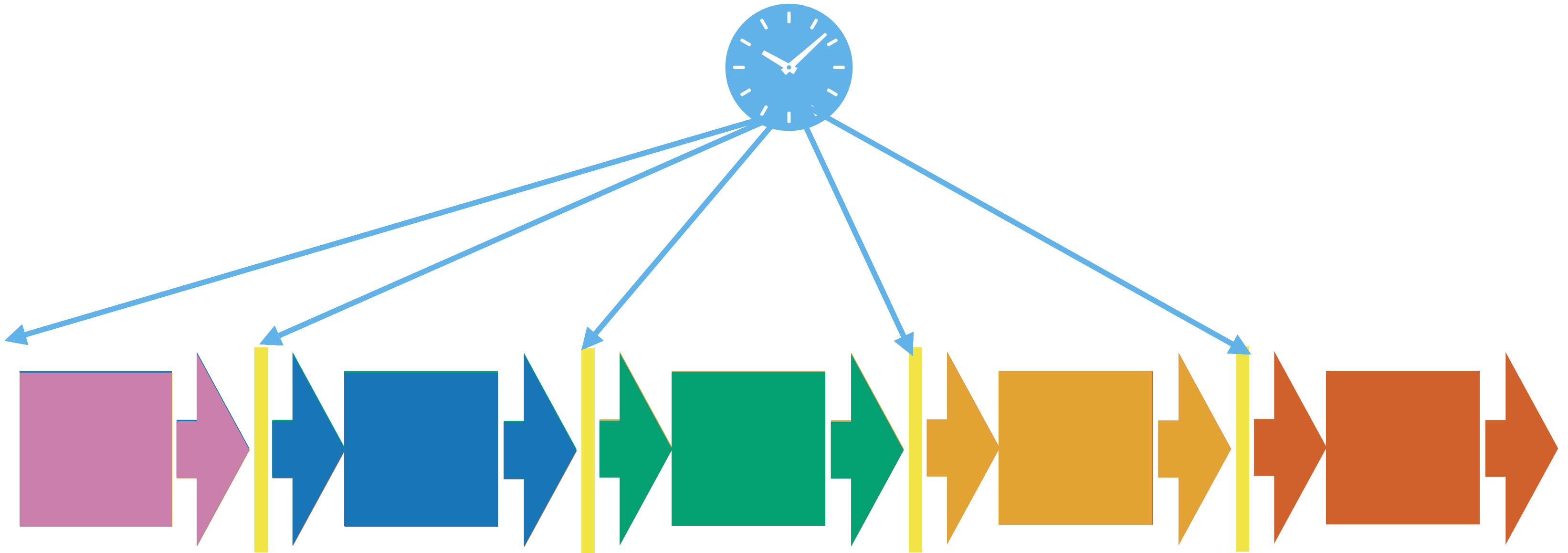


# Data Hazards & Dynamic Instruction Scheduling (I)

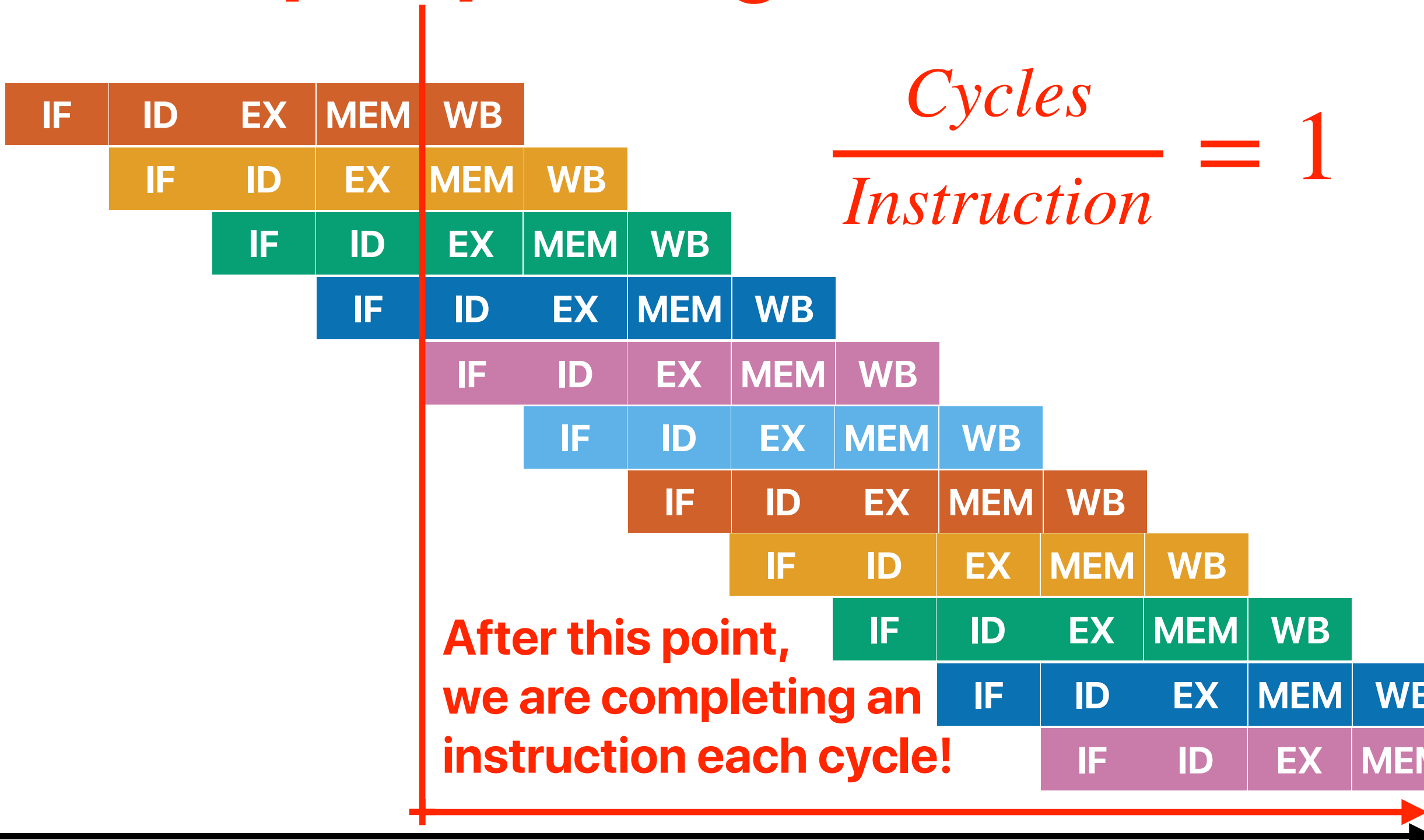
Hung-Wei Tseng

# 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)
```



# 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.
- Review your undergraduate architecture materials
  - [http://cseweb.ucsd.edu/classes/su19\\_2/cse141-a/](http://cseweb.ucsd.edu/classes/su19_2/cse141-a/)

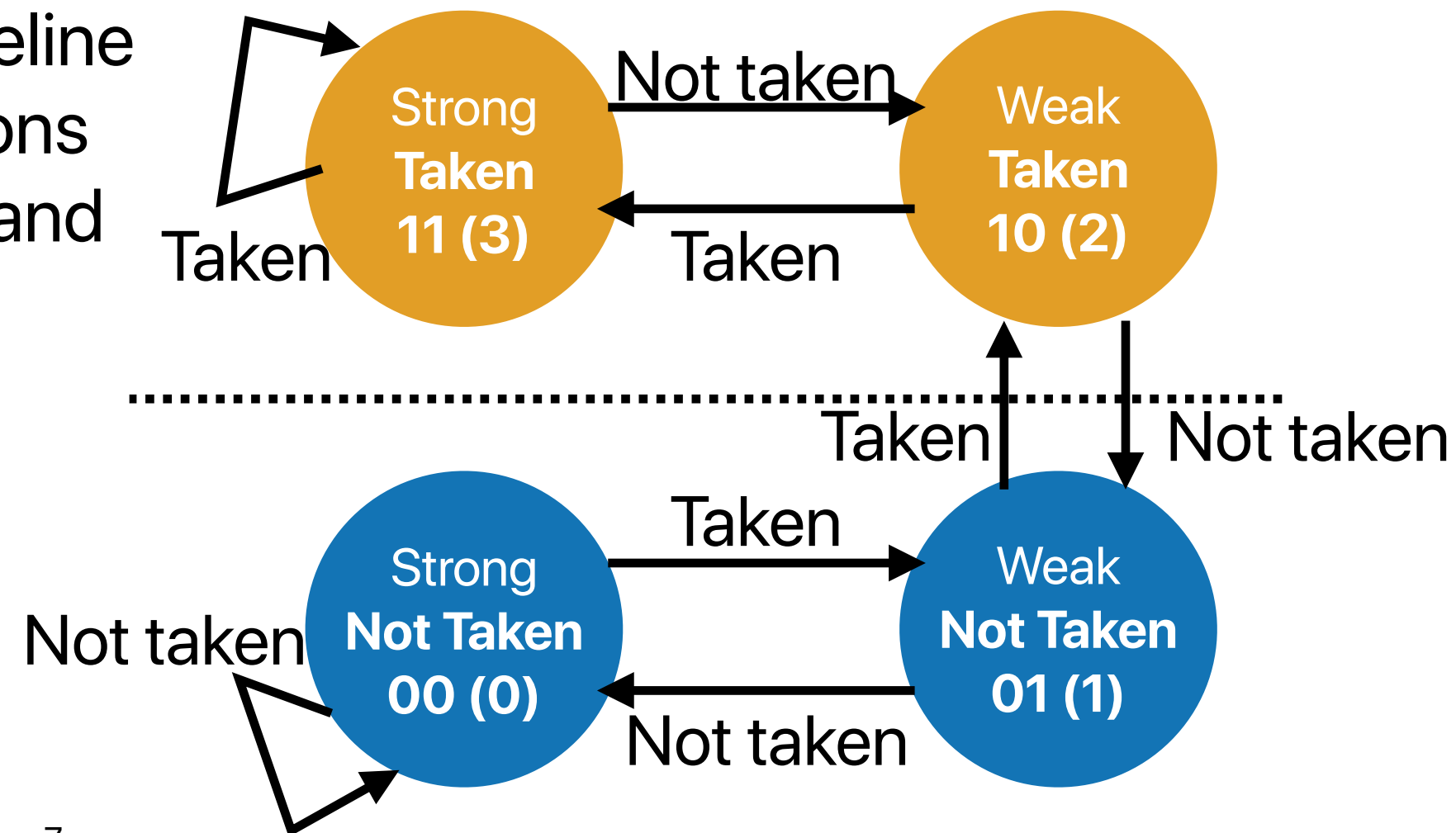
# Recap: addressing hazards

- Structural hazards
  - Stall
  - Modify hardware design
- Control hazards
  - Stall
  - Static prediction
  - Dynamic prediction

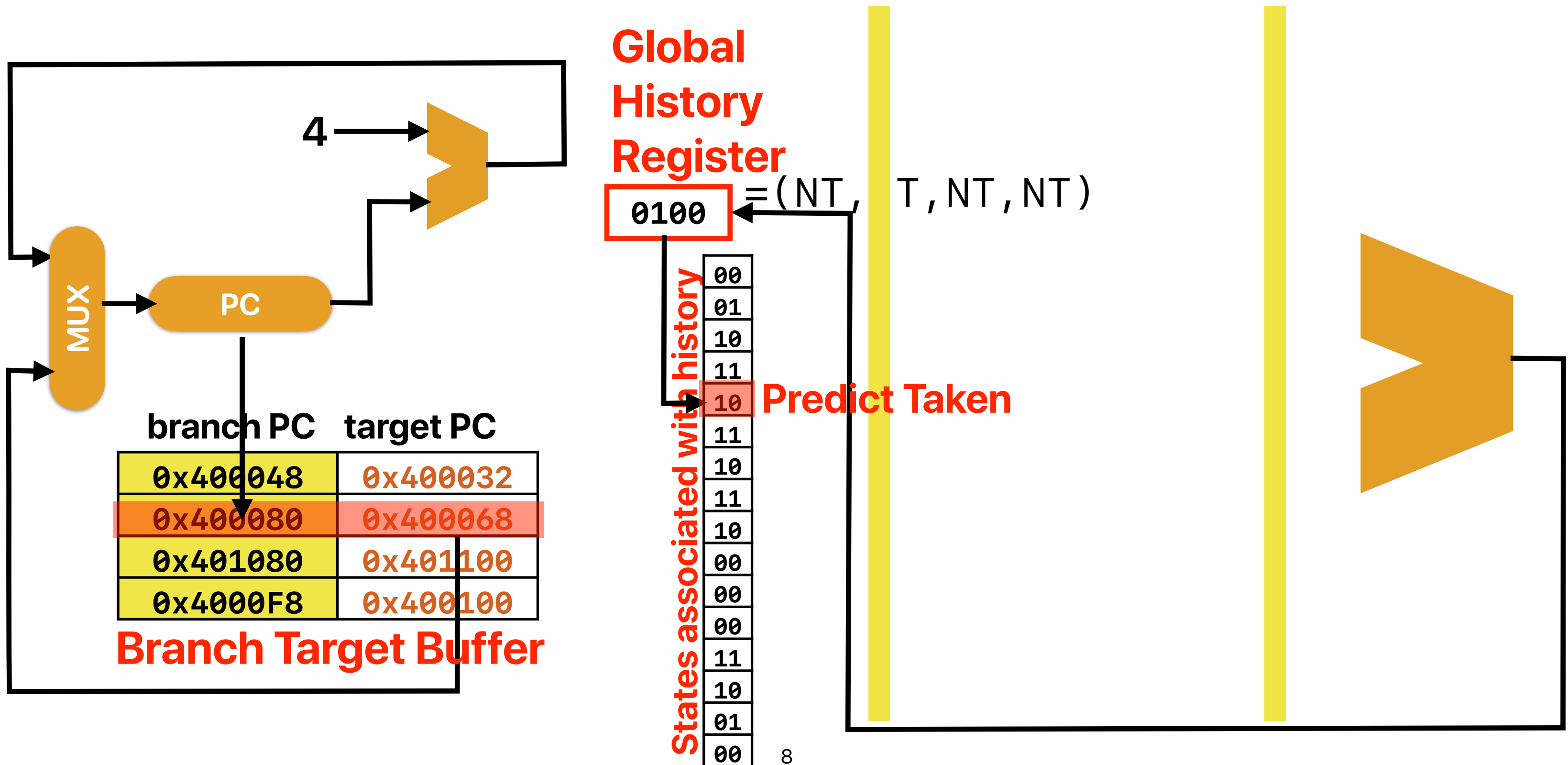
# Recap: 2-bit/Bimodal local predictor

- 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
- 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
Predict Taken	0x400080	0x400068	11
	0x401080	0x401100	00
	0x4000F8	0x400100	01

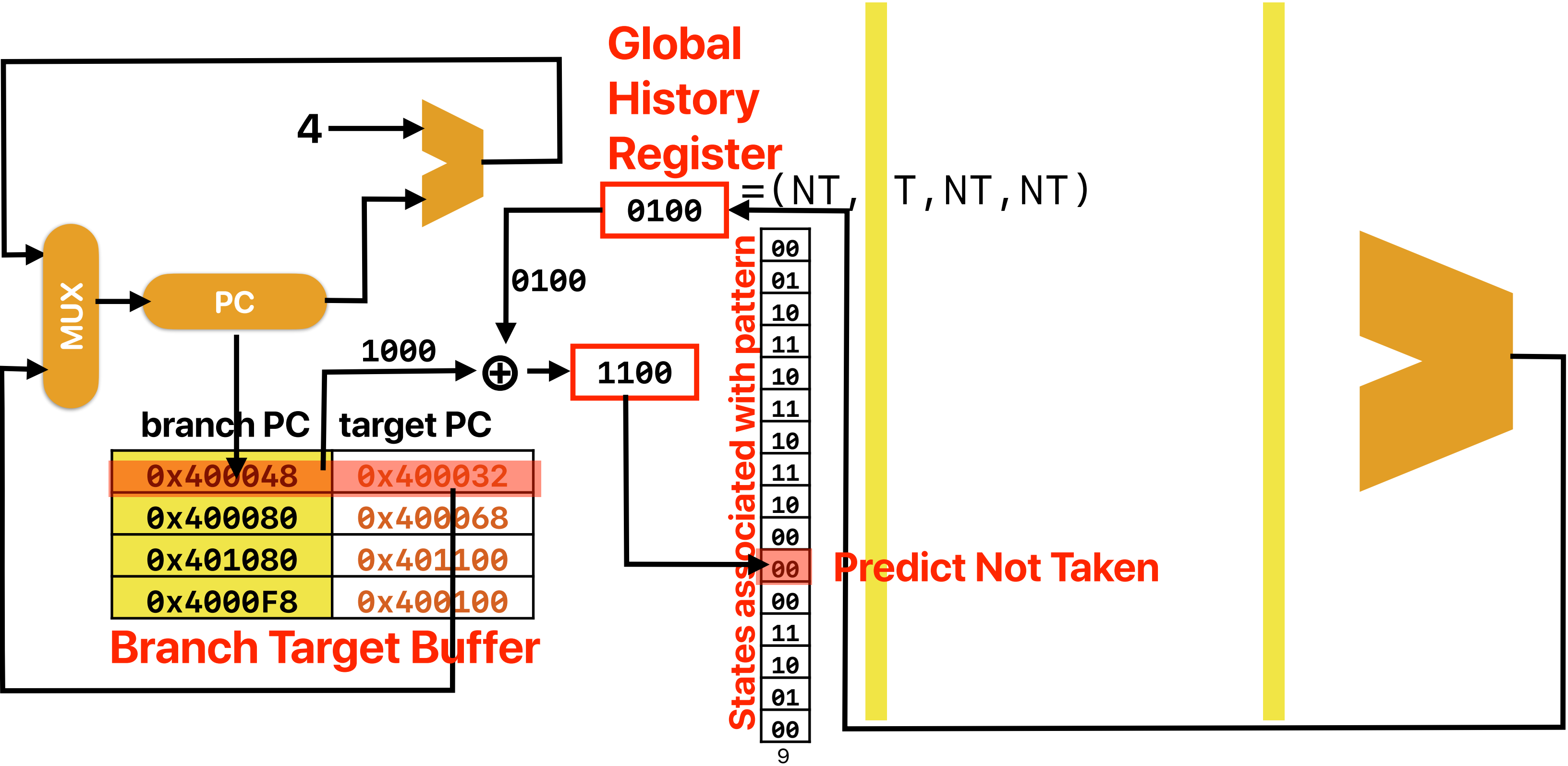


# Recap: Global history (GH) predictor

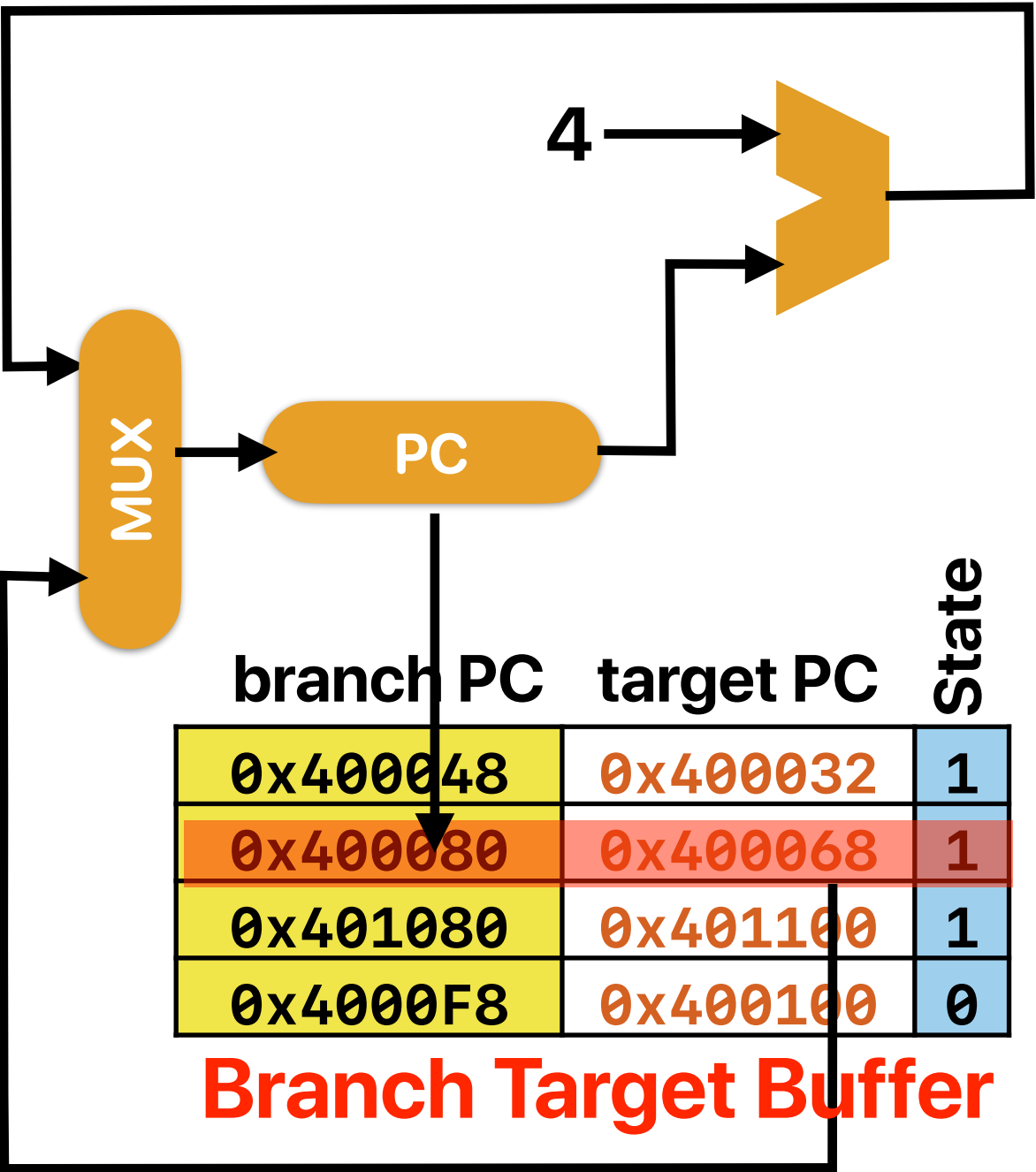




# Recap: gshare predictor



# Recap: tournament Predictor



Branch Target Buffer

Global  
History  
Register

0100

States associated with history

00
01
10
11
10
11
10
11
10
11
10
00
00
00
00
00
11
10
10
01
00

Local  
History  
Predictor

branch PC local history

0x400048	1000
0x400080	0110
0x401080	1010
0x4000F8	0110

Predict Taken

States associated with history

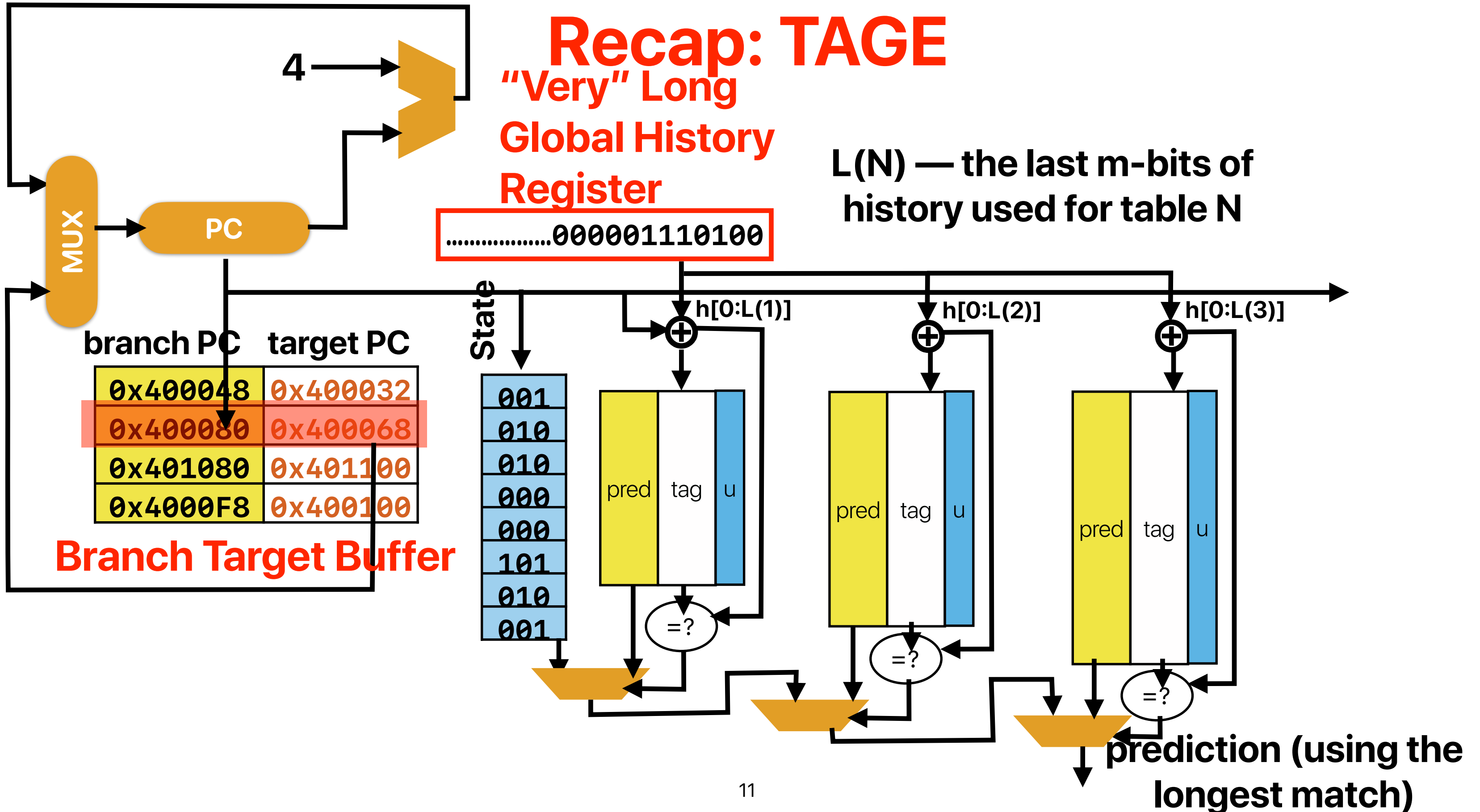
00
01
10
11
10
11
10
11
10
11
10
00
00
00
00
00
11
10
10
01
00

# Recap: TAGE

"Very" Long  
Global History  
Register

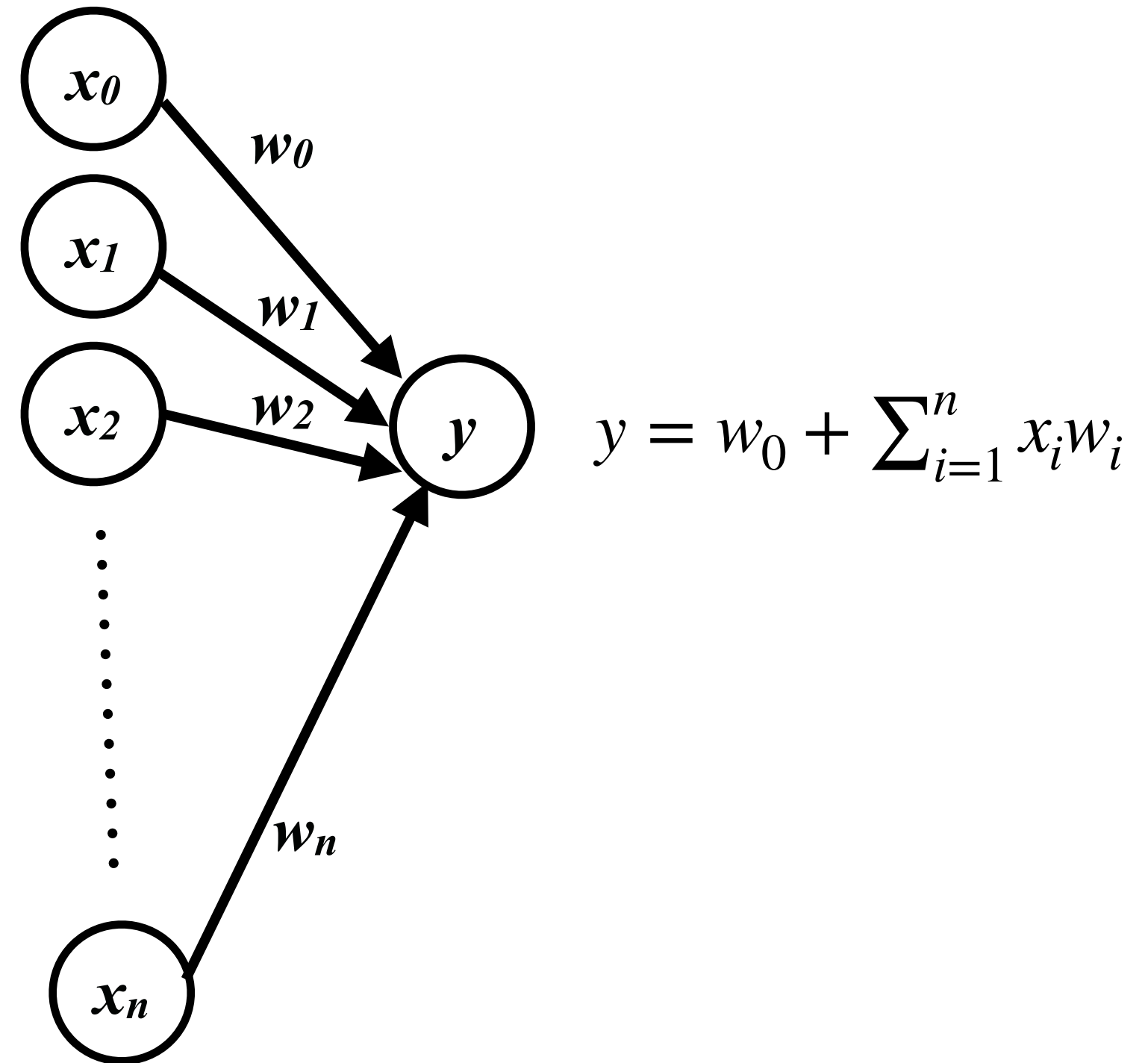
$L(N)$  — the last  $m$ -bits of  
history used for table  $N$

.....000001110100



# Recap: 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  $y \geq 0$
- Training finds correlations between history and outcome



# Four 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;
}
```

**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;
}
```

**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;
}
```



# Team scores



6



10.5



7



6

# Outline

- Branch prediction and performance (cont.)
- Data hazards
- Tomasulo's algorithm

# 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 memory accesses

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?

- 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 memory accesses

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;  
}
```

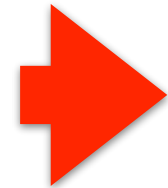
# Why is B better than A?

A

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

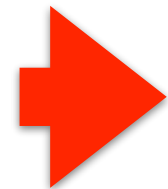
B

```
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;
}
```



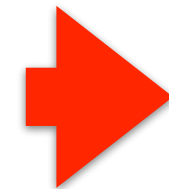
```
and    x2, x1, 1
add    x3, x3, x2
shr    x1, x1, 1
bne    x1, x0, LOOP
```

**4\*n instructions**



```
and    x2, x1, 1
add    x3, x3, x2
shr    x1, x1, 1
and    x2, x1, 1
add    x3, x3, x2
shr    x1, x1, 1
and    x2, x1, 1
add    x3, x3, x2
shr    x1, x1, 1
bne    x1, x0, LOOP
```

**13\*(n/4) = 3.25\*n instructions**



```
and    x2, x1, 1
shr    x4, x1, 1
shr    x5, x1, 2
shr    x6, x1, 3
shr    x1, x1, 4
and    x7, x4, 1
and    x8, x5, 1
and    x9, x6, 1
add    x3, x3, x2
add    x3, x3, x7
add    x3, x3, x8
add    x3, x3, x9
bne    x1, x0, LOOP
```

18 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 memory accesses

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;  
    }  
    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 memory accesses

A. 0

B. 1

C. 2

D. 3

E. 4



```
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;  
}
```



```
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 memory accesses

A. 0

B. 1

C. 2

D. 3

E. 4



```
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;  
}
```



```
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 memory accesses  
— Probably not. In fact, the load may have negative effect without architectural supports

A. 0

B. 1

C. 2

D. 3

E. 4

```
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;
}
```

```
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 D better than C?

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

A. 0

B. 1

C. 2

D. 3

E. 4



```
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;
}
```



```
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;
}
```





# Why is D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - D has lower dynamic instruction count than C
  - D has significantly lower branch mis-prediction rate than C
  - D has significantly fewer branch instructions than C
  - D can incur fewer memory accesses than C

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



```
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;
}
```



```
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;
}
```



# Why is D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations

- ① ✓ D has lower dynamic instruction count than C  
— Compiler can do loop unrolling — no branches
- ② ✓ D has significantly lower branch mis-prediction rate than C  
— Could be
- ③ ✓ D has significantly fewer branch instructions than C  
— maybe eliminated through loop unrolling...
- ④ D can incur fewer memory accesses than C  
— about the same

A. 0

B. 1

C. 2

D. 3

E. 4

```
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;  
}
```

```
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;  
}
```

# All branches are gone with loop unrolling

```

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};
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    return c;
}

```

26

# 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;
}
```

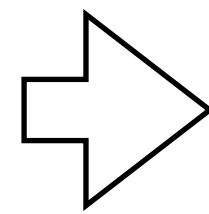
# Data hazards

# 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

# Example: vector scaling

```
i = 0;  
do {  
    vector[i] += scale;  
} while ( ++i < size )
```



```
                                shl    X5, X11, 3  
                                add    X5, X5, X10  
LOOP: ld    X6, 0(X10)  
                                add    X7, X6, X12  
                                sd     X7, 0(X10)  
                                addi   X10, X10, 8  
                                bne    X10, X5, LOOP
```

# How many dependencies do we have?

- How many pairs of data dependencies are there in the following RISC-V instructions?

```
ld      X6, 0(X10)
add     X7, X6, X12
sd      X7, 0(X10)
addi    X10, X10, 8
bne     X10, X5, LOOP
```

- 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 RISC-V instructions?

```
ld      X6, 0(X10)
add     X7, X6, X12
sd      X7, 0(X10)
addi    X10, X10, 8
bne     X10, X5, LOOP
```

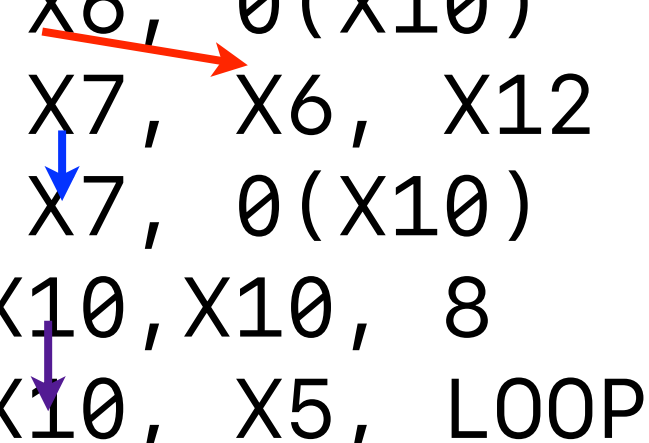
- A. 1
- B. 2
- C. 3
- D. 4
- E. 5



# How many dependencies do we have?

- How many pairs of data dependencies are there in the following RISC-V instructions?

```
ld      X6, 0(X10)
add     X7, X6, X12
sd      X7, 0(X10)
addi    X10, X10, 8
bne     X10, X5, LOOP
```



A. 1

B. 2

C. 3

D. 4

E. 5

# Solution 1: Let's try "stall" again

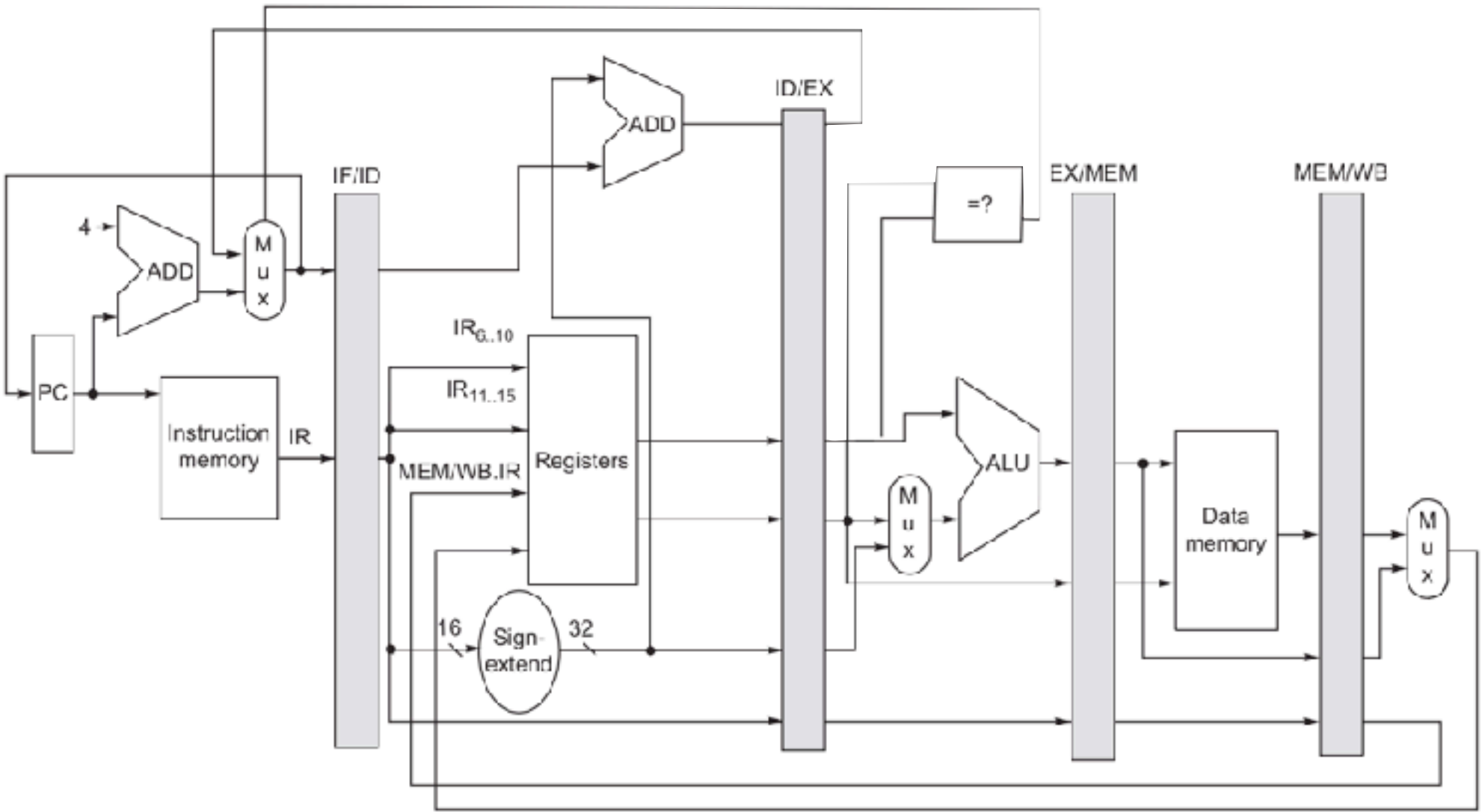
- Whenever the input is not ready when the consumer is decoding, just stall — the consumer stays at ID.

# How many of data hazards?

- How many pairs of instructions in the following RISC-V instructions will results in data hazards/stalls in a basic 5-stage RISC-V pipeline?

```
ld    X6, 0(X10)
add   X7, X6, X12
sd    X7, 0(X10)
addi  X10, X10, 8
bne   X10, X5, LOOP
```

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5



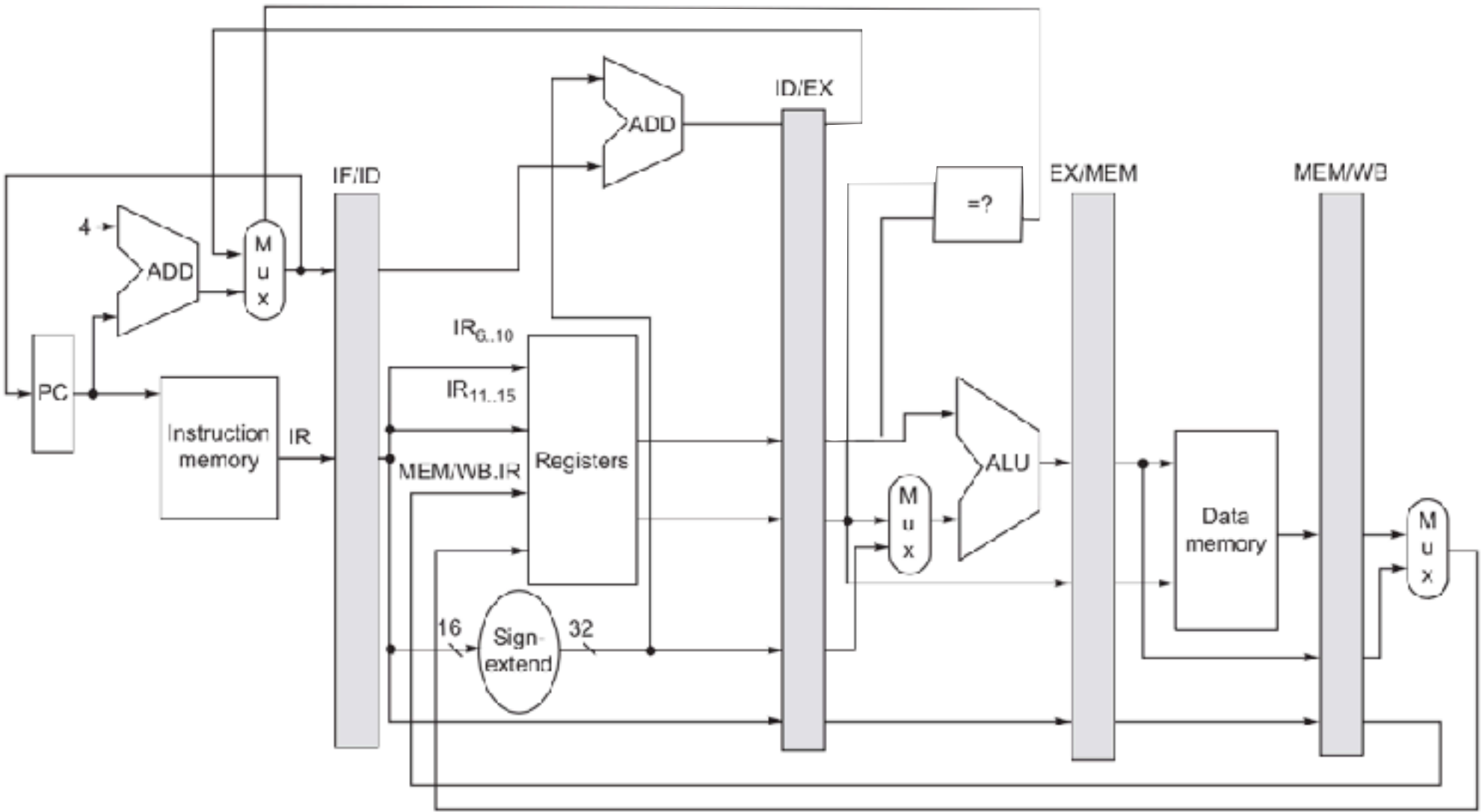


# How many of data hazards?

- How many pairs of instructions in the following RISC-V instructions will results in data hazards/stalls in a basic 5-stage RISC-V pipeline?

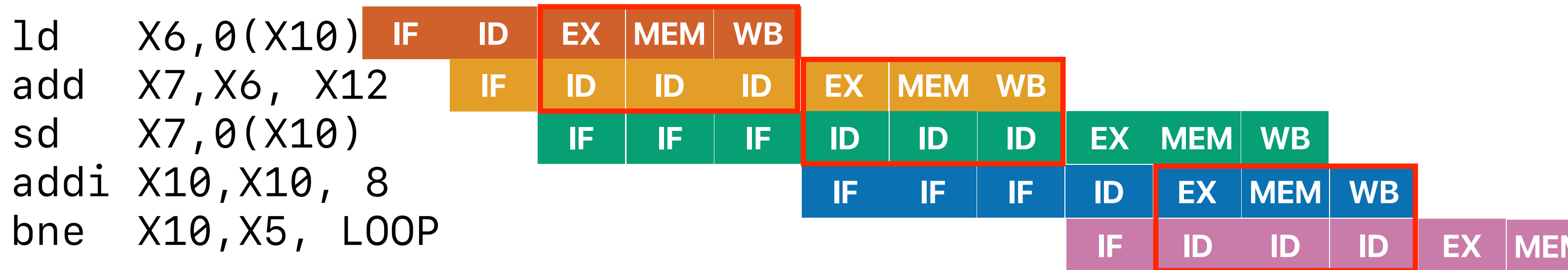
```
ld    X6, 0(X10)
add   X7, X6,  X12
sd    X7, 0(X10)
addi  X10, X10, 8
bne   X10, X5,  LOOP
```

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5



# How many of data hazards?

- How many pairs of instructions in the following RISC-V instructions will results in data hazards/stalls in a basic 5-stage RISC-V pipeline?

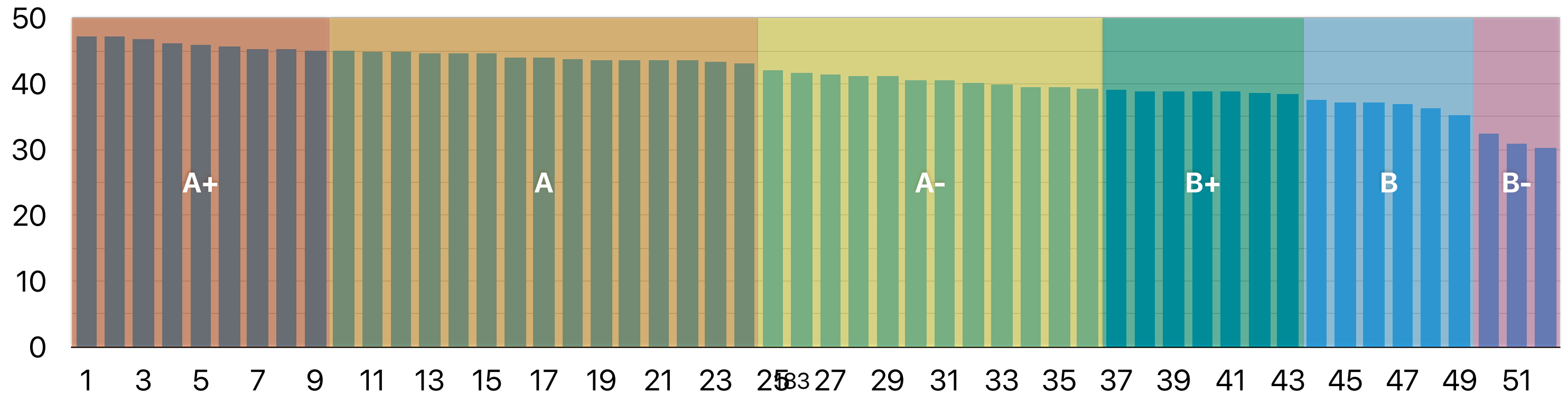


- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

# Announcement

- Project is up — check the website
- Assignment #4 is up — start EARLY!!!
- Office Hours on Zoom (the office hour link, not the lecture one)
  - Hung-Wei/Prof. Usagi: M 8p-9p, W 2p-3p
  - Quan Fan: F 1p-3p
- Regarding projected grades
  - Based on your “weighted total” column in iLearn — we only have 50% offered so far, that’s why the max is only 48 now
  - Our final grading is based on “relative ranking” and scale may change

Current “Weighted Total” in iLearn and “Projected” Letter Grades



# Computer Science & Engineering

# 203

# つづく

