

# COMP201

## Computer Systems & Programming

Lecture #03 –Representing and Operating on Integers



**KOÇ**  
**UNIVERSITY**

Aykut Erdem // Koç University // Fall 2020

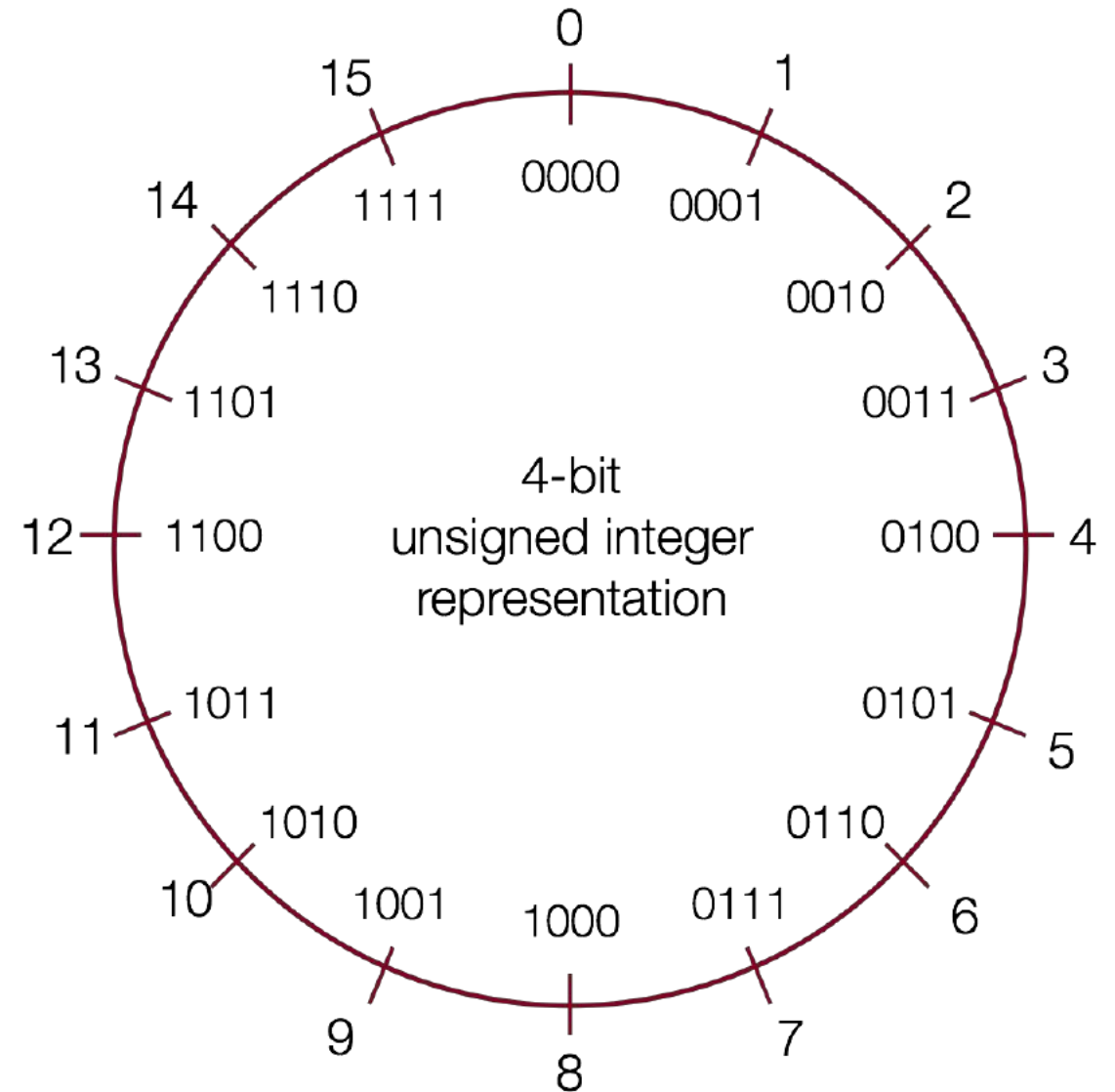
# Plan For Today

- Signed Integers
- Overflow
- Casting and Combining Types

**Disclaimer:** Slides for this lecture were borrowed from  
—Nick Troccoli's Stanford CS107 class

# Recap: Unsigned Integers

- An **unsigned** integer is 0 or a positive integer (no negatives).
- Converting between decimal and binary has a nice 1:1 relationship.
- 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).



# Lecture Plan

- Signed Integers
- Overflow
- Casting and Combining Types

# 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?

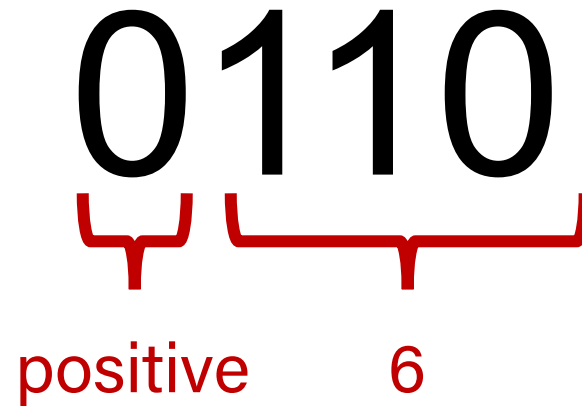
# Signed Integers

- 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

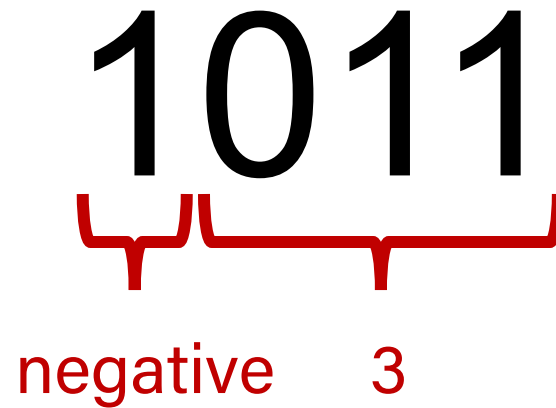
0 1 1 0



positive 6

The diagram shows the binary sequence 0110. A red bracket under the first bit (0) indicates the sign is positive. A red bracket under the remaining three bits (110) indicates the magnitude is 6.

1 0 1 1



negative 3

The diagram shows the binary sequence 1011. A red bracket under the first bit (1) indicates the sign is negative. A red bracket under the remaining three bits (011) indicates the magnitude is 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.

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

# 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.

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

# 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.

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

# 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.

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

# 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.

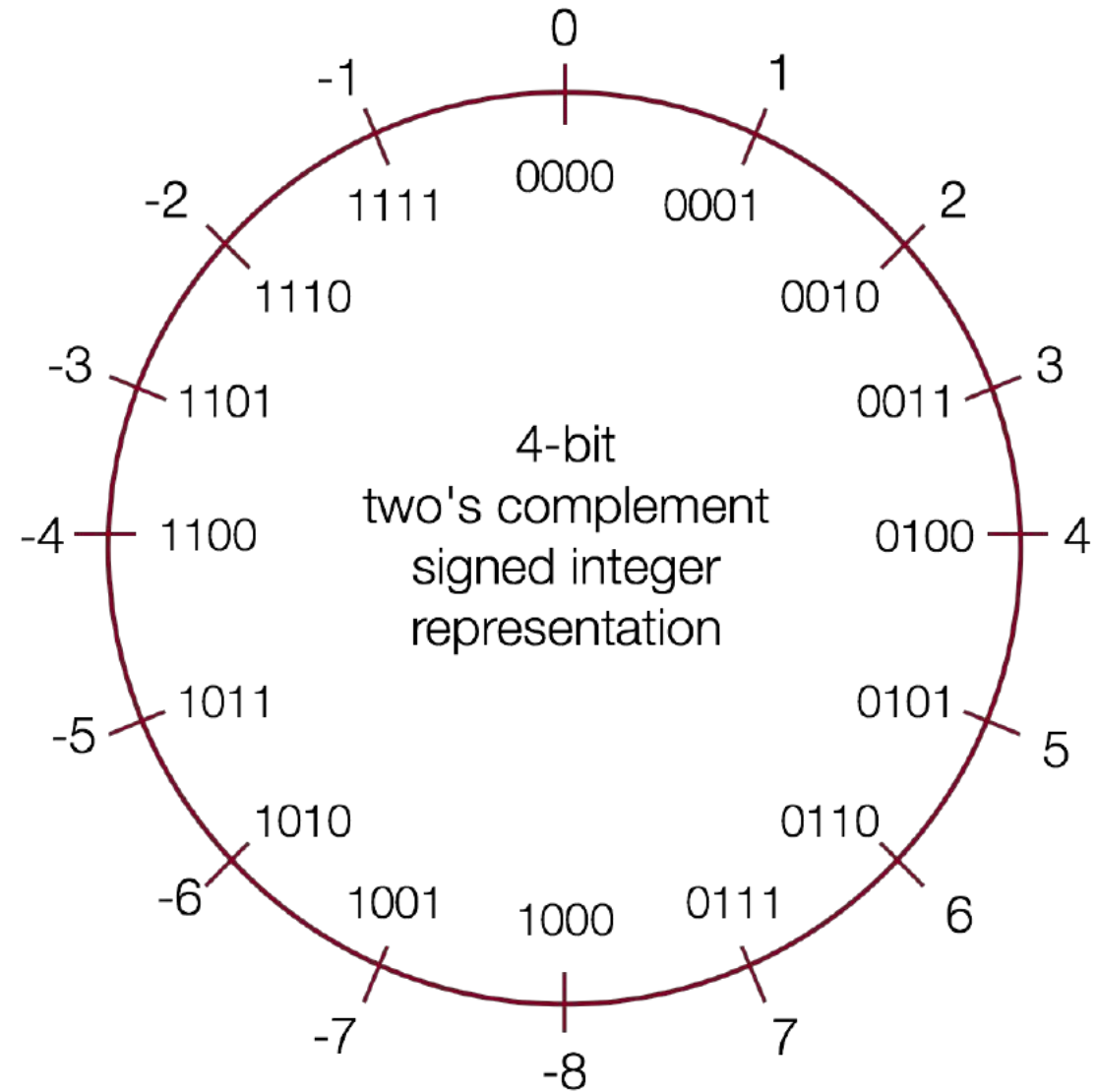
$$\begin{array}{r} 100100 \\ + \textcolor{red}{???100} \\ \hline 000000 \end{array}$$

# 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.

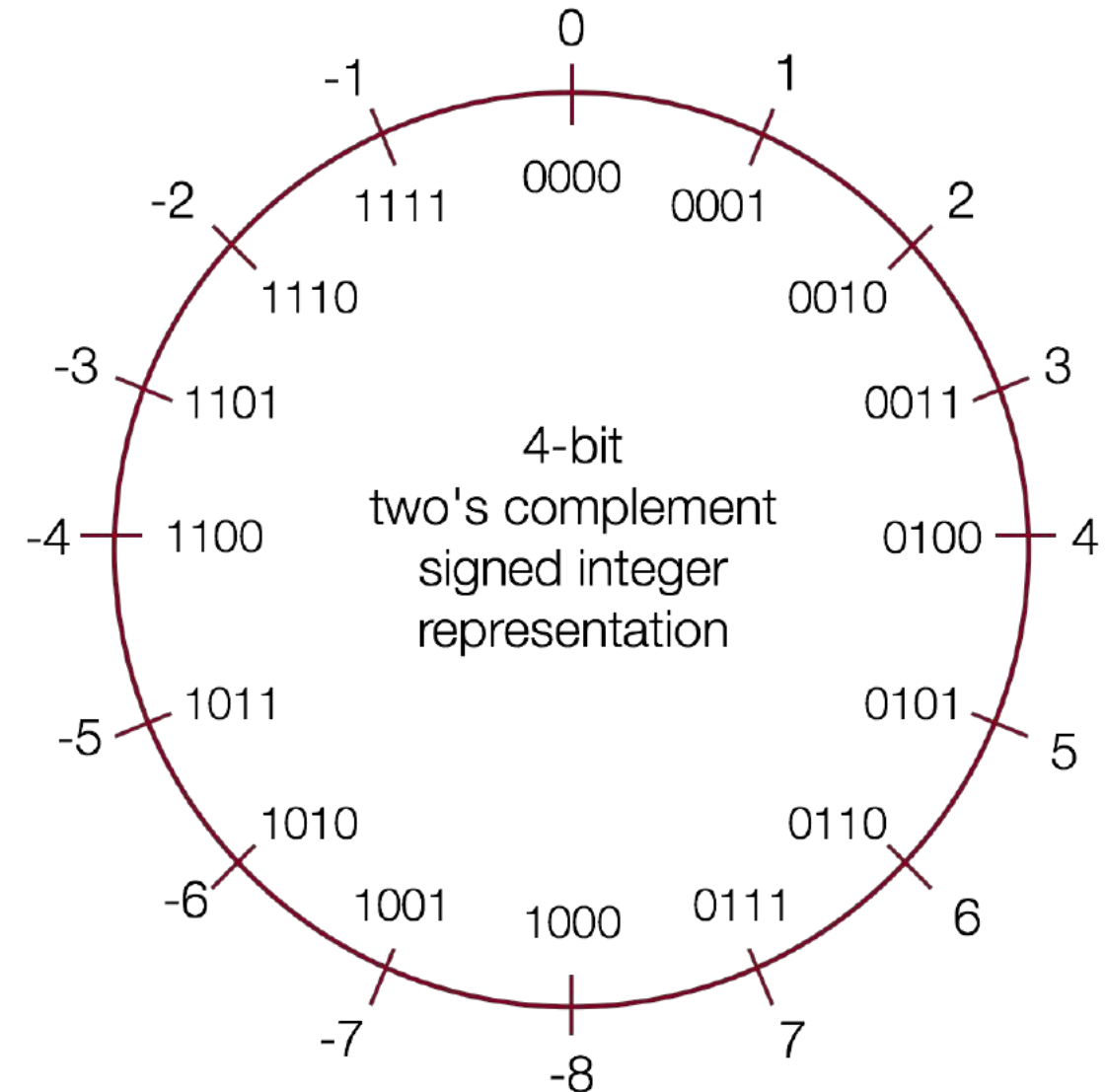
$$\begin{array}{r} 100100 \\ + 011100 \\ \hline 000000 \end{array}$$

# Two's Complement



# Two's Complement

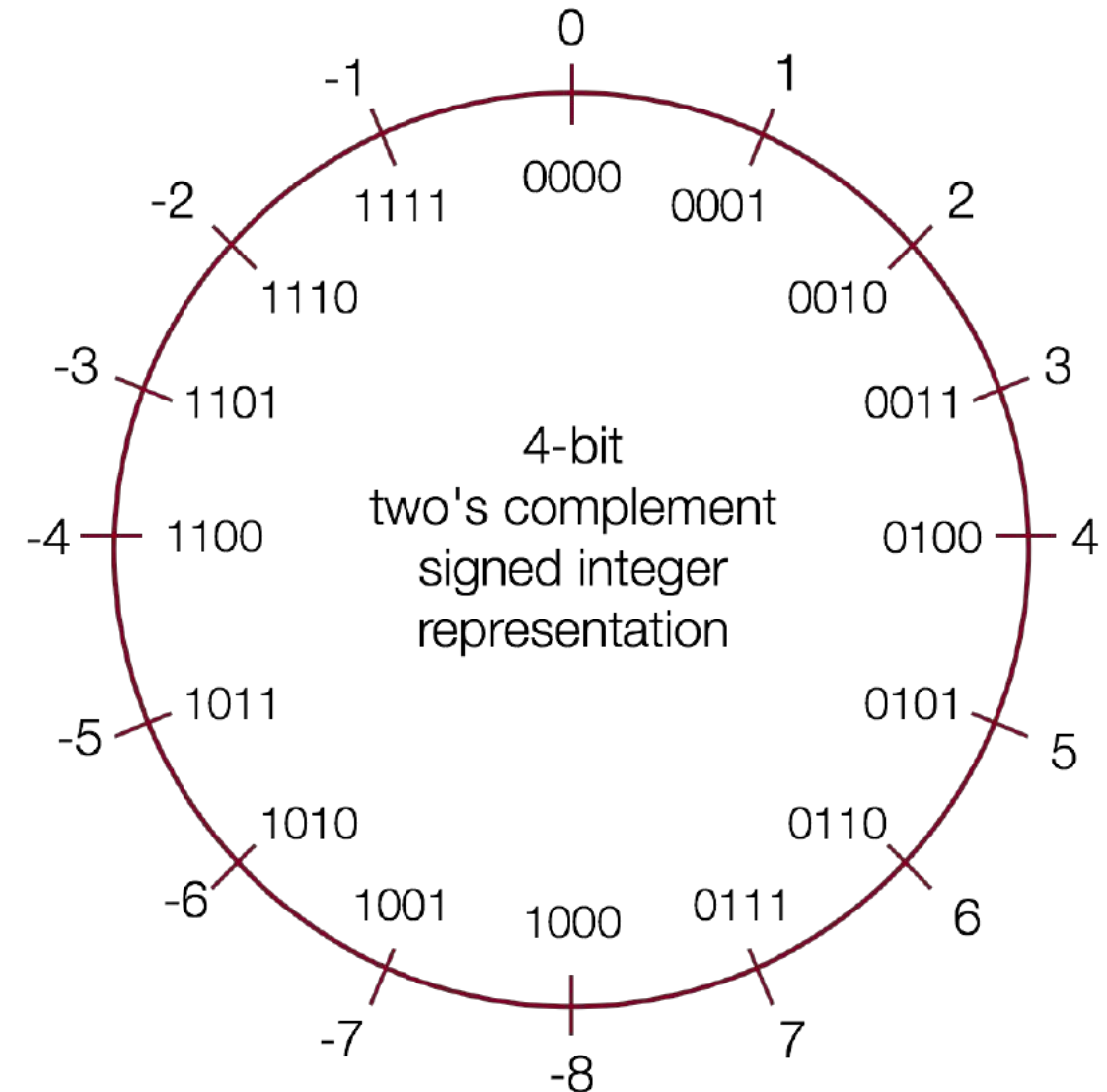
- In **two's complement**, we represent a positive number as **itself**, and its negative equivalent as the **two's complement of itself**.
- The **two's complement** of a number is the binary digits inverted, plus 1.
- This works to convert from positive to negative, **and** back from negative to positive!





# 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

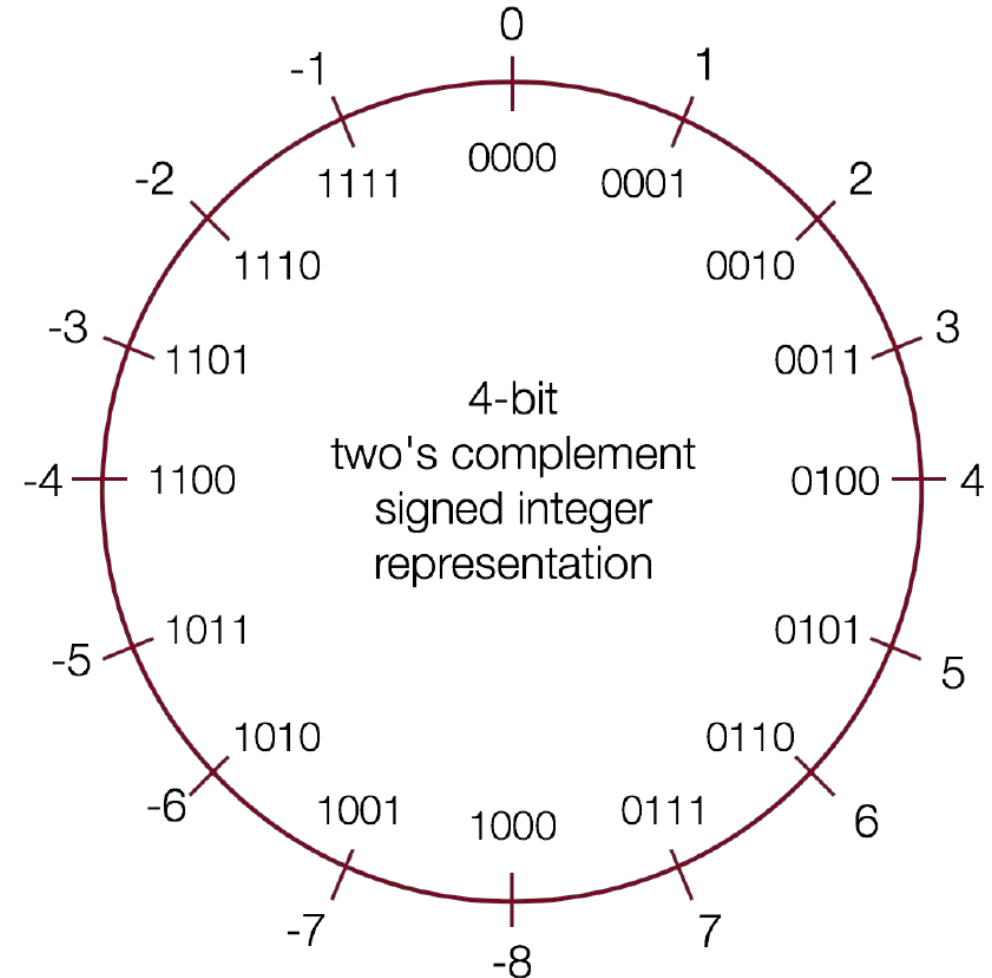
- Subtracting two numbers is just performing the two's complement on one of them and then adding. E.g.  $4 - 5 = -1$ .

0100	4		0100	4
-0101	5	→	+1011	-5
<hr/>			<hr/>	
			1111	-1

# 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)



# Lecture Plan

- Signed Integers
- **Overflow**
- Casting and Combining Types

# 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

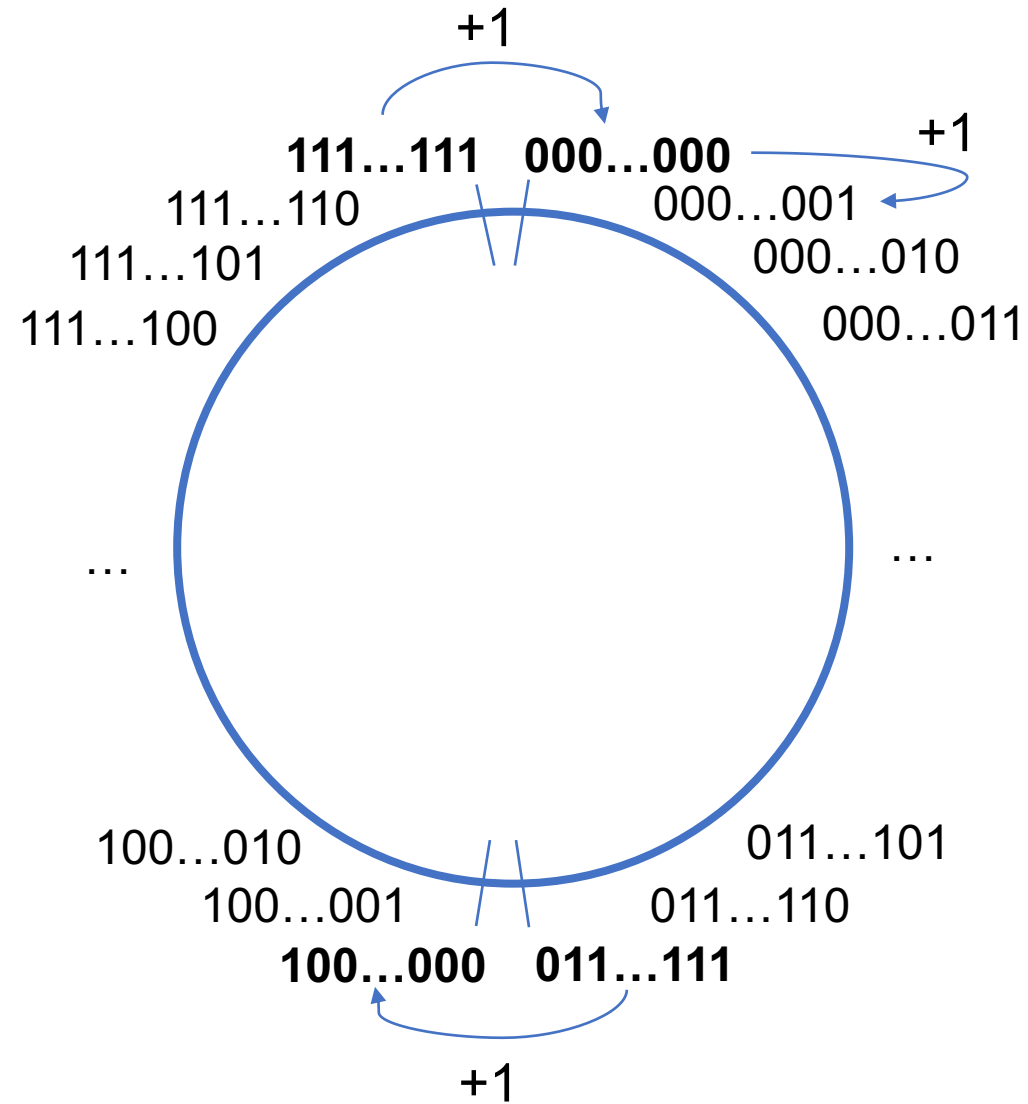
Type	Size (Bytes)	Minimum	Maximum
char	1	-128	127
unsigned char	1	0	255
short	2	-32768	32767
unsigned short	2	0	65535
int	4	-2147483648	2147483647
unsigned int	4	0	4294967295
long	8	-9223372036854775808	9223372036854775807
unsigned long	8	0	18446744073709551615

# Min and Max Integer Values

**INT\_MIN, INT\_MAX, UINT\_MAX, LONG\_MIN, LONG\_MAX,  
ULONG\_MAX, ...**



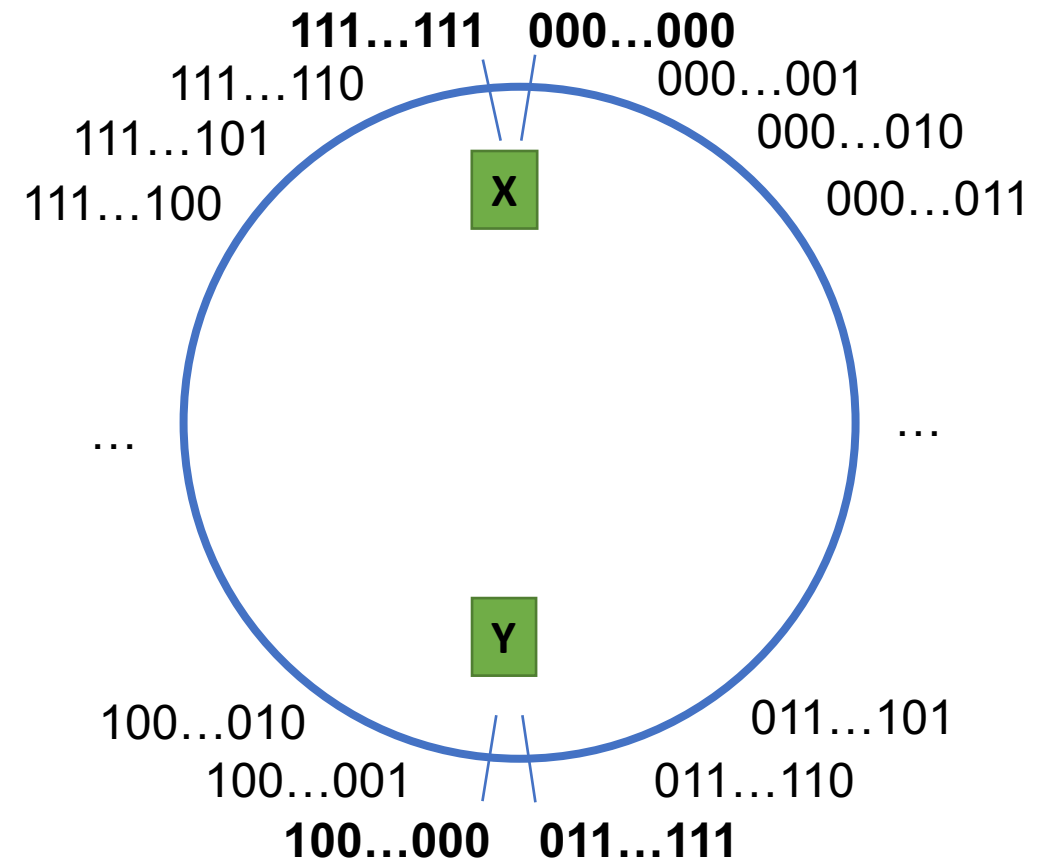
# Overflow



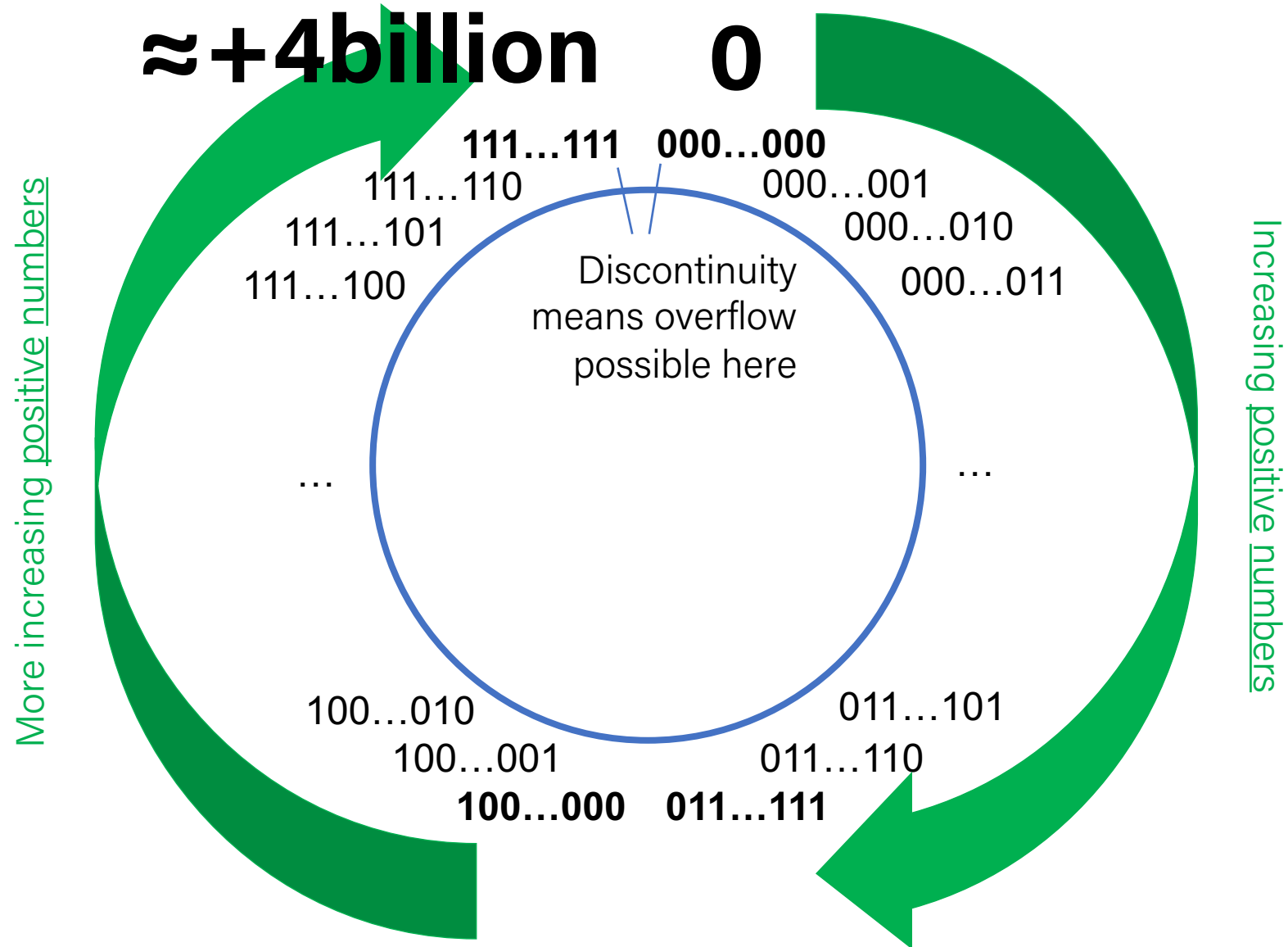
# Overflow

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

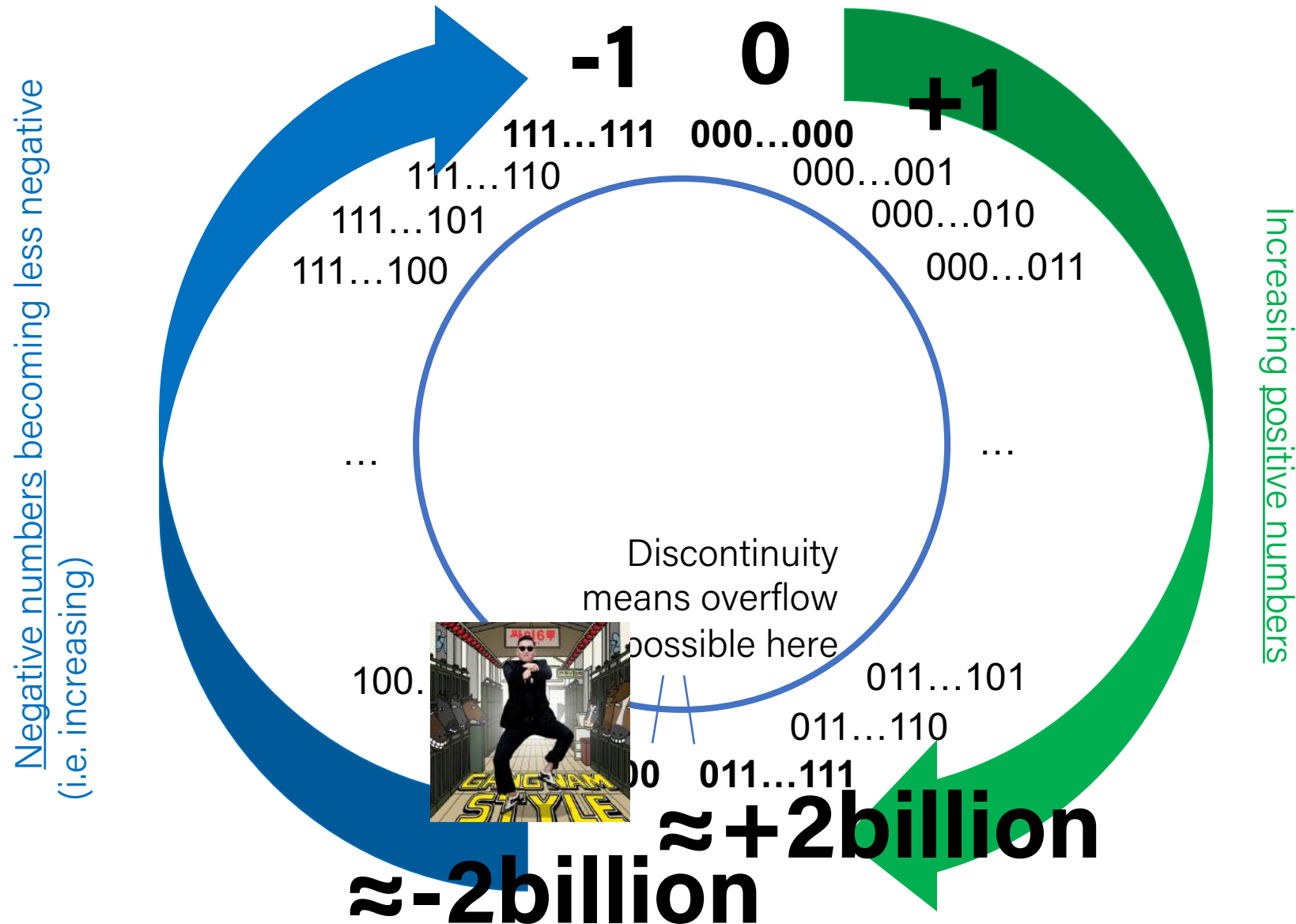
- A. Signed and unsigned can both overflow at points X and Y
- B. Signed can overflow only at X, unsigned only at Y
- C. Signed can overflow only at Y, unsigned only at X
- D. Signed can overflow at X and Y, unsigned only at X
- E. Other



# Unsigned Integers






# Signed Numbers




# Overflow In Practice: PSY



PSY - GANGNAM STYLE (강남스타일) M/V

 officialpsy 

 7,634,774

-2130754499

+ Add to  Share ... More

 8,871,284  1,154,582

Published on Jul 15, 2012

► Watch HANGOVER feat. Snoop Dogg M/V @ <http://youtu.be/HkMNOIYcpHg>

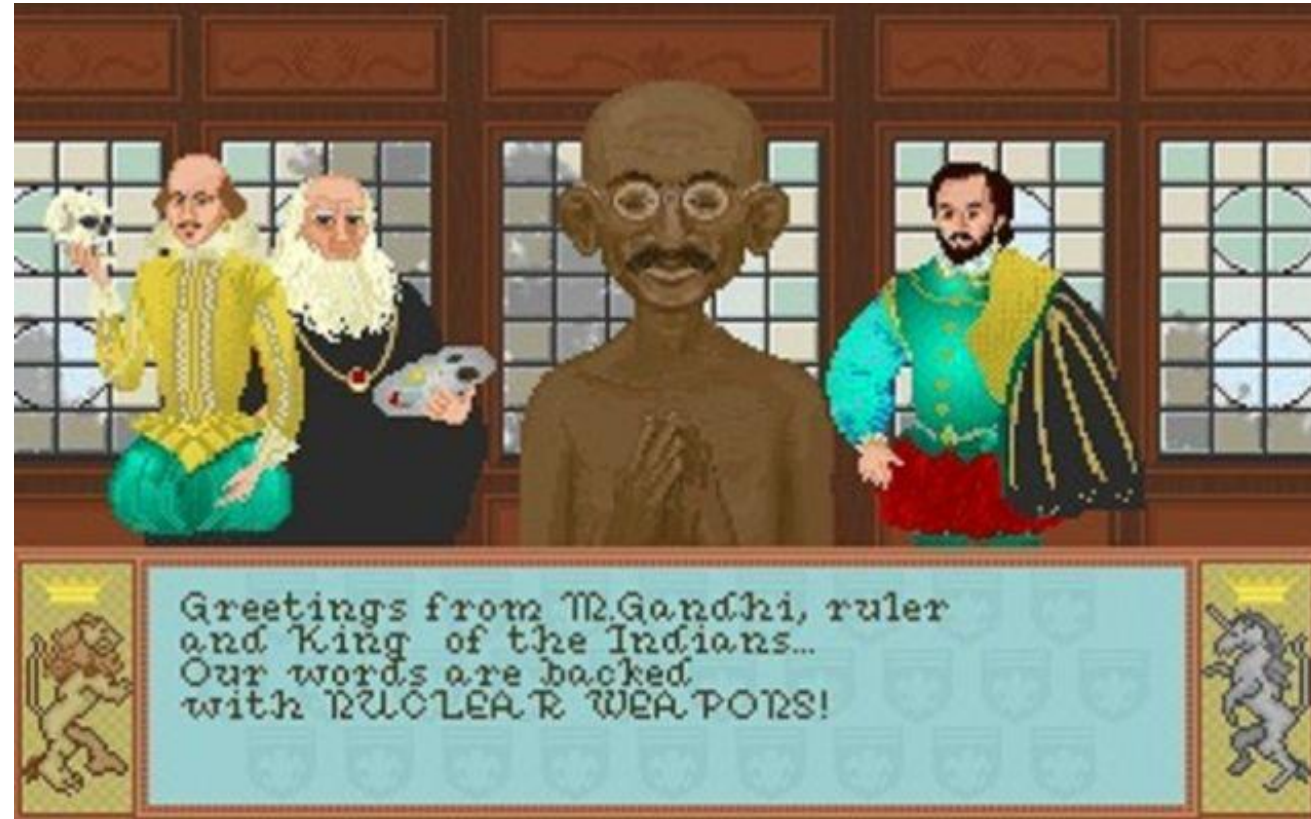
**YouTube:** "We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!"

# 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

- Signed Integers
- Overflow
- Casting and Combining Types

# 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.**

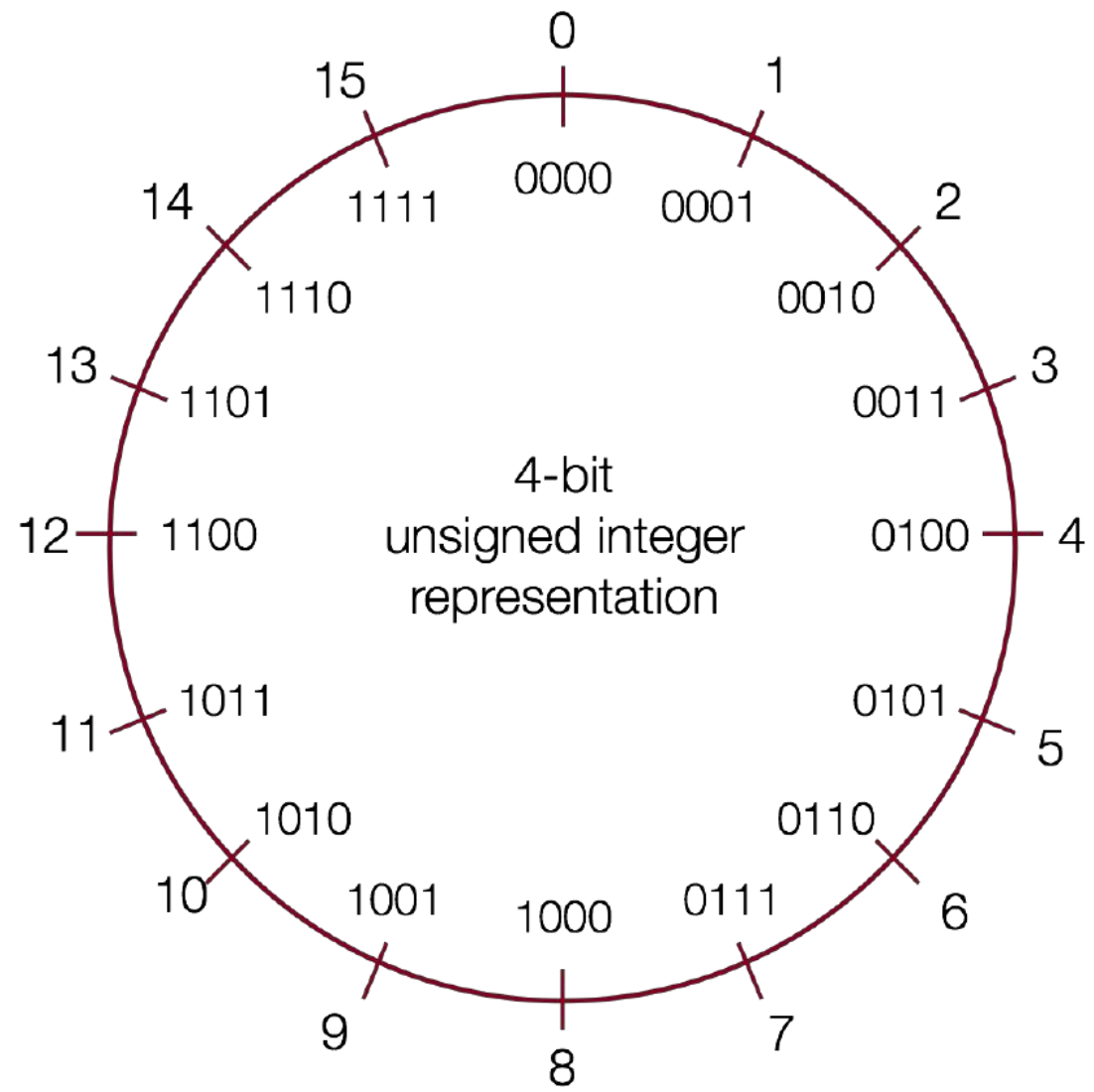
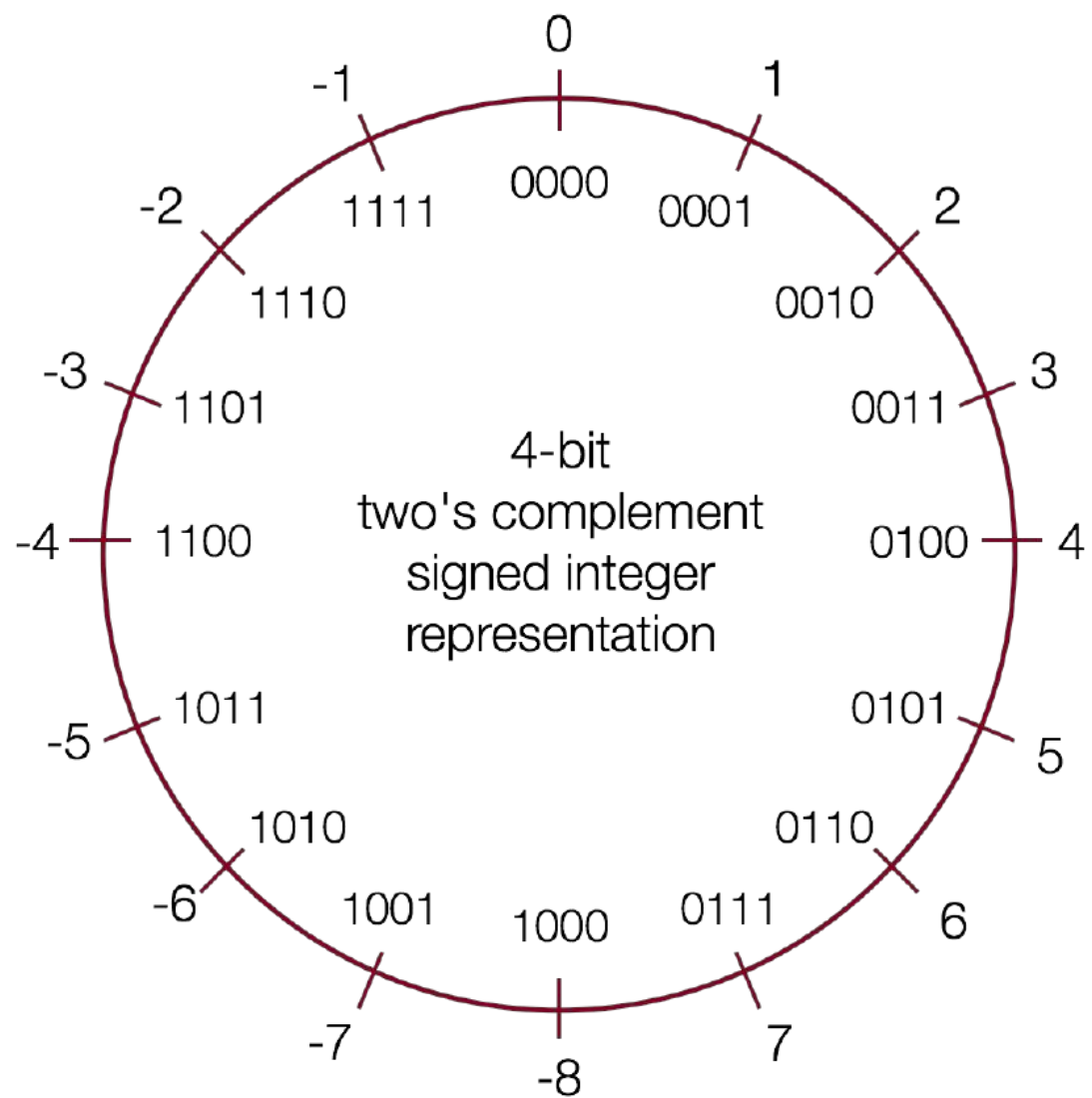
```
int v = -12345;  
unsigned int uv = v;  
printf("v = %d, uv = %u\n", v, uv);
```

The bit representation for -12345 is

0b**11111111111111111111111100111111000111**.

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

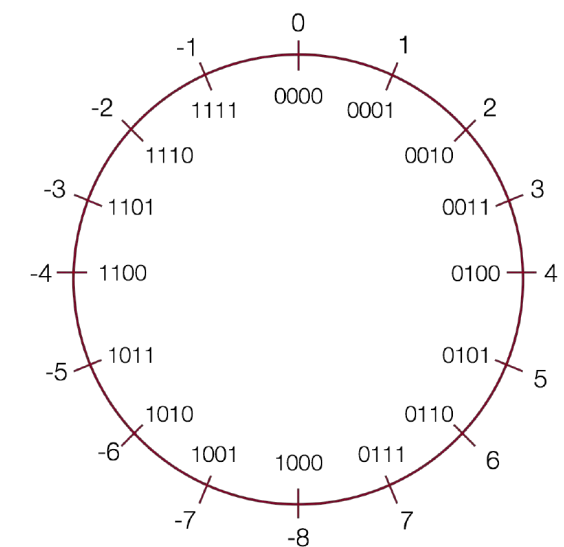
# Casting



# 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.

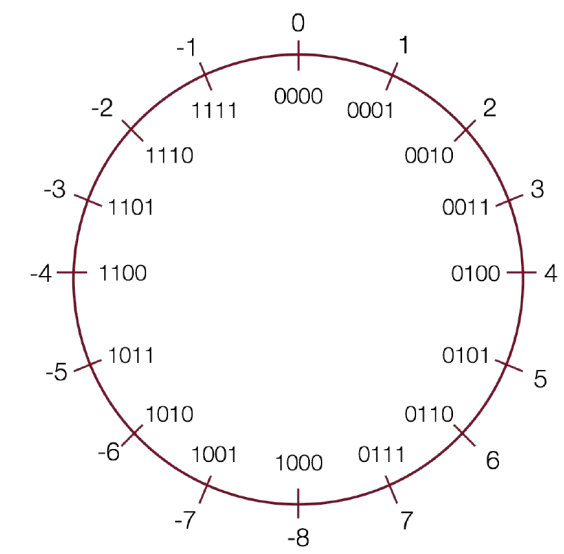
Expression	Type	Evaluation	Correct?
0 == 0U			
-1 < 0			
-1 < 0U			
2147483647 > -			
2147483647 - 1			
2147483647U > -			
2147483647 - 1			
2147483647 >			
(int)2147483648U			
-1 > -2			
(unsigned)-1 > -2			



# 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?
0 == 0U	Unsigned	1	yes
-1 < 0			
-1 < 0U			
2147483647 > -			
2147483647 - 1			
2147483647U > -			
2147483647 - 1			
2147483647 >			
(int)2147483648U			
-1 > -2			
(unsigned)-1 > -2			

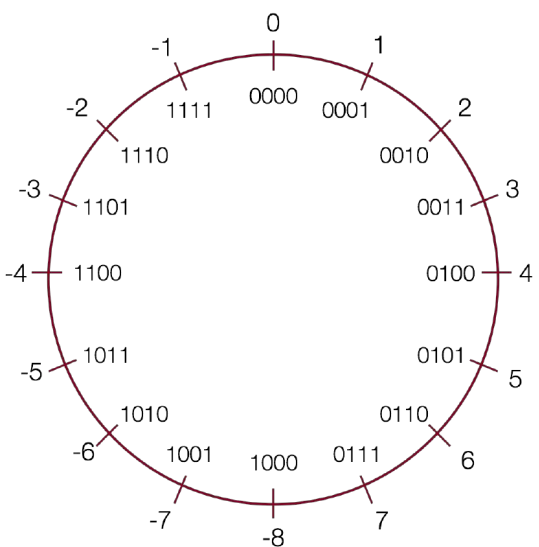




# Comparisons Between Different Types

- **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.

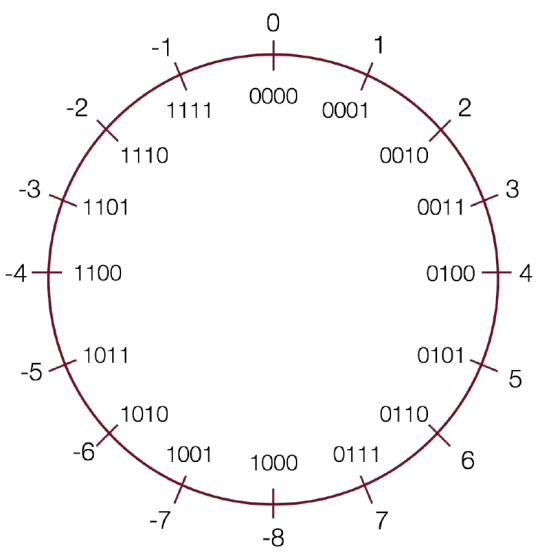
Expression	Type	Evaluation	Correct?
0 == 0U	Unsigned	1	yes
-1 < 0	Signed	1	yes
-1 < 0U			
2147483647 > -			
2147483647 - 1			
2147483647U > -			
2147483647 - 1			
2147483647 >			
(int)2147483648U			
-1 > -2			
(unsigned)-1 > -2			



# 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.

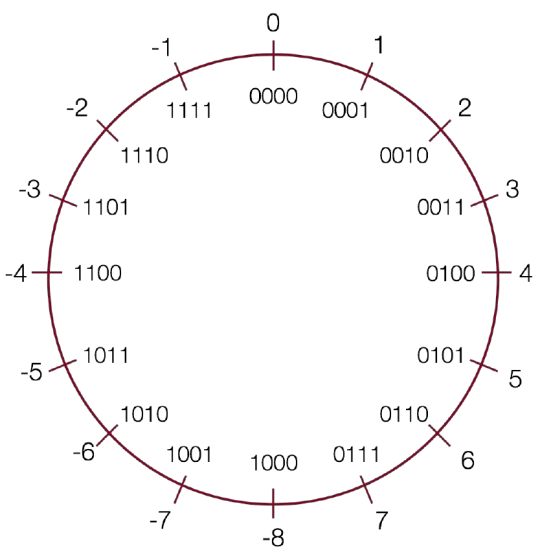
Expression	Type	Evaluation	Correct?
0 == 0U	Unsigned	1	yes
-1 < 0	Signed	1	yes
-1 < 0U	Unsigned	0	<b>No!</b>
2147483647 > -			
2147483647 - 1			
2147483647U > -			
2147483647 - 1			
2147483647 >			
(int)2147483648U			
-1 > -2			
(unsigned)-1 > -2			



# 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.

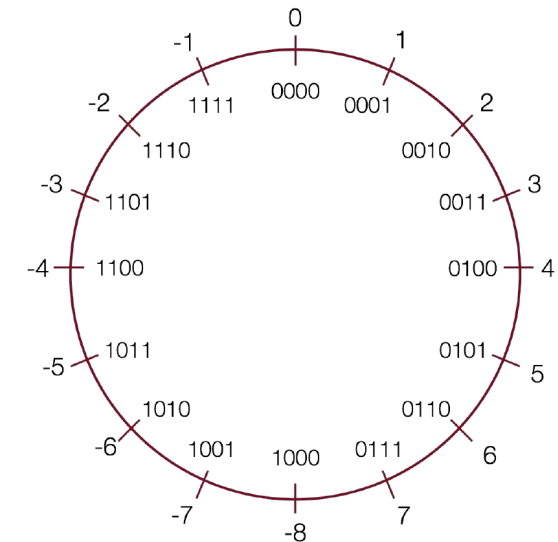
Expression	Type	Evaluation	Correct?
0 == 0U	Unsigned	1	yes
-1 < 0	Signed	1	yes
-1 < 0U	Unsigned	0	No!
2147483647 > -	Signed	1	yes
2147483647 - 1			
2147483647U > -	Unsigned	0	No!
2147483647 - 1			
2147483647 >	Signed	1	yes
(int)2147483648U			
-1 > -2	Signed	1	yes
(unsigned)-1 > -2	Unsigned	0	No!



# 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.

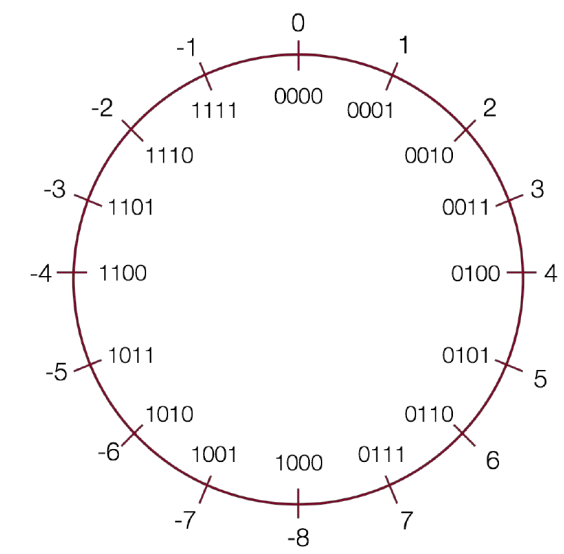
Expression	Type	Evaluation	Correct?
0 == 0U	Unsigned	1	yes
-1 < 0	Signed	1	yes
-1 < 0U	Unsigned	0	No!
2147483647 > -	Signed	1	yes
2147483647 - 1			
2147483647U >	Unsigned	0	No!
-2147483647 - 1			
2147483647 >	(int)2147483648U		
(int)2147483648U			
-1 > -2			
(unsigned)-1 > -2			



# 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.

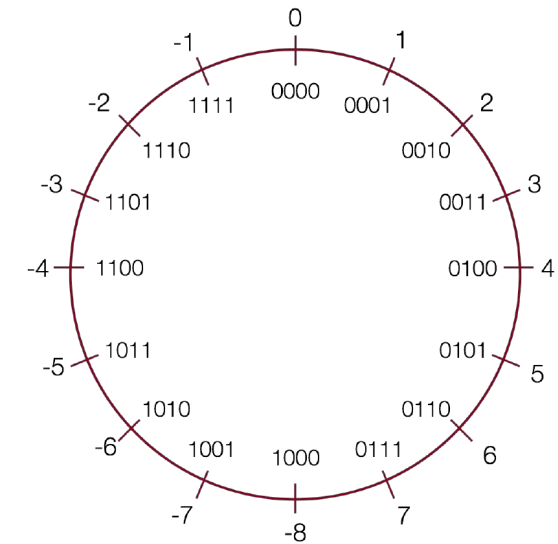
Expression	Type	Evaluation	Correct?
0 == 0U	Unsigned	1	yes
-1 < 0	Signed	1	yes
-1 < 0U	Unsigned	0	No!
2147483647 > -	Signed	1	yes
2147483647 - 1			
2147483647U > -	Unsigned	0	No!
2147483647 - 1			
2147483647 >	Signed	1	No!
(int)2147483648U			
-1 > -2			
(unsigned)-1 > -2			



# 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.

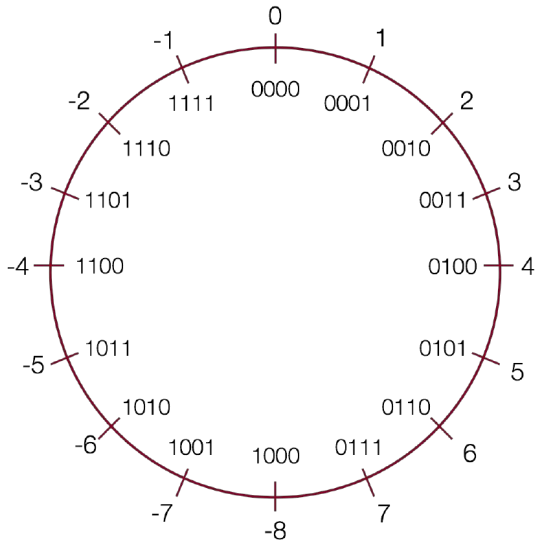
Expression	Type	Evaluation	Correct?
0 == 0U	Unsigned	1	yes
-1 < 0	Signed	1	yes
-1 < 0U	Unsigned	0	No!
2147483647 > -	Signed	1	yes
2147483647 - 1			
2147483647U > -	Unsigned	0	No!
2147483647 - 1			
2147483647 >	Signed	1	No!
(int)2147483648U			
-1 > -2	Signed	1	yes
(unsigned)-1 > -2			



# 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>	Unsigned	0	No!
<code>2147483647 &gt;</code> <code>(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**

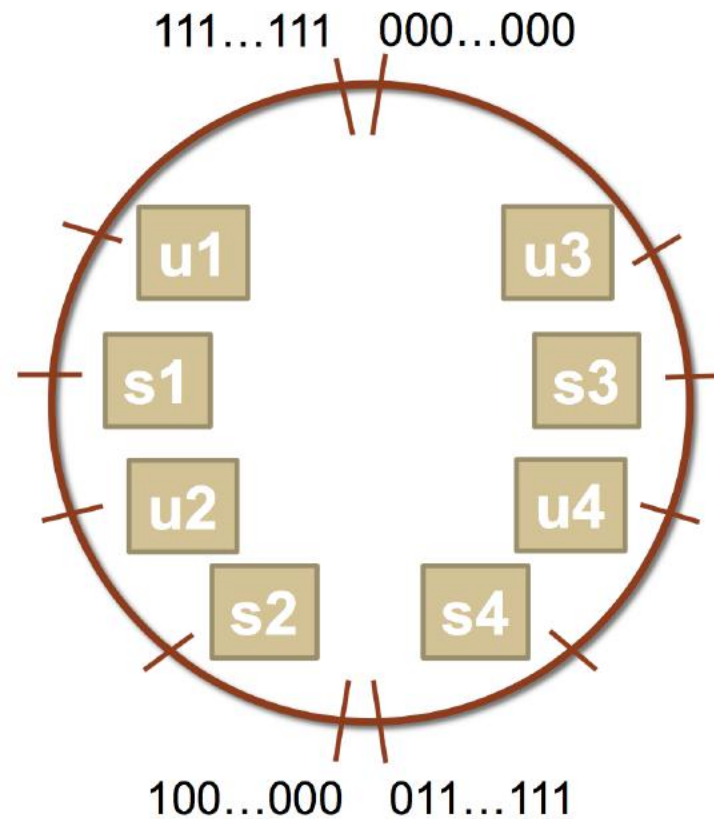
**u2 > u4**

**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**

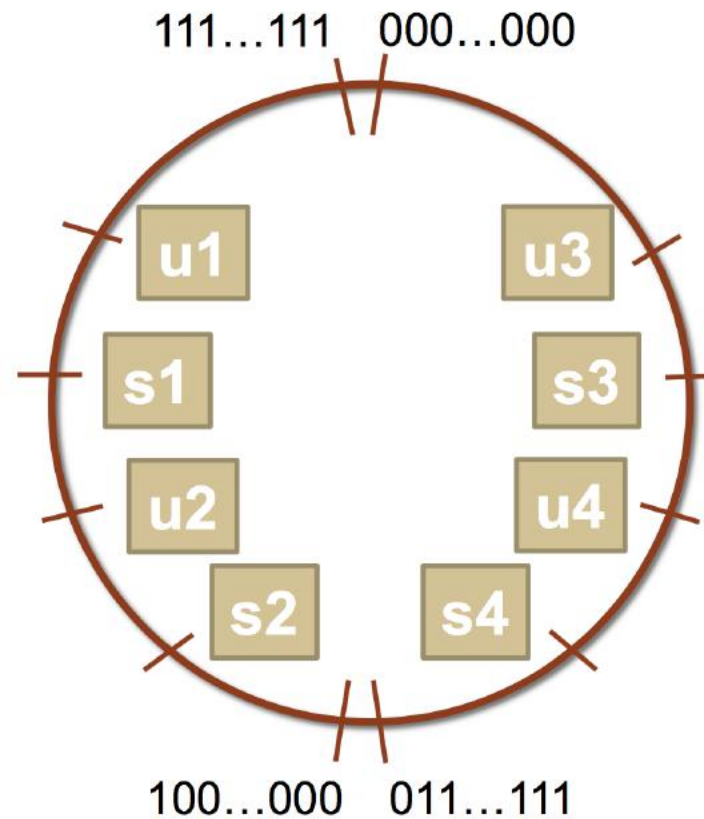
u2 > u4

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**

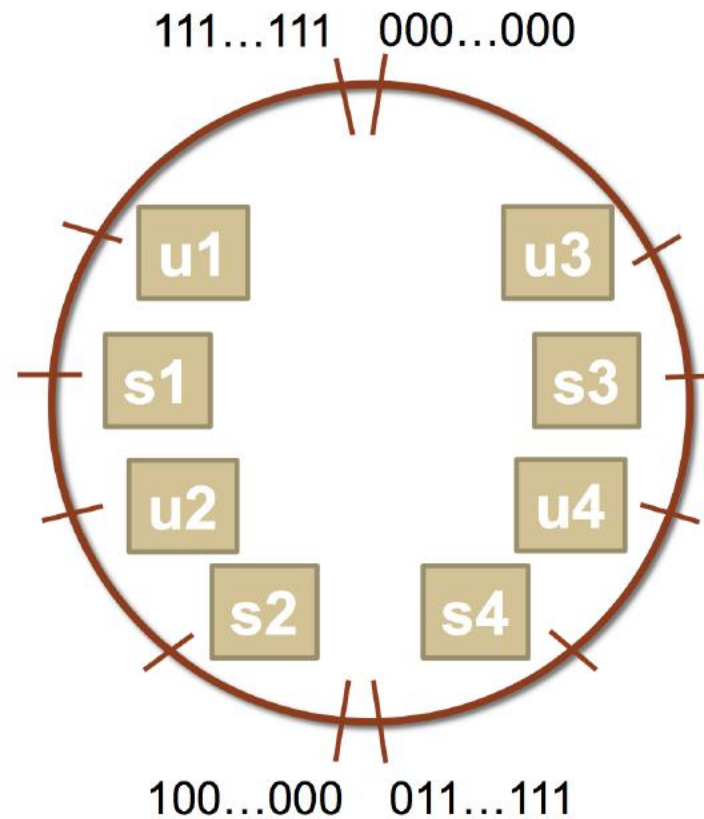
**u2 > u4 - true**

**s2 > s4**

**s1 > s2**

**u1 > u2**

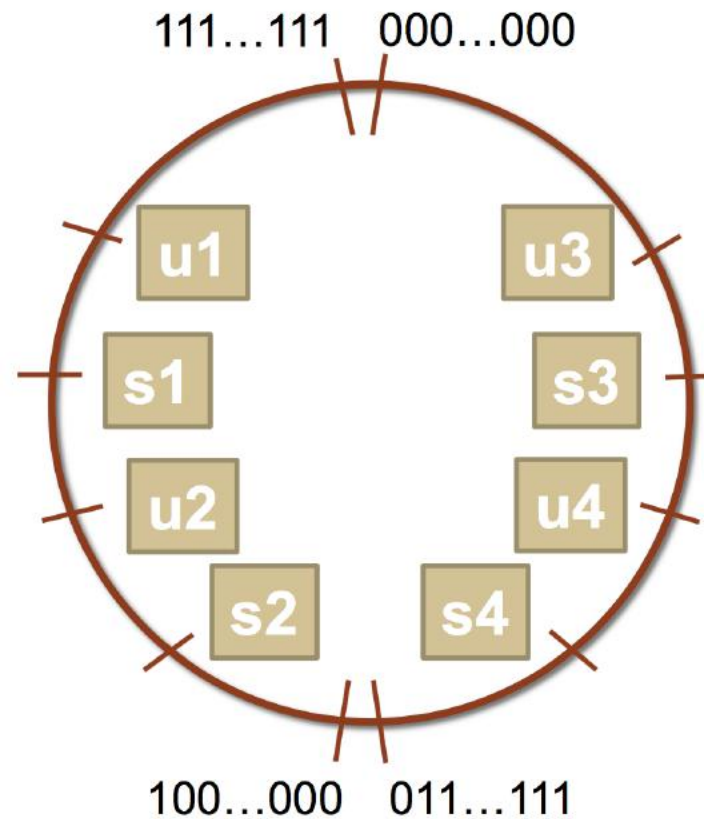
**s1 > u3**



# 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*)

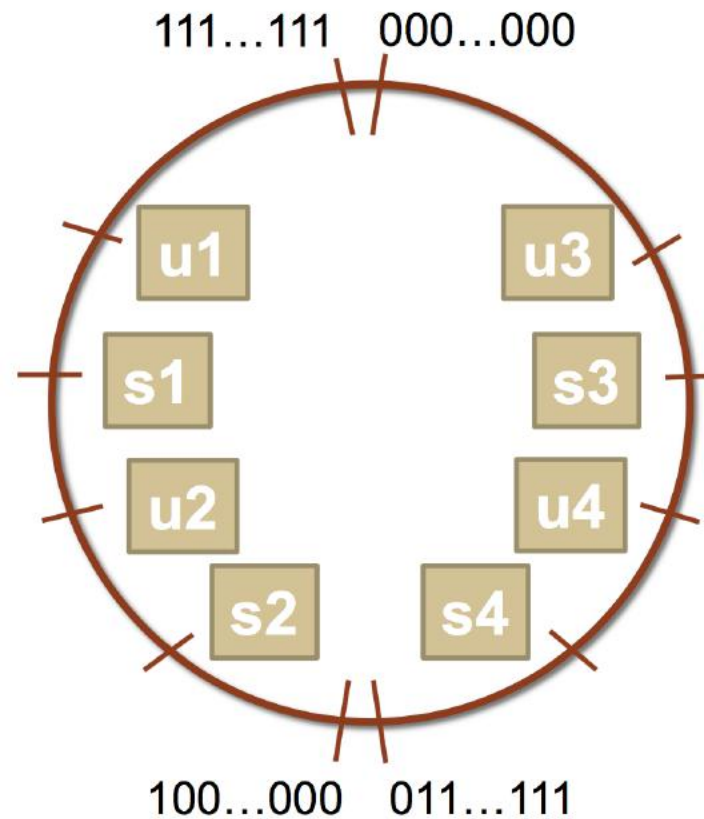
**s3 > u3 - true**  
u2 > u4 - true  
s2 > s4 - false  
s1 > s2  
u1 > u2  
s1 > u3



# 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*)

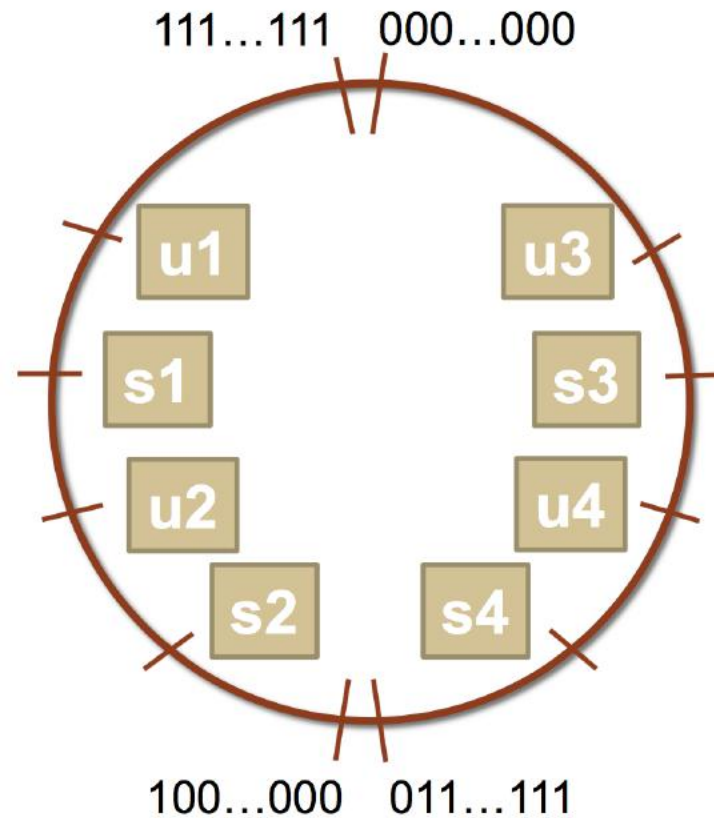
**s3 > u3 - true**  
**u2 > u4 - true**  
**s2 > s4 - false**  
**s1 > s2 - true**  
**u1 > u2**  
**s1 > u3**



# 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*)

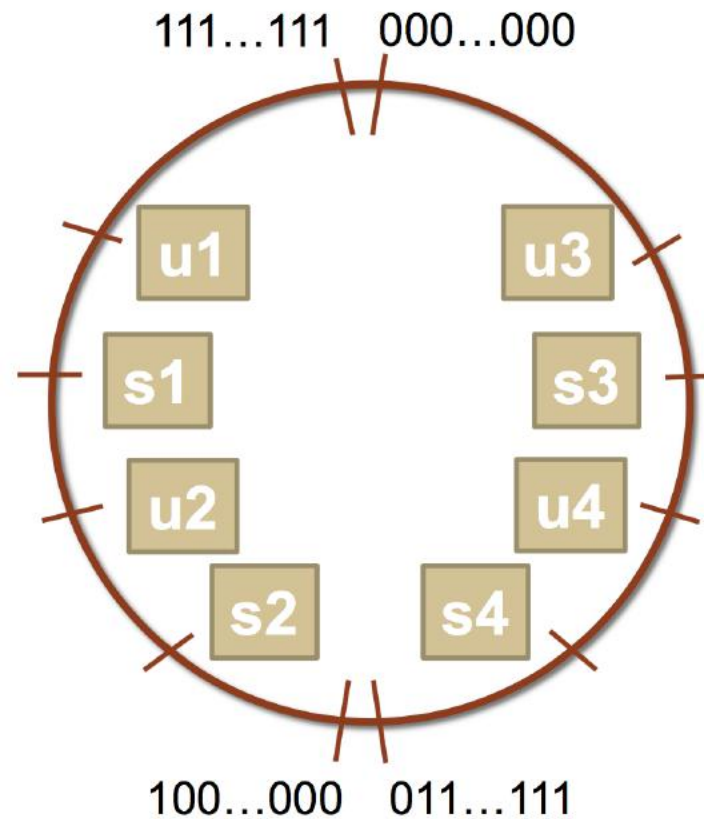
**s3 > u3 - true**  
**u2 > u4 - true**  
**s2 > s4 - false**  
**s1 > s2 - true**  
**u1 > u2 - true**  
**s1 > u3**



# 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  
0100b
```

```
s = 0000 0000 0000
```

```
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

- Signed Integers
- Overflow
- Casting and Combining Types

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