Integers II

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang

Announcements, Reminders

- Due Today:
  - HW 3 (11:59pm)
- Due Monday, 7/1
  - RD 6 (1pm)
  - HW 4 (11:59pm)
- Lab 1b releases today, due 7/10
  - Bit manipulation on a custom encoding scheme
  - Bonus slides at the end might be helpful :)

Review Questions

- What is the value and encoding of **`Tmin`** (minimum *signed* value) for a fictional 7-bit wide integer data type? encoding = 1000000
  value = $-2^6 = \boxed{-64}$

- For `unsigned char uc = 0xB3;`, what the result (in hex) of the cast `(unsigned short)uc`? in unsigned, pad extra space with 0s → $\boxed{0x00B3}$
  1B
  2B

- What is the result of the following expressions? 0xB3 = 0b 1011 0011
  - `(signed char)uc >> 2` signed = pad w/ most-significant bit → 0b 11101100 $= \boxed{0xEC}$
  - `(unsigned char)uc >> 3` unsigned = pad w/ 0 → 0b 00010110 $= \boxed{0x16}$

Integers

- **Binary representation of integers**
    - **Unsigned and signed**
    - **Casting in C**
    - **Arithmetic operations**
- Consequences of finite width representations
    - Overflow
- Shifting operations

Values to Remember

Unsigned

- **UMin** = 0
  - 0b00…00
- **UMax** = 2^w-1
  - 0b11…11

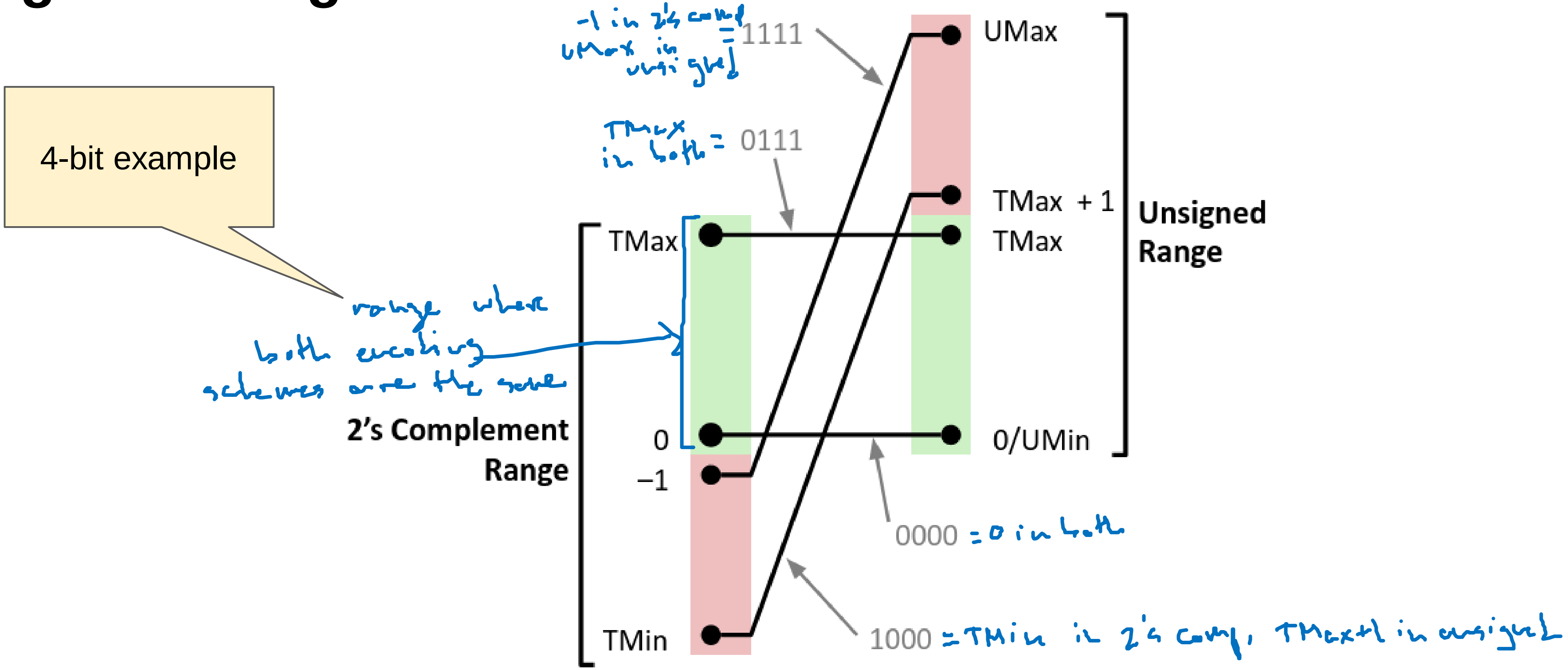
Signed (2's Complement)

- **TMin** = -2^{w-1}
  - 0b10…00
- **TMax** = $2^{w-1} - 1$
  - 0b01…11

Example: if w = 64

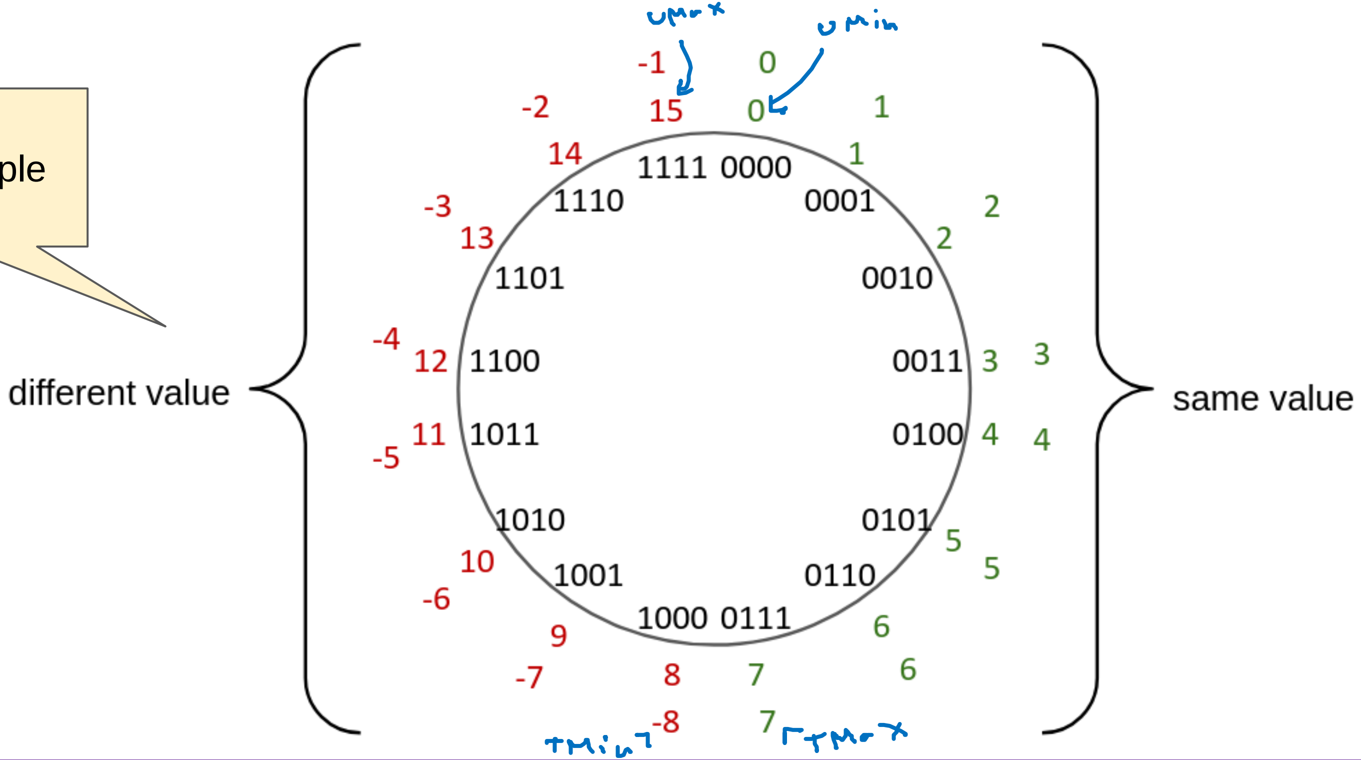
	Hex	Decimal
UMax	FF FF FF FF FF FF FF FF	18,446,744,073,709,551,615
TMax	7F FF FF FF FF FF FF FF	9,223,372,036,854,775,807
UMin	00 00 00 00 00 00 00 00	0
TMin	80 00 00 00 00 00 00 00	-9,223,372,036,854,775,808

Signed/Unsigned Conversion Visualized



Signed/Unsigned Conversion Visualized (pt 2)

4-bit example



C Integer Casting (Review)

- Bits are unchanged, just *interpreted* differently
  - Ex:

```c
int tx, ty;
unsigned int ux, uy;
```

- **Explicit** casting:
  - Ex:

```c
tx = (int)ux;
uy = (unsigned int)ty;
```

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

```c
tx = ux;
uy = ty;
```

Common source of bugs!

Casting Surprises (Review)

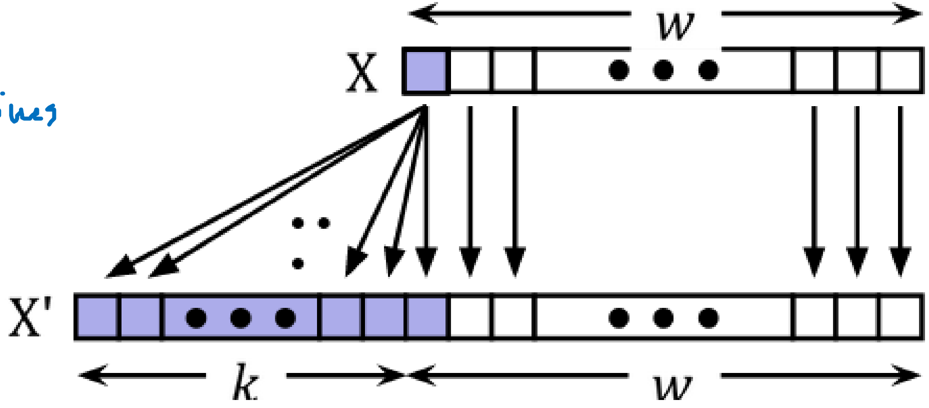
- Integer literals (constants)
  - By default, treated as *signed* ints
  - Hex constants already have an explicit binary representation
  - Use “U” (or “u”) suffix to explicitly force *unsigned*
  - Ex: 4294967259u
- Expression Evaluation
  - When you mixed unsigned and signed in a single expression, then signed values are implicitly cast to unsigned
  - Including comparison operators <, >, ==, <=, >=
  - Yeah, no idea why. Thanks, C…

Sign Extension (Review)

- Given a *w*-bit integer, how can we extend it to a (*w*+*k*)-bit integer while keeping the value the same?
  - Unsigned - pad with 0s
    - Ex: 0b1000 = 0b00001000 = 8
  - Signed - pad with the **most significant bit**
    - Ex: 0b1000 = 0b11111000 = -8

Fun fact: can duplicate MSb any # of times in 2's comp!

ex: 1000 = -8
11000 = -16+8 = -8
111000 = -32+16+8 = -8



Two's Complement Arithmetic

- Same as unsigned!
  - Simplifies hardware, no special algorithm needed
  - Just add as normal, then discard the highest carry bit
    - **Modular addition**: result = sum modulo 2^w

Example:

carry

```
                                 1111
   0011      =     3              1101      =     -3
  +0001      =     1             +1111      =     -1
  -----                          -----
   0100      =     4             11100      =     -4
```

Why Does Two's Complement Work?

- For all representable numbers *x*, we theoretically want *additive inverse*:
  - i.e. (bit representation of *x*) + (bit representation of *-x*) = 0
- What are the 8-bit negative encodings for the following?

```
  00000001        00000010        11000011
+ ????????      + ????????      + ????????
  --------        --------        --------
  00000000        00000000        00000000
```

Why Does Two's Complement Work? (pt 2)

- For all representable numbers *x*, we theoretically want *additive inverse*:
  - i.e. (bit representation of *x*) + (bit representation of -*x*) = 0
- What are the 8-bit negative encodings for the following?

dropped off   carry

```
1  1111111
   00000001
+  11111111
   00000000
```

```
   00000010
+  11111110
   00000000
```

```
   11000011
+  00111101
   00000000
```

These are the bitwise complement plus 1!

`-x == ~x + 1`

Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
  - Arithmetic operations
- **Consequences of finite width representations**
  - **Overflow**
- Shifting operations

Arithmetic Overflow (Review)

- What happens if a calculation produces a result that *can't* be represented in the current encoding scheme? **Overflow!**
  - Remember: fixed width integers can't represent every possible number
  - Occurs in both signed and unsigned
  - Can occur in both positive *and* negative directions
- Both C and Java ignore overflow exceptions
  - You end up with a bad value in your program and no indication/warning

Overflow: Unsigned

Occurs when result is *less than* both operands for addition, or *greater than* for subtraction

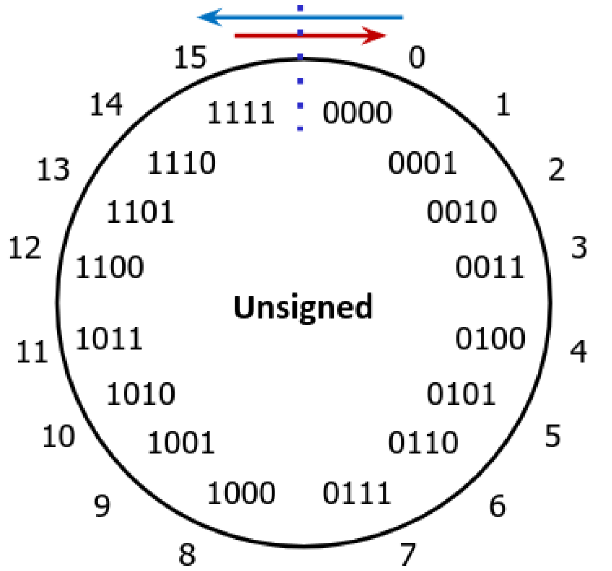
- Addition: drop carry bit (result is 2^w too small)

$$\begin{array}{rcl} 1111 & = & 15 \\ +\ 0001 & = & 1 \\ \hline \cancel{1}0000 & = & \cancel{16}\ 0 \end{array}$$

- Subtraction: "borrow" extra bit (result is 2^w too large)

Note: no actual bit to borrow from in HW, just theoretical

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



Overflow: Signed

Occurs when both operands for an *addition* have the same sign, and result doesn't match

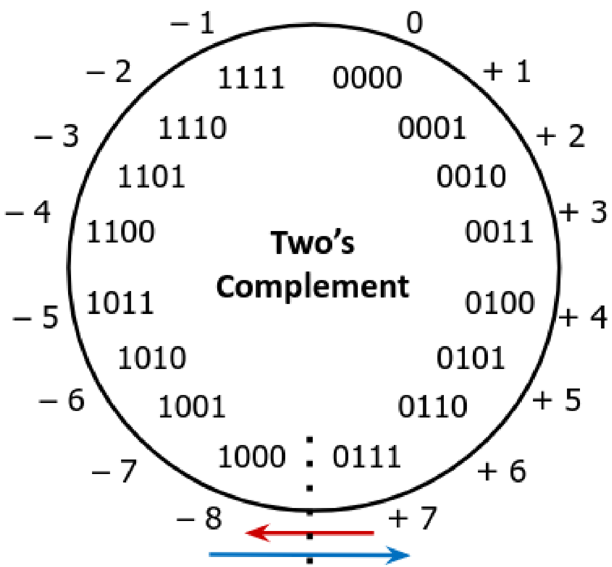
- Positive addition: (+) + (+) = (-)

$$\begin{array}{rcr} 0110 & = & 6 \\ +\ 0011 & = & 3 \\ \hline 1001 & = & -7??? \end{array}$$

- Negative addition (i.e. subtraction): (-) + (-) = (+)

same as
-7 + -3
1001
+ 1101
0110 (carry bit dropped off)

$$\begin{array}{rcr} 1001 & = & -7 \\ -\ 0011 & = & 3 \\ \hline 0110 & = & 6??? \end{array}$$



Why does this matter?

adds errors together

If there are 256 errors, overflows back to 0!

- **1985:** Therac-25 radiation therapy machine
  - Overdoses of radiation due to arithmetic overflow on 1-byte safety flag
- **2000:** Y2K problem
  - Limited representation (2-digit decimal year)
  - Similar issue will occur with Unix time in 2038!

stored as # of seconds since 1/1/1970 in a signed int. will overflow in 2038

- **2013:** Deep impact spacecraft lost
  - Suspected integer overflow from storing time as tenth-seconds in unsigned int
    - Lost on 8/11/13, 00:38:49.6

00:38:49.5 is the last representable time

Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
  - Arithmetic operations
- Consequences of finite width representations
  - Overflow
- **Shifting operations**

Shift Operations (Review)

- Move all bits left or right, extra bits “fall off” the end
- Left shift by *n* positions (`x << n`)
  - Lose the most-significant *n* bits, fill in the least-significant *n* bits with 0s
- Right shift by *n* positions (`x >> n`)
  - Lose the least-significant *n* bits
  - Unsigned, use **logical**: fill with most-significant *n* bits with 0s
  - Signed, use **arithmetic**: replicate the previous most-significant bit

Ex: 0x22

`x`	`0010 0010`
`x << 3`	`0001 0000`
(logical) `x >> 2`	`0000 1000`
(arithmetic) `x>> 2`	`0000 1000`

Ex: 0xA2

`x`	`1010 0010`
`x << 3`	`0001 0000`
(logical) `x >> 2`	`0010 1000`
(arithmetic) `x>> 2`	`1110 1000`

Shift Operations (Review) (pt 2)

in base 10, multiply / divide by 10^n
ex: $3 \ll 1 = 30$, $3 \ll 2 = 300$, etc.

- Arithmetic
  - Left shift (x << n) == multiply by 2^n
  - Right shift (x >> n) == divide by 2^n
    - For signed values, logical right shift preserves the sign
  - **Fun fact:** Shifting is often *faster* than the general multiply and divide operations!
- Notes:
  - Shifts by less than 0 or more than *w* (width of the variable) are **undefined**
    - i.e. we don't know what will happen!
  - In Java, arithmetic shift is >>, logical is >>>

Left Shifting, 8-bit Example

- Shifting can cause overflow!
- In theory `x << n` should be $x*2^n$

Signed overflow

Code	Binary	Signed	Unsigned	Theoretical Value
`x = 25`	`00011001`	25	25	25
`L1 = x << 2`	`00 01100100`	100	100	100
`L2 = x << 3`	`000 11001000`	-56	200	200
`L3 = x << 4`	`0001 10010000`	-112	114	400

Unsigned overflow

Right Shifting, 8-bit Example

- Unsigned = logical shift
- In theory, `x >> n` should be $x \div 2^n$

Code	Binary	Unsigned	Theoretical Value
`x = 240u`	`11110000`	240	240
`R1 = x >> 3`	`00011110 000`	30	30
`R2 = x >> 5`	`00000111 10000`	7	7.5?

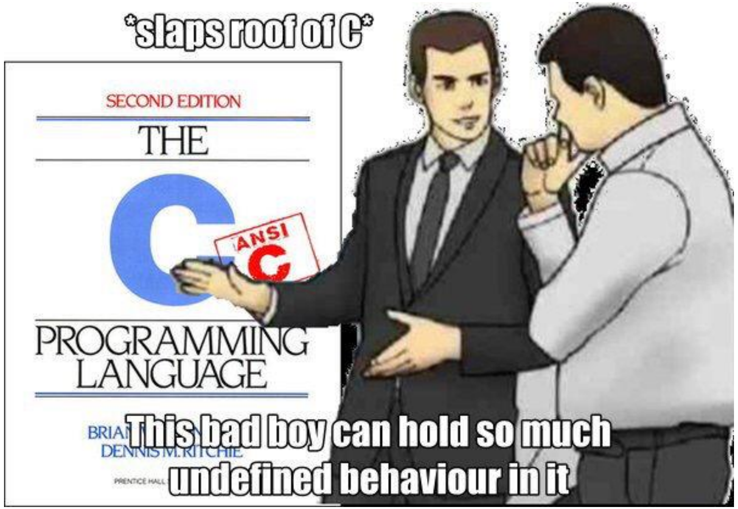
Right Shifting, 8-bit Example (pt 2)

- Signed = arithmetic shift
- In theory, `x >> n` should be $x \div 2^n$

Code	Binary	Unsigned	Theoretical Value
`x = -16`	`11110000`	-16	-16
`R1 = x >> 3`	`11111110 000`	-2	-2
`R2 = x >> 5`	`11111111 10000`	-1	-0.5?

Undefined Behavior in C

- Not defined in C standard, may get different behavior depending on your OS, architecture, compiler, etc.
- How much **undefined behavior** have we talked about in just the last few lectures?
  - Shifting by more than size of type
  - Indexing arrays out of bounds
  - Using a variable before initializing (mystery data)
  - … and there will be more!



C Language

- Development began in 1971, standardized in 1978
  - Developed to write Unix (precursor to Linux and MacOS)
- Computers were much more limited in the 70s!
- Computer *users* were also very different!
  - Not as accessible
  - Computers were “for experts”
- Goals:
  - Portability
  - Performance
- <u>*Non-Goals*</u>:
  - Safety
  - Ease

Summary

- Casting between signed and unsigned in C
  - <u>Bit pattern remains the same, just interpreted differently</u>
  - Cast can be **explicit** or **implicit**
- We can represent a limited number of values in *w* bits
  - When we exceed the limit (in either direction), we get **overflow**
- **Shifting** is a useful bitwise behavior
  - Can be used to remove certain bits (similar to masking), or in place of multiplication
  - Right shift can be **logical** or **arithmetic**
    - Logical pads with 0s, used for unsigned
    - Arithmetic pads with MSB, used for signed

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

Practice Question 1

- Assuming 8-bit data (*i.e.*, bit position 7 is the MSB), what will the following expression evaluate to?
    - UMin = 0, UMax = 255, TMin = -128, TMax = 127

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

0b10000000 = -128 in 2's comp

127 < -128 is False !

Practice Questions 2

- For the following additions, did signed and/or unsigned overflow occur?
  - `0x27 + 0x81` = 39 - 127 = -88, or 39u + 129u = 168u
  - `0x7F + 0xD9` = 127 - 39 = 88, or 127u + 217u = 344u
- Helpful values (assuming 8-bit integers):
  - **`0x27`** = 39 (signed) = 39 (unsigned)
  - **`0xD9`** = -39 (signed) = 217 (unsigned)
  - **`0x7F`** = 127 (signed) = 127 (unsigned)
  - **`0x81`** = -127 (signed) = 129 (unsigned)

carry

```
                   1 1 1
0x27 + 0x81 = 0b0011 0111
            + 0b1000 0001
              ---------
              1011 1000
```

no unsigned bc no dropped bit

no signed bc we're adding values w/ different signs

carry

```
              1 1 1 1 1 1 1 1
0x7F + 0xD9 = 0b 0111 1111
            + 0b 1101 1001
              -----------
               1 0101 1000
```

yes unsigned bc extra 1 is dropped

no signed bc we're adding values w/ different signs

Exploration Questions

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 integers:

other solutions also possible

  - `x == (unsigned char) x` — x=-1: (unsigned char)x = 255
  - `x >= 128U` — x=-1: when mixing types, defaults to unsigned
  - `x != (x>>2)<<2` — x=1: x>>2 = 0, (x>>2)<<2 = 0
  - `x == -x` — x=-128: x=0b10000000, -x = ~x+1 = 0b01111111+1 = 0b10000000
    - Hint: there are two solutions

0 also works!

  - `(x < 128U) && (x > 0x3F)`

= 3·16+15 = 63

when mixing signs, defaults to unsigned, so anything between 64 and 127 will work

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 & 0xFF)>>16	00000000 00000000 00000000 00000010

Using Shifts and Masks (pt 2)

- 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 (pt 3)

- 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;}` is the same as…
    - `a=(((!!x<<31)>>31)&y) | (((!x<<31)>>31)&z);`