Bitwise operators

& AND

| OR

~ COMPLEMENT

^ XOR

These operators do not modify their operands

```
int x = 2;   // 10 binary
int y = 1;   // 1 binary
x | y;   // doesn't modify x or y
x + y;   // also doesn't modify x or y
x = x | y;   // now x is 11 binary = 3 decimal
```

Shift operators

- compute a value resulting from shifting a number left or right.

>> - right shift

<< - left shift

`x << 3` - shifts the value of x left by three bits, filling in zeros in the rightmost 3 bits.

```
int x = 0xf;
int y = x << 5; // y is 00..0111100000
```

F hex = 1111 binary

Since each 1-bit shift left doubles the value of the number, shifting left by 5 multiplies the number by $2^5 = 32$.

- shifting n bits left multiplies a number by 2^n.

What happens if the number is negative, such that it looks like:

10...01

↑ negative

and we shift left by 1 bit:

0...010

↑ positive!

Overflow situation!
- number is too big to be contained in the bits we have.

Right shift:

$x >> 4$

Computes the result of shifting the value of x to the right by 4.

- if x is unsigned (positive or zero) then zeros are written into the leftmost 4 bits.

- if x is signed, then the right shift has to preserve the sign of x.
  - if x is positive or zero, fill in the leftmost bits with zero.
  - if x is negative, fill in the leftmost bits with 1.

This causes a right shift by n bits to compute $x / 2^n$.

```
unsigned int x = 0xFA;
                    // 0...011111010
```

F A

```
unsigned int y = x >> 3;
                    // 0...000111111
```

```
int w = -2;          // 11...110
int z = w >> 1;      // 11...11
```

shifted off

new bit = 1

We use bitwise and shift operators to read and set individual bits of a variable.

Example: how do we determine if the rightmost bit of a number is 0 or 1?

- use the & operator with a "mask" that contains all zeros and a 1 in the rightmost bit position.

```
unsigned int n = 37;
unsigned int result = n & 0x1;
```

mask = 00..01

n		00..100101
0x1	&	00 000001
		0000...1

So, result will be 1 since n had a 1 in the rightmost position.

```
unsigned int x = 36;
unsigned int result = x & 0x1
```

```
  0...000100100
& 0...000000001
---------------
  000 ... 00
```

result is 0, telling us that x had a zero in the rightmost position.

What if I wanted to know if the n^{th} bit of x is 1 or 0?

Solution 1: Perform an & on x and a mask with a 1 in the n^{th} position and zero everywhere else.

```
unsigned int result = x & (0x1 << n)
```

(0x1 << n): mask w/ 1 in the n^{th} position

n: bit n

```
x      000101010110
mask   000000100000
result 000000000000
```

Result will be 0 if the n^{th} bit of x is 0.

What if the n^{th} bit of x is 1?

Result is not 0, but it's not 1 either.

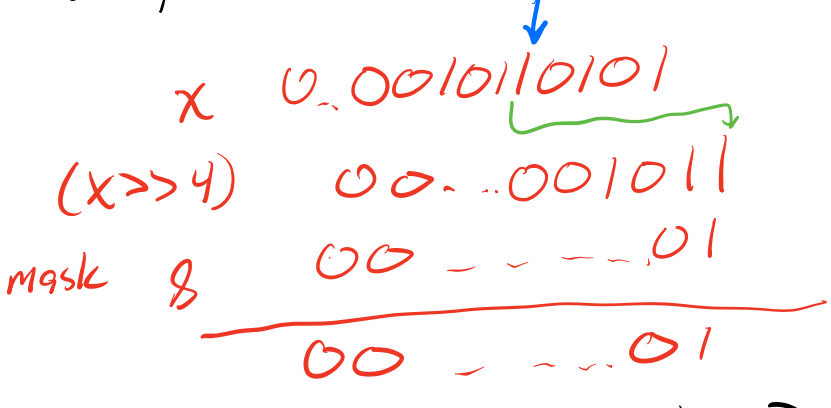
- in fact, it's 2^n (which is not important)

```
if (x & (0x1 << n))
    printf("yes");
else
    printf("no");
```

Solution 2:

Shift x right by n bits and & with 0x1;

```
unsigned int result = (x >> n) & 0x1;
```



Here result will either be 0 or 1;

How do we determine the value of the lowest 4 bits of x.

```
x      &   00..1011011101
mask       0000000001111      ← F hex
       ------------------
           0000..01101
           all zeros  ← lowest 4 bits of x.
```

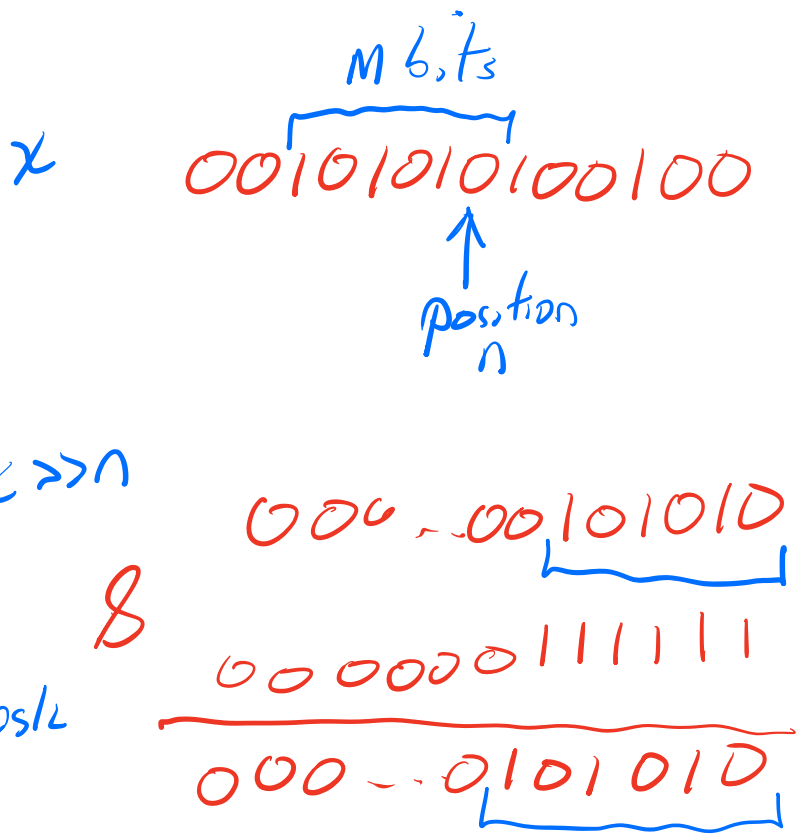
```
unsigned int result = x & 0xf;
```

instead

```
#define MASK 0xf
unsigned int result = x & MASK;
```

In general, if I want to know the value of the M bits starting at position n:

- shift x to the right by n bits
- & it with a mask where lower M bits are 1.



A mask with M 1's in the lowest position would be

$$(1 << M) - 1$$

This value is $2^M - 1$, which is M 1's in binary (111...1) (M)

```
unsigned int result = (x >> n) & ((1<<m)-1);
```

How do we set the n^{th} bit of x to 1?

- OR, "|" the number w/ a mask that has a 1 in the n^{th} position and a zero everywhere else.