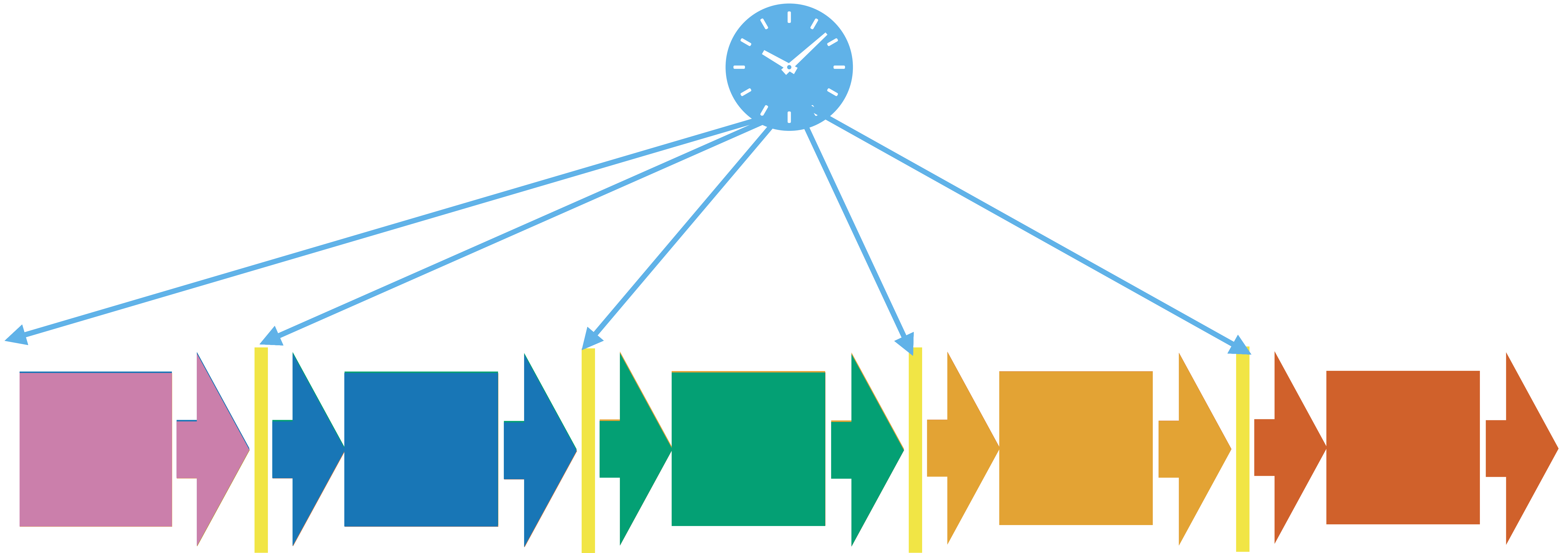
Data Hazards & Dynamic Instruction Scheduling (I)

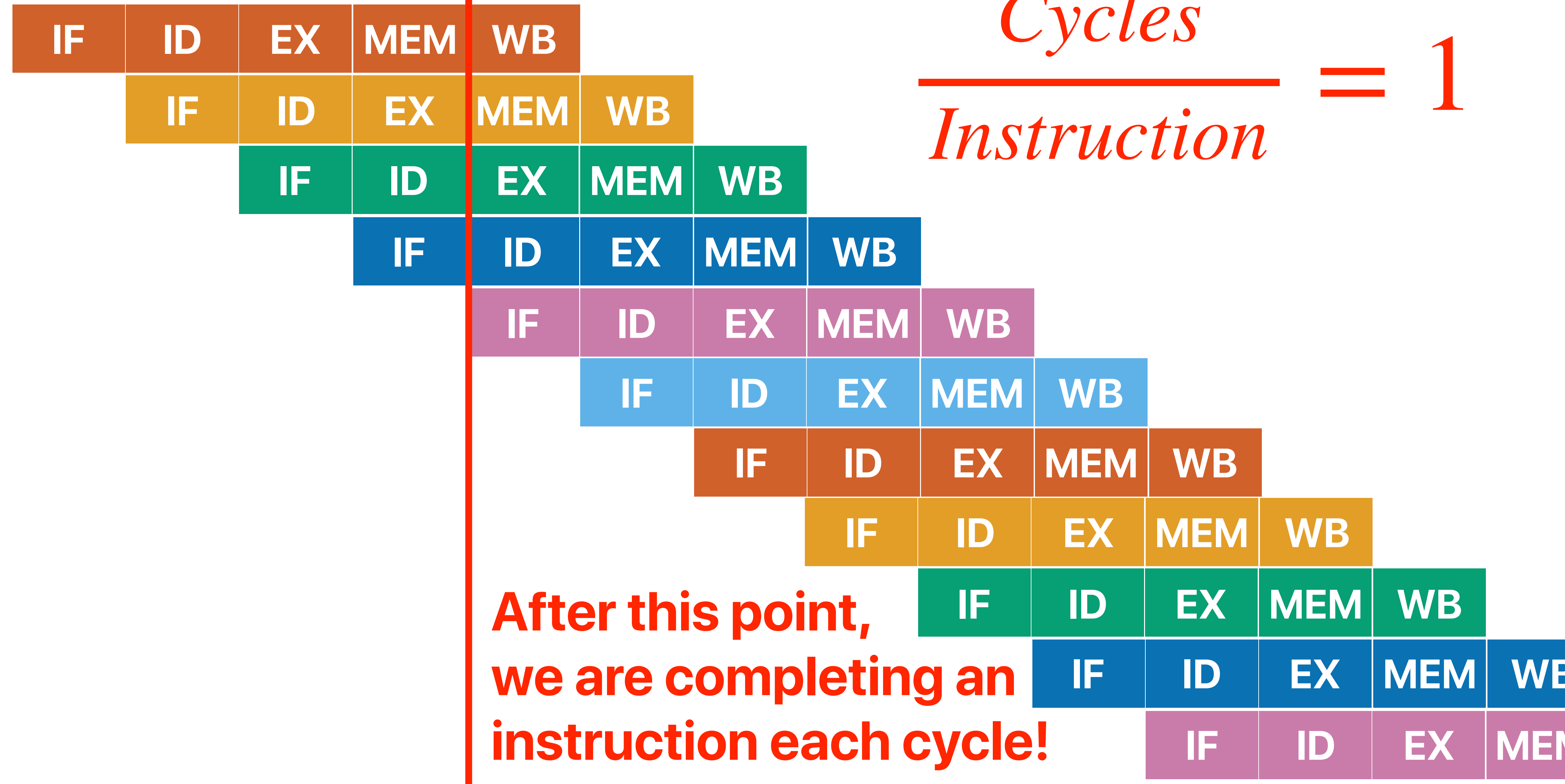
Hung-Wei Tseng

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

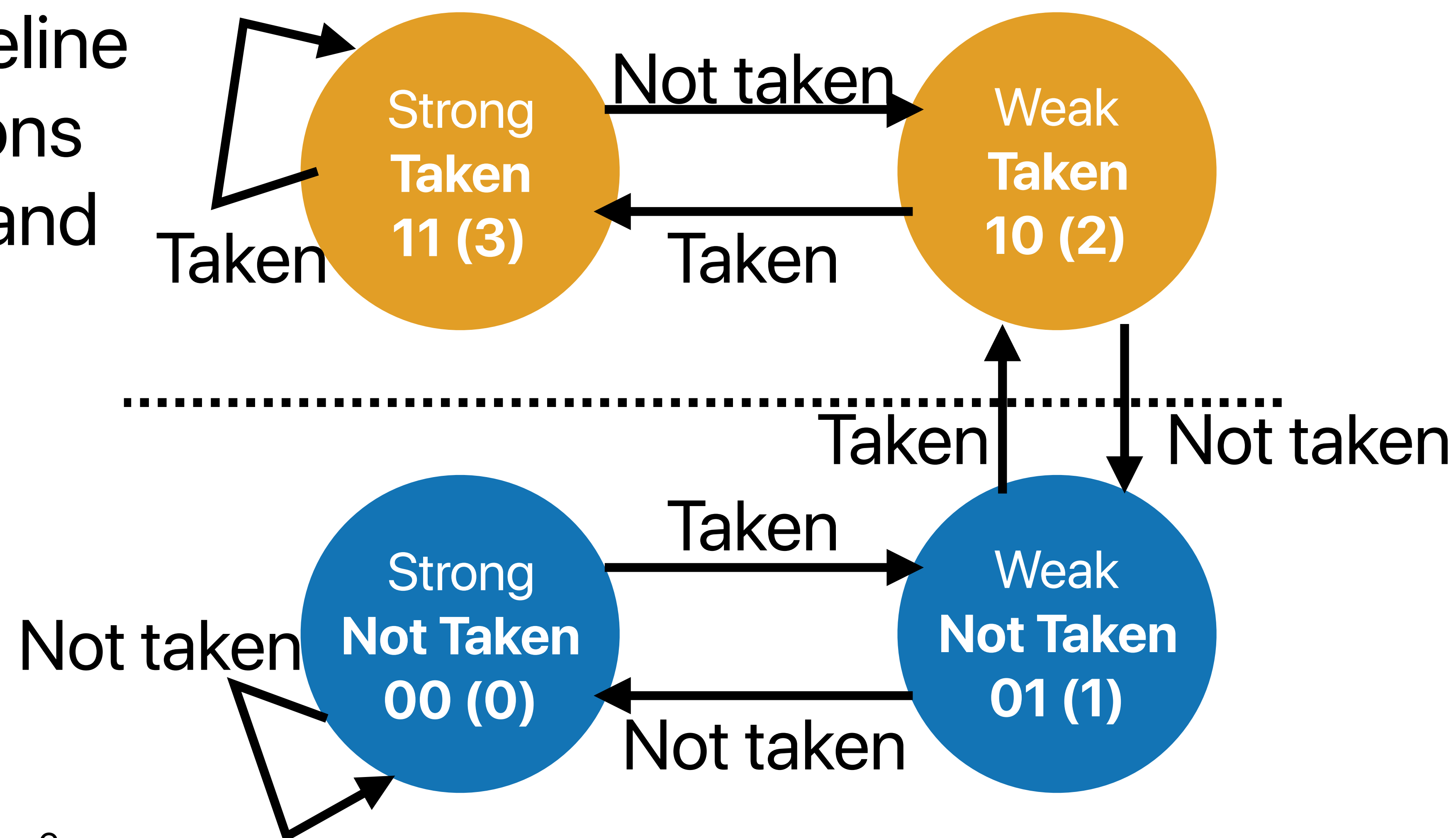
- Structural hazards — resource conflicts cannot support simultaneous execution of instructions in the pipeline
- Control hazards — the PC can be changed by an instruction in the pipeline
- Data hazards — an instruction depending on a the result that's not yet generated or propagated when the instruction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

Recap: addressing hazards

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

Recap: 2-bit/Bimodal local predictor

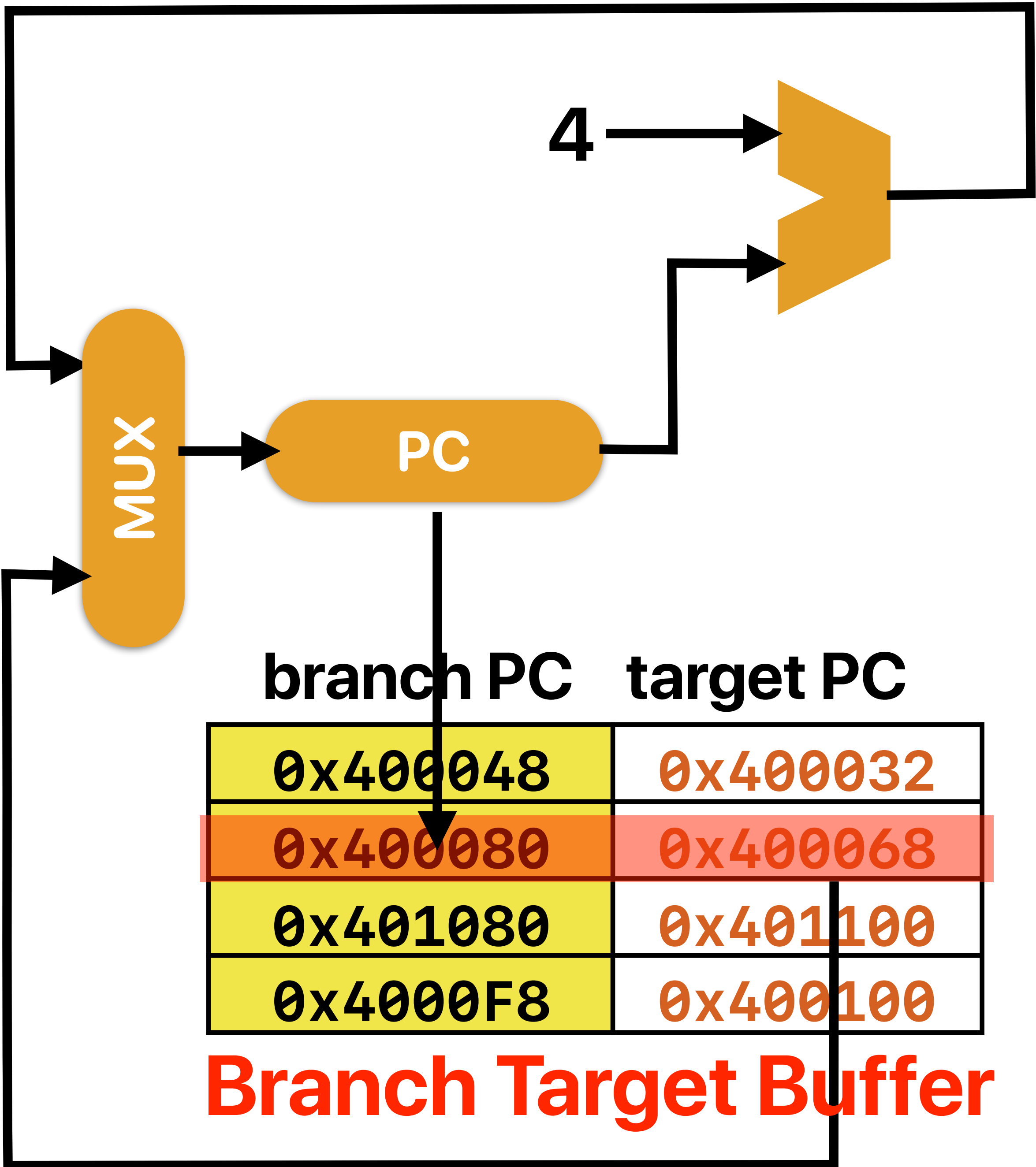
- Local predictor — every branch instruction has its own state
- 2-bit — each state is described using 2 bits
- Change the state based on **actual** outcome
- If we guess right — no penalty
- If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF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



Predict Taken (row 0x400080)

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

Recap: Global history (GH) predictor



Global History Register

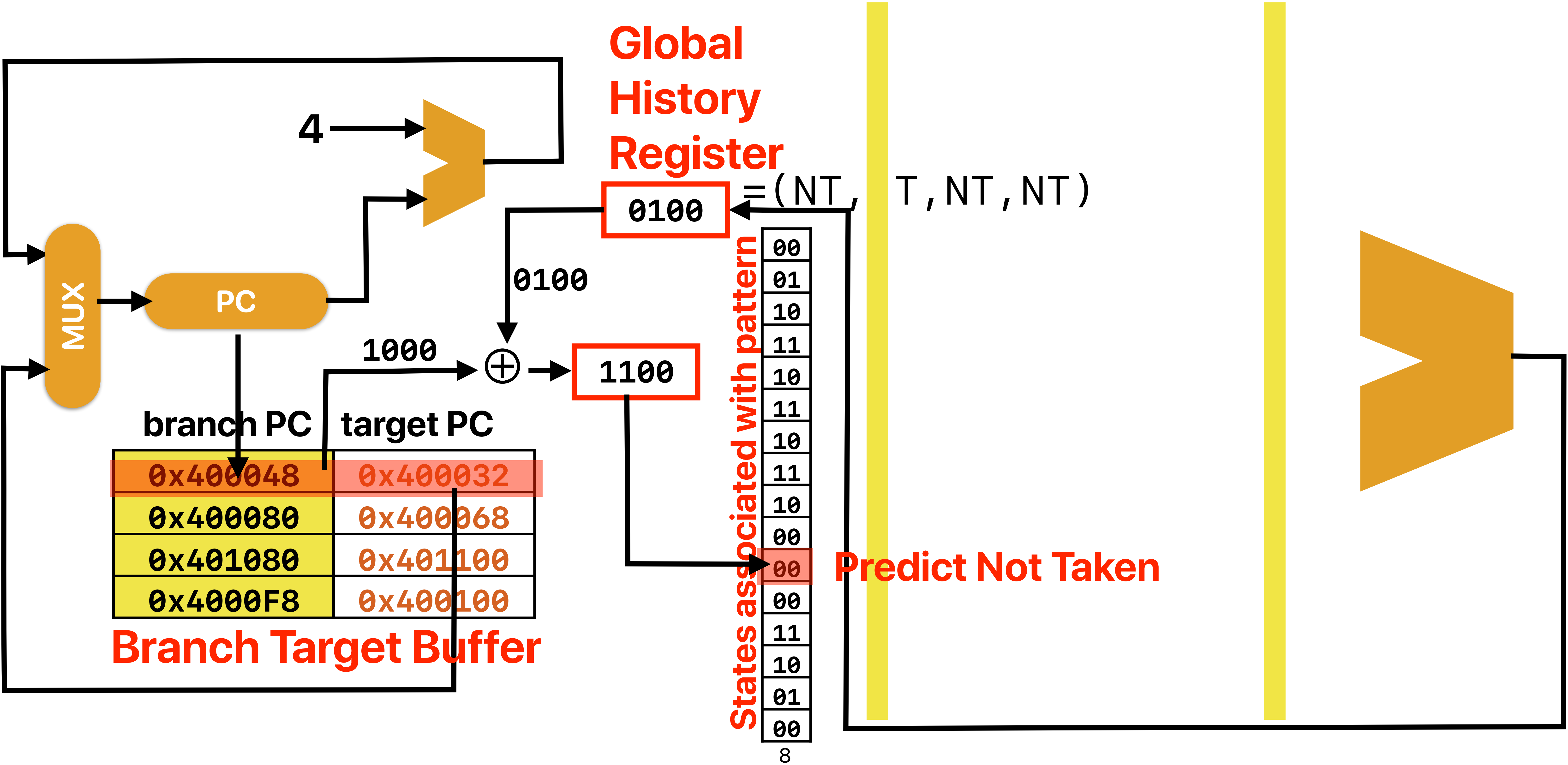
0100 =(NT, T,NT,NT)

States associated with history

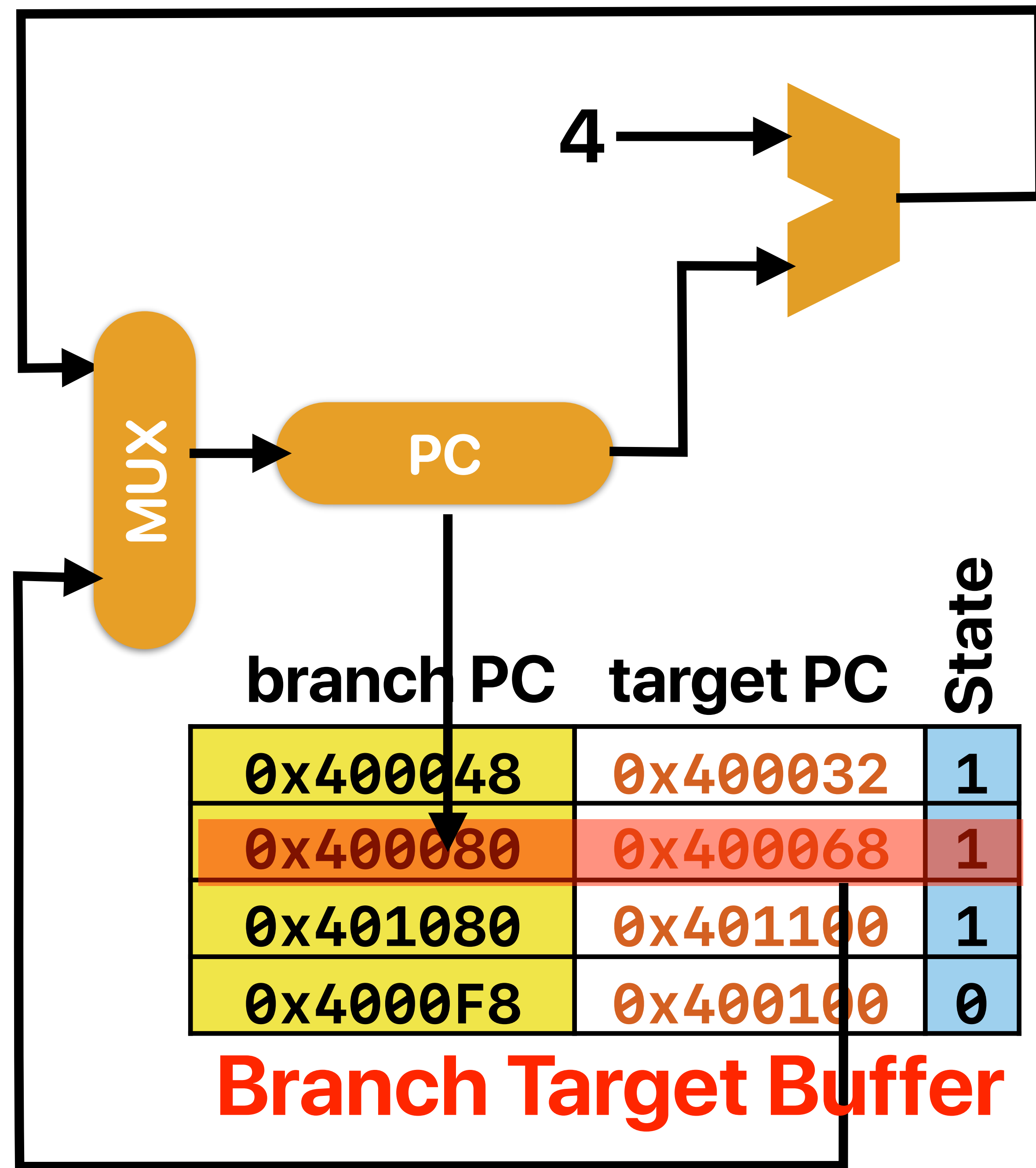
00
01
10
11
10
11
10
11
10
00
00
00
11
10
01
00

Predict Taken

Recap: gshare predictor



Recap: 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

Four implementations

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

A

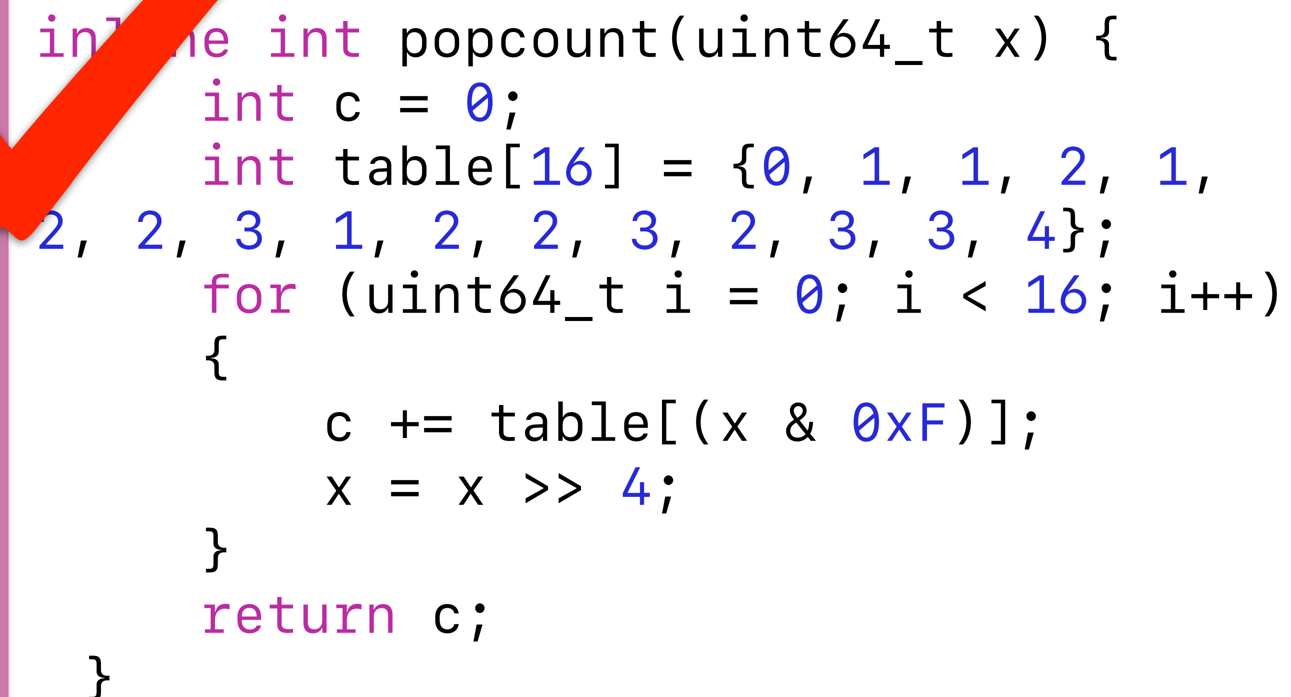
```c
inline int popcount(uint64_t x){
  int c=0;
  while(x)  {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

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;
}
```



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 data hazards 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;
}
```

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;
}
```

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

```
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) branch X
            sum ++;
    }
}
```

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

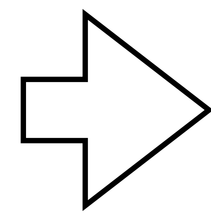
Data hazards

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

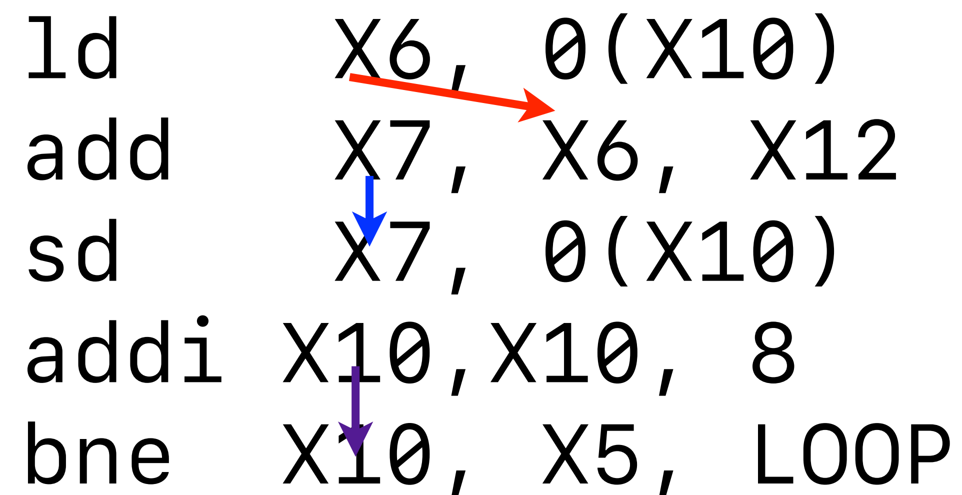
```
i = 0;
do {
    vector[i] += scale;
} while ( ++i < size )
```



```
      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 dependences are there in the following RISC-V instructions?



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

Solution 1: Let’s try “stall” again

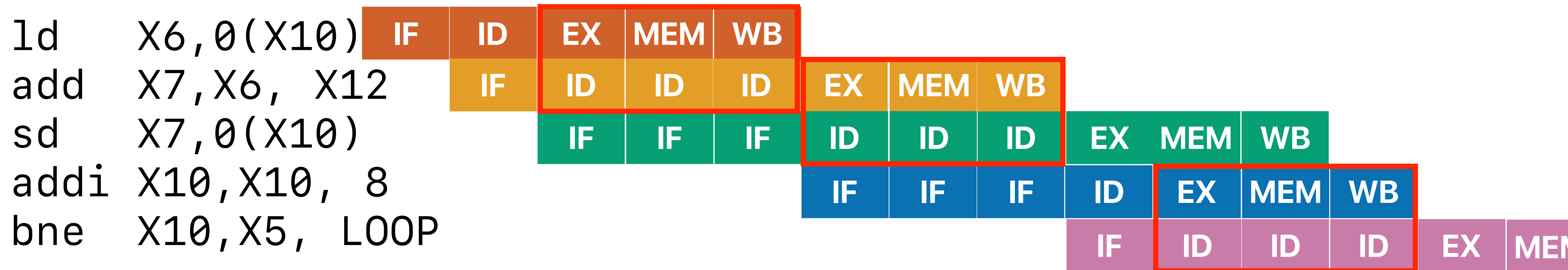
- Whenever the input is not ready when the consumer is decoding, just stall — the consumer stays at ID.

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/

How many of data hazards?

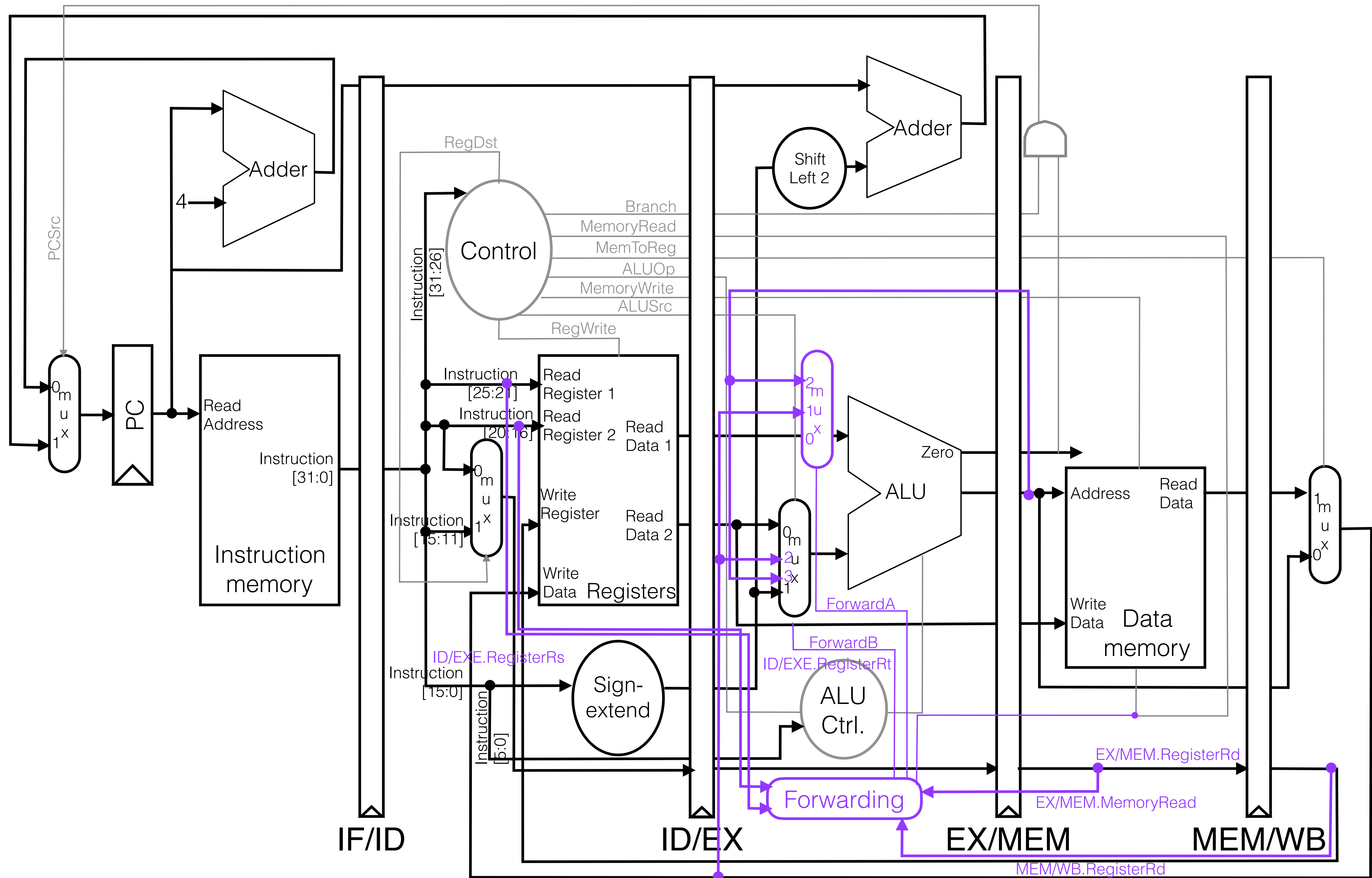
- How many pairs of instructions in the following RISC-V instructions will results in data hazards/stalls in a basic 5-stage RISC-V pipeline?



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

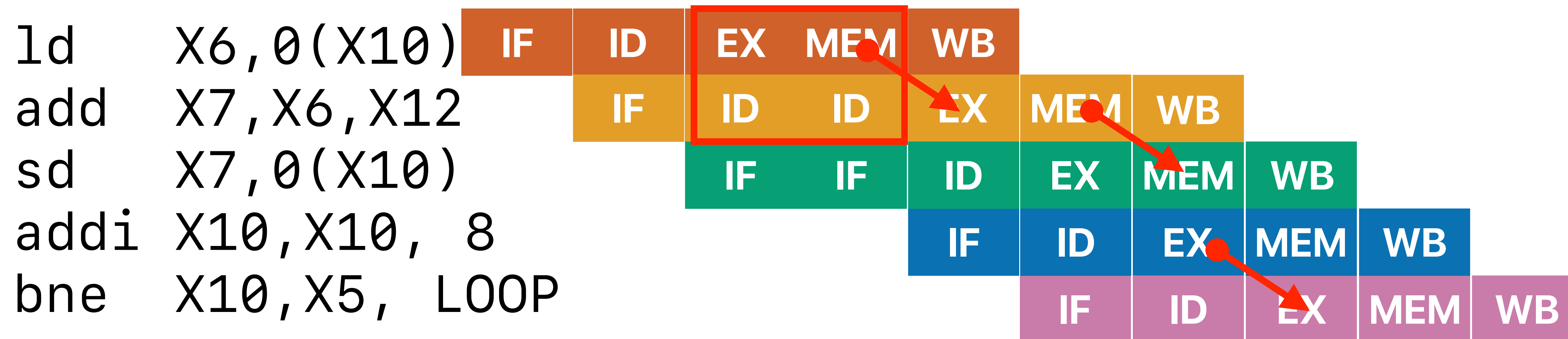
Solution 2: Data forwarding

- 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

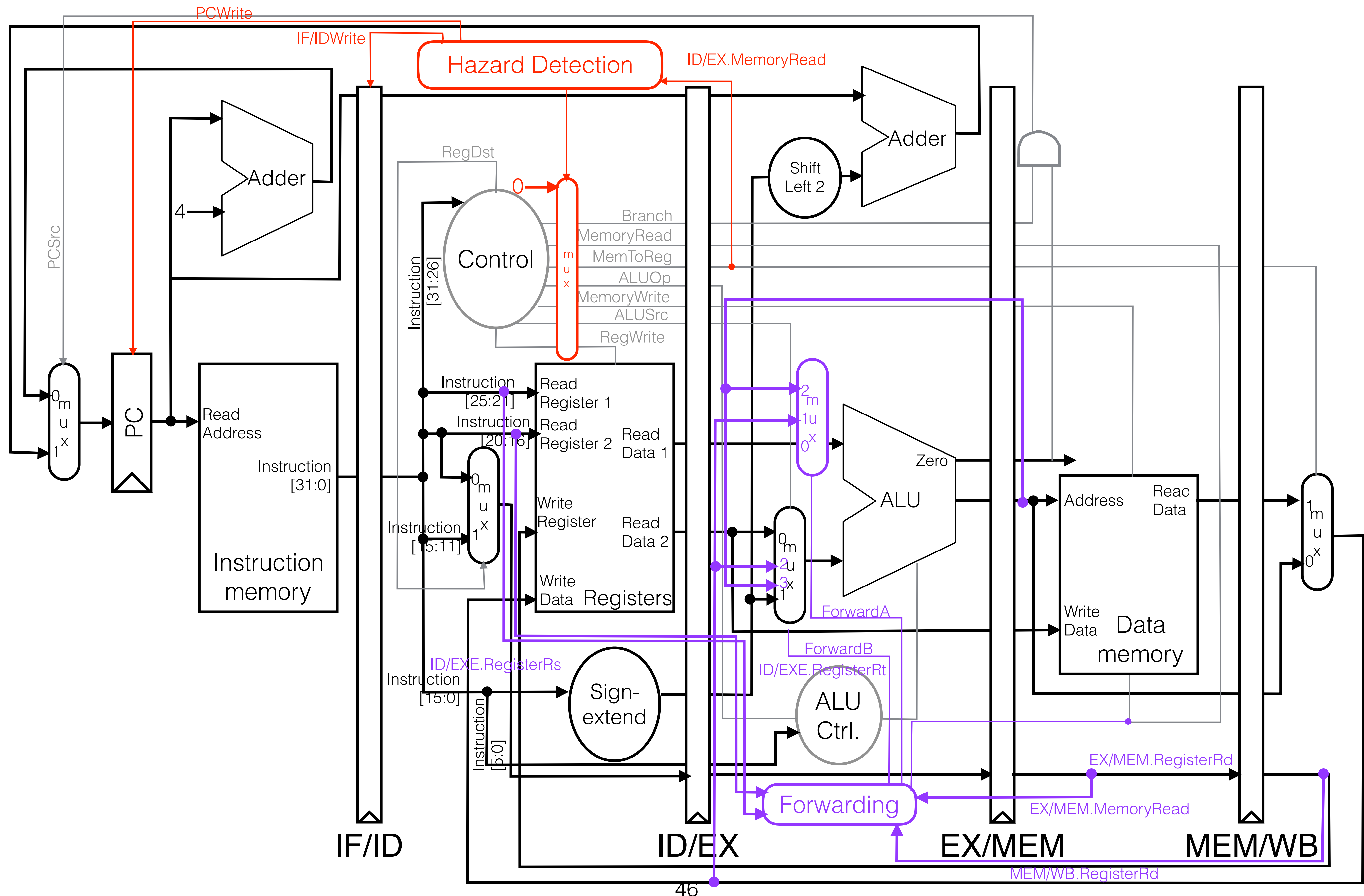

PCSrc
Adder
4
Adder
RegDst
Control
Instruction [31:26]
Branch
MemoryRead
MemToReg
ALUOp
MemoryWrite
ALUSrc
RegWrite
Shift Left 2
Adder
PC
Read Address
Instruction [31:0]
Instruction memory
Instruction [25:21]
Instruction [20:16]
Instruction [15:11]
Read Register 1
Read Register 2
Write Register
Write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

Do we still have to stall?

- 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?



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

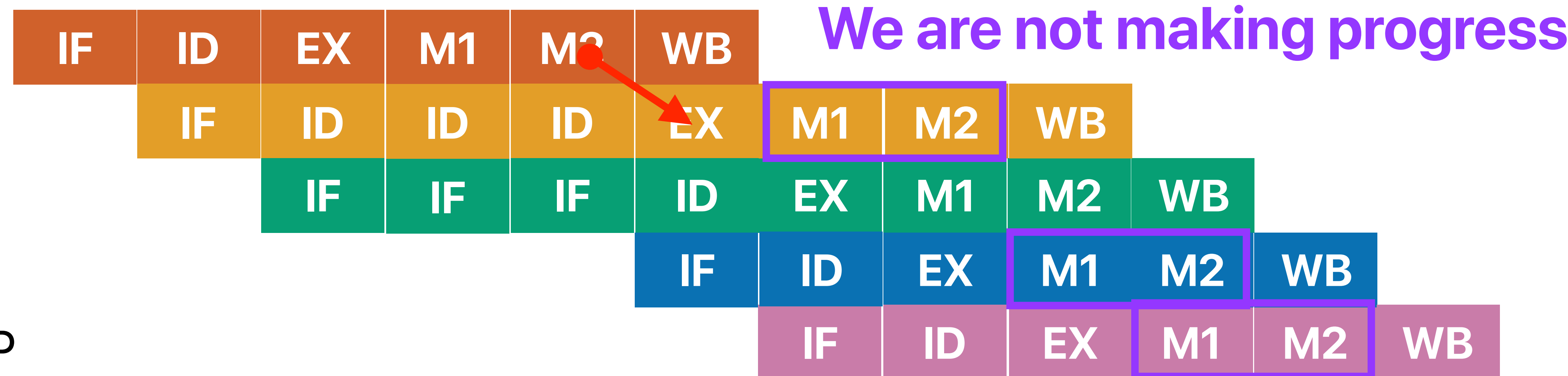


PCWrite
IF/IDWrite
Hazard Detection
ID/EX.MemoryRead
Adder
Adder
4
Shift Left 2
RegDst
0
Control
m u x
Branch
MemoryRead
MemToReg
ALUOp
MemoryWrite
ALUSrc
RegWrite
PCSrc
Instruction [31:26]
PC
Read Address
Instruction [31:0]
Instruction memory
Instruction [25:21]
Instruction [20:16]
Instruction [15:11]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

Problems with data forwarding

- What if our pipeline gets deeper? — Considering a newly designed pipeline where memory stage is split into 2 stages and the memory access finishes at the 2nd memory stage. 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
```



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

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
```

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

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)

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

Announcement

- Homework #3 due Wednesday
- iEval submission — attach your "confirmation" screen, you get an extra/bonus homework
- Project due on 12/2 — roughly three weeks from now
  - You can only turn-in "helper.c"
    - `mcfutil.c:refresh_potential()` creates helper threads
    - `mcfutil.c:refresh_potential()` calls `helper_thread_sync()` function periodically
    - It's your task to think what to do in `helper_thread_sync()` and `helper_thread()` functions
    - Please DO READ papers before you ask what to do
  - Formula for grading — min(100, speedup*100)
  - No extension
- Office hour for Hung-Wei this week — MWF 1p-2p — no office hour next week