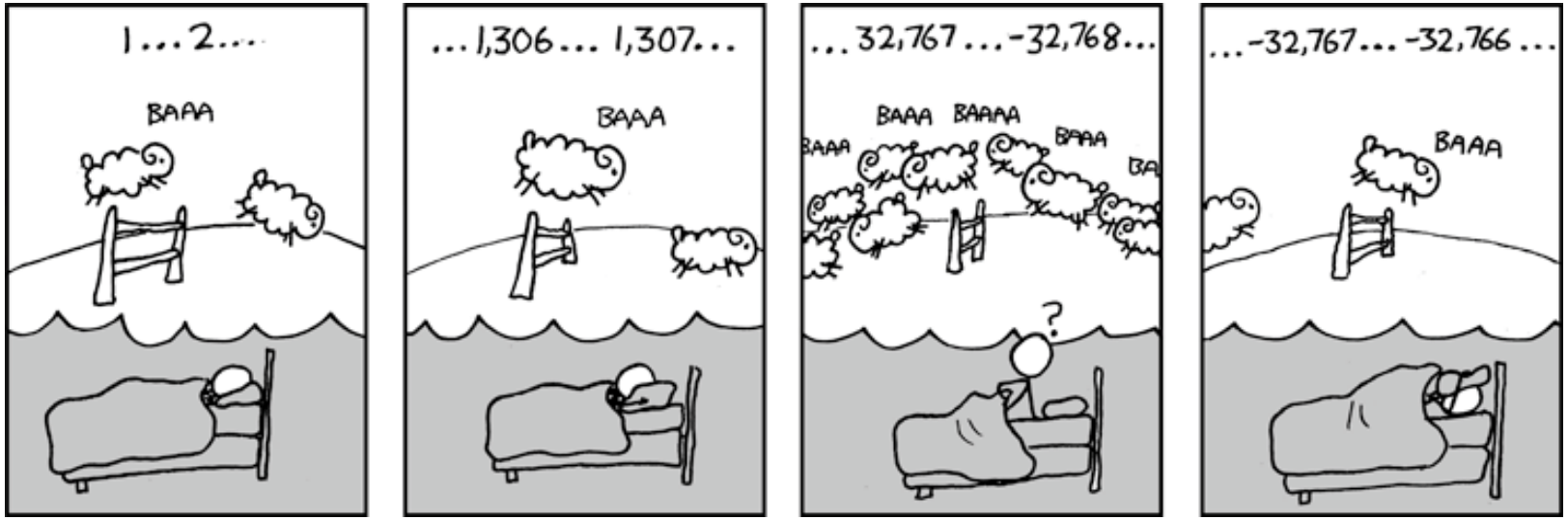
Integers II

CSE 351 Winter 2024

Instructor:
Justin Hsia

Teaching Assistants:
Adithi Raghavan
Aman Mohammed
Connie Chen
Eyoel Gebre
Jiawei Huang
Malak Zaki
Naama Amiel
Nathan Khuat
Nikolas McNamee
Pedro Amarante
Will Robertson



http://xkcd.com/571/

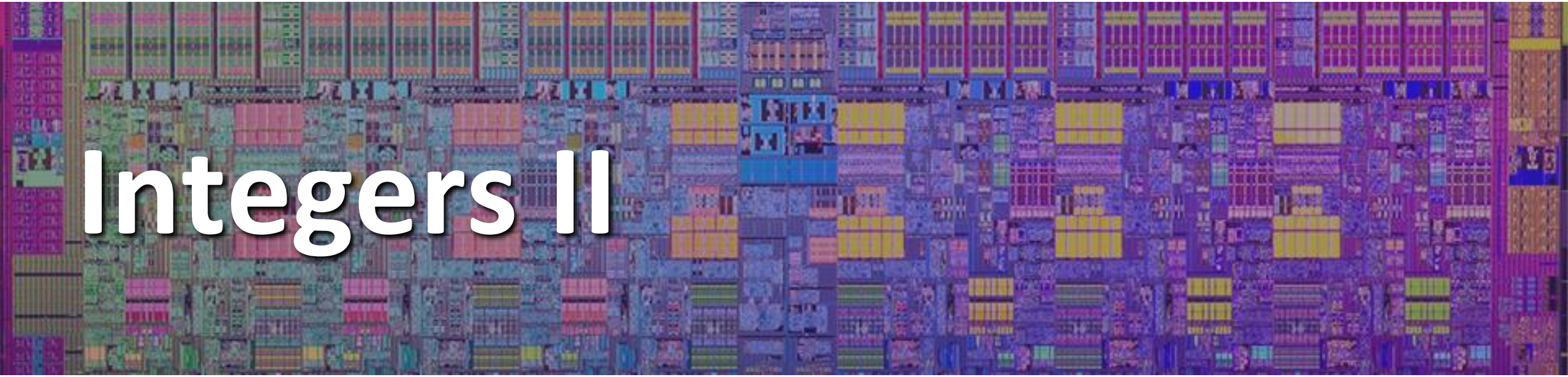
Relevant Course Information

- Monday is MLK Jr. Holiday
  - There will still be some virtual support hours
- HW4 due Wednesday, HW5 due next Friday
- Lab 1a due Monday (1/15)
  - 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 1/22
  - Bit manipulation on a custom encoding scheme

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

- 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

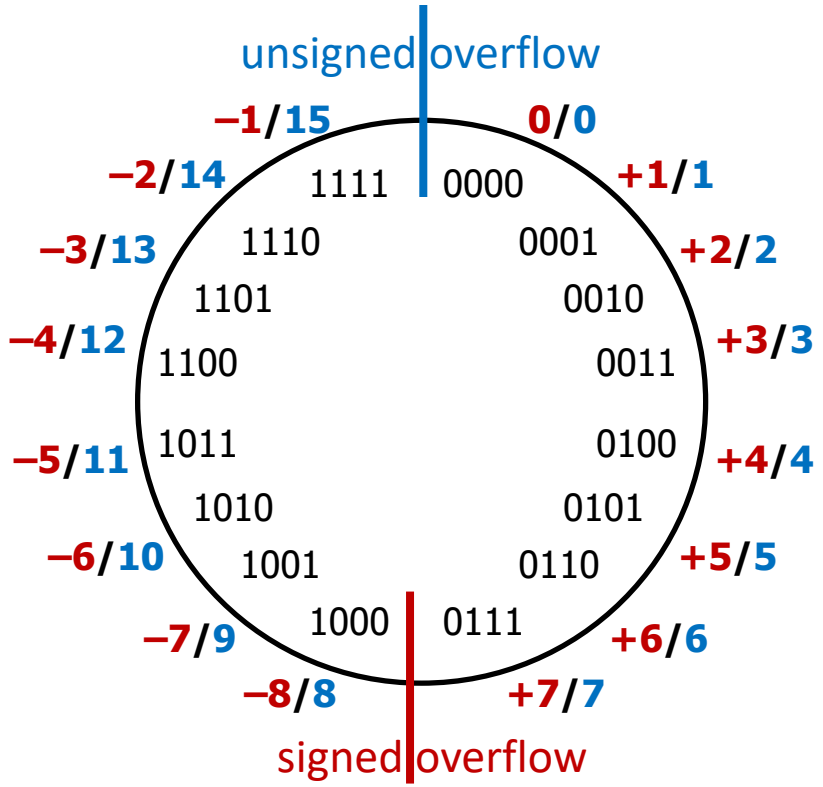


Integers II

Lesson 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:



Lesson 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)
```

Lesson Summary (3/3)

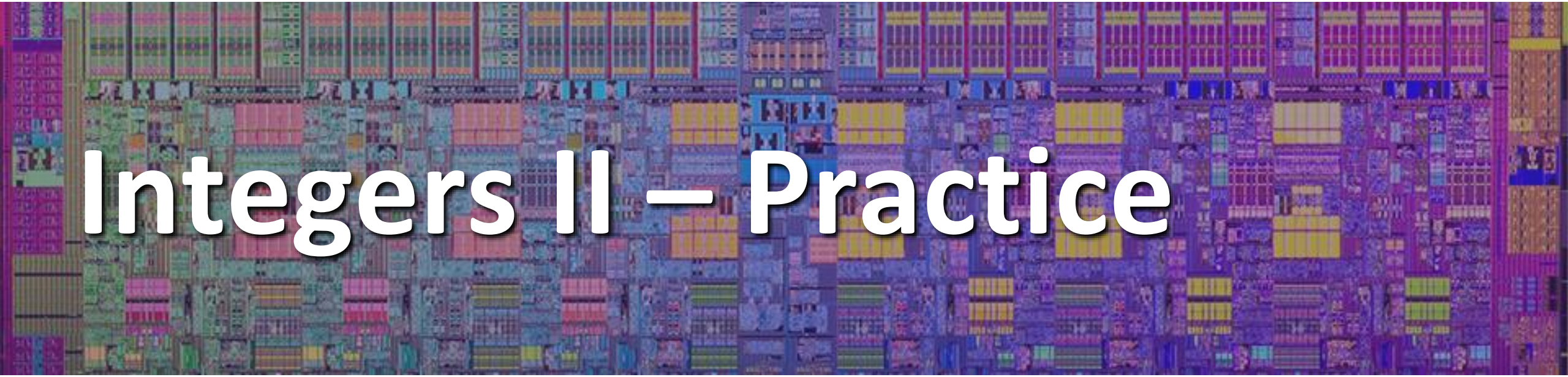
- Shifting is a useful bitwise operator
  - Throw away (drop) extra bits that "fall off" the end
  - Left shifting always fills with 0's
  - Right shifting can be arithmetic (fill with copies of sign bit) or logical (fill with 0's)
  - Shifts by `n<0` or `n≥w` (`w` is bit width) are *undefined*

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

Lesson Q&A

- Learning Objectives:
  - Identify when integer limitations are encountered (*e.g.*, overflow).
  - Identify the effect of C casts (both implicit and explicit) on stored values and the behavior of operations.

- What lingering questions do you have from the lesson?
  - Chat with your neighbors about the lesson for a few minutes to come up with questions



Integers II – Practice

Practice Problems (1/2)

signed

most negative

represent $2^6 = 64$ numbers

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

0b $\underset{-2^5}{1}\ \underset{2^4}{0}\ \underset{2^3}{0}\ \underset{2^2}{0}\ \underset{2^1}{0}\ \underset{2^0}{0}$ $\quad -2^{n-1} = -2^5 = \boxed{-32}$

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

2 bytes

unsigned → zero extension $\quad \boxed{0x00A1}$

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

signed: 0b 1010 0001 —arithmetic→ 0b 1110 1000 = $\boxed{0xE8}$

unsigned: 0b 1010 0001 —logical→ 0b 0001 0100 = $\boxed{0x14}$

Practice Problems (2/2)

- Assuming 8-bit integers:
  - 0x27 = 39 (signed) = 39 (unsigned)
  - 0xD9 = -39 (signed) = 217 (unsigned)
  - 0x7F = 127 (signed) = 127 (unsigned)
  - 0x81 = -127 (signed) = 129 (unsigned)

[TMin, TMax] = [-128, 127]
[UMin, UMax] = [0, 255]

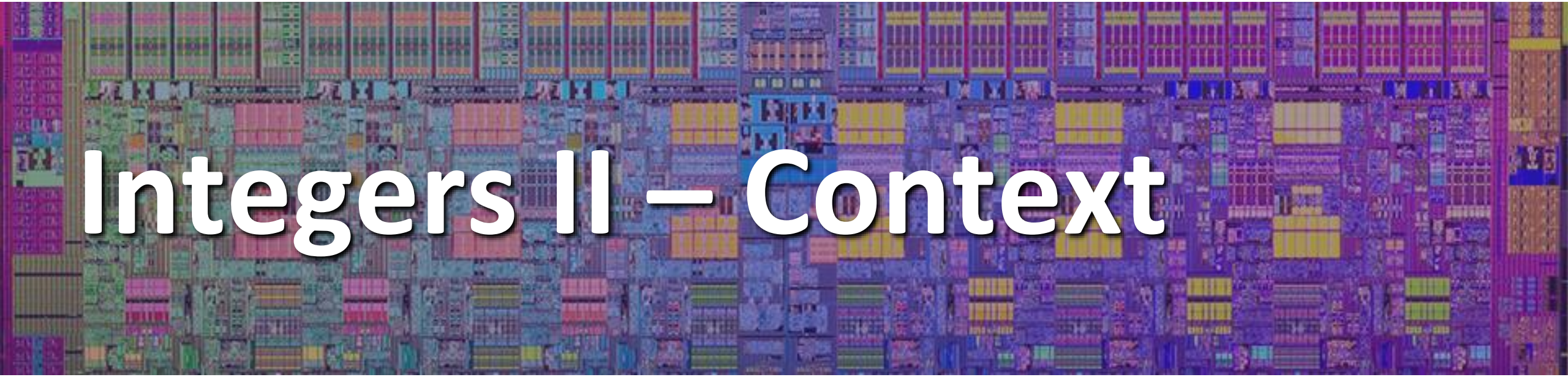
- For the following additions, did signed and/or unsigned overflow occur?
  - **0x27 + 0x81**

    signed: 39 + (-127) = -88
    no signed overflow

    unsigned: 39 + 129 = 168
    no unsigned overflow
  - **0x7F + 0xD9**

    signed: 127 + (-39) = 88
    no signed overflow

    unsigned: 127 + 217 = 344
    unsigned overflow



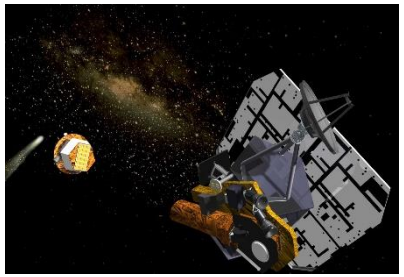
Integers II – Context

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.

Group Work Time

- During this time, you are encouraged to work on the following:
  1) If desired, continue your discussion
  2) Work on the homework problems
  3) Work on the lab (if applicable)

- Resources:
  - You can revisit the lesson material
  - Work together in groups and help each other out
  - Course staff will circle around to provide support