# The Hardware/Software Interface

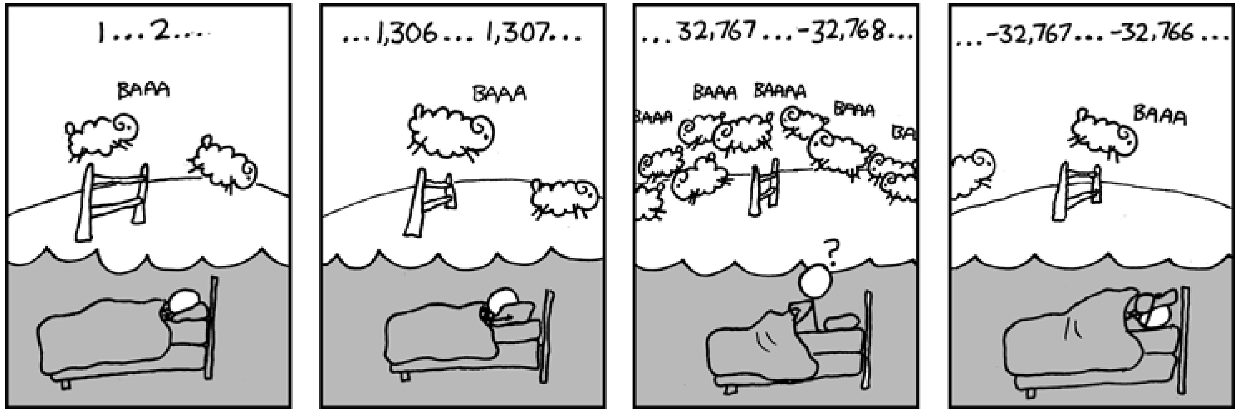
## Integers II

**Instructors:**

Amber Hu, Justin Hsia

**Teaching Assistants:**

Anthony Mangus, Divya Ramu, Grace Zhou, Jessie Sun, Jiuyang Lyu, Kanishka Singh, Kurt Gu, Liander Rainbolt, Mendel Carroll, Ming Yan, Naama Amiel, Pollux Chen, Rose Maresh, Soham Bhosale, Violet Monserate




http://xkcd.com/571/

# Relevant Course Information

- HW3 due tonight, HW4 due Monday, HW5 due Wednesday
- Lab 1a due Monday (10/9)
  - Use `ptest` and `dlc.py` to check your solution for correctness (on the CSE Linux environment)
  - Submit `pointer.c` and `lab1Asynthesis.txt` to Gradescope
    - Make sure you pass the File and Compilation Check – all the correct files were found and there were no compilation or runtime errors
- Lab 1b released today, due 10/16
  - Bit manipulation on a custom encoding scheme
  - Bonus slides at the end of today's lecture have relevant examples
- Reading 6 is dense, do it early if you can!

# Runnable Code Snippets on Ed

- Ed allows you to embed runnable code snippets (*e.g.*, readings, homework, discussion)
  - These are *editable* and *rerunnable*!
  - Hides compiler warnings, but will show compiler errors and runtime errors
  - Code must be inside of an `int main()` function
  - To use `printf()`, you must `#include<stdio.h>`

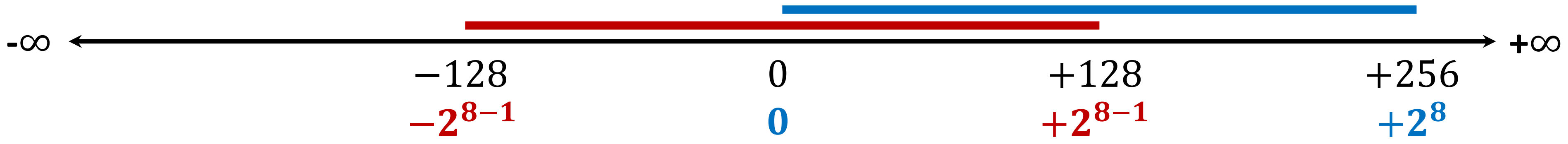
- Suggested use
  - Good for experimental questions about basic behaviors in C
  - *NOT* entirely consistent with the CSE Linux environment, so should not be used for any lab-related work

# Lecture Outline (1/4)

- **Integer Limitations**
- Casting in C
- Bit Shifting
- Integer Representation Issues in Real Life

# Integer Limits for *w* bits (Review)

- Unsigned range
  - UMin = `0b00…0` = $0$
  - UMax = `0b11…1` = $2^w - 1$
- Signed (Two's Complement) values
  - TMin = `0b10…0` = $-2^{w-1}$
  - TMax = `0b01…1` = $2^{w-1} - 1$
- Example: $w$ = 8 (*e.g.*, `char`)



# Integer Arithmetic

- The same addition procedure works for both unsigned and signed (Two's Complement) integers
  - **Simplifies hardware:** Only one algorithm for addition!
  - **Algorithm:** Normal binary addition, discard the highest carry bit
    - Called modular addition: result is sum *modulo* $2^w$

- 4-bit Examples: (HW = hardware, US = unsigned, TC = signed)

| HW | US | TC |
|---|---|---|
| 0100 | | |
| +0011 | | |
| = | | |

| HW | US | TC |
|---|---|---|
| 1100 | | |
| +0011 | | |
| = | | |

| HW | US | TC |
|---|---|---|
| 1101 | | |
| +0100 | | |
| = | | |

# Arithmetic Overflow

| Bits | Unsigned | Signed |
|---|---|---|
| `0000` | 0 | 0 |
| `0001` | 1 | 1 |
| `0010` | 2 | 2 |
| `0011` | 3 | 3 |
| `0100` | 4 | 4 |
| `0101` | 5 | 5 |
| `0110` | 6 | 6 |
| `0111` | 7 | 7 |
| `1000` | 8 | -8 |
| `1001` | 9 | -7 |
| `1010` | 10 | -6 |
| `1011` | 11 | -5 |
| `1100` | 12 | -4 |
| `1101` | 13 | -3 |
| `1110` | 14 | -2 |
| `1111` | 15 | -1 |

- When a calculation produces a result that can’t be represented in the current encoding scheme
  - Integer range limited by fixed width
  - Can occur in both the positive and negative directions
- C and Java ignore overflow exceptions
  - You end up with a bad value in your program and no warning/indication… oops!

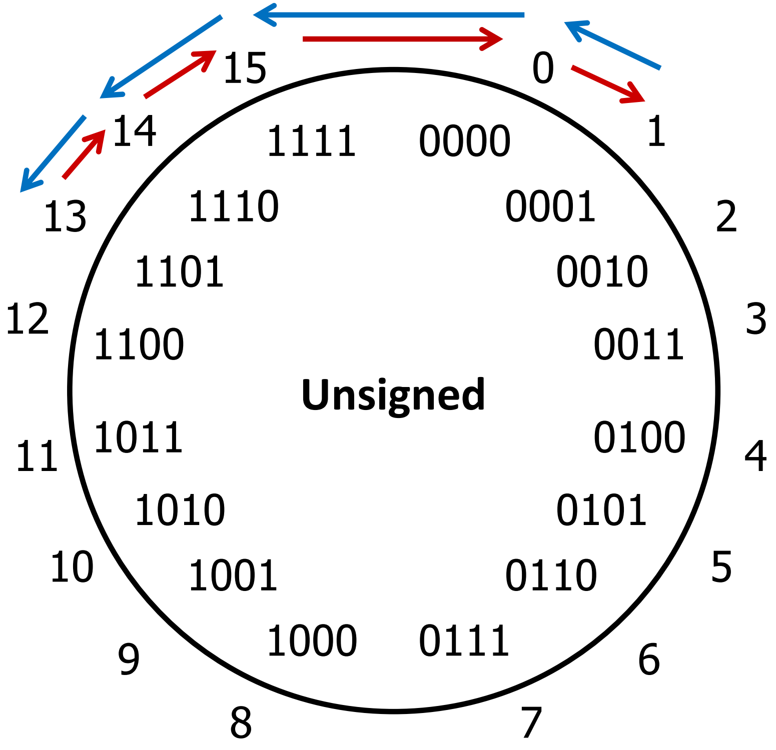
# Overflow: Unsigned

- **Addition:** drop carry bit ($-2^w$)

$$\begin{array}{r} 13 \\ +\ \ 4 \\ \hline \cancel{17} \\ \color{red}{1} \end{array} \qquad \begin{array}{r} \texttt{1101} \\ +\ \ \texttt{0100} \\ \hline \cancel{\texttt{1}}\texttt{0001} \end{array}$$

- **Subtraction:** borrow ($+2^w$)

$$\begin{array}{r} 1 \\ -\ \ 4 \\ \hline \cancel{-3} \\ \color{blue}{13} \end{array} \qquad \begin{array}{r} \texttt{10001} \\ -\ \ \texttt{0100} \\ \hline \texttt{1101} \end{array}$$



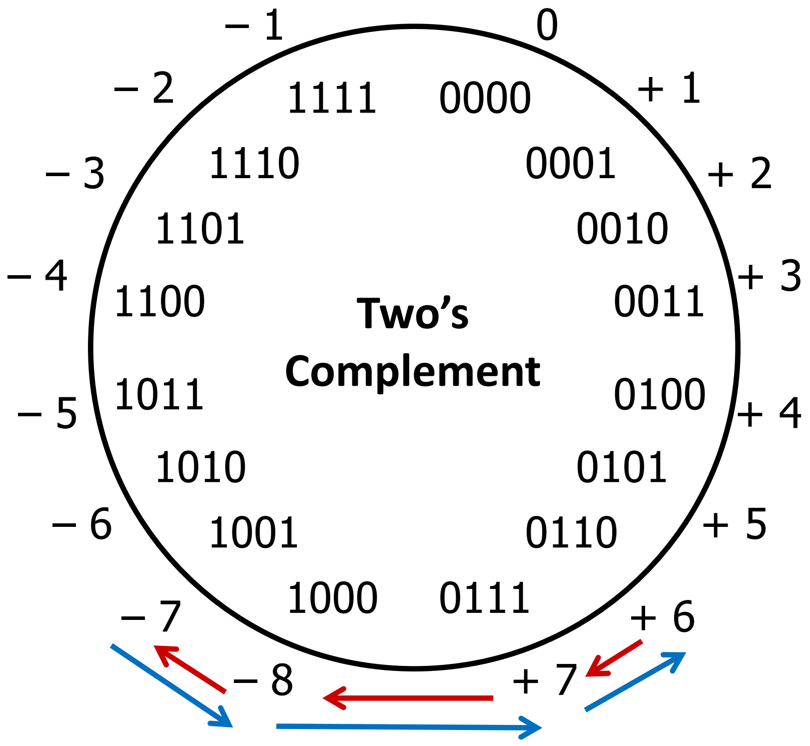
# Overflow: Two's Complement

- **Addition:** (+) + (+) = (−) result?

| | |
|---|---|
| 6 | 0110 |
| + 3 | + 0011 |
| ~~9~~ | 1001 |
| −7 | |

- **Subtraction:** (−) + (−) = (+)?

| | |
|---|---|
| −7 | 1001 |
| − 3 | − 0011 |
| ~~−10~~ | 0110 |
| 6 | |



# Arithmetic Overflow Summary

- Error is always a multiple of $\pm 2^w$ because of modular arithmetic
  - *Unsigned overflow* occurs if result falls outside of [UMin, UMax]
    - There is a carryout from the MSB
  - *Signed overflow* occurs if result falls outside of [TMin, TMax]
    - Signs of both inputs to addition are the same, but the sign of the output is different

- *Independent* properties of the arithmetic operation
  - All four combinations of signed OF and unsigned OF are possible!

| HW | US | TC |
|---|---|---|
| 1101 | 13 | -3 |
| +0100 | + 4 | + 4 |
| =~~1~~0001 | = 1 | = 1 |

✓ unsigned overflow
✗ signed overflow

# Polling Questions (1/2)

- What is the value (and encoding) of **`TMin`** for a fictional 6-bit wide integer data type?

- For the following 8-bit integer additions, did signed and/or unsigned overflow occur?
  - [TMin, TMax] = [-128, 127]
  - [UMin, UMax] = [0, 255]

a) **`0x27 + 0x81`**

b) **`0x7F + 0xD9`**

| Numeral | Signed | Unsigned |
|---|---|---|
| `0x27` | 39 | 39 |
| `0xD9` | -39 | 217 |
| `0x7F` | 127 | 127 |
| `0x81` | -127 | 129 |

# Lecture Outline (2/4)

- Integer Limitations
- **Casting in C**
- Bit Shifting
- Integer Representation Issues in Real Life

# Data Types

- How does a data type affect a variable?
  - Size of allocation (*e.g.*, `char` vs. `long`)
  - How to interpret the bits (*e.g.*, `int` vs. `unsigned`)
  - Valid operators/operations and their behavior (*e.g.*, can't use subscript notation `[]` on integral types, normal vs. pointer arithmetic)

- What does it mean or what are the consequences of *changing* your data type?

# Literals

- Constants/literals in your code also have "types"
  - Affect the stored/manipulated data and the behavior of operators
  - In C:
    - By default, literals (decimal or hex) are treated as *signed integers*
    - Use "U" (or "u") suffix to explicitly force *unsigned* (*e.g.*, 100U, 4294967259u)
    - Integer literals generally have an assumed size of 4 bytes unless longer is needed
    - We will learn about floating point literals next lesson

- Can be confusing if types don't match
  - Example: `signed char c = 255u; printf("%d", c);`
  - Example: `int* ip = 0x40210 + 1;`

# Type Casting: Implicit (Review)

- ***Casting*** converts data of one data type into a different data type
  - Different programming languages may not allow casting or only in certain cases
- C is known for having very flexible casts, with different effects:
  - Changes in bit width (*e.g.*, `short` to `int`)
  - Changes in interpretations (*e.g.*, `int` to `unsigned int`, `long int` to `char*`)
  - Full changes in representations (*e.g.*, `int` to `float`)
- An ***implicit cast*** is done automatically by the compiler to fix type mismatches
  - Needs to be a well-defined conversion between the two types
  - Examples: `int int_var = short_var;`, `printf("%c", short_var);`

# Type Casting: Explicit (Review)

- ***Casting*** converts data of one data type into a different data type
  - Different programming languages may not allow casting or only in certain cases
- C is known for having very flexible casts, with different effects:
  - Changes in bit width (*e.g.*, `short` to `int`)
  - Changes in interpretations (*e.g.*, `int` to `unsigned int`, `long int` to `char*`)
  - Full changes in representations (*e.g.*, `int` to `float`)
- An ***explicit cast*** can be performed by the programmer by using the syntax: `(data_type)expression`
  - Suppress compiler warnings for implicit casts
  - Forcibly cause changes in interpretation or representation

# Casting: Bit Width Change (Mostly Review)

- Longer to shorter
  - *e.g.*, `long` → `int` → `short` → `char`
  - ***Truncation*** (*i.e.*, drop upper bytes)

```
short s = 0x0351;
char c = s;
```

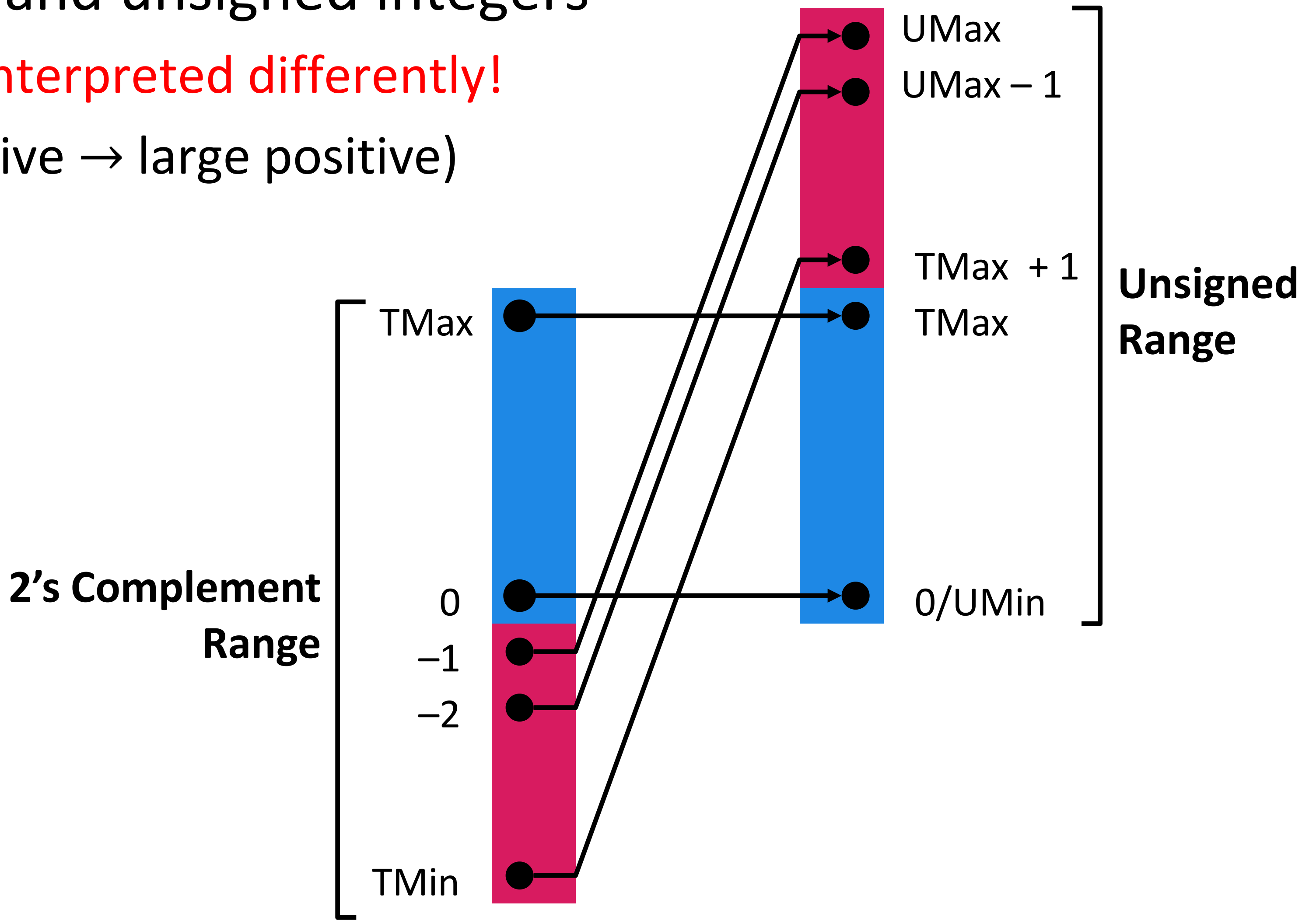
- Shorter to longer
  - *e.g.*, `char` → `short` → `int` → `long`
  - ***Zero extension***: Add all zeros
    - In C, done for unsigned data
  - ***Sign extension***: Add all {old sign bit/MSB}
    - In C, done for signed data to preserve value

```
unsigned char uc = 0xFF;
unsigned short us = uc;

signed char sc = 0xFF;
short ss = sc;
sc = 0x10; ss = sc;
```

# Casting: Interpretation Change

- Casting between signed and unsigned integers
  - Bits are unchanged, just interpreted differently!
  - Ordering Inversion (negative → large positive)



# Data Types: Operator Behavior (Review)

- Expression Evaluation
  - When you mixed unsigned and signed in a single expression, then **signed values are implicitly cast to <u>unsigned</u>**
  - Including comparison operators <, >, ==, <=, >=

- <u>Examples</u>: For 8-bit data, what will the following expressions evaluate to?
  - `127 < 128u`
  - `127 < (signed char) 128u`

# Lecture Outline (3/4)

- Integer Limitations
- Casting in C
- **Bit Shifting**
- Integer Representation Issues in Real Life

# Shift Operations (Review, 1/2)

- Throw away (drop) extra bits that "fall off" the end

- Left shift (`x<<n`) bit vector `x` by `n` positions
  - Fill with `0`'s on right
- Right shift (`x>>n`) bit-vector `x` by `n` positions
  - Logical shift (for unsigned values)
    - Fill with `0`'s on left
  - Arithmetic shift (for signed values)
    - Replicate most significant bit on left (maintains sign of `x`)

# Shift Operations (Review, 2/2)

- Arithmetic:
  - Left shift (`x<<n`) is equivalent to multiply by $2^n$
  - Right shift (`x>>n`) is equivalent to divide by $2^n$
  - Shifting is faster than general multiply and divide operations!

- Notes:
  - Shifts by `n<0` or `n`≥`w` (`w` is bit width of `x`) are *undefined*
  - **In C:** behavior of `>>` is determined by the compiler
    - In gcc / C lang, depends on data type of `x` (signed/unsigned)
  - **In Java:** logical shift is `>>>` and arithmetic shift is `>>`

# Left Shifting 8-bit Example

- No difference in left shift operation for unsigned and signed numbers (just manipulates bits)
  - Difference comes during interpretation: x*2^n?

| | | | Signed | Unsigned |
|---|---|---|---|---|
| `x = 25;` | 00011001 | = | 25 | 25 |
| `L1 = x<<2;` | 0001100100 | = | 100 | 100 |
| `L2 = x<<3;` | 00011001000 | = | -56 | 200 |
| `L3 = x<<4;` | 000110010000 | = | -112 | 144 |

signed overflow

unsigned overflow

# Logical Right Shifting 8-bit Example

- **Reminder:** C operator >> does *logical* shift on unsigned values and *arithmetic* shift on signed values
  - Logical Shift: $x/2^n$?

```
 xu = 240u;   11110000      = 240
R1u = xu>>3; 00011110000    =  30
R2u = xu>>5; 0000011110000  =   7
```

rounding (down)

# Arithmetic Right Shifting 8-bit Negative Example

- **Reminder:** C operator >> does *logical* shift on unsigned values and *arithmetic* shift on signed values
  - Arithmetic Shift: x / $2^n$?

```
 xs = -16;     11110000          = -16
R1s = xs>>3; 11111110000         =  -2
R2s = xs>>5; 1111111110000       =  -1
```

rounding (down)

# Arithmetic Right Shifting 8-bit Positive Example

- **Reminder:** C operator >> does *logical* shift on unsigned values and *arithmetic* shift on signed values
  - Arithmetic Shift: $x/2^n$?

```
 xs = 112;    01110000       = 112
R3s = xs>>3;  00001110000    =  14
R4s = xs>>5;  0000001110000  =   3
```

rounding (down)

# Polling Questions (2/2)

- For `unsigned char uc = 0xA1;`, what are the produced data for the cast **`(unsigned short)uc`**?

- What is the result of the following expressions?
  - **`(signed char)uc >> 2`**
  - **`(unsigned char)uc >> 3`**

# Lecture Outline (4/4)

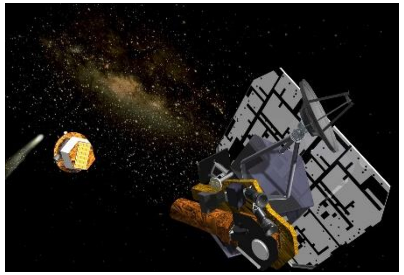
- Integer Limitations
- Casting in C
- Bit Shifting
- **Integer Representation Issues in Real Life**

# Integer Representation Issues in Real Life

- **1985**: Therac-25 radiation therapy machine
  - Overdoses of radiation due to arithmetic overflow of incrementing a 1-byte safety flag variable

- **2000**: Y2K problem
  - Limited representation (two-digit decimal year)


- **2013**: Deep Impact spacecraft lost
  - Suspected integer overflow from storing time as tenth-seconds in unsigned int: 8/11/2013, 00:38:49.6



- **2038**: Unix epoch time rollover (seconds since 1/1/1970)
  - Signed 32-bit integer representation rolls over to TMin in 2038

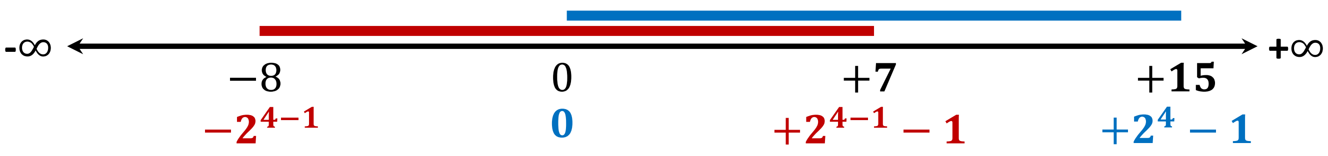
# Discussion Question

- Discuss the following question(s) in groups of 3-4 students
  - I will call on a few groups afterwards so please be prepared to share out
  - Be respectful of others' opinions and experiences

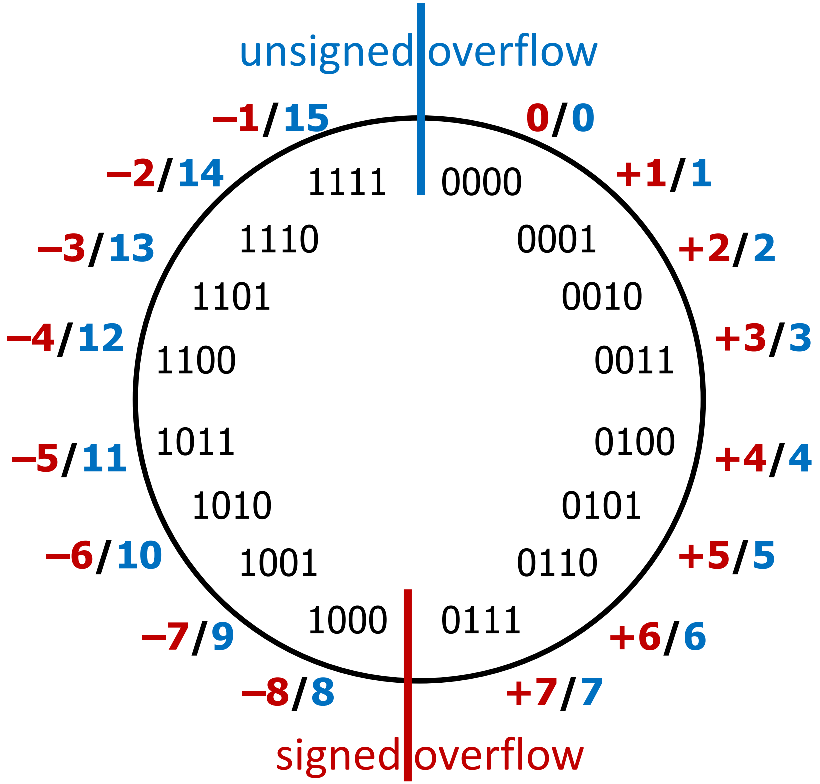
- Given that **arithmetic overflow** is a well-known property of integers in computing, what do you think are some of the *causes* and *pressures* that perpetuate these issues?
  - Think broadly! Ideas could be technical, economic, societal, etc.

# Summary (1/3)

- We can only represent a limited range of numbers in $w$ bits ($2^w$ things)
  - Unsigned: [UMin, UMax]
  - Signed: [TMin, TMax]




- Integer arithmetic is the same in hardware regardless of interpretation
  - When we exceed the limits, *arithmetic overflow* occurs following the rules of modular arithmetic
    - Signed vs. unsigned overflow depends on interpretation of numbers:



# Summary (2/3)

- Data types determine size, interpretations, and operator behaviors
- Casting (implicit or explicit) can convert values between different data types
  - Be careful of the possible consequences of casting (truncation, zero/sign extension, change in interpreted value, change in operator behaviors like comparisons and shifting)

```
int i = -1;
long c = i;             // changed size (sign extension)
unsigned int ui = i;    // changed interpretation

// i < 1 (True) is different than ui < 1 (False)
```

# Summary (3/3)

- Shifting is a useful bitwise operator
  - Throw away (drop) extra bits that "fall off" the end
  - Left shifting always fills with 0's
  - Right shifting can be arithmetic (fill with copies of sign bit) or logical (fill with 0's)
  - Shifts by `n<0` or `n≥w` (`w` is bit width) are *undefined*

- Common use cases: constant multiplication, bit masking
  - `x = x << 3;            // equivalent to 8*x`
  - `x = (x >> 8) << 8;     // zeros out lowest byte of x`

# BONUS SLIDES

Some examples of using shift operators in combination with bitmasks, which you may find helpful for Lab 1b.

- Extract the 2nd most significant byte of an `int`
- Extract the sign bit of a signed `int`
- Conditionals as Boolean expressions

# Using Shifts and Masks

- Extract the 2nd most significant *byte* of an `int`:
  - First shift, then mask: `(x>>16) & 0xFF`

| | |
|---|---|
| **x** | 00000001 00000010 00000011 00000100 |
| **x>>16** | 00000000 00000000 00000001 00000010 |
| **0xFF** | 00000000 00000000 00000000 11111111 |
| **(x>>16) & 0xFF** | 00000000 00000000 00000000 00000010 |

  - Or first mask, then shift: `(x & 0xFF0000)>>16`

| | |
|---|---|
| **x** | 00000001 00000010 00000011 00000100 |
| **0xFF0000** | 00000000 11111111 00000000 00000000 |
| **x & 0xFF0000** | 00000000 00000010 00000000 00000000 |
| **(x&0xFF0000)>>16** | 00000000 00000000 00000000 00000010 |

# Using Shifts and Masks

- Extract the sign bit of a signed int:
  - First shift, then mask: (x>>31) & 0x1
    - Assuming arithmetic shift here, but this works in either case
    - Need mask to clear 1s possibly shifted in

| | |
|---|---|
| `x` | 00000001 00000010 00000011 00000100 |
| `x>>31` | 00000000 00000000 00000000 00000000 |
| `0x1` | 00000000 00000000 00000000 00000001 |
| `(x>>31) & 0x1` | 00000000 00000000 00000000 00000000 |

| | |
|---|---|
| `x` | 10000001 00000010 00000011 00000100 |
| `x>>31` | 11111111 11111111 11111111 11111111 |
| `0x1` | 00000000 00000000 00000000 00000001 |
| `(x>>31) & 0x1` | 00000000 00000000 00000000 00000001 |

# Using Shifts and Masks

- Conditionals as Boolean expressions
  - For **`int`** `x`, what does `(x<<31)>>31` do?

| | |
|---|---|
| `x=!!123` | `00000000 00000000 00000000 00000001` |
| `x<<31` | `10000000 00000000 00000000 00000000` |
| `(x<<31)>>31` | `11111111 11111111 11111111 11111111` |
| `!x` | `00000000 00000000 00000000 00000000` |
| `!x<<31` | `00000000 00000000 00000000 00000000` |
| `(!x<<31)>>31` | `00000000 00000000 00000000 00000000` |

  - Can use in place of conditional:
    - In C: `if(x) {a=y;} else {a=z;}` equivalent to `a=x?y:z;`
    - `a=(((!!x<<31)>>31)&y) | (((!x<<31)>>31)&z);`