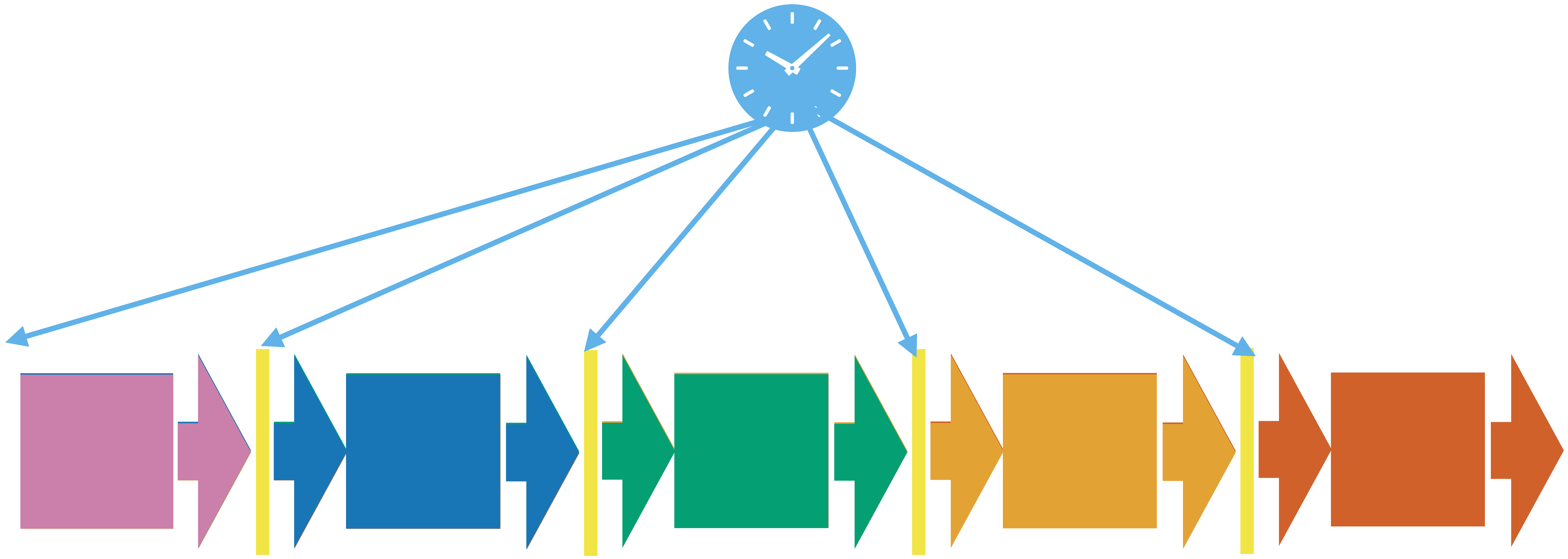
Dynamic Branch Prediction

Hung-Wei Tseng

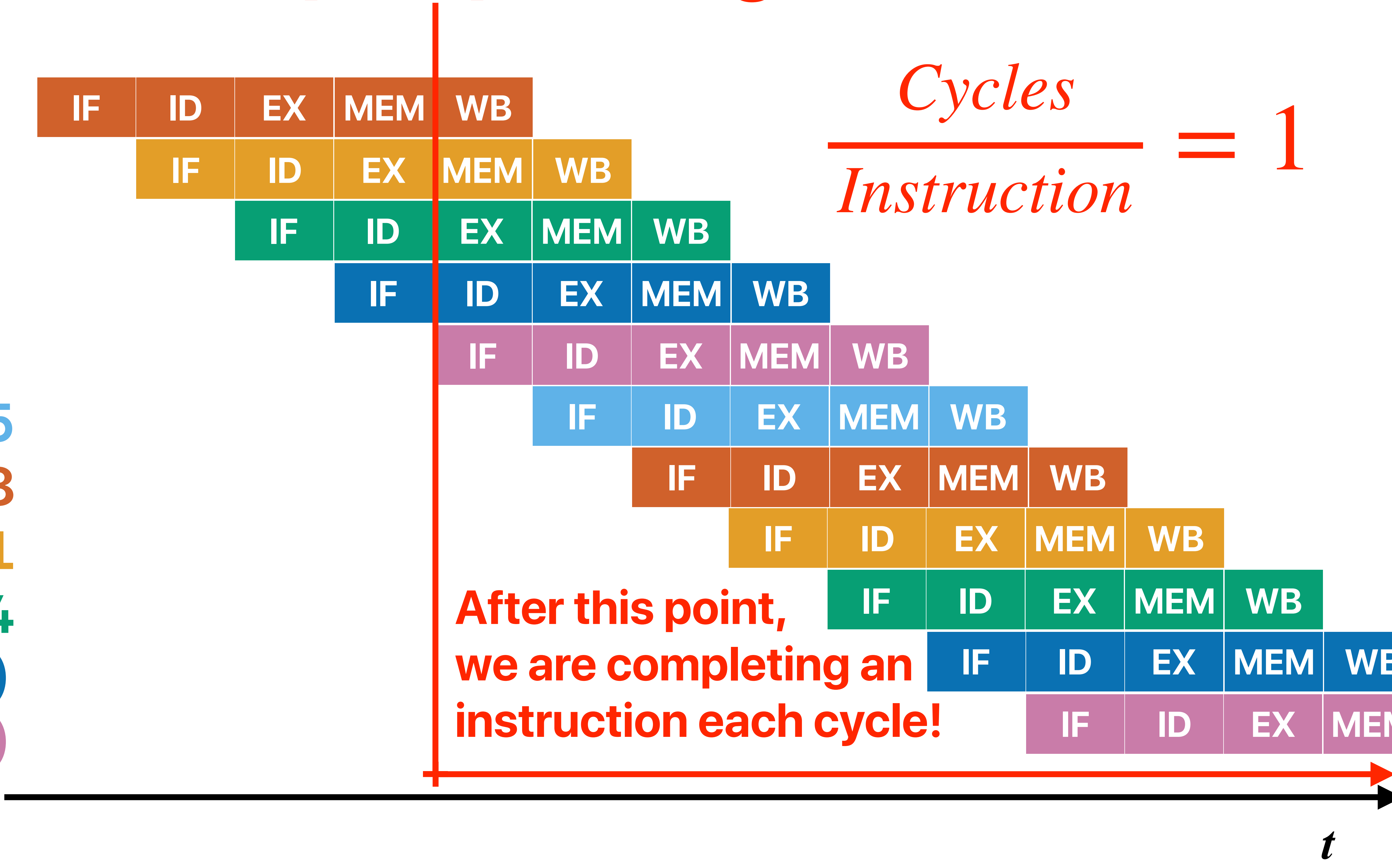
Recap: Pipelining

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)
```



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

After this point, we are completing an instruction each cycle!

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: Solving Structural Hazards

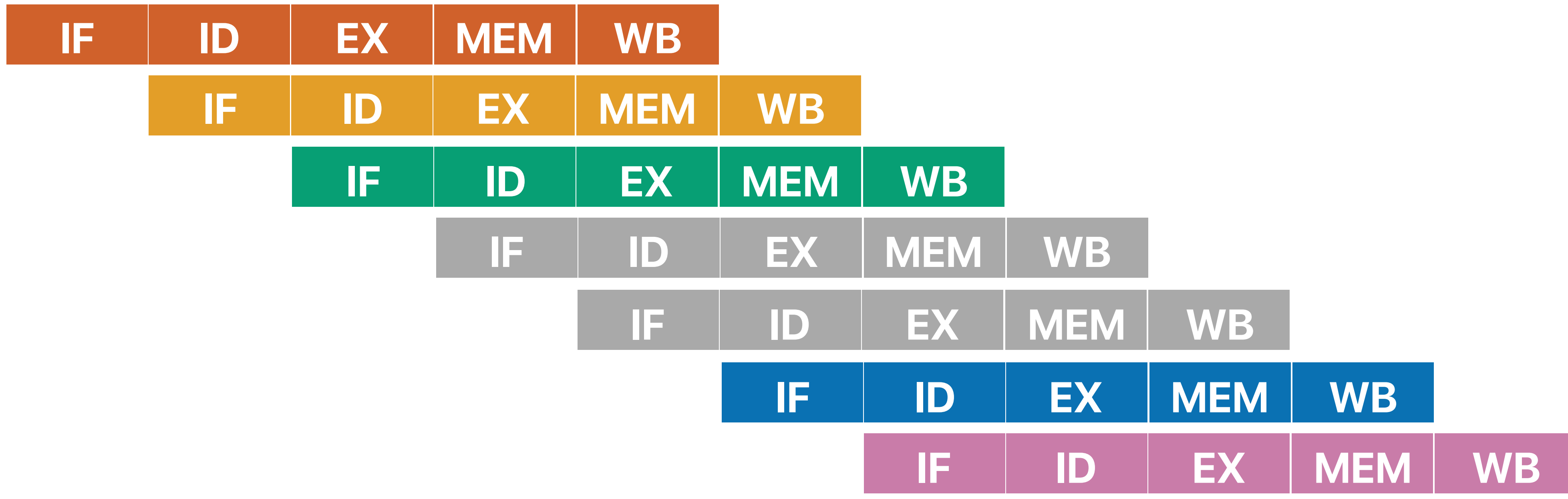
- Stall can address the issue — but slow
- Improve the pipeline unit design to allow parallel execution

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)
```



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

Outline

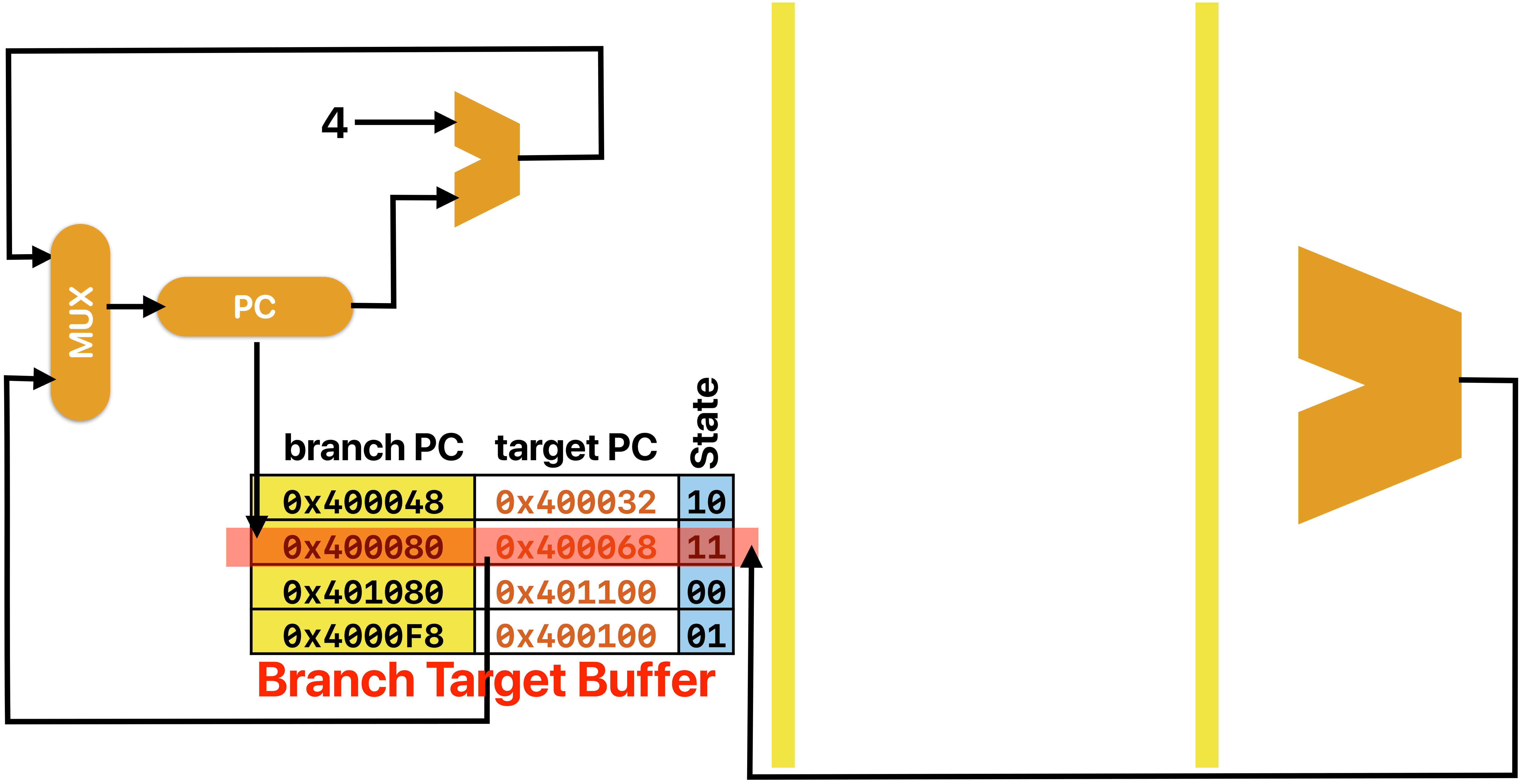
- 2-bit local predictor
- 2-level global predictor
- Hybrid predictors
- Branch and coding

Dynamic Branch Prediction

Why can't we proceed without stalls/no-ops?

- How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor
  - ✓ ① The target address when branch is taken is not available for instruction fetch stage of the next cycle **You need a cheatsheet for that — branch target buffer**
  - ② The target address when branch is not-taken is not available for instruction fetch stage of the next cycle **You need to predict that — history/states**
  - ✓ ③ The branch outcome cannot be decided until the comparison result of ALU is not out
  - ④ The next instruction needs the branch instruction to write back its result
  - A. 0
  - B. 1
  - [✓ C. 2]
  - D. 3
  - E. 4

A basic dynamic branch predictor

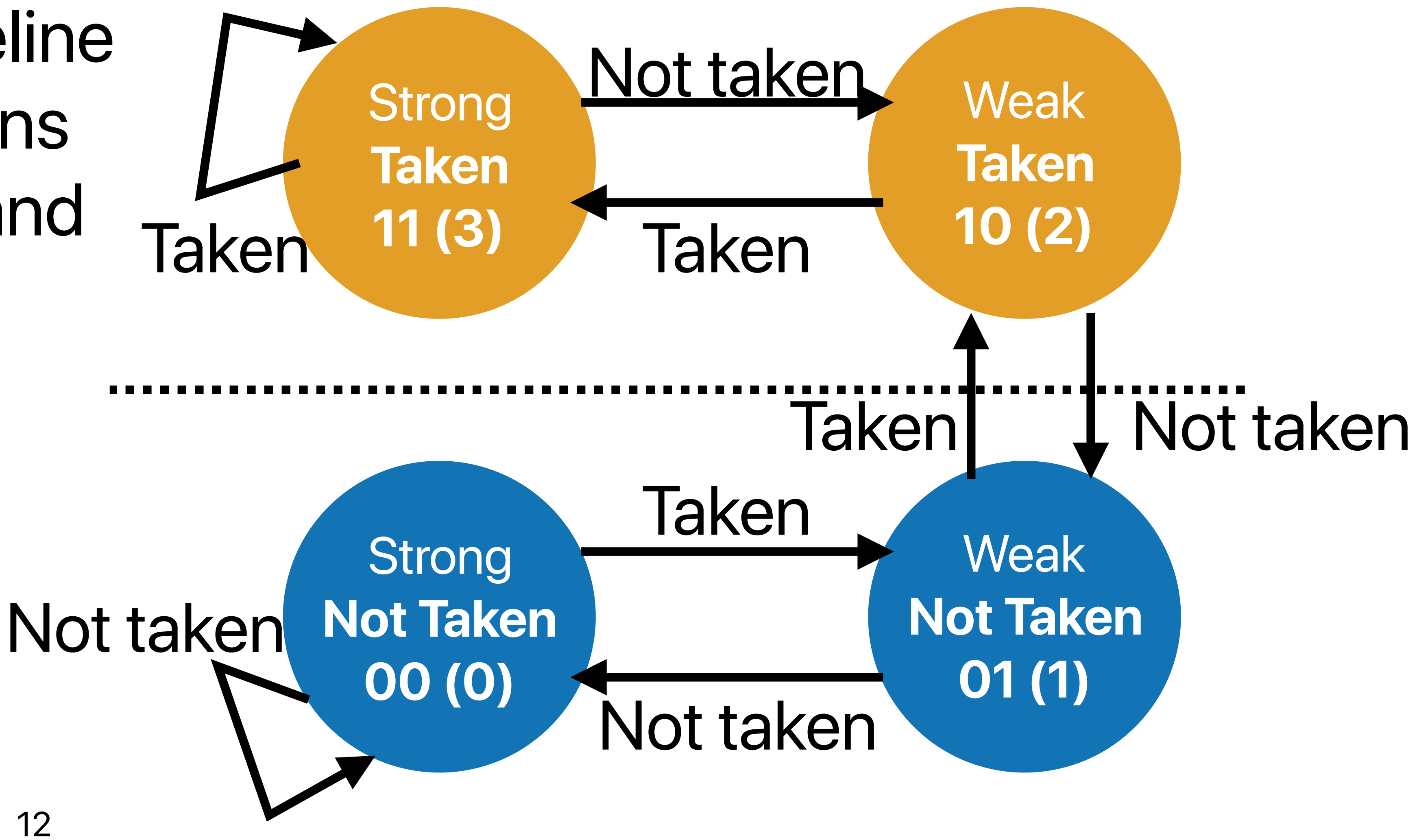


branch PC	target PC	State
0x400048	0x400032	10
0x400080	0x400068	11
0x401080	0x401100	00
0x4000F8	0x400100	01

2-bit/Bimodal local predictor

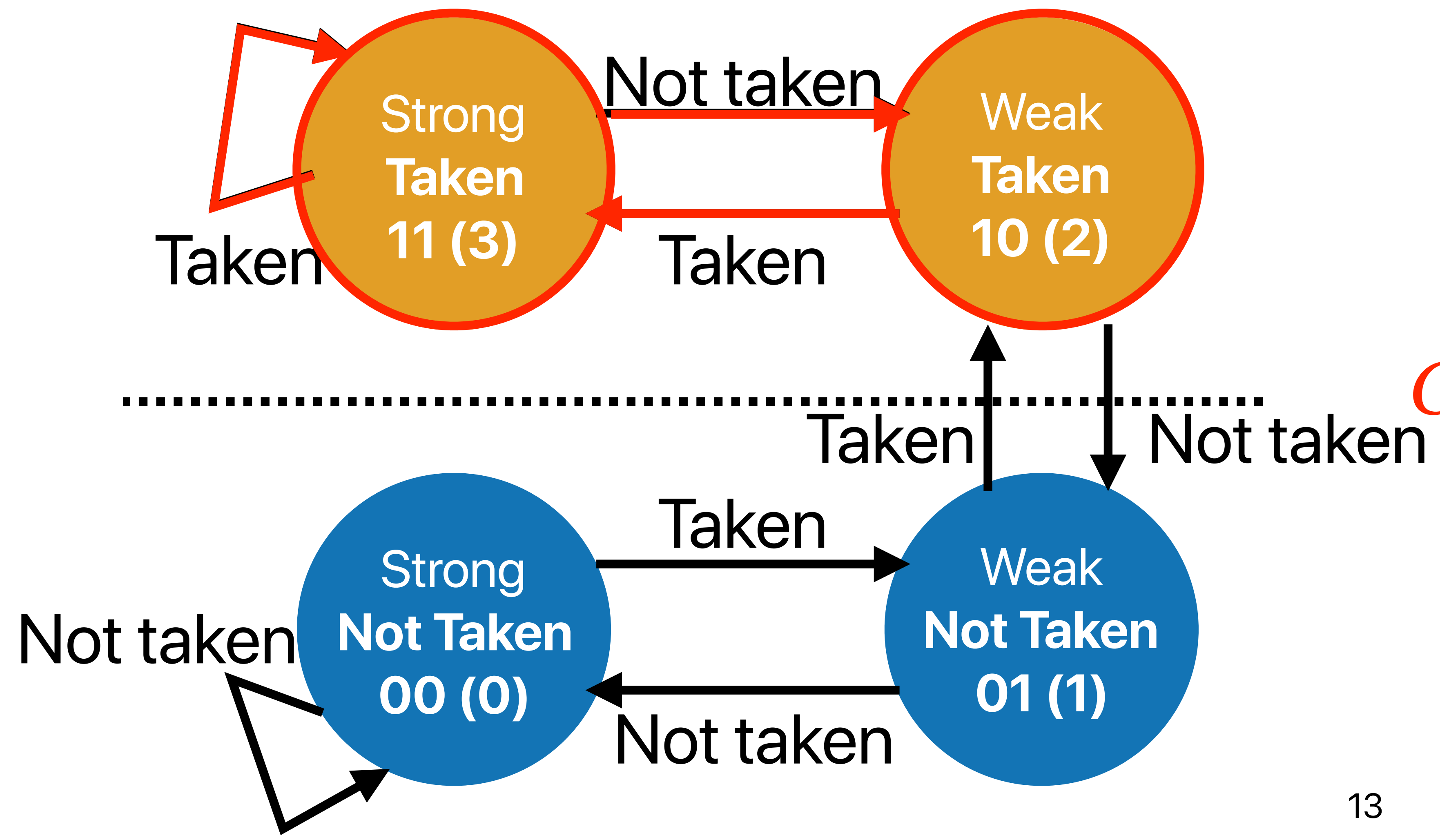
- Local predictor — every branch instruction has its own state
- 2-bit — each state is described using 2 bits
- Change the state based on **actual** outcome
- If we guess right — no penalty
- If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF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



2-bit local predictor

```
i = 0;
do {
      sum += a[i];
} while(++i < 10);
```



i	state	predict	actual
1	10	T	T
2	11	T	T
3	11	T	T
4-9	11	T	T
10	11	T	NT

90% accuracy!

$$CPI_{average} = 1 + 20\% \times 10\% \times 2 = 1.04$$

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

Two-level global predictor

Reading: Scott McFarling. Combining Branch Predictors. Technical report WRL-TN-36, 1993.

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
```

(assume all states started with 00)

A. ~25%

B. ~33%

C. ~50%

D. ~67%

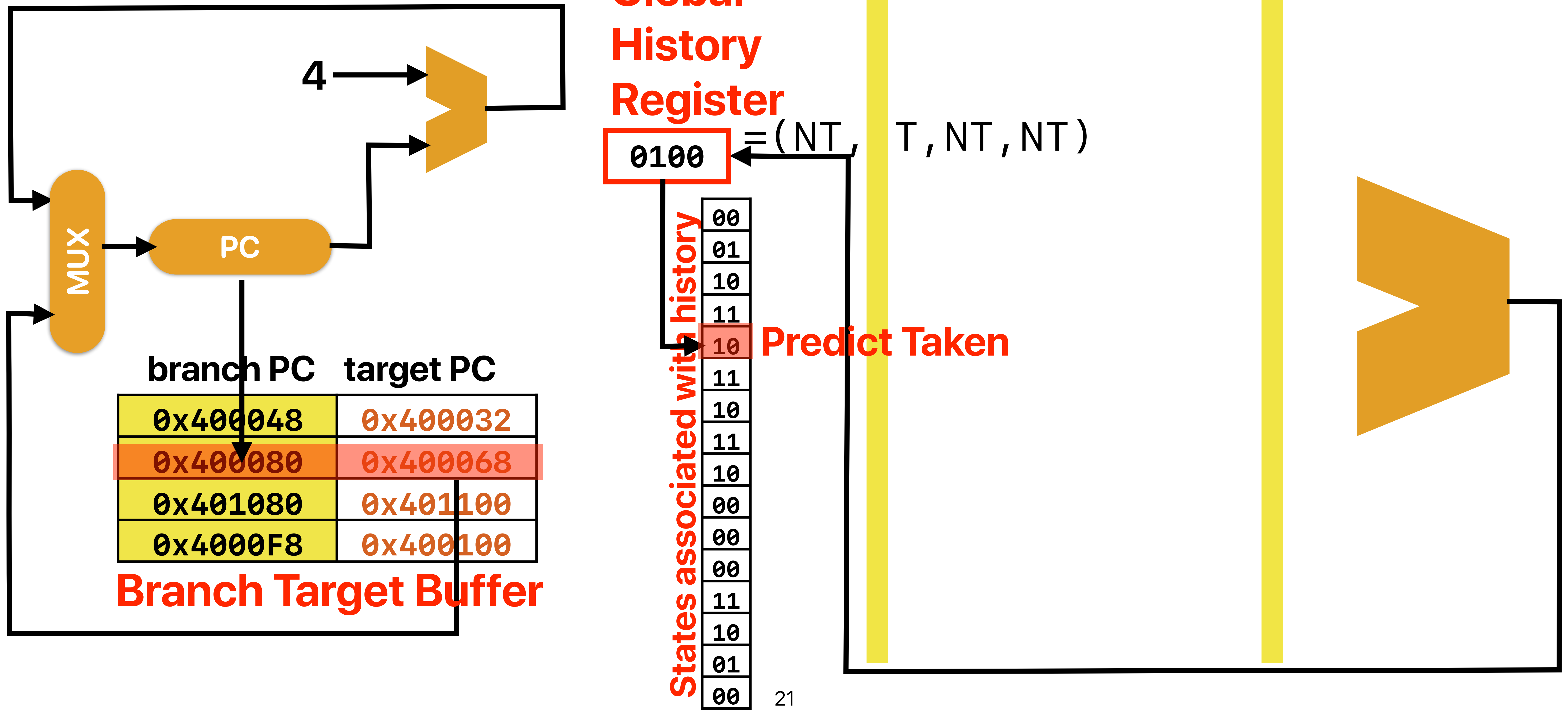
[✓ E. ~75%]

This pattern repeats all the time!

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



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

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
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

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

about the same

```
i = 0;
do {
    if( i % 10 != 0)
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100);
```

II

about the same

```
i = 0;
do {
    a[i] += i;
} while ( ++i < 100);
```

III

✓

```
i = 0;
do {
    j = 0;
    do {
      sum += A[i*2+j];
    }
    while( ++j < 2);
} while ( ++i < 100);
```

IV

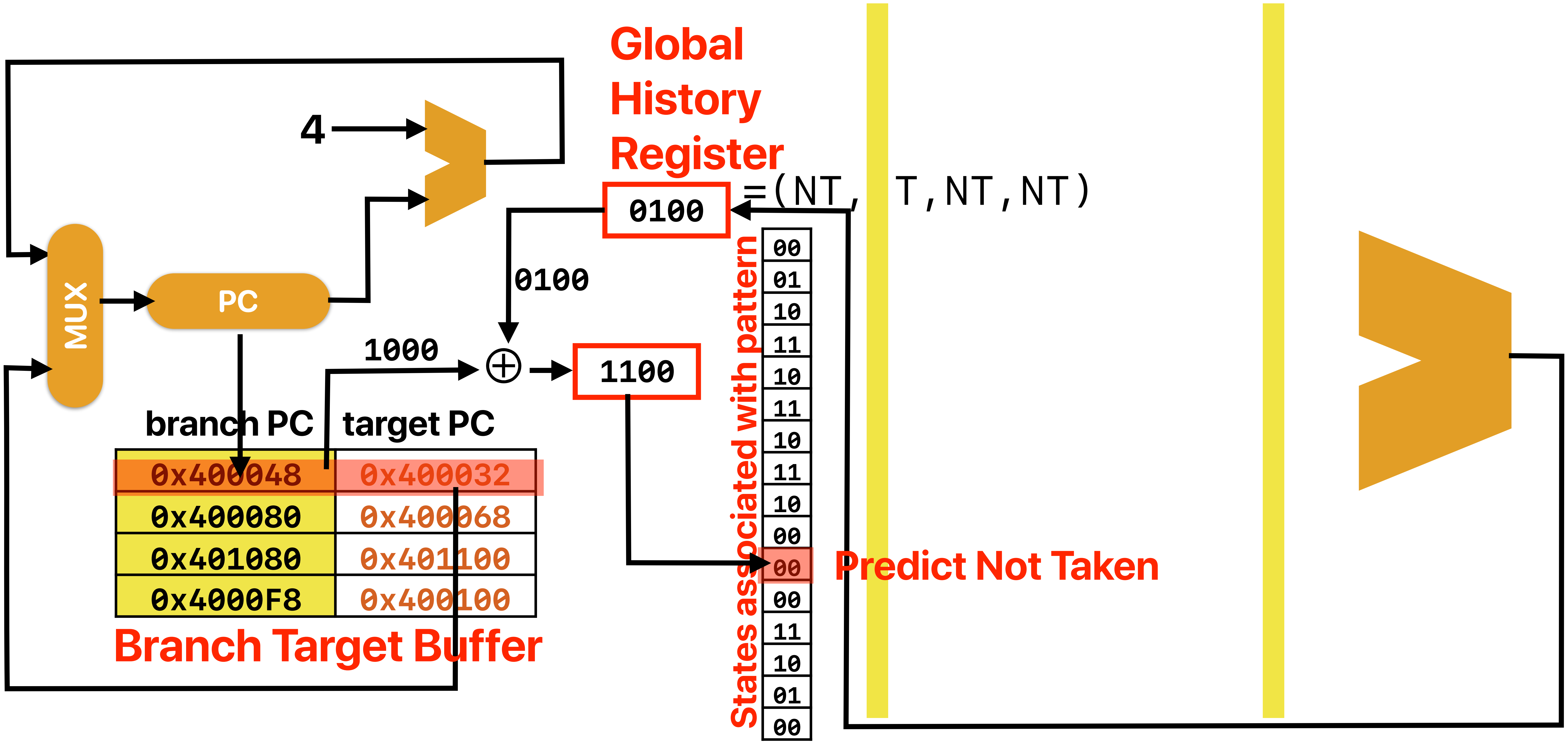
L could be bette

```
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



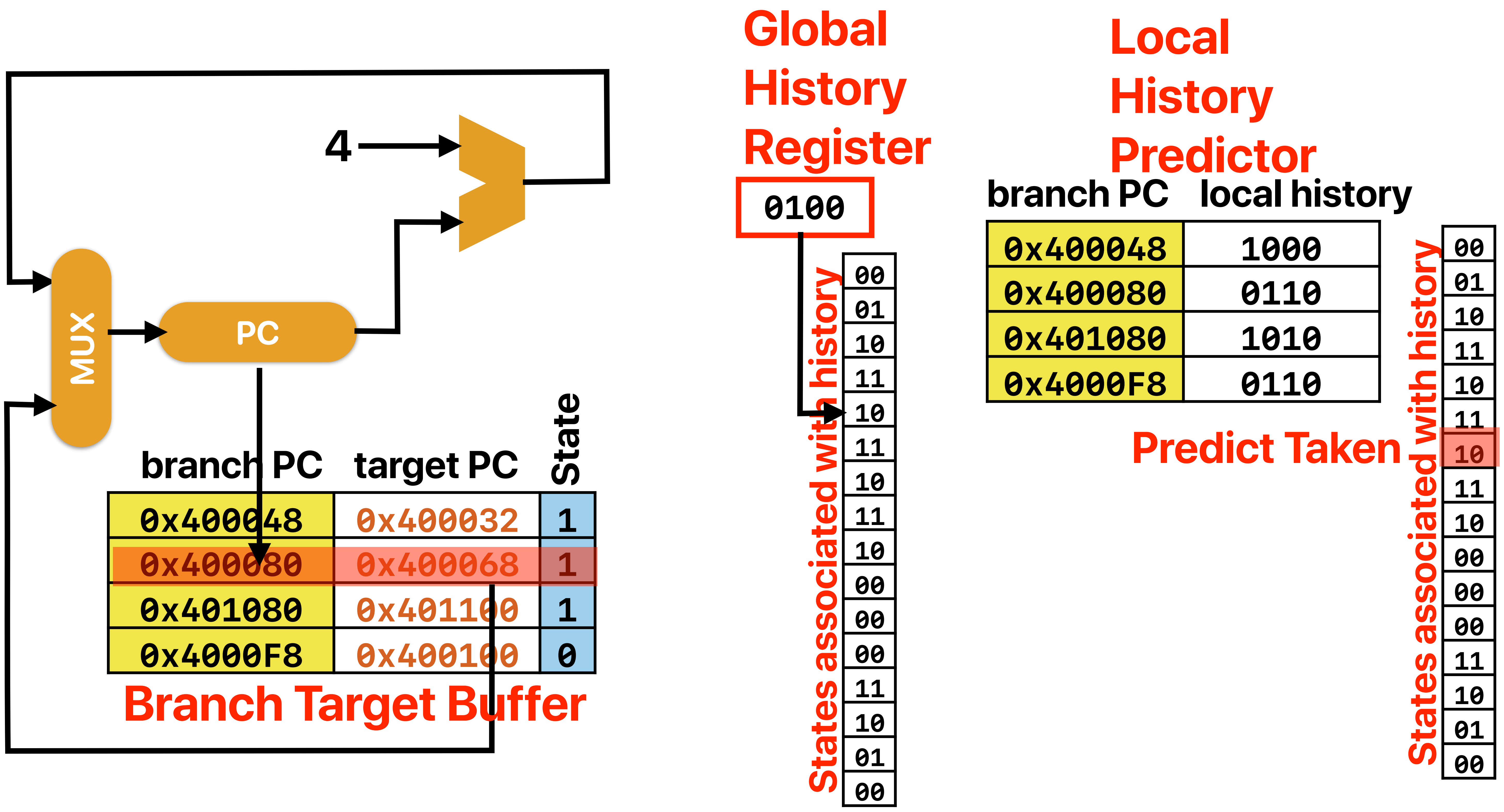
branch PC	target PC
0x400048	0x400032
0x400080	0x400068
0x401080	0x401100
0x4000F8	0x400100

Branch Target Buffer

gshare predictor

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

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

States associated with history: 00, 01, 10, 11, 10, 11, 10, 11, 10, 00, 00, 00, 11, 10, 01, 00

Predict Taken

Tournament Predictor

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

Branch predictor 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?

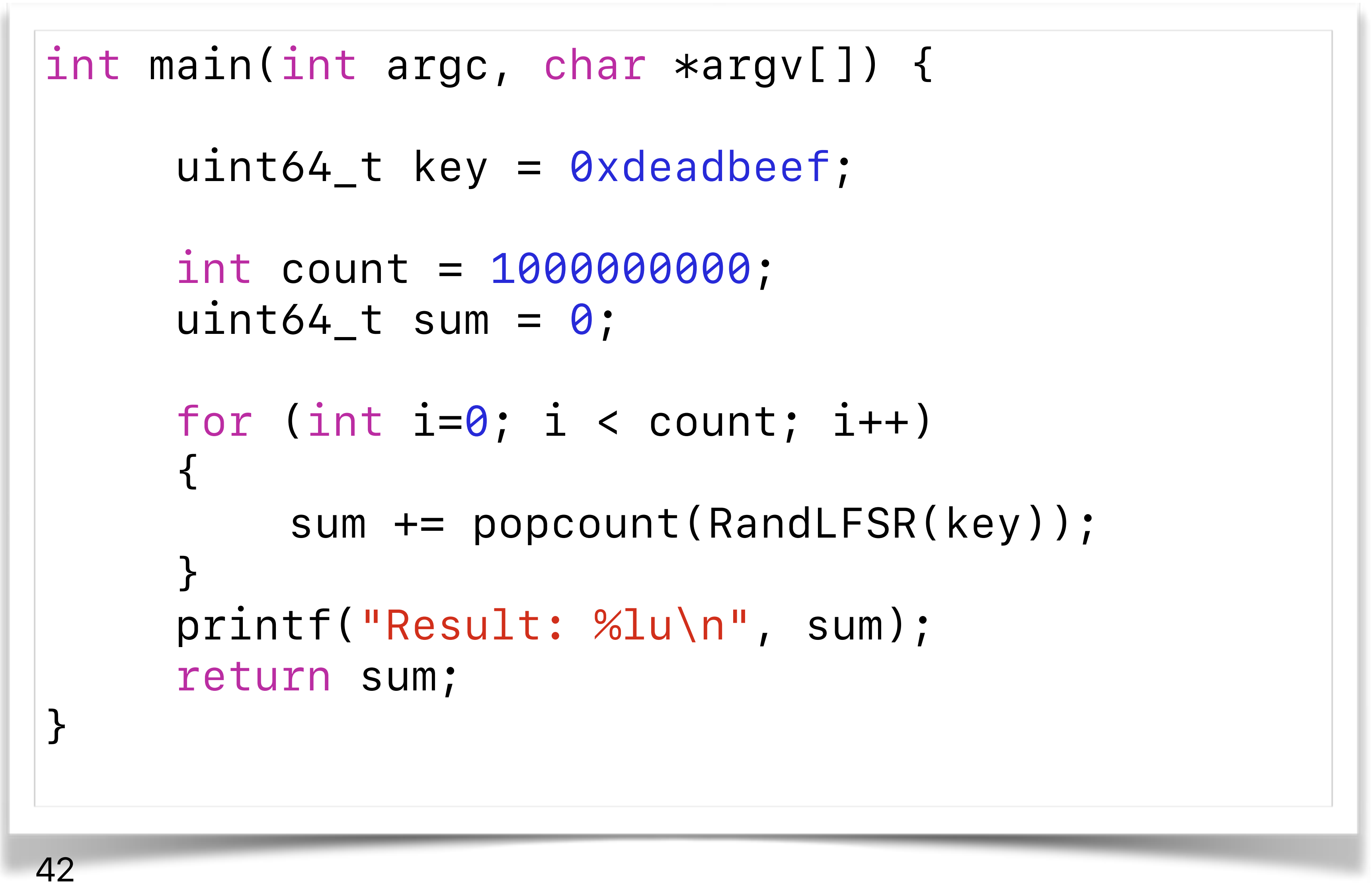
```
if(option)
    std::sort(data, data + arraySize);

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;
}
```

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;
}
```

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;
}
```

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 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 hazards

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 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 hazards — 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;
}
```

Announcement

- Reading quiz due next Monday
- Homework #3 due next Wednesday
- 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