Lab 11: Bitwise Operators

CS F111

Authors: CS F111 FDTAs
(Aditya Tailor,
Mitra Shah,
Ronit Kunkolienker)

Introduction

- Bitwise operators in C are fundamental tools for manipulating **individual bits** within variables.
- These operators allow you to perform operations at the **binary level**, which can be useful for tasks such as **setting** or **clearing specific bits**, **checking if a particular bit is set**, or performing **bitwise logic operations** like AND, OR, XOR, and NOT.
- In this tutorial, we will be covering 6 basic bitwise operations:

1. bitwise AND (&) operator
2. bitwise OR (|) operator
3. bitwise XOR (^) operator
4. bitwise NOT (~) operator
5. left shift (<<) operator
6. right shift (>>) operator

Why do we use BITWISE operators?

BITWISE operators offer quite a few advantages such as:

- **Efficient Memory Usage**: Bitwise operators allow us to manipulate individual bits within a variable. This means we can store more information in a smaller amount of space. For example, instead of using an entire integer to store a simple yes/no value, we could use a single bit. This can be particularly useful in systems where memory is at a premium.

- **Faster Execution**: Bitwise operations are typically faster than arithmetic operations. This is because they are simpler for the CPU to perform - they don’t require any complex calculations, just simple setting or clearing of bits. This can lead to significant performance improvements in time-critical code.

continued...

- **Bit Manipulation**: Bitwise operators allow us to manipulate individual bits within a variable. This can be useful in a variety of applications. For example, in graphics programming, individual bits in a color value might represent the red, green, and blue components of the color. With bitwise operators, we can manipulate these components individually. It is also used in cryptography

- **Bit Masking**: Bitwise operators are commonly used for bit masking, which involves setting or retrieving only certain bits of a variable. This can be useful for extracting specific pieces of information from a variable. For example, if we have a byte where each bit represents a different setting in a configuration, we can use a bit mask to extract the value of a specific setting.

Quick Revision!

Before we start with each operator, let’s quickly revise some basics of binary representation of data that we learnt earlier…

- **Set & Unset bit**

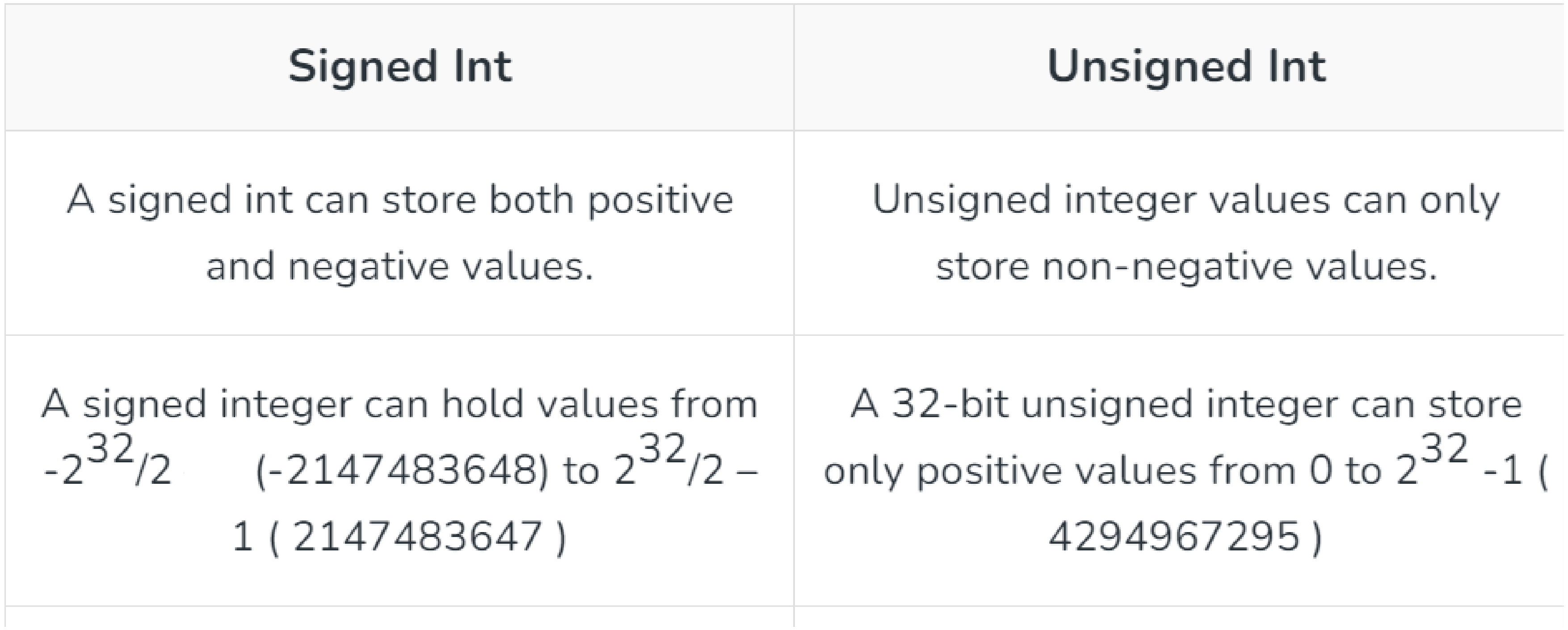
  We call a bit set if it is 1, and unset if its 0. Eg: 10110 - has 3 set bits and 2 unset bits.

- **Positioning of bits**

  We usually start numbering the bits from right to left, assigning the rightmost bit as 0th index and so on. This is called the **Little Endian** format and most systems nowadays use this. The **rightmost bit** is called **LSB(** Least Significant Bit) & **leftmost bit** is called **MSB**(Most Significant Bit).

unsigned int data type

- unsigned int is used when we want to store only non-negative integers.



Signed Int	Unsigned Int
A signed int can store both positive and negative values.	Unsigned integer values can only store non-negative values.
A signed integer can hold values from $-2^{32}/2$ (-2147483648) to $2^{32}/2 - 1$ (2147483647)	A 32-bit unsigned integer can store only positive values from 0 to 2^{32} -1 (4294967295)

continued...

- In C, int data type is signed by default
- MSB in signed int represents sign:- 0 for positive and 1 for negative numbers.
- In unsigned int , all bits represent only magnitude of the number;

```
unsigned int x = 55;
```

The binary representation of 55 is: 110111.
So, x is stored as:

00000000 00000000 00000000 00110111

Most Significant Bit b_{31}

Least Significant Bit b_0

BITWISE AND (&) OPERATOR

- The BITWISE ‘&’ operator performs the **logical ‘AND’** operation between the corresponding bits of 2 operands, and returns the answer.
- It returns 1, if both the bits are 1, else it returns 0 in all other cases.
- Eg: Let’s say we have 2 variables a and b, with the following values-

  a=6, b=4

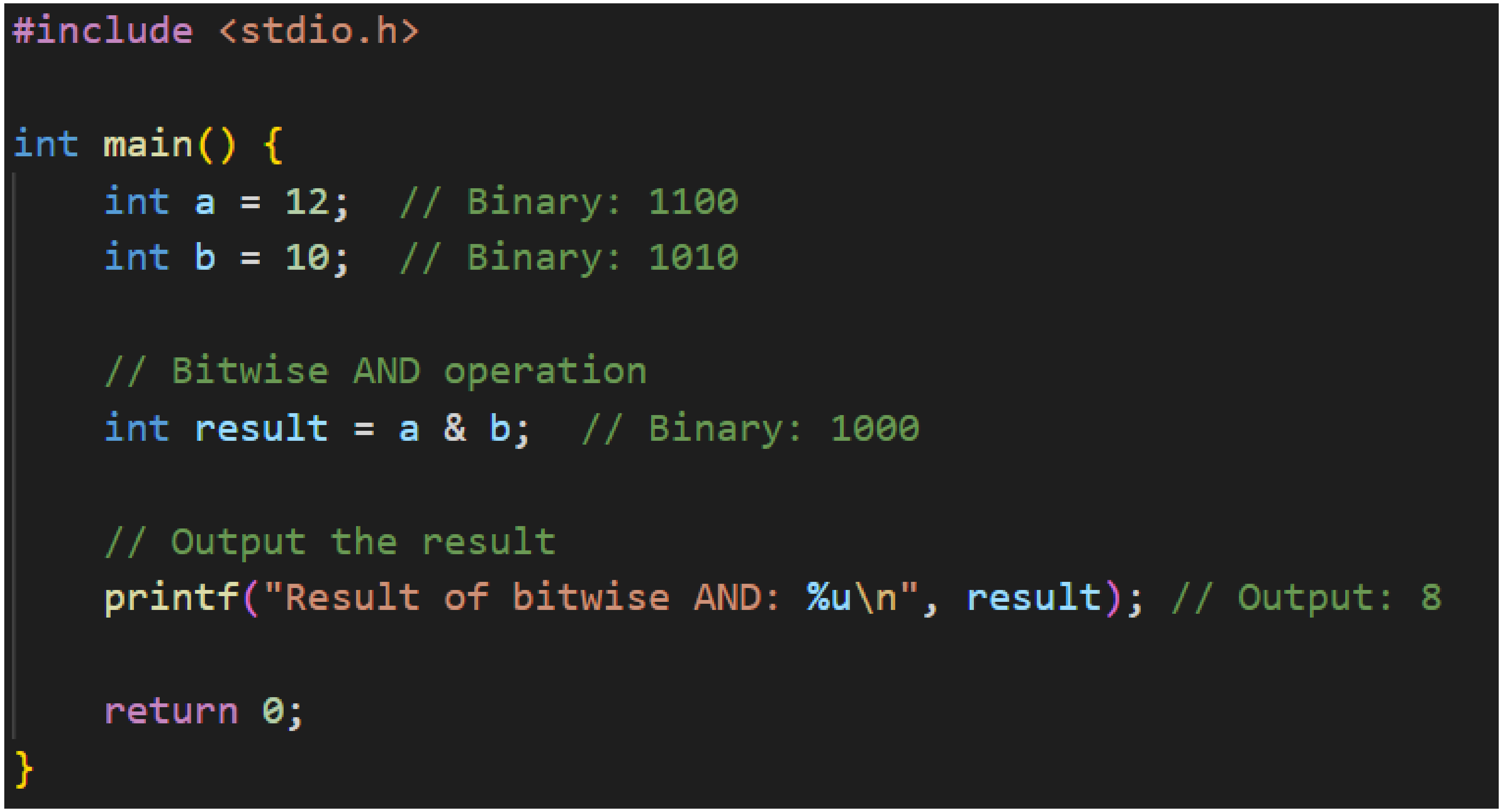
  Their binary representation is as follows:

  a=0110, b=0100

- Now if we perform bitwise AND on the 2 variables, we will get the following result:

```
a    0110
b    0100
---------
a&b  0100
```

And we know that **0100** in binary is equivalent to **4** in decimal. Hence **a & b == 4**

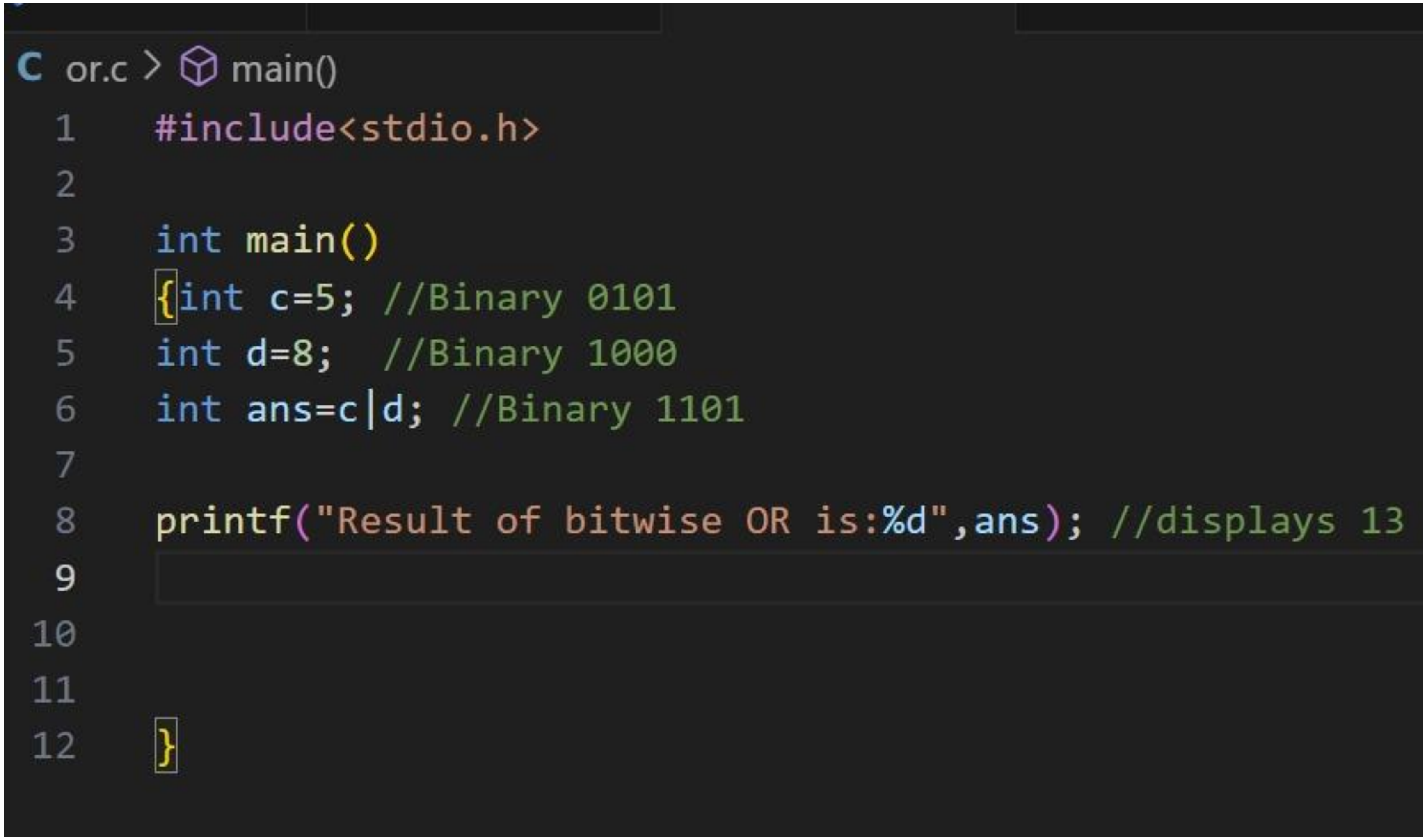

```c
#include <stdio.h>

int main() {
    int a = 12;  // Binary: 1100
    int b = 10;  // Binary: 1010

    // Bitwise AND operation
    int result = a & b;  // Binary: 1000

    // Output the result
    printf("Result of bitwise AND: %u\n", result); // Output: 8

    return 0;
}
```

BITWISE OR(‘|’) OPERATOR

- The Bitwise | operator performs the logical OR operation between the corresponding bits of 2 operands.
- It returns 1 if at least one of the bits is 1.
- It returns 0 only if both bits are 0.
- For example, let’s consider 2 variables c and d with values 6 and 4 respectively.
    - The binary representation of c (6) is 0110.
    - The binary representation of d (4) is 0100.

```
c      0110
d      0100
-----------
c|d    0110
```

- Hence c | d = 0110
- The binary number 0110 is equivalent to 6 in decimal.
- Hence, c | d equals 6.

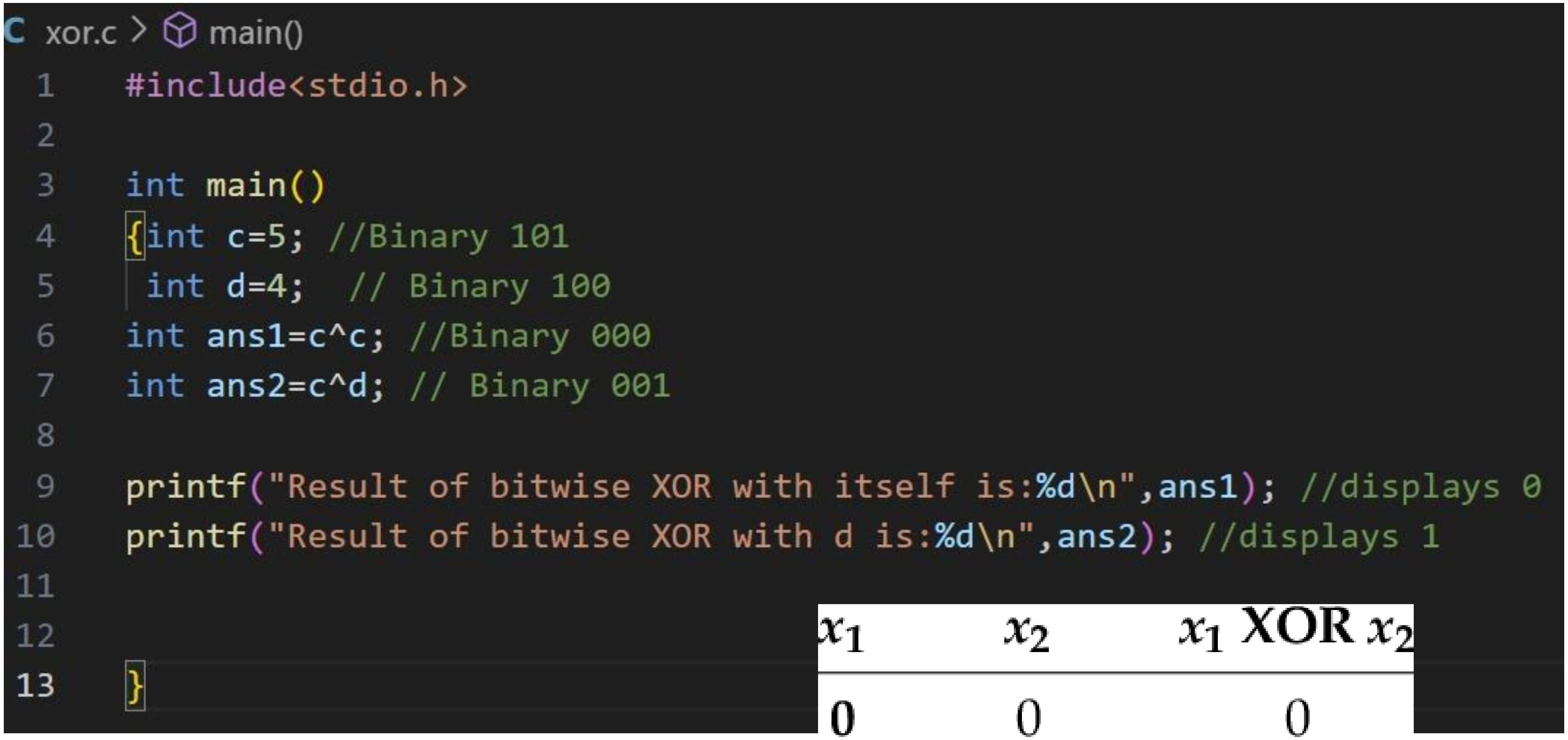


```c
#include<stdio.h>

int main()
{int c=5; //Binary 0101
int d=8;  //Binary 1000
int ans=c|d; //Binary 1101

printf("Result of bitwise OR is:%d",ans); //displays 13



}
```

BITWISE XOR('^') OPERATOR

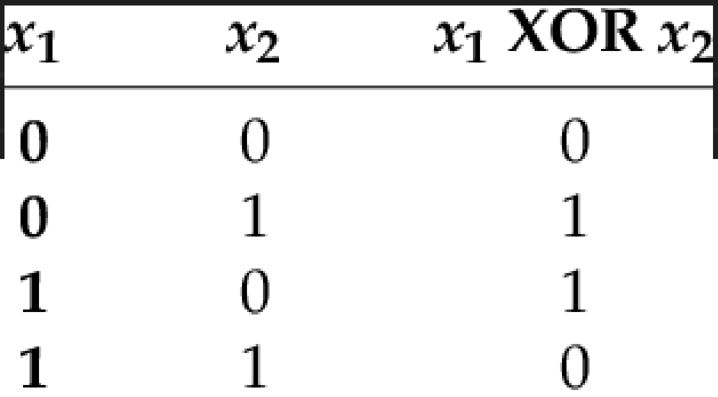
- The Bitwise ^ operator performs the logical XOR (exclusive OR) operation between the corresponding bits of 2 operands.
- It returns 1 if the bits are different (one is 1 and the other is 0).
- It returns 0 if the bits are the same (both 0 or both 1).
- For example, let's consider 2 variables g and h with values 6 and 4 respectively.
    - The binary representation of g (6) is 0110.
    - The binary representation of h (4) is 0100.
- If we perform the bitwise XOR operation on these 2 variables, we get:
    - g ^ h = 0010
- The binary number 0010 is equivalent to 2 in decimal.
- Hence, g ^ h equals 2.

```
g    0110
h    0100
---------
g^h  0010
```



xor.c > main()

```c
#include<stdio.h>

int main()
{int c=5; //Binary 101
 int d=4;  // Binary 100
int ans1=c^c; //Binary 000
int ans2=c^d; // Binary 001

printf("Result of bitwise XOR with itself is:%d\n",ans1); //displays 0
printf("Result of bitwise XOR with d is:%d\n",ans2); //displays 1



}
```



x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

the bitwise shift operators

`a<<n`	`a>>n`
(Bitwise left-shift)	(Bitwise right-shift)
Returns: `a` with each bit shifted **left** by `n` bits	Returns: `a` with each bit shifted **right** by `n` bits
The rightmost n bits are now clear (0)	The leftmost n bits are now clear

⚠ n must be a **positive** integer, otherwise the result becomes unpredictable

visualising bitshifts

Think of the **left**-shift `a<<n` as `n` 0s being "pushed" **from the right** into the binary representation of the number `a`.

The **right**-shift `a<<n` is like `n` 0s being "pushed" **from the left** into the binary representation of the number `a`.

Let's try evaluating:

`(unsigned int)24<<3`

If 24 is an unsigned integer (32-bit), then in binary it is:

0000 0000 0000 0000 0000 0000 0001 1000

b31 b30 b29 b28 b27 b26 ....b4 b3 b2 b1 b0

24<<3: Step 0

0

0

0

0000 0000 0000 0000 0000 0000 0001

1000

b31 b30 b29 b28 b27 b26 ....b4   b3 b2

b1 b0

Decimal Tracker

$0 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 + 0 \times 2^5 + 0 \times 2^6 + 0 \times 2^7 + \ldots + 0 \times 2^{31}$

$= 8 + 16$

= 24

24<<3: Step 1

0

0

0000 0000 0000 0000 0000 0000 0011

0000

b31 b30 b29 b28 b27 b26 ....b4 b3 b2

b1 b0

Decimal Tracker

$0 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 + 0 \times 2^6 + 0 \times 2^7 + \ldots + 0 \times 2^{31}$

$= 16 + 32$

$= 48$

24<<3: Step 2

0

0000 0000 0000 0000 0000 0000 011

00000

b31 b30 b29 b28 b27 b26 ....b4 b3 b2

b1 b0

Decimal Tracker

$0 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 + 0 \times 2^4 + 1 \times 2^5 + 1 \times 2^6 + 0 \times 2^7 + \ldots + 0 \times 2^{31}$

$= 32 + 64$

$= 96$

24<<3: Step 3

0000 0000 0000 0000 0000 0000 11
000000 ✓

b31 b30 b29 b28 b27 b26 ....b4   b3 b2
b1 b0

Decimal Tracker

$0 \times 2^{0} + 0 \times 2^{1} + 0 \times 2^{2} + 0 \times 2^{3} + 0 \times 2^{4} + 0 \times 2^{5} + 1 \times 2^{6} + 1 \times 2^{7} + \ldots + 0 \times 2^{31}$

$= 64 + 128$

$= 192$ ✓

shift by n bits ≣ shift by 1 bit n times

Notice that to find 24<<3, we have shifted 24 left 3 times, 1 bit at a time.

That is to say, we could have obtained the same result using:
(((24<<1)<<1)<<1) or
((24<<1)<<2) or
((24<<2)<<1)

$$\boxed{a<<(n_1+n_2) \text{ is } (a<<n_1)<<n_2}$$

left-bitshift multiplies by 2

Observe how the (decimal) value of the expression changes with every left bitshift.

24<<1 == 48

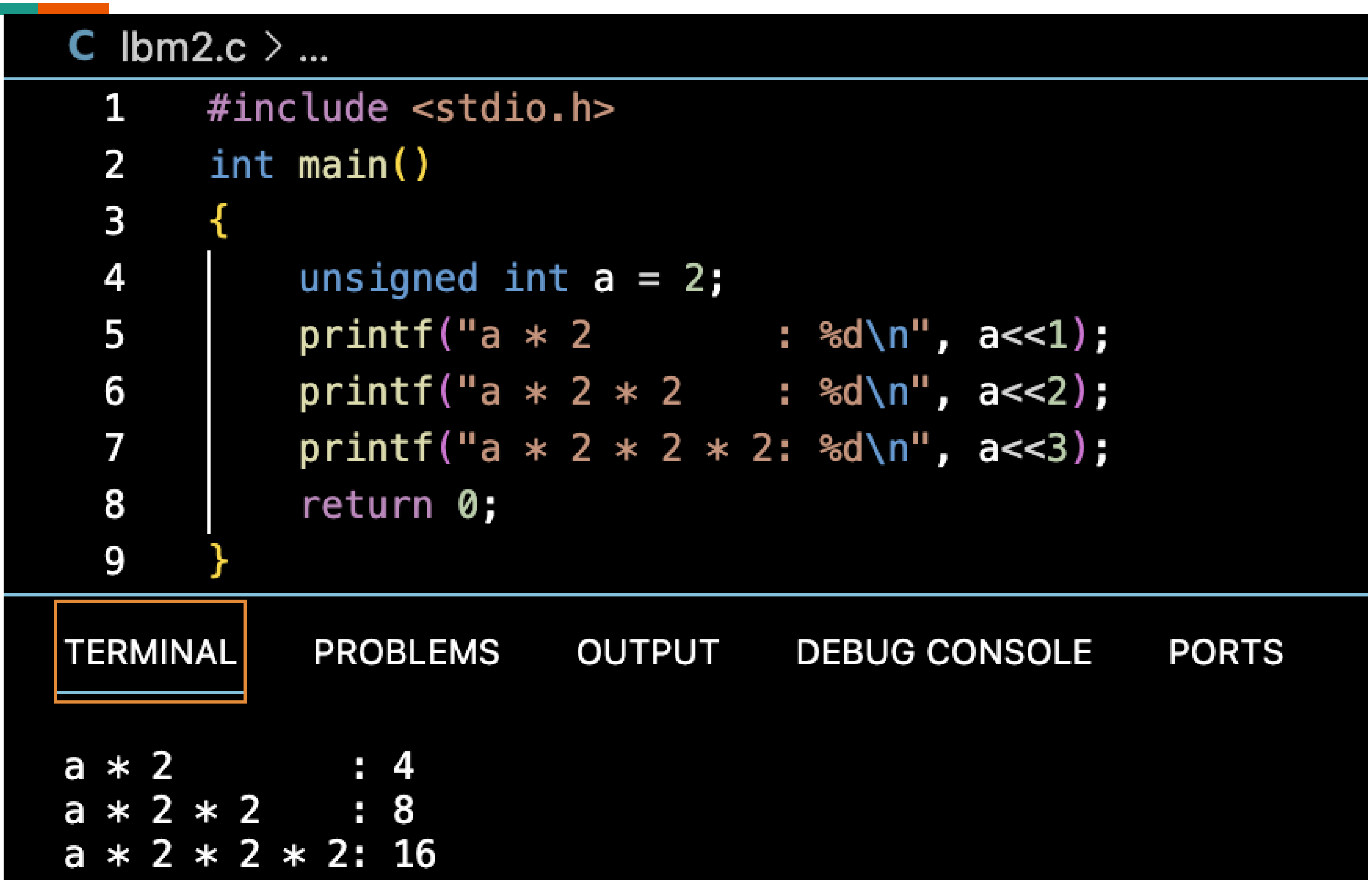
24<<2 == 96

24<<3 == 192

Left bitshift is multiplication by 2.

a<<1 is 2a

left bitshift multiplies by 2



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

```
a * 2        : 4
a * 2 * 2    : 8
a * 2 * 2 * 2: 16
```

…and the right-bitshift?

The last binary bit is the remainder after dividing by 2.

$(24)_{10}$ = $(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 1000)_2$.

$(25)_{10}$ = $(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 1001)_2$

After shifting right the last bit is lost.

$(24>>1)_{10}$ = $(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 00001\ 100)_2$ = $(12)_{10}$

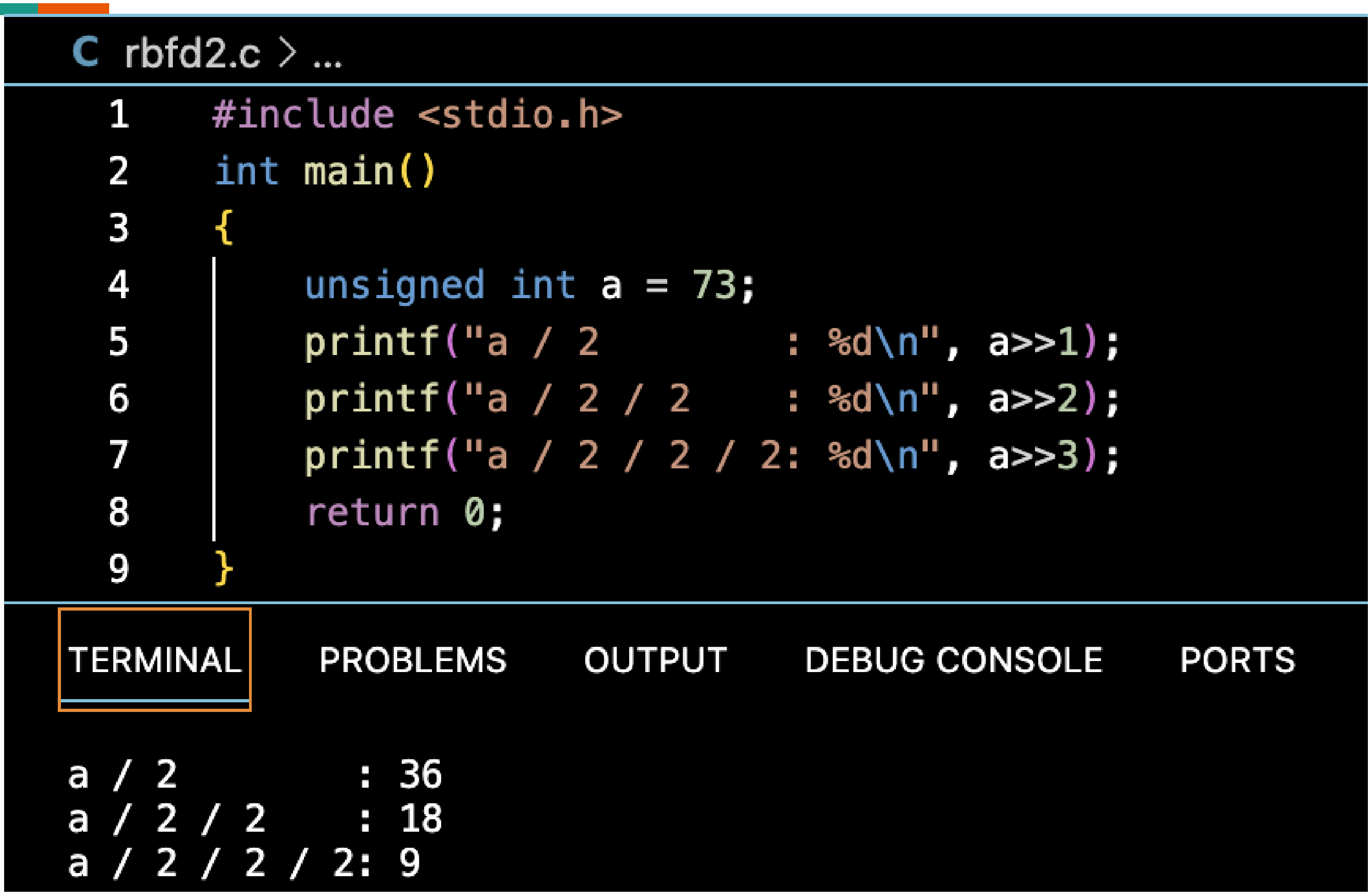
$(25>>1)_{10}$ = $(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 00001\ 100)_2$ = $(12)_{10}$

So (2k>>1) and ((2k+1)>>1) both give k.

Right-bitshift is *floor* division by 2.

a>>1 is $\lfloor a/2 \rfloor$

...and the right-bitshift?



rbfd2.c > ...

```c
#include <stdio.h>
int main()
{
    unsigned int a = 73;
    printf("a / 2         : %d\n", a>>1);
    printf("a / 2 / 2     : %d\n", a>>2);
    printf("a / 2 / 2 / 2: %d\n", a>>3);
    return 0;
}
```

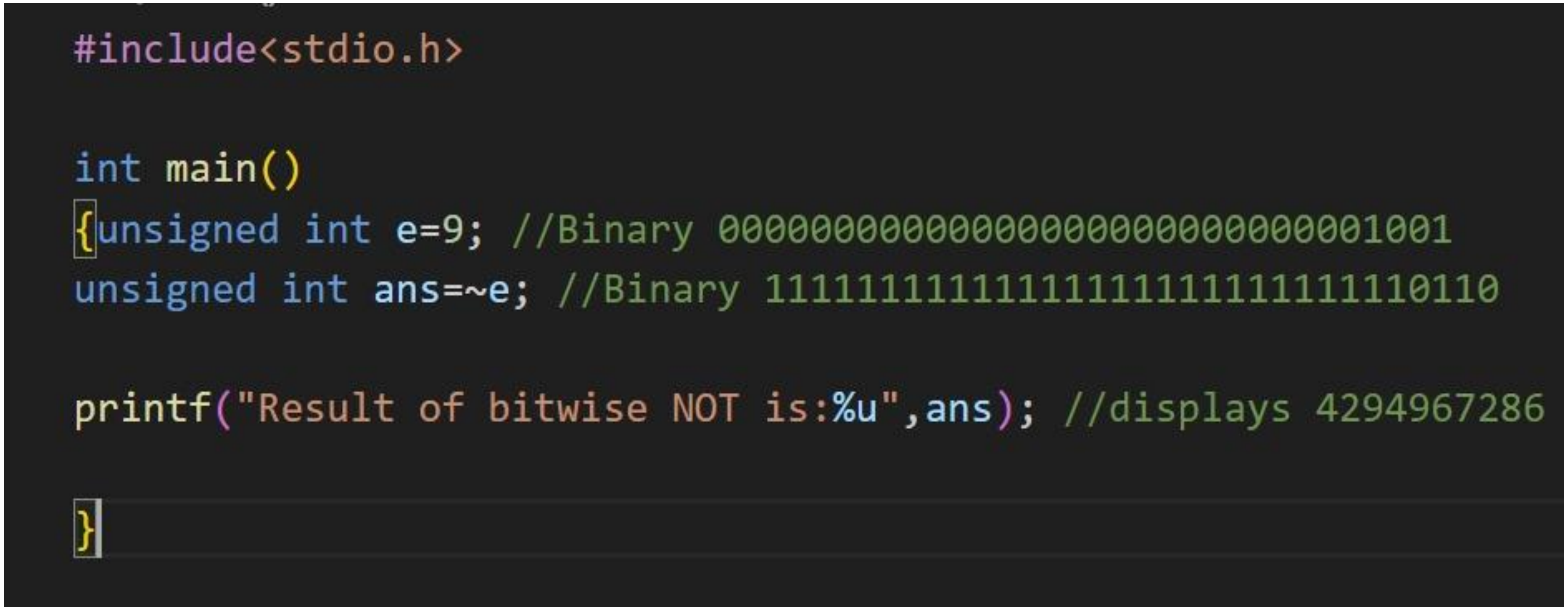
```
a / 2        : 36
a / 2 / 2    : 18
a / 2 / 2 / 2: 9
```

BITWISE NOT(‘~’) OPERATOR

- The Bitwise ~ operator carries out a logical NOT operation on each bit of a number.
- This operation flips the bits of the number, meaning 0 turns into 1 and 1 turns into 0.
- For instance, let’s take an unsigned variable e with a value of 5.
    - The binary representation of e (5) is 0101.
- When we apply the bitwise NOT operation to this variable, we get:
    - ~e = 1010
- In many programming languages, including C, the bitwise NOT operation for signed numbers is implemented using the 2’s complement representation. This topic will be covered in the next semester if you are a CS/electronics student.
- However, in this discussion, we are only covering the bitwise NOT operation for unsigned integers.

continued...

- Since unsigned int are 32-bit integers in C the actual binary representation of 5 is 00000000000000000000000000000101. So, ~e results in flipping each bit of this 32-bit number, which gives 11111111111111111111111111111010.
- Basically, we just need to add enough zeroes in front to make it 32-bit.
- This binary number is equivalent to 4294967290 in decimal.



```
#include<stdio.h>

int main()
{unsigned int e=9; //Binary 00000000000000000000000000001001
unsigned int ans=~e; //Binary 11111111111111111111111111110110

printf("Result of bitwise NOT is:%u",ans); //displays 4294967286

}
```