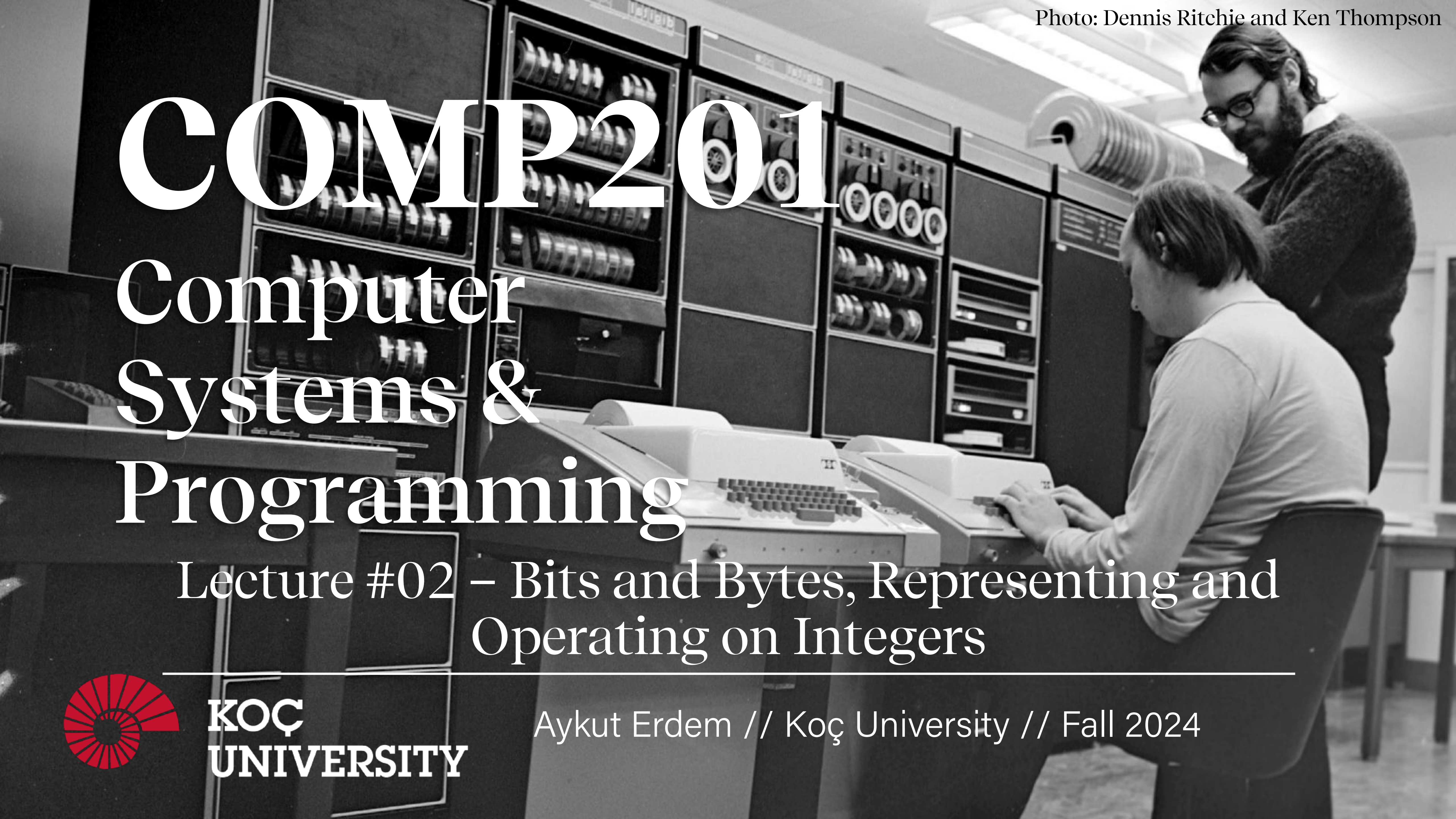


Photo: Dennis Ritchie and Ken Thompson

COMP201

Computer Systems & Programming

Lecture #02 – Bits and Bytes, Representing and Operating on Integers

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2024

Recap

- Course Introduction
- COMP201 Course Policies
- Unix and the Command Line
- Getting Started With C

Plan For Today

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

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
—Randal E. Bryant and David R. O'Hallaron's CMU 15-213 class

COMP201 Topic 1: How can a computer represent integer numbers?

THE WALL STREET JOURNAL.

US Air, Comair Scramble To Get Back to Normal

A Wall Street Journal NEWS ROUNDUP
Dec. 27, 2004 12:01 am ET

Air travelers on two airlines continued to face canceled flights and lost luggage after weather, worker absences and computer glitches left thousands stranded in airports over the holiday weekend. Comair Inc., a regional carrier owned by Delta Air Lines that canceled its entire schedule Saturday, resumed limited flights yesterday but said it wouldn't return to normal until midweek.

US Airways Group Inc. blamed more than 400 canceled flights and thousands of pieces of stranded luggage on large numbers of workers who called in sick, as well as on a heavy winter storm. A spokesman said the carrier had no evidence of a concerted job action, but the troubles underscore the problems low morale could cause the carrier as it struggles to emerge from bankruptcy-court protection.

It was unclear how many holiday travelers were affected, though the major disruptions appeared to be limited to US Airways and Comair. UAL Corp.'s United Airlines and Northwest Airlines reported weather difficulties in Chicago and Detroit, respectively. AMR Corp.'s American Airlines said it experienced problems due to unusual snowfall at its Dallas-Fort Worth hub over the weekend

Demo: Unexpected Behavior

`airline.c`

Lecture Plan

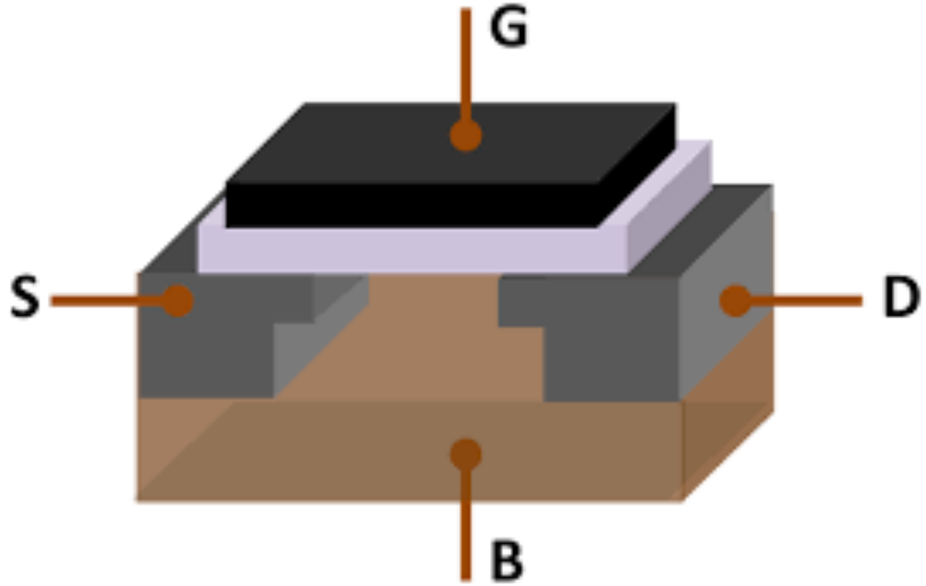
- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Casting and Combining Types

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

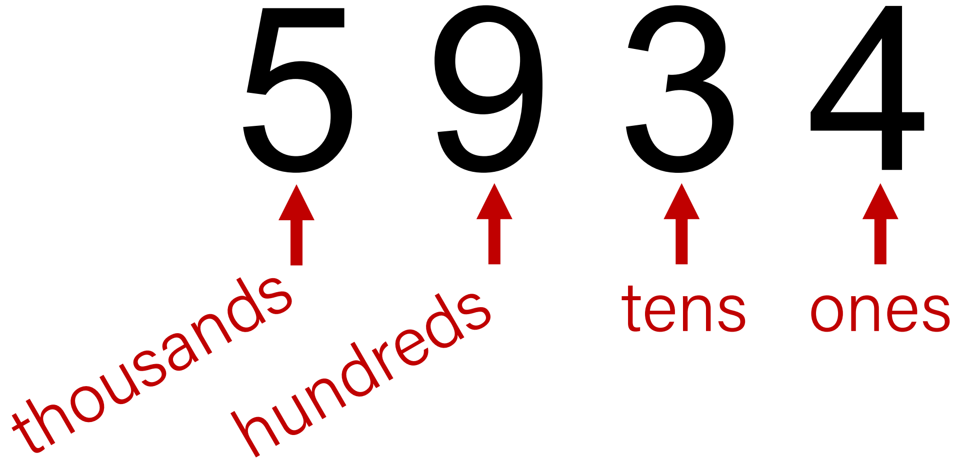
- We can combine bits, like with base-10 numbers, to represent more data. **8 bits = 1 byte**.
- Computer memory is just a large array of bytes! It is *byte-addressable*; you can't address (store location of) a bit; only a byte.
- Computers still fundamentally operate on bits; we have just gotten more 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***1000 + **9***100 + **3***10 + **4***1

Base 10

5 9 3 4

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

Base 10

5 9 3 4

10^X: 3 2 1 0

Base 2

1 0 1 1

2^X: 3 2 1 0

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

Base 2

1 0 1 1

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

Base 2

Most significant bit (MSB)

Least significant bit (LSB)

1 0 1 1

eights fours twos ones

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

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

0	1	1	_
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	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!

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

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

Practice: Base 2 to Base 10

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

a) 20
b) 101
c) 10
d) 5
e) Other

Practice: Base 10 to Base 2

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

a) 1111
b) 1110
c) 1010
d) Other

Byte Values

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

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 = ?**

11111111

2^x: 7 6 5 4 3 2 1 0

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

Byte Values

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

11111111

2^x: 7 6 5 4 3 2 1 0

- **Strategy 1:** $1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 255$
- **Strategy 2:** $2^8 - 1 = 255$

Multiplying by Base

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

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

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

Hexadecimal

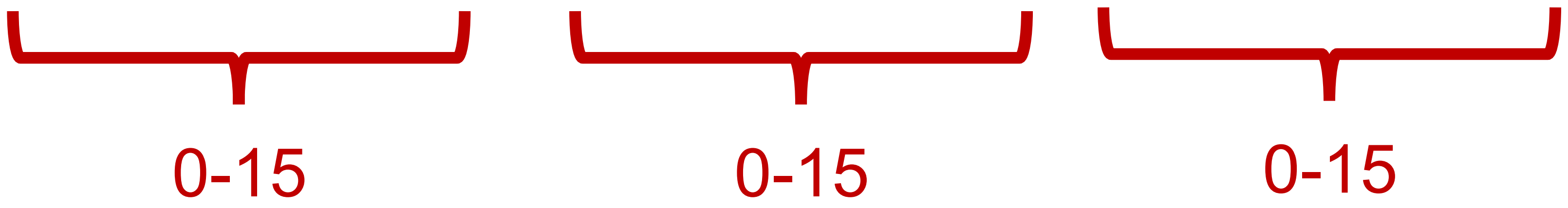
- When working with bits, oftentimes we have large numbers with 32 or 64 bits.
- Instead, we'll represent bits in *base-16 instead;* this is called **hexadecimal.**

0110 1010 0011

0-15 0-15 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**.



Each is a base-16 digit!

Hexadecimal

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

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

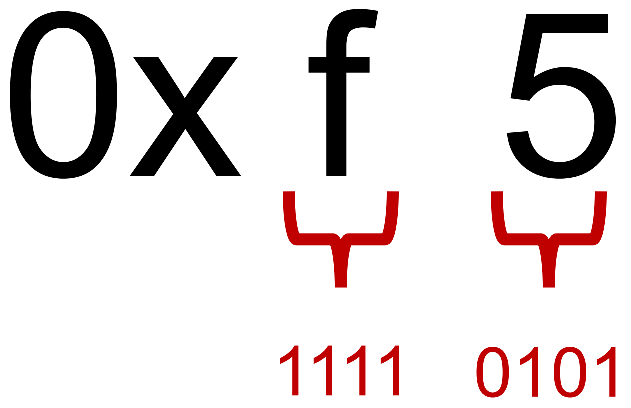
Hexadecimal

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Hexadecimal

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



Practice: Hexadecimal to Binary

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

Hexadecimal	**1**	**7**	**3**	**A**
Binary	**0001**	**0111**	**0011**	**1010**

Practice: Hexadecimal to Binary

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

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

Question Break!

Lecture Plan

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- 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, $1.5x10^{12}$)

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, $1.5x10^{12}$)

→ More on this next week!

Number Representations

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

In The Days Of Yore…

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

Transitioning To Larger Datatypes

- **Early 2000s:** most computers were **32-bit.** This means that pointers were **4 bytes (32 bits).**
- 32-bit pointers store a memory address from 0 to 2^{32}-1, equaling **2^{32} bytes of addressable memory**. This equals **4 Gigabytes**, meaning that 32-bit computers could have at most **4GB** of memory (RAM)!
- Because of this, computers transitioned to **64-bit.** This means that datatypes were enlarged; pointers in programs were now **64 bits.**
- 64-bit pointers store a memory address from 0 to 2^{64}-1, equaling **2^{64} bytes of addressable memory.** This equals **16 Exabytes**, meaning that 64-bit computers could have at most **1024*1024*1024 GB** of memory (RAM)!

Lecture Plan

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

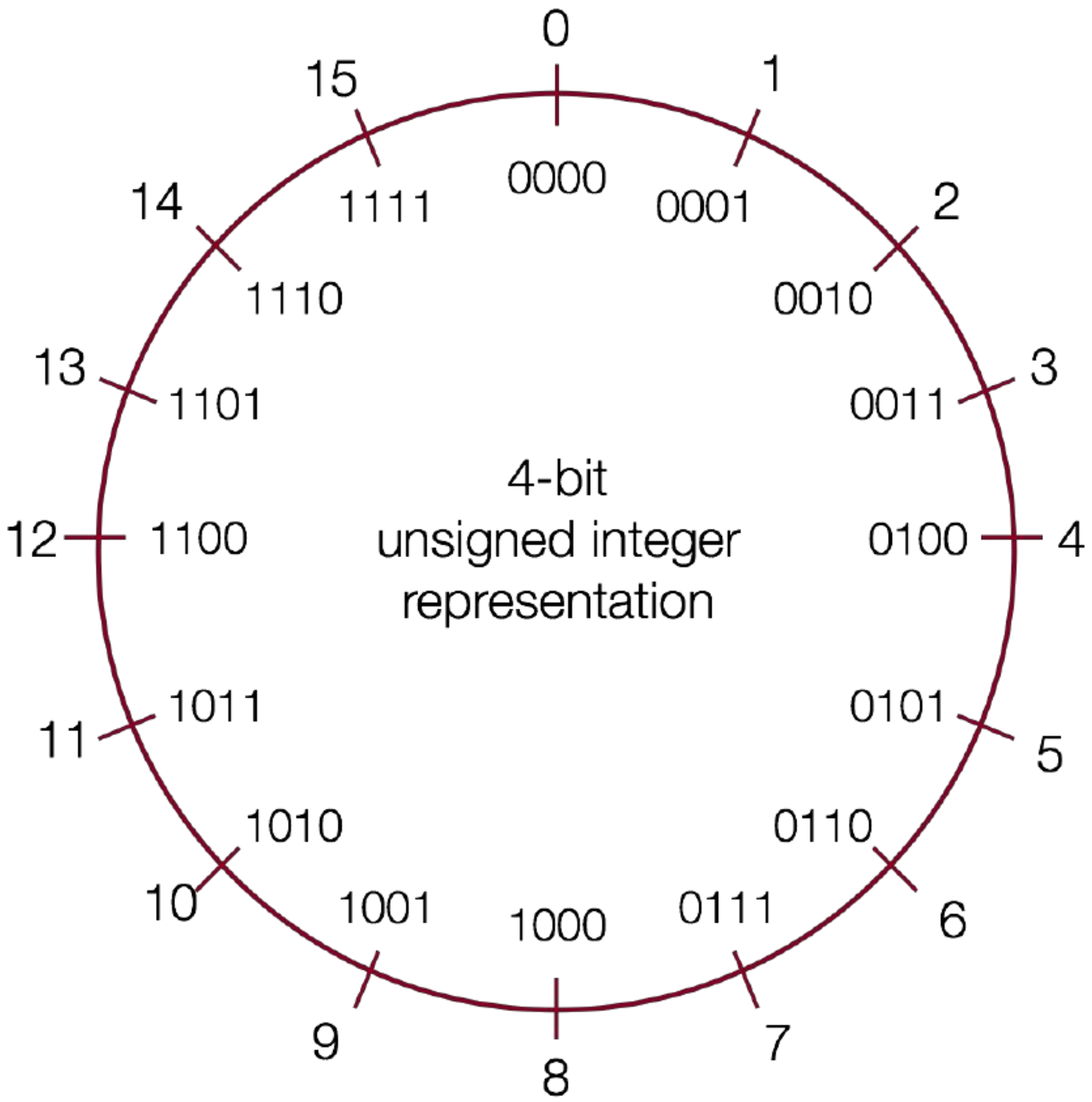
Unsigned Integers

- 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 → 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



Lecture Plan

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

0110

positive 6

1011

negative 3

Sign Magnitude Representation

0000

positive 0

1000

negative 0

Sign Magnitude Representation

1 000 = -0	0 000 = 0
1 001 = -1	0 001 = 1
1 010 = -2	0 010 = 2
1 011 = -3	0 011 = 3
1 100 = -4	0 100 = 4
1 101 = -5	0 101 = 5
1 110 = -6	0 110 = 6
1 111 = -7	0 111 = 7

- We've only represented 15 of our 16 available numbers!

Sign Magnitude Representation

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

Can we do better?

A Better Idea

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

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

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.

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

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…

0101	0011	0000
+ 1011	+ 1101	+ 0000
0000	0000	0000

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

There Seems Like a Pattern Here…

A binary number plus its inverse is all 1s.

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

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

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

Another Trick

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

```
  100100
+ ??????
--------
  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.

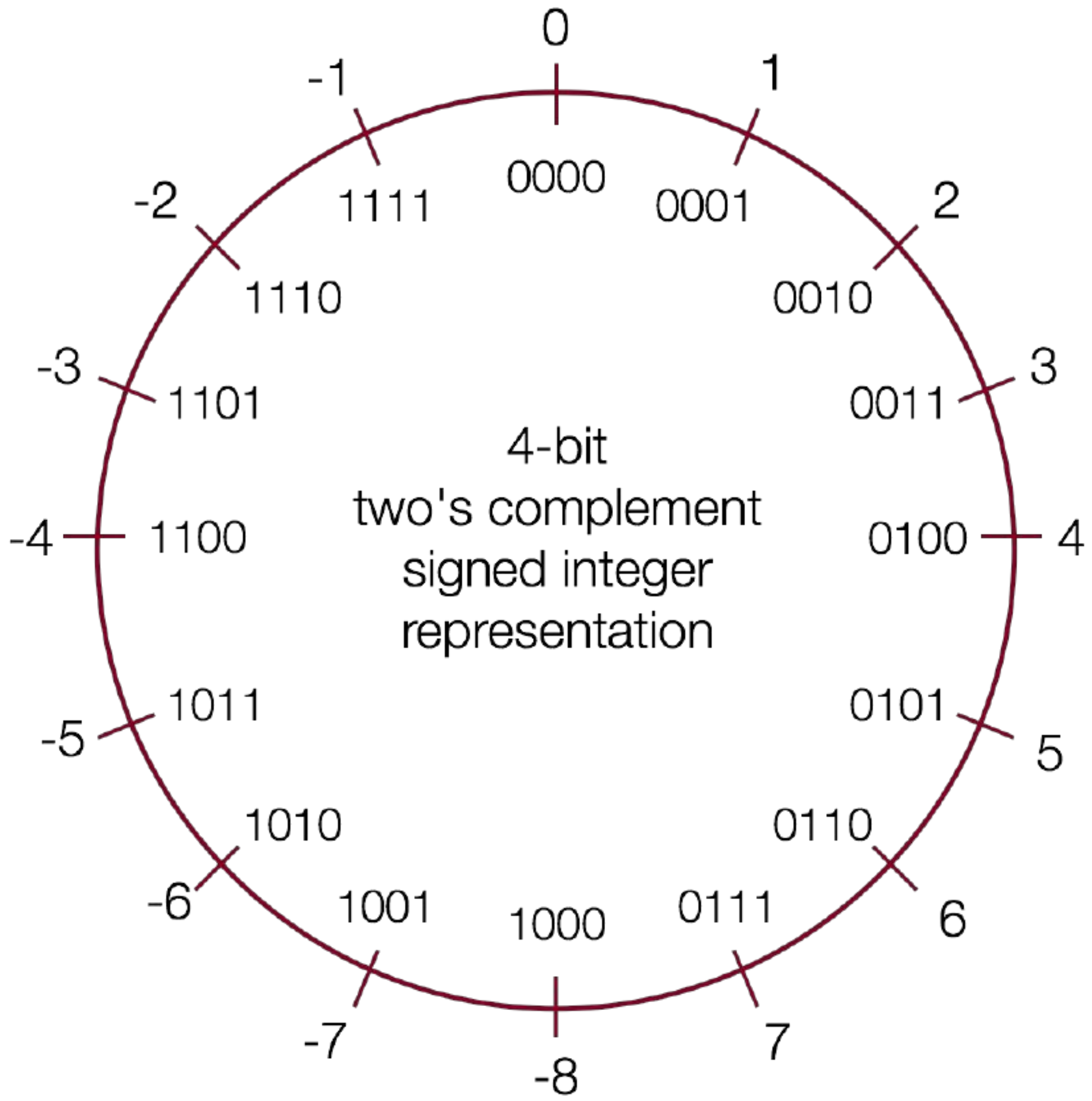
```
  100100
+ ???100
--------
  000000
```

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.

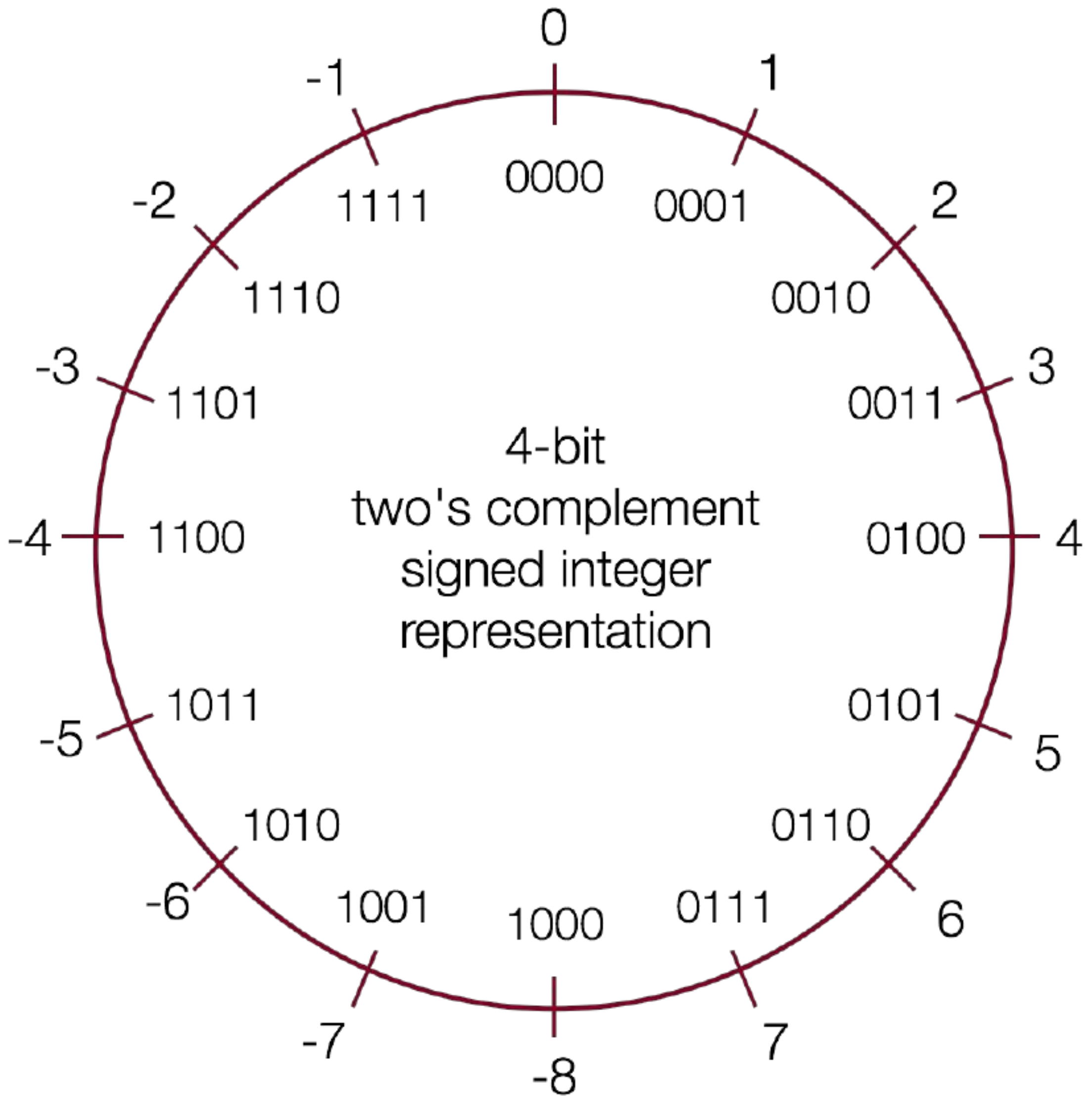
$$
\begin{array}{r}
100100 \\
+\,011100 \\
\hline
000000
\end{array}
$$

Two's Complement



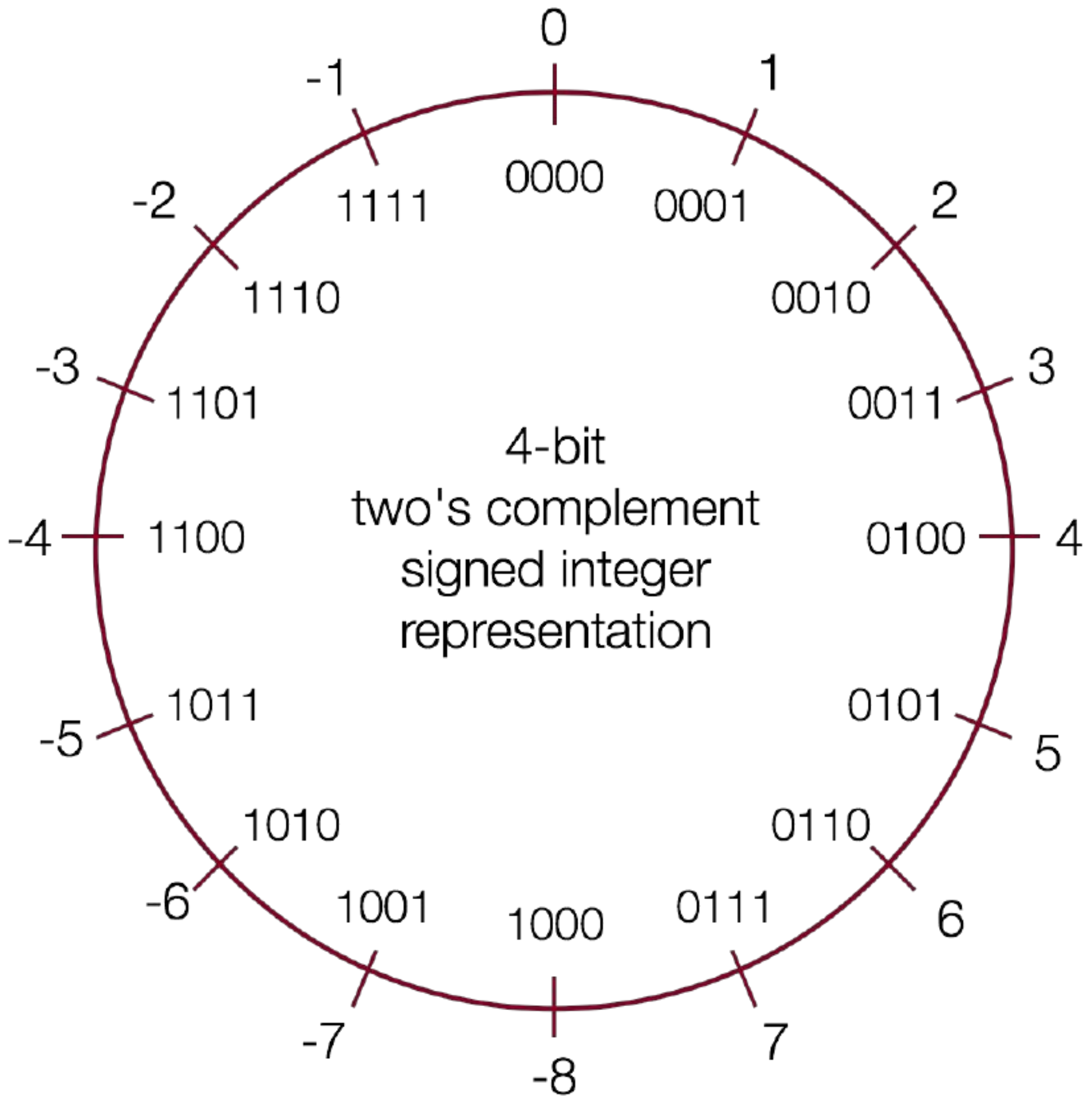
Two's Complement

- 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?

	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
-	0101	5

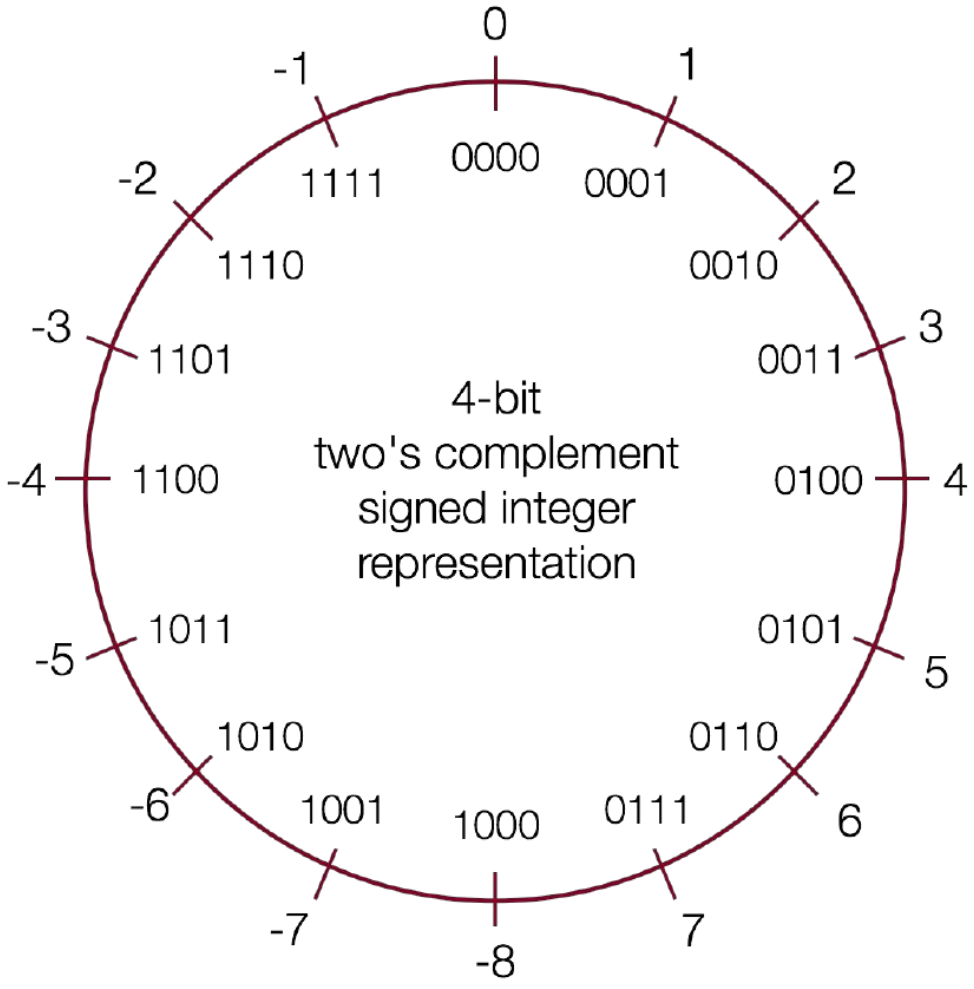
→

	0100	4
+	1011	-5
	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

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- 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.

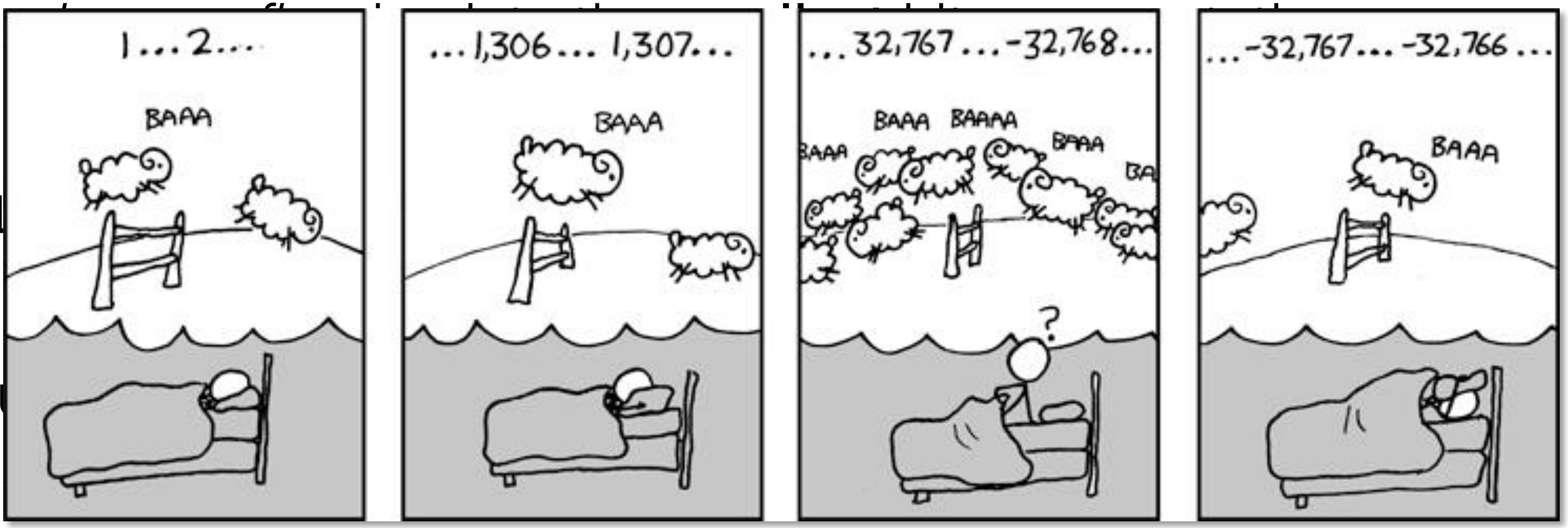
```
0b0000 - 0b1 = 0b1111
```

Overflow

- If you exceed the **maximum** value of your bit representation, you *wrap arou*

0b111

- If yo *wrap arou*



https://xkcd.com/571 **Can't Sleep**

Title text: If androids someday DO dream of electric sheep, don't forget to declare sheepCount as a long int.

`0b0000 - 0b1 = 0b1111`

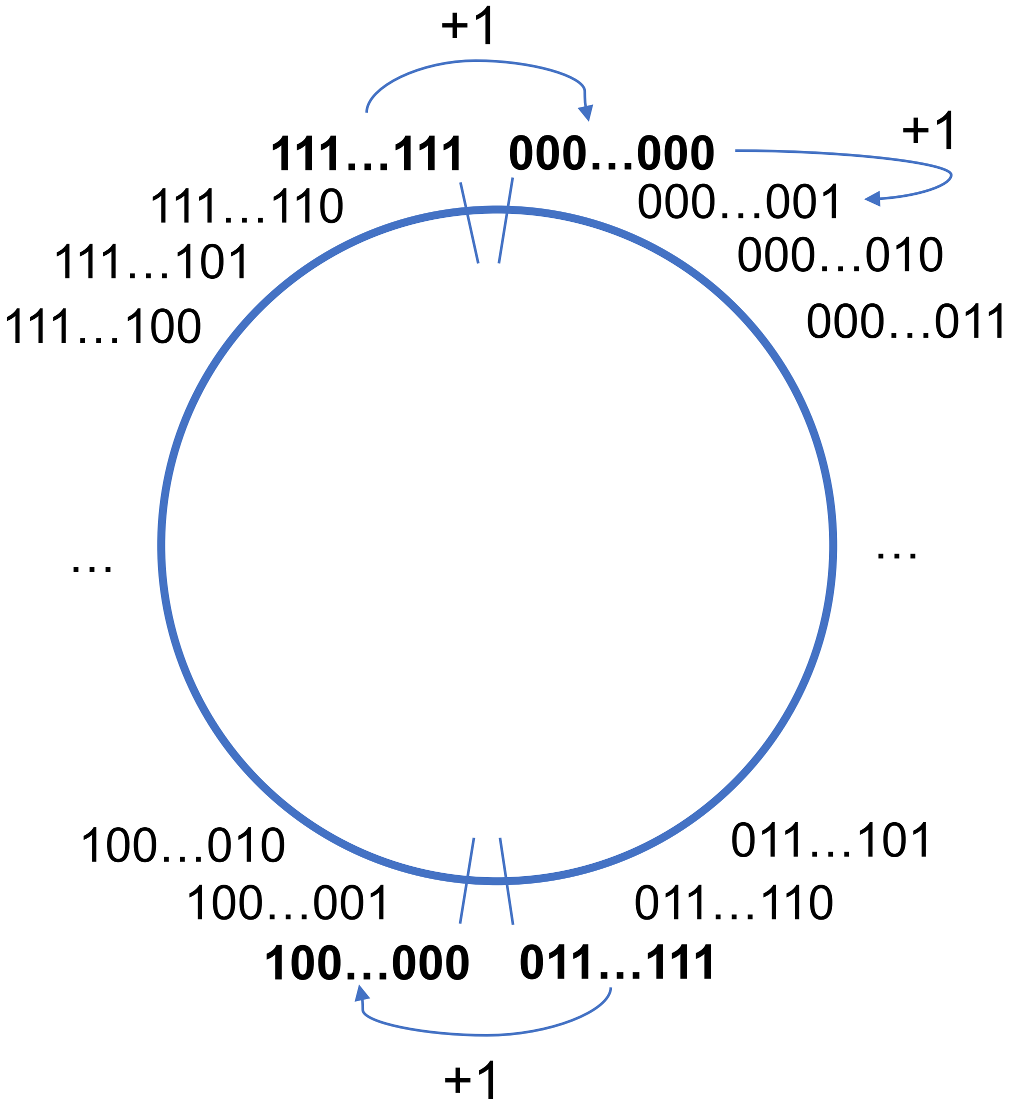
Min and Max Integer Values

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


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

Practice: Overflow

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

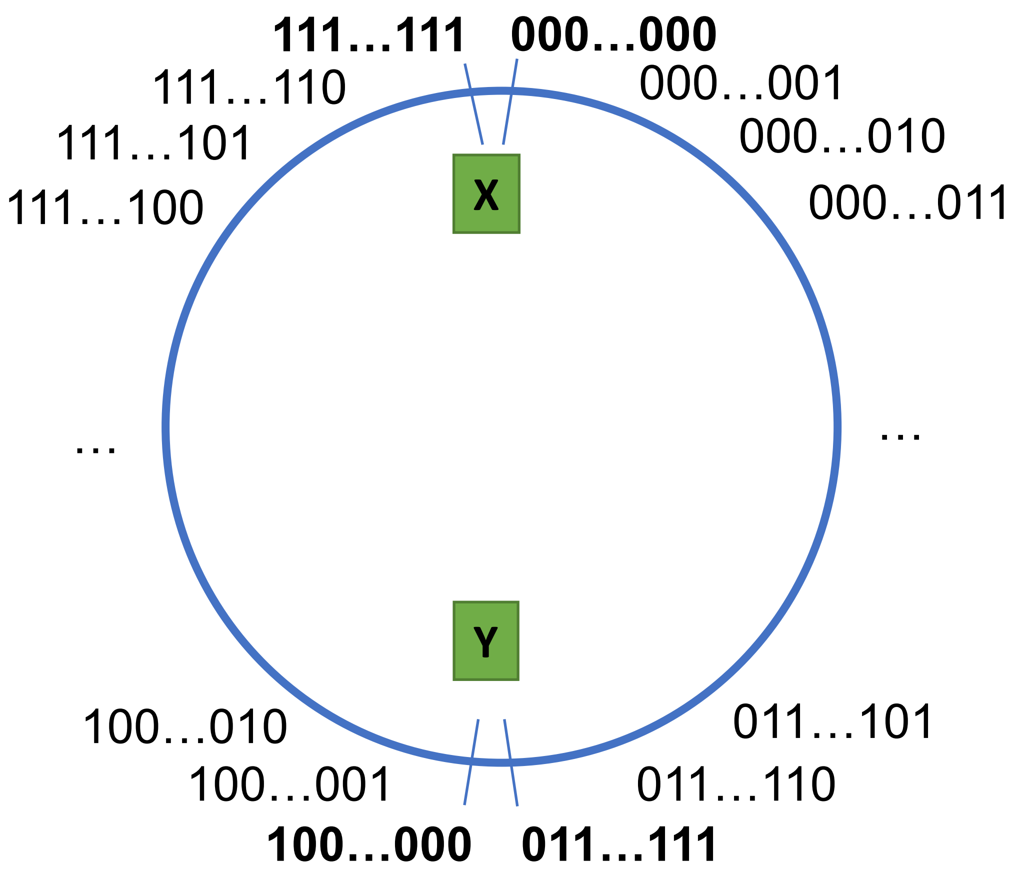
A. Signed and unsigned can both overflow at points X and Y

B. Signed can overflow only at X, unsigned only at Y

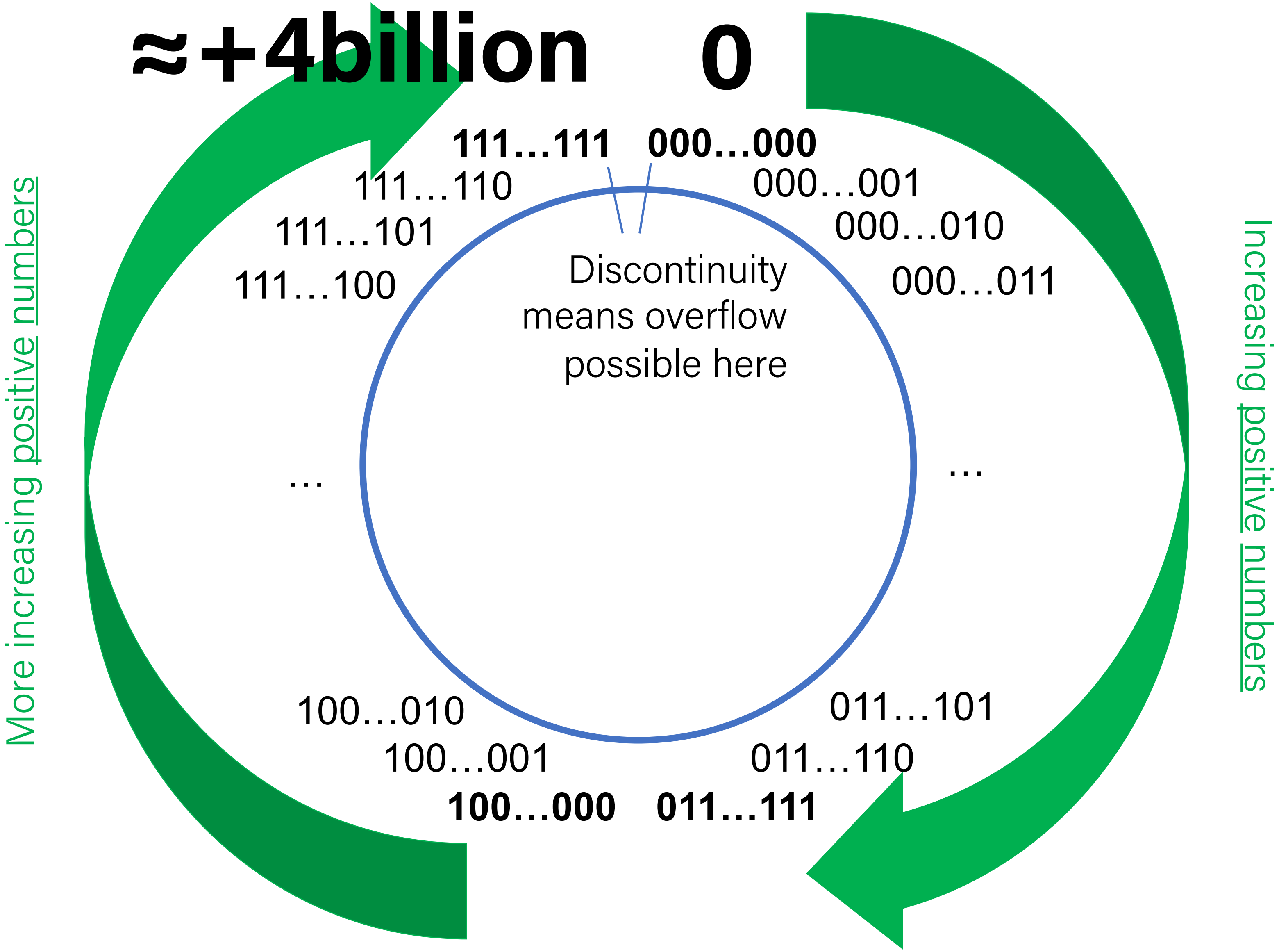
C. Signed can overflow only at Y, unsigned only at X

D. Signed can overflow at X and Y, unsigned only at X

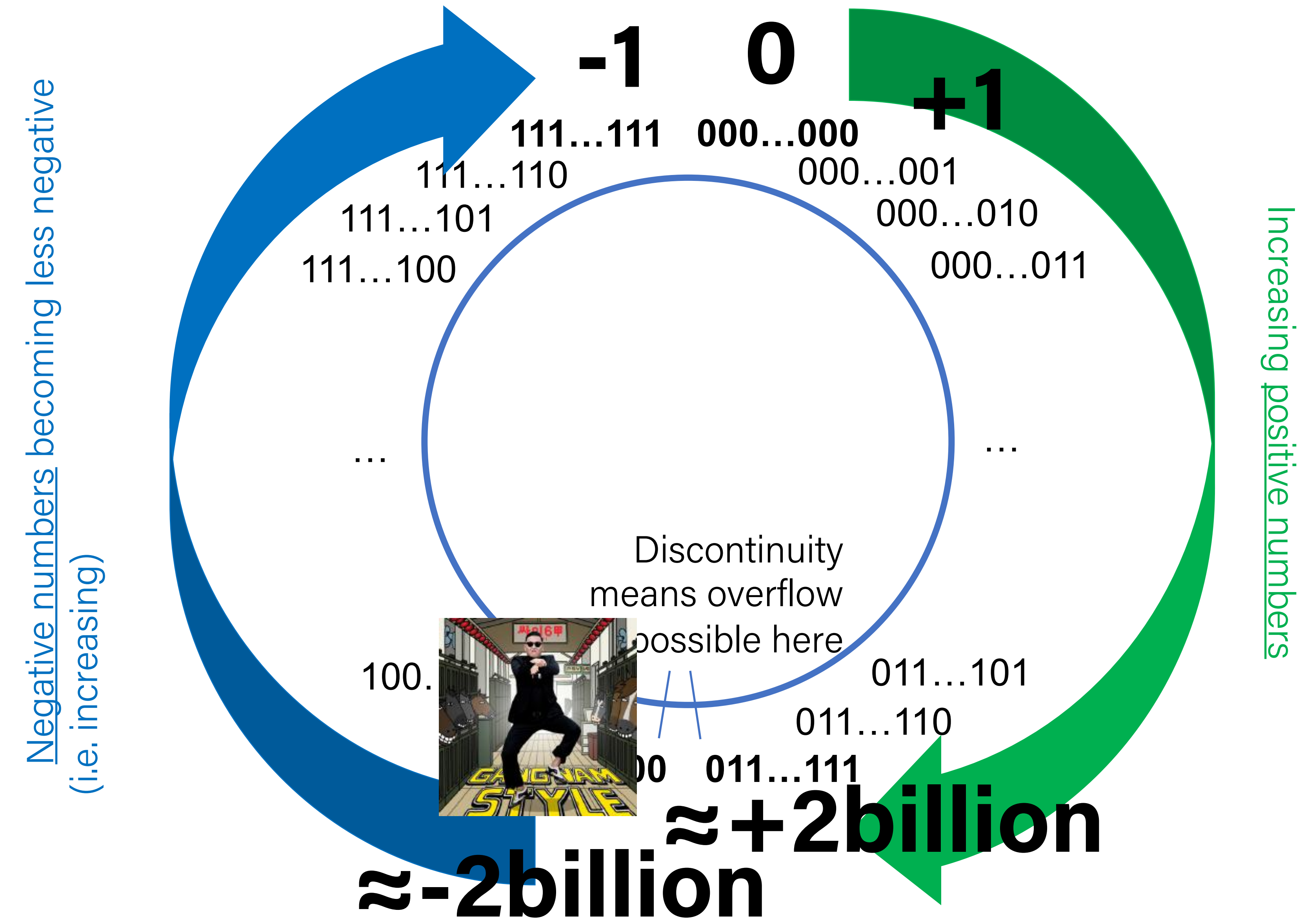
E. Other



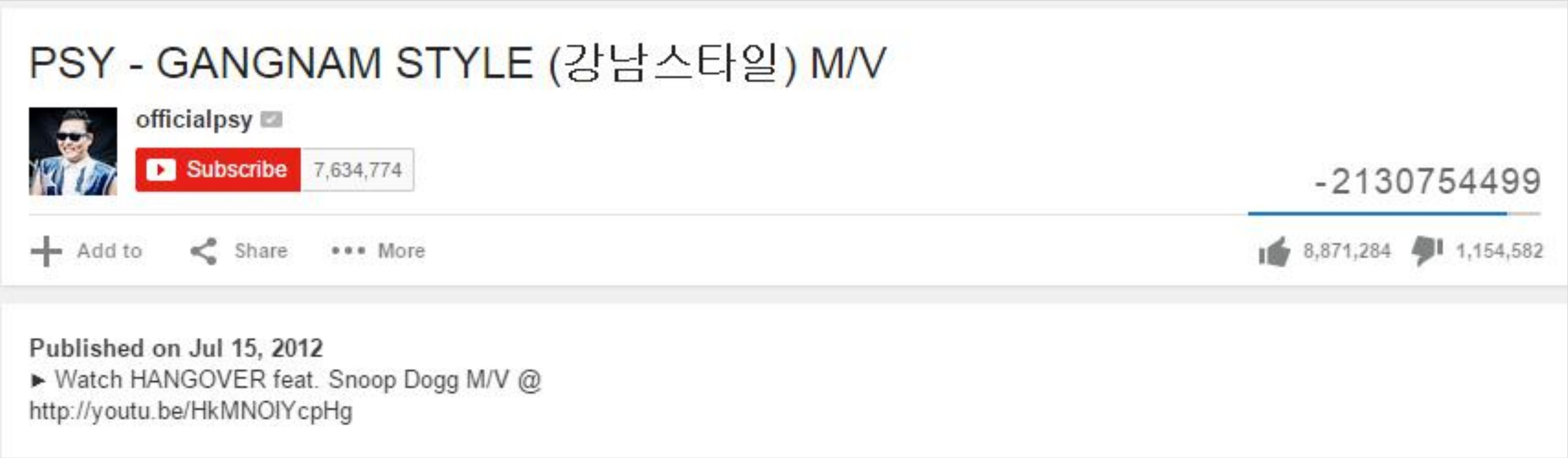
Unsigned Integers



Signed Numbers



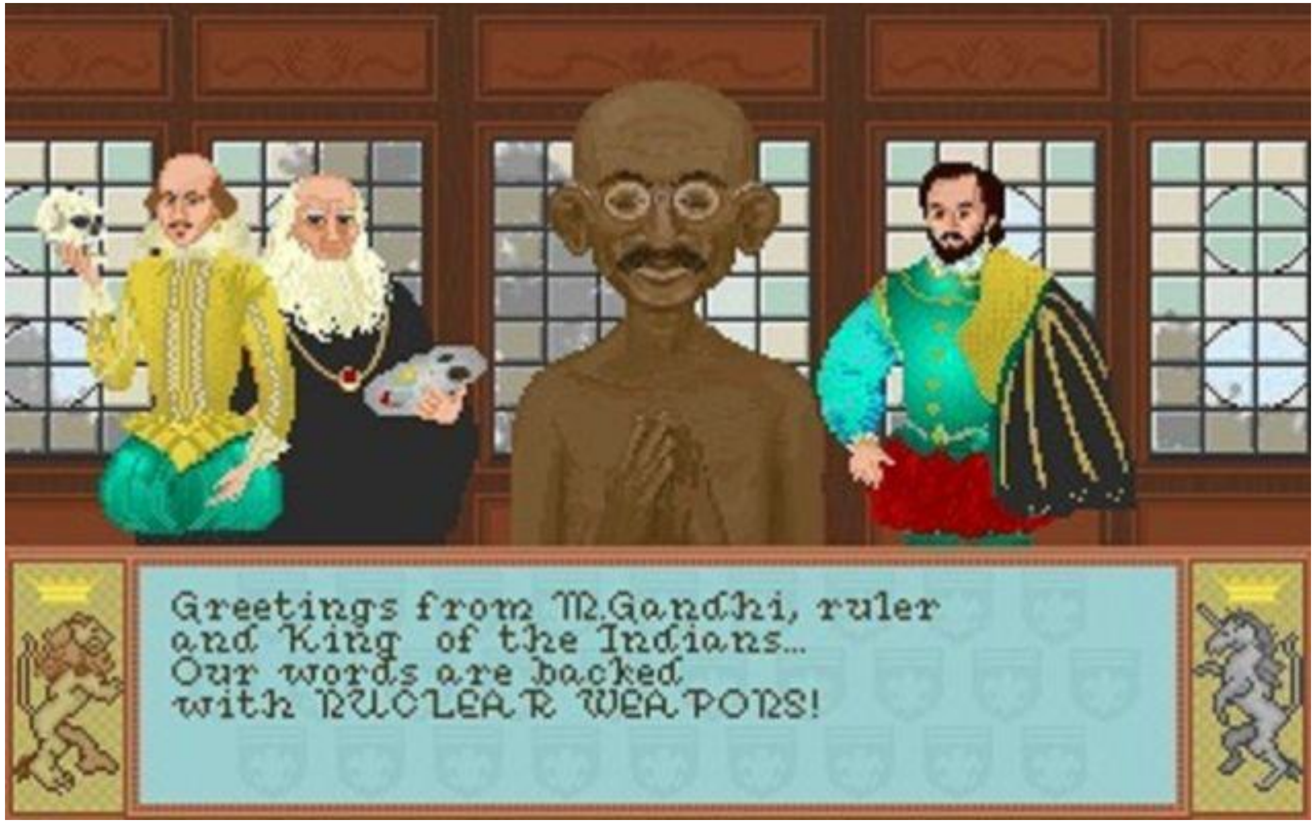
Overflow In Practice: PSY



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: Gandhi

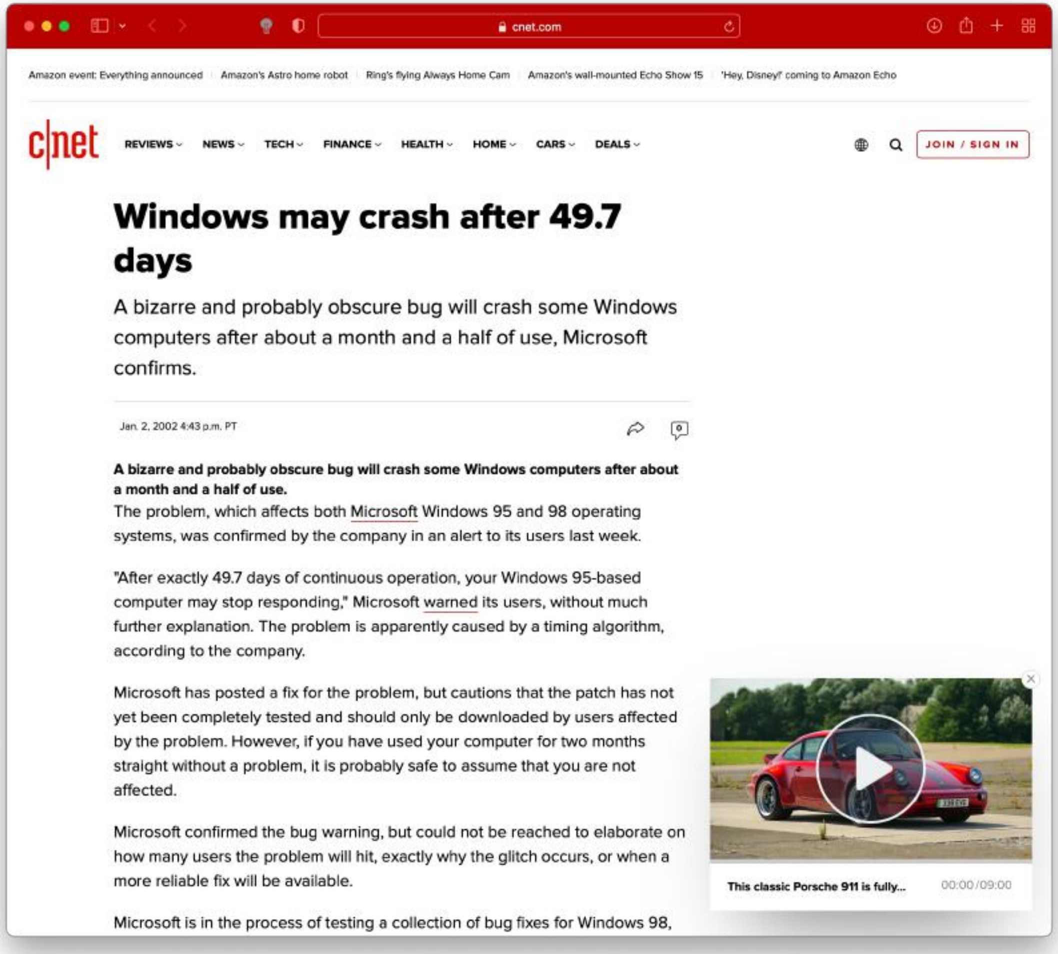
- In the game “Civilization”, each civilization leader had an “aggression” rating. Gandhi was meant to be peaceful, and had a score of 1.
- If you adopted “democracy”, all players’ aggression reduced by 2. Gandhi’s went from 1 to **255**!
- Gandhi then became a big fan of nuclear weapons.



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

Windows 95 can only run for 49.7 days before crashing,

- Windows 95 was unable to run longer than 49.7 days of runtime!
- There exists `GetTickTime` function – part of the Windows API – which returns the number of milliseconds which has elapsed since the system has started up as a 32-bit uint.
- And there's 86M ms in a day, i.e. 1000 * 60 * 60 * 24 = 86,400,000 and 32 bits is 4,294,967,296 so 4,294,967,296 / 86,400,000 = 49.7102696 days!



Windows may crash after 49.7 days

A bizarre and probably obscure bug will crash some Windows computers after about a month and a half of use, Microsoft confirms.

Jan. 2, 2002 4:43 p.m. PT

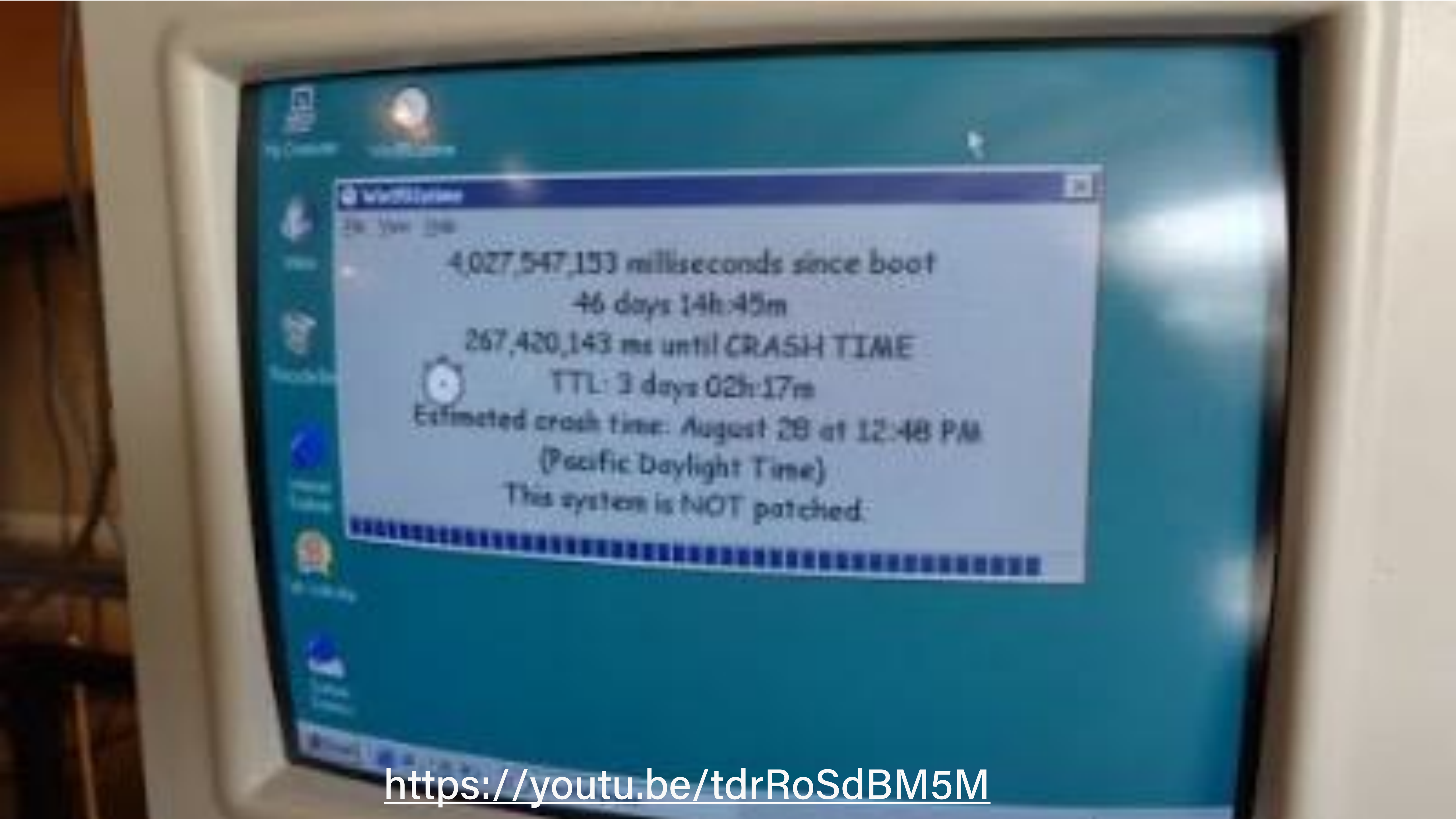
A bizarre and probably obscure bug will crash some Windows computers after about a month and a half of use.
The problem, which affects both Microsoft Windows 95 and 98 operating systems, was confirmed by the company in an alert to its users last week.

"After exactly 49.7 days of continuous operation, your Windows 95-based computer may stop responding," Microsoft warned its users, without much further explanation. The problem is apparently caused by a timing algorithm, according to the company.

Microsoft has posted a fix for the problem, but cautions that the patch has not yet been completely tested and should only be downloaded by users affected by the problem. However, if you have used your computer for two months straight without a problem, it is probably safe to assume that you are not affected.

Microsoft confirmed the bug warning, but could not be reached to elaborate on how many users the problem will hit, exactly why the glitch occurs, or when a more reliable fix will be available.

Microsoft is in the process of testing a collection of bug fixes for Windows 98,



https://youtu.be/tdrRoSdBM5M

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
- Hexadecimal
- Integer Representations
- Unsigned Integers
- 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

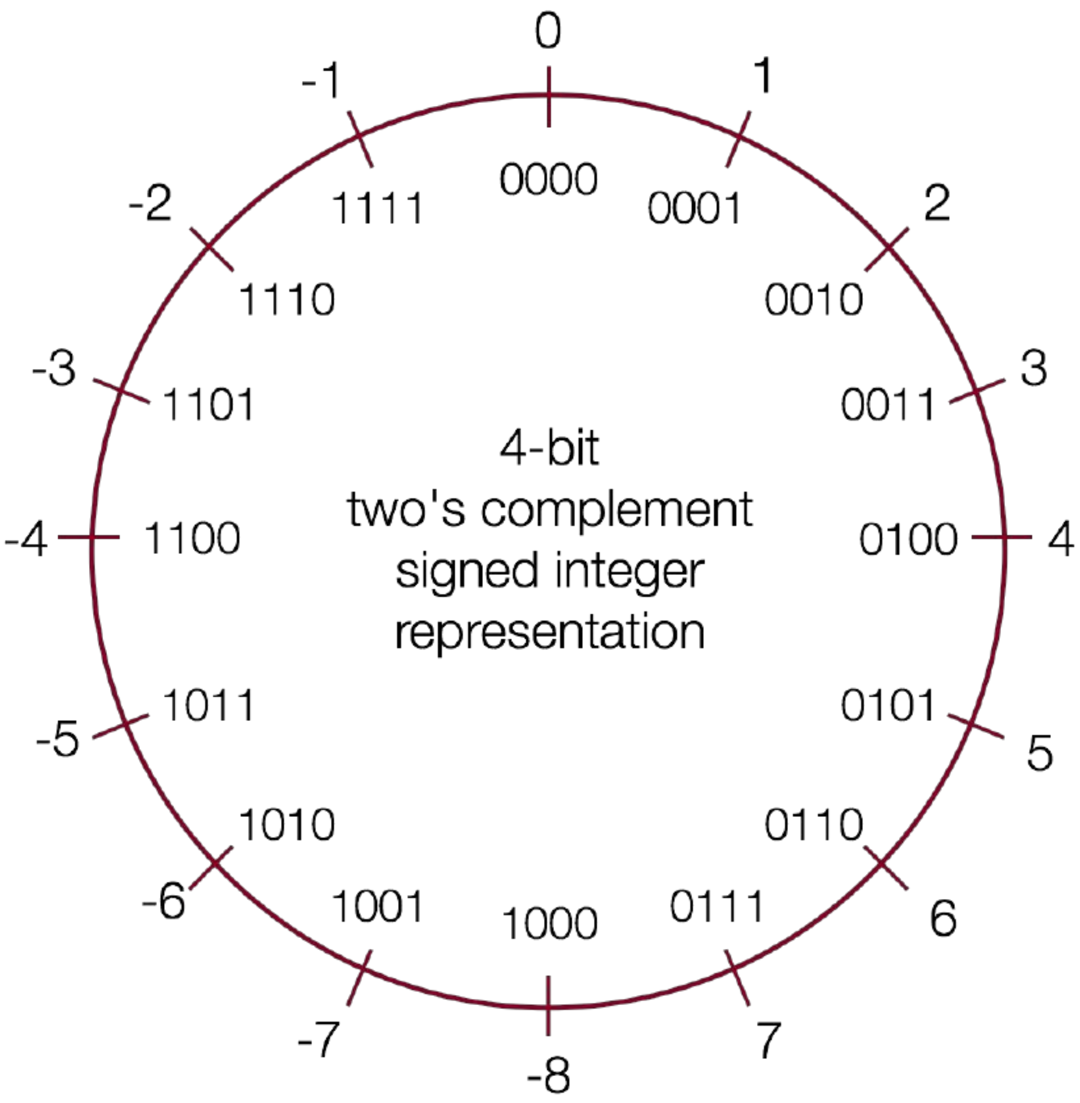
- What happens at the byte level when we cast between variable types? **The bytes remain the same!** **This means they may be interpreted differently depending on the type.**

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

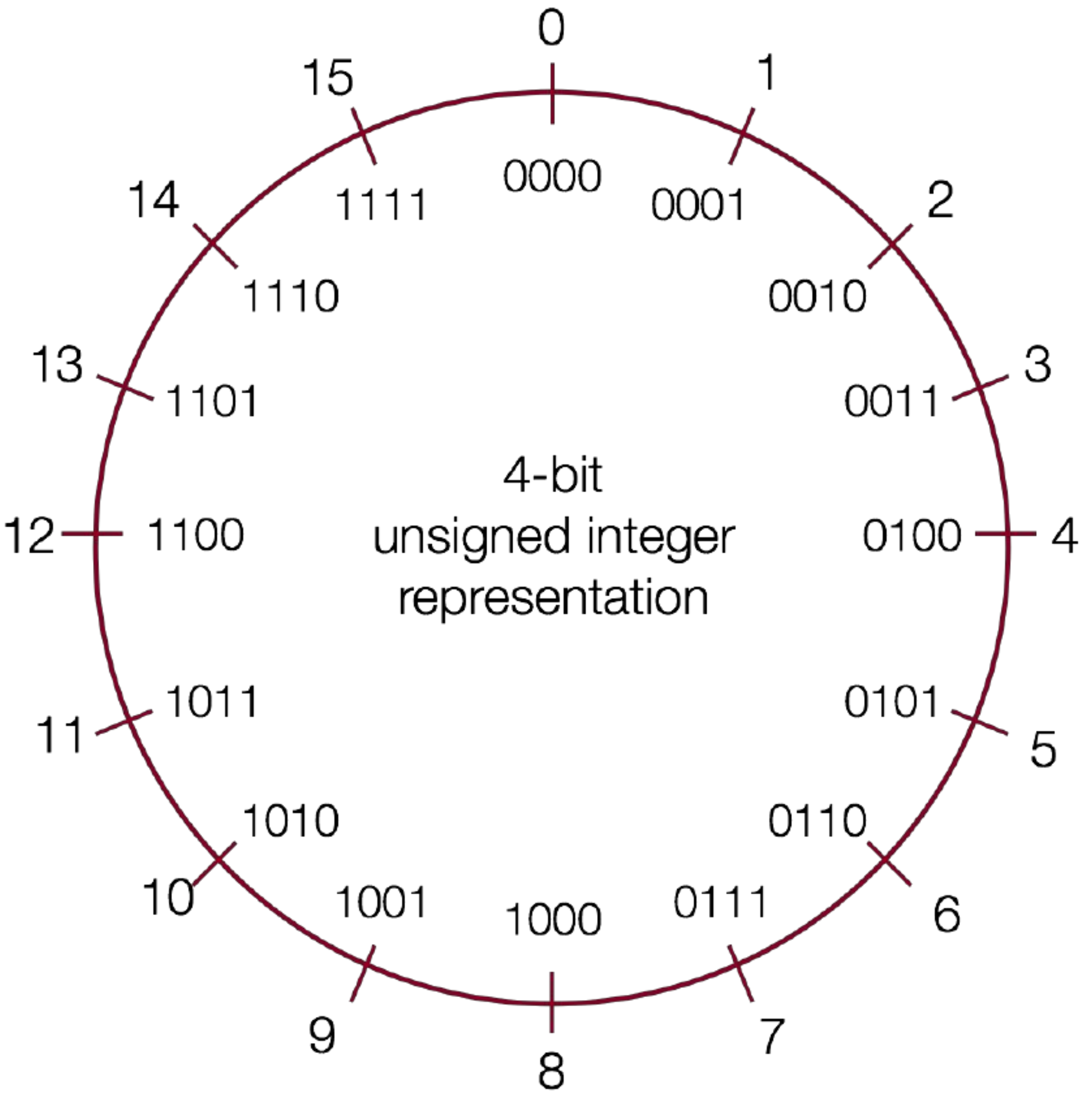
The bit representation for -12345 is `0b11111111111111111100111111000111`.

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

Casting



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
-7
1001
-6
1010
-5
1011
-4
1100
-3
1101
-2
1110
-1
1111

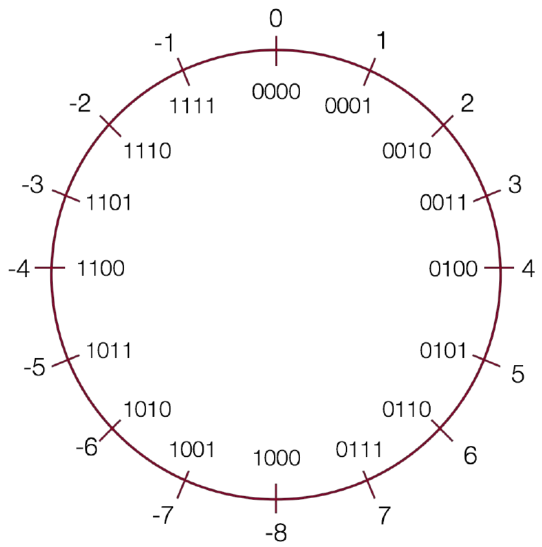


4-bit
unsigned integer
representation
0
0000
1
0001
2
0010
3
0011
4
0100
5
0101
6
0110
7
0111
8
1000
9
1001
10
1010
11
1011
12
1100
13
1101
14
1110
15
1111

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

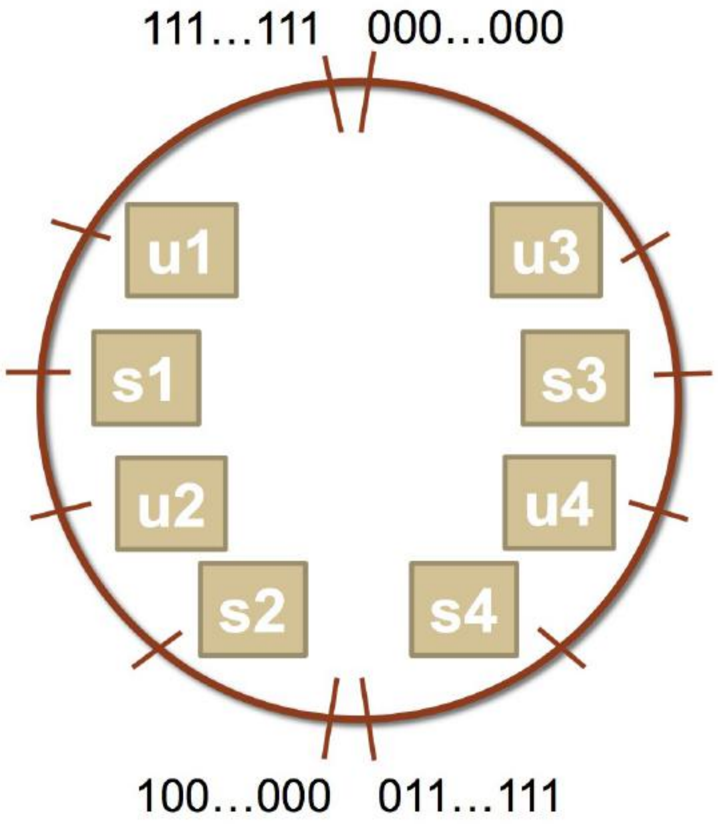


Type	Size (Bytes)	Minimum	Maximum
int	4	-2147483648	2147483647
unsigned int	4	0	4294967295

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)*

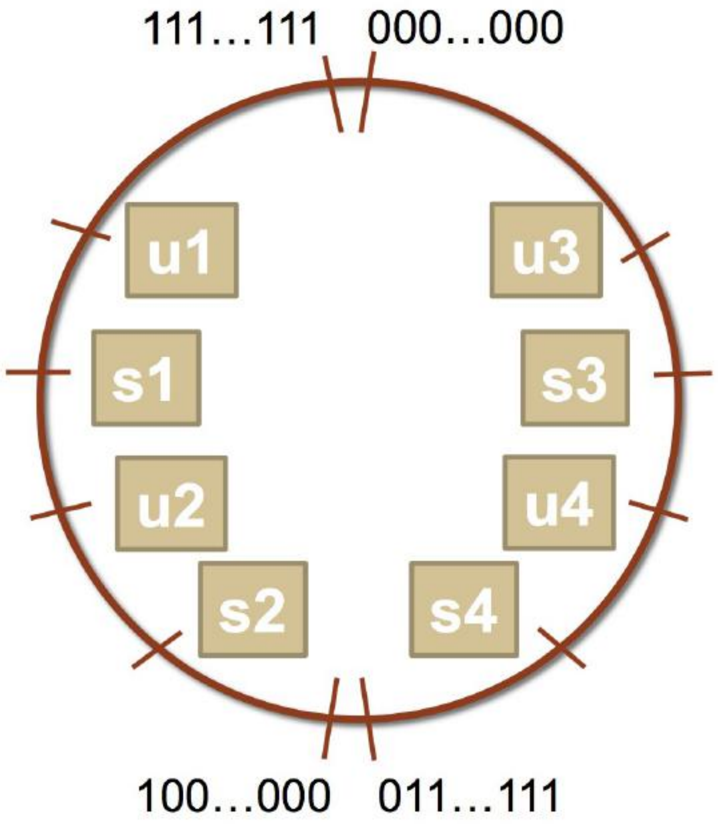
```
s3 > u3
u2 > u4
s2 > s4
s1 > s2
u1 > u2
s1 > u3
```



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)*

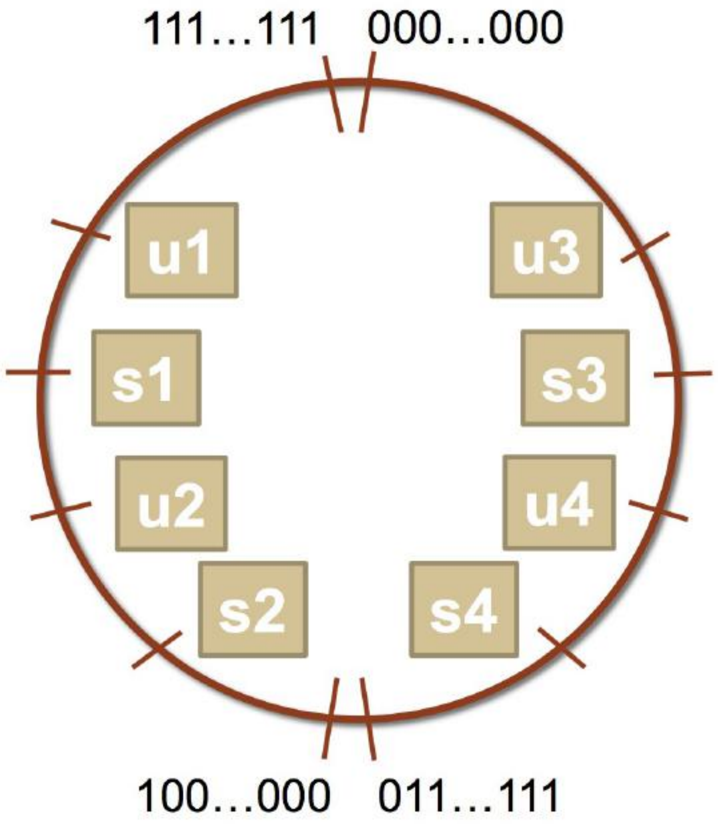
```
s3 > u3 - true
u2 > u4
s2 > s4
s1 > s2
u1 > u2
s1 > u3
```



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)*

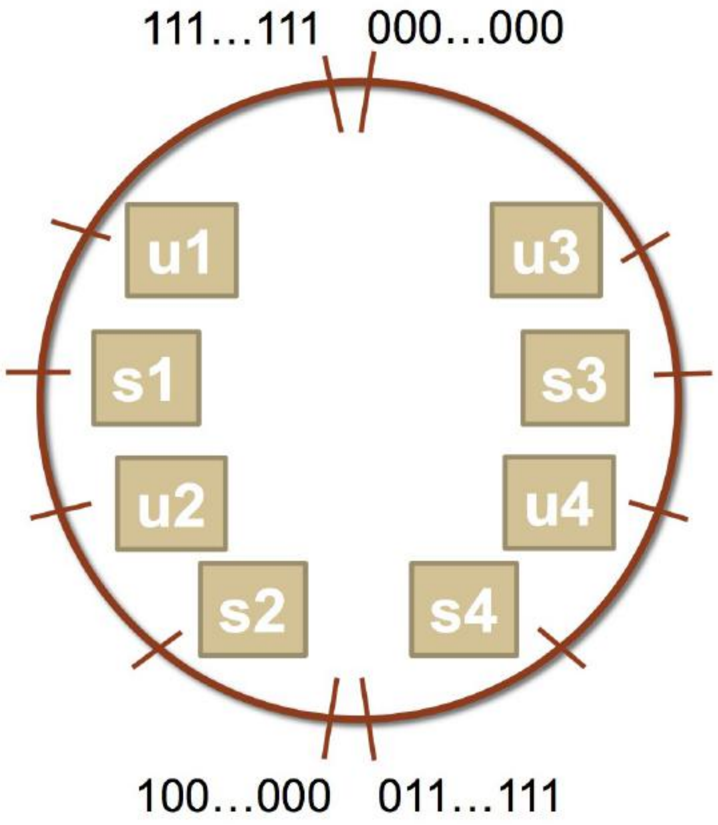
```
s3 > u3 - true
u2 > u4 - true
s2 > s4
s1 > s2
u1 > u2
s1 > u3
```



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)*

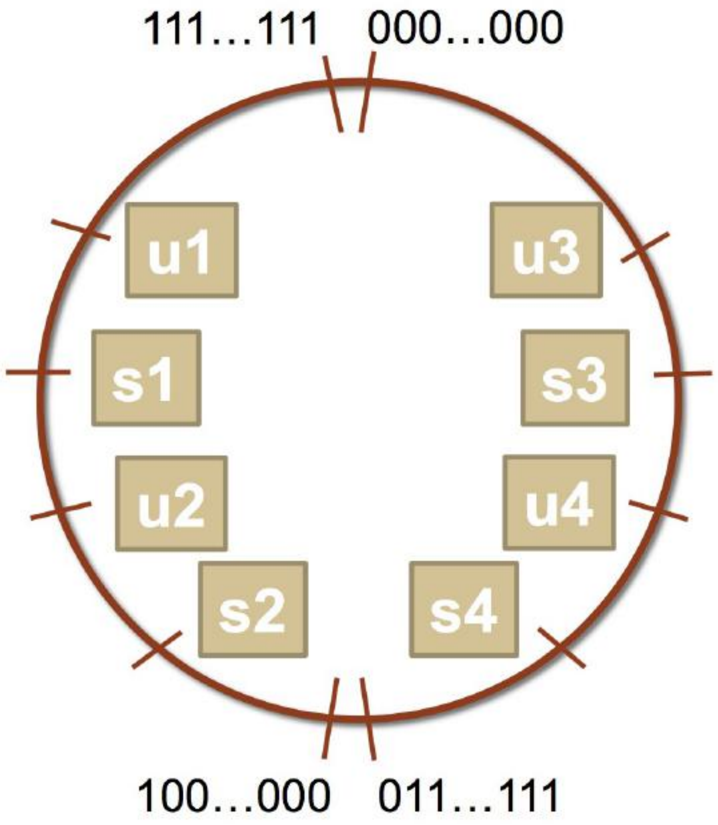
```
s3 > u3 - true
u2 > u4 - true
s2 > s4 - false
s1 > s2
u1 > u2
s1 > u3
```



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)*

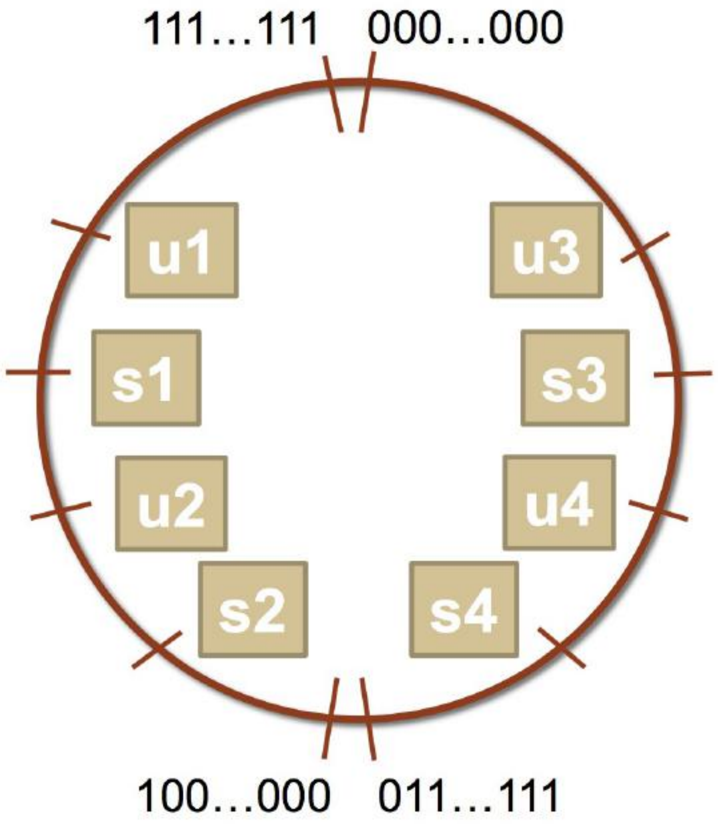
```
s3 > u3 - true
u2 > u4 - true
s2 > s4 - false
s1 > s2 - true
u1 > u2
s1 > u3
```



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)*

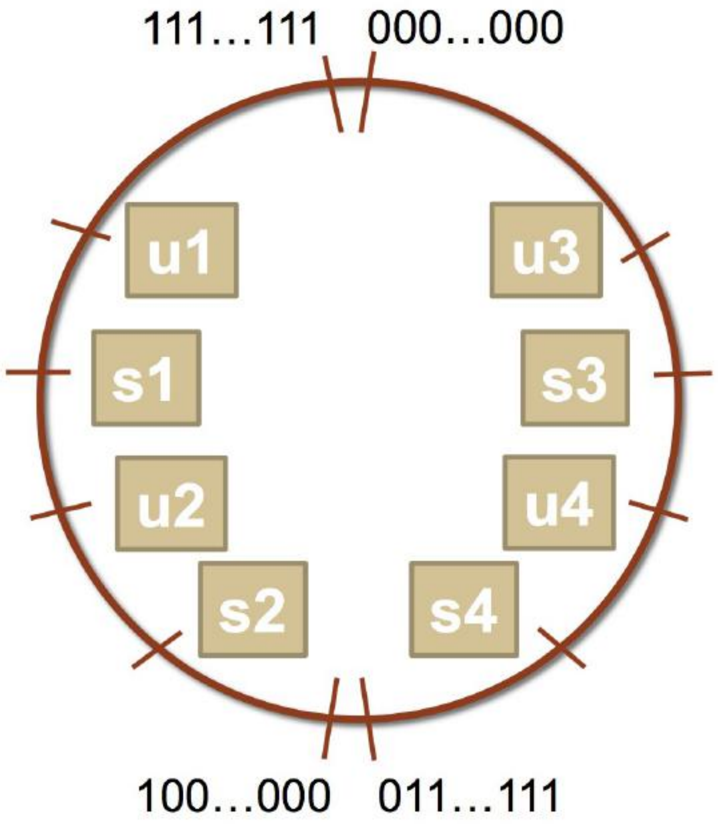
```
s3 > u3 - true
u2 > u4 - true
s2 > s4 - false
s1 > s2 - true
u1 > u2 - true
s1 > u3
```



Comparisons Between Different Types

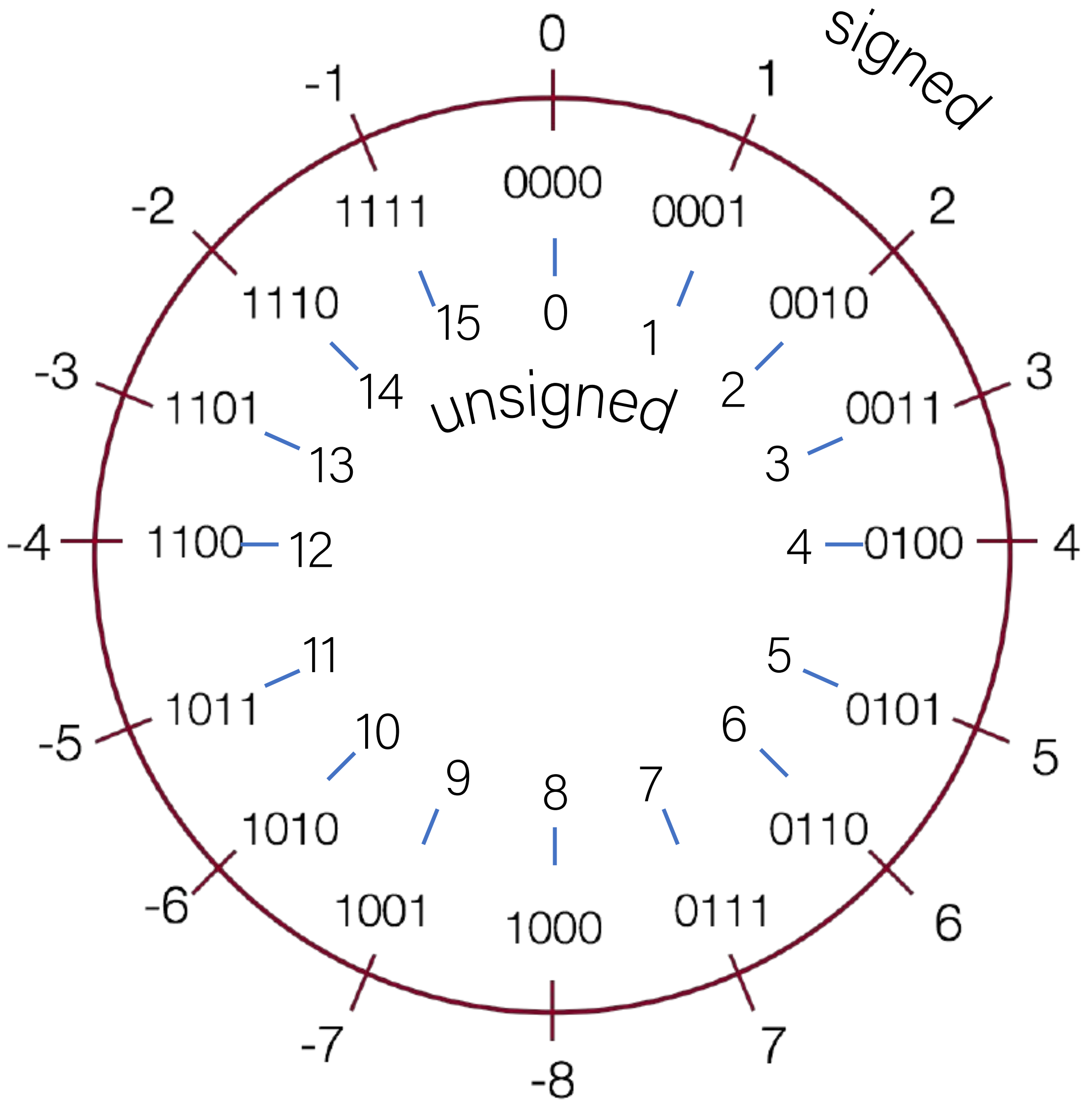
Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*

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



Recap

- Getting Started With C
- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow



Next time: How can we manipulate individual bits and bytes? How can we represent floating point numbers?