Integers II, Floating Point I

CSE 351 Summer 2020

Instructor:

Porter Jones

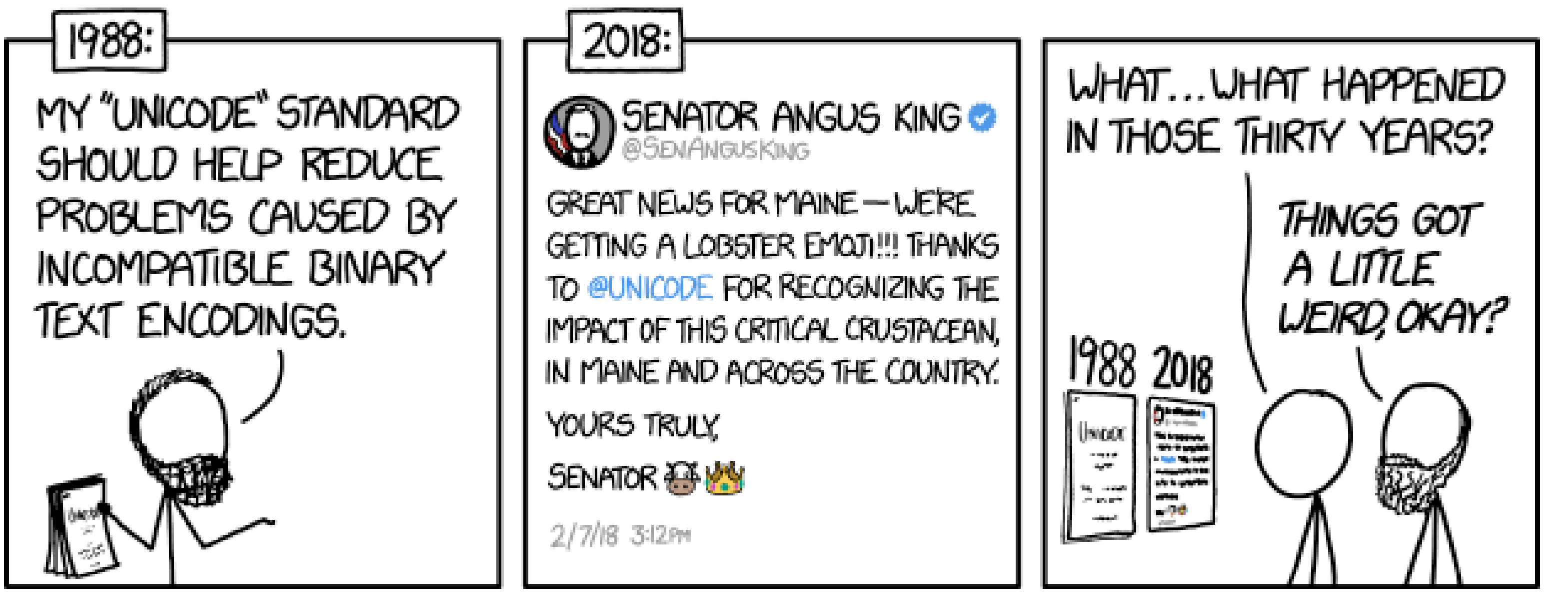
Teaching Assistants:

Amy Xu

Callum Walker

Sam Wolfson

Tim Mandzyuk



http://xkcd.com/1953/

Administrivia

- Questions doc: https://tinyurl.com/CSE351-7-1
- hw4 and hw5 due Monday 7/6 – 10:30am
- hw6 and hw7 due Friday 7/10 – 10:30am
  - Will post Monday's slides later today so you can get started
- Lab 1a due Monday (7/6) **(try to finish by Friday!)**
  - Submit `pointer.c` and `lab1Areflect.txt` to Gradescope
- Lab 1b released tomorrow, due 7/10
  - Bit manipulation problems using custom data type
  - Today's bonus slides have helpful examples, omorrow's section will have helpful examples too

Gradescope Lab Turnin

- Make sure you pass the File and Compilation Check!
- Doesn't indicate if you passed all tests, just indicates that all the correct files were found and there were no compilation or runtime errors.
- Use the testing programs we provide to check your solution for correctness (on attu or the VM)

Quick Aside: C Macros

- Lab1b will have you use some C macros for bit masks
- Syntax is of the form:

```
#define NAME expression
```

- Can now use "NAME" instead of "expression" in code
- Useful to help with readability/factoring in code
  - Especially useful for defining constants such as bit masks!
- Are NOT exactly the same as a constant in Java
  - Does naïve copy and replace *before* compilation.
  - Everywhere the characters "NAME" appear in the code, the characters "expression" will now appear instead.
- See Lecture 4 (Integers I) slides for example usages

Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and **arithmetic operations** – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension

Two's Complement Arithmetic

- The same addition procedure works for both unsigned and two's complement integers
  - **Simplifies hardware:** only one algorithm for addition
  - **Algorithm:** simple addition, discard the highest carry bit
    - Called modular addition: result is sum *modulo* 2^w

- **4-bit Examples:**

HW	TC
0100	
+0011	
=	

HW	TC
1100	
+0011	
=	

HW	TC
0100	
+1101	
=	

Why Does Two's Complement Work?

- For all representable positive integers x, we want:

$$\begin{array}{r} \textit{bit representation of } x \\ + \textit{ bit representation of } -x \\ \hline 0 \end{array}$$ (ignoring the carry-out bit)

  - What are the 8-bit negative encodings for the following?

$$\begin{array}{r} 00000001 \\ +\ \ ???????? \\ \hline 00000000 \end{array} \qquad \begin{array}{r} 00000010 \\ +\ \ ???????? \\ \hline 00000000 \end{array} \qquad \begin{array}{r} 11000011 \\ +\ \ ???????? \\ \hline 00000000 \end{array}$$

Why Does Two's Complement Work?

- For all representable positive integers x, we want:

$$\begin{array}{r} \textit{bit representation of } \; x \\ + \; \textit{bit representation of} -x \\ \hline 0 \end{array}$$ (ignoring the carry-out bit)

  - What are the 8-bit negative encodings for the following?

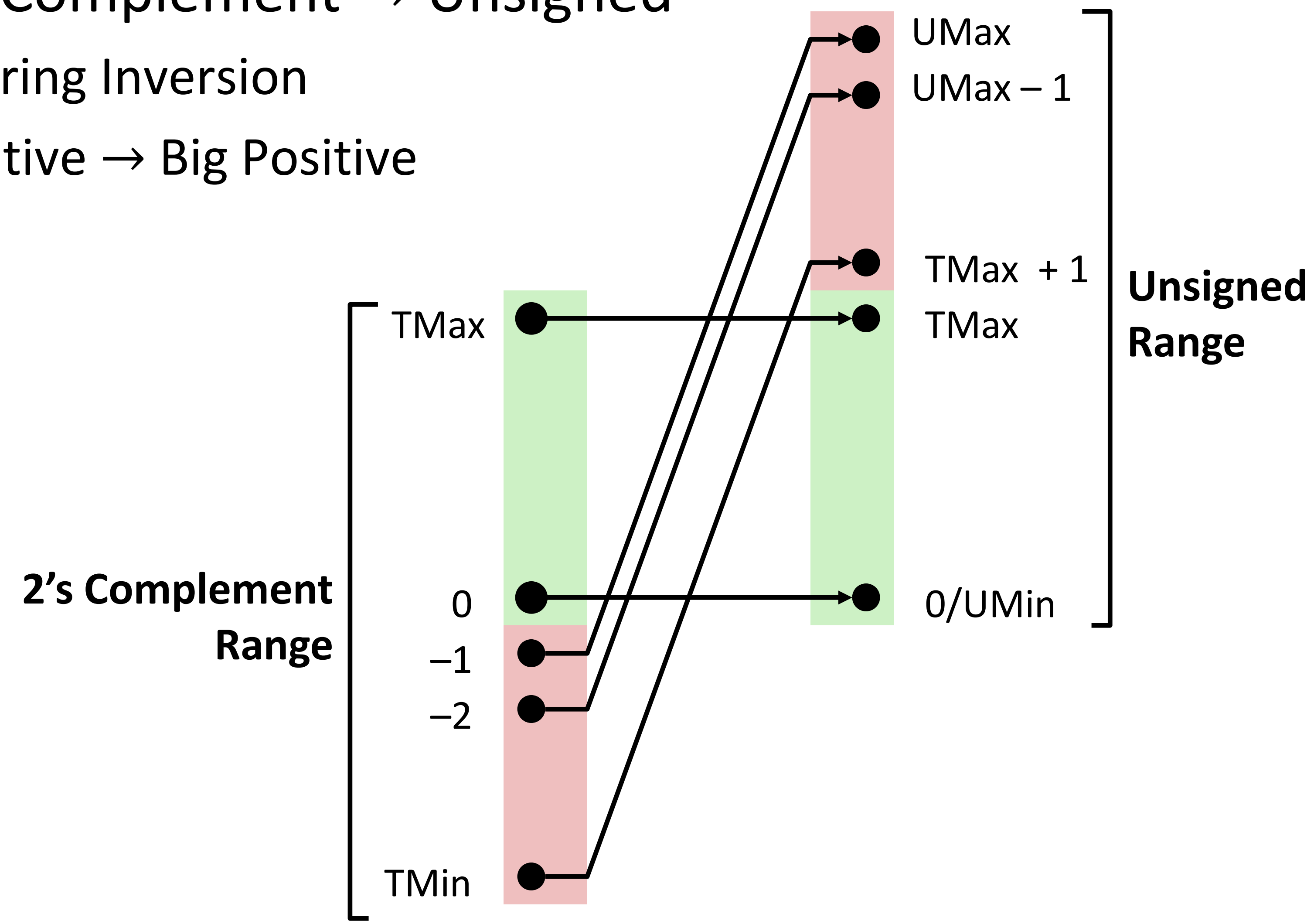
$$\begin{array}{r} 00000001 \\ + \; 11111111 \\ \hline 100000000 \end{array} \qquad \begin{array}{r} 00000010 \\ + \; 11111110 \\ \hline 100000000 \end{array} \qquad \begin{array}{r} 11000011 \\ + \; 00111101 \\ \hline 100000000 \end{array}$$

These are the bitwise complement plus 1!

`-x == ~x + 1`

Signed/Unsigned Conversion Visualized

- Two's Complement → Unsigned
  - Ordering Inversion
  - Negative → Big Positive



Values To Remember

- Unsigned Values
  - UMin = `0b00…0` = 0
  - UMax = `0b11…1` = $2^w - 1$
- Two's Complement Values
  - TMin = `0b10…0` = -2^{w-1}
  - TMax = `0b01…1` = $2^{w-1} - 1$
  - -1 = `0b11…1`
- **Example:** Values for $w = 64$

	Decimal	Hex
UMax	18,446,744,073,709,551,615	FF FF FF FF FF FF FF FF
TMax	9,223,372,036,854,775,807	7F FF FF FF FF FF FF FF
TMin	-9,223,372,036,854,775,808	80 00 00 00 00 00 00 00
-1	-1	FF FF FF FF FF FF FF FF
0	0	00 00 00 00 00 00 00 00

Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – useful for Lab 1a
- **In C: Signed, Unsigned and Casting**
- Consequences of finite width representations
  - Overflow, sign extension

In C: Signed vs. Unsigned

- Casting
  - Bits are unchanged, just interpreted differently!
    - **int** tx, ty;
    - **unsigned int** ux, uy;
  - *Explicit* casting
    - tx = (**int**) ux;
    - uy = (**unsigned int**) ty;
  - *Implicit* casting can occur during assignments or function calls
    - tx = ux;
    - uy = ty;

Casting Surprises

!!!

- Integer literals (constants)
  - By default, integer constants are considered *signed* integers
    - Hex constants already have an explicit binary representation
  - Use “U” (or “u”) suffix to explicitly force *unsigned*
    - **Examples:** 0U, 4294967259u

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

Casting Surprises

!!!

- 32-bit examples:
  - TMin = -2,147,483,648, TMax = 2,147,483,647

Left Constant	Order	Right Constant	Interpretation
0 0000 0000 0000 0000 0000 0000 0000 0000		**0U** 0000 0000 0000 0000 0000 0000 0000 0000	
-1 1111 1111 1111 1111 1111 1111 1111 1111		**0** 0000 0000 0000 0000 0000 0000 0000 0000	
-1 1111 1111 1111 1111 1111 1111 1111 1111		**0U** 0000 0000 0000 0000 0000 0000 0000 0000	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111		**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	
2147483647U 0111 1111 1111 1111 1111 1111 1111 1111		**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	
-1 1111 1111 1111 1111 1111 1111 1111 1111		**-2** 1111 1111 1111 1111 1111 1111 1111 1110	
(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111		**-2** 1111 1111 1111 1111 1111 1111 1111 1110	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111		**2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111		**(int) 2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	

Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- **Consequences of finite width representations**
  - **Overflow, sign extension**

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!

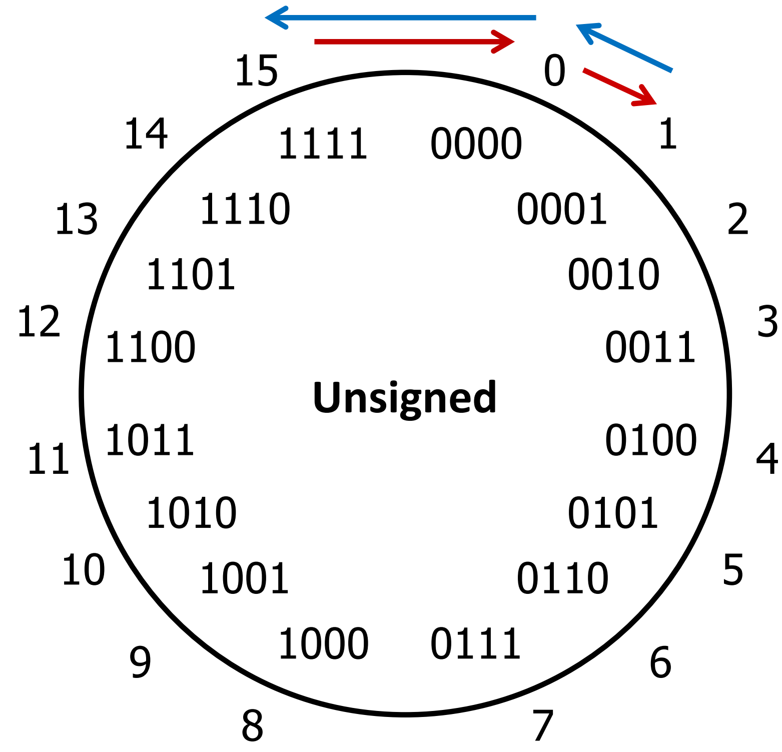
Overflow: Unsigned

- **Addition:** drop carry bit (-2^N)

$$\begin{array}{rr} & 15 \\ + & 2 \\ \hline & \cancel{17} \\ & 1 \end{array} \qquad \begin{array}{rr} & 1111 \\ + & 0010 \\ \hline & \cancel{1}0001 \end{array}$$

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

$$\begin{array}{rr} & 1 \\ - & 2 \\ \hline & \cancel{-1} \\ & 15 \end{array} \qquad \begin{array}{rr} & 10001 \\ - & 0010 \\ \hline & 1111 \end{array}$$



$\pm 2^N$ because of modular arithmetic

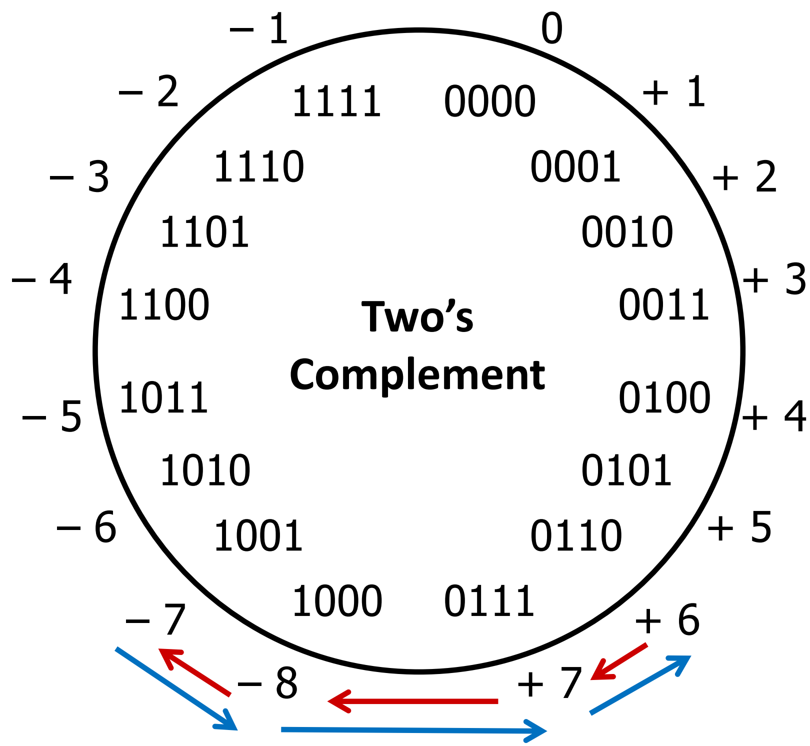
Overflow: Two's Complement

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

$$\begin{array}{r} 6 \\ +\ \ 3 \\ \hline \cancel{9} \\ {\color{red}-7} \end{array} \qquad \begin{array}{r} 0110 \\ +\ \ 0011 \\ \hline 1001 \end{array}$$

- **Subtraction:** (–) + (–) = (+)?

$$\begin{array}{r} -7 \\ -\ \ 3 \\ \hline \cancel{-10} \\ {\color{blue}6} \end{array} \qquad \begin{array}{r} 1001 \\ -\ \ 0011 \\ \hline 0110 \end{array}$$



For signed: overflow if operands have same sign and result's sign is different

Sign Extension

- What happens if you convert a *signed* integral data type to a larger one?
  - *e.g.* `char` → `short` → `int` → `long`
    (1 byte, 2 bytes, 4 bytes, 8 bytes)
- **4-bit → 8-bit Example:**
  - Positive Case
    - ✓ Add 0's?
  - Negative Case?

4-bit:	0010	=	+2
8-bit:	00000010	=	+2

Polling Question [Int II - a]

- Which of the following 8-bit numbers has the same *signed* value as the 4-bit number **0b1100**?
  - Underlined digit = MSB
  - Vote at http://pollev.com/pbjones

A. **0b 0000 1100**

B. **0b 1000 1100**

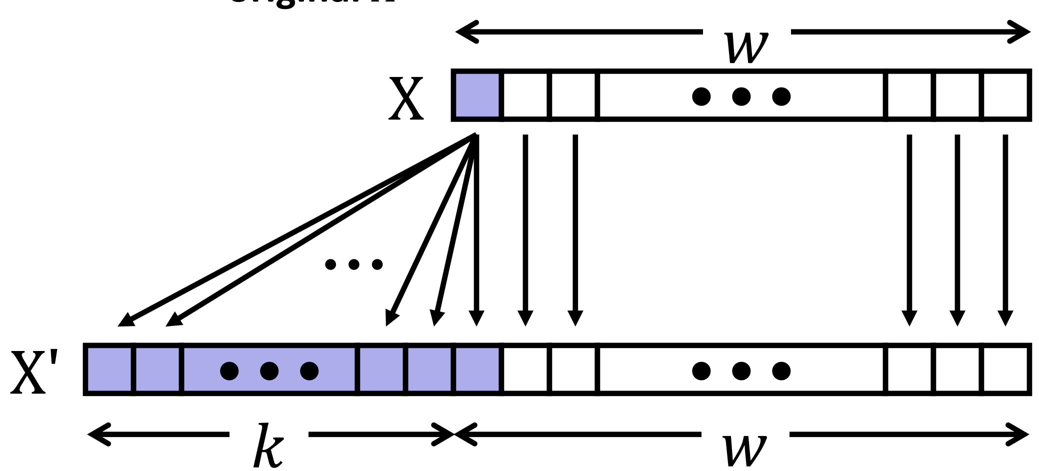
C. **0b 1111 1100**

D. **0b 1100 1100**

E. **We're lost…**

Sign Extension

- **Task:** Given a w-bit signed integer X, convert it to w+k-bit signed integer X' *with the same value*
- **Rule: Add k copies of sign bit**
  - Let x_i be the i-th digit of X in binary
  - $\mathrm{X}' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, \underbrace{x_{w-1}, x_{w-2}, \ldots, x_1, x_0}_{\text{original } \mathrm{X}}$



Sign Extension Example

- Convert from smaller to larger integral data types
- C automatically performs sign extension
  - Java too

```
short int x =   12345;
int      ix = (int) x;
short int y =  -12345;
int      iy = (int) y;
```

0b 0011

Var	Decimal	Hex	Binary
x	12345	30 39	00110000 00111001
ix	12345	00 00 30 39	00000000 00000000 00110000 00111001
y	-12345	CF C7	11001111 11000111
iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

0b 1100

Practice Question

For the following expressions, find a value of **`signed char`** `x`, if there exists one, that makes the expression `TRUE`. Compare with your neighbor(s)!

- Assume we are using 8-bit arithmetic:

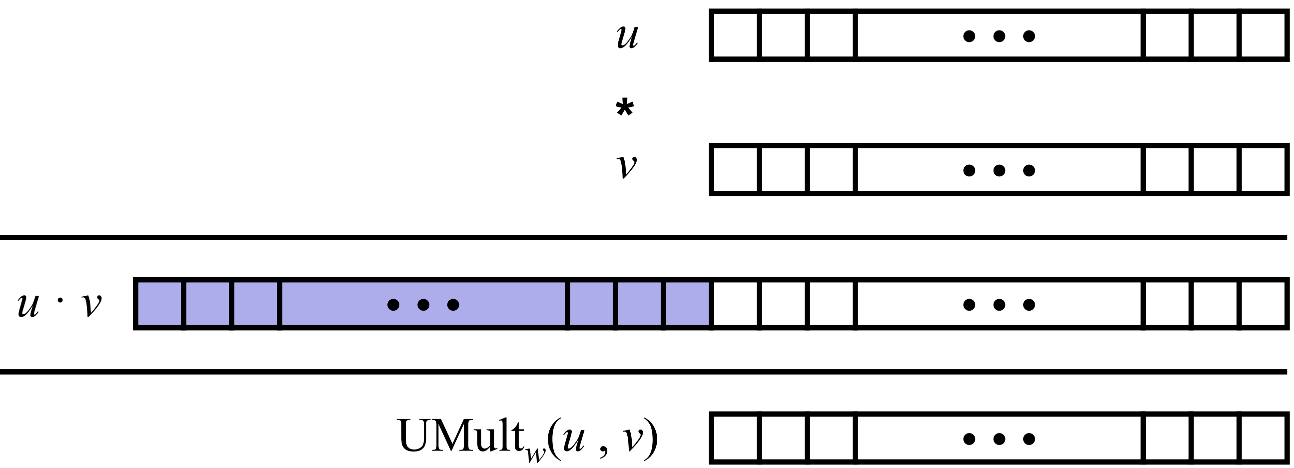
	Example:	All solutions:
`x == (unsigned char) x`		
`x >= 128U`		
`x != (x>>2)<<2`		
`x == -x` • Hint: there are two solutions		
`(x < 128U) && (x > 0x3F)`		

Aside: Unsigned Multiplication in C

Operands:
w bits

True Product:
$2w$ bits

Discard w bits:
w bits



- Standard Multiplication Function
  - Ignores high order w bits
- Implements Modular Arithmetic
  - $\text{UMult}_w(u , v) = u \cdot v \mod 2^w$

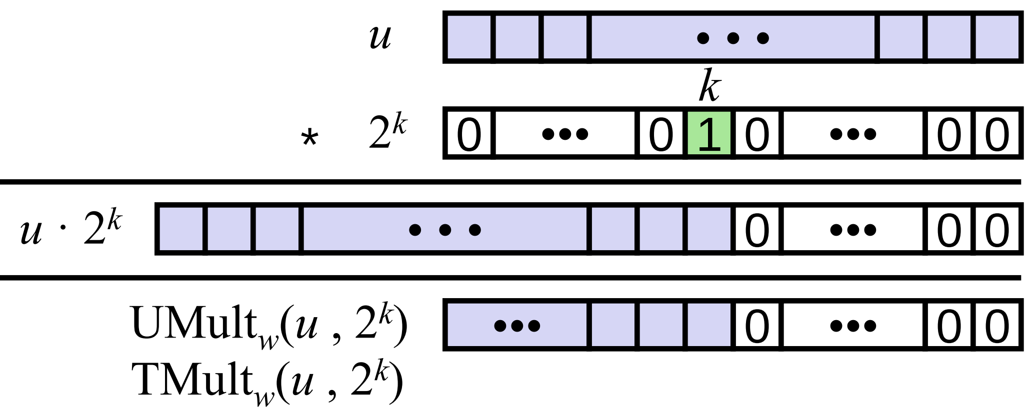
Aside: Multiplication with shift and add

- Operation `u<<k` gives `u*`2^k
  - Both signed and unsigned

Operands: ***w*** **bits**

True Product: ***w*** **+** ***k*** **bits**

Discard k bits: ***w*** **bits**



- Examples:
  - `u<<3           ==   u * 8`
  - `u<<5 - u<<3    ==   u * 24`
  - Most machines shift and add faster than multiply
    - ***Compiler generates this code automatically***

Number Representation Revisited

- What can we represent so far?
  - Signed and Unsigned Integers
  - Characters (ASCII)
  - Addresses

- How do we encode the following:
  - Real numbers (*e.g.* 3.14159)
  - Very large numbers (*e.g.* 6.02×10^{23})
  - Very small numbers (*e.g.* 6.626×10^{-34})
  - Special numbers (*e.g.* ∞, NaN)

Floating Point

Floating Point Topics

- **Fractional binary numbers**
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won’t cover
  - It’s a 58-page standard…

Representation of Fractions

- "Binary Point," like decimal point, signifies boundary between integer and fractional parts:

  Example 6-bit representation: **xx.yyyy**

  2^{1} 2^{0} 2^{-1} 2^{-2} 2^{-3} 2^{-4}

- <u>Example</u>: $10.1010_2 = 1 \times 2^{1} + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{10}$

Representation of Fractions

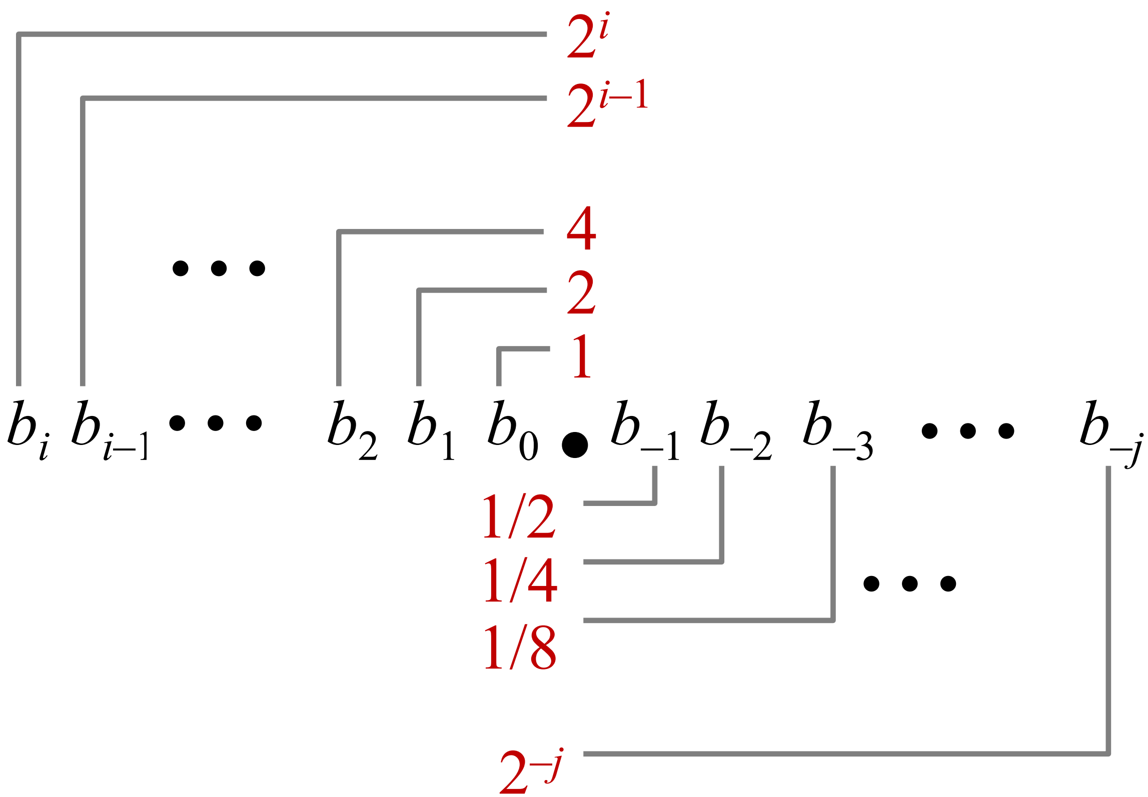
- "Binary Point," like decimal point, signifies boundary between integer and fractional parts:

Example 6-bit representation: **xx.yyyy**

2^{1} 2^{0} 2^{-1} 2^{-2} 2^{-3} 2^{-4}

- In this 6-bit representation:
  - What is the encoding and value of the smallest (most negative) number?
  - What is the encoding and value of the largest (most positive) number?
  - What is the smallest number greater than 2 that we can represent?

Fractional Binary Numbers



- **Representation**
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $$\sum_{k=-j}^{i} b_k \cdot 2^k$$

Fractional Binary Numbers

- Value Representation
  - 5 and 3/4 `101.11`$_2$
  - 2 and 7/8 `10.111`$_2$
  - 47/64 `0.101111`$_2$
- Observations
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form `0.111111`…$_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Limits of Representation

- Limitations:
  - Even given an arbitrary number of bits, can only **exactly** represent numbers of the form $x * 2^y$ (y can be negative)
  - Other rational numbers have repeating bit representations

Value:	Binary Representation:
1/3 = $0.333333\ldots_{10}$ =	$0.01010101[01]\ldots_2$
1/5 =	$0.001100110011[0011\ ]\ldots_2$
1/10 =	$0.0001100110011[0011\ ]\ldots_2$

Fixed Point Representation

- Implied binary point. Two example schemes:

  #1: the binary point is between bits 2 and 3

  $b_7\ b_6\ b_5\ b_4\ b_3\ [.]\ b_2\ b_1\ b_0$

  #2: the binary point is between bits 4 and 5

  $b_7\ b_6\ b_5\ [.]\ b_4\ b_3\ b_2\ b_1\ b_0$
- Wherever we put the binary point, with fixed point representations there is a trade off between the amount of range and precision we have
- Fixed point = fixed *range* and fixed *precision*
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers
- Hard to pick how much you need of each!

Floating Point Representation

- Analogous to scientific notation
  - In Decimal:
    - Not 12000000, but 1.2×10^{7} In C: 1.2e7
    - Not 0.0000012, but 1.2×10^{-6} In C: 1.2e-6
  - In Binary:
    - Not 11000.000, but 1.1×2^{4}
    - Not 0.000101, but 1.01×2^{-4}
- We have to divvy up the bits we have (e.g., 32) among:
  - the sign (1 bit)
  - the mantissa (significand)
  - the exponent

Scientific Notation (Decimal)

$$\overbrace{6.02}^{\text{mantissa}}{}_{10} \times 10^{23}$$

exponent

decimal point

radix (base)

- *Normalized form*: exactly one digit (non-zero) to left of decimal point
- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: $0.1\times10^{-8}, 10.0\times10^{-10}$

Scientific Notation (Binary)

$$\overbrace{1.01}^{\text{mantissa}}{}_2 \times 2^{-1}$$

exponent

binary point

radix (base)

- Computer arithmetic that supports this called floating point due to the "floating" of the binary point
  - Declare such variable in C as `float` (or `double`)

Scientific Notation Translation

$2^{-1} = 0.5$
$2^{-2} = 0.25$
$2^{-3} = 0.125$
$2^{-4} = 0.0625$

- Convert from scientific notation to binary point
  - Perform the multiplication by shifting the decimal until the exponent disappears
    - Example: $1.011_2 \times 2^4 = 10110_2 = 22_{10}$
    - Example: $1.011_2 \times 2^{-2} = 0.01011_2 = 0.34375_{10}$
- Convert from binary point to *normalized* scientific notation
  - Distribute out exponents until binary point is to the right of a single digit
    - Example: $1101.001_2 = 1.101001_2 \times 2^3$
- **Practice:** Convert 11.375_{10} to normalized binary scientific notation

Summary

- Sign and unsigned variables in C
  - Bit pattern remains the same, just *interpreted* differently
  - Strange things can happen with our arithmetic when we convert/cast between sign and unsigned numbers
    - Type of variables affects behavior of operators (shifting, comparison)
- We can only represent so many numbers in *w* bits
  - When we exceed the limits, *arithmetic overflow* occurs
  - *Sign extension* tries to preserve value when expanding
- Floating point approximates real numbers
  - We will discuss more details on Monday!

BONUS SLIDES

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

- 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 `1`s 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);`