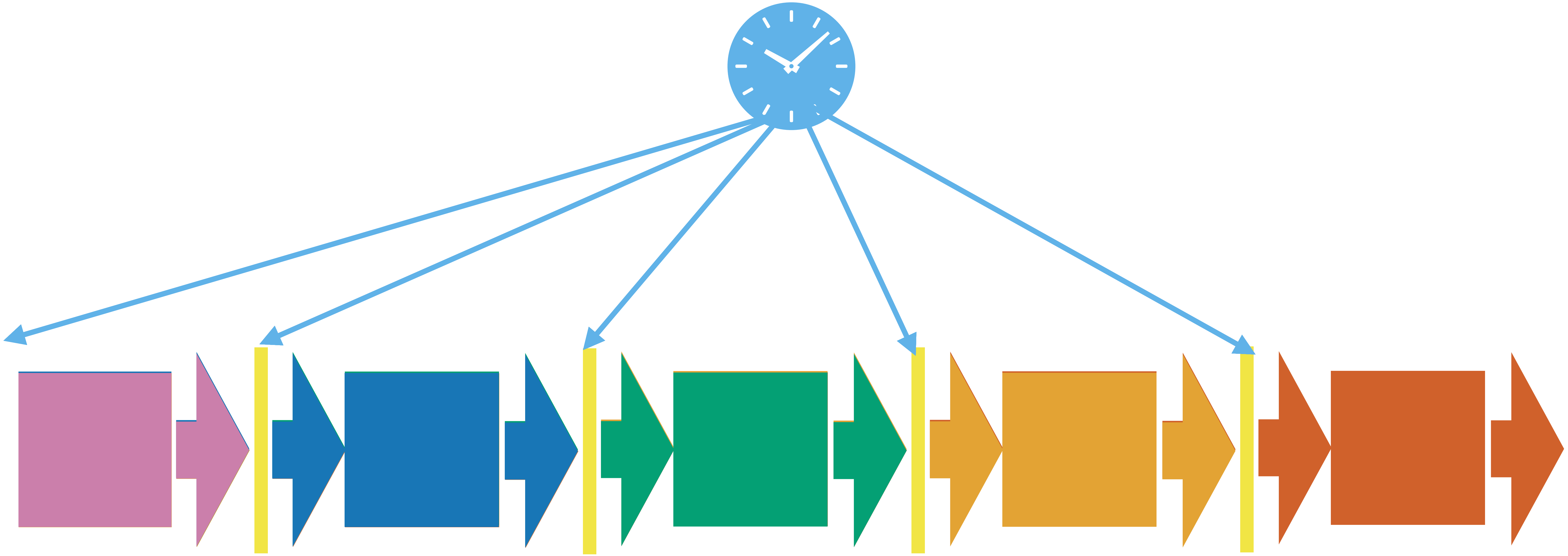
Dynamic Branch Prediction

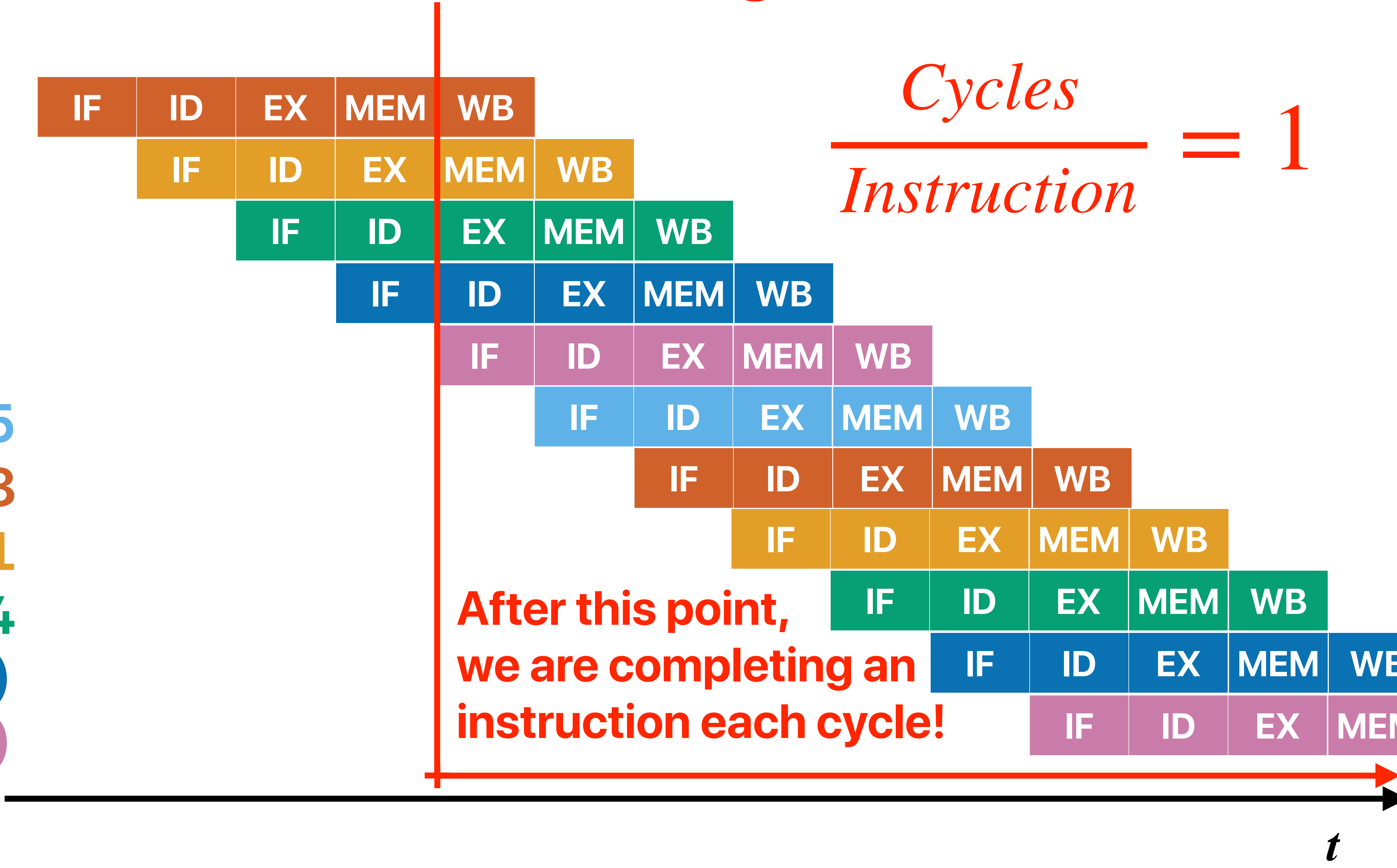
Hung-Wei Tseng

Recap: Pipelining



Recap: Pipelining

```
add x1,  x2,  x3
ld  x4,  0(x5)
sub x6,  x7,  x8
sub x9,x10,x11
sd  x1,  0(x12)
xor x13,x14,x15
and x16,x17,x18
add x19,x20,x21
sub x22,x23,x24
ld  x25, 4(x26)
sd  x27, 0(x28)
```



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

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

Recap: Tips of drawing a pipeline diagram

- Each instruction has to go through all 5 pipeline stages: IF, ID, EXE, MEM, WB in order — only valid if it's single-issue, RISC-V 5-stage pipeline
- An instruction can enter the next pipeline stage in the next cycle if
  - No other instruction is occupying the next stage
  - This instruction has completed its own work in the current stage
  - The next stage has all its inputs ready and it can retrieve those inputs
- Fetch a new instruction only if
  - We know the next PC to fetch
  - We can predict the next PC
  - Flush an instruction if the branch resolution says it's mis-predicted.

Recap: Solving Structural Hazards

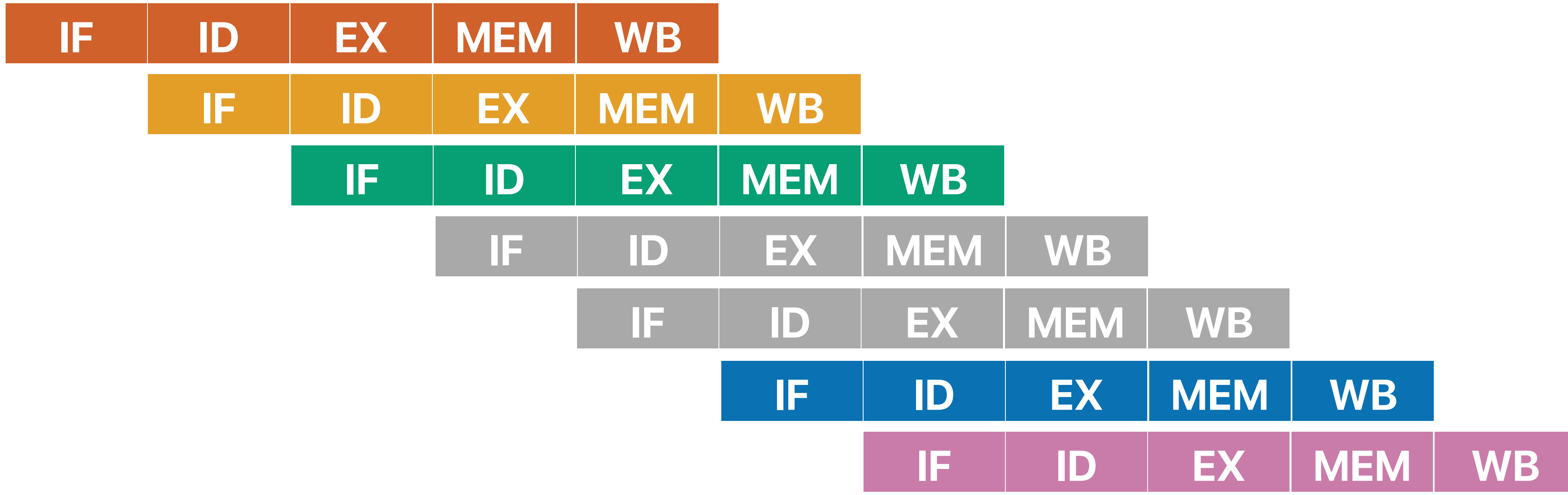
- Stall can address the issue — but slow
- Improve the pipeline unit design to allow parallel execution

Recap: The impact of control hazards

- Assuming that we have an application with 20% of branch instructions and the instruction stream incurs no data hazards. When there is a branch, we disable the instruction fetch and insert no-ops until we can determine the PC. What's the average CPI if we execute this program on the 5-stage RISC-V pipeline?

A. 1
B. 1.2
C. 1.4
D. 1.6
E. 1.8

```
add x1, x2, x3
ld  x4, 0(x5)
bne x0, x7, L
add x0, x0, x0
add x0, x0, x0
sub x9,x10,x11
sd  x1, 0(x12)
```



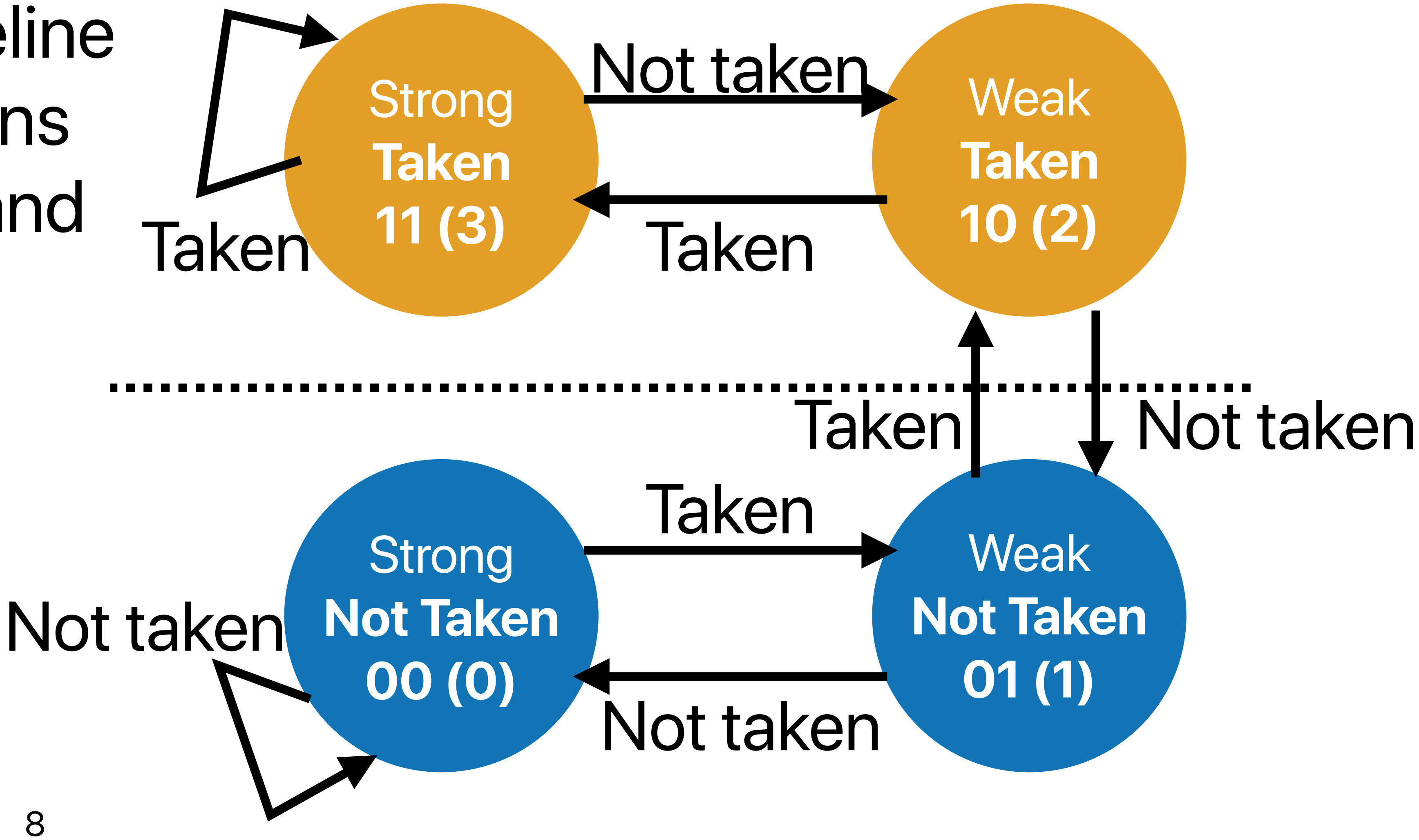
$$1 + 20\% \times 2 = 1.4$$

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 (row 0x400080)



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%

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

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

Outline

- 2-level global predictor
- Hybrid predictors
- Perceptrons
- Branch and coding

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
```

This pattern repeats all the time!

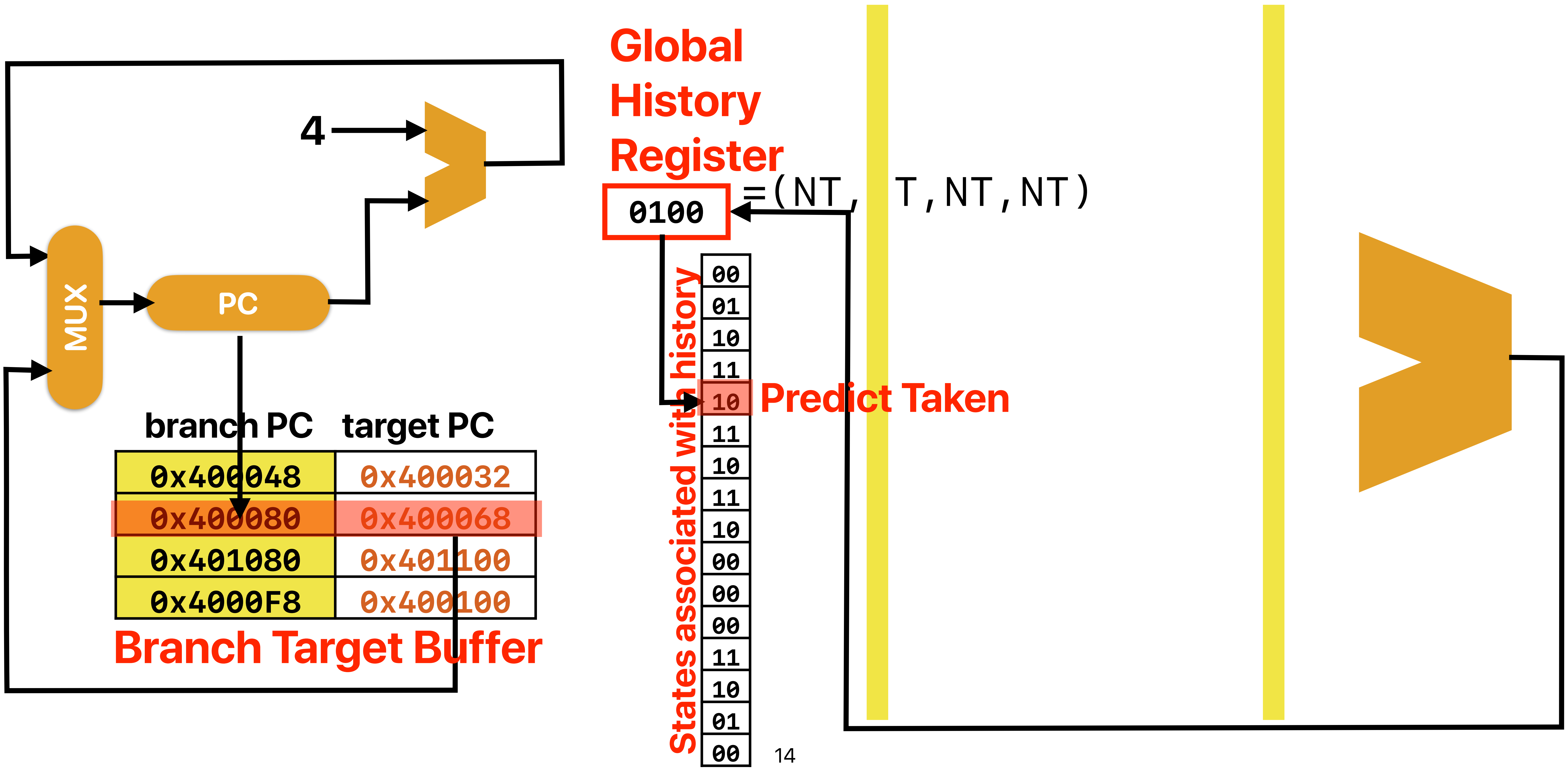
(assume all states started with 00)

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

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

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

Global history (GH) predictor



Performance of GH predictor

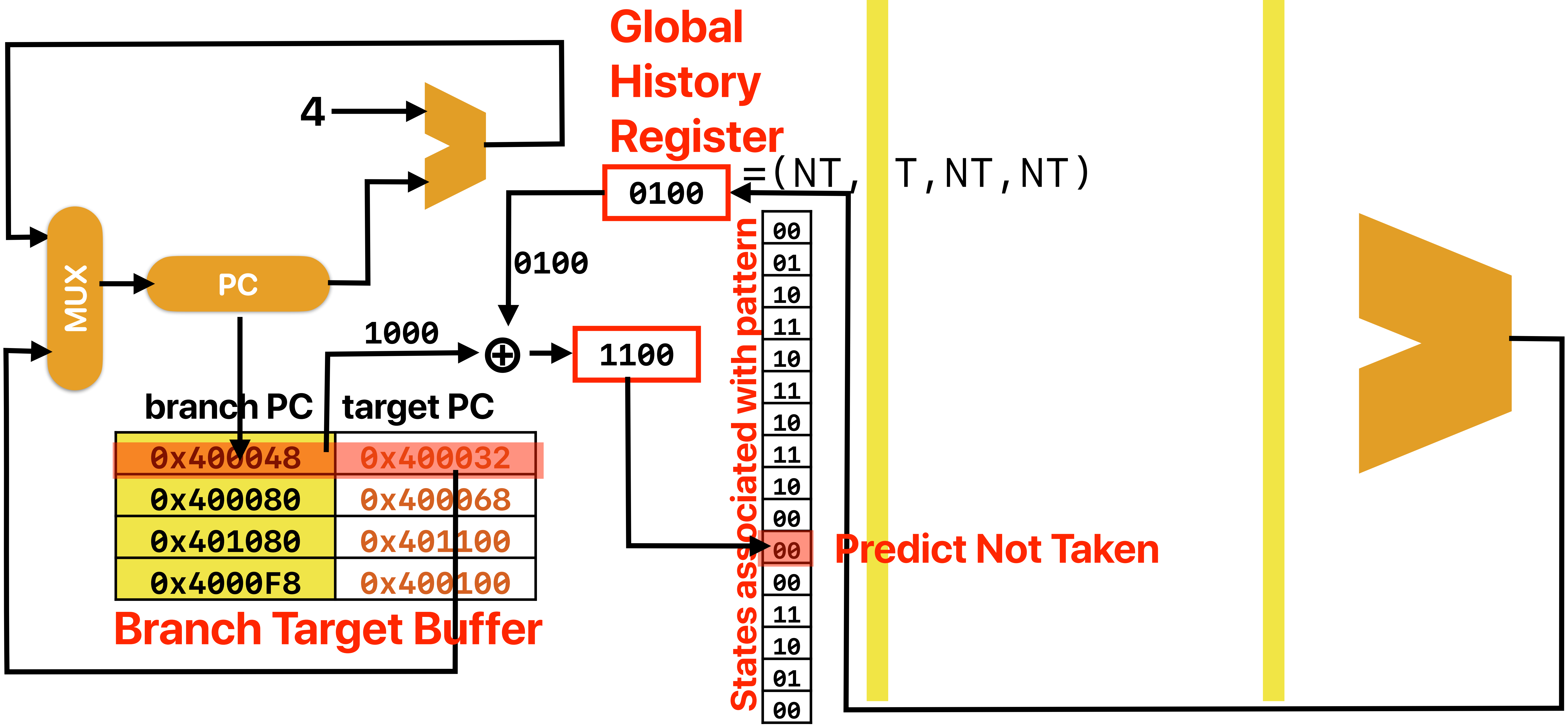
```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```

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

Near perfect after this

Hybrid predictors

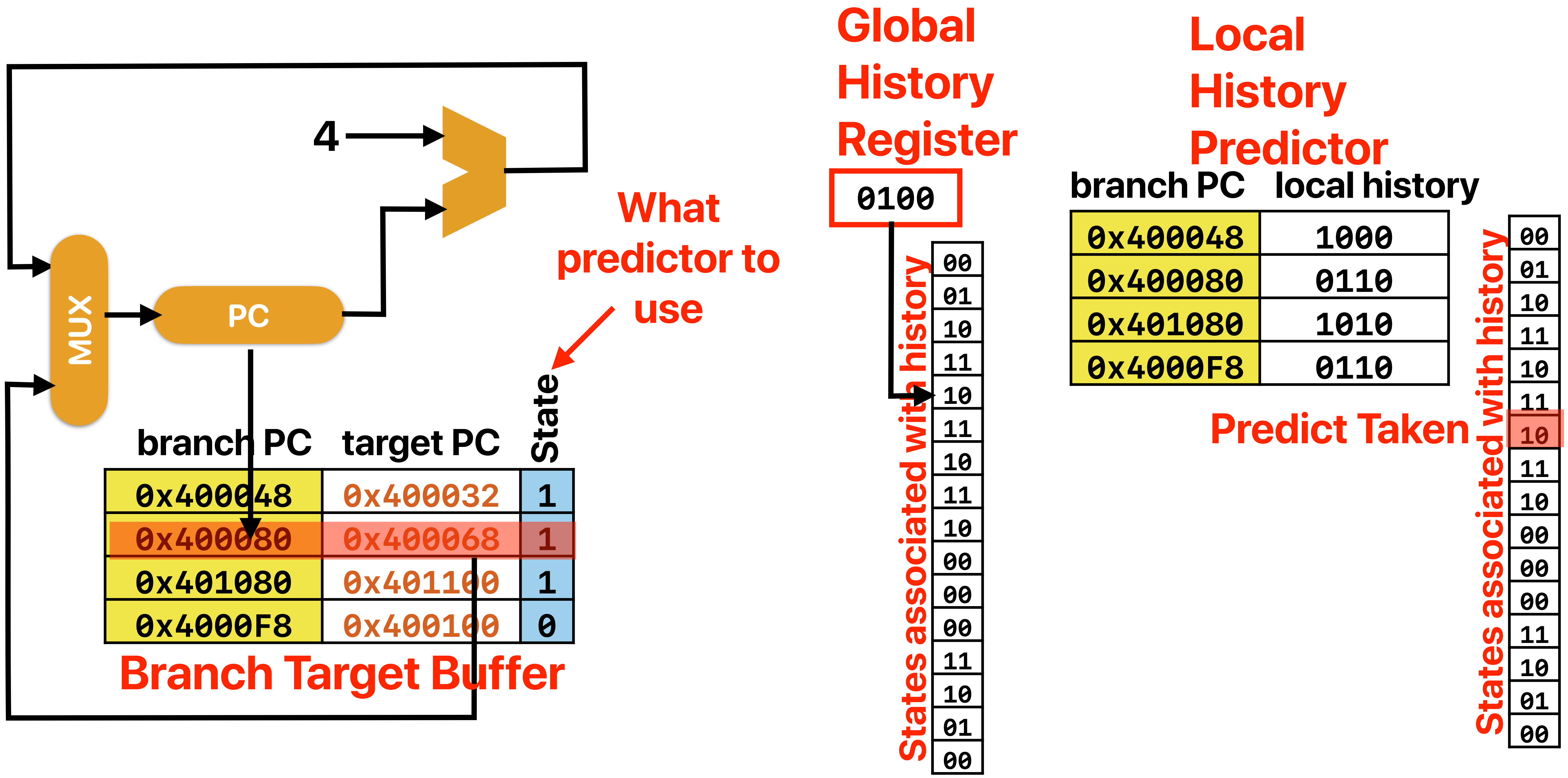
gshare predictor



gshare predictor

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

Tournament Predictor



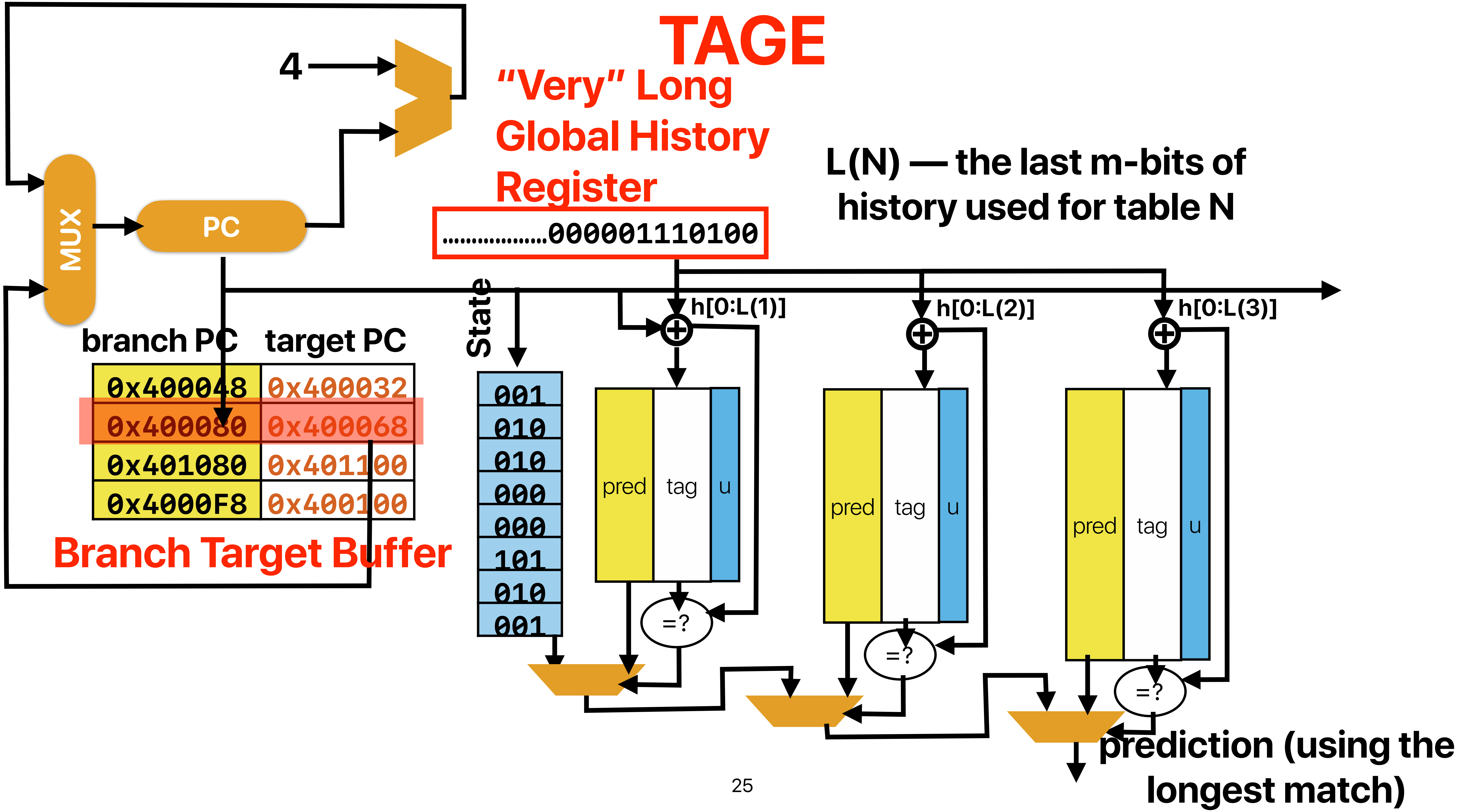
Tournament Predictor

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

TAGE

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

TAGE



4
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
State
001
010
010
000
000
101
010
001
h[0:L(1)]
h[0:L(2)]
h[0:L(3)]
pred
tag
u
=?
prediction (using the
longest match)

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

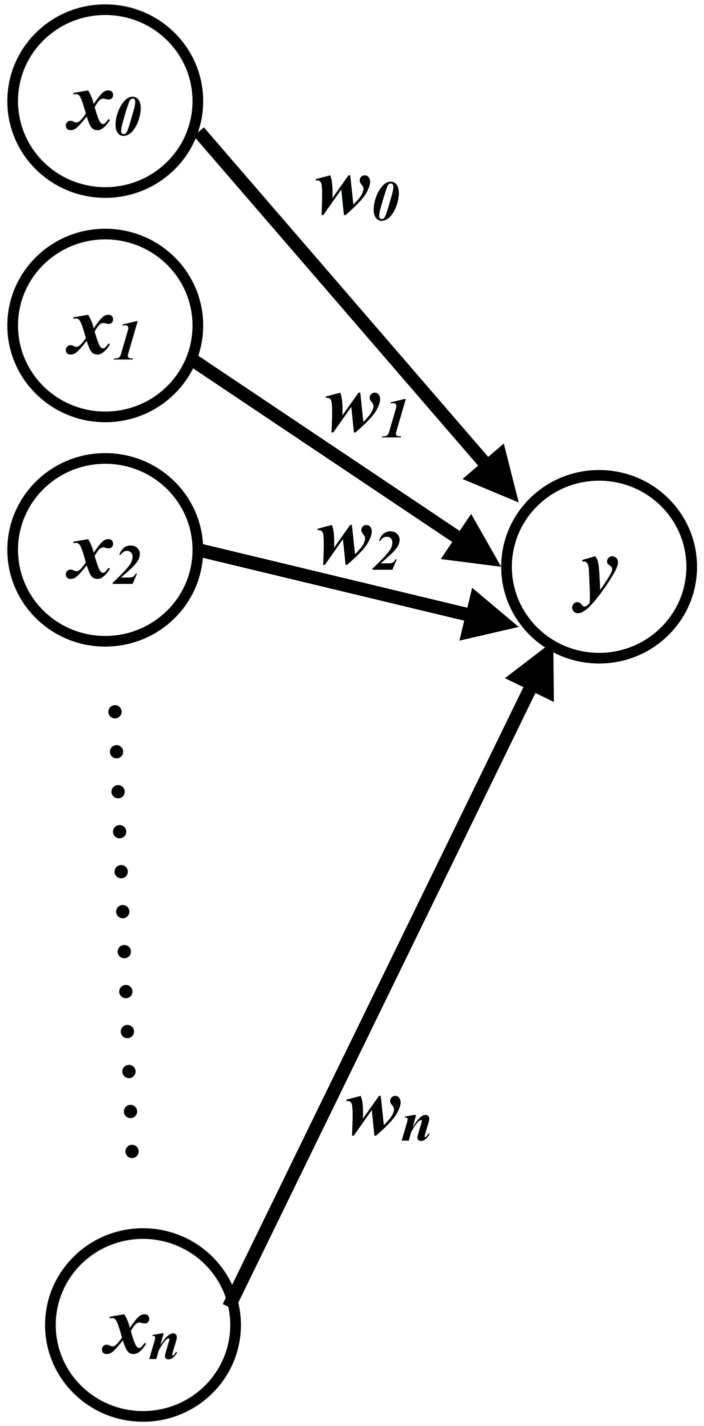
- The machine learns to predict conditional branches
- Artificial neural networks
  - Simple model of neural networks in brain cells
  - Learn to recognize and classify patterns

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



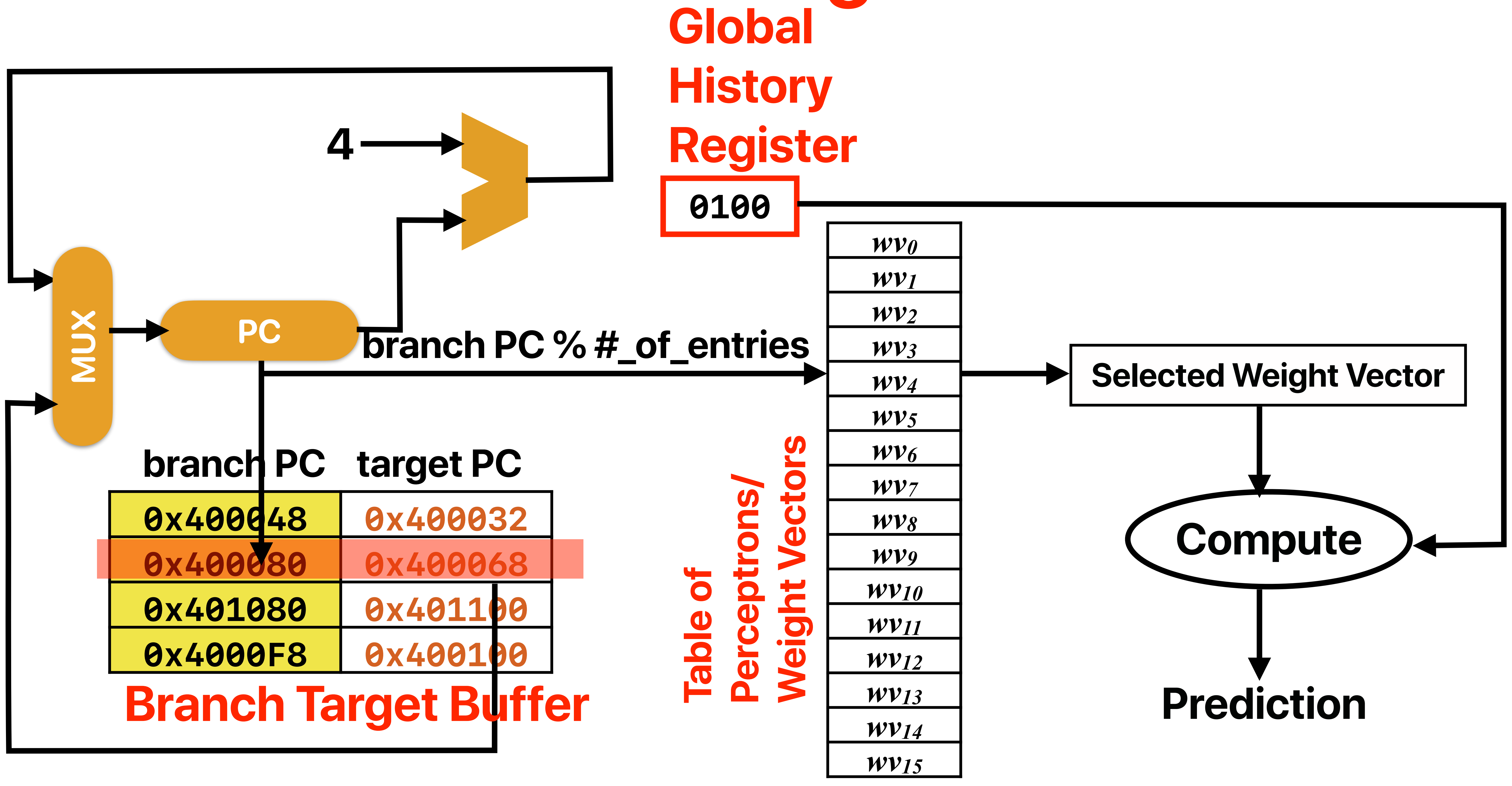
$$y = w_0 + \sum_{i=1}^{n} x_i w_i$$

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| ≤ 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



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

- Why the sorting the array speed up the code despite the increased instruction count?

```
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 ++;
    }
}
```

Demo: Popcount

- The population count (or popcount) of a specific value is the number of set bits (i.e., bits in 1s) in that value.
- Applications
  - Parity bits in error correction/detection code
  - Cryptography
  - Sparse matrix
  - Molecular Fingerprinting
  - Implementation of some succinct data structures like bit vectors and wavelet trees.

Demo: pop count

- Given a 64-bit integer number, find the number of 1s in its binary representation.
- Example 1:
  Input: 9487
  Output: 7
  Explanation: 9487's binary representation is 0b10010100001111

```
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;
}
```

Hardware acceleration

- Because popcount is important, both intel and AMD added a POPCNT instruction in their processors with SSE4.2 and SSE4a
- In C/C++, you may use the intrinsic "_mm_popcnt_u64" to get # of "1"s in an unsigned 64-bit number
  - You need to compile the program with -m64 -msse4.2 flags to enable these new features

```
#include <smmintrin.h>
inline int popcount(uint64_t x) {
     int c = _mm_popcnt_u64(x);
     return c;
}
```

Computer
Science &
Engineering

203

つづく