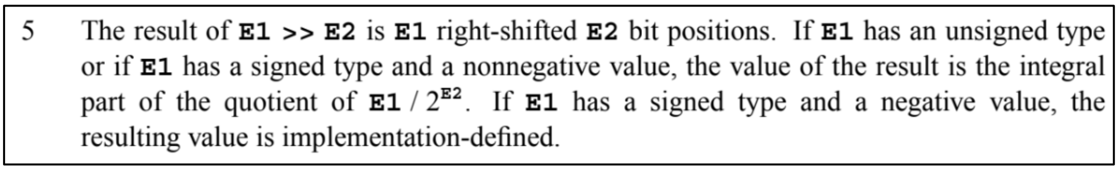
Name: 陳柏瑜
Student ID: 41047054S
Department: Computer Science and Information Engineering 114
Computer Programming II HW0206

This is the **bitwise shift operation** issue. First of all, we see how the SPEC says about the right shift operation.

In C99 6.5.7,



5 The result of **E1 >> E2** is **E1** right-shifted **E2** bit positions. If **E1** has an unsigned type or if **E1** has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of **E1** / $2^{\mathbf{E2}}$. If **E1** has a signed type and a negative value, the resulting value is implementation-defined.

In brief, we say that it uses sign-extension when doing right shift.
If the bit is a signed positive integer, which means the sign bit is 0. When doing right shifting, the zero is shifted and the previous value is still zero. Below gives a short example,

```
00001010
00000101
00000010
```

If the bit is a signed negative integer, which means the sign bit is 1. When doing right shifting, the one is shifted, and the previous value keeps all one. Below gives another short example,

```
10001010
11000101
11100010
```

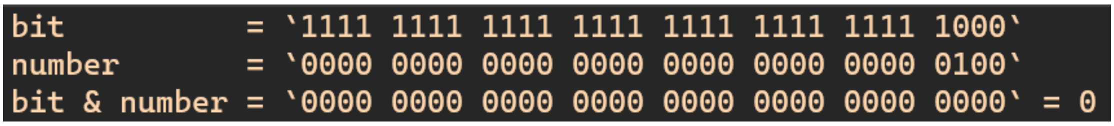
Then, we back to our hw0206.c example code. We know that the bit is a signed integer. In line 11, it makes bit becomes 1000 0000 0000 0000 0000 0000 0000 0000. As entering for loop, it right shifts one each time, and we can get the following steps.

```
1000 0000 0000 0000 0000 0000 0000 0000
1100 0000 0000 0000 0000 0000 0000 0000
1110 0000 0000 0000 0000 0000 0000 0000
1111 0000 0000 0000 0000 0000 0000 0000
…
```

Now, we see that **“when the first-time bit & number is one, the rest of the value is one”**.

Here is a short example for number = 4.
Assume the for loop has run for 27 times,
bit = 1111 1111 1111 1111 1111 1111 1111 0000.

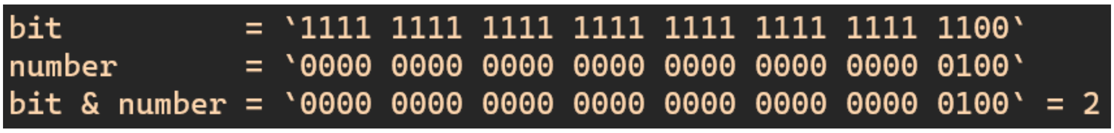
The 28th times,



```
bit          = `1111 1111 1111 1111 1111 1111 1111 1000`
number       = `0000 0000 0000 0000 0000 0000 0000 0100`
bit & number = `0000 0000 0000 0000 0000 0000 0000 0000` = 0
```

➔ It prints zero.

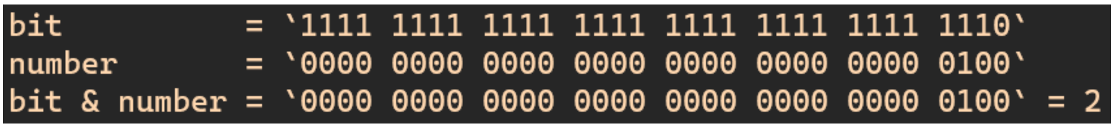
The 29th times,



```
bit          = `1111 1111 1111 1111 1111 1111 1111 1100`
number       = `0000 0000 0000 0000 0000 0000 0000 0100`
bit & number = `0000 0000 0000 0000 0000 0000 0000 0100` = 2
```

➔ It prints one.

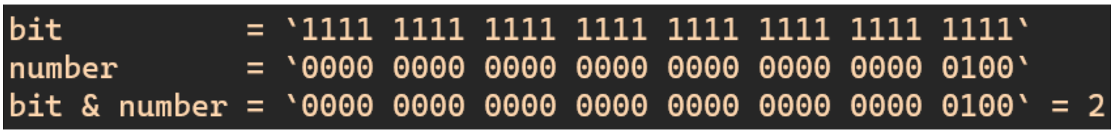
The 30th times,



```
bit          = `1111 1111 1111 1111 1111 1111 1111 1110`
number       = `0000 0000 0000 0000 0000 0000 0000 0100`
bit & number = `0000 0000 0000 0000 0000 0000 0000 0100` = 2
```

➔ It prints one.

The 31st times,



```
bit          = `1111 1111 1111 1111 1111 1111 1111 1111`
number       = `0000 0000 0000 0000 0000 0000 0000 0100`
bit & number = `0000 0000 0000 0000 0000 0000 0000 0100` = 2
```

➔ It prints one.

That is the reason we get 0000 0000 0000 0000 0000 0000 0000 0111 instead of 0000 0000 0000 0000 0000 0000 0000 0100 when number = 4.

To solve this problem, we simply exchange the type of bit from int32_t into uint32_t.