Processor Design (III)

Hung-Wei Tseng

Branch prediction to reduce the overhead of control hazards

Why do we need to stall for branch instructions

- 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**
  - 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 out **You need to predict that**
  - ④ The next instruction needs the branch instruction to write back its result

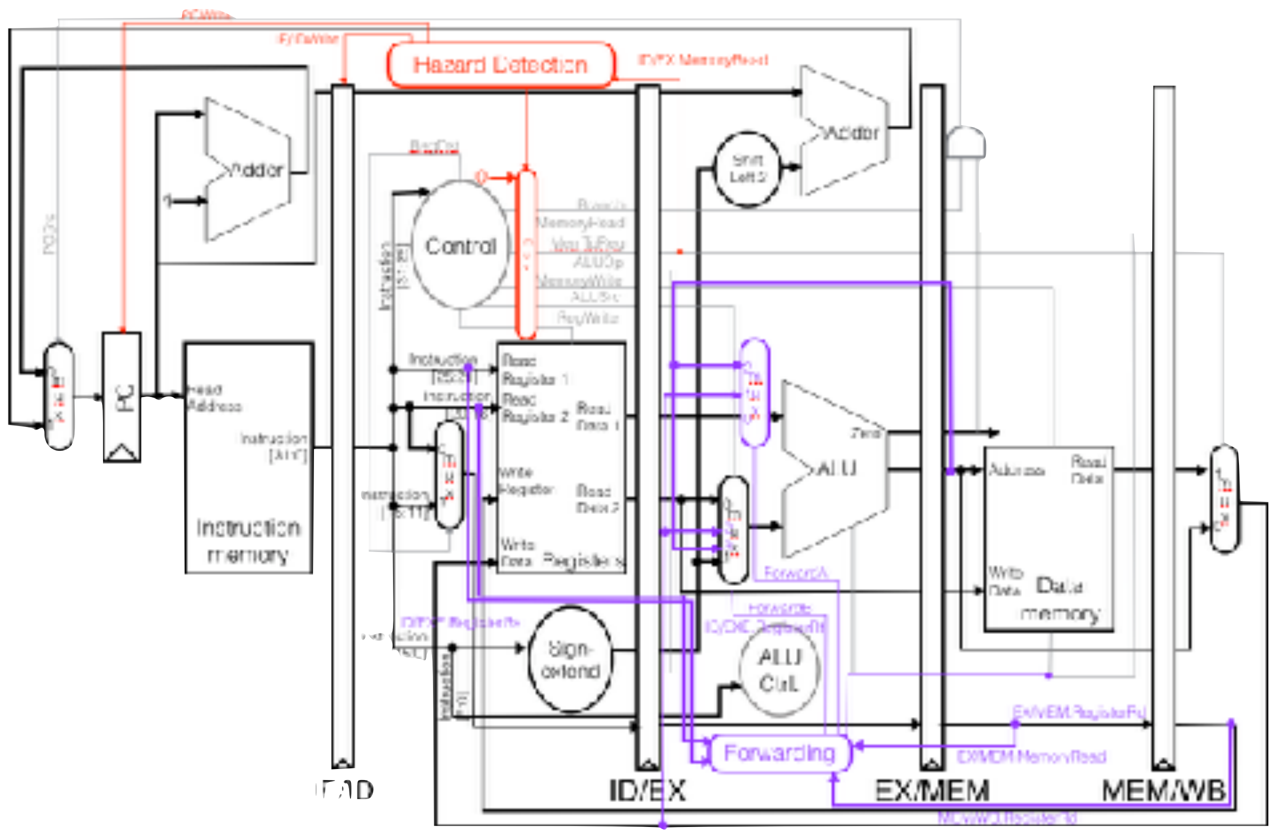
A. 0

B. 1

C. 2

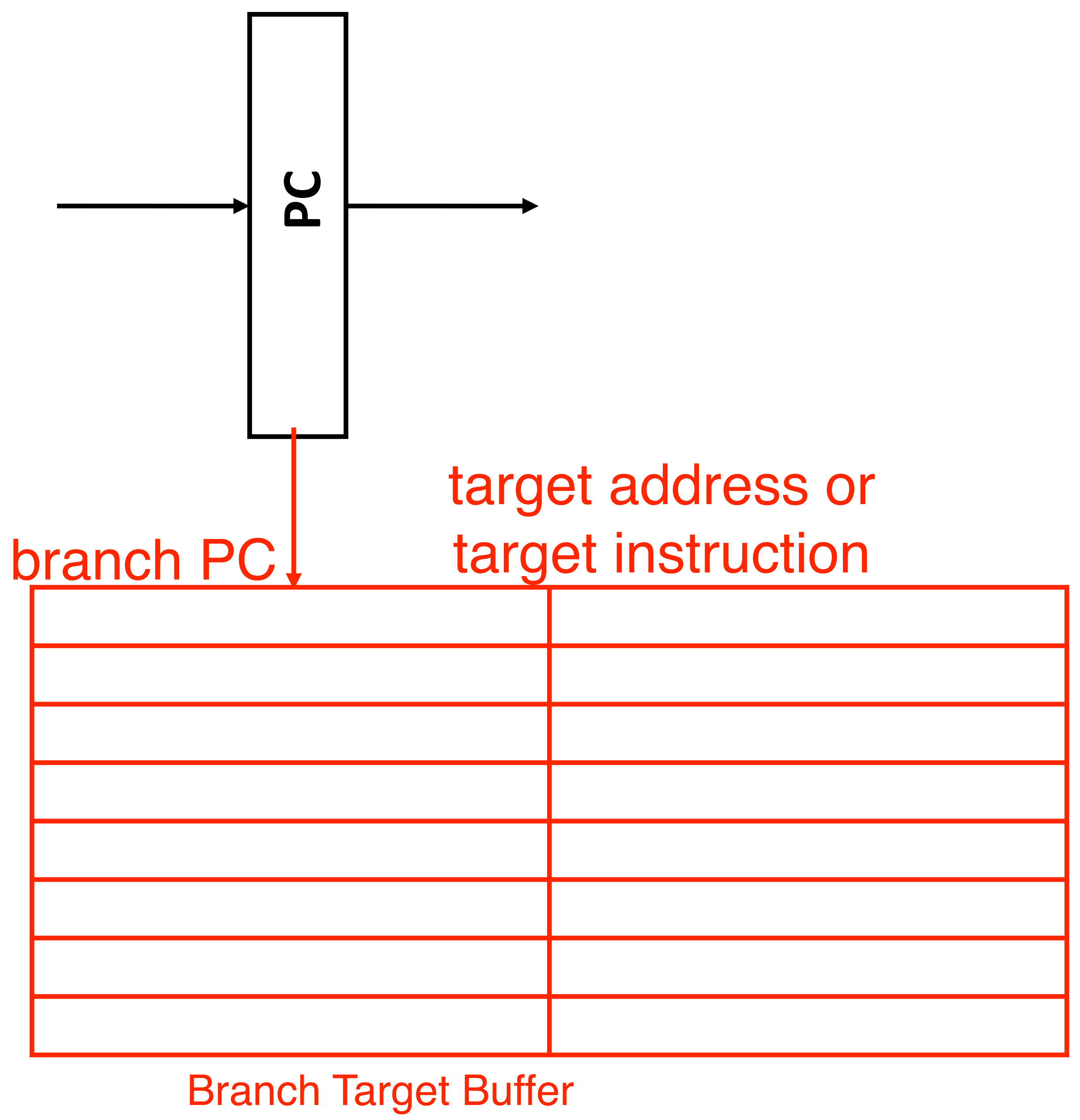
D. 3

E. 4



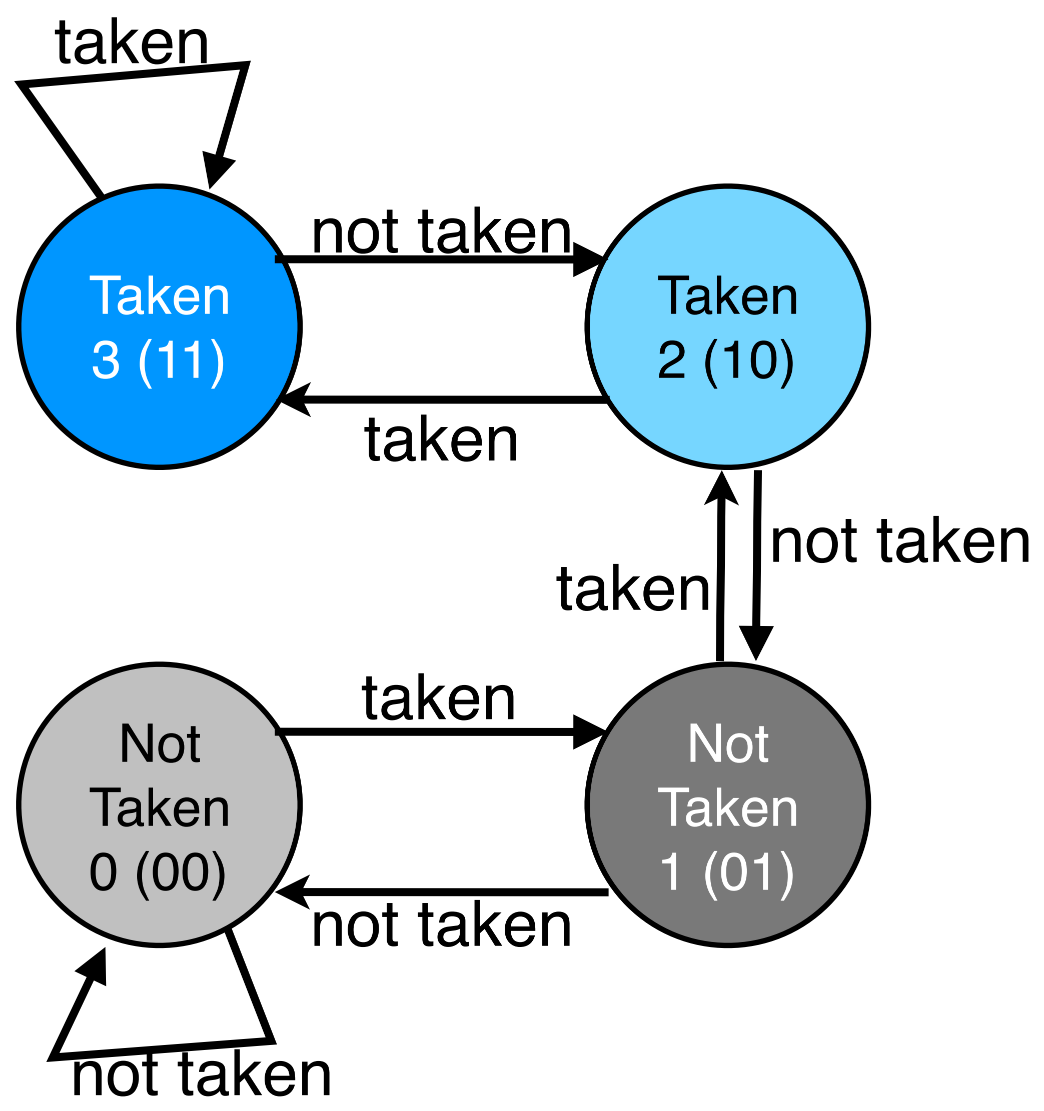
Branch Target Buffer

- The processor needs a “cheat sheet” for where the branch is going without calculating it



Dynamic branch prediction

- A 2-bit counter for each branch
- Predict taken if the counter value >= 2
- If the prediction in taken states, fetch from target PC, otherwise, use PC+4
  - 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



PC = 0x400420

0x400420	0x8048324	11
0x400464	0x8048392	10
0x400578	0x804850a	00
0x41000C	0x8049624	01

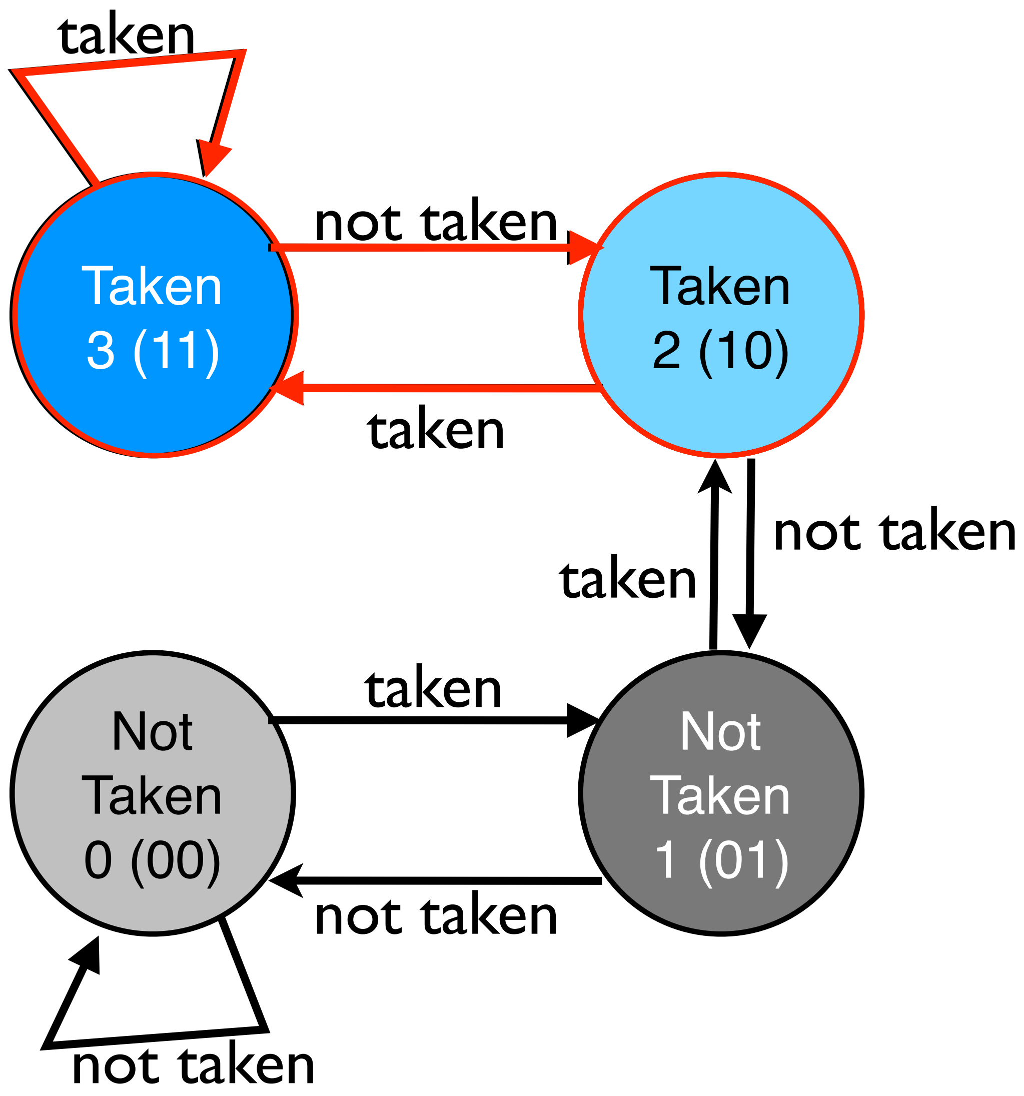
Taken!

Branch Target Buffer

Performance of 2-bit counter

- 2-bit state machine for each branch

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



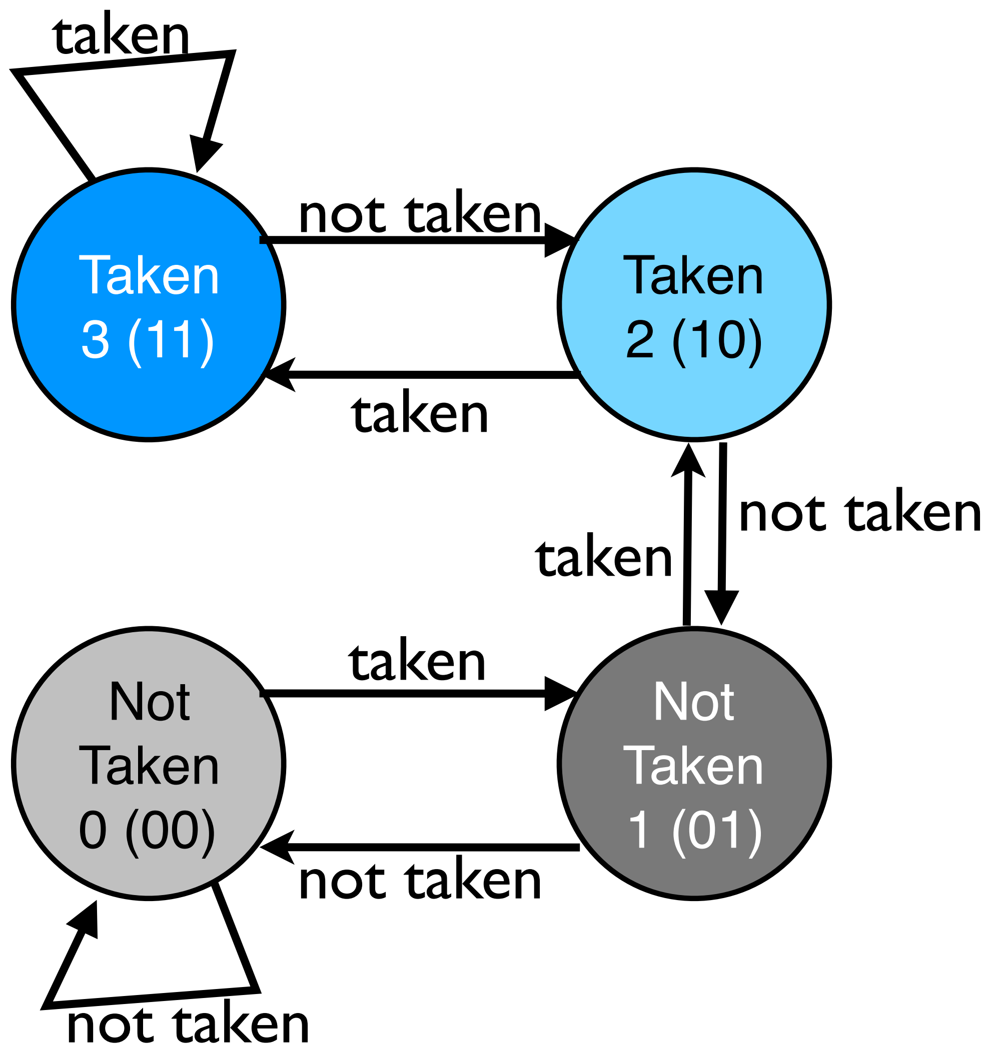
90% accuracy!

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 ✕

- Application: 80% ALU, 20% Branch, and branch resolved in EX stage, average CPI?
  - 1+20%*(1-90%)*2 = 1.04

Local 2-bit predictor

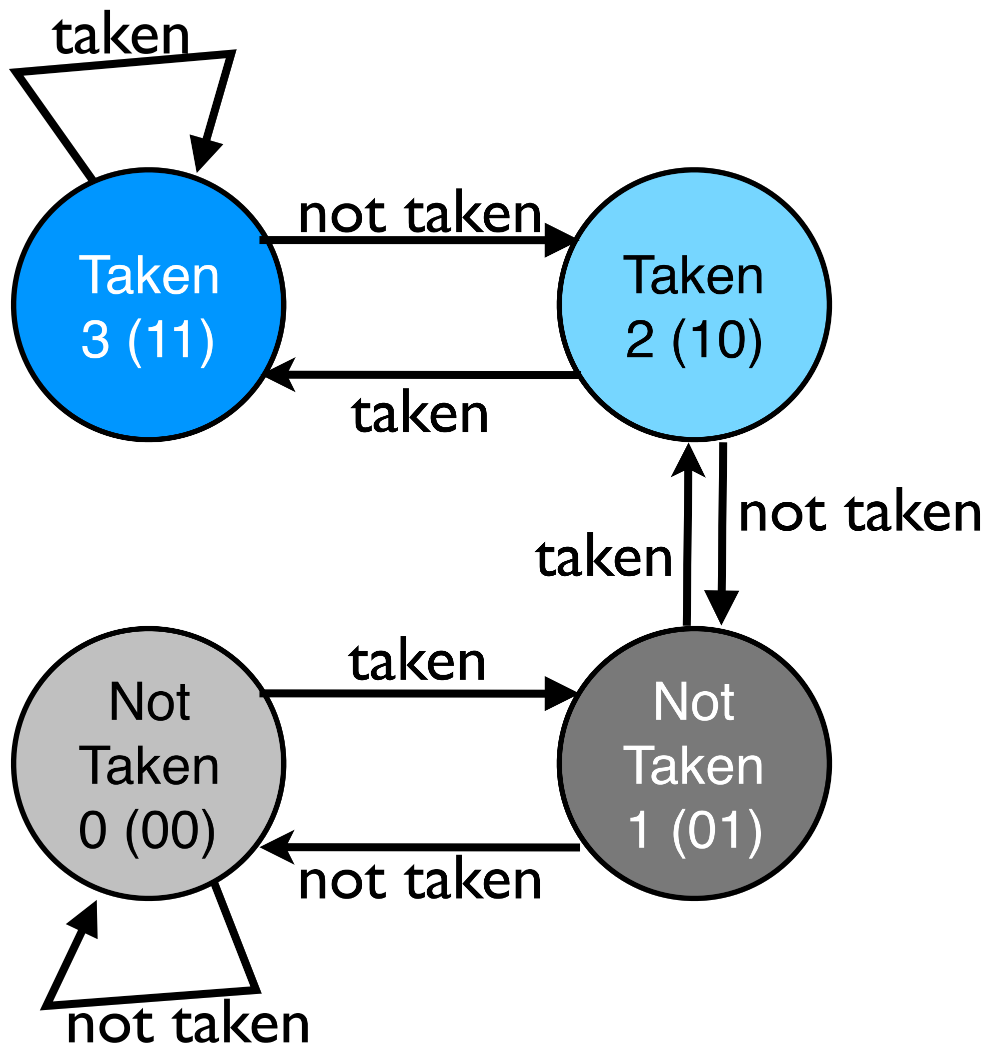
```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```



- What's the overall branch prediction (include both branches) accuracy for this nested for loop? (assume all states started with 00)
  A. ~25%
  B. ~33%
  C. ~50%
  D. ~67%
  E. ~75%

Local 2-bit 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
```

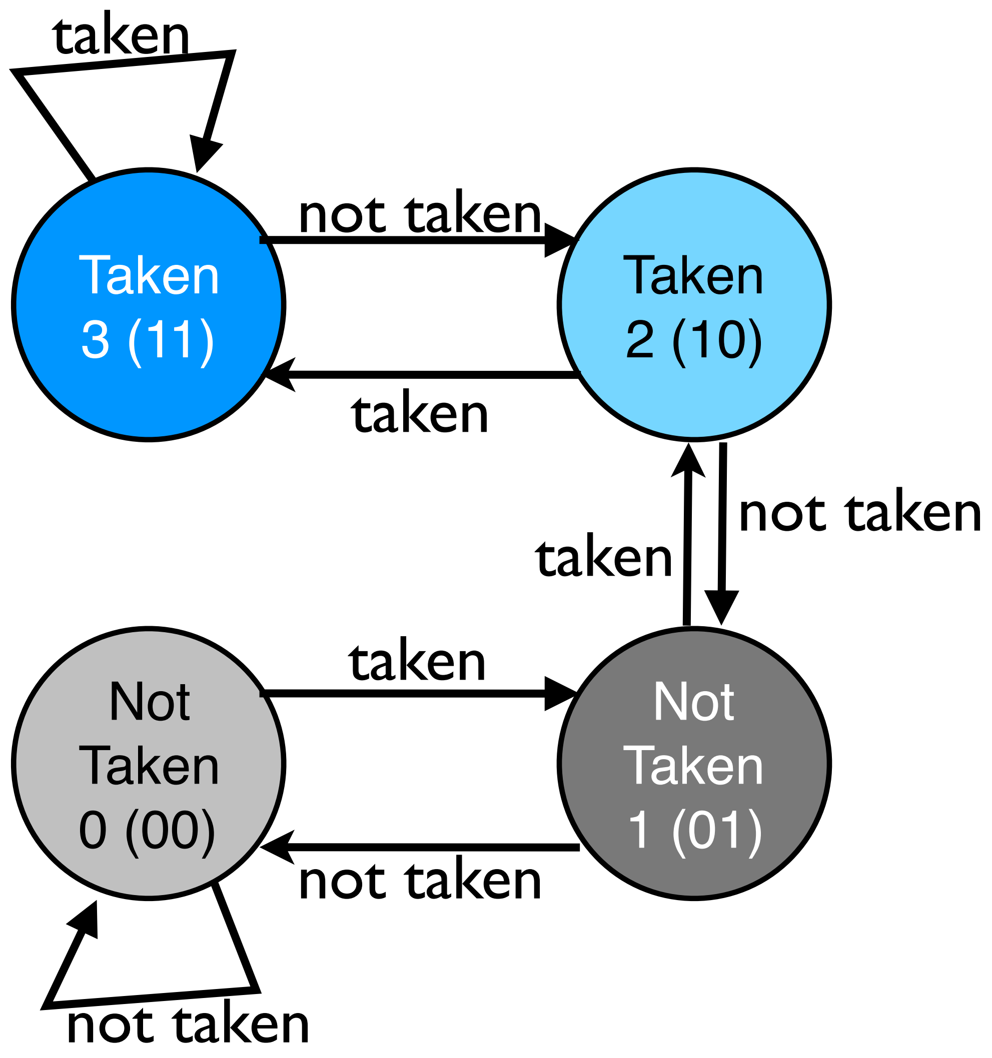


i	branch?	state	prediction	actual
0	X	00	NT	T
0	Y	00	NT	T
1	X	01	NT	NT
1	Y	01	NT	T
2	X	00	NT	T
2	Y	10	T	T
3	X	01	NT	NT
3	Y	11	NT	T
4	X	00	NT	T
4	Y	10	T	T
5	X	01	NT	NT
5	Y	11	NT	T
6	X	00	NT	T
6	Y	10	T	T

For branch Y, almost 100%,
For branch X, only 50%

Local 2-bit predictor

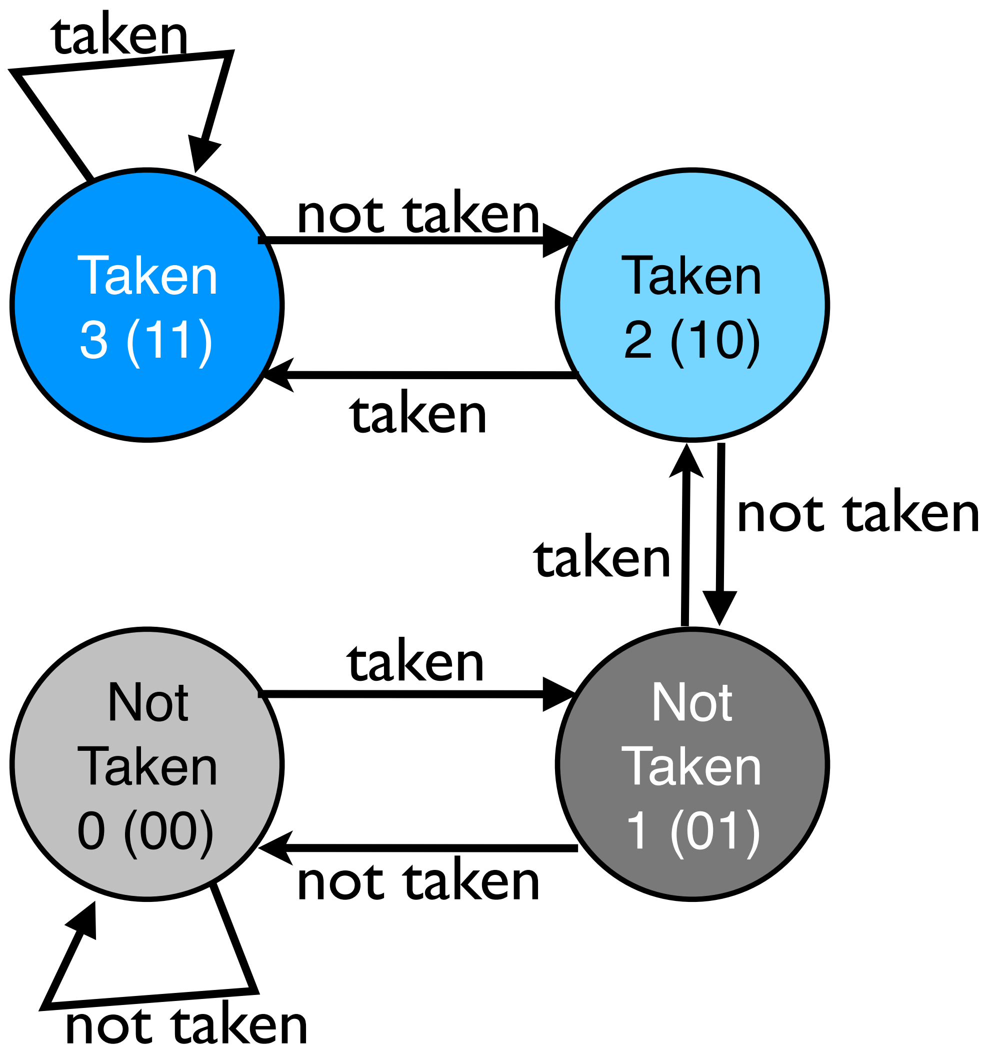
```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```



- What's the overall branch prediction (include both branches) accuracy for this nested for loop? (assume all states started with 00)
  - A. ~25%
  - B. ~33%
  - C. ~50%
  - D. ~67%
  - E. ~75%

Local 2-bit 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?	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 10	T	T

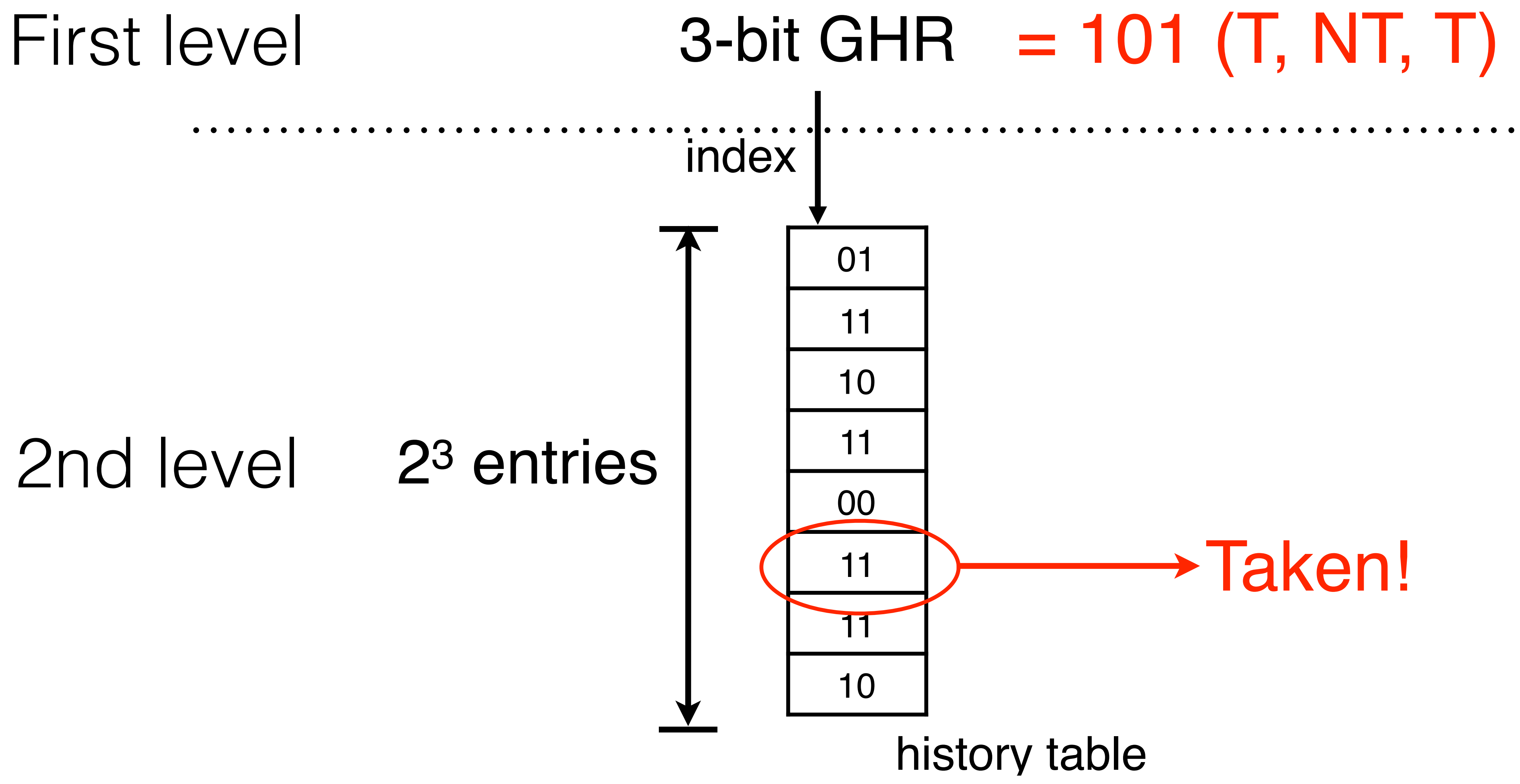
For branch Y, almost 100%,
For branch X, only 50%

Can we capture the pattern?

Branch prediction using global history

2-level global predictor

- Instead of using the PC to choose the predictor, use a bit vector (global history register, GHR) made up of the previous branch outcomes.
- Global predictor: predictor using results from all branches
- Local predictor: predictor tracking states/history for each branch
- Each entry in the history table has its own counter.



Pentium Pro uses this predictor

Performance of the 2-bit global 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

Nearly perfect after this

Branch prediction & your code

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 ++;
    }
}
```

Branch performance

- 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%

Demo: popcount

- How many 1s in binary representations
- Applications
  - Hamming weight
  - Encryption/decryption

```
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

- 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-predictions 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 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   $t2, $t1, 1
add   $t3, $t3, $t2
shr   $t1, $t1, 1
bne   $t1, $zero, 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   $t2, $t1, 1
add   $t3, $t3, $t2
shr   $t1, $t1, 1
and   $t2, $t1, 1
add   $t3, $t3, $t2
shr   $t1, $t1, 1
and   $t2, $t1, 1
add   $t3, $t3, $t2
shr   $t1, $t1, 1
and   $t2, $t1, 1
add   $t3, $t3, $t2
shr   $t1, $t1, 1
bne   $t1, $zero, LOOP
```

```
and   $t2, $t1, 1
shr   $t4, $t1, 1
shr   $t5, $t1, 2
shr   $t6, $t1, 3
shr   $t1, $t1, 4
and   $t7, $t4, 1
and   $t8, $t5, 1
and   $t9, $t6, 1
add   $t3, $t3, $t2
add   $t3, $t3, $t7
add   $t3, $t3, $t8
add   $t3, $t3, $t9
bne   $t1, $zero, LOOP
```

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

Only one branch for four iterations in A

Two versions of B

Before re-ordering

```
and   $t2, $t1, 1
add   $t3, $t3, $t2
shr   $t1, $t1, 1
and   $t2, $t1, 1
add   $t3, $t3, $t2
shr   $t1, $t1, 1
and   $t2, $t1, 1
add   $t3, $t3, $t2
shr   $t1, $t1, 1
and   $t2, $t1, 1
add   $t3, $t3, $t2
shr   $t1, $t1, 1
bne   $t1, $zero, LOOP
```

Lots of back-to-back data dependencies — likely to introduce data hazards

After re-ordering

```
and   $t2, $t1, 1
shr   $t4, $t1, 1
shr   $t5, $t1, 2
shr   $t6, $t1, 3
shr   $t1, $t1, 4
and   $t7, $t4, 1
and   $t8, $t5, 1
and   $t9, $t6, 1
add   $t3, $t3, $t2
add   $t3, $t3, $t7
add   $t3, $t3, $t8
add   $t3, $t3, $t9
bne   $t1, $zero, LOOP
```

This re-ordering is only possible after you “unrolled” your loop — this technique is called “**loop unrolling**”

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-predictions 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-predictions 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

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

Announcement

- Homework #2 due next Monday
- Quiz due next Monday
- Midterm
  - Next Wednesday — only from 8am-9:20a
  - Will have review on next Monday