Dynamic Branch Prediction

Hung-Wei Tseng

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
  - ③ The branch outcome cannot be decided until the comparison result of ALU is not out **You need to predict that — history/states**
  - ④ The next instruction needs the branch instruction to write back its result

A. 0

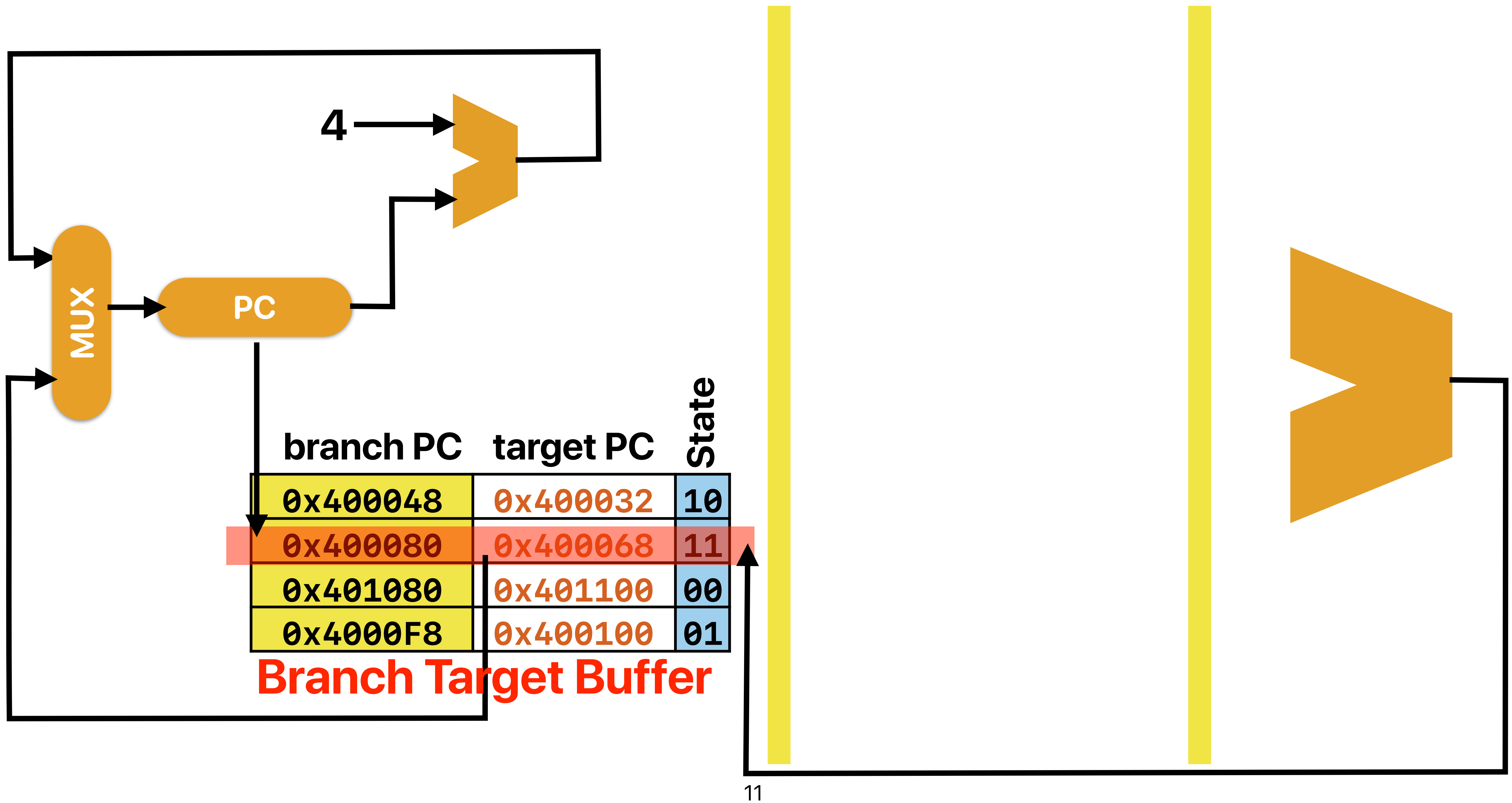
B. 1

C. 2

D. 3

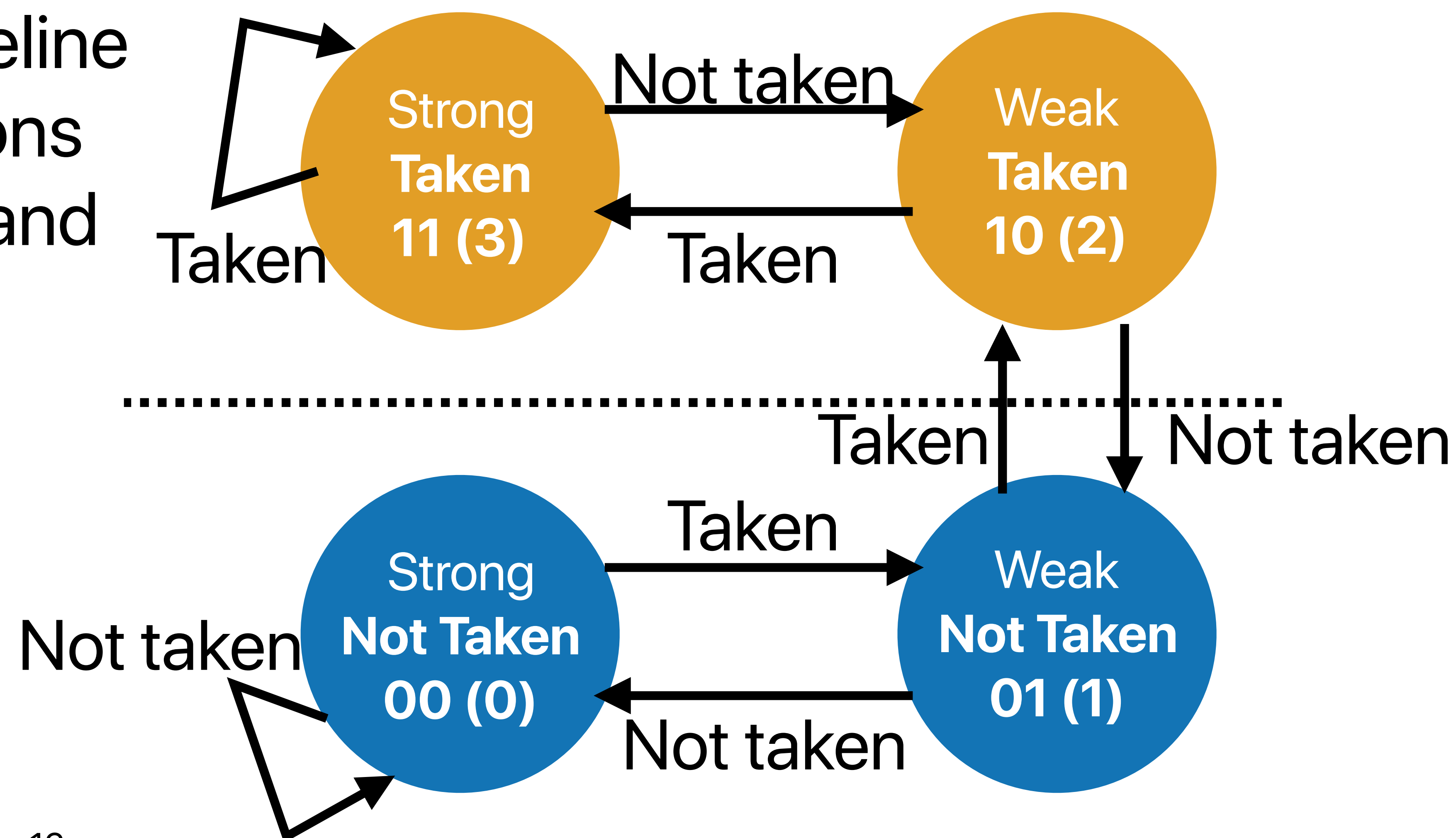
E. 4

A basic dynamic branch predictor



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

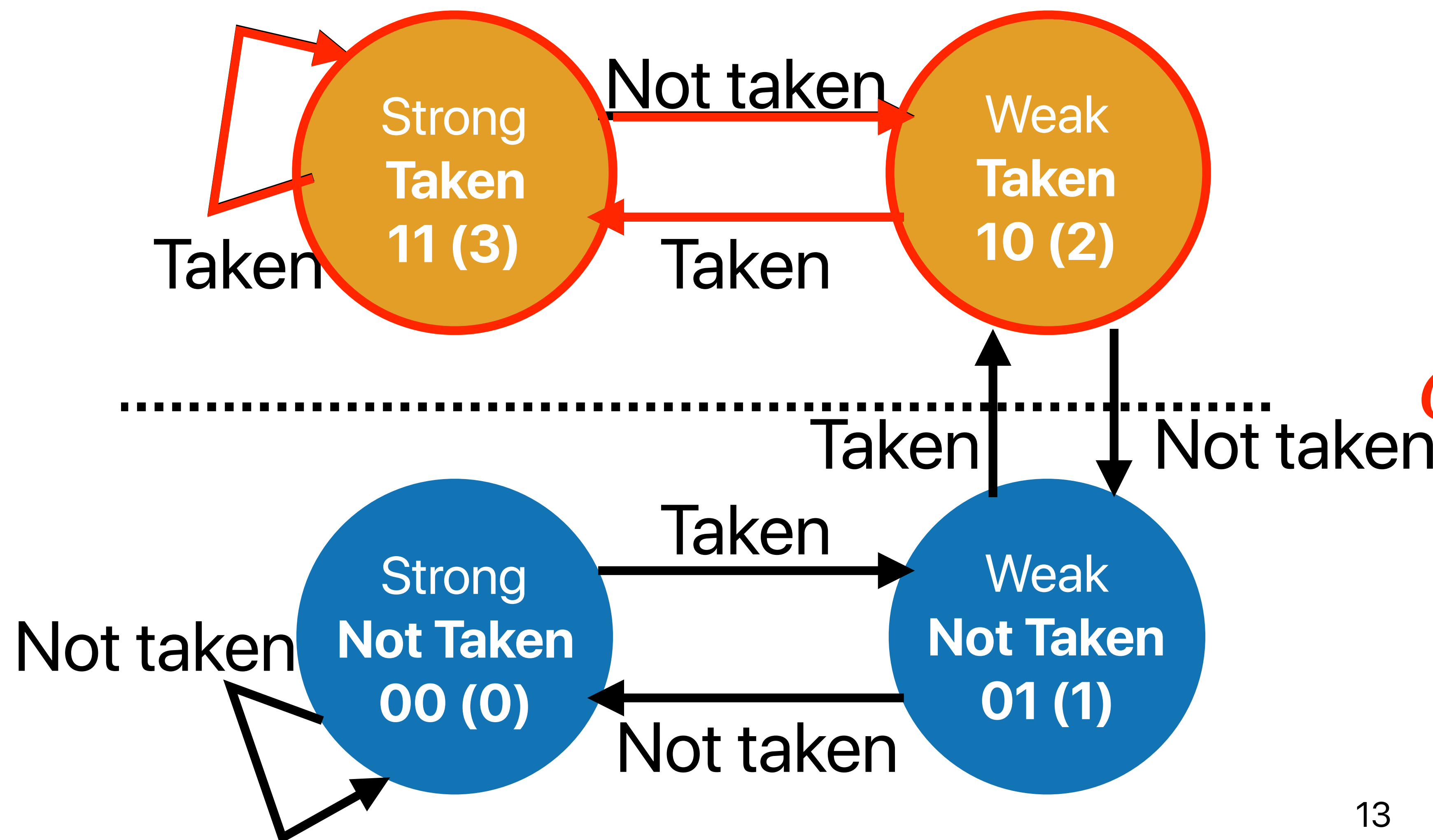


branch PC	target PC	State
0x400048	0x400032	10
0x400080	0x400068	11
0x401080	0x401100	00
0x4000F8	0x400100	01

Predict Taken

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!

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

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
```

This pattern repeats all the time!

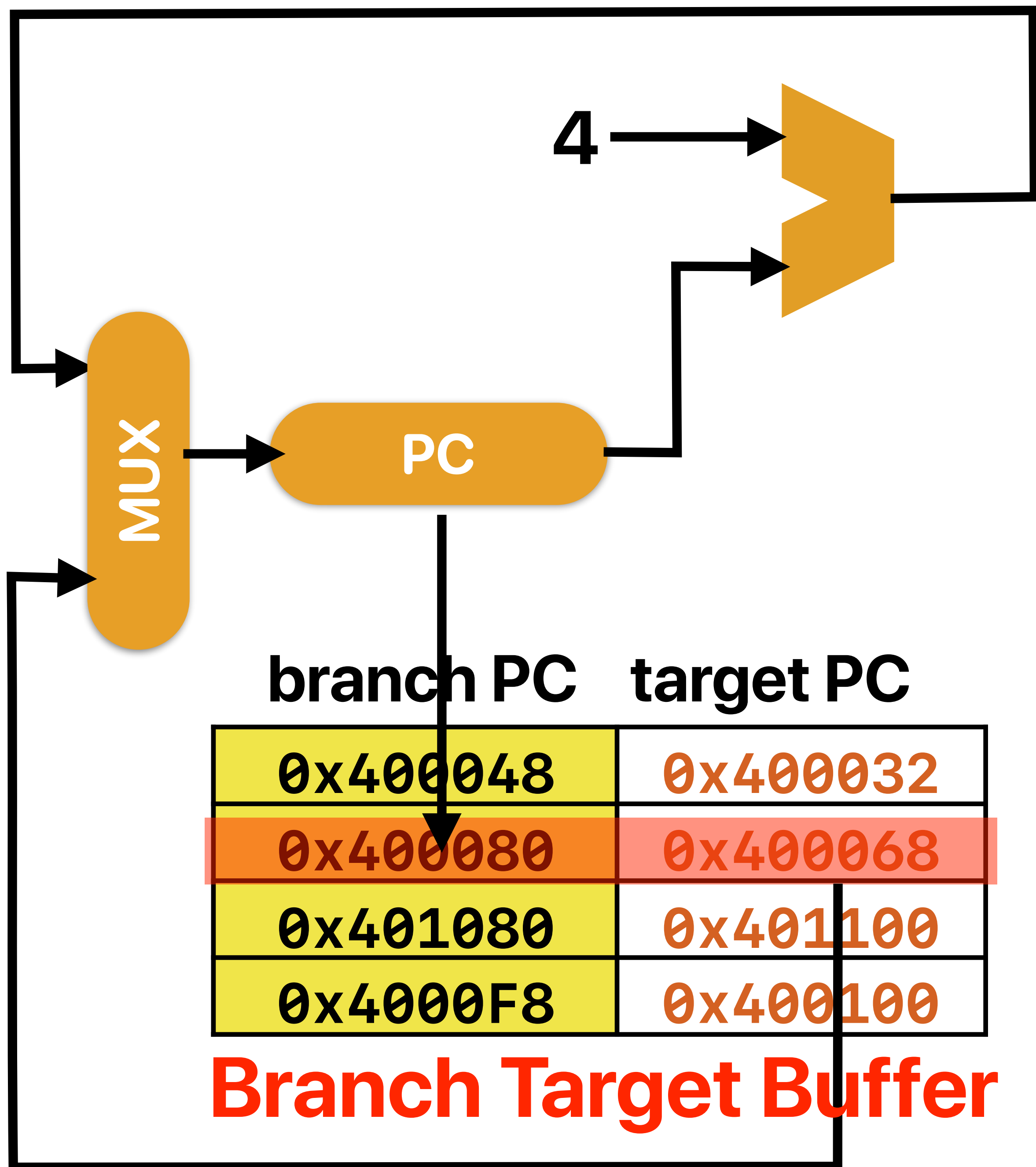
(assume all states started with 00)

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	NT	T
4	X	00	NT	T
4	Y	10	T	T
5	X	01	NT	NT
5	Y	11	NT	T
6	X	00	NT	T
6	Y	10	T	T

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

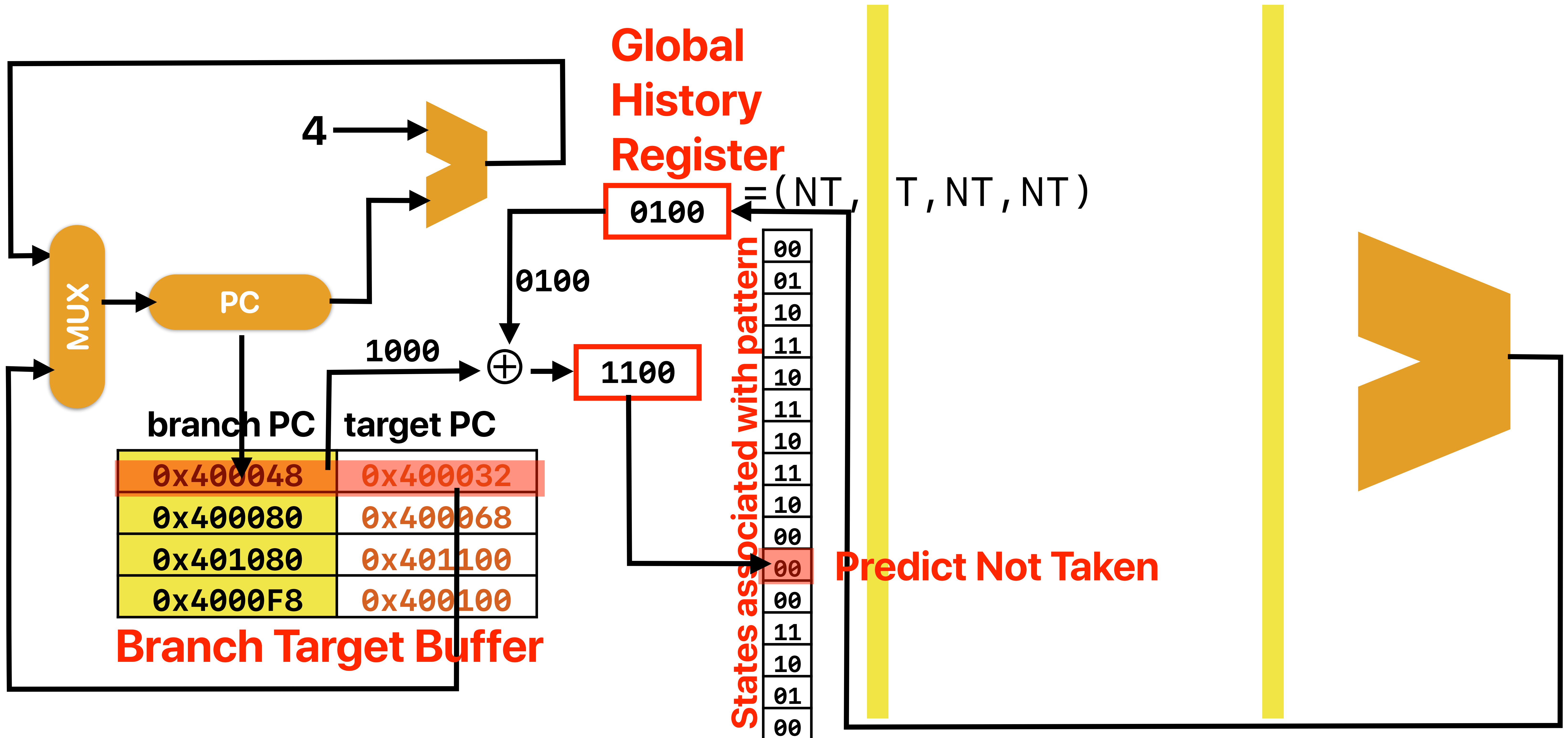
Performance of GH predictor

```
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

Hybrid predictors

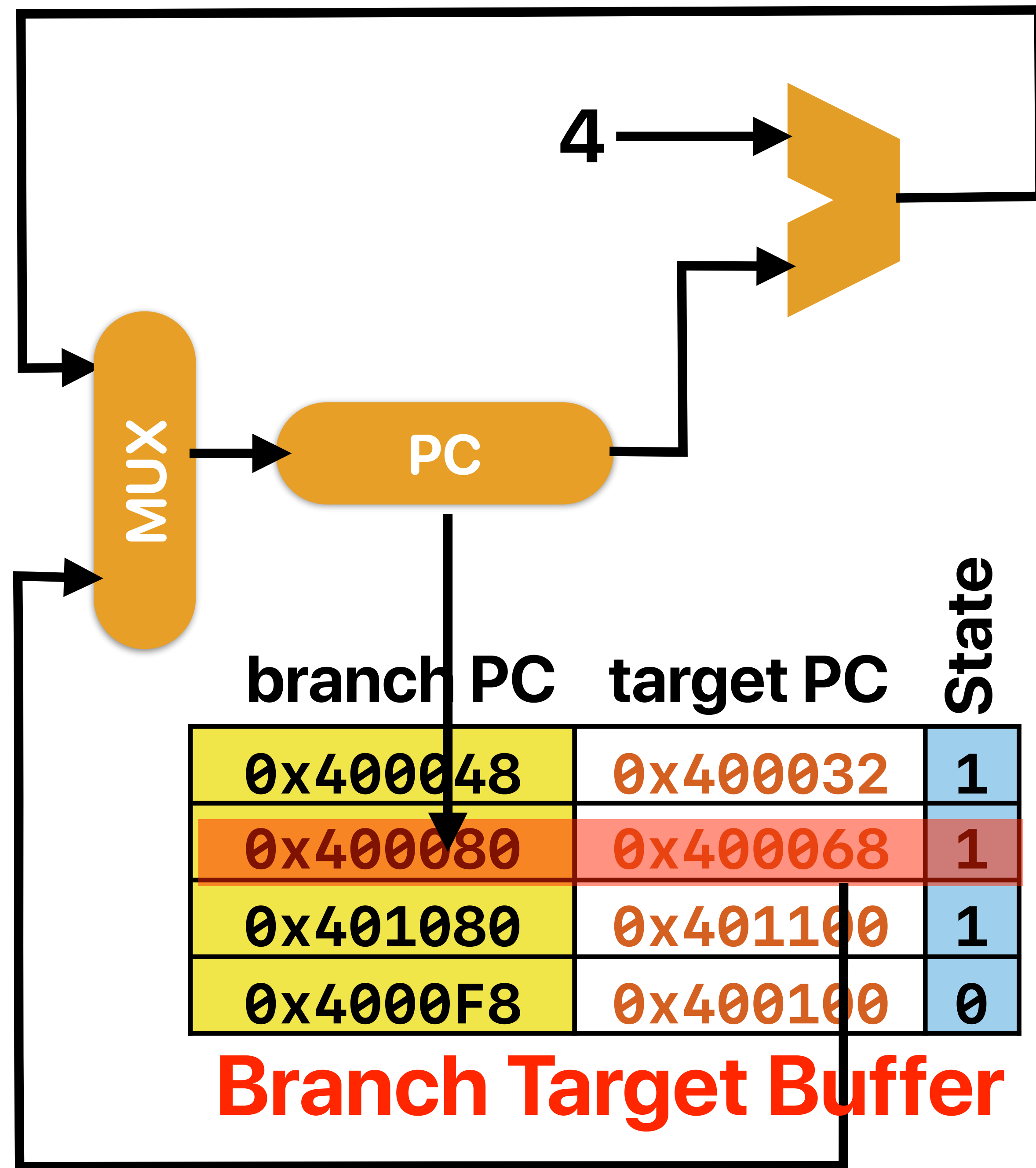
Gshare predictor



GShare predictor

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

Tournament Predictor



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

Tournament Predictor

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

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: pop count

- Given a 64-bit integer number, find the number of 1s in its binary representation.
- Example 1: — Very useful for Hamming distance, sparse matrix

  Input: 9487

  Output: 7

  Explanation: 9487's binary representation is 0b10010100001111

```c
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;
}
```