CS107 Lecture 2

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

- 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 its impacts

Demo: Unexpected Behavior

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

Lecture Plan

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

Lecture Plan

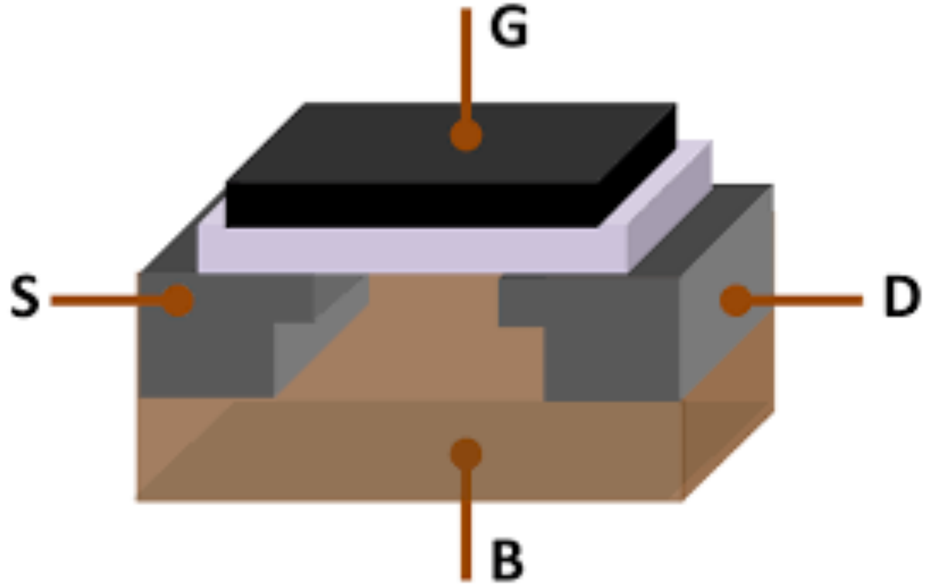
- **Bits and Bytes**
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow

0

1

Bits

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



One Bit At A Time

- 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 still fundamentally operate on bits; we have just gotten more creative about how to represent different data as bits!
  - Images
  - Audio
  - 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

$$= \mathbf{5}*1000 + \mathbf{9}*100 + \mathbf{3}*10 + \mathbf{4}*1$$

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

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

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

$1100_2 \times 2 = 1100\underline{\mathbf{0}}$

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

- Bits and Bytes
- **Hexadecimal**
- Integer Representations
- 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**.

$$\underbrace{0110}_{0\text{-}15}\ \underbrace{1010}_{0\text{-}15}\ \underbrace{0011}_{0\text{-}15}$$

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**.

0-15 0-15 0-15

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`**

$$0\mathrm{x}\,\underbrace{\mathrm{f}}_{1111}\;\underbrace{5}_{0101}$$

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: 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

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

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})

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!

Number Representations

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

In The Days Of Yore…

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

Transitioning To Larger Datatypes

- **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 have at most **4GB** of memory (RAM)!
- Because of this, computers transitioned to **64-bit.** This means that datatypes were enlarged; pointers in programs were now **64 bits.**
- 64-bit pointers store a memory address from 0 to 2^{64}-1, equaling **2^{64} bytes of addressable memory.** This equals **16 Exabytes**, meaning that 64-bit computers could have at most **1024*1024*1024*16 GB** of memory (RAM)!

Lecture Plan

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

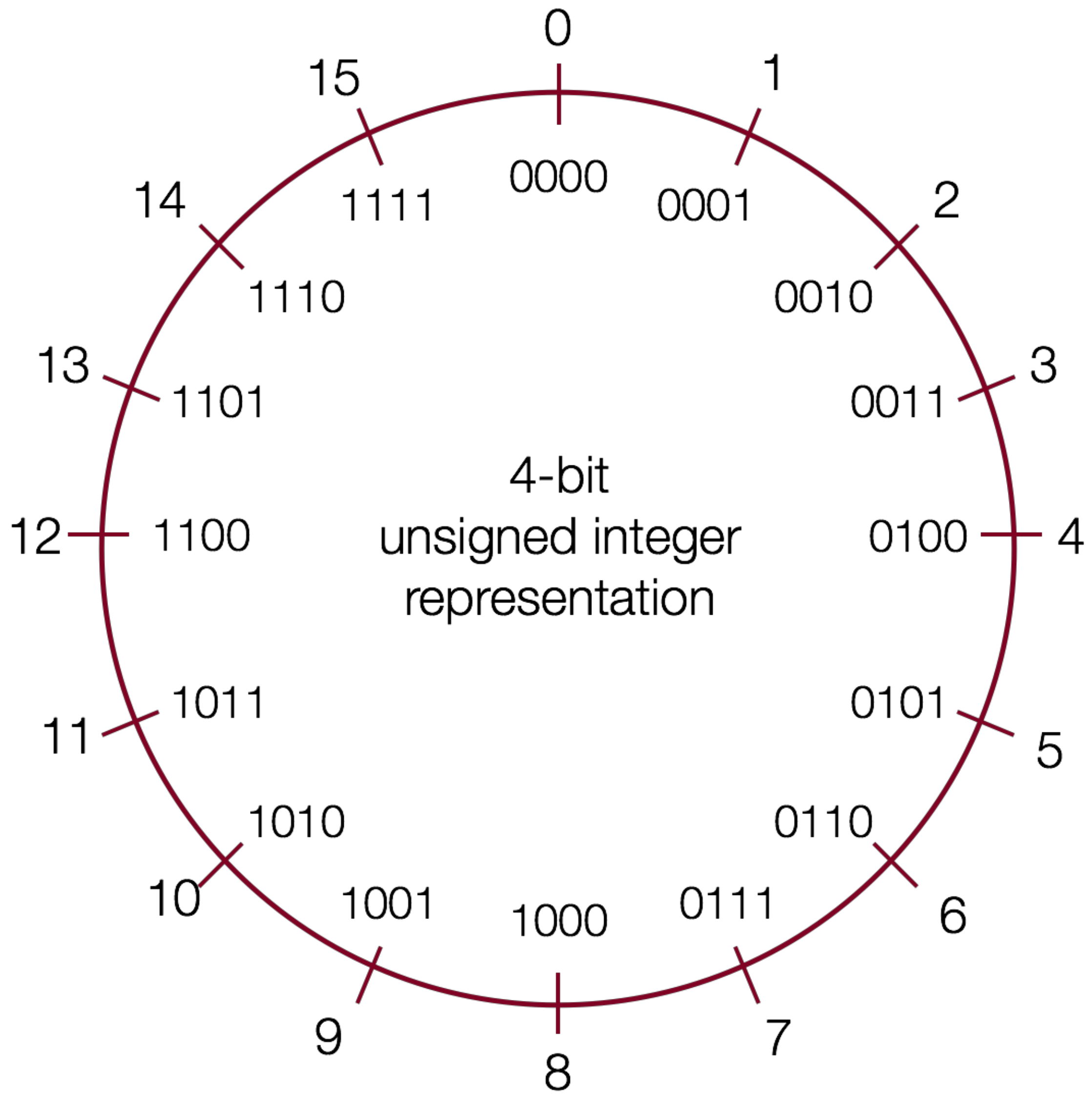
Unsigned Integers

- 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).

Unsigned Integers



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

- Bits and Bytes
- Hexadecimal
- Integer Representations
- 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?

Idea: let's reserve the *most significant bit* to store the sign.

Sign Magnitude Representation

$$\underbrace{0}_{\text{positive}}\underbrace{110}_{6}$$

$$\underbrace{1}_{\text{negative}}\underbrace{011}_{3}$$

Sign Magnitude Representation

$$\underbrace{0}_{\text{positive}}\underbrace{000}_{0}$$

$$\underbrace{1}_{\text{negative}}\underbrace{000}_{0}$$

Sign Magnitude Representation

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’ve only represented 15 of our 16 available numbers!

Sign Magnitude Representation

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

Can we do better?

A Better Idea

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

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

A Better Idea

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

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

A Better Idea

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

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

A Better Idea

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

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

A Better Idea

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

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

A Better Idea

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

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

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
```

Another Trick

To find the negative equivalent of a number, work right-to-left and write down all digits *through* when you reach a 1. Then, invert the rest of the digits.

$$
\begin{array}{r}
100100 \\
+\,?????? \\
\hline
000000
\end{array}
$$

Another Trick

To find the negative equivalent of a number, work right-to-left and write down all digits *through* when you reach a 1. Then, invert the rest of the digits.

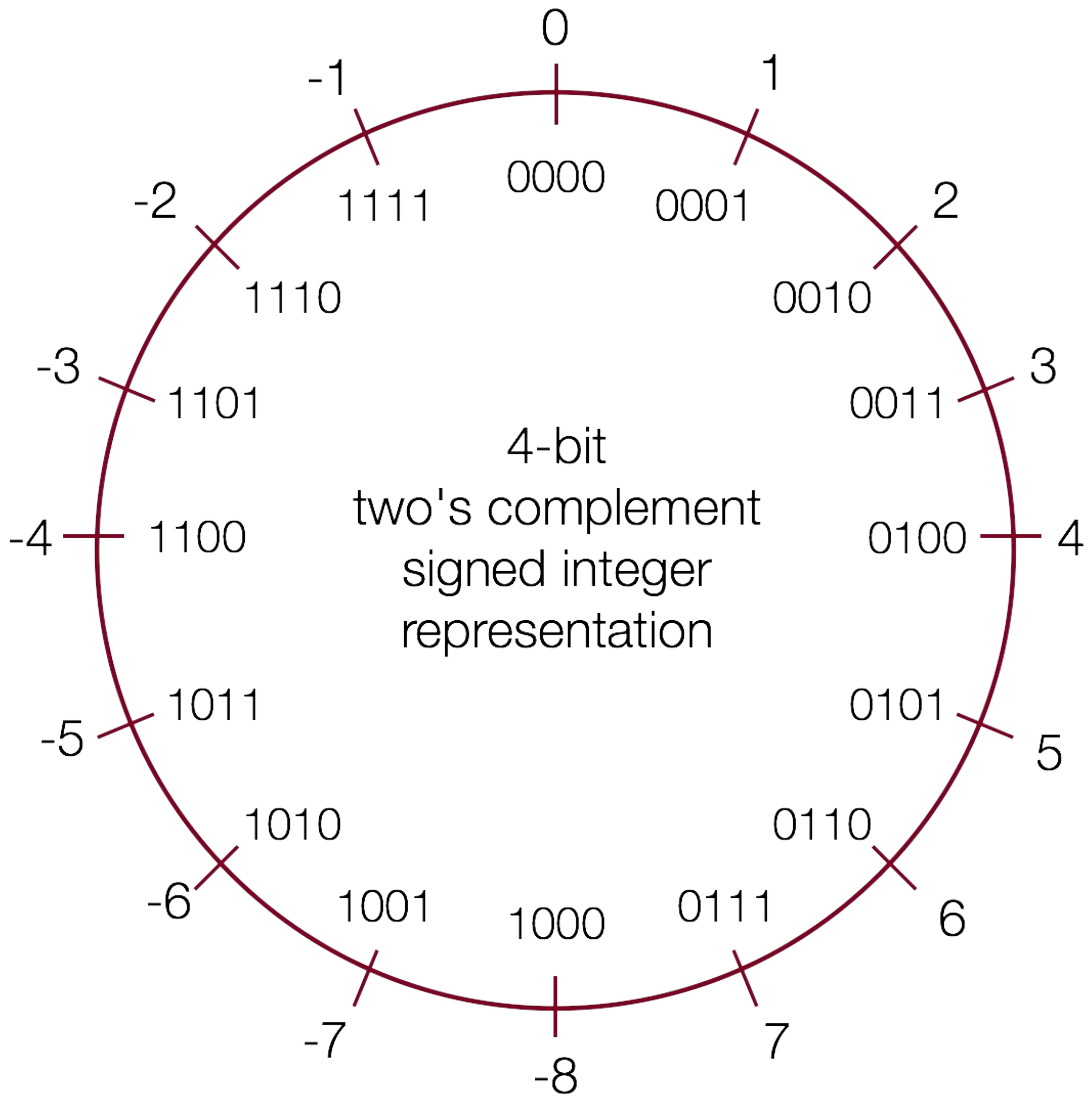
$$
\begin{array}{r}
100100 \\
+\,???100 \\
\hline
000000
\end{array}
$$

Another Trick

To find the negative equivalent of a number, work right-to-left and write down all digits *through* when you reach a 1. Then, invert the rest of the digits.

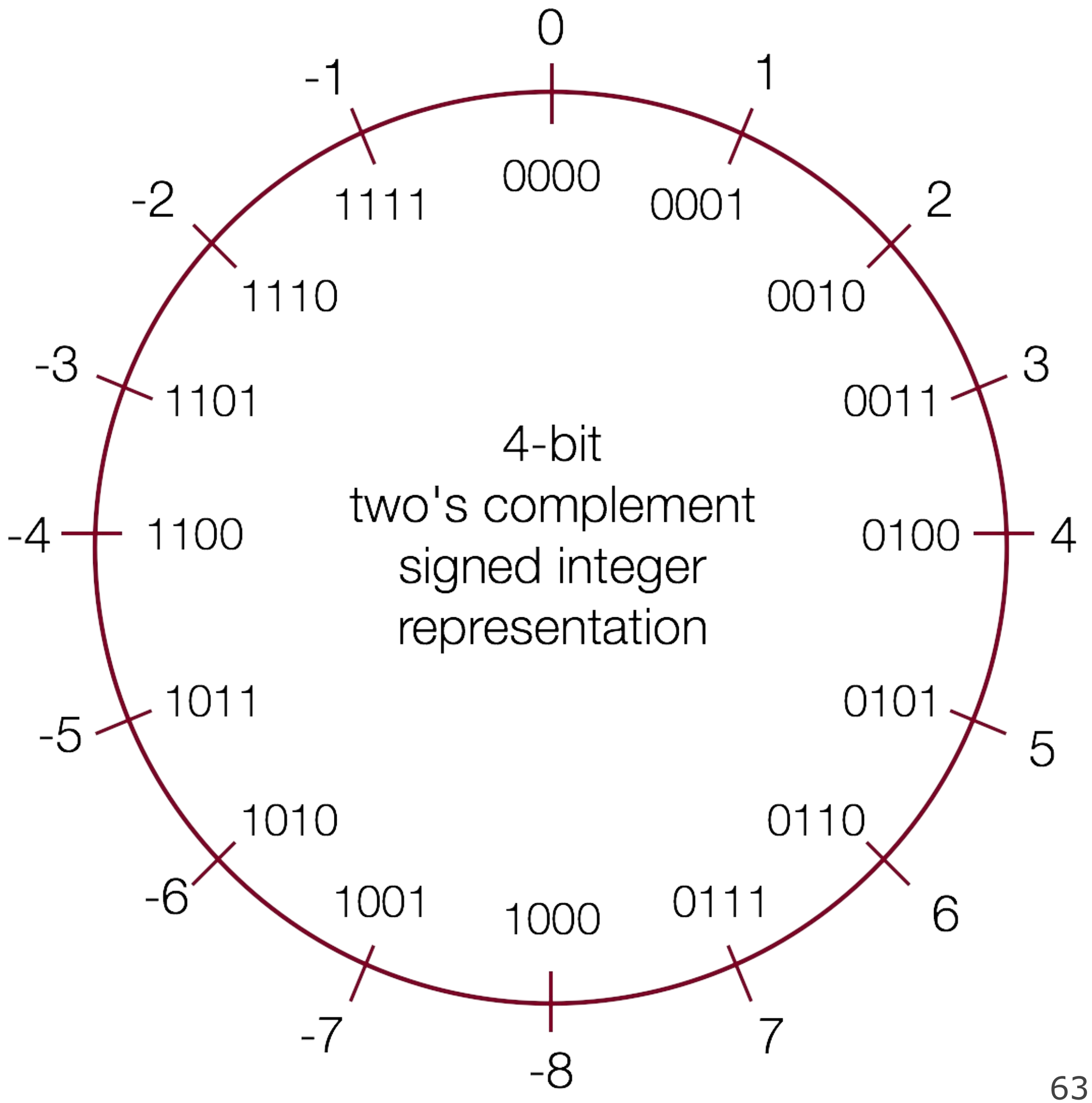
$$\begin{array}{r} 100100 \\ +\ 011100 \\ \hline 000000 \end{array}$$

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!



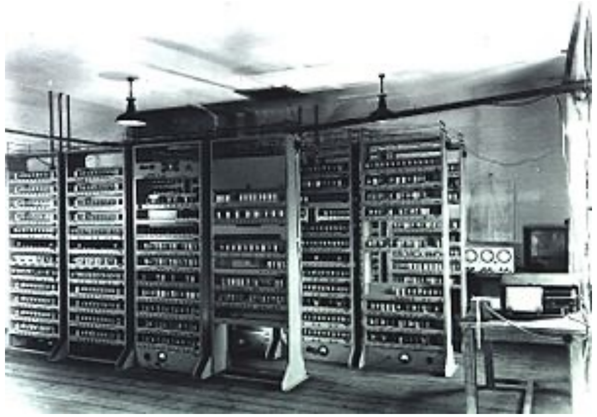
History: Two’s complement

- The 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 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 (had 1024KB memory) and established two’s complement as the dominant binary representation of integers

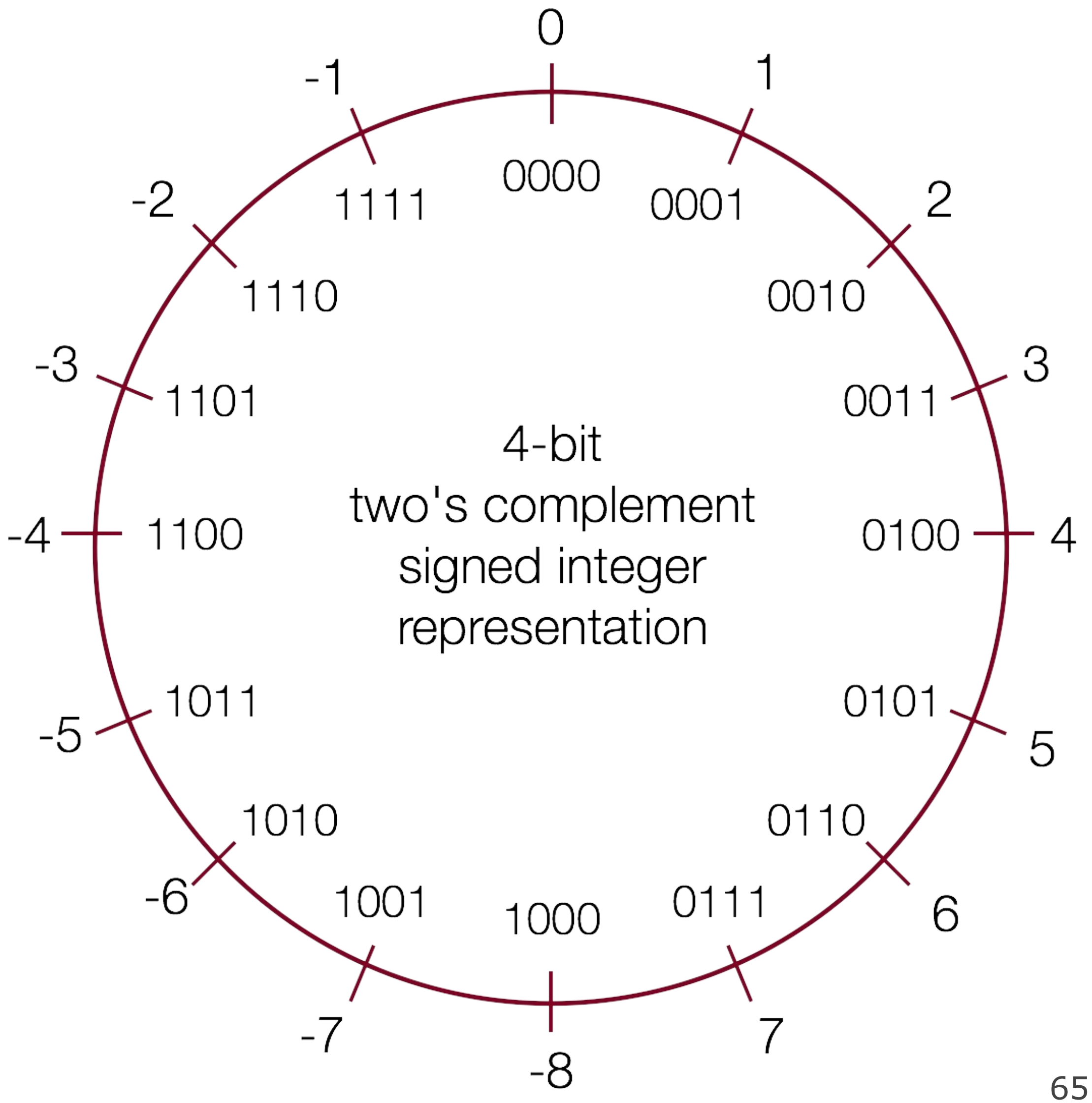


EDSAC (1949)

System/360 (1964)

Two's Complement

- **Con:** more difficult to represent, and difficult to convert to/from decimal and between positive and negative.
- **Pro:** only 1 representation for 0!
- **Pro:** all bits are used to represent as many numbers as possible
- **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…adding! There is no special case needed for negatives. 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 one of them and then adding. E.g. 4 – 5 = -1.

$$\begin{array}{r} 0100 \\ -\,0101 \\ \hline \\ \end{array} \quad \begin{array}{r} 4 \\ 5 \\ \\ \end{array} \quad \longrightarrow \quad \begin{array}{r} 0100 \\ +\,\color{red}{1011} \\ \hline 1111 \end{array} \quad \begin{array}{r} 4 \\ -5 \\ -1 \end{array}$$

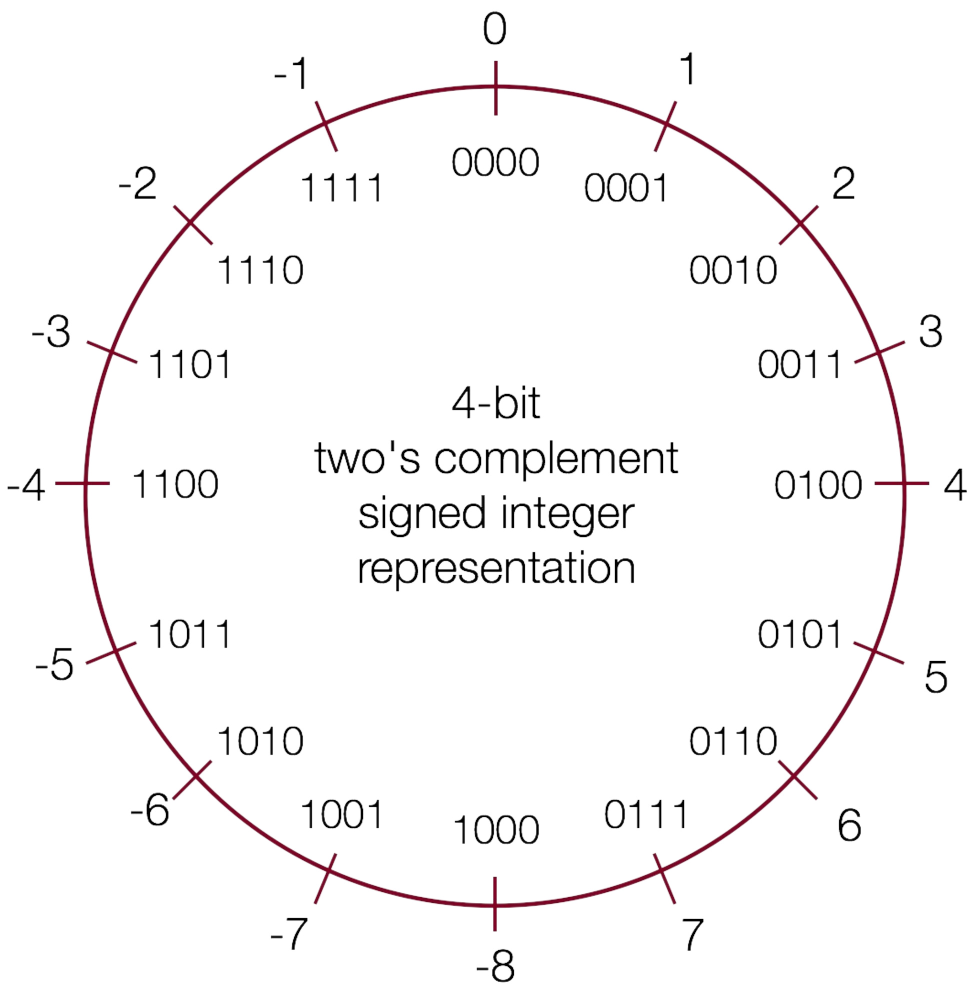
Practice: Two's Complement

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

a) -4 (1100)

b) 7 (0111)

c) 3 (0011)



Break Time!

To think about during the break:

How can what we’ve learned so far about integer representations help us understand the behavior of the airline program from the start of lecture?

Lecture Plan

- Bits and Bytes
- Hexadecimal
- Integer Representations
- 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.

```
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.

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

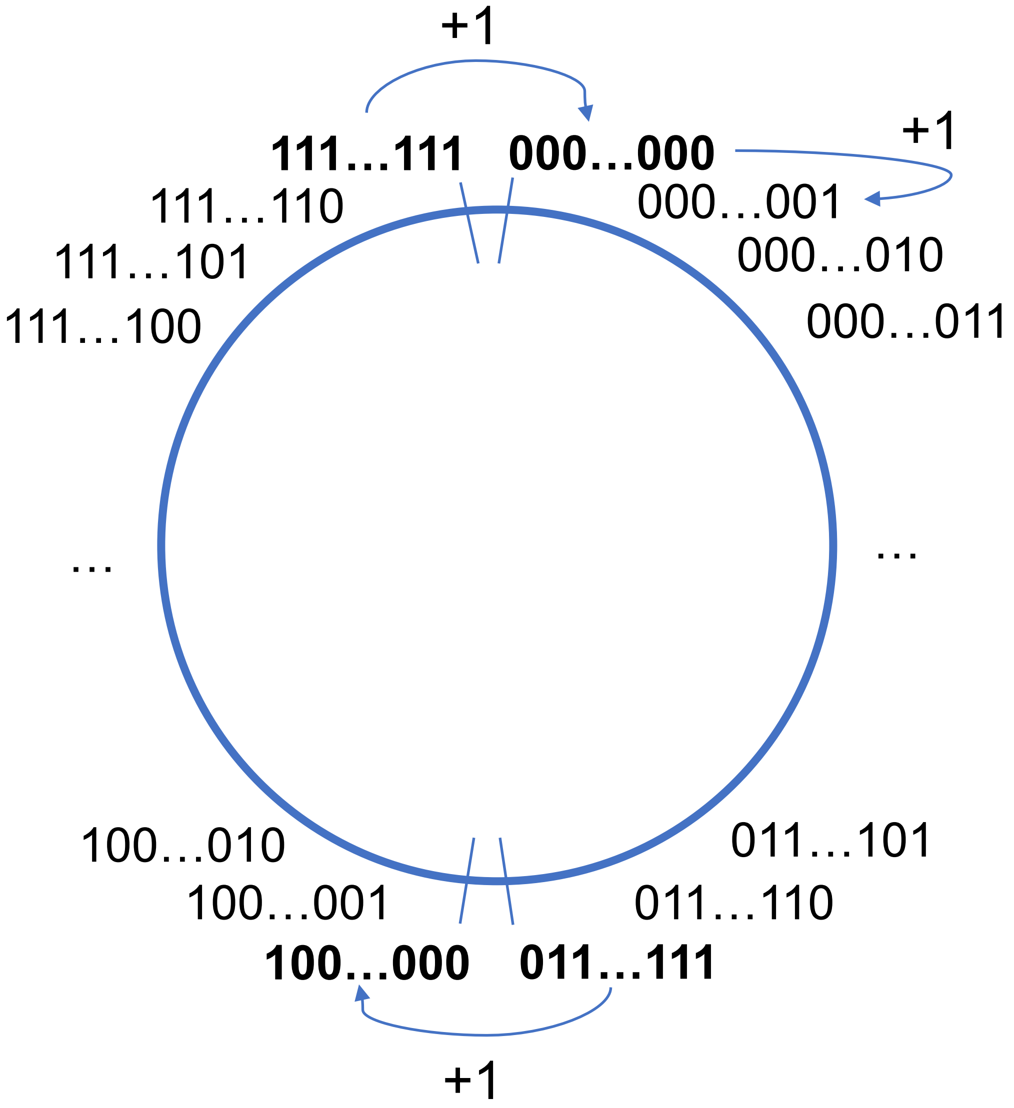
Min and Max Integer Values

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

Min and Max Integer Values

In C, there are various constants that represent these minimum and maximum values: `INT_MIN`, `INT_MAX`, `UINT_MAX`, `LONG_MIN`, `LONG_MAX`, `ULONG_MAX`, …

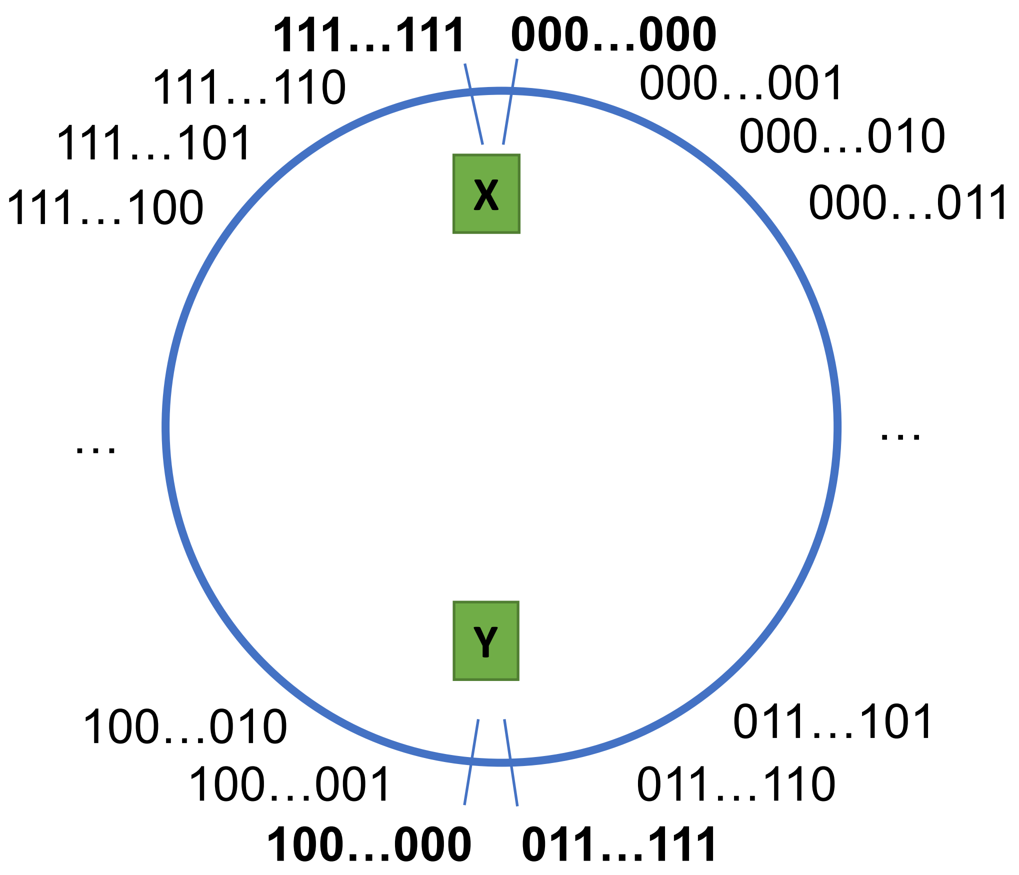
Overflow


+1
111…111
000…000
+1
111…110
000…001
111…101
000…010
111…100
000…011
…
…
100…010
011…101
100…001
011…110
100…000
011…111
+1

Overflow

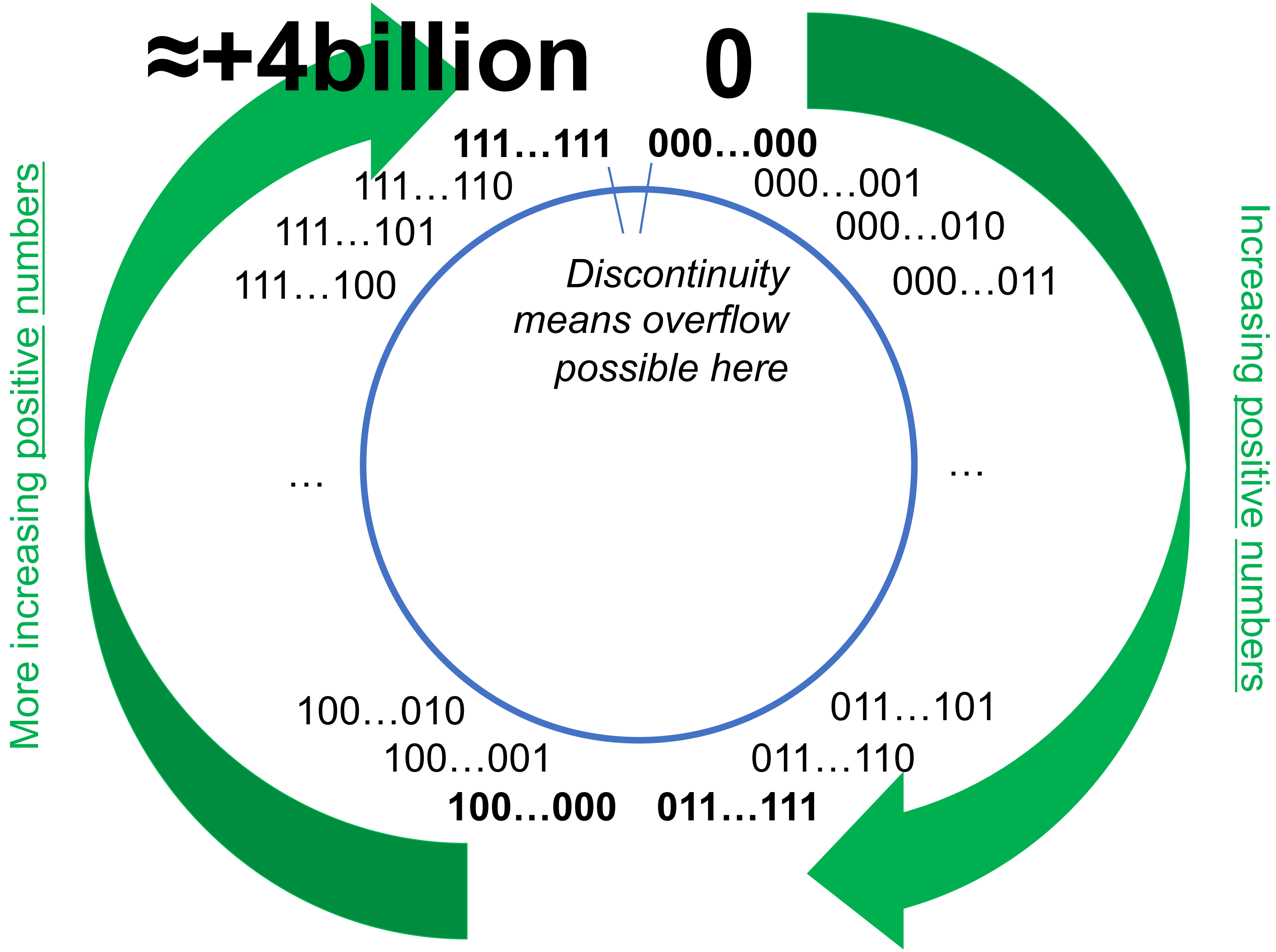
At which points can overflow occur for signed and unsigned int? *(assume binary values shown are all 32 bits)*

A. Signed and unsigned can both overflow at points X and Y
B. Signed can overflow only at X, unsigned only at Y
C. Signed can overflow only at Y, unsigned only at X
D. Signed can overflow at X and Y, unsigned only at X
E. Other



Key Idea: Overflow means *discontinuity*

Unsigned Integers



Signed Numbers

-1 0 +1

111…111 **000…000**

111…110
111…101
111…100

…

100.

000…001
000…010
000…011

…

Discontinuity means overflow possible here

00 **011…111**

011…101
011…110

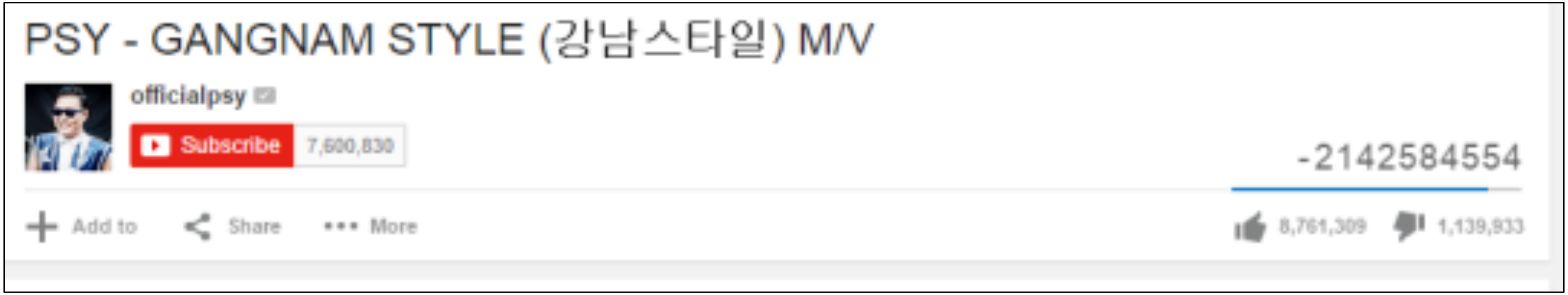
≈+2billion

≈-2billion

Negative numbers becoming less negative (i.e. increasing)

Increasing positive 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]

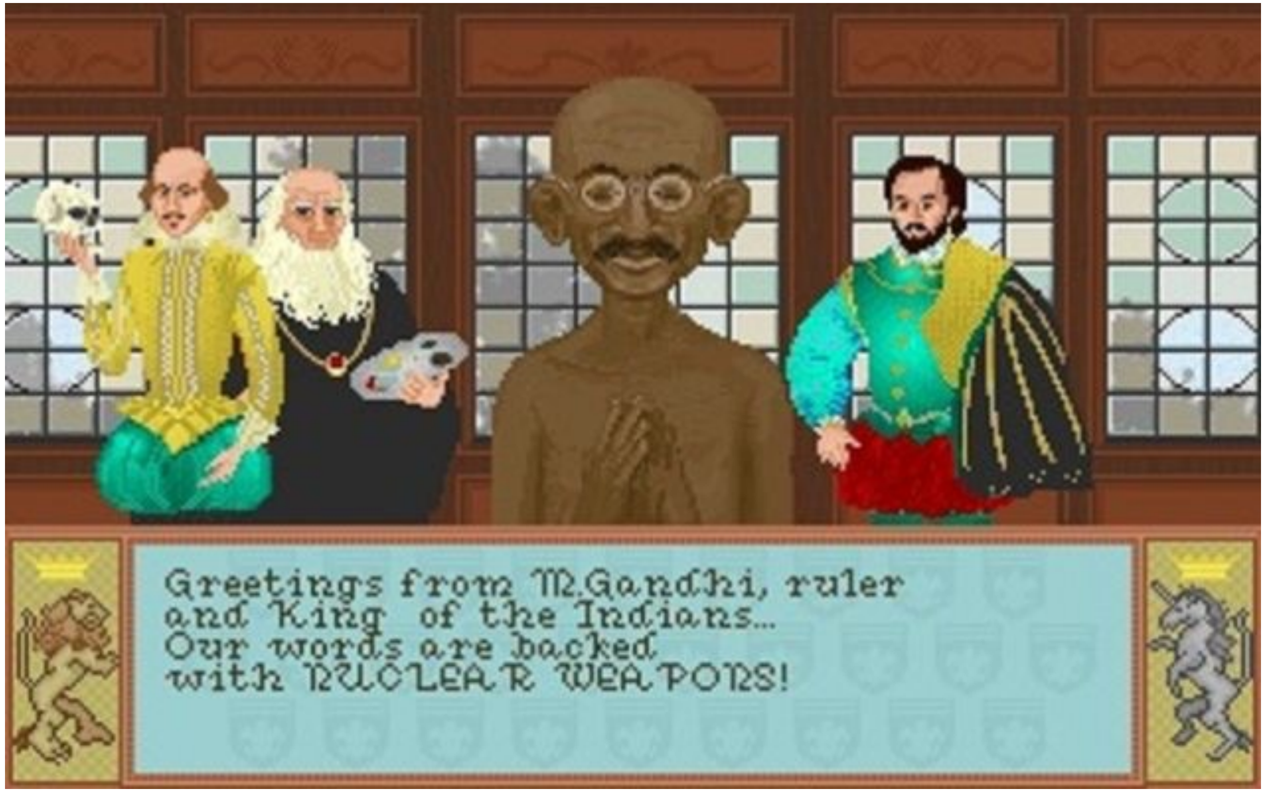
Overflow In Practice: Timestamps

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: Gandhi

- In the game "Civilization", each civilization leader had an "aggression" rating. Gandhi was meant to be peaceful, and had a score of 1.
- If you adopted "democracy", all players' aggression reduced by 2. Gandhi's went from 1 to **255**!
- Gandhi then became a big fan of nuclear weapons.



https://kotaku.com/why-gandhi-is-such-an-asshole-in-civilization-1653818245

Overflow in Practice:

- Pacman Level 256
- Make sure to reboot Boeing Dreamliners every 248 days
- Comair/Delta airline had to cancel thousands of flights days before Christmas
- Reported vulnerability CVE-2019-3857 in libssh2 may allow a hacker to remotely execute code
- Donkey Kong Kill Screen

Demo Revisited: Unexpected Behavior

`airline.c`

Recap

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

Lecture 2 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 Practice

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		

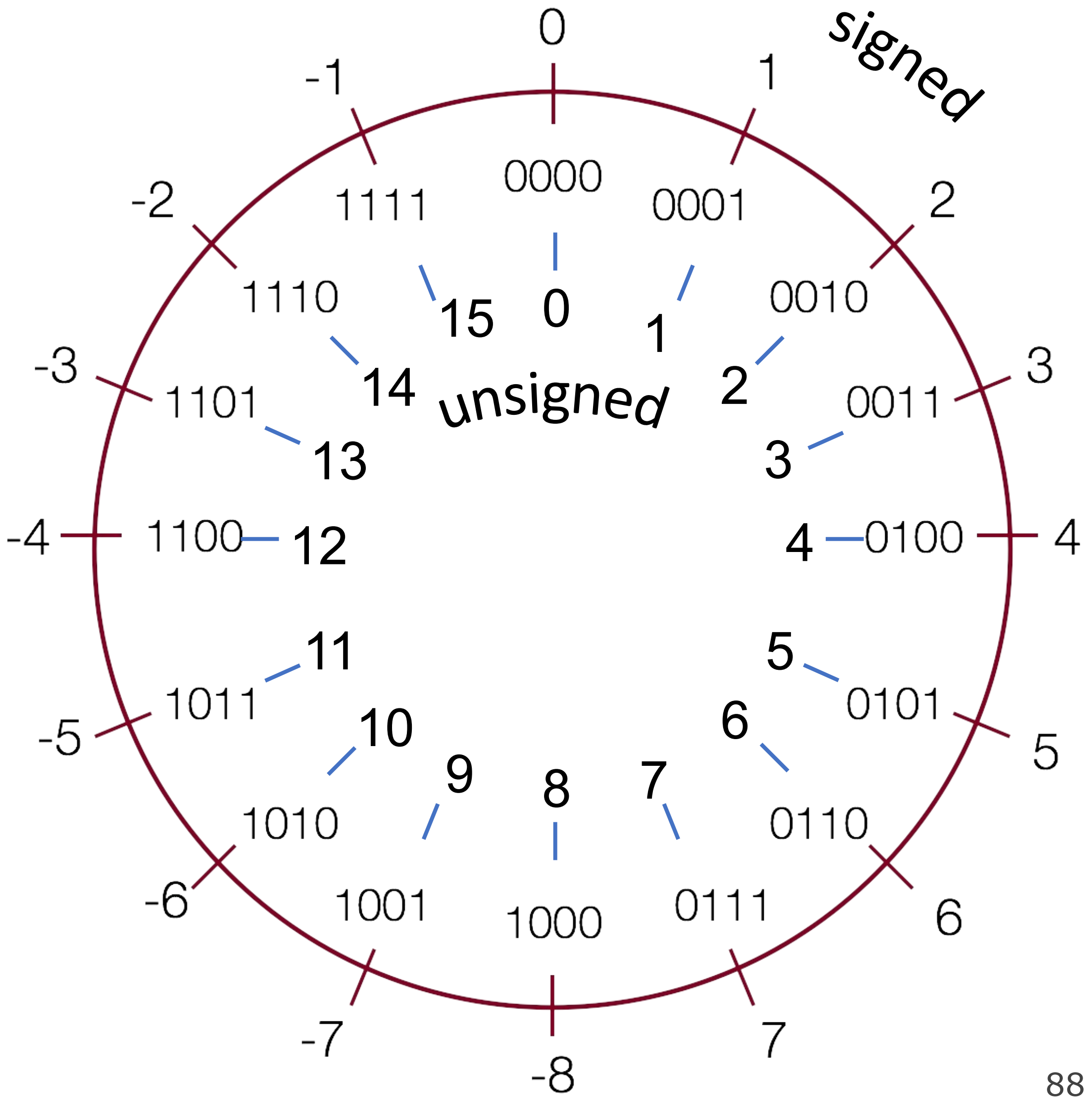
Practice: Two's Complement

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

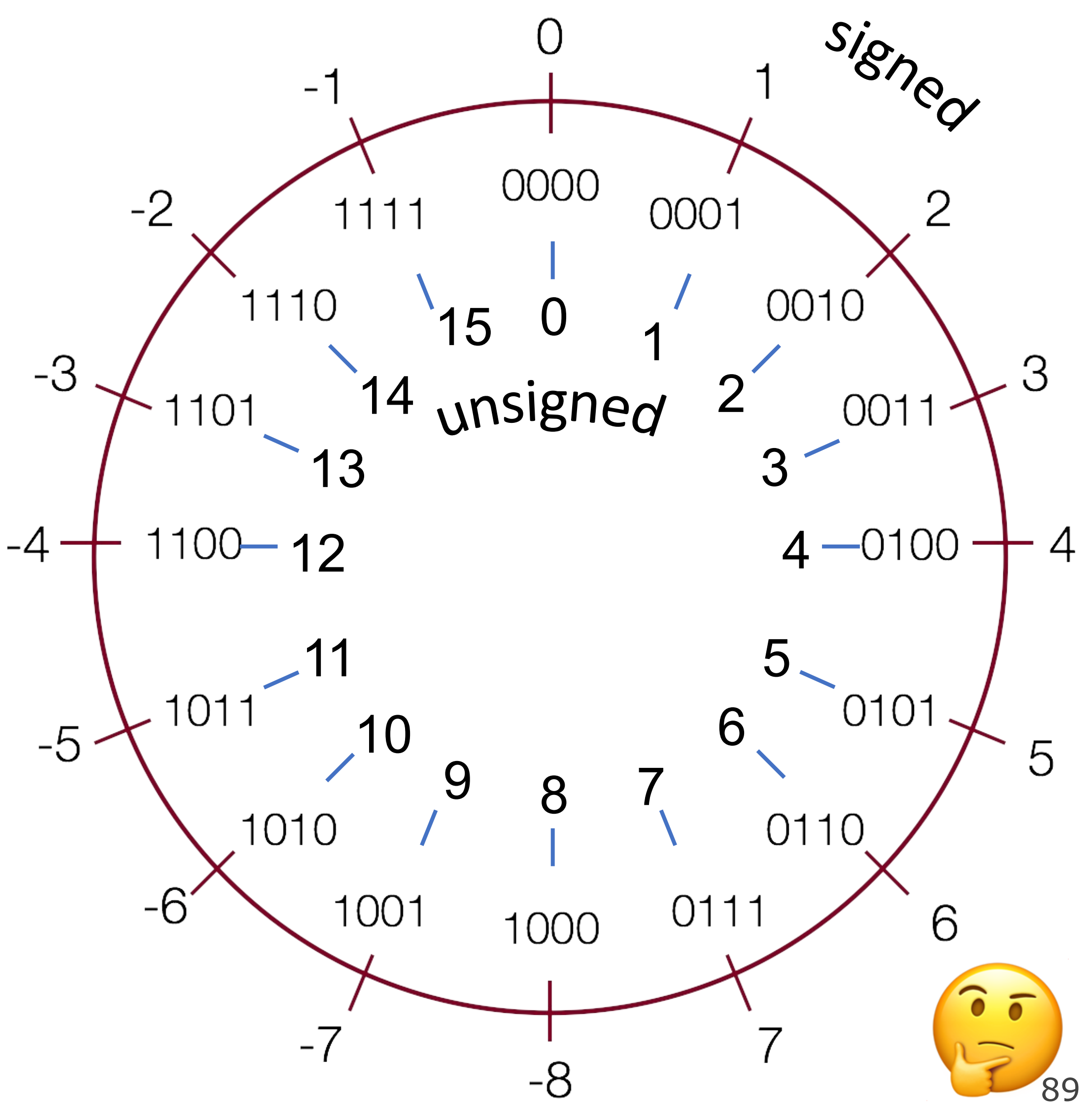
Signed vs. Unsigned Integers



Underspecified question

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

```
0b1101
```



Underspecified question

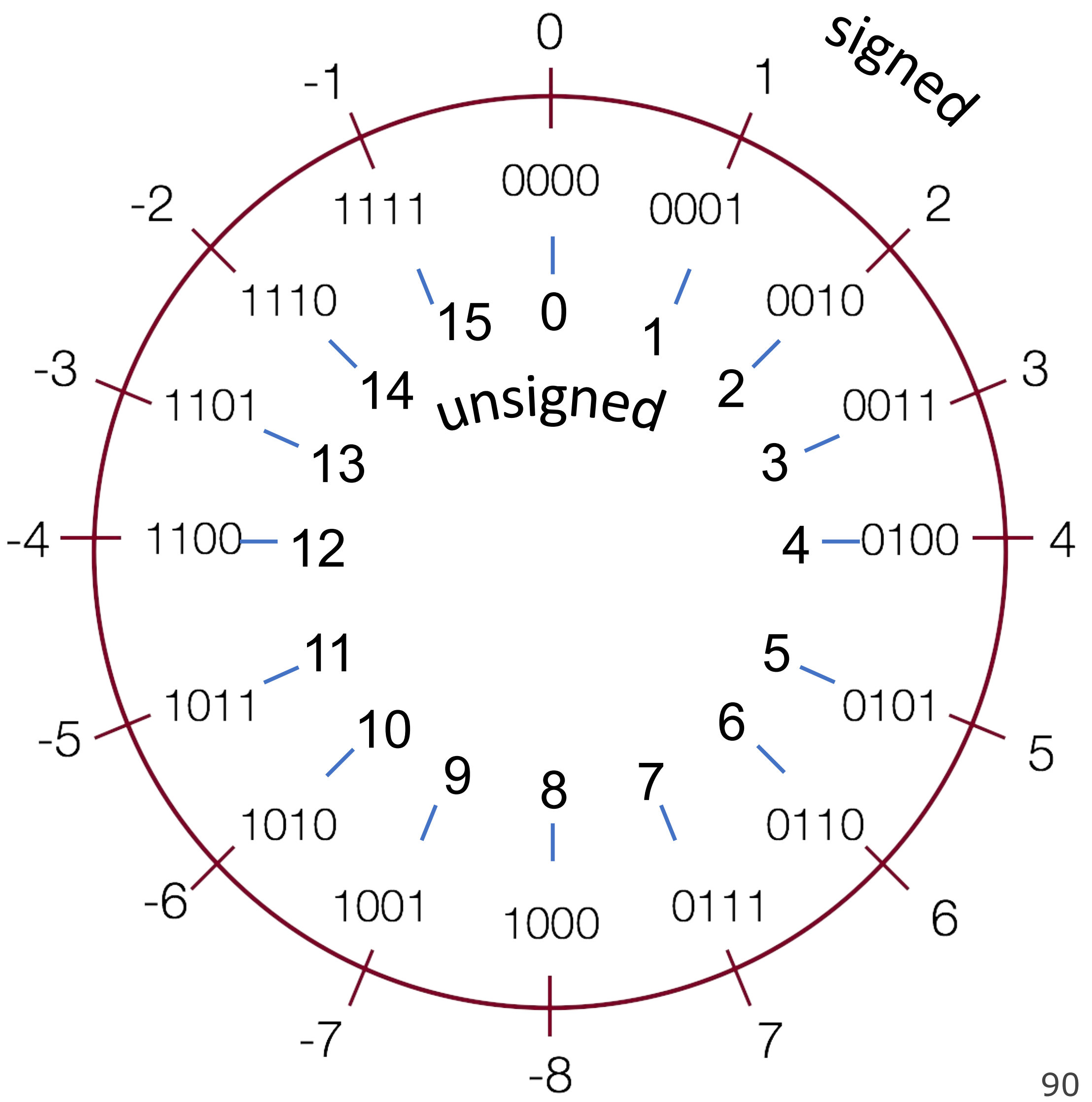
What is the following base-2 number in base-10?

`0b1101`

If 4-bit signed: -3

If 4-bit unsigned: 13

If >4-bit signed or unsigned: 13

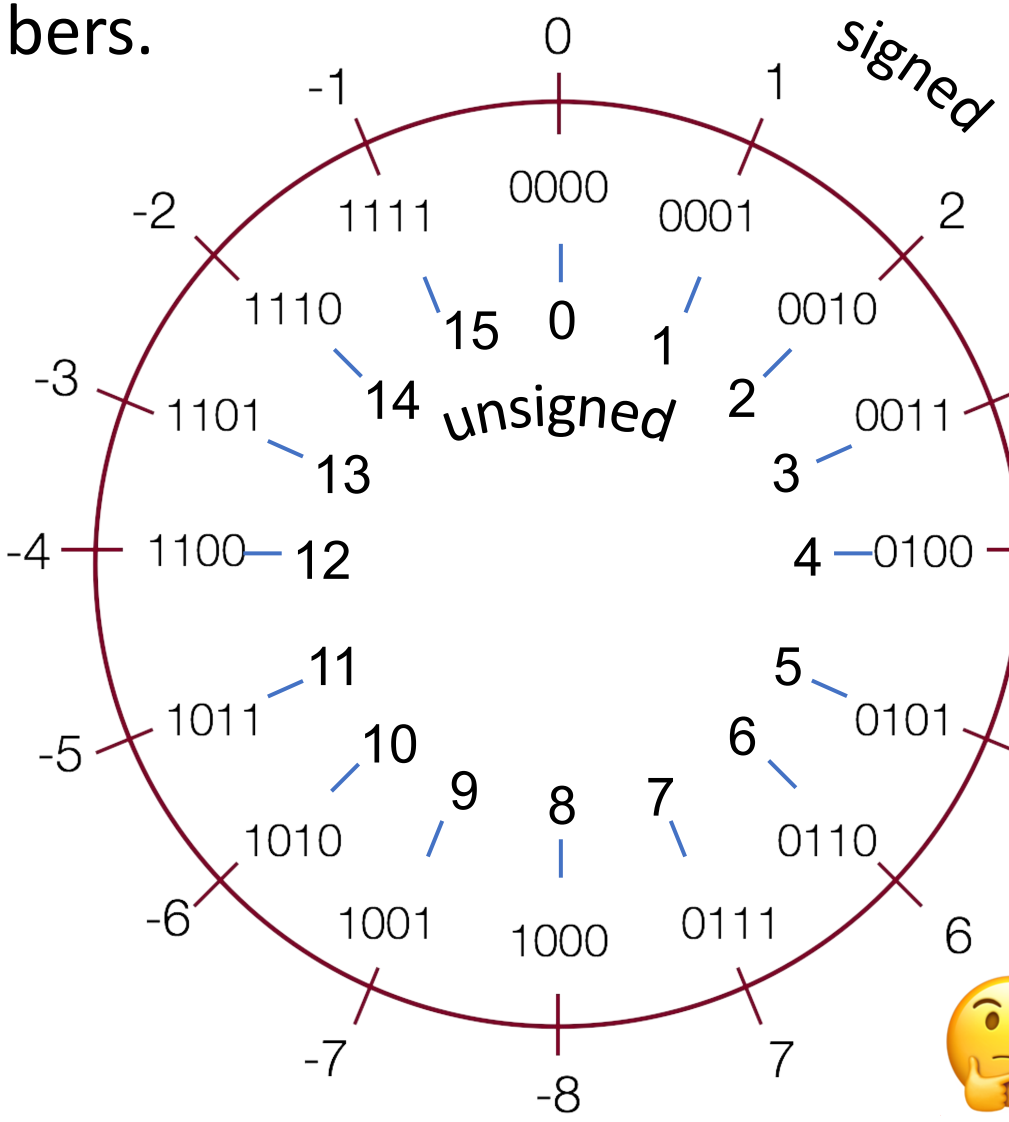


You need to know the type to determine the number! (Note by default, numeric constants in C are `signed int`s)

Overflow

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

```
  0b1101
+ 0b0100
```



Overflow

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

```
  0b1101
+ 0b0100
________
```

<u>Signed</u>

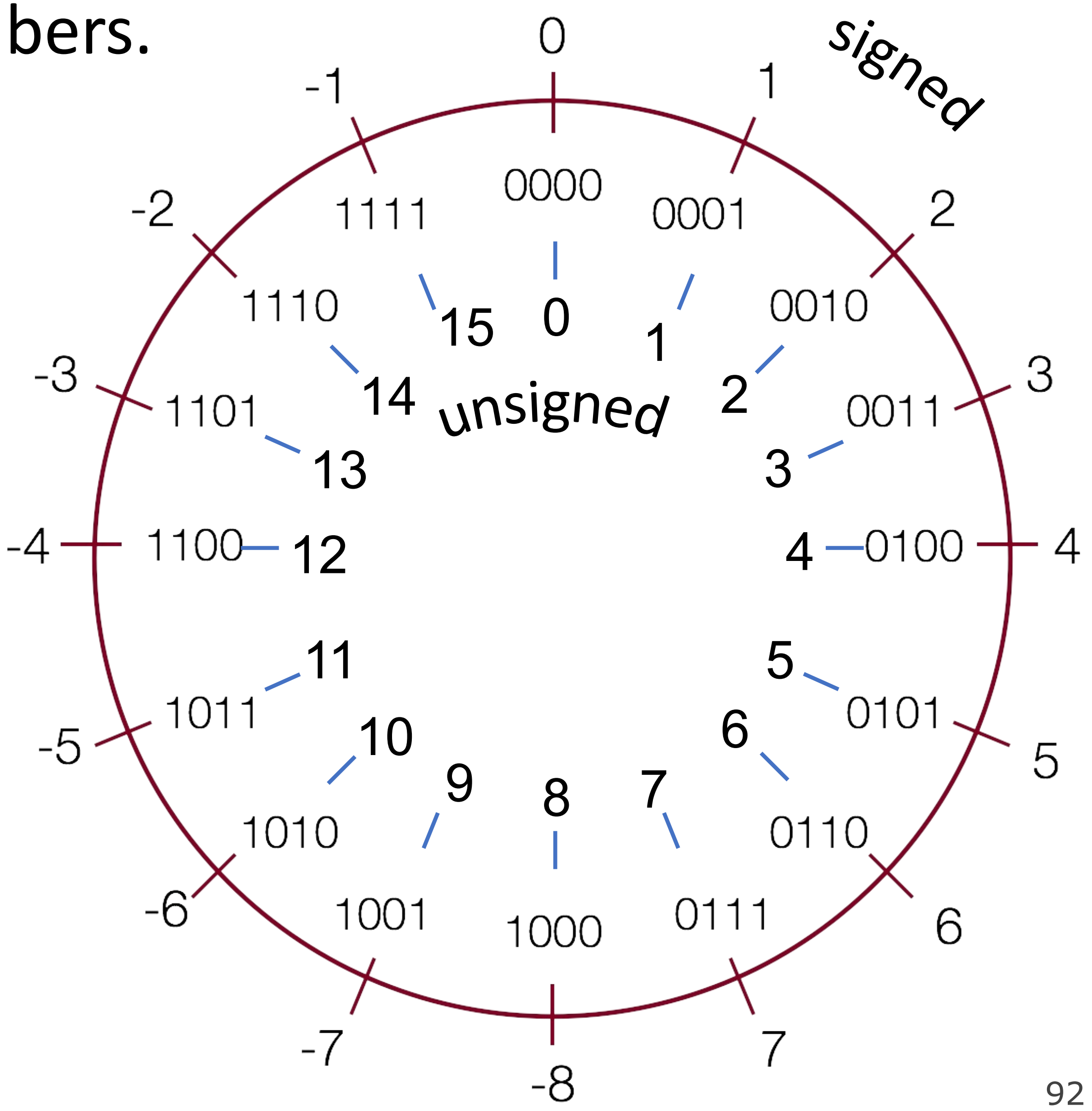
`-3 + 4 = 1`

No overflow

<u>Unsigned</u>

`13 + 4 = 1`

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.