CS107 Spring 2019, Lecture 2
Bits and Bytes; Integer Representations

reading:

Bryant & O'Hallaron, Ch. 2.2-2.3

<u>CS107 Topic 1</u>: How can a computer represent integer numbers?

Demo: Unexpected Behavior

```
cp -r /afs/ir/class/cs107/samples/lectures/lect2 .
```

Plan For Today

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- **Break:** Announcements
- Signed Integers
- Casting and Combining Types

Plan For Today

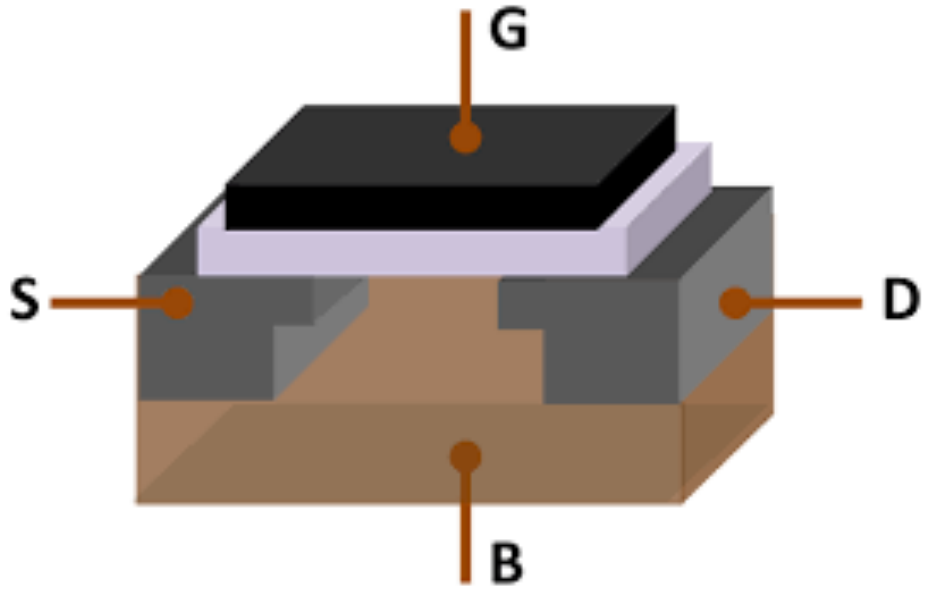
- **Bits and Bytes**
- Hexadecimal
- Integer Representations
- Unsigned Integers
- **Break:** Announcements
- Signed Integers
- Casting and Combining Types

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

5	9	3	4
thousands	hundreds	tens	ones

Base 10

5 9 3 4

thousands hundreds tens ones

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

Base 10

5 9 3 4

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

Base 10

5 9 3 4

10^X: 3 2 1 0

Base 2

1 0 1 1

2^X: 3   2   1   0

Digits 0-1 (*0* to *base-1*)

Base 2

1 0 1 1

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

Base 2

1 0 1 1

eights fours twos ones

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

Base 2

Most significant bit (MSB)

Least significant bit (LSB)

1 0 1 1

eights fours twos ones

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

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 ≤ 6?

Base 10 to Base 2

Question: What is 6 in base 2?

- Strategy:
  - What is the largest power of 2 ≤ 6? **2^2=4**

0 1 _ _

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

Base 10 to Base 2

Question: What is 6 in base 2?

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

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

Base 10 to Base 2

Question: What is 6 in base 2?

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

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

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

Base 10 to Base 2

Question: What is 6 in base 2?

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

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

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!

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

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

Practice: Base 2 to Base 10

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

a) **20** (text code: 641180)
b) **101** (text code: 642224)
c) **10** (text code: 642225)
d) **5** (text code: 642226)
e) **Other** (text code: 642227)

Respond at pollev.com/nicktroccoli901 or text a code above to 22333.

To show this poll

1

Install the app from
pollev.com/app

2

Start the presentation

Still not working? Get help at pollev.com/app/help
or
Open poll in your web browser

Practice: Base 10 to Base 2

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

a) **1111** (text code: 642232)

b) **1110** (text code: 642233)

c) **1010** (text code: 642235)

d) **Other** (text code: 642236)

Respond at pollev.com/nicktroccoli901 or text a code above to 22333.

To show this poll

1

Install the app from
pollev.com/app

2

Start the presentation

Still not working? Get help at pollev.com/app/help
or
Open poll in your web browser

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 = ?**

11111111

2^x: 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 = ?**

$$\begin{array}{lcccccccc} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2^x: & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{array}$$

- **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**

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

$1453 \times 10 = 1453\underline{\mathbf{0}}$

$1101_2 \times 2 = 1101\underline{\mathbf{0}}$

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

Plan For Today

- Bits and Bytes
- **Hexadecimal**
- Integer Representations
- Unsigned Integers
- **Break:** Announcements
- Signed Integers
- Casting and Combining Types

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**.

0110 1010 0011

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**.

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

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`**

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

Practice: Hexadecimal to Binary

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

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 **`0b1111001010110110110011`** in hexadecimal? (*Hint: start from the right*)

Practice: Hexadecimal to Binary

What is **`0b1111001010110110110011`** in hexadecimal? (*Hint: start from the right)*

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

Plan For Today

- Bits and Bytes
- Hexadecimal
- **Integer Representations**
- Unsigned Integers
- **Break:** Announcements
- Signed Integers
- Casting and Combining Types

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})

Stay tuned until week 5!

32-Bit and 64-Bit

- In the early 2000's, most computers were **32-bit**. This means that pointers in programs were **32 bits.**
- 32-bit pointers could store a memory address from 0 to 2^{32}-1, for a total of **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 pointers in programs were **64 bits.**
- 64-bit pointers could store a memory address from 0 to 2^{64}-1, for a total of **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)!

Number Representations

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

Number Representations

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

Myth

Plan For Today

- Bits and Bytes
- Hexadecimal
- Integer Representations
- **Unsigned Integers**
- **Break:** Announcements
- Signed Integers
- Casting and Combining Types

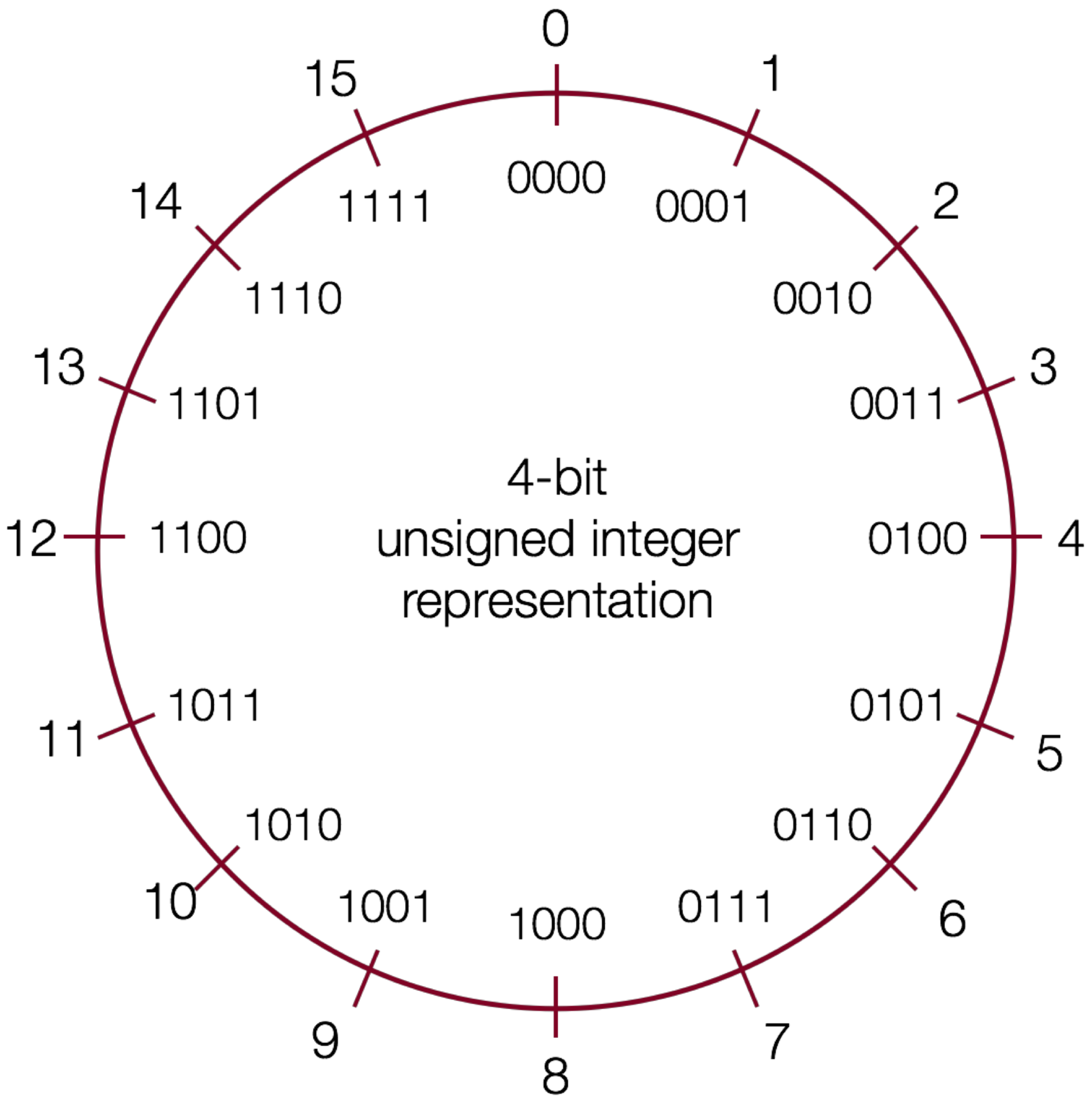
Unsigned Integers

- 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



Plan For Today

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- **Break:** Announcements
- Signed Integers
- Casting and Combining Types

Announcements

- Sign up for Piazza on the Help page if you haven't already!
- Lab signups opened earlier this week, start next week.
  - Labs posted on the course website at the start of each week
- Office Hours started earlier this week
  - **You must fill out signup questions completely when signing up**
- Please send course staff OAE letters for accommodations!

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?

Plan For Today

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- **Break:** Announcements
- **Signed Integers**
- 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

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

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

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 might get ugly!

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.

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

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.

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

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…

$$\begin{array}{r} 0101 \\ +\,1011 \\ \hline 0000 \end{array} \qquad \begin{array}{r} 0011 \\ +\,1101 \\ \hline 0000 \end{array} \qquad \begin{array}{r} 0000 \\ +\,0000 \\ \hline 0000 \end{array}$$

- 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.

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

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.

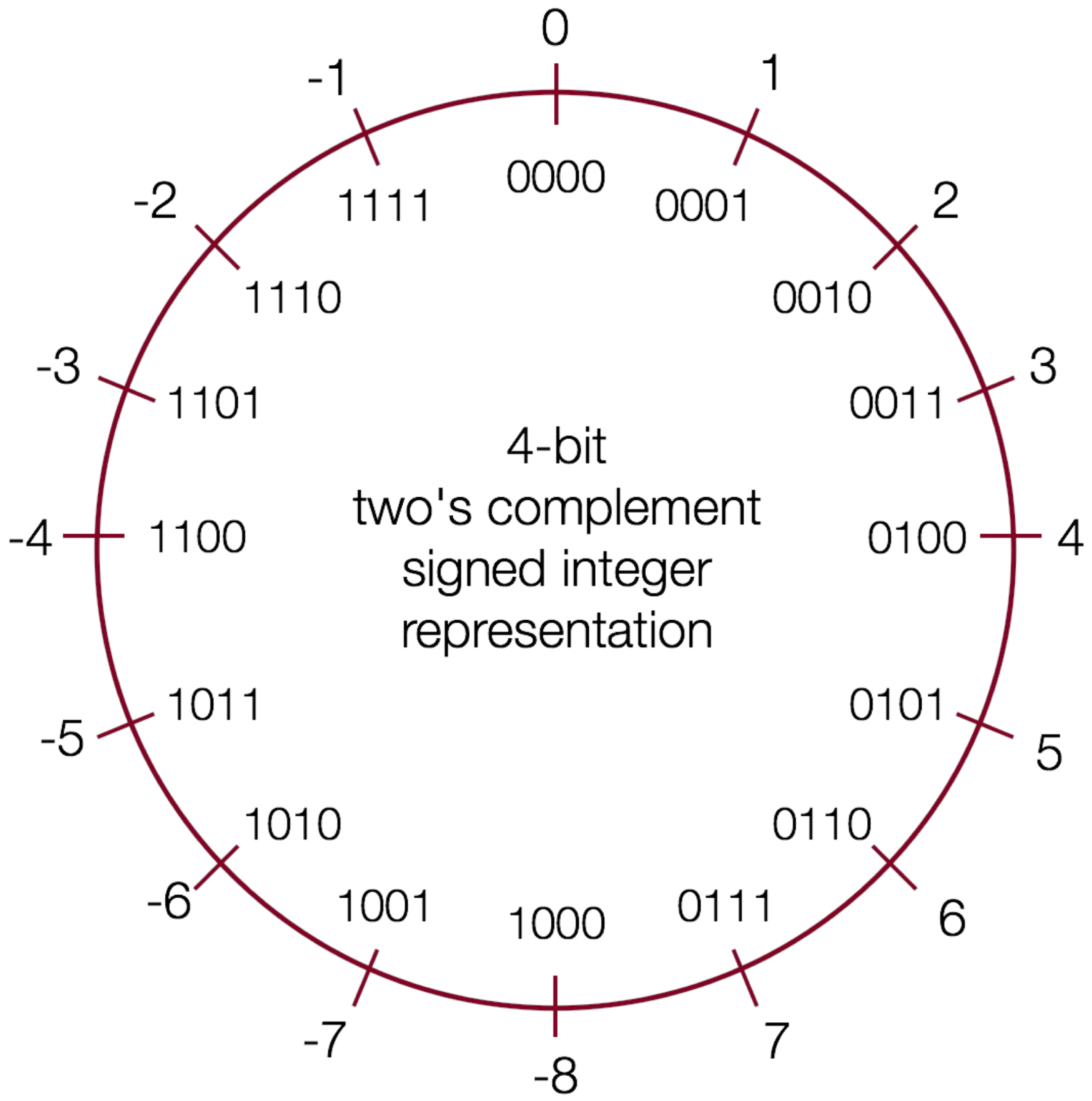
```
  100100
+ ???100
  ------
  000000
```

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.

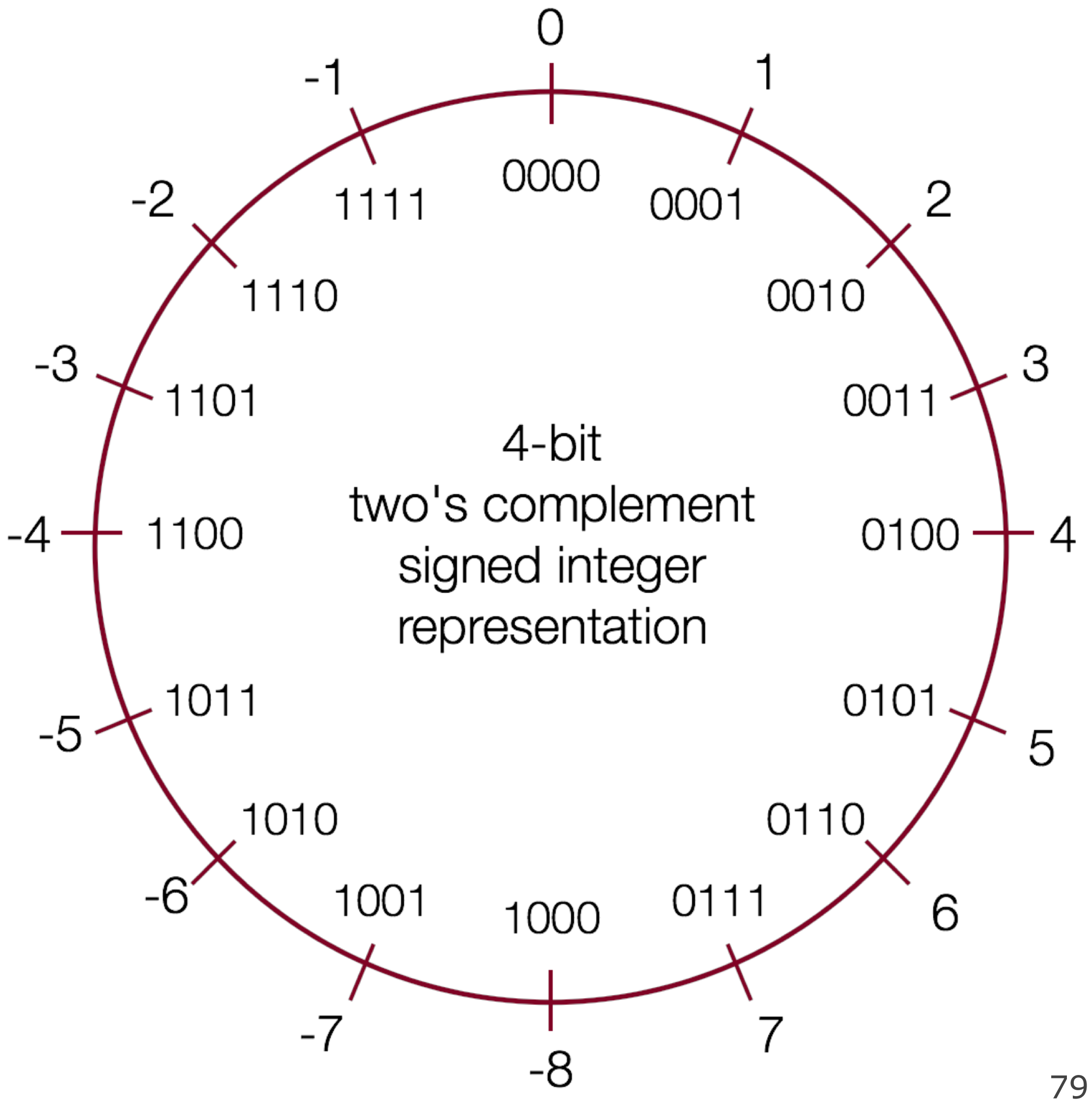
$$
\begin{array}{r}
100100 \\
+\ {\color{red}011100} \\
\hline
000000
\end{array}
$$

Two's Complement



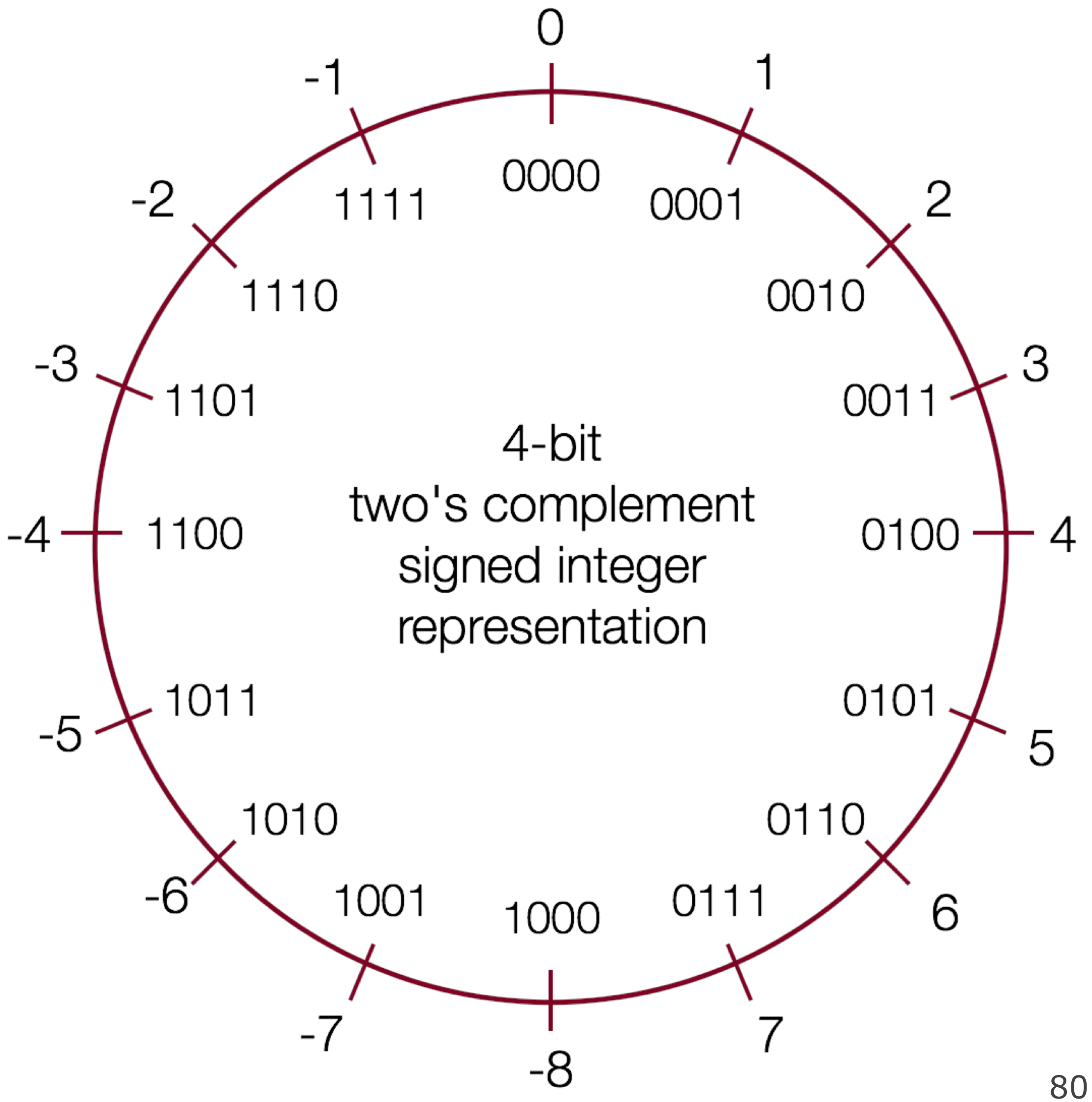
Two’s Complement

- 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:** it turns out that the most significant bit *still indicates the sign* of a number.
- **Pro:** arithmetic is easy: we just add!



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.

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

Two’s Complement

- While you don’t need to worry about multiplication, it turns out that with two’s complement, multiplying two numbers is just multiplying, and discarding overflow digits! E.g. -2 x -3 = 6.

1110 (-2)

x1101 (-3)

1110

0000

1110

+1110

~~1011~~0110 (6)

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

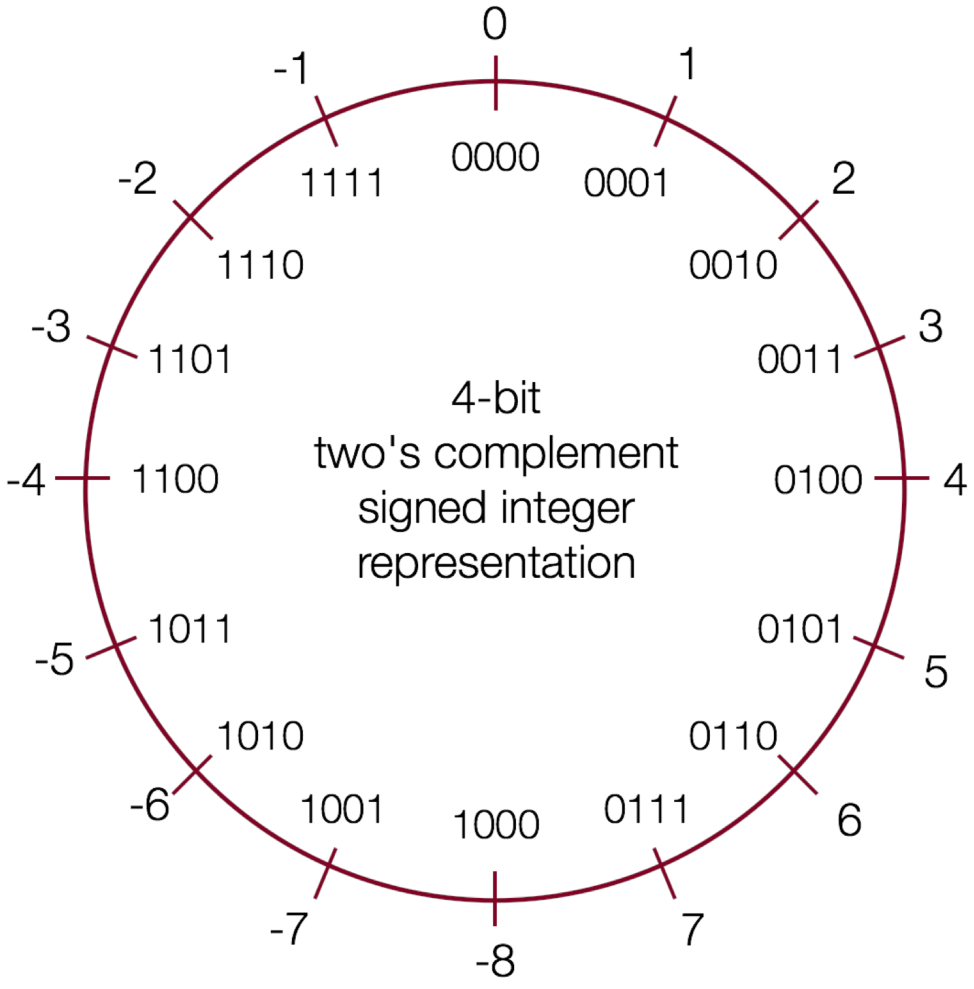
What are the negative or positive equivalents of the numbers below?

a) -4 (1100)

b) 7 (0111)

c) 3 (0011)

d) -8 (1000)



Plan For Today

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- **Break:** Announcements
- Signed Integers
- **Casting and Combining Types**

Overflow and Underflow

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

```
0b1111 + 0b1 = 0b0000
```

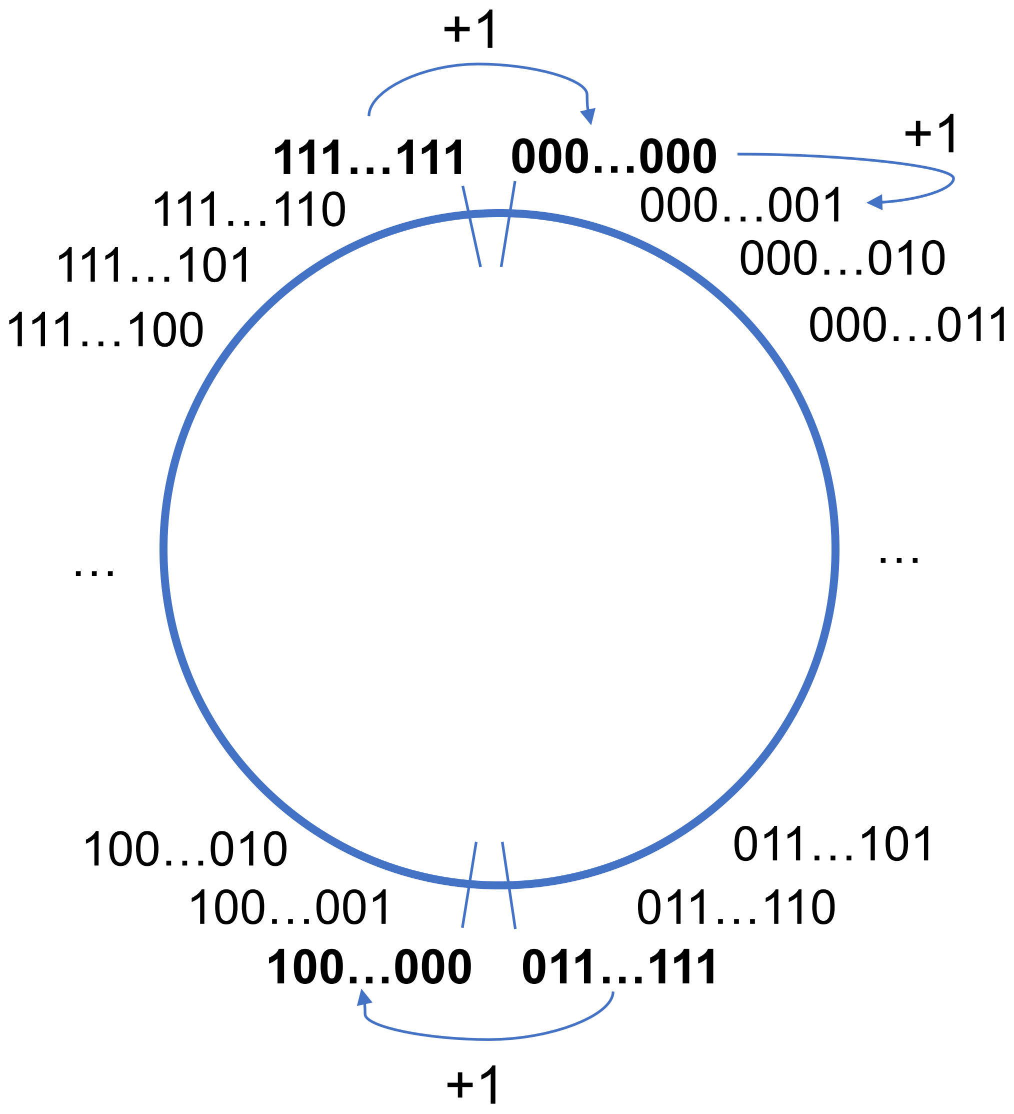
- If you go below the **minimum** value of your bit representation, you *wrap around* or *underflow* back to the **largest** bit representation.

```
0b0000 - 0b1 = 0b1111
```

Min and Max Integer Values

Type	**Width (bytes)**	**Width (bits)**	**Min in hex (name)**	**Max in hex (name)**
`char`	1	8	80 (CHAR_MIN)	7F (CHAR_MAX)
`unsigned char`	1	8	0	FF (UCHAR_MAX)
`short`	2	16	8000 (SHRT_MIN)	7FFF (SHRT_MAX)
`unsigned short`	2	16	0	FFFF (USHRT_MAX)
`int`	4	32	80000000 (INT_MIN)	7FFFFFFF (INT_MAX)
`unsigned int`	4	32	0	FFFFFFFF (UINT_MAX)
`long`	8	64	8000000000000000 (LONG_MIN)	7FFFFFFFFFFFFFFF (LONG_MAX)
`unsigned long`	8	64	0	FFFFFFFFFFFFFFFF (ULONG_MAX)

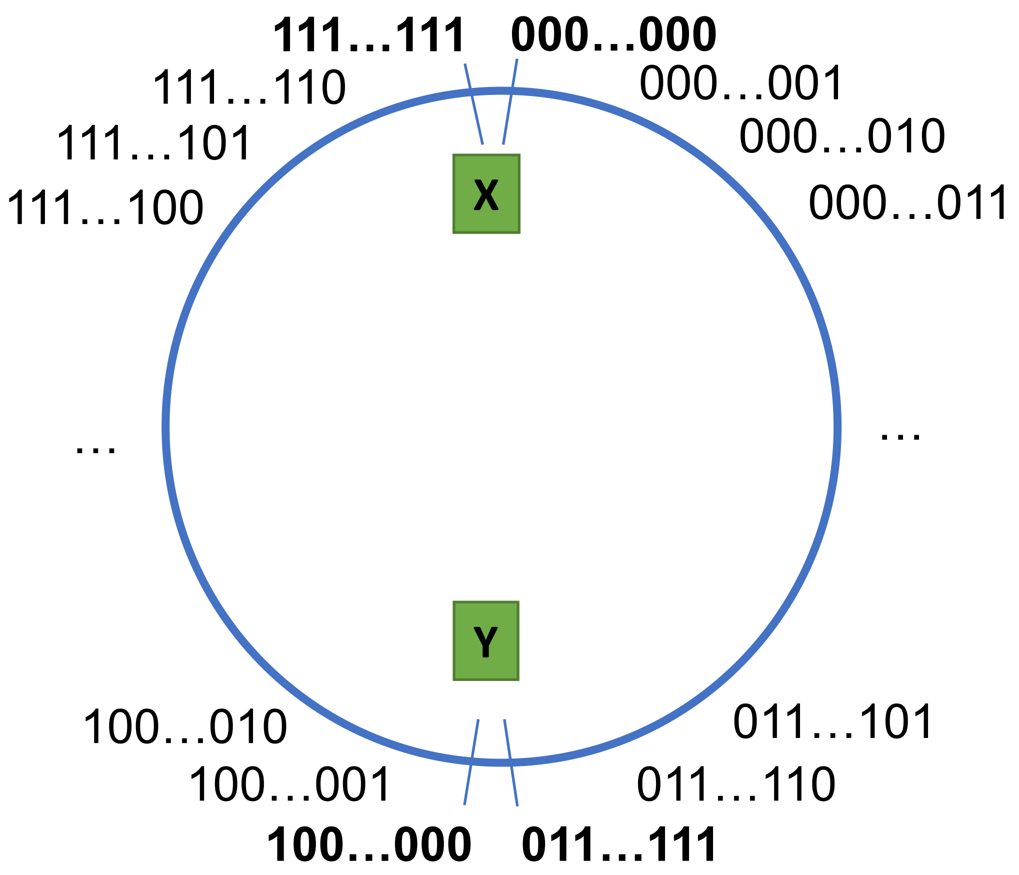
Overflow and Underflow



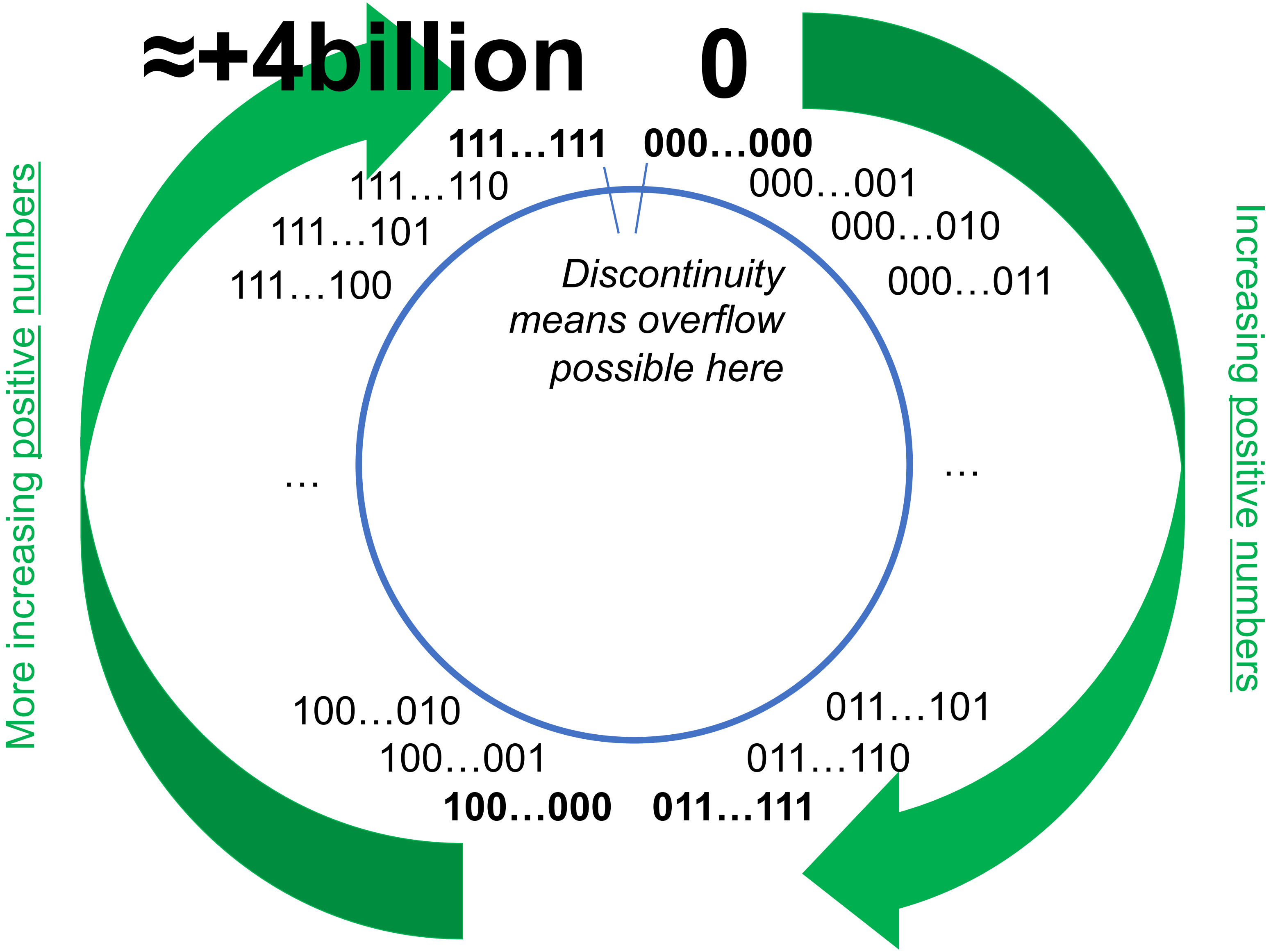
Overflow

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

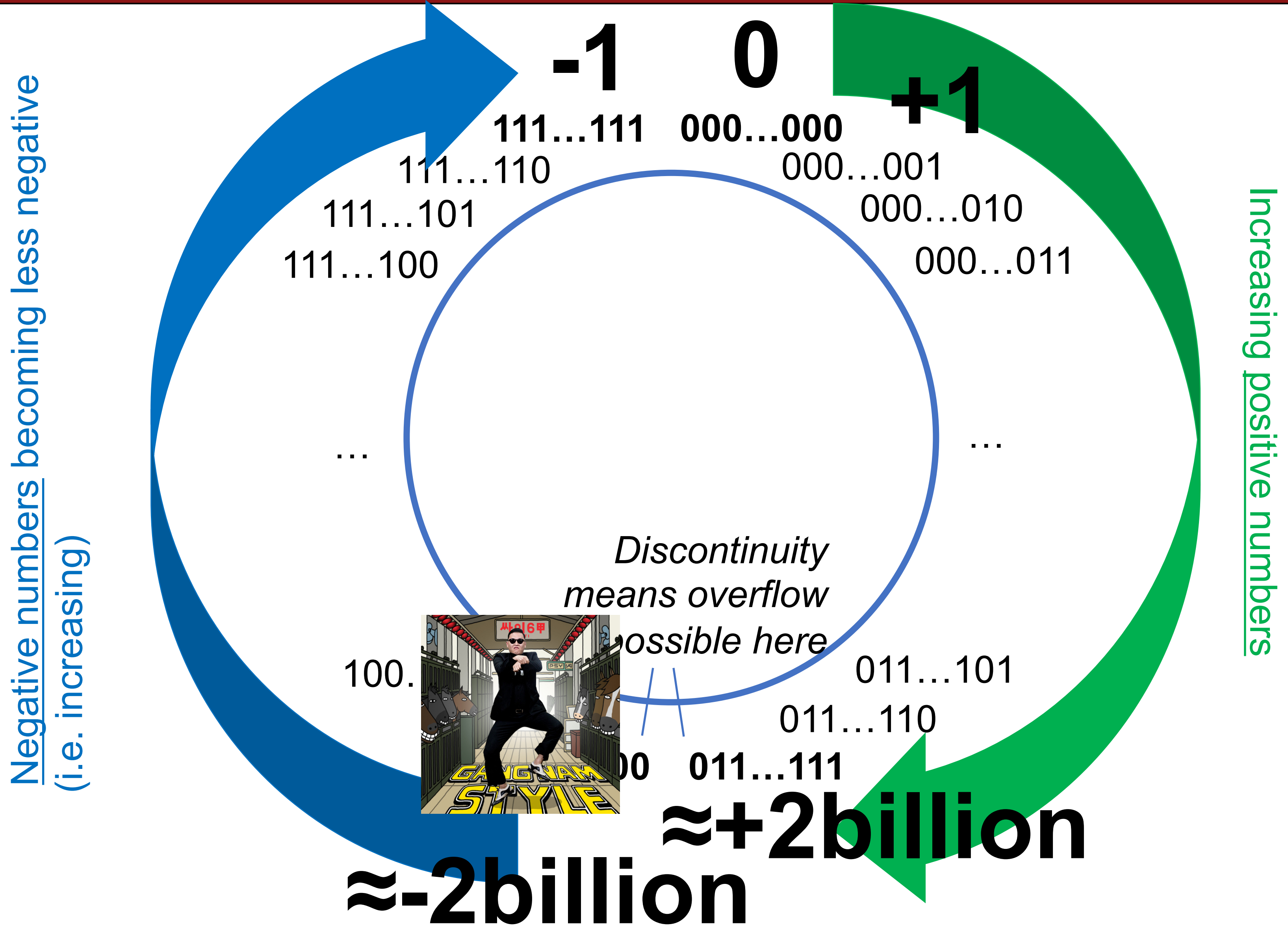
A. Signed and unsigned can both overflow at points X and Y
B. Signed can overflow only at X, unsigned only at Y
C. Signed can overflow only at Y, unsigned only at X
D. Signed can overflow at X and Y, unsigned only at X
E. Other



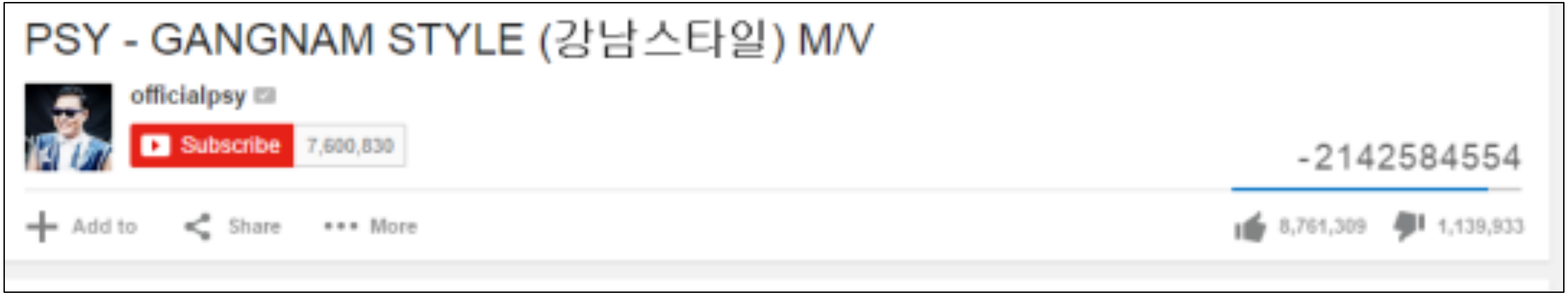
Unsigned Integers



Signed Numbers



Overflow In Practice: PSY



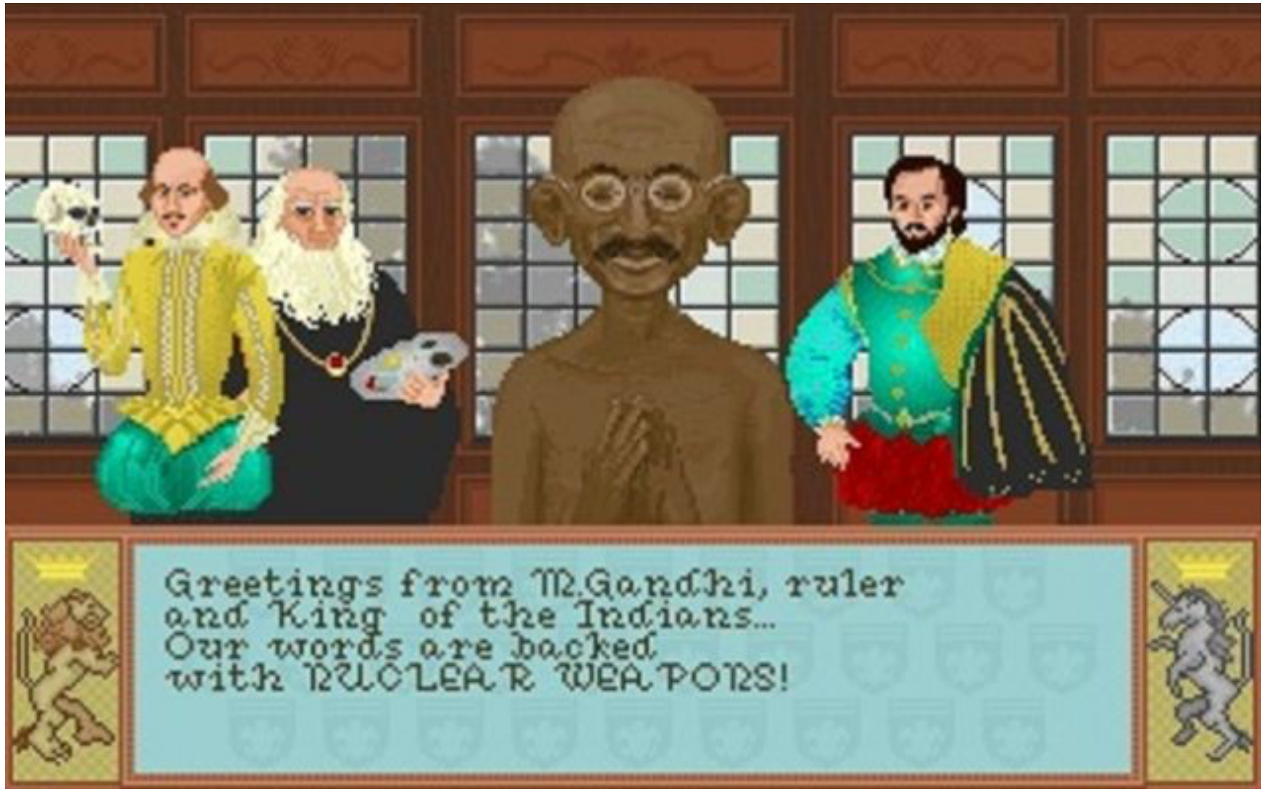
YouTube: “We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!”

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!

Underflow 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

Recap

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- **Break:** Announcements
- Signed Integers
- Casting and Combining Types

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

Extra Slides

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
- As long as the value is a 32-bit type, `printf` will **treat it according to the placeholder!**

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

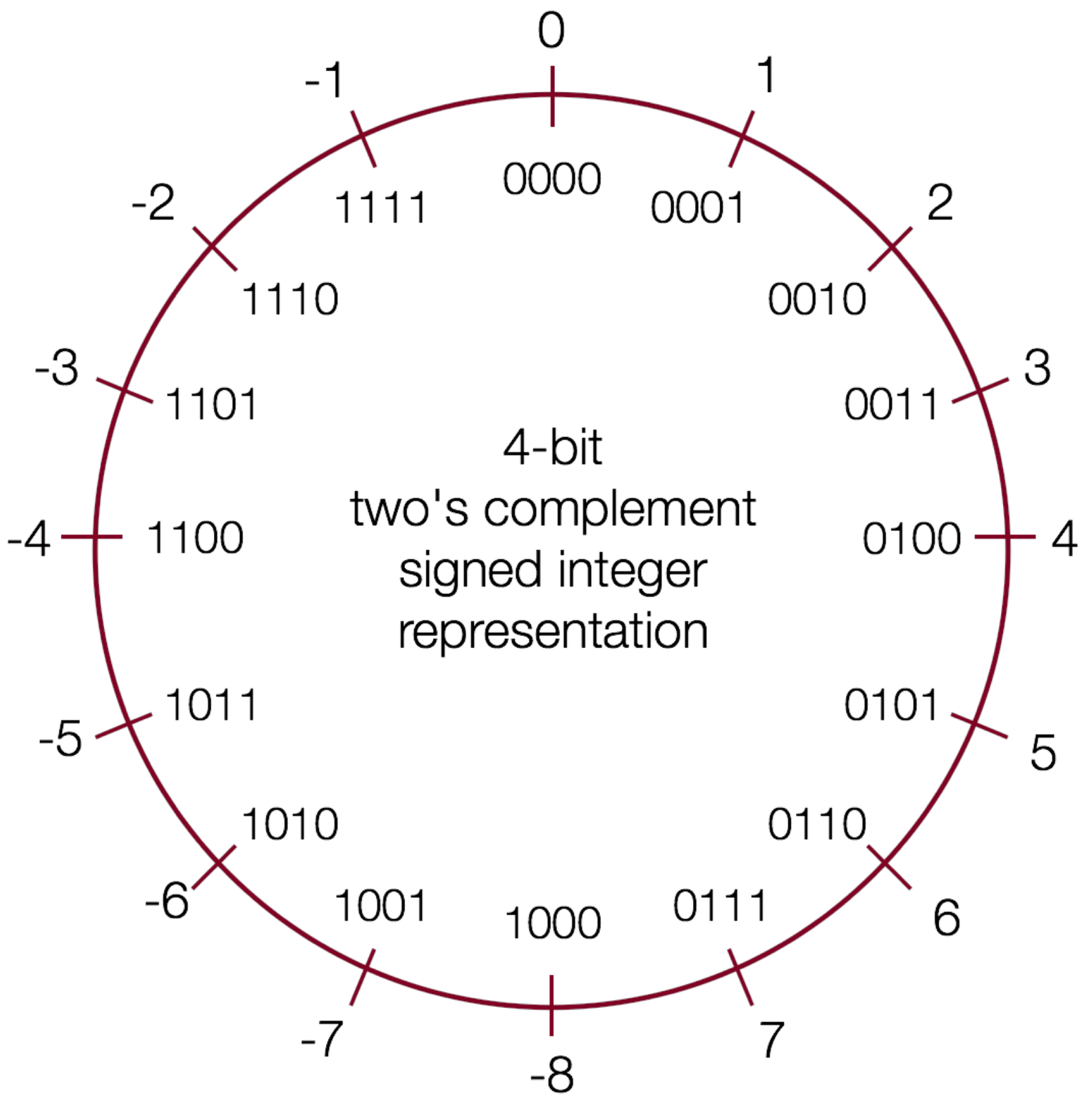
- What happens at the byte level when we cast between variable types? **The bytes remain the same!** **This means they may be interpreted differently depending on the type.**

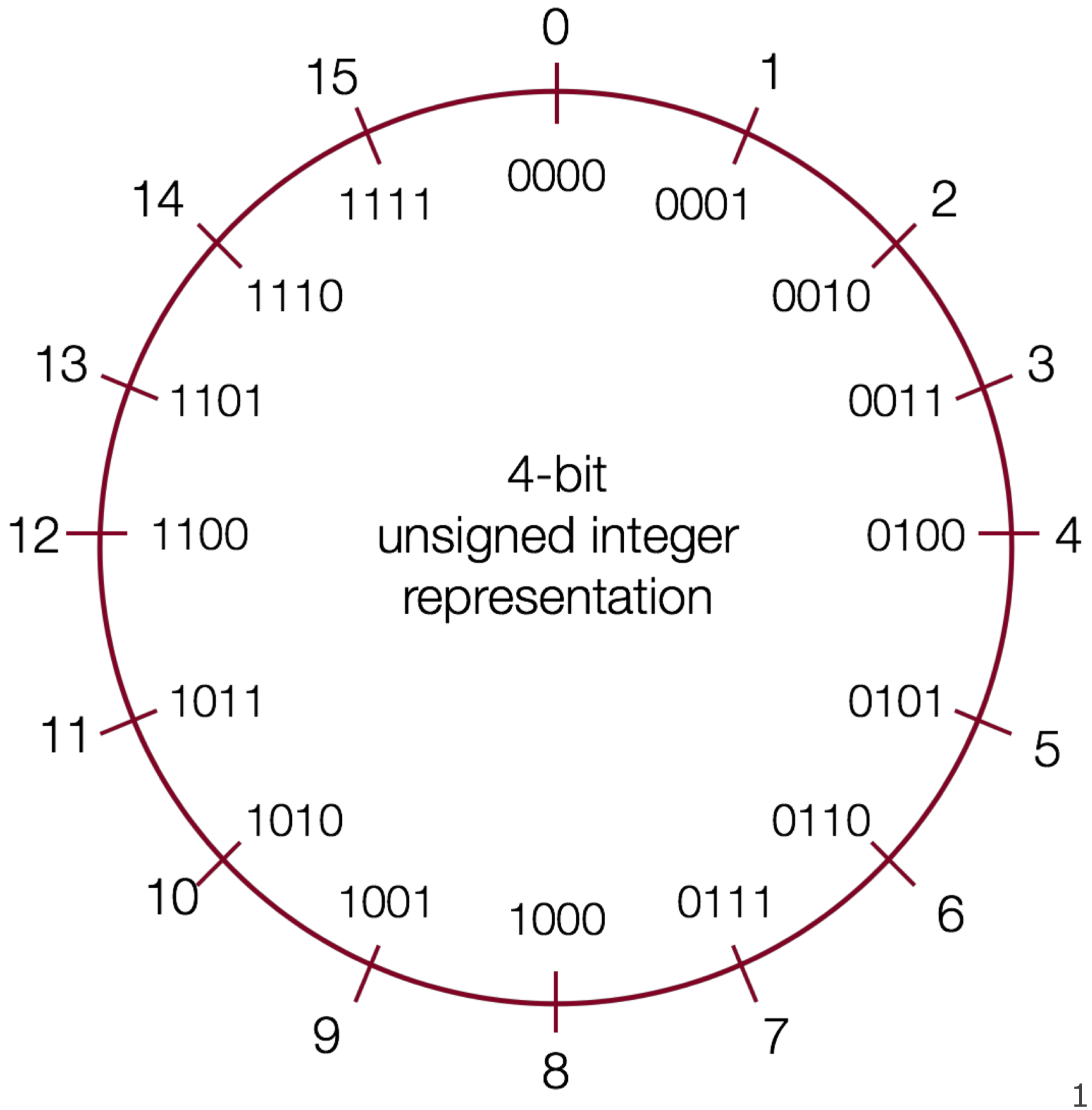
```
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

The bit representation for -12345 is **`0b11000000111001.`**

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

Casting

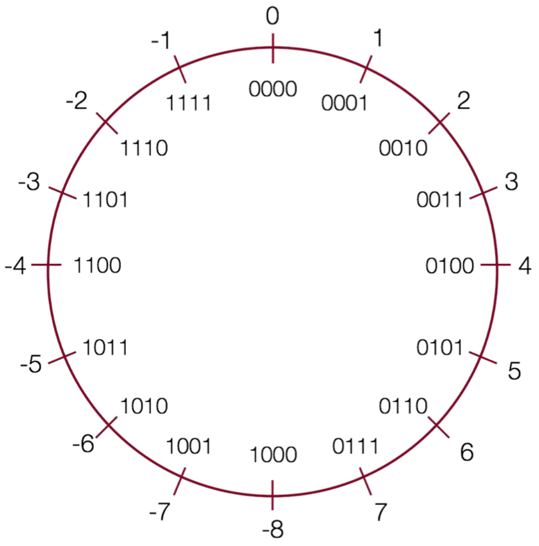

0
0000
1
0001
2
0010
3
0011
4
0100
5
0101
6
0110
7
0111
-8
1000
-7
1001
-6
1010
-5
1011
-4
1100
-3
1101
-2
1110
-1
1111
4-bit
two's complement
signed integer
representation


0
0000
1
0001
2
0010
3
0011
4
0100
5
0101
6
0110
7
0111
8
1000
9
1001
10
1010
11
1011
12
1100
13
1101
14
1110
15
1111
4-bit
unsigned integer
representation

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.

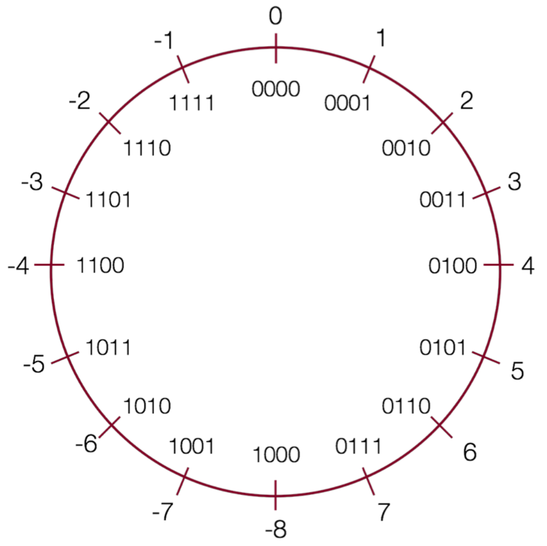
Expression	Type	Evaluation	Correct?
`0 == 0U`			
`-1 < 0`			
`-1 < 0U`			
`2147483647 > -` `2147483647 - 1`			
`2147483647U > -` `2147483647 - 1`			
`2147483647 >` `(int)2147483648U`			
`-1 > -2`			
`(unsigned)-1 > -2`			



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.

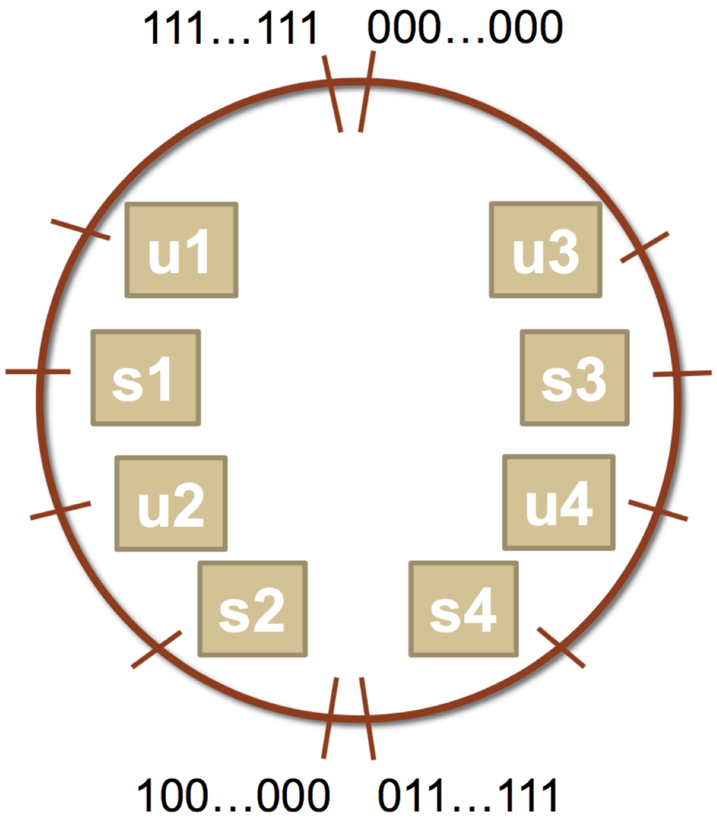
`Expression`	Type	Evaluation	Correct?
`0 == 0U`	Unsigned	1	yes
`-1 < 0`	Signed	1	yes
`-1 < 0U`	Unsigned	0	No!
`2147483647 > -2147483647 - 1`	Signed	1	yes
`2147483647U > -2147483647 - 1`	Unsigned	0	No!
`2147483647 > (int)2147483648U`	Signed	1	No!
`-1 > -2`	Signed	1	yes
`(unsigned)-1 > -2`	Unsigned	1	yes



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*)

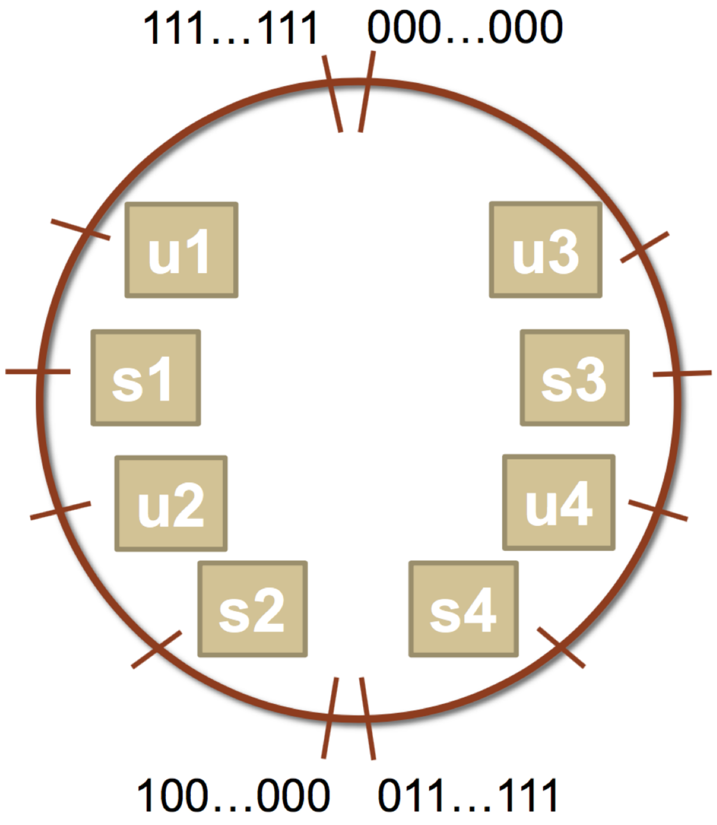
```
s3 > u3
u2 > u4
s2 > s4
s1 > s2
u1 > u2
s1 > u3
```



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*)

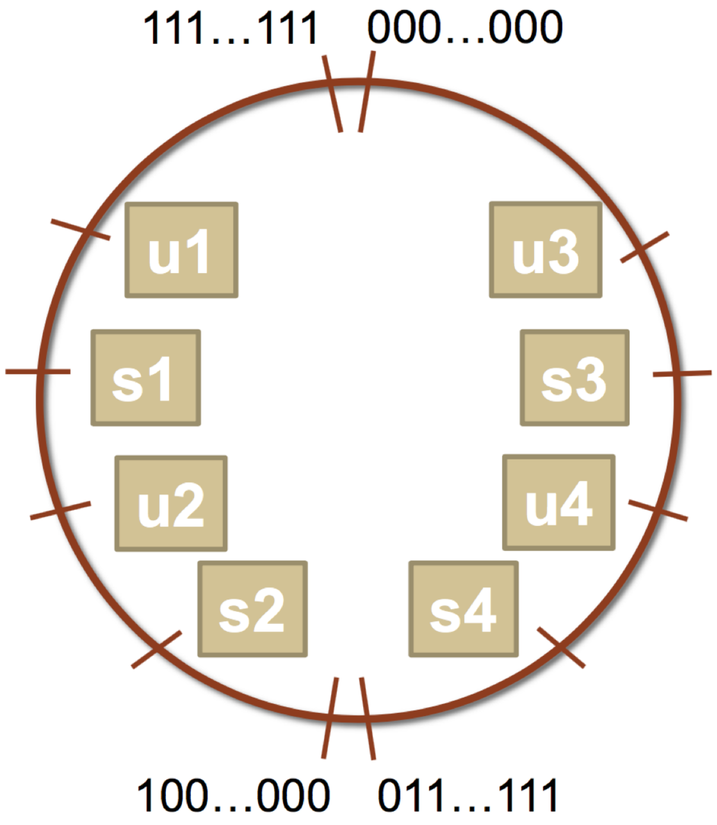
```
s3 > u3
u2 > u4
s2 > s4
s1 > s2
u1 > u2
s1 > u3
```



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 - 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

- **`sizeof`** takes a variable type as a parameter and returns the number of bytes that type uses.

```
printf("sizeof(char): %d\n", (int) sizeof(char));
printf("sizeof(short): %d\n", (int) sizeof(short));
printf("sizeof(int): %d\n", (int) sizeof(int));
printf("sizeof(unsigned int): %d\n", (int) sizeof(unsigned int));
printf("sizeof(long): %d\n", (int) sizeof(long));
printf("sizeof(long long): %d\n", (int) sizeof(long long));
printf("sizeof(size_t): %d\n", (int) sizeof(size_t));
printf("sizeof(void *): %d\n", (int) sizeof(void *));
```

```
$ ./sizeof
sizeof(char): 1
sizeof(short): 2
sizeof(int): 4
sizeof(unsigned int): 4
sizeof(long): 8
sizeof(long long): 8
sizeof(size_t): 8
sizeof(void *): 8
```

Type	Width in bytes	Width in bits
char	1	8
short	2	16
int	4	32
long	8	64
void *	8	64