

# 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?**

# Demo: Unexpected Behavior



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

# Lecture Plan

• Bits and Bytes	5
• Hexadecimal	33
• Integer Representations	41
• Unsigned Integers	47
• Signed Integers	51
• Overflow	77
• Casting and Combining Types	90
• Live Session	120

# Lecture Plan

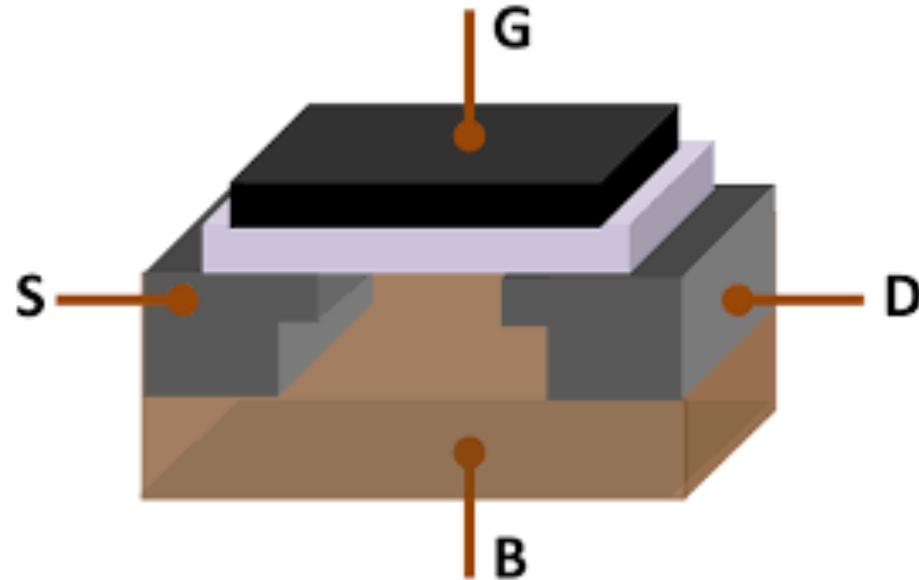
• <b>Bits and Bytes</b>	5
• Hexadecimal	33
• Integer Representations	41
• Unsigned Integers	47
• Signed Integers	51
• Overflow	77
• Casting and Combining Types	90
• Live Session	120

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

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

# 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

$2^x$ :      1   0   1   1  
              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  $\leq 6$ ?

# Base 10 to Base 2

**Question:** What is 6 in base 2?

- Strategy:
  - What is the largest power of  $2 \leq 6$ ?  $2^2=4$

$$\begin{array}{cccc} 0 & 1 & & \\ \hline 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

# Base 10 to Base 2

**Question:** What is 6 in base 2?

• Strategy:

- What is the largest power of  $2 \leq 6$ ?  $2^2=4$
- Now, what is the largest power of  $2 \leq 6 - 2^2$ ?

$$\begin{array}{cccc} 0 & 1 & & \\ \hline 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

# Base 10 to Base 2

**Question:** What is 6 in base 2?

• Strategy:

- What is the largest power of  $2 \leq 6$ ?  $2^2=4$
- Now, what is the largest power of  $2 \leq 6 - 2^2$ ?  $2^1=2$

$$\begin{array}{cccc} 0 & 1 & 1 & \\ \hline 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

# Base 10 to Base 2

**Question:** What is 6 in base 2?

• Strategy:

- What is the largest power of  $2 \leq 6$ ?  $2^2=4$
- Now, what is the largest power of  $2 \leq 6 - 2^2$ ?  $2^1=2$
- $6 - 2^2 - 2^1 = 0$ !

$$\begin{array}{cccc} 0 & 1 & 1 & \\ \hline 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

# Base 10 to Base 2

**Question:** What is 6 in base 2?

• Strategy:

- What is the largest power of  $2 \leq 6$ ?  $2^2=4$
- Now, what is the largest power of  $2 \leq 6 - 2^2$ ?  $2^1=2$
- $6 - 2^2 - 2^1 = 0$ !

$$\begin{array}{cccc} 0 & 1 & 1 & 0 \\ \hline 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

# Base 10 to Base 2

**Question:** What is 6 in base 2?

• Strategy:

- What is the largest power of  $2 \leq 6$ ?  $2^2=4$
- Now, what is the largest power of  $2 \leq 6 - 2^2$ ?  $2^1=2$
- $6 - 2^2 - 2^1 = 0$ !

$$\begin{array}{cccc} 0 & 1 & 1 & 0 \\ \hline 2^3 & 2^2 & 2^1 & 2^0 \\ \hline \end{array} \\ = 0*8 + 1*4 + 1*2 + 0*1 = 6$$

# 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?

# Byte Values

- What is the minimum and maximum base-10 value a single byte (8 bits) can store?      **minimum = 0**      **maximum = ?**

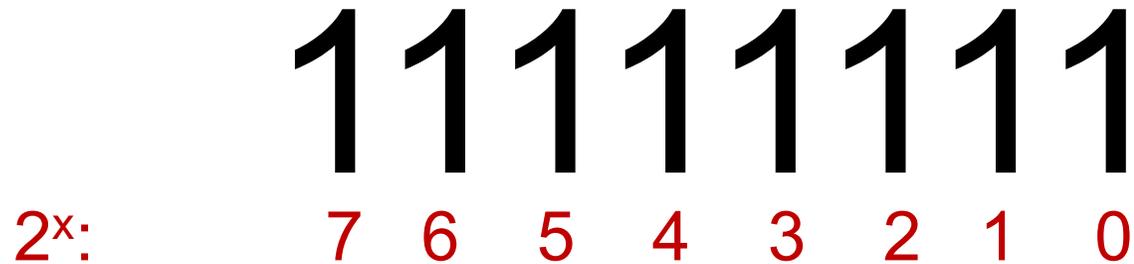
# Byte Values

- What is the minimum and maximum base-10 value a single byte (8 bits) can store?      **minimum = 0**      **maximum = ?**

2<sup>x</sup>:      1 1 1 1 1 1 1 1  
            7 6 5 4 3 2 1 0

# Byte Values

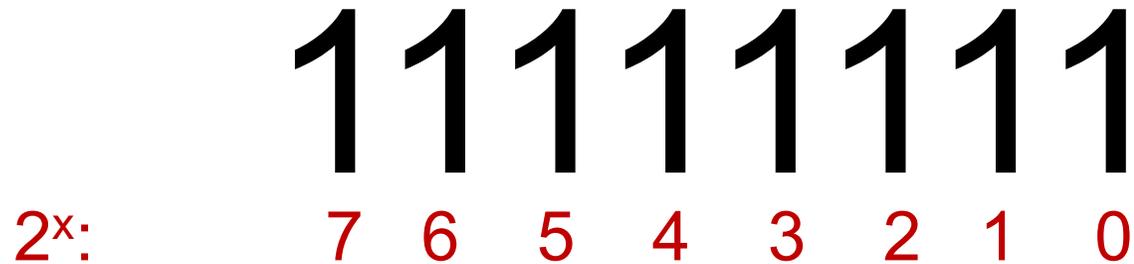
- What is the minimum and maximum base-10 value a single byte (8 bits) can store?      **minimum = 0**      **maximum = ?**



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

# Byte Values

- What is the minimum and maximum base-10 value a single byte (8 bits) can store?      **minimum = 0**      **maximum = 255**



- **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

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

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

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

# Dividing by Base

$$1450 / 10 = 145$$

$$1100_2 / 2 = 110$$

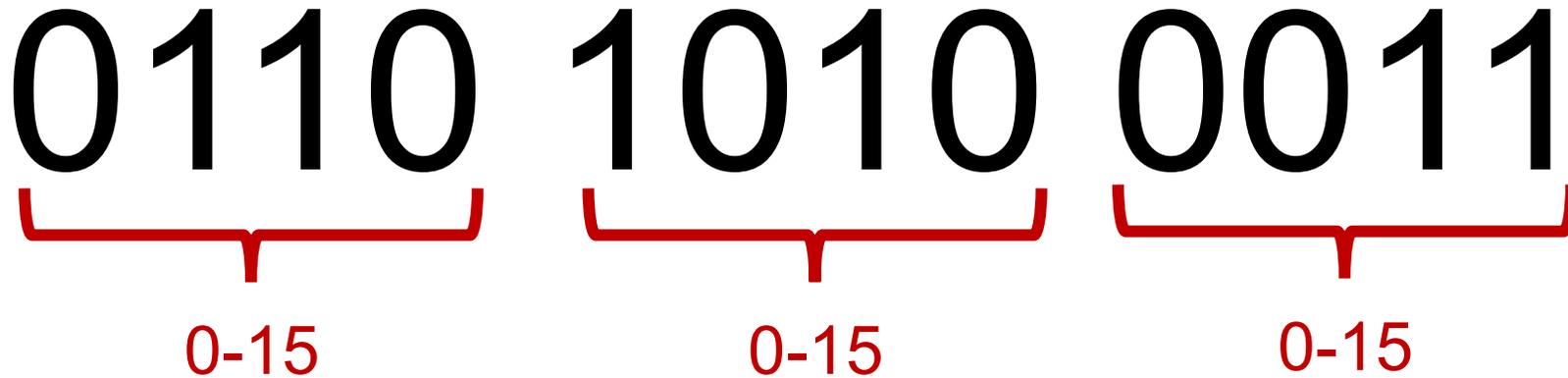
*Key Idea:* removing 0 at the end divides by the base!

# Lecture Plan

• Bits and Bytes	5
• <b>Hexadecimal</b>	33
• Integer Representations	41
• Unsigned Integers	47
• Signed Integers	51
• Overflow	77
• Casting and Combining Types	90
• Live Session	120

# 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**

0x f 5  
1111 0101

The diagram illustrates the conversion of the hexadecimal number 0xf5 to binary. The prefix '0x' is shown in black. The hexadecimal digits 'f' and '5' are shown in black. Below 'f' is a red bracket pointing to the binary string '1111'. Below '5' is a red bracket pointing to the binary string '0101'.

# Practice: Hexadecimal to Binary

What is **0x173A** in binary?

---

<b>Hexadecimal</b>	<b>1</b>	<b>7</b>	<b>3</b>	<b>A</b>
<b>Binary</b>	<b>0001</b>	<b>0111</b>	<b>0011</b>	<b>1010</b>

---

# Practice: Hexadecimal to Binary

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

---

<b>Binary</b>	<b>11</b>	<b>1100</b>	<b>1010</b>
<b>Hexadecimal</b>	<b>3</b>	<b>C</b>	<b>A</b>

---

# Lecture Plan

• Bits and Bytes	5
• Hexadecimal	33
• <b>Integer Representations</b>	41
• Unsigned Integers	47
• Signed Integers	51
• Overflow	77
• Casting and Combining Types	90
• Live Session	120

# 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,  $1.5 \times 10^{12}$ )

# 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,  $1.5 \times 10^{12}$ )
  - ↳ **Look up IEEE floating point if you're interested!**

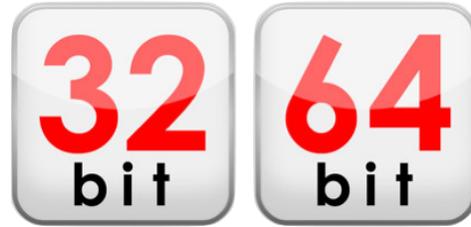
# Number Representations

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

# In The Days Of Yore...

C Declaration	Size (Bytes)
<code>int</code>	4
<code>double</code>	8
<code>float</code>	4
<code>char</code>	1
<code>char *</code>	4
<code>short</code>	2
<code>long</code>	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 GB** of memory (RAM)!

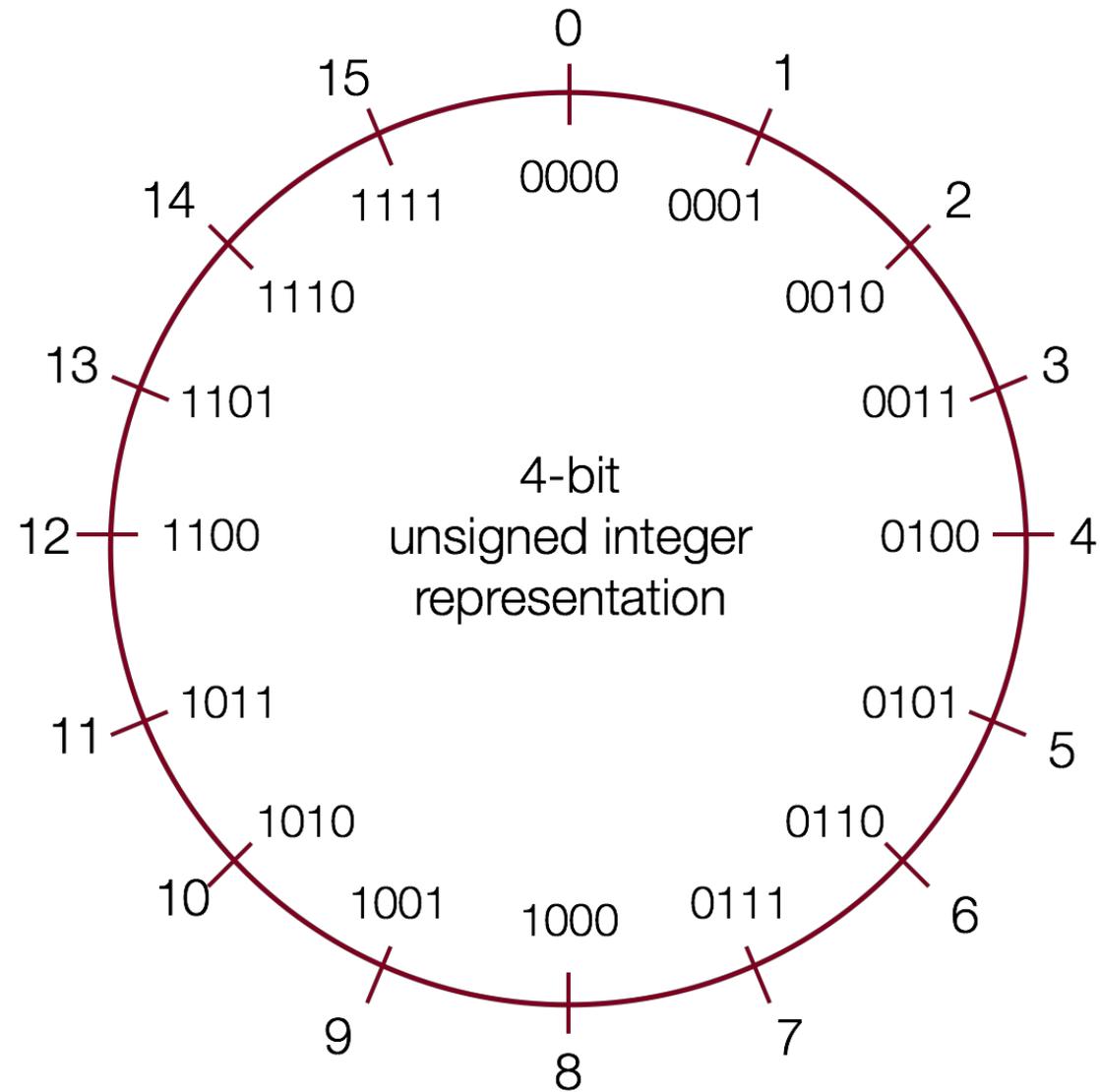
# Lecture Plan

• Bits and Bytes	5
• Hexadecimal	33
• Integer Representations	41
• Unsigned Integers	47
• <b>Signed Integers</b>	51
• Overflow	77
• Casting and Combining Types	90
• Live Session	120

# 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  $0 \rightarrow 2^w - 1$ , 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



# Let's Take A Break

To ponder during the break:

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	5
• Hexadecimal	33
• Integer Representations	41
• Unsigned Integers	47
• <b>Signed Integers</b>	51
• Overflow	77
• Casting and Combining Types	90
• Live Session	120

# 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?

# 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

0 1 1 0  
└─┘ └──────────┘  
positive 6

1 0 1 1  
└─┘ └──────────┘  
negative 3

# Sign Magnitude Representation

0000  
positive 0

1000  
negative 0



# Sign Magnitude Representation

$$1\ 000 = -0 \quad 0\ 000 = 0$$

$$1\ 001 = -1 \quad 0\ 001 = 1$$

$$1\ 010 = -2 \quad 0\ 010 = 2$$

$$1\ 011 = -3 \quad 0\ 011 = 3$$

$$1\ 100 = -4 \quad 0\ 100 = 4$$

$$1\ 101 = -5 \quad 0\ 101 = 5$$

$$1\ 110 = -6 \quad 0\ 110 = 6$$

$$1\ 111 = -7 \quad 0\ 111 = 7$$

- 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:**  $\pm 0$  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.

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

# A Better Idea

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

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

# A Better Idea

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

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

# A Better Idea

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

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

# 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.

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

# 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...

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

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

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

- 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.

---

$$\begin{array}{r} 0101 \\ + 1010 \\ \hline 1111 \end{array}$$

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

---

$$\begin{array}{r} 1111 \\ + 0001 \\ \hline 0000 \end{array}$$

# 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.

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

# 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.

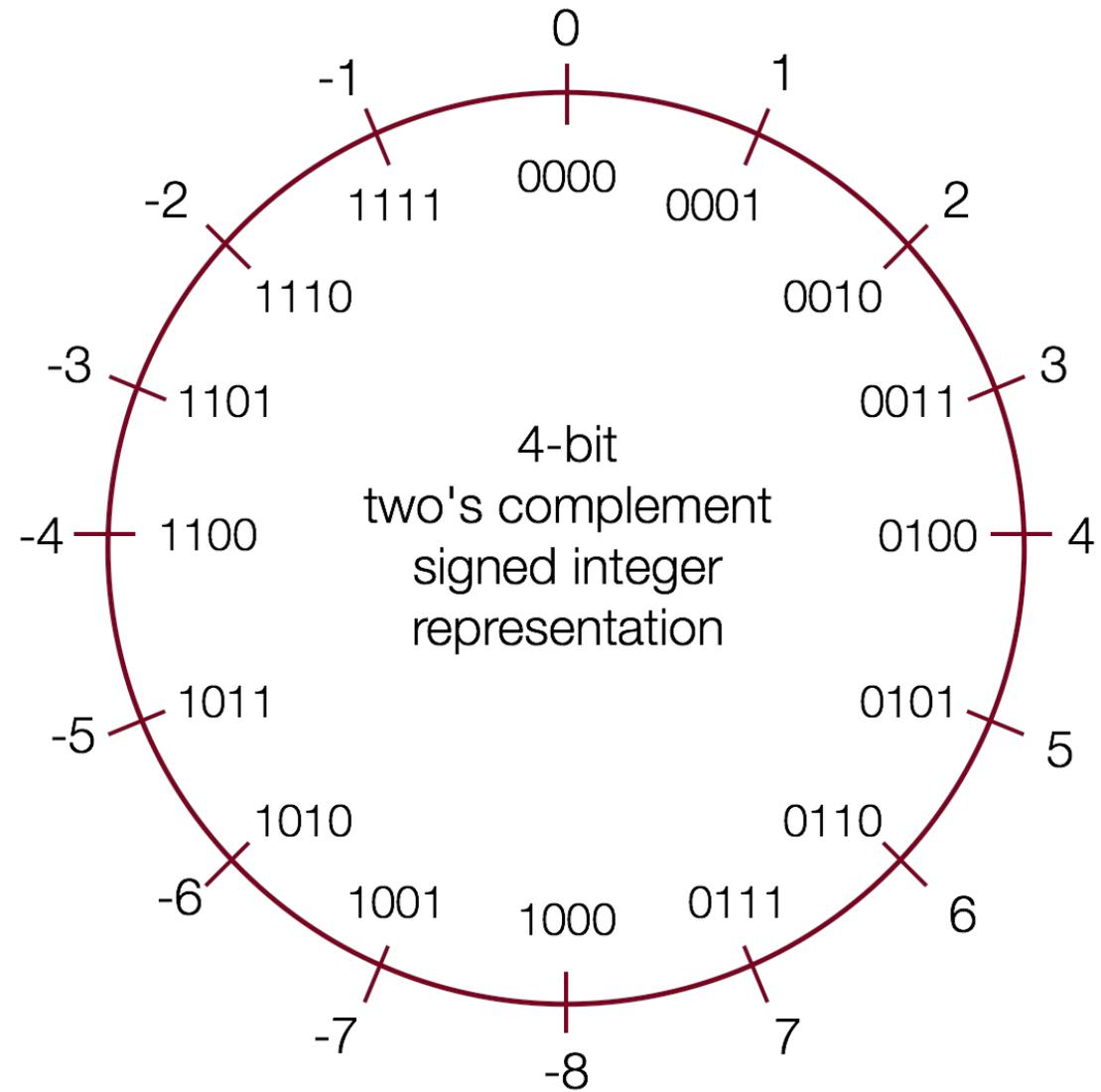
$$\begin{array}{r} 100100 \\ + \color{red}{???100} \\ \hline 000000 \end{array}$$

# 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.

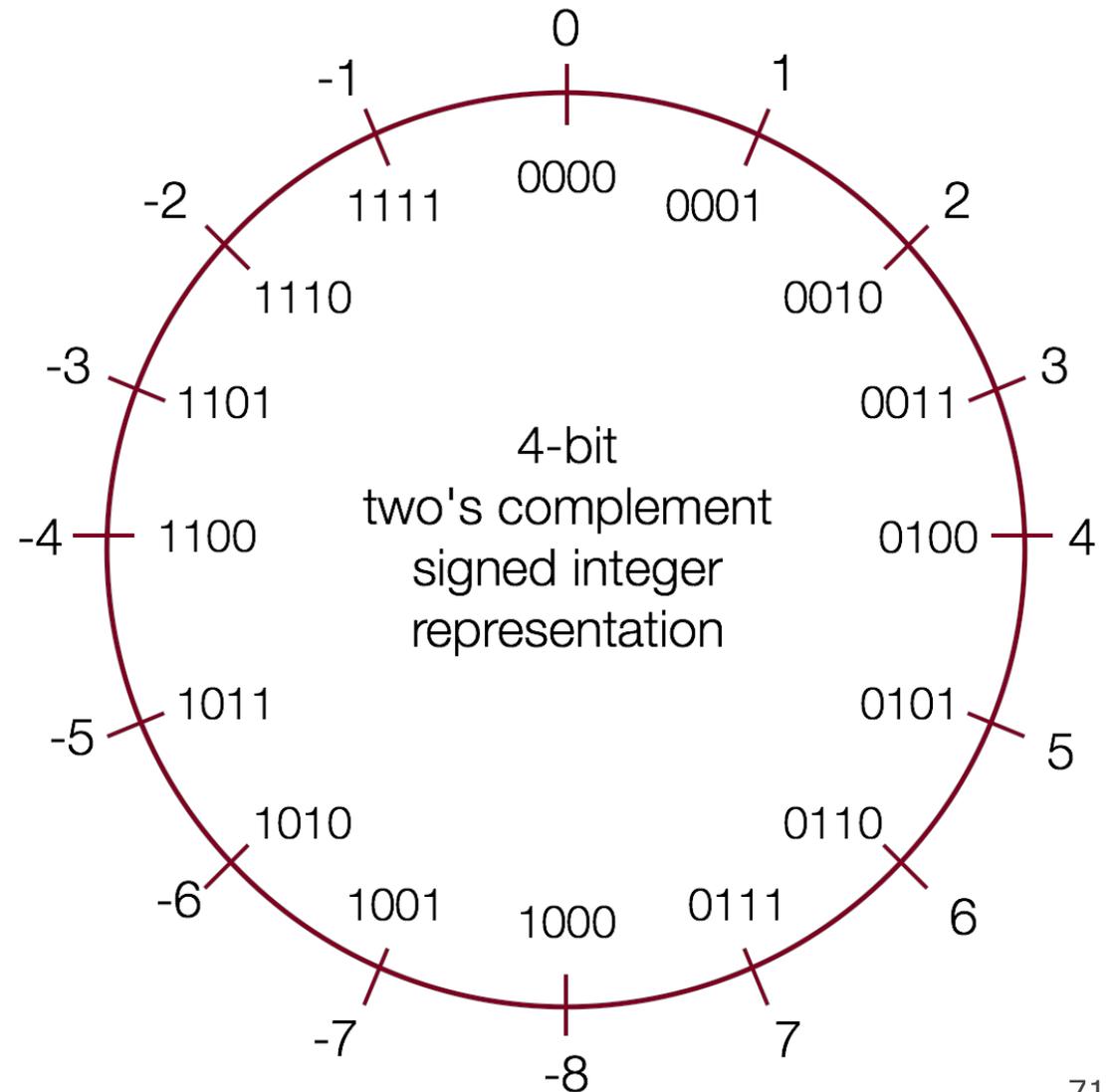
$$\begin{array}{r} 100100 \\ + 011100 \\ \hline 000000 \end{array}$$

# Two's Complement



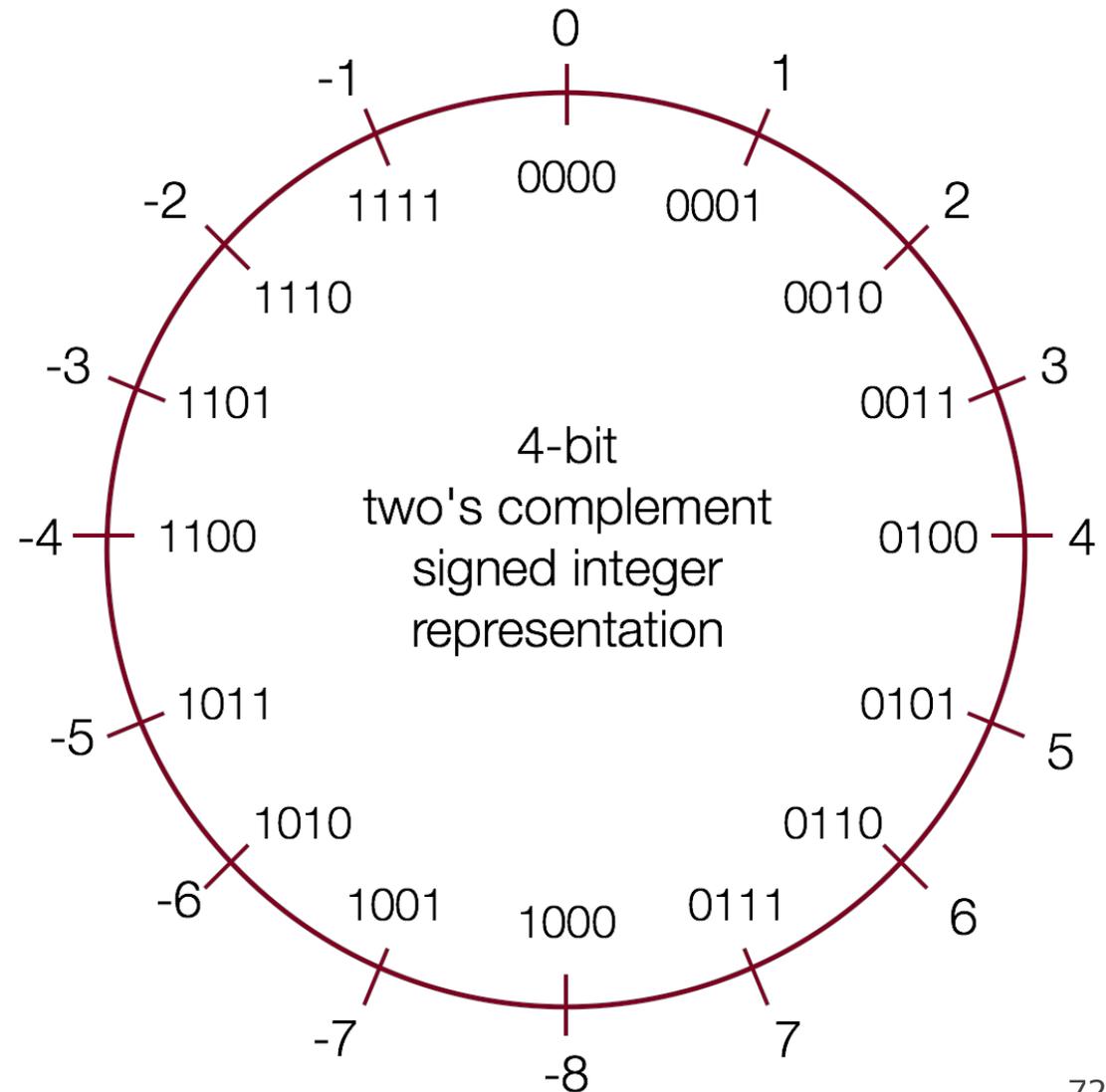
# Two's Complement

- 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!



# 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  $2 + -5$ ?

$$\begin{array}{r} 0010 \\ +1011 \\ \hline 1101 \end{array}$$

2  
-5  
-3

# Two's Complement

- Subtracting two numbers is just performing the two's complement on one of them and then adding. E.g.  $4 - 5 = -1$ .

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

# Practice: Two's Complement

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

a) -4 (1100)

b) 7 (0111)

c) 3 (0011)

d) -8 (1000)

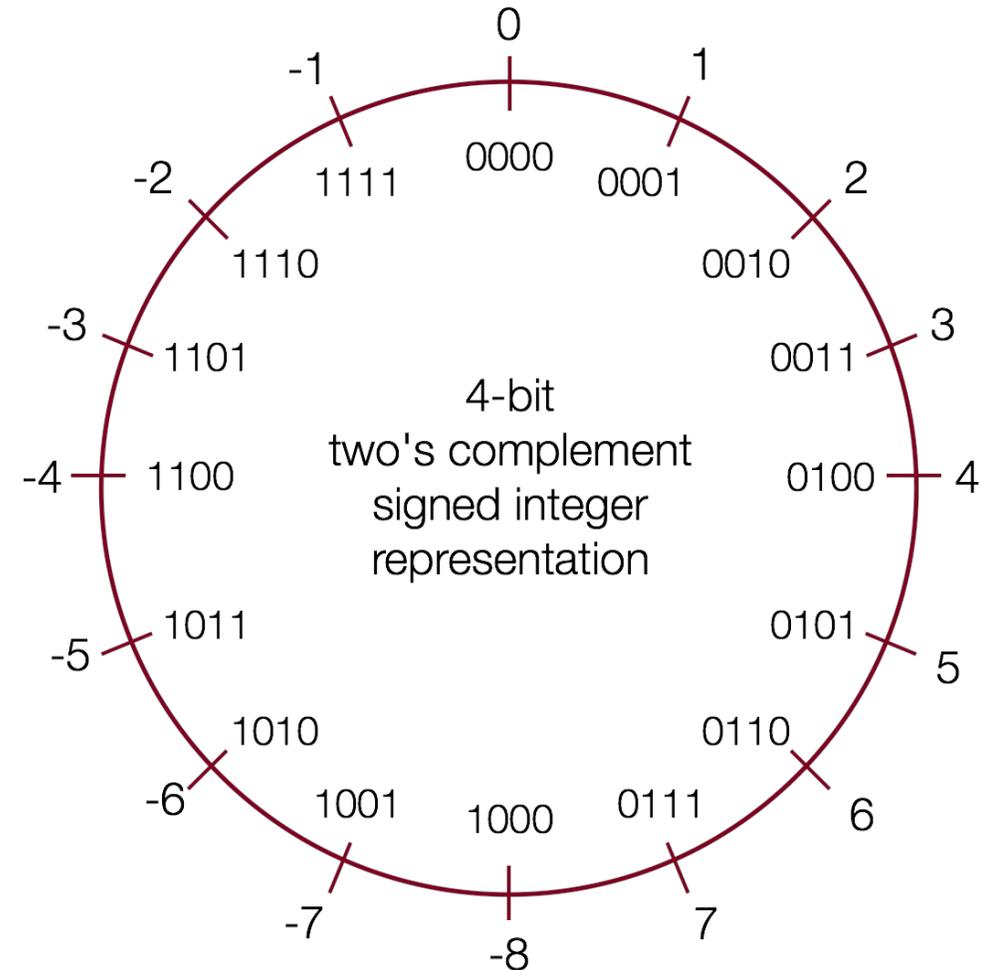
# Practice: Two's Complement

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

a) -4 (1100)

b) 7 (0111)

c) 3 (0011)



# Lecture Plan

• Bits and Bytes	5
• Hexadecimal	33
• Integer Representations	41
• Unsigned Integers	47
• Signed Integers	51
• <b>Overflow</b>	77
• Casting and Combining Types	90
• Live Session	120

# Overflow

- If you exceed the **maximum** value of your bit representation, you *wrap around* or *overflow* back to the **smallest** bit representation.

$$0b1111 + 0b1 = 0b0000$$

- If you go below the **minimum** value of your bit representation, you *wrap around* or *overflow* back to the **largest** bit representation.

$$0b0000 - 0b1 = 0b1111$$

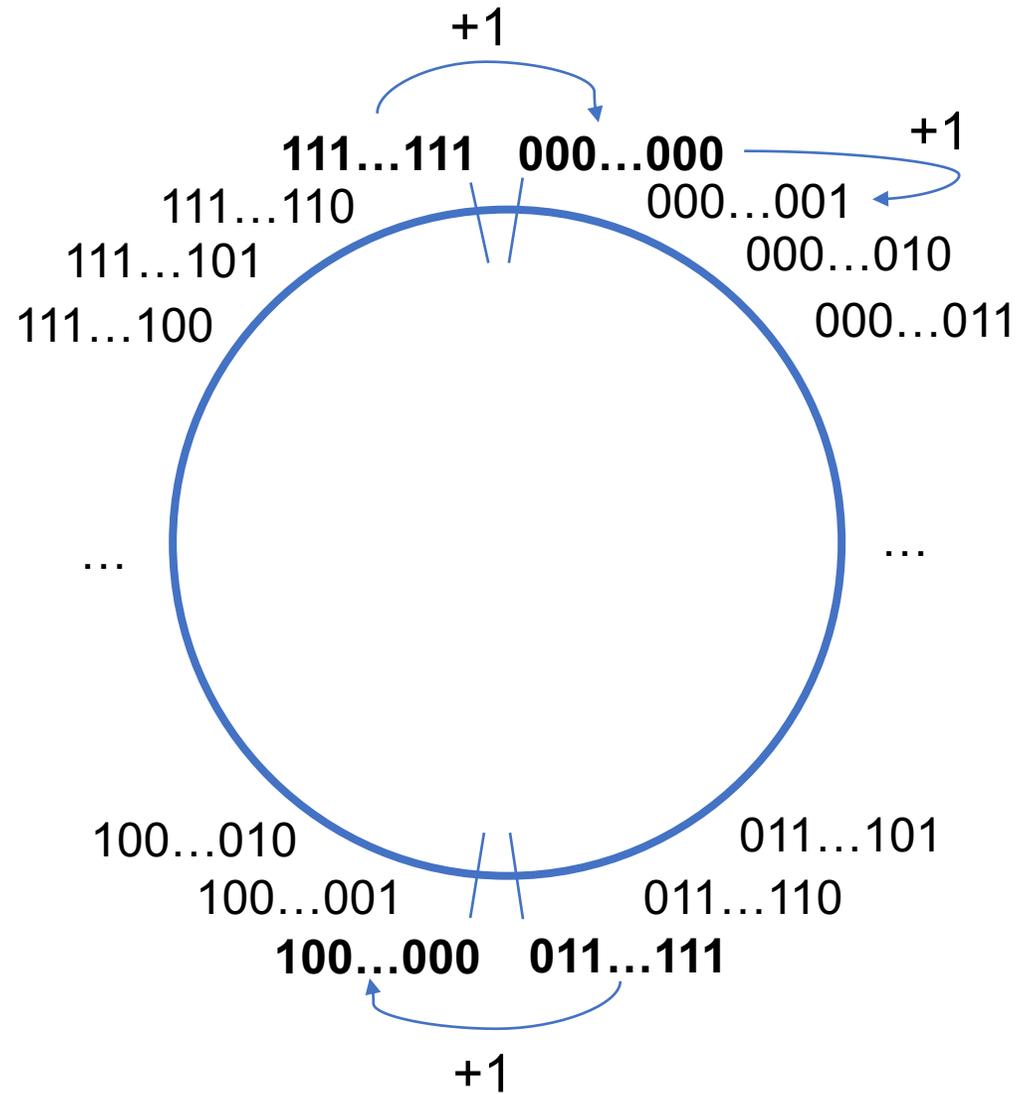
# Min and Max Integer Values

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

`INT_MIN, INT_MAX, UINT_MAX, LONG_MIN, LONG_MAX,  
ULONG_MAX, ...`

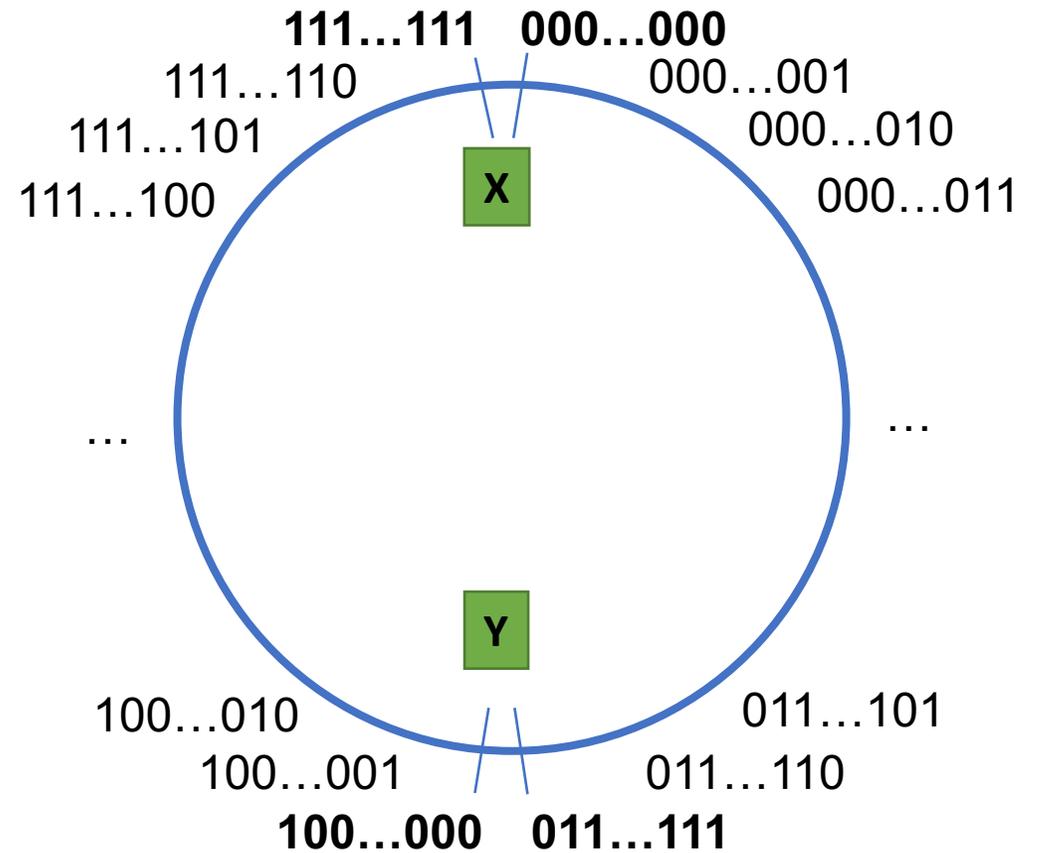
# Overflow



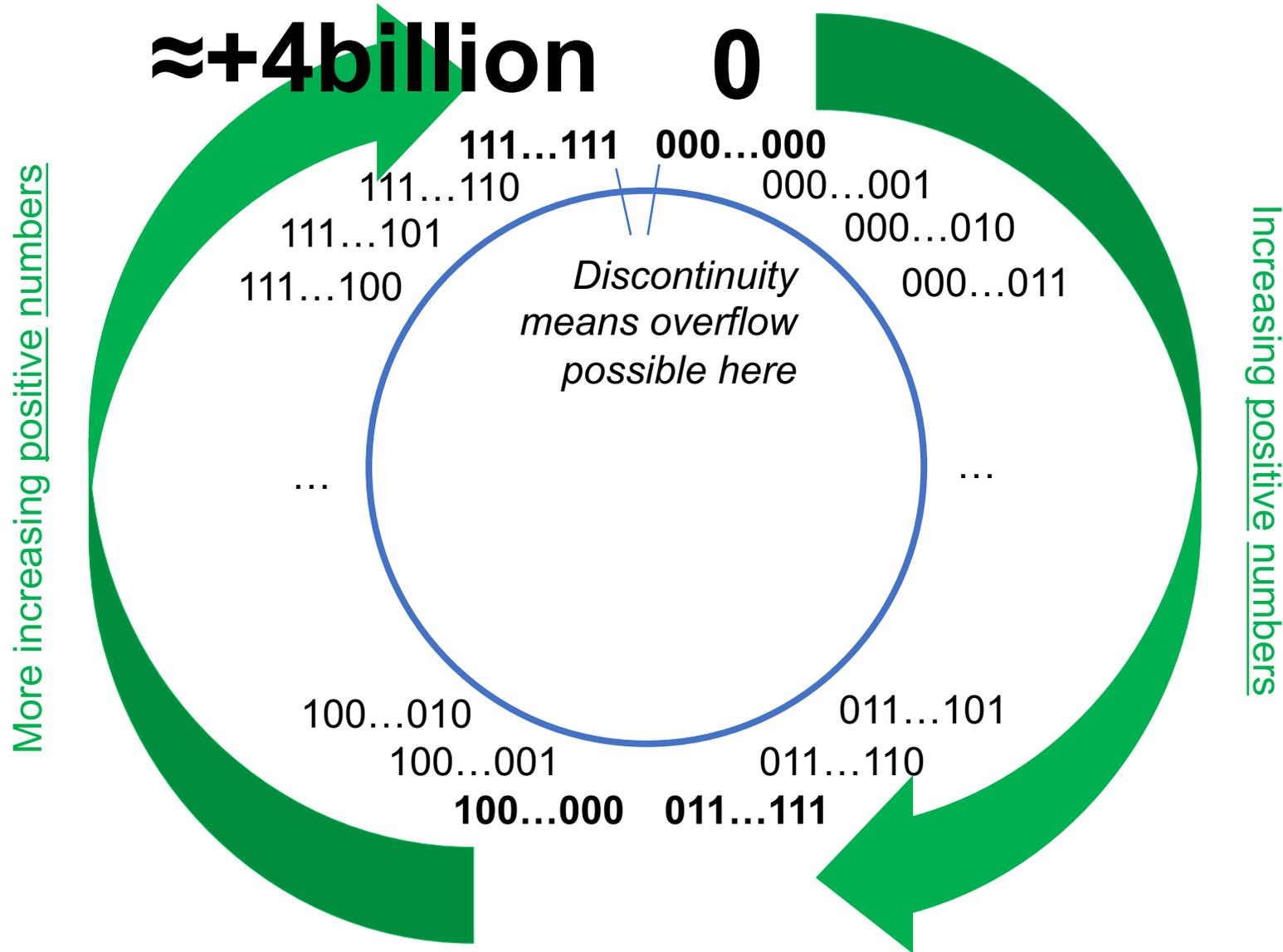
# Overflow

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

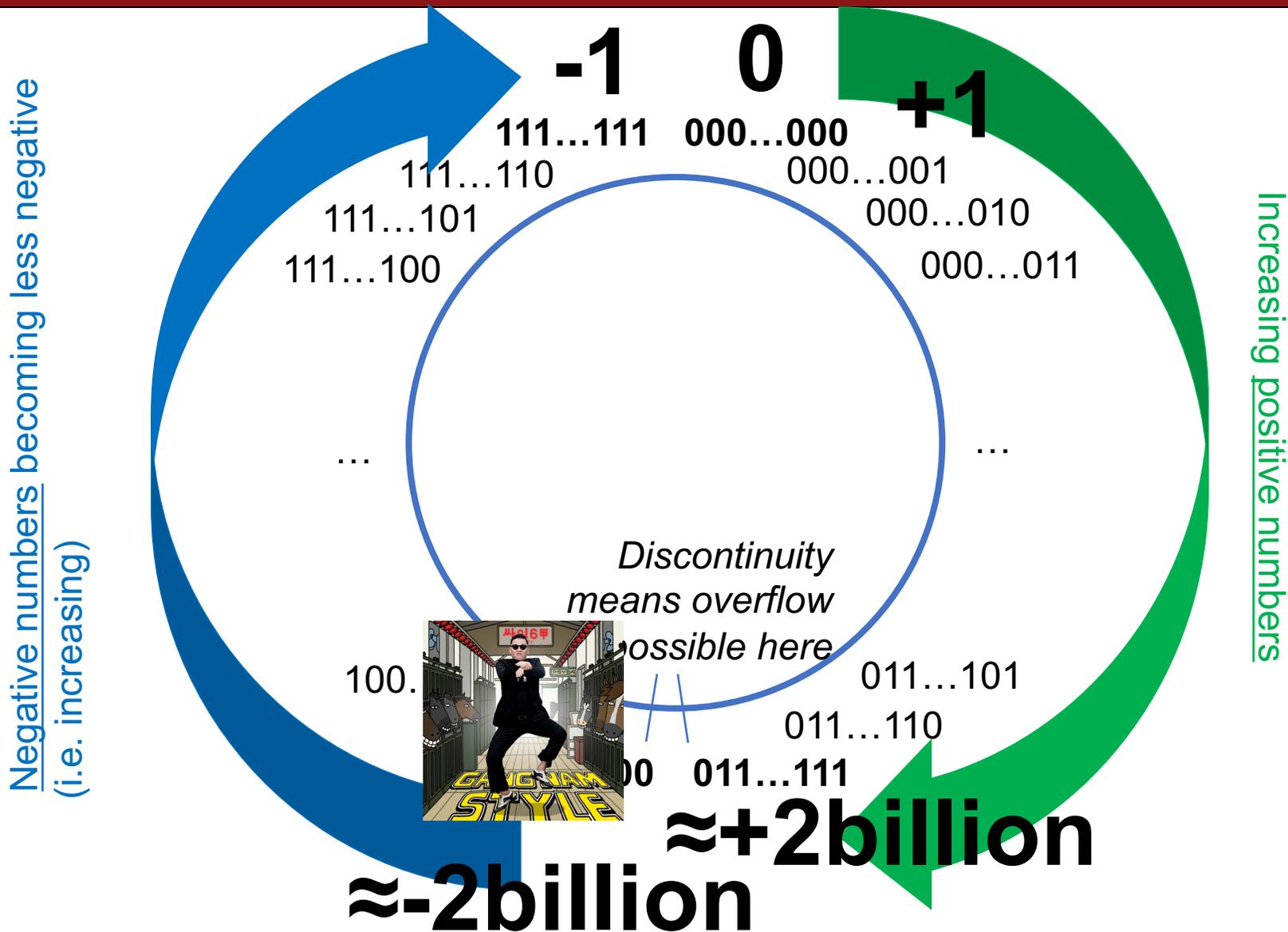
- A. Signed and unsigned can both overflow at points X and Y
- B. Signed can overflow only at X, unsigned only at Y
- C. Signed can overflow only at Y, unsigned only at X
- D. Signed can overflow at X and Y, unsigned only at X
- E. Other



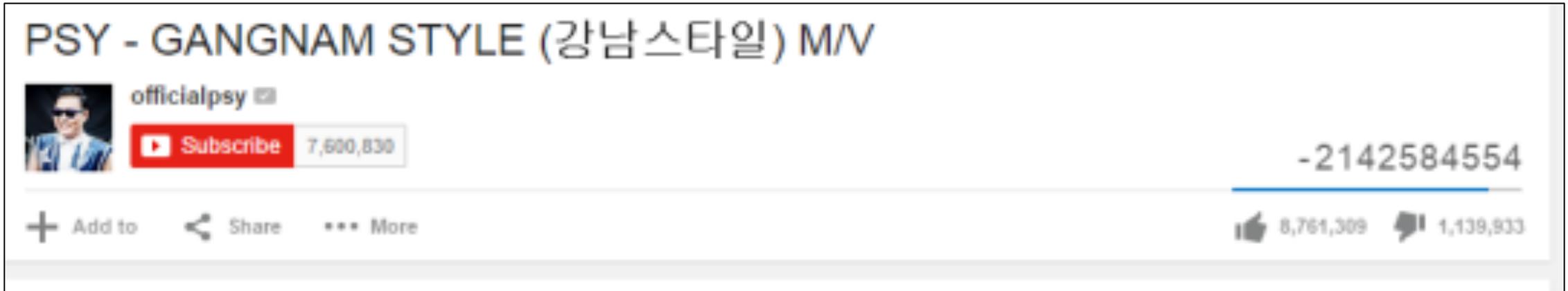
# Unsigned Integers



# Signed Numbers



# Overflow In Practice: PSY



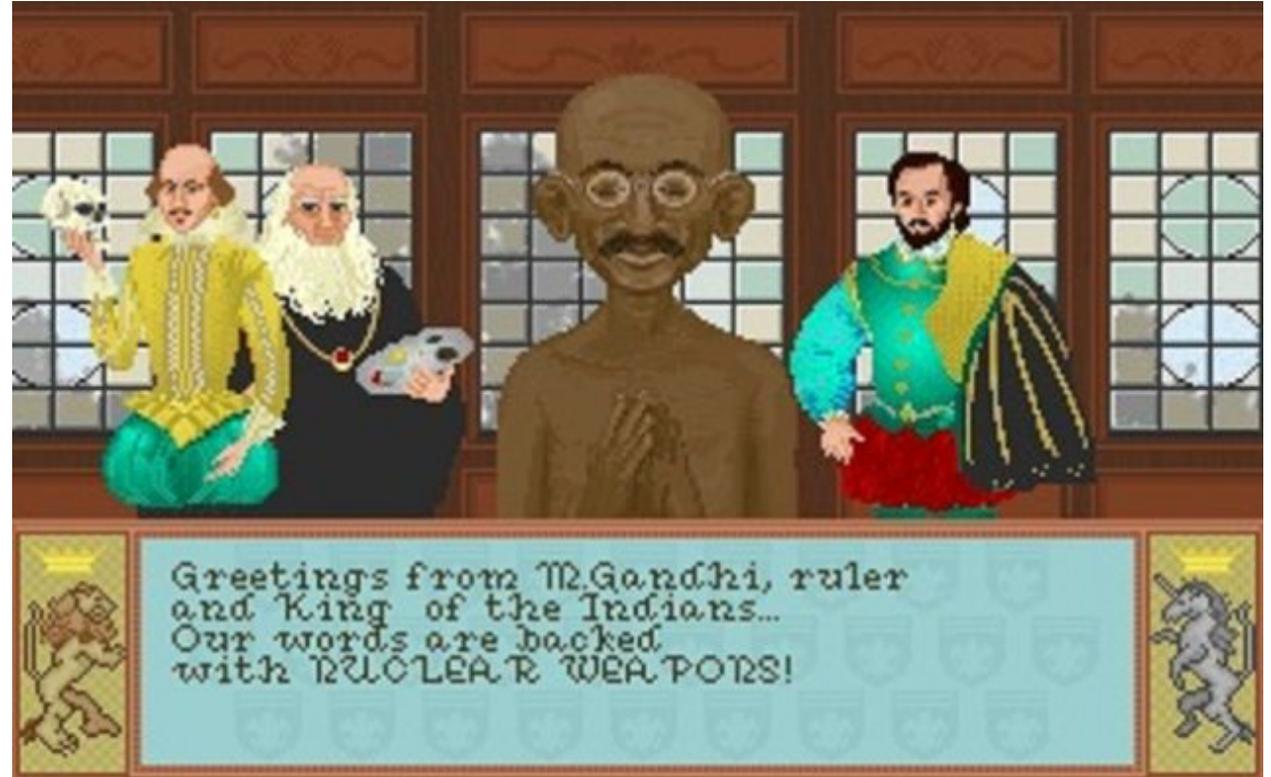
**YouTube:** “We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!”

# 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

# Lecture Plan

• Bits and Bytes	5
• Hexadecimal	33
• Integer Representations	41
• Unsigned Integers	47
• Signed Integers	51
• Overflow	77
• <b>Casting and Combining Types</b>	90
• Live Session	120

# printf and Integers

- There are 3 placeholders for 32-bit integers that we can use:
  - %d: signed 32-bit int
  - %u: unsigned 32-bit int
  - %x: hex 32-bit int
- **The placeholder—not the expression filling in the placeholder—dictates what gets printed!**

# Casting

- What happens at the byte level when we cast between variable types? **The bytes remain the same! This means they may be interpreted differently depending on the type.**

```
int v = -12345;  
unsigned int uv = v;  
printf("v = %d, uv = %u\n", v, uv);
```

This prints out: "v = -12345, uv = 4294954951". **Why?**

# Casting

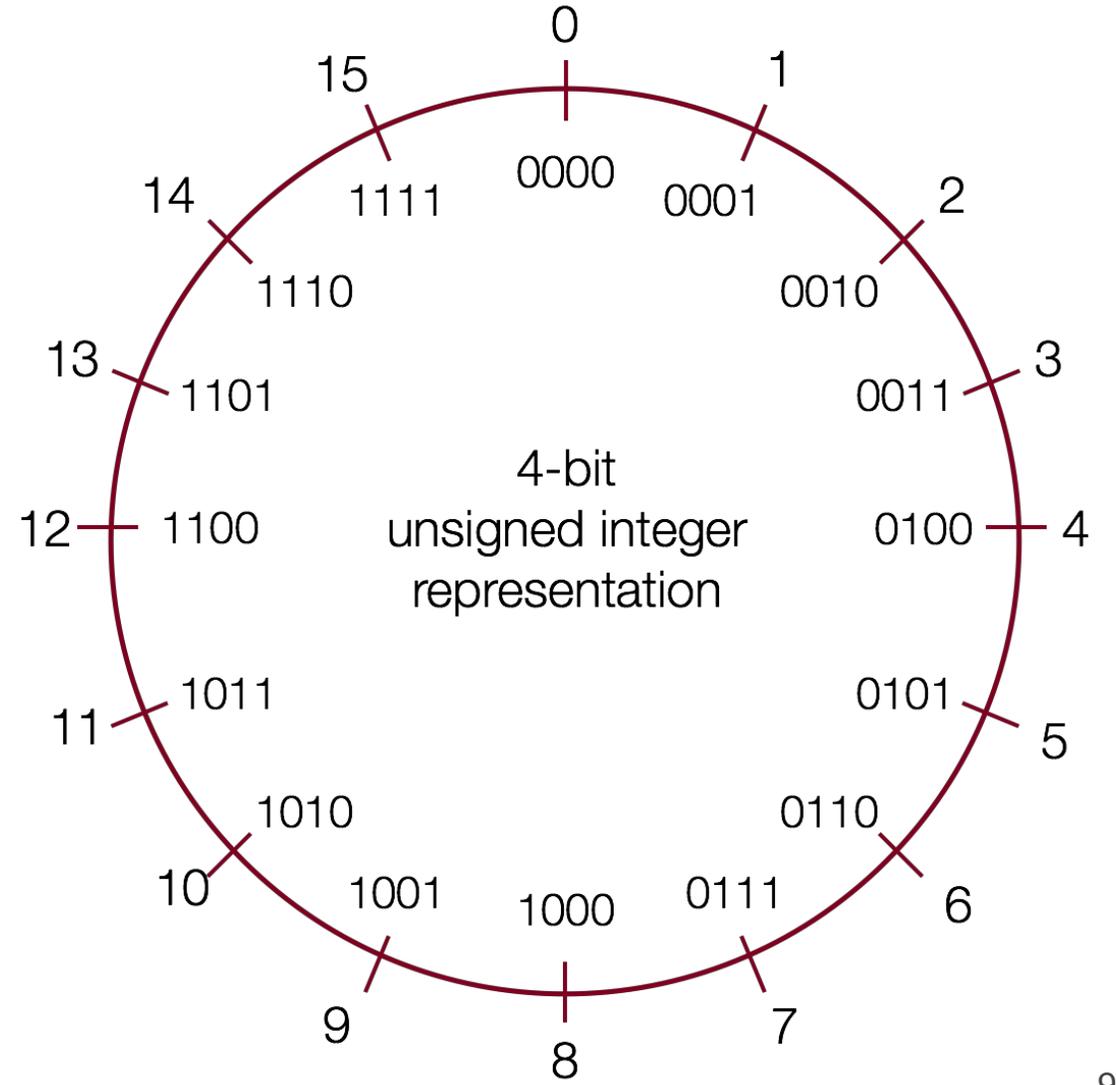
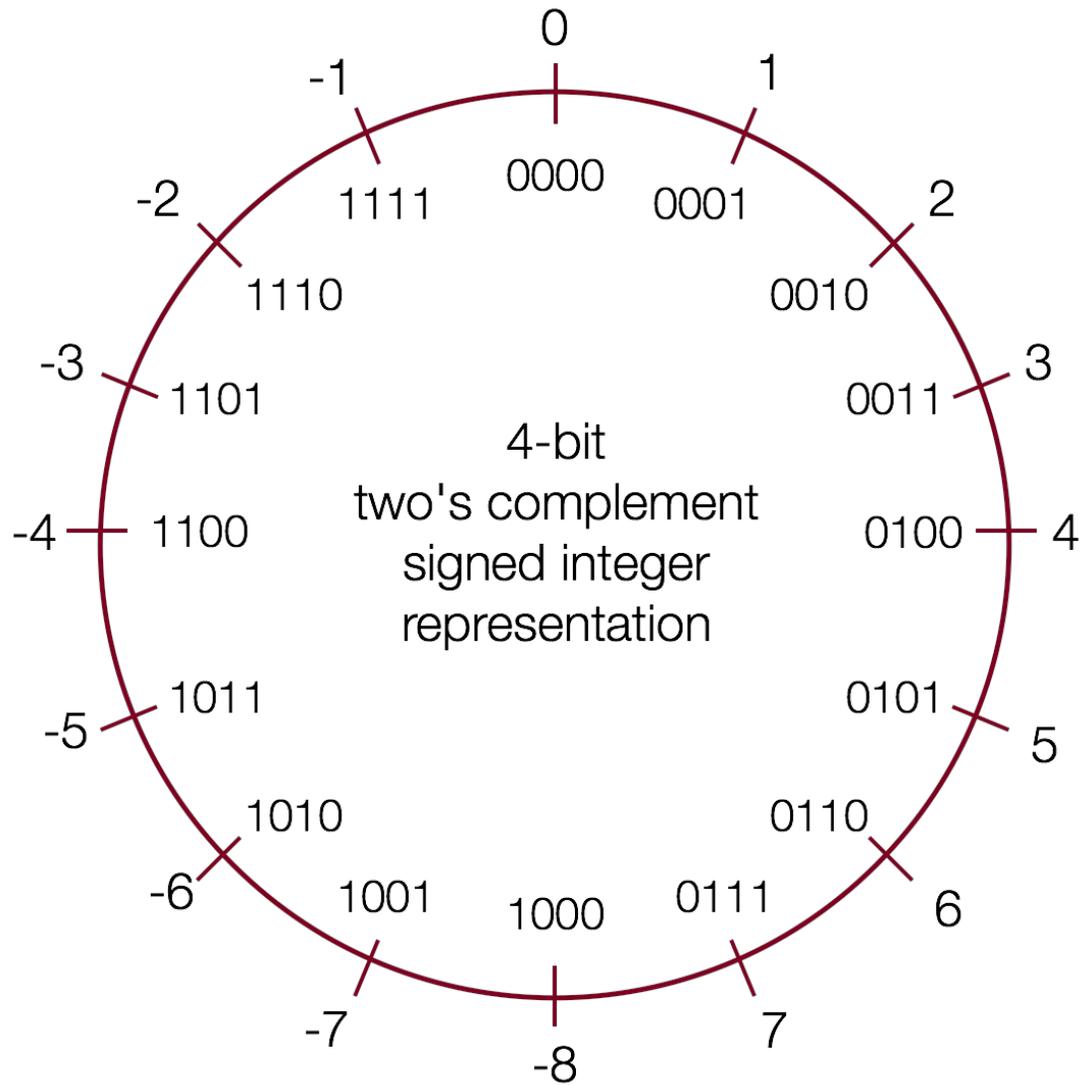
- What happens at the byte level when we cast between variable types? **The bytes remain the same! This means they may be interpreted differently depending on the type.**

```
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

The bit representation for -12345 is  
0b11111111111111111111111100111111000111.

If we treat this binary representation as a positive number, it's *huge*!

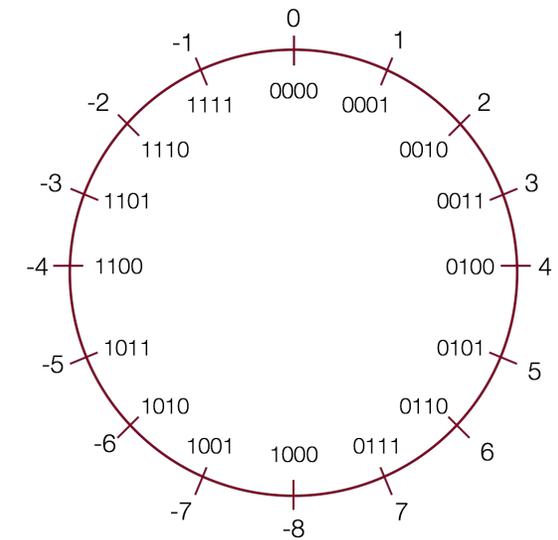
# Casting



# Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

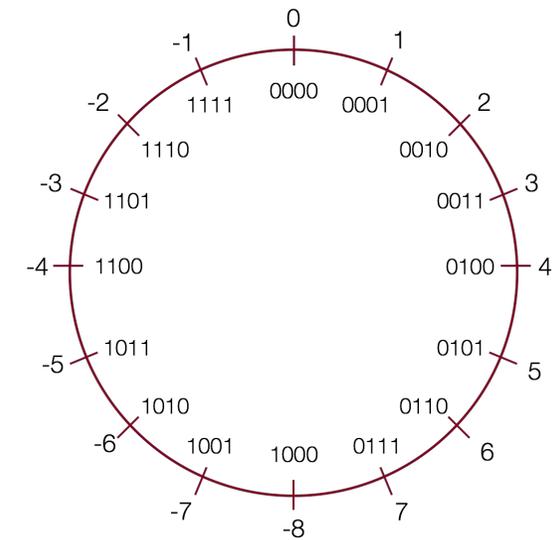
Expression	Type	Evaluation	Correct?
<code>0 == 0U</code>			
<code>-1 &lt; 0</code>			
<code>-1 &lt; 0U</code>			
<code>2147483647 &gt; -</code> <code>2147483647 - 1</code>			
<code>2147483647U &gt; -</code> <code>2147483647 - 1</code>			
<code>2147483647 &gt;</code> <code>(int)2147483648U</code>			
<code>-1 &gt; -2</code>			
<code>(unsigned)-1 &gt; -2</code>			



# Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

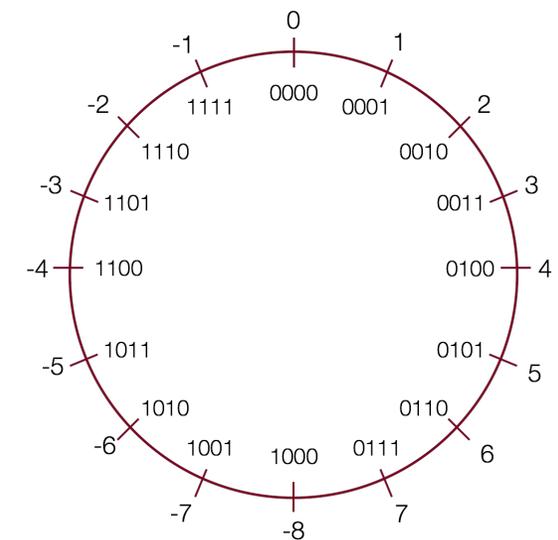
Expression	Type	Evaluation	Correct?
<code>0 == 0U</code>	Unsigned	1	yes
<code>-1 &lt; 0</code>			
<code>-1 &lt; 0U</code>			
<code>2147483647 &gt; -</code> <code>2147483647 - 1</code>			
<code>2147483647U &gt; -</code> <code>2147483647 - 1</code>			
<code>2147483647 &gt;</code> <code>(int)2147483648U</code>			
<code>-1 &gt; -2</code>			
<code>(unsigned)-1 &gt; -2</code>			



# Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

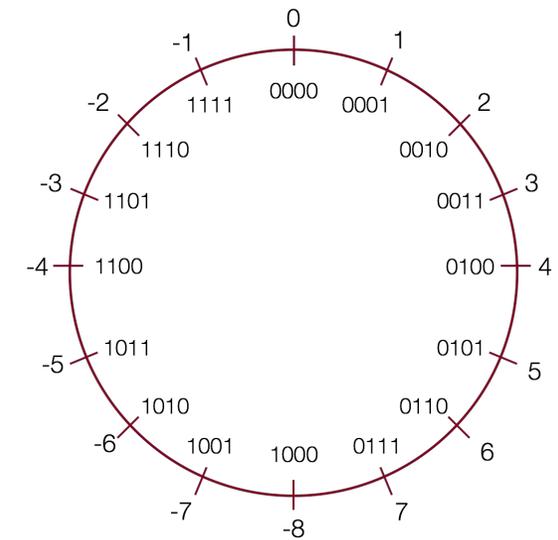
Expression	Type	Evaluation	Correct?
<code>0 == 0U</code>	Unsigned	1	yes
<code>-1 &lt; 0</code>	Signed	1	yes
<code>-1 &lt; 0U</code>			
<code>2147483647 &gt; -</code> <code>2147483647 - 1</code>			
<code>2147483647U &gt; -</code> <code>2147483647 - 1</code>			
<code>2147483647 &gt;</code> <code>(int)2147483648U</code>			
<code>-1 &gt; -2</code>			
<code>(unsigned)-1 &gt; -2</code>			



# Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

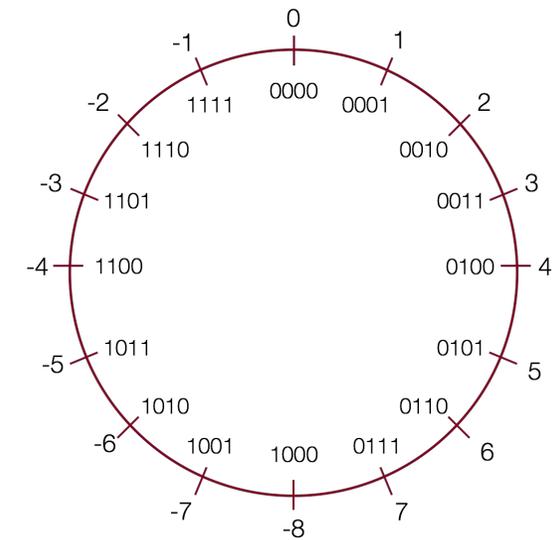
Expression	Type	Evaluation	Correct?
<code>0 == 0U</code>	Unsigned	1	yes
<code>-1 &lt; 0</code>	Signed	1	yes
<code>-1 &lt; 0U</code>	Unsigned	0	No!
<code>2147483647 &gt; -</code> <code>2147483647 - 1</code>			
<code>2147483647U &gt; -</code> <code>2147483647 - 1</code>			
<code>2147483647 &gt;</code> <code>(int)2147483648U</code>			
<code>-1 &gt; -2</code>			
<code>(unsigned)-1 &gt; -2</code>			



# Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

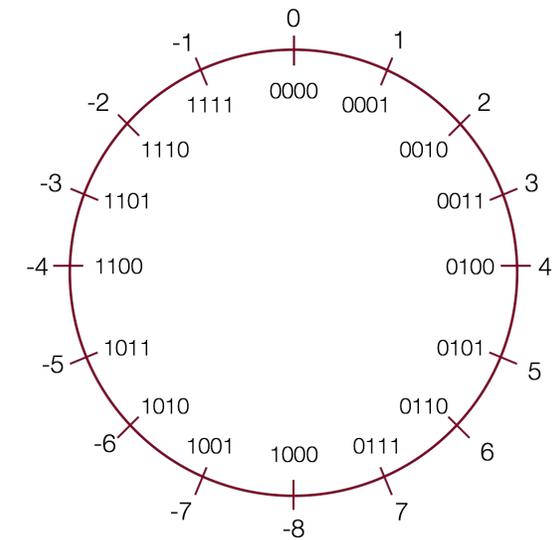
Expression	Type	Evaluation	Correct?
<code>0 == 0U</code>	Unsigned	1	yes
<code>-1 &lt; 0</code>	Signed	1	yes
<code>-1 &lt; 0U</code>	Unsigned	0	No!
<code>2147483647 &gt; -</code> <code>2147483647 - 1</code>	Signed	1	yes
<code>2147483647U &gt; -</code> <code>2147483647 - 1</code>			
<code>2147483647 &gt;</code> <code>(int)2147483648U</code>			
<code>-1 &gt; -2</code>			
<code>(unsigned)-1 &gt; -2</code>			



# Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

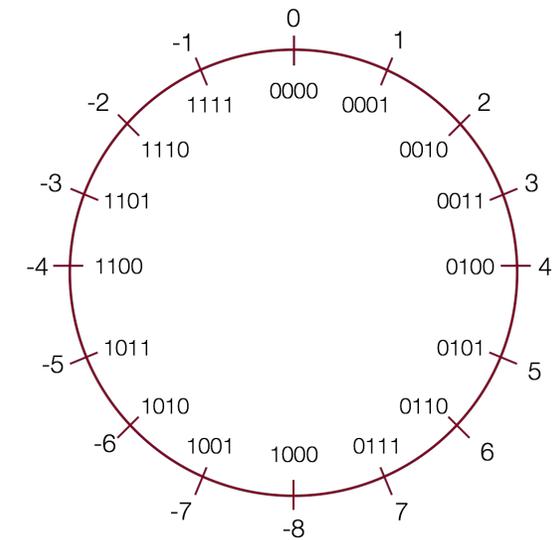
Expression	Type	Evaluation	Correct?
<code>0 == 0U</code>	Unsigned	1	yes
<code>-1 &lt; 0</code>	Signed	1	yes
<code>-1 &lt; 0U</code>	Unsigned	0	No!
<code>2147483647 &gt; -1</code> <code>2147483647 - 1</code>	Signed	1	yes
<code>2147483647U &gt; -1</code> <code>2147483647 - 1</code>	Unsigned	0	No!
<code>2147483647 &gt; (int)2147483648U</code>			
<code>-1 &gt; -2</code>			
<code>(unsigned)-1 &gt; -2</code>			



# Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

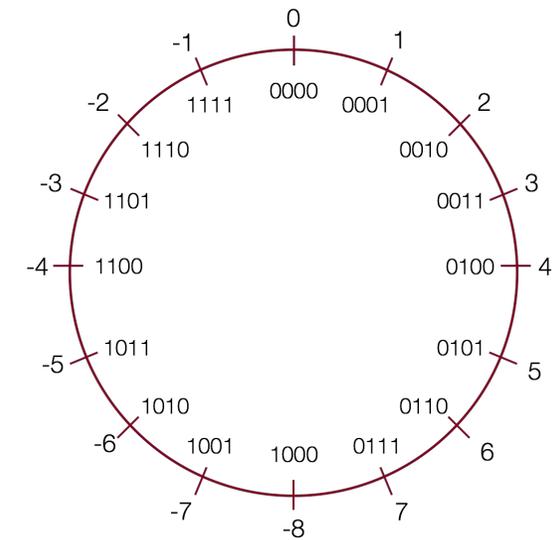
Expression	Type	Evaluation	Correct?
<code>0 == 0U</code>	Unsigned	1	yes
<code>-1 &lt; 0</code>	Signed	1	yes
<code>-1 &lt; 0U</code>	Unsigned	0	No!
<code>2147483647 &gt; -1</code> <code>2147483647 - 1</code>	Signed	1	yes
<code>2147483647U &gt; -1</code> <code>2147483647 - 1</code>	Unsigned	0	No!
<code>2147483647 &gt; (int)2147483648U</code>	Signed	1	No!
<code>-1 &gt; -2</code>			
<code>(unsigned)-1 &gt; -2</code>			



# Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

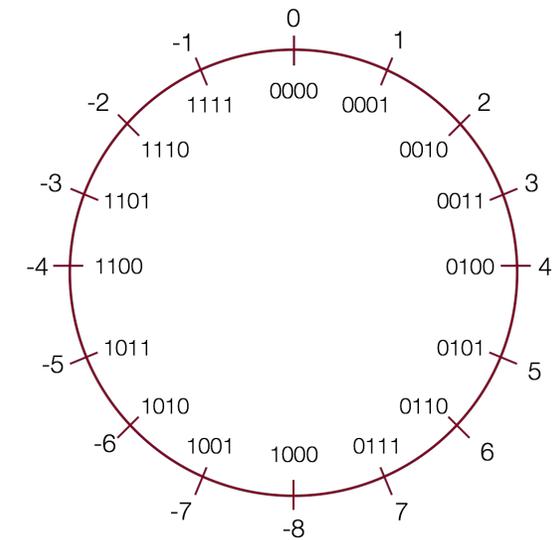
Expression	Type	Evaluation	Correct?
<code>0 == 0U</code>	Unsigned	1	yes
<code>-1 &lt; 0</code>	Signed	1	yes
<code>-1 &lt; 0U</code>	Unsigned	0	No!
<code>2147483647 &gt; -1</code> <code>2147483647 - 1</code>	Signed	1	yes
<code>2147483647U &gt; -1</code> <code>2147483647 - 1</code>	Unsigned	0	No!
<code>2147483647 &gt; (int)2147483648U</code>	Signed	1	No!
<code>-1 &gt; -2</code>	Signed	1	yes
<code>(unsigned)-1 &gt; -2</code>			



# Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

Expression	Type	Evaluation	Correct?
<code>0 == 0U</code>	Unsigned	1	yes
<code>-1 &lt; 0</code>	Signed	1	yes
<code>-1 &lt; 0U</code>	Unsigned	0	No!
<code>2147483647 &gt; -1</code> <code>2147483647 - 1</code>	Signed	1	yes
<code>2147483647U &gt; -1</code> <code>2147483647 - 1</code>	Unsigned	0	No!
<code>2147483647 &gt; (int)2147483648U</code>	Signed	1	No!
<code>-1 &gt; -2</code>	Signed	1	yes
<code>(unsigned)-1 &gt; -2</code>	Unsigned	1	yes



# Comparisons Between Different Types

Which many of the following statements are true? (*assume that variables are set to values that place them in the spots shown*)

**s3 > u3**

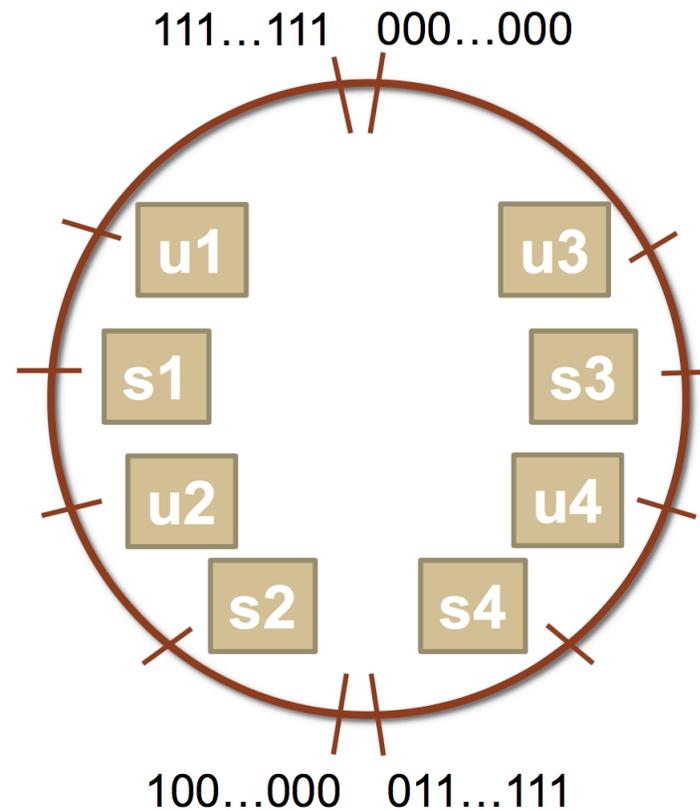
**u2 > u4**

**s2 > s4**

**s1 > s2**

**u1 > u2**

**s1 > u3**



# Comparisons Between Different Types

Which many of the following statements are true? (*assume that variables are set to values that place them in the spots shown*)

**s3 > u3 - true**

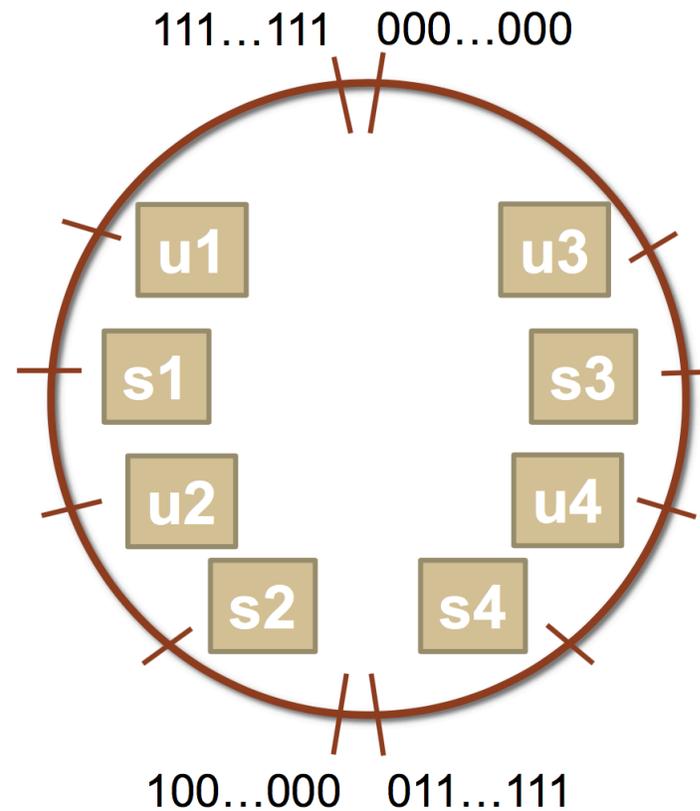
u2 > u4

s2 > s4

s1 > s2

u1 > u2

s1 > u3



# Comparisons Between Different Types

Which many of the following statements are true? (*assume that variables are set to values that place them in the spots shown*)

**s3 > u3 - true**

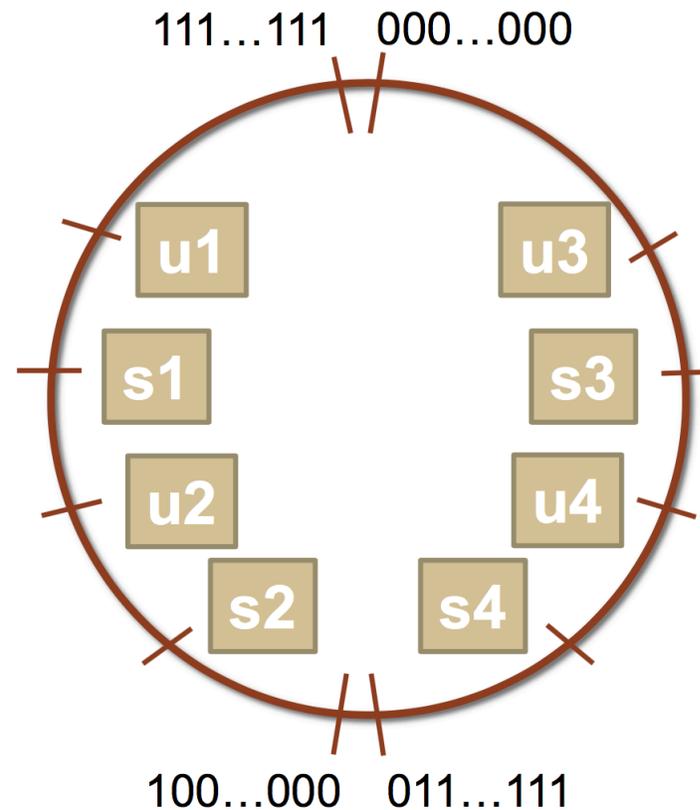
**u2 > u4 - true**

**s2 > s4**

**s1 > s2**

**u1 > u2**

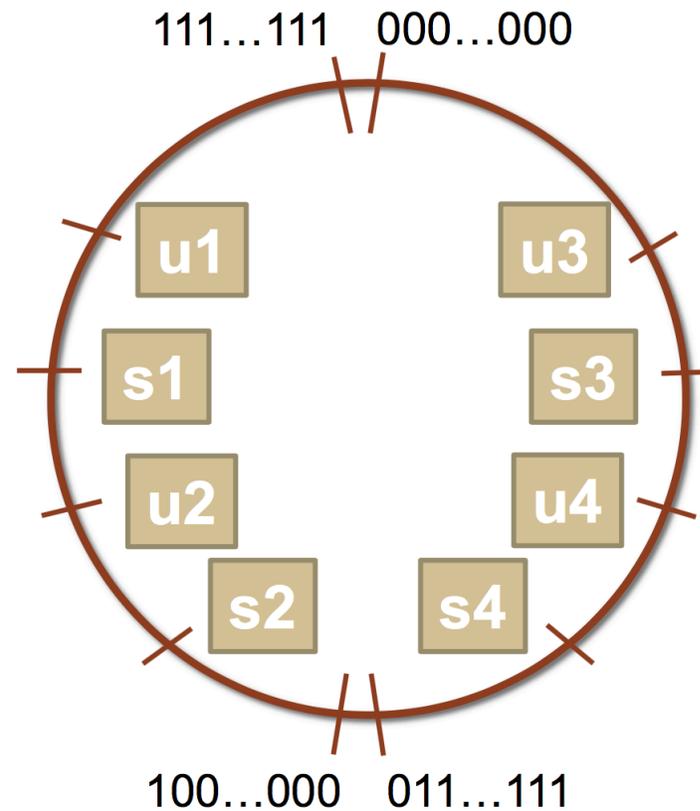
**s1 > u3**



# Comparisons Between Different Types

Which many of the following statements are true? (*assume that variables are set to values that place them in the spots shown*)

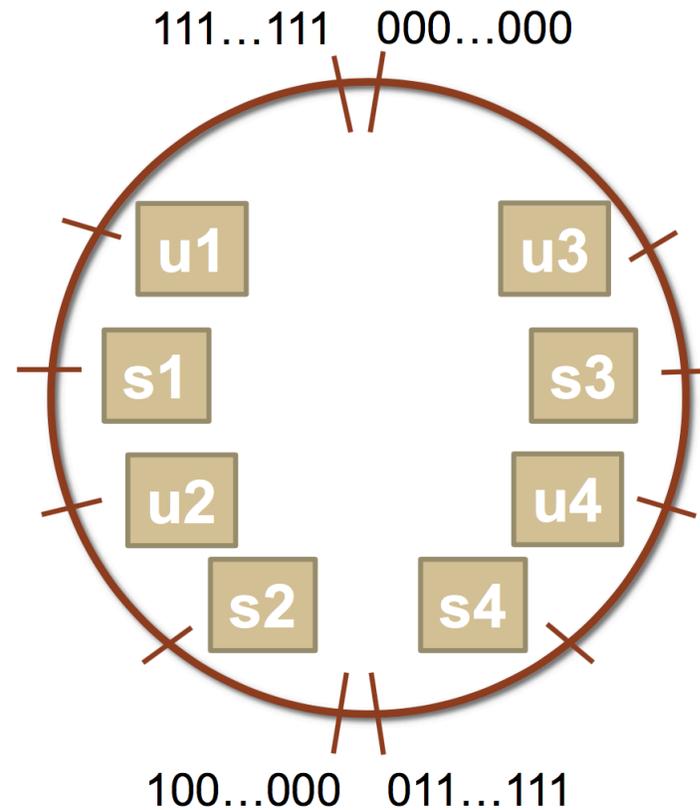
- s3 > u3 - true**
- u2 > u4 - true**
- s2 > s4 - false**
- s1 > s2**
- u1 > u2**
- s1 > u3**



# Comparisons Between Different Types

Which many of the following statements are true? (*assume that variables are set to values that place them in the spots shown*)

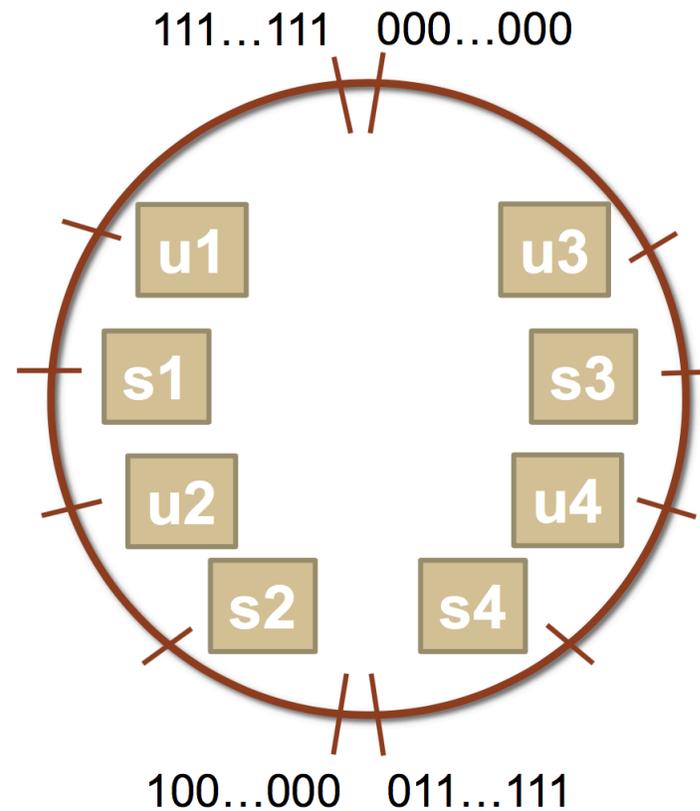
- s3 > u3 - true**
- u2 > u4 - true**
- s2 > s4 - false**
- s1 > s2 - true**
- u1 > u2**
- s1 > u3**



# Comparisons Between Different Types

Which many of the following statements are true? (*assume that variables are set to values that place them in the spots shown*)

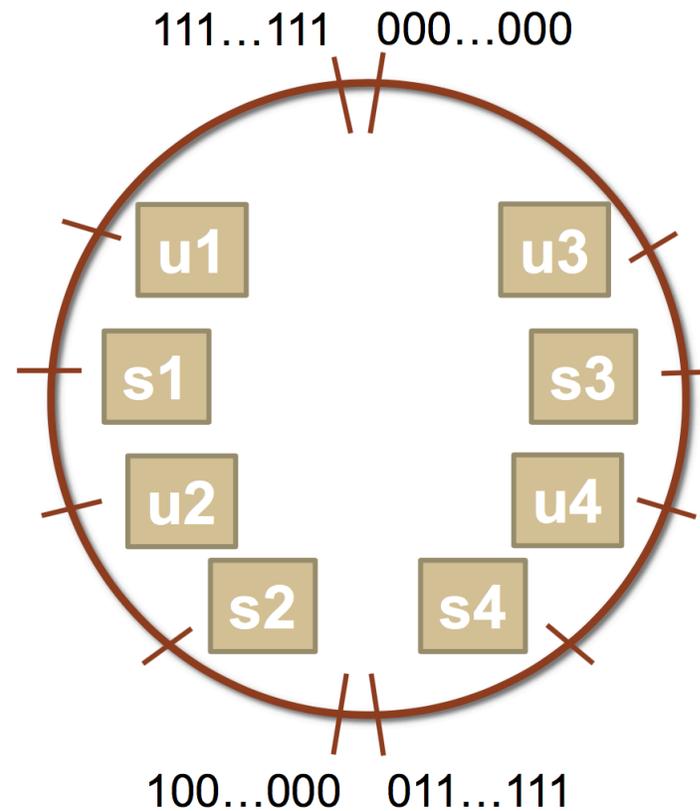
- s3 > u3 - true**
- u2 > u4 - true**
- s2 > s4 - false**
- s1 > s2 - true**
- u1 > u2 - true**
- s1 > u3**



# Comparisons Between Different Types

Which many of the following statements are true? (*assume that variables are set to values that place them in the spots shown*)

- s3 > u3 - true**
- u2 > u4 - true
- s2 > s4 - false
- s1 > s2 - true
- u1 > u2 - true
- s1 > u3 - true**



# Expanding Bit Representations

- Sometimes, we want to convert between two integers of different sizes (e.g. **short** to **int**, or **int** to **long**).
- We might not be able to convert from a bigger data type to a smaller data type, but we do want to always be able to convert from a **smaller** data type to a **bigger** data type.
- For **unsigned** values, we can add *leading zeros* to the representation (“zero extension”)
- For **signed** values, we can *repeat the sign of the value* for new digits (“sign extension”)
- Note: when doing  $<$ ,  $>$ ,  $<=$ ,  $>=$  comparison between different size types, it will *promote to the larger type*.

# Expanding Bit Representation

```
unsigned short s = 4;
```

```
// short is a 16-bit format, so
```

```
s = 0000 0000 0000 0100b
```

```
unsigned int i = s;
```

```
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```

# Expanding Bit Representation

```
short s = 4;  
// short is a 16-bit format, so          s = 0000 0000 0000 0100b
```

```
int i = s;  
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```

— or —

```
short s = -4;  
// short is a 16-bit format, so          s = 1111 1111 1111 1100b
```

```
int i = s;  
// conversion to 32-bit int, so i = 1111 1111 1111 1111 1111 1111 1111 1100b
```

# 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;
int y = sx;
```

What happens here? Let's look at the bits in *x* (a 32-bit int), 53191:

**0000 0000 0000 0000 1100 1111 1100 0111**

When we cast *x* to a short, it only has 16-bits, and *C truncates* the number:

**1100 1111 1100 0111**

This is -12345! And when we cast *sx* back an int, we sign-extend the number.

**1111 1111 1111 1111 1100 1111 1100 0111** // still -12345

# 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;  
int y = sx;
```

What happens here? Let's look at the bits in *x* (a 32-bit int), -3:

**1111 1111 1111 1111 1111 1111 1111 1101**

When we cast *x* to a short, it only has 16-bits, and *C truncates* the number:

**1111 1111 1111 1101**

This is -3! **If the number does fit, it will convert fine.** *y* looks like this:

**1111 1111 1111 1111 1111 1111 1111 1101 // still -3**

# Truncating Bit Representation

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

```
unsigned int x = 128000;  
unsigned short sx = x;  
unsigned int y = sx;
```

What happens here? Let's look at the bits in *x* (a 32-bit unsigned int), 128000:

**0000 0000 0000 0001 1111 0100 0000 0000**

When we cast *x* to a short, it only has 16-bits, and *C truncates* the number:

**1111 0100 0000 0000**

This is 62464! **Unsigned numbers can lose info too.** Here is what *y* looks like:

**0000 0000 0000 0000 1111 0100 0000 0000 // still 62464**

# The sizeof Operator

```
long sizeof(type);
```

```
// Example
```

```
long int_size_bytes = sizeof(int);    // 4
```

```
long short_size_bytes = sizeof(short); // 2
```

```
long char_size_bytes = sizeof(char);  // 1
```

`sizeof` takes a variable type as a parameter and returns the size of that type, in bytes.

# Recap

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types

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

# Recap

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types

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

# Additional Live Session Slides

- exercises
- questions & answers

# Practice: Two's Complement

- Post any questions you want answered today in Ed:  
<https://edstem.org/us/courses/3085/discussion/205856>

Schedule →  
Lecture 2

- While you wait, fill in the below table:

	<code>char x = _____;</code>	<code>char y = -x;</code>
	decimal      binary	decimal      binary
1.		
2.		
3.		
4.		



# Practice: Two's Complement

- Post any questions you want answered today in Ed: <https://edstem.org/us/courses/3085/discussion/205856>

- While you wait, fill in the below table:

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

Handwritten binary calculations for two's complement:

```

    1111 1100
+ 0000 0011
-----
    0000 0100
+ 0000 0100
-----
    0000 0100

    1101 1111
+ 0010 0000
-----
    0001 0000
+ 0001 0001
-----
    0010 0001
  
```

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

# Practice: Two's Complement

- Post any questions you want answered today in Ed:

<https://edstem.org/us/courses/3085/discussion/205856>

0x12    0b11?

- While you wait, fill in the below table:

	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

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

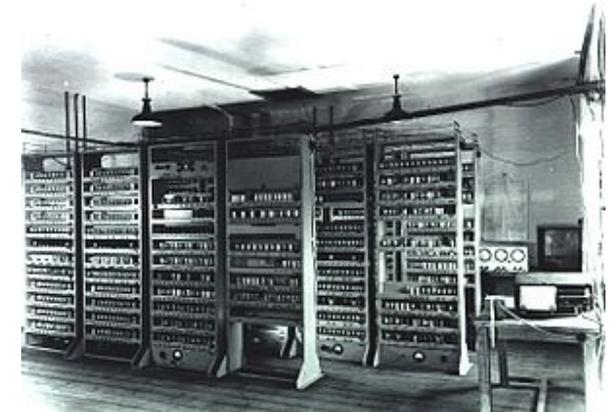
**Question Break!**

# 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



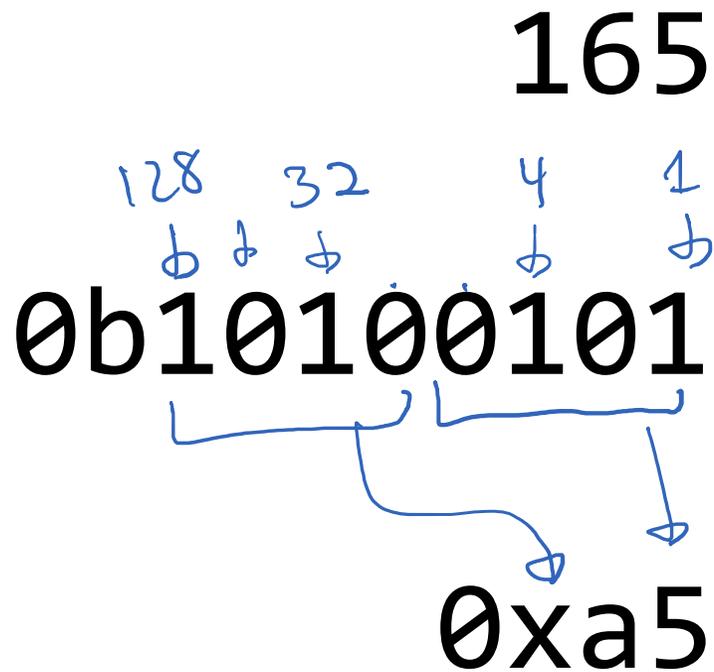
EDSAC (1949)



System/360 (1964)

# Hexadecimal: It's funky but concise

- Let's take a byte (8 bits): *unsigned char* *long?*



Base-10: Human-readable, but cannot easily interpret on/off bits

Base-2: Yes, computers use this, but not human-readable

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)

C: sizeof:  
→ #B

# Breakout: Hexadecimal

1. Introduce yourself: Name, and coolest fact you learned in this unit

2. What is the smallest type that can hold the following hex numbers?

i. 0xfaca

ii. 0x0a

iii. 0xdec1de

iv. 0xc0ffecaca0

Options: A) char 1 B) short 2  
C) int 4 D) long 8

3. For each initialization of x, what will be printed?

i. x = 130; // 0x82

ii. x = -132; // 0xff7c

iii. x = 25; // 0x19

```
short x = _____;  
char cx = x;  
printf("%d", cx);
```



# Hexadecimal

1. Introduce yourself: Name, and coolest fact you learned in this past lecture

2. What is the smallest type that can hold the following hex numbers?

- i. `0xface` `0000 1000` **short**
- ii. `0x0a` `01010` `0xa` **char**
- iii. `0xdec1de` **int**
- iv. `0xc0ffeecacaca0` **long**

Options: A) char 1    B) short 2  
 C) int 4    D) long 8

3. For each initialization of x, what will be printed?

- i. `x = 130; // 0x82` `0b1000 0010`  
`0x00 82` `0111 1110`  
`+126`
- ii. `x = -132; // 0xff7c`
- iii. `x = 25; // 0x19`

```

short x = _____; ←
char cx = x; // -126
printf("%d", cx);
```

`%u` unsigned int

# Hexadecimal

3. For each initialization of x, what will be printed?

```

short x = _____;
char cx = x;
printf("%d", cx);
    
```

**-126** i. x = 130; // 0x82

short: 0x0082  
char: 0x82

// print signed "number"

**124** ii. x = -132; // 0xff7c

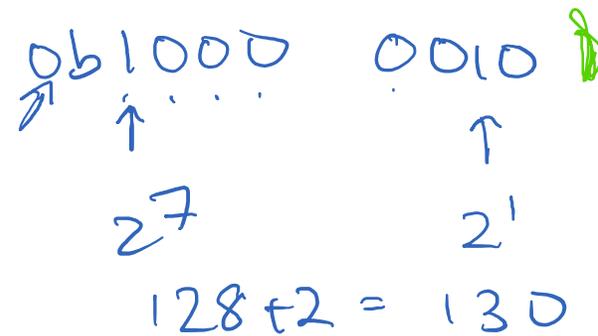
short: 0xff7c  
char: 0x7c

// %d

base 2

0111 1100

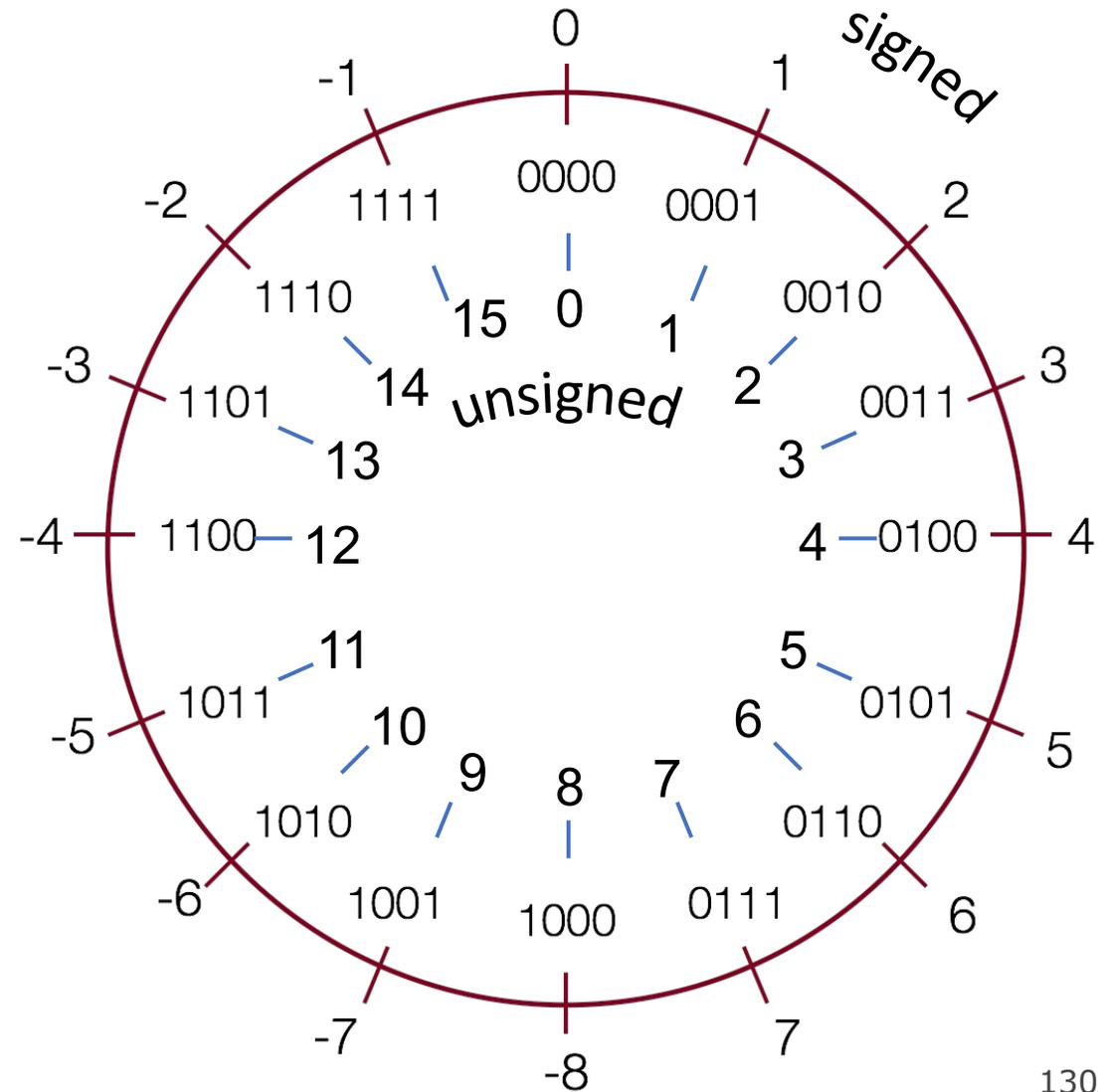
base 10



%u  
unsigned int

**25** iii. x = 25; // 0x19

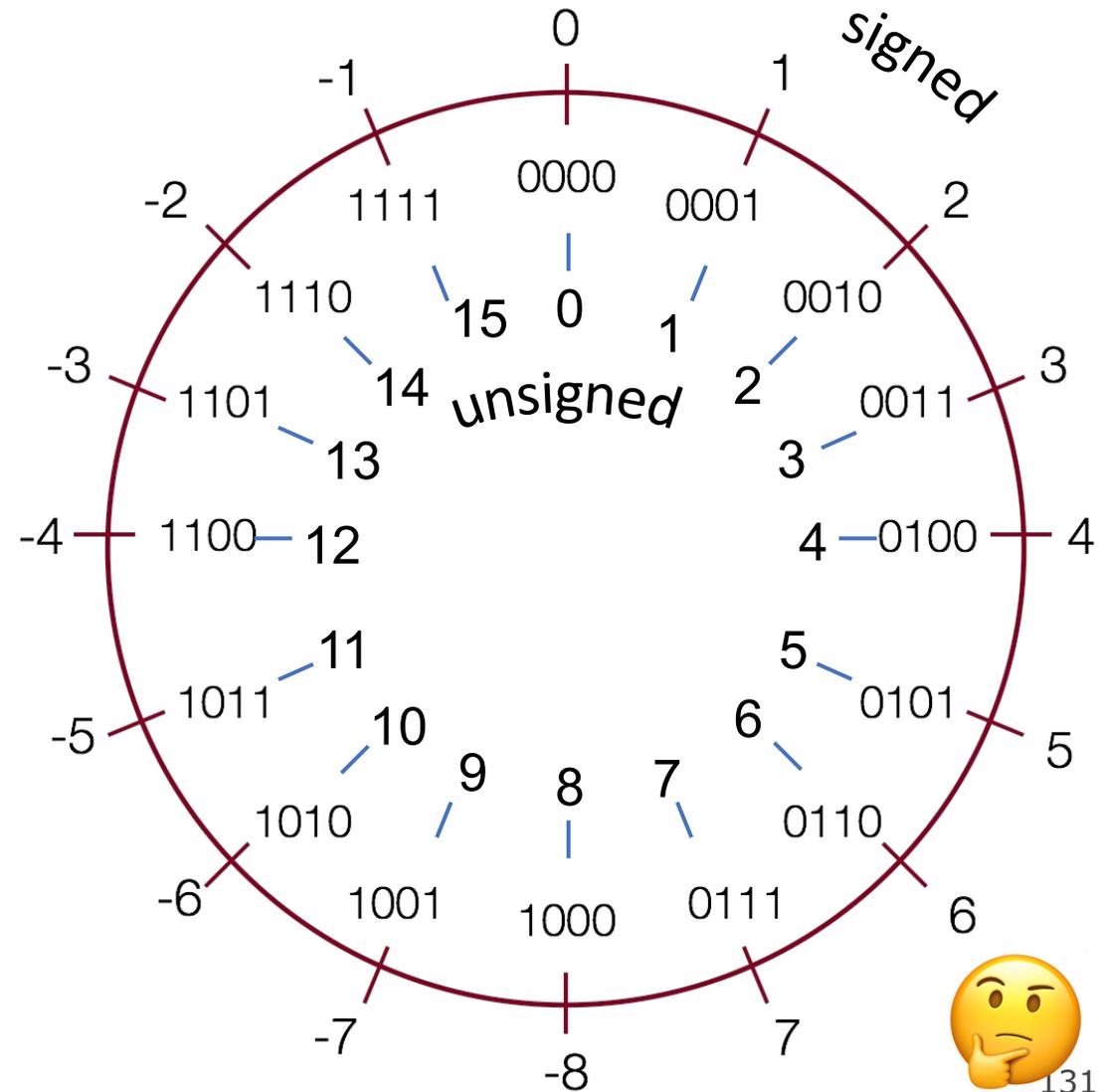
# Signed vs. Unsigned Integers



# 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?

$0b1101$

*unsigned int x = 42*

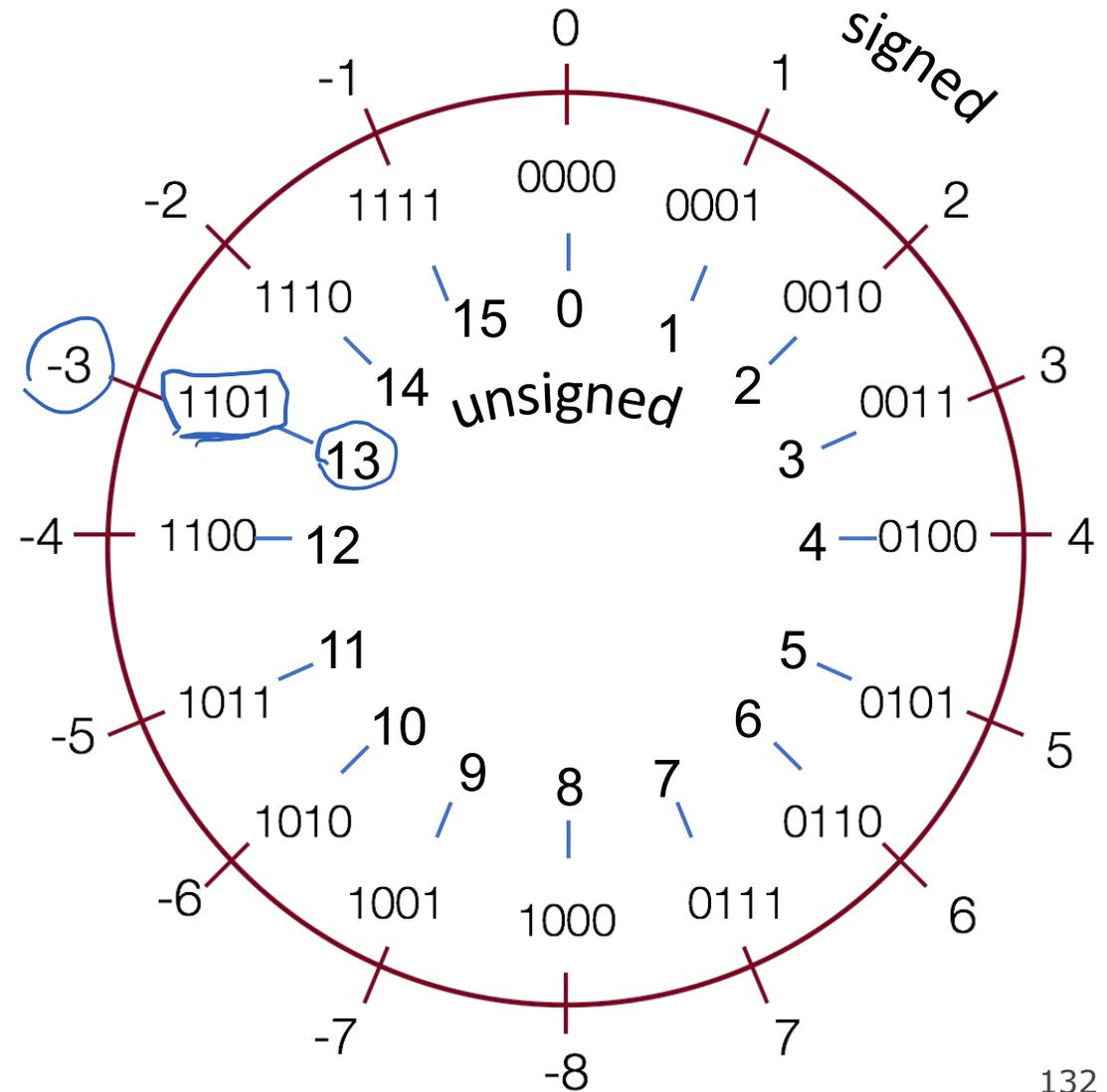
If 4-bit signed: -3

If 4-bit unsigned: 13

If >4-bit signed or unsigned: 13 ←

*0b001101*

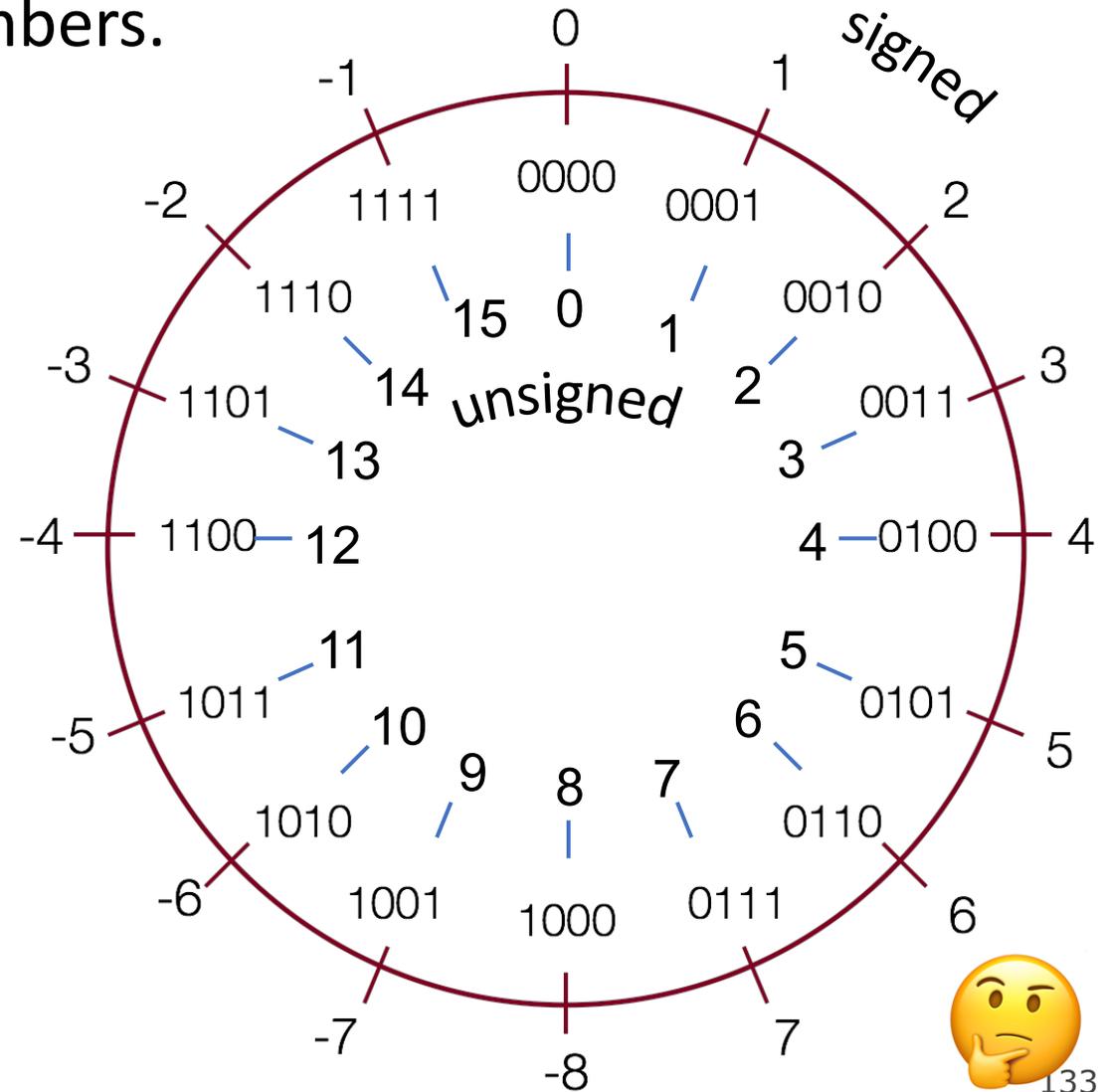
You need to know the type to determine the number! (Note by default, numeric constants in C are signed ints)



# Overflow

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

$$\begin{array}{r} \overset{1}{0}b1101 \\ + 0b0100 \\ \hline 0b\cancel{1}0001 \end{array}$$



# Overflow

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

$$\begin{array}{r} 0b1101 \\ + 0b0100 \\ \hline \end{array}$$

*Handwritten annotations: blue arrows pointing to the 1s in the 4th bit position of both numbers, and blue circles around the 0s in the 3rd, 2nd, and 1st bit positions of the second number.*

Signed

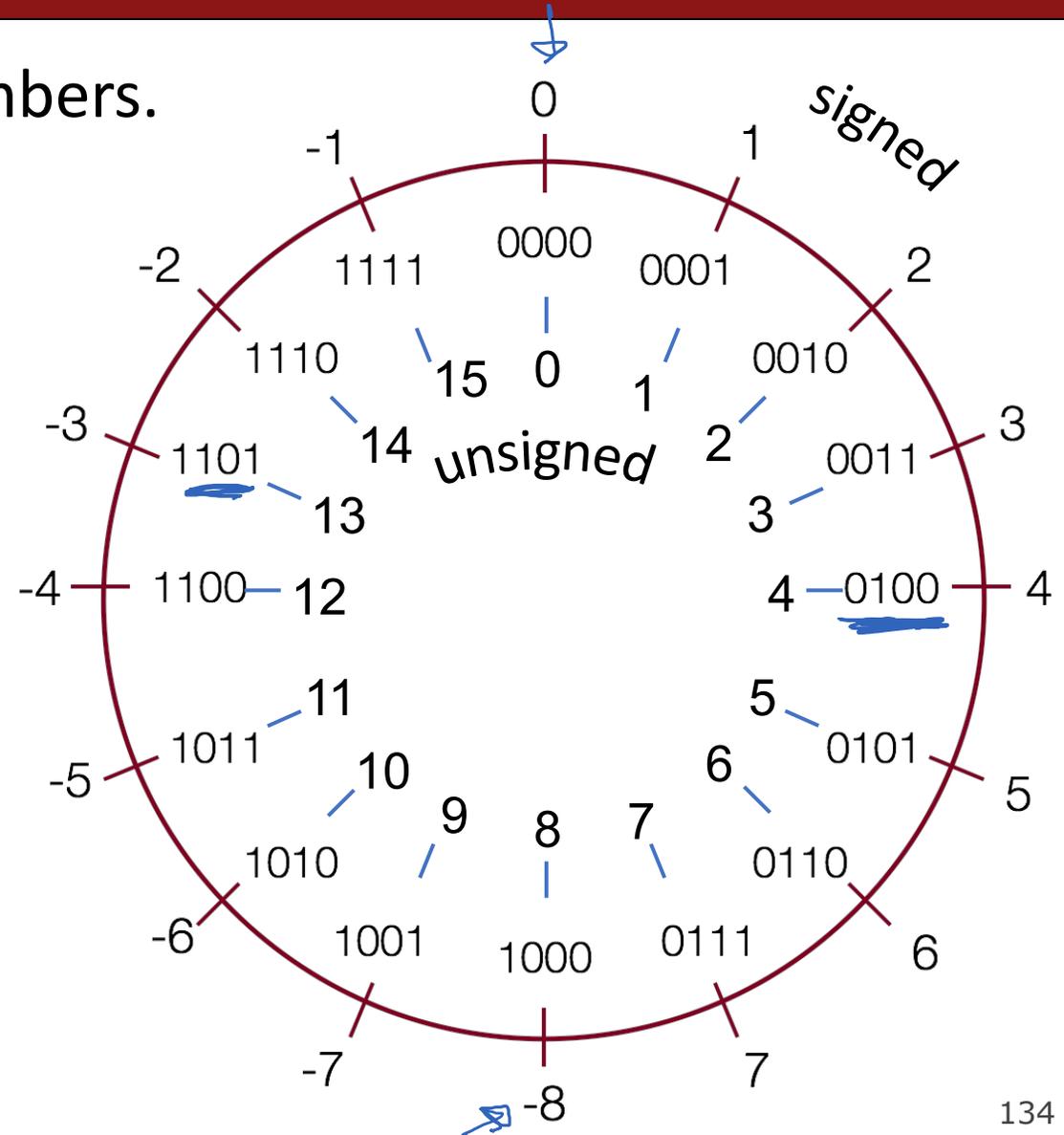
$$-3 + 4 = 1$$

No overflow

Unsigned

$$13 + 4 = 1$$

Overflow



# Limits and Comparisons

1. What is the...

	Largest unsigned?	Largest signed?	Smallest signed?
char			
int			

2. Will the following char comparisons evaluate to true or false?

i.  $-7 < 4$

iii.  $(\text{char})\ 130 > 4$

ii.  $-7 < 4u$

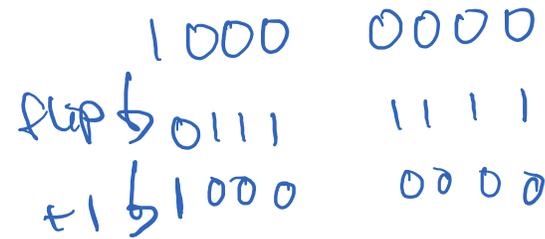
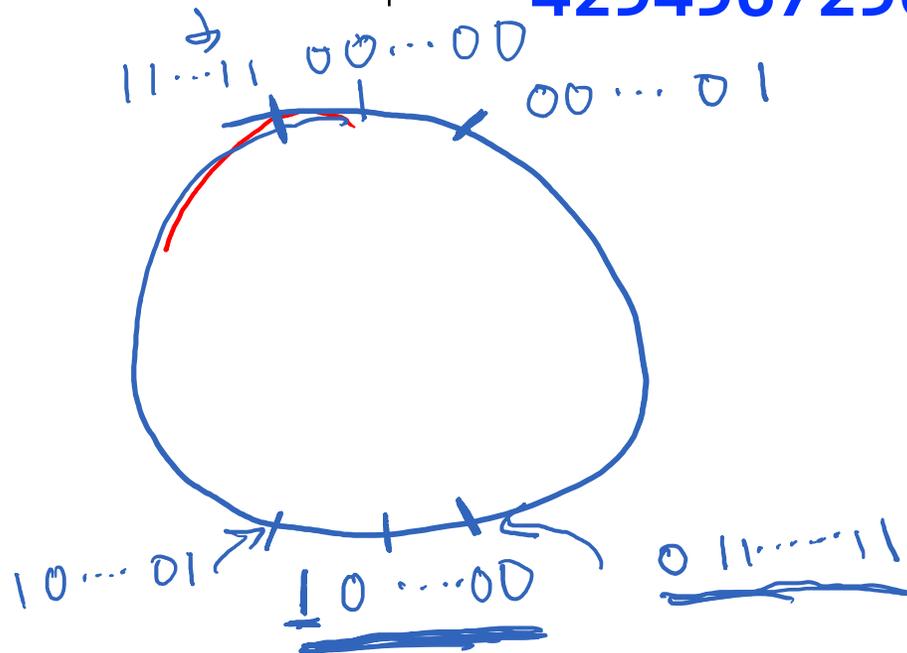
iv.  $(\text{char})\ -132 > 2$



# Limits and Comparisons

1. What is the...

	$0b \begin{array}{cc} 10000 & 0000 \\ 1111 & 1111 \end{array} \leftarrow$ Largest unsigned?	$0b \begin{array}{cc} 1000 & 0000 \\ 0111 & 1111 \end{array}$ Largest signed?	$1000 & 0000$ $\uparrow$ 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.

# Limits and Comparisons

2. Will the following char comparisons evaluate to true or false?

i.  $-7 < 4$       **true**

iii.  $(\text{char})\ 130 > 4$       **false**

ii.  $-7 < 4u$       **false**

iv.  $(\text{char})\ -132 > 2$       **true**

By default, numeric constants in C are signed ints, unless they are suffixed with u (unsigned) or L (long).

# Tools: A binary/hex calculator



Is there a program to quickly convert between hex, binary, and decimal?

- **Yes.** Next week, we will learn more about **gdb**, our debugger.
- gdb can print out variables/constants in any format: hex, decimal, unsigned...
- To look ahead, check out <https://cs107.stanford.edu/resources/gdb> and read about print format codes. Or watch Lecture 3!

$$\begin{array}{r}
 \downarrow \downarrow \\
 010 \\
 1101 \\
 - 0010 \\
 \hline
 1011
 \end{array}$$



$$\begin{array}{r}
 + \left[ \begin{array}{r} 1101 \\ - 0010 \\ \hline \end{array} \right. \\
 \left. \begin{array}{r} 1101 \\ + 1110 \\ \hline \end{array} \right] \\
 \hline
 11011
 \end{array}$$

$$\begin{array}{r}
 \left. \begin{array}{r} 1101 \\ 110 \\ \hline \end{array} \right\} \\
 1101 \\
 \leftarrow \\
 110
 \end{array}$$