CS107, Lecture 2
Unix, C, Bits and Bytes, Integer Representations

Reading: Bryant & O'Hallaron, Ch. 2.2-23 (skim)

Ed Discussion: https://edstem.org/us/courses/46162/discussion/3504481

CS107 Topic 1

How can a computer represent integer numbers?

Why is answering this question important?

- Helps us understand the limitations of computer arithmetic (today and Monday)
- Shows us how to more efficiently perform arithmetic (Monday and Tuesday)
- Shows us how we can encode data more compactly and efficiently (Tuesday)

assign1: implement 3 programs that manipulate binary representations to (1) work around the limitations of arithmetic with addition, (2) simulate an evolving colony of cells, and (3) print Unicode text to the terminal.

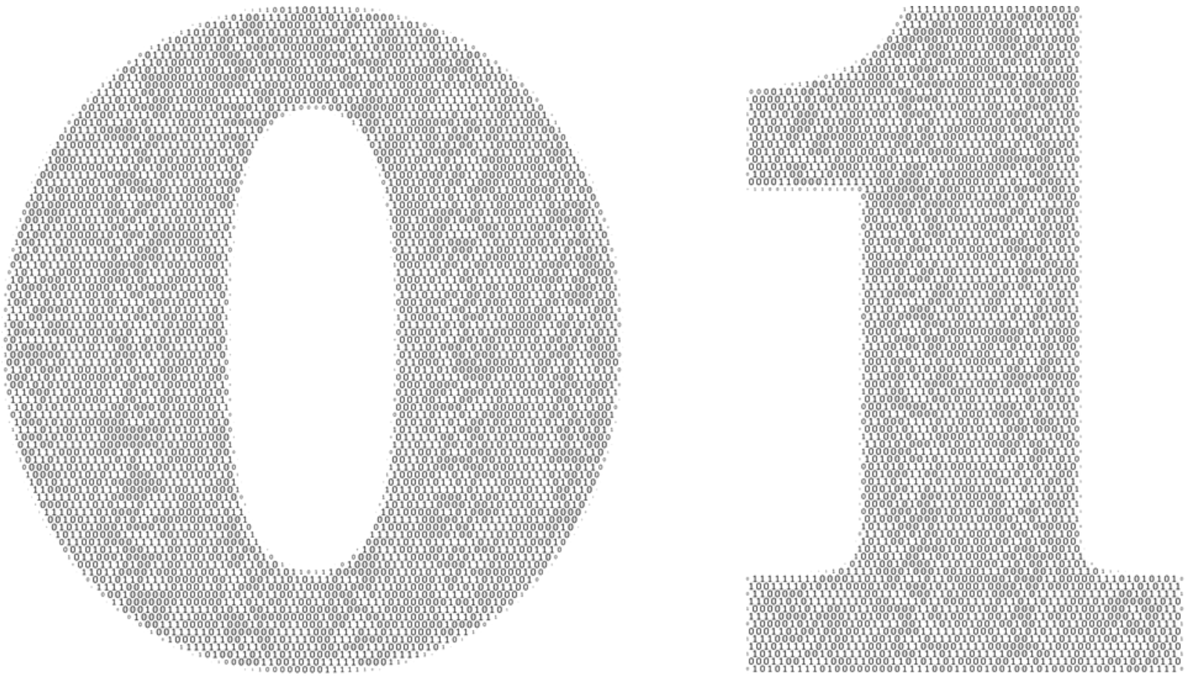
Learning Goals

- Learn about the binary and hexadecimal number systems and how to convert between number systems
- Understand how positive and negative numbers are represented in binary
- Learn about overflow, why it occurs, and how overflow might impact program execution

Demo: Unexpected Behavior

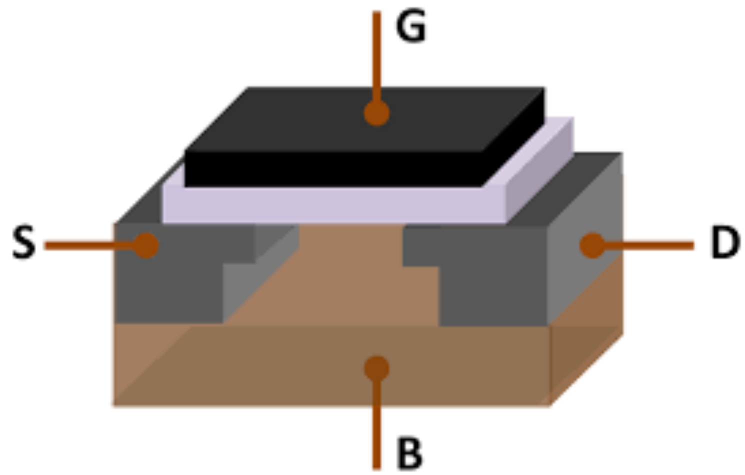
```
cp -r /afs/ir/class/cs107/lecture-code/lect2 .
```

Bits



Bits

Computers are built around the idea of two states: "on" and "off". Transistors implement this in hardware, and bits represent this in software.



One Bit At A Time

- We can combine bits, as with base-10 numbers, to represent more data.
  8 bits = 1 byte.
- Computer memory is just a large array of bytes. It is **byte-addressable**. meaning you can't address a bit in isolation, only a full byte.
- Computers still fundamentally operate using bits. We have just gotten more creative about how to represent data.
  - Images
  - Audio
  - Video
  - Text
  - And more…

Base 10

5 9 3 4

digits 0 – 9

(or rather, 0 through base – 1)

Base 10

5 9 3 4

thousands hundreds tens ones

= **5** * 1000 + **9** * 100 + **3** * 10 + **4** * 1

Base 10

5 9 3 4

↑ ↑ ↑ ↑

10^3 10^2 10^1 10^0

Base 10

5 9 3 4

10^X: 3 2 1 0

Base 2

	1	0	1	1
2^X:	3	2	1	0

digits 0 – 1
(or rather, 0 through base – 1)

Base 2

1 0 1 1

2^3 2^2 2^1 2^0

Base 2

Most significant bit (MSB) Least significant bit (LSB)

1 0 1 1

eights fours twos ones

$= \mathbf{1} * 8 + \mathbf{0} * 4 + \mathbf{1} * 2 + \mathbf{1} * 1 = 11_{10}$

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 ≤ 6? **2^2=4**
  - Now, what is the largest power of 2 ≤ 6 – 2^2? **2^1=2**
  - 6 – 2^2 – 2^1 = 0!

0	1	1	0
2^3	2^2	2^1	2^0

= **0***8 + **1***4 + **1***2 + **0***1 = 6

Practice: Base 2 to Base 10

What is the base-2 value 1010 in base-10?

a) **20**
b) **101**
c) **10**
d) **5**
e) **Other**

Practice: Base 10 to Base 2

What is the base-10 value 14 in base 2?

a) **1111**

b) **1110**

c) **1010**

d) **Other**

Byte Values

What are the minimum and maximum base-10 values a single byte (8 bits) can represent?

minimum = 0 **maximum = 255**

11111111

2^x: 7 6 5 4 3 2 1 0

- **Strategy 1:** $1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 255$
- **Strategy 2:** $2^8 – 1 = 255$

Multiplying by Base

$$1450 \times 10 = 1450\underline{\mathbf{0}}$$

$$1100_2 \times 10_2 = 1100\underline{\mathbf{0}}$$

Key Idea: appending 0 to the end effectively multiplies by the base!

Dividing by Base

$$1450 / 10 = 145$$

$$1100_2 / 10_2 = 110$$

Key Idea: chomping off 0 from the end divides by the base!

Question Break

Hexadecimal

When working with bits, oftentimes we have large numbers with 32 or 64 bits.

- Instead, we'll generally encode numbers in **base-16**, or **hexadecimal**, instead.

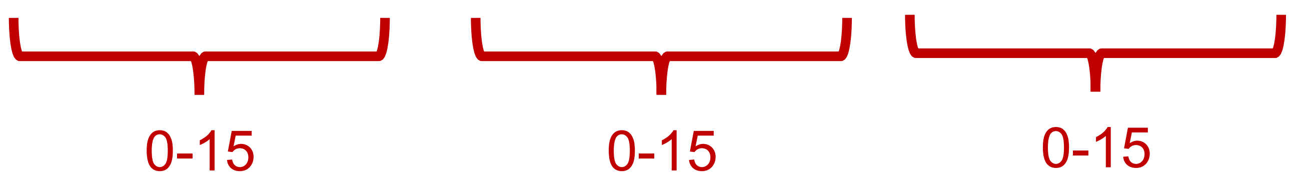
0110 1010 0011

0-15 0-15 0-15

Hexadecimal

When working with bits, oftentimes we have large numbers with 32 or 64 bits.

- Instead, we'll generally encode numbers in **base-16**, or **hexadecimal**, instead**.**



Each quartet of bits can be rewritten as a single digit, in **base-16**!

Hexadecimal

Hexadecimal is **base-16**, so we need digits for 1-15. How?

0 1 2 3 4 5 6 7 8 9

Hexadecimal

- If it's not clear from context, we can explicitly identify numbers as hexadecimal by prefixing them with **`0x`** and identify numbers as binary by prefixing with **`0b`**.
- e.g., **`0xf5`** is **`0b11110101`**

0x f 5

1111 0101

Hexadecimal

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

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

Practice: Hexadecimal to Binary

What is **`0x173A`** in binary?

Hexadecimal	**1**	**7**	**3**	**A**
Binary	**0001**	**0111**	**0011**	**1010**

Practice: Hexadecimal to Binary

What is **`0b1111001010`** in hexadecimal? (*Hint: start from the right)*

`Binary`	**`11`**	**`1100`**	**`1010`**
`Hexadecimal`	**`3`**	**`C`**	**`A`**

Hexadecimal: Quirky but concise

- Let's take a single byte (8 bits):

`165`	base-10: Human-readable, but cannot easily interpret on/off bits
`0b10100101`	base-2: Computers love this, but normal humans do not.
`0xa5`	base-16: Easy to convert to base-2, More easily digested format (fun fact: a half-byte is called a nibble.. tee hee)

Number Representations

- **Unsigned Integers**: positive integers and 0. (e.g., 0, 1, 2, … 99999…)
- **Signed Integers:** negative, positive integers and 0. (e.g., …-2, -1, 0, 1,… 9999…)

- **Floating Point Numbers:** real numbers. (e,g. 0.1, -12.2, 1.5×10^{12})

→ **Look up IEEE floating point if you're interested!**

Number Representations

C Declaration	Size (Bytes)
`int`	4
`double`	8
`float`	4
`char`	1
`char *`	8
`short`	2
`long`	8

Back When Jerry Learned C

C Declaration	Size (Bytes)
int	4
double	8
float	4
char	1
char *	4
short	2
long	4

Transitioning To Larger Data Types

- **Early 2000s:** most computers were **32-bit.** This means that pointers were **4 bytes (32 bits).**
- 32-bit pointers store a memory address from 0 to 2^{32} - 1, equaling **2^{32} bytes of addressable memory**. This equals **4 gigabytes**, meaning that 32-bit computers could address *at most* **4GB** of memory!
- Most computers now are to **64-bit.** Many data types got more memory, and pointers in programs were now **64 bits.**
- 64-bit pointers can distinguish between addresses 0 to 2^{64} - 1, equaling **2^{64} bytes of addressable memory.** This equals **16 exabytes**, meaning that 64-bit computers could address up to **16 * 1024 * 1024 * 1024 GB** of memory!

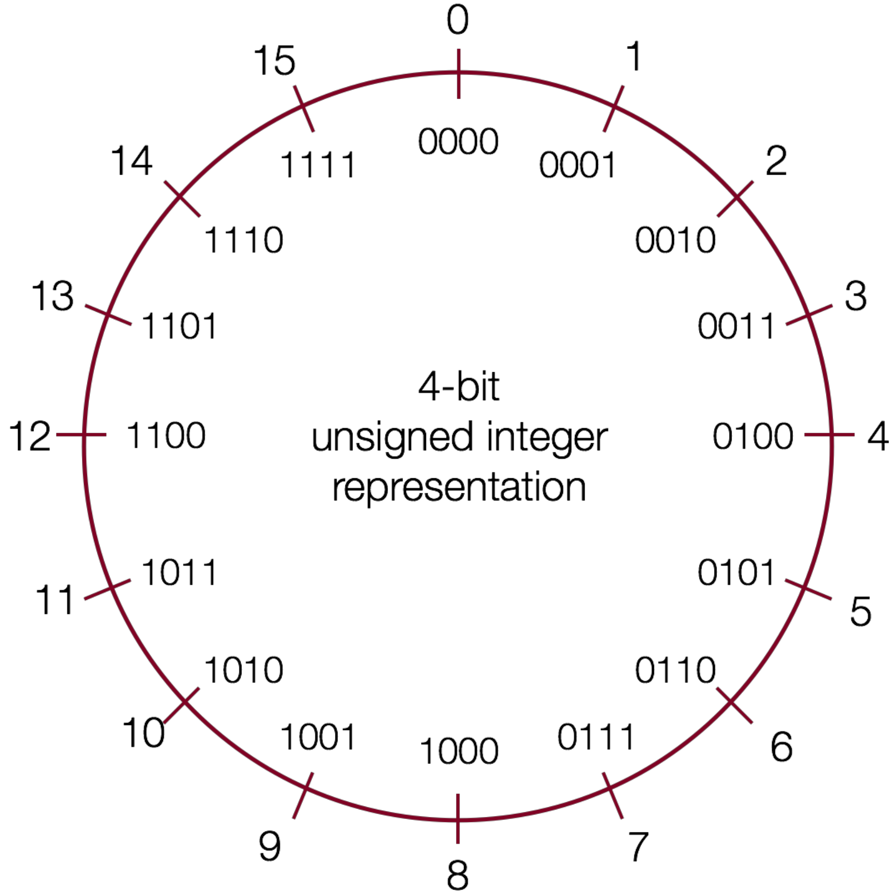
Unsigned Integers

- An **unsigned** integer is either 0 or some positive integer (no negatives).
- We have already discussed the conversion between decimal and binary. Examples:

```
0b0001 = 1
0b0101 = 5
0b1011 = 11
0b1111 = 15
```

- The range of an unsigned number is 0 → 2^w - 1, where w is the number of bits. e.g., a 32-bit integer can represent 0 to $2^{32} – 1$ (4,294,967,295).

Unsigned Integers



Question Break

Signed Integers

A **signed** integer is a negative integer, 0, or a positive integer.

- *Problem:* How can we represent negative *and* positive numbers in binary?

Idea: let the **most** significant bit represent sign and let the others represent magnitude.

Sign Magnitude Representation: 4-bit

0110

positive 6

1011

negative 3

Sign Magnitude Representation: 4-bit

0000

positive 0

1000

negative 0

Sign Magnitude Representation: 4-bit

1 000 = -0	0 000 = 0
1 001 = -1	0 001 = 1
1 010 = -2	0 010 = 2
1 011 = -3	0 011 = 3
1 100 = -4	0 100 = 4
1 101 = -5	0 101 = 5
1 110 = -6	0 110 = 6
1 111 = -7	0 111 = 7

We're only representing 15 different values via 16 different patterns.
#sadness

Sign Magnitude Representation

- **Pro:** easy to represent, and easy to convert to and from decimal.
- **Con:** +/-0 is 🤯
- **Con:** we lose a bit that could be used to represent more numbers
- **Con:** arithmetic is tricky: we need to find the sign, perhaps subtract (borrow and carry, etc.), maybe change the sign, maybe not. This complicates how hardware supports something as fundamental as addition.

Can we do better?

A Better Idea

Ideally, standard binary addition should work whether numbers are positive or negative.

```
  0101
+ ????
------
  0000
```

A Better Idea

Ideally, binary addition would work whether the numbers are positive or negative.

$$
\begin{array}{r}
0101 \\
+\ 1011 \\
\hline
0000
\end{array}
$$

A Better Idea

Ideally, binary addition would work whether the numbers are positive or negative.

$$
\begin{array}{r}
0011 \\
+\ ???? \\
\hline
0000
\end{array}
$$

A Better Idea

Ideally, binary addition would work whether the numbers are positive or negative.

$$\begin{array}{r} 0011 \\ +\ 1101 \\ \hline 0000 \end{array}$$

A Better Idea

Ideally, binary addition would work whether the numbers are positive or negative.

```
  0000
+ ????
  ----
  0000
```

A Better Idea

Ideally, binary addition would work whether the numbers are positive or negative.

$$\begin{array}{r} 0000 \\ +\,{\color{red}0000} \\ \hline 0000 \end{array}$$

There Seems To Be A Pattern

0101	0011	0000
+ 1011	+ 1101	+ 0000
0000	0000	0000

The negative number is the positive number **bitwise inverted, plus one!**

There Seems To Be A Pattern

A binary number plus its inverse is all 1s.

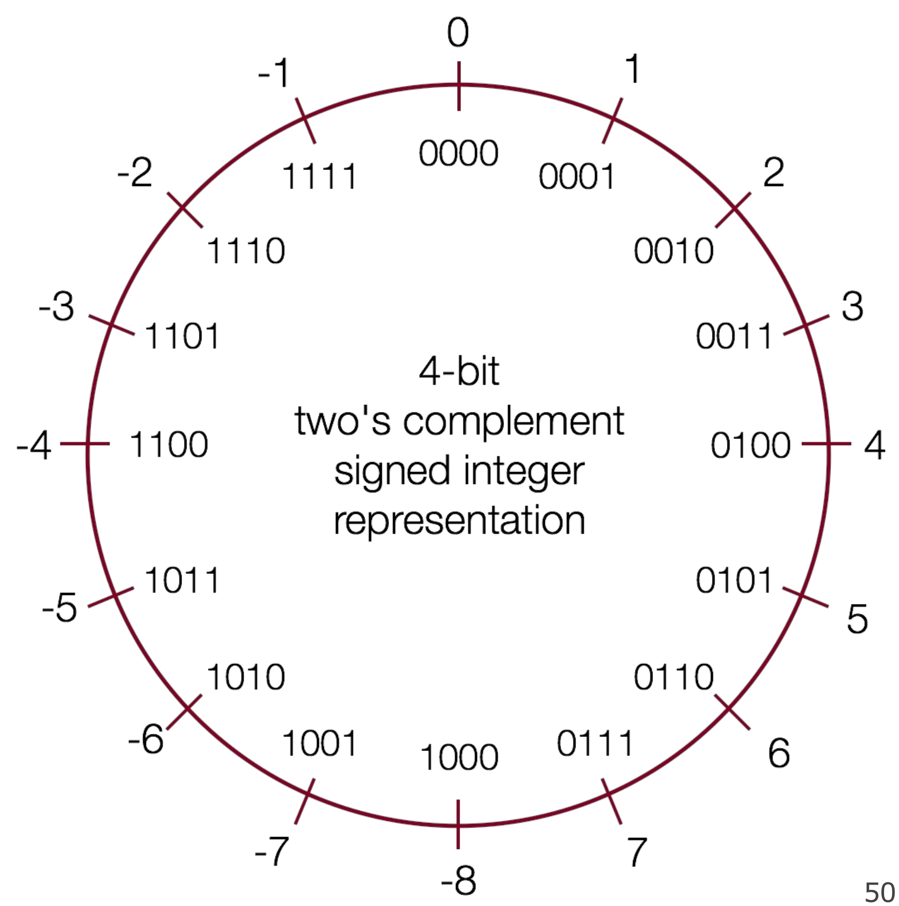
```
  0101
+ 1010
------
  1111
```

Add 1 to this to carry over all 1s and get 0!

```
  1111
+ 0001
------
  0000
```

Two’s Complement

- With **two’s complement**, we represent a positive number as **itself**, and its negative equivalent as the **two’s complement of itself.**
- The **two’s complement** of a number is the binary digits inverted, plus 1.
- This works to convert from positive to negative, **and** back from negative to positive!



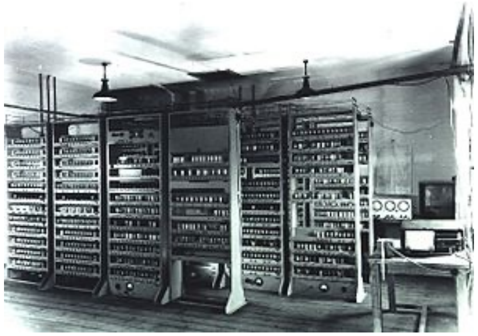
History: Two's complement

- Binary representation was first proposed by John von Neumann in *First Draft of a Report on the EDVAC* (1945).
  - That same year, he also invented the merge sort algorithm
- Many early computers used either sign-magnitude or one's complement.

```
+7    0b0000 0111
-7    0b1111 1000
```

8-bit one's complement

- The System/360, developed by IBM in 1964, was widely popular (it had 1024KB memory!) and established two's complement as the dominant binary representation of integers.

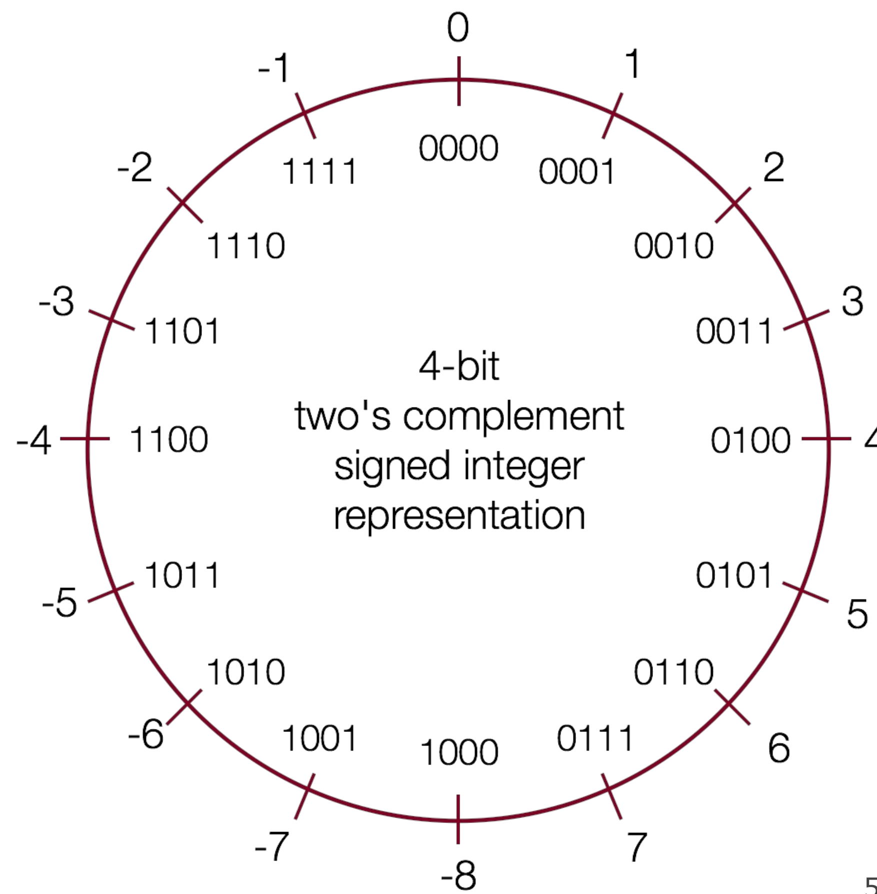


EDSAC (1949)

System/360 (1964)

Two's Complement

- **Con:** more difficult to represent, and difficult to convert to and from decimal, between positive and negative.
- **Pro:** only 1 representation for 0! 😍
- **Pro:** the most significant bit still indicates the sign of a number.
- **Pro:** addition works for any combination of positive and negative!



Two's Complement

Adding two numbers is just that: adding! And there is no special case needed for negative numbers. e.g., what is 2 + -5?

	0010	2
+	1011	-5
	1101	-3

Two's Complement

Subtracting two numbers is just performing the two's complement on the second of them and then adding instead of subtracting, e.g., 4 – 5 = -1.

0100	4		0100	4
- 0101	5	→	+ 1011	-5
			1111	-1