

# Integers II

CSE 351 Winter 2018

## Instructor:

Mark Wyse

## Teaching Assistants:

Kevin Bi Parker, DeWilde, Emily Furst,

Sarah House, Waylon Huang, Vinny Palaniappan



FUN FACT: DECADES FROM NOW, WITH SCHOOL A DISTANT MEMORY, YOU'LL STILL BE HAVING THIS DREAM.

# Administrivia

- ❖ Lab 1 due next Friday (1/19)
  - Prelim submission (3+ of `bits.c`) due on Monday (1/15)
  - Bonus slides at the end of today's lecture have relevant examples
- ❖ HW 2 will be release on Monday 1/15
  - Due Wednesday, 1/24
- ❖ No class on Monday 1/15

# Integers

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

# Two's Complement Negatives

❖ Accomplished with one neat mathematical trick!

$b_{w-1}$  has weight  $-2^{w-1}$ , other bits have usual weights  $+2^i$



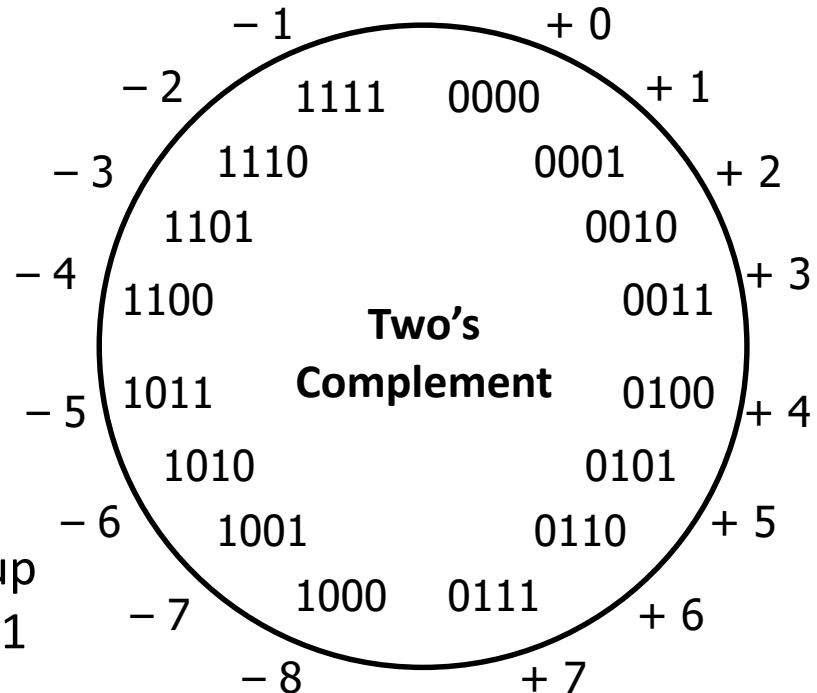
■ 4-bit Examples:

- $1010_2$  unsigned:  
 $1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10$
- $1010_2$  two's complement:  
 $-1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = -6$

■ -1 represented as:

$$1111_2 = -2^3 + (2^3 - 1)$$

- MSB makes it super negative, add up all the other bits to get back up to -1



# 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 Example:

-4    1100	4    0100
+3    +0011	-3    +1101
= -1	= 1

# Why Does Two's Complement Work?

- ❖ For all representable positive integers  $x$ , we want:

$$\frac{\begin{array}{l} \textit{bit representation of } x \\ + \textit{ bit representation of } -x \end{array}}{\quad} 0 \quad (\text{ignoring the carry-out bit})$$

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

$$\begin{array}{r} 00000001 \\ + \text{????????} \\ \hline 00000000 \end{array}$$

$$\begin{array}{r} 00000010 \\ + \text{????????} \\ \hline 00000000 \end{array}$$

$$\begin{array}{r} 11000011 \\ + \text{????????} \\ \hline 00000000 \end{array}$$

# Why Does Two's Complement Work?

- ❖ For all representable positive integers  $x$ , we want:

$$\frac{\text{bit representation of } x \\ + \text{bit representation of } -x}{0} \quad (\text{ignoring the carry-out bit})$$

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

$$\begin{array}{r} 00000001 \\ + 11111111 \\ \hline 100000000 \end{array}$$

$$\begin{array}{r} 00000010 \\ + 11111110 \\ \hline 100000000 \end{array}$$

$$\begin{array}{r} 11000011 \\ + 00111101 \\ \hline 100000000 \end{array}$$

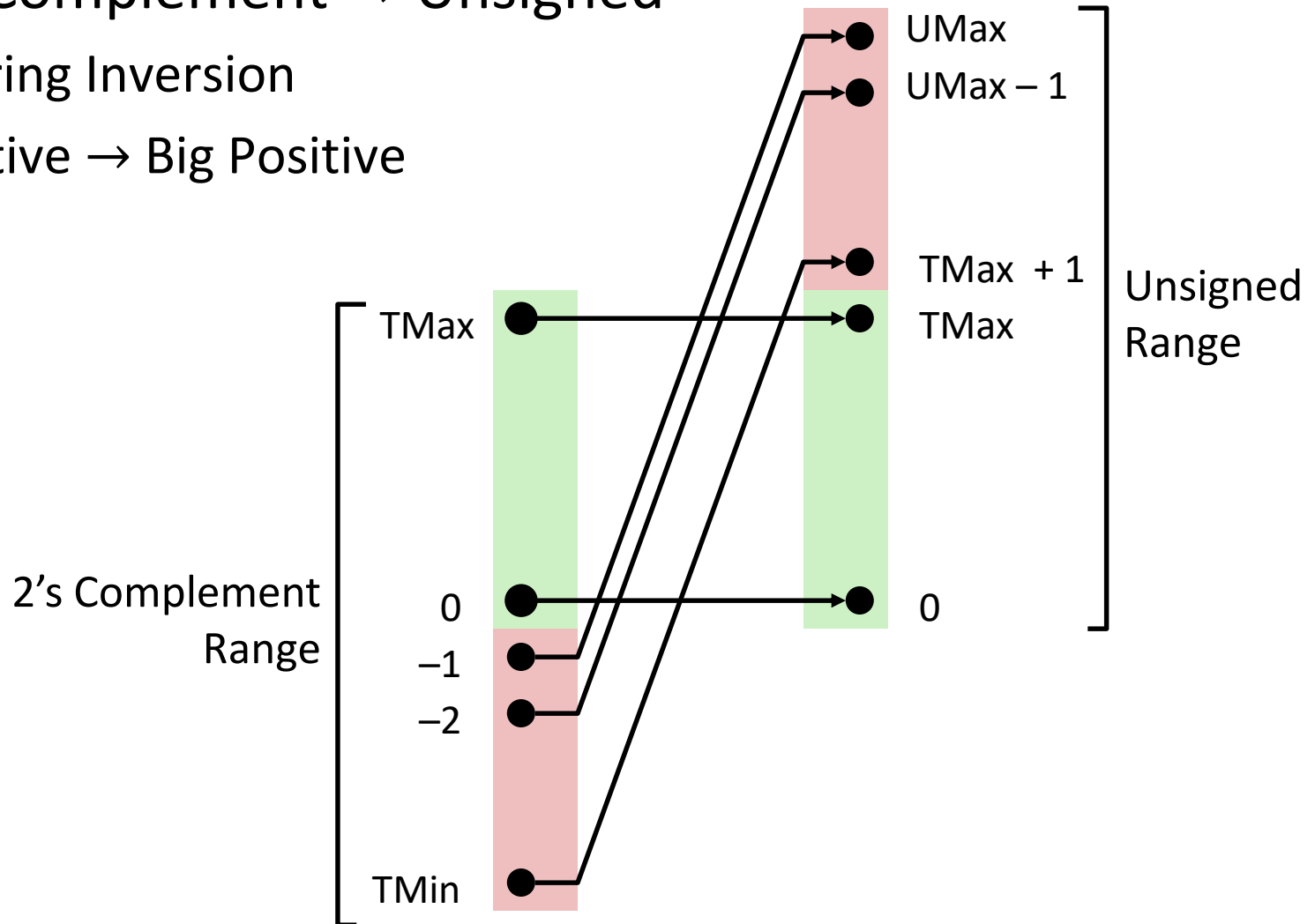
These are the bitwise complement plus 1!

$$-x == \sim x + 1$$

# Signed/Unsigned Conversion Visualized

## ❖ Two's Complement $\rightarrow$ Unsigned

- Ordering Inversion
- Negative  $\rightarrow$  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

# 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**
  - Including comparison operators `<`, `>`, `==`, `<=`, `>=`



# Casting Surprises

## ❖ 32-bit examples:

- TMin = -2,147,483,648, TMax = 2,147,483,647

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

# Integers

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

# 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!

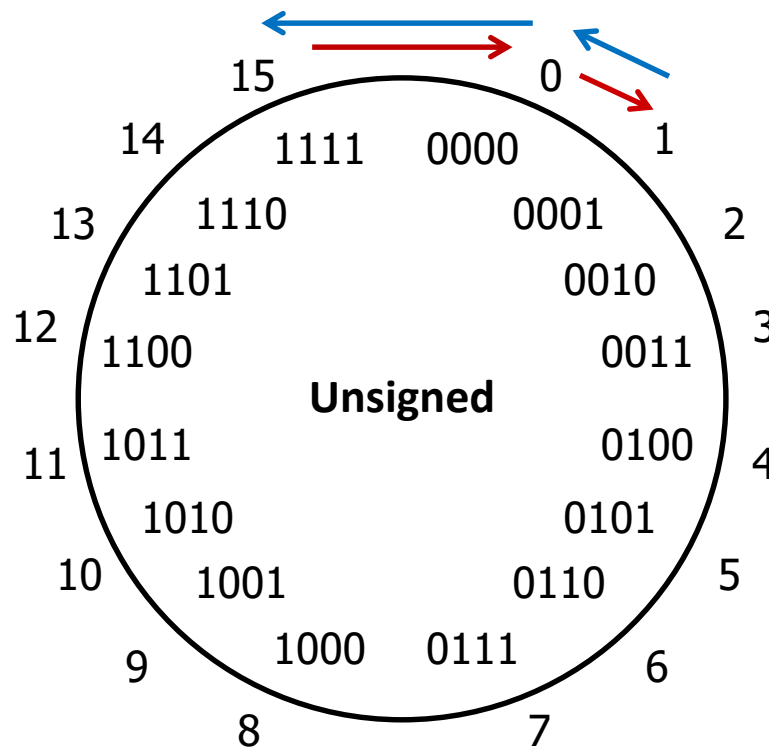
# Overflow: Unsigned

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

15	1111
+ 2	+ 0010
<del>17</del>	<del>10001</del>
1	

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

1	10001
- 2	- 0010
<del>-1</del>	1111
15	



±2<sup>N</sup> because of modular arithmetic

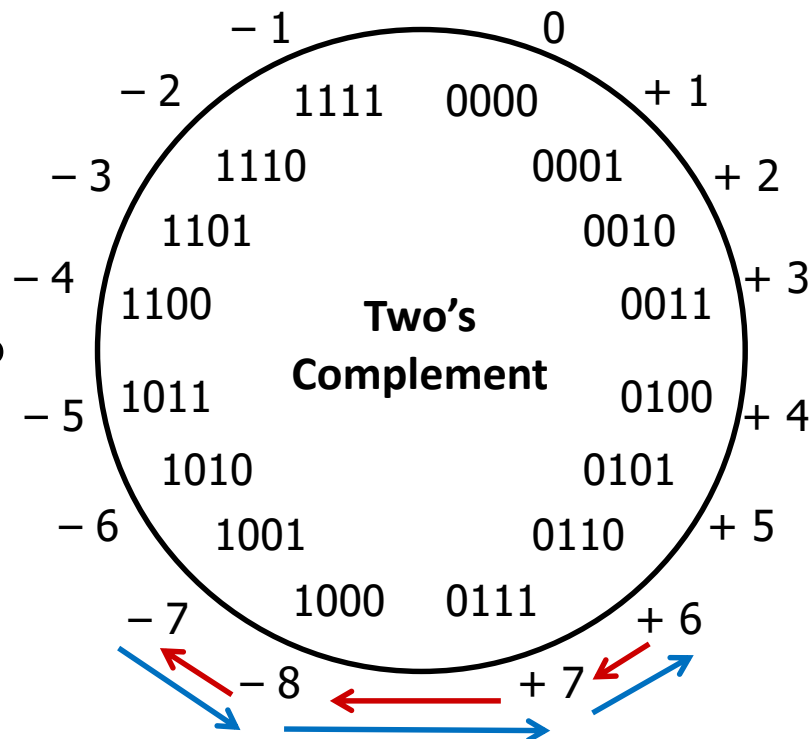
# Overflow: Two's Complement

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

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

❖ **Subtraction:** (-) + (-) = (+)?

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



**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`

- ❖ **4-bit → 8-bit Example:**

- Positive Case

- ✓ • Add 0's?

**4-bit:**            0010 = +2

**8-bit:**    00000010 = +2

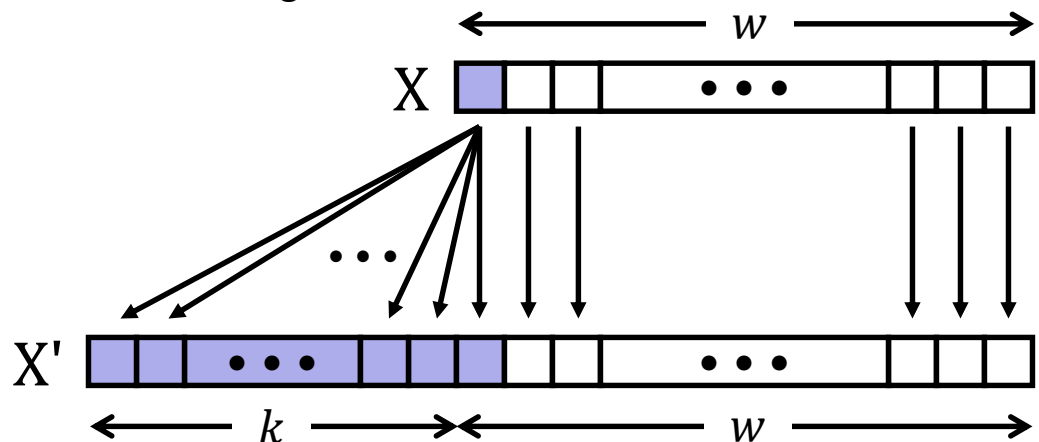
# Sign Extension

❖ **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

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



# Peer Instruction Question

- ❖ Which of the following 8-bit numbers has the same *signed* value as the 4-bit number **0b1100**?
  - Underlined digit = MSB
  
- A. 0b 0000 1100
- B. 0b 1000 1100
- C. 0b 1111 1100
- D. 0b 1100 1100
- E. We're lost...

# 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;

```

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

# Integers

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

# Shift Operations

- ❖ Left shift ( $x \ll n$ ) bit vector  $x$  by  $n$  positions
  - Throw away (drop) extra bits on left
  - Fill with 0s on right
- ❖ Right shift ( $x \gg n$ ) bit-vector  $x$  by  $n$  positions
  - Throw away (drop) extra bits on right
  - Logical shift (for **unsigned** values)
    - Fill with 0s on left
  - Arithmetic shift (for **signed** values)
    - Replicate most significant bit on left
    - Maintains sign of  $x$

# Shift Operations

❖ Left shift ( $x \ll n$ )

- Fill with 0s on right

❖ Right shift ( $x \gg n$ )

- Logical shift (for **unsigned** values)

- Fill with 0s on left

- Arithmetic shift (for **signed** values)

- Replicate most significant bit on left

❖ Notes:

- Shifts by  $n < 0$  or  $n \geq w$  (bit width of  $x$ ) are *undefined*
- **In C:** behavior of  $\gg$  is determined by compiler
  - In gcc / C lang, depends on data type of  $x$  (signed/unsigned)
- **In Java:** logical shift is  $\ggg$  and arithmetic shift is  $\gg$

$x$	0010 0010
$x \ll 3$	0001 0 <b>000</b>
logical: $x \gg 2$	<b>00</b> 00 1000
arithmetic: $x \gg 2$	<b>00</b> 00 1000

$x$	1010 0010
$x \ll 3$	0001 0 <b>000</b>
logical: $x \gg 2$	<b>00</b> 10 1000
arithmetic: $x \gg 2$	<b>11</b> 10 1000

# Shifting Arithmetic?

## ❖ What are the following computing?

### ■ $x \gg n$

- $0b\ 0100 \gg 1 = 0b\ 0010$
- $0b\ 0100 \gg 2 = 0b\ 0001$
- Divide by  $2^n$

### ■ $x \ll n$

- $0b\ 0001 \ll 1 = 0b\ 0010$
- $0b\ 0001 \ll 2 = 0b\ 0100$
- Multiply by  $2^n$

## ❖ Shifting is faster than general multiply and divide operations



# 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
<code>x = 25;</code>	00011001 =	25	25
<code>L1=x&lt;&lt;2;</code>	0001100100 =	100	100
<code>L2=x&lt;&lt;3;</code>	00011001000 =	-56	200
<code>L3=x&lt;&lt;4;</code>	000110010000 =	-112	144

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

`R1u=xu>>3;`    `00011110000`    = 30

`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

rounding (down)

# Peer Instruction 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:

■  $x == (\text{unsigned char}) x$

■  $x \geq 128U$

■  $x \neq (x \gg 2) \ll 2$

■  $x == -x$

• Hint: there are two solutions

■  $(x < 128U) \ \&\& \ (x > 0x3F)$

# 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 1.

We will try to cover these in lecture or section if we have the time.

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

# Using Shifts and Masks

- ❖ Extract the 2<sup>nd</sup> most significant *byte* of an `int`:
  - First shift, then mask:  $(x \gg 16) \ \& \ 0xFF$

<b>x</b>	00000001	00000010	00000011	00000100
<b>x &gt;&gt; 16</b>	00000000	00000000	00000001	00000010
<b>0xFF</b>	00000000	00000000	00000000	11111111
<b>(x &gt;&gt; 16) &amp; 0xFF</b>	00000000	00000000	00000000	00000010

- Or first mask, then shift:  $(x \ \& \ 0xFF0000) \gg 16$

<b>x</b>	00000001	00000010	00000011	00000100
<b>0xFF0000</b>	00000000	11111111	00000000	00000000
<b>x &amp; 0xFF0000</b>	00000000	00000010	00000000	00000000
<b>(x &amp; 0xFF0000) &gt;&gt; 16</b>	00000000	00000000	00000000	00000010

# Using Shifts and Masks

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

<b>x</b>	00000001 00000010 00000011 00000100
<b>x&gt;&gt;31</b>	00000000 00000000 00000000 00000000
<b>0x1</b>	00000000 00000000 00000000 00000001
<b>(x&gt;&gt;31) &amp; 0x1</b>	00000000 00000000 00000000 00000000

<b>x</b>	10000001 00000010 00000011 00000100
<b>x&gt;&gt;31</b>	11111111 11111111 11111111 11111111
<b>0x1</b>	00000000 00000000 00000000 00000001
<b>(x&gt;&gt;31) &amp; 0x1</b>	00000000 00000000 00000000 00000001



# Using Shifts and Masks

## ❖ Conditionals as Boolean expressions

- For `int x`, what does `(x<<31)>>31` do?

<code>x=!!123</code>	00000000 00000000 00000000 00000000 <b>1</b>
<code>x&lt;&lt;31</code>	<b>1</b> 00000000 00000000 00000000 00000000
<code>(x&lt;&lt;31)&gt;&gt;31</code>	<b>11111111 11111111 11111111 11111111</b>
<code>!x</code>	00000000 00000000 00000000 00000000 <b>0</b>
<code>!x&lt;&lt;31</code>	<b>0</b> 00000000 00000000 00000000 00000000
<code>(!x&lt;&lt;31)&gt;&gt;31</code>	<b>00000000 00000000 00000000 00000000</b>

- 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;`