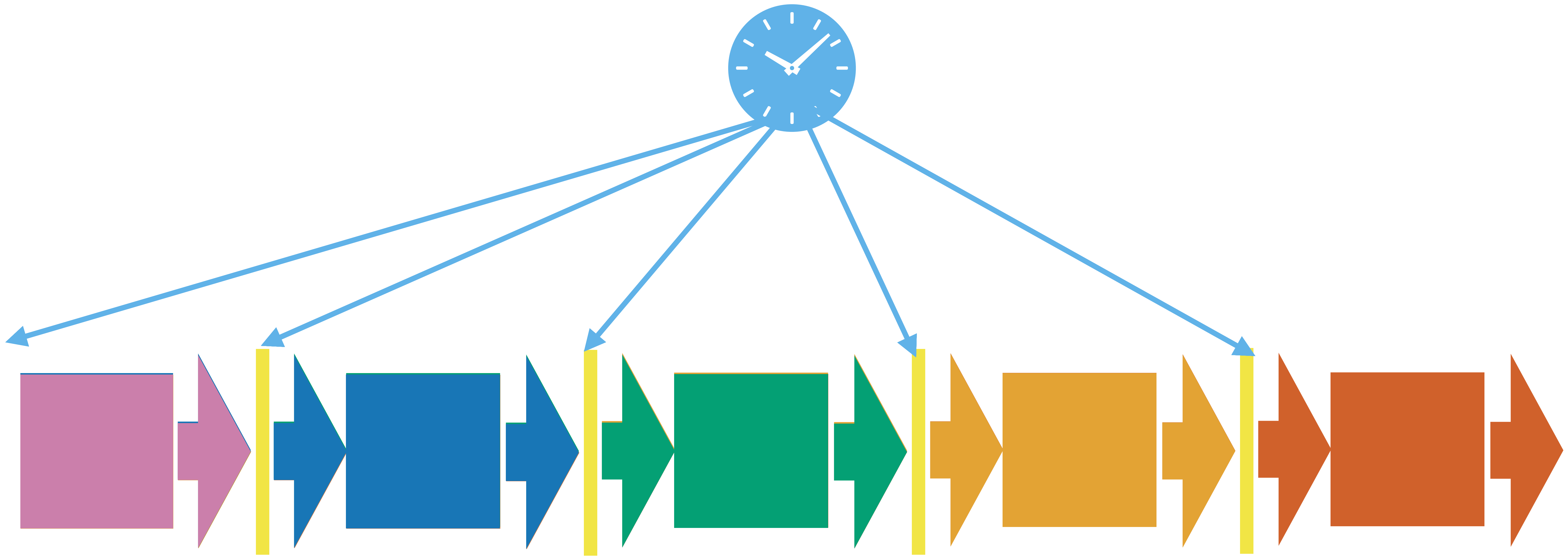
Data Hazards & Dynamic Instruction Scheduling (I)

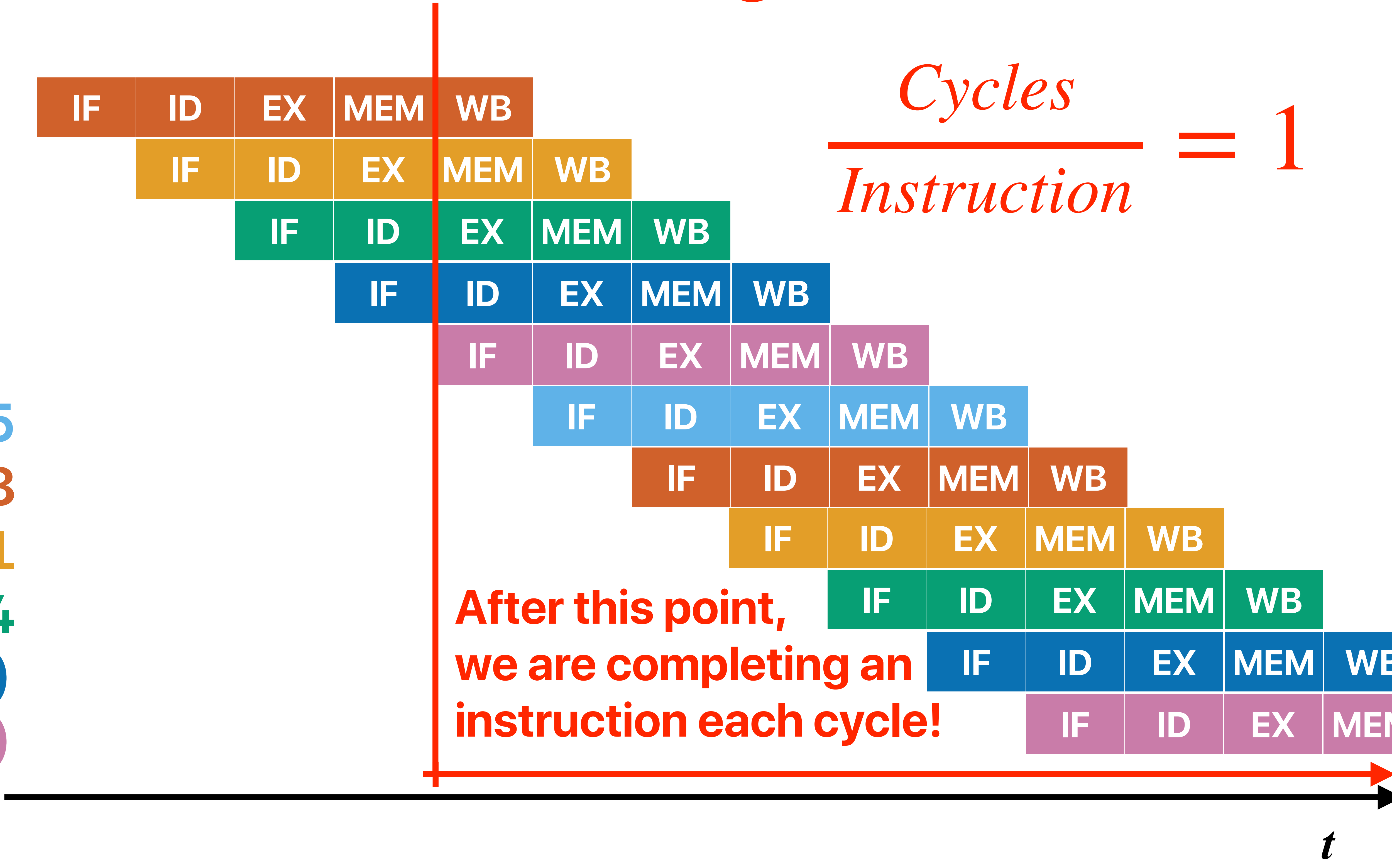
Hung-Wei Tseng

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)
```



IF
ID
EX
MEM
WB
After this point, we are completing an instruction each cycle!
t

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

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.
- Review your undergraduate architecture materials — http://cseweb.ucsd.edu/classes/su19_2/cse141-a/

Recap: addressing hazards

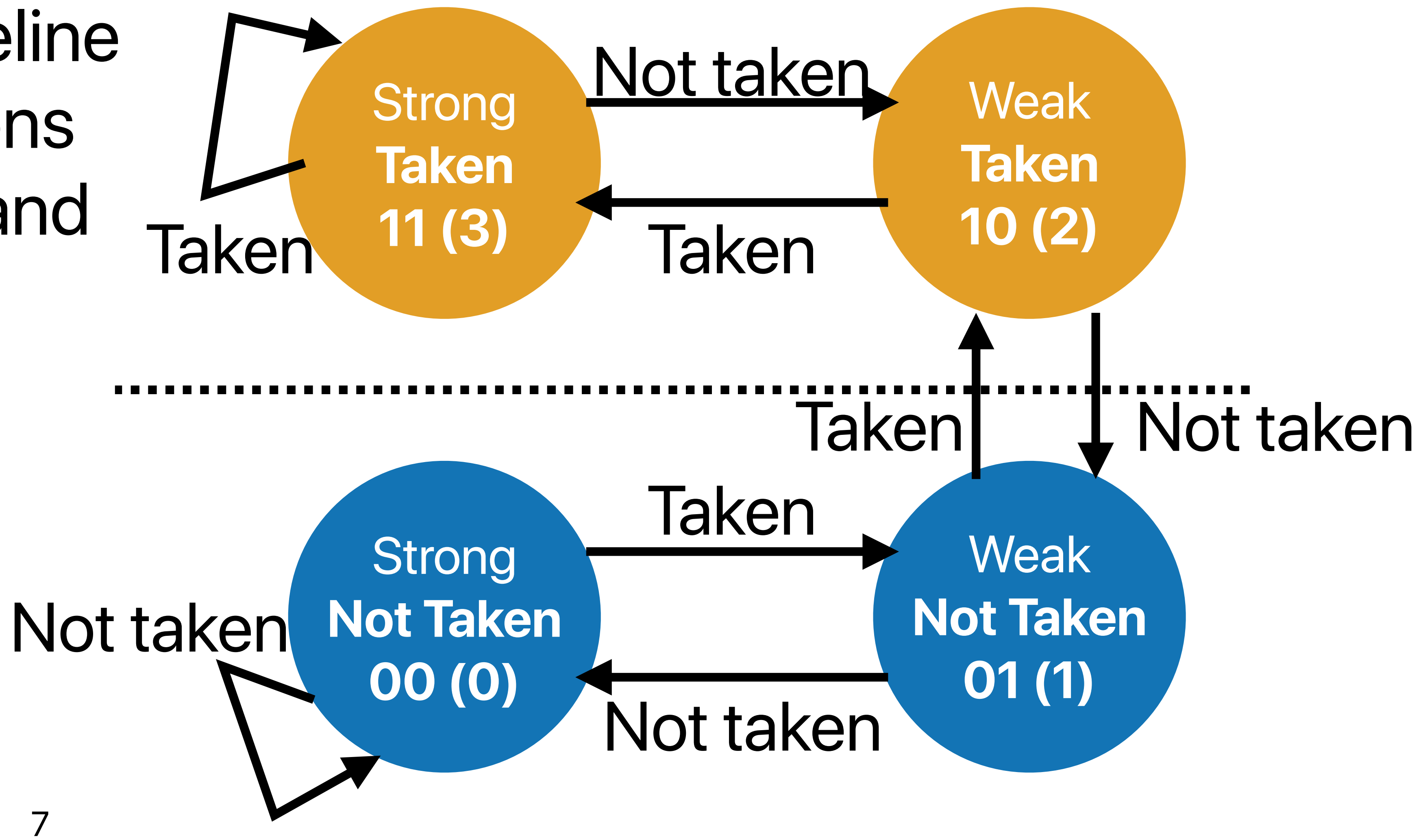
- Structural hazards
  - Stall
  - Modify hardware design
- Control hazards
  - Stall
  - Static prediction
  - Dynamic prediction

Recap: 2-bit/Bimodal local predictor

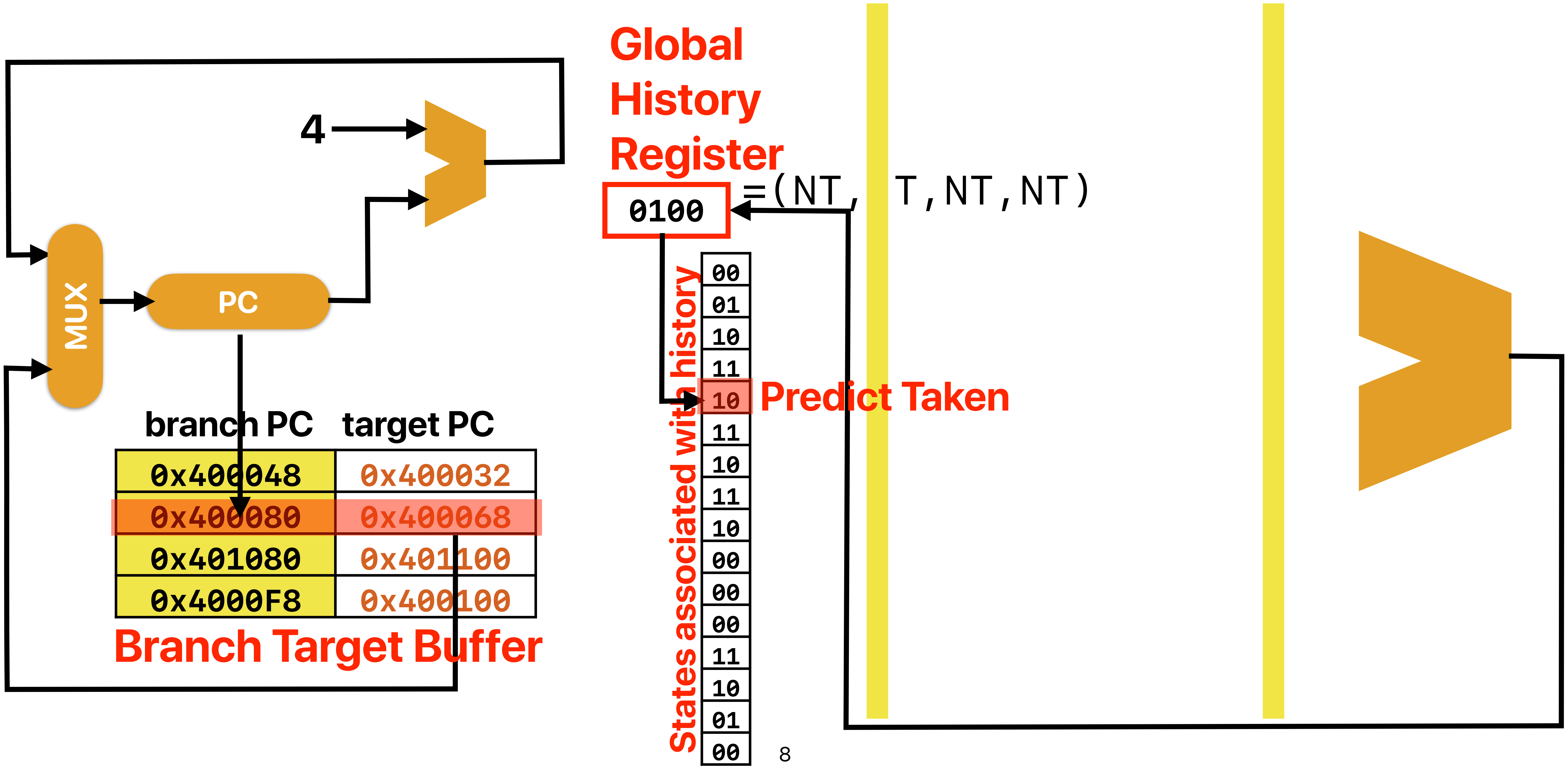
- Local predictor — every branch instruction has its own state
- 2-bit — each state is described using 2 bits
- Change the state based on **actual** outcome
- If we guess right — no penalty
- If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF and ID stages and reset the PC

Predict Taken

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



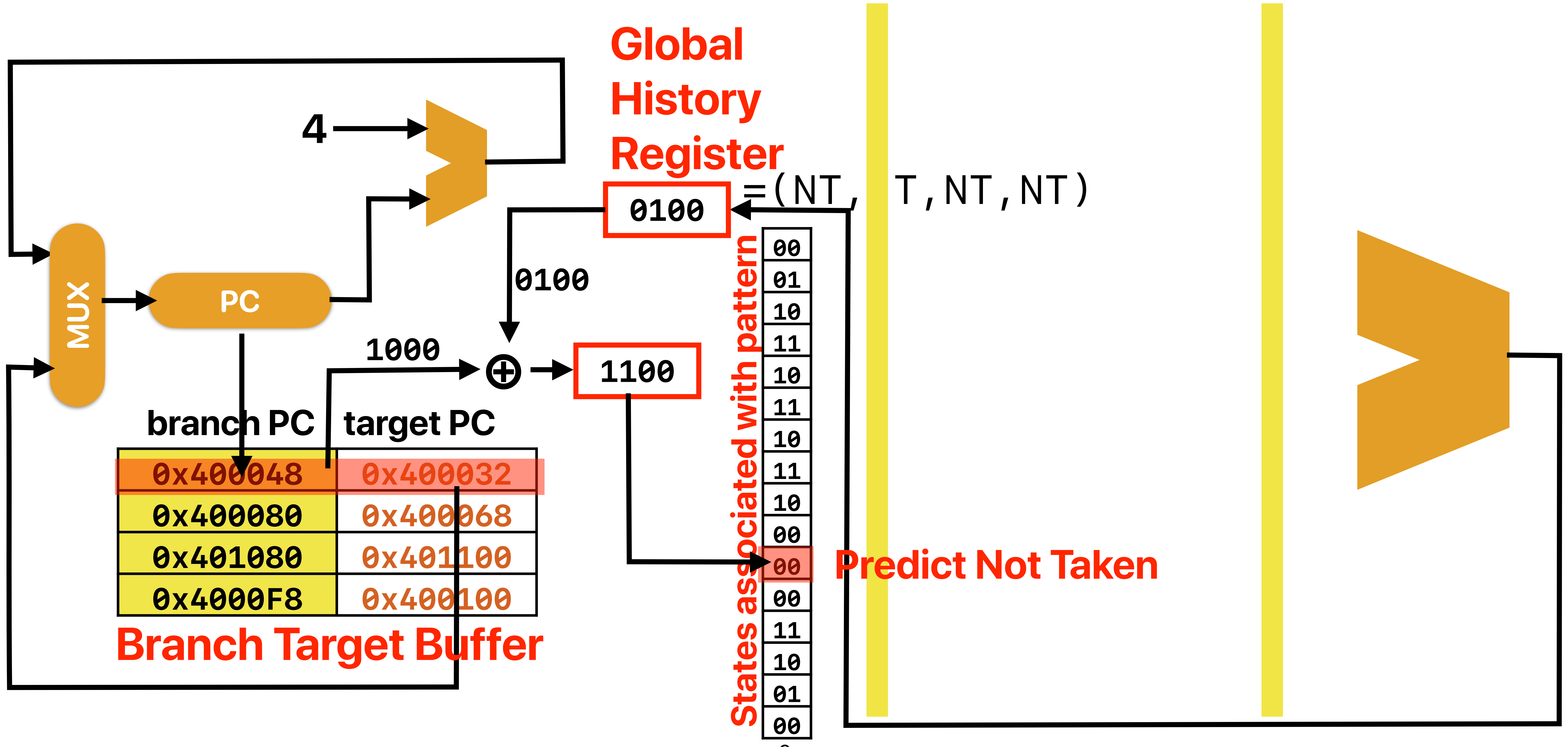
Recap: Global history (GH) predictor



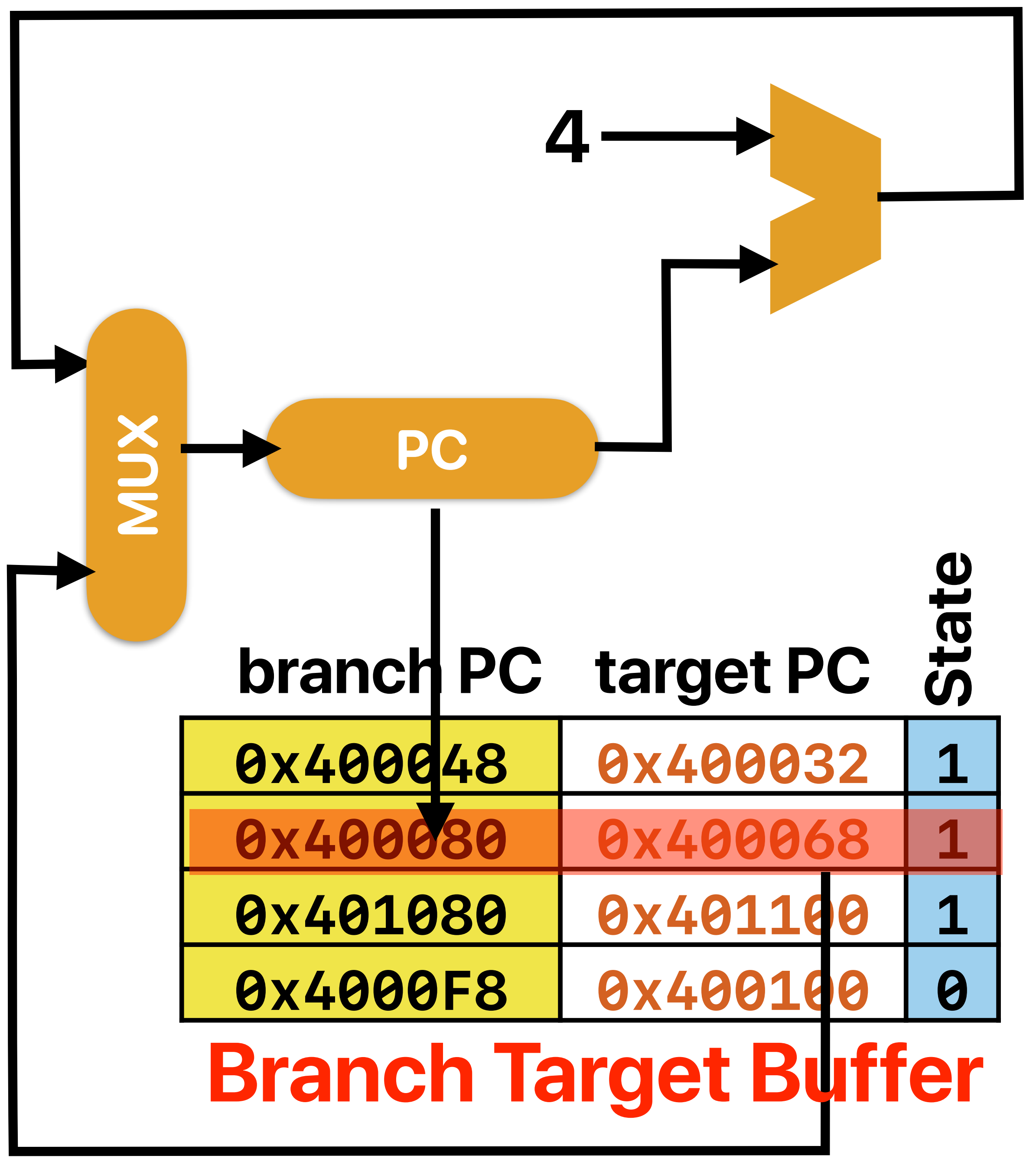
branch PC	target PC
0x400048	0x400032
0x400080	0x400068
0x401080	0x401100
0x4000F8	0x400100

Branch Target Buffer

Recap: gshare predictor



Recap: tournament Predictor



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

Branch Target Buffer

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

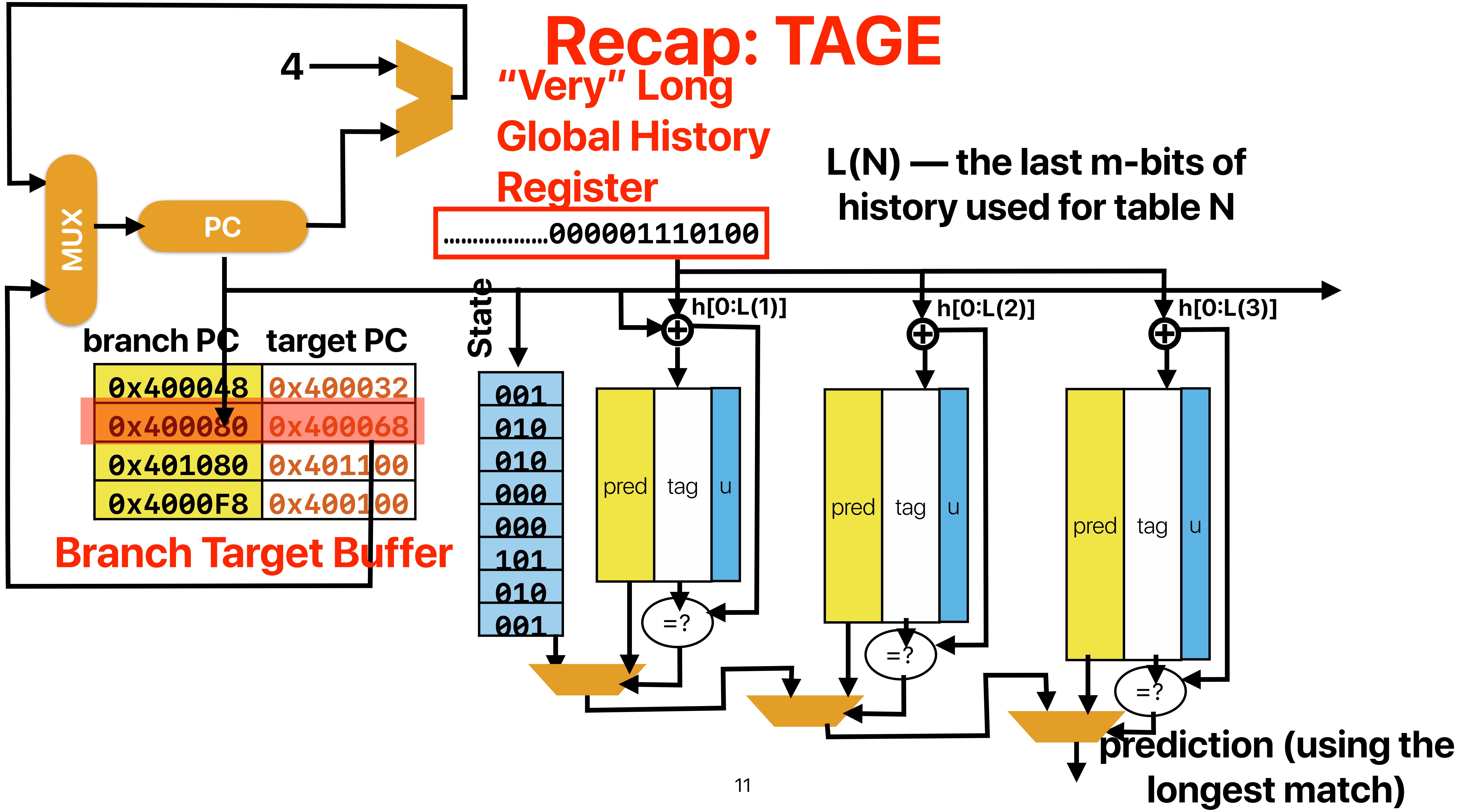
branch PC	local history
0x400048	1000
0x400080	0110
0x401080	1010
0x4000F8	0110

Predict Taken

States associated with history

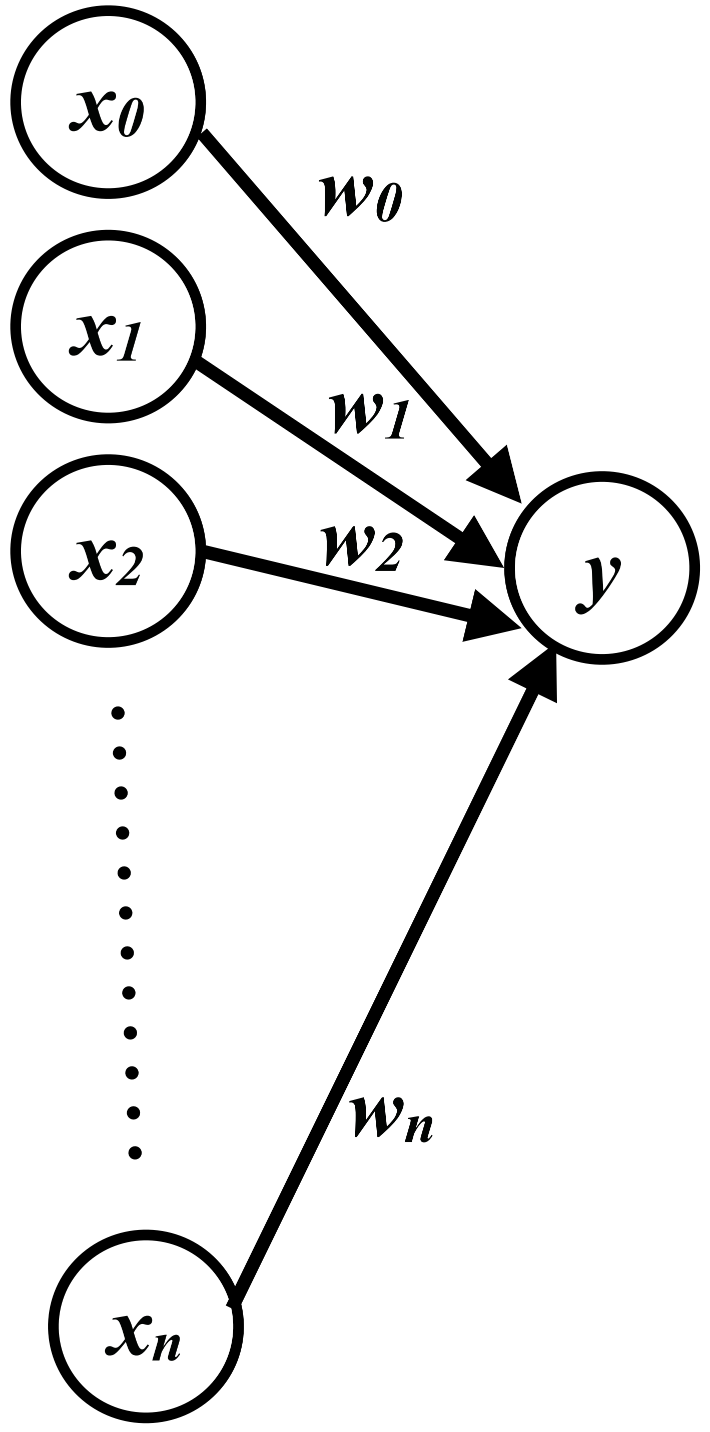
00
01
10
11
10
11
10
11
10
00
00
00
11
10
01
00

Recap: TAGE


4
MUX
PC
"Very" Long Global History Register
.................000001110100
L(N) — the last m-bits of history used for table N
h[0:L(1)]
h[0:L(2)]
h[0:L(3)]
State
branch PC
target PC
0x400048
0x400032
0x400080
0x400068
0x401080
0x401100
0x4000F8
0x400100
Branch Target Buffer
001
010
010
000
000
101
010
001
pred
tag
u
pred
tag
u
pred
tag
u
=?
=?
=?
prediction (using the longest match)

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 0
- Training finds correlations between history and outcome



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

Four implementations

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

A

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

B

```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;
}
```

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 ✓

```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;
 }
```

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

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;
}
```

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

4*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;
}
```

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

```
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
```

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

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;
     c += x & 1;
     x = x >> 1;
   }
   return c;
}
```

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

Why is C better than B?

- How many of the following statements explains the reason why C outperforms B 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

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 C outperforms B 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

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;
}
```

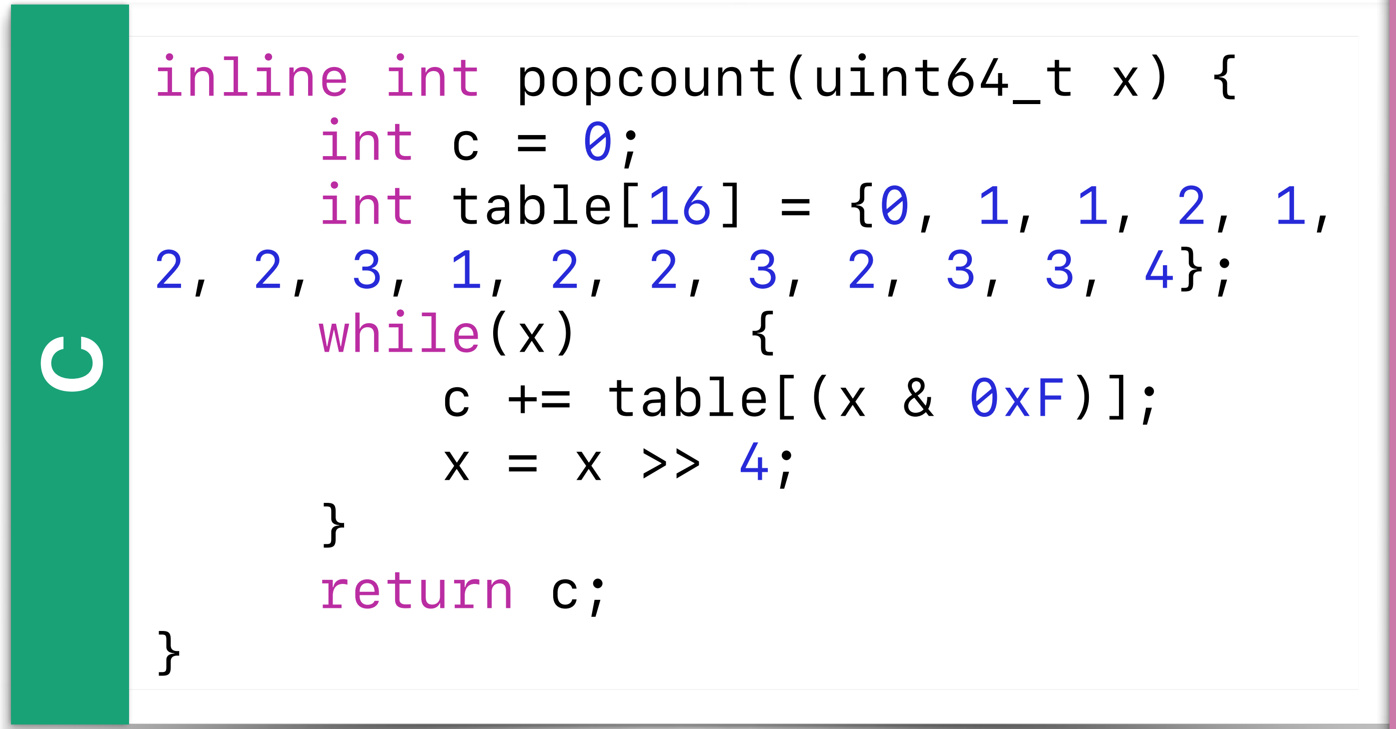
B

```
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;
}
```

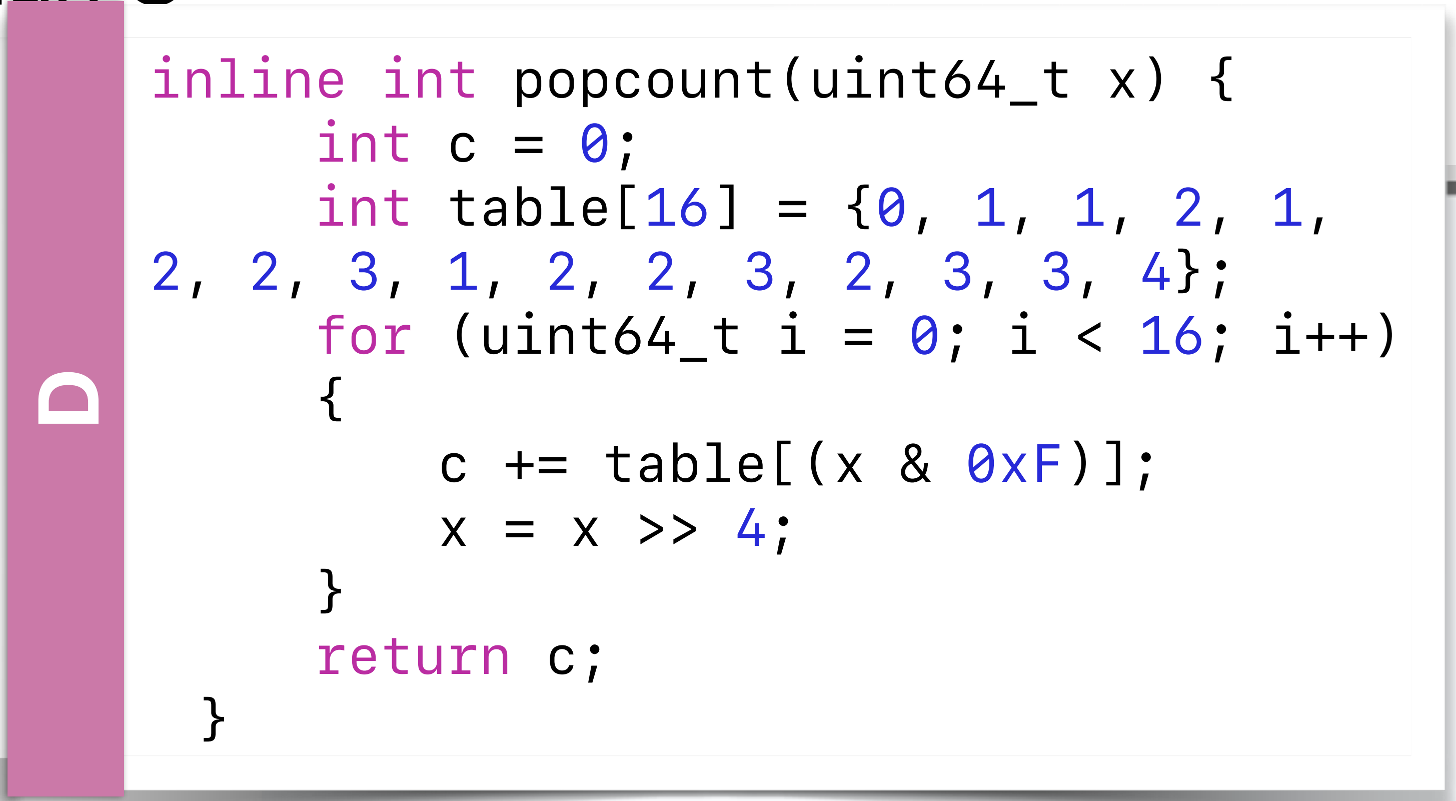
Why is D better than C?

- How many of the following statements explains the main reason why D 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


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;
  }
```

Why is D better than C?

- How many of the following statements explains the main reason why D 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

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;
 }
```

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

Outline

- Data hazards
- Tomasulo’s algorithm

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)

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 RISC-V instructions?

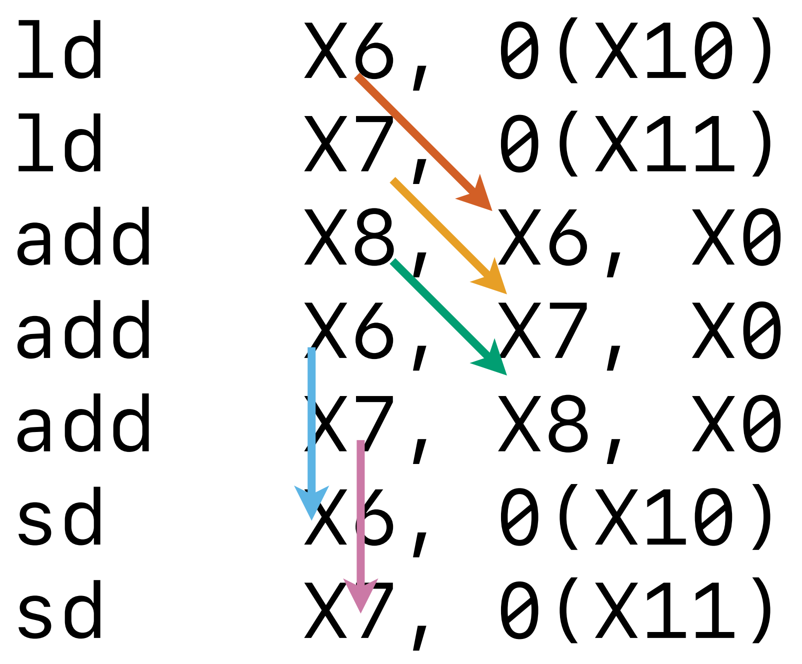
```
ld     X6, 0(X10)
ld     X7, 0(X11)
add    X8, X6, X0
add    X6, X7, X0
add    X7, X8, X0
sd     X6, 0(X10)
sd     X7, 0(X11)
```

```
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 RISC-V instructions?



```
ld      X6, 0(X10)
ld      X7, 0(X11)
add     X8, X6, X0
add     X6, X7, X0
add     X7, X8, X0
sd      X6, 0(X10)
sd      X7, 0(X11)
```

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

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.

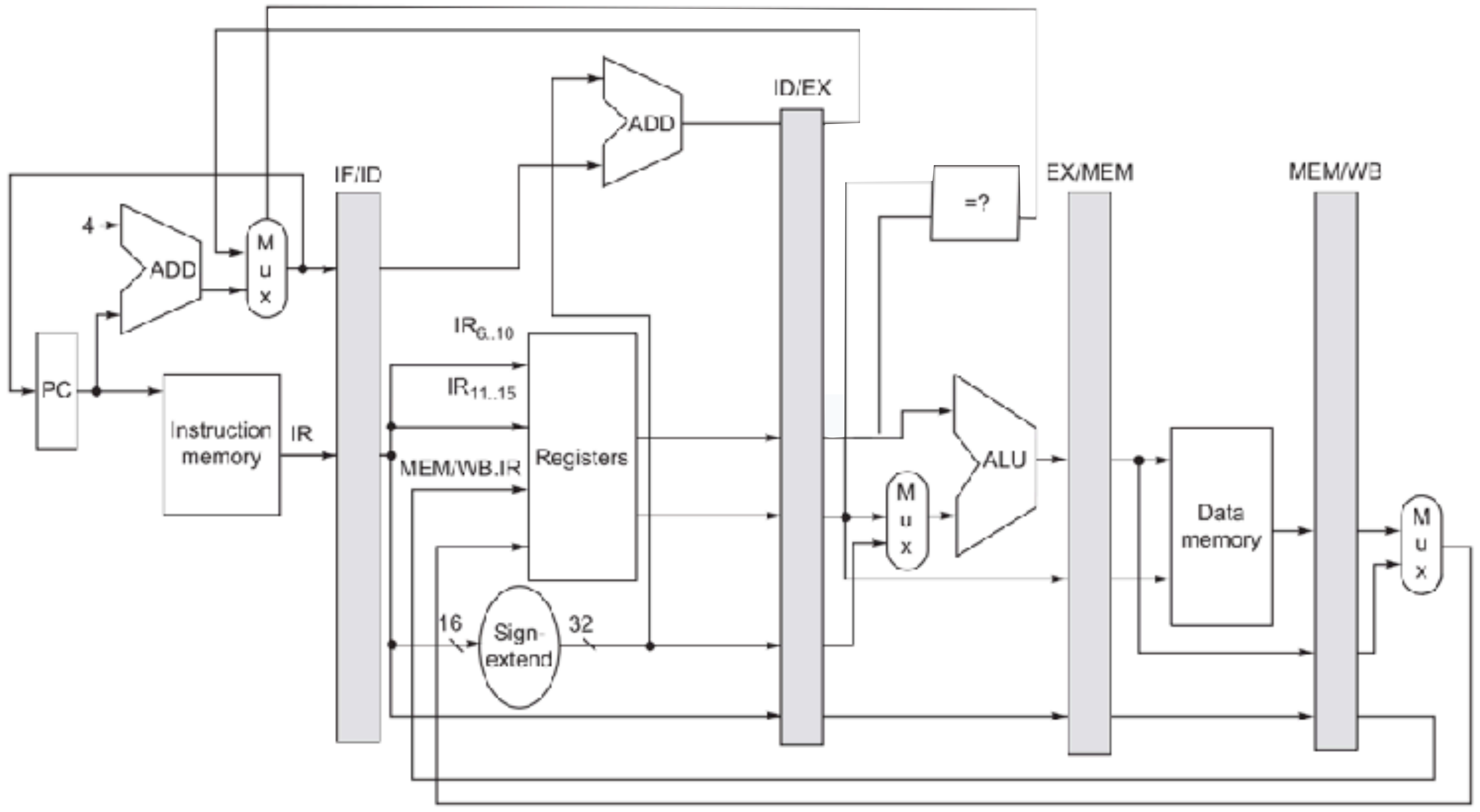
https://www.pollev.com/hungweitseng close in 1:30

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)
ld     X7, 0(X11)
add    X8, X6, X0
add    X6, X7, X0
add    X7, X8, X0
sd     X6, 0(X10)
sd     X7, 0(X11)
```

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

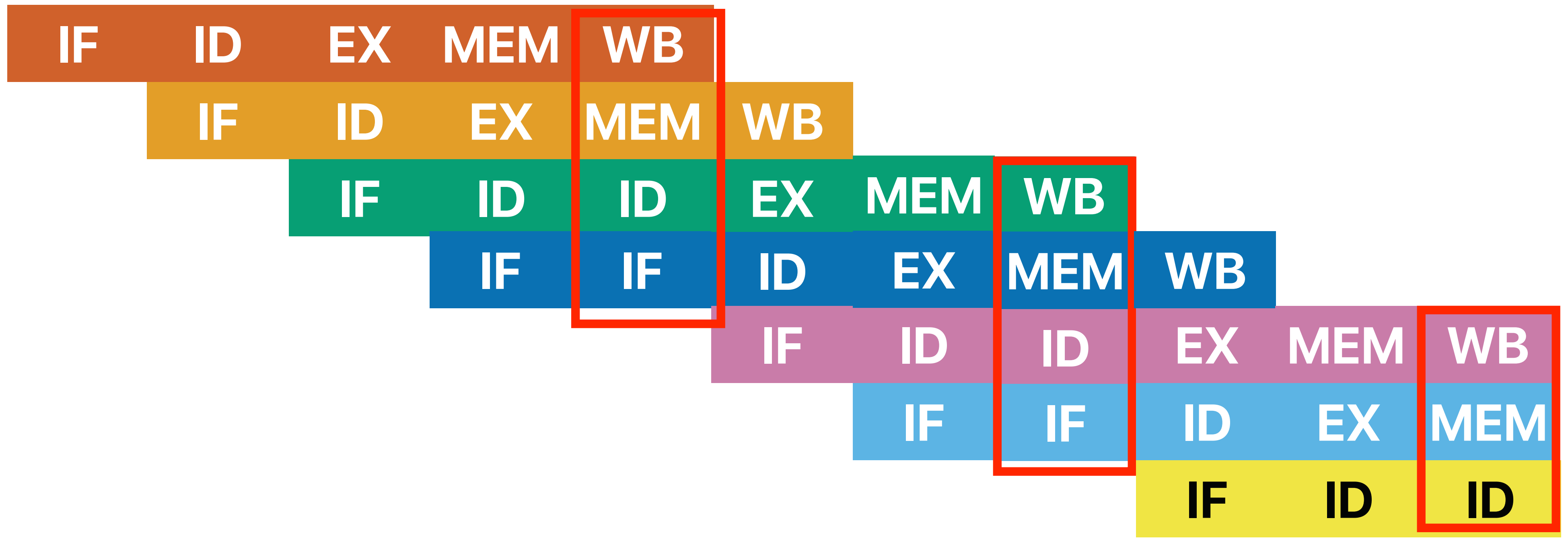


A B C D E

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?

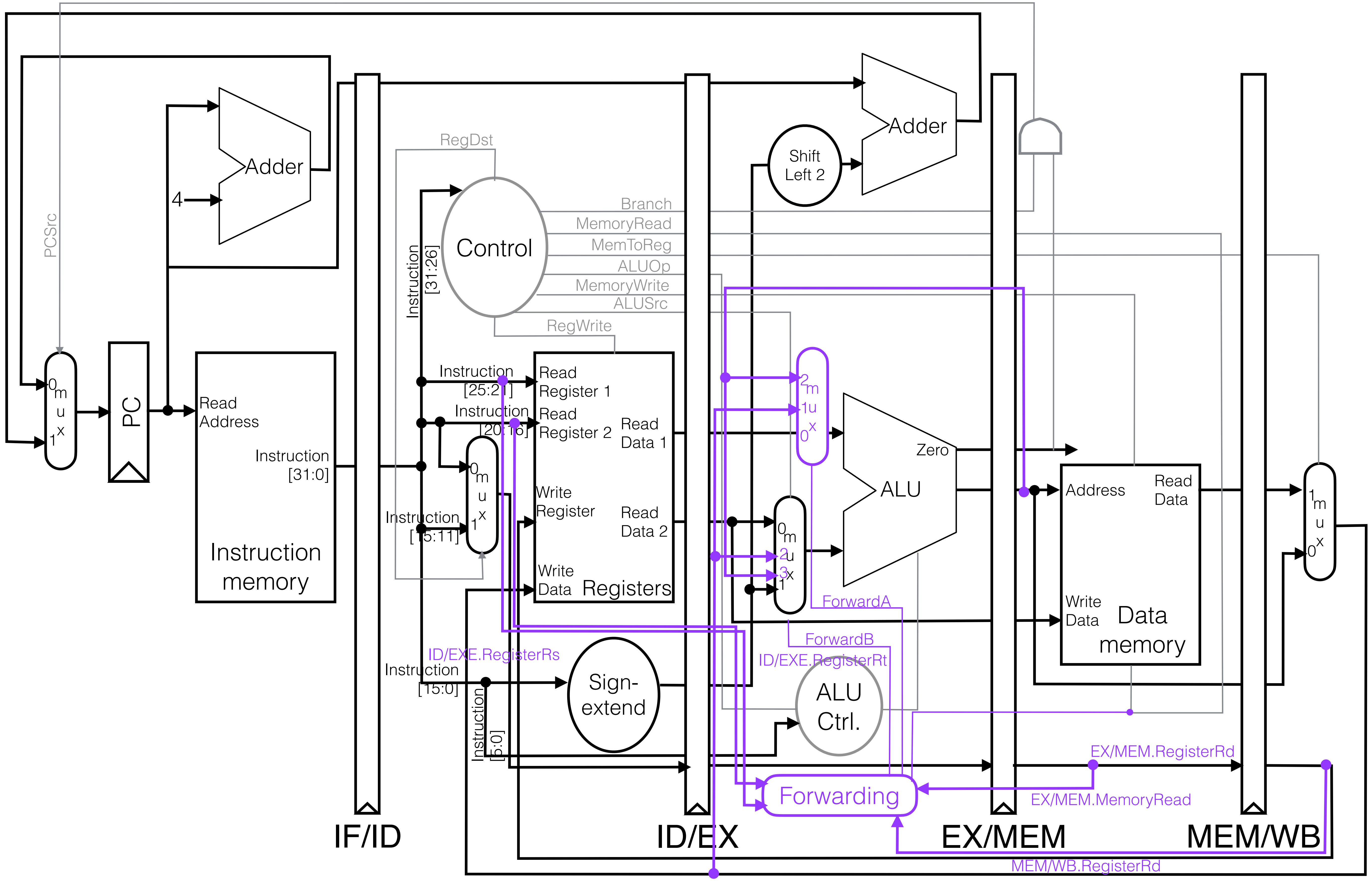
```
ld     X6, 0(X10)
ld     X7, 0(X11)
add    X8, X6, X0
add    X6, X7, X0
add    X7, X8, X0
sd     X6, 0(X10)
sd     X7, 0(X11)
```

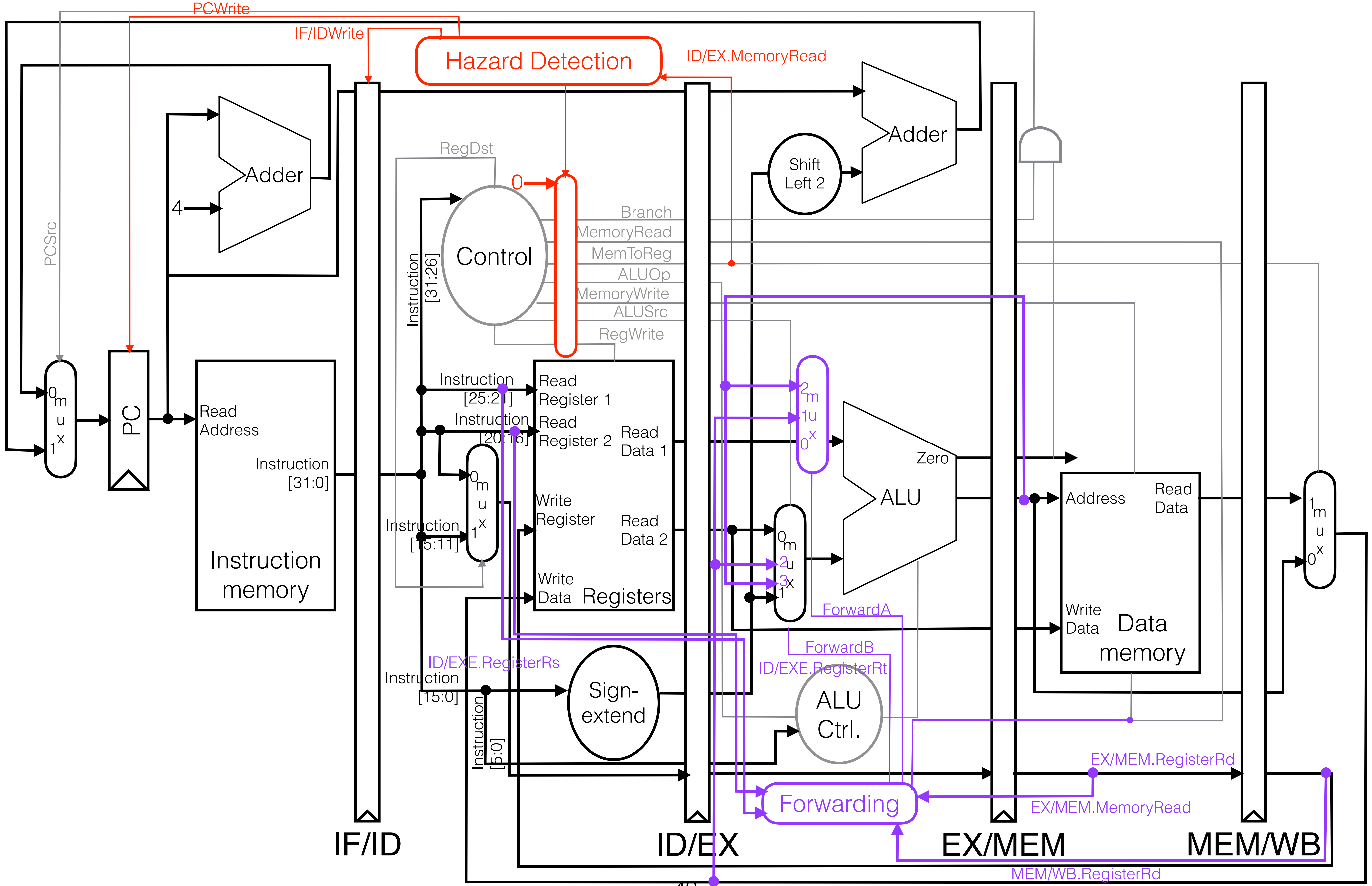


A. 1
B. 2
[✓ C. 3]
D. 4
E. 5

Solution 2: Data forwarding

- Add logics/wires to forward the desired values to the demanding instructions
- In our five stage pipeline — if the instruction entering the EXE stage consumes a result from a previous instruction that is entering MEM stage or WB stage
  - A source of the instruction entering EXE stage is the destination of an instruction entering MEM/WB stage
  - The previous instruction must be an instruction that updates register file


PCSrc
m u x
0
1
PC
Adder
4
Adder
Read Address
Instruction [31:0]
Instruction memory
IF/ID
RegDst
Control
Instruction [31:26]
Branch
MemoryRead
MemToReg
ALUOp
MemoryWrite
ALUSrc
RegWrite
Instruction [25:21]
Instruction [20:16]
Instruction [15:11]
Read Register 1
Read Register 2
Write Register
Write Data
Read Data 1
Read Data 2
Registers
Instruction [15:0]
Sign-extend
Instruction [5:0]
ID/EXE.RegisterRs
ID/EX
Shift Left 2
Adder
ALU
Zero
ALU Ctrl.
ForwardA
ForwardB
ID/EXE.RegisterRt
Forwarding
EX/MEM
Address
Read Data
Write Data
Data memory
EX/MEM.RegisterRd
EX/MEM.MemoryRead
MEM/WB
MEM/WB.RegisterRd


PCWrite
IF/IDWrite
Hazard Detection
ID/EX.MemoryRead
PCSrc
Adder
4
Adder
Shift Left 2
RegDst
0
Control
Instruction [31:26]
Branch
MemoryRead
MemToReg
ALUOp
MemoryWrite
ALUSrc
RegWrite
m u x
PC
Read Address
Instruction [31:0]
Instruction memory
Instruction [25:21]
Instruction [20:16]
Instruction [15:11]
Read Register 1
Read Register 2
Write Register
Write Data
Read Data 1
Read Data 2
Registers
Zero
ALU
Address
Read Data
Write Data
Data memory
ForwardA
ForwardB
ID/EXE.RegisterRs
ID/EXE.RegisterRt
Instruction [15:0]
Sign-extend
Instruction [5:0]
ALU Ctrl.
EX/MEM.RegisterRd
EX/MEM.MemoryRead
Forwarding
MEM/WB.RegisterRd
IF/ID
ID/EX
EX/MEM
MEM/WB

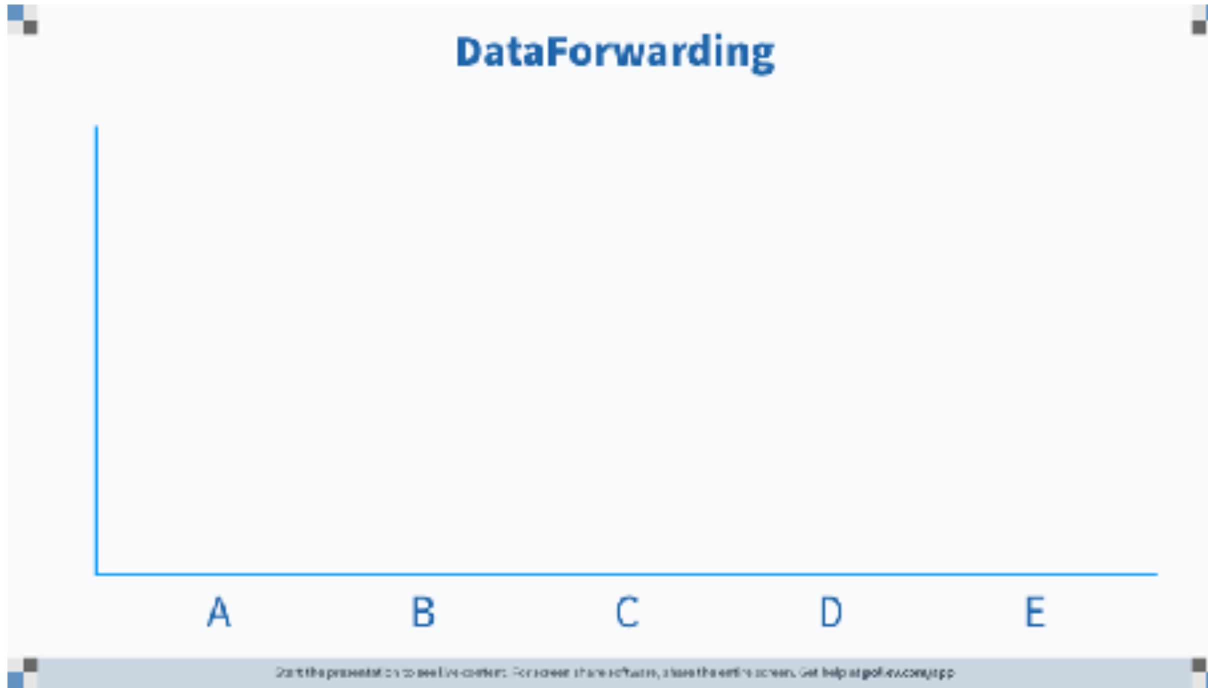
https://www.pollev.com/hungweitseng close in 1:30

How many of data hazards w/ Data Forwarding?

- 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 with "full" data forwarding?

```
ld     X6, 0(X10)
ld     X7, 0(X11)
add    X8, X6, X0
add    X6, X7, X0
add    X7, X8, X0
sd     X6, 0(X10)
sd     X7, 0(X11)
```

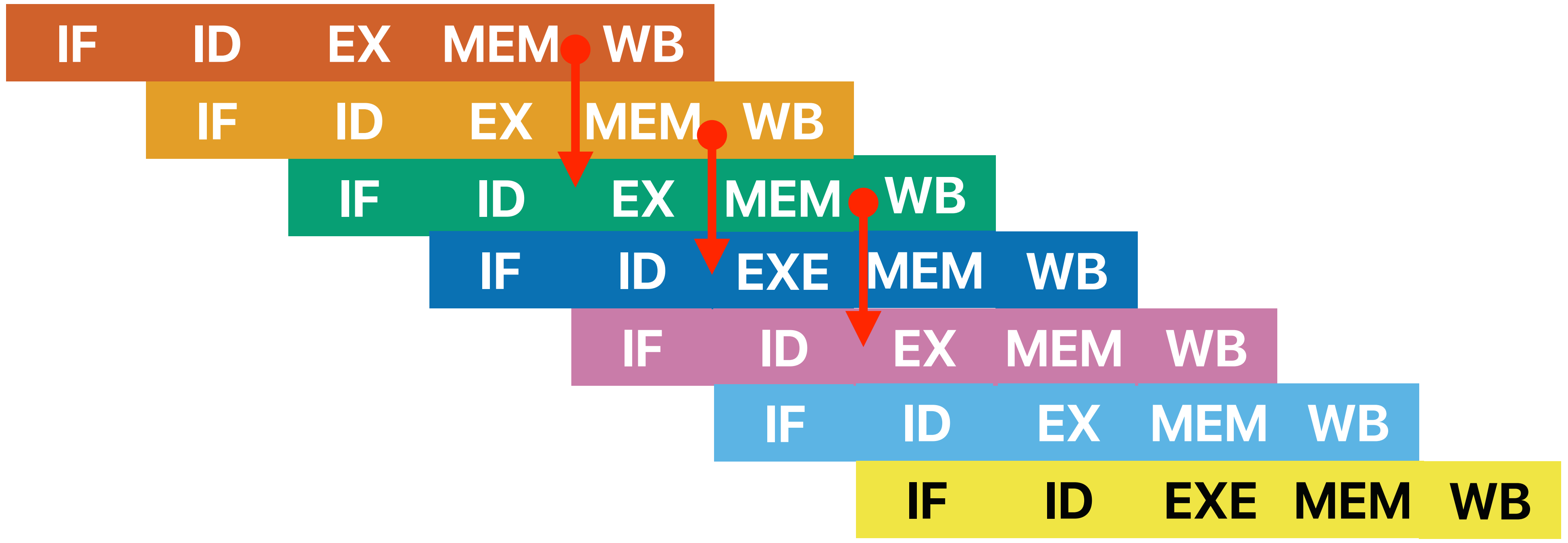
A. 0
B. 1
C. 2
D. 3
E. 4



How many of data hazards w/ Data Forwarding?

- 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 with “full” data forwarding?

```
ld     X6, 0(X10)
ld     X7, 0(X11)
add    X8, X6, X0
add    X6, X7, X0
add    X7, X8, X0
sd     X6, 0(X10)
sd     X7, 0(X11)
```



[✓ A. 0]
B. 1
C. 2
D. 3
E. 4

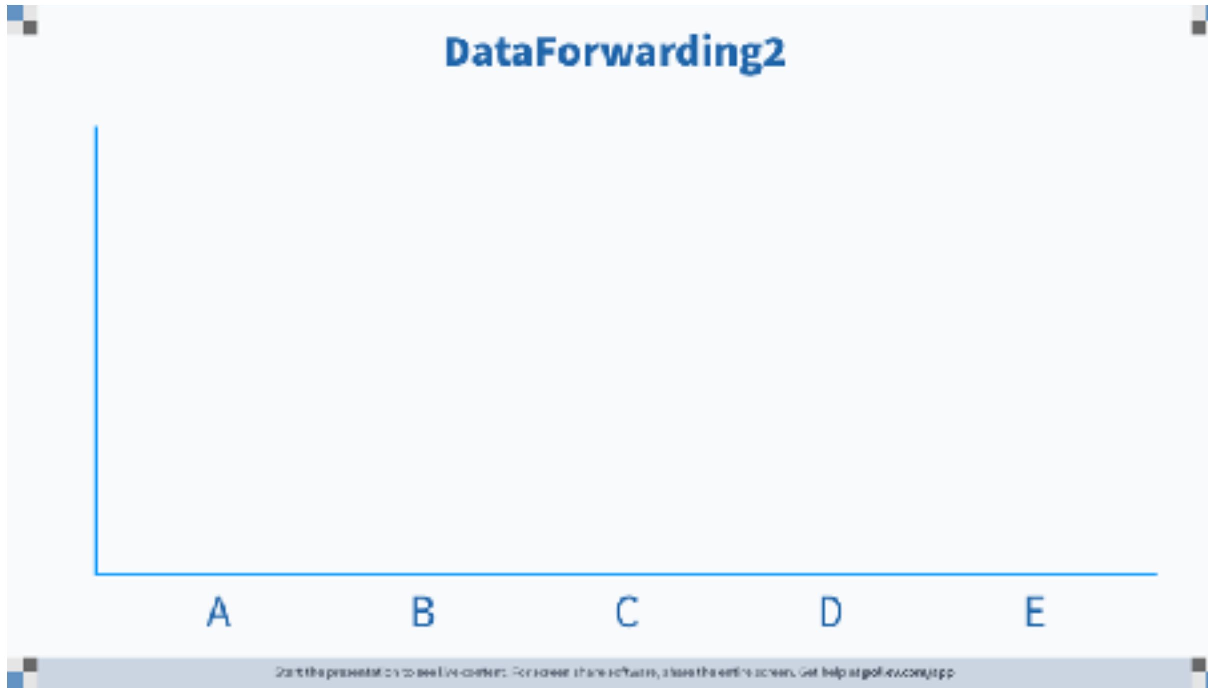
https://www.pollev.com/hungweitseng close in 1:30

How many of data hazards w/ Data Forwarding?

- 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 with "full" data forwarding?

```
ld     X6, 0(X10)
ld     X7, 0(X11)
xor    X6, X6, X7
xor    X7, X7, X6
xor    X6, X6, X7
sd     X6, 0(X10)
sd     X7, 0(X11)
```

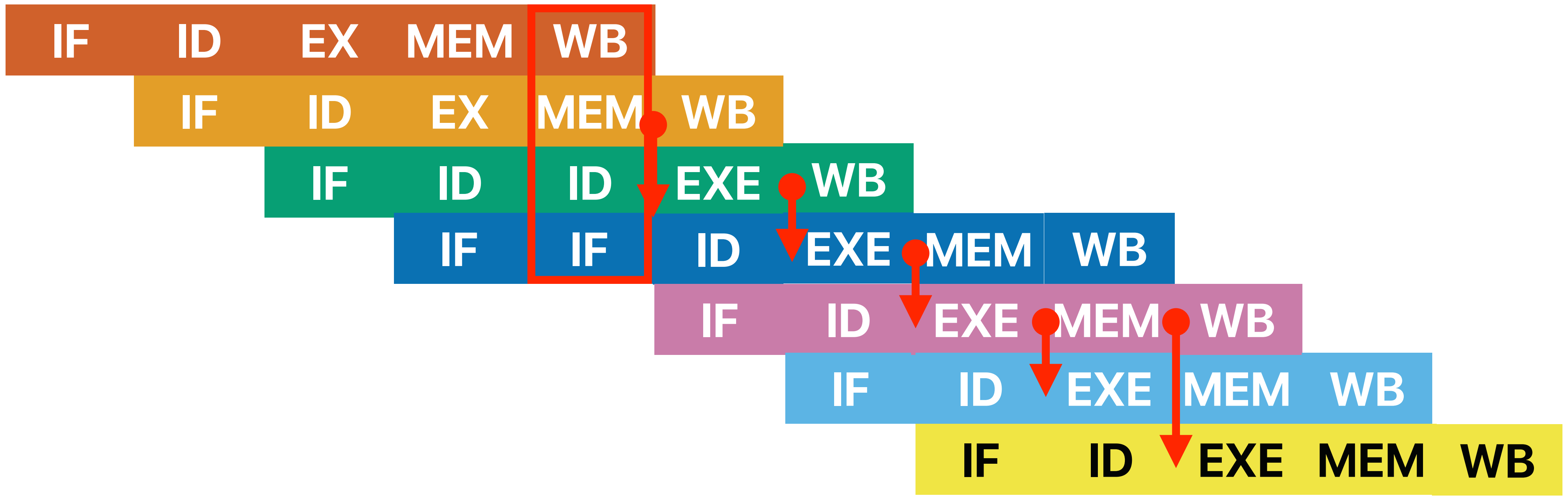
A. 0
B. 1
C. 2
D. 3
E. 4



How many of data hazards w/ Data Forwarding?

- 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 with "full" data forwarding?

```
ld     X6, 0(X10)
ld     X7, 0(X11)
xor    X6, X6, X7
xor    X7, X7, X6
xor    X6, X6, X7
sd     X6, 0(X10)
sd     X7, 0(X11)
```

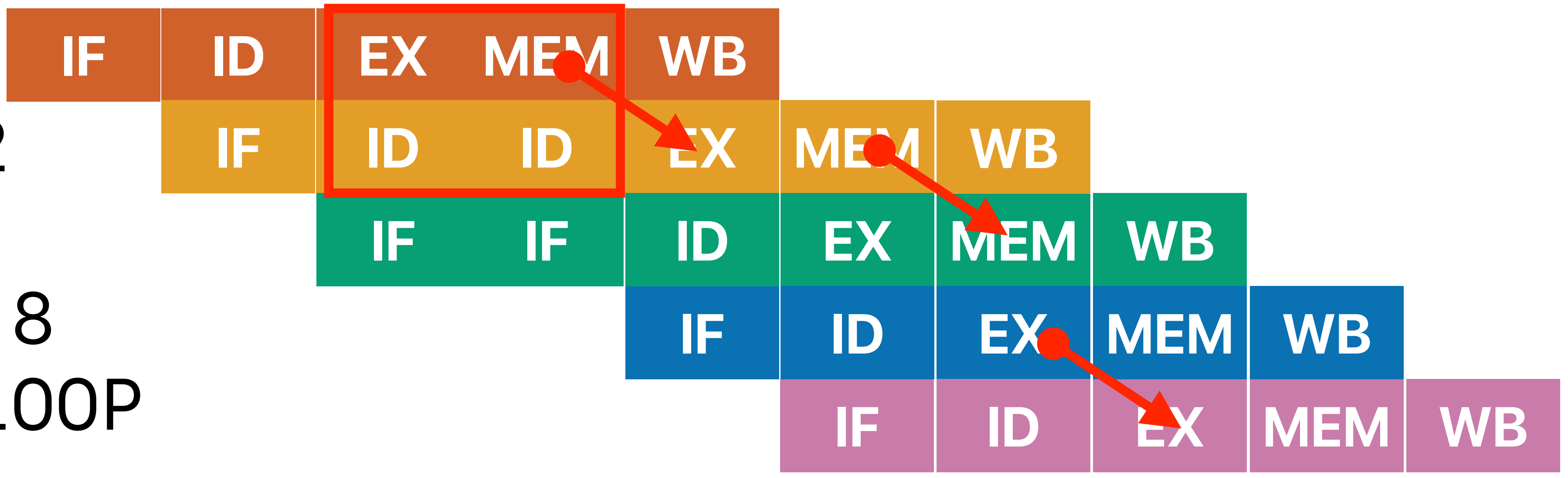


A. 0
[✓ B. 1]
C. 2
D. 3
E. 4

How many of data hazards w/ Data Forwarding?

- 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 with “full” data forwarding?

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



A. 0
[✓ B. 1]
C. 2
D. 3
E. 4

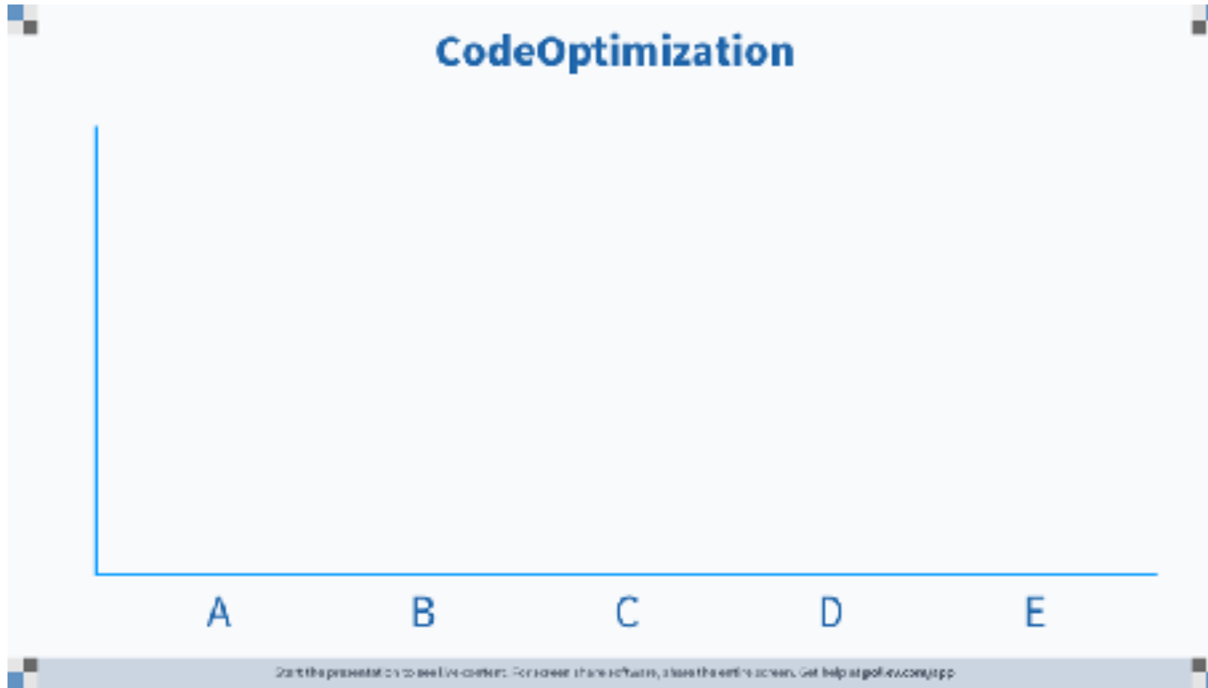
https://www.pollev.com/hungweitseng close in 1:30

The effect of code optimization

- By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

```
① ld   X6,0(X10)
② add  X7,X6, X12
③ sd   X7,0(X10)
④ addi X10,X10, 8
⑤ bne  X10,X5, LOOP
```

A. (1) & (2)
B. (2) & (3)
C. (3) & (4)
D. (4) & (5)
E. None of the pairs can be reordered



The effect of code optimization

- By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

```
① ld   X6,0(X10)
② add  X7,X6, X12
③ sd   X7,0(X10)
④ addi X10,X10, 8
⑤ bne  X10,X5, LOOP
```

A. (1) & (2)
B. (2) & (3)
C. (3) & (4)
D. (4) & (5)
[✓ E. None of the pairs can be reordered]

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

If we can predict the future …

- Consider the following dynamic instructions:

```
① ld   X6,0(X10)
② add  X7,X6, X12
③ sd   X7,0(X10)
④ addi X10,X10, 8
⑤ bne  X10,X5, LOOP
⑥ ld   X6,0(X10)
⑦ add  X7,X6, X12
⑧ sd   X7,0(X10)
⑨ addi X10,X10, 8
⑩ bne  X10,X5, LOOP
```

Which of the following pair can we reorder without affecting the correctness if the **branch prediction is perfect**?

A. (2) and (4)
B. (3) and (5)
C. (5) and (6)
D. (6) and (9)
E. (9) and (10)

If we can predict the future …

- Consider the following dynamic instructions:

```
① ld   X6,0(X10)
② add  X7,X6, X12
③ sd   X7,0(X10)
④ addi X10,X10, 8
⑤ bne  X10,X5, LOOP
⑥ ld   X6,0(X10)
⑦ add  X7,X6, X12
⑧ sd   X7,0(X10)
⑨ addi X10,X10, 8
⑩ bne  X10,X5, LOOP
```

Which of the following pair can we reorder without affecting the correctness if the **branch prediction is perfect**?

A. (2) and (4)
B. (3) and (5)
[✓ C. (5) and (6)]
D. (6) and (9)
E. (9) and (10)

Can we use "branch prediction" to predict the future and reorder instructions across the branch?

Dynamic instruction scheduling/ Out-of-order (OoO) execution

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
- 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.

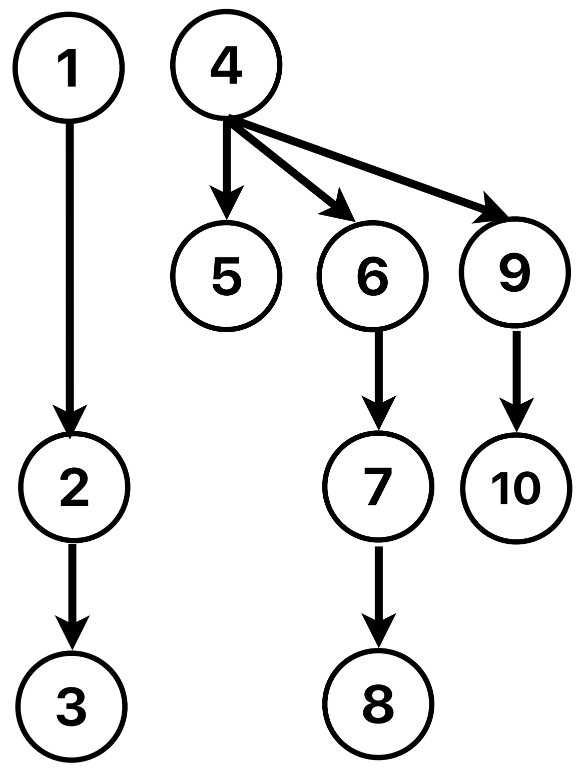
What do you need to execution an instruction?

- Whenever the instruction is decoded — put decoded instruction somewhere
- Whenever the inputs are ready — **all data dependencies are resolved**
- Whenever the target functional unit is available

Scheduling instructions: based on data dependencies

- Draw the data dependency graph, put an arrow if an instruction depends on the other.

```
①  ld    X6,0(X10)
②  add   X7,X6,X12
③  sd    X7,0(X10)
④  addi  X10,X10,8
⑤  bne   X10,X5,LOOP
⑥  ld    X6,0(X10)
⑦  add   X7,X6,X12
⑧  sd    X7,0(X10)
⑨  addi  X10,X10,8
⑩  bne   X10,X5,LOOP
```

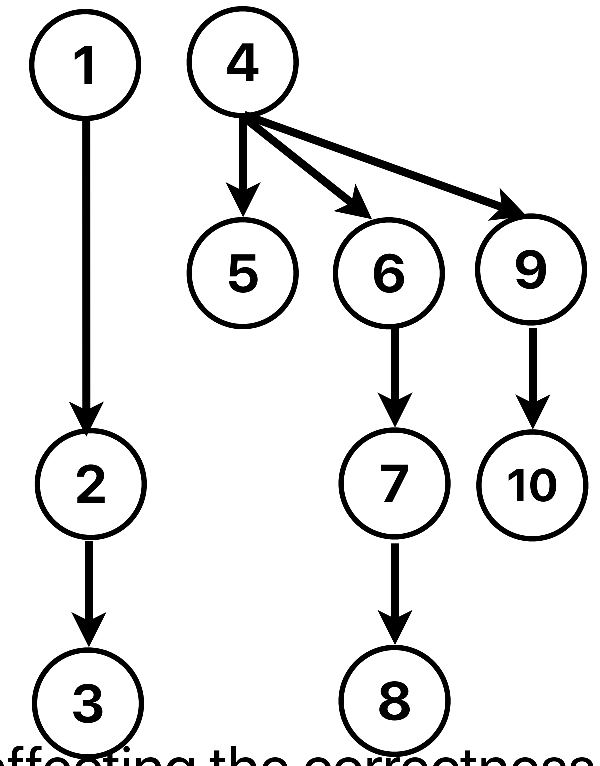

$$\begin{tikzcd}
1 \arrow[d] & 4 \arrow[d] \arrow[dr] \arrow[drr] & & \\
 & 5 & 6 \arrow[d] & 9 \arrow[d] \\
2 \arrow[d] & & 7 \arrow[d] & 10 \\
3 & & 8 &
\end{tikzcd}$$

- **In theory**, instructions without dependencies can be executed in parallel or out-of-order
- Instructions with dependencies can never be reordered

If we can predict the future …

- Consider the following dynamic instructions:

```
① ld   X6,0(X10)
② add  X7,X6, X12
③ sd   X7,0(X10)
④ addi X10,X10, 8
⑤ bne  X10,X5, LOOP
⑥ ld   X6,0(X10)
⑦ add  X7,X6, X12
⑧ sd   X7,0(X10)
⑨ addi X10,X10, 8
⑩ bne  X10,X5, LOOP
```



Which of the following pair can we reorder without affecting the correctness if the **branch prediction is perfect**?

A. (2) and (4)
B. (3) and (5)
[✓ C. (5) and (6)]
D. (6) and (9)
E. (9) and (10)

We still can only reorder (5) and (6) even though (2) & (4) are not depending on each other!

Announcements

- Assignment #3 due next **Wednesday**
- Reading Quiz due 11/22
- Project is released
  - Please check website to the link of GitHub repo
  - You may discuss, but each needs an individual/distinguishable version of code
  - You need to write a brief report
  - Grading rubrics
    - 20% — report
    - 20% — if you code can compile and run
    - 60% — performance based. The sample prefetcher is the baseline. We calculate your score at this part using min(Speedup-1, 1). If you can speedup by 2, you score full credits in this part
  - Due 11/29 — **no extension**

Computer
Science &
Engineering

つづく