Integers II

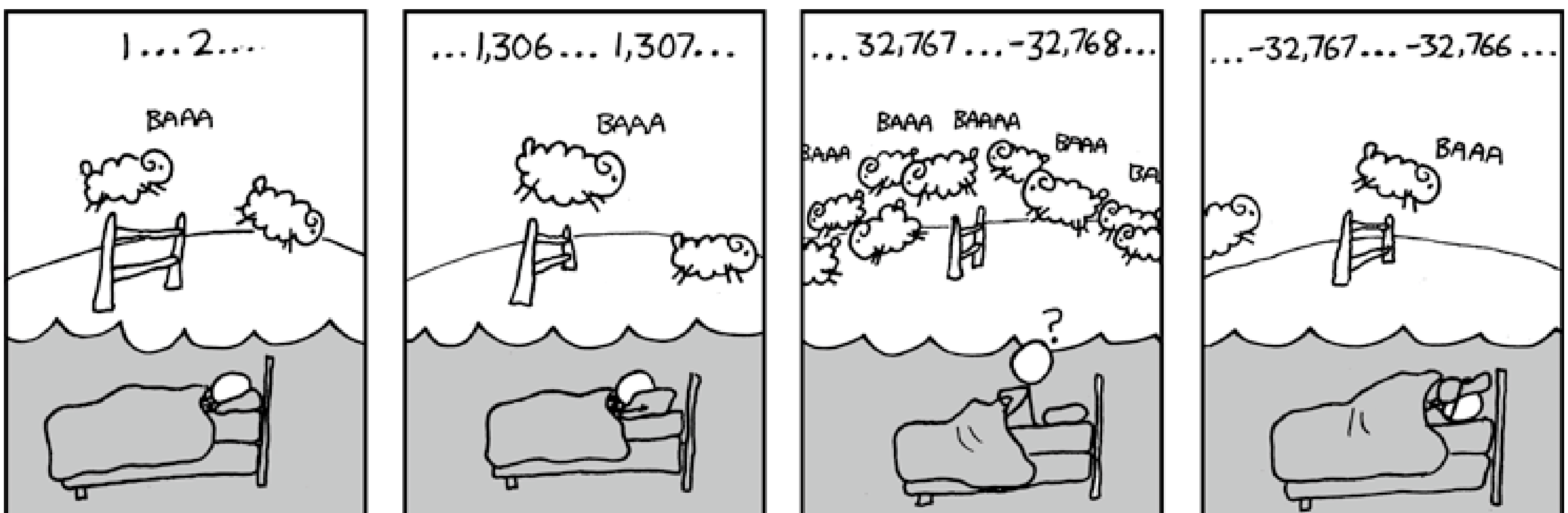
CSE 351 Autumn 2022

Instructor:

Justin Hsia

Teaching Assistants:

Angela Xu, Arjun Narendra, Armin Magness, Assaf Vayner, Carrie Hu, Clare Edmonds, David Dai, Dominick Ta, Effie Zheng, James Froelich, Jenny Peng, Kristina Lansang, Paul Stevans, Renee Ruan, Vincent Xiao



http://xkcd.com/571/

Relevant Course Information

- hw4 due Monday, hw5 due Wednesday
- Lab 1a due Monday (10/10)
  - 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 10/17
  - Bit manipulation on a custom encoding scheme
  - Bonus slides at the end of today's lecture have relevant examples

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

Reading Review

- Terminology:
  - `UMin`, `UMax`, `TMin`, `TMax`
  - Type casting: implicit vs. explicit
  - Integer extension: zero extension vs. sign extension
  - Modular arithmetic and arithmetic overflow
  - Bit shifting: left shift, logical right shift, arithmetic right shift

- Questions from the Reading?

Review Questions

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

  represent $2^6 = 64$ numbers; signed; most negative

  $-2^{n-1} = -2^5 = -32$

  0b 1 0 0 0 0 0 (-2^5, 2^4, 2^3, 2^2, 2^1, 2^0)

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

  2 bytes

  unsigned → zero extension: 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 = 0xE8

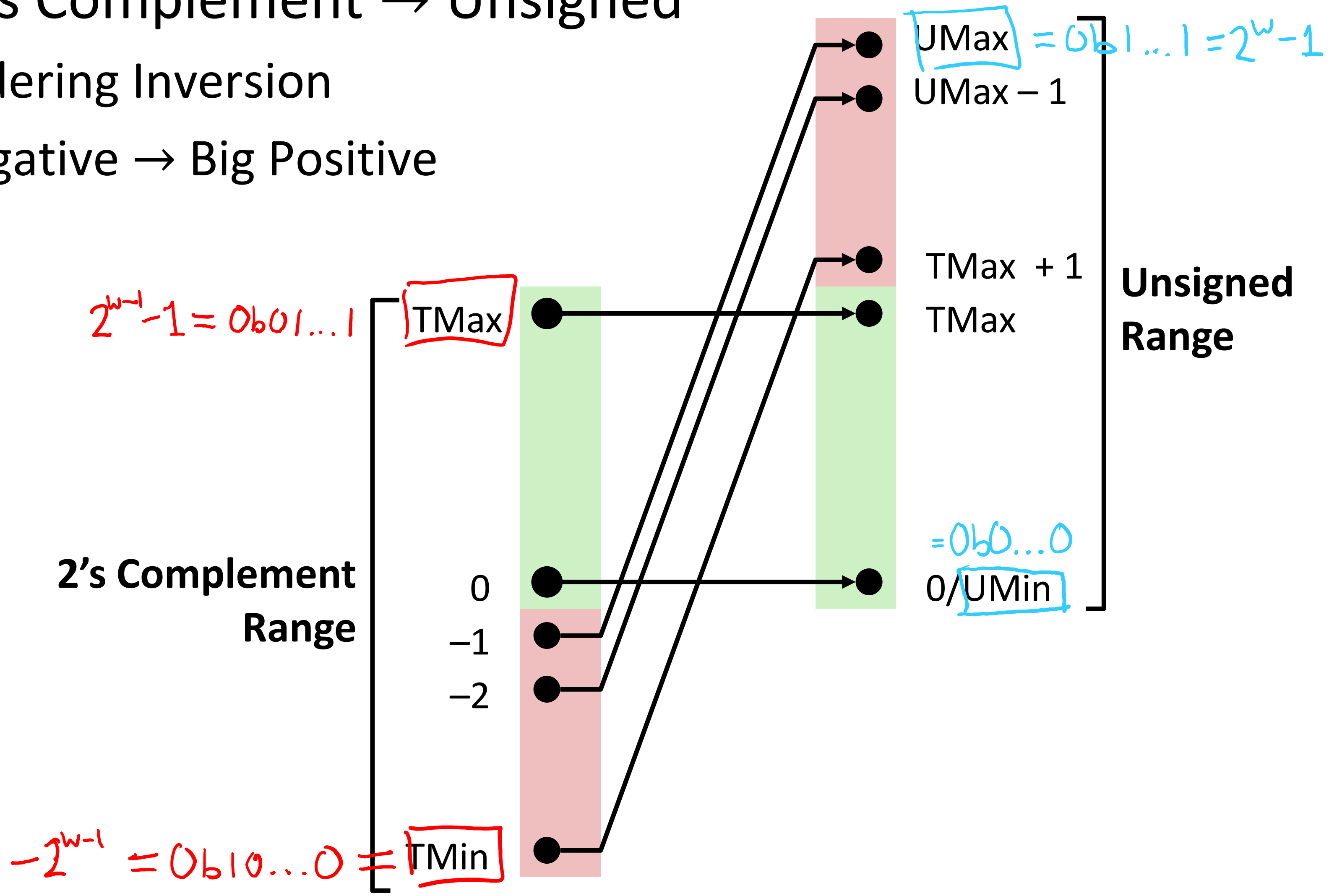
  unsigned: 0b 1010 0001 —logical→ 0b 0001 0100 = 0x14

Integers

- **Binary representation of integers**
  - **Unsigned and signed**
  - **Casting in C**
- Consequences of finite width representations
  - Sign extension, overflow
- Shifting and arithmetic operations

Signed/Unsigned Conversion Visualized

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



Values To Remember (Review)

- 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

In C: Signed vs. Unsigned (Review)

- Casting
  - Bits are unchanged, just interpreted differently!
    - **int** tx, ty;
    - **unsigned int** ux, uy;
  - *Explicit* casting
    - tx = (**int**) ux;
    - uy = (**unsigned int**) ty;

(new_type) expression

  - *Implicit* casting can occur during assignments or function calls

cast to target variable/parameter type

    - tx = ux;
    - uy = ty;

(also implicitly occurs with printf format specifiers)

Casting Surprises (Review)

!!!

- 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 <u>unsigned</u>**
  - Including comparison operators <, >, ==, <=, >=

(unsigned "dominates")

Expression Evaluation Examples

- Assuming 8-bit data (*i.e.*, bit position 7 is the MSB), what will the following expression evaluate to?
  - signed `127` < unsigned `128u`

    0b0111 1111   0b1000 0000

    unsigned comparison: 0b0111 1111 (+127) < 0b1000 0000 (+128) → True

  - signed `127 < (signed char) 128u` (signed)

    0b0111 1111   0b1000 0000

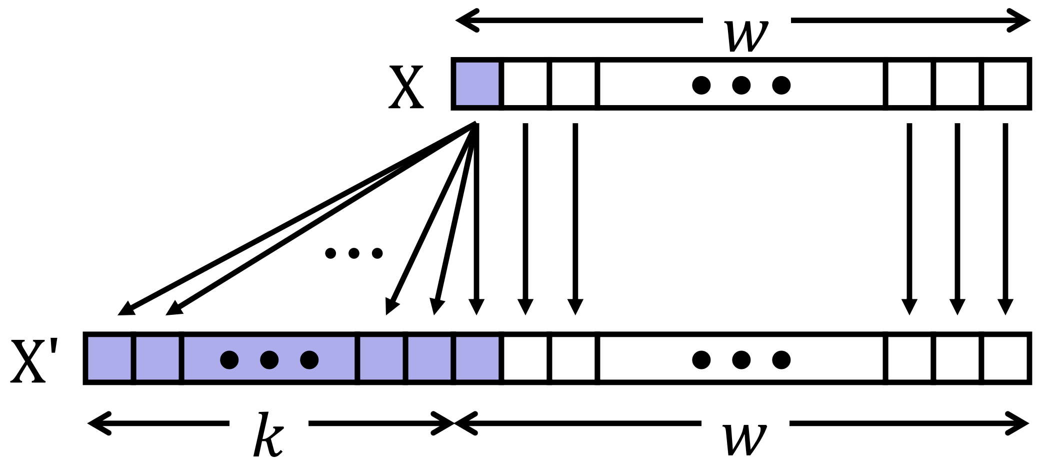
    signed comparison: 0b0111 1111 (+127) < 0b1000 0000 (−128) → False

Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
- **Consequences of finite width representations**
  - **Sign extension, overflow**
- Shifting and arithmetic operations

Sign Extension (Review)

- **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
  - $\mathrm{X}' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, \underbrace{x_{w-1}, x_{w-2}, \ldots, x_1, x_0}_{\text{original } \mathrm{X}}$



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

Arithmetic Overflow (Review)

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

- When a calculation produces a result that can't be represented in the current encoding scheme
  - Integer range limited by fixed width (UMin – UMax, TMin – TMax)
  - 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!

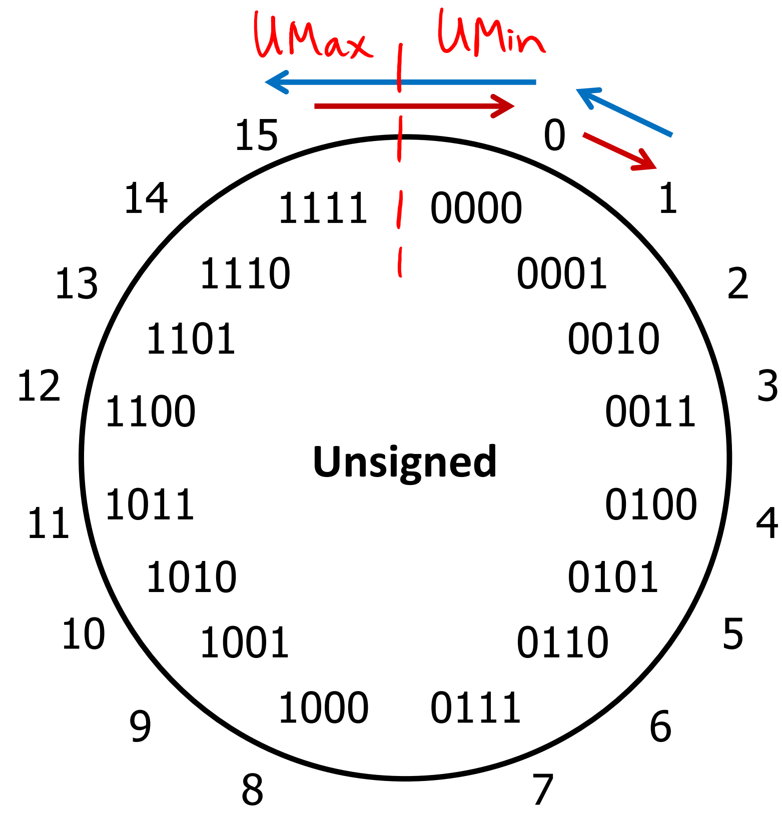
Overflow: Unsigned

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

```
   15        1111
 +  2     +  0010
 ----     -------
   17       10001
    1
```

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

```
    1       10001
 -  2     -  0010
 ----     -------
   -1        1111
   15
```



$\pm 2^N$ because of modular arithmetic

$2^4 = 16$

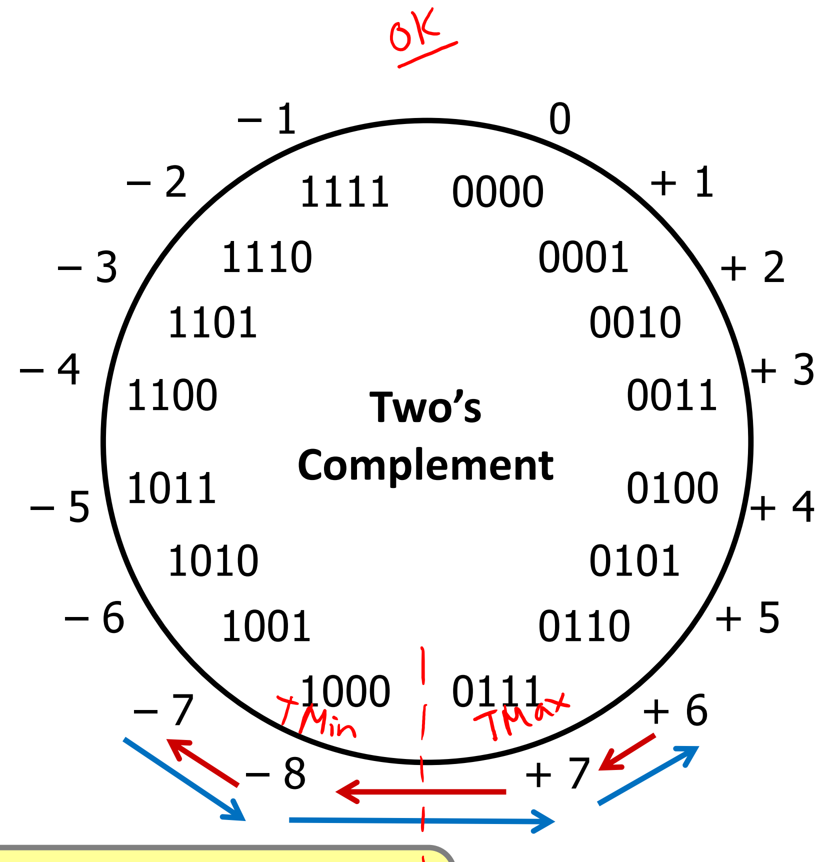
Overflow: Two's Complement

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

```
   6        0110
+  3     +  0011
----     -------
   9        1001
  −7
```

- **Subtraction:** (−) + (−) = (+)?

```
 −7         1001
−  3     −  0011
----     -------
−10         0110
   6
```



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

Practice Questions

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

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

- 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

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
- Consequences of finite width representations
  - Sign extension, overflow
- **Shifting and arithmetic operations**

Shift Operations (Review)

- 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)

8-bit example:

	x	0010 0010
	x<<3	0001 0000
logical:	x>>2	0000 1000
arithmetic:	x>>2	0000 1000

	x	1010 0010
	x<<3	0001 0000
logical:	x>>2	0010 1000
arithmetic:	x>>2	1110 1000

Shift Operations (Review)

digit $d_i \times 2^i$ changes power of 2 by n because it moved positions

- 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! (compiler will try to optimize for you)

- Notes:
  - Shifts by n<0 or n≥w (w is bit width of x) are *undefined* (behavior not guaranteed)
  - **In C:** behavior of >> is determined by the compiler
    - In gcc / C lang, depends on data type of x (signed/unsigned) (arithmetic/logical)
  - **In Java:** logical shift is >>> and arithmetic shift is >>

Left Shifting Arithmetic 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;`	~~00~~01100100	=	100	100
`L2=x<<3;`	~~000~~11001000	=	-56	200
`L3=x<<4;`	~~0001~~10010000	=	-112	144

200 − 256 → 2^8

400 − 256 → 2^8

signed overflow

unsigned overflow

Right Shifting Arithmetic 8-bit Examples

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

```
xu  = 240u;  11110000       =  240   /8 = 30
R1u=xu>>3;   00011110000    =   30   /4 = 7.5
R2u=xu>>5;   0000011110000  =    7
```

rounding (down)

Right Shifting Arithmetic 8-bit Examples

- **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=xu>>3;    11111110000       =  -2
R2s=xu>>5;    1111111110000     =  -1
```

-2 /4 = -0.5

rounding (down)

Exploration Questions

UMin = 0, UMax = 255
8-bits, so TMin = -128, TMax = 127

For the following expressions, find a value of `signed char x`, if there exists one, that makes the expression True.

- Assume we are using 8-bit arithmetic:
  - `x == (unsigned char) x` unsigned
    - Example: x = 0
    - All solutions: works for all x
  - `x >= 128U` unsigned 0b1000 0000
    - Example: x = -1
    - All solutions: any $x < 0$
  - `x != (x>>2)<<2`
    - Example: x = 3
    - All solutions: any x where lowest two bits are not 0b00
  - `x == -x`
    - Hint: there are two solutions
    - Example: x = 0
    - All solutions: ① x = 0b0...0 = 0 ② x = 0b10...0 = -128
  - `(x < 128U) && (x > 0x3F)`
    - All solutions: any x where upper two bits are exactly 0b01

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
- Shifting is a useful bitwise operator
  - Right shifting can be arithmetic (sign) or logical (0)
  - Can be used in multiplication with constant or bit masking

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 `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);`