# C Programming

## Lecture 10: Bitwise Operations

$$\begin{array}{r} 00111100 \\ \odot\ 00001101 \\ \hline 11001110 \end{array} = 206_{(10)}$$

Lecturer: *Dr*. Wan-Lei Zhao

*Autumn Semester* 2022

Email: wlzhao@xmu.edu.cn, 

# Outline

# What are bit operations?

- Data onside computers are kept in binary form, such as 10101111
- One binary code is a data item, it could be an integer, a float number, or a string
- In some scenarios, we need to operate them bit-wisely

- Given a binary code 10101111
- How could we extract out its lower 4 bits

# The bitwise operators

- There are 6 bit operators
- bit and &
- bit or |
- bit xor ^
- bit not ~
- left shift $\ll$
- right shift $\gg$

# Truth tables for &, | and ^

| c1 | c2 | c1 & c2 |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

| c1 | c2 | c1 \| c2 |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

| c1 | c2 | c1 ^ c2 |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

- Notice that it is applied on one bit ONLY
- If there are multiple bits, the operator is applied on each bit
- The result of one bit operation has NO impact on the other bit

# AND & and OR |

- Given two variables a = 60 and b = 13 of unsigned char
- See what are the result for a & b
- See what are the result for a | b

$$\begin{array}{r} 00111100 \\ \& \;\; 00001101 \\ \hline 00001100 \end{array} == 12_{(10)} \qquad \begin{array}{r} 00111100 \\ | \;\; 00001101 \\ \hline 00111101 \end{array} == 61_{(10)}$$

```c
#include <stdio.h>
int main(){
   unsigned char a = 60, b = 13;
   unsigned char c = a & b;
   unsigned char d = a | b;
   printf("c = %d, d = %d\n", c, d);
   return 0;
}
```

# OR | and xOR^

- Given two variables a = 60 and b = 13 of unsigned char
- See what are the result for a | b
- See what are the result for a^b

$$\begin{array}{r} 00111100 \\ |\ \ 00001101 \\ \hline 00111101 \end{array} == 61_{(10)} \qquad \begin{array}{r} 00111100 \\ \wedge\ \ 00001101 \\ \hline 00110001 \end{array} == 49_{(10)}$$

```
1 #include <stdio.h>
2 int main(){
3    unsigned char a = 60, b = 13;
4    unsigned char c = a | b;
5    unsigned char d = a ^ b;
6    printf("c = %d, d = %d\n", c, d);
7    return 0;
8 }
```

# NOT ~ (1)

| c1 | ~c1 |
|---|---|
| 1 | 0 |
| 0 | 1 |

- Flip a bit
- $1 \rightarrow 0$, $0 \rightarrow 1$
- The result of one bit operation has NO impact on the other bit

# NOT ~ (2)

- Given one variable a = 60 of unsigned char
- See what are the result for ~a

$$\sim 00111100$$

$$11000011 == 195_{(10)}$$

```
#include <stdio.h>
int main(){
   unsigned char a = 60;
   unsigned char c = ~a;
   unsigned char d = !a;
   printf("c = %d, d = %d\n", c, d);
   return 0;
}
```

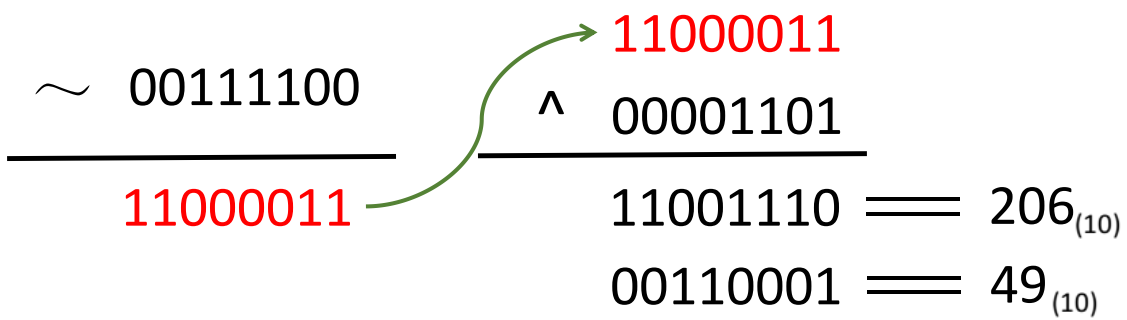
# Example-1: implement $\odot$ operation (1)

| c1 | c2 | c1 $\odot$ c2 |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

- In some cases, we need 1 for bits of the same, while 0 for bit of difference
- There is NO such operator in C
- Can we realize it with provided operators?

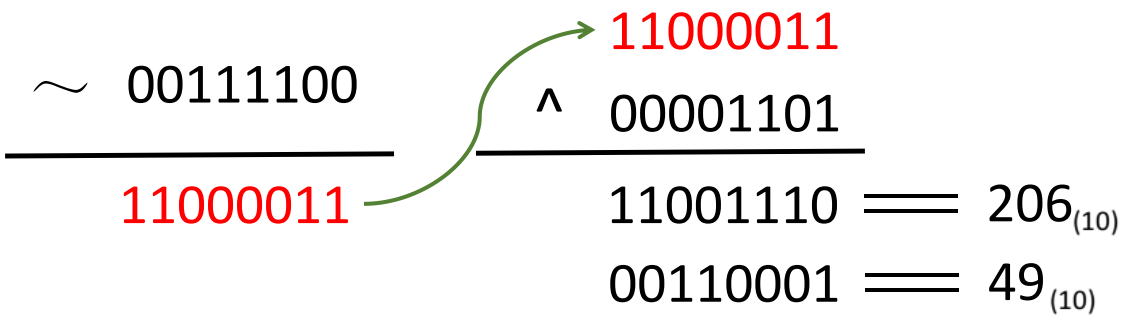
Think about it in five minutes...

# Example-1: implement $\odot$ operation (2)




- We achieve this in two steps
  1. Flip one of the numbers
  2. Apply XOR between the flipped number and another number

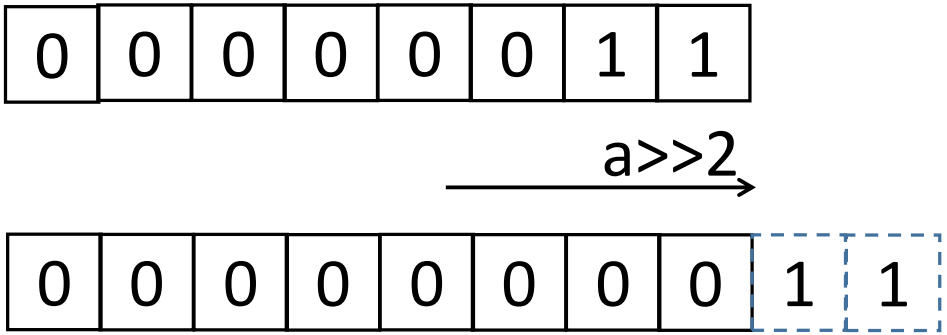
# Example-1: implement ⊙ operation (3)




```c
#include <stdio.h>
int main(){
   unsigned char a = 60, b = 13;
   unsigned char c = ~a;
   unsigned char d = c ^ b;
   printf("c = %d, d = %d\n", c, d);
   return 0;
}
```

- You will get the same result if you flip b

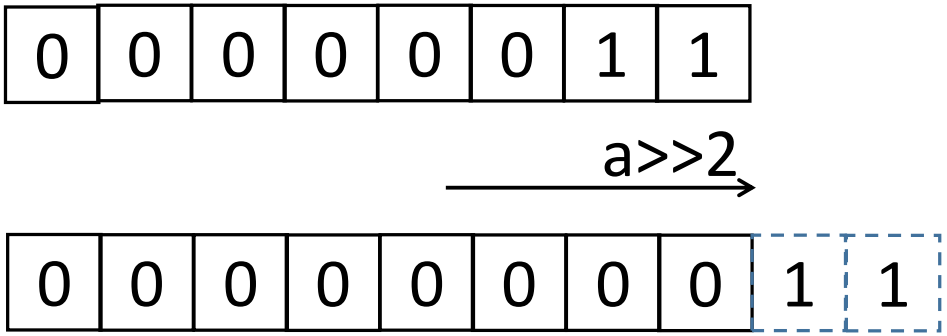
# Left shift $val \ll numb$




- Shift the binary code towards the left in numb bits
- Append the lower bits with 0s
- For example, a = 3; a≪2
- The result is 12

```c
#include <stdio.h>
int main(){
   unsigned char a = 3, b = 0;
   b = a << 2;
   printf("a = %d, b = %d\n", a, b);
   return 0;
}
```

# Right shift $val \gg numb$




- Shift the binary code towards the right in numb bits
- Append the higher bits with 0s
- For example, a = 3; a≫2
- The result is 0

```c
#include <stdio.h>
int main(){
  unsigned char a = 3, b = 10, c = 0;
  b = a >> 2;
  c = a >> 1;
  printf("a = %d, b = %d, c = %d\n", a, b,
    c);
  return 0;
}
```

# Outline

# Example-2: extract out specified bits from a number (1)

- Given a binary code 10101110
- How could we extract out its higher 4 bits
- Given int a=0xAE

Think about it in five minutes....

# Example-2: extract out specified bits from a number (2)

- How could we extract out its higher 4 bits
- Given int a=0xAE
- We introduce a template number b = 0xF0

Try this operation: a & b

# Example-2: extract out specified bits from a number (3)

- How could we extract out its higher 4 bits
- Given int a=0xAE
- We introduce a template number b = 0xF0

```
1 #include <stdio.h>
2 int main(){
3    unsigned char a = 0xAE, b = 0xF0, c = 0;
4    c = a & b;
5    c = c>>4;
6    printf("a = %x, c = %x\n", a, c);
7    return 0;
8 }
```

# Example-3: check whether a number is odd (1)

- Given a number **n**, we want to know whether it is odd or even
- We check **n%2** != 1
- Now we have another option
- We only need to check the last bit of an integer number
  1. If it is 1, it is odd
  2. Otherwise, it is even

# Example-3: check whether a number is odd (2)

- Given a number **n**, we want to know whether it is odd or even
- We check **n%2** != 1
- Now we have another option

```
1  #include <stdio.h>
2  int main(){
3     int a = 7;
4     scanf("%d", &a);
5     if( a & 1)
6           printf("It is odd\n");
7      else
8           printf("It is even\n");
9     return 0;
10 }
```

# Example-4: count how many bits is 1 (1)

- Given an integer number **n**, we want to know how many bits is '1'
- We shift the number to right one bit at once
- We check whether the last bit of the shifted number is '1'
  1. If it is 1, counted in
  2. Otherwise, do nothing

# Example-4: count how many bits is 1 (1)

- We shift the number to right one bit at once
- We check whether the last bit of the shifted number is '1'
  1. If it is 1, counted in
  2. Otherwise, do nothing

```c
#include <stdio.h>
int main(){
   int a = 11, count = 0, b = 0;
   scanf("%d", &a);
   while(a>0){
       b = a & 1;
       if(b == 1){
            count++;
       }
       a = a >> 1;
   }
   printf("count = %d\n", count);
   return 0;
}
```

# Example-5: set the k-th bit to 1 (1)

- Given a number **n**=01010000
- We want to set the 4-th bit to 1
  1. We left shift 1 3 times
  2. Perform OR between n and the shifted number

# Example-5: set the k-th bit to 1 (2)

- Given a number **n**=01010000
- We want to set the 4-th bit to 1
  1. We left shift 1 3 times
  2. Perform OR between n and the shifted number

```
1 #include <stdio.h>
2 int main(){
3    int a = 0x50, b = 0;
4    b = 1 << 3;
5    a = a | b;
6    printf("count = %x\n", a);
7    return 0;
8 }
```