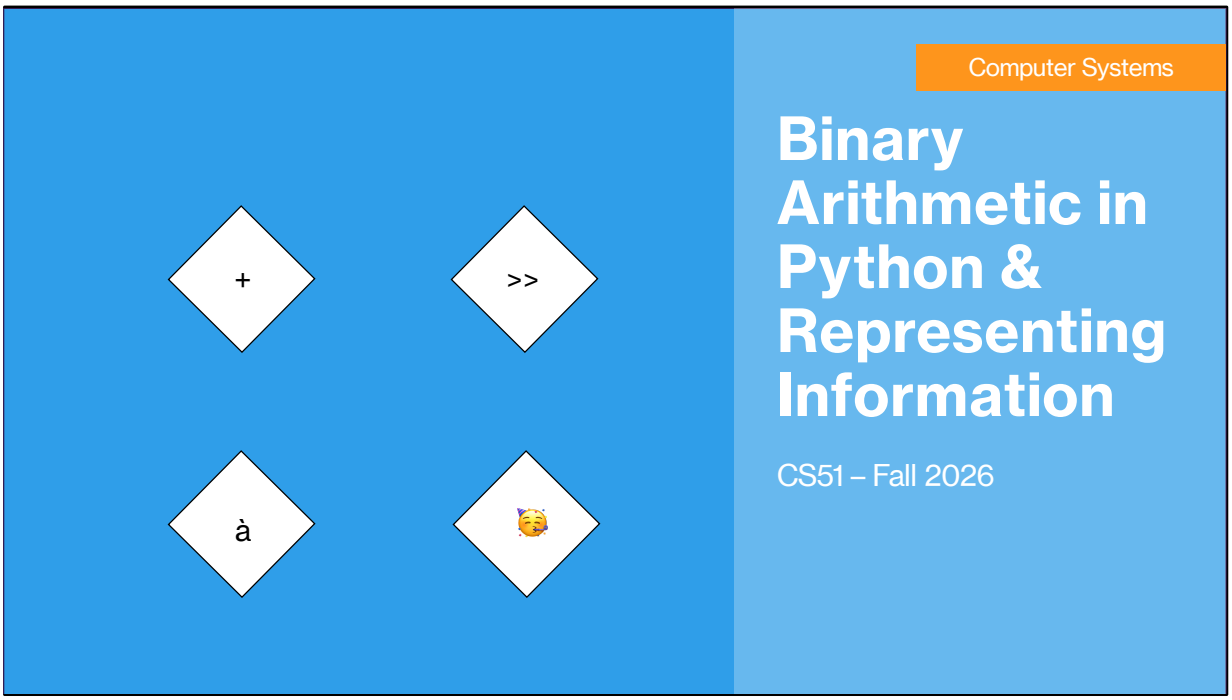


Today we will continue with binary arithmetic, seeing some tricks we can do in Python. We will also see how to use them to represent information that goes past integers (e.g., characters, emojis, even colors!)

Bitwise operations

2

The first thing we will see is how we can apply apply bitwise operations on binary numbers

Bitwise operations

- If we are given two numbers, we can apply the operators NOT, AND, OR, and XOR **bitwise.**
- Treat numbers as two's complement.
- Perform Boolean operation *per bit*.
- Convert result to decimal.

AND (&)

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

NOT (~)

X	¬ X
0	1
1	0

OR (|)

X	Y	X ∨ Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

3

We have seen that in Boolean Algebra that we have the logical operators NOT, AND, OR, and XOR. Given two numbers, we can apply each of these operators bitwise. We treat the numbers as two's complement representations in their binary form and then apply the Boolean operation for each bit of the two numbers. Note that we have slightly different notation to denote bitwise operations. AND is known as &, OR as |, XOR as ^. The only unary operator is NOT and is denoted as ^. Be careful because ^ in Boolean Algebra is AND but in bitwise operations is XOR.

Bitwise operation AND (&)

- 6 & 5 = ?

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- Convert result to decimal.

AND (&)

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

4

Given this recipe, what do you think the result of bitwise AND would be if we applied it on 6 and 5, that is what is 6 & 5 equal to?

Bitwise operation AND (&)

- 6 & 5 = ?

- **Treat numbers as two's complement.**
- Perform Boolean operation *per bit.*
- Convert result to decimal.

$6_{10} = ?_2$

$5_{10} = ?_2$

AND (&)

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

5

First, we convert 6 and 5 to their binary form in two's complement. What are they equal to?

Bitwise operation AND (&)

- 6 & 5 = ?

- **Treat numbers as two's complement.**
- Perform Boolean operation *per bit.*
- Convert result to decimal.

$$6_{10} = 0\ 1\ 1\ 0_2$$
$$5_{10} = 0\ 1\ 0\ 1_2$$

AND (&)

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

6

Let's assume 4-bits here (if we didn't have the fourth bit, 110 would be seen as a negative number!). We get 0110 and 0101, respectively.

Bitwise operation AND (&)

- 6 & 5 = ?

- Treat numbers as two's complement.
- **Perform Boolean operation *per bit*.**
- Convert result to decimal.

$$6_{10} = 0\ 1\ 1\ 0_2$$
$$\&\ \ 5_{10} = 0\ 1\ 0\ 1_2$$

AND (&)

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

7

Next, can you apply the operation of AND on each pair of bits starting from right and going left?

Bitwise operation AND (&)

- 6 & 5 = ?

- Treat numbers as two's complement.
- **Perform Boolean operation *per bit*.**
- Convert result to decimal.

$$6_{10} = 0\,1\,1\,0_2$$
$$\&\ \ 5_{10} = 0\,1\,0\,1_2$$
$$0\,1\,0\,0_2$$

AND (&)

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

8

Did you get 0100?

Bitwise operation AND (&)

AND (&)

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

- 6 & 5 = ?

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- **Convert result to decimal.**

$$6_{10} = 0\,1\,1\,0_2$$
$$\&\quad 5_{10} = 0\,1\,0\,1_2$$
$$?_{10} = 0\,1\,0\,0_2$$

9

Assuming that 0100 is in the two's complement form of a decimal number, which one is it?

Bitwise operation AND (&)

- 6 & 5 = 4

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- **Convert result to decimal.**

$$6_{10} = 0\,1\,1\,0_2$$
$$\&\ \ 5_{10} = 0\,1\,0\,1_2$$
$$4_{10} = 0\,1\,0\,0_2$$

AND (&)

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

10

That's right, it's 4! So putting it together 6&5=4

Bitwise operation OR (|)

- 11 | -9 = ?

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- Convert result to decimal.

OR (|)

X	Y	X ∧ Y
0	0	0
0	1	1
1	0	1
1	1	1

11

Let's do the same now with bitwise OR applied on 11 and -9.

Bitwise operation OR (|)

- 11 | -9 = ?

- **Treat numbers as two's complement.**
- Perform Boolean operation *per bit.*
- Convert result to decimal.

$11_{10} = ?_2$

$-9_{10} = ?_2$

OR (|)

X	Y	X ∧ Y
0	0	0
0	1	1
1	0	1
1	1	1

12

What are the two's complement binary representations of 11 and -9?

Bitwise operation OR (|)

- 11 | -9 = ?

- **Treat numbers as two's complement.**
- Perform Boolean operation *per bit.*
- Convert result to decimal.

$11_{10} = 0\ 1\ 0\ 1\ 1_2$

$-9_{10} = 1\ 0\ 1\ 1\ 1_2$

OR (|)

X	Y	X ∧ Y
0	0	0
0	1	1
1	0	1
1	1	1

13

Did you get 01011 and 10111?

Bitwise operation OR (|)

- 11 | -9 = ?

- Treat numbers as two's complement.
- **Perform Boolean operation *per bit*.**
- Convert result to decimal.

OR (|)

X	Y	X ∧ Y
0	0	0
0	1	1
1	0	1
1	1	1

$$11_{10} = 0\ 1\ 0\ 1\ 1_2$$
$$|\ -9_{10} = 1\ 0\ 1\ 1\ 1_2$$

14

Next, apply OR on each pair of bits starting from right and going to left.

Bitwise operation OR (|)

- 11 | -9 = ?

- Treat numbers as two's complement.
- **Perform Boolean operation *per bit*.**
- Convert result to decimal.

OR (|)

X	Y	X ∧ Y
0	0	0
0	1	1
1	0	1
1	1	1

$$11_{10} = 0\,1\,0\,1\,1_2$$
$$|\ -9_{10} = 1\,0\,1\,1\,1_2$$
$$1\,1\ \ 1\,1\,1_2$$

15

You should gave found it to be equal to 11111.

Bitwise operation OR (|)

- 11 | -9 = ?

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- **Convert result to decimal.**

OR (|)

X	Y	X ∧ Y
0	0	0
0	1	1
1	0	1
1	1	1

$$11_{10} = 0\,1\,0\,1\,1_2$$
$$|\ -9_{10} = 1\,0\,1\,1\,1_2$$
$$?_{10} = 1\,1\ \ 1\,1\,1_2$$

16

Which decimal number is this equal to?

Bitwise operation OR (|)

- 11 | -9 = -1

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- **Convert result to decimal.**

OR (|)

X	Y	X ∧ Y
0	0	0
0	1	1
1	0	1
1	1	1

$$11_{10} = 0\ 1\ 0\ 1\ 1_2$$
$$|\ \ -9_{10} = 1\ 0\ 1\ 1\ 1_2$$
$$-1_{10} = 1\ 1\ \ 1\ 1\ 1_2$$

17

-1 (remember, it's always -1 if it's all 1s). So 11 | -9 = -1

Bitwise operation NOT (~)

- ~9 = ?

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- Convert result to decimal.

NOT (~)

X	¬ X
0	1
1	0

18

Let's see how bitwise NOT play out by applying it on 9.

Bitwise operation NOT (~)

NOT (~)

X	¬ X
0	1
1	0

- ~9 = ?

- **Treat numbers as two's complement.**
- Perform Boolean operation *per bit.*
- Convert result to decimal.

$$9_{10} = ?_2$$

19

You know the drill. Start by converting 9 to its two's complement representation.

Bitwise operation NOT (~)

NOT (~)

X	¬ X
0	1
1	0

- ~9 = ?

- **Treat numbers as two's complement.**
- Perform Boolean operation *per bit.*
- Convert result to decimal.

$$9_{10} = 0\ 1\ 0\ 0\ 1_2$$

20

It should be 01001. Don't forget we need to pad with a zero at the beginning otherwise 1001 would be seen as a negative number since we're working with signed numbers.

Bitwise operation NOT (~)

NOT (~)

X	¬ X
0	1
1	0

- ~9 = ?

- Treat numbers as two's complement.
- **Perform Boolean operation *per bit.***
- Convert result to decimal.

$$\sim 9_{10} = 0\ 1\ 0\ 0\ 1_2$$

21

Proceed with applying NOT to each bit.

Bitwise operation NOT (~)

NOT (~)

X	¬ X
0	1
1	0

- ~9 = ?

- Treat numbers as two's complement.
- **Perform Boolean operation *per bit*.**
- Convert result to decimal.

$$\sim 9_{10} = 0\,1\,0\,0\,1_2$$

$$1\,0\,1\,1\,0_2$$

22

You shuld get 10110

Bitwise operation NOT (~)

NOT (~)

X	¬ X
0	1
1	0

- ~9 = ?

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- **Convert result to decimal.**

$$\sim 9_{10} = 0\,1\,0\,0\,1_2$$

$$?_{10} = 1\,0\,1\,1\,0_2$$

23

What is 10110 in decimal?

Bitwise operation NOT (~)

NOT (~)

X	¬ X
0	1
1	0

- ~9 = -10

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- **Convert result to decimal.**

$$\sim 9_{10} = 0\ 1\ 0\ 0\ 1_2$$

$$-10_{10} = 1\ 0\ 1\ 1\ 0_2$$

24

It's -10. So ~9=-10.

Bitwise operation XOR (^)

- 10^1 = ?

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- Convert result to decimal.

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

25

And finally, let's see what bitwise XOR would look like when applied to 10 and 1.

Bitwise operation XOR (^)

- 10^1 = ?

- **Treat numbers as two's complement.**
- Perform Boolean operation *per bit.*
- Convert result to decimal.

$$10_{10} = ?_2$$
$$1_{10} = ?_2$$

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

26

Start by converting 10 and 1 to their binary two's complement form.

Bitwise operation XOR (^)

- 10^1 = ?

- **Treat numbers as two's complement.**
- Perform Boolean operation *per bit.*
- Convert result to decimal.

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

$$10_{10} = 0\ 1\ 0\ 1\ 0_2$$

$$1_{10} = 0\ 1_2$$

27

10 is 01010 (again don't forget to pad with the 0 at the beginning so that it's positive) and 1 is. Notice anything weird?

Bitwise operation XOR (^)

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

- 10^1 = ?

- **Treat numbers as two's complement.**
- Perform Boolean operation *per bit.*
- Convert result to decimal.

$$10_{10} = 0\ 1\ 0\ 1\ 0_2$$
$$1_{10} = 0\ 0\ 0\ 0\ 1_2$$

28

Exactly, they are unequal lengths. So how we will resolve this is with sign extension so that the smaller number has the same number of bits with the larger one.

Bitwise operation XOR (^)

- 10^1 = ?

- Treat numbers as two's complement.
- **Perform Boolean operation *per bit*.**
- Convert result to decimal.

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

$$10_{10} = 0\ 1\ 0\ 1\ 0_2$$
$$\wedge\quad 1_{10} = 0\ 0\ 0\ 0\ 1_2$$

29

Now proceed as usual by applying XOR on each pair of bits

Bitwise operation XOR (^)

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

- 10^1 = ?

- Treat numbers as two's complement.
- **Perform Boolean operation *per bit*.**
- Convert result to decimal.

$$\begin{array}{rr} 10_{10} = & 0\,1\;\;0\,1\,0_2 \\ \hat{}\quad 1_{10} = & 0\,0\,0\,0\,1_2 \\ \hline & 0\,1\;\;0\;\;1\,1_2 \end{array}$$

30

Now proceed as usual by applying XOR on each pair of bits

Bitwise operation XOR (^)

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

- 10^1 = ?

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- **Convert result to decimal.**

$$10_{10} = 0\ 1\ 0\ 1\ 0_2$$
$$\wedge\quad 1_{10} = 0\ 0\ 0\ 0\ 1_2$$
$$?_{10} = 0\ 1\ 0\ 1\ 1_2$$

31

Which decimal number is this?

Bitwise operation XOR (^)

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

- 10^1 = 11

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- **Convert result to decimal.**

$$10_{10} = 0\ 1\ 0\ 1\ 0_2$$
$$\wedge\ \ 1_{10} = 0\ 0\ 0\ 0\ 1_2$$
$$11_{10} = 0\ 1\ 0\ 1\ 1_2$$

32

It's 11, so 10^1=11

Bitwise operation XOR (^)

- 10^-1 = ?

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- Convert result to decimal.

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

33

What if instead of 10^1 we applied 10^-1?

Bitwise operation XOR (^)

- 10^-1 = ?

- **Treat numbers as two's complement.**
- Perform Boolean operation *per bit.*
- Convert result to decimal.

$10_{10} = ?_2$

$-1_{10} = ?_2$

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

34

Start by converting 10 and -1 to their binary two's complement form.

Bitwise operation XOR (^)

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

- 10^-1 = ?

- **Treat numbers as two's complement.**
- Perform Boolean operation *per bit.*
- Convert result to decimal.

$$10_{10} = 0\ 1\ 0\ 1\ 0_2$$

$$-1_{10} = 1_2$$

35

10 is 01010 (again don't forget to pad with the 0 at the beginning so that it's positive) and 1 is 1. Notice anything weird?

Bitwise operation XOR (^)

- 10^-1 = ?

- **Treat numbers as two's complement.**
- Perform Boolean operation *per bit.*
- Convert result to decimal.

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

$10_{10} = 0\ 1\ 0\ 1\ 0_2$

$-1_{10} = 1\ 1\ 1\ 1\ 1_2$

36

Again, they are unequal lengths. So again, how we will resolve this is with sign extension so that the smaller number has the same number of bits with the larger one.

Bitwise operation XOR (^)

- 10^-1 = ?

- Treat numbers as two's complement.
- **Perform Boolean operation *per bit*.**
- Convert result to decimal.

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

$$10_{10} = 0\ 1\ 0\ 1\ 0_2$$
$$\wedge\ \ -1_{10} = 1\ 1\ 1\ 1\ 1_2$$

37

Now proceed as usual by applying XOR on each pair of bits

Bitwise operation XOR (^)

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

- 10^-1 = ?

- Treat numbers as two's complement.
- **Perform Boolean operation *per bit*.**
- Convert result to decimal.

$$10_{10} = 0\ 1\ 0\ 1\ 0_2$$
$$\wedge\ \ -1_{10} = 1\ 1\ 1\ 1\ 1_2$$
$$1\ 0\ 1\ 0\ 1_2$$

38

Now proceed as usual by applying XOR on each pair of bits

Bitwise operation XOR (^)

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

- 10^-1 = ?

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- **Convert result to decimal.**

$$10_{10} = 0\ 1\ 0\ 1\ 0_2$$
$$\wedge\ \ -1_{10} = 1\ 1\ 1\ 1\ 1_2$$
$$?_{10} = 1\ 0\ 1\ 0\ 1_2$$

39

Which decimal number is this?

Bitwise operation XOR (^)

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

- 10^-1 = -11

- Treat numbers as two's complement.
- Perform Boolean operation *per bit.*
- **Convert result to decimal.**

$$10_{10} = 0\ 1\ 0\ 1\ 0_2$$
$$\wedge\quad 1_{10} = 1\ 1\ 1\ 1\ 1_2$$
$$-11_{10} = 1\ 0\ 1\ 0\ 1_2$$

40

It's -11, so 10^-1=-11

PRACTICE TIME – Bitwise operations

- Let's assume we have $x = 5_{10}$ and $y = -9_{10}$
  - $\sim x$
  - $\sim y$
  - $x \& y$
  - $x | y$
  - $x \hat{} y$

AND (&)

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

NOT (~)

X	¬ X
0	1
1	0

OR (|)

X	Y	X ∨ Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

Let's check our understanding with these two numbers.

ANSWER – Bitwise operations

- Let's assume we have $x = 5_{10}$ and $y = -9_{10}$
  - $\sim x = 11010_2 = -6_{10}$
  - $\sim y = 01000_2 = 8_{10}$
  - $x\&y = 00101_2 = 5_{10}$
  - $x|y = 10111_2 = -9_{10}$
  - $x\hat{}y = 10010_2 = -14_{10}$

AND (&)

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

NOT (~)

X	¬ X
0	1
1	0

OR (|)

X	Y	X ∨ Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR (^)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

Did you get these results?

Masking

- We can use bitwise operations to **mask** a certain number of bits in a number.
- By masking, we can reset, set, or invert certain bits in a number.
- Bitwise AND will reset a subset of the bits to 0
- Bitwise OR will set a subset of the bits to 1
- Bitwise XOR will invert a subset of the bits

43

Bitwise operations often go hand in hand with masking. When we apply a mask on an input, we isolate certain bits and reset them to 0 (logical AND), set them to 1 (logical OR) or invert them (XOR). We will see each one with examples and an opportunity to practice.

Bitwise AND – Resetting bits to 0

We want to reset the three least significant bits to 0

&

Bit	6th	5th	4th	3rd	2nd	1st	0th
Data	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Mask							
Result							

What number should we bitwise-AND (&) to do this?

44

Say we have a 7-bit number and want to reset the three least significant (right-most) bits to 0. What number should we bitwise-AND to do this?

Bitwise AND – Resetting bits to 0

We want to reset the three least significant bits to 0

&

Bit	6th	5th	4th	3rd	2nd	1st	0th
Data	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Mask	1	1	1	1	0	0	0
Result							

What number should we bitwise-AND (&) to do this?

45

We'd need 1111 in the first 4 bits, and 000 in the three bits, resulting to the number 1111000

Bitwise AND – Resetting bits to 0

We want to reset the three least significant bits to 0

&

Bit	6th	5th	4th	3rd	2nd	1st	0th
Data	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Mask	1	1	1	1	0	0	0
Result							

What number is this in decimal?

46

What number is this in decimal?

Bitwise AND – Resetting bits to 0

We want to reset the three least significant bits to 0

& -8_{10}

Bit	6th	5th	4th	3rd	2nd	1st	0th
Data	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Mask	1	1	1	1	0	0	0
Result							

What number is this in decimal?

47

It's -10. Here's a trick, you can ignore all but the rightmost 1 for the sign and then that's -8+0+0+0=-8.

Bitwise AND – Resetting bits to 0

Say we have 47 & -8. *What's the result?*

47_{10}
& -8_{10}

Bit	6th	5th	4th	3rd	2nd	1st	0th
Data	0	1	0	1	1	1	1
Mask	1	1	1	1	0	0	0
Result	0	1	0	1	**0**	**0**	**0**

48

Say we now try to apply a mask to reset the last three bits of the number 47, that is we apply 47&-8. What's the result?

Bitwise AND – Resetting bits to 0

Say we have 47 & -8. *What's the result?*

	Bit	6th	5th	4th	3rd	2nd	1st	0th
47_{10}	**Data**	0	1	0	1	1	1	1
& -8_{10}	**Mask**	1	1	1	1	0	0	0
40_{10}	**Result**	0	1	0	1	**0**	**0**	**0**

49

It should be 0101000 or 40 in decimal.

PRACTICE TIME - Bitwise AND

- What would the result be of applying the mask 42_{10} on the input 85_{10} using bitwise AND?

50

Your turn. Start by converting 42 and 85 to binary before you apply bitwise AND. What was its effects on the bits? What will the result be in decimal?

ANSWER - Bitwise AND

- What would the result be of applying the mask 42_{10} on the input 85_{10} using bitwise AND? First, we convert from decimal to binary, working with 8 bits for both numbers:
  - $42_{10} = 00101010_2$
  - $85_{10} = 01010101_2$
- We then apply the bitwise AND operation which will reset the the 7th (no real effect), 6th, 4th, 2nd, and 0th bit of the input, resulting to 0_{10}:

	Bit	7th	6th	5th	4th	3rd	2nd	1st	0th
85_{10}	**Data**	0	1	0	1	0	1	0	1
& 42_{10}	**Mask**	0	0	1	0	1	0	1	0
0_{10}	**Result**	**0**	**0**	0	**0**	0	**0**	0	**0**

51

Don't forget that since we are working with positive numbers we'd need 8 bits for 85. And since 42 can be described with 7 bits, we'd need to extend the sign by adding an extra 0.

Bitwise OR – Setting bits to 1

We want to set the two most significant bits to 1

Bit	6th	5th	4th	3rd	2nd	1st	0th
Data	0/1	0/1	0/1	0/1	0/1	0/1	0/1
| Mask							
Result							

What number should we bitwise-OR (|) to do this?

52

Say we have a 7-bit number and want to set the two most significant (left-most) bits to 1. What number should we bitwise-OR to do this?

Bitwise OR – Setting bits to 1

We want to set the two most significant bits to 1

Bit	6th	5th	4th	3rd	2nd	1st	0th
Data	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Mask	1	1	0	0	0	0	0
Result							

What number should we bitwise-OR (|) to do this?

53

We'd need 11 in the first 2 bits, and 00000 in the next five bits, resulting to the number 1100000

Bitwise OR – Setting bits to 1

We want to set the two most significant bits to 1

Bit	6th	5th	4th	3rd	2nd	1st	0th
Data	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Mask	1	1	0	0	0	0	0
Result							

What number is this in decimal?

54

What number is this in decimal?

Bitwise OR – Setting bits to 1

We want to set the two most significant bits to 1

-32_{10}	Bit	6^{th}	5^{th}	4^{th}	3^{rd}	2^{nd}	1^{st}	0^{th}
	Data	0/1	0/1	0/1	0/1	0/1	0/1	0/1
|	**Mask**	1	1	0	0	0	0	0
	Result							

What number is this in decimal?

55

It's -32. Here's a trick, you can ignore all but the rightmost 1 for the sign and then that's -32

Bitwise OR – Setting bits to 1

Say we have 47 | - 32. *What's the result?*

47_{10}
$| \ -32_{10}$

Bit	6th	5th	4th	3rd	2nd	1st	0th
Data	0	1	0	1	1	1	1
Mask	1	1	0	0	0	0	0
Result	**1**	**1**	0	1	1	1	1

56

Say we have 47 | -32. *What's the result?*

Bitwise OR – Setting bits to 1

Say we have 47 | -32. *What's the result?*

47_{10}
$| \ -32_{10}$
-17_{10}

Bit	6th	5th	4th	3rd	2nd	1st	0th
Data	0	1	0	1	1	1	1
Mask	1	1	0	0	0	0	0
Result	**1**	**1**	0	1	1	1	1

57

It should be 1101111 or -17 in decimal.

PRACTICE TIME - Bitwise OR

- What would the result be of applying the mask 17_{10} on the input -27_{10} using bitwise OR?

58

Your turn! As before, convert 17 and -27 to binary (using signed extension for two's complement) .

ANSWER - Bitwise OR

- What would the result be of applying the mask 17_{10} on the input -27_{10} using bitwise OR? You can assume you have 8 bits.
- First, we convert from decimal to binary:
  - $17_{10} = 010001_2$
  - $-27_{10} = 100101_2$
- We then apply the bitwise OR operation which will set the the 4th and 0th bit of the input to 1, resulting to -11_{10}:

-27_{10}
$| \ 17_{10}$
-11_{10}

Bit	5th	4th	3rd	2nd	1st	0th
Data	1	0	0	1	0	1
Mask	0	1	0	0	0	1
Result	1	**1**	0	1	0	**1**

59

This will result in the 4th and 0th bit being set to 1.

Bitwise XOR – Inverting bits

We want to invert the two most and two least significant bits.

^

Bit	7th	6th	5th	4th	3rd	2nd	1st	0th
Data	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Mask								
Result								

What number should we bitwise-XOR (^) to do this?

60

Say we have a 7-bit number and want to invert the two most significant (left-most) and two least significant (right-most) bits. What number should we bitwise-XOR to do this?

Bitwise XOR – Inverting bits

We want to invert the two most and two least significant bits.

^

Bit	7th	6th	5th	4th	3rd	2nd	1st	0th
Data	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Mask	1	1	0	0	0	0	1	1
Result								

What number should we bitwise-XOR (^) to do this?

61

We'd need 11 in the first and last 2 bits, that is 11000011

Bitwise XOR – Inverting bits

We want to invert the two most and two least significant bits.

^

Bit	7th	6th	5th	4th	3rd	2nd	1st	0th
Data	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Mask	1	1	0	0	0	0	1	1
Result								

What number is this in decimal?

62

What number is this in decimal?

Bitwise XOR – Inverting bits

We want to invert the two most and two least significant bits.

^ -61_{10}

Bit	7th	6th	5th	4th	3rd	2nd	1st	0th
Data	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Mask	1	1	0	0	0	0	1	1
Result								

What number is this in decimal?

63

It's -61. Here's a trick, you can ignore all but the rightmost 1 for the sign and then that's -64+2+1=-61

Bitwise XOR – Inverting bits

Say we have 117 ^ -61. *What's the result?*

117_{10}
$\wedge -61_{10}$

Bit	7th	6th	5th	4th	3rd	2nd	1st	0th
Data	0	1	1	1	0	1	0	1
Mask	1	1	0	0	0	0	1	1
Result								

64

Say we have 117 ^ -61. *What's the result?*

Bitwise XOR – Inverting

Say we have 117 ^ -61. *What's the result?*

117_{10}
$\wedge -61_{10}$
-74_{10}

Bit	7th	6th	5th	4th	3rd	2nd	1st	0th
Data	0	1	1	1	0	1	0	1
Mask	1	1	0	0	0	0	1	1
Result	**1**	**0**	1	1	0	1	**1**	**0**

65

It should be 10110110 or -128+32+16+4+2=-74 in decimal.

Digital logic mysteries

- Bitwise operations can be quite useful in digital logic and can make code both more concise and faster.
  - We can represent many Boolean values with a single integer (e.g., in memory constrained environments)
  - Are fast, so can speed up computation (if you can pose your problem as a bitwise operation)
  - Interacting with low-level hardware
- Next assignment will give you more practice with bitwise operations.
- In the meantime, let’s work on some mystery functions.

66

Bitwise operations can be quite useful in digital logic and can make code both more concise and faster.
This week's assignment will give you more practice with bitwise operations.
In the meantime, let’s work on some mystery functions. You want to try different numbers to come up with theories of what they do.

PRACTICE TIME - Mystery 1

- What does the following Python function do?

```
def mystery1(a, b):
      return (a ^ b) == 0
```

67

Think, what type will this function return? If you assume a and b are integers, what are good inputs to test? When would the result of applying a bitwise xor result to the decimal number 0?

ANSWER - Mystery 1

- What does the following Python function do?

```
def mystery1(a, b):
      return (a ^ b) == 0
```

- The function takes two numbers (a,b), applies the bitwise XOR (a^b), turns it into a decimal and compares the result to 0. If it is equal to 0, it returns True, otherwise it returns False.
- If both numbers have n bits, we have $a_{n-1}a_{n-2} \dots a_0$
- Tests if a and b are equal.
- $5_{10} \wedge 5_{10}$ is $0101_2 \wedge 0101_2 = 0000_2 = 0_{10}$
- $5_{10} \wedge 6_{10}$ is $0101_2 \wedge 0110_2 = 0100_2 = 4_{10}$

68

The function returns a bool. It seems like it's converting the result of the XOR bitwise operation on a and b to a decimal number and comparing it against 0. 0 in decimal is 0000…0 in binary. For the result of a bitwise XOR operation to always be 0, we would need every single bit we compare between the two numbers to be equal.

PRACTICE TIME - Mystery 2

- What does the following Python function do? You can assume we are working with 4 bits

```
def mystery2(a):
      return (a & (1<<3)) != 0
```

69

Again, what is the return type? Try a few different inputs for a.

ANSWER - Mystery 2

- What does the following Python function do? You can assume we are working with 4 bits

```
def mystery2(a):
      return (a & (1<<3)) != 0
```

- Checks whether a is negative by isolating the MSB.

70

Don't forget that if you have an n-bit number, the MSB would be at the n-1 bit (starting 0 to the right and advancing to the left)

PRACTICE TIME - Mystery 3

- What does the following Python function do?

```
def mystery3(a,b):
      return (a & (1<<b)) != 0
```

71

This should feel very familiar. The only thing we changed is 3 to b.

ANSWER - Mystery 3

- What does the following Python function do?

```
def mystery3(a,b):
      return (a & (1<<b)) != 0
```

- Tests if the b-th bit in a is 1.

72

So now we test the b-th bit, not necessarily the MSB. Ideally, we should have checked that b is within the n bits that a has but that's a bit complicated in languages like Python.

PRACTICE TIME - Mystery 4

- What does the following Python function do?

```
def mystery4(a):
      return a | 1
```

73

Last head scratcher. What does this super concise function do? Let's start by realizing that it does not return a bool but a decimal number

ANSWER - Mystery 4

- What does the following Python function do?

```
def mystery4(a):
      return a | 1
```

- If a is even, it returns a+1, if a is odd, it leaves it unchanged.

74

Cool, huh?

Beyond numbers

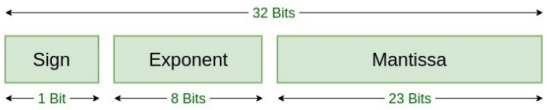
Representing information

- So far, we have seen that computers store information in bits and we have interpreted these bits as integers.
- But what about non-integer numbers, such as floating-point numbers, or any other type of information, such as letters, emojis, colors, sound etc?
- There are different international standards that determine how binary is **encoded** into other representations.

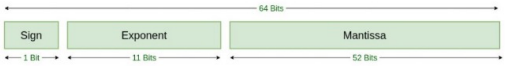
76

So far, we have seen that computers store information in bits and we have interpreted these bits as integers.
But what about non-integer numbers, such as floating-point numbers, or any other type of information, such as letters, emojis, colors, sound etc?
There are different international standards that determine how binary is **encoded** into other representations.

Floating point numbers



Single Precision
IEEE 754 Floating-Point Standard



Double Precision
IEEE 754 Floating-Point Standard

77

For example, the most used standard to encode real numbers is the IEEE 754 standard.
But under the hood, it's all bits!

Encoding text - ASCII

- **ASCII** (American Standard Code for Information Interchange) which was established in the '60s and used 7 bits to represent 128 characters: These included the capital and lowercase letters of the English alphabet, digits 0-9, punctuation and special symbols like @.
  - For example, 01000011 01010011 00110101 00110001 00100001 represents the text "CS51!"
- ASCII was adopted by different computer manufacturers ensuring **interoperability**.
- Soon an 8th bit was used to expand the available set to 255 characters.
- In the US, this was used for accented characters and mathematical symbols and many other languages used these extra characters to encode their own alphabets.
- But that meant that text written in different languages encoded to gibberish when read in a different context.

78

ASCII (American Standard Code for Information Interchange) which was established in the '60s and used 7 bits to represent 128 characters: These included the capital and lowercase letters of the English alphabet, digits 0-9, punctuation and special symbols like @.

For example, 01000011 01010011 00110101 00110001 00100001 represents the text "CS51!"

ASCII was adopted by different computer manufacturers ensuring **interoperability**.

Soon an 8th bit was used to expand the available set to 255 characters.

In the US, this was used for accented characters and mathematical symbols and many other languages used these extra characters to encode their own alphabets.

But that meant that text written in different languages encoded to gibberish when read in a different context.

Encoding text - Unicode

- In 1990s, **Unicode** expanded encoding to thousands of characters to account for all different schemes individual languages used and even includes emojis!
- The most common implementations of Unicode are **UTF-8** , which was designed for backward compatibility with ASCII, and **UTF-16**.

Smileys & Emotion

face-smiling

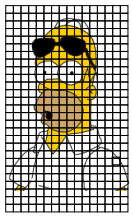
№	Code	Browser	Sample	GMail	SB	DCM	KDDI	CLDR Short Name
1	U+1F600	😀	😀	😀	–	–	–	grinning face
2	U+1F603	😃	😃	😃	😃	😃	😃	grinning face with big eyes
3	U+1F604	😄	😄	😄	😄	–	–	grinning face with smiling eyes
4	U+1F601	😁	😁	😁	😁	😁	😁	beaming face with smiling eyes
5	U+1F606	😆	😆	😆	–	😆	–	grinning squinting face
6	U+1F605	😅	😅	😅	–	😅	–	grinning face with sweat
7	U+1F923	🤣	🤣	–	–	–	–	rolling on the floor laughing
8	U+1F602	😂	😂	😂	😂	–	😂	face with tears of joy
9	U+1F642	🙂	🙂	🙂	–	–	–	slightly smiling face
10	U+1F643			–	–	–	–	upside-down face

In 1990s, **Unicode** expanded encoding to thousands of characters to account for all different schemes individual languages used and even includes emojis!
The most common implementations of Unicode are **UTF-8** ,which was designed for backward compatibility with ASCII, and **UTF-16**.

How is an image represented?

Say we have a picture of Homer. How are images represented?

How is an image represented?



- images are made up of pixels
- for a color image, each pixel corresponds to an RGB value (i.e. three numbers)

Images are made up of pixels so we can think of them as a grid of pixels where each cell corresponds to an RGB value.

Encoding images

- Each pixel stores a **color** using three **color channels**: Red, Green, Blue (RGB).
- Each color channel can be encoded using binary numbers
  - Typical: 8 bits per channel, that is 24 bits per pixel. With 8 bits, we represent 0,..., 255
  - For example, 00000000 represents no red and 11111111 represents full red. Same for green, blue.
- Putting it together, we have triples of numbers to represent pixels. E.g.,
  - Pure red = 11111111 00000000 00000000→ (255,0,0) – only red
  - Pure white = 11111111 11111111 11111111→ (255,255,255) – all three colors
  - Pure black =00000000 00000000 00000000→ (0, 0, 0) – absence of all three colors
- An entire image would be represented as a matrix of pixels (width × height), each pixel's RGB values encoded in binary

82

Images consist of pixels, the smallest image unit.
Each pixel stores a **color** using three **color channels**: **Red, Green, Blue (RGB)**.
Each color channel can be encoded using binary numbers

Typical: 8 bits per channel, that is 24 bits per pixel. With 8 bits, we can represent 0,..., 255
For example, 00000000 represents no red and 11111111 represents full red. Same for green, blue.

Putting it together, we have triples of numbers to represent pixels. E.g.,

Pure red = 11111111 00000000 00000000→ (255,0,0) – only red
Pure white = 11111111 11111111 11111111→ (255,255,255) – all three colors
Pure black =00000000 00000000 00000000→ (0, 0, 0) – absence of all

three colors

An entire image would be represented as a matrix of pixels (width × height), each pixel's RGB values encoded in binary

PRACTICE TIME - Encoding images

- If we have 8 bits for each of the three color channels, how many colors can we support in a 24-bit RGB system?

83

Let's think for a moment, if we have 8 bits for each of the three color channels, how many colors can we support in a 24-bit RGB system?

ANSWER - Encoding images

- If we have 8 bits for each of the three color channels, how many colors can we support in a 24-bit RGB system?
- Each channel supports $2^8 = 256$ possible values.
- Thus, the total number of supported colors is $256 \times 256 \times 256 = 16{,}777{,}216$
- That means that a 24-bit RGB system, can represent over 16 million colors.

84

Each channel supports $2^8 = 256$ possible values.
Thus, the total number of supported colors is
$256 \times 256 \times 256 = 16{,}777{,}216$
That means that a 24-bit RGB system can represent over 16 million colors.

RGBA format

- Sometimes, the RGB format is supplemented with one more channel called the **alpha channel**.
- The alpha channel indicates how opaque a pixel is.
- By convention RGBA colors are stored in hex. For example, for alpha, 00 would be fully transparent and FF fully opaque. For colors, 00 would be lack of color and FF would be pure color.
- For example, the RGBA color #FF00FF80 is a semi-transparent purple:
  - FF_{16} stands for a full red in the red channel
  - 00_{16} stands for no green in the green channel
  - FF_{16} stands for a full blue in the blue channel
  - and $80_{16} = 128_{10}$ represents 50% opacity.

85

Beyond rbg, we often encode one more piece of information about pixels, their alpha value which stands for how opaque a pixel is. We typically work with hex numbers, that is 00 would be fully transparent for alpha or lack of color for rgb, and ff would be fully opaque for alpha and full on color for rgb. You can now see why #FF00FF80 is a semi transparent purple.

Practice Problems

86

Practice Problems – Problem 1

- Convert the following decimal numbers to their *unsigned* binary representation and add them:
  - $45_{10} + 27_{10}$
  - How many bits do you need to not have overflow?
- Assume 6-bit signed numbers in two's complement representation. Add them and state whether overflow occurs:
  - $101110_2 + 010101_2$
  - Convert these three numbers to decimal to double check your work.
- Add the hex numbers 0xA7+0x5C
  - Convert these three numbers to decimal to double check your work.

Practice Problems – Problem 2

- What will the results of the following operations be in binary and decimal assuming 8 bits?

- $-28_{10} \gg 2$

Practice Problems – Problem 3

- Compute the following expressions in Python:
- $101101012_2 \& 11110000_2$
- $01011011_2 | 00101101_2$
- $11001010_2 \wedge 10101100_2$

89

Practice Problems – Answer 1

- Convert the following decimal numbers to their *unsigned* binary representation and add them:
  - $45_{10} + 27_{10} = 101101_2 + 011011_2 = 1001000$
  - How many bits do you need to not have overflow? 7
- Assume 6-bit signed numbers in two's complement representation. Add them and state whether overflow occurs:
  - $101110_2 + 010101_2 = 000011_2$. No overflow as we drop the left most bit (1).
  - Convert these three numbers to decimal to double check your work. -18+21=-3
- Add the hex numbers $A7_{16} + 5C_{16}$
  - 103_{16}
  - Convert these three numbers to decimal to double check your work. $167_{10} + 92_{10} = 259_{10}$

Practice Problems – Answer 2

- What will the results of the following operations be in binary and decimal assuming 8 bits?

- $-28_{10} \gg 2 = 11100100_2 \gg 2 = 11111001_2 = -7_{10}$

Practice Problems – Answer 3

- Compute the following expressions in Python:
- $101101012_2 \& 11110000_2 = 10110000_2$
- $01011011_2 | 00101101_2 = 01111111_2$
- $11001010_2 \wedge 10101100_2 = 01100110_2$

92