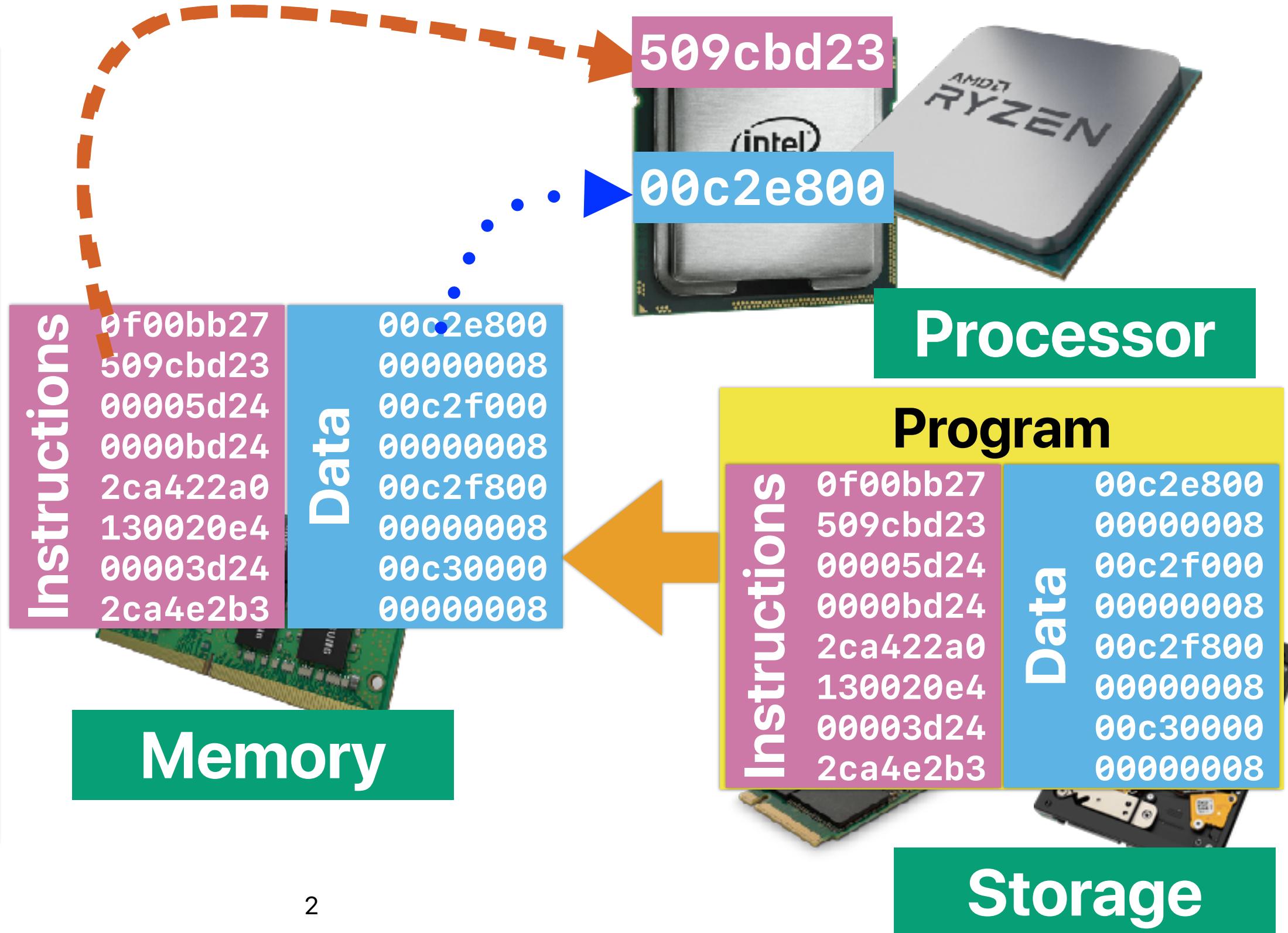


Modern Processor Design (III): I Just Can't Wait

Hung-Wei Tseng

Recap: von Neumann Architecture

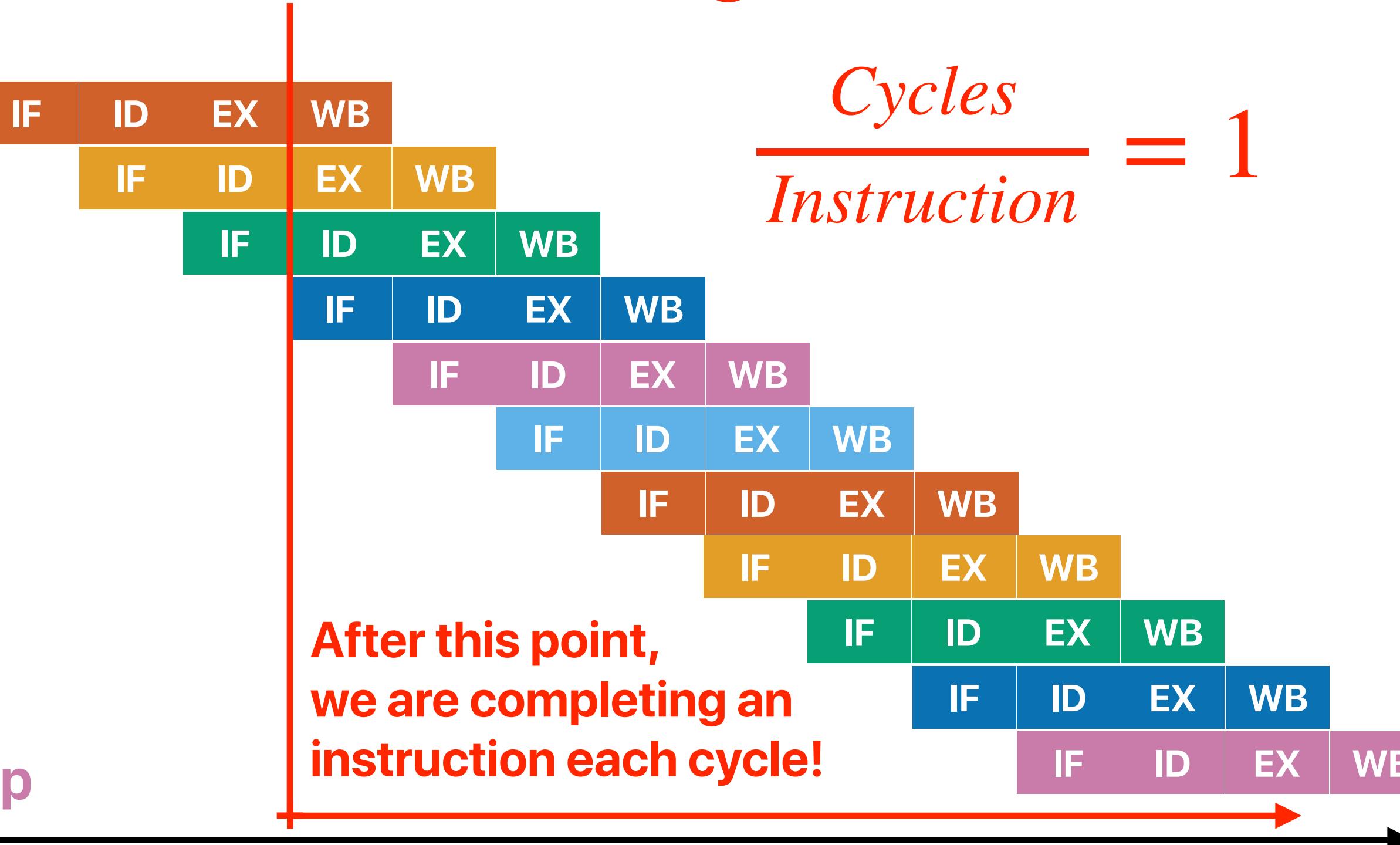


Recap: The “life” of an instruction

- Instruction Fetch (**IF**) — fetch the instruction from memory
- Instruction Decode (**ID**)
 - Decode the instruction for the desired operation and operands
 - Reading source register values
- Execution (**EX**)
 - ALU instructions: Perform ALU operations
 - Conditional Branch: Determine the branch outcome (taken/not taken)
 - Memory instructions: Determine the effective address for data memory access
- Data Memory Access (**MEM**) — Read/write memory
- Write Back (**WB**) — Present ALU result/read value in the target register
- Update PC
 - If the branch is taken — set to the branch target address
 - Otherwise — advance to the next instruction — current PC + 4

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



Recap: Three pipeline hazards

- Structural hazards — resource conflicts cannot support simultaneous execution of instructions in the pipeline
- Control hazards — the PC can be changed by an instruction in the pipeline
- Data hazards — an instruction depending on a the result that's not yet generated or propagated when the instruction needs that

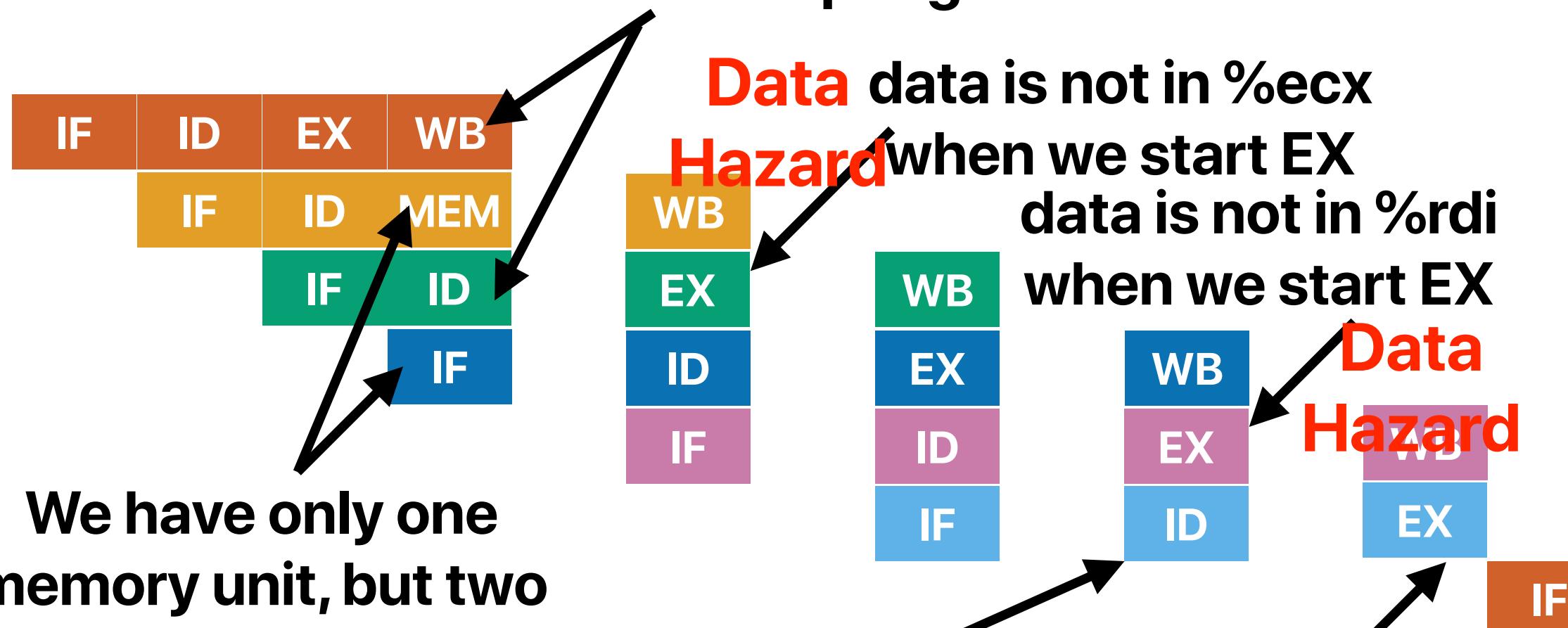
Structural Hazard

Recap: hazards

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

- 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

Structural Hazard

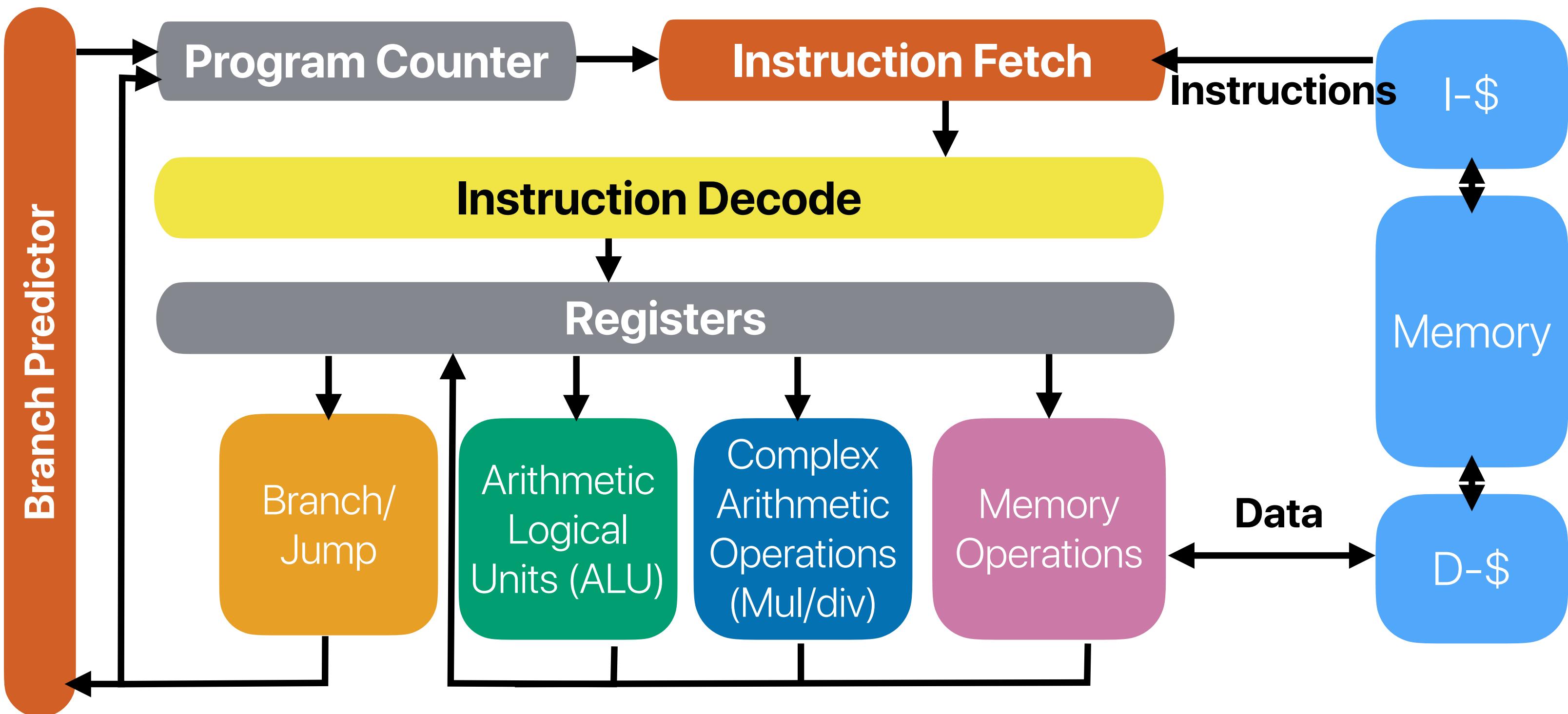


We have only one memory unit, but two access requests!

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

Control Hazard

Recap: Microprocessor with a “branch predictor”



Recap: branch predictors

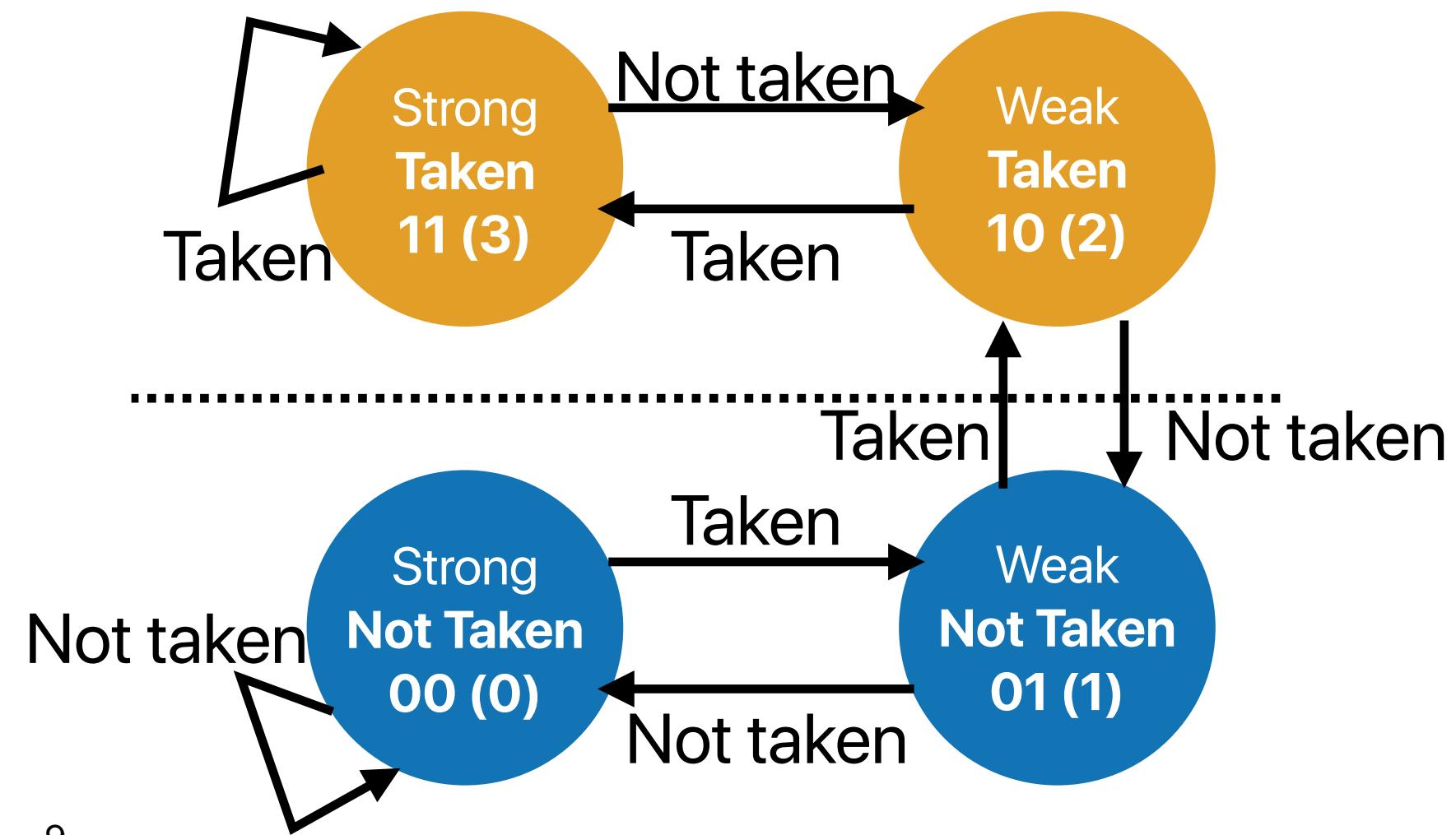
- If we guess right — no penalty
- **If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF and ID stages and reset the PC**
- Global
 - Predictors do not keep states for each branch instructions
 - Predictors do not rely on the outcome of single branch instructions to predict outcome
- 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

Recap: 2-bit/Bimodal local predictor

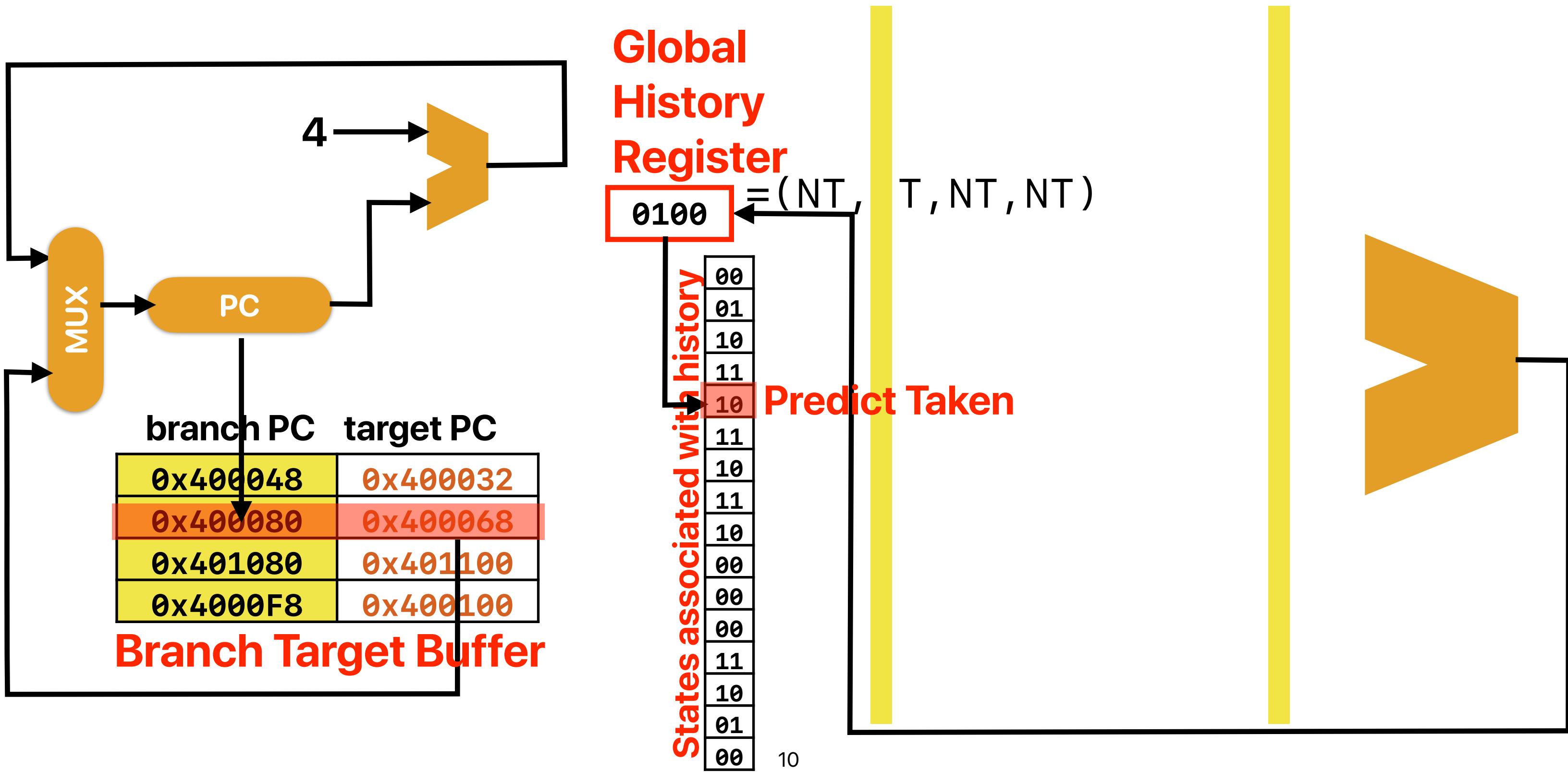
- Local predictor — every branch instruction has its own state
- 2-bit — each state is described using 2 bits
- Change the state based on **actual outcome**

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

Predict Taken



Recap: Global history (GH) 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
```

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

Who do you trust the most?
Friends? Parents? SOs? Or?

Outline

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

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 = 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

Global v.s. local

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

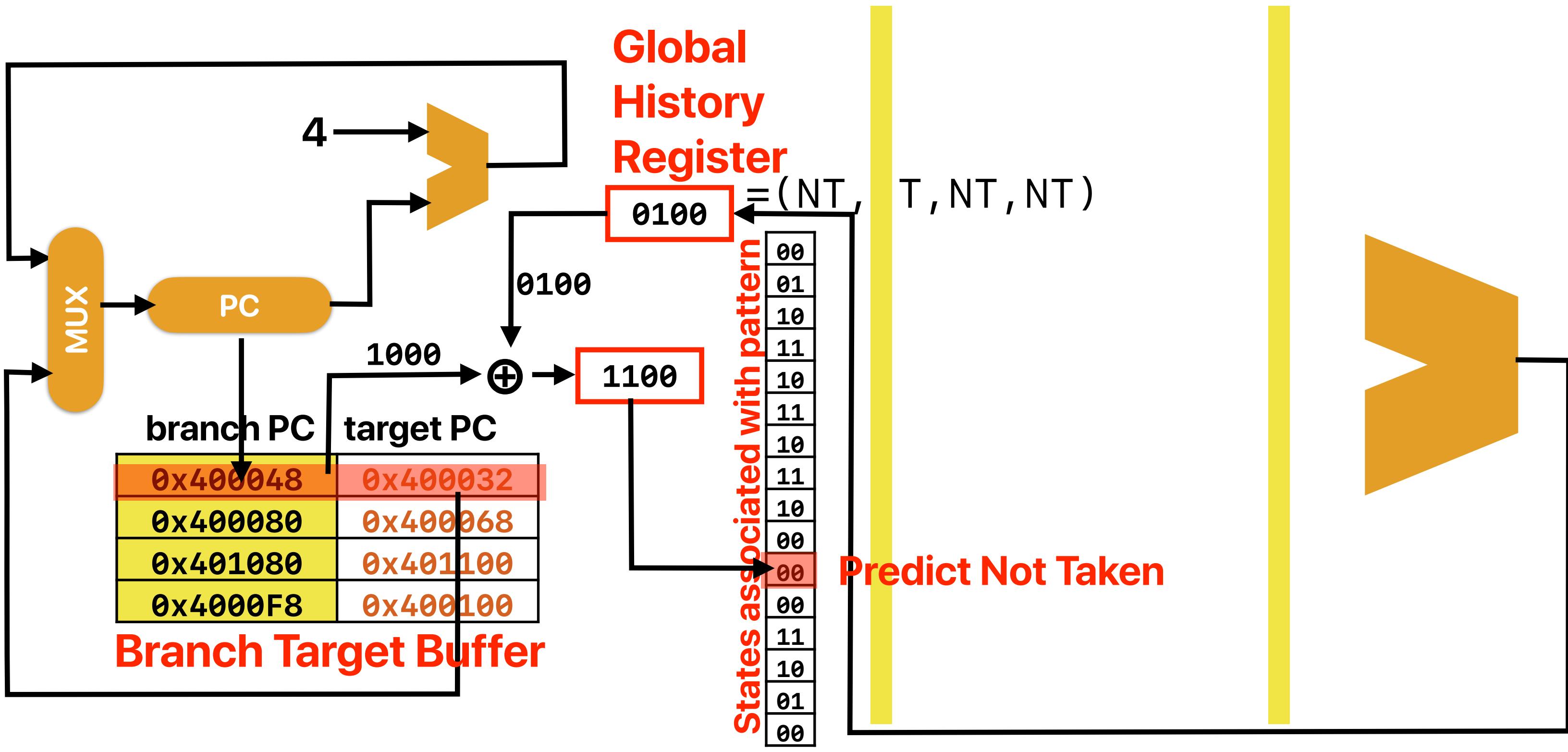
C. 2

D. 3

E. 4

Hybrid predictors

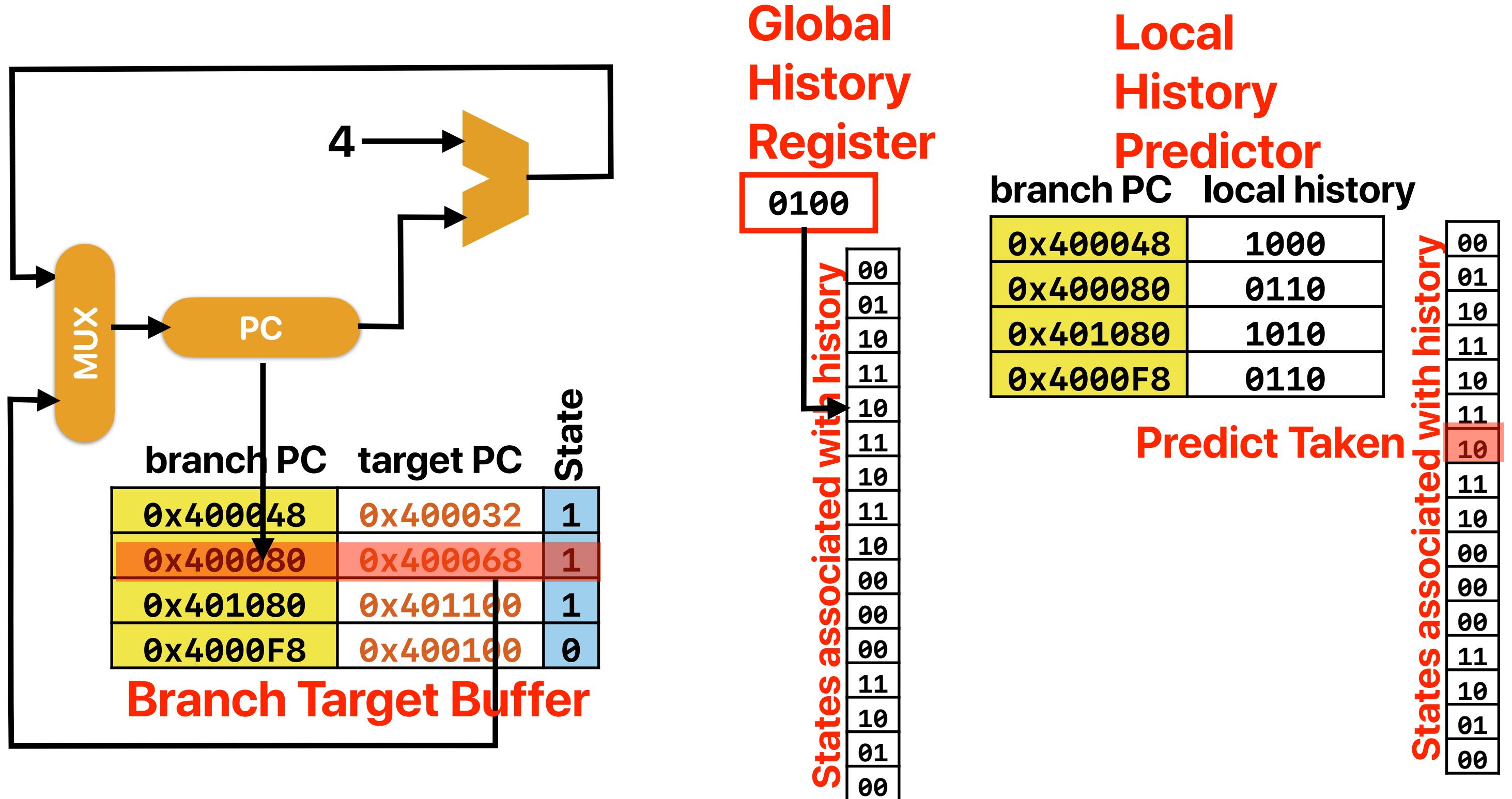
gshare predictor



gshare predictor

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

Tournament Predictor

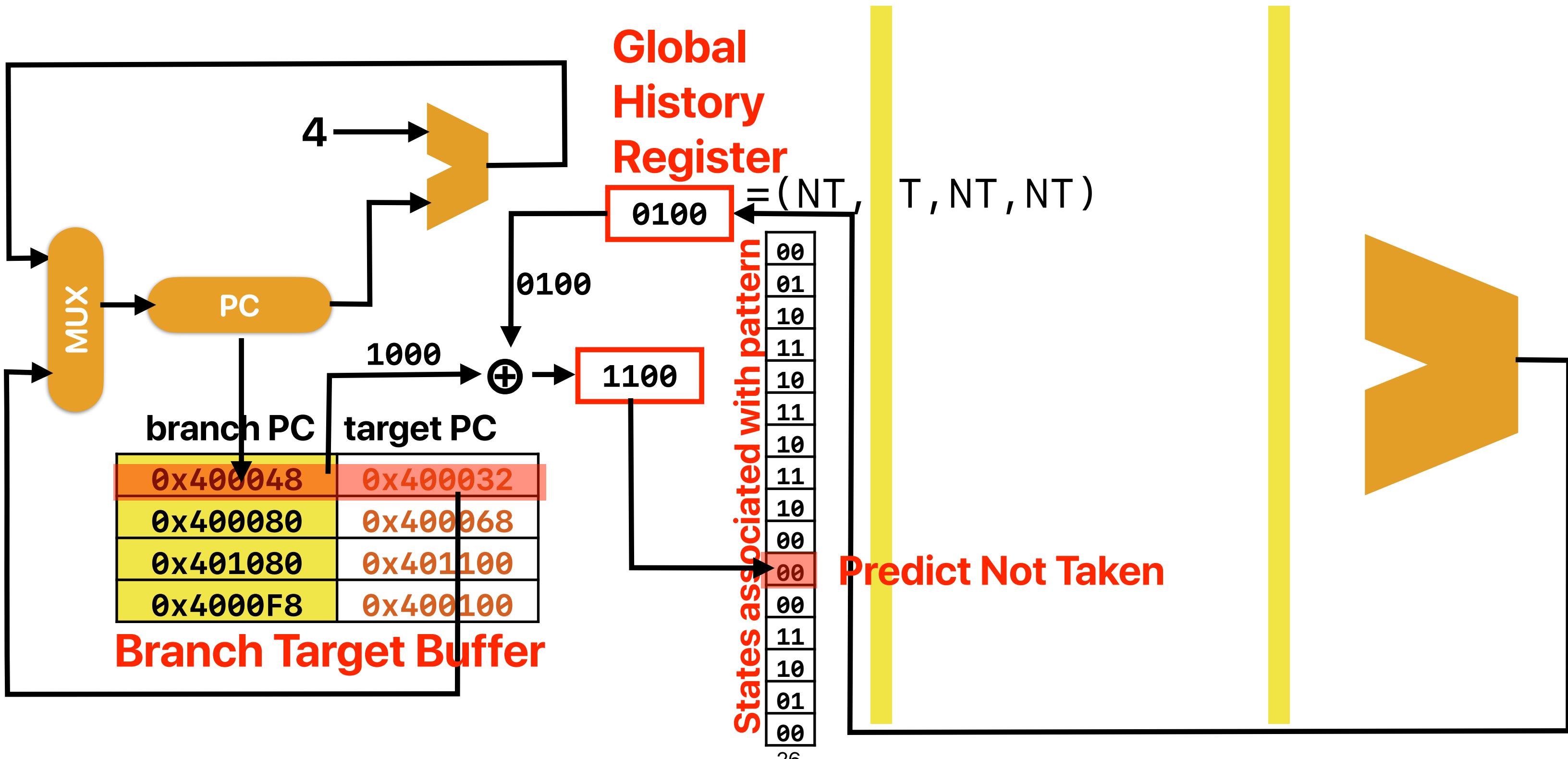


Tournament Predictor

- The state predicts “which predictor is better”
 - Local history
 - Global history
- The predicted predictor makes the prediction

Hybrid predictors

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)?

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);
```

 **L could be better**

```
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)
```

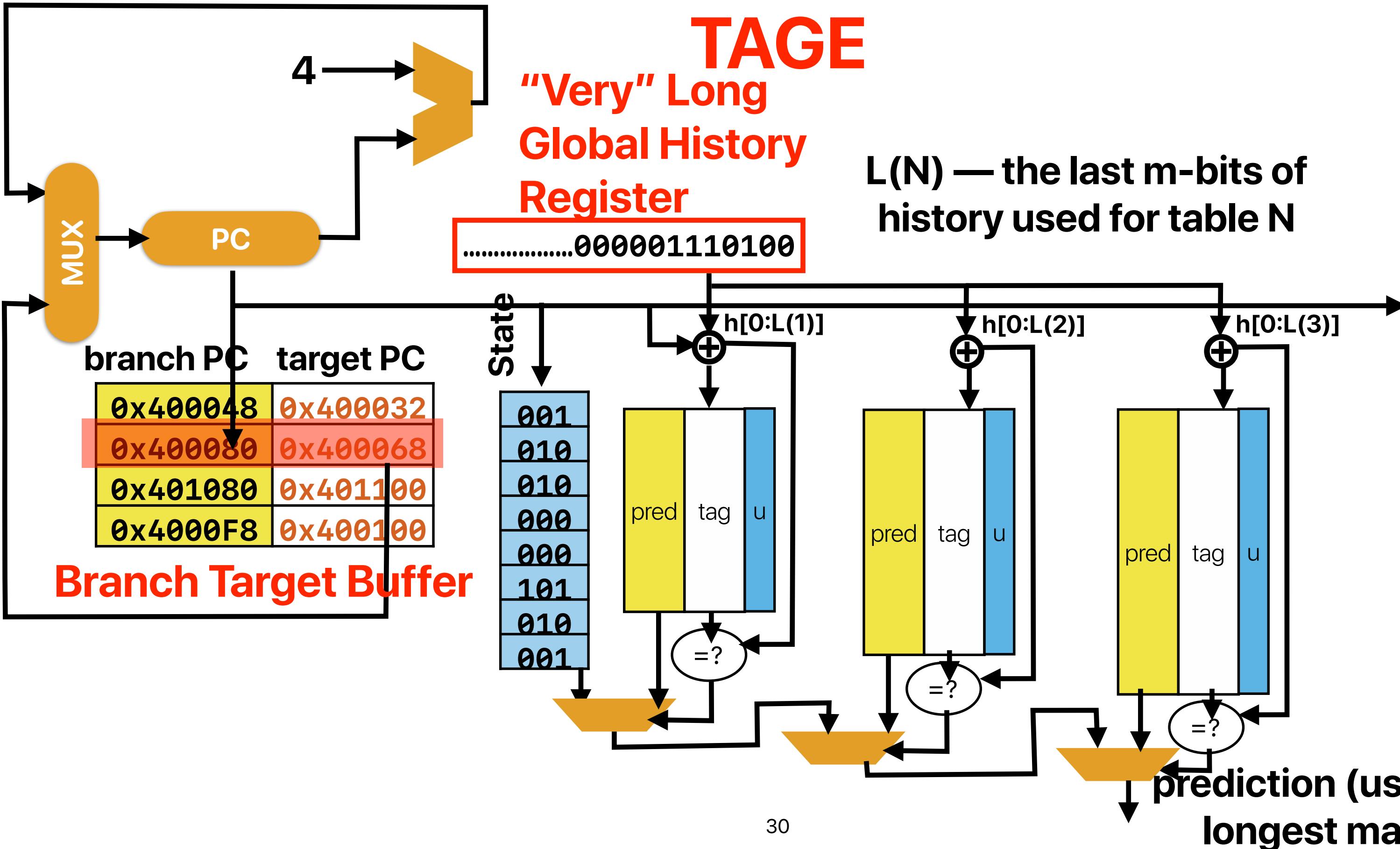
- A. 0
- B. 1**
- C. 2
- D. 3
- E. 4

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



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

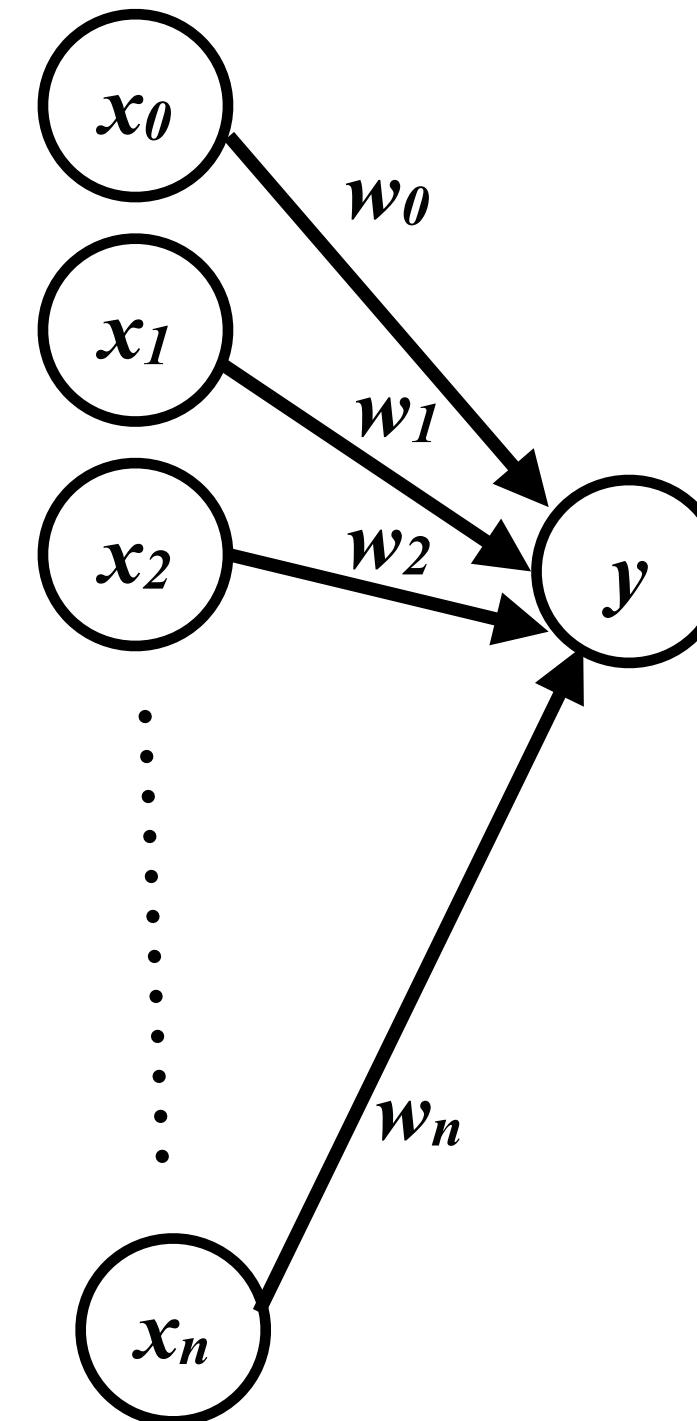
- The machine learns to predict conditional branches
- Artificial neural networks
 - Simple model of neural networks in brain cells
 - Learn to recognize and classify patterns

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.

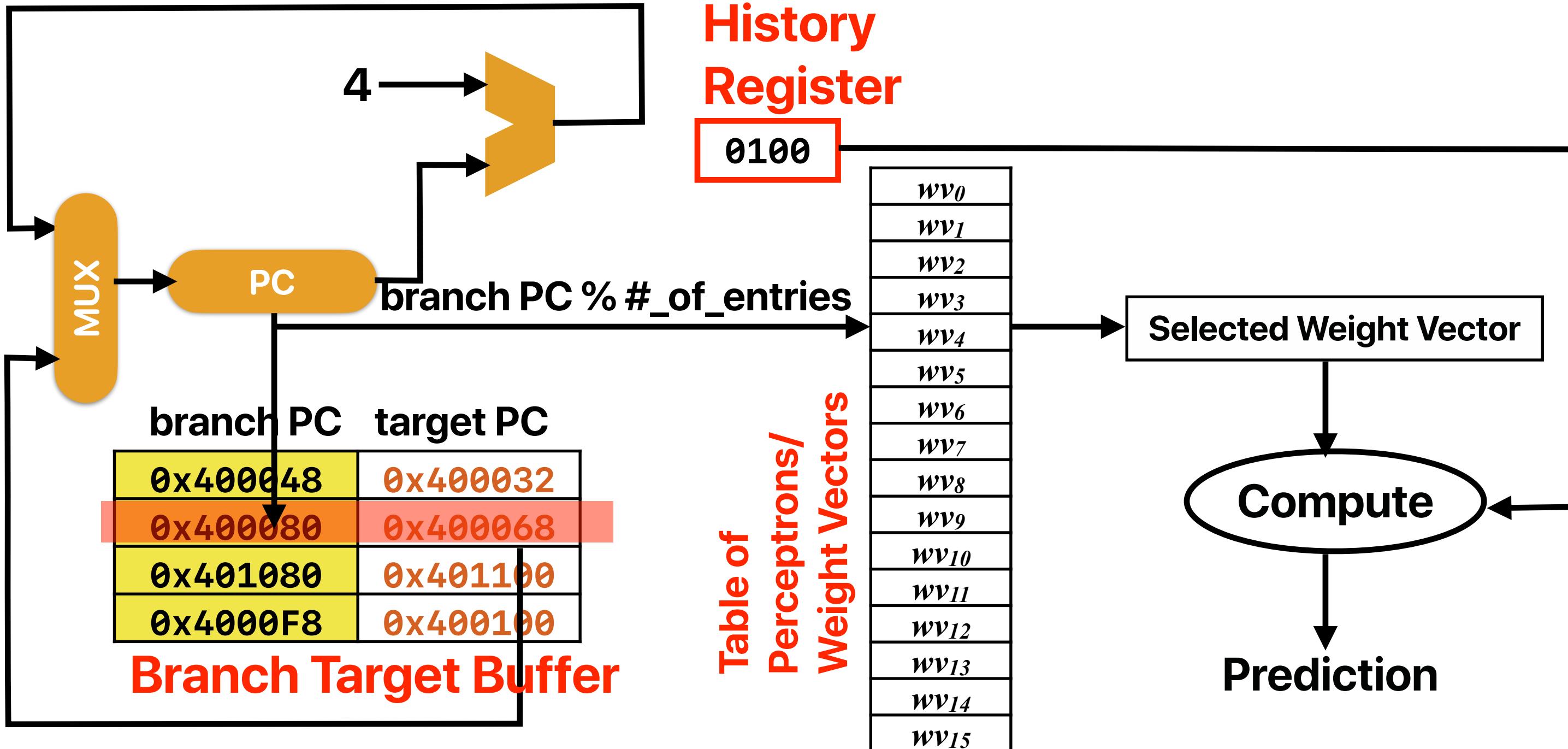
```
if  $|y| \leq \theta$  or  $((y \geq 0) \neq t)$  then
    for each  $0 \leq i \leq n$  in parallel
        if  $t = x_i$  then
             $w_i := w_i + 1$ 
        else
             $w_i := w_i - 1$ 
        end if
    end for
end if
```

Predictor Organization

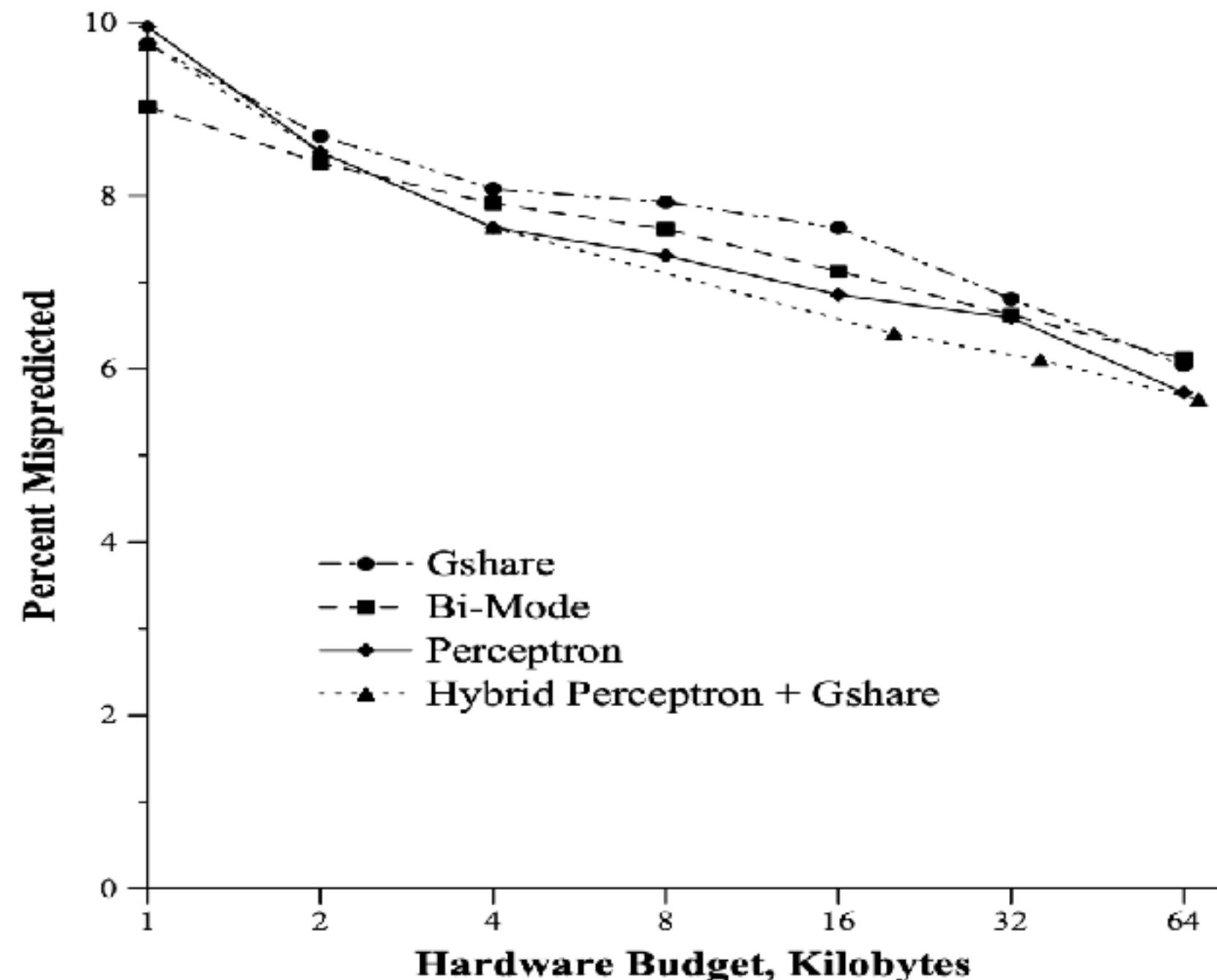
Global

History
Register

0100



How good is prediction using perceptrons?



Perceptron vs. other techniques, Context Switching

How good is prediction using perceptrons?

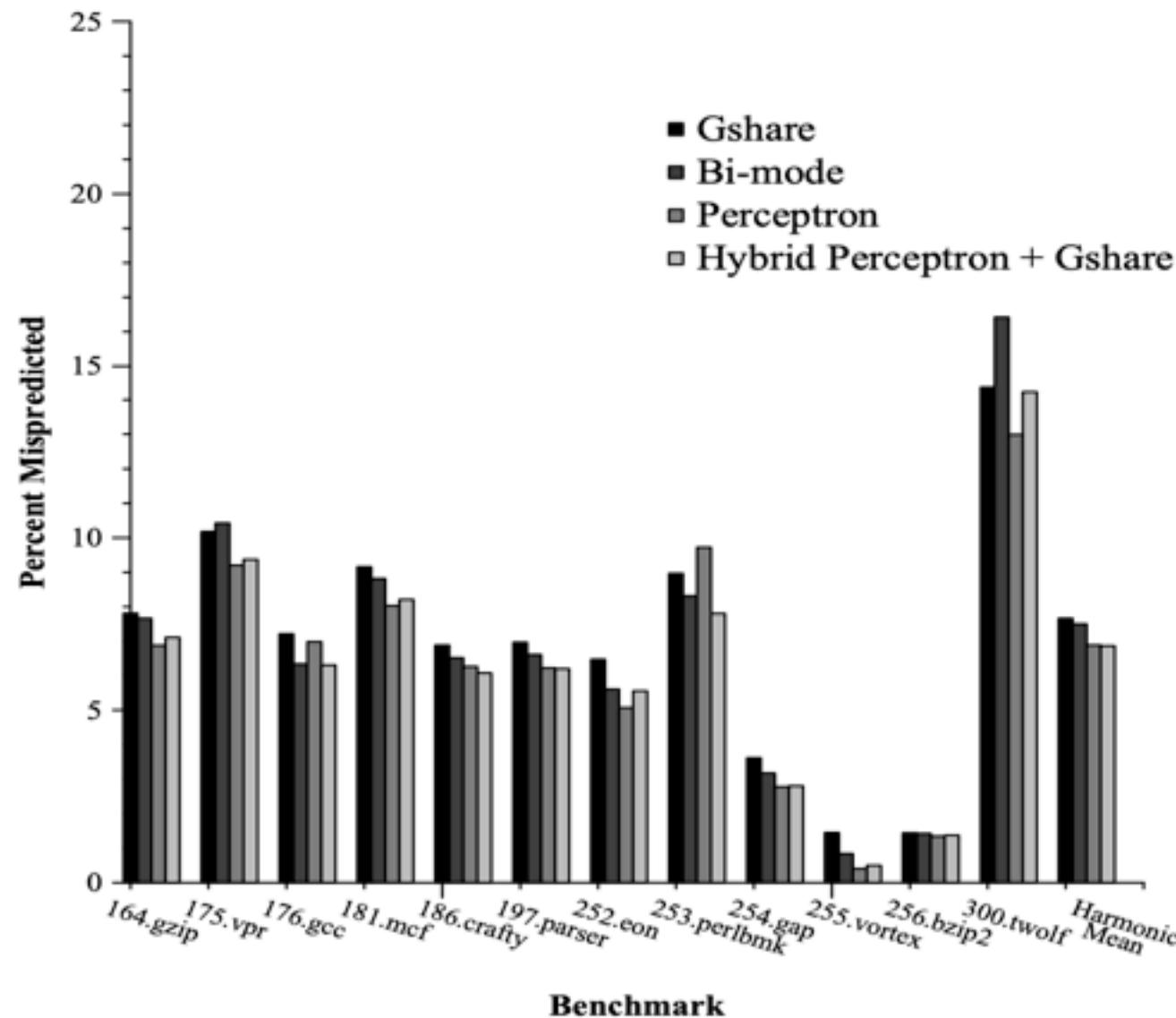


Figure 4: Misprediction Rates at a 4K budget. The perceptron predictor has a lower misprediction rate than *gshare* for all benchmarks except for *186.crafty* and *197.parser*.

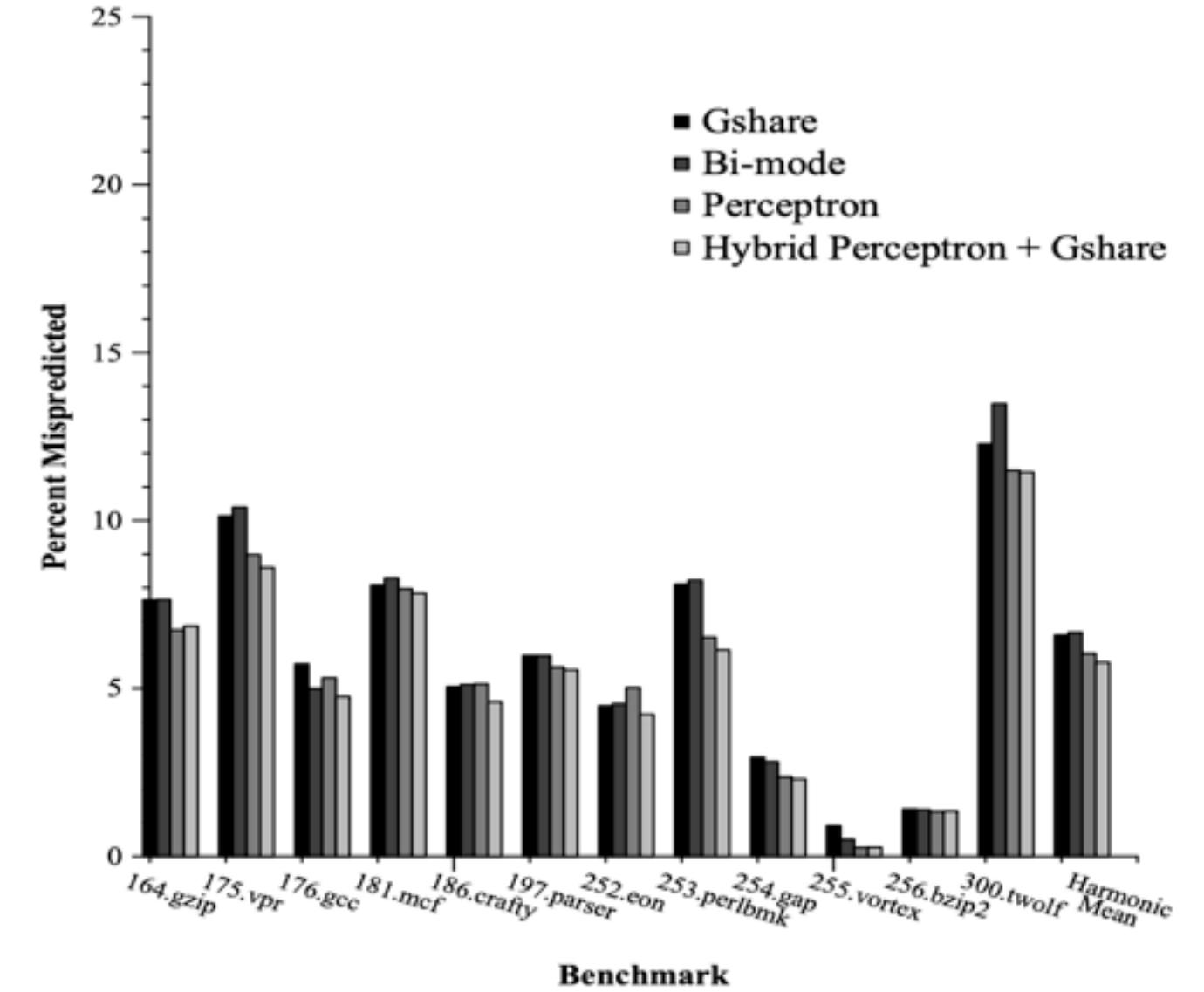
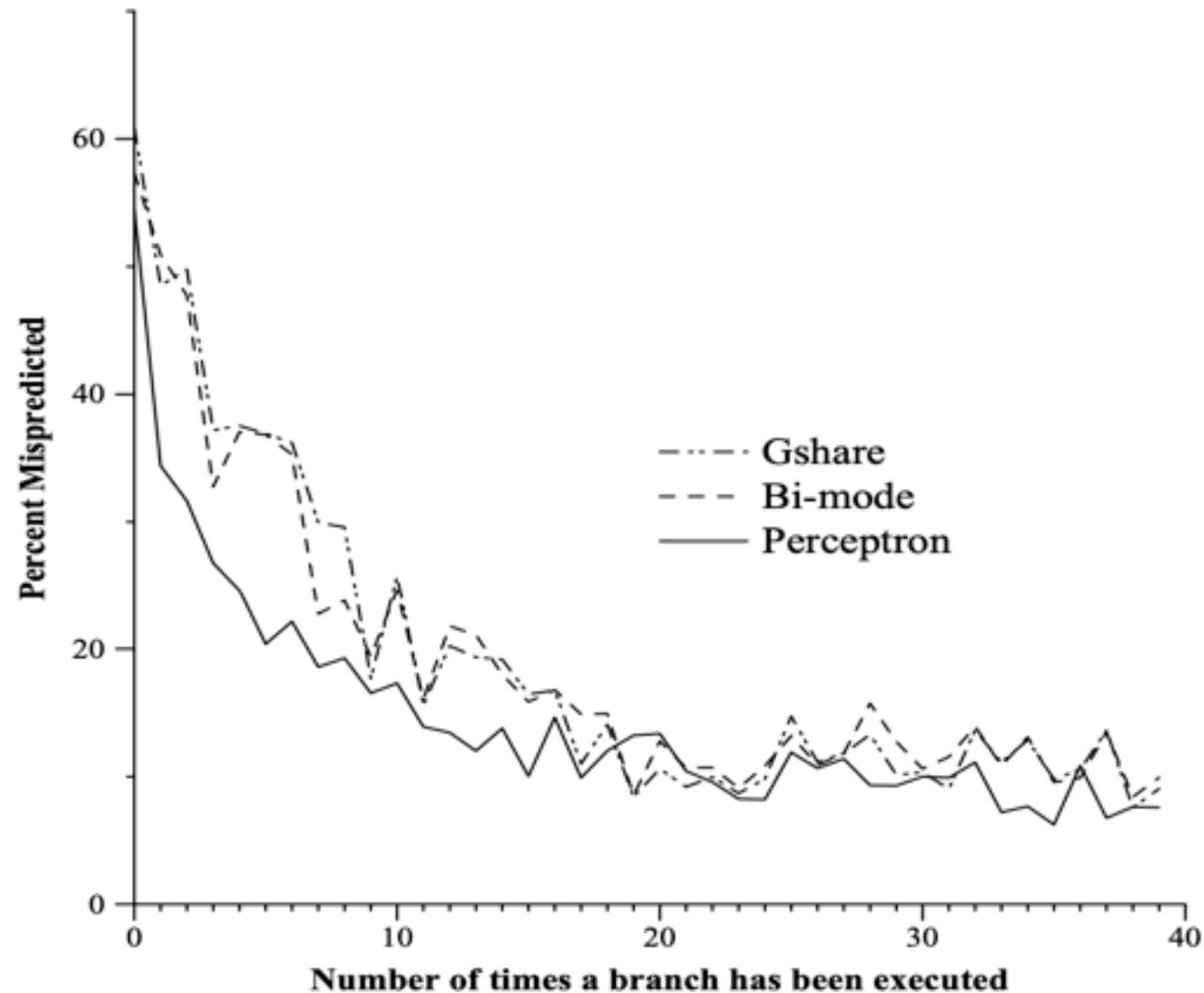


Figure 5: Misprediction Rates at a 16K budget. *Gshare* outperforms the perceptron predictor only on *186.crafty*. The hybrid predictor is consistently better than the PHT schemes.

History/training for perceptrons



Hardware budget in kilobytes	History Length		
	<i>gshare</i>	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.

What modern processors use?

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.

The branch capacity in Zen 2 is nearly double that of Zen. The L0 BTB was increased from 8 to 16 entries. The L1 BTB was increased from 256 to 512 entries. The L2 BTB was increased from 4096 to 7168 entries. The indirect target array was increased from 512 to 1024 entries. The combination of improved conditional predictor and increased branch capacity allows Zen 2 to target a 30% lower mispredict rate than Zen.

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

```
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++;
    }
}
```

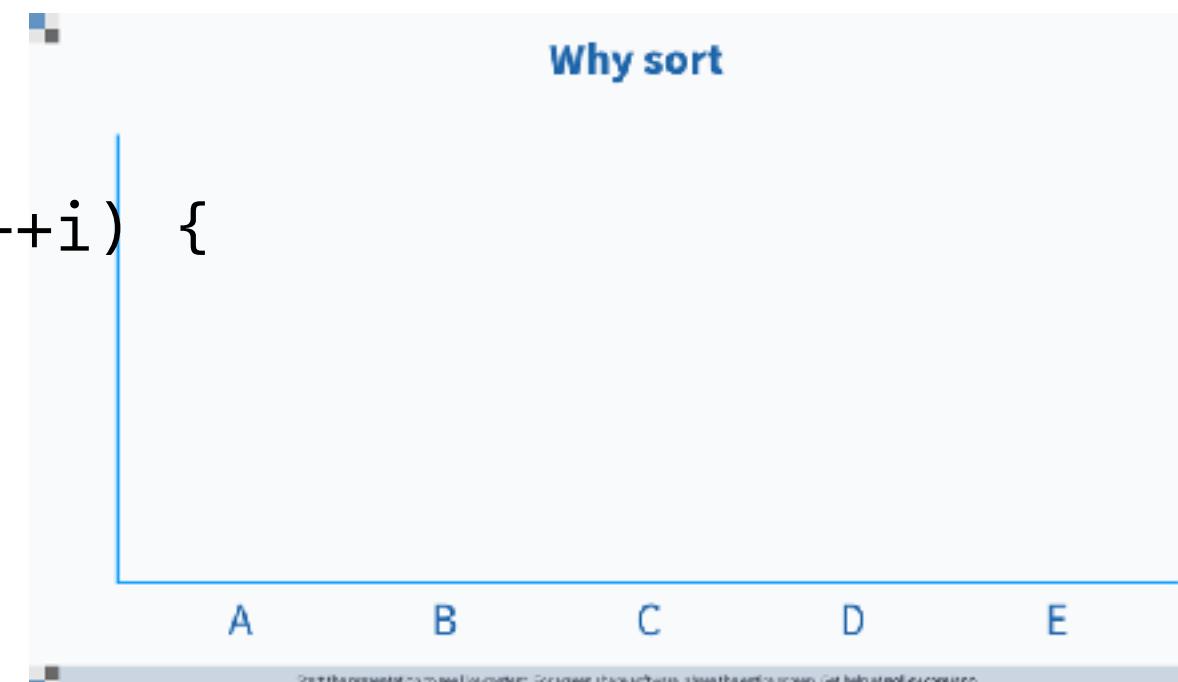
option = 1 is faster!!!

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 if(option)
      std::sort(data, data + arraySize);
B. 1
C. 2   for (unsigned i = 0; i < 100000; ++i) {
D. 3       int threshold = std::rand();
E. 4       for (unsigned i = 0; i < arraySize; ++i) {
          if (data[i] >= threshold)
              sum++;
}
```



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 if(option)
    std::sort(data, data + arraySize);
B. 1 for (unsigned i = 0; i < 100000; ++i) {
C. 2     int threshold = std::rand();
D. 3     for (unsigned i = 0; i < arraySize; ++i)
E. 4         if (data[i] >= threshold) branch X
F. 5         sum++;
G. 6 }
```

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

Why is branch mis-prediction expensive?

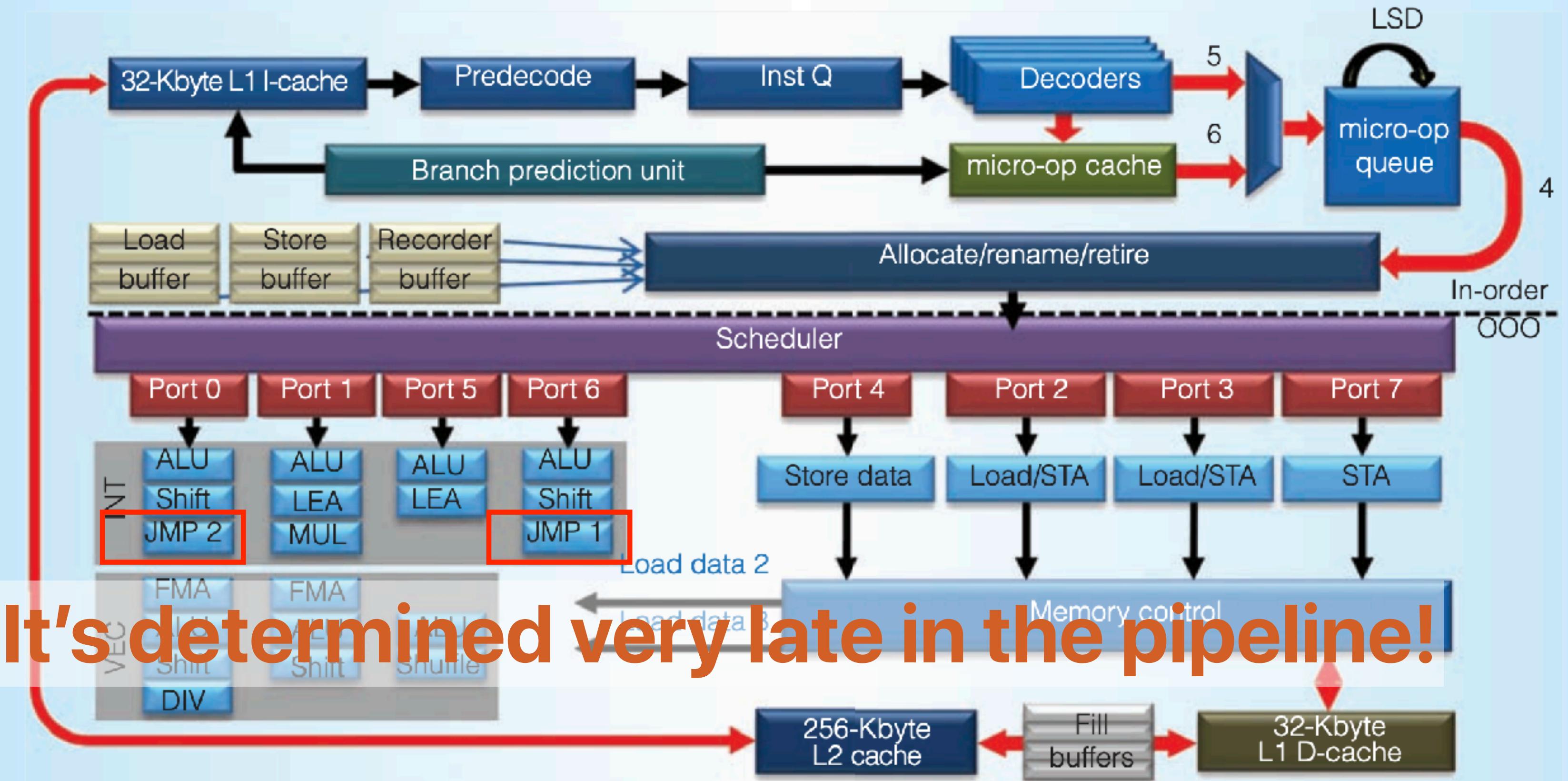


Figure 4. Skylake core block diagram.

It's determined very late in the pipeline!

Why is branch mis-prediction expensive?

- The pipeline of modern processors are very deep
- “Flushing pipeline” is actually not free
 - You need to ramp up the instruction cache again

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?
 - Version A
 - Version B
 - They are about the same (sometimes A is faster, sometimes B is)

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

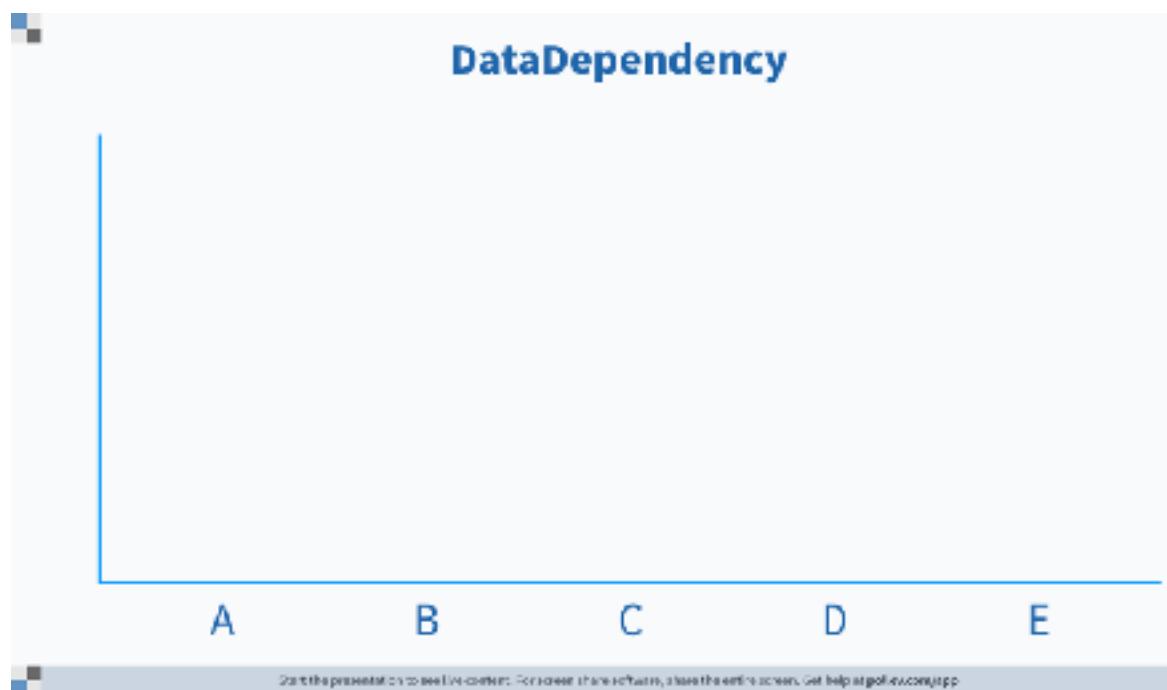
How many dependencies do we have?

- 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

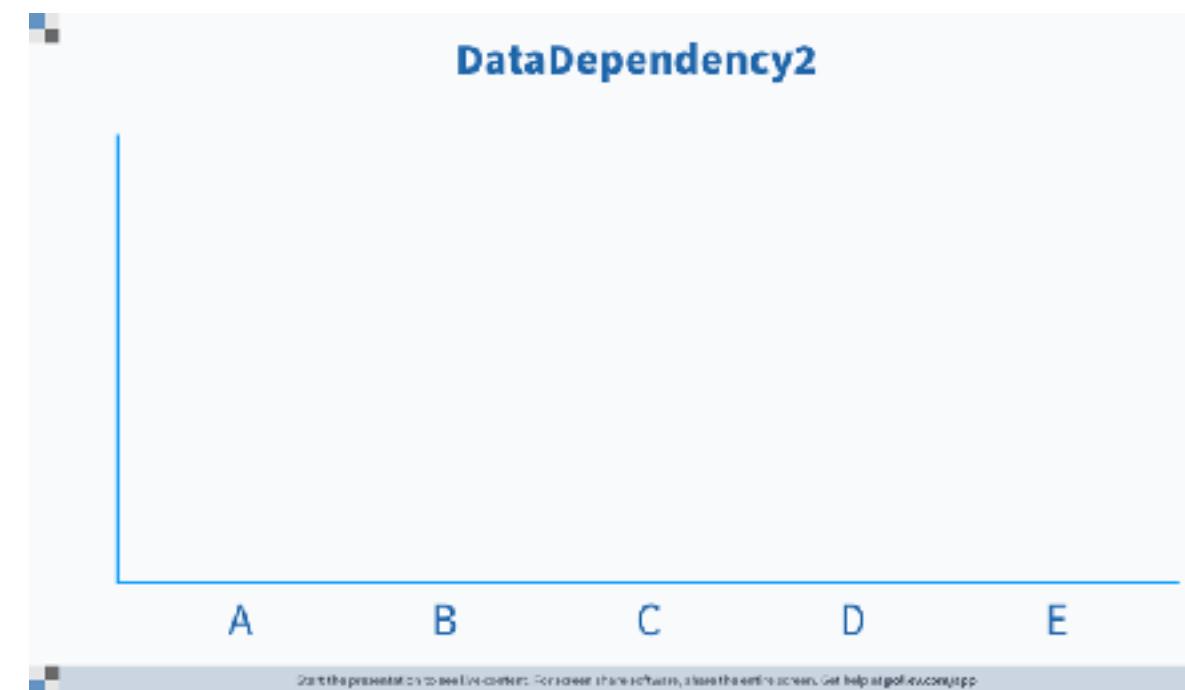
How many dependencies do we have?

- 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



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

Announcement

- Assignment 3 is already up
 - A total of 18 questions to answer
 - Also a programming assignment
 - Please do not expect any last minute help
- Reading quiz due next Tuesday
- Pick up your midterm
 - Outside of WCH 406
 - Please respect others when you pick up
 - Don't mess up the order
 - Don't disturb people who are working in this area

Computer Science & Engineering

203

つづく

