CSCI 145 · Introduction to Computing · Fall 2026

Representing Values

Lecture 15 — numbers as bits

Outline

- Bits
- Binary numbers
- Hexadecimal

What's inside a color?

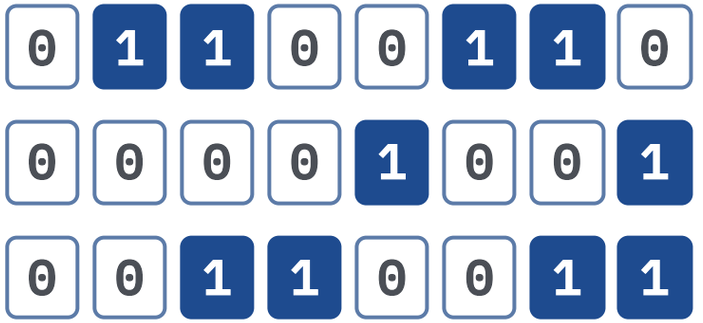


(102, 9, 51)

#660933

From Lecture 12: one color, written two ways

- Why does each color component stop at 255?
- How is red component `66` in the hex color code the same as `102` in the tuple?
- What does the computer actually store?



- By the end of today, you can read all three

Part 1 of 3

Bits

A **bit** is a single 0 or 1

Everything a computer stores is made of bits

One switch, one bit



off

`0`

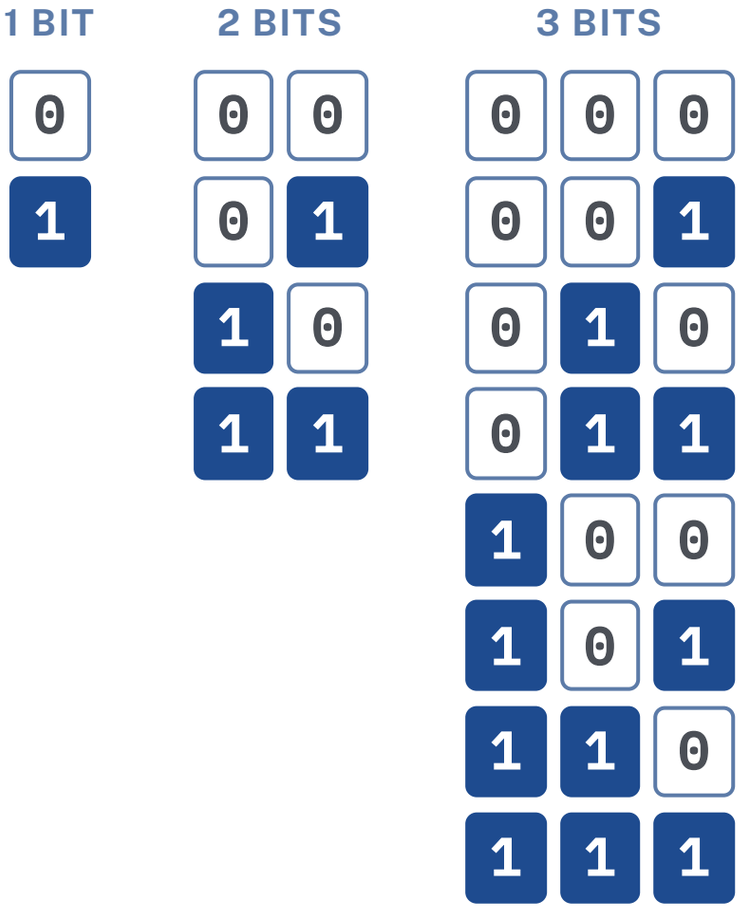


on

`1`

- A computer chip holds billions of tiny switches called **transistors**
- Each switch is **off** or **on**: current flows or it doesn't
- We write **off** as 0 and **on** as 1
  One switch is one **bit** (binary digit)
- One bit can mean no/yes, False/True, black/white, and then we run out of options

More bits, more patterns



1 bit: 2 patterns

2 bits: 2 × 2 = 4 patterns

3 bits: 2 × 2 × 2 = 8 patterns

4 bits: 2 × 2 × 2 × 2 = 16 patterns

`n` bits: 2^n patterns

- Each new bit doubles the count: every old pattern with a 0 in front, then every old pattern with a 1

Names for groups of bits

Name	# of Bits	# of Patterns
Bit	1	2
Nibble	4	16
Byte	8	256
Word (on most laptops)	64	18,446,744,073,709,551,616

- `n` bits give 2^n patterns:
  the numbers 0 to $2^n - 1$
- A byte holds 0 to 255
- A color stores each component in one byte:
  that's why each one stops at 255

```
>>> 2 ** 8
256
>>> 2 ** 64
18446744073709551616
```

How many bits?

What's the fewest bits that can store every whole number from 0 to 1,000?

# of Bits	# of Patterns	Numbers	
8	256	0 to 255	Too few
9	512	0 to 511	Too few
10	1,024	0 to 1,023	Enough

A 8 bits

B 9 bits

C 10 bits

D 11 bits

Answer: C

1,001 numbers need at least 1,001 patterns
10 bits is the first to give that many

Why not the others?

A and B: 256 and 512 patterns are too few
D: 11 bits works, but 10 is enough

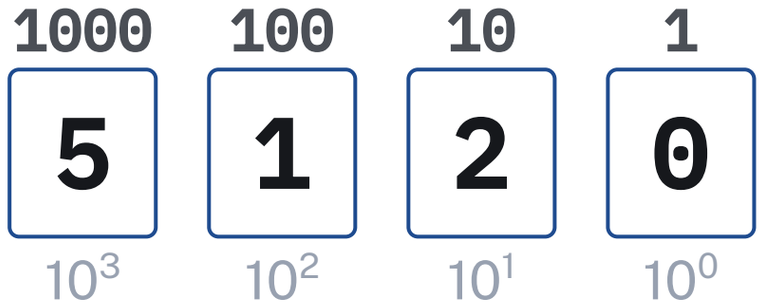
Part 2 of 3

Binary numbers

Binary is base 2: the only digits are 0 and 1

Each place is worth twice the place to its right

Place value, the way you learned it

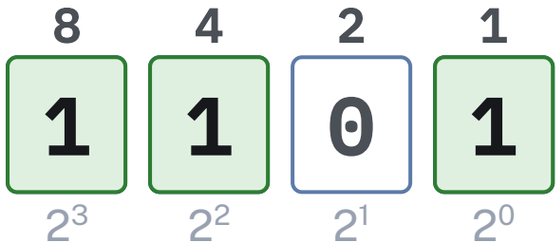


$$5 \times 1000 + 1 \times 100 + 2 \times 10 + 0 \times 1$$

$$5 \times 10^3 + 1 \times 10^2 + 2 \times 10^1 + 0 \times 10^0$$

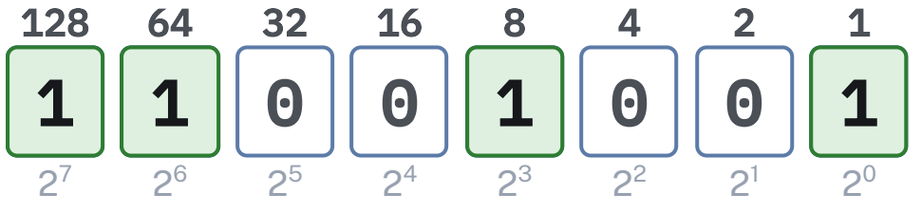
- 5 thousands, 1 hundred, 2 tens, 0 ones
- Base 10 has ten digits: 0 to 9
- Each place is worth 10 times the place to its right

Place value in base 2



Each place is a power of 2

8 + 4 + 1 = 13

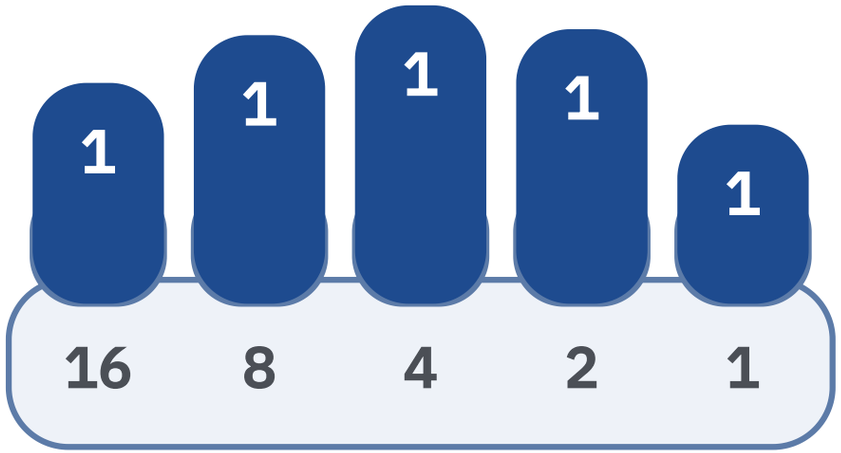


128 + 64 + 8 + 1 = 201

- Add up the places that hold a 1
- Write 1101_2 when the base isn't clear
  No subscript means base 10

Count on one hand

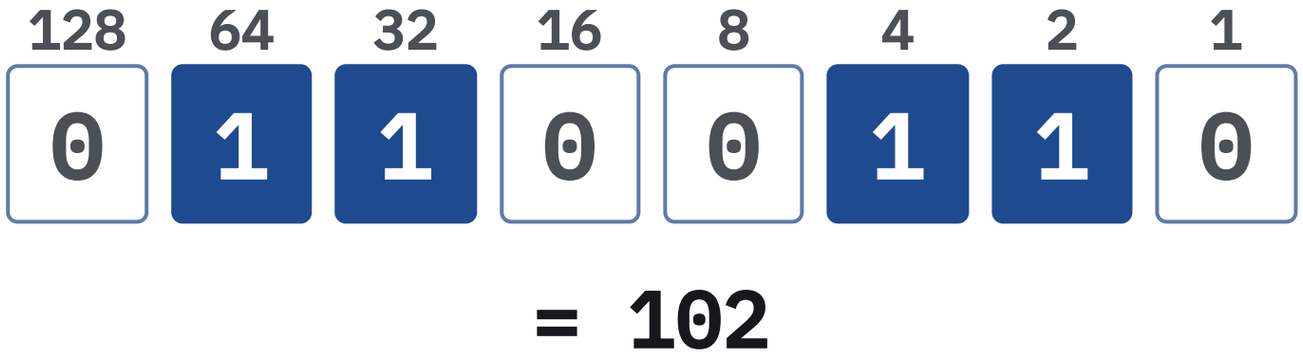
OPTIONAL





- Hold up your left hand, palm toward you
  A raised finger is a 1
- Show 5, then 12, then 31
- One hand counts from 0 to 31:
  5 bits give $2^5 = 32$ patterns
- Two hands count to 1,023

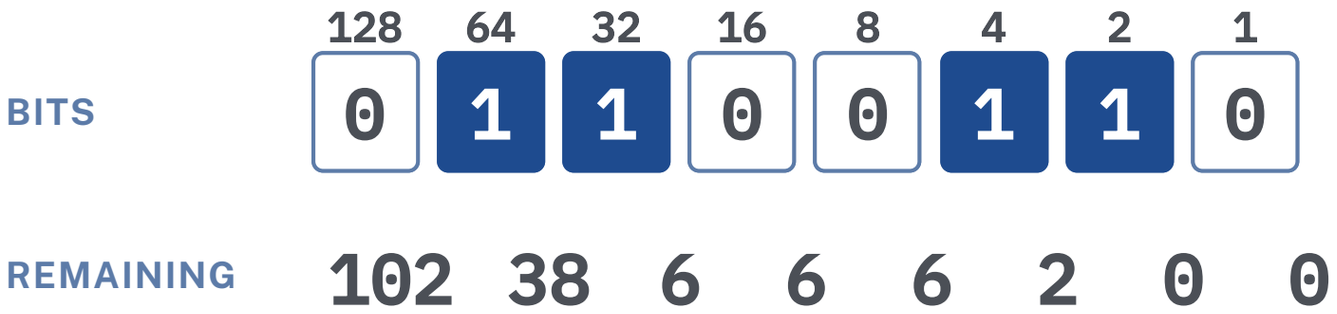
Flip the bits



- Click a bit to flip it
- All off: 0
  All on: 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
- 01100110 is 102, the red in our burgundy color at the beginning of class

Decimal to binary: subtract places

Write 102 in binary



- From the biggest place down: if it fits, write 1 and subtract

  102 is `01100110`, the bits from the start of class

Decimal to binary: divide by 2

Write 19 in binary

num	num // 2	num % 2
19	9	**1**
9	4	**1**
4	2	**0**
2	1	**0**
1	0	**1**



- `num % 2` is the last bit: 0 if even, 1 if odd
- `num // 2` is what's left after dropping that bit
- Keep going until `num` is 0
- Read the bits from the bottom up:
  19 is `10011` (16 + 2 + 1)
- You'll write this as code in Activity C

Get set up

1. Download day15.zip
2. Drag `day15.zip` to your `class` folder and extract it there
   Mac — double-click it. **Windows** — right-click it and choose "Extract All..."
   Once you see the `day15` folder, delete `day15.zip`
3. Launch Thonny and open `cs145/class/day15/day15_values.py`
   We'll use the shell to check our work today

Binary in Python

```
>>> 0b1101
13
>>> bin(13)
'0b1101'
>>> int("1101", 2)
13
>>> bin(102)
'0b1100110'
```

- `0b` in front of a number means binary
- `bin(num)` gives back a string, with `0b` in front
- `int(text, 2)` reads a string in base 2
- No leading zeros: 102 is `1100110`

Adding in binary

Add	Sum
0 + 0	0
0 + 1	1
1 + 1	10
1 + 1 + 1	11

```
  1  1  1  1
     1  0  1  1
+    0  1  1  1
-----------------
  1  0  0  1  0
```

- Same as adding by hand, right to left
- 1 + 1 is 2, which is 10 in binary
- Check: 11 + 7 = 18 = 16 + 2
- A sum of two 4-bit numbers can need a 5th bit

Multiplying in binary

OPTIONAL

```
   110
×  101
------
   110
  0000
 11000
+
------
 11110
```

- Same as long multiplication
- A digit of 1: copy 110, shifted left
- A digit of 0: all zeros
- Add the rows
- Check: 6 × 5 = 30 = 16 + 8 + 4 + 2

Activity A

On the worksheet, do problems 1–3 by hand

Then check your answers in the Thonny shell

PATTERNS

Binary to decimal	`int("1011", 2)`
Decimal to binary	`bin(19)`
Add binary numbers	`bin(0b1011 + 0b111)`

Activity A: answers

1a. `10110`	**22**	16 + 4 + 2
1b. `111000`	**56**	32 + 16 + 8
1c. `1000001`	**65**	64 + 1
2a. 37	**100101**	32 + 4 + 1
2b. 100	**1100100**	64 + 32 + 4
3. `1101 + 0111`	**10100**	13 + 7 = 20 = 16 + 4

Part 3 of 3

Hexadecimal

Hexadecimal (hex) is base 16: digits 0 to 9, then A to F

One hex digit stands for exactly 4 bits

Sixteen digits

Decimal	Binary	Hex
0	0000	**0**
1	0001	**1**
2	0010	**2**
3	0011	**3**
4	0100	**4**
5	0101	**5**
6	0110	**6**
7	0111	**7**

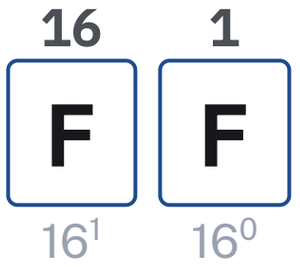
Decimal	Binary	Hex
8	1000	**8**
9	1001	**9**
10	1010	**A**
11	1011	**B**
12	1100	**C**
13	1101	**D**
14	1110	**E**
15	1111	**F**

- After 9 come the letters A to F
  A is 10, B is 11, ..., F is 15
- Four bits have 16 patterns,
  one for each hex digit
- So one nibble is one hex digit

Reading hex



$2 \times 16 + 10 \times 1 = 42$



$15 \times 16 + 15 \times 1 = 255$

- Each place is worth 16 times the place to its right
- **A** stands for 10
- Two hex digits go from **00** to **FF**: 0 to 255, one byte
- The next place is $16^2 = 256$:
  $\mathbf{100}_{16} = 256_{10}$
- No calculator needed: the worksheet's hex table lists
  0×16 to 15×16, and so will the quiz reference sheet

Decimal to hex: divide by 16

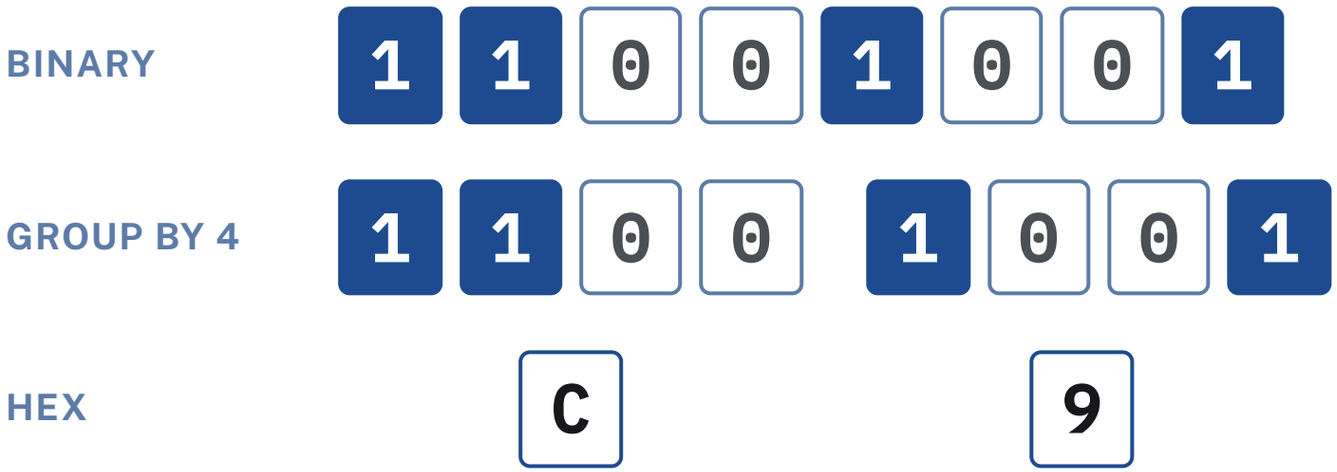
Write 981 in hex

num	num // 16	num % 16	Hex
981	61	5	**5**
61	3	13	**D**
3	0	3	**3**



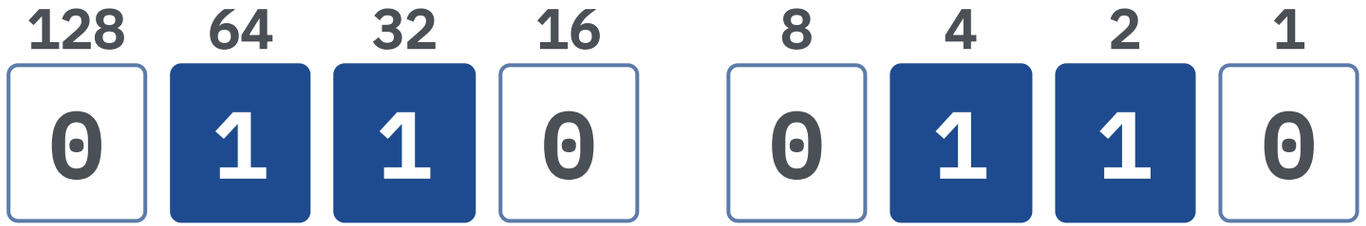
- Same steps as dividing by 2, with 16
- `num % 16` is the last hex digit: 0 to 15
- 13 is `D`
- Read the digits from the bottom up: 981 is `3D5`
- Check: 3 × 256 + 13 × 16 + 5 = 981

Binary to hex: four bits at a time



- `1100` is 12, so `C`, and `1001` is 9
- `11001001` is `C9`: 12 × 16 + 9 = 201, as before
- Hex to binary works the same way backwards:
  `3F` is `0011 1111`
- Programmers like hex: it's short, and two digits are one byte

One byte, three ways



decimal 102
hex 66

- Each group of 4 bits is one hex digit
- Flip a bit and watch both readouts change

Back to our burgundy color



#660933

(102, 9, 51)

Hex	Binary	Value	Decimal
66	0110 0110	6 × 16 + 6	102
09	0000 1001	0 × 16 + 9	9
33	0011 0011	3 × 16 + 3	51

- Each pair of hex digits is one byte: one color amount
- A color is 3 bytes, 24 bits:
  2^{24} = 16,777,216 colors

Hex in Python

```
>>> 0x2a
42
>>> hex(255)
'0xff'
>>> int("ff", 16)
255
>>> int("66", 16)
102
```

- `0x` in front of a number means hex
- `hex(num)` gives back a string, with `0x` in front and lowercase letters
- `int(text, 16)` reads a string in base 16 Uppercase and lowercase both work
- `int("66", 16)` is 102: the red in our color

Activity B

On the worksheet, do problems 4–6 by hand

Then check your answers in the Thonny shell

PATTERNS

Hex to decimal	`int("2a", 16)`
Decimal to hex	`hex(42)`
Binary to hex	`hex(0b11001001)`
Hex to binary	`bin(0x3f)`

Activity B: answers

4a. `1F`	**`31`**	1 × 16 + 15
4b. `C8`	**`200`**	12 × 16 + 8
5a. `10110110`	**`B6`**	`1011 0110`
5b. `7E`	**`01111110`**	`0111 1110`
6a. `#FFA500`	**`(255, 165, 0)`**	A5 is 10 × 16 + 5
6b. (0, 128, 128)	**`#008080`**	128 is 8 × 16

Activity C

In `day15_values.py`, write `to_binary` and `hex_to_rgb`
If you have time, write `rgb_to_hex`

PATTERNS

Last bit of `num`

```
num % 2
```

Drop the last bit

```
num // 2
```

Repeat until `num` is 0

```
while num > 0:
```

Put a bit in front

```
result = str(bit) + result
```

Characters 1 and 2 of `code`

```
code[1:3]
```

Hex text to a number

```
int(text, 16)
```

Hex digits without `0x`

```
hex(num)[2:]
```

To do

- Review this slide deck and Notes 13
- Exam 3 (Lectures 10–14) is Friday 11/6 in lab
  See practice materials

Next time

Lecture 16 — representing data

Why is `3 * 0.1 == 0.3` *False? How is text stored?*