CS107 Lecture 3

Bits and Bytes; Integer Representations

reading:

Bryant & O'Hallaron, Ch. 2.2-2.3

CS107 Topic 1: How can a computer represent integer numbers?

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)
- Shows us how to more efficiently perform arithmetic (next time)
- Shows us how we can encode data more compactly and efficiently (next time)

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

- Understand the limitations of computer arithmetic and how that can impact our programs, such as with overflow
- Understand how positive and negative numbers stored in **ints, longs**, etc. are represented in binary
- Learn about the binary and hexadecimal number systems and how to convert between number systems

Delta/Comair Airline Holiday Chaos

Case study: Comair/Delta airline had to cancel thousands of flights days before Christmas due to a system malfunction. An unusually high number of crew reassignments caused a bug in the system. What happened?

Demo: Unexpected Behavior

```
cp -r /afs/ir/class/cs107/lecture-code/lect3 .
```

Lecture Plan

- Integer Representations
- Bits and Bytes
- Hexadecimal
- Unsigned Integers
- Signed Integers
- Overflow

Lecture Plan

- **Integer Representations**
- Bits and Bytes
- Hexadecimal
- Unsigned Integers
- Signed Integers
- Overflow

Number Representations

- Numeric types are generally a fixed size (e.g. **int** is 4 bytes). This means there is a limit to the range of numbers they can store.
- **Overflow** occurs when we exceed the maximum value or go below the minimum value of what a numeric type can store. It can cause unintended bugs!

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

Number Representations

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

Lecture Plan

- Integer Representations
- **Bits and Bytes**
- Hexadecimal
- Unsigned Integers
- Signed Integers
- Overflow

One Bit At a Time

- A bit is 0 or 1
- Computers are built around the idea of two states: “on” and “off”. Bits represent this idea in software! (transistors represent this in hardware).
- We can combine bits, like 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*; you can’t address (store location of) a bit; only a byte.
- Computers fundamentally operate on bits; but we creatively represent different data as bits!
  - Images
  - Video
  - Text
  - And more…

Base 10

5 9 3 4

Digits 0-9 (*0* to *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 (*0* to *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

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

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**

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 ≤ 6? $\mathbf{2^2=4}$
  - Now, what is the largest power of 2 ≤ 6 – 2^2? $\mathbf{2^1=2}$
  - 6 – 2^2 – 2^1 = 0!

$$\underset{2^3}{\underline{0}} \quad \underset{2^2}{\underline{1}} \quad \underset{2^1}{\underline{1}} \quad \underset{2^0}{\underline{0}}$$

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

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

- 1111
- 1110
- 1010
- Other

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

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

1111 — 0%

1110 — 0%

1010 — 0%

Other — 0%

Byte Values

What is the minimum and maximum base-10 value a single byte (8 bits) can store? **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 x 10 = 1450**<u>0</u>**

1100_2 x 2 = 1100**<u>0</u>**

Key Idea: inserting 0 at the end multiplies by the base!

Dividing by Base

$$1450 / 10 = 145$$

$$1100_2 / 2 = 110$$

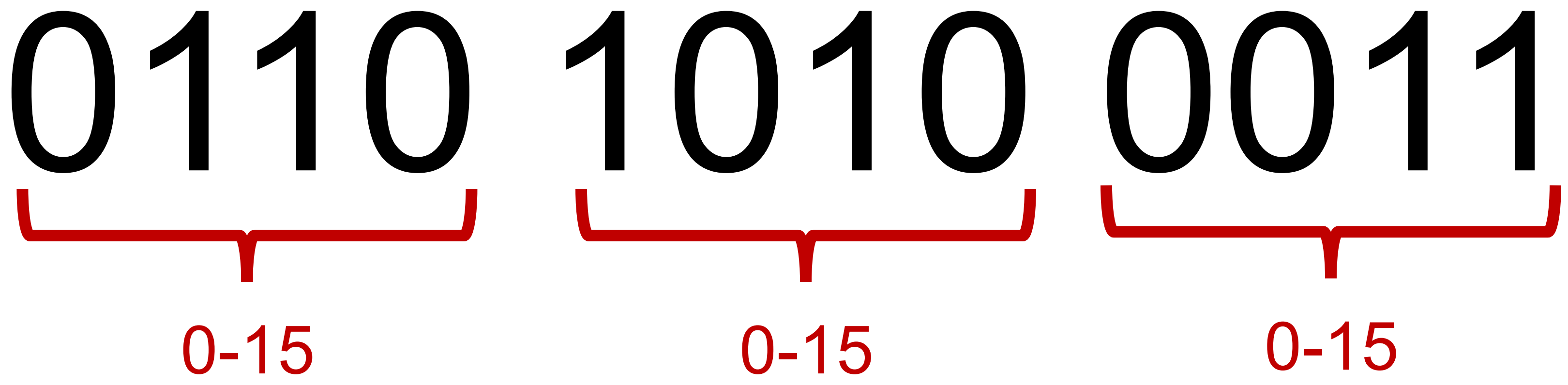
Key Idea: removing 0 at the end divides by the base!

Lecture Plan

- Integer Representations
- Bits and Bytes
- **Hexadecimal**
- Unsigned Integers
- Signed Integers
- Overflow

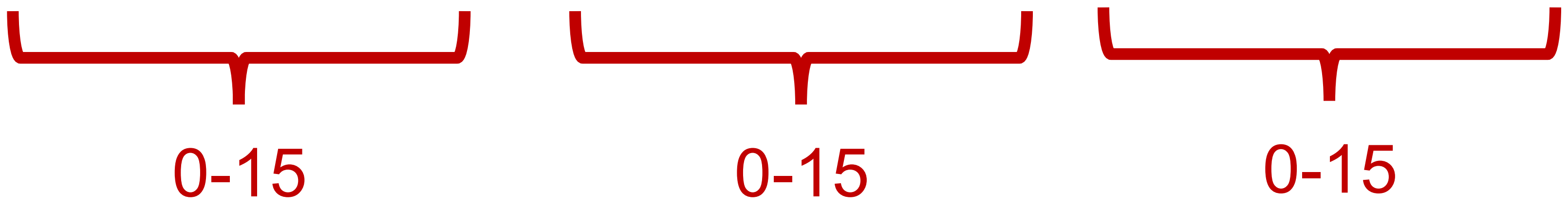
Hexadecimal

- When working with bits, oftentimes we have large numbers with 32 or 64 bits.
- Instead, we'll represent bits in *base-16 instead;* this is called **hexadecimal**.



Hexadecimal

- When working with bits, oftentimes we have large numbers with 32 or 64 bits.
- Instead, we'll represent bits in *base-16 instead;* this is called **hexadecimal**.



Each is a base-16 digit!

Hexadecimal

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

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
										10	11	12	13	14	15

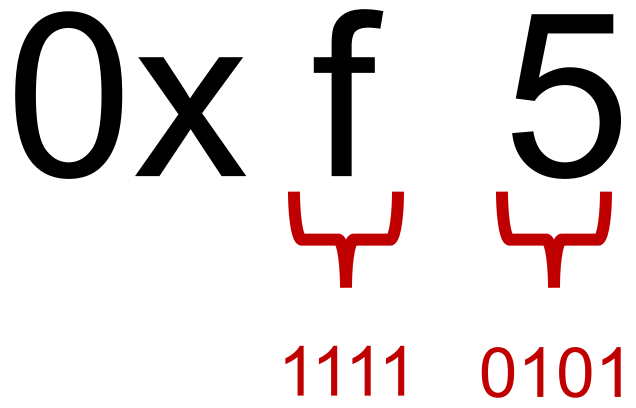
Hexadecimal

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

Hexadecimal

- We distinguish hexadecimal numbers by prefixing them with **`0x`**, and binary numbers with **`0b`**.
- E.g. **`0xf5`** is **`0b11110101`**



Practice: Hexadecimal to Binary

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

Hexadecimal	1	7	3	A
Binary	0001	0111	0011	1010

Practice: Binary to Hexadecimal

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

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

Hexadecimal: It’s funky but concise

Let’s take a byte (8 bits):

`165`	Base-10: Human-readable, but cannot easily interpret on/off bits
`0b10100101`	Base-2: Yes, computers use this, but not human-readable
`0xa5`	Base-16: Easy to convert to Base-2, More “portable” as a human-readable format (fun fact: a half-byte is called a nibble or nybble)

Lecture Plan

- Integer Representations
- Bits and Bytes
- Hexadecimal
- **Unsigned Integers**
- Signed Integers
- Overflow

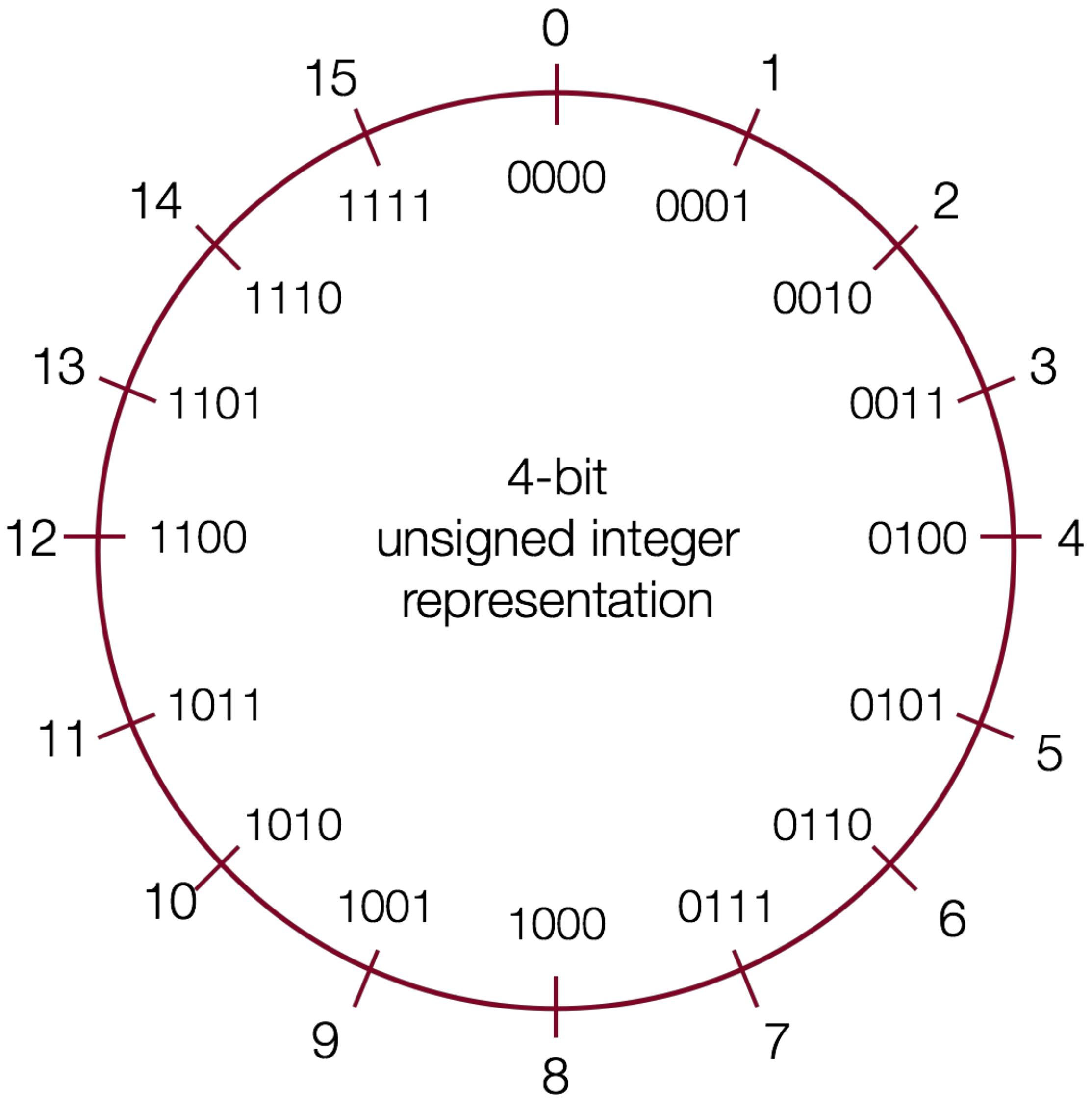
Unsigned Integers

- An **unsigned** integer is 0 or a positive integer (no negatives).
- We have already discussed converting between decimal and binary, which is a nice 1:1 relationship. 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).

Example: 4-bit Unsigned Integer



From Unsigned to Signed

A **signed** integer is a negative, 0, or positive integer. How can we represent both negative *and* positive numbers in binary?

Lecture Plan

- Integer Representations
- Bits and Bytes
- Hexadecimal
- Unsigned Integers
- **Signed Integers**
- Overflow

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?

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

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

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

```
  0101
+ 1011
------
  0000
```

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

```
  0011
+ ????
------
  0000
```

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

```
  0011
+ 1101
------
  0000
```

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

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

A Better Idea

- Ideally, binary addition would *just work* **regardless** of whether the number is positive or negative.

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

A Better Idea

Decimal	Positive	Negative
0	0000	0000
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001

Decimal	Positive	Negative
8	1000	1000
9	1001 (same as -7!)	NA
10	1010 (same as -6!)	NA
11	1011 (same as -5!)	NA
12	1100 (same as -4!)	NA
13	1101 (same as -3!)	NA
14	1110 (same as -2!)	NA
15	1111 (same as -1!)	NA

There Seems Like a Pattern Here…

```
  0101    0011    0000
+ 1011  + 1101  + 0000
------  ------  ------
  0000    0000    0000
```

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

There Seems Like a Pattern Here...

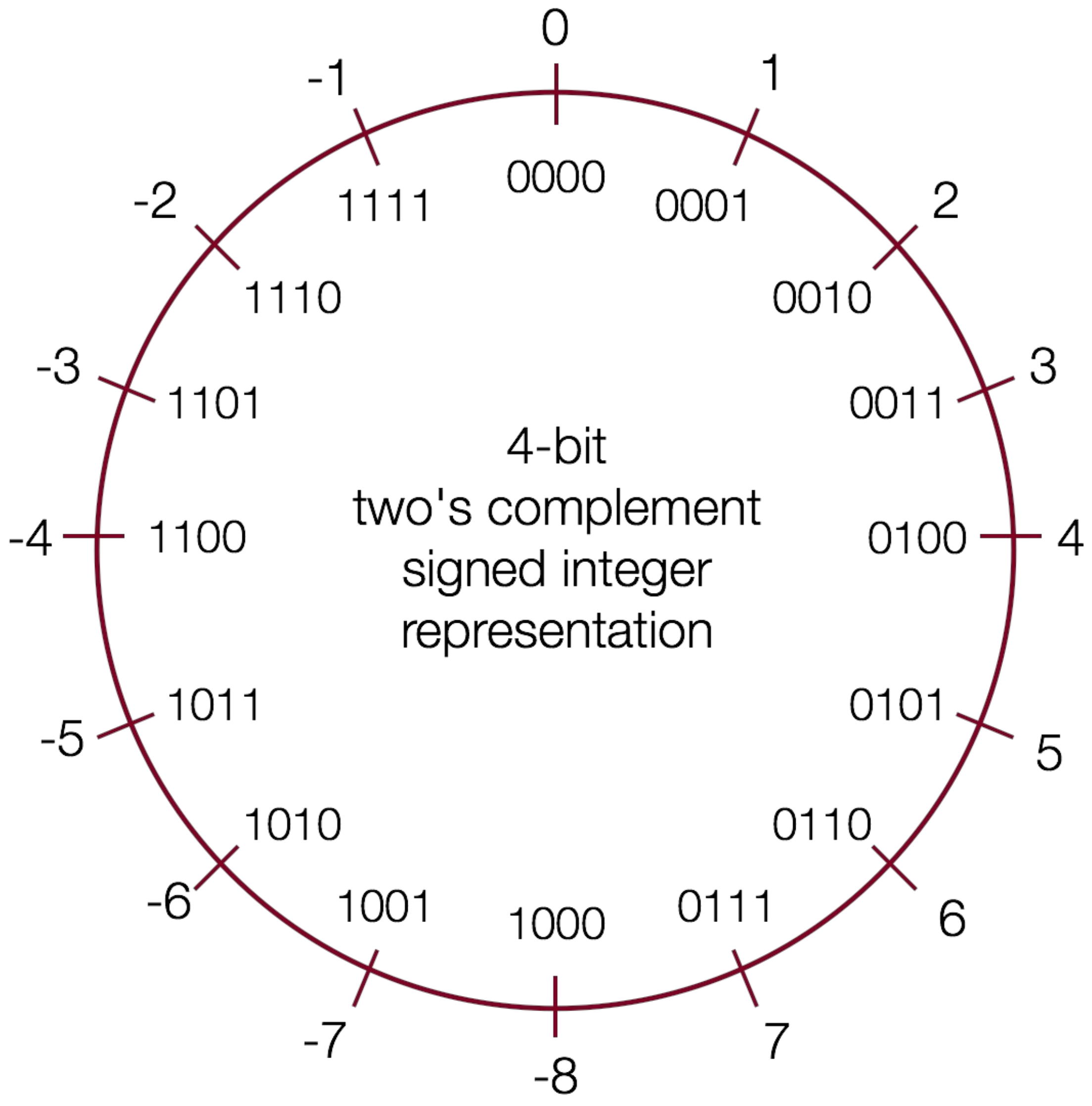
A binary number plus its inverse is all 1s.

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

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

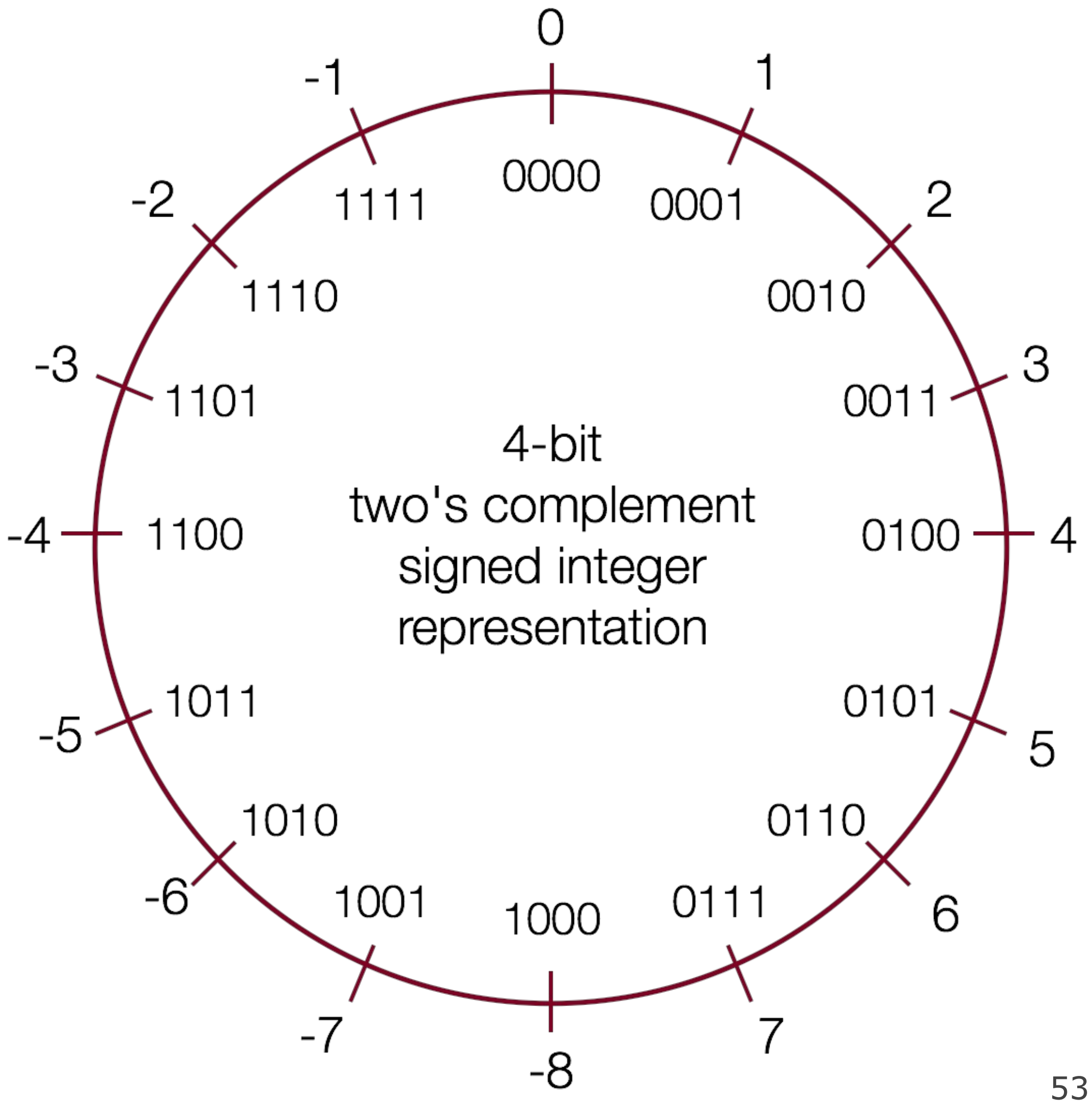
```
  1111
+ 0001
------
  0000
```

Two's Complement



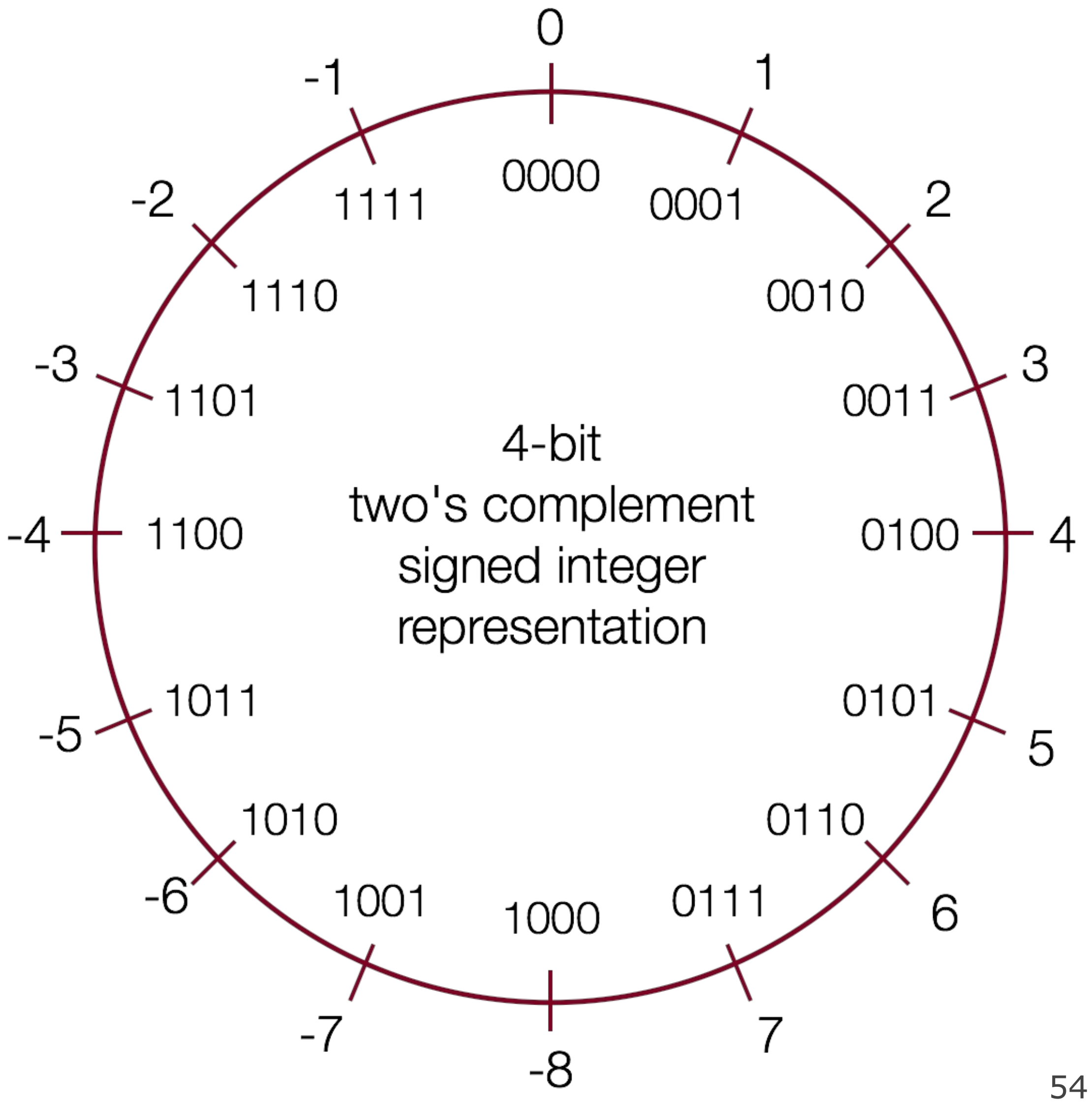
Two's Complement

- In **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!



Two’s Complement

- **Con:** takes some steps to convert between positive and negative, can’t eyeball a negative number and immediately know what value it is.
- **Pro:** addition works for any combination of positive and negative!
- **Pro:** the most significant bit indicates the sign of a number.



Two's Complement

Adding two numbers is just…adding! There is no special case needed for negatives. E.g. what is 2 + -5?

	0010	2
+	1011	-5
	1101	-3

Lecture Plan

- Integer Representations
- Bits and Bytes
- Hexadecimal
- Unsigned Integers
- Signed Integers
- **Overflow**

Overflow

If you exceed the **maximum** value of your bit representation, you *wrap around* or *overflow* back to the **smallest** bit representation. E.g. with 4 bits:

```
0b1111 + 0b1 =  0b0000
0b1111 + 0b10 = 0b0001
```

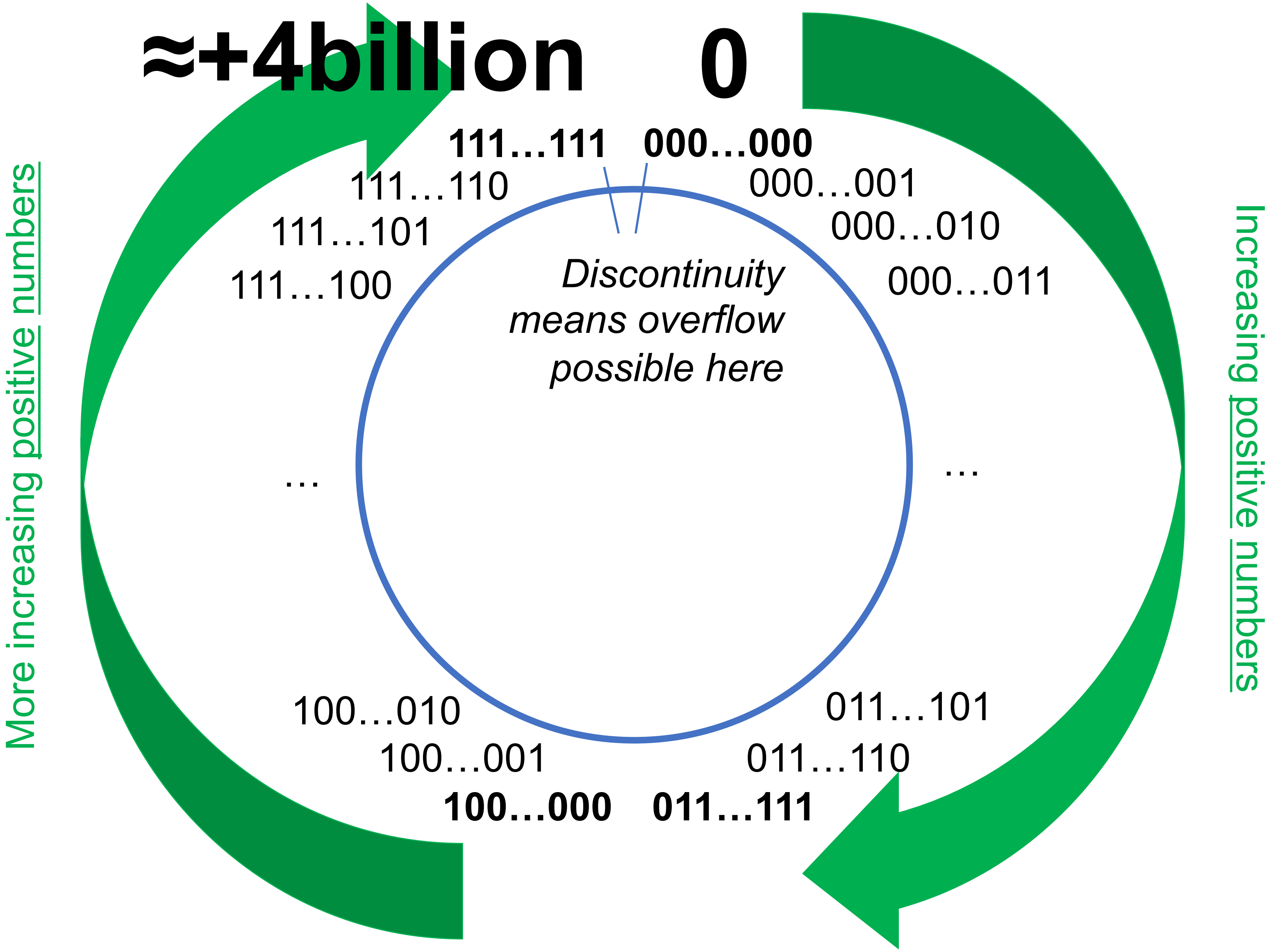
If you go below the **minimum** value of your bit representation, you *wrap around* or *overflow* back to the **largest** bit representation. E.g. with 4 bits:

```
0b0000 - 0b1 =  0b1111
0b0000 - 0b10 = 0b1110
```

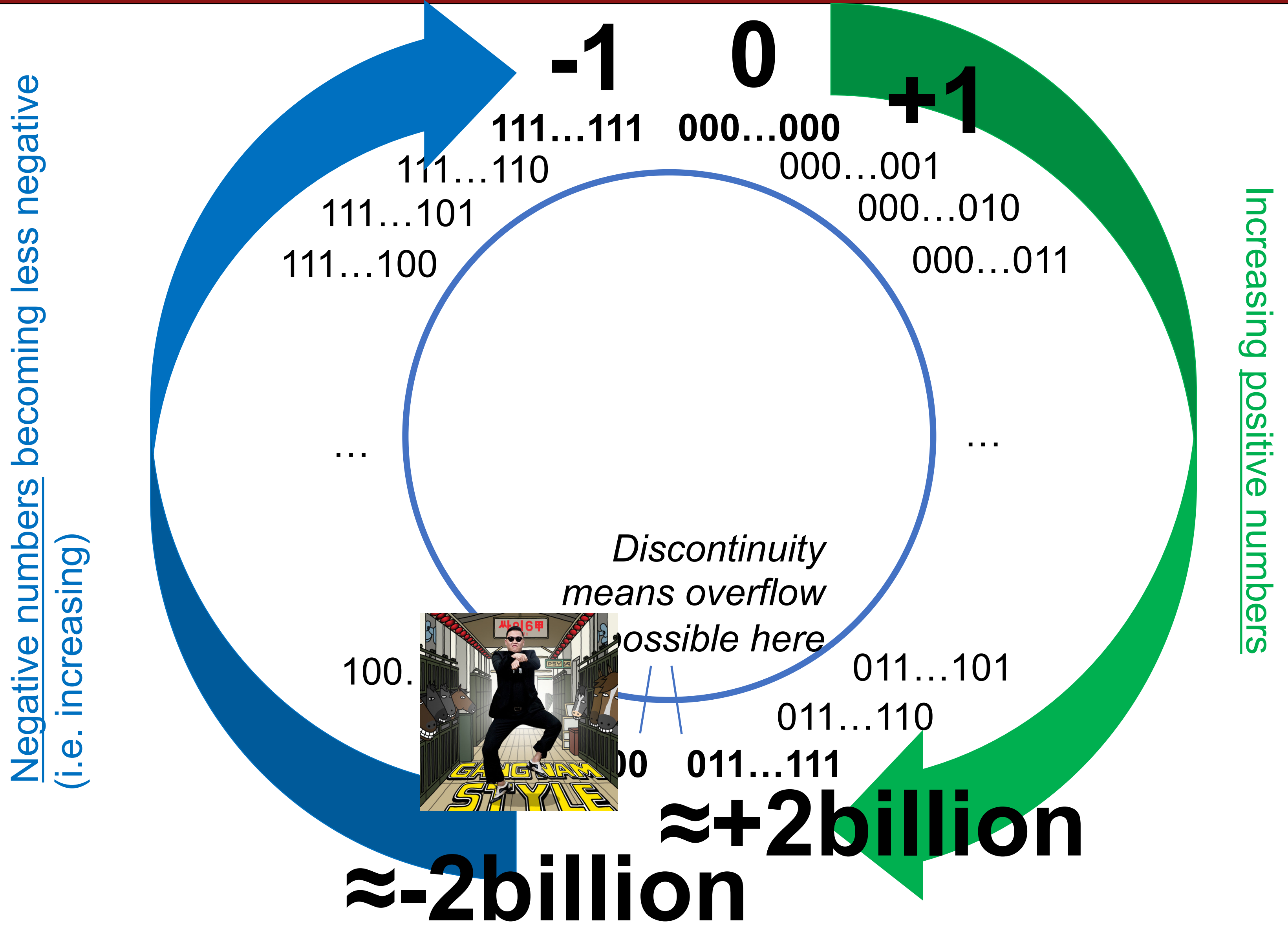
Min and Max Integer Values

Type	Size (Bytes)	Minimum	Maximum
char	1	-128	127
unsigned char	1	0	255
short	2	-32768	32767
unsigned short	2	0	65535
int	4	-2147483648	2147483647
unsigned int	4	0	4294967295
long	8	-9223372036854775808	9223372036854775807
unsigned long	8	0	18446744073709551615

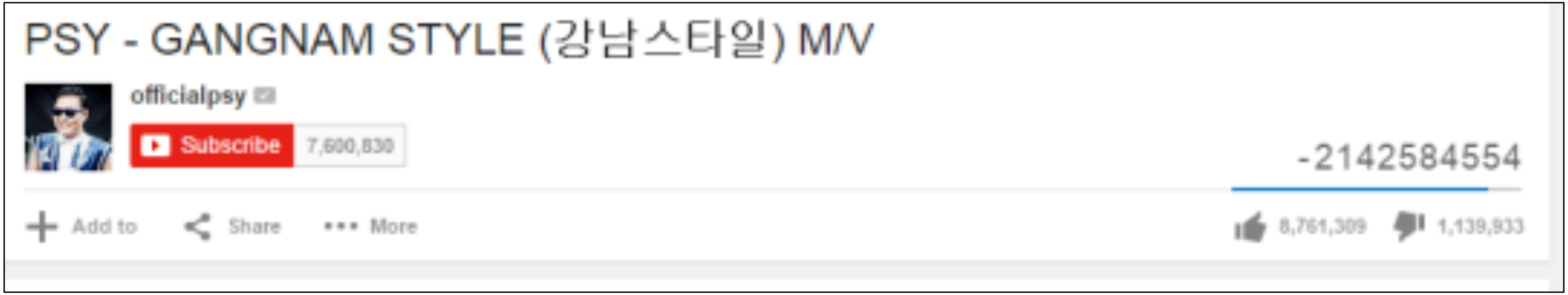
Unsigned Integers



Signed Numbers



Overflow In Practice: PSY



YouTube: “We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!” [link]

“We saw this coming a couple months ago and updated our systems to prepare for it” [link]

Overflow In Practice: Timestamps

Many systems store timestamps as **the number of seconds since Jan. 1, 1970** in a **signed 32-bit integer**.

- **Problem:** the latest timestamp that can be represented this way is 3:14:07 UTC on Jan. 13 2038!

Overflow in Practice:

- Pacman Level 256
- Make sure to reboot Boeing Dreamliners every 248 days
- Reported vulnerability CVE-2019-3857 in libssh2 may allow a hacker to remotely execute code
- Donkey Kong Kill Screen

Demo Revisited: Unexpected Behavior

Comair/Delta airline had to cancel thousands of flights days before Christmas because of integer overflow – they exceeded 32,767 crew changes (limit of **short**).

```
int main(int argc, char *argv[]) {
  short airlineCrewChangesThisMonth = 0;

  for (int i = 0; i < 31; i++) {
    airlineCrewChangesThisMonth += 1200;
    printf(...);
  }
}
```

Recap

- Integer Representations
- Bits and Bytes
- Hexadecimal
- Unsigned Integers
- Signed Integers
- Overflow

Lecture 3 takeaway: computers represent everything in binary. We must determine how to represent our data (e.g., base-10 numbers) in a binary format so a computer can manipulate it. There may be limitations to these representations! (overflow)

Next time: How can we manipulate individual bits and bytes?

Extra Slides

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
int x = 53191;
short sx = x; // -12345!
```

```
 x = 0000 0000 0000 0000 1100 1111 1100 0111
sx =                     1100 1111 1100 0111
```

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
int x = -3;
short sx = x; // still -3
```

```
 x = 1111 1111 1111 1111 1111 1111 1111 1101
sx =                     1111 1111 1111 1101
```

Expanding Bit Representations

Sometimes, we want to carry over a value to a larger variable (e.g. make an **int** and set it equal to a `short`).

- For **unsigned** values, C adds *leading zeros* to the representation ("zero extension")
- For **signed** values, C *repeats the sign of the value* for new digits ("sign extension"

Expanding Bit Representation

If we want to **expand** the bit size of an **unsigned** number, C *adds leading zeros.*

```
unsigned short s = 4;
unsigned int i = s; // still 4
```

```
s =                     0000 0000 0000 0100
i = 0000 0000 0000 0000 0000 0000 0000 0100
```

Expanding Bit Representation

If we want to **expand** the bit size of an **signed** number, C *adds repeats the sign.*

```
short s = -4;
int i = s; // still -4
```

```
s =                     1111 1111 1111 1100
i = 1111 1111 1111 1111 1111 1111 1111 1100
```

Expanding Bit Representation

If we want to **expand** the bit size of an **signed** number, C *adds repeats the sign.*

```
short s = 4;
int i = s; // still 4
```

```
s =                          0000 0000 0000 0100
i = 0000 0000 0000 0000 0000 0000 0000 0100
```

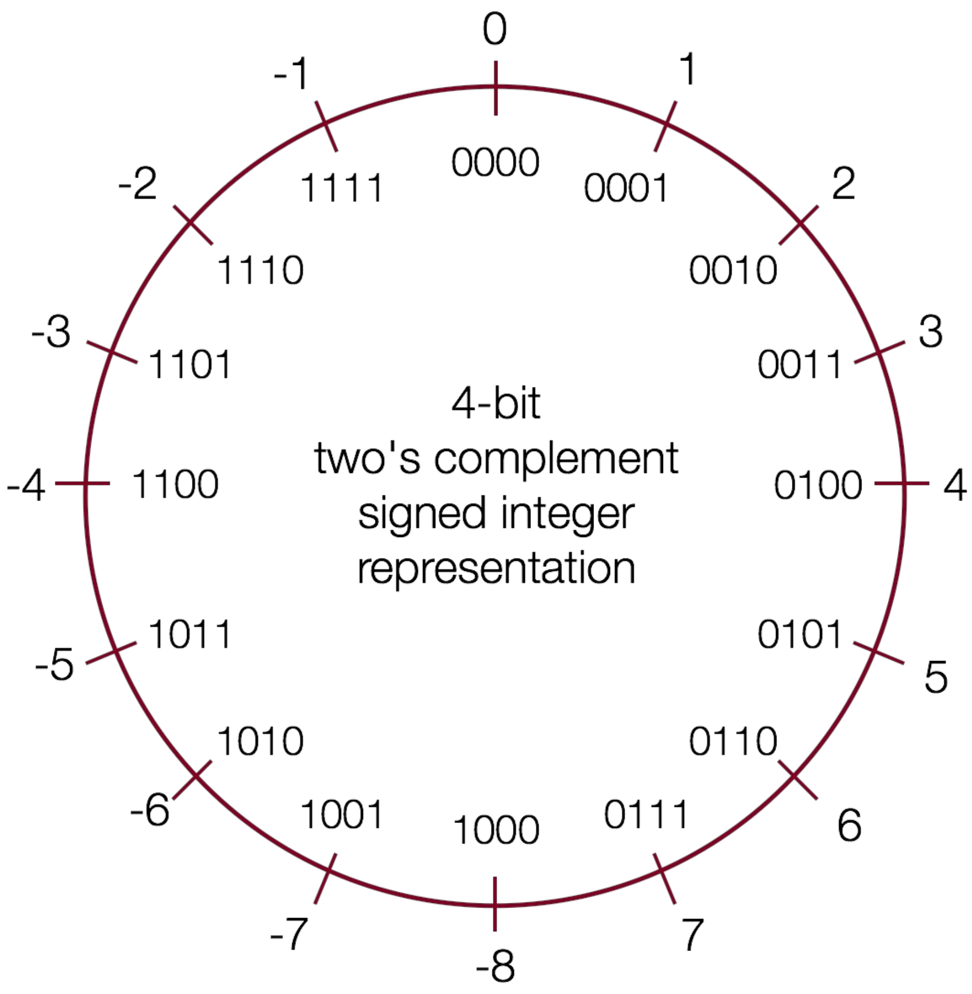
Practice: Two's Complement

What are the negative or positive equivalents of the numbers below?

a) -4 (1100)

b) 7 (0111)

c) 3 (0011)



Practice: Two’s Complement

Fill in the below table:

> It’s easier to compute base-10 for positive numbers, so use two’s complement first if negative.

	`char x = ____;`		`char y = -x;`	
	decimal	binary	decimal	binary
1.		`0b1111 1100`		
2.		`0b0001 1000`		
3.		`0b0010 0100`		
4.		`0b1101 1111`		

Practice: Two's Complement

Fill in the below table:

It's easier to compute base-10 for positive numbers, so use two's complement first if negative.

	`char x = ____;`		`char y = -x;`	
	decimal	binary	decimal	binary
1.	-4	0b1111 1100	4	0b0000 0100
2.		0b0001 1000		
3.		0b0010 0100		
4.		0b1101 1111		

Practice: Two's Complement

Fill in the below table:

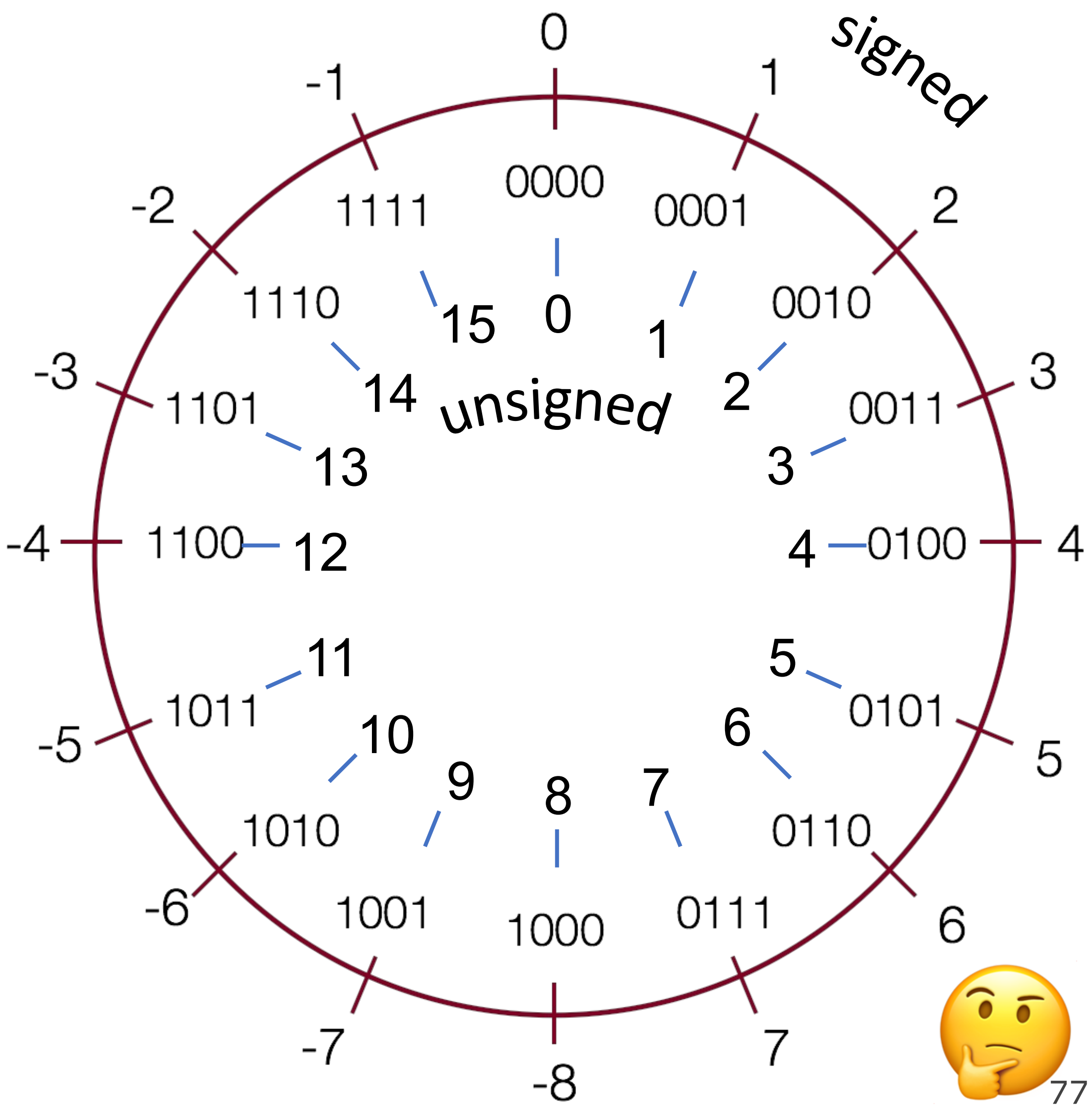
It's easier to compute base-10 for positive numbers, so use two's complement first if negative.

	`char x = ____;`		`char y = -x;`	
	decimal	binary	decimal	binary
1.	-4	0b1111 1100	4	0b0000 0100
2.	24	0b0001 1000	-24	0b1110 1000
3.	36	0b0010 0100	-36	0b1101 1100
4.	-33	0b1101 1111	33	0b0010 0001

Underspecified question

What is the following base-2 number in base-10?

`0b1101`



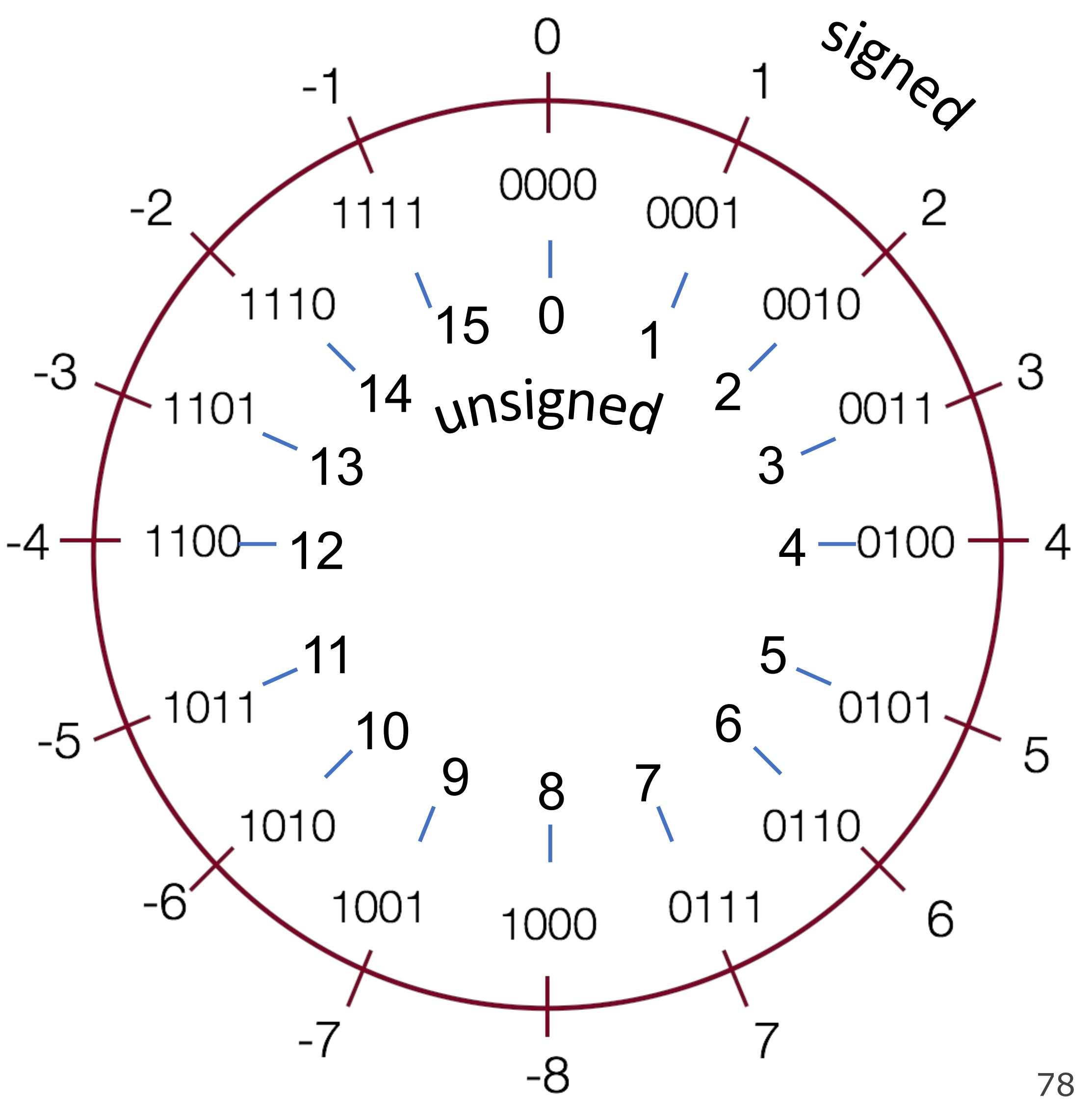
Underspecified question

What is the following base-2 number in base-10?

`0b1101`

If 4-bit signed: -3
If 4-bit unsigned: 13
If >4-bit signed or unsigned: 13

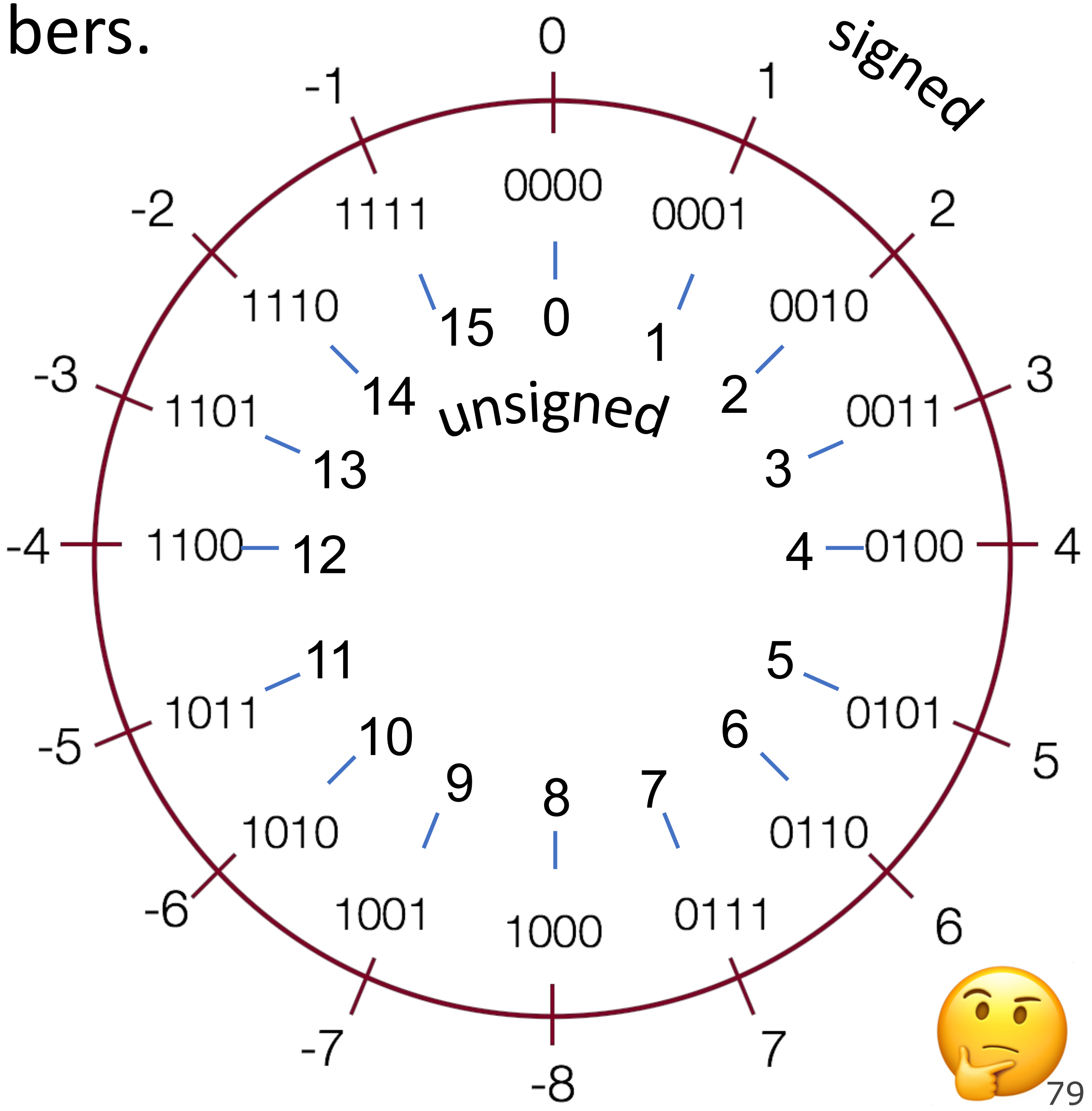
You need to know the type to determine the number! (Note by default, numeric constants in C are `signed ints`)



Overflow

- What is happening here? Assume 4-bit numbers.

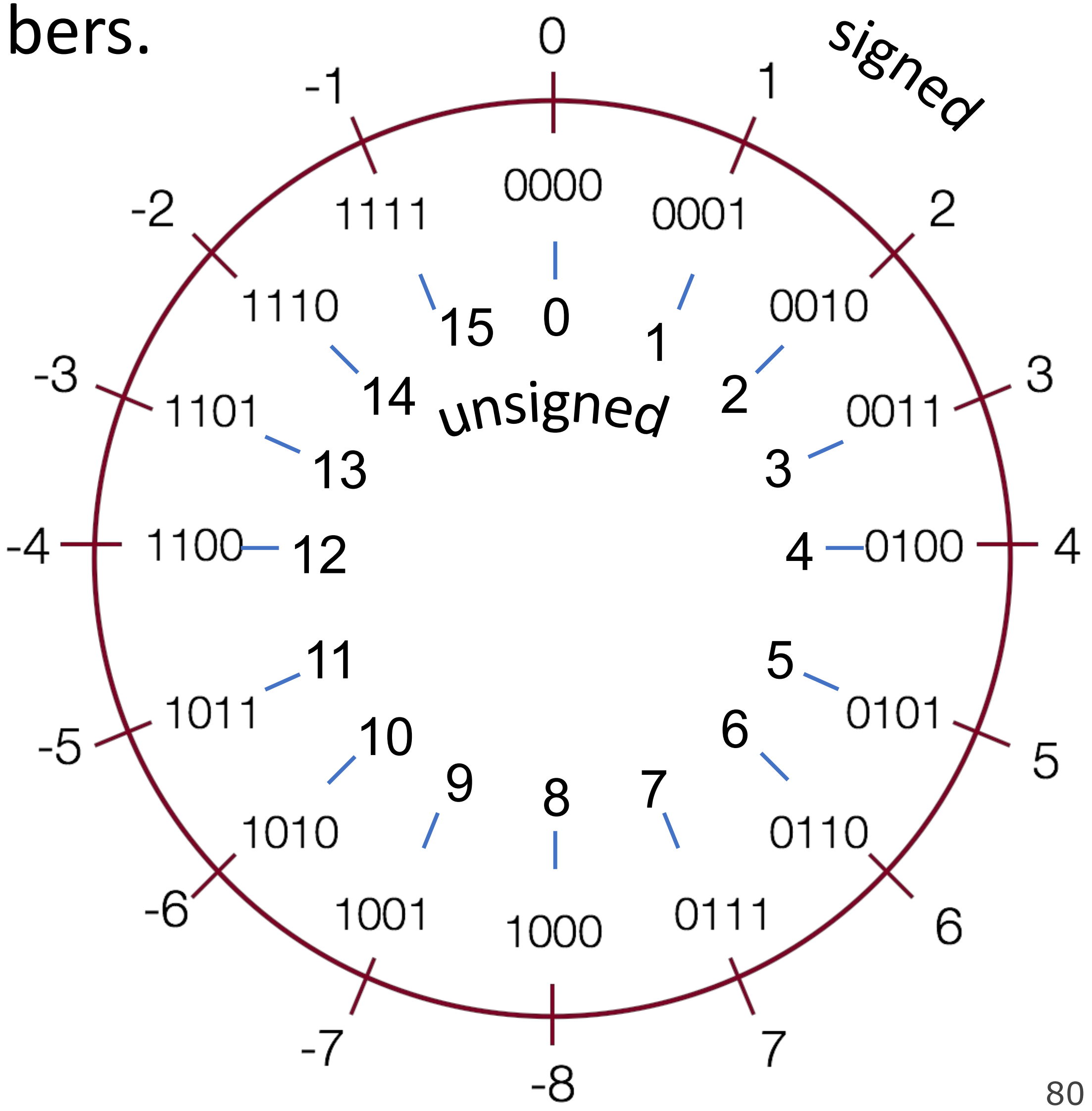
```
  0b1101
+ 0b0100
```



Overflow

- What is happening here? Assume 4-bit numbers.

```
  0b1101
+ 0b0100
```



Signed	Unsigned
-3 + 4 = 1	13 + 4 = 1
No overflow	Overflow

Limits and Comparisons

1. What is the…

	Largest unsigned?	Largest signed?	Smallest signed?
`char`			
`int`			

Limits and Comparisons

1. What is the…

	Largest unsigned?	Largest signed?	Smallest signed?
char	$2^8 - 1 = 255$	$2^7 - 1 = 127$	$-2^7 = -128$
int	$2^{32} - 1 = 4294967296$	$2^{31} - 1 = 2147483647$	$-2^{31} = -2147483648$

These are available as UCHAR_MAX, INT_MIN, INT_MAX, etc. in the `<limits.h>` header.