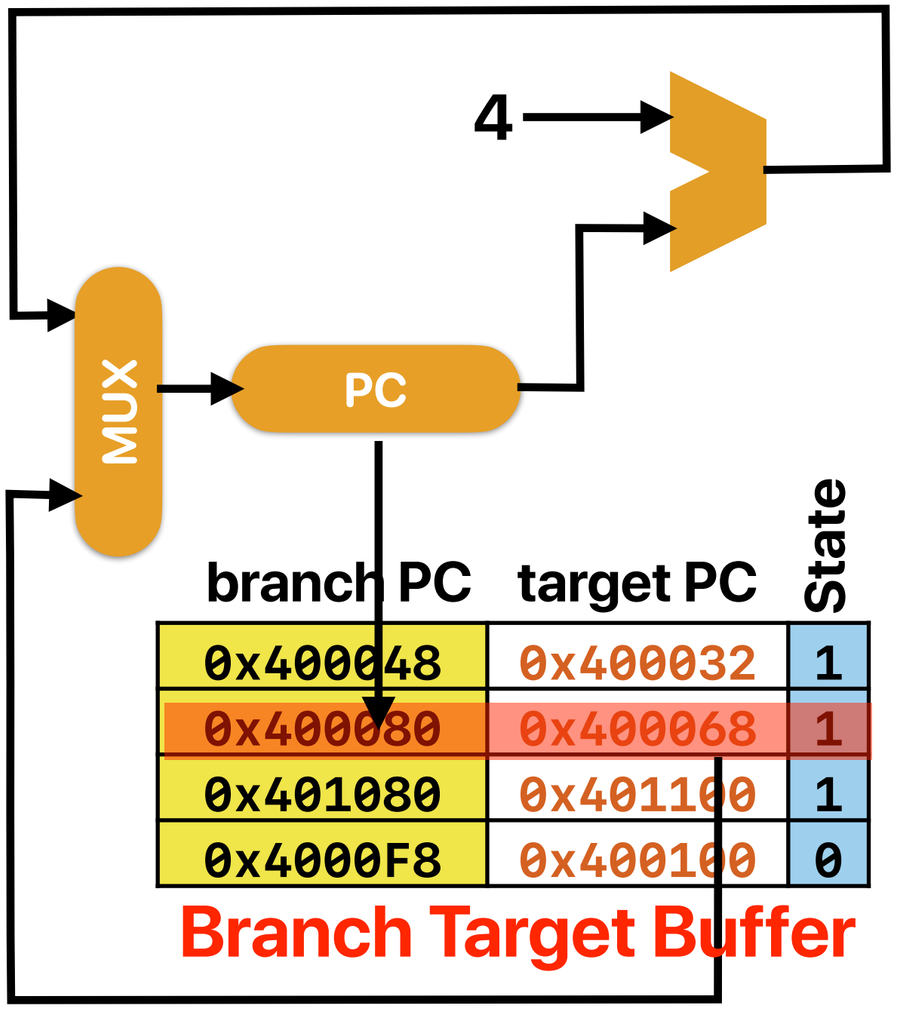
E. 4

Outline

- Dynamic branch prediction (cont.)
  - Perceptron
  - TAGE
- Programming on processors with advanced branch predictors

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

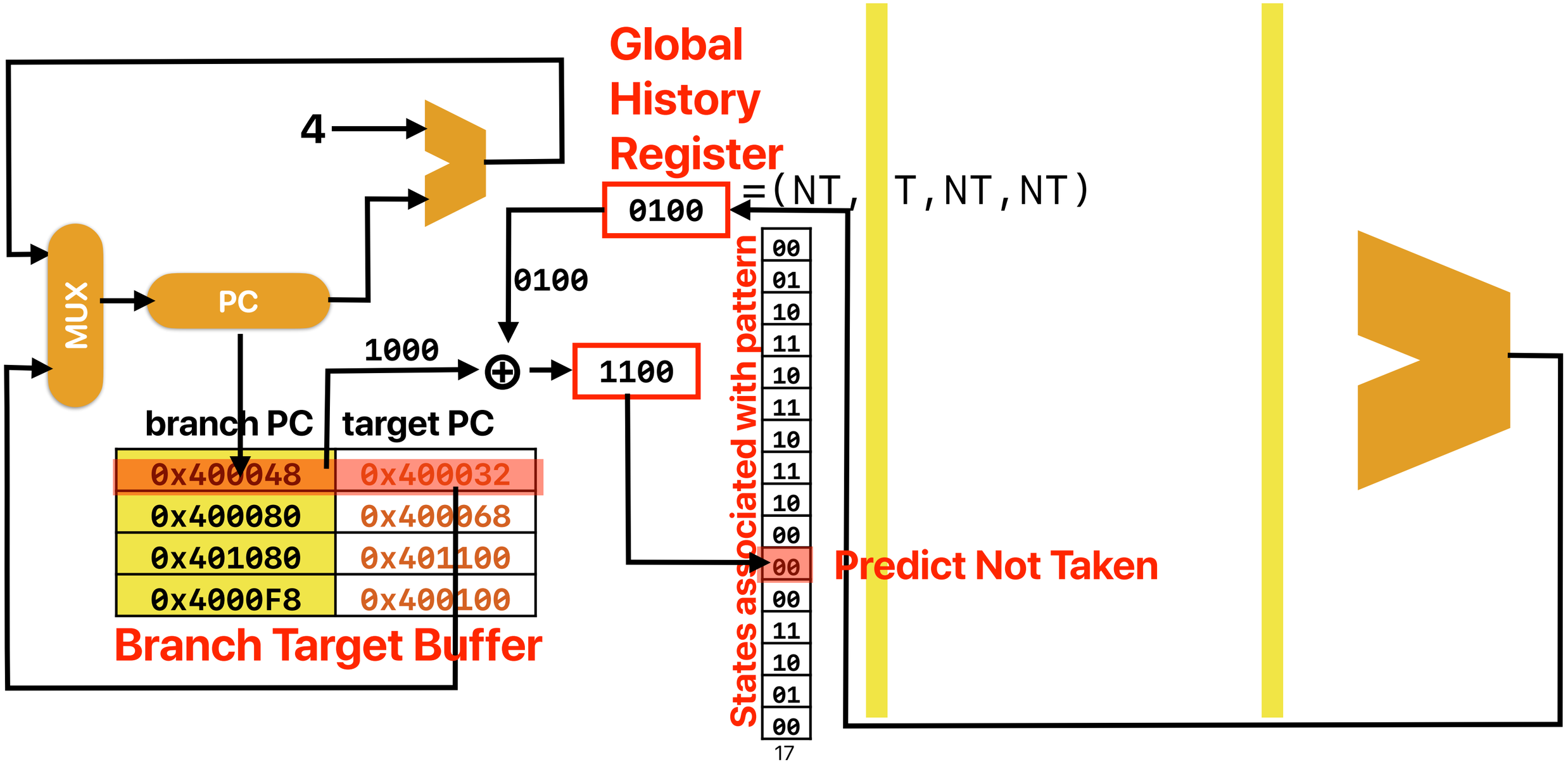
States associated with history

00
01
10
11
10
11
10
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

```
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);
```

about the same

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)
```

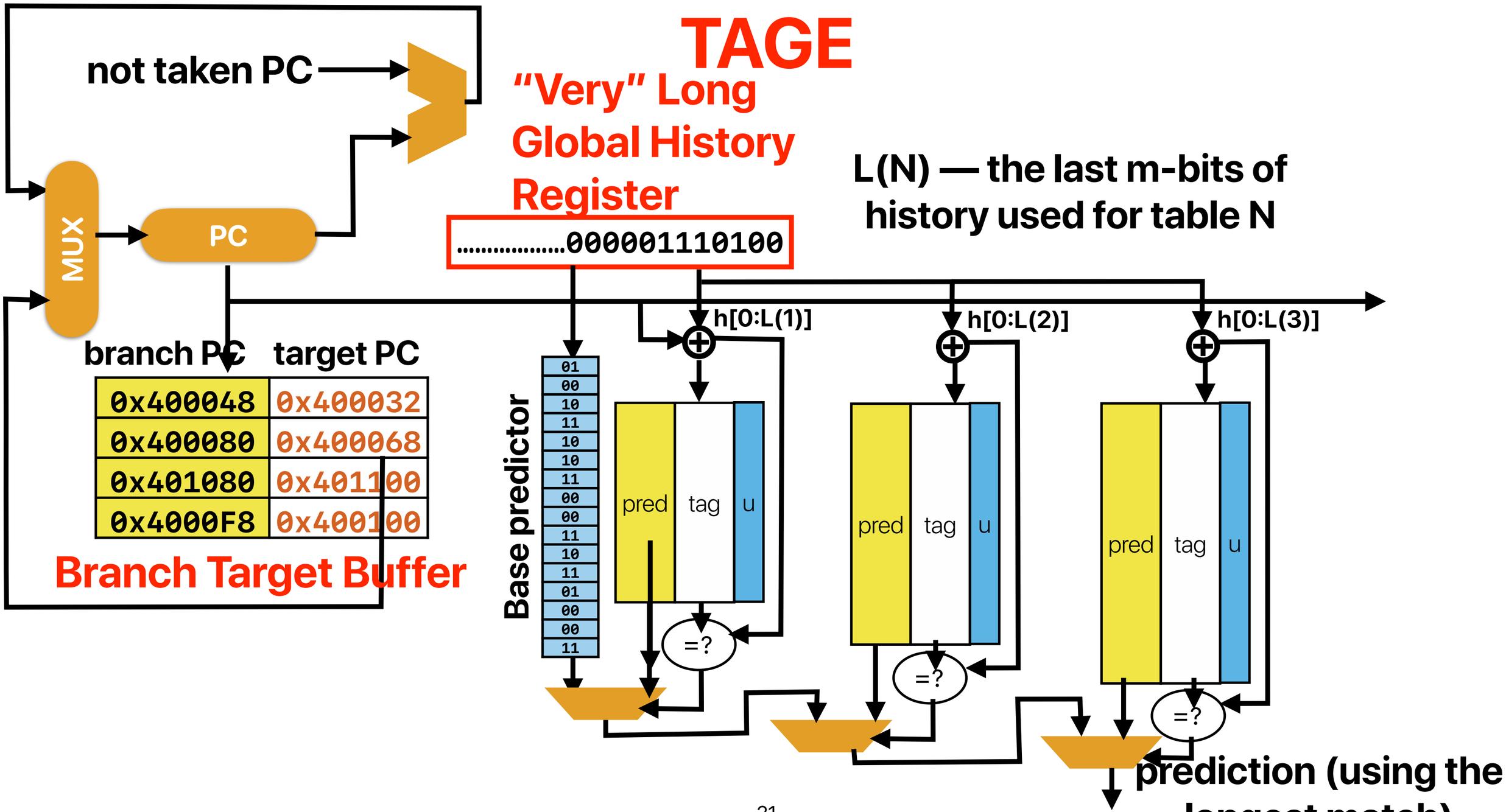
L could be better

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

different branch needs different length of history

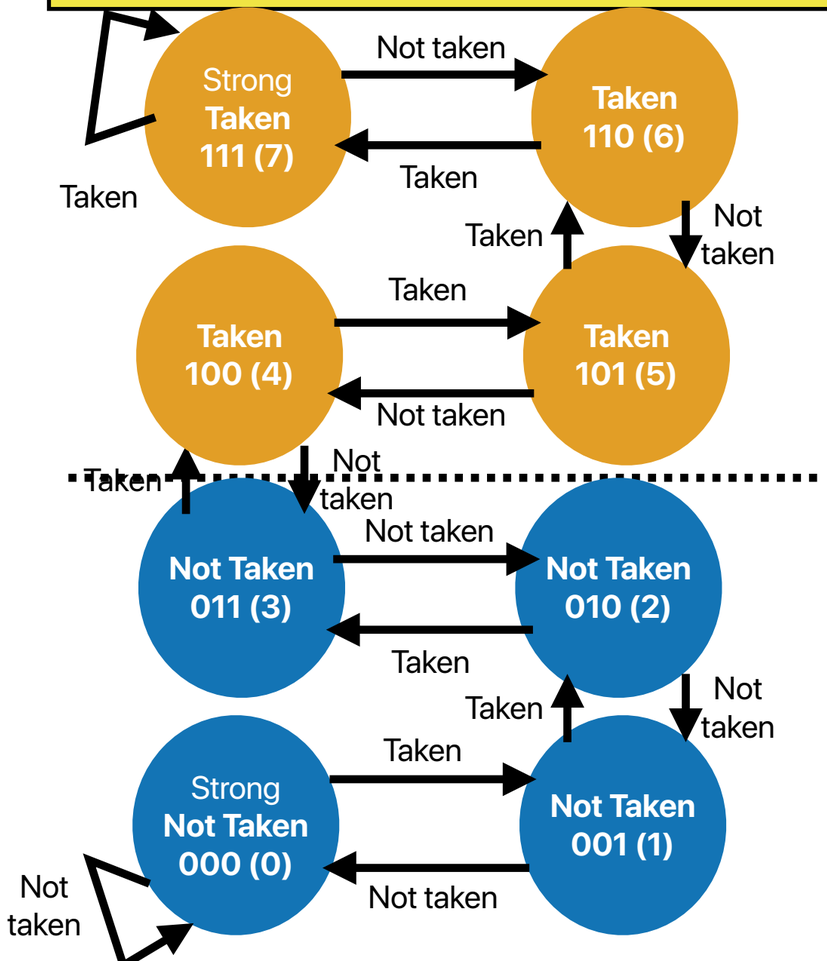
global predictor can work if the history is long enough!

TAGE


not taken PC
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
=?
prediction (using the longest match)

What's inside each table?

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

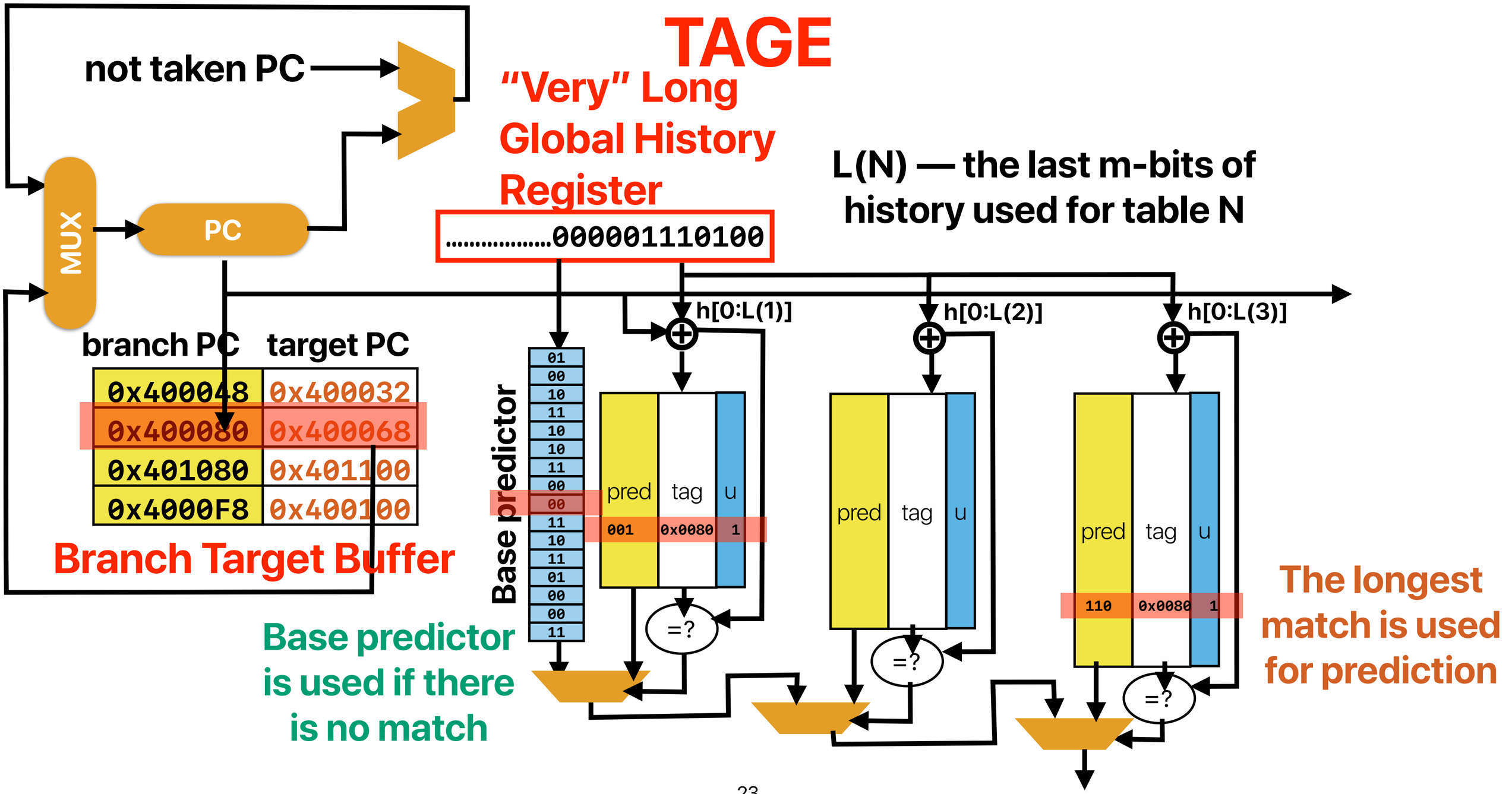


$$if\ prediction(alt\_predictor) \neq prediction(pred):$$

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

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

TAGE


not taken PC
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
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

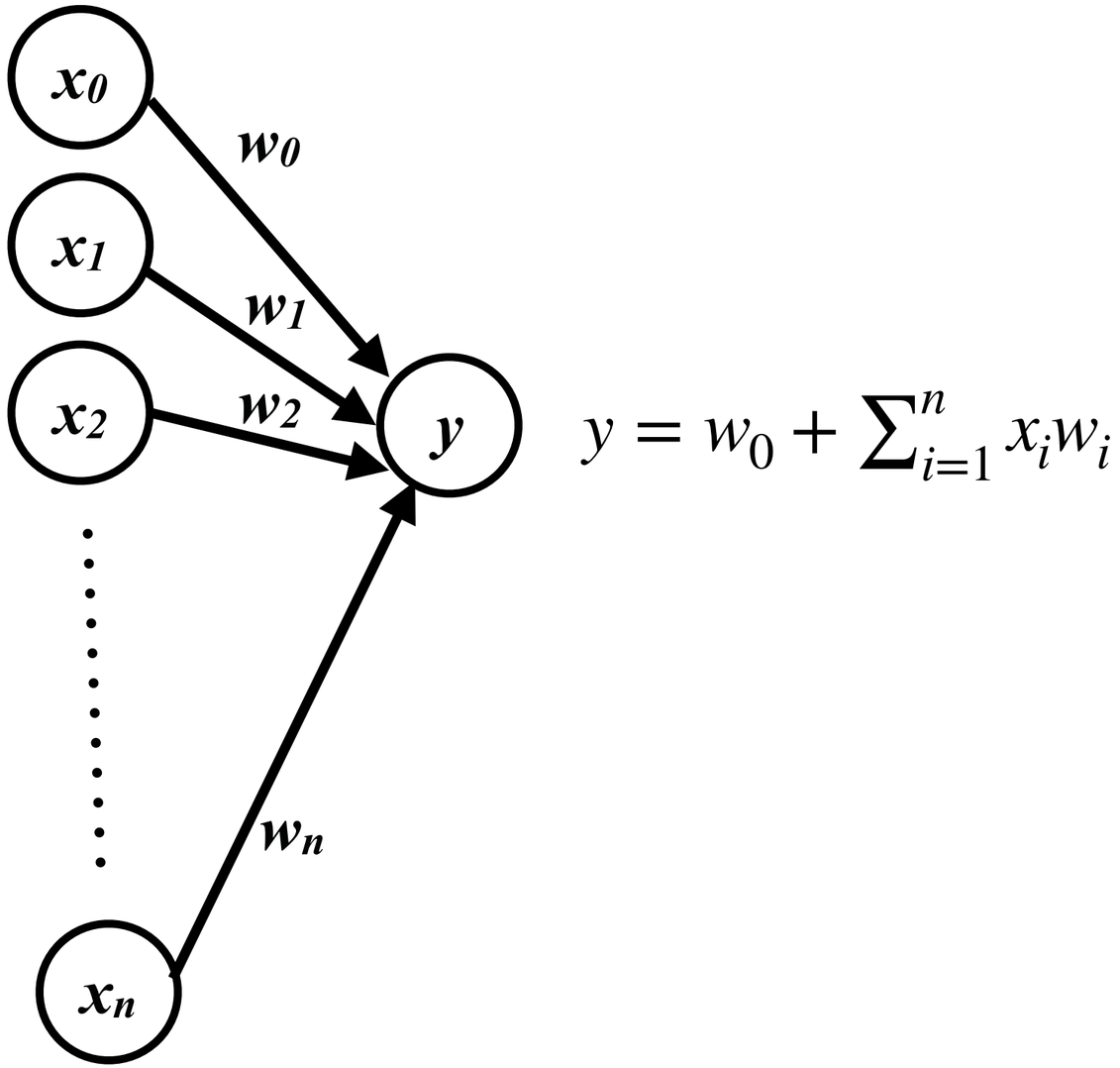
- The machine learns to predict conditional branches
- The formation of the problem
  {branch address, branch history, some other attributes.. } —
  {“taken”, “not taken”}
- Artificial neural networks
  - Simple model of neural networks in brain cells
  - Learn to recognize and classify patterns

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

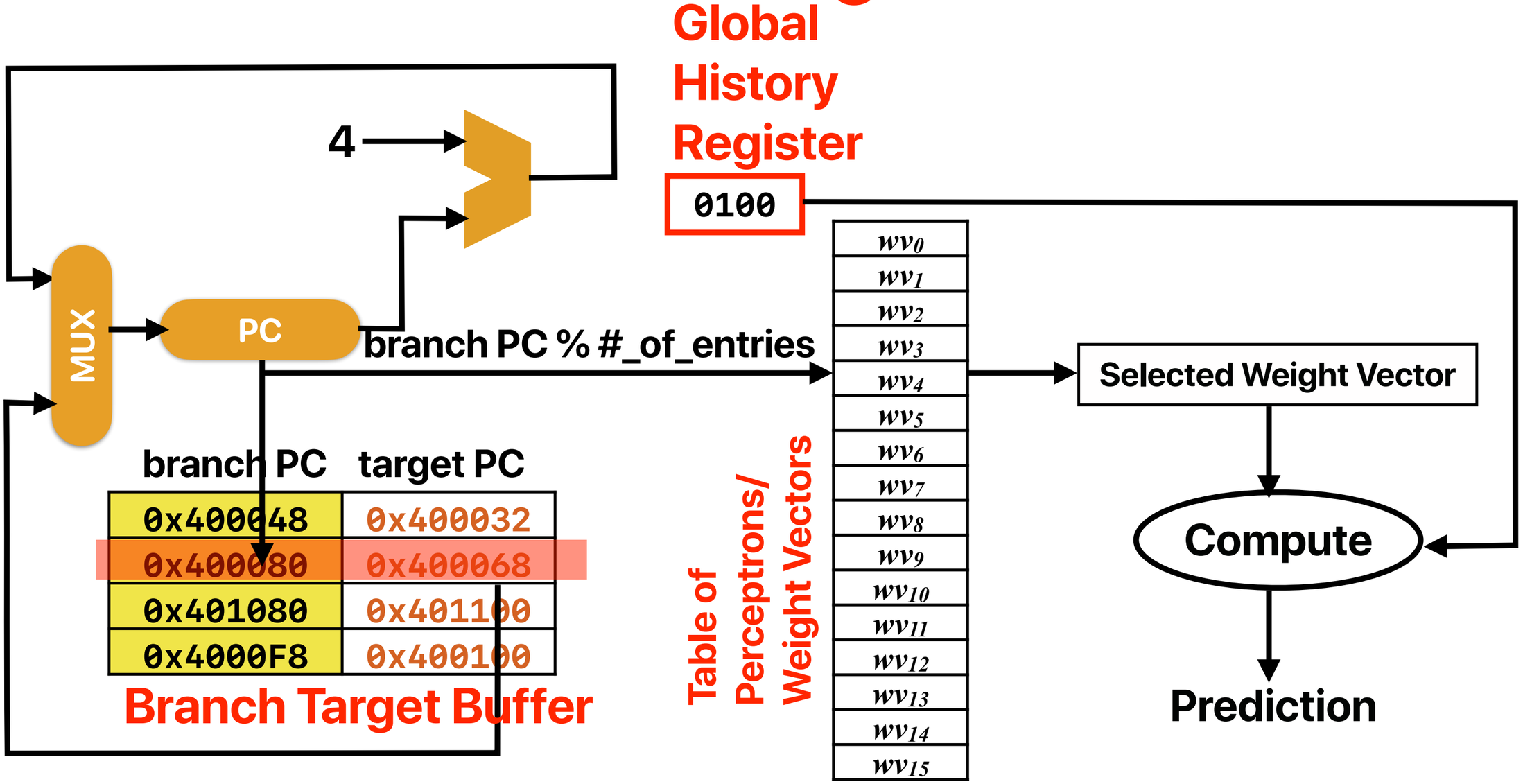


Training Algorithm

$x_{1..n}$ 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 = x_i then
            w_i := w_i + 1
        else
            w_i := w_i − 1
        end if
    end for
end if
```

Predictor Organization



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);
```

about the same

II

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

about the same

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)
```

L could be better

A. 0

B. 1

C. 2

D. 3

E. 4

Perceptron can discount this when predicting while

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

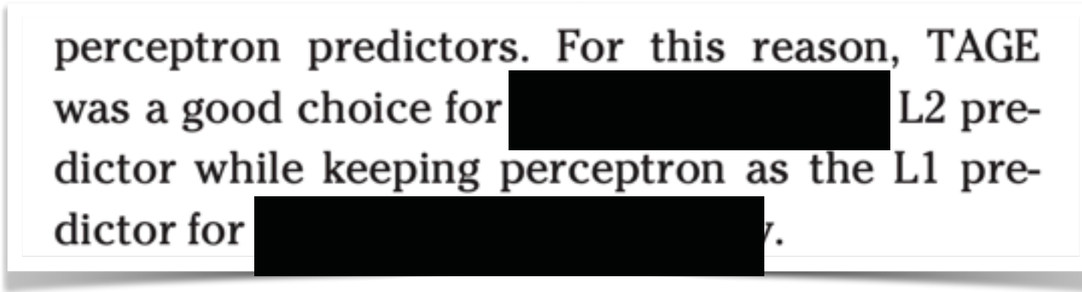
Design decisions in real practice

- 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

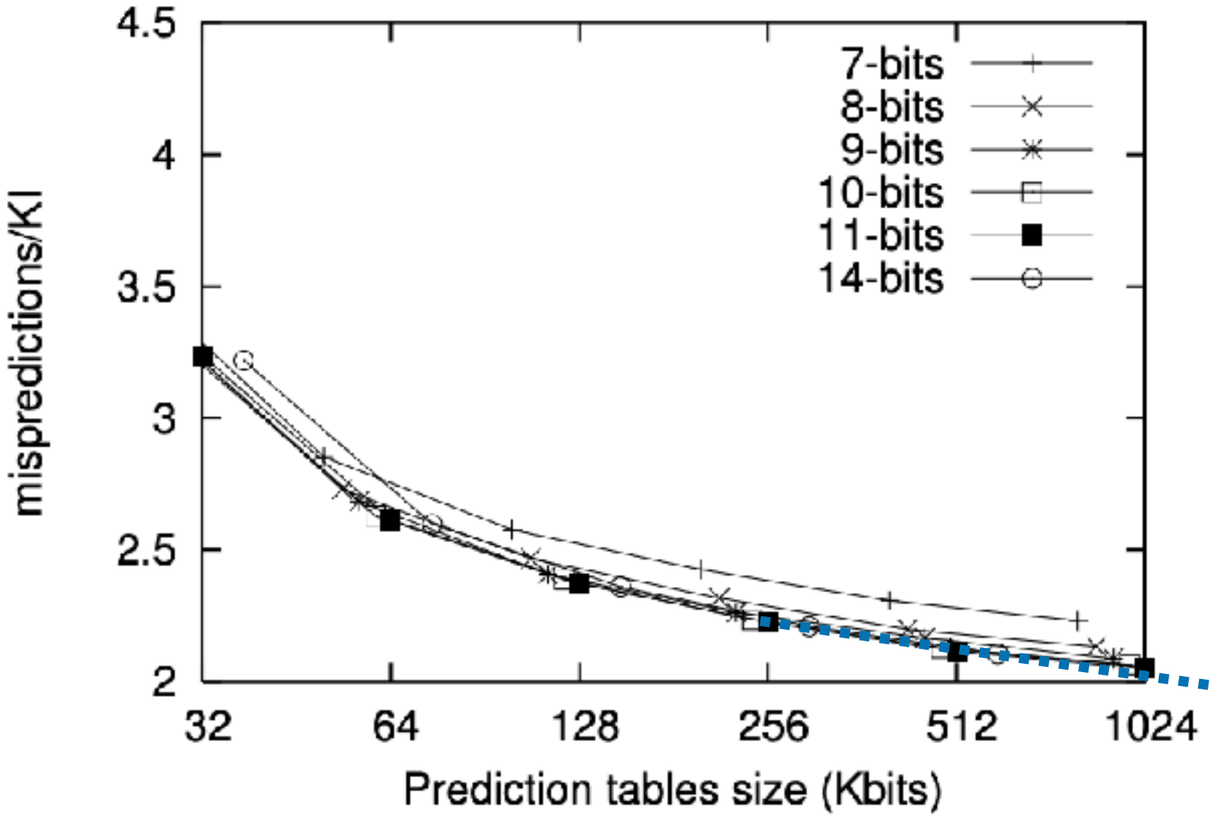
perceptron predictors. For this reason, TAGE was a good choice for [redacted] L2 predictor while keeping perceptron as the L1 predictor for [redacted].

Design decisions in real practice

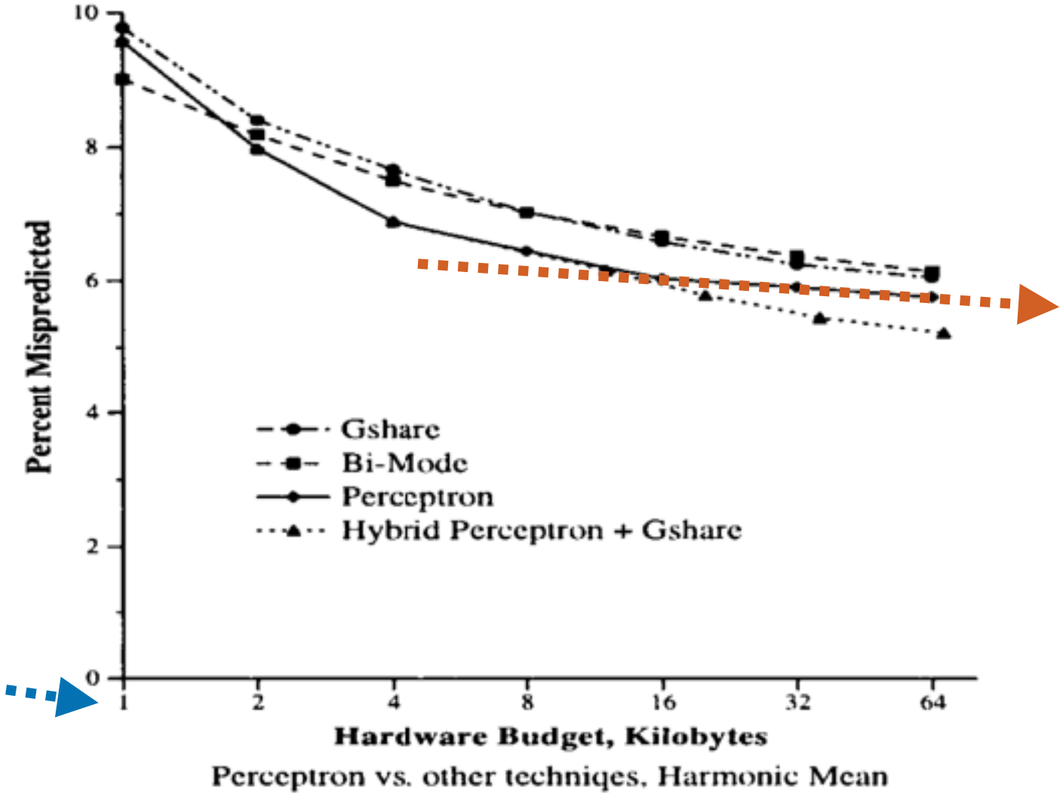
- AMD Zen 2 (RyZen 3000 series processors) adopts a design with first level predictor using perceptron and using TAGE for the 2nd level. What such a design decision implies about the characteristics of TAGE and Perceptron?
  - ① Perceptron takes longer to train than TAGE
  - ② Perceptron takes longer 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?

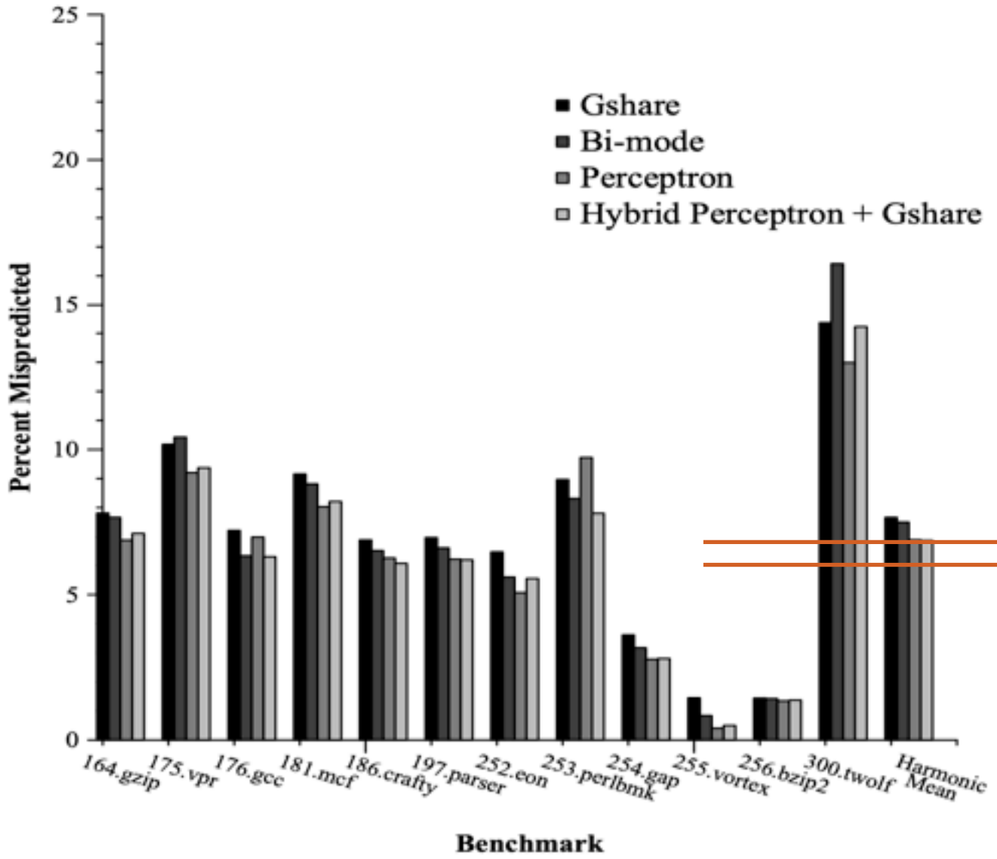


Figure 4: Misprediction Rates at a 4K budget. The perceptron predictor has a lower misprediction rate than *gshare* for all benchmarks except for `186.crafty` and `197.parser`.

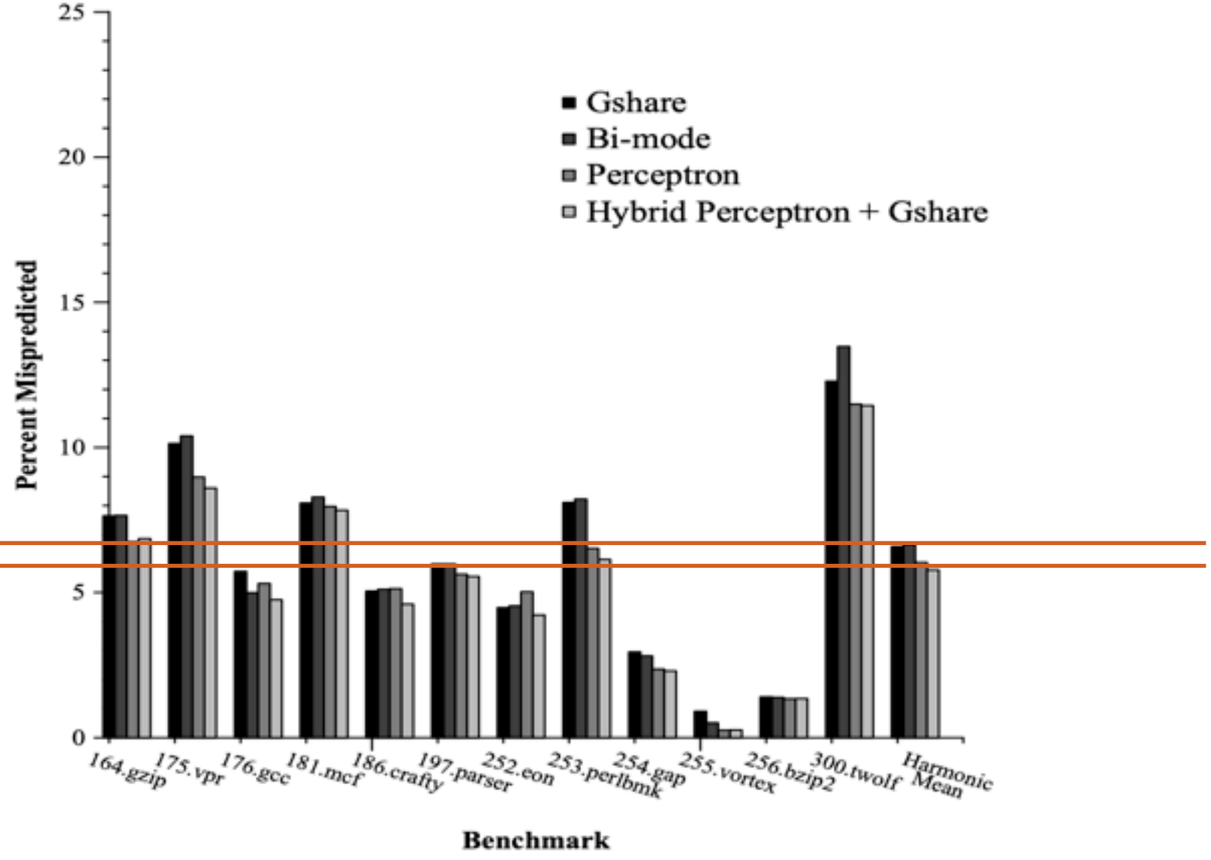
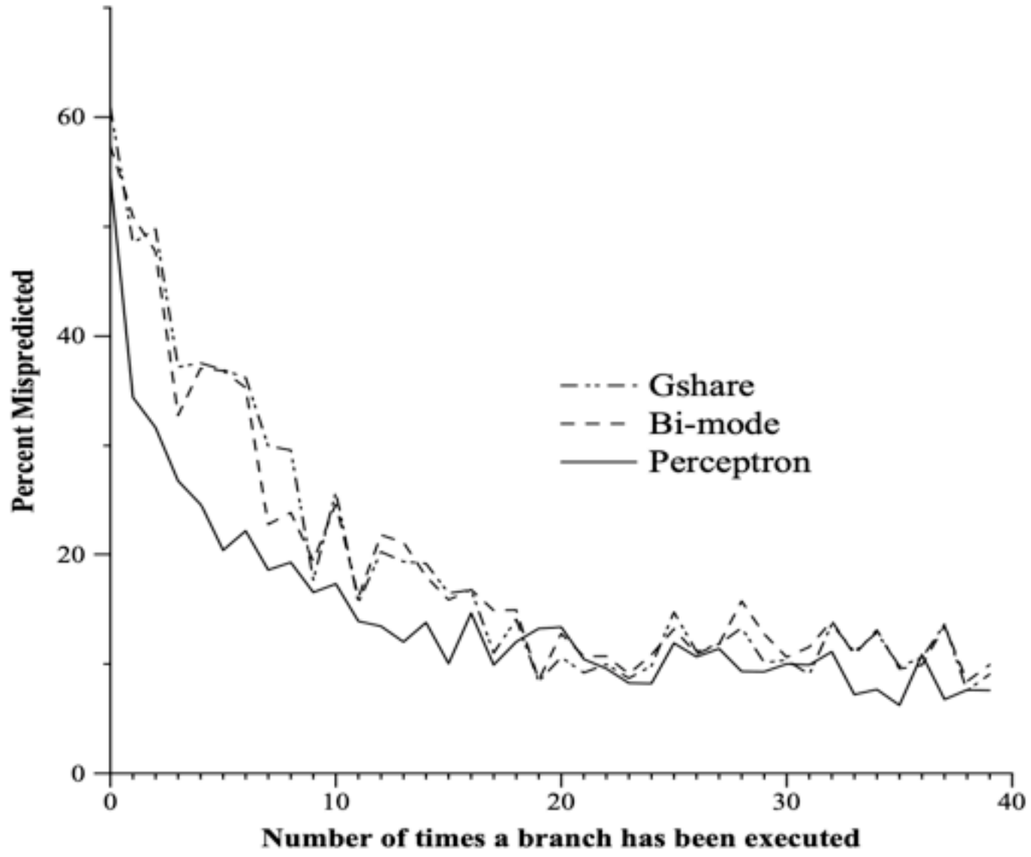


Figure 5: Misprediction Rates at a 16K budget. Gshare outperforms the perceptron predictor only on `186.crafty`. The hybrid predictor is consistently better than the PHT schemes.

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

A. 0

no — based on the paper

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

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

```
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!!!

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

Demo: Popcount

- Given a 64-bit integer number, find the number of 1s in its binary representation.
- Example 1:
  Input: 59487
  Output: 10
  Explanation: 59487's binary representation is 0b1110100001011111

```
int main(int argc, char *argv[]) {

      uint64_t key = 0xdeadbeef;

      int count = 1000000000;
      uint64_t sum = 0;

      for (int i=0; i < count; i++)
      {
            sum += popcount(RandLFSR(key));
      }
      printf("Result: %lu\n", sum);
      return sum;
}
```

Five implementations

- Which of the following implementations will perform the best on modern pipeline processors?

A

```
inline int popcount(uint64_t x){
  int c=0;
  while(x)  {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

B

```
inline int popcount(uint64_t x) {
   int c = 0;
   while(x)     {
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
   }
   return c;
}
```

C

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     while(x)     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
}
```

D

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
  }
```

E

```
inline int popcount(uint64_t x) {
     int c = 0;
     for (uint64_t i = 0; i < 16; i++)
     {
         switch((x & 0xF))
         {
             case 1: c+=1; break;
             case 2: c+=1; break;
             case 3: c+=2; break;
             case 4: c+=1; break;
             case 5: c+=2; break;
             case 6: c+=2; break;
             case 7: c+=3; break;
             case 8: c+=1; break;
             case 9: c+=2; break;
             case 10: c+=2; break;
             case 11: c+=3; break;
             case 12: c+=2; break;
             case 13: c+=3; break;
             case 14: c+=3; break;
             case 15: c+=4; break;
             default: break;
         }
         x = x >> 4;
     }
     return c;
}
```

Five implementations

- Which of the following implementations will perform the best on modern pipeline processors?

A

```
inline int popcount(uint64_t x){
  int c=0;
  while(x)  {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

B

```
inline int popcount(uint64_t x) {
   int c = 0;
   while(x)     {
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
   }
   return c;
}
```

C

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     while(x)     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
}
```

D

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
 }
```

E

```
inline int popcount(uint64_t x) {
     int c = 0;
     for (uint64_t i = 0; i < 16; i++)
     {
         switch((x & 0xF))
         {
             case 1: c+=1; break;
             case 2: c+=1; break;
             case 3: c+=2; break;
             case 4: c+=1; break;
             case 5: c+=2; break;
             case 6: c+=2; break;
             case 7: c+=3; break;
             case 8: c+=1; break;
             case 9: c+=2; break;
             case 10: c+=2; break;
             case 11: c+=3; break;
             case 12: c+=2; break;
             case 13: c+=3; break;
             case 14: c+=3; break;
             case 15: c+=4; break;
             default: break;
         }
         x = x >> 4;
     }
     return c;
}
```

Why is B better than A?

- How many of the following statements explains the reason why B outperforms A with compiler optimizations
  - ① B has lower dynamic instruction count than A
  - ② B has significantly lower branch mis-prediction rate than A
  - ③ B has significantly fewer branch instructions than A
  - ④ B can incur fewer data hazards

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

A

```
inline int popcount(uint64_t x){
  int c=0;
  while(x)  {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

B

```
inline int popcount(uint64_t x) {
    int c = 0;
    while(x)      {
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
    }
    return c;
}
```

Why is B better than A?

A

```
inline int popcount(uint64_t x){
  int c=0;
  while(x)  {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

```
movl    %eax, %ecx
andl    $1, %ecx
addl    %ecx, %edx
shrq    %rax
jne     .L6
```

5*n instructions

B

```
inline int popcount(uint64_t x) {
   int c = 0;
   while(x)     {
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
   }
   return c;
}
```

15*(n/4) = 3.75*n instructions

```
movl    %ecx, %eax
andl    $1, %eax
addl    %edx, %eax
movq    %rcx, %rdx
shrq    %rdx
andl    $1, %edx
addl    %eax, %edx
movq    %rcx, %rax
shrq    $2, %rax
andl    $1, %eax
addl    %edx, %eax
movq    %rcx, %rdx
shrq    $3, %rdx
andl    $1, %edx
addl    %eax, %edx
shrq    $4, %rcx
jne     .L6
```

Only one branch for four iterations in A

Why is B better than A?

- How many of the following statements explains the reason why B outperforms A with compiler optimizations
  - ① ✓ B has lower dynamic instruction count than A
  - ② B has significantly lower branch mis-prediction rate than A
  - ③ ✓ B has significantly fewer branch instructions than A
  - ④ ✓ B can incur fewer data hazards

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

A

```
inline int popcount(uint64_t x){
  int c=0;
  while(x)  {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

B

```
inline int popcount(uint64_t x) {
    int c = 0;
    while(x)      {
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
    }
    return c;
}
```

Why is C better than B?

- How many of the following statements explains the reason why B outperforms C with compiler optimizations
  - ① C has lower dynamic instruction count than B
  - ② C has significantly lower branch mis-prediction rate than B
  - ③ C has significantly fewer branch instructions than B
  - ④ C can incur fewer data hazards

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

C

```
inline int popcount(uint64_t x) {
      int c = 0;
      int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
      while(x)      {
          c += table[(x & 0xF)];
          x = x >> 4;
      }
      return c;
}
```

B

```
inline int popcount(uint64_t x) {
    int c = 0;
    while(x)      {
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
    }
    return c;
}
```

Why is C better than B?

- How many of the following statements explains the reason why B outperforms C with compiler optimizations
  - ① C has lower dynamic instruction count than B — C only needs one load, one add, one shift, the same amount of iterations
  - ② C has significantly lower branch mis-prediction rate than B — the same number being predicted.
  - ③ C has significantly fewer branch instructions than B — the same amount of branches
  - ④ C can incur fewer data hazards — Probably not. In fact, the load may have negative effect without architectural supports

A. 0

B. 1

C. 2

D. 3

E. 4

C

```
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    while(x)     {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

B

```
inline int popcount(uint64_t x) {
    int c = 0;
    while(x)     {
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
    }
    return c;
}
```

Announcements

- **Assignment 4** due next **Thursday**
- **Reading Quiz 7** due **next Tuesday** before the lecture

Computer Science & Engineering

203

つづく